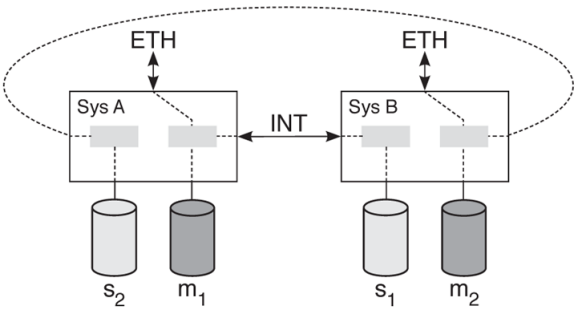
Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik und Kommunikationsnetze (TIK) Prof. Dr. L. Thiele

Network Attached Storage

SA Sommersemester 2004
(SA-2004-31)



9. Juli 2004

David H. Gehring
Beat Schilbach

Betreuer
Martin Frey (SCS AG)
Matthias Dyer (TIK)

Professor
Prof. Dr. Lothar Thiele

Inhaltsverzeichnis

Vorwort I
Einleitung III
Aufgabenstellung V

I Einführung in das Thema **1**

1 Begriffserklärung und Terminologie **3**
1.1 In diesem Dokument verwendete Begriffe 3
1.2 Namenskonvention 4

2 Einführung in das Themengebiet **5**
2.1 DAS, NAS und SAN 5
2.2 Die RAID Technologie 6
2.3 Über FPGAs 8

3 Die Problematik **9**
3.1 Aktuelle Situation 9
3.2 Problemdefinition 10
3.3 Zieldefinition 10

II Systemdesign **13**

4 Lösungserarbeitung **15**
4.1 Anwendungsbereich und Lokalisation 16
4.1.1 Anwendungsbereich des Systems 16
4.1.2 Lokalisation im Netzwerk 16
4.2 Varianten und Alternativen 17
4.2.1 RAID und ähnliche Lösungsansätze 17

4.2.2 Die Grundidee . . . 18
4.2.3 Skalierbarkeit der Systeme . . . 19
4.2.4 Harddisk-Anbindung . . . 20
4.2.5 Verwendung von RAM . . . 22
4.2.6 Software . . . 23
4.3 Zahlen und Abschätzungen . . . 24

5 Systemdesign **27**
5.1 Beschreibung der Komponenten/Funktionsblöcke . . . 27
5.2 Error-Handling . . . 29
5.2.1 Mögliche Fehler . . . 29
5.2.2 Systemstatus . . . 30
5.2.3 Reaktionen bei Fehlern . . . 31
5.3 Der Reconstruction-Vorgang . . . 33
5.3.1 Error Recovery . . . 33
5.3.2 Reconstruction-Algorithmen . . . 34
5.4 Systeminitialisierung und Systemreset . . . 35

6 Detailbeschreibungen der entwickelten Funktionsblöcke **37**
6.1 Einführung . . . 37
6.2 Host Interface . . . 38
6.3 Master Kontroller . . . 39
6.3.1 Systemstatus . . . 40
6.3.2 Definierte Zustände im Controller . . . 40
6.3.3 Intern verwendete Register . . . 42
6.4 Aufbau des FIFO . . . 44
6.4.1 Internes FIFO des Virtex II Pro . . . 44
6.4.2 Eigene Implemenation . . . 45
6.5 Der RAID0 Kontroller . . . 47
6.5.1 Error-Handling . . . 47
6.5.2 Striping-Algorithmus . . . 48
6.5.3 Definierte Zustände im Controller . . . 49
6.5.4 Intern verwendete Register . . . 51

III Anhang **55**

7 Fazit **57**

8 VHDL Code **59**

8.1 Main File 59
8.2 Host Interface (Dummy) 65
8.3 FSM_Master 68
8.4 FIFO 76
8.5 RAID0 77
8.6 Link (Dummy) 85
8.7 SATA (Dummy) 87

IV Verzeichnisse **91**

Abbildungsverzeichnis 93
Tabellenverzeichnis 95
Literaturverzeichnis 97
Index 99

Vorwort

Wir bedanken uns bei der Firma Supercomputing Systems AG (SCS AG), welche uns die Durchführung dieser Semesterarbeit bei ihr im Technopark ermöglicht hat. Herzlicher Dank im Speziellen gilt Herrn Martin Frey, Herrn Leo Breuss und Herrn Markus Herrli, welche uns mit konstruktiven Gesprächen und technischer Hilfe tatkräftig unterstützt haben.

Im Weiteren bedanken wir uns bei Herrn Dr. Prof. Lothar Thiele, dass wir diese Arbeit am Institut für Technische Informatik und Kommunikationsnetze (TIK) unseres Departements Informationstechnologie und Elektrotechnik (D-ITET) der ETH Zürich machen durften. Ebenso gilt unser Dank Herrn Matthias Dyer, welcher die Betreuung und Organisation seitens der ETH bereitwillig übernommen hat.

Diese Semesterarbeit war für uns sehr interessant und lehrreich. Sie hat uns einen grossen Einblick in die Unternehmenswelt ermöglicht. Mit ihr beenden wir unsere Vorlesungszeit an der ETH und führen die angefangene Arbeit in unserer Masterarbeit fort. In dieser ist es das Ziel, eine auf die Problemstellung zugeschnittene Hardware zu produzieren und einen funktionstüchtigen Prototypen zu entwickeln.

Zürich, den 9. Juli 2004

David H. Gehring

Beat Schilbach

II

Einleitung

Einfache NAS Systeme sind mit der Planung unsere Semesterarbeit stark im Aufwind. Um uns aber von den einfachen Lösungen (eine Festplatte ohne Datensicherheit) abzugrenzen, ist unser Ziel, ein System mit Daten- und Kontrollersicherheit zu entwickeln. Solche Systeme bestehen bereits, jedoch nur in Verbindung von grossen Speicher- und Serversystemen. Mit dieser Arbeit wollen wir die Kluft zwischen diesen Low-End- beziehungsweise High-End-Systemen überwinden und aufzeigen, dass die Möglichkeit besteht, ein kleines, hochverfügbares und billiges System zu entwickeln. Dazu setzen wir auf eine Ein-Chip-Lösung und Standardkomponenten, welche jedoch die Leistungsanforderungen erfüllen.

Mit dieser Semesterarbeit geht es erst einmal um das Systemdesign, die Implementation von wichtigen Komponenten und um das Austesten des Prototypen auf einem bestehenden Entwicklungsboard. Als Entwicklungsplattform steht uns eine FPGA der Firma Xilinx zur Verfügung, welcher einen eingebetteten Prozessor hat und somit die Möglichkeit bietet, ein ≪System on a Chip≫ (SoC) zu betreiben.

Warum schreiben wir diese Arbeit?

Der Computer ist in unserer Zeit zu einem festen Bestandteil der täglichen Arbeit und der Freizeit geworden. Durch den regelmässigen Umgang mit der Computertechnik entstehen immer wieder neue Wünsche und Bedürfnisse. Hinzu kommt der Fortschritt der Technik. Die Technologien für Speichermedien werden laufend verbessert. Die Geschwindigkeit und die Speichermasse nehmen zu; die physikalische Grösse sowie der Energieverbrauch nehmen ab. Diese Neuerungen bieten eine gute Grundlage für Neuentwicklungen.
Wir haben festgestellt, dass in den KMU-Betrieben (Klein- und Mittlere Unternehmen) ein grosser Bedarf an Netzwerk-Speicher besteht. Ein ausfallsicheres, schnelles und immer verfügbares Speichersystem stellt ein grosses Bedürfnis dar. Entweder sind die Preise auf dem Markt für bestehende Angebote zu hoch oder diese Angebote bieten nicht den benötigten Funktionsumfang. Dieser Hintergrund motiviert uns, ein eigenes System zu entwickeln, welches die benötigte Funktionalität anbietet und in der unteren Preisregion liegt.

Technischer Hintergrund

Nebst den genannten Gründen hat auch eine Vorlesung[1] über rekonfigurierbare Speicher-Arrays[2] unser Interesse an der Materie geweckt. Die Möglichkeiten, die diese Technologie bietet, sind ziemlich weit gestreut und faszinierend. Ein überzeugender Punkt ist der, dass man eine Entwicklung rasch testen und so auch verbessern kann. Im Vergleich zu einem eigens für eine Applikation entwickelten Hardware-Baustein, dessen Konfiguration nicht mehr geändert werden kann, ein wichtiges und zeitsparendes Kriterium.
Diese Vorteile veranlassen uns, unser System vollständig in Hardware zu realisieren. Das Ziel ist ein System mit einer minimalen Anzahl von Komponenten. Ein SoC (System on a Chip) ist die von uns angestrebte Lösung.

[1] Platzner: *Reconfigurable Computing*, Wintersemester 2003/2004, ETHZ
[2] FPGA – Field Programmable Gate Array

Aufgabenstellung

Ziel dieser Arbeit ist es, die Grundlagen eines replizierten, hoch verfügbaren Network Attached Storage (NAS) Systems zu erarbeiten. Als Plattform soll der PowerPC im FPGA von XILINX (Virtex II Pro) eingesetzt werden. In dieser Semesterarbeit werden die notwendigen Grundlagen erarbeitet, welche insbesondere die Kenntnisse über RAID1 Algorithmen und Disk Interfaces enthalten.

Einleitung

Diskkapazität wird immer günstiger, die Kosten für das Storage-Management hingegen nicht. Die erzielten Diskdurchsätze wachsen weniger schnell als Diskkapazität und Rechenleistung der Prozessoren. Diesen beiden Trends wird mit verschiedenen Ansätzen entgegengesteuert:

- Network Attached Storage (NAS): durch die Einführung von Storage Appliances, die Storage direkt ins Netzwerk einbinden, können tiefere Storage-Managementkosten erzielt werden.
- Intelligente Disks: ein Teil der Diskverwaltung wird vom Server in die Disk ausgelagert. Das Object Storage Model überlässt so zum Beispiel dem Diskkontroller die Diskblockallokation. Der Server wird dadurch entlastet.
- Storage Virtualisierung: Ressourcen in einem Storage Area Network (SAN) können flexibel verschiedenen Volumen zugeordnet werden. Umlagern von Daten, Snapshoting, Spiegelung werden im SAN transparent für die Server ausgeführt.

XILINX bietet heute FPGAs in der Virtex II Pro Linie an, die einen PowerPC Prozessor enthalten. Dadurch wird es möglich, innerhalb eines Chips einen kompletten Software Stack bis und mit Filesystemexport (NFS, CIFS usw.) zu implementieren. Für die Virtex II Pro FPGAs existiert so z.B. ein Linux Port.

Diese Arbeit soll klären, ob sich obige Trends mittels einer kostengünstigen Lösung, basierend auf Virtex II Pro, durchbrechen lassen.

Teilaufgaben

1. Erstellen Sie in den ersten zwei Wochen zusammen mit Ihrem Betreuer einen realistischen Zeitplan, welcher Meilensteine festlegt. Überlegen Sie sich, wie Sie die Arbeit effizient aufteilen können.
2. Machen Sie sich mit dem XILINX Evaluation Board, der Linux Umgebung und der VHDL Toolchain vertraut.
3. Machen Sie sich mit dem ATA und SATA Interface vertraut. Entscheiden Sie in Absprache mit dem Betreuer, ob ein ATA oder SATA Interface implementiert werden soll.
4. Evaluieren Sie Algorithmen zur Implementation eines RAID1 Kontrollers.

5. Erstellen Sie ein Systemdesign für ein NAS mit Fast Ethernet Interface, (S)ATA Disk Interfaces und RAID1 Kontroller. Der RAID1 Kontroller soll in einer späteren Arbeit erweitert werden können, damit er die Daten zu einem anderen NAS System spiegeln kann. Die RAID1 Funktionalität soll dabei in Hardware implementiert werden.
6. Reviewen Sie den Systemdesign mit Ihrem Betreuer.
7. Implementieren Sie ein (S)ATA Disk Interface im FPGA.
8. Implementieren Sie ein Host Interface im FPGA.
9. Implementieren Sie einen RAID1 Mechanismus, welcher zwischen Host Interface und (S)ATA Disk Interface eingefügt wird.
10. Testen Sie Ihre Implementation.
11. Dokumentieren Sie Ihre Arbeit sorgfältig mit einem Vortrag, einer kleinen Demonstration sowie mit einem Schlussbericht.

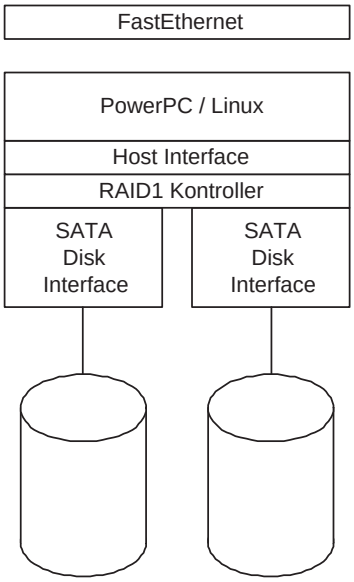


Abbildung 1: Aufbau einer möglichen Storage Appliance

Durchführung der Semesterarbeit

- Der Verlauf des Projekts Semesterarbeit soll laufend anhand des Projektplanes und der Meilensteine evaluiert werden. Unvorhergesehene Probleme beim eingeschlagenen Lösungsweg können Änderungen am Projektplan erforderlich machen. Diese sollen dokumentiert werden.
- Stellen Sie Ihr Projekt zu Beginn der Semesterarbeit in einem Kurzvortrag vor und präsentieren Sie die erarbeiteten Resultate am Schluss im Rahmen des Institutskolloquiums Ende Semester.
- Besprechen Sie Ihr Vorgehen regelmässig mit Ihren Betreuern. Verfassen Sie dazu auch einen kurzen wöchentlichen Statusbericht (EMail).

Abgabe

- Geben Sie zwei unterschriebene Exemplare des Berichts spätestens am *9. Juli 2004* dem betreuenden Assistenten oder seinem Stellvertreter ab. Diese Aufgabenstellung soll vorne im Bericht eingefügt werden.

Zürich, April 2004

Teil I

Einführung in das Thema

Kapitel 1

Begriffserklärung und Terminologie

1.1 In diesem Dokument verwendete Begriffe

Die nachfolgend erklärten Begriffe werden in diesem Dokument öfters verwendet. Auf die meisten Begriffe oder Standards wird dabei an passender Stelle detaillierter eingegangen. Damit sich der Leser mit den besprochenen Themen auseinandersetzen kann, sind hier die wichtigsten Begriffe kurz erläutert.

FIFO Ein FIFO (First In First Out) ist ein Prinzip für Warteschlangen. Die ersten der Warteschlange hinzugefügten Daten werden als erste Daten ausgelesen und gelöscht.

FPGA Ein FPGA (Field Programmable Gate Array) ist ein nach seiner Herstellung wiederprogrammierbarer Hardware-Baustein, der die Entwicklung und das rasche Testen von ≪Eingebetteten Systemen≫ mit hardware-basierten Teilen ermöglicht.

FSM Eine FSM (Finite State Machine) ist eine abstrakte Maschine, die eine konstante und endliche Menge von Zuständen hat. Eine FSM wird oft für Berechnungen verwendet.

NVRAM Das NVRAM (Non-volatile Random Access Memory) ist ein Memory, das batteriegestützt ist und somit keine Daten verliert. Es wird deshalb auch als *nicht flüchtig* bezeichnet.

RAID Ein RAID (Redundant Array of Independent Disks) besteht aus mehreren Harddisks, die dem Betriebssystem als eine einzige logische Speichereinheit erscheinen. Je nach Level bietet RAID bessere Performance und vor allem Datenredundanz.

Reconstruction Nach einem Fehler im System fehlen auf einer Festplatte im Normalfall Daten. Der Reconstruction-Prozess ist der für den Datenabgleich zuständige Vorgang.

Rocket I/O Diese äusserst schnelle Schnittstelle[1] wird in den FPGAs von Xilinx zur Verfügung gestellt. Der multi-gigabit Transceiver bietet die höchste Performance und die grösste Funktionalität für eingebettete Serializer/Deserializer (Daten-Links).

[1] Es ist eine Baud-Rate von bis zu 3,25 Gbps erreichbar.

SATA Serial-ATA[2] wurde für den Transfer von Daten zwischen dem Prozessor und der Harddisk entwickelt. SATA ist der weiterentwickelte und verbesserte Nachfolge-Standard von PATA (Parallel ATA).

1.2 Namenskonvention

Das Institut für Mikroelektronik am Departement ≪Informationstechnologie und Elektrotechnik (ITET)≫ der ETH Zürich bietet auf ihrer Homepage [5] ≪Microelectronics Design Center≫ Support für das Design von *Microelectronic Chips* (VLSI Chips und ASICs) und von *printed circuit boards* (PCBs). Sie geben Empfehlungen ab für die Bennenung von Signalen und Variablen. In dieser Arbeit richten sich die in VHDL-Code verwendeten Namen nach diesen vorgeschlagenen Konventionen.
Es werden Namen verwendet, die den Inhalt möglichst treffend beschreiben und dennoch nicht allzu lange sind. Die Namen geben den Zweck und nicht den Typ eines Objekts an. Spezielle Eigenschaften oder Typen werden mit einem Suffix verdeutlicht (Tabelle 1.1). Komponenten, Architekturen und Entities entsprechen in ihrer Namensgebung im Code den Dokumentnamen des VHDL-Files.

Namen für	**Konvention**	**Beispiel**
Konstanten	alles in Grossbuchstaben geschrieben	RAM_DEPTH
Signale	starten mit Grossbuchstaben, haben folgende Suffix	
	Abk. C: Clock	ClkxC
	Abk. R: Reset (*asynchroner Reset*)	RstxRB
	Abk. E: Enable (*synchrone Events*)	LoadCntxE
	Abk. S: Control/Status	FullxS
	Abk. D: Data/Address	RamAdrxD
	Abk. T: Test	RamIso1xT
Variabeln	starten mit Kleinbuchstaben, haben keinen Suffix	temp, currentState
Typen	Suffix *type*	stateType, stdLogicArray

Tabelle 1.1: Übersicht der Namenskonventionen für VHDL-Code

[2] ATA ist die Abkürzung für Advanced Technology Attachment [8].

Kapitel 2

Einführung in das Themengebiet

Diskkapazität wird immer günstiger, die Kosten für die Speicherverwaltung hingegen nicht. Die erzielten Diskdurchsätze wachsen weniger schnell als die Diskkapazität und die Rechenleistung der Prozessoren. Die verschiedenen Ansätze, die diese Trends angehen, werden nachfolgend erläutert. Zusätzlich werden weitere Technologien aufgezeigt, die diese Ansätze optimal ergänzen.

2.1 DAS, NAS und SAN

≪Direct Attached Storage≫ (DAS) ist die gängigste Variante, Speicher von einem Computer aus anzusprechen. Der Speicher befindet sich dabei in einem System, welches autonom und unabhängig von allen anderen Systemen agiert. Ein Beispiel dafür ist der Personal Computer.
≪Network Attached Storage≫ (NAS) bietet Benutzern (Clients) Speicherbereich in einem Netzwerk an. NAS wurde entwickelt, um die Probleme mit DAS, welche die Administration und die Wartung wie auch die ungenügende Skalierbarkeit und Systemverfügbarkeit umfassen, zu umgehen. Dabei ist ein eigenes File System[1] in das Speicher-Subsystem eingebunden. NAS Systeme ermöglichen einen einfachen Gebrauch und bieten verschiedenen Betriebssystemen Daten-Austausch an. Ebenfalls sind sie darauf spezialisert, möglichst viele Aspekte des File Systems in die Speichereinheit einzubetten. Im Weiteren automatisieren und vereinfachen sie das Daten-Management einer Firma. [1]
≪Storage Attached Network≫ (SAN) ist ein spezielles Speichernetzwerk, welches *Hard-Disk-Arrays* und *Tape-Silos* umfasst. Sie sind der beste Weg, sehr grosse Datenmengen dynamisch den Verbrauchern (Servers) zuzuordnen. Auch setzen SAN Systeme auf optische Links mit eigenem Protokoll, wobei NAS Systeme auf Kupfertechnologie mit TCP/IP setzen.
Die gestiegene Forderung nach übers Netz erreichbarem Speicherplatz bestimmt Nachfrage und Preis. Investitionen für Datenzentren werden heute hauptsächlich in Speicheranschaffung getätigt. Sowohl NAS als auch SAN haben ihre Vorteile bzw. Nachteile (Tabelle 2.1, Seite 6).

[1] Das File System verwaltet die Dokumente, welche im Speicher abgelegt sind.

Vorteile	**Nachteile**
Einfaches Speichermanagement	Wachsende Komplexität
Optimierter Umgang mit Speicherressourcen	Korrektes Funktionieren vieler Maschinen
Vergrössert Datenerreichbarkeit (Availability)	Teureres Netzwerk-Protokoll
Data-Sharing zwischen Clients	Datenintegrität und Privacy angreifbarer
Backup in ≪not busy≫ Phasen	

Tabelle 2.1: Vorteile und Nachteile von Network Attached Storage

2.2 Die RAID Technologie

Ursprünglich stand das Akronym RAID für ≪Redundant Array of Inexpensive Disks≫ [2]. Heutzutage ist es bekannt als ≪Redundant Array of Independant Disks≫. Drei Professoren der Universität von Kalifornien, David Petterson, Garth Gibson und Randy Katz, haben das Konzept von RAID zum ersten Mal beschrieben.
Unter einem RAID versteht man mehrere Harddisks, die dem Host Computer als eine einzige logische Speichereinheit erscheinen [3]. Die Technologie ermöglicht eine Vergrösserung von Diskkapazität, während zugleich Performance, Reliability und Data Protection garantiert werden. Dies ist mit einer einzelnen Harddisk nicht möglich. Aufgrund der redundanten Speicherung von Daten auf mehreren physikalischen Harddisks wird Data Availability und (Single) Fault Tolerance erreicht. So kann das System auch bei Ausfall einer Harddisk weiterhin ohne Unterbruch und ohne Datenverlust operieren.
Die RAID Technik basiert auf drei Praktiken: *Striping* um die Performance zu erhöhen, *Mirroring* und *Parity* für die Datenredundanz (Datensicherheit). Ursprünglich wurden fünf RAID-Level definiert (1-5). Im Laufe der Jahre kamen noch weitere Levels in Form von Kombinationen mit Ergänzungen hinzu.

RAID0: Striping RAID0 unterstützt und erhöht die Perfomance am meisten, garantiert aber keine Fehlertoleranz; es ist folglich nur nach der ursprünglichen Definition ein echtes RAID. Wenn eine Harddisk ausfällt, sind alle Daten verloren. RAID0 ist ideal für Applikationen mit hohen Bandbreiten (Video Produktion, Pre-Press, ...), wobei Data Protection nicht vordringlich sein darf.
Bei RAID0 wird ein File aufgeteilt in mehrere gleich grosse Segmente, so genannte Stripes. Die Stripes sind abwechslungsweise auf den Harddisks gespeichert. Es können Stripes auf verschiedenen Harddisks gleichzeitig angefordert werden, was die Performance erhöht (Abbildung 2.1, Seite 7).

RAID1: Mirroring Die Daten werden auf eine zweite Harddisk exakt gespiegelt. Diese Variante bietet hohe Datenverfügbarkeit und – falls die Daten abwechselnd von beiden Harddisks gelesen werden – eine schnelle Read Perfomance. Bei einem Ausfall einer Harddisk sind die Daten auf der anderen Harddisk weiterhin verfügbar (Abbildung 2.2, Seite 7).

RAID2 RAID2 verwendet den Hamming Error Correction Code um Fehlertoleranz und parallelen Zugriff für die Harddisks zu garantieren.

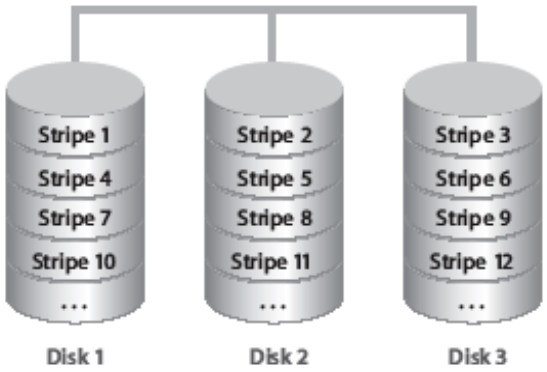


Abbildung 2.1: Striping, veranschaulicht an einem einfachen Beispiel

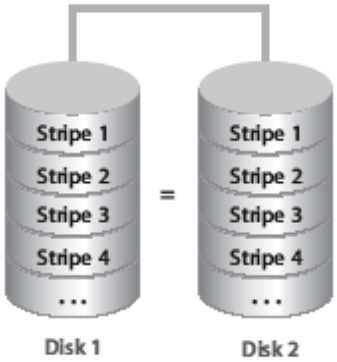


Abbildung 2.2: Mirroring, veranschaulicht an einem einfachen Beispiel

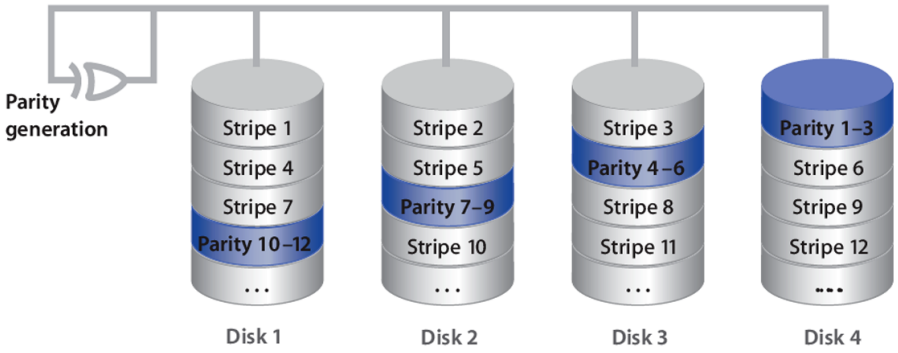


Abbildung 2.3: Striping mit verteiltem Parity, veranschaulicht an einem einfachen Beispiel

RAID3: Striping mit Parity Die Daten werden über mehrere Disks auf die Stripes verteilt. Die zugehörigen Paritiy Daten (XOR-Berechnung der Daten-Stripes) sind auf einer weiteren Harddisk gespeichert. RAID3 bietet schnelle Leseraten und hohe Datenverfügbarkeit mit grösserer Speichereffizienz. Es sind mindestens drei Harddisks notwendig.

RAID5: Striping mit verteiltem Parity RAID5 funktioniert ähnlich wie RAID3. Nur werden die Parity-Stripes regelmässig auf alle Harddisks verteilt. Dadurch können nicht auf einmal alle Parity-Stripes ausfallen, und eine Rekonstruktion der Daten ist weniger rechenintensiv. Dies ermöglicht maximale Read Performance und eine hohe Write-Performance (Abbildung 2.3, Seite 7).

Verschiedene RAID-Level Kombinationen Um die Vorteile der einzelnen RAID-Level miteinander zu verbinden, können sie beinahe beliebig miteinander kombiniert werden. So realisiert ein RAID10 zum Beispiel ein Mirroring über ein Striping.

2.3 Über FPGAs

FPGA steht für ≪Field Programmable Gate Array≫. Ein VHDL-File oder -Projekt wird synthetisiert. Das resultierende Bit-File wird auf das FPGA geladen. Anhand der im Bit-File enthaltenen Informationen entsteht in kürzester Zeit eine Hardware, welche die logischen Blöcke des FPGA repräsentieren. Diese ≪Hardware≫ kann rasch und flexibel wieder geändert, erneut generiert und getestet werden, indem ≪Änderungen≫ neu synthetisiert und erneut auf das FPGA geladen werden.
XILINX bietet heute FPGAs in der Virtex II Pro Linie an, die einen PowerPC Prozessor enthalten [4], auf denen ein Betriebssystem gestartet werden kann. Dadurch sind Möglichkeiten gegeben, einen TCP/IP-Stack oder ein File-System (NFS, CIFS, etc.) zu instantieren. Mit dem Betriebssystem können auch Applikationen ausgeführt werden. Ein Beispiel ist ein Server für web-basierten Zugriff auf die Konfiguration der Software. In der Virtex II Pro Linie sind FPGAs mit bis rund 125 000 Logic Blocks, mehreren PowerPC Prozessoren, gegen 24 Rocket I/O Transceivers im Angebot. Der ≪On-Chip Peripheral Bus≫ (OPB) stellt einen schnellen Bus dar, an den verschiedene Eigenentwicklungen oder ≪Intellectual Property Cores≫ (IP Cores) angeschlossen werden können. Interessant sind vor allem der OPB und die Rocket I/O Transceivers. Im Falle der Rocket I/O wird eine serielle Verbindung zwischen zwei Systemen mit 600 Mbps bis 3,125 Mbps pro Kanal ermöglicht. Für die Anbindung eines (eigenen) IP Cores an den (parallelen) OPB existiert eine definierte Schnittstelle als Modul, welches die Anbindung stark vereinfacht.

Kapitel 3

Die Problematik

3.1 Aktuelle Situation

Der wachsende Markt für einen an ein Netzwerk angeschlossenen Speicher ist das Resultat der zunehmenden Nachfrage nach Speicherkapzität. Für Benutzer, insbesondere Systemadminstratoren und Systemverantwortliche, ist es zunehmend erstrebenswert, die laufende Entwicklung der aufkommenden Technologien mitzuverfolgen.
Um eine Festplatte direkt in einem Netzwerk ansprechbar zu machen, braucht es dafür spezialisierte Hardware. ≪Network Attached Storage≫ (NAS) ist dazu ein weitverbreiteter Lösungsansatz. Wie nun dieses System schlussendlich realisiert wird, ist abhängig von der verwendeten Technologie und den Zielsetzungen für das Produkt. Nur schon die drei verschiedenen Standards (SCSI, SATA und PATA), eine Festplatte anzusprechen, bieten eine Vielfalt an Möglichkeiten.
Hinzu kommen noch weitere Punkte, die das Design beeinflussen. Zum einen soll das angestrebte System möglichst viel Speicherplatz bieten, und zum anderen soll es nicht zu teuer sein. Es sollte sich ohne Probleme konfigurieren und warten lassen. Anforderungen an Datensicherheit und -integrität führen zu zusätzlichen Systemerweiterungen. Das Kundensegment für die meisten auf dem Markt angebotenen Systeme sind nicht der private Benutzer zuhause oder eine kleinere Firma (KMU), sondern vielmehr Grossfirmen mit einem höheren Budget. Darum sind die meisten der brauchbaren Systeme in der oberen Preisregion angesiedelt.
Nicht unbedingt aber müssen alle erwünschten Funktionalitäten implementiert sein. Falls hohe, wenn nicht sogar vollständige Ausfall- und Betriebssicherheit (Reliability) gefordert ist und nicht der Wunsch nach viel Speicherplatz überwiegt, ist das beste auf Speicheransprechung optimierte System nicht gut genug.

3.2 Problemdefinition

Ein System soll die Vorteile von NAS optimal nutzen und gleichzeitig in einem vernünftigen Preisrahmen liegen. Mit dem Einsatz von aktueller Technologie wie zum Beispiel einem FPGA mit integriertem Prozessor ist die Entwicklung eines solchen Systems möglich. Mit dieser Arbeit soll abgeklärt werden, ob sich die Trends betreffend NAS mit einem kostengünstigen Ansatz durchbrechen lassen.
Dabei stehen nicht einzelne Punkte, in welchem das System besonders gut abschneidet, im Vordergrund. Vielmehr interessiert eine gleichmässige Verteilung der Kräfte auf verschiedene Bereiche. So deckt das Endsystem möglichst viele Punkte des Systemdesigns ab. Bei kritischen Elementen, wie der Datensicherheit oder der Datenverfügbarkeit, ist noch eine weiterreichende Optimierung gefragt.
Die meisten bisherigen Lösungen konzentrieren und spezialisieren sich auf einen bestimmten Teilbereich. Sie decken meistens den Ausfall einer Harddisk ab und bieten somit Redundanz auf Diskebene. Es ist jedoch ein System erwünscht, das sowohl auf Disk-Ebene als auch auf Kontroller-Ebene redundant ist. Das Systemdesign soll auch den Punkt der Skalierbarkeit abdecken. Die Flexibilität steigert sich dadurch noch mehr.

3.3 Zieldefinition

Im Folgenden sind die oben angesprochenen Punkte als Zieldefinitionen erläutert (Kurzübersicht in Tabelle 3.1, Seite 11). Das NAS-System ist direkt via Netzwerk (IP-Adresse) von einem Client (oder Server) aus per File-Services wie NFS (Network File System) oder SMB (Server Meassage Block) ansprechbar. Beim Design werden die modernsten Technologien wie SATA, Embedded OS oder Gigabit-Ethernet berücksichtigt.

Performance Bietet das System einen hohen Datendurchsatz bzw. hohe Datentransferraten, wird die Performance bereits optimiert. Maximiert man nun noch den Speicherplatz im Speichersubsystem und minimiert man die Antwortzeit – die Antwortzeit oder Response Time ist die benötigte Zeit um einen Request an einem Disksystem abzuarbeiten – lässt sich die Performance noch weiter steigern.

Reliability Verlust von Daten oder ein verweigerter Zugriff auf Daten aufgrund von physikalischen Defekten oder anderen Ungereimtheiten ist ein kritischer Aspekt für ein System. Inbegriffen ist u.a. der Umgang mit Applikationen und Daten. Der Betrieb muss zu jedem Zeitpunkt sichergestellt sein. Der Anwender merkt nicht, wo ein Fehler aufgetreten ist. Das System ist *single fault tolerant*. Dabei dürfen keine Daten verloren gehen.

Availability Ein Anwender erwartet, dass die Daten immer präsent und erreichbar sind. Das System erkennt Fehler automatisch und reagiert mit entsprechenden Massnahmen. Es bietet möglichst guten Schutz bei einzeln auftretenden Fehlern. Diese sind für den Anwender transparent.

Data Integrity Die Daten müssen zu jedem Zeitpunkt korrekt und frei von jeglicher Korruption sein.

Scalability Das System unterstützt die Möglichkeit, mehrere gleichartige NAS-Systeme miteinander zu koppeln und zu erweitern, um die Performance sowie die Kapazität bei gleichbleibendem Aufwand zu erhöhen.

Interoperability Ein IT-Umfeld wird mehr und mehr zu einer heterogenen Umgebung. Ein Speichersystem muss mit verschiedenen Betriebssystemen und anderen Netzwerk-Speicherelementen unter Verwendung von Industriestandards interagieren können[1]. Die Zugangskontrolle für die Dokumentenverwaltung wird durch ein eingebettetes Betriebssystem sichergestellt.

Ziel	**Eigenschaft**
Performance	- Hoher Datendurchsatz, hohe Datentransferraten - Maximierung des Speichersubsystems - Minimierung der Antwortzeit
Reliability	- Ausfall-, Betriebs- und Funktionssicherheit - Korrekte physikalische Funktion des Systems
Availability	- Beinhaltet Verfügbarkeit und effektive Ausfallsicherheit - Hohe Wahrscheinlichkeit, dass ein Disk Array einen Request beantwortet - Die Daten sind immer vorhanden und verfügbar - Datenredundanz (das Disk Array ist ≪single fault tolerant≫)
Data Integrity	- Daten zu jedem Zeitpunkt korrekt - Daten frei von Korruption
Scalability	- Skalierbarkeit des System mit mehreren gleichen NAS-Systemen - Erhöhung der Performance und Kapazität bei gleichbleibendem Aufwand
Interoperability	- Interoperabilität mit verbreiteten Betriebssystemen - Accesscontrol durch ein ≪Embedded Linux≫ sichergestellt

Tabelle 3.1: Zieldefinitionen für das Systemdesign

[1] SMB (Server Message Block) für Windows Systeme und NFS (Network File System) für Apple und Linux Systeme

Teil II

Systemdesign

Kapitel 4

Lösungserarbeitung

Um die gesetzten Ziele zu erreichen, bieten sich verschiedene ergänzende Technologien an. Mit dem RAID-Ansatz lässt sich vor allem das Hauptproblem der Daten-Erreichbarkeit angehen. Die Anbindung der Festplatten basiert auf Standard-Technologie.
Die Idee ist ein SOC (System On a Chip) mit einer Aufteilung in Hardware- und Software-Ebene. Die Implementierung der Kontroller und der Interfaces erfolgt in Hardware auf dem FPGA. Das verwendete Betriebssystem für den integrierten PowerPC ist Embedded Linux. Das System sollte weitgehendst unabhängig sein vom Betriebssystem. Erweiterungen wie zum Beispiel die Verschlüsselung der Daten (Kryptographie) durch separate Algorithmen oder Bausteine sind denkbar.
In den folgenden Abschnitten werden die einzelnen Gesichtspunkte des Systemdesigns diskutiert. Tabelle 4.1 auf Seite 15 zeigt eine Übersicht einiger Systemdaten.

Komponente	**Angaben**	**Bemerkungen**
Speicheradresslänge	48 Bit	mehrere Terabytes ansprechbar
Datenblöcke	512 Byte	entspricht 4kBit
Stripe-Grösse	4 kByte	variabel [1]
Festplattengrösse	3.5 e13 Bytes	maximal (eine Festplatte)
Ethernet	1 GBit / s	
Externes RAM	mind. 64 MByte	DIMM, evtl. auch fix integriert
RAM im FPGA	100 kByte	im FPGA bereits vorhandenes Block-RAM
Flash	512 MByte	Betriebssystem und FPGA Konfiguration
Link	bis 3 GBit/s	über Rocket I/O von FPGA
SATA	1,5 GBit/s	über Rocket I/O von FPGA

Tabelle 4.1: Überblick über die wichtigsten Systemdaten

[1] Sind die Stripes zu klein, ist der Aufwand beim Aufteilen einer grossen Datei zu hoch; sind die Stripes zu gross, ergibt sich für kleine Dateien keinen Nutzen.

4.1 Anwendungsbereich und Lokalisation

4.1.1 Anwendungsbereich des Systems

Das System-Design orientiert sich hauptsächlich an den Anforderungen für kleinere Speichersysteme. Was aber eine Erweiterung auf ein komplexeres und grösseres Speichersystem nicht ausschliesst.
Für den Einsatz in KMU[2] eignen sich Speichergrössen bis wenige Terrabytes. In diesem Grössenbereich macht auch das Mirroring – welches doppelt so viele Harddisks benötigt, wie effektiv verwendet werden können – von Daten Sinn. Bei anderen Grössenordnungen werden andere RAID-Level für das Erreichen von Performance in einem vernünftigen Preis-Leistungsverhältnis eingesetzt. Ein hochverfügbarer Speicher mit einem speziellen Augenmerk auf die Redundanz sowohl auf Disk- als auch auf Kontrollerebene ist auch für kleinere Systeme interessant.

4.1.2 Lokalisation im Netzwerk

Wo im Computernetzwerk lokalisiert sich das System? Mehrere Clients sollen gleichzeitig auf den Speicher Zugriff haben, der sich direkt im Netzwerk befindet. Die Festplatten sind von den Herstellern gegeben, das Netzwerk besteht im Normalfall auch schon. Es braucht ein Bindeglied zwischen Netzwerk und Speicher, welches die Anfragen an den Speicher verarbeitet (Abbildung 4.1). Diese Blackbox kommuniziert auf beiden Seiten per Industriestandards.
An einen Speicher im Netzwerk werden die unterschiedlichsten Erwartungen gestellt. Für eine Datenbank-Applikation einer grossen Firma bestehen andere Erwartungen an ein Speichersystem als für jene einer kleineren Firma, die einen gemeinsamen Speicher für ihre Daten benötigt. Je nach Anwendung spielen Durchsatz, Verfügbarkeit, Wartung und Bedienbarkeit eines solchen Systems eine wichtige Rolle.

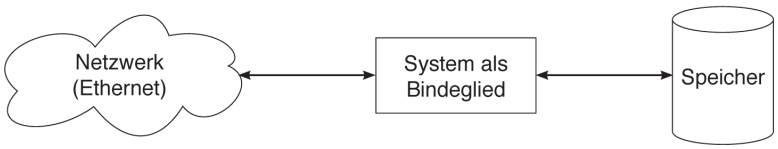


Abbildung 4.1: Ein Kontroller als Verbindung zwischen Netzwerk und Speicher

[2] Klein- und Mittlere Unternehmen

4.2 Varianten und Alternativen

4.2.1 RAID und ähnliche Lösungsansätze

Systemintern bestehen viele Möglichkeiten, mit RAID oder ähnlichen Ansätzen Datensicherheit, Performance und *Single Fault Tolerance* zu implementieren bzw. zu optimieren. Je mehr logische Speicher-Elemente angestrebt werden, desto mehr Speicherkapazität hat ein System, desto leistungsfähiger muss der Kontroller sein.
Zuest gilt es zu bestimmen, wie viele logische Speicherelemente, die dem Benutzer als eigene Festplatte erscheinen, pro System verwendet werden sollen. Mehrere logische Speicherelemente erhöhen die Komplexität des Kontrollers. Zudem sind bei einem Ausfall des Kontrollers gleich mehrere solche Harddisks nicht mehr ansprechbar. Das Ziel ist Redundanz auf Kontroller- und Festplatten-Ebene. Dies kann erreicht werden, wenn man sich auf eine logische Einheit beschränkt. Nachfolgende Überlegungen konzentrieren sich daher auf letzeren Fall.

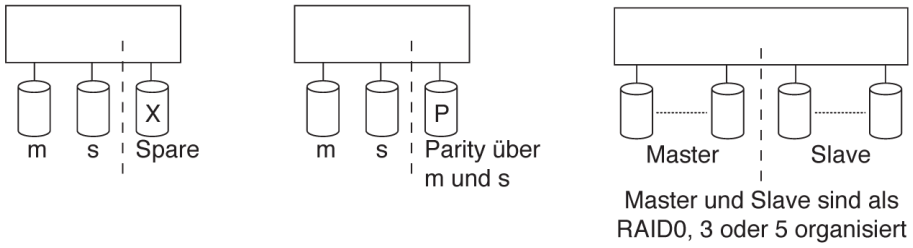


Abbildung 4.2: Die verschiedenen Anwendungsmöglichkeiten von mehreren Festplatten

Die Anzahl der Harddisks ist je nach verwendeter Hardware gegen oben begrenzt. Die einfachste Lösung stellen zwei einzelne Harddisks als Master und Slave[3] dar. Wird nun eine zusätzliche Harddisk – entspricht einer *Spare-Harddisk* – als Ersatz geführt, kann sie bei einem Ausfall einer Harddisk deren Aufgaben übernehmen. Der Kontroller muss allerdings dementsprechend programmiert sein. Dabei ist während der meisten Betriebszeit ein Harddisk-Anschluss selten verwendet, was im Vergleich zu der relativ geringen Ausfallwahrscheinlichkeit einer Festplatte einem verschwendeten Anschluss gleichkommt.
Dieselbe Festplatte könnte auch als Parity-Harddisk verwendet werden. Das Paritybit wird dabei über den Master und den Slave berechnet. Dies entspricht nicht ganz RAID3, da die Daten auf den beiden Harddisks nicht derselben logischen Einheit entspringen. Die erforderliche zusätzliche XOR-Funktionalität kann im FPGA schnell ausgeführt werden. Ein Vorteil ist u.a. das lokale Wiederherstellen der Daten nach einem Harddisk-Ausfall.
Im Weiteren sind die verschiedenen RAID-Implementationen wie Striping und Parity-Berechnung möglich. Die RAID-Konfigurationen sind hierbei über den Master und den Slave angewendet. Bei einem RAID0 steigt die Read- bzw. Write-Performance an, und die Ausnutzung der Diskkapazität erhöht sich. Bei all diesen Varianten sind allerdings mehrere Harddisks nötig, was zu einem Overhead an Harddisks führen kann bzw. den Preis eines Systems ansteigen lässt. Abbildung 4.2 zeigt die verschiedenen Varianten, und Tabelle 4.2 gibt einen Überblick über die Möglichkeiten.

[3] Auf dem Slave-System werden die gespiegelten Daten gespeichert.

Variante	**Vorteile**	**Nachteile**
Spare-HD	Funktionsübernahme bei Ausfall	WSK von Ausfall relativ klein HD-Anschluss selten verwendet
Parity	lokales Reconstruction	zusätzliche XOR-Berechnung
RAID0	höhere Read-/Write-Performance Kapazität erhöht	keine Datensicherheit
RAID3/5	Performance lokaler Datenabgleich	teuer zusätzliche HDs

Tabelle 4.2: Vor- und Nachteile der verschiedenen Varianten von Harddisk-Anordnungen im lokalen System

4.2.2 Die Grundidee

Kontrollersicherheit erreicht man entweder durch das doppelte Vorhandensein des Kontrollers im gleichen System oder durch das Aufteilen auf mehrere Systeme. Die Umsetzung des zweiten Vorschlags erfolgt mit RAID1. RAID1 kann als einziges der RAID-Level auf zwei autonome Komponenten aufgeteilt werden, weshalb es auch als oberste Ebene für die Systeme vorgesehen wird. Die gespiegelten Daten werden auf einem anderen System gespeichert. Somit wird die Redundanz auf Disk- und Kontrollerebene garantiert.
Beim Beispiel von zwei zusammengeschalteten Systemen, spiegelt jedes System seine Daten über den internen Link zum jeweiligen rechten Nachbarn (Abbildung 4.3). Von den beiden Festplatten ist die eine als *Master* benannt und die andere als *Slave*. Der Slave ist dabei diejenige Festplatte, auf welche die Daten (vom Master) gespiegelt werden.

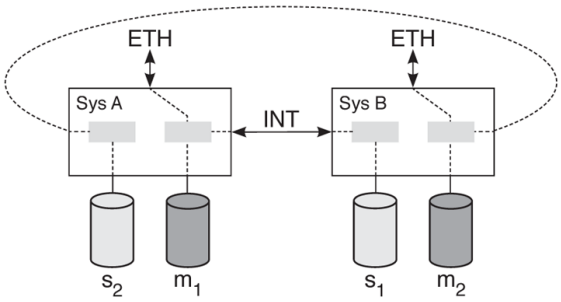


Abbildung 4.3: Zwei Systeme spiegeln ihre Daten auf den jeweiligen Nachbarn

Auf diese Weise wird die Ausfallsicherheit sowohl auf Harddisks- als auch auf Kontroller-Ebene garantiert. Fällt zum Beispiel eine Harddisk aus, sind die Daten auf der gespiegelten Harddisk nach wie vor verfügbar. Bei einem Kontroller-Ausfall übernimmt das benachbarte System die Funktionaliät des ausgefallenen Systems, d.h. dessen IP-Adresse wird auf die MAC-Adresse des gesunden Systems – zusätzlich zu dessen eigenen IP-Adresse – gemappt (Abbildung 4.3, Seite 18).
Natürlich lässt sich diese Anordnung mit beliebig vielen Systemen weiter skalieren. Allerdings braucht es bei steigender Anzahl von Systemen ebenso viele unterschiedliche IP-Adressen, und die Kosten für die Harddisks steigen. Für das Mirroring braucht es doppelt so viele Festplatten wie benutzt werden können. Durch die Standard-Schnittstelle Ethernet und ohne die Verwendung von eigenen Protokollen bleibt das System flexibel im Umgang mit gängigen Computer- und Netzwerk-Architekturen. Einzelne Fehler sind für den Benutzer transparent, die Anordnung ist *Single Fault Tolerant*.

4.2.3 Skalierbarkeit der Systeme

Unter Berücksichtigung von verschiedenen Faktoren, die einschränkende Wirkung auf die Skalierbarkeit haben können, ergeben sich mehrere Möglichkeiten.
Die naheliegendste Kombination ist wohl die Anordnung von mehreren Systemen in einem Netzwerk oder das Verbinden durch einen grossen Switch (Abbildung 4.4). Der Link für die Duplikation der Daten auf ein benachbartes System ist unabhängig vom Netzwerk-Interface. Dieses stellt allerdings einen Flaschenhals dar. Damit man genug Systemleistung erhält, sollte das Netzwerk mit Gigabit-Ethernet laufen. Dies erfordert eine zügige Abarbeitung des TCP/IP-Stacks mittels eines leistungsfähigen Prozessors. Dies kann entweder ein im FPGA interner oder ein externer, spezialisierter Prozessor sein. Ausserdem bildet der Switch einen *Single Point of Failure*. Die Variante mit einem oder mehreren Switches oder anderen Netzwerkstrukturen werden in dieser Arbeit nicht weiter untersucht.

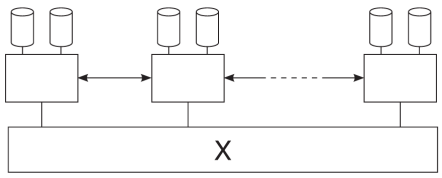


Abbildung 4.4: Die Systeme sind in einem Netzwerk angeordnet und durch einen Gigabit-Switch verbunden

Jedes System ist autonom im Netzwerk ansprechbar. Die Systeme sind zu einer Ringstruktur verbunden. Über den speziellen Link werden die Daten ebenfalls unabhängig vom Netzwerk übertragen. Diese Links können im Ring entweder unidirektional oder bidirektional betrieben werden. Die Komplexität bei bidirektionalem Betrieb ist kaum spürbar, da die Systeme einerseits nur mit einem benachbarten System aufs Mal kommunizieren müssen und andererseits kein kompliziertes Protokoll für den Datenaustausch verwendet werden muss.

In einem Ring wird pro System ein Storage-Element gemountet und stets auf die Harddisk des rechten Nachbarn gespiegelt. Falls ein Kontroller aussteigt, sind die zugehörigen Links bis zur Fehlerbehebung unterbrochen. Bei Verwendung von einem einzigen System steht die Kontrollersicherheit nicht zur Verfügung, da die Daten über den zurückgeführten Link via den gleichen Kontroller ins gleiche System geschrieben werden. Datenredundanz allerdings wird damit ebenso garantiert wie mit mehreren Systemen.

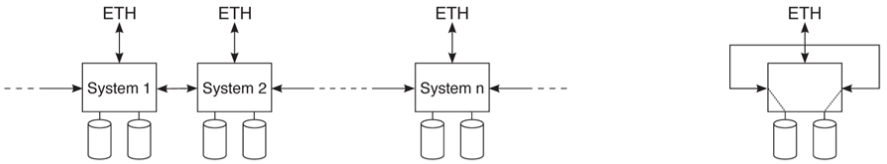


Abbildung 4.5: Die Systeme sind in einem Ring angeordnet

Ein regelmässiges Versenden von Testnachrichten an die Nachbarn kontrolliert deren Vorhandensein. Ein Systemausfall wird durch eine negative oder ausbleibende Antwort festgestellt.

Variante	**Vorteile**	**Nachteile**
Netzwerk	totaler Speicherplatz zentral	Single Point of Failure (Switch) erhöhte Komplexität
Ring	beliebig erweiterbar Ethernet unabh. vom Link	zusätzliche Verbindungen

Tabelle 4.3: Verschiedene Varianten, ein grösseres Netzwerk mit mehreren Systemen zu skalieren

4.2.4 Harddisk-Anbindung

Ein Ansatz, um die wachsenden Harddisk Transferraten zu bewältigen, ist Serial-ATA (SATA). Nebst ATA ist auch SCSI (Small Computer System Interface) ein Standard für die Anbindung von Peripheriegeräten. SCSI ist wie Parallel-ATA eine parallele Schnittstelle.[8]

Parallel Ultra ATA
Der Standard benutzt einen 16 Bit breiten parallelen Datenbus und 16 Kontrollsignale über einen 40-Pin Connector. Um den idealen Durchsatz von 100 MBytes/s zu erreichen, muss der Bus mit 50 MHz getaktet werden. Das Taktsignal wird bei der Quelle generiert und mit den Daten zusammen übertragen. Über 40 Leitungen werden 66 MByte/s Daten übertragen, bei alternierenden Ground-Leitungen sind es 80 Leitungen und 133 MByte/s.

Serial ATA
Serial ATA benutzt einen einzelnen Signalpfad, um Daten seriell zu übermitteln. Ein zweiter Signalpfad dient dem Senden von Bestätigungen an den Sender. Die beiden unidriektionalen Links bestehen aus je zwei Kabeln, verwendet als differentielles Paar (Differenzensignal); somit hat ein SATA-Bus nur vier Datenleitungen und ist mit einem einfachen Transmitter-/Receiver-Glied implementierbar. Um den Verlust der Parallelität auszugleichen, muss der Bus mit einem höheren Takt betrieben werden. Mit einem Bit pro Takt, 8b10b Codierung[4] und einem Takt von 1500 MHz erreicht man einen theoretischen Durchsatz von maximal 150 MBytes pro Sekunde.

SCSI
Die SCSI-Technik ist ausgereift und wird laufend verbessert. Derzeit sind Datentransferraten von bis zu 320 MByte/s erreichbar. Mit dem Einsatz der LVD-Schnittstelle (Low Voltage Differential – Differenzensignal) ist der Anschluss von 15 Geräten über Kabel mit einer Länge von 12 Metern möglich. Es werden 16 Bit parallel übertragen. Weit höhere Geschwindigkeiten erreicht man mit serieller Übertragung (Serial Attached SCSI).
SCSI ist keine Punkt-zu-Punkt Verbindung. Es sind mehrere Initiatoren, die eine Bustransaktion beginnen können. Als Bus benötigt SCSI eine Terminierung. SCSI ermöglicht auch das Anschliessen externer Geräte. Die CPU-Belastung bei SCSI ist wesentlich geringer als bei IDE. Je nach Busbreite ist eine unterschiedlich hohe Anzahl von Leitungen nötig [5].

	PATA	**SATA**	**SCSI**
Maximum Speed	100 MByte/s	150 MByte/s	320 MByte/s
Cable Pins	80	7	50 / 68
Data Wires Used	26	4	8 / 16
Signalling Level	5V	250mV	
Hot Swappable	No	Yes	No
Connection Method	Master & Slave	Point-to-Point	Bus mit max 15 Geräten

Tabelle 4.4: Die wichtigsten Unterschiede von Serial-ATA, Parallel Ultra ATA und SCSI

Weitere Gründe für SATA
Drei Ground-Kabel für vier Datenkabel mit einem differentiellen Signal haben einen abschirmenden Einfluss. Sie beheben Crosstalk[6], und es entsteht auch kein *Ground Bounce*. Dank der 8b10b-Codierung kann der Takt aus dem Signal gelesen werden. Da SATA nicht mit alten ATA-Standards kompatibel ist, muss SATA nicht mehr 5V-tolerant sein. Differentielle Leitungspaare ermöglichen ein *Low Voltage Signaling*.

[4] Bei der 8b10b Codierung werden 8-Bit-Wörter auf 10-Bit-Wörter abgebildet. Dabei wird berücksichtigt, dass eine Bitgruppe genügend Pegelwechsel aufweist, damit das zur Synchronisation nötige Taktsignal abgeleitet werden kann.
[5] SCSI mit 8 Bit Busbreite hat ein 50-poliges Kabel, dasjenige mit 16 Bit Busbreite ein 68-poliges Kabel
[6] Bei Crosstalk werden Ströme durch die magnetischen Felder in benachbarten Kabeln induziert.

Die Schnittstellen von SATA und PATA gegenüber dem Betriebssystem sind genau gleich. Softwareseitig ist SATA kompatibel mit dem heutigen PATA. Ein Shadow-Registersatz emuliert den bisher in der Festplatte untergebrachten Registersatz bei PATA. Die Vorteile von SATA liegen in folgenden Punkten, weshalb für das System-Design der SATA-Standard sinnvoller ist:

- Punkt-zu-Punkt Verbindungen garantieren maximal 150MBytes pro Sekunde
- Geringere Komplexität und weniger Platz für Routing nötig
- Längere Leitungen sind möglich (bis 1 Meter)
- CRC Error Checking für alle Daten und Kontrollinformationen
- Hot Swappable
- First-Party[7] DMA Support

4.2.5 Verwendung von RAM

Der Zugriff auf eine Festplatte ist um den Faktor 1000 zeitintensiver als der Zugriff auf ein RAM. Die Antwortzeit des Systems ist ein wichtiger Punkt für die Performance. Ein Request soll möglichst rasch beantwortet werden können. Lagert man die zu schreibenden Daten in einem RAM, sind sie gegenüber dem Betriebssystem schneller bestätigt. Um Daten zu lesen muss so oder so auf die Festplatte zugegriffen werden. Ein RAM bringt folglich nur für das Schreiben Vorteile.

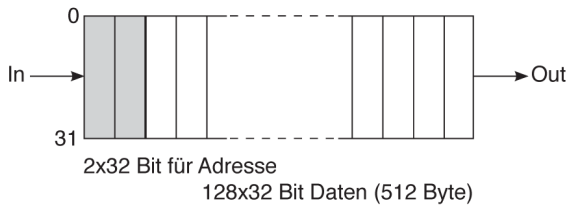


Abbildung 4.6: Die Organisation des FIFO im NVRAM

Da die Daten bei einem Stromunterbruch nicht verloren gehen dürfen, muss das RAM nonvolatile (nicht flüchtig) und somit batteriegestützt sein (NVRAM). Daten werden auch bei einem Datenabgleich nach einem Ausfall auf eine (neue) Festplatte geschrieben. Auch hier hilft ein NVRAM. Bei der Verwendung des RAMs als FIFO-Queue für die Daten ergeben sich die beiden folgenden Fragen.

[7] Das Gerät selbst und nicht der externe DMA-Kontroller (Direct Memory Access) übernimmt die Arbeit des Datentransfers.

- **Art und Weise der Datenspeicherung im FIFO**
  Ein Datenpaket des Betriebssystems beträgt 512KByte. Im FIFO können entweder ganze Blöcke aufs Mal gespeichert werden – fordert ein sehr breites FIFO – oder ein Block wird in mehrere Segmente gleicher Grösse aufgeteilt. Letztere Variante ist sinnvoller, da die Daten im Kontroller auf einem Bus weitergereicht werden können. Ansonsten müsste ein breites Register als Zwischenspeicher fungieren. Dies behindert aber die Parallelität. Die FIFO-Breite wird der Busbreite angepasst.

- **Speichern zugehöriger Adressen von Daten**
  Zu jedem Datenpaket – ob aufgeteilt in Segmente oder nicht – ist die zugehörige Adresse zwingend mitzuspeichern. Eine Adresse ist 48 Bit breit. Sobald die Daten in Segmente aufgeteilt werden, lohnt es sich nicht, zu jedem Segment die Adresse zu speichern. Diese wird dann besser zu Beginn einmal gespeichert (Abbildung 4.6, Seite 22). Die Unabhängigkeit der Kontroller rührt von der (internen) Organisation im FIFO und nicht von der Art und Weise der Speicherdefinition im FIFO.

4.2.6 Software

Als Betriebssystem steht das Linux von MontaVista [13] zur Verfügung. Es bietet eine Plattform für gebräuchliche Source- und Binary-Formate. Die professionelle Version unterstützt moderne Betriebssystemfunktionen.
Damit die Daten vom Netzwerk via das Betriebssystem und den lokalen Datenbus zum RAID-Kontroller gelangt, braucht es einen Treiber, der mit den an den Bus angeschlossenen Komponenten entsprechend kommunizieren kann. Damit sich dieser Treiber und das System verstehen, muss der Treiber eigens dafür programmiert und optimiert werden.
Für die gegenseitige Überwachung zweier Systeme (Kontrolle bei Systemausfall) wird eine spezielle, für solche Aufgaben entwickelte Applikation verwendet: das Linux-HA (Linux High-Availability).

Linux-HA
Es exisiteren zwei physikalische Verbindungen zwischen jeweils zwei Systemen. Die eine Verbindung besteht über das Netzwerk (Ethernet), die andere über einen dedizierten seriellen Link. Eine hardware-basierte Lösung mit einem zyklischen Testsignal über den Link alleine genügt nicht. Vor allem dann nicht, wenn bei einem totalen Systemausfall das Nachbarsystem die Funktionalität und somit die IP-Adresse übernehmen soll.
Es ist also unumgänglich, einen System-Ausfall oder die Wiederverfügbarkeit eines Systems zu entdecken. Der ≪Heartbeat-Code≫ nimmt sich dieser Probleme an. In Linux-HA ist ein solcher Heartbeat-Algorithmus implementiert. Linux-HA sendet sowohl über die serielle Schnittstelle, als auch über das Netzwerk *keepalive*-Nachrichten als Spezialfall von generellen Kommunikationsnachrichten. Bei einem Ausfall zum Beispiel kann ein *gesundes* System mit Linux-HA die IP-Adresse des fehlerhaften Systems übernehmen. Die IP-Adressen müssen aber vorher programmiert sein. So kann hohe Verfügbarkeit garantiert werden. Nachfolgende Auflistung zeigt die wichtigsten Punkte von Linux-HA.[9]

- Unterstützt Authentifikation (Schutz gegen Angriffe auf ein System mit den One-Way-Hashfunktionen[8] MD5 oder HMAC-SHA1, oder einem 32-Bit CRC)
- Schlüssel und Authentifikationsmethoden können bei laufendem Betrieb ausgetauscht werden
- Massnahmen (Sequenznummern) gegen Reply-Attacken
- Einfaches Format (auf ASCII basierend), welches für alle (auch ältere) Versionen von Linux-HA verständlich ist
- API (Application Programming Interface) mit Basisfunktionen

4.3 Zahlen und Abschätzungen

Einschränkende Faktoren – eine Übersicht gibt Abbildung 4.7 – sind je nach verwendeter Technologie der interne Link zwischen zwei Systemen, dem FPGA oder die Schnittstelle zur Festplatte (ATA). Der Durchsatz ist allerdings am meisten von der Performance des Ethernet-Interfaces abhängen. Der TCP/IP-Stack braucht viele Ressourcen des PowerPC. Der Prozessor läuft theoretisch mit 400MHz, praktisch mit 300 MHz. Er muss eine gewaltige Rechenleistung aufbringen, um den TCP/IP-Stack abzuarbeiten und so den Durchsatz im Ethernet zu bewerkstelligen.

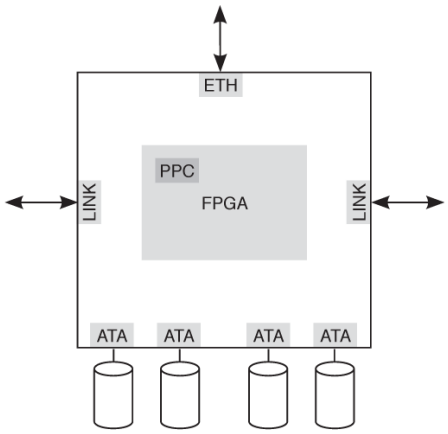


Abbildung 4.7: Die kritischen Stellen im System können an verschiedenen Orten liegen

[8] Eine Hashfunktion erzeugt per Verschlüsselung aus einem Input immer eine gleich lange Zeichenkette

Die Rocket I/O Schnittstellen am FPGA sind mit maximal 3,125 GBit/s (minimal 622MBit/s, je nach Standard) genug schnell, um die geforderte Datenrate zu liefern. Je nach dem wie schnell der Prozessor den TCP/IP-Stack abarbeiten kann, sind Datenraten zwischen 12,5MByte/s (100MBit Ethernet) bis 125MByte/s (Gigabit Ethernet) möglich. Eventuell ist ein externer Prozessor-Chip für diese Arbeit geeigneter. Für den internen Link kann ein einfaches Protokoll (siehe Tabelle 4.5 Seite 25) verwendet und dadurch auch eine sehr hohe Datenrate erreicht werden.

Serial Standard	**Datenrate/Kanal (Gb/s)**	**Baud-Rate/Kanal (Gb/s)**	**Referenztakt (MHz)**
InfiniBand	2.0	2.5	125
1 Gb Ethernet	1.0	1.25	62.5
10 Gb Ethernet	2.5	3.125	156.25
Fibre Channel	0.85 1.7 2.5	1.06 2.12 3.1875	53.125 106.25 159.375
Serial RapidIO	2.5	3.125	156.25

Tabelle 4.5: Datenrate und Baud-Rate über die Rocket I/O Schnittstelle des FPGA für verschiedene serielle Standards

Die beiden Parameter *Suchzeit*[9] und *Maximale Bandbreite* definieren die Performance bei einem Harddisk-Zugriff. Der resultierende Durchsatz wird für kleine Requests von der Zugriffszeit und für grosse Requests von der maximalen Bandbreite dominiert. Ein I/O-System braucht u.a. folgende Techniken um die Zugriffszeit und den Durchsatz zu verbessern.

Caching hält oft benutzte Informationen in einem Speicher, der schneller ist als das Disk-Subsystem.

Read-Ahead liest bereits Informationen in den Cache, die eventuell erst zu einem späteren Zeitpunkt referenziert werden.

Write-Behind unterstützt das Schreiben der Harddisk während die Applikation bereits weitere Berechnungen durchführen kann.

Aggregation kombiniert viele kleinere Zugriffe in einen einzigen grossen Zugriff. Aggregation basiert auf den Mechanismen von Write-Behind und Read-Ahead.

[9] Die Suchzeit (Seek Time) einer Festplatte ist definiert als die Zeit, welche benötigt wird um den Lese- und Schreibkopf auf den richtigen Track zu bewegen und die Wartezeit, bis der richtige Block unter dem Kopf ist.

Mit SATA können Festplatten mit 150 MBytes/s Datenrate angesprochen werden. Im Durchschnitt erreicht eine Festplatte für Read- bzw. Write-Request einen Durchsatz von ungefähr 50 MByte/s. Dank RAID0 werden gleichzeitig mehrere Festplatten angesprochen, womit die Daten genug schnell zurückgegeben werden. Abbildung 4.8 zeigt einen typischen Durchsatz einer Harddisk, abhängig von der Request-Grösse.

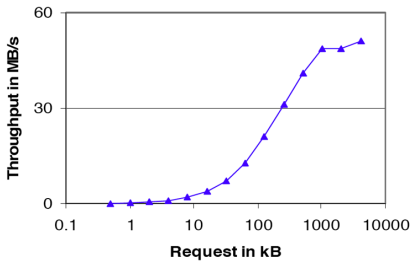


Abbildung 4.8: Typischer Durchsatz einer Schreibanfrage an eine einzelne Harddisk

Schlussfolgerung

Wenn die Netzwerkschnittstelle von einem 100 MBit Ethernet ausgeht, liegt der Flaschenhals des Systems sowohl für eine Festplatte als auch für zwei Festplatten (Striping) beim Netzwerk-Interface. Sobald aber Gigabit Ethernet verwendet und auch erreicht wird, bildet sich der Flaschenhals in der Verbindung zu den Festplatten.

Aus all den erläuterten Überlegungen resultiert ein System-Design, welches in den folgenden Kapiteln näher beschrieben wird.

Kapitel 5

Systemdesign

Die Überlegungen aus dem vorherigen Kapitel führen zum Systemdesign, welches nun detaillierter beschrieben wird. Abbildung 5.1, Seite 28, zeigt eine Übersicht über das System. Die Komponenten sind unterteilbar in Netzwerk, Betriebssystem, Kontroller und Harddisks. Nachfolgend werden hauptsächlich die Bereiche der Kontroller mit den Funktionsblöcken und die zugehörigen Abläufe wie Systeminitialisierung, Fehlerbehandlung sowie Wiederinstandstellung (Reconstruction) erläutert.

5.1 Beschreibung der Komponenten/Funktionsblöcke

Betriebssystem Als Betriebssystem (OS - Operating System) wird *Embedded Linux* verwendet, welches auf dem FPGA-internen PowerPC läuft. Das OS startet ab einem externen Flash und benötigt zusätzlich externes RAM als Arbeitsspeicher. Es bietet gegenüber dem Netzwerk Speicher-Services an und ≪sieht≫ nur eine logische Speichereinheit, welche gemountet wird. Es dient weiter für die Authentifikation, die Benutzerverwaltung sowie für allgemeine Services wie zum Beispiel einen Web-Server.

Host Interface Das Host Interface bildet die Kommunikationsschnittstelle zwischen dem Betriebssystem (Software) und dem Kontroller (Hardware).

FSM_Master Der FSM_Master ist das Herz des Systems. Er steuert den Datenfluss zwischen dem Host Interface und den Festplatten-Schnittstellen, kennt den aktuellen Systemstatus und behandelt die auftretenden Fehler. Leseanfragen werden direkt an den RAID0-Kontroller weitergeleitet. Bei Schreibanfragen werden die Daten gemäss RAID1 dupliziert und in die FIFOs von Master und Slave (via LINK) geschrieben.
Im Weiteren ist der FSM_Master zuständig für den Datenabgleich (Reconstruction) nach einem System- oder Komponentenausfall. Beim Systemstart oder -reset folgt der FSM_Master einem vorbestimmten Ablauf.

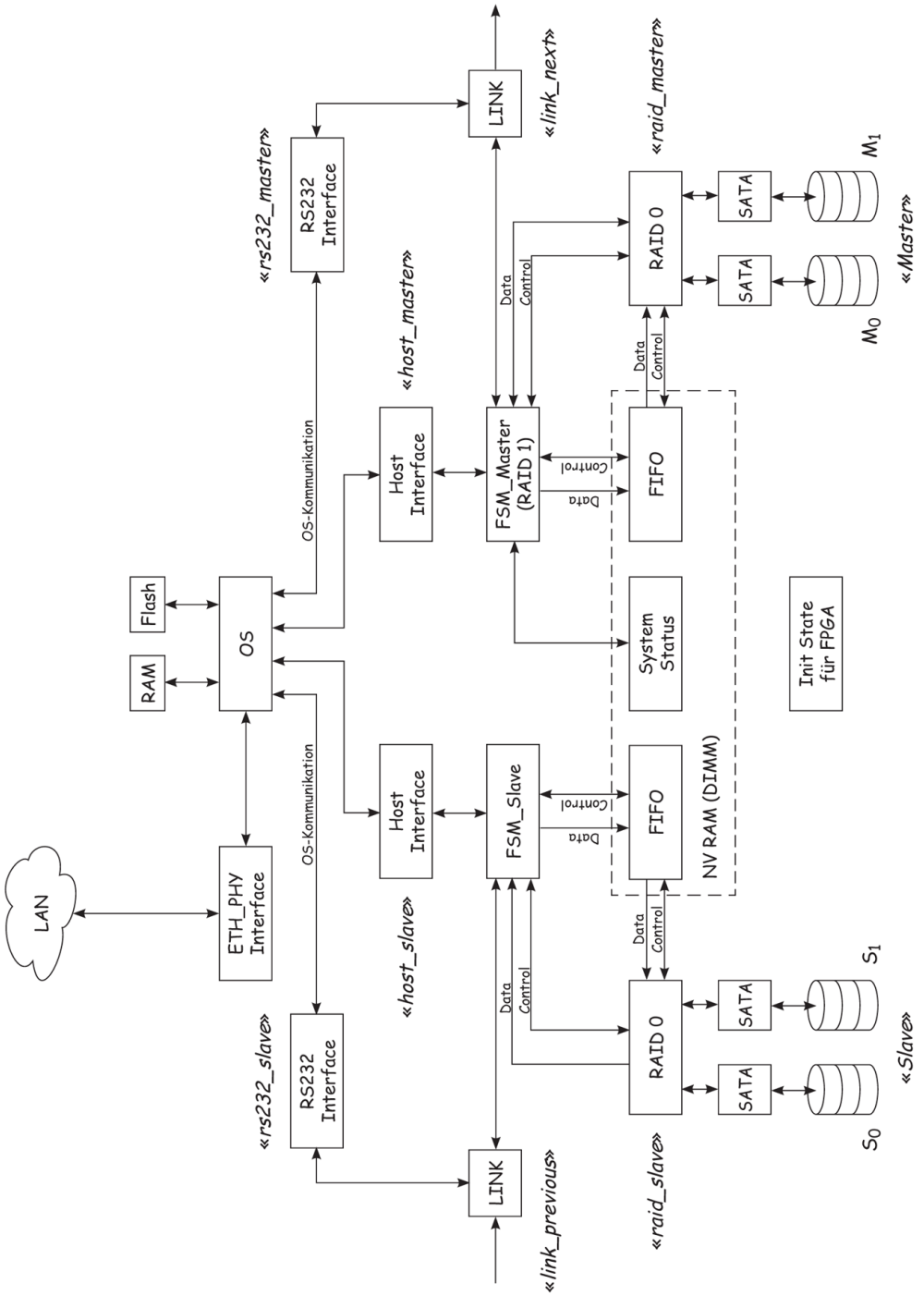


Abbildung 5.1: Übersicht über das System – Komponenten und Pfade

FSM_Slave Der FSM_Slave ist das Gegenstück zum FSM_Master. Der Slave besitzt keine RAID1-Funktionalität, sondern dient nur der Kommunikation mit dem Master und der lokalen Fehler- und Informationsverarbeitung. Für den Fall, dass das System wegen eines Ausfalls des Masters die Benutzer-Anfragen beantworten muss – die Slave-Speichereinheit wird im Slave-OS gemountet –, benötigt der FSM_Salve ebenfalls die Lese- bzw. Schreibe-Funktionalität.

LINK Die LINK-Komponente bildet die Schnittstelle zum jeweiligen rechten Nachbarn. Es ist eine serielle Schnittstelle, über welche sowohl Daten als auch Kontrollsequenzen bidirektional ausgetauscht werden. Entweder kommunizieren Master und Slave miteinander oder es werden vom OS Pakete gesendet. Über den Link wird gleichzeitig das Vorhandensein des Nachbarn festgestellt.

NVRAM Im NVRAM (Non Volatile RAM) sind der Systemstatus, temporäre Adressen und Daten gespeichert. Das FIFO bekommt die Daten im Falle einer Schreib-Anfrage vom FSM_Master und gibt sie weiter an den RAID0-Kontroller. Der Systemstatus ist für die Konsistenz des Systems wichtig und darf deshalb bei einem Stromunterbruch nicht verloren gehen (siehe auch Kapitel 5.2.2, Seite 30).

RAID0 Die beiden RAID0-Kontroller entsprechen sich sowohl auf der Master- als auch auf der Slave-Seite. Sobald Daten im FIFO vorhanden sind, liest der Kontroller diese aus, entscheidet anhand der Adresse, auf welche Harddisk sie geschrieben werden sollen, und schickt die Daten diesem Entscheid entsprechend an die jeweilige Harddisk. Bei einer Leseanfrage fordert er die Daten bei den Harddisks an und sendet diese an den FSM_Master weiter.

SATA SATA ist die Schnittstelle zu den Harddisks, welche mit *Serial ATA* an das System angebunden sind. Die Schnittstelle implementiert den SATA-Standard.

RS232 Interface Das RS232 Interface dient als Schnittstelle zwischen dem LINK und dem Betriebssystem. Es simuliert gegenüber dem Betriebssystem eine RS232 Schnittstelle.

5.2 Error-Handling

Fehler haben verschiedene Ursachen. Das Anliegen dieser Arbeit liegt vor allem im Bereich der Datensicherheit. Die Beschreibung der Fehler und deren Behebung konzentriert sich deshalb hauptsächlich auf zwei Ursachen: Einwirkungen von ausserhalb des Systems oder fehlerhafte physikalische Komponenten. Nicht betrachtet wird u.a. die elektromagnetische Verträglichkeit (EMV). Nebst diesen eigentlichen Fehlern gibt es auch die Situation, dass die Daten abgeglichen werden müssen, obwohl das System fehlerfrei funktioniert. Dies ist zum Beispiel der Fall, wenn zwei Systeme nacheinander eingeschaltet werden und das erste System bereits Daten schreibt. Auch wenn ein System abgeschaltet wird und noch nicht alle Daten aus dem FIFO geschrieben worden sind, ist ein Datenabgleich nötig. Im Systemstatus wird der augenblicklich aktuelle Systemzustand festgehalten.

5.2.1 Mögliche Fehler

Harddisk Sektorfehler werden von der Harddisk transparent behandelt. Eine Harddisk gilt nur als defekt, wenn sie keine Daten mehr entgegennimmt. Dies kann der Fall sein, wenn der

Schreibarm defekt oder das Speichermedium beschädigt ist. Ebenfalls als defekt betrachtet wird eine Harddisk dann, wenn sie überhaupt nicht vorhanden ist. Die Folge eines Harddisks-Ausfalls ist, dass keine Daten mehr gespeichert und gelesen werden können.

Kontroller Der Kontroller ist im FPGA konfiguriert. Fällt das FPGA aus, ist auch das System betroffen.

NVRAM Das RAM ist physikalisch nicht vorhanden oder es ist nicht ansprechbar. Ohne NVRAM fehlt dem System eine grundlegende Komponente, und es kann nicht funktionieren.

Physikalische Defekte Jedes Bauteil kann wegen fehlerhafter Benutzung, Fehlern in der Produktion oder auch wegen physischer Einwirkungen wie zum Beispiel Überhitzung seinen Dienst verweigern. Das Board innerhalb eines Systems ist nur einfach vorhanden. Bei einem Ausfall ist es folglich nicht ersetzbar, und das System ist nicht mehr funktionsfähig.

Software Bei einem Absturz des Betriebssystems oder der Kontroll-Software (Linux-HA) fehlt die ≪oberste≫ Kommunikationsschnittstelle zwischen Benutzer und System.

Link Ist die Verbindung zwischen zwei Systemen unterbrochen, fehlt der Kommunikationspfad zwischen Master und Slave. Es findet kein Datenaustausch (Mirroring) statt und die Daten sind nicht mehr redundant gespeichert.

Obige Fehler führen zu einem Teilausfall oder zu einem kompletten Systemausfall. Sofern das benachbarte System weiter operieren kann, bleiben diese Fehler gegenüber dem Benutzer transparent. Tritt noch ein zusätzlicher Fehler auf, kann dies zu einem kompletten Ausfall des Systems führen.

5.2.2 Systemstatus

Für den Datenabgleich nach einem Fehler im System müssen Informationen vorhanden sein, anhand derer die Daten bereinigt werden können. Mit dem Speichern der Adressen wird der Aufwand für diesen Abgleich minimiert. Dank dem FIFO und dem NVRAM sind die Daten nicht verloren. Deren Korrektheit übernimmt das FIFO bzw. der RAID0 Kontroller, der Daten nur auf bestimmte Kontrollsignale hin aus dem FIFO löscht.
Diese benötigten Angaben werden im System als Systemstatus gespeichert. Der Systemstatus ist ebenfalls im NVRAM abgespeichert. Informationen über den Zustand jeder Harddisk – ist die Harddisk erreichbar oder defekt – und den Zustand des Links sowie die Information, ob ein Full Reconstruction (FC) nötig sein wird, sind im Systemstatus in sechs Bits codiert (Tabelle 5.1, Seite 31). Das erste Bit *LINK* bezeichnet den Zustand des Links. Je zwei Bits sind für die beiden Master-Harddisks (M0 und M1) bzw. die beiden Slave-Harddisks (S0 und S1) gedacht. Das Full Reconstruction Bit (FC) zeigt an, ob nach der Fehlerbehebung alle Daten abgeglichen werden müssen. Die Adress-Spalte (Adr) besagt, ob sich der Systemstatus die Adressen merken muss. In der Daten-Spalte (Data) wird aufgelistet, ob Daten über den Link gesendet werden müssen. Mit diesen Informationen ist es möglich festzustellen, auf welche Art und Weise ein späterer Datenabgleich erfolgen muss.

LINK	M0	M1	S0	S1	FC	Adr	Data	Note
1	1	1	1	1	1	no	yes	alles i.O. (Standard)
1	1	1	1	0	0	no	yes	S1 ausgefallen
1	1	1	0	1	0	no	yes	S0 ausgefallen
1	1	0	1	1	0	no	yes	M1 ausgefallen
1	0	1	1	1	0	no	yes	M0 ausgefallen
0	1	1	1	1	1	yes	no	Link ausgefallen
0	1	1	1	1	0	no	no	Address Overflow
0	0	1	1	1	0	no	no	Doppelfehler, Daten ablehnen
0	1	0	1	1	0	no	no	Doppelfehler, Daten ablehnen

Tabelle 5.1: Mögliche Kombinationen der sechs Bits des Systemstatus

Solange die Verbindung über den Link noch besteht, kann maximal eine Harddisk ausfallen, ohne dass Adressen zwischengespeichert und Daten abgewiesen werden müssen. Sobald der Link ausfällt, müssen die Adressen, nicht aber die zugehörigen Daten gespeichert werden. Doppelte Fehler wie der Ausfall des Links und zugleich der Ausfall einer oder beider Master-Harddisks kann das System nicht auffangen. Die Requests des Betriebssystems werden abgewiesen.

5.2.3 Reaktionen bei Fehlern

Sobald ein Fehler eintritt, muss sichergestellt sein, dass die Daten nicht verloren gehen. Das System merkt sich mit Hilfe des Systemstatus, welche Daten von wo nach wo geschrieben werden müssen. Diese Informationen sind im NVRAM gespeichert. So sind sie auch nach einem längeren (Strom-)Unterbruch nach wie vor verfügbar.

Verbindung zwischen Systemen (Link)
Die Verbindung zwischen zwei Systemen gilt als defekt, wenn sie entweder physikalisch nicht vorhanden ist – d.h. wenn das Kabel fehlt oder wenn das eine System nicht angeschaltet ist. In letzerem Fall ist zwar nicht der Link als solcher defekt, es können aber dennoch keine Daten ausgetauscht werden.
Ursache für einen Verbindungsausfall sind u.a. verzögerte Ein-/Ausschalt-Vorgänge, das vergessen gegangene Einschalten des Slave-Systems oder natürlich ein Fehler in der Hardware. Bei einer fehlerhaften Verbindung merkt sich der Master alle Adressen, die erfolgreich auf die Master-Harddisks geschrieben aber nicht auf die Slave-Harddisks dupliziert werden konnten. Er speichert die Adressen in den reservierten Speicherplatz im NVRAM.
Sobald der für die Adress-Speicherung reservierte Speicherplatz voll ist und nach wie vor Daten gespeichert werden sollen, merkt sich der Systemstatus die betroffenenen Adressen nicht mehr. In diesem Fall wird – sobald die Verbindung wieder besteht – ein Full Reconstruction durchgeführt (siehe Abschnitt 5.3.2, Seite 34).

Ansonsten werden die Daten anhand der gespeicherten Adressen abgeglichen. Dabei liest der Kontroller unabhängig vom Betriebssystem die Daten von derjenigen Harddisk, auf der die Daten erfolgreich gespeichert wurden und schreibt sie in das FIFO desjenigen Systemteils, welcher vom Fehler betroffen war.

Harddisk-Ausfall oder Harddisk-Fehler
Wenn eine Harddisk einen Fehler meldet, ist sie nicht mehr erreichbar, und die Daten werden nicht mehr geschrieben bzw. der RAID0-Kontroller sendet der defekten Harddisk keine Daten mehr. Es gibt aber auch Fehler, welche die Harddisk weitgehendst selbst behebt. Zum Beispiel führt die Harddisk eine interne Mapping-Tabelle, welche bei einem Sektorfehler festhält, auf welchem ≪Ersatzsektor≫ die Daten geschrieben worden sind.
Der Systemstatus merkt sich die defekte Harddisk und speichert sie im Systemstatus. So ist nach der Behebung des Fehlers bekannt, von welcher Harddisk auf welche Harddisk die Daten abgeglichen werden müssen. Bei einem Harddisk-Fehler lohnt es sich nicht, die Adressen zu speichern. Zu gross ist die Unsicherheit oder das Unwissen, ob die gleiche Harddisk wieder angeschlossen wurde. Viel grösser ist die Wahrscheinlichkeit, dass die Harddisk erneuert wurde und somit ein *Full Reconstruction* nötig sein wird.

Überfülltes FIFO
Solange das FIFO voll mit Daten ist, werden neu ankommende Schreibanfragen nicht mehr angenommen. Das Betriebssystem muss die Anfrage erneut starten. Erst wenn wieder ein vollständiger Block Platz hat – der RAID0-Kontroller löscht nur ganze Blöcke auf einmal – wird die erste Sequenz dieses Blocks entgegengenommen. Ursache für diesen Fehler kann zum Beispiel sein, dass der RAID0-Kontroller die Daten zu langsam verarbeitet, oder dass das FIFO zu klein berechnet worden ist und die Daten zu schnell geschrieben werden.

Doppelte Fehler
Einzelne Fehler entdeckt das System selbstständig und reagiert automatisch mit oben erläuterten Massnahmen. Doppelfehler wie zum Beispiel der Ausfall des Links und der gleichzeitige Ausfall einer Master-Harddisk führen im Normalfall zu einem totalen Systemausfall, da weder Daten geschrieben noch gelesen werden können.
Es muss mindestens eine Speichereinheit – entweder der Master oder der Slave – verfügbar sein, damit Requests beantwortet werden können. Gewisse Doppelfehler allerdings dürfen vorkommen, ohne dass das ganze System abstürzt. Ist zum Beispiel eine Harddisk des Slave defekt und fällt zugleich der Link aus, werden die Adressen im NVRAM gespeichert. Allerdings ist man nicht sicher, ob nicht ein Full Reconstruction auf die ausgefallene Harddisk nötig wäre. Das System ist *Single Failure Tolerant*, garantiert aber keine Redundanz bei jeglichen Doppelfehlern.

Unterschiedliche Datenspeicherung
Das Systemdesign garantiert mit Hilfe von Systemstatus und Reconstruction, dass die gespiegelten Daten exakt den Originaldaten auf der Master-Harddisk entsprechen. Zwischenzeitlich besteht bis zum Reconstruction eine Daten-Inkonsistenz. Sobald aber alle Daten abgeglichen sind, kommt es nicht vor, dass auf der einen Harddisk mehr Daten gespeichert sind als auf der anderen. Master und Slave sind nie ungleich mit Daten gefüllt.

5.3 Der Reconstruction-Vorgang

5.3.1 Error Recovery

Bei Systemen, welche hohe Performance und Datensicherheit garantieren, ist das schnelle Entdecken von Fehlern während des online-Betriebs wichtig. Solche Systeme verlangen aber nicht nur die Möglichkeit, Fehler ohne Datenverlust zu erkennen, sondern auch eine Wiederherstellung der Daten, ohne dass das System abgeschaltet werden muss. Es sollte schnell in einen fehlerfreien Zustand gebracht werden und dabei keine Auswirkungen auf die System-Performance haben. [10]
In der fehlerfreien Situation befindet sich das System im *normalen Betriebszustand*. Nach dem Auftreten eines einzelnen Fehlers ist das System im *verschlechterten Zustand*, und nur wenn zwei oder mehrere Fehler existieren, befindet sich das System im *Betriebszustand bei komplettem Ausfall*.
Der traditionelle Ansatz für die Fehlerbehebung ist der direkte Übergang vom ≪Error State≫ in den Zustand, der den Fehler behebt. Diese Methode ist auch bekannt als *Forward Error Recovery*. Sie braucht ein hohe Anzahl von Fehler-Zuständen, die selektiv Korrekturen anbringen. Dazu werden genaue Informationen über den Fehler benötigt, da die Auswirkungen eines Fehlers abhängig sind vom Kontext, in welchem der Fehler aufgetreten ist. Um die Komplexität der Fehlerzustände zu verkleinern, muss deren Anzahl – zum Beispiel durch Gruppierung ähnlicher Fehler – minimiert werden.

Typ	**Bemerkungen**
Forward Error Recovery	- Skaliert nicht bei zunehmender Komplexität - Benötigt hohe Anzahl von Fehler-Zuständen - Genaue Informationen über Fehlertyp benötigt - Vom System abhängig
Backward Error Recovery	- Ressourcen-intensiv - Wiederholung von Arbeits-Vorgängen - Recovery Points müssen gespeichert werden - Unabhängig vom Kontext des Fehlers

Tabelle 5.2: Übersicht über *Forward Error Recovery* und *Backward Error Recovery*

Ein zweiter Ansatz ist das *Backward Error Recovery*. Wenn ein Fehler auftritt, setzt diese Methode das System in einen fehlerfreien Zustand zurück und versucht die fehlerverursachenden Operationen nochmals durchzuführen. Dazu sind sogenannte *Recovery Points*, bei denen der augenblickliche Systemzustand als Teil des normalen Ablaufs gespeichert wird, nötig. Allerdings muss die Arbeit, welche zwischen dem letzten *Recovery Point* und dem Fehler gemacht wurde, wiederholt werden. Die Methode ist einerseits unabhängig vom Kontext des Fehlers, andererseits aber braucht sie viele Ressourcen und Performance (Tabelle 5.2). [11]
Das Systemdesign sieht einen Weg zwischen *Forward Error Recovery* und *Backward Error Recovery* vor. Einerseits wird der Systemstatus immer aktuell gehalten, andererseits werden die Daten nach dem Auftreten eines Fehlers gemerkt und später automatisch abgeglichen. Es ergibt sich dadurch auch nach der Entdeckung des Fehlers keinen Datenverlust.

5.3.2 Reconstruction-Algorithmen

Nach einem Fehler erfolgt ein Ablauf, welcher die Daten abgleicht und somit Master und Slave auf den gleichen Stand bringt. Dieser Ablauf wird auch *Reconstruction* genannt. Es gibt zwei mögliche Ansätze für die Umsetzung: eine stripe-orientierte und eine disk-orientierte Rekonstruktion. Die erste Variante hat eine schlechte Ausnützung der Performance. Die zweite verwendet die ganze Harddisk-Bandbreite, welche nicht von Benutzer-Prozessen gebraucht wird. Anstelle von einem einzigen Stripe werden viele Stripes derselben Harddisk parallel behandelt.
Im Systemdesign erfolgt ein Reconstruction in zwei unterschiedlichen Fällen. Zum einen werden die Adressen im NVRAM gespeichert (Link-Ausfall) und es ist ein *Normal Reconstruction* nötig. Zum anderen wird bei einem Harddisk-Ausfall ein *Full Reconstruction* notwendig. Der Unterschied besteht darin, dass die Daten entweder anhand der Adressen von der gesunden Harddisk gelesen werden und somit keinen Overhead an Arbeit erfordern. Oder es werden sämtliche Daten von der einen Harddisk auf die andere Harddisk kopiert (identische Kopie). Dies braucht allerdings viel Zeit und blockiert das System. [12]
Der Reconstruction-Vorgang wird vom Master-Kontroller ausgelöst, sobald er anhand des Systemzustands die Fehlerbehebung erkannt hat. Der Master-Kontroller schickt Leseanfragen an die gesunden Harddisks und schreibt die Daten in das FIFO des vom Fehler vefreiten Teilsystems. Der RAID0-Kontroller erkennt nicht, ob die Daten im FIFO von einem Reconstruction-Prozess herrühren oder ob sie einer normalen Schreibanfrage entspringen. Auch bei Leseanfragen liest er anhand der Adresse die Daten, ob sie für einen Reconstruction-Ablauf bestimmt sind oder nicht. Da im FIFO nur ganze Datenblöcke gespeichert sind (Abschnitt 6.4.2, Seite 45), ist es nicht notwendig, das FIFO vor einem Reconstruction-Vorgang zu leeren. Es werden höchstens zwei Mal die gleichen Daten geschrieben (überschrieben).

5.4 Systeminitialisierung und Systemreset

Bei jedem Systemstart werden bestimmte System-Komponenten getestet. Anhand dieser Tests wird entschieden, in welchem Zustand sich das System befindet. Wenn es sich in keinem Fehler-Status befindet, gilt es anhand der gesammelten Informationen den aktuellen Systemstatus zu speichern (Tabelle 5.3).
Für den Start notwendige Vorgaben sind das physikalische Vorhandensein des NVRAM, des RAM für das OS und des Flash sowie deren korrekte Ansprechbarkeit. Ist kein NVRAM vorhanden, ist auch der Systemstatus – der zuletzt vor dem Abschalten gespeichert wurde – nicht mehr vorhanden. Man kann folglich nicht mehr evaluieren, welche Daten und ob überhaupt noch Daten abgeglichen werden müssen. Ebenfalls fehlt der Speicherort der Adressen, die bei einem Ausfall des Links oder einer Harddisk in das NVRAM geschrieben werden. Auch das für Schreibanfragen verwendete FIFO ist in diesem NVRAM implementiert.
Im Flash ist das Bitfile[1] gespeichert, welches die Konfiguration des FPGAs und das Betriebssystems enthält. Da das FPGA eine stete Spannungsversorgung für das Speichern der Konfiguration braucht, muss es nach jedem (gewollten) Stromunterbruch neu konfiguriert werden. Mit dieser Konfiguration werden die Kontroller in Hardware abgebildet. Nebst diesen Tests, ob die Bausteine vorhanden sind, kontrolliert das System auch das Vorhandensein des Nachbarn. Dazu muss zuerst der Link zwischen den Systemen existieren.
Sind alle Komponenten vorhanden, wird der Systemstatus aus dem NVRAM gelesen. Er steht nun in Form von Signalen zur Verfügung.
Falls ein Kontroller in einem Zustand verweilt, in welchem er auf ein Signal eines anderen Kontrollers wartet (z.B. Bestätigung, die Daten erhalten zu haben), dieses Signal aber nicht bekommt, ist der Kontroller in besagtem Zustand blockiert. Somit ist aber auch das System blockiert. Im Normalfall sollte diese Situation nicht auftreten – falls aber doch, kann durch einen Reset-Knopf das ganze System auf den Initialzustand zurückgesetzt werden. Allenfalls gehen dabei Daten verloren, welche den Ausnahmefall verursacht haben.

Komponente	**Bemerkungen**
NVRAM	- Zuletzt gespeicherter Systemstatus nicht mehr vorhanden - Speicherort für Adressen fehlt
Flash	- Bitfile für FPGA-Konfiguration fehlt - Betriebssystem kann nicht starten
Nachbar-System	- Spiegeln der Dateien nicht möglich - Kein Datenabgleich

Tabelle 5.3: Bei einer System-Initialisierung durchgeführte Tests

[1] Das Bitfile wird in das FPGA geladen. Dadurch wird bestimmt, wie die Logic Blocks verbunden werden und welche Aufgaben sie zu übernehmen haben.

Kapitel 6

Detailbeschreibungen der entwickelten Funktionsblöcke

6.1 Einführung

Einschränkungen

In diesem Kapitel werden die einzelnen Funktionsblöcke des Systems genauer definiert und beschrieben. Aus zeitlichen und organisatorischen Gründen beschränken wir uns in dieser Arbeit allerdings auf diejenigen Funktionsblöcke, welche für das Erreichen der Ziele der Arbeit nötig sind. Die Planung des Projekts verlangt ein Design, welches spätere Erweiterungen zulässt.

Allgemeines

Für jede Verbindung zwischen den verschiedenen Funktionsblöcken ist eine eigene Schnittstelle definiert. Zur besseren Übersicht sind die Signale jeweils mit dem Namen des Ziel-Blockes versehen.
Die Funktionsblöcke werden in der Reihenfolge beschrieben, wie sie im Sysemdesign erscheinen.

6.2 Host Interface

Das Host Interface dient als Schnittstelle zwischen dem Betriebssystem und dem Kontroller. Setzt ein Prozess eine Anfrage an ein Gerät, wird dessen Treiber gestartet. Der Gerätetreiber kennt den momentanen Zustand ≪seines≫ Gerätes und kann mit ihm kommunizieren. Der Code für diesen Gerätetreiber ist im Betriebssystem integeriert.
Im Fall des in dieser Arbeit benutzten FPGAs existiert ein *On-Chip Peripheral Bus* (OPB). Dieser dient der Anbindung von verschiedenen Geräten wie der Ethernet- oder der UART-Schnittstelle. Die Firma Xilinx stellt zur Anbindung von Intellectual Property Cores (IP Cores) an den OPB ein definiertes Modul zur Verfügung. Dieses Modul, auch IPIF (Intellectual Property Interface) genannt, bietet standardisierte Funktionen und Services wie FIFOs, DMA, automated DMA, Software Reset, Interrupt Support und Bus-Master Zugriff an.

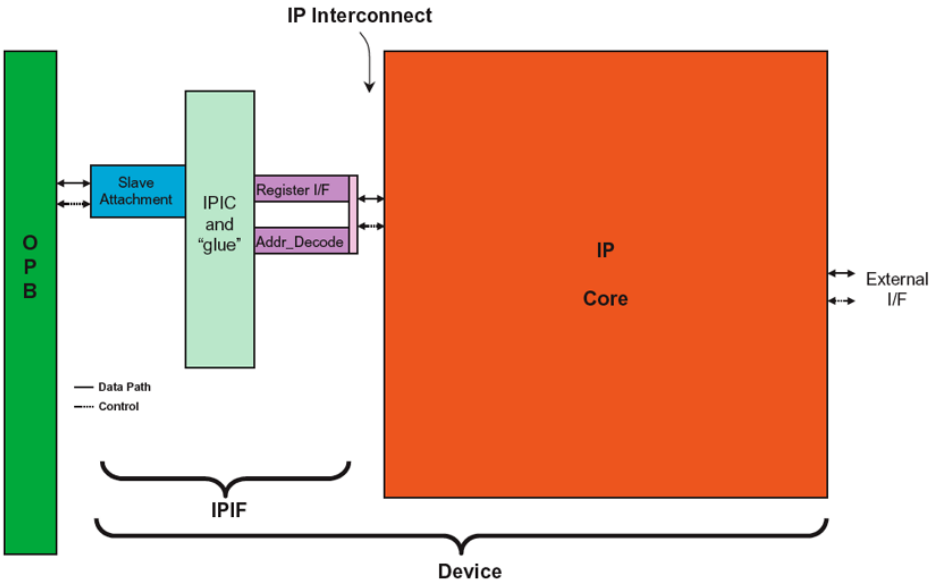


Abbildung 6.1: Das OPB IPIF Modul ist zwischen dem OPB und dem IP Core positioniert.

Über die OPB-Schnittstelle empfängt und sendet das Host-Interface, welches als IP Core mit dem IPIF verbunden ist, Daten. Allenfalls müssen die Daten gebuffert werden. Das Host-Interface darf Daten erst weitergeben, wenn die vorhergehenden Daten vom FSM_Master oder vom Gerätetreiber korrekt bestätigt worden sind.

6.3 Master Kontroller

Der FSM_Master ist der Kontroller, welcher den Kontroll- und Datenfluss innerhalb des Systems steuert. Er verwaltet die Error-Meldungen, aufgrund derer er jeweils den Systemzustand speichert und erneuert. Bei einem Systemstart wird als Erstes eine Initialiserungsphase durchlaufen, welche einerseits bestimmte Komponenten überprüft und das Vorhandensein des zugeörigen System-Nachbars kontrolliert, und andererseits den gespeicherten Systemstatus in die lokalen Register lädt. Der Master ist ausserdem dafür verantwortlich, dass jeweils ein ganzer 512Byte Block im FIFO gespeichert ist, auch wenn die Daten später verworfen werden.
Die Zustandsmaschine besteht somit aus mehreren Zyklen: Read-, Write-, Initialisierungs- und Reconstruction-Zyklus sowie dem Error-Handling (Abbildung 6.2). In den folgenden Kapiteln werden nur die beiden Zyklen *Read* und *Write* betrachtet und implementiert.

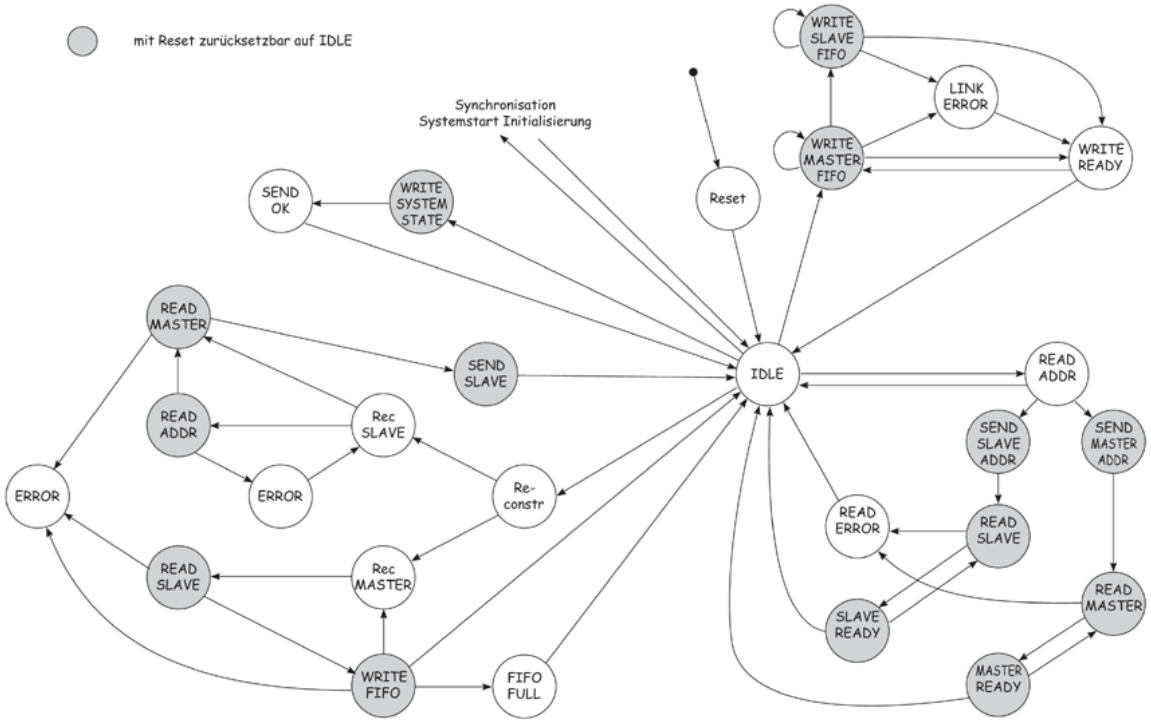


Abbildung 6.2: Finite State Machine für den Master-Controller (RAID1)

6.3.1 Systemstatus

Der Systemstatus (Abschnitt 5.2.2, Seite 30) wird bei jeder Adresse, die es abzugleichen gilt, mitgespeichert. Die Adresse ist 48 Bit lang und in 64 Bit abgespeichert. D.h. es bleiben 16 Bit übrig, von welchen der Systemstatus nun fünf Bits verwendet.
In der Implementation sind diese fünf Bits der Übersichtlichkeit zuliebe auf zwei Signale *StatusHDxDP/DN* und *StatusLinkxDP/DN* aufgeteilt. Das erste Signal ist ein Bitvektor von vier Bit Breite, jeder Eintrag entspricht einer Harddisk. Das zweite Signal bezeichnet den Zustand des Links. Allgemein gilt:

- 0 bedeutet, Element ist nicht in Ordnung
- 1 bedeutet, Element ist in Ordnung

6.3.2 Definierte Zustände im Kontroller

IDLE ist der Basiszustand. Sobald das Host-Interface Daten bereitstellt, wird anhand der zusätzlich angelegten Signale entschieden, in welchen Zyklus – Read, Write oder Reconstruction – die Zustandsmaschine wechseln soll. Ist weder die Master-Harddisks noch die Slave-Harddisks erreichbar, wird direkt eine Fehlermeldung an das Host-Interface zurückgegeben. Es gibt keinen direkten Wechsel von READ_MASTER zu READ_SLAVE. Der Read-Request muss neu gestellt werden.

RESET Bei jedem Systemstart wird vor dem IDLE-Zustand der Systemstatus auf die Standardwerte zurückgesetzt (5.2.2 Seite 30). Dies ist der Default-Zustand, wenn das System eingeschaltet oder einem Reset unterzogen wird.

WRITE_FIFO_MASTER Die Daten werden solange dem FIFO bereitgestellt, bis sie von diesem erfolgreich geschrieben und bestätigt worden sind. Meldet das FIFO, dass es keinen freien Platz mehr hat, wartet die FSM, bis das FIFO wieder Daten aufnehmen kann. Falls die Verbindung zum Nachbarsystem im Systemstatus als fehlerhaft bezeichnet ist, werden die Daten sogleich an den lokalen RAID0-Kontroller weitergereicht. Falls die Verbindung soeben ausgefallen ist, muss die Adresse, welche in den ersten beiden Durchgängen in lokale Register geladen wurde, im NVRAM abgelegt werden.

WRITE_FIFO_SLAVE Sofern der Link existiert, liegen an dessen Datenleitung (Bus) die Daten an. Nach korrekter Bestätigung der Daten erfolgt eine erneute Abfrage nach Daten beim Host-Interface. Ein Fehler des Slave (Link down oder Harddisk(s) nicht erreichbar) fordert den Übergang in LINK_ERROR.

LINK_ERROR Falls der Link unterbrochen ist, merkt sich das System die Adressen der zu schreibenden Daten. Der Linkstatus wird nachgeführt. Ein Reconstruction wird dann angezeigt, wenn der für die Adressen-Zwischenspeicherung reservierte Bereich im NVRAM überfüllt ist.

WRITE_READY Nach jedem vollständigen Schreiben eines Datenblocks (512 Bytes) beginnt der Zyklus via IDLE von vorne. Die erfolgreiche Datenverarbeitung wird dem Host-Interface bestätigt, und sofern noch Daten vom Host-Interface anliegen, werden diese im Datenregister gespeichert.

READ_ADDRESS In diesem Zustand wird die Adresse ins lokale Adressregister geschrieben. Anhand des Systemstatus' wird entschieden, ob der Read-Request an den Master oder an den Slave weitergeleitet wird.

SEND_MASTER_ADDRESS Die Adresse wird an den lokalen RAID0-Kontroller des Masters gesendet. Der RAID0-Kontroller muss die Adresse bestätigen, bevor der Zustand verlassen werden kann.

READ_MASTER Falls beide Harddisks als erreichbar markiert sind, verarbeitet der Master die Daten, welche an der Schnittstelle zum lokalen RAID0-Kontroller anliegen. Ansonsten wechselt die FSM in den READ_ERROR Zustand.

Present State	Signale		Next State
IDLE	StatusLinkxDP = 0 AND StatusHDxDP(1,0) <11	->	IDLE
	HI_ReadDataxE = 1	->	WRITE_FIFO_MASTER
	HI_WriteDatexACK = 1	->	READ_ADDRESS
WRITE_FIFO_MASTER	FM_IsFull = 1	->	WRITE_FIFO_MASTER
	FM_WriteDataxACK = 1 AND FM_WriteDataxERR = 0 AND StatusLinkxDP = 1	->	WRITE_FIFO_SLAVE
	FM_WriteDataxACK = 1 AND FM_WriteDataxERR = 0 AND StatusLinkxDP = 0 AND OffsetxDN = 1 AND StatusHDxDP(3,2) >00	->	LINK_ERROR
	FM_WriteDataxACK = 1 AND FM_WriteDataxERR = 0 AND StatusLinkxDP = 0 AND OffsetxDN != 1 AND StatusHDxDP(3,2) <= 00	->	WRITE_READY
WRITE_FIFO_SLAVE	Link_WriteDataxACK = 1	->	WRITE_READY
	Link_WriteDataxERR = 1	->	LINK_ERROR
WRITE_READY	OffsetxDN = 130	->	IDLE
	HI_ReadDataxE = 1	->	WRITE_FIFO_MASTER
LINK_ERROR	always	->	WRITE_READY

Tabelle 6.1: Alle Zustände und Übergangssignale, die für den Ablauf innerhalb des Write-Zyklus relevant sind

READ_MASTER_READY Die Daten stehen dem Host-Interface zur Verfügung. Hat das Host-Interface die Entgegennahme der Daten bestätigt und keinen Fehler zurückgegeben, wird das nächste Datenpaket mit READ_MASTER entgegengenommen. Gleichzeitig erfolgt gegenüber dem RAID0-Kontroller eine Bestätigung, damit die nächsten Daten bereitgestellt werden können.

SEND_SLAVE_ADDRESS Die Adresse wird über den Link an RAID0-Kontroller des Slave gesendet.

READ_SLAVE Falls beide Harddisks als erreichbar markiert sind, verarbeitet der Master die Daten, welche über den Link anliegen. Ansonsten wechselt die FSM in den READ_ERROR Zustand.

READ_SLAVE_READY Die Daten stehen dem FSM_Master zur Verfügung. Hat das Host-Interface die Entgegennahme der Daten bestätigt und keinen Fehler zurückgegeben, wird das nächste Datenpaket mit READ_SLAVE entgegengenommen. Gleichzeitig erfolgt gegenüber dem RAID0-Kontroller eine Bestätigung, damit die nächsten Daten bereitgestellt werden können.

READ_ERROR In diesem Zustand wird ein ERR (Error) an das Host-Interface gesendet. Die Daten konnten nicht erfolgreich gelesen werden. Requests werden abgelehnt, falls sowohl Master- als auch Slave-Harddisk nicht erreichbar sind.

6.3.3 Intern verwendete Register

Damit verschiedene Signale, welche intern zwischen verschiedenen Status verwendet werden, temporär zwischengespeichert sind, werden weitere Register benötigt (Abschnitt 5.2.2, Seite 30).

OffsetxDP, OffsetxDN ist als Counter gedacht, der die Anzahl von 32 Bit Datenpaketen während eines Read- oder Write-Request zählt (Datenblock-Offset). Es ist ein 8 Bit Register.

Address1xDP, Address1xDN beinhaltet den ersten Teil der 48 Bit Adresse eines Read- oder Write-Requests. Es ist ein 32 Bit breites Register.

Address2xDP, Address2xDN beinhaltet den zweiten Teil der 48 Bit Adresse eines Read- oder Write-Requests. Es ist ein 32 Bit breites Register.

DataxDP, DataxDN ist für das Zwischenspeichern von einzelnen 32 Bit Datenblöcken pro Clock-Cycle gedacht und hat eine Breite von 32 Bit.

StatusHDxDP, StatusHDxDN speichert den Zustand der Harddisks. Pro Harddisk bezeichnet ein Bit deren Funktionstüchtigkeit. Dieses Register ein Teil des Systemstatus (Abschnitt 6.3.1, Seite 6.3.1).

StatusLinkxDP, StatusLinkxDN speichert als 1 Bit breites Signal den Zustand des Links (Abschnitt 6.3.1, Seite 6.3.1).

Present State	Signale		Next State
READ_ADDRESS	OffsetxDP <2	->	READ_ADDRESS
	OffsetxDP = 2 AND StatusHDxD(1,0) = 11	->	SEND_MASTER _ADDRESS
	OffsetxDP = 2 AND StatusHDxD(3,2) = 11 AND StatusLinkxDP = 1	->	SEND_SLAVE _ADDRESS
SEND_MASTER _ADDRESS	OffsetxDP = 0	->	SEND_MASTER _ADDRESS
	OffsetxDP = 1 AND Raid_WriteDataxACK = 1 AND Raid_WriteDataxERR = 0	->	READ_MASTER
READ_MASTER	StatusHDxDP(1,0) <11	->	READ_ERROR
	Raid_ReadDataxE = 1	->	READ_MASTER _READY
READ_MASTER _READY	HI_WriteDataxACK = 1 AND HI_WriteDataxERR = 0 AND OffsetxDP = 128	->	IDLE
	HI_WriteDataxACK = 1 AND HI_WriteDataxERR = 0 AND OffsetxDP <128	->	READ_MASTER
	HI_WriteDataxACK = 0	->	IDLE
SEND_SLAVE _ADDRESS	OffsetxDP = 0 Link_WriteDataxERR = 1	->	READ_ERROR
	OffsetxDP = 1 AND Link_WriteDataxACK = 1	->	READ_SLAVE
	OffsetxDP = 1 AND Link_WriteDataxERR = 1	->	READ_ERROR
READ_SLAVE	StatusHDxDP(3,2) <11	->	READ_ERROR
	Link_ReadDataxE = 1	->	READ_SLAVE _READY
READ_SLAVE _READY	Link_WriteDataxACK = 1 AND Link_WriteDataxERR = 0 AND OffsetxDP = 128	->	IDLE
	Link_WriteDataxACK = 1 AND Link_WriteDataxERR = 0 AND OffsetxDP <128	->	READ_SLAVE
READ_ERROR	always	->	IDLE

Tabelle 6.2: Alle Zustände und Übergangssignale, die für den Ablauf innerhalb des Read-Zyklus relevant sind

6.4 Aufbau des FIFO

Ein FIFO nimmt Daten geordnet entgegen, speichert sie und gibt sie auf Verlangen wieder in der gleichen Reihenfolge aus, in der sie eingetroffen sind (*First In, First Out*). Es ist eine Speicherstruktur, welche in einem RAM implementiert ist.
Da die Zugriffszeiten auf ein RAM mit Faktor 1000 viel kleiner sind gegenüber denjenigen auf eine Harddisk, werden die Write-Requests in das FIFO gespeichert. Sobald die Daten eines Blocks vollständig im RAM gespeichert sind, kann die Bestätigung an das Betriebssystem erfolgen. Obwohl die Daten noch nicht auf der Harddisk gespeichert sind, sind sie bereits bestätigt, und das Betriebssystem kann weiter operieren.
Damit nun bei einem Stromunterbruch die Daten nicht aus dem RAM gelöscht werden, muss ein ≪nicht flüchtiges≫ (non volatile) RAM benutzt werden. Der Vorteil ist die schnelle Antwortzeit und die eingehaltene Ordnung der Daten. Nachteile sind vor allem der Bedarf an (genügend) Speicherplatz und die Handhabung des FIFOs.
Das FIFO wird vom FSM_Master angesprochen, welcher die Daten schreibt, und vom RAID0-Kontroller, welcher die Daten aus dem FIFO liest (Abbildung 6.3).

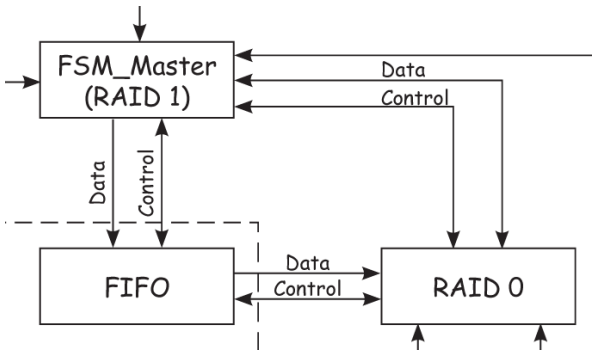


Abbildung 6.3: Das FIFO ist zwischen Master- und RAID0-Controller eingefügt

6.4.1 Internes FIFO des Virtex II Pro

Der Virtex II Pro von Xilinx bietet bereits internes Block-RAM an. Dieses RAM wird allerdings für verschiedene Zwecke (System, Kombinatorik, Speicher usw.) verwendet.
Für die Implementation eines umfangreichen FIFO reicht dieser Speicherplatz nicht aus. Aus zeitlichen Gründen wird es in dieser Arbeit dennoch verwendet. Es dient lediglich Testzwecken und soll die Funktionalität eines FIFO für die Kommunikation mit den benachbarten Komponenten erfüllen. Xilinx stellt ein ≪IP Core[1]≫ für ein FIFO zur Verfügung [7].

[1] *IP cores* (Intellectual Property) stellen fertig entworfene, bereits optimierte und in sich abgeschlossene Codestrukturen zur Verwendung zur Verfügung.

Dieses FIFO bietet als Grundeinstellung die Input-Leitungen *DIN (Data In)*, *WR_EN (Write Enable)* bzw. *RD_EN (Read Enable)*, sowie die Output-Leitungen *DOUT (Data Out)*, *FULL* und *EMPTY* an und ist durch ein Clocksignal *CLK* getaktet. Optional können noch zusätzliche Kontrollsignale implementiert werden (Abbildung 6.4).

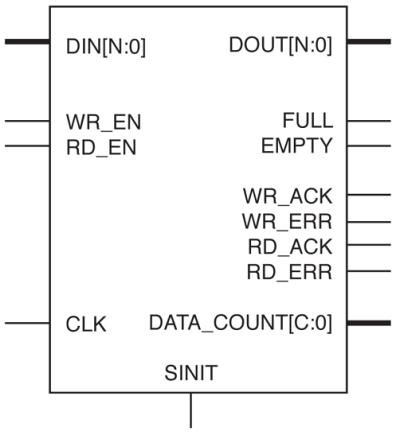


Abbildung 6.4: Pinbelegung des internen FIFO

6.4.2 Eigene Implemenation

Zu einem späteren Zeitpunkt – vorgesehen in der nachfolgenden Diplomarbeit – soll das FIFO allerdings nicht mehr mit internem Block-RAM realisiert werden, sondern in einem externen, grösseren RAM. Dadurch können zusätzliche Funktionalitäten implementiert und weitere Optimierungen vorgenommen werden.
Es wird allerdings eine eigene FSM für die Steuerung und Kontrolle nötig sein. Das FIFO hat in unserem Fall eine Breite von 32 Bit, so dass pro Takt jeweils 32 Bit bearbeitet werden können[2]. Ein besonderes Augenmerk verdient das Block-Handling. Ein logisch zusammenhängender Block ist mindestens 512 Byte gross. Dies bedeutet 130 (inklusive 64 Bit für die Adresse) kleine 32 Bit-Blöcke im FIFO für den gleichen logischen Block.

Inkonsistenz und Datenmüll im FIFO
Sind der Master und der RAID0-Kontroller gleichzeitig und unabhängig voneinander am Daten schreiben bzw. lesen, tritt bei einem Unterbruch des Datenflusses (Stromausfall, Link Down, ...) die Situation auf, dass im FIFO noch 32-Bit-Blöcke vorhanden sind, die keinem logischen Block mehr zugeordnet werden können (Abbildung 6.5, Seite 46).
Lösungsansätze sind zum Beispiel das Mitführen eines Bits, welches zu Beginn (beim FIFO-Block 1) der 130 Einträge und am Ende (beim FIFO-Block 130) auf Eins gesetzt ist (Erkennen von Anfang und Ende). Auch das Modulo-Rechnen durch die Blockgrösse kann per Zufall aufgehen, wenn sich die beiden unvollständigen Bereiche zu einem vermeintlich korrekten 512 Byte Block ergänzen.

[2] Zwischen den Hardware-Komponenten im FPGA sind 32 Bit-Busse für Daten und Adressen vorgesehen.

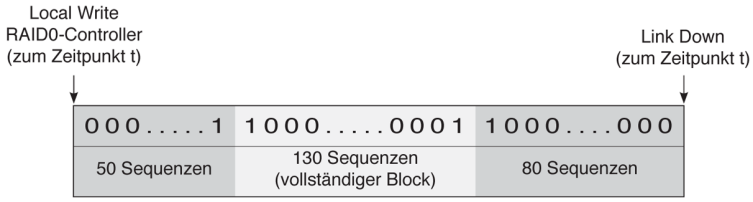


Abbildung 6.5: Situation im FIFO bei einem Unterbruch im Datenfluss

Allgemein ergibt sich das Problem, dass zwar Anfang und Ende sowie die Länge erkannt werden können, allerdings weiss man nicht mehr, an welche Adresse die Daten gespeichert werden müssen, da die Adresse (entspricht den ersten beiden FIFO-Eintragen 1 und 2) bereits gelesen und gelöscht wurde. Als mögliche Lösung dürfen die 130 FIFO-Einträge und somit der ganze logische Block erst gelöscht werden, wenn alle Blöcke gespeichert und von der Festplatte bestätigt worden sind. Dies erfordert allerdings zusätzlichen Implementationsaufwand und braucht mehr Abarbeitungszeit.

6.5 Der RAID0 Kontroller

Der RAID0-Kontroller hat die einfache Aufgabe, Read- und Write-Requests an die richtige Festplatte zu leiten. Der Kontroller befindet sich per Default im IDLE Zustand. Dieser Zustand wird erst aufgrund von Signalen verlassen, die entweder der Master (RAID1) Kontroller oder das FIFO setzen.
Ein Write-Request wird unabhängig vom Master Kontroller abgearbeitet. Der RAID0-Kontroller nimmt die Daten aus dem FIFO und sendet sie an die SATA-Schnittstellen weiter. Er geht davon aus, dass er – sobald ein Request behandelt wird – jeweils einen ganzen Block abarbeiten kann. Es werden keine Vorsichtsmassnahmen hinsichtlich einer Kontrolle getroffen.
Bei einem Read-Request hingegen wird eine direkte Kommunikation mit dem Master-Kontroller unumgänglich, da er von diesem die Adresse benötigt. Write- und Read-Requests unterbrechen sich nicht gegenseitig. Es muss jeweils ein vollständiger 512 Byte Block (in mehreren Zyklen) verarbeitet werden. Die beiden Abläufe sind in sich abgeschlossen.

6.5.1 Error-Handling

Falls ein Fehler beim Lesen oder Schreiben einer Harddisk auftritt, wird dies intern mit Hilfe eines Registers gespeichert und gleichzeitig dem Master-Controller gemeldet. Ein Write-Request wird weiter ausgeführt. Die Daten, welche für die defekte Disk bestimmt sind, werden verworfen. Sie sind schliesslich noch im Mirror vorhanden.

Strom-Unterbruch während eines Requests
Ein 512 Byte Block wird in mehrere Sequenzen von 32 Bit Länge aufgeteilt. Für die Übersicht und Kontrolle dieser Sequenzen dient ein Counter. Tritt nun während eines Write-Requests, bei welchem diese Sequenzen in einem FIFO gelagert werden, ein Stromunterbruch auf, kennt man bei einem Neustart den letzten Zustand des Counters nicht mehr. Somit liegen im FIFO u.U. Datenleichen, und die Abgrenzung von grossen Datenblöcken ist nicht mehr möglich. Abhilfe könnten folgende Erweiterungen schaffen:

- Der Counter muss an einem Ort gespeichert werden, welcher nicht flüchtig ist. In Frage käme wohl der non-volatile Speicher.
- Im FIFO wird der Counter mitabgespeichert. Bei einem 32 Bit breiten FIFO wirken zusätzliche acht Bit für den Counter allerdings als zu grossen Overhead.
- Für jeden neuen 512 Byte Block – d.h. bei dessen erster 32 Bit Sequenz – wird ein Flag eingefügt. Ob aber mit Bit-Stuffing eine Gleichheit mit Datensequenzen in geeigneter Performance erreicht wird, ist zu überprüfen.
- Die übriggebliebenen Sequenzen sollen per Zählen der totalen Anzahl Blöcke im FIFO erkannt werden. Nur ergibt sich hier die Problematik, dass das Zählen kein brauchbares Resultat liefert, falls beim Unterbruch der RAID-Master gerade am Daten ins FIFO schreiben war.
- Eine etwas aufwändigere Möglichkeit wäre eine ununterbrechbare Stromversorgung, die ein korrektes Beenden eines Write-Requests sicherstellen würde.

Allerdings beschränken sich die Ausführungen und die Implementation in diesem Dokument auf den Fall, dass dieser Fehler nicht vorkommt. Ein komplexeres Error-Handling wird in der nachfolgenden Diplomarbeit betrachtet.

6.5.2 Striping-Algorithmus

Beim Striping sind zwei Harddisks angeschlossen. Der (lineare) Speicherbereich wird im Master-Kontroller verwaltet. Er ist doppelt so gross wie der ebenfalls lineare Speicherbereich einer einzigen Harddisk. Damit diese beiden Harddisks als ein grosser Speicherbereich betrachtet werden können, braucht es einen Algorithmus, welcher die Adresse des Requests erfasst und zu einer neuen Adresse verarbeitet, die dann auf einer Harddisk angesprochen werden kann (siehe Abbildung 6.6).
Die Stripegrösse ist in einer Zweier-Potenz angegeben. Anhand eines einzigen Bits der Original-Speicheradresse wird entschieden, auf welche Harddisk dieser Block gespeichert wird. Die Blöcke werden also abwechslungsweise auf die eine bzw. die andere Harddisk geschrieben.

- Harddisk 0: Blöcke, die an der genannten Stelle eine Null haben
- Harddisk 1: Blöcke, die an der genannten Stelle eine Eins haben

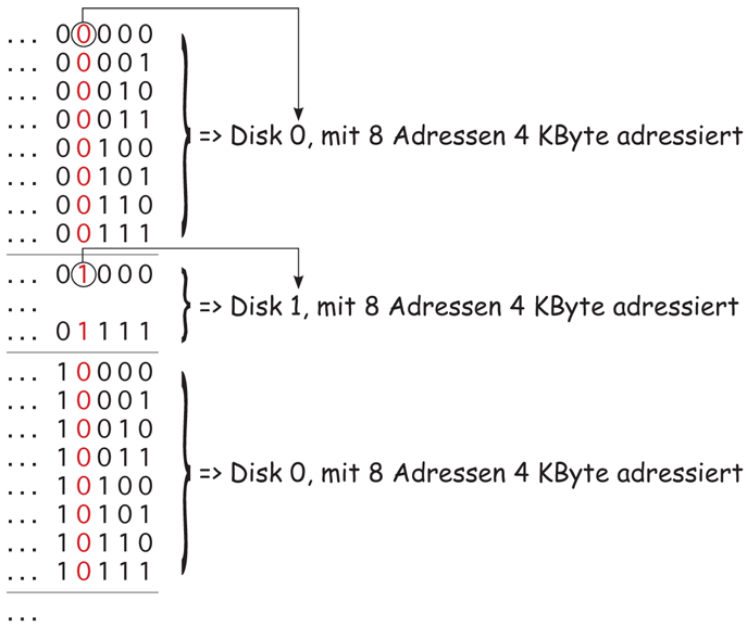


Abbildung 6.6: Beispiel zum Striping-Algorithmus im RAID0-Controller

Diejenigen Speicheradressen, welche auf der Partner-Harddisk besetzt werden, sind folglich auf der betrachteten Harddisk unbesetzt. Damit dieser freibleibende Speicherplatz auf den Harddisks nicht ungenutzt bleibt, wird das für die Aufteilung betrachtete Bit aus der Adresse gelöscht. So ergibt sich ein neuer zusammenhängender Speicherbereich, der allerdings abwechselnd nur jede zweite Originaladresse berücksichtigt.

Hinweis
Der Offset, welcher die Stelle dieses Entscheidungs-Bits angibt, ist in der VHDL-Implementation als Konstante mit Namen STRIPE definiert. Damit nur eine einzige Harddisk angesprochen werden kann, beträgt der initiale Wert der Konstante für diese Arbeit 40. Das vierzigste Bit ist für den zusammenhängenden Speicherbereich einer Test-Harddisk nicht relevant. Die Daten sind vollständig auf dieser einen Test-Harddisk gespeichert.

6.5.3 Definierte Zustände im Controller

IDLE ist der Basiszustand und als Default gesetzt. Die Finite State Machine wartet auf Signale entweder vom Master Controller oder vom FIFO. Sobald das FIFO Daten meldet, startet die FSM den WRITE-Ablauf. Ein Read-Request wird vom Master Controller durch das Senden der Adresse gemeldet.

GET_DATA Dieser Zustand wird nur eingenommen, wenn Daten im FIFO vorhanden sind. Nun bleibt man solange in GET_DATA, bis das FIFO die Daten liefert (Es müssen Daten vorhanden sein. Dies wird durch den Master sichergestellt.). Falls die ersten 64 Bit erfolgreich empfangen werden, entscheidet der Zustand anhand dieser Adresse, auf welche Harddisk geschrieben werden soll, und speichert sich diese Variable. Gleichzeitig wird die Adresse um dieses eine Bit verkleinert.

WRITE_READY Die Adresse wird bestimmt und im Adressregister gespeichert. Sobald die 32 Bit Daten erfolgreich aus dem FIFO gelesen worden sind, wird sogleich der nächste FIFO-Request gestellt. Bei jedem Empfang von Daten erhöht sich der Offset um Eins. Falls eine Harddisk als ≪nicht erreichbar≫ markiert ist, wird sofort ein neuer Request ans FIFO gestellt. Die Daten werden verworfen und nicht weitergeschickt.

WRITE_HD0 Dieser Zustand sendet die Daten an die SATA Controller der ersten Harddisk und wartet auf ein Acknowledge-Signal oder auf ein Fehlersignal von der Harddisk. Bei erfolgreichem Schreiben werden sofort wieder Daten aus dem FIFO gelesen, solange der Offset noch nicht erreicht ist. Der Schreibvorgang muss vor Verlassen dieses Zustands beendet und bestätigt sein.

WRITE_HD1 Dieser Zustand sendet die Daten an die SATA Controller der zweiten Harddisk und wartet auf ein Acknowledge-Signal oder auf ein Fehlersignal von der Harddisk. Bei erfolgreichem Schreiben werden sofort wieder Daten aus dem FIFO gelesen, solange der Offset noch nicht erreicht ist. Der Schreibvorgang muss vor Verlassen dieses Zustands beendet und bestätigt sein.

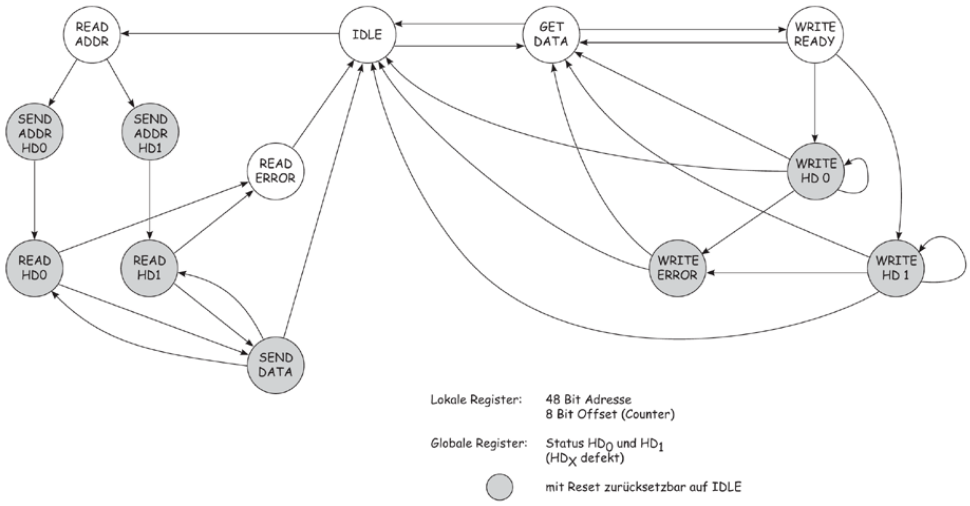


Abbildung 6.7: Finite State Machine für den RAID0-Controller

WRITE_ERROR Falls während des Schreibvorgangs ein Fehler auftritt (zum Beispiel Harddisk-Ausfall), wird dies dem Master gemeldet. Konnten die Daten nicht erfolgreich an die gewünschte Adresse geschrieben werden, generiert dies eine Fehlermeldung an den Master. Der Zustand wartet auf ein ACK und fährt dann mit dem normalen Schreibzyklus fort. Das System bleibt funktionsfähig, es sei denn auch die zweite Platte falle aus.

READ_ADDRESS Die Adresse wird mit Bestätigung an den Master gelesen. Anhand dieser Adresse fällt nun der Entscheid, von welcher Harddisk die Daten gelesen werden sollen.

SEND_ADDRESS_HD0 Sendet die Adresse an den SATA Controller der ersten Festplatte und wiederholt das Senden, falls die Festplatte kein ACK zurückschickt. Sobald die Adresse erfolgreich gesendet wurde, ist der nächste Zustand READ HD0.

SEND_ADDRESS_HD1 Sendet die Adresse an den SATA Controller der zweiten Festplatte und wiederholt das Senden, falls die Festplatte kein ACK zurückschickt. Sobald die Adresse erfolgreich gesendet wurde, ist der nächste Zustand READ HD1.

READ_HD0 Man wartet solange, bis die Daten von der ersten Festplatte korrekt anliegen und liest sie dann in das Datenregister. Falls der letzte 32Bit-Block von insgesamt 128 Blocks gelesen wurde, wechselt man in IDLE, da dieser Read-Zyklus abgeschlossen ist.

READ_HD1 Man wartet solange, bis die Daten von der zweiten Festplatte korrekt anliegen und liest sie dann in das Datenregister. Falls der letzte 32Bit-Block von insgesamt 128 Blocks gelesen wurde, wechselt man in IDLE, da dieser Read-Zyklus abgeschlossen ist.

READ_ERROR Falls eine Festplatte einen Fehler meldet, wird dieser an den Master weitergeleitet. Der Read-Vorgang wird abgebrochen, es sind nicht alle Daten verfügbar.

SEND_DATA In SEND_DATA werden die Datenblöcke an den Master weitergeleitet. Nur wenn dieser eine positive Bestätigung schickt – d.h. er hat die Daten abgeholt –, gilt der Vorgang für diesen Block als beendet. Der Vorgang wird für jeden Datenblock wiederholt.

6.5.4 Intern verwendete Register

Damit verschiedene Signale, welche intern zwischen verschiedenen Status verwendet werden, temporär zwischengespeichert sind, werden nebst dem (aktuellen) Status-Signal weitere Register (Signale) benötigt.

HD0	HD 1	
0	0	beide Harddisks sind fehlerhaft
0	1	Harddisk 1 ist defekt
1	0	Harddisk 0 ist defekt
1	1	beide Harddisks sind in Ordnung

Tabelle 6.3: Code für Harddisk-Status im RAID0 Controller

OffsetxDP, OffsetxDN ist als Counter gedacht, der die Anzahl von 32 Bit Datenpaketen während eines Read- oder Write-Request zählt (Datenblock-Offset). Es ist ein 8 Bit Register. Somit ist sichergestellt, dass der Counter bis 130 erhöht werden kann.

AddressxDP, AddressxDN beinhaltet die 48 Bit Adresse eines Read- oder Write-Requests. Es ist folglich ein 48 Bit breites Register.

DataxDP, DataxDN ist für das Zwischenspeichern von einzelnen 32 Bit Datenblöcken pro Clock-Cycle gedacht und hat eine Breite von 32 Bit.

HarddiskxDP, HarddiskxDN bestimmt, auf welcher Harddisk operiert wird. Da dieses Signal nur ein Bit breit ist, kann auch nur maximal zwischen zwei Harddisks – Harddisk 0 und Harddisk 1 – auf binärer Ebene unterschieden werden.

StatusHDxDP, StatusHDxDN speichert den Zustand einer Harddisk. Von den zwei Bit wird das erste für die eine Harddisk und das zweite Bit für die andere Harddisk verwendet. Null signalisiert einen Harddisk-Fehler, Eins bezeugt, dass die entsprechende Harddisk in Ordnung ist. Man beachte den Zusammenhang zwischen der Bitbreite von HarddiskHDxDP und StatusHDxDP.

Present State	Signale		Next State
IDLE	FIFO_IsEmptyxE = 0	->	GET DATA
	FSM_ReadDataxE = 1	->	READ ADDRESS
GET DATA	FIFO_ReadDataxACK = 1 AND FIFO_ReadDataxERR = 0	->	WRITE READY
	FIFO_ReadDataxACK = 0	->	GET DATA
WRITE READY	OffsetxDP = 0 OffsetxDP = 1	->	GET DATA
	OffsetxDP >1 AND HarddiskxDP = 0	->	WRITE HD0
	OffsetxDP >1 AND HarddiskxDP = 1	->	WRITE HD0
WRITE HD0	SATA0_WriteDataxACK = 1 AND OffsetxDP <130	->	GET DATA
	SATA0_WriteDataxACK = 1 AND OffsetxDP = 130	->	IDLE
	SATA0_WriteDataxERR = 1	>	WRITE ERROR
WRITE HD1	SATA1_WriteDataxACK = 1 AND OffsetxDP <130	->	GET DATA
	SATA0_WriteDataxACK = 1 AND OffsetxDP = 130	->	IDLE
	SATA1_WriteDataxERR = 1	>	WRITE ERROR
WRITE ERROR	OffsetxDP <130	>	GET DATA
	OffsetxDP = 130	>	IDLE

Tabelle 6.4: Alle Zustände und Übergangssignale, die für den Ablauf eines Write-Requests relevant sind

Present State	Signale		Next State
READ ADDRESS	OffsetxDP = 0 OR OffsetxDP = 1	>	READ ADDRESS
	OffsetxDP = 2 AND AddressxDP(STRIPE) = 0	>	SEND ADDRESS HD0
	OffsetxDP = 2 AND AddressxDP(STRIPE) = 1	>	SEND ADDRESS HD1
SEND ADDRESS HD0	OffsetxDP <2	>	SEND ADDRESS HD0
	OffsetxDP = 2 AND SATA0_WriteDataxACK = 0	>	SEND ADDRESS HD0
	OffsetxDP = 2 AND SATA0_WriteDataxACK = 1	>	READ HD0
SEND ADDRESS HD1	OffsetxDP <2	>	SEND ADDRESS HD1
	OffsetxDP = 2 AND SATA1_WriteDataxACK = 0	>	SEND ADDRESS HD1
	OffsetxDP = 2 AND SATA1_WriteDataxACK = 1	>	READ HD1
READ HD0	OffsetxDP <130 AND SATA0_ReadDataxE = 1	>	SEND DATA
	OffsetxDP = 130	>	IDLE
READ HD1	OffsetxDP <130 AND SATA1_ReadDataxE = 1	>	SEND DATA
	OffsetxDP = 130	>	IDLE
SEND DATA	FSM_WriteDataxACK = 1 AND HarddiskxDP = 0	>	READ HD0
SEND DATA	FSM_WriteDataxACK = 1 AND HarddiskxDP = 1	>	READ HD0
	FSM_WriteDataxERR = 1	>	READ ERROR
READ ERROR	all	>	IDLE

Tabelle 6.5: Alle Zustände und Übergangssignale, die für den Ablauf eines Read-Requests relevant sind

Teil III

Anhang

Kapitel 7

Fazit

Je länger je mehr wird in der Informatikwelt viel Speicherplatz benötigt. Der Speicher muss ein möglichst grosses Fassungsvolumen, ein einfach verwaltbares Speichermanagement und eine kleine Antwortzeit haben. Hinzu kommt die Forderung nach Datensicherheit und Datenverfügbarkeit. NAS und RAID sind mögliche Lösungen um obigen Trends gerecht zu werden. NAS bietet ein integriertes File-System und ist direkt aus einem gängigen Netzwerk ansprechbar. RAID berücksichtigt den Aspekt der Datenredundanz. Die Kombination dieser beiden Technologien zusammen mit den neuen Standards für die Netzwerk- bzw. Harddisk-Anbindung ergibt ein Systemdesign, das viele Vorteile miteinander verbindet.

Diese Semesterarbeit hat uns Einsicht in die komplexen Zusammenhänge der verwendeten Technologien gegeben. Die Umsetzung von einzelnen Komponenten, insbesondere die Implementation des SATA-Kontrollers ist sehr zeitaufwändig. Deshalb haben wir beschlossen, nicht näher auf den SATA-Kontroller einzugehen, und haben uns vermehrt auf das Systemdesign konzentriert.
Leider konnten wir das Ziel, unsere Umsetzung auf dem Evaluationsboard von Xilinx zu testen, nicht erreichen. Der krankheitsbedingte Ausfall eines Mitarbeiters, der das Know-How im Umgang mit dem Board hat, verhinderte eine frühzeitige Auseinandersetzung mit der Inbetriebnahme des Boards. So haben wir uns entschieden, uns nicht mit dem Schreiben eines Linux-Treibers für die Kommunikation mit dem FPGA-Bus auseinanderzusetzen. Vielmehr haben wir versucht, das Board in Eigeninitiative in Betrieb zu nehmen, was uns auch für eine Beispielsapplikation gelungen ist.

Trotz manchen Änderungen konnten wir die Arbeit zu einem guten Ende bringen. Geholfen hat uns dabei vor allem die Offenheit und Hilfsbereitschaft der Mitarbeiter der Firma SCS. Die Betreuung und der Informationsaustausch gaben uns bei Unsicherheiten neue Motivation.
Der Umstand, dass wir den Abschluss-Vortrag bereits nach zwei Dritteln unserer Arbeit halten mussten, ist im Nachhinein gesehen kein Nachteil gewesen. Dank einer Art Probelauf – wir haben den Vortrag vor dem eigentlichen Termin an der ETH auch in der Firma gehalten – konnten noch ein paar Kleinigkeiten verbessert werden.

Die Semesterarbeit umfasst ein grosses Spektrum an Themen, in welche wir uns einarbeiten mussten. Die Erkenntnis, sich in bestimmten Fragen auf Grundlegendes einzuschränken, oder gewisse Themen ganz wegzulassen, ist ein lehrreicher Aspekt. Die Diplomarbeit, in welcher wir die Arbeit weiterführen, gibt uns die Möglichkeit, diese Themengebiete nochmals vertiefter zu betrachten. Selbstverständlich stehen wir für weitere Fragen gerne zur Verfügung.

David H. Gehring
Beat Schilbach

Kapitel 8

VHDL Code

8.1 Main File

```
----------------------------------------------------------------------------------------------------
--  Project      : NAS
--  Part:        : SATA Controller
--  Filename     : sata_raid.vhd
--  Purpose      : Main File
--  Responsible  : David H. Gehring, Beat Schilbach
----------------------------------------------------------------------------------------------------
--  Compiler & Synthesis Settings:
--
----------------------------------------------------------------------------------------------------
--  Description:
--  This is the Main File of the SATA Controller. It connects all entities together
--
----------------------------------------------------------------------------------------------------
--  Limitations & Assumptions:
--
----------------------------------------------------------------------------------------------------
--  Known Errors:
--
----------------------------------------------------------------------------------------------------
--  History Of Change:
----------------------------------------------------------------------------------------------------
--  Version | Date     | Author              | Remarks
--  0.1     | 11.06.04 | David H. Gehring    | File created
----------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity sata_raid is
    Port(   -- Global Signals
            RST                 : in  std_logic;                        -- Asynchronous RESET
            CLK                 : in  std_logic;                        -- CLOCK
            -- Port Signals
            OS_ReadDataxD       : in  std_logic_vector(31 downto 0);    -- DATA IN
            OS_ReadDataxE       : in  std_logic;                        -- DATA IN Enable
            OS_ReadDataxACK     : out std_logic;                        -- DATA IN Achnowledge
```

```
            OS_ReadDataxERR     : out std_logic;                        -- DATA IN Error
            OS_WriteDataxD      : out std_logic_vector(31 downto 0);    -- DATA OUT
            OS_WriteDataxE      : out std_logic;                        -- DATA OUT Enable
            OS_WriteDataxACK    : in  std_logic;                        -- DATA OUT Acknowledge
            OS_WriteDataxERR    : in  std_logic                         -- DATA OUT Error
    );
end;

architecture Behavioral of sata_raid is

    -- ***********************************************************************************************
    -- *                                     INTERNAL SIGNALS                                        *
    -- ***********************************************************************************************
    -- RAID0 out => SATA in
    signal R0M_SM0_D        : std_logic_vector(31 downto 0);
    signal R0M_SM0_E        : std_logic;
    signal R0M_SM0_ACK      : std_logic;
    signal R0M_SM0_ERR      : std_logic;
    signal R0M_SM1_D        : std_logic_vector(31 downto 0);
    signal R0M_SM1_E        : std_logic;
    signal R0M_SM1_ACK      : std_logic;
    signal R0M_SM1_ERR      : std_logic;
    -- SATA out => RAID0 in
    signal SM0_R0M_D        : std_logic_vector(31 downto 0);
    signal SM0_R0M_E        : std_logic;
    signal SM0_R0M_ACK      : std_logic;
    signal SM0_R0M_ERR      : std_logic;
    signal SM1_R0M_D        : std_logic_vector(31 downto 0);
    signal SM1_R0M_E        : std_logic;
    signal SM1_R0M_ACK      : std_logic;
    signal SM1_R0M_ERR      : std_logic;
    -- FIFO out => RAID0 in
    signal FIM_R0M_D        : std_logic_vector(31 downto 0);
    signal FIM_R0M_E        : std_logic;
    signal FIM_R0M_ACK      : std_logic;
    signal FIM_R0M_ERR      : std_logic;
    -- FIFO out
    signal FIM_Empty        : std_logic;
    signal FIM_Full         : std_logic;
    -- FSMMASTER out => FIFO in
    signal R1M_FIM_D        : std_logic_vector(31 downto 0);
    signal R1M_FIM_E        : std_logic;
    signal R1M_FIM_ACK      : std_logic;
    signal R1M_FIM_ERR      : std_logic;
    -- RAID0 out => FSMMASTER in
    signal R0M_R1M_D        : std_logic_vector(31 downto 0);
    signal R0M_R1M_E        : std_logic;
    signal R0M_R1M_ACK      : std_logic;
    signal R0M_R1M_ERR      : std_logic;
    -- FSMMASTER out => RAID0 in
    signal R1M_R0M_D        : std_logic_vector(31 downto 0);
    signal R1M_R0M_E        : std_logic;
    signal R1M_R0M_ACK      : std_logic;
    signal R1M_R0M_ERR      : std_logic;
    -- HOSTINTERFACE out => FSMMASTER in
    signal HIM_R1M_D        : std_logic_vector(31 downto 0);
    signal HIM_R1M_E        : std_logic;
    signal HIM_R1M_ACK      : std_logic;
    signal HIM_R1M_ERR      : std_logic;
    -- Raid1 out => HOSTINTERFACE in
    signal R1M_HIM_D        : std_logic_vector(31 downto 0);
    signal R1M_HIM_E        : std_logic;
    signal R1M_HIM_ACK      : std_logic;
    signal R1M_HIM_ERR      : std_logic;
    -- LINK out => FSMMASTER in
    signal LIM_R1M_D        : std_logic_vector(31 downto 0);
    signal LIM_R1M_E        : std_logic;
```

```
signal LIM_R1M_ACK      : std_logic;
signal LIM_R1M_ERR      : std_logic;
-- FSMMASTER out => LINK in
signal R1M_LIM_D        : std_logic_vector(31 downto 0);
signal R1M_LIM_E        : std_logic;
signal R1M_LIM_ACK      : std_logic;
signal R1M_LIM_ERR      : std_logic;

-- ******************************************************************************************
-- *                                        COMPONENTS                                      *
-- ******************************************************************************************
component sata_raid_sata
    port (
            RST                 : in  std_logic;
            CLK                 : in  std_logic;
            Raid_WriteDataxD    : out std_logic_vector (31 downto 0);
            Raid_WriteDataxE    : out std_logic;
            Raid_WriteDataxACK  : in  std_logic;
            Raid_WriteDataxERR  : in  std_logic;
            Raid_ReadDataxD     : in  std_logic_vector (31 downto 0);
            Raid_ReadDataxE     : in  std_logic;
            Raid_ReadDataxACK   : out std_logic;
            Raid_ReadDataxERR   : out std_logic
        );
end component;

component sata_raid_raid0
    port (
            RST                 : in  std_logic;
            CLK                 : in  std_logic;
            SATA0_WriteDataxD   : out std_logic_vector (31 downto 0);
            SATA0_WriteDataxE   : out std_logic;
            SATA0_WriteDataxACK : in  std_logic;
            SATA0_WriteDataxERR : in  std_logic;
            SATA0_ReadDataxD    : in  std_logic_vector (31 downto 0);
            SATA0_ReadDataxE    : in  std_logic;
            SATA0_ReadDataxACK  : out std_logic;
            SATA0_ReadDataxERR  : out std_logic;
            SATA1_WriteDataxD   : out std_logic_vector (31 downto 0);
            SATA1_WriteDataxE   : out std_logic;
            SATA1_WriteDataxACK : in  std_logic;
            SATA1_WriteDataxERR : in  std_logic;
            SATA1_ReadDataxD    : in  std_logic_vector (31 downto 0);
            SATA1_ReadDataxE    : in  std_logic;
            SATA1_ReadDataxACK  : out std_logic;
            SATA1_ReadDataxERR  : out std_logic;
            FIFO_ReadDataxD     : in  std_logic_vector (31 downto 0);
            FIFO_ReadDataxE     : out std_logic;
            FIFO_ReadDataxACK   : in  std_logic;
            FIFO_ReadDataxERR   : in  std_logic;
            FIFO_IsEmptyxE      : in  std_logic;
            FSM_WriteDataxD     : out std_logic_vector (31 downto 0);
            FSM_WriteDataxE     : out std_logic;
            FSM_WriteDataxACK   : in  std_logic;
            FSM_WriteDataxERR   : in  std_logic;
            FSM_ReadDataxD      : in  std_logic_vector (31 downto 0);
            FSM_ReadDataxE      : in  std_logic;
            FSM_ReadDataxACK    : out std_logic;
            FSM_ReadDataxERR    : out std_logic
        );
end component;

component sata_raid_fifo
  port (
            RST                 : in  std_logic;
            CLK                 : in  std_logic;
            DataOutxD           : out std_logic_vector (31 downto 0);
```

```
            DataOutxE           : in  std_logic;
            DataOutxACK         : out std_logic;
            DataOutxERR         : out std_logic;
            DataInxD            : in  std_logic_vector (31 downto 0);
            DataInxE            : in  std_logic;
            DataInxACK          : out std_logic;
            DataInxERR          : out std_logic;
            FifoIsEmptyxE       : out std_logic;
            FifoIsFullxE        : out std_logic
        );
end component;

component sata_raid_hostinterface
  port (
            RST                 : in  std_logic;
            CLK                 : in  std_logic;
            OS_WriteDataxD      : out std_logic_vector(31 downto 0);
            OS_WriteDataxE      : out std_logic;
            OS_WriteDataxACK    : in  std_logic;
            OS_WriteDataxERR    : in  std_logic;
            OS_ReadDataxD       : in  std_logic_vector(31 downto 0);
            OS_ReadDataxE       : in  std_logic;
            OS_ReadDataxACK     : out std_logic;
            OS_ReadDataxERR     : out std_logic;
            FSM_WriteDataxD     : out std_logic_vector (31 downto 0);
            FSM_WriteDataxE     : out std_logic;
            FSM_WriteDataxACK   : in  std_logic;
            FSM_WriteDataxERR   : in  std_logic;
            FSM_ReadDataxD      : in  std_logic_vector (31 downto 0);
            FSM_ReadDataxE      : in  std_logic;
            FSM_ReadDataxACK    : out std_logic;
            FSM_ReadDataxERR    : out std_logic
        );
end component;

component sata_raid_fsmmaster
    port (
            CLK                 : in  std_logic;
            RST                 : in  std_logic;
            Raid_WriteDataxD    : out std_logic_vector (31 downto 0);
            Raid_WriteDataxE    : out std_logic;
            Raid_WriteDataxACK  : in  std_logic;
            Raid_WriteDataxERR  : in  std_logic;
            Raid_ReadDataxD     : in  std_logic_vector (31 downto 0);
            Raid_ReadDataxE     : in  std_logic;
            Raid_ReadDataxACK   : out std_logic;
            Raid_ReadDataxERR   : out std_logic;
            FM_WriteDataxD      : out std_logic_vector (31 downto 0);
            FM_WriteDataxE      : out std_logic;
            FM_WriteDataxACK    : in  std_logic;
            FM_WriteDataxERR    : in  std_logic;
            FM_IsFull           : in  std_logic;
            HI_WriteDataxD      : out std_logic_vector (31 downto 0);
            HI_WriteDataxE      : out std_logic;
            HI_WriteDataxACK    : in  std_logic;
            HI_WriteDataxERR    : in  std_logic;
            HI_ReadDataxD       : in  std_logic_vector (31 downto 0);
            HI_ReadDataxE       : in  std_logic;
            HI_ReadDataxACK     : out std_logic;
            HI_ReadDataxERR     : out std_logic;
            LINK_WriteDataxD    : out std_logic_vector (31 downto 0);
            LINK_WriteDataxE    : out std_logic;
            LINK_WriteDataxACK  : in  std_logic;
            LINK_WriteDataxERR  : in  std_logic;
            LINK_ReadDataxD     : in  std_logic_vector (31 downto 0);
            LINK_ReadDataxE     : in  std_logic;
            LINK_ReadDataxACK   : out std_logic;
```

```
            LINK_ReadDataxERR   : out std_logic
        );
    end component;

    component sata_raid_link
        port (
            CLK                 : in  std_logic;
            RST                 : in  std_logic;
            FSM_WriteDataxD     : out std_logic_vector (31 downto 0);
            FSM_WriteDataxE     : out std_logic;
            FSM_WriteDataxACK   : in  std_logic;
            FSM_WriteDataxERR   : in  std_logic;
            FSM_ReadDataxD      : in  std_logic_vector (31 downto 0);
            FSM_ReadDataxE      : in  std_logic;
            FSM_ReadDataxACK    : out std_logic;
            FSM_ReadDataxERR    : out std_logic
        );
    end component;

begin

    -- ***********************************************************************************************
    -- *                                    MAP CONNECTIONS                                          *
    -- ***********************************************************************************************
    -- Serial ATA Master 0
    SM0 : sata_raid_sata
        port map (
                CLK                     => CLK,
                RST                     => RST,
                Raid_ReadDataxD         => ROM_SM0_D,
                Raid_ReadDataxE         => ROM_SM0_E,
                Raid_ReadDataxACK       => ROM_SM0_ACK,
                Raid_ReadDataxERR       => ROM_SM0_ERR,
                Raid_WriteDataxD        => SM0_ROM_D,
                Raid_WriteDataxE        => SM0_ROM_E,
                Raid_WriteDataxACK      => SM0_ROM_ACK,
                Raid_WriteDataxERR      => SM0_ROM_ERR
            );

    -- Serial ATA Master 1
    SM1 : sata_raid_sata
        port map (
                CLK                     => CLK,
                RST                     => RST,
                Raid_ReadDataxD         => ROM_SM1_D,
                Raid_ReadDataxE         => ROM_SM1_E,
                Raid_ReadDataxACK       => ROM_SM1_ACK,
                Raid_ReadDataxERR       => ROM_SM1_ERR,
                Raid_WriteDataxD        => SM1_ROM_D,
                Raid_WriteDataxE        => SM1_ROM_E,
                Raid_WriteDataxACK      => SM1_ROM_ACK,
                Raid_WriteDataxERR      => SM1_ROM_ERR
            );


    -- Raid0 Master
    ROM : sata_raid_raid0
      port map  (
                CLK                     => CLK,
                RST                     => RST,
                SATA0_WriteDataxD       => ROM_SM0_D,
                SATA0_WriteDataxE       => ROM_SM0_E,
                SATA0_WriteDataxACK     => ROM_SM0_ACK,
                SATA0_WriteDataxERR     => ROM_SM0_ERR,
                SATA0_ReadDataxD        => SM0_ROM_D,
                SATA0_ReadDataxE        => SM0_ROM_E,
                SATA0_ReadDataxACK      => SM0_ROM_ACK,
```

```
                SATA0_ReadDataxERR      => SM0_ROM_ERR,
                SATA1_WriteDataxD       => ROM_SM1_D,
                SATA1_WriteDataxE       => ROM_SM1_E,
                SATA1_WriteDataxACK     => ROM_SM1_ACK,
                SATA1_WriteDataxERR     => ROM_SM1_ERR,
                SATA1_ReadDataxD        => SM1_ROM_D,
                SATA1_ReadDataxE        => SM1_ROM_E,
                SATA1_ReadDataxACK      => SM1_ROM_ACK,
                SATA1_ReadDataxERR      => SM1_ROM_ERR,
                FIFO_ReadDataxD         => FIM_ROM_D,
                FIFO_ReadDataxE         => FIM_ROM_E,
                FIFO_ReadDataxACK       => FIM_ROM_ACK,
                FIFO_READDATAxERR       => FIM_ROM_ERR,
                FIFO_IsEmptyxE          => FIM_Empty,
                FSM_WriteDataxD         => ROM_R1M_D,
                FSM_WriteDataxE         => ROM_R1M_E,
                FSM_WriteDataxACK       => ROM_R1M_ACK,
                FSM_WriteDataxERR       => ROM_R1M_ERR,
                FSM_ReadDataxD          => R1M_ROM_D,
                FSM_ReadDataxE          => R1M_ROM_E,
                FSM_ReadDataxACK        => R1M_ROM_ACK,
                FSM_ReadDataxERR        => R1M_ROM_ERR
            );

-- FIFO Master
FIM : sata_raid_fifo
  port map  (
                CLK                     => CLK,
                RST                     => RST,
                DataOutxD               => FIM_ROM_D,
                DataOutxE               => FIM_ROM_E,
                DataOutxACK             => FIM_ROM_ACK,
                DataOutxERR             => FIM_ROM_ERR,
                DataInxD                => R1M_FIM_D,
                DataInxE                => R1M_FIM_E,
                DataInxACK              => R1M_FIM_ACK,
                DataInxERR              => R1M_FIM_ERR,
                FifoIsEmptyxE           => FIM_Empty,
                FifoIsFullxE            => FIM_Full
            );

-- Raid1 Master
R1M : sata_raid_fsmmaster
  port map  (
                CLK                     => CLK,
                RST                     => RST,
                FM_WriteDataxD          => R1M_FIM_D,
                FM_WriteDataxE          => R1M_FIM_E,
                FM_WriteDataxACK        => R1M_FIM_ACK,
                FM_WriteDataxERR        => R1M_FIM_ERR,
                FM_IsFull               => FIM_Full,
                Raid_WriteDataxD        => R1M_ROM_D,
                Raid_WriteDataxE        => R1M_ROM_E,
                Raid_WriteDataxACK      => R1M_ROM_ACK,
                Raid_WriteDataxERR      => R1M_ROM_ERR,
                Raid_ReadDataxD         => ROM_R1M_D,
                Raid_ReadDataxE         => ROM_R1M_E,
                Raid_ReadDataxACK       => ROM_R1M_ACK,
                Raid_ReadDataxERR       => ROM_R1M_ERR,
                HI_WriteDataxD          => R1M_HIM_D,
                HI_WriteDataxE          => R1M_HIM_E,
                HI_WriteDataxACK        => R1M_HIM_ACK,
                HI_WriteDataxERR        => R1M_HIM_ERR,
                HI_ReadDataxD           => HIM_R1M_D,
                HI_ReadDataxE           => HIM_R1M_E,
                HI_ReadDataxACK         => HIM_R1M_ACK,
                HI_ReadDataxERR         => HIM_R1M_ERR,
```

```
                    Link_WriteDataxD        => R1M_LIM_D,
                    Link_WriteDataxE        => R1M_LIM_E,
                    Link_WriteDataxACK      => R1M_LIM_ACK,
                    Link_WriteDataxERR      => R1M_LIM_ERR,
                    Link_ReadDataxD         => LIM_R1M_D,
                    Link_ReadDataxE         => LIM_R1M_E,
                    Link_ReadDataxACK       => LIM_R1M_ACK,
                    Link_ReadDataxERR       => LIM_R1M_ERR
                );

    -- Hostinterface Master
    HIM : sata_raid_hostinterface
      port map  (
                    CLK                     => CLK,
                    RST                     => RST,
                    OS_WriteDataxD          => OS_WriteDataxD,
                    OS_WriteDataxE          => OS_WriteDataxE,
                    OS_WriteDataxACK        => OS_WriteDataxACK,
                    OS_WriteDataxERR        => OS_WriteDataxERR,
                    OS_ReadDataxD           => OS_ReadDataxD,
                    OS_ReadDataxE           => OS_ReadDataxE,
                    OS_ReadDataxACK         => OS_ReadDataxACK,
                    OS_ReadDataxERR         => OS_ReadDataxERR,
                    FSM_WriteDataxD         => HIM_R1M_D,
                    FSM_WriteDataxE         => HIM_R1M_E,
                    FSM_WriteDataxACK       => HIM_R1M_ACK,
                    FSM_WriteDataxERR       => HIM_R1M_ERR,
                    FSM_ReadDataxD          => R1M_HIM_D,
                    FSM_ReadDataxE          => R1M_HIM_E,
                    FSM_ReadDataxACK        => R1M_HIM_ACK,
                    FSM_ReadDataxERR        => R1M_HIM_ERR
                );

    -- Link Master
    LIM : sata_raid_link
      port map  (
                    CLK                     => CLK,
                    RST                     => RST,
                    FSM_WriteDataxD         => LIM_R1M_D,
                    FSM_WriteDataxE         => LIM_R1M_E,
                    FSM_WriteDataxACK       => LIM_R1M_ACK,
                    FSM_WriteDataxERR       => LIM_R1M_ERR,
                    FSM_ReadDataxD          => R1M_LIM_D,
                    FSM_ReadDataxE          => R1M_LIM_E,
                    FSM_ReadDataxACK        => R1M_LIM_ACK,
                    FSM_ReadDataxERR        => R1M_LIM_ERR
                );

end Behavioral;
```

8.2 Host Interface (Dummy)

```
----------------------------------------------------------------------------------------------------
--  Project      : NAS
--  Part:        : SATA Controller
--  Filename     : sata_raid_hostinterface.vhd
--  Purpose      : Hostinterface
--  Responsible  : David H. Gehring, Beat Schilbach
----------------------------------------------------------------------------------------------------
--  Compiler & Synthesis Settings:
--
----------------------------------------------------------------------------------------------------
--  Description:
```

```
--  This is the file, which connects the SATA Controller to the OPB Bus of the FPGA
--  It is also the interface between hardware (Controller) and software (Operating System)
--
-------------------------------------------------------------------------------------------------------
--  Limitations & Assumptions:
--
-------------------------------------------------------------------------------------------------------
--  Known Errors:
--
-------------------------------------------------------------------------------------------------------
--  History Of Change:
-------------------------------------------------------------------------------------------------------
--  Version | Date     | Author              | Remarks
--  0.1     | 11.06.04 | David H. Gehring    | File created
-------------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity sata_raid_hostinterface is
    Port (  -- Entity-Global Signals
            RST                 : in  std_logic;                        -- Asynchronous RESET
            CLK                 : in  std_logic;                        -- CLOCK

            -- Connection to OS
            OS_WriteDataxD      : out std_logic_vector(31 downto 0);    -- Data OUT
            OS_WriteDataxE      : out std_logic;                        -- Data OUT Enable
            OS_WriteDataxACK    : in  std_logic;                        -- Data OUT Acknowledge
            OS_WriteDataxERR    : in  std_logic;                        -- Data OUT Error
            OS_ReadDataxD       : in  std_logic_vector(31 downto 0);    -- Data IN
            OS_ReadDataxE       : in  std_logic;                        -- Data IN Enable
            OS_ReadDataxACK     : out std_logic;                        -- Data In Acknowledge
            OS_ReadDataxERR     : out std_logic;                        -- Data In Error

            -- Connection to FSM
            FSM_WriteDataxD     : out std_logic_vector(31 downto 0);
            FSM_WriteDataxE     : out std_logic;
            FSM_WriteDataxACK   : in  std_logic;
            FSM_WriteDataxERR   : in  std_logic;
            FSM_ReadDataxD      : in  std_logic_vector(31 downto 0);
            FSM_ReadDataxE      : in  std_logic;
            FSM_ReadDataxACK    : out std_logic;
            FSM_ReadDataxERR    : out std_logic

          );
end sata_raid_hostinterface;

architecture Behavioral of sata_raid_hostinterface is
    -- ***********************************************************************************************
    -- *                                     INTERNAL SIGNALS                                        *
    -- ***********************************************************************************************
    signal DataxDP, DataxDN         : std_logic_vector(31 downto 0);        -- data register
    type state_values is (IDLE, RECEIVE_OS, SEND_OS, WRITE_FSM, READ_FSM);  -- fsm states
    signal StatexDP, StatexDN       : state_values;                         -- fsm state register

begin
    -- ***********************************************************************************************
    -- *                                 THE LOGIC COMBINATORIAL PROCESS                             *
    -- ***********************************************************************************************
    logic_inc : process (   DataxDP, StatexDP,
                            FSM_ReadDataxE, FSM_ReadDataxD, FSM_WriteDataxACK, FSM_WriteDataxERR,
                            OS_ReadDataxE, OS_ReadDataxD, OS_WriteDataxERR, OS_WriteDataxACK
                        )
    BEGIN
```

```
-- default behaviour
DataxDN             <= DataxDP;
StatexDN            <= StatexDP;

-- default values
OS_WriteDataxD      <= (others => '0');
OS_WriteDataxE      <= '0';
OS_ReadDataxACK     <= '0';
OS_ReadDataxERR     <= '0';
FSM_WriteDataxD     <= (others => '0');
FSM_WriteDataxE     <= '0';
FSM_ReadDataxACK    <= '0';
FSM_ReadDataxERR    <= '0';


-- FSM sata
CASE StatexDP IS

    -- ****
    -- IDLE
    -- ****
    WHEN IDLE =>
        IF (OS_ReadDataxE = '1') THEN
            StatexDN            <= RECEIVE_OS;
        ELSIF (FSM_ReadDataxE = '1') THEN
            StatexDN            <= READ_FSM;
        ELSE
            StatexDN            <= StatexDP;
        END IF;


    -- **********
    -- RECEIVE_OS
    -- **********
    WHEN RECEIVE_OS =>
        DataxDN                 <= OS_ReadDataxD;
        StatexDN                <= WRITE_FSM;

    -- *********
    -- WRITE_FSM
    -- *********
    WHEN WRITE_FSM =>
        IF (FSM_WriteDataxERR = '1') THEN
            OS_ReadDataxERR     <= '1';
            StatexDN            <= IDLE;
        ELSIF (FSM_WriteDataxACK = '1') THEN
            OS_ReadDataxACK     <= '1';
            StatexDN            <= IDLE;
        ELSE
            FSM_WriteDataxE     <= '1';
            FSM_WriteDataxD     <= DataxDP;
        END IF;

    -- ********
    -- READ_FSM
    -- ********
    WHEN READ_FSM =>
        DataxDN                 <= FSM_ReadDataxD;
        StatexDN                <= SEND_OS;

    -- *******
    -- SEND_OS
    -- *******
    WHEN SEND_OS =>
        IF (OS_WriteDataxERR = '1') THEN
            FSM_ReadDataxERR    <= '1';
            StatexDN            <= IDLE;
```

```
                ELSIF (OS_WriteDataxACK = '1') THEN
                    FSM_ReadDataxACK    <= '1';
                    StatexDN            <= IDLE;
                ELSE
                    OS_WriteDataxE      <= '1';
                    OS_WriteDataxD      <= DataxDP;
                END IF;

        END CASE;


    end process logic_inc;  -- combinatorial process


    -- ***************************************************************************************
    -- *                                    SEQUENTIAL PROCESS                               *
    -- ***************************************************************************************
    -- purpose: update registers (counter)
    reg_update : process (RST, CLK, DataxDN, StatexDN)
    begin                                               -- process reg_update

        IF (RST = '0') THEN                             -- asynchronous reset (active low)
            DataxDP     <= (others => '0');
            StatexDP    <= IDLE;
        ELSIF ( (CLK'event) AND (CLK = '1') ) THEN      -- rising clock edge
            DataxDP     <= DataxDN;
            StatexDP    <= StatexDN;
        END IF;

    end process reg_update;


end Behavioral;
```

8.3 FSM_Master

```
----------------------------------------------------------------------------------------------------
--  Project      : NAS
--  Part:        : SATA Controller
--  Filename     : sata_raid_fsmmaster.vhd
--  Purpose      : Hostinterface
--  Responsible  : David H. Gehring, Beat Schilbach
----------------------------------------------------------------------------------------------------
--  Compiler & Synthesis Settings:
--
----------------------------------------------------------------------------------------------------
--  Description:
--  This is the Heart of the SATA Controller
--  It's functionality is the RAID1 Controller, tracking the system state and offer the
--  reconstruction process (which is implemented in hardware)
--
----------------------------------------------------------------------------------------------------
--  Limitations & Assumptions:
--
----------------------------------------------------------------------------------------------------
--  Known Errors:
--
----------------------------------------------------------------------------------------------------
--  History Of Change:
----------------------------------------------------------------------------------------------------
--  Version | Date     | Author              | Remarks
--  0.1     | 11.06.04 | David H. Gehring    | File created
----------------------------------------------------------------------------------------------------
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity sata_raid_fsmmaster is
    Port (  -- Global Signals
            RST                 : in  std_logic;                        -- Asynchronous RESET
            CLK                 : in  std_logic;                        -- CLOCK

            -- Signals to Hostinterface
            HI_WriteDataxD      : out std_logic_vector(31 downto 0);    -- Data OUT
            HI_WriteDataxE      : out std_logic;                        -- Data OUT Enable
            HI_WriteDataxACK    : in  std_logic;                        -- Atat OUT Acknowledge
            HI_WriteDataxERR    : in  std_logic;                        -- Atat OUT Error
            HI_ReadDataxD       : in  std_logic_vector(31 downto 0);    -- Data IN
            HI_ReadDataxE       : in  std_logic;                        -- Data IN Enable
            HI_ReadDataxACK     : out std_logic;                        -- Data IN Achnowledge
            HI_ReadDataxERR     : out std_logic;                        -- Data IN Error

            -- Signals to FIFO_Master
            FM_WriteDataxD      : out std_logic_vector(31 downto 0);
            FM_WriteDataxE      : out std_logic;
            FM_WriteDataxACK    : in  std_logic;
            FM_WriteDataxERR    : in  std_logic;
            FM_IsFull           : in  std_logic;                        -- Fifo Full

            -- Signals to RAID0
            Raid_ReadDataxD     : in  std_logic_vector(31 downto 0);
            Raid_ReadDataxE     : in  std_logic;
            Raid_ReadDataxACK   : out std_logic;
            Raid_ReadDataxERR   : out std_logic;
            Raid_WriteDataxD    : out std_logic_vector(31 downto 0);
            Raid_WriteDataxE    : out std_logic;
            Raid_WriteDataxACK  : in  std_logic;
            Raid_WriteDataxERR  : in  std_logic;

            -- Signals to Link
            Link_WriteDataxD    : out std_logic_vector(31 downto 0);
            Link_WriteDataxE    : out std_logic;
            Link_WriteDataxACK  : in  std_logic;
            Link_WriteDataxERR  : in  std_logic;
            Link_ReadDataxD     : in  std_logic_vector(31 downto 0);
            Link_ReadDataxE     : in  std_logic;
            Link_ReadDataxACK   : out std_logic;
            Link_ReadDataxERR   : out std_logic
        );

end sata_raid_fsmmaster;

architecture Behavioral of sata_raid_fsmmaster is
    -- ***********************************************************************************************
    -- *                                    INTERNAL SIGNALS                                         *
    -- ***********************************************************************************************
    signal OffsetxDP, OffsetxDN         : std_logic_vector(7 downto 0);     -- data block offset
    signal DataxDP, DataxDN             : std_logic_vector(31 downto 0);    -- data register
    signal Address1xDP, Address1xDN     : std_logic_vector(31 downto 0);    -- first part of address
    signal Address2xDP, Address2xDN     : std_logic_vector(31 downto 0);    -- second part of address
    signal StatusHDxDP, StatusHDxDN     : std_logic_vector(3 downto 0);     -- system state
    signal StatusLinkxDP, StatusLinkxDN : std_logic;                        -- link state
    type state_values is (RESET, IDLE, WRITE_FIFO_MASTER, WRITE_FIFO_SLAVE, WRITE_READY, LINK_ERROR,
                          READ_ADDRESS, SEND_MASTER_ADDRESS, SEND_SLAVE_ADDRESS, READ_MASTER,
                          READ_SLAVE,READ_MASTER_READY, READ_SLAVE_READY, READ_ERROR);
    signal StatexDP, StatexDN           : state_values;                     -- fsm state register

    -- signal declaration
    --             31        23       15       7        0
```

```
    --              ------------------------------------
    -- Address2     |        |        |@@@@@@@@|@@@@@@@@|
    --              ------------------------------------
    --              ------------------------------------
    -- Address1     |@@@@@@@@|@@@@@@@@|@@@@@@@@|@@@@@@@@|
    --              ------------------------------------
    -- status       StatusHD(0) = Address2(28) = M0   0 = NOK
    --              StatusHD(1) = Address2(29) = M1   1 = OK
    --              StatusHD(2) = Address2(30) = S0
    --              StatusHD(3) = Address2(31) = S1
    --              StatusLink  = Address2(27) = LNK
    --
    -- coding
    --              LNK | M0 | M1 | S0 | S1 | Adr | Data | Note
    --              ---------------------------------------------------------------
    --               1  | 1  | 1  | 1  | 1  | no  | yes  | Everything ok (standard)
    --              ---------------------------------------------------------------
    --               1  | 1  | 1  | 1  | 0  | no  | yes  | S1 failed
    --               1  | 1  | 1  | 0  | 1  | no  | yes  | S0 failed
    --               1  | 1  | 0  | 1  | 1  | no  | yes  | M1 failed
    --               1  | 0  | 1  | 1  | 1  | no  | yes  | M0 failed
    --              ---------------------------------------------------------------
    --               0  | 1  | 1  | 1  | 1  | yes | no   | Link down
    --               0  | 1  | 1  | 0  | 0  | no  | no   | Address overflow
    --              ---------------------------------------------------------------
    --               0  | 0  | 1  | 1  | 1  | no  | no   | double error (deny data)
    --               0  | 1  | 0  | 1  | 1  | no  | no   | double error (deny data)


begin
    -- ***********************************************************************************************
    -- *                                THE LOGIC COMBINATORIAL PROCESS                              *
    -- ***********************************************************************************************
    logic_inc : process (   StatexDP, OffsetxDP, DataxDP, Address1xDP, Address2xDP, StatusHDxDP,
                            StatusLinkxDP, HI_WriteDataxACK, HI_WriteDataxERR, HI_ReadDataxD,
                            HI_ReadDataxE, FM_WriteDataxACK, FM_WriteDataxERR, FM_IsFull,
                            Raid_ReadDataxD, Raid_ReadDataxE, Raid_WriteDataxACK, Raid_WriteDataxERR,
                            Link_WriteDataxACK, Link_WriteDataxERR, Link_ReadDataxD, Link_ReadDataxE
                        )

    BEGIN

        -- default behaviour
        DataxDN             <= DataxDP;
        StatexDN            <= StatexDP;
        Address1xDN         <= Address1xDP;
        Address2xDN         <= Address2xDP;
        StatusHDxDN         <= StatusHDxDP;
        StatusLinkxDN       <= StatusLinkxDP;

        -- default output values
        HI_ReadDataxACK     <= '0';
        HI_ReadDataxERR     <= '0';
        HI_WriteDataxD      <= (others => '0');
        HI_WriteDataxE      <= '0';
        Link_ReadDataxACK   <= '0';
        Link_ReadDataxERR   <= '0';
        Link_WriteDataxD    <= (others => '0');
        Link_WriteDataxE    <= '0';
        RAID_ReadDataxACK   <= '0';
        RAID_ReadDataxERR   <= '0';
        RAID_WriteDataxD    <= (others => '0');
        RAID_WriteDataxE    <= '0';
        FM_WriteDataxD      <= (others => '0');
        FM_WriteDataxE      <= '0';

        CASE StatexDP IS
```

```
-- *****
-- RESET
-- *****
WHEN RESET =>
    -- dummy => get system status from nv-ram
    StatusHDxDN     <= (others => '1');
    StatusLinkxDN   <= '1';
    StatexDN        <= IDLE;


-- ****
-- IDLE
-- ****
WHEN IDLE =>
    IF ( (StatusLinkxDP = '0') AND (StatusHDxDP(1 downto 0) < "11") ) THEN
        -- double error, deny any data-transaction
        HI_ReadDataxERR         <= '1';
    ELSIF (HI_ReadDataxE = '1') THEN
        -- go to write cycle
        OffsetxDN               <= (others => '0');
        DataxDN                 <= HI_ReadDataxD;
        StatexDN                <= WRITE_FIFO_MASTER;
    ELSIF ( (HI_WriteDataxACK = '1') AND (HI_WriteDataxERR = '0') ) THEN
        -- go to read cycle (with missuse of the HI_Write* signal)
        OffsetxDN               <= (others => '0');
        Address1xDN             <= HI_ReadDataxD;
        StatexDN                <= READ_ADDRESS;
    ELSE
        StatexDN                <= StatexDP;
    END IF;

-- *****************
-- WRITE_FIFO_MASTER
-- *****************
WHEN WRITE_FIFO_MASTER =>
    IF (FM_IsFull = '1') THEN
        -- FIFO is full wait until space available
        StatexDN                <= StatexDP;
    ELSIF ( (FM_WriteDataxACK = '1') AND (FM_WriteDataxERR = '0') ) THEN
        -- Data written successfully to master
        IF (StatusLinkxDP = '1') THEN
            -- link is up
            StatexDN            <= WRITE_FIFO_SLAVE;
        ELSE
            -- link is down and master harddisks ok
            IF ( (OffsetxDP = 1) AND (StatusHDxDP(3 downto 2) > "00") ) THEN
                -- write address to nv-ram
                StatexDN        <= LINK_ERROR;
            ELSE
                StatexDN        <= WRITE_READY;
                HI_ReadDataxACK <= '1';
            END IF;
        END IF;
    ELSE
        -- write data to fifo_master
        FM_WriteDataxD          <= DataxDP;
        FM_WriteDataxE          <= '1';
    END IF;
    -- get address out of data stream
    IF (OffsetxDP = 0) THEN
        Address1xDN             <= DataxDP;
    ELSIF (OffsetxDP = 1) THEN
        Address2xDN             <= DataxDP;
    END IF;

-- *****************
```

```
-- WRITE_FIFO_SLAVE
-- *****************
WHEN WRITE_FIFO_SLAVE =>
    IF (Link_WriteDataxACK = '1') THEN
        -- Data written successfully to link (to fsm-slave)
        StatexDN            <= WRITE_READY;
        HI_ReadDataxACK     <= '1';
    ELSIF (Link_WriteDataxERR = '1') THEN
        -- an link error occurd
        -- address has to be written to nv-ram
        StatusLinkxDN       <= '0';
        StatexDN            <= LINK_ERROR;
    ELSE
        -- write data to link => for fifo_slave
        Link_WriteDataxD    <= DataxDP;
        Link_WriteDataxE    <= '1';
    END IF;


-- ***********
-- WRITE_READY
-- ***********
WHEN WRITE_READY =>
    IF (OffsetxDP = 130) THEN
        -- written all data
        StatexDN            <= IDLE;
    ELSIF (HI_ReadDataxE = '1') THEN
        -- read data from hostinterface
        DataxDN             <= HI_ReadDataxD;
        StatexDN            <= WRITE_FIFO_MASTER;
        OffsetxDN           <= OffsetxDP + 1;
    ELSE
        HI_ReadDataxACK     <= '1';
    END IF;

-- ***********
-- LINK_ERROR
-- ***********
WHEN LINK_ERROR =>
    IF (OffsetxDP = 1) THEN
        -- address is not available
        StatexDN            <= WRITE_READY;
    ELSE
        -- write address to system status
        -- TODO
        StatexDN            <= WRITE_READY;
    END IF;

-- ************
-- READ_ADDRESS
-- ************
WHEN READ_ADDRESS =>
    IF (OffsetxDP = 0) THEN
        -- first part of address read
        Address1xDN         <= HI_ReadDataxD;
        HI_ReadDataxACK     <= '1';
        OffsetxDN           <= OffsetxDP + 1;
    ELSIF ( (OffsetxDP = 1) AND (HI_ReadDataxE = '1') ) THEN
        -- second par of address can be read
        Address2xDN         <= HI_ReadDataxD;
        HI_ReadDataxACK     <= '1';
        OffsetxDN           <= OffsetxDP + 1;
    ELSIF (OffsetxDP = 2) THEN
        -- decide from which raid0 to read
        IF (StatusHDxDP(1 downto 0) = "11") THEN
            -- read from raid_master
            HI_ReadDataxACK <= '1';
```

```
            OffsetxDN       <= (others => '0');
            StatexDN        <= SEND_MASTER_ADDRESS;
        ELSIF ( (StatusLinkxDP = '1') AND (StatusHDxDP(3 downto 2) = "11") ) THEN
            -- read from raid_slave
            HI_ReadDataxACK <= '1';
            OffsetxDN       <= (others => '0');
            StatexDN        <= SEND_SLAVE_ADDRESS;
        ELSE
            -- can not read
            HI_ReadDataxERR <= '1';
        END IF;
    ELSE
        -- stay in state
        StatexDN            <= StatexDP;
    END IF;

-- ******************
-- SEND_MASTER_ADDRESS
-- ******************
WHEN SEND_MASTER_ADDRESS =>
    IF (OffsetxDP = 0) THEN
        IF ( (Raid_WriteDataxACK = '1') AND (Raid_WriteDataxERR = '0') ) THEN
            -- first part of address read
            OffsetxDN           <= OffsetxDP + 1;
            Raid_WriteDataxD    <= Address2xDP;
            Raid_WriteDataxE    <= '1';
        ELSE
            -- send first part of address
            Raid_WriteDataxD    <= Address1xDP;
            Raid_WriteDataxE    <= '1';
        END IF;
    ELSIF (OffsetxDP = 1) THEN
        IF ( (Raid_WriteDataxACK = '1') AND (Raid_WriteDataxERR = '0') ) THEN
            -- second part of address read
            OffsetxDN           <= OffsetxDP + 1;
            StatexDN            <= READ_MASTER;
        ELSE
            -- send second part of address
            Raid_WriteDataxD    <= Address2xDP;
            Raid_WriteDataxE    <= '1';
        END IF;
    ELSE
        -- should not be here!
        StatexDN                <= StatexDP;
    END IF;

-- ***********
-- READ_MASTER
-- ***********
WHEN READ_MASTER =>
    IF (StatusHDxDP(1 downto 0) < "11") THEN
        -- Harddisk M0 or M1 has a failure
        OffsetxDN           <= (others => '0');
        StatexDN            <= READ_ERROR;
    ELSIF (Raid_ReadDataxE = '1') THEN
        -- read data from master
        DataxDN             <= Raid_ReadDataxD;
        OffsetxDN           <= OffsetxDP + 1;
        StatexDN            <= READ_MASTER_READY;
    ELSE
        StatexDN            <= StatexDP;
    END IF;

-- *****************
-- READ_MASTER_READY
-- *****************
WHEN READ_MASTER_READY =>
```

```
    IF ( (HI_WriteDataxACK = '1') AND (HI_WriteDataxERR = '0') ) THEN
        -- data written to host master
        Raid_ReadDataxACK   <= '1';
        IF (OffsetxDP = 128) THEN
            StatexDN        <= IDLE;
        ELSE
            StatexDN        <= READ_MASTER;
        END IF;
    ELSE
        -- wait until HI has read data
        HI_WriteDataxD      <= DataxDP;
        HI_WriteDataxE      <= '1';
    END IF;


-- ******************
-- SEND_SLAVE_ADDRESS
-- ******************
WHEN SEND_SLAVE_ADDRESS =>
    IF (OffsetxDP = 0) THEN
        IF (Link_WriteDataxACK = '1') THEN
            -- first part of address read
            OffsetxDN           <= OffsetxDP + 1;
            Link_WriteDataxD    <= Address2xDP;
        ELSIF (Link_WriteDataxERR = '1') THEN
            -- link error
            StatusLinkxDN       <= '0';
            StatexDN            <= READ_ERROR;
        ELSE
            -- send first part of address
            Link_WriteDataxD    <= Address1xDP;
            Link_WriteDataxE    <= '1';
        END IF;
    ELSIF (OffsetxDP = 1) THEN
        IF (Link_WriteDataxACK = '1') THEN
            -- second part of address read
            OffsetxDN           <= OffsetxDP + 1;
            StatexDN            <= READ_SLAVE;
        ELSIF (Link_WriteDataxERR = '1') THEN
            -- link error
            StatusLinkxDN       <= '0';
            StatexDN            <= READ_ERROR;
        ELSE
            -- send second part of address
            Raid_WriteDataxD    <= Address2xDP;
            Raid_WriteDataxE    <= '1';
        END IF;
    ELSE
        -- should not be here!
        StatexDN                <= StatexDP;
    END IF;

-- ***********
-- READ_SLAVE
-- ***********
WHEN READ_SLAVE =>
    IF (StatusHDxDP(3 downto 2) < "11") THEN
        -- Harddisk S0 or S1 has a failure
        OffsetxDN           <= (others => '0');
        StatexDN            <= READ_ERROR;
    ELSIF (LINK_ReadDataxE = '1') THEN
        -- read data from slave
        DataxDN             <= Link_ReadDataxD;
        OffsetxDN           <= OffsetxDP + 1;
        StatexDN            <= READ_SLAVE_READY;
    ELSE
        StatexDN            <= StatexDP;
```

```
                END IF;

            -- ****************
            -- READ_SLAVE_READY
            -- ****************
            WHEN READ_SLAVE_READY =>
                IF ( (Link_WriteDataxACK = '1') AND (Link_WriteDataxERR = '0') ) THEN
                    -- data written to host master
                    Link_ReadDataxACK   <= '1';
                    IF (OffsetxDP = 128) THEN
                        StatexDN        <= IDLE;
                    ELSE
                        StatexDN        <= READ_SLAVE;
                    END IF;
                ELSE
                    -- wait until Link has read data
                    Link_WriteDataxD    <= DataxDP;
                    Link_WriteDataxE    <= '1';
                END IF;

            -- **********
            -- READ_ERROR
            -- **********
            WHEN READ_ERROR =>
                -- do error handling
                StatexDN    <= IDLE;



        END CASE;


    end process logic_inc;  -- combinatorial process


    -- ***************************************************************************************
    -- *                                   SEQUENTIAL PROCESS                                *
    -- ***************************************************************************************
    -- purpose: update registers (counter)
    reg_update : process ( RST, CLK, OffsetxDN, StatexDN, Address1xDN, Address2xDN, StatusHDxDN,
                           StatusLinkxDN)
    begin       -- process reg_update

        IF (RST = '0') THEN                         -- asynchronous reset (active low)
            OffsetxDP       <= (others => '0');
            DataxDP         <= (others => '0');
            Address1xDP     <= (others => '0');
            Address2xDP     <= (others => '0');
            StatexDP        <= RESET;
            StatusHDxDP     <= (others => '1');
            StatusLinkxDP   <= '0';
        ELSIF ( (CLK'event) AND (CLK = '1') ) THEN  -- rising clock edge
            OffsetxDP       <= OffsetxDN;
            StatexDP        <= StatexDN;
            DataxDP         <= DataxDN;
            Address1xDP     <= Address1xDN;
            Address2xDP     <= Address2xDN;
            StatusHDxDP     <= StatusHDxDN;
            StatusLinkxDP   <= StatusLinkxDN;
        END IF;

    end process reg_update;

end Behavioral;
```

8.4 FIFO

```
----------------------------------------------------------------------------------------------------
--  Project      : NAS
--  Part:        : SATA Controller
--  Filename     : sata_raid_fifo.vhd
--  Purpose      : FIFO
--  Responsible  : David H. Gehring, Beat Schilbach
----------------------------------------------------------------------------------------------------
--  Compiler & Synthesis Settings:
--
----------------------------------------------------------------------------------------------------
--  Description:
--  This file implements a fifo (uses an ip-core)
--
----------------------------------------------------------------------------------------------------
--  Limitations & Assumptions:
--
----------------------------------------------------------------------------------------------------
--  Known Errors:
--
----------------------------------------------------------------------------------------------------
--  History Of Change:
----------------------------------------------------------------------------------------------------
--  Version | Date     | Author              | Remarks
--  0.1     | 11.06.04 | David H. Gehring    | File created
----------------------------------------------------------------------------------------------------

library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.STD_LOGIC_ARITH.ALL;
    use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- library ADV;
--  use ADV.ADV_PKG.all;
--  use ADV.ADV_COMPONENTS_PKG.all;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--  library UNISIM;
--  use UNISIM.VCOMPONENTS.all;

entity sata_raid_fifo is
    Port(   -- Global Signals
            RST             : in  std_logic;                        -- Asynchronous RESET
            CLK             : in  std_logic;                        -- CLOCK
            -- Port Signals
            DataInxD        : in  std_logic_vector(31 downto 0);    -- Data IN
            DataInxE        : in  std_logic;                        -- Data IN Enable
            DataInxACK      : out std_logic;                        -- Data In Acknowledge
            DataInxERR      : out std_logic;                        -- Data In Error
            DataOutxD       : out std_logic_vector(31 downto 0);    -- Data OUT
            DataOutxE       : in  std_logic;                        -- Data OUT Enable
            DataOutxACK     : out std_logic;                        -- Data OUT Acknowledge
            DataOutxERR     : out std_logic;                        -- Data OUT Error
            FifoIsEmptyxE   : out std_logic;                        -- FIFO is empty
            FifoIsFullxE    : out std_logic                         -- FIFO is full
        );
end sata_raid_fifo;


architecture Behavioral of sata_raid_fifo is

    -- ***********************************************************************************************
    -- *                                     INTERNAL SIGNALS                                        *
    -- ***********************************************************************************************
```

```
    -- ***********************************************************************************************
    -- *                                          COMPONENTS                                         *
    -- ***********************************************************************************************
    component sata_raid_fifo_generated
        port (
                clk     : in  std_logic;
                sinit   : in  std_logic;
                din     : in  std_logic_vector(31 downto 0);
                wr_en   : in  std_logic;
                rd_en   : in  std_logic;
                dout    : out std_logic_vector(31 downto 0);
                full    : out std_logic;
                empty   : out std_logic;
                rd_ack  : out std_logic;
                wr_ack  : out std_logic;
                rd_err  : out std_logic;
                wr_err  : out std_logic
            );
    end component;


begin

    -- ***********************************************************************************************
    -- *                                       MAP CONNECTIONS                                       *
    -- ***********************************************************************************************

    inst_sata_raid_fifo_generated: sata_raid_fifo_generated
        port map(

                    CLK             => CLK,
                    SINIT           => RST,
                    DIN             => DataInxD,
                    DOUT            => DataOutxD,
                    WR_EN           => DataInxE,
                    RD_EN           => DataOutxE,
                    WR_ACK          => DataOutxACK,
                    RD_ACK          => DataInxACK,
                    WR_ERR          => DataOutxERR,
                    RD_ERR          => DataInxERR,
                    FULL            => FifoIsFullxE,
                    EMPTY           => FifoIsEmptyxE
                );

end Behavioral;
```

8.5 RAID0

```
----------------------------------------------------------------------------------------------------
--  Project      : NAS
--  Part:        : SATA Controller
--  Filename     : sata_raid_raid0.vhd
--  Purpose      : RAID 0 Controller
--  Responsible  : David H. Gehring, Beat Schilbach
----------------------------------------------------------------------------------------------------
--  Compiler & Synthesis Settings:
--
----------------------------------------------------------------------------------------------------
--  Description:
--  This file implements the raid0 controller
--
----------------------------------------------------------------------------------------------------
--  Limitations & Assumptions:
```

```
--
-------------------------------------------------------------------------------------------------
--  Known Errors:
--
-------------------------------------------------------------------------------------------------
--  History Of Change:
-------------------------------------------------------------------------------------------------
--  Version | Date     | Author               | Remarks
--  0.1     | 11.06.04 | David H. Gehring     | File created
-------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity sata_raid_raid0 is
    Port (  -- Entity-Global Signals
                RST                 : in  std_logic;                        -- Asynchronous RESET
                CLK                 : in  std_logic;                        -- CLOCK

                -- Signals to FSM (FSM_Master)
                FSM_WriteDataxD     : out std_logic_vector(31 downto 0);    -- Data OUT
                FSM_WriteDataxE     : out std_logic;                        -- Data OUT Enable
                FSM_WriteDataxACK   : in  std_logic;                        -- Data Out Acknowledge
                FSM_WriteDataxERR   : in  std_logic;                        -- Data OUT Error
                FSM_ReadDataxD      : in  std_logic_vector(31 downto 0);    -- Data IN
                FSM_ReadDataxE      : in  std_logic;                        -- Data IN Enable
                FSM_ReadDataxACK    : out std_logic;                        -- Data IN Acknowledge
                FSM_ReadDataxERR    : out std_logic                         -- Data IN Error

                -- SATA0
                SATA0_WriteDataxD   : out std_logic_vector(31 downto 0);
                SATA0_WriteDataxE   : out std_logic;
                SATA0_WriteDataxACK : in  std_logic;
                SATA0_WriteDataxERR : in  std_logic;
                SATA0_ReadDataxD    : in  std_logic_vector(31 downto 0);
                SATA0_ReadDataxE    : in  std_logic;
                SATA0_ReadDataxACK  : out std_logic;
                SATA0_ReadDataxERR  : out std_logic;

                -- SATA1
                SATA1_WriteDataxD   : out std_logic_vector(31 downto 0);
                SATA1_WriteDataxE   : out std_logic;
                SATA1_WriteDataxACK : in  std_logic;
                SATA1_WriteDataxERR : in  std_logic;
                SATA1_ReadDataxD    : in  std_logic_vector(31 downto 0);
                SATA1_ReadDataxE    : in  std_logic;
                SATA1_ReadDataxACK  : out std_logic;
                SATA1_ReadDataxERR  : out std_logic;

                -- Signals to FIFO_Master
                FIFO_ReadDataxD     : in  std_logic_vector(31 downto 0);
                FIFO_ReadDataxE     : out std_logic;
                FIFO_ReadDataxERR   : in  std_logic;
                FIFO_ReadDataxACK   : in  std_logic;
                FIFO_IsEmptyxE      : in  std_logic;                        -- FIFO is empty

           );
end sata_raid_raid0;


architecture Behavioral of sata_raid_raid0 is
    -- ***********************************************************************************
    -- *                                  INTERNAL SIGNALS                               *
    -- ***********************************************************************************
    signal OffsetxDP, OffsetxDN     : std_logic_vector(7 downto 0);     -- data block offset
```

```
    signal AddressxDP, AddressxDN   : std_logic_vector(47 downto 0);    -- address register
    signal DataxDP, DataxDN         : std_logic_vector(31 downto 0);    -- data register
    signal HarddiskxDP, HarddiskxDN : std_logic;                        -- on which HD to operate
    signal StatusHDxDP, StatusHDxDN : std_logic_vector(1 downto 0);     -- harddisk status
    type state_values is (IDLE, GET_DATA , WRITE_READY, WRITE_HD0, WRITE_HD1, WRITE_ERROR,
                      READ_ADDRESS, SEND_ADDRESS_HD0, SEND_ADDRESS_HD1, READ_HD0, READ_HD1,
                      SEND_DATA, READ_ERROR);
    signal StatexDP, StatexDN       : state_values;                     -- fsm state register
    -- sets which bit should be addressed for raid0 addressing: constant for Stripe-Size (= 2^Stripe)
    constant STRIPE             : integer := 40;                        -- ( 1 < STRIPE < 47)

begin
    -- ***********************************************************************************************
    -- *                                THE LOGIC COMBINATORIAL PROCESS                              *
    -- ***********************************************************************************************
    logic_inc : process (   OffsetxDP, AddressxDP, StatexDP, DataxDP, HarddiskxDP, StatusHDxDP,
                      FIFO_IsEmptyxE, FIFO_ReadDataxACK, FIFO_ReadDataxERR, FIFO_ReadDataxD,
                      SATA0_WriteDataxACK, SATA0_WriteDataxERR, SATA0_ReadDataxD, SATA0_ReadDataxE,
                      SATA1_WriteDataxACK, SATA1_WriteDataxERR, SATA1_ReadDataxD, SATA1_ReadDataxE,
                      FSM_WriteDataxACK, FSM_WriteDataxERR, FSM_ReadDataxD, FSM_ReadDataxE
                  )
    BEGIN

        -- default behaviour
        DataxDN     <= DataxDP;
        StatexDN        <= StatexDP;
        OffsetxDN       <= OffsetxDP;
        AddressxDN  <= AddressxDP;
        HarddiskxDN <= HarddiskxDP;
        StatusHDxDN <= StatusHDxDP;

        -- default output values
        FIFO_ReadDataxE <= '0';
        SATA0_WriteDataxE   <= '0';
        SATA0_WriteDataxD   <= (others => '0');
        SATA0_ReadDataxACK  <= '0';
        SATA0_ReadDataxERR  <= '0';
        SATA1_WriteDataxE   <= '0';
        SATA1_WriteDataxD   <= (others => '0');
        SATA1_ReadDataxACK  <= '0';
        SATA1_ReadDataxERR  <= '0';
        FSM_WriteDataxE <= '0';
        FSM_WriteDataxD <= (others => '0');
        FSM_ReadDataxACK    <= '0';
        FSM_ReadDataxERR    <= '0';

        CASE StatexDP IS

            -- ****
            -- IDLE
            -- ****
            WHEN IDLE =>
                IF (FIFO_IsEmptyxE = '0') THEN
                    -- GOTO WRITE cycle if data are present in the fifo
                    StatexDN                <= GET_DATA;
                    FIFO_ReadDataxE     <= '1';
                    -- reset local address and offset counter
                    AddressxDN          <= (others => '0');
                    OffsetxDN               <= (others => '0');
                ELSIF (FSM_ReadDataxE = '1') THEN
                    -- GOTO READ cycle if read request was received
                    StatexDN                <= READ_ADDRESS;
                    -- reset local address and offset counter
                    AddressxDN          <= (others => '0');
                    OffsetxDN               <= (others => '0');
                ELSE
                    -- stay in IDLE per default
```

```
        StatexDN                <= StatexDP;
    END IF;

-- ********
-- GET_DATA
-- ********
WHEN GET_DATA =>
    -- Data must be read within this clock (only problem: no data @ fifo)
    IF ( (FIFO_ReadDataxACK = '1') AND (FIFO_ReadDataxERR = '0') ) THEN
        -- Data are to be read from FIFO
        DataxDN             <= FIFO_ReadDataxD;
        StatexDN                <= WRITE_READY;
        -- do address manipulation, if address is read compleately
        IF (OffsetxDP = 2) THEN
            IF ( AddressxDP(STRIPE) = '0') THEN
                HarddiskxDN     <= '0';
            ELSE
                HarddiskxDN     <= '1';
            END IF;
            -- shorten address
            AddressxDN(47 downto 0)     <= '0' & AddressxDP(47 downto (STRIPE + 1))
                                           & AddressxDP((STRIPE - 1) downto 0);
        END IF;
    ELSE
        -- stay in GET_DATA until data are available in fifo
        FIFO_ReadDataxE     <= '1';
        StatexDN                <= StatexDP;
    END IF;

-- ***********
-- WRITE_READY
-- ***********
WHEN WRITE_READY =>
    IF    (OffsetxDP = 0) THEN
        -- first 32bit are address
        AddressxDN(31 downto 0) <= DataxDP(31 downto 0);
        OffsetxDN               <= OffsetxDP + 1;
        StatexDN                <= GET_DATA;
        -- request new data befor transition to GET_DATA
        FIFO_ReadDataxE     <= '1';
    ELSIF ( OffsetxDP = 1) THEN
        -- second 16bit (out of 32bit) are address
        AddressxDN(47 downto 32)    <= DataxDP(15 downto 0);
        OffsetxDN               <= OffsetxDP + 1;
        StatexDN                <= GET_DATA;
        -- request new data befor transition to GET_DATA
        FIFO_ReadDataxE     <= '1';
    ELSIF (StatusHDxDP < "11") THEN
        -- Harddisk error -> empty FIFO
        StatexDN                <= GET_DATA;
        FIFO_ReadDataxE     <= '1';
    ELSE
        -- now 128x32bit data are coming
        -- Chose to which harddisk to write
        IF (HarddiskxDP = '0') THEN
            IF (StatusHDxDP(0) = '0') THEN
                -- Harddisk0 is broken
                StatexDN            <= GET_DATA;
                -- request new data befor transition to GET_DATA
                FIFO_ReadDataxE     <= '1';
            ELSE
                -- write to HD0
                SATA0_WriteDataxD   <= DataxDP;
                SATA0_WriteDataxE   <= '1';
                OffsetxDN           <= OffsetxDP + 1;
                StatexDN                <= WRITE_HD0;
            END IF;
```

```
        ELSE
            IF  (StatusHDxDP(1) = '0') THEN
                -- Harddisk1 is broken
                StatexDN            <= GET_DATA;
                -- request new data befor transition to GET_DATA
                FIFO_ReadDataxE     <= '1';
            ELSE
                -- write to HD1
                SATA1_WriteDataxD   <= DataxDP;
                SATA1_WriteDataxE   <= '1';
                OffsetxDN           <= OffsetxDP + 1;
                StatexDN            <= WRITE_HD1;
            END IF;
        END IF;
    END IF;

-- *********
-- WRITE_HD0
-- *********
WHEN WRITE_HD0 =>
    -- wait until ack from HD0
    -- data are valid as long as no ack is received
    SATA0_WriteDataxD           <= DataxDP;
    SATA0_WriteDataxE           <= '1';
    IF ( SATA0_WriteDataxACK = '1') THEN
        IF (OffsetxDP = 130) THEN
            -- 128x32Bit Data + 2x32Bit ADDR are read
            StatexDN            <= IDLE;
        ELSE
            StatexDN            <= GET_DATA;
            FIFO_ReadDataxE <= '1';
        END IF;
    ELSIF ( SATA0_WriteDataxERR = '1') THEN
        StatusHDxDN(0)          <= '0';
        StatexDN                <= WRITE_ERROR;
    ELSE
        StatexDN                <= StatexDP;
    END IF;

-- *********
-- WRITE_HD1
-- *********
 WHEN WRITE_HD1 =>
    -- wait until ack from HD1
    -- data are valid as long as no ack is received
    SATA1_WriteDataxD   <= DataxDP;
    SATA1_WriteDataxE           <= '1';
    IF ( SATA1_WriteDataxACK = '1') THEN
        IF (OffsetxDP = 130) THEN
            -- 128x32Bit Data + 2x32Bit ADDR are read
            StatexDN            <= IDLE;
        ELSE
            StatexDN            <= GET_DATA;
            FIFO_ReadDataxE <= '1';
        END IF;
    ELSIF ( SATA1_WriteDataxERR = '1') THEN
        StatusHDxDN(1)          <= '0';
        StatexDN                <= WRITE_ERROR;
    ELSE
        StatexDN                <= StatexDP;
    END IF;

-- ***********
-- WRITE_ERROR
-- ***********
 WHEN WRITE_ERROR =>
    -- tell fsm master and wait for ok, afterwards:
```

```
    IF (OffsetxDP = 130) THEN
        -- (4096bit = 1 Block) + addr are read
        StatexDN                <= IDLE;
    ELSE
        -- return to cycle to delet data
        StatexDN                <= GET_DATA;
        FIFO_ReadDataxE     <= '1';
    END IF;

-- ************
-- READ_ADDRESS
-- ************
 WHEN READ_ADDRESS =>
    -- loop in state READ_READY until address is read (2 loops)
    IF (OffsetxDP = 0) THEN
        -- first 32bit are address
        AddressxDN(31 downto 0) <= FSM_ReadDataxD(31 downto 0);
        OffsetxDN               <= OffsetxDP + 1;
        FSM_ReadDataxACK        <= '1';
    ELSIF ( (OffsetxDP = 1) AND (FSM_ReadDataxE = '1') ) THEN
        -- second 16bit (out of 32bit) are address
        AddressxDN(47 downto 32)    <= FSM_ReadDataxD(15 downto 0);
        OffsetxDN               <= OffsetxDP + 1;
        FSM_ReadDataxACK        <= '1';
    ELSIF (OffsetxDP = 2) THEN
        -- decide from which hd to read
        IF ( AddressxDP(STRIPE) = '0') THEN
            -- read from HD0
            HarddiskxDN     <= '0';
            StatexDN            <= SEND_ADDRESS_HD0;
        ELSE
            -- read from HD1
            HarddiskxDN     <= '1';
            StatexDN            <= SEND_ADDRESS_HD1;
        END IF;
        -- shorten address and reset Offset
        AddressxDN(47 downto 0) <= '0' & AddressxDP(47 downto (STRIPE + 1))
                                   & AddressxDP((STRIPE - 1) downto 0);
        OffsetxDN               <= (others => '0');
    ELSE
        -- default: wait in state
        StatexDN                <= StatexDP;
    END IF;

-- ****************
-- SEND_ADDRESS_HD0
-- ****************
WHEN SEND_ADDRESS_HD0 =>
    -- send ADDRESS to SATA
    IF    (OffsetxDP = 0) THEN
        -- first part of address (total 48bit)
        SATA0_WriteDataxD       <= AddressxDP(31 downto 0);
        SATA0_WriteDataxE       <= '1';
        OffsetxDN               <= OffsetxDP + 1;
    ELSIF (OffsetxDP = 1) THEN
        IF (SATA0_WriteDataxACK = '1') THEN
            -- second part of address (total 48 bit)
            SATA0_WriteDataxD(15 downto 0) <= AddressxDP(47 downto 32);
            SATA0_WriteDataxE   <= '1';
            OffsetxDN           <= OffsetxDP + 1;
        ELSE
            -- wait until ack from sata-controller
            SATA0_WriteDataxD   <= AddressxDP(31 downto 0);
            SATA0_WriteDataxE   <= '1';
        END IF;
    ELSIF (OffsetxDP = 2) THEN
        IF (SATA0_WriteDataxACK = '1') THEN
```

```
            -- signal with SATA0_ReadDataxACK to READ DATA
            StatexDN                <= READ_HD0;
            SATA0_ReadDataxACK      <= '1';
        ELSE
            SATA0_WriteDataxD(15 downto 0) <= AddressxDP(47 downto 32);
            SATA0_WriteDataxE   <= '1';
        END IF;
    ELSE
        StatexDN                <= StatexDP;
    END IF;

-- ****************
-- SEND_ADDRESS_HD1
-- ****************
WHEN SEND_ADDRESS_HD1 =>
    -- send ADDRESS to SATA
    IF    (OffsetxDP = 0) THEN
        -- first part of address (total 48bit)
        SATA1_WriteDataxD       <= AddressxDP(31 downto 0);
        SATA1_WriteDataxE       <= '1';
        OffsetxDN               <= OffsetxDP + 1;
    ELSIF (OffsetxDP = 1) THEN
        IF (SATA1_WriteDataxACK = '1') THEN
            -- second part of address (total 48 bit)
            SATA1_WriteDataxD(15 downto 0) <= AddressxDP(47 downto 32);
            SATA1_WriteDataxE   <= '1';
            OffsetxDN           <= OffsetxDP + 1;
        ELSE
            -- wait until ack from sata-controller
            SATA1_WriteDataxD   <= AddressxDP(31 downto 0);
            SATA1_WriteDataxE   <= '1';
        END IF;
    ELSIF (OffsetxDP = 2) THEN
        IF (SATA1_WriteDataxACK = '1') THEN
            -- signal with SATA1_ReadDataxACK to READ DATA
            StatexDN                <= READ_HD1;
            SATA1_ReadDataxACK      <= '1';
        ELSE
            SATA1_WriteDataxD(15 downto 0) <= AddressxDP(47 downto 32);
            SATA1_WriteDataxE   <= '1';
        END IF;
    ELSE
        StatexDN                <= StatexDP;
    END IF;

-- ********
-- READ_HD0
-- ********
 WHEN READ_HD0 =>
    -- read data from hd0
    IF (OffsetxDP = 130) THEN
        -- (128x4096bit = 1 Block) + (48bit = addr) are read
        StatexDN                <= IDLE;
    ELSIF (SATA0_ReadDataxE = '1') THEN
        -- data are read from sata to be sent to fsm
        DataxDN             <= SATA0_ReadDataxD;
        SATA0_ReadDataxACK      <= '1';
        StatexDN                <= SEND_DATA;
    ELSE
        StatexDN                <= StatexDP;
    END IF;

-- ********
-- READ_HD1
-- ********
 WHEN READ_HD1 =>
    -- read data from hd1
```

```
            IF (OffsetxDP = 130) THEN
                -- (4096bit = 1 Block) + addr are read
                StatexDN               <= IDLE;
            ELSIF (SATA1_ReadDataxE = '1') THEN
                -- data are read from sata to be sent to fsm
                DataxDN             <= SATA1_ReadDataxD;
                SATA1_ReadDataxACK      <= '1';
                StatexDN               <= SEND_DATA;
            ELSE
                StatexDN               <= StatexDP;
            END IF;

        -- *********
        -- SEND_DATA
        -- *********
         WHEN SEND_DATA =>
            -- write data to FSM
            FSM_WriteDataxD         <= DataxDP;
            FSM_WriteDataxE         <= '1';
            IF (FSM_WriteDataxAck = '1') THEN
                -- update offset-counter
                OffsetxDN           <= OffsetxDP + 1;
                IF (HarddiskxDP = '0') THEN
                    StatexDN            <= READ_HD0;
                ELSE
                    StatexDN            <= READ_HD1;
                END IF;
            ELSIF (FSM_WriteDataxERR = '1') THEN
                StatexDN               <= READ_ERROR;
            ELSE
                StatexDN               <= StatexDP;
            END IF;

        -- **********
        -- READ_ERROR
        -- **********
         WHEN READ_ERROR =>
            -- tell others
            StatexDN                   <= IDLE;

    END CASE;

end process logic_inc;  -- combinatorial process


-- ***************************************************************************************
-- *                                   SEQUENTIAL PROCESS                                *
-- ***************************************************************************************
-- purpose: update registers (counter)
reg_update : process (RST, CLK, OffsetxDN, AddressxDN, StatexDN, HarddiskxDN)
begin                                       -- process reg_update

    IF (RST = '0') THEN                     -- asynchronous reset (active low)
        OffsetxDP       <= (others => '0');
        AddressxDP  <= (others => '0');
        DataxDP     <= (others => '0');
        StatusHDxDP <= (others => '1');
        StatexDP        <= IDLE;
        HarddiskxDP <= '0';
    ELSIF ( (CLK'event) AND (CLK = '1') ) THEN  -- rising clock edge
        OffsetxDP       <= OffsetxDN;
        AddressxDP  <= AddressxDN;
        StatexDP        <= StatexDN;
        HarddiskxDP <= HarddiskxDN;
        DataxDP     <= DataxDN;
        StatusHDxDP <= StatusHDxDN;
    END IF;
```

```
    end process reg_update;

end Behavioral;
```

8.6 Link (Dummy)

```
----------------------------------------------------------------------------------------------------
--  Project      : NAS
--  Part:        : SATA Controller
--  Filename     : sata_raid_link.vhd
--  Purpose      : Link
--  Responsible  : David H. Gehring, Beat Schilbach
----------------------------------------------------------------------------------------------------
--  Compiler & Synthesis Settings:
--
----------------------------------------------------------------------------------------------------
--  Description:
--  This File establishes the connection to the neighbour systems
--
----------------------------------------------------------------------------------------------------
--  Limitations & Assumptions:
--
----------------------------------------------------------------------------------------------------
--  Known Errors:
--
----------------------------------------------------------------------------------------------------
--  History Of Change:
----------------------------------------------------------------------------------------------------
--  Version | Date     | Author               | Remarks
--  0.1     | 11.06.04 | David H. Gehring     | File created
----------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity sata_raid_link is
    Port (  -- Entity-Global Signals
            RST                 : in  std_logic;                        -- Asynchronous RESET
            CLK                 : in  std_logic;                        -- CLOCK

            -- Connection to FSM
            FSM_WriteDataxD     : out std_logic_vector(31 downto 0);    -- Data OUT
            FSM_WriteDataxE     : out std_logic;                        -- Data OUT Enable
            FSM_WriteDataxACK   : in  std_logic;                        -- Data OUT Acknowledge
            FSM_WriteDataxERR   : in  std_logic;                        -- Data OUT Error
            FSM_ReadDataxD      : in  std_logic_vector(31 downto 0);    -- DATA IN
            FSM_ReadDataxE      : in  std_logic;                        -- DATA IN Enable
            FSM_ReadDataxACK    : out std_logic;                        -- DATA IN Acknowledge
            FSM_ReadDataxERR    : out std_logic                         -- DATA IN Error

        );
end sata_raid_link;

architecture Behavioral of sata_raid_link is
    -- ***************************************************************************************
    -- *                                   INTERNAL SIGNALS                                  *
    -- ***************************************************************************************
    signal DataxDP, DataxDN         : std_logic_vector(31 downto 0);    -- data register
    type state_values is (IDLE, SEND_DATA, RECEIVE_DATA);               -- fsm states
    signal StatexDP, StatexDN       : state_values;                     -- fsm state register
begin
```

```
-- ***********************************************************************************************
-- *                                THE LOGIC COMBINATORIAL PROCESS                              *
-- ***********************************************************************************************
logic_inc : process ( DataxDP, StatexDP, FSM_ReadDataxE, FSM_ReadDataxD, FSM_WriteDataxACK,
                      FSM_WriteDataxERR )
BEGIN

    -- default behaviour
    DataxDN             <= DataxDP;
    StatexDN            <= StatexDP;

    -- default values
    FSM_WriteDataxD     <= (others => '0');
    FSM_WriteDataxE     <= '0';
    FSM_ReadDataxACK    <= '0';
    FSM_ReadDataxERR    <= '0';

    CASE StatexDP IS

        -- ****
        -- IDLE
        -- ****
        WHEN IDLE =>
            IF (FSM_ReadDataxE = '1') THEN
                StatexDN            <= SEND_DATA;
            ELSIF ( (FSM_WriteDataxACK = '1') AND (FSM_WriteDataxERR = '0') ) THEN
                StatexDN            <= RECEIVE_DATA;
            ELSE
                StatexDN            <= StatexDP;
            END IF;

        -- *********
        -- SEND_DATA
        -- *********
        WHEN SEND_DATA =>
            DataxDN                 <= FSM_ReadDataxD;
            FSM_ReadDataxACK        <= '1';
            StatexDN                <= IDLE;

        -- ************
        -- RECEIVE_DATA
        -- ************
        WHEN RECEIVE_DATA =>
            FSM_WriteDataxE         <= '1';
            FSM_WriteDataxD         <= DataxDP;
            StatexDN                <= IDLE;

    END CASE;


end process logic_inc;  -- combinatorial process


-- ***********************************************************************************************
-- *                                    SEQUENTIAL PROCESS                                       *
-- ***********************************************************************************************
-- purpose: update registers (counter)
reg_update : process (RST, CLK, DataxDN, StatexDN)
begin                                           -- process reg_update

    IF (RST = '0') THEN                         -- asynchronous reset (active low)
        DataxDP     <= (others => '0');
        StatexDP    <= IDLE;
    ELSIF ( (CLK'event) AND (CLK = '1') ) THEN      -- rising clock edge
        DataxDP     <= DataxDN;
        StatexDP    <= StatexDN;
    END IF;
```

```
    end process reg_update;

end Behavioral;
```

8.7 SATA (Dummy)

```
----------------------------------------------------------------------------------------------------
--  Project      : NAS
--  Part:        : SATA Controller
--  Filename     : sata_raid_sata.vhd
--  Purpose      : Serial ATA Harddisk core
--  Responsible  : David H. Gehring, Beat Schilbach
----------------------------------------------------------------------------------------------------
--  Compiler & Synthesis Settings:
--
----------------------------------------------------------------------------------------------------
--  Description:
--  This File implements the connection to the serial ata harddisks
--
----------------------------------------------------------------------------------------------------
--  Limitations & Assumptions:
--
----------------------------------------------------------------------------------------------------
--  Known Errors:
--
----------------------------------------------------------------------------------------------------
--  History Of Change:
----------------------------------------------------------------------------------------------------
--  Version | Date     | Author               | Remarks
--  0.1     | 11.06.04 | David H. Gehring     | File created
----------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity sata_raid_sata is
Port    (   -- Entity-Global Signals
            RST                 : in  std_logic;                        -- Asynchronous RESET
            CLK                 : in  std_logic;                        -- CLOCK

            -- Connection to Raid
            Raid_WriteDataxD    : out std_logic_vector(31 downto 0);    -- Data OUT
            Raid_WriteDataxE    : out std_logic;                        -- Data OUT Enable
            Raid_WriteDataxACK  : in  std_logic;                        -- Data Out Acknowledge
            Raid_WriteDataxERR  : in  std_logic;                        -- Data Out Error
            Raid_ReadDataxD     : in  std_logic_vector(31 downto 0);    -- Data IN
            Raid_ReadDataxE     : in  std_logic;                        -- Data IN Enable
            Raid_ReadDataxACK   : out std_logic;                        -- Data IN Acknowledge
            Raid_ReadDataxERR   : out std_logic                         -- Data IN Error
        );
end sata_raid_sata;

architecture Behavioral of sata_raid_sata is
    -- ***************************************************************************************
    -- *                                 INTERNAL SIGNALS                                    *
    -- ***************************************************************************************
    signal DataxDP, DataxDN         : std_logic_vector(31 downto 0);
    -- FSM-States
    type state_values is (IDLE, READ_DATA, WRITE_DATA);
    signal StatexDP, StatexDN       : state_values;
begin
```

```
-- ***********************************************************************************************
-- *                                THE LOGIC COMBINATORIAL PROCESS                              *
-- ***********************************************************************************************
logic_inc : process (   DataxDP, StatexDP, Raid_WriteDataxACK, Raid_WriteDataxERR,
                        Raid_ReadDataxE, Raid_ReadDataxD
                    )
BEGIN

    -- default behaviour
    DataxDN             <= DataxDP;
    StatexDN            <= StatexDP;

    -- default values
    Raid_ReadDataxACK   <= '0';
    Raid_ReadDataxERR   <= '0';
    Raid_WriteDataxE    <= '0';
    Raid_WriteDataxD    <= (others => '0');

    CASE StatexDP IS

        -- ****
        -- IDLE
        -- ****
        WHEN IDLE =>
            IF (Raid_ReadDataxE = '1') THEN
                DataxDN             <= Raid_ReadDataxD;
                StatexDN            <= READ_DATA;
            ELSIF ( (Raid_WriteDataxACK = '1') AND (Raid_WriteDataxERR = '0') ) THEN
                StatexDN            <= WRITE_DATA;
            ELSE
                StatexDN            <= StatexDP;
            END IF;

        -- *********
        -- READ_DATA from RAID0
        -- *********
        WHEN READ_DATA =>
            Raid_ReadDataxACK       <= '1';
            StatexDN                <= IDLE;

        -- **********
        -- WRITE_DATA to RAID0
        -- **********
        WHEN WRITE_DATA =>
            Raid_WriteDataxE        <= '1';
            Raid_WriteDataxD        <= DataxDP;
            StatexDN                <= IDLE;

    END CASE;

end process logic_inc;  -- combinatorial process

-- ***********************************************************************************************
-- *                                    SEQUENTIAL PROCESS                                       *
-- ***********************************************************************************************
-- purpose: update registers (counter)
reg_update : process (RST, CLK, DataxDN, StatexDN)
begin                                           -- process reg_update

    IF (RST = '0') THEN                         -- asynchronous reset (active low)
        DataxDP     <= (others => '0');
        StatexDP    <= IDLE;
    ELSIF ( (CLK'event) AND (CLK = '1') ) THEN  -- rising clock edge
        DataxDP     <= DataxDN;
        StatexDP    <= StatexDN;
    END IF;
```

```
    end process reg_update;

end Behavioral;
```

Teil IV

Verzeichnisse

Abbildungsverzeichnis

1 Möglicher Aufbau VI

2.1 Beispiel Striping 7
2.2 Mirroring 7
2.3 Striping mit verteiltem Parity 7

4.1 Einbettung des Systems 16
4.2 Variierbare Anzahl Festplatten 17
4.3 Grundidee des Systems 18
4.4 Systeme mit Switch 19
4.5 Systeme als Ring 20
4.6 FIFO im NVRAM 22
4.7 Flaschenhälse im System 24
4.8 Durchsatz einer Harddisk 26

5.1 Systemdesign 28

6.1 Das OPB-IPIF Modul 38
6.2 FSM RAID1 Controller 39
6.3 Lokalität des FIFO 44
6.4 Internes FIFO 45
6.5 Datenmüll im FIFO 46
6.6 Idee für Striping 48
6.7 FSM RAID0 Controller 50

Tabellenverzeichnis

1.1 Namenskonventionen 4

2.1 NAS 6

3.1 Ziele für System 11

4.1 Systemangaben 15
4.2 Übersicht lokale HD-Anordnung 18
4.3 Übersicht Skalierbarkeit 20
4.4 PATA vs. SATA vs. SCSI 21
4.5 Rocket I/O 25

5.1 Systemzustand 31
5.2 Error Recovery 33
5.3 System-Initialisierung 35

6.1 Zustände RAID1-Write 41
6.2 Zustände RAID1-Read 43
6.3 Code Harddisk-Status 51
6.4 Zustände RAID0-Write 52
6.5 Zustände RAID0-Read 53

Literaturverzeichnis

[1] Garth A. Gibson & Rodney Van Meter: *Network Attached Storage Architecture*, November 2000

[2] Adaptec Certified Storage Professional (ACSP) course manual, chapter 4: *What is RAID?*, www.adaptec.com (2004)

[3] Apple Xserve RAID: *RAID Basics*, www.apple.com/xserve/raid/ (2004)

[4] Xilinx, www.xilinx.com (2004)

[5] Microelectronics Design Center, dz.ee.ethz.ch (2004)

[6] Supercomputing Systems, 8005 Zürich, www.scs.ch

[7] Xilinx: *Synchronious FIFO v4.0*, März 2003, www.xilinx.com

[8] Serial ATA: *A Comparison with Ultra ATA Technology*, www.SerialATA.org

[9] Alan Robertson, *SuSE Labs*: *Linux-HA Heartbeat System Design*, linux-ha.org

[10] Mark Holland, Garth A. Gibson: *Fast, On-Line Failure Recovery in Redundant Disk Arrays*, 1993

[11] William V. Courtright, Gart A. Gibson: *Backward Error Recovery in Redundant Disk Arrays*, 1994

[12] Mark Holland, Garth A.Gibson: *Architectures and Algorithms for On-Line Failure Recovery in Redundant Disk Arrays*, Juli 1994

[13] Monta Vista, www.mvista.com

Index

Antwortzeit, 10
ASIC, 4
ATA
    Parallel-ATA, 4, 9, 20
    Serial-ATA, 4, 9, 21
Ausfallssicherheit, 19
Availability, 10

Bandbreite
    maximale, 25
Betriebssystem
    Flash, 27
    Komponente, 27
    RAM, 27

Crosstalk, 21

DAS, 5
Daten
    -integrität, 10
    Abgleich, 3
    Durchsatz, 10
    Redundanz, 3, 6
    Sicherheit, 6
    Speicherung, unterschiedliche, 32
    Transferraten, 10
    Verfügbarkeit, 6, 10
Diskkapazität, 5
Durchsatz
    SATA, PATA, 26

Error Recovery
    Backward, 33
    Forward, 33
    online, 33

FIFO
    Aufbau, 44
    Begriffserklärung, 3
    Datenspeicherung, 23
    Fehlerbehandlung, 32
    Signale, 45
    Systemstatus, 30
FPGA, 3, 8, 10, 15, 17
FSM
    Begriffserklärung, 3
    fürFIFO, 45
    Master, Komponente, 27
    Master-Kontroller, 39
    RAID0, 49

Harddisk
    Fehler, 29
    Fehler, Ausfall, 32
Host Interface
    Komponente, 27
    Schnittstelle, 38

Interoperabilität, 11
IP Core, 8, 38, 44
IP-Adresse, 19

KMU, III, 9, 16
Kontroller
    Fehler, 30
    Master, RAID1, 39
    RAID0, 47

Link
    Fehler, 30
    Komponenten, 29
Linux-HA, 23

MAC-Adresse, 19
Master, 18, 39

NAS, 5, 9
NFS, 10
NVRAM
    Adressen, 32
    allgemein, 3, 22
    Fehler, 30

FIFO, 44
Komponenten, 29
Systemstart, 35
Systemstatus, 30

OPB, 8, 38

PCB, 4
Performance, 3, 6, 10, 25

RAID
allgemein, 6, 17
Begriffserklärung, 3
Komponenten, 29
Mirroring, 6
Parity, 8
Striping, 6
RAM, 22, 44
Reconstruction
allgemein, 27
Begriffserklärung, 3
disk-orientiert, 34
Error Recovery, 33
Full Reconstruction, 32
stripe-orientiert, 34
Reliability, 6, 10
Ring, 19
Rocket I/O, 3, 8, 25

SAN, 5
SATA
Komponenten, 29
Schnittstelle
Host Interface, 38
Komponenten, 29
seriell, 23
zur Festplatte, 24
SCSI, 9, 21
Skalierbarkeit, 11, 19
Slave
allgemein, 18
Komponenten, 29
SMB, 10
SOC, 15
SoC, III
Software
Fehler, 30
Speicheradressen, 49
Speicherbereich, linearer, 48
Speicherplatz, 5, 44
Stripe, 6, 49
Striping Algorithmen, 48
System
Initialisierung, 35
Reset, 35
Systemstatus
allgemein, 30
Error Recovery, 33
Signale, 40
Systemstart, 35

Verbindung
Fehler, 31