Entwurf und Implementierung eines FireWire Interface in VHDL

Baccalaureatsarbeit

vorgelegt von:
Marcin Blaszkowski
Matrikelnummer: 5212026

Betreuer:
Dr. Andreas Mäder

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Technische Aspekte Multimodaler Systeme

Hamburg den 14. Juli 2004

Inhaltsverzeichnis

1 Einleitung **2**
1.1 Motivation 2
1.2 Übersicht über das System 2

2 Architektur der Schnittstelle **6**
2.1 Signale des ioFSM 8
2.1.1 Schnittstelle zu der NIOS-CPU 8
2.1.2 Schnittstelle zum Link Layer Controller (LLC) 9
2.1.3 Schnittstelle zum ISO Channel Filter 10
2.2 Signale des ISO Channel Filters 10
2.2.1 Schnittstelle zum Zeilenfilter 10
2.2.2 Schnittstelle zu der NIOS-CPU 11

3 Entwurf und Implementierung des ioFSM **12**
3.1 Der TSB12LV01B Link Layer Controller 12
3.1.1 Interner Aufbau 12
3.1.2 Interne Register - Speicherlayout des LLC 13
3.2 Die ioFSM-Komponente 18
3.2.1 Der Automatenentwurf 18
3.2.2 Interrupt-Handling des ioFSM 24
3.3 Der ISO Channel Filter 24

4 Zusammenfassung und Ausblick **26**

A VHDL-Quellcodes **27**
A.1 ioFSM 27
A.2 ISO Channel Filter 34
A.3 Adresskonstanten 36

Kapitel 1

Einleitung

1.1 Motivation

Diese Arbeit ist im Rahmen des Projektes „Entwurf eines hochintegrierten Embedded Systems“ entstanden. Das Ziel des Projektes war die Realisierung einer echtzeitfähigen Bildverarbeitungskomponente für das Sichtsystem eines intelligenten Service Roboters (siehe Abbildung 1.1). Damit sollte die CPU des Roboters von bestimmten Regelungsaufgaben (Fokussierung) und Bildvorverarbeitungsaufgaben (Datenreduktion, Merkmalsextraktion) entlastet werden. So sind zum Beispiel die Kameras des Roboters zwar von guter Qualität, verfügen aber über keine Autofokus-Funktion. Die Durchführung der Autofokussierung, durch die ohnehin schon stark ausgelastete (Kontrolle der Aktoren, Lokalisation, Planung von Routen und Aktionen) Roboter-CPU, kommt aber nicht in Frage.

1.2 Übersicht über das System

Da die Übertragung der Bilddaten von den Kameras an die CPU des Roboters über ein FireWire Bus erfolgt, waren dementsprechend für die Realisierung des Bildverarbeitungssystems folgende Systemkomponenten vorgesehen: eine FireWire Platine (siehe Abbildung 1.2), die die unteren Protokollebenen (physical layer und link layer) von IEEE 1394a-1995 Standard (siehe [11]) implementiert, ein FPGA-Prototypenboard von Altera (siehe Abbildung 1.3).

Die FireWire Platine wird an den FireWire Bus angeschlossen und ist über einen 32 Bit Daten-, 8 Bit Adressbus und einige Kontrollleitungen mit dem FPGA-Board verbunden. Insgesamt ergibt sich damit die in Abbildung 1.4 dargestellte Verschaltung der Systemkomponenten. Auf dem FPGA-Prototypenboard sollte letztendlich die Hardware (Interface zur FireWire Platine) und Software realisiert werden, die die Kameras ansteuert und die Bildvorverarbeitung übernimmt.

Für das FPGA-Board stand eine leistungsstarke Entwicklungsumgebung zur Verfügung. Damit war es möglich eine komplette CPU (NIOS) mit Speicher und

Abbildung 1.1: Intelligenter Service Roboter

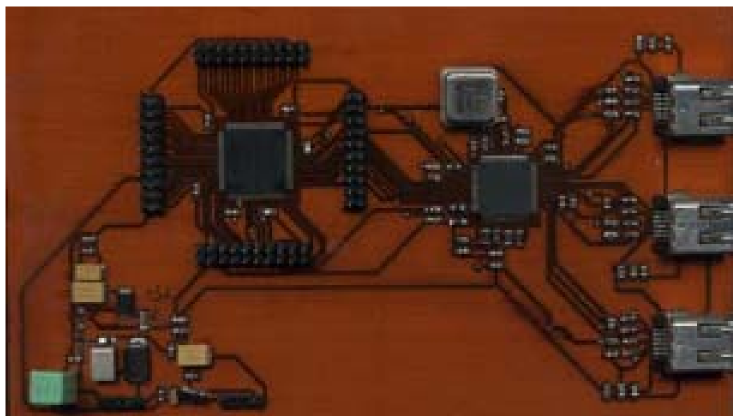

Abbildung 1.2: FireWire Platine

Abbildung 1.3: FPGA-Prototypenboard von Altera

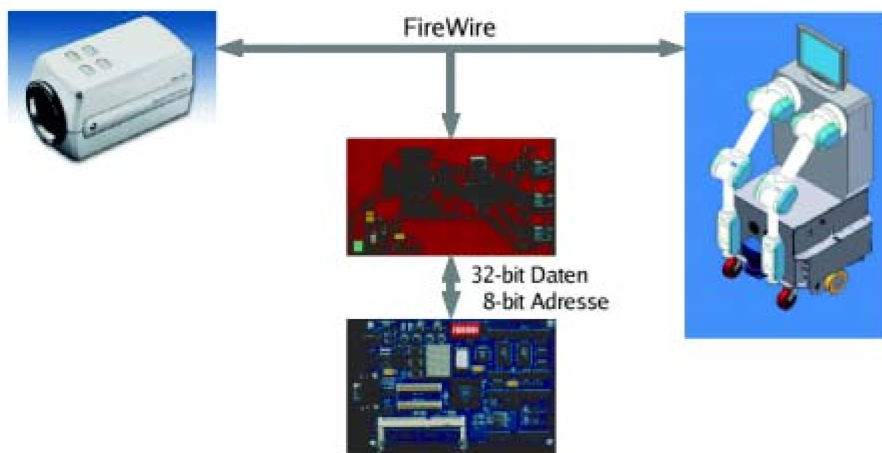

Abbildung 1.4: Verschaltung der Systemkomponenten

vielen anderen Komponenten praktisch per Knopfdruck in Form von VHDL-Dateien zu generieren (siehe [8, 9]). Die Software für die NIOS-CPU konnte in C geschrieben werden. Passende Header-Dateien, die gleichzeitig mit der CPU von der Altera-Software erzeugt werden, vereinfachen die Programmierung der I/O-Ports. Mit einem C-Cross-Compiler kann der C-Code in die Maschinensprache der NIOS-CPU übersetzt werden. Der, auf diese Weise erzeugte Maschinencode wird dann auf das FPGA-Board übertragen und automatisch gestartet.

Die Aufgabe bestand nun darin, eine Schnittstelle zwischen der NIOS-CPU und der FireWire Platine in VHDL zu entwerfen, so dass die NIOS-CPU mit dem link layer chip (LLC) auf der FireWire Platine kommunizieren konnte. Im Laufe des Projektes hat sich eine bestimmte Architektur der Schnittstelle herauskristallisiert, die es überhaupt erst möglich gemacht hat, die großen Mengen an Daten von der NIOS-CPU zu verarbeiten. Der Entwurf und Implementierung einer zentralen Komponente dieser Schnittstellen-Architektur ist Gegenstand dieser Baccalaureatsarbeit und wird im Nachfolgendem im Detail beschrieben.

Kapitel 2

Architektur der Schnittstelle

Bei Betrachtung der in Abbildung 2.1 dargestellten Architektur lässt sich leicht erkennen, dass Daten die vom LLC empfangen werden auf zwei Wegen auf den Avalon Bus (siehe [10]) gelangen: zum einen über die General-Purpose I/O Komponente und zum anderen über den DMA Controller. Dabei sorgt der ioFSM durch Überprüfung der Paket-Header dafür, dass isochrone Pakete (Bilddaten) an den DMA Controller und von da aus über den Avalon Bus direkt in den SDRAM geleitet werden und asynchrone Pakete über die General-Purpose I/Os und den Avalon Bus an die NIOS-CPU weitergereicht werden. In diesem Zusammenhang sei an den Unterschied zwischen isochroner und asynchroner Datenübertragung erinnert:

Asynchrone Datenübertragung: Bei der asynchronen Datenübertragung können Daten zu einem beliebigen Zeitpunkt beginnen und an einem beliebigen Zeitpunkt wieder enden. Dazu müssen im Protokoll ein Datenbeginnzeichen und ein Datenendezeichen definiert werden. Die asynchrone Übertragung wird bei der seriellen Datenübertragung genutzt.

Isochrone Datenübertragung: Bei der Isochronen Datenübertragung liegt zwischen zwei Datenpaketen eine festgelegte Pause. Im Gegensatz zur asynchronen Übertragung, bei welcher zwischen zwei Paketen eine variable Pausenlänge vorliegt, ist diese bei der isochronen Übertragung konstant. Die Isochrone Übertragung wird benötigt, wenn eine konstante Bandbreite in bestimmten Intervallen benötigt wird (z. B. bei der Übertragung von Video und Audio).

Die Entwurfsentscheidung die isochronen Pakete über den DMA Controller in den SDRAM zu übertragen und nicht über die NIOS-CPU (dies hätte den Entwurf viel einfacher gemacht) hat sich als notwendig erwiesen und resultierte aus der Tatsache, dass der FPGA-Chip mit maximal 33MHz getaktet werden darf. Mit dieser Taktrate und einer Bildauflösung von 640×480 Pixel bei 30fps und 16bit pro Pixel (YUV 4:2:2) wäre nämlich die NIOS-CPU nicht in der Lage die Bilddaten in Echtzeit zu verarbeiten. So ist dem Technical Manual der Kamera (siehe [7])

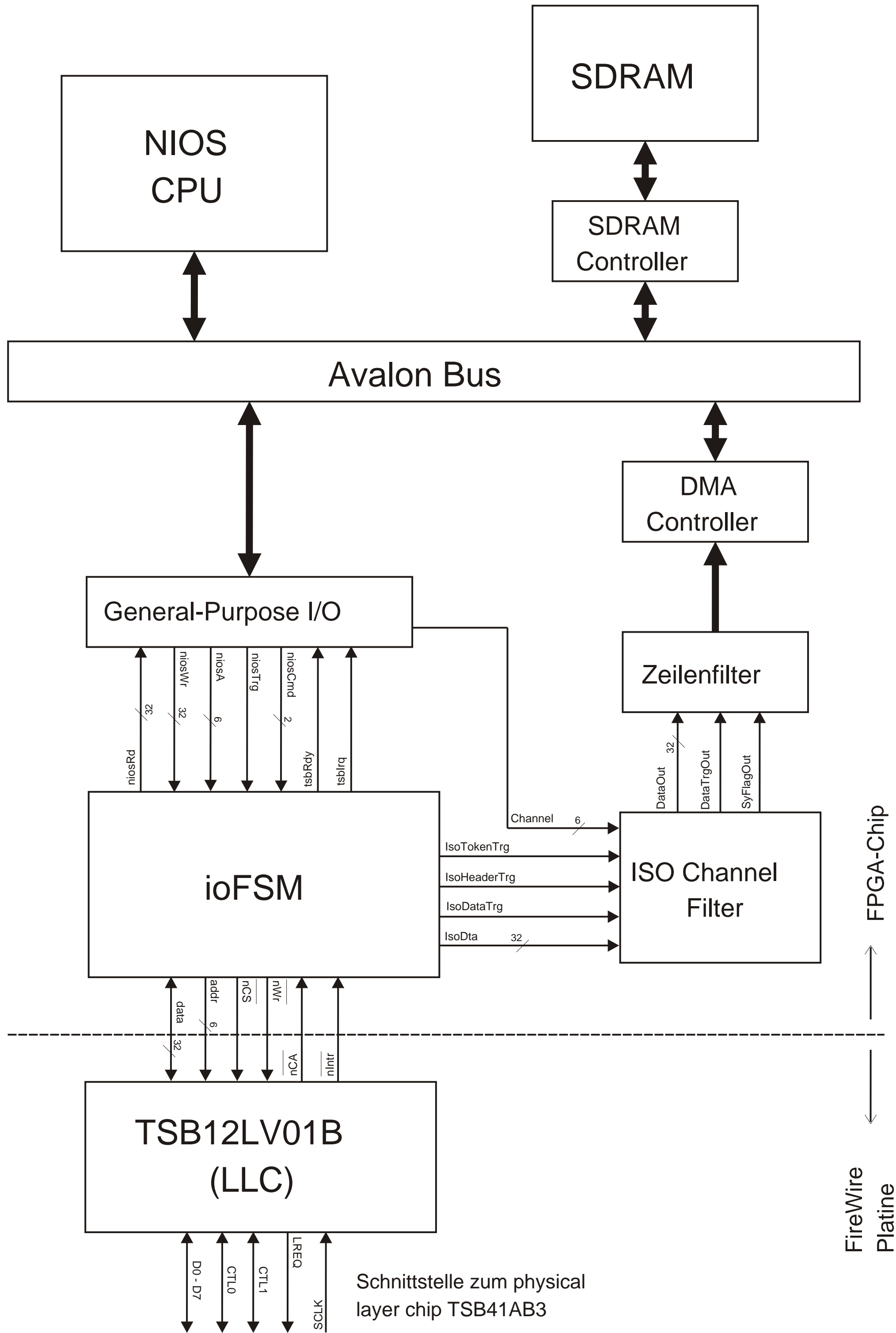

Abbildung 2.1: Blockschaltbild der Schnittstellenarchitektur

zu entnehmen, dass bei diesen Kameraeinstellungen die Übertragung der Bilddaten mit 400Mbps erfolgt. Bei 33MHz Taktrate kommt der ioFSM im Burst-Modus auf eine Datenrate von 33MHz · 32bit = 1056Mbps. Das bedeutet, die NIOS-CPU müsste praktisch alle $\lfloor \frac{1056\text{Mbps}}{400\text{Mbps}} \rfloor = \lfloor 2{,}64 \rfloor = 2$ Taktzyklen die Daten vom ioFSM entgegennehmen und sie in den SDRAM übertragen. Dies würde natürlich die CPU bei weitem überfordern.

Der Zeilenfilter hat bereits die Aufgabe den Bildaufbau aus den einzelnen Paketen zu unterstützen, und soll in dieser Arbeit nicht weiter betrachtet werden, da er nicht Gegenstand der eigenen Arbeit war.

2.1 Signale des ioFSM

Bei dem ioFSM handelt es sich um eine zentrale Komponente der Schnittstellenarchitektur, die das Bindeglied zwischen der NIOS-CPU und dem LLC (Link Layer Controller) darstellt. Der ioFSM sorgt vor allem für die Trennung der asynchronen von den isochronen Paketen und stellt sicher, dass das Timing bei der Kommunikation mit dem LLC eingehalten wird (z. B. müssen min. 3 Buszyklen vergehen, bevor $\overline{\mathrm{nCS}}$ erneut gesetzt werden darf).

Den folgenden Signal-Beschreibungen liegt das Blockschaltbild der Schnittstellenarchitektur in Abbildung 2.1 zu Grunde.

2.1.1 Schnittstelle zu der NIOS-CPU

niosRd
Auf diesem 32 Bit Datenbus werden im Lesezyklus Daten der NIOS-CPU bereitgestellt.

niosWr
Über diesen 32 Bit Datenbus übergibt die NIOS-CPU Daten an den ioFSM im Schreibzyklus.

niosA
6 Bit Adressbus zum Adressieren der Speicherbereiche des LLC.

niosTrg
Durch einen Signalwechsel auf der $\mathrm{niosTrg}$-Leitung signalisiert die NIOS-CPU dem ioFSM, dass sie eine Aktion ausführen möchte - die CPU triggert den ioFSM an. Um was für eine Aktion es sich handeln soll, hängt von der Belegung der beiden $\mathrm{niosCmd}$-Leitungen ab.

niosCmd
Diese 2 Leitungen geben an ob die NIOS-CPU lesen (10), schreiben (11) oder keine Aktion ausführen (01) möchte.

tsbRdy
Ist die ioFSM Komponente bereit Befehle von der NIOS-CPU entgegenzunehmen, so macht sie das, durch Setzen des tsbRdy-Signals auf hohen Pegel, der CPU bekannt.

tsbIrq
Sollte auf der Seite des LLC sich ein Interrupt ereignet haben, so wird dies (mit Ausnahme des Receiver has data Interrupts) der NIOS-CPU sofort gemeldet, indem der tsbIrq-Leitung ein Signal mit hohen Pegel zugeführt wird.

2.1.2 Schnittstelle zum Link Layer Controller (LLC)

data
Ein bidirektionaler 32 Bit Datenbus, der für Datenaustausch zwischen dem ioFSM und dem LLC verwendet wird. Da der gleiche Datenbus sowohl für das Lesen als auch Schreiben benutzt wird, muss der ioFSM Tri-State-Ausgänge verwenden. Das heißt, beim Lesen sperrt der ioFSM seine Ausgänge (hochohmiger Ausgangszustand), um den Bus nicht zu belasten und damit dem LLC ein fehlerfreies Schreiben auf den Bus zu ermöglichen.

addr
Obwohl der LLC genaugenommen über 8 Adressleitungen adressiert wird, benötigt der ioFSM zum adressieren des LLC nur 6 Adressleitungen. Die ersten beiden Adressleitungen führen konstant ein (0-Signal). Damit ist gewährleistet, dass der LLC quadletweise (1 Quadlet = 32 Bit) adressiert wird.

$\overline{\mathbf{nCS}}$
Ein 0-Signal auf der $\overline{\mathrm{nCS}}$-Leitung initiiert einen Zugriff auf den LLC. Abhängig davon welchen Signalwert die $\overline{\mathrm{nWr}}$-Leitung führt, wird damit ein Lesezugriff oder ein Schreibzugriff eingeleitet.

$\overline{\mathbf{nCA}}$
Das Cycle acknowledge-Signal liegt auf einem niedrigen Pegel, wenn der Zugriff auf den LLC abgeschlossen ist und signalisiert damit dem ioFSM, dass z. B. Daten auf dem Datenbus (data) zum Lesen bereitgestellt sind oder dass der LLC mit dem Schreiben von Daten fertig ist.

$\overline{\mathbf{nWr}}$
Liegen das $\overline{\mathrm{nWr}}$-Signal zusammen mit dem $\overline{\mathrm{nCS}}$-Signal auf einem niedrigem Pegel, so erkennt damit der LLC, dass ein Schreibzugriff erfolgen soll. Für den Fall, dass $\overline{\mathrm{nWr}}$ auf hohen Pegel liegt, wird ein Lesezugriff eingeleitet.

$\overline{\mathbf{nIntr}}$
Ein niedriger Pegel signalisiert dem ioFSM das Auftreten eines Interrupts.

2.1.3 Schnittstelle zum ISO Channel Filter

IsoDta
Ein unidirektionaler 32 Bit Datenbus für die Übertragung von isochronen Daten an den ISO Channel Filter.

IsoTokenTrg
Ein hoher Pegel bedeutet, dass die Daten, die am Datenbus (IsoDta) anliegen dem ersten Quadlet eines isochronen Pakets (token) entsprechen.

IsoHeaderTrg
Ein hoher Pegel bedeutet, dass die Daten, die am Datenbus (IsoDta) anliegen dem zweiten Quadlet eines isochronen Pakets entsprechen. In diesem ist unter anderem die channel number und ein Synchronisationsfeld enthalten, das für den Fall, dass das Paket von einer Kamera kommt, angibt, ob es sich, um das erste Paket eines Bildes handelt. Ausführliche Beschreibungen aller Paketformate des LLC sind in [5] zu finden.

IsoDataTrg
Triggert den Channel Filter an (hoher Pegel), wenn neue Daten (keine Header-Quadlets) am Bus (IsoDta) anliegen.

2.2 Signale des ISO Channel Filters

Der ISO Channel Filter ist zuständig für die Überprüfung der Kanalnummer (channel number), der die isochronen Pakete angehören. Dies ist notwendig, da bei zwei Kameras auch zwei Channel Filter benötigt werden, die dann jeweils die Bilddaten der einen Kamera nicht weiterleiten und damit die Bilddaten getrennt halten. Für Synchronisationszwecke erkennt der Channel Filter, ob ein Paket das erste eines Bildes ist und signalisiert dies dem Zeilenfilter.

Die Schnittstelle zum ioFSM, die aus den Signalen IsoDta, IsoTokenTrg, IsoHeaderTrg und IsoDataTrg besteht, wurde bereits im Abschnitt 2.1.3 erläutert. Daher wird im Folgenden auf erneute Erläuterung der genannten Signale verzichtet.

2.2.1 Schnittstelle zum Zeilenfilter

DataOut
Ein unidirektionaler 32 Bit Datenbus für die Übertragung von isochronen Daten an den Zeilenfilter.

DataTrgOut
Triggert den Zeilenfilter an (hoher Pegel), wenn neue Daten (keine Header-Quadlets) am Bus (DataOut) anliegen. Header-Quadlets werden an den Zeilenfilter

nicht weitergereicht, da dieser sie nicht benötigt.

SyFlagOut
Ein Synchronisationsflag, das auf 1 gesetzt wird, wenn das erste Quadlet eines Bildes zum Zeilenfilter übertragen wird.

2.2.2 Schnittstelle zu der NIOS-CPU

Channel
Gibt die Kanalnummer (channel number) an, auf der die NIOS-CPU isochrone Daten empfangen möchte. Sie sollte mit der Kanalnummer in einem der beiden Ports im Control-Register des LLC (Adresse 18h) übereinstimmen (siehe Abbildung 3.2 auf Seite 14). Die 6 Leitungen werden direkt von der NIOS-CPU gesetzt. Die Angabe der Kanalnummer bei dem Channel Filter wird vor allem dann benötigt, wenn isochrone Daten von zwei Kameras gleichzeitig empfangen werden sollen. Dies impliziert die Verwendung von zwei Channel Filtern, die mit unterschiedlicher Kanalnummer konfiguriert sind und damit die isochronen Daten der beiden Kameras voneinander trennen.

Kapitel 3

Entwurf und Implementierung des ioFSM

Dieses Kapitel beschreibt im Detail die eigenen Arbeiten, die im Rahmen des Projektes geleistet worden sind.

Für den Entwurf des ioFSM ist es notwendig den Aufbau und die Funktionsweise des Link Layer Controllers zu kennen. Im Folgenden soll der LLC nur insoweit beschrieben werden, wie das für den Entwurf des ioFSM von Bedeutung ist. Die vollständige Beschreibung des TSB12LV01B Link Layer Controllers findet der interessierte Leser in [5].

3.1 Der TSB12LV01B Link Layer Controller

3.1.1 Interner Aufbau

Der interne Aufbau des LLC ist in Abbildung 3.1 dargestellt. Wichtige Komponenten sind der FIFO-Speicher, der in die 3 FIFOs: ATF, ITF und GRF unterteilt ist, die Transmitter und Receiver-Einheit und die Konfigurationsregister (CFR).

Die FIFOs ATF, ITF und GRF sind in ihrer Größe variabel. Insgesamt ist der FIFO-Speicher 2kB = 512 Quadlets groß. Die Größe der beiden Sende-FIFOs ATF und ITF kann direkt in den Konfigurationsregistern vorgegeben werden. Die Größe des Empfangs-FIFOs (GRF) errechnet sich nach der Formel: GRFSize = 512 - (ATFSize + ITFSize). Der Asynchronous Transmit-FIFO (ATF) wird zum Versenden von asynchronen Paketen verwendet und der Isochronous Transmit-FIFO zum Versenden von isochronen Paketen. Da der ioFSM isochrone Daten nur empfangen soll, ist der ITF nicht weiter wichtig.

Der Zugriff auf die FIFOs erfolgt über interne Register (memory mapping) die im nächsten Abschnitt näher erläutert werden.

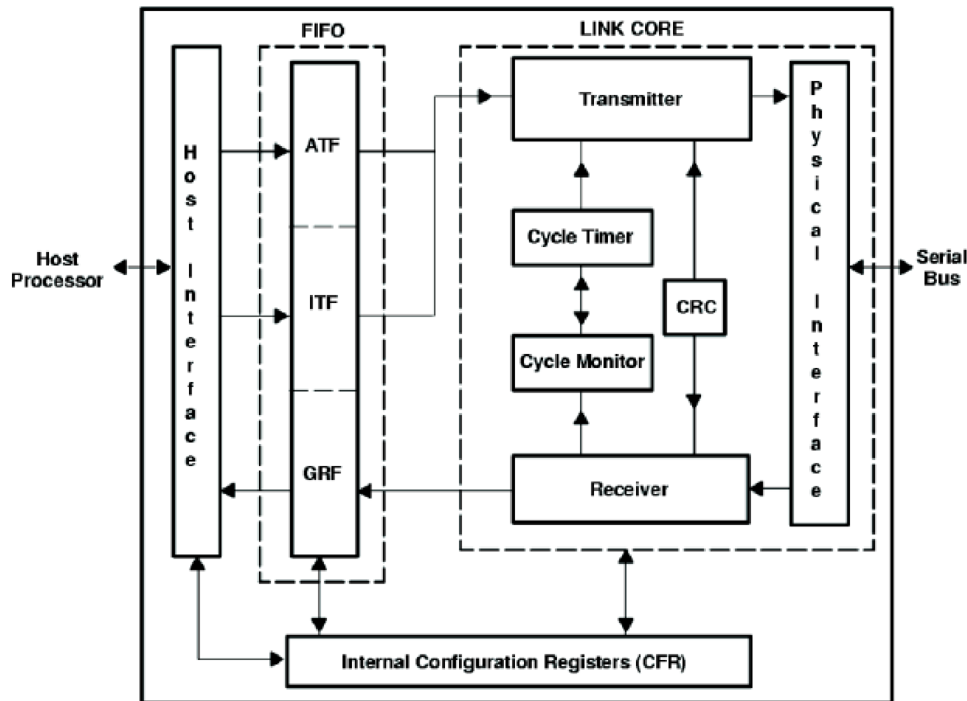

Abbildung 3.1: Blockschaltbild des TSB12LV01B

Der Transmitter ist zuständig für das Versenden von Paketen aus dem ATF oder ITF. Seine Aufgabe besteht u. a. darin, die Pakete für das Versenden entsprechend zu formatieren.

Der Receiver schaut bei jedem ankommenden Paket nach, ob dieser an den Knoten adressiert ist und, wenn das der Fall ist, überprüft er automatisch den CRC. Damit soll sichergestellt werden, dass nur korrekte Pakete im GRF gespeichert werden.

Internal Configurations Registers (CFR): Diese Register dienen der Konfiguration des LLC (siehe Abschnitt 3.1.2)

3.1.2 Interne Register - Speicherlayout des LLC

Bei den internen Registern ist zu unterscheiden zwischen Konfigurationsregistern (siehe Abbildung 3.2) und Registern, die den Zugriff auf die FIFOs (siehe Abbildung 3.3) ermöglichen.

Offset	Bits 0–31	Register
00h	Version (3031h) | Revision (3043h)	Version
04h	Bus Number | Node Number | Root | Reserved | ATAck | Reserved | AckV	Node Address
08h	IdVal | RxSld | BsyCtrl | RAI | RcvCyst | TxAEn | RxAEn | TxIEn | RxIEn | AckCen | RstTx | RstRx | Reserved | CyMas | CySrc | CyTEn | TrgEn | IRP1En | IRP2En | Reserved | FhBad	Control
0Ch	Int | PhInt | PhRRx | PhRst | SIDCom | TxRdy | RxDta | CmdRst | ACKRCV | Reserved | ITBadF | ATBadF | Reserved | SntRj | HdrEr | TCErr | Reserved | CyTm0 | CySec | CySt | CyDne | CyPnd | CyLst | CArbFl | Reserved | ArbGp | FrGp | IArbFl	Interrupt
10h	Int | PhInt | PhRRx | PhRst | SIDCom | TxRdy | RxDta | CmdRst | ACKRCV | Reserved | ITBadF | ATBadF | Reserved | SntRj | HdrEr | TCErr | Reserved | CyTm0 | CySec | CySt | CyDne | CyPnd | CyLst | CArbFl | Reserved | ArbGp | FrGp | IArbFl	Interrupt Mask
14h	Seconds Count (7 Bits) | Cycle Count (Rollover @ 8000, 13 Bits) | Cycle Offset (Rollover @ 3072, 12 Bits)	Cycle Timer
18h	TAG1 | IR Port1 | TAG2 | IR Port2 | Reserved | MonTag	Isoch Port Number
1Ch	CLrATF | ClrITF | ClrGRF | Reserved | Trigger Size | ATFSize | ITFSize	FIFO Control
20h	ENSp | Reserved | regRW | Reserved	Diagnostics
24h	RdPhy | WrPhy | Reserved | PhyRgAd | PhyRgData | Reserved | PhyRxAd | PhyRxData	PHY Chip Access
28h	Reserved	Reserved
2Ch	Reserved	Reserved
30h	Full | Empty | ConErr | AdrClr | Control | RAMTest | AdrCounter | Reserved | ATFSpaceCount	ATF Status (Read/Write)
34h	Full | Empty | Reserved | ITFSpaceCount	ITF Status (Read Only)
38h	Reserved	Reserved
3Ch	Empty | cd | PacCom | GRFTotalCnt | GRFSize | WriteCount	GRF Status (Read Only)
40h	AccFl | AccFM | LPS | SRst | Reserved	Host Control (see Note B)
44h	Reserved | GPO2 | Reserved | GPO1 | Reserved | GPO0	Mux Control (see Note B)

NOTES: A. All gray areas (bits) are reserved bits.
B. This register is new to the TSB12LV01B and does not exist in the TSB12LV01A.

Abbildung 3.2: Die internen Konfigurationsregister

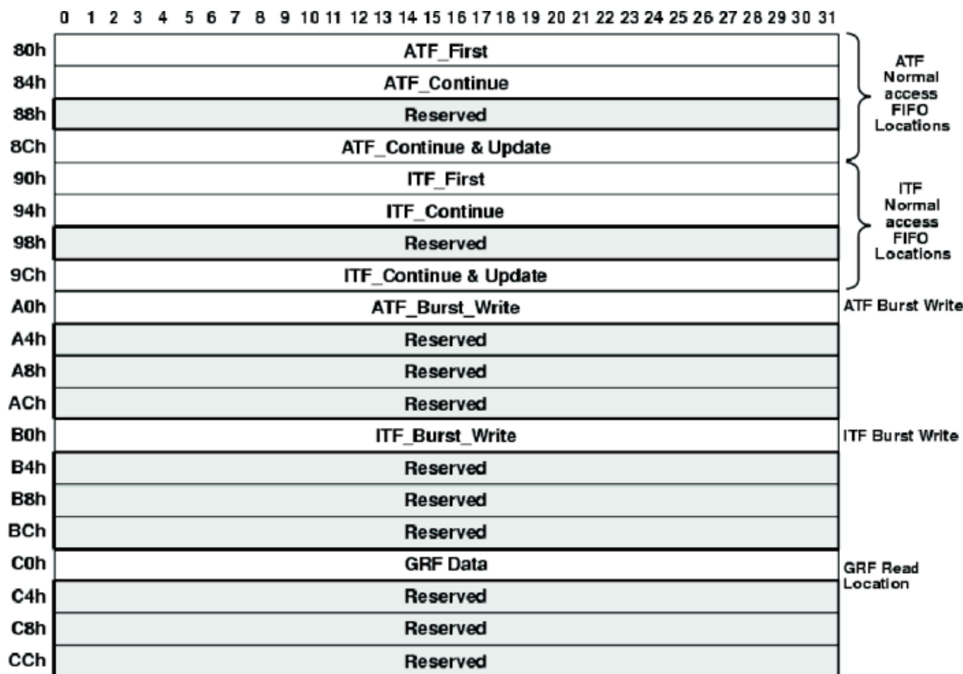

Abbildung 3.3: Die internen FIFO-Register

Die Zugriffsmodi des LLC

Der Zugriff auf die Register erfolgt durch memory-mapping, sie werden also, wie ein normaler Speicher adressiert. Der Speicherbereich der Konfigurationsregister erstreckt sich von der Adresse 00h bis 44h, der FIFO-Zugriffsregister von 80h bis CCh. Je nach Speicherbereich kann der Zugriff auf die Register in verschiedenen Modi erfolgen. Es gibt insgesamt drei Zugriffsmodi: den normal mode, den quick mode und den burst mode. Allen drei liegen unterschiedliche Timing-Diagramme zu Grunde (siehe Abbildung 3.4).

Der normal mode: Im Adressraum 00h bis 2Ch kann der Zugriff auf die Register nur im normal mode erfolgen. Dies hat zur Folge, dass man unter Umständen bis zu 9 Taktzyklen lang warten muss, bis man von dem LLC eine Bestätigung bekommt ($\overline{\mathrm{CA}}$[1] = 0), dass der Zugriff vollendet ist. Das heißt, der LLC ist mit dem Schreiben fertig (Write-Cycle) oder die zu lesenden Daten liegen am Bus an (Read-Cycle).

Der quick mode: Ab Adresse 30h ist ein Zugriff auf den LLC im quick mode möglich. Dieser Modus unterscheidet sich von dem normal mode nur darin,

[1]Entspricht dem $\overline{\mathrm{nCA}}$ in Abbildung 2.1

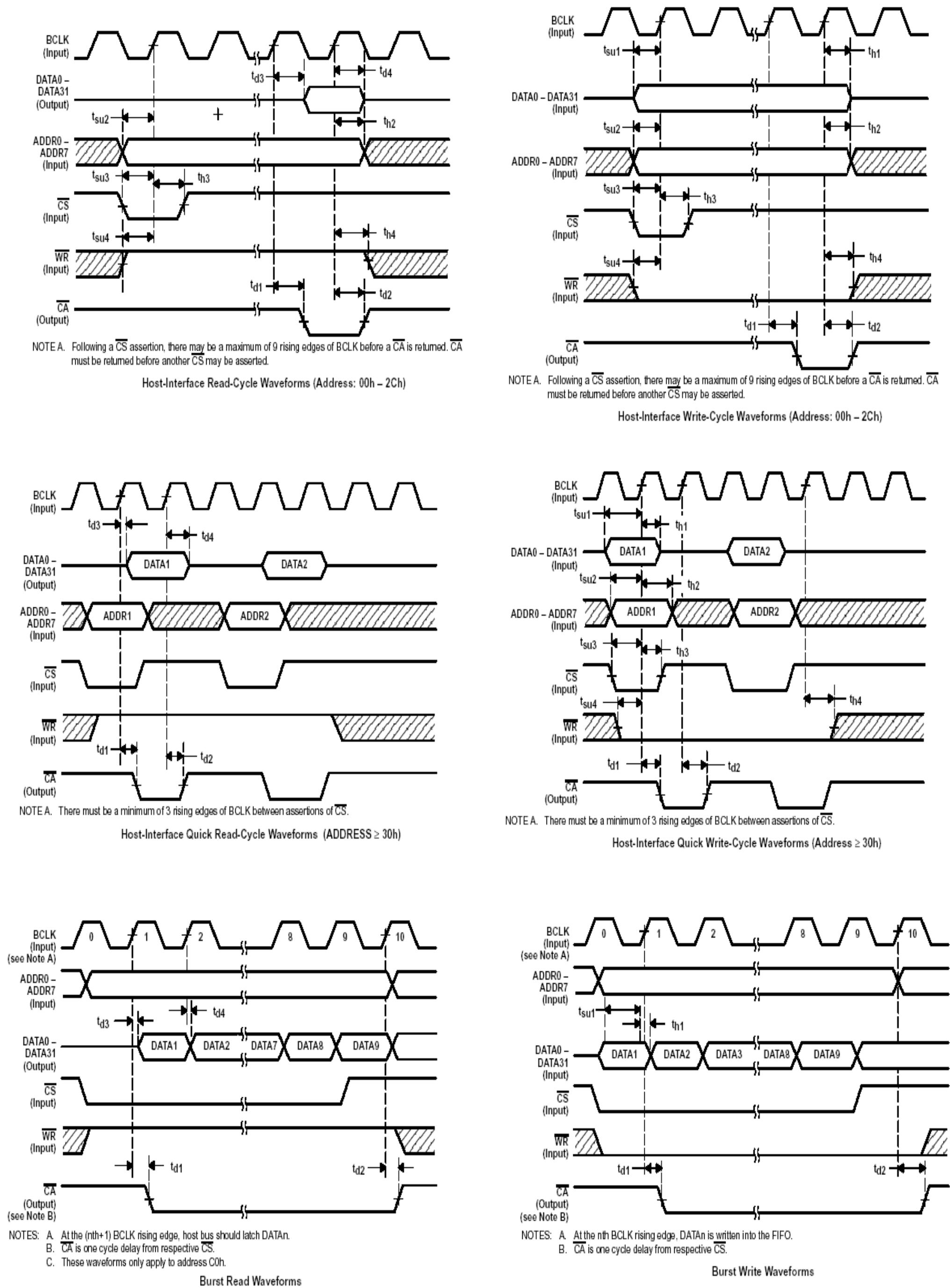

Abbildung 3.4: Timing-Diagramme des TSB12LV01B

dass der Zugriff bereits nach einem Taktzyklus vollendet ist. Allerdings ist zu beachten, dass die $\overline{\mathrm{CS}}$-Leitung[2] erst nach drei Taktzyklen wieder gesetzt werden darf. Das bedeutet also, dass der nächste Zugriff auf den LLC erst drei Taktzyklen nach Beginn des letzten Zugriffs erfolgen darf. Dies wird noch Konsequenzen für den Entwurf des ioFSM haben.

Der burst mode: Noch effizienter als der Zugriff im quick mode ist der Zugriff im burst mode. Wie aus dem Timing-Diagramm in Abbildung 3.4 zu ersehen ist, können im burst mode in jedem Takt neue Daten gelesen bzw. geschrieben werden. Der Zugriff im burst mode wird eingeleitet, indem die $\overline{\mathrm{CS}}$-Leitung gesetzt bleibt und bleibt dann solange bestehen, bis das $\overline{\mathrm{CS}}$-Signal wieder auf hohen Pegel liegt. Zu beachten ist, dass im burst mode sinnvollerweise nur auf den FIFO-Speicher zugegriffen werden kann.

Wichtige Register

Wichtig für den Entwurf des ioFSM sind vor allem die folgenden Register: GRF Status (3Ch), GRF Data (C0h), Interrupt (0Ch) und Interrupt Mask (10h). Es wird sich noch zeigen, dass der ioFSM den Zugriff auf diese Register seitens der NIOS-CPU gesondert behandeln muss. Was damit genau gemeint ist, soll hier erstmal nicht weiter erläutert werden, sondern nur die Bedeutung der einzelnen Register beschrieben werden.

Das GRF Data-Register: Über das GRF Data-Register wird auf den GRF zugegriffen. Dabei soll der ioFSM den GRF im burst mode auslesen können.

Das GRF Status-Register: Im GRF Status-Register sind vor allem die Bits Empty und cd wichtig, die angeben, ob der GRF leer ist (Empty = 1) bzw., ob das nächste Quadlet im GRF ein Token-Quadlet ist (cd = 1), also das erste Quadlet eines Pakets ist.

Entscheidend ist auch das WriteCount-Feld, welches die Anzahl der Quadlets im Paket angibt, wobei das Token-Quadlet nicht mitgezählt wird. Das WriteCount-Feld ist nur dann gültig, wenn cd = 1 gilt, also immer nur zum Anfang eines Pakets.

Die Interrupt- und Interrupt Mask-Register: Im Zusammenhang mit dem Interrupt- und Interrupt Mask-Register ist das RxDta-Bit (Receiver has data) von Bedeutung. Das RxDta-Bit wird immer dann auf 1 gesetzt, wenn ein Paket vom LLC empfangen wurde. Ist dann auch das entsprechende Bit im Interrupt Mask-Register gesetzt, so führt dies zum Auslösen eines Interrupts das heißt, das $\overline{\mathrm{nIntr}}$-Signal liegt dann auf niedrigem Pegel.

[2]Entspricht der $\overline{\mathrm{nCS}}$-Leitung in Abbildung 2.1

3.2 Die ioFSM-Komponente

Der Stand der Technik ist, dass der Entwurf einer digitalen Hardwarekomponente in der Regel auf Register-Transfer Ebene vollzogen wird. Synthesewerkzeuge erzeugen dann eine Netzliste, die die modellierten Schaltnetze bzw. Schaltwerke realisiert. Bei den Steuerungsaufgaben, die der ioFSM übernehmen soll, kommt man mit einem Schaltnetz mit Sicherheit nicht aus. Es gilt also ein Schlatwerk zu entwerfen, welches das geforderte Verhalten realisiert. In solchen Fällen empfiehlt sich, vor der Implementierung in VHDL, das Schaltwerk zuerst durch einen endlichen Automaten zu modellieren.

3.2.1 Der Automatenentwurf

Wichtiges Kriterium für den Entwurf des Automaten ist die Bedingung, dass das Schaltwerk synchron mit dem LLC arbeiten soll. Das heißt, der ioFSM und der LLC bekommen das gleiche Taktsignal zugeführt. Nach Analyse der Timing-Diagramme in Abbildung 3.4 stellt sich heraus, dass der ioFSM, vor allem, wenn Lesen im burst mode unterstützt werden soll, auf der Rückflanke arbeiten sollte.

In Abbildung 3.5 ist das vollständige Automatenmodell des ioFSM dargestellt. Alle Zustände mit dem Präfix 'grf' hängen grundsätzlich mit dem Zugriff auf den GRF zusammen. Die restlichen Zustände dienen vor allem der Verarbeitung der Befehle von der NIOS-CPU. An den Kanten zwischen den Zuständen stehen entscheidende Bedingungen für den Zustandsübergang und/oder Aktionen die noch vor dem Zustandsübergang oder direkt beim Zustandsübergang (Signalzuweisungen) ausgeführt werden. Die Aktionen sind meistens in VHDL-Syntax notiert, da sie direkt aus dem VHDL-Quellcode des ioFSM (siehe Anhang) übernommen wurden.

Der Automat an sich wurde so konzipiert, dass das Auslesen, der Daten aus dem GRF nicht interruptgesteuert, sonder durch "polling" erfolgt. Außerdem wurde mit Bedacht darauf verzichtet, die Abarbeitung der Request's von der NIOS-CPU nebenläufig zu der Übertragung der isochronen Daten an den Channel Filter zu gestalten. Dabei hat die CPU, ausgehend vom idleSt-Zustand, Vorrang gegenüber dem Lesen von Daten aus dem GRF, muss jedoch warten, wenn gerade ein Paket aus dem GRF gelesen und an den Channel Filter übertragen (isochrone Daten) oder in den internen FIFO (siehe Abschnitt: Der interne FIFO und Register-Shadowing) geschrieben wird (asynchrone Daten). Nach jedem vollständig gelesenem Paket aus dem GRF kehrt dann der Automat in den idleSt-Zustand zurück, so dass wieder Request's von der CPU abgearbeitet werden können.

Der interne FIFO und Register-Shadowing

Da der ioFSM neben der Kommunikation mit der NIOS-CPU selbstständig isochrone Daten an den ISO Channel Filter übertragen soll, muss dafür gesorgt werden, dass, wenn gerade asynchrone Daten im GRF vorliegen, diese von dem ioFSM

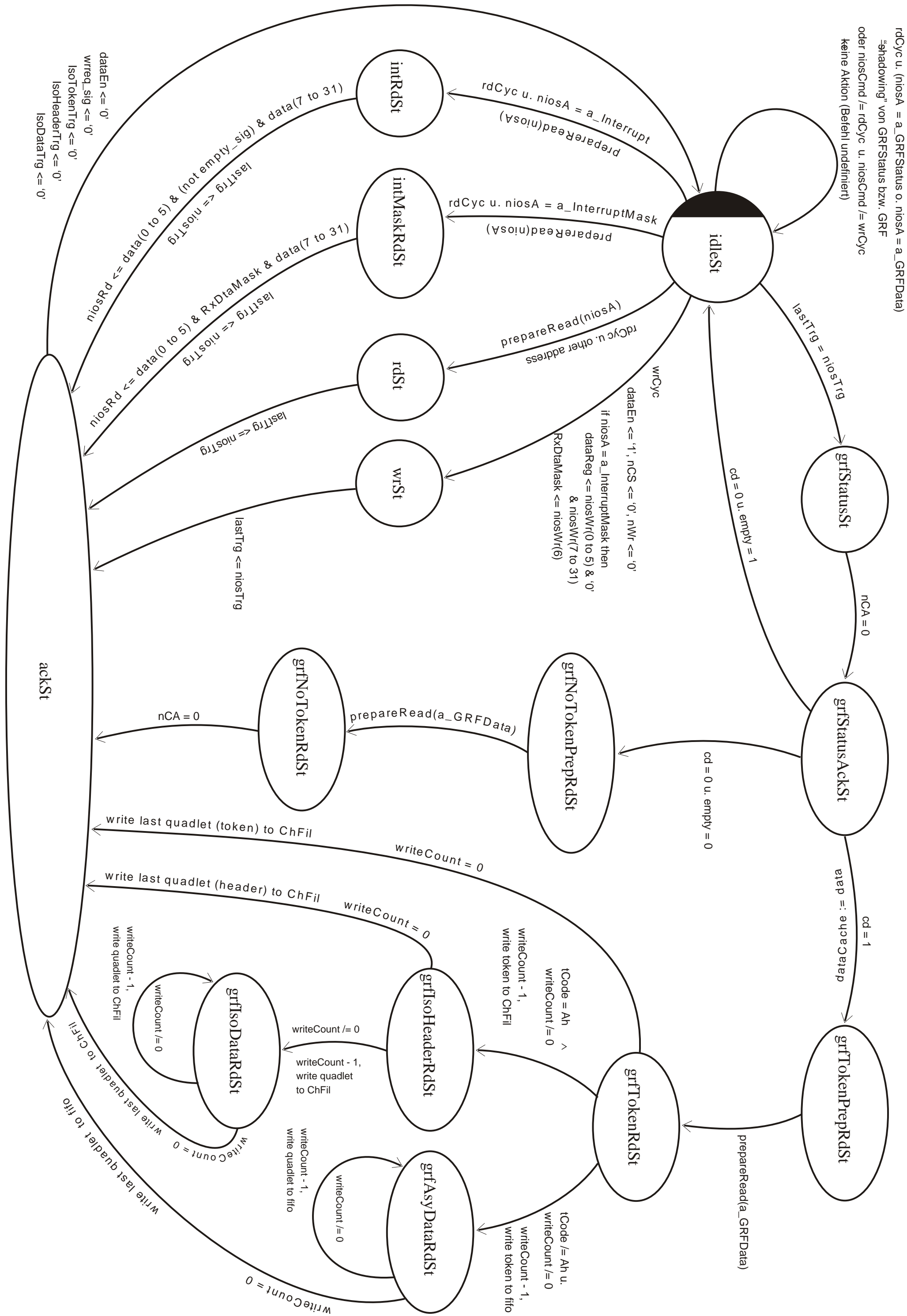

Abbildung 3.5: Automatenmodell der ioFSM-Komponente

zwischengespeichert werden können. Die Lösung hierfür ist ein interner FIFO in dem letztendlich nur asynchrone Pakete verwaltet werden. Allerdings impliziert die Verwaltung eines eigenen FIFOs Inkonsistenzen mit bestimmten Konfigurationsregistern des LLCs. Betroffen sind die Register: GRF Status, Interrupt und Interrupt Mask. Der ioFSM muss also den Zugriff seitens der NIOS-CPU auf diese Register abfangen und der CPU Registerinhalte liefern, die den Zustand des eigenen internen FIFOs wiedergeben und nicht den des GRFs (Shadowing). Das jeweilige Register wird also versteckt und durch ein eigenes Register ersetzt. Im Falle des GRF Status-Registers betrifft das alle Felder und Bits (Empty, cd, WriteCount usw.), im Falle der Interrupt- und Interrupt Mask-Register nur das RxDta-Bit. Es soll nämlich nur dann ein Receiver has data-Interrupt ausgelöst werden, wenn es sich, um asynchrone Daten handelt und diese bereits in dem FIFO des ioFSM gespeichert sind. Wie das alles im Einzelnen beim Entwurf des ioFSM realisiert wurde, wird noch in den folgenden Abschnitten genauer beschrieben.

Da ein FIFO eine Standard-Komponente ist, war es möglich sich diesen mit Hilfe der ALTERA-Software in VHDL zu generieren. Die Statusinformationen, die der FIFO nach Außen bereitstellt, reichen für das Shadowing der Register aus. Außerdem war mit der automatisch Generierung auch sichergestellt, dass der FIFO effizient implementiert ist.

Verarbeitung der Befehle von der NIOS-CPU

Beginnend im Zustand idleSt — das ist der Zustand der nach einem asynchronen Reset eingenommen wird — wird zunächst überprüft, ob die NIOS-CPU eine Aktion ausführen möchte. Dies ist der Fall, wenn die Bedingung lastTrg /= niosTrg erfüllt ist, d. h., der Wert des niosTrg-Signals sich gegenüber dem zuletzt gespeicherten Wert (lastTrg) verändert hat. Sollte nun die NIOS-CPU durch einen Signalwechsel der niosTrg-Leitung den ioFSM angetriggert haben, so wird überprüft, ob es sich um einen Lese- oder Schreibzugriff handeln soll. Sollte die CPU bestimmte Speicherstellen auslesen wollen (Befehl rdCyc), so müssen dann wieder, abhängig von der Speicheradresse, mehrere Fälle unterschieden werden:

niosA = a_Interrupt: Mit dieser Adresse würde die NIOS-CPU das Interrupt-Register auslesen wollen. Durch die Prozedur prepareRead(niosA) wird ein Lesezugriff eingeleitet und es folgt ein Übergang in den Zustand intRdSt. Ist der Lesezugriff auf den LLC vollendet, dann geht der Automat in den Zustand ackSt über. Dabei wird der Variable lastTrg der Signal-Wert von niosTrg zugewiesen und auf dem Datenbus (niosRd) werden die gelesenen Daten bereitgestellt. Da es sich hierbei um den Inhalt des Interrupt-Registers handelt, wird das RxDta-Bit ausgeblendet und durch den negierten Wert des internen Signals empty_sig ersetzt (Shadowing des RxDta-Bits). Das Signal empty_sig liefert den Wert 1, wenn der interne FIFO leer ist. Dementsprechend darf das RxDta-Bit nur dann den Wert 1 haben, wenn der FIFO nicht leer ist, also empty_sig = 0 bzw. not empty_sig = 1 gilt.

niosA = a_InterruptMask: Hinter der Konstante a_InterruptMask verbirgt sich die Adresse des Interrupt Mask-Registers. Der Lesezugriff läuft genauso ab, wie beim Übergang in den intRdSt-Zustand. Beim Übergang in den ackSt-Zustand und Bereitstellen der gelesenen Daten der NIOS-CPU am Datenbus (niosRd), muss wieder das RxDta-Bit ausgeblendet werden. Dabei wird für das RxDta-Bit der Wert der RxDtaMask-Variable eingesetzt, die auch, für den Fall, dass die NIOS-CPU auf das Interrupt Mask-Register schreiben will, jedes mal entsprechend gesetzt werden muss.

In dem LLC bleibt dabei das RxDta-Bit immer auf 0 gesetzt. Damit ist sichergestellt, dass der LLC beim Eintreffen von Daten, keinen Interrupt auslöst.

niosA = a_GRFStatus: Mit der Adresse a_GRFStatus möchte die NIOS-CPU das GRF Status-Register auslesen. Das GRF Status-Register wird jedoch vollständig von dem ioFSM ersetzt (Shadowing), d. h., die Daten, die auf dem Datenbus landen, werden zusammengesetzt aus empty_sig, q_sig(1 downto 0), "00", usedw_sig, GRFSIZE und q_sig(18 downto 10). empty_sig gibt an ob der interne FIFO leer ist. q_sig(1 downto 0) sind die ersten beiden Bits eines Eintrags in dem internen FIFO, die den Bits cd und PacCom im GRF Status-Register entsprechen. Dazu muss man wissen, dass der interne FIFO mit der Wortbreite von 34 Bit entworfen wurde und jedes mal, wenn ein Quadlet aus dem GRF in den FIFO geschrieben wird, auch festgehalten wird (durch Setzen der ersten beiden Bits in dem FIFO) ob, es das Token-Quadlet ist (cd = 1) und wenn ja, welchen Wert das PacCom-Bit hat (das PacCom-Bit gibt an, ob es sich bei dem zu lesenden Paket, um den letzten Block eines FireWire-Pakets handelt). Die beiden Nullen und das interne usedw_sig-Signal entsprechen dem GRFTotalCnt-Feld, welches den Füllstand des FIFOs angibt. Die Konstante gibt die Kapazität des FIFOs und q_sig(18 downto 10) den Wert des WriteCount-Feldes an. Dieser kann dementsprechend nur gültig sein, wenn cd = 1 gilt.

niosA = a_GRFData: In diesem Fall fängt der ioFSM den Lesezugriff auf den GRF ab und liefert der CPU Daten aus dem internen FIFO.

sonst: Für sonstige Adressen erfolgt der Lesezugriff auf den LLC durch Aufruf der prepareRead(niosA)-Prozedur; es wird gewartet bis die Daten verfügbar sind (nCA = 0) und beim Übergang in den ackSt-Zustand werden die gelesenen Daten der NIOS-CPU am Bus (niosRd) bereitgestellt.

Ein wichtiger Unterschied zwischen den Lese-Request's der NIOS-CPU, die keinen Lesezugriff auf den LLC benötigen (niosA = a_GRFStatus oder niosA = a_GRFData) und solchen, die auf den LLC lesend zugreifen, sollte nicht unerwähnt bleiben. Muss nämlich auf den LLC nicht zugegriffen werden, so kann innerhalb von 2 Taktzyklen (1. FIFO requesten, 2. Daten an die CPU übertragen) ein Request

von der NIOS-CPU beantwortet werden. Muss jedoch auf den LLC zugegriffen werden, so kann im günstigsten Fall (Zugriff im quick mode) erst nach drei Taktzyklen ein Request beantwortet werden. Bei dem Zugriff auf den LLC muss man nämlich (siehe auch Abschnitt 3.1.2) selbst im quick mode sicherstellen, dass die $\overline{\text{CS}}$-Leitung frühstens nach drei Taktzyklen wieder gesetzt wird. Dies ist auch der Grund warum man nicht direkt von den Zuständen intRdSt, intMaskRdSt und rdSt in den idleSt-Zustand springen darf. Das heißt, der Weg über den ackSt-Zustand ist notwendig und das nicht nur im Zusammenhang mit den oben erwähnten drei Zuständen (siehe Abbildung 3.5).

Was jetzt noch übrig bleibt, ist die Verarbeitung der Schreib-Requests (Befehl wrCyc). Hierbei ist nur in einem Fall eine spezielle Abarbeitung nötig:

niosA = a_InterruptMask: Mit dieser Adresse möchte die NIOS-CPU in das Interrupt Mask-Register schreiben. Da, wie bereits beschrieben, das RxDta-Bit einer speziellen Behandlung bedarf, wird dieses jedes mal auf 0 gesetzt. Der Wert, auf den die NIOS-CPU das RxDta-Bit setzen wollte, wird dann der RxDtaMask-Variable zugewiesen. Auf diese Variable greift der Automat auch zurück, wenn die NIOS-CPU das Interrupt Mask-Register auslesen möchte. Darüberhinaus spielt der Wert der RxDtaMask-Variable eine Rolle beim Auslösen von Interrupts durch den ioFSM, wenn der interne FIFO nicht leer ist.

sonst: Bei sonstigen Adressen sollte die NIOS-CPU nur darauf achten, ob an der jeweiligen Adresse Schreibzugriffe möglich sind. Zum Beispiel auf das GRF Data-Register sind Schreibzugriffe nicht möglich.

Generell sollte noch erwähnt werden, dass bei Schreibzugriffen auf den LLC der Tri-State-Treiber eingeschaltet wird. Dies geschieht durch die Anweisung

```
dataEn <= '1'
```

und ist notwendig, da der Datenbus zum LLC bidirektional ist.

Zugriff auf den GRF

Bevor der Automat lesend auf den GRF zugreifen kann muss er zunächst das GRF Status-Register auslesen, um festzustellen, ob der GRF nicht leer. Ausgehend vom idleSt-Zustand kann das GRF Status-Register nur dann ausgelesen werden, wenn die NIOS-CPU den ioFSM nicht “angetriggert” hat, also nur dann wenn niosTrg = lastTrg gilt. Mit prepareRead(a_GRFStatus) erfolgt dann der Lesezugriff auf das GRF Status-Register und der Automat geht in den Zustand grfStatusSt über. Ist der Lesezugriff beendet (nCA = 0), so nimmt der Automat die Daten entgegen (dataCache := data) und geht in den Zustand grfStatusAckSt über. In diesem Zustand wird der Inhalt des GRF Status-Register ausgewertet:

empty = 1: Ist das Empty-Bit auf 1 gesetzt (GRF ist leer) springt der Automat wieder in den idleSt-Zustand zurück.

cd = 0 und empty = 0: Der Fall cd = 0 darf normalerweise nicht eintreten, wurde aber sicherheitshalber im Entwurf berücksichtigt. cd = 0 und empty = 0 würde bedeuten, dass der FIFO ein fehlerhaftes, da ohne den Token, Paket enthält. In solchem Fall würde der Automat den GRF so lange auslesen (nicht im burst mode) bis entweder der Fall cd = 1 oder empty = 1 eintritt.

cd = 1: Gilt cd = 1, d. h., ist das nächste zu lesende Quadlet der Token, dann kann der GRF nicht leer sein und der Automat geht in den Zustand grfTokenPrepRdSt über, wobei vorher noch der writeCount aus dem GRF Status in der Variable writeCount gespeichert wird. Im Zustand grfTokenPrepRdSt wird mit prepareRead(a_GRFData) der Lesezugriff auf den GRF eingeleitet und es folgt der Übergang in den Zustand grfTokenRdSt.

Im grfTokenRdSt-Zustand wird zunächst überprüft, ob der writeCount nicht gleich 0 ist. Gilt writeCount = 0, dann bedeutet das, dass das Paket nur aus dem Token besteht. Handelt es sich um den Token eines isochronen Pakets (tCode = Ah), dann wird der Token verworfen. Handelt es sich um den Token eines asynchronen Pakets (tCode /= Ah), dann wird der Token in dem internen FIFO gespeichert. In beiden Fällen geht der Automat anschließend in den ackSt-Zustand über.

Gilt jedoch writeCount /= 0, dann wird der writeCount um 1 dekrementiert, der Token an den Channel Filter übertragen (tCode = Ah) bzw. im FIFO gespeichert (tCode /= Ah) und der Automat geht in den Zustand grfIsoHeaderRdSt bzw. grfAsyDataRdSt über.

Im grfIsoHeaderRdSt-Zustand wird bereits im burst mode das zweite Quadlet des isochronen Pakets gelesen, an den Channel Filter übertragen und abhängig von dem Wert der writeCount-Variable springt der Automat in den ackSt-Zustand (writeCount = 0) oder den grfIsoDataRdSt-Zustand (writeCount /= 0). Im zweiten Fall wird der writeCount vorher noch um 1 dekrementiert.

Im grfIsoDataRdSt-Zustand liest der ioFSM im burst mode die isochronen Daten aus dem GRF und überträgt sie an den Channel Filter so lange, bis writeCount = 0 gilt. Bei writeCount = 0 überträgt der Automat das letzte Quadlet des Pakets an den Channel Filter und geht in den ackSt-Zustand über. Bei der Übertragung von isochronen nicht-Header-Quadlets ist IsoDataTrg auf 1 gesetzt.

Beim Lesen von isochronen Paketen ist der Zwischenzustand grfIsoHeaderRdSt notwendig, damit der ioFSM dem Channel Filter signalisieren kann,

dass das zweite Quadlet eines Pakets übertragen wird. Die isochronen Pakete haben nämlich immer einen aus 2 Quadlets bestehenden Header, wobei in dem zweiten Quadlet, die für den Channel Filter wichtige channel number enthalten ist.

Im grfAsyDataRdSt werden ebenfalls bereits im burst mode asynchrone Daten aus dem GRF gelesen und in den FIFO übertragen, bis writeCount = 0 gilt. Bei writeCount = 0 schreibt der Automat das letzte Quadlet eines asynchronen Pakets in den FIFO und geht in den ackSt-Zustand über.

3.2.2 Interrupt-Handling des ioFSM

Zusammen mit dem internen empty_sig-Signal (empty_sig = 1, wenn interner FIFO leer) macht der ioFSM das Auslösen eines Interrupts abhängig von dem Wert der RxDtaMask-Variable. Nur wenn RxDtaMask = 1 und empty_sig = 0 wird ein Interrupt ausgelöst. Es sei denn, der LLC hat einen Interrupt ausgelöst der ungleich dem Receiver has data Interrupt (RxDta) war und im Interrupt Mask-Register entsprechend gesetzt war.

Der VHDL-Code dazu sieht wie folgt aus:

```
intr: tsbIrq <= '1' when (nIntr = '0') or
                         ((empty_sig = '0') and
                          (RxDtaMask = '1')) else
                 '0';
```

3.3 Der ISO Channel Filter

Der Entwurf des ISO Channel Filters ist sehr einfach. Er besteht aus einem Schaltnetz, aus zwei Flags und nur einem Register, wenn man von den Registern, die für die Ausgangsports benötigt werden, absieht (siehe VHDL-Quellcode im Anhang). Der Channel Filter bekommt das gleiche Taktsignal, wie der ioFSM zugeführt und arbeitet auf der Vorderflanke.

In dem Register (Variable token) wird immer das aktuelle Token-Quadlet gespeichert. Da (zur Zeit) die einzige Information, die der Zeilenfilter aus dem Token jedes Pakets benötigt, nur der Inhalt des Synchronisationsfeldes ist, könnte man sich dieses Register sogar sparen. Der Channel Filter wertet nämlich den Inhalt des Synchronisationsfeldes in jedem Token aus und merkt sich immer (Variable synchFlag wird auf 1 gesetzt), wenn es sich bei dem aktuellen Paket um den Anfang eines Bildes handelt. Mit der Übertragung des ersten Quadlets dieses Pakets an den Zeilenfilter, signalisiert dann der Channel Filter über das SyFlagOut-Signal (siehe Abbildung 2.1 und Abschnitt 2.2) dem Zeilenfilter, dass es sich um das erste Quadlet eines Bildes handelt. Insofern wurde das Register nur für Debugging-Zwecke eingebaut und für den Fall, dass in zukünftigen Entwicklungen es doch noch nötig

sein könnte, das Token-Quadlet jedes Pakets an den Zeilenfilter zu übertragen. Dieses kann nämlich nur dann übertragen werden, wenn die channel number, die im zweiten Quadlet in jedem isochronen Paket enthalten ist, mit dem Channel-Signal (siehe Abschnitt 2.2) übereinstimmt. Das Zwischenspeichern des Tokens in einem Register wäre dann also notwendig.

Das zweite Flag (Variable chCorrect) verwendet der Channel Filter, um sich zu merken, ob die channel number des aktuellen Pakets korrekt ist. Nur wenn chCorrect = 1 gilt, werden isochrone Daten an den Zeilenfilter übertragen.

Die einzelnen Signale des ISO Channel Filters wurden bereits im Abschnitt 2.2 erläutert.

Kapitel 4

Zusammenfassung und Ausblick

Mit der entwickelten ioFSM-Komponente und dem ISO Channel Filter sollte es möglich sein, eine effiziente Übertragung der Bilddaten zu realisieren. Der Entwurf erfüllt die gestellte Forderung nach Trennung der isochronen von den asynchronen Daten. Die NIOS-CPU muss nur asynchrone Pakete entgegennehmen und braucht sich um die Übertragung der isochronen Daten nicht zu kümmern. Die Rechenleistung der CPU kann nahezu vollständig für die Auswertung der Bilddaten genutzt werden. Dem Einsatz von, in C geschriebenen, Bildverarbeitungsalgorithmen stünde damit nichts im Wege.

Die Implementierung der ioFSM-Komponente und des ISO Channel Filters in VHDL ist auch insofern abgeschlossen, dass die Simulation der Entwürfe mit einem VHDL-Simulator positive Ergebnisse lieferte. Die simulierten Entwürfe zeigten gewünschtes Verhalten.

Die Systemintegration mit dem Zeilenfilter und den, für die NIOS-CPU in C geschriebenen, Programmen findet jedoch noch statt. Die Aufeinanderabstimmung, der im Projekt entwickelten Komponenten erwies sich als keine einfache Aufgabe, zumal die Debugging-Möglichkeiten auf dieser hardwarenahen Ebene relativ beschränkt sind.

Anhang A

VHDL-Quellcodes

A.1 ioFSM

```
use ieee.numeric_std.all;
use work.niosPkg.all;
use work.tsb12Pkg.all;

entity ioFsm2 is
  generic (CPU_QUEUE_LENGTH : integer := 256);
  port (clk         : in    std_logic;      -- clock 33MHz
        nRst        : in    std_logic;      -- reset, async L
        addr        : out   addrTy;         -- address

        -- data bus
        data        : inout std_logic_vector (0 to 31);

        nWr         : out   std_logic;      -- write ena.   L
        nCS         : out   std_logic;      -- cycle start  L
        nCA         : in    std_logic;      -- cycle ack.   L
        nIntr       : in    std_logic;      -- interrupt    L
        niosTrg     : in    std_logic;      -- cmd-trigger

        -- command
        niosCmd     : in    std_logic_vector (0 to  1);

        niosA       : in    addrTy;         -- addr->tsb12

        -- data->tsb12
        niosWr      : in    std_logic_vector (0 to 31);

        -- data<-tsb12
        niosRd      : out   std_logic_vector (0 to 31);

        niosRdF     : out   std_logic;      -- new data flag
        tsbRdy      : out   std_logic;      -- tsb12 ready  H
```

```
        tsbIrq      : out   std_logic;     -- tsb12 IRQ    H

        -- our extensions

        IsoDta       : out    std_logic_vector(0 to 31);
        IsoTokenTrg  : out    std_logic;
        IsoHeaderTrg : out    std_logic;
        IsoDataTrg   : out    std_logic;
end entity ioFsm2;

------------------------------------------------------------

architecture behave of ioFsm2 is

  type stateTy is (idleSt, intRdSt, intMaskRdSt, rdSt,
                   wrSt, ackSt, grfStatusSt, grfStatusAckSt,
                   grfNoTokenPrepRdSt, grfNoTokenRdSt,
                   grfTokenPrepRdSt, grfTokenRdSt,
                   grfAsyDataRdSt, grfIsoHeaderRdSt,
                   grfIsoDataRdSt);

  constant GRFSIZE : std_logic_vector (9 downto 0)
                     := "1000000000";

  signal State     : stateTy;
  signal dataEn    : std_logic;
  signal dataReg   : std_logic_vector (0 to 31);
  signal RxDtaMask : std_logic;
  signal lastTrg   : std_logic;              -- last-trigger

  -- fifo
  signal data_sig  : STD_LOGIC_VECTOR (33 DOWNTO 0);
  signal wrreq_sig : STD_LOGIC ;
  signal rdreq_sig : STD_LOGIC ;
  signal clock_sig : STD_LOGIC ;
  signal aclr_sig  : STD_LOGIC ;
  signal q_sig     : STD_LOGIC_VECTOR (33 DOWNTO 0);
  signal full_sig  : STD_LOGIC ;
  signal empty_sig : STD_LOGIC ;
  signal usedw_sig : STD_LOGIC_VECTOR (7 DOWNTO 0);

begin

  fifo_inst : fifo PORT MAP (
    data     => data_sig,
    wrreq    => wrreq_sig,
    rdreq    => rdreq_sig,
    clock    => clock_sig,
    aclr     => aclr_sig,
```

```
    q        => q_sig,
    full     => full_sig,
    empty    => empty_sig,
    usedw    => usedw_sig
    );

  clock_sig <= clk;

  tsbRdy <= '1' when niosTrg = lastTrg else
            '0';

  -- tristate driver
  triS: data <= dataReg when dataEn = '1' else
                (others => 'Z');

  intr: tsbIrq <= '1' when (nIntr = '0') or
                          ((empty_sig = '0') and
                           (RxDtaMask = '1')) else
                  '0';

  fsmP: process (clk, nRst) is

  variable writeCount
             : unsigned (0 to 8);
  variable dataCache
             : std_logic_vector (0 to 31);
  variable fifoRequested
             : boolean;
  variable temp_fifo_emptyFlag
             : std_logic;
  variable usedw_var
             : std_logic_vector (7 downto 0);

  procedure prepareRead (address : in addrTy) is
  begin
    addr <= address;
    nCS  <= '0';
    nWr  <= '1';
  end procedure prepareRead;

  procedure writeIsoData (itt : in std_logic;
                          iht : in std_logic;
                          idt : in std_logic) is
  begin
    IsoDta       <= data;
    IsoTokenTrg  <= itt;
    IsoHeaderTrg <= iht;
    IsoDataTrg   <= idt;
  end procedure writeIsoData;
```

```
begin
  if nRst = '0' then
    state        <= idleSt;
    addr         <= (others => '0');
    dataEn       <= '0';
    nCS          <= '1';
    RxDtaMask    <= '0';
    lastTrg      <= '0';
    IsoTokenTrg  <= '0';
    IsoHeaderTrg <= '0';
    IsoDataTrg   <= '0';
    aclr_sig     <= '1';
    writeCount          := (others => '0');
    fifoRequested       := false;
    temp_fifo_emptyFlag := '0';
  elsif falling_edge(clk) then
    case State is
      when idleSt =>
        aclr_sig <= '0';
        if lastTrg /= niosTrg then   -- niosCommands
          case niosCmd is
            when rdCyc =>
              if niosA = a_GRFStatus then  -- rd grfStatus
                                           -- (fifo)
                if not fifoRequested then
                  fifoRequested       := true;
                  temp_fifo_emptyFlag := empty_sig;
                  usedw_var           := usedw_sig;
                  rdreq_sig <= '1';
                else
                  rdreq_sig <= '0';
                  niosRd    <= empty_sig &
                               q_sig(1 downto 0) & "00" &
                               usedw_var & GRFSIZE &
                               q_sig(18 downto 10);
                  lastTrg   <= niosTrg;
                  if temp_fifo_emptyFlag = '1' then
                    fifoRequested := false;
                  end if;
                end if;
              elsif niosA = a_GRFData then  -- rd grfData
                                            -- (fifo)
                if not fifoRequested then
                  fifoRequested := true;
                  rdreq_sig <= '1';
                else
                  fifoRequested := false;
                  rdreq_sig <= '0';
```

```
                niosRd    <= q_sig(33 downto 2);
                lastTrg   <= niosTrg;
              end if;

            -- rd interrupt register
            elsif niosA = a_Interrupt then
              prepareRead(niosA);
              State <= intRdSt;
            elsif niosA = a_InterruptMask then
              prepareRead(niosA);
              State <= intMaskRdSt;
            else                      -- other rd address
              prepareRead(niosA);
              State <= rdSt;
            end if;
          when wrCyc =>
            addr <= niosA;

            -- wr interrupt-mask register
            if niosA = a_InterruptMask then
              dataReg   <= (niosWr(0 to 5) & '0' &
                             niosWr(7 to 31));
              RxDtaMask <= niosWr(6);
            else                      -- other wr address
              dataReg <= niosWr;
            end if;
            dataEn <= '1';
            nCS    <= '0';
            nWr    <= '0';
            State  <= wrSt;
          when others => null;
        end case;
      else    -- lastTrg = niosTrg -> rd grfStatus (LLC)
        prepareRead(a_GRFStatus);
        State <= grfStatusSt;
      end if;
    when intRdSt =>
      nCS <= '1';
      if nCA = '0' then
        niosRd  <= (data(0 to 5) & (not empty_sig) &
                    data(7 to 31));
        lastTrg <= niosTrg;
        State   <= ackSt;
      end if;
    when intMaskRdSt =>
      nCS <= '1';
      if nCA = '0' then
        niosRd  <= (data(0 to 5) & RxDtaMask &
                    data(7 to 31));
```

```
            lastTrg <= niosTrg;
            State   <= ackSt;
          end if;
        when rdSt =>
          nCS <= '1';
          if nCA = '0' then
            niosRd  <= data;
            lastTrg <= niosTrg;
            State   <= ackSt;
          end if;
        when wrSt =>
          nCS <= '1';
          if nCA = '0' then
            lastTrg <= niosTrg;
            State   <= ackSt;
          end if;
        when ackSt =>
          dataEn       <= '0';
          nCS          <= '1';
          wrreq_sig    <= '0';
          IsoTokenTrg  <= '0';
          IsoHeaderTrg <= '0';
          IsoDataTrg   <= '0';
          State        <= idleSt;

-- Access to the GRF (Burst Read) --
------------------------------------------------------------

        when grfStatusSt =>
          nCS <= '1';
          if nCA = '0' then
            dataCache := data;
            State <= grfStatusAckSt;
          end if;
        when grfStatusAckSt =>
          if dataCache(1) = '1' then      -- if cd = 1
            writeCount := unsigned(dataCache(23 to 31));
            State <= grfTokenPrepRdSt;
          elsif dataCache(0) = '0' then   -- if cd = 0 and
                                          -- empty = 0
            State <= grfNoTokenPrepRdSt;
          else                            -- if cd = 0 and
                                          -- empty = 1
            State <= idleSt;
          end if;
        when grfNoTokenPrepRdSt =>
          prepareRead(a_GRFData);      -- rd GRF (no token)
          State <= grfNoTokenRdSt;
        when grfNoTokenRdSt =>
```

```
  nCS <= '1';
  if nCA = '0' then
    State <= ackSt;
  endif;
when grfTokenPrepRdSt =>
  prepareRead(a_GRFData);      -- rd GRF (token)
  State <= grfTokenRdSt;
when grfTokenRdSt =>

  -- burst rd GRF -> nCS remains set to 0
  -- if writeCount /= 0
  if nCA = '0' then
    if data(24 to 27) = x"A" then -- if tCode = Ah
                                  -- -> ISO
      if writeCount /= 0 then

        -- decrement before reading first
        -- not-token-quadlet
        writeCount := writeCount - 1;

        -- wr token-quadlet to ChFil
        writeIsoData('1', '0', '0');
        State <= grfIsoHeaderRdSt;
      else               -- writeCount = 0 -> not wr
                         -- token_quadlet to ChFil
        nCS   <= '1';
        State <= ackSt;
      end if;
    else        -- ASYN
      if writeCount /= 0 then

        -- decrement before reading first
        -- not-token-quadlet
        writeCount := writeCount - 1;
        State <= grfAsyDataRdSt;
      else                        -- writeCount = 0
        nCS   <= '1';
        State <= ackSt;
      end if;

      -- wr token-quadlet to fifo
      data_sig  <= (data & '1' & data(11));
      wrreq_sig <= '1';
    end if;
  end if;
when grfAsyDataRdSt =>
  if nCA = '0' then
    if writeCount /= 0 then     -- not last quadlet
      writeCount := writeCount - 1;
```

```
          else                          -- last quadlet
            nCS       <= '1';
            State     <= ackSt;
          end if;

          -- wr quadlet to fifo
          data_sig  <= (data & "00");
          wrreq_sig <= '1';
        end if;
      when grfIsoHeaderRdSt =>
        if nCA = '0' then
          if writeCount /= 0 then     -- not last quadlet
            writeCount := writeCount - 1;
            State <= grfIsoDataRdSt;

          -- last quadlet -> no data-quadlets
          -- (only header)
          else
            nCS   <= '1';
            State <= ackSt;
          end if;

          -- wr header-quadlet to ChFil
          writeIsoData('0', '1', '0');
        end if;
      when grfIsoDataRdSt =>
        if nCA = '0' then
          if writeCount /= 0 then
            writeCount := writeCount - 1;
          else                          -- last quadlet
            nCS   <= '1';
            State <= ackSt;
          end if;
          writeIsoData('0', '0', '1');  -- wr quadlet to
                                        -- ChFil
        end if;
      when others => null;
    end case;
  end if;
end process fsmP;
end architecture behave;
```

A.2 ISO Channel Filter

```
library ieee;
use ieee.std_logic_1164.all;
```

```
entity ChannelFilter is
  port (Clk          : in     std_logic;
        nRst         : in     std_logic;
        DataIn       : in     std_logic_vector (0 to 31);
        DataOut      : out    std_logic_vector (0 to 31);
        Channel      : in     std_logic_vector (0 to 5);
        IsoTokenTrg  : in     std_logic;
        IsoHeaderTrg : in     std_logic;
        IsoDataTrg   : in     std_logic;
        SyFlagOut    : out    std_logic;
        DataTrgOut   : out    std_logic);
end entity ChannelFilter;


architecture Behav_ChFil of ChannelFilter is
begin

  Filter: process (Clk, nRst)
    variable chCorrect : std_logic;
    variable synchFlag : std_logic;
    variable token     : std_logic_vector (0 to 31);
  begin
    if nRst = '0' then
      synchFlag := '0';
      chCorrect := '0';
      DataOut    <= (others => '0');
      SyFlagOut  <= '0';
      DataTrgOut <= '0';
    elsif rising_edge(Clk) then
      if IsoTokenTrg = '1' then
        token := DataIn;
      elsif IsoHeaderTrg = '1' then
        if Channel = DataIn(18 to 23) then
          chCorrect := '1';
          if DataIn(28 to 31) = "0001" then
            synchFlag := '1';
          else
            synchFlag := '0';
          end if;
        else
          chCorrect := '0';
          synchFlag := '0';
        end if;
        SyFlagOut  <= '0';
        DataTrgOut <= '0';
      elsif IsoDataTrg = '1' and chCorrect = '1' then
        DataOut    <= DataIn;
        DataTrgOut <= '1';
        if synchFlag = '1' then
```

```
          synchFlag := '0';
          SyFlagOut <= '1';
        else
          SyFlagOut <= '0';
        end if;
      else
      -- channel number incorrect or IsoDataTrg = '0'
        DataTrgOut <= '0';
        SyFlagOut  <= '0';
      end if;
    end if;
  end process Filter;
end architecture Behav_ChFil;
```

A.3 Adresskonstanten

```
-- Address constants              ---------------------------------
constant c_Version                : byteTy        := x"00";
constant c_NodeAddress            : byteTy        := x"04";
constant c_Control                : byteTy        := x"08";
constant c_Interrupt              : byteTy        := x"0c";
constant c_InterruptMask          : byteTy        := x"10";
constant c_CycleTimer             : byteTy        := x"14";
constant c_IsochPortNumber        : byteTy        := x"18";
constant c_FIFOControl            : byteTy        := x"1c";
constant c_Diagnostics            : byteTy        := x"20";
constant c_PHYChipAccess          : byteTy        := x"24";
constant c_ATFStatus              : byteTy        := x"30";
constant c_ITFStatus              : byteTy        := x"34";
constant c_GRFStatus              : byteTy        := x"3c";
constant c_HostControl            : byteTy        := x"40";
constant c_MuxControl             : byteTy        := x"44";
constant c_ATFFirst               : byteTy        := x"80";
constant c_ATFContinue            : byteTy        := x"84";
constant c_ATFUpdate              : byteTy        := x"8c";
constant c_ITFFirst               : byteTy        := x"90";
constant c_ITFContinue            : byteTy        := x"94";
constant c_ITFUpdate              : byteTy        := x"9c";
constant c_ATFBurstWrite          : byteTy        := x"a0";
constant c_ITFBurstWrite          : byteTy        := x"b0";
constant c_GRFData                : byteTy        := x"c0";

-- Address (value)                ---------------------------------
constant a_Version                : addrTy := c_Version(0 to 5);
constant a_NodeAddress            : addrTy := c_NodeAddress(0 to 5);
constant a_Control                : addrTy := c_Control(0 to 5);
constant a_Interrupt              : addrTy := c_Interrupt(0 to 5);
constant a_InterruptMask          : addrTy := c_InterruptMask(0 to 5);
```

```
constant a_CycleTimer           : addrTy := c_CycleTimer(0 to 5);
constant a_IsochPortNumber      : addrTy := c_IsochPortNumber(0 to 5);
constant a_FIFOControl          : addrTy := c_FIFOControl(0 to 5);
constant a_Diagnostics          : addrTy := c_Diagnostics(0 to 5);
constant a_PHYChipAccess        : addrTy := c_PHYChipAccess(0 to 5);
constant a_ATFStatus            : addrTy := c_ATFStatus(0 to 5);
constant a_ITFStatus            : addrTy := c_ITFStatus(0 to 5);
constant a_GRFStatus            : addrTy := c_GRFStatus(0 to 5);
constant a_HostControl          : addrTy := c_HostControl(0 to 5);
constant a_MuxControl           : addrTy := c_MuxControl(0 to 5);
constant a_ATFFirst             : addrTy := c_ATFFirst(0 to 5);
constant a_ATFContinue          : addrTy := c_ATFContinue(0 to 5);
constant a_ATFUpdate            : addrTy := c_ATFUpdate(0 to 5);
constant a_ITFFirst             : addrTy := c_ITFFirst(0 to 5);
constant a_ITFContinue          : addrTy := c_ITFContinue(0 to 5);
constant a_ITFUpdate            : addrTy := c_ITFUpdate(0 to 5);
constant a_ATFBurstWrite        : addrTy := c_ATFBurstWrite(0 to 5);
constant a_ITFBurstWrite        : addrTy := c_ITFBurstWrite(0 to 5);
constant a_GRFData              : addrTy := c_GRFData(0 to 5);
```

Literaturverzeichnis

[1] **Andreas Mäder:** Skript - VHDL Kompakt.

[2] **Andreas Mäder:** Skript - VHDL Syntax und Praxis.

[3] **David Pellerin, Douglas Taylor:** VHDL Made Easy!, Prentice Hall, Inc., 1997.

[4] **Gunther Lehmann, Bernhard Wunder, Manfred Selz:** Schaltungsdesign mit VHDL, Franzis'-Verlag, 1994.

http://www.itiv.uni-karlsruhe.de/opencms/opencms/de/study/ lectures/liv1/vhdl_download.html

[5] **Texas Instruments:** Data Manual - TSB12LV01B IEEE 1394-1995 High-Speed Serial-Bus Link-Layer Controller, 2002.

http://focus.ti.com/docs/prod/folders/print/tsb12lv01b.html

[6] **Texas Instruments:** Data Manual - TSB41AB3 IEEE 1394a-2000 Three-Port Cable Transceiver/Arbiter, 2003.

http://focus.ti.com/docs/prod/folders/print/tsb41ab3.html

[7] **Sony:** Technical Manual - CCD Color Digital Camera Module DFW-V500/ DFW-VL500 (Ver. 1.0).

www.sony.net/Products/ISP/pdf/guide/GDFWV500E.pdf

[8] **Altera:** User Guide - SOPC Builder (Ver. 1.0), 2003.

http://www.altera.com/literature/ug/ug_sopcbuilder.pdf

[9] **Altera:** NIOS Embedded Processor 32-Bit Programmer's Reference Manual (Ver. 3.1), 2003.

http://www.altera.com/literature/manual/mnl_nios_programmers32.pdf

[10] **Altera:** Reference Manual - Avalon Interface Specification (Ver. 2.4), 2004.

[11] **Don Anderson:** FireWire System Architecture: IEEE 1394a, MindShare, Inc., 2nd Ed. 1998.