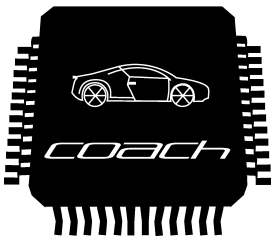

Abschlussbericht

Projektgruppe 533
„Car on a Chip“ (CoaCh)

http://ess.cs.tu-dortmund.de/DE/Teaching/PGs/coach/

Wintersemester 2008/2009 und Sommersemester 2009

Teilnehmer:
David Austin, Adrian Ben-Shlomo, Christoph Borchert, Elena-Crina Bostan, Boris Golubovic, Jens Kirch, Christoph Mertens, Jan-Philipp Niewerth, Arthur Pyka, Christian Schindler, Matthias Steinkamp, Jiong Zou

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk, Dr. Michael Engel,
Horst Schirmeier, Jochen Streicher

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl XII
Arbeitsgruppe Eingebettete Systemsoftware

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Bei dem vorliegenden Dokument handelt es sich um den Abschlussbericht der Projektgruppe (PG) 533 „Car on a Chip“ (CoaCh) des Lehrstuhls XII der Technischen Universität Dortmund.

1.1 Motivation

Kraftfahrzeuge in der heutigen Zeit sind viel mehr als nur die Summe aus mechanischen Bauteilen. Moderne Fahrzeuge verfügen über mehr als 70 eingebettete Rechnersysteme, die über verschiedene Bussysteme miteinander kommunizieren und verschiedenste Funktionen ausführen, wie z. B. die Anzeige von Informationen auf Instrumenten, Steuerung von elektrischen Schiebedächern oder Fensterhebern [66]. Diese eingebetteten Rechnersysteme werden im Automobilbereich auch als „Steuergeräte“ oder ECUs („embedded control units“) bezeichnet.

Kraftfahrzeuge haben in den letzten Dekaden eine Metamorphose vom mechanischen Meisterwerk hin zum heterogenen, verteilten, eingebetteten Rechnersystem vollzogen. Allerdings haben diese im Vergleich zu eingebetteten Systemen des Alltags, wie Mobiltelefonen oder MP3-Playern, eine wesentlich höhere Lebensdauer. Da Ersatzteile für Kraftfahrzeuge über einen Zeitraum von 30 Jahren nach Produktionsbeginn verfügbar sein müssen [54], ergibt sich ein nicht unwesentlicher Konflikt mit dem vergleichsweise schnelllebigen Halbleitermarkt. Die Verfügbarkeit von Komponenten (wie Mikrocontroller oder Peripheriebausteine) für ECUs kann nicht über den Zeitraum von 30 Jahren garantiert werden. Die Lösung, die für dieses Problem zur Zeit verwendet wird, ist die aufwändige und kostenintensive Lagerhaltung der benötigten Bauteile.

Aktuelle Entwicklungen in der Industrie versuchen die Anzahl der ECUs zu reduzieren, indem Funktionalität auf einem Controller-System integriert wird. Anstelle der Isolation von einzelnen Softwarekomponenten durch die klare Abgrenzung von Hardware würde dies durch Softwaremethoden geschehen, wie z. B. AutoSAR [4]. Dieses Verfahren ist jedoch sehr zeit- und kostenintensiv, da die Erfüllung dieses Ziels fast immer eine komplette Neuentwicklung bzw. aufwändige Portierung der vorhandenen Software erfordert.

Wünschenswert wäre es jedoch, existierende Software und Werkzeuge ohne oder mit nur geringem Aufwand weiterverwenden zu können. Einen neuen, vielversprechenden Ansatz für ECU-Architekturen, der diese Möglichkeit bietet, liefert die Integration von Systemen auf reprogrammierbarer Hardware (sog. FPGAs) als System-on-Chip (SoC). Hier können die digitalen Komponenten eines bereits existenten eingebetteten Systems mit Hilfe einer Hardwa-

rebeschreibungssprache wie VHDL auf einem FPGA integriert werden. Diese Systemkomponenten werden dann auch als „soft core“-Implementierung bezeichnet. Komponenten wie 8/16/32-Bit-Mikrocontroller oder auch Bussysteme sind bereits sowohl von kommerziellen Anbietern als auch von Open-Source-Projekten (z. B. auf [20]) verfügbar.

1.2 Wissenschaftlicher Kontext

Die Integration eingebetteter Systeme und die damit einhergehenden Isolationsprobleme sind aktuelle Forschungsthemen, die u. a. auch Thema des von der Arbeitsgruppe „Eingebettete Systemsoftware“ (ESS) ausgerichteten IIES-Workshops sind ([47] und [48]). Die verschiedenen Ansätze zur Integration von Systemen auf Hard- wie Software-Ebene führen zu jeweils spezifisch zu lösenden Problemen, wie der Integration von Legacy-Komponenten oder der Entwicklung für eingebettete Multicore-Systeme [64].

Der für diese Projektgruppe gewählte Ansatz der Integration von Komponenten auf einem FPGA als Network-on-Chip (siehe Abbildung 1) [40] ist eine Methode der Hardware-Spezialisierung, die als Alternative zu aktuellen Trends der Verwendung von Multicore-Systemen (Hardware-Vervielfachung) steht und damit andere Entwicklungsansätze erfordert, zugleich aber andere Strukturen nachbilden kann. Hier ist von Interesse, inwieweit der gewählte Isolations- und Integrationsprozess praxistauglich ist. Dazu ist eine Evaluierung des Kommunikations- und Zeitverhaltens im Vergleich zu realen Systemen vorgesehen.

Ein weiterer forschungsrelevanter Aspekt ist die Evaluation von querschneidenden Belangen über die Hardware-/Software-Grenzen hinaus [46]. Hier ist insbesondere die durchgängige Konfigurierung von eingebetteten Systemen auf Hardware- und Software-Ebene von Interesse. Dieser Ansatz soll die bisherigen Ergebnisse der Arbeitsgruppe im Bereich der Software-Produktlinien [72, 74] auf integrierte eingebettete Systeme ausweiten.

1.3 Ziele der PG

Im Rahmen einer der CoaCh-Projektgruppe vorangegangenen PG namens „AutoLab“ wurde der Grundstein für das Automobil-Labor der Arbeitsgruppe gelegt. Die in „AutoLab“ erzielten Erkenntnisse werden nun verwendet, um in CoaCh eine Gesamtsicht auf Hard- wie Software eines eingebetteten Systems zu geben.

Damit ist dann die Erfüllung folgender Einzelziele möglich:

- Entwicklung von eingebetteten Systems-on-Chip für den Automobilbereich

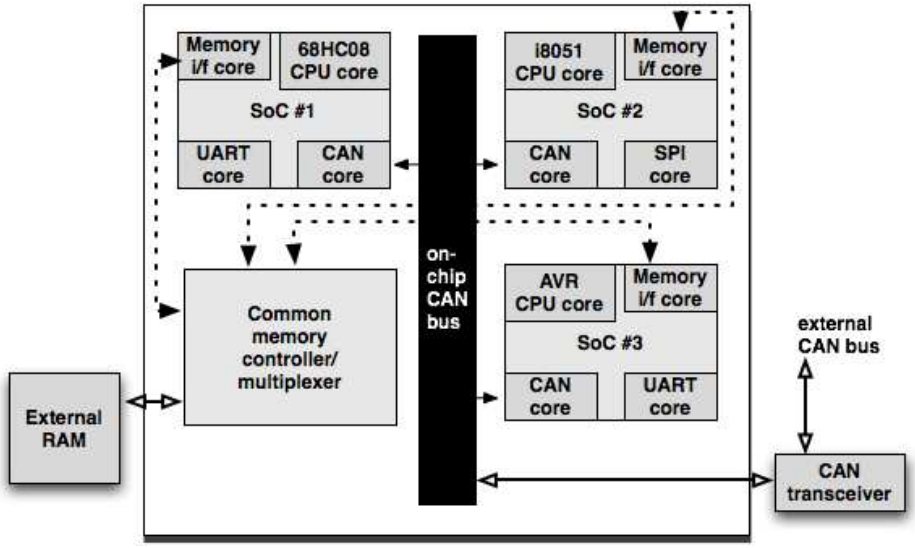

Abbildung 1: Automobiles Network-on-Chip

- Integration heterogener Systeme auf einem Chip als Network-on-Chip
- Gesamtkonfigurierung von eingebetteten Systemen auf Hardware-, Software- und Netzwerkebene

Die im Zuge der AutoLab-PG initiierte Zusammenarbeit mit Industriepartnern soll auch im Rahmen von CoaCh weitergeführt werden.

1.3.1 Minimalziele

Die folgenden Punkte stellen die Minimalziele dar und somit das zu absolvierende Pensum der Projektgruppenteilnehmer:

1. Entwicklung von Systems-on-Chip
   (a) Detaillierter Entwurf von Systems-on-Chip für automobile Controller und Evaluierung verfügbarer Open-Source-Komponenten
   (b) Implementierung der jeweiligen Systeme unter Verwendung der evaluierten Komponenten
2. Erstellung einer Beispielanwendung für das jeweilige System-on-Chip
3. Planung der Integration der SoCs in ein Network-on-Chip
   (a) Ermittlung von Integrationsproblemen

   (b) Entwicklung von Konzepten für die Buskomponente und weitere Integrationskomponenten

4. Implementierung des Network-on-Chip, Integration der in (1) und (2) erstellten Hard- und Software-Komponenten unter Anwendung der in (3) erarbeiteten Konzepte

5. Dokumentation aller Ergebnisse, insbesondere in Form des Zwischen- und Abschlussberichts

1.4 Mitglieder

Von Seiten des Lehrstuhls XII wird diese Projektgruppe betreut von:

- Prof. Dr.-Ing. Olaf Spinczyk
- Dr. Michael Engel
- Horst Schirmeier
- Jochen Streicher

Die Projektgruppe CoaCh besteht des weiteren aus folgenden Teilnehmern:

- David Austin
- Adrian Ben-Shlomo
- Christoph Borchert
- Elena-Crina Bostan
- Boris Golubovic
- Jens Kirch
- Christoph Mertens
- Jan-Philipp Niewerth
- Arthur Pyka
- Christian Schindler
- Matthias Steinkamp
- Jiong Zou

Alle Teilnehmer der Projektgruppe sind zugleich gemeinschaftlich Autoren dieses Berichts.

1.5 Aufbau des Dokuments

Zu Beginn der Projektgruppe wurden einige Seminarvorträge gehalten, um jedem Teilnehmer die selbe Wissensbasis zu vermitteln. Eine Zusammenfassung der einzelnen Vorträge findet sich in Kapitel 2.

Um einen Überblick über die auf dem Markt frei verfügbaren SoCs zu erhalten, wurden im Rahmen einer Evaluationsphase mehrere SoCs betrachtet. Die Ergebnisse dieser Evaluation sind in Kapitel 3 zu finden.

Eine Einführung in den Entwicklungsprozess von SoCs liefert das Kapitel 4. In diesem Kapitel wird unterteilt in die Compiler-Werkzeugkette (Kapitel 4.1) und die FPGA-Werkzeugkette (Kapitel 4.2).

Kapitel 5 beschäftigt sich mit der Entwicklung von SoCs. Es werden die verschiedenen Bestandteile eines SoCs sowie einige Peripheriekomponenten genau beschrieben, um einen Einblick in die Arbeitsweise mit SoCs zu vermitteln.

Im darauf folgenden Kapitel 6 werden drei Demonstratoren vorgestellt, die wir mit Hilfe der entwickelten SoCs aufgebaut haben.

In Kapitel 7 wird die Entwicklung der zur Integration der entwickelten SoCs in ein Network-On-Chip notwendigen Komponenten beschrieben.

Darauffolgend wird in Kapitel 8 der von uns entwickelte Demonstrationsaufbau vorgestellt.

Mit Kapitel 9 werden anschließend einige Evalutionsergebnisse vorgestellt.

Das letzte Kapitel 10 beinhaltet schließlich ein kurzes Fazit über die während der Projektgruppe erzielten Leistungen.

2 Seminarphase

Zu Beginn der Projektgruppe haben wir uns in einer Seminarphase mit den Grundlagen für das anstehende Projekt vertraut gemacht. Dieses Kapitel soll einen Überblick über die behandelten Themen geben, die im Folgenden in Kurzzusammenfassungen aufgeführt werden. Die Ausarbeitungen zu den jeweiligen Vorträgen sind unter [7] zu finden.

2.1 Architektur von Computersystemen: Systems-on-Chip

Im Bereich der eingebetteten Systeme erfreuen sich Systems-on-a-Chip immer größerer Beliebtheit. Die Technologie von heute macht es möglich viele Millionen Gatter in einem Chip zu integrieren. So können komplette Systeme auf einem einzigen Chip untergebracht werden. Das Entwickeln solch komplexer Systeme kann daher nicht mehr auf der Gatterebene beginnen. In diesem Vortrag wurden zu Beginn einige Tools und Techniken vorgestellt, um auf hoher Abstraktionsebene das Entwickeln von Systems-on-a-Chip zu starten [66]. Während des Entwicklungsprozesses ist es daher auch unabkömmlich die einzelnen Entwurfsschritte zu simulieren. Im Designprozess müssen darüber hinaus viele Entscheidungen getroffen werden, die die Effizienz des Endproduktes wesentlich beeinflussen können. Daher ging es im zweiten Teil des Vortrags um Problemstellungen im Bereich des Scheduling, der Speicherhierarchie und der Bussysteme für Systems-on-a-Chip. Exemplarisch wurden CoreConnect [58] und Wishbone [69] als Standard-on-Chip-Bussysteme vorgestellt.

2.2 Networks-on-Chip

Die Netzwerke auf einem Chip stellen sowohl einen neuen Trend als auch ein Bedürfnis in der Weiterentwicklung der Systeme auf einem Chip dar: Für 2010 wird vorhergesagt, dass die Anzahl der Transistoren auf einem Chip mehr als 4 Milliarden erreichen wird [37], was kritisch für die Verbindungen zwischen den Komponenten sein wird. Wir haben ein Mikronetzwerkmodell vorgestellt, das auf dem ISO/OSI-Modell basiert und das den Vorteil hat, dass die Nachrichten in Paketen [68] gesendet werden. Diese neue Perspektive lässt mehr Platz für Skalierbarkeit, Modularität und Wiederverwendbarkeit der Komponenten, was sehr wichtig für zukünftige Entwicklungen ist. Das vorgestellte Mikronetzwerk hat fünf Schichten [44]: die physikalische Schicht, die Sitzungsschicht, die Netzwerkschicht, die Transportschicht und die Anwendungsschicht. Vieles für den Entwurf dieser Schichten kann

man aus dem ISO/OSI-Modell nehmen; der Unterschied besteht darin, dass man in unserem Fall lokale Nähe und dadurch mehr Vorhersagbarkeit für die Komponenten des Mikronetzwerkes hat [37].

2.3 Architektur des Intel 8051

Die Geräte der 8051-Reihe werden heute als die Klassiker der Mikrocontrollergeschichte gehandelt. Hierbei handelt es sich um eine 8-Bit-CISC-Architektur, die in ihrer ursprünglichen Form bereits vor 30 Jahren entwickelt wurde und seitdem in diversen Spielekonsolen, Taschenrechnern und in Tastaturen aus dem Hause IBM verwendet wird [32]. Der Vortrag zur 8051-Familie behandelte den technischen Aufbau des Mikrocontrollers, die CPU-Architektur, Peripherie und ging insbesondere auf den Umgang mit den verschiedenen Speicherbereichen ein und wie man diese beim Programmieren richtig anspricht. Die häufigsten Derivate wurden vorgestellt und es wurde erläutert, um welche Funktionen diese die 8051-Architektur erweitern. Zusätzlich wurden einfache Programmierbeispiele in C und Assembler zur Ansteuerung der LEDs gezeigt [77].

2.4 Architektur des Freescale 9S12

Der Freescale 9S12 [18] ist ein beliebter 16-Bit-Mikrocontroller von Motorola, der in Packages mit umfangreicher auf dem Chip integrierter Peripherie verfügbar ist. So sind neben den auf dem Chip integrierten Speichern, die bis zu 128KB Flash-ROM und bis zu 4KB RAM umfassen, viele zur Kommunikation mit der Außenwelt erforderliche Peripheriemodule integriert. Insbesondere das integrierte CAN-Bus-Interface macht ihn für den Einsatz in eingebetteten Systemen im Automobilbereich interessant. Der Vortrag ging auf die Herkunft und die Funktionalität des eigentlichen CPU-Kerns ein. Hier wurden speziell der Registersatz, die verfügbaren Datentypen sowie die unterstützten Adressierungsarten betrachtet. Außerdem wurde die Integration der On-Chip-Peripherie mit dem CPU-Kern vorgestellt, bevor der Vortrag im letzten Teil über die Verfügbarkeit von Compilersoftware sowie VHDL-spezifizierten Softcores handelte.

2.5 Architektur des Atmel AVR

Der AVR [5] von Atmel ist eine relativ neue Architektur, verglichen mit anderen Mikrocontrollern. Der Kern besteht aus einem 8-Bit-RISC-Aufbau, dessen Arbeitsablauf pro Takt durchschnittlich einen Befehl ausführen kann. Die Produktfamilie des AVR bietet ein Spektrum von kleinen ATtiny-Modellen,

mit 32Byte RAM und 1KB Flash-ROM, bis zu großen ATmega-Modellen, mit 8KB RAM sowie 256KB Flash-ROM. Des weiteren werden in einigen Modellen auch spezielle Peripheriebausteine, wie USB-, LIN- oder auch CAN-Controller [38] verbaut. Im Vortrag wurde erläutert, dass der AVR einige Möglichkeiten bietet sparsam zu arbeiten. Hierzu wurden die verschiedenen Schlafmodi, sowie exemplarisch das Abschalten einer Komponente gezeigt. Weiterhin wurden zentrale Aspekte wie die verschiedenen Register, der verbaute CAN-Controller und das notwendige Softwarepaket [39] vorgestellt. Zuletzt behandelte der Vortrag die vorhandenen Softcores [76, 65, 43, 56], die den AVR simulieren.

2.6 Projektmanagement im Ingenieurbereich

Als Projektmanagement bezeichnet man die Herausforderung ein Projekt zu führen, zu lenken und zu kontrollieren. In dem Vortrag wurden dazu einige Agile Methoden vorgestellt. Diese neue Art des Projektmanagements zeichnet sich durch eine hohe Flexibilität, wenige einfache Regeln und das inkrementelle Vorgehen aus. Man erhofft sich damit auf geänderte Umstände schneller reagieren zu können. eXtreme Programming (XP) [55] wurde genauer vorgestellt und auf einzelne Methoden eingegangen, wie beispielsweise „Schnelles Feedback“, wobei versucht wird, nach kürzester Zeit Ergebnisse zu haben, um einen Soll-Ist-Vergleich durchzuführen. Auch Programmieren in Paaren, bei dem immer zu zweit am Rechner gearbeitet wird, wurde vorgestellt. Neben XP wurde auch Scrum [70] vorgestellt. Für unser Projekt wollen wir jedoch keine Methode vollständig anwenden, da sie nicht für die hardwarenahe Arbeit und unsere Gruppenaufteilung (Kleingruppen) optimiert sind, sondern lediglich von einzelnen Techniken profitieren.

2.7 Zeitmanagement

Beim Zeitmanagement geht es darum, die Zeit für die wesentlichen Dinge zu nutzen. Einige der Vorteile, die mit konsequentem Zeitmanagement erreicht werden können, sind: Unterscheiden zwischen Wichtigem und weniger Wichtigem, Setzen von Prioritäten, Verringern von negativem Stress, Erhöhung der Selbstdisziplin und der Selbstkontrolle. Für eine Projektarbeit, die sich am Projektziel orientiert, ist effektives und ganzheitliches Zeitmanagement unerlässlich. Zu den kritischen Erfolgsfaktoren zählen: die Einhaltung von Terminen, die Umsetzung von Qualitätsvorgaben, die Erfüllung des vereinbarten Leistungsumfangs, die Zufriedenheit aller Beteiligten. Um ein effektives Zeitmanagement in den Projekten zu erreichen, sollte man die folgenden Tipps beachten: Vereinbaren von Spielregeln mit dem Projektteam; die umfangrei-

chen komplizierten Aufgaben schrittweise erledigen; die ergebnisorientierte, effektive Durchführung von Besprechungen; die sinnvolle Koordination von Terminen zwischen den Projekt-Beteiligten. Aktivitäten werden an klaren, messbaren Zielformulierungen ausgerichtet. Eine regelmäßige Planung und die Überprüfung der Ergebnisse sind unerlässlich. Hierfür können Werkzeuge wie Zeitmanagementsysteme oder auch elektronische Tools genutzt werden. Das Prinzip der Schriftlichkeit ist sehr hilfreich, weil man auf diese Weise den Überblick über das ganze Projekt haben kann. Es lohnt sich also, sich kritisch mit den Vorgehensweisen und Einstellungen in Bezug auf den Umgang mit der Zeit und sich selbst auseinanderzusetzen [52].

2.8 Zeitverhalten in Echtzeitsystemen

In der heutigen modernen Gesellschaft findet man eine große Anzahl von eingebetteten Systemen. Von diesen müssen viele Echtzeitanforderungen erfüllen [12, 63]. Vor allem im Bereich der Automobilindustrie müssen die Systeme nicht nur die richtigen Ergebnisse berechnen, sondern diese auch zur richtigen Zeit bereitstellen. Daher ist es eine Notwendigkeit das Zeitverhalten von Echtzeitsysteme zu analysieren, zu verstehen und zu optimieren. Die Optimierung kann in verschiedenen Bereichen passieren. Für eingebettete Systeme aus dem Automotive-Kontext sind vor allem Optimierungen im Bereich der Hard- bzw. Software wichtig. Auf der Hardwareebene wird vor allem die statische Timing-Analyse verwendet [75, 31]. Auf der Softwareebene kann das Timing vor allem durch die Wahl der Art des Echtzeitsystems, zeitgesteuert bzw. ereignisgesteuert, optimiert werden [62, 61, 42].

2.9 FPGA: Field Programmable Gate Array

Ein FPGA [13] ist eine anwenderprogrammierbare integrierte Schaltung. Sie besteht im Wesentlichen aus einer Matrix von Logikzellen, mit deren Hilfe beliebige Logikfunktionen realisiert werden können. Die Logikzellen werden in modernen FPGAs durch Flip-Flops auf SRAM-Basis hergestellt. Somit ist der FPGA-Baustein vom Anwender konfigurierbar, verliert aber bei Spannungsverlust die Konfiguration und muss erneut „programmiert“ werden. Mittlerweile gibt es auch teurere, auf Flash-Speicher basierende FPGAs [26], die ihre Konfiguration permanent speichern können. Alle Logikzellen sind durch ein sehr komplexes und ebenfalls konfigurierbares Verbindungsnetzwerk verbunden. Weiterhin findet man Ein-/Ausgabeblöcke, RAM-Bausteine, Multiplizierer und Taktgeber auf dem FPGA-Chip [28]. Bei kleinen Stückzahlen sind FPGAs sehr kostengünstig gegenüber ASICs (Anwendungsspezifische Integrierte Schaltung). Allerdings ist die erreichbare Taktfrequenz

geringer und auch die Verlustleistung von FPGAs ist größer als die eines ASICs [13].

2.10 VHDL

VHDL („Very High Speed Integrated Circuit Hardware Description Language“) ist eine Hardwarebeschreibungssprache. Mit VHDL ist es möglich große und komplizierte digitale Systeme zu entwickeln [34]. Der Entwurf einer Schaltung mit VHDL entspricht einem Top-Down-Prozess. Zuerst modelliert man das gewünschte Verhalten einer Schaltung auf Verhaltensebene und erstellt ein VHDL-Modell. Danach wird dieses Modell auf Register-Transfer-Ebene [35] beschrieben. Ist das VHDL-Modell nun vollständig spezifiziert, so kann es mit einem Synthese-Werkzeug synthetisiert werden. Dabei sucht das Synthese-Werkzeug aus einer vorhandenen Hardware-Bibliothek die entsprechenden Gatter und Flip-Flops heraus, um das Modell nachzubilden. Ergebnis der Synthese ist eine Netzliste, die wiederum in einen Bitstream umgewandelt werden kann, der dann direkt in einen FPGA (Field Programmable Gate Array) oder CPLD (Complex Programmable Logic Device) geladen werden.

2.11 Entwicklungsumgebung Xilinx ISE

Dieser Vortrag gab eine kurze Einleitung in den Aufbau und den Umgang mit der Xilinx-Entwicklungsumgebung „ISE“ [11]. Es wurde zu Beginn auf das sogenannte „Xilinx Design Flow“ eingegangen, dies ist der komplette Ablauf einer Projektentwicklung, wie er von den Xilinx-Tools unterstützt wird. Im weiteren Verlauf wurden die einzelnen Tools zur Synthese und Implementation genauer betrachtet, und welche Dateiformate eine Rolle spielen. Es wurde auf die Definition der User Constraints [8] sowie den Umgang mit dem ISE-Simulator eingegangen. Neben der Arbeit mit der Benutzeroberfläche „Project Navigator“ wurde auch ein Augenmerk auf die Verwendung der Tools direkt über die Kommandozeile gelegt. Zuletzt wurde kurz auf die Möglichkeiten der Programmierung eines FPGA eingegangen.

2.12 Struktur von Prozessoren in VHDL

Der Vortrag „Struktur von Prozessoren in VHDL“ beschäftigte sich mit der Modellierung einer 8-Bit-RISC-CPU [53] in VHDL-Code [57]. Es wurde zunächst der grundlegende Aufbau einer CPU und deren Architektur beschrieben [51]. Anschließend gab es eine Einführung in VHDL-Code [57], welche anhand von Beispielen verdeutlicht wurde. Das Hauptaugenmerk der Ausarbei-

tung beschäftigte sich mit dem 8-Bit-Mikrocontroller von Dimo Pepelyashev [1], der unter anderem auch auf der opencores-Website gefunden werden kann [20]. Der Controller [19] ist zwar erst im Alpha-Status und besitzt noch viele „Kinderkrankheiten“, bzw. es fehlt quasi die komplette Peripherie, trotzdem eignet er sich aufgrund seiner einfachen Struktur und Übersichtlichkeit gut, um sich mit dem Stoff vertraut zu machen und Anfangswissen zu vertiefen. Abschließend wurde noch ein kurzer Ausblick auf mögliche Anwendungsgebiete, wie z. B. einem FPGA, gegeben [27].

2.13 CAN-Bus

Der CAN-Bus (Controller Area Network) [9] hat seit der Erfindung in den 80er Jahren des letzten Jahrhunderts viele Anwendungsbereiche, wie z. B. in der Automatisierungs-, Medizin-, der Flugzeug- und Raumfahrttechnik, aber vor allem in der Automobilindustrie, gefunden. In der Automobilindustrie spielt der CAN-Bus die zentrale Rolle in der Kommunikation zwischen Steuergeräten und Aktoren. Dies begründet sich u. a. in der hohen Störunanfälligkeit und den zahlreichen Methoden zur Fehlererkennung und -korrektur [10]. Daher wird er zum Großteil in Bereichen des Fahrzeuges eingesetzt, wo es auf Zuverlässigkeit ankommt, wie zum Beispiel beim Airbag oder ABS. Aber auch dort, wo schnelle Kommunikation erforderlich ist, findet sich der CAN-Bus wieder. Man kann ihn grob in zwei Kategorien einteilen, zum einen die Low-Speed- (CAN 2.0A) und zum anderen die High-Speed-Variante (CAN 2.0B) [41]. Bei CAN 2.0B wird ein Teil der Störungsunanfälligkeit zugunsten der Geschwindigkeit (bis zu 1MBit/s) eingebüßt. Beide CAN-Varianten kommunizieren jedoch über ein Twisted-Pair-Kabel, wobei eine Ader als CAN_HIGH und die andere als CAN_LOW bezeichnet wird. Sie führen das jeweils komplementäre Signal, so dass die Information anstelle vom Pegel von der Differenz der beiden Signale getragen wird, man spricht hier von einem differentiellen Signal.

Der Vortrag befasste sich außerdem auch noch mit Varianten des CAN-Bus, wie dem Time-Triggered-CAN (TTCAN), der Echtzeitverhalten sicherstellt, oder dem Single-Wire-CAN [50]. Darüber hinaus wurden die Ankopplung von CAN-Controllern an den Bus, Buszugriffsverfahren, sowie der Nachrichtenaustausch, Signalkodierung und Methoden zur Fehlererkennung behandelt [50]. Auch andere Bussysteme, die zum Teil in der Komfortelektronik im Automobilbereich eingesetzt werden, wurden angesprochen.

3 Evaluation der verfügbaren Soft-Cores

Im Rahmen einer Evaluationsphase haben wir uns verschiedene zur Verfügung stehende Soft-Cores angesehen und anschließend entschieden, welche Soft-Cores sich für unser Vorhaben eignen könnten. Dafür wurden sowohl die Hardwareeigenschaften der Prozessoren, wie auch die Codequalität der Dateien, in denen die Cores vorlagen, beurteilt.

Das folgende Kapitel gibt einen Überblick über die evaluierten Soft-Cores und ihre Eigenschaften.

3.1 ARM

ARM bietet eine große Anzahl von 32-Bit-RISC-CPUs an. Allerdings werden diese nicht als Chip, sondern als Intellectual Property (IP) verkauft. Diese liegen in Form von Hardwarebeschreibungen vor und für deren Nutzung müssen Lizenzgebühren gezahlt werden. Neben den eigentlichen CPU-Cores stellt ARM auch eine Reihe von Peripheriekomponenten ebenfalls als IP zur Verfügung. Eingesetzt werden sie vor allem bei rechenintensiven Prozessen wie Audio- und Videoapplikationen, oder in PDAs, Spielkonsolen, bis hin zu Navigationsgeräten. Im Automotive-Bereich kommen sie außerdem in Steuergeräten für ABS oder im Antriebsstrang zum Einsatz. Uns standen drei Softcores der ARM7-Familie zur Evaluation zur Verfügung. Konkret wurden folgende Implementierungen begutachtet:

- nnARM [71]
- core_arm [45]
- arm_core [73]

nnARM

nnARM steht für *Not aN ARM* und lässt direkt vermuten, dass die hier verwendete Architektur zwar den ARM-vt4-Instruktionssatz der ARM7-Familie nachbildet, aber sonst nicht viel mit den Originalen zu tun hat. Zusätzlich soll der Thumb-Instruktionssatz unterstützt werden. Umgesetzt ist der ARM-Core mittels einer vierstufigen Pipeline. Bis jetzt sind noch nicht alle Befehle implementiert worden, darunter auch die für den Coprozessor. Die Speicheranbindung ist auf das Einfachste reduziert worden und müsste für unsere Zwecke wahrscheinlich komplett neu entworfen werden. Die CPU ist sehr modular in einzelne Dateien aufgeteilt und daher recht übersichtlich gestaltet. Als Hardware-Beschreibungssprache wurde hierbei jedoch Verilog verwendet,

was zusätzliche Arbeit bedeuten würde, da man sich ständig von VHDL- auf Verilog-Implementierungen und umgekehrt umstellen müsste. Leider scheint die Weiterentwicklung des nnARM auf Grund von Schwierigkeiten mit der Firma ARM nicht vorangetrieben zu werden. Weiter fällt negativ auf, dass keinerlei Peripherie vorhanden zu sein scheint. Insgesamt ist der nnARM auf 40 Quelldateien aufgeteilt und umfasst 11.802 Zeilen Code mit nur 2379 Kommentaren. Allerdings gibt es eine Dokumentation dazu, in welcher die wesentlichen Ideen und Implementierungsdetails näher erläutert werden.

core_arm

Der core_arm bildet keinen konkreten ARM-Prozessor nach. Er soll den ARM-Instruktionssatz implementieren, wobei nicht ganz klar ist, welchen. Leider ist die Instruction-Unit nicht komplett fertig, daher unterstützt auch dieser Softcore noch nicht alle Befehle. Im Gegensatz zum nnARM jedoch ist das Interface zum Coprozessor bereits funktionsfähig. Die Entwickler haben für die Peripherie die frei erhältliche Implementierung des Sparc-Prozessors LEON zu Hilfe genommen. So werden Teile der Peripherie des LEON übernommen, unter anderem der Speicher und die UART-Schnittstelle. Ohne die LEON-Quelldateien umfasst der core_arm 13025 Zeilen Code von denen etwa 3084 Kommentare darstellen. Auch dieser Softcore ist modular aufgebaut und auf 61 Dateien verteilt. Mit allen Dateien des LEON kommen die 525 Dateien auf über 70.000 Zeilen Quellcode. Neben den eher dürftigen Kommentaren kommt negativ hinzu, dass zum Teil generierter Code benutzt wurde, was die Lesbarkeit deutlich erschwert. So werden für Signale und Variablen oft kryptische Namen verwendet. Die Tatsache, dass der LEON zum Teil in diesem Projekt mit eingebettet ist, macht die Verwendung dieses Softcores nicht einfacher. Leider fehlt hier eine Dokumentation.

arm_core

Das arm_core-Projekt hat sich mit dem ARM7TDMI-S Prozessor beschäftigt. Auch hier mussten wir feststellen, dass nicht alle Befehle des ARM-Instruktionssatzes umgesetzt wurden. So fehlt der Thumb-Decoder komplett, und das Interface für den Coprozessor ist unvollständig. Ebenso muss beim Speicher-Interface noch einiges an Arbeit geleistet werden. Laut Autor befinden sich auch noch zahlreiche Bugs in der Implementierung, so dass dieser Softcore noch nicht sinnvoll eingesetzt werden kann. Jedoch ist er als einziger der vorgestellten ARM-Softcores ohne großen Aufwand synthetisierbar. Allerdings benötigt dieser ca. 66% der auf einem Spartan 3S500E verfügbaren LUTs. Insgesamt ist der arm_core übersichtlich und modular aufgebaut, mit

insgesamt 44 Dateien und 7106 Zeilen Code (2206 Kommentarzeilen). Nach einer Dokumentation sucht man auch hier vergebens.

Compiler- und Betriebssystemsupport

Für die ARM-Prozessoren gibt es viele Compiler, unter anderem auch eine GNU-Toolchain. Neben freien Tools gibt es auch ein großes Angebot an kommerziellen Entwicklungsumgebungen für die unterschiedlichen Prozessorfamilien. Mit der 32-Bit-Architektur gibt es zu dem auch eine Menge Betriebssysteme, die ARM unterstützen: Windows CE, Linux, FreeRTOS, PalmOS, RISC OS und einige mehr.

Fazit

Unterm Strich macht keiner der Softcores den Eindruck, dass sie für unsere Zwecke geeignet sind. Die Implementierungslücken würden einen nicht unerheblichen Aufwand nach sich ziehen. Ohne Peripherie und durch zum Teil kryptischen Quellcode ist die Entwicklung eines System-on-a-Chip im Rahmen dieser Projektgruppe daher nicht möglich. Erschwerend kommt die 32-Bit-Architektur hinzu, die möglicherweise zu viel Platz auf einem FPGA verbraucht, als dass wir damit effektiv arbeiten könnten.

3.2 Atmel AVR

Atmel bietet in der AVR-Familie einige 8-Bit-Mikrocontroller an. Es handelt sich dabei um RISC-Prozessoren mit 52 bis 135 Befehlen, die überwiegend einen Taktzyklus benötigen. Die Modelle haben alle 32 General-Purpose-Register und bis zu 256 KB Flash-Speicher. Programm- und Datenspeicher sind getrennt, es handelt sich also um eine Harvard-Architektur. Es gibt einige Vertreter mit USB-, CAN- und LIN-Controllern. Der Aufbau ist vergleichsweise übersichtlich und bietet daher auch Hobby-Anwendern einige Möglichkeiten. Die folgenden vier Softcores wurden von uns untersucht:

- AVRtinyX61core [56]
- pAVR [43]
- AX8 [76]
- AVR_Core [65]

AVRtinyX61core

Der AVRtinyX61core ist, wie der Name schon sagt, recht klein und würde nur 17% der auf dem 3S500E verfügbaren LUTs benötigen. Jedoch fehlen der Implementierung sämtliche Load-/Store-Befehle, so dass hier einiges an Zusatzaufwand nötig wäre, um sinnvolle Anwendungen zu implementieren. Der VHDL-Code ist zwar ausreichend mit Kommentaren versehen (diese machen ein Drittel der Zeilen aus), besteht jedoch nur aus einer Entity. Diese fehlende Modularisierung würde die Arbeit weiter erschweren. Die Peripherie fehlt leider ebenfalls.

pAVR

Der pAVR hat als hervorstechendes Merkmal eine 6-stufige Pipeline. Dadurch ist er relativ groß und benötigt fast die Hälfte (46%) der LUTS des S500E. Damit ist er für ein später angedachtes Network-on-Chip zu groß. Die Pipeline ist vor allem für Berechnungen entwickelt, bei Sprüngen oder auftretenden Interrupts wird sie komplett geleert. Diese Eigenschaft macht den Core eher ungeeignet für unsere Versuche. Die Dokumentation ist zwar recht ausführlich, jedoch bildet der pAVR keinen speziellen AVR nach.

AX8

Der AX8 ist leider nicht direkt kompilierbar. Er simuliert ältere AT90S2313 / AT90S1200, die nicht im Automobilbereich verwendet werden. Der AX8 bietet zumindest Peripherie wie Timer, Ports und UART an. Der VHDL-Code ist jedoch äußerst unübersichtlich und enthält sehr wenige Kommentare.

AVR_Core

Der AVR_Core erscheint auf den ersten Blick sehr übersichtlich. Die einzelnen Elemente sind sehr gut modularisiert (insgesamt 8 Verzeichnisse mit 45 Dateien). Von den 9870 Zeilen VHDL-Code sind 2443 kommentiert und lassen sich so recht gut verstehen. Auf dem 3S500E benötigt der Kern (ohne Peripherie) 20 Prozent der vorhanden LUTs. Der AVR_Core bringt auch einiges an Peripherie wie Timer, Ports, UART und externe Interrupts mit. Er simuliert den ATmega103, der dem ATmega128 ähnelt, welchen es auch mit einem CAN-Controller gibt (AT90CAN128 [38]).

Compiler- und Betriebssystemsupport

Die Entwicklung für die AVR-Controller wird vom Hersteller Atmel durch die Entwicklungsumgebung AVR-Studio unterstützt. Darüber hinaus gibt es eine GNU-Toolchain, die sehr viele Modelle unterstützt und die Entwicklung in der Hochsprache C herstellt. Lauffähig auf den kleinen Controllern sind einige Betriebssysteme, u. a. eCos und FreeRTOS.

Fazit

Wir entschieden uns für den AVR_Core: Er ist gut dokumentiert und modularisiert. Weiter arbeitet er mit einem identischem Zeitverhalten im Vergleich zum richtigen ATmega103 und bietet dank der Peripherie eine gute Basis, um weiter damit arbeiten zu können.

3.3 Freescale 9S12

Der Freescale 9S12 [18] ist ein beliebter 16-bit-Mikrocontroller von Freescale, der in Packages mit umfangreicher, auf dem Chip integrierter Peripherie verfügbar ist. So sind neben den auf dem Chip integrierten Speichern, die bis zu 128KB Flash-ROM und bis zu 4KB RAM umfassen, viele zur Kommunikation mit der Außenwelt erforderliche Peripheriemodule integriert. Insbesondere die integrierte CAN-Bus-Schnittstelle macht ihn für den Einsatz in eingebetteten Systemen im Automobilbereich interessant.

Des weiteren hat die Vorgängerprojektgruppe AutoLab [21] den Controller in mehreren Anwendungsgebieten (u. a. Highspeed-Lowspeed CAN-Gateway) verwendet. Auch die der Projektgruppe zur Verfügung stehenden Dachmodule des BMW 3er haben einen 9S12 integriert. Auch bei dieser Architektur wurden die bereits zur Verfügung stehenden Softcores evaluiert. Konkret wurden folgende Implementierungen begutachtet (alle auf [20] zu finden):

- 68hc08 – Community-Projekt
- System11 – Community-Projekt
- Gator uProcessor – University of Florida

68hc08

Schon am Namen dieses Softcores erkennt man, dass dieser statt einem 9S12 seinen frühen Vorgänger 68hc08 nachbildet. Dies stellt natürlich durch das

Fehlen wichtiger Register (z. B. den 16-Bit-Registern D und Y) sowie vieler Befehle (insbesondere alle 16-Bit-Befehle) ein erhebliches Problem dar. Die zum 9S12 fehlende Funktionalität müsste komplett nachimplementiert werden, was einen begehbaren Weg darstellen würde. Weitere Schwächen liegen jedoch bei dieser Implementierung in der Codequalität. Der komplette Quellcode ist sehr schlecht strukturiert, was sich u. a. dadurch bemerkbar macht, dass die gesamte Funktionalität sowohl nur in einer VHDL-Entity, als auch in nur einem VHDL-Prozess implementiert ist. Auch ist der 2429 Zeilen lange Quellcode mit 339 Kommentarzeilen (13%) nur sehr dürftig kommentiert. Weiterhin erschwerend kommt hinzu, dass keinerlei Peripherie mitgeliefert wird. Versucht man, den 68hc08 auf einem Spartan 3S500-FPGA zu synthetisieren, bemerkt man schnell, dass er mit seinen 3454 LUTs im Vergleich zu den anderen Softcores sehr viel FPGA-Fläche (37%) verbraucht.

System11

Das System11 ist dem 68hc08 in Bezug auf unseren Anwendungszweck (die Nachbildung eines Freescale 9S12) insofern voraus, dass er zumindest schon einmal die Funktionalität eines Freescale HC11 nachbildet, welcher der direkte Vorgänger des 9S12 ist. So stehen anders als beim 68hc08 hier schon der 16-Bit-Akkumulator D und das zweite Indexregister Y zu Verfügung. Auch beim Befehlssatz gibt es im Vergleich zum 9S12 nur wenige Unterschiede, sofern man über die fehlenden Fuzzy-Logic-Befehle hinweg sieht. Negativ fiel hier jedoch die nur halb fertige Implementierung auf. So werden z. B. die Divisionsbefehle und Bitoperationsbefehle noch nicht unterstützt. Dafür kommt das System11 mit einem Peripherieblock, der u. a. über einen UART und General-Purpose-IO-Ports verfügt. Außerdem ist der Quelltext relativ gut strukturiert. Auch hier ist der ganze Prozessor in nur einer Entity implementiert, welche jedoch in gut zu überblickende Teilprozesse aufgeteilt ist, die jeweils einem der bekannten typischen Funktionsblöcke von CPUs entsprechen. So befinden sich die ALU, das Register-Array, usw. jeweils in eigenen Prozessen. Der Quelltext des reinen Cores umfasst 4793 Zeilen, die mit 849 (17%) Kommentaren versehen sind, und sich auch subjektiv besser lesen und verstehen lassen als die des 68hc08. Die synthetisierte Größe von 1712 LUTs (17% des S3E500) verwundert zunächst, da es sich um einen größeren Controller handelt, erklärt sich jedoch vor allem durch die fehlende Divisionseinheit.

Gator uProcessor

Der an der University of Florida entwickelte Gator uProcessor macht den besten Eindruck. Dies liegt zum einen an der hervorragenden Modularisierung (einzelne VHDL-Entities für alle Funktionsblöcke), zum anderen an der sehr guten Dokumentation. Das Design ist Thema mehrerer Diplomarbeiten, und der Softcore wird produktiv in der Lehre eingesetzt. Er bildet den vollen Funktionsumfang eines HC11 nach und schießt dabei sogar ein wenig über das Ziel hinaus, indem zumindest intern bereits 16 Bit breite Datenpfade verwendet werden. Der bei anderen Softcores einen Großteil des Quelltextes ausmachende Microstate-Sequenzer fehlt hier zu Gunsten einer mikroprogrammierbaren State-Machine. Diese arbeitet den vorher mit dem ebenfalls beiliegenden Microcode-Assembler generierten Microcode ab. Das hat den Vorteil, dass statt teurer Lookuptables hier Block-RAMs für den Microcode verwendet werden können. So benötigt der reine Core in synthetisiertem Zustand auch nur 1385 LUTs (14% des S3E500), also ohne die ebenfalls mitgelieferte reiche Peripherie. Diese besteht u. a. aus UART, General-Purpose-IO-Ports, Sieben-Segment-LED-Ansteuerung, Interruptcontroller und Bus-Switch und ist in Grenzen mit der ursprünglichen HC11-Peripherie kompatibel.

Compiler- und Betriebssystemsupport

Es existiert ein reichhaltiges Angebot an Compilern und Betriebssystemen für den 9S12. So steht neben der GNU-Toolchain (bestehend aus C-Compiler, Assembler, Linker, Debugger usw.) auch eine ganze Reihe kommerzieller Compiler zur Verfügung. Auch das Betriebssystemangebot ist umfangreich. Hier sind u. a. Free RTOS, CMX RTOS, eCos und RTA-OSEK zu nennen.

Fazit

Den besten Eindruck machte der Gator uProcessor, insbesondere wegen der guten Dokumentation und des modularen Aufbaus. Leider stellte sich heraus, dass ein Umbau zur 9S12-Kompatibilität bei allen untersuchten Cores den Rahmen der Projektgruppe sprengen würde. Wir haben uns deshalb dazu entschlossen, mit dem Gator uProcessor als HC11-Clone weiterzuarbeiten.

3.4 8051-Familie

Die ersten Modelle der 8051-Prozessorfamilie kamen 1980 auf den Markt und zählen seitdem zu den verbreitetsten Mikrocontrollern der Welt. Der interne Aufbau des Rechners entspricht der sogenannten CISC-Architektur, was

damals die gängige Art und Weise war, Prozessoren zu bauen. Daten- und Programmspeicher sind separat, was das gleichzeitige Schreiben von Ergebnissen und Lesen von Befehlen ermöglicht (Harvard-Architektur). Die Größe des Datenspeichers reicht von 128 bis 256 Byte, die des Programmspeichers auf einigen Derivaten bis 64 Kb. Die 8051er-Serie ist eine 8-Bit-Architektur. Es existieren allerdings einige Ableger, die eine 16-Bit-Architektur besitzen.

Diese Controllerfamilie ist sehr verbreitet in Tastaturen, USB-Webcams, Taschenrechnern und einigen Spielekonsolen der 80er Jahre. Sie wurde in den letzten drei Jahrzehnten in großen Stückzahlen hergestellt und ist daher sehr günstig zu erwerben.

Ebenso existieren auch viele Softcores, von denen wir folgende drei untersucht haben:

- oc8051 von Simsic/Teran [20]
- i8051 Dalton-Projekt der UCLA
- t51 (8052) von Voggeneder/Wallner [20]

oc8051

Der oc8051 befindet sich zur Zeit noch im Alphastadium. Es funktioniert nur der Basis-Core, da Teile des Prozessors noch nicht implementiert worden sind. Ein Beispiel hierfür wäre die vorzeichenbehaftete Addition. Der Softcore beinhaltet die komplette Peripherie. Es existiert eine ausführliche Spezifikation mit Erklärungen für die einzelnen Befehle. Weiterhin ist der Code gut kommentiert, obwohl es sich um einen Opensource-Core handelt. Da der Code modular aufgebaut ist, ist er auch sehr gut strukturiert und zusätzlich noch innerhalb der einzelnen Dateien sehr übersichtlich. Die „Lines of Code“ betragen ungefähr 12700, wobei davon 2300 Kommentarzeilen sind. Allerdings ist der Code in der Sprache Verilog verfasst.

i8051

Der i8051 der UCLA ist ebenfalls nicht vollständig implementiert. Es fehlt zum Beispiel das Interrupt-Handling und große Teile der Peripherie. Der Softcore enthält allerdings externen RAM und ist für Xilinx-Boards optimiert. Dokumentation ist leider keine vorhanden. Im Quellcode wurden nur die Ports ausreichend kommentiert, der Rest blieb leider ohne erklärende Kommentare. Positiv zu bemerken ist, dass der Code sehr gut strukturiert und

daher sehr übersichtlich ist. Der Core ist modular auf mehrere Dateien aufgeteilt. Er umfasst insgesamt ca. 7650 Zeilen. Der Anteil an Kommentarzeilen ist mit 185 recht gering.

t51

Der t51 ist im „stable"-Stadium und ist vollständig implementiert und lauffähig. Daher umfasst der Code auch die komplette Peripherie. Ebenso ist der Core mit der Xilinx-Software getestet worden und ist kompatibel zu den Spartan-Boards. Auch hier fehlen eine Dokumentation sowie ausführliche Kommentare. Der Code ist eher unübersichtlich gestaltet, verfügt aber über einen modularen Aufbau. Der Core ist mit ungefähr 15000 Zeilen der mächtigste unter den untersuchten Cores, jedoch sind nur 785 Kommentarzeilen vorhanden.

Compiler- und Betriebssystemsupport

Allgemein existiert ein guter Compiler-Support für die 8051 Prozessorfamilie. Neben kommerziellen Compilern von der Firma μVision gibt es auch diverse Open-Source-Compiler. Der Betriebssystemsupport ist ebenfalls durch kommerzielle wie auch kostenlose Echtzeit-Betriebssysteme gewährleistet; zu nennen wären hier Free RTOS, 8051 RTOS von Altium (OSEK) und KR51 Tiny.

Fazit

Der Softcore, der am meisten überzeugte, war der t51. Er ließ sich als einziger ohne Fehler kompilieren, ist dabei vollständig und benötigt nur 1844 LUTs (19%).

3.5 MSP430

Der MSP430 von Texas Instruments ist ein 16-Bit-Mikroprozessor, der besonders in Bereichen eingesetzt wird, in denen eine geringe Leistungsaufnahme von Bedeutung ist. Er realisiert die Von-Neumann-Architektur und besitzt maximal 54kByte adressierbaren Speicher. Der RISC-Befehlssatz umfasst 27 Grundinstruktionen sowie 24 weitere, die mit Hilfe der Grundinstruktionen emuliert werden, darunter auch der Multiply-Accumulate-Befehl.

Da der Prozessor auf geringsten Stromverbrauch optimiert ist, wird er oft in tragbaren Geräten in der Messtechnik und im medizinischen Bereich verwendet. Auch für stationäre Geräte, bei denen die Leistungsaufnahme

eine Rolle spielt (Feuermelder etc.) eignet sich der MSP430. Im Automotive-Bereich ist der Mikroprozessor momentan kaum vertreten.

s430

Mit dem s430 existiert eine frei verfügbare Verilog-Beschreibung des MSP430-Mikroprozessors. Der Status dieses noch nicht abgeschlossenen Projektes ist vom Autor selbst als Pre-Alpha angegeben, also noch in einer frühen Entwicklungsphase. Der Code umfasst 22 Dateien und 110kb. Es existiert keine Dokumentation, und der Quellcode ist sehr dürftig kommentiert. Der Code enthält den Prozessor selbst, jedoch keinerlei Peripherie. Diese müsste von der Gruppe selbst hinzugefügt werden.

Der Projektbeschreibung zufolge ist der Core mit der MSPGCC-Toolchain getestet und bereits auf einem Spartan-3 synthetisiert worden. Laut Autor belegt der Core 20% auf einem XC3S400 FPGA. Diese 1613 benötigten LUTs entsprechen etwa 17% auf unseren XC3S500.

Weiterhin existiert eine Vielzahl von kompatiblen Compilern (MSPGCC-Toolchain, IAR Embedded Workbench, HI-TECH MSP430 C etc.) und Betriebssystemen (Salvo, CMX RTOS, EmbOS, FreeRTOS).

Fazit

Trotz einiger Argumente, die für diesen Core sprechen, wie z. B. die Auswahl an Compilern und Betriebssystemen, sowie die erfolgreiche Synthese auf einem Spartan-3, hat sich die PG dagegen entschieden ihn zu verwenden.

Ein Grund dafür ist die Spezifikation in Verilog. Unsere PG hat VHDL als Beschreibungssprache gewählt, so dass der Code erst in VHDL hätte übersetzt werden müssen. Dies funktioniert nach den Erfahrungen der Betreuer nicht immer reibungslos. Ein weiterer Grund ist der frühe Entwicklungsstatus des Projektes. Die Quellen sind seit einigen Jahren nicht mehr bearbeitet worden und der Autor selbst gibt den Stand des CPU-Cores als „mostly complete“ an. Es sind sonst weder Informationen noch eine Dokumentation zu diesem Projekt zu finden. Das Risiko war daher zu groß und wir haben uns gegen diesen Core entschieden.

Kurze Zeit später war es möglich, den Autor zu kontaktieren. Dieser gab an, dass er die Arbeit an dem Projekt eingestellt hat. Die CPU sei zwar funktionsfähig, doch habe er die Testphase nie komplett abgeschlossen.

4 Grundlagen des Entwicklungsprozesses

Das nun folgende Kapitel wird sich mit den Grundlagen zur Entwicklung eines System-on-a-Chip (SoC) beschäftigen. In dem Abschnitt 4.1 wird die Compiler-Werkzeugkette für vier verschiedene Prozessoren (AVR, 8051, hc11 und hc08) vorgestellt und hinsichtlich der Nutzung erläutert. Zentral ist hier die Fragestellung, wie man von einem C-Programm zu einem kompilierten Programm in Maschinencode gelangt, das auf der Zielhardware direkt ausführbar ist.

In dem darauf folgenden Abschnitt 4.2 wird die FPGA-Werkzeugkette vorgestellt. Hier geht es darum, wie man von VHDL-Code zur gewünschten Hardwareimplementierung auf einem FPGA-Board gelangt. Dies wird jeweils für die vier oben genannten Softcores an konkreten Beispielen festgemacht.

Da es sich bei der Xilinx-Software nicht um die einzige Möglichkeit handelt, digitale Schaltungen auf VHDL-Basis zu simulieren, wird eine Alternative, bestehend aus GHDL und GTKWave, vorgestellt. Es wird ausführlich auf die Funktionsweise und die Benutzung der beiden Komponenten eingegangen, die jeweils den Vorteil haben, unter GNU General Public License (GPL) zu stehen.

Außerdem wird erläutert, wie die in den meisten Mikrocontrollern bereits integrierten Speicher (RAMs bzw. ROMs) mit Hilfe von Block-RAMs, die auf dem von uns verwendeten FPGA-Board Xilinx Spartan 3E-500 integriert sind, implementiert werden können, und warum dies sinnvoll ist.

4.1 Die Compiler-Werkzeugkette

Im Folgenden soll erläutert werden, wie ein C-Programm in einen äquivalenten VHDL-Programmspeicher umgewandelt werden kann. Voraussetzung dafür ist jeweils ein syntaktisch korrektes Programm. Es werden die nötigen Programme, ihre jeweils genutzten Versionen sowie die passenden Optionen für jeden verwendeten Softcore beschrieben.

4.1.1 AVR-Werkzeugkette

Um C-Programme für den AVR zu kompilieren, nutzen wir den avr-gcc [6]. Als Eingabe benötigt dieser mindestens die Zielplattform (-mmcu=atmega103) und das Programm (demo.c). Die Option -Os optimiert hinsichtlich der Programmgröße, -g fügt Debuginformationen zum Dekompilieren und Debuggen hinzu. Diese Informationen sind optional und werden auch nicht auf das FPGA übertragen. -c verhindert das sofortige Linken, da evtl. später mehrere Dateien zusammengelinkt werden müssen.

```
avr-gcc -g -Os -mmcu=atmega103 -c demo.c
```

Die entstandene Objektdatei (demo.o) bindet man wie folgt zusammen:

```
avr-gcc -g -mmcu=atmega103 -Wl,-Map,demo.map -o demo.out demo.o
```

Dabei gibt `-Wl,-Map,demo.map` an, dass eine Datei mit zusätzlichen Informationen als Ausgabe gewünscht ist.

Nun hat man eine ausführbare elf-Datei (demo.out). Diese muss man noch in das Zielformat umwandeln, das die AVR-Controller erwarten. Folgender Befehl leistet dies und liefert eine hex-Datei (rom.hex):

```
avr-objcopy -O ihex demo.out rom.hex
```

Das hex-Format beinhaltet hauptsächlich den auszuführenden Binärcode in seiner hexadezimalen Repräsentation. Für den normalen Umgang mit AVR-Controllern wäre man hier bereits fertig. Da aber der Controller auf dem FPGA simuliert wird, fehlt noch der letzte Schritt. Dieser erfolgt, wie bei dem T51-Projekt, mit Hilfe von hex2rom, welches in Abschnitt 4.2.3 erklärt wird.

Die genutzten Programme avr-gcc und avr-objcopy findet man in den Paketen avr-gcc (Version 4.1 oder 4.3) und binutils-avr (Version 2.16 oder 2.18). Weiter benötigt man zum komfortablen Programmieren noch die avr-libc [39] (Version 1.4.5 oder 1.6.2).

4.1.2 T51-Werkzeugkette

Das Programmieren von 8051/8052-Mikrocontrollern ist dank entsprechender Software in einer Hochsprache wie C ohne Probleme möglich. Wir verwenden hier die Software μVision [2] in der Version 3.23 von Keil, die wegen ihrer komfortablen Oberfläche und ihres guten Funktionsumfangs sehr zu empfehlen ist. Die Software unterstützt ein breites Spektrum an Mikrocontrollern und so braucht der Benutzer beim Anlegen eines Projektes nur darauf zu achten, dass der richtige Controller ausgewählt ist. Wenn man sich das Programmieren seines Controllers vereinfachen möchte, ist es sinnvoll, eine Header-Datei zu implementieren, in der den verschiedenen Bit-Adressen, die man zum Ansteuern der Peripherie und der Schnittstellen benötigt, Variablennamen zugeordnet sind. Eine entsprechende Datei findet sich unter: `www.keil.com/dd/docs/c51/reg52.h`

Zusätzlich ist es wichtig, dass man in den Projekteinstellungen im Karteireiter „Output“ die Option „Create Hex File“ aktiviert. Nun wird beim Compilieren im Projektordner eine Datei <Projektname>.hex erzeugt, die mit hex2rom weiter bearbeitet werden muss.

4.1.3 68HC11-Werkzeugkette

Für den 68HC11 benutzen wir den Compiler m68hc11-gcc [17] in der Version 3.3.6-m68hc1x-20060122 im Zusammenspiel mit dem m68hc11-ld Linker aus den GNU-Binutils in der Version 2.18. Mit Hilfe des Standard-Makefiles des GNU-Pakets übersetzen wir unsere C-Programme.

Um dem Compiler die Zielarchitektur mitzuteilen, benutzen wir die Option -m68hc11. -Wall und Wmissing-prototypes schaltet die gängigsten Warnungen ein. Für kleine Programme bietet es sich an -msoft-reg-count=0 und -ffixed-z zu benutzen, so dass keine Softregister verwendet werden. Softregister sind im RAM liegende simulierte Prozessorregister, die für große Funktionen aber durchaus sinnvoll sein können. Die Option -fshort-enums sorgt dafür, dass Enums den Datentyp verwenden, der ausreicht, um diese repräsentieren zu können. Mit Hilfe von -fomit-frame-pointer wird der erzeugte Code nochmals verkleinert, macht das Debuggen aber schwerer. Die letzte Option -mshort zwingt den Compiler dazu, für Integer nur 16 Bit zu verwenden. Fehlen noch Quell- und Zieldatei. Mit -o beispiel.o wird die Zieldatei angegeben. Der Dateiname beispiel.c der Quelle ist nun der letzte Parameter.

```
m68hc11-gcc -m68hc11 -Os -g0 -Wall -Wmissing-prototypes
   -msoft-reg-count=0 -ffixed-z -fshort-enums
   -fomit-frame-pointer -mshort -o beispiel.o beispiel.c
```

Neben einigen Optionen, die auch der Compiler benutzt, übergeben wir dem Linker die Option -m m68hc11elfb, um unser Speicherlayout in der Datei memory.x festzulegen. Mit -defsym _io_ports=0x1000 teilen wir dem Linker mit, ab welcher Adresse die IO-Register im Speicher liegen. Außerdem benötigt der Linker eine Datei vectors.s, die im selben Verzeichnis liegen muss. Wir haben uns hier auch nur der Beispieldatei aus dem GNU-Paket bedient. Sie stellt im wesentlichen die Interrupt-Sprungtabelle dar.

```
m68hc11-ld -m68hc11 -mshort -m m68hc11elfb
   -defsym _io_ports=0x1000 -o beispiel beispiel.o
```

Die ausführbare elf-Datei muss noch in das Zielformat umgewandelt werden. Die benötigte s19-Datei erhalten wir durch Objectcopy mit dem Parameter --output-target=srec. Da die page0-Section in den meisten Fällen überflüssig ist, schneiden wir diese mit --remove-section=page0 und --remove-section=.page0 heraus.

```
m68hc11-objcopy --output-target=srec --remove-section=page0
   --remove-section=.page0 beispiel beispiel.s19
```

Die so erhaltene s19-Datei kann auf einem HC11-Mikrocontroller ausgeführt werden. Der letzte Schritt Richtung FPGA erfolgt bei den Motorola-Softcores nicht mit hex2rom, sondern mit der Software s19conv, die ebenfalls in Abschnitt 4.2.3 beschrieben wird.

4.1.4 68HC08-Werkzeugkette

Für den 68HC08 benutzen wir den SDCC (Small Device C Compiler) [49], der in den Versionen 2.8.0 und 2.8.5 einwandfrei funktioniert. Der SDCC ist ein optimierender Compiler, der nicht nur für die Motorola-Prozessoren gedacht ist. Als Einstellungen muss zuerst wieder die Zielplattform eingegeben werden, danach der Beginn des Codesegments, der dem Anfang unseres ROMs entspricht; zuletzt muss die letzte Adresse des RAMs als Anfangsadresse des Stackpointer festgelegt werden:

```
sdcc demo.c -mhc08 --code-loc 0xe000 --stack-loc 0x03ff
```

Nach diesem Schritt erhält man bereits die gewünschte s19-Datei, die, wie bei dem 68HC11, mit s19conv weiterverarbeitet wird. Daneben entstehen noch folgende Dateien:

- demo.asm, die vom Compiler generierten Assembler-Quellen
- demo.lst, ein Assembler-Listing, durch den Assembler generiert
- demo.rst, ein Assembler-Listing, durch den Linkage-Editor generiert
- demo.sym, ein Symbol-Listing, durch den Assembler generiert
- demo.rel, eine Objektdatei, durch den Assembler generiert
- demo.map, die Memory-Map für das Lademodul, durch den Linker generiert
- demo.mem, eine Datei, in der die Speicherbesetzung eingegeben ist
- demo.s19, das Lademodul in Motorola-S19-Format

4.2 Die FPGA-Werkzeugkette

Ein immer wiederkehrender Prozess in der Arbeit unserer Projektgruppe ist die Spezifikation neuer Komponenten unseres Systems und die schrittweise Implementierung auf die Entwicklungsboards. Hierzu stehen uns die vielfältigen Xilinx-Tools zur Verfügung. Es bringt jedoch auch Vorteile, auf Open-Source-Alternativen zurückzugreifen, um die Synthese, Simulation, aber auch

Erstellung vom Block-RAMs zu vereinfachen. Im Folgenden werden die einzelnen Programme und ihr Zusammenspiel genauer betrachtet.

4.2.1 Xilinx-Werkzeugkette

Dieser Abschnitt gibt eine kurze Einleitung in den Aufbau und den Umgang mit der Xilinx-Entwicklungsumgebung „ISE“. Es wird auf das sogenannte „Xilinx Design Flow“ eingegangen, dies ist der komplette Ablauf einer Projektentwicklung, wie er von den Xilinx-Tools unterstützt wird. Weiterhin werden die einzelnen Tools genauer betrachtet, und welche Dateiformate eine Rolle spielen. Neben der Arbeit mit der Benutzeroberfläche „Project Navigator“ wird auch ein Augenmerk auf die Verwendung der Tools direkt über die Kommandozeile gelegt.

Xilinx bietet neben zahlreichen FPGA-Boards, wie der Spartan- und Virtex-Reihe, auch die Entwicklungsumgebungen dazu an. Hier unterscheidet Xilinx zwischen Logic-, Embedded- und DSP-Design und stellt zu diesen Bereichen passende Toolsammlungen bereit.

Die für uns interessante ISE (Integrated Software Environment) aus den Logic Design Tools ist in der Foundation-Version sowie der Webpack-Version erhältlich. Die Varianten unterscheiden sich nur unwesentlich in ihren Funktionsumfang. Letztere unterstützt nicht alle FPGA-Boards, glücklicherweise aber unser Spartan-3E. Das ISE Webpack steht für jedermann kostenlos als Download zur Verfügung. Aktuell ist ISE in der Version 10.1 erhältlich. Wir verwenden jedoch aus Gründen der Stabilität die Vorgängerversion 9.2.

Xilinx Design Flow Die Entwicklung eines Projektes von der Spezifikation bis hin zur Programmierung verläuft in mehreren Teilschritten. Diese Teilschritte sind in der Regel Spezifikation, Synthese, Mapping, „Placement and Routing“ und Programmierung. Geht man tiefer ins Detail, kommen die unterschiedlichen Tools und Dateiformate ins Spiel.

Wir betrachten hier detaillierter die einzelnen Schritte, die bei der Entwicklung mit Xilinx ISE durchlaufen werden. Das Xilinx Design Flow stellt den Zusammenhang der verschiedenen Tools dar und zeigt, welche Rolle sie in der Entwicklung des Projektes spielen. Im **Development System Reference Guide** [11] finden sich zahlreiche Graphiken, die das Design Flow sehr detailliert darstellen.

Abbildung 2 zeigt ein vereinfachtes Modell des Entwicklungszyklus sowie die Möglichkeiten zur Verifikation.

Project Navigator Wie bereits erwähnt, ist Xilinx ISE eine Sammlung von unterschiedlichen Tools. Einige Tools haben eine graphische Oberfläche,

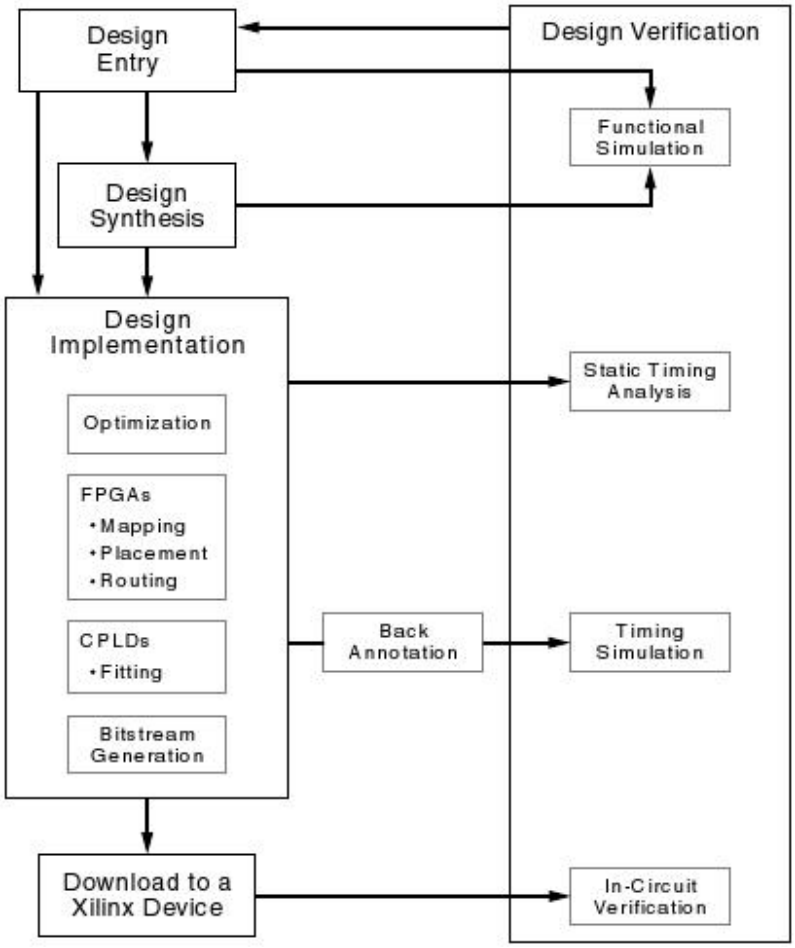

Abbildung 2: Entwicklungszyklus und Möglichkeiten zur Verifikationn, /[11]

andere lassen sich nur mittels Parameter über die Konsole steuern. Jedes Tool kann als eigenständiges Programm gesehen werden, das seine spezielle Aufgabe innerhalb des Design Flows erfüllt. Im „Development System Reference Guide“ werden diese Tools genau beschrieben, ihre Syntax sowie Ein- und Ausgabedateien. Der Benutzer muss sich jedoch nicht zwingend mit den Kommandozeilen-Tools auseinandersetzen. Xilinx bietet mit dem „Project Navigator“ eine Benutzeroberfläche an, die die nötigen Programme bei Bedarf startet. Man muss aber erwähnen, dass der Umgang mit dem Project Navigator eine gewisse Übung erfordert. Durch seine nicht immer intuitive Navigation und zahlreiche Macken stellt er den Benutzer vor einige Herausforderungen.

Spezifikation Der Einstieg in ein Projekt führt über die Spezifikation in einer „Hardware Description Language“ wie VHDL oder Verilog. Der Benutzer hat die Möglichkeit, in einem Editor seine VHDL- oder Verilog-Spezifikationen per Hand zu bearbeiten. Eine Syntax-Prüfung unterstützt ihn dabei. Für automatenbasierte Entwürfe bietet sich das Programm StateCAD an, das aus Zustandsdiagrammen entsprechenden VHDL-Code erzeugt. Dabei entsteht jedoch selbst für einfache Entwürfe sehr viel Code, der hinterher schwer zu bearbeiten ist. Für simple Standard-Konstrukte wie Gatter, Zähler oder Multiplexer bieten die „Language Templates“ eine große Auswahl vorgefertigten Code-Schnipseln an.

Wer das Entwerfen auf einem Schaltplan vorzieht, hat mit dem „Scematic Editor“ die Möglichkeit, aus einer Datenbank von Basiselementen sowie selbst erstellten VHDL-Modulen sein gewünschtes Design zu erstellen. Weiterhin kann der Benutzer komplette Xilinx-IP-Cores (Intellectual Property Cores) von Speicher bis hin zu Prozessoren nutzen, sofern er eine Lizenz erwirbt.

Jeder VHDL-Code muss nach der Spezifikation synthetisiert werden. Dabei wird die sprachliche Beschreibung in eine Beschreibung in Form von Bausteinen wie Gatter, FlipFlops, RAM etc. übersetzt. Externe Synthese-Tools können diese Aufgabe erledigen. Das dabei übliche Format EDIF (Electronic Design Interchange Format) kann dann für die weiteren Schritte gelesen und verwendet werden. Dem Benutzer steht aber auch das „Xilinx Synthesis Tool“ zur Verfügung. XST erzeugt dabei seine eigene Repräsentation, die „Native Generic Circuit File (NGC)“. Weitere Details finden sich im **XST User Guide** [36].

Für welchen Weg der Spezifikation man sich auch entscheidet, dass Resultat ist eine Netzliste, eine Repräsentation in Form von elementaren logischen Komponenten. Solch eine „Native Generic Database“, eine Datei mit der Endung .ngd, wird vom Programm NGDBuild erzeugt. In diese Netzliste fließen außerdem noch Nebenbedingungen ein, die der Benutzer festlegen kann. Diese „User Constraints“ halten Bedingungen für Timing und Layout fest. Sie können per Hand in die „User Constraints File (UFC)“ eingetragen oder mit speziellen Editoren wie PACE und Xilinx Constraints Editor erzeugt werden. Das Festlegen von Layoutbedingungen ist ein notwendiger Schritt, da alle Inputs und Outputs des Designs mit den richtigen PINs auf dem FPGA-Board verbunden werden müssen. Weitere Informationen finden sich im **Constraints Guide** [8].

Implementierung Die Native Generic Database dient als Ausgangspunkt für die Implementierung des Designs. Die Implementierung besteht, wie bereits erwähnt, aus den Schritten Mapping, Placement und Routing.

Mapping bezeichnet das Abbilden der Netzliste auf die elementaren Bausteine eines FPGAs. Hier kommt also zum ersten Mal das konkrete Board ins Spiel. Das hierfür zuständige Tool „MAP“ erstellt eine „Native Circuit Description File“ mit Endung .ncd. Diese fungiert wiederum als Eingabe für das Programm „PAR“, Placement und Routing. Dieser zeitraubende Schritt sucht nach der optimalen Platzierung und Verdrahtung der Bausteine, die das Design auf dem Board benötigt. Ziel ist es hierbei, möglichst wenig Platz auf dem FPGA zu belegen, und gleichzeitig eine kurze Signallaufzeit zu erreichen. Eine vollständig geroutete NCD ist dabei das Endprodukt.

Der letzte Schritt ist das Erzeugen eines Bitstreams. Mit Hilfe des Programms BitGen wird eine Datei (.bit) erzeugt, mit der das FPGA-Board direkt programmiert werden kann. Diese Aufgabe übernimmt wiederum das Programm iMPACT. Mittels einer USB-JTAG Schnittstelle kann das gewünschte Board direkt programmiert werden, verliert jedoch nach einem Neustart sein Gedächtnis. Um das FPGA langfristig zu programmieren, hat der Benutzer die Möglichkeit, den Bitstream ins Flash-PROM zu schreiben. Wird das Board entsprechend gejumpert, programmiert sich das FPGA bei jedem Neustart neu aus dem PROM. Nähere Informationen finden sich im **Spartan-3E Starter Kit Board User Guide** [30].

Simulation Die Verifikation der Designs ist an vielen Stellen der Entwicklung möglich (siehe Abb. 2). Für die funktionale Simulation, also die Simulation des Verhaltens der Logik, stehen innerhalb der ISE zwei Programme zur Verfügung: ISE Simulator und ModelSim. Beide Simulatoren sind ähnlich mächtig. Im Folgenden wird auf den ISE Simulator etwas näher eingegangen.

Die Basis einer Simulation ist ein VHDL-Modul, das mindestens über einen Eingang für das Taktsignal verfügt. Zu Beginn wird das Taktsignal definiert, sowie Details zu High-Time, Low-Time, Setup-Time und Offset. Dieses Taktsignal liegt dann unveränderbar für die Dauer der Simulation an. Die Werte aller anderen Eingänge können zu jedem Zeitpunkt der Simulation geändert werden. Nach Durchführung der Simulation hat der Benutzer die Möglichkeit, die Werte aller Ein- und Ausgänge sowie aller internen Signale zu jedem Zeitpunkt zu lesen.

4.2.2 GHDL/GTKWave – Open-Source-Alternativen

Überblick Für die Simulation digitaler Schaltungen auf Basis von VHDL-Projekten gibt es seit Ende 2006 [16] eine weitere Alternative: GHDL und GTKWave sind zwei dafür geeignete Anwendungen, die unter der GNU General Public License (GPL) stehen. Auf einem Linux-System, das bereits die GTK-Bibliotheken installiert hat, benötigen beide Programme zusammen

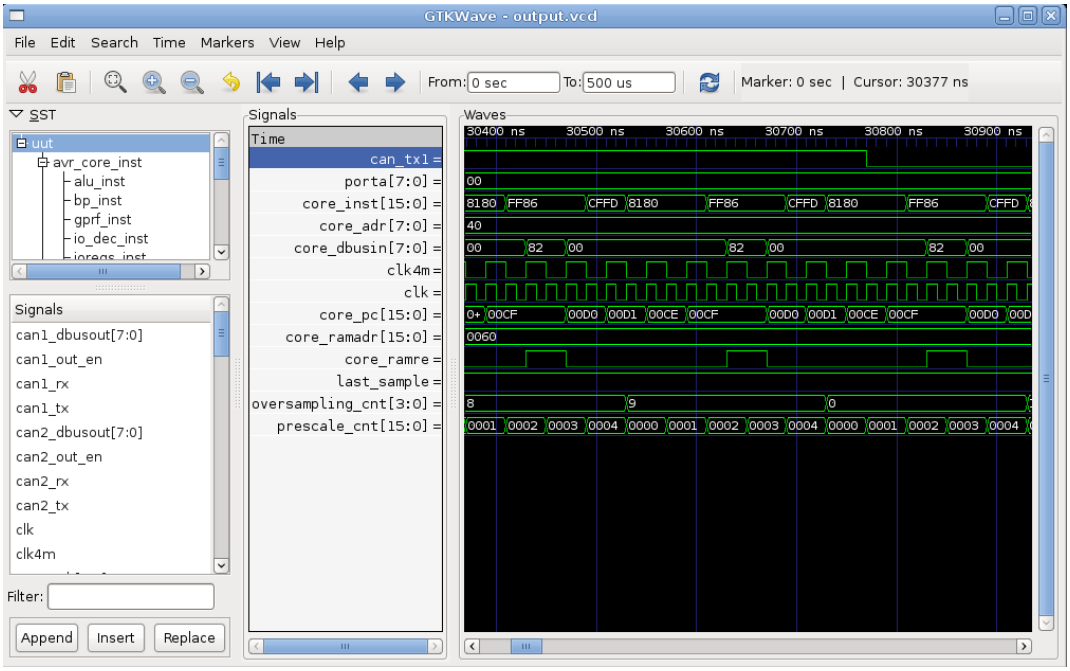

Abbildung 3: GTKWave-Simulation des AVR-Cores

etwa 100 Megabyte. Damit erhält man einen ressourcensparenden VHDL-Simulator im Vergleich zur Xilinx-ISE-Toolsammlung, die mehr als 3 Gigabyte benötigt. Beide Programme sind für Linux, Unix, MacOS und auch für Windows verfügbar.

Funktionsprinzip Um ein digitales VHDL-Projekt zu simulieren, benötigt man die beiden oben beschriebenen Programme. Da VHDL stark an die Programmiersprache Ada angelehnt ist, besteht GHDL im Wesentlichen aus einem umgebauten GNU-Ada-Compiler. Damit lässt sich aus den gewünschten VHDL-Dateien eine ausführbare Binärdatei erstellen, genauso wie man es von jeder beliebigen Programmiersprache kennt. Allerdings soll aus dem VHDL-Code kein Programm im eigentlichen Sinne entstehen, sondern es soll eine digitale Schaltung simuliert werden. Wenn man also die mit GHDL kompilierte Binärdatei ausführt, so wird diese das VHDL-Projekt simulieren und alle Änderungen innerhalb der zu simulierenden Schaltung protokollieren, d. h. es wird eine Protokoll-Datei im VCD Format [33] erstellt, die sämtliche Signale des VHDL-Projektes während der Simulation festhält.

Und hier kommt GTKWave (Abb. 3) ins Spiel, das die von GHDL erstellte Protokoll-Datei graphisch darstellt. GTKWave bietet dabei etwa den gleichen Funktionsumfang wie kommerzielle Simulatoren und lässt sich sehr einfach

bedienen. Weiterhin bietet es ein paar sehr komfortable Funktionen, wie z. B. `Find Next Edge`, welches für ein gegebenes Signal die nächste Änderung findet. Außerdem ist es möglich, die von GHDL erstellte Protokoll-Datei über eine „named Pipe“ zu lesen, d. h. man kann während der Simulation schon Zwischenergebnisse betrachten. GTKWave ist Teil des gEDA-Projektes [14], welches an GPL-Werkzeugen zum Design elektronischer Schaltungen arbeitet. Folglich lassen sich mit GTKWave auch eine ganze Reihe von anderen Dateiformaten darstellen.

Anleitung zur Simulation

Taktquelle erstellen Genau wie der Xilinx ISE-Simulator benötigt man als erstes eine „Test-Bench-Waveform“, d. h. im Wesentlichen eine Quelle für ein Takt-Signal. Was bei ISE per GUI erstellt werden kann, muss man hier selbst von Hand erledigen. Dazu erstellt man eine neue VHDL-Datei, die genau eine `ENTITY` ohne Signale enthält. Die `ARCHITECTURE` dieser `ENTITY` instantiiert dann die toplevel `ENTITY` des eigentlichen VHDL-Projektes, das man simulieren will. Dabei wird einfach die Schnittstelle der toplevel `ENTITY` auf interne Signale „gemappt“ und ein neuer Prozess für das Takt-Signal erstellt, der periodisch den Takt generiert. Die meisten Mikrocontroller haben einen Reset-Anschluss, der auch noch geeignet belegt werden muss.

VHDL-Dateien importieren Jetzt muss man dem GHDL-Compiler mitteilen, welche VHDL-Dateien simuliert werden sollen. Das geschieht mit:

```
ghdl -i *.vhd
```

Der Parameter „-i“ steht für Import und fügt alle übergebenen VHDL-Dateien in einem GHDL-Projekt zusammen. Dieses Projekt wird automatisch in einer Datei „work-obj93.cf“ gespeichert.

Kompilieren und Linken Als nächstes muss ein ausführbares Simulationsprogramm erstellt werden. Das erledigt man mit:

```
ghdl -m --ieee=synopsys -fexplicit soctest
```

Der Name der zu simulierenden `ENTITY` ist dabei „soctest“, in unserem Fall also die in Schritt 1 erstellte `ENTITY`, die den Takt generiert. Der Parameter „-m“ steht für make. Genau wie bei einem Makefile werden dann nur Dateien, die sich seit der letzten Kompilierung verändert haben (sowie davon abhängige Dateien), übersetzt. Das spart Rechenzeit und macht ein externes Makefile überflüssig.

Simulation Jetzt kann man das ausführbare Simulationsprogramm für eine beliebige Zeit laufen lassen. Das funktioniert mit:

```
ghdl -r soctest --vcd=output.vcd --stop-time=100000ns
```

Bei diesem Schritt wird nun das Design bis zu einer Zeit von 100000ns simuliert und es entsteht eine VCD-Datei namens „output.vcd“. Weitere mögliche Optionen erhält man mit „--help“ als Parameter. Der Parameter „--stop-time“ ist optional, man kann ihn auch weglassen und STRG-C drücken, um die Simulation zu beenden.

Anzeige in GTKWave Die bei der Simulation gewonnene VCD-Datei kann nun in GTKWave geladen werden:

```
gtkwave output.vcd
```

Probleme mit GHDL und GTKWave Wenn es Fehler in dem VHDL-Code gibt, so liefert GHDL passende Fehlermeldungen dazu, die beim Debuggen helfen sollen. Leider stimmen in seltenen Fällen die Zeilenangaben zu den Fehlerquellen nicht, so dass man auf andere Tools (z. B. Xilinx ISE) ausweichen muss, um die Fehlerstellen zu finden. Das Problem wurde mit Version 0.26 beobachtet.

4.2.3 Vom Binary zum Block-RAM

Die meisten auf dem Markt verfügbaren Mikrocontroller integrieren bereits Speicher in Form von auf dem Chip verfügbaren RAM oder ROM. Bei der Nachbildung von automobilen eingebetteten Controllern im FPGA sind wir also darauf angewiesen, auch diese Komponenten nachzubilden. Da der Aufbau von großen Speichern aus den auf dem FPGA vorhandenen LUTs sehr teuer ist, sind in der Regel spezialisierte Block-RAM-Bausteine enthalten. Diese können sowohl zur Emulation von RAM als auch ROM genutzt werden. Auch auf dem von uns eingesetzten Xilinx Spartan 3E-500 befinden sich 20 Block-RAM-Primitive mit einer Kapazität von jeweils 18KBit. Bei der VHDL-Synthese erkennt das Tool Xilinx ISE automatisch RAM-ähnliche Strukturen und versucht, die aufwändige Implementierung in LUTs durch einen Block-RAM-Baustein zu substituieren. Die Beschreibung eines RAM-Bausteins in VHDL ist relativ einfach. Exemplarisch sei hier als Beispiel der im HC11-Core verwendete RAM-Baustein aufgeführt:

```
ENTITY int_ram IS
PORT
```

```
(
address : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
clock : IN STD_LOGIC ;
data : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
wren : IN STD_LOGIC;
q : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
);
END int_ram;
type typ_arr is array(0 to 1023) of std_logic_vector(7 downto 0);
signal wea : std_logic_vector(0 downto 0);
signal arr_dat:typ_arr;
begin
process(clock,address,data,wren) is
begin
   if clock'event and clock='1' then
       if wren='1' then
          q <= data;
       else
          q <= arr_dat(to_integer(unsigned(address)));
       end if;
   end if;
   if clock'event and clock='0' then
       if wren='1' then
              arr_dat(to_integer(unsigned(address))) <= data;
       end if;
   end if;
end process;
```

Dieses Konstrukt wird während der Synthese von der Software Xilinx ISE als Speicher erkannt und entsprechend substituiert. Alternativ kann man sich mit dem Werkzeug „Xilinx ISE Core Generator“ auch eine solche VHDL-Entität nach Angabe von Parametern wie Adress- und Datenbusbreite über eine graphische Oberfläche generieren lassen. Statt einer konkreten Implementierung des Speicherbausteins wird dann allerdings nur ein Verweis auf ein in der Xilinx-Core-Bibliothek enthaltenes Simulationsmodell generiert.

Auch ROM wird im FPGA auf einen oder mehrere Block-RAM-Bausteine abgebildet. Jedoch gibt es einen bedeutenden Unterschied im Vergleich zu RAM: Während bei letzteren der initiale Speicherinhalt keine Rolle spielt, da dieser durch den Prozessor beschrieben wird, so sieht dies bei der Konstruktion von ROM anders aus. Hier muss der Speicher zwingend mit dem Programm initialisiert werden, das der Controller ausführen soll. Um den initialen (und bei ROMs auch finalen) Speicherinhalt in das Block-RAM zu bekommen, gibt es nun wieder zwei Ansätze.

Als erste Möglichkeit kann analog zu dem RAM-Beispiel einfach eine konkrete Implementierung eines ROM-Bausteines angegeben werden. Beispiel:

```
ENTITY int_rom IS
PORT
(
address : IN STD_LOGIC_VECTOR (12 DOWNTO 0);
clock : IN STD_LOGIC ;
q : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
);
END int_ram;
begin
process(clock,address) is
begin
   if clock'event and clock='1' then
       with to_integer(unsigned(address)) select
 q <=
                 "10001110" when 0,
                 "00000011" when 1,
...
                 "10100010" when 8191;
   end if;
end process;
```

Auch ein solches Konstrukt erkennt das Synthesetool als speicherähnliches Konstrukt und ist bemüht, einen Block-RAM-Baustein zu benutzen, der mit den in der Implementierung enthaltenden Werten initialisiert wird. Allerdings ist es relativ mühsam ein solches VHDL-Modell eines ROMs manuell zu erstellen. Aus diesem Grund gibt es bestimmte Werkzeuge (z. B. „hex2rom“), die aus einer Binärdatei automatisch diesen VHDL-Quelltext generieren.

Die zweite Alternative besteht wiederum darin, den „Xilinx Core Generator“ zu benutzen. Neben den Angaben für Daten- und Adressbusbreiten sowie Timingverhalten kann man in der graphische Oberfläche eine Speicherinitialisierungsdatei angeben, die das Werkzeug im `.coe`-Format verlangt. Für die Konvertierung aus den bekannten herstellerspezifischen Formaten (z. B. Intel Hex oder Motorola S-Record) gibt es auch hier Werkzeuge (z. B. das in dieser Projektgruppe entwickelte Programm „`s19conv`“).

Ein sehr leistungsfähiges Werkzeug für die Konvertierung in VHDL-ROM-Modelle steht mit dem Open-Source-Programm „`hex2rom`“ zur Verfügung, das in der Projektgruppe ausgiebig verwendet wurde. Das Werkzeug bietet einstellbare Adress- und Datenbusbreiten und kann die Speicher entweder als Big-Endian oder als Little-Endian formatieren. Außerdem kann eine von mehreren Alternativimplementierungen ausgewählt werden. Auch der Name

der generierten VHDL-Entity muss per Kommandozeilenoption vorgegeben werden. Details der Benutzung werden bei Aufruf des Programms ohne Parameter wie folgt ausgegeben:

```
$ hex2rom
Usage: hex2rom [-b] <input file> <entity name> <format>
If the -b option is specified the file is read as a binary file
Hex input files must be intel hex or motorola s-record
The format string has the format AEDOS where:
  A = Address bits
  E = Endianness, l or b
  D = Data bits
  O = ROM type: (one optional character)
      z for tri-state output
      a for array ROM
      s for synchronous ROM
      u for XST ucf
      l for Leonardo ucf
  S = SelectRAM usage in 1/16 parts (only used when O = u)
Example:
  hex2rom test.hex Test_ROM 18b16z
```

Zur Konvertierung in die vom „Core Generator“ benötigten coe-Dateien kann das in der Projektgruppe entwickelte s19conv verwendet werden. Dieses kann allerdings nur das Motorola S-Record und nicht das Intel-Hex-Format verarbeiten. Der Aufruf ohne Parameter liefert auch hier genauere Angaben zur Benutzung:

```
$ s19conv
S19 file converter v0.9
=======================
PG 533 CoaCh, Technical University Dortmund, Germany
Usage: s19conv {coe|mif|vhdlrom} FILE [STARTADDRESS] [LENGTH]
       FILE          - S19 File to be converted
       coe           - convert to Xilinx COE-File for Memory
                       Core Generator
       mif           - convert to Xilinx mif-File for Behavioral
                       Simulation
       vhdlrom       - convert to a VHDL ROM model
       STARTADDRESS  - starting address of the generated memory file
       LENGTH        - length of the generated memory file
```

5 Entwicklung des SoCs

Ziel der Projektgruppe *CoaCh – Car on a Chip* für das erste Semester war es, basierend auf verschiedenen Mikroprozessoren SoCs zu entwickeln. Diese sollten dann einzelne Komponenten aus einem Auto bedienen, wie z. B. Scheinwerfer oder auch einen Gangwahlschalter.

In diesem Kapitel soll die Entwicklung eines solchen SoC beschrieben werden. Dazu werden in den einzelnen Abschnitten die verschiedenen Bestandteile eines Softcores und zusätzlich einige Peripheriekomponenten beschrieben und erläutert. Dabei besteht ein SoC in der Regel aus folgenden Bausteinen:

1. Prozessor
2. Speicher
3. Peripherie (Timer, UART etc.)
4. Daten- und Adressbusse

Diese Bausteine werden in normalen Systemen mit vielen ICs zusammen auf eine Platine angebracht. Beim Ansatz eines SoC werden alle Komponenten auf *einem* IC integriert. An dieser Stelle haben wir rekonfigurierbare Logik benutzt, so dass die Komponenten leicht verändert werden konnten. Die Bausteine wurden in einer Hardware-Beschreibungssprache implementiert und per Bussystem miteinander verbunden. Über einen Adressmultiplexer kann bestimmt werden, welche Komponente Zugriff auf den Bus haben darf.

Der Speicher wird ebenfalls mittels eines Busses verbunden. Da es sich um das Anwendungsgebiet der eingebetteten Systeme handelt, ist die Größe des Speichers begrenzt. Sollte dieser nicht ausreichen kann zusätzlicher Speicher extern angeschlossen werden.

Um die in der Hardware-Beschreibungssprache implementierten Komponenten mit der echten Hardware zu verbinden, können die verschiedenen Ausgänge der Bausteine über ein Constraint-File gemappt werden.

5.1 Taktteiler

Die Spartan 3 S500E FPGA-Boards von Xilinx sind von Haus aus mit einem 50MHz-Taktgenerator ausgestattet. Allerdings kann es in digitalen Schaltungen notwendig sein mit unterschiedlichen Taktfrequenzen zu arbeiten. Aufgrund von Verzögerungszeiten innerhalb digitaler Schaltungen muss der Takt gegebenenfalls geteilt werden, um eine korrekte Funktionsweise zu gewährleisten. So ist es oft erforderlich, dass ein kombinatorischer Schaltkreis [23], der

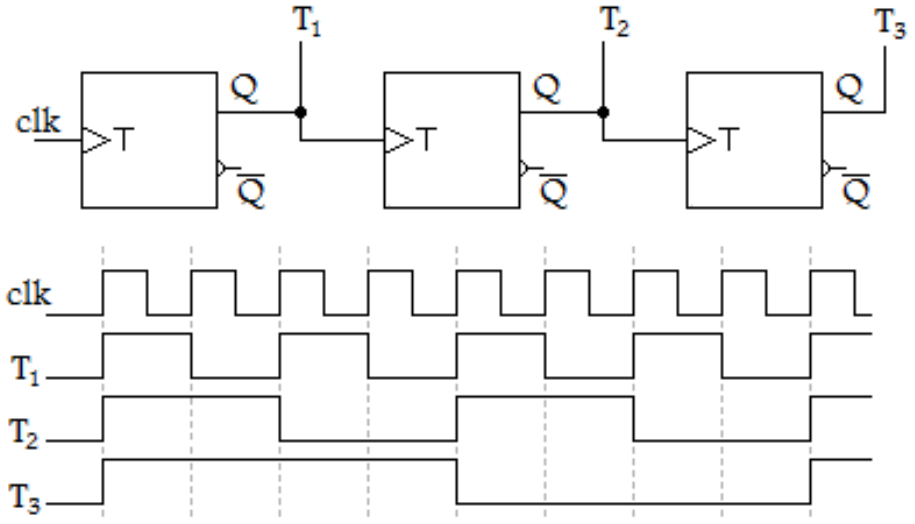

Abbildung 4: Taktteilung mit Hilfe von Toggle-Flipflops

eine beliebige boolesche Funktion realisiert, aus den Eingaben innerhalb eines Taktes entsprechende Ausgaben generiert. Die in der Schaltung verwendeten Gatter unterliegen beim Auswerten ihrer Eingänge geringen Verzögerungen. Für eine Taktfrequenz von 50MHz muss die Verzögerung der Logik also kleiner als 20 Nanosekunden sein, sonst würden beim nächsten Takt noch nicht korrekt berechnete Daten übernommen. Dabei ergibt sich die Gesamtverzögerung einer Schaltung aus dem Pfad durch die Schaltung mit der größten Summe der Verzögerungen der beteiligten Gatter. Synthese-Tools können in der Regel eine ungefähre Angabe zum Timingverhalten des Designs erstellen. Für eine genauere Timinganalyse sind allerdings Information aus der sogenannten *Place and Route*-Phase notwendig. Die Xilinx-Software stellt dafür einen entsprechenden Static-Timing-Report zur Verfügung.

Die wohl einfachste Art einen Taktteiler zu realisieren besteht im Hintereinanderschalten von einfachen Toggle-Flipflops. Das erste Flipflop halbiert bereits den Ausgangstakt, wie in Abbildung 4 zu sehen ist. Zusätzliche T-Flipflops ermöglichen also eine Teilung des Taktes um den Faktor $\frac{1}{2^n}$.

Digital Clock Manager Zum Generieren eines beliebigen Taktteilers ist es beim Umgang mit den Xilinx-FPGA Boards sinnvoll, die integrierten Digital Clock Manager [78] (DCM) zu benutzen. Zusätzliche Buffer ermöglichen es schließlich die auf den FPGAs vorhandenen Ressourcen für Taktsignale auszuschöpfen. So wird gewährleistet, dass ein Taktsignal gleichzeitig in den verschiedenen Regionen des FPGAs ankommt. Neben einfacher Taktteilung kann der Ursprungstakt auch multipliziert oder sogar phasenverschoben

werden. Zum Instanzieren eines DCMs auf Xilinx-FPGAs wird ein VHDL-Template verwendet. Anhand des nachfolgenden Beispiels wird gezeigt, wie ein DCM für die Spartan-3s500e-FPGAs zum Generieren eines durch zwei geteilten Taktes genutzt werden kann. Dazu abstrahieren wir ein wenig von den DCMs und erstellen uns eine eigene, sie umgebende Entity.

```
entity clkgen is
   port ( CLKIN_IN        : in    std_logic;
          RST_IN          : in    std_logic;
          CLKDV_OUT       : out   std_logic;
          CLK_IBUFG_OUT   : out   std_logic;
          CLK0_OUT        : out   std_logic;
          LOCKED_OUT      : out   std_logic);
end clkgen;
```

CLKDV_OUT ist das neue, geteilte Taktsignal, gepuffert durch einen Global Clock Buffer. CLK_IBUFG_OUT und CLK0_OUT entsprechen dem Ursprungstaktsignal, wobei CLK_IBUFG_OUT nur gepuffert und CLK0_OUT zusätzlich durch den DCM aufbereitet wurde. LOCKED_OUT signalisiert, ob die neuen Taktsignale gültig sind. Eine Komponentendeklaration für den DCM und die Clock Buffer ist hier nicht nötig.

```
architecture BEHAVIORAL of clkgen is
   signal CLKDV_BUF       : std_logic;
   signal CLKFB_IN        : std_logic;
   signal CLKIN_IBUFG     : std_logic;
   signal CLK0_BUF        : std_logic;
begin
   CLK_IBUFG_OUT <= CLKIN_IBUFG;
   CLK0_OUT <= CLKFB_IN;

   DCM_SP_INST : DCM_SP
   generic map(CLKDV_DIVIDE => 2.0,
            CLKIN_PERIOD => 20.000,
            STARTUP_WAIT => TRUE)
      port map (CLKFB=>CLKFB_IN,
                CLKIN=>CLKIN_IBUFG,
                DSSEN=>'0',
                PSCLK=>'0',
                PSEN=>'0',
                PSINCDEC=>'0',
                RST=>RST_IN,
                CLKDV=>CLKDV_BUF,
                CLKFX=>open,
```

```
                CLKFX180=>open,
                CLK0=>CLK0_BUF,
                CLK2X=>open,
                CLK2X180=>open,
                CLK90=>open,
                CLK180=>open,
                CLK270=>open,
                LOCKED=>LOCKED_OUT,
                PSDONE=>open,
                STATUS=>open);
```

Mit der generic map wird der DCM entsprechend konfiguriert. Für den gewünschten Teiler setzen wir CLKDV_DIVIDE auf 2.0. CLKIN_PERIOD ist ein optionaler Parameter, der zu besseren Ergebnissen führen kann. Er gibt die Zeit eines Taktzyklus in Nanosekunden an. Durch Setzen von STARTUP_WAIT auf TRUE erreichen wir, dass die Taktausgänge nach der Konfigurationsphase des FPGAs gültig sind. Alle anderen Parameter belassen wir auf ihren Standardwerten.

```
   CLKIN_IBUFG_INST : IBUFG
      port map (I=>CLKIN_IN,
                O=>CLKIN_IBUFG);

   CLKDV_BUFG_INST : BUFG
      port map (I=>CLKDV_BUF,
                O=>CLKDV_OUT);

   CLK0_BUFG_INST : BUFG
      port map (I=>CLK0_BUF,
                O=>CLKFB_IN);
end BEHAVIORAL;
```

Abschließend werden noch die Port Maps für die benötigten Clock-Buffer angegeben. Abbildung 5 zeigt eine schematische Darstellung des Entwurfs. Alle nicht benötigten Signale und Ports wurden weggelassen. Die Bezeichnungen außen beziehen sich auf die Ports der Top-Level Entity, die im Inneren stellen die wichtigen Ports des DCM dar.

Clock Enable Signal Da in der Regel für den Systemtakt und das entsprechende Verbindungsnetzwerk höhere Anforderungen gelten als für normale Signale, sollten diese nicht durch selbst generierte, kombinatorische Logik erzeugt werden. Allerdings können selbst entworfene Taktteiler, wenn auch in etwas abgewandelter Form, zum Steuern des Zeitverhaltens einer Systemkomponente eingesetzt werden, z. B. für eine UART-Komponente. Diese muss

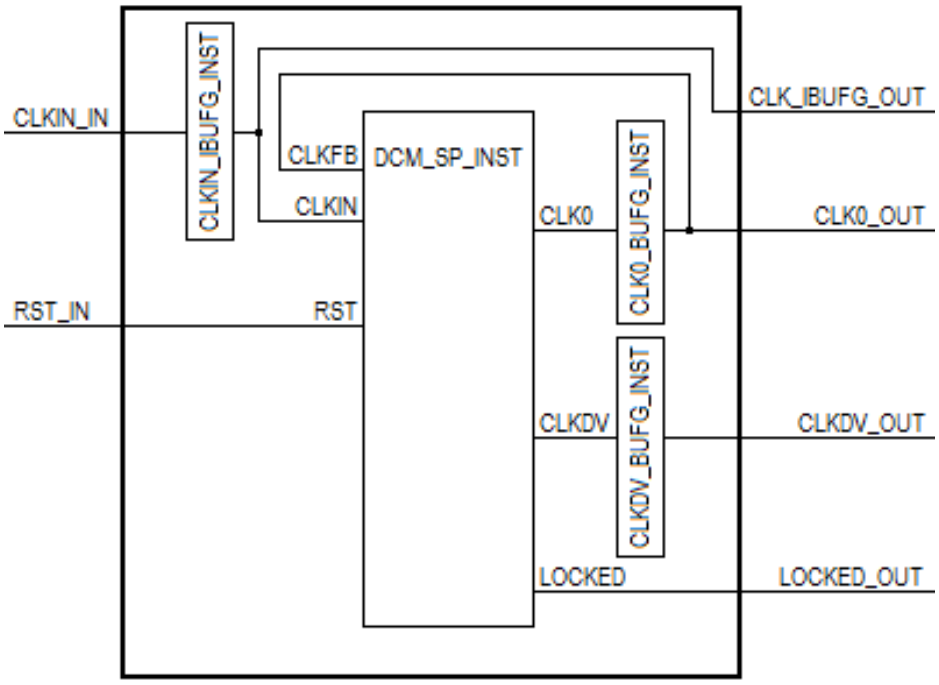

Abbildung 5: Architektur des Taktteilers

den Signalpegel auf dem Bus für eine Bitzeit konstant halten. Die Bitzeit ist in der Regel um ein Vielfaches länger als ein Taktzyklus. Es kann z. B. neben dem Systemtakt ein sogenanntes Clock Enable Signal generiert werden, das immer zu Beginn der spezifizierten Bitzeit für einen Taktzyklus den logischen Wert Eins annimmt und sonst auf Null bleibt. Änderungen auf dem Bus können dann anhand des Clock Enable Signals bewirkt werden.

Für einen 8-Bit-Mikrocontroller könnte dies wie folgt aussehen:

```
ENTITY clock_enable IS PORT (
   clock    : IN  STD_LOGIC; -- Systemtakt
   wr_en    : IN STD_LOGIC; -- Enable Signal für das Teiler-Register
   teiler   : IN STD_LOGIC_VECTOR(7 downto 0);
   clock_en : OUT STD_LOGIC; -- Taktgeteilter Takt (gerade)
);
END clock_enable;
```

Es werden zwei Register und ein Zähler benötigt. Für das Clock-Enable-Signal ist es sinnvoll, ein Register zu verwenden. Sonst müsste asynchrone Logik eingesetzt werden, und bei den Vergleichen zwischen Zähler und Teiler-Register könnte es zu einem sogenannten Glitch [15] kommen, bedingt durch die Änderung des Zählers, dessen Stellen nicht alle gleichzeitig den korrekten

neuen Wert annehmen.

```
ARCHITECTURE Behavioral of taktteiler is
signal clock_enable : std_logic;
signal reg_teiler   : std_logic_vector(7 downto 0) := "00000000";
signal counter      : std_logic_vector(7 downto 0) := "00000000";
begin
```

Für das Beschreiben des Teiler-Registers dient folgender Prozess. Bei jeder steigenden Taktflanke des Systemtaktes ist es damit möglich, den Teiler umzustellen.

```
process(clock,wr_en) begin
   if clock'event and clock = '1' and wr_en = '1'then
      reg_teiler <= teiler;
   end if;
end process;
```

Das Generieren des Clock-Enable-Signals übernimmt ein weiterer Prozess.

```
process(clock) begin
   if clock'event and clock = '1' then
      counter <= counter + 1;
      clock_enable <= '0';

      if counter = 0 then
         clock_enable <= '1';
      end if;

      if counter = reg_teiler then
         counter <= (others '0');
      end if;
   end if;
end process;
end Behavioral;
```

Bei dieser Implementierung sollte beachtet werden, dass der eingestellte Teiler um eins kleiner sein muss als der gewünschte Teiler. Durch weitere **if**-Abfragen könnten auch zusätzliche Enable-Signale erzeugt werden. So sieht beispielsweise die CAN-Bus-Spezifikation [41] ein Oversampling vor, bei dem während eines festgelegten Zeitpunktes Änderungen auf dem Bus bewirkt werden und nach einer festgelegten Zeit der Bus schließlich abgetastet wird.

5.2 Peripherie

5.2.1 Anbindung von Peripherie an die Softcores

Adressmultiplexing Ein Softcore besteht aus der CPU und meist aus Peripherie. Bei manchen Modellen muss die Peripherie noch ergänzt bzw. komplett hinzugefügt werden. Der Hauptunterschied zwischen den Softcores besteht in der Architektur und der Implementierung der jeweiligen CPU. Die Peripherie ist übergreifend gleich oder ähnlich und kann oft mit wenigen Änderungen kompatibel zu anderen Softcores gemacht werden. Die eigentliche Verbindung zwischen eben der CPU und ihrer Peripherie liegt aber im Bussystem. Die CPU berechnet Werte und gibt diese aus. Ob der Wert jetzt an einem Port angelegt oder doch in den Speicher geschrieben werden soll, ergibt sich meist durch die gerade anliegende Speicheradresse. Man legt also fest, bei welchen Adressen welcher Peripheriebaustein ausschließlich angesprochen werden soll. Dies nennt man dann *Memory Mapped I/O*. Wie man dieses Verfahren in VHDL für einen 8-Bit-Mikrocontroller implementiert, soll hier dargestellt werden, da es in den von uns genutzten Softcores auch angewendet wird.

Als Schnittstelle der CPU zum Bus setzen wir einige Signale voraus:

```
ENTITY cpu IS PORT (
cpu_datain : IN  STD_LOGIC_VECTOR (7 DOWNTO 0); -- Daten an CPU
cpu_clock  : OUT STD_LOGIC; -- Takt
cpu_adr    : OUT STD_LOGIC_VECTOR (15 DOWNTO 0); --Speicheradresse
cpu_we     : OUT STD_LOGIC; -- CPU will schreiben
cpu_re     : OUT STD_LOGIC; -- CPU will lesen
cpu_dataout: OUT STD_LOGIC_VECTOR(7 DOWNTO 0) -- Daten von CPU
);
END cpu;
```

Die Arbeit der CPU besteht also darin festzustellen, ob der aktuelle Befehl speichern bzw. laden möchte. Ein Schreibbefehl mit dem Ziel Speicher würde also eine gültige Adresse bei cpu_adr anlegen und cpu_we aktivieren. Weiter muss das Signal cpu_re deaktiviert sein! Die zu speichernden Daten liegen dann bei cpu_dataout an und werden beim nächsten Takt in den Speicher geschrieben. Wie man sieht, ist es sehr leicht, den Speicher an die CPU anzubinden. Der Speicher habe dabei folgende Schnittstelle (1kB):

```
ENTITY ram IS PORT (
ram_clock  : IN  STD_LOGIC; -- Von CPU
ram_adr    : IN  STD_LOGIC_VECTOR (9 DOWNTO 0); -- Von CPU
ram_we     : IN  STD_LOGIC; -- Von CPU, später Busmodul
ram_datain : IN  STD_LOGIC_VECTOR (7 DOWNTO 0); -- Von CPU
```

```
ram_dataout: OUT STD_LOGIC_VECTOR (7 DOWNTO 0) -- An Busmodul
);
END ram;
```

Wir können eine erste, einfache Version des Busmoduls angeben, welches nur dann Daten aus dem Speicher liefert, wenn der aktuelle Befehl diese wirklich lesen will:

```
ENTITY busmodule IS PORT (
bus_re        : IN  STD_LOGIC; -- Von CPU
bus_ram_datain: IN  STD_LOGIC_VECTOR (7 DOWNTO 0); -- Vom RAM
bus_dataout   : OUT STD_LOGIC_VECTOR (7 DOWNTO 0); -- An CPU
);
END busmodule;

ARCHITECTURE behaviour OF busmodule IS BEGIN
bus_dataout <= bus_ram_datain when (bus_re='1') else (others => '0');
END behaviour;
```

Geschrieben wird ohne Umwege direkt in den Speicher, falls dieser das Schreibrecht über das Signal cpu_we bekommt. Erweitert man das Szenario um einen weiteren Peripheriebaustein, wie z. B. einem Ein-/Ausgabeport, so muss auch das Busmodul erweitert werden! Der Port sei sehr einfach gehalten. Er ist im Prinzip ein 8-Bit-Register, welches eine einfache Ansteuerung für z. B. LEDs oder Taster darstellt. Zunächst die für uns relevante Schnittstelle des Ports:

```
ENTITY ioport IS PORT (
port_clock   : IN  STD_LOGIC_VECTOR (7 DOWNTO 0); -- Von CPU
port_we      : IN  STD_LOGIC; -- Vom Busmodul
port_datain  : IN  STD_LOGIC_VECTOR (7 DOWNTO 0); -- Von CPU
port_dataout : OUT STD_LOGIC_VECTOR (7 DOWNTO 0); -- An Busmodul
);
END ioport;
```

Wenn man nun diesen Port in den Speicher einbindet, d. h. bei einer bestimmten Speicheradresse anspricht, muss man diese noch explizit festlegen. Zusätzlich definieren wir eine Konstante, die angibt, ab welcher Adresse schließlich der Speicher beginnt.

```
constant PORT_ADDRESS = x"0000";
constant RAM_START_ADDRESS = x"0001";
```

Wir legen den Port also an die erste Speicherstelle. Danach soll wie gewohnt in den Speicher geschrieben werden. Dadurch ist das erste Byte des Speichers nicht mehr als Speicher nutzbar! Das Busmodul erhält neue Signale:

```
ENTITY busmodule IS PORT (
bus_adr        : IN  STD_LOGIC_VECTOR (15 DOWNTO 0); --Von CPU
bus_we         : IN  STD_LOGIC; -- Von CPU
bus_re         : IN  STD_LOGIC; -- Von CPU
bus_ram_datain : IN  STD_LOGIC_VECTOR (7 DOWNTO 0); --Vom RAM
bus_port_datain: IN  STD_LOGIC_VECTOR (7 DOWNTO 0); --Vom Port
bus_ram_we     : OUT STD_LOGIC; -- An RAM
bus_port_we    : OUT STD_LOGIC; -- An Port
bus_dataout    : OUT STD_LOGIC_VECTOR (7 DOWNTO 0); --An CPU
);
END busmodule;
```

Die vorherige Funktionalität wird mit Hilfe der Adresse, die die CPU liefert, erweitert:

```
ARCHITECTURE behaviour OF busmodule IS BEGIN
bus_dataout <= bus_port_datain when (bus_adr=PORT_ADDRESS and
               bus_re='1') else
               bus_ram_datain  when (bus_adr>=RAM_START_ADDRESS
               and bus_re='1') else (others => '0');
bus_port_we <= '1' when (bus_adr=PORT_ADDRESS and bus_we='1')
               else '0';
bus_ram_we  <= '1' when (bus_adr>=RAM_START_ADDRESS and bus_we='1')
               else '0';
END behaviour;
```

Der Datenausgang von der CPU wird mit allen Eingängen der Peripheriebausteine verbunden, die Schreibrechte erhalten diese jedoch vom Busmodul. Dabei bekommt jedoch nur ein Baustein gleichzeitig das Schreibrecht. Analog bekommt das Busmodul von jeder Peripherie einen Datenbus, leitet aber nur einen von diesen an die CPU weiter. Bindet man größere Bausteine, mit mehreren Kontrollregistern, an, so wird lediglich die Abfrage nach den passenden Adressen komplizierter. Die Schnittstelle des Busmoduls muss nur um einen Dateneingang und ein Schreibrechtausgang ergänzt werden.

Interrupts Ein Interrupt ist eine kurzfristige Unterbrechung eines Programms, um eine Verarbeitung eines anderen Programmstücks zu ermöglichen, welche in diesem Moment eine höhere Priorität hat. Dabei wird das Ereignis, das die Verarbeitung auslöst, Unterbrechungsanforderung (im Englischen Interrupt Request, kurz IRQ) genannt. Nach dem Auftreten eines Interrupts wird eine für diesen Interrupt spezifische Routine im Programm ausgeführt. Danach springt der Befehlszähler zurück an die Stelle im Hauptprogramm, an der es unterbrochen wurde.

Konkurrierend zu dem Interrupt-Mechanismus kann auch das *Polling-Verfahren* verwendet werden. Bei diesem Verfahren wird in regelmäßigen Abständen ein Wert abgefragt und verglichen. Allerdings hat dieses Verfahren zwei große Nachteile. Der erste ist, dass der Wert von dem Hauptprogramm aktiv abgefragt werden muss und daher wesentlich mehr Rechenleistung als beim Interrupt-Mechanismus verbraucht wird. Zum anderen muss ein gutes Intervall gewählt werden, in dem der Wert abgefragt wird, da ansonsten eine Veränderung des Werts übersehen werden könnte und es zu schwerwiegenden Folgen kommen kann (vor allem bei sicherheitskritischen Systemen). Wählt man das Intervall zu klein, wird zu oft abgefragt, und der erste Nachteil wird nur noch weiter verstärkt.

Dies soll ein kleines Beispiel verdeutlichen:

Sei *Ready* die abzufragende Variable, die den booleschen Wert enthält, ob eine Nachricht versandt wurde. Im Durchschnitt braucht die Versendung einer Nachricht 100ms, kann aber von diesem Wert, je nach Belastung, abweichen. Bei dem Polling-Verfahren würde nun z. B. alle 10ms der Wert der Variablen abgefragt werden, so dass eine schnelle Verarbeitung ermöglicht werden würde. Würde nun die Variable nach 100ms getriggert werden, hätte man in dieser Zeit schon 10 mal den Wert von *Ready* abgefragt. Würde man den Interrupt-Mechanismus benutzen bräuchte man keine einzige Abfrage(!). Nach erfolgreichem Versenden der Nachricht würde ein Interrupt aktiviert werden und die zugehörige Interrupt-Routine ausgelöst. Wie hieraus leicht ersichtlich ist: Der Interrupt-Mechanismus ist wesentlich effizienter im Bezug auf die verbrauchte Rechenleistung.

In den meisten Systemen gibt es verschiedene Arten von Interrupts, so dass viele verschiedene Interrupt-Routinen implementiert werden können und dem Programmierer viele Freiheiten lassen. Für das Beispiel des 8051-Mikrocontroller sollen nun einige typische Interrupt-Register und deren Funktion vorgestellt werden. Der 8051-Mikrocontroller hat die folgenden fünf Interrupt-Register:

1. Für den Timer 0
2. Für den Timer 1
3. Für den seriellen Anschluss
4. Für einen externen Interrupt
5. Für einen weiteren externen Interrupt

Hieran lassen sich drei typische Interrupt-Arten unterscheiden, und zwar der Timerüberlauf, das Senden/Empfangen eines Zeichens über die serielle

Schnittstelle und ein externes Event. Die beiden externen Interrupts lassen sich leicht für viele verschiedene Zwecke benutzen und können angepasst werden.

Allerdings müssen allgemein die Interruptfunktion bzw. die einzelnen Interrupts aktiviert werden, damit der jeweilige Interrupt in Funktion treten kann. Dies geschieht auf dem 8051 über das *IE-Register*. Dabei gibt jedes Bit des Registers an, ob der jeweilige Interrupt bzw. allgemein Interrupts aktiv sind. Diese Sicherheitsmaßnahme wird vor allem zum Schutz des Systems vor Missbrauch genutzt. Diese Funktion kann ebenfalls dazu genutzt werden die Interrupts aus einer bestimmten Quelle für kurze Zeit zu sperren. So könnte der Interrupt-Mechanismus für die serielle Übertragung nach dem Empfangen einer Instruktion so lange blockiert werden, bis das Programm die übertragene Instruktion abgearbeitet hat, damit das System nicht mit Instruktionen überflutet wird. Nach der Abarbeitung könnte die Routine das Bit für den seriellen Interrupt wieder setzen.

Was geschieht aber, wenn zwei verschiedene Interrupts gleichzeitig ausgelöst werden?

Um eine deterministische Ausführung zu gewährleisten werden die Interrupts nach einer bestimmten Reihenfolge abgefragt und dann der Reihe nach abgearbeitet. Bei dem 8051 werden die Interrupts in folgender Reihenfolge abgearbeitet:

1. externer Interrupt 0
2. Timer-Interrupt 0
3. externer Interrupt 1
4. Timer-Interupt 1
5. Serieller Interrupt

Zusätzlich kann man den verschiedenen Interrupts eine Priorität zuweisen. Je höher die Priorität, desto früher wird der Interrupt abgearbeitet. Liegt dieselbe Priorität bei zwei oder mehr Interrupts vor, werden die Interrupts mit der gleichen Priorität wieder in der normalen Reihenfolge abgearbeitet. Im Beispiel des 8051-Mikrocontroller gibt es zwei Prioritäten, die sich über das *IP-Register* für die verschiedenen Interrupts konfigurieren lassen. Da nur eine hohe und eine niedrige Priorität existiert, reicht ein Bit pro Interrupt aus. Ein Interrupt mit der hohen Priorität kann einen Interrupt mit der niedrigen Priorität unterbrechen. Falls eine Interrupt-Routine mit hohe Priorität ausgefürt wird, wird ein Interrupt mit niedriger Priorität zeitverzögert bearbeitet. Ähnlich wie oben beschrieben wird bei zwei oder mehr gleichzeitig

auftretenden Interrupts derjenige Interrupt zuerst abgearbeitet, der in der normalen Ausführungsreihenfolge zuerst kommen würde.

Was passiert wenn ein Interrupt ausgelöst wird?

Nachdem ein Interrupt ausgelöst wurde, werden bestimmte Schritte notwendig, bevor die Interrupt-Routine gestartet werden kann. Der wichtigste Schritt ist, die Register auf den Stack zu sichern, da diese in der Interrupt-Routine überschrieben werden könnten. Zum Beispiel könnte eine Berechnung durch einen Interrupt unterbrochen werden. Ein mögliches Zwischenergebnis würde nun im Akkumulator stehen. Wird nun in der Interrupt-Routine ebenfalls eine Berechnung ausgeführt, wird der Wert im Akkumulator überschrieben. Beim Rücksprung aus der Routine würde die unterbrochene Rechnung einen fehlerhaften Wert berechnen. Daher müssen bestimmte Register immer gesichert werden! Je nach Architektur können dies verschieden viele Register sein.

Im Beispiel des 8051-Mikrocontroller werden folgende vier Arbeitsschritte automatisch vor der Interrupt-Routine ausgeführt:

1. Programmzähler speichern
2. Interrupts der gleichen bzw. niedriger Priorität werden blockiert
3. Bei Timer und externen Interrupts werden die entsprechenden Flags gelöscht
4. Der Programmzähler wird auf den Anfang der Interrupt-Routine gesetzt

Zusätzlich sollten, wie oben beschrieben, die wichtigen Register gesichert werden.

Insgesamt lässt sich sagen, dass Interrupts ein mächtiges Werkzeug sind, die die Programmierarbeit deutlich vereinfachen können. Allerdings sollten bei Benutzung der Interrupts alle Arbeitsschritte gut durchdacht werden, um eine fehlerhafte Ausführung zu vermeiden.

5.2.2 Externes Mapping per User Constraints File

Der Benutzer kann für das gewünschte Design Bedingungen für das Timing und für das Layout festlegen. Diese können entweder per Hand in das sog. „User Constraints File“ (UCF) eingetragen oder mit speziellen Editoren wie PACE und Xilinx Constraints Editor erzeugt werden.

Die Constraints finden auf der logischen Ebene statt und beeinflussen die Art, wie das logische Design auf der Zielhardware implementiert wird.

Constraints, die während der Design-Entry-Phase automatisch spezifiziert wurden, werden dadurch überschrieben.

Die Syntax für die Constraints-Spezifikation für eine Instanz ist:

```
INST instanceName constraintName =
          constraintValue | constraintName = constraintValue;
```

Die Constraints, die wir für unsere SoCs benutzt haben, haben wir aus dem „Spartan-3E Starter Kit Board User Guide“ entnommen. Hier werden für die verschiedenen logischen Komponenten

- die richtige Platzierung auf dem FPGA festgelegt (*Location Constraints*), das heißt, es werden die I/O-Pin-Zuweisung und die I/O-Standards festgelegt:

  ```
  NET "can_tx_pin" LOC="B4" | IOSTANDARD=LVTTL | SLEW=SLOW |
                            DRIVE=6 ;
  NET "can_rx_pin" LOC="A4" | IOSTANDARD=LVTTL | SLEW=SLOW |
                            DRIVE=6 ;
  ```

- *Time Constraints* z. B. es werden die Clock-Periode, Clock-Frequenz und der duty cycle des eingehenden Clock-Signals festgelegt:

  ```
  # Define clock period for 50 MHz oscillator
  NET "CLK_50MHZ" PERIOD = 20.0ns HIGH 40%;
  ```

Die von uns zum Konfigurieren der SoCs benutzten Optionen sind:

- LOC
- IOSTANDARD
- SLEW
- DRIVE
- PULLUP; PULLDOWN
- FAST

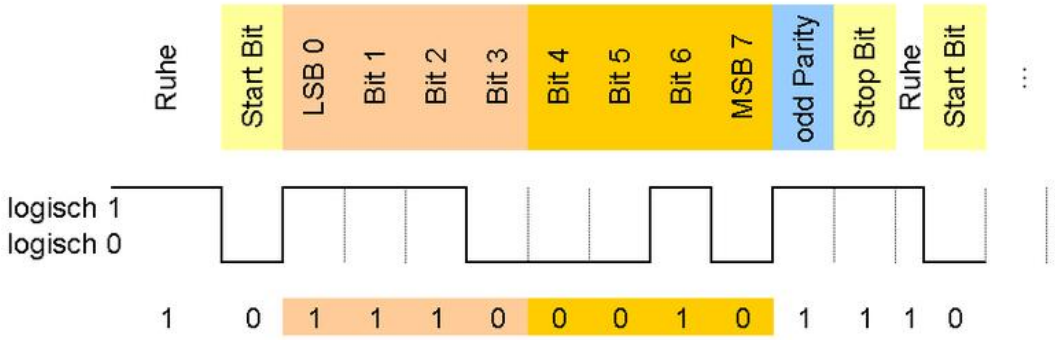

Abbildung 6: Aufbau einer seriellen Nachricht [25]

5.2.3 UART

Das Senden und Empfangen von Daten über eine serielle Schnittstelle gehört zu den wichtigen Fähigkeiten eines Mikrocontrollers. Bei diesen Daten ist zu beachten, dass sie einer bestimmten Norm entsprechen müssen, um korrekt von der Gegenstelle interpretiert zu werden. Zum Beispiel beinhalten die Nachrichten jeweils ein Start- und ein Stopbit, die eine asynchrone Datenübertragung ermöglichen. Sender und Empfänger müssen dadurch über kein gemeinsames Taktsignal verfügen, da das Senden des Startbits beim Empfänger das Lesen der Nachricht einleitet. Wichtig ist lediglich eine einheitliche Baud-Rate bei beiden Teilnehmern, wobei jedoch zu beachten ist, dass zwei kommunizierende Rechner niemals wirklich gleich schnell sind. Bei einem Unterschied von nur einem Promille wären zwei Rechner nach dem Senden von 1000 Zeichen um ein Zeichen „auseinandergelaufen“. Die Konsequenz aus dieser Tatsache ist, dass die Nachrichten möglichst kurz sein müssen, sprich: nur ein Zeichen lang. Bei jedem weiteren Startbit vor jedem gesendeten Zeichen synchronisieren sich beide Teilnehmer neu und die Übertragung bleibt korrekt. Ist ein Startbit übertragen, wird daraufhin das aus 8 Bit bestehende Zeichen, was den Inhalt der Nachricht darstellt, mit dem niederwertigsten Bit voran gesendet. Optional kann ein Paritätsbit mitübertragen werden, bevor danach die Nachricht vom Stopbit beendet wird.

Eine Erzeugung eines korrekten Signals in Software wäre relativ aufwendig. Man müsste durch geeignete Timingroutinen umständlich sicherstellen, dass die strikte Norm der Nachrichten erhalten und das Signal für die Gegenstelle lesbar bleibt. Ein so gestalteter Code wäre schwer zu verstehen und zu warten. Aus diesem Grund verfügen Mikrocontroller über den sogenannten UART, abgekürzt für „Universal Asynchronous Receiver Transmitter“. Dieser Baustein bildet die Schnittstelle zwischen dem Rechner und dem seriellen

Port (meistens RS-232) und löst diese Aufgabe in Hardware. Er versetzt die zu sendende Nachricht in die nötige Form, führt den Sendevorgang durch und meldet der CPU, dass Nachrichten eingegangen sind. Gute UARTs arbeiten voll-duplex, d. h., sie sind in der Lage, gleichzeitig eine Nachricht über den Tx-Pin der seriellen Schnittstelle zu senden und über den Rx-Pin zu empfangen. Die Handhabung der UARTs variiert von Mikrocontroller zu Mikrocontroller. Es existieren sowohl Geräte, die über einen Puffer verfügen, in dem man eine mehrere Zeichen lange Nachricht speichern kann, als auch Modelle ohne Puffer, denen man die Zeichen einzeln zufügen muss. Üblich ist eine Konfiguration über Special-Function-Register (SFR), bei der man zwischen verschiedenen Betriebsmodi wählen kann. Einstellbar sind meistens die Baudrate, ob ein Paritätsbit gewünscht ist, oder ob man ausschließlich senden und nicht empfangen möchte.

Hier folgt ein Beispiel für eine in C verfasste Sende- und Empfangsroutine bei einem Chip der 8051-Bauart. Diese Chips verfügen über keinen UART-Puffer. Sie melden über das Register `TI`, dass ein Zeichen erfolgreich gesendet wurde und über das Register `RI`, dass ein Zeichen eingegangen ist.

```
void serial_send(unsigned char dat){
        while(!TI);
        TI = 0;
        SBUF = dat;
}

unsigned char serial_read(){
        while(!RI);
        RI = 0;
        return SBUF;
}
```

Die while-Schleife zu Beginn der Senderoutine lässt den Rechner so lange warten, bis das vorher gesendete Zeichen erfolgreich gesendet und der UART wieder frei ist. `TI` muss anschließend manuell auf 0 gesetzt werden, bevor mit der Zuweisung an `SBUF` das Senden des nächsten Zeichens (`dat`) eingeleitet wird. In der Senderoutine wartet der Rechner solange, bis der UART mittels `RI` ein eingegangenes Zeichen meldet, bevor dieses dann per `return`-Anweisung zurückgegeben wird. An diesem Beispiel kann man deutlich erkennen, wie komfortabel das Senden von seriellen Signal mit Hilfe eine UARTs ist.

5.2.4 CAN-Controller

Damit die entwickelten SoCs mit anderen CAN-Bus-Teilnehmern kommunizieren konnten, war es nötig jeweils CAN-Controller in die Designs zu integrieren. Die Evaluation verfügbarer Open Source CAN-Controller hat jedoch ergeben, dass diese nur eingeschränkt für unsere Zwecke zu verwerten waren. Wir haben uns sowohl den auf opencores.org verfügbaren CAN-Controller angeschaut, der dem weit verbreiteten Philips SJA 1000 nachempfunden ist, als auch den DMA-fähigen CAN-Controller von Gaisler Research. Beide haben uns aus folgenden Gründen nicht zugesagt:

- Der opencores.org-Core hatte eine sehr schlechte Codequalität, da dieser nur in einer automatisiert aus Verilog konvertierten VHDL-Version verfügbar ist. Da wir uns auf VHDL als Implementierungssprache der PG geeinigt hatten, wollten wir die Verilog-Version nicht verwenden.
- Der Code des Gaisler Research DMA-CAN-Controllers war zwar recht übersichtlich, allerdings hat dieser relativ umfangreiche Bibliotheksabhängigkeiten zur GRLIB. Dies hätte bei Verwendung unsere Code-Basis unnötig vergrößert.

Deshalb haben wir, basierend auf der BOSCH CAN-Bus-2.0-Spezifikation [41], einen eigenen CAN-2.0-Controller implementiert. In diesem Kapitel werden die interne Architektur sowie die Schnittstellen zu Hard- und Software, in Form des Bus Interfaces bzw. der Treibersoftware, des entstandenen Cores beschrieben.

Architektur Der Controller besteht aus folgenden Modulen:

- can_controller.vhd – Die Top-Level-Entity mit folgenden Aufgaben:
  Sie beherbergt alle folgend aufgeführten Sub-Entities und generiert deren Timing, indem sie aus dem anliegenden Systemtaktsignal mit Hilfe eines einstellbaren Vorteilers ein Clock-Enable-Signal erzeugt, dessen Frequenz der 10fachen Busdatenrate entspricht. Dies ist notwendig, da laut der CAN-Spezifikation eine Bitzeit aus 10 Zeitquanten besteht. Des weiteren wird hier der am Bus aktuell anliegende Pegel in einem Flip-Flop zwischengepuffert, um dieses Signal zeitlich kontrollierbar zu machen und Hazards in weiter innenliegenden Teilen des Designs zu verhindern.
- can_transmitter.vhd – Das Sende-Modul mit folgenden Aufgaben:
  Der can_transmitter ist für das Senden von Nachrichten zuständig und die Autorität in Bezug auf das Setzen eines dominanten Buspegels.

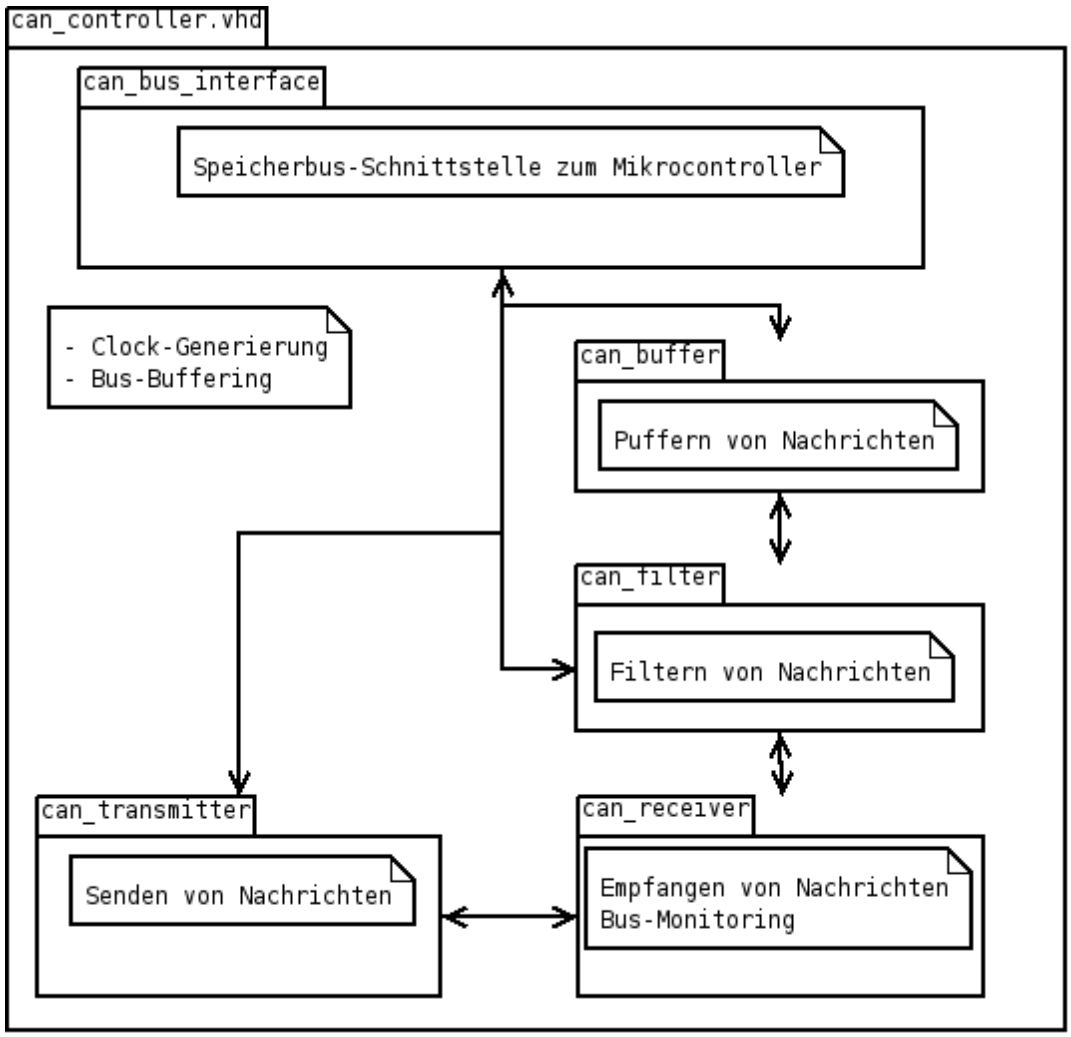

Abbildung 7: Architektur des CAN-Controllers

Er übernimmt an seiner Außenschnittstelle die zu sendende Nachricht nebst msg_ready-Flag, sowie ein vom can_receiver generiertes Startsignal (bus_idle_you_may_send) und sendet die Nachricht auf den Bus. Im Falle einer Buskollision bricht er das Senden direkt ab und wartet auf erneutes Setzen des bus_idle-Signals. Er ist im Wesentlichen als Zustandsautomat realisiert und besteht aus zwei Prozessen. Der erste Prozess sorgt für alle zustandsquerschneidenden Aktivitäten sowie für das Weiterschalten des States. Außerdem hält dieser den sogenannten bit_counter vor, einen Zähler, der angibt wie viele Bitzeiten sich der Automat schon im aktuellem Zustand befindet. Der zweite Prozess realisiert den eigentlichen Automaten, dessen Zustände im Wesentli-

chen den Nachrichtensegmenten einer CAN-Bus-Nachricht entsprechen. Hier gibt es eine große Fallunterscheidung, die je nach Zustand und erreichter Verweildauer in diesem Zustand (bit_counter) vorgibt, welcher Pegel auf den Bus zu legen ist und in welchen Folgezustand gesprungen werden soll. Diese Anweisungen werden vom ersten Prozess ausgeführt, d. h. der gewünschte Pegel wird unter Beachtung der Bit-Stuffing-Regeln auf den Bus gelegt und der Zustand zugunsten des Wunsches des zweiten Prozesses weitergeschaltet. Sollte laut Bit-Stuffing-Regel ein Stopfbit gesendet werden müssen, so wird zunächst dieses für eine Bitzeit auf den Bus gelegt und der Zustandswechsel erst eine Bitzeit später ausgeführt. So durchläuft der Automat alle Nachrichtensegmente der CAN-Nachricht und meldet bei Erreichen des letzten Segments (end_of_frame) den Vollzug des Sendevorgangs durch Setzen des msg_sent-Signals. Dadurch signalisiert er seinem Auftraggeber (dem can_bus_interface bzw. der dieses ansprechenden Software), dass er die an seiner Außenschnittstelle anliegende Nachricht nun nicht mehr verwenden muss. Bevor es mit der nächsten Nachricht weitergehen kann, wartet der Automat darauf, dass die Software das msg_ready-Signal zurücksetzt, was er seinerseits mit Zurücksetzen des msg_sent-Flags quittiert.

- can_receiver.vhd – Das Empfangs-Modul mit folgenden Aufgaben:
  Der can_receiver sorgt nicht nur für das Senden von Nachrichten, sondern ist auch die Autorität, was die Überwachung des Busses und das Zählen eventueller Protokollfehler angeht. Er „liest“ ständig auf dem Bus mit und versucht seine Zustandsmaschine immer in dem Zustand zu halten, der dem auf dem Bus gerade gesendeten Nachrichtensegment entspricht. So ist er auch für das Senden von ACK-Signalen und Error-Frames zuständig, indem er beim can_transmitter zum richtigen Zeitpunkt das Setzen eines dominanten Buspegels über ein Signal anfordert. Er stellt an seiner Außenschnittstelle u. a. die empfangenen Nachrichten zur Verfügung. Sein Aufbau ist ähnlich zum can_transmitter, das heißt, es gibt auch hier im Wesentlichen zwei Prozesse. Der Erste sorgt sich um die Weiterschaltung des Zustandes, das Sampling des Buspegels, die Implementierung des Bitstuffing, sowie die Fehlerdetektion. Der zweite Prozess entscheidet abhängig vom aktuellen Zustand, der Verweildauer in diesem Zustand sowie des gesampleten Buspegels, welche Aktion unternommen werden muss. Diese Entscheidung beinhaltet u. a. die Frage, ob und in welchen Folgezustand gewechselt werden soll, welche Teile des Nachrichtenpuffers beschrieben werden sollen, ob in der nächsten Bitzeit ein ACK-Segment gesendet werden muss, sowie

ob der can_transmitter im aktuellen Buszustand anfangen darf zu senden. Den Empfang einer vollständigen Nachricht signalisiert er durch Setzen des msg_read-Flags, das vom can_bus_interface bzw. der Software durch Setzen eines msg_ack-Flags quittiert werden muss.

- can_filter.vhd – Das Acceptance-Filtering-Modul:
  Der can_filter ist dafür zuständig, die vom can_receiver empfangenen Nachrichten zu filtern und sie nur im Falle eines positiv verlaufenden Vergleichs zum can_buffer durchzureichen. Hierfür verwaltet er in einem internen Block-RAM ein Feld von bis zu 256 zugelassenen 28-Bit-Identifiern nebst den zugehörigen 28 Bit breiten Masken. In dem Moment, in dem der can_transmitter den Empfang eines Identifiers durch Setzen des can_filter_en-Flags signalisiert, fängt der can_filter an, diesen Identifier nacheinander mit allen im Block-RAM vorgehaltenen Identifier-Masken-Kombinationen zu vergleichen. Hierzu wendet eine logisches UND sowohl auf den empfangenen Identifier und die Filter-Maske, als auch den Filter-Identifier und die Filter-Maske an. Stimmen die Ergebnisse beider Operationen überein, signalisiert der can_filter dem can_receiver über das filter_match_any-Signal, dass die Software an dieser Nachricht interessiert ist. Durch den begrenzten Zeitraum zwischen dem Empfangen eines Identifiers und dem Ende der kürzestmöglichen CAN-Nachricht, ist es je nach Busgeschwindigkeit nur möglich eine bestimmte Menge von Vergleichen durchzuführen. Diese Anzahl hängt vom eingestellten Prescaler ab. Es gilt die Faustregel: Es sind PRESCALER+1 Filtervergleiche möglich. Bei einem Systemtakt von 50Mhz und einer Busgeschwindigkeit von 500000 Baud können also 10 Filtervergleiche durchgeführt werden. Die Überdimensionierung der Anzahl theoretisch möglicher Filter erklärt sich durch die Nutzung eines ganzen Block-RAM-Bausteines. Theoretisch wäre es möglich alle Filter parallel zu testen, dann könnte jedoch zur Implementierung kein Block-RAM-Primitiv verwendet werden, und es müssten die Filter in einer sehr grossen Menge LUTs gespeichert werden, was vermieden werden sollte.

- can_buffer.vhd – Das Nachrichtenpuffer-Modul:
  Der can_buffer ist dafür zuständig, die vom can_receiver empfangenen und vom can_filter vorgefilterten Nachrichten in einem FIFO-Puffer zwischenzuspeichern. Auch hier wird ein Block-RAM verwendet, was Platz für bis zu 128 CAN-Nachrichten bietet.

- can_bus_interface.vhd – Das Speicherbus-Schnittstellen-Modul:
  Die Ansteuerung des CAN-Controllers erfolgt in der Regel über

Memory-Mapped I/O. Das can_bus_interface stellt die Schnittstelle zum Speicherbus des Mikrocontrollers dar und besteht aus Adresslatches für die Schnittstellen-Register der weiteren Module. Auf den Registersatz soll im nächsten Abschnitt eingegangen werden.

Bus-Interface Der CAN-Controller verfügt über folgende Register, welche in den Adressraum des Mikrocontrollers eingeblendet werden können:

Offset	Beschreibung
	Status- und Kontrollregister
0x00	Control Register
0x01	Prescaler Register
0x02	RX-Error-Counter
0x03	TX-Error-Counter
	Acceptance Filtering
0x06	Filter-Index*
0x07	Filter-Identifier Bit 7-0 / x
0x08	Filter-Identifier Bit 15-8 / x
0x09	Filter-Identifier Bit 23-16 / 5-0
0x0a	Filter-Identifier Bit 28-24 / 10-6
0x0b	Filter-Mask Bit 7-0 / x
0x0c	Filter-Mask Bit 15-8 / x
0x0d	Filter-Mask Bit 23-16 / 5-0
0x0e	Filter-Mask Bit 28-24 / 10-6
	Sendepuffer
0x10	Extended-Identifier Bit 7-0
0x11	Extended-Identifier Bit 15-8
0x12	Extended-Identifier Bit 23-16
0x13	Extended-Identifier Bit 28-24
0x14	Standard-Identifier Bit 7-0
0x15	Standard-Identifier Bit 10-8
0x16	Data Length Code / Message-Format-Type
0x20	Databyte 0
0x21	Databyte 1
0x22	Databyte 2
0x23	Databyte 3
0x24	Databyte 4
0x25	Databyte 5
0x26	Databyte 6
0x27	Databyte 7
	Empfangspuffer

Offset	Beschreibung
0x18	Extended-Identifier Bit 7-0
0x19	Extended-Identifier Bit 15-8
0x1a	Extended-Identifier Bit 23-16
0x1b	Extended-Identifier Bit 28-24
0x1c	Standard-Identifier Bit 7-0
0x1d	Standard-Identifier Bit 10-8
0x1e	Data Length Code / Message-Format-Type
0x28	Databyte 0
0x29	Databyte 1
0x2a	Databyte 2
0x2b	Databyte 3
0x2c	Databyte 4
0x2d	Databyte 5
0x2e	Databyte 6
0x2f	Databyte 7

Tabelle 1: Registerbelegung des CAN-Controllers

Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Beschreibung	OMRD	OMST	TXIRQ_EN	0	IMRCV	IMACK	RXIRQ_EN	0

Tabelle 2: CAN-Controller: Das Control Register

Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Beschreibung	Extended?	Remote?	0	0	DLC3	DLC2	DLC1	DLC0

Tabelle 3: CAN-Controller: Length/Format/Type Register

Auf die genaue Semantik der Register soll an dieser Stelle nicht eingegangen werden (siehe Quelltext von `can_driver.c`).

Treibersoftware Es wurde ein CAN-Treiber in der Programmiersprache C entwickelt, welcher von den Besonderheiten des Bus-Interfaces abstrahiert. Dieser verfügt über folgende Schnittstellen:

```
//Basisfunktionalität
struct can_message {
uint8 async;
        uint8 is_extended;
        uint8 is_remote;
        uint8 identifier[4];
        uint8 length;
        uint8 data[8];
};

void can_init(uint8 prescaler);
void can_send(struct can_message * msg);
int can_recv(struct can_message * msg);


//Interrupts
void can_register_recv_callback( void(*proc)(void)  );
void can_register_sent_callback( void(*proc)(void)  );
void can_handle_irq();


//Hardware-Acceptance-Filter
struct can_filter {
uint8 identifier[4];
uint8 mask[4];
uint8 is_extended;
};
void can_set_filter(uint8 idx, struct can_filter *filter);
void can_get_filter(uint8 idx, struct can_filter *filter);
```

Initialisierung Zunächst kann der CAN-Controller initialisiert werden. Hierzu genügt der Aufruf von `can_init()` mit einem Wert für den Prescaler als Parameter. Dieser Wert kann mit Hilfe des Präprozessor-Makros `PRESCALER(clk,baud)` errechnet werden. Um den CAN-Controller bei einem Systemtakt von 50Mhz auf eine Busgeschwindigkeit von 500 kBaud zu konfigurieren lautet der Aufruf also: `can_init(PRESCALER(50000000,500000));`. Die explizite Initalisierung des CAN-Controllers ist optional. Sollte der Programmierer ihn nicht initialisieren, so wird er bei Bedarf automatisch mit den Standardwerten für Systemtakt und Busgeschwindigkeit aus der Datei

`can_driver.h` initalisiert.

Senden und Empfangen Zum Versenden einer Nachricht genügt es, einen `can_message`-struct zu füllen und im Anschluss die Prozedur `can_send()` mit einem Zeiger auf ebendiesen struct aufzurufen. Beispiel:

```
void main() {
  struct can_message msg;

  msg.is_extended=0;        // keine extended-Nachricht senden
  msg.is_remote=0;          // es soll ein Data-Frame gesendet
                            // werden (kein Remote-Frame)
  msg.identifier[0]=0x10;
  msg.identifier[1]=0x2;    // msg_identifier 0x210
  msg.length=4;             // Länge der Nachricht: 4 Bytes
  msg.data[0]=0xde;
  msg.data[1]=0xad;
  msg.data[2]=0xbe;
  msg.data[3]=0xef;         // Nachrichteninhalt: "de ad be ef"

  can_send(&msg);           // versendet die Nachricht

}
```

Zum Empfangen einer Nachricht stehen drei Möglichkeiten zur Verfügung. Die erste Möglichkeit ist blockierendes Empfangen mit aktivem Warten. Es genügt die Prozedur `can_recv()` mit einem Zeiger auf einen `can_message`-struct aufzurufen, bei dem die Eigenschaft `async` mit `0` initialisiert wurde. Die Prozedur kehrt erst dann zurück, wenn eine neue Nachricht vorliegt. Beispiel:

```
//blockierend empfangen
void main() {
  struct can_message msg;
  msg.async=0;
  can_recv(&msg);
  printf("message received. length=%i\n",msg.length);
}
```

Hier findet aktives Warten in der Prozedur `can_recv` statt. Optional gibt es die Möglichkeit, nicht-blockierend zu empfangen. Dies wird erreicht, indem vor dem Aufruf der Prozedur `can_recv()` die `async`-Eigenschaft mit einem von `0` unterschiedlichen Wert initialisiert wird. Beispiel

```
//nicht-blockierend empfangen
void main() {
  struct can_message msg;
  msg.async=1;
  while(!can_recv(&msg)) {
     printf("nothing received.\n");
     sleep(1);
  }
  printf("message received. length=%i\n",msg.length);
}
```

Interrupts Die dritte Möglichkeit des Nachrichtenempfangs nutzt die vom CAN-Controller erzeugten Interruptanforderungen. Hier muss beim CAN-Treiber ein Funktionspointer registriert werden, der auf eine Behandlungsfunktion für empfangene Nachrichten zeigt. Jedes Mal, wenn eine neue Nachricht empfangen wurde, wird diese Funktion im Interruptkontext automatisch aufgerufen. Beispiel:

```
//Empfangen mit Interrupts

void on_recv() {
  struct can_message msg;
  can_recv(&msg);
  printf("message received. length=%i\n",msg.length);
}

void main() {

  can_register_recv_callback( &on_recv );
  while(1) {
     printf("do something useful.\n");
  }

}
```

Analog zu `can_register_recv_callback()` gibt es ebenfalls die Möglichkeit, mit `can_register_sent_callback` eine Funktion zu registrieren, die jedesmal dann aufgerufen wird, wenn eine CAN-Nachricht erfolgreich versendet wurde. Beide Funktionen kümmern sich eigenständig darum, die Interrupts beim CAN-Controller zu aktivieren. Bei Übergabe eines NULL-Pointers werden die Interrupts wieder ausgeschaltet. Die Prozedur `can_handle_irq()` wird als eigentlicher Interrupthandler in der Vektortabelle des Mikrocontrollers eingetragen und ruft dann die jeweils richtige Prozedur auf.

Acceptance Filter Der CAN-Controller bietet die Möglichkeit, Nachrichten bereits während der Dekodierung nach vom Benutzer vorgegebenen Identifiern und Masken zu filtern. Hierzu stellt der CAN-Treiber die Funktionen `can_set_filter()` sowie `can_get_filter()` bereit. Die Filtermasken funktionieren analog zu z. B. TCP/IP-Netzmasken. Sowohl die Identifier der empfangenen Nachrichten als auch der Filter-Identifier werden über ein logisches UND mit der Maske verknüpft. Stimmen die Ergebnisse beider Operationen überein, so handelt es sich um einen Treffer, und die Nachricht wird akzeptiert. Im CAN-Controller können mehrere Filter konfiguriert werden. In folgendem Beispiel werden zwei Filter konfiguriert. Der Filter mit dem Index 0 wird so konfiguriert, dass er Nachrichten mit Standard-Identifiern zwischen `0x200` und `0x2ff` akzeptiert (die wahlfreien Bits sind hier in der Maske ausgenullt). Der Filter mit dem Index 1 wird für Identifier zwischen `0x100` und `0x1ff` freigeschaltet:

```
void main() {
  struct can_filter fil;
  struct can_message msg;

  fil.identifier[0]=0x00
  fil.identifier[1]=0x2;
  fil.mask[0]=0x00;
  fil.mask[1]=0xff;
  fil.is_extended=1;
  can_set_filter(0,fil);    //set filter with idx 0

  fil.identifier[1]=0x1;
  can_set_filter(1,fil);    //set additional filter with idx 1


  can_recv(&msg);
  printf("msg recv with identifier between 0x200 and
```

```
                0x2ff or between 0x100 and 0x1ff \n");

}
```

Die eingestellten Filter lassen sich auch wieder auslesen. Hierfür wird die Funktion `can_get_filter()` bereitgestellt.

5.2.5 LCD-Controller

LCD-Modul Um Zeichen auf dem LCD-Display des Spartan-FPGAs ausgeben zu können, benötigt das SoC ein Modul für die Ansteuerung des Displays. Basierend auf einem Test-Modul, das die Initialisierung des LCDs mit entsprechenden Signalen durchführt, entwickelten wir eine Erweiterung, die das Einlesen von Zeichen und Konfigurationsbefehlen erlaubt.

Wenn das Modul in das System eingebunden wurde, kann der Benutzer über zwei spezielle Register auf nahezu alle Funktionen des Displays zugreifen. Nicht implementiert sind Funktionen zum Auslesen der Zeichen. Auf die Einzelheiten des Moduls soll hier nicht weiter eingegangen werden. Es erfüllt die Anforderungen, die im Spartan3-User-Guide detailliert angegeben sind. Zu diesen Anforderungen gehört auch das Timing beim Senden der Signale an das Display. Hierfür besitzt das Modul zwei Counter, deren Grenzwerte an den Systemtakt angepasst werden müssen. Da das VHDL-Modul keine Puffer bereitstellt, ist für die korrekte Darstellung von Text auf dem LCD auch das richtige Timing beim Senden der einzelnen Zeichen wichtig. Die Handhabung des Timings ist dabei dem Benutzer überlassen. In unserem Fall geschieht dies in der C-Bibliothek, die weiter unten beschrieben wird.

Das VHDL-Modul besitzt die Ausgangssignale, die im UCF an die richtigen PINs gemapped werden müssen. Für das Spartan-3E sind folgende Zeilen im User Constraints File nötig:

```
NET "LCD_E"   LOC = "M18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
              SLEW = SLOW ;
NET "LCD_RS"  LOC = "L18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
              SLEW = SLOW ;
NET "LCD_RW"  LOC = "L17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
              SLEW = SLOW ;
NET "SF_D<8>" LOC  = "R15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
              SLEW = SLOW ;
NET "SF_D<9>" LOC  = "R16" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
              SLEW = SLOW ;
NET "SF_D<10>" LOC = "P17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
```

```
                SLEW = SLOW ;
NET "SF_D<11>" LOC = "M15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
                SLEW = SLOW ;
```

Das LCD wird über zwei 8-Bit-Register „lcd_char_addr“ und „lcd_set_addr“ gesteuert. Daten, die in das Register „lcd_set_addr“ geschrieben werden, werden vom Modul als Konfigurationsbefehle interpretiert. So lässt sich z. B. die Position des darzustellenden Zeichens festlegen. Die Inhalte des Registers „lcd_char_addr“ wiederum werden direkt auf dem Display dargestellt. Beim Starten des FPGAs wird ein Begrüßungstext auf dem LCD angezeigt, der fest im VHDL-Modul integriert ist und als Abschluss der Initialisierungsphase angezeigt wird. Anschließend steht das LCD dem Benutzer zur Verfügung.

LCD-Bibliothek In der Datei „lcd.c“ ist eine kleine C-Bibliothek, die den Umgang mit dem LCD-Controller erleichtert, hinterlegt. Wie oben erwähnt, muss auf das korrekte Timing beim Senden von mehreren Zeichen geachtet werden. Dies wird in der Funktion initLCD() und den Schreibfunktionen über Warteschleifen, die abhängig vom Prozessor und seinem Takt sind, sichergestellt. Alternativ kann hier das Timer-Modul „PTM“ verwendet werden, um noch genauere Wartezyklen zu erreichen. Die weiteren Funktionen dienen der bequemeren Handhabung. So ist das Senden von Strings beliebiger Länge mit einem Befehl möglich.

Hier eine Übersicht über die Funktionen:

Funktion	**Beschreibung**
waitLCD(time)	Wartezyklus, realisiert über eine Schleife. Die Wartezeiten hängen vom Takt ab und sind über Makros festgelegt.
initLCD()	Ruft waitLCD() für die Zeit auf, die das LCD zu Beginn für die Initialisierung braucht.
writeStringtoLCD(start, text, length)	Schreibt von einer gewünschten Position (start) an, eine Anzahl (length) von Zeichen (text) auf das Display.

Funktion	**Beschreibung**
writeChartoLCD(text)	Schreibt an der aktuellen Position ein Zeichen in das Display.
writeInsttoLCD(inst)	Sendet eine Instruktion an das Display.
clearLCD()	Löscht das Display.
LCDonoff()	Schaltet das Display an bzw. aus.
shiftString(line, pointer, length, speed)	Diese Funktion lässt einen beliebig langen Text auf der oberen oder unteren Zeile des LCD durchfließen. Zusätzlich kann die Geschwindigkeit angegeben werden.
PGLogo()	Stellt den PG-Titel auf dem LCD dar (PG 533: CoaCh, Car on a Chip).
PGIntro()	Ein kurzes Intro, das alle Namen der Teilnehmer zeigt.

Tabelle 4: Funktionen des LCD

5.2.6 SPI-Controller

Das Serial Peripheral Interface (SPI) [24] ist ein einfacher synchroner Datenbus, der von Motorola entwickelt wurde. Dabei handelt es sich um einen Vier-Draht-Bus, mit dem verschiedene Komponenten miteinander verbunden werden können. Es muss immer einen ausgezeichneten Bus-Master geben, der die anderen Busteilnehmer (Slaves) mit dem Taktsignal versorgt. Für den Takt gibt es eine eigene Leitung namens „SCK“ (Serial Clock). Der Datentransfer erfolgt full-duplex über zwei Datenleitungen mit den Bezeichnungen „MOSI“ (Master Out Slave In) und „MISO“ (Master In Slave Out). Zusätzlich gibt es noch mindestens eine „CS“-Leitung (Chip-Select), über die der Master einen Slave zur Kommunikation auswählt und sich mit diesem synchronisiert. Üblicherweise wird immer ein Byte übertragen, so dass der Master die Polarität der „CS“-Leitung nach jedem Byte kurz wechselt, um dem Slave einen neuen Transfer zu signalisieren. Pro Takt wird immer genau ein Bit vom Master zum Slave und umgekehrt transferiert. Daher kann man sich ein 8-Bit-Datenregister des Masters zusammen mit einem 8Bit-Datenregister des

Slaves als ein ringförmiges 16-Bit-Schieberegister, das pro Datentransfer um 8 Stellen geschoben wird, vorstellen.

Der Atmel ATmega103 besitzt eine SPI-Schnittstelle, welche bei dem Softcore „AVR-Core“ von www.opencores.org fehlt. Wir haben uns also nach frei verfügbaren SPI-Controllern umgesehen. Auf www.opencores.org fanden wir zwei Controller, die allerdings in Verilog geschrieben sind und damit nur bedingt für uns geeignet waren, so dass wir uns letztendlich entschieden, einen eigenen SPI-Controller für den AVR-Core in VHDL zu implementieren.

Die Implementierung des SPI-Controllers beschränkt sich zur Zeit auf die Master-Seite der SPI-Schnittstelle. Das liegt daran, dass wir bis jetzt keine anderen SPI-Master-Bausteine zur Verfügung hatten und auch noch keine Notwendigkeit bestand, den AVR-Core als SPI-Slave zu betreiben. Die Chip-Select-Leitungen wurde daher durch einfache GPIO-Ports ersetzt um den SPI-Controller möglichst klein und einfach zu halten. Die Register des SPI-Controllers stimmen genau mit denen des originalen ATmega103 überein, so dass sich der SPI-Controller bequem über die bereits vorhandene avr-libc ansteuern lässt. Es gibt ein Statusregister (SPSR), ein Kontrollregister (SPCR) und ein Datenregister (SPDR). Im Kontrollregister lassen sich ein Taktvorteiler einstellen, Interrupts aktivieren und einer von vier verschiedenen Betriebsmodi auswählen. Dabei kann eingestellt werden, ob bei der ersten Taktflanke die Daten übernommen werden sollen, oder ob dies erst bei der zweiten Flanke geschehen soll und welche Polarität das Taktsignal haben soll. Da dies nicht von Motorola spezifiziert wurde, gibt es also vier unterschiedliche SPI-Betriebsmodi. Das Datenregister dient zum Auslesen der empfangenen Daten und zum Senden von Daten. Wenn in das Datenregister geschrieben wird, so wird ein neuer Datentransfer eingeleitet.

Architektur: Der SPI-Controller besteht aus einer VHDL-Entity mit vier Prozessen. Es gibt einen kleinen Prozess, der zur Auswahl des Taktvorteilers zuständig ist. Ein weiterer Prozess steuert dann darauf aufbauend die eigentliche Taktgenerierung, womit der gesamte SPI-Bus getriggert wird. Der interne Zustand des Controllers sowie das Senden von Interrupts wird von einem eigenen Prozess verwaltet. Weiterhin gibt es noch einen Prozess, der für den Datentransfer zuständig ist, d. h. die MISO/MOSI-Leitungen ansteuert. Alle Prozesse werden synchron getriggert, um ein möglichst einfaches Design und eine einfache Synthetisierung zu ermöglichen.

Analog-Digital-Wandler: Auf dem Xilinx Spartan 3E Starter Kit ist ein Analog-Digital-Wandler (ADC) der Firma Linear Technology [29] enthalten. Dieser hat zwei Kanäle mit je 14 Bit Auflösung und einen vorgeschal-

teten Operationsverstärker (AMP), der konfigurierbar ist. Der ADC sowie der AMP werden über SPI angesprochen, weisen aber unterschiedliche SPI-Betriebsmodi auf. Betreiben kann man den ADC bei einer Sample-Rate von maximal 1,5 MHz. Um den ADC und AMP bequem aus einem C-Programm zu benutzen, haben wir dafür einen Treiber geschrieben. Dieser baut auf der avr-libc auf und benutzt den von uns erstellen SPI-Controller mit Interrupts, d. h. das Auslesen von analogen Werten verbraucht kaum Prozessorzeit. Der Treiber bietet folgende Funktionen:

`adc_init()`:

Initialisiert die Operationsverstärker und den ADC.

`adc_sample()`:

Starten eine neue Analog-Digital-Wandlung.

`adc_rdy()`:

Gibt 1 zurück, falls die Wandlung abgeschlossen ist.

`adc_recv(int *channel_A, int *channel_B)`:

Hiermit kann man das Ergebnis einer Wandlung auslesen. Ist ein Parameter ein Null-Pointer, so wird der entsprechende Kanal nicht ausgelesen.

6 Demonstratoren

Im Folgenden werden drei Aufbauten vorgestellt, die wir mit Hilfe der von uns auf FPGAs integrierten SoCs errichtet haben.

In Abschnitt 6.1 wird ein Aufbau, bestehend aus einem Audi-Lenksäulenmodul und Gas- und Bremspedal aus einem PC-Spielecontroller, vorgestellt.

Abschnitt 6.2 beschäftigt sich mit einem weiteren Testaufbau, bestehend aus zwei Scheinwerfern und einem Lenkrad, beides jeweils von Audi.

Im letzten Abschnitt 6.3 wird ein CAN-Gateway beschrieben, mit Hilfe dessen man die Möglichkeit hat, CAN-Busse mit verschiedenen Transferraten (High-Speed- und Low-Speed-CAN) miteinander kommunizieren zu lassen.

6.1 Lenksäulenmodul

6.1.1 Beschreibung

Der Aufbau beinhaltet ein original Lenksäulenmodul von Audi, sowie Gas- und Bremspedal eines Spielelenkrads. Ziel war es die Informationen über Lenkwinkel, Gas und Bremse mit Hilfe von CAN-Nachrichten an ein Spiel namens TORCS zu senden. TORCS ist ein Open-Source-Autorennspiel. Im Rahmen der Vorgängerprojektgruppe AutoLab wurde es so erweitert, dass es über spezielle CAN-Nachrichten gesteuert werden kann. Damit das Spiel nicht erneut verändert werden musste, konvertiert der AVR-Softcore die Nachrichten in das notwendige Format.

6.1.2 Hardware

Das Lenksäulenmodul, das Spartan-3E-Entwicklungsboard und ein CAN-USB-Adapter sind über einen Highspeed-CAN-Bus verbunden. Die Pedale bestehen aus zwei Potentiometern, die in Reihe geschaltet sind, sodass jede Pedalstellung einen bestimmten Widerstand hervorruft. Durch einen Spannungsteiler wird dieser Widerstand proportional umgewandelt in eine Spannung. Diese kann man schließlich mit dem auf dem Spartan-3E-Entwicklungsboard verbauten Analog-Digital-Wandler messen. Die Kommunikation mit dem Analog-Digital-Wandler erfolgt dabei über den selbst geschriebenen SPI-Controller, der in den AVR integriert wurde.

6.1.3 Programm

Das Programm in dem AVR ist einfach gehalten: Es liest die Lenkwinkel-Nachricht über den CAN-Bus, interpretiert und konvertiert diese Daten und

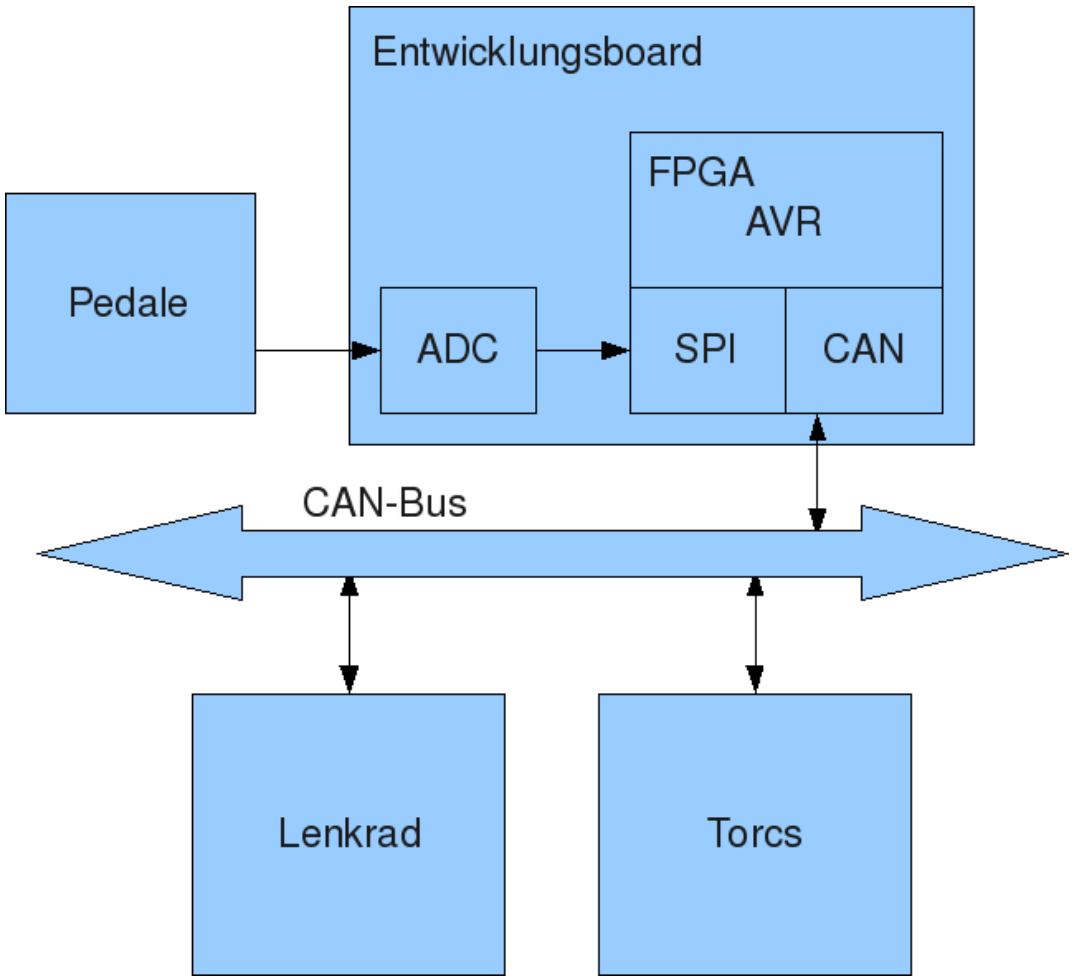

Abbildung 8: Schematische Darstellung des Testaufbaus

sendet sie unter einer neuen ID auf den Bus. Anschließend wird die Messung der Pedalstellung durchgeführt. Dieser Messwert wird ebenfalls in eine CAN-Nachricht verpackt und an das Spiel gesendet. Hierbei muss beachtet werden, welche IDs das Spiel erwartet und wie die Daten aussehen sollen. Abbildung 9 zeigt den Lenkrad-Testaufbau.

6.2 Scheinwerfer

6.2.1 Beschreibung

Der Testaufbau unserer Gruppe umfasst das Spartan-3E-Entwicklungsboard sowie die beiden Audi-Scheinwerfer und das Lenkrad. Ziel ist es über CAN-Nachrichten die Lichter an- und auszuschalten (Taglicht, Abblendlicht, Fern-

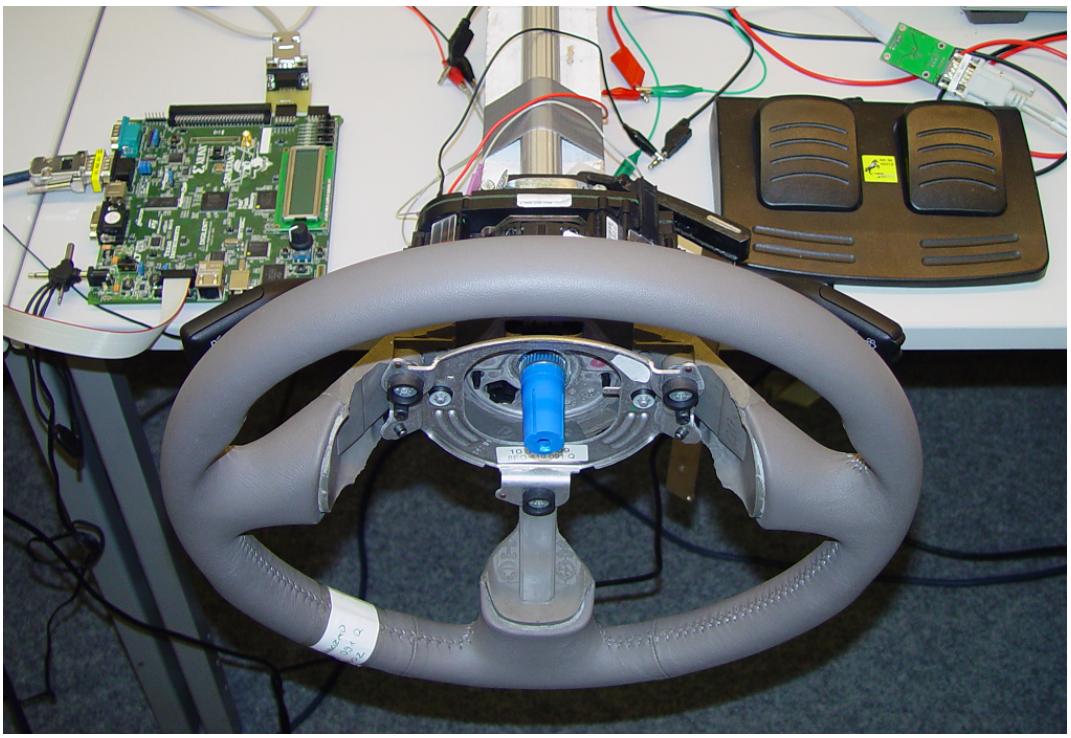

Abbildung 9: Lenkrad-Testaufbau

licht, Blinklicht) und den Schwenk- und Neigungswinkel des Abblendlichtes zu steuern. Dazu ist das Lenkmodul an den Highspeed-CAN-Bus angeschlossen und sendet laufend Nachrichten über den aktuellen Lenkwinkel. Die restlichen Nachrichten werden mit Hilfe des Diagnosetools „CANoe“ verschickt. Unsere Software auf dem HC08 verarbeitet diese Nachrichten und steuert die Scheinwerfer bzw. sendet selbst entsprechende CAN-Nachrichten.

6.2.2 Hardware

Scheinwerfer Der Gruppe stehen zwei Scheinwerfer von Audi zur Verfügung. Diese verfügen über Taglicht (14 in Reihe liegende LEDs), Abblendlicht und Fernlicht (lässt sich über umklappen eines Spiegels anschalten) sowie Blinklicht. Die Lichter werden einzeln mit 12V betrieben. Da die Spartan-Boards nur 3,3V liefern können, wird mittels Transistoren die Spannung auf 12V gewandelt. Das Abblendlicht kann über einen Motor horizontal und vertikal bewegt werden. Die LEDs des Taglichtes sind über Reduzierung der Spannung dimmbar.

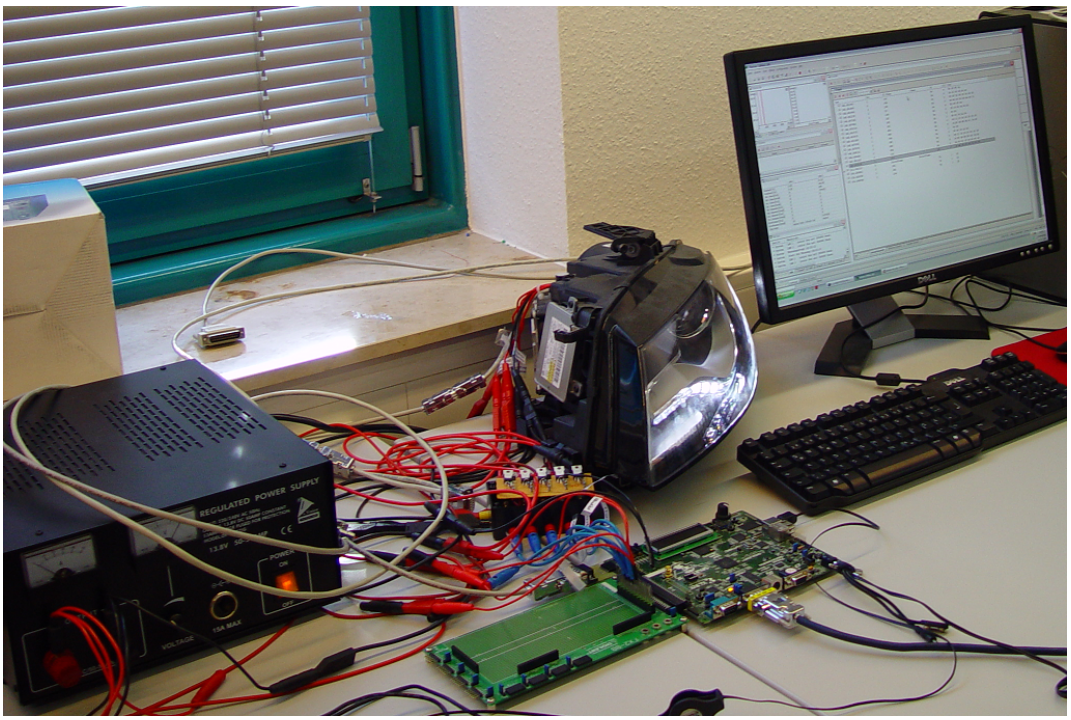

Abbildung 10: Scheinwerfer-Testaufbau

Lenkrad Das Lenkrad besitzt einen Lenkwinkelsensor und sendet regelmäßig den Lenkwinkel als Nachricht über den Highspeed-CAN.

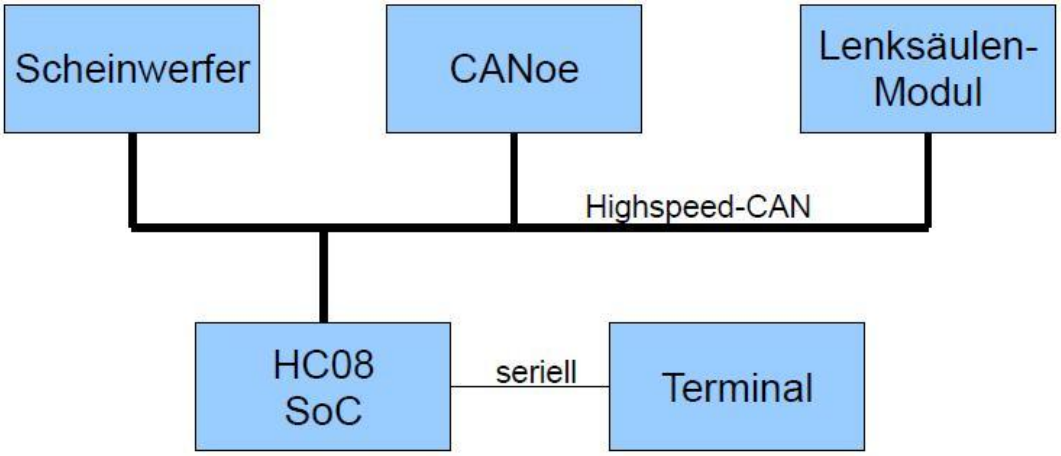

Abbildung 11: Schematische Darstellung des Testaufbaus

6.2.3 Pretty Timer Module

Da zur Ansteuerung einiger Komponenten sekundengenaues Warten nötig ist, wurde ein Hardware-Timer in den Aufbau integriert. Das Pretty-Timer-Modul verfügt dafür über einen einfacher 16-Bit-Zähler. Ein Prescaler (1, 4, 16, 64, 256, 1024 und 4096) ermöglicht es dem Timer, auch für mehr als 2^{16} Wartezyklen eingesetzt zu werden. Neben dem einfachen Abfragen des aktuellen Zählerstandes verfügt das PTM über einen Interruptausgang sowie eine automatische Reset-Funktion nach Zählerüberlauf.

6.2.4 Taglicht, Abblendlicht, Fernlicht und Blinklicht

Die einzelnen Lichter der Scheinwerfer werden einfach durch Anlegen der Spannung zum Leuchten gebracht. Unser SoC besitzt ein LIGHT-Modul, das über ein Register gesteuert wird und nach Wunsch die richtigen PINs am Spartanboard mit Spannung versorgt. So lassen sich alle Leuchten seperat aktivieren. Weiterhin können über die Werte des Registers acht verschiedene Spannungswerte zum Dimmen der LEDs festgelegt werden. Die Spannung wird mit Pulsweitenmodulation geregelt. Wegen des für LEDs typischen Leuchtverhaltens steigen die acht Spannungswerte quadratisch, um einen möglichst linearen Anstieg der Helligkeit zu erhalten. Über das Pretty-Timer-Modul (PTM) wird das Intervall für das Blinklicht auf eine Sekunde geregelt.

6.2.5 Dynamisches Kurvenlicht

Das Lenkrad sendet in regelmäßigen Abständen CAN-Nachrichten, die den Lenkwinkel angeben. Diese Informationen werden werden von unserer Software in CAN-Nachrichten gewandelt, die den Motor für die Position des Abblendlichtes steuern. Bei einer Bewegung des Lenkrades um bis zu 90 Grad nach links oder rechts folgen die Scheinwerfer der Lenkrichtung. Die Neigung der Abblenlichtes kann wiederum über CAN-Nachrichten gesteuert werden.

6.3 CAN-Gateway

6.3.1 Beschreibung

Bei der Herstellung von Automobilen, die einen CAN-Bus verwenden, ist es durchaus üblich mehrere CAN-Geschwindigkeitsvarianten zu verbauen. „Wichtige“ Komponenten, wie Lenkung, Bremsen, Motorsteuerung und ähnliche Geräte, die harte Echtzeitbedingungen erfüllen müssen, kommunizieren dabei über einen Hochgeschwindigkeits-CAN, während weniger wichti-

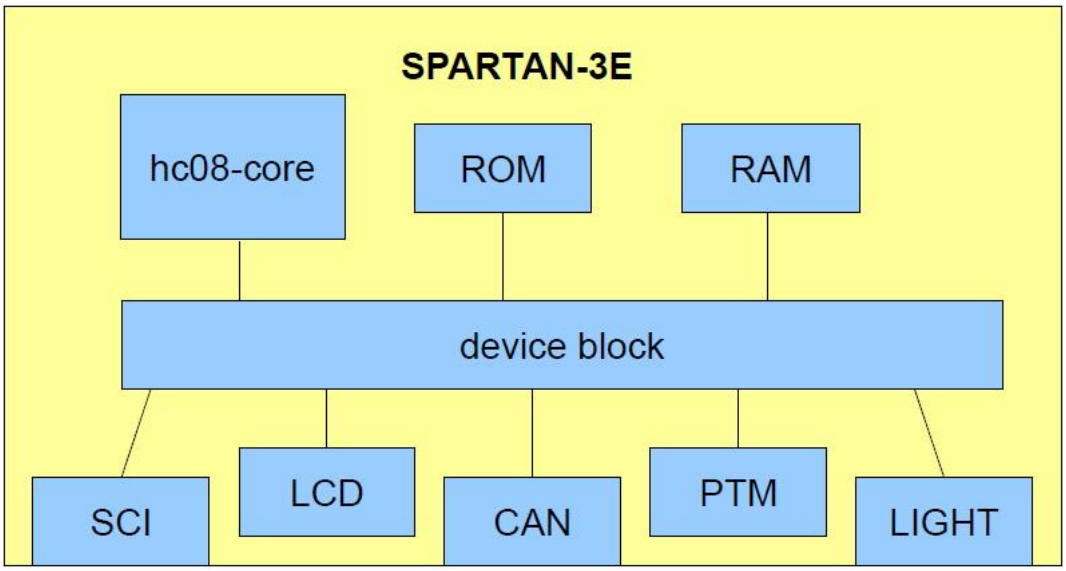

Abbildung 12: Schematische Darstellung des HC08-System on Chip

ge Komponenten, wie z. B. Schiebedach oder die Reifendruckkontrolle, über langsamere Busse laufen. Dennoch ist es häufig notwendig, dass Komponenten, z. B. zu Diagnosezwecken, auf Informationen zurückgreifen müssen, die auf dem jeweils anderen Bus übertragen werden. An dieser Stelle bedarf es eines Gateways, der Nachrichten zwischen den unterschiedlichen Bustypen weiterleitet und bei Bedarf auch filtert.

6.3.2 Hardware

Innerhalb unseres Projektes haben wir so ein Gerät mittels eines t51-Softcores realisiert. Am FPGA sind zwei verschiedene CAN-Transceiver angeschlossen, die jeweils mit unterschiedlichen Bussen kommunizieren. Auf VHDL-Ebene befinden sich neben dem t51 noch zwei CAN-Controller, die beide an die XRAM-Leitungen des t51 angeschlossen sind.

6.3.3 Software

Die Software empfängt unterbrechungsgesteuert Nachrichten von einem CAN-Controller, filtert diese und überträgt sie an den jeweils anderen Controller. Das Filtern hat im wesentlichen zwei praktische Vorteile: Zum einen ist es nicht immer notwendig, alle Nachrichten eines CAN-Busses an den anderen Bus weiterzuleiten und verhindert somit einen unnötigen Informationsfluss. Zum anderen verhindert das Filtern, dass eben übertragene Nachrichten

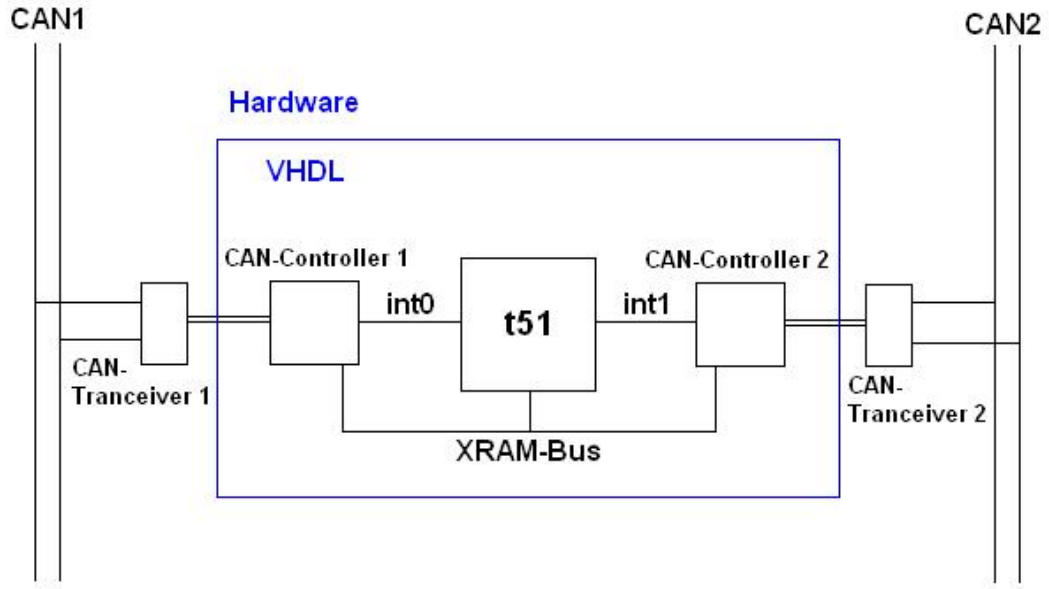

Abbildung 13: Schematische Darstellung eines CAN-Gateways

sofort wieder vom Gateway aufgegriffen und an den anderen Bus zurück übertragen werden. Wenn wir eine Motorsteuerungskomponente gegeben haben, die an einem High-Speed-CAN angeschlossen ist und ihre Nachrichten an ein Diagnosegerät leiten wollen, das mit einem Low-Speed-CAN verbunden ist, so ist das Gateway sinnvollerweise so konfiguriert, dass es Nachrichten mit der Motorsteuerungs-ID nur in der High-zu-Low-Richtung weiterleitet.

7 NoC-Integration

7.1 On Chip CAN-Bus

7.1.1 Beschreibung

Die integrierte Schaltung (Abbildung 14) enthält drei unabhängige Mikrocontroller (μC 1 bis 3), in denen je ein CAN-Controller (5.2.4) eingebettet ist. Über diese CAN-Bus-Schnittstellen sind alle Mikrocontroller miteinander vernetzt und bilden ein sogenanntes NoC (2.2). Weiterhin ist dieses NoC über genau einen CAN-Transceiver mit externen Mikrocontrollern über den physikalischen CAN-Bus verbunden. Der Buszugriff erfolgt auf der integrierten Schaltung genauso wie auf den externen Differenzsignalleitungen (2.13). Die `UND`-Verknüpfung der Sendeleitungen (`tx`) stellt sicher, dass immer nur die höchst priorisierte Nachricht übermittelt wird. Diese „on chip"-Arbitrierung ist für die einzelnen Mikrocontroller völlig transparent, so dass bereits bestehende Software ohne Modifikationen problemlos funktioniert. Sowohl im Fahrzeug können CAN-Nachrichten mit anderen Steuergeräten ausgetauscht werden, als auch die Kommunikation der einzelnen Mikrocontroller untereinander (z. B. Interprozesskommunikation) über CAN-Nachrichten ist möglich. Die Adressierung ist dabei in beiden Fällen exakt gleich und erfolgt über die CAN-ID im Header der Nachricht.

Zu beachten ist, dass das rechte `UND`-Gatter des on-Chip-CAN-Busses zwar nicht zwingend nötig ist, dafür aber zwei Vorteile bietet: Zum einen lässt sich so die CAN-Bus-Kommunikation mit einem Logiksimulator simulieren, denn eine Nachricht gilt nur dann als erfolgreich versendet, wenn der sendende CAN-Controller seine eigene Nachricht selbst korrekt empfängt. Dazu ist die „Rückkopplung" der Sendeleitungen (`tx`) über das rechte `UND`-Gatter notwendig. Zum anderen ermöglicht diese Schaltung eine einfache Kommunikation der einzelnen Mikrocontroller untereinander ohne externen physikalischen CAN-Bus.

7.1.2 Vorteile

Der on-Chip-CAN-Bus bietet eine hardware-unterstützte Isolation der einzelnen Mikrocontroller. Das Programmiermodell entspricht dabei exakt dem eines einzelnen Controllers. Es lassen sich also beliebige Mikrocontroller miteinander vernetzen, welche durchaus heterogen seien können. Dabei müssen weder die Hardware- noch die Softwarekomponenten angepasst werden. Es entsteht somit keine weitere Komplexität durch Isolationsmechanismen in Software, so dass eine potentielle Fehlerquelle generell ausgeschlossen wird. Weiterhin benötigt ein solches NoC nur genau einen CAN-Transceiver zur

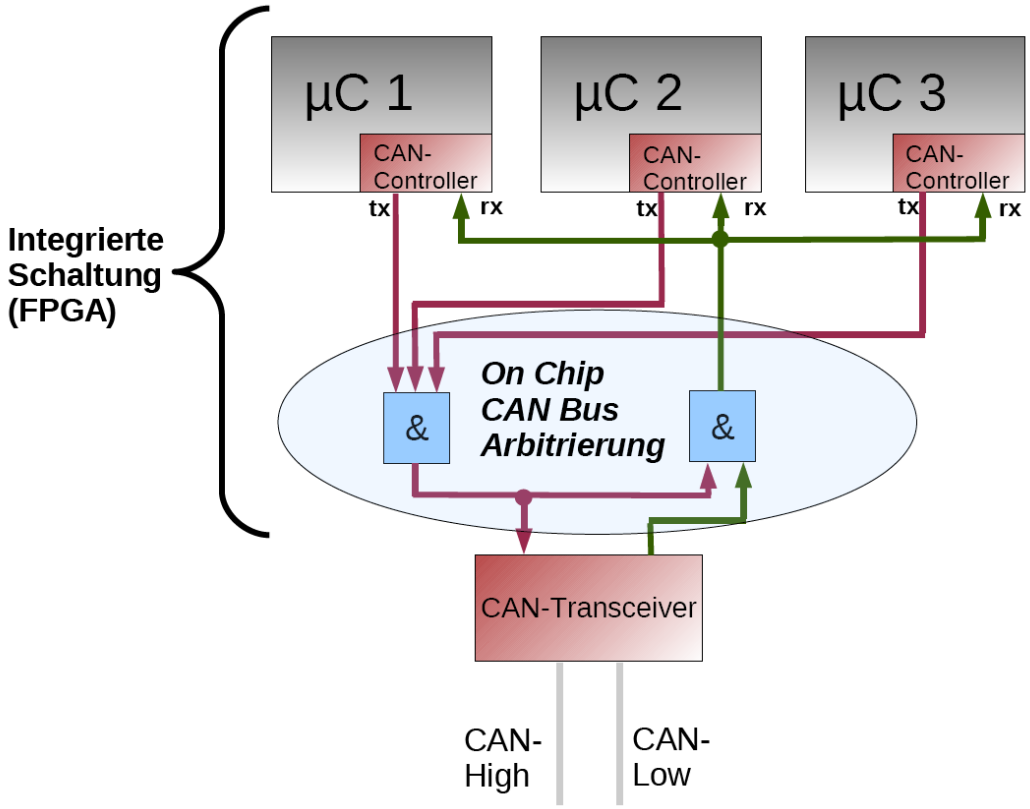

Abbildung 14: On Chip CAN-Bus

Anbindung an den physikalischen CAN-Bus, wodurch Hardware- und Energiekosten eingespart werden können.

7.1.3 Nachteile

Zum Versenden und Empfangen von CAN-Nachrichten benötigt jeder Mikrocontroller einen eigenen CAN-Controller. Ein solcher CAN-Controller benötigt auf einem Xilinx Spartan3E FPGA etwa 1200 Lookup-Tabellen (LUTs) und hat damit einen nicht unerheblichen Hardwarebedarf. Dies entspricht etwa der Hälfte einer der von uns verwendeten 8-Bit-CPUs. Einfache Kommunikationsschnittstellen wie UART (5.2.3) oder SPI (5.2.6) verbrauchen mit etwa 300 LUTs vergleichsweise wenig, bieten aber natürlich deutlich weniger Funktionalität, unter anderem keine inhaltsbasierte Adressierung und Arbitrierung. Bei der Verwendung von 16- oder 32-Bit-CPUs relativiert sich dieser Nachteil, da eine solche CPU viel mehr Hardwarekapazität als ein CAN-Controller benötigt.

7.1.4 Isolation durch ein Gateway

Eine Erweiterungsmöglichkeit für den on-Chip-CAN-Bus stellt ein Gateway (6.3) dar, welches auf die Schaltung (FPGA) integriert wird. Dieses Gateway kann dazu eingesetzt werden, den on-Chip-CAN-Bus und den externen physikalischen CAN-Bus zu entkoppeln. Es entstehen also zwei völlig getrennte Bussysteme, und das Gateway leitet nur solche Nachrichten weiter, die für den jeweils anderen Bus adressiert sind. Sinnvoll ist diese Lösung, wenn zwischen den einzelnen Mikrocontrollern des on-Chip-CAN-Busses Interprozesskommunikation (IPC) über CAN-Nachrichten stattfindet. Über das Gateway können Störungen auf dem externen physikalischen CAN-Bus nicht auf den on-Chip-CAN-Bus übertragen werden, so dass dort unabhängig eine ausfallsichere Kommunikation stattfindet. Weiterhin wird so die Buslast auf dem physikalischen CAN-Bus reduziert, da die IPC-Nachrichten nur auf dem on-Chip-CAN-Bus transportiert werden. Allerdings hat ein CAN-Gateway auch seinen Preis: Ein 8-Bit-Mikrocontroller mit zwei CAN-Controllern benötigt bis zu 4500 LUTs, so dass der Einsatz eines Gateways wohl überlegt sein muss.

7.2 Memory-Multiplexing

7.2.1 Vorteile von DDR-RAM

Bislang verwendete jedes System-on-Chip seinen eigenen RAM, der exklusiv als Block-RAM auf dem FPGA synthetisiert wurde. Die Anzahl der zur Verfügung stehenden Block-RAMs auf den Spartan- sowie auf den Virtex-Boards ist jedoch begrenzt. Bei der Synthese mehrerer Cores als Network-on-Chip ist es daher sinnvoll, den DDR-RAM auf den Boards mitzubenutzen.

7.2.2 Multiplexing

Auf dem Spartan-3E Starter Kit Board sind 64MB DDR-Speicher integriert. Um jedem Core den Zugriff auf den DDR zu ermöglichen und gleichzeitig die strikte Trennung der einzelnen Cores untereinander zu gewährleisten, entwickelten wir einen Memory-Multiplexer. Dieser teilt momentan acht verschiedenen CPUs einen 16 Bit breiten Adressbereich des DDR_RAMs zu. Das Design des Multiplexers besitzt hierbei noch genug Spielraum, um die Anzahl der Cores um ein Vielfaches zu erweitern. Die Trennung der Adressbereiche innerhalb des Multiplexers ist so gestaltet, dass keine CPU auf den Addressraum einer anderen zugreifen kann. Somit ist die strikte Trennung der Systeme sichergestellt.

7.2.3 Anpassung des DDR-Controllers

Der Memory-Multiplexer basiert auf einem DDR-Controller, der Teil des Plasma-Microcontroller-Cores [22] ist. Dieser DDR-Controller ist auf das Micron 46V32M16-DDR-Modul angepasst, welches sich auch auf dem Spartan-Board befindet. Um diesen Controller für unser Network-on-Chip zu nutzen, waren einige Anpassungen notwendig.

Der DDR-RAM auf dem Spartan-Board muss richtig initialisiert werden, bevor er benutzt werden kann. Dies muss beim DDR-Controller aus dem Plasma per Software mittels Zugriff auf entsprechende Adressen im DDR-RAM gemacht werden. Wollen wir mehrere CPUs auf den DDR-RAM zugreifen lassen, kommt eine Initialisierung per Software nicht mehr in Frage. So wurde zuerst die Initialisierung in die Logik des DDR-Controllers verschoben. Nach einem Reset wird der DDR-RAM neu initialisiert und kann von den einzelnen Cores nach ca. 210 Taktzyklen verwendet werden. In der Zeit der Initialisierung werden alle Cores angehalten, damit es keine verfrühten Zugriffe gibt.

Tiefergehende Änderungen waren nötig, um den Controller kompatibel zum Memory-Multiplexer zu machen. Das Scheduling der Zugriffe der verschieden Cores wurde im Round-Robin-Verfahren implementiert. Schwierigkeiten machte hierbei besonders das genaue Timing zwischen den Zugriffen verschiedener Cores. So entsteht eine recht hohe Verzögerung bei jedem Schreib- und Lesezugriff. Damit ist bei einer hohen Anzahl verwendeter Cores und exzessiven Zugriffen auf den DDR-RAM nicht ausgeschlossen, dass ein Core blockiert wird. Ein dynamisches Scheduling-Verfahren mit Prioritäten wäre hier nötig, kann aber im Rahmen der Projektgruppe nicht mehr implementiert werden.

7.3 Network-on-Chip-Debugging

Bei der Entwicklung von eingebetteten Systemen ist ein effizientes Debugging von großer Bedeutung. Da ein Mikrocontroller bzw. FPGA naturgemäß über beschränkte Ausgabemöglichkeiten verfügt, ist es wichtig, dass sich der Entwickler zu den für ihn wichtigen Informationen Zugang verschafft, wie z. B. dem Zustand des System. Hierbei könnten unter anderem der Inhalt des Speichers bzw. der Register von Bedeutung sein. Eine einfache Lösung ist die Ausgabe von Debug-Informationen über den UART oder eine Zustandsanzeige über LEDs. Jedoch ist es ebenfalls wünschenswert, dass man von außen in die Arbeit des Rechners eingreifen kann, z. B. Speicherinhalte verändern oder den Rechner sein Programm schrittweise ausführen lassen.

7.3.1 Der GNU Debugger

Der GNU Debugger ist ein sehr leistungsfähiges Tool in der Entwicklung von eingebetteten Systemen. Hierbei kann von einem Host-System aus, auf dem die GDB-Software läuft, das Zielsystem über eine serielle Schnittstelle oder Netzwerkverbindung bei der Ausführung der zu untersuchenden Software beobachtet und gesteuert werden. Von der Kommandozeilenoberfläche aus ist es möglich den Zielrechner neuzustarten, schrittweise Code auszuführen, Speicherinhalte auszulesen oder zu verändern. Hierzu ist es notwendig, dass der Zielrechner über einen sogenannten Stub verfügt, der mit dem GDB-Hostsystem kommuniziert und die vom Host gesendeten Befehle auf dem Zielsystem ausführt. Die Kommunikation ist durch das sogenannte ASCII-basierte „GDB Remote Serial Protocol“ standardisiert. Eine GDB-Kommunikation beginnt immmer mit einem Dollarzeichen, es folgen der Befehl des Hostes und danach die Antwort des Zielsystems. Die Kommunikation endet mit einer Raute und wird von der Checksumme abgeschlossen. Die Checksumme wird über die Zeichen zwischen dem Dollarzeichen und der Raute gebildet. Eine erfolgreiche Kommunikation wird von einem '+' quittiert. Sollte die Checksumme nicht korrekt sein oder die Syntax des Befehls nicht stimmen, wird ein '-' ausgegeben. Nachfolgend eine Auswahl von nützlichen GDB-Befehlen und ihrer Syntax:

Funktion	**Befehl**	**Beispiel**
Register lesen	g	g
Register 0 mit 1234, Register 1 mit abcd beschreiben	G	$G1234abcd#
Register 0x10 beschreiben	P	$P10=0040149c#
Step	s	$s#
Continue	c	$c#

7.3.2 Der GDB-Stub

Im Rahmen unserer PG haben wir begonnen, einen eigenen Stub in VHDL zu entwickeln. Da die Arbeit am Stub nicht komplett abgeschlossen werden konnte, folgt nun eine Beschreibung der angestrebten Funktionalitäten.

Unsere Testumgebung beinhaltet zwei T51-Softcores als Zielsysteme, die Ausgaben über ihren UART liefern. Bei den Cores sind die Registerinhalte nach außen gelegt worden, sodass sie vom Stub ausgelesen werden können. Des Weiteren wird der Takt für alle Komponenten von einem Clockgenerator zur Verfügung gestellt. Der Stub kann über Enable-Signale dem Generator mitteilen, dass er einem bestimmten Core den Takt entziehen soll, um ihn anzuhalten. Der Stub ist weiterhin mit einem sogenannten Minimal UART

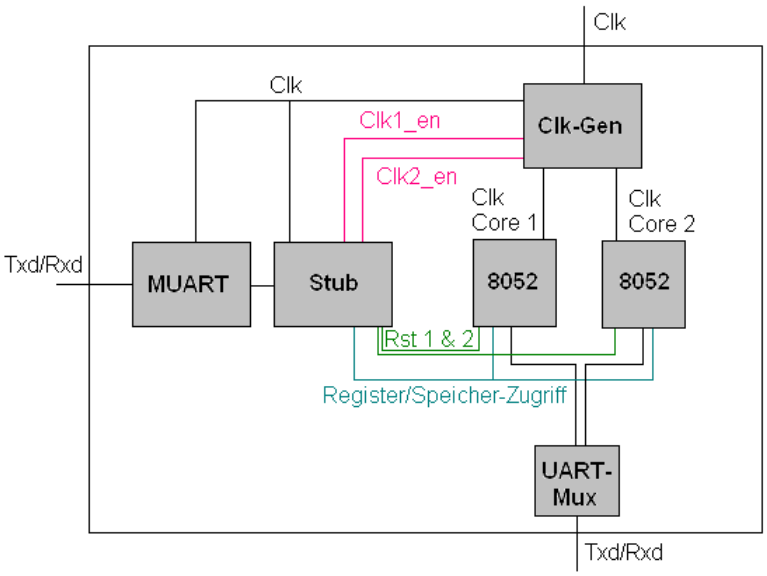

Abbildung 15: Schematische Darstellung der Stub-Testumgebung

(MUART) verbunden, der als Schnittstelle zum Hostsystem fungiert. Dieser ist in VHDL implementiert und kann nur im 8-Bit-Modus Senden bzw. Empfangen.

Intern ist der Stub als Zustandsautomat in einem VHDL-Prozess realisiert, der sensitiv auf eingehende Zeichen im MUART reagiert. Die korrekte Kommunikation beginnt mit einem Dollarzeichen. Nachfolgend wird die Eingabe des Zielrechners in Form einer Zahl erwartet. Dieser Zustand ist nicht in dem ursprünglichen „GDB Remote Serial Protocol“ vorgesehen. Da aber ein Debugging für ein Multicore-System betrieben werden soll, muss die Auswahl des Zielrechners erfolgen. Daraufhin kommt der Automat in den Befehlsauswahlzustand, in dem auf eines der folgenden Zeichen gewartet wird:

- g: Register auslesen
- G: Register beschreiben
- m: Speicher auslesen
- M: Speicher beschreiben
- S: Zielrechner stoppen
- s: Step
- c: Betrieb fortsetzen

- r: Zielrechner resetten

Auch hier weicht die Implementierung im Detail etwas vom Protokoll ab, da im Protokoll mehr Befehle vorgesehen sind bzw. anders implementiert werden. Zum Beispiel ist es nicht vorgesehen ein Zielsystem auszuschalten, da aber mehrere Soft-Cores debuggt werden sollen, ist dies in diesem Kontext eine nützliche Funktion.

Nach der Befehlsauswahl wird der Befehl ausgeführt. Bei Auslese-Befehlen soll der Inhalt einer vom Core ausgehenden Leitung in ASCII umgewandelt und über den MUART ausgegeben werden. Bei Speicherbefehlen sollen die zu den Registern bzw. zu dem Speicher führende Leitungen mit der Eingabe des Hostrechners belegt werden. Bei Stopbefehlen wird, wie oben beschrieben, die zum Clockgenerator führende Enable-Leitung auf '0' gesetzt, worauf der Clockgenerator den Takt zum jeweiligen Core aussetzt. Bei einem Step-Befehl werden dem Ziel 12 Takte zugeführt, da ein Befehlszyklus beim 8052 aus ebensovielen Takten besteht.

Zum Ende der Projektgruppe existierte die Anbindung der einzelnen Teilkomponenten und des Zustandsautomaten des Stubs. Die Funktionen des Stubs waren noch nicht implementiert.

8 NoC-Testaufbau

Ziel des zweiten Semesters der PG war die Integration der im ersten Semester entwickelten Systems-on-Chip zu einem Network-on-Chip. Das folgende Kapitel beschäftigt sich mit dem von uns entwickelten Testaufbau.

8.1 Ausgangssituation

Um die entwickelte Plattform zu evaluieren, wurde als Demonstrator der ebenfalls am Lehrstuhl entstandene „DLR SchoolLab“-Aufbau gewählt. Bei diesem handelt es sich um das Netzwerk eines fiktiven Autos, dessen Verhalten im Rahmen von Schülerexperimenten mit der sogenannten „Autolab-Entwicklungsumgebung“ grafisch programmiert werden kann (siehe Abbildung 18). Das System besteht aus mehreren Mikrocontroller-Boards, welche über einen CAN-Bus miteinander vernetzt sind. In den Versuchsaufbau eingebunden sind eine Reihe von echten Automobilkomponenten. So sind neben den beiden Scheinwerfern aus dem aktuellen Modell des Audi A6, ein Gangwahlschalter aus dem BMW X5, ein Dachmodul aus dem 3er BMW sowie ein Lenksäulenmodul aus dem Audi A4 integriert. Desweiteren wurde das Gaspedal eines Spielzeug-Lenkrades in den Versuchsaufbau eingebunden. Unser Ziel bestand darin, dieses System mittels unseres Ansatzes in einem FPGA funktionsgetreu nachzubilden und dadurch unser Konzept zu evaluieren. Im Folgenden soll zunächst ein Einblick in den Aufbau des nachzubildenen Systems gegeben werden.

8.2 Funktionale Komponenten

8.2.1 Scheinwerfersteuerung und Scheinwerfer

Eine Komponente des Systems bilden die beiden vorderen Scheinwerfermodule. Jedes von ihnen enthält die typischen aus neueren Personenkraftwagen bekannten Bestandteile, insbesondere:

- die Leuchte für das Abblend- bzw. Fernlicht,
- eine elektromagnetisch bewegbare Linse zur Beeinflussung der vertikalen (Auf- bzw. Abblenden) und horizontalen (Kurvenlicht) Richtung des ausstrahlenden Lichtkegels,
- eine Leiste aus Leuchtdioden für das Tagfahrlicht, und
- die Leuchte des Fahrtrichtungsanzeiger (ugs.: Blinker).

Die Schnittstelle eines Scheinwerfermoduls zum restlichen Kraftwagen besteht aus mehreren Anschlüssen. Sie enthält eine Reihe von analogen Anschlüssen mit 12V-Pegel, sowie einen digitalen in Form eines CAN-Bus-Steckers. Die Busgeschwindigkeit beträgt 500.000 Bit/s. Die richtige Ansteuerung dieser Schnittstelle ist die Aufgabe eines Scheinwerfercontrollers. Dieser ist im System mit einem Tricore-Entwicklungsboard realisiert. Er nimmt die SchoolLab-Steuerbefehle, die die Scheinwerfer betreffen, vom CAN-Bus entgegen und setzt diese in weitere Scheinwerfer-kompatible Nachrichten um. Desweiteren steuert das Board eine Schaltbox. Diese besteht aus einer zweistufigen Transistorschaltung und wandelt Signale des Controllers, die einen Pegel von 3,3 Volt besitzen, in die benötigte belastungsfähige 12V-Spannung um. Im Einzelnen ist der Scheinwerfer je nach gewünschter Funktion mit folgenden Spannungen zu versorgen:

- Spannung für das Abblendlicht
- Spannung für das Fernlicht
- zur Helligkeitsregelung pulsweitenmodulierte Spannung für das Tagfahrlicht
- Spannung für die Blinkerleuchte
- ständige Spannung zur Versorgung der Stellmotoren

Der CAN-Bus-Anschluss der Scheinwerfer nimmt die vom Controller umgewandelten Nachrichten zur Steuerung des Kurvenlichtes entgegen.

Abbildung 16: Das verwendete rechte Scheinwerfermodul des Audi A6

8.2.2 Fahr- und Bremspedal

Eine weitere Komponente stellt die Fußkonsole eines Spielkonsolen-Lenkrades sowie ein Pedalcontroller dar. Die Fußkonsole besteht aus zwei Pedalen (Gas

und Bremse), dessen aktuelle Stellung sich durch die Messung des Widerstandes eines eingebauten Potentiometers ermitteln lässt. Genau dies ist auch die Aufgabe des Pedalcontrollers. Realisiert ist die Messung durch die Analog-Digital-Wandlung des Spannungspegels, der an einem Spannungsteiler bestehend aus Fußkonsolenpotentiometer und Konstantwiderstand entsteht. Die gemessenen Werte werden vom Controller in den Bereich von 0 bis 100 normiert und periodisch (100ms) über jeweils eine CAN-Bus-Nachricht für Gas- und Bremspedalstand bekanntgegeben (500.000 Bit/s). Auch die Funktion des Pedalcontrollers wird von einem Tricore-Entwicklungsboard übernommen.

Abbildung 17: Das verwendete Spielzeug-Fahr- und -Bremspedal

8.2.3 Lenksäulenmodul

Weiterhin in den Versuchsaufbau integriert worden ist ein Lenksäulenmodul, das u. a. auch im Audi A4 Verwendung findet. Das Modul besteht sowohl aus einem Drehwinkelgeber, der den aktuell eingeschlagenen Lenkwinkel ermittelt, als auch aus den üblichen am Lenkrad befestigten Scheibenwischer- und Blinkerhebeln. Im echten PKW wird von ihm auch der Airbag mit Daten aus dem CAN-Bus versorgt, der in unserem Versuchsaufbau aus naheliegenden Gründen fehlt. Die digitale Außenschnittstelle besteht aus zwei verschiedenen CAN-Bus-Anschlüssen, von denen einer im „Fault Tolerant Mode“ (auch: Low-Speed-CAN-Bus) mit 100.000 Bits pro Sekunde betrieben wird. Auf diesem sendet das Modul die Informationen über den aktuelle Status der Blinker- und Wischerhebel, während auf dem schnellen Bus (500.000 Bit/s) u. a. Informationen zum Lenkwinkel geliefert werden. Die Low-Speed-Seite des Moduls muss durch das periodische Senden von bestimmten Netzwerkmanagement-Nachrichten am „Leben“ gehalten werden. Fehlen diese, so schaltet sich dieser Teil des Moduls ab und liefert keine weiteren Statusnachrichten.

8.2.4 Gangwahlschalter

Ein softwareseitig modifizierter Gangwahlschalter aus dem BMW X5 ist ebenfalls Bestandteil des Systems. Dieser gibt per CAN-Bus Auskunft über die aktuelle Position des Fahrstufenwahlhebels, sowie den Zustand der eingebauten Taster zum Umschalten in den Park- oder Sportmodus. Diverse Leuchtdioden zur Anzeige der aktuellen Fahrstufe können über Steuernachrichten ein- und ausgeschaltet werden.

8.2.5 Dachmodul

Das Dachmodul bietet verschiedene Möglichkeiten der Innenraumbeleuchtung (gezieltes Leselicht für den linken bzw. rechten vorderen Innenraum, Komplettbeleuchtung des vorderen Innenraums, Leuchtdioden zur sehr dezenten ambienten Beleuchtung der Mittelkonsole), sowie Schaltern zur Steuerung derselben. Ähnlich wie der Gangwahlschalter verwendet es eine modifizierte Software, die primitive Nachrichten zum Schalterstatus versendet bzw. zur Leuchtensteuerung empfängt. Das Modul ist wie das Lenksäulenmodul an den Low-Speed-CAN-Bus angeschlossen.

8.2.6 CAN-Bus-Gateway

Um zwischen den beiden im System vorhandenen CAN-Bussen und den dort jeweils angeschlossenen Komponenten zu vermitteln, gibt es ein CAN-Bus-Gateway. Am High-Speed-CAN-Bus befinden sich die Scheinwerfer, der Pedalcontroller, das Lenksäulenmodul, der Zentralcontroller sowie der PC mit der AutoLab-Entwicklungsumgebung und der Autorennsimulation. Auf Low-Speed-Seite befinden sich Dach- und Lenksäulenmodul. Die Aufgabe des Gateways ist die gezielte Weiterleitung bestimmter Nachrichtentypen von einem Bus auf den anderen. Diese muss gezielt geschehen, da ansonsten der langsamere der beiden Busse durch eine hohe Auslastung des anderen vollständig saturiert würde und es zu dramatischen Nachrichtenverlusten kommen würde. Zur Realisierung kommt ein Entwicklungsboard mit einem 9S12-Mikrocontroller von Freescale zum Einsatz. Dem bereits auf dem Board vorhandenen CAN-Transceiver wurde ein zweiter nach Fault-Tolerant-CAN-Spezifikation zur Seite gestellt. Besonders geeignet ist der 9S12 in der verwendeten Variante, da der integrierte CAN-Controller bis zu vier verschiedene CAN-Busse bedienen kann.

8.2.7 Zentralcontroller / AutoLab Virtual Machine

Ein weiteres Tricore-Entwicklungsboard setzt die Funktion des Zentralcontrollers um. Dieser ist am High-Speed-CAN-Bus angeschlossen, um mit den anderen Controllern kommunizieren zu können. Die Hauptaufgabe des fiktiven Autonetzwerks übernimmt diese Komponente. Auf dem Tricore-Entwicklungsboard ist ein OSEK-Betriebssystem installiert, welches ein häufig im Automobilbereich eingesetztes Betriebssystem darstellt. Dieses Betriebssystem beherbergt die AutoLab Virtual Maschine (AVM). Die Aufgabe der AVM ist das Umsetzen des vom Anwender programmierten Verhaltens des simulierten Fahrzeugs. So kann die AutoLab Virtual Machine auf die Statusnachrichten der anderen Controller reagieren und entsprechend des Programms neue Steuernachrichten für die übrigen Steuergeräte erzeugen. So ist es zum Beispiel möglich die Statusmeldungen des Lenkrads für den Lenkwinkel zum Ansteuern der Leuchten zu verwenden. Neben den Steuergeräten für die diversen Hardwarekomponenten kann auch das Rennspiel TORCS mit entsprechenden Steuermitteilungen versorgt werden. Das aktuelle Programm wird über den CAN-Bus von der AutoLab-Entwicklungsumgebung an die AVM übertragen. Der eigentliche Transport des Programms erfolgt mit dem folgenden Protokoll.

Transportschicht Die eigens implementierte Transportschicht basiert auf der Spezifikation 15765-2:2004 der internationalen Organisation für Normung [59]. Die Hauptaufgabe der Transportschicht besteht darin, Datenpakete, die größer als 8 Byte sind, zu verschicken. Dies geschieht durch Segmentierung bzw. Desegmentierung der Daten. Dabei soll zusätzlich eine Überlastung des Empfängers durch verschiedene Maßnahmen vermieden werden. Beispielsweise wird nach dem Senden einer (Teil-)Nachricht eine vorher vereinbarte Wartezeit eingehalten. So ist es möglich bis zu 4095 Bytes zu versenden. Die segmentierten Nachrichten werden mit einer zyklischen, aufsteigenden Seriennummer im Bereich von 0 bis 15 versehen. Im Fehlerfall (z.B. wenn der Empfänger eine falsche Seriennummer bemerkt) meldet die Transportschicht dies lediglich der eigenen Anwendung, nicht aber dem Kommunikationspartner. Daher sind weitere Protokolle nötig, die z.B. ein erneutes Senden veranlassen.

8.2.8 AutoLab-Entwicklungsumgebung

Um Programme für den Zentralcontroller zu erstellen, wurde am Lehrstuhl eine grafische Entwicklungsumgebung geschaffen. Sie bietet Schülern die Möglichkeit, ohne die Hürde, eine textuelle Programmiersprache erler-

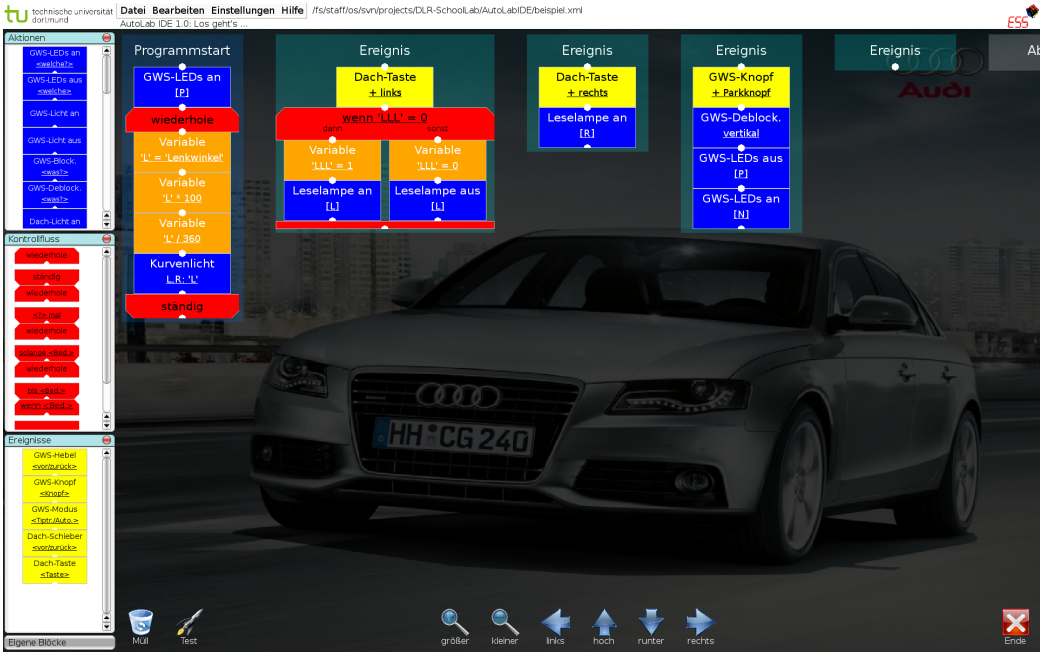

Abbildung 18: Screenshot der Autolab-Entwicklungsumgebung

nen zu müssen, das Verhalten des Fahrzeugaufbaus zu definieren und anschließend in Form eines generierten AVM-Byte-Code in den Zentralcontroller zu übertragen. Die grafische Programmiersprache bietet die wesentlichen Konstrukte der imperativen Programmierung, globale Variablen, Zuweisung, einfache arithmetische Ausdrücke, bedingte Verzweigung, und Schleifen, jedoch nicht die Möglichkeit von Subprozeduraufrufen. Diese „Kontrollfluss“-Elemente werden in Form von roten und orangen Bausteinen angeboten. Für die Domäne „Fahrzeugnetz“ gibt es die Möglichkeit Aktionen anzustoßen (z. B. „Schalte linken Blinker ein“). Diese werden in Form von blauen Bausteinen bereitgestellt und führen zum Versand einer CAN-Nachricht in das Fahrzeugnetz. Das Programmierparadigma ist ereignisgetrieben. So besteht ein Autolab-Programm im Wesentlichen aus Ereignisbehandlern (grün am oberen Bildschirmrand), an die eine Kette aus den bereits vorgestellten Bausteinen gehängt werden kann, von denen jeder jeweils bei Eintreffen einer vom Programmierer definierten CAN-Nachricht abgearbeitet wird. Zudem gibt es ein „Programmstart“-Ereignisbehandler, der beim Einschalten des Systems ausgeführt wird.

8.2.9 Autorennsimulation TORCS

Die vorhandenen Autohardwarekomponenten werden durch das Rennspiel TORCS ergänzt. Es handelt sich dabei um ein quelloffenes Spiel, das für die AutoLab-Umgebung angepasst wurde. TORCS ist für diesen Zweck mit einer Schnittstelle zum High-Speed-CAN-Bus ausgestattet und kann so Nachrichten von der AutoLab Virtual Maschine entgegen nehmen und auch Statusmitteilungen in das Fahrzeugnetz einspeisen (z. B. aktuelle Fahrgeschwindigkeit, usw.).

Abbildung 19: Screenshot der Autorennsimulation „TORCS“

8.3 Beschreibung des Zielsystems

Die Aufgaben des bestehenden Systems sollten nun durch ein neues hochintegriertes Network-on-Chip abgebildet werden. Da uns Hardwarebeschreibungen der ursprünglich eingesetzten Mikrocontroller nicht zur Verfügung standen, legten wir zunächst für jede der bisherigen Komponenten einen von

Bezeichnung	Hardware	Aufgaben
Scheinwerfersteuerung	Tricore 1796 Board / 150Mhz	Ansteuerung der Leistungstreiber für die Scheinwerfer, Umrechnung des Kurvenlichtwinkels
Pedalcontroller	Tricore 1796 Board / 150Mhz	Analog-Digitalwandlung des Stellwertes für Fahr- und Bremspedal
CAN-Bus Gateway	Freescale 9S12 Board / 2x CAN-Transceiver (Hi/Lo)	Routing von Nachrichten zwischen den beiden CAN-Bussen
Zentralcontroller	Tricore 1796 Board / 150Mhz	Autolab Virtual Machine

Tabelle 5: Aufgabenverteilung im herkömmlichen System

Bezeichnung	Hardware	Aufgaben
Scheinwerfersteuerung	HC11/HC08 / je 4KB RAM / 8KB ROM / je 1x CAN-Controller	Ansteuerung der Leistungstreiber für die Scheinwerfer, Umrechnung des Kurvenlichtwinkels
Pedalcontroller	AVR / 2KB RAM / 2KB ROM / 1x CAN-Controller / 1x SPI	Analog-Digitalwandlung des Stellwertes für Fahr- und Bremspedal
CAN-Bus-Gateway	T51 / 512Byte RAM / 8KB ROM / 2x CAN-Controller	Routing von Nachrichten zwischen den beiden CAN-Bussen
Zentralcontroller	AVR / 32KB RAM / 16KB ROM / 1x CAN-Controller	Autolab Virtual Machine

Tabelle 6: Aufgabenverteilung im neuen, FPGA-basierten System

uns evaluierten Controllertyp fest, der die Aufgaben im Zielsystem übernehmen soll. Die Aufgabe der Scheinwerfersteuerung übernehmen im neuen System zwei Softcores der Typen HC08 und HC11, von denen jeweils einer den linken bzw. rechten Scheinwerfer ansteuert. Die Standardkonfiguration mit 4KB RAM bzw. 8KB ROM reicht für diese einfache Aufgabe vollkommen aus. Der ehemalige, ebenfalls Tricore-basierte, Pedalcontroller wird von einem auf die Minimalkonfiguration abgespeckten AVR-Core abgelöst. Hier waren 2KB-RAM sowie 2KB-ROM ausreichend und an Peripherie nur ein CAN-Controller sowie ein SPI-Controller zur Ansteuerung des externen ADC-Wandlers nötig. Die Rolle des CAN-Bus-Gateways übernimmt im neuen System ein um zwei CAN-Controller angereicherter T51-Core. Für den Zentralcontroller wurde der AVR-Core über die am Markt verfügbaren Ausstattungsvarianten hinaus mit 32KB RAM sowie 16KB ROM ausgestattet. An Peripherie ist auch hier nur ein CAN-Controller notwendig. Über die nötige Peripherie hinaus wurde allerdings in allen Cores ein UART zu Debug-Zwecken verbaut. Diese werden über einen einfachen Multiplexer je nach DIP-Schalterstellung am FPGA-Board auf die 9-polige SUB-D-Buchse desselben gelegt. Im Testsystem laufen alle Controller mit einer Taktfrequenz von 25Mhz. Theoretisch ist es jedoch möglich, diese mit unterschiedlichen, auf den jeweiligen Controller angepassten Frequenzen zu betreiben. Die Controller wurden mit Bedacht unterschiedlich gewählt, um den heterogenen Charakter eines Automobilnetzwerks nachzubilden. Aus diesem Grund wurden auch die Aufgaben der Scheinwerfersteuerung auf die beiden Controller HC11 und HC08 aufgeteilt. Die einzelnen Komponenten wurden zunächst in Kleingruppen bis zur Funktionsreife entwickelt und getestet. Erst in einem zweiten Schritt wurden diese in ein gemeinsames VHDL-Design integriert. Die Kommunikation erfolgt über einen im Design integrierten on-Chip-CAN-Bus. Die Anbindung eines geteilten externen Speichers war aufgrund der recht geringen Speicheranforderungen nicht erforderlich. Es standen auf dem von uns zur Integration verwendeten FPGA ausreichend Block-RAM-Bausteine zur Verfügung.

8.4 Beschreibung der Zielhardware

Leider sprengte das integrierte Network-on-Chip die Kapazität des eigentlich von uns verwendeten Xilinx Spartan 3-500E FPGAs um ein Vielfaches, so dass eine Synthese nicht möglich war. Es war also erforderlich ein entsprechend größeres FPGA-Modell zu verwenden. Die Wahl fiel hier auf ein am Lehrstuhl vorhandenes Entwicklungsboard mit einem Xilinx Virtex2p FPGA, welches eine etwa 8-fache Kapazität unseres Spartan Entwicklungsboard aufweist. Neben einem höheren Bedarf an Logikzellen war für das Network-

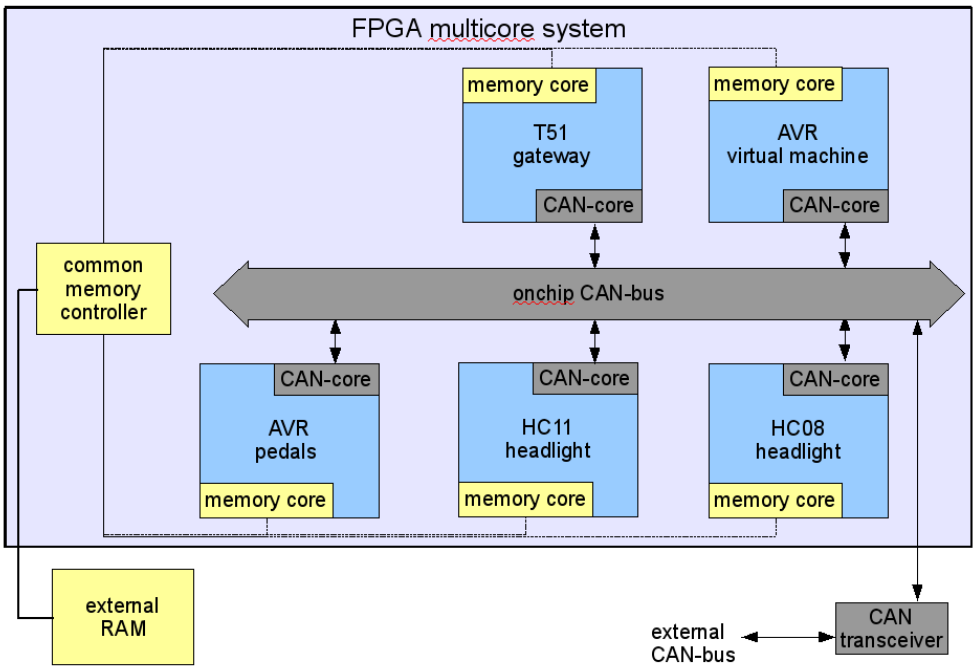

Abbildung 20: Schematische Darstellung des entwickelten Network-on-Chip

on-Chip auch eine gößere Anzahl an integrierten Block-RAMs erforderlich. Zusätzlich kann das Virtex-Entwicklungsboard durch ein DDR-Modul mit bis zu 2GB Speicher erweitert werden. Der auf dem Spartan-Board vorhandene Analog-Digital-Wandler fehlt leider auf dem Virtex-Board, so dass ein externer AD-Wandler der Firma Digilent angeschlossen wurde. Ein weiterer Unterschied besteht in dem auf dem Board vorhandenen Quartz-Oszillator, der einen 100MHz-Takt für das FPGA bereitstellt.

9 Evaluation der Endergebnisse

Das im Rahmen der PG entwickelte Network-on-Chip soll im folgendem Kapitel einer Bewertung unterzogen werden. Dabei spielen sowohl funktionale als auch nicht-funktionale Anforderungen eine Rolle.

9.1 Isolation

Eine wichtige Anforderung an das Network-on-Chip ist die strikte Isolation der einzelnen Systeme untereinander. Besondere Aufmerksamkeit verlangt dabei die gemeinsame Nutzung des DDR-RAM.

Wie im Kapitel 7.2 „Memory-Multiplexing“ bereits erwähnt, ist dies durch die Trennung des Adressbereiches des DDR-RAM in verschiedene Sektionen sichergestellt, die ein System on Chip exklusiv nutzen kann. Mit Hilfe eines Bubblesort-Benchmarks haben wir die korrekte Nutzung des Speichers verifiziert. Getestet wurden die Korrektheit und die Laufzeit des Sortierens von Daten im DDR.

9.1.1 Benchmark

Unser Benchmark schreibt ein 256 Byte langes Array mit absteigend sortierten Werten in den DDR und stellt sie auf dem Terminal dar. Dies entspricht der wostcase-Eingabe für Bubblesort. Anschließend wird das Array mit dem bekannten Bubblesort-Algorithmus aufsteigend sortiert. Der Algorithmus hat in diesem Fall eine quadratische Laufzeit über der Eingabemenge. Da auch Hilfsvariablen des Algorithmus im DDR liegen, ist beim Sortieren des Arrays mit einer Anzahl von Lese- und Schreibzugriffen im 6-stelligen Bereich zu rechnen. Zuletzt wird das sortierte Array wieder auf dem Terminal ausgegeben um die Werte zu überprüfen. Gemessen wird dabei nur die Zeit für das Sortieren.

9.1.2 Ergebnisse

Verglichen wurde der Zugriff eines einzelnen HC08-SoCs auf den DDR mit dem parallelen Zugriff zweier beliebiger HC08. Wir führten jeweils 10 Messungen durch. Für beide Fälle konnte die Korrektheit nachgewiesen werden. Das Bild „Laufzeit des Benchmarks im Vergleich“ zeigt die Laufzeit des Bubblesort-Alogorithmus bei den einzelnen System-on-Chip. Die Werte entsprechen dem Mittelwert über alle 10 Messungen. Die Laufzeit wurde mikrosekundengenau mit Hilfe des Logic-Analysers “HP Agilent 16500 A“ im Labor getestet. Man sieht, dass der parallele Zugriff zweier HC08 die Laufzeit des Benchmarks im Vergleich zu einem Core leicht erhöht. Das „naive“

Scheduling nach Round-Robin hat den Effekt, dass einer der beiden Cores in seiner Ausführung verzögert wird und eine längere Lauftzeit hat.

9.2 Größe und Geschwindigkeit

Baut man mehrere Systems-on-Chip zu einem Network-on-Chip zusammen, so stellt sich die Frage, wie groß das gesamte Design nach der Synthese sein wird und wie schnell man das Netzwerk takten kann. Um diese Fragen zu beantworten wurden einige Konfigurationen synthetisiert und ihre Ausgaben verglichen. Die einzelnen Netzwerke bestanden aus einer Anzahl von AVR-Cores, die auf ein Minimum konfiguriert wurden. D. h., dass sie lediglich einen CAN-Controller als Peripherie besitzen. Durch diese Einschränkung gelingt es, bis zu zehn Cores zu einem Netzwerk zu bündeln und gleichzeitig die Kapazitäten einzuhalten, die das Virtex 2 Pro zur Verfügung stellt.

Der Synthesevorgang wurde mit den folgenden Parametern durchgeführt:

- Virtex 2 Pro, XC2VP30, FF896
- Optimierung nach Größe
- Speedgrade -7

Aus Abbildung 22 kann man schließen, dass der LUTs-Verbrauch mit zunehmender Anzahl an Softcores linear steigt. Anscheinend sind keine Übergreifenden Optimierungen möglich. Die Geschwindigkeit des Netzwerks ist stets abhängig von der längsten Signallaufzeit. Da die einzelnen Softcores in diesem Test identisch sind, ist es nicht verwunderlich, dass die maximal erreichbaren Geschwindigkeiten sich stark ähneln. Basierend auf diesen Erkenntnissen kann nun mit Hilfe der Angaben einzelner Komponenten ein System „zusammengesteckt“ werden, von dem man bereits vor der Synthese die ungefähren Größenordnungen kennt.

9.3 Nicht-funktionale Anforderungen für den Einsatz von FPGAs in Automobilen

Im Rahmen der in der Projektgruppe durchgeführten Exkursion haben wir u. a. die Abteilung für Technische Entwicklung der Audi AG in Ingolstadt besucht. In Gesprächen mit einigen Mitarbeitern wurden wir schnell darauf aufmerksam, dass es neben den funktionalen auch eine Reihe von nicht-funktionalen Anforderungen an Elektronik im automotiven Einsatz gibt, die auch von einer FPGA-Lösung erfüllt werden müssen. So herrschen in einem Automobil schwierige Umgebungsbedingungen, die sich nicht negativ auf die

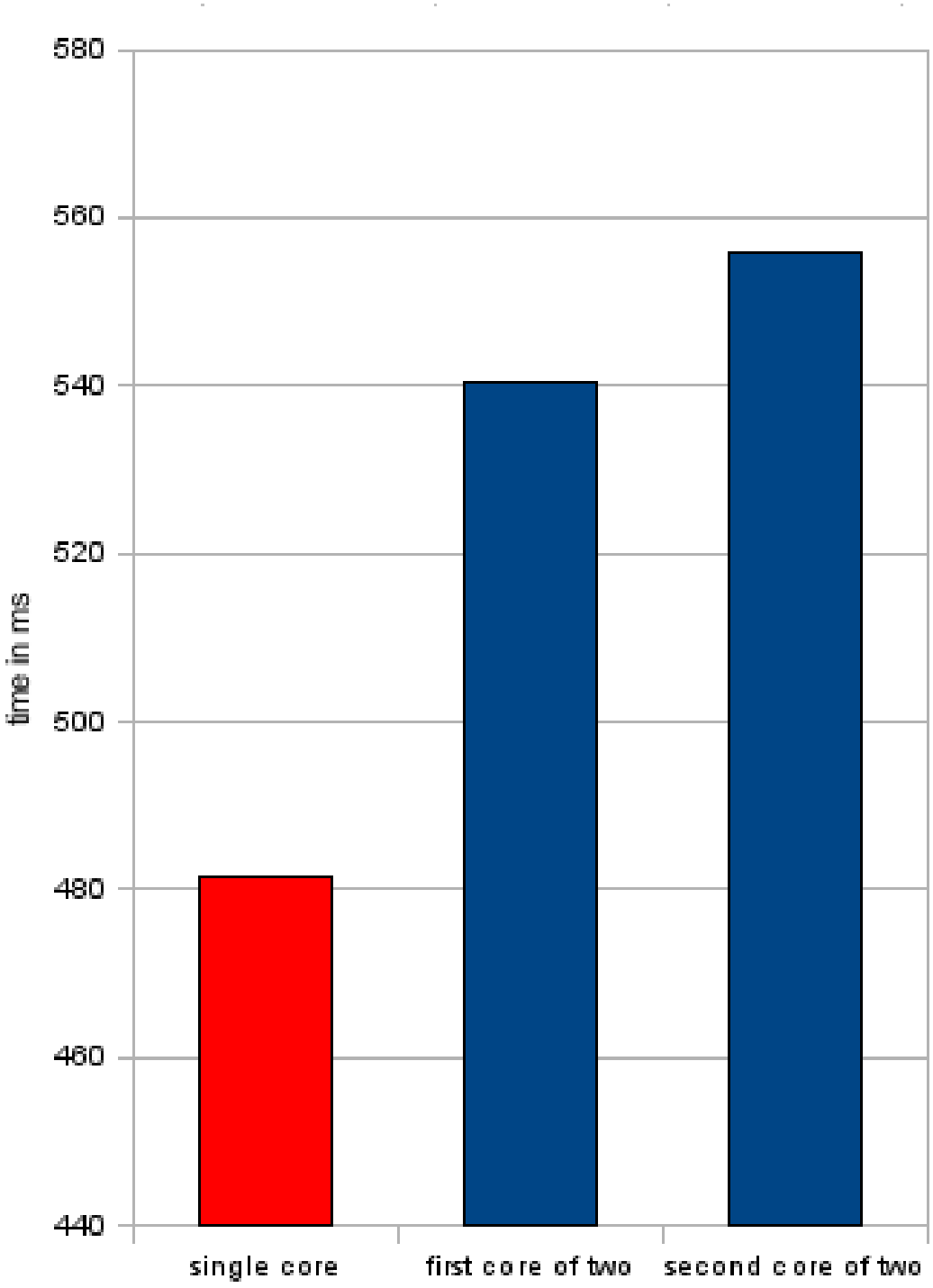

Abbildung 21: Laufzeit des Benchmarks im Vergleich

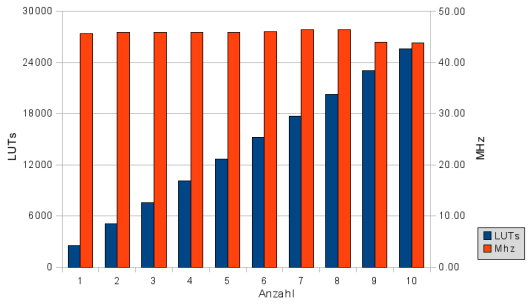

Abbildung 22: Größe und Geschwindigkeit von Networks-on-Chip

zum Einsatz kommende Elektronik auswirken dürfen. Des weiteren wurden einige konkrete Probleme genannt, die im Zusammenhang mit FPGAs auftreten könnten.

9.3.1 Temperaturbereiche

Elektronik im Automobil ist viel größeren Temperaturschwankungen ausgesetzt, als zum Beispiel in Haushalts- oder Unterhaltungselektronik. Automobile werden weltweit eingesetzt, und so muss sowohl sehr kalten als auch sehr hohen Temperaturen standgehalten werden. Auch können, z. B. in der Nähe des Motors, sehr hohe Temperaturen auftreten. Hier kommt als zusätzliche Belastung noch die vergleichsweise schnelle Temperaturänderung hinzu. So kann, beispielsweise an einem Wintertag, die Temperatur im Motorraum innerhalb kürzester Zeit von Minusgraden auf sehr hohe Temperaturen ansteigen. Diese Temperaturschwankungen haben sowohl elektrische (Halbleitereigenschaft) als auch mechanische (thermische Dehnung bzw. Stauchung) Auswirkungen. Deswegen haben verschiedene Standardisierungsgremien (wie ISO oder das „Automotive Electronics Council“) entsprechende Anforderungsprofile erarbeitet, die Automobilelektronik erfüllen muss. Das „Automotive Electronics Council“ (AEC) klassifiziert in ihrer Spezifikation AEC-Q100 [3] z. B. folgende Temperatureinstufungen, welche die eingesetzte Elektronik je nach Einsatzort und -zweck erfüllen muss:

- Stufe 0: $-40°C$ bis $+150°C$ Umgebungstemperatur
- Stufe 1: $-40°C$ bis $+125°C$ Umgebungstemperatur
- Stufe 2: $-40°C$ bis $+105°C$ Umgebungstemperatur
- Stufe 3: $-40°C$ bis $+85°C$ Umgebungstemperatur
- Stufe 4: $0°C$ bis $+70°C$ Umgebungstemperatur

Nach unseren Recherchen bieten die gängigen FPGA-Hersteller (z. B. Xilinx, Altera, Actel) spezielle Serien an, die zumindest die Bedingungen der Anforderungsstufe 1 erfüllen können.

9.3.2 Deterministischer Systemstart

Des weiteren wurde bei Audi das Problem vom Hochfahren des Systems angesprochen. Hier bestanden Bedenken, dass ein FPGA-basiertes System undefiniertes Verhalten in der Phase des Systemstarts zeigen könnte. Ein typisches Problem von SRAM-FPGAs ist, dass nach Anlegen der Versorgungsspannung erst einmal keine definierte Konfiguration auf dem FPGA vorliegt und diese erst aus einem externen nicht-flüchtigen Speicher geladen werden muss. In der Zwischenzeit liegen auf den Außen-Pins des Chips nicht immer voraussagbare Pegel an, was gerade bei sicherheitskritischen Komponenten ein Problem darstellen kann. Einen Ausweg aus dieser Situation können nicht-flüchtige FPGAs bieten. Diese basieren entweder auf Flash- oder aber Antifuse-Technology. Hier wird die Konfiguration statt in SRAM-Zellen in Flash-Speicher gehalten bzw. irreversibel eingebrannt, was dazu führt, dass der FPGA direkt nach dem Anlegen der Versorgungsspannung deterministisches Verhalten zeigt. Ein Beispiel für Flash-basierte FPGAs ist die ProASIC3-Serie des Herstellers Actel. Nicht zu verwechseln sind echte Flash-basierte FPGAs mit hybriden Modellen (z. B. Xilinx Spartan-3AN). Letztere integrieren lediglich einen SRAM-basierten FPGA und SPI-Flash in einem Chip. Auch in diesem Fall muss nach dem Anstieg der Versorgungsspannung das SRAM erst mit den Werten aus dem Flash-Speicher initialisiert werden.

9.3.3 Analoge Komponenten

Der von uns untersuchte Ansatz der Konsolidierung von automotiven Mikrocontrollern auf FPGAs muss mit seinen Vorbildern konkurrieren können. Die echten Mikrocontroller verfügen oft, neben dem eigentlichen CPU-Kern, über relativ umfangreiche On-Chip-Peripherie, welche die Interaktion mit analogen Schaltkreisen ermöglicht. So sind selbst bei kleinen AVR-Controllern AD-Wandler bereits integriert. Möchte man mit einem klassischen FPGA einen solchen Controller emulieren, wäre man auf zusätzliche externe Hardware in Form dedizierter AD-Wandler angewiesen. Allerdings gibt es auch hier eine Lösung in Form von sog. Mixed-Signal FPGAs, wie die Fusion-Serie des bereits erwähnten Herstellers Actel. Diese verfügen auch über analoge Komponenten, wie analoge Eingänge nebst zugehörigen Multiplexern, AD-Wandlern, MOSFET-Treibern usw.

Der Analogteil ist hier in den sog. Analog-Quads hierarchisch gegliedert, die jeweils gleich aufgebaut sind und die gleichen Primitive enthalten. Aus diesen können über Makros verschiedene Funktionen konfiguriert werden. So können die Analog-Eingänge bei diesem Modell auch als 12V-feste Digital-Eingänge genutzt werden. Andere Makros existieren zum zeitlich gemultiplexten Sampling der Eingänge oder der Nutzung eines Analog-Quads als Spannungsregler (wenn auch mit externem Transistor).

9.3.4 Neuprogrammierung im Betrieb

Zur Herstellung einer Wartbarkeit des Automobils als eingebettetes System wird die Möglichkeit von Softwareupdates bestimmter Steuergeräte benötigt. Da diese häufig an nicht ohne großen Aufwand erreichbaren Stellen ihren Dienst verrichten, ist es notwendig, dass die Versorgung mit neuer Software über eine gemeinsame, möglichst zentrale Schnittstelle erfolgen kann. Die Kraftfahrzeughersteller stellen in aktuellen Fahrzeugen diese Funktionalität über das Bordnetz zur Verfügung. Im Bereich der Mikrocontroller ist es üblich, dass diese sich, z. B. über einen partitionierten Flash-Bereich wie beim Atmel AVR, selbst rekonfigurieren können. In diesem Fall läuft in einem separaten Bereich des Flash-Speichers ein Bootloader, der die mittels spezieller Transportprotokolle (8.2.7) in CAN-Nachrichten verpackten Softwareupdates vom Bordnetz entgegen nimmt und in den anderen Bereich schreibt. So kann ohne größeren Aufwand und Kosten die Update-Fähigkeit hergestellt werden. Bei FPGAs ist diese Möglichkeit zumindest zur Zeit noch nicht ohne weiteres gegeben. Vielfältige, teils experimentelle, Ansätze zur partiellen Rekonfiguration (siehe z. B. [67]) von FPGAs könnten diese Situation in Zukunft jedoch verändern.

10 Fazit

Im Laufe des Wintersemester 2008/2009 wurden verschiedene Systems-on-Chip auf FPGAs integriert, teilweise um Peripherie erweitert und an unsere Bedürfnisse angepasst. Dazu wurden zunächst in VHDL verfügbare Open-Source-Prozessormodelle auf ihre Tauglichkeit für unsere Zwecke überprüft (siehe Kapitel 3). Bezüglich der Peripherie wurde z. B. ein eigener CAN-Controller entwickelt, der das CAN-Protokoll vollständig unterstützt, was uns die Kommunikation via CAN-Bus mit Bauteilen aus dem Automotive-Bereich ermöglichte (siehe Kapitel 5.2.4). Desweiteren wurde der HC08-Core um einen kompletten Peripherieblock erweitert. Für jedes der implementierten Systems-on-Chip wurden im weiteren Verlauf Beispielanwendungen entwickelt (siehe Kapitel 6).

Im darauffolgenden Sommersemester 2009 war damit die Grundlage geschaffen, um die bis Dato auf jeweils einem eigenen FPGA laufenden SoCs, auf nur einem FPGA als Netwok-on-Chip zu integrieren und die Demonstratoren mit nur noch einem FPGA-Board und zu einem Testaufbau zusammengefasst, zu betreiben.

Hier wurden die Kommunikation der Systeme untereinander, die unter Umständen beschränkte Speicherkapazität des Zielsystems sowie das schwierige Debugging eines solchen Network-On-Chip als Probleme identifiziert.

Bezüglich der Kommunikation wurde das Konzept des on-Chip-CAN-Bus entwickelt, der die Kommunikation der ansonsten isolierten Mikrocontroller untereinander auf einem FPGA ermöglicht (siehe Kapitel 7.1).

Für das eventuell auftretende Problem, dass auf dem Ziel-FPGA zu wenig RAM für die zu integrierenden Systeme vorhanden ist, wurde mit der Entwicklung eines Memory-Multiplexers eine mögliche Lösung aufgezeigt. Mit diesem ist es bei Bedarf möglich, externen Speicher mit hoher Kapazität (hier: DDR-RAM) für die enthaltenen SoCs geteilt nutzbar zu machen (siehe Kapitel 7.2).

Um ein umfangreiches Debugging der integrierten System zu ermöglichen, sollte ein in Hardware implementierter GDB-Stub entwickelt werden. Die Arbeiten an diesem wurden bis zum Ende der Projektgruppe jedoch leider nicht erfolgreich beendet (siehe Kapitel 7.3).

Es konnte, mit Hilfe eines Demonstrationsaufbaus (siehe Kapitel 8), gezeigt werden, dass die Integration mehrerer Automobilsteuergeräte auf einem FPGA technisch durchaus machbar ist. Weiterhin konnte gezeigt werden, dass die für den Einsatz im Automobilbereich unumgängliche Isolation der einzelnen Komponenten mit der Network-on-Chip-Lösung möglich ist (siehe Kapitel 9.1) und evtl. eine vielversprechende Alternative zum aktuell verfolgten Ansatz der Softwareisolation (z. B. AUTOSAR) darstellt. Der

Ansatz skaliert gut (siehe Kapitel 9.2). Bis zur Produktionsreife ist allerdings noch viel Forschungsarbeit zu leisten. Im Besonderen konnten die nichtfunktionalen Anforderungen an eingebettete Systeme im Automobilbereich von uns nur oberflächlich behandelt werden (siehe Kapitel 9.3). Zukünftige Arbeiten könnten sich intensiver mit diesen beschäftigen. Hier sind vor allem Aspekte wie die Kostensituation (sowohl Entwicklungskosten als auch variable Kosten), die Zuverlässigkeit, Wartbarkeit und Lebensdauer zu untersuchen.

A Anhang

A.1 Änderungen an den Softcores

Die von uns verwendeten Softcores sind alle als Open-Source unter OpenCores.net [20] verfügbar. Abgesehen vom Intel 8051 waren bei allen Cores Änderungen vorzunehmen, da diese zum Teil erhebliche Mängel aufwiesen bzw. an unsere Bedürfnisse angepasst werden mussten.

A.1.1 68HC08

Peripherie Die Quellen für den 68HC08, die auf Opencores.org zur Verfügung stehen, umfassen nur der Prozessor selbst und keinerlei Peripherie. Der Core ist „übersichtlich“ in einer VHDL-Datei zusammengefasst und verfügt über eine Multiplizier- und Dividier-Einheit. Es mussten also noch eine Bus-Architektur sowie ROM und RAM integriert werden.

Frequenzteiler Das Spartan-Board wird standardmäßig mit 50 MHz betrieben. Nach der Synthese des Cores lag die maximale Frequenz unseres Designs bei etwa 25 MHz. Wir entschieden uns zunächst den Takt mittels eines Frequenzteilers auf 8.33 MHz zu begrenzen. Dies entspricht der Original-Taktfrequenz des 68HC08. Mit dieser Frequenz ließ sich jedoch die Baudrate des CAN-Bus (500000) nicht genau genug erzeugen, daher wechselten wir auf 10MHz.

Fehlerhafte Instruktion Die Instruktion `MOV ,X+,opr8a` ist im HC08-Core fehlerhaft implementiert. Die Inkrementierung des Index-Registers wurde vergessen. Nach Behebung des Fehlers lief der Core einwandfrei.

Fallende Flanke Der Speicher muss beim HC08 mit der fallenden Flanke getriggert werden. Dies muss bei der Anbindung verschiedener Module beachtet werden.

A.1.2 Atmel AVR

Speicher Das erste Problem mit dem Softcore war der vorliegende Daten- und Programmspeicher. Zwar wurde der Speicher nach dem Synthesevorgang auf die vorhandenen Block-RAMs des FPGA abgebildet, jedoch auf mehr als tatsächlich vorhanden waren. Mit Hilfe des XST-Guides wurde eine simple Implementierung des Datenspeichers vorgenommen. Der Programmspeicher

wird abhängig vom Programm durch eine feste Tabelle repräsentiert. Hierdurch wird kein Speicher verschwendet!

Frequenzteiler Da die Mikrocontroller in der Realität mit nur wenigen Megahertz takten, das FPGA aber einen festen Takt von 50MHz vorgibt, wurde ein Frequenzteiler eingebaut. Dieses Bauteil kann den Takt flexibel herabsenken. Dadurch konnten schließlich Tests mit UART oder einem LCD erfolgreich durchgeführt werden. Im Prinzip ist der Frequenzteiler nur ein getaktetes Zählregister, das bei einem Überlauf des höchsten Bits einen Takt generiert.

JTAG Die JTAG-Schnittstelle in der uns vorliegenden, freien Version des AVR_Core unterstützt kein On-Chip-Debugging. Wegen zusätzlicher Probleme mit dem nachträglich eingebautem Frequenzteiler wurde entschieden, diese Komponente vorerst zu entfernen. Weil JTAG in den Programmspeicher schreiben kann, mussten nach dem Entfernen noch einige Signale umgelenkt und direkt an den Programmspeicher angeschlossen werden.

A.1.3 Freescale 9S12

Änderungen am HC11 Der original HC11-Softcore wurde ursprünglich für Altera-FPGAs entwickelt, so dass Teile des Softcores erst nach VHDL portiert werden mussten. Die ursprünglich angedachte Erweiterung des HC11 zu einer Nachbildung eines 9S12 musste leider aufgeben werden. Aufgrund des Aufbaus des HC11-Softcores wäre die Implementierung sämtlicher neuen Features des 9S12 etwa so aufwändig wie ein neu entwickelter Nachbau des 9S12. So wurde beschlossen den HC11 weiter zu verwenden.

Fehler im Mikrocode Nach einigen Tests mit selbst kompilierten Programmen sind zwei Fehler im Mikrocode aufgefallen. Die 16-Bit-Operationen SUBD und ADDD wurden im Mikrocode als 8-Bit-Operationen implementiert. Da kein Mikrocodeassembler zur Verfügung stand, musste der Mikrocode-Speicher angepasst werden. Dazu war es erforderlich, anhand der Simulationsdaten die entsprechende Speicherzeile zu ermitteln und die dort abgelegten Befehle zu korrigieren.

A.1.4 Intel 8051

Änderungen am t51-Core Da der Core voll funktionsfähig und mit seiner gesamten Peripherie vollständig ist, war es nicht notwendig, Änderungen vorzunehmen.

A.2 Projektgruppenfahrt nach Erlangen

Die Anreise nach Erlangen am Dienstag, den 03.03.2009, erfolgte in zwei angemieteten Minibussen. Direkt nach der Ankunft am frühen Nachmittag wurde die Firma Elektrobit in Tennenlohe besucht, einer der größten Software-Zulieferer der Automobilindustrie, der u. a. der PG kostenlos Systemsoftware zur Verfügung stellt. Die Studenten stellten die Arbeiten der PG in einem Vortrag vor, dem eine angeregte Diskussion mit Entwicklern der Firma folgte. Danach wurden uns spannende Einblicke in die Softwareentwicklung und deren begleitende Prozesse bei Elektrobit gegeben.

Nach Bezug der Jugendherberge „Frankenhof“ in Erlangen am Abend war noch Gelegenheit, die Erlanger Gastronomie zu testen und den Tag ausklingen zu lassen. Am Mittwoch wurde die Messe Embedded World auf dem Nürnberger Messegelände besucht. Dort suchten die Exkursionsteilnehmer unter anderem das Gespräch mit Vertretern der Firmen, die das Vorhaben der Projektgruppe bereits über das Wintersemester hinweg mit Hardware und Know-How unterstützt haben. Darüber hinaus konnten auch Kontakte zu weiteren Firmen geknüpft werden, die die PG möglicherweise in Zukunft unterstützen werden, und teilweise bereit waren, kostenlose Hardware-Samples zur Verfügung zu stellen.

Der Mittwochabend wurde nach einer Kurzbesichtigung der Nürnberger Innenstadt und der Burg im „Bratwursthäusle“ (ein Traditionsrestaurant) abgeschlossen.

Am Donnerstag Morgen wurde die Firma Audi in Ingolstadt besucht. Zunächst demonstrierte eine Werksführung eindrucksvoll, wie moderne Industriefertigung von Kraftfahrzeugen abläuft. Am frühen Nachmittag gab dann der Besuch der Technische Entwicklung tiefere Einblicke in die Forschungsabteilung des Herstellers: Großangelegte Testaufbauten und Fahrzeug-Prototypen waren zu besichtigen, und Entwickler der Software-Abteilung berichteten über die Vorgehensweisen in Forschung und Serie. Abschließend präsentierte auch hier die PG ihre Ziele und die bisher erreichten Ergebnisse, was ebenso in einer Diskussionsrunde mündete.

Gegen 17:30 traten wir wieder die Heimreise nach Dortmund an, wo die Exkursion etwa um 22:30 endete.

Literatur

[1] 8-Bit Microcontroller of Dimo Pepelyashev. Website. Verfügbar unter URL http://www.opencores.org/projects.cgi/web/mcu8/overview.

[2] 8051 C-Compiler. Verfügbar unter URL http://www.keil.com/uvision.

[3] Automotive Electronics Council (AEC). Website. Verfügbar unter URL http://www.aecouncil.com/.

[4] AutoSAR. Website. Verfügbar unter URL http://www.autosar.org.

[5] AVR 8-Bit RISC CPU. Website. Verfügbar unter URL http://www.atmel.com/products/avr.

[6] AVR C-Compiler. Verfügbar unter URL http://gcc.gnu.org.

[7] Car on a Chip. Website. Verfügbar unter URL http://ess.cs.tu-dortmund.de/Teaching/PGs/coach/index.html.

[8] Constraints Guide. Website. Verfügbar unter URL http://www.xilinx.com/itp/xilinx10/books/manuals.pdf.

[9] Controller Area Network. Website. Verfügbar unter URL http://de.wikipedia.org/wiki/Controller_Area_Network.

[10] Controller Area Network (CAN in Automation). Website. Verfügbar unter URL http://www.can-cia.org.

[11] Development System Reference Guide. Website. Verfügbar unter URL http://www.xilinx.com/itp/xilinx10/books/manuals.pdf.

[12] Echtzeitsystem. Website. Verfügbar unter URL http://de.wikipedia.org/wiki/Echtzeitsystem.

[13] Field Programmable Gate Array. Website. Verfügbar unter URL http://de.wikipedia.org/wiki/Field_Programmable_Gate_Array.

[14] gEDA: GPL'd suite and toolkit of Electronic Design Automation. Website. Verfügbar unter URL http://www.geda.seul.org.

[15] Glitch. Website. Verfügbar unter URL http://de.wikipedia.org/wiki/Glitch_(Elektronik).

[16] GTKWave. Website. Verfügbar unter URL http://sourceforge.net/projects/gtkwave.

[17] HC11 C-Compiler. Verfügbar unter URL http://gcc.gnu.org.

[18] MC9S12GC Family Reference Manual. Freescale Semiconductor, Inc. Verfügbar unter URL http://www.freescale.com.

[19] Mikrocontroller - Hardware (forum). Website. Verfügbar unter URL http://www.mikrocontroller.net.

[20] Opencores - Opensource Project. Website. Verfügbar unter URL http://www.opencores.org.

[21] PG 522: AutoLab Endbericht. Embedded Systems Group, TU Dortmund / PG 522. Verfügbar unter URL http://ess.cs.tu-dortmund.de/Teaching/PGs/autolab/AutoLab-Endbericht.pdf.

[22] Plasma-Microcontroller auf OpenCores.org. Website. Verfügbar unter URL http://www.opencores.org/?do=project&who=plasma.

[23] Schaltnetz. Website. Verfügbar unter URL http://de.wikipedia.org/wiki/Kombinatorischer_Schaltkreis.

[24] Serial Peripheral Interface. Website. Verfügbar unter URL http://de.wikipedia.org/wiki/Serial_Peripheral_Interface.

[25] Serial signal. Website. Verfügbar unter URL http://upload.wikimedia.org/wikipedia/de/d/de/RS-232_timing.png.

[26] Spartan-3 Generation FPGA User Guide. Website. Verfügbar unter URL http://www.xilinx.com/support/documentation/user_guides/ug331.pdf.

[27] Spartan-3E FPGA. Website. Verfügbar unter URL http://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf.

[28] Spartan-3E FPGA Family: Complete Data Sheet. Website. Verfügbar unter URL http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf.

[29] Spartan-3E FPGA Starter Kit Board User Guide. Website. Verfügbar unter URL http://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf.

[30] Spartan-3E Starter Kit Board User Guide. Website. Verfügbar unter URL http://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf.

[31] Static Timing Analysis. Website. Verfügbar unter URL http://en.wikipedia.org/wiki/Static_timing_analysis.

[32] The Unofficial 8051 History. Website. Verfügbar unter URL http://www.efton.sk/t0t1/history8051.pdf.

[33] Value change dump. Website. Verfügbar unter URL http://en.wikipedia.org/wiki/Value_change_dump.

[34] Very High Speed Integrated Circuit Hardware Description Language. Website. Verfügbar unter URL http://de.wikipedia.org/wiki/Very_High_Speed_Integrated_Circuit_Hardware_Description_Language.

[35] VHDL Tutorial. Website. Verfügbar unter URL http://www.vhdl-online.de/tutorial/.

[36] XST User Guide. Website. Verfügbar unter URL http://www.xilinx.com/itp/xilinx10/books/manuals.pdf.

[37] W. W. Ahmed Jerraya. *Multiprocessors Systems on Chip*. Morgan Kaufmann, 2004.

[38] Atmel Corporation. *8-bit AVR Microcontroller with 32K/64K/128K Bytes of ISP Flash and CAN Controller*, 8 2008.

[39] Atmel Corporation et al. *AVR Libc*, 1999-2008. Verfügbar unter URL http://www.nongnu.org/avr-libc.

[40] H. T. E. Axel Jantsch. Networks on Chip. Kluwer Academic Publishers, Boston; 2003. Hardcover, pp. 303, plus VIII, ISBN 1-4020-7392-5. *Microelectronics Reliability*, 44(7):1203–1204, 2004.

[41] BOSCH. *CAN-Specification Version 2.0*, 1991/1997.

[42] S. Braun. Zeit- und ereignisgesteuerte echtzeitsysteme. Seminarvortrag für die Technische Universität Dortmund.

[43] D. Cuturela. pAVR. Website. Verfügbar unter URL http://www.opencores.org/projects.cgi/web/pavr/overview.

[44] M. Deyhadgari, M. Nickray, A. Afzali-kusha, and Z. Navabi. A new protokol stack model for network on chip. *Proceedings of the 2006 Emerging VLSI Technologies and Architectures (ISVLSI 06)*, 2006.

[45] K. Eisele. ARM Core. Website. Verfügbar unter URL http://www.opencores.org/cvsweb.shtml/core_arm/.

[46] M. Engel and O. Spinczyk. Aspects in Hardware – What Do They Look Like? In *Proceedings of the 7th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '08)*, Brussels, Belgium, Apr. 2008. ACM Press.

[47] M. Engel and O. Spinczyk, editors. *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems (IIES 2008)*, Glasgow, UK, Apr. 2008. ACM Press.

[48] M. Engel and O. Spinczyk. A Radical Approach to Network-on-Chip Operating Systems. In *Proceedings of the 42nd Hawai'i International Conference on System Sciences (HICSS '09)*, Waikoloa, Big Island, Hawaii, Jan. 2009. IEEE Computer Society Press.

[49] S. D. et al. Small Device C-Compiler. Verfügbar unter URL http://sdcc.sourceforge.net.

[50] K. Etschberger. *Controller Area Network: Grundlagen, Protokolle, Bausteine, Anwendungen*. HANSER, 2002.

[51] P. G. Fink. Computer Architecture - (Rechnerarchitektur). Website. Verfügbar unter URL http://www.irf.tu-dortmund.de/cms/en/IS/Teaching/SS08/ComputerArchitecture/index.html.

[52] P. Geipel. *Der IT-Projektmanager*. Addison-Wesley Verlag, 2003.

[53] J. Gray. Designing a Simple FPGA-Optimized RISC CPU and System-on-a-Chip. Website. Verfügbar unter URL http://www.fpgacpu.org/papers/soc-gr0040-paper.pdf.

[54] T. Heinz. Preserving temporal behaviour of legacy real-time software across static binary translation. In *Proceedings of the FirstWorkshop on Isolation and Integration in Embedded Systems*, Glasgow, UK, Apr. 2008. ACM Press.

[55] S. R. M. L. Henning Wolf. *eXtreme Programming*. dpunkt.verlag, 2005.

[56] A. Hilvarsson. AVRtinyX61core. Website. Verfügbar unter URL http://www.opencores.org/projects.cgi/web/avrtinyx61core/overview.

[57] E. Hwang. *Microprocessor Design with VHDL*. Brooks/Cole, 2004.

[58] IBM. *The CoreConnect Bus Architecture*, 1999.

[59] ISO. *ISO 15765-2:2004, Network layer services*, 2004.

[60] Joachim Quade. Echtzeitsysteme i+ii. Website. Verfügbar unter URL https://ezs.kr.hsniederrhein.de/lectures/ezs/html/index.html.

[61] H. Kopetz. Should responsive systems be event-triggered or time-triggered? Technical Report 8, Technische Universität Wien, 1991.

[62] H. Kopetz. Event-triggered versus time-triggered real-time systems. Technical Report 16, Technische Universität Wien, 1993.

[63] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 2001.

[64] H. Kopetz, R. Obermaisser, C. E. Salloum, and B. Huber. Automotive Software Development for a Multi-Core System-on-a-Chip. In *SEAS '07: Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

[65] R. Lepetenok. AVR Core. Website. Verfügbar unter URL http://www.opencores.org/projects.cgi/web/avr_core/overview.

[66] P. Marwedel. *Eingebettete Systeme*. Springer-Verlag, 2007.

[67] D. Mesquita, F. Moraes, J. Palma, L. Moller, and N. Calazans. Remote and partial reconfiguration of fpgas: tools and trends. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 8 pp.–, April 2003.

[68] M. W. Muhammad Ali and M. Zwicknagl. Networks on chips: Scalable interconnects for future systems on chips. *IEEE*, 2008.

[69] OpenCores. *WISHBONE SoC Architecture Specification, Revision B.3*, 2002.

[70] R. Pichler. *Scrum. Agiles Projektmanagement erfolgreich einsetzen.* dpunkt.verlag, 2008.

[71] S. Shen. nnARM Core. Website. Verfügbar unter URL http://www.opencores.org/projects.cgi/web/nnARM/overview.

[72] J. Sincero, O. Spinczyk, and W. Schröder-Preikschat. On the Configuration of Non-Functional Properties in Software Product Lines. In *Proceedings of the 11th Software Product Line Conference, Doctoral Symposium (SPLC '07)*, pages 167–173, 2007.

[73] M. Sisto. ARM Core. Website. Verfügbar unter URL http://www.opencores.org/projects.cgi/web/core_arm/overview.

[74] O. Spinczyk and H. Papajewski. Using Feature Models for Product Derivation. In *Proceedings of the 10th Software Product Line Conference (SPLC '06)*, page 225, Baltimore, Maryland, Aug. 2006. tutorial description.

[75] P. D.-I. E. B. und Mitarbeiter. Entwurfsautomatisierung in der Mikroelektronik. Website. Verfügbar unter URL http://edascript.ims.uni-hannover.de/260b_StatischeTimingAnalyse/start.html.

[76] D. Wallner. AX8 mcu. Website. Verfügbar unter URL http://www.opencores.org/projects.cgi/web/ax8/overview.

[77] J. Walter. *Mikrocomputertechnik mit der 8051-Controller Familie.* Springer, 2008.

[78] Xilinx. Using Digital Clock Managers (DCMs) in Spartan-3 FPGAs. Website. Verfügbar unter URL http://www.xilinx.com/support/documentation/application_notes/xapp462.pdf.