DIPLOMARBEIT

Entwurf von Soft-Cores für eine FPGA-basierte PCI-Karte für die Echtzeitbildverarbeitung

Markus Köchy

1. August 2003

Erstgutachter:
Dr.-Ing. habil. G. Kemnitz
Institut für Informatik
TU Clausthal

Zweitgutachter:
Prof. Dr. C. Siemers
Institut für Informatik
TU Clausthal

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Clausthal-Zellerfeld, 1. August 2003

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung und Aufgabenstellung

Die Diplomarbeit ist Teil eines Projekts, das sich mit der Entwicklung eines visuellen Systems für einen mobilen Industrieroboter befasst. Aus den von zwei Kameras gelieferten Bildfolgen soll ein dreidimensionales Modell für die Umgebung des Roboters in Echtzeit berechnet werden.

Der hierfür erforderliche Rechenaufwand liegt deutlich über dem, was heutige Rechner (Standard-PCs) leisten können. Deshalb sollen die ersten Vorverarbeitungsschritte in Hardware mit einem FPGA (Field Programmable Gate Array) realisiert werden. Die vorverarbeiteten Daten werden anschließend über den PCI-Bus in den Computer zur Weiterverarbeitung übertragen.

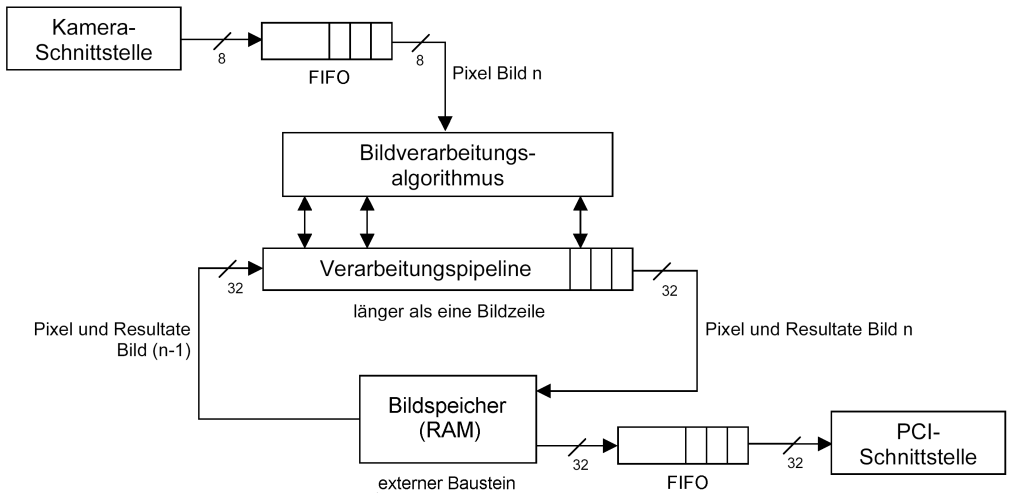

Abbildung 1.1: Entwurfsskizze des Framegrabbers mit Bildvorverarbeitung

Abbildung 1.1 skizziert die angestrebte Gesamtstruktur. Der Bildverarbeitungsalgorithmus ist als Black Box dargestellt. Über die Kameraschnittstelle und den nachgeschalteten FIFO-Speicher treffen die Pixel des aktuellen Bildes ein. Zugleich hat das Bildverarbeitungsmodul Zugriff auf die Pixel und Resultate des vorherigen Bildes. Sie werden aus dem Bildspeicher gelesen und synchron zum aktuellen Bild in der Verarbeitungspipeline „vorbeigeschoben“. Dort stehen sie zunächst als Operand zur Verfügung und werden später durch die Pixel und Ergebnisse des aktuellen Bildes ersetzt. Auf diese Weise kann der Algorithmus auch benachbarte Pixel des aktuellen Bildes in die Rechnung einbeziehen. Die Resultatdaten werden schließlich an einer anderen Stelle des Bildspeichers gespeichert, von wo sie über die PCI-Schnittstelle

vom Computer abgerufen werden können. Bis auf den Bildspeicher lassen sich alle Module im FPGA integrieren.

Ziel dieser Diplomarbeit war es, die technischen Voraussetzungen für die Kopplung von FPGA und PC über den PCI-Bus zu schaffen.

Als technologische Plattform wurde das Spartan-II 200 PCI Development Board von der Firma Memec Design ausgewählt. Zentrale Komponenten sind ein leistungsfähiger Spartan-II FPGA mit direktem PCI-Anschluss und ein SDRAM-Baustein.

Verarbeitet werden die Grauwertbilder von zunächst nur einer Hochgeschwindigkeitskamera, die 120 Vollbilder pro Sekunde liefert. Sie wird über eine Erweiterungsplatine, die im Rahmen einer parallel laufenden Diplomarbeit [5] entworfen wurde, an das Entwicklungsboard angeschlossen.

Die Verbindung der eigenen Hardwareentwürfe mit dem PCI-Bus erleichtert das LogiCORE PCI Interface von der Firma Xilinx. Es handelt sich dabei um ein vorimplementiertes und getestetes Modul für Xilinx Spartan- und Virtex-FPGAs, für das das Institut für Informatik eine Lizenz erworben hat.

Die Entwicklung erfolgte in der Entwicklungsumgebung Xilinx ISE 5.1i mit der Hardwarebeschreibungssprache VHDL.

Im einzelnen waren folgende Teilaufgaben zu bearbeiten:

- Einrichten der Entwicklungssoftware und Inbetriebnahme des Entwicklungsboards mit Beispiel- und Testdesigns
- Einarbeitung in den Entwurf von PCI-Interfaces
- Entwurf eines auf die Bildverarbeitungsanwendung ausgerichteten PCI-Busmasters
- Programmierung von zugehörigen Gerätetreibern und Anwendersoftware unter Linux
- Ausführliche Dokumentation aller Arbeitsschritte in einer Form, die nachfolgenden Bearbeitern den Einstieg in die Thematik und die Fortsetzung des Projekts erleichtert.

2 Das Spartan-II 200 PCI Development Board

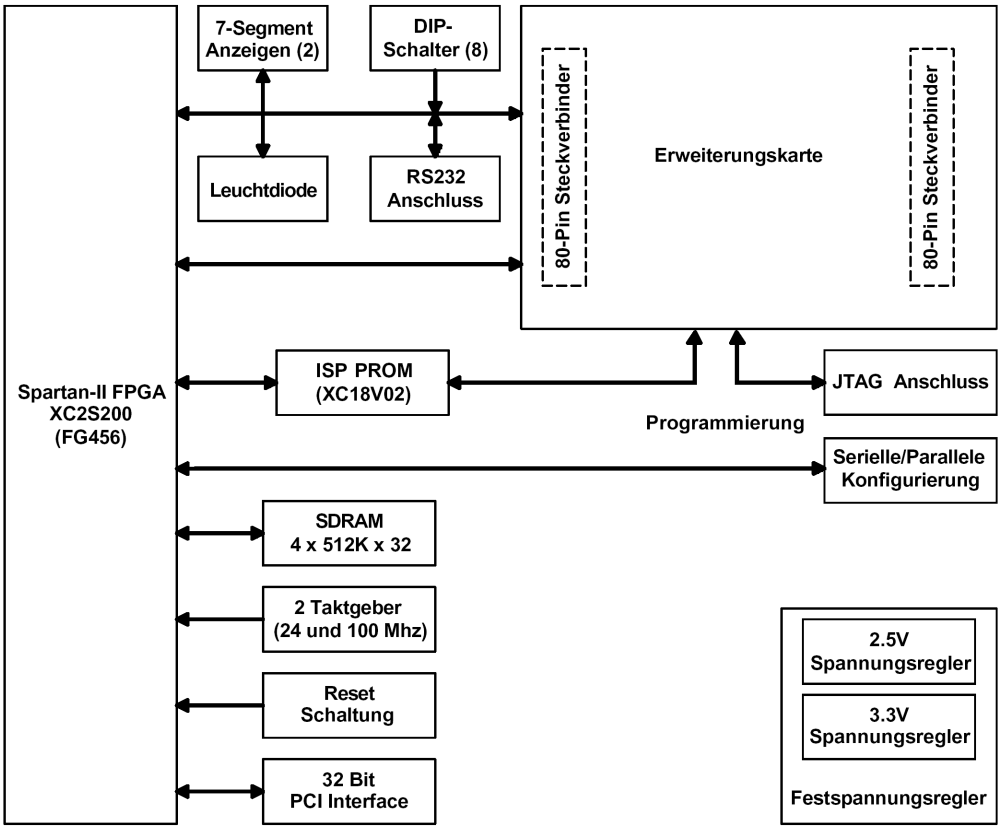

Abbildung 2.1: Blockdiagramm des Spartan-II 200 PCI Development Boards

Dieses Kapitel gibt einen Überblick über das Spartan-II 200 PCI Development Board. Eine genaue technische Dokumentation enthält das Benutzerhandbuch [7].

Abbildung 2.1 zeigt die Komponenten der Entwicklungskarte schematisch als Blockdiagramm. Der zentrale Baustein ist das FPGA XC2S200, der größte Vertreter der Spartan-II Architektur von Xilinx [10]. Um ihn herum sind zwei Taktgeber (24 MHz und 100 MHz), der 32 Bit PCI-Anschluss, ein SDRAM-Baustein und Möglichkeiten zur direkten Dateneingabe und -ausgabe gruppiert.

Weiterhin befinden sich auf der Karte zwei 80-polige Steckverbinder (P160 Expansion Slot), über die die Hardware anwendungsspezifisch erweitert werden kann.

2.1 Das SDRAM

Das 64 Mbit SDRAM HY57V653220BTC-7 der Firma Hynix [8] ist intern in 4 Bänken organisiert, die jeweils 512 K Speicherfelder (2048 Zeilen × 256 Spalten) mit 32 Bit Wortbreite umfassen. Alle Ein- und Ausgangssignale sind mit der steigenden Flanke des Systemtakts synchronisiert, der maximal 143 MHz betragen darf. Über ein Konfigurationsregister ist die Anzahl der Datenzyklen eines Lese- oder Schreibtransfers (Burstlänge) sowie die CAS-Zugriffszeit (CAS-Latency) programmierbar. Ein laufender Transfer kann jederzeit mit dem Terminierungsbefehl beendet oder durch einen neuen Lese- bzw. Schreibbefehl abgebrochen werden. Zudem enthält das SDRAM Mechanismen, die die notwendige periodische Auffrischung der Speicherzellen vereinfacht.

Bemerkung: Die Datenblätter des SDRAMs setzen Vorwissen über die grundsätzliche Funktionsweise von dynamischen RAMs voraus. Diese kann beispielsweise in [1] gut nachvollzogen werden.

2.2 I/O-Komponenten

Für die direkte Dateneingabe und -ausgabe verfügt die PCI-Karte über zwei 7-Segment-Anzeigen, acht DIP-Schiebeschalter, einen Taster und die technischen Voraussetzungen für eine serielle RS232-Schnittstelle.

Bei der seriellen Schnittstelle sind lediglich die beiden Datenleitungen RxD und TxD ausgeführt. Zwischen FPGA und dem seriellen Anschluss passt der Wandler MAX3221 die Signalpegel an. Die Schnittstellenlogik ist komplett im FPGA zu implementieren.

Für jedes Segment der 7-Segment-Anzeige ist eine separate Leitung zum FPGA vorgesehen.

2.3 Kamera- und Monitoranschluss

Für den Anschluss der Pulnix TM-6710 Digitalkamera [9] und eines VGA-Monitors zur Bildausgabe hat Carsten Giesemann im Rahmen seiner Diplomarbeit eine für den P160 Expansion Slot passende I/O-Platine [5] entworfen. Sie besteht im Wesentlichen aus Treiberelektronik für die Kamerasignale und einem 3 × 10 Bit Digital-Analog-Wandler zur Erzeugung der RGB-Farbsignale des VGA-Anschlusses. Die Steuersignale für die VGA-Bildsynchronisation hingegen müssen im FPGA generiert werden.

Die Kamera liefert 120 Grauwertbilder pro Sekunde bei der vollen Auflösung von 640 × 480 Pixeln. Zu der Erweiterungsplatine gehört ein synthesefähiges VHDL-Modul, das die Kamera ansteuert und VGA-kompatible Daten- und Steuersignale für den Monitoranschluss bzw. für den Digital-Analog-Wandler erzeugt. Die Beispielanwendung Framegrabber in Kapitel 8 macht von diesem Modul unmittelbar Gebrauch.

2.4 Programmierung

Der Konfigurationsspeicher des FPGA ist flüchtig; nach dem Einschalten der Spannungsversorgung muss das FPGA von außen konfiguriert werden. Wie in Abbildung 2.1 dargestellt, ist dies entweder direkt über die JTAG-Programmierschnittstelle oder durch das nicht-flüchtige ISP PROM auf der Karte, das seinerseits zuvor über die JTAG-Schnittstelle programmiert werden muss, möglich.

Für beide Fälle liegt dem Entwicklungsboard ein Adapterkabel bei, mit dem die Karte über den JTAG-Anschluss mit der parallelen Schnittstelle des Entwicklungsrechners verbunden werden kann. Auf der Karte muss der Jumper J25 gesetzt werden, um die JTAG-Kette, die auch über den P160 Expansion Slot führt, zu schließen. Außerdem wird der Jumper J12 in Position 2-3 gebracht.

2.4.1 Programmierung des ISP PROMs

Um das ISP PROM über den JTAG-Anschluss zu programmieren, sind die folgenden vier Schritte durchzuführen:

1. Der Konfigurationsmodus des FPGAs wird gemäß Tabelle 2.1 auf den *Master Serial Mode* eingestellt.
2. Mit dem Xilinx Programmierwerkzeug *iMPACT* wird die Konfigurationsdatei für das FPGA (*.bit) in die Binärdatei für das PROM (*.mcs) eingebettet. Dazu wählt man in dem Menü *Operation Mode Selection*, das beim Öffnen der Anwendung angezeigt wird, die Option *Prepare Configuration Files*. Das Programm startet daraufhin einen Assistenten, der die weiteren Schritte begleitet. Alternativ ist dieser Assistent im laufenden Programm über *Edit* → *Launch Wizard...* zu erreichen.
3. *iMPACT* wird erneut gestartet. Mit der zuvor erzeugten Datei (*.mcs) kann nun das PROM programmiert werden. Da sich das FPGA in der JTAG-Kette befindet, muss ihm die Dummy-Datei xc2s200_fg456.bsd aus dem Xilinx ISE Programmordner als Platzhalter zugeordnet werden.
4. Nach der Programmierung löst ein Druck des PROGn-Tasters auf dem Entwicklungsboard oder das Aus- und Einschalten der Spannungsversorgung die FPGA-Konfiguration über das PROM aus.

Konfigurations-	**Pfostenleiste J1**		
Modus	**1 – 2**	**3 – 4**	**5 – 6**
JTAG	offen	geschlossen	geschlossen
Master Serial	geschlossen	geschlossen	geschlossen

Tabelle 2.1: Wahl des Konfigurationsmodus

2.4.2 Direkte Konfiguration des Spartan-II FPGAs

Um das FPGA direkt über den JTAG-Anschluss zu konfigurieren, sind die folgenden beiden Schritte durchzuführen:

1. Der Konfigurationsmodus des FPGAs wird gemäß Tabelle 2.1 auf den *JTAG Mode* eingestellt.
2. Mit dem Xilinx Programmierwerkzeug *iMPACT* kann die Binärdatei (*.bit), die den Entwurf enthält, in das FPGA geladen werden. Dem PROM, das sich in der JTAG-Kette befindet, muss die Dummy-Datei xc18v02_vq44.bsd aus dem Xilinx ISE Programmordner als Platzhalter zugeordnet werden.

Bei der Verwendung des Entwicklungsboards als PCI-Karte in einem PC[1] kann diese Art der Programmierung zu einem Konflikt mit der PCI-Kartenerkennung durch das PCI-BIOS des PCs führen. Die Ursache liegt darin, dass das PCI-BIOS das Hinzufügen von PCI-Karten im laufenden Betrieb nicht vorsieht: Der PC erwartet beim Einschalten bzw. nach der Einschalt-Resetphase funktionierende PCI-Karten. Steckt das Entwicklungsboard unkonfiguriert in einem der PCI-Slots, wird dieser Slot auch nach Systemresets ignoriert. Zur FPGA-Konfiguration muss der PC jedoch eingeschaltet sein, um das Entwicklungsboard mit Energie zu versorgen.

Eine Lösung für dieses Problem ist, das FPGA beim Einschalten des Rechners wie im letzten Abschnitt beschrieben über das ISP PROM mit einem funktionierenden Design zu konfigurieren. Hat das PCI-BIOS erst einmal erkannt, dass ein PCI-Gerät im betreffenden Slot vorhanden ist, kann das FPGA durch die direkte Konfiguration umprogrammiert werden. Nach einem Systemreset initialisiert das PCI-BIOS alle PCI-Karten neu. Unter bestimmten Bedingungen kann sogar auf den Neustart des Rechners verzichtet werden, siehe Abschnitt 7.5.

[1] Testrechner war ein Microstar 815E Pro Mainboard mit Pentium-III 700 MHz Prozessor

3 Der PCI-Bus

3.1 Überblick über den PCI-Bus

PCI (Peripheral Component Interconnect) ist eine von der Firma Intel geschaffene und erstmals im Juni 1992 veröffentlichte Busnorm zur Verbindung verschiedendster Computerperipherie. Intel schaffte es, eine vollständige, realistische und gut strukturierte Spezifikation für ein nach oben hin offenes Bussystem durchzusetzen, die neben der funktionalen Beschreibung auch das elektrische Verhalten, mechanische Vorgaben und eine präzise Beschreibung des Zeitverhaltens enthält. Im Gegensatz zu dem auf x86-Architekturen festgelegten und inkonsequent genormten ISA-Bus [3] ist der PCI-Bus nicht an die Intel-Architektur gebunden. Er ersetzte schnell den leistungsschwachen ISA-Bus und vereinheitlichte bzw. beendete die Versuche anderer Computerhersteller, den Flaschenhals ISA-Bus zu überwinden[1].

Die Spezifikation wird inzwischen von einem Industriekonsortium, der *PCI Special Interest Group*[2] (PCI SIG), verwaltet. Die Ausführungen in diesem Kapitel beruhen auf dem Buch [2], dessen Herausgeber Mitglied der PCI SIG ist.

3.2 Merkmale und Vorteile des PCI-Busses

Prozessorunabhängigkeit

Die Anbindung des PCI-Busses an einen spezifischen Prozessorbus erfolgt mit einer Hostbridge. Die für den PCI-Bus entworfenen Komponenten sind PCI-spezifisch, nicht prozessorspezifisch.

Busgeschwindigkeit und Busbreite

Bis zur Spezifikation 2.0 war der maximale Bustakt auf 33 MHz festgelegt. Seit der Version 2.1 gibt es auch eine Variante mit 66 MHz Bustakt. Die Busbreite beträgt 32 Bit und ist auf 64 Bit erweiterbar.

[1] Ende der 80er Jahre gab es mehrere Alternativen für den ISA-Bus, z.B. den MCA-Bus (Micro Channel Architecture) von IBM, den EISA-Bus (enhanced ISA) und den VLB-Bus (VESA Local Bus).
[2] http://www.pcisig.com

Bursttranfers

Eine markante Eigenschaft des PCI-Busses ist das „Bursten“, also das einmalige Übertragen einer Adresse gefolgt von mehreren aufeinanderfolgenden Datenwörtern. Ein Bursttransfer kann im Prinzip beliebig lang sein, wenn dem Sender nicht die Busrechte entzogen werden.

Ein 32 Bit PCI-Bus mit 33 MHz Bustakt kann damit eine maximale Transferrate von 132 MByte pro Sekunde erzielen[3], bei der maximalen Ausbaustufe (64 Bit bei 66 MHz Bustakt) 528 MByte pro Sekunde.

Busmaster-Unterstützung

Sofern implementiert, kann ein PCI-Gerät über spezielle Steuerleitungen den PCI-Bus anfordern und Ende-zu-Ende Lese- oder Schreibtransaktionen mit einem anderen PCI-Gerät durchführen. In der PCI-Terminologie heißen die Kommunikationspartner *Initiator* (Master) und *Target* (Slave). Die Bridges (Hostbridge, PCI-to-PCI Bridge, PCI-to-ISA Bridge, ...) vermitteln dabei transparente Transfers über den PCI-Bus des Initiators hinaus mit dem Hauptspeicher und mit an anderen Bussen angeschlossenen Geräten.

Drei separate Adressräume

Hierbei handelt es sich um den Speicher-, den I/O- und den Konfigurationsadressraum.

Plug-and-Play

Jedes PCI-Gerät verfügt über genau spezifizierte Konfigurationsregister im Konfigurationsadressraum (*PCI Configuration Space Header*), die die automatische Erkennung und Konfiguration des PCI-Geräts durch das PCI-BIOS oder durch das Betriebssystem ermöglichen.

Einfache Mechanik

Bedingt durch das Zusammenfassen der Adressleitungen und Datenleitungen durch Zeitmultiplex benötigt ein minimales PCI-Target lediglich 47 Leitungen, ein Initiator 49 Leitungen. Die PCI-Chips können daher relativ klein gehalten werden. Der PCI-Platinenstecker kann sehr kostengünstig hergestellt werden.

[3] In der Literatur findet man manchmal 133 MHz. 33,0 MHz × 32 Bit sind aber 132 MHz.

Schwache Leitungstreiber

Der PCI-Bus ist auf elektrischer Ebene nicht terminiert. Die geschickte Nutzung der Signalreflexion an den offenen Enden erlaubt die Verwendung relativ schwacher Ausgangstreiber in den PCI-Geräten: Der Treiber bringt den Spannungspegel seiner Leitung nur auf den halben Zielwert. Die Wellenfront erreicht das unterminierte Busende und wird reflektiert. Beim Zurücklaufen addieren sich die Spannungen. Die hochohmigen Eingangstreiber des PCI-Geräts, das von der Wellenfront gerade passiert wird, erhalten so einen gültigen Logikpegel. Schließlich wird die reflektierte Welle von dem niederohmigen Ausgangstreiber absorbiert. Man nennt dieses Prinzip *Reflected-Wave Switching*.

Jedes PCI-Gerät arbeitet synchron zum zentralen Bustakt. Die Taktleitung ist im Gegensatz zu den Datenleitungen terminiert. Mit der steigenden Flanke werden die Busleitungen abgetastet. Damit dieses Zusammenspiel überhaupt funktionieren kann, macht die Spezifikation sehr genaue Vorgaben hinsichtlich des Timings und der physischen Beschaffenheit des Busses.

3.3 Aufbau eines PCI-Systems

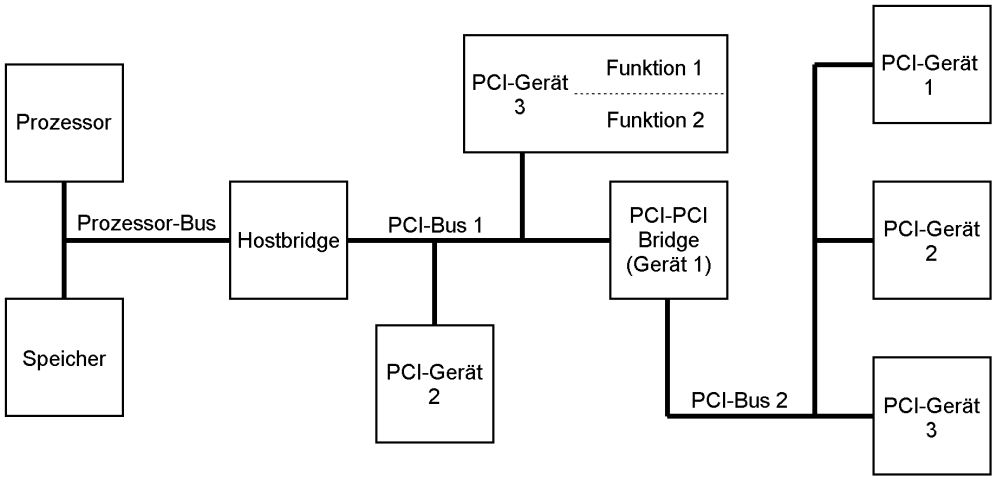

Abbildung 3.1: Beispielaufbau eines PCI-Systems

Jedes PCI-Gerät wird in der Konfigurationsphase eindeutig durch das Tripel (Bus, Device, Function) hierarchisch unterschieden. Die einzelnen Busse sind durch Bridges miteinander gekoppelt. Ein PCI-Device an einem PCI-Bus kann mehrere Funktionen in einem Chip beinhalten, die dann über die Function-ID unterschieden werden. Wurden alle Geräte einzeln initialisiert, d.h. Ressourcen wie Interrupts und Speicheradressräume durch das PCI-BIOS bzw. durch das Betriebssystem zugeteilt,

kann jedes Gerät von jedem Bus aus über die zugewiesenen Adressbereiche eindeutig angesprochen werden. Abbildung 3.3 zeigt einen typischen Aufbau.

Auf die Konfiguration eines PCI-Geräts und die zugrunde liegenden Mechanismen wird in Abschnitt 3.8 eingegangen.

3.4 Die wichtigsten PCI-Bus Signale

Die folgende Auflistung ist eine kurze nach Funktionsgruppen gegliederte Beschreibung der wichtigsten Signale des PCI-Busses, die auch das LogiCORE PCI Interface verwendet. Low-aktive Signale sind mit # gekennzeichnet.

3.4.1 Systemsignale

`CLK` ist der Bustakt.

`RST#` ist die Resetleitung für alle am Bus angeschlossenen Geräte.

3.4.2 Adress- und Datenpfad

`AD[31:0]` ist der gemultiplexte Adress- und Datenbus. Jede Transaktion beginnt mit einer Adressphase, der eine oder mehrere Datenphasen folgen.

`C/BE[3:0]#` ist der gemultiplexte Befehlsbus und Byte Enable Bus. In der Adressphase wird der Busbefehl auf den Bus gelegt, die Byte Enables während der Datenphasen. Mit den Byte Enable Signalen können einzelne Bytes des 32 Bit Datenbusses maskiert werden.

`PAR` ist die um einen Takt verschobene gerade Parität über `AD[31:0]` und `C/BE[3:0]#`.

3.4.3 Transfersteuerung

`FRAME#` Durch das Setzen von `FRAME#` startet der Initiator einen Transfer. Während der Transaktion bleibt das Signal gesetzt. Ein deaktiviertes `FRAME#`-Signal bedeutet, dass die Übertragung gerade beendet wird oder bereits beendet ist.

`DEVSEL#` (Device Selected) wird von einem Target gesetzt. Das Target zeigt damit an, dass es die in der Adressphase übermittelte Zieladresse erfolgreich decodiert hat, d.h. die Adresse in seinen Adressraum fällt.

IRDY# Mit IRDY# (Initiator Ready) signalisiert der momentane Busmaster (der Initiator der Transaktion), dass er bereit ist, die aktuelle Datenphase abzuschließen. Im Falle eines Lesetransfers bedeutet dies, dass er bereit ist, Daten vom Target zu übernehmen. Bei Schreibtransfers erklärt er so die von ihm auf den Datenbus gestellten Daten für gültig.

TRDY# (Target Ready) wird von dem adressierten Target bedient. Es signalisiert damit, dass es bereit ist, ein Datenwort zu übernehmen oder zu übergeben.

STOP# Ein Target setzt STOP#, um dem Initiator anzuzeigen, dass der laufende Transfer gestoppt werden soll.

IDSEL Mit IDSEL (Initialization Device Select) werden in der Konfigurationsphase die einzelnen PCI-Geräte selektiert. Für jedes PCI-Gerät und jeden PCI-Slot gibt es eine separate Leitung.

3.4.4 Fehlersignale

PERR# Ein PCI-Gerät, das Daten empfängt (Target bei Schreibtransfers oder Initiator bei Lesetransfers) setzt PERR#, wenn es einen Daten-Paritätsfehler festgestellt hat.

SERR# zeigt einen Adressen-Paritätsfehler, einen Daten-Paritätsfehler beim Busbefehl „Special Cycle“ oder einen allgemeinen Systemfehler an.

PERR# und SERR# weisen auf eine schwere Störung des Systems hin. Meist sind die Signale auf den nicht-maskierbaren Interrupteingang (NMI) des Prozessors gelegt.

3.4.5 Interrupts

INT[A:D]# sind die vier PCI-Interruptleitungen. Jede der Leitungen darf von mehreren PCI-Geräten getrieben werden (Shared Interrupts). Im PC werden die Leitungen durch einen Interruptrouter auf die bekannten IRQ0 – IRQ15 abgebildet.

3.4.6 Bus-Arbitrierung

REQ# Mit REQ# (Request) kann ein busmasterfähiges PCI-Gerät den Bus bei der zentralen Zugangssteuerung anfordern.

GNT# Die zentrale Bussteuerung teilt mit dem Signal GNT# (Grant) einem PCI-Gerät den zuvor angeforderten Bus zu.

Die REQ#/GNT#-Leitungspaare sind Punkt-zu-Punkt-Verbindungen zwischen der zentralen Bussteuerung und jedem Master. Man spricht in diesem Fall auch von *versteckter Busarbitrierung*, da sie über separate Steuerleitungen abgewickelt wird und somit stattfinden kann, während noch ein anderer Busmaster eine Transaktion durchführt.

3.5 Adressräume im PCI-Bus

Im PCI-Bus sind drei voneinander getrennte Adressräume definiert.

Im *Konfigurationsadressraum* wird die initialisierende Konfiguration und die Verwaltung des PCI-Geräts im Betrieb abgewickelt. Jedes PCI-Gerät hat dafür einen Konfigurationsspeicher.

Jedes PCI-Gerät kann sich Bereiche im *Speicheradressraum* und im *I/O-Adressraum* in der benötigten Größe reservieren. Die Anfangsadresse wird in der Konfigurationsphase vom PCI-BIOS oder dem Betriebssystem zugeteilt.

3.6 PCI-Busbefehle

Ein Initiator muss in der Adressphase eines Transfers außer der Zieladresse angeben, welche Art von Transaktion er durchführen möchte. Dafür sind während der Adressphase die PCI-Busleitungen C/BE[3:0]# vorgesehen. Die Angabe der Richtung des Datentransfers und des Adressraums, auf den sich die Zieladresse bezieht (Konfigurationsspeicher, I/O-Ports oder der „normale“ Speicheradressraum), ist unerlässlich. Außerdem gibt es noch erweiterte Modi für den Speicherzugriff und spezielle Buszyklen.

Interrupt Acknowledge (0000) Erkennt ein (Intel x86-basierter) Prozessor einen Interrupt an seinem Interrupteingang, reagiert er darauf, indem er den Interruptvektor vom Interruptcontroller abfragt. Auf dem PCI-Bus wird dazu ein spezieller *Interrupt Acknowledge* Buszyklus generiert.

Special Cycle (0001) ist dafür vorgesehen, bestimmte Nachrichten (Broadcast Messages) auf den Bus abzugeben.

I/O Read (0010) ist der lesende Zugriff auf den Speicher, der in den I/O-Adressraum eingeblendet ist. Im PC sind dies die I/O-Ports.

I/O Write (0011) ist der schreibende Zugriff auf den Speicher, der in den I/O-Adressraum eingeblendet ist.

Memory Read (0110) liest aus dem Speicher, der im Speicheradressraum liegt.

Memory Write (0111) schreibt in den Speicher, der im Speicheradressraum liegt.

Configuration Read (1010) ermöglicht das Lesen aus dem Konfigurationsspeicher eines PCI-Geräts.

Configuration Write (1011) ermöglicht das Schreiben in den Konfigurationsspeicher eines PCI-Geräts.

Memory Read Multiple (1100) ist ein spezieller Lesebefehl für einen optimierten Speicherzugriff: Der Initiator möchte Daten aus dem Speicher im Umfang mehrerer Cachezeilen lesen. Wenn das Target (im allgemeinen die Hauptspeichersteuerung) diesen Befehl unterstützt, wird es damit angewiesen, Speicherzeilen schon im voraus aus dem Speicher zu holen und für den Transfer bereitzuhalten.

Dual Address Cycle (1101) In einem System mit 64 Bit Adressen benötigt ein 32 Bit Initiator zwei Adressphasen auf dem 32 Bit PCI-Bus, um die 64 Bit Zieladresse zu übertragen. Im ersten Schritt werden die unteren 32 Bit der Adresse zusammen mit dem *Dual Address Cycle* Kommando geschickt, in der zweiten Phase dann die oberen 32 Bit mit dem eigentlichen Busbefehl.

Memory Read Line (1110) ist ein spezieller Lesebefehl für einen optimierten Speicherzugriff: Eine ganze Cachezeile soll aus dem Speicher gelesen werden.

Memory Write and Invalidate (1111) ist ein spezieller Schreibbefehl für einen optimierten Speicherzugriff: Der Initiator möchte Daten vom Umfang einer ganzen Cachezeile in den Speicher schreiben. Nun kann es sein, dass gerade diese Zeile im Cache oder im Write-Back Cache der PCI-Bridge vorhanden und als dirty markiert ist, also noch in den Speicher zurückgeschrieben werden müsste. Da der Initiator aber sowieso die gesamte Zeile im Speicher mit seinen Daten überschreiben wird, kann das Rückschreiben aus dem Cache in den Speicher vor dem PCI-Transfer des Initiators eingespart werden. Es genügt, die betreffende Zeile im Cache als ungültig zu markieren.

Für die im Rahmen dieser Arbeit beschriebenen Hardwareentwürfe, die auf dem LogiCORE PCI Interface aufbauen, sind die PCI-Busbefehle *I/O Read* und *I/O Write* bzw. *Memory Read* und *Memory Write* wichtig. Zugriffe auf den Konfigurationsspeicher bearbeitet das LogiCORE PCI Interface selbständig.

3.7 Transferprotokoll

Basis der Ablaufsteuerung eines Tranfers sind die Signale `FRAME#`, `IRDY#` (Initiator Ready), `TRDY#` (Target Ready), `DEVSEL#` (Device Selected) und `STOP#`.

Der Initiator zeigt mit FRAME# den Beginn der Übertragung an. Dabei legt er die Zieladresse und den Busbefehl auf den Bus. Wenn eines der PCI-Geräte erkennt, dass die Adresse in seinem Adressraum liegt, setzt es DEVSEL#. Damit endet die Adressphase, und es folgen eine oder mehrere Datenphasen bei aktiviertem IRDY# und TRDY#. Kurze Pausen in der Übertragung können bei Bedarf vom Initiator durch Deaktivieren von IRDY# bzw. vom Target durch Deaktivieren von TRDY# eingefügt werden.

Um das Ende oder den Abbruch der Übertragung einzuleiten, deaktiviert der Initiator das Signal FRAME#. Der Transfer ist jedoch erst dann beendet, wenn auch IRDY# und TRDY# deaktiviert worden sind.

Das Target kann seinerseits mit STOP# einen Transferabbruch erzwingen. Wenn beispielsweise der Initiator eine (Burst-) Transaktion mit einem nicht-burstfähigen Target beginnt, setzt das Target sofort das STOP#-Signal, um den Transfer nach der ersten Datenphase zu beenden.

Der Zustand der übrigen Steuerleitungen entscheidet nach einem Target-Transferstopp darüber, ob der Initiator eine neue Transaktion einleitet, um die Übertragung fortzusetzen, oder ganz abbricht. Bei der Wiederaufnahme der Übertragung muss gegebenenfalls das letzte Datenwort des vorausgehenden Transfers nochmals übertragen werden.

3.7.1 Beispiel für einen Lesetransfer

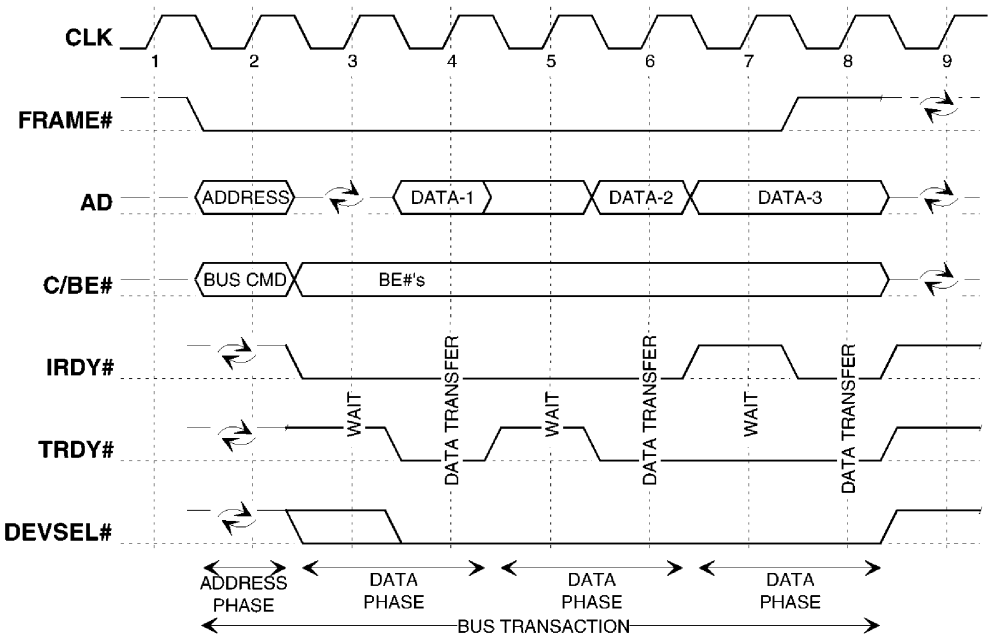

Abbildung 3.2: Beispiel für einen PCI-Lesetransfer

Abbildung 3.2 zeigt das Ablaufschema eines typischen Burst-Lesetransfers. Ein Initiator fordert Daten von einem Target an.

Charakterisch für Lesetransfers ist der sogenannte *Turn Around Zyklus* zwischen der Adressphase und der ersten Datenphase: In der Adressphase werden die Leitungen AD[31:0] von den Ausgangstreibern des Initiators getrieben, in den Datenphasen von den Ausgangstreibern des Targets. Um sicherzustellen, dass die Initiatortreiber abgeschaltet sind, bevor die Targettreiber den Datenbus übernehmen, wird ein zusätzlicher Takt eingefügt.

In der Datenphase wartet der Initiator auf Daten. Mit IRDY# zeigt er dem Target an, dass er bereit ist, in dem folgenden Takt Daten anzunehmen. Das Target signalisiert seinerseits mit TRDY#, dass die von ihm auf den Bus gelegten Daten gültig sind.

Um die Übertragung zu beenden, deaktiviert der Initiator das FRAME#-Signal. Im darauf folgenden Takt wird noch das letzte Datenwort übertragen und der Transfer beendet.

Bemerkung: In dem Beispiel in Abbildung 3.2 fügen Initiator und Target abwechselnd Wartezyklen ein. Dadurch verlängern sich die Datenphasen 1 und 3 um jeweils einen Takt. Ziel eines guten PCI-Hardwareentwurfs ist es, ihn so auszulegen, dass er ohne Wartezyklen auskommt.

3.7.2 Beispiel für einen Schreibtransfer

Abbildung 3.3 zeigt das Ablaufschema eines typischen Burst-Schreibtransfers. Ein Initiator überträgt Daten zu einem Target.

Wie beim Lesetransfer steuern IRDY# und TRDY# die Übertragung – allerdings mit vertauschten Rollen: Das Signal TRDY# zeigt an, dass das Target bereit ist, Daten zu übernehmen. Mit IRDY# kennzeichnet der Initiator ein gültiges Datenwort auf dem Datenbus.

3.8 Konfiguration eines PCI-Geräts

Jedes PCI-Gerät muss einen Konfigurationsspeicher implementieren, dessen erste 64 Bytes fest vorgegeben sind. Dieser Bereich bildet den *PCI Configuration Space Header* wie in Tabelle 3.1 dargestellt.

Einige der Register wie z.B. die Basisaddressregister oder das Befehlsregister dienen zugleich zur Information und zur Konfiguration. Mit einem Lesezugriff ermittelt die Konfigurationssoftware des BIOS oder des Betriebssystems den Ressourcenbedarf und die technischen Möglichkeiten des PCI-Geräts und schreibt dann die Ressourcenzuteilungen auf die gleiche Registeradresse.

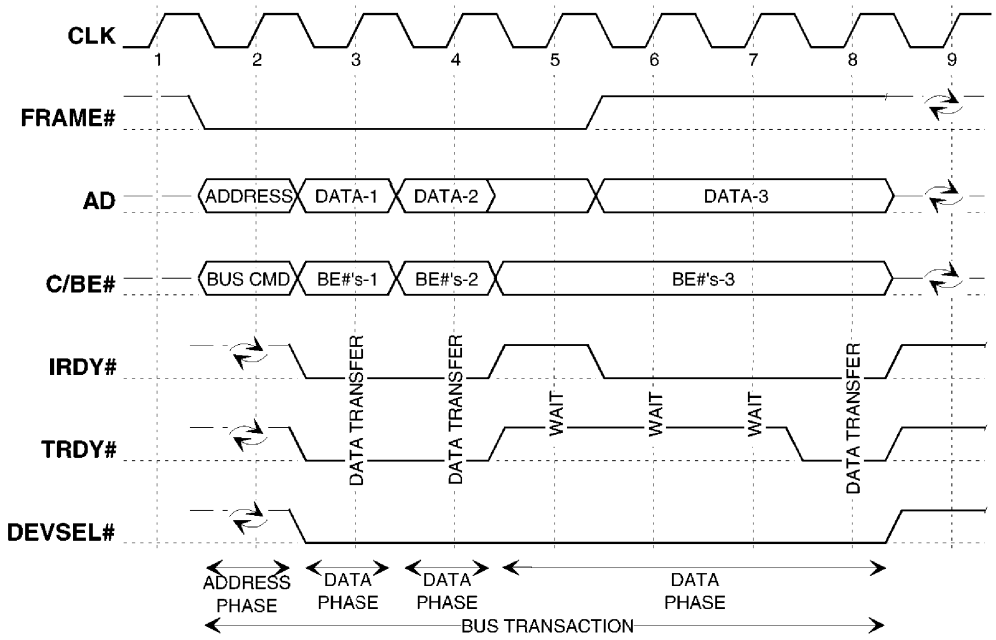

Abbildung 3.3: Beispiel für einen PCI-Schreibtransfer

Die *Vendor ID* ist die von der PCI SIG zugewiesene Herstellernummer. Die *Device ID* und die *Revision ID* werden dagegen vom Hardware-Hersteller vergeben und identifizieren den Typ und die Revision der PCI-Karte.

Der *Class Code* ist in drei 8 Bit große Felder unterteilt (Base Class, Sub Class, Programming Interface), die das PCI-Gerät nach einem von der PCI SIG vorgegebenen Schema gemäß seiner Funktion hierarchisch klassifizieren. Das Betriebssystem wird so bei der Identifikation der Karte unterstützt; im Idealfall kann es mithilfe dieser Angaben Standardtreiber bereitstellen und die PCI-Karte betreiben (z.B. VGA-kompatible Grafikkarten).

Die *Subsystem Vendor ID* wird ebenfalls von der PCI SIG vergeben, während die *Subsystem Device ID* vom Hardware-Hersteller zugeordnet wird. Eine komplexe PCI-Karte kann mehrere Subsysteme enthalten, die um das gleiche PCI-Interface gruppiert sind. Unter Zuhilfenahme der beiden Register kann das Betriebsystem trotz des gemeinsamen Buszugangs zwischen den Subsystemen unterscheiden und die richtigen Treiber laden. Der Wert 0 zeigt an, dass kein Subsystem vorhanden ist.

Das *Command Register* vermittelt eine Möglichkeit zur Steuerung der Einheit. Seine Bits geben an, ob und in welcher Weise das Gerät auf bestimmte Busbefehle reagieren soll. Ein Gerät lässt sich über das Register auch komplett deaktivieren; es reagiert dann nur noch auf Konfigurationszyklen.

Neben dem Befehlsregister ist auch ein *Status Register* vorhanden, das Angaben über den Gerätezustand für einen PCI-Vorgang enthält.

31 16	15 0	
Device ID	Vendor ID	0x00
Status Register	Command Register	0x04
Class Code	Revision ID	0x08
BIST / Header Type	Latency Timer / Cache Line Size	0x0C
Base Address Register 0		0x10
Base Address Register 1		0x14
Base Address Register 2		0x18
Base Address Register 3		0x1C
Base Address Register 4		0x20
Base Address Register 5		0x24
CardBus CIS Pointer		0x28
Subsystem ID	Subsystem Vendor ID	0x2C
Expansion ROM Base Address		0x30
reserved	Capabilities	0x34
reserved		0x38
Max_Lat / Min_Gnt	Interrupt Pin / Interrupt Line	0x3C

Tabelle 3.1: Der PCI Configuration Space Header

In das Register *Cache Line Size* wird die Länge einer Cachezeile des Systems in Einheiten zu 32 Bytes eingetragen. PCI-Geräte, die den Busbefehl „Memory Write and Invalidate“ unterstützen, müssen dieses Register implementieren.

Das Register *BIST* (Build-In Self-Test) gibt an, ob die Einheit einen Selbsttest ausführen kann. Durch einen passenden Schreibzugriff auf dieses Register kann er gestartet werden.

Das *Header Type Register* definiert das Format des PCI Configuration Space Headers. PCI-to-PCI Bridges und CardBus Bridges haben ein anderes Headerformat als die „normalen“ PCI-Geräte.

Das optionale *CardBus CIS Pointer Register* wird von Geräten implementiert, die zwischen CardBus und PCI vermitteln. Sein Inhalt zeigt auf die CardBus Informationsstruktur (CIS), die sich dann an anderer Stelle im Konfigurationsspeicher befindet.

Das Register *Expansion ROM Base Address* wird benötigt, um den Zugriff auf ein eventuell vorhandenes ROM im PCI-Gerät zu ermöglichen. Das ROM kann damit in den Speicheradressraum eingeblendet werden.

Der *Capabilities List Pointer* wird von komplexen PCI-Geräten benutzt, bei denen mehr Angaben über die Hardware nötig sind, als der verbindliche Teil des Konfi-

gurationsspeichers vorsieht (z.B. PCI Power Management Interface, AGP, Message Signaled Interrupts). Der Zeiger verweist auf den Beginn einer Datenstruktur, die hinter dem PCI Configuration Space Header liegt. Diese Datenstrukturen haben je nach Art der Hardware einen festgelegten Aufbau.

3.8.1 Die Basisadressregister

Die sechs *Base Address Register* werden zur Anforderung und Zuweisung von I/O-Adressräumen (I/O-Ports im PC) oder Speicheradressräumen (Memory Mapped I/O) verwendet, in die die Karte eingeblendet wird: Die Konfigurationssoftware schreibt dazu zunächst in alle Bits eine 1. Das PCI-Gerät hat jedoch nur die höherwertigen Bits implementiert. Beim erneuten Auslesen werden die nicht implementierten niederwertigen Bits als 0 zurückgegeben. Die Konfigurationssoftware erkennt daran, welchen Umfang der benötigte Adressraum haben muss und schreibt dann eine passende Anfangsadresse, deren untere Bits 0 sind, in das Basisadressregister. Die untersten vier Bits der Basisadressregister beim Memory Space bzw. die untersten zwei Bits beim I/O Space sind von diesem Mechanismus ausgenommen. Sie enthalten Angaben über den Speichertyp und Speichereigenschaften.

I/O Space oder Memory Space

Moderne PCI-Geräte verwenden in der Regel ausschließlich den Memory Space des Rechners. Zum einen ist der Adressraum der I/O-Ports in PCs beschränkt und bei den x86-Architekturen wohl nur noch wegen der Abwärtskompatibilität vorhanden. Andere moderne Architekturen, insbesondere die „echten“ RISC-Prozessoren, haben nur einen Adressraum, in den auch die periphere Hardware eingeblendet wird [1]. Zum anderen verbietet die Spezifikation – obwohl es möglich wäre – mehr als 256 zusammenhängende Bytes im I/O Space pro Basisadressregister zu belegen.

Die daraus resultierenden kleinstmöglichen und größtmöglichen Adressräume gibt die Tabelle 3.2 an.

Speicherraum	**Minimum**	**Maximum**
I/O Space	4 Bytes (1 DWord)	256 Bytes (64 DWords)
Memory Space	16 Bytes (4 DWords)	2 GBytes (512 K DWords)

Tabelle 3.2: Größenbeschränkungen der Adressräume

Die PCI-Spezifikation empfiehlt, im Falle des Speicheradressraums mindestens einen 4 KByte großen Bereich zu belegen, um die Anzahl der Bitvergleiche im Adressdecoder des PCI-Interfaces in einem vernünftigen Rahmen zu halten.

Das Prefetch-Attribut (nur im Memory Space)

Das gesetzte Prefetch-Attribut erlaubt der PCI-Bridge vorauseilende spekulative Lesezugriffe (Read Prefetching) und das Zusammenfassen bestimmter Schreibzugriffe (Byte Merging). Ein im PCI-Gerät realisierter Speicherbereich darf mit dem Prefetch-Attribut markiert werden, wenn die Bedingungen eines „wellbehaved memory“ erfüllt sind:

1. Bei einem Lesezugriff dürfen keine Seiteneffekte auftreten, d.h. ein Lesezugriff ändert nicht den Inhalt des Speichers oder den Zustand des PCI-Geräts.

   Beispiel: Das Lesen aus einem RAM hat keine Seiteneffekte, während das Lesen aus einem FIFO-Speicher den Zustand des FIFO-Speichers ändert: Die gelesenen Daten sind anschließend nicht mehr im FIFO. Automatisches spekulatives Lesen zerstört unvorhersehbar Daten.

2. Bei einem Lesezugriff sind immer alle Bytes auf voller Busbreite (32 Bit bzw. 64 Bit) gültig. Bei PCI gibt es grundsätzlich die Möglichkeit, einzelne Bytes des Datenworts zu maskieren.

3. Das PCI-Gerät arbeitet auch dann korrekt, wenn die PCI-Bridge einzelne Schreibzugriffe zusammenfasst (Byte Merging).

   Beispiel: Der Prozessor führt zwei 16 Bit Schreibzugriffe aus, den ersten auf die Adressen 0x00000100 und 0x00000101, den zweiten auf die Adressen 0x00000102 und 0x00000103, wobei ein 16 Bit Zugriff ein 32 Bit Zugriff mit zwei maskierten Bytes ist. Da die Adressen aufeinanderfolgen, kann die PCI-Bridge die beiden 16 Bit Schreibzugriffe zu einem 32 Bit Schreibzugriff zusammenfassen.

Position im Gesamtadressraum und Registerbreite (nur im Memory Space)

- Typ 00: 32 Bit Register. Die Anfangsadresse liegt in den unteren 4 GByte (2^{32} Bytes) des gesamten Speicheradressraums und ist an 32 Bit ausgerichtet, d.h. durch 4 teilbar. Die unteren zwei Bits der 32 Bit Adresse sind 0 bzw. werden ignoriert.

- Typ 10: 64 Bit Register. Die zugewiesene Startadresse liegt in einem 2^{64} Bytes umfassenden Speicheradressraum. Voraussetzung ist, dass das PCI-Gerät die 64 Bit Adressierung unterstützt. Das Basisadressregister braucht dementsprechend auch zwei Doppelwörter usw. (in dieser Arbeit nicht weiter relevant).

- Typ 01 ist seit der PCI-Spezifikation 2.2 nicht mehr vorgesehen. Der Speichertyp erzwingt, dass die Anfangsadresse in den ersten 1 MByte des Speicheradressraums liegt.

3.8.2 Max_Lat, Min_Gnt und der Latency Timer

Alle drei Register sind optional und nur für Busmaster relevant.

Das Nur-Lese-Register *Max_Lat* gibt in 250 ns Zeitschritten an, *wie oft* das PCI-Gerät im Mittel den Bus benötigt. Eine intelligente programmierbare Busarbitrierung kann mit dieser Angabe sinnvolle Prioritäten bei konkurrierenden Busanforderungen mehrerer PCI-Geräte setzen. Der Wert 0 zeigt an, dass das Gerät keine besonderen Anforderungen stellt.

Das Nur-Lese-Register *Min_Gnt* kann zur Optimierung von Bursttransfers verwendet werden. Es gibt in 250 ns Schritten an, *für wie lange* der Busmaster den Bus im Mittel benötigt, um performant arbeiten zu können. Der Wert 0 zeigt an, dass das Gerät keine besonderen Anforderungen stellt. Auf der Basis des Wertes im Min_Gnt-Register kann die Konfigurationssoftware im Wissen um die tatsächliche Bustaktfrequenz einen Wert für das *Latency Timer Register* berechnen und dort hineinschreiben, sofern es überhaupt im betreffenden Gerät implementiert ist. Der Latency Timer (LT) gibt die minimale Zeit in Bustakten an, für die der PCI-Master den Bus für Bursttransfers benutzen darf.

Der Ablauf stellt sich vereinfacht so dar: Der Initiator fordert den Bus für einen Bursttransfer an (`REQ#`-Leitung). Die Bussteuerung teilt den Bus zu (`GNT#`-Leitung), wenn er frei ist und kein anderes wartendes Gerät Vorrang hat. Der Initiator hat den Bus nun für mindestens LT PCI-Taktzyklen garantiert für sich reserviert. Werden ihm im laufenden Transfer die Busrechte entzogen, braucht er nicht sofort zu unterbrechen und den Bus wieder freizugeben, sondern darf noch so lange fortfahren, bis LT Zyklen erreicht sind. Wenn kein anderes Gerät den Bus anfordert, darf er einen beliebig langen Bursttransfer durchführen.

3.8.3 Interrupt Pin und Interrupt Line

Der PCI-Bus verfügt über vier Interruptleitungen `INTA#` bis `INTD#`, die von mehreren Geräten getrieben werden dürfen (Shared Interrupts). Das Nur-Lese-Register *Interrupt Pin* gibt an, ob und welche bzw. wieviele der vier Leitungen für die Signalisierung von Interrupts verwendet werden.

Im *Interrupt Line Register* trägt die Konfigurationssoftware die Nummer des Eingangs des Interruptcontrollers ein, dem die PCI-Interruptleitung zugeordnet wurde.

Beispiel: Für den Prozessor eines typischen PCs sind die vier PCI-Interruptleitungen unsichtbar. Sie werden mit einem Interruptrouter zusammengefasst und auf die Eingänge des Interruptcontrollers abgebildet. Der so zugeordnete IRQ, d.h. ein Wert zwischen 0x00 und 0x0F, wird im Interrupt Line Register eingetragen. Durch Auslesen des Interrupt Line Registers stellt der Gerätetreiber fest, für welchen Interrupt er seine Interrupt Service Routine anmelden muss.

4 Das LogiCORE PCI Interface

Das LogiCORE PCI Interface [12] ist ein vorimplementiertes und getestetes Modul für Xilinx Spartan- und Virtex-FPGAs, das die PCI-Spezifikation 2.3 voll erfüllt. Die Pinbelegung für jedes FPGA und die relative Anordnung der internen Logik ist vordefiniert. Kritische Pfade werden durch Constraint-Dateien überwacht, die so ein vorhersagbares Zeitverhalten sicherstellen.

Das PCI-Interface unterstützt mit den durch das verwendete FPGA bedingten Beschränkungen

- 32 Bit und 64 Bit Busbreite
- 33 MHz und 66 MHz PCI-Takt
- 3,3 V und 5 V Logik

Grundlage aller Hardwareentwürfe, die auf dem LogiCORE PCI Interface aufbauen, ist der *LogiCORE PCI Design Guide* [13]. Im Rahmen dieser Arbeit entstand außerdem ein Tutorial [6], das mit Hintergrundinformationen und Schritt-für-Schritt-Anleitungen die Konfiguration und die Verwendung des Moduls anhand typischer Beispielapplikationen beschreibt.

Die verwendete Entwicklungsumgebung ist die Xilinx ISE v5.1i.

4.1 Funktionale Beschreibung

Wie die Abbildung 4.1 zeigt, ist das LogiCORE PCI Interface in fünf große Blöcke unterteilt.

PCI I/O Interface Block

Der PCI I/O Interface Block kümmert sich um die Anbindung an den PCI-Bus auf unterster Ebene. Dies schließt die gesamte Signalgebung, die Eingangs- und Ausgangssynchronisation, die Steuerung der Ausgangstreiber und die Logik zur Busarbitrierung für Initiatordesigns ein.

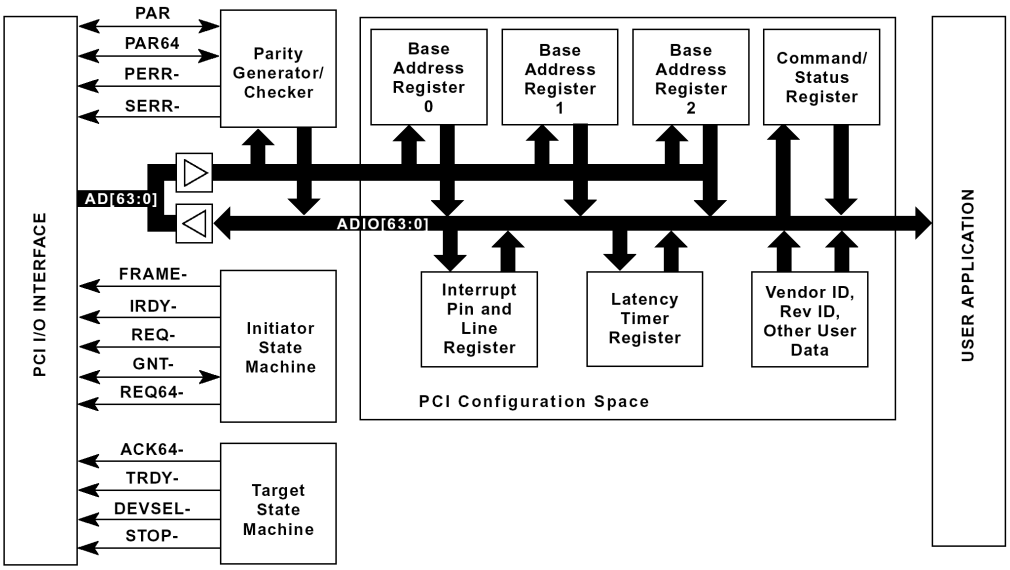

Abbildung 4.1: Blockdiagramm des LogiCORE PCI Interface

PCI Configuration Space

Dieser Block enthält die 64 verbindlichen Bytes des Konfigurationsspeichers des PCI-Geräts (PCI Configuration Space Header). Die in der Konfigurationsphase des Systems notwendigen Konfigurationstransfers wickelt das Interface selbständig ab.

Parity Generator/Checker

Dieser Block erzeugt und prüft die über den AD-Bus und den C/BE#-Bus gebildete Parität und setzt bei Datenfehlern das PERR#-Signal, bei Adressfehlern das SERR#-Signal.

Initiator State Machine

Dieser Block ist für die Initiatorfunktionen des PCI-Interfaces verantwortlich. Bei reinen Targetentwürfen bleibt er ungenutzt.

Unterstützt werden die folgenden Busbefehle (siehe Abschnitt 3.6): Configuration Read, Configuration Write, Memory Read, Memory Write, Memory Read Multiple, Memory Read Line, Interrupt Acknowledge, Special Cycle, I/O Read, I/O Write.

Target State Machine

Dieser Block enthält die Logik für die Targetfunkionalität des PCI-Interfaces. Merkmale sind:

- drei Basisadressregister zur Unterstützung von bis zu drei Adressbereichen, wahlweise im I/O- oder Speicheradressraum, im Umfang von 16 Bytes bis zu 2 GBytes
- Adressdecoder mittlerer Geschwindigkeit, d.h. DEVSEL# wird im zweiten auf die Adressübernahme folgenden Takt gesetzt
- die unterstützten Busbefehle Configuration Read, Configuration Write, Memory Read, Memory Write, Memory Read Multiple, Memory Read Line, Interrupt Acknowledge, I/O Read, I/O Write
- Logik zur Beendigung eines Transfers (Target Abort, Target Retry, Target Disconnect)

4.2 Schnittstelle zum eigenen VHDL-Entwurf

Abbildung 4.2 zeigt das LogiCORE PCI Interface (Mitte) als VHDL-Modul. Auf der linken Seite befinden sich die Anschlüsse zum PCI-Bus und auf der rechten Seite die nach Funktionsgruppen geordneten Signale, die im eigenen Hardwareentwurf verwendet werden können. Eine tabellarische Beschreibung der einzelnen Signale befindet sich zusammen mit generellen Hinweisen zum Entwurf eigener Applikationen im LogiCORE PCI Design Guide [13] sowie in den Beispielentwürfen des Tutorials [6].

4.3 Signal Pipelining

Um die strengen Zeitvorgaben der PCI-Spezifikation einzuhalten, leitet das LogiCORE PCI Interface alle Steuersignale des PCI-Busses und den Datenpfad über Flipflops bzw. Register. Abbildung 4.3 zeigt, wie dies generell geschieht.

Wenn das PCI-Interface ein Signal vom PCI-Bus empfängt (Fall 1), wird dieses von einem Eingangsregister erfasst. Das Signal ist einen Takt, nachdem es auf dem PCI-Bus erschien, verfügbar.

Bei Ausgangssignalen (Fall 2) sind zwei Fälle zu unterscheiden: Handelt es sich um ein kombinatorisches Signal, muss es einen Takt, bevor es auf dem PCI-Bus erscheinen soll, an das PCI-Interface abgegeben werden. Die meisten Ausgangssignale haben jedoch vorgeschaltete Ausgangsregister. Solche Signale müssen daher schon zwei Takte, bevor sie sich auf dem PCI-Bus auswirken sollen, dem PCI-Interface übergeben werden.

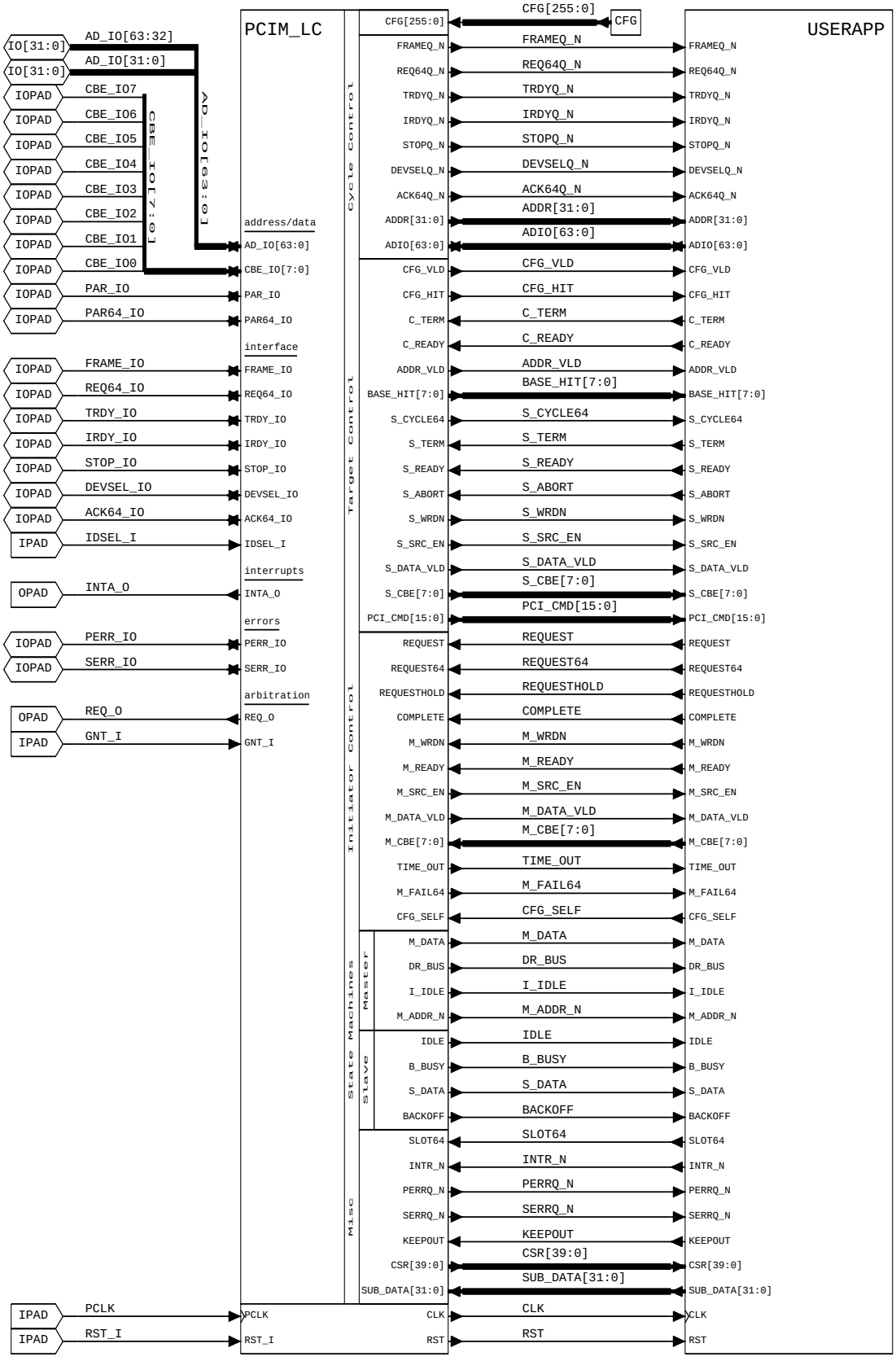

Abbildung 4.2: Das LogiCORE PCI Interface als VHDL-Modul

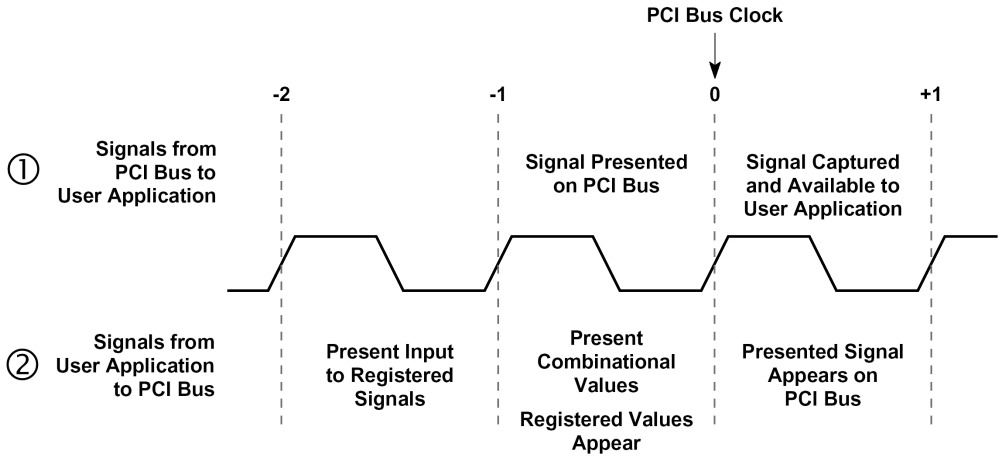

Abbildung 4.3: Verzögerungen durch Pipelining

4.4 Download des LogiCORE PCI Interface

Das LogiCore PCI Interface ist über einen durch ein Kennwort geschützten Link[1] auf der Xilinx-Website zu erreichen. Dem Download ist ein Java-Applet vorgeschaltet, das den PCI Configuration Space Header grafisch darstellt. Dort kann das PCI-Interface vorab konfiguriert werden. Die vorgenommenen Einstellungen finden ihren Niederschlag in einer Konfigurationsdatei; das eigentliche Interface-Modul wird nicht verändert.

Im praktischen Umgang mit dem PCI-Interface zeigte es sich, dass es einfacher ist, die Einstellungen direkt per Hand in der Konfigurationsdatei vorzunehmen anstatt den Download-Vorgang für jedes Entwurfsprojekt erneut zu durchlaufen.

In einem zweiten Schritt muss die für das Entwicklungsboard passende Constraint-Datei xc2s200fg456_32_33.ucf erzeugt werden. Sie gibt später in der Implementierungsphase die Pinbelegungen vor, definiert die relative Anordnung der Logik im FPGA und kontrolliert das Timing der kritischen Pfade. Xilinx stellt dafür den PCI UCF Generator zur Verfügung, der ebenfalls über die oben angegebene Webseite zu erreichen ist. Die dort vorzunehmenden Einstellungen sind in [6] beschrieben.

4.5 Zusammenstellen eines neuen Entwurfsprojekts

Aus dem heruntergeladenen und entpackten Gesamtpaket, das auch die Dokumentation und Dateien für die Simulation enthält, werden die für die Synthese und die Implementierung benötigten sieben Dateien in den zukünftigen Projektordner kopiert.

[1] http://www.xilinx.com/pci/

1. */vhdl/src/xpci/pci_lc_i.ngo*
   ist die Netzliste des LogiCORE PCI Interface in einem Xilinx-spezifischen Binärformat.
2. */vhdl/src/xpci/pci_lc_i.vhd*
   ist der Low-Level Wrapper für das PCI-Interface.
3. */vhdl/src/wrap/pcim_lc_33_5_s.vhd*
   ist die Wrapper-Datei für das PCI-Interface. Im Ordner /src/wrap/ stehen mehrere Dateien für unterschiedliche PCI-Busse zur Auswahl, in diesem Fall 33 MHz bei 5 V.
4. */vhdl/src/xpci/cfg.vhd*
   ist das Konfigurationsmodul.
5. */vhdl/src/xpci/pcim_top.vhd*
   stellt die Top-Level Datei des VHDL-Entwurfs dar.
6. */vhdl/src/xpci/userapp.vhd*
   wird später die selbstgeschriebene Applikation aufnehmen.
7. *xc2s200fg456_32_33.ucf*
   ist die mit dem UCF Generator erstellte Constraint-Datei.

Im Project Navigator der Xilinx ISE wird nun ein neues Projekt in dem Projektordner angelegt und die kopierten VHDL-Dateien (*.vhd) als VHDL-Module dem Projekt hinzugefügt. Die Constraint-Datei wird der Top-Level Komponente pcim_top zugeordnet. Abbildung 4.4 zeigt, wie sich die Module hierarchisch im Dateifenster (Module View) des Project Navigators anordnen.

Wie in [6] ausführlich beschrieben, müssen einige Optionen für die Synthese und Implementation richtig gesetzt werden. Die beiden wichtigsten Einstellungen für die Synthese sind die Aktivierung der Option *Keep Hierarchy* und das Ausschalten der Anweisung *Read Cores*. Dadurch wird die von der Constraint-Datei für die Implementationsphase vorausgesetzte VHDL-Modulhierarchie in der Synthesephase beibehalten und verhindert, dass das PCI-Interface nochmals optimiert wird.

Diese Optionen erreicht man nur, wenn man im Project Navigator den *Property Display Level* auf *Advanced* setzt.

Bemerkung 1: Es bietet sich an, von dem einmal erstellten Grundgerüst Kopien für spätere Entwürfe anzulegen.

Bemerkung 2: Der LogiCORE PCI Implementation Guide [14] erläutert die praktische Vorgehensweise in den einzelnen Phasen eines Entwurfs (Simulation, Synthese, Implementation, Timinganalyse) für verschiedene Entwurfswerkzeuge.

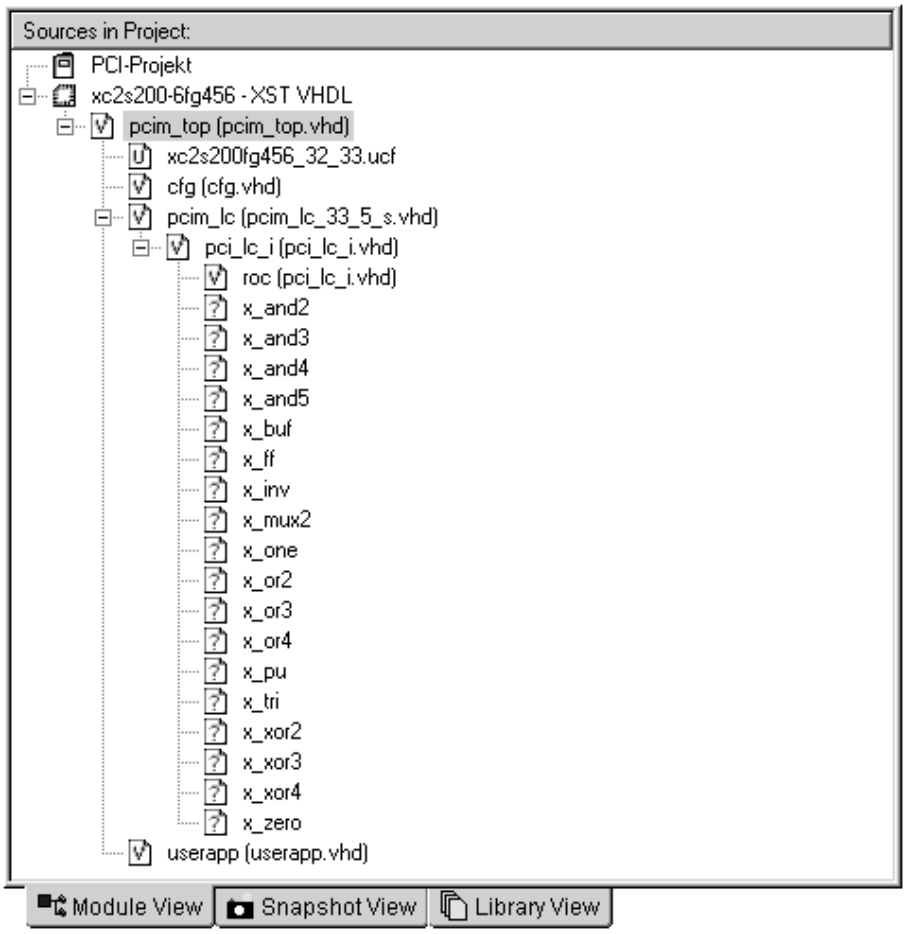

Abbildung 4.4: Die Modulhierarchie des LogiCORE PCI Interface

4.6 Konfiguration des LogiCORE PCI Interface

Die Komponente cfg (Datei cfg.vhd) implementiert einen mit konstanten Werten belegten Bus, über den das PCI-Interface durch symbolische Konstanten konfiguriert wird.

```
entity cfg is port (
  CFG : out std_logic_vector(255 downto 0)
);
end cfg;
```

Die meisten Einstellungen hinsichtlich Funktionsumfang, Verhalten und Ressourcenbedarf des PCI-Geräts entsprechen unmittelbar den Registern im PCI Configuration Space Header, der deshalb sehr ausführlich in Abschnitt 3.8 erläutert wurde. Außerdem enthält [6] eine ausführliche Besprechung der einzelnen Optionen.

5 Tutorial zum Entwurf von PCI-Interfaces

Das innerhalb dieser Arbeit erarbeitete Wissen wurde fortlaufend in Form von HTML-Dokumenten festgehalten. So entstand ein Tutorial [6], das Grundlagen zum PCI-Bus und zum LogiCORE PCI Interface erklärt, praktische Tipps gibt und typische Beispielentwürfe vorstellt, die als Ausgangspunkt eigener Anwendungen verwendet werden können. Zu jedem Hardwareentwurf werden ausführlich erläuterte Gerätetreiber und Testprogramme für Linux geliefert.

Das Tutorial gliedert sich in folgende Kapitel:

1. Vorbereitungen

Dieses Kapitel beschreibt, wie man das LogiCORE PCI Interface von der Xilinx-Website herunterlädt und ein Projekt-Grundgerüst im Project Navigator der Entwicklungsumgebung Xilinx ISE 5.1i erstellt.

2. Der PCI Configuration Header

Dieser Abschnitt stellt den PCI Configuration Space Header aus der Sicht des LogiCORE PCI Interface sowie einige technische Besonderheiten und Mechanismen des PCI-Busses vor.

3. Adapterplatine für den Agilent 1671G Logic Analyzer

Für komplexe Messungen und Analysen im laufenden Betrieb wurde eine Adapterplatine entworfen, mit der der Agilent 1671G Logic Analyzer des Instituts für Informatik an das Entwicklungsboard angeschlossen werden kann. Insgesamt werden drei 16 Bit Datenbusse zur Verfügung gestellt. Die Adapterplatine kann auch dazu verwendet werden, um auf die I/O-Pins des FPGAs direkt zuzugreifen, z.B. in eigenen Versuchsschaltungen. Zwei zusätzliche Taster und vier Leuchtdioden erweitern die Möglichkeit zur direkten Interaktion mit dem eigenen Hardwareentwurf.

4. Beispielentwurf: Board-Test

Dieses Projekt ist ein einfaches, minimales Single Transfer PCI-Target, das die LED, die 7-Segment-Anzeigen und die DIP-Schalter des Entwicklungsboards ansteuert.

5. Beispielentwurf: PCI-Target mit 4 KByte RAM

Der hier vorgestellte Entwurf ist ein PCI-Target, das den Zugriff auf 4 KByte des Spartan-II internen statischen Block RAMs realisiert. Das erste Design ist ein Single Transfer PCI-Target, das in einem zweiten Schritt auf ein Burst Transfer PCI-Target erweitert wird.

6. Beispielentwurf: PCI-Interrupts

Dieses Beispiel demonstriert die Interruptfunktion auf der Hardwareseite und auf der Gerätetreiberseite. Drückt man den User-Taster auf dem Entwicklungsboard, wird ein Interrupt ausgelöst, den der Interrupthandler des zugehörigen Treibers mit der Ausgabe einer Meldung quittiert.

7. Beispielentwurf: Ein einfacher Single Transfer PCI-Initiator

Dieser Entwurf ist ein einfacher Single Transfer PCI-Initiator (PCI-Busmaster) und zugleich ein Beispiel für den Zugriff auf die I/O-Ports des PCs und Bytemaskierung. Sobald die Stellung der DIP-Schalter verändert wird, fordert die Karte selbständig den PCI-Bus an und sendet das durch die DIP-Schalter dargestellte Byte an den Daten-I/O-Port der parallelen Schnittstelle des Rechners. Dort kann es z.B. mit Leuchtdioden angezeigt werden.

8. Beispielentwurf: PCI-Busmaster mit Burstunterstützung und Interrupt

Dieses Projekt implementiert einen vollständigen PCI-Initiator mit Burstunterstützung für DMA. Das Gerät verfügt über einen 4 KByte großen Puffer, in den es die über den PCI-Bus angeforderten Daten abspeichern oder den Pufferinhalt über den PCI-Bus an ein anderes PCI-Gerät (z.B. den Hauptspeicher) senden kann. Nach Abschluss eines Transfers wird ein Interrupt ausgelöst, um den Gerätetreiber zu informieren.

6 Der Entwurf des PCI-Busmasters

Alle in diesem Kaptitel zitierten Dateien befinden sich auf der beiliegenden CD im Verzeichnis /projekte/pci-busmaster/.

In diesem Kapitel wird der Entwurf des PCI-Interfaces und dem vorgeschalteten FIFO, der das übrige System an den PCI-Busmaster anbindet, beschrieben. Abbildung 6.1 gibt einen Überblick über das Zusammenspiel der beiden Komponenten.

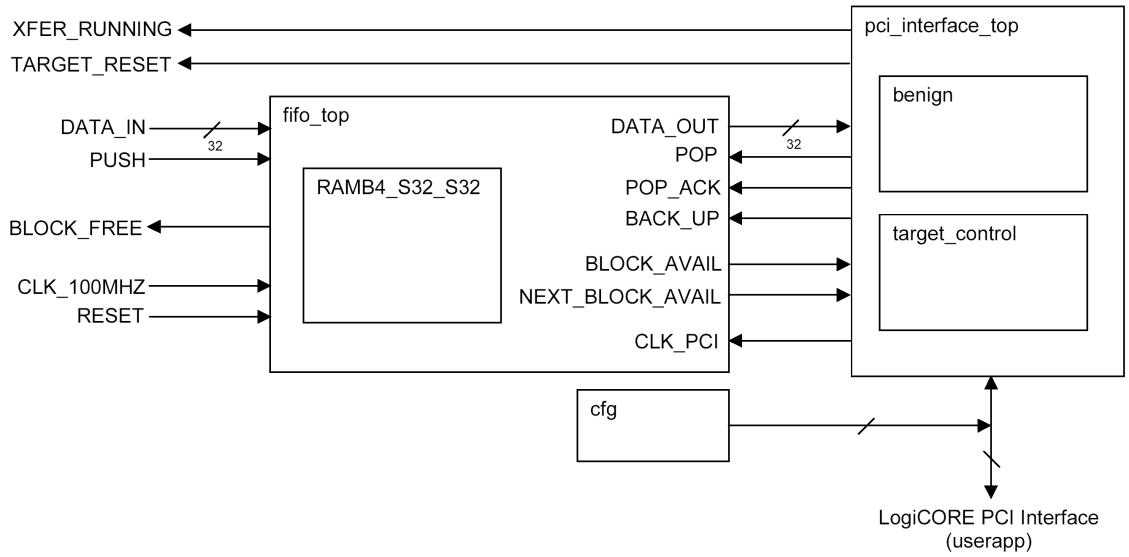

Abbildung 6.1: Der PCI-Busmaster mit vorgeschaltetem FIFO-Speicher

6.1 Der FIFO-Speicher

Grundlage des FIFOs sind zwei 256 × 16 Bit Dual Port Block RAMs des FPGAs (Primitiv RAMB4_S16_S16 aus der Xilinx Entwurfsbibliothek [11]), die durch „Parallelschaltung“ zum Modul RAMB4_S32_S32 zusammengefasst sind. Man erhält so einen Dual Port RAM mit 256 × 32 Bit Speicherplätzen, auf dem der FIFO als Ringpuffer implementiert ist. Alle Eingangssignale werden mit der steigenden Taktflanke des jeweiligen Ports abgetastet. Die beiden Ports sind vollständig voneinander unabhängig.

Der eine Port ist mit dem 100 MHz Systemtakt getaktet und dient als Dateneingang. Der Schreibzeiger `write_ptr`, ein zyklischer 8 Bit Adresszähler, zeigt auf die nächste freie Speicherstelle. Mit `PUSH` werden die an `DATA_IN[31:0]` anliegenden Daten in das RAM geschrieben und der Schreibzeiger erhöht.

Der andere Port ist mit dem 33 MHz PCI-Takt getaktet und dient als Datenausgang. Der Lesezeiger `read_req_ptr`, ebenfalls ein zyklischer 8 Bit Adresszähler, zeigt auf den als nächstes auszulesenden Speicherplatz. Mit `POP` wird der Inhalt der Speicherstelle in das Ausgangsregister des Block RAMs übernommen und zugleich der Lesezeiger erhöht, so dass in dem auf ein gesetztes `POP`-Signal folgenden Takt die gewünschten Daten an `DATA_OUT[31:0]` anliegen, während das Block RAM intern schon das nächste Datenwort für das Ausgangsregister bereitstellt. Intern wird dazu der EN-Eingang des Block RAM Moduls verwendet [10].

Der Speicher des FIFOs ist in 32 Blöcke mit acht Wörtern eingeteilt. Alle Angaben über den Füllstand des FIFOs beziehen sich immer auf freie oder belegte Datenblöcke. Diese Einteilung ist aus mehreren Gründen notwendig und sinnvoll:

- Das SDRAM und das PCI-Interface führen Bursttransfers mit dem FIFO durch. Ein freier bzw. belegter Datenblock bedeutet den garantierten Zugriff auf acht Speicherplätze („Planungssicherheit“).
- Da Schreibzeiger und Lesezeiger mit untereinander asynchronen unterschiedlichen Takten arbeiten, müssen für die Ermittlung des Füllstands des FIFOs die Stände beider Zähler synchronisiert werden. Die Berechnung orientiert sich dabei an dem langsamen PCI-Takt, so dass es aus der Sicht des schnellen Systemtakts mehrere Takte dauert, bis ein neuer Füllstandswert an den Statusausgängen des FIFOs sichtbar wird. Durch die Organisation des Speichers in Blöcken ändert sich der Füllstand immer nur dann, wenn einer der Zeiger in einen neuen Datenblock eintritt. Die FIFO-Steuerung hat nun mehrere Takte Zeit, um den veränderten Füllstand zu berechnen und an das SDRAM bzw. an das PCI-Interface weiterzugeben, während diese noch in ihren Datenblöcken schreiben bzw. lesen.

6.1.1 Ermittlung des Füllstands des FIFO-Speichers

Zur Berechnung des Füllstands des FIFOs wird die Differenz zwischen dem Lesezeiger und dem Schreibzeiger gebildet. Um dabei die Zustände „FIFO leer“ und „FIFO voll“ unterscheiden zu können, darf der Schreibzeiger den Lesezeiger nicht einholen. Der letzte Datenblock vor dem Datenblock, in dem sich der Lesezeiger befindet, ist daher für den Schreibprozess gesperrt.

Der Lesezeiger ändert sich mit der steigenden Flanke des 33 MHz PCI-Bustakts und der Schreibzeiger mit der steigenden Flanke des 100 MHz Systemtakts. Für die Füllstandsberechnung müssen beide Zeiger synchronisiert werden:

- Vor der Übernahme der Differenz in das Ergebnisregister müssen *beide* Zeiger ausreichend lange stabil gewesen sein, da die Berechnung der Differenz Zeit benötigt.
- Der Füllstand bzw. die aus dem Füllstand gewonnenen Signale `BLOCK_FREE`, `BLOCK_AVAIL` und `NEXT_BLOCK_AVAIL` werden sowohl von dem Modul, das Daten in den FIFO schreibt, als auch von dem Modul, das Daten aus dem FIFO liest, synchron zu ihren jeweiligen Takten (Systemtakt und PCI-Bustakt) ausgewertet. Deshalb müssen diese Signale im Bereich der steigenden Flanke *beider* Takte stabil sein.

Die Anforderungen werden mit zwei Prozessen realisiert.

Der erste Prozess tastet mit dem 100 MHz Systemtakt `CLK_100MHZ` den 33 MHz PCI-Bustakt `CLK_PCI` ab und bildet so einen mit dem Systemtakt synchronisierten PCI-Bustakt `clk_pci_sync`, der anschließend noch um eine Systemtaktperiode verzögert wird. Man erhält so das Taktsignal `clk_pci_delay`.

```
process (CLK_100MHZ)
begin
  if CLK_100MHZ'event and CLK_100MHZ = '1' then
    clk_pci_sync  <= CLK_PCI;
    clk_pci_delay <= clk_pci_sync;
  end if;
end process;
```

Abbildung 6.2 zeigt die Simulation dieses Ansatzes in den zwei Grenzsituationen. Im oberen Timingdiagramm liegt die steigende Flanke des PCI-Bustakts kurz vor der steigenden Flanke des Systemtakts, mit der dann auch die steigende Flanke des PCI-Bustakts erkannt wird (frühstmöglicher Abtastzeitpunkt). Im unteren Timingdiagramm liegt die steigende Flanke des PCI-Bustakts kurz hinter der steigenden Flanke des Systemtakts. Erst im folgenden Systemtakt wird die steigende Flanke des PCI-Bustakts erkannt (spätestmöglicher Zeitpunkt). In beiden Fällen ist das erzeugte Taktsignal `clk_pci_delay` mit dem Systemtakt und damit auch mit dem Schreibzeiger synchronisiert. Seine steigende Flanke liegt im Bereich der fallenden Flanke des PCI-Bustakts. Der Lesezeiger ist damit ausreichend lange vorher stabil. Das Ergebnis der Differenzbildung kann mit steigender Flanke des Systemtakts und des PCI-Bustakts sicher abgefragt werden.

Der zweite Prozess realisiert ein mit `clk_pci_delay` getaktetes Register, das die Differenz aus Lese- und Schreibzeiger speichert. Es interessiert dabei nur die Anzahl der belegten Datenblöcke. Deshalb werden die unteren drei Bits der Zeiger nicht mit einbezogen.

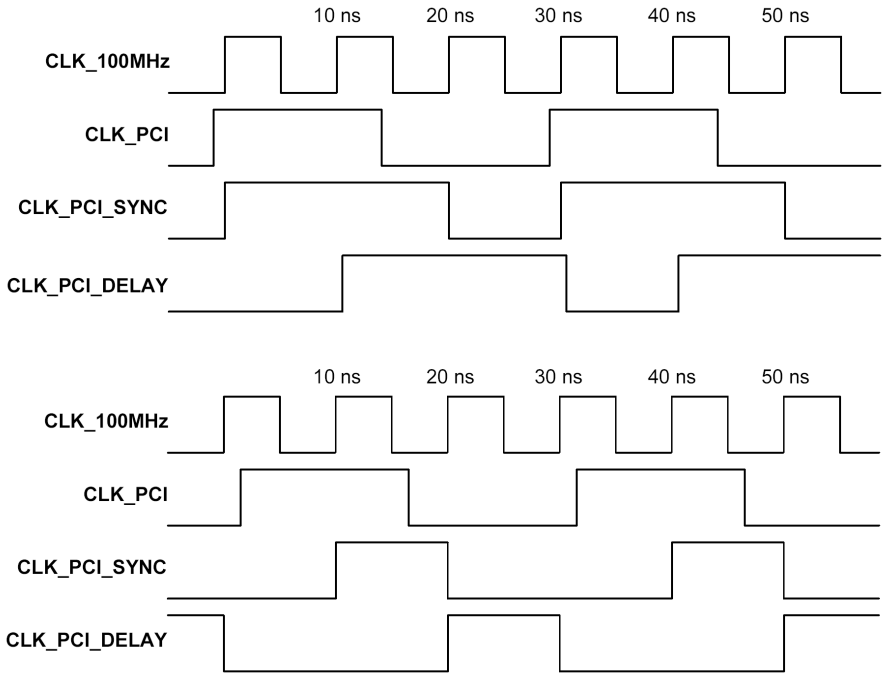

Abbildung 6.2: Simulation der Taktsynchronisation

```
process (clk_pci_delay)
begin
  if clk_pci_delay'event and clk_pci_delay = '1' then
    fill_level_reg <= write_ptr(7 downto 3) - read_ack_ptr(7 downto 3);
  end if;
end process;
```

Für das PCI-Interface werden aus dem Füllstand zwei Steuersignale gewonnen:

`BLOCK_AVAIL` zeigt an, dass mindestens ein vollständiger Datenblock im FIFO bereitsteht. Dies ist dann der Fall, wenn sich der Lesezeiger und der vorausgehende Schreibzeiger in unterschiedlichen Datenblöcken befinden und der Lesezeiger auf das erste Datenwort eines Datenblocks zeigt.

```
BLOCK_AVAIL <= '0' when fill_level_reg = "00000" else '1';
```

Die zweite Bedingung wird dadurch erfüllt, dass das PCI-Interface das Signal nur dann auswertet, wenn keine Übertragung stattfindet.

Noch während das PCI-Interface einen Datenblock abarbeitet, muss es entscheiden, ob es den Transfer fortsetzen kann oder die Transferterminierung einleiten muss. Als Entscheidungskriterium dient das Signal `NEXT_BLOCK_AVAIL`. Es ist gesetzt, wenn zwischen dem Datenblock, in dem sich der Lesezeiger der laufenden PCI-Übertragung befindet und dem Datenblock, in dem sich der Schreibzeiger befindet, mindestens ein vollständiger Datenblock liegt.

```
NEXT_BLOCK_AVAIL <= '0' when fill_level_reg(4 downto 1) = "0000"
                    else '1';
```

Für das Modul, das Daten in den FIFO schreibt, gibt es das Steuersignal BLOCK_FREE, das signalisiert, dass noch mindestens ein Speicherblock frei ist.

```
BLOCK_FREE <= '0' when fill_level_reg(4 downto 1) = "1111" else '1';
```

Da sich Schreib- und Lesezeiger nicht einholen dürfen, ist der Block vor dem Datenblock, in dem sich der Lesezeiger befindet, gesperrt. Tritt der Schreibzeiger in den letzten erlaubten Block ein (fill_level_reg = "11110"), wird im weiteren Verlauf BLOCK_FREE auf 0 gesetzt. Bedingt durch die Verzögerungen bei der Taktsynchronisation kann ab dem fünften Datenwort spätestens mit einem gültigen BLOCK_FREE-Signal gerechnet werden. Falls das Signal früher benötigt wird, kann man den verbotenen Bereich um einen Block ausdehnen, um so bei Bedarf einen zusätzlichen Speicherblock als Reserve für die aufgrund des verspäteten BLOCK_FREE-Signals in den FIFO geschriebenen Daten zur Verfügung zu haben.

Entwurfsalternative: Kombinatorische Füllstandsberechnung mit Graycode

Die Schwierigkeit bei der Ermittlung des Füllstands besteht darin, dass sich bei der Erhöhung der Adresszähler im Binärcode im allgemeinen *mehrere* Bits an den Ausgängen der Adresszähler und am Ausgang der kombinatorischen Logik, die die Differenz bildet, ändern. Das Resultat darf deshalb nur zu Zeitpunkten ausgewertet werden, zu denen die Ausgangssignale stabil und alle Laufzeiteffekte abgeklungen sind.

Eine andere Möglichkeit, verlässliche Aussagen über den Füllstand von asynchronen FIFOs zu erhalten, ist es, für den Vergleich zwischen Lese- und Schreibzeiger den Graycode zu verwenden. Das Verfahren kommt ohne eine Synchronisation der Taktsignale aus und ist somit für beliebige Taktfrequenzen geeignet.

Dazu werden der Lesezeiger- und der Schreibzeigerprozess um Ausgangsregister erweitert, die zusätzlich zu den Adressen im Binärcode die mit kombinatorischer Logik in den Graycode umgerechneten Zählerstände liefern. Da sich zwei benachbarte Graycode-Zahlen nur in *einem* Bit unterscheiden, sind die im Graycode dargestellten Zählerstände zu jedem Zeitpunkt gültig. Die Graycode-Adresszeiger werden kombinatorisch miteinander verglichen und die Vergleichsergebnisse in Flipflops übernommen, die entweder vom FIFO-Eingangstakt oder vom FIFO-Ausgangstakt getriggert werden.

6.1.2 Die Backup-Funktion des FIFO-Speichers

Aufgrund des Pipelings innerhalb des LogiCORE PCI Interface kann das PCI-Interface dem tatsächlichen Geschehen auf dem PCI-Bus intern um bis zu zwei Takte

voraus sein. Oder anders gesagt: Nicht alle Daten, die aus dem FIFO gelesen werden, werden auch erfolgreich über den PCI-Bus übertragen.

Wird eine PCI-Übertragung aufgrund eines Fehlerzustands oder durch das Target abgebrochen, muss es eine Möglichkeit geben, die ausgelesenen, aber nicht erfolgreich übertragenen Daten im FIFO wiederherzustellen. Zwei verschiedene Lösungen bieten sich hierbei an:

- Neben dem Lesezeiger, der auf das nächste auszugebende Datenwort zeigt und der nach einem Lesezugriff erhöht wird, wird zweiter Adresszähler implementiert, der immer dann inkrementiert wird, wenn ein Datenwort erfolgreich übertragen wurde. Kommt es zu einer Transferunterbrechung, wird mit einem speziellen Steuersignal der Inhalt des Lesezeigers durch den Wert des zweiten Adresszählers ersetzt.

- Der LogiCORE PCI Design Guide [13] verwendet in seinen Referenzdesigns einen anderen Ansatz, den „Oops-Counter“[1]:

  Darauf aufbauend, dass die Differenz zwischen den bereitgestellten, aber (noch) nicht gesendeten Daten maximal zwei betragen kann, verfolgt der Automat während der PCI-Übertragung das Verhältnis zwischen den mit `M_SRC_EN` angeforderten Daten und den mit `M_DATA_VLD` bestätigten, tatsächlich übertragenen Daten. Dabei sind vier verschiedene Situationen möglich:

M_SRC_EN	**M_DATA_VLD**	**Interpretation**
1	0	Daten angefordert, aber keine übertragen ⇒ Differenz erhöht sich um 1.
0	1	Daten übertragen und keine weiteren angefordert ⇒ Differenz erniedrigt sich um 1.
0	0	Keine Daten angefordert oder übertragen ⇒ Differenz bleibt konstant.
1	1	Daten angefordert und übertragen ⇒ Differenz bleibt konstant.

  Am Ende des Transfers wird dann der FIFO-Lesezeiger um die ermittelte Differenz 0, 1, oder 2 durch eine Subtraktion oder schrittweises Dekrementieren angepasst.

Die erste Variante ist zwar übersichtlicher, bei großen Puffern jedoch ressourcenaufwendiger als die zweite Lösung, da zwei vollständige Adresszähler implementiert

[1] Im Design Guide wird lediglich eine tabellarische 32-zeilige Überführungsfunktion angegeben, ohne die Funktionsweise genauer zu erläutern.

werden müssen. Aufgrund der Einfachheit wurde für diesen FIFO-Speicher die erste Methode gewählt. POP_ACK ist das Inkrementsignal des zweiten Adresszählers. Mit BACK_UP übernimmt der Lesezeiger den Wert des zweiten Adresszählers.

6.2 Das PCI-Interface

Aufgabe des PCI-Busmasters ist es, eine große Menge von Daten, z.B. im Umfang von einem oder mehreren Bildern, aus dem FIFO-Speicher selbständig in den Arbeitsspeicher zu schreiben. Über das im Busmaster enthaltene PCI-Target können die für den DMA-Transfer notwendigen Parameter eingestellt und auch im Verlauf der Übertragung abgefragt werden.

Der Zugriff auf ein Bild ist damit für den Gerätetreiber bzw. für die Anwendersoftware sehr einfach: Der Treiber stellt einen entsprechend großen zusammenhängenden Bereich im Arbeitsspeicher zur Verfügung und beauftragt dann den PCI-Busmaster, dort Daten hineinzuschreiben. Das Ende des Gesamttransfers signalisiert der Busmaster per Interrupt; der Treiber kann, wenn kein Fehler aufgetreten ist, die Daten an das Anwendungsprogramm weiterreichen.

Für den Busmaster gestaltet sich ein gesamter Transfer komplizierter: Der FIFO kann nur einen Bruchteil der angeforderten Daten vorhalten. Da es von der Gesamtsteuerung des Systems abhängt, wann nach einem Leerlaufen des FIFOs neue Daten bereitstehen, ist das Einfügen von Wartezyklen in den laufenden Transfer nicht sinnvoll. Zum einen würden sie den PCI-Bus für andere Geräte blockieren, zum anderen setzt die PCI-Spezifikation der Anzahl der erlaubten Wartezyklen gerade deshalb sehr enge Grenzen [2].

Aus diesen Gründen enthält die Busmasterlogik zwei Automaten, den *PCI-Automat* für die Abwicklung einer Burstübertragung über den PCI-Bus und den übergeordneten *Transferautomat*, der den gesamten DMA-Transfer koordiniert.

Wie der FIFO-Speicher arbeitet auch der PCI-Busmaster mit Datenblöcken zu je acht Doppelwörtern. Der Gerätetreiber kann nur Datenmengen (DWords) anfordern, die ein Vielfaches von acht sind. Zugleich bestimmt dieses Achter-Schema die internen Abläufe des PCI-Busmasters.

Alle in der folgenden Beschreibung zitierten Codestücke sind gemäß Abbildung 6.1 aus der Datei pci_interface_top.vhd entnommen.

6.2.1 Die PCI-Targetschnittstelle

Die Komponente target_control innerhalb des PCI-Busmasters in Abbildung 6.1 implementiert die Steuerlogik für ein einfaches Single Transfer PCI-Target mit vier 32 Bit Registern.

```
entity target_control is
port (
  RST         : in std_logic;
  CLK         : in std_logic;
  BASE_HIT    : in std_logic_vector(7 downto 0);
  S_DATA      : in std_logic;
  S_WRDN      : in std_logic;
  ADDR        : in std_logic_vector(31 downto 0);
  S_DATA_VLD  : in std_logic;
  S_READY     : out std_logic;
  S_TERM      : out std_logic;
  S_ABORT     : out std_logic;

  READ_REG    : out std_logic_vector(3 downto 0);
  WRITE_REG   : out std_logic_vector(3 downto 0)
);
end target_control;
```

Die Busse READ_REG[3:0] und WRITE_REG[3:0] (im Folgenden read_target_reg und write_target_reg) zeigen einen Lese- oder Schreibzugriff auf das entsprechende Register in One-Hot-Codierung an. Die Register selbst befinden sich im Modul pci_interface_top.

Register DMA-Zieladresse (lesen und schreiben), Offset 0x0

Das Register pci_addr[31:2] enthält die Zieladresse für den PCI-Transfer. Sie wird mit jedem erfolgreich übertragenen Datenwort inkrementiert. Da es sich um 32 Bit Transfers handelt, sind die unteren zwei Adressbits nicht relevant, siehe Abschnitt 3.8.1. In der Adressphase einer PCI-Übertragung oder bei einem Lesezugriff auf das Register wird sein Inhalt auf den ADIO-Bus gelegt.

```
-- Adressoffset des Registers
constant TARGET_ADDR_STARTADDRESS : integer := 0;
signal pci_addr : std_logic_vector(31 downto 2);

process (RST, CLK)
begin
  if RST = '1' then
    pci_addr <= (others => '0');
  elsif CLK'event and CLK = '1' then
    if write_target_reg(TARGET_ADDR_STARTADDRESS) = '1' then
      pci_addr <= ADIO(31 downto 2);
```

```
    elsif M_DATA_VLD = '1' then
      pci_addr <= pci_addr + '1';
    end if;
  end if;
end process;

ADIO <= pci_addr & "00" when M_ADDR_N = '0'
          or read_target_reg(TARGET_ADDR_STARTADDRESS) = '1'
        else (others => 'Z');
```

Register DMA-Transferzähler (lesen und schreiben), Offset 0x4

In diesem Register wird die Anzahl der zu übertragenden Datenwörter vor dem Beginn eines DMA-Transfers gespeichert. Da jede Transferlänge ein Vielfaches von acht ist, werden die unteren drei Bits bei der Initialisierung des Registers auf 0 gesetzt. Mit jedem erfolgreich übertragenen Datenwort wird der Wert dekrementiert.

```
-- Adressoffset des Registers
constant TARGET_ADDR_XFER_CNT : integer := 1;
signal xfer_cnt : std_logic_vector(31 downto 0);

process (RST, CLK)
begin
  if RST = '1' then
    xfer_cnt <= (others => '0');
  elsif CLK'event and CLK = '1' then
    if write_target_reg(TARGET_ADDR_XFER_CNT) = '1' then
      xfer_cnt <= ADIO(31 downto 3) & "000";
    elsif M_DATA_VLD = '1' then
      xfer_cnt <= xfer_cnt - '1';
    end if;
  end if;
end process;

ADIO <= xfer_cnt when read_target_reg(TARGET_ADDR_XFER_CNT) = '1'
        else (others => 'Z');
```

Steuer- / Befehlsregister (nur schreiben), Offset 0x8

Das Steuerregister umfasst 32 Bit. Bislang sind nur zwei Befehle in One-Hot-Codierung implementiert:

- Bit 0: Start des DMA-Transfers. Das Bit `start_xfer` bleibt so lange gesetzt, bis der Transfer abgeschlossen oder durch den Resetbefehl abgebrochen wurde.
- Bit 1: Reset. Über dieses Bit können Teile der Hardware durch den Treiber in den Einschaltzustand versetzt werden, z.B. der PCI-Automat nach einem Master Abort oder einem Target Abort.

Bei Bedarf können weitere Steuerbits hinzugefügt werden.

```
-- Adressoffset des Registers
constant TARGET_ADDR_CONTROL      : integer := 2;
-- Bits innerhalb des Registers
constant TARGET_CMD_XFER_START    : integer := 0;
constant TARGET_CMD_RESET         : integer := 1;

signal start_xfer     : std_logic;

process (CLK, RST)
begin
  if RST = '1' then
    start_xfer <= '0';
  elsif CLK'event and CLK = '1' then
    if xfer_cnt = x"00000000" then
      start_xfer <= '0';
    elsif write_target_reg(TARGET_ADDR_CONTROL) = '1'
          and ADIO(TARGET_CMD_XFER_START) = '1' then
      start_xfer <= '1';
    end if;
  end if;
end process;

TARGET_RESET <= write_target_reg(TARGET_ADDR_CONTROL)
                and ADIO(TARGET_CMD_RESET);
XFER_RUNNING <= start_xfer;
```

Statusregister (nur lesen), Offset 0xC

Im Status-Register wird bislang nur das Bit 0 verwendet. Es enthält das Interrupt-Flag `intr_n_flag`.

```
-- Adressoffset des Registers
constant TARGET_ADDR_STATUS       : integer := 3;
```

```
ADIO <= x"0000000" & "000" & intr_n_flag
          when read_target_reg(TARGET_ADDR_STATUS) = '1'
        else (others => 'Z');
```

Bei Bedarf können weitere Statusinformationen auf den verbleibenden Bits ausgegeben werden.

6.2.2 Der Transferautomat

Die Aufgabe des Transferautomaten ist es, den Burstzähler `pci_burst_cnt[3:0]` für den PCI-Automaten zu bilden.

```
pci_burst_cnt <= pci_burst_continue & xfer_cnt(2 downto 0);
```

In den unteren drei Bits läuft er mit den unteren drei Bits des DMA-Transferzählers synchron. Mit dem oberen Bit `pci_burst_continue` steuert der Transferautomat die Fortsetzung bzw. Beendigung der laufenden PCI-Burstübertragung.

Durch das Setzen von `pci_burst_continue` auf 1 kann der Transfer verlängert werden, da der Burstzähler dann zyklisch zwischen 15 und 8 herunterzählt. Anderenfalls liegen die Werte zwischen 7 und 0. Auch hier kann die Burstübertragung noch durch Setzen von `pci_burst_continue` auf 1 wieder verlängert werden, mit einer Einschränkung jedoch: Hat der Burstzähler einmal den Wert 4 erreicht, beginnt die Terminierungsphase der PCI-Übertragung. Der Transfer darf dann nicht mehr verlängert werden. Der Transferautomat muss sicherstellen, dass der Burstzähler nun bis zum Wert 0 kontinuierlich herunterzählt, damit die Übertragung ordnungsgemäß beendet wird.

Der Transferautomat hat zwei Zustände: den Ruhezustand `IDLE_S` und den Transferzustand `XFER_S`.

Wird vom Gerätetreiber über das PCI-Target ein DMA-Transfer mit dem Signal `start_xfer` gestartet, wechselt der Transferautomat in den Transferzustand, wenn ein vollständiger Datenblock im FIFO bereitsteht und sich der PCI-Automat im Ruhezustand befindet. Das Startsignal für den PCI-Automaten wird kombinatorisch gebildet:

```
start_pci_burst <= '1' when pci_burst_continue = '1'
                   and xfer_state = XFER_S else '0';
```

Im Transferzustand wird fortlaufend geprüft, ob auch schon der nächste Datenblock vollständig im FIFO vorhanden ist.

1. Fall: `fifo_next_block_available = '1'`
Die PCI-Übertragung wird fortgesetzt, wenn mindestens ein weiterer Datenblock zu übertragen ist, d.h. der DMA-Transferzähler `xfer_cnt` größer als 7 ist, und sich die Übertragung noch nicht in der Terminierungsphase befindet.

2. Fall: `fifo_next_block_available = '0'`
Es steht im Moment kein weiterer Datenblock zur Übertragung an. Deshalb muss in diesem Fall sichergestellt werden, dass der Burstzähler `pci_burst_cnt` korrekt abwärts zählt.

Hat der PCI-Automat seine Burstübertragung abgeschlossen, wechselt der Transferautomat in jedem Fall wieder in den Ruhezustand, auch wenn der DMA-Transfer insgesamt noch nicht vollständig ist.

6.2.3 Der PCI-Automat

Der PCI-Automat wickelt eine Burstübertragung über den PCI-Bus ab. Eine Übertragung wird durch den Transferautomaten initiiert und über den vom Transferautomaten gebildeten Burstzähler `pci_burst_cnt[3:0]` gesteuert.

Im Verlauf eines Bursttransfers kommt es vor, dass dem Busmaster die Busrechte vorzeitig entzogen werden, weil z.B. ein anderes PCI-Gerät den Bus benötigt. Um die erneute Busanforderung und die Wiederaufnahme der Übertragung kümmert sich der PCI-Automat selbständig.

Automateneingänge

`start_pci_burst` wird von dem Transferautomaten gesetzt und zeigt an, dass der PCI-Automat eine Übertragung beginnen soll.

`m_data_fell` signalisiert das Ende der Datenphase des PCI-Transfers. Dieses ist genau dann erreicht, wenn am Ausgangssignal `M_DATA` des LogiCORE PCI Interface eine fallende Flanke beobachtet wird.

```
process (RST, CLK)
begin
  if RST = '1' then
    m_data_delay <= '0';
  elsif CLK'event and CLK = '1' then
    m_data_delay <= M_DATA;
  end if;
end process;

m_data_fell <= m_data_delay and not M_DATA;
```

Das Signal fatal zeigt an, dass ein Master Abort oder ein Target Abort aufgetreten ist. Es wird aus den erweiterten Statussignalen CSR[39:32] gewonnen. Da das entsprechende Ereignis dort nur für einen Takt, nachdem es aufgetreten ist, signalisiert wird, speichert ein Flipflop das Fehlersignal. Während der Adressphase der nächsten PCI-Übertragung wird es wieder gelöscht.

```
process (RST, CLK)
begin
  if RST = '1' then
    fatal <= '0';
  elsif CLK'event and CLK = '1' then
    if M_ADDR_N = '0' then
      fatal <= '0';
    elsif M_DATA = '1' then
      fatal <= CSR(39) or CSR(38);
    end if;
  end if;
end process;
```

Wenn der Transferzähler pci_burst_cnt[3:0] des PCI-Automaten den Wert 0 erreicht hat, ist eine Übertragungssequenz abgeschlossen, und das Signal done wird gesetzt. Da pci_burst_cnt[3:0] durch den übergeordneten Transferautomaten gebildet wird und sich dort ändern kann, wird done in einem Flipflop gespeichert.

```
process (CLK, RST)
begin
  if RST = '1' then
    done <= '0';
  elsif CLK'event and CLK = '1' then
    if pci_burst_cnt = x"0" then
      done <= '1';
    elsif pci_state = PCI_IDLE_S or pci_state = PCI_REQ_S then
      done <= '0';
    end if;
  end if;
end process;
```

Der Zustandsgraph

Abbildung 6.3 zeigt den Zustandsgraphen des PCI-Automaten.

PCI_IDLE_S ist der Anfangs- und Ruhezustand des Automaten.

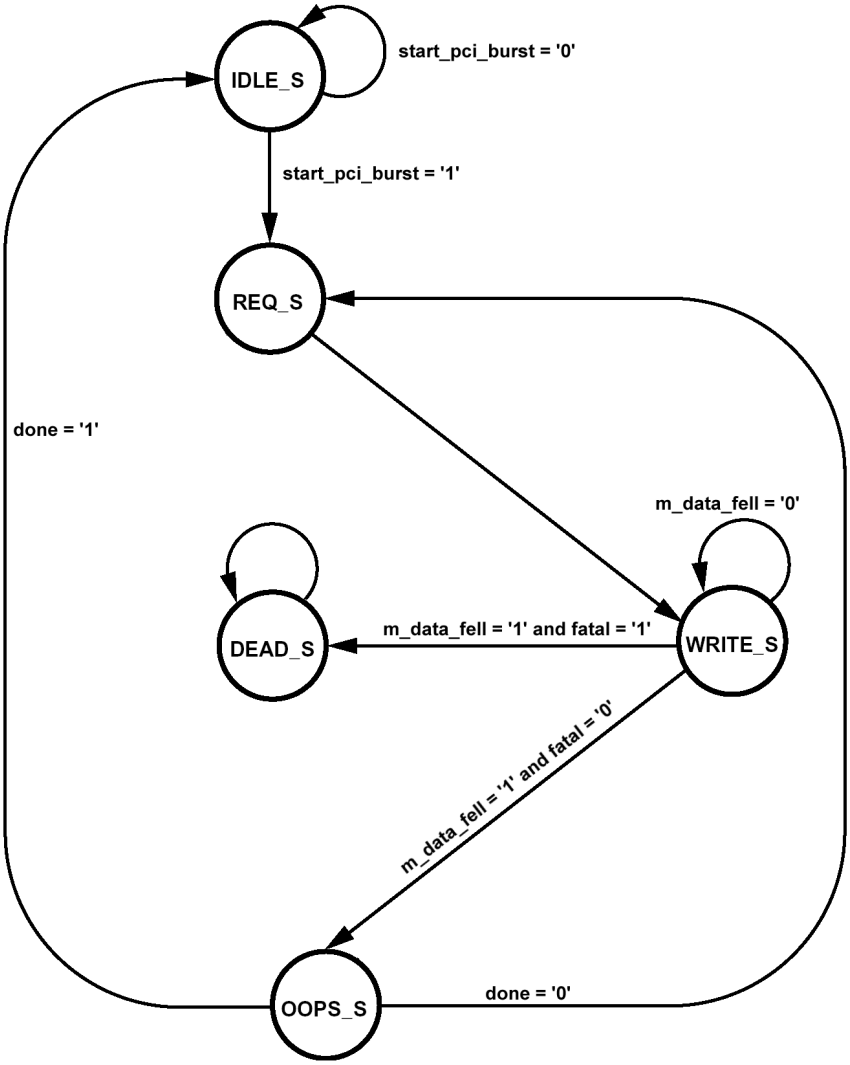

Abbildung 6.3: Zustandsgraph des PCI-Automaten

Der Automat wechselt in den Zustand `PCI_REQ_S`, wenn das Signal `start_pci_burst` vom übergeordneten Tansferautomaten gesetzt wird.

Mit der nächsten steigenden Taktflanke geht der Automat in den Zustand `PCI_WRITE_S` über und verbleibt dort, bis die Datenphase der Übertragung abgeschlossen wurde. Das Signal `fatal` entscheidet über den Folgezustand.

Wenn ein Fehler bei der Übertragung aufgetreten ist, springt der Automat in den Fehlerzustand `PCI_DEAD_S`, der nur durch einen Reset des Automaten verlassen werden kann.

Im Normalfall wechselt der Automat in den Zustand `PCI_OOPS_S`, in dem der Lesezeiger des FIFOs um die Anzahl der ausgelesenen, aber nicht über den PCI-Bus übertragenen Datenwörter korrigiert wird. Der Zustand des Signals `done` entschei-

det darüber, ob der Transfer abgeschlossen ist oder aufgrund einer Unterbrechung fortgesetzt werden muss.

Ausgangssignale zum PCI Interface

Um einen PCI-Transfer einzuleiten, muss das Signal REQUEST für einen Takt gesetzt werden. Dies geschieht im Zustand PCI_REQ_S.

```
req <= '1' when pci_state = PCI_REQ_S else '0';
REQUEST <= req;
```

Um einen PCI-Transfer zu beenden, muss das Signal COMPLETE zu Beginn der vorletzten Datenphase (pci_burst_cnt=2) gesetzt sein und gesetzt bleiben, bis die Übertragung abgeschlossen wurde.

```
assert_complete <= fin1 or fin2 or fin3;  -- siehe unten

process (RST, CLK)
begin
  if RST = '1' then
    hold_complete <= '0';
  elsif CLK'event and CLK = '1' then
    if m_data_fell = '1' then
      hold_complete <= '0';
    elsif assert_complete = '1' then
      hold_complete <= '1';
    end if;
  end if;
end process;

COMPLETE <= assert_complete or hold_complete;

cnt3 <= '1' when pci_burst_cnt = x"3" else '0';
cnt2 <= '1' when pci_burst_cnt = x"2" else '0';
cnt1 <= '1' when pci_burst_cnt = x"1" else '0';
```

Drei Fälle müssen bei der Bildung von assert_complete unterschieden werden:

1. Wenn nur ein Datenwort übertragen werden soll, wird COMPLETE gleich zu Beginn des Transfers gesetzt. Dies ist genau dann der Fall, wenn im Zustand PCI_REQ_S die Burstlänge pci_burst_cnt=1 ist.

   ```
   fin1 <= cnt1 and req;
   ```

2. COMPLETE wird gesetzt, wenn pci_burst_cnt=2 ist und sich der PCI-Transfer in der Datenphase befindet.

```
fin2 <= cnt2 and m_data_delay;
```

3. COMPLETE wird gesetzt, wenn pci_burst_cnt=3 und M_DATA_VLD gesetzt ist. Mit der nächsten steigenden Flanke wird pci_burst_cnt durch M_DATA_VLD dekrementiert, und COMPLETE muss dann schon gesetzt sein. Bei M_DATA_VLD='0' ändert sich pci_burst_cnt nicht, also darf COMPLETE noch nicht gesetzt werden.

```
fin3 <= cnt3 and M_DATA_VLD;
```

Zuletzt muss noch der M_CBE-Bus (Command/Byte Enables) bedient werden. Er enthält während der Adressphase der PCI-Übertragung den PCI-Busbefehl, siehe Abschnitt 3.6. In dieser Anwendung ist es immer der Befehl *Memory Write* (0111). Da Transfers im Speicheradressraum immer mit voller 32 Bit Busbreite erfolgen, werden in den Datenphasen keine Bytes maskiert.

```
M_CBE <= "0111" when M_ADDR_N = '0' else "0000";
```

6.2.4 Anbindung an den FIFO-Speicher

Im Zustand PCI_OOPS_S wird der Lesezeiger des FIFOs um die Anzahl der ausgelesenen, aber nicht übertragenen Datenwörter korrigiert.

```
FIFO_BACK_UP <= '1' when pci_state = PCI_OOPS_S else '0';
```

Das LogiCORE PCI Interface fordert neue Daten während der Übertragung mit dem Signal M_SRC_EN an. Außerdem muss zu Beginn jeder Übertragung das erste Datenwort aus dem FIFO geholt werden.

```
FIFO_POP <= '1' when (M_SRC_EN = '1' and pci_state = PCI_WRITE_S)
            or pci_state = PCI_REQ_S else '0';
```

Mit M_DATA_VLD bestätigt das PCI-Interface jedes erfolgreich übertragene Datenwort.

```
FIFO_POP_ACK <= M_DATA_VLD;
```

In der Datenphase des PCI-Transfers wird der Datenausgang des FIFOs FIFO_DATA_OUT[31:0] auf den ADIO-Bus des PCI-Interfaces durchgeschaltet.

```
ADIO <= FIFO_DATA_OUT when M_DATA = '1' else (others => 'Z');
```

6.2.5 Interruptlogik

Ein Interrupt soll nach erfolgreichem Transferabschluss oder im Fehlerfall nach einem Master Abort oder Target Abort ausgelöst werden. Das Interrupt-Flag bleibt so lange gesetzt, bis ein Lesezugriff auf das Statusregister des PCI-Targets erfolgte.

```
process (CLK, RST)
variable read_target_reg_delay : std_logic;
begin
  if RST = '1' then
    read_target_reg_delay := '0';
    intr_n_flag <= '1';
  elsif CLK'event and CLK = '1' then
    if (xfer_cnt = x"00000000" and pci_burst_continue = '0')
          or CSR(39) = '1' or CSR(38) = '1' then
      intr_n_flag <= '0'; -- negierte Logik!
    elsif read_target_reg(TARGET_ADDR_STATUS) = '0'
          and read_target_reg_delay = '1' then
      intr_n_flag <= '1';
    end if;
    read_target_reg_delay := read_target_reg(TARGET_ADDR_STATUS);
  end if;
end process;

INTR_N <= intr_n_flag;
```

6.3 Gerätetreiber und Beispielprogramme

Der Gerätetreiber für den PCI-Busmaster ist immer auch von der konkreten Anwendung abhängig, in die der PCI-Busmaster integriert wird. Das Kernelmodul und die Programme für die Beispielanwendung Framegrabber in Abschnitt 8.2 sind eine Referenz für eigene Entwicklungen.

7 Gerätetreiber unter Linux

Referenz für die Gerätetreiberentwicklung unter Linux (Kernelversion 2.4) ist das Buch Linux Device Drivers [4]. Darüber hinaus enthält das während dieser Arbeit entstandene Tutorial [6] zu allen dort vorgestellten Beispielentwürfen ausführlich erläuterte Gerätetreiber.

Aufgrund der Komplexität der Thematik kann an dieser Stelle nur ein kurzer allgemeiner Einblick in den Aufbau und die Funktionsweise eines Treibers für zeichenorientierte Geräte gegeben werden.

Treiber für DMA-fähige Geräte haben eine Besonderheit: Sie müssen einen Bereich im Hauptspeicher allozieren, der den Anforderungen für einen DMA-Transfer genügt. Darauf wird in den Abschnitten 7.3 und 7.4 eingegangen.

Der Abschnitt 7.5 schließlich stellt einen Trick vor, mit dem sich das zeitraubende obligatorische Neustarten des Rechners nach einer Rekonfiguration des FPGAs mit Hilfe des Gerätetreibers umgehen lässt.

7.1 Überblick

Grundsätzlich wird die PCI-Karte als zeichenorientiertes Gerät (character device) aufgefasst. Zeichenorientiert bedeutet, dass auf das Gerät wie auf einen Stream von Bytes bzw. wie auf eine Datei zugegriffen wird.

Zeichengeräte werden mit Dateisystemknoten verwaltet, die für das entsprechende Gerät angelegt werden, z.B. /dev/xilinx_pci. Der Unterschied zwischen einem Zeichengerät und einer normalen Datei besteht darin, dass sich bei der Abstraktion einer Hardwarekomponente als Datei /dev/xilinx_pci nicht alle Dateifunktionen sinnvoll auf das Gerät übertragen lassen. Der Hardwareentwurf Framegrabber in Kapitel 8 beispielsweise gestattet nur den sequentiellen Lesezugriff auf Bilddaten. Schreiboperationen sind damit ebenso wenig anwendbar wie die Funktionen, mit denen man in einer regulären Datei den internen Dateizeiger positionieren kann.

Die Aufgabe eines Treibers für ein zeichenorientiertes Gerät ist es, dieses dateiähnliche Verhalten umzusetzen, indem er die Systemaufrufe wie z.B. `open()`, `close()`, `read()` und `write()` implementiert. Wird also in einem Anwendungsprogramm eine der Dateifunktionen im Zusammenhang mit der zur Hardware gehörenden Gerätedatei aufgerufen, so werden die zugehörigen Gegenstücke im Kernelmodul ak-

tiv. Wird keine eigene Funktion angegeben, hat der Kernel ein eingebautes Defaultverhalten ohne weitere Auswirkungen für den Fall, dass ein Programm es trotzdem versucht, die Funktion auf die Gerätedatei anzuwenden.

Die eigenen Dateiroutinen werden in einer Struktur vom Typ `file_operations` eingetragen. Sie ist ein Feld von Zeigern auf die Funktionen. Die genaue Definition der Struktur `file_operations` mit allen theoretisch anwendbaren Dateioperationen der Kernelversion 2.4 befindet sich in der oben angegebenen Literatur.

In der Funktion `init_module()` werden die eigenen Dateifunktionen mit dem Befehl `register_chrdev()` für das Gerät registriert. Neben einem Zeiger auf die Dateioperationen-Struktur werden noch ein Name für den Treiber sowie die sogenannte Major-Nummer und die Minor-Nummer der Gerätedatei übergeben. Die Major-Nummer bezeichnet eine Gerätefamilie, während die Minor-Nummer einem bestimmten Gerät innerhalb der Familie entspricht. Gültige Werte und die zugeordneten Geräte werden in der Datei /usr/src/linux/Documentation/devices.txt im Linux-System beschrieben. Die Major-Nummer 240 ist für experimentelle Zwecke vorgesehen und wird deshalb für das Entwicklungsboard benutzt.

Die Funktion `init_module()` ist in jedem Treiber vorhanden. Sie wird beim Laden bzw. Einbinden des Moduls in den laufenden Kernel ausgeführt. Sie identifiziert die Zielhardware und initialisiert den Treiber. Analog dazu wird die Funktion `cleanup_module()` beim Entfernen des Moduls aus dem System aufgerufen.

7.2 Inbetriebnahme des Gerätetreibers

Ein Kernelmodul (hier xilinx_pci.c) muss zwingend mit den beiden Präprozessoroptionen `-D__KERNEL__` und `-DMODULE` compiliert werden. Sie geben bestimmte Datenstrukturen in den importierten, systemnahen Headerdateien frei.

```
gcc -Wall -D__KERNEL__ -DMODULE -c xilinx_pci.c
```

Bevor der Treiber erstmalig in den laufenden Kernel eingebunden werden kann, muss die Gerätedatei (hier /dev/xilinx_pci) mit

```
mknod /dev/xilinx_pci c 240 0
chmod a+rw /dev/xilinx_pci
```

angelegt werden. Der Parameter c steht für ein zeichenorientiertes Gerät. 240 ist die Major-Nummer und 0 die Minor-Nummer.

Das compilierte Modul xilinx_pci.o wird mit

```
insmod xilinx_pci.o
```

(insert module) geladen. Aufschluss über erfolgreich geladene Module geben die Kommandos

```
lsmod
cat /proc/modules
```

Mit dem Befehl

```
rmmod xilinx_pci
```

kann man das geladene Modul wieder entladen.

7.3 Speicher für DMA reservieren

Die Besonderheit bei einem DMA-Puffer besteht darin, dass es ein im physischen Speicher zusammenhängender Bereich sein muss. Umfasst die Puffergröße mehr als eine Speicherseite, müssen die für den Puffer allozierten Seiten nebeneinander liegen, weil das DMA-Gerät, das Daten über den PCI-Bus transportiert, mit der physischen Adresse arbeitet.

Die folgenden Abschnitte stellen die geläufigen Methoden zur Speichereservierung vor:

7.3.1 Allokation mit kmalloc()

Der Allokationsmechanismus von `kmalloc()` ist ein mächtiges Werkzeug, das wegen seiner Ähnlichkeit zu `malloc()` einfach zu benutzen ist. Sofern sie nicht blockiert, arbeitet die Funktion schnell, und der allozierte Bereich ist auch im physischen Speicher zusammenhängend.

```
void *kmalloc (size_t size, int flags);
void kfree (void *obj);
```

Das Argument `size` gibt die gewünschte Speichergröße an. Das Argument `flags` enthält Angaben über die Art und die Position des Speicherbereichs. Dafür sind symbolische Konstanten definiert. Rückgabewert ist bei Erfolg ein Zeiger auf den allozierten Speicher, bei Misserfolg NULL.

`kmalloc()` kann maximal 128 KByte Speicher belegen. Wenn mehr als ein paar KBytes benötigt werden, gibt es bessere Möglichkeiten, den Speicher anzufordern.

7.3.2 Allokation mit get_free_pages()

Wenn ein Kernelmodul große Speicherblöcke allozieren muss, ist es besser, eine seitenorientierte Technik zu verwenden. Die Funktion `__get_free_pages()` versucht, die angegebene Anzahl physisch zusammenhängender Seiten im Hauptspeicher zu allozieren. In den Kernelversionen nach Kernel 2.0 können maximal $2^9 = 512$ Seiten alloziert werden, was bei der üblichen Seitengröße von 4 KByte immerhin 2 MByte ergibt.

```
unsigned long __get_free_pages (int flags, unsigned long order);
void free_pages (unsigned long addr, unsigned long order);
```

Das Argument `flags` enthält Angaben über die Art und die Position des Speicherbereichs wie bei `kmalloc()`. `order` ist der Exponent der Zweierpotenz der Anzahl der Seiten, die angefordert oder freigegeben werden sollen, also $order = \lceil log_2 N \rceil$ mit N als der Anzahl der gewünschten Seiten. Ist kein zusammenhängender Speicherbereich der gewünschten Größe vorhanden, schlägt die Allokation fehl.

Zur Berechnung der Anzahl der benötigten Seiten kann die Größe einer Seite über das Makro `PAGE_SIZE` bestimmt werden.

7.3.3 Selbstgemachte Allokation

Für noch größere Puffer kann man das obere Ende des physischen RAMs von vornherein reservieren, indem man das Argument `mem=...` an den Kernel beim Systemstart übergibt.

Beispiel: Sind 32 MByte RAM vorhanden, verbietet das Argument `mem=31MB` dem Kernel, das oberste MByte als normalen Hauptspeicher zu nutzen. Um auf diesen Speicher zuzugreifen, kann dann im Kerneltreiber die Funktion `ioremap()` verwendet werden:

```
dmabuf = ioremap (0x1F00000,     /* Startadresse 31MB */
                  0x100000);     /* Puffergröße   1MB */
```

7.3.4 Allokation zur Bootzeit

Die Allokation zur Bootzeit ist sehr unflexibel. Über entsprechende Funktionen lässt sich physischer Speicher zur Boot-Zeit reservieren. Das setzt jedoch voraus, dass der Treiber fest in den Kernel gelinkt ist.

7.4 DMA auf dem PCI-Bus

Der Kernel 2.4 enthält mit den DMA-Einblendungen (DMA Mappings) einen flexiblen Mechanismus, der DMA auf dem PCI-Bus unterstützt:

DMA-Einblendungen sind eine Kombination aus der Allokation eines DMA-Puffers und dem Erzeugen einer Adresse für diesen Speicherbereich, mit der das PCI-Gerät auf den Puffer zugreifen kann. Die DMA-Einblendung führt einen neuen Typ namens `dma_addr_t` ein, um Busadressen zu repräsentieren. Die einzigen zulässigen Operationen für diesen Datentyp sind die Übergabe an die DMA-Hilfsroutinen und an das Gerät selbst.

Je nachdem, wie lange der DMA-Puffer vorgehalten werden soll, wird zwischen zwei Typen von DMA-Einblendungen unterschieden:

7.4.1 Konsistente DMA-Einblendungen

Konsistente DMA-Einblendungen (Consistent DMA Mappings) existieren während der gesamten Lebenszeit des Treibers. Ein konsistent eingeblendeter Puffer muss gleichzeitig sowohl der CPU als auch dem Peripheriegerät zur Verfügung stehen. Der Puffer sollte auch keine Caching-Probleme haben, die dazu führen könnten, dass Aktualisierungen der einen Partie von der jeweils anderen nicht gesehen werden können.

```
void *pci_alloc_consistent (struct pci_dev *pdev, size_t size,
                            dma_addr_t *bus_addr);
```

Diese Funktion erledigt sowohl die Allokation als auch die Einblendung des Puffers. Die ersten beiden Argumente sind die PCI-Gerätestruktur und die Größe des benötigten Puffers in Bytes. Der Rückgabewert ist die virtuelle Kerneladresse des Puffers, die vom Treiber verwendet werden kann. Die zugehörige Busadresse wird im Argument `bus_addr` zurückgegeben. Die Allokation wird in dieser Funktion erledigt, damit der Puffer an einer für DMA geeigneten Stelle eingerichtet wird. In der Regel wird der Speicher einfach mit `get_free_pages()` alloziert.

Wenn der Puffer nicht mehr benötigt wird, was normalerweise beim Entladen des Moduls der Fall ist, wird er mit

```
void pci_free_consistent (struct pci_dev *pdev, size_t size,
                          void *cpu_addr, dma_handle_t bus_addr);
```

an das System zurückgegeben. Die Funktion benötigt sowohl die virtuelle Adresse als auch die Busadresse.

7.4.2 Streaming DMA-Einblendungen

Streaming DMA-Einblendungen (Streaming DMA Mappings) werden für eine einzelne Operation eingerichtet. Manche Architekturen ermöglichen in diesem Fall nennenswerte Optimierungen, jedoch unterliegen diese Einblendungen auch strengeren Zugriffsregeln. Die Literatur [4] empfiehlt, Streaming-Einblendungen den konsistenten Einblendungen vorzuziehen, wo immer das möglich ist.

Bei der Einrichtung einer Streaming-Einblendung wird ein bereits vom Treiber allozierter Puffer vorausgesetzt. Außerdem muss dem Kernel über symbolische Konstanten mitgeteilt werden, in welcher Richtung sich die Daten bewegen sollen. Werden Daten vom Gerät gelesen, z.B. als Reaktion auf einen `read()`-Systemaufruf, dann sollte `PCI_DMA_FROMDEVICE` verwendet werden, anderenfalls `PCI_DMA_TODEVICE`. Soll über den Puffer bidirektionaler Datenverkehr abgewickelt werden, wird `PCI_DMA_BIDIRECTIONAL` verwendet. Auf manchen Plattformen wird man mit einem Leistungsverlust bestraft, wenn man nicht den richtigen exakten Wert für die Richtung einer Streaming DMA-Einblendung angibt und stattdessen `PCI_DMA_BIDIRECTIONAL` wählt.

Der DMA-Puffer wird mit

```
dma_addr_t pci_map_single (struct pci_dev *pdev, void *buffer,
                           size_t size, int direction);
```

eingeblendet. Der Rückgabewert ist die Busadresse, die dann an das PCI-Gerät übergeben werden kann, oder NULL im Fehlerfall.

Wenn die Übertragung abgeschlossen ist, wird die Einblendung mit

```
void pci_unmap_single (struct pci_dev *pdev, dma_addr_t bus_addr,
                       size_t size, int direction);
```

wieder aufgehoben, wobei die Argumente `size` und `direction` mit den zuvor verwendeten übereinstimmen müssen. `buffer` ist der Zeiger auf den zuvor z.B. mit `kmalloc()` oder `__get_free_pages()` reservierten Speicherbereich.

Wenn ein Puffer eingeblendet worden ist, gehört er dem Gerät, nicht dem Prozessor. Bis der Speicherbereich wieder ausgeblendet worden ist, darf der Treiber den Inhalt in keiner Weise anfassen. Diese Regel impliziert, dass ein Puffer, dessen Inhalt auf das Gerät geschrieben wird, nicht eingeblendet werden kann, bevor nicht alle zu schreibenden Daten vorliegen. Andererseits darf der Puffer nicht ausgeblendet werden, solange die DMA-Übertragung noch läuft, ansonsten ist die Systemstabilität gefährdet.

7.5 FPGA-Rekonfiguration ohne Neustart des Rechners

Eine PCI-Karte wird nach dem Einschalten des Rechners durch das PCI-BIOS und gegebenenfalls später noch einmal durch das Betriebssystem konfiguriert. Ressourcen wie Anfangsadressen für die angeforderten Adressräume oder ein IRQ werden dabei zugeteilt.

Bei der Rekonfiguration des FPGAs im laufenden Betrieb gehen diese im PCI Configuration Space Header des PCI-Geräts eingetragenen Informationen verloren. Der Rechner muss neu gestartet werden, um die PCI-Karte wieder zu initialisieren.

Ist die Konfiguration des LogiCORE PCI Interface in dem neuen Hardwareentwurf jedoch gleich geblieben, d.h. im VHDL-Modul cfg hat sich nichts geändert, kann mit Hilfe des Gerätetreibers der Inhalt des PCI Configuration Space Headers wiederhergestellt werden.

Dazu liest der Treiber den PCI Configuration Space Header der durch einen Neustart nach der Programmierung korrekt initialisierten PCI-Karte aus und speichert ihn. Dann wird das FPGA mit dem neuen Entwurf programmiert; der Kerneltreiber bleibt dabei geladen. Das Programm restore_config.c (siehe unten) veranlasst schließlich den Treiber, den uninitialisierten PCI Configuration Space Header der PCI-Karte mit der zuvor gesicherten Konfiguration zu überschreiben.

Erweiterung des Gerätetreibers

In der Funktion `init_module()` wird der PCI Configuration Space Header der korrekt initialisierten PCI-Karte beim Laden des Treibers gespeichert.

```
struct pci_dev *pci_dev;  /* Software-Repraesentation der PCI-Karte */
u32 config_space[64];     /* Konfigurationsspeicher der PCI-Karte   */

int init_module (void) {
  [...]
  for (i = 0; i < 64; i++)
    pci_read_config_dword (pci_dev, i * 4, config_space + i);
  [...]
}
```

Eine `ioctl()`-Funktion schreibt den zuvor gesicherten PCI Configuration Space Header wieder auf das Gerät zurück. Die Funktion muss dafür als erlaubte Dateioperation für das Gerät in der Struktur `file_operations` angemeldet werden.

```
int xilinx_pci_ioctl (struct inode *inode, struct file *file,
                      unsigned int cmd, unsigned long arg) {
  int i;
  if (cmd == 0) {
    for (i = 0; i < 64; i++)
      pci_write_config_dword (pci_dev, i * 4, config_space[i]);
    return 0;
  }
  return -EIO;
}
```

Wiederherstellung des PCI Configuration Headers

Das Programm restore_config.c öffnet die zum PCI-Gerät gehörende Gerätedatei und ruft die `ioctl()`-Funktion für die Wiederherstellung auf.

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/ioctl.h>

#define PCI_DEVICE "/dev/xilinx_pci"

int main () {
  int pci_dev = open (PCI_DEVICE, O_RDWR, 0);
  if (pci_dev == -1)
    printf ("Fehler beim Oeffnen des PCI-Devices.\n");
  else {
    ioctl (pci_dev, 0, NULL);
    close (pci_dev);
  }
  return 0;
}
```

8 Beispielanwendung Framegrabber

Alle in diesem Kaptitel zitierten Dateien befinden sich auf der beiliegenden CD im Verzeichnis /projekte/framegrabber/.

8.1 Der Hardwareentwurf

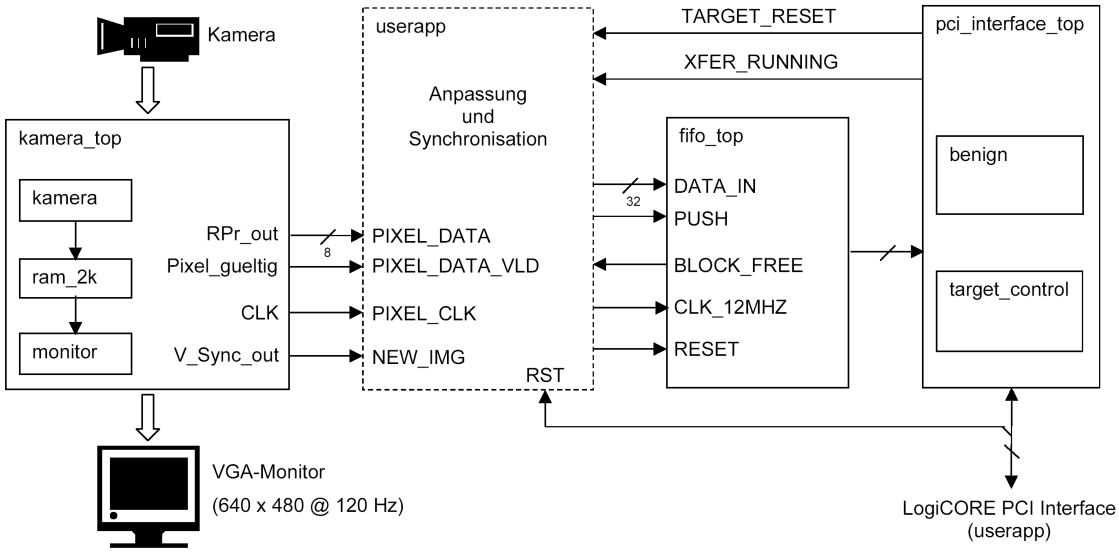

Abbildung 8.1: Blockdiagramm des Framegrabbers

Der Framegrabber baut auf Carsten Giesemanns Arbeit [5] auf. Das VHDL-Modul kamera_top mit seinen untergeordneten Modulen kamera, ram_2k und monitor steuert die an der Erweiterungskarte angeschlossene Kamera an und gibt das empfangene Bild auf dem VGA-Ausgang der Erweiterungskarte als 640 × 480 Pixel großes 8 Bit Grauwertbild mit einer Bildwiederholfrequenz von 120 Hz aus.

Für den Framegrabber werden vier der Monitorsignale abgegriffen:

`PIXEL_DATA[7:0]` sind die Pixel-Datensignale `RPr_out[9:2]`, aus denen der Digital-Analog-Wandler auf der Erweiterungskarte später das analoge VGA-Rotsignal erzeugt. Da es sich um ein Grauwertbild handelt, spielt es keine Rolle, welches der drei Farbsignale verwendet wird. Sie haben immer den gleichen Wert.

PIXEL_CLK	ist der 50 MHz Pixeltakt. Es handelt sich dabei um den halbierten 100 MHz Systemtakt.
PIXEL_DATA_VLD	ist immer dann gesetzt, wenn Pixelwerte ausgegeben werden, die zum Bild gehören. Nicht zum Bild gehören beispielsweise die schwarzen Pixel, die während der vertikalen und horizontalen Bildsynchronisationsphasen übertragen werden.
NEW_IMG	ist das Steuersignal V_Sync_out für die vertikale Bildsynchronisation. Es zeigt an, dass ein neues Bild beginnt.

Im Modul userapp werden kontinuierlich jeweils vier 8 Bit Pixel zu einem 32 Bit Datenwort durch einen Automaten zusammengefasst. Ein Bild besteht damit aus $\frac{640*480}{4}$ =76800 Datenwörtern.

Findet kein Transfer statt (XFER_RUNNING = '0'), wird das Signal NEW_IMG benutzt, um den FIFO über den Reset-Eingang zu leeren, um sicherzustellen, dass sich keine Daten im FIFO befinden, wenn eine neue Übertragung begonnen wird.

Fordert der Gerätetreiber Daten, z.B. ein komplettes Bild, an, so setzt der PCI-Busmaster das Signal XFER_RUNNING und beginnt mit der Übertragung, sobald Daten im FIFO bereitstehen. Die Ablaufsteuerung im Modul userapp wartet ab, bis das nächste neue Bild begonnen hat und schreibt dann die zusammengefassten gültigen Pixeldaten in den FIFO, wenn das Signal BLOCK_FREE des FIFOs dies zulässt. Es werden so lange Daten in den FIFO geschrieben, bis der PCI-Busmaster das Signal XFER_RUNNING nach Abschluss des Transfers deaktiviert. Daher können auch mehrere Bilder mit einem DMA-Transfer abgerufen werden.

Der Datendurchsatz zwischen FIFO und PCI-Interface hängt entscheidend von der Verfügbarkeit des PCI-Busses ab. Es kann zwar nicht garantiert werden, dass immer ein vollständiges Bild übertragen wird, jedoch sind die Voraussetzungen dafür günstig:

- Durch das Zusammenfassen von jeweils vier Pixeln zu einem 32 Bit Datenwort werden die Pixeldaten mit einer Frequenz von nur noch $\frac{50MHz}{4}$ =12,5 MHz in den FIFO geschrieben.
- Da eine Bildzeile 160 Datenwörter umfasst, kann der FIFO 1½ Zeilen puffern.
- Nur ca. 78% einer Monitor-Bildzeile enthält gültige Pixel (640 von 816 Pixeln pro Zeile insgesamt, siehe Modul monitor). Bei einem Pixeltakt von 50 MHz bleibt damit eine Pause von 3,52 µs zwischen zwei Zeilen des Bildes. Diese Pause entspricht 116 PCI-Bustakten, in denen keine Daten in den FIFO eingespeist werden.

Abbildung 8.1 zeigt das Blockschaltbild des Framegrabbers. Links befindet sich das Kamera/Monitor-Interface, rechts der PCI-Busmaster mit vorgeschaltetem FIFO und in der Mitte der Block, in dem die Verknüpfung beider Teile geschieht.

8.2 Ein Gerätetreiber für den Framegrabber

8.2.1 Funktion init_module()

Die Funktion `init_module()` wird beim Laden des Treibers ausgeführt. Sie initialisiert die Hardware und den Treiber. Die wichtigsten Schritte werden im Folgenden erläutert.

Mit `register_chrdev()` wird der Treiber mit der Gerätedatei /dev/xilinx_pci verbunden und die folgenden Dateifunktionen angemeldet:

```
struct file_operations xilinx_pci_fops =
{
  open    : xilinx_pci_open,
  release : xilinx_pci_release,  /* close()-Systemaufruf */
  read    : xilinx_pci_read,
  ioctl   : xilinx_pci_ioctl     /* Konfigurationswiederherstellung */
};
```

Anschließend wird die PCI-Karte anhand der Hersteller-ID und der Geräte-ID gesucht und aktiviert. Rückgabewert der Suchfunktion `pci_find_device()` ist ein Zeiger auf eine Struktur vom Typ `pci_dev`, die das PCI-Gerät repräsentiert und für alle weiteren Funktionen benötigt wird.

Der Framegrabber hat eine Targetschnittstelle, der das PCI-BIOS oder das Betriebssystem beim Systemstart einen bestimmten Adressbereich zugewiesen hat. Damit der Treiber auf die Targetregister zugreifen kann, muss dieser Speicherbereich mit `ioremap()` eingeblendet werden.

Als nächstes wird das Interrupt Line Register des PCI Configuration Space Headers des Framegrabbers gelesen, um festzustellen, welcher IRQ zugewiesen wurde. Der Interrupthandler kann dann mit `request_irq()` angemeldet werden. Wichtig ist dabei, das Shared-Interrupt-Flag `SA_SHIRQ` zu setzen.

Da der Framegrabber immer nur einen DMA-Transfer zur gleichen Zeit durchführen kann, wird zum Schluss ein Semaphor (mutual exclusion) zur Zugangsregelung eingerichtet.

8.2.2 Funktion cleanup_module()

Die Routine `cleanup_module()` wird ausgeführt, wenn der Treiber entladen wird. Sie gibt die belegten Ressourcen wieder frei.

8.2.3 Dateifunktionen open() und release()

Die Dateifunktionen `open()` und `release()` beinhalten ein Makro, das mitzählt, wieviele Anwenderprogramme die Gerätedatei zur Zeit geöffnet haben. Das Kernelmodul kann nur dann entladen werden, wenn der Zähler den Wert 0 hat.

8.2.4 Ablauf eines DMA-Transfers

1. Ein Prozess, d.h. ein laufendes Anwenderprogramm, hat die Gerätedatei /dev/xilinx_pci geöffnet und startet einen `read()`-Systemaufruf zum Lesen eines Bildes aus /dev/xilinx_pci.

2. Der `read()`-Systemaufruf führt zum Aufruf der Funktion `xilinx_pci_read()` des Kerneltreibers, die zunächst mit dem Semaphor prüft, ob das Gerät verfügbar ist oder gerade den `read()`-Aufruf eines anderen Anwenderprozesses bearbeitet. Ist es frei, wird als nächstes versucht, einen ausreichend großen zusammenhängenden Bereich im physischen Speicher zu reservieren, in den der Framegrabber das Bild per DMA hineinschreiben kann. Bei Erfolg wird anschließend die Funktion `xilinx_pci_transfer()` aufgerufen.

3. Die Routine `xilinx_pci_transfer()` setzt die Parameter für den DMA-Transfer über die Targetschnittstelle des Framegrabbers (Adresse des DMA-Puffers, Anzahl der zu übertragenden Datenwörter) und gibt den Transfer-Startbefehl.

4. Zurück in der Funktion `xilinx_pci_read()` wird der Prozess in den Zustand „schlafend“ versetzt.

5. Der Framegrabber führt den Transfer des Bildes in den DMA-Puffer ohne Beteiligung des Prozessors durch und löst nach Abschluss einen Interrupt aus.

6. Der Interrupthandler `xilinx_pci_intr_handler()` wird aktiv. Er untersucht, ob der Framegrabber Ursache des Interrupts ist (Shared Interrupts bei PCI), bestätigt gegebenenfalls den Interrupt und weckt den Prozess wieder.

7. Die `xilinx_pci_read()`-Funktion wird fortgesetzt. War der Transfer erfolgreich, wird der Inhalt des DMA-Puffers (Kernelspace) in den vom Anwenderprogramm vorgesehenen Speicherbereich (Userspace) umkopiert. Ein Zeiger auf diesen Bereich wurde als Parameter beim `read()`-Systemaufruf vom Anwenderprogramm übergeben.

8. Der DMA-Puffer wird wieder freigegeben, das Semaphor inkrementiert und der Systemaufruf mit dem vorgeschriebenen Rückgabewert beendet.

Der Speicherbereich für den DMA-Puffer wird mit `__get_free_pages()` reserviert und als Streaming DMA-Einblendung verwendet.

8.3 Beispielprogramme

Alle Beispielprogramme arbeiten nach dem gleichen Prinzip. Sie öffnen die Gerätedatei /dev/xilinx_pci, führen einen `read()`-Befehl über ein komplettes Bild (307200 Bytes) aus und schließen die Gerätedatei wieder. Kommandozeilenparameter können mit `-?` oder mit `--help` abgefragt werden.

read_img_seq

Das Programm read_img_seq.c liest eine angegebene Anzahl von Bildern und speichert sie fortlaufend numeriert als Bitmaps (*.bmp) ab.

```
Aufruf: ./read_img_seq number prefix
    number gibt die Anzahl der zu lesenden Bilder an.
    prefix ist der Pfad und der Dateinamenanfang der Bilddateien.
```

framerate

Das Programm framerate.c fordert innerhalb einer angegebenen Zeitspanne kontinuierlich Bilder vom Framegrabber an und bestimmt so die Bildtransferrate.

```
Aufruf: ./framerate [interval]
    Der optionale Parameter interval legt das Messintervall
    in Sekunden fest. Der Standardwert ist 10 Sekunden.
```

camera_sdl

Das Programm camera_sdl.c öffnet unter X ein Fenster, das die fortlaufend vom Framegrabber angeforderten Bilder als Videobild darstellt. Ein Tastendruck schließt das Fenster wieder. Das Programm basiert auf den Videoausgabefunktionen der *Simple DirectMedia Layer (SDL)*[1] Bibliothek [15]. Die SDL stellt eine einheitliche, plattformunabhängige Schnittstelle zur Multimediaprogrammierung zur Verfügung. Sie entspricht ungefähr dem, was DirectX unter Microsoft Windows ist und eignet sich ideal für Multimediaanwendungen. Die verwendeten Funktionen sind im Quelltext kommentiert.

```
Aufruf: ./camera_sdl [-f] [-d] [-s] [-h]
    -f   Vollbildmodus
    -d   Doublebuffering
    -h   Bildspeicher im Videospeicher
    -s   Bildspeicher im Hauptspeicher
```

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

9 Experimente mit dem SDRAM-Baustein

Alle in diesem Kapitel zitierten Dateien befinden sich auf der beiliegenden CD im Verzeichnis /projekte/sdram-test/.

Bei der Auseinandersetzung mit den technischen Gegebenheiten des Spartan-II 200 PCI Development Boards entstand ein einfacher 100 MHz Controller für den SDRAM-Chip auf der Karte. Der Controller ist eingebettet in eine Testumgebung und kann als Ausgangspunkt für weitere Experimente und Entwicklungen verwendet werden.

9.1 Bedienung

Dieser Beispielentwurf ermöglicht es, Daten über die RS232-Schnittstelle des Entwicklungsboards im SDRAM zu speichern und aus dem SDRAM zu lesen. Die serielle Schnittstelle des Entwicklungsboards wird mit einem 1:1-Kabel mit der seriellen Schnittstelle eines Computers verbunden. Dort kann nun z.B. über ein Terminalprogramm mit dem Entwicklungsboard kommuniziert werden. Die Schnittstellenparameter sind: 9600 Baud, keine Parität, 8 Datenbits, 1 Stopbit, keine Flusssteuerung.

Nach der Konfiguration des FPGAs mit dem Beispielentwurf muss zunächst das SDRAM initialisiert werden. Im VHDL-Entwurf ist dafür eigentlich das Modul sdram_cfg vorgesehen. Zum Experimentieren wurde es jedoch durch die DIP-Schalter 4 bis 8 des Entwicklungsboards ersetzt. Sie geben gemäß Tabelle 9.1 die Burstlänge eines Speicherzugriffs und den Arbeitsmodus (sequentiell oder interleave) vor. Zu beachten ist, dass die Datenwortbreite des SDRAMs 32 Bit beträgt. Vier Bytes der seriellen Schnittstelle bilden ein SDRAM-Datenwort. Ein Druck des User-Tasters versetzt die gesamte Schaltung in den Anfangszustand (Reset) und initialisiert dann das SDRAM mit den vorgenommenen Einstellungen.

Ein Schreibbefehl beginnt mit dem ASCII-Zeichen 'W', gefolgt von einer sechsstelligen (Anfangs-) Adresse im Hexadezimalformat, wobei Groß- und Kleinbuchstaben gleichermaßen akzeptiert werden. Je nach gewählter Burstlänge müssen nun entsprechend viele Datenbytes folgen.

Ein Lesebefehl beginnt mit dem ASCII-Zeichen 'R', gefolgt von einer sechsstelligen (Anfangs-) Adresse. Das FPGA liest daraufhin das SDRAM ab der angegebenen Adresse aus und gibt die Daten über die serielle Schnittstelle an den Computer aus.

DIP4	DIP5	DIP6	DIP7	DIP8	Modus
on	x	x	x	x	sequentiell
off	x	x	x	x	interleave
x	on	on	on	off	Burstlänge 1 (4 Bytes)
x	on	on	off	on	Burstlänge 2 (8 Bytes)
x	on	off	on	on	Burstlänge 4 (16 Bytes)
x	off	on	on	on	Burstlänge 8 (32 Bytes)

Tabelle 9.1: Erlaubte SDRAM-Konfigurationseinstellungen

Der Gesamtadressraum des SDRAMs umfasst 2^{21} Adressen. Die obersten 3 Bits der übergebenen 24 Bit Adresse (6 Hexadezimalziffern) werden ignoriert. Die Adresse zerfällt in die Komponenten

Bit 20 - Bit 19 Bankadresse
Bit 18 - Bit 08 Zeilenadresse
Bit 07 - Bit 00 Spaltenadresse

Beispiel: Ist die Burstlänge auf 2 eingestellt, schreibt die Zeichenfolge “W01A78Cqwertzui“ die Datenwörter “qwer“ und “tzui“ in einem Bursttransfer an die Stellen 0x01A78C und 0x01A78D des SDRAMs. Mit “R01A78C“ werden die Speicherstellen wieder ausgelesen.

Zu Kontrollzwecken sind den Zuständen der Automaten der Ablaufsteuerung Nummern zugeordnet, die über die beiden 7-Segment-Anzeigen ausgegeben werden. Die drei wichtigsten Nummern sind:

00 Ruhezustand
01 Adresszustand. Das Zeichen ’R’ oder ’W’ wurde erkannt.
02 Datenzustand. Die Adresse wurde vollständig eingegeben.

Das Beispielprogramm sdram-test.c

Alternativ zum Terminalprogramm kann auch das beiliegende Programm sdram-test.c verwendet werden. Es schreibt eine zufällige Zeichenfolge an eine zufällig gewählte Adresse des SDRAMs, liest die Daten anschließend wieder aus und vergleicht sie mit der Originalzeichenfolge. Die Burstlänge wird als Kommandozeilenparameter übergeben.

9.2 Der Hardwareentwurf

Abbildung 9.1 zeigt die Struktur und die wesentlichen VHDL-Module des Testentwurfs. Die Kommunikation zwischen serieller Schnittstelle (links) und dem SDRAM-Controller (rechts) erfolgt über den Puffer RAMB4_S8_S32, der aus zwei „parallelgeschalteten“ Dual Port Block RAMs (Primitiv RAMB4_S4_S16 aus der Xilinx Entwurfsbibliothek [11]) aufgebaut ist. Automat 1 verbindet die serielle Schnittstelle mit dem 8 Bit Port des Puffers und ist für den Gesamtablauf verantwortlich. Automat 2 koordiniert das Zusammenspiel zwischen dem 32 Bit Port des Puffers und dem SDRAM-Controller.

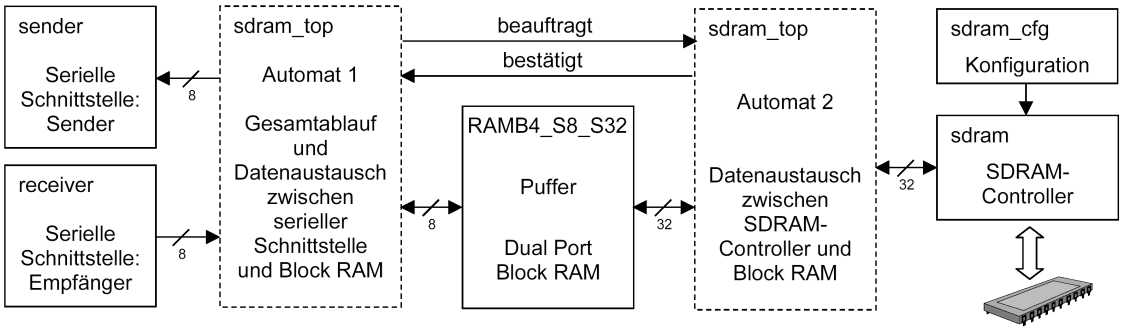

Abbildung 9.1: Struktur des SDRAM-Controller Testentwurfs

Der Ablauf ist streng sequentiell organisiert: Automat 1 nimmt das Befehlsbyte und die sechs durch das Decodermodul hexrom umgerechneten Adressbytes von der seriellen Schnittstelle entgegen. Im Falle eines Schreibtransfers werden die auf die Adressbytes folgenden Datenbytes im Puffer gespeichert. Anschließend beauftragt Automat 1 den Automaten 2, den Pufferinhalt über den SDRAM-Controller im SDRAM zu speichern. Im Falle eines Lesetransfers wird nach dem Empfang der Adressbytes zunächst Automat 2 beauftragt, die gewünschten Daten über den SDRAM-Controller aus dem SDRAM zu lesen und im Puffer zu speichern. Anschließend gibt Automat 1 den Pufferinhalt byteweise über die serielle Schnittstelle aus.

9.2.1 Der SDRAM-Controller sdram.vhd

Die Kommunikation mit dem SDRAM-Baustein wird über einen Automaten abgewickelt. Nach einem Reset durchläuft er zunächst eine Initialisierungsphase, in der der Arbeitsmodus des SDRAMs (Burstlänge, sequentiell/interleave) eingestellt wird und mit dem SDRAM-Befehl *Precharge All Banks* alle Bänke vorgeladen werden. Danach geht der Automat in den Ruhezustand über.

Der Automat verwendet symbolische Zustände, mit denen die einzelnen Phasen der Initialisierungssequenz, der Speicherauffrischung und eines Bursttransfers unterschieden werden. Oberste Priorität hat die Auffrischung des SDRAMs, die mit dem *Autorefresh*-Mechanismus des SDRAMs alle 64 ms (Konstante REFRESH_CNT_MAX) durchgeführt wird. Anderenfalls kann ein Lese- oder Schreibtransfer begonnen werden. Der Befehl *Row Activate* wählt dazu die Speicherbank und die Speicherzeile innerhalb der Bank aus. Der Befehl *Read With Autoprecharge* bzw. *Write With Autoprecharge* adressiert das gewünschte Datenwort innerhalb der Zeile und startet den Bursttransfer. Am Ende des Transfers werden die Leseverstärker wieder automatisch vorgeladen, und der Automat kehrt in den Ruhezustand zurück.

Der Controller ist für den wahlfreien Zugriff auf das SDRAM konzipiert. Daher wird jedes Mal *Row Activate* und *Precharge* durchgeführt, auch wenn der folgende Zugriff auf die gleiche Bank und Zeile zielt. Ein Vorgang ist nicht unterbrechbar und wird immer komplett abgearbeitet, bevor ein neuer Zugriff eingeleitet werden kann.

Einem SDRAM-Befehl entspricht eine bestimmte Belegung der Steuerleitungen RAS_N, CAS_N und WE_N, die deshalb zu einem 3 Bit Vektor zusammengefasst sind. Jedem Befehl ist außerdem eine Zeitkonstante zugeordnet, die die Anzahl der Takte angibt, die das SDRAM für den Befehl braucht. Beispielsweise muss nach einem *Read*-Befehl die CAS-Zugiffszeit (CAS-Latency) abgewartet werden, bis das erste Datenwort auf dem Datenbus anliegt. Bevor der Automat von einem symbolischen Zustand in den nächsten wechselt, wartet er die dem jeweiligen Befehl zugeordneten Wartetakte ab.

Schnittstelle zur Benutzeranwendung

CLK	ist der Systemtakt (100 MHz).
RST	ist der Reset-Eingang.
CFG[4:0]	ist der Konfigurationsbus, siehe VHDL-Modul sdram_cfg.
ADDR[20:0]	ist der Adressvektor.
WRDN	gibt die Transferrichtung an.
DATA_IN[31:0]	ist der Eingangsdatenbus.
DATA_OUT[31:0]	ist der Ausgangsdatenbus.
DATA_VLD	zeigt bei Lesezugriffen an, dass die Daten auf dem Ausgangsdatenbus gültig sind.
XFER_START	ist das Startsignal für einen Transfer. Wird ein Transfer begonnen, müssen ADDR und WRDN richtig gesetzt sein; bei einem Schreibtransfer wird außerdem das erste Datenwort auf DATA_IN übergeben. Die Eingangssignale müssen so lange stabil bleiben, bis der SDRAM-Controller den Transferbeginn mit XFER_STARTED bestätigt hat.

XFER_STARTED	zeigt an, dass ein Transfer eingeleitet wurde. Bei Burst-Schreibtransfers ist es zugleich das Signal dafür, dass neue Daten an DATA_IN bereitgestellt werden müssen: XFER_STARTED wird gesetzt, wenn *Row Activate* an das SDRAM geschickt wird. Nach Ablauf dieses Kommandos (2 Takte, siehe Konstante TME_ROW_ACTIVE) wird das erste Datenwort, das schon bei der Einleitung des Transfers übergeben wurde, an das SDRAM geschickt und zugleich das zweite Datenwort von DATA_IN in ein internes Register übernommen. Mit jedem folgenden Takt muss ein weiteres Datenwort an DATA_IN bereitgestellt werden.

9.2.2 Besonderheiten bei der Implementierung

Neben der funktionalen Beschreibung besteht die große Herausforderung dieses Entwurfs darin, alle Zeitbedingungen innerhalb des FPGAs und zwischen FPGA und SDRAM zu erfüllen. Da der SDRAM-Controller im FGPA und das SDRAM mit einer Taktfrequenz von 100 MHz betrieben werden, ist der Spielraum für Signalverzögerungen und -verzerrungen allgemein und im besonderen für die Taktsignale eng begrenzt.

Zur Reduktion der kombinatorischen Logik bei der Auswertung werden die Automatenzustände one-hot-codiert.

Taktverteilung

Zur Lösung des Taktsignalproblems enthalten FPGAs der Spartan-II Familie vier *Delay-Locked Loop* (DLL) Schaltkreise auf dem Chip, die die Ausbreitungsverzögerungen des Taktsignals im FPGA eliminieren und Verzerrungen zwischen den Taktsignalen minimieren.

In diesem Entwurf wird eine DLL (Primitiv CLKDLLHF aus der Xilinx Entwurfsbibliothek [11]), wie in Abbildung 9.2 dargestellt, verwendet. Das Taktsignal des Oszillators auf dem Entwicklungsboard wird über einen *Global Clock Input Buffer* (IBUFG) der DLL zugeführt. Die DLL generiert daraus zwei Taktsignale für die Logik im FPGA und ein Taktsignal für das externe SDRAM.

Die beiden internen Taktsignale werden über zwei *Global Clock Buffers* (BUFG) in die Taktverteilungsnetzwerke des FPGAs eingespeist. Die Taktverteilungsnetzwerke minimieren die durch unterschiedliche Lasten bedingten Taktverzerrungen. Mit dem Rückkopplungseingang CLKFB überwacht die DLL den Ausgangstakt und kann so Verzögerungen auf dem Verteilungsnetzwerk kompensieren. Das Taktsignal für das SDRAM ist um 180° phasenverschoben und direkt über einen *Output Buffer* (OBUF_F_12) mit dem Takteingang des SDRAMs verbunden.

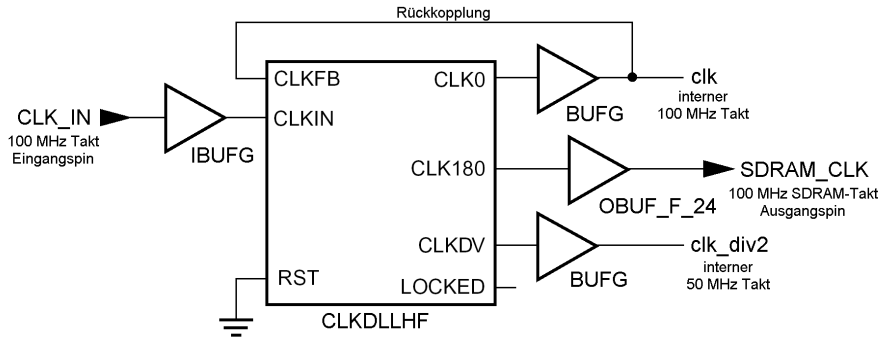

Abbildung 9.2: Verwendung einer DLL für die Taktsignale (sdram_top.vhd)

Der Testentwurf verwendet zwei Taktsignale: den 100 MHz Takt `clk` für die Komponenten, die in Abbildung 9.1 rechts vom Block RAM liegen, und den 50 MHz Takt `clk_div2` für die Komponenten, die links vom Block RAM liegen. Die DLL stellt dabei sicher, dass der halbierte Takt `clk_div2` mit dem Takt `clk` in Phase ist. Der halbierte Takt wird verwendet, um das Timing in den Modulen zu entspannen, die nicht mit 100 MHz betrieben werden brauchen.

Anschluss des SDRAMs

Alle Signale, die vom SDRAM kommen, werden im VHDL-Entwurf zunächst in Eingangsregister übernommen, bevor sie weiterverarbeitet werden. Alle Signale, die zum SDRAM gehen, werden mit Ausnahme des Taktsignals von Ausgangsregistern bereitgestellt. In der Implementierungsphase des synthetisierten Entwurfs werden diese Register auf die in den I/O-Blöcken vorhandenen D-Flipflops abgebildet, was die Analyse des Zeitverhaltens der Signale sehr vereinfacht.

Für die Signale zum SDRAM werden *Fast Slew Rate* Ausgangstreiber (OBUF_F_12) verwendet. Die Eingangstreiber werden in dem in der User Constraint Datei festgelegten *Nodelay Mode* betrieben.

Abschätzung des Zeitverhaltens zwischen FPGA und SDRAM

Der SDRAM-Takt wird von der DLL um 180° gegen den FPGA-Takt verschoben. Hinzu kommt noch die Verzögerung $t_{IOOP} = 2,9ns$ (propagation delay, OBUF_F_12 input to pad) durch den Ausgangstreiber.

Die übrigen mit sequentieller Logik implementierten FPGA-Ausgangssignale (Ausgangstreiber mit vorgeschaltetem D-Flipflop) haben eine Verzögerung von $t_{IOCKON} = 3,3ns$ (sequential delay, clock clk to valid data on pad).

Da die D-Flipflops in den I/O-Blöcken mit der steigenden Flanke des FPGA-Takts getriggert werden, fällt der Umschaltzeitpunkt der Signale an den SDRAM-Eingängen kurz hinter die fallende Flanke des SDRAM-Takts. Die von der SDRAM-Spezifikation geforderten Mindestzeiten $t_{S\_SDRAM} = 1,75ns$ (input setup time) und $t_{H\_SDRAM} = 1ns$ (input hold time), für die ein SDRAM-Eingangssignal vor bzw. nach der steigenden Taktflanke stabil sein muss, werden so bestmöglich eingehalten.

Messungen mit dem Oszilloskop bestätigen die Überlegungen, siehe Abbildung 9.3. Die obere Kurve ist das Signal am Takteingang des SDRAMs. Die untere Kurve zeigt das Chip Select Signal. Es ändert sich nur im Bereich der fallenden Flanke des Taktsignals.

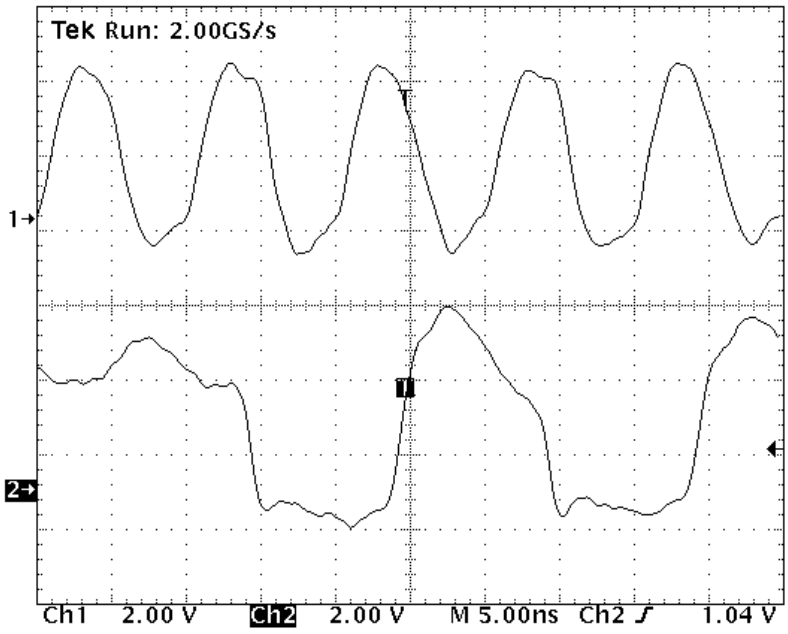

Abbildung 9.3: Messung des Taktsignals und des Chip Select Signals am SDRAM

Die Gesamtverzögerung des externen SDRAM-Takts zum internen FPGA-Takt beträgt $t_{FPGA\_to\_SDRAM} = 5ns + t_{IOOP} = 7,9ns$. Die steigende Flanke des FPGA-Takts, mit der die Eingangsflipflops des FPGAs die SDRAM-Ausgangssignale übernehmen, liegt damit $10ns - t_{FPGA\_to\_SDRAM} = 2,1ns$ hinter der steigenden Flanke des SDRAM-Takts. Eingehalten werden muss die Zeitbedingung $t_{IOPICK} = 1,7ns$ (input setup time, no-delay mode) der FPGA-Eingänge. Da das SDRAM die Gültigkeit seiner Ausgangssignale nach der steigenden Taktflanke für weitere $t_{OH\_SDRAM} = 3ns$ (data-out hold time) garantiert, wird auch diese Bedingung erfüllt.

Bemerkung 1: Nicht berücksichtigt wurden die nicht bekannten Verzögerungen durch die Leitungen zwischen FPGA und SDRAM auf dem Entwicklungsboard.

Bemerkung 2: Auf das Attribut *Nodelay* bei den Eingangstreibern des FPGAs kann auch verzichtet werden. Die Setup-Zeit vergrößert sich dadurch zwar auf $t_{IOPICK} = 3,9\,ns$, jedoch sind wegen $t_{AC\_SDRAM} = 6\,ns$ (access time from clock) die Daten an den SDRAM-Ausgängen schon $4\,ns$ vor der SDRAM-Taktflanke gültig.

Bemerkung 3: Auch für eine SDRAM-Taktverschiebung von 270° werden alle Zeitbedingungen eingehalten. Der interne FPGA-Takt und der externe SDRAM-Takt sind dann wegen $t_{FPGA\_to\_SDRAM} = 7,5\,ns + t_{IOOP} = 10,4\,ns$ nur um $0,4\,ns$ phasenverschoben. Der Toleranzbereich bei den SDRAM-Eingängen verkleinert sich; der Toleranzbereich für die FPGA-Eingänge wird größer.

10 Zusammenfassung und Ausblick

Die am Anfang gesetzten Ziele wurden erreicht. Die Entwicklungssoftware und das Entwicklungsboard konnten erfolgreich in Betrieb genommen und getestet werden. Allerdings nahm diese Phase aufgrund anfänglicher technischer Schwierigkeiten mehr Zeit in Anspruch als dafür eingeplant war.

Das aufwendigste der dabei entstandenen Testbeispiele ist ein einfacher 100 MHz SDRAM-Controller. Er wurde deshalb in dieser Arbeit vorgestellt.

Bei der Auseinandersetzung mit dem PCI-Bus entstand neben dieser schriftlichen Arbeit ein Tutorial [6] für das LogiCORE PCI Interface. Es gibt praktische Tipps und stellt insgesamt sechs typische Beispielentwürfe vor, die als Ausgangspunkt für eigene Anwendungen verwendet werden können. Zu jedem VHDL-Entwurf werden ausführlich erläuterte Gerätetreiber und Testprogramme für Linux geliefert.

Der für die Bildverarbeitung entworfene PCI-Busmaster verfügt über eine einfache, leicht modifizierbare Schnittstelle, die es erlaubt, im Prinzip beliebig viele Daten in den Hauptspeicher des Computers mit einem DMA-Tansfer zu übertragen. Der interne FIFO-Speicher schafft zeitliche Spielräume und kann bei Bedarf vergrößert werden, sofern das FPGA noch über freie Ressourcen verfügt. Durch die Verwendung der statischen Dual Port Block RAMs des FPGAs können die Daten asynchron zum PCI-Bustakt in den FIFO eingespeist werden.

In der Beispielanwendung Framegrabber wird der PCI-Busmaster schließlich erfolgreich eingesetzt und zugleich die Verbindung zu Carsten Giesemanns Diplomarbeit [5] hergestellt.

Inhalt der beiliegenden CD

/literatur/
: Dieser Ordner enthält die im Literaturverzeichnis angegebenen, in elektronischer Form vorliegenden Quellen.

/logicore_pci_interface/
: Dieser Ordner enthält das unveränderte LogiCORE PCI Interface Paket und die für das Spartan-II 200 PCI Development Board angepasste User Constraint Datei xc2s200fg456_32_33.ucf.

/projekte/
: Dieser Ordner enthält in seinen weiteren Unterverzeichnissen die in dieser Arbeit besprochenen Hardwareentwürfe und die zugehörige Software.

  /projekte/pci-busmaster/
  : Dieses Projekt ist der PCI-Busmaster aus Kapitel 6 (PCI-Interface mit vorgeschalteter FIFO), das als Grundlage für eigene auf dem Busmasterinterface aufbauende Entwürfe verwendet werden kann. Der eigene Entwurf beginnt im Modul userapp in der Datei userapp.vhd.

  /projekte/framegrabber/
  : Dieses Projekt ist der Framegrabber aus Kapitel 8, der auf dem PCI-Busmaster aus Kapitel 6 basiert.

  /projekte/sdram-test/
  : Dieses Projekt ist die SDRAM-Beispielanwendung aus Kapitel 9.

  Neben den VHDL-Quelltexten, der User Contraint Datei und dem LogiCORE PCI Interface enthält jeder Projektordner eine Projektdatei (*.npl) für den Xilinx ISE 5.1i Project Navigator und eine fertige Binärdatei (*.bit) zur Konfiguration des FPGAs. Die Gerätetreiber und die Testprogramme für Linux (Kernelversion 2.4) sind in C geschrieben. Beiliegende Makefiles helfen bei der Übersetzung.

/tutorial/
: Dieser Ordner enthält das in Kapitel 5 beschriebene Tutorial zum Entwurf von PCI-Interfaces. Startseite ist die Datei index.html.

Aus lizenzrechtlichen Gründen fehlen auf den CD-ROMs der öffentlich zugänglichen Exemplare der Diplomarbeit die Dateien des LogiCORE PCI Interface.

Literaturverzeichnis

[1] Hans-Peter Messmer: *PC-Hardwarebuch*,
Addison Wesley Longman, 5. Auflage 1998

[2] Tom Shanley / Don Anderson: *PCI System Architecture (PCI Revision 2.2)*,
Addison Wesley Longman (MindShare, Inc.), 4. Auflage 1999

[3] Tom Shanley / Don Anderson: *ISA System Architecture*,
Addison Wesley Longman (MindShare, Inc.), 3. Auflage 1995

[4] Alessandro Rubini / Jonathan Corbet: *Linux Device Drivers*,
O'Reilly, 2. Auflage 2001 (Linux Kernel 2.4),
auf der CD unter /literatur/linux_device_drivers_v2.pdf

[5] Carsten Giesemann: *Kamera- und VGA-Monitorerweiterung für das Spartan-II 200 PCI Development Board*,
interne Arbeitsnotizen, Veröffentlichung als Diplomarbeit geplant

[6] Markus Köchy: *Tutorial zum Entwurf von PCI-Interfaces*,
auf der CD unter /tutorial/index.html

[7] Memec Design: *Spartan-II 200 PCI Development Board User's Guide*,
auf der CD unter /literatur/pci_board_user_guide.pdf,
Anmerkungen unter /literatur/pci_board_user_guide_notes.txt

[8] Hynix / Samsung: *SDRAM Datenblätter und Funktionsbeschreibungen*,
zusammengefasst auf der CD unter /literatur/sdram/index.html

[9] Pulnix: *Pulnix TM-6710 Progressive Scan High-Speed Shutter Camera*,
auf der CD unter /literatur/pulnix_kamera.pdf

[10] Xilinx: *Spartan-II Datenblätter*,
zusammengefasst auf der CD unter /literatur/spartan/index.html

[11] Xilinx: *Xilinx Libraries Guide – ISE 5*,
auf der CD unter /literatur/xilinx_libraries_guide_ise5.pdf

[12] Xilinx: *PCI32 Virtex/Spartan Family Interface Data Sheet*,
auf der CD unter /literatur/logicore_pci_data_sheet.pdf

[13] Xilinx: *LogiCORE PCI Design Guide*,
auf der CD unter /literatur/logicore_pci_design_guide.pdf

[14] Xilinx: *LogiCORE PCI Implementation Guide*,
auf der CD unter /literatur/logicore_pci_implementation_guide.pdf

[15] *Simple DirectMedia Layer Version 1.2*,
auf der CD unter /literatur/sdl_documentation.pdf