Hochschule für Angewandte Wissenschaften Hamburg
*Hamburg University of Applied Sciences*

# Bachelorarbeit

Saman Farshbaf-Masalehdan

## Abstandsregelung für ein autonomes Fahrzeug implementiert auf einer FPGA basierten SoC Plattform

*Fakultät Technik und Informatik*
*Department Informatik*

*Faculty of Engineering and Computer Science*
*Department of Computer Science*

Saman Farshbaf-Masalehdan

# Abstandsregelung für ein autonomes Fahrzeug implementiert auf einer FPGA basierten SoC Plattform

Bachelorarbeitarbeit 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. -Ing. B. Schwarz
Zweitgutachter : Prof. Dr. rer. nat. R. Baran

Abgegeben am 6. Juli 2009

**Saman Farshbaf-Masalehdan**

## Thema

Abstandsregelung für ein autonomes Fahrzeug implementiert auf einer FPGA basierten SoC Plattform

## Stichworte

System on Chip, Xilinx Spartan-3E FPGA, MicroBlaze Mikrocontroller, FreeRTOS, Digitaler PD-Regler, SPI, NET 1, IP-CORE

## Kurzzusammenfassung

Diese Bachelorarbeit befasst sich mit dem Aufbau einer FPGA basierten System on Chip Hardwareplattform für das SoC-CC Modellfahrzeug. Für den Entwurf der SoC Plattform wurden im Rahmen dieser Arbeit Hardware- und Software-Module für die seitliche Abstandserfassung entwickelt. Die Abstandswerte werden vom Einparkassistenten für die Abstandsregelung und Parklückensuche verwendet. Des Weiteren wurde eine drahtlose Verbindung zwischen SoC und Host-PC für Datenlogger Funktionalität aufgebaut. Das Echtzeitbetriebssystem FreeRTOS wurde auf eine MicroBlaze-Plattform portiert und Software-Module in dessen Task-System eingebettet.

**Saman Farshbaf-Masalehdan**

## Title of the paper

Distance controller for an autonomous vehicle implemented on FPGA based SoC platform

## Keywords

System on Chip, Xilinx Spartan-3E FPGA, MicroBlaze Microcontroller, FreeRTOS, digital PD-controller, SPI, NET 1, IP-CORE

## Abstract

This bachelor thesis deals with the construction of a FPGA based system-on-chip hardware platform for the SoC-CC model vehicle. For the design of the SoC platform, as part of this work hardware and software modules have been developed for the lateral distance detection. The distance values are used by parking assistant for the park distance control and parking space search. Furthermore a wireless connection between SoC and Host-PC for data logging purposes has been set up. The real time operating system FreeRTOS has been ported to a MicroBlaze-platfrom ans software modules have been embedded into its task system.

# Inhaltsverzeichnis

**1. Einleitung** **7**

**2. Autonome Einparkfunktion des SoC-CC-Fahrzeugs** **10**
2.1. Kinematisches Einspurfahrzeugmodell des SoC-CC-Fahrzeugs . . . . . . . . . . 10
2.2. Parklückensuche und Messung des zurückgelegten Weges mit den seitlichen Infrarotsensoren . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.3. Abstandsregelung mit einem digitalen PD-Regler für die Vorgabe des Lenkwinkels 12
2.3.1. Abstandsregelung mit einem digitalen PD-Regler . . . . . . . . . . . . . 12
2.3.2. Berechnung eines Abstandsdreiecks zur Auswahl zwischen virtueller und infrarotgestützter Positionsermittlung . . . . . . . . . . . . . . . . . . . . . 14

**3. System on Chip Hardwareplattform des SoC-CC-Fahrzeugs** **15**
3.1. Systemübersicht . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.1.1. Nexys 2 Board . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.1.2. MicroBlaze Softcore Prozessor . . . . . . . . . . . . . . . . . . . . . . 17
3.1.3. Cache und Speicher des MicroBlaze . . . . . . . . . . . . . . . . . . . 19
3.1.4. Bussysteme des Microblaze Prozessors . . . . . . . . . . . . . . . . . . 19
3.2. EDK Tool Chain . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.3. IP CORE Erstellung mit dem Peripheral Wizard des EDK . . . . . . . . . . . . 23

**4. Sensorplattform für die seitliche Abstandsmessung des SoC-CC-Fahrzeugs** **30**
4.1. HAW SPI IP CORE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.1.1. Standard Serial Peripheral Interface (SPI) . . . . . . . . . . . . . . . . 32
4.1.2. Transferformate des SPI Busses . . . . . . . . . . . . . . . . . . . . . 34
4.1.3. SPI Bustransfermessungen . . . . . . . . . . . . . . . . . . . . . . . . 36
4.2. Analog-Digital-Umsetzter National ADCS7476 . . . . . . . . . . . . . . . . . . 39
4.2.1. Timingverhalten des National ADCS7476 . . . . . . . . . . . . . . . . . 39
4.2.2. Timingverhalten des ADCS7476 mit HAW SPI IP CORE . . . . . . . . . 40
4.3. Infrarotsensor Sharp GP2D12 . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.3.1. Umrechnung der Ausgangsspannung $V_O$ in einen Abstand . . . . . . . . 42
4.4. Ermittlung der maximalen Abtastfrequenz des ADCS7476 mit SPI Transfers . . . 44

**5. Datenlogger und JTAG Programmierung über WLAN** **47**
5.1. Digilent NET 1 Kommunikationsmodul . . . . . . . . . . . . . . . . . . . . . . 48
5.2. Digilent Design Pattern nach dem EPP-Protokoll . . . . . . . . . . . . . . . . . 48
5.2.1. Adress Write . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.2.2. Adress Read . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.2.3. Data Write . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.2.4. Data Read . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.3. NET 1 IP CORE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
5.3.1. FSM Schaltbild und Zustandsdiagramm des EPP Controllers . . . . . . . 52

5.3.2. Write und Read Hardware IPIF FIFO . . . . . . . . . . . . . . . . . . . . 55
5.4. Linksys WET54G W-LAN Bridge . . . . . . . . . . . . . . . . . . . . . . . . . 58

**6. FreeRTOS Echtzeitbetriebssystem 60**
6.1. Dateistruktur des FreeRTOS . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
6.2. Kernel . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
6.2.1. Scheduler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
6.2.2. RTOS Tick . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
6.2.3. Speicherverwaltung . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
6.3. FreeRTOS Task . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
6.4. FreeRTOS Co-Routines . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
6.5. Interprozesskommunikation und Synchronisation . . . . . . . . . . . . . . . . 64
6.6. FreeRTOS Demoanwendung . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

**7. Zusammenfassung 68**

**Tabellenverzeichnis 69**

**Abbildungsverzeichnis 70**

**Literaturverzeichnis 73**

**A. HAW SPI IP CORE Beschreibung 76**
A.1. Einführung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
A.2. Funktion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
A.3. Funktionale Beschreibung . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
A.4. HAW SPI IP Core Design Parameter . . . . . . . . . . . . . . . . . . . . . . . 78
A.5. HAW SPI IP Core I/O Signale . . . . . . . . . . . . . . . . . . . . . . . . . . 79
A.6. HAW SPI IP Core Parameter-Port Abhängigkeiten . . . . . . . . . . . . . . . . 81
A.7. HAW SPI IP Core Registerbeschreibung . . . . . . . . . . . . . . . . . . . . . 82
A.7.1. HAW SPI Status Register (HAWSPISR) . . . . . . . . . . . . . . . . . 82
A.7.2. HAW SPI Control Register (HAWSPICR) . . . . . . . . . . . . . . . . . 84
A.7.3. HAW SPI Slave Select Register . . . . . . . . . . . . . . . . . . . . . 85
A.7.4. SPI Data Receive Register (SPIDRR 1-4) . . . . . . . . . . . . . . . . 85
A.7.5. Software Reset Register (SRR) . . . . . . . . . . . . . . . . . . . . . 86
A.8. HAW SPI IP Core Interrupt Register beschreibung . . . . . . . . . . . . . . . . 86
A.8.1. Device Global Interrupt Enable Register (DGIER) . . . . . . . . . . . . 86
A.8.2. IP Interrupt Status Register (IPISR) . . . . . . . . . . . . . . . . . . . 87
A.8.3. IP Interrupt Enable Register (IPIER) . . . . . . . . . . . . . . . . . . . 87
A.9. HAW SPI Master Konfiguration . . . . . . . . . . . . . . . . . . . . . . . . . . 88
A.10.HAW SPI IP Core Transfer Format . . . . . . . . . . . . . . . . . . . . . . . . 89
A.10.1.CPHA = 0 Transferformat . . . . . . . . . . . . . . . . . . . . . . . . . 89
A.10.2.CPHA = 1 Transferformat . . . . . . . . . . . . . . . . . . . . . . . . . 90
A.11.HAW SPI IP Core Slave Select Modes . . . . . . . . . . . . . . . . . . . . . . 91
A.11.1.automatische Konfiguartion . . . . . . . . . . . . . . . . . . . . . . . . 91
A.11.2.manuelle Konfiguartion . . . . . . . . . . . . . . . . . . . . . . . . . . 91
A.12.HAW SPI Register Flow Beschreibung . . . . . . . . . . . . . . . . . . . . . . 92
A.12.1.HAW SPI Master mit manueller Slave Select Konfiguration(Interrupt) . . . 92
A.12.2.HAW SPI Master mit manueller Slave Select Konfiguration(polling) . . . . 93

A.13. Ressourcenverbrauch . . . . . . . . . . 93
A.14. Low- und High Level Treiber . . . . . . . . . . 94

**B. C-CODE Übersicht** **103**
B.1. C-Code zur Untersuchung der maximalen Abtastfrequenz durch SPI Transfers und Signalrekonstruktion . . . . . . . . . . 103
B.2. C-Code FreeRTOS Demoanwendung . . . . . . . . . . 105
B.2.1. C-Code FreeRTOS Demoanwendung Trace Funktion . . . . . . . . . . 109

**C. VHDL CODE DPIM** **110**

# 1. Einleitung

Die FAUST (Fahrerassistenz- und Autonome Systeme) Projekte der Hochschule für angewandte Wissenschaften Hamburg bieten verschiedene Fahrzeugplattformen für Forschung und Entwicklung von Technologien für Fahrerassistenz- und autonome Systeme. Der Carolo Cup ist ein Wettbewerb für Hochschulen, die ihre Modellfahrzeuge aus Eigenentwicklung miteinander messen. Die mit Fahrerassistenzsystemen bestückten Modellfahrzeugemüssen dabei autonome Disziplinen wie Bahnführung, Ausweichmanöver bei Hindernissen und Parkfunktion bewältigen.

Solche Systeme erfordern eine hohe Rechengeschwindigkeit der Komponenten für die digitale Echtzeitsignalverarbeitung. Die bisherigen Ansätze der Fahrzeugsteuerung des CC-FAUST-Projekts basieren auf Mikrocontroller-Lösungen. FPGA basierte System on Chip bieten dazu die HW-Ressourcen für Beschleunigermodule und zusätzliche $\mu$Prozessoren für parallele SW-Lösungen. Durch die Integration vieler Systemfunktionalitäten auf einem Chip reduziert sich die Anzahl der ICs, was zur Kosteneinsparung und Miniaturisierung führt. Der Entwurf einer FPGA basierten SoC Plattform beruht auf vorhandenen und für die jeweilige Anwendung spezifischen IP(Intellectual Property) COREs, die zu einem Mikrocontrollersystem zusammengesetzt werden.

- Hard IP CORE : besitzt ein fest vorgegebenes Layout und Timing mit fixen Interfaces. Werden auf Siliziumfläche und Geschwindigkeit optimiert. Einige Virtex FPGAs besitzen IBM Power PC HARD IP Prozessoren.

- Soft IP CORE : beschreibt die digitale Schaltung der einzelnen Systemkomponenten wie z.B. MicroBlaze Prozessor und Standard Peripherals in Form einer Hardwarebeschreibungssprache wie VHDL und Verilog auf einer hohen Abstraktionsebene oder als Netzliste.

Der Einsatz von IP-COREs ermöglicht die optimale Anpassung des Systems an die Anforderungen der jeweiligen Anwendung. Veränderungen im Design können aufgrund der Rekonfigurierbarkeit des FPGA durgeführt werden.

Im Rahmen dieser Bachelorarbeit wird eine FPGA basierte SoC(System on Chip) Hardwareplattform vorgestellt und anhand von Fahrerassistenzsystemen erprobt (vgl. Abbildung 1.1). Die $\mu$Controller basierenden Lösungen des CC-FAUST Projekts werden durch eine FPGA basierte SoC Hardwareplattform für die Zielanwendungen *Einparkfunktion* und *Spurführung* ersetzt.

Die Ziele dieser Bachelorarbeit für den Aufbau der System on Chip basierten Fahrzeugsteuerungsplattform des SoC-CC-Fahrzeugs sind :

1. Das Übertragen der Konzepte für die autonome Einparkfunktion aus dem SCV-FAUST-Projekt.
2. Der Aufbau einer Sensorplattform für die seitliche Abstandsmessung mit dem Schwerpunkt auf die Erstellung des Kommunikationsmoduls *HAW SPI IP CORE*. Der Entwurf eines Interfacekonzepts für die Sensordatenerfassung auf einem MicroBlaze Prozessorsystem.
3. Ein Interface vom FPGA zu einem Host-Rechner für Datenlogger Funktionalitäten mit dem Schwerpunkt auf die Erstellung des *NET 1 IP CORE*.
4. Die Portierung eines Echtzeitbetriebssystems mit Einbettung der Software Module in ein Task-System.

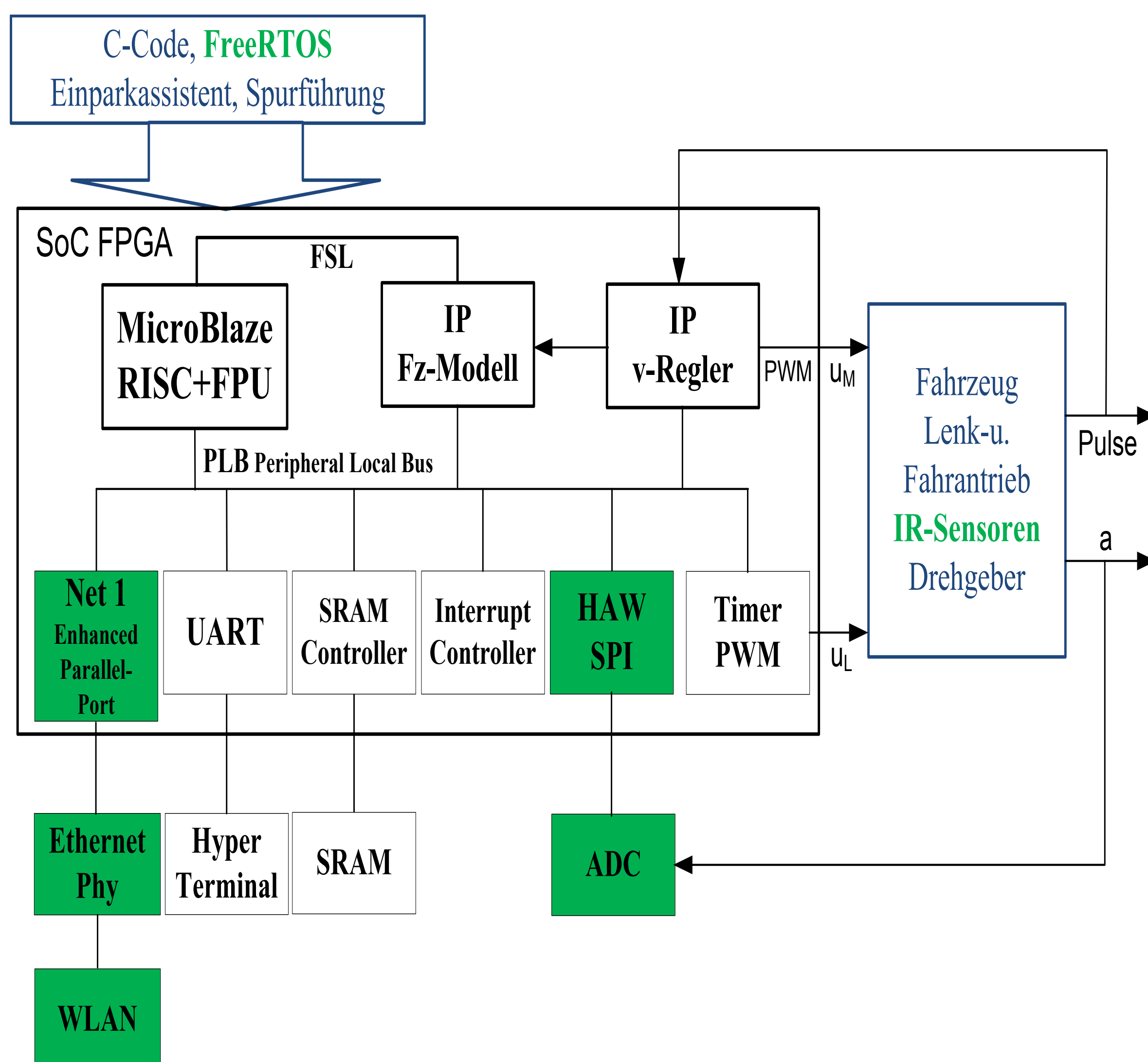



Abbildung 1.1.: SoC Übersicht des SoC-CC-Fahrzeugs

- Diese Arbeit beginnt mit der Beschreibung des kinematischen Einspurmodells und das Übertragen der Konzepte für die autonome Einparkfunktion aus dem SCV-FAUST-Projekt.

- In Kapitel 3 werden die EDK Tools für den Entwurf und eine Übersicht über due Hardware Komponenten des SoC vorgestellt. Weiterhin wird die Erstellung von IP COREs mit dem EDK beschrieben.

- In Kapitel 4 wird der Hardwareaufbau der Sensorik für die seitliche Abstandsmessung und der Entwurf des Kommunikationsmoduls *HAW SPI IP CORE* beschrieben.

- In Kapitel 5 wird der Aufbau des Datenloggers und die JTAG Programmierung des FPGA über WLAN (vom Host-PC), das über drei Bussysteme erfolgt beschrieben. Das *NET 1 IP CORE* stellt die Verbindung zum Microblaze über den PLB und als externe Schnittstelle über den Enhanced Parallel Port(EPP) zum Ethernet Controller her.

- In Kapitel 6 wird die Portierung des Echtzeitbetriebssystems FreeRTOS mit der Erstellung einer Demoanwendung und Messungen des Taskverlaufs beschrieben.

# 2. Autonome Einparkfunktion des SoC-CC-Fahrzeugs

Dieses Kapitel gibt eine Übersicht zur Abstandsregelung des Einparkassistenten, die weiter vertieft werden kann, wenn eine vollständige Sensorplattform vorliegt. Das Konzept der Einparkfunktion wird aus dem FAUST-SCV-Projekt übertragen [Liu (2008)]. Das kinematische Fahrzeugmodell des SoC-CC-Fahrzeugs wird vorgestellt, das ein Einspurmodell ist. Die autonome Parkfunktion läuft in zwei Phasen ab. In der ersten Phase wird eine passende Parklücke mit mindestens doppelter Fahrzeuglänge gesucht. In der zweiten Phase erfolgt eine Trajektorien-Berechnung mittels Odometrie und infrarotgestützter Abstandsregelung für die Lenkwinkelsollvorgabe.

## 2.1. Kinematisches Einspurfahrzeugmodell des SoC-CC-Fahrzeugs

Das stationäre Verhalten des Einspurmodells wird durch den Lenkwinkel unter der Annahme einer konstanten Geschwindigkeit charakterisiert. Für die Herleitung der Systemgleichung wird die Fahrt des SoC-CC-Fahrzeugs auf einer Kreisbahn mit dem eingeschlagenen Lenkwinkel $\alpha$ und der konstanten Geschwindigkeit $v_m$ des Vorderrades **M** betrachtet (vgl. Abbildung 2.1). Die Kreisbahnradien $r_m$ und $r_p$ stehen orthogonal zum Vorder- bzw. Hinterrad und schneiden sich im Bezugspunkt **Z**. Der Achswinkel $\theta$ beschreibt den Winkel zwischen der Fahrzeuglängs- und x-Achse.

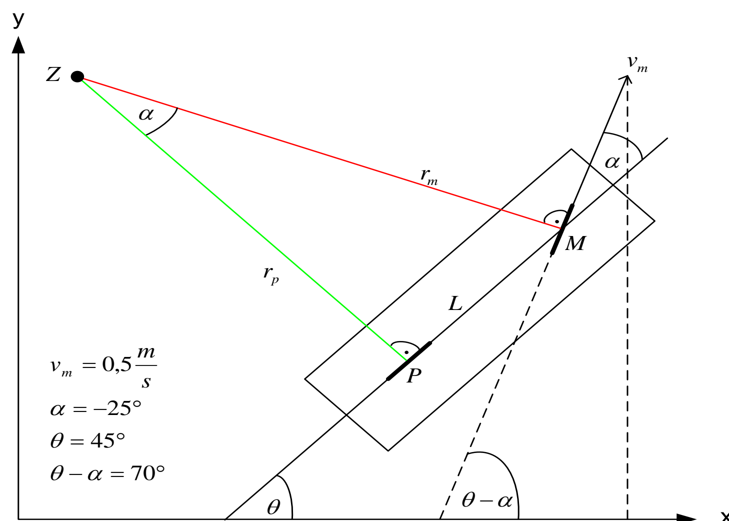



Abbildung 2.1.: Kinematisches Einspurfahrzeugmodell

Für die Fahrzeugkinemtik wurden folgende Systemgleichung aufgestellt:

**Achswinkel**

$$\theta(t) = \frac{v_m \cdot 360}{L \cdot \pi} \int_0^T sin(\alpha)dt \tag{2.1}$$

**Vorderradposition**

$$x_m(t) = v_m \int_0^T cos(\theta - \alpha)dt \tag{2.2}$$

$$y_m(t) = v_m \int_0^T sin(\theta - \alpha)dt \tag{2.3}$$

**Hinterradgeschwindigkeit**

$$v_p(t) = v_m \cdot cos(\alpha) \tag{2.4}$$

**Hinterradposition**

$$x_p(t) = \int_0^T v_p \cdot cos(\theta)dt \tag{2.5}$$

$$y_p(t) = \int_0^T v_p \cdot sin(\theta)dt \tag{2.6}$$

Mit den Eingangsgrössen Ist-Lenkwinkel $\alpha$ und Geschwindigkeit des Vorderrads $v_m$ und den obigen Systemgleichungen der Fahrzeugkinemtik berechnet die Odometrie die neue Position des SoC-CC-Fahrzeugs. Dadurch können Rückschlüsse über die Bewegung des SoC-CC-Fahrzeugs in der Ebene gemacht werden.

## 2.2. Parklückensuche und Messung des zurückgelegten Weges mit den seitlichen Infrarotsensoren

Die erste Phase des autonomen Parkvorgangs beginnt mit der Suche nach einer passenden Lücke mit einer Tiefe von mindestens 25cm und einer Mindestlänge von 80cm. Dabei fährt das SoC-CC-Fahrzeug mit einer konstanten Geschwindigkeit und erfasst zyklisch über Infrarotsensoren die seitlichen Abstände (vgl. Abbildung 2.2). Sobald die erforderte Tiefe erfasst worden ist, beginnt die Berechnung des gefahrenen Weges. Ist die Lücke zu klein, so wird die Berechnung verworfen und es beginnt eine neue Suche.

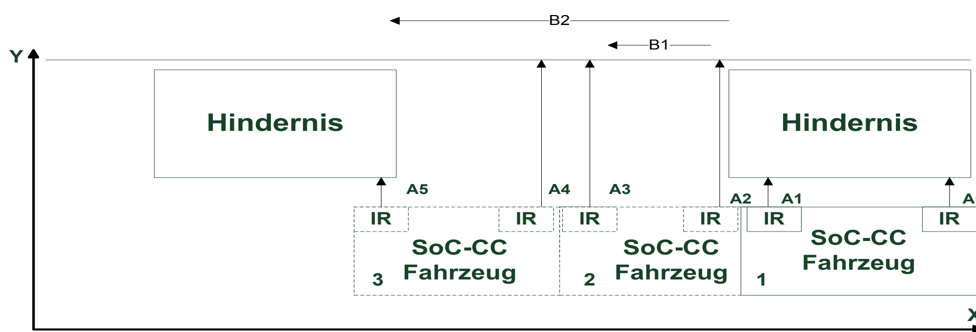



Abbildung 2.2.: Szenario der Parklückensuche mit den Infrarotsensoren für die seitliche Tiefenmessung und Berechnung des gefahrenen Weges

1. Die erfassten seitlichen Abstände sind zu klein und der Verfahrweg wird nicht berechnet.
2. Die Tiefe der Parklücke ist ausreichend und der Verfahrweg B1 wird berechnet.
3. Der vordere Infrarotsensor erfasst wieder einen kleinen Abstand A5, der nicht ausreicht. Der Verfahrweg B2 reicht für das Starten eines Parkvorganges aus.

## 2.3. Abstandsregelung mit einem digitalen PD-Regler für die Vorgabe des Lenkwinkels

Ist die Parklückensuche abgeschlossen, werden aus Tiefe und Länge der Parklücke die Grenzen für einen virtuellen Raum in der Ebene aufgestellt. An der Position, an dem der Parkvorgang beginnt, wird der Nullpunkt gesetzt. Das Fahrzeug fährt mit einer longitudinalen Bewegung Rückwärts in die Parklücke bis Position 1 und richtet sich mittig zu Position 2 aus (vgl. Abbildung 2.3). Der digitale PD-Regler errechnet den zu setzenden Lenkwinkel $\alpha$, der benötigt wird, um das Fahrzeug entlang der Trajektorie zu führen. Die Y-Koordinate des Hinterrads wird durch den Abstandswert des Infrarotsensors ersetzt. Anhand der Auswertung vom berechneten Abstandsdreieck wird entschieden, ob der Sensor beim Einparkvorgang die seitliche Begrenzung oder ein stehendes Hindernis erfasst hat (vgl. Kapitel 2.3.2). Wird ein stehendes Hindernis erfasst, können die Abstandswerte des Infrarotsensors nicht dem PD-Regler zugeführt werden. Die aus der Odometrie berechnete Y-Koordinate des Hinterrads wird dem PD-Regler zugeführt.

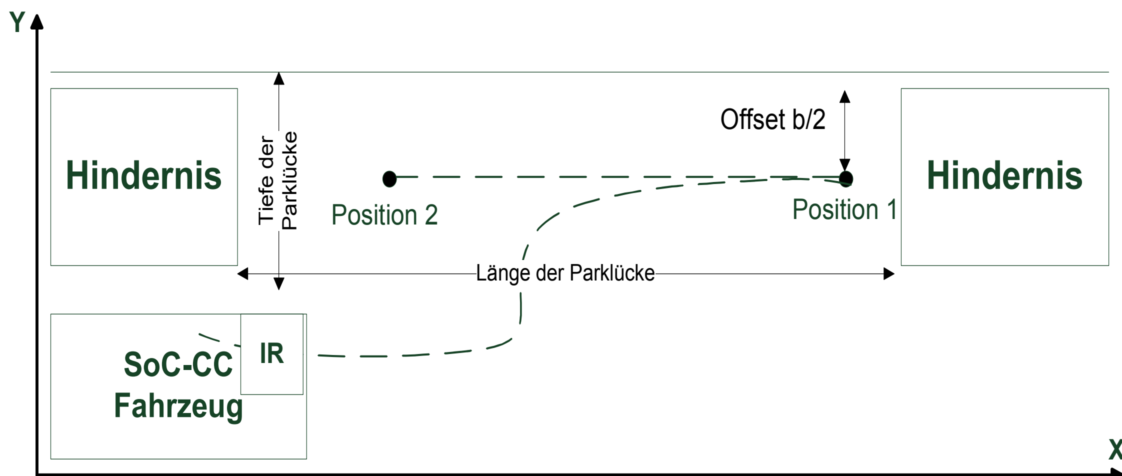



Abbildung 2.3.: Autonomer Parkvorgang des SoC-CC-Fahrzeugs

### 2.3.1. Abstandsregelung mit einem digitalen PD-Regler

Der Regler wurde für einen Ausweich-Assisten entworfen und muss für die Einparkfunktion parametrisiert werden [Schetler (2007)]. Die Reglerparameter $z_{a0}$, $z_{a1}$, $z_{b0}$,$z_{b1}$ konnten aufgrund der noch nicht fertiggestellten Fahrzeuggeometrie berechnet werden. Der digitale PD-Regler nähert die Y-Koordinate des Hinterrads der Y-Koordinate von Position 1 an. Aus der Differenz zwischen der Führungsgröße $Y_h$ und dem Abstandswert $Y_m$ ergibt sich die Regeldifferenz $e$ (vgl. Abbildung 2.4). Mit dem Offset $\frac{b}{2}$ wird die Kollision der hinteren Fahrzeugkante mit der seitlichen Parklückenbegrenzung verhindert.

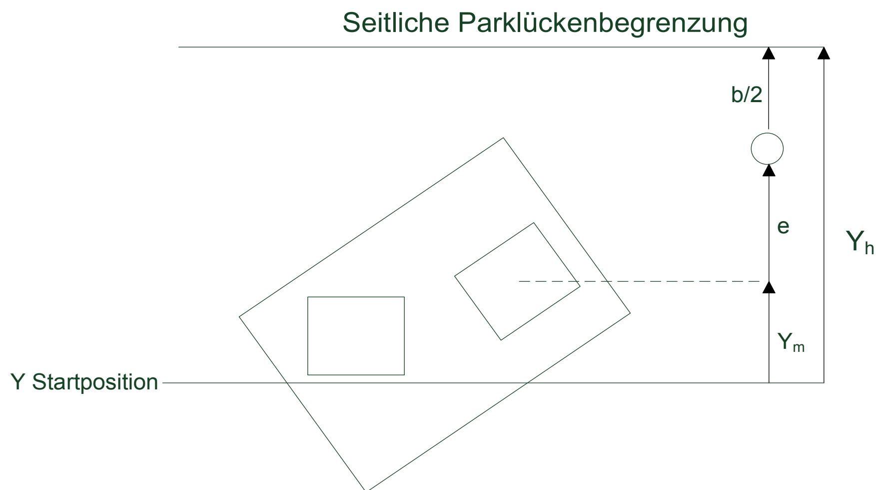



Abbildung 2.4.: Abstandsregelung der Y-Koordinate des Hinterrads $Y_m$ mit einem digitalen PD-Regler

$$e = Y_h - \frac{b}{2} - y_m \tag{2.7}$$

Aus der Regeldifferenz $e$ wird jene Stellgröße $\alpha$ berechnet, die die Regelabweichung minimiert (vgl. Kapitel 2.5).

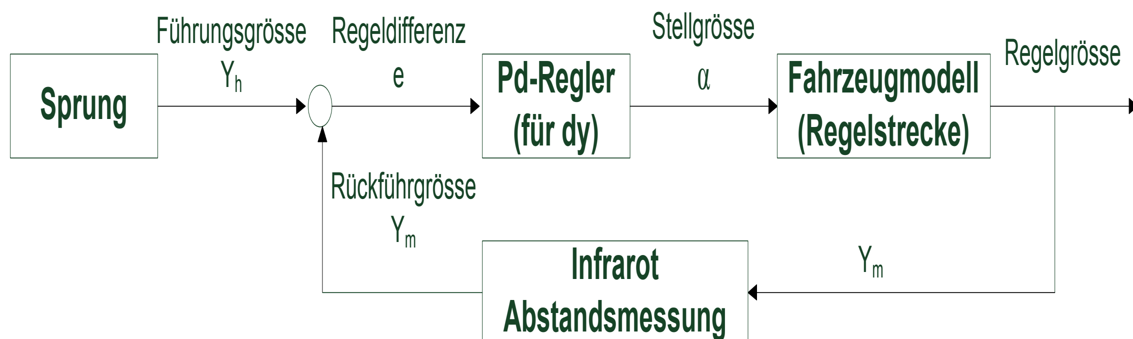



Abbildung 2.5.: Regelkreis mit Infrarot gestützter Abstandserfasssung

Für die zyklische Berechnung des Lenkwinkels fließen der Lenkwinkel und die Regeldifferenz der vorangegangenen Iteration in die Gleichung mit ein (vgl. Gleichung 2.8). Der Abstandswert $Y_m$ wird zwischen dem Infrarot-Sensor Abstandswert und den aus der Odometrie berechnete Y-Koordinate unterschieden.

$$\alpha_{Stellgre}(t) = (e(t) \cdot z_{b0} + e(t-1)z_{b1} - \alpha(t-1) * z_{a1}) \cdot z_{a0} \tag{2.8}$$

Der Lenkwinkel und die konstante Geschwindigkeit des Vorderrads werden der Odometrie für die zyklische Positionsberechnung zugeführt.

### 2.3.2. Berechnung eines Abstandsdreiecks zur Auswahl zwischen virtueller und infrarotgestützter Positionsermittlung

Beim Start des Einparkvorgangs dreht sich das SoC-CC-Fahrzeug um die eigene Achse, sodass die Abstände zum vorderen Hindernis erfasst werden (vgl. Abbildung 2.6). Aus dem Ver-

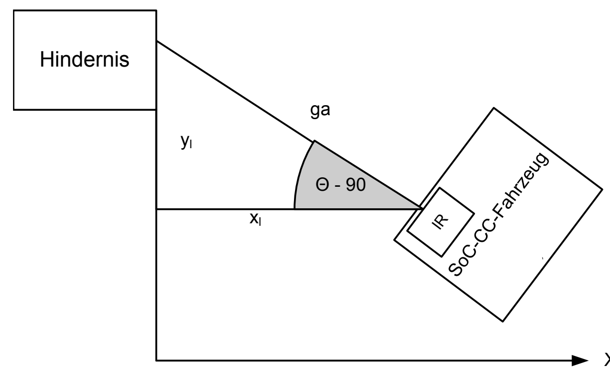


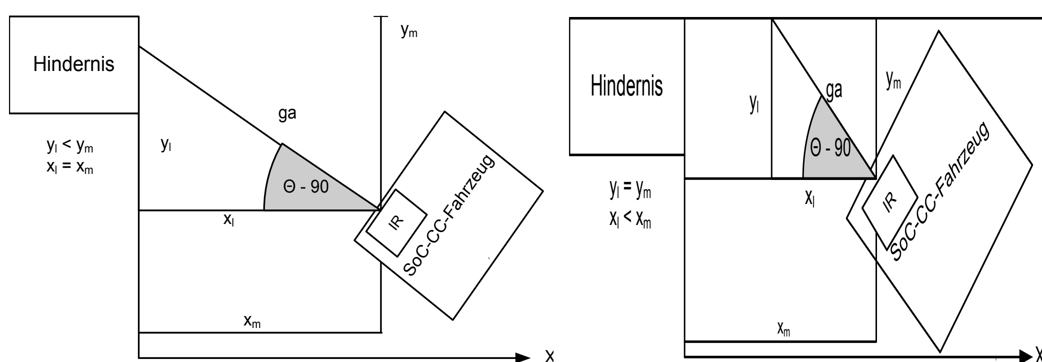

Abbildung 2.6.: Abstandserfassung ga mit dem Infrarotsensor des vorderen Hindernis

gleich zwischen den berechneten Katheten des Abstandsdreiecks und der Odometriedaten in Längstrichtung wird die Reglerzuführung bestimmt (Y-Koordinate des Hinterrads aus der Odometrie bzw. Infrarot-Abstandswert). Die Hypothenuse des Abstandsdreiecks ist der erfasste Infrarot-Sensor Wert. Die Katheten $x_l$ und $y_l$ beschreiben die Längstrichtung und Tiefe.

$$x_l = ga \cdot cos(90^\circ - \theta) \tag{2.9}$$

$$y_l = ga \cdot sin(90^\circ - \theta) \tag{2.10}$$

Die aus dem Abstandsdreieck errechneten Katheten $x_l$,$y_l$ werden mit den Koordinaten $x_m$,$y_m$ aus dem kinemtischen Fahrzeugmodell verglichen.




(a) Wird das Hindernis vertikal erfasst, ist die Kathete der Längstrichtung $x_l$ gleich $x_m$ und die Kathete der Parklückentiefe $y_l < y_m$.

(b) Wird die seitliche Abgrenzung erfasst, ist die Kathete $x_l < x_m$ und $y_l = y_m$

Abbildung 2.7.: Auswahl zwischen virtueller und infrarotgestützter Positionsermittlung

# 3. System on Chip Hardwareplattform des SoC-CC-Fahrzeugs

Die System on Chip Hardwareplattform des SoC-CC-Fahrzeugs wurde mit dem Embedded Development Kit (EDK) von Xilinx entworfen. Das EDK enthält die Entwicklungsumgebungen Platform Studio und Platform Studio SDK. Das Platform Studio enthält eine Reihe von Tools, Dokumentationen und IP COREs für die Integration von Hard- und Software-Komponenten eines Embedded Systems mit Hard- und/oder Soft-CORE Prozessoren. Das Platform Studio SDK basiert auf einer Eclipse IDE. Die Zielhardware des SoC ist ein Nexys 2 Board von Digilent. Das Support Package für das Nexys 2 Board beinhaltet die Board-Information und IP-COREs für die Erstellung der Hardwareplattform mit dem EDK.

## 3.1. Systemübersicht

Die im Rahmen dieser Arbeit entwickelten IP COREs *HAW SPI* und *NET 1* und die Portierung des Echtzeitbetriebssystems FreeRTOS wurden auf einem Nexys 2 Board evaluiert. Dazu wurde mit dem EDK und dem zum Nexys 2 Board zugehörigen Support Package eine SoC-Hardware-Plattform mit dem MicroBlaze Soft-Core Prozessor erstellt (vgl. Abbildung 3.1).

### 3.1.1. Nexys 2 Board

Die kompakte Bauweise des Nexys 2 Boards ermöglicht dessen Einsatz auf dem SoC-CC-Fahrzeug. Die Pin Interfaces PMOD und FX2 des Boards sind abgestimmt auf die Digilent Peripherie-Module.

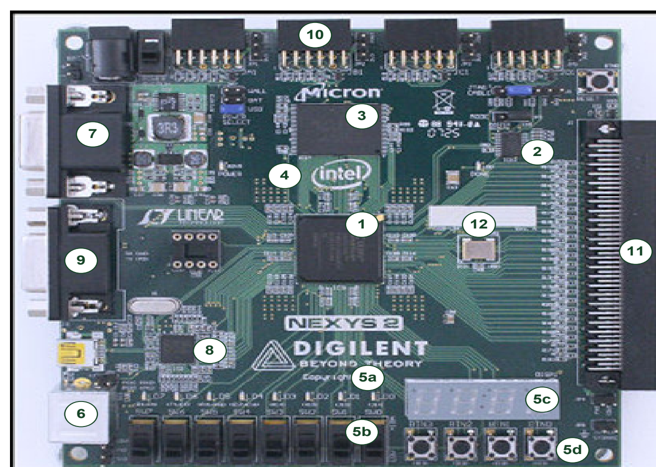



Abbildung 3.1.: Nexys 2 Evaluierungsboard

1. **Xilinx Spartan 3E FPGA :** Das XC3S1200E FPGA Device ist das zweitgrößte FPGA der Spartan 3E Serie mit folgenden Eigenschaften:
   - 304 I/O Pins
   - 64-KB BLOCK RAM
   - 19512 Logic Cells(1 Logic Cell = 1 * Flip Flop + 1 * 4LUT)
   - 2168 configurable Logic Blocks (CLB) $\approx$ 8672 Slices
   - 8 * Digital Clock Manager
   - 28 * Hardware Multiplizierer für zwei 18 Bit Operanden
2. **Platform Flash ROM Konfigurationsspeicher :** Der Platform Flash Xilinx XCF02 speichert die Bitstream-Konfigurationsdaten für den FPGA.
3. **16 MB Micron SRAM :** Der 128MBit Micron M45W8MW16 SRAM ist als 8MB * 16Bit aufgebaut. Im asynchronen Modus benötigen Lese- und Schreibzyklen 70ns. Im synchronen Modus werden Transfers mit bis zu 80MHZ getrieben. Das Micron SRAM teilt sich den 16-Bit Datenbus und 24-Bit Adressbus mit dem Intel StrataFlash.
4. **16 MB Intel StrataFlash :** Der Intel TE28F128J3D75-110 StrataFlash ist als 8MB * 16Bit in 128 Blöcken aufgebaut. Die Blöcke können separat gelöscht werden. Die Programmierung des FLASH Speichers erfolgt über einen internen 32-Byte Cache. Schreibzyklen in das Cache erfolgen alle 70ns und die Übernahme in das FLASH-Array beträgt 218$\mu$s. Ein Lesezyklus für 16-Bit Daten aus einem Block benötigt 110ns im normal mode. Im 4 bzw. 8 page mode werden 8 bzw. 16 Bytes Daten aus demselben Block ausgelesen. Die Adressierung erfolgt einmalig für einen Lesezyklus. Jeder einzelne Lesevorgang benötigt 25ns.
5. **User I/O :**
   a) 8 LEDs
   b) 8 Switches
   c) 7-Segment Anzeige
   d) 4 pushbuttons
6. **PS/2 Port für Maus und Tastatur**
7. **VGA Port**
8. **Cypress CY7C68013A USB 2.0 Slave Controller**
9. **RS-232 Kommunikations Controller**
10. **Peripheral Connectors(PMOD) :** Das Board verfügt über 8 Pmod Buchsen mit je 4 Signal- und Spannungsversorgungspins. Die ADC-Module sind über die PMODs verbunden.
11. **FX 2 Hirose 100 Pin Expansion connector :**
    - 40 I/O Pins zum FPGA
    - 47 Ground Pins

- 5 VCC Pins
- 5 JTAG Pins
- 3 Clock I/O Pins

Das Net1 Modul mit dem Parallelport und der JTAG-chain Schnittstelle werden über die FX2 Hirose Schnittstelle mit dem FPGA verbunden.

12. **50 MHZ Oscillator**

### 3.1.2. MicroBlaze Softcore Prozessor

Der MicroBlaze ist ein Softcore Prozessor der als IP im EDK für alle Xilinx FPGA Familien zur Verfügung gestellt wird (vgl. Abbildung 3.2).

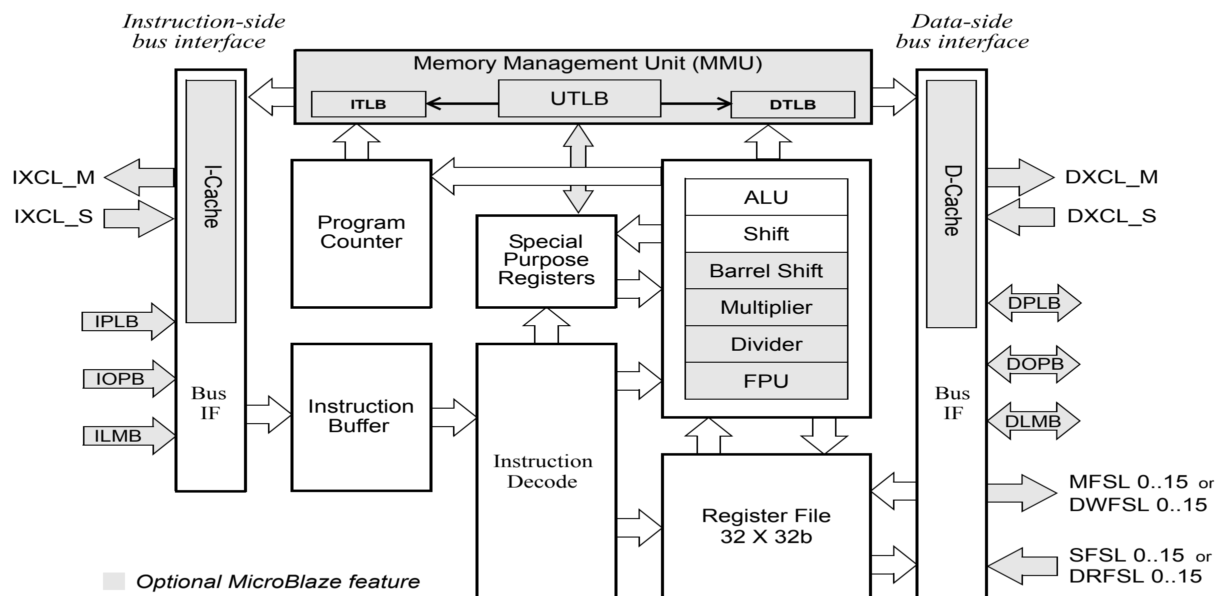



Abbildung 3.2.: Blockschaltbild des Microblaze Prozessors [Xilinx (2004)]

Eigenschaften des MicroBlaze :

- 32 Bit Reduced Instruction Set Computing(RISC) Prozessor
- Harvard Architektur für separaten Zugriff auf Daten und Instruktionen
- Fünfstufige Pipeline als Standard-Konfiguration (Fetch, Decode, Execute, Memory Access, Write Back)
- Dreistufige Pipeline zur Reduzierung der Hardware-Ressourcen (Fetch, Decode, Execute)
- Datenrepräsentation erfolgt nach Big-Endian Format für die Datentypen Wort, Halbwort und Byte
- 32 * 32 Bit General Purpose Register
- 32 Bit Instruktionssatz
- Memory Mapped I/O

- Barrel Shifter, Multiplizier, Dividierer und FPU
- Virtual Memory Management

#### Floating Point Unit

Der MicroBlaze Prozessor verfügt über eine Hardware Floating Point Unit mit einfacher Genauigkeit, die auf den *IEEE 754* Standard basiert [IEEE (1991)].

Das *IEEE 754* Standard für eine FPU mit einfacher Genauigkeit wird im folgenden Format dargestellt.

1. 1 Bit für das Vorzeichen
2. 8 Bit für den Exponent
3. 23 Bit für die Mantisse

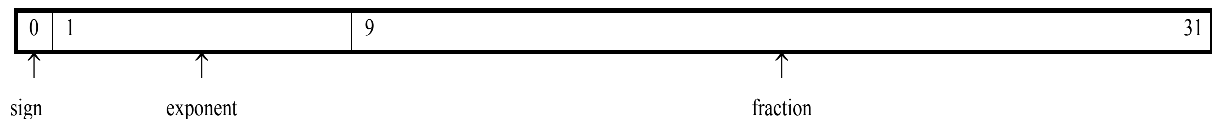



Abbildung 3.3.: IEEE 754 Single Precision Format [IEEE (1991)]

| Exponent und Mantisse | Wert |
|---|---|
| Exponent = 255 und Mantisse <> 0 | NaN, unabhängig vom Sign Bit |
| Exponent = 255 und Mantisse = 0 | $(-1)^{sign} * \infty$ |
| 0 < Exponent < 255 | $(-1)^{sign} * 2^{(exponent-127)} * (1.Mantisse)$ |
| Exponent = 0 und Mantisse <> 0 | $(-1)^{sign} * 2^{-126} * (0.Mantisse)$ |
| Exponent = 0 und Mantisse = 0 | $(-1)^{sign} * 0$ |

- Die FPU des MicroBlaze Prozessors verfügt über folgende Arithmetische Operation :
  1. Addition, fadd –> 4 Takte
  2. Subtraktion, fsub –> 4 Takte
  3. Multiplikation, fmul –> 4 takte
  4. Division, fdiv –> 28 Takte
  5. Quadratwurzel, fsqrt –> 27 - 29 Takte
- Die folgenden Vergleichsoperatoren erfordern nur einen Takt :
  1. less-than, fcmp.lt
  2. less-than, fcmp.lt
  3. equal, fcmp.eq
  4. less-or-equal, fcmp.le

5. greater-than, fcmp.gt
6. not-equal, fcmp.ne
7. greater-or-equal, fcmp.ge
8. unordered, fcmp.un (für NaN)

- Die Konvertierung vom Integer Format <–> Fließkommaeinheit
  1. Vom signed Integer zu Floating Point, flt –> 4 - 6 Takte
  2. Vom Floating Point zu signed Integer, fint –> 5 - 7 Takte
- Folgende Exceptios werden von der FPU unterstützt :
  1. underflow
  2. overflow
  3. Division durch 0
  4. ungültige Operationen wie z.B. arithmetische Operation mit NaN oder Infinity

Die FPU des MicroBlaze kann als Alternative z.B. zu speziellen CoProzessoren im Q-Format für Fließkommaberechnungen verwendet werden. Der GCC[GNU] kompiliert für Float-Operation die erforderlichen FPU Instruktionen.

### 3.1.3. Cache und Speicher des MicroBlaze

Aus den Spartan 3E FPGA internen BLOCK RAM Modulen können bis zu 64KB Daten und Instruktions Cache für den MicroBlaze über den Cache Link (XCL) mit einen externen Memory Controller (EMC) verbunden werden. Bei voller Ausnutzung der BLOCK RAM-Module als Cache-Speicher würde kein Speicher mehr für den schnellen Local Memory Bus(LMB) und den Read/Write FIFOs vom NET 1 IP CORE zur Verfügung stehen (vgl. Kapitel 3.1.4, Kapitel 5.3.2). Aufgrund der Programmgröße von FreeRTOS und den Software-Modulen, die über 64KB betragen, ist der interne 64KB BLOCK RAM-Speicher nicht ausreichend. Die Verwendung eines externen Speichers ist erforderlich. Hierfür wird der Micron SRAM verwendet, der über den EMC mit dem MicroBlaze Cachelink für Instruktionen verbunden wird. Für den Instruktions Cache werden 32KB interne BLOCK RAM verwendet. Für Daten werden 16KB interner BLOCK RAM am LMB verbunden. Die restlichen 16KB stehen den Read/Write FIFOs und anderen Komponenten zur Verfügung.

### 3.1.4. Bussysteme des Microblaze Prozessors

#### Local Memory Bus (LMB)

Der LMB ist ein synchroner Single Master Bussystem für den FPGA internen Block RAM Zugriff. Für Anwendungen bis 64KB ist der LMB die schnellste Variante (Lese- / Schreibzyklen benötigen nur einen Systemtakt). Der MicroBlaze hat zwei LMB Interfaces für Daten und Instruktionen (vgl. 3.2), die mit separaten LMB BRAM Interface Controllern verbunden sind. Über die Interface Controller werden Daten und Instruktionen vom Dual Port BRAM Block parallel transferiert (vgl.

3.4). Der Dual Port BRAM Block steuert die Zugriffe auf die internen BLOCK RAM Module des FPGA.

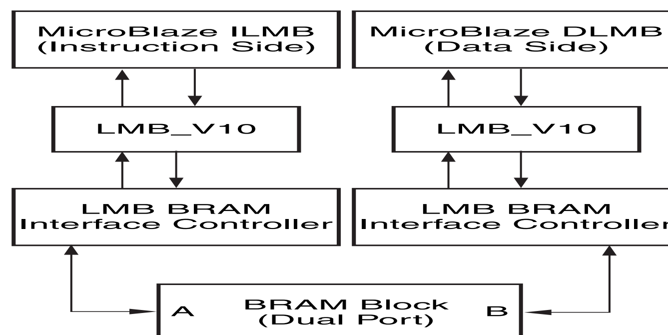



Abbildung 3.4.: Harvardarchitektur des MicroBlaze mit dem Lokal Memory Bus [Xilinx (2009)]

#### Processor Local Bus (PLB)

Der PLB ist ein synchrones mehrfach Master mehrfach Slave Bussystem mit entkoppelten Schreib- und Lesebussen.

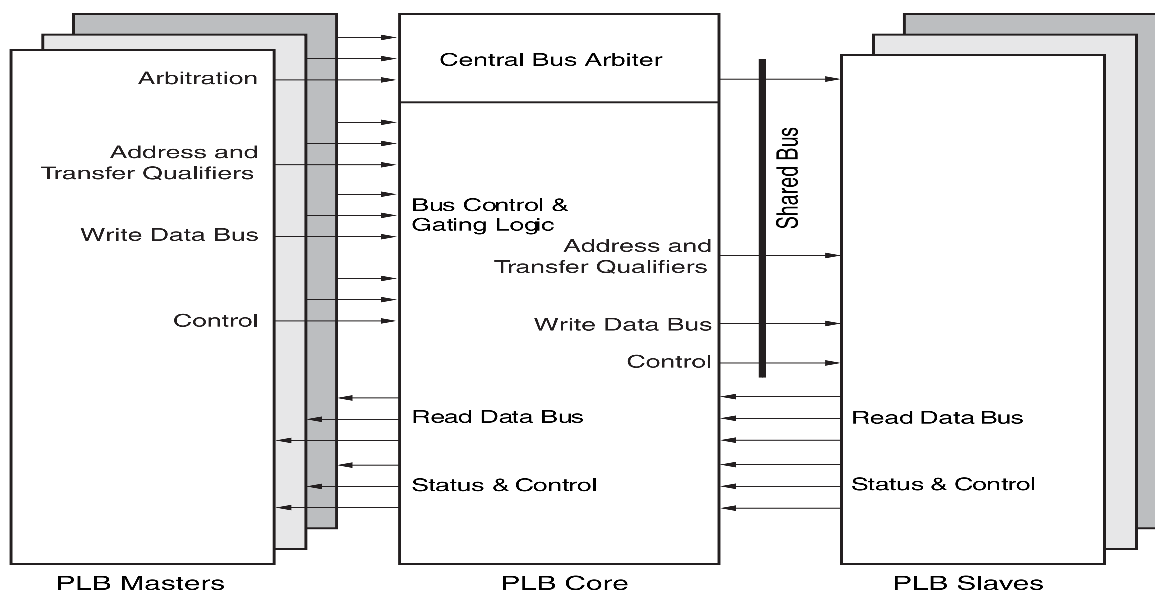



Abbildung 3.5.: Processor Local Bus Topologie [Xilinx (2007)]

Bis zu 16 verschiedene Bus-Master mit getrennten Daten- und Adressbussen können mit beliebig vielen Slaves mit dem gleichen Adress- und Datenbus am PLB Kern verbunden werden (vgl. Abbildung 3.5). Die Entkopplung des Lese- und Schreibpfades und die Verwendung unterschiedlicher Adressbusse ermöglichen ein nebenläufiges Lesen und Schreiben in einem Taktzyklus. Die Slave-Adressierung erfolgt über unterschiedliche Memory Mapped Basisadressen. Der PLB Arbiter koordiniert den Bus über 4 Prioritätsebenen für die Master-Module. Der PLB CORE verbindet den Master Microblaze mit folgender Slave Peripherie:

- HAW SPI IP CORE

- NET1 IP CORE
- Interrupt Controller
- RS232 PORT, für die Standardausgabe
- General Purpose I/O für LEDs, buttons, switches
- 7 Segment IP CORE

#### Fast Simplex Link (FSL)

Der FSL ist eine undirektionale Punkt zu Punkt Verbindung über zwei FIFOs. Der Microblaze verfügt über 8 Master- und 8-Slave FSL Interfaces, die direkt mit den Register File verbunden sind (vgl. Abbildung 3.2). Hardware-Beschleuniger/CoProzessoren greifen über den FSL_0 FIFO direkt auf die Register Files des Microblaze zu, indem Operanden geladen und Berechnungen parallel zur CPU ausgeführt werden. Das Ergebnis wird über das FSL_1 FIFO zurück in das Register File geschrieben (vgl. Abbildung 3.6). Die Zugriffe auf die FIFOs erfolgen mit den Instruktionen *putfsl* und *getfsl*. Im blocking mode wartet der Microblaze, bis gültige Daten auf dem Bus bereitstehen und im non-blocking-mode läuft der MicroBlaze weiter, gültigen Daten werden mit einem Carry-Bit signalisiert.

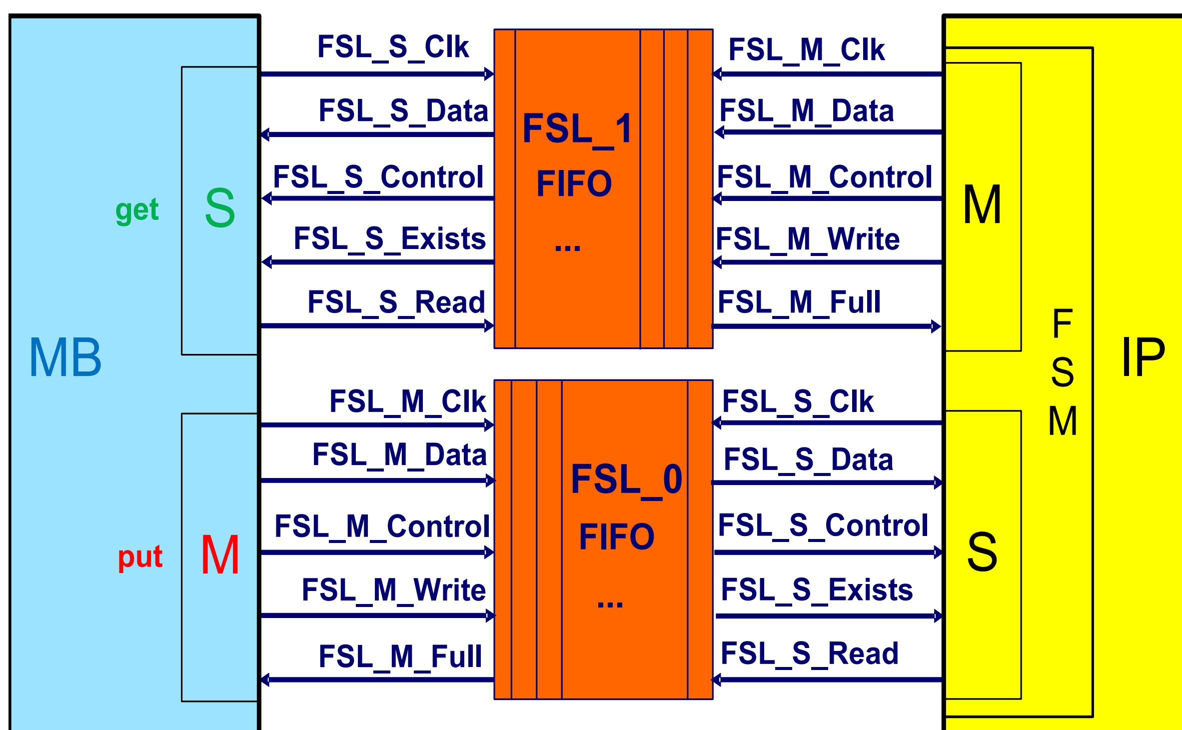



Abbildung 3.6.: Anbindung von Hardware-Beschleuniger/CoProzessoren an die Register Files des MicroBlaze über FSL

## 3.2. EDK Tool Chain

Die SoC-Hardwareplattform wird mit den folgenden EDK Tools erstellt (vgl. Abbildung 3.7).

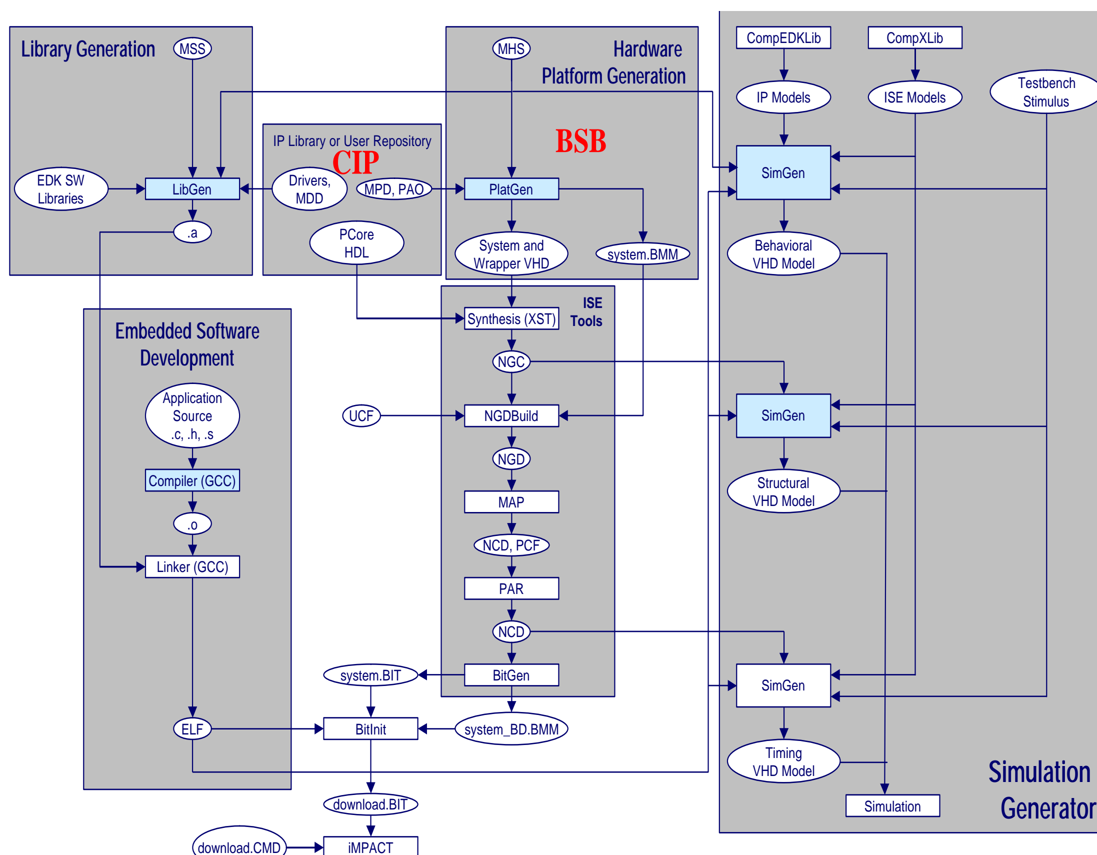



Abbildung 3.7.: Embedded Development Kit Tool Chain

- **Hardware Platform Generation :** Mit dem Base System Builder Wizard (BSB) erfolgt die Auswahl der FPGA-Plattform und es werden die Basis-Hardware-Systemkomponenten wie Prozessor, Bussysteme, Speicher, Cache und Peripherie selektiert. Veränderungen im Design können zu jeder Zeit über die IDE des Platform Studios durchgeführt werden. Aus den selektierten IP COREs werden die Hardware Spezifikationen in die MHS und die zugehörigen Treiber in die MSS Datei geschrieben.

  Das PlatGen Tool generiert die Top Entity des Systems, indem alle Hardware-Komponenten als Wrapper-Module instanziiert werden. Aus der Top Entity wird das FPGA-Konfigurations-Bitstream *system.bit* mittels ISE Tools generiert. BitInit generiert aus dieser Konfigurationsdatei und der elf-Datei das Bitstream *download.bit*, mit dem das FPGA programmiert wird und die BLOCK RAM-Module initialisiert werden.

- **ISE Design Flow :** Von der abstrakten Beschreibung des digitalen Systems bis zur konkreten Schaltung für die gewählte Zielhardware werden sukzessiv die folgenden ISE Tools durchlaufen.

  1. Das Synthesewerkzeug XST erstellt aus den Hardwarebeschreibungssprachen VHDL oder VERILOG für eine FPGA Zielarchitektur zunächst eine Register-Transfer-

Level (RTL) Beschreibung unter Einbeziehung der IP-Bibliothekskomponenten. Aus der RTL-Beschreibung wird eine Netzliste im NGC-Format erstellt, die die Schaltung auf der Ebene der FPGA Zielarchitektur unter Berücksichtigung der Timing Constraints durch CLBs beschreibt.

2. Das Tool NGDBuild liest die Netzliste ein und erzeugt eine Native Generic Datenbank( NGD) mit allen benötigten Komponenten.

3. Aus der NGD Datei werden die Schaltnetzte und Register auf die Logikzellen der Zielarchitektur durch das Tool Map in einer Native Circuit Description (NCD) Datei beschrieben. Es wird nicht festgelegt, welche Logikzellen auf den Chip zu verwenden sind.

4. Place and Route bildet anhand der NCD Datei die abgebildete Schaltung auf die konkreten Bausteine der Zielarchitekur ab. Es wird festgelegt, welche CLBs benutzt werden und wie die Verdrahtung unter Berücksichtung der Timing Constraints zu realisieren ist.

5. BitGen generiert aus der NCD Datei ein Bitstream, das über JTAG oder PROM das FPGA programmiert.

- **Library Generation und Embedded Software Development :** Aus der MSS Datei wird für die verwendeten Hardware-Komponenten durch LibGen eine Library für die Software-Treiber generiert. Mit dem GCC[GNU] wird die Executable and Linking Format (elf) Datei der Anwendung erstellt, das durch BitInit ins Block Ram initialisiert wird.

- **Simulation Generator :** Mit dem SimGen Tool werden Bibliotheken und Testbenches für die Verhaltens-, RTL- und Gatter Timing-Simulation des Hardwaremodells generiert.

## 3.3. IP CORE Erstellung mit dem Peripheral Wizard des EDK

Die Erstellung eines IP COREs mit einem PLB IP Interface zum MicroBlaze Prozessor erfolgt mit dem Peripheral Wizard Tool. Mit dem Tool erfolgen die ersten Entwurfsschritte eines IP-COREs durch Selektierung

1. des PLB Slave Interface,
2. der PLB Slave IPIF Optionen und
3. der Anzahl an Software-Registern für die User Logic.

Das PLB IPIF unterstützt für User-IPs eine konfigurierbare Bus Schnittstelle mit optionalen Features (vgl. Abbildung 3.8).

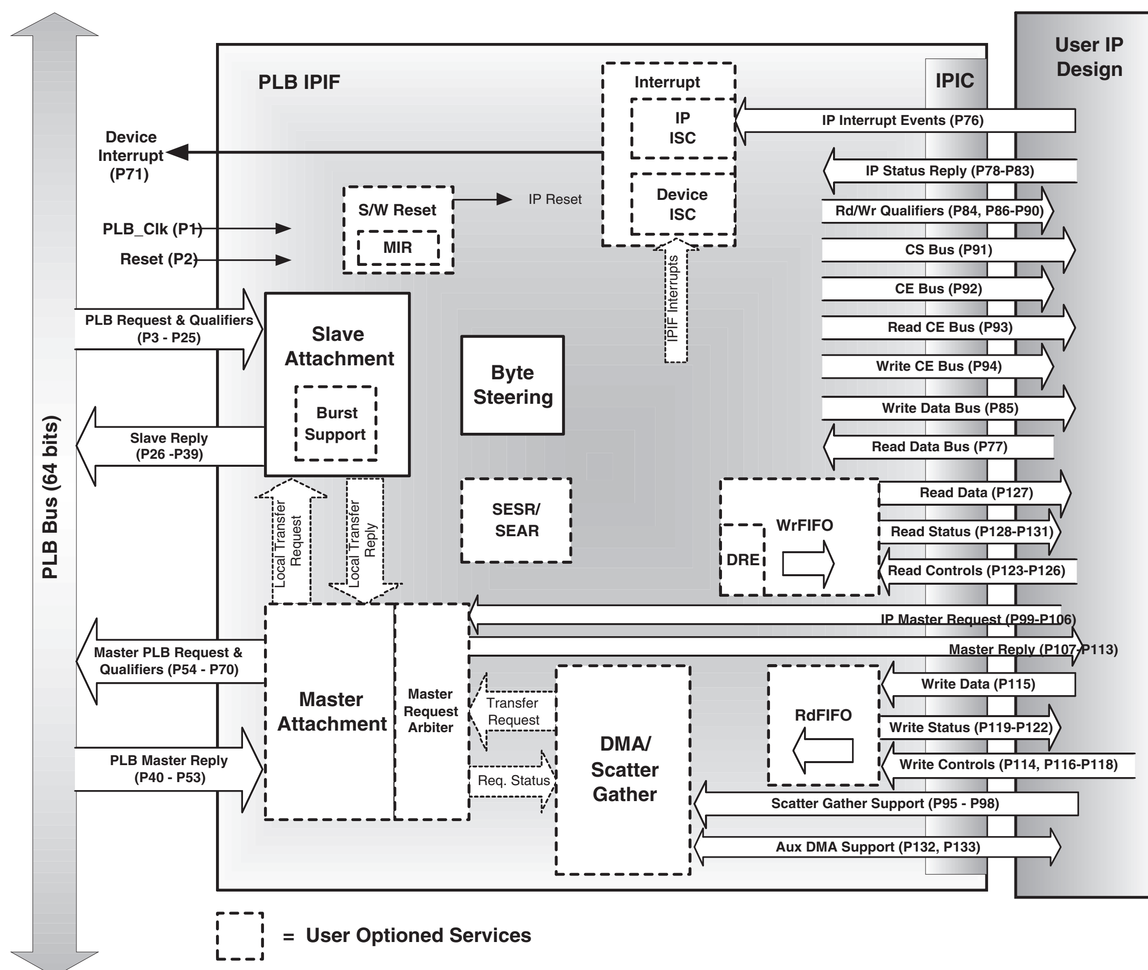



Abbildung 3.8.: Blockschaltbild PLB IPIF [Xilinx (2005)]

- **PLB Slave Attachment (PLBSA)** Das PLBSA ist die Basisfunktionalität des PLB IPIF und dient der Dekodierung des IP Adressraums. Die Lese- / Schreib-Transaktion des PLB wird in ein äquivalentes IP Interconnect (IPIC) Lese- / Schreib-Transaktionformat für die User-IP übersetzt. Mit der Burst-Funktionalität wird eine fixe Anzahl Daten über die Cacheline mit einer höheren Datenrate transferiert.

- **Byte Steering** Das Modul wird bei unterschiedlichen Datenbusbreiten von PLB und User-IP für die Einhaltung der korrekten Bytefolge in Read- / Write-Operationen verwendet.

- **PLB Master Attachment (PLBMA)** Das PLBMA wird nur in Kombination mit dem DMA Controller verwendet. Der DMA Controller muss als Master fungieren und nutzt dieses Master Interface.

- **Direct Memory Access (DMA)** Mit dem DMA wird der Transfer größerer Datenmengen zwischen User IP/IPIF FIFO und anderer PLB Peripherie wie z.B. Speicher oder Bridge automatisiert.

- **Software (S/W) Reset** Mit dem Modul erfolgt ein Reset der Register im IP CORE, ohne andere Register des Systems zurückzusetzen.

- **Interrupt Control (IC)** Der IC dient der Interrupt Funktionalität des IPs. Das Modul hat als Eingang alle vom IP generierten Interrupt-Signale und ein Device Interrupt-Ausgang

zum MicroBlaze/Interrupt Controller. Die Register für Status und Aktivierung der einzelnen Interrrupts und dem Ausgangs-Interrupt werden über den Adressraum des IP CORE ausgelesen/beschrieben.

- **Read/Write FIFO** Mit diesen Modulen werden separate Lese- / Schreib-Puffer zwischen PLB und der User Logic erstellt. Das Lesen/Schreiben der Daten vom Microblaze erfolgt über den Memory Mapped-Adressbereich des IP CORE. Für die User Logic werden PLB FIFO IPIF Signal Interfaces im IPIC-Format für die Steuerung bereitgestellt (vgl. Kapitel5.3.2).

- **SESR/SEAR** Mit dem Modul werden fehlerhafte Bustransaktion in den Slave Error Status Register und Slave Error Address Register geschrieben. Dadurch kann die Anwendung auf fehlerhafte Bustransaktionen reagieren.

Das Tool generiert eine Template-Struktur aus den selektierten Komponenten und zusätzlichen Projekt Dateien für das EDK. Die Template Struktur beinhaltet zwei VHDL-Dateien. Die erste Datei beinhaltet die Top Entity des IP COREs, in der die PLB IPIF-Komponenten und die User Logic instanziiert werden. Die zweite Datei beinhaltet die Entity des Anwendermoduls User Logic mit den Signal Interfaces der Software-Register und IPIF-Optionen im IPIC Format (vgl. Abbildung 3.9).

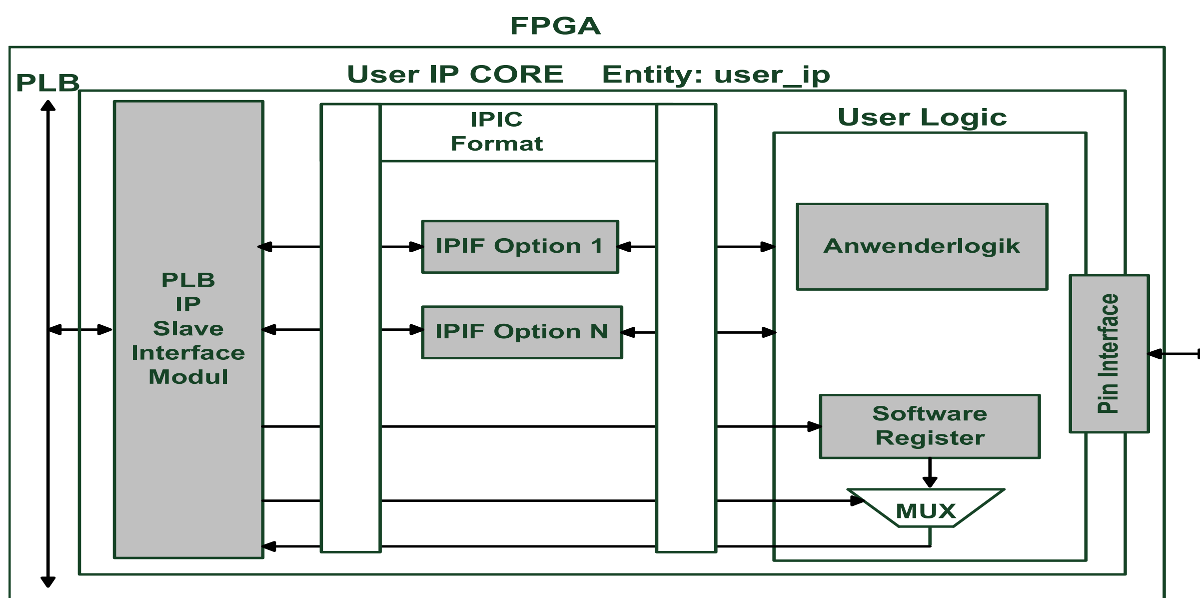



Abbildung 3.9.: Blockschaltbild eines User IP CORE

In der User Logic werden die Software-Register als Status-, Steuer- und Daten-Register für die Anwenderlogik genutzt. Das Auslesen der Register-Inhalte erfolgt kombinatorisch über ein Multiplexer, das Beschreiben sequentiell über D-Flip-Flop Register (vgl. Abbildung 3.10).

Das MicroBlaze PLB Master Modul führt Lese- und Schreib-Transaktion über den Memory Mapped-Adressraum des User IP COREs zum PLB Slave Modul. Das PLB Slave Modul übersetzt die Anfragen in das IPIC Format für die Busanbindung des User IP COREs über die PLB Slave IPIF Signal Interfaces (vgl. Tabelle 3.1).

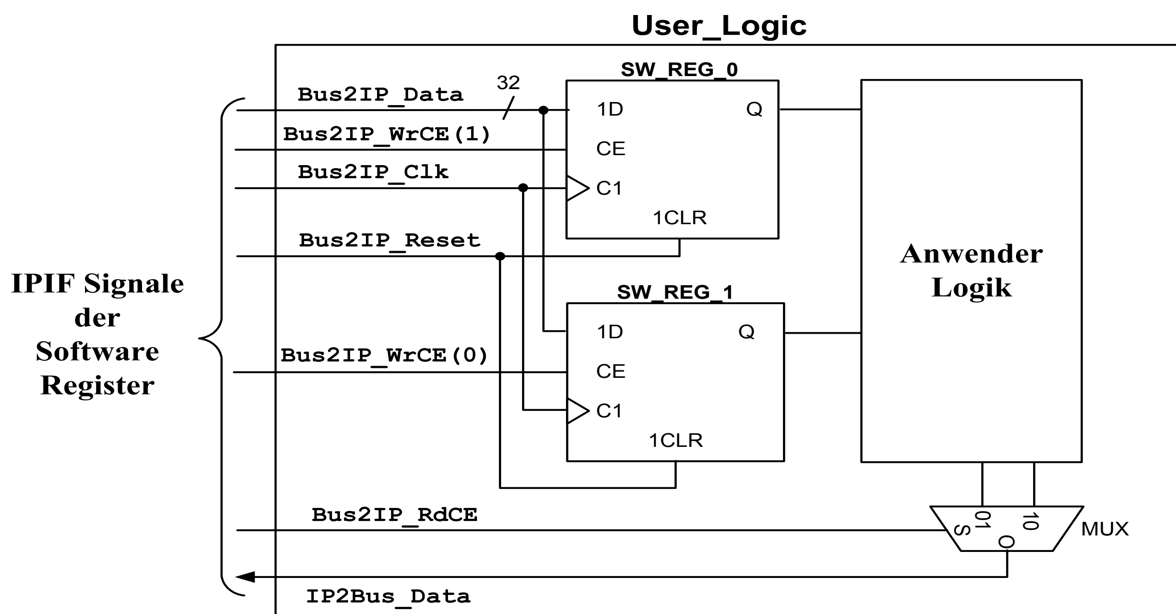



Abbildung 3.10.: Blockschaltbild User Logic mit zwei Software-Registern

| IPIF Signalname | E/A | Beschreibung |
|---|---|---|
| ipif_Bus2IP_Clk | A | PLB Takt für das User IP CORE |
| ipif_Bus2IP_Reset | A | PLB Reset für das User IP CORE |
| ipif_Bus2IP_CS | A | Das CS Signal ist das Multiplexer Steuersignal für das kombinatorische Auslesen der User IP CORE Register aus den einzelnen Adressbereichen. Die Vektorbreite ergibt sich aus der Anzahl der Adressbereiche. |
| ipif_Bus2IP_RdCE | A | Read Chip Enable Signal zum Lesen der User IP CORE-Register (Software, FIFO Daten, Status usw.) Die Vektorbreite hängt von der Gesamtanzahl der adressierbaren Register im User IP CORE |
| ipif_Bus2IP_WrCE | A | Write Chip Enable Signal zum Schreiben der User IP CORE Register (Software, FIFO Daten, Steuer usw.) Die Vektorbreite ist genau so groß, wie ipif_Bus2IP_RdCE, da die Read- und Write-Enable-Signale jeweils für ein Register gelten. |
| ipif_Bus2IP_Addr | A | Die Memory Mapped Adresse bei dem eine Lese- / Schreib-Transaktion ausgeführt wird. |
| ipif_Bus2IP_Data | A | Die PLB 32-Bit-Busdaten für das User IP CORE. Die Übernahme der Daten ins adressierte Register erfolgt durch das Chip Enable-Signal. |
| ipif_Bus2IP_RNW | A | R/W Request an das User IP CORE für die gewählten Register. |
| ipif_IP2Bus_Data | E | 32-Bit Busdaten vom User IP CORE für den PLB. |
| ipif_IP2Bus_WrAck | E | Acknowledge vom User IP CORE für ein Write Request. |
| ipif_IP2Bus_RdAck | E | Acknowledge vom User IP CORE für ein Read Request. |
| ipif_IP2Bus_Error | E | Wird in Verbindung mit ipif_IP2Bus_WrAck und ipif_IP2Bus_RdAck gesetzt, wenn beim User IP CORE in dieser Phase ein Fehler aufgetreten ist und ein Lese- / Schreib-Vorgang der Daten nicht möglich ist. |

Tabelle 3.1.: IPIF Signal Interfaces im IPIC Format vom PLB Slave für das User IP CORE

Der Aufbau der *HAW SPI* und *NET 1* IP COREs basieren auf derselben Grundstruktur eines User IP COREs mit Unterschieden in der Anwenderlogik, IPIF-Optionen und Anzahl der Software-Register. Für eine spätere Modifikation im IP CORE Design z.B. :

- bei Veränderung der Anzahl der Software Register oder
- beim Hinzufügen/Entfernen von PLB IPIF-Optionen.

ist eine Übersicht der VHDL Konstanten-Arrays für die Instanziierung der PLB Slave-Komponente und den daraus resultierenden IPIF Signal Interfaces im IPIC-Format erforderlich. Der folgende Aufbau wird mit dem Beispiel des NET 1 IP COREs mit zwei IPIF FIFOs und einem Software-Register für Status Informationen dargestellt.

Die Vektorbreiten der IPIF-Steuersignale Chip Select und Chip Enable des NET 1 IP COREs resultieren aus der Instanziierung der PLB Slave Komponente. Abhängig von der Anzahl der Software-Register und den Registern der Hardware Komponenten (Read/Write FIFO) ergibt sich der Adressraum des IP COREs, der durch Konstantendefinition der Adressbereiche dem PLB Slave bei der Parametrisierung übergeben wird.

Listing 3.1: Adressraum definition des NET 1 IP CORE

```
constant ZERO_ADDR_PAD          : std_logic_vector(0 to 31) := (others => '0');
constant USER_SLV_BASEADDR      : std_logic_vector     := C_BASEADDR or X"00000000";
constant USER_SLV_HIGHADDR      : std_logic_vector     := C_BASEADDR or X"000000FF";
constant RFF_REG_BASEADDR       : std_logic_vector     := C_BASEADDR or X"00000100";
constant RFF_REG_HIGHADDR       : std_logic_vector     := C_BASEADDR or X"000001FF";
constant RFF_DAT_BASEADDR       : std_logic_vector     := C_BASEADDR or X"00000200";
constant RFF_DAT_HIGHADDR       : std_logic_vector     := C_BASEADDR or X"000002FF";
constant WFF_REG_BASEADDR       : std_logic_vector     := C_BASEADDR or X"00000300";
constant WFF_REG_HIGHADDR       : std_logic_vector     := C_BASEADDR or X"000003FF";
constant WFF_DAT_BASEADDR       : std_logic_vector     := C_BASEADDR or X"00000400";
constant WFF_DAT_HIGHADDR       : std_logic_vector     := C_BASEADDR or X"000004FF";

constant IPIF_ARD_ADDR_RANGE_ARRAY      : SLV64_ARRAY_TYPE      :=
    (
      ZERO_ADDR_PAD & USER_SLV_BASEADDR,  -- user logic slave space base for SW-
                                          -- Register address
      ZERO_ADDR_PAD & USER_SLV_HIGHADDR,  -- user logic slave space high for SW-
                                          -- Register address
      ZERO_ADDR_PAD & RFF_REG_BASEADDR,   -- read pfifo register space base address
      ZERO_ADDR_PAD & RFF_REG_HIGHADDR,   -- read pfifo register space high address
      ZERO_ADDR_PAD & RFF_DAT_BASEADDR,   -- read pfifo data space base address
      ZERO_ADDR_PAD & RFF_DAT_HIGHADDR,   -- read pfifo data space high address
      ZERO_ADDR_PAD & WFF_REG_BASEADDR,   -- write pfifo register space base address
      ZERO_ADDR_PAD & WFF_REG_HIGHADDR,   -- write pfifo register space high address
      ZERO_ADDR_PAD & WFF_DAT_BASEADDR,   -- write pfifo data space base address
      ZERO_ADDR_PAD & WFF_DAT_HIGHADDR    -- write pfifo data space high address
    );
```

Die Deklaration der einzelnen Adressbereiche erfolgt in Tupel mit der Base- und High-Adresse. Die Größe des Adressbereichs muss eine Potenz zur Basis 2 haben. Die Basisadresse muss mit einem Wert, der ein Vielfaches der Größe des Adressraums ist, beginnen. Für das Net 1 IP CORE ergibt sich eine Vektorbreite von 5 Chip Select Signalen. Für jeden Adressbereich

müssen die Anzahl der adressierbaren Register durch Konstantendefinitionen für die IPIC Chip Enable-Signale der Lese- / Schreib-Zugriffe auf die Register deklariert werden.

Mit der Konstanten USER_SLV_NUM_REG wird die Anzahl der Software-Register deklariert. Für das NET 1 IP CORE wurde ein Software-Register zum Auslesen des Status vom CORE über den PLB gewählt. Die Read/Write FIFOs besitzen ein Pop Control-, Reset-, ModuleID- und Status-Register und eines zum Schreiben/Lesen der FIFO-Daten.

Listing 3.2: Deklaration der Anzahl für die Chip Enable Signale für die Register der jeweiligen Adressbereiche

```
constant USER_SLV_NUM_REG          : integer              := 1;
constant USER_NUM_REG              : integer              := USER_SLV_NUM_REG;
constant RFF_NUM_REG_CE            : integer              := 4;
constant RFF_NUM_DAT_CE            : integer              := 1;
constant WFF_NUM_REG_CE            : integer              := 4;
constant WFF_NUM_DAT_CE            : integer              := 1;

constant IPIF_ARD_NUM_CE_ARRAY     : INTEGER_ARRAY_TYPE   :=
    (
      0  => pad_power2(USER_SLV_NUM_REG), -- number of ce for user logic slave space
      1  => RFF_NUM_REG_CE,               -- number of ce for read pfifo register space
      2  => RFF_NUM_DAT_CE,               -- number of ce for read pfifo data space
      3  => WFF_NUM_REG_CE,               -- number of ce for write pfifo register space
      4  => WFF_NUM_DAT_CE                -- number of ce for write pfifo data space
    );
```

Daraus ergibt sich eine Gesamtanzahl von 11 Rd/Wr Chip Enable-Signalen. Mit diesen zwei konstanten Arrays und den Parametern, die systemabhängig sind, wie :

- Basisadresse,
- PLB Datenbusbreite,
- PLB Adressbusbreite und
- IPIF Datenbusbreite (User IP)

wird die PLB Slave-Komponente parametrisiert. Für die korrekte Zuweisung der einzelnen Bitpositionen der Chip Select und Chip Enable-Signal-Vektoren an die User Logic für die Software-Register und den Read- / Write FIFO werden durch Konstanten die Indexe des Vektors berechnet und gesetzt.

Listing 3.3: Index Berechnung für das Chip Select Signal der Adressbereiche und Chip Enable Signal der adressierbaren Register im Adressbereich

```
constant USER_SLV_CS_INDEX : integer := 0;
constant USER_SLV_CE_INDEX : integer := calc_start_ce_index(IPIF_ARD_NUM_CE_ARRAY,
                                                              USER_SLV_CS_INDEX);
constant RFF_REG_CS_INDEX  : integer  := 1;
constant RFF_REG_CE_INDEX  : integer  := calc_start_ce_index(IPIF_ARD_NUM_CE_ARRAY,
                                                              RFF_REG_CS_INDEX);
constant RFF_DAT_CS_INDEX  : integer  := 2;
constant RFF_DAT_CE_INDEX  : integer  := calc_start_ce_index(IPIF_ARD_NUM_CE_ARRAY,
                                                              RFF_DAT_CS_INDEX);
constant WFF_REG_CS_INDEX  : integer  := 3;
constant WFF_REG_CE_INDEX  : integer  := calc_start_ce_index(IPIF_ARD_NUM_CE_ARRAY,
```

```
                                                                WFF_REG_CS_INDEX);
constant WFF_DAT_CS_INDEX : integer  := 4;
constant WFF_DAT_CE_INDEX : integer  := calc_start_ce_index(IPIF_ARD_NUM_CE_ARRAY,
                                                                WFF_DAT_CS_INDEX);
constant USER_CE_INDEX     : integer  := USER_SLV_CE_INDEX;
```

Durch diese Parametrisierung wird der Adressraum des NET 1 IP COREs auf die Rd/Wr Chip Enable-Signale der Register abgebildet.

| **Register Name** | **Memory Mapped Adressraum** | **IPIF Rd/Wr Chip Enable-Signal** | **index** |
|---|---|---|---|
| Net 1 Status SW-Register | Baseadress + 0x0 | ipif_Bus2IP_RdCE(USER_CE_INDEX to USER_CE_INDEX + USER_NUM_REG-1); | 0 |
| Read FIFO Reset Register | Baseadress + 0x100 | ipif_Bus2IP_WrCE(RFF_REG_CE_INDEX to RFF_REG_CE_INDEX+RFF_NUM_REG_CE-1) | 1 |
| Read FIFO Module ID Register | Baseadress + 0x100 | ipif_Bus2IP_RDCE(RFF_REG_CE_INDEX to RFF_REG_CE_INDEX+RFF_NUM_REG_CE-1) | 2 |
| Read FIFO Status Register | Baseadress + 0x104 | ipif_Bus2IP_RdCE(RFF_REG_CE_INDEX to RFF_REG_CE_INDEX+RFF_NUM_REG_CE-1) | 3 |
| Read FIFO Pop Control Register | Baseadress + 0x108 | ipif_Bus2IP_WrCE(RFF_REG_CE_INDEX to RFF_REG_CE_INDEX+RFF_NUM_REG_CE-1) | 4 |
| Read FIFO Data Register | Baseadress + 0x200 | ipif_Bus2IP_RdCE(RFF_DAT_CE_INDEX) | 5 |
| Write FIFO Reset Register | Baseadress + 0x300 | ipif_Bus2IP_WrCE(WFF_REG_CE_INDEX to WFF_REG_CE_INDEX+WFF_NUM_REG_CE-1) | 6 |
| Write FIFO Module ID Register | Baseadress + 0x300 | ipif_Bus2IP_RdCE(WFF_REG_CE_INDEX to WFF_REG_CE_INDEX+WFF_NUM_REG_CE-1) | 7 |
| Write FIFO Status Register | Baseadress + 0x304 | ipif_Bus2IP_RdCE(WFF_REG_CE_INDEX to WFF_REG_CE_INDEX+WFF_NUM_REG_CE-1) | 8 |
| Write FIFO Push Control Register | Baseadress + 0x308 | ipif_Bus2IP_WrCE(WFF_REG_CE_INDEX to WFF_REG_CE_INDEX+WFF_NUM_REG_CE-1) | 9 |
| Write FIFO Data Register | Baseadress + 0x400 | ipif_Bus2IP_RdCE(WFF_DAT_CE_INDEX) | 10 |

Tabelle 3.2.: IPIC Signal Interfaces vom PLB Slave für das User IP CORE

# 4. Sensorplattform für die seitliche Abstandsmessung des SoC-CC-Fahrzeugs

In diesem Kapitel wird der Hardwareaufbau der Sensorplattform für die seitliche Abstandsmessung vorgestellt (vgl. Abbildung 4.1). Die Erfassung der seitlichen Abstände erfolgt über Infrarotsensoren. Auf jeder Seite des SoC-CC-Fahrzeugs sollen zwei Sharp GP2D12 Infrarotsensoren angebracht werden, die mit separaten Analog-Digital-Umsetztern von National ausgelesen werden. Die Ansteuerung des ADCS7476 erfolgt über den Serial Peripheral Interface (SPI) Bus. Jeweils zwei ADCS7476 ICs sind auf einem PMOD.

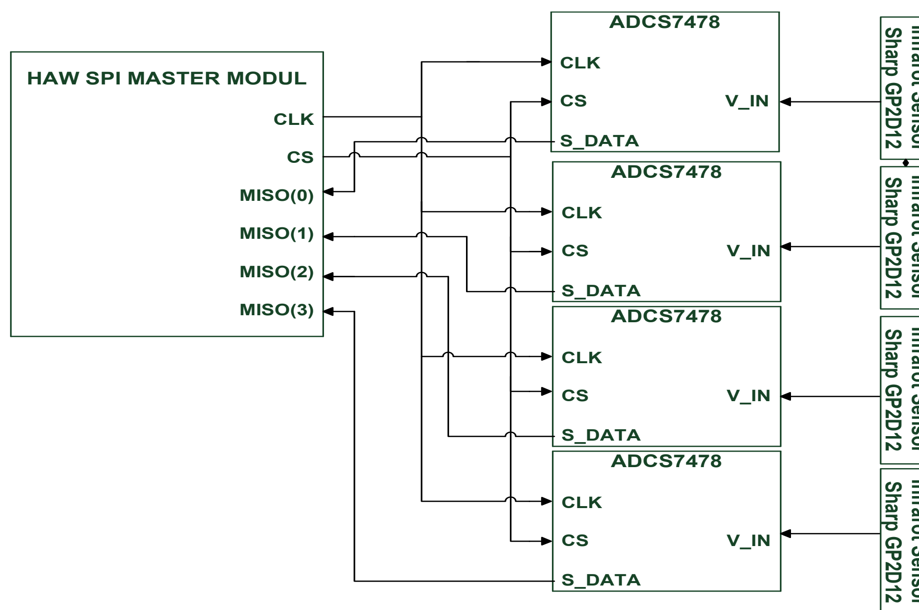



Abbildung 4.1.: Parallele Abstandsmessung mit dem HAW SPI Bus, vier Analog-Digital-Umsetzer S7476 und Sharp GP2D12 Infrarotsensoren

Verschiedene Situationen erfordern nicht nur den Abstandswert eines einzelnen Sensors zu einem Zeitpunkt, sondern denen von zweien oder allen vieren. Dem digitalen PD-Regler wird der hintere Abstandswert einer Seite zugeführt, wohingegen bei der Parklückensuche zwei seitliche Abstandswerte benötigt werden. Nach einem Ausweichmanöver sollen die Abstände vom Fahrzeug zwischen seitlicher Fahrbahnbegrenzung und dem Hindernis erfasst werden. Werden Abstandswerte mehrerer Sensoren zeitparallel benötigt, ist eine paralle Erfassung erforderlich.

Die parallele Sensordatenerfassung wurde durch das HAW SPI IP CORE mit den folgenden Entwurfszielen realisiert :

- Parallele Ansteuerung der Analog-Digital-Umsetzer für einen parallelen Empfang der seriellen Messdaten.
- Die Verwendung beider ADCS7476 ICs auf einem PMOD.
- PLB-Busanbindung zum MicroBlaze Prozessor

## 4.1. HAW SPI IP CORE

Die parallele Ansteuerung und der Empfang der Messdaten von den ADCS7476-Modulen erfordert einen neuen Entwurf der Bustopologie, die vom Standard SPI abweicht (vgl. Kapitel 4.1.1). Die korrekte Ansteuerung der ADCS7476-Module setzt die Einhaltung der Transferformate vom Standard SPI Bus beim HAW SPI IP CORE vorraus (vgl. Kapitel 4.1.2).

Das im Rahmen dieser Arbeit erstellte HAW SPI IP CORE ermöglicht den parallelen Empfang der Datenströme der vier gleichartigen (auf zwei PMODs platzierten) ADCS7476 SPI Slave Devices (vgl. Abbildung 4.2, Anhang A).

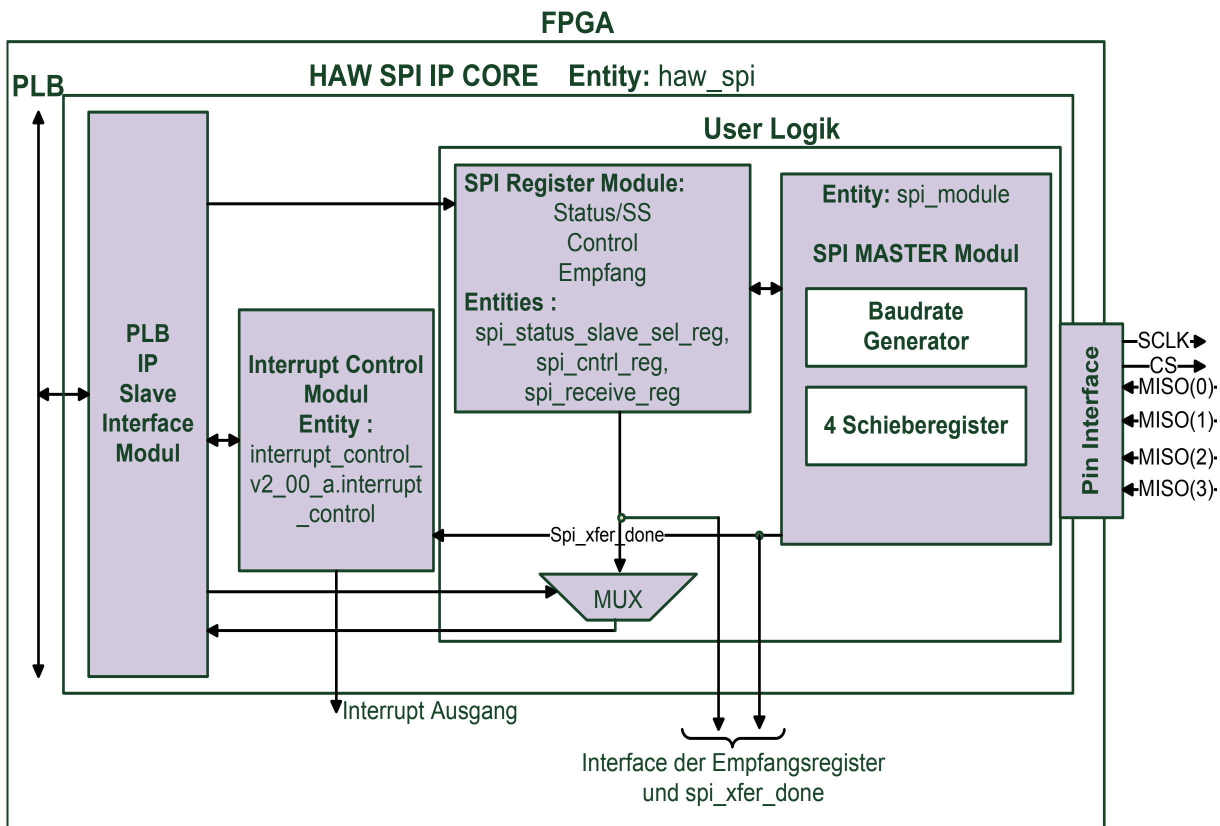



Abbildung 4.2.: Top Entity-Blockschaltbild des HAW SPI IP COREs

Mit dem Create Peripheral Tool wurden für das HAW SPI IP CORE

- ein PLB Slave Interface,
- die PLB Slave IPIF Interrupt Control-Option und
- 7 Software Register für Status, Control, Slave Select und Daten (4)

selektiert.

In der Anwenderlogik wurden aus dem Xilinx XPS SPI die SPI Master-Funktionalitäten übernommen und für das HAW SPI IP CORE angepasst (vgl. [Xilinx (2008)]). Dadurch werden die erforderlichen Standard SPI Transferformate vom HAW SPI eingehalten.
Die parallele Ansteuerung erfolgt durch das selbe Chip Select Signal für alle vier SPI Slave Devices. Der parallele Empfang erfolgt durch vier separate MISO Signalleitungen, die mit den Eingängen der vier Schieberegister verbunden sind. Mit dem SCKL-Takt-Signal werden die SPI Slave Devices und die Schieberegister getaktet. Dadurch wird die erforderliche Bustopologie für den parallelen Empfang und voller Ausnutzung der ADCS7476 ICs auf den PMODs realisiert (vgl. Abbildung 4.1). Die Empfangsregister und das Status-Signal spi_xfer_done, das mit einem Flag das Ende eines Transfers kennzeichnet, werden als CORE Interface nach außen geführt. Dadurch können z.B. Beschleuniger-Module, die Busdaten direkt erhalten, ohne das die über den PLB ausgelesen werden.

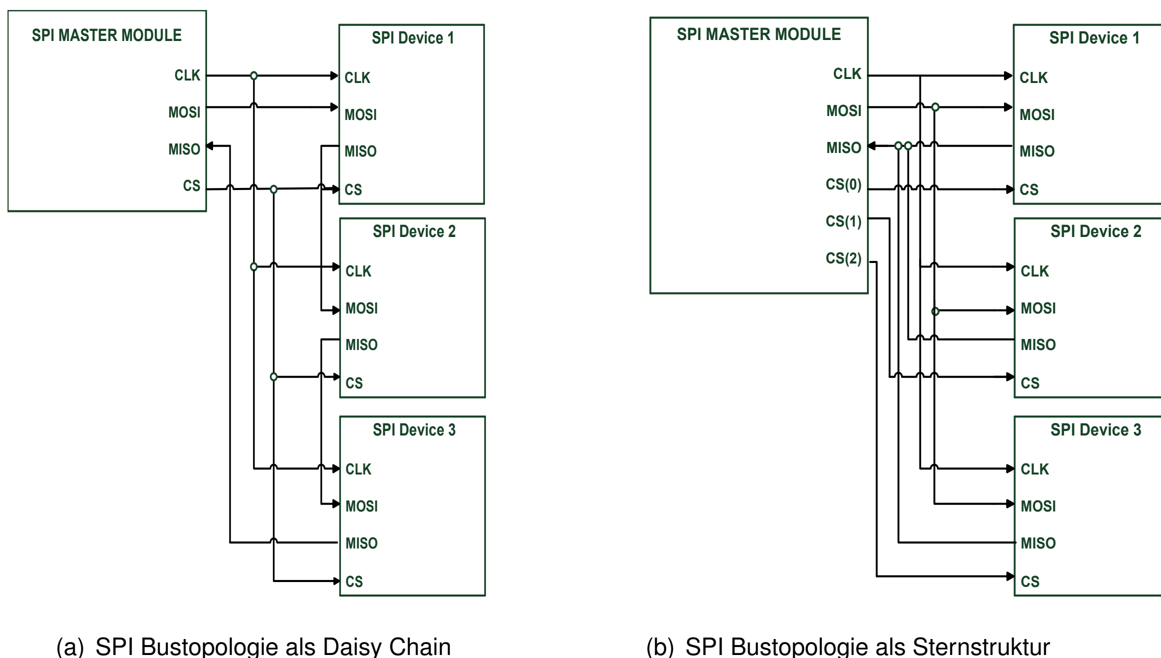
### 4.1.1. Standard Serial Peripheral Interface (SPI)

Das Serial Peripheral Interface (SPI) ist ein von Motorola entwickelter synchroner serieller Datenbus. Das Bussystem arbeitet nach dem Master-Slave-Prinzip und ist Vollduplexfähig.

| **Signal Name** | **Beschreibung** |
|---|---|
| SCLK | Das SPI Master-Modul generiert den Takt für die synchrone Datenübertragung |
| MOSI | Master out Slave in : Das Master-Modul taktet die Datenbits für den Slave über diese Signalleitung mit SCLK herraus. |
| MISO | Master in Slave out : Das Master-Modul taktet die Datenbits vom Slave über diese Signalleitung mit SCLK ein. |
| $\overline{SS}$ | Die Selektierung des Slaves durch das Master-Modul. |

Tabelle 4.1.: Signalleitungen des SPI Busses

Die parallele Erfassung der vier seitlichen Abstände ist aufgrund der möglichen Bustopologien dieses Standards nicht möglich (vgl. Abbildung 4.3).




(a) SPI Bustopologie als Daisy Chain (b) SPI Bustopologie als Sternstruktur

Abbildung 4.3.: Zwei verschiedene Bustopologien lassen sich mit den Signalleitungen eines Standard SPI Master Moduls aufbauen.

Durch das Layout und die Verdrahtung der ADCS7476 ICs auf einem PMOD ist der Aufbau der Bustopologie eingeschränkt (vgl. Abbildung 4.4). Für eine Daisy Chain-Topologie fehlen die MISO-Schieberegister (vgl. Abbildung 4.3). Eine Sterntopologie ist aufgrund der Verdrahtung der CS-Signalleitungen nicht möglich (vgl. Abbildung 4.3). Bei Selektierung der IC durch das CS-Signal würden beide die MISO-Signalleitung treiben. Durch Verwendung eines Standard SPI Master-Moduls, wie z.B. dem Xilinx XPS SPI, ist nur die Ansteuerung eines ADCS7476 ICs auf einem PMOD zu einem Zeitpunkt möglich.

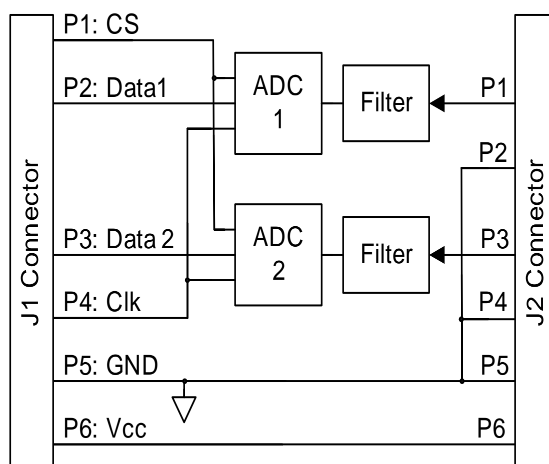



Abbildung 4.4.: PMOD Modul mit zwei ADCS7476 (vgl. Digilent (2004b))

### 4.1.2. Transferformate des SPI Busses

Ein Protokoll für die Datenübertragung wurde von Motorola nicht festgelegt. In der Praxis haben sich somit vier verschiedene Konfiguration durchgesetzt, die durch die Taktphase (CPHA) und Taktpolarität (CPOL) festgelegt sind. Die vier Konfigurationen ergeben sich aus zwei Transferformaten. Der Master ist der Initiator eines Bustransfers. Master und Slave müssen die gleiche Konfiguration für einen erfolgreichen Transfer haben. Die vier Konfigurationen, bei dem der Master Empfänger ist und der Slave Sender, werden untersucht, um die korrekte Konfiguration für den ADCS7478 zu erlangen (vgl. Abbildung 4.5). Das Chip Select ($\overline{CS}$) Signal ist äquivalent zum Slave Select ($\overline{SS}$) Signal.

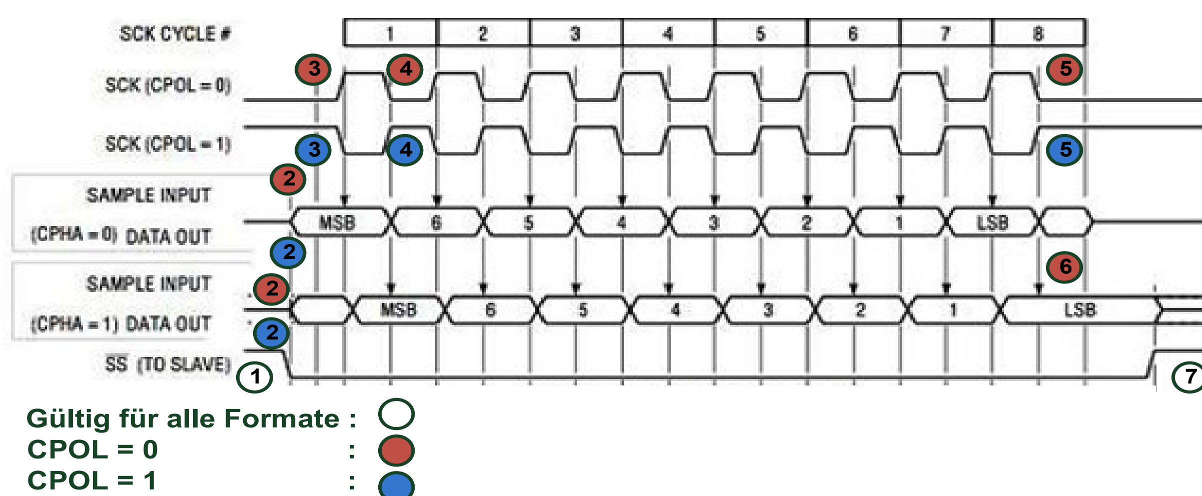



Abbildung 4.5.: Timingverhalten der vier Transferformate des SPI Busses [Motorola (2005)]

**Clock Polarität = Idle Low, CPOL = 0**
**Master : Phasenlage, Gültigkeit der Daten bei der ersten Flanke, CPHA = 0**
**Slave : Transfer beginnt mit fallender Flanke von $\overline{SS}$, CPHA = 0**

1. Der Master startet einen Buszyklus durch das low aktive $\overline{SS}$ Signal.
2. Durch das low aktive $\overline{SS}$ Signal verlässt der Ausgang des SPI Slave Device den hochohmigen Zustand. Das MSB wird auf den Bus gelegt und bis zur fallenden Flanke stabil gehalten.
3. Das SPI Slave Device muss nach jeder steigenden Flanke den Wert für eine halbe Taktperiode stabil halten, damit das Master Modul diesen Wert abtasten kann. Das Master Modul tastet die acht Datenbits am Eingang nach jeder steigenden Flanke ab.
4. Bei der fallenden Flanke hat das SPI Slave Device eine halbe Taktperiode Zeit zum schieben des nächsten Datenbits und muss es in der anderen Hälfte stabil halten.
5. Nach der achten fallenden Flanke wurde das Byte vom Master Empfangen und der Buszyklus wird beendet. Nachdem das $\overline{SS}$ Signal = 1 ist geht der Ausgang des SPI Slave Device wieder in den hochohmigen Zustand.

**Clock Polarität = Idle High, CPOL = 1**
**Master : Phasenlage, Gültigkeit der Daten bei der ersten Flanke, CPHA = 0**
**Slave : Transfer beginnt mit fallender Flanke von $\overline{SS}$, CPHA = 0**

1. Der Master startet einen Buszyklus durch das low aktive $\overline{SS}$ Signal.
2. Durch das low aktive $\overline{SS}$ Signal verlässt der Ausgang des SPI Slave Device den hochohmigen Zustand. Das MSB wird auf den Bus gelegt und bis zur steigenden Flanke stabil gehalten.
3. Das SPI Slave Device muss nach jeder fallenden Flanke den Wert für eine halbe Taktperiode stabil halten, damit das Master Modul diesen Wert abtasten kann. Das Master Modul tastet die acht Datenbits am Eingang nach jeder fallenden Flanke ab.
4. Bei der steigenden Flanke hat das SPI Slave Device eine halbe Taktperiode Zeit zum schieben des nächsten Datenbits und muss es in der anderen Hälfte stabil halten.
5. Nach der achten steigenden Flanke wurde das Byte vom Master Empfangen und der Buszyklus wird beendet. Nachdem das $\overline{SS}$ Signal = 1 ist geht der Ausgang des SPI Slave Device wieder in den hochohmigen Zustand.

**Clock Polarität = Idle Low, CPOL = 0**
**Master : Phasenlage, Gültigkeit der Daten bei der zweiten Flanke, CPHA = 1**
**Slave : Transfer beginnt mit 1. Flanke von SCK und dem low aktiven $\overline{SS}$ Signal, CPHA = 1**

1. Der Master startet einen Buszyklus durch das low aktive $\overline{SS}$ Signal.
2. Durch das low aktive $\overline{SS}$ Signal verlässt der Ausgang des SPI Slave Device den hochohmigen Zustand. Bedingt durch die Phasenlageneinstellung des SPI Slave Moduls wird nun ein Transfer durch die erste Flanke von SCK initiiert.
3. Nach der steigenden Flanke hat das SPI Slave Device eine halbe Taktperiode Zeit das nächste Bit zu schieben und es in der anderen Hälfte stabil zu halten.
4. Das SPI Master Modul tastet die Datenbits nach jeder fallenden Flanke ab.
5. Bei der achten fallenden Flanke tastet das SPI Master Modul das letzte Datenbit ab.
6. Mit dem Setzten des low aktive $\overline{SS}$ Signals wird der Buszyklus beendet und der Ausgangs des SPI Slave Device wechselt in den hochohmigen Zustand.

**Clock Polarität = Idle High, CPOL = 1**
**Master : Phasenlage, Gültigkeit der Daten bei der zweiten Flanke, CPHA = 1**
**Slave : Transfer beginnt mit 1. Flanke von SCK und dem low aktiven $\overline{SS}$ Signal, CPHA = 1**

1. Der Master startet einen Buszyklus durch das low aktive $\overline{SS}$ Signal.
2. Durch das low aktive $\overline{SS}$ Signal verlässt der Ausgang des SPI Slave Device den hochohmigen Zustand. Bedingt durch die Phasenlageneinstellung des SPI Slave Moduls wird nun ein Transfer durch die erste Flanke von SCK initiiert.
3. Nach der fallenden Flanke hat das SPI Slave Device eine halbe Taktperiode Zeit das nächste Bit zu schieben und es in der anderen Hälfte stabil zu halten.
4. Das SPI Master Modul tastet die Datenbits nach jeder steigenden Flanke ab.
5. Bei der achten steigenden Flanke tastet das SPI Master Modul das letzte Datenbit.
6. Mit dem Setzten des low aktive $\overline{SS}$ Signals wird der Buszyklus beendet und der Ausgangs des SPI Slave Device wechselt in den hochohmigen Zustand.

### 4.1.3. SPI Bustransfermessungen

Für die vier verschiedene Konfigurationen wurde der Bustransfer des HAW SPI gemessen.

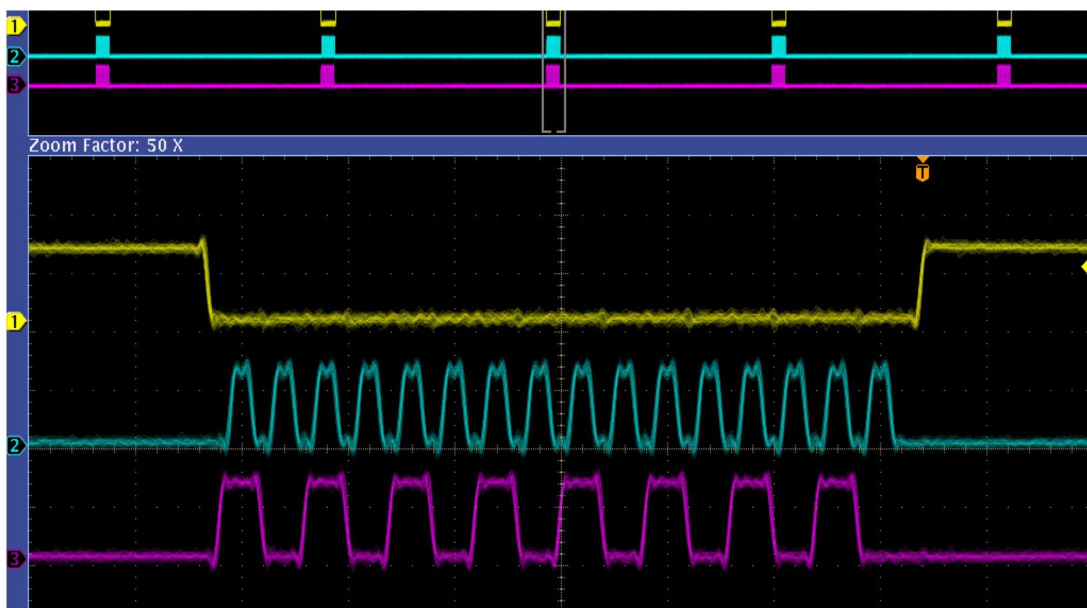



Abbildung 4.6.: SPI Bustransfer mit CPOL = 0; CPHA = 0; SCLK = 12,5MHZ; 16 Datenbits; (1) CS; (2) SCLK; (3) MISO 0xaaaa

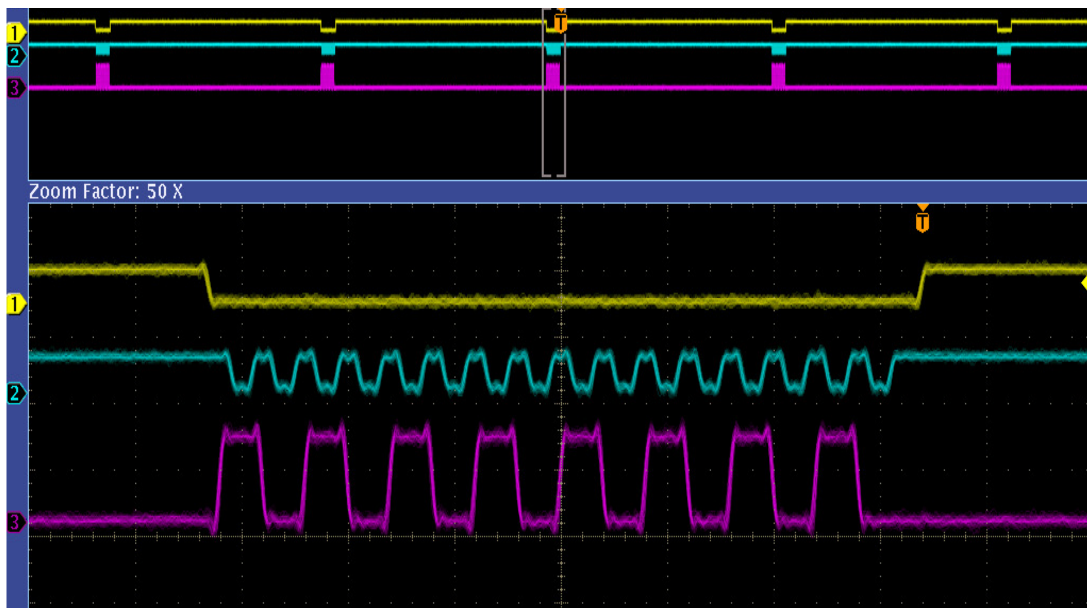



Abbildung 4.7.: SPI Bustransfer mit CPOL = 1; CPHA = 0; SCLK = 12,5MHZ; 16 Datenbits; (1) CS; (2) SCLK; (3) MISO 0xaaaa

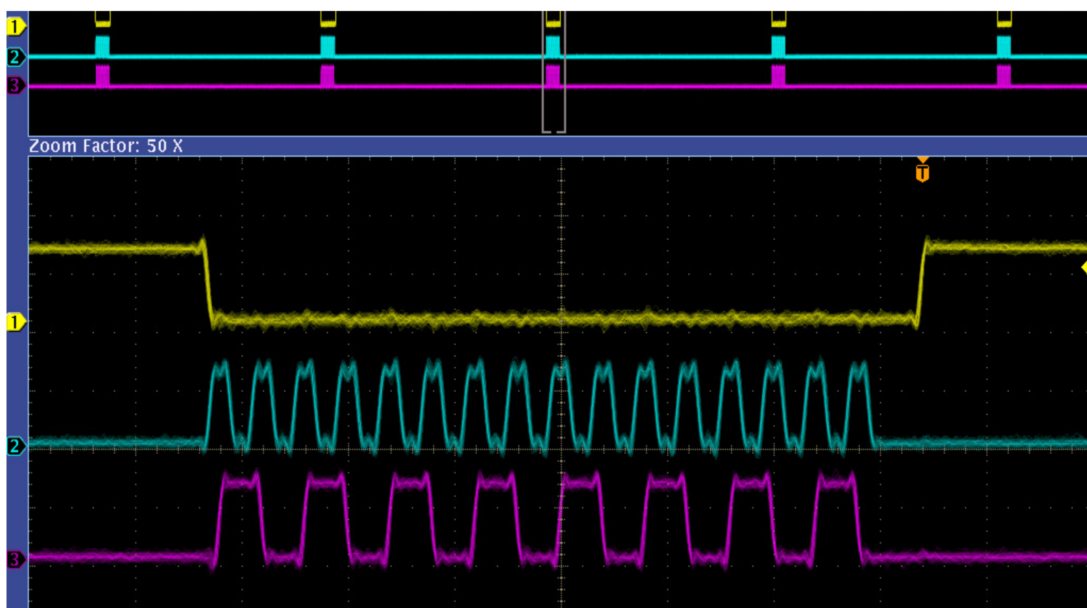



Abbildung 4.8.: SPI Bustransfer mit CPOL = 0; CPHA = 1; SCLK = 12,5MHZ; 16 Datenbits; (1) CS; (2) SCLK; (3) MISO 0xaaaa

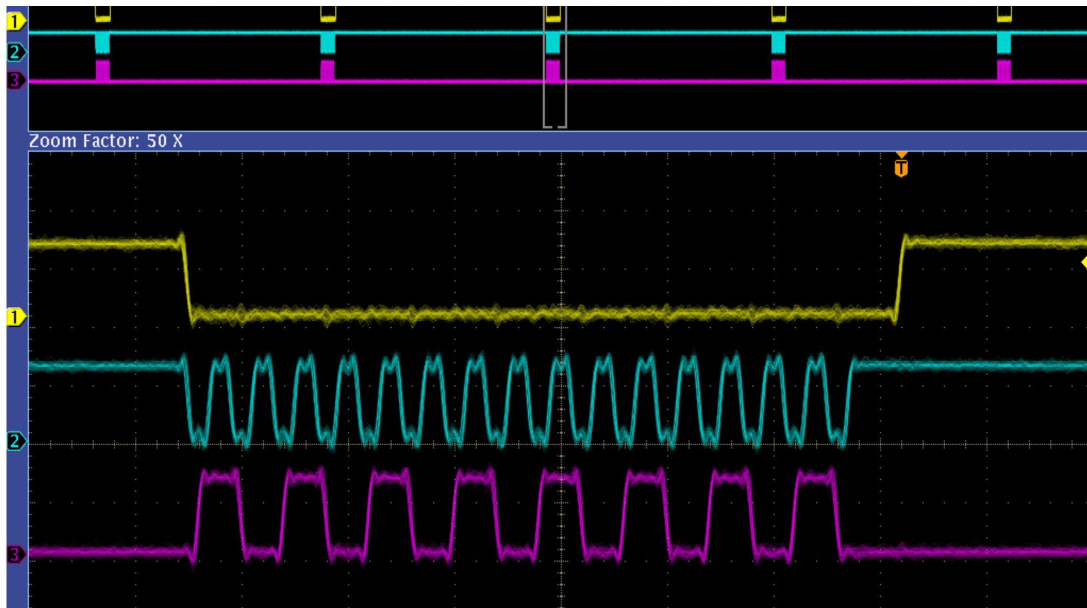



Abbildung 4.9.: SPI Bustransfer mit CPOL = 1; CPHA = 1; SCLK = 12,5MHZ; 16 Datenbits; (1) CS; (2) SCLK; (3) MISO 0xaaaa

## 4.2. Analog-Digital-Umsetzter National ADCS7476

Die Abtastrate des ADCS7476 beträgt 1MHZ bei einer Auflösung von 12-Bit für Spannungen zwischen 0V - 3,3V.

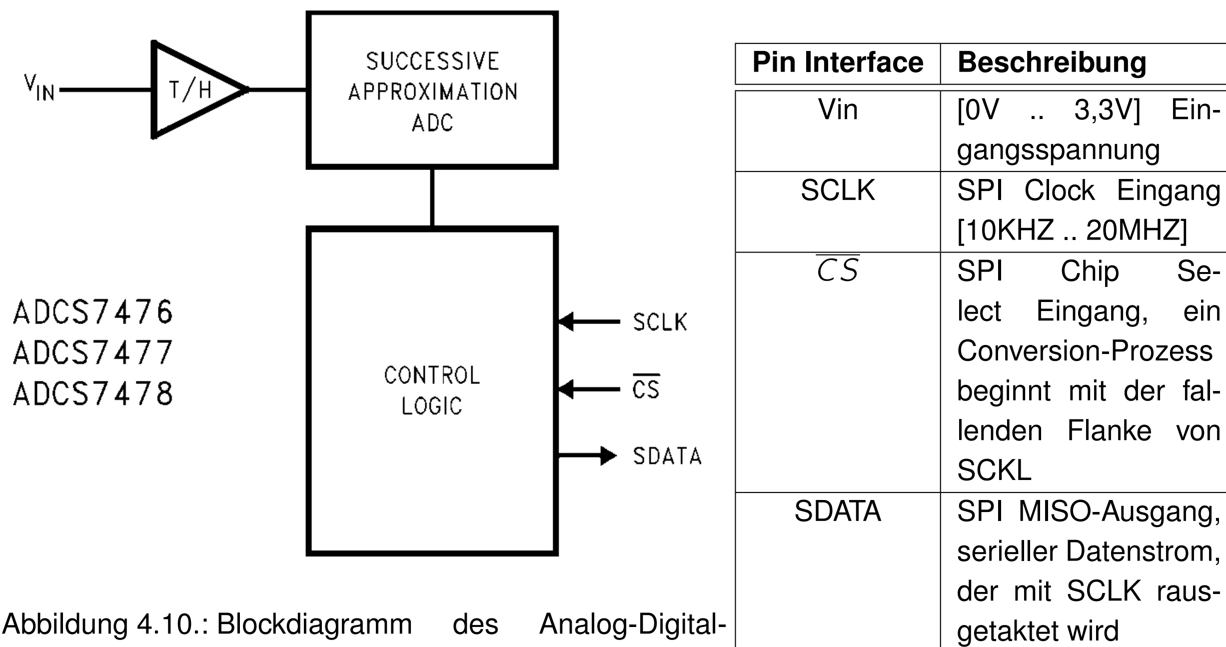



Abbildung 4.10.: Blockdiagramm des Analog-Digital-Umsetzter ICs [National (2007)]

| Pin Interface | Beschreibung |
|---|---|
| Vin | [0V .. 3,3V] Eingangsspannung |
| SCLK | SPI Clock Eingang [10KHZ .. 20MHZ] |
| $\overline{CS}$ | SPI Chip Select Eingang, ein Conversion-Prozess beginnt mit der fallenden Flanke von SCKL |
| SDATA | SPI MISO-Ausgang, serieller Datenstrom, der mit SCLK rausgetaktet wird |

### 4.2.1. Timingverhalten des National ADCS7476

Ein Conversion Prozess und der dazugehörige Datentransfer beginnt mit der fallenden Flanke des ChipSelect(CS) Signals. Nachdem das CS Signal auf Low getrieben wurde, verlässt die Datenleitung SDATA den hochohmigen Zustand. Die Datenleitung SDATA kommt erst dann wieder in den hochohmigen Zustand falls nach der 16 fallenden Flanke nachdem die CS Leitung auf Low getrieben wurde oder bei einer steigenden Flanke von CS (vgl.National (2007)).

Insgesamt werden 16 Takte benötigt, um ein komplettes Abtast Wert zu erhalten. Jeder Abtast Wert(beinhaltet alle vorangegangene Nullen und 12 Datenbits) werden mit der fallenden Flanke geschoben und am Bus angelegt, der Master hat dann bis zur nächsten fallenden Flanke Zeit den Wert zu abzutasten. Das letzte Datenbit ist auf der sechzehnten fallenden Flanke gültig.

Abhängig von der Anwendung ist die erste Flanke, nachdem CS auf Low getrieben wurde, eine fallende oder steigende Flanke. Falls es eine steigende Flanke ist, so sind alle vier vorangegangen Nullen auf der fallenden Flanke gültig, sonst nur drei (vgl. National (2007)), dass in der Abbildung 4.11 mit der Konfiguration CPOL = Idle High dargestellt ist.

Eine Widersprüchliche Aussage, dass nach der sechzehnten fallenden Flanke der Ausgang in den hochohmigen Zustand wechselt und das letzte Datenbit auf der sechzehnten fallenden Flanke gültig ist (vgl.National (2007),Abbildung 4.11).
Die erste Flanke nach dem das CS Signal auf Low getrieben wurde ist eine fallende, d.h. nicht vier sondern drei Nullen werden gesendet bzw. sind gültig (vgl. Abbildung 4.11). Somit hängt

es vom Master ab, ob die erste Flanke eine steigende oder fallende ist. Nach der fünfzehnten steigenden Flanke ist das LSB gültig und die darauf folgende Sechzehnte fallende Flanke bewirkt, dass die Datenleitung in den Hochohmigen Zustand wechselt.

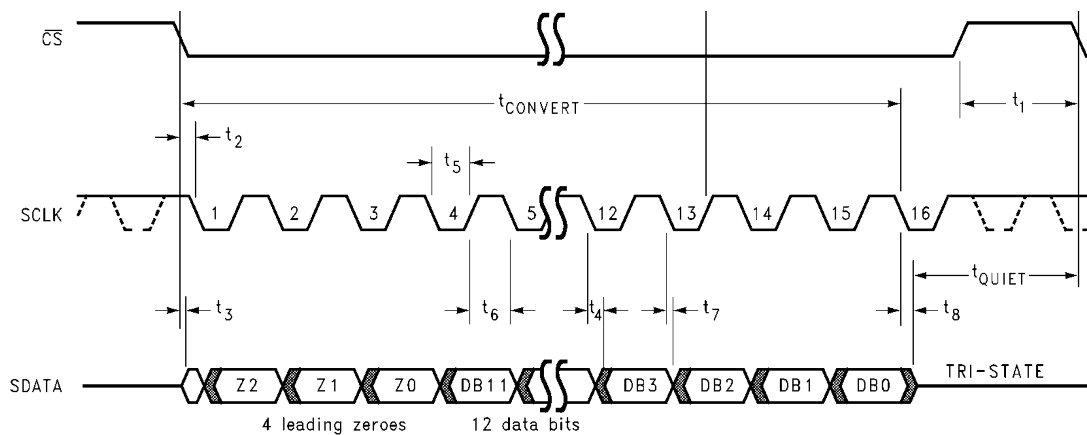



Abbildung 4.11.: Timingdiagramm des ADCS7476 (vgl.National (2007))

### 4.2.2. Timingverhalten des ADCS7476 mit HAW SPI IP CORE

Für einen erfolgreichen Transfer müssen Master und Slave dieselbe Phasenlage und Clock Polarität aufweisen. Der Master hält eine halbe Clock Periode, nachdem das CS Signal auf Low getrieben wurde die Clock Polarität im Idle Zustand und wechselt erst dann in den aktiven Zustand. Dadurch, dass der ADCS7476 im SCK Idle High Zustand befindet, muss die Clock Polarität des HAW SPI IP CORE mit CPOL = 1 konfiguriert werden.

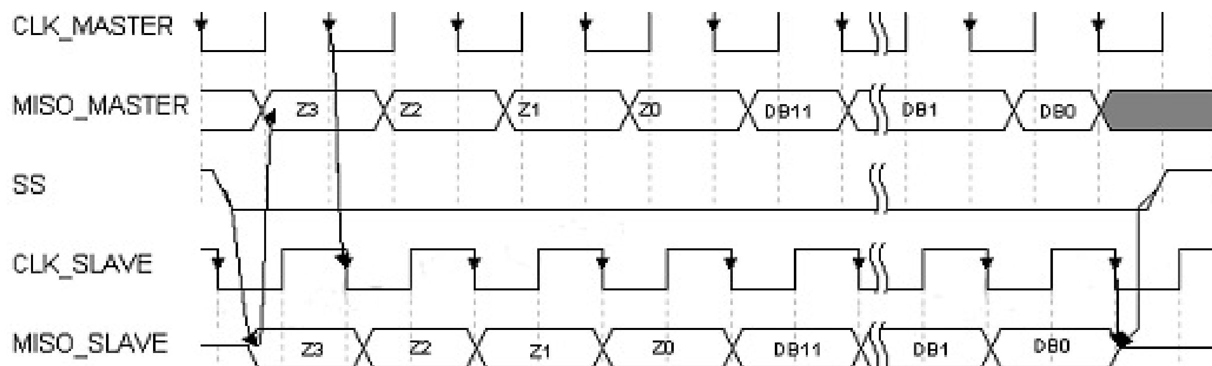



Abbildung 4.12.: Timingdiagramm ADCS7476 mit HAW SPI IP (CPOL = 1, CPOHA = 0)

Abbildung 4.12 zeigt für die Konfiguration CPOL = 1 und CPHA = 0, dass nachdem CS auf Low getrieben wurde die Datenleitung SDATA den hochohmigen Zustand verlässt und ein definierter Pegel am Ausgang liegt, hier die Null.

Nach der ersten fallenden Flanke ist die 1. Null gültig auf der Master Seite und auf der Slave Seite wird nach jeder fallenden Flanke der nächste Wert geschoben und am Ausgang gelegt, dadurch ist der Wert für den Master bei der nächsten fallenden Flanke gültig. Jedes einzelne Bit(vorangegangenen Nullen und Datenbits) werden nach jeder fallenden Flanke getaktet (vgl.National (2007)), dass ein Widerspruch ist, denn die erste Null ist durch den Zustandswechsel der Datenleitung bedingt. In den darauf folgenden 15. fallenden Flanken werden dann jeweils

drei Nullen und zwölf Datenbits an den Ausgang geschoben und zum abtasten bereitgestellt, so dass nach der 16. fallenden Flanke die Datenleitung wieder in den Hochohmigen Zustand wechselt.

Eine Alternativ Konfiguration mit CPOL = 1 und CPHA = 1 verhält sich analog wie in Abbildung 4.12 mit der Ausnahme, dass die erste Null nicht gültig ist. Die Konfiguration mit CPOL = 0 und CPHA = * funktioniert nicht, weil die Clock Polarität des ADCS7476 sich im Idle High Zustand befindet.

## 4.3. Infrarotsensor Sharp GP2D12

Der Sharp GP2D12 Infrarotsensor erfasst Abstände zwischen 10cm und 80cm.

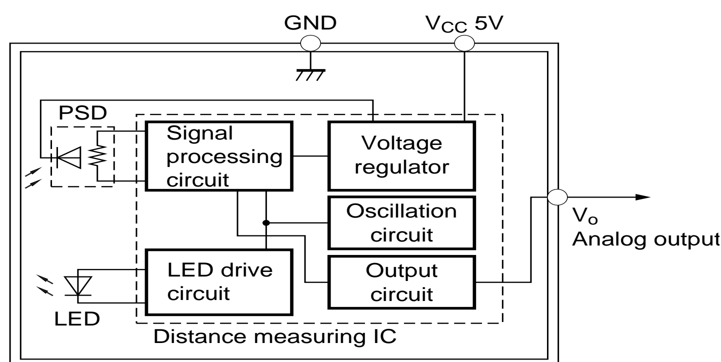



Abbildung 4.13.: Sharp GP2D12 Infrarotabstandssensor [SHARP]

Durch die Infrarot-LED wird das Licht zum Objekt gestreut und aus dem reflektierenden Strahl mit einer Diode erfasst (vgl. Abbildung 4.13). Das Distance measuring IC berechnet aus der Zeit, mit dem der Infrarotstrahl gesendet und empfangen wurde, den Abstand zum reflektierenden Objekt und setzt den analogen Ausgang. Für eine Messung benötigt das IC im best case 28,7ms, und im worst case 47,9ms was einer Abtastrate von $\approx$ 20 Hz entspricht (vgl. Abbildung 4.14).

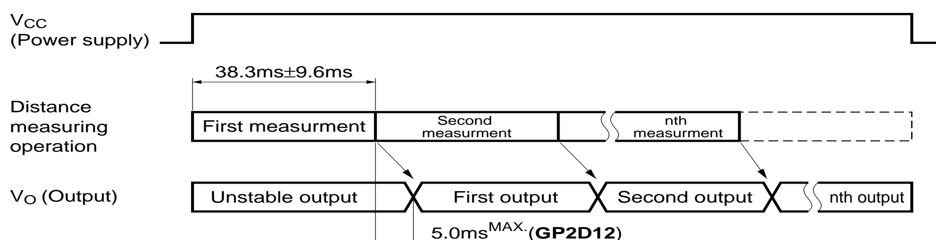
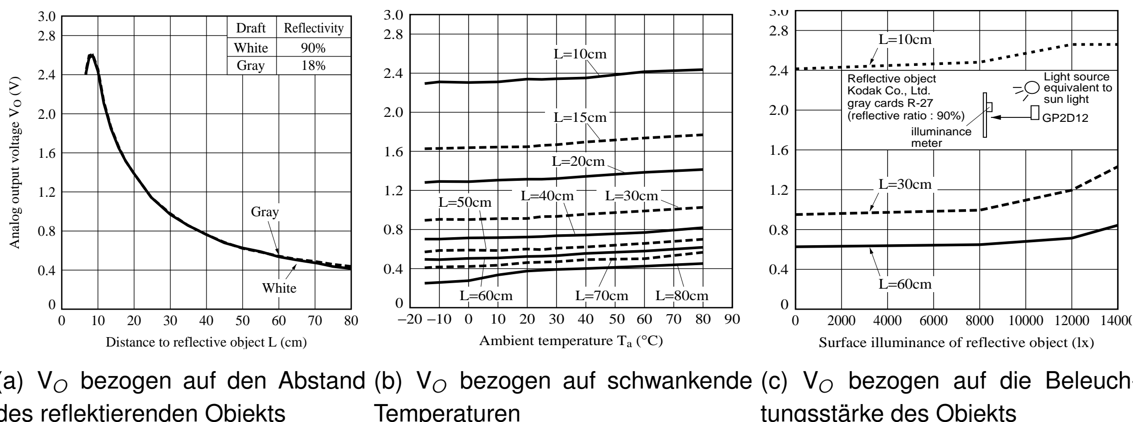



Abbildung 4.14.: Sharp GP2D12 Timing [SHARP]

Der nichtlineare Verlauf der Ausgangspannung bezogen auf die Entfernung des Objekts hängt von der Temparatur, Beleuchtungsstärke und Reflexionsmaterial der Umgebung ab (vgl. Abbildung 4.15).




(a) $V_O$ bezogen auf den Abstand des reflektierenden Objekts

(b) $V_O$ bezogen auf schwankende Temperaturen

(c) $V_O$ bezogen auf die Beleuchtungsstärke des Objekts

Abbildung 4.15.: Die nichtlineare Ausgangsspannung $V_O$ bezogen auf die Entfernung des Objekts mit den Einflussfaktoren Temperatur, Beleuchtungsstärke und Reflexionsmaterial der Umgebung [SHARP]

### 4.3.1. Umrechnung der Ausgangsspannung $V_O$ in einen Abstand

Für die Umrechnung der Ausgangsspannung in einen Abstand wird die Umkehrfunktion der Ausgangsspannung über den Abstand des reflektierenden Objekts gebildet (vgl. Abbildung 4.16).

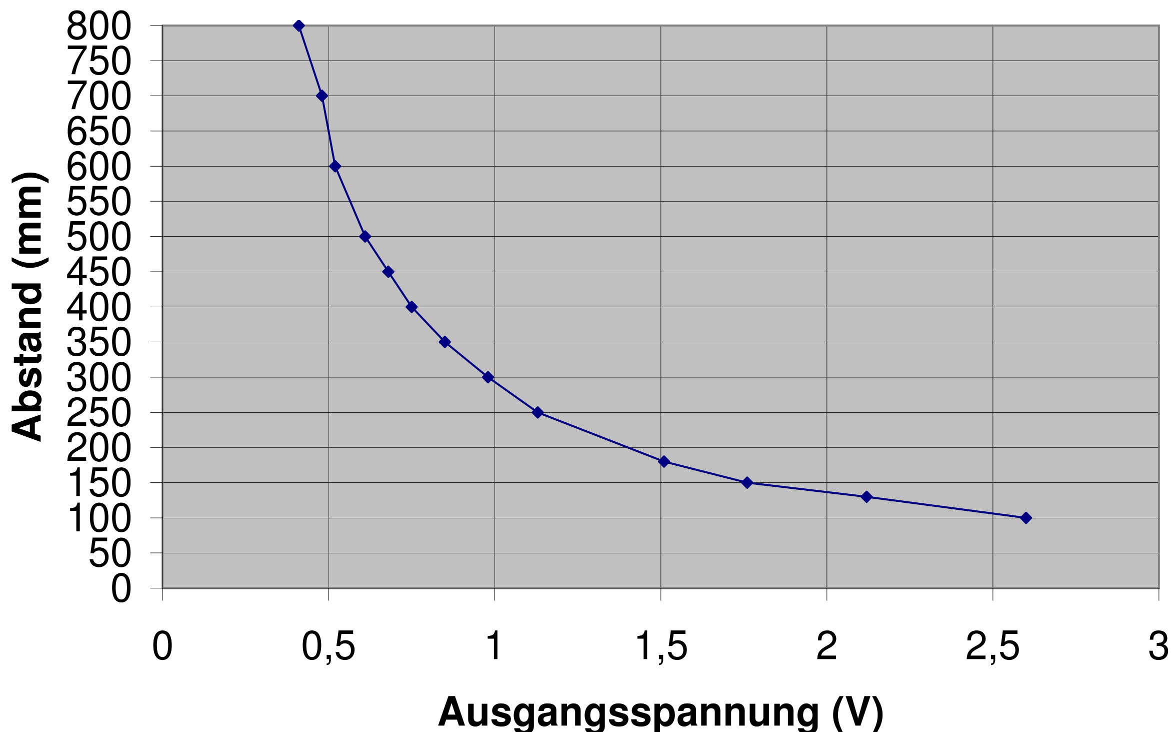



Abbildung 4.16.: Umkehrfunktion der Ausgangsspannung über die Entfernung des reflektierenden Objekts

| $V_O$ | Abstand (mm) | $V_O$ | Abstand (mm) | $V_O$ | Abstand (mm) | $V_O$ | Abstand (mm) |
|---|---|---|---|---|---|---|---|
| 0,41 | 800 | 0,68 | 450 | 0,98 | 300 | 1,76 | 150 |
| 0,48 | 700 | 0,75 | 400 | 1,13 | 250 | 2,12 | 130 |
| 0,52 | 600 | 0,85 | 350 | 1,51 | 180 | 2,6 | 100 |
| 0,61 | 500 | | | | | | |

Tabelle 4.2.: Wertetabelle der Umkehrfunktion

Die Umkehrfunktion für die Umrechnung der Ausgangsspannung $V_O$ in einen Abstand wird durch ein Polynom vierten Grades approximiert (vgl. Gleichung 4.1)

$$P(V_O) = a_4 \cdot V_O^4 + a_3 \cdot V_O^3 + a_2 \cdot V_O^2 + a_1 \cdot V_O + a_0 \tag{4.1}$$

Die Ermittlung der Koeffizienten $a_4$,$a_3$,$a_2$,$a_1$ und $a_0$ des Polynoms erfolgt durch die *Polynomial curve fitting*-Funktion von MATLAB [TheMathWorks]. Mit der Wertetabelle 4.2 und dem Grad des Polynoms (= 4) wurden mit der *Polynomial curve fitting*-Funktion folgende Koeffizienten ermittelt :

- $a_4$ = 0.1711 · $10^3$
- $a_3$ = -1.2092 · $10^3$
- $a_2$ = 3.1406 · $10^3$
- $a_1$ = -3.6678 · $10^3$
- $a_0$ = 1.8416 · $10^3$

Durch Einsetzten der Koeffizienten in die Gleichung 4.1 wird das Polynom für die Umrechnung der Ausgangsspannung $V_O$ in einen Abstand (in mm) gebildet (vgl. Gleichung 4.2).

$$P(V_O) = (0.1711 \cdot V_O^4 + -1.2092 \cdot V_O^3 + 3.1406 \cdot V_O^2 + -3.6678 \cdot V_O + 1.8416) \cdot 10^3 \tag{4.2}$$

Die Berechnung der Funktionswerte erfolgt durch die Floating Point Unit des MicroBlaze Prozessors.

Mit den HAW SPI IP CORE Interfaces der Empfangsregister und spi_xfer_done wurde eine Option für die direkte Anbindung an einen CoProzessor geschaffen. Daduch würde die Berechnung der Funktionswerte durch einen CoProzessor erfolgen.

## 4.4. Ermittlung der maximalen Abtastfrequenz des ADCS7476 mit SPI Transfers

Eine gesonderte Aufgabenstellung dieser Arbeit war es, die maximal realisierbare Abtastfrequenz des ADCS7476 über den SPI Bus zu ermitteln. Die Abtastzeit beschreibt die Dauer zwischen zwei Abtastpunkten, in der das Signal abgetastet wird und daraus die maximal realisierbare Abtastfrequenz resultiert. Nach dem Nyquist-Shannon-Abtasttheorem muss die Abtastfrequenz mindestens doppelt so gross sein wie die höchste Signal vorkommende Frequenz.

Die maximal realisierbare Abtastfrequenz wird ermittelt, indem das Zeitintervall zwischen dem Start und Empfang der Daten eines SPI-Bustransfers gemessen wird. Der Systemtakt beträgt 50MHZ. Für den SPI-Bustransfer wurde der kleinste zulässige Teiler C_SCK_RATIO = 4 gewählt. Daraus resultiert ein 12,5MHZ SPI-Takt. Bei 16 Datenbits beträgt die Hardwarelaufzeit 1,28$\mu$s. Mit einem Hardware Controller würde die maximal realisierbare Abtastfrequenz des Busses $f_{max} = \frac{1}{1{,}28\mu s} \approx$ 781KHZ betragen. Der Spi-Bus wird aber softwaretechnisch über PLB gesteuert, wodurch sich $f_{max}$ verringert

Ein Timer startet mit einer fixen Periode einen High-Level Funktionsaufruf für einen HAW SPI-Bustransfer. Durch die Interrupt-Steuerung werden die Busdaten am Ende des Transfers in der ISR ausgelesen. Das Zeitintervall $\Delta t_1$, das zwischen einem HAW SPI Transfer-Start und dem Auslesen des Abtastwertes in der ISR liegt, bestimmt die maximal realisierbare Abtastfrequenz (vgl. Abbildung 4.17). Für die Messung des Zeitintervalls $\Delta t_1$ wurde ein GPIO-Pin als Ausgang gesetzt, der am Anfang eines Bustransfers auf High und nach Auslesen des Abtastwertes auf Low gesetzt wird (Trigger-Signal).

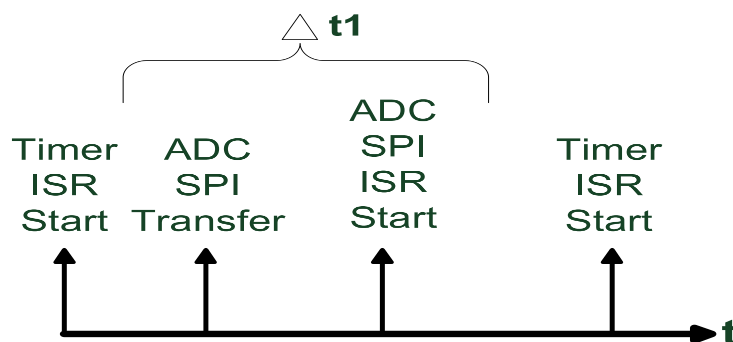



Abbildung 4.17.: Das Zeitintervall $\Delta t_1$ bestimmt die maximal realisierbare Abtastfrequenz

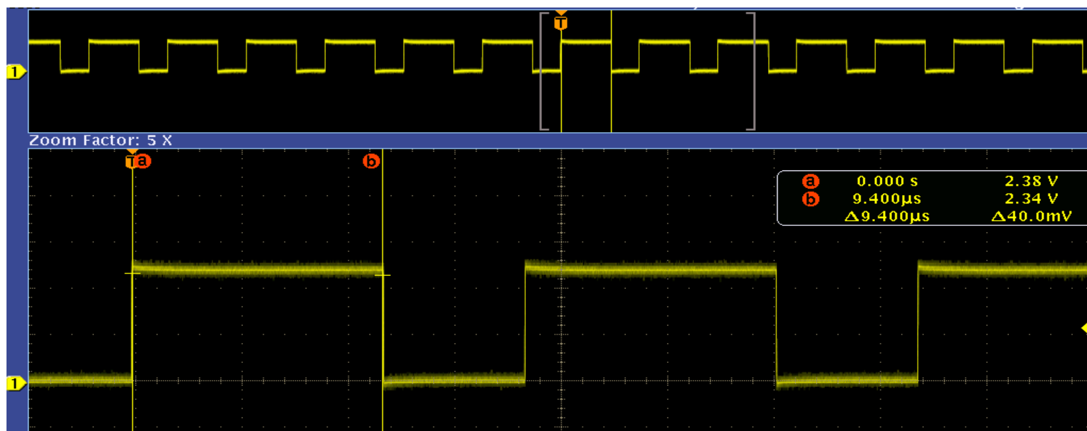



Abbildung 4.18.: Zeitmessung $\Delta t_1$ ohne Compiler Optimierungsstufe mit SCLK = 12,5MHZ; 16 Datenbits; Das Zeitintervall der Low Phase entspricht der restlichen Zeit der Timer Periode nach der Abtastung

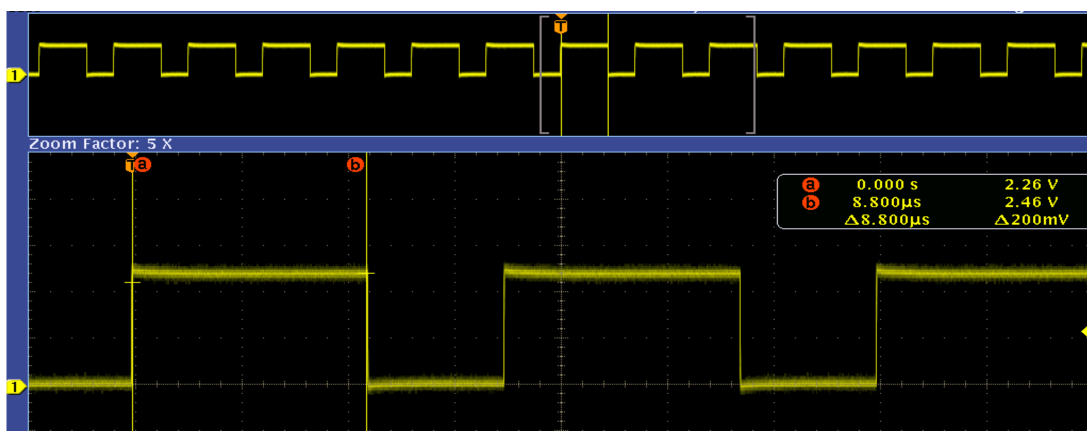



Abbildung 4.19.: Zeitmessung $\Delta t_1$ für Compiler Optimierungsstufen -O1, -O2, -O3, -Os mit SCLK = 12,5MHZ; 16 Datenbits; Das Zeitintervall der Low Phase entspricht die restliche Zeit der Timer Periode nach der Abtastung

| **Compiler Optimierungsstufe** | **Dauer** | $f_{max}$ |
|---|---|---|
| ohne Optimierung | 9,4 $\mu$s | $\approx$ 106KHZ |
| -O1 | 8,8 $\mu$s | $\approx$ 113KHZ |
| -O2 | 8,8 $\mu$s | $\approx$ 113KHZ |
| -O3 | 8,8 $\mu$s | $\approx$ 113KHZ |
| -Os | 8,8 $\mu$s | $\approx$ 113KHZ |

Tabelle 4.3.: Aus $\Delta t_1$ für die Compiler Optimierungsstufen -O1, -O2, -O3, Os ergibt sich die maximal realisierbare Abtastfrequenz eines SPI Bustransfers mit SCLK = 12,5 MHZ, 16 Datenbits, Systemtakt = 50 MHZ. Fixer Zeitanteil der Abtastzeit

Mit dem HP 3310B Funktionsgenerator wurde das Eingangssignal für den ADCS7476 in Form eines Sinus im Frequenzbereich von 1KHZ - 22KHZ generiert. Ein Timer startet mit einer Frequenz von 48KHZ (z.B. Audiosignal-Abtastung) einen SPI-Bustransfer zum Abtasten des analogen Wertes. Das digitalisierte Signal wird zur Rekonstruktion über den SPI-Bus an einen Digital-Analog-Umsetzter DAC121S101 von National gesendet. (vgl. Abbildung 4.20).

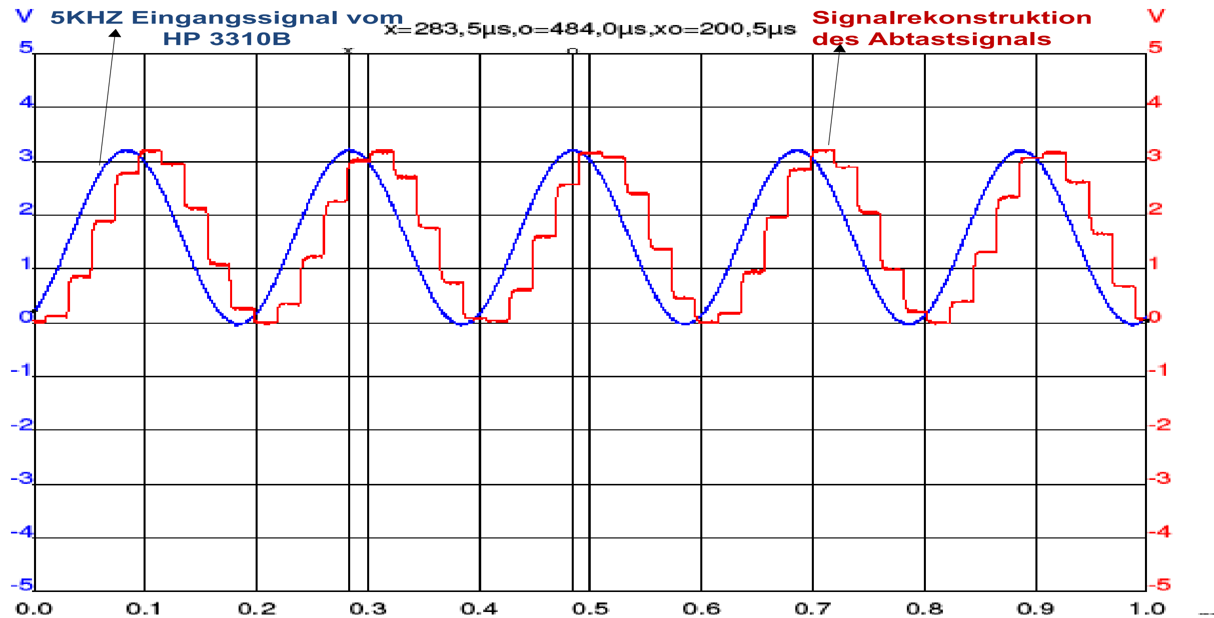



Abbildung 4.20.: Signalrekonstruktion des abgetasteten Signals

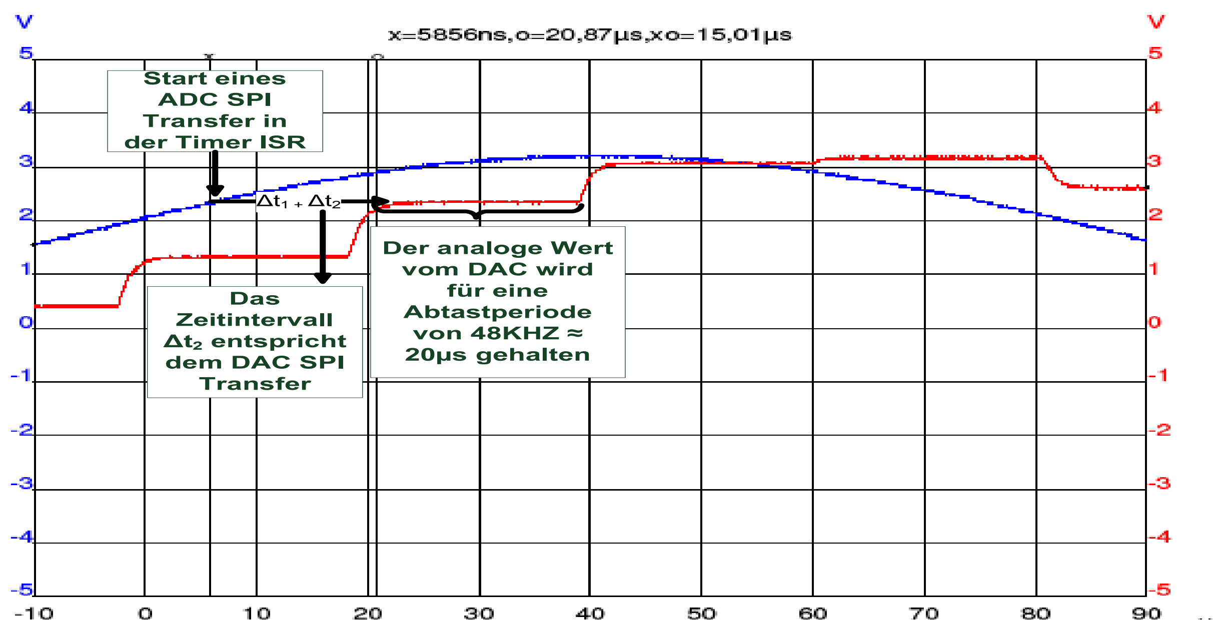



Abbildung 4.21.: Verzögerungszeit $\Delta t_1 + \Delta t_2$ zwischen Eingangs- und rekonstruiertem Ausgangssignal

# 5. Datenlogger und JTAG Programmierung über WLAN

In diesem Kapitel wird der Hardwareaufbau des Datenloggers vorgestellt (vgl. Abbildung 5.1). Für die Parametrisierung der einzelnen Systemkomponenten und die Messdatenerfassung ist eine Schnittstelle vom FPGA zu einem Host-PC erforderlich. Zur Analyse der Systemfunktionalitäten des Fahrzeugs müssen Messdaten während der Fahrt erfasst werden. Durch die Fernfeldtelemetrie werden dann die Messdaten vom Host-PC ausgewertet und zur Analyse bereitgestellt.

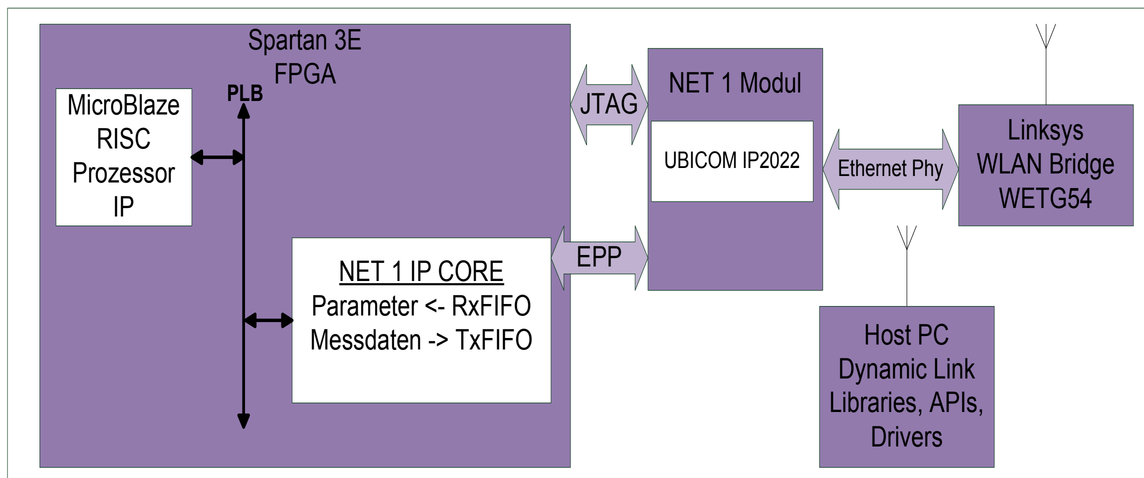



Abbildung 5.1.: Hardwareaufbau für die Funktion des Fernfeldtelemetriesystems, Parametrisierung von Systemkomponenten und JTAG Programmierung mit dem NET 1 Modul über eine WLAN Bridge

Das NET 1 Kommunikationsmodul von Digilent dient als Basis des Datenlogger-Hardware-Aufbaus. Über das NET 1 Modul wird die Verbindung für Datentransfers zwischen Host-PC und FPGA-Plattform hergestellt (vgl. Kapitel 5.1). Für die Realisierung der Datenloggerfunktionalitäten mit dem NET 1 Modul wurde das NET 1 IP CORE mit den folgenden Entwurfszielen erstellt:

- EPP Interface zum NET 1 Kommunikationsmodul
- FIFO Sende- / Empfang-FIFO
- PLB-Anbindung zum MicroBlaze Prozessor

Die erfassten Messwerte wie z.B. Abstände, Geschwindigkeit, Lenkwinkel usw. werden ins Tx-FIFO geschrieben, die kontinuierlich vom Host-PC ausgelesen werden. Die Parameter werden vom Host-PC ins RxFIFO geschrieben, die vom MicroBlaze ausgelesen und entsprechend einem spezifischen Protokollformat für die Systemkomponenten übernommen.

## 5.1. Digilent NET 1 Kommunikationsmodul

Das Digilent NET 1 Modul ist ausgestattet mit einem 10Mbit Ethernet-Port und einem UBICOM IP2022 $\mu$Controller (vgl. Abbildung 5.2). Die Firmware für den UBICOM IP2022 unterstützt den parallelen Datentransfer vom/zum Host-PC und der JTAG-Programmierung über eine Ethernet Verbindung mit den bereitgestellten Dynamic Link Libraries (DLL) und Treiber. Hierfür wurde das proprietäre Softwarepaket ADEPT SUITE mit folgenden Tools verwendet:

- **ExPort :** Mit dem Tool wird die FPGA-Konfigurationsdatei über die Ethernet-Verbindung an den UBICOM $\mu$Controller gesendet. Die Firmware des UBICOM $\mu$Controller programmiert das FPGA über die JTAG chain.
- **TransPort :** bietet zwei verschiedene Möglichkeiten zum Auslesen und Beschreiben der FIFO-Daten. Dies erfolgt über eine Maske oder File I/O.
- **Ethernet Administrator :** konfiguriert das NET 1 Modul für die oben beschriebenen Tools.

Die Firmware stellt die Kommunikation zwischen dem UBICOM Chip und dem FPGA über den Module Bus nach dem Enhanced Parallel Port (EPP) Protokoll (vgl. Abbildung **??**).

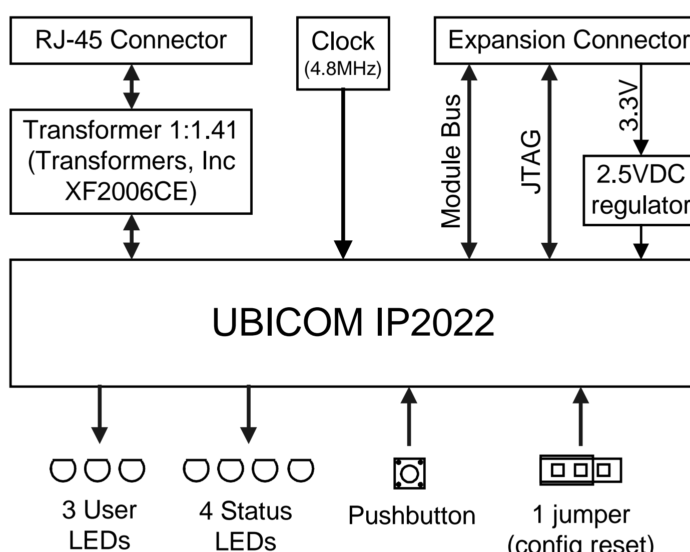



Abbildung 5.2.: UBICOM IP2022 $\mu$Controller [Digilent (2004a)]

## 5.2. Digilent Design Pattern nach dem EPP-Protokoll

Der Datenfluss zwischen FPGA und UBICOM $\mu$Controller erfolgt nach dem VHDL Design Pattern Parallel Interface Model (DPIM) von Digilent, das nach dem EPP-Protokoll ausgelegt ist [vgl. Digilent (2008)]. Mit dem Referenzmodell dpimref wird von Digilent die Anbindung des FPGA durch ein EPP Interface zu einem Digilent Peripherie-Produkt wie z.B. dem NET 1 Modul bereitgestellt. Nach diesem Modell erfolgt der Datentransfer zwischen einem Host-PC und Registern aus dem FPGA. Das DPIM unterstützt einen 8-Bit bidirektionalen Adress/Datenbus und sechs Steuerleitungen (vgl. Tabelle 5.1). Hieraus ergeben sich maximal $2^8$ = 256 adressierbare Register im FPGA mit einer Grösse von einem Byte die vom Host PC ausgelesen und beschrieben werden können.

| Name | Quelle | Beschreibung |
|---|---|---|
| DB0-DB7 | bidirektional | 8 Bit breiter bidirektionaler Daten- und Adressbus |
| WRITE | Host | Steuert die Richtung des Datenbusses : HIGH = Lesen (Host input, Peripheral output), LOW = Schreiben (Host output, Peripheral input) |
| ASTB | Host | Adress Strobe: zeigt an, dass eine Adresse ins Adress-Register geschrieben oder ausgelesen werden soll |
| DSTB | Host | Data Strobe: zeigt an, dass Daten ins Register mit der Adresse im Adress-Register geschrieben oder gelesen werden sollen. |
| WAIT | Peripheral | Das Wait-Signal dient der Synchronisation zwischen Host und Peripherie. Mit dem Wait-Signal wird die Bereitschaft der Peripherie für einen Buszyklus angezeigt. Falls nach einem Transferstart (WAIT = LOW) nicht innerhalb von 10ms WAIT von der DPIM Logik auf High gesetzt wird, so wird der Buszyklus abgebrochen. |
| INT | peripheral | Interrupt Request an den Host vom Peripheral |
| RESET | Host | Zum Zurücksetzten der DPIM-Logik auf dem FPGA. |

Tabelle 5.1.: Digilent Parallel Interface Model

Ein Zugriff auf ein Register erfolgt durch Adressierung vom Host-PC ins Adress-Register des DPIM und dem darauf folgenden Lesen/Schreiben der Daten, das in vier Buszyklen unterteilt ist. Die Buszyklen werden mit einem 2-Phasen-Handshake in Hardware synchronisiert. Der Transferstart wird in der ersten Handshakephase vom Host mit einem Request für einen Buszyklus durch Abfrage des WAIT-Signals iniitiert. Sobald das WAIT-Signal von der DPIM-Logik auf Low gesetzt wird, ist die erste Handshakephase beendet und der Buszyklus wird gestartet. Für die zweite Handshakephase wird vom Host mit dem Adress oder Data Strobe-Signal, das auf Low gesetzt wird, ein Request initiiert. Das DPIM hat 10ms Zeit, die Daten entweder auf den Bus zu legen oder abzuholen und den Request mit dem WAIT-Signal zu bestätigen, ansonsten wird der Buszyklus vom Host aus beendet.

### 5.2.1. Adress Write

Nach Abschluss der ersten Handshakephase, setzt der Host die Datenrichtung mit dem WRITE-Signal auf Low (vgl. Abbildung 5.3). Die neue Adresse wird vom Host an den Datenbus gelegt. Das Adress Strobe (ASTB) Signal wird auf Low gesetzt, sodass die zweite Handshakephase gestartet wird. Das DPIM holt die Adresse vom Bus und schreibt sie ins Adress-Register. Das DPIM setzt das WAIT-Signal auf High , sodass dem Host die Adressübernahme signalisiert wird. Das Beenden des Buszyklus vom Host wird gestartet. Der Host setzt als Reaktion das Adress-Strobe-Signal wieder auf High, sodass der Buszyklus des DPIM beendet wird.

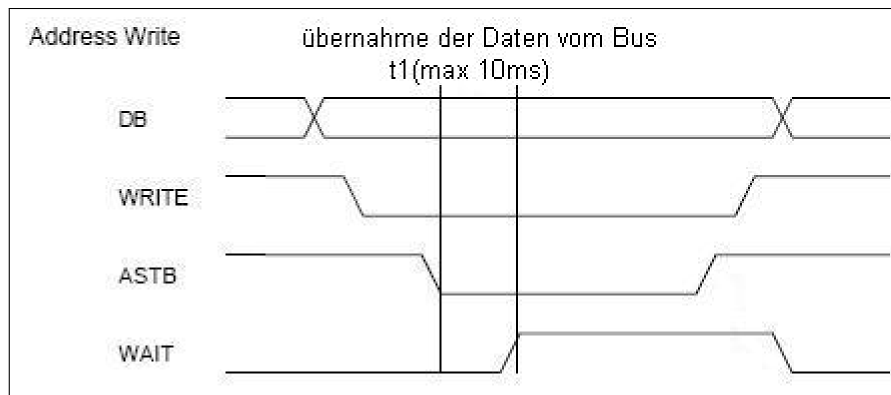



Abbildung 5.3.: Adress Write Buszyklus (Host output, Peripheral input) [Digilent (2008)]

### 5.2.2. Adress Read

Nach der ersten Handshakephase wird das WRITE-Signal auf High gesetzt, um einen Lesezyklus anzuzeigen. In der zweiten Handshakephase wird die Adresse aus dem Adress-Register des DPIM auf den Bus gelegt. Anschließend wird mit dem WAIT-Signal dem Host mitgeteilt, dass die Adresse am Bus zum Auslesen bereit liegt. Die zweite Handshakephase kann beendet werden (vgl. Abbildung 5.4).

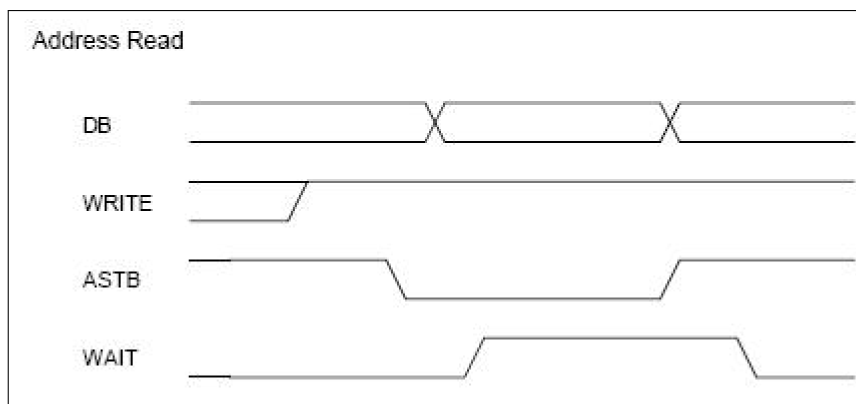



Abbildung 5.4.: Adress Read Buszyklus (Host input, Peripheral output) [Digilent (2008)]

### 5.2.3. Data Write

Für das Senden der Daten vom Host zum DPIM ist ein Data Write Buszyklus erforderlich (vgl. Abbildung 5.5). Die Daten werden in die Register mit der Adresse, die im Adress-Register steht, beschrieben.

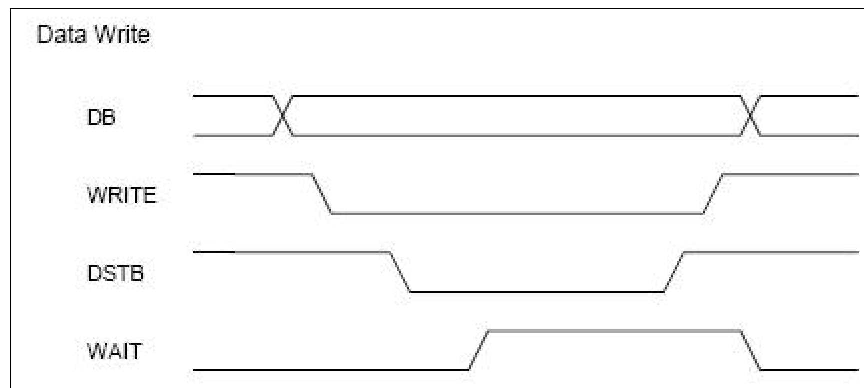



Abbildung 5.5.: Data Write Buszyklus (Host output, Peripheral input) [Digilent (2008)]

Nach der ersten Handshakephase setzt der Host die Datenrichtung. Die Daten werden auf den Bus gelegt und die zweite Handshakephase wird mit dem Data Strobe-Signal angeregt. Das Device erkennt den Data Strobe (DSTB), schreibt die Daten vom Bus ins adressierte Register und setzt das WAIT-Signal auf High. Die zweite Handshakephase kann durch den Host beendet werden.

### 5.2.4. Data Read

Für den Empfang der Daten des DPIM durch den Host ist ein Data Read Buszyklus erforderlich (vgl. Abbildung 5.6). Die Daten werden aus dem Register mit der Adresse, die im Adress-Register steht, ausgelesen und auf den Bus gelegt.

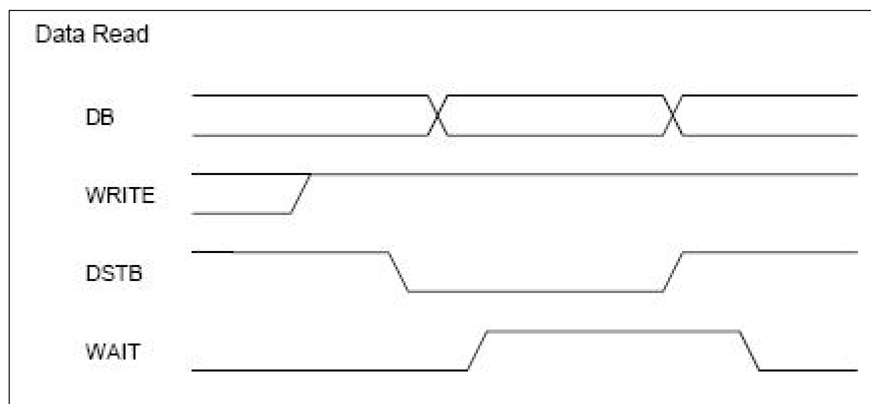



Abbildung 5.6.: Data Read Buszyklus (Host input, Peripheral output) [Digilent (2008)]

Nach der ersten Handshakephase setzt der Host die Datenrichtung. Das Device erkennt den Data Strobe, legt die Daten des adressierten Registers auf den Bus und setzt das WAIT-Signal auf High. Der Host liest die Daten ein und die zweite Handshakephase kann beendet werden.

## 5.3. NET 1 IP CORE

Das NET 1 IP CORE stellt die Verbindung zwischen MicroBlaze Prozessor und den NET 1 Modul her. Über den PLB greift der MicroBlaze auf die FIFOs des NET 1 IP COREs zu. Der EPP-Controller stellt die Verbindung zwischen NET 1 Modul und den FIFOs des NET 1 IP COREs.

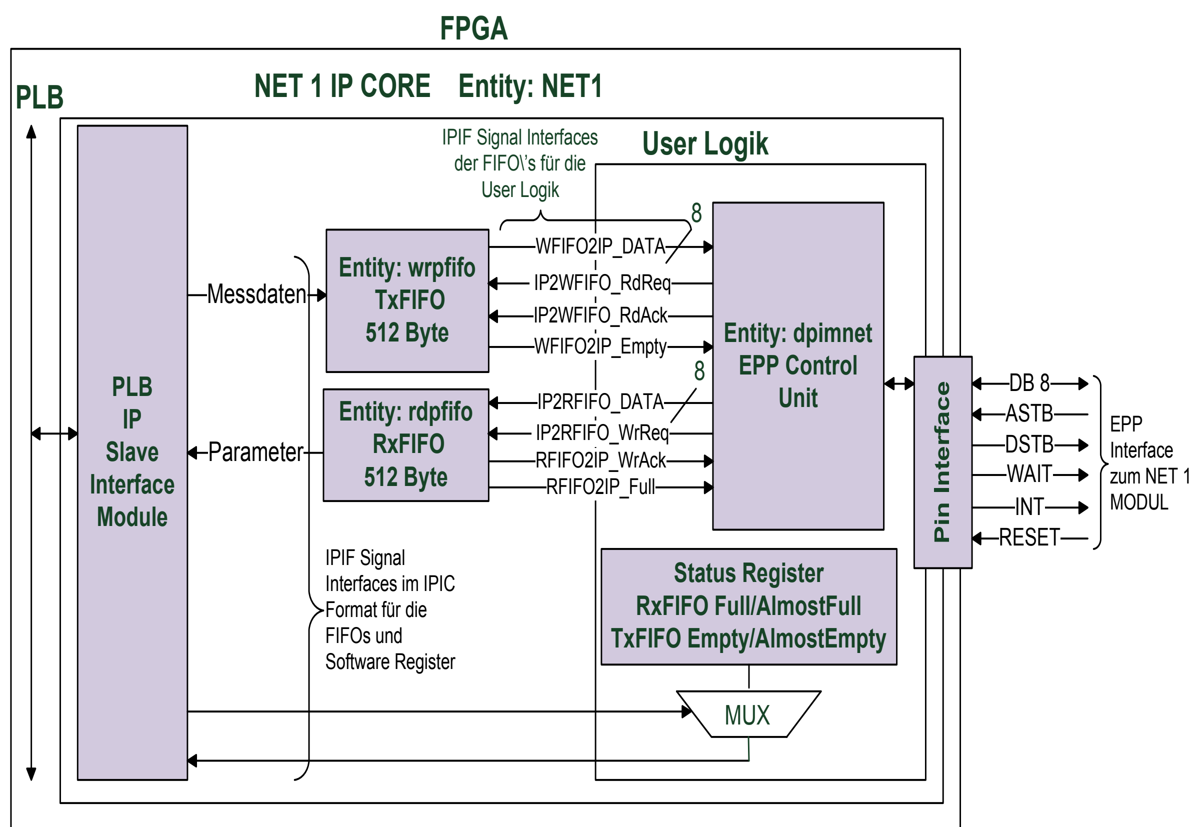



Abbildung 5.7.: Net 1 IP CORE Blockschaltbild

Mit dem Create Peripheral Tool wurden für das Net 1 IP CORE

1. ein PLB Slave Interface,
2. zwei PLB Slave IPIF FIFO Option und
3. ein Software-Register für Status-Informationen

selektiert. Aus dem Referenzmodell dpimref des EPP Controllers, wurde das dpimnet (vgl. Anhang C) für das NET 1 IP CORE erstellt und in der Anwenderlogik eingebunden. Mit dem Modell des dpimnet werden anstelle von Registern die Read- / Write-FIFOs verwendet.

### 5.3.1. FSM Schaltbild und Zustandsdiagramm des EPP Controllers

Das Automatenmodell des dpimnet entspricht einem Medwedew-Automaten (vgl. Abbildung 5.8). Ein Medwedew-Automat ist ein Moore-Automat ohne Ausgangschaltnetz. Die Ausgänge werden entsprechend der Zustands-Flip-Flops geschaltet.

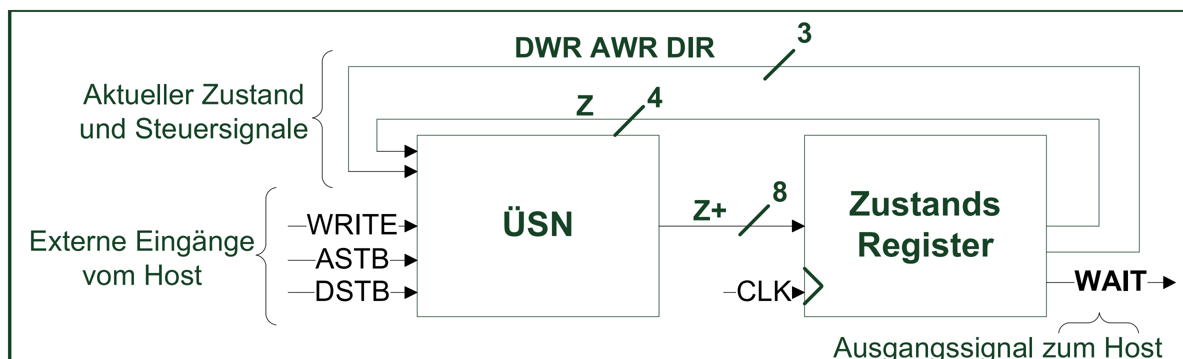



Abbildung 5.8.: Medwedew-Automatenmodell des dpimnet Controllers

Der dpimnet Medwedew-Automat besteht aus 9 (vgl. Abbildung 5.9). Für die Kodierung der Zustände werden 8 Bit verwendet. Das erste Nibble dient der Kodierung der Zustände. Das zweite Nibble setzt sich zusammen aus den drei Steuersignalen DWR, AWR, DIR und dem Ausgangssignal WAIT. Die Steuersignale DWR und AWR sind Enable-Eingänge der D-Flip-Flop Register für Daten und Adresse. Durch die Modifizierung entspricht das Steuersignal DWR einem Request für einen FIFO-Eintrag. Der bidirektionale Bus mit dem Tri-State-Treiber erfordert die sichere Steuerung, das nur einer zur Zeit den Bus treibt. Der Host gibt die Datenrichtung vor und der Automat des dpimnet muss sicherstellen, dass der Ausgang nur getrieben wird, wenn das Write Signal auf High ist. Das Steuersignal DIR wird mit dem Eingangssignal WRITE logisch nand verknüpft und als Enable-Signal für die Tri-State-Treiber verwendet (vgl. Tabelle 5.3.1). Dadurch müssen beide Signale logisch 1 sein, damit der Ausgang den hochohmigen Zustand verlässt.

| DIR | WRITE | Tri-State enable | Data Bus |
|---|---|---|---|
| 0 | 0 | 1 | Z |
| 0 | 1 | 1 | Z |
| 1 | 0 | 1 | Z |
| 1 | 1 | 0 | 0 / 1 |

Tabelle 5.2.: Wahrheitstabelle für Tri-State-Treiber

Das Steuersignal DIR kann nur logisch 1 werden wenn eine Zustandstransition stattgefunden hat, bei dem das Eingangssignal WRITE = 1 war und ein Lesezyklus durch ein Adress- oder Data-Strobe initiiert wurde. Durch das Steuersignal DIR ist vom Automaten sichergestellt, dass der Ausgang nur den hochohmigen Zustand verlässt, wenn durch das Write-Signal und ein Strobe Signal ein Zustandswechsel angeregt wurde. Dieses Verhalten stellt sicher, dass der Datenbus-Ausgang nicht nur durch das WRITE-Signal (Datenrichtung) gesteuert wird, sondern durch Automatenzustände.

- ***READY:*** Der Automat ist bereit, einen von vier Buszyklen zu starten, das mit dem WAIT Signal = 0 signalisiert wird. Die Tri-State-Treiber sind aktiviert. Die Steuersignale DWR, AWR, DIR sind auf Low gesetzt.
- ***AwrA:*** Ein Adress-Schreibzyklus wurde gestartet durch eine vorangegangene Transition mit den Eingangssignalwerten WRITE = 0 und ASTB = 0 waren. Das enable Signal AWR aus den Zustandbits bewirkt die Übernahmne der Adresse aus dem Bus in das Adress-Register. Die Tri-State-Treiber sind aktiviert. Die Transition in den AwrB Zustand folgt im nächsten Takt.

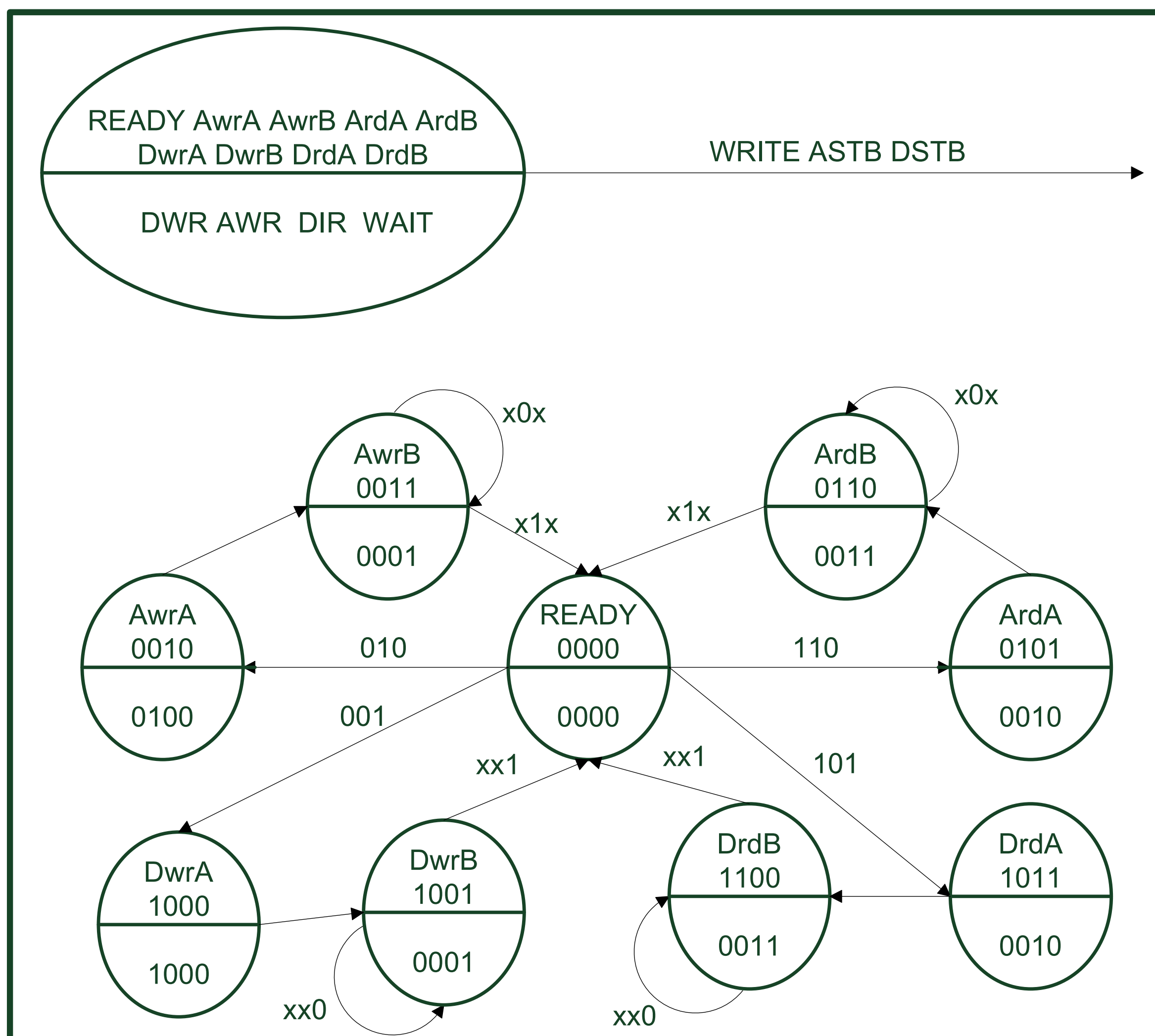



Abbildung 5.9.: Zustandsdiagramm des dpimnet Controllers

- ***AwrB:*** Die Adresse aus dem Adress-Register wird verwendet. Das Ausgangssignal ist auf High, um den Host zu signalisieren, dass die Adresse vom Bus übernommen wurde. Es wird solange in diesem Zustand gewartet, bis der Host die Transaktionen mit dem Setzten des Adress-Strobe bestätigt und eine Transition in den Ready-Zustand folgt.
- ***ArdA:*** Ein Adress-Lesezyklus wurde durch eine vorangegangene Transition mit den Eingangssignalwerten WRITE = 1 und ASTB = 0 gestartet. Durch das Steuersignal DIR verlässt der Ausgang den hochohmigen Zustand und die Adresse im Adress-Register wird auf den Bus gelegt. Die Transition in den ArdB-Zustand folgt im nächsten Takt.
- ***ArdB:*** Mit dem Setzen des Wait-Signals wird dem Host signalisiert, dass die Daten am Bus bereitliegen. Der Automat wartet solange, bis der Host die Daten hat und die Transition in den Ready-Zustand folgt.
- ***DwrA:*** Ein Daten Schreibzyklus wurde durch eine vorangegangene Transition mit den Eingangssignalwerten WRITE = 0 und DSTB = 0 gestartet. Der Datenbusausgang bleibt weiterhin im hochohmigen Zustand bedingt durch DIR = 0 und WRITE = 0. Ein Request-Signal wird für ein im Adress-Register adressiertes FIFO gesetzt, um die Übernahme der Daten zu initiieren. Es folgt eine Transition in den DwrB-Zustand im nächsten Takt.

- ***DwrB:*** Der Automat signalisiert mit dem WAIT-Signal, dass die Daten übernommen worden sind und wartet solange, bis der Host ihm die Transaktion mit dem Setzten des DSTB quittiert.
- ***DrdA:*** Ein Daten-Lesezyklus wurde durch eine vorangegangene Transition mit den Eingangssignalwerten WRITE = 1 und DSTB = 0 gestartet. Der Datenbusausgang verlässt den hochohmigen Zustand und ein Request-Signal an das adressierte FIFO wird gesetzt. Das FIFO legt die Daten auf den Bus und eine Transition den DrdB-Zustand folgt.
- ***DrdB:*** Mit dem Setzten des Wait-Signals wird dem Host signalisiert, dass die Daten am Bus bereitliegen. Der Automat wartet solange, bis der Host die Daten hat und die Transition in den Ready-Zustand folgt.

### 5.3.2. Write und Read Hardware IPIF FIFO

Die Write und Read IPIF FIFOs wurden mit folgenden Generics für das NET 1 IP CORE parametrisiert:

| GENERIC | Wert | Beschreibung |
|---|---|---|
| C_OPB_PROTOCOL | 0 | 0 = FIFO ist am PLB IPIF verbunden. 1 = FIFO ist am OPB IPIF verbunden. |
| C_MIR_ENABLE | 0 | Aktivieren des Module ID-Registers |
| C_BLOCK_ID | 0 | Wert des Module ID |
| C_NUM_REG_CE | 4 | Anzahl interner Register des FIFOs für Reset, Status, Push und Write-Daten. |
| C_FIFO_DEPTH_LOG2X | 9 | Grösse des FIFOs als Exponent zur Basis 2 |
| C_FIFO_WIDTH | 8 | FIFO-Datenbreite |
| C_INCLUDE_PACKET_MODE | 0 | 0 = Lese- / Schreib-Transaktion werden von der User IP einmalig vorgenommen. 1 = Die Packet-Funktionalität ermöglicht der User IP eine Wiederholung der Lese- / Schreib-Transaktion. |
| C_INCLUDE_VACANCY | 1 | Aktiviert die Berechnung für die Anzahl der freien Speicherzellen. |
| C_INCLUDE_DRE | 0 | Data Realignment Engine für Power PC Prozessoren |

Tabelle 5.3.: Parametrisierung der Write und Read FIFOs

Bei einer FIFO-Tiefe bis 16 Elementen und keiner Packet Mode-Funktionalität werden die Daten in den SRL16 (Schieberegister LUTs) gespeichert. Bei jeder anderen Konfiguration werden die Daten in den internen Dual Port BLOCK RAM-Modulen gespeichert.

#### Write FIFO

Ein Write FIFO hat als Read Interface die User IP und als Write Interface den PLB Slave (vgl. Abbildung 5.10).

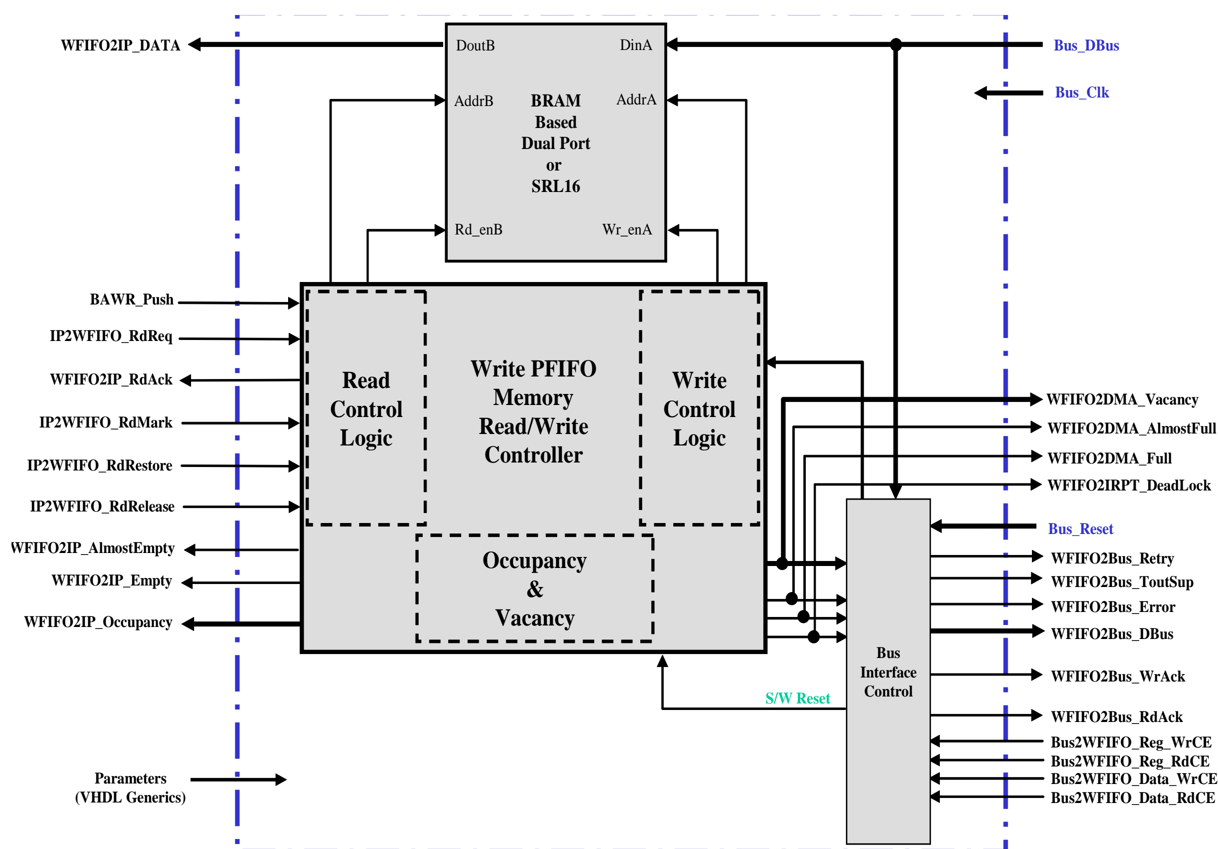



Abbildung 5.10.: Write FIFO Block Diagramm [Xilinx (2006b)]

Das dpimnet in der User-Logic wurde mit folgenden IPIF Signal Interfaces der Read Control-Logic verbunden:

| Signalname | Beschreibung |
|---|---|
| WFIFO2IP_DATA | 8-Bit Daten, die vom DPIM ausgelesen werden. |
| IP2WFIFO_RdReq | Das DPIM setzt das Steuersignal für eine Anfrage des nächsten FIFO-Wertes, falls das FIFO nicht leer ist. |
| IP2WFIFO_RdAck | Die Bestätigung für die Anfrage und das Bereitstellen der FIFO Daten im nächsten Takt. |
| WFIFO2IP_Empty | Signalisiert dem DPIM, dass keine Daten da sind. |

Tabelle 5.4.: IPIF Signal Interfaces zwischen der User-Logic und Read Control-Logic des Write FIFO

Das Beschreiben der Write FIFO erfolgt über den PLB mit einem Register-Offset zur Basisadresse des Net 1 IP CORE. In der Datei *net.h* befinden sich die vom EDK generierten Konstanten für Adressen, Bitmasken und Makrofunktion für die Zugriffe auf die Status- und Daten-Register des Write FIFO. Zum Auslesen der Write FIFO mit TransPort vom Host PC wurde folgender Adressbereich gewählt :

| Beschreibung | Read/Write | Adresse |
|---|---|---|
| FIFO Daten | Read | 0x00 |
| Write FIFO Occupancy 1. Byte | Read | 0x01 |
| Write FIFO Occupancy 2. Byte | Read | 0x02 |

Tabelle 5.5.: Adressraum des Write FIFO für TransPort vom Host-PC

### Read FIFO

Ein Read FIFO hat als Read Interface den PLB Slave und als Write Interface die User IP (vgl. Abbildung 5.11).

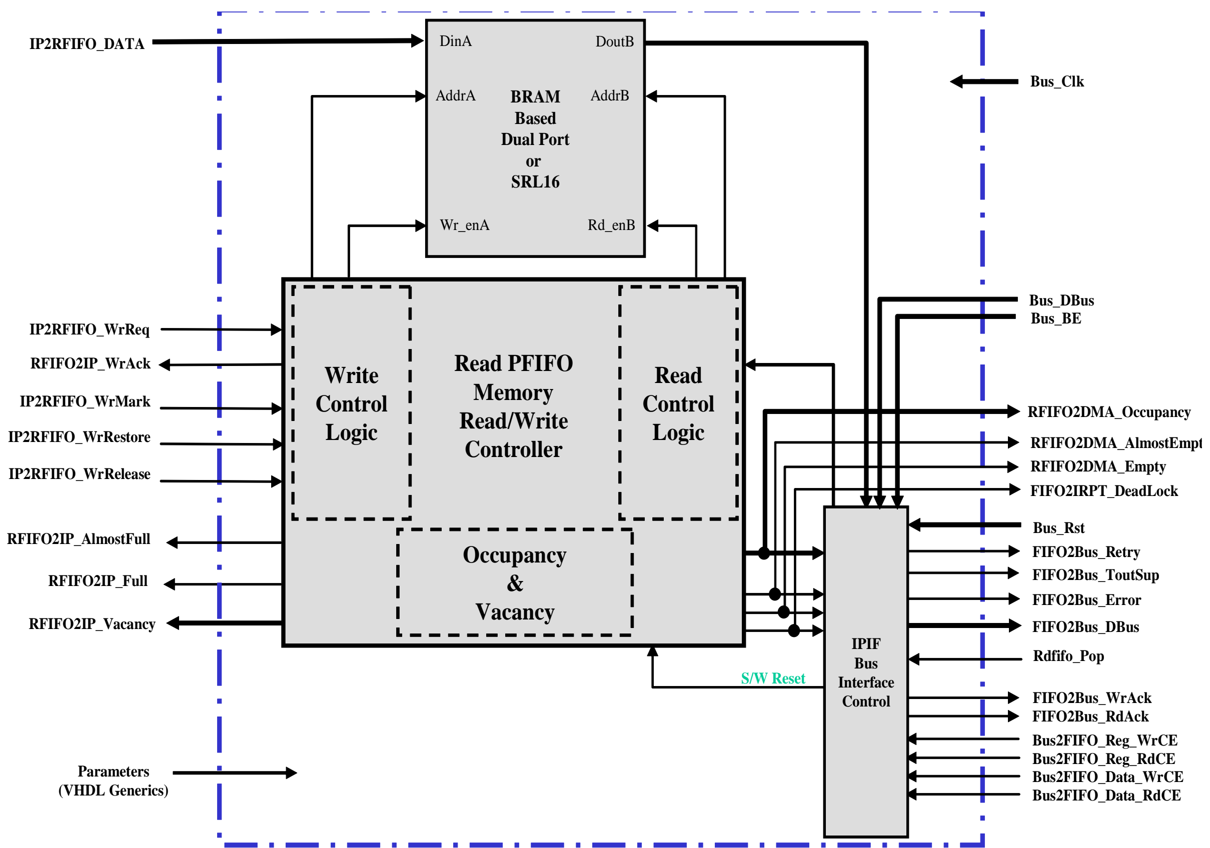



Abbildung 5.11.: Read FIFO Block Diagramm [Xilinx (2006a)]

Das dpimnet in der User-Logic wurde mit folgenden IPIF Signal Interfaces der Write Control-Logic verbunden:

| Signalname | Beschreibung |
|---|---|
| IP2RFIFO_DATA | 8-Bit Daten, die vom DPIM ins FIFO geschrieben werden. |
| IP2RFIFO_WrReq | Das DPIM setzt das Steuersignal für eine Anfrage zum Schreiben des nächsten FIFO Wertes, falls das FIFO nicht voll ist. |
| RFIFO2IP_WrAck | Die Bestätigung für die Anfrage und Übernahme der FIFO Daten im nächsten Takt. |
| RFIFO2IP_Full | Signalisiert dem DPIM, dass das FIFO voll ist. |

Tabelle 5.6.: IPIF Signal Interfaces zwischen der User-Logic und Write Control-Logic des Read FIFO

Das Auslesen der Read FIFO über den PLB erfolgt wie beim Write FIFO. Für das Beschreiben der Read FIFO mit TransPort vom Host PC wurde folgender Adressbereich gewählt:

| Beschreibung | Read/Write | Adresse |
|---|---|---|
| FIFO Daten | Write | 0x03 |
| Read FIFO Vacancy 1.Byte | Read | 0x04 |
| Write FIFO Vacancy 2.Byte | Read | 0x05 |

Tabelle 5.7.: Adressraum des Read FIFO für TransPort vom Host PC

## 5.4. Linksys WET54G W-LAN Bridge

Für die drahtlose Datenübertragung wurde das Ethernet Port des Net1-Moduls mit dem Linksys WET54G W-LAN Bridge verbunden. Die korrekte Verbindung zwischen Host-PC und NET 1 Modul erfordert die Konfiguration der W-LAN Bridge.

Abbildung 5.12.: Linksys WET54G W-LAN Bridge

Eigenschaften des WET54G:

- **Standards :** IEEE 802.11g, IEEE 802.11b, IEEE 802.3, IEEE802.3u
- **Ports :** 1 $*$ 10/100 Auto-Crossover (MDI/MDI-X) Port, 1 $*$ Power Port
- **Security Features :** WPA, 64/128-bit WEP Encryption
- **Protocols :** 802.11b: CCK (11Mbps), CCK (5.5Mbps), DQPSK (2Mbps), DBPSK (1Mbps) 802.11g: OFDM (54Mbps)

Die bereitgestellten DLL und Adept Treiber-Software von Digilent erkennt das Modul anhand von drei Kriterien:

- dem Netzwerk-Device-Namen *DModNet1*
- fixe IP-Adresse *192.168.0.2/24*
- MAC-Adresse

Konfiguration des WET54G :

1. Anhand des Netzwerknamen (SSID) und Passworts (WEP/WPA) wird festgelegt, mit welchem W-LAN-Netzwerk sich die Bridge nach einem Systemstart verbindet. Hierfür wurde ein privates Netzwerk aufgebaut, an dem der Host-PC und die Bridge mit folgenden Eigenschaften verbunden sind :
    - Dynamic Host Configuration Protocol (DHCP) aktiviert
    - IP-Bereich : 192.168.0.2 - 192.168.0.10
    - default Gateway : 192.168.0.1
    - SSID : SoCWLAN
2. Der Netzwerkname und die MAC-Adresse vom NET 1 Modul werden von der Bridge übernommen.

# 6. FreeRTOS Echtzeitbetriebssystem

In diesem Kapitel wird die Portierung des Echtzeitbetriebssystems FreeRTOS auf die SoC Hardwareplattform vorgestellt. FreeRTOS stellt eine Laufzeitumgebung für zeitkritische Software-Anwendung bereit. Durch das Task-System werden die Abläufe mit einem deterministischen Laufzeitverhalten quasiparallel gesteuert.

FreeRTOS ist ein Open Source Echtzeitbetriebssystem mit einem minimalen Kernel, das zur Portierung auf verschiedene Hardwarearchitekturen zur Verfügung steht [FreeRTOS]. Für die Portierung auf die SoC-Hardwareplattform mit einem

- Nexys 2 Board,
- Spartan 3E FPGA und
- MicroBlaze Prozessor

ist das Einbinden des MicroBlaze Port für das Nexys 2 Board erforderlich. Die Struktur von FreeRTOS und die Portierung werden im Folgenden weiter erläutert.

## 6.1. Dateistruktur des FreeRTOS

Die Source Code Organisation von FreeRTOS ist in den zwei relevanten Verzeichnissen *DEMO* und *SOURCE* strukturiert (vgl. Abbildung 6.1).

```
FreeRTOS
    |
    +-Demo
    |   |
    |   +-Common    The demo application files that are used by all the ports.
    |   +-Dir x     The demo application build files for port x
    |   +-Dir y     The demo application build files for port y
    |
    +-Source           Contains all directories associated with the scheduler source code
       |
       |               3 core scheduler files common to all ports (4 is using co-routines)
       |
        +-include                 Scheduler header files
        |
        +-portable                Scheduler port layer for all ports
            |
            +-MemMang                 Sample memory allocators can be used for all ports
            |
            |
            +-GCC                     Scheduler port layer for ports using GCC compiler
            |   +-ATmega32                Scheduler port files for AVR using GCC compiler
            |   +-MSP430F449              Scheduler port files for MSP430 using GCC compiler
            |   +-ARM7_LPC2000            Scheduler port files for LPC2106 using GCC compiler
            |   +-ARM7_AT91FR40008        Scheduler port files for AT91 using GCC compiler
            |   +-H8S2329                 Scheduler port files for H8/S using GCC compiler
            |   +-Microblaze              Scheduler port files for Microblaze using GCC compiler
            |   +-ARM_MC3                 Scheduler port files for ARM Cortex-M3 using GCC compile
            |   +-AVR32_UC3               Scheduler port files for AVR32 AT32UC3A using GCC compil
```

Abbildung 6.1.: Auszug aus der Verzeichnisstruktur des FreeRTOS [FreeRTOS]

Im Demo-Ordner befinden sich jeweils für vorhandene Hardwarearchitekturen eine Beispiel-Anwendung. Für diesen Ordner wurde im Rahmen dieser Arbeit zusätzlich mit dem EDK für ein MicroBlaze System mit einem Nexys 2 Board und einem Spartan 3E FPGA eine Demoanwendung erstellt, das in Kapitel 6.6 weiter erläutert wird.

Im Ordner *Source* befinden sich die folgenden Dateien, die den Kern des Schedulers bilden :

- *croutine.c*
- *queue.c*
- *task.c*
- *list.c*

An den Dateien wurde keine Modifikation vorgenommen, da diese Dateien für alle Ports gleich sind. Die zugehörigen Deklarationen sind im include-Ordner. Im Portable-Ordner sind die für den Scheduler erforderlichen Port-spezifischen Dateien (vgl. Abbildung 6.1). Für die MicroBlaze Plattform erforderliche Port Dateien befinden sich im Ordner *Microblaze*, die sowohl C-, als auch Assembler Code enthalten.

- *port.c*
- *portmacro.c*
- *portasm.s*

Im Ordner *MemMang* werden Standard-Varianten der Speicherverwaltung zur Verfügung gestellt, die aber nicht zwingend verwendet werden müssen.

- *heap_1.c*
- *heap_2.c*
- *heap_3.c*

Die Datei *FreeRTOSConfig.h* ist in jedem XPS-Software-Projekt einzubinden, um die FreeRTOS Konfiguration zu erstellen. Für die Integration von FreeRTOS in ein XPS-Projektverzeichnis müssen folgende Dateien eingebunden werden :

- *croutine.c*
- *queue.c*
- *task.c*
- *list.c*
- *port.c*
- *portmacro.c*
- *portasm.s*
- *heap_1.c* oder *heap_2.c* und optional *heap_3.c*
- *FreeRTOSConfig.h*

Dies erfolgt durch eine relative Pfadangabe der Datein zum Projektverzeichnis in den Compiler-Optionen *Paths and Options* in den Feldern *Library(-L)* und *Include(-I)*.

## 6.2. Kernel

Bei nicht-Echtzeitbetriebssystemen liegen die Anforderungen des Schedulings darin, allen aktiven Tasks möglichst gleiche Zeitscheiben für die Ausführung zuzuteilen. Die Zielsetzung bei Echtzeitbetriebssystemen liegt darin, die Echtzeitanforderung der realen Welt einzuhalten und zu garantieren, dass die vorgegebenen Zeiten auf Ereignisse eingehalten werden.

### 6.2.1. Scheduler

Der Scheduler ist eine Hauptkomponente des FreeRTOS-Kernels, der je nach Konfiguration ein kooperatives- oder präemptives Multitasking-Konzept realisiert. Beim kooperativen Multitasking werden alle Tasks in eine Liste eingetragen, bei dem der aktive Task selbst bestimmt, wann er die Rechenzeit an den nächsten Task in der Liste übergibt. Die Schwäche dieses Konzepz liegt in der System-Stagnation, die auftreten kann, sobald ein Task die Kooperation abbricht. Beim präemptiven Konzept steuert der Kern die Rechenzeitvergabe. Die Prozessorvergabe für die Tasks erfolt durch Prioritäten. Bei äquivalenter Priorität und Ready-Zustand der Tasks wird die gleiche Zeitscheibe vom Scheduler vergeben. Blockierte oder Suspendierte Tasks stehen dem Scheduler nicht zur Verfügung. Ein Task kann entweder durch

- Preämption vom Kernel,
- sich selbst mit einem Delay von einer festen Zeit,
- warten auf eine Ressource und
- ein Event

in einer dieser zwei Zustände bringen.

### 6.2.2. RTOS Tick

Durch einen Timer wird die Frequenz für den Tick-Interrupt im *FreeRTOSConfig.h* gesetzt. In der ISR des Timers wird die Tickvariable inkrementiert, wodurch die Zeitmessung erfolgt. Zusätzlich muss überprüft werden, ob eine höher priorisierter Task im Running-Zustand ist oder aus dem Blocked- / Sleeped-Zustand durch ein internes oder externes Event in den Ready-Zustand gebracht werden muss, um ein Kontextwechsel durchzuführen.

### 6.2.3. Speicherverwaltung

Für die Speicherverwaltung werden zwei Varianten und ein Standard-Wrapper für die dynamische Speicherverwaltung während der Laufzeit vom FreeRTOS bereitgestellt.

1. In dieser Variante wird der Speicher, der vergeben wurde, nicht mehr freigegeben. Der deterministische Algorithmus teilt ein Array in kleinere Speicherblöcke entsprechend der Datenbreite des Ram-Moduls. Die Grösse des Arrays wird in FreeRTOSConfig.h mit dem Makro TOTAL_HEAP_SIZE gesetzt. Diese Lösung ist für Anwendungen geeignet, die keine Tasks und Queues zur Laufzeit erstellen oder löschen.

2. Implementiert nach dem Best Fit Algorithmus und ermöglicht schon allokierten Speicher wieder freizugeben. Die Kombination von benachbarten Speicherblöcken in einen grösseren Block ist nicht möglich. Diese Lösung bietet die Möglichkeit Tasks und Queues dynamisch zur Laufzeit zu erstellen und zu löschen.
   Durch die Speicherfunktion *pvPortMalloc()* und *vPortFree()*, die aufgerufen werden, würde es bei verschiedenen Stackgrössen der Tasks und der Tiefe der Queues zu einem Speicher-Fragmentierungsproblem führen. Der Algorithmus ist nicht deterministisch.

3. Die Variante ist ein Wrapper für die Standard Speicherfunktion *malloc()* und *free()*, das Thread Safe ist.

## 6.3. FreeRTOS Task

Ein Task kann sich in einem von vier Zuständen befinden (vgl. Abbildung 6.2).

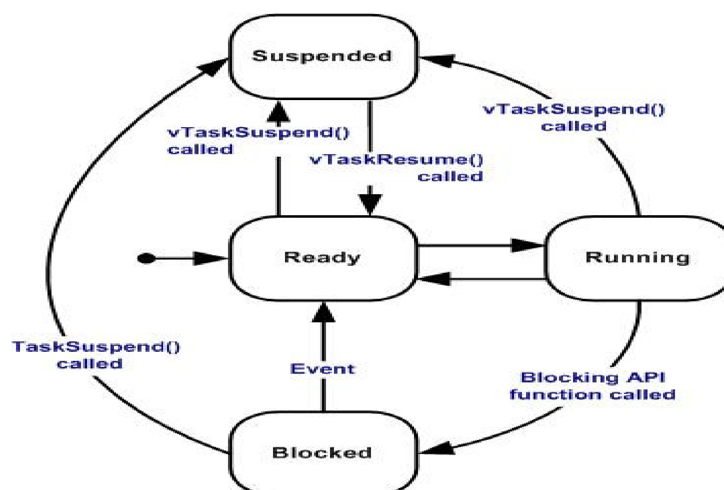



Abbildung 6.2.: FreeRTOS Task-Zustände und die verwendete API für die Zustandswechsel.[FreeRTOS]

Jeder Task besitz seinen eigenen Stack und wird in seinem eigenen Kontext ausgeführt, das ohne jede Abhängigkeit zu anderen Tasks im System oder zum Scheduler besteht. Der Idle-Task wird vom FreeRTOS Kernel erstellt und ausgeführt, sobald kein Task zum Ausführen bereit ist. Der Idle-Task führt die Freigabe der Ressourcen der gelöschten Tasks aus. Zusätzlich kann der Idle Task eine Hook-Funktion enthalten, die eine Callback-Funktion ist und im Idle-Task zyklisch aufgerufen wird, falls kein Task aktiv ist. Die Anwendung muss mit der in der API bereitgestellten Funktion *vTaskDelete()* sicherstellen, dass auch jeder Task gelöscht wird. Jeder Task bekommt eine Priorität zwischen 0 und dem festgelegtem Maximalwert in *FreeRTOSConfig.h* zugewiesen.

## 6.4. FreeRTOS Co-Routines

Eine Co-Routine kann sich in einem von drei Zustände befinden (vgl. Abbildung 6.3).

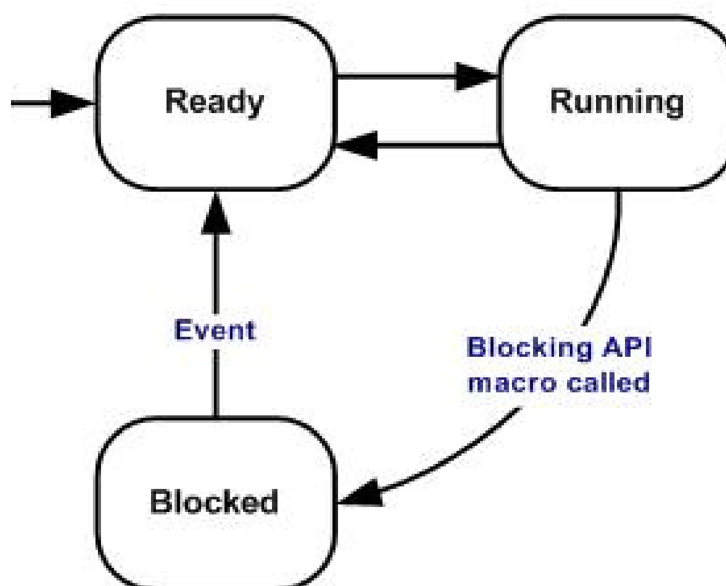



Abbildung 6.3.: FreeRTOS Co-Routines Zustände [FreeRTOS]

Jede Co-Routine bekommt eine Priorität zwischen 0 und dem festgelegtem Wert in *FreeRTOS-Config.h*, die nur für Co-Routinen gilt. Ein Task wird einer Co-Routine immer vorgezogen. Für den Ein- und Austritt aus der Co-Routine-Funktion müssen die in der API bereitgestellten Funktionen *crSTART( xHandle )* und *crEND()* aufgerufen werden. Das Scheduling von Co-Routines erfolgt durch wiederholtes Aufrufen der Funktion *vCoRoutineSchedule()*. In einer Anwendung, die ausschließlich aus Co-Routines besteht, ist der optimale Funktionsaufruf innerhalb der Idle-Hook-Funktion. Dadurch existieren neben dem Task keine weiteren Tasks, sodass sämtliche Co-Routines scheduled werden. Co-Routines besitzen keinen eigenen Stack und die Erhaltung der Variablen muss mit dem keyword *static* sichergestellt werden.

## 6.5. Interprozesskommunikation und Synchronisation

Die Interprozesskommunikation im FreeRTOS erfolgt durch MessageQueues, die zwischen Tasks oder Interrupts und Tasks erfolgen können. Die Queue-Zugriffe sind durch ein internes Mutex gesichert, sodass eine Synchronisation durch die Anwendung nicht notwendig ist. Beim Empfang der Daten aus der Queue kann eine maximale Zeit übergeben werden, für die der Task in den blockierten Zustand geht, falls die Queue leer ist. Werden in dieser Zeit keine Daten empfangen, liefert die Funktion eine Fehlermeldung zurück. Warten mehrere Tasks und Daten sind Verfügbar, so verlässt der höchst-priorisierte Task den Blocked-Zustand. Analog funktioniert das Senden von Daten in die Queue.

FreeRTOS bietet fertige Implementierungen von Synchronisationsmechanismen wie Binäre- und zählende Semaphore und Mutexe, die optional noch rekursiv sein können. Beim rekursiven

Mutex kann der Task, der den Mutex erhalten hat, mehrmals einen Request Aufruf auf den gleichen Mutex machen und muss in der gleichen Anzahl die Freigabe erteilen, damit ein anderer Task den Mutex erhält.

## 6.6. FreeRTOS Demoanwendung

Für die Portierung des FreeRTOS auf die MicroBlaze Plattform wurde eine Demoanwendung erstellt. Die Anwendung soll die korrekte Arbeitsweise des Schedulers in den Funktionen *Kontextwechsel* und *Zeitscheibenvergabe* der Tasks demonstrieren. Die in Kapitel 6.1 aufgelisteten Dateien für den Kern und MicroBlaze Port wurden in das *freer* XPS-Software Projekt eingebunden. Für die Speicherverwaltung wurde die erste Variante gewählt, da keine Task/Queue während der Laufzeit in der Demoanwendung erzeugt wird.
Für die Demoanwendung wurden fünf Tasks mit Prioritäten zwischen 1 - 3 für die folgenden Abläufe implementiert (vgl. Anhang B.2) :

- **Read FIFO :** Dieser Task wird alle 100ms mit einer Priorität von 3 gestartet. Wenn das FIFO nicht leer ist, wird der Wert ausgelesen und auf dem Hyper Terminal angezeigt.
- **HAW SPI und Write FIFO :** Diese Task wird alle 300ms mit einer Priorität von 3 für einen HAW SPI Bustransfer gestartet. Die erfassten Messdaten werden in das Write FIFO geschrieben, falls dieses nicht voll ist.
- **LED :** Mit einer Priorität von 2 werden alle 50ms die LEDs mit einem neuen Muster aktualisiert.
- **7 Segment :** Aktualisiert alle 250ms mit einer Priorität von 2 die 7-Segment Anzeige.
- **Printer :** Alle 50ms wird mit der Priorität 1 erfolgt ein Print auf dem Hyper Terminal.

Die Konfiguration des FreeRTOS wird in der *FreeRTOSConfig.h* eingestellt. Für die Demoanwendung wurde folgende Konfiguration vom FreeRTOS gewählt :

| Macro | Wert | Beschreibung |
|---|---|---|
| configUSE_PREEMPTION | 1 | Für ein Präemptives Multitasking-Scheduling |
| configUSE_IDLE_HOOK | 0 | Zum Aktivieren/Deaktivieren der Callback-Funktion Idle-Hook, wird aufgerufen falls kein Task zum Ausführen bereit ist. |
| configUSE_TICK_HOOK | 0 | Die Callback-Funktion wird nach jedem Tick-Interrupt aufgerufen. |
| configCPU_CLOCK_HZ | 50000000 | Ein Makro für die CPU-Frequenz, das zur Berechnung der Timerperiode genutzt wird. |
| configTICK_RATE_HZ | 50000 | Ein Makro für die Frequenz des Tick-Interrupts, dass zur Berechnung der Timerperiode genutzt wird. |
| configMAX_PRIORITIES | 4 | Setzt die höchste Priorität für ein Task fest. |
| configUSE_TRACE_FACILITY | 1 | Makro zur Beobachtung und Aufzeichnung des Laufzeitverhaltens der Tasks vom Kernel. |

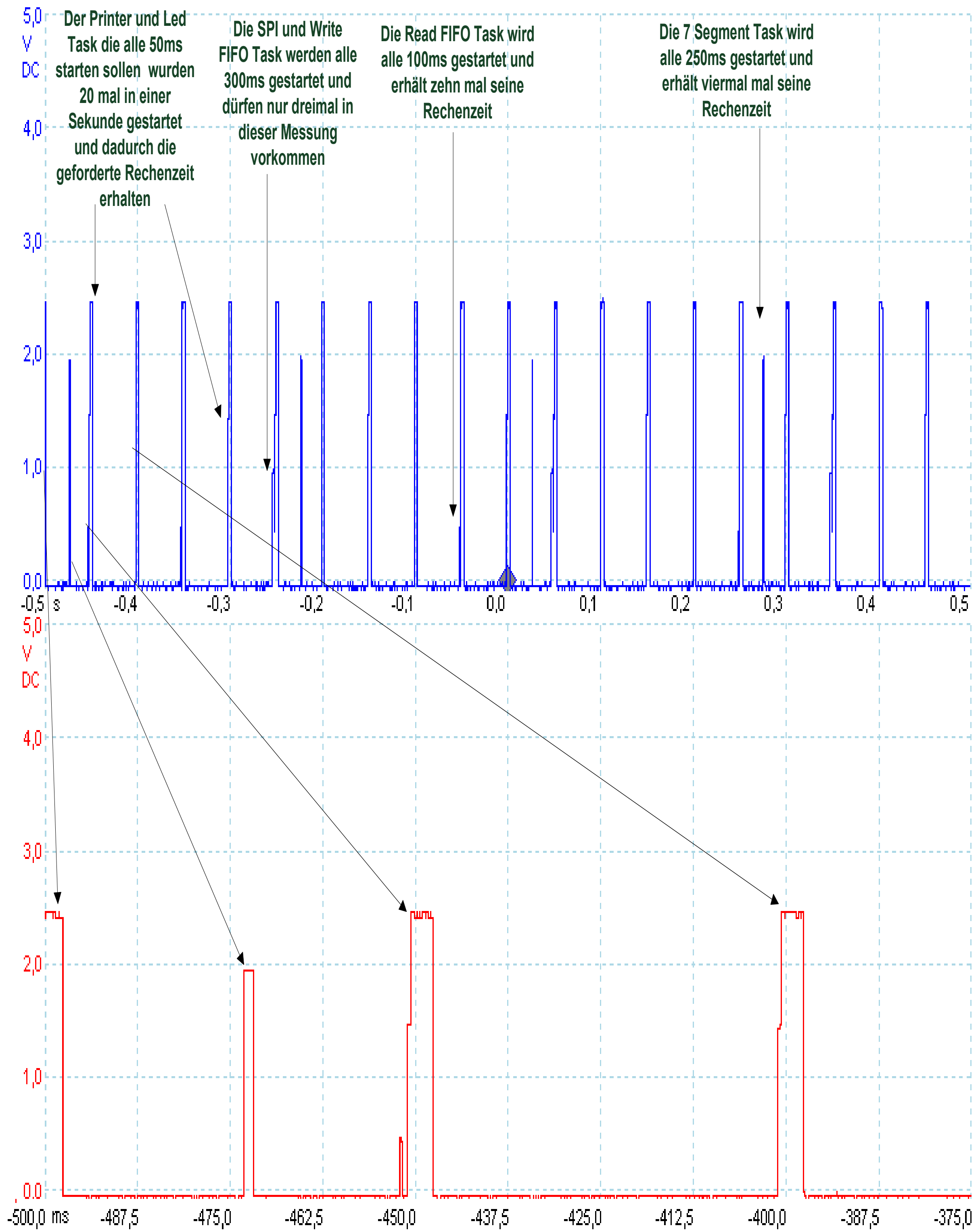
| Macro | Wert | Beschreibung |
|---|---|---|
| configMINIMAL_STACK_SIZE | 120 | Gibt die Stackgrösse eines Task an. Abhängig von der Datenbreite des Speichers ergibt sich die Stackgrösse. Bei einer Datenbreite von 32 Bit werden mit 120 insgesamt 480 Bytes allokiert. |
| configTOTAL_HEAP_SIZE | 18 * 1024 | Gibt die Komplette grösse des Heap-Speichers an, die dem Kernel zur Verfügung steht. Bei einer Datenbreite von 32 Bit werden mit 18 * 1024 insgesamt 73728 Bytes zur Verfügung gestellt. |
| configMAX_TASK_NAME_LEN | 5 | Setzt die Maximallänge eines Strings für den Namen des Tasks fest. |
| configUSE_16_BIT_TICKS | 0 | Für die Zeitmessung wird ein 16-Bit unsigned Integer anstelle eines 32 Bit unsigned Integer initialisiert. |
| configUSE_CO_ROUTINES | 0 | Zur Aktivierung der Co Routines API Funktion. |
| configMAX_CO_ROUTINE_ PRIORITIES | 0 | Setzt die höchste Priorität für eine Co Routine fest. |
| INCLUDE_vTaskPrioritySet<br>INCLUDE_uxTaskPriorityGet<br>INCLUDE_vTaskDelete<br>INCLUDE_vTaskClean UpResources<br>INCLUDE_vTaskSuspend<br>INCLUDE_vTaskDelayUntil<br>INCLUDE_vTaskDelay | 1<br>1<br>0<br>0<br>1<br>1<br>1 | Zum Aktivieren verwendeter API-Funktionen. |

Tabelle 6.1.: FreeRTOS Konfiguration in *FreeRTOSConfig.h*

Mit der Trace Funktionalität des FreeRTOS werden Messungen über den Taskverlauf der Anwendung durchgeführt. Jede Task bekommt eine eindeutige Nummer. Anhand dieser Nummer wird beim Kontextwechsel die zugehörige analoge Spannung mit dem DAC durch den Kernel gesetzt (vgl. Tabelle 6.2).

| Task Name | Trace Nummer | analoger Spannungswert |
|---|---|---|
| Idle Task | 0 | 0 V |
| Read FIFO | 1 | 0,5 V |
| HAW SPI und Write FIFO | 2 | 1,0 V |
| LED | 3 | 1,5 V |
| 7 Segment | 4 | 2,0 V |
| Printer | 5 | 2,5 V |

Tabelle 6.2.: Trace Nummer und zugehörige Ausgangsspannung der Tasks


Der Printer und Led Task die alle 50ms starten sollen wurden 20 mal in einer Sekunde gestartet und dadurch die geforderte Rechenzeit erhalten
Die SPI und Write FIFO Task werden alle 300ms gestartet und dürfen nur dreimal in dieser Messung vorkommen
Die Read FIFO Task wird alle 100ms gestartet und erhält zehn mal seine Rechenzeit
Die 7 Segment Task wird alle 250ms gestartet und erhält viermal mal seine Rechenzeit
5,0
V
DC
4,0
3,0
2,0
1,0
0,0
-0,5 s
-0,4
-0,3
-0,2
-0,1
0,0
0,1
0,2
0,3
0,4
0,5
-500,0 ms
-487,5
-475,0
-462,5
-450,0
-437,5
-425,0
-412,5
-400,0
-387,5
-375,0

# 7. Zusammenfassung

Im Rahmen dieser Arbeit wurden für den Aufbau der SoC Hardwareplattform des SoC-CC-Fahrzeugs mit der Zielanwendung der autonomen Einparkfunktion Hardware- und Software-Module entwickelt und ein Echtzeitbetriebssystem portiert.

Mit dem Xilinx EDK erfolgte der Entwurf der SoC Hardwareplattform mit einem MicroBlaze Prozessor. Das Digilent Nexys 2 Board mit dem Spartan 3E FPGA diente dabei als Evaluierungsplattform.
Das HAW SPI IP CORE wurde erstellt, um die parallele Ansteuerung und den parallelen Empfang der digitalisierten Analogspannungen (Abstandswerte) aus den ADC-Modulen zu ermöglichen. Die Abstandswerte werden für die Parklückensuche und die Trajektorien-Berechnung durch den PD-Regler des Einparkassistenten verwendet. Das Hardware-Modul kann bis zu vier SPI Slave Devices für einen parallelen Empfang ansteuern. Für die Ansteuerung der ADC-Module mit dem MicroBlaze Prozessor über den PLB wurden die High Level Treiber entwickelt und für andere SPI Slave Devices wurden die vom EDK generierten Low Level Treiber verwendet. Für die Berechnung der Kennlinie des Infrarot Sensors mit einem CoProzessor wurden das Interface des HAW SPI IP CORE um die Signalinterfaces der Datenregister und *spi_xfer_done* erweitert. Dadurch erhält der CoProzessor die Daten direkt vom IP CORE, ohne diese über den PLB auslesen zu müssen.

Für den Aufbau des Datenlogger zwischen SoC Plattform und Host-PC mit dem NET 1 Kommunikationsmodul wurde das NET 1 IP CORE erstellt. Der Host-PC initiiert über eine Ethernet-Verbindung einen Datentransfer zum NET 1 Kommunikationsmodul. Für die drahtlose Verbindung wurde dieser Port mit einer W-LAN-Bridge verbunden, der die Verbindung zum Netzwerk des Host-PC aufbaut. Der Datentransfer erfolgt vom NET 1 Kommunikationsmodul über das Digilent Parallel Interface-Modell zum FPGA, das nach dem Enhanced Paralell Port (EPP) ausgelegt ist. Für das NET 1 IP CORE wurde aus dem Referenzmodell dpimref des EPP Controllers das dpimnet erstellt. Dadurch erfolgt die Anbindung des NET 1 Moduls mit dem NET 1 IP CORE. Zu sendene Daten (z.B. Parameter) vom Host-PC werden vom dpimnet ins Read FIFO des NET 1 IP CORE geschrieben, die vom MicroBlaze über den PLB ausgelesen werden. Vom Host-PC zu empfangene Daten (Logs, Messwerte) werden vom MicroBlaze in das Write FIFO geschrieben und vom Host-PC aus diesem ausgelesen.

Mit FreeRTOS wurde ein Echtzeitbetriebssytem für das MicroBlaze-System portiert. Durch Einbettung der Software-Module, wie z.B. HAW SPI oder NET 1 Treiber, in ein Task-System erfolgt die Ansteuerung der Peripherie. Für die korrekte Arbeitsweise des Kernels wurde eine Demoanwendung erstellt, der Taskverlauf gemessen und evaluiert.

# Tabellenverzeichnis

3.1. IPIF Signal Interfaces im IPIC Format vom PLB Slave für das User IP CORE . . . 26
3.2. IPIC Signal Interfaces vom PLB Slave für das User IP CORE . . . . . . . . . . . 29

4.1. Signalleitungen des SPI Busses . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.2. Wertetabelle der Umkehrfunktion . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.3. Aus $\Delta t_1$ für die Compiler Optimierungsstufen -O1, -O2, -O3, Os ergibt sich die maximal realisierbare Abtastfrequenz eines SPI Bustransfers mit SCLK = 12,5 MHZ, 16 Datenbits, Systemtakt = 50 MHZ. Fixer Zeitanteil der Abtastzeit . . . . . 45

5.1. Digilent Parallel Interface Model . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.2. Wahrheitstabelle für Tri-State-Treiber . . . . . . . . . . . . . . . . . . . . . . 53
5.3. Parametrisierung der Write und Read FIFOs . . . . . . . . . . . . . . . . . . . 55
5.4. IPIF Signal Interfaces zwischen der User-Logic und Read Control-Logic des Write FIFO . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
5.5. Adressraum des Write FIFO für TransPort vom Host-PC . . . . . . . . . . . . . 57
5.6. IPIF Signal Interfaces zwischen der User-Logic und Write Control-Logic des Read FIFO . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
5.7. Adressraum des Read FIFO für TransPort vom Host PC . . . . . . . . . . . . . 58

6.1. FreeRTOS Konfiguration in *FreeRTOSConfig.h* . . . . . . . . . . . . . . . . . 66
6.2. Trace Nummer und zugehörige Ausgangsspannung der Tasks . . . . . . . . . . 66

A.1. HAW SPI IP Core Design Parameter . . . . . . . . . . . . . . . . . . . . . . . 78
A.2. XPS SPI IP Core I/O Signal Beschreibung . . . . . . . . . . . . . . . . . . . . 79
A.3. XPS SPI IP Core I/O Signal Beschreibung . . . . . . . . . . . . . . . . . . . . 80
A.4. XPS SPI IP Core I/O Signal Beschreibung . . . . . . . . . . . . . . . . . . . . 81
A.5. XPS SPI IP Core Register Beschreibung . . . . . . . . . . . . . . . . . . . . . 82
A.6. HAW SPI Status Register (HAWSPISR) Beschreibung(C_BASEADDR + 0x4) . . 83
A.7. HAW SPI Control Register (HAWSPICR) Beschreibung(C_BASEADDR + 0x4) . . 84
A.8. Slave Select Register(SRR) Beschreibung(C_BASEADDR + 0x8) . . . . . . . . 85
A.9. HAW SPI Data Receive Register 1-4 (SPIDRR 1-4) Beschreibung(C_BASEADDR + (0xC - 0x18)) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
A.10.Software Reset Register(SRR) Beschreibung(C_BASEADDR + 0x100) . . . . . 86
A.11.Device Global Interrupt Enable Register(DGIER) Beschreibung(C_BASEADDR + 0x21C) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
A.12.IP Interrupt Status Register (IPISR) Beschreibung(C_BASEADDR + 0x220) . . . 87
A.13.IP Interrupt Enable Register (IPIER) Beschreibung(C_BASEADDR + 0x228) . . . 88
A.14.Transferformat Konfiguration des HAW SPI IP CORE . . . . . . . . . . . . . . 89
A.15. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94

# Abbildungsverzeichnis


1.1. SoC Übersicht des SoC-CC-Fahrzeugs . . . . . . . . . . . . . . . . . . . . . . 8

2.1. Kinematisches Einspurfahrzeugmodell . . . . . . . . . . . . . . . . . . . . . . 10
2.2. Szenario der Parklückensuche mit den Infrarotsensoren für die seitliche Tiefenmessung und Berechnung des gefahrenen Weges . . . . . . . . . . . . . . . . 11
2.3. Autonomer Parkvorgang des SoC-CC-Fahrzeugs . . . . . . . . . . . . . . . . 12
2.4. Abstandsregelung der Y-Koordinate des Hinterrads $Y_m$ mit einem digitalen PD-Regler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.5. Regelkreis mit Infrarot gestützter Abstandserfasssung . . . . . . . . . . . . . 13
2.6. Abstandserfassung ga mit dem Infrarotsensor des vorderen Hindernis . . . . . . 14
2.7. Auswahl zwischen virtueller und infrarotgestützter Positionsermittlung . . . . . . 14

3.1. Nexys 2 Evaluierungsboard . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.2. Blockschaltbild des Microblaze Prozessors [Xilinx (2004)] . . . . . . . . . . . . 17
3.3. IEEE 754 Single Precision Format [IEEE (1991)] . . . . . . . . . . . . . . . . 18
3.4. Harvardarchitektur des MicroBlaze mit dem Lokal Memory Bus [Xilinx (2009)] . . 20
3.5. Processor Local Bus Topologie [Xilinx (2007)] . . . . . . . . . . . . . . . . . . 20
3.6. Anbindung von Hardware-Beschleuniger/CoProzessoren an die Register Files des MicroBlaze über FSL . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.7. Embedded Development Kit Tool Chain . . . . . . . . . . . . . . . . . . . . . 22
3.8. Blockschaltbild PLB IPIF [Xilinx (2005)] . . . . . . . . . . . . . . . . . . . . 24
3.9. Blockschaltbild eines User IP CORE . . . . . . . . . . . . . . . . . . . . . . 25
3.10. Blockschaltbild User Logic mit zwei Software-Registern . . . . . . . . . . . . . 26

4.1. Parallele Abstandsmessung mit dem HAW SPI Bus, vier Analog-Digital-Umsetzer S7476 und Sharp GP2D12 Infrarotsensoren . . . . . . . . . . . . . . . . . . 30
4.2. Top Entity-Blockschaltbild des HAW SPI IP COREs . . . . . . . . . . . . . . . 31
4.3. Zwei verschiedene Bustopologien lassen sich mit den Signalleitungen eines Standard SPI Master Moduls aufbauen. . . . . . . . . . . . . . . . . . . . . . . . 33
4.4. PMOD Modul mit zwei ADCS7476 (vgl. Digilent (2004b)) . . . . . . . . . . . . 33
4.5. Timingverhalten der vier Transferformate des SPI Busses [Motorola (2005)] . . . 34
4.6. SPI Bustransfer mit CPOL = 0; CPHA = 0; SCLK = 12,5MHZ; 16 Datenbits; (1) CS; (2) SCLK; (3) MISO 0xaaaa . . . . . . . . . . . . . . . . . . . . . . . . 36
4.7. SPI Bustransfer mit CPOL = 1; CPHA = 0; SCLK = 12,5MHZ; 16 Datenbits; (1) CS; (2) SCLK; (3) MISO 0xaaaa . . . . . . . . . . . . . . . . . . . . . . . . 37
4.8. SPI Bustransfer mit CPOL = 0; CPHA = 1; SCLK = 12,5MHZ; 16 Datenbits; (1) CS; (2) SCLK; (3) MISO 0xaaaa . . . . . . . . . . . . . . . . . . . . . . . . 37
4.9. SPI Bustransfer mit CPOL = 1; CPHA = 1; SCLK = 12,5MHZ; 16 Datenbits; (1) CS; (2) SCLK; (3) MISO 0xaaaa . . . . . . . . . . . . . . . . . . . . . . . . 38
4.10. Blockdiagramm des Analog-Digital-Umsetzter ICs [National (2007)] . . . . . . . 39

4.11. Timingdiagramm des ADCS7476 (vgl.National (2007)) . . . . . 40
4.12. Timingdiagramm ADCS7476 mit HAW SPI IP (CPOL = 1, CPOHA = 0) . . . . . 40
4.13. Sharp GP2D12 Infrarotabstandssensor [SHARP] . . . . . 41
4.14. Sharp GP2D12 Timing [SHARP] . . . . . 41
4.15. Die nichtlineare Ausgangsspannung $V_O$ bezogen auf die Entfernung des Objekts mit den Einflussfaktoren Temperatur, Beleuchtungsstärke und Reflexionsmaterial der Umgebung [SHARP] . . . . . 42
4.16. Umkehrfunktion der Ausgangsspannung über die Entfernung des reflektierenden Objekts . . . . . 42
4.17. Das Zeitintervall $\Delta t_1$ bestimmt die maximal realisierbare Abtastfrequenz . . . . . 44
4.18. Zeitmessung $\Delta t_1$ ohne Compiler Optimierungsstufe mit SCLK = 12,5MHZ; 16 Datenbits; Das Zeitintervall der Low Phase entspricht der restlichen Zeit der Timer Periode nach der Abtastung . . . . . 45
4.19. Zeitmessung $\Delta t_1$ für Compiler Optimierungsstufen -O1, -O2, -O3, -Os mit SCLK = 12,5MHZ; 16 Datenbits; Das Zeitintervall der Low Phase entspricht die restliche Zeit der Timer Periode nach der Abtastung . . . . . 45
4.20. Signalrekonstruktion des abgetasteten Signals . . . . . 46
4.21. Verzögerungszeit $\Delta t_1 + \Delta t_2$ zwischen Eingangs- und rekonstruiertem Ausgangssignal . . . . . 46

5.1. Hardwareaufbau für die Funktion des Fernfeldtelemetriesystems, Parametrisierung von Systemkomponenten und JTAG Programmierung mit dem NET 1 Modul über eine WLAN Bridge . . . . . 47
5.2. UBICOM IP2022 $\mu$Controller [Digilent (2004a)] . . . . . 48
5.3. Adress Write Buszyklus (Host output, Peripheral input) [Digilent (2008)] . . . . . 50
5.4. Adress Read Buszyklus (Host input, Peripheral output) [Digilent (2008)] . . . . . 50
5.5. Data Write Buszyklus (Host output, Peripheral input) [Digilent (2008)] . . . . . 51
5.6. Data Read Buszyklus (Host input, Peripheral output) [Digilent (2008)] . . . . . 51
5.7. Net 1 IP CORE Blockschaltbild . . . . . 52
5.8. Medwedew-Automatenmodell des dpimnet Controllers . . . . . 53
5.9. Zustandsdiagramm des dpimnet Controllers . . . . . 54
5.10. Write FIFO Block Diagramm [Xilinx (2006b)] . . . . . 56
5.11. Read FIFO Block Diagramm [Xilinx (2006a)] . . . . . 57
5.12. Linksys WET54G W-LAN Bridge . . . . . 58

6.1. Auszug aus der Verzeichnisstruktur des FreeRTOS [FreeRTOS] . . . . . 60
6.2. FreeRTOS Task-Zustände und die verwendete API für die Zustandswechsel.[FreeRTOS] . . . . . 63
6.3. FreeRTOS Co-Routines Zustände [FreeRTOS] . . . . . 64
6.4. Messung vom Taskverlauf der Demoanwendung . . . . . 67

A.1. Top Level Blockschaltbild des HAW SPI . . . . . 77
A.2. HAW SPI Status Register . . . . . 82
A.3. HAW SPI Control Register . . . . . 84
A.4. Slave Select Register . . . . . 85
A.5. Data Receive Register 1-4(SPIDRR) . . . . . 85
A.6. Software Reset Register . . . . . 86
A.7. Device Global Interrupt Enable Register . . . . . 86
A.8. HAW SPI IP Interrupt Status Register . . . . . 87

A.9. HAW SPI IP Interrupt Enable Register . . . . . . . . . . . . . . . . . . . . . . . 88
A.10. HAW SPI Bustopologie . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
A.11. PMOD Analog Digital Umsetzter (vgl. Digilent (2004b)) . . . . . . . . . . . . . . 89
A.12. Timing für Receive mit CPHA = 0, CPOL = 0, SCK_RATIO = 4, C_NUM_TRANSFER_BITS = 8 . . . . . . . . . . . . . . . . . . . . . . . . . . 90
A.13. Timing für Receive mitCPHA = 1, CPOL = 0, SCK_RATIO = 4, C_NUM_TRANSFER_BITS = 8 . . . . . . . . . . . . . . . . . . . . . . . . . . 91

# Literaturverzeichnis


[Digilent 2004a] DIGILENT: *Digilent Ethernet Module Reference Manual*, 2004. – URL http://www.digilentinc.com/Data/Products/Net1/NET1-rm.pdf

[Digilent 2004b] DIGILENT: Digilent PmodAD1 Analog To Digital Module Converter Board Reference Manual. (2004). – URL http://digilentinc.com/Data/Products/PMOD-AD1/Pmod%20AD1_rm.pdf

[Digilent 2008] DIGILENT: *Supported Devices*. 2008. – URL http://digilentinc.com/Data/AppNotes/dpimref.vhd

[FreeRTOS ] FREERTOS: *Digilent Parallel Interface Model Reference Manual*. – URL http://www.freertos.org/

[GNU ] GNU: *Homepage des GNU. C Compile und Debugger für verschiedene Architekturen*. – URL http://www.gnu.org

[IEEE 1991] IEEE: *IEEE Standard for Binary Floating-Point Arithmetic*. 1991. – URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=30711&isnumber=1316

[Liu 2008] LIU, Ning: *Ein automatischer Parkassistent auf Basis einer Laserscanner-Abstandserfassung für ein fahrerloses Transportsystem*, Hochschule für Angewandte Wissenschaften Hamburg, Masterarbeit, 2008

[Motorola 2005] MOTOROLA: *M68HC11E*, 2005. – URL http://www.freescale.com/files/microcontrollers/doc/data_sheet/M68HC11E.pdf

[National 2007] NATIONAL, Semiconductor: *ADCS7478*, 2007. – URL http://www.national.com/ds/DC/ADCS7476.pdf

[Schetler 2007] SCHETLER, Denis: *Automatischer Ausweichassistent mit einer Laserscanner - basierten Abstandsregelung für ein fahrerloses Transportsystem*, Hochschule für Angewandte Wissenschaften Hamburg, Masterarbeit, 2007

[SHARP ] SHARP: *GP2D12/GP2D15*. – URL http://www.robotikhardware.de/download/gp2d12.pdf

[TheMathWorks ] THEMATHWORKS: *MATLAB Dokumentation Mathematics Polynomials*. – URL http://www.mathworks.com/access/helpdesk/help/techdoc/index.html?/access/helpdesk/help/techdoc/math/brfaisd-17.html

[Xilinx 2004] XILINX: *MicroBlaze Processor Reference Guide*, 2004. – URL http://www.xilinx.com/ise/embedded/edk6_2docs/mb_ref_guide.pdf

[Xilinx 2005] XILINX: *PLB IPIF (v2.02a) Product Specification*, 2005. – URL http://www.xilinx.com/support/documentation/ip_documentation/plb_ipif.pdf

[Xilinx 2006a] XILINX: *Read Packet FIFO (v4.00a) Product Specification*, 2006

[Xilinx 2006b] XILINX: *Write Packet FIFO (v4.00a) Product Specification*, 2006

[Xilinx 2007] XILINX: *Processor Local Bus (PLB) v4.6 (v1.00a) Product Specification*, 2007. – URL http://www.xilinx.com/support/documentation/ip_documentation/plb_v46.pdf

[Xilinx 2008] XILINX: *XPS Serial Peripheral Interface (SPI) (v2.00b) Product Specification*, 2008. – URL Http://www.xilinx.com/support/documentation/ip_documentation/xps_spi.pdf

[Xilinx 2009] XILINX: *LMB BRAM Interface Controller (v2.10b) Product Specification*, 2009. – URL http://www.xilinx.com/support/documentation/ip_documentation/lmb_bram_if_cntlr.pdf

Hochschule für Angewandte Wissenschaften Hamburg
*Hamburg University of Applied Sciences*

# HAW SPI IP CORE Beschreibung

HAW SPI IP CORE v 1.0

Saman Farshbaf-Masalehdan

*Fakultät Technik und Informatik*
*Department Informatik*

*Faculty of Engineering and Computer Science*
*Department of Computer Science*

# A. HAW SPI IP CORE Beschreibung

## A.1. Einführung

Das HAW Serial Peripheral Interface (SPI) ist ein unidirektionales serielles Interface für vier SPI Devices wie z.B. den ADCS7478 von National zum parallelen Empfang von 4 Datenströmen.

<table>
<tr><th colspan="3">IP Core Daten</th></tr>
<tr><td>Unterstützte Device Familien</td><td colspan="2">Spartan®-3,Spartan-3E,Spartan-3A,Spartan-3ADSP,Spartan-3AN,Virtex®-II Pro, Virtex-4,QPro Virtex-4 Hi Rel,QPro Virtex-4 Rad Tolerant,Virtex-5</td></tr>
<tr><td>Version des Cores</td><td>haw_spi</td><td>v1.00a</td></tr>
<tr><th colspan="3">genutzte Ressourcen</th></tr>
<tr><td></td><td>Min</td><td>Max</td></tr>
<tr><td>Slices<br>LUTs<br>FFs</td><td colspan="2">Siehe Tabelle A.15</td></tr>
<tr><td>Block RAMs</td><td colspan="2">N/A</td></tr>
<tr><td>Special Features</td><td colspan="2">N/A</td></tr>
</table>

## A.2. Funktion

- Mit dem $\mu$Blaze verbunden als 32-bit Slave am PLB V4.6 mit 32,64 oder 128 bits
- Drei Signal Schnittstellen(MISO,SCK und $\overline{SS}$)
- Master Mode
- Konfigurierbar für Clock Phase und Clock Polarität
- Automatischer oder manueller $\overline{SS}$ Mode
- Transferlängen von 8,16 und 32 bits

## A.3. Funktionale Beschreibung

- Der HAW SPI IP Core(vgl. Abbildung A.1) ist ein unidirektionaler synchroner Kommunkationskanal, der nur die Master Funktionalität für die parallele Steuerung (SCK,$\overline{SS}$) und den Empfang(MISO) von bis zu 4 gleichartigen SPI Slave Modulen realisiert.

- Der HAW SPI IP Core unterstützt das manuele Toggeln des $\overline{SS}$ Signals als Standard Konfiguration. Die automatische Konfiguration toggelt das $\overline{SS}$ Signal nach jedem Transfer, der beendet wurde.
- Alle SPI und INTR Register sind 32-bit breit.

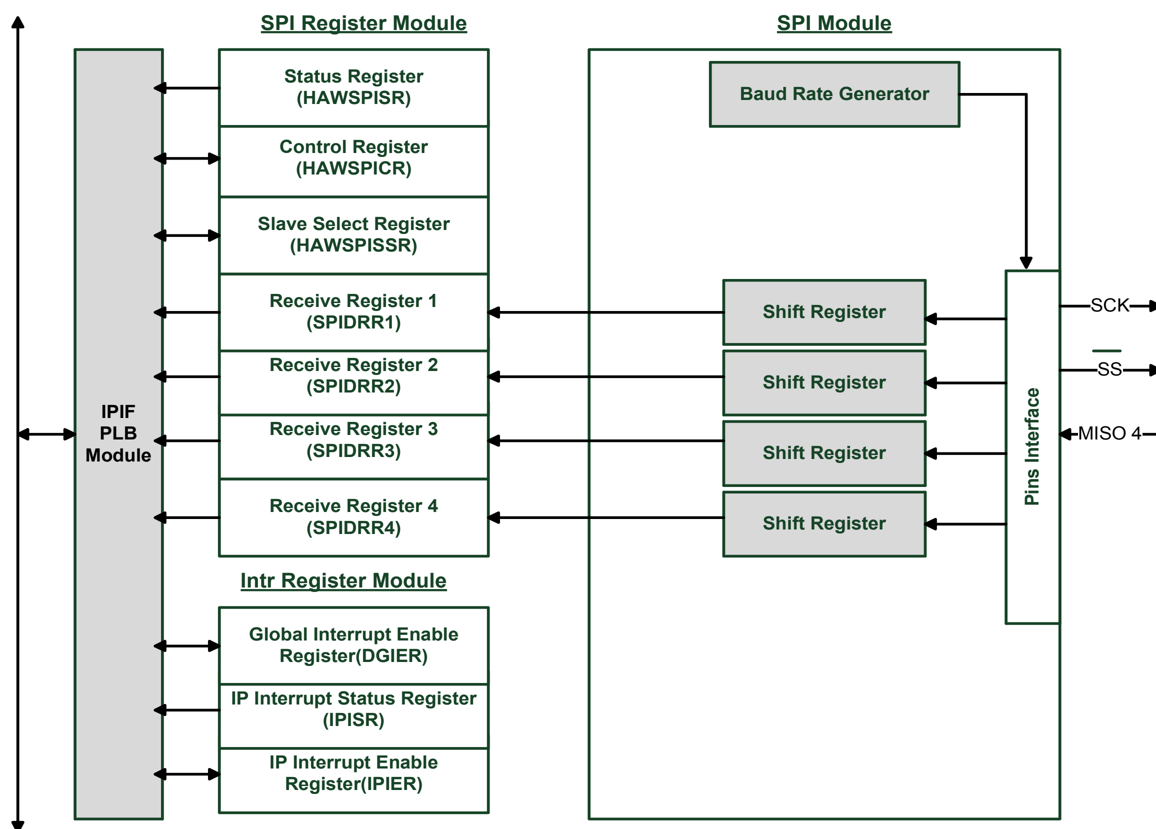



Abbildung A.1.: Top Level Blockschaltbild des HAW SPI

**PLB Interface Modul:** Das PLB Interface Modul ist die Standardanbindung des IP Cores als Slave zum PLB V4.6 zur Decodierung der Adressen. Die Lese/Schreib Transaktion des PLBs werden in ein äquivalentes IP Interconnect(IPIC) Lese/Schreib Transaktion Format übersetzt. Die Software Register sind mit dem IPIC verbunden.

**SPI Modul:** Das SPI Modul besitzt vier Empfangsschieberegister mit einer Breite von 16-bit, deren Eingänge mit dem 4-bit breiten MISO Port verbunden sind. Die Frequenz des Taktsignals SCK, das die Empfangschieberegister und die Slave Glieder taktet, wird über den Baudrategenerator parametrisiert.

**SPI Register Modul:** Das SPI Register Modul beinhaltet die Memory mapped Steuer, Status und Daten SW-Register.

**INTR Register Modul:** Für die Interrupt Steurung des HAW SPI IP Cores wurde das Interface Interrupt Control eingebunden(vgl. xilinx ic).Die zugehörigen Register sind im INTR Register Modul. Das INTR Regsiter Modul beinhaltet die zugehörigen Interrupt Register wie device global interrupt enable(DGIER), IP interrupt enable register(IPIER) und IP interrupt status Register(IPISR).

## A.4. HAW SPI IP Core Design Parameter

Zur Abstimmung des HAW SPI IP Coreauf die Anforderungen der jeweiligen Anwendung sind die in Tabelle A.1 aufgelisteten Generics zur Parametrisierung des IP Cores zu wählen.

| Gen. | Feature / Description | Parameter Name | Allowable Values | Default Value | VHDL Type |
|---|---|---|---|---|---|
| **System Parameters** | | | | | |
| G1 | Target FPGA family | C_FAMILY | spartan-3,3e,3a,3adsp, 3an,virtex- 2p,4,5, q-qrvirtex4 | virtex5 | string |
| **PLB Parameters** | | | | | |
| G2 | PLB base address | C_BASEADDR | Valid Address(1) | None(2) | std l. vec |
| G3 | PLB high address | C_HIGHADDR | Valid Address(1) | None(2) | std l. vec |
| G4 | PLB least significant address bus width | C_SPLB_ AWIDTH | 32 | 32 | integer |
| G5 | PLB data width | C_SPLB_ DWIDTH | 32, 64, 128 | 32 | integer |
| G6 | Shared bus topology | C_SPLB_P2P | 0 = Shared bus topology(3) | 0 | integer |
| G7 | PLB master ID bus Width | C_SPLB_MID_ WIDTH | log2(C_SPLB_ NUM_MASTERS) min. val. = 1 | 1 | integer |
| G8 | Number of PLB masters | C_SPLB_NUM_ MASTERS | 1 - 16 | 1 | integer |
| G9 | Width of the slave data bus | C_SPLB_ NATIVE_ DWIDTH | 32 | 32 | integer |
| G10 | Burst support | C_SPLB_ SUPPORT_ BURSTS | 0 = No burst support(4) | 0 | integer |
| **XPS SPI IP Core Parameters** | | | | | |
| G11 | SPI clock frequency ratio | C_SCK_RATIO | 2, 4, 16, 32, Nx16 for N = 1, 2, 3,...,128 | 32(6) | integer |
| G12 | SPI number of Transfer bits | C_NUM_TRANSFER _BITS | 8,16,32 | 16 | integer |

Tabelle A.1.: HAW SPI IP Core Design Parameter

## A.5. HAW SPI IP Core I/O Signale

Die HAW SPI IP Core Signale sind in den Tabellen A.2 und A.3 aufgelistet.

| Port | Signal Name | Interface | I/O | Initial State | Description |
|---|---|---|---|---|---|
| **System Signals** | | | | | |
| P1 | SPLB_Clk | System | I | - | PLB clock |
| P2 | SPLB_Rst | System | I | - | PLB reset, active high |
| **PLB Master Interface Signals** | | | | | |
| P3 | IP2INTC_Irpt | System | O | 0 | Interrupt control signal from SPI PLB Master Interface Signals |
| P4 | PLB_ABus[0 : 31] | PLB | I | - | PLB address bus |
| P5 | PLB_PAValid | PLB | I | - | PLB primary address valid |
| P6 | PLB_masterID[0 : C_SPLB_MID_WIDTH - 1] | PLB | I | - | PLB current master identifier |
| P7 | PLB_RNW | PLB | I | - | PLB read not write |
| P8 | PLB_BE[0 : (C_SPLB_DWIDTH/8) - 1] | PLB | I | - | PLB byte enables |
| P9 | PLB_size[0 : 3] | PLB | I | - | PLB size of requested transfer |
| P10 | PLB_type[0 : 2] | PLB | I | - | PLB transfer type |
| P11 | PLB_wrDBus[0 : C_SPLB_DWIDTH - 1] | PLB | I | - | PLB write data bus |
| **Unused PLB Master Interface Signals** | | | | | |
| P12 | PLB_UABus[0 : 31] | PLB | I | - | PLB upper address bits |
| P13 | PLB_SAValid | PLB | I | - | PLB secondary address valid |
| P14 | PLB_rdPrim | PLB | I | - | PLB secondary to primary read request indicator |
| P15 | PLB_wrPrim | PLB | I | - | PLB secondary to primary write request indicator |
| P16 | PLB_abort | PLB | I | - | PLB abort bus request |
| P17 | PLB_busLock | PLB | I | - | PLB bus lock |
| P18 | PLB_MSize[0 : 1] | PLB | I | - | PLB data bus width indicator |
| P19 | PLB_lockErr | PLB | I | - | PLB lock error |
| P20 | PLB_wrBurst | PLB | I | - | PLB burst write transfer |
| P21 | PLB_rdBurst | PLB | I | - | PLB burst read transfer |
| P22 | PLB_wrPendReq | PLB | I | - | PLB pending bus write request |

Tabelle A.2.: XPS SPI IP Core I/O Signal Beschreibung

| | | | | | |
|---|---|---|---|---|---|
| P23 | PLB_rdPendReq | PLB | I | - | PLB pending bus read request |
| P24 | PLB_wrPendPri[0 : 1] | PLB | I | - | PLB pending write request priority |
| P25 | PLB_rdPendPri[0 : 1] | PLB | I | - | PLB pending read request priority |
| P26 | PLB_reqPri[0 : 1] | PLB | I | - | PLB current request priority |
| P27 | PLB_TAttribute[0 : 15] | PLB | I | - | PLB transfer attribute |
| **PLB Slave Interface Signals** | | | | | |
| P28 | Sl_addrAck | PLB | O | 0 | Slave address acknowledge |
| P29 | Sl_SSize[0 : 1] | PLB | O | 0 | Slave data bus size |
| P30 | Sl_wait | PLB | O | 0 | Slave wait |
| P31 | Sl_rearbitrate | PLB | O | 0 | Slave bus rearbitrate |
| P32 | Sl_wrDAck | PLB | O | 0 | Slave write data acknowledge |
| P33 | Sl_wrComp | PLB | O | 0 | Slave write transfer complete |
| P34 | Sl_rdDBus[0 : C_SPLB_DWIDTH - 1] | PLB | O | 0 | Slave read data bus |
| P35 | Sl_rdDAck | PLB | O | 0 | Slave read data acknowledge |
| P36 | Sl_rdComp | PLB | O | 0 | Slave read transfer complete |
| P37 | Sl_MBusy[0 : C_SPLB_NUM_ MASTERS - 1] | PLB | O | 0 | Slave busy |
| P38 | Sl_MWrErr[0 : C_SPLB_NUM_ MASTERS - 1] | PLB | O | 0 | Slave write error |
| P39 | Sl_MRdErr[0 : C_SPLB_NUM_ MASTERS - 1] | PLB | O | 0 | Slave read error |
| **Unused PLB Slave Interface Signals** | | | | | |
| P40 | Sl_wrBTerm | PLB | O | 0 | Slave terminate write burst transfer |
| P41 | Sl_rdWdAddr[0 : 3] | PLB | O | 0 | Slave read word address |
| P42 | Sl_rdBTerm | PLB | O | 0 | Slave terminate read burst transfer |
| P43 | Sl_MIRQ[0 : C_SPLB_NUM_ MASTERS - 1] | PLB | O | 0 | Master interrupt request |
| **SPI Interface Signals** | | | | | |
| P44 | SCK | SPI | O | 0 | SPI bus clock output |
| P45 | MISO_I[0 : 3] | SPI | I | - | Master input slave output |
| P46 | $\overline{SS}$[0 : N] | SPI | O | 1 | Output active low slave select vector of max. length for N = 4 |

Tabelle A.3.: XPS SPI IP Core I/O Signal Beschreibung

## A.6. HAW SPI IP Core Parameter-Port Abhängigkeiten

Die Abhängigkeiten der HAW SPI IP Core Design Parameter und I/O Signalen ist in Tabelle A.4 aufgelistet.

| Genric or Port | Name | Affects | Depends | Relationship Descr. |
|---|---|---|---|---|
| **Design Parameter** | | | | |
| G5 | C_SPLB_DWIDTH | P8, P11, P34 | - | Affects the number of bits in data bus |
| G7 | C_SPLB_MID_WIDTH | P6 | G8 | This value is calculated as: log2 (C_SPLB_NUM_MASTERS) with a minimum value of 1 |
| G8 | C_SPLB_NUM_MASTERS | P37, P38, P39, P43 | - | Affects the number of PLB masters |
| **I/O Signale** | | | | |
| P6 | PLB_masterID[0 : C_SPLB_MID_WIDTH-1] | - | G7 | Width of the PLB_ masterID varies according to C_SPLB_MID_WIDTH |
| P8 | PLB_BE[0 : (C_SPLB_DWIDTH/8)-1] | - | G5 | Width of the PLB_BE varies according to C_SPLB_DWIDTH |
| P11 | PLB_wrDBus[0 : C_SPLB_DWIDTH-1] | - | G5 | Width of the PLB_wrDBus varies according to C_SPLB_DWIDTH |
| P34 | Sl_rdDBus[0 : C_SPLB_DWIDTH-1] | - | G5 | Width of the Sl_rdDBus varies according to C_SPLB_DWIDTH |
| P37 | Sl_MBusy[0 : C_SPLB_NUM_MASTERS-1] | - | G8 | Width of the Sl_MBusy varies according to C_SPLB_NUM_MASTERS |
| P38 | Sl_MWrErr[0 : C_SPLB_NUM_MASTERS-1] | - | G8 | Width of the Sl_MWrErr varies according to C_SPLB_NUM_MASTERS |
| P39 | Sl_MRdErr[0 : C_SPLB_NUM_MASTERS-1] | - | G8 | Width of the Sl_MRdErr varies according to C_SPLB_NUM_MASTERS |
| P43 | Sl_MIRQ[0 : C_SPLB_NUM_MASTERS-1] | - | G8 | Width of the Sl_MIRQ varies according to C_SPLB_NUM_MASTERS |

Tabelle A.4.: XPS SPI IP Core I/O Signal Beschreibung

## A.7. HAW SPI IP Core Registerbeschreibung

| Baseadress + Offset | Register Name | Type | Def. Value | Description |
|---|---|---|---|---|
| C_BASEADDR+0x0 | HAWSPICR | R/W | 0x180 | HAW_SPI Control Register |
| C_BASEADDR+0x4 | HAWSPISR | Read | 0x85 | HAW_SPI_SPI Status Register |
| C_BASEADDR+0x8 | HAWSPI-SSR | R/W | 0x0 | HAW_SPI_SPI Slave Select Register |
| C_BASEADDR+0xC | SPIDRR1 | Read | 0x0 | SPI Data Receive Register 1 |
| C_BASEADDR+0x10 | SPIDRR2 | Read | 0x0 | SPI Data Receive Register 2 |
| C_BASEADDR+0x14 | SPIDRR3 | Read | 0x0 | SPI Data Receive Register 3 |
| C_BASEADDR+0x18 | SPIDRR4 | Read | 0x0 | SPI Data Receive Register 4 |
| C_BASEADDR+0x1C | SRR | Write | N/A | Software Reset Register |
| C_BASEADDR+0x21C | DGIER | R/W | 0x0 | Device Global Interrupt Enable Register |
| C_BASEADDR+0x220 | IPISR | R/TOW | 0x0 | IP Interrupt Status Register |
| C_BASEADDR+0x28 | IPIER | R/W | 0x0 | IP Interrupt Enable Register |

Tabelle A.5.: XPS SPI IP Core Register Beschreibung

### A.7.1. HAW SPI Status Register (HAWSPISR)

Das HAWSPI Status Register(HAWSPICR) zeigt den Status der einzelnen Empfangsregister (SPIDRR 1-4) an: voll, leer oder Überlauf. Ein Überlauf bedeutet hier das versucht wird auf ein nicht leeres Empfangsregister zu schreiben. Die Bitpositionen des HAWSPISR sind Abbildung A.2 dargestellt und in der Tabelle A.6 erläutert.

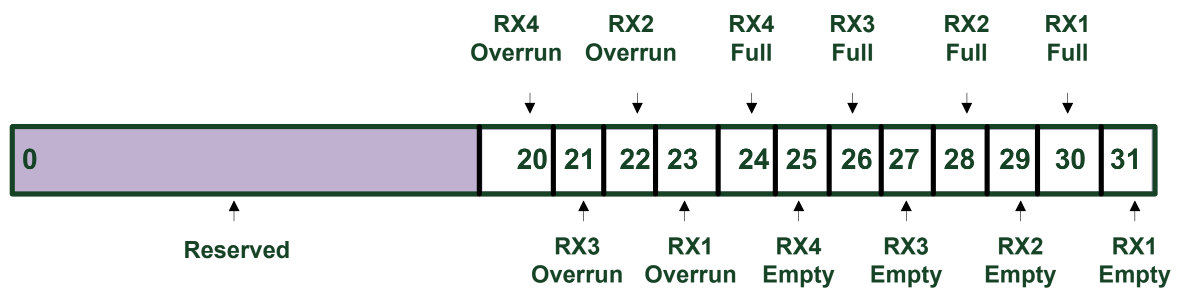



Abbildung A.2.: HAW SPI Status Register

| Bit (s) | Name | Core Access | Reset Value | Description |
|---|---|---|---|---|
| 0 - 19 | Reserved | N/A | N/A | Reserved |
| 20 | RX1_DRR_O | Read | 0 | **Receive Register 1 Overruns**, wenn das HAW SPI IP Core Logik versucht, in ein volles RX1 Register zu schreiben. |
| 21 | RX2_DRR_O | Read | 0 | **Receive Register 2 Overruns**, wenn das HAW SPI IP Core Logik versucht, in ein volles RX2 Register zu schreiben. |
| 22 | RX3_DRR_O | Read | 0 | **Receive Register 3 Overruns**, wenn das HAW SPI IP Core Logik versucht, in ein volles RX3 Register zu schreiben. |
| 23 | RX4_DRR_O | Read | 0 | **Receive Register 4 Overruns**, wenn das HAW SPI IP Core Logik versucht, in ein volles RX4 Register zu schreiben. |
| 24 | RX4_Full | Read | 0 | **Receive Register 4 Full**, wenn ein Empfang abgeschlossen ist, wird das Bit gesetzt und ist das Komplement zum RX4_Empty Bit. |
| 25 | RX4_Empty | Read | 1 | **Receive Register 4 Empty**, durch das Lesen des Register wird das Bit gesetzt und nach einem abgeschlossen Empfang gelöscht. |
| 26 | RX3_Full | Read | 0 | **Receive Register 3 Full**, wenn ein Empfang abgeschlossen ist, wird das Bit gesetzt und ist das Komplement zum RX3_Empty Bit. |
| 27 | RX3_Empty | Read | 1 | **Receive Register 3 Empty**, durch das Lesen des Register wird das Bit gesetzt und nach einem abgeschlossen Empfang gelöscht. |
| 28 | RX2_Full | Read | 0 | **Receive Register 2 Full**, wenn ein Empfang abgeschlossen ist, wird das Bit gesetzt und ist das Komplement zum RX2_Empty Bit. |
| 29 | RX2_Empty | Read | 1 | **Receive Register 2 Empty**, durch das Lesen des Register wird das Bit gesetzt und nach einem abgeschlossen Empfang gelöscht. |
| 30 | RX1_Full | Read | 0 | **Receive Register 1 Full**, wenn ein Empfang abgeschlossen ist, wird das Bit gesetzt und ist das Komplement zum RX1_Empty Bit. |
| 31 | RX1_Empty | Read | 1 | **Receive Register 1 Empty**, durch das Lesen des Register wird das Bit gesetzt und nach einem abgeschlossen Empfang gelöscht. |

Tabelle A.6.: HAW SPI Status Register (HAWSPISR) Beschreibung(C_BASEADDR + 0x4)

### A.7.2. HAW SPI Control Register (HAWSPICR)

Das HAWSPI Control Register(HAWSPICR) für die Steuerung des HAW SPI IP Cores. Die Bitpositionen des HAWSPICR(vgl. Abbildung A.3) werden in der Tabelle A.7 erläutert.

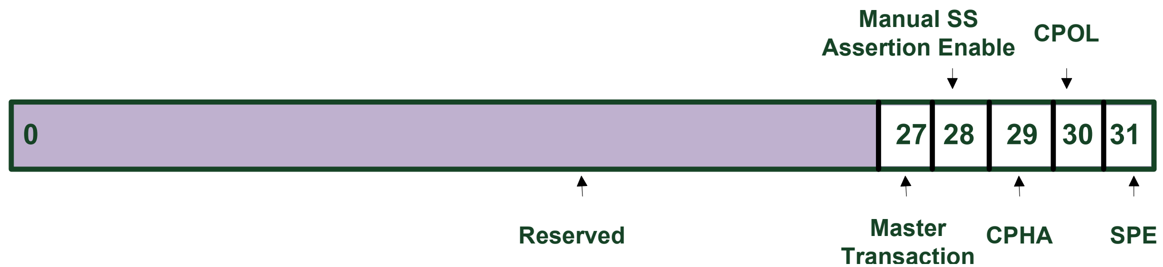



Abbildung A.3.: HAW SPI Control Register

| Bit(s) | Name | C.A. | R.V. | Description |
|---|---|---|---|---|
| 0-26 | N/A | N/A | N/A | Reserved |
| 27 | Master Transaction Inhibit | R/W | 1 | Master Transaction Inhibit. Das Bit verhindert Master-Transaktionen. 0 = Master transactions enabled 1 = Master transactions disabled. |
| 28 | Manual Slave Select Assertion Enable | R/W | 1 | Manual Slave Select Assertion Enable. Das Setzten des Bits bewirkt, dass der Wert des HAWSPISSR direkt auf den $\overline{SS}$ pin zugewiesen wird. Die Anwendung muss nach jeder Transaktion das Bit Toggeln. Ist das Bit nicht gesetz, so toggelt die Core Logik nach jeder Transaktion. |
| 29 | CPHA | R/W | 0 | Clock Phase. Zum setzen des Transferformats. 0 = Daten sind gültig auf der ersten Flanke 1 = Daten sind gültig auf der zweiten Flanke |
| 30 | CPOL | R/W | 0 | Clock Polarity. 0 = Active high clock, SCK idles low 1 = Active low clock, SCK idles high |
| 31 | SPE | R/W | 0 | SPI System Enable. Wenn das Bit auf 1 gesetz wird, verhält sich das HAW SPI device wie folgt. 0 = SPI System disabled. MISO Eingang wird ignoriert, es wird kein Takt für SCK generiert. 1 = SPI System enabled. SCK in idle state. Ein Empfang startet sobald das Master Transaktion Inhibit bit gelöscht wird. |

Tabelle A.7.: HAW SPI Control Register (HAWSPICR) Beschreibung(C_BASEADDR + 0x4)

### A.7.3. HAW SPI Slave Select Register

Für die Paralelle Steuerung der vier SPI Slave Devices erfordert es ein $\overline{SS}$ Signal. Je nach Konfiguration des Slave Select Modes im HAWSPICR repräsentiert das aktiv Low LSB Bit im Slave Select Register das $\overline{SS}$ Signal. Die Bitpositionen des HAWSPISSR sind in Abbildung A.4 dargestellt und in der Tabelle A.8 erläutert.

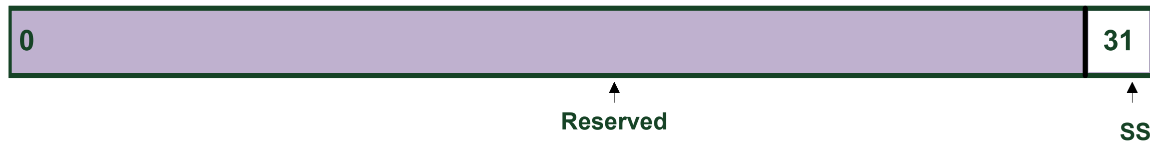



Abbildung A.4.: Slave Select Register

| Bit(s) | Name | C.A. | R.V. | Description |
|---|---|---|---|---|
| 0 - 30 | Reserved | N/A | N/A | Reserved |
| 31 | Reset | R/Write | N/A | aktiv low Slave Select bit. |

Tabelle A.8.: Slave Select Register(SRR) Beschreibung(C_BASEADDR + 0x8)

### A.7.4. SPI Data Receive Register (SPIDRR 1-4)

Die empfangenen Daten über die Signale MISO_I werden nach Abschluss eines Transfers aus den Schieberegistern in die vier Receive Register übertragen. Falls ein Register voll ist und nicht beschrieben werden kann, folgt daraus der Overrun Interrupt. Die Anzahl N der Datenbits wird mit dem Generic C_NUM_TRANSFER_BITS festgelegt(vgl. Abbildung A.5).

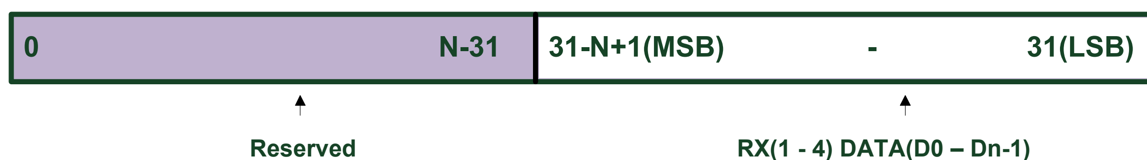



Abbildung A.5.: Data Receive Register 1-4(SPIDRR)

| Bit(s) | Name | C.A. | R.V. | Description |
|---|---|---|---|---|
| 0 - [31-N] | Reserved | N/A | N/A | Reserved |
| [31-N+1] - 31 | Rx Data(1) (D0 - Dn-1) | Read only | 0 | N-Bit SPI Empfangsdaten. N kann 8, 16 oder 32 sein. Die Bitpostion 31 repräsentiert das N-1te Datenbit. |

Tabelle A.9.: HAW SPI Data Receive Register 1-4 (SPIDRR 1-4) Beschreibung(C_BASEADDR + (0xC - 0x18))

### A.7.5. Software Reset Register (SRR)

Das Software Reset Register resetet das HAW SPI IP Cores unabhängig von allen anderen Cores im System. Zur Aktivierung des Reset ist der Wert 0x0000000A ins SR Register zu schreiben. Alle anderen Werte führen zu einer Fehlerbedingung und undefiniertem Verhalten. Ein Lesezugriff auf das SRR liefert ein undefinierten Wert.

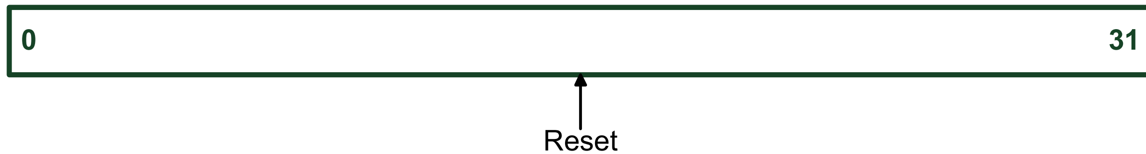



Abbildung A.6.: Software Reset Register

| Bit(s) | Name | C. A. | R.V. | Description |
|---|---|---|---|---|
| 0 - 31 | Reset | Write only Type | N/A | Die einzig erlaubte Operation ist das Schreiben des Wertes 0x0000000A ins SR Register, das den Core resetet. |

Tabelle A.10.: Software Reset Register(SRR) Beschreibung(C_BASEADDR + 0x100)

## A.8. HAW SPI IP Core Interrupt Register beschreibung

Das HAW SPI IP Core generiert bei Abschluss eines Transfers oder bei einem Überlauf ein Interrupt. Jeder Interrupt muss separat durch das IP Interrupt Enable Register(IPIER) aktiviert werden.

### A.8.1. Device Global Interrupt Enable Register (DGIER)

Das Device Global Interrupt Enable Register wird verwendet, um den Interruptausgang des HAW SPI IP Cores zum Interrupt Controller oder $\mu$Blaze zu aktivieren. Die Bitposition des DGIER sind in Abbildung A.7 dargestellt und in der Tabelle A.11 erläutert.

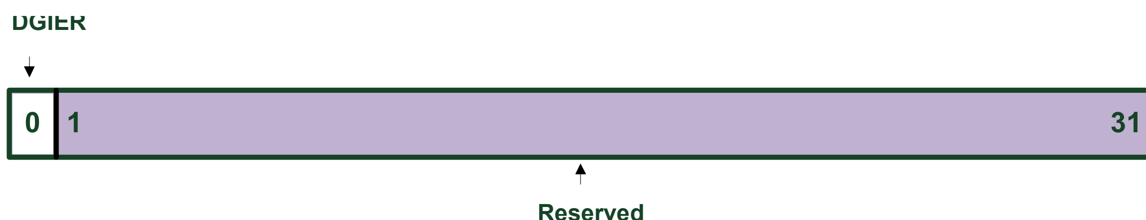



Abbildung A.7.: Device Global Interrupt Enable Register

| Bit(s) | Name | C. A. | R. V. | Description |
|---|---|---|---|---|
| 0 | GIE | R/W | 0 | Global Interrupt Enable. Aktivert den Interruptausgang des HAW SPI IP Cores zum Interrupt Controller oder $\mu$Blaze. 0 = Disabled 1 = Enabled. |
| 1 - 31 | Reserved | N/A | N/A | Reserved |

Tabelle A.11.: Device Global Interrupt Enable Register(DGIER) Beschreibung(C_BASEADDR + 0x21C)

### A.8.2. IP Interrupt Status Register (IPISR)

Das IP Interrupt Status Register (IPISR) zeigt den Status der zwei Interrupts an, die das HAW SPI IP Core generieren kann. Das IPISR ist ein Lese/Toggle on Write Register, das beim Schreiben der eins für eine Bitposition das Toggeln bewirkt(vgl. Abbildung A.8).

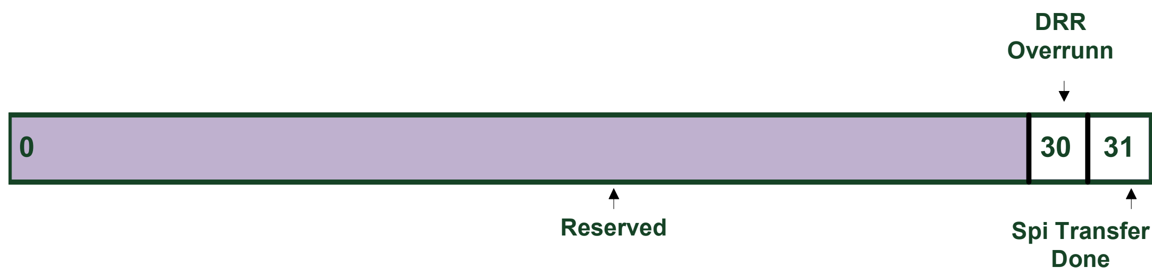



Abbildung A.8.: HAW SPI IP Interrupt Status Register

| Bit(s) | Name | C. A. | R.V. | Description |
|---|---|---|---|---|
| 0 - 29 | Reserved | N/A | N/A | Reserved |
| 30 | DRR | R/TOW | 0 | DRR Overrun. Das IPISR Bit(30) ist das DRR Overrun Bit, das für einen Takt gesetzt wird wenn ein Transfer beendet wurde und die Datenströme aus den Empfangsschieberegistern nicht in die DRR übernommen werden konnten. Das passiert wenn ein vorangegangener Transfer beendet wurde aber die Daten nicht aus den DRR über den PLB ausgelesen wurden. Das IPISR bit(30) repräsentiert die logische Oderung der vier DRR Overrun Ausgänge der einzelnen Empfangsregister. |
| 31 | transfer done | R/TOW | 0 | Spi Transfer Done. Das IPISR Bit(26) ist das Transfer Komplett Bit, das nach dem Beenden eines Empfangs für einen Takt gesetzt wird. |

Tabelle A.12.: IP Interrupt Status Register (IPISR) Beschreibung(C_BASEADDR + 0x220)

### A.8.3. IP Interrupt Enable Register (IPIER)

Das IP Interrupt Enable Register (IPIER) aktiviert die einzelnen Interrupts die in IPISR beschrieben wurden.Die Bitposition des IPIER(vgl. Abbildung A.9) werden in der Tabelle A.13 erläutert.

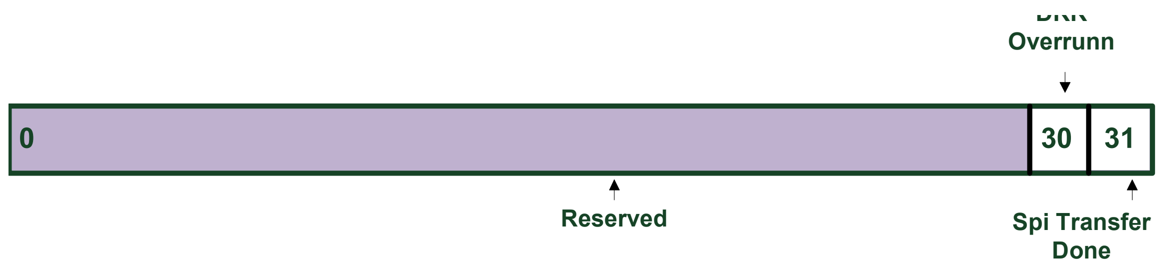



Abbildung A.9.: HAW SPI IP Interrupt Enable Register

| Bit(s) | Name | C. A. | R.V. | Description |
|---|---|---|---|---|
| 0 - 29 | Reserved | N/A | N/A | Reserved |
| 30 | DRR-Overrun | R/W | 0 | DRR Overrun 0 = Diabled 1 = Enabled. |
| 31 | transfer done | R/W | 0 | Spi Transfer Done. 0 = Diabled 1 = Enabled. |

Tabelle A.13.: IP Interrupt Enable Register (IPIER) Beschreibung(C_BASEADDR + 0x228)

## A.9. HAW SPI Master Konfiguration

Das HAW SPI IP Core arbeitet nur im Master Mode mit der Receiver-Funktionalität für vier gleichartige SPI Devices, die ein unidirektionales Interface haben(vgl. Abbildung A.10). Das CS Signal entspricht dem $\overline{SS}$ Signal. Die Vektorbreite N des $\overline{SS}$ Signals ist abhängig von der Bustopologie und den SPI Slave Devices.

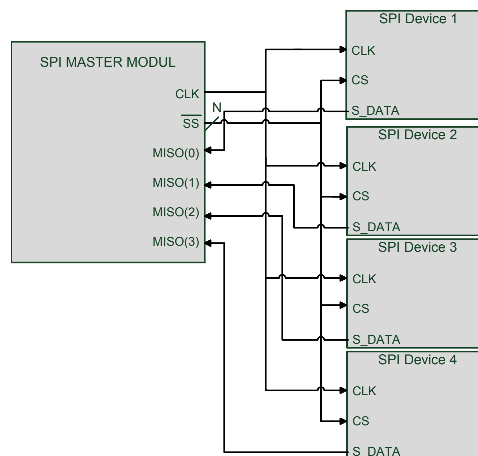



Abbildung A.10.: HAW SPI Bustopologie

Ein Vektor ist notwendig, da ein Port Ausgang für das $\overline{SS}$ Signal nicht auf mehrere Pins zugewiesen werden kann. Für ein SPI Slave Device wie z.B. das PMOD_AD1 mit zwei Analog Digital Umsetzter benötigt nur ein Pin für die Ansteuerung von zwei ICs(vgl. Abbildung A.11) durch die Verdrahtung(vgl. Abbildung A.11). Die Anzahl N der Pins für das $\overline{SS}$ Signal hängt von den SPI Slave Devices ab. Im worst case sind vier Pins und im best case ein Pin für das $\overline{SS}$ Signal erforderlich.

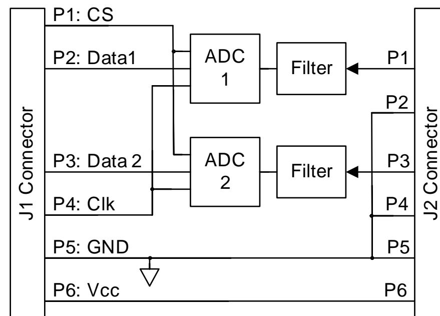



Abbildung A.11.: PMOD Analog Digital Umsetzter (vgl. Digilent (2004b))

## A.10. HAW SPI IP Core Transfer Format

Durch die Bits CPOL und CPHA im HAWSPICR werden vier Konfigurationen gewählt, die durch die Taktphase und Taktpolarität bestimmt sind(vgl. Tabelle A.10). Master und Slave müssen die selbe Konfiguration vorweisen, damit ein Transfer korrekt abläuft. Abhängig davon, ob eine fallende oder steigende Flanke von Bedeutung ist, hängt vom Idle State des Clocks ab, dass durch CPOL bestimmt wird. Durch CPHA wird der Transferformat bestimmt.

| | | CPOL | CPHA |
|---|---|---|---|
| 1 | Clock Polarität : Idle Low, Gültigkeit der Daten auf der erste Flanke | 0 | 0 |
| 2 | Clock Polarität : Idle Low, Gültigkeit der Daten auf der zweiten Flanke | 0 | 1 |
| 3 | Clock Polarität : Idle High, Gültigkeit der Daten auf der erste Flanke | 1 | 0 |
| 4 | Clock Polarität : Idle High, Gültigkeit der Daten auf der zweiten Flanke | 1 | 1 |

Tabelle A.14.: Transferformat Konfiguration des HAW SPI IP CORE

### A.10.1. CPHA = 0 Transferformat

Ein Transfer wird mit dem low aktiven $\overline{SS}$ Signal = 0 durch den Host gestartet. Das HAW SPI Master Modul hält die Clock SCK für eine halbe Taktperiode im Idle Zustand. Das SPI Slave Device muss bis die Clock ihren Idle Zustand verlässt das erste Bit auf den Bus legen.

**1. Konfiguration: CPHA = 0, CPOL = 0** Das HAW SPI Master Modul Sampelt nach jeder steigenden Flanke den Wert am MISO Eingang. Das Spi Slave Device hat eine halbe Taktperiode nach jeder fallenden Flanke um das nächste Bit zu schieben und es in der anderen hälft stabil zu halten(vgl. Abbildung A.12).

**2. Konfiguration: CPHA = 0, CPOL = 1** Das HAW SPI Master Modul Sampelt nach jeder fallenden Flanke den Wert am MISO Eingang. Das Spi Slave Device hat eine halbe Taktperiode nach jeder steigenden Flanke um das nächste Bit zu schieben und es in der anderen hälft stabil zu halten.

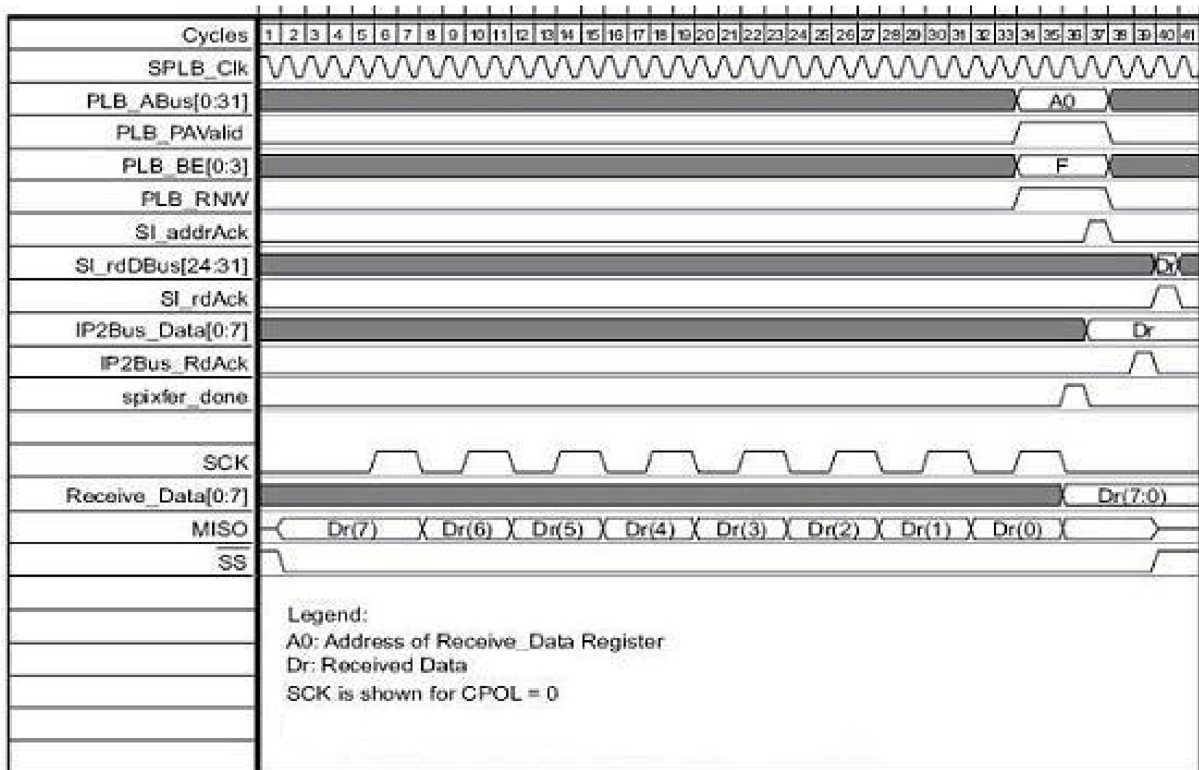



Abbildung A.12.: Timing für Receive mit CPHA = 0, CPOL = 0, SCK_RATIO = 4, C_NUM_TRANSFER_BITS = 8

## A.10.2. CPHA = 1 Transferformat

Die Daten sind auf der zweiten Flanke gültig. Der Transferstart beginnt mit der ersten Flanke von SCK und dem low aktiven $\overline{SS}$ Signal = 0.

#### 3. Konfiguration: CPHA = 1, CPOL = 0

Das HAW SPI Master Modul Sampelt den Wert am Miso Eingang nach jeder fallenden Flanke. Das SPI Slave Device hat eine halbe Taktperiode nach jeder steigenden Flanke die Daten zu schieben und in der zweiten Hälfte zum Sampeln bereitzustellen(vgl. Abbildung A.13).

#### 4. Konfiguration: CPHA = 1, CPOL = 1

Das HAW SPI Master Modul Sampelt den Wert am Miso Eingang nach jeder steigenden Flanke. Das SPI Slave Device hat eine halbe Taktperiode nach jeder fallenden Flanke die Daten zu schieben und in der zweiten Hälfte zum Sampeln bereitzustellen.

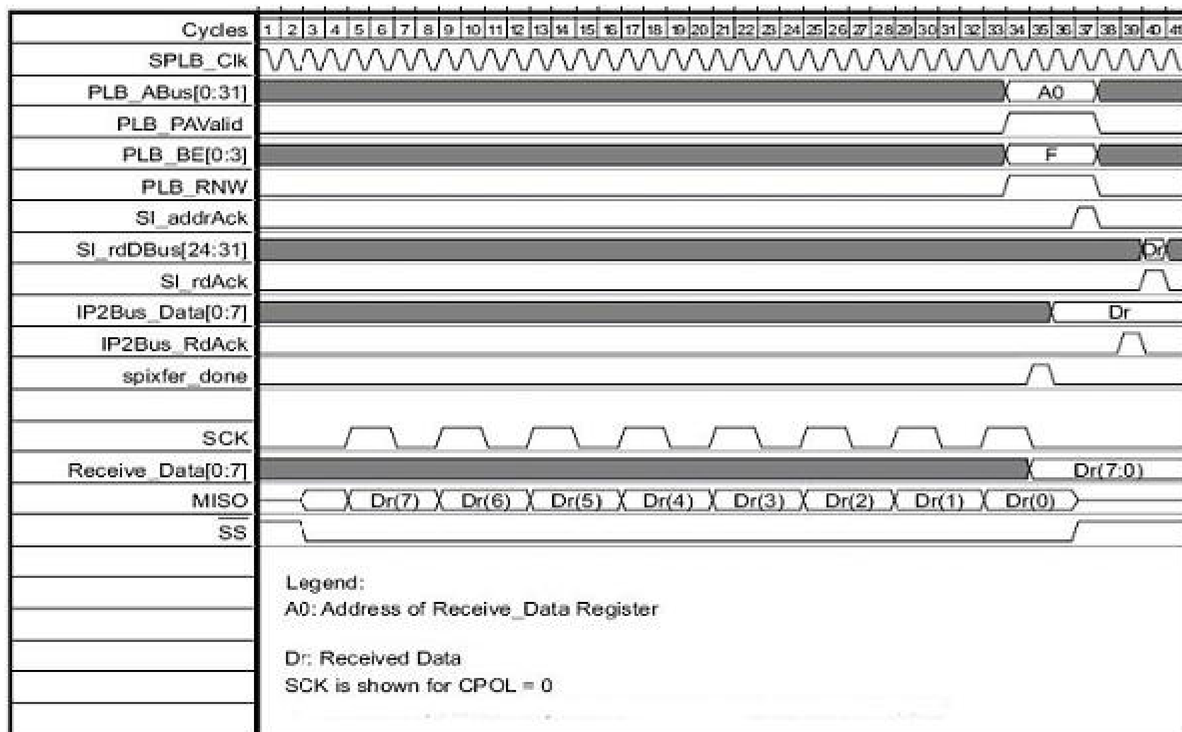



Abbildung A.13.: Timing für Receive mitCPHA = 1, CPOL = 0, SCK_RATIO = 4, C_NUM_TRANSFER_BITS = 8

## A.11. HAW SPI IP Core Slave Select Modes

Durch zwei Konfiguration erfolgt das Steuern des $\overline{SS}$ Signals. Beide Konfigurationen sind gültig für beide Transferformate. Ein SPI Device wird aktiviert, wenn das $\overline{SS}$ = 0 ist.

### A.11.1. automatische Konfiguartion

Manual Slave Select Assertion Enable Bit im HAWSPICR = 0 , so toggelt die HAW SPI Core Logik das $\overline{SS}$ Signal automatisch. Am Anfang eines Transfers weist das LSB Bit im Slave Select Register direkt auf die $\overline{SS}$ Leitung. Daher muss eine 0 im LSB stehen, da alle SPI Devices mit dem nicht SS Signal gesteuert werden. Am Ende des Transfers setzt die HAW SPI IP Core Logik das $\overline{SS}$ Signal auf 1, ohne eine Veränderung des LSB im Slave Select Register. Wenn ein neuer Transfer, beginnt wird wieder das LSB im Slave Select Register der SS Leitung zugewiesen. Diese Konfiguration bietet einen kontinuierlichen Empfang der Daten, ohne das die Anwendung das $\overline{SS}$ Signal Toggeln muss. Für einen neuen Transfer muss immer das $\overline{SS}$ Signal getoggelt werden.

### A.11.2. manuelle Konfiguartion

Manual Slave Select Assertion Enable Bit im HAWSPICR = 1, so muss die Anwendung das $\overline{SS}$ Signal Toggeln. Das LSB Bit im Slave Select Register entspricht dem $\overline{SS}$ Signal, das den N Pins

zugewiesen wird. Für jeden Transferstart muss eine 0 ins Slave Select Register geschrieben werden und am Ende eines Tranfers eine 1.

## A.12. HAW SPI Register Flow Beschreibung

Zur Übersicht einer HAW SPI Bus Transaktion sind hier einige Beispiele für den Register Flow des HAW SPI IP Cores dargestellt.

### A.12.1. HAW SPI Master mit manueller Slave Select Konfiguration(Interrupt)

Diese Konfiguration erlaubt den parallelen Empfang von N byte/half-word/word Elementen für vier MISO Ports durch eine Interrupt Steuerung.

1. Starte im Anfangszustand mit einen Soft Reset durch ein Schreibbefehl ins Software Reset Register(SRR)(C_BASEADDR + 0x100) mit dem Wert 0x0000000A.

2. Den Globalen Interrupt Ausgang vom Core zum Interrrupt Controller/ $\mu$ durch setzen des MSB im DGIER(C_BASEADDR + 0x21C) freigeben

3. Das Spi Transfer Done- und DRR Overrun Bit im IPISR(Beschreibung(C_BASEADDR + 0x220) setzten.

4. Transferformat durch die Bits CPOL und CPHA im HAWSPICR(C_BASEADDR + 0x0) setzten.

5. Die manuelle Konfiguration für das $\overline{SS}$ Signal ist die Standard Konfiguration

6. Das HAW Spi System enable Bit im HAWSPICR setzten.

7. Löschen des LSB im SPISSR, damit das $\overline{SS}$ Signal auf low gesetz werden kann

8. Das Löschen des Master Transaktion Inhibt Bits im HAWSPICR zum Starten eines Transfers

9. Warten bis der Spi Transfer Done- oder DRR Overrun Interrupt auftritt.

10. In der ISR das Master Transaktion Inhibt Bit und das LSB im SPISSR wieder sezten. Bei einem SPIDRR Overrun Interrupt sind die Daten aus den SPIDRR 1-4 nicht verwertbar. Durch ein Lesebefehl die Registerinhalte leeren und bei Punkt 11 fortsetzten. Bei einem Spi Transfer Done Interrupt sind die Daten in den DRR Register verwertbar und müssen alle ausgelesen werden um einen neuen Transfer zu starten. Löschen des Interrupt durch

schreiben des Register IPSR.Das IPSR ist ein Toggle on Write und somit kann der Wert 0 oder 1 betragen.

11. Wiederholen der vier vorangegangenen Schritte bis alle N Elemente Empfangen wurden.

### A.12.2. HAW SPI Master mit manueller Slave Select Konfiguration(polling)

Diese Konfiguration ist für den parallelen Empfang von N byte/half-word/word Elementen für vier MISO Ports durch eine polling Steuerung.

1. Starte im Anfangszustand mit einen Soft Reset durch ein Schreibbefehl ins Software Reset Register(SRR)(C_BASEADDR + 0x100) mit dem Wert 0x0000000A.

2. Transferformat durch die Bits CPOL und CPHA im HAWSPICR(C_BASEADDR + 0x0) setzten.

3. Die manuelle Konfiguration für das $\overline{SS}$ Signal ist die Standard Konfiguration

4. Das HAW Spi System enable Bit im HAWSPICR setzten.

5. Löschen des LSB im SPISSR, damit das $\overline{SS}$ Signal auf low gesetz werden kann

6. Das Löschen des Master Transaktion Inhibt Bits im HAWSPICR zum Starten eines Transfers

7. Pollen bis ein Transfer abgeschlossen ist durch Maskierung der RXx_FULL Bits im HAWSPISR(C_BASEADDR + 0x4)

8. Das Master Transaktion Inhibt Bit und das LSB im SPISSR wieder sezten. Durch Maskierung der RXx_Overrun Bits im HAWSPISR prüfen ob ein Overrun stattgefunden hat. Die Daten aus den SPIDRR lesen.
9. Wiederholen der vier vorangegangenen Schritte bis alle N Elemente Empfangen wurden.

## A.13. Ressourcenverbrauch

Das Timing und der Verbrauch an Hardware Ressourcen hängt vom Systemdesign ab. Die folgende Auflistung bezieht sich auf ein Spartan 3E FPGA und varriert bei anderen FPGAs.

| Parameter | | Ressourcen | | | Performance |
|---|---|---|---|---|---|
| SCK_RATIO | TRANSFER_BITS | Slices | Slice FFs | LUTs | Fmax(MHZ) |
| 4 | 8 | 226 | 294 | 271 | 136.724 |
| 16 | 8 | 227 | 297 | 274 | 146.972 |
| 32 | 8 | 228 | 298 | 275 | 146.972 |
| 4 | 16 | 275 | 395 | 280 | 146.972 |
| 16 | 16 | 277 | 398 | 283 | 146.972 |
| 32 | 16 | 289 | 399 | 285 | 146.972 |
| 4 | 32 | 403 | 603 | 347 | 143.431 |
| 16 | 32 | 405 | 606 | 350 | 143.431 |
| 32 | 32 | 418 | 607 | 354 | 146.972 |

Tabelle A.15.:

## A.14. Low- und High Level Treiber

Die Low Level Treiber wie z.B. Adressmasken Lese/Schreib Makros für Register wurden vom EDK erstellt. Für die Ansteuerung des Busses mit dem ADCS7476 wurden die High Level Treiber Funktion und Bit-Masken implementiert. Diese Software Module wurden im FreeRTOS eingebunden.

Listing A.1: Low Level Treiber des HAW SPI IP CORE : Makrofunktion,Adress- und Bit Masken

```
#ifndef HAW_SPI_H
#define HAW_SPI_H

/**************************** Include Files ************/

#include "xbasic_types.h"
#include "xstatus.h"
#include "xio.h"

/************************* Constant Definitions ********/

/**
 *HAW_SPI Number of Transfer Bits to Mask
 */
#define HAW_SPI_Number_Transfer_Bits_Mask 0x00000fff

/**
 *  HAW_SPI Register  Offsets
 * -- CR   : Controll       Register  HAWSPI
 * -- SR   : Status         Register  HAWSPI
 * -- SS   : Slave Select   Register  HAWSPI
 * -- RCV1 : Receive 1      Register  HAWSPI
 * -- RCV2 : Receive 2      Register  HAWSPI
 * -- RCV3 : Receive 3      Register  HAWSPI
 * -- RCV4 : Receive 4      Register  HAWSPI
 */
#define HAW_SPI_USER_SLV_SPACE_OFFSET (0x00000000)
#define HAW_SPI_CR_OFFSET (HAW_SPI_USER_SLV_SPACE_OFFSET

#define HAW_SPI_SR_OFFSET (HAW_SPI_USER_SLV_SPACE_OFFSET
```

```
#define HAW_SPI_SS_OFFSET (HAW_SPI_USER_SLV_SPACE_OFFSET

#define HAW_SPI_RCV1_OFFSET (HAW_SPI_USER_SLV_SPACE_OFFSET

#define HAW_SPI_RCV2_OFFSET (HAW_SPI_USER_SLV_SPACE_OFFSET

#define HAW_SPI_RCV3_OFFSET (HAW_SPI_USER_SLV_SPACE_OFFSET

#define HAW_SPI_RCV4_OFFSET (HAW_SPI_USER_SLV_SPACE_OFFSET


/**
 * HAW_SPI Control Register Masks
 */

/*< System enable */
#define HAW_SPI_CR_ENABLE_MASK              0x1

/*< Clock polarity high or low */
#define HAW_SPI_CR_CLK_POLARITY_MASK    0x2

/*< Clock phase 0 or 1 */
#define HAW_SPI_CR_CLK_PHASE_MASK                 0x4

/*< Manual slave sel assert */
#define HAW_SPI_CR_MANUAL_SS_MASK                 0x8

/*< Master transaction inhibit */
#define HAW_SPI_CR_TRANS_INHIBIT_MASK    0x10

 /**
 * HAW_SPI Receive Register select Masks
 */
#define HAW_SPI_RCV1        0x1  /*  1 Receive Register */
#define HAW_SPI_RCV2         0x2  /*  2 Receive Register */
#define HAW_SPI_RCV3             0x3  /*  3 Receive Register */
#define HAW_SPI_RCV4             0x4  /*  4 Receive Register */

 /**
 * HAW_SPI Control Register Masks
 */
 /* Receive Reg1 is empty */
#define HAW_SPI_SR_RX1_EMPTY_MASK                  0x1
 /* Receive Reg1 is full */

#define HAW_SPI_SR_RX1_FULL_MASK                   0x2
/* Receive Reg2 is empty */

#define HAW_SPI_SR_RX2_EMPTY_MASK                  0x4
/* Receive Reg2 is full */

#define HAW_SPI_SR_RX2_FULL_MASK                   0x8
 /* Receive Reg3 is empty */

#define HAW_SPI_SR_RX3_EMPTY_MASK                 0x10
/* Receive Reg3 is full */
```

```
#define HAW_SPI_SR_RX3_FULL_MASK                    0x20
/* Receive Reg4 is empty */

#define HAW_SPI_SR_RX4_EMPTY_MASK                   0x40
/* Receive Reg4 is full */

#define HAW_SPI_SR_RX4_FULL_MASK                    0x80
/* Receive Reg1 Overrun */

#define HAW_SPI_SR_RX1_DRR_Overrun_MASK          0x100
/* Receive Reg2 Overrun */

#define HAW_SPI_SR_RX2_DRR_Overrun_MASK          0x200
 /* Receive Reg3 Overrun */

#define HAW_SPI_SR_RX3_DRR_Overrun_MASK          0x400
/* Receive Reg4 Overrun */

#define HAW_SPI_SR_RX4_DRR_Overrun_MASK          0x800

  /**
 *  HAW_SPI SS Mask
 */

  /* Not SS Signal => SS(CS) goes High */
#define HAW_SPI_SS_Set_Value                     0x00000000

/* Not SS Signal => SS(CS) goes Low */
#define HAW_SPI_SS_Clear_Value              0x00000001

/**
 * Software Reset Space Register Offsets
 * -- RST : software reset register
 */
#define HAW_SPI_SOFT_RST_SPACE_OFFSET (0x00000100)
#define HAW_SPI_RST_REG_OFFSET (HAW_SPI_SOFT_RST_SPACE_OFFSET
                                               + 0x00000000)

/**
 * Software Reset Masks
 * -- SOFT_RESET : software reset
 */
#define SOFT_RESET (0x0000000A)

/**
 * Interrupt Controller Space Offsets
 * -- INTR_DGIER : device global interrupt enable register
 * -- INTR_ISR   : ip interrupt status register
 * -- INTR_IER   : ip interrupt enable register
 */
#define HAW_SPI_INTR_CNTRL_SPACE_OFFSET (0x00000200)
#define HAW_SPI_INTR_DGIER_OFFSET (HAW_SPI_INTR_CNTRL_SPACE_OFFSET
                                                    + 0x0000001C)
#define HAW_SPI_INTR_IPISR_OFFSET (HAW_SPI_INTR_CNTRL_SPACE_OFFSET
                                                    + 0x00000020)
#define HAW_SPI_INTR_IPIER_OFFSET (HAW_SPI_INTR_CNTRL_SPACE_OFFSET
```

```
                                                        + 0x00000028)

/**
 * Interrupt Controller Masks
 * -- INTR_GIE_MASK  : global interrupt enable
 */
#define INTR_GIE_MASK (0x80000000UL)
#define INTR_DRRIE_MASK (0x00000001)
#define INTR_DRR_FULL_MASK (0x00000001)
#define DRR_INTR (0x00000001)

/************* Macros (Inline Functions) Definitions ********/

/**
 *
 * Write a value to a HAW_SPI register. A 32 bit write is
 *
 * If the component is implemented in a smaller width, only
 * the least significant data is written.
 *
 * @param   BaseAddress is the base address of the HAW_SPI
 *
 * @param   RegOffset is the register offset from the base
 *
 * @param   Data is the data written to the register.
 *
 * @return  None.
 *
 * @note
 * C-style signature:
 *      void HAW_SPI_mWriteReg(Xuint32 BaseAddress,
 *                             unsigned RegOffset,   Xuint32 Data)
 *
 */
#define HAW_SPI_mWriteReg(BaseAddress, RegOffset, Data) \
        XIo_Out32((BaseAddress) + (RegOffset), (Xuint32)(Data))

/**
 *
 * Read a value from a HAW_SPI register. A 32 bit read is
 * performed. If the component is implemented in a smaller width,
 * only the least significant data is read from the register. The
 * most significant data will be read as 0.
 *
 * @param   BaseAddress is the base address of the HAW_SPI device.
 * @param   RegOffset is the register offset from the base to
 *
write to.
 * @return  Data is the data from the register.
 *
 * @note
 * C-style signature:
 *      Xuint32 HAW_SPI_mReadReg(Xuint32 BaseAddress, unsigned RegOffset)
 *
 */
#define HAW_SPI_mReadReg(BaseAddress, RegOffset) \
        XIo_In32((BaseAddress) + (RegOffset))
```

```
/**
 *
 * Write/Read 32 bit value to/from HAW_SPI user logic slave registers.
 *
 * @param   BaseAddress is the base address of the HAW_SPI device.
 * @param   RegOffset is the offset from the slave register to write
 *
 * @param   Value is the data written to the register.
 *
 * @return  Data is the data from the user logic slave register.
 *
 * @note
 * C-style signature:
 *      void HAW_SPI_mWriteSlaveRegn(Xuint32 BaseAddress,
 *                                   unsigned RegOffset, Xuint32 Value)
 *      Xuint32 HAW_SPI_mReadSlaveRegn(Xuint32 BaseAddress,

 *
 */
#define HAW_SPI_mWriteCR(BaseAddress, RegOffset, Value) \
        XIo_Out32((BaseAddress) + (HAW_SPI_CR_OFFSET) + (RegOffset),

#define HAW_SPI_mWriteSS(BaseAddress, RegOffset, Value) \
        XIo_Out32((BaseAddress) + (HAW_SPI_SS_OFFSET) + (RegOffset),



#define HAW_SPI_mReadCR(BaseAddress, RegOffset) \
        XIo_In32((BaseAddress) + (HAW_SPI_CR_OFFSET) + (RegOffset))
#define HAW_SPI_mReadSR(BaseAddress, RegOffset) \
        XIo_In32((BaseAddress) + (HAW_SPI_SR_OFFSET) + (RegOffset))
#define HAW_SPI_mReadSS(BaseAddress, RegOffset) \
        XIo_In32((BaseAddress) + (HAW_SPI_SS_OFFSET) + (RegOffset))
#define HAW_SPI_mReadRCV1(BaseAddress, RegOffset) \
        XIo_In32((BaseAddress) + (HAW_SPI_RCV1_OFFSET) + (RegOffset))
#define HAW_SPI_mReadRCV2(BaseAddress, RegOffset) \
        XIo_In32((BaseAddress) + (HAW_SPI_RCV2_OFFSET) + (RegOffset))
#define HAW_SPI_mReadRCV3(BaseAddress, RegOffset) \
        XIo_In32((BaseAddress) + (HAW_SPI_RCV3_OFFSET) + (RegOffset))
#define HAW_SPI_mReadRCV4(BaseAddress, RegOffset) \
        XIo_In32((BaseAddress) + (HAW_SPI_RCV4_OFFSET) + (RegOffset))


/**
 *
 * Reset HAW_SPI via software.
 *
 * @param   BaseAddress is the base address of the HAW_SPI device.
 *
 * @return  None.
 *
 * @note
 * C-style signature:
 *      void HAW_SPI_mReset(Xuint32 BaseAddress)
 *
```

```
 */
#define HAW_SPI_mReset(BaseAddress) \
        XIo_Out32((BaseAddress)+(HAW_SPI_RST_REG_OFFSET), SOFT_RESET)

/************************ Function Prototypes **************/
void HAW_SPI_Initialize(void * baseaddr_p);

Xuint16 HAW_SPI_getReceiveReg(void * baseaddr_p, Xuint8 RegIndex);

void HAW_SPI_Start(void * baseaddr_p);

void HAW_SPI_Stop(void * baseaddr_p);

void HAW_SPI_Set_SS(void * baseaddr_p);

void HAW_SPI_Clear_SS(void * baseaddr_p);

/**
 *
 * Enable all possible interrupts from HAW_SPI device.
 *
 * @param   baseaddr_p is the base address of the HAW_SPI device.
 *
 * @return  None.
 *
 * @note    None.
 *
 */
void HAW_SPI_EnableInterrupt(void * baseaddr_p);

/**
 *
 * Example interrupt controller handler.
 *
 * @param   baseaddr_p is the base address of the HAW_SPI device.
 *
 * @return  None.
 *
 * @note    None.
 *
 */
void HAW_SPI_Intr_DefaultHandler(void * baseaddr_p);

/**
 *
 * Run a self-test on the driver/device. Note this may be
 * a destructive test if resets of the device are performed.
 *
 * If the hardware system is not built correctly, this function
 * may never return to the caller.
 *
 * @param  baseaddr_p is the base address of the HAW_SPI instance
 *
 * @return
 *
 *     - XST_SUCCESS   if all self-test code passed
 *     - XST_FAILURE   if any self-test code failed
```

```
 *
 * @note     Caching must be turned off for this function to work.
 * @note     Self test may fail if data memory and device are not
 * on the same bus.
 *\
XStatus HAW_SPI_SelfTest(void * baseaddr_p);

#endif // HAW_SPI_H
```

Listing A.2: High Level Treiber HAW SPI IP CORE für das ADCS7476

```
/**************************** Include Files *************/

#include "haw_spi.h"

/************************** Function Definitions *********/
/**
 *
 * Initialize The SPI System with Clock Polarity and SPI System enable
 *
 * @param    baseaddr_p is the base address of the HAW_SPI device.
 *
 * @return   None.
 *
 * @note     None.
 *
 */
void HAW_SPI_Initialize(void * baseaddr_p)
{
  Xuint32 baseaddr, ctrlReg;
  baseaddr = (Xuint32) baseaddr_p;
  ctrlReg   = HAW_SPI_mReadCR(baseaddr, 0);
  HAW_SPI_mWriteCR(baseaddr, 0, ctrlReg | HAW_SPI_CR_CLK_POLARITY_MASK | HAW_SPI_CR_ENABLE_MASK);

}

/**
 *
 * Start The SPI System by clearing Master Tranaction Inhibt bit
 *
 * @param    baseaddr_p is the base address of the HAW_SPI device.
 *
 * @return   None.
 *
 * @note     None.
 *
 */
void HAW_SPI_Start(void * baseaddr_p)
{
  Xuint32 baseaddr, ctrlReg;
  baseaddr = (Xuint32) baseaddr_p;
  ctrlReg   = HAW_SPI_mReadCR(baseaddr, 0);
  HAW_SPI_mWriteCR(baseaddr, 0, ctrlReg & ~HAW_SPI_CR_TRANS_INHIBIT_MASK);
}

/**
 *
 * Stop The SPI System by setting Master Tranaction Inhibt bit
```

```
 *
 * @param    baseaddr_p is the base address of the HAW_SPI device.
 *
 * @return   None.
 *
 * @note     None.
 *
 */
void HAW_SPI_Stop(void * baseaddr_p)
{
  Xuint32 baseaddr,ctrlReg;
  baseaddr = (Xuint32) baseaddr_p;
  ctrlReg  = HAW_SPI_mReadCR(baseaddr, 0);
  HAW_SPI_mWriteCR(baseaddr, 0, ctrlReg | HAW_SPI_CR_TRANS_INHIBIT_MASK);
}

/**
 *
 * set the CS Signal to 0 for not SS Signal => not SS goes High
 *
 * @param    baseaddr_p is the base address of the HAW_SPI device.
 *
 * @return   None.
 *
 * @note     None.
 *
 */
void HAW_SPI_Set_SS(void * baseaddr_p)
{
  Xuint32 baseaddr,ctrlReg;
  baseaddr = (Xuint32) baseaddr_p;
  HAW_SPI_mWriteSS(baseaddr, 0, HAW_SPI_SS_Set_Value);
}

/**
 *
 * Clears the CS Signal to 1 for not SS Signal => not SS goes Low
 *
 * @param    baseaddr_p is the base address of the HAW_SPI device.
 *
 * @return   None.
 *
 * @note     None.
 *
 */
void HAW_SPI_Clear_SS(void * baseaddr_p)
{
  Xuint32 baseaddr,ctrlReg;
  baseaddr = (Xuint32) baseaddr_p;
  HAW_SPI_mWriteSS(baseaddr, 0, HAW_SPI_SS_Clear_Value);
}

/**
 *
 * gets the Receive Value
 *
 * @param    baseaddr_p is the base address of the HAW_SPI device.
```

```
 *          RegIndex is the digit for the selected Register
 *
 * @return  The Register Value or -1 by wrong RegIndex.
 *
 * @note    None.
 *
 */
Xuint16 HAW_SPI_getReceiveReg(void * baseaddr_p, Xuint8 RegIndex)
{
  Xuint32 baseaddr, registerValue = -1;
  baseaddr = (Xuint32) baseaddr_p;
  switch(RegIndex){
        case HAW_SPI_RCV1 : registerValue = HAW_SPI_mReadRCV1(baseaddr, 0);break;
        case HAW_SPI_RCV2 : registerValue = HAW_SPI_mReadRCV2(baseaddr, 0);break;
        case HAW_SPI_RCV3 : registerValue = HAW_SPI_mReadRCV3(baseaddr, 0);break;
        case HAW_SPI_RCV4 : registerValue = HAW_SPI_mReadRCV4(baseaddr, 0);break;
         default :;
  }
  return registerValue & HAW_SPI_Number_Transfer_Bits_Mask;
}

/**
 *
 * Enable all possible interrupts from HAW_SPI device.
 *
 * @param   baseaddr_p is the base address of the HAW_SPI device.
 *
 * @return  None.
 *
 * @note    None.
 *
 */
void HAW_SPI_EnableInterrupt(void * baseaddr_p)
{
  Xuint32 baseaddr,intrReg;

  baseaddr = (Xuint32) baseaddr_p;
  /*
   * Enable DRR Interrupt source from user logic.
   */
HAW_SPI_mWriteReg(baseaddr, HAW_SPI_INTR_IPIER_OFFSET, INTR_DRRIE_MASK);

  /*
   * Enable all possible interrupt sources from device.
   */

  /*
   * Set global interrupt enable.
   */
  HAW_SPI_mWriteReg(baseaddr, HAW_SPI_INTR_DGIER_OFFSET, INTR_GIE_MASK);

}
void HAW_SPI_Intr_DefaultHandler(void * baseaddr_p)
{}
```

# B. C-CODE Übersicht

## B.1. C-Code zur Untersuchung der maximalen Abtastfrequenz durch SPI Transfers und Signalrekonstruktion

Für die Zeitmessung wurden die auskommentierten Timer Einstellung verwendet, um einen periodischen Ausgangssignal des GPIO zu haben. Für die Signalrekonstruktion muss nach einem Timer Interrupt der Timer sofort wieder weiter laufen für einen konstanten Abtast Intervall.
Für den DAC wurden die Treiberstrukturen vom Xilinx xps SPI verwendet. Mit dem Anlegen einer SPI Slave Device Instanz Struktur wird der DAC gesteuert. Für den ADC wurden die High Level Treiber vom HAW SPI verwendet.

```
/*Die Timer ISR die mit der vorgegebenen Frequenz einen SPI Bustransfer
  startet um den ADC Wert zu Sampeln.*/
void timer_int_handler(void * ppt) {
  Xuint32 csr;

  /*Auslesen des Timer Status Register, um zu sehen ob ein Timer
        Interrupt aufgetreten */
  csr = XTmrCtr_mGetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR,0);

  /* Wenn ja starte ein SPI Bustransfer mit dem ADC */
  if (csr & XTC_CSR_INT_OCCURED_MASK) {
      /* Clear the timer interrupt */
        XTmrCtr_mSetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR,0,csr);

      /*Begin Zeitmessung */
      //XGpio_mSetDataReg(XPAR_XPS_GPIO_0_BASEADDR, 1, 1);

     /*Toggeln des SS(CS) Signals und starten des nächsten Transfer
        durch löschen des Master Transaktion Inhibit Bit*/
        HAW_SPI_Clear_SS((void *)XPAR_HAW_SPI_0_BASEADDR);
        HAW_SPI_Set_SS((void *)XPAR_HAW_SPI_0_BASEADDR);
        HAW_SPI_Start((void *)XPAR_HAW_SPI_0_BASEADDR);

     /* Löschen der timer interrupt */
        XTmrCtr_mSetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR,0,csr);


     /* Laden des LoadReg ins CountReg für die Zeitmessung*/
      //XTmrCtr_mSetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR, 0,
      //XTC_CSR_INT_OCCURED_MASK | XTC_CSR_LOAD_MASK );

     /* Starte den timers für die Zeitmessung*/
      //XTmrCtr_mSetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR, 0,
      //           XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK
```

```
        //                                      | XTC_CSR_DOWN_COUNT_MASK);
    }

}
```

/*In der ISR des HAW SPI wird der Sampel Wert ausgelesen und über den Xilinx XPS SPI ein DAC Transfer gestartet.*/

```
void haw_spi_int_handler(void *baseaddr_p) {
    Xuint32 IntrStatus,baseaddr,i,j;
    baseaddr = (Xuint32)baseaddr_p;
    IntrStatus = HAW_SPI_mReadReg(baseaddr,
                  HAW_SPI_INTR_IPISR_OFFSET);

    /* Wenn HAW SPI Transfer Done aufgetreten ist auslesen des*/
    if ( (IntrStatus & INTR_DRR_FULL_MASK) == DRR_INTR ) {
       /*Stopen des Busses durch setzten des Master Transaktion
        Inhibit Bit*/
        HAW_SPI_Stop((void *)XPAR_HAW_SPI_0_BASEADDR);

       /*Auslesen des Messwertes*/
        measurement =  HAW_SPI_getReceiveReg((void *)
                  XPAR_HAW_SPI_0_BASEADDR,HAW_SPI_RCV1);
       /*Ende Zeitmessung */
        //XGpio_mSetDataReg(XPAR_XPS_GPIO_0_BASEADDR, 1, 0);

       /*Starten eines DAC Transfers um das Signal zu rekonstruieren*/
       XSpi_Transfer(&dacSlave,&measurement,&measurement,2);

       /*Clearen des Interrupt Request*/
       HAW_SPI_mWriteReg(XPAR_HAW_SPI_0_BASEADDR, HAW_SPI_INTR_IPISR_OFFSET, DRR_INTR);
    }
}

int main(void) {
    /*Setzen des HAW_SPI transfer done Interrupt*/
    HAW_SPI_EnableInterrupt((void *)XPAR_HAW_SPI_0_BASEADDR);

    /*Setzen des SPI_EN and CPOL idle high*/
    HAW_SPI_Initialize((void *)XPAR_HAW_SPI_0_BASEADDR);

    /*Setzen des SS(CS) Signals*/
    HAW_SPI_Set_SS((void *)XPAR_HAW_SPI_0_BASEADDR);

    /*Registrieren der Timer und HAW SPI BUS ISR Handler
       für den Interrupt Controller*/
    XIntc_RegisterHandler(XPAR_XPS_INTC_0_BASEADDR,
            XPAR_XPS_INTC_0_HAW_SPI_0_IP2INTC_IRPT_INTR,
            haw_spi_int_handler, // ISR name
            (void *)XPAR_HAW_SPI_0_BASEADDR);

    XIntc_RegisterHandler(XPAR_XPS_INTC_0_BASEADDR,
            XPAR_XPS_INTC_0_XPS_TIMER_0_INTERRUPT_INTR,
            timer_int_handler, // ISR name
            (void *) XPAR_XPS_TIMER_0_BASEADDR);

    /*Aktivieren des Interrrupt Controller*/
```

```
        XIntc_mMasterEnable(XPAR_XPS_INTC_0_BASEADDR);

      /*Anzahl der Timer Ticks mit 50MHz PLB Takt bis zum Interrupt*/
      XTmrCtr_mSetLoadReg(XPAR_XPS_TIMER_0_BASEADDR, 0, 521 - 2);

      /* Laden des LoadReg ins CountReg */
      XTmrCtr_mSetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR,0,
                XTC_CSR_INT_OCCURED_MASK | XTC_CSR_LOAD_MASK );

      /*Aktivieren timer und HAW SPI interrupt im interrupt C.*/
      XIntc_mEnableIntr(XPAR_XPS_INTC_0_BASEADDR, XPAR_XPS_TIMER_0_INTERRUPT_MASK
                                              |XPAR_HAW_SPI_0_IP2INTC_IRPT_MASK);

    /*initialisiere ein xilinx xps SPI Bus für den DAC*/
    XSpi_Initialize(&dacSlave,XPAR_HAW_SPI_0_DEVICE_ID);

    /*Master SPI Device und Idle High Clock für DAC*/
    XSpi_SetOptions(&dacSlave, XSP_MASTER_OPTION | XSP_CLK_ACTIVE_LOW_OPTION);

    /*starte den SPI BUS für den DAC Slave*/
    XSpi_Start(&dacSlave);

    /*Zum setzten des SS Wertes in der Xspi Struktur */
     XSpi_SetSlaveSelect(&dacSlave,0x0000001);

      /* Starte den timer für die Zeitmessung ohne Auto Reload*/
   //XTmrCtr_mSetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR,0,XTC_CSR_ENABLE_TMR_MASK
                              //| XTC_CSR_ENABLE_INT_MASK | XTC_CSR_DOWN_COUNT_MASK);

/* Starte den timer*/
      XTmrCtr_mSetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR, 0, XTC_CSR_ENABLE_TMR_MASK
                                |XTC_CSR_ENABLE_INT_MASK | XTC_CSR_AUTO_RELOAD_MASK
                                                      | XTC_CSR_DOWN_COUNT_MASK);



    /*Initialisieren eines GPIO\'s für die Zeitmessung*/
    XGpio_Initialize(&out, XPAR_XPS_GPIO_0_DEVICE_ID); //
    XGpio_SetDataDirection (&out,1, 0x00000000); //

    microblaze_enable_interrupts();
     while(1);

     /*Sollte niemals hier gelangen*/
     return 0;
}
```

## B.2. C-Code FreeRTOS Demoanwendung

```
/* FreeRTOS Scheduler includes. */
#include <stdio.h>
#include <xintc.h>
#include <xtmrctr.h>
#include "FreeRTOS.h"
#include "task.h"
```

```
#include "xgpio_l.h"
#include "xgpio.h"
#include "net1.h"
#include "haw_spi.h"
#include <xbasic_types.h>
#include <xparameters.h>
#include "traceconf.h"

/* Trace Funktion der vom Scheduler beim Kontext Wechsel ausgeführt wird*/
#define traceTASK_SWITCHED_IN() vSetAnalogueOutput( 0, ( int ) pxCurrentTCB->pxTaskTag )

void dispLED( Xuint32 Val, Xuint8 DP );
XGpio led;
Xuint8 Digit[] =
{
0xC0,
0xF9,
0xA4,
0xB0,
0x99,
0x92,
0x82,
0xF8,
0x80,
0x90
};


/* Alle 100ms werden die Daten aus der Read FIFO gelesen und
   über die serielle Schnittstelle am Hyperterminal vom Host PC ausgegeben */
static portTASK_FUNCTION( vReadFifo, pvParameters ) {

unsigned int wert;
portTickType xLastWakeTime;
/* Eindeutiger Identifer zum setzten der analogen Ausgangsspannung für die Trace
Funktion */
      vTaskSetApplicationTaskTag( NULL, ( void * ) 1 );

      while(1) {
                 xLastWakeTime = xTaskGetTickCount();
                 if(NET1_mReadFIFOEmpty(XPAR_NET1_0_BASEADDR) == 0){
                        wert = NET1_mReadFromFIFO(XPAR_NET1_0_BASEADDR,0) >> 24;
                        xil_printf("FIFO_LESE_WERT:%d\r\n",wert);
                 }

                  vTaskDelayUntil(&xLastWakeTime, 100);
          }
}


/* Startet alle 300ms einen HAW SPI Transfer und schreibt die Messdaten ins Write
   FIFO, die vom HOST PC ausgelesen werden*/
static portTASK_FUNCTION( vHawWriteFifo, pvParameters ) {

volatile Xuint16 value=0;
portTickType xLastWakeTime;
unsigned char wert = 0;
```

```
int haw_sr;
    NET1_mResetWriteFIFO(XPAR_NET1_0_BASEADDR);
    vTaskSetApplicationTaskTag( NULL, ( void * ) 2 );

    while(1) {
            xLastWakeTime = xTaskGetTickCount();
            vTaskDelayUntil(&xLastWakeTime, 300);
            HAW_SPI_Set_SS((void *)XPAR_HAW_SPI_0_BASEADDR);
            HAW_SPI_Start((void *)XPAR_HAW_SPI_0_BASEADDR);

            do{
              haw_sr =HAW_SPI_mReadReg(XPAR_HAW_SPI_0_BASEADDR,HAW_SPI_SR_OFFSET);
              }while( haw_sr & HAW_SPI_SR_RX1_EMPTY_MASK == 1);

            HAW_SPI_Stop((void *)XPAR_HAW_SPI_0_BASEADDR);
            HAW_SPI_Clear_SS((void *)XPAR_HAW_SPI_0_BASEADDR);
            value = HAW_SPI_getReceiveReg((void *)XPAR_HAW_SPI_0_BASEADDR,
                                                    HAW_SPI_RCV1);
            xil_printf("Messwert in mV : %d in cm\r\n", (value*805)/1000);

            if(NET1_mWriteFIFOVacancy(XPAR_NET1_0_BASEADDR) > 0){
                wert = (value & 0xff00) > 8;
                NET1_mWriteToFIFO(XPAR_NET1_0_BASEADDR,0,wert);
                wert =  value & 0x00ff;
                NET1_mWriteToFIFO(XPAR_NET1_0_BASEADDR,0,wert);
              }
            }
}


/* Alle 50ms wird auf dem Hyperterminal ein Print ausgeführt */
static portTASK_FUNCTION( vPrintTask, pvParameters ) {
unsigned int i=0;
portTickType xLastWakeTime;

    vTaskSetApplicationTaskTag( NULL, ( void * ) 3 );

    while(1) {
            xLastWakeTime = xTaskGetTickCount();
            xil_printf("Printer Task at iteration   %d \r\n", i++);
            vTaskDelayUntil(&xLastWakeTime, 50);
            }
}

/*Setzt alle 50ms ein neues Muster der LED Lampen*/
static portTASK_FUNCTION( vLedTask, pvParameters ){
portTickType xLastWakeTime;
Xuint8 leds = 0;

    vTaskSetApplicationTaskTag( NULL, ( void * ) 4 );
    while(1) {
            xLastWakeTime = xTaskGetTickCount();
            XGpio_mSetDataReg(XPAR_LED_BASEADDR, 1, leds++);
            vTaskDelayUntil(&xLastWakeTime, 50);
           }
}
```

```
/* Aktualisiert alle 250ms die 7 Segment Anzeige*/
static portTASK_FUNCTION( vSiebenseg, pvParameters ) {
portTickType xLastWakeTime;
int count = 0;

      vTaskSetApplicationTaskTag( NULL, ( void * ) 5 );
      while(1) {
                  xLastWakeTime = xTaskGetTickCount();
                  dispLED(count,0);
                  vTaskDelayUntil(&xLastWakeTime, 250);
                  count++;
                }
}

/*Für die Initialisierung der einzelnen Hardware Komponenten */
static void prvSetupHardware( void )
{
/* Aktivieren des Interrupt Controller für die Timer ISR zur Zeitmessung */
      XIntc_mMasterEnable( XPAR_XPS_INTC_0_BASEADDR );

/*HAW SPI initialisierung*/
      HAW_SPI_mReset(XPAR_HAW_SPI_0_BASEADDR);
      HAW_SPI_Initialize((void *)XPAR_HAW_SPI_0_BASEADDR);

/*GPIO initialisierung für die LED*/
      XGpio_Initialize(&led, XPAR_LED_DEVICE_ID);
      XGpio_SetDataDirection (&led,1, 0x00000000);
      initTrace();
}


int main() {

      microblaze_disable_interrupts();
      xil_printf("Start\r\n");

      prvSetupHardware();

/* Erstellen der Tasks*/
      xTaskCreate( vPrintTask, "print", configMINIMAL_STACK_SIZE, NULL,
                                                   tskIDLE_PRIORITY+1, NULL );
      xTaskCreate( vLedTask, "led0", configMINIMAL_STACK_SIZE, NULL,
                                                   tskIDLE_PRIORITY+2, NULL );
      xTaskCreate( vSiebenseg, "siebSeg", configMINIMAL_STACK_SIZE, NULL,
                                                   tskIDLE_PRIORITY+2, NULL );
      xTaskCreate( vHawWriteFifo, "doSpiWF", configMINIMAL_STACK_SIZE, NULL,
                                                   tskIDLE_PRIORITY+3, NULL );
      xTaskCreate( vReadFifo, "doRF", configMINIMAL_STACK_SIZE, NULL,
                                                   tskIDLE_PRIORITY+3, NULL );

/*  Starten des Scheduler*/
      vTaskStartScheduler();

      return 0;
}

/*Funktion zum aktualisieren der 7 Segment Anzeige*/
```

```
void dispLED( Xuint32 Val, Xuint8 DP )
{
Xuint32 i;
Xuint32 *ledptr, dig;

ledptr = (Xuint32 *)XPAR_LED7SEG_0_BASEADDR;

for ( i=0; i<4; i++ ){
      dig = Digit[Val % 10];

      if ( i == DP )
          dig &= 0x7F;

      *(ledptr + i) = dig;
      Val = Val / 10;
      }
}
```

### B.2.1. C-Code FreeRTOS Demoanwendung Trace Funktion

```
/* FreeRTOS Scheduler includes. */
#include "traceconf.h"
#include "xspi.h"
#include "xparameters.h"

XSpi dacSlave;
volatile Xuint16 taskoutput;


void initTrace(){
        XSpi_Initialize(&dacSlave,XPAR_HAW_SPI_0_DEVICE_ID);
        XSpi_SetOptions(&dacSlave, XSP_MASTER_OPTION | XSP_CLK_ACTIVE_LOW_OPTION);
        XSpi_Start(&dacSlave);
        XSpi_SetSlaveSelect(&dacSlave,0x0000001);
}

void vSetAnalogueOutput( int port,  int pxTaskTag ){

switch(pxTaskTag){

case 0 : taskoutput = 0;
         XSpi_Transfer(&dacSlave,&taskoutput,&taskoutput,2); break;
case 1 : taskoutput = 620;
         XSpi_Transfer(&dacSlave,&taskoutput,&taskoutput,2); break;
case 2 : taskoutput = 1240;
         XSpi_Transfer(&dacSlave,&taskoutput,&taskoutput,2);break;
case 3 : taskoutput = 1860;
         XSpi_Transfer(&dacSlave,&taskoutput,&taskoutput,2); break;
case 4 : taskoutput = 2480;
         XSpi_Transfer(&dacSlave,&taskoutput,&taskoutput,2);break;
case 5 : taskoutput = 3100;
         XSpi_Transfer(&dacSlave,&taskoutput,&taskoutput,2);break;
default : xil_printf("Keine gueltige Trace zahl fuer den Analogen Ausgang");
}
}
```

# C. VHDL CODE DPIM

Listing C.1: VHDL Code des dpimnet Controllers

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity dpimnet is
generic
  (
   -- Neue FIFO-Datenbreite
   C_NEW_FIFO_WIDTH        : integer                    :=   8;
   -- Datenbusbreite
   C_SLV_DWIDTH            : integer                    :=  32;
   -- FIFO-Konstanten
   C_RDFIFO_DEPTH          : integer                    := 512;
   C_WRFIFO_DEPTH          : integer                    := 512;
   C_LOG2_DEPTH            : integer                    :=   9
  );
    Port (
       --Digilent Parallel Signal Interfaces
       mclk      : in std_logic;
       pdb_I     : in  std_logic_vector(0 to 7);
       pdb_O     : out std_logic_vector(0 to 7);
       pdb_T     : out std_logic;
       astb      : in std_logic;
       dstb      : in std_logic;
       pwr       : in std_logic;
       pwait     : out std_logic;
       Bus2IP_Reset             : in   std_logic;
       -- WFIFO-Ports
       WFIFO2IP_Data            : in   std_logic_vector(0 to C_NEW_FIFO_WIDTH-1);
       WFIFO2IP_Occupancy       : in   std_logic_vector(0 to C_LOG2_DEPTH);
       WFIFO2IP_AlmostEmpty     : in   std_logic;
       WFIFO2IP_Empty           : in   std_logic;
       IP2WFIFO_RdReq           : out  std_logic;
       -- RFIFO-Ports
       IP2RFIFO_Data            : out  std_logic_vector(0 to C_NEW_FIFO_WIDTH-1);
       RFIFO2IP_Vacancy         : in   std_logic_vector(0 to C_LOG2_DEPTH);
       RFIFO2IP_AlmostFull      : in   std_logic;
       RFIFO2IP_Full            : in   std_logic;
       IP2RFIFO_WrReq           : out  std_logic;
       -- Status-Signale etc.
       slv_reg0                 : out  std_logic_vector(0 to 31));
end dpimnet;

architecture Behavioral of dpimnet is
```

```
--------------------------------------------------------------------------------
-- Constant Declarations
--------------------------------------------------------------------------------

  -- Hard Coding of the State Bits
  -- 1 Nibble = Bits for the current State
  -- 2 Nibble = Output from the Statemachine
  constant stEppReady : std_logic_vector(0 to 7) := "0000" & "0000";

  constant stEppAwrA  : std_logic_vector(0 to 7) := "0010" & "0100";
  constant stEppAwrB  : std_logic_vector(0 to 7) := "0011" & "0001";

  constant stEppArdA  : std_logic_vector(0 to 7) := "0101" & "0010";
  constant stEppArdB  : std_logic_vector(0 to 7) := "0110" & "0011";

  constant stEppDwrA  : std_logic_vector(0 to 7) := "1000" & "1000";
  constant stEppDwrB  : std_logic_vector(0 to 7) := "1001" & "0001";

  constant stEppDrdA  : std_logic_vector(0 to 7) := "1011" & "0010";
  constant stEppDrdB  : std_logic_vector(0 to 7) := "1100" & "0011";


--------------------------------------------------------------------------------
-- Signal Declarations
--------------------------------------------------------------------------------

   signal      stEppCur,stEppNext    : std_logic_vector(0 to 7) := stEppReady;

   -- Attribute für die zum Ausschalten der FSM Kodierung aufgrund
   -- der Hard Kodierten Zustandbits
  attribute fsm_extract : string;
  attribute fsm_extract of stEppCur: signal is "no";
  attribute fsm_extract of stEppNext: signal is "no";

  attribute fsm_encoding : string;
  attribute fsm_encoding of stEppCur: signal is "user";
  attribute fsm_encoding of stEppNext: signal is "user";

  attribute signal_encoding : string;
  attribute signal_encoding of stEppCur: signal is "user";
  attribute signal_encoding of stEppNext: signal is "user";

  signal      clkMain         : std_logic;
  -- Internal control signales
  signal      ctlEppWait      : std_logic;
  signal      ctlEppAstb      : std_logic;
  signal      ctlEppDstb      : std_logic;
  signal      ctlEppDir       : std_logic;
  signal      ctlEppWr        : std_logic;
  signal      ctlEppAwr       : std_logic;
  signal      ctlEppDwr       : std_logic;
  signal      busEppOut       : std_logic_vector(0 to 7);
  signal      busEppIn        : std_logic_vector(0 to 7);
  signal      busEppData      : std_logic_vector(0 to 7);

  -- Adressregister
  signal      regEppAdr       : std_logic_vector(0 to 7);
```

```
  -- Statussignals of the FIFOs
  signal  StateSigs : std_logic_vector(0 to 3);

begin

  slv_reg0 <= x"0000000" & StateSigs; -- Status Register

  ctlEppDwr  <= stEppCur(4);
  ctlEppAwr  <= stEppCur(5);
  ctlEppDir  <= stEppCur(6);
  ctlEppWait <= stEppCur(7);

  clkMain <= mclk;
  ctlEppAstb <= astb;
  ctlEppDstb <= dstb;
  ctlEppWr   <= pwr;
  pwait      <= ctlEppWait;
  pdb_O      <= busEppOut;
  busEppIn   <= pdb_I;
  pdb_T      <= ctlEppWr nand ctlEppDir;

  -- Selekt between Adress or Daten
  busEppOut <= regEppAdr when ctlEppAstb = '0' else busEppData;
  StateSigs <= RFIFO2IP_AlmostFull & RFIFO2IP_Full & WFIFO2IP_AlmostEmpty & WFIFO2IP_Empty;
  IP2RFIFO_Data <= busEppIn;

  -- Decode the address register and select the appropriate data register
  busEppData <= WFIFO2IP_Data                          when regEppAdr = x"00" else
                "000000" & WFIFO2IP_Occupancy(0 to 1) when regEppAdr = x"01" else
                WFIFO2IP_Occupancy(2 to 9)            when regEppAdr = x"02" else
                "000000" & RFIFO2IP_Vacancy(0 to 1)   when regEppAdr = x"04" else
                RFIFO2IP_Vacancy(2 to 9)              when regEppAdr = x"05" else
                                              "00000000";

------------------------------------------------------------------------
-- EPP Interface Control State Machine
------------------------------------------------------------------------

-- This process moves the state machine to the next state
-- on each clock cycle
process (clkMain)
    begin
        if rising_edge(clkMain) then
            if Bus2IP_Reset = '1' then
                stEppCur <= stEppReady;
            else
                stEppCur <= stEppNext;
                if ctlEppAwr = '1' then
                    regEppAdr <= busEppIn;
                end if;
            end if;
        end if;
end process;

-- This process determines the next state machine state based
-- on the current state and the state machine inputs.
process (stEppCur, stEppNext, ctlEppAstb, ctlEppDstb, ctlEppWr)
```

```
begin
    IP2WFIFO_RdReq <= '0';
    IP2RFIFO_WrReq <= '0';

    case stEppCur is

        -- Ready-Zustand: Auf den Beginn eines Epp-Zyklus warten
        when stEppReady =>
            stEPPNext <= stEppReady;
            if ctlEppWr = '0' then
                if ctlEppAstb = '0' then
                    -- Address write
                    stEppNext <= stEppAwrA;
                elsif ctlEppDstb = '0' then
                    -- Data write
                    stEppNext <= stEppDwrA;
                end if;
            else
                if ctlEppAstb = '0' then
                    -- Address read
                    stEppNext <= stEppArdA;
                elsif ctlEppDstb = '0' then
                    -- Data read
                    stEppNext <= stEppDrdA;
                end if;
            end if;

        when stEppAwrA =>
            stEppNext <= stEppAwrB;

        when stEppAwrB =>
            if ctlEppAstb = '0' then
                stEppNext <= stEppAwrB;
            else
                stEppNext <= stEppReady;
            end if;

        when stEppArdA =>
            stEppNext <= stEppArdB;

        when stEppArdB =>
            if ctlEppAstb = '0' then
                stEppNext <= stEppArdB;
            else
                stEppNext <= stEppReady;
            end if;
        when stEppDwrA =>
            --user_logic writes to RFIFO
            if RFIFO2IP_Full = '0' and regEppAdr = x"03"then
                IP2RFIFO_WrReq <= '1';
            end if;
            stEppNext <= stEppDwrB;
        when stEppDwrB =>
            if ctlEppDstb = '0' then
                stEppNext <= stEppDwrB;
            else
                stEppNext <= stEppReady;
```

```
                end if;

            when stEppDrdA =>
                if WFIFO2IP_Empty ='0' and regEppAdr = x"00"  then
                   IP2WFIFO_RdReq <= '1';
                end if;
                stEppNext <= stEppDrdB;
            when stEppDrdB =>
                if ctlEppDstb = '0' then
                   stEppNext <= stEppDrdB;
                else
                   stEppNext <= stEppReady;
                end if;
            when others =>
                -- unknown State
                stEppNext <= stEppReady;
        end case;
    end process;
end Behavioral;
```

# 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, 6. Juli 2009 | |
| --- | --- |
| Ort, Datum | Unterschrift |