Distributed Computing Device

Johannes Natter

BACHELORARBEIT

Nr. 0910306027-A

eingereicht am
Fachhochschul-Bachelorstudiengang

Hardware Software Design

in Hagenberg

im September 2012

Diese Arbeit entstand im Rahmen des Gegenstands

Seminar zur Bachelorarbeit

im

Sommersemester 2012

Betreuer:

FH-Prof. DI Dr. Markus Pfaff

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 8. Januar 2013

Johannes Natter

Inhaltsverzeichnis

Erklärung **iii**

Kurzfassung **vi**

Abstract **vii**

1 Einleitung **1**
1.1 Motivation . . . 2

2 Entwicklung **3**
2.1 Kommunikation . . . 3
2.1.1 Protokolle . . . 4
2.1.2 Arbeitsablauf . . . 5
2.2 Anforderungen . . . 5
2.2.1 Hardware . . . 5
2.2.2 Verwendung (Toolchain) . . . 5
2.3 Übersicht . . . 6

3 Hardware **8**
3.1 DE0-Nano . . . 8
3.2 Platinenerweiterung . . . 9
3.3 Prozessor . . . 9
3.3.1 Ext. Memory Interface . . . 10
3.3.2 GPIO-Modul . . . 12
3.3.3 SPI-Modul . . . 12
3.3.4 ALTREMOTE-Modul . . . 13
3.3.5 Zähler-Modul . . . 13
3.3.6 Spezialmodul . . . 13
3.3.7 Co-Prozessor-Modul . . . 14
3.3.8 SDRAM-Controller . . . 14

4 Software **20**
4.1 Benutzerprogramm und *BSL* . . . 20
4.1.1 BSL-Protokoll . . . 20

4.2 Speicheraufteilung 23
4.3 Bootkonzept 25
4.4 Verpackungsprogramm 27

5 Ergebnisse **29**
5.1 Entwicklungstand 29
5.1.1 Hardware 29
5.1.2 Software 29
5.1.3 Übersicht 30
5.2 Verwendbarkeit 30
5.3 Mögliche Erweiterungen 32
5.3.1 Hardware 32
5.3.2 Software 32

A SCARTS32 Pipeline-Erweiterung **33**

B ALTREMOTE-Modul **34**

Literaturverzeichnis **35**

Kurzfassung

Für die Berechnung komplexer Probleme in vielen Bereichen der Wissenschaft wird aktuell die Rechenleistung verschiedenster Computersysteme genutzt. Solche universell ausgelegten Systeme sind jedoch zumeist nicht für spezifischere Probleme angepasst.

Diese Arbeit behandelt die Entwicklung eines erweiterbaren Computersystems, welches mit Hilfe eines FPGA-(Field Programmable Gate Array) Bausteins einerseits die Möglichkeit bereitstellt, mit programmierbaren Hard- und Softwarekomponenten das System für eine Aufgabe zu spezialisieren, andererseits soll dieses Gemisch aus Hard- und Software durch ein Framework schnell und flexibel austauschbar sein.

Das Ergebnis dieser Arbeit ist ein Konzept, sowie die Hard- und Software des vielseitig einsetzbaren Computersystems. Die Aktualisierung und der Austausch des Systems über das Internet ist durchführbar. Die Hardwaremodule können jedoch derzeit nicht, wie in dieser Arbeit gefordert, per Software aktiviert werden, sondern benötigen das Aus- und Einschalten des Gerätes.

Abstract

Today the processing power of multiple computer systems is used to compute complex problems in various fields of science. Yet such systems can't be adjusted to more specified and individual problems.

This thesis treats the development of a flexible and extendable computer system, which makes it possible to adapt the system to the needs of a specific problem with the help of a FPGA (Field Programmable Gate Array) device and programmable hard- and software components. This compound of hard and software should also be exchangeable in a fast and flexible way by an appropriate framework.

The result of this thesis is the concept as well as the hard- and software of this versatile computer system. The update and alteration of the system via the Internet is feasible. However, the hardware modules cannot be activated by software at the moment, but need a power off-on of the device.

Kapitel 1

Einleitung

Die Verwendung der ungenutzten Rechenleistung vieler verteilter PCs bzw. deren Grafikprozessoren und sogar der Einsatz von Spielkonsolen unterstützt inzwischen einige rechenintensive Projekte in den unterschiedlichsten Bereichen der Forschung (Mathematik, Physik, Chemie, Medizin, Finanzwesen, usw...) mit kostenloser Rechenleistung. Das Lösen von Berechnungsaufgaben durch verteilte Computersysteme wird als verteiltes Rechnen bezeichnet[1]. Durch die bis zu mehreren hunderttausend vernetzten Computersysteme können diese Projekte auf eine Rechenkraft äquivalent aktueller Supercomputer zurückgreifen [11].

Im Rahmen dieser Bachelorarbeit soll ein netzwerkfähiges Gerät, im folgenden *DCD* (Distributed Computing Device) genannt, zur Unterstützung solcher Projekte im Internet entstehen. Dieses soll nicht an ein bestimmtes Projekt gebunden sein, sondern durch eine leicht austauschbare *Applikation* für unterschiedliche Projekte arbeiten können. Zu Beginn dieser Arbeit wird schrittweise ein Konzept (Kap. 2) für dieses Vorhaben erstellt und jeweils die Motivation der Entscheidungen dokumentiert. Kapitel 3 beschreibt ausführlich die ausgewählten Hardwaremodule. Der Aufbau der Software und die Verwendung der Toolchain werden in Kapitel 4 erläutert. Kapitel 5 untersucht die Verwendbarkeit des *DCD* anhand des Beispielprojektes *Bitcoin*, fasst die Eigenschaften des Gerätes zusammen und listet mögliche Erweiterungen der Hard- und Software des Computersystems auf.

[1] Verteiltes Rechnen, http://de.wikipedia.org/wiki/Verteiltes_Rechnen

1.1 Motivation

Des Systems des verteilten Rechnens bedienen sich mittlerweile viele Projekte im Internet. Einige davon sind in der *BOINC*[2] organisatorisch zusammengefasst. *BOINC* ist ein Framework für Projekte, welche verteiltes Rechnen nutzen möchten.

Das Projekt *Bitcoin*[3] bedient sich im Gegensatz dazu einer eigenen Infrastruktur. *Bitcoin* ist ein dezentral organisiertes, elektronisches Geld, welches aktuell (Stand August 2012) einen Wechselkurs von 8,49 Euro [4] hat.

Die Informationen einer getätigten Transaktion zwischen zwei Bitcoin Benutzern werden vom jeweiligen Sender an alle beteiligten Nutzer des dezentralen Netzwerkes gesendet. Diese sammeln alle empfangenen und im Netzwerk als unbestätigt geltenden Transaktionen in einer Liste, welche als Block bezeichnet wird. Ein beliebiger Benutzer kann seinen Block abschließen und somit die darin enthaltenen Transaktionen vom Netzwerk als bestätigt markieren lassen. Zum Abschließen eines Blockes muss der SHA256[4]-Hashwert des Blockheaders berechnet werden. Der Header enthält unter anderem einen Zeitstempel, den für das Abschließen des Blockes verantwortlichen Benutzer und einen 32 Bit langen, variablen Wert (Nonce). Anschließen wird der SHA256-Hashwert des erhaltenen Hashwertes vom Blockheader berechnet. Ist der Wert dieses Ergebnisses nun kleiner-gleich als der eines, vom Netzwerk bestimmten, Zielwertes, so ist der Block erfolgreich abgeschlossen. Ist der Wert des Ergebnisses größer, muss der Benutzer den variablen Wert des Blockheaders um eins erhöhen und den SHA256 Hashwert erneut zweimal berechnen. Das Lösen dieses Problems kann, abhängig vom rechnenden Computersystem, mehrere Minuten benötigen. Ob ein Block gültig ist, kann jedoch vom Netzwerk schnell überprüft werden[5]. Für das Abschließen eine Blockes erhält der verantwortliche Benutzer von Netzwerk als Anerkennung Bitcoins, da die Benutzer des Bicoin-Netzwerkes dadurch die getätigten Transaktionen nachvollziehen können.

Die Berechnung der Bitcoin-Hashes dient als exemplarische Aufgabe zum Einsatz des *DCD*. Unabhängig von der vorhandenen Infrastruktur und den verwendeten Protokollen zur Kommunikation soll das *DCD* auch für andere Projekte arbeiten können.

[2] Berkeley Open Infrastructure for Network Computing, http://de.wikipedia.org/wiki/Berkeley_Open_Infrastructure_for_Network_Computing

[3] Bitcoin, http://de.wikipedia.org/wiki/Bitcoin

[4] Secure Hash Algorithm, http://de.wikipedia.org/wiki/SHA256

[5] Einwegfunktion, http://de.wikipedia.org/wiki/Einwegfunktion

Kapitel 2

Entwicklung

Dieses Kapitel widmet sich den gestellten Anforderungen an das *DCD*. Es wird hierzu aus Sicht des Benutzers vorgegangen, also die Verwendung des *DCD* in den Vordergrund gestellt. Zusätzlich enthält dieses Kapitel die Auswahl der Hardwarekomponenten. Die ausführliche Beschreibung der gewählten Hardwarekomponenten erfolgt in Kapitel 3.

2.1 Kommunikation

Zu Beginn stellt sich die Frage, welche Parteien an der Kommunikation beteiligt sind und was übertragen werden soll. Das *DCD* soll die Arbeitspakete eines Projektservers empfangen, diese verarbeiten und die Ergebnisse an den Projektserver zur Validierung zurück senden. Diese Aufgabe wird von einem Programm am *DCD*, im folgenden als *Applikation* bezeichnet, bearbeitet.

Zusätzlich kann der Benutzer jederzeit die aktuelle *Applikation* durch eine andere ersetzen. Dies ist die Aufgabe eines weiteren Programmes am *DCD*, welches in diesem Dokument *BSL* (*bootstrap loader*) genannt wird. Bevor das *DCD* eine andere *Applikation* empfangen kann, muss der *BSL* gestartet werden. Dieser kann von einem Benutzerprogramm per Internet eine neue *Applikation* empfangen, am *DCD* speichern und anschließend starten.

Die *Applikation* kann nun mit dem selben Projektserver kommunizieren oder auch mit einem anderen und für ein anderes Projekt rechnen. Die Standorte aller beteiligten Parteien können unterschiedlich sein. Abbildung 2.1 zeigt die Kommunikationspartner.

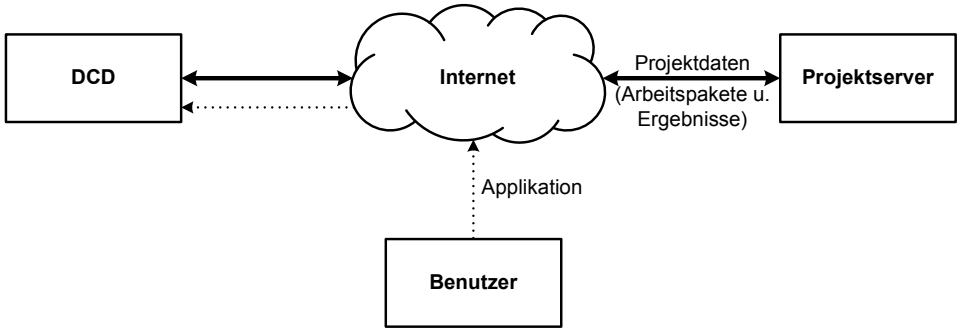

Abbildung 2.1: Kommunikationspartner

2.1.1 Protokolle

Die Übertragung der Daten aller Kommunikationspartner erfolgt über das Internet. *DCD* und Benutzerprogramm benötigen daher einen entsprechenden Protokollstack. Für die Übertragung der *Applikation* steht ein, vom Autor dieses Dokumentes spezifiziertes, Protokoll auf Ethernetebene zur Verfügung. Das Benutzerprogramm ist ebenfalls vorhanden. Kapitel 4 beschreibt Benutzerprogramm und Protokoll im Detail. Für die Kommunikation über das Internet muss das Benutzerprogramm erweitert und das Internetprotokoll mit Transportschicht in den Protokollstack integriert werden. Als Transportschicht wird das *User Datagram Protocol (UDP)* gewählt, da es eine geeignete einfache Datenüberprüfung besitzt und weniger Resourcen im *DCD* beansprucht als die Implementierung des *Transmission Control Protocol's (TCP)*. Die Erweiterung ist in Abbildung 2.2 dargestellt. Die verwendeten Protokolle zwischen Projektserver und *DCD* sind projektspezifisch.

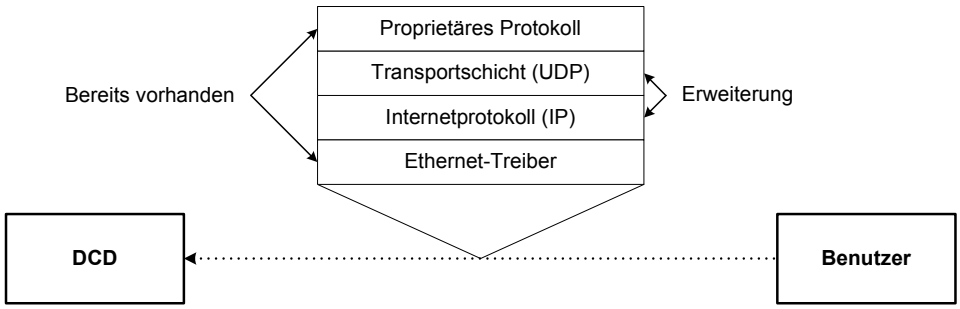

Abbildung 2.2: Erweiterung des Benutzerprogramms und dessen Protokollstacks

2.1.2 Arbeitsablauf

Die *Applikation* wird vom Benutzer an das *DCD* gesendet und aktiviert. Sobald diese startet, wird eine Verbindung mit dem Projektserver aufgebaut und ein Arbeitspaket angefordert. Nach erfolgter Verarbeitung der Daten, schickt das *DCD* die Ergebnisse zur Beurteilung an den Server zurück und fordert sogleich neue Aufgaben an. Alle Ergebnisse werden vom Projektserver auf Richtigkeit überprüft und die Resultate an das *DCD* gesendet.

Eventuelle Statusmeldungen, Statistiken und Arbeitsprotokolle können applikationsspezifisch vorhanden sein. Dies soll jedoch nicht Teil dieser Arbeit sein.

2.2 Anforderungen

2.2.1 Hardware

Folgende Punkte sind aus Hardwaresicht vom *DCD* zu erfüllen:

- Möglichst Kompakt,
- leicht Erweiterbar,
- arbeitet autonom

Zur Erfüllung dieser Aufgaben wird eine Kombination aus Prozessor und FPGA verwendet. Hierbei kann zum einen die gesamte Netzwerkkommunikation komfortabel in C implementiert werden, zum anderen können rechenintensive und zeitkritische Aufgaben vom FPGA übernommen werden. Aus Sicht des Prozessors sind diese unterstützenden Hardwaremodule memory-mapped als *Spezialregister* verfügbar.

Um das Rechengerät kompakt zu halten, wird ein Soft-Core Prozessor verwendet. Hierbei handelt es sich um einen Prozessor, welcher in einem FPGA-Design integrierbar ist. Somit kann das gesamte Framework inklusive *BSL* und *Applikation* auf einem Chip bzw. Entwicklungsboard ausgeführt werden.

Als Entwicklungsboard wird das *DE0-Nano Development and Education Board*[1] der Firmwa Terasic verwendet.

2.2.2 Verwendung (Toolchain)

- Die *Applikation* besteht aus einer Datei am PC und beinhaltet alle, für die Erfüllung der Aufgaben erforderlichen, Programme und Hardwaremodule
- Die *Applikation* kann per Doppelklick auf die Datei, oder per Konsole an das Gerät gesendet und aktiviert werden

[1] DE0-Nano Board, http://www.terasic.com.tw/cgi-bin/page/archive.pl?No=593

2.3 Übersicht

Abbildung 2.3 zeigt den Aufbau des *DCD's*. Das Cyclone-IV-FPGA, der SDRAM und der Konfigurationsspeicher des DE0-Nano Board's werden verwendet. Erweitert wird dieses durch einen Flash Speicher und einen Netzwerk-Controller. Der Flash Speicher enthält den *BSL* und die *Applikation*, welche beim Start in den SDRAM kopiert werden. Im Konfigurationsspeicher ist das FPGA-Design gespeichert. Dieses beinhaltet den Soft-Core-Prozessor mit SDRAM-Controller und alle Hardwaremodule inklusive applikationsspezifischem Spezialmodul und Co-Prozessor Modul, falls diese für die *Applikation* benötigt werden.

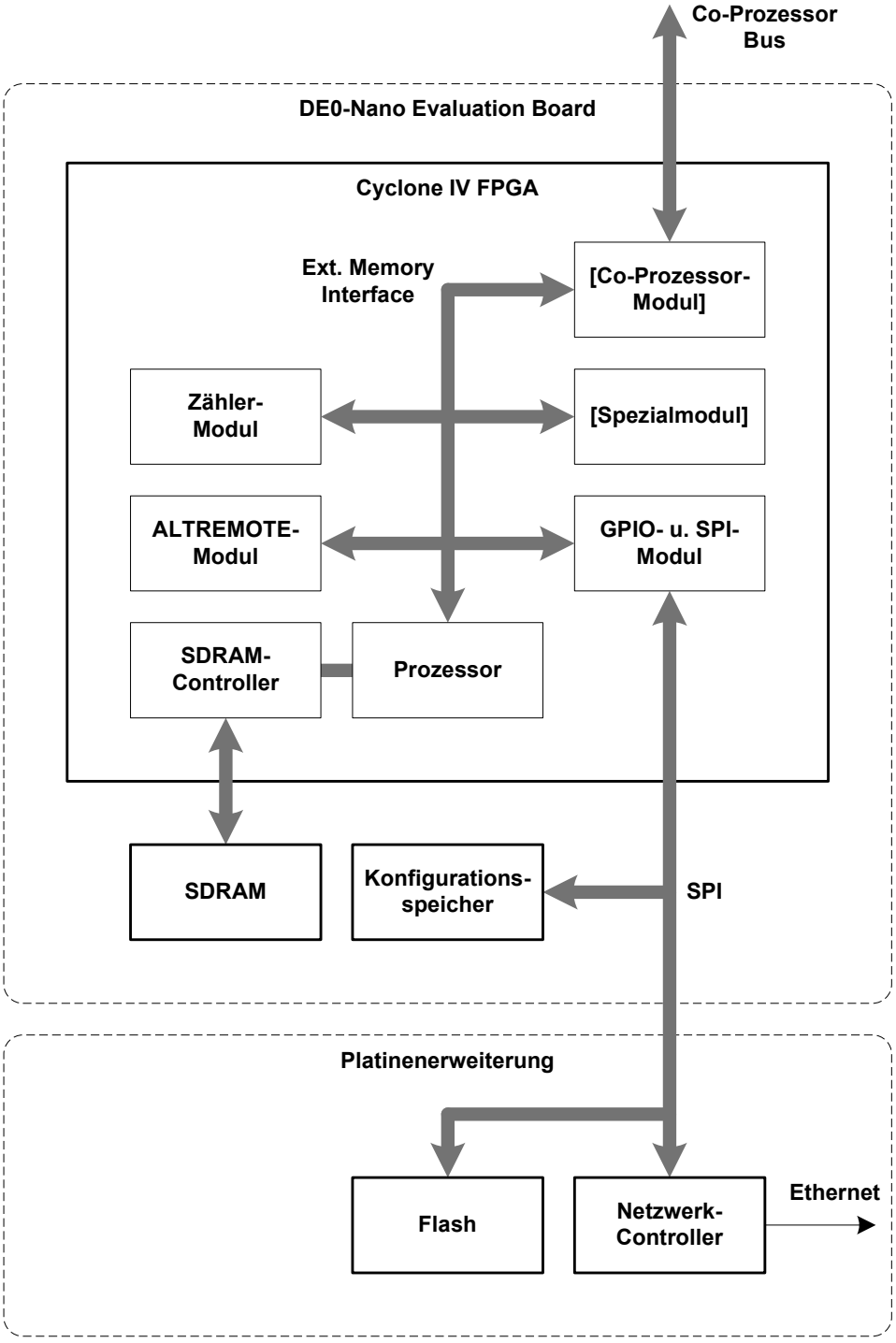

Abbildung 2.3: *DCD*-Übersicht

Kapitel 3

Hardware

Dieses Kapitel beschreibt die, in Abbildung 2.3 gezeigten, Hardwarekomponenten.

3.1 DE0-Nano

Das Entwicklungsboard von Terasic[1] hat eine Abmessung von 49x79.2mm und zeichnet sich durch die geringe Anzahl an Bauelementen aus. Für das *DCD* werden das FPGA, der SDRAM und der Konfigurationsspeicher benötigt.

Das Cyclone-IV-FPGA von Altera besitzt 22320 Logikelemente [5] und wird am DE0-Nano Board mit einem 50MHz-Taktgeber versorgt [6]. 594 Kbits stehen als interner Speicher zur Verfügung [5].

Der Konfigurationsspeicher (EPCS16) von Altera ist ein Flash Speicher mit SPI. Der Speicher ist unterteilt in 32 Sektoren zu jeweils 256 Pages [2]. Eine Page enthält 256 Bytes [2]. Der Baustein kann somit 2 Mbytes (16 Mbits) speichern. Die Verfügbaren SPI-Kommandos sind detailliert in [2] beschrieben. Der EPCS16 benutzt *active serial (AS) configuration* [1] für die Konfiguration der Cyclone- IV-FPGA's [2]. Bei *active serial (AS) configuration* liest das FPGA als Master das Design vom Konfigurationsspeicher. Die Steuerleitungen zwischen EPCS16 und FPGA können nach beendeter Konfiguration als *user IO's* benutzt werden[2].

Der SDRAM (IS42S16160B) von ISSI hat eine Speicherkapazität von 32 Mbytes und ist unterteilt in 4 Bänke zu je 8192 Reihen [12]. Eine Reihe enthält 512 Spalten mit einer Größe von jeweils 2 Byte [12].

[1] DE0-Nano, http://www.terasic.com.tw/cgi-bin/page/archive.pl?No=593
[2] Dual-Purpose Pins, http://www.alteraforum.com/forum/

3.2 Platinenerweiterung

Für die Kommunikation über das Internet ist eine Erweiterung des DE0-Nano Board's erforderlich. Als Netzwerk-Controller wird der ENC28J60 von Microchip verwendet. Dieser kann per SPI konfiguriert und gesteuert werden, siehe [8].

Der SPI-Flash (A25LQ023) von AMIC ist organisiert in 64 Blöcke zu je 16 Sektoren [3]. Ein Sektor enthält 16 Pages mit jeweils 256 Bytes [3]. Es können daher 4 Mbytes gespeichert werden.

3.3 Prozessor

Als CPU verwendet das *DCD* den *SCARTS* Soft-Core-Prozessor der Technischen Universität Wien[3], welcher unter der GNU LGPL[4] frei auf OpenCores[5] erhältlich ist. Die Anforderung an den Prozessor, der Aufbau und ein Vergleich mit anderen Soft-Core Prozessoren kann in [10] nachgeschlagen werden. Für diese Arbeit wurde der *SCARTS* verwendet, da der Quellcode frei zugänglich und änderbar ist. Somit kann der Prozessor, falls erforderlich, jederzeit angepasst werden, um die Anforderungen des *DCD's* zu erfüllen.

Der *SCARTS* benutzt für das zu exekutierende Programm und dessen Daten getrennte Speicher ⇒ Harvard-Architektur. Die Größe dieser Speicher kann bei der Instanziierung des Prozessors festgelegt werden. Realisiert sind diese als RAM/ROM Module im FPGA. Der Programmspeicher besteht aus zwei Modulen. Im ersten Speichermodul, in [10, 16] als *boot memory* bezeichnet, ist das Boot-Programm abgelegt. Dieses wird nach Hardware-Reset ausgeführt und beginnt ab Adresse 4000 0000h [16, Kap. 2.2.2]. Das zweite Speichermodul, in [10, 16] *instruction memory* genannt, beinhaltet das Hauptprogramm beginnend ab Adresse 0000 0000h [16, Kap. 2.2.2]. Da das DE0-Nano Board über einen SDRAM Baustein verfügt, wird das *SCARTS*-interne Modul für das *instruction memory* durch einen SDRAM-Controller ersetzt. Das Hauptprogramm befindet sich somit zur Laufzeit im SDRAM und dessen Daten im FPGA. Das Boot-Programm kopiert den *BSL* und die *Applikation*, welche im Flash Speicher abgelegt sind, in den SDRAM und startet eines dieser Programme (siehe Abschnitt 4.3).

Der Datenpfad des Prozessors ist konfigurierbar entweder 16 oder 32 Bit groß. Verwendet wird die 32-Bit-Konfiguration. Näheres hierzu in [10, Kap. 5.2].

Für die Fehlersuche stellt die Toolchain des *SCARTS*-Prozessors einen Protokollinterpreter des *gdb Remote Serial Protocol*'s[6] zur Verfügung [13],

[3] SCARTS Homepage, http://ti.tuwien.ac.at/ecs/teaching/courses/hwsw_lu_WS2011

[4] GNU Lesser General Public License, http://www.gnu.org/licenses/lgpl.html

[5] *SCARTS* OpenCores Projekt, http://opencores.org/project,scarts

[6] gdb Remote Serial Protocol,
http://sourceware.org/gdb/onlinedocs/gdb/Remote-Protocol.html

welcher mittels prozessor-internem UART Modul [15] den aktuellen Zustand des laufenden Programms an den GDB senden kann.

3.3.1 Ext. Memory Interface

Die CPU kann durch externe Hardwaremodule erweitert werden. Tabelle 3.1 zeigt die zur Verfügung gestellten Signale.

Signal	Bit Breite	Beschreibung
i_data	32	Dateneingang
o_data	32	Datenausgang
o_extsel	1	*SCARTS* Anforderung an externe Module
o_write_en	1	Write enable
o_byte_en	4	Byte enable
o_addr(14 downto 5)	10	Modul Adresse
o_addr(4 downto 0)	5	Modul interne Adresse. 5 Bits ⇒ 32 Bytes Adressraum pro Modul

Tabelle 3.1: Signale des ext. Memory IF

In Abbildung 3.1 ist die Schnittstelle zwischen *SCARTS*-Prozessor und den externen Hardwaremodulen dargestellt.

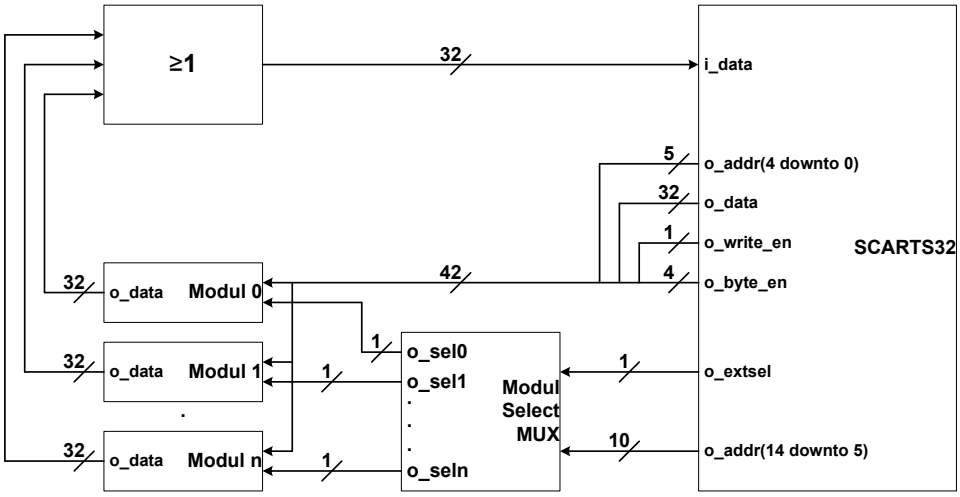

Abbildung 3.1: Ext. Memory Interface

Der Adressraum für externe Module beginnt ab Adresse FFFF8000h. Werden Daten ab dieser Adresse angesprochen, so setzt der *SCARTS* das Signal o_extsel. Durch o_addr(14 downto 5) kann ein konkretes Modul selektiert werden. Die Einheit *Modul Select MUX* legt die Basisadressen aller

externen Module fest. Die Adresse in o_addr(4 downto 0) ist allen Modulen zugänglich und adressiert eines von 32 Bytes. Ist ein Modul deselektiert, so muss dieses am Datenausgang 00000000h ausgeben. Dadurch können die Datenausgänge aller Module über ein Oder-Gatter in den Dateneingang des *SCARTS* geführt werden.

Die Adressen 0 und 2 aller Module sind reserviert entsprechend [16, Kap. 2.3.1] bzw. [10, Kap. 5.6]. Die verbleibenden Adressen können als Daten-/Steuerregister genutzt werden.

Abbildung 3.2 zeigt die Basisadressen der, in Abbildung 2.3 dargestellten, Module. Am Beginn des Adressraumes von Adresse 0000h bis 7FFFh ist der Datenspeicher des *SCARTS*-Prozessors, welcher im FPGA als RAM Modul realisiert ist, gemapped.

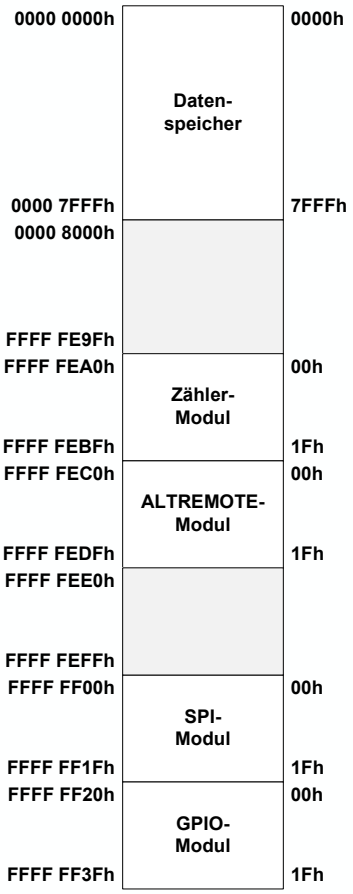

Abbildung 3.2: Memory Map des Adressraumes

3.3.2 GPIO-Modul

Dieses Modul wird für die Generierung der Chipselect-Signale des SPI verwendet. Zusätzlich können die LED's des DE0-Nano Board's zur Fehlersuche hinzugefügt werden. Tabelle 3.2 zeigt die Registerbelegung des GPIO Modul's.

Modul interne Adresse [Bytes]	3	2	1	0
00h	-	Reserviert	-	Reserviert
04h	-	-	-	GPIO_REG_GPO

Tabelle 3.2: Registerbelegung des GPIO-Modul's

Bit	Beschreibung
0	Chipselect des Netzwerk-Controllers
1	Chipselect des Flash Speichers
2	Chipselect des Konfigurationsspeichers

Tabelle 3.3: Belegung von *GPIO_REG_GPO*

3.3.3 SPI-Modul

Das *DCD* benötigt zur Kommunikation mit Konfigurationsspeicher, Flash Speicher und Netzwerk-Controller ein SPI-Master-Modul. Alle SPI-Slaves können im Modus[7] CPOL=0, CPHA=0 angesprochen werden, siehe [2, 3, 8]. Der Modus kann in diesem Modul daher nicht geändert werden. Ein Schreibzugriff auf *SPI_REG_WDATA* startet den SPI-Transfer eines Bytes. Die Daten in *SPI_REG_WDATA* werden beginnend mit dem MSB an MOSI ausgegeben. Während des Transfers ist Bit 0 in Register *SPI_REG_STAT* gesetzt (Busy Bit). Die seriellen Daten vom SPI Slave werden beginnend mit dem MSB in *SPI_REG_RDATA* abgelegt. Die gelesenen SPI-Daten können verwendet oder ignoriert werden.

Ist Bit 0 in Register *SPI_REG_CTRL* gesetzt, so werden die Daten beginnend mit dem LSB gesendet bzw. abgelegt. Abschnitt *Programming and Configuration File Support* in [2] beschreibt die Notwendigkeit für diesen Modus.

Die Taktrate des SPI ist 12.5MHz. Der Versorgungstakt von 50MHz des DE0-Nano Board's wird mit einem entsprechenden Teiler herabgesetzt. Der Netzwerk-Controller benötigt eine SPI-Taktfrequenz größer 8MHz siehe [9] und höchstens 20MHz [8].

Tabelle 3.4 zeigt die Registerbelegung des SPI-Modul's.

[7] Serial Peripheral Interface, http://de.wikipedia.org/wiki/Serial_Peripheral_Interface

Modul interne Adresse [Bytes]	3	2	1	0
00h	SPI_REG_CTRL	Reserviert	SPI_REG_STAT	Reserviert
04h	-	-	SPI_REG_RDATA	SPI_REG_WDATA

Tabelle 3.4: Registerbelegung des SPI-Modul's

3.3.4 ALTREMOTE-Modul

Die *Applikation* eines Projektes beinhaltet neben dem auszuführenden Programm ein FPGA- Design mit *SCARTS*-Prozessor und falls erforderlich Spezial- und Co-Prozessor- Modulen. Beim Austausch der *Applikation* wird somit der Soft-Core-Prozessor gewechselt und im Konfigurationsspeicher abgelegt. Der *BSL* muss nach Empfang der *Applikation* durch das Benutzerprogramm eine Rekonfiguration des FPGA's auslösen, damit nach Start der *Applikation* eventuelle Spezial/Co-Prozessor-Module zur Verfügung stehen.

Hierfür wird das ALTREMOTE-Modul verwendet, welches die *Remote System Upgrade (ALTREMOTE_UPDATE) Megafunction* von Altera instantiiert und laut [14] ansteuert. Zum Auslösen der Rekonfiguration des FPGA's wird Bit 0 in *AR_REG_CTRL* gesetzt.

Modul interne Adresse [Bytes]	3	2	1	0
00h	AR_REG_CTRL	Reserviert	-	Reserviert

Tabelle 3.5: Registerbelegung des ALTREMOTE- Modul's

3.3.5 Zähler-Modul

Der DHCP Client von Microchip[8] benötigt für die Behandlung von Timeout-Ereignissen ein Zeit- bzw. Zähler-Modul. Hierfür wurde ein 16-Bit-Zähler implementiert. Dieser wird zyklisch im 250-μs-Raster inkrementiert und ist als Register *CNT_REG_TICK250US* im Zähler-Modul verfügbar. Der DHCP-Client bildet mittels *CNT_REG_TICK250US* eine Differenz in $[TICKS]$ und vergleicht diese mit der internen Konstante TICKS_PER_MSEC. Diese Konstante muss den Wert 4 enthalten.

Das Zähler-Modul wird von einem 250 μs Strobe-Generator versorgt, welcher im *SCARTS*-Prozessor instantiiert ist.

3.3.6 Spezialmodul

Dieses Modul kann im FPGA-Design der *Applikation* vorhanden sein, ist jedoch nicht zwingend erforderlich. Als Spezialmodul werden jene Hardware-

[8] Microchip TCP/IP Stack, http://www.microchip.com/

Modul interne Adresse [Bytes]	3	2	1	0
00h	-	Reserviert	-	Reserviert
04h	-	-	CNT_REG_TICK250US	

Tabelle 3.6: Registerbelegung des Zähler Moduls

Module bezeichnet, welche die Berechnung einer Projektaufgabe unterstützen. Im Falle von Bitcoin könnte dies ein SHA256-Modul sein.

3.3.7 Co-Prozessor-Modul

Das *DCD* kann für die Berechnung einer Projektaufgabe die Rechenkraft weiterer Prozessoren oder FPGA's nutzen. Zur Kommunikation mit solchen Berechnungseinheiten ist ein Hardware-Modul im FPGA-Design der *Applikation* notwendig. Dieses wird als Co-Prozessor-Modul bezeichnet und ist ebenfalls nicht zwingend erforderlich.

3.3.8 SDRAM-Controller

Der Datenspeicher des *SCARTS*-Prozessors ist als RAM-Modul im FPGA realisiert. Befindet sich der Programmspeicher ebenfalls im RAM des FPGA's, so teilen sich diese beiden Speicher die zur Verfügung stehenden RAM-Zellen. Damit die RAM-Zellen im FPGA größtenteils für den Datenspeicher verwendet werden können, muss das *instruction memory* in den externen SDRAM ausgelagert werden. Das *boot memory* des Programmspeichers und das darin gespeicherte Boot-Programm liegen weiterhin im FPGA.

Hierfür wird ein SDRAM-Controller implementiert. Abbildung 3.3 zeigt dessen Schnittstellen zwischen *SCARTS*-Prozessor und SDRAM.

nReset ist low-aktiv und setzt die FSM[9] des SDRAM-Controllers zurück. Das Signal *clk* versorgt den Controller mit dem internen Takt des *SCARTS*-Prozessors.

Die Initialisierungphase des SDRAM-Speichers beinhaltet eine Wartezeit von mindestens 200 μs [12]. Hierfür wird das 250-μs-Strobesignal *strobe250us* verwendet. Die Wartezeit bei der Initialisierung des SDRAM's beträgt daher 250 μs.

Die Signale *wdata*, *waddr*, *wen*, *raddr*, *rdata* bilden das Speicher-Interface für den *SCARTS*-Prozessor. Da die Adressen *raddr* und *waddr* den gleichen Speicher adressieren und das Interface zum SDRAM maximal einen Schreib-/Lesebefehl pro Takt verarbeiten kann, muss der Zugriff arbitriert werden. Ist *wen* aktiv, so führt der SDRAM-Controller den Schreibvorgang vorrangig aus. Dabei wird das Signal *hold* gesetzt und der *SCARTS*-Prozessor angehalten, bis der Schreibvorgang abgeschlossen ist.

[9] Endlicher Automat, http://de.wikipedia.org/wiki/Finite_State_Machine

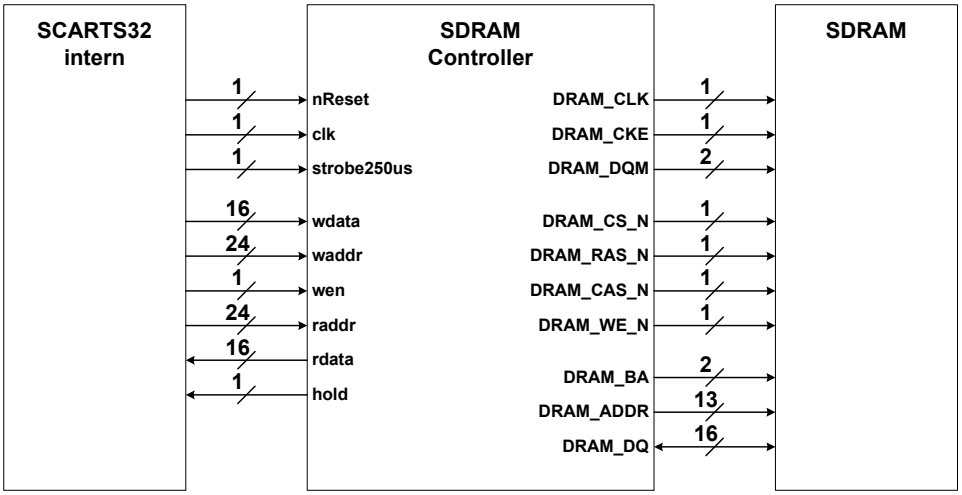

Abbildung 3.3: SDRAM-Controller-Interface

Der adressierbare Speicherbereich ist begrenzt mit der Größe des SDRAM-Speichers. Bei 32 Mbytes (2^{25}) bzw. 2^{24} Spalten zu je 16 Bit werden aus Sicht des Prozessors 24 Adressleitungen benötigt (*raddr*/*waddr*). Eine Instruktion benötigt 16 Bit zur Speicherung (*rdata*/*wdata*) siehe [10, Kap. 3.4].

Die Signale mit den Prefix *DRAM_* versorgen alle Leitungen für die Kommunikation zum SDRAM. *CKE* (Clock Enable [12]) ist stets aktiv. Die zwei Leitungen von *DQM* (Output Enable [12]) sind ebenfalls stets aktiv.

Die Signale *CS_N*, *RAS_N*, *CAS_N* und *WE_N* bilden eines von 13 möglichen Kommandos, wie z.B. Daten lesen/schreiben, selektieren/abwählen einer Reihe und Self-Refresh [12]. *BA* selektiert eine von 4 Bänken [12]. Mit *ADDR* kann eine Reihe bzw. eine Spalte adressiert werden [12]. Eine Spalte des SDRAM's speichert eine *SCARTS*-Instruktion und kann mit *DQ* beschrieben bzw. gelesen werden. Beim Wechsel einer Reihe, können keine Speicherdaten transportiert werden. Der SDRAM-Controller hält den *SCARTS*-Prozessor daher mit *hold* an.

Der SDRAM benötigt in 64 ms 8192 Refreshzyklen bzw. einen Zyklus pro 7.8125 μs. Bedingt durch die Implementierung des SDRAM-Controllers wird ein Zyklus im 6 μs Raster ausgeführt. Während eines Refreshzyklus' hält der Controller den *SCARTS*-Prozessor mit *hold* an.

Als Taktversorgung für den SDRAM dient das Signal *DRAM_CLK*. Dieses ist nicht gleichzusetzten mit der Taktversorgung des SDRAM-Controllers (*clk*), sondern bedarf einer besonderen Aufmerksamkeit und wird im folgenden behandelt.

Taktversorgung des SDRAM's

Das *instruction memory* Modul soll ohne Änderung am *SCARTS*-Prozessor durch den SDRAM-Controller ersetzt werden. Daher ergeben sich seitens des Speicher Interfaces des *SCARTS* bestimmte Anforderungen. Abbildung 3.4 stellt diese dar.

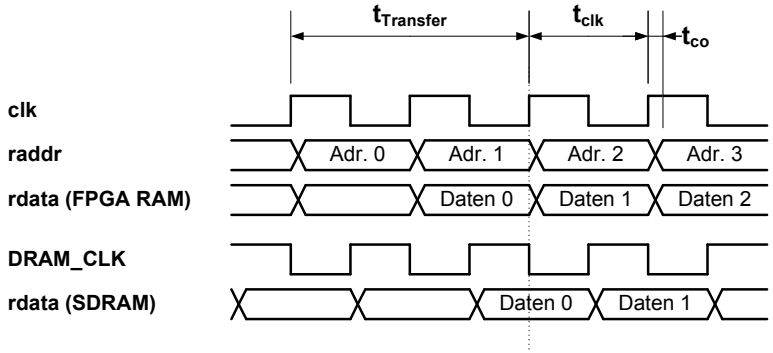

Abbildung 3.4: Anforderung an den SDRAM-Controller

Bezeichnung	Quelle	Beschreibung
$t_{Transfer}$	-	Zeit in der Datentransfer abgeschlossen sein muss
t_{clk}	-	Periodendauer des Prozessortaktes
t_{co}	QCR	Clock-to-Output. Zeit bis Daten nach steigender Taktflanke stabil anliegen

QCR ... Quartus Compilation Report

Tabelle 3.7: Legende zu Abbildung 3.4

Wird eine Adresse n bei positiver Taktflanke m von *clk* ausgegeben, so erwartet der *SCARTS*-Prozessor die Daten n bei positiver Taktflanke $m+2$ von *clk* $\Rightarrow t_{Transfer}$ bzw. 2 t_{clk}. Hierfür wird versucht, das invertierte *clk*-Signal als Takteingang für den SDRAM zu verwenden.

Durch t_{co} kann es jedoch bei höheren Traktfrequenzen zu Setup- Time-Verletzungen am SDRAM kommen. Wird der 50-MHz-Taktgeber am DE0-Nano-Board als Taktversorgung für den *SCARTS*-Prozessor verwendet, so ist $t_{clk} = 20ns$. Somit sind die positiven Taktflanken von *clk* und *DRAM_CLK* durch $10ns$ getrennt. Die Setup-Zeiten des SDRAM-Speichers betragen $1.5ns$ [12]. Es muss daher folgende Ungleichung erfüllt sein:

$$t_{co} < \frac{1}{2}t_{clk} - 1.5ns \Rightarrow t_{co} < 8.5ns \ @ \ 50MHz \tag{3.1}$$

Das Design wurde mit Quartus synthetisiert. Die maximale Clock-to-Output-Zeit aller SDRAM-relevanten Signale beträgt $12.5ns$. Die Ungleichung 3.1 ist somit nicht erfüllt. Es wird daher versucht, das Taktsignal des SDRAM's durch Verwendung einer PLL entsprechend zu verschieben. Als Basis wird die Berechnung in [7, Kap. 2] verwendet. Abbildung 3.5 zeigt den Datentransfer mit Berücksichtigung der Setup-, Hold- und Clock-to-Output-Zeiten, welche aus dem Quartus-Compilation-Report und [12] entnommen wurden. Tabelle 3.8 listet alle Zeiten und deren Werte auf.

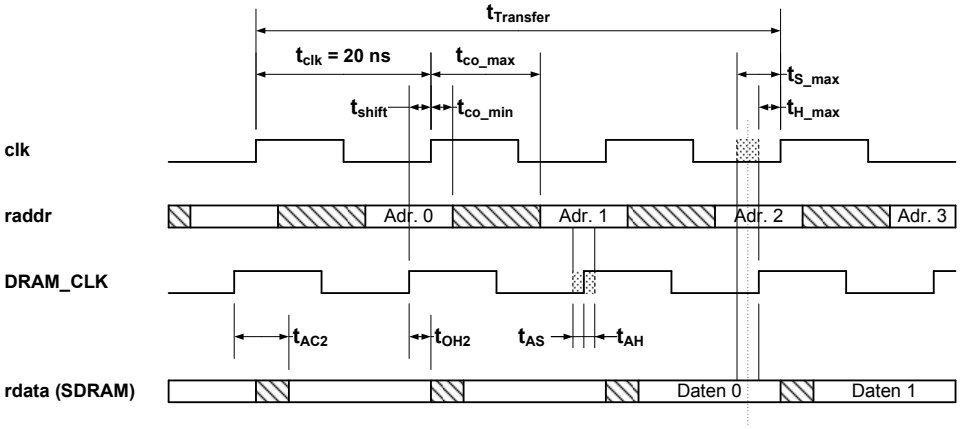

Abbildung 3.5: Datentransfer auf Hardware mit $t_{shift} = -1.893ns$

Bezeichnung	Wert [ns]	Quelle	Beschreibung
t_{co_max}	12.508	QCR	Maximale Clock-to-Ouput- Zeit für alle SDRAM-relevanten Signale
t_{co_min}	3.006	QCR	Minimale Clock-to-Ouput-Zeit
t_{S_max}	9.688	QCR	Maximale Setup-Time für clk
t_{H_max}	-3.486	QCR	Maximale Hold-Time
t_{AC2}	6.5	[12]	Access Time From CLK bei CAS-Latency = 2. Entspricht Clock-to-Output bei SDRAM
t_{OH2}	3	[12]	Output-Data-Hold-Time bei CAS-Latency = 2
t_{AS}	1.5	[12]	Address-Setup-Time
t_{AH}	0.8	[12]	Address-Hold-Time
t_{shift}	-1.893	-	Phasenshift des $DRAM\_CLK$-Signals

QCR ... Quartus Compilation Report

Tabelle 3.8: Legende zu Abbildung 3.5

Die Berechnung in [7, Kap. 2] erfolgt in 3 Schritten. Im ersten Schritt wird der maximale negative Shift des $DRAM\_CLK$-Signals entlang der Zeitachse ermittelt. Im zweiten Schritt ist der maximale positive Shift zu berechnen. Ein negativer Shift des $DRAM\_CLK$-Signals relativ zu *clk* wird in [7] als Lag, ein positiver Shift als Lead bezeichnet. Die maximalen Werte für Lead und Lag werden jeweils in Schreib- und Leserichtung begrenzt, siehe Auszug aus [7]:

$$Maximum\ Lag = minimum(Read\ Lag,\ Write\ Lag) \tag{3.2}$$

$$Maximum\ Lead = minimum(Read\ Lead,\ Write\ Lead) \tag{3.3}$$

Im letzten Schritt wird aus den maximalen Lead- und Lag-Werten ein Fenster gebildet. Die positive Taktflanke des $DRAM\_CLK$-Signals wird in die Mitte dieses Fensters geshiftet $\Rightarrow t_{shift}$.

Im folgenden wird die Berechnung von t_{shift} mit den Werten aus Tabelle 3.8 durchgeführt:

$$Read\ Lag = t_{OH2} - t_{H\_max} = 6.486ns \tag{3.4}$$

$$Write\ Lag = t_{clk} - t_{co\_max} - t_{AS} = 5.992ns \tag{3.5}$$

$$Maximum\ Lag = minimum(Read\ Lag,\ Write\ Lag) = 5.992ns \tag{3.6}$$

$$Read\ Lead = t_{clk} - t_{AC2} - t_{S\_max} = 3.812ns \tag{3.7}$$

$$Write\ Lead = t_{co\_min} - t_{AH} = 2.206ns \tag{3.8}$$

$$Maximum\ Lead = minimum(Read\ Lead,\ Write\ Lead) = 2.206ns \tag{3.9}$$

$$\underline{t_{shift}} = \frac{Maximum\ Lead - Maximum\ Lag}{2} = \underline{-1.893ns} \tag{3.10}$$

Abbildung 3.5 zeigt, dass durch das Verschieben des $DRAM\_CLK$- Signals mittels PLL keine Setup/Hold-Time Verletzungen auftreten. Die Zeit $t_{Transfer}$ beträgt jedoch 3 t_{clk} und erfüllt die, in Abbildung 3.4 gegebene, Anforderung nicht.

Die Taktversorgung des *SCARTS*-Prozessors wird daher auf eine Taktfrequenz von 25MHz herabgesetzt, die PLL entfernt und $DRAM\_CLK$ zu *clk* invertiert ausgegeben. Abbildung 3.6 zeigt den finalen Datentransfer zwischen SDRAM-Controller und Speicher. Damit die Setup-Zeiten in Abbildung 3.6 nicht verletzt werden, müssen folgende Ungleichungen erfüllt sein:

$$t_{co\_max} < \frac{t_{clk}}{2} - t_{AS} \tag{3.11}$$

$$t_{AC2} < \frac{t_{clk}}{2} - t_{S\_max} \tag{3.12}$$

Die in Tabelle 3.8 gelisteten Zeiten erfüllen diese Ungleichungen.

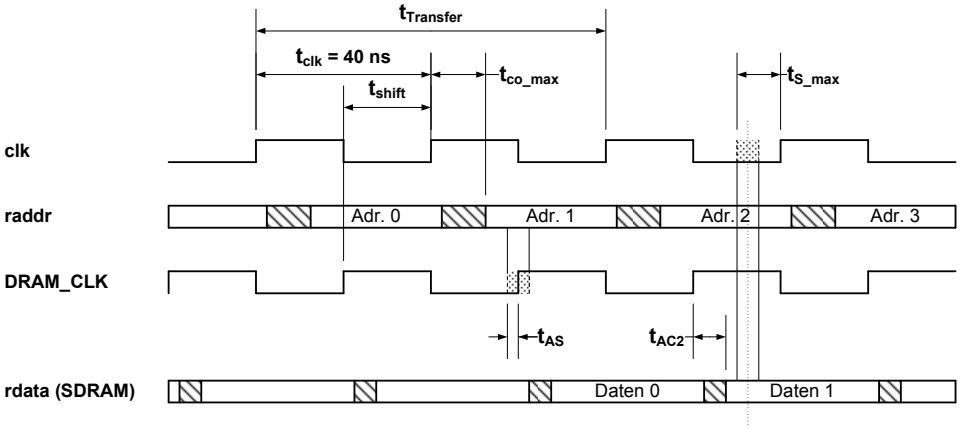

Abbildung 3.6: Finaler Datentransfer

Kapitel 4

Software

Dieses Kapitel behandelt die Kommunikation zwischen Benutzerprogramm und *BSL*, die Nutzung der zur Verfügung stehenden Speicher, das Bootkonzept des *DCD*, sowie das Verpackungsprogramm, welches die, in der *Applikation* benötigten, Dateien zu einer Applikationsdatei zusammenfügt und beim Senden an das *DCD* entpackt.

4.1 Benutzerprogramm und *BSL*

Für die Übertragung der *Applikation* zum *DCD* wird seitens des Benutzers das Benutzerprogramm und seitens *DCD* der *BSL* benötigt. Das Benutzerprogramm ist ein Konsolenprogramm und sendet Dateien am PC mittels proprietären Protokolls, in dieser Arbeit BSL-Protokoll genannt, an das *DCD*. Der *BSL* ist ein Protokollinterpreter und wird am *DCD* ausgeführt. Der *SCARTS*-Prozessor führt entweder die *Applikation* oder den *BSL* aus. Das BSL-Protokoll beinhaltet ein Kommando, welches den Wechsel zwischen *Applikation* und *BSL* ermöglicht. Der Wechsel zwischen diesen beiden Programmen wird in Punkt 4.3 beschrieben.

4.1.1 BSL-Protokoll

Das Protokoll ermöglicht das Lesen und Beschreiben von Daten eines Speichers am *DCD*. Die verfügbaren Speicherbausteine sind der Flash und der Konfigurationsspeicher. Zusätzlich stellt das Protokoll einen Befehl zum Auslesen der Version des laufenden Programmes am *DCD* und einen Befehl für den Wechsel zwischen *Applikation* und *BSL* bereit.

Die Protokolldaten können im Datenbereich eines Ethernet-, oder UDP-Paketes enthalten sein. Für den Transport über das Internet wird UDP/IP benötigt. Abbildung 4.1 zeigt den Aufbau eines BSL-Protokollpaketes.

Das Feld *BSL-Protokollkennung* ist 12 Bytes lang und beinhaltet die Zeichenfolge „BSL_Protocol“. Die *Speicher-ID* selektiert einen Speicherbaustein

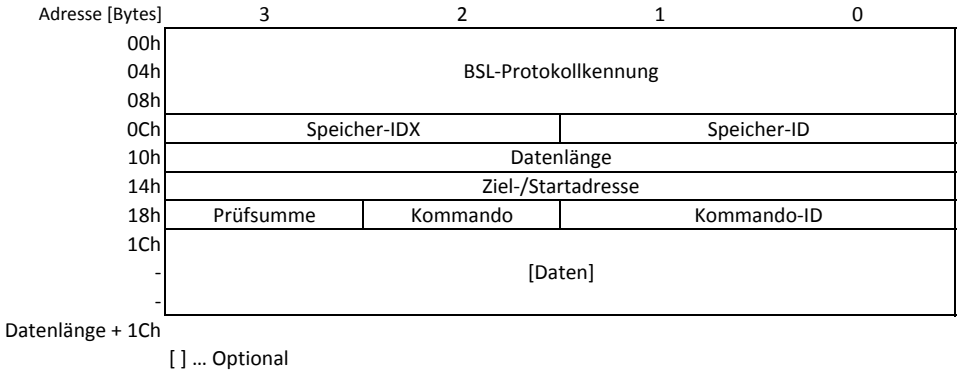

Tabelle 4.1: BSL Protokoll Paket

am *DCD*. Tabelle 4.2 zeigt die verfügbaren *Speicher ID*'s. *Speicher-IDX* wird in dieser Arbeit nicht verwendet.

ID	Speicherbaustein
0	Flash
1	Konfigurationsspeicher

Tabelle 4.2: Zuweisung der Speicher-ID's

Das Feld *Datenlänge* ist ein 32-Bit-Wert und gibt die Länge des optionalen Datenfeldes, oder der angeforderten Daten in Bytes an. Die *Ziel-/ Startadresse* ist eine Offsetadresse in Bytes, welche den Standort der zu lesenden/schreibenden Daten im Speicher bestimmt. Die *Kommando ID* ist ein vom Benutzerprogramm generierter 16-Bit-Wert. Bei einer Antwort vom *BSL* ist das Einerkomplement des generierten Wertes in diesem Feld gespeichert. Das *Kommando* bestimmt die auszuführende Aktion. Die verfügbaren Kommandos sind in Tabelle 4.3 aufgelistet. Die *Prüfsumme* ist ein 8-Bit-Wert zur Sicherung der Kopfdaten des Protokolls. Berechnet wird dieses Feld durch Addition aller Bytes von Adresse 00h bis 1Ah im Paket. Von dem LSB der Summe wird das Einerkomplement gebildet und dieses als *Prüfsumme* des Paketes verwendet.

Das Benutzerprogramm initiiert alle Übertragungen und erwartet bei *bsl_rd_vers*, *bsl_rd_mem* und *bsl_wr_mem* eine Antwort vom *BSL*. Beim Übertragen einer Datei werden mittels Kommando *bsl_wr_mem* die Daten in einen beliebigen Speicher geschrieben. Das Benutzerprogramm liest anschließend mittels *bsl_rd_mem* die geschriebenen Daten aus und vergleicht diese mit der Datei am PC. Wird ein Unterschied zwischen gelesenen Daten und der Datei am PC festgestellt, so wird die Übertragung mit einer Fehlermeldung abgebrochen.

Kommando	Bezeichnung	Beschreibung
0	bsl_rd_vers	Lese Version des laufenden Programmes
1	bsl_start	Starten eines Programmes bzw. Programmwechsel
2	bsl_rd_mem	Speicher lesen
3	bsl_wr_mem	Speicher schreiben

Tabelle 4.3: Verfügbare Kommandos im BSL-Protokoll

Die Kommandos *bsl_rd_vers* und *bsl_start* müssen vom *BSL* und der *Applikation* unterstützt werden. Nachfolgende Grafiken zeigen Beispielpakete für alle Kommandos. Das Feld *BSL-Protokollkennung* wird nicht dargestellt.

bsl_rd_vers

Lesen der Version des laufenden Programmes. Die Version eines Programmes ist ein 16-Bit-Wert und ist im *BSL* zwischen 100h und FFFh bzw. in der *Applikation* zwischen 1000h und FFFFh zu wählen.

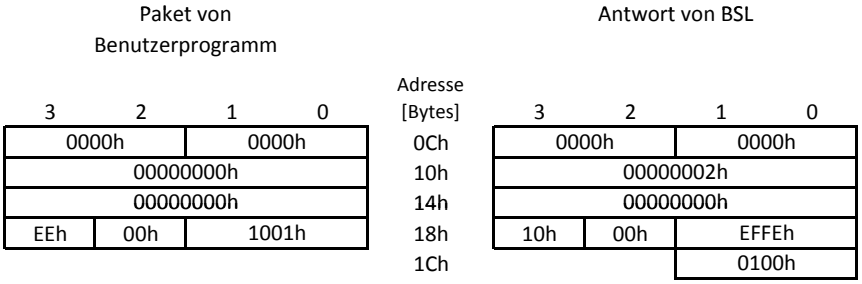

Abbildung 4.1: Kommando *bsl_rd_vers*

bsl_start

Starten der *Applikation*.

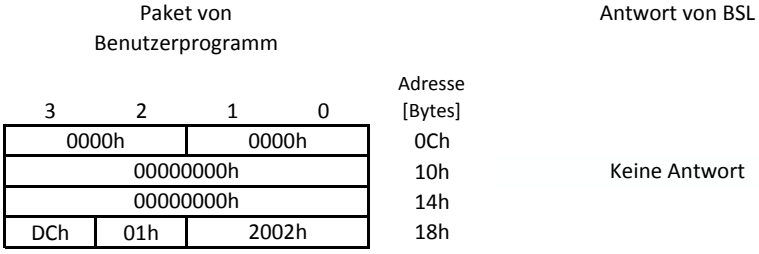

Abbildung 4.2: Kommando *bsl_start*

bsl_rd_mem

Lesen von 8 Datenbytes ab Adresse 10000h des Konfigurationsspeichers.

Paket von Benutzerprogramm

3	2	1	0
0000h		0001h	
00000008h			
00010000h			
F2h	02h	0001h	

Adresse [Bytes]
0Ch
10h
14h
18h
1Ch
20h

Antwort von BSL

3	2	1	0
0000h		0001h	
00000008h			
00010000h			
F6h	02h	FFFEh	
03h	02h	01h	00h
07h	06h	05h	04h

Abbildung 4.3: Kommando *bsl_rd_mem*

bsl_wr_mem

Schreiben von 4 Datenbytes auf Adresse 00000000h des Flashspeichers.

Paket von Benutzerprogramm

3	2	1	0
0000h		0000h	
00000004h			
00000000h			
F7h	03h	0001h	
DDh	CCh	BBh	AAh

Adresse [Bytes]
0Ch
10h
14h
18h
1Ch

Antwort von BSL

3	2	1	0
0000h		0000h	
00000000h			
00000000h			
FFh	03h	FFFEh	

Abbildung 4.4: Kommando *bsl_wr_mem*

4.2 Speicheraufteilung

Abbildung 4.5 stellt die Verwendung der permanenten Speicher am *DCD* dar. Im Flash sind die Instruktionen und die Daten der vorinitialisierten Variablen des *BSL* und der *Applikation* gespeichert. Die Instruktionen werden beim Start des *DCD* vom Boot-Programm des *SCARTS*-Prozessors in den SDRAM kopiert. Die Datenbereiche der Programme im Flash sind Speicherabbilder des Datenspeichers im FPGA. Vor Ausführung eines Programmes muss dessen Datenbereich in den Datenspeicher des *SCARTS*-Prozessors kopiert werden. Dies ist die Aufgabe der C-Runtime (CRT), auch als Startup Code bezeichnet, welche im *BSL* und in der *Applikation* enthalten ist.

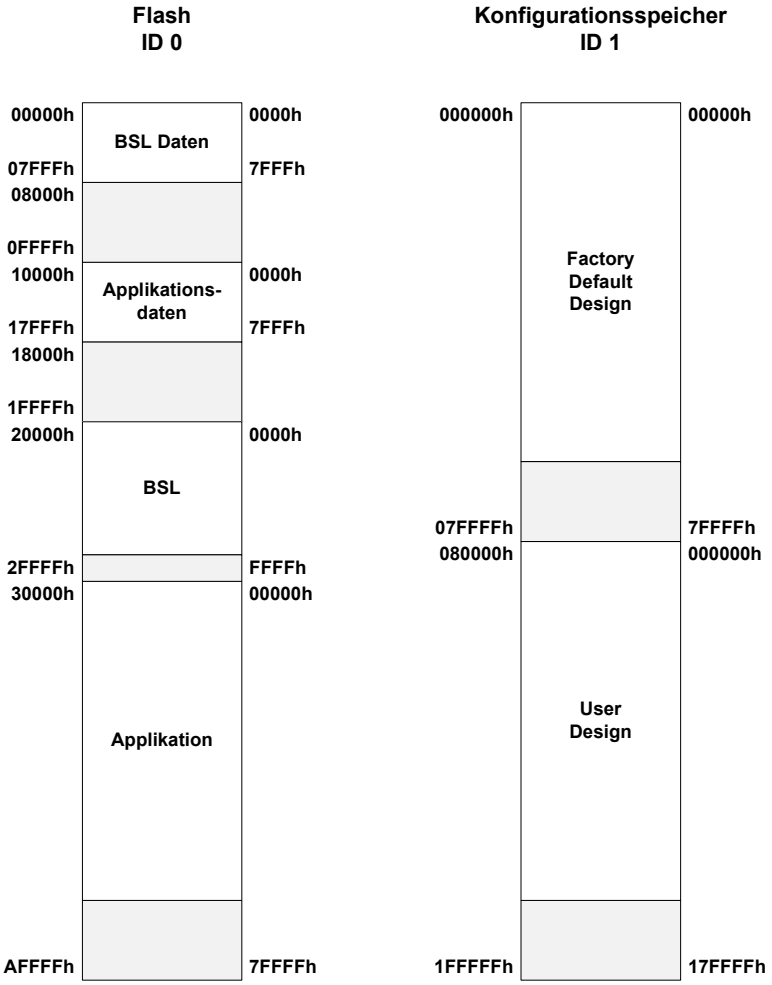

Abbildung 4.5: Memory Mapping des Flash- und Konfigurationsspeichers

Der Konfigurationsspeicher enthält zwei FPGA-Designs. Beginnend ab Adresse 0 das *Factory-Default-Design*, welches im *Remote Configuration Mode* [14] von Altera benötigt wird. Die Aufgabe dieses Designs ist, den Auslöser für die Konfiguration des FPGA's zu ermitteln. Dieser ist im Statusregister der *ALTREMOTE_UPDATE Megafunction* [14] gespeichert. Die *ALTREMOTE_UPDATE Megafunction* ist im *Factory-Default-Design* und im *User-Design* instantiiert. Ist der Auslöser ein PowerOn des FPGA's, oder eine Rekonfigurationsanforderung des *User-Designs*, so wird das *User-Design* in das FPGA geladen. Tritt ein Fehler bei der Konfiguration des *User-Designs* auf, aktiviert die *ALTREMOTE_UPDATE Megafunction* erneut das *Factory-Default-Design*, welches den Fehler aus dem Statusregister der *ALTREMOTE_UPDATE Megafunction* liest und in einen definierten Fehlerzustand übergeht.

Das *User-Design* ist im Konfigurationsspeicher ab Adresse 80000h gespeichert. Dieses enthält den *SCARTS*-Prozessor und dessen Erweiterungsmodule. Bei einem Wechsel der *Applikation* wird dieses Design überschrieben.

4.3 Bootkonzept

Die Inbetriebnahme des *DCD* erfolgt in drei Phasen. In der ersten Phase muss der *BSL* in den Flash-Speicher des *DCD* abgelegt werden. Phase 2 behandelt den Start des *BSL* und das Speichern der *Applikation*. Die Aufgabe von Phase 3 ist es, die *Applikation* zu starten und der Wechsel in den *BSL*. Für das Bootkonzept verwendet das *DCD* die, in Abbildung 4.6 dargestellten, permanenten und flüchtigen Speicher.

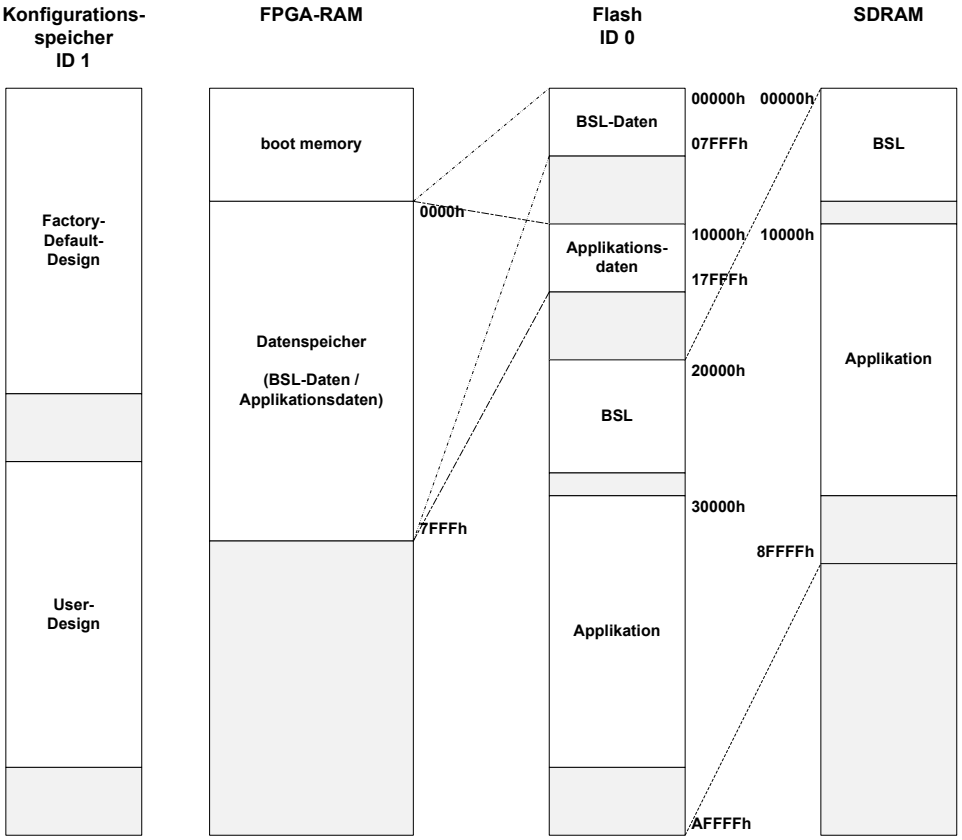

Abbildung 4.6: Übersicht aller Speicher

Phase 1

1. Per USB [6, Kap. 3.1] wird ein, speziell für die Inbetriebnahme des *DCD* angepasstes, Design in das FPGA des DE0-Nano-Board's übertragen. Dieses Design enthält den *SCARTS*-Prozessor mit ALTREMOTE-, GPIO- und SPI-Modulen. Im *boot memory* ist der *BSL* exklusive Internet-Stack gespeichert. Das Design kann somit per Ethernet und BSL-Protokoll Daten empfangen und in den Flash bzw. Konfigurationsspeicher ablegen. Der SDRAM-Controller ist in diesem Design nicht vorhanden.
2. Das *Factory-Default-Design* wird per BSL-Protokoll empfangen und ab Adresse 0 im Konfigurationsspeicher abgelegt.
3. Das *User-Design* mit *SCARTS*-Prozessor, GPIO-, SPI-, ALTREMOTE-, Zähler-Modul und SDRAM-Controller wird im Konfigurationsspeicher ab Adresse 80000h abgelegt. Das *boot memory* dieses Designs enthält das Boot-Programm, welches den *BSL* beim Start vom Flash in den SDRAM kopiert und anschließend ausführt.
4. Die Instruktionen und der Datenbereich des *BSL* werden empfangen und in den Flash-Speicher geschrieben.
5. Der Ethernet-*BSL* empfängt das Kommando *bsl_start* und leitet mittels ALTREMOTE-Modul eine Rekonfiguration des FPGA's ein. Das *User-Design* wird dadurch in das FPGA übertragen.

Phase 2

6. Das Boot-Programm des *User-Designs* kopiert den Inhalt des Flash-Speichers von Adresse 20000h bis AFFFFh, welcher den *BSL* beinhaltet, in den SDRAM ab Adresse 0 und startet die Ausführung des *BSL*.
7. Der Startup-Code des *BSL* kopiert dessen Datenbereich vom Flash in den Datenspeicher des *SCARTS*-Prozessors. Der *BSL* kann nun Daten per Internet empfangen.
8. Das *User-Design* der *Applikation* wird im Konfigurationsspeicher ab Adresse 80000h abgelegt. Dieses Design enthält den *SCARTS*-Prozessor mit SDRAM- Controller, GPIO-, SPI-, ALTREMOTE-, Zähler- und falls erforderlich Spezial- und/oder Co-Prozessor-Modul.
9. Die Instruktionen und der Datenbereich der *Applikation* werden empfangen und in den Flash-Speicher geschrieben.
10. Der *BSL* empfängt das Kommando *bsl_start* und leitet mittels ALTREMOTE-Modul eine Rekonfiguration des FPGA's ein. Das *User-Design* der *Applikation* wird dadurch in das FPGA übertragen.

Phase 3

11. Das Boot-Programm des *User-Designs* kopiert den Inhalt des Flash-Speichers von Adresse 20000h bis AFFFFh, welcher nun den *BSL* und die *Applikation* beinhaltet, in den SDRAM ab Adresse 0 und startet die Ausführung der *Applikation*. Ist eine *Applikation* am *DCD* vorhanden, beginnt der Bootvorgang nach einem PowerOn des Gerätes in diesem Schritt.
12. Der Startup-Code der *Applikation* kopiert dessen Datenbereich vom Flash in den Datenspeicher des *SCARTS*-Prozessor. Die *Applikation* kann nun die Kommunikation mit einem Projektserver aufnehmen.
13. Für den Wechsel der *Applikation* empfängt die aktuelle *Applikation* das Kommando *bsl_start*, beendet die Kommunikation mit dem Projektserver und startet die Ausführung des *BSL* durch einen Jump auf Adresse 0 im SDRAM. Anschließend werden die Schritte in Phase 2 beginnen mit 7. durchgeführt.

4.4 Verpackungsprogramm

Die benötigten Dateien der *Applikation* werden in einer Applikationsdatei vereint, dessen Endung *.dca ... Distributed Computing Application* lautet. Die Applikationsdatei enthält hierfür ein einfaches Dateisystem. Die einzelnen Dateien werden binär aufeinanderfolgend in die Applikationsdatei gespeichert. Vor jeder Datei liegt ein 4 Byte Header der die Größe der Datei in Bytes enthält. Beim Entpacken liest das Verpackungsprogramm die ersten 4 Bytes der Applikationsdatei aus und interpretiert diese als Länge n der ersten Datei. Die folgenden n Bytes werden als Daten interpretiert und in eine Datei mit dem Namen *dat0.bin* gespeichert. Die nächsten 4 Bytes repräsentieren nun die Größe der zweiten Datei. Das Verpackungsprogramm speichert die Daten in *dat1.bin*. Dieser Vorgang wird wiederholt, bis das Ende der Applikationsdatei erreicht ist.

Das Verpackungsprogramm ist ein Konsolenprogramm, welches einen oder mehrere Dateinamen als Argumente benötigt. Von der Anzahl der Programmargumente wird der auszuführende Vorgang abgeleitet. Ist die Anzahl 1, so wird das Argument als Dateiname einer Applikationsdatei interpretiert und diese entpackt. Ist die Anzahl größer 1, werden die Dateien in einer Applikationsdatei vereint. Das Speichern der Dateien erfolgt in der Reihenfolge der Programmargumente.

Sobald alle enthaltenen Dateien der Applikationsdatei entpackt wurden, sind diese mit Hilfe des Benutzerprogramms an das *DCD* zu senden. Der Inhalt einer Applikationsdatei ist wie folgt definiert:

1. Instruktionen der *Applikation*
2. *Applikationsdaten*
3. *User-Design* der *Applikation*

Somit sind die Instruktionen der *Applikation* in den Flash des *DCD* ab Adresse 30000h, die *Applikationsdaten* ab Adresse 10000h und das *User-Design* in den Konfigurationsspeicher ab Adresse 80000h zu speichern. Die, bei der Entpackung erzeugten, Dateien dat0.bin, dat1.bin und dat2.bin können nach der Übertragung an das *DCD* gelöscht werden.

Kapitel 5

Ergebnisse

Dieses Kapitel fasst die Ergebnisse dieser Arbeit zusammen. Darin enthalten ist der Entwicklungsstand der Hard- und Software, die Verwendbarkeit und mögliche Erweiterungen des Gerätes.

5.1 Entwicklungstand

5.1.1 Hardware

Es wurden folgende Hardware-Teile des *DCD* fertiggestellt:

- Platinenerweiterung mit Flash und Netzwerk-Controller
- GPIO-Modul
- SPI-Modul
- Zähler-Modul
- SDRAM-Controller

Die Instantiierung und Bedienung der *ALTREMOTE_UPDATE-Megafunction* im ALTREMOTE Modul wurde begonnen, jedoch aus Zeitgründen nicht fertiggestellt. Dieses Modul ist nicht funktionsfähig. Es kann daher per Software keine Rekonfiguration des FPGA's eingeleitet werden. Das *Factory-Default-Design* ist nicht implementiert. Als Workaround wird das *User-Design* im Konfigurationsspeicher ab Adresse 0 abgelegt. Eine Rekonfiguration des FPGA's wird durch Power-Off-On des *DCD* durchgeführt.

5.1.2 Software

Folgende Teile der Software sind funktionsfähig:

- Benutzerprogramm mit UDP/IP Erweiterung
- Verpackungsprogramm
- Boot-Programm des *boot memorys* im *SCARTS*-Prozessor
- *BSL* mit folgenden Programmteilen:

- C-Runtime/Startup-Code
- Treiber für SPI-Modul, Flash, Konfigurationsspeicher und Netzwerk-Controller
- Microchip-TCP/IP-Stack mit IP, ARP, UDP, DHCP, ICMP
- BSL Protokollinterpreter

Der *BSL* kann sich selbst und eine *Applikation* per Internet empfangen und am *DCD* speichern. Eine *Applikation* wurde in dieser Arbeit nicht implementiert.

Das gesetzte Ziel wurde somit fast erreicht. Da die Rekonfiguration und daher der Neustart des *DCD* jedoch nicht per Software durchführbar ist, muss das *DCD* für den Benutzer erreichbar sein. Die dafür erforderlichen Hardwarekomponenten sind das ALTREMOTE-Modul und das *Factory-Default-Design*.

5.1.3 Übersicht

In Tabelle 5.1 sind die Eckdaten der Hard- und Software des *DCD* aufgelistet.

Software	
Prozessor	*SCARTS32*
Prozessortakt	25MHz
Größe des Datenspeichers	32KB
Maximale Programmgröße der *Applikation*	512KB
Benötigte Bootzeit	ca. 1s
Hardware	
Anzahl der benötigten Logikeinheiten für das *User-Design* mit *SCARTS*-Prozessor, SDRAM-Controller, GPIO-, SPI-, ALTREMOTE- und Zähler-Modul	6.734 von 22.320
Anzahl der benötigten Speicherbits für das *User-Design*	350.208 von 608.256
Taktversorgung des FPGA's	50MHz
Netzwerkanschluss	10Base-T Ethernet

Tabelle 5.1: Eigenschaften des *DCD*

5.2 Verwendbarkeit

Das Abschließen eines Bitcoin Blocks, siehe Punkt 1.1, benötigt die Berechnung von SHA256-Hashwerten, welche durch ein Spezialmodul der *Applikation* effizient in Hardware durchgeführt werden könnte. Das Spezialmodul verwendet hierfür das auf OpenCores frei erhältliche SHA256-Hardware-Modul

von Arif Endro Nugroho[1]. Dieses Modul Berechnet den Hashwert von 512 Bit langen Datenblöcken beginnend mit einem initialen Hashwert in 40 Taktzyklen und benötigt hierfür ca. 2200 Logikeinheiten. Die Berechnung eines Bitcoin Hashwertes erfordert die Berechnung von zwei SHA256-Hashwerten und ist demnach in 80 Taktzyklen abgeschlossen. Wird nun das 256 Bit lange Ergebnis in 8 Taktzyklen mit den Zielwert verglichen, benötigt der Vorgang zum Berechnen und Validieren eines Bitcoin Hashwertes 88 Taktzyklen. Bei einer Taktversorgung des Hardwaremoduls von 50MHz liegt die Leistung eines einzelnen Moduls daher bei ca. 568 kBitcoinhashes/s. Durch Instantiierung zusätzlicher, parallel arbeitender Hardwaremodule kann die Leistung gesteigert werden. Das *User-Design* stellt hierfür 15586 verbleibende Logikeinheiten bereit. Bei 7 instantiierbaren SHA256-Hardwaremodulen könnte die Leistung somit 3.976 MBitcoinhashes/s betragen. Dieser Wert ist im Vergleich zu aktuellen (Stand August 2012) Grafikkarten, welche ebenfalls für die Berechnung von Bitcoinhashes Verwendung finden und eine Leistung von mehreren hundert MBitcoinhashes/s erzeugen, sehr gering[2].

Für die Verwendung des Gerätes als Bitcoin-Rechner ist jedoch weiters die spezifische Leistung in „Anzahl der Bitcoinhashes pro verrichteter elektrischer Arbeit“ [Bitcoinhashes/J] heranzuziehen und berechnet sich wie folgt:

$$P_{spez} = \frac{P_{Bh}}{P_{el}} \tag{5.1}$$

Zeichen	Einheit	Beschreibung
P_{spez}	$\left[\frac{Bitcoinhashes}{J}\right]$	Spezifische Rechenleistung
P_{Bh}	$\left[\frac{Bitcoinhashes}{s}\right]$	Rechenleistung
P_{el}	$\left[\frac{J}{s}\right]$	Elektrische Leistung

Tabelle 5.2: Legende zu Gleichung 5.1

Die spezifische Leistung lässt eine Beurteilung über die Rentabilität des Rechengerätes zu, da es die Anzahl der errechneten Bitcoinhashes mit der dafür aufgebrachten Energie vergleicht. Im allgemeinen ist die Rechenleistung aktueller Grafikchips höher als die von FPGA Lösungen. Die Grafikkarten erwirtschaften daher schneller Bitcoins. Jedoch ist der Stromverbrauch verhältnismäßig höher.

Um die spezifische Leistung des *DCD* für die Berechnung von Bitcoinhashes ermitteln zu können, ist die Ermittlung der Stromaufnahme des Gerä-

[1] SHA256-Modul von Arif Endro Nugroho, http://opencores.org/project,nfhc

[2] Mining hardware comparison,
https://en.bitcoin.it/wiki/Mining_hardware_comparison

tes erforderlich, welche bei voller Rechenlast auftritt. Es müsste hierfür das Spezialmodul mit SHA256-Einheiten implementiert und die Stromaufnahme bei Rechenlast gemessen werden. Mit Hilfe des aktuellen Wechselkurses von Bitcoins, einigen Parametern des Bitcoin-Netzwerkes und den Stromkosten, kann somit ein Gewinn/Verlust bei Einsatz des Rechengerätes ermittelt werden.

Mittels Co-Prozessor Modul bietet das *DCD* die Möglichkeit weitere FPGA Bausteine zur Unterstützung bei der Berechnung von Bitcoinhashes heranzuziehen.

Außerhalb der Verwendung als Rechengerät für Bitcoins, oder sonstigen Projekten ist das *DCD* durch den Verbund von Hard- und Software vielseitig einsetzbar. Das DE0-Nano-Board stellt zudem 96 Benutzerpins zur Verfügung.

5.3 Mögliche Erweiterungen

Das *DCD* kann in vielen Bereichen der Hard- und Software verbessert und erweitert werden. Einige Punkte sollen hier aufgelistet werden.

5.3.1 Hardware

- Fertigstellung des ALTREMOTE-Moduls und Erstellung des *Factory-Default-Designs* um das FPGA per Software rekonfigurieren zu können.
- Modifizierung des *SCARTS*-Prozessors, sodass die Übertragung der Instruktionen vom SDRAM zum Prozessor mit einer Taktrate von 50MHz möglich ist.
- Erweiterung des *SCARTS*-Prozessors mit einer Hardware-Multiplizier-einheit.

5.3.2 Software

- Fertigstellung des *gdb-Remote-Serial-Protocol*-Interpreters für die Kommunikation zum GDB über UART, oder auch Ethernet.
- Verwendung des TCP-Moduls im TCP/IP Stack.
- Unterstützung von Applikationsdateien im Benutzerprogramm, um das Entpacken zu umgehen und die Daten in der Applikationsdatei direkt per BSL-Protokoll an das *DCD* zu senden.
- Verwendung des HTTP- und FTP-Moduls im TCP/IP-Stack für Diagnosezwecke.

Anhang A

SCARTS32 Pipeline-Erweiterung

Anhang B

ALTREMOTE-Modul

Literaturverzeichnis

[1] *1. Configuring Altera FPGAs.* Altera Corporation. 101 Innovation Drive - San Jose, CA 95134, USA, 2008.

[2] *3. Serial Configuration Devices (EPCS1, EPCS4, EPCS16, EPCS64, and EPCS128) Data Sheet.* Altera Corporation. 101 Innovation Drive - San Jose, CA 95134, USA, 2011.

[3] *A25LQ032 Series.* AMIC Technology Corporation. No. 2 Li-Hsin 6th Road, Science-based industrial Park, Hsin-Chu City, 300, Taiwan, 2010.

[4] *Bitcoin Wechselkurs.* URL: https://www.bitcoin.de/de?sj_cultrue=de (besucht am 10.08.2012).

[5] *Cyclone IV Device Handbook.* Volume 1. Altera Corporation. 101 Innovation Drive - San Jose, CA 95134, USA, 2011.

[6] *DE0-Nano User Manual.* Terasic Technologies Inc. 9F., No.176, Sec.2, Gongdao 5th Rd, East Dist, Hsinchu City, 30070. Taiwan, 2011.

[7] *Embedded Peripherals IP User Guide.* Altera Corporation. 101 Innovation Drive - San Jose, CA 95134, USA, 2011.

[8] *ENC28J60 Data Sheet.* Microchip Technology Inc. 2355 West Chandler Blvd. - Chandler, Arizona, USA 85224-6199, 2006.

[9] *ENC28J60 Rev. B4 Silicon Errata.* Microchip Technology Inc. 2355 West Chandler Blvd. - Chandler, Arizona, USA 85224-6199, 2006.

[10] Martin Fletzer. „SPEAR2 - An Improved Version of SPEAR“. Diplomarbeit. Fakultät für Informatik der Technischen Universität Wien, 2008. URL: http://ti.tuwien.ac.at/ecs/teaching/courses/hwsw_lu_WS2011.

[11] *Folding@Home.* URL: http://fah-web.stanford.edu/cgi-bin/main.py?qtype=osstats (besucht am 10.03.2012).

[12] *IS42S16160B.* Integrated Silicon Solution, Inc. 1940 Zanker Road, San Jose, CA 95112-4216, 2005.

[13] Ludwig Meier. *Portierung des GDB für die SPEAR 2 Architektur.* Fakultät für Informatik der Technischen Universität Wien. 2008. URL: http://ti.tuwien.ac.at/ecs/teaching/courses/hwsw_lu_WS2011.

[14] *Remote System Upgrade (ALTREMOTE_UPDATE) Megafunction User Guide*. Altera Corporation. 101 Innovation Drive - San Jose, CA 95134, USA, 2012.

[15] Roman Steiger. *miniUART Dokumentation Version 0.9*. Fakultät für Informatik der Technischen Universität Wien. 2007. URL: http://ti.tuwien.ac.at/ecs/teaching/courses/hwsw_lu_WS2011.

[16] Martin Walter. „The SPEAR2 Hardware/Software Interface“. Diplomarbeit. Fakultät für Informatik der Technischen Universität Wien, 2011. URL: http://ti.tuwien.ac.at/ecs/teaching/courses/hwsw_lu_WS2011.