Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Alexander Pautz

Design und Implementierung eines Mikroprozessorinterfaces für ARM Prozessoren einschließlich einer Beispielapplikation

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Alexander Pautz

Design und Implementierung eines Mikroprozessorinterfaces für ARM Prozessoren einschließlich einer Beispielapplikation

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Schäfers
Zweitgutachter : Prof. Dr. rer. nat. Klemke

Abgegeben am 26. Mai 2009

Alexander Pautz

Thema der Bachelorarbeit

Design und Implementierung eines Mikroprozessorinterfaces für ARM Prozessoren einschließlich einer Beispielapplikation

Stichworte

Mikroprozessorinterface, FGPA, VHDL, Register, Prozessorbus, ARM, AMBA, APB, SPI.

Kurzzusammenfassung

Thema dieser Arbeit ist die Entwicklung eines vielseitig einsetzbaren und wiederverwendbaren Mikroprozessorinterface für ARM Prozessoren. Das Interface soll einem Entwickler von Peripherie Zeit und Arbeit ersparen, indem es immer wiederkehrende Funktionen in vordefinierten Registern bereithält. Es ist dabei so flexibel, dass es leicht um neue Registertypen erweitert oder mit nur wenig Aufwand an einen neuen Prozessorbus angeschlossen werden kann. Mit einem SPI-Interface wurde eine erste Anwendung, welche das Mikroprozessorinterface nutzt, entwickelt.

Alexander Pautz

Title of the paper

Design und implementation of a microprocessor interface for ARM processors including an exemplary application

Keywords

Microprocessor interface, FGPA, VHDL, register, processorbus, ARM, AMBA, APB, SPI.

Abstract

The subject of this thesis is the development of a versatile and reusable microprocessor interface for ARM processors. The interface will save time and work while developing peripherals, offering predefined registers for recurring functions, It is flexible because it can be easily expanded with new registertypes and can be attached to another processorbus with little effort. As a first application using the microprocessor interface, a SPI interface was implemented.

Inhaltsverzeichnis

Tabellenverzeichnis 7

Abbildungsverzeichnis 9

1 Einleitung 11
- 1.1 Motivation . . . 11
- 1.2 Zielsetzung . . . 11

2 Bereits existierende Microprocessorinterface 13
- 2.1 Support von Xilinx, Altera und ARM Ltd. . . . 13
- 2.2 Andere Verfügbare μPIs . . . 14
- 2.3 Entschluss zur Neuentwicklung . . . 15

3 Microprocessorinterface Designüberlegungen 16
- 3.1 Auswahl des Prozessorbusses . . . 17
  - 3.1.1 AMBA 3 APB Protokoll . . . 19
- 3.2 Register . . . 23
  - 3.2.1 Anforderungen an die Register . . . 23
  - 3.2.2 (A)Synchrone Register . . . 23

4 Microprocessorinterface Implementation 25
- 4.1 Allgemeine Implementationsentscheidungen . . . 25
  - 4.1.1 Anzahl der VHDL-Prozesse . . . 25
  - 4.1.2 Signale und Variablen . . . 26
  - 4.1.3 Namensschema der Signale . . . 26
  - 4.1.4 Synchroner und asynchroner Reset . . . 26
  - 4.1.5 Unabhängigkeit von Bibliotheken . . . 27
- 4.2 Register . . . 27
  - 4.2.1 Datenregister . . . 30
  - 4.2.2 Round Rotating Counterregister . . . 31
- 4.3 APB-Interface . . . 33
- 4.4 Verbindung von Register und APB-Interface (Generator) . . . 35

5 Microprocessorinterface Qualitätssicherung **36**
5.1 Testbench des P2B_Dataregister 36
5.2 Testbench des B2P_Dataregister 37
5.3 Testbench des Round Rotating Counterregister 38
5.4 Testbench des APB-Interfaces 42
5.5 1024 Register Test 46
5.5.1 Testergebnis 47

6 SPI (Beispielapplikation) **48**
6.1 SPI Beschreibung 48
6.1.1 Signale des SPI 48
6.1.2 SPI Datenübertragung 50
6.2 SPI Implementation 52
6.2.1 Verwendete Register 53
6.2.2 Einbinden des μPI 59
6.2.3 Interner Ablauf 67
6.3 VHDL Simulationen 70
6.3.1 Konfiguration als Master 70
6.3.2 Sendebeginn 71
6.3.3 Liste der durchgeführten Tests 72
6.4 Tests in Hardware 73
6.4.1 Aufbau der Testhardware 73
6.4.2 Die Software des AVRs 80
6.4.3 Testen der Verbindung zwischen PC und AVR 81
6.4.4 Testen des APB-Interfaces und der Register 82
6.4.5 Testen des SPI-Interfaces 83
6.4.6 8 Stunden Test 87

7 Zusammenfassung und Ausblick **92**
7.1 Ausblick 93

Literaturverzeichnis **94**

A μPI Benutzerbeschreibung **97**
A.1 APB Interface 97
A.2 Register- und Signalnamensschema 97
A.3 Register 99
A.3.1 Einfaches Datenregister 99
A.3.2 Einfaches Kontrollregister 100
A.3.3 Reset on Read Kontrollregister 101
A.3.4 Einfaches Statusregister 102

A.3.5 Reset on Read Statusregister 103
A.3.6 Impulse Statusregister 104
A.3.7 Negative Impulse Statusregister 106
A.3.8 Saturation Counterregister 107
A.3.9 Round Rotating Counterregister 109
A.3.10 Pufferregister 111
A.3.11 Haupt IRQ-Register 117
A.4 Globale Definitionen 119
A.5 Der Generator 120

B Aufbau des Quelltextes 122

C Zusatzplatine mit MAX3232 125

D CD-Inhalt 127
D.1 Literatur 127
D.2 Quellcode 127
D.2.1 AVR-C-Code 127
D.2.2 VHDL 128
D.3 Sonstiges 129

Tabellenverzeichnis

4.1 Definierte Signale für Register zu Businterface 28
4.2 Definierte Signale für Register zu Peripherie 29

6.1 SPI Register Map 54
6.2 SPI Kontrollregister 1 54
6.3 SPI Kontrollregister 2 55
6.4 SPI Statusregister 1 56
6.5 SPI Statusregister 2 57
6.6 SPI Empfangspuffer im Slave Modus 58
6.7 FPGA sendet als Master Daten per SPI 86

A.1 Signale des p2b_dataregisters 99
A.2 Signale des b2p_dataregisters 100
A.3 Signale des controlregisters 101
A.4 Generics des controlregisters 101
A.5 Signale des ror_controlregisters 102
A.6 Generics des ror controlregisters 102
A.7 Signale des statusregisters 103
A.8 Signale des ror_statusregisters 104
A.9 Signale des impulse_statusregisters 105
A.10 Generics des ror controlregister 105
A.11 Signale des negative_impulse_statusregister 106
A.12 Generics des ror controlregister 107
A.13 Signale des saturation_counterregister 108
A.14 Generics des saturation_counterregister 108
A.15 Signale des round_counterregister 110
A.16 Generics des round_rotating_counterregister 110
A.17 Signale des b2p_bufferregister 112
A.18 Generics des b2p_bufferregister 113
A.19 Signale des b2p_bufferregister_b 113
A.20 Generics des b2p_bufferregister_b 114
A.21 Signale des p2b_bufferregister 115
A.22 Generics des p2b_bufferregister 115

A.23 Signale des p2b_bufferregister_b 116
A.24 Generics des p2b_bufferregister_b 117
A.25 Signale des mainirqregister 118
A.26 Generics des mainirqregister 118

Abbildungsverzeichnis

2.1 Microprocessorinterface von CAST 14

3.1 Interner Aufbau eines allgemeinen Mikrocontrollers 16
3.2 Interner Aufbau eines allgemeinen Mikrocontrollers mit APB-Bus . . 19
3.3 APB Lesezugriff ohne Wartezyklen 21
3.4 APB Lesezugriff mit Wartezyklen 21
3.5 APB Schreibzugriff ohne Wartezyklen 22
3.6 APB Schreibzugriff mit Wartezyklen 22

4.1 Grundaktivität eines Registers 30
4.2 Aktivitätsdiagramm des P2B Dataregisters 30
4.3 Aktivitätsdiagramm des B2P Dataregisters 31
4.4 Aktivitätsdiagramm des Round-Rotating-Counterregisters 32
4.5 Aktivitätsdiagramm des APB-Interfaces 34

5.1 Timing Simulation des P2B_Dataregisters 37
5.2 P2B_Dataregister Synthesereportauszug 37
5.3 Timing Simulation des P2B_Dataregisters 38
5.4 Timing Simulation ”Zählen” des Round_Rotating_Counterregisters . 39
5.5 Timing Simulation ”Lesen” des Round_Rotating_Counterregisters . 39
5.6 Timing Simulation ”Wert setzen” des Round_Rotating_Counterregisters 40
5.7 Timing Simulation ”Überlauf” des Round_Rotating_Counterregisters 40
5.8 Timing Simulation ”Überlauf am Maximum” des Round_Rotating_Counterregisters 41
5.9 Timing Simulation ”Schreiben/Schreiben - Lesen/Zählen” des Round_Rotating_Counterregisters 41
5.10 Simulation des APB-Interfaces - Schreiben 43
5.11 Simulation des APB-Interfaces - Lesen 44
5.12 Simulation des APB-Interfaces - Fehler 45
5.13 1024 Register Test, 1 47
5.14 1024 Register Test, 2 47

6.1 SPI Master mit 3 Slaves 49
6.2 Datenübertragungsbeispiel bei SPI 50

6.3 SPI Datenübertragung mit CPHA = 0 51
6.4 SPI Datenübertragung mit CPHA = 1 52
6.5 Schematische Darstellung eines FPGAs mit μPI-SPI 53
6.6 Quelltextauszug defs.vhd 60
6.7 Quelltextauszug spi_main.vhd (1) 62
6.8 Quelltextauszug spi_main.vhd (2) 64
6.9 Quelltextauszug spi_main.vhd (3) 65
6.10 Quelltextauszug spi_main.vhd (4) 65
6.11 Quelltextauszug spi_main.vhd (5) 65
6.12 Quelltextauszug spi_main.vhd (6) 66
6.13 Quelltextauszug Haupt-IRQ-Register 66
6.14 SPI Grundaktivität 67
6.15 SPI Masteraktivität 68
6.16 SPI Slaveaktivität 69
6.17 Timinigsimulation: Konfiguration als Master 70
6.18 Timinigsimulation: Sendestart des SPIs 71
6.19 Foto des NEXYS 2 Boards 73
6.20 Foto eines AT90CAN128-Boards 74
6.21 Foto des geänderten AT90CAN128-Boards 76
6.22 Schematische Darstellung des Aufbaues 78
6.23 Gesamtansicht des Testaufbaues 79
6.24 Gesamtansicht des Testaufbaues 80
6.25 AVR-Menu 81
6.26 Schreiben und Lesen des Kontrollregisters 0x08 82
6.27 Schreiben und Lesen des Statusregisters 0x10 83
6.28 FPGA als SPI-Master 84
6.29 FPGA sendet als Master Daten per SPI 85
6.30 8h Test, Fehler 1 88
6.31 8h Test, Fehler 2 89
6.32 Hardwaretest - Fehleranalyse mit dem Oszilloskop 89
6.33 Hardwaretest - Fehleranalyse mit dem Oszilloskop, 2 90
6.34 Hardwaretest - Fehleranalyse mit dem Oszilloskop 91

B.1 Quelltextaufbau, Auszug 1 122
B.2 Quelltextaufbau, Auszug 2 123
B.3 Quelltextaufbau, Auszug 3 124

C.1 Schaltplan der MAX3232 Zusatzplatine 125
C.2 Layout der MAX3232 Zusatzplatine 126
C.3 Fotos der fertig aufgebauten MAX3232 Zusatzplatine 126

1 Einleitung

In diesem Kapitel soll zunächst einmal dargelegt werden, warum diese Arbeit geschrieben wird und welche Ziele damit erreicht werden sollen.

1.1 Motivation

In der heutigen Zeit werden Mikrocontroller in vielen elektrischen Geräten verbaut. In der Massenproduktion großer Firmen werden Controller für einzelne Aufgaben entwickelt und zu Millionen gefertigt. Für kleine Produktionsserien ist die Implementierung des Controllers in einem FPGAs[1] aus wirtschaftlichen Gründen die geeignete Form. FPGAs sind synthetisierbare Hardwarebausteine, welche von verschiedenen Herstellern mit unterschiedlichen Eigenschaften verkauft werden. Als Hardwarebeschreibungssprachen gibt es VHDL[2] und Verilog. Solch ein System wird System on Chip (SoC) bzw. Ein-Chip-System genannt.

Hardwareentwicklung, welche innerhalb der HAW durchgeführt wird, fällt in den Bereich von Prototypen und Kleinserien. Bisher gibt es das Problem, dass bei einem Wechsel einzelner Komponenten auf dem FPGA oder Wechsel des FPGAs selbst teils große Änderungen an den bisher entwickelten Komponenten vorgenommen werden müssen. Des weiteren müssen verschiedene Entwickler ähnliche Komponenten oftmals neu erstellen.

1.2 Zielsetzung

Ziel dieser Bachelorarbeit ist es, eine für viele Entwickler nutzbare Mikroprozessorschnittstelle (μPI) zu schaffen, welche die Kommunikation mit dem Prozessor

[1] Field Programmable Gate Array
[2] Very High Speed Integrated Circuit Hardware Description Language

abnimmt und möglichst viele sinnvolle einfach zu nutzende Register für den Entwickler bereitzustellen. Es soll eine möglichst hohe Kompatibilität zu den verschiedenen FPGA-Herstellern sichergestellt werden. Das Interface ist so aufzubauen, dass einzelne Teile, also Register oder die Anbindung an den Prozessorbus leicht gewechselt werden können, um eine möglichst hohe Flexibilität zu erreichen.

Weiterhin soll als erstes Interface, welches dass μPI benutzte und somit als Test und zur Demonstration des μPI dient, ein Serial Peripherial Interface (SPI) entwickelt werden.

2 Bereits existierende Microprocessorinterface

Ein Mikroprozessor Interface ist eine typische Anwendung, die immer wieder implementiert wird. Bevor hier ein neues entwickelt wird, soll analysiert werden, ob eines der bestehenden genutzt werden kann.

2.1 Support von Xilinx, Altera und ARM Ltd.

Die beiden großen FPGA Hersteller Xilinx und Altera stellen beide unterschiedliche Prozessoren und Interface für diese bereit. Xilinx bietet den PowerPC und seine Eigenentwicklung - den Microblaze - frei verfügbar mit diversen IP-Cores[1] an [siehe Xilinx Inc., 2009]. Altera hat den Nios II entwickelt und unterstützte auch andere Prozessoren wie den V1 Coldfire von Freescale Semiconductor [siehe Altera Corporation, 2009]. ARM Ltd. bietet diverse Prozessoren für FPGAs an, darunter befindet sich der angestrebte ARM9 Prozessor. Für den Prozessor gibt es frei verfügbare IP-Cores [siehe ARM Limited, 2009].

Xilinx und Altera bieten keine eigene Implementation eines ARM Prozessor an und wie es zu erwarten ist, unterstützen sie nur ihre eigenen FPGAs. Das zu entwickelnde μPI soll aber herstellerunabhängig von einem FPGA auf einen anderen übertragen werden können. ARM bietet für den angestrebten Prozessor zwar IP-Cores an, aber kein universelles μPI, welches die Anforderungen erfüllt (vgl. Kapitel 1.2) und zur Entwicklung von eigener Peripherie genutzt werden kann.

[1] (Intellectual Property Core

2.2 Andere Verfügbare μPIs

Es gibt eine ganze Reihe von Firmen die μPIs als fertige Bibliotheken anbieten. Als ein Beispiel soll hier die Firma CAST Inc. genannt werden. Die Firma CAST Inc. stellt IP-Cores für diverse FPGA Hersteller zur Verfügung. Darunter befindet sich auch der "PIP-AMBA-E SoC Kernel" für ARM9 Prozessoren [siehe CAST Inc.]. Die folgende Grafik (Abb. 2.1) stammt von der Webseite von Cast und gibt einen Überblick über das Microprocessorinterface von CAST.

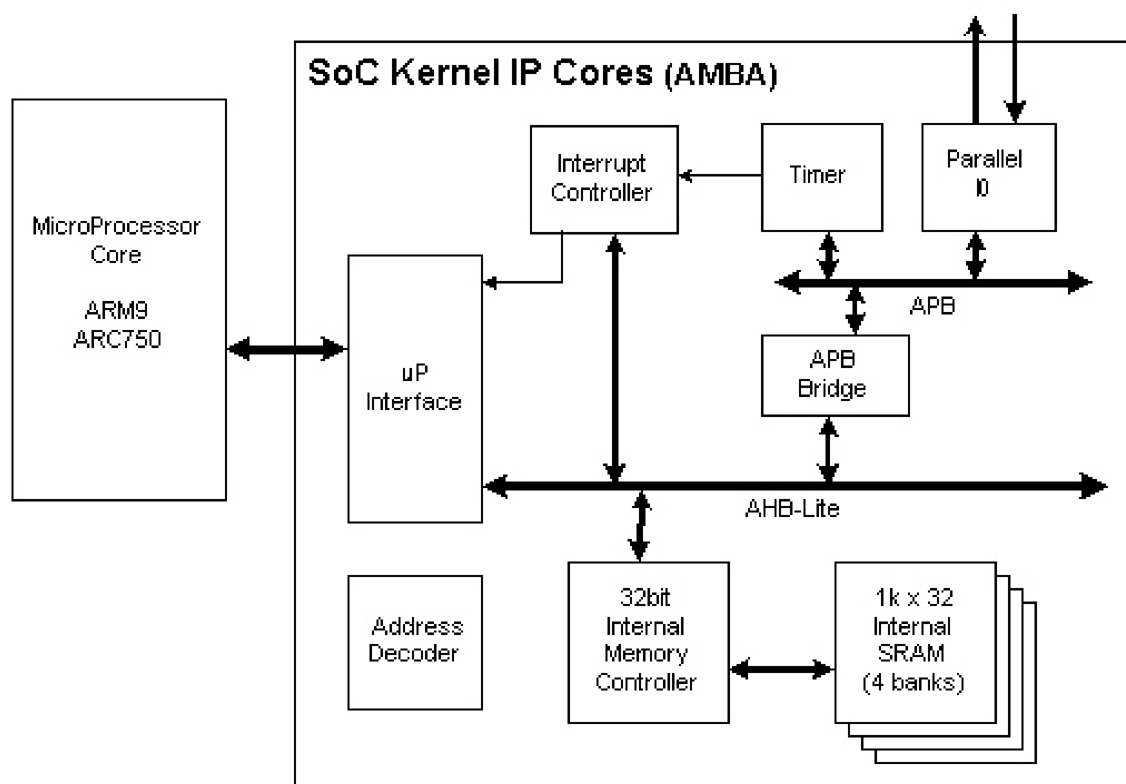


Abbildung 2.1: Microprocessorinterface von CAST

Das Interface bietet aber nicht, die in der Zielsetzung (Kapitel 1.2) verlangten Register. Ebenso ist das Interface kostenpflichtig und nicht frei verfügbar, womit eventuell jede Neuentwicklung, die auf diesem Interface beruht, Kosten verursacht.

2.3 Entschluss zur Neuentwicklung

Es gibt noch einige andere Microprocessorinterface, die meist aber nicht mit dem angestrebten Prozessor kompatibel sind. Teilweise sind die Interface nur käuflich erwerbbar und keines der untersuchten Interface bietet bereits fertige Register, die je nach Bedarf zusammengestellt werden können. Gerade die fertigen Register sollen einem Entwickler von Peripherie, aber viel Arbeit abnehmen.

Da die Anforderungen von den existierenden μPIs nicht erfüllt werden und auch nicht durch kleine Erweiterungen erfüllt werden können, wird hier ein komplett neues μPI entwickelt.

3 Microprocessorinterface Designüberlegungen

Als erster Schritt wird eine Analyse der Grundfunktionalitäten durchgeführt, der die Grundlage für die nachfolgende Implementierung bildet.

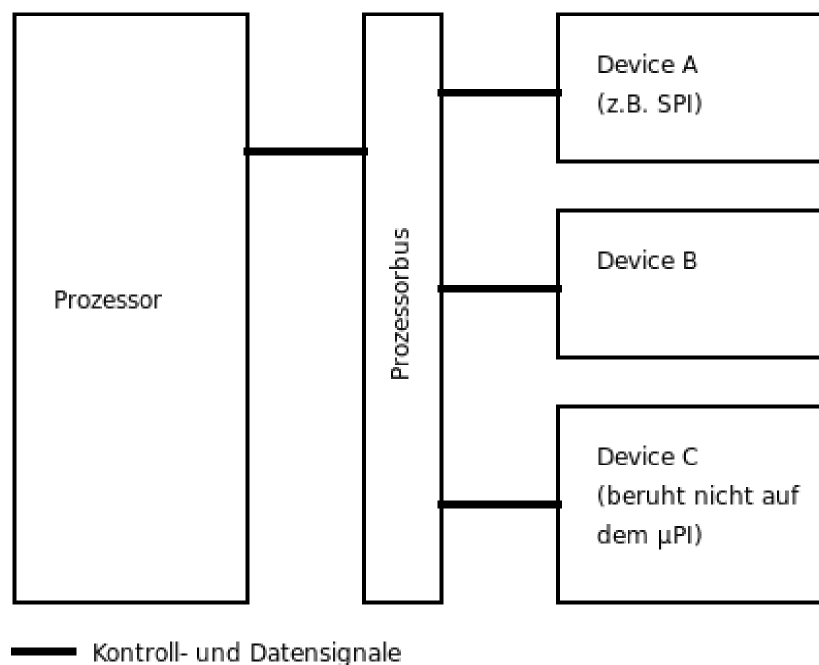


Abbildung 3.1: Interner Aufbau eines allgemeinen Mikrocontrollers

Die obige Abbildung 3.1 zeigt den schematischen Aufbau eines einfachen Mikrocontrollers. Device A und B beruhen auf dem hier zu entwickelnden μPI. Für den Prozessor und Prozessorbus sind sie nicht von anderen Peripheriegeräten zu unterscheiden. Dies ist auch wichtig, damit es keiner speziellen Anpassung des Prozessors oder Busses bedarf, bevor sie verwendet werden können.

3.1 Auswahl des Prozessorbusses

Um die von der Peripherie genutzten Register vom Prozessor aus ansprechbar zu machen, muss eine Anbindung an den Prozessorbus erfolgen. Da das endgültige Board noch nicht vorhanden ist und somit weder Prozessor noch Bus endgültig feststehen, gibt es hier einige Spielräume. Als Prozessor wird ein ARM9v4$_{xTDMI}$ favorisiert. Bezüglich des Busses gibt es bisher jedoch keine Vorgaben.

Die Kommunikation zwischen Bus und μPI soll zügig ablaufen, damit der Prozessor möglichst wenig Zeit mit Buszugriffen verbringt und viel Zeit für die eigentlichen Berechnungen hat. Es ist zu analysieren, welche Art von Bus die geringste Belastung für den Prozessor darstellt. Da hier nicht die Entwicklung eines Prozessorbusses, sondern die Entwicklung eines Mikroprozessorinterface im Vordergrund steht, muss der gewählte Bus für den gewählten Prozessor verfügbar sein.

Die schnellsten Busse sind asynchrone Busse. Sie ermöglichen Schreib- und Leseoperationen in nur einem Takt. Die ARM Ltd., welche die Designs für ARM-Prozessoren entwirft, hat in ihren Spezifikationen jedoch keinen asynchronen Prozessorbus. Auch heutige PowerPCs oder frei verfügbare Prozessorbusse, wie der Wishbone[1], bieten keine asynchronen Prozessorbusse an. Auch sonst sind zu asynchronen Bussen und speziell Prozessorbussen kaum Informationen zu finden. Der Grund dafür liegt in der allgemein hohen Geschwindigkeit der heutigen Systeme. Außerdem bieten aktuelle Mikroprozessoren teilweise Cache, wodurch sich die Anzahl an Buszugriffen erheblich verringern lässt. Asynchrone Busse haben auch den Nachteil, dass sie Hazards auf Steuerleitungen als gültige Signale erkennen können und damit die Funktionalität nicht mehr gewährleistet ist. Synchrone Busse agieren nur auf der Clockflanke und verarbeiten Signale, die zu diesem Zeitpunkt anliegen. Hazards dürfen bei synchronen Bussen nur auf der Clockleitung und zum Zeitpunkt der Clockflanke nicht auftreten.

Da hier nicht die Entwicklung eines neuen Prozessorbusses zur Aufgabe steht und das μPI möglichst flexibel sein soll, wird ein synchroner Prozessorbus gewählt. Der verwendete Prozessor wird sehr wahrscheinlich ein ARM sein. Es ist also sinnvoll, einen Bus zu wählen, der in vielen ARM-Prozessoren genutzt werden kann. Die ARM Ltd. stellt mit ihrer "Advanced Microcontroller Bus Architecture 3" (AMBA 3) Spezifikation 3 Busse zur Verfügung.

[1] siehe http://www.opencores.org/?do=wishbone

Der derzeit schnellste und umfangreichste Bus ist das ”AMBA Advanced eXtensible Interface” (AXI / Quelle: ARM Limited [2004b]). Dieser Bus ist sehr schnell und relativ komplex. Er beherrscht zum Beispiel DMA und ”out-of-order transaction”. Dieser Bus ist vor allem für schnelle Datentransfers wie zum Beispiel die Anbindung von internem Speicher gedacht. Der zweite Bus ist der ”Advanced High-performance Bus” (AHB / Quelle: ARM Limited [2006]). Dieser Bus ist ebenfalls für hohe Geschwindigkeiten ausgelegt, bietet jedoch einen nicht so großen Funktionsumfang. Der dritte Bus heißt ”Advanced Peripheral Bus” (APB / Quelle: ARM Limited [2004a]) und ist kein eigenständiger Bus. Er benötigt immer eine Bridge, welche die angeschlossene Peripherie mit einem der anderen beiden Busse verbindet. Sein Funktionsumfang ist geringer als der der anderen beiden, dafür ist sein Interface sehr einfach. Alle Busse schaffen es einen Lese- oder Schreibzugriff in zwei Takten abzuarbeiten, sofern die Peripherie schnell genug ist.

Bevor ein Bus gewählt wird, ist zu analysieren, welchen Funktionsumfang und welche Geschwindigkeit vom Großteil der Peripherie gefordert wird.

Es wird davon ausgegangen, dass die meiste Peripherie im Verhältnis zum Prozessor langsam arbeiten wird. Beispiele wären unter anderem SPI, UART[2], USB Host Controller[3], CAN[4] und GPIO[5]. Das μPI an den AHB oder das AXI zu binden, bremst das ganze System in solchen Fällen aus. In der Spezifikation des AHB steht dazu ”Die am häufigsten verwendeten AHB-Lite Slaves sind interner Speicher, externe Speicher-Interface und Peripherie mit hoher Bandbreite. Auch wenn Peripherie mit niedriger Bandbreite als AHB-Lite Slave arbeiten kann, ist es aus Performancegründen für das System besser, diese an den AMBA Advanced Peripheral Bus (APB) anzuschließen. Die Überbrückung von diesem höherem Buslevel und dem APB wird von einem AHB-Lite Slave - der APB-Bridge - übernommen.” [übersetztes Zitat: ARM Limited, 2006, S. 18]. Auch wenn der APB den geringsten Funktionsumfang bietet und der langsamste Bus ist, ist er für die meiste Peripherie immer noch ausreichend schnell. Auch lässt sich der APB durch die Bridges sehr einfach in Systeme integrieren, die kein APB-Interface haben.

Nach Abschluss der Analyse fällt die Wahl auf den APB, da er für den gesteckten Einsatzbereich am besten geeignet ist. Entspricht der APB doch einmal nicht den Anforderungen, kann er mit nur geringem Aufwand gegen einen anderen Bus ausgetauscht werden. Eine Anforderung an das μPI ist, dass es möglichst flexibel ist. Deshalb wird es so modular aufgebaut, dass einzelne Teile

[2]Universal Asynchronous serial Receiver and Transmitter (serielle Schnittstelle)
[3]Universal serial Bus (Serieller Bus Master)
[4]Controler Area Network (Bussystem)
[5]General Purpose Input/Output (einfache Ein-/Ausgabeports)

ausgetauscht werden können. Wird für einzelne Entwicklungen eine schnellere Prozessoranbindung benötigt, so kann das APB-Interface ausgetauscht werden, ohne dass das gesamte μPI neu geschrieben werden muss.

Die Abbildung 3.1 aus Kapitel 3 (Microprocessorinterface Designüberlegungen) muss wie folgt (Abb. 3.2) durch eine APB-Bridge erweitert werden. Dabei beruhen Device A und B wieder auf dem μPI.

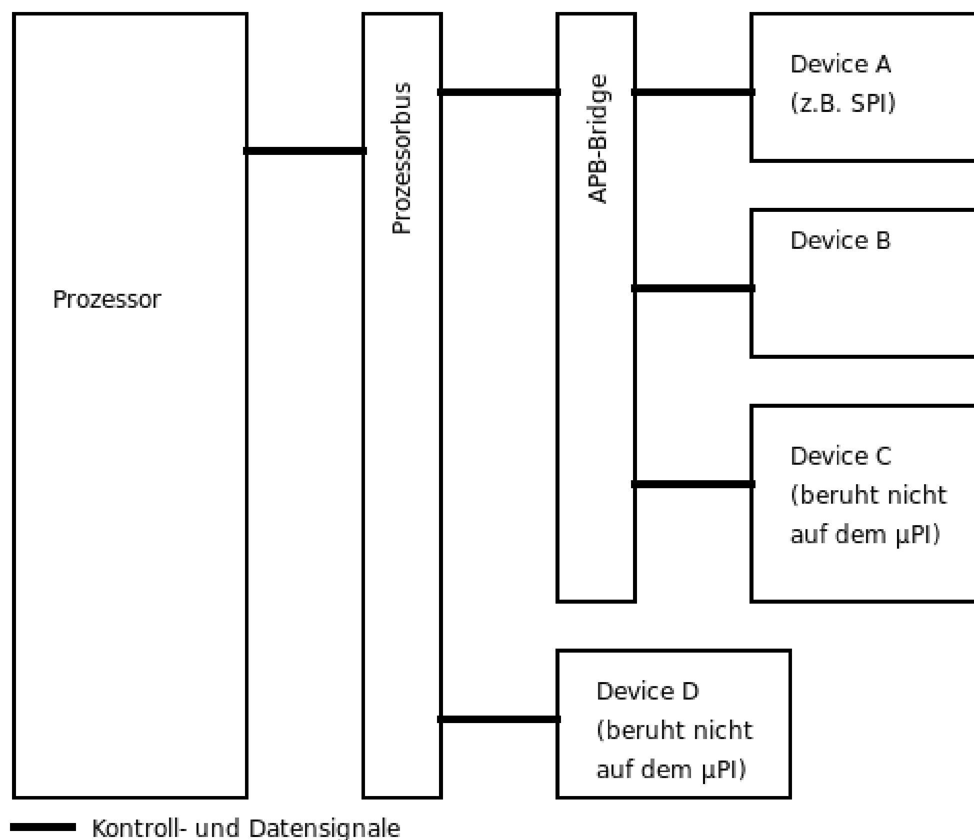


Abbildung 3.2: Interner Aufbau eines allgemeinen Mikrocontrollers mit APB-Bus

3.1.1 AMBA 3 APB Protokoll

Dieser Bus bietet nur einfache Lese- und Schreiboperationen, sowie eine Fehlerrückmeldung vom Slave an den Master. Der Slave ist dabei immer das Device also die Peripherie. Der Master ist immer die APB-Bridge.

Alle Operation auf dem Bus laufen synchron zum Takt (PCLK). Es wird auf der steigenden Taktflanke gearbeitet. Ein Datentransfer wird immer vom Master eingeleitet und wird in zwei Phasen abgewickelt. Die erste Phase ist die sogenannte Setup-Phase, welche immer einen Takt lang ist. In dieser Phase gibt der Master die Adresse (PADDR) vor, stellt ein, ob es sich um eine Lese- oder Schreiboperation (PWRITE) handelt und wählt den Slave über ein Selectsignal (PSEL) aus. Handelt es sich um einen Schreibzugriff, so legt der Master auch die Daten (PWDATA) auf den Bus. Die zweite Phase ist die sogenannte Zugriffsphase. Der Master signalisiert den Start der Zugriffsphase, indem er das Signal PENABLE auf high setzt. Während der Zugriffsphase liest der Slave die Daten vom Bus oder legt selbst Daten auf den Bus (PRDATA), je nachdem ob es sich um einen Lese- oder Schreibzugriff handelt. Diese Phase ist im Takt, nachdem der Slave signalisiert hat, dass er fertig ist, beendet. Dies tut er, indem er PREADY auf high setzt. Der Slave hat somit die Möglichkeit, die Zugriffsphase von einem Takt auf eine beliebige Anzahl von Takten zu strecken. Eine maximale Anzahl von Takten ist in den Spezifikationen nicht angegeben. Tritt beim Slave während des Zugriffs ein Fehler auf, kann er dies durch das Signal PSLVERR signalisieren. Das Signal PSLVERR ist optional. Fehlt es, können allerdings auch keine Fehler vom Slave an den Master gemeldet werden. Der Slave zieht zum Signalisieren eines Fehlers PSLVERR auf high und signalisiert mit PREADY, dass er fertig ist. Während eines Zugriffs, in dem ein Fehler entstand, kann - muss sich der interne Zustand des Slaves aber nicht verändert haben. Tritt ein Fehler während eines Lesezugriffs auf, wird damit automatisch signalisiert, dass die Daten nicht gültig sind.

Dazu folgen einige Grafiken, welche den zeitlichen Ablauf darstellen. Lesezugriffe meinen immer das Auslesen des Slaves durch den Master. Bei Schreibzugriffen sendet der Master Daten an den Slave.

Die Abbildung 3.3 [ARM Limited, 2004a, S. 22] zeigt die Busaktivitäten bei einem Lesevorgang ohne Wartezyklen. In der Setup-Phase, welche im Takt T1 statt findet, wird die Adresse eingestellt, PWRITE wird auf LOW gezogen (nicht Schreiben) und mittels PSEL wird das Device ausgewählt. Im darauf folgendem Takt T2 findet die Zugriffsphase statt, in der der Master PENABLE setzt. Das Device legt die Daten auf PRDATA an und bestätigt den Vorgang mit PREADY. Im Takt T3 ist der Lesevorgang abgeschlossen.

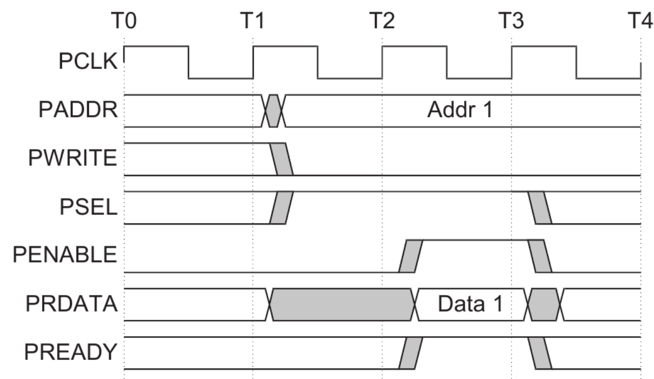


Abbildung 3.3: APB Lesezugriff ohne Wartezyklen

Die Abbildung 3.4 [ARM Limited, 2004a, S. 23] stellt die Busaktivitäten bei einem Lesevorgang mit Wartezyklus dar. Die Setup-Phase findet genauso wie in der vorhergehenden Abbildung statt. Im ersten Takt der Zugriffsphase (T2) setzt der Master wieder PENABLE, aber das Device legt die Daten erst einige Takte später an PRDATA an und setzt PREADY ebenfalls erst in dem Takt T4, in dem es die Daten anlegt.

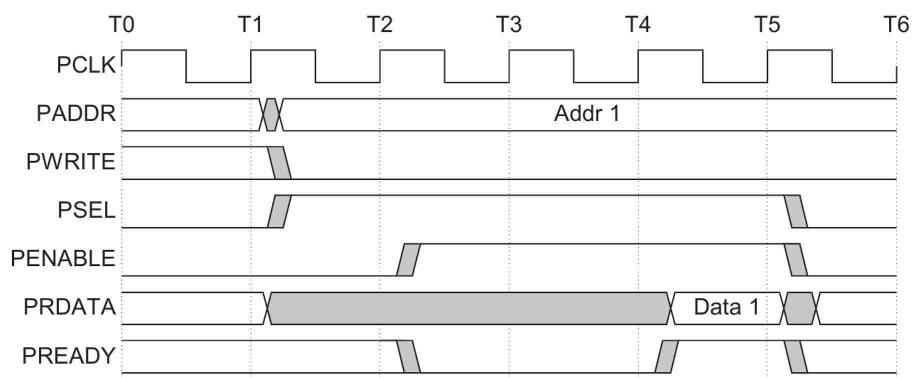


Abbildung 3.4: APB Lesezugriff mit Wartezyklen

Die Abbildung 3.5 [ARM Limited, 2004a, S. 20] stellt die Busaktivitäten bei einem Schreibvorgang ohne Wartezyklen dar. In der Setup-Phase, welche im Takt T1 statt findet, wird die Adresse eingestellt. PWRITE wird auf high gezogen. Mittels PSEL wird das Device ausgewählt und der Master legt die Daten an PWDATA an. Im darauf folgenden Takt T2 findet die Zugriffsphase statt, in der der Master PENABLE setzt. Das Device liest die Daten von PWDATA aus und bestätigt den Vorgang mit PREADY. In Takt T3 ist der Schreibvorgang abgeschlossen und alle Signale werden zurückgenommen.

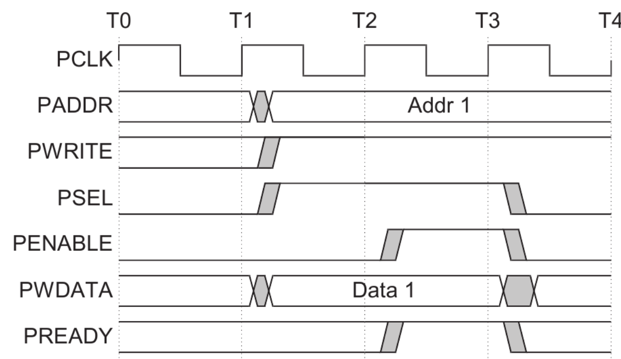


Abbildung 3.5: APB Schreibzugriff ohne Wartezyklen

In der Abbildung 3.6 [ARM Limited, 2004a, S. 21] findet ein Schreibzugriff mit Wartezyklen statt. Die Setup-Phase verläuft im Takt T1 genauso wie im vorhergehendem Beispiel. Das Device benötigt jedoch mehr Zeit für das Einlesen der Daten und setzt PREADY erst, wenn es mit dem Einlesen der Daten fertig ist. Das Einlesen beginnt in Takt T2 und endet erst in Takt T4, in dem PREADY gesetzt wird. In Takt T5 ist die Übertragung abgeschlossen und alle Signale werden zurückgenommen.

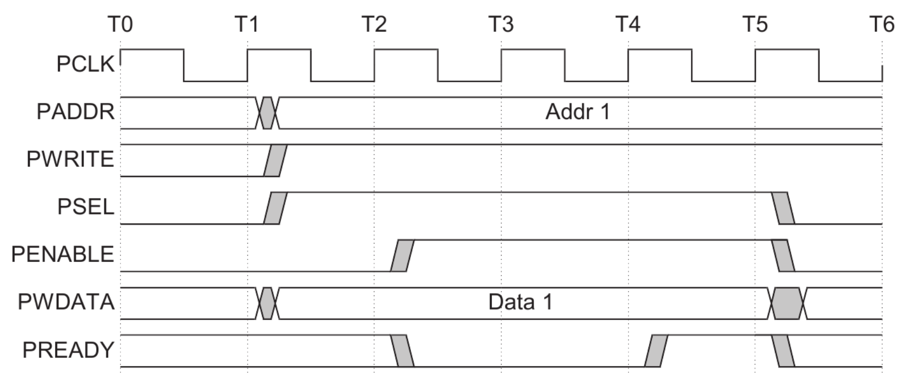


Abbildung 3.6: APB Schreibzugriff mit Wartezyklen

3.2 Register

Nachfolgend wird analysiert, wie die Register gebaut werden und was für Register benötigt werden.

3.2.1 Anforderungen an die Register

Die Register sollen möglichst für sich alleine arbeiten können und dabei verschiedene, von einander klar getrennte Aufgaben erfüllen. Dabei sind folgende Aufgaben gefordert:

- Kontroll- und Statusregister sind klar voneinander zu trennen
- Kontrollregister sollen wieder ausgelesen werden können
- Statusregister sollen optional Reset-on-Read unterstützen
- Es soll Zählregister geben, welche Saturation-Counter und Round-Rotating-Counter unterstützen
- Neben einfachen Datenregistern soll es auch Pufferregister(FIFO) geben
- Edge- und Level-Sensitive Interrupts sollen unterstützt werden

Um Probleme bei der Datenkonsistenz zu vermeiden, darf nur jeweils der Bus oder die Peripherie auf ein Register schreiben. Die andere Seite besitzt höchstens die Möglichkeit, ein Register zurück zu setzen und es somit als gelesen zu markieren. Da der Adressraum eines 32Bit Prozessors wie dem ARM groß genug ist, müssen keine Tricks angewendet werden, um die ein oder andere Adresse einzusparen, in dem einige Register teils vom Bus und teils von der Peripherie beschreibbar sind.

3.2.2 (A)Synchrone Register

Es ist zu analysieren, ob die Register synchron oder asynchron zum Prozessor bzw. Prozessorbus arbeiten sollen.

Asynchrone Register können schneller auf die Außenwelt reagieren, da sie frei vom Prozessortakt arbeiten. Sie können sofern nötig auch schneller als der Prozessor arbeiten und könnten somit genutzt werden, um einem langsam getakteten Prozessor komplexe Berechnungen abzunehmen. Dies wird jedoch relativ selten

benötigt, da selbst Peripherie, die mit dem gleichen Takt wie der Prozessor läuft und für eine bestimmte komplexe Berechnungen optimiert ist, diese schneller ausführt, als eine für allgemeine Aufgaben ausgelegte ALU.

Die meisten heutigen Mikrocontroller arbeiten mit einer Geschwindigkeit von mehreren MHz. Die meisten externen Busse im Mikrocontrollerbereich arbeiten mit nur maximal 1 MBit/s. Die Reaktion auf externe Ereignisse ist mit synchronen Registern schnell genug. Die Reaktion auf die Eingabe eines Menschen an einer Tastatur spielt sich beispielsweise gar im Millisekundenbereich ab.

Auch ist die Kommunikation von asynchron laufender Hardware untereinander aufwändiger, da es eben keinen gemeinsamen Takt zur Abstimmung gibt. Zur sicheren Übertragung von Daten ist ein Handshakeverfahren notwendig. Ist der Prozessorbus selbst asynchron, ist der Aufwand kaum höher, da bereits ein Handshakeverfahren zur Synchronisierung mit dem Prozessor vorhanden ist. Ist der Prozessorbus hingegen synchron, kommt zu dem zusätzlichem Hardwareaufwand noch zusätzlicher Zeitaufwand, da der Bus nur zu jedem Takt einen Schritt im Handshakeverfahren abarbeiten kann.

Es wurde analysiert, dass asynchrone Register sicherlich einige Vorteile bringen, aber der zusätzliche Aufwand an Hardware, der eventuelle Zeitnachteil und die wahrscheinlich seltene Ausnutzung der Vorteile der asynchronen Register und dem damit verbundenem Nachteil für alle anderen Nutzer stehen nicht im Verhältnis. Somit werden alle Register synchron zum Prozessor gebaut.

4 Microprocessorinterface Implementation

Nachdem die Analyse aller Anforderungen und Vorgaben, welche in Kapitel 3 statt fand, abgeschlossen ist. Wird hier nun die Umsetzung der Ergebnisse beschrieben.

4.1 Allgemeine Implementationsentscheidungen

In diesem Abschnitt geht es um allgemeine Entscheidungen, wie etwas in VHDL codiert wird, um den FPGA möglichst effizient zu nutzen und dem Synthesetool die Arbeit möglichst einfach zu machen.

Das grundlegende Vorgehen bei der VHDL-Codierung, welches Stoff der Fächer Digitaltechnik 1 und 2 ist, soll hier nicht diskutiert werden.

4.1.1 Anzahl der VHDL-Prozesse

Es gibt die Möglichkeit, viele VHDL-Prozesse zu erzeugen, um so parallel laufende Aufgaben möglichst weit aufzufächern. Damit lassen sich die einzelnen Teilaufgaben eventuell leichter erkennen und bei Bedarf ändern und anpassen.

Bei der Verwendung von nur wenigen Prozessen sind die Abhängigkeiten einzelner Teilaufgaben untereinander aber sehr viel leichter zu erkennen und Änderungen an einer Teilaufgabe leichter durchzuführen.

Teilaufgaben in einem großen Prozess zu finden, kann durch gute Kommentare im VHDL-Code gut unterstützt werden, so dass in dieser Arbeit wenige Prozesse verwendet werden. Es gibt jedoch mindestens zwei Prozesse - einen getakteten und einen kombinatorischen. Gibt es klar getrennte Aufgaben, die keine oder klar erkennbare Abhängigkeiten zu anderen Prozessen haben, werden eventuell auch mehr als zwei Prozesse pro Architecture verwendet.

4.1.2 Signale und Variablen

Wird ein Signal in einem kombinatorischen kurzen Prozess mehr als einmal zugewiesen, wird dieses Signal am Anfang des Prozesses auf eine Variable geführt und am Ende des Prozesses die Variable zurück auf das Signal geführt. Dies wird getan, da Signale erst am Ende des Prozesses aktualisiert werden, Variablen aber auch innerhalb eines Prozesses. Damit ist es eindeutiger, welchen Wert ein Signal innerhalb und am Ende des Prozesses haben wird. Innerhalb von langen VHDL-Prozessen werden auf Grund besserer Übersicht alle Signale auf Variablen geführt.

4.1.3 Namensschema der Signale

Alle Signale und Variablen folgen einem Namensschema, um die Zuordnung des Signales oder der Variable zu vereinfachen.

Ein ”R” am Anfang eines Signales steht für die Zugehörigkeit zu einem Register. Externe Signale werden von ”P2R”, ”R2P”, ”B2R” oder ”R2B” gefolgt. Dieses Schema gibt die Datenrichtung des Signales an. Ein ”R” bezeichnet das Register. Ein ”P” steht für Peripherie. Ein ”B” steht für (Prozessor-)Bus oder Businterface. Die ”2” steht für englisch ”to” (zu). Ein ”R2P” bedeutet in Langform ”Register to Peripherie”.

Eine ausführlichere Beschreibung des Namensschema befindet sich im Kapitel Register- und Signalnamensschema (A.2)

4.1.4 Synchroner und asynchroner Reset

Bei der Wahl des Resets ist neben den Eigenschaften, dass ein asynchroner Reset vielleicht durch einen Hazard ausgelöst werden kann und ein synchroner Reset mindestens einen Takt lang anliegen muss, viel mehr die Eigenschaft des FPGAs zu berücksichtigen.

Der Hersteller Altera macht in ”IC Designers – An Optimal Approach to Programmable Logic” dazu folgende Aussage: ”The synchronous versus asynchronous reset methodology approach for FPGAs is a little different than that for ASICs. In FPGAs, synchronous and asynchronous reset circuitry is already built into each flip flop, so you cannot save any area by using one versus the other.” [Chandrashekar und

Mahmud, 2005, S. 4] Dies bedeutet übersetzt soviel, dass die Wahl des Resets egal ist, da jedes FlipFlop sowohl den synchronen wie asynchronen Reset fest eingebaut hat und somit kein Platz im FPGA gespart werden kann.

Die Firma Xilinx gibt in ihrem ”Xcell Journal” vom 15.03.2005 eine ganze Reihe von Gründen für den synchronen Reset an. Selbst einfache Schaltungen können durch die Verwendung eins synchronen Resets weniger Platz benötigen. Komplexe Schaltungen mit Blockram können zu dem um bis zu 150 % schneller ausgeführt werden [vgl. Garrault und Philofsky, 2005, S. 31 - 35].

Da es nach Aussage von Altera egal ist, welchen Reset man verwendet, nach Aussage von Xilinx auf ihren FGPAs aber große Unterschiede zu Gunsten des synchronen Resets entstehen können, wird hier der synchrone Reset verwendet.

4.1.5 Unabhängigkeit von Bibliotheken

Da das gesamte μPI unabhängig von einem speziellen FPGA oder Hersteller sein soll, ist es notwendig, dass auch keine herstellerspezifischen Bibliotheken genutzt werden. Um dies zu erreichen, ist unabhängig von dieser Bachelorarbeit das TICEP_CONV Package aus dem Rechnerstrukturen-Praktikum auf Synthetisierbarkeit zu prüfen. Des weiteren ist das Package so anzupassen, dass es unabhängig von FPGA-Hersteller eingesetzt werden kann. Da dies nicht Aufgabe dieser Bachelorarbeit ist und diese Unabhängigkeit noch nicht sichergestellt wurde, wird in dieser Bachelorarbeit die Hülle des TICEP_CONV Package benutzt und die Bibliotheksfunktionen werden innerhalb des Paketes aufgerufen. Somit muss später nicht das gesamte μPI geändert werden, sondern nur die Funktionen innerhalb des TICEP_CONV Packages.

4.2 Register

In diesem Kapitel werden einige grundlegende Eigenschaften aller Register beschrieben und anschließend ein paar Register beispielhaft erklärt.

Eine der wichtigsten Grundvoraussetzungen ist, dass alle Register möglichst die gleichen Ein- und Ausgangssignale haben. Dadurch lassen sich Register während des Entwicklungsprozesses leicht austauschen und es ist leichter, die Signale der einzelnen Register in größere Gruppen zu bündeln (Arrays). Auf der anderen Seite ist es besser, alle Register mit so wenig Signalen wie möglich auszustatten. Es kostet unnötig Hardware, wenn in ein Register Signale für einen Schreibzugriff

führen, obwohl das Register diesen gar nicht unterstützt. Um beide Aufgaben zu erfüllen, besitzen alle Register aus einem Pool von definierten Signalen nur die Signale, die sie wirklich benötigen. Die Adaption, dass alle Register von außen gleich aussehen, übernimmt ein Generator (siehe 4.4).

Ein Register hat bildlich gesehen zwei Seiten. An der einen sitzt das Bus-Interface, dass das Register mit dem Prozessor verbindet. An der anderen Seite sitzt die Peripherie, die der Benutzer entwickelt. Für beide Seite werden Datenschreib- und Datenlesesignale benötigt. Außerdem benötigt man Möglichkeiten, Fehler zu melden. Für einige spezielle Register werden auch weitere Signale, wie Zählimpulse, benötigt. Die folgenden beiden Tabellen listen alle Signale, die definiert sind auf. Die Abkürzungen der Namen ist in Kapitel A.2 (Register- und Signalnamensschema) beschrieben.

Die Tabelle 4.1 beschreibt alle vorhandenen Signale, welche das Register mit dem Businterface verbinden.

Name	Bitbreite	Beschreibung
R_R2B_DATA	32	Datenübertragung vom Register zum Bus. Da alle Register auslesbar sein sollen, besitzt jedes Register dieses Signal.
R_B2R_DATA	32	Datenübertragung vom Businterface zum Register. Dieses Signale besitzen alle Register, die vom Prozessor beschrieben werden können.
R_B2R_DI_EN	1	Writeenable. Wird für alle Register benötigt, die vom Prozessor beschrieben werden können.
R_B2R_DP	1	Daten verarbeitet. Durch dieses Signal können Register erfahren, wenn sie ausgelesen wurden. Ist zum Beispiel für Read-on-Reset nötig.
R_R2B_DI_ERR	1	Gibt Registern die Möglichkeit fehlerhafte Schreibzugriffe mitzuteilen. Register, die nicht vom Prozessor beschrieben werden können liefern durch den Generator automatisch einen Fehler auf diesem Signal.
R_R2B_DO_ERR	1	Gibt dem Register die Möglichkeit, einen Fehler beim Lesen mitzuteilen, wenn die gespeicherten Daten also wissentlich fehlerhaft sind und nicht verarbeitet werden dürfen.

Tabelle 4.1: Definierte Signale für Register zu Businterface

Die Tabelle 4.2 beschreibt alle vorhandenen Signale, welche das Register mit der Peripherie verbindet.

Name	Bitbreite	Beschreibung
R_P2R_DATA	32	Datenübertragung von der Peripherie zum Register. Wird für alle Register benötigt, die von der Peripherie beschrieben werden können.
R_R2P_DATA	32	Datenübertragung vom Register zur Peripherie. Ermöglicht der Peripherie das Register auszulesen.
R_P2R_DI_EN	1	Writeenable. Wird für alle Register benötigt, die durch die Peripherie beschreibbar sind und nur auf Aufforderung neue Daten übernehmen sollen.
R_P2R_DP	1	Daten verarbeitet. Ermöglicht dem Register zu erfahren, wenn Daten durch die Peripherie verarbeitet wurden.
R_R2P_ERR	1	Fehlerrückmeldung. Ermöglicht dem Register Fehler an die Peripherie zu melden.
R_P2R_COUNT	1	Zählimpuls. Wird für Counterregister benötigt. Die genaue Bedeutung ist beim jeweiligem Register in Kapitel A.3 beschrieben.

Tabelle 4.2: Definierte Signale für Register zu Peripherie

Einige Register bieten auch noch R_IRQ_OUT als Signal, um einen Interrupt zu melden. Da es keine Möglichkeit gibt, diese Signale automatisch mit dem Haupt IRQ-Register zu verbinden, muss dieses innerhalb der Peripherie von Hand geschehen. Das Haupt IRQ-Register liefert dafür das Signal R_IRQS_IN. Zusätzlich liefert es das Signal R_IRQ_EN, mit dessen Hilfe einzelne Interrupts ein- und ausgeschaltet werden können. Für die genaue Handhabung der Signale siehe A.3.11.1.

In den folgenden Unterkapiteln wird die Implementation einiger Register veranschaulicht. Dazu dienen Aktivitätsdiagramme. Die Diagramme sind teils vereinfacht, aber alle Aktivitäten sind im VHDL-Code der Register wiederzufinden.

Alle Register haben die gleiche Grundfunktionalität. Bei einem Reset werden alle Register in ihren Ursprung zurück versetzt. Es wird immer von Takt zu Takt gearbeitet. Die Grafik 4.1 spiegelt dieses Verhalten wider.

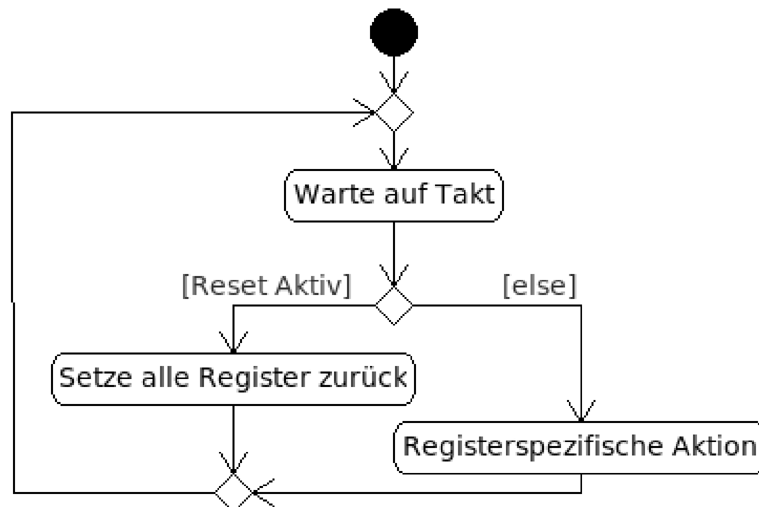


Abbildung 4.1: Grundaktivität eines Registers

4.2.1 Datenregister

Das Datenregister gibt es in zwei Versionen, die eine kann nur vom Prozessor und die andere nur von der Peripherie beschrieben werden.

Die Version, welche von der Peripherie beschrieben wird, übernimmt anliegende Werte mit jedem Takt. Das Aktivitätsdiagramm (Abb. 4.2) ist entsprechend einfach.

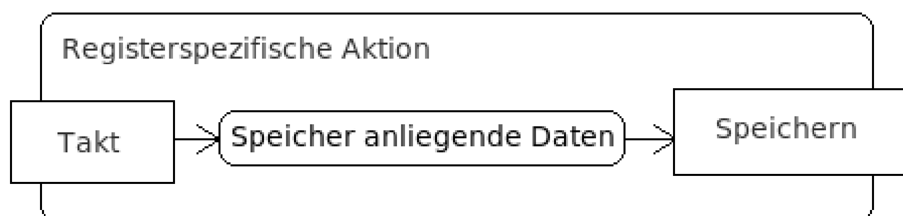


Abbildung 4.2: Aktivitätsdiagramm des P2B Dataregisters

Das Register, welches vom Bus beschrieben wird, darf die Werte nur übernehmen, wenn es auch angewählt wurde. Vor der Datenübernahme ist also eine Abfrage nötig. Das Diagramm mit Abbildung Nr. 4.3 zeigt das Verhalten des Bus-two-Peripherie Dataregister.

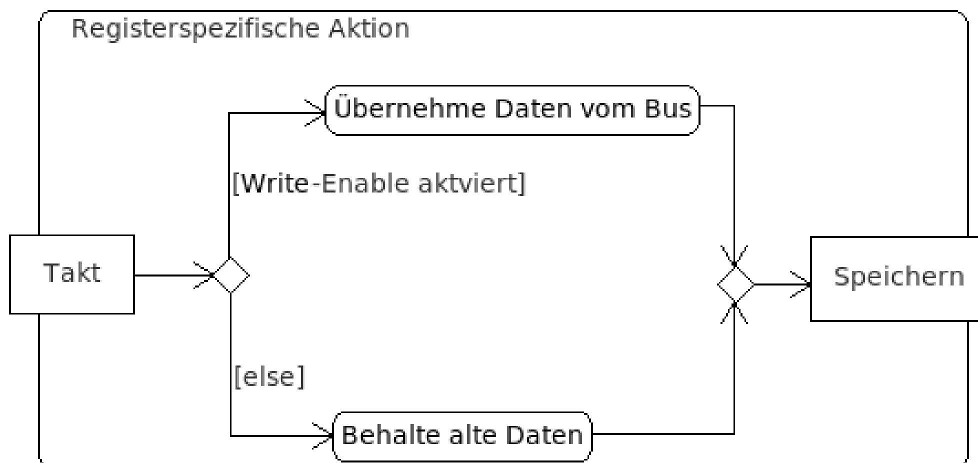


Abbildung 4.3: Aktivitätsdiagramm des B2P Dataregisters

4.2.2 Round Rotating Counterregister

Das Round Rotating Counterregister wird hier als Beispiel für ein komplexeres Register verwendet. Eine genaue Funktionsbeschreibung des Round Rotating Counterregisters gibt es in Kapitel A.3.9.

Die einzelnen Teilaufgaben dieses Registers ließen sich zwar in getrennten Prozessen verarbeiten. Allerdings haben sie Abhängigkeiten untereinander, welche leichter erkennbar sind, wenn sich die gesamten Aufgaben in einem Prozess befinden. Das Aktivitätsdiagramm Abb. Nr. 4.4 zeigt den internen Ablauf des Round Rotating Counterregisters.

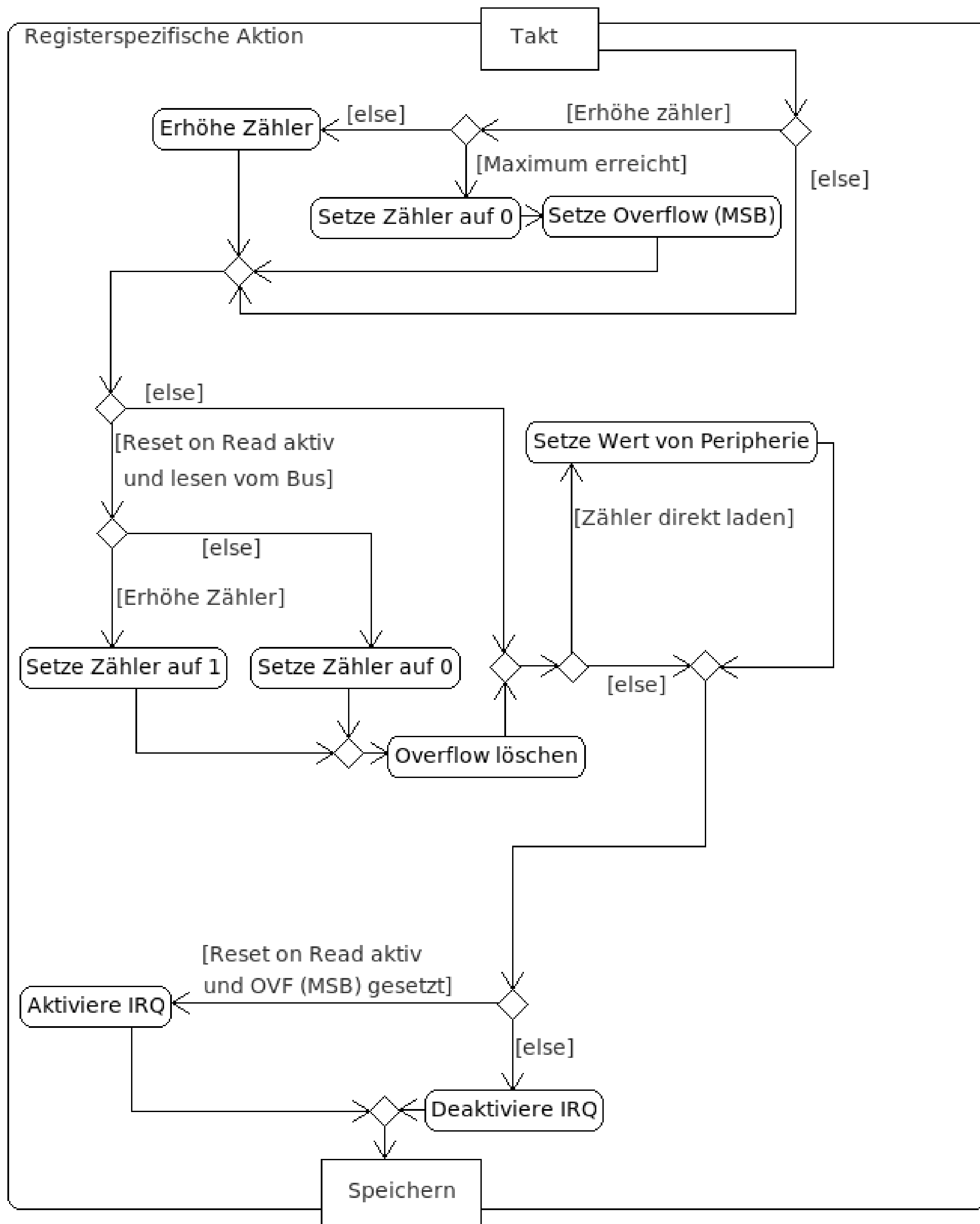


Abbildung 4.4: Aktivitätsdiagramm des Round-Rotating-Counterregisters

4.3 APB-Interface

In diesem Kapitel wird die Implementation des APB-Interfaces beschrieben.

Um einen Datentransfer möglichst schnell abzuschließen, sollte nur die minimal nötige Anzahl an Takten für eine Übertragung verwendet werden. Beim APB-Bus sind dies zwei Takte. Dies bedeutet jedoch, dass zwischen dem Auswerten und Anlegen der Daten kein Takt bleibt, um diese vom Register ins Interface zu kopieren. Das APB-Interface muss die Daten also direkt vom Register zur APB-Bridge durchreichen. Ein direktes Durchreichen der Daten spart außerdem FlipFlops im APB-Interface.

Nachdem das Device durch die ABP-Bridge angewählt wurde (PSEL=1), muss das APB-Interface als erstes die Gültigkeit der Adresse prüfen. Bei einer ungültigen Adresse meldet das Interface über PSLVERR einen Fehler und bricht die Bearbeitung ab. Bei einer gültigen Adresse berechnet das Interface die interne Adresse des gewählten Registers. Über diese interne Adresse werden alle nötigen Signale vom und zum Register über (De-)Multiplexer weitergeleitet.

Das folgende Aktivitätsdiagramm (Abb. 4.5) veranschaulicht den Vorgang eines Lese- und Schreibzugriffs. Man sieht, dass das Interface nur während der Setup-Phase arbeitet. Während der Access-Phase liegen alle Daten für das Register und die APB-Bridge bereits an. Am Ende der Access-Phase werden alle Daten zurückgenommen. Im Vergleich zum VHDL-Code ist dieses Diagramm vereinfacht, jedoch lässt sich jeder Abschnitt des Diagramms auf ein oder mehrere Zeilen im VHDL-Code übertragen.

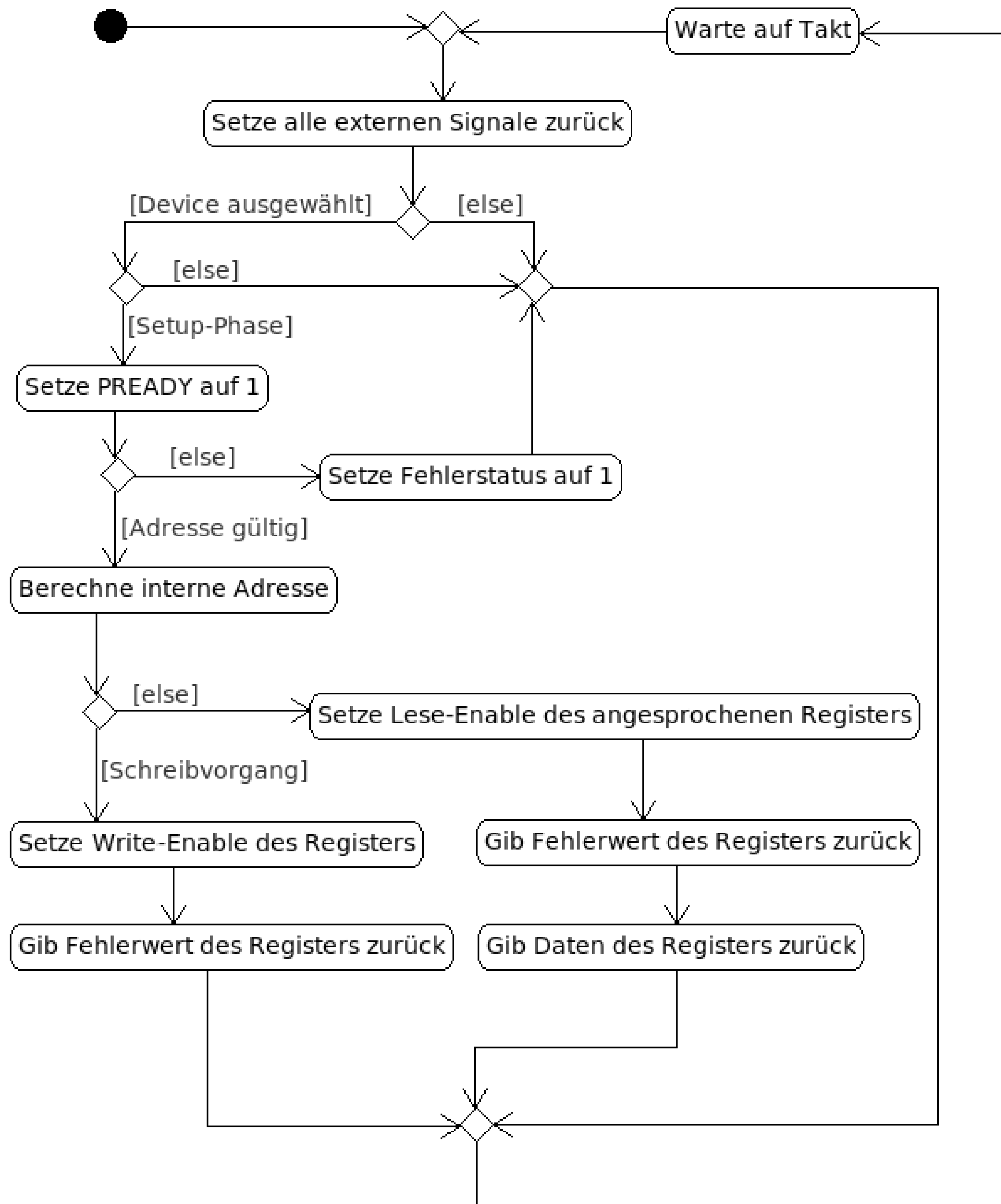


Abbildung 4.5: Aktivitätsdiagramm des APB-Interfaces

Wie man in der obigen Abbildung (4.5) sieht, wird im Interface keine Unterscheidung zwischen den einzelnen Registern gemacht. Für das APB-Interface haben alle Register den vollen Funktionsumfang bezüglich der Möglichkeit vom Bus beschrieben und gelesen zu werden. Außerdem besitzen aus der Sichtweise des APB-Interfaces alle Register die benötigten Signale, um Schreib- und Lesefehler zu melden. Da nicht alle Register diesen vollen Funktionsumfang bieten, aber alle Register das gleiche Verhalten am APB-Interface haben, wird eine zusätzliche Ebene zwischen APB-Interface und Register benötigt, welche dafür sorgt, dass alle Register das gleiche Verhalten haben. Diese Stufe übernimmt der in Kapitel 4.4 beschriebene Generator.

4.4 Verbindung von Register und APB-Interface (Generator)

Der Generator enthält keinen VHDL-Code, der in der Zielhardware ausgeführt wird. Er enthält nur Anweisungen, zum Generieren von Registern, belegt Signale der Register mit Standardwerten, wenn die Register das Signal nicht unterstützen und verbindet das APB-Interface mit den Registern. Die peripherieseitigen Signale der Register stehen dann in mehreren Arrays zur Verfügung.

Wird kein Haupt IRQ-Register (A.3.11) verwendet, ist das spezielle Signal G_MAIN_IRQ_OUT, welches der Generator für dieses Register bereit hält, zunächst einmal offen. Es gibt in VHDL keine Möglichkeit, dies zur Compile- oder Synthesezeit zu überprüfen oder abzufangen. Dies muss bei der Synthese beachtet werden (vgl. A.5).

5 Microprocessorinterface Qualitätssicherung

Während und nach der Entwicklung ist es notwendig, die Funktionalität der Implementation zu testen. Für jedes Register wurde dafür mindestens eine Testbench, welche alle Funktionen des jeweiligen Registers überprüft, geschrieben. Nachdem alle Testbenches fehlerfrei liefen, wurde das gesamte μPI zusammen mit der SPI Schnittstelle in Hardware getestet. Mehr zu dem Hardwaretest befindet sich in Kapitel 6.4.

In diesem Kapitel werden beispielhaft die Testbenches einiger Register und des APB-Interfaces gezeigt. Die Simulationen und Timing Simulationen wurden mit ModelSim XE 6.3 durchgeführt. Das Erstellen der Timing Simulationen erfolgte mit ISE 10.1.03. Die Testbenches für alle Register befinden sich im Anhang.

Mit dem APB-Interface alleine kann keine Timing Simulation durchgeführt werden, da es bereits bei 4 Registern über 281 IO-Signale benötigen würde, worüber der in dieser Bachelorarbeit verwendete Spartan 3 FPGA nicht verfügt. Es wird deshalb nur eine Post-Syntheses Simulation durchgeführt. Eine Timing Simulation befindet sich in Kapitel 6.3 wo die Timing Simulation des gesamten SPI durchgeführt wird.

Alle Testbenches befinden sich auf der beigelegten CD. Im Anhang D.2.2.3 befindet sich ein Hinweis, wo auf CD sich diese Dateien befinden.

5.1 Testbench des P2B_Dataregister

Als erstes wird das einfachste Register getestet. Das Datenregister, welches im Prinzip 32 FlipFlops mit je einem Eingang und je zwei Ausgängen ist, lässt kaum Fehler erwarten. Es ist vor allem ein Test, ob die verwendete Software richtig arbeitet.

Das Peripherie-to-Bus Dataregister enthält neben Clock und Reset nur einen Eingang. Wenn man Daten hineinschreibt, müssen diese einen Takt später an

den Ausgängen anliegen. Die Grafik (s. Abb. 5.1) ist ein Ausschnitt der Timing Simulation des P2B_Dataregisters. Vom Beginn bis 48 ns ist der Reset aktiv. Danach werden alle Daten, die an R_P2R_DATA angelegt sind, mit der nächsten positiven Taktflanke übernommen.

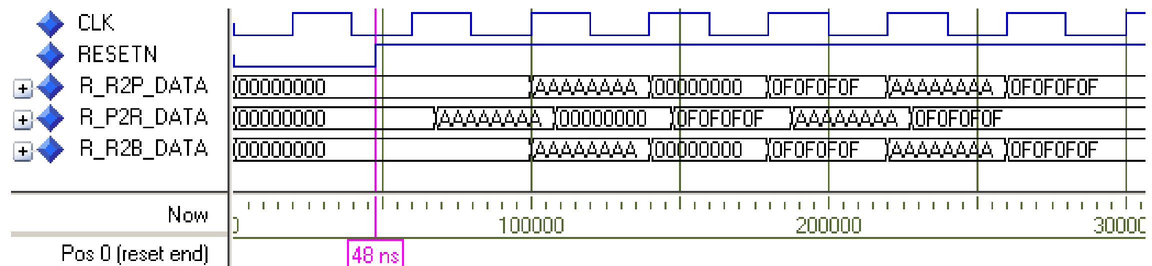


Abbildung 5.1: Timing Simulation des P2B_Dataregisters

Neben dem Ändern der Daten, die in das Register geschrieben werden, ist für das P2B_Dataregister nur ein weiterer Test durchgeführt worden. Dabei wurde die Taktperiode immer weiter gesenkt und das Verhalten beobachtet. Der kleinste Takt bei dem das P2B_Dataregister noch korrekt arbeitete, wurde anschließend mit den Angaben aus dem Synthesereport verglichen. Abbildung 5.2 enthält dazu einen Auszug aus dem Synthesereport des Registers. Das Register wurde in der Simulation bis zu einer Clock von 5ns erfolgreich getestet.

```
==================================================
Minimum period: No path found
Minimum input arrival time before clock: 4.525ns
Maximum output required time after clock: 4.283ns
Maximum combinational path delay: No path found
==================================================
```

Abbildung 5.2: P2B_Dataregister Synthesereportauszug

5.2 Testbench des B2P_Dataregister

In diesem Kapitel wird die Auswirkung des Writeenables anhand des Bus-to-Peripherie Dataregister gezeigt. Die Grafik (s. Abb. 5.3) zeigt die Timing Simulation des B2P_Dataregister.

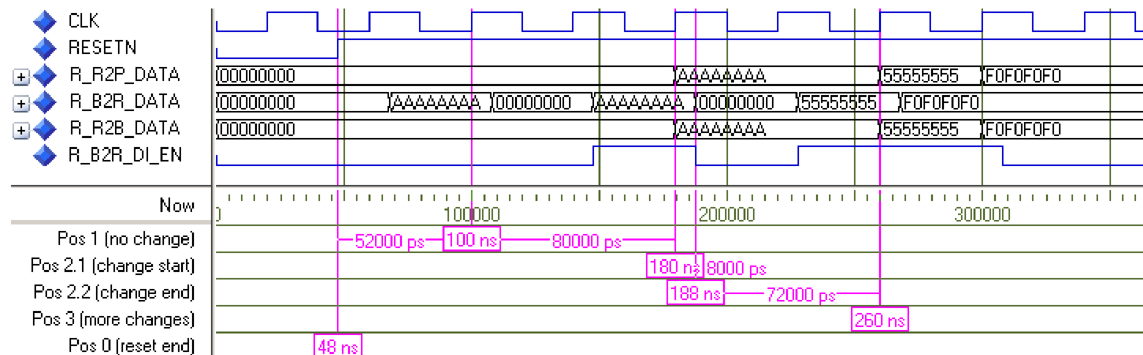


Abbildung 5.3: Timing Simulation des P2B_Dataregisters

Bei Pos. 0 wird der Resetzustand aufgehoben. Kurz darauf werden die ersten Daten an R_B2R_DATA angelegt, welche mit dem nächsten Takt (Pos. 1) nicht übernommen werden, da das Writeenable (R_B2R_DI_EN) nicht mit aktviert wurde. Erst bei Pos 2.1 bis Pos 2.2 werden Daten erfolgreich in das Register gespeichert, in dem sowohl die Daten selbst (R_B2R_DATA) angelegt werden, als auch das Writeenable (R_B2R_DI_EN) aktiviert wird. Da das Writeenable zurückgenommen wird, bleiben die Daten im Register auch beim nächsten Takt erhalten. Pos. 3 zeigt nochmals zwei Schreibvorgänge direkt hintereinander.

Für das B2P_Dataregister gibt es keine weiteren Tests.

5.3 Testbench des Round Rotating Counterregister

Die Simulation des Round Rotating Counterregister dient hier als Beispiel für die Verifikation eines komplexeren Register. In allen folgenden Grafiken ist das Signal ”MAXVALUE” vorhanden, welches im implementierten Register nicht von außen ausgelesen werden kann. Er wird hier für eine bessere Übersicht angezeigt.

Als erstes wird ein einfaches Zählen des Registers getestet. Dafür wird nach dem Reset (Pos. 0), das COUNT-Signal auf high gesetzt. An den DATA-Ausgängen sieht man, wie der Wert des Registers sich jeden Takt um eins erhöht. Die Grafik (s. Abb. 5.4) verdeutlicht diesen Ablauf.

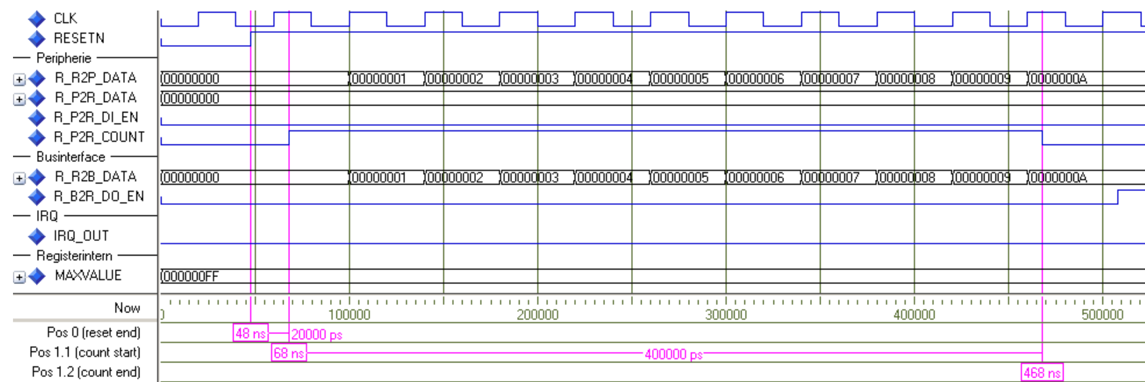


Abbildung 5.4: Timing Simulation "Zählen" des Round_Rotating_Counterregisters

Das Round Rotating Counterregister kann mittels Generic eingestellt werden, ob es Reset-On-Read unterstützt oder nicht. Ist Reset-On-Read aktiv, wird das Register beim Lesen vom Bus auf 0 gesetzt und der Interrupt gelöscht. Bei dieser Simulation ist Reset-On-Read aktiv. Die Abbildung 5.5 zeigt einen Lesevorgang. Der Interrupt bleibt unbeeinflusst, da er zuvor auch nicht gesetzt war.

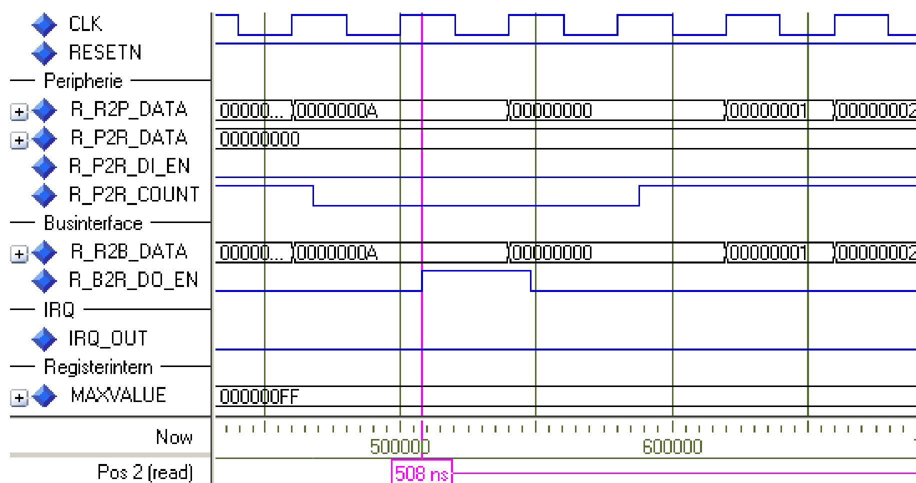


Abbildung 5.5: Timing Simulation "Lesen" des Round_Rotating_Counterregisters

Eine Eigenschaft des Registers ist, dass es von der Peripherie auch direkt gesetzt werden kann. Dafür wird der gewünschte Wert an R_P2R_DATA angelegt und R_R2P_DI_EN auf high gesetzt, um anzuzeigen, dass der Wert übernommen werden soll. In der Grafik (s. Abb. 5.6) wird solch ein Schreibvorgang bei Pos. 3 vorgenommen.

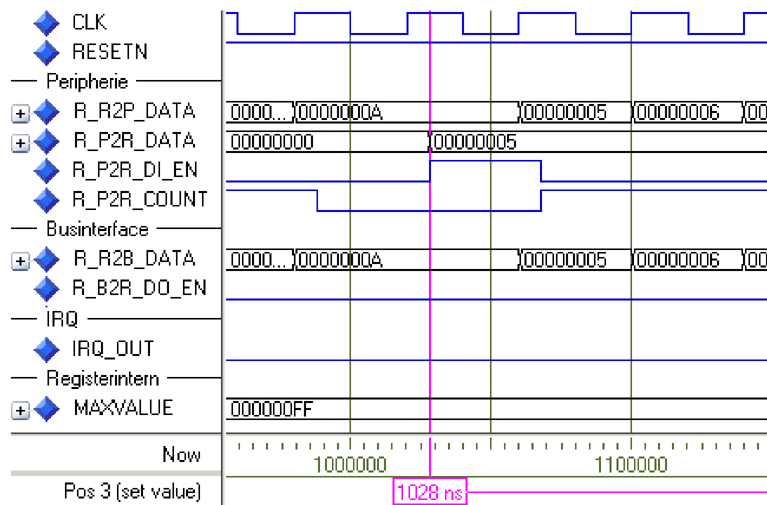


Abbildung 5.6: Timing Simulation ”Wert setzen” des Round_Rotating_Counterregisters

Der nächste Test (Abb. 5.7) zeigt das Verhalten des Registers, wenn beim Zählen der eingestellte Maximalwert erreicht wird. Da Reset-On-Read aktiv ist, wird beim ersten Überlauf (mittlere Markierung bei 1860 ns ”Pos. 5.2”) der Interrupt IRQ_OUT von 0 auf 1 gesetzt. Unabhängig von Reset-On-Read wird das MSB des Counters gesetzt und der Counter auf 0 gesetzt. Beim zweiten Überlauf wird der Counter nur auf 0 zurückgesetzt. Es gibt keine Hinweise darauf, wie oft der Zähler bereits übergelaufen ist.

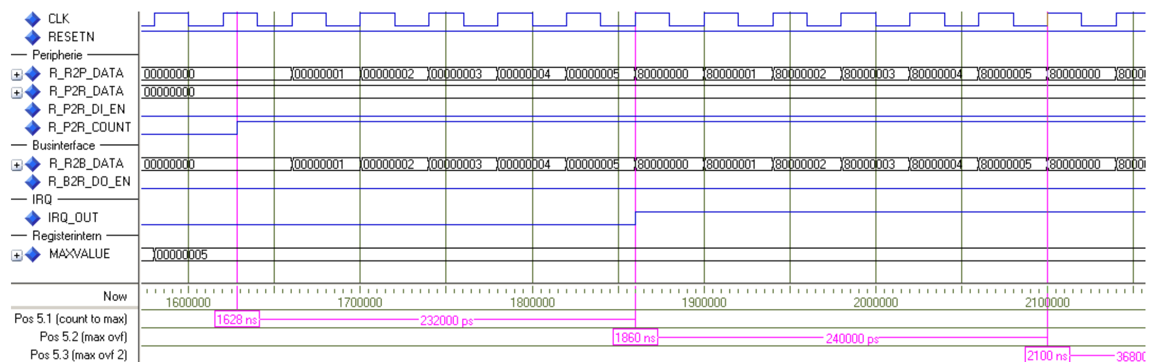


Abbildung 5.7: Timing Simulation ”Überlauf” des Round_Rotating_Counterregisters

Die folgende Abbildung (5.8) zeigt den obigen Test mit absoluten Maximalwerten. ”MAXVALUE” wurde in dieser Testbench auf den größten möglichen Wert eingestellt. Für den aktuellen Wert des Zählers wurde ein Wert nahe des Maximums

geladen. An Position 6 wird der Maximalwert erreicht. Das Verhalten beim Überlauf ist das gleiche wie beim obigen Test (Abb. 5.7) an Position 5.3. Das MSB des Counter und der Interrupt waren bereits beide gesetzt.

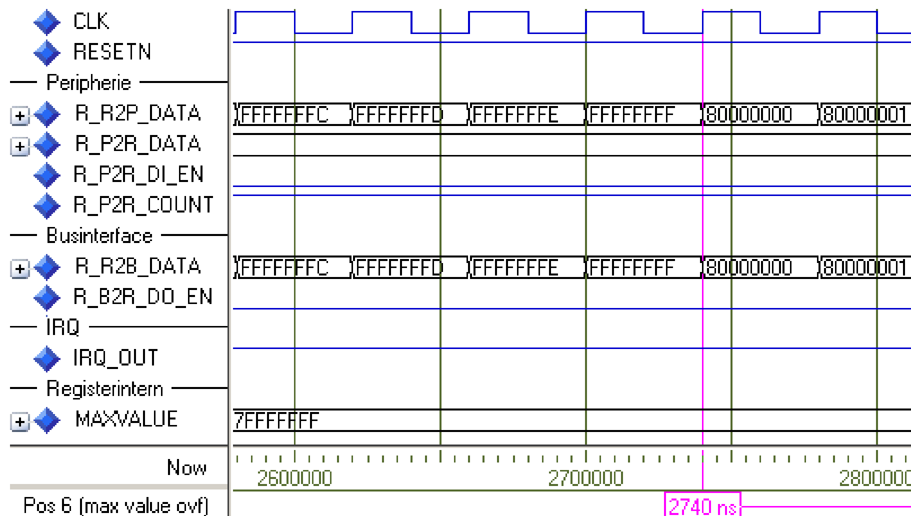


Abbildung 5.8: Timing Simulation "Überlauf am Maximum" des Round_Rotating_Counterregisters

Abschließend ist noch ein Test abgebildet (Abb. 5.9). Bei diesem Test ist Pos. 8 entscheidend. Es wird gleichzeitig gelesen und gezählt. Dabei muss der Counter hinterher den Wert von 1 haben, da der Zählimpuls sonst verloren geht.

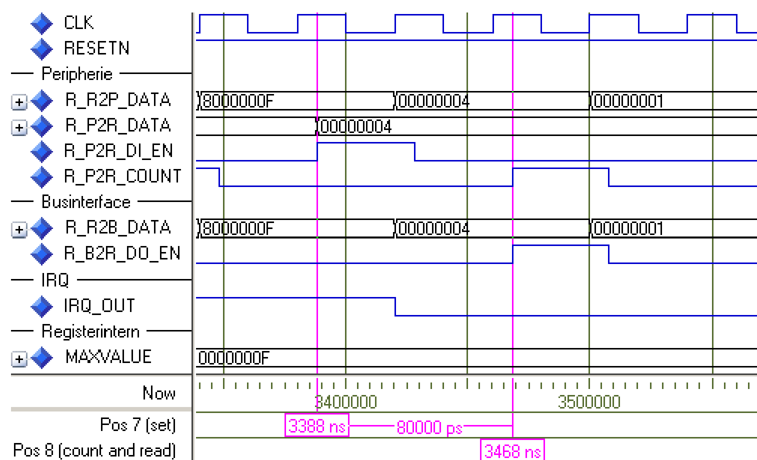


Abbildung 5.9: Timing Simulation "Schreiben/Schreiben - Lesen/Zählen" des Round_Rotating_Counterregisters

Folgende Tests wurde mit dem Round Rotating Counterregister zusätzlich zu den hier abgebildeten durchgeführt:

- alle hier abgebildeten Tests mit deaktiviertem Reset on Read
- alle Tests der Testbench für das P2B_Dataregister
- alle Tests der Testbench für das P2B_Bufferregister

Die Testbenches des P2B_Dataregister und P2B_Bufferregister mit weiteren Informationen befinden sich auf der beigefügten CD.

5.4 Testbench des APB-Interfaces

Wie bereits am Anfang dieses Kapitels erwähnt, wird hier nur eine einfache Simulation des APB-Interfaces durchgeführt.

Die Adresse des APB-Interface kann gemäß des Abschnittes Globale Definitionen (Kap. A.4) aus der μPI Benutzerbeschreibung (Kap. A) geändert werden. Dies wurde während der Tests auch getan, um zu erkennen, ob das APB-Interface an allen gültigen Adressen korrekt arbeitet. Bei den hier gezeigten Tests ist die Startadresse des APB-Interfaces auf 0x00 gesetzt.

Der erste Test zeigt einen Schreibzugriff auf das achte Register, welches an der Adresse 0x1C liegt. Das Verhalten der APB-Bridge wird gemäß des AMBA 3 APB Protokoll (Kap. 3.1.1) simuliert. Innerhalb des APB-Interfaces wird das Signal PWDATA ohne Verzögerung an den Ausgang B_B2R_DATA weitergereicht, womit es von allen Registern gelesen werden kann. Register reagieren jedoch nur, wenn das Data-In-Enable Signal des jeweiligen Registers gesetzt wird. Im zweiten Takt des Schreibvorganges - der Zugriffsphase - wird dieses Data-In-Enable Signal des achten Registers gesetzt, womit dieses Register die angelegten Daten übernimmt. Ebenfalls wird der APB-Bridge mit dem Setzen des Signales PREADY signalisiert, dass der Schreibvorgang abgeschlossen ist. Mit dem nächsten Takt nehmen sowohl die APB-Bridge als auch das APB-Interface alle Signale zurück. Die Abbildung 5.10 zeigt den kompletten Schreibvorgang.

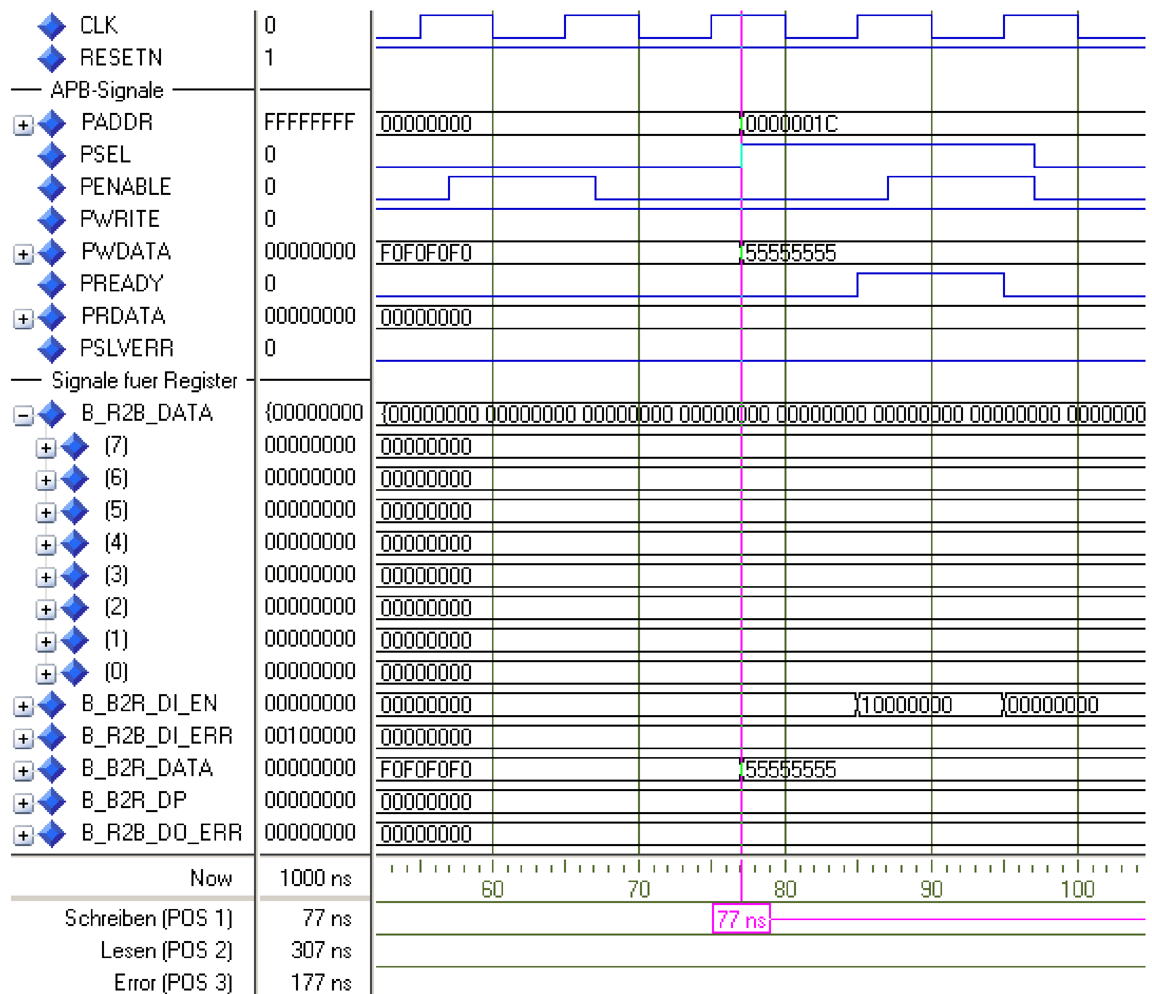


Abbildung 5.10: Simulation des APB-Interfaces - Schreiben

Der zweite hier gezeigte Test des APB-Interfaces stellt einen Lesevorgang des vierten Registers dar, welches an Adresse 0x0C liegt. Die Signale zum Lesen durch die APB-Bridge werden wieder gemäß des AMBA 3 APB Protokoll (Kap. 3.1.1) simuliert. Der Wert für das vierte Register wird für den Test auf 0x55555555 gesetzt, um das Ergebnis zu veranschaulichen. Der Wert des Registers liegt während der Zugriffsphase an PRDATA an und kann von der APB-Bridge zur weiteren Verarbeitung an den Prozessor gesendet werden. Genau wie bei einem Schreibvorgang auf ein Register setzt das APB-Interface das Signal PREADY, um der APB-Bridge zu signalisieren, dass es fertig ist. Für das ausgelesene Register wird das Data-Processed Signal gesetzt, womit ein Reset-on-Read fähiges Register weiß, dass es ausgelesen wurde. Die Abbildung 5.11 zeigt den eben beschriebenen Vorgang.

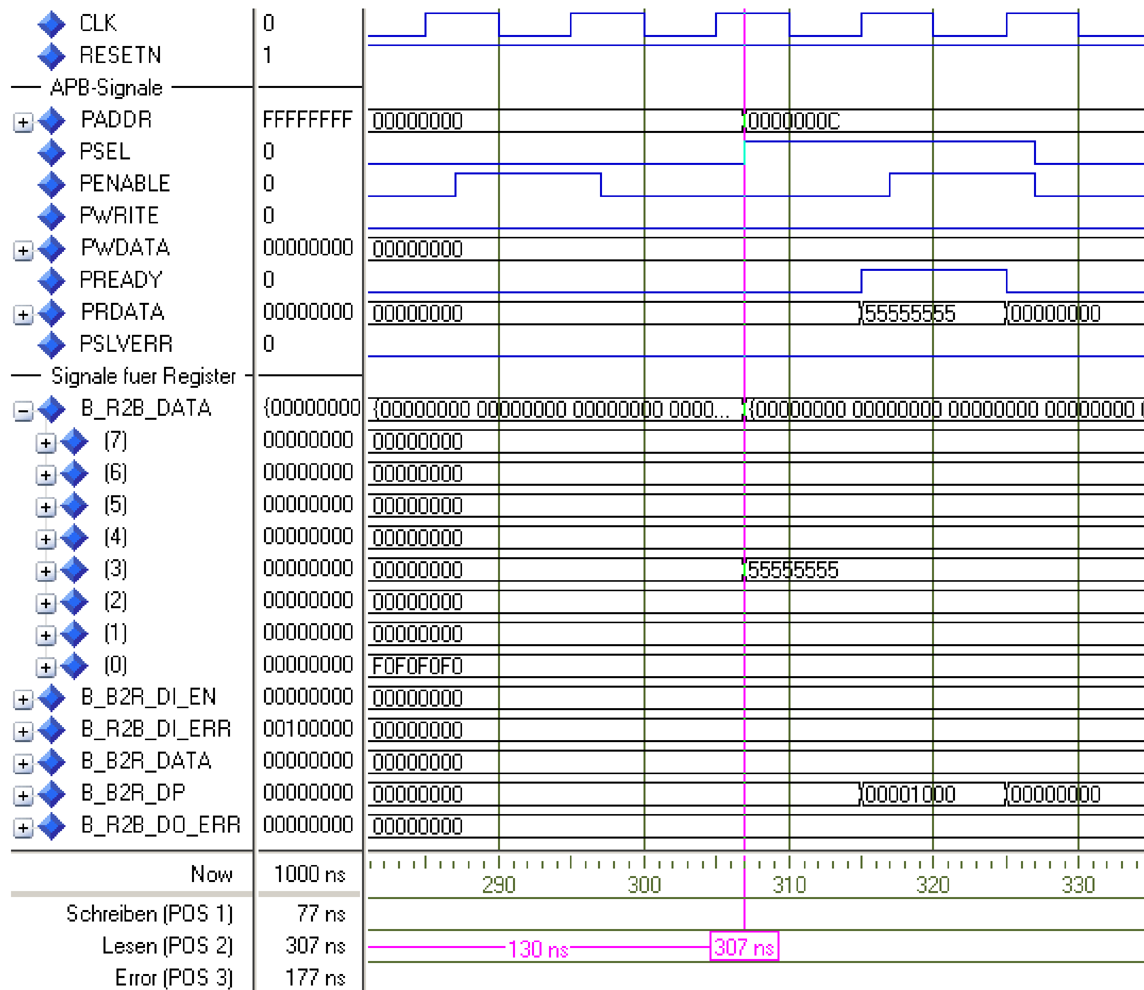


Abbildung 5.11: Simulation des APB-Interfaces - Lesen

Abschließend wird noch ein Test gezeigt, bei dem es zu einem Fehler kommt, den das APB-Interface auffängt und an die APB-Bridge meldet. In der Simulation wird versucht eine zu große Adresse zu beschreiben. Es sind acht Register vorhanden. Die höchste gültige Adresse ist somit 0x1C. Der Schreibvorgang erfolgt aber auf die Adresse 0xF0. Das APB-Interface erkennt diesen Fehler und setzt PSLVERR und PREADY in der Zugriffsphase. Eine weitere Verarbeitung durch das APB-Interface erfolgt nicht. Die Abbildung 5.12 zeigt den eben beschriebenen Vorgang.

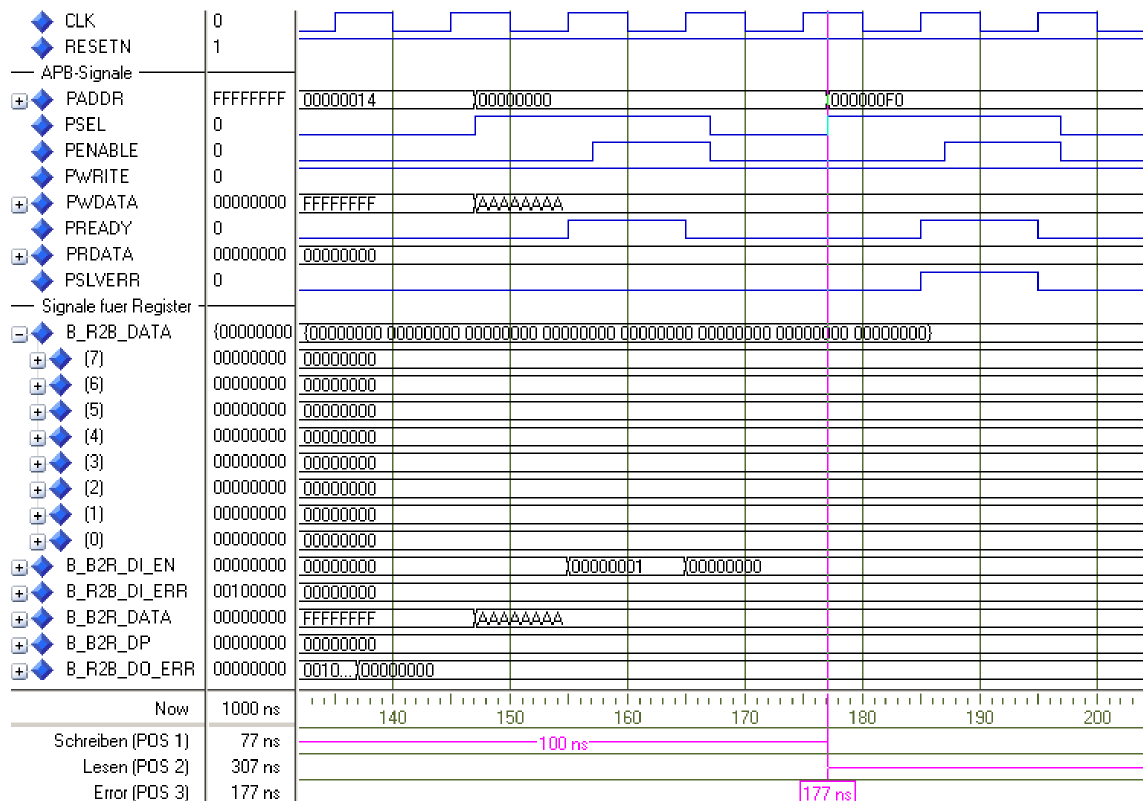


Abbildung 5.12: Simulation des APB-Interfaces - Fehler

Mit dem APB-Interface wurden folgende, einschließlich der oben abgebildeten Tests durchgeführt:

- lesen von einer zu kleinen Adresse
- lesen von einer zu großen Adresse
- lesen von einer nicht durch vier teilbaren Adresse
- lesen vom ersten verfügbaren Register
- lesen vom letzten verfügbaren Register
- lesen eines beliebigen Registers
- weglassen des PENABLE Signales bei einem Lesezugriff
- überprüfen, ob Fehlerrückmeldung der Register beim Lesen durchgeleitet werden
- alle eben genannten Tests, wobei schreibend auf die Register zugegriffen wird
- mehrfaches Durchführen aller aufgeführten Tests, wobei die Adresse des APB-Interfaces variiert wurde (spez. Anfang und Ende des Adressbereiches)

5.5 1024 Register Test

Eine der Anforderungen an diese Arbeit ist es, dass für einen Peripheriebaustein bis zu 1024 Register erzeugt werden können.

Bei diesem Test kommt es nicht darauf an, ihn in Hardware zu testen, sondern vielmehr zu erfahren, ob das Synthesetool die große Menge an Registern verarbeiten kann. Da es hier nur um die Synthese und das Mapping geht, muss der für die Synthese verwendete FPGA nicht vorhanden sein, sondern nur genug Platz für die Hardware bieten.

Für diesen Test wurde ein Virtex 5 von Xilinx gewählt. Das Synthesetool ist wie in der gesamten Bachelorarbeit ISE 10.1.03 von Xilinx.

Um diesen Test erfolgreich durchzuführen, ist es erforderlich eine Peripherie zu implementieren, die nicht mehr Ausgänge benötigt, als der für diesen Test gewählte FPGA bereitstellt, da die Synthese auf Grund von mangelnden Ein- / Ausgängen sonst frühzeitig abbrechen würde. Würde man nur die 1024 Register ohne Nachgeschaltete ein Peripheriemodul synthetisieren, würde dies 32768 (32 * 1024) I/Os am FPGA alleine für die Datenleitungen der Register benötigen. Es muss also ein Peripheriemodul implementiert werden, welches die Anzahl der I/Os auf eine im FPGA verfügbare Anzahl reduziert. Ebenfalls müssen alle Register im Peripheriebaustein verwendet werden, so dass einzelne Register bei der Optimierung nicht wegfallen. Hierfür werden alle Bits aller Register mittels XOR verknüpft. Bei der Verknüpfung der XORs werden jeweils alle 32 Bits eines Register miteinander verknüpft (xor.vhd). Das Ergebnis der Register wird dann in zwei Stufen zu je 32 Bit wieder miteinander verknüpft, so dass am Ende ein einzelnes Bit übrig bleibt.

Alle Register sind B2P_Dataregister. Der Einfachheit halber wurde der Generator so angepasst, dass er die eingestellte Anzahl an Registern erzeugt, ohne dass ein Array vorhanden ist, welches vorgibt, welcher Registertyp erzeugt werden soll.

5.5.1 Testergebnis

Das Synthesetool lief bei diesem Test ohne abzubrechen erfolgreich durch. Im Gegensatz zu allen anderen in dieser Bachelorarbeit erzeugten Schaltungen dauerte die Synthese allerdings erheblich länger. Aus dem Ausschnitt des Synthesereports (Abb. 5.13) ist zu entnehmen, dass die Synthese über eine Stunde und zehn Minuten dauerte.

```
Total REAL time to Xst completion: 4208.00 secs
```

Abbildung 5.13: 1024 Register Test, 1

Der folgende Auszug (5.14) zeigt einen weiteren Ausschnitt aus dem Synthesereport. Es finden sich die 1024 32-Bit Register für die Datenregister wieder. Bei den 1057 XOR32 handelt es sich um XORs mit 32 Bit am Eingang und einem Ausgangsbit. Die Anzahl von 1057 kommt durch die mehrstufige Verschachtelung zustande. Dabei fallen 1024 XORs auf die erste Stufe - jeweils 32 Bit des Registers - 32 auf die zweite und 1 XOR auf die letzte Stufe. Ein 32-BitRegister und die beiden Vergleichsoperatoren gehören zum APB-Interface. Die 1-Bit-Register sind ebenfalls zur Ansteuerung der Register nötig.

```
===================================================
HDL Synthesis Report

Macro Statistics
# Registers : 4099
1-bit register : 3074
32-bit register : 1025
# Comparators : 2
32-bit comparator greatequal : 1
32-bit comparator lessequal : 1
# Xors : 1057
1-bit xor32 : 1057
===================================================
```

Abbildung 5.14: 1024 Register Test, 2

6 SPI (Beispielapplikation)

Das Serial Peripheral Interfaces, welches für die Verbindung von ICs auf einer Platine oder in einem Gerät gedacht ist, ist ein offener Standard. In diesem Kapitel wird die Funktionsweise und anschließend die Implementation des SPIs mit Hilfe des μPI beschrieben.

6.1 SPI Beschreibung

Wie bereits erwähnt, handelt es sich bei dem SPI nicht um einen festen Standard, wie beispielsweise dem I2C Bus. Der Bus wurde von Motorola entwickelt, allerdings wurde er nicht bis zum letzten Detail festgelegt, wodurch es mehrere Unterschiede in Geräten, die das SPI nutzen, gibt. Da es keinen festen Standard gibt, ist für das SPI keine DIN, RFC, IEEE oder andere Norm vorhanden. Dadurch ist es zwar nicht möglich, auf das SPI Lizenzgebühren zu erheben, aber Informationen für die Implementation lassen sich nur aus Datenblättern anderer Bausteine gewinnen. Die Informationen für diese Bachelorarbeit wurden aus [Atmel Corporation, 2008, S. 168 bis 176] und Schwerdtfeger [2000] gewonnen.

Allgemein gilt, dass es sich um einen seriellen Bus handelt, bei dem die Daten Full-Duplex übertragen werden.

6.1.1 Signale des SPI

Das SPI wird auf Grund seiner vier Signale auch als four-wire Bus bezeichnet.

Damit der Master Daten zum Slave senden kann, ist zunächst einmal das ”Master out Slave in” (MOSI) Signal notwendig. Dieses Signal ist sowohl beim Master, als auch beim Slave vorhanden. Wie der Name es sagt, ist es beim Master als Ausgang und beim Slave als Eingang geschaltet. Durch dieses Signal werden die Daten seriell gesendet.

Damit der Slave dem Master Daten senden kann, wird als zweites Signal das ”Master in Slave out” (MISO) Signal benötigt. Es ist ebenso wie MOSI beim Master und beim Slave vorhanden, allerdings ist es beim Master ein Eingang und beim Slave ein Ausgang. Über MISO werden die Daten ebenfalls seriell übertragen. Für die Slaves ist es wichtig, dass das MISO-Signal als Tristate ausgeführt ist, da es ohne zusätzliche Logik sonst bei mehreren Slaves zu undefinierten Zuständen auf dem Bus kommt. Auch würden die Ausgangstreiber der Chips bei gegensätzlichen Signalen auf dem Bus durchbrennen.

Als alternative Bezeichnung für MOSI und MISO gibt es bei einigen Geräten auch ”Serial Data Out” (SDO) und ”Serial Data In” (SDI). Dabei wird nicht die Funktion des Gerätes, sondern nur die Datenrichtung des jeweiligen Signales angegeben.

Zur Synchronisierung des Masters und Slaves wird eine Clockleitung verwendet. Diese wird ”Serial Clock” (SCK) genannt und wird immer vom Master vorgegeben.

Beim SPI werden die Slaves nicht durch eine Adresse, sondern durch einzelne Signale vom Master zum Slave identifiziert. Zur Auswahl eines Slaves besitzt der Master für jeden Slave eine eigene ”Slave Select” (SS) Leitung. Diese ist low aktiv und heißt manchmal auch "Chip Select"(CS).

Eine schematische Darstellung eines Masters mit drei Slaves sieht dann wie in Abbildung 6.1 aus.

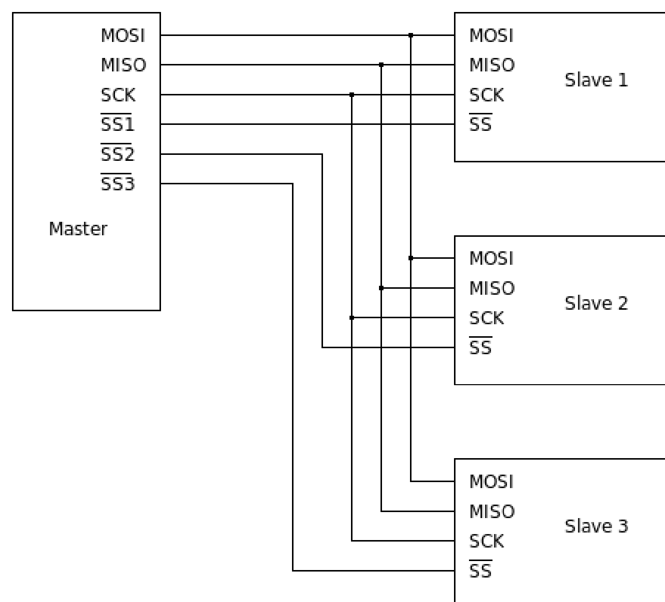


Abbildung 6.1: SPI Master mit 3 Slaves

Das Signal MISO wird von jedem Slave beschrieben, wenn er zum Senden aufgefordert wird. In der Zeit in der ein Slave nicht sendet, muss er seinen MISO-Ausgang hochohmig halten, damit es nicht zu Störungen auf Bus und Zerstörungen der Ausgangstreiber kommt. Wenn kein Slave sendet ist der Zustand dieses Signales undefiniert. Damit der Eingang vom Master auf Grund des offenen Einganges nicht zerstört wird, ist es erforderlich einen Pull-up oder ähnliche Maßnahmen gegen undefinierte Signale zu installieren. Das gleiche gilt für die Clock und das MOSI Signal, wenn der Master diese bei Systemstart hochohmig schaltet, beispielsweise, weil er auch als Slave arbeiten kann und immer im Slavemodus startet.

6.1.2 SPI Datenübertragung

Die Datenübertragung wird immer vom Master eingeleitet. Der Slave kann dem Master nicht mitteilen, wenn neue Daten zum Senden bereit liegen.

Die Datenübertragung beginnt immer, indem der Master das Slave Select Signal vom Slave, mit dem er kommunizieren will, auf low zieht. Danach senden Master und Slave mit jedem Takt (SCK) ein Bit, bis der Master das Select Signal wieder auf high zieht. Im SPI-Standard ist nicht festgelegt, ob die Daten zuerst mit dem MSB oder LSB übertragen werden. Dies muss selbst festgelegt werden, sofern ein fertiges Gerät dieses nicht fest vorgibt. Auch ist nicht festgelegt, wie viele Daten auf einmal übertragen werden. Eine Datenübertragung kann byteweise erfolgen, es können aber auch mehrere hundert Bytes auf einmal übertragen werden, ohne dass das SS-Signal zwischendurch high wird. Wünscht der Benutzer eine Fehlerprüfung der übertragenen Daten, so muss er dies in einem höherem Protokoll selbst tun. Das SPI bietet keine Fehlerprüfung über ein Parity-Bit oder eine CRC-Checksum.

Die folgende Grafik (Abb. 6.2) zeigt eine Datenübertragung per SPI. Dabei werden die Daten vom Master und vom Slave immer auf der steigenden Taktflanke auf den Bus gelegt und auf der fallenden werden sie eingelesen.

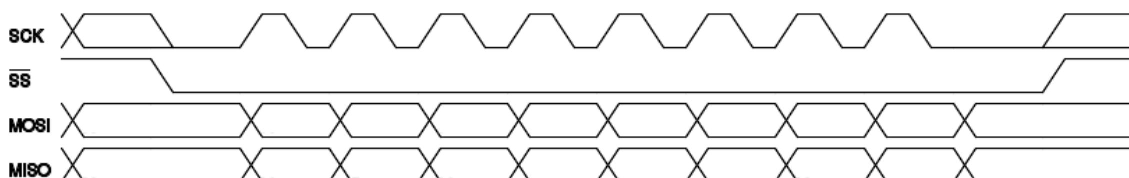


Abbildung 6.2: Datenübertragungsbeispiel bei SPI

Neben der Möglichkeit Daten mit LSB oder MSB zuerst zu übertragen, gibt es noch zwei weitere Einstellungen, die von Motorola nicht festgelegt wurden. Dabei handelt es sich um den Zustand der Clockleitung im Ruhezustand und, ob Daten bei der ersten oder zweiten Taktflanke gelesen werden. In allen analysierten Datenblättern finden sich die Einstellungen CPOL (”Clock Polarity”) und CPHA (”Clock Phase”), um genau diese Einstellungen vorzunehmen. CPOL gibt den Pegel der Clock im Ruhezustand an. CPHA gibt an, ob die Daten auf dem Bus bei der ersten oder zweiten Taktflanke vom Master und Slave eingelesen werden.

Die folgende Grafik (Abb. 6.3 [Atmel Corporation, 2008, S. 176]) zeigt die Datenübertragung beim SPI, wenn CPHA auf 0 gesetzt ist. Die Clock ist in der ersten und zweiten Zeile des Diagramms. Einmal wurde CPOL dabei auf low und einmal auf high gestellt. Der Samplepoint ist durch CPHA festgelegt und verdeutlicht die Position, an der Master und Slave die Daten vom Bus einlesen. Da der erste Samplepoint bereits auf der ersten Taktflanke liegt, muss das erste Bit bereits mit der Anwahl des Slaves angelegt werden. Wichtig ist, auch hierbei zu beachten, dass nicht festgeschrieben ist, ob das LSB oder das MSB zuerst übertragen werden.

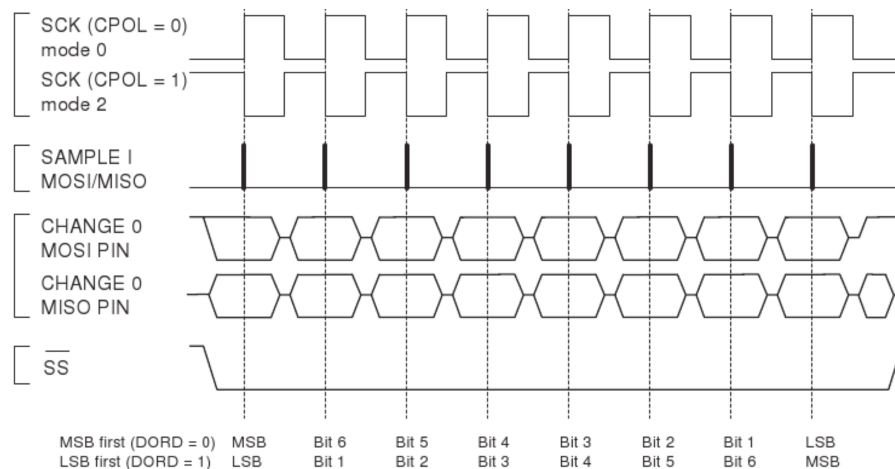


Abbildung 6.3: SPI Datenübertragung mit CPHA = 0

Nachdem die obige Grafik (Abb. 6.3) das Verhalten bei CPHA gleich 0 zeigt, verdeutlicht die nun folgende Grafik (Abb. 6.4 [Atmel Corporation, 2008, S. 176]) den Ablauf einer SPI-Datenübertragung mit CPHA gleich 1. Wieder sind beide Einstellungen für CPOL in der Grafik vorhanden und der Samplepoint gibt den Zeitpunkt zum Einlesen der Daten an. Ebenfalls ist nicht angegeben, ob LSB oder MSB zuerst übertragen werden.

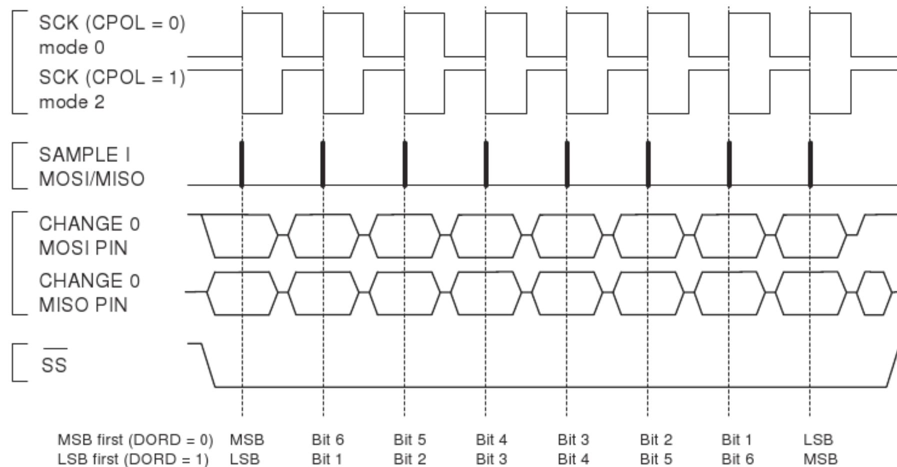


Abbildung 6.4: SPI Datenübertragung mit CPHA = 1

Bei der Implementation eines SPI Gerätes ergeben sich die oben aufgeführten Freiheiten, die als SPI Master berücksichtigt werden müssen, um mit allen auf dem Markt verfügbaren SPI Slaves kommunizieren zu können. Diese sind - nochmals zusammengefasst - die Clock Phase (CPHA), die Clock Polarity (CPOL), die Datenrichtung (LSB <-> MSB) und die Anzahl Bytes, die auf einmal übertragen werden.

6.2 SPI Implementation

Nachdem das μPI bereits beschrieben wurde und das SPI auch bekannt ist, wird hier nun die Implementation des SPI mit Hilfe des μPI beschrieben. Das μPI nimmt dem SPI dabei die Kommunikation mit dem Prozessor ab und stellt Register mit unterschiedlichen Fähigkeiten bereit. Das folgende Bild (Abb. 6.5) zeigt eine schematische Darstellung der Komponenten. Man sieht, dass das μPI dabei Teil des SPI ist und später von außen nicht mehr erkannt werden kann, ob eine Peripherie das μPI benutzt oder nicht.

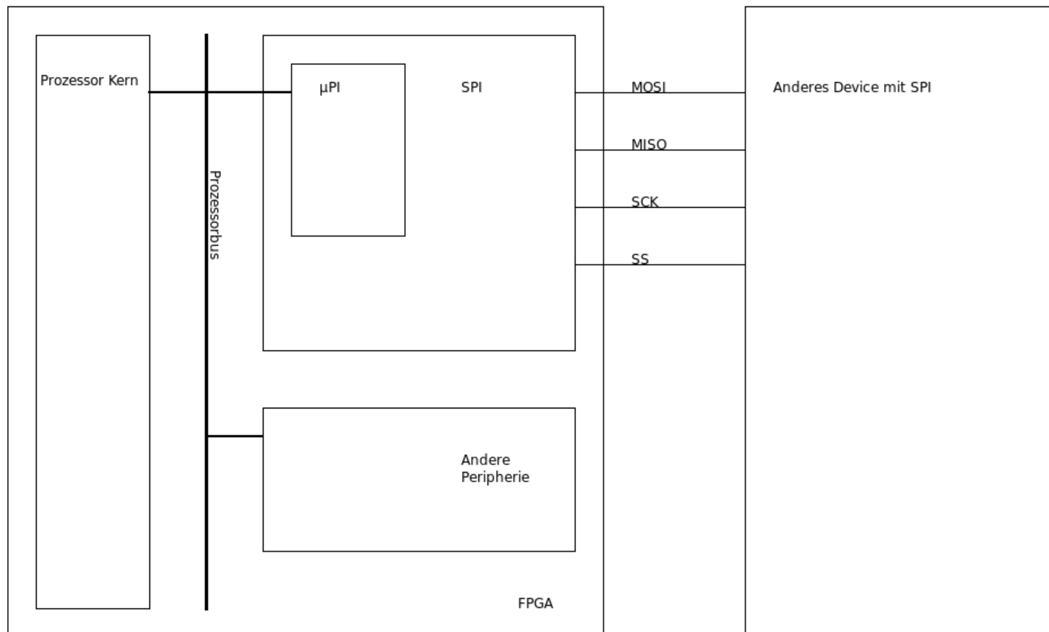


Abbildung 6.5: Schematische Darstellung eines FPGAs mit μPI-SPI

Alle "Allgemeine Implementationsentscheidungen" (4.1) die für das μPI getroffen wurden, werden auch bei der Implementation des SPIs verwendet.

6.2.1 Verwendete Register

Für die unterschiedlichen Aufgaben bei der Kommunikation zwischen SPI und Prozessor werden die jeweils passenden Register des μPI verwendet.

Die folgende Tabelle (6.1) bietet zunächst einen Überblick über die verwendeten Register und deren Adressen innerhalb des SPIs. Die Register werden in den folgenden Unterkapiteln noch genauer beschrieben.

Typ	Addressoffset	Kurzbeschreibung
Einfaches Kontrollregister	0x00	Bietet grundlegende Konfiguration
Reset on Read Kontrollregister	0x04	Sendestart im Mastermodus
Einfaches Kontrollregister	0x08	Taktteiler im Mastermodus
Einfaches Kontrollregister	0x0C	Anzahl zu versendende Bits im Mastermodus
Einfaches Statusregister	0x10	Sende- / Empfangspufferstatus und Übertragungsstatus
Impulse Statusregister	0x14	Verfeinerter Übertragungsstatus mit Interruptmöglichkeit
P2B_Bufferregister	0x18	Empfangspuffer
B2P_Bufferregister	0x1C	Sendepuffer

Tabelle 6.1: SPI Register Map

6.2.1.1 Kontrollregister 1

Das erste Kontrollregister, welches auch die niedrigste Adresse im SPI hat, ist ein einfaches Kontrollregister. Mit diesem Register setzt der Benutzer die in der folgenden Tabelle (6.2) aufgeführten Einstellungen.

Name	Bit(s)	Beschreibung
SLV_CS	0 bis 3	Wählt die Slaves aus, mit denen kommuniziert wird
MSB_FIRST	28	Wenn high, wird das MSB eines Bytes zuerst gesendet
CPOL	29	Gibt die Clock Polarity an (siehe 6.1.2)
CPHA	30	Gibt die Clock Phase an (siehe 6.1.2)
IS_MASTER	31	Wenn high, arbeitet das Devices als Master

Tabelle 6.2: SPI Kontrollregister 1

Alle sich in diesem Register befindenden Bits sollen sich nur nach Einstellung durch die Software ändern. Es wird hierfür lediglich der einfachste Typ von Register benötigt. Diese Aufgabe könnten das einfache B2P_Datenregister und das einfache Kontrollregister übernehmen. Da es sich hier um Steuersignale und nicht um Datensignale handelt, wird das Kontrollregister verwendet.

6.2.1.2 Kontrollregister 2

Das zweite Register ist ein Reset on Read Kontrollregister. Schreibt man eine ”Eins” in das LSB und das Device arbeitet als Master, fängt das SPI mit allen bisher gemacht Einstellungen an zu senden. Dieses Register wird nach Sendebeginn vom SPI zurückgesetzt. Der Vollständigkeit halber folgt hier die Tabelle (6.3) für das Register.

Name	Bit	Beschreibung
START_SEND	0	Beim Beschreiben mit Eins beginnt das SPI mit dem Sendevorgang, wenn es im Mastermodus ist

Tabelle 6.3: SPI Kontrollregister 2

Damit das Register nach dem Übertragungsstart von der Peripherie zurückgesetzt werden kann, ist hier ein Register mit Reset on Read Funktionalität notwendig. Dies wird in Zusammenhang mit einem Kontrollaufgaben nur von dem Reset on Read Kontrollregister erfüllt.

6.2.1.3 Kontrollregister 3 (Taktteiler)

Das dritte Kontrollregister ist ein einfaches Kontrollregister und ebenfalls nur im Mastermodus von Bedeutung. Dieses Register stellt den Takt für SCK ein. Dabei gilt folgende Formel:

$F_{SCK} = \frac{F_{Dev_SPI}}{Controlreg3+1}$

Der maximale Takt für das SPI ist auf ein Viertel des Taktes des Devices beschränkt. Da jeweils Zeit bleiben muss, um gesendete Daten aus dem Sendepuffer zu löschen bzw. empfangene Daten in den Empfangspuffer zu schreiben.

Der Taktteiler soll sich nur auf Anforderung der Software ändern und soll auch über mehrere Übertragungen hinweg erhalten bleiben können. Es wird damit keine Reset on Read Funktionalität gewünscht. Ein einfaches Kontrollregister erfüllt diese Aufgabe hinreichend.

6.2.1.4 Kontrollregister 4 (Transmit Counter)

Das vierte Kontrollregister, welches ebenfalls ein einfaches Kontrollregister und nur im Mastermodus von Bedeutung ist, gibt an, wie viele Bits gesendet bzw. empfangen werden sollen, bevor das Slave Select für den ausgewählten Slave wieder high wird. Sollen mehr Daten versendet werden, als der Datenpuffer an Daten halten kann oder es liegen zu Sendebeginn noch nicht alle Daten vor, können diese auch nach Sendebeginn in den Puffer gefüllt werden. Für das rechtzeitige Auffüllen des Sendepuffers bzw. Auslesen des Empfangspuffers ist der Benutzer selbst verantwortlich. Es muss aber mindestens ein Datensatz bei Sendebeginn bereit liegen, da sonst nicht definierte Daten versendet werden.

Wenn bei der Kommunikation mit der Peripherie immer wieder die gleichen Befehle ausgeführt werden, wie zum Beispiel das Auslesen eines Temperatursensors, dann ist die Anzahl der zu übertragenden Bits immer gleich. Es ist also wünschenswert, dass diese Einstellung erhalten bleibt. Ändert sich die Anzahl der zu übertragenden Bits immer wieder, muss dies auch immer wieder von der Software eingestellt werden. Die Hardware kann diesen Wert nicht voraussehen. Ein einfaches Kontrollregister erfüllt die Aufgabe des Transmit Counters dabei am besten.

6.2.1.5 Statusregister 1

Das fünfte Register ist ein einfaches Statusregister, welches den aktuellen Zustand über den Sende- und Empfangspuffer bereit hält. Außerdem gibt es an, ob gerade eine Übertragung im SPI statt findet. Die genauen Bits sind der Tabelle 6.4 zu entnehmen.

Name	Bit	Beschreibung
SEND_BUFFER_EMPTY_SIG	0	Ist gesetzt, wenn der Sendepuffer leer ist
RECV_BUFFER_FULL_SIG	1	Ist gesetzt, wenn der Empfangspuffer voll ist
TRANSMIT_SIG	2	Ist gesetzt, wenn eine SPI-Übertragung statt findet

Tabelle 6.4: SPI Statusregister 1

Mit diesem Register soll der Software die Möglichkeit gegeben werden, zu jeder Zeit schnell einen Überblick über die wichtigsten Stati zu erlangen. Da alle anliegenden Daten im SPI selbst noch einmal vorhanden sind, wird ein Register benötigt, das von außen anliegende Daten ohne großen Aufwand übernimmt. Dafür wird hier das einfache Statusregister gewählt. Interrupts werden hier nicht benötigt.

6.2.1.6 Statusregister 2

Das sechste Register, bei dem es sich um ein Impulse Statusregister handelt, gibt verschiedene Fehler, Warnungen und Hinweise aus. Da das Impulse Status Register eine IRQ-Leitungen nach außen gibt, kann auf die Fehler, Warnungen und Hinweise mittels Interrupt Service Routine reagiert werden. Alle Werte bleiben so lange erhalten, bis die Ursache behoben und das Register ausgelesen wurde. Die einzelnen Fehler und Warnungen sind in der nachstehenden Tabelle (6.5) aufgeführt.

Name	Bit	Beschreibung
SEND_ERR	0	Ist gesetzt, wenn Fehler beim Senden entstanden sind, da keine Daten zum Senden vorhanden waren (Sendepuffer leer)
RECV_ERR	1	Ist gesetzt, wenn Fehler beim Empfangen entstanden sind, da kein Platz zum Speichern von empfangenen Daten vorhanden war (Empfangspuffer voll)
TRANSMIT_END	2	Ist gesetzt, wenn eine Übertragung beendet wurde
TRANSMIT_START	3	Ist gesetzt, wenn eine Übertragung begonnen wurde
BYTES_RECEIVED	4	Ist gesetzt, wenn neue Daten empfangen wurden und zum Auslesen bereit liegen
RECV_BUFFER_NEARLY_FULL	5	Ist gesetzt, wenn nur noch wenig Platz im Empfangspuffer ist

Tabelle 6.5: SPI Statusregister 2

Da es bei den Stati dieses Registers sehr hilfreich ist, wenn der Prozessor schnell etwas über deren Zustände erfährt, um möglichst schnell darauf zu reagieren, wurde hier ein Register mit Interruptmöglichkeit gewählt. Dies vermeidet ein Pollen des Registers. Der Zustand soll auch nach dem Ereignis noch ausgelesen werden können, deshalb wird hier ein Impulse Statusregister gewählt, welches auch die Möglichkeit eines Interrupts bietet. Da kein Bedarf nach negativer Logik besteht, wird nicht das negative Impulse Statusregister verwendet.

6.2.1.7 Empfangspuffer

Das siebte Register ist der Empfangspuffer. Dieses Register ist ein FIFO, der mehrere Datensätze speichern kann, ohne das sofort alle ausgelesen werden müssen. Das Verhalten im Master und Slave Modus unterscheidet sich.

Im Mastermodus werden bis zu 32 Bit an Daten pro Datensatz gespeichert. Wurden 32 Bits empfangen, werden die Daten gespeichert. Ist die Anzahl Bits bei einer Übertragung kein vielfaches von 32, werden die restlichen Bits nach dem Ende der Übertragung sofort im Empfangspuffer gespeichert. Es wird nicht bis zur nächsten Übertragung gewartet, um auf jeden Fall 32 Bit pro Datensatz voll zu haben. Dadurch können die Daten möglichst zeitnah ausgewertet werden und es kommen keine Daten aus unterschiedlichen Übertragungen, von vielleicht sogar unterschiedlichen Geräten, zusammen. Die Anzahl der Bits eines Datensatzes, die jeweils gültig sind, ist durch die Anzahl der versendeten Bits bekannt.

Im Slavemodus werden bis zu 24 Datenbits pro Datensatz gespeichert. Die acht restlichen noch freien Bits, werden mit Informationen über die empfangenen Daten aufgefüllt. Ebenso wie im Mastermodus gilt, dass Datensätze, die nach dem Ende einer Übertragung nicht voll sind, gespeichert werden. Am einfachsten erklärt den genauen Aufbau des Empfangspuffer im Slavemode die Tabelle 6.6.

Name	Bit(s)	Beschreibung
Datenbits	0 bis 23	Enthält die empfangen Daten
Datencounter	24 bis 28	Gibt an, wieviele Bits der empfangenen Daten gültig sind
TRANSMIT_END	30	Ist gesetzt, wenn dies das erste Datenpaket einer Übertragung ist
TRANSMIT_START	31	Ist gesetzt, wenn dies das letzte Datenpaket einer Übertragung ist

Tabelle 6.6: SPI Empfangspuffer im Slave Modus

Beim Empfangspuffer ist es vor allem wichtig, mehrere Datensätze auf einmal zu speichern. Welches der beiden peripherieseitig beschreibbaren Pufferregister gewählt wird, ist nur von der gewünschten Größe des Puffers und des zu Grunde liegenden FPGAs abhängig.

6.2.1.8 Sendepuffer

Das achte und letzte Register ist der Sendepuffer. In diesem werden die zu sendenden Daten hinterlegt, bevor sie über das SPI gesendet werden. Sowohl im Master als auch im Slave Modus werden alle 32 Bit zum Senden genutzt, es sei denn die Übertragung wird vor dem Versenden der 32 Bit beendet. In dem Fall, dass keine Vielfachen von 32 Bit gesendet wurden, werden die restlichen Daten Bits des Datenpakets verworfen und beim nächsten Sendevorgang wird das nächste Datenpaket verarbeitet.

Beim Sendepuffer ist es vor allem wichtig, mehrere Datensätze auf einmal zu speichern. Welches der beiden busseitig beschreibbaren Pufferregister gewählt wird, ist nur von der gewünschten Größe des Puffers und des zu Grunde liegenden FPGAs abhängig.

6.2.2 Einbinden des μPI

Welche Register des μPIs verwendet werden, ist festgelegt. Es wird nun die Generierung des μPIs für das SPI beschrieben. Alle Schritte zu Erstellung des μPI können in der μPI Benutzerbeschreibung (Anhang A) nachgelesen werden. Alle VHDL-Dateien befinden sich auf der beigefügten CD.

Als erster Schritt werden Anzahl, Art und Reihenfolge der Register in der Datei "defs.vhd" des μPIs festgelegt. Des weiteren wird die spätere Startadresse der Peripherie bestimmt. Im folgenden Quelltextauszug 6.6 aus der "defs.vhd" Datei werden alle nötigen Schritte zur Generierung gezeigt.

```
1 --Allgemeine Register - Definitionen
2 constant REG_WIDTH   : integer := 32;
3 constant REG_NUM     : integer := 8;
4
5 --APB-BUS Spezifische Definitionen
6 constant APB_ADDR    : integer := 0;
7
```

```
8  --Definitionen der zu verwendenden Register
9  --Enum mit den Namen aller verfuegbaren Register
10 type REG_NAMES is
11   (P2B_DATA, B2P_DATA, P2B_BUFFER, P2B_BUFFER_B,
12    B2P_BUFFER, B2P_BUFFER_B, CONTROL, ROR_CONTROL,
13    STATUS, ROR_STATUS, IMPULSE_STATUS, NEG_IMPULSE_STATUS,
14    RND_CNT, SAT_CNT, MAINIRQ);
15
16 --Arraydefinition, Type der Register mit Groesse aller Register
17 type REG_LAYOUT_TYPE_ARRAY is array
18   (REG_NUM -1 DOWNTO 0) of REG_NAMES;
19
20 --Arraydefinition, Parameter 1 als integer
21 type REG_LAYOUT_PAR1_ARRAY is array
22   (REG_NUM -1 DOWNTO 0) of integer;
23
24 --Arraydefinition, Parameter 2 als std_ulogic
25 type REG_LAYOUT_PAR2_ARRAY is array
26   (REG_NUM -1 DOWNTO 0) of std_ulogic;
27
28
29 --Array mit allen zu verwendenden Registern und Parametern
30 constant REG_LAYOUT_TYPE : REG_LAYOUT_TYPE_ARRAY
31   := (B2P_BUFFER, P2B_BUFFER, IMPULSE_STATUS, STATUS,
32      CONTROL, CONTROL, ROR_CONTROL, CONTROL);
33 constant REG_LAYOUT_PAR1 : REG_LAYOUT_PAR1_ARRAY
34   := (4, 4, 0, 0, 0, 0, 0, 0);
35 constant REG_LAYOUT_PAR2 : REG_LAYOUT_PAR2_ARRAY
36   := ('0', '0', '0', '0', '0', '0', '0', '0');
```

Abbildung 6.6: Quelltextauszug defs.vhd

Alle Konstanten im oben gezeigten Quelltextauszug 6.6 und dessen detailierte Beschreibungen befinden sich im Anhang A.4. Hier folgt nur eine kurze Erläuterung, um den Gesamtzusammenhang beim Erstellen eines auf dem μPI basierendem Peripheriebausteins zu verstehen. Die erste Konstante (REG_WIDTH; Zeile 2) definiert die Breite der Register. Diese kann nicht geändert werden, ohne das Anpassungen an einigen Teilen des μPIs vorgenommen werden müssen. In diesem Fall ist eine andere Breite als 32 Bit auch nicht gewünscht, da das SPI für einen 32 Bit ARM Prozessor gebaut wird. Die zweite Konstante (REG_NUM; Zeile 3) ist sehr viel wichtiger. Sie enthält, die Anzahl der zu erstelleneden Register. Wie aus Kapitel 6.2.1 zu entnehmen ist, werden acht Register für das SPI-Interface benötigt. Die

APB_ADDR in Zeile 6 kann für die jeweiligen Bedürfnisse eingestellt werden. Es wird damit die Startadresse des μPIs festgelegt. In Zeile 10 bis 14 sind alle Register definiert, die erzeugt werden können. Die Namen der Enums für die Register sind von den Namen der Register abgeleitet, lediglich das "REGISTER" am Ende fehlt. Die Definition der Arrays darunter (Zeile 16 - 26) erfolgt automatisch und benötigt keine Anpassung durch den Benutzer.

Im unterem Teil des Quellcodes (ab Zeile 29) wird festgelegt, welche Register erzeugt werden und welche Parameter diesen mitgegeben werden. Dabei ist zu beachten, dass diese - ebenso wie alle anderen Arrays und Vektoren im μPI mittels "DOWNTO" definiert sind, Register, die im Text am Ende der Registerdeklaration stehen haben die niedrigste Adresse. In diesem Fall ist ein einfaches Kontrollregister ("CONTROL") das Register mit der niedrigsten Adresse. Der Sendepuffer (B2P_BUFFER) hat die höchste Adresse im SPI. Bei der Synthese der Schaltung werden durch den VHDL-Code im Generator alle in diesem Array definierten Register erzeugt. Die beiden Arrays darunter (Zeilen 33/34 und 35/36) geben den Registern Parameter für die Erstellung mit. Der Generator setzt eventuell vorhandene Generics der Register mit den Werten aus diesen beiden Array. Welche Register über welche Generics verfügen und welchen Einfluss diese auf ein Register haben ist aus der μPI-Beschreibung im Anhang A.3 zu entnehmen. Im obigem Fall wird eine Pufferanzahl von 4 für beide Pufferregister angegeben. Alle anderen Register kommen mit den Standardeinstellungen aus.

Der wichtigste Schritt bei der Generierung eines μPIs für eine eigene Peripherie ist bereits getan. Jetzt müssen die Signale des μPIs mit denen des SPIs verbunden werden.

Das SPI ist in zwei Entities aufgeteilt, die innere enthält die eigentliche Funktionalität des SPIs, die äußere verbindet das SPI mit dem μPI und leitet sowohl die APB Signale des μPIs, als auch die Signale des SPIs nach außen weiter. Die äußere SPI Entity hat folgende Port-Map. Es sind alle Signale, die aus dem SPI nach außen geführt werden. Der Quelltext Auszug 6.7 zeigt die äußere SPI Entity, welche in der spi_main.vhd Datei definiert ist.

```
1  entity SPI_MAIN is
2    port (
3      CLK         : in std_ulogic;
4      RESETN      : in std_ulogic;
5
6      --Die folgenden Signale sind die Ein-
7      --und Ausgangssignale des SPI Interfaces
8      SLV_SELECT   : out std_ulogic_vector(NUM_SLAVES -1 downto 0);
```

```
9       SELECT_ME     : in std_ulogic;
10      DATA_OUT      : out std_ulogic;
11      DATA_IN       : in std_ulogic;
12      DATA_CLK      : inout std_logic;
13
14      --Die folgenden Signale sind mit dem
15      --jeweils gleichnamigen Signalen der APB bridge zu verbinden
16      PADDR       : in std_ulogic_vector
17                  ( work.DEFS.REG_WIDTH -1 DOWNTO 0 );
18      PSEL        : in std_ulogic;
19      PENABLE     : in std_ulogic;
20      PWRITE      : in std_ulogic;
21      PWDATA      : in std_ulogic_vector
22                  ( work.DEFS.REG_WIDTH -1 DOWNTO 0 );
23      PREADY      : out std_ulogic;
24      PRDATA      : out std_ulogic_vector
25                  ( work.DEFS.REG_WIDTH -1 DOWNTO 0 );
26      PSLVERR     : out std_ulogic;
27      --Ende APB Signale
28
29      IRQ_OUT     : out std_ulogic
30    );
31  end entity SPI_MAIN;
```

Abbildung 6.7: Quelltextauszug spi_main.vhd (1)

In der äußeren SPI Architektur werden die Signale von SPI und μPI miteinander verbunden. Der folgende Quelltextauszug 6.8 zeigt nur einen Teil der Signale. Alle Signale abzubilden, ist zu umfangreich. Im ersten Abschnitt sind Signale definiert, um die Signale aus der GENERATOR Komponente mit denen aus der SPI Komponente zu verbinden. Danach sind auszugsweise die beiden Komponenten vom GENERATOR und SPI abgebildet. Der Quelltextauszug, sowie alle folgenden in diesem Kapitel sind, sofern nicht anders angegeben, aus der spi_main.vhd Datei entnommen.

```
1  architecture SPI_MAIN_ARC of SPI_MAIN is
2    ----------------------------
3    --Signale auskommentiert
4    ----------------------------
5
6    signal STATUS_REG_2        : std_ulogic_vector
7                               (work.DEFS.REG_WIDTH -1 downto 0);
```

```
8     signal CONTROL_REG_1     : std_ulogic_vector
9                              (work.DEFS.REG_WIDTH −1 downto 0);
10    signal CONTROL_REG_2     : std_ulogic_vector
11                             (work.DEFS.REG_WIDTH −1 downto 0);
12    signal CONTROL_REG_2_DP  : std_ulogic;
13
14    ------------------------------------
15    --Weitere Signale auskommentiert
16    ------------------------------------
17
18    component GENERATOR
19    port (
20      --Start APB Signale
21      --APB Signale teils auskommentiert
22      PADDR : in std_ulogic_vector
23            ( work.DEFS.REG_WIDTH −1 DOWNTO 0 );
24      PSEL  : in std_ulogic;
25      --Ende APB Signale
26
27      --Signale die an die Peripherie weitergeleitet werden
28
29      --Signale teilweise auskommentiert
30
31      G_R2P_DATA      : out REG_WIDTH_VECTOR_ARRAY;
32      G_P2R_DP        : in STD_ULOGIC_ARRAY;
33
34      G_P2R_DATA      : in REG_WIDTH_VECTOR_ARRAY;
35    );
36    end component;
37
38    component SPI
39    port (
40      --Signale teils auskommentiert
41      DATA_OUT      : out std_ulogic;
42      DATA_IN       : in std_ulogic;
43
44      DATA_CLK      : inout std_logic;
45
46      --Folgend kommen die Busleitungen, ueber welches das Device
47      --seine Konfiguration und Befehle erhaelt
48
49      --Signale teilweise auskommentiert
50
51      STATUS_REG_2        : out std_ulogic_vector
```

```
52                               (work.DEFS.REG_WIDTH -1 downto 0);
53
54      CONTROL_REG_1            : in std_ulogic_vector
55                               (work.DEFS.REG_WIDTH -1 downto 0);
56      CONTROL_REG_2            : in std_ulogic_vector
57                               (work.DEFS.REG_WIDTH -1 downto 0);
58      CONTROL_REG_2_DP         : out std_ulogic;
59    );
```

Abbildung 6.8: Quelltextauszug spi_main.vhd (2)

Nachdem die Komponenten erstellt sind, werden die Signale der Komponenten im folgenden Quelltextausschnitt 6.9 mit den Signalen der äußeren SPI Schicht (SPI_MAIN) verbunden.

```
1  begin
2    DEVICE : SPI port map
3    (
4      --Signale nach aussen
5      --Signale teilweise auskommentiert
6      DATA_OUT      => DATA_OUT,
7      DATA_IN       => DATA_IN,
8      DATA_CLK      => DATA_CLK,
9
10
11     --Signale zum uPI
12     --Signale teilweise auskommentiert
13     STATUS_REG_1        => STATUS_REG_1,
14
15     CONTROL_REG_1       => CONTROL_REG_1,
16     CONTROL_REG_2       => CONTROL_REG_2,
17     CONTROL_REG_2_DP       => CONTROL_REG_2_DP,
18   );
19
20   BUS_INTERFACE : GENERATOR port map
21   (
22     --APB-Signale
23     --Signale teilweise auskommentiert
24     PADDR        => PADDR,
25     PSEL         => PSEL,
26     --Ende APB Signale
27
28
```

```
29      --Signale, die an die Peripherie weitergeleitet werden
30      --Signale teilweise auskommentiert
31      G_R2P_DATA      => G_R2P_DATA,
32      G_P2R_DP      => G_P2R_DP,
33
34      G_P2R_DATA      => G_P2R_DATA,
35    );
```

Abbildung 6.9: Quelltextauszug spi_main.vhd (3)

Nachdem bereits die Signale der GENERATOR und SPI Entity mit den Signalen der SPI_MAIN Architektur verbunden sind, muss nun (Auszug 6.10) innerhalb der SPI_MAIN Architektur dafür gesorgt werden, dass die Signale vom GENERATOR und SPI miteinander verbunden werden. Ein Auszug zeigt, wie das erste Statusregister, sowie das erste und zweite Kontrollregister miteinander verbunden werden. Beim zweiten Kontrollregister gibt es neben der Datenverbindung noch ein Data-Processed Signal, da es sich um ein Reset-on-Read Kontrollregister handelt.

```
1    G_P2R_DATA(5) <= STATUS_REG_2;
2
3    CONTROL_REG_1 <= G_R2P_DATA(0);
4    CONTROL_REG_2 <= G_R2P_DATA(1);
5    G_P2R_DP(1) <= CONTROL_REG_2_DP;
```

Abbildung 6.10: Quelltextauszug spi_main.vhd (4)

Das SPI ist nun mit dem μPI verbunden. Um dem Synthesetool die Arbeit zu erleichtern und Fehler bzw. Warnungen zu vermeiden, werden nicht benötigte Signale vom μPI nun noch auf einen Standardwert gesetzt (vgl Quelltextauszug 6.11). Als Beispiel werden hier die Data-Processed Signale des zweiten Statusregisters und des ersten Kontrollregisters auf "0" gesetzt. Beide Register benötigen dieses Signal nicht. Da der Generator aber für alle Register alle Signale erstellen muss, existieren diese Signale auch, wenn sie im μPI nie benutzt werden.

```
1    --nicht benoetigte signale
2    G_P2R_DP(0) <= '0';
3    G_P2R_DP(2) <= '0';
```

Abbildung 6.11: Quelltextauszug spi_main.vhd (5)

Das μPI stellt speziell für Interrupts zwei Signale zur Verfügung, welche zum einen die Signale Interruptausgänge der einzelnen Register und zum anderem die Interruptenable mit einem Kontrollregister verbindet. Wird wie im SPI kein Haupt-Interruptregister verwendet, sind diese Signale wie folgt (Auszug 6.12) zu beschalten.

```
1  --nur ein IRQ, kein Main-Irqregister verwendet
2  G_MAIN_IRQ_IN <= (others => '0');
3  G_MAIN_IRQ_EN <= (others => '0');
4
5  --Leite IRQ des Impulsestatusregister direkt nach aussen
6  IRQ_OUT <= G_IRQ(5);
```

Abbildung 6.12: Quelltextauszug spi_main.vhd (6)

Geht man davon aus, dass ein Haupt-Interruptregister erstellt wurde, verbindet man den Ausgang eines Kontrollregisters mit G_MAIN_IRQ_EN und die einzelnen Interruptausgänge der Register mit den Eingängen des Haupt-Interruptregisters. Der folgende Quelltext 6.13 zeigt diesen Vorgang. Das erste Register ist wie im SPI ein einfaches Kontrollregister und das sechste Register ein interruptfähiges Register, wie zum Beispiel das Impulsestatusregister wie im SPI.

```
1  --IRQ vom Impulsestatusregister
2  G_MAIN_IRQ_IN(0) <= G_IRQ(5);
3  --ungenutzte IRQs
4  G_MAIN_IRQ_IN(31 DOWNTO 1) <= (others => '0');
5
6  --Enable mit Controlregister verbinden
7  G_MAIN_IRQ_EN <= G_R2P_DATA(0);
8
9  --IRQ-Ausgang des Haupt-IRQ-Registers nach aussen
10 IRQ_OUT <= G_MAIN_IRQ;
```

Abbildung 6.13: Quelltextauszug Haupt-IRQ-Register

Die Erstellung des μPIs für das SPI, sowie die Anbindung des SPIs an das μPI sind abgeschlossen.

6.2.3 Interner Ablauf

Da das SPI entweder als Master oder als Slave arbeitet, ist auch der VHDL-Code so aufgeteilt. Auch wenn somit die Funktion des Sendens und Empfangens von Daten teils doppelt vorhanden ist, ist der VHDL-Code aber sehr viel einfacher lesbar, da beim Empfangen im Slavemode nur 24 Bit, im Mastermode allerdings 32 Bit auf einmal in den Empfangspuffer gespeichert werden. Auch wird im Mastermode die Clock selbst generiert und im Slavemode auf die Clock von außen reagiert. Diese und weitere Unterschiede machen den VHDL-Code nur schwer verständlich und wartbar, wenn man den Master- und Slavemode erst auf einer tieferen Ebene trennt.

Ein erstes grobes Aktivitätsdiagramm sieht wie folgt (Abb. 6.14) aus.

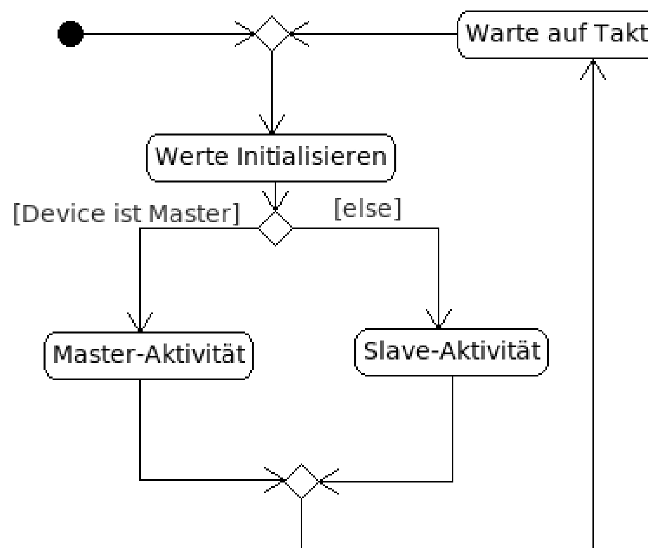


Abbildung 6.14: SPI Grundaktivität

Im Mastermode werden alle nötigen Teilaufgaben nacheinander abgelaufen. Zuerst wird überprüft, ob eine neue Übertragung gestartet werden soll. Danach wird die neue oder eine bereits laufende Datenübertragung vorangebracht, in dem die Bits weiter gezählt und an den entsprechenden Taktflanken die Daten eingelesen oder ausgegeben werden. Die Bits werden nur weiter gezählt, wenn der Taktteiler für SCK einen Überlauf hatte. Anschließend wird geprüft, ob 32 Bit gesendet wurden, wenn ja wird der Sendepuffer geleert und alle bis dahin empfangenen Daten gespeichert. Als letztes wird überprüft, ob alle zu übertragenden Daten versendet wurden. Sind keine Daten mehr zu versenden, wird der Sendezustand beendet. Die folgende Grafik (Abb. 6.15) veranschaulicht den Ablauf nochmals bildlich.

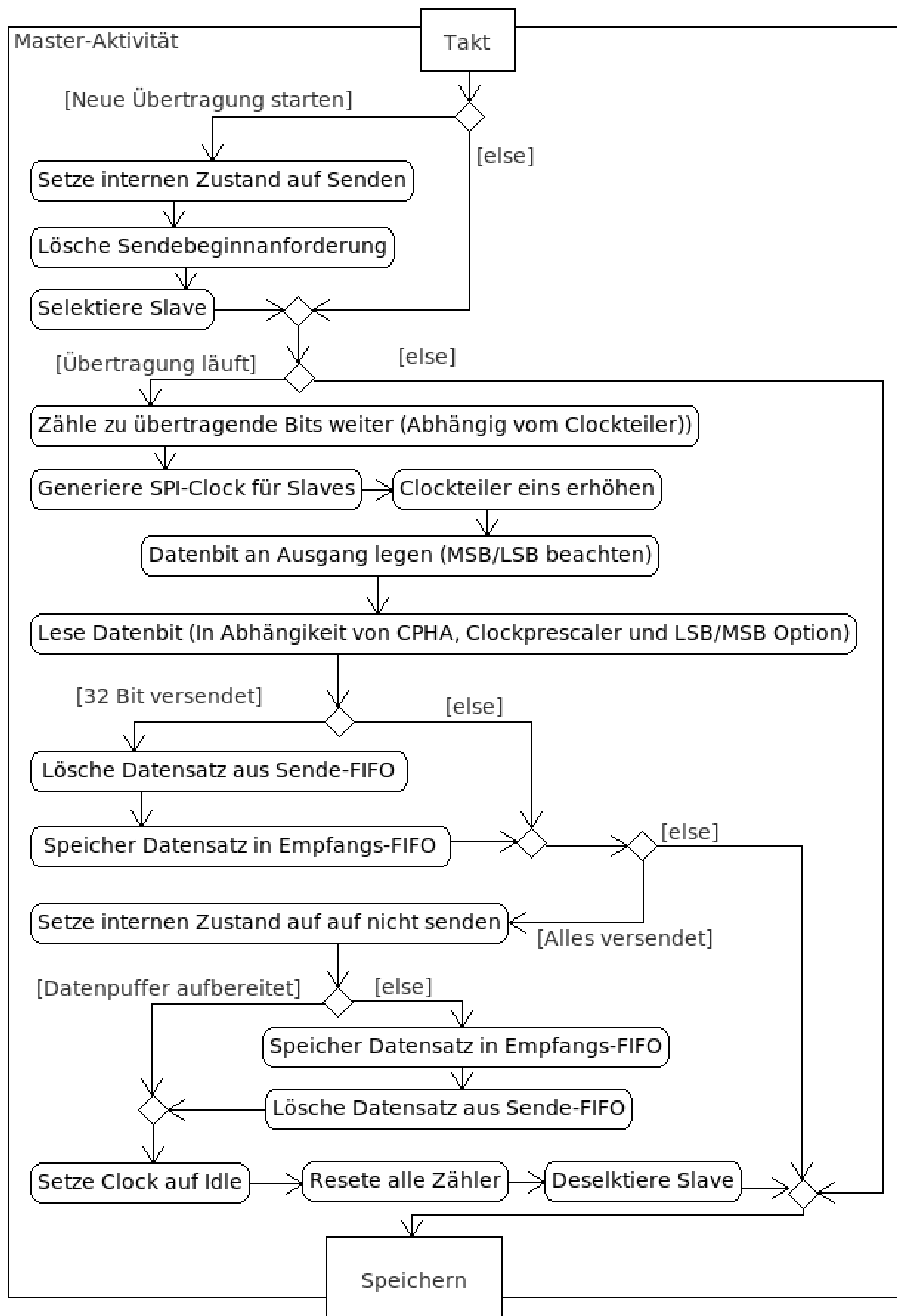


Abbildung 6.15: SPI Masteraktivität

Der grundsätzliche Ablauf im Slavemode sieht ähnlich aus, wobei der Beginn einer Übertragung und der Takt von außen vorgegeben wird. Es gibt also keinen internen Taktteiler. Dafür muss das Interface auf die externen Taktflanken reagieren. Das Interface weiß erst, ob eine Übertragung beendet ist, wenn das Slave Select Signal wieder high ist. Letzte Dinge, wie das Löschen des Sendepuffers und Speichern nicht verarbeiteter empfangener Daten können erst erfolgen, wenn das Interface deselektiert wurde. Ein Diagramm, welches den Ablauf zeigt folgt (Abb. 6.16).

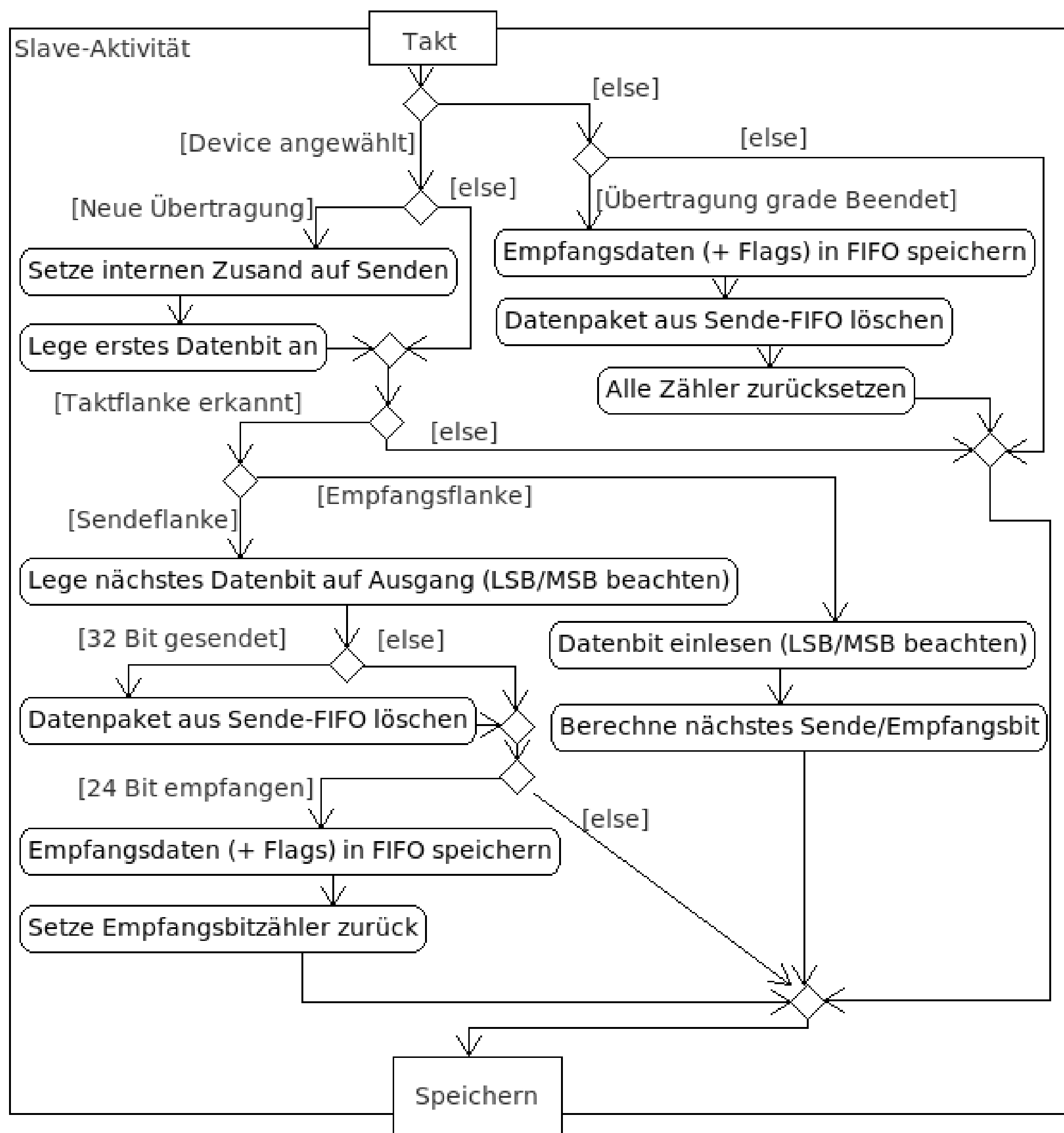


Abbildung 6.16: SPI Slaveaktivität

6.3 VHDL Simulationen

Genau wie alle Register und der APB-Bus wird auch das SPI simuliert. Zunächst wird eine einfache Simulation durchgeführt und anschließend eine Timingsimulation. Im Anschluss daran wird die Hardware aufgebaut (siehe Kapitel 6.4.1) und getestet (siehe Kapitel 6.4.4, 6.4.5 und 6.4.6). In diesem Kapitel werden zwei Ausschnitte der Timingsimulation gezeigt und eine Zusammenfassung über alle durchgeführten Tests gegeben.

6.3.1 Konfiguration als Master

Die unten stehende Grafik (Abb. 6.17) zeigt einen Schreibvorgang an Adresse 0x00 des SPI-Interfaces. Dabei wird der Wert 0x90000001 in das Register geschrieben. Der Beschreibung des ersten Kontrollregisters des SPIs (Kapitel 6.2.1.1) ist zu entnehmen, dass das Device dabei als Master konfiguriert ist und festgelegt wird, welchen Slave das Device beim Senden anspricht. Die Bits CPHA, CPOL und MSB_FIRST sind nicht gesetzt. Der Idle-Zustand der Clock ist damit low. Beim Senden werden die Daten bei der ersten Flanke übernommen und es wird das LSB zuerst versendet.

Abbildung 6.17: Timinigsimulation: Konfiguration als Master

Es fällt auf, dass es keine drei Signale für PRDATA, PWDATA und PADDR gibt. Statt dessen gibt es das Signal DATA_IO und mit dem Signal ADDR_XTR ein zusätzliches Steuersignal. Die Begründung dafür findet sich in der Anzahl der verfügbaren IO Leitungen der verwendeten Hardware. Eine genaue Beschreibung des Problems befindet sich in "Verbinden des FPGAs und AVRs" (Kapitel 6.4.1.2). Die Übertragung der Adresse findet statt, während das Signal ADDR_XTR high ist (Pos 1). Danach werden die Daten übertragen.

Mit der Clock, nachdem die Übertragung abgeschlossen ist, wechselt die Taktleitung des SPIs den Zustand von hochohmig zu low (Pos 2). Bis zu diesem Zeitpunkt war das Device als Slave konfiguriert und durfte keine Daten auf die Clockleitung schreiben. Ein Pull-up, wie in Kapitel 6.1.1 beschrieben, ist auf Grund der Startphase notwendig.

6.3.2 Sendebeginn

Wenn das gesamte Device konfiguriert ist, kann man, durch Schreiben einer "1" in das LSB des zweiten Konfigurationsregisters, den Sendevorgang starten (vgl. Kapitel 6.2.1.2). In der folgenden Abbildung 6.18 wird bei "Pos 1" genau diese "1" ins Register geschrieben. Anschließend beginnt das SPI bei "Pos 2" mit der Datenübertragung. Das Schreiben ins Register ist nur sehr verkürzt darstellbar, da das SPI zum Senden sehr viel mehr Takte benötigt.

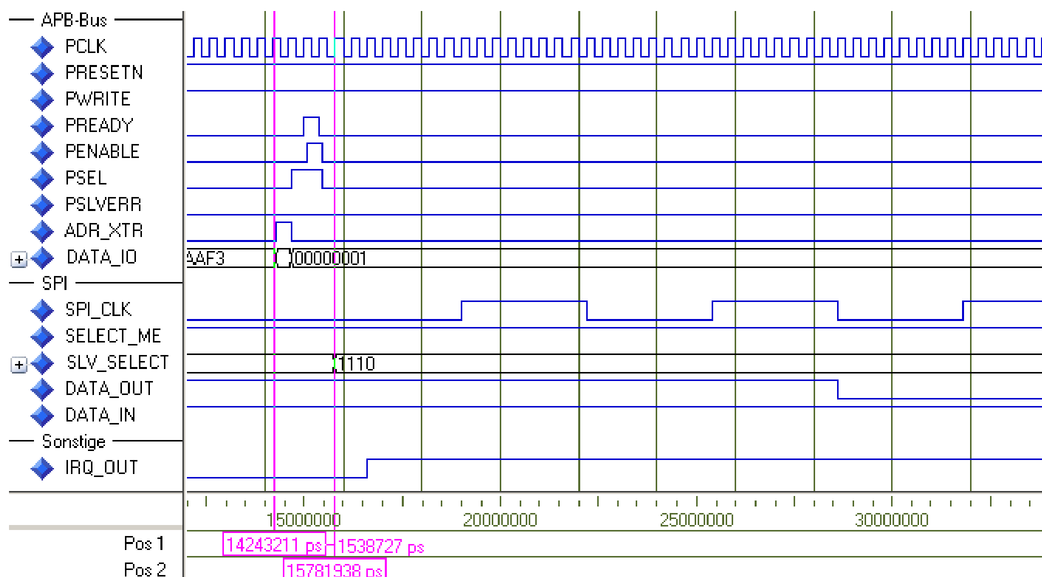


Abbildung 6.18: Timinigsimulation: Sendestart des SPIs

Bei ”Pos 2” wird der zuvor eingestellte Slave mittels der ”SLV_SELECT” ausgewählt. Das erste zu übertragende Byte ist 0xF3 (siehe Testbench im Anhang). Das LSB wird zuerst übertragen. Die ersten beiden Bits sind also high. Erst beim dritten Bit ändert sich der Zustand des DATA_OUT Signales des SPIs auf low.

6.3.3 Liste der durchgeführten Tests

Wie zu beginn des Kapitels bereits erwähnt, wurden nur zwei Tests gezeigt. Eine Liste mit allen Simulationen folgt hier.

- Schreibversuche auf alle Register - unter Beachtung, dass einige Register nicht beschreibbar sind (erneuter Test der Register und des APB-Interfaces)
- Testen des Sendens und Empfangens mit allen Kombinationsmöglichkeiten aus Master / Slave, CPHA 0 / 1, CPOL 0 / 1, MSB_FIRST 0 / 1
- Versenden / Empfangen von unterschiedlich langen Datenblöcken mit diversen Kombinationen aus Master / Slave, CPHA 0 / 1, CPOL 0 / 1, MSB_FIRST 0 / 1
- Einstellen von unterschiedlichen SPI-Taktgeschwindigkeiten im Mastermodus in diversen Kombinationen aus CPHA 0 / 1, CPOL 0 / 1, MSB_FIRST 0 / 1
- Vorgabe von unterschiedlichen SPI-Taktgeschwindigkeiten im Slavemodus in diversen Kombinationen aus CPHA 0 / 1, CPOL 0 / 1, MSB_FIRST 0 / 1
- Überprüfen der Statusregister nach jedem Sendevorgang
- Beobachten der Statusregister und des Verhaltens, bei leerem Sende- und vollem Empfangspuffer
- Alle aufgeführten Tests als einfache und Timingsimulation

6.4 Tests in Hardware

Nachdem alle Simulationen abgeschlossen sind, wird die gesamte Schaltung in Hardware aufgebaut.

6.4.1 Aufbau der Testhardware

Die Hardwaretests in dieser Bachelorarbeit erfolgen mit einem Spartan 3E-1200 FPGA der Firma Xilinx[1]. Der FPGA ist auf einem NEXYS 2 Board der Firma Digilent[2] (Dokumentation: Digilent Inc. [2008] / Digilent Inc. [2007]) verarbeitet. Das Board ist auf Abbildung 6.19 zu sehen.

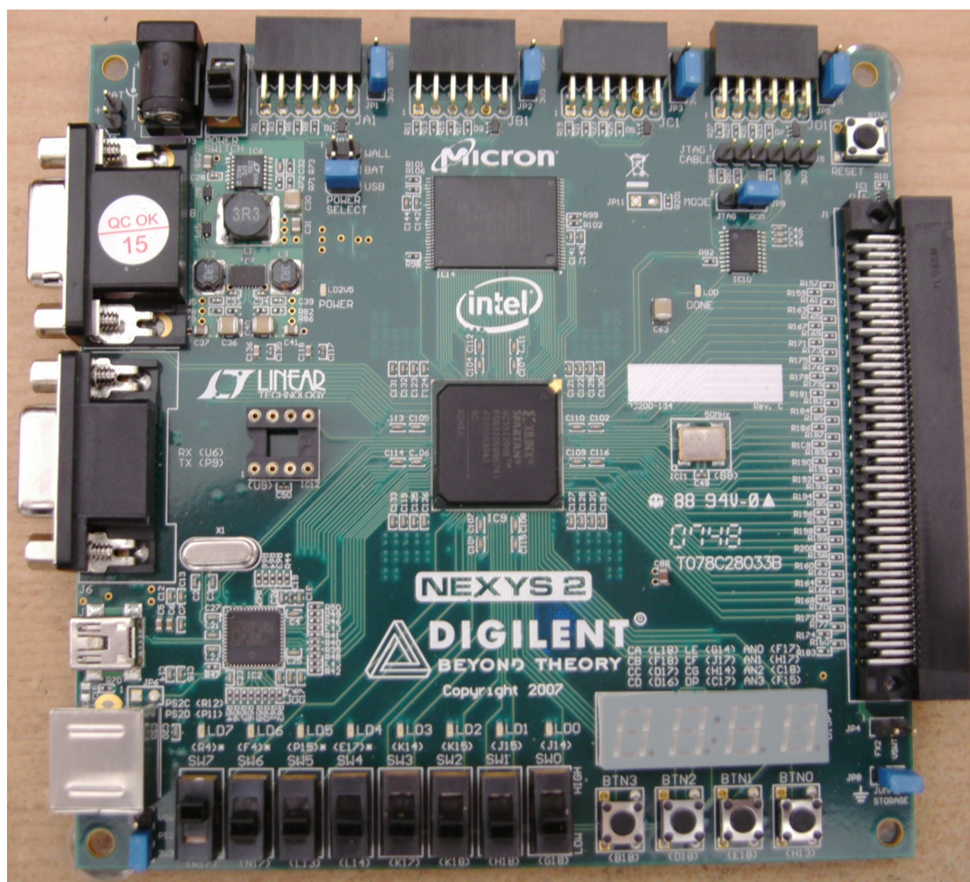


Abbildung 6.19: Foto des NEXYS 2 Boards

[1] www.xilinx.com
[2] www.digilentinc.com

Auf dem FPGA befindet sich nur das SPI-Interface zusammen mit dem μPI und dem APB-Interface. Es fehlt also ein Prozessor für die Ansteuerung des APB-Interfaces. Als Prozessor hierfür wird ein AT90CAN128 der Firma ATMEL[3] (Datenblatt: Atmel Corporation [2008]) gewählt. Dieser 8-Bit Prozessor wird sowohl den APB-Bus simulieren als auch ein zweites SPI als Gegenstelle für das SPI im FPGA bereitstellen. Da der APB-Bus synchron ist, wird der AVR[4] auch die Clock für den FPGA vorgeben. Der AVR befindet sich auf einem Board, welches durch den Laborassistenten Bruno Carstensen der HAW Hamburg entwickelt wurde. Ein solches Board ist auf dem folgenden Bild (Abb. 6.20) zu sehen. Der Schaltplan (Carstensen [2006b]) und der Bestückungsplan (Carstensen [2006a]) des Boards befindet sich auf der beigefügten CD.

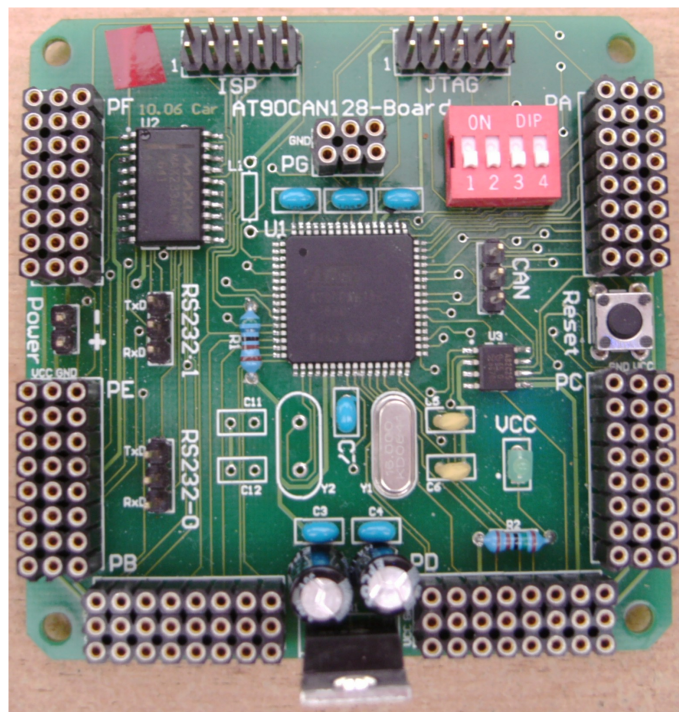


Abbildung 6.20: Foto eines AT90CAN128-Boards

Das Board mit dem AT90CAN128 wird bei 5 Volt betrieben. Das NEXYS 2 Board wird mit 3,3 Volt betrieben. Um eine Kommunikation zwischen beiden Boards zu ermöglichen, müssen entweder die Pegel mittels zusätzlicher Hardware angepasst, oder beide Boards mit der gleichen Spannung betrieben werden.

[3] www.atmel.com
[4] AVR bezeichnet eine 8-Bit Prozessorfamilie, zu der auch der AT90CAN128 gehört.

Eine Analyse des Problems zeigt, dass eine Anpassung der Pegel mittels zusätzlicher Hardware bei vielen benötigten Signalen aufwendig ist. Der FPGA sowie die meisten anderen Bausteine auf dem NEXYS 2 Board können nicht mit 5 Volt betrieben werden. Der AVR kann mit einer Spannung von 2,7 bis 5,5 Volt arbeiten, wenn die Arbeitsgeschwindigkeit des AVRs angepasst wird. Einige Hardwarebausteine auf dem AT90CAN128 Board können nur mit 5 Volt arbeiten.

6.4.1.1 Anpassung des AT90CAN128 Boards

Die Analyse hat gezeigt, dass auch wenn einige Bauteile auf dem AVR-Board ausgewechselt werden müssen, es der einfachste Weg ist, wenn der AVR mit 3,3 Volt arbeitet. Es müssen alle Bauteile, welche nicht bei 3,3 Volt arbeiten können, entfernt oder ausgetauscht werden. Der CAN-Bustreiber PCA82C250 arbeitet nicht bei 3,3 Volt [vgl. Philips Semiconductors, 2000, S. 2]. Da für diese Arbeit kein CAN-Bus benötigt wird, wird auch kein Ersatz benötigt. Ebenfalls muss der Treiber für die serielle Schnittstelle MAX233 entfernt werden [vgl. Maxim Integrated Products, Inc., 2006, S. 1]. Da die Bedienung des AVRs über eine serielle Schnittstelle erfolgt, muss ein anderer 3,3 Volt kompatibler Treiber verwendet werden. Zu dem MAX233 von Maxim[5] gibt es keinen Pin-Kompatiblen Treiber, der mit 3,3 Volt arbeitet. Es wird deshalb ein einfach zu erhaltender MAX3232 (Maxim Integrated Products, Inc. [2007]) verwendet, welcher sich auf einer kleinen zusätzlichen Platine[6] befindet. Der Spannungsregler 7805 auf dem Board ist für 5 Volt ausgelegt [vgl. Fairchild Semiconductor Corporation, 2001, S. 2]. Dieser wird entfernt. Das Board wird während der Arbeit mit einem Labornetzteil versorgt, welches auf 3,3 Volt eingestellt wird. Der letzte Umbau gilt der Arbeitsgeschwindigkeit des AT90CAN128, dieser kann bei 5 Volt mit bis zu 16 MHz arbeiten. Auf dem Board ist auch ein entsprechendes 16 MHz Quarz verbaut. Bei 3,3 Volt ist laut Datenblatt der höchste mögliche Takt 10,67 MHz [vgl. Atmel Corporation, 2008, S. 368]. Es muss also ein langsameres Quarz verbaut werden. Eine übliche Taktrate bei AVR Prozessoren ist 8 MHz. Das 16 MHz Quarz wird somit durch ein 8 MHz Quarz ersetzt.

[5] www.maxim-ic.com
[6] Schaltplan und Layout im Anhang (Kapitel C)

Zusammengefasst sind folgende Umbauten nötig, um das AT90CAN128 Board mit 3,3 V zu betreiben:

- Entfernen des CAN-Treibers (PCA82C250)
- Ersetzen des MAX233 durch einen anderen seriellen Treiber (MAX3232)
- Entfernen des Spannungsregler
- Tauschen des 16 MHz Quarzes durch ein 8 MHz Quarz

Das umgebaute AT90CAN128-Board sieht dann wie in Abbildung 6.21 aus.

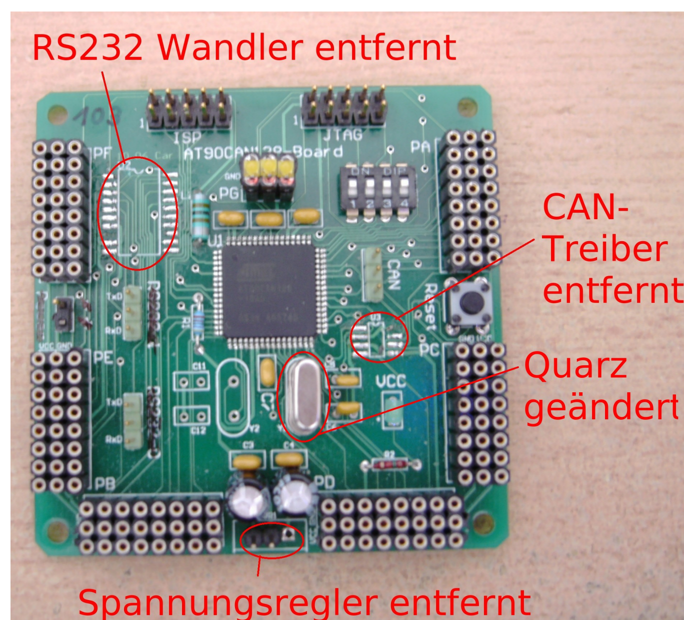


Abbildung 6.21: Foto des geänderten AT90CAN128-Boards

6.4.1.2 Verbinden des FPGAs und AVRs

Nachdem die gesamte Testumgebung mit 3,3 Volt arbeitet, können die beiden Boards miteinander verbunden werden.

Zum Verbinden der Hardware benötigt man über hundert Signale. Darunter befinden sich folgende Signale:

- 32 Datenleitungen des APB-Interfaces vom AVR zum FPGA
- 32 Datenleitungen des APB-Interfaces vom FPGA zum AVR
- 32 Adressleitungen des APB-Interfaces vom AVR zum FPGA

- 7 Steuerleitungen des APB-Interfaces (PCLK, PENABLE, PREADY, PSEL, PSLVERR, PWRITE, PRESET)
- 4 Signale vom SPI Interface (MOSI, MISO, SCK, SS)
- 1 Interruptleitung vom Impulse Statusregister

PRESET wird nicht vom AVR kontrolliert. Dieses Signal wird mit einem Schalter auf dem NEXYS 2 Board verbunden. Ebenso wird der Interruptausgang von dem Impulse Statusregister mit einer Led auf dem NEXYS 2 Board verbunden.

Über das NEXYS 2 Board kann man maximal 72 IO-Signale des FPGAs benutzen. Der AVR besitzt 53 programmierbare IO Signale. Selbst wenn man die 64 Datenleitungen zusammenfasst, da bei einem Zugriff über den APB nie gleichzeitig gelesen und geschrieben wird, reichen die verfügbaren IO-Pins am FPGA bzw. AVR nicht aus.

Um noch mehr Signale einzusparen, werden sowohl die Daten- als auch die Adressleitungen zusammengefasst. Dafür wird bei einem Datentransfer zuerst die Adresse übertragen und anschließend erfolgt die Übertragung der Daten. Um das APB-Interface im FPGA zu erhalten, wird vor das Interface eine zusätzliche Schicht gesetzt, welche die Adresse puffert. Dieser Puffer übernimmt ebenfalls die Kontrolle über die Datenrichtung der gemeinsamen Adress- / Datenleitungen während einer Datenübertragung und sorgt dafür, das die Daten entweder vom AVR zum FPGA oder vom FGPA zum AVR gelangen. Die Datenrichtung wird aus dem PWRITE Signal des APB-Interfaces erkannt. Um zu erkennen, ob eine Adresse übertragen wird, ist ein weiteres Signal notwendig, welches ADDR_XTRA genannt wurde. Die zusätzliche Schicht ist eine zusätzliche Entity in VHDL, welche oberhalb aller anderen Entities sitzt.

Es sind jetzt noch 43 Signale nötig, um den AVR mit dem FPGA zu verbinden. Folgende Signale werden jetzt noch benötigt:

- 32 kombinierte Adress- und Datenleitungen
- 6 Steuerleitungen des APB-Interfaces (PCLK, PENABLE, PREADY, PSEL, PSLVERR, PWRITE)
- 4 Signale vom SPI Interface (MOSI, MISO, SCK, SS)
- 1 zusätzliches Signal, um anzugeben, wann eine Adresse übermittelt wird (ADDR_XTRA)

Die Signale zwischen AVR und FPGA sind wie in der unten stehenden schematischen Darstellung (Abb. 6.22) verbunden. Je nachdem, ob der FPGA als SPI-Master oder SPI-Slave arbeitet, wird das ”SLAVE SELECT” oder ”SELECT ME” Signal nach außen verbunden.

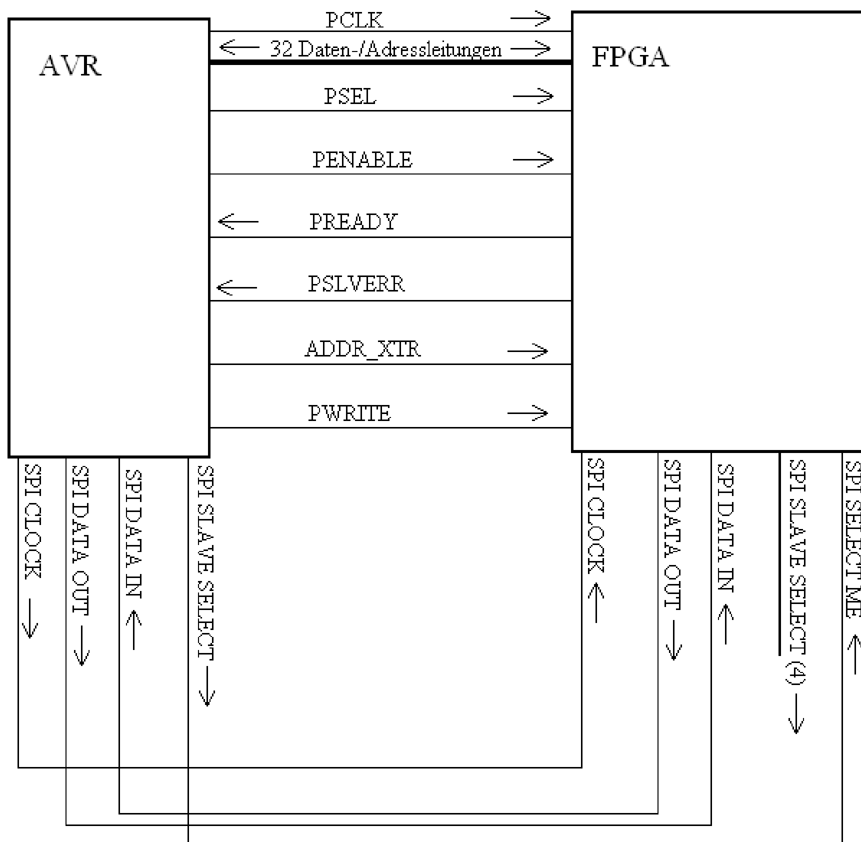


Abbildung 6.22: Schematische Darstellung des Aufbaues

6.4.1.3 Der fertige Testaufbau

Um eine etwas bessere Vorstellung von dem Aufbau zu vermitteln, wird der Aufbau der folgenden Tests hier etwas ausführlicher beschrieben und durch Fotos unterstützt.

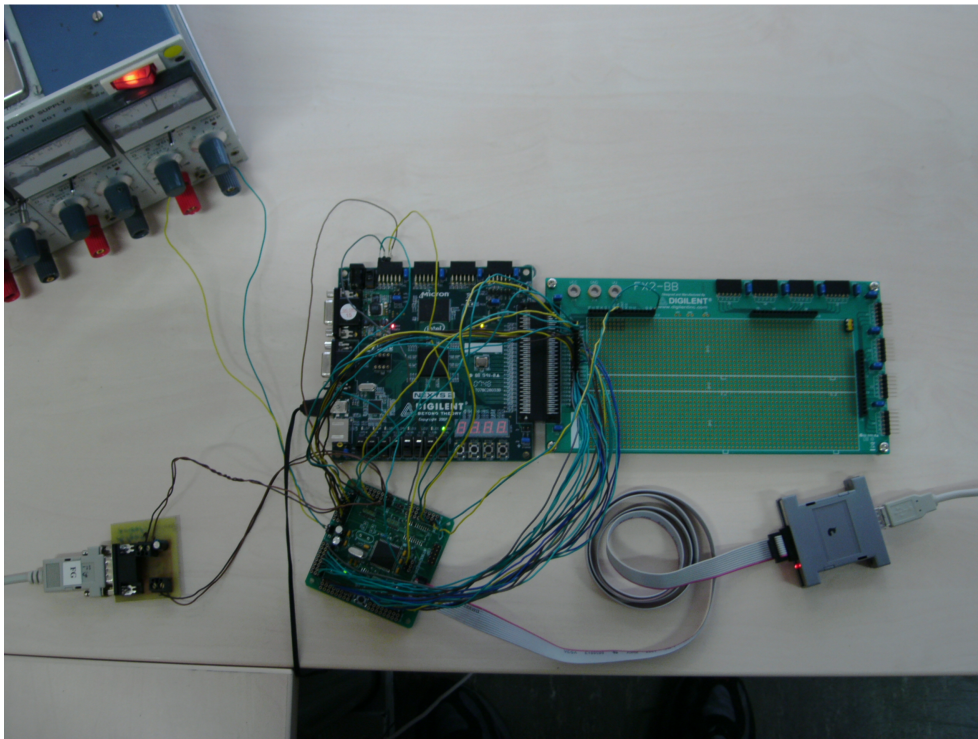


Abbildung 6.23: Gesamtansicht des Testaufbaues

Auf dem ersten Foto (Abb. 6.23) sieht man den gesamten Testaufbau. Oben links befindet sich das Labornetzteil, welches den AVR mit Strom versorgt. Unten links befindet sich die kleine Zusatzplatine, welche die Spannungen für die serielle Schnittstelle des AVRs auf die Spannung der seriellen Schnittstelle des PCs anpasst. Das Kabel, welches von der Platine nach links abgeht, ist mit dem PC verbunden. Der kleine graue Kasten unten rechts ist ein JTAG-Device, mit dem der AVR programmiert und in Echtzeit debuggt werden kann. Das Kabel auf der rechten Seite des Kastens führt zum PC. An dem anderem Ende des Flachbandkabels, welches im JTAG-Device steckt, ist die Platine angeschlossen, welche mit dem AT90CAN128 bestückt ist. Die Platine oberhalb des AT90CAN128-Boards ist das NEXYS 2 Board. Rechts in das NEXYS 2 Board ist eine Zusatzplatine gesteckt, welche nur dazu dient, die benötigten Pins für die Kommunikation mit dem AVR bereit zu stellen. Das schwarze Kabel auf der linken Seite des NEXYS 2 Boards geht zum PC. Es versorgt das NEXYS 2 Board mit Spannung. Ebenso wird der FPGA sowie ein sich auf dem Board befindlicher EEPROM[7] mit diesem Kabel programmiert.

[7] Electrically Erasable Programmable Read-Only Memory

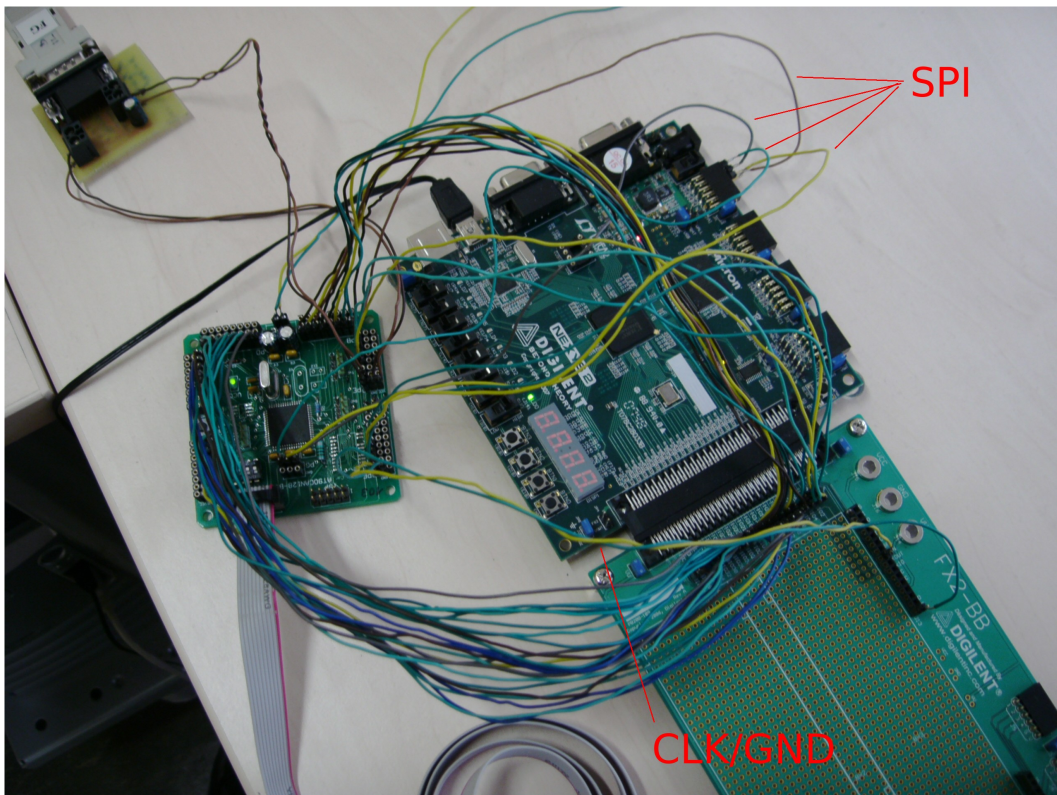


Abbildung 6.24: Gesamtansicht des Testaufbaues

Das zweite Foto (Abb. 6.24) zeigt die beiden wichtigen Platinen etwas näher. Die vier markierten Kabel im oberen Bereich stellen die Verbindung des SPIs zwischen den beiden Boards sicher. Die beiden markierten Kabel weiter unten sind die Clock, womit der AVR den FPGA versorgt und die GND-Leitung. Die beiden Kabel werden im Laufe der Tests miteinander verdreht, um eine bessere Fehlersicherheit zu erreichen (siehe dazu Kapitel 6.4.6). Die restlichen Kabel, welche vom AT90CAN128 Board zum NEXYS 2 Board laufen, gehören entweder zu den kombinierten Adress- und Datenleitungen oder zu den Steuersignalen für den Datenaustausch.

6.4.2 Die Software des AVRs

Damit der AVR ein APB-Master und ein SPI-Interface simulieren kann, wurde hierfür eine entsprechende Software entwickelt. Die Entwicklung der Simulationssoftware stellt nur ein Randgebiet dieser Arbeit dar und wird hier nicht weiter beschrieben.

Vor dem Einsatz der Software wurde diese mit dem AVR Studio, der von Atmel kostenlos erhältlichen Entwicklungs- und Simulationsumgebung, ausführlich getestet. Wo sich die gut kommentierte Software auf der CD befindet, steht im Anhang (siehe Anhang D.2.1).

6.4.3 Testen der Verbindung zwischen PC und AVR

Der Test besteht darin, die Verbindung des AVRs mit dem PC zu testen. Dafür wird auf dem PC ein Terminalprogramm gestartet und die Schnittstelle mit der richtigen Konfiguration zum AVR ausgewählt. In der Softwareversion vom Anhang ist die serielle Schnittstelle des AVRs auf 9600 Baud 8N1 eingestellt. Ein Handshake wird nicht unterstützt.

Wird der AVR resetet, gibt er einen Starthinweis und ein Menü aus. Der Starthinweis und das Menü sind in Abbildung 6.25 zu sehen. Wenn diese Ausgabe im Terminalprogramm sichtbar ist und der AVR Eingaben vom PC korrekt empfängt, ist die Verbindung hergestellt. Der AVR sendet alle gemachten Eingaben als Echo zurück.

```
===================================================
-Start-

Hilfe:

S: Sende Daten
L: Lese Daten
C: Setze Geschwindigkeit der Clock

1: Init Spi als Slave
2: Init Spi als Master
3: Sende per SPI Daten
4: Lese per SPI empfangene Daten
-: Dauertest mit AVR als SPI-Master
+: APB Dauertest mit AVR als APB-Master
===================================================
```

Abbildung 6.25: AVR-Menu

6.4.4 Testen des APB-Interfaces und der Register

Bevor das SPI getestet wird, ist sicher zu stellen, dass das APB-Interface korrekt arbeitet und eventuelle Fehler, die man im SPI sucht, nicht im APB-Interface vorhanden sind.

Als erstes wird versucht, Werte in alle Register zu schreiben und Werte aus den Registern zu lesen. Dabei müssen alle Statusregister und das Empfangpufferregister den Schreibvorgang mit einem Fehler verweigern. Der Schreib- und Lesezugriff auf das Kontrollregister an Adresse 0x08 ist in Abbildung 6.26 dargestellt. Aus Platzgründen und zu Gunsten der Übersicht, wird das Menü in der Abbildung weggelassen. Eingaben, die vom PC kommen, sind blau eingefärbt.

```
===================================================
-Start-
Hilfe:  ...Menü nicht abgebildet...
s
Bitte Adresse in 2 Bloecken in Hex eingeben (Bsp: 0x1234 0x5678) :
0x0000 0x0008
Bitte Daten in 2 Bloecken in Hex eingeben (Bsp: 0x1234 0x5678) :
0x1234 0x5678
Daten erfolgreich gesendet
Hilfe:  ...Menü nicht abgebildet...
l
Bitte Adresse in 2 Bloecken in Hex eingeben (Bsp: 0x1234 0x5678) :
0x0000 0x0008
Daten erfolgreich empfangen
Empfangene Daten sind: 0x1234 0x5678
===================================================
```

Abbildung 6.26: Schreiben und Lesen des Kontrollregisters 0x08

Ein Schreibzugriff auf das einfache Statusregister an Adresse 0x10 ist folgend abgebildet (Abb. 6.27). Da das Statusregister nicht vom Bus aus beschrieben werden kann, ist der Vorgang nicht erfolgreich. Der AVR gibt einen entsprechenden Fehler im Terminal aus. Wird das Statusregister hinterher gelesen, hat es den Wert, der durch den internen Status des SPIs gerade vorgegeben wird. In diesem Fall ist der Sendepuffer leer, wodurch das LSB gesetzt ist. Vergleiche dazu Kapitel 6.2.1.5.

```
===================================================
Hilfe:  ...Menü nicht abgebildet...
s
Bitte Adresse in 2 Bloecken in Hex eingeben (Bsp: 0x1234 0x5678) :
0x0000 0x0010
Bitte Daten in 2 Bloecken in Hex eingeben (Bsp: 0x1234 0x5678) :
0xABCD 0xEF01
Device hat Fehler gemeldet
Hilfe:  ...Menü nicht abgebildet...
l
Bitte Adresse in 2 Bloecken in Hex eingeben (Bsp: 0x1234 0x5678) :
0x0000 0x0010
Daten erfolgreich empfangen
Empfangene Daten sind: 0x0 0x1
===================================================
```

Abbildung 6.27: Schreiben und Lesen des Statusregisters 0x10

Für das APB-Interface wurden alle Tests in Hardware, welche auch in Kapitel 5.4 angegeben sind, durchgeführt. In die Kontrollregister wurden verschiedene Werte geschrieben und wieder gelesen. In das Sendepufferregister wurden verschiedene Werte geschrieben und dabei die Rückmeldungen durch das Statusregister an Adresse 0x10 beobachtet. Außerdem wurde darauf geachtet, dass man nicht mehr Daten in den Sendepuffer schreiben kann, als dieser Platz bietet.

6.4.5 Testen des SPI-Interfaces

Nachdem die Funktionalität des APB-Interfaces sichergestellt ist, geht es nun an das Testen des SPIs. Zunächst einmal werden der Eingang und der Ausgang des SPI-Interfaces des FPGAs miteinander verbunden. Somit empfängt der FPGA alle Daten, welche er auch sendet. Die Leitungen am FPGA werden wie in der folgenden schematischen Darstellung (Abb. 6.28) verbunden. Ob die Selectleitungen ihre Funktion erfüllen, wird in diesem Aufbau mit einem Oszilloskop überprüft.

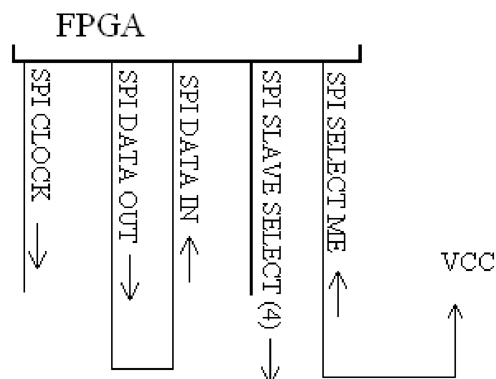


Abbildung 6.28: FPGA als SPI-Master

Es folgt eine Abbildung (Abb. 6.29) der Kommunikation zwischen AVR und PC, um den FPGA als SPI Master einzustellen, die Daten zur Übertragung vorzubereiten, zu versenden und zum Auslesen der empfangenen Daten.

```
===================================================
-Start-
Hilfe:  ...Menü nicht abgebildet...
s
Bitte Adresse in 2 Bloecken in Hex eingeben (Bsp: 0x1234 0x5678) :
0x0000 0x0000
Bitte Daten in 2 Bloecken in Hex eingeben (Bsp: 0x1234 0x5678) :
0x8000 0x0001
Daten erfolgreich gesendet
Hilfe:  ...Menü nicht abgebildet...
s
Bitte Adresse in 2 Bloecken in Hex eingeben (Bsp: 0x1234 0x5678) :
0x0000 0x0008
Bitte Daten in 2 Bloecken in Hex eingeben (Bsp: 0x1234 0x5678) :
0x8000 0x000F
Daten erfolgreich gesendet
Hilfe:  ...Menü nicht abgebildet...
s
Bitte Adresse in 2 Bloecken in Hex eingeben (Bsp: 0x1234 0x5678) :
0x0000 0x000C
Bitte Daten in 2 Bloecken in Hex eingeben (Bsp: 0x1234 0x5678) :
```

```
0x0000 0x0010
Daten erfolgreich gesendet
Hilfe:  ...Menü nicht abgebildet...
s
Bitte Adresse in 2 Bloecken in Hex eingeben (Bsp: 0x1234 0x5678) :
0x0000 0x001C
Bitte Daten in 2 Bloecken in Hex eingeben (Bsp: 0x1234 0x5678) :
0x0000 0xF271
Daten erfolgreich gesendet
Hilfe:  ...Menü nicht abgebildet...
s
Bitte Adresse in 2 Bloecken in Hex eingeben (Bsp: 0x1234 0x5678) :
0x0000 0x0004
Bitte Daten in 2 Bloecken in Hex eingeben (Bsp: 0x1234 0x5678) :
0x0000 0x0001
Daten erfolgreich gesendet
Hilfe:  ...Menü nicht abgebildet...
l
Bitte Adresse in 2 Bloecken in Hex eingeben (Bsp: 0x1234 0x5678) :
0x0000 0x0014
Daten erfolgreich empfangen
Empfangene Daten sind: 0x0 0x1C
Hilfe:  ...Menü nicht abgebildet...
l
Bitte Adresse in 2 Bloecken in Hex eingeben (Bsp: 0x1234 0x5678) :
0x0000 0x0018
Daten erfolgreich empfangen
Empfangene Daten sind: 0x0000 0xF271
===================================================
```

Abbildung 6.29: FPGA sendet als Master Daten per SPI

Im Einzelnen geschehen die in Tabelle 6.7 gezeigten Schritte. Die Werte können auch in Kapitel 6.2.1 nachgelesen werden.

Adresse	Wert	Vorgang	Beschreibung
0x0000 0x0000	0x8000 0x0001	Schreiben	Device ist Master, CPOL=0, CPHA=0, LSB first, SS-Leitung 1
0x0000 0x0008	0x0000 0x000F	Schreiben	Taktteiler auf 16 einstellen
0x0000 0x000C	0x0000 0x0010	Schreiben	Anzahl Bits zum versenden = 16
0x0000 0x001C	0x0000 0xF271	Schreiben	Übergebe zu sendende Daten an Sendepuffer
0x0000 0x0004	0x0000 0x0001	Schreiben	Starte SPI-Datenübertragung
0x0000 0x0014	0x0000 0x001C	Lesen	Status der letzten Übertragung - Bytes empfangen, Datenübertragung begonnen / beendet
0x0000 0x0018	0x0000 0xF271	Lesen	Empfangene Daten

Tabelle 6.7: FPGA sendet als Master Daten per SPI

Da ein Aufzeigen aller durchgeführten Tests hier zu lang wäre, wird folgend eine Liste aller durchgeführten Tests gegeben.

- Überprüfen der Slave Select Signales mittels Oszilloskop
- Versenden von unterschiedlich langen Nachrichten von ein bis über 128 Bits (Überlauf im Empfangspuffer)
- Verstellen des Taktteilers bis unter die Grenze von vier.
- Anpassen der Einstellungen CPOL, CPHA, MSB First in verschiedenen Kombinationen.

Nachdem alle Tests mit dem FPGA als SPI-Master abgeschlossen sind, werden folgende Tests mit dem FPGA als SPI-Slave durchgeführt. die Verkabelung erfolgt dabei wie in Abbildung 6.22, welche bereits weiter vorne zusehen ist.

- Versenden von unterschiedlich langen Nachrichten bis über 96 Bits (Überlauf im Empfangspuffer)
- Testen der Einstellungen CPOL, CPHA, MSB First in verschiedenen Kombinationen.
- Alle Tests bei unterschiedlichen Geschwindigkeiten des SPI-Taktes

6.4.6 8 Stunden Test

Nachdem die Funktionalität des SPIs getestet wurde, wird nun die Fehlersicherheit des gesamten Systems über eine Zeitspanne von acht Stunden getestet.

Die Software für den AVR bietet dafür zwei Tests an. Im ersten Test wird ein Register mit pseudozufälligen Werten beschrieben und immer wieder gelesen. Anschließend wird der geschriebene mit dem gelesenem Wert verglichen. Bei einem Fehler wird eine entsprechende Meldung ausgegeben. Beim zweiten Test werden Daten über das SPI gesendet, wobei der AVR die Rolle des SPI-Masters übernimmt. Alle gesendeten Daten werden auf Richtigkeit überprüft.

Die Pseudozufallszahlen wurden mit einer PRBS (engl. pseudorandom binary sequence) generiert.[8]

Da der zweite Test auch die Funktionalität des APB-Busses sicher stellt wird hier nur dieser aufwendigere Test dargestellt.

Vor Beginn des eigentlichen Tests muss das SPI im FPGA als SPI-Slave konfiguriert werden. Anschließend muss der AVR so eingestellt werden, dass sein SPI als Master arbeitet.

Bei dem Test selbst werden im einzelnen folgende Schritte durchgeführt:

- Prüfen, ob der Sendepuffer leer ist. Wenn nicht, Sendepuffer leeren und Fehlermeldung ausgeben
- Prüfen, ob der Empfangspuffer leer ist. Wenn nicht, Empfangspuffer leeren und Fehlermeldung ausgeben
- Mittels PRBS Funktion die nächsten 16 Bit zum versenden generieren
- Die generierten 16 Bit mittels APB-Bus in den Sendepuffer des SPIs im FPGA übertragen
- Die generierten 16 Bit dem SPI im AVR zum Senden übergeben
- Warten bis die Daten über SPI vom AVR an den FPGA und vom FPGA an den AVR gesendet wurden
- Überprüfen, ob die vom AVR empfangenen Daten mit den gesendeten identisch sind. Bei einem Fehler eine entsprechende Meldung ausgeben
- Überprüfen, ob die Daten im Empfangspuffer im FPGA den gesendeten Daten entsprechen
- Überprüfen, ob beim Lesen des Registers keine Fehler aufgetreten sind

[8]Die PRBS-Funktion lautete: $f(x) = x^{15} + x^5 + x^3 + x^2 + x^1$.

- Überprüfen, ob das Impulsestatusregister den erwarteten Wert enthält
- Alle tausend Durchläufe einen Hinweis auf aktuellen Durchlauf geben
- Bei Punkt eins des Tests anfangen

Der Test läuft so lange, bis er durch äußere Eingriffe gestoppt wird.

Der Test läuft bei der ersten Durchführung nicht stabil. Bei etwa zwei von tausend Tests kommt es zu einem Fehler. Zwei zufällig ausgewählte Fehler werden hier gezeigt.

```
===================================================
Durchlauf Nummer: 1000
Gesendet und empfangen stimmen nicht ueberein (AVR-Recv). Empfangen:
0x2548 - Erwartet: 0x3548 - I: 1261
===================================================
```

Abbildung 6.30: 8h Test, Fehler 1

Die obige Fehlerausgabe (Abb. 6.30) zeigt an, dass der AVR andere Daten über das SPI empfangen hat, als dem FPGA zum Senden übergeben wurden. Der Fehler kann sowohl bei der Übertragung der Daten zum FPGA als auch beim Senden über das SPI aufgetreten sein. "I" gibt an, bei dem wievielten Durchlauf der Fehler auftrat.

Folgend sind zwei Fehler abgebildet (Abb. 6.31). Der erste Fehler zeigt an, dass das Impulse-Statusregister andere Werte als erwartet enthielt. Zu erwarten ist der Wert 0x1C[9] (Übertragung begonnen, Übertragung beendet, Daten empfangen). Der Wert 0x3C gibt an, dass der Empfangspuffer fast voll ist. Dieses Flag ist so eingestellt, dass es gesetzt wird, sobald sich zwei oder mehr Datenpakete im Puffer befinden. Die darauffolgende Fehlerausgabe bestätigt, dass sich zwei Datensätze im Empfangspuffer befanden. Die Nummer des Durchlaufes ist deshalb um eins erhöht, da der Empfangspuffer immer vor einem Test überprüft wird. Es ist davon auszugehen, dass das SPI im FPGA das Slave-Select-Signal fälschlicher Weise zu oft als low gelesen hat.

[9] Die Flags des Statusregisters können in Kapitel 6.2.1.6 noch einmal nachgelesen werden.

```
===================================================
Durchlauf Nummer: 4000
STATUS-Register enthielt andere Werte als erwartet: 0x3C - I: 4927
Empfangspuffer nicht leer. I: 4928
Durchlauf Nummer: 5000
===================================================
```

Abbildung 6.31: 8h Test, Fehler 2

Auch wenn die Pseudozufallssequenz immer mit dem selben Werten initialisiert werden und somit nach einen neuen Start des Tests immer wieder die selben Pseudozufallszahlen erzeugt werden, treten die Fehler nicht bei den gleichen Durchläufen auf. Es gibt auch keinen Fehler, der häufiger als andere auftritt. Eine Zuordnung zu einem VHDL-Designfehler oder einem Softwarefehler in dem AVR ist nicht möglich. Die Fehler sind mit ca. 0,2% so gering, dass sie beim manuellen Testen nicht aufgefallen sind. Für eine produktive Nutzung ist die Menge aber zu hoch. Der Fehler muss also analysiert und beseitigt werden.

Zunächst einmal werden die Signale mittels Oszilloskop analysiert. Dazu wird ein Zweikanal-Oszilloskop an die Taktleitung vom AVR zum FPGA und an das Signal PWRITE geklemmt. Obwohl das Oszilloskop nur eine Impedanz von 20 pF am Eingang aufweist [vgl. Pico Technology Limited, 2007, S. 8], gibt es nach dem Anschließen der Testköpfe nahezu keine fehlerfreie Übertragung mehr zwischen FPGA und AVR. Dies ist ein erster Hinweis darauf, dass der Fehler in der Art der Übertragung zu suchen ist.

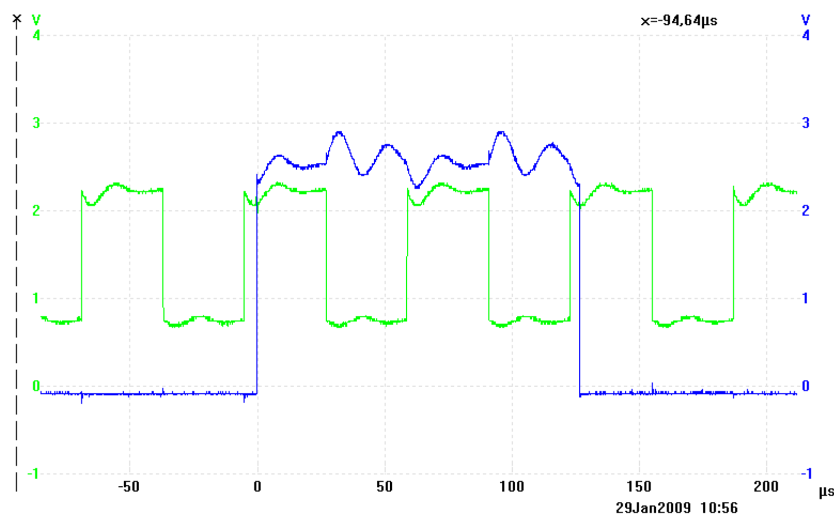


Abbildung 6.32: Hardwaretest - Fehleranalyse mit dem Oszilloskop

Die Abbildung 6.32 zeigt die Signale, welche mittels Oszilloskop aufgenommen wurden. Der Takt ist grün und PWRITE blau abgebildet. Es fällt auf, dass das Taktsignal nicht, wie das PWRITE-Signal auf 0 Volt zurückgeht. Des Weiteren schwingt der Takt lange nach, bevor das Signal stabil ist. Die Schwingungen übertragen sich in verstärkter Form auf alle anderen Signale, wie man am Beispiel des abgebildeten PWRITE Signales sieht. Wird die Verbindung des Taktsignales zwischen AVR und FPGA getrennt, ist das Taktsignal sehr sauber. Die Zeit in der das Signal schwingt, ist sehr gering und die Spannung geht bei einem low Signal bis auf 0 Volt herunter. Auch gehen die Schwingungen auf den anderen Signalen stark zurück. Dieses Verhalten ist in Abbildung 6.33 zu sehen, wo das Taktsignal vom FPGA getrennt ist. Das PWRITE-Signal geht nicht wieder auf low, da der AVR auf die Bestätigung durch den FPGA wartet, was dieser mangels Taktquelle nicht tun kann.

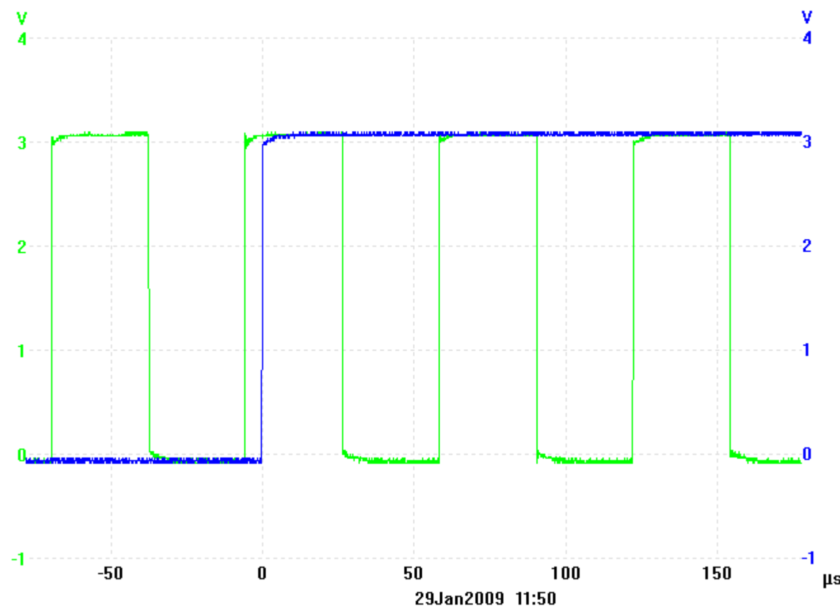


Abbildung 6.33: Hardwaretest - Fehleranalyse mit dem Oszilloskop, 2

Auf Grund des hohen Spannunganstieges, während das Signal low ist, ist anzunehmen, dass das Signal durch den FPGA sehr stark belastet wird. Im Datenblatt des AVRs sind zwei Grafiken angegeben, welche den Spannungsanstieg bei einem low Signal an einem belasteten Ausgangspin zeigen. Eine Abbildung ist für 5 Volt und eine für 2,7 Volt VCC angegeben. Beide Abbildungen zeigen das gleiche Verhalten, lediglich der Wert des Spannungsanstieg unterscheidet sich ein wenig im Vergleich zur Strombelastung. Die Abbildung 6.34 [Atmel Corporation,

2008, S.395] zeigt den Anstieg der Spannung an einem IO-Port des AVRs, bei einer VCC Spannung von 2,7 Volt. Diese ist näher an den verwendeten 3,3 Volt, als die andere vorhandene Grafik mit VCC gleich 5 Volt.

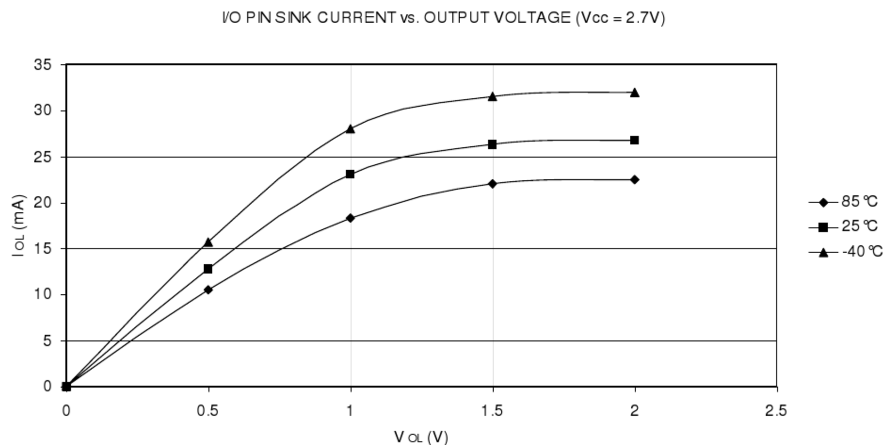


Abbildung 6.34: Hardwaretest - Fehleranalyse mit dem Oszilloskop

Durch die oben gezeigte Analyse liegt die Vermutung nahe, dass es auf Grund der Schwingungen auf den Signalen zu Fehlinterpretationen der Signale durch den AVR und den FPGA kommt. Der Assistent der HAW Dipl. Ing. Volker Reinecke bestätigte die Vermutung, dass zu steile Flanken bei der Datenübertragung Schwingungen erzeugen können, welche wiederum fehlerhaft interpretiert werden können, in einem Gespräch.

Nachdem die Anstiegsgeschwindigkeit (engl. slew rate) der IO-Ports im FPGA, mittels der Anweisung "SLEW=SLOW" in der UCF-Datei, heruntergesetzt ist und das Taktsignal, wie auf dem Foto 6.24 gezeigt, mit der Masseleitung verdreht ist, liegt der Anteil der Fehler bei 0,0 %. Während der gesamten acht Stunden des Tests gab es keinen einzigen Fehler. Es wurden insgesamt 340.000 Pseudozufallszahlen fehlerfrei versendet und empfangen. Die gesamte Ausgabe des AVRs, des erfolgreichen Tests befindet sich auf der CD.

Es darf kein Oszilloskop angeschlossen werden, da sonst wieder viele Fehler auftreten. Das Signal ist auf dem Oszilloskop nicht sichtbar besser. Der gesamte Aufbau wie er für diese Bachelorarbeit genutzt wird, befindet sich also weiterhin im Grenzbereich, was das fehlerfreie Laufen betrifft. Für einen erneuten Aufbau sollten keine Einzelleiter genutzt werden. Die Benutzung von Flachbandkabel, in dem jede zweite Leitung Masse ist, wird für einen sicheren Aufbau empfohlen.

7 Zusammenfassung und Ausblick

In Kapitel 2 hat sich gezeigt, dass keines der vorhandenen Mikroprozessorinterface die gestellten Anforderungen erfüllt. Somit wurden Analysen durchgeführt, wie eine Eigenentwicklung am besten umzusetzen ist (vgl. Kapitel 3). In Kapitel 4 wurden die gemachten Analysen umgesetzt und in Kapitel 5 diese Umsetzungen auf Fehler geprüft. Mit dem SPI wurde der erster Peripheriebaustein, welcher auf dem neu entwickeltem μPI basiert, implementiert und erfolgreich getestet (vgl. Kapitel 6). Es steht somit ein getestet μPI, welches alle geforderten Aufgaben erfüllt und darüber hinaus einige zusätzliche Register mit weiteren Fähigkeiten besitzt, für viele Entwicklungen bereit.

Das der Prozessor, für welchen dieses μPI entwickelt wurde, noch nicht vorhanden ist, machte die Wahl des Prozessorbusses nicht ganz leicht. Es wurde versucht einen Bus zu wählen, der in dem späteren Prozessor sehr wahrscheinlich vorhanden sein wird oder durch eine APB-Bridge zur Verfügung gestellt werden kann.

Während der Entwicklung wuchs der Umfang des μPI schnell. Zur Zeit existieren 15 verschiedene Register. Es hat sich gezeigt, dass es wichtig ist, alle Register so weit wie möglich nach dem selben Schema aufzubauen. Dies erleichtert die Wart- und Erweiterbarkeit des Quelltextes stark. Nachdem nun alle Register die gleiche Struktur haben, ist es für jeden, der dieses μPI um zusätzliche Register erweitern möchte, leicht möglich. Als Vorlage dafür stehen die bereits erwähnten 15 Register bereit.

Diese Arbeit hat die Grenzen von VHDL aufgezeigt. Durch das automatische Generieren von Registern, wobei unterschiedliche Register unterschiedliche Signale benötigen, bleiben Signale zum Teil ohne Verwendung. Es gibt in VHDL keine Möglichkeit, solche Signale als nicht verwendet zu markieren, um Warnungen durch das Synthesewerkzeug zu vermeiden. Dieses Warnungen können bei einem Entwickler bei erstmaliger Verwendung des μPIs zu der Annahme führen, dass das μPI nicht korrekt arbeitet.

Beim Schreiben der Arbeit hat sich gezeigt, dass es nötig sein kann, Abstriche bei kleinen Details zu machen. Würde man alle teils sehr interessanten, aber sehr kleinen Details einbringen, wird es für den Leser schnell unübersichtlich. Er würde den Gesamtzusammenhang nicht mehr verstehen. Auch würde die ohnehin recht umfangreiche Arbeit weiter an Umfang zulegen. In diesem Rahmen ist ein weiterer wichtiger Aspekt die Nutzung von Grafiken. Sie führen in Verbindung mit dem Text zu einer besseren Übersicht und besserem Verständnis beim Leser. Grafiken wurden im Verlauf der Arbeit immer stärker eingesetzt.

7.1 Ausblick

Das μPI und das SPI-Interface sind einsatzbereit. Dennoch gibt es Dinge, die noch offen sind. Der wichtigste Schritt ist die Wahl eines Prozessors mit dem das μPI gemeinsam arbeiten wird. Ebenso wichtig ist es, das TICEP_CONV-Package herstellerunabhängig aufzubauen, so dass auch das μPI und SPI herstellerunabhängig eingesetzt werden können.

Sind die beiden oben genannten Aufgaben erfüllt, kann die eigentliche Entwicklungsarbeit beginnen. Das μPI ist kein fertiges Produkt. Es nimmt anderen Entwickler von Peripherie viel Arbeits- und Zeitaufwand ab. Wichtig hierbei ist, dass das μPI flexibel aufgebaut ist. Die mitgelieferten 15 Register erfüllen viele Aufgaben, aber es wird sicherlich der Punkt kommen, wo ein abgewandeltes Register oder ein Register mit komplett neuen Fähigkeiten benötigt wird. Das μPI selbst wird sich, je mehr es benutzt wird, immer weiter entwickeln und immer weitere Fähigkeiten gewinnen.

Auch wenn mit dem SPI-Interface eine erste Entwicklung erfolgreich abgeschlossen ist, kann diese Bachelorarbeit nur ein Anfang gewesen sein.

Literaturverzeichnis

[Altera Corporation 2009] ALTERA CORPORATION ; ALTERA CORPORATION (Hrsg.): *Embedded Products Overview*. 2009. – URL http://www.altera.com/technology/embedded/overview/emb-overview.html. – Zugriffsdatum: 03.01.2009

[ARM Limited 2004a] ARM LIMITED ; ARM LIMITED (Hrsg.): *AMBA 3 APB Protocol*. 2004. – URL http://www.arm.com/securedownloads/user/index.php?subpage=download&action=download&fileid=5&&nomenu=1¬ext=1. – Zugriffsdatum: 08.08.2008. – Download nur nach kostenloser Anmeldung

[ARM Limited 2004b] ARM LIMITED ; ARM LIMITED (Hrsg.): *AMBA AXI Protocol*. 2004. – URL http://www.arm.com/securedownloads/user/index.php?subpage=download&action=download&fileid=3&&nomenu=1¬ext=1. – Zugriffsdatum: 08.08.2008. – Download nur nach kostenloser Anmeldung

[ARM Limited 2006] ARM LIMITED ; ARM LIMITED (Hrsg.): *AMBA 3 AHB-Lite Protocol*. 2006. – URL http://www.arm.com/securedownloads/user/index.php?subpage=download&action=download&fileid=26&&nomenu=1¬ext=1. – Zugriffsdatum: 08.08.2008. – Download nur nach kostenloser Anmeldung

[ARM Limited 2009] ARM LIMITED ; ARM LIMITED (Hrsg.): *ARM FPGA Solutions*. 2009. – URL http://www.arm.com/fpga/. – Zugriffsdatum: 03.01.2009

[Atmel Corporation 2008] ATMEL CORPORATION: *AT90CAN32/64/128*, 2008. – URL http://atmel.com/dyn/resources/prod_documents/doc7679.pdf. – Zugriffsdatum: 08.01.2009

[Carstensen 2006a] CARSTENSEN, Bruno: *AT90CAN128 Bestueckung*. 2006

[Carstensen 2006b] CARSTENSEN, Bruno: *AT90CAN128-Board*. 2006

[CAST Inc.] CAST INC. ; CAST INC. (Hrsg.): *PIP-AMBA-E SoC Kernel for ARM9 AMBA Bus Systems*. – URL http://www.cast-inc.com/systemip/pips/pip-amba-e/index.shtml. – Zugriffsdatum: 03.01.2009

[Chandrashekar und Mahmud 2005] CHANDRASHEKAR, Shekar ; MAHMUD, Rafey ; ALTERA CORPORATION (Hrsg.): *IC Designers – An Optimal Approach to Programmable Logic*. 2005. – URL http://www.altera.com/literature/cp/gspx/ic-designers.pdf. – Zugriffsdatum: 08.09.2008

[Chapman 2008] CHAPMAN, Ken ; XILINX, INC. (Hrsg.): *Get Smart About Reset: Think Local, Not Global*. 2008. – URL http://www.xilinx.com/support/documentation/white_papers/wp272.pdf. – Zugriffsdatum: 08.09.2008

[Digilent Inc. 2007] DIGILENT INC.: *Nexys II*, 2007. – URL http://www.digilentinc.com/Data/Products/NEXYS2/Nexys2_sch.pdf. – Zugriffsdatum: 17.10.2008

[Digilent Inc. 2008] DIGILENT INC.: *Digilent Nexys2 Board Reference Manual*, 2008. – URL http://www.digilentinc.com/Data/Products/NEXYS2/Nexys2_rm.pdf. – Zugriffsdatum: 17.10.2008

[Fairchild Semiconductor Corporation 2001] FAIRCHILD SEMICONDUCTOR CORPORATION: *KA78XX/KA78XXA 3-Terminal 1A Positive Voltage Regulator*, 2001. – URL http://www.datasheetcatalog.org/datasheets/228/390068_DS.pdf. – Zugriffsdatum: 08.01.2009

[Garrault und Philofsky 2005] GARRAULT, Philippe ; PHILOFSKY, Brian: HDL Coding Practices to Accelerate Design Performance. In: *Xcell Journal* (2005)

[Maxim Integrated Products, Inc. 2006] MAXIM INTEGRATED PRODUCTS, INC.: *+5V-Powered, Multichannel RS-232 Drivers/Receivers*, 2006. – URL http://datasheets.maxim-ic.com/en/ds/MAX220-MAX249.pdf. – Zugriffsdatum: 08.01.2009

[Maxim Integrated Products, Inc. 2007] MAXIM INTEGRATED PRODUCTS, INC.: *3.0V to 5.5V, Low-Power, up to 1Mbps, True RS-232 Transceivers Using Four 0.1μF External Capacitors*, 2007. – URL http://datasheets.maxim-ic.com/en/ds/MAX3222-MAX3241.pdf. – Zugriffsdatum: 08.01.2009

[Philips Semiconductors 2000] PHILIPS SEMICONDUCTORS: *PCA82C250 CAN controller interface*, 2000. – URL http://www.datasheetcatalog.org/datasheet/philips/PCA82C250.pdf. – Zugriffsdatum: 08.01.2009

[Pico Technology Limited 2007] PICO TECHNOLOGY LIMITED: *Series PicoScope 3000 PC-Oszilloskopen Handbuch*, 2007. – URL http://www.picotech.com/document/pdf/ps3000049.pdf. – Zugriffsdatum: 06.02.2009

[Schwerdtfeger 2000] SCHWERDTFEGER, Martin ; MCT PAUL UND SCHERER MIKROCOMPUTERTECHNIK GMBH (Hrsg.): *SPI made Simple - ein modulares SPI Konzept*. 2000. – URL http://www.mct.de/faq/spi.html. – Zugriffsdatum: 15.09.2008

[www.mikrocontroller.net 2008] WWW.MIKROCONTROLLER.NET ; WWW.MIKROCONTROLLER.NET (Hrsg.): *AVR-GCC-Tutorial*. 2008. – URL http://www.mikrocontroller.net/articles/AVR-GCC-Tutorial#UART_initialisieren. – Zugriffsdatum: 17.11.2008

[Xilinx Inc. 2009] Xilinx Inc. ; Xilinx Inc. (Hrsg.): *Embedded Processing*. 2009. – URL http://www.xilinx.com/products/design_resources/proc_central/index.htm. – Zugriffsdatum: 03.01.2009

A μPI Benutzerbeschreibung

In diesem Kapitel wird beschrieben, wie das μPI benutzt wird. Es werden alle Register, deren Signale und deren Eigenschaften beschrieben. Außerdem wird der verwendete Bus beschrieben.

A.1 APB Interface

Das APB Interface ist nach den Spezifikationen des AMBA 3 APB Protokolls, welches in Kapitel 3.1.1 beschrieben wird, modelliert. Auch wenn die APB Spezifikationen mehrere Takte für die Zugriffs-Phase zulassen, arbeitet diese Implementation des APB-Interface die Zugriffsphase in nur einen Takt für alle Lese- und Schreibzugriffe ab. Dies sind zusammen mit der Setup-Phase insgesamt zwei Takte für einen kompletten Lese- oder Schreibzugriff.

Das APB-Interface setzt ”PSLVERR”, wenn eine Adresse außerhalb des gültigen Bereiches beschrieben oder gelesen werden soll. Siehe dazu Kapitel A.4. Weiterhin leitet das APB-Interface die Fehlerrückmeldung der einzelnen Register oder die Standardwerte, welche vom Generator (A.5) gesetzt sind, durch.

A.2 Register- und Signalnamensschema

In diesem Abschnitt werden die Bedeutungen der Abkürzen der einzelnen Signale beschreiben. Dies hilft dabei, die Signalnamen der danach folgenden Register schneller ihren Aufgaben zuzuordnen.

Alle Register, welche es in einer vom Bus und einer von der Peripherie beschreibbaren Version gibt, beginnen mit ”P2B” oder ”B2P”. Dies gibt die Datenrichtung an. Ein ”P” steht dabei für Peripherie und ein ”B” für Bus.

Fast alle Signale der Register beginnen mit ”R” gefolgt von ”P2R”, ”R2P”, ”B2R” oder ”R2B”. Das erste ”R” gibt die Zugehörigkeit zum Register an. Die folgenden

Buchstaben geben die Richtung an, in welche die Signale fließen. Das ”P” und das ”B” stehen wieder für Peripherie und Bus. Ein ”R” steht für Register. Die Sichtweisen ”IN” und ”OUT” beziehen sich immer vom Standpunkt des Registers. Ein ”IN” meint also ein Signal, das von außen in das Register fließt. Ein ”OUT” meint ein Signal, das von einem Register kommt.

Alle Register verfügen über die Signale CLK und RESETN. Beide Signale sind vom Typ std_ulogic und gehen in das Register hinein. Das Resetsignal ist low-aktiv. Diese beiden Signale werden bei den Registern nicht noch einmal extra aufgeführt.

Alle Register, die vom Bus aus beschrieben werden - und einige andere Register - besitzen Data-Input-Enables (DI_EN). Nur wenn dieses high ist, werden mit dem nächsten Takt Daten ins Register übernommen. Alle Register, die dieses Signal nicht besitzen und von der Peripherie beschrieben werden, übernehmen die neuen Daten bei jedem neuen Takt.

Einige Register besitzen das Data-Processed-Signal (DP). Dieses dient dazu, dem Register mitzuteilen, dass die zum Lesen anliegenden Daten verarbeitet wurden und neue Daten angelegt werden sollen. Dies führt bei Registern mit Reset on Read dazu, dass die Daten am Ausgang mit dem nächsten Takt gelöscht werden. Bei Pufferregistern wird mit dem nächsten Takt der nächste gespeicherte Datensatz an den Ausgang angelegt. Ist der Puffer leer, ist der Wert des Ausganges nicht definiert. Ein Error-Signal teilt diesen Zustand mit.

Einige Register verfügen über Konfigurationsmöglichkeiten, die während der Hardwaredesignzeit festgelegt werden können. Dafür stehen bis zu zwei Generics zur Verfügung, einer vom Typ std_ulogic und einer vom Typ integer. Der std_ulogic Wert stellt dabei meist die Funktionalität des Reset on Read Signales ein bzw. aus. Der Integerwert ist bei den Pufferregistern für deren interne Größe und bei den Counterregistern für deren Maximalwert zuständig.

A.3 Register

Im Folgenden wird der Funktionsumfang der Register beschrieben, außerdem werden die vorhanden Signale erklärt.

A.3.1 Einfaches Datenregister

Dieses Register ist das einfachste Register. Es ermöglicht einen direkten Datenaustausch zwischen Peripherie und Bus. Daten, die hineingeschrieben werden, stehen einen Takt später zum Auslesen bereit. Ein Lesezugriff ändert den Inhalt des Registers nicht. Dieses einfache Datenregister gibt es in zwei Versionen. Einmal ist es so gebaut, dass der Bus Schreibrechte hat. In der anderen Version hat die Peripherie den Schreibzugriff.

A.3.1.1 p2b_dataregister

In Tabelle A.1 ist eines der beiden einfachen Datenregister dargestellt. Der Datenstrom fließt von der Peripherie zum Bus. Es gibt keine einstellbaren Werte.

Name	Typ	Richtung	Bits[1]	Beschreibung
R_R2P_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten von der Peripherie aus
R_P2R_DATA	std_ulogic_vector	IN	32	Schreibender Zugriff auf die Daten von der Peripherie aus. Anliegende Daten werden mit dem nächsten Takt übernommen
R_R2B_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten vom Bus aus

Tabelle A.1: Signale des p2b_dataregisters

[1] 32 Bits stehen für die jeweilige Systembusbreite.

A.3.1.2 b2p_dataregister

Tabelle A.2 zeigt ist eines der beiden einfachen Datenregister. Der Datenstrom fließt vom Bus zur Peripherie. Es gibt keine einstellbaren Werte.

Name	Typ	Richtung	Bits[2]	Beschreibung
R_R2P_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten von der Peripherie aus
R_B2R_DATA	std_ulogic_vector	IN	32	Schreibender Zugriff auf die Daten vom Bus aus. Anliegende Daten werden mit dem nächsten Takt übernommen, in dem R_B2R_DI_EN high ist
R_R2B_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten vom Bus aus
R_B2R_DI_EN	std_ulogic	IN	1	Writeenable für den Bus

Tabelle A.2: Signale des b2p_dataregisters

A.3.2 Einfaches Kontrollregister

Das einfache Kontrollregister unterscheidet sich im Verhalten nicht vom einfachen Datenregister, das nur vom Bus aus beschreibbar ist. Das einfache Kontrollregister bietet jedoch die Möglichkeit, dass zur Hardwaredesignzeit festgelegt werden kann, welchen Wert das Register nach einem Reset besitzt.

[2] 32 Bits stehen für die jeweilige Systembusbreite.

A.3.2.1 controlregister

In Tabelle A.3 folgen die Signale für das Kontrollregister.

Name	Typ	Richtung	Bits[3]	Beschreibung
R_R2P_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten von der Peripherie aus
R_B2R_DATA	std_ulogic_vector	IN	32	Schreibender Zugriff auf die Daten vom Bus aus. Anliegende Daten werden mit dem nächsten Takt übernommen, in dem R_B2R_DI_EN high ist
R_R2B_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten vom Bus aus
R_B2R_DI_EN	std_ulogic	IN	1	Writeenable für den Bus

Tabelle A.3: Signale des controlregisters

Das Kontrollregister bietet folgenden Generic (Tab. A.4), um den Wert nach einem Reset festzulegen.

Name	Typ	Standard	Beschreibung
RESET_VALUE	integer	0	Wert des Registers nach einem Reset

Tabelle A.4: Generics des controlregisters

A.3.3 Reset on Read Kontrollregister

Dieses Kontrollregister lässt sich ebenfalls nur vom Bus beschreiben. Es bietet jedoch die Möglichkeit, von der Peripherie zurückgesetzt zu werden. Damit kann der CPU zum Beispiel der Status über einen Abarbeitungszustand eines Befehls mitgeteilt werden. Genauso wie das einfache Kontrollregister (A.3.2) gibt es die Möglichkeit, den Wert, welches das Register nach einem Reset annimmt, einzustellen.

[3] 32 Bits stehen für die jeweilige Systembusbreite.

A.3.3.1 ror_controlregister

In Tabelle A.5 folgen die Signale für das Reset on Read Kontrollregister.

Name	Typ	Richtung	Bits[4]	Beschreibung
R_R2P_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten von der Peripherie aus
R_P2R_DP	std_ulogic	IN	1	Daten verarbeitet - Wenn aktiv, wird das Register beim nächsten Takt zurückgesetzt
R_B2R_DATA	std_ulogic_vector	IN	32	Schreibender Zugriff auf die Daten vom Bus aus. Anliegende Daten werden mit dem nächsten Takt übernommen, in dem R_B2R_DI_EN high ist
R_R2B_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten vom Bus aus
R_B2R_DI_EN	std_ulogic	IN	1	Writeenable für den Bus

Tabelle A.5: Signale des ror_controlregisters

Es gibt einen Generic (Tab. A.6), welcher den Wert des Registers nach dem Reset fest legt.

Name	Typ	Standard	Beschreibung
RESET_VALUE	integer	0	Wert des Registers nach einem Reset

Tabelle A.6: Generics des ror controlregisters

A.3.4 Einfaches Statusregister

Das einfache Statusregister unterscheidet sich nicht vom einfachen Datenregister, das nur von der Peripherie beschreibbar ist. Es soll durch seinen Namen dem Entwickler helfen die Ausgaben von Status- und Datenregistern klarer zu trennen.

[4] 32 Bits stehen für die jeweilige Systembusbreite.

A.3.4.1 statusregister

In Tabelle A.7 folgen die Signale für das Statusregister. Es gibt keine einstellbaren Werte.

Name	Typ	Richtung	Bits[5]	Beschreibung
R_R2P_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten von der Peripherie aus
R_P2R_DATA	std_ulogic_vector	IN	32	Schreibender Zugriff auf die Daten von der Peripherie aus. Anliegende Daten werden mit dem nächsten Takt übernommen
R_R2B_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten vom Bus aus

Tabelle A.7: Signale des statusregisters

A.3.5 Reset on Read Statusregister

Dieses Statusregister ist nur von der Peripherie beschreibbar. Es wird jedoch bei einem Lesezugriff durch den Prozessor zurückgesetzt, womit die CPU bei einem erneutem Lesezugriff weiß, ob sie einen Zustand schon verarbeitet hat.

A.3.5.1 ror_statusregister

In Tabelle Tab. A.8 folgen die Signale für das Reset on Read Statusregister. Es gibt keine einstellbaren Werte.

[5] 32 Bits stehen für die jeweilige Systembusbreite.

Name	Typ	Richtung	Bits[6]	Beschreibung
R_R2P_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten von der Peripherie aus
R_P2R_DATA	std_ulogic_vector	IN	32	Schreibender Zugriff auf die Daten von der Peripherie aus. Anliegende Daten werden mit dem nächsten Takt übernommen
R_R2B_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten vom Bus aus
R_B2R_DP	std_ulogic	IN	1	Daten verarbeitet vom Bus aus - wenn aktiv wird das Register zurück gesetzt

Tabelle A.8: Signale des ror_statusregisters

A.3.6 Impulse Statusregister

Wie alle Statusregister kann das Impulse Statusregister nur von der Peripherie beschrieben werden. Anders als bei den anderen Statusregistern können jedoch nur positive Bits in dieses Register geschrieben werden. Wird nach dem Schreiben eines high Bits ein low Bit angelegt, bleibt das high Bit erhalten. Die Bits werden durch einen Lesezugriff vom Bus gelöscht (Reset on Read).

Ist mindestens ein Bit innerhalb des Registers high, wird ein Interrupt nach außen weitergeleitet, welcher durch Verwendung des Haupt IRQ-Register (A.3.11) verwendet werden kann.

Bei diesem Statusregister kann zur Hardwaredesignzeit festgelegt werden, welchen Wert das Register nach einem Reset annimmt.

[6] 32 Bits stehen für die jeweilige Systembusbreite.

A.3.6.1 impulse_statusregister

In Tabelle A.9 folgen die Signale für das Impulse Statusregister.

Name	Typ	Richtung	Bits[7]	Beschreibung
R_R2P_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten von der Peripherie aus
R_P2R_DATA	std_ulogic_vector	IN	32	Schreibender Zugriff auf die Daten von der Peripherie aus. Anliegende Daten werden mit dem nächsten Takt übernommen
R_R2B_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten vom Bus aus
R_B2R_DP	std_ulogic	IN	1	Daten verarbeitet vom Bus aus - wenn aktiv wird das Register zurück gesetzt
R_IRQ_OUT	std_ulogic	OUT	1	Interruptausgang - wenn dieser Interrupt benötigt wird, muss dieses Signal mit dem Haupt IRQ-Register verbunden werden

Tabelle A.9: Signale des impulse_statusregisters

Das impulse_statusregister bietet folgenden Generic (Tab. A.10), um den Wert nach einem Reset festzulegen.

Name	Typ	Standard	Beschreibung
RESET_VALUE	integer	0	Wert des Registers nach einem Reset

Tabelle A.10: Generics des ror controlregister

[7] 32 Bits stehen für die jeweilige Systembusbreite.

A.3.7 Negative Impulse Statusregister

Dieses Register erfüllt den gleichen Funktionsumfang wie das Impulse Statusregister (A.3.6), mit dem Unterschied, dass nur low Bits in das Register geschrieben werden können und eine Leseoperationen vom Bus alle Register auf high setzt.

Ist mindestens ein Bit innerhalb des Registers low, wird ein Interrupt nach außen weitergeleitet, welcher durch Verwendung des Haupt IRQ-Register (A.3.11) verwendet werden kann.

Ebenso wie beim Impulse Statusregister (A.3.6) besteht beim Negative Impulse Statusregister die Möglichkeit den Wert, den das Register nach einem Reset annimmt, zur Hardwaredesignzeit festzulegen.

A.3.7.1 negative_impulse_statusregister

In Tabelle A.11 folgen die Signale für das Negative Impulse Statusregister.

Name	Typ	Richtung	Bits[8]	Beschreibung
R_R2P_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten von der Peripherie aus
R_P2R_DATA	std_ulogic_vector	IN	32	Schreibender Zugriff auf die Daten von der Peripherie aus. Anliegende Daten werden mit dem nächsten Takt übernommen
R_R2B_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten vom Bus aus
R_B2R_DP	std_ulogic	IN	1	Daten verarbeitet vom Bus aus - wenn aktiv wird das Register zurück gesetzt
R_IRQ_OUT	std_ulogic	OUT	1	Interruptausgang - wenn dieser Interrupt benötigt wird, muss dieses Signal mit dem Haupt IRQ-Register verbunden werden

Tabelle A.11: Signale des negative_impulse_statusregister

[8] 32 Bits stehen für die jeweilige Systembusbreite.

Es gibt einen Generic (Tab. A.12), welcher den Wert des Registers nach dem Reset fest legt.

Name	Typ	Standard	Beschreibung
RESET_VALUE	integer	0	Wert des Registers nach einem Reset

Tabelle A.12: Generics des ror controlregister

A.3.8 Saturation Counterregister

Dieses Register zählt jeden Takt, zu dem ein Zählimpuls angelegt wird, seinen internen Counter eins hoch. Die maximale Zählfrequenz ist somit auf die Frequenz der Peripherie beschränkt. Der aktuelle Wert des Counters kann sowohl durch die CPU als auch die Peripherie ausgelesen werden. Der Counter kann von der Peripherie auch direkt gesetzt werden.

Zur Hardwaredesignzeit wird ein Maximalwert eingestellt. Ist dieser voreingestellte Maximalwert erreicht, stoppt der Counter.

Zur Entwicklungszeit der Hardware wird bestimmt, ob Reset on Read für das Lesen vom Bus aktiv sein soll. Wurde Reset on Read aktiviert, wird bei jeder Leseoperation vom Bus der Wert des Counters zurückgesetzt. Weiterhin wird bei aktivem Reset on Read und Erreichen des maximalen Wertes ein Interrupt nach außen weitergeleitet, welcher durch Verwendung des Haupt IRQ-Register (A.3.11) verwendet werden kann.

A.3.8.1 saturation_counterregister

In Tabelle A.13 folgen die Signale für den Saturation Counter.

Name	Typ	Richtung	Bits[9]	Beschreibung
R_R2P_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten von der Peripherie aus
R_P2R_DATA	std_ulogic_vector	IN	32	Schreibender Zugriff auf den Counter von der Peripherie aus. Anliegende Daten werden nur bei aktivem Enable übernommen
R_P2R_DI_EN	std_ulogic_vector	IN	32	Writeenable - um den Counter zu setzen
R_P2R_COUNT	std_ulogic	IN	1	Löst beim nächsten Takt einen Zählimpuls aus
R_R2B_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten vom Bus aus
R_B2R_DP	std_ulogic	IN	1	Daten verarbeitet vom Bus aus - wenn Reset on Read aktiv wird der Counter zurückgesetzt
R_IRQ_SAT	std_ulogic	OUT	1	Interruptausgang - wenn dieser Interrupt benötigt wird, muss dieses Signal mit dem Haupt IRQ-Register verbunden werden

Tabelle A.13: Signale des saturation_counterregister

Das Saturation Counterregister bietet folgende Generics (Tab. A.14), um das Verhalten einzustellen.

Name	Typ	Standard	Beschreibung
SATCNT_REG_MAX_VALUE	integer	255	Maximalwert des Counters. (Von 0 bis $2^{32}-1$ einstellbar)
SATCNT_REG_ROR	std_ulogic	’1’	Wenn 1, ist Reset on Read aktiv

Tabelle A.14: Generics des saturation_counterregister

[9] 32 Bits stehen für die jeweilige Systembusbreite.

A.3.9 Round Rotating Counterregister

Dieses Register zählt jeden Takt, zu dem ein Zählimpuls angelegt wird, seinen internen Counter eins hoch. Die maximale Zählfrequenz ist somit auf die Frequenz der Peripherie beschränkt. Der aktuelle Wert des Counters kann sowohl durch die CPU als auch die Peripherie ausgelesen werden. Der Counter kann von der Peripherie auch direkt gesetzt werden.

Ist ein voreingestellter Maximalwert erreicht, beginnt der Counter wieder von vorne und setzt das MSB als Overflowstatus. Dadurch wird der maximale Wert des Counters auf $2^{31} - 1$ beschränkt. Der Maximalwert wird zur Entwicklungszeit festgelegt und kann zur Laufzeit nicht geändert werden.

Zur Entwicklungszeit der Hardware wird bestimmt, ob ein Reset on Read für das Lesen vom Bus aktiv sein soll. Wurde Reset on Read aktiviert, wird bei jeder Leseoperation vom Bus der Wert und der Overflowstatus des Counters zurückgesetzt. Weiterhin wird bei aktivem Reset on Read und gesetztem Overflowbit ein Interrupt nach außen weitergeleitet, welcher durch Verwendung des Haupt IRQ-Register (A.3.11) verwendet werden kann.

A.3.9.1 round_rotating_counterregister

In Tabelle A.15 folgen die Signale für das Round Rotating Counterregister.

Name	Typ	Richtung	Bits[10]	Beschreibung
R_R2P_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten von der Peripherie aus
R_P2R_DATA	std_ulogic_vector	IN	32	Schreibender Zugriff auf den Counter von der Peripherie aus. Anliegende Daten werden nur bei aktivem Enable übernommen
R_P2R_DI_EN	std_ulogic_vector	IN	32	Writeenable - um den Counter zu setzen
R_P2R_COUNT	std_ulogic_vector	IN	1	Löst beim Takt einen Zählimpuls aus
R_R2B_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten vom Bus aus
R_B2R_DP	std_ulogic	IN	1	Daten verarbeitet vom Bus aus - wenn Reset on Read aktiv wird der Counter zurückgesetzt
R_IRQ_OVF	std_ulogic	OUT	1	Interruptausgang - wenn dieser Interrupt benötigt wird, muss dieses Signal mit dem Haupt IRQ-Register verbunden werden

Tabelle A.15: Signale des round_counterregister

Das Round Rotating Counterregister bietet, die in Tabelle A.16 gezeigten, Generics, um das Verhalten einzustellen.

Name	Typ	Standard	Beschreibung
RNDCNT_REG_MAX_VALUE	integer	255	Maximalwert des Counters. (Von 0 bis $2^{31} - 1$ einstellbar)
RNDCNT_REG_ROR	std_ulogic	'1'	Wenn 1, ist Reset on Read aktiv

Tabelle A.16: Generics des round_rotating_counterregister

[10] 32 Bits stehen für die jeweilige Systembusbreite.

A.3.10 Pufferregister

Diese Register bieten unter einer Adresse mehrere hintereinander liegende Register an. Damit wird innerhalb des Registers ein FIFO gebildet. Eine Schreiboperation beschreibt immer das nächste freie Register. Am Ausgang liegen immer die ältesten nicht verarbeiteten Werte an. Ist kein Wert gespeichert, ist der Wert des Ausgangs undefiniert.

Das Pufferregister gibt es in mehreren Versionen. Ein Unterschied besteht darin, ob die Register vom Bus oder der Peripherie beschrieben werden können. Die jeweils andere Seite kann das FIFO-Register wieder auslesen. Wird von der Seite gelesen, welche die Schreibrechte auf das Register besitzt, erhält man die Anzahl der noch ungelesenen Datensätze im Register.

Der andere Unterschied zwischen den Versionen spiegelt sich nur im internen Aufbau wider. Das Verhalten nach außen ist identisch. Es kann also jederzeit der Registertyp gewechselt werden, ohne das die Software oder andere Teile der Hardware angepasst werden müssen. Die Version, welche für kleinere FIFOs gedacht ist, verwendet intern ein Schieberegister. Datensätze, welche gelesen wurden, werden so einfach rausgeschoben. Außerdem gibt es noch einen Zeiger für Schreiboperationen. Die Version, welche für größere FIFOs gedacht ist, benutzt intern zwei Zeiger. Dadurch wird etwas mehr Hardware im FPGA benötigt, jedoch kann durch das Vermeiden des Schieberegisters internes Blockram verwendet werden. Die Verwendung von internem Blockram kann durch Einstellungen im Synthesetool veranlasst werden.

Um Werte im Pufferregister zu speichern, müssen die Werte am Dateneingang anliegen und es muss das entsprechende Data-Input-Enable (DI_EN) für einen Takt high sein. Die Daten werden mit dem Takt übernommen. Ist ein Datensatz fertig verarbeitet, wird Data-Processed (DP) Signal für einen Takt auf high gelegt. Die Daten werden dann mit dem Takt gelöscht und es werden die nächst älteren Daten an den Ausgang angelegt.

Ist das FIFO voll geht das entsprechende Fehlersignal auf high und es können keine weiteren Daten im Register gespeichert werden. Ist das FIFO leer, ist der Wert des Ausganges nicht definiert. Auch hierfür gibt es ein entsprechendes Fehlersignal, welche dann auf high geht.

Die Größe des FIFOs wird während des Hardwaredesigns festgelegt.

A.3.10.1 b2p_bufferregister

Es folgen die Signale (Tab. A.17) des Pufferregisters, welches vom Bus aus beschrieben wird und intern mit einem Schieberegister arbeitet. Es hat die gleichen Signale und das gleiche Verhalten wie das b2p_bufferregister_b (A.3.10.2).

Name	Typ	Richtung	Bits[11]	Beschreibung
R_R2P_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die ältesten Daten im FIFO
R_P2R_DP	std_ulogic	IN	1	Daten verarbeitet - entfernt die ältesten Daten aus dem FIFO
R_R2P_ERR	std_ulogic	OUT	1	Fehlerrückmeldung - ist high, wenn keine Daten im FIFO vorhanden sind
R_B2R_DATA	std_ulogic_vector	IN	32	Schreibender Zugriff vom Bus aus - um neue Daten ins FIFO zu schreiben
R_R2B_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Anzahl nicht verarbeiteter Daten im FIFO
R_B2R_DI_EN	std_ulogic	IN	1	Writeenable vom Bus aus - wenn aktiv, werden neue Werte übernommen
R_R2B_DI_ERR	std_ulogic	OUT	1	Fehlerrückmeldung - ist high, wenn der FIFO voll ist

Tabelle A.17: Signale des b2p_bufferregister

Das b2p_bufferregister Register verfügt über ein Generic (Tab. A.18), über den die Größe des Puffers eingestellt werden kann.

[11] 32 Bits stehen für die jeweilige Systembusbreite.

Name	Typ	Standard	Beschreibung
BUFFERREG_NUM	integer	4	Anzahl der internen Register

Tabelle A.18: Generics des b2p_bufferregister

A.3.10.2 b2p_bufferregister_b

In Tabelle A.19 folgen die Signale des Pufferregisters, welches vom Bus aus beschrieben wird und intern mit zwei Zeigern arbeitet. Es hat die gleichen Signale und das gleiche Verhalten wie das b2p_bufferregister (A.3.10.1).

Name	Typ	Richtung	Bits[12]	Beschreibung
R_R2P_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die ältesten Daten im FIFO
R_P2R_DP	std_ulogic	IN	1	Daten verarbeitet, entfernt die ältesten Daten aus dem FIFO
R_R2P_ERR	std_ulogic	OUT	1	Fehlerrückmeldung - ist high, wenn keine Daten im FIFO vorhanden sind
R_B2R_DATA	std_ulogic_vector	IN	32	Schreibender Zugriff vom Bus aus, um neue Daten ins FIFO zu schreiben
R_R2B_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Anzahl nicht verarbeiteter Daten im FIFO
R_B2R_DI_EN	std_ulogic	IN	1	Writeenable vom Bus aus, wenn aktiv, werden neue Werte übernommen
R_R2B_DI_ERR	std_ulogic	OUT	1	Fehlerrückmeldung - ist high, wenn der FIFO voll ist

Tabelle A.19: Signale des b2p_bufferregister_b

[12] 32 Bits stehen für die jeweilige Systembusbreite.

Das b2p_bufferregister_b Register verfügt über ein Generic (Tab. A.20), über den die Größe des Puffers eingestellt werden kann.

Name	Typ	Standard	Beschreibung
BUFFERREG_NUM	integer	4	Anzahl der internen Register

Tabelle A.20: Generics des b2p_bufferregister_b

A.3.10.3 p2b_bufferregister

Hier folgen die Signale (Tab. A.21) des Pufferregisters, welches von der Peripherie aus beschrieben wird und intern mit einem Schieberegister arbeitet. Es hat die gleichen Signale und das gleiche Verhalten wie das p2b_bufferregister_b (A.3.10.4).

Name	Typ	Richtung	Bits[13]	Beschreibung
R_R2P_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Anzahl gespeicherter Daten im FIFO
R_P2R_DATA	std_ulogic_vector	IN	32	Schreibender Zugriff auf die Daten im FIFO. Neue Daten werden nur bei aktivem Enable übernommen
R_P2R_DI_EN	std_ulogic	IN	1	Writeenable - wenn aktiv, werden neue Daten ins FIFO übernommen
R_R2P_ERR	std_ulogic	OUT	1	Fehlerrückmeldung - ist high, wenn der FIFO voll ist
R_R2B_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die ältesten Daten im FIFO
R_B2R_DP	std_ulogic	IN	1	Daten verarbeitet vom Bus aus - wenn aktiviert, wird der älteste Datensatz im FIFO entfernt
R_R2B_DO_ERR	std_ulogic	OUT	1	Fehlerrückmeldung - ist high, wenn der FIFO leer ist

Tabelle A.21: Signale des p2b_bufferregister

Das p2b_bufferregister Register verfügt über ein Generic (Tab. A.22), über den die Größe des Puffers eingestellt werden kann.

Name	Typ	Standard	Beschreibung
BUFFERREG_NUM	integer	4	Anzahl der internen Register

Tabelle A.22: Generics des p2b_bufferregister

[13] 32 Bits stehen für die jeweilige Systembusbreite.

A.3.10.4 p2b_bufferregister_b

In Tabelle A.23 folgen die Signale des Pufferregisters, welches von der Peripherie aus beschrieben wird und intern mit zwei Zeigern arbeitet. Es hat die gleichen Signale und das gleiche Verhalten wie das p2b_bufferregister (A.3.10.3).

Name	Typ	Richtung	Bits[14]	Beschreibung
R_R2P_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Anzahl gespeicherter Daten im FIFO
R_P2R_DATA	std_ulogic_vector	IN	32	Schreibender Zugriff auf die Daten im FIFO - Neue Daten werden nur bei aktivem Enable übernommen
R_P2R_DI_EN	std_ulogic	IN	1	Writeenable - wenn aktiv, werden neue Daten ins FIFO übernommen
R_R2P_ERR	std_ulogic	OUT	1	Fehlerrückmeldung - ist high, wenn der FIFO voll ist
R_R2B_DATA	std_ulogic_vector	32	OUT	Lesender Zugriff auf die ältesten Daten im FIFO
R_B2R_DP	std_ulogic	IN	1	Daten verarbeitet vom Bus aus - wenn aktiviert, wird der älteste Datensatz im FIFO entfernt
R_R2B_DO_ERR	std_ulogic	OUT	1	Fehlerrückmeldung - ist high, wenn der FIFO leer ist

Tabelle A.23: Signale des p2b_bufferregister_b

[14] 32 Bits stehen für die jeweilige Systembusbreite.

Das p2b_bufferregister_b Register verfügt über ein Generic (Tab. A.24), über den die Größe des Puffers eingestellt werden kann.

Name	Typ	Standard	Beschreibung
BUFFERREG_NUM	integer	4	Anzahl der internen Register

Tabelle A.24: Generics des p2b_bufferregister_b

A.3.11 Haupt IRQ-Register

Dieses Register bündelt bis zu 32 Interrupts der anderen Register und der Peripherie und gibt sie als ein Interrupt weiter. Im Falle eines Interrupts muss dieses Register zuerst ausgelesen werden, um die genaue Quelle des Interrupts zu erfahren.

Das Haupt IRQ-Register kann im gesamten μPI nur einmal verwendet werden. Um einzelne Interrupts ein- und ausschalten zu können, wird ein einfaches Kontrollregister benötigt, welches mit dem Haupt IRQ-Register verbunden wird.

Zur Hardwaredesignzeit wird festgelegt, ob Reset on Read aktiviert ist. Bei einem Reset on Read wird der nach außen weitergegebene Interrupt zurückgenommen. Ein erneutes Auslesen des Registers, ohne das sich der Zustand eines IRQs geändert hat, liefert 0x00. Ändert sich einer der Interrupteingänge, werden alle noch anstehenden Interrupts wieder angezeigt.

A.3.11.1 mainirqregister

In Tabelle A.25 folgen die Signale des Haupt IRQ-Register, welches die IRQs des Peripheriebausteins verwaltet.

Name	Typ	Richtung	Bits[15]	Beschreibung
R_R2P_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten von der Peripherie aus
R_R2B_DATA	std_ulogic_vector	OUT	32	Lesender Zugriff auf die Daten vom Bus aus
R_B2R_DP	std_ulogic	IN	1	Daten verarbeitet vom Bus aus - wird für Reset on Read benötigt
R_IRQ_OUT	std_ulogic	OUT	1	Gesamtinterrupt, der mit dem Interruptcontroller des Prozessors verbunden werden muss
R_IRQS_IN	std_ulogic_vector	IN	32	IRQ-Eingänge. Zu verbinden mit anderen IRQ-Ausgängen anderer Register und der Peripherie
R_IRQ_EN	std_ulogic_vector	IN	32	Interruptenable-Inputs - muss mit einem einfachen Kontrollregisterausgang verbunden werden, um einzelne IRQs ein- und ausschalten zu können (siehe unten)

Tabelle A.25: Signale des mainirqregister

Das Haupt IRQ-Register bietet ein Generic (Tab. A.26).

Name	Typ	Standard	Beschreibung
MAINIRQ_REG_ROR	std_ulogic	’0’	Wenn 1, ist Reset on Read aktiv

Tabelle A.26: Generics des mainirqregister

Einige Register bieten R_IRQ_OUT als Signal, um einen Interrupt zu melden.

[15] 32 Bits stehen für die jeweilige Systembusbreite.

Da es keine Möglichkeit gibt, diese Signale automatisch mit dem mainirqregister zu verbinden, muss dieses innerhalb der Peripherie von Hand geschehen. Das mainirqregister liefert dafür das Signal R_IRQS_IN. R_IRQS_IN ist mit R_IRQ_OUT der Register zu verbinden. Das Signal R_IRQ_EN, mit dessen Hilfe einzelne Interrupts ein- und ausgeschaltet werden können, wird am besten mit dem peripherieseitigem Ausgang eines einfachen Kontrollregisters verbunden, damit einzelne Interrupts von der Software (de)aktiviert werden können.

A.4 Globale Definitionen

In der Datei defs.vhd werden alle globalen Konstanten definiert. Einige können leicht geändert werden, andere erfordern ein Umschreiben der Register.

Es gibt drei allgemeine globale Definitionen. Die erste ist ”REG_WIDTH”. Sie gibt die Breite der Register und vieler anderer Signale, wie die Adress- und Datenbusbreite an. Dieser Wert kann nicht ohne Eingriffe in einige Register verändert werden. Einige Register müssen intern leicht angepasst werden, damit sie mit einer anderen Bitbreite arbeiten können.

Der nächste Wert ist ”REG_NUM” und muss auf die Anzahl der verwendeten Register eingestellt werden. Der größte getestete Wert ist 1024. Der Wert ”REG_ADDR_WIDTH” kann entsprechend der Anzahl der verwendeten Register angepasst werden. Er spiegelt die Breite des internen Adressbusses wieder. Der Wert errechnet sich $log_2(REG_NUM) = REG_ADDR_WIDTH$.

Für den APB-Bus gibt es die Definition ”APB_ADDR”. Diese gibt die Startadresse der Peripherie im System an. Das APB-Interface reagiert nur auf Anfragen, die innerhalb von APB_ADDR bis $APB_ADDR + (REG_NUM * 4)$ liegen. Alle Lese- und Schreibzugriffe, die eine Adresse außerhalb dieses Bereiches haben, werden mit einer Fehlerrückmeldung abgewiesen. Da der APB-Bus für 32 Bit ausgelegt ist, bei einer Adresse jedoch immer vier Byte auf einmal angesprochen werden, muss ”APB_ADDR” eine durch vier teilbare Adresse enthalten.

Als letztes folgen einige Definitionen, welche für den Generator nötig sind. Für den Benutzer sind vor allem ”REG_LAYOUT_TYPE”, ”REG_LAYOUT_PAR1” und ”REG_LAYOUT_PAR2” wichtig. Mit ”REG_LAYOUT_TYPE” legt der Benutzer die Typen und die Reihenfolge der Register fest, welche er benutzen möchte. Es ist darauf zu achten, dass das Register als ”downto” definiert ist. Der Eintrag, welcher in der VHDL-Datei als erstes steht, hat die höchste Adresse. Mit ”REG_LAYOUT_PAR1” und ”REG_LAYOUT_PAR2” legt der Benutzer die

Parameter für die Register fest. ”REG_LAYOUT_PAR1” entspricht dabei dem Integerparameter der Register. ”REG_LAYOUT_PAR2” entspricht dem std_ulogic Parameter der Register. Welche Register welche Parameter berücksichtigen, steht in der Beschreibung der jeweiligen Register. Register, die einen oder beide Parameter nicht verarbeiten, ignorieren diese.

A.5 Der Generator

In der Datei generator.vhd werden die Register nach Vorgabe aus der defs.vhd Datei zusammengefügt. Wird ein neues Register entwickelt, muss dieses nach dem Vorbild der anderen Register in die generator.vhd Datei aufgenommen werden.

Der Generator übernimmt die Erstellung der Register und das Einbinden des Prozessorbus-Interfaces. Der Generator setzt für alle Register, die sich nicht selbst um die Fehlerrückmeldung an das Prozessorinterface kümmern, die Data-Input und Data-Output Errorsignale. Dabei ist der Fehlerwert bei einer Leseoperation low - also kein Fehler. Für alle Register, die vom Bus aus beschrieben werden können, ist der Standardfehlerwert ebenfalls low. Für alle Register, die nicht vom Bus aus beschrieben werden können, ist der Fehlerwert high.

Da das Board und der Prozessor noch nicht feststehen, steht die verwendete APB-Bridge auch noch nicht fest. Dies führt dazu, dass die einzelnen APB-Signale von Hand mit der ABP-Bridge verbunden werden müssen. Die Signale der Register stehen in Arrays zur Verfügung. Dabei ist zu beachten, das nicht alle Register alle Signale besitzen und somit einige Einträge in den Arrays nicht verbunden sind.

Weiterhin übernimmt der Generator nicht das Verbinden des Haupt IRQ-Register mit dem dazu nötigen Kontrollregister. Der Leseausgang des Kontrollregisters auf der Peripherieseite muss noch mit dem Signal ”G_MAIN_IRQ_EN” verbunden werden. Des weiteren müssen die Interruptausgänge (G_IRQ) der einzelnen Register mit dem Interrupteingang des Hauptinterruptregisters (G_MAIN_IRQ_IN) verbunden werden.

Wird kein Haupt IRQ-Register verwendet, ist G_MAIN_IRQ_OUT offen und hat keinen zugewiesenen Wert. Dies muss bei der Synthese beachtet werden. In der aktuellen Version von ISE (10.1.03) wird das Signal automatisch auf low gelegt. Ist dies bei einem anderen Synthesewerkzeug nicht der Fall, sollte das Signal zur Sicherheit auskommentiert werden.

Dadurch, das nicht alle Register alle im μPI vorgesehenen Signale benötigen, sind einige Signale ohne Zuweisung. Dadurch enstehen bei der Synthese sehr viele Warnungen. Diesen Warnungen sollte zumindest beim Wechsel des Synthesewerkzeuges Beachtung geschenkt werden, ob das Werkzeug diese am Ausgang offenen entfernt bzw. die am Eingang offenen Signale (nur bei nicht Verwendung des Hauptinterruptregisters vorhanden) auf einen definierten Wert legt. Das in dieser Bachelorarbeit genutzte Synthesewerkzeug ISE 10.1.03 hat diese Signale alle richtig erkannt und behandelt.

B Aufbau des Quelltextes

Als erstes muss gesagt werden, dass sich der gesamte Quelltext auf der beigelegten CD befindet, da er zum Ausdrucken zu umfangreich ist. Genauere Angaben dazu befinden sich im Anhang D.2.2.

Die meisten VHDL-Dateien sind nach dem selben Muster aufgebaut. Dieses Muster soll hier an einem Beispiel kurz erläutert werden. Als Vorlage wird hier das Saturation Counter Register genutzt.

Am Dateianfang befindet sich ein Hinweis auf den Autor und einen kurzen Kommentar zum Inhalt der VHDL-Datei. Im Anschluss folgt die Bekanntmachung der verwendeten Bibliotheken (vgl Auszug B.1).

```
1  --Author: Alexander Pautz (C) 2008 - 2009
2  --Student Technische Informatik an der HAW-Hamburg
3
4  --Dies ist das Saturation Counter, welche von der Peripherie
5  --geschrieben und von der Busseite gelesen wird.
6
7  library std;          -- "STANDARD" package as defined in the
8  use std.all;          -- VHDL 87/93 Language Reference Manual (LRM)
9  use std.standard.all;
10 use std.textio.all;
11
12 --weitere Bibliotheken ...
```

Abbildung B.1: Quelltextaufbau, Auszug 1

Die Entity enthält alle Ports und bei Registern bis zu zwei Generics (vgl. Auszug B.2). Über diese Generics werden für ein spezielles Register spezielle Eigenschaften gesetzt. Mehr dazu befindet sich bei der Beschreibung der einzelnen Register in Kapitel A.3. Alle anderen Entities wie das APB- und SPI-Interface besitzen keine Generics, diese beziehen spezielle Einstellungen aus den Dateien defs.vhd bzw. spi_defs.vhd.

```
1  entity SATURATION_COUNTERREGISTER is
2
3    generic (
4      --nur fuer Register, bis zu zwei Generics
5    );
6
7    port (
8      -- Port definitionen, mit CLK, RESETN und allen anderen
9      -- benoetigten Ports
10   );
11
12 end entity SATURATION_COUNTERREGISTER;
```

Abbildung B.2: Quelltextaufbau, Auszug 2

Zu jeder Entity gibt es eine Architektur, in der das Verhalten beschrieben wird. Am Anfang der Architektur werden alle Signale definiert, welche zum Speichern des internen Zustandes benötigt werden. Dabei gibt es jedes Signal zweimal, einmal als getaktetes und einmal als kombinatorisches Signal. Die kombinatorischen Signale dienen zum Übertragen der Informationen vom kombinatorischen zum getaktetem Prozess. Die getakteten Signale bilden die internen FlipFlops der Register. Die zusammengehörigen Signale tragen den selben Namen. Lediglich die Endung der Namen unterscheidet sich. Alle kombinatorischen Signale enden auf "_C". Alle getakteten Signale enden auf "_F".

In den meisten Architekturen gibt es genau zwei Prozesse. Davon ist der erste getaktet und sein Name endet auf "_FF". Der zweite Prozess ist rein kombinatorisch, in ihm werden die eigentlichen Berechnungen vorgenommen, sein Name endet auf "_C" (vgl. Kapitel 4.1.1). Aus Gründen der Übersicht werden innerhalb des kombinatorischen Prozesses alle Signale auf Variablen geführt. Genaueres dazu befindet sich in Kapitel 4.1.2. Die Variablen haben wiederum die gleichen Namen wie die Signale, enden aber auf "_V". Ein Beispiel zeigt der folgende Quelltext Auszug B.3.

```
1  architecture SATURATION_COUNTERREGISTER_ARC of
       SATURATION_COUNTERREGISTER is
2
3    --Getaktete Signale
4    --Getaktete Signale, diese enden alle auf _F
5
6    --Kombinatorische Signale
7    --Kombinatorische Signale, diese enden alle auf _C
```

```
8
9
10 begin
11
12   --Register, speichern oder zurueckgesetzen
13   REGISTER_FF : process (CLK)
14   begin
15     if (CLK = '1' AND CLK'event) then --'
16       if(RESETN = '0') then
17         --Reset ausgeloest
18         --Alle Register zuruecksetzen
19       else
20         --Kein Reset
21         --Die Werte der kombinatorischen Signale in die
22         --Register uebernehmen
23       end if;
24     end if;
25   end process REGISTER_FF;
26
27   --Kombinatorischer Prozess, fuehrt eigentlich Berechnung durch
28   REGISTER_C : process (COUNTERREGISTER_F, R_P2R_COUNT,
29                R_P2R_DI_EN, R_B2R_DP, R_P2R_DATA)
30
31   --Definition der benoetigten Variablen mit der Endung _V
32
33   begin
34     --Zuweisung der Defaultwerte fuer Variablen
35     --entweder ein Fester Wert oder der Wert des
36     --entsprechendem getakteten Signales
37
38     ----------------------------------------
39     --Eigentliche Aufgaben des Prozesses ausfuehren
40     ----------------------------------------
41
42     --Zuweisen der Variablen auf die kombinatorischen Signale
43   end process REGISTER_C;
44
45   --Eventuell getaktete Signale auf Ports nach aussen leiten
46
47 end architecture SATURATION_COUNTERREGISTER_ARC;
```

Abbildung B.3: Quelltextaufbau, Auszug 3

C Zusatzplatine mit MAX3232

Da das AT90CAN128-Board angepasst werden musste, damit es mit 3,3 Volt arbeitet und es keinen 3,3 Volt Pin kompatiblen Treiber für die serielle Schnittstelle zum MAX233 gibt, wird ein MAX3232 verwendet, welcher sich auf einer kleinen Zusatzplatine befindet. Der Schaltplan wurde bis auf die Anschlüsse und einen zusätzlichen Kondensator aus dem Datenblatt des MAX3232 übernommen ([vgl. Maxim Integrated Products, Inc., 2007]). Die Abbildung C.1 zeigt den Schaltplan. Im oberen Bereich ist der MAX3232 mit den Kondensatoren, die Anschlussklemme für die Mikrocontrollerseite und weiter rechts die Sub-D Buchse zu sehen. Im unterem Bereich ist die Spannungsversorgung mit dem Abblockkondensator C1 und Stabilisierungskondensator C2 zu sehen.

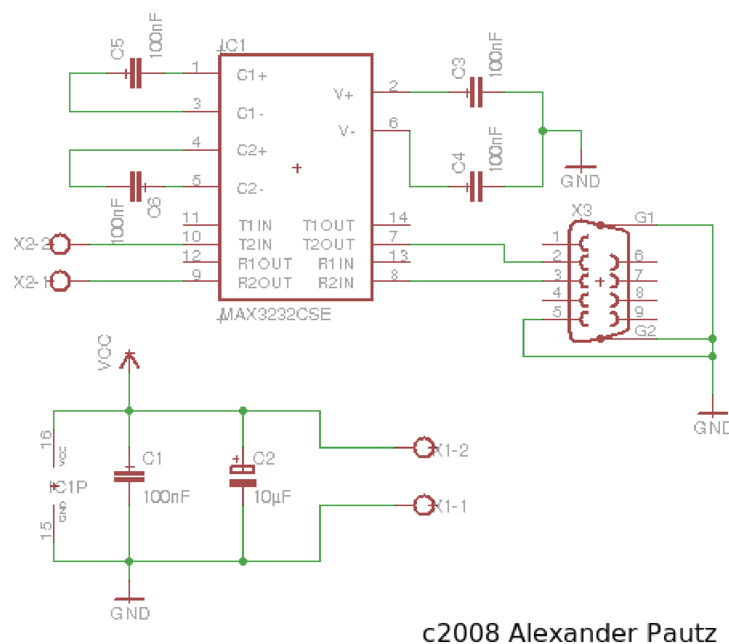
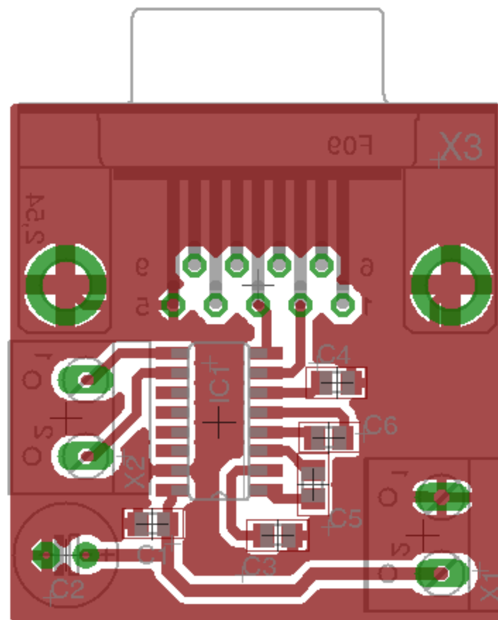


Abbildung C.1: Schaltplan der MAX3232 Zusatzplatine

Das Layout ist auf Abbildung C.2 zu sehen. In der Mitte sitzt der MAX3232. Links daneben befindet sich der Anschluss für die seriellen Daten vom Mikrocontroller. Die Sub-D Buchse oben ist für die Verbindung zum PC und rechts unten liegt der Anschluss für die Stromversorgung.



c2008 Alexander Pautz

Abbildung C.2: Layout der MAX3232 Zusatzplatine

Zum Abschluss folgen noch zwei Fotos (Abb. C.3) von der fertig aufgebauten MAX3232 Platine.

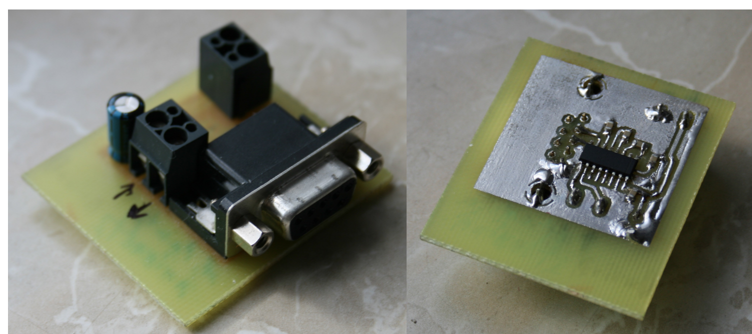


Abbildung C.3: Fotos der fertig aufgebauten MAX3232 Zusatzplatine

D CD-Inhalt

Es wird ein kurzer Überblick über den Inhalt über den CD-Inhalt gegeben.

D.1 Literatur

Dieser Ordner enthält alle Dokumente, welche für die Anfertigung dieser Bachelorarbeit verwendet wurden, frei verfügbar sind und in elektronischer Form vorliegen.

`Ordner: Literatur`

D.2 Quellcode

Dieser Ordner ist in mehrere Unterordner unterteilt, er enthält sowohl den Quelltext, welcher für den AVR geschrieben wurde, als auch den VHDL-Code des μPI und SPI.

`Ordner: Quellcode`

D.2.1 AVR-C-Code

In diesem Ordner liegt der gesamte Quelltext, der für den AVR geschrieben wurde. Der Quelltext wurde für den GNU Compiler geschrieben und lässt sich nur mit diesem fehlerfrei compilieren.

`Ordner: Quellcode/AVR-C-Code`

D.2.2 VHDL

Der VHDL-Code ist in verschiedene Abschnitte unterteilt. Es befinden sich nur die VHDL-Dateien ohne die erstellten Projekte auf der CD. Da ModelSim innerhalb der Projekte absolute Pfade verwendet, sind die Projekte auf einem anderem PC nicht zu gebrauchen. Alle Projektdaten von ISE wären zu umfangreich und würden den Rahmen der CD übersteigen. Neue Projekte für beide Programme können leicht erstellt werden. Beide Programme bieten die Möglichkeit bei der Erstellung von Projekten bereits vorhandenen Code zu integrieren.

`Ordner: Quellcode/VHDL`

D.2.2.1 1024-Register-Test

Dieser Ordner enthält alle für die Erstellung des in Kapitel 5.5 beschrieben 1024 Register-Tests benötigten Dateien. Die Dateien sollten nicht zur Erstellung von anderen Projekten, welche auf dem μPI basieren, genutzt werden, da die Dateien teilweise angepasst wurden.

`Ordner: Quellcode/VHDL/1024-Register-Test`

D.2.2.2 SPI

Im SPI-Ordner befinden sich das in dieser Bachelorarbeit entwickelte SPI-Interface mit allen dafür nötigen Dateien des μPIs. Aus diesen Dateien kann direkt ein neues Projekt für ModelSim oder ISE erstellt werden.

`Ordner: Quellcode/VHDL/SPI`

D.2.2.3 Testbenches

Um die Übersicht zu verbessern, wurden alle Testbenche der Register, des APB-Interfaces und des SPIs in diesen Ordner ausgelagert.

`Ordner: Quellcode/VHDL/Testbenches`

D.2.2.4 uPI

Der uPI-Ordner enthält alle Register-, die APB-Interface-, die Generator- und die Definitionsdatei, welche zum Erstellen von eigenen Anwendungen, die auf dem μPI basieren, nötig sind.

`Ordner: Quellcode/VHDL/uPI`

D.3 Sonstiges

Alles was in keinen der anderen Ordner passte, wurde in diesen Ordner gepackt. Dazu gehören der Schaltplan und das Layout der MAX3232 Zusatzplatine (siehe Kapitel C), in der Bachelorarbeit angesprochene Synthesereports und die Ausgabe des AVRs beim 8 Stunden Dauertest (siehe. Kapitel 6.4.6)

`Ordner: Sonstiges`

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 26. Mai 2009

Ort, Datum

Unterschrift