Universität Karlsruhe (TH)

Forschungsuniversität • gegründet 1825

Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Technische Informatik (ITEC)
Lehrstuhl für Eingebettete Systeme (CES)

Studienarbeit

„Implementierung einer I²C Schnittstelle für ein FPGA Prototyping Board und Anbindung eines Touch-Screen LCDs zur Steuerung von ASIPs“

Bearbeitet von: cand. Inform. Christian Krämer
Betreuer: Dipl.-Inform. Lars Bauer

Karlsruhe im Oktober 2006

Inhaltsverzeichnis

1 Einleitung

In diesem Kapitel wird die Motivation für diese Arbeit vorgestellt. Die Gesamtstruktur dieser Arbeit, und die gesetzten Ziele werden beschrieben.

1.1 Motivation

Die auf einem FPGA entwickelten Hardware-Elemente lassen sich nur aufwändig über das Beobachten von Signalleitungen debuggen. Software die auf einem auf dem FPGA laufenden Prozessor ausgeführt wird, kann ebenfalls nur Signalleitungen zur Kommunikation nach außen nutzen.

Ein an den FPGA angebundenes Display bietet sowohl in der Hardwareentwicklung, als auch beim Ausführen von Software auf einer CPU auf dem FPGA eine übersichtlichere und vor allem vielfältigere Möglichkeit zur Statusausgabe und Rückführung von Rechenergebnissen. Ein als Touchscreen ausgelegtes Display bietet darüber hinaus zusätzlich die Möglichkeit zur Steuerung, und verwirklicht so eine 2-Wege-Kommunikation.

Eine vorausschauende Wahl der Schnittstelle bietet darüber hinaus vielfältige Möglichkeiten für Erweiterungen, so dass auch weitere Geräte einfach an den FPGA angeschlossen werden können.

1.2 Strukturierung

Zuerst werden die Anforderungen an das Display genauer betrachtet, das gewählte Display vorgestellt und die vorhandenen Schnittstellen am Display eingehend beschrieben. Danach wird die gewählte I²C Schnittstelle und der I²C Standard im allgemeinen detailliert vorgestellt. Nachfolgend wird der gesamte Systemaufbau mit den einzelnen Komponenten beschrieben, sowie auf ausgewählte Details der Implementierung genauer eingegangen. Es wird die Anbindung des Diplays an eine CPU beschrieben, sowie die zu diesem Zweck erstellte Funktionsbibliothek vorgestellt. Schlussendlich werden Messergebnisse bezüglich des Platzverbrauches der Implementierung und zum Zeitbedarf der auf der CPU implementierten Funktionen vorgestellt.

1.3 Aufgabenstellung und Ziele

Für ein FPGA Prototyping Board soll die Anbindung eines LC-Displays entworfen werden. Das Display soll einmal als Ausgabemöglichkeit für auf dem FPGA laufende Hardwaredesigns dienen, als auch Möglichkeit zur Interaktion mit den Hardwaredesigns bieten. Zu diesem Zweck wurde beschlossen ein Touch Screen LCD zu verwenden. Ebenso steht eine möglichst einfache Programmierung des LC-Displays im Vordergrund. Da das Display unter Umständen von einem VHDL-Modul angesteuert wird (also nicht über einen programmierbaren Prozessor), soll hiermit eine in der Entwick-

lung komplexe und auf dem FPGA platzverschwenderische Programmierung verhindert werden. Die wäre z.B. der Fall, wenn das Display keinen integrierte Zeichensatz hätte, wie es bei einem „nackten“ Punktmatrixdisplay beispielsweise der Fall ist. Folglich wurde beschlossen ein Display mit möglichst viel „Eigenintelligenz“ zu verwenden.

Die gesamte Aufgabe besteht daraus ein geeignetes Display auszuwählen, die Anbindung an das FPGA Prototyping Board zu entwerfen, die Art der Implementierung (Hardware / Software - CoDesign) zu wählen, sowie auch letztendlich diese zu realisieren.

Gesamtziel ist eine möglichst vielfältig wieder verwendbare Implementierung zu erreichen, welche auch von einfachen und kleinen Projekten des Hardwareentwurfs ohne viel Aufwand verwendet werden kann. Damit liegt fest, dass eine Schnittstelle geschaffen werden muss, die eine unkomplizierte Verwendung direkt auf VHDL-Ebene erlaubt. Dabei ist ebenfalls zu beachten, dass eine möglichst allgemeine Implementierung der verwendeten standardisierten Schnittstelle von Nöten ist (d.h. im konkreten Fall keine explizite Zuschneidung auf das LCD), um diese auch anderweitig für Geräte mit der gleichen Schnittstelle verwenden zu können.

Ebenso soll es möglich sein, dass mehrere parallel ausgeführte Module auf dem FPGA die Schnittstelle im Wechsel nutzen können.

2 Auswahl des LC-Displays

In diesem Kapitel werden die Anforderungen an das gewählten LC-Displays und dessen Eigenschaften dargestellt.

2.1 Anforderungen an das Display

Das Display soll einen eingebauten Zeichensatz und nach Möglichkeit auch einen Grafikmodus besitzen, so dass eine aufwändige Programmierung eines Zeichensatzes, grafischer Funktionen und dergleichen entfällt. Es soll ausreichend Platz für mehrzeilige Textausgaben, Grafiken, Menüs und dergleichen bieten. Das Display soll als Touch-Screen ausgelegt sein, um Rückmeldungen an die angeschlossene Hardware zu erlauben. Die Schnittstelle des Displays soll standardisiert sein, so dass die geschaffene Schnittstelle auch zum Anschluss weiterer Geräte genutzt werden kann.

2.2 Eigenschaften EA-eDIP240

2.2.1 Eckdaten u.A. [eDIP240]

- Eingebaute Grafikfunktionen (Gerade, Punkt, Bereich, graphische UND/ODER/XOR Verknüpfung, Bargraph, Pull-Down-Menüs, ...)
- Text Konsolenmodus
- 8 eingebaute Fonts
- Schnittstellen: RS-232, RS-485, I²C-Bus und SPI-Bus
- 240 x 128 Pixel mit LED-Beleuchtung
- Versorgung +5V
- Pixelgenaue Positionierung bei allen Grafikfunktionen
- Eingebauter Menügenerator
- Makroprogrammierbar
- Analoges Touchpanel
- Frei definierbare Touchbereiche
- Integriertes Übertragungsprotokoll zur Fehlererkennung

2.3 Die Schnittstellen des EA-eDIP240

2.3.1 RS-232

Das Display bietet mit der RS-232 Schnittstelle einen weit verbreiteten Schnittstellentyp, die beispielsweise an jedem Rechner als sog. „serielle Schnittstelle“ zu finden ist. Für diese Schnittstelle sind zwei Leitungen erforderlich, wobei eine Leitung für das Versenden von Daten, und die andere für den Empfang zuständig ist. Die Schnittstelle am Display arbeitet mit 5V Pegeln, der RS-232 Standard sieht jedoch 12V Pegel vor, weshalb zum Anschluss des Displays zusätzlich ein Pegelwandler notwendig ist, um die Eingänge des Displays vor zu hohen Spannungspegeln zu

schützen [eDIP240]. Beschränkt man sich auf die Verwendung der 5V Pegel, wird dennoch eine Pegelwandlung zum Anschluss an das FPGA-Board wegen der dort erwarteten CMOS33 Pegel benötigt [Xilinx2].

Ein Nachteil ist, dass diese Schnittstelle nur eine Punkt-Zu-Punkt Verbindung erlaubt, somit keinen Anschluss von mehr als einem Geräte an der Schnittstelle ermöglicht.

Zusätzlich kann die Schnittstelle auch als RS-485 konfiguriert werden. Diese Schnittstellenart erlaubt bis zu 32 Geräte an einem Zwei-Draht-Bus. Jedoch wird auch für diese Anschlussvariante ein externer Umsetzter für die 5V Pegel benötigt.

2.3.2 SPI

Dieses von Motorola entwickelte Bus-System (***S**erial **P**eripheral **I**nterface*) erlaubt den Anschluss mehrerer Geräte. Dafür ist jeweils eine Datenleitung für Eingabe und Ausgabe, sowie eine Clock-Leitung notwendig. Zusätzlich wird für jedes an dem Bus angeschlossene Gerät eine so genannte Slave-Selector Leitung benötigt. Damit gibt es genau einen Master (im konkreten Fall wäre dies das FPGA Board), welches den Datentransfer initialisiert. Die Implementierung einer Multi-Master-Variante ist zwar möglich, erfordert aber einen weiteren Controller, der alle Master miteinander verbindet, sowie eine Controllerschnittstelle an allen angeschlossenen Mastern. Da die SPI Schnittstelle kein offizieller Standard von Motorola ist, ist diese Eigenschaft nicht per se gegeben.

Der Datentransfer selbst erfolgt folgendermaßen: nach Aktivierung des Slave Gerätes über die Slave-Selector Leitung werden die Datenbits nach Wahl zur negativen oder positiven Taktflanke auf der Clock-Leitung auf der jeweiligen E/A-Leitung geschrieben, bzw. von dort gelesen. Die Signalgeschwindigkeit kann im konkreten Falle des Displays bis zu 100kHz betragen, was einer Datenübertragung mit 12,5kB/sec abzüglich Overhead durch das LCD Übertragungsprotokoll (s.u.) entspricht, sie kann mit anderen Geräten jedoch auch deutlich höher liegen [SPI].

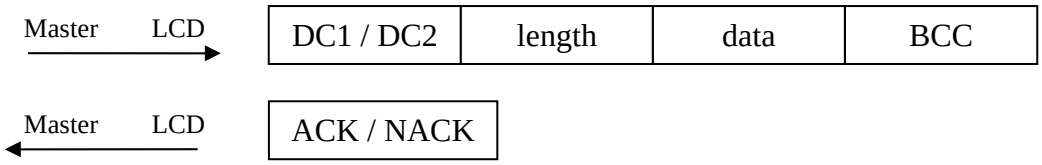
2.3.3 I²C

Dieses von Phillips entwickelte und standardisierte Bussystem ermöglicht ebenfalls den Anschluss mehrerer Geräte über nur zwei Drähte (Daten- und Clock-Leitung). Im direkten Vergleich zum Motorola SPI Bus entfällt damit die zusätzliche Slave-Selector Leitung für jedes angeschlossene Gerät – somit kann man hier also von einem echten Bus sprechen, an den weitere Geräte problemlos angeschlossen werden können. Zusätzlich dazu ist der I²C Bus Multi-Master fähig. Die maximale Übertragungsgeschwindigkeit ist im konkreten Fall des Displays die gleiche wie beim SPI Bus, im I²C Standard sind allerdings auch Übertragungsraten bis zu 3,4 MBit/sec vorgesehen [I²CStandard].

Da dieser Schnittstellentyp dank der Standardisierung eine breite Palette erhältlicher Bausteine bietet, wurde diese Schnittstelle zur Verbindung des Displays an das FPGA Board gewählt. Näheres zur Schnittstelle ist in Kapitel 3 Grundlagen I²C zu finden.

2.4 Das Übertragungsprotokoll des EA-eDIP240

Das Display bietet ein eingebautes Protokoll zur Erkennung von Übertragungsfehlern. Dieses Protokoll ist unabhängig von der gewählten Übertragungsart (RS-232, RS-485, SPI oder I²C). Der Protokollaufbau ist dabei wie folgt:



- DC1 / DC2: signalisiert Datentransfer zum Display bzw. Aufforderung an das Display Daten zu senden, 1 Byte.
- length: gibt die Länge der Nutzdaten in Byte an, 1 Byte.
- data: maximal 64Bytes Nutzdaten sind möglich
- BCC: Prüfsummenbit, besteht aus der Summe MOD 256 von DC1 / DC2, length und Daten, 1 Byte.
- ACK / NACK: das Display sendet nach jedem empfangenen Datenpakte ein ACK, bzw. ein NACK zurück, je nachdem ob die Datenübertragung fehlerfrei war oder nicht, 1 Byte.

Von dem Display gesendete Datenpakete sind ebenfalls in den Protokollrahmen eingeschlossen, womit auf Empfängerseite die Richtigkeit der Daten ebenfalls geprüft werden kann. [eDIP240]

3 Grundlagen I²C

In diesem Kapitel werden die Eigenschaften des I²C Bus beschrieben, und wie der Datentransfer und die Zusammenarbeit zwischen den Bausteinen am Bus funktioniert.

3.1 Die Geschichte des I²C Bus

I²C, gesprochen *I-quadrat-C* oder *I-square-C*, entstanden aus *IIC = inter-IC*, ist eine Entwicklung der Firma Philips Semiconductors. Dieser serielle Bus wurde zu Beginn der 1980er Jahre entwickelt um auf einfache Art und Weise ICs wie Steuerungschips, Audio- / Video-Decoder, Signalprozessoren, EEPROMS etc. in Fernsehern zu verbinden. 1992 erfolgte die offizielle Standardisierung (Version 1.0) die bereits zahlreiche Erweiterungen gegenüber der ursprünglichen Implementierung enthielt (beispielsweise der Fast-Mode mit einer Vervierfachung der ursprünglichen Datenübertragungsrate). Im Jahre 1998 und 2000 folgten weitere Revisionen (2.0 und 2.1), welche weitere Änderungen und Erweiterungen (u.A. ein High-Speed Modus mit 3,4 MBit/sec Übertragungsgeschwindigkeit) umsetzten. Mittlerweile hatte sich der Bus zu einem Industriestandard entwickelt, und es sind unzählige Bausteine - alleine Philips führt mehr als 150 Bausteine mit I²C Schnittstelle - zur Anbindung an diesen Bus erhältlich. [I²CStandard] [I²CAN]

3.2 Busaufbau

Der I²C ist ein serieller Bus bestehend aus zwei bidirektionalen Signalleitungen, SDA und SCL.

- SDA: Datenleitung (**S**erial **Da**ta Line)
- SCL: Taktleitung (**S**ystem **C**lock **L**ine)

Es handelt sich hierbei um ein Master-Slave Bus, d.h. es existiert mindestens ein Master, und maximal 1023 Slaves (10Bit Adressbreite) an einem Bus. Die tatsächliche Anzahl wird meist vor erreichen des Limits durch elektrotechnische Voraussetzungen begrenzt – die Gesamtkapazität aller angeschlossenen Geräte darf 400 pF nicht überschreiten – jedoch existieren Bus-Bridge-Bausteine, welche diese Grenzen nach oben erweitern. Die Leitungen sind bidirektional, da die jeweilig am Bus aktiven Baustein beide Leitungen treiben. Die Signale auf der Datenleitung haben während des High-Pegels auf der Clock Leitung ihre Gültigkeit. [I²CStandard]

3.3 Technische Spezifikationen und Synchronisation

Alle angeschlossenen Bausteine am I²C Bus sind an beiden Busleitungen angeschlossen – siehe Abbildung 3-1. Die Leitungen werden über einen Pull-Up Widerstand auf den High-Pegel, logisch Eins, gezogen. Der jeweils treibende Baustein zieht die Leitung auf den Low-Pegel, logisch Null, um eine Null auf den Bus zu schreiben - somit muss ein Baustein an einer Leitung nur aktiv werden um eine Null zu schreiben. Sobald ein Baustein eine Null auf eine Leitung legt, kann kein anderer Baustein diesen Wert auf der Leitung ändern, weshalb diese Art der Verknüpfung auch als „Wired-AND-Verknüpfung“ bezeichnet wird.

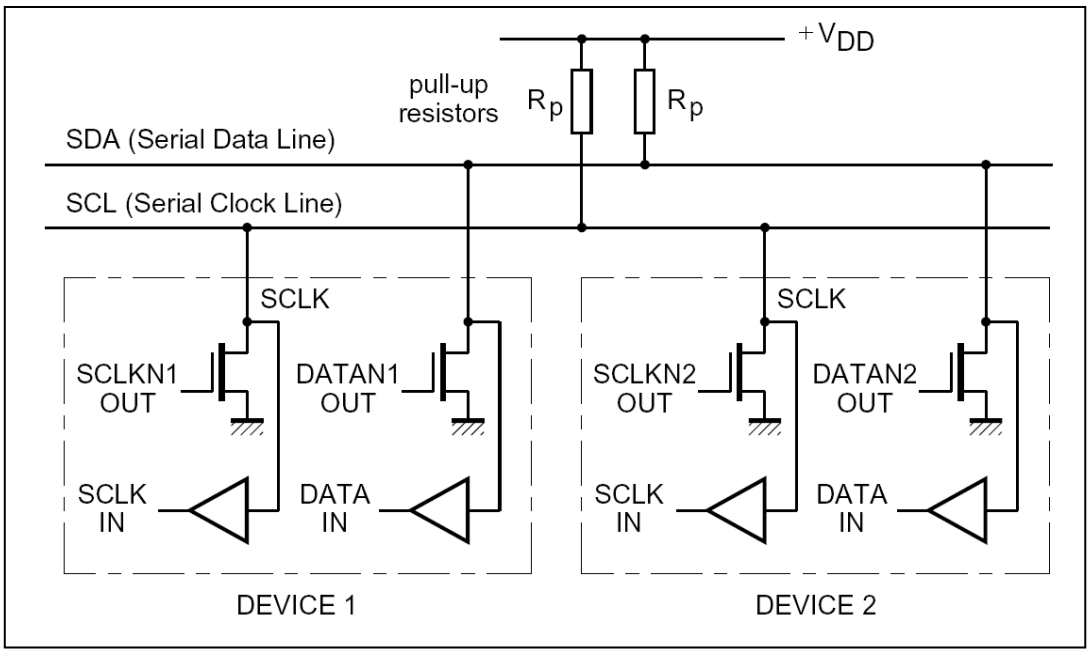


Abbildung 3-1
Elektrischer Anschluss der Geräte am I²C Bus
[I²CAN]

Diese Verknüpfungsart wird unter anderem dazu genutzt unterschiedlich schnelle Bausteine miteinander zu synchronisieren. Dabei wird die Geschwindigkeit auf der Clock-Leitung (SCL) vom langsamsten Gerät bestimmt: jedes Gerät hat seinen eigenen Taktzyklus. Bei einem Pegelwechsel von HIGH zu LOW auf der SCL beginnt für alle betroffenen Bausteine ein Countdown. Ist dieser Abgelaufen geben sie die SCL frei (sind also nicht mehr aktiv am Bus) und warten dann auf den Wechsel von LOW nach HIGH. Dieser geschieht jedoch erst wenn der Countdown des letzten Bausteines abgelaufen ist - somit werden alle betroffenen Bausteine auf den langsamsten synchronisiert. [I²CAN]

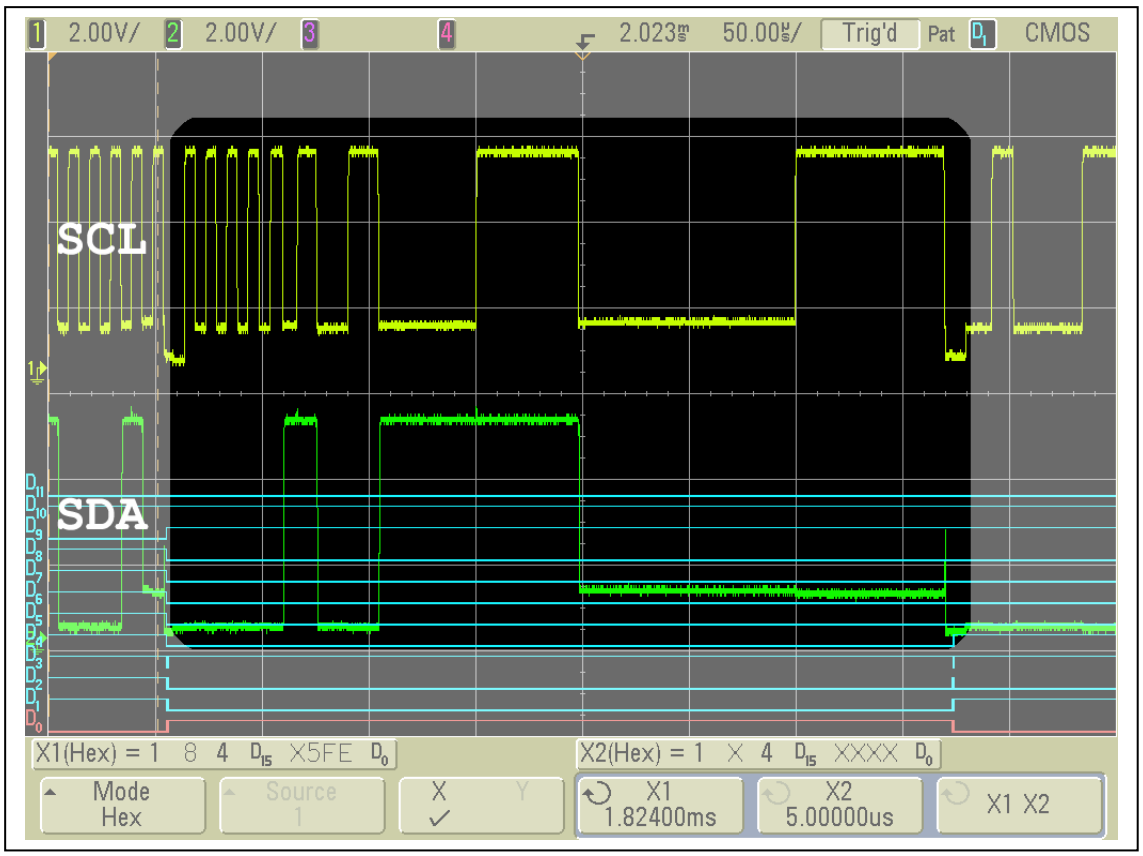


Abbildung 3-2
Taktdrosselung am I²C Bus

In Abbildung 3-2 ist der Ablauf einer Taktdrosselung anhand des Transfers eines Bytes vom I²C Interface zum Display zu sehen. Bei der Abbildung handelt es sich um einen Screenshot, der mit dem Agilent MSO6054A Oszilloskop bei einer Messung an der Prototyp-Platine erstellt wurde. [Agilent]

Nach anfänglich hoher Geschwindigkeit wird die Taktzyklusdauer auf der SDA Leitung immer größer. Offensichtlich ist einer der beiden Kommunikationspartner (vermutlich das Display) mit der Geschwindigkeit überfordert.

3.4 Datentransfer auf dem Bus

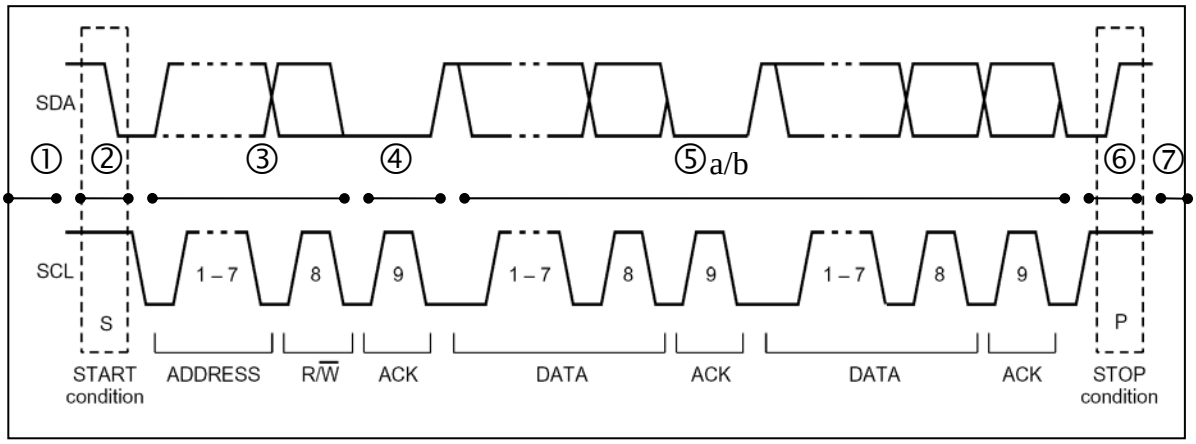


Abbildung 3-3
Ablauf eines Datentransfers auf dem I²C Bus
[I²CAN]

Als Master-Slave-System konzipiert ist es die Aufgabe des Bus-Masters einen Transfer zu initiieren. Hierzu muss der Master erst einmal den Bus belegen, bevor der eigentliche Transfervorgang stattfinden kann. Dies ist notwendig, da der Spezifikation gemäß mehrere Master an einem Bus existieren können. Im folgenden wird nun detailliert geschildert, wie ein kompletter Datentransfer anhand der in Abbildung 3-3 eingezeichneten Punkte auf dem I²C-Bus erfolgt. [I²CAN]

1. Ist der Bus frei, so sind SCL und SDA jeweils HIGH.

2. Nun können theoretisch mehrere Bus-Master gleichzeitig versuchen den Bus zu erlangen. Sie senden dazu ein START-Kommando auf den Bus (SDA und SCL werden nacheinander auf LOW gezogen).

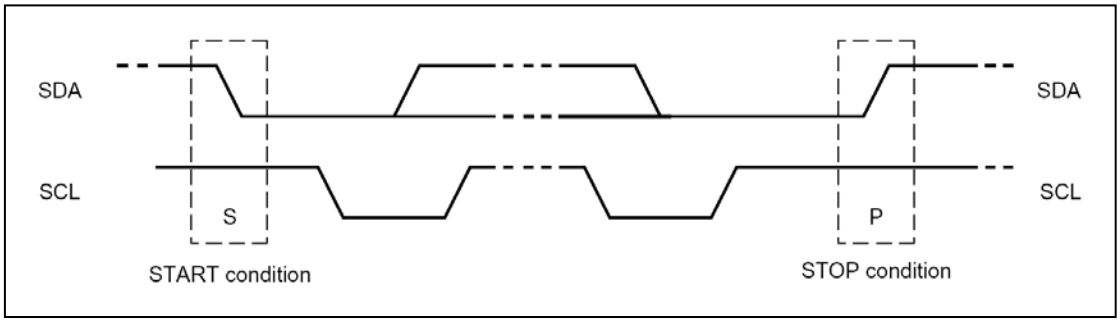


Abbildung 3-4
Start und Stop auf dem I²C Bus
[I²CAN]

Noch ist in einem Multi-Master-System nicht entschieden welcher Master den Bus bekommt. Denn bei gleichzeitigem Beginn der Busaquirierung durch mehrere Master verliert bei dem ersten nicht übereinstimmenden Bit derjenige Master den Bus, der versucht eine 1 senden, da der/die konkurrierende Master die Datenleitung auf 0 hält/halten.

3. Es wird die Adresse des Slave-Gerätes auf den Bus gelegt (im Basis-Modus 7 Bit, im erweiterten Adressmodus 10 Bit). An die Adresse angehängt befindet sich das R/W Bit, welches angibt ob das Slave-Gerät die Quelle oder das Ziel eines Datentransfers sein soll. Damit entscheidet sich auch ob der Master später als Master-Transmitter oder Master-Receiver arbeitet.

4. Das Gerät mit der entsprechenden Adresse antwortet mit einem ACK Signal. Dafür gibt der Master für einen Clock-Puls die SDA Leitung frei (diese ist dann durch den Pull-Up HIGH), und der Slave kann diese dann auf LOW ziehen – das ACK Signal.

An dieser Stelle ist nun zu unterscheiden, ob der Master als Transmitter oder Receiver fungiert.

Master Transmitter: der Master möchte dem Slave Daten schicken.

5.a Die Daten werden byteweise übermittelt, jedes empfangene Byte wird vom Slave mit einem ACK quittiert (vergleiche Punkt 4).

Master Receiver: der Master forderte den Slave auf Daten zu schicken. Hierzu muss der Master wissen wie viele Bytes zu empfangen sind!

5.b Das Slave Gerät hat volle Kontrolle über SDA und SCL. Jedes übertragene Byte wird mit einem ACK von dem Master quittiert. Das letzte Byte wird nicht quittiert - hierfür muss der Master wissen wie viele Bytes er lesen soll - danach gibt der Slave den Bus wieder frei

Hierbei ist zu beachten: im Falle eines Multi-Master-Systems kann es sich bei übereinstimmender Ziel-Adresse und R/W-Bit im Master Receiver Modus – d.h. mehrere Master möchten gleichzeitig Daten von dem selben Gerät lesen – erst beim Abschluss des Datentransfers durch einen der Master – also beim finalen Nichtquittieren – entscheiden, welcher Master den Bus erhält. Konkret erhält derjenige Master den Bus, welcher noch weitere Daten lesen möchte, da dieser beim ACK eine logische Null auf den Bus gelegt hat.

Randbemerkung: somit ist es also theoretisch möglich, dass mehrere absolut synchron laufende Master von einem Gerät parallel Daten lesen, solange sie exakt die selbe Datenmenge anfordern.

6. Nach abgeschlossenem Datentransfer sendet der Master ein STOP-Kommando auf den Bus, SCL und SDA werden nacheinander auf HIGH gesetzt.

7. Nach einer definierten Wartezeit, in der der Bus ruhen muss, kann der Prozess von vorne beginnen.

Da es im I²C kein Protokoll gibt um anzuzeigen, dass keine Daten mehr vorliegen, **muss** der Slave also Daten schicken, auch wenn er keine mehr hat (z.B. sein interner Sendepuffer leer ist). Das eDIP240-7 schickt in diesem Falle Nullbytes.

4 Systemaufbau

In diesem Kapitel wird das System im Ganzen, dessen Komponenten sowie deren Funktion beschrieben. Ebenfalls werden die Entscheidungsgrundlagen, aufgetretene Probleme und deren Lösung vorgestellt.

4.1 Grundriss des Gesamtsystems

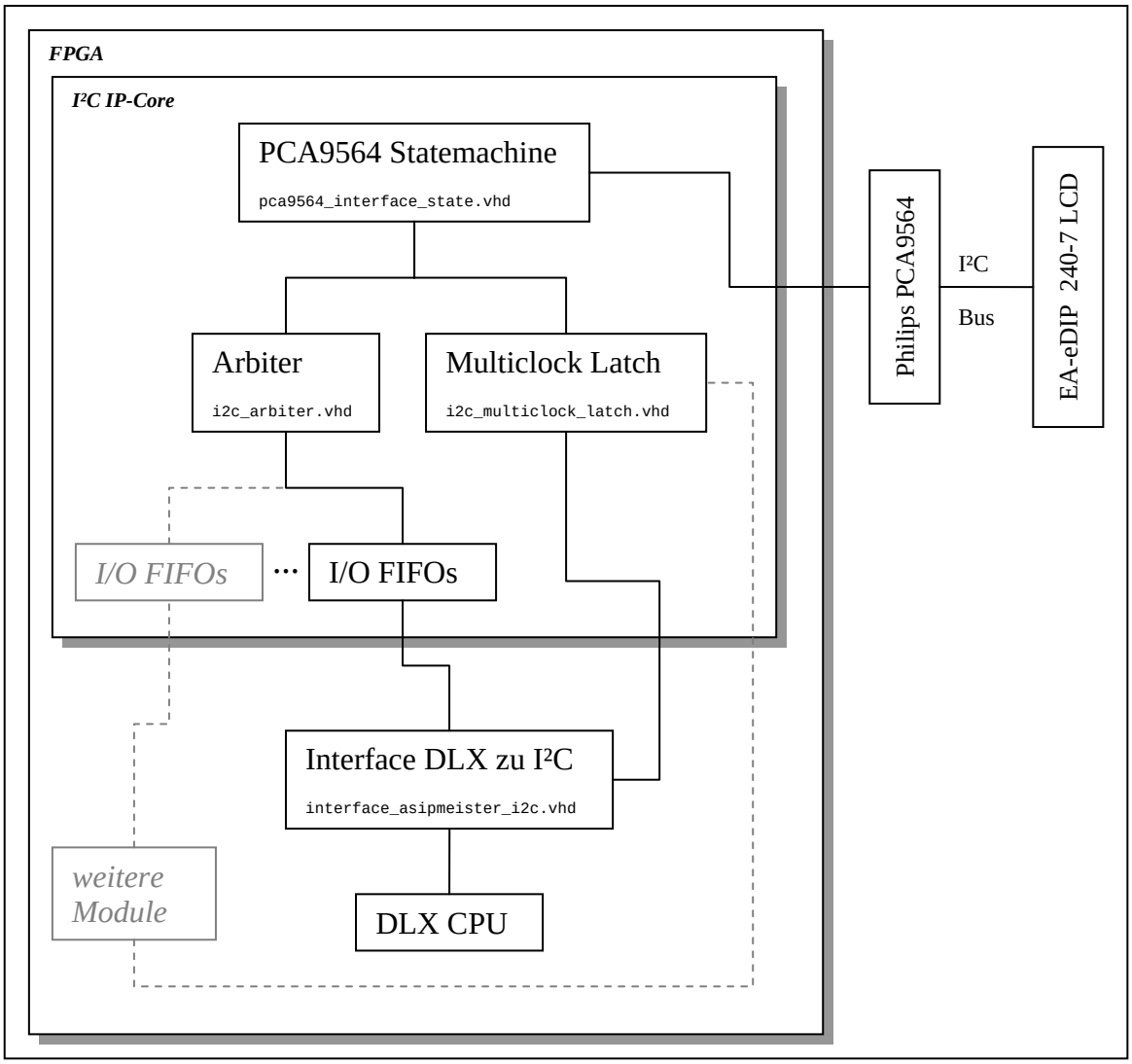


Abbildung 4-1
Aufbau des Gesamtsystems nach Abschluss der Entwicklung

Bei den im Rahmen dieser Studienarbeit entwickelten Komponenten ist der Quellcode aufgeführt. Dieser findet sich im Anhang A – Anhang E. Zu den weiteren Komponenten siehe auch [PCA9564AN] und [eDIP240].

Im folgenden wird auf die einzelnen Komponenten des Systems eingegangen:

4.1.1 Die FPGA Prototyping Boards

Das angefertigte System, bestehend aus VHDL Code, sowie der Platine für I²C Interface und dem LCD, kann mit zwei am Lehrstuhl vorhandenen FPGA Protoyping Boards betrieben werden.

Das HW-AFX-FF1152-200 Prototyping Board von Xilinx enthält einen Virtex-II FPGA (XC2V3000) mit insgesamt 720 für den Nutzer frei belegbaren I/O-Pins. Diese Pins sind in vier Feldern rund um den FPGA nach außen geführt. [Xilinx]

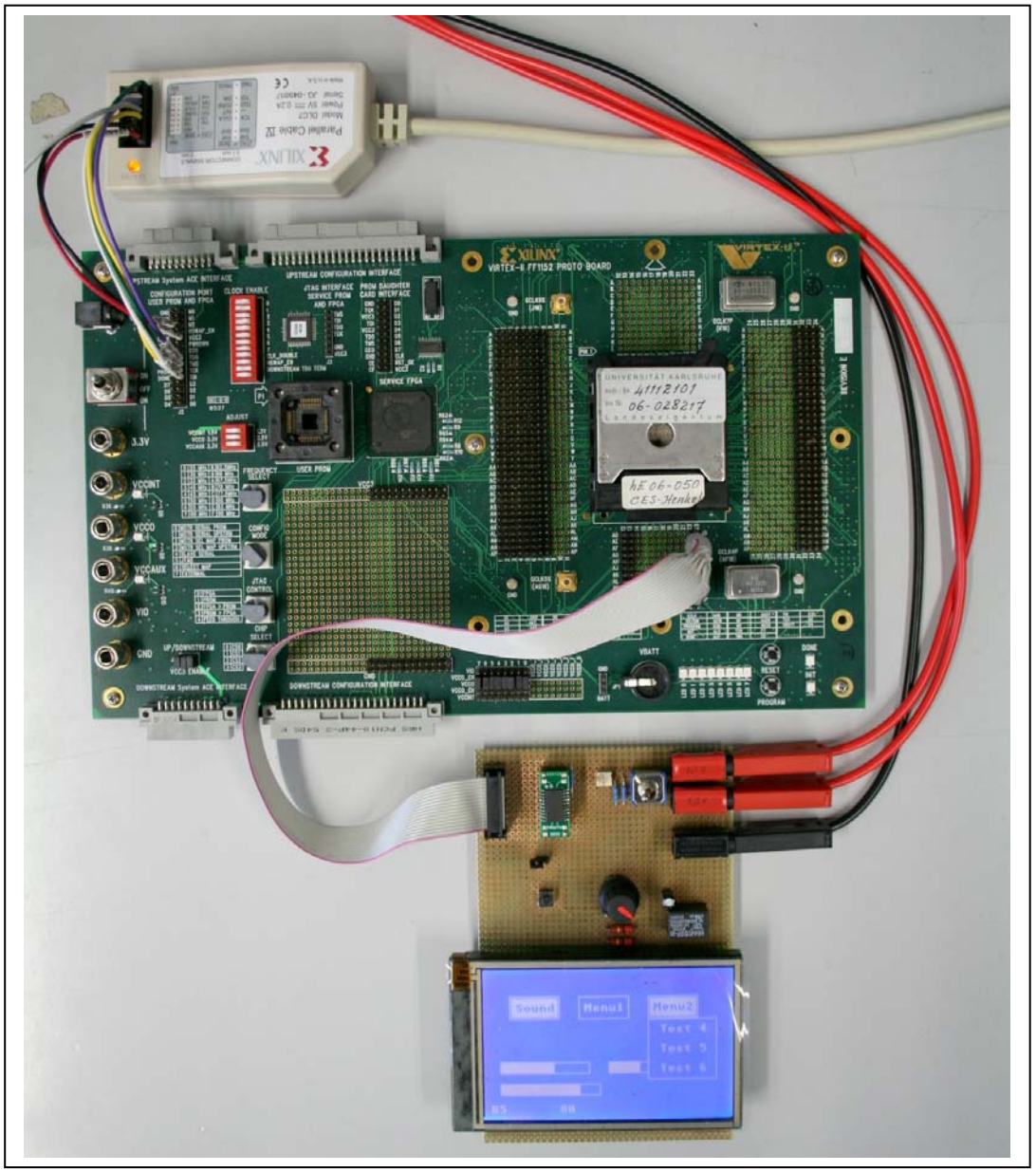


Abbildung 4-2
Xilinx HW-AFX-FF1152-200 mit LCD Platine

Über die I/O Pins können Signale aus dem FPGA heraus, sowie hineingeführt werden. Des Weiteren existieren ein EEPROM Sockel, sowie JTAG und eine System-ACE

Schnittstelle, welche zur Programmierung des FPGA genutzt werden können. Der FPGA wurde über JTAG und einem Xilinx Programmierkabel konfiguriert.

Es erfolgte ebenfalls eine Portierung auf das Digilent Virtex-II Pro Development System mit Virtex-II Pro FPGA (XC2VP30). Der FPGA ist direkt über eine USB Schnittstelle konfigurierbar. Mit diesem System wurde eine Case-Study für das DodOrg (Digital on Demand computing Organism for Real Time Systems) Projekt im Rahmen des Organic Computing Schwerpunktprogramms der Deutschen Forschungsgemeinschaft aufgebaut. Näheres hierzu siehe Kapitel 5.3 Die Case Study im Projekt DodOrg [Digilent] [DodOrg] [DodOrg2]

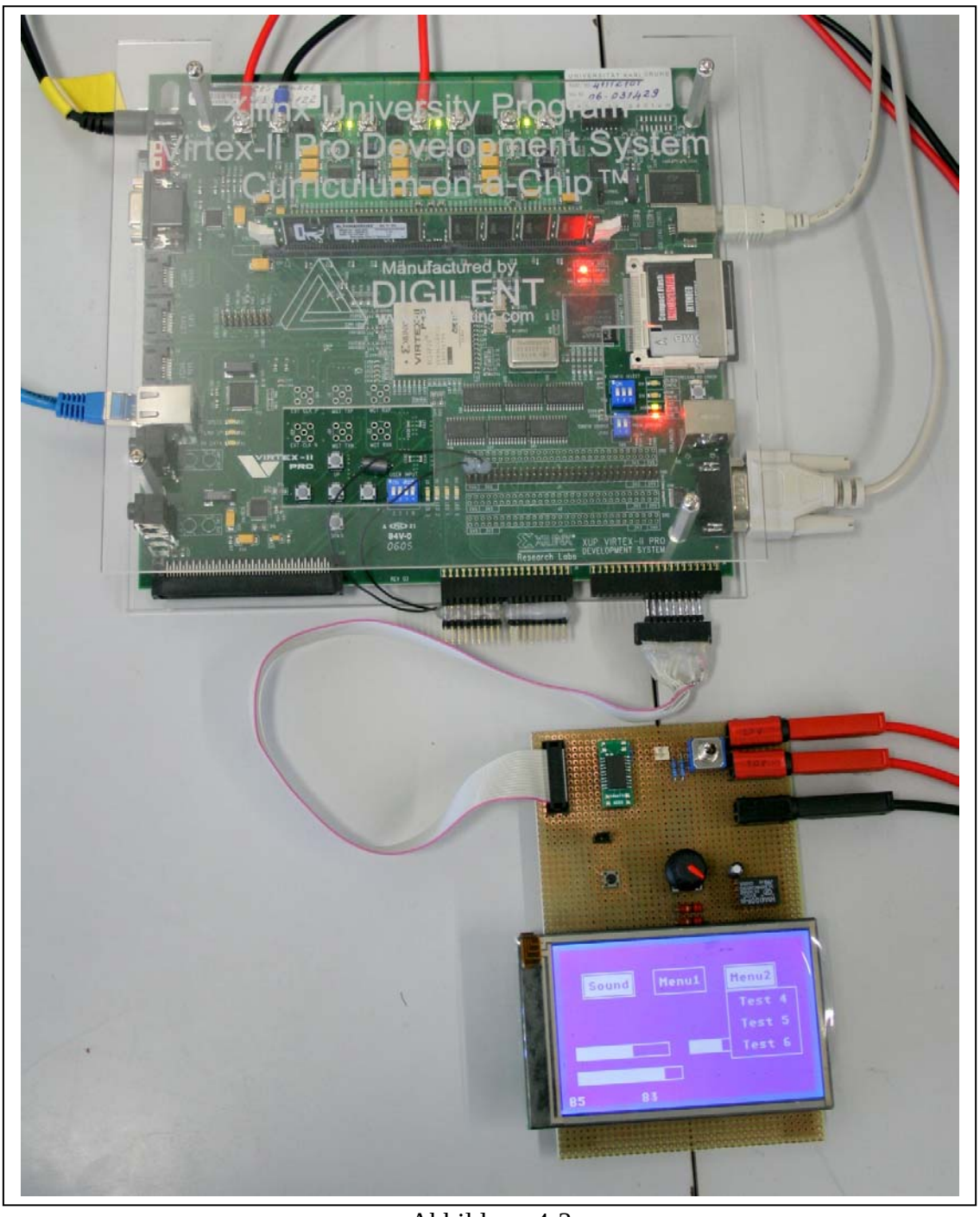


Abbildung 4-3
Digilent Virtex-II Pro Development System mit LCD Platine

4.1.2 Der PCA9564

Für die Anbindung des LCD an das Board wurde die I²C Schnittstelle gewählt, da sich im Gegensatz zur RS232 weitere Geräte an diesen Bus anschließen lassen können, sowie das Bussystem im Gegensatz zur SPI Schnittstelle standardisiert ist (siehe Kapitel 2.3 Die Schnittstellen des EA-eDIP240 und Kapitel 3 Grundlagen I²C). Eine Möglichkeit zur Realisierung der Schnittstelle am FPGA hätte auch über eine auf www.opencores.org frei erhältliche VHDL Implementierung der I²C Funktionen bestanden. Jedoch ist hierfür eine Pegelwandlung von den am Ausgang der FPGA anliegenden TTL / CMOS33 Pegel auf CMOS5 Ebene (High-Pegel 4,7V) des I²C Bus notwendig.

Da beide I²C Leitungen bidirektional verwendet werden, wird eine der Taktrate des I²C Bus angepasste Pegelwandlung in beide Richtungen benötigt. Zu diesem Zweck existieren zahlreiche Wandlerbausteine (z.B. der MAX3373 von Maxim), welche jedoch alle durchwegs für SMD gefertigt werden. Diese Gehäuseformen (z.B. TSSOP) sind auf hochintegrierte Schaltungen ausgelegt, mit folglich sehr kleinem Gehäuse und einem nicht mehr handverlötbaren Pin-Abstand von 0.65mm.

Die Lösung bestand darin einen dedizierten I²C Interfacebaustein zu verwenden, welcher auch in handlicher Bauform mit größerem Pin-Abstand erhältlich ist - im konkreten Fall des verwendeten Philips PCA9564 sind es 1.27mm Pin-Abstand. Mit Hilfe einer Adapterplatine lässt sich der Chip nun in einen DIL Sockel platzieren.

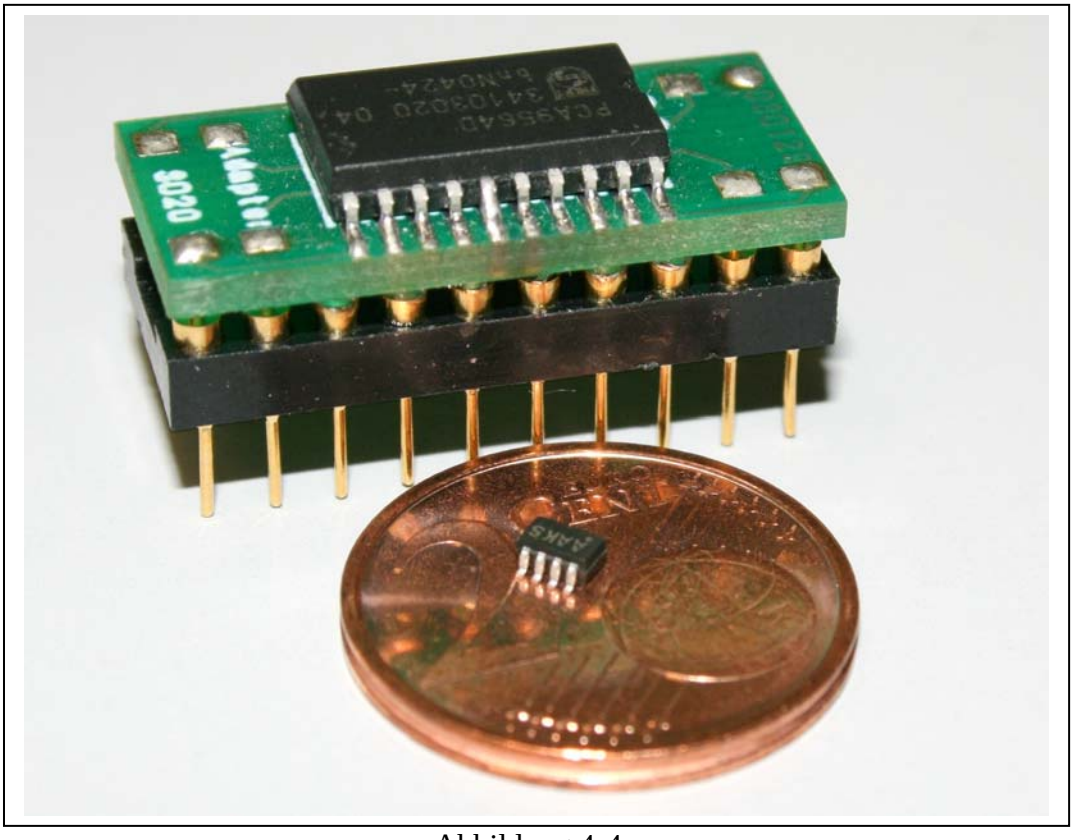


Abbildung 4-4
Philips PCA9564 auf Adapterplatine (hinten) und Maxim MAX3373 (vorne) im Größenvergleich

Neben der handlicheren Bauform bietet dieser Bausteines gegenüber der frei erhältlichen VHLD Implementierung einen weiteren signifikanten Vorteil: der Baustein wurde vom Entwickler des I²C Standards entworfen, und unterstützt somit auch alle im I²C Standard vorgesehenen Funktionen – wogegen die frei erhältliche Open Source Implementierung natürlich keinen Anspruch auf Korrektheit und Vollständigkeit erhebt!

Der Interfacebaustein wird mit 3,3V betrieben, kann aber bis zu 6V an den I/O-Pins verkraften. Damit eignet er sich zusätzlich als Interface zwischen CMOS33 und CMOS5 Spannung.

Der Philips PCA9564 ist ein Master / Slave fähiger I²C Interfacebaustein – kann also sowohl als Master, sowie auch als Slave am Bus betrieben werden. Der Datentransfer zu / vom Baustein erfolgt über 8 Leitungen (D0-D7), so dass von der Steuereinheit aus ein ganzes Byte angelegt werden kann. Der Chip besitz fünf interne Register: Status-, Daten-, Address- Timeout- und Kontroll-Register, welche über zwei Leitungen adressiert werden (A0-A1) sowie einige Steuerleitungen (CE, WR, RD, INT, RESET).

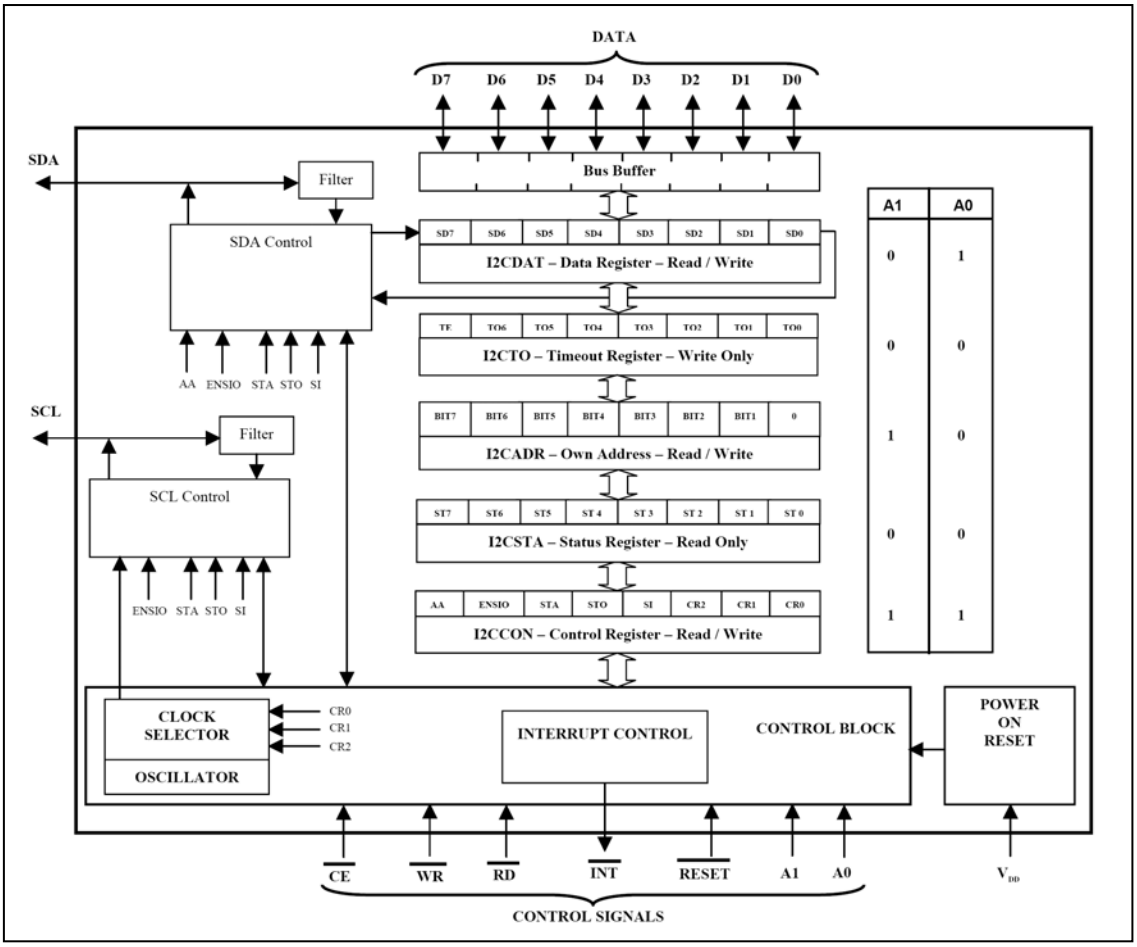


Abbildung 4-5
PCA9564 Blockdiagramm
[PCA9564AN]

4.1.3 Die PCA9564 Statemachine

Zur Ansteuerung des PCA9564 musste eine in VHDL programmierte Statemachine entwickelt werden. Dies war notwenig, da für einen Transfer auf dem I²C Bus mehrere Befehle nacheinander an den PCA9564 Interfacebaustein geschickt werden müssen. Mit Hilfe der Statemachine ist es möglich taktweise die Kommandos an den Baustein zu senden und auf Rückmeldungen entsprechend zu reagieren.

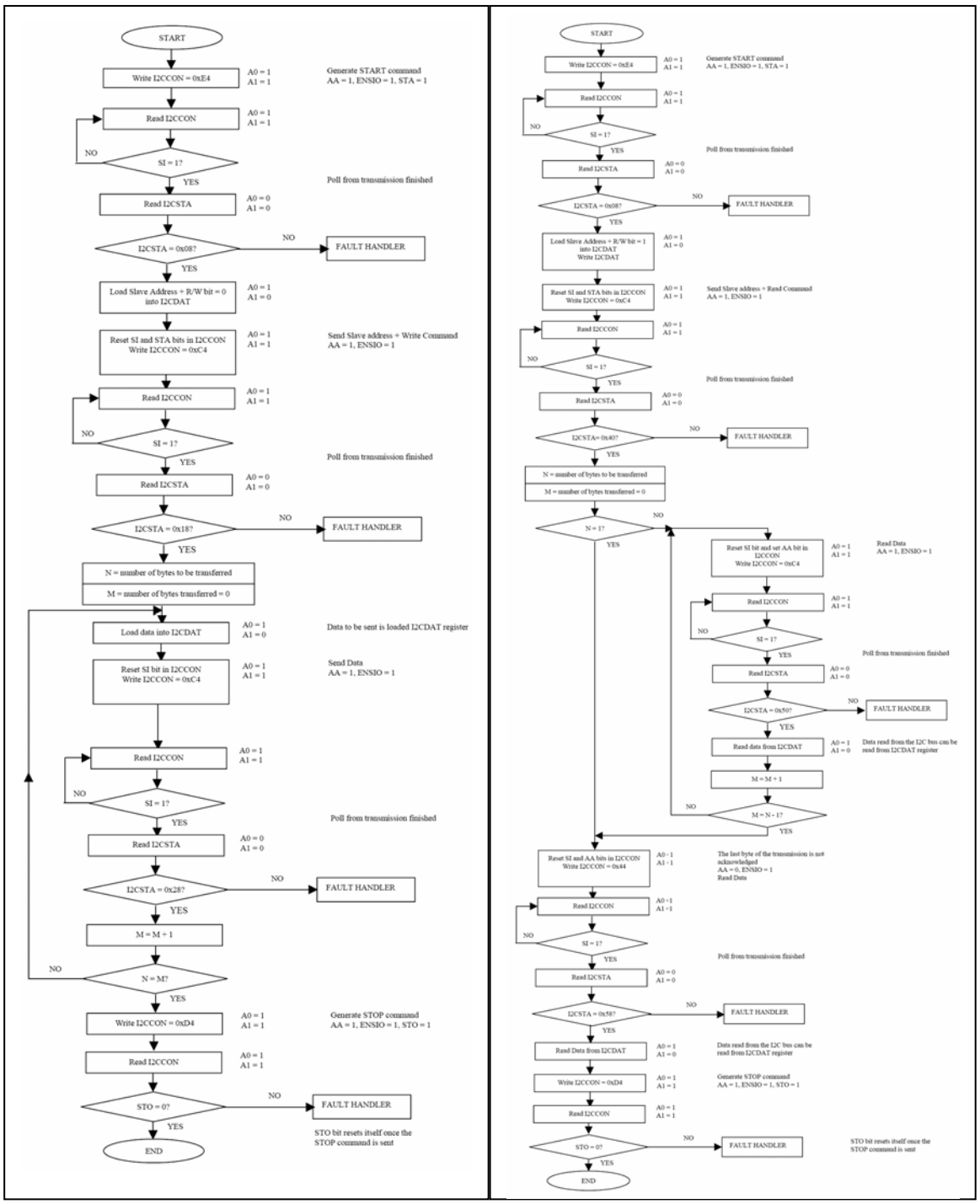


Abbildung 4-6
Flussdiagramm Master Transmitter und Master Receiver Mode
[PCA9564AN]

Der Quellcode der aus den in Abbildung 4-6 dargestellten Flussdiagrammen abgeleiteten Statemachine ist im Anhang A kommentiert aufgeführt.

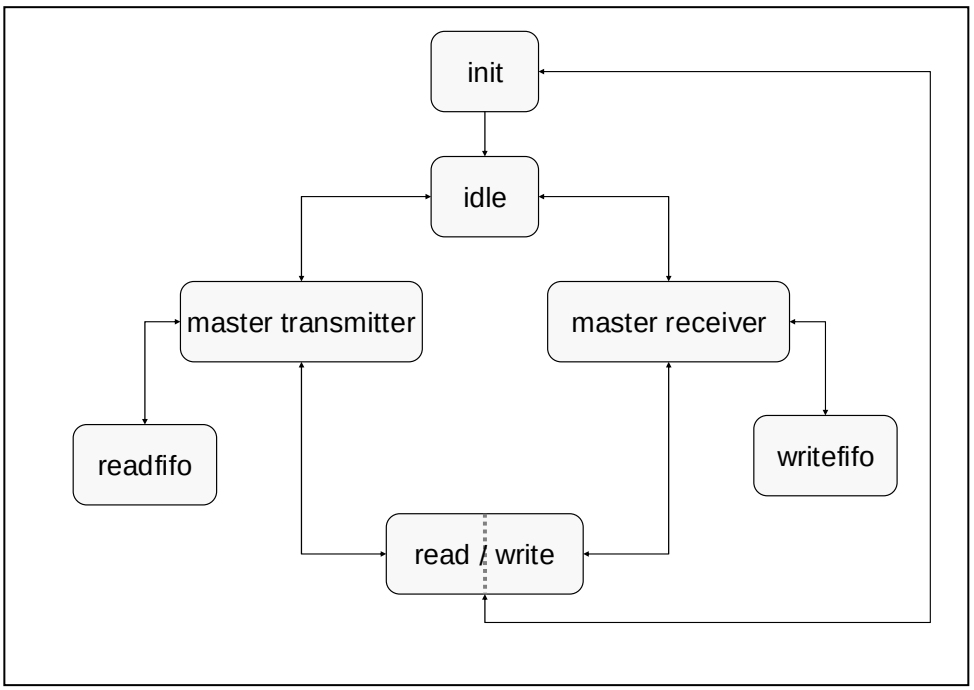


Abbildung 4-7
Vereinfachter, aus dem Flussdiagramm abgeleiteter Automat

In Abbildung 4-7 ist ein vereinfachter Automat dargestellt, der im wesentlichen die Struktur der erstellten Statemachine wiedergibt. Hier nun im Detail die Funktionen der dargestellten Zustände:

- init: wird nach einem Reset der Statemachine ausgeführt, initialisiert den PCA9564 – u.A. wird die Bus Frequenz und die Adresse gesetzt. Besteht aus 9 Zuständen.
- idle: Wartezustand auf Übertragungsanforderungen. Besteht aus 2 Zuständen.
- master transmitter: Transfermodus in dem der PCA9564 als Master Daten über den Bus schickt. Besteht aus 12 Zuständen.
- master receiver: Transfermodus in dem der PCA9564 als Master Daten vom Bus liest. Besteht aus 17 Zuständen.
- readfifo / writefifo: Lesen von über den I²C Bus zu sendenden Daten, bzw. Schreiben von über den I²C Bus empfangenen Daten. Bestehen aus jeweils 3 Zuständen.
- read / write: Lesen, bzw. Schreiben von Registern auf dem PCA9564. Bestehen aus 2, bzw. 3 Zuständen.

Es existieren noch weitere Zustände, die für Aufgaben wie z.B. Wartezyklen zuständig sind. Jedoch wurde der Übersichtlichkeit wegen darauf verzichtet diese einzuzeichnen.

4.1.4 Der Arbiter, die FIFOs und das Multiclock Latch

Um mehreren Geräten gleichzeitig den Zugang zum I²C Interface zu ermöglichen, existiert für jedes Gerät eine eigene Eingabe- und Ausgabe-FIFO, in der auf den I²C Bus zu sendende und vom Bus empfangene Daten zwischengespeichert werden. Dabei wurde auf von Xilinx Coregen erzeugte FIFOs zurückgegriffen. [FIFO]

Der zwischen FIFOs und Statemachine geschaltete Arbiter aus Abbildung 4-1 hat die Aufgabe Zugangsanforderungen an den I²C Bus von den an den I²C IP-Core angeschlossenen Modulen zu arbitrieren, sowie die FIFOs und Steuersignale des jeweilig gerade aktiven Moduls an die Statemachine anzubinden. Der Arbiter verwendet ein Round-Robin Verfahren, gewährt also reihum den angeschlossenen Modulen Zugriff auf den I²C Bus.

Da die Steuersignale an den I²C-IP Core aus anderen sog. Clock Domänen stammen können – d.h. die angeschlossenen Geräte können mit einem höheren oder niedrigeren Takt als der IP-Core betrieben werden – müssen die Signale zur jeweiligen (hier positiven) Taktflanke der Quell Domäne abgegriffen, und zur Taktflanke der Ziel Domain geschrieben werden. Ohne diesen Zwischenschritt ist es möglich, das zwischen zwei Taktflanken der anderen Clock Domain gelesen wird. Da jedoch nur an den Taktflanken ein stabiles Signal durch das Synthesetool garantiert wird, Signale sich zwischen den Taktflanken in unbekanntem Zustand befinden können, ist dieses Puffern der Signale unerlässlich.

Sowohl die Anschlussbreite für die Steuersignale, wie auch die Anzahl der FIFO-Anschlüsse wird über ein GENERIC festgelegt. Damit kann bei der Instantiierung des I²C IP-Cores festgelegt werden wie viele Geräte maximal an den Core angeschlossen werden können. Die dafür notwendige Anzahl FIFOs wird dann automatisch erzeugt, sowie die Bit-Breite der Interface Signale entsprechend konfiguriert.

4.1.5 Der I²C IP-Core

Die Zusammenschaltung der PCA9564 Statemachine, des Arbiters, des Multiclock Latches und der FIFOs ergibt den I²C IP-Core, der ein allgemeines I²C Interface darstellt. Mit diesem ist es also möglich aus VHDL Code heraus mit mehreren Geräten auf den I²C Bus zu schreiben, bzw. davon zu lesen.

4.1.6 Eine Beispielanbindung

Die unter ASIPMeister [ASIPMeister] erzeugte DLX CPU [Hennessy96] wurde an das I²C Interface angebunden. Zu diesem Zweck wurde ein weiteres Interface entworfen (siehe auch Abbildung 4-1), welches es ermöglicht die Steuerleitungen für den I²C IP-Core aus Sicht der DLX CPU zu multiplexen und die CPU von Steueraufgaben zu entlasten. Letztendlich muss die DLX CPU nur in eine FIFO im Interface-Baustein die Zieladresse, Datenmenge und zu sendende Daten schreiben. Der Datentransfer zum IP-Core, sowie die Steuerung des IP-Cores (Adressierung des Gerätes am I²C Bus, Handshakes mit dem Arbiter und der Statemachine, etc.) wird von dem Interfacebaustein übernommen. Leseanforderungen an den I²C Bus sind ebenfalls möglich, hierzu muss die CPU die Geräteadresse und die Anzahl der zu lesenden Bytes übermitteln. Der Interface-Baustein wickelt den Transfer ab, wonach die CPU die Daten dann direkt aus

der Ausgangs-FIFO des I²C IP-Cores lesen kann. Details zur CPU-seitigen Implementierung der Anbindung sind in Kapitel 5.1 Anbindung der CPU an das I²C Interface zu finden. Der Quellcode des Interface-Bausteines ist im Anhang E aufgeführt.

4.2 Der Aufbau der LCD Platine

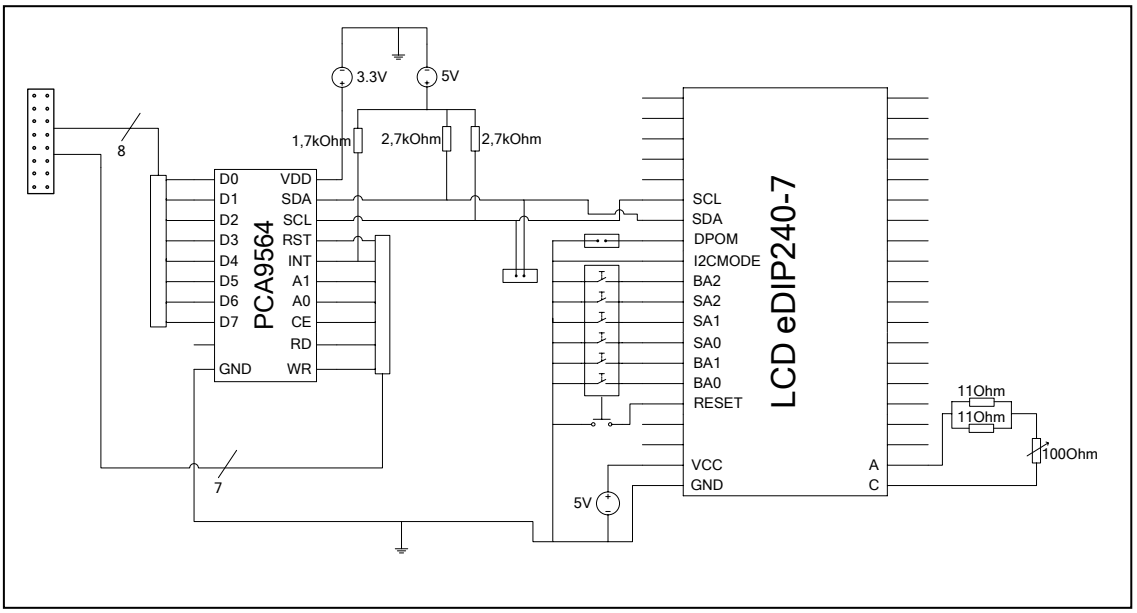


Abbildung 4-8
LCD Platine Schema

Philips PCA9564	
D0-D7	Datenleitungen
SDA	I²C Datenleitung
SCL	I²C Taktleitung
RST	Reset Pin
INT	Low wenn Daten anliegen
A0-A1	Addressleitungen
CE	Chip Enable
RD, WR	Lesen/Schreiben

[PCA9564AN]

LCD eDIP240-7	
SDA	I²C Datenleitung
SCL	I²C Taktleitung
DPOM	Deaktiviert Power On Makro
I2CMODE	Pin auf Masse aktiviert den I²C Modus
SA0-SA2	Slave Address
BA0-BA2	Base Address
RESET	Setzt Display zurück

[eDIP240]

Der grundlegende Entwurf der Schaltung erfolgte auf einem Steckbrett. Wie aus Abbildung 4-8 ersichtlich sind sowohl die beiden I²C Leitungen SDA und SCL, als auch die Interrupt Leitung des PCA9564 über einen Pull Up mit der Spannungsquelle verbunden. Die Widerstände an den I²C Leitungen müssen entsprechend der am Bus angeschlossenen Kapazitäten angepasst werden, siehe hierzu Kapitel 3.2 Busaufbau und [I2CStandard].

Für den dauerhaften Betrieb wurde eine Platine gelötet welche in Abbildung 4-9 zu sehen ist.

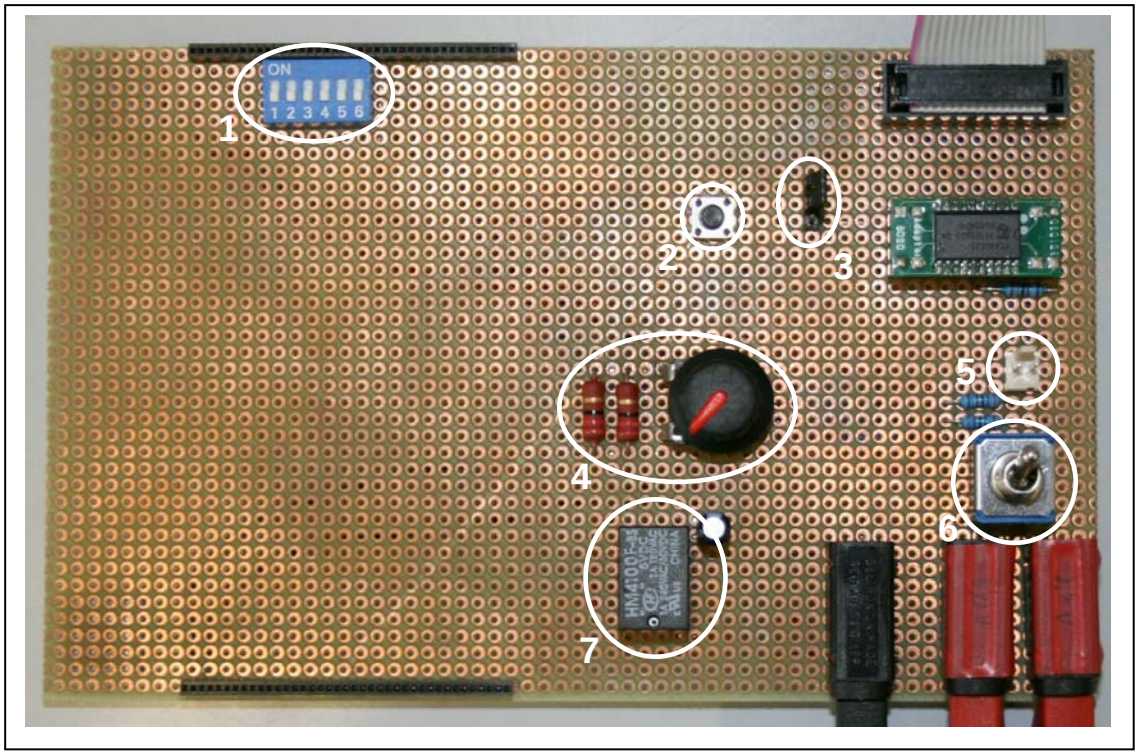


Abbildung 4-9
LCD Platine Bestückungsseite

Dabei wurde zusätzlich zu den notwendigen Elementen folgendes hinzugefügt:

1. DIP Schalter um dem Display eine Adresse zuzuweisen (SA0-SA2, BA0-BA2).
2. Resetschalter um das Display zurückzusetzen.
3. Jumper um ein (fehlerhaftes) Bootmakro des LCD zu deaktivieren.
4. Potentiometer um die Displayhelligkeit zu steuern.
5. Anschlusspins für den I²C Bus um weitere Geräte an den Bus anschließen zu können.
6. Ein-/Ausschalter um das gesamte Board vom Stromkreis zu trennen.
7. Relais um das Display und den I²C Bus erst bei vorhandener 3,3V Spannung am PCA9564 Interfacechip mit der 5V Spannung zu versorgen. Dies verhindert, dass an den Eingangpins des PCA9564 eine Spannung anliegt, bevor der Chip selbst mit Spannung versorgt wird. Damit wird eine Zerstörung des Chips verhindert, der ohne Spannungsversorgung nur 3,6V an den Eingängen verkraftet. [PCA9564]

4.3 Die Implementierung

In den nun folgenden Punkten greife ich einige prominente Probleme der Implementierung auf die gelöst werden mussten, um eine stabile Betriebsfrequenz der Statemachine von bis zu 35MHz zu erreichen.

4.3.1 Organisation der Statemachine

Zur Steuerung des PCA9564 wurde eine Statemachine implementiert – siehe hierzu Kapitel 4.1.3 Die PCA9564 Statemachine. Diese Statemachine war zu Beginn als zweistufig hierarchische Statemachine ausgelegt. Dies brachte den Vorteil der Strukturierung und damit erhöhter Übersichtlichkeit mit sich – z.B. war der Master Transmitter und der Master Receiver Modus jeweils ein Zustand auf der oberen Ebene, während der in den Zuständen auszuführenden Code die Subzustände bildeten.

Jedoch stellte sich im Laufe der Entwicklung heraus, das dies zu größeren Timingproblemen führte, da das verwendete Xilinx Synthesetool den hierarchischen Automaten offensichtlich nicht zu einem einstufigen umfunktionierte. Dies hatte zur folge, das offensichtlich der Wechsel von einer Sub-Level-Statemachine zu einer anderen nicht sauber geschah, was zu fehlerhaften Werten im Zustandsregister führte. Mit dem hierarchischen Automaten war ein stabiler Betrieb nur bis 35kHz (!) möglich, wogegen das Xilinx Synthesetool eine maximale Geschwindigkeit von 106MHz angab.

Durch Umarbeiten in eine einstufige Statemachine konnte die Schaltung stabil mit einer Frequenz von 135kHz betrieben werden.

4.3.2 Signalabfrage und Metastabilität

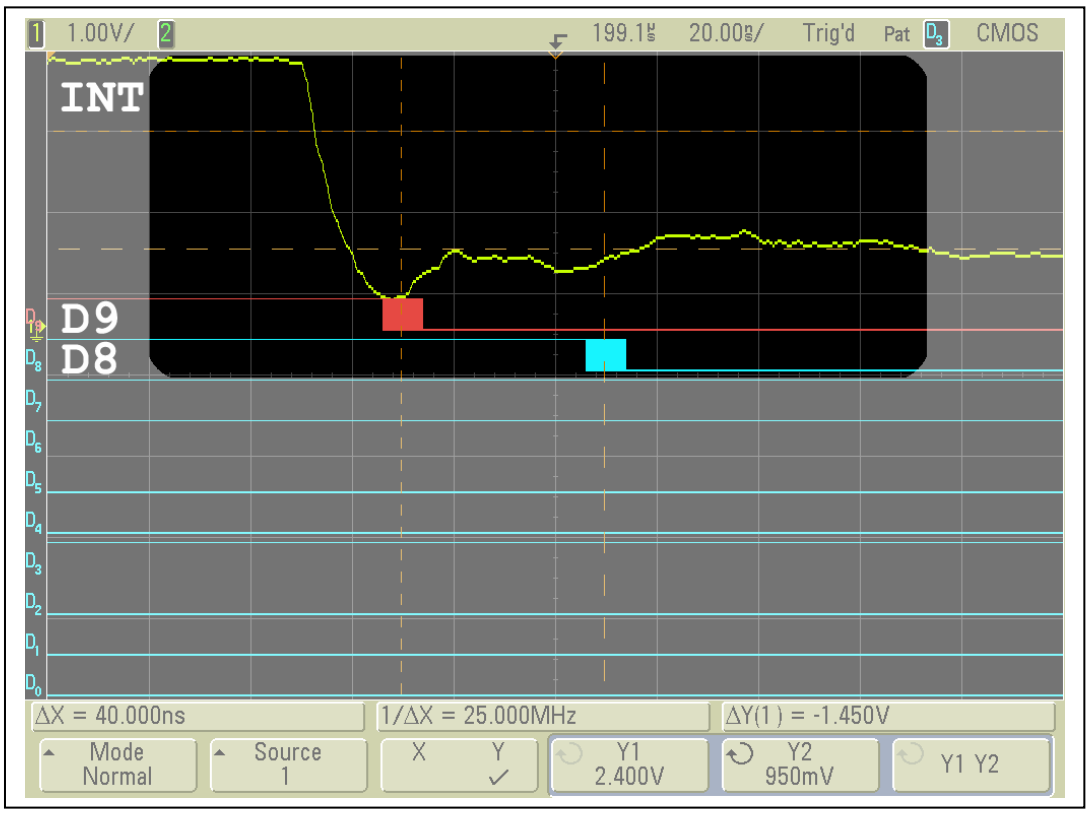


Abbildung 4-10
Analoges und digitalisiertes INT Signal

Weitere Frequenzsteigerungen waren möglich, nachdem das Lesen des INT Signals verändert wurde. Im Detail stellte sich das Problem wie folgt dar: das INT Signal des PCA9564 dient dazu dem angeschlossenen Mikrocontroller über den Abschluss einer

Transaktion, und damit einem neuen Wert im Statusregister zu informieren. Die Abfrage dieses Signals geschieht in der Statemachine sehr häufig, da nach jedem Befehl wie z.B. Start / Stop auf dem Bus, Adresse senden, Byte senden / empfangen, etc. auf den Abschluss dieser Transaktion gewartet werden muss, um danach das Statusregister auf eventuelle Fehler zu prüfen. An dieser Stelle zeigt sich nun folgendes Verhalten der Statemachine: wie in Abbildung 4-10 deutlich zu sehen wechselt das analog gemessene INT Signal von High auf Low – ebenso wechselt das zum Debugging wieder hinausgeführte, digitalisierte INT Signal D9 von High auf Low. Es scheint jedoch dann, als ob die Statemachine dies ignoriert, da sie weiterhin im Wartezustand verharrt, also keine Änderung der Zustandsausgabe auf D0-D7 erfolgt.

Wird anstatt das INT Signal direkt zu lesen dessen Wert durch einen weiteren VHDL-Prozess zuerst gepuffert – in der Synthese sollte dies dazu führen, dass ein Register eingefügt wird, welches dann von der Statemachine anstatt des Signals gelesen wird – so tritt dieser Fehler nicht auf. Im direkten Vergleich ist dieses gepufferte Signal als D8 ebenfalls in Abbildung 4-10 zu sehen. Die Verzögerung gegenüber dem ungepufferten Signal D9 entsteht durch den VHDL-Prozess.

Dieses Fehlerverhalten ist vermutlich auf den Effekt der Metastabilität zurückzuführen. Sollte das INT Signal genau während einer Änderung des Signals gelesen werden (dies entspricht einer Verletzung der Setup oder Hold Time), kann dies zu einem durchpropagieren der Signaländerung durch den Schaltkreis führen, und damit zu unvorhersagbarem Verhalten – im Fall der Statemachine das sie hängen bleibt. Dieses Problem tritt immer dann auf, wenn asynchrone Signale (hier: die Signale des PCA9564) in einen synchronen Schaltkreis überführt werden. In der Praxis verringert man die Wahrscheinlichkeit dieses Fehlers durch Einfügen von einem oder mehreren hintereinander geschalteten Registern, denn damit propagiert die Metastabilität nur in den Schaltkreis wenn zum Lesezeitpunkt des Registers dessen Ausgabe in einem metastabilen Zustand ist. Mit dieser Änderung lies sich die Statemachine stabil mit einem Takt von 3,5MHz betreiben.

Im Übrigen ist dieser Effekt bei den ebenfalls vom PCA9564 gelesenen Datensignalen D0-D7 nicht zu beobachten, da hierfür im Datenblatt Setup Zeiten vermerkt sind [PCA9564], die natürlich in der Statemachine beachtet werden.

Bei der Abbildung handelt es sich um einen Screenshot, der mit dem Agilent MSO6054A Oszilloskop bei einer Messung an der Prototyp-Platine erstellt wurde. Die von der Messrate und der Messdauer abhängigen Messauflösung sind der Grund für die bei D8 und D9 auftretenden Blöcke. [Agilent]

4.3.3 Timingprobleme

Eine weitere Erhöhung des Taktes führte zu Problemen mit dem Timing. Beispielsweise bei aufeinander folgenden Lesebefehlen an den PCA9564, die von dem PCA9564 mit INT=0 quittiert werden, las der zweite Zugriff den vorherigen (ersten) Wert zurück. Der Grund dafür liegt darin, dass die Statemachine nach dem ersten INT=0 Lesen so schnell den zweiten Befehl schickt, dass in diesem Zeitraum das INT Signal noch nicht den High-Pegel erreicht hat. Somit ist es also notwendig nach Rücksetzen des INT Signals auf INT=1 zu warten, bevor weitere Befehle an den PCA9564 geschickt werden können.

Damit ist ein stabiler Betrieb der Statemachine bei 35MHz möglich. Bei höheren Frequenzen scheinen unsaubere Signale vermehrt ein Problem darzustellen, und es wären auch zusätzliche Wartezyklen notwendig um eindeutige Signalwerte sicherzustellen. Da das LCD ohne spezifische Anpassung der Datenübertragung (Wartezyklen zwischen jedem Byte) nicht schneller als 100kHz betrieben werden kann – somit einen höher getakteten Bus wie in Kapitel 3.3 Technische Spezifikationen und Synchronisation geschildert drosseln wird – wurde darauf verzichtet die im Zusammenhang mit dem Aufbau maximal mögliche Frequenz der Statemachine weiter zu erhöhen. Denn bei der letztendlich gewählten Betriebsfrequenz von 20MHz ist die maximale Übertragungsgeschwindigkeit auf einem mit 100kHz betriebenen I²C Bus problemlos möglich (d.h. der PCA9564 muss nicht auf Eingaben von der Statemachine warten).

4.3.4 Handshake-Protokoll

Um eine Kommunikation des I²C IP-Cores mit den angeschlossenen Modulen zu ermöglichen wurde ein Handshake-Protokoll entwickelt. Dies ist notwendig, um sicherzustellen, dass sowohl das an den IP-Core angeschlossene Modul, als auch der IP-Core wissen in welchem Zustand sich der jeweils andere befindet.

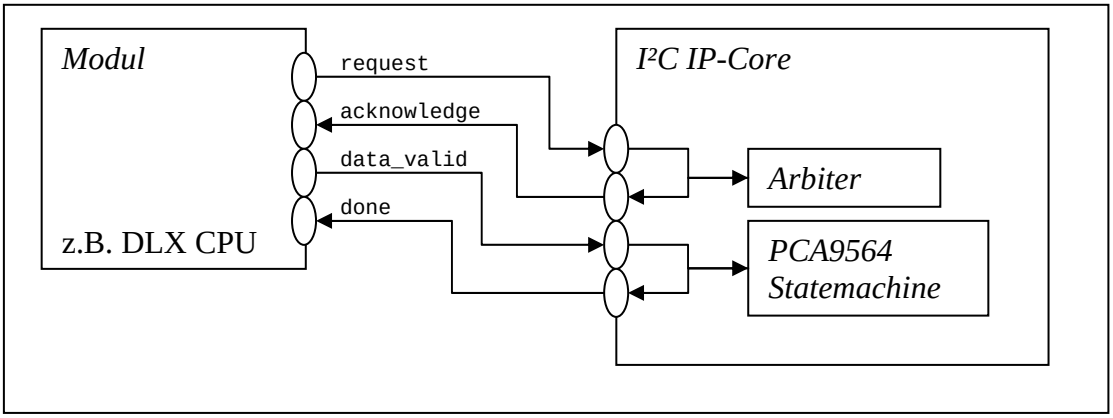


Abbildung 4-11
Kommunikationssignale zwischen Modul und I²C IP-Core

Konkret sieht ein Ablauf mit Hilfe der in Abbildung 4-11 gezeigten Signale wie folgt aus:

1. Das angeschlossener Modul signalisiert über eine High-Pegel auf `request` das es Zugriff auf den I²C Bus wünscht. Dieser Pegel muss gehalten werden, solange Zugriff auf den Bus gewünscht ist.

2. Der Arbiter des IP-Cores antwortet mit einem High-Pegel auf `acknowledge` wenn er den Zugang gewährt. Dieser Wert bleibt bestehen, solange `request` auf High gehalten wird.

3. Von dem Modul angelegte Steuersignale wie beispielsweise die Adresse des Kommunikationspartners auf dem I²C Bus und die I/O FIFOs werden nun an die Statemachine von dem Arbiter weitergeschaltet. Das Modul signalisiert der Statemachine mit einem High-Pegel auf `data_valid`, dass alle benötigten Steuersignale anliegen, nun also der eigentliche Transfer auf dem Bus begin-

nen kann. Der High-Pegel auf `data_valid` und die Steuersignale müssen bis zum Abschluss des Datentransfers stabil bleiben.

4. Ist der Datentransfer auf dem I²C Bus abgeschlossen, legt die Statemachine einen High-Pegel an `done` an.

5. Das Modul quittiert dies mit einem Low-Pegel auf `data_valid`, die Statemachine dies wiederum mit einem Low-Pegel auf `done`.

6. Das Modul selbst kann mit einem Low-Pegel auf `request` den Zugriff auf den I²C Bus für andere Module wieder freigeben, oder mit einem erneuten High-Pegel auf `data_valid` einen erneuten Transfer auf dem Bus starten. Dies muss nicht innerhalb eines bestimmten Zeitfensters erfolgen, da `request` lediglich ein Signal an den Arbiter darstellt die vom Modul angelegten Signale an die Statemachine weiterzuleiten.

5 Anbindung an die ASIPMeister CPU

In diesem Kapitel wird die softwareseitige Anbindung der ASIPMeister [ASIPMeister] DLX CPU [Hennessy96] an das Hardwareinterface des I²C Bus beschrieben. Ebenso wird beschrieben welche softwareseitigen Interfaces geschaffen wurden, um eine einfache und schnelle Nutzung der LCD Funktionen zu ermöglichen

5.1 Anbindung der CPU an das I²C Interface

Wie in Kapitel 4.1.6 Eine Beispielanbindung geschildert ist die CPU hardwareseitig über ein weiteres Interface an den I²C IP-Core angeschlossen.

Die DLX CPU wurde hierfür dahingehend modifiziert, indem zwei Assembler Befehle eingefügt wurden, um ein Byte in eine FIFO zu schreiben (`putc`), bzw. ein Byte aus einer FIFO zu lesen (`getc`). Somit können mit Hilfe von Inline Assembly im C-Quellcode Zeichen direkt in die angebundenen FIFOs geschrieben werden. Die Aufrufparameter sind zwar vom Typ integer, jedoch werden nur die unteren 8 Bits an die FIFOs weitergegeben. Der C Quellcode ist im Anhang F enthalten.

Ein Protokoll wurde entworfen, um über diese Schnittstelle auch die Steuerdaten für den I²C IP-Core zu übertragen. Dabei sind die im Nachfolgenden aufgeführten Felder jeweils 1 Byte groß:

Lesen:	data length	address + R	bytes to read
Schreiben:	data length	address + W	data

- data length: gibt die Anzahl Bytes von data an. Wenn vom I²C Bus gelesen wird, ist diese Zahl irrelevant.
- address + R/W: enthält die 7-Bit Adresse auf dem I²C Bus, sowie ein Bit das Lesen oder Schreiben signalisiert. Dabei bedeutet R/W = 1 lesen, R/W = 0 schreiben.
- bytes to read: dieses Feld gibt die Anzahl der vom I²C zu lesenden Bytes an.
- data: enthält das zu schreibende Datenpaket.

Mit Hilfe dieses Protokolls sind die folgenden Szenarien abgedeckt:

1. Die CPU schreibt langsamer in die FIFO als der Interfacebaustein die Daten ließt. Damit der Interfacebaustein weis, wann die Daten vollständig sind und

der I²C Transfervorgang angestoßen werden kann, muss zuvor die Anzahl der zu sendenden Bytes bekannt sein.

2. Die CPU schreibt schneller in die FIFO, als der Interfacebaustein die Daten an den I²C IP-Core weitergeben kann. In diesem Fall ist es durch die bekannte Anzahl der zu sendenden Bytes möglich, mehrere Befehlspakete in der FIFO zu unterscheiden. Ein Programm kann also mehrere Datenpakete auf den I²C Bus schreiben, ohne deren Übertragung abwarten zu müssen.

5.2 Die LCD Funktions-Library

Zur einfacheren Verwendung der LCD Funktionen aus einem C Programm heraus wurde eine Library programmiert. Details zu allen vom LCD unterstützen Funktionen sind im Datenblatt des LCDs zu finden [eDIP240]. Im Anhang F ist der Quellcode der LCD Library aufgeführt. Im folgenden werden nun die einzelnen Funktionen dieser Library beschrieben:

5.2.1 Prüfen des Inhaltes des LCD Sendepuffer

```
int checkbuffer()
```

Gibt die Anzahl der im LCD Sendepuffer verfügbaren Bytes zurück. Dieses Kommando kann genutzt werden, um auf einen Rückgabewert des Displays zu warten. Beispielsweise beim Drücken einer Taste auf dem Display wird von dem Display die mit der Taste verknüpfte Rückgabenummer in den Sendepuffer geschrieben.

5.2.2 Anfordern von Daten vom LCD Sendepuffer

```
int getbytes (char* dest, int bytes_to_read)
```

Liest eine zuvor in `bytes_to_read` zu spezifizierende Anzahl von Bytes aus dem Sendepuffer aus. Dies ist notwendig, da die Menge der zu übertragenden Daten nach I²C Standard im Voraus bekannt sein muss, siehe Kapitel 3.4 Datentransfer auf dem Bus. Die Menge der zur Verfügung stehenden Daten lässt sich mittels `checkbuffer()` prüfen. Siehe hierzu Kapitel 5.2.1 Prüfen des Inhaltes des LCD Sendepuffer. Der Zeiger auf `dest` gibt das Ziel der Daten an. Die Funktion gibt die Anzahl der gelesenen Bytes zurück.

5.2.3 Allgemeine Sendeoperation

```
int sendcommand (const char cmd0, const char cmd1,
     const int options[], const char text[], int intcount,
     int charcount, int address)
```

Dies ist eine allgemein gehaltene Sendeoperation, die es ermöglicht alle vom Display angebotenen Funktionen aufzurufen. Ein Displaykommando setzt sich wie folgt zusammen:

- zwei Zeichen die das Kommando spezifizieren (hier `cmd0`, `cmd1`)

- je nach Kommando mehrere Zahlenoperatoren (hier `options[]` mit Längenangabe in `intcount`)
- je nach Kommando eine Zeichenkette (hier `text[]` mit Längenangabe in `charcount`)
- die Adresse des Displays (spezifiziert als `#define` in lib_lcd.c)

Dieses Kommando wird von allen nachfolgenden Operationen genutzt – es handeln sich damit also um reine Convenience Funktionen. Eine vollständige Liste aller vom Display angebotenen Funktionen ist in [eDIP240] zu finden.

5.2.4 Ausgabe auf dem Terminal des Displays

```
int t_print (const char* str)
```

Schreibt eine Zeichenkette auf das Terminal. Hier ist die Verwendung von Steuerzeichen wie \r (Carriage Return) und \n (Newline) möglich.

5.2.5 Cursor auf dem Terminal ein- / ausschalten

```
int t_cursor (int onoff)
```

Schaltet den blinkenden Cursor des Terminals ein (`onoff != 0`) oder aus (`onoff = 0`).

5.2.6 Terminal ein- / ausschalten

```
int t_enable (int onoff)
```

Schaltet das Terminal ein (`onoff != 0`) oder aus (`onoff = 0`). Ist das Terminal ausgeschaltet, werden Ausgaben auf dem Terminal verworfen. Bereits auf dem Terminal ausgegebener Text ist gepuffert, wird also beim Einschalten des Terminals wieder angezeigt.

5.2.7 Graphisch eine Zeichenkette ausgeben

```
int g_print (const char* str, int x, int y)
```

Schreibt die Zeichenkette in `str` an die Position an den Koordinaten `x` und `y`. Dabei dürfen keine Steuerzeichen wie \r (Carriage Return) und \n (Newline) verwendet werden, da diese vom Display als Terminalzeichen gewertet werden.

5.2.8 Rechteck zeichnen

```
int g_drawrect (int x1, int y1, int x2, int y2)
```

Zeichnet ein leeres Rechteck an den angegebenen Eckkoordinaten.

5.2.9 Linie zeichnen

```
int g_drawline (int x1, int y1, int x2, int y2)
```

Zeichnet eine Linie von den Start- zu den Zielkoordinaten.

5.2.10 Bargraphen definieren

```
int g_makebar (int x1, int y1, int x2, int y2, int low_val,
     int high_val, int init_val, int type, int fill_type, int touch)
```

Erzeugt einen Bargraphen an den angegebenen Koordinaten `x1`, `y1`, `x2` und `y2`. `low_val` gibt den unteren Wert, `high_val` den oberen Wert und `init_val` den initialen Wert des Bargraphen an. `type` und `fill_type` spezifizieren das Aussehen des Bargraphen. `touch` legt fest ob der Bargraph über das Touchfeld veränderbar sein soll.

Dabei schreibt das Display den Wert zusammen mit der Nummer des Bargraphen in den Sendepuffer des Displays wenn er durch den Nutzer verändert wurde. Die Nummer des Bargraphen wird fortlaufend von der Funktion mitgezählt und im Rückgabewert zurückgegeben. Maximal sind 32 Bargraphen auf dem Display möglich.

5.2.11 Vorhandenem Bargraphen einen Wert zuweisen

```
int g_setbar (int barnum, int value)
```

Weist einem bereits vorhandenen Bargraphen mit der Nummer `barnum` einen Wert `value` zu.

5.2.12 Schalter definieren

```
int g_makeswitch (const char* str, int x1, int y1, int x2, int y2,
     int down, int up)
```

Erzeugt einen Schalter an den angegebenen Koordinaten `x1`, `y1`, `x2` und `y2`. Die Beschriftung des Schalters wird in `str` übergeben. Dabei wird das erste Zeichen als Steuerzeichen für die Ausrichtung der Beschriftung des Schalters ausgewertet: C = zentriert, L = linksbündig, R = rechtsbündig. `down` und `up` geben den Code an, der beim Drücken, bzw. beim Loslassen des Schalters in den Sendepuffer geschrieben wird. Wird als Wert von down und up jeweils 0 definiert, wird nichts in den Sendepuffer geschrieben.

5.2.13 Radiogruppe definieren

```
int g_makeradiogroup (int group_number)
```

Neu erzeugte Schalter werden der in `group_number` definierten Radiogruppe zugeordnet. Es ist immer nur ein Schalter einer Radiogruppe aktiv. Wird die Radiogruppe 0 gewählt, bedeutet dies eine Deaktivierung der Funktion – d.h. neue Schalter gehören damit keiner Gruppe an.

5.2.14 Menüknopf definieren

```
int g_makemenubutton (const char* str, int x1, int y1, int x2, int y2,
     int down, int up, int select, int space)
```

Ein Schalter wird an den angegebenen Koordinaten `x1, y1, x2` und `y2` erzeugt. Wird der Schalter gedrückt klappt ein Menü auf. Der Schalter und die Menüeinträge werden über `str` definiert. Dabei gibt das erste Zeichen an, wie das Menü ausklappen soll (R = rechts, L = links, O = oben, U = Unten), das zweite Zeichen gibt an wie der Text auf dem Schalter positioniert werden soll (C = zentriert, L = linksbündig, R = rechtsbündig). Menüeinträge werden über das Zeichen | getrennt. `down` ist der Rückgabewert wenn der Menüschalter gedrückt wird, `up` wird beim Abbrechen des Menüs ohne Auswahl zurückgegeben, und `select` gibt den Basisrückgabewert für die Auswahl eines Menüeintrages an. Der Rückgabewert eines Menüeintrages errechnet sich wie folgt: Basisrückgabewert + Eintragsnummer – 1. `space` gibt den Abstand in Pixeln zwischen den Menüeinträgen an. Dieser Wert gilt global, also auch für eventuell bereits definierte Menüschalter

5.2.15 Display löschen

```
int d_clear()
```

Löscht den graphischen Inhalt des Displays (nicht den Terminal Inhalt!)

5.3 Die Case Study im Projekt DodOrg

Im Schwerpunktprogramm der Deutschen Forschungsgemeinschaft [DodOrg2] wurde eine Case Study entwickelt. Diese enthielt unter anderem die DLX CPU, das I²C Interface und einen ebenfalls an die CPU angebundenen Tongenerator. Auf der DLX CPU wurde eine speziell für dieses Projekt entwickelte Laufzeitumgebung namens LarsIX ausgeführt. Für das Projekt wurde ein Thread implementiert, welcher für die Aktualisierung der in der Laufzeitumgebung integrierten LCD-Verwaltungsstrukturen zuständig ist. Neu erzeugte Objekte auf dem LCD (z.B. Bargraphen, Schalter etc.) werden beim Erzeugen über eine für die Laufzeitumgebung programmierte LCD-Library in die LCD-Verwaltungsstrukturen eingetragen. Der LCD Thread prüft fortwährend, ob Daten im Sendepuffer des LCD anliegen, fordert diese gegebenenfalls an, wertet sie aus und aktualisiert die LCD-Verwaltungsstruktur.

Mit diesem Aufbau ist es möglich über das LCD gesteuert Töne auszugeben, die Tonhöhe zu variieren, und die CPU Auslastung durch die Tongenerierung zu visualisieren. Ebenso wird die Ausführung der für die Tonausgabe zuständigen Threads gesteuert, da nach Wahl entweder ein Thread aktiviert ist, der den Ton ungefiltert ausgibt, oder ein Thread der den erzeugten Ton vor der Ausgabe über einen Rauschfilter glättet. Dies geschieht über die Abfrage der LCD-Verwaltungsstrukturen, in denen der Zustand eines Schalters auf dem LCD geprüft wird, und je nach Wert eben Code ausgeführt wird, oder ein sofortiger Threadwechsel veranlasst wird.

6 Ergebnisse

In diesem Kapitel werden Messergebnisse zum Platzverbrauch der Implementierung auf dem FPGA vorgestellt. Ebenso werden Messergebnisse für den Zeitverbrauch von Funktionen aufgeführt.

6.1 Syntheseergebnisse

6.1.1 I²C IP-Core

Da die Anzahl der an das Interface anschließbaren Bausteine variabel ist, ist folglich auch der Platzverbrauch abhängig von der gewählten Anzahl der Bausteine, da damit unter anderem die Zahl der generierten FIFOs variiert. Im nachfolgenden werden die Syntheseergebnisse unter Xilinx ISE 8.1 (SP3) für den Xilinx XC2V3000 FPGA für sechs verschiedene Konfigurationen aufgeführt:

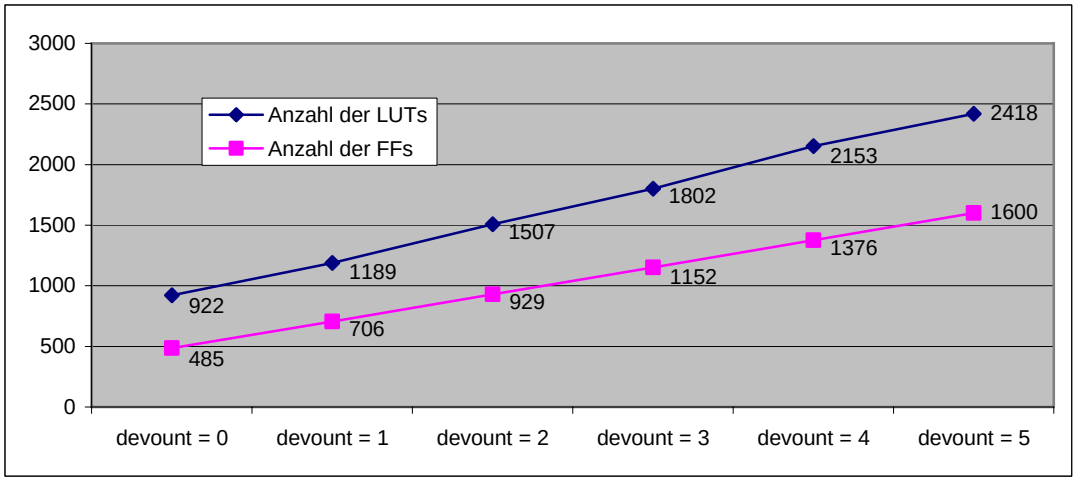


Abbildung 6-1
Platzbedarf des IP-Cores bei unterschiedlicher Anzahl anschließbarer Module

Die vom Synthesetool angegebene maximale Geschwindigkeit variiert zwischen 176 MHz und 186 MHz.

6.1.2 I²C IP-Core mit ASIPMeister DLX CPU

Im nachfolgenden werden die Syntheseergebnisse der DLX CPU, welche über das für diese CPU erstellte Interface mit dem I²C IP-Core verbunden ist, aufgeführt. Siehe hierzu auch Kapitel 5.1 Anbindung der CPU an das I²C Interface. Hierfür wurde der IP-Core so konfiguriert, das nur ein Baustein mit ihm verbunden werden kann.

Das Projekt wurde unter Xilinx ISE 8.1 auf dem Xilinx XC2V3000 FPGA synthetisiert.

	genutzt	vorhanden	Nutzungsgrad
Anzahl der FFs	3.858	28.672	13%
Anzahl der LUTs	10.161	28.672	35%
entsprechende Gatteranzahl	4.306.738		

6.2 Geschwindigkeitsmessungen

Ein Großteil der in Kapitel 5.2 Die LCD Funktions-Library aufgeführten Funktionen wurden einer Laufzeitmessungen unterzogen. Hierfür wurde ein an die CPU angebundener Zyklenzähler jeweils vor und nach dem Ausführen der Funktion ausgelesen. Dabei gilt folgendes zu beachten:

1. Die Messergebnisse beinhalten den Overhead für das Auslesen des Zählers. Dieser beläuft sich auf 4 CPU-Takte.

2. Die Zeit wurde gemessen von der Ausführung der Funktion bis zur Bestätigung des Displays über den Erhalt des Befehls. Ob das Display noch weitere Zeit benötigt den Befehl umzusetzen, oder ob es den Befehlserhalt erst bestätigt wenn dieser bereits ausgeführt wurde ist nicht bekannt.

3. Die Ausführungszeit eines Befehles unterliegt einer gewissen Varianz, vermutlich herbeigeführt durch dynamisch unterschiedlich schnelle Taktung des I²C Busses (siehe hierzu Kapitel 3.3 Technische Spezifikationen und Synchronisation). Dazu wurde um dies zu Teilen zu kompensieren für die im nachfolgenden aufgeführten Messungen ein Mittelwert aus 20 Messungen gebildet. Zu Ermittlung der Häufigkeitsverteilung der benötigten CPU-Takte wurde automatisiert 10.000 Ausführungen der Funktion `g_makeswitch()` gemessen. Das Ergebnis ist in Abbildung 6-2 dargestellt.

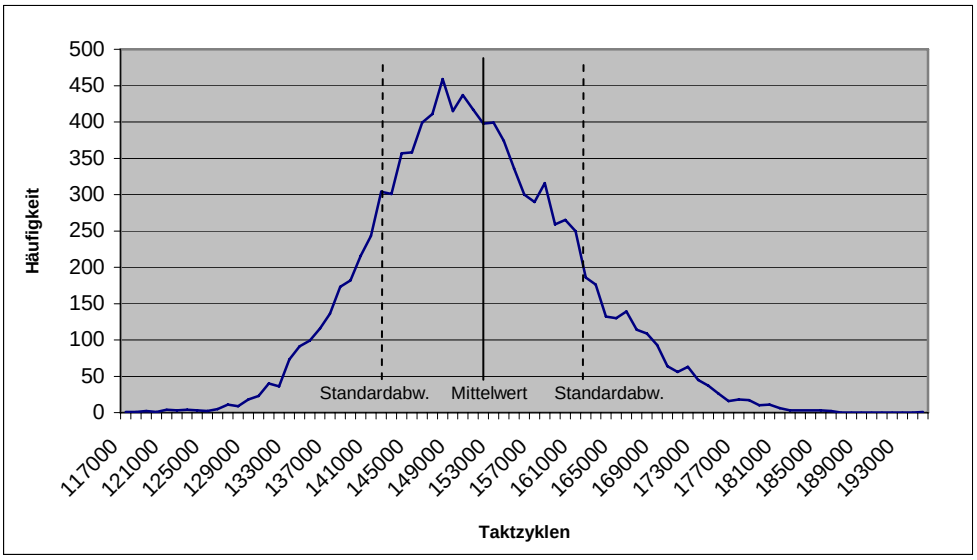


Abbildung 6-2
Häufigkeitsverteilung der von `g_makeswitch()` benötigten Taktzyklen

Die für jede Ausführung gemessene Anzahl Taktzyklen wurde auf 4 Stellen vor dem Komma gerundet. Der rechnerische Mittelwert liegt bei 152.000 Takte, der gemessene, und nicht gerundete Mittelwert aus 20 Messungen beläuft sich auf 149.754 Takte. Dies entspricht einer Abweichung von 1,5%. Die Standardabweichung liegt bei 9.716 Taktzyklen (gerundet 10.000). Somit liegen also bei einer Abweichung von 6,6% (10.000 von 152.000) um den rechnerischen Mittelwert bereits 72,3% der Messergebnis in diesem Bereich. Zur Verdeutlichung ist die untere und obere Grenze der gerundeten Standardabweichung ebenfalls eingezeichnet.

4. Der I²C IP-Core läuft fest mit einer Geschwindigkeit von 20MHz, die DLX CPU wurde mit 25MHz getaktet. Für die Funktion `g_makebar()` wurde eine Versuchsreihe mit weiteren CPU Taktraten durchgeführt. Das Ergebnis war wie zu erwarten ein linearer Anstieg – doppelte Geschwindigkeit bedeutet doppelt so viele Wartezyklen, wie aus den in Abbildung 6-3 aufgeführten Messwerten deutlich wird.

CPU Takt [MHz]	CPU-Zyklenzahl	Ausführungsdauer [msec]
25	272.035	10,88
30	322.561	10,75
33	356.522	10,80
40	433.271	10,83
45	483.976	10,76
50	541.126	10,82
55	581.919	10,58
60	634.946	10,58

Abbildung 6-3
Ausführungsdauer von `g_makebar()` bei variierender CPU Taktgeschwindigkeit

Für die in Abbildung 6-4 gezeigten Ergebnisse wurde die DLX CPU mit 25MHz betrieben, gemessen wurde jeweils automatisiert 20 Mal und das Ergebnis gemittelt.

Funktionsaufruf	CPU Zyklenzahl	Ausführungszeit [msec]
`t_print("Hallo Welt!\r\n")`	158.034	6,32
`g_print("Hallo Welt!!!", 0, 0)`	190.458	7,62
`g_drawrect(10,10,100,50)`	94.159	3,77
`g_drawline(10,10,100,50)`	96.153	3,85
`g_makebar(10,10,100,30,0,19,[i],1,0)`	200.256	8,01
`g_makebar(10,10,100,30,0,19,[i],1,1)`	272.035	10,88
`g_makeswitch("text",10,10,100,30,0,0)`	149.754	5,99

Abbildung 6-4
Ausführungsdauer ausgewählter Befehle auf der DLX CPU bei 25MHz

Anhang A Quellcode der I²C Interface Statemachine

Der folgende Code stammt aus `pca9564_interface_state.vhd` und ist für die Steuerung des Philips PCA9564 I²C Interface Chip zuständig. Alles Zeitangaben die im Quellcode enthalten sind, beziehen sich auf Angaben aus [PCA9564AN], [PCA9564] und [I²CAN]

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY pca9564_interface IS

  GENERIC (
    CLOCK_CYCLE_DURATION_NS : positive := 50);

  PORT (
    -- connections to the other entities of the IP core
    -- connction to the data_in FIFO
    fifo_data_in : IN  std_logic_vector (7 DOWNTO 0);  -- 8 bits input
    fifo_rd_ack  : IN  std_logic;       -- read was successful
    fifo_rd_en   : OUT std_logic;       -- read strobe on FIFO
    fifo_empty   : IN  std_logic;       -- the data_in FIFO is empty

    -- connection to the data_out FIFO
    fifo_data_out : OUT std_logic_vector (7 DOWNTO 0);  -- 8 bits output
    fifo_wr_ack   : IN  std_logic;      -- write was successful
    fifo_wr_en    : OUT std_logic;      -- write strobe on FIFO
    fifo_full     : IN  std_logic;      -- the data_out FIFO is full

    clock          : IN  std_logic;     -- clock signal
    reset          : IN  std_logic;     -- reset the I2C bus interface
    device_address : IN  std_logic_vector (6 DOWNTO 0);  -- address of the device to
                                        -- be accessed on the bus
    rw             : IN  std_logic;     -- select if data should be written to
                                        -- / read from the device selected with
                                        -- device_address; rw = '0' = write
    data_valid     : IN  std_logic;     -- chip enable signal, indicates that
                                        -- rw = '0': data in fifo is present,
                                        --           device_address is valid
                                        -- rw = '1': device_address is valid
    bytes_to_read  : IN  std_logic_vector (7 DOWNTO 0);  -- amount of bytes
                                                         -- to be read from bus
    done           : OUT std_logic;     -- signals that requested transfer is complete

    -- connections to the PCA9564, to be mapped to FPGA IO pins
    pca9564_data    : INOUT std_logic_vector (7 DOWNTO 0);  -- 8 bits to / from PCA9564
    pca9564_wr      : OUT   std_logic;  -- write strobe
    pca9564_rd      : OUT   std_logic;  -- read strobe
    pca9564_ce      : OUT   std_logic;  -- chip enable
    pca9564_address : OUT   std_logic_vector (1 DOWNTO 0);  -- select PCA9564
                                        -- register, format is A1A0
    pca9564_int     : IN    std_logic;  -- interrupt line from PCA9564
    pca9564_reset   : OUT   std_logic;  -- reset PCA9564

    -- connections to debug leds, to be mapped to FPGA IO pins
    debug_led_extern : OUT std_logic_vector (31 DOWNTO 0);
                                         -- external debug LED connection
    debug_led_intern : OUT std_logic_vector (7 DOWNTO 0)
                                        -- internal debug LED connection
    );

END pca9564_interface;

ARCHITECTURE pca9564_interface_arch OF pca9564_interface IS
  -- some constant definitions for easier reference
  -- the 5 register names of the PCA9564
  CONSTANT CTO : std_logic_vector (1 DOWNTO 0) := "00";
  CONSTANT STA : std_logic_vector (1 DOWNTO 0) := "00";
```

```
  CONSTANT DAT : std_logic_vector (1 DOWNTO 0) := "01";
  CONSTANT ADR : std_logic_vector (1 DOWNTO 0) := "10";
  CONSTANT CON : std_logic_vector (1 DOWNTO 0) := "11";

  -- set speed of I2C bus, this is used to overwrite the lower 3 bits
  -- of every data value to be written in the CON register
  -- "100" is 88kHz, but speed may increase up to 109kHz depending on chip temperature
  CONSTANT SPD : std_logic_vector (2 DOWNTO 0) := "100";

  SIGNAL debug_led_extern_state : std_logic_vector (31 DOWNTO 0) := X"00000000";
                                       -- state of the external debug leds
  SIGNAL debug_led_intern_state : std_logic_vector (7 DOWNTO 0)  := X"00";
                                       -- state of the internal debug leds

  SIGNAL pca9564_interrupt : std_logic := '1';
  SIGNAL done_i            : std_logic := '0';

  -- the states for the state machine
  TYPE machine IS ( init0, init1, init2, init3, init4, init5, init6, init7, init8,
                    idle0, idle1,
                    waitcycles,
                    read0, read1, read2,
                    write0, write1, write2, write3,
                    readfifo0, readfifo1, readfifo2,
                    writefifo0, writefifo1, writefifo2,
                    mt0, mt1, mt2, mt3, mt4, mt5, mt6, mt7, mt8, mt9, mt10, mt11,
                    mr0, mr1, mr2, mr3, mr4, mr5, mr6, mr7, mr8, mr9, mr10, mr11, mr12,
                    mr13, mr14, mr15, mr16,
                    panic, fail, handler);

BEGIN  -- pca9564_interface_arch

  -- register int signal because of metastability issues
  pca9564_interrupt <= pca9564_int WHEN clock'EVENT AND clock = '1';

  -- purpose: contains the state machine to interface with the PCA9564
  -- type : combinational
  interfaceToPCA9564 : PROCESS (clock, reset)
    VARIABLE dataval : std_logic_vector (7 DOWNTO 0) := X"00";  -- data variable
    VARIABLE address : std_logic_vector (1 DOWNTO 0) := "00";
                                            -- register address to write to / read from

    VARIABLE state       : machine := init0;
    VARIABLE returnstate : machine := init0;
    VARIABLE temp        : machine := init0;

    VARIABLE bytecounter    : integer := 0;  -- amount of bytes to be read
    VARIABLE cycles_to_wait : integer := 0;  -- amount of waitcycles

    -- in case of error count how often rechecked for possible error recovery
    VARIABLE errorcount_mt11 : integer := 0;
    VARIABLE errorcount_mr2  : integer := 0;
    VARIABLE errorcount_mr5  : integer := 0;
    VARIABLE errorcount_mr13 : integer := 0;

  BEGIN  -- PROCESS interfaceToPCA9564
    IF reset = '1' THEN
      state       := init0;
      returnstate := init0;
      temp        := init0;

      pca9564_ce    <= '0';
      pca9564_reset <= '1';
      pca9564_wr    <= '1';
      pca9564_rd    <= '1';

      pca9564_address <= (OTHERS => 'Z');
      pca9564_data    <= (OTHERS => 'Z');

      fifo_wr_en <= '0';
      fifo_rd_en <= '0';

      done_i <= '0';

      debug_led_extern_state <= X"FFFFFFFF";
      debug_led_intern_state <= X"99";
```

```
    ELSIF clock'EVENT AND clock = '1' THEN
      CASE state IS

-- initialize the state machine and the PCA9564
        WHEN init0 =>
          debug_led_intern_state <= X"00";
          debug_led_extern_state <= X"00000000";

          pca9564_ce    <= '0';
          pca9564_reset <= '0';
          pca9564_wr    <= '1';
          pca9564_rd    <= '1';

          pca9564_address <= (OTHERS => 'Z');
          pca9564_data    <= (OTHERS => 'Z');

          fifo_wr_en <= '0';
          fifo_rd_en <= '0';

          done_i <= '0';

          -- WAIT FOR 250 ns
          cycles_to_wait := (250 / CLOCK_CYCLE_DURATION_NS) + 1;

          -- set return variable
          returnstate := init1;
          -- set state for next turn
          state       := waitcycles;

        WHEN init1 =>
          debug_led_intern_state <= X"01";
          pca9564_reset          <= '1';
          -- timeout register CTO is 0xFF by default
          -- address register ADR is 0x00 by default

          -- set clock frequency
          dataval := X"44";
          address := CON;

          -- set return variable
          returnstate := init2;
          -- set state for next turn
          state       := write0;

        WHEN init2 =>
          debug_led_intern_state <= X"02";
          -- set address of pca9564
          dataval := X"26";
          address := ADR;

          -- set return variable
          returnstate := init3;
          -- set state for next turn
          state       := write0;

        WHEN init3 =>
          debug_led_intern_state <= X"03";
          -- WAIT FOR 500 us
          cycles_to_wait := (50000 / CLOCK_CYCLE_DURATION_NS) + 1;

          -- set return variable
          returnstate := init4;
          -- set state for next turn
          state       := waitcycles;

        WHEN init4 =>
          debug_led_intern_state <= X"04";
          -- read back register contents for debug purpose
          address := STA;

          -- set return variable
          returnstate := init5;
          -- set state for next turn
          state       := read0;

        WHEN init5 =>
          debug_led_intern_state <= X"05";
```

```
          address := DAT;

          -- set return variable
          returnstate := init6;
          -- set state for next turn
          state       := read0;

        WHEN init6 =>
          debug_led_intern_state <= X"06";
          address := ADR;

          -- set return variable
          returnstate := init7;
          -- set state for next turn
          state       := read0;

        WHEN init7 =>
          debug_led_intern_state <= X"07";
          address := CON;

          -- set return variable
          returnstate := init8;
          -- set state for next turn
          state       := read0;

        WHEN init8 =>
          debug_led_intern_state <= X"08";
          -- set slave receiver mode !?!
          address := CON;
          dataval := X"C4";

          -- set return variable
          returnstate := idle0;

          -- set state for next turn
          state := write0;

-- main idle state
        WHEN idle0 =>
          debug_led_intern_state <= X"80";
          IF data_valid = '1' AND done_i = '0' THEN
            IF rw = '0' THEN
              state       := mt0;
            ELSE
              bytecounter := conv_integer(bytes_to_read);
              state       := mr0;
            END IF;
          ELSIF data_valid = '0' AND done_i = '1' THEN
            done_i               <= '0';
          ELSE
            NULL;
          END IF;

        WHEN idle1 =>
          done_i <= '1';
          state := idle0;

        WHEN mt0 =>
          debug_led_intern_state <= X"C0";
          -- generate START command on I2C bus
          address := CON;
          dataval := X"E4";

          -- set return variable
          returnstate := mt1;
          -- set state for next turn
          state       := write0;

        WHEN mt1 =>
          debug_led_intern_state <= X"C1";
          -- wait on pca9564_interrupt = '0'
          IF pca9564_interrupt = '0' THEN
            -- read STA register
            address := STA;

            -- set return variable
            returnstate := mt2;
```

```
    -- set state for next turn
    state       := read0;
  ELSE
    state       := mt1;
  END IF;

WHEN mt2 =>
  debug_led_intern_state <= X"C2";
  IF dataval /= X"08" THEN
    -- we have an error here, go to the fault handler
    -- set return variable
    returnstate          := state;
    -- set state for next turn
    state                := handler;
  ELSE
    -- write slave address to DAT register
    address              := DAT;
    dataval (7 DOWNTO 1) := device_address;
    dataval (0)          := '0';

    -- set return variable
    returnstate := mt3;
    -- set state for next turn
    state       := write0;
  END IF;

WHEN mt3 =>
  debug_led_intern_state <= X"C3";
  -- write to CON register to send address on bus, issue write command
  address := CON;
  dataval := X"C4";

  -- set return variable
  returnstate := mt4;
  -- set state for next turn
  state       := write0;

WHEN mt4 =>
  debug_led_intern_state <= X"C4";
  -- wait on pca9564_interrupt = '0'
  IF pca9564_interrupt = '0' THEN
    -- read STA register
    address := STA;

    -- set return variable
    returnstate := mt5;
    -- set state for next turn
    state       := read0;
  ELSE
    state       := mt4;
  END IF;

WHEN mt5 =>
  debug_led_intern_state <= X"C5";
  IF dataval /= X"18" THEN
    -- we have an error here, go to the fault handler
    -- set return variable
    returnstate := state;
    -- set state for next turn
    state       := handler;
  ELSE
    -- request data from the data_in FIFO
    -- set return variable
    returnstate := mt6;
    -- set state for next turn
    state       := readfifo0;
  END IF;

WHEN mt6 =>
  debug_led_intern_state <= X"C6";
  -- write data to be sent into DAT register
  address := DAT;
  dataval := fifo_data_in;

  -- set return variable
  returnstate := mt7;
  -- set state for next turn
```

```
  state       := write0;

WHEN mt7 =>
  debug_led_intern_state <= X"C7";
  -- write CON register to send data on bus
  address := CON;
  dataval := X"C4";

  -- set return variable
  returnstate := mt8;
  -- set state for next turn
  state       := write0;

WHEN mt8 =>
  debug_led_intern_state <= X"C8";
  -- wait on pca9564_interrupt = '0'
  IF pca9564_interrupt = '0' THEN
    -- read STA register
    address := STA;

    -- set return variable
    returnstate := mt9;
    -- set state for next turn
    state       := read0;
  ELSE
    state       := mt8;
  END IF;

WHEN mt9 =>
  debug_led_intern_state <= X"C9";
  IF dataval /= X"28" THEN
    -- we have an error here, go to the fault handler
    -- set return variable
    returnstate := state;
    -- set state for next turn
    state       := handler;
  ELSE
    IF fifo_empty /= '1' THEN
      -- we have more packets to send
      -- set dataval so the IF in state mt5 succeeds
      dataval   := X"18";
      -- set state for next turn
      state     := mt5;
    ELSE
      -- generate STOP command on I2C bus
      address   := CON;
      dataval   := X"D0";

      -- set return variable
      returnstate := mt10;
      -- set state for next turn
      state       := write0;
    END IF;
  END IF;

WHEN mt10 =>
  debug_led_intern_state <= X"CA";

  -- read CON register
  address := CON;

  -- set return variable
  returnstate := mt11;
  -- set state for next turn
  state       := read0;

WHEN mt11 =>
  debug_led_intern_state <= X"CB";
  IF dataval (4) /= '0' THEN
    -- we have an error here, go to the fault handler
    -- set return variable

    returnstate     := state;
    -- set state for next turn
    state           := handler;
  ELSE
    errorcount_mt11 := 0;
```

```
            --wait 4,7 usec before next transfer on bus
            cycles_to_wait  := (4700 / CLOCK_CYCLE_DURATION_NS) + 1;

            returnstate := idle1;
            -- set state for next turn
            state       := waitcycles;
          END IF;

-- master receiver mode
        WHEN mr0 =>
          debug_led_intern_state <= X"60";
          -- generate START command on I2C bus
          address := CON;
          dataval := X"E4";

          -- set return variable
          returnstate := mr1;
          -- set state for next turn
          state       := write0;

        WHEN mr1 =>
          debug_led_intern_state <= X"61";
          -- wait on pca9564_interrupt = '0'
          IF pca9564_interrupt = '0' THEN
            -- read STA register
            address := STA;

            -- set return variable
            returnstate := mr2;
            -- set state for next turn
            state       := read0;
          ELSE
            state       := mr1;
          END IF;

        WHEN mr2 =>
          debug_led_intern_state <= X"62";
          IF dataval /= X"08" AND dataval /= X"10" THEN
            -- we have an error here, go to the fault handler
            -- set return variable
            returnstate          := state;
            -- set state for next turn
            state                := handler;
          ELSE
            errorcount_mr2       := 0;
            -- write slave address to DAT register
            address              := DAT;
            dataval (7 DOWNTO 1) := device_address;
            dataval (0)          := '1';

            -- set return variable
            returnstate := mr3;
            -- set state for next turn
            state       := write0;
          END IF;

        WHEN mr3 =>
          debug_led_intern_state <= X"63";
          -- write to CON register to send address on bus, issue read command
          address := CON;
          dataval := X"C4";

          -- set return variable
          returnstate := mr4;
          -- set state for next turn
          state       := write0;

        WHEN mr4 =>
          debug_led_intern_state <= X"64";
          -- wait on pca9564_interrupt = '0'
          IF pca9564_interrupt = '0' THEN
            -- read STA register
            address := STA;

            -- set return variable
            returnstate := mr5;
            -- set state for next turn
```

```
    state       := read0;
  ELSE
    state       := mr4;
  END IF;

WHEN mr5 =>
  debug_led_intern_state <= X"65";
  IF dataval /= X"40" THEN
    -- we have an error here, go to the fault handler
    -- set return variable
    returnstate    := state;
    -- set state for next turn
    state          := handler;
  ELSE
    errorcount_mr5 := 0;
    -- request data from the PCA9564, allow slave data transfer
    address        := CON;
    IF bytecounter > 1 THEN
      dataval      := X"C4";
    ELSE
      dataval      := X"44";
    END IF;

    -- set return variable
    returnstate := mr6;
    -- set state for next turn
    state       := write0;
  END IF;

WHEN mr6 =>
  debug_led_intern_state <= X"66";
  -- wait on pca9564_interrupt = '0'
  IF pca9564_interrupt = '0' THEN
    -- read STA register
    address := STA;

    -- set return variable
    IF bytecounter > 1 THEN
      returnstate := mr7;
    ELSE
      -- we jump the loop here and go to mr9
      returnstate := mr9;
    END IF;
    -- set state for next turn
    state         := read0;
  ELSE
    state         := mr6;
  END IF;

WHEN mr7 =>
  debug_led_intern_state <= X"67";
  IF dataval /= X"50" THEN
    -- we have an error here, go to the fault handler
    -- set return variable
    returnstate := state;
    -- set state for next turn
    state       := handler;
  ELSE
    -- read data from DAT register
    address     := DAT;

    -- set return variable
    returnstate := mr8;
    -- set state for next turn
    state       := read0;
  END IF;

WHEN mr8 =>
  debug_led_intern_state <= X"68";
  fifo_data_out          <= dataval;
  bytecounter := bytecounter - 1;
  -- set dataval so the IF in mr5 succeeds
  dataval     := X"40";

  -- set return variable
  returnstate := mr5;
  -- set state for next turn
```

```
          state       := writefifo0;

        WHEN mr9 =>
          debug_led_intern_state <= X"69";
          IF dataval /= X"58" THEN
            -- we have an error here, go to the fault handler
            -- set return variable
            returnstate := state;
            -- set state for next turn
            state       := handler;
          ELSE
            -- read data from DAT register
            address     := DAT;

            -- set return variable
            returnstate := mr10;
            -- set state for next turn
            state       := read0;
          END IF;

        WHEN mr10 =>
          debug_led_intern_state <= X"6A";
          fifo_data_out          <= dataval;

          -- set return variable
          returnstate := mr11;
          -- set state for next turn
          state       := writefifo0;

        WHEN mr11 =>
          debug_led_intern_state <= X"6B";
          -- write to CON register, generate STOP command
          address := CON;
          dataval := X"D4";

          -- set return variable
          returnstate := mr12;
          -- set state for next turn
          state       := write0;

        WHEN mr12 =>
          debug_led_intern_state <= X"6C";
          -- read CON register
          address := CON;

          -- set return variable
          returnstate := mr13;
          -- set state for next turn
          state       := read0;

        WHEN mr13 =>
          debug_led_intern_state <= X"6D";

          IF dataval (4) /= '0' THEN
            -- we have an error here, go to the fault handler
            -- set return variable
            returnstate     := state;
            -- set state for next turn
            state           := handler;
          ELSE
            errorcount_mr13 := 0;
            --wait 4.7 usec before next transfer on bus
            cycles_to_wait  := (4700 / CLOCK_CYCLE_DURATION_NS) + 1;

            -- set return variable
            returnstate := idle1;
            -- set state for next turn
            state       := waitcycles;
          END IF;

-- the helper functions
-- waiting in cycles_to_wait denoted amount of cycles
        WHEN waitcycles =>
          debug_led_intern_state <= X"40";

          IF cycles_to_wait > 1 THEN
            cycles_to_wait := cycles_to_wait - 1;
```

```
          ELSE
            cycles_to_wait := 0;
            state          := returnstate;
          END IF;

-- writing to the PCA9564
        WHEN write0 =>
          debug_led_intern_state <= X"20";
          -- save old values of return states
          temp := returnstate;

          -- check if we write to the CON register
          IF address = CON THEN
            dataval (2 DOWNTO 0) := SPD;
          END IF;

          pca9564_data    <= dataval;
          pca9564_address <= address;

          -- allow some setup time for address and dataval, thus we
          -- continue in the next state (next clock cycle)
          state := write1;

        WHEN write1 =>
          debug_led_intern_state <= X"21";
          -- change return values, so we go to write2 after waiting is done
          returnstate := write2;

          pca9564_wr <= '0';
          -- hold WR for at least 7 ns
          cycles_to_wait := (7 / CLOCK_CYCLE_DURATION_NS) + 1;

          state := waitcycles;

        WHEN write2 =>
          debug_led_intern_state <= X"22";
          pca9564_wr             <= '1';

          -- output some debug information
          CASE address IS
            WHEN STA    =>
              debug_led_extern_state (31 DOWNTO 24) <= dataval;
            WHEN DAT    =>
              debug_led_extern_state (23 DOWNTO 16) <= dataval;
            WHEN ADR    =>
              debug_led_extern_state (15 DOWNTO 8)  <= dataval;
            WHEN CON    =>
              debug_led_extern_state (7 DOWNTO 0)   <= dataval;
            WHEN OTHERS => NULL;
          END CASE;

          state := write3;

        WHEN write3                         =>
          debug_led_intern_state <= X"23";
          -- remove data from the bus
          pca9564_data           <= (OTHERS => 'Z');
          -- remove address from the bus
          pca9564_address        <= (OTHERS => 'Z');

          IF address = CON AND pca9564_interrupt = '0' THEN
            -- wait until pca9564_int is high
            state := write3;
          ELSE
            -- we return to the state following the one calling write
            state := temp;
          END IF;

-- reading from any register of the PCA9564
        WHEN read0 =>
          debug_led_intern_state <= X"10";
          -- save old values of return states
          temp := returnstate;

          pca9564_address <= address;

          -- allow some setup time for address, thus we continue in the
```

```
            -- next state (next clock cycle)
            state := read1;

          WHEN read1 =>
            debug_led_intern_state <= X"11";
            -- change return values, so we go to read2 after waiting is done
            returnstate := read2;

            pca9564_rd <= '0';
            -- hold RD for at least 17 ns
            cycles_to_wait := (17 / CLOCK_CYCLE_DURATION_NS) + 1;

            state := waitcycles;

          WHEN read2 =>
            debug_led_intern_state <= X"12";
            -- we return to the state following the one calling read after
            -- waiting is done
            returnstate := temp;

            dataval := pca9564_data;
            pca9564_rd <= '1';

            -- remove address from the bus
            pca9564_address <= (OTHERS => 'Z');

            -- output some debug information
            CASE address IS
              WHEN STA    =>
                debug_led_extern_state (31 DOWNTO 24) <= dataval;
              WHEN DAT    =>
                debug_led_extern_state (23 DOWNTO 16) <= dataval;
              WHEN ADR    =>
                debug_led_extern_state (15 DOWNTO 8)  <= dataval;
              WHEN CON    =>
                debug_led_extern_state (7 DOWNTO 0)   <= dataval;
              WHEN OTHERS => NULL;
            END CASE;

            -- allow the data bus some time to get floating again: min 17 ns
            cycles_to_wait := (17 / CLOCK_CYCLE_DURATION_NS) + 1;

            state := waitcycles;

-- reading from the data_in FIFO
          WHEN readfifo0 =>
            debug_led_intern_state <= X"E0";
            -- save old values of return states
            temp := returnstate;

            IF fifo_empty = '1' THEN
              state := panic;
            ELSE
              fifo_rd_en <= '1';

              -- change return values, so we go to readfifo1 after waiting is done
              state := readfifo1;
            END IF;

          WHEN readfifo1 =>
            debug_led_intern_state <= X"E1";
            fifo_rd_en             <= '0';

            -- wait for FIFO
            cycles_to_wait := 2;
            returnstate    := readfifo2;

            state := waitcycles;

          WHEN readfifo2 =>
            debug_led_intern_state <= X"E2";

            IF fifo_rd_ack = '1' THEN
              -- we return to the state following the one calling readfifo
              state := temp;
            ELSIF fifo_rd_ack = '0' THEN
              state := fail;
```

```
          ELSE
            state := panic;
          END IF;

-- writing to the data_out FIFO
        WHEN writefifo0 =>
          debug_led_intern_state <= X"E2";
          -- save old values of return states
          temp := returnstate;

          IF fifo_full = '1' THEN
            state := panic;
          ELSE
            -- change return values, so we go to writefifo1 after waiting is done
            state := writefifo1;
            fifo_wr_en <= '1';
          END IF;

        WHEN writefifo1 =>
          debug_led_intern_state <= X"E3";
          fifo_wr_en             <= '0';

          -- wait some time because of signal runtime through arbiter
          cycles_to_wait := 2;
          returnstate    := writefifo2;

          state := waitcycles;

        WHEN writefifo2 =>
          debug_led_intern_state <= X"E4";

          IF fifo_wr_ack = '1' THEN
            -- we return to the state following the one calling writefifo
            state := temp;
          ELSIF fifo_wr_ack = '0' THEN
            state := fail;
          ELSE
            state := panic;
          END IF;

-- the fault modes
        WHEN panic =>
          debug_led_intern_state <= X"FF";

-- signal fail with blinking of internal leds
        WHEN fail =>
          debug_led_intern_state <= X"F1";

-- error handler
        WHEN handler =>
          debug_led_intern_state <= X"F0";

          CASE returnstate IS
            WHEN mt11 =>
              IF errorcount_mt11 < 1000 THEN
                errorcount_mt11 := errorcount_mt11 + 1;
                state           := mt10;
              END IF;

            WHEN mr2 =>
              IF errorcount_mr2 < 3 THEN
                errorcount_mr2 := errorcount_mr2 + 1;
                state          := mr1;
              END IF;

            WHEN mr5 =>
              IF errorcount_mr5 < 5 THEN
                errorcount_mr5 := errorcount_mr5 + 1;
                state          := mr0;
              END IF;

            WHEN mr13 =>
              IF errorcount_mr13 < 1000 THEN
                errorcount_mr13 := errorcount_mr13 + 1;
                state           := mr12;
              END IF;
```

```
            WHEN OTHERS => NULL;
          END CASE;

        WHEN OTHERS =>
          state := panic;
      END CASE;

    END IF;
  END PROCESS interfaceToPCA9564;

  debug_led_extern <= debug_led_extern_state;
  debug_led_intern <= debug_led_intern_state;

  done <= done_i;

END pca9564_interface_arch;
```

Anhang B Quellcode des I²C Arbiter

Der folgende Code stammt aus `i2c_arbiter.vhd` und ist für die Arbitrierung der an den IP-Core angeschlossenen Geräte zuständig.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.ALL;

USE IEEE.std_logic_unsigned.ALL;

ENTITY i2c_arbiter IS

  GENERIC (
    devcount : positive := 2);          -- number of connected devices

  PORT (
    -- mutiple inputs
    -- data input / output and address input, rw-selector
    mport_data_in       : IN  std_logic_vector ((8 * devcount) - 1 DOWNTO 0);
    mport_data_out      : OUT std_logic_vector ((8 * devcount) - 1 DOWNTO 0);
    mport_address       : IN  std_logic_vector ((7 * devcount) - 1 DOWNTO 0);
    mport_rw            : IN  std_logic_vector (devcount - 1 DOWNTO 0);
                                                        -- rw = '0' = write
    mport_bytes_to_read : IN  std_logic_vector ((8 * devcount) - 1 DOWNTO 0);

    -- read and write enable signals, fifo ack and valid signal, empty and full
    -- signals of the connected FIFOs
    mport_rd_en    : OUT std_logic_vector (devcount - 1 DOWNTO 0);
    mport_wr_en    : OUT std_logic_vector (devcount - 1 DOWNTO 0);
    mport_wr_ack   : IN  std_logic_vector (devcount - 1 DOWNTO 0);
    mport_rd_valid : IN  std_logic_vector (devcount - 1 DOWNTO 0);
    mport_empty    : IN  std_logic_vector (devcount - 1 DOWNTO 0);
    mport_full     : IN  std_logic_vector (devcount - 1 DOWNTO 0);

    -- reset signal, data valid
    mport_reset      : IN std_logic_vector (devcount - 1 DOWNTO 0);
    mport_data_valid : IN std_logic_vector (devcount - 1 DOWNTO 0);

    -- status information of the ip-core
    mport_done : OUT std_logic_vector (devcount - 1 DOWNTO 0);  -- active low

    -- handshake signals / check which connected device requests access on bus
    request     : IN  std_logic_vector (devcount - 1 DOWNTO 0);
    acknowledge : OUT std_logic_vector (devcount - 1 DOWNTO 0);

    -- internal clock signal of the i2c IP core
    clock_arbiter : IN std_logic;

    -- one output
    -- data input / output and address input, rw-selector
    sport_data_in       : IN  std_logic_vector (7 DOWNTO 0);
    sport_data_out      : OUT std_logic_vector (7 DOWNTO 0);
    sport_address       : OUT std_logic_vector (6 DOWNTO 0);
    sport_rw            : OUT std_logic;  -- rw = '0' = write
    sport_bytes_to_read : OUT std_logic_vector (7 DOWNTO 0);

    -- read and write enable signals, fifo ack and valid signal, empty and full
    -- signals of the coneccted FIFOs
    sport_rd_en    : IN  std_logic;
    sport_wr_en    : IN  std_logic;
    sport_wr_ack   : OUT std_logic;
    sport_rd_valid : OUT std_logic;
    sport_empty    : OUT std_logic;
    sport_full     : OUT std_logic;

    -- reset signal, data valid
    sport_reset      : OUT std_logic;
    sport_data_valid : OUT std_logic;

    -- status information of the ip-core
```

```
    sport_done : IN std_logic           -- active low
    );

END i2c_arbiter;

ARCHITECTURE i2c_arbiter_arch OF i2c_arbiter IS

BEGIN  -- i2c_arbiter_arch

  -- purpose: select the input signal and divert it to the output (the fifos and the
PCA9564 interface
  -- type   : combinational
  select_input            : PROCESS (clock_arbiter)
    VARIABLE sel_input    : integer := 0;  -- select a input
    VARIABLE range_low_8  : integer := 0;
    VARIABLE range_high_8 : integer := 0;
    VARIABLE range_low_7  : integer := 0;
    VARIABLE range_high_7 : integer := 0;

    VARIABLE waitcount : std_logic := '1';

  BEGIN  -- PROCESS select_input
    IF clock_arbiter'EVENT AND clock_arbiter = '1' THEN
      IF mport_reset (sel_input) = '1' THEN
        -- hand reset signal through, reset acknowledge
        sport_reset             <= '1';
        acknowledge (sel_input) <= '0';

        -- pass signals through, so the connected device sees the resulting signals
        -- in reset case
        sport_address           <= "0000000";
        sport_rw                <= '0';
        sport_bytes_to_read     <= X"00";
        mport_rd_en (sel_input) <= '0';
        mport_wr_en (sel_input) <= '0';
        sport_rd_valid          <= mport_rd_valid (sel_input);
        sport_wr_ack            <= mport_wr_ack (sel_input);
        sport_empty             <= mport_empty (sel_input);
        sport_full              <= mport_full (sel_input);
        sport_data_valid        <= mport_data_valid (sel_input);
        mport_done (sel_input)  <= sport_done;

        -- wait one clock cycle before we connect the next device
        IF waitcount = '1' THEN
          waitcount := '0';
        ELSE
          waitcount := '1';

          sel_input   := sel_input + 1;
          IF sel_input >= devcount THEN
            sel_input := 0;
          END IF;
        END IF;
      ELSE
        sport_reset <= '0';

        IF request (sel_input) = '1' THEN
          acknowledge (sel_input) <= '1';

          -- compute range of used bits - do it this complicated, else ISE will
          -- fail to synthesize correctly
          range_high_8 := sel_input + 1;
          range_high_8 := range_high_8 * 8;
          range_high_8 := range_high_8 - 1;
          range_low_8  := range_high_8 - 7;

          range_high_7 := sel_input + 1;
          range_high_7 := range_high_7 * 7;
          range_high_7 := range_high_7 - 1;
          range_low_7  := range_high_7 - 6;

          -- assign signals
          sport_data_out       <= mport_data_in (range_high_8 DOWNTO range_low_8);
          mport_data_out (range_high_8 DOWNTO range_low_8) <= sport_data_in;
          sport_address        <= mport_address (range_high_7 DOWNTO range_low_7);
          sport_rw             <= mport_rw (sel_input);
          sport_bytes_to_read  <= mport_bytes_to_read (range_high_8 DOWNTO range_low_8);
```

```
          mport_rd_en (sel_input) <= sport_rd_en;
          mport_wr_en (sel_input) <= sport_wr_en;
          sport_rd_valid          <= mport_rd_valid (sel_input);
          sport_wr_ack            <= mport_wr_ack (sel_input);
          sport_empty             <= mport_empty (sel_input);
          sport_full              <= mport_full (sel_input);

          sport_reset      <= mport_reset (sel_input);
          sport_data_valid <= mport_data_valid (sel_input);

          mport_done (sel_input) <= sport_done;

        ELSE
          -- if we don't know of a pending request, we put some "reset" values
          -- on the signals
          acknowledge (sel_input) <= '0';

          sport_address       <= "0000000";
          sport_rw            <= '0';
          sport_bytes_to_read <= X"00";

          mport_rd_en (sel_input) <= '0';
          mport_wr_en (sel_input) <= '0';
          sport_rd_valid          <= mport_rd_valid (sel_input);
          sport_wr_ack            <= mport_wr_ack (sel_input);
          sport_empty             <= mport_empty (sel_input);
          sport_full              <= mport_full (sel_input);

          sport_reset      <= '0';
          sport_data_valid <= '0';

          mport_done (sel_input) <= '0';

          -- check for one device request every clock tick. We do not check all
          -- devices in one clock tick, because then the connected devices would
          -- not be treated equal. A check on all devices with a simple loop would
          -- mean daisy-chaining.
          sel_input   := sel_input + 1;
          IF sel_input = devcount THEN
            sel_input := 0;
          END IF;

        END IF;
      END IF;
    END IF;
  END PROCESS select_input;

END i2c_arbiter_arch;
```

Anhang C Quellcode des I²C Multiclock Latch

Der folgende Code stammt aus `i2c_multiclock_latch.vhd` und ist für die Pufferung der Signale am Übergang der Clock Domains zuständig.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.ALL;

USE IEEE.std_logic_unsigned.ALL;

ENTITY multiclock_latch IS

  PORT (
    -- the clocks from the two domains
    clock_1 : IN std_logic;
    clock_2 : IN std_logic;

    -- input signals, those marked with cl1 will be read from clock domain 1,
    -- cl2 from clock domain 2
    i_cl1_address       : IN std_logic_vector (6 DOWNTO 0);
    i_cl1_rw            : IN std_logic;
    i_cl1_bytes_to_read : IN std_logic_vector (7 DOWNTO 0);
    i_cl1_data_valid    : IN std_logic;
    i_cl2_done          : IN std_logic;
    i_cl1_request       : IN std_logic;
    i_cl2_acknowledge   : IN std_logic;

    -- output signals, those marked with cl1 will be written to clock domain 1,
    -- cl2 to clock domain 2
    o_cl2_address       : OUT std_logic_vector (6 DOWNTO 0) := (OTHERS => '0');
    o_cl2_rw            : OUT std_logic                     := '0';
    o_cl2_bytes_to_read : OUT std_logic_vector (7 DOWNTO 0) := (OTHERS => '0');
    o_cl2_data_valid    : OUT std_logic                     := '0';
    o_cl1_done          : OUT std_logic                     := '0';
    o_cl2_request       : OUT std_logic                     := '0';
    o_cl1_acknowledge   : OUT std_logic                     := '0'
    );

END multiclock_latch;

ARCHITECTURE multiclock_latch_arch OF multiclock_latch IS
  -- internal signals for buffering between the two clock domains
  SIGNAL address_i       : std_logic_vector (6 DOWNTO 0);
  SIGNAL rw_i            : std_logic;
  SIGNAL bytes_to_read_i : std_logic_vector (7 DOWNTO 0);
  SIGNAL data_valid_i    : std_logic;
  SIGNAL done_i          : std_logic;
  SIGNAL request_i       : std_logic;
  SIGNAL acknowledge_i   : std_logic;
BEGIN  -- multiclock_latch_arch

  -- purpose: latches the input signals on positive clock edge
  -- type   : combinational
  latch_clock_1 : PROCESS (clock_1)
  BEGIN  -- PROCESS latch
    IF clock_1'EVENT AND clock_1 = '1' THEN
      -- store the data from clock-domain clock_1
      address_i       <= i_cl1_address;
      rw_i            <= i_cl1_rw;
      bytes_to_read_i <= i_cl1_bytes_to_read;
      data_valid_i    <= i_cl1_data_valid;
      request_i       <= i_cl1_request;

      -- write sampled data from clock-domain clock_1
      o_cl1_done        <= done_i;
      o_cl1_acknowledge <= acknowledge_i;
    END IF;
  END PROCESS latch_clock_1;

  -- purpose: latches the input signals on positive clock edge
```

```
  -- type   : combinational
  latch_clock_2 : PROCESS (clock_2)
  BEGIN  -- PROCESS latch
    IF clock_2'EVENT AND clock_2 = '1' THEN
      -- store the data from clock-domain clock_2
      done_i        <= i_cl2_done;
      acknowledge_i <= i_cl2_acknowledge;

      -- write sampled data from clock-domain clock_1
      o_cl2_address       <= address_i;
      o_cl2_rw            <= rw_i;
      o_cl2_bytes_to_read <= bytes_to_read_i;
      o_cl2_data_valid    <= data_valid_i;
      o_cl2_request       <= request_i;
    END IF;
  END PROCESS latch_clock_2;

END multiclock_latch_arch;
```

Anhang D Quellcode des I²C Toplevel Moduls

Der folgende Code stammt aus `i2c_toplevel.vhd` und bildet das Interface de I²C IP-Cores. Hier werden die drei zuvor im Quellcode vorgestellten Komponenten miteinander verbunden.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.ALL;
USE IEEE.std_logic_unsigned.ALL;

LIBRARY UNISIM;
USE UNISIM.vcomponents.ALL;

ENTITY i2c_toplevel IS

  GENERIC (
    devcount : positive := 2
    );

  PORT (
    -- clock signal for the IP core, independent from the clock signal of the
    -- devices connected to the IP core
    clock_ip : IN std_logic;

    -- data input / output and address input, rw-selector
    data_in       : IN  std_logic_vector ((8 * devcount) - 1 DOWNTO 0);
    data_out      : OUT std_logic_vector ((8 * devcount) - 1 DOWNTO 0);
    address       : IN  std_logic_vector ((7 * devcount) - 1 DOWNTO 0);
    rw            : IN  std_logic_vector (devcount - 1 DOWNTO 0);  -- rw = '0' = write
    bytes_to_read : IN  std_logic_vector ((8 * devcount) - 1 DOWNTO 0);

    -- fifo control signals: read and write enable signals, empty and full
    -- signals, acknowledge and valid signals
    rd_en    : IN  std_logic_vector (devcount - 1 DOWNTO 0);
    wr_en    : IN  std_logic_vector (devcount - 1 DOWNTO 0);
    empty    : OUT std_logic_vector (devcount - 1 DOWNTO 0);
    full     : OUT std_logic_vector (devcount - 1 DOWNTO 0);
    wr_ack   : OUT std_logic_vector (devcount - 1 DOWNTO 0);
    rd_valid : OUT std_logic_vector (devcount - 1 DOWNTO 0);

    -- clock and reset signal, data valid signal
    clock      : IN std_logic_vector (devcount - 1 DOWNTO 0);
    reset      : IN std_logic_vector (devcount - 1 DOWNTO 0);
    data_valid : IN std_logic_vector (devcount - 1 DOWNTO 0);

    -- status information of the ip-core
    done : OUT std_logic_vector (devcount - 1 DOWNTO 0);  -- active low

    -- handshake signals / check which connected device requests access on bus
    request     : IN  std_logic_vector (devcount - 1 DOWNTO 0);
    acknowledge : OUT std_logic_vector (devcount - 1 DOWNTO 0);

    -- connections to the PCA9564, to be mapped to FPGA IO pins
    pca9564_data    : INOUT std_logic_vector (7 DOWNTO 0);  -- 8 bits to / from PCA9564
    pca9564_wr      : OUT   std_logic;  -- write strobe
    pca9564_rd      : OUT   std_logic;  -- read strobe
    pca9564_ce      : OUT   std_logic;  -- chip enable
    pca9564_address : OUT   std_logic_vector (1 DOWNTO 0);  -- select PCA9564
                                        -- register, format is A1A0
    pca9564_int     : IN    std_logic;  -- interrupt line from PCA9564
    pca9564_reset   : OUT   std_logic;  -- reset PCA9564

    -- connections to debug leds, to be mapped to FPGA IO pins
    debug_pca9564_registers  : OUT std_logic_vector (31 DOWNTO 0);
                                         -- external debug LED connection
    debug_statemachine_state : OUT std_logic_vector (7 DOWNTO 0)
                                         -- internal debug LED connection
    );

END i2c_toplevel;
```

```
ARCHITECTURE i2c_toplevel_arch OF i2c_toplevel IS
  -- signals to latch input signals, connect toplevel input and arbiter
  SIGNAL mport_done_i          : std_logic_vector (devcount - 1 DOWNTO 0);
  SIGNAL mport_acknowledge_i   : std_logic_vector (devcount - 1 DOWNTO 0);
  SIGNAL address_latched       : std_logic_vector ((7 * devcount) - 1 DOWNTO 0);
  SIGNAL rw_latched            : std_logic_vector (devcount - 1 DOWNTO 0);
  SIGNAL bytes_to_read_latched : std_logic_vector ((8 * devcount) - 1 DOWNTO 0);
  SIGNAL data_valid_latched    : std_logic_vector (devcount - 1 DOWNTO 0);
  SIGNAL request_latched       : std_logic_vector (devcount - 1 DOWNTO 0);

  -- signals to connect arbiter and input fifo
  SIGNAL mport_data_in_i  : std_logic_vector ((8 * devcount) - 1 DOWNTO 0);
  SIGNAL mport_rd_en_i    : std_logic_vector (devcount - 1 DOWNTO 0);
  SIGNAL mport_rd_valid_i : std_logic_vector (devcount - 1 DOWNTO 0);
  SIGNAL mport_empty_i    : std_logic_vector (devcount - 1 DOWNTO 0);

  -- signals to connect arbiter and output fifo
  SIGNAL mport_data_out_i : std_logic_vector ((8 * devcount) - 1 DOWNTO 0);
  SIGNAL mport_wr_en_i    : std_logic_vector (devcount - 1 DOWNTO 0);
  SIGNAL mport_wr_ack_i   : std_logic_vector (devcount - 1 DOWNTO 0);
  SIGNAL mport_full_i     : std_logic_vector (devcount - 1 DOWNTO 0);

  -- signals to connect arbiter and the PCA9564 interface
  SIGNAL sport_data_in_i       : std_logic_vector (7 DOWNTO 0);
  SIGNAL sport_data_out_i      : std_logic_vector (7 DOWNTO 0);
  SIGNAL sport_address_i       : std_logic_vector (6 DOWNTO 0);
  SIGNAL sport_rw_i            : std_logic;
  SIGNAL sport_bytes_to_read_i : std_logic_vector (7 DOWNTO 0);
  SIGNAL sport_rd_en_i         : std_logic;
  SIGNAL sport_wr_en_i         : std_logic;
  SIGNAL sport_wr_ack_i        : std_logic;
  SIGNAL sport_rd_valid_i      : std_logic;
  SIGNAL sport_empty_i         : std_logic;
  SIGNAL sport_full_i          : std_logic;
  SIGNAL sport_reset_i         : std_logic;
  SIGNAL sport_data_valid_i    : std_logic;
  SIGNAL sport_done_i          : std_logic;

  -- signals for clock management
  SIGNAL clock_ip_half : std_logic;
  SIGNAL dcm_clk0_bufg : std_logic;
  SIGNAL dcm_dv_bufg   : std_logic;
  SIGNAL clock_fb      : std_logic;

  COMPONENT multiclock_latch
    PORT (
      clock_1             : IN  std_logic;
      clock_2             : IN  std_logic;
      i_cl1_address       : IN  std_logic_vector (6 DOWNTO 0);
      i_cl1_rw            : IN  std_logic;
      i_cl1_bytes_to_read : IN  std_logic_vector (7 DOWNTO 0);
      i_cl1_data_valid    : IN  std_logic;
      i_cl2_done          : IN  std_logic;
      i_cl1_request       : IN  std_logic;
      i_cl2_acknowledge   : IN  std_logic;
      o_address           : OUT std_logic_vector (6 DOWNTO 0);
      o_rw                : OUT std_logic;
      o_bytes_to_read     : OUT std_logic_vector (7 DOWNTO 0);
      o_data_valid        : OUT std_logic;
      o_done              : OUT std_logic;
      o_request           : OUT std_logic;
      o_acknowledge       : OUT std_logic);
  END COMPONENT;

  COMPONENT pca9564_interface
    PORT (
      fifo_data_in     : IN    std_logic_vector (7 DOWNTO 0);
      fifo_rd_ack      : IN    std_logic;
      fifo_rd_en       : OUT   std_logic;
      fifo_empty       : IN    std_logic;
      fifo_data_out    : OUT   std_logic_vector (7 DOWNTO 0);
      fifo_wr_ack      : IN    std_logic;
      fifo_wr_en       : OUT   std_logic;
      fifo_full        : IN    std_logic;
      clock            : IN    std_logic;
      reset            : IN    std_logic;
```

```
      device_address   : IN    std_logic_vector (6 DOWNTO 0);
      rw               : IN    std_logic;
      data_valid       : IN    std_logic;
      bytes_to_read    : IN    std_logic_vector (7 DOWNTO 0);
      done             : OUT   std_logic;
      pca9564_data     : INOUT std_logic_vector (7 DOWNTO 0);
      pca9564_wr       : OUT   std_logic;
      pca9564_rd       : OUT   std_logic;
      pca9564_ce       : OUT   std_logic;
      pca9564_address  : OUT   std_logic_vector (1 DOWNTO 0);
      pca9564_int      : IN    std_logic;
      pca9564_reset    : OUT   std_logic;
      debug_led_extern : OUT   std_logic_vector (31 DOWNTO 0);
      debug_led_intern : OUT   std_logic_vector (7 DOWNTO 0));
  END COMPONENT;

  COMPONENT i2c_arbiter
    GENERIC (
      devcount            :       positive);
    PORT (
      mport_data_in       : IN  std_logic_vector ((8 * devcount) - 1 DOWNTO 0);
      mport_data_out      : OUT std_logic_vector ((8 * devcount) - 1 DOWNTO 0);
      mport_address       : IN  std_logic_vector ((7 * devcount) - 1 DOWNTO 0);
      mport_rw            : IN  std_logic_vector (devcount - 1 DOWNTO 0);
      mport_bytes_to_read : IN  std_logic_vector ((8 * devcount) - 1 DOWNTO 0);
      mport_rd_en         : OUT std_logic_vector (devcount - 1 DOWNTO 0);
      mport_wr_en         : OUT std_logic_vector (devcount - 1 DOWNTO 0);
      mport_wr_ack        : IN  std_logic_vector (devcount - 1 DOWNTO 0);
      mport_rd_valid      : IN  std_logic_vector (devcount - 1 DOWNTO 0);
      mport_empty         : IN  std_logic_vector (devcount - 1 DOWNTO 0);
      mport_full          : IN  std_logic_vector (devcount - 1 DOWNTO 0);
      mport_reset         : IN  std_logic_vector (devcount - 1 DOWNTO 0);
      mport_data_valid    : IN  std_logic_vector (devcount - 1 DOWNTO 0);
      mport_done          : OUT std_logic_vector (devcount - 1 DOWNTO 0);
      request             : IN  std_logic_vector (devcount - 1 DOWNTO 0);
      acknowledge         : OUT std_logic_vector (devcount - 1 DOWNTO 0);
      clock_arbiter       : IN  std_logic;
      sport_data_in       : IN  std_logic_vector (7 DOWNTO 0);
      sport_data_out      : OUT std_logic_vector (7 DOWNTO 0);
      sport_address       : OUT std_logic_vector (6 DOWNTO 0);
      sport_rw            : OUT std_logic;
      sport_bytes_to_read : OUT std_logic_vector (7 DOWNTO 0);
      sport_rd_en         : IN  std_logic;
      sport_wr_en         : IN  std_logic;
      sport_wr_ack        : OUT std_logic;
      sport_rd_valid      : OUT std_logic;
      sport_empty         : OUT std_logic;
      sport_full          : OUT std_logic;
      sport_reset         : OUT std_logic;
      sport_data_valid    : OUT std_logic;
      sport_done          : IN  std_logic);
  END COMPONENT;

  COMPONENT fifo_generator_v2_1
    PORT (
      din    : IN  std_logic_vector(7 DOWNTO 0);
      rd_clk : IN  std_logic;
      rd_en  : IN  std_logic;
      rst    : IN  std_logic;
      wr_clk : IN  std_logic;
      wr_en  : IN  std_logic;
      dout   : OUT std_logic_vector(7 DOWNTO 0);
      empty  : OUT std_logic;
      full   : OUT std_logic;
      valid  : OUT std_logic;
      wr_ack : OUT std_logic);
  END COMPONENT;

  COMPONENT BUFG
    PORT (O : OUT std_ulogic;
          I : IN  std_ulogic);
  END COMPONENT;

BEGIN  -- i2c_toplevel_arch

  input_fifo          : FOR k IN 1 TO devcount GENERATE
    input_fifo_asynch : fifo_generator_v2_1
```

```
    PORT MAP (
      din    => data_in ( ((k * 8) - 1) DOWNTO ((k * 8) - 8)),
      rd_clk => clock_ip_half,
      rd_en  => mport_rd_en_i (k - 1),
      rst    => reset (k - 1),        -- toplevel extern
      wr_clk => clock (k - 1),        -- toplevel extern
      wr_en  => wr_en (k - 1),        -- toplevel extern
      dout   => mport_data_in_i ( ((k * 8) - 1) DOWNTO ((k * 8) - 8)),
      empty  => mport_empty_i (k - 1),
      full   => full (k - 1),         -- toplevel extern
      valid  => mport_rd_valid_i (k - 1),
      wr_ack => wr_ack (k - 1)        -- toplevel extern
      );
END GENERATE input_fifo;

output_fifo          : FOR k IN 1 TO devcount GENERATE
  output_fifo_asynch : fifo_generator_v2_1
    PORT MAP (
      din    => mport_data_out_i ( ((k * 8) - 1) DOWNTO ((k * 8) - 8)),
      rd_clk => clock (k - 1),        -- toplevel extern
      rd_en  => rd_en (k - 1),        -- toplevel extern
      rst    => reset (k - 1),        -- toplevel extern
      wr_clk => clock_ip_half,
      wr_en  => mport_wr_en_i (k - 1),
      dout   => data_out ( ((k * 8) - 1) DOWNTO ((k * 8) - 8)),
      empty  => empty (k - 1),        -- toplevel extern
      full   => mport_full_i (k - 1),
      valid  => rd_valid (k - 1),     -- toplevel extern
      wr_ack => mport_wr_ack_i (k - 1));
END GENERATE output_fifo;

multiclock         : FOR k IN 1 TO devcount GENERATE
  clock_transition : multiclock_latch
    PORT MAP (
      clock_1             => clock (k - 1),  -- toplevel extern
      clock_2             => clock_ip_half,
      i_cl1_address       => address ( ((k * 7) - 1) DOWNTO ((k * 7) - 7)),
                                       -- toplevel extern
      i_cl1_rw            => rw (k - 1),  -- toplevel extern
      i_cl1_bytes_to_read => bytes_to_read ( ((k * 8) - 1) DOWNTO ((k * 8) - 8)),
                                       -- toplevel extern
      i_cl1_data_valid    => data_valid (k - 1),  -- toplevel extern
      i_cl2_done          => mport_done_i (k - 1),
      i_cl1_request       => request (k - 1),  -- toplevel extern
      i_cl2_acknowledge   => mport_acknowledge_i (k - 1),
      o_address           => address_latched ( ((k * 7) - 1) DOWNTO ((k * 7) - 7)),
      o_rw                => rw_latched (k - 1),
      o_bytes_to_read  => bytes_to_read_latched ( ((k * 8) - 1) DOWNTO ((k * 8) - 8)),
      o_data_valid        => data_valid_latched (k - 1),
      o_done              => done (k - 1),  -- toplevel extern
      o_request           => request_latched (k - 1),
      o_acknowledge       => acknowledge (k - 1));  -- toplevel extern
END GENERATE multiclock;

arbiter : i2c_arbiter
  GENERIC MAP (
    devcount            => devcount)
  PORT MAP (
    mport_data_in       => mport_data_in_i,
    mport_data_out      => mport_data_out_i,
    mport_address       => address_latched,
    mport_rw            => rw_latched,
    mport_bytes_to_read => bytes_to_read_latched,
    mport_rd_en         => mport_rd_en_i,
    mport_wr_en         => mport_wr_en_i,
    mport_wr_ack        => mport_wr_ack_i,
    mport_rd_valid      => mport_rd_valid_i,
    mport_empty         => mport_empty_i,
    mport_full          => mport_full_i,
    mport_reset         => reset,     -- toplevel extern
    mport_data_valid    => data_valid_latched,
    mport_done          => mport_done_i,
    request             => request_latched,
    acknowledge         => mport_acknowledge_i,
    clock_arbiter       => clock_ip_half,
    sport_data_in       => sport_data_in_i,
    sport_data_out      => sport_data_out_i,
```

```
      sport_address       => sport_address_i,
      sport_rw            => sport_rw_i,
      sport_bytes_to_read => sport_bytes_to_read_i,
      sport_rd_en         => sport_rd_en_i,
      sport_wr_en         => sport_wr_en_i,
      sport_wr_ack        => sport_wr_ack_i,
      sport_rd_valid      => sport_rd_valid_i,
      sport_empty         => sport_empty_i,
      sport_full          => sport_full_i,
      sport_reset         => sport_reset_i,
      sport_data_valid    => sport_data_valid_i,
      sport_done          => sport_done_i);

  pca9564 : pca9564_interface
    PORT MAP (
      fifo_data_in     => sport_data_out_i,
      fifo_rd_ack      => sport_rd_valid_i,
      fifo_rd_en       => sport_rd_en_i,
      fifo_empty       => sport_empty_i,
      fifo_data_out    => sport_data_in_i,
      fifo_wr_ack      => sport_wr_ack_i,
      fifo_wr_en       => sport_wr_en_i,
      fifo_full        => sport_full_i,
      clock            => clock_ip_half,
      reset            => sport_reset_i,
      device_address   => sport_address_i,
      rw               => sport_rw_i,
      data_valid       => sport_data_valid_i,
      bytes_to_read    => sport_bytes_to_read_i,
      done             => sport_done_i,
      pca9564_data     => pca9564_data,
      pca9564_wr       => pca9564_wr,
      pca9564_rd       => pca9564_rd,
      pca9564_ce       => pca9564_ce,
      pca9564_address  => pca9564_address,
      pca9564_int      => pca9564_int,
      pca9564_reset    => pca9564_reset,
      debug_led_extern => debug_pca9564_registers,
      debug_led_intern => debug_statemachine_state);

  bufg_clock_ip : BUFG
    PORT MAP (O => clock_ip_half,
              I => dcm_dv_bufg);

  bufg_clock : BUFG
    PORT MAP (O => clock_fb,
              I => dcm_clk0_bufg);

  DCM_toplevel : DCM
    -- The following generics are only necessary if you wish to change the default
behavior.
    GENERIC MAP (
      CLKDV_DIVIDE          => 2.0,     -- Divide by: 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5,
                                           5.0, 5.5, 6.0, 6.5
      CLK_FEEDBACK          => "1X",    -- Specify clock feedback of NONE, 1X or 2X
      DESKEW_ADJUST         => "SYSTEM_SYNCHRONOUS",  -- SOURCE_SYNCHRONOUS,
                                            SYSTEM_SYNCHRONOUS or an integer from 0 to 15
      DFS_FREQUENCY_MODE    => "LOW",   -- HIGH or LOW frequency mode for frequency
                                           synthesis
      DLL_FREQUENCY_MODE    => "LOW",   -- HIGH or LOW frequency mode for DLL
      DUTY_CYCLE_CORRECTION => TRUE,    -- Duty cycle correction, TRUE or FALSE
      STARTUP_WAIT          => FALSE)   -- Delay configuration DONE until DCM LOCK,
                                           TRUE/FALSE
    PORT MAP (
      CLK0                  => dcm_clk0_bufg,  -- 0 degree DCM CLK ouptput
      CLKDV                 => dcm_dv_bufg,  -- Divided DCM CLK out (CLKDV_DIVIDE)
      CLKFB                 => clock_fb,  -- DCM clock feedback
      CLKIN                 => clock_ip  -- Clock input (from IBUFG, BUFG or DCM)
      );
  -- End of DCM_inst instantiation

END i2c_toplevel_arch;
```

Anhang E Quellcode der ASIPMeister zu I²C Verbindung

Der folgende Code stammt aus `interface_asipmeister_i2c.vhd` und dient einer einfachen Verbindung der ASIPMeister DLX CPU mit dem I²C IP-Core,

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY interface_asipmeister_i2c IS

  PORT (
    clock : IN std_logic;
    reset : IN std_logic;

    -- connections to the ASIPMeister CPU
    asip_data_in  : IN  std_logic_vector (7 DOWNTO 0);
    asip_data_out : OUT std_logic_vector (7 DOWNTO 0);

    asip_rd_en    : IN  std_logic;
    asip_wr_en    : IN  std_logic;
    asip_empty    : OUT std_logic;
    asip_full     : OUT std_logic;
    asip_wr_ack   : OUT std_logic;
    asip_rd_valid : OUT std_logic;

    -- connections to the I2C interface core
    i2c_bus_data_in   : IN    std_logic_vector (7 DOWNTO 0);
    i2c_bus_data_out  : OUT   std_logic_vector (7 DOWNTO 0);
    i2c_address       : OUT   std_logic_vector (6 DOWNTO 0);
    i2c_rw            : INOUT std_logic;
    i2c_bytes_to_read : OUT   std_logic_vector (7 DOWNTO 0);

    i2c_rd_en    : OUT std_logic;
    i2c_wr_en    : OUT std_logic;
    i2c_empty    : IN  std_logic;
    i2c_full     : IN  std_logic;
    i2c_wr_ack   : IN  std_logic;
    i2c_rd_valid : IN  std_logic;

    i2c_data_valid : OUT std_logic;
    i2c_done       : IN  std_logic;

    i2c_request     : OUT std_logic;
    i2c_acknowledge : IN  std_logic;
    );
END interface_asipmeister_i2c;

ARCHITECTURE interface_asipmeister_i2c_arch OF interface_asipmeister_i2c IS

  TYPE machine_state IS (idle, rd_bytes, rd_addr_rw, rd_bytes_to_read, write_data,
                         req_i2c, wait_i2c, read_data);

  SIGNAL fifo_empty_i    : std_logic;
  SIGNAL fifo_rd_en_i    : std_logic;
  SIGNAL fifo_data_out_i : std_logic_vector(7 DOWNTO 0);
  SIGNAL fifo_valid_i    : std_logic;

  COMPONENT fifo_generator_v2_1
    PORT (
      din    : IN  std_logic_vector(7 DOWNTO 0);
      rd_clk : IN  std_logic;
      rd_en  : IN  std_logic;
      rst    : IN  std_logic;
      wr_clk : IN  std_logic;
      wr_en  : IN  std_logic;
      dout   : OUT std_logic_vector(7 DOWNTO 0);
      empty  : OUT std_logic;
      full   : OUT std_logic;
```

```
      valid  : OUT std_logic;
      wr_ack : OUT std_logic);
  END COMPONENT;

BEGIN  -- interface_asipmeister_i2c_arch

  input_fifo : fifo_generator_v2_1
    PORT MAP (
      din    => asip_data_in,           -- external
      rd_clk => clock,                  -- external
      rd_en  => fifo_rd_en_i,
      rst    => reset,                  -- external
      wr_clk => clock,                  -- external
      wr_en  => asip_wr_en,             -- external
      dout   => fifo_data_out_i,
      empty  => fifo_empty_i,
      full   => asip_full,              -- external
      valid  => fifo_valid_i,
      wr_ack => asip_wr_ack);           -- external

  PROCESS (clock, reset)
    VARIABLE state         : machine_state := idle;
    VARIABLE bytes_to_read : integer       := 0;

  BEGIN  -- PROCESS
    IF reset = '1' THEN                      -- asynchronous reset (active low)
      i2c_address           <= "0000000";
      i2c_bus_data_out      <= X"00";
      i2c_rw                <= '0';
      i2c_bytes_to_read     <= X"00";
      i2c_wr_en             <= '0';
      i2c_data_valid        <= '0';
      i2c_request           <= '0';
      state := idle;
    ELSIF clock'EVENT AND clock = '1' THEN  -- rising clock edge
      CASE state IS
        WHEN idle =>
          -- do some cleanup
          i2c_address       <= "0000000";
          i2c_bus_data_out  <= X"00";
          i2c_rw            <= '0';
          i2c_bytes_to_read <= X"00";
          i2c_wr_en         <= '0';
          i2c_data_valid    <= '0';
          i2c_request       <= '0';

          -- wait for data to be sent
          IF fifo_empty_i = '0' THEN
            state := rd_bytes;
          END IF;

        WHEN rd_bytes =>
          IF fifo_rd_en_i = '1' THEN
            fifo_rd_en_i     <= '0';
          ELSE
            IF fifo_valid_i = '0' THEN
              -- no valid data at fifo_data_out - so we start another read
              IF fifo_empty_i = '0' THEN
                fifo_rd_en_i <= '1';
              END IF;

            ELSIF fifo_valid_i = '1' THEN
              -- read amount of bytes to read from fifo
              bytes_to_read := conv_integer(fifo_data_out_i);
              state         := rd_addr_rw;
            END IF;
          END IF;

        WHEN rd_addr_rw =>
          IF fifo_rd_en_i = '1' THEN
            fifo_rd_en_i     <= '0';
          ELSE
            IF fifo_valid_i = '0' THEN
              -- no valid data at fifo_data_out - so we start another read
              IF fifo_empty_i = '0' THEN
                fifo_rd_en_i <= '1';
              END IF;
```

```
    ELSIF fifo_valid_i = '1' THEN
      -- read address and rw bit
      i2c_address <= fifo_data_out_i (7 DOWNTO 1);
      i2c_rw      <= fifo_data_out_i (0);

      -- check if read or write operation is requested
      IF fifo_data_out_i (0) = '1' THEN
        state := rd_bytes_to_read;
      ELSIF fifo_data_out_i (0) = '0' THEN
        state := write_data;
      END IF;
    END IF;
  END IF;

WHEN rd_bytes_to_read =>
  IF fifo_rd_en_i = '1' THEN
    fifo_rd_en_i     <= '0';
  ELSE
    IF fifo_valid_i = '0' THEN
      -- no valid data at fifo_data_out - so we start another read
      IF fifo_empty_i = '0' THEN
        fifo_rd_en_i <= '1';
      END IF;

    ELSIF fifo_valid_i = '1' THEN
      i2c_bytes_to_read <= fifo_data_out_i;
      i2c_data_valid    <= '1';

      state := req_i2c;

    END IF;
  END IF;

WHEN write_data =>
  IF fifo_rd_en_i = '1' THEN
    fifo_rd_en_i <= '0';
  ELSE
    IF bytes_to_read > 0 THEN

      IF fifo_valid_i = '0' THEN
        i2c_wr_en      <= '0';
        -- no valid data at fifo_data_out - so we start another read
        IF fifo_empty_i = '0' THEN
          fifo_rd_en_i <= '1';
        END IF;

      ELSIF fifo_valid_i = '1' THEN
        -- we have valid data at fifo_data_out - write it to I2C IP Core
        i2c_wr_en        <= '1';
        i2c_bus_data_out <= fifo_data_out_i;

        bytes_to_read := bytes_to_read - 1;

      END IF;

    ELSIF bytes_to_read = 0 THEN
      i2c_wr_en <= '0';
      i2c_data_valid <= '1';
      state := req_i2c;
    END IF;

  END IF;


WHEN req_i2c =>
  -- request access to I2C bus
  i2c_request     <= '1';
  IF i2c_acknowledge = '1' THEN
    state := wait_i2c;
  END IF;

WHEN wait_i2c =>
  IF i2c_done = '1' THEN
    i2c_data_valid <= '0';
    i2c_request    <= '0';
```

```
            -- check if there is yet data to be transferred from the I2C core to the
            -- ASIPMeister CPU
            IF i2c_rw = '1' THEN
              state := read_data;
            ELSIF i2c_rw = '0' THEN
              state := idle;
            END IF;
          END IF;

        WHEN read_data =>
          interface_state <= X"8";
          -- wait until the output fifo of the I2C core is empty
          IF i2c_empty = '1' THEN
            state := idle;
          END IF;

        WHEN OTHERS => NULL;
      END CASE;
    END IF;
  END PROCESS;

  -- line through the I2C output fifo signals
  asip_data_out <= i2c_bus_data_in;
  asip_empty    <= i2c_empty;
  asip_rd_valid <= i2c_rd_valid;
  i2c_rd_en     <= asip_rd_en;


END interface_asipmeister_i2c_arch;
```

Anhang F Quellcode der LCD Library

Der folgende Code stammt aus `lib_lcd_c`. Es handelt sich um eine Sammlung von Funktionen um die Funktionalität des Displays aus einem C-Programm heraus nutzen zu können.

```
#include "loadStoreByte.h" // function to support byte reading / writing on ASIPMeister
                           // DLX CPU

#define LCD_ADDR 111

asm void PUTC_SINAS (int c)
{
@[
.barrier
]

  nop
  nop
  nop
  putc @{c}
  nop
  nop
  nop
}

asm int GETC_SINAS ()
{
@[
.barrier
]

  nop
  nop
  nop
  getc @{}
  nop
  nop
  nop
}

/* ------------------------
   some stuff to help ...
   ------------------------ */

int strlength (const char* str) {
  char count = 0;

  // use loadByteUnsigned function to prevent usage of DLX CPU lb instruction
  while (loadByteUnsigned(str++) != 0) count++;

  return count;
}

// burn some CPU cycles
int wait (int waitvalue) {
  int count = 0;
  while (count++ != waitvalue);

  return --count;
}

// read acknowledge from display
int getack (int address) {
  PUTC_SINAS(0);
  PUTC_SINAS(++address);
  PUTC_SINAS(1);

  if (GETC_SINAS == 15) return 1;
  else return 0;
}
```

```
int checkbuffer () {
  int value = 0;
  int checksum;
  int length = 0;
  int bytes_ready = 0;
  int bytes_free = 0;
  int bcc = 0;
  const adr = LCD_ADDR << 1 ;

  // send command
  PUTC_SINAS(4);
  PUTC_SINAS(adr);
  PUTC_SINAS(18); // DC2
  checksum = 18;
  value = 1;
  PUTC_SINAS(1);
  checksum += value;
  value = 73;
  PUTC_SINAS(73); // I
  checksum += value;
  PUTC_SINAS(checksum);

  checksum = -1;
  while (checksum == -1) {
    // fetch buffer information
    PUTC_SINAS(0);
    PUTC_SINAS(adr + 1);
    PUTC_SINAS(6);

    value = GETC_SINAS();  // ACK
    value = GETC_SINAS();  // DC2
    checksum = value;
    length = GETC_SINAS(); // length
    checksum += length;
    bytes_ready = GETC_SINAS();   // data
    checksum += bytes_ready;
    bytes_free = GETC_SINAS();   // data
    checksum += bytes_free;
    bcc = GETC_SINAS();    // bcc

    // check for transmission errors (SmallProtokoll)
    checksum = checksum & 0xFF;
    if (checksum != bcc) {
      checksum = -1;  // loop runs again

      // rerequest last data package
      PUTC_SINAS(4);
      PUTC_SINAS(adr);
      PUTC_SINAS(18); // DC2
      checksum = 18;
      value = 1;
      PUTC_SINAS(1);
      checksum += value;
      value = 82;
      PUTC_SINAS(82); // R
      checksum += value;
      PUTC_SINAS(checksum);
    }
  }

  return bytes_ready;
}

// read specified amount of data from display, returns amount of bytes read
int getbytes (char* dest, int bytes_to_read) {
  int bytes_read = 0;
  int value;
  int checksum;
  int length = 0;
  int bcc = 0;
  const adr = LCD_ADDR << 1 ;

  // send command
  PUTC_SINAS(4);
  PUTC_SINAS(adr);
  PUTC_SINAS(18); // DC2
```

```
  checksum = 18;
  value = 1;
  PUTC_SINAS(1);
  checksum += value;
  value = 83;
  PUTC_SINAS(83); // S
  checksum += value;
  PUTC_SINAS(checksum);

  checksum = -1;
  while (checksum == -1) {
    // fetch buffer data
    PUTC_SINAS(0);
    PUTC_SINAS(adr + 1);
    PUTC_SINAS(bytes_to_read + 4);

    value = GETC_SINAS();  // ACK
    value = GETC_SINAS();  // DC1
    checksum = value;
    length = GETC_SINAS(); // length
    checksum += length;
    while (bytes_read != bytes_to_read) {
      value = GETC_SINAS();
      // use storeByte function to prevent usage of DLX CPU sb instruction
      storeByte(dest + bytes_read, value);
      bytes_read++;
      checksum += value;
    }
    bcc = GETC_SINAS();    // bcc

    // check for transmission errors (SmallProtokoll)
    checksum = checksum & 0xFF;
    if (checksum != bcc) {
      checksum = -1;  // loop runs again

      // rerequest last data package
      PUTC_SINAS(4);
      PUTC_SINAS(adr);
      PUTC_SINAS(18); // DC2
      checksum = 18;
      value = 1;
      PUTC_SINAS(1);
      checksum += value;
      value = 82;
      PUTC_SINAS(82); // R
      checksum += value;
      PUTC_SINAS(checksum);
    }
  }

  return bytes_read;
}

/* ------------------------
   Send individual commands
   ------------------------ */

int sendcommand (const char cmd0, const char cmd1, const int options[], const char
text[], int intcount, int charcount, int address) {
  int checksum;
  int* options_ptr;
  char* text_ptr;
  int intcount_local;
  int charcount_local;

  // amount of control characters
  const int ctrl_chars = 3;
  char value;

  address <<= 1;

  do {
    options_ptr = options;
    text_ptr = text;
    intcount_local = intcount;
    charcount_local = charcount;
```

```
    // length + start byte, data length byte, newline character,
    // carriage return character and checksum
    PUTC_SINAS(intcount + charcount + ctrl_chars + 3);
    PUTC_SINAS(address);
    PUTC_SINAS(17); // DC1
    checksum = 17;
    PUTC_SINAS(intcount + charcount + ctrl_chars);
    checksum += intcount + charcount + ctrl_chars;

    // now we put special code characters here
    PUTC_SINAS(27); // ESC
    checksum += 27;
    PUTC_SINAS(cmd0);
    checksum += cmd0;
    PUTC_SINAS(cmd1);
    checksum += cmd1;

    while (intcount_local-- != 0) {
      value = (char)(*options_ptr++);
      PUTC_SINAS(value);
      checksum += value;
    }

    while (charcount_local-- != 0) {
      // use loadByteUnsigned function to prevent usage of DLX CPU lb instruction
      value = loadByteUnsigned(text_ptr++);
      PUTC_SINAS(value);
      checksum += value;
    }

    PUTC_SINAS(checksum);
  }
  while (getack(address) != 0);

  return 0;
}

/* -------------------------------------
   Convenience functions in Terminal Mode
   ------------------------------------- */

// print text at current cursor position
int t_print (const char* str) {
  sendcommand('Z', 'T', 0, str, 0, strlength(str), LCD_ADDR);

  return 0;
}

int t_cursor (int onoff) {
  sendcommand('T', 'C', &onoff, 0, 1, 0, LCD_ADDR);

  return 0;
}

int t_enable (int onoff) {
  if (onoff) sendcommand('T', 'E', 0, 0, 0, 0, LCD_ADDR);
  else sendcommand('T', 'A', 0, 0, 0, 0, LCD_ADDR);

  return 0;
}

/* -------------------------------------
   Convenience functions in Graphic Mode
   ------------------------------------- */

int g_print (const char* str, int x, int y) {
  int args[2];

  args[0] = x;
  args[1] = y;

  sendcommand('Z', 'L', args, str, 2, strlength(str) + 1, LCD_ADDR); // print string
graphically

  return 0;
}
```

```
int g_drawrect (int x1, int y1, int x2, int y2) {
  int args[4];

  args[0] = x1;
  args[1] = y1;
  args[2] = x2;
  args[3] = y2;

  sendcommand('G', 'R', args, 0, 4, 0, LCD_ADDR); // draw a rectangle

  return 0;
}

int g_drawline (int x1, int y1, int x2, int y2) {
  int args[4];

  args[0] = x1;
  args[1] = y1;
  args[2] = x2;
  args[3] = y2;

  sendcommand('G', 'D', args, 0, 4, 0, LCD_ADDR); // draw a line

  return 0;
}

int g_makebar (int x1, int y1, int x2, int y2, int low_val, int high_val, int init_val,
int type, int fill_type, int touch) {
  static int barcounter = 1;
  int args[9];

  if (barcounter < 32) {
    args[0] = barcounter;  // select internal number for bar
    args[1] = x1;
    args[2] = y1;
    args[3] = x2;
    args[4] = y2;
    args[5] = low_val;       // low value of bar
    args[6] = high_val;      // high value of bar
    args[7] = type;          // type of bar (filled, line ...)
    args[8] = fill_type;     // look of bar (line-breadth ...)

    sendcommand('B', 'R', args, 0, 9, 0, LCD_ADDR); // draw bargraph

    if (touch != 0) {
      args[0] = barcounter;
      sendcommand('A', 'B', args, 0, 1, 0, LCD_ADDR); // bargraph is touch
    }

    args[0] = barcounter++;
    args[1] = init_val;
    sendcommand('B', 'A', args, 0, 2, 0, LCD_ADDR); // bargraph is set to init_val

    return barcounter - 1;
  }
  else return -1;
}

int g_setbar (int barnum, int value) {
  int args[2];

  args[0] = barnum;
  args[1] = value;
  sendcommand('B', 'A', args, 0, 2, 0, LCD_ADDR); // bargraph is set to value

  return 0;
}


int g_makeswitch (const char* str, int x1, int y1, int x2, int y2, int down, int up) {
  int args[6];

  args[0] = x1;
  args[1] = y1;
  args[2] = x2;
  args[3] = y2;
  args[4] = down;
```

```
  args[5] = up;
  sendcommand('A', 'K', args, str, 6, strlength(str) + 1 , LCD_ADDR); // define touch
switch

  return 0;
}

int g_makemenubutton (const char* str, int x1, int y1, int x2, int y2, int down, int up,
int select, int space) {
  int args[7];

  args[0] = x1;
  args[1] = y1;
  args[2] = x2;
  args[3] = y2;
  args[4] = down;
  args[5] = up;
  args[6] = select;
  sendcommand('A', 'M', args, str, 7, strlength(str) + 1, LCD_ADDR); // define
menubutton

  if (space > 0) {
    sendcommand('N', 'Y', &space, 0, 1, 0, LCD_ADDR); // define space inbetween menu
items
  }

  return 0;
}

int g_makeradiogroup(int group_number) {
  sendcommand('A', 'R', &group_number, 0, 1, 0, LCD_ADDR); // define radiogroup

  return 0;
}


/* ----------------
   Display Commands
   ---------------- */

int d_clear() {
  sendcommand('D', 'L', 0, 0, 0, 0, LCD_ADDR);  // clear display

  return 0;
}
```

Anhang G Constraints für die Synthese unter Xilinx ISE

Hier werden die .ucf Constraints Files aufgeführt, die unter Xilinx ISE notwendig sind um die Signale des I²C IP-Cores nach außen an die I/O Pins zu führen.

Xilinx HW-AFX-FF1152-200 Prototyping Board:

```
NET "clock_ip"  LOC = "AF18" ;
NET "pca9564_address<0>"  LOC = "AJ23" | IOSTANDARD = LVCMOS33  | SLEW = FAST  | PULLUP;
NET "pca9564_address<1>"  LOC = "AL22" | IOSTANDARD = LVCMOS33  | SLEW = FAST  | PULLUP;
NET "pca9564_ce"  LOC = "AH23" | IOSTANDARD = LVCMOS33  | SLEW = FAST  | PULLUP;
NET "pca9564_data<0>"  LOC = "AN22" | IOSTANDARD = LVCMOS33  | SLEW = FAST  | PULLUP;
NET "pca9564_data<1>"  LOC = "AL23" | IOSTANDARD = LVCMOS33  | SLEW = FAST  | PULLUP;
NET "pca9564_data<2>"  LOC = "AK22" | IOSTANDARD = LVCMOS33  | SLEW = FAST  | PULLUP;
NET "pca9564_data<3>"  LOC = "AJ22" | IOSTANDARD = LVCMOS33  | SLEW = FAST  | PULLUP;
NET "pca9564_data<4>"  LOC = "AH22" | IOSTANDARD = LVCMOS33  | SLEW = FAST  | PULLUP;
NET "pca9564_data<5>"  LOC = "AG22" | IOSTANDARD = LVCMOS33  | SLEW = FAST  | PULLUP;
NET "pca9564_data<6>"  LOC = "AF22" | IOSTANDARD = LVCMOS33  | SLEW = FAST  | PULLUP;
NET "pca9564_data<7>"  LOC = "AE22" | IOSTANDARD = LVCMOS33  | SLEW = FAST  | PULLUP;
NET "pca9564_int"  LOC = "AM23" | IOSTANDARD = LVCMOS33;
NET "pca9564_rd"  LOC = "AG23" | IOSTANDARD = LVCMOS33  | SLEW = FAST;
NET "pca9564_reset"  LOC = "AN23" | IOSTANDARD = LVCMOS33  | SLEW = FAST;
NET "pca9564_wr"  LOC = "AF23" | IOSTANDARD = LVCMOS33  | SLEW = FAST;
```

Digilent Virtex-II Pro Development System:

```
NET "clock_ip"  LOC = "AH16" | IOSTANDARD = LVCMOS25 ;
NET "pca9564_address<0>"  LOC = "V5" | IOSTANDARD = LVTTL  | SLEW = FAST ;
NET "pca9564_address<1>"  LOC = "V6" | IOSTANDARD = LVTTL  | SLEW = FAST ;
NET "pca9564_ce"  LOC = "U7" | IOSTANDARD = LVTTL  | SLEW = FAST ;
NET "pca9564_data<0>"  LOC = "V8" | IOSTANDARD = LVTTL  | SLEW = FAST ;
NET "pca9564_data<1>"  LOC = "V7" | IOSTANDARD = LVTTL  | SLEW = FAST ;
NET "pca9564_data<2>"  LOC = "U8" | IOSTANDARD = LVTTL  | SLEW = FAST ;
NET "pca9564_data<3>"  LOC = "Y1" | IOSTANDARD = LVTTL  | SLEW = FAST ;
NET "pca9564_data<4>"  LOC = "V4" | IOSTANDARD = LVTTL  | SLEW = FAST ;
NET "pca9564_data<5>"  LOC = "U9" | IOSTANDARD = LVTTL  | SLEW = FAST ;
NET "pca9564_data<6>"  LOC = "W2" | IOSTANDARD = LVTTL  | SLEW = FAST ;
NET "pca9564_data<7>"  LOC = "U5" | IOSTANDARD = LVTTL  | SLEW = FAST ;
NET "pca9564_int"  LOC = "W3" | IOSTANDARD = LVTTL ;
NET "pca9564_rd"  LOC = "W1" | IOSTANDARD = LVTTL  | SLEW = FAST ;
NET "pca9564_reset"  LOC = "AA1" | IOSTANDARD = LVTTL  | SLEW = FAST ;
NET "pca9564_wr"  LOC = "V3" | IOSTANDARD = LVTTL  | SLEW = FAST ;
```

Abbildungsverzeichnis

Literaturverzeichnis

[I²CStandard] Philips I²C Bus Standard
http://www.nxp.com/acrobat_download/literature/9398/39340011.pdf
[Stand 27.10.2006]

[I²CAN] Philips I²C Application Note
http://www.nxp.com/acrobat/applicationnotes/AN10216_1.pdf
[Stand 27.10.2006]

[PCA9564AN] Philips PCA9564 Application Note
http://www.nxp.com/acrobat_download/applicationnotes/AN10148_4.pdf [Stand 27.10.2006]

[PCA9564] Philips PCA9564 Datenblatt
http://www.nxp.com/acrobat_download/datasheets/PCA9564_4.pdf
[Stand 29.9.2006]

[Xilinx] Webseite zum Xilinx Prototyping Board
http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?sGlobalNavPick=&sSecondaryNavPick=&category=-21481&iLanguageID=1&key=HW-AFX-FF1152-200
[Stand 29.9.2006]

[Xilinx2] Virtex-II Platform FPGA
http://www.xilinx.com/publications/matrix/virtexmatrix.pdf

[FIFO] Datenblatt zum Xilinx FIFO Core Generator
http://www.xilinx.com/ipcenter/catalog/logicore/docs/fifo_generator.pdf#search=%22xilinx%20core%20generator%20fifo%22
[Stand 29.9.2006]

[Digilent] Webseite zum Digilent Protoyping System
http://www.digilentinc.com/Products/Detail.cfm?Prod=XUPV2P&Nav1=Products&Nav2=Programmable [Stand 29.9.2006]

[eDIP240] Datenblatt zum LCD eDIP240-7 von Electronic Assembly
http://www.lcd-module.de/deu/pdf/grafik/edip240-7.pdf
[Stand 29.9.2006]

[DodOrg] Poster zur DodOrg Case Study “Task mapping, scheduling and swapping-on-the-fly”
http://ces.univ-karlsruhe.de/~bauer/pdf/DodOrg_Poster_Stuttgart06.pdf [Stand 29.9.2006]

[DodOrg2] Webseite Organic Computing
http://www.organic-computing.de/SPP [Stand 29.9.2006]

[I²CVoltage] Application Note von Maxim zur I²C Spannungswandlung
http://www.maxim-ic.com/appnotes.cfm/appnote_number/1159
[Stand 29.9.2006]

[MAX3373] Datenblatt Maxim MAX3373 Low Voltage Level Translator
http://datasheets.maxim-ic.com/en/ds/MAX3372E-MAX3393E.pdf [Stand 29.9.2006]

[SPI] Motorola Serial Peripheral Interface
http://www.mct.de/faq/spi.html [Stand 29.9.2006]

[ASIPMeister] ASIPMeister Homepage
http://www.eda-meister.org/asip-meister/ [Stand 23.10.2006]

[Agilent] Agilent MSO6054A
http://cp.literature.agilent.com/litweb/pdf/5989-2000EN.pdf
[Stand 23.10.2006]

[Hennessy96] John L. Hennessy, David A Patterson “Computer Architecture – A Quantitative Approach” 2nd edition 1996; Morgan Kaufmann Publishers, Inc.

Ich versichere, dass ich die vorliegende Arbeit selbständig angefertigt und mich fremder Hilfe nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß veröffentlichtem oder unveröffentlichtem Schrifttum entnommen sind, habe ich als solche kenntlich gemacht.

Ort/Datum Unterschrift