HAW HAMBURG

Bachelorarbeit

Jana Miericke

Entwicklung und Validierung der Datenkommunikation zwischen FPGA und Processing System mit Linux-Betriebssystem auf einem System on Chip für die Regelung einer Magnetstromversorgung

Faculty of Computer Science and Engineering
Department of Information and Electrical Engineering

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Jana Miericke

Entwicklung und Validierung der Datenkommunikation zwischen FPGA und Processing System mit Linux-Betriebssystem auf einem System on Chip für die Regelung einer Magnetstromversorgung

Bachelor Thesis based on the examination and study regulations
for the Bachelor of Engineering degree programme
Bachelor of Science Elektro- und Informationstechnik
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg

Erstprüfer: Prof. Dr. Lutz Leutelt
Zweitprüfer: Prof. Dr. Ulrich Sauvagerd
Betreut durch Dr. Christian Putscher

Tag der Abgabe: 19. Mai 2023

Inhaltsverzeichnis

1. Zusammenfassung 1
1.1. Zusammenfassung . . . 1
1.2. Abstract . . . 1

2. Einleitung 3

3. Grundlagen 5
3.1. Mercury XU5 . . . 5
3.2. Xilinx Developer Tools . . . 6
3.3. Embedded Linux auf dem Zynq UltraScale+ . . . 7
3.4. Das AXI-Protokoll . . . 8
3.5. Unterarten des AXI-Protokolls . . . 11
3.6. Der AXI-GPIO-Core . . . 12

4. Anforderungsanalyse 14
4.1. Übergeordnetes System . . . 14
4.2. Stakeholderanalyse . . . 15
4.3. Zusammenfassung des Anforderungen . . . 18

5. Konzept 20
5.1. Die Xilinx Entwicklungsumgebung . . . 20
5.2. Das AXI-Interface . . . 21
5.3. Das AXI-Subordinate . . . 21
5.4. Der AXI-Manager . . . 23
5.5. Das Linux-Betriebssystem . . . 23
5.6. Validierung der Datenkommunikation . . . 24

6. Implementierung 26
6.1. Erstellung der Entwicklungsumgebung . . . 26
6.2. Erstellung eines Logik-Moduls . . . 27
6.3. Implementierung der AXI-Kommunikation mit dem AXI-GPIO . . . 27
6.4. Generierung des Linux-Boot-Image . . . 30
6.5. Programmierung des Processing Systems ohne Linux-Betriebssystem . . . 31
6.6. Programmierung des Processing-Systems mit Linux-Betriebssystem . . . 35
6.7. Messung der Funktionalität und Performance des AXI-Interface . . . 37

6.8. Messung des zeitlichen Unterschieds beim Hardwarezugriff mit und ohne Linux-Betriebssystem 38

7. Auswertung 40
7.1. Funktionalität des AXI-Interface 40
7.2. Performance der AXI-Kommunikation 45
7.3. Der Einsatz des Linux-Betriebssystems 51

8. Fazit 53

9. Ausblick 55

10. Literaturverzeichnis 57

A. Appendix 62

B. Eidesstattliche Erklärung 66

1. Zusammenfassung

1.1. Zusammenfassung

Für die Implementierung einer Magnetstromregelung wurde die Entwicklungsplattform Mercury XU5 von Enclustra in Betrieb genommen. Auf dem System on Chip wurde zwischen dem FPGA und dem Processing-System mit Linux-Betriebssystem eine Datenkommunikation mit dem AXI-Interface aufgebaut. Die wichtigste Anforderung war die Evaluation der zeitlichen Performance dieser Übertragung, da diese der Echtzeitanforderung der Regelung unterliegt. Die Datenkommunikation wurde mit dem AXI-GPIO-IP-Core von Xilinx aufgebaut. Es wurde eine Software geschrieben, die den AXI-GPIO-Core aus dem Linux-Betriebssystem heraus auslesen und beschreiben kann. Die Performance der Datenübertragung wurde mit dem Integrated-Logic-Analyzer von Xilinx gemessen. Dabei stellte sich heraus, dass diese auf dem AXI4-Lite-Interface basierende Kommunikation für die Anwendung innerhalb der Magnetstromregelung zu langsam ist. Es wurde außerdem ein weiterer Ansatz implementiert, der unter der Nutzung des vollen AXI4-Interfaces die Daten in Bursts überträgt. Dabei konnte eine deutliche Verbesserung der Übertragungsgeschwindigkeit festgestellt werden. Allgemein stellt die Arbeit eine funktionierende Datenkommunikation und einen Ansatz für die Verbesserung deren Übertragungsgeschwindigkeit mit Burst-Transaktionen bereit.

1.2. Abstract

For the implementation of a magnet-current-control, the developement platform Mercury XU5 by Enclustra was set up. A data communication was established on the System on Chip between the FPGA and the Processing-System running a Linux operating system. The main requirement was the evaluation of the timing performance of the data transmission, as it has to meet the real-time requirement of the control loop. The data communication was built by using the AXI GPIO IP core from Xilinx. A Software was written, that reads from and writes to the AXI GPIO core within the Linux operating system. The performance of the data communication was measured by using the Integrated Logic Analyzer from Xilinx. The result was, that the performance of the communication, based on the AXI4-Lite interface is too slow for an application in the magnet-current-control. A second approach was implemented, that uses the full AXI4-Interface and transmits data

in bursts. The approach led to a significant improvement of transfer speed. In general the thesis provides a working data communication and an approach for the improvement of its transfer speed by using burst transactions.

2. Einleitung

Das Deutsche Elektronen Synchrotron DESY ist die größte Beschleuniger-Anlage Deutschlands. Der dort gebaute Beschleuniger PETRA III ist seit 2014 eine der brillantesten Röntgenlichtquellen der Welt. Seit einigen Jahren werden nun jedoch die Beschleuniger auf der ganzen Welt weiter ausgebaut. Das Ziel des Ausbaus sind Röntgenlichtquellen 4. Generation. Diese neue Technologie soll den Röntgenstrahl noch weiter fokussieren, sodass auch Strukturen im Nanobereich aufgelöst werden können. Auch DESY ist in den Umbau zu Röntgenlichtquellen 4. Generation eingestiegen und möchte mit dem neuen Beschleuniger PETRA IV ein 3D-Röntgenmikroskop der Superlative bauen. Das PETRA IV-Projekt hat das Ziel den PETRA III-Beschleuniger in eine „ultra-low-emittance"-Röntgenlichtquelle umzubauen. Die Emittanz ist charakteristisch für einen Beschleuniger, da sie von den Störgrößen, die auf die Elektronen im Speicherring einwirken, abhängig ist. Das Ziel ist es den Röntgenstrahl annähernd auf das Beugungslimit fokussieren zu können. Das Beugungslimit beschreibt den kleinsten Abstand, den zwei Objekte zueinander haben dürfen, um noch als zwei separate Objekte von einem Röntgenstrahl mit diskreter Wellenlänge aufgelöst werden zu können. Für Energien von bis zu 10 keV soll PETRA IV eine Auflösung nahe dem Beugungslimit erreichen und damit einen extrem kohärenten Röntgenstrahl erzeugen können. Um bei Energien von bis zu 10 keV das Beugungslimit zu erreichen wird eine Emittanz von kleiner gleich 10 pm rad benötigt. Die vertikale Emittanz von PETRA III beträgt 10 pm rad. Die horizontale Emittanz liegt mit 1300 pm rad jedoch deutlich über dem für die Erreichung des Beugungslimits benötigtem Wert. Das Ziel von PETRA IV ist es eine horizontale Emittanz von 10 bis 20 pm rad zu erhalten.

Um dies zu erreichen wird der gesamte Beschleuniger PETRA III umgebaut. Es werden neue Experimentierhallen, neue Beamlines und ein neues Magnetsystem entwickelt und gebaut. In PETRA IV sollen 2562 Magneten und 700 Korrektur-Magneten eingebaut werden. Eine solche Menge an Magneten erfordert außerdem eine hohe Anzahl an Netzteilen für die Magnetstromversorgung. Die Regelkomponenten für die Überprüfung dieser werden vom DESY selbst entwickelt, da diese zu den sogenannten „accuracy-defining components" gehören. Die Kontrolle der Netzteile und die Beobachtung sämtlicher Regelparameter soll über Ethernet vorgenommen werden.[1]

Für die Umsetzung dieser unterschiedlichen Funktionalitäten wie die Implementierung

von Regelungen, die Datenkommunikation über Server und das Nutzen von Ethernet-Schnittstellen innerhalb eines Anwendungsgebiets werden häufig System-on-Chips (SoC) verwendet. Bei einem SoC handelt es sich um die Kombination von Software und Hardware. Die Software ist im CPU-Subsystem untergebracht. Das CPU-Subsystem enthält einen oder mehrere CPU-Kerne, einen On-Chip-Speicher und externe Speicher. Es enthält außerdem Kommunikationsprotokolle wie USB und Ethernet. Die Hardwarekomponente enthält den FPGA (Field Programmable Gate Array). Dabei handelt es sich um Halbleiter-Gerät, dessen Funktion durch programmierbare Logikbausteine konfigurierbar ist.[2–4] Hersteller wir AMD Xilinx[5] oder Intel[6] bieten jeweils eine Reihe von SoCs für unterschiedlichste Anwendungsgebiete, wie Industrie, Medizin, Automobil und Luftfahrt an.[7]

Diese Arbeit ist Teil eines großen Projekts zur Neuentwicklung der Magnetstromversorgung für den Beschleuniger Petra IV. Mit dem Ziel die Auflösung des Röntgenstrahls an physikalische Grenzen zu treiben, gilt es in jeder Entwicklungsstufe innovative Technologien zu finden und zu entwickeln. In dieser Arbeit soll die Plattform für die Implementierung der digitalen Magnetstromregelung in Betrieb genommen und vorbereitet werden. Die Aufgabe dieser Arbeit ist die Entwicklung einer Datenkommunikation zur Übertragung der Regelparameter aus dem FPGA des SoCs an den Prozessor. Diese Datenkommunikation soll dabei den speziellen Anforderungen dieses Projekts entsprechen und sich flexibel in die Entwicklung der Magnetstromregelung eingliedern.

3. Grundlagen

3.1. Mercury XU5

Mercury XU5 ist ein Processing-System der Firma Enclustra. Das Board bietet neben einem System-on-Chip (SoC) weitere Hardware-Komponenten wie DDR4-SDRAM (Double Data Rate 4 - Synchronous Dynamic Random Access Memory), eMMC (embedded Multi Media Card) und QSPI (Quad Serial Peripheral Interface) Flash, Gigabit-Ethernet-PHY (PHYsical Layer) und USB-3.0-PHY.[8]
Um das Mercury XU5 als vollständige Entwicklungsumgebung nutzen zu können, kann es mit einem Baseboard von Enclustra, beispielsweise dem Mercury+ ST1 kombiniert werden.[9]

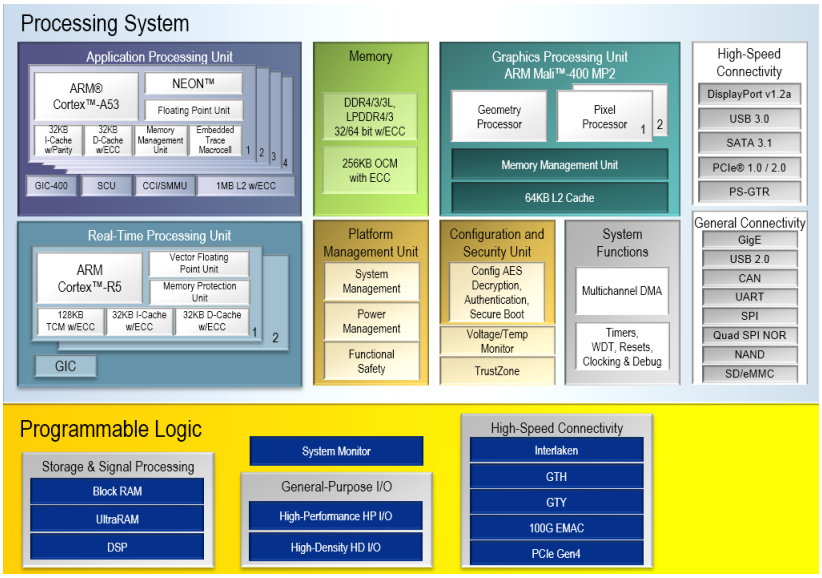

Abb. 3.1. Blockdiagramm des Zynq UltraScale+ mit Processing-System und Programmable-Logic[10]

Die Hauptkomponente ist der Zynq Ultrascale+ MPSoC (Multiprozessor-System-on-Chip) von Xilinx. Dieser besteht aus einem Processing-System (PS) und einer Programmable-Logic (PL) (siehe Abb. 3.1).
Das Processing-System ist mit einer Application-Processing-Unit (APU) basierend auf einem quad-core ARM (Advanced RISC Machines) Cortex A53 und einer Real-Time-Processing-Unit (RPU) basierend auf einem dual-core ARM Cortex RF5 ausgestattet.
Die Hauptkomponente der PL-Einheit ist der FPGA. Die PL-Einheit besitzt jedoch auch

eigene Peripheriemodule wie einen 100-GB-Ethernet-PHY.[11–13]

3.1.1. PS-PL-AXI-Interface

Der Zynq UltraScale+ bietet mehrere General-Purpose-Interconnect-Blöcke für den Daten-Transfer zwischen Processing-System und Programmable-Logic. Für die Verbindung von PL und PS gibt es unter anderem das High-Performance PS-PL-AXI (Advanced eXtensible Interface)-Interface M_AXI_HPM_FPD, welches die Programmable-Logic mit der Full-Power-Domain des Processing-System verbindet. Über dieses kann das Processing-System große Datenmengen zur Programmable-Logic transferieren. Der Interconnect-Block vergibt für diese AXI-Transaktionen (siehe Abschnitt 3.4.3) jeweils IDs, welche die Kohärenz der Daten während des Transfers garantieren.
Für das Debuggen von Applikationen, die eine PS-PL-Kommunikation beinhalten, gibt es einen Debug-Cross-Trigger.[13]

3.2. Xilinx Developer Tools

Zur Entwicklung von Designs und der Programmierung des System on Chips von Xilinx gibt es die Xilinx Developer Tools. Die zwei größten Software-Pakete daraus sind die Vivado Design Suite und die Vitis Unified Software Platform.[14]

3.2.1. Die Vivado Design Suite

Die Vivado Design Suite ist ein Kombination aus verschiedenen Features zur Erstellung, Implementierung und Validierung von FPGA-Designs. Die Vivado Integrated Design Environment (IDE) bietet eine graphische Nutzeroberfläche für die Entwicklung von Designs.[15] Dort können Quell-Dateien in verschiedenen Hardware-Description-Languages geschrieben werden oder für die gewünschte Funktionalität ein IP-Core eingebunden werden. IP steht für Intellectual Property und beschreibt die Zusammenfassung von fertigen Subsystemen in direkt anwendbaren Cores.[3,16] Diese können mit dem IP-Integrator-Tool[17] aus dem Vivado-IP-Katalog ausgewählt und in das FPGA-Design eingebunden werden. Das FPFA-Design wird in der IDE mit Hardware Description Languages auf Register-Transfer-Ebene beschrieben. Vivado beinhaltet Synthese-Tools, die fertige Designs auf Register-Transfer-Ebene in eine Repräsentation auf Gate-Ebene, also eine Beschreibung durch weniger abstrakte, logische Gatter, umwandeln.[3,18] Durch die Synthese entstehen Netzlisten, die als Eingangsprodukt für die Implementierung genutzt werden. Die Implementierung besteht aus der Optimierung der Logik, Platzierung von logischen Zellen und dem Routen von Verbindungen zwischen diesen Zellen.[19] Nach der Implementierung kann der Bitstream erzeugt werden. Mit dieser .bit-Datei wird der FPGA mit dem erstellten Design programmiert.[20] Das Design kann in Form einer Hardware-

Specification-Datei, oder auch .xsa-Datei (Xilinx Shell Archive) exportiert werden, um in Vitis auf Basis des FPGA-Designs Software zu erstellen.[21]

3.2.2. Die Vitis Unified Software Platform

Die Vitis Unified Software Platform fasst alle Software-Entwicklungs-Tools von Xilinx zusammen. Mit der Vitis Integrated Design Environment (IDE) können Software-Applikationen für Xilinx-Embedded-Prozessoren entwickelt werden. Die Entwicklung von Sotware für SoCs geschieht über das Platform-Projekt und das System-Projekt (siehe Abb. 3.2).

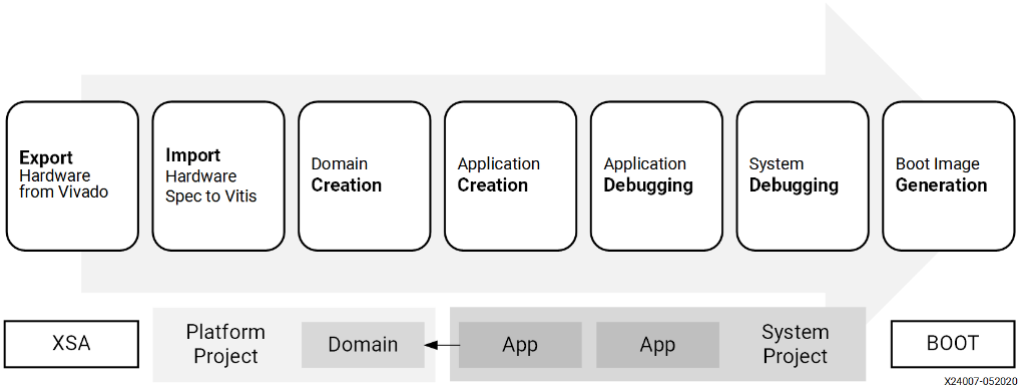

Abb. 3.2. Vitis Workflow mit Platform-Projekt und System-Projekt[21]

Das Platform-Projekt beinhaltet die Hardware-Informationen und wird auf Basis der aus Vivado exportieren .xsa-Datei erstellt. Innerhalb des Platform-Projekts kann nun ein System-Projekt erstellt werden, in welchem wiederum Application-Projekte erstellt werden können. Ein Apllication-projekt enthält die Quell- und Header-Dateien für das Projekt. Durch die Kompilierung des Application-Projekts wird eine binäre Output-Datei erstellt, die .ELF-Datei. Diese kann auf dem Prozessor ausgeführt werden.[21]

3.3. Embedded Linux auf dem Zynq UltraScale+

Der Zynq UltraScale+ bootet im Multi-Stage-Boot-Modus. Während des Bootvorgangs des Systems erfüllen mehrere Komponenten unterschiedliche Aufgaben und bereiten jeweils die Umgebung für die folgende Komponente vor. Der Zynq UltraScale+ bietet verschiedene Boot-Devices von denen gebootet werden kann: JTAG (Joint Test Action Group), SD-Speicherkarte (Secure Digital Memory Card), QSPI-Flash, eMMC-Flash.
Zu Beginn des Boot-Vorgangs (siehe Abb. 3.3) wird anhand der Device-Mode-Pins der eingestellte Boot-Modus ausgelesen. Die Configuration-Security-Unit (CSU) kopiert den First-Stage-Boot-Loader (FSBL) vom Boot-Device in den On-Chip-Random-Access-Memory (OCM). Die APU führt den FSBL aus und initialisiert somit den RAM-Controller. Der FSBL lädt den Second-Stage-Boot-Loader (SSBL) in den RAM. Bei einem Embedded Linux auf dem Zynq UltraScale+ handelt es sich bei dem SSBL um den Boot-Loader

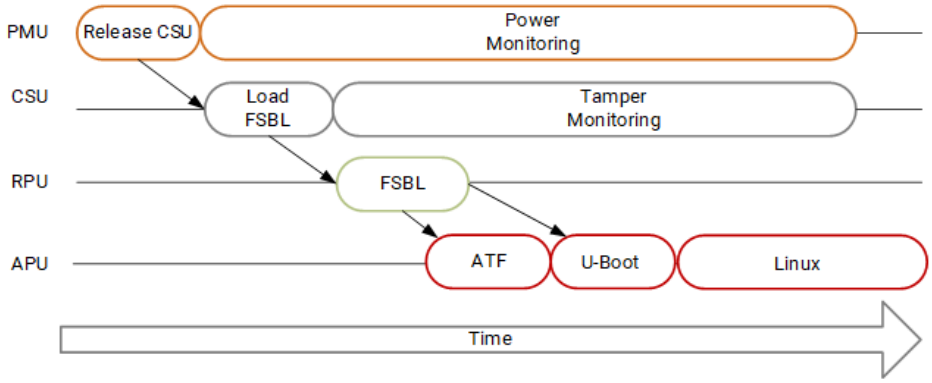

Abb. 3.3. Bootvorgang auf dem Zynq UltraScale+[22]

U-Boot.[23,24] Der FSBL konfiguriert außerdem die PL mit dem Bitstream. Der U-Boot Boot-Loader lädt den Linux-Kernel, den Devicetree und andere Dateien in den RAM und führt den Linux-Kernel aus. Der Linux-Kernel konfiguriert dann weiterführend die Peripheriemodule und das User-Space-Root-File-System und führt die Init-Applikation aus. Die Init-Applikation führt wiederum User-Space-Applikationen aus.[11,22,25,26]

3.3.1. Der Devicetree

Der Devicetree ist eine Struktur, die für das Betriebssystem nicht-detektierbare Hardware beschreibt. Die Hardware-Systeme werden in der Devicetree-Spezifikation auch Devices genannt. Im Devicetree gibt es jeweils Einträge für diese Devices, die in einer baumartigen Struktur angeordnet sind. So ein Eintrag wird auch Node genannt. Jede Node legt sogenannte Properties für das Device fest, welche bestimmte Eigenschaften des Gerätes beschreiben.[27,28]

3.4. Das AXI-Protokoll

Das AXI (Advanced eXtensible Interface)-Protokoll ist ein Interface-Protokoll aus dem AMBA (Advanced Microcontroller Bus Architecture)-Standard von ARM. Dieser spezifiziert Protokolle für die Kommunikation auf SoCs.[29]
Das AXI-Protokoll legt zwei Arten von Interfaces fest: das AXI-Manager-Interface und das AXI-Subordinate-Interface. Eine AXI-Verbindung besteht immer zwischen diesen beiden Interfaces. Damit zwei Komponenten über das AXI-Protokoll kommunizieren können, muss eine Komponente ein AXI-Manager-Interface und die andere Komponente ein AXI-Subordinate-Interface aufweisen.[30]
Im weiteren Verlauf der Arbeit werden mit den Begriffen AXI-Manager und AXI-Subordinate auch die Komponenten, die diese Interfaces beinhalten gemeint.

3.4.1. Die Kanäle des AXI-Interfaces

Das AXI-Interface definiert fünf Kanäle: Drei Kanäle für einen Schreibzugriff und zwei Kanäle für einen Lesezugriff (siehe Abb. 3.4). Für beide Zugriffe gibt es jeweils einen Adress-Kanal und einen Datenkanal. Der AXI-Manager sendet vor dem Senden der Daten die zu beschreibende oder auszulesende Adresse und weitere Kontrollsignale über den Adresskanal (AW bzw. AR) zum AXI-Subordinate. Diese Aktion wird als Request bezeichnet.

Bei einem Lesezugriff sendet das AXI-Subordinate die Daten über den separaten Datenkanal (R) von der angefragten Adresse zum AXI-Manager. Über denselben Kanal sendet das Subordinate eine Leseantwort, die den Abschluss des Lesevorgangs signalisiert.
Bei einem Schreibzugriff werden die Daten über den Datenkanal (W) zum AXI-Subordinate geschrieben. Das Subordinate schreibt die erhaltenen Daten dann an die angefragte Adresse.
Da die Kanäle unidirektional sind und über den Datenkanal (W) nur Daten vom Manager zum Subordinate geschrieben werden können, benötigt der Schreibvorgang einen weiteren Kanal (B). Auf diesem teilt das Subordinate mit der Schreibantwort dem Manager mit, dass der Vorgang korrekt beendet wurde.[30,31]

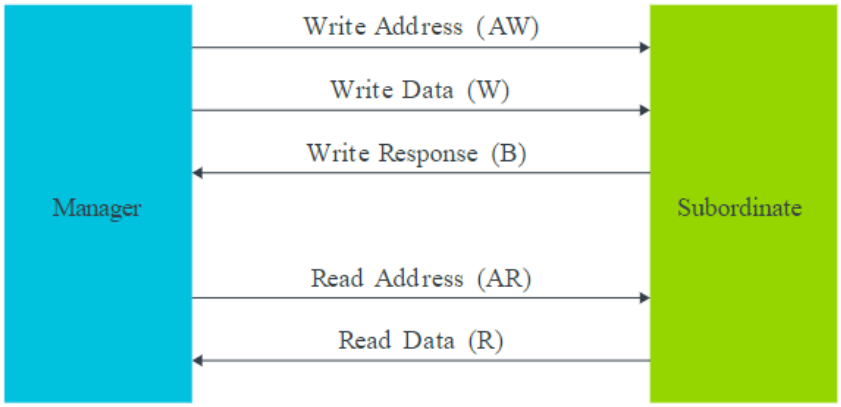

Abb. 3.4. Kanäle des AXI-Interface zwischen AXI-Manager und AXI-Subordinate[30]

3.4.2. Der AXI-Handshake

Jede Kommunikation auf einem AXI-Kanal beruht auf dem AXI-Handshake-Prinzip. Dieser Handshake wird zwischen dem AXI-Manager und dem AXI-Subordinate mittels der Signale READY und VALID ausgeführt. Das VALID-Signal wird von der Quelle auf HIGH gesetzt, wenn die zu schreibenden Daten korrekt und zum Schreib- bzw. Lesevorgang bereit sind. Die Quelle setzt das READY-Signal auf HIGH, wenn sie bereit ist neue Daten zu empfangen. Sowohl AXI-Manager, als auch AXI-Subordinate können Quelle und Ziel sein, je nachdem, ob es sich um einen Schreib- oder Lesevorgang handelt. Beim

Schreibvorgang ist der Manager die Quelle und beim Lesevorgang ist es das Subordinate. Wenn beide Signale HIGH sind kommt es zum sogenannten Transfer. Ein Transfer ist im AMBA-AXI-Protokoll als einzelner Informationsaustausch auf einem Kanal über einen einzigen READY-VALID-Handshake definiert.[31,32]

3.4.3. Die AXI-Transaktion

Die gesamte Kommunikation zwischen Manager und Subordinate, die einen Adresskanal und einen Datenkanal benötigt, über den ein oder mehrere Datentransfers geschehen, wird Transaktion genannt.

Der Ablauf von mehreren Datentransfers in einer Transaktion wird als Burst bezeichnet. Im Burst-Modus ist es möglich, dass nach einem einzigen Adresstransfer mehrere Datentransfers folgen. Dazu sendet der AXI-Manager nur die Adresse für den ersten Datentransfer und das AXI-Subordinate berechnet die Adressen für die darauffolgenden Datentransfers.
Für die Burst-Transaktion gibt es drei Kontrollsignale, die während des Requests an das AXI-Subordinate gesendet werden. Das Signal AWBURST/ARBURST gibt an, wie die Adressen der nachfolgenden Datentransfers berechnet werden. Das Signal AWLEN/ARLEN gibt die Anzahl der Datentransfers an. Die maximale Anzahl an Bytes, die pro Datentransfer übertragen wird, wird von dem Signal AWSIZE/ARSIZE angegeben.[31,32]

3.4.4. Das AXI-Interconnect

Um zwei Komponenten über das AXI-Protokoll miteinander zu verbinden, wird ein AXI-Interconnect benötigt. Dabei handelt es sich um eine weitere Komponente, die mindestens ein Manager- und mindestens ein Subordinate-Interface besitzt. Das Interconnect wird jeweils über das AXI-Manager- und AXI-Subordinate-Interface mit den anderen beiden Komponenten verbunden[30] (siehe Abb. 3.5).
Das Interconnect zwischen der AXI-Manager-Komponente und der AXI-Subordinate-Komponente übernimmt in der Datenkommunikation verschiedene Aufgaben. Das AXI-Protokoll unterstützt die Kommunikation zwischen zwei Komponenten, die einen unterschiedlich großen Adressraum haben. Bei der Datenübertragung zwischen solchen Komponenten muss das Interconnect die fehlerfreie Übertragung gewährleisten.
Ein Interconnect kann mit mehreren Managern und Subordinates verbunden sein. Es muss den korrekten Ablauf der Transaktionen regeln, damit die Transaktion jeweils mit den zusammengehörigen Komponenten geschieht.
Interconnects machen es außerdem möglich, dass Manager mehrere Anfragen hintereinander losschicken können. Dafür enthalten die Kanäle des Interfaces das sogenannte Transaction-Identifier-Signal. Dadurch kann der Manager eine weitere Transaktion beginnen, bevor die vorherige abgeschlossen ist. Das Interconnect ordnet die angeforderten

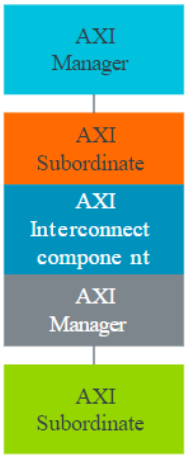

Abb. 3.5. Verbindung einer AXI-Manager-Komponente und einer AXI-Subordinate-Komponente mit einem AXI-Interconnect[30]

Transaktionen anhand der ID und garantiert so die korrekte Beendigung jeder Transaktion.[31]

3.5. Unterarten des AXI-Protokolls

Der AMBA-Standard spezifiziert unter anderem drei Arten von AXI-Interfaces: AXI4, AXI4-Lite und AXI4-Stream. AXI4-Lite ist eine Unterart von AXI4 und implementiert nicht alle Signale des AXI-Protokolls. Der wichtigste Unterschied zu AXI4 ist, dass das AXI4-Lite-Interface keine Bursts unterstützt. Die Datenübertragung mit diesem Interface ist auf einen Datentransfer pro Transaktion beschränkt. Das AXI4-Interface dagegen kann bis zu 256 Datentransfers pro Transaktion ausführen.[33]

3.5.1. Memory-Mapped-Protokolle

Bei AXI4 und AXI4-Lite handelt es sich um Memory-Mapped-Protokolle. Diese Protokolle zeichnen sich dadurch aus, dass sie vor einem Datentransfer eine Adresse an das Subordinate senden. Die Daten werden in einem bestimmten Bereich des Speichers, der sogenannten Memory-Map gespeichert[33] (siehe Abschnitt 3.6.1). Bei einem Burst-Transfer wird nicht der Memory-Map-Speicher genutzt, sondern der interne Speicher.[34]

3.5.2. Das AXI-Stream-Interface

Das AXI4-Stream-Interface ist kein Memory-Mapping-Protokoll. Es verwendet nur einen unidirektionalen Kanal, auf dem Daten vom AXI-Manager zum AXI-Subordinate gesendet werden können. Dabei kann das AXI4-Stream-Interface Bursts von uneingeschränkter Länge streamen. Da es das Senden der Adresse und weiterer Kontrollsignale einspart, kann es hohe Übertragungsgeschwindigkeiten erreichen.[33]

3.6. Der AXI-GPIO-Core

AMD Xilinx hat das AXI-Interface adaptiert und in mehreren IP-Cores umgesetzt.[33] Der AXI-GPIO ist ein IP-Core von Xilinx. Er bildet ein Interface zwischen General Purpose In– und Outputs und dem AXI4-Lite-Interface. Der IP-Core besteht aus drei Komponenten, einem AXI-Interface-Modul, einem GPIO-Core und einem Interrupt-Modul (siehe Abb. 3.6).

Das AXI-Interface-Modul verbindet den GPIO-Core mit dem AXI-Subordinate-Interface des AXI-GPIO-Cores über das AXI4-Lite-IP-Interface (IPIF).[35]
Der GPIO-Core kann Interrupt-Signale an den Prozessorkern senden. Wenn ein Interrupt an einem der Eingangs-Ports des GPIO-Cores auftritt, wird dieses Signal zum Interrupt-Modul gesendet. Das Interrupt-Modul besitzt die entsprechenden Register, um Interrupts global und lokal zu aktivieren und den Status der Interrupts zu verwalten.[36]

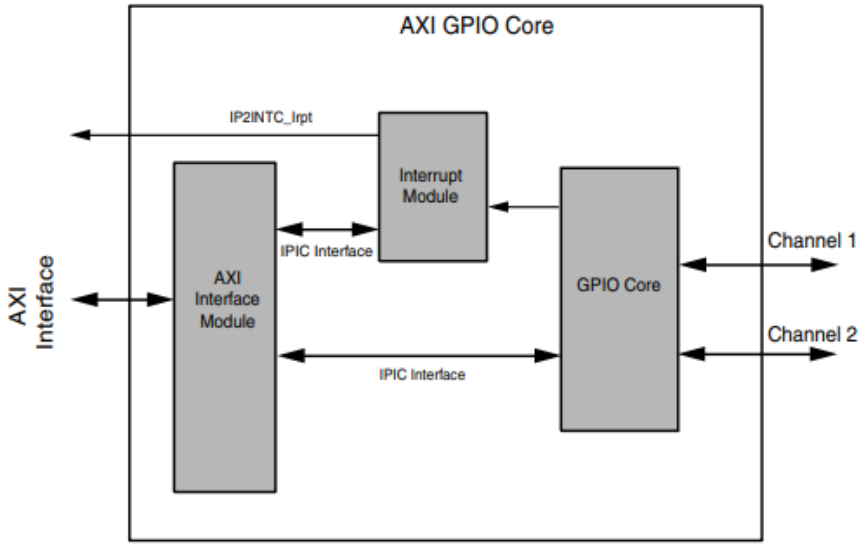

Abb. 3.6. Blockdiagramm des AXI-GPIO-IP-Cores[36]

3.6.1. Der GPIO-Core

Der GPIO-Core stellt zwei Kanäle mit einer Breite von 32 Bit zur Verfügung. Diese sind jeweils einzeln konfigurierbar und können als Ein- der Ausgänge genutzt werden. Um beide Kanäle nutzen zu können, muss der Dual-Channel-Modus aktiviert werden. Wie in Abbildung 3.7 zu erkennen besitzen beide Kanäle jeweils zwei Register: das GPIO_DATA-Register und das GPIO_TRI-Register (3-State-Register).
Bei einem Schreibzugriff auf den GPIO-Core werden die Daten vom Eingang des Cores im GPIO_DATA-Register gespeichert. Bei einem Lesezugriff werden die Daten aus dem Register am Ausgang des GPIO-Cores zur Verfügung gestellt.
Mithilfe des GPIO_TRI-Registers können alle 32 möglichen Ports individuell als Ein-

Base Address + Offset (hex)	Register Name	Access Type	Default Value (hex)	Description
C_BASEADDR + 0x00	GPIO_DATA	Read/Write	0x0	Channel 1 AXI GPIO Data Register
C_BASEADDR + 0x04	GPIO_TRI	Read/Write	0x0	Channel 1 AXI GPIO 3-state Register
C_BASEADDR + 0x08	GPIO2_DATA	Read/Write	0x0	Channel 2 AXI GPIO Data Register
C_BASEADDR + 0x0C	GPIO2_TRI	Read/Write	0x0	Channel 2 AXI GPIO 3-state Register

Abb. 3.7. Register des GPIO-Cores[36]

oder Ausgang konfiguriert werden.

Das AXI4-Lite-Interface des AXI-GPIO-Cores ist ein Memory-Mapping-Interface. Die Register befinden sich in einem bestimmten Bereich im Speicher, der sogenannten Memory Map. Die Adresse dieses Bereichs wird durch das Attribut C_BASEADDR festgelegt. Die Register befinden sich an festen Offsets dieser Basisadresse. Die Register von solchen Peripherie-Modulen wie dem GPIO-Core werden als Memory-Mapped-I/O (MMIO) bezeichnet. Diese Art von Speicher wird auch oft als Device Memory bezeichnet, dabei handelt es sich um Speicher für Peripheriemodule.[33,36]

4. Anforderungsanalyse

4.1. Übergeordnetes System

Die Magneten des Beschleunigers werden von Leistungsnetzteilen mit Strom versorgt. Da die induzierten Magnetfelder hochgenaue Anforderungen erfüllen müssen, muss dieser Strom präzise[1] geregelt werden (siehe Abb. 4.1). Dazu wird der Strom am Magneten mit einem hochpräzisen Stromsensor, einem DCCT-Sensor[37] (Direct-Current Current Transformer) gemessen. Nach der Strommessung wird das Signal mittels eines ADCs digitalisiert und in die digitale Magnetstromregelung eingespeist. Dort wird zunächst die Abweichung des erfassten Ist-Stroms vom Soll-Strom berechnet. Diese Abweichung wird einem PI-Regler zugeführt, welcher eine Stellgröße ausgibt und diese an einen Pulsgenerator weiterleitet. Dieser generiert aus der Stellgröße ein Pulssignal mit dem Tastverhältnis α, welches an den MOSFET des Tiefsetzstellers angelegt wird. Der MOSFET wird entsprechend des Pulssignals geschaltet und es ergibt sich somit die benötigte Stromstärke $I = \frac{\alpha \cdot U_\mathrm{E}}{R_\mathrm{Last}}$ am Magneten.

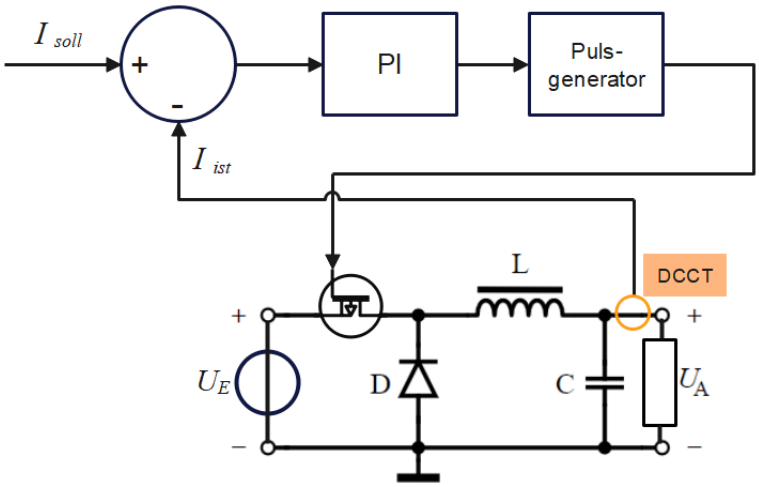

Abb. 4.1. Übergeordnetes System Magnetstromregelung mit PI-Regler, Pulsgenerator, DCCT-Sensor und Tiefsetzsteller[38]

Zur Umsetzung der in Abbildung 4.1 gezeigten Regelung wird im Beschleuniger PETRA III ein SoC von Altera[39] genutzt. Im Rahmen der Neuentwicklung der digitalen Magnetstromregelung für den Beschleuniger PETRA IV soll ebenfalls ein SoC genutzt werden,

[1] Abweichung vom Sollwert < 10 ppm

um diese darauf zu implementieren. Die Stromregelung soll dabei auf dem FPGA des SoCs implementiert werden, da so eine höhere Schnelligkeit der Regelung erreichbar ist. Die Ausgangswerte des Stromsensors sollen dort eingelesen werden und dienen als Eingang für den Regelkreis. Um die Ausgangsdaten der Regelung für Benutzer*innen sichtbar zu machen, soll auf dem Prozessorkern ein Server implementiert werden. Für die Implementierung dieses Servers wurde der Standard OPCUA[40] gewählt, um die Übertragung von Informationen auf dem ganzen DESY-Gelände zu vereinheitlichen. Auf dem Prozessorkern soll ein Betriebssystem installiert werden, das als Grundlage für den OPCUA-Server und den Zugriff auf die Daten im FPGA dient.
Von der Messung des Magnetstroms bis zur Ausgabe der Regelungsparameter in einem Clientprogramm spannt sich somit, wie in Abbildung 4.2 zu sehen eine Kette der Datenkommunikation auf.

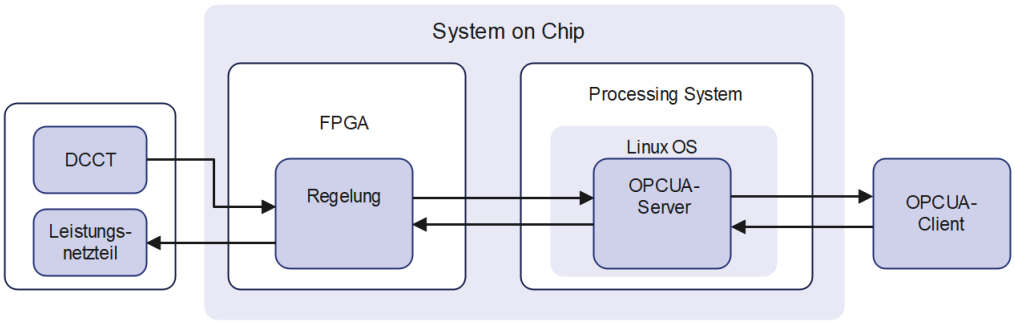

Abb. 4.2. Funktionale Kette der Datenkommunikation

4.2. Stakeholderanalyse

Die funktionale Kette der Datenkommunikation bildet verschiedene Funktionen ab, welche in den Verantwortungsbereichen unterschiedlicher Stakeholder liegen. Aus der Stakeholder-Analyse ergeben sich die Anforderungen an die aufzustellende Datenkommunikation.

4.2.1. Entwickler der Magnetstromregelung auf dem FPGA

Aus der Analyse des ersten Stakeholders gehen die funktionalen Anforderungen an die Datenkommunikation hervor (siehe Tabelle 4.2). Die Regelung befindet sich aktuell noch in der Entwicklungsphase, daher ist die Art der Implementierung noch nicht eindeutig festgelegt. Sie soll entweder vollständig auf dem FPGA implementiert oder teilweise auf den Prozessor ausgelagert werden.
Um die Flexibilität für die Entwicklung der Regelung zu bewahren, soll während dem Entwurf der Datenkommunikation davon ausgegangen werden, dass ein Teil der Regelung auf den Prozessorkern ausgelagert wird und die Datenkommunikation den zeitlichen Anforderungen der Regelung genügen muss.

Die Regelung selbst unterliegt einer harten Echtzeitanforderung, welche bei der Implementierung der Datenkommunikation beachtet werden muss. Der Takt der Regelung beträgt 132 kHz. Dieser Wert muss genauestens eingehalten werden, damit die Schaltfrequenz andere Systeme nicht stört. Im Falle einer zum Teil auf den Prozessorkern ausgelagerten Regelung, muss die Datenkommunikation zwischen FPGA und Prozessorkern ebenfalls diesen Echtzeitanforderungen genügen. Die zu verarbeitenden Signale müssen vom FPGA zum Prozessorkern gesandt, dort entsprechend verarbeitet und zurück zum FPGA gesendet werden. Zum aktuellen Stand der Entwicklung liegen noch keine genaueren Informationen über die zeitliche Performance der Regelung selbst vor. Es wird lediglich als Schätzwert vorgegeben, dass die Kommunikation nicht mehr als 1/20 des Taktes der Regelung in Anspruch nehmen soll. Damit muss die Datenkommunikation zwischen FPGA und Prozessorkern für den Hin- und Rückweg insgesamt schneller als $\frac{1}{20} \cdot \frac{1}{132kHz} \approx 0{,}38\ \mu s$ sein.

Innerhalb eines jeden Taktes der Regelung muss eine bestimmte Datenmenge zwischen FPGA und Prozessorkern hin und her gesendet werden. Der Prozessorkern benötigt vom FPGA die aktuellen Strom- und Spannungswerte und Statuswerte. An den FPGA werden Sollwerte, Statuswerte und weitere Regelparameter gesendet. Diese Werte müssen jeweils pro Regelungstakt zum Prozessorkern gesendet, dort verarbeitet und dem Regelkreis auf dem FPGA wieder zur Verfügung gestellt werden. Zum aktuellen Stand der Entwicklung ist noch nicht bekannt wie viele Daten, in welcher Dimension jeweils pro Takt versendet werden müssen. Es wurden Schätzwerte, wie in Tabelle 4.1 zu sehen für den Entwurf der Datenkommunikation vorgegeben. Diese Schätzwerte enthalten einige Datenworte der Größe 32 Bit und 16 Bit, welche Stromwerte, Spannungswerte oder andere größere Daten der Berechnung bzw. Regelung repräsentieren. Außerdem wird der Gebrauch von vielen 1-Bit-großen Datenworten für Status- und Steuersignale vermutet.

Anzahl der zu übertragenden Daten	Größe der Datenwörter in Bit
10	32
5	16
50	1

Tab. 4.1: Pro Regelungstakt zu übertragendes Datenvolumen

Die beschriebenen Anforderungen des Stakeholders sollen in der zu entwickelnden Datenkommunikation umgesetzt werden. Dazu sollen geeignete Kommunikationsinterfaces gefunden werden. Die zeitliche Performance der aufzustellende Datenkommunikation soll, insbesondere bezüglich der teilweisen Auslagerung der Regelung auf den Prozessorkern, untersucht werden.

Stakeholder	Entwickler der Magnetstromregelung auf dem FPGA
Anforderungen an die Datenkommunikation	• Flexibilität und Erweiterbarkeit der Datenkommunikation • Implementierung und Validierung verschiedener Kommunikationsinterfaces • Implementierung der Datenkommunikation für teilweise Auslagerung der Regelung auf den Prozessor • Echtzeitanforderung an Datenkommunikation <0,38 µs • Festgelegtes Datenvolumen pro Regelungstakt • Untersuchung der zeitlichen Performance der Datenkommunikation

Tab. 4.2: Stakeholderanalyse - Entwickler der Magnetstromregelung auf dem FPGA

4.2.2. Entwickler der allgemeinen Magnetstromversorgung

Aus der Analyse des zweiten Stakeholders gehen die nicht-funktionalen Anforderungen hervor (siehe Tab. 4.3). Diese ergeben sich im Wesentlichen aus der Tatsache, dass ein Beschleuniger ein Großprojekt ist, welches viele materielle und finanzielle Ressourcen beansprucht. Da die zuverlässige Funktion der Anlage von den verbauten Komponenten abhängig ist, muss garantiert werden, dass diese auch auf lange Zeit die gewünschte Funktionalität liefern. Das obsolet werden von Softwarekomponenten könnte in diesem Fall zur Gefährdung der zuverlässigen Funktion der Gesamtanlage führen. Aus diesem Grund soll es sich bei dem Betriebssystem auf dem Prozessorkern um eine Linuxversion handeln. In der Vergangenheit sei es oft zu Problemen mit erzwungenen Updates von Closed-Source-Betriebssystemen gekommen. Der vollständig vorliegende Source-Code ermöglicht außerdem eine auf das spezielle System zugeschnittene Konfigurierung. Jedoch stellte sich zu Beginn des Projekts die Frage, ob und wie gut sich die gewünschte Funktionalität auch ohne Betriebssystem auf dem SoC umsetzen lässt. Daher soll bei der Entwicklung die Implementierung und Nutzung mit und ohne Betriebssystem durchgeführt und evaluiert werden.

In der bisherigen Magnetstromregelung wurde als Hardware-Description-Language AHDL[41] verwendet. Für die neue Magnetstromregelung soll die relativ neue Sprache SystemVerilog[42] verwendet werden.
Mit diesen beiden Anforderungen soll vor allem garantiert werden, dass die Anlage auch im Verlaufe der nächsten Jahre betriebsfähig ist und nicht aufgrund veralteter Betriebssysteme und Programmiersprachen funktionsunfähig wird.

Die Programmiersprache für die Software auf dem Processing System soll C++ sein, da der Code des OPCUA-Servers, der später dort implementiert werden soll ebenfalls in C++ geschrieben wurde. Die objektorientierte Programmiersprache eignet sich außerdem sehr gut für die Anwendung in einem solch großen Projekt wie diesem. Die Abstraktion erlaubt, dass verschiedene Teile der Software von verschiedenen Teams erstellt und

später zusammengestellt werden können.

Stakeholder	Entwickler der allgemeinen Magnetstromversorgung
Anforderungen an die Datenkommunikation	• Betriebssystem: Linux • HDL: SystemVerilog • Programmiersprache: C++

Tab. 4.3: Stakeholderanalyse: Entwickler der allgemeinen Magnetstromversorgung

4.2.3. Projektleiter der Magnetstromversorgung

Der dritte Stakeholder (siehe Tab. 4.4), der Projektleiter der Magnetstromversorgung muss garantieren, dass der finanzielle und zeitliche Rahmen des Projekts eingehalten wird. Aufgrund der aktuellen Marktlage der letzten Jahre gab es oft Beschaffungsprobleme von elektronischen Komponenten. Für die Neuentwicklung der Magnetstromversorgung soll daher sichergestellt werden, dass die Gruppe nicht abhängig von einer Plattform ist. Die Magnetstromregelung von PETRA III wurde mit einem SoC von Altera umgesetzt. Für die Neuentwicklung soll nun neben Altera auch ein SoC von Xilinx in Betrieb genommen und so eine mögliche Ausweichoption geschaffen werden. Zu diesem Zweck soll das SoC-Modul Mercury XU5 von Enclustra in Betrieb genommen und hinsichtlich des Einsatzes im beschriebenen System (siehe Abschnitt 4.1) untersucht werden.

Stakeholder	Projektleiter der Magnetstromversorgung
Anforderungen an die Datenkommunikation	• Entwicklungsplattform: Mercury XU5 mit Zynq UltraScale+ MPSoC von AMD Xilinx

Tab. 4.4: Stakeholderanalyse: Projektleiter der Magnetstromversorgung

4.3. Zusammenfassung des Anforderungen

Aus der Stakeholder-Analyse gehen die Anforderungen für diese Bachelorarbeit hervor. Zwischen FPGA und Processing-System auf der Platform Mercury XU5 ist eine Datenkommunikation zu implementieren. Dazu soll ein geeignetes Kommunikationsinterface gefunden, implementiert und evaluiert werden. Das Interface soll bidirektional Daten zwischen den beiden Einheiten übertragen. Dabei müssen große Datenmengen innerhalb von 0,38 µs vom Processing-System ausgelesen und wieder zum FPGA geschrieben werden können. Bei der Hardware-Description-Language soll es sich um SystemVerilog handeln.
Auf dem Prozessorkern ist ein Linux-Betriebssystem zu installieren. Auf diesem soll eine Software die Lese- und Schreibbefehle zum FPGA ausführen. Dabei soll die Programmiersprache C++ genutzt werden. Während des gesamten Arbeitsprozesses sollen

die Unterschiede zwischen der Implementierung des Interfaces mit und ohne Linux-Betriebssystem untersucht werden. Die aufgestellte Datenkommunikation soll bezüglich Der Eignung für die teilweise Auslagerung der Regelung auf den Prozessorkern auf ihre zeitliche Performance untersucht werden.

5. Konzept

Um die Aufgabenstellung zu erfüllen, soll eine Datenkommunikation entworfen und implementiert werden, die im Hinblick auf den Einsatz im besprochenen System (siehe Abschnitt 4.1) bewertet werden soll. Für die Implementierung soll nun ein Konzept entworfen werden, welches zum Ziel hat die besprochenen Anforderungen zu erfüllen.

5.1. Die Xilinx Entwicklungsumgebung

Im Rahmen der Bachelorarbeit soll die Plattform Mercury XU5 in Betrieb genommen und die Xilinx-Umgebung als mögliche Ausweichoption für spätere Projekte untersucht werden. Daher sollen für die Entwicklung der Datenkommunikation die Developer Tools von Xilinx genutzt werden.
Mit der Vivado Design Suite und dem dazugehörigen IP-Integrator soll das FPGA-Design erstellt und synthetisiert werden. Für das Software-Design soll das Tool Vitis Unified Software Platform genutzt werden. Mit dem zuvor in Vivado generierten Hardware-Specification-File können dort auf Basis des FPGA-Designs Platform-Projekte erstellt werden. So kann in daraufhin erstellten Application-Projekten auf die Hardware zugegriffen werden. Für das Application-Projekt ist als Grundlage ein C++-Projekt auszuwählen. So werden beim Build des Application-Projekts die C++-Bibliotheken automatisch eingebunden. In Vitis kann für Bare-Metal-Projekte ein Boot-Image erstellt werden,[21] sodass das fertige Projekt inklusive Bitstream und C-Code aus Vitis heraus auf den SoC geladen werden kann.

Für die Inbetriebnahme mit Linux ist das Konzept ähnlich. Das Boot-Image wird separat mit der Enclustra Build Environment erstellt (siehe Abschnitt 5.5). Bei dem in Vitis zu erstellenden Platform-Projekt muss lediglich Linux als Betriebssystem ausgewählt werden. Beim Build des Application-Projekts wird dann von Vitis eine .ELF-Datei erstellt. Dies ist eine ausführbare Datei und kann in das Linux-Filesystem auf dem Prozessorkern geladen und dort ausgeführt werden.

Für das Laden von solchen Linux-Applications auf den Prozessorkern soll aufgrund der Schnelligkeit Ethernet genutzt werden. Dies hat den Vorteil, dass während dem Testen von Software die SD-Karte (siehe Abschnitt 5.5) nicht ständig zwischen SoC und Entwicklungs-PC hin- und her gesteckt werden muss. Des Weiteren wird für die Ent-

wicklung ein Windows-PC verwendet. Daher soll für den Transfer die Software WinSCP[43] genutzt werden. Diese ermöglicht es von einem Windows-PC über die IP-Adresse eine remote Verbindung zum SoC aufzubauen und Dateien zu senden.

5.2. Das AXI-Interface

Zwischen FPGA und Prozessorkern soll ein fortlaufender Datenaustausch stattfinden. Dazu muss ein Kommunikationsinterface zwischen diesen beiden Einheiten aufgebaut werden, mit welchem Daten in beide Richtungen übertragen werden können. Xilinx bietet für den Austausch von Daten zwischen einem Manager und einem Subordinate das AXI4-Interface an. Das AXI-Interface teilt sich in drei Unterarten auf: AXI4-Stream, AXI4 und AXI4-Lite.

AXI4-Stream ist das schnellste Interface, da es im Gegensatz zu den beiden anderen ohne Memory-Mapping arbeitet und so das Übertragen einer Adressphase einspart. Jedoch ist AXI4-Stream nur unidirektional nutzbar und kommt daher für den bidirektionalen Austausch zwischen FPGA und Prozessorkern nicht in Frage.

AXI4 und die vereinfachte Version AXI4-Lite arbeiten mithilfe von Memory-Mapping und weisen daher eine etwas niedrigere Übertragungsgeschwindigkeit, als AXI4-Stream auf. Die Übertragung der Adressphase hat jedoch den Vorteil, dass sie bidirektional Daten übertragen können.
AXI4-Lite beinhaltet nur einen Teil der Kontrollsignale des AXI-Protokolls. Es ist daher nur zur Ausführung von einem Datentransfer pro Transaktion fähig. Jedoch nimmt es weniger Platz in einem FPGA-Design ein und ist einfacher in dieses einzubinden. Das AXI4 (full)-Interface unterstützt alle Kontrollsignale des AXI-Protokolls und ist daher detaillierter zu konfigurieren. Dies geht mit höherer Performance aber auch einem komplexeren FPGA-Design einher. Es ist außerdem dazu in der Lage mehrere Datenwörter innerhalb einer Transaktion zu übertragen.
Für den Aufbau der Datenkommunikation zwischen FPGA und Processing System soll zunächst das AXI4-Lite-Interface genutzt werden, da die Implementierung dessen sowohl im FPGA-Design als auch in der Software auf dem Processing System weniger komplex ist.

5.3. Das AXI-Subordinate

Das AXI-Interface wird zwischen einem AXI-Manager und einem AXI-Subordinate aufgebaut. Der AXI-Manager ist der Prozessorkern. Er sendet Lese- und Schreibbefehle an das AXI-Subordinate. Diese Befehle sollen Daten aus der Regelung auslesen und ihr Da-

ten zur Verfügung stellen. Es muss also ein Logikmodul auf dem FPGA erstellt werden, in welchem später die Regelung implementiert werden kann. Um die Funktionalität und Performance der Datenkommunikation feststellen zu können, soll in diesem Modul eine simple Logik aufgebaut werden. Diese Logik soll Daten vom Eingang des Subordinate erhalten, diese mit zwei multiplizieren und wieder an den Ausgang des Moduls legen. Die Logik soll deshalb so simpel sein, da die Performancemessungen der Datenkommunikation nicht durch etwaige Zeitverzögerungen einer zu komplexen Beispiellogik beeinflusst werden sollen.

Damit das Modul als AXI-Subordinate vom AXI-Manager angesprochen werden kann, benötigt es ein AXI-Interface.
Xilinx bietet verschiedenen IP-Cores an, die zur Datenübertragung über das AXI-Interface entwickelt wurden. Zum einen gibt es für das Transferieren von Daten zwischen dem internen Speicher des SoC und dem FPGA den AXI-GPIO-IP-Core. Dieser dient als Interface zwischen General-Purpose-Ein- und Ausgängen eines Peripheriemoduls und dem AXI4-Lite-Interface und nimmt die Rolle des AXI-Subordinate ein.
Der AXI-Manager kann somit die Register des AXI-Subordinates, also des AXI-GPIO auslesen und beschreiben. Die Ein- und Ausgänge des IP-Cores, bei denen es sich nicht um AXI-Interface-Ports handelt, werden mit den Ein- und Ausgängen des Logik-Moduls verbunden. So kann die Software auf dem Prozessorkern über den AXI-GPIO-IP-Core mit der Logik auf dem FPGA kommunizieren.

Das AXI-GPIO-Modul kann im Dual-Channel-Modus konfiguriert werden. Ein Channel wird mit dem Eingang, der zweite Channel mit dem Ausgang des Logik-Moduls verbunden. Dies dient sowohl der Übersicht im FPGA-Design, als auch in der Software. Bei der Nutzung mehrerer AXI-GPIOs kann somit eine Zugehörigkeit von bestimmten Ports bzw. Größen zueinander geschaffen werden, indem sie mit demselben AXI-GPIO verbunden sind.

Ein weiterer IP-Core von Xilinx, welcher Daten vom Systemspeicher zu Peripheriemodulen überträgt, ist der AXI-DMA. Dieser kann Daten über ein Memory-Mapped-AXI-Interface vom Processing-System empfangen oder zu diesem senden. Über ein Memory-Mapped-to-Stream-Interface werden diese Daten dann zu einem Peripheriemodul geschrieben bzw. über ein Stream-to-Memory-Mapped-Interface von diesem gelesen. Auf diese Weise können große Datenmengen direkt aus dem Hauptspeicher an das Peripheriemodul übertragen werden.[44] Um auf dieser Art Daten an das Logik-Modul übertragen zu können, benötigt dieses ein AXI-Stream-Interface, mit welchem der DMA kommunizieren kann. Diese Übertragungsart beinhaltet die Konfigurierung vieler Komponenten, wie dem AXI-DMA und dem Memory-Controller.[45] Außerdem muss im Logik-Modul ein AXI-Wrapper geschrieben werden, damit zwischen den Komponenten eine AXI-Kommunikation stattfinden kann. Aus diesem Grund wird die Datenübertragung

mit dem AXI-DMA-IP-Core in dieser Arbeit nicht näher untersucht.
Als AXI-Subordinate wird an dieser Stelle daher der AXI-GPIO-IP-Core für den Aufbau der Datenkommunikation gewählt. Bei dieser Art der Datenkommunikation ist lediglich der AXI-GPIO-Core zu konfigurieren, welcher außerdem bereits ein Interface zwischen den Ein- und Ausgängen des Peripheriemoduls und dem AXI-Interface bietet.

5.4. Der AXI-Manager

Der Prozessor soll Daten vom AXI-Subordinate lesen, diese verarbeiten und Daten zurück zum AXI-Subordinate senden. Daher muss eine Software geschrieben werden, die auf dem Prozessorkern ausgeführt wird und von dort aus Lese- und Schreibbefehle an das AXI-Subordinate senden kann. Dazu muss die Software einen Zugriff auf das zuvor erstellte FPGA-Design haben, um in bestimmten Adressbereichen Daten zu lesen und zu schreiben. Das in dem FPGA-Design implementierte AXI-GPIO-Modul muss in der Software konfiguriert werden. Der Adressbereich des Moduls und die Datenrichtung der Channel muss festgelegt werden.

Da das AXI-Manager-Interface des Processing-Systems ein AXI4(-full)-Interface ist, soll auch das Schreiben von Daten im Burst-Modus und damit eine andere Art von Speicherzugriff, als beim AXI-GPIO-Core implementiert und getestet werden.
Für die Nutzung des AXI-GPIO-Moduls mit und ohne Betriebssystem müssen jeweils verschiedene Programme geschrieben werden. Der Zugriff ohne ein Betriebssystem auf die Register des Moduls geschieht mithilfe von Xilinx-Libraries.
Wird die Software in einem Betriebssystem ausgeführt, müssen Linux-Driver genutzt werden, um auf die Register zugreifen zu können. Die zu manipulierenden Register des AXI-GPIO-Moduls müssen zuvor in ein virtuelles Linux-Filesystems exportiert werden, um diese von der Software beschreiben zu können.

5.5. Das Linux-Betriebssystem

Auf dem Prozessorkern soll später ein OPCUA-Server laufen. Für dessen Implementierung wird ein Linux-Betriebssystem benötigt. Dieses soll auf dem SoC installiert werden. Xilinx bietet zwar auch an dieser Stelle Unterstützung durch ihre PetaLinux-Tools,[46] jedoch soll zur Erstellung des Embedded Linux an dieser Stelle das Tool Enclustra Build Environment[47] (EBE) genutzt werden. Für den Build eines Embedded Linux-Systems werden hardwarespezifische Dateien wie beispielsweise Device-Tree-Files benötigt. Da es sich bei dem vorliegenden System um ein SoC von Enclustra handelt, ist dieses Tool zur ersten Inbetriebnahme des Boards mit Linux die bessere Wahl. Das Tool greift auf ein GitHub-Repository[48] zu, in welchem alle benötigten hardwarespezifischen Dateien

zur Generierung des Boot-Images für alle Enclustra Targets vorliegen. Die EBE erstellt auf Basis der hardwarespezifischen Dateien ein Boot-Image für das Booten des SoC in Linux. Das Target, der Boot-Modus und die benötigten Boot-Dateien sind in einer GUI einfach auszuwählen. Ein weiterer Vorteil ist die Anpassungsmöglichkeit an rekonfigurierte Hardware. Die Bitstream-Datei kann in der GUI einfach gegen selbst geschriebene Bitstream-Files ausgetauscht werden. Wenn im Nachhinein eine andere Linuxversion verwendet werden soll, kann das Image ausgetauscht und so in einer anderen Version gebootet werden.

Zunächst können so alle benötigten Dateien zum Booten in Linux für das Reference Design für die Mercury-XU5-ST1-Plattform[49] erstellt werden. Nach dem Hinzufügen von weiterer Hardware, wie dem AXI-GPIO-Modul muss das Device-Tree-File um diesen erweitert werden, damit vom Linux-Betriebssystem auch diese nicht detektierbare Hardware manipuliert werden kann.

Als Boot-Medium soll während der Entwicklung die SD-Karte verwendet werden. Diese bietet die Möglichkeit Dateien schnell auszutauschen und im SoC zu testen. Dies kann sich während der Entwicklung als sehr nützlich erweisen, da währenddessen aufgrund von Änderungen am FPGA-Design das Boot-Image ständig erneuert werden muss. Im späteren Betrieb eignen sich dann andere Boot-Modi wie beispielsweise QSPI-Flash besser, da dieser eine höhere Bootgeschwindigkeit aufweist[50] und SD-Karten eine beschränkte Lebensdauer besitzen.

5.6. Validierung der Datenkommunikation

Um die entwickelte Datenkommunikation bewerten zu können, soll die Funktionalität und die Performance dieser gemessen werden. Zunächst soll die Funktion des AXI-Protokolls zwischen AXI-Manager und AXI-Subordinate verifiziert werden. Dazu soll der korrekte Ablauf der Adress- und Datenphasen und der Verlauf der Signale auf den fünf AXI-Kanälen untersucht werden. Für diesen Zweck soll das Tool Integrated-Logic-Analyzer[51] genutzt werden. Bei diesem handelt es sich um einen IP-Core von Xilinx, welcher in das zu untersuchende FPGA-Design eingefügt wird. Die zu untersuchenden Signale werden dazu an die Slots des ILA-Cores angeschlossen. Die Software kann anschließend von Vitis aus gestartet werden. In Vivado können im Hardwaremanager über ein Waveform-Fenster alle Signale beobachtet werden, die mit dem ILA-Core verbunden wurden. Die Messung muss dabei plattformübergreifend gestartet werden, da der Befehl des Hardwarezugriffs auf dem Prozessorkern geschieht, während sich das AXI-Subordinate auf dem FPGA befindet.

Um mit dem ILA-Core die Funktionalität des AXI-Interfaces zu verifizieren, muss der

ILA-Core jeweils an die Signale zwischen AXI-GPIO und SmartConnect und zwischen SmartConnect und Zynq-Processeing-System angeschlossen werden. Mit dieser Konfigurierung können die Signale des AXI-Interfaces entlang der gesamten Datenkommunikation zwischen AXI-Manager und AXI-Subordinate untersucht werden. Auch der zeitliche Verlauf der Signale des AXI-Protokolls kann mit dem ILA-Core beobachtet werden, da die x-Achse des Waveform-Windows den Takten des Clock-Signals des ILA-Cores entspricht. Daher muss der ILA-Core mit demselben Clock-Signal verbunden werden, wie das System, dass beobachtet werden soll.[52]

Außerdem soll der zeitliche Einfluss des Hardwarezugriffs vom Prozessor untersucht werden. Dabei soll besonders auf den Unterschied des Zugriffs mit und ohne Betriebssystem geachtet werden.
Zur Messung dieses Unterschieds wird im Logikmodul des FPGAs ein Counter implementiert, der die Taktzyklen während einer Lese- und Schreibtransaktion zählt. Diese Lese- und Schreibtransaktion wird dann auf dem Prozessorkern mit und ohne Linux-Betriebssystem ausgeführt.

6. Implementierung

Die Datenkommunikation zwischen Processing System und Programmable-Logic wird nun auf Grundlage des entworfenen Konzepts implementiert.

6.1. Erstellung der Entwicklungsumgebung

Zur Implementierung der entworfenen Datenkommunikation sollen die Xilinx Developer Tools verwendet werden. Der Entwurf des FPGA-Designs erfolgt mit der Vivado Design Suite. Um Hardwarekomponenten zu designen und das Logik-Modul zu schreiben, muss zunächst ein Projekt in Vivado erstellt werden, das auf der board-spezifischen Hardware des Mercury XU5 basiert. Für die Erstellung dieses Projekts wird das Reference Design von Enclustra genutzt. Das dazugehörige GitHub-Repository enthält die dazu nötigen TCL-Skripte und das Haupt-VHDL-Modul Mercury_XU5_ST1.vhdl,[49] welches den Mercury XU5 beschreibt. Die TCL-Skripte (Tool Command Language) erstellen mithilfe des VHDL-Moduls und weiteren Konfigurationsdateien ein Projekt, in dem die VHDL-Datei und ein Blockdesign der Hardware des Reference Designs vorliegt.
Die Hauptkomponente des Blockdesigns ist das Processing System, das durch den PS UltraScale+ Block (siehe Abb. 6.1) repräsentiert wird. Dieser besitzt bereits einige nach außen geführte Signale wie unter anderem Taktsignale und ein AXI-Manager-Interface.

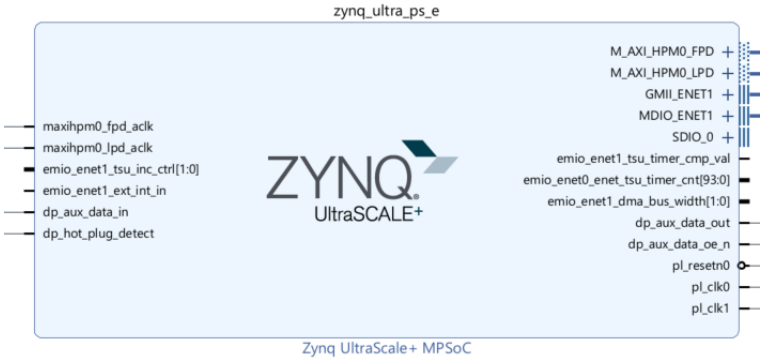

Abb. 6.1. PS-UltraScale+ -Block aus dem Vivado-Blockdiagramm[49]

6.2. Erstellung eines Logik-Moduls

Die Regelung soll auf dem FPGA in SystemVerilog implementiert werden. Dazu wird eine weitere Quell-Datei zum Projekt hinzugefügt, in der später die Regelung implementiert werden kann. Um eine vollständige Kommunikation aufzubauen, wird an dieser Stelle stellvertretend für die Regelung eine simple Logik implementiert, die die Eingangsdaten mit zwei multipliziert und wieder ausgibt.
Die genutzt Version Vivado 2022.1 unterstützt im Allgemeinen die Synthese von SystemVerilog, jedoch können mit SystemVerilog keine Top-Level-RTL-Module definiert werden.[17] Daher wird zusätzlich ein Verilog-Wrapper geschrieben. Dabei handelt es sich um eine weitere Quell-Datei, die die zuvor geschriebene SystemVerilog-Datei als Modul instanziiert. Die Ein- und Ausgänge des SystemVerilog-Moduls werden dabei den Ein- und Ausgängen des Verilog-Wrappers zugeordnet. Im weiteren Verlauf der Arbeit wird die Gesamtheit dieser beiden Module als Logik-Modul bezeichnet.

6.3. Implementierung der AXI-Kommunikation mit dem AXI-GPIO

Das FPGA-Design enthält nun die beiden Komponenten, zwischen welchen die Datenkommunikation aufgebaut werden soll. Der Block des Processing Designs besitzt bereits ein AXI-Manager-Interface. Damit das Processing System mit dem Logik-Modul kommunizieren kann, benötigt dieses ein AXI-Subordinate-Interface. Daher wird der AXI-GPIO-IP-Core in das Blockdesign eingefügt. Dieser wird zunächst entsprechend der Anforderungen konfiguriert. Um beide GPIO-Kanäle nutzen zu können, wird der Dual-Channel-Modus freigeschaltet. Für die Datenbreite der Kanäle wird 32 Bit eingestellt. Der erste Kanal wird als Ausgang und der zweite Kanal als Eingang konfiguriert.[53]
Die GPIO-Ports des AXI-GPIO-Cores werden mit den Ein- und Ausgängen des Logik-Moduls verbunden. Der Ausgang wird mit einem Eingang des Logik-Moduls verbunden, sodass das Processing-System über den AXI-GPIO-Core Daten an dieses senden kann. Der Eingang des AXI-GPIO-Cores wird mit dem Ausgang des Logik-Moduls verbunden. Auf diese Weise kann das Processing-System Daten von diesem auslesen. Für die Verbindung mit dem Processing System wird die Auto-Connect-Funktion von Vivado genutzt.[15] Beim Ausführen dieser Funktion werden das Taktsignal s_axi_aclk, das Resetsignal s_axi_aresetn und das AXI-Interface S_AXI automatisch mit dem Processing System verbunden (siehe Abb. 6.2). Bei der automatischen Verbindung zwischen AXI-Manager und AXI-Subordinate wird ein AXI-Smartconnect im Blockdesign generiert. Dies ist eine neue Version des AXI-Interconnects, welche zwei Blöcke mit Manager- und Subordinate-Interface miteinander verbindet.[54]

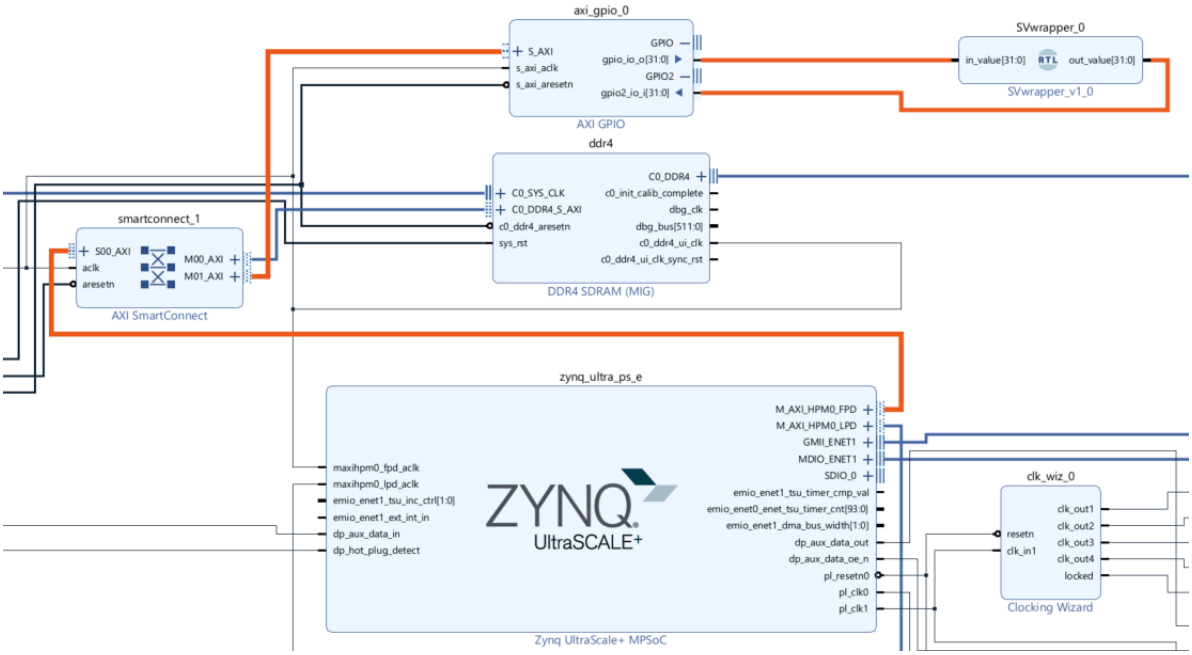

Abb. 6.2. Ausschnitt des Blockdiagramms mit Zynq UltraScale+, SmartConnect, AXI-GPIO-Core und Logik-Modul

Die Anforderungen an die Datenkommunikation sehen die Übertragung eines gewissen Datenvolumens innerhalb eines Regelzyklus vor. Daher muss auf der Seite des Subordinate eine Struktur geschaffen werden, die diesen Datensatz empfangen kann. Die Daten müssen über diese Struktur vom AXI-Manager wieder ausgelesen werden können. Ein einzelner AXI-GPIO-Core eignet sich dazu nicht. Der AXI-Manager könnte zwar viele Daten hintereinander an den AXI-GPIO-Core senden, jedoch besitzt dieser nur zwei GPIO-Ports. Diese sind mit dem Logik-Modul verbunden, welches die Daten vom AXI-GPIO-Core annehmen bzw. an diesen ausgeben soll. Bei den übertragenden Daten handelt es sich jedoch um verschiedene Größen (siehe Abschnitt 4.2.1), die getrennt voneinander abgespeichert werden müssen. Es müsste daher im Logik-Modul eine Logik implementiert werden, die die Daten von diesen zwei Ports annimmt und richtig sortiert abspeichern kann. Für Lesezugriffe müssten diese Daten ebenfalls koordiniert an die zwei GPIOs des AXI-GPIO-Core ausgegeben werden. Eine solche Logik wäre sehr unübersichtlich und unflexibel in der Anwendung.

Daher werden, wie in Abbildung 6.3 zu sehen, mehrere AXI-GPIO-Cores implementiert, sodass das Logik-Modul einen festen Eingang für jede zu sendende Größe hat. Jeder Ein- und Ausgang wird entsprechend mit einem GPIO-Port vom AXI-GPIO-Core verbunden.

Im SystemVerilog-Modul (Abb. 6.4) und im Verilog-Wrapper-Modul (Abb. 6.5) werden entsprechend Ein- und Ausgänge ergänzt.

Abb. 6.3. Ausschnitt des Blockdiagramms mit SmartConnect, vier AXI-GPIO-Cores und Verbindung zum Logik-Modul

```
30 module CommunicationSV(
31
32     input clk,
33     input logic [31:0] data_in_1,
34     input logic [31:0] data_in_2,
35     input logic [31:0] data_in_3,
36     input logic [31:0] data_in_4,
37     output logic [31:0] data_out_1 = 0,
38     output logic [31:0] data_out_2 = 0,
39     output logic [31:0] data_out_3 = 0,
40     output logic [31:0] data_out_4 = 0
41     );
42
43     always @(posedge clk)
44     begin
45         data_out_1 <= data_in_1 * 2;
46         data_out_2 <= data_in_2 * 2;
47         data_out_3 <= data_in_3 * 2;
48         data_out_4 <= data_in_4 * 2;
49     end
50
51 endmodule
```

Abb. 6.4. SystemVerilog-Modul

```
23 module SVwrapper(
24
25     input wire clk_in,
26     input wire [31:0] in_1,
27     input wire [31:0] in_2,
28     input wire [31:0] in_3,
29     input wire [31:0] in_4,
30     output wire [31:0] out_1,
31     output wire [31:0] out_2,
32     output wire [31:0] out_3,
33     output wire [31:0] out_4
34     );
35
36     CommunicationSV Instance_0
37     (
38     .clk(clk_in),
39     .data_in_1(in_1),
40     .data_in_2(in_2),
41     .data_in_3(in_3),
42     .data_in_4(in_4),
43     .data_out_1(out_1),
44     .data_out_2(out_2),
45     .data_out_3(out_3),
46     .data_out_4(out_4)
47     );
48 endmodule
```

Abb. 6.5. Verilog-Wrapper für SystemVerilog-Modul

Um vom Processing System auf das zuvor generierte FPGA-Design zugreifen zu können, muss das Design in Vivado synthetisiert und ein Bitstream generiert werden. Die Hardware kann inklusive Bitstream in Form einer .xsa-Datei aus Vivado exportiert werden. Diese wird für die PS-Programmierung in Vitis benötigt. Der Bitstream wird außerdem nochmal separat aus Vivado exportiert, da dieser für die Erstellung eines Boot-Images

gebraucht wird.

6.4. Generierung des Linux-Boot-Image

Um das Processing-System mit einem Linux-Betriebssystem booten zu können, muss zuvor ein Boot-Image erstellt werden. Dazu wird das Tool Enclustra Build Environment genutzt. Es bietet ein Graphical User Interface (GUI) und ein Command Line Interface (CLI). In der GUI wird zunächst der Zynq UltraScale+, der Mercury XU5 und das Baseboard ST1 ausgewählt. Als Boot-Device wird die Multi Media Card (MMC) ausgewählt. In der Enclustra Build Environment können die zu erstellenden Boot-Dateien ausgewählt werden für den Fall, dass bei späteren Anpassungen nur bestimmte Dateien neu erstellt werden müssen. Für die erste Generierung des Boot-Images werden die Boot-Targets Linux, U-Boot und Buildroot[55] ausgewählt. Im nächsten Schritt wird der Bitstream des erstellten FPGA-Designs gegen die Default-Bitstream-Datei ausgetauscht. Mit diesen Einstellungen wird das Boot-Image generiert und die entstehenden Dateien werden in einen Output-Ordner abgelegt. Der Output-Ordner enthält unter anderem fünf Dateien, die für den Boot in einem Linux-Betriebssystem von der SD-Karte notwendig sind. Die Datei boot.bin enthält unter anderem den FSBL und den Bitstream. Die Datei Image enthält den Linux-Kernel[26] und die Datei rootfs.tar enthält das Linux-File-System. Außerdem werden die Datei devicetree.dtb und uboot.scr benötigt.
Auf der SD-Karte werden zwei Partitionen erstellt. Die erste Partition wird FAT-formatiert (File Allocation Table) und bildet die Boot-Partition, auf der die boot.bin-Datei, das Image, der Devicetree und die uboot.scr-Datei kopiert werden. Die zweite Partition wird ext4-formartiert und die rootfs.tar-Datei wird dorthin entpackt.[48]
Während der Implementierung stellte sich heraus, dass das Linux-File-System den zweiten Kanal eines AXI-GPIO-Cores zwar erkennen, aber nicht beschreiben oder auslesen kann. Daher wird das in Abschnitt 6.3 erstellte FPGA-Design so angepasst, dass die vier AXI-GPIO-Cores nur im Single-Channel-Modus betrieben werden. Zwei AXI-GPIO-Cores werden als Ausgang und die zwei anderen als Eingang genutzt.

6.4.1. Anpassung des Devicetree

Da im Vergleich zum Reference-Design, auf welchem die Default-Boot-Dateien basieren, Veränderungen im FPGA-Design vorgenommen wurden, muss der Devicetree dahingehend angepasst werden. Bei dem AXI-GPIO-IP-Core handelt es sich um für das Betriebssystem nicht-detektierbare Hardware, daher muss dieser dem Devicetree hinzugefügt werden. Unter der übergeordneten Node amba_pl wird eine weitere Node für den AXI-GPIO-Core erstellt (siehe Abb. 6.6). Dieser Node muss als erstes die Adresse der Hardware zugewiesen werden. Für den AXI-GPIO-Core handelt es sich dabei um die Basisadresse des GPIO-Cores.

Der Eigenschaft *#gpio-cells* wird der Wert 2 zugewiesen. Dieser Wert gibt an, wie viele Attribute angegeben werden müssen, wenn auf diese AXI-GPIO-Node beispielsweise im Rahmen einer Child-Node verwiesen werden sollte. Der Eigenschaft *compatible* werden mehrere Strings übergeben, die die Kompatibilität mit anderen Devices ausdrücken und dem System dazu dienen, passende Driver für das Device zu finden. Der Node muss außerdem das Attribut *gpio-controller* zugewiesen werden, welches diesen als ein GPIO-Device kennzeichnet, welches mehrere GPIO-Signale beinhaltet. Die Eigenschaft *reg* legt die Adresse und die Länge des Adressbereiches für das Device fest. Dort wird die Basis-Adresse und außerdem die Differenz von *High Address* und *Base Address* aus dem Vivado-Address-Editor angegeben. Darunter folgen Eigenschaften, die der Konfiguration des AXI-GPIO-IP-Cores in Vivado entsprechen.[56]

```
gpio@a0000000 {
    #gpio-cells = <0x02>;
    compatible = "xlnx,axi-gpio-2.0\0xlnx,xps-gpio-1.00.a";
    gpio-controller;
    reg = <0x00 0xa0000000 0x00 0x10000>;
    xlnx,all-inputs = <0x00>;
    xlnx,all-inputs-2 = <0x00>;
    xlnx,all-outputs = <0x01>;
    xlnx,all-outputs-2 = <0x00>;
    xlnx,dout-default = <0x00>;
        xlnx,dout-default-2 = <0x00>;
        xlnx,gpio-width = <0x20>;
        xlnx,gpio2-width = <0x20>;
        xlnx,is-dual = <0x00>;
        xlnx,tri-default = <0xffffffff>;
        xlnx,tri-default-2 = <0xffffffff>;
};
```

Abb. 6.6. Node im Devicetree-Code für einen AXI-GPIO-Core

Auf diese Weise werden die vier im FPGA-Design erstellten AXI-GPIO-Cores dem Devicetree zugefügt (siehe vollständigen Devicetree-Code im Anhang).

6.5. Programmierung des Processing Systems ohne Linux-Betriebssystem

Auf Basis der zuvor erstellten Hardware-Specification-Datei wird in Vitis ein Platform-Projekt erstellt. Im Feld Operating System wird dabei standalone, also ohne Betriebssystem ausgewählt. Nachdem das Platform-Projekt kompiliert wurde, kann auf Basis dessen ein Application-Projekt erstellt werden. In diesem wird als Grundlage ein C++-Projekt ausgewählt.

6.5.1. Programmcode für Single-Data-Transfer-Transaktionen

Für den Zugriff auf die erstellte Hardware ohne Betriebssystem werden Bibliotheken von Xilinx genutzt. Die Header-Datei *xparameters.h*.[57] definiert Makros für Adressen und grundlegende Eigenschaften der Hardwaresysteme. Für den AXI-GPIO-IP-Core ist, wie in Abbildung 6.7 zu sehen, der Adressbereich, basierend auf dessen Basisadresse und Einstellungen wie Interrupts und Dual-Channel-Modus, dort festgelegt.

```
/* Canonical definitions for peripheral AXI_GPIO_0 */
#define XPAR_GPIO_0_BASEADDR 0xA0010000
#define XPAR_GPIO_0_HIGHADDR 0xA001FFFF
#define XPAR_GPIO_0_DEVICE_ID XPAR_AXI_GPIO_0_DEVICE_ID
#define XPAR_GPIO_0_INTERRUPT_PRESENT 0
#define XPAR_GPIO_0_IS_DUAL 1
```

Abb. 6.7. Definition von Makros für den AXI-GPIO-Core 0 in der Datei xparameters.h

Um Daten lesen und schreiben zu können, muss zu Beginn die zu beschreibende Hardware in der Software initialisiert werden. Dazu werden zunächst zwei Variablen vom Typ *XGpio* erstellt, die für den Eingang bzw. den Ausgang des AXI-GPIO-Cores stehen sollen (siehe Abb. 6.8, Zeile 14 f.). Bei dem Datentyp *XGpio* handelt es sich um eine Struktur, welche in der Header-Datei *xgpio.h* definiert wird und die Attribute *BaseAddress*, *IsReady*, *InterruptPresent* und *IsDual* besitzt. Der Header Xgpio.h definiert außerdem Funktionen um diese Struktur zu bearbeiten. Um den Attributen Werte zuzuweisen, wird die Funktion *XGpio_Initialize()* angewandt (siehe Abb. 6.8, Zeile 20 f.). Der erste Übergabeparameter ist die Adresse der XGpio-Strukturen. Der zweite Parameter ist das Makro *XPAR_AXI_GPIO_x_DEVICE_ID*, welches im Header *xparameters.h* definiert wird und angibt mit welchem AXI-GPIO die Struktur initialisiert werden soll. Daraufhin wird mit der Funktion *XGpio_SetDataDirection* die Datenrichtung für die Struktur festgelegt, dessen Adresse als erste Parameter übergeben wird (siehe Abb. 6.8, Zeile 23 f.). Bei dem zweiten Parameter handelt es sich um den zu nutzenden Kanal des AXI-GPIOs. Der dritte Parameter legt die Datenrichtung fest, wobei 0 für Output und 1 für Input steht. Um Daten zu schreiben, wird die Funktion *XGpio_DiscreteWrite* verwendet (siehe Abb. 6.8, Zeile 27). Diese erwartet ebenfalls als ersten Parameter die Adresse auf eine XGpio-Struktur und als zweiten Parameter den zu beschreibenden Kanal des AXI-GPIOs. Der dritte Übergabeparameter ist der zu schreibende Wert. Das Lesen von Daten geschieht mithilfe der Funktion *XGpio_DiscreteRead()*. Diese erwartet als ersten Parameter die Adresse einer XGpio-Struktur und als zweiten Parameter die Kanalnummer. Die Funktion gibt die ausgelesenen Daten mit dem Rückgabewert an die Software zurück (siehe Abb. 6.8, Zeile 30 ff).

In Abbildung 6.8 ist die Konfigurierung und das Schreiben und Lesen von Werten bezüglich einem AXI-GPIO als Beispiel dargestellt. Mit den anderen drei AXI-GPIO-Cores wird genauso verfahren (siehe vollständigen Code im Anhang).

```
10 int main()
11 {
12     init_platform();
13
14     XGpio input_0;
15     XGpio output_0;
16
17     int axi_0 = 0;
18
19     //configuration of AXI-GPIO 0
20     XGpio_Initialize(&input_0, XPAR_AXI_GPIO_0_DEVICE_ID);
21     XGpio_Initialize(&output_0, XPAR_AXI_GPIO_0_DEVICE_ID);
22
23     XGpio_SetDataDirection(&input_0,2,1);//instance, channel, direction(1:input, 0:output)
24     XGpio_SetDataDirection(&output_0,1,0);
25
26     //write
27     XGpio_DiscreteWrite(&output_0, 1, 1);
28
29     //read
30     axi_0=XGpio_DiscreteRead(&input_0,2);
31
32     printf("axi_0: %i\n",axi_0);
33
34     cleanup_platform();
35     return 0;
36
37 }
```

Abb. 6.8. Konfigurierung des AXI-GPIO-Core 0 und Senden von Schreib- und Lesebefehlen

6.5.2. Programmcode für Burst-Transaktionen

Der größte Unterschied zwischen Burst-Transaktion und Single-Transfer-Transaktionen ist der Speicherzugriff. Zuvor wurden die Daten in den Peripherie-Speicher geschrieben. Nun werden sie in den System-Speicher geschrieben. Die Technik des Memory-Caching, also das vorübergehende Speichern von Daten im Hauptspeicher, erlaubt einen schnelleren Zugriff auf die Daten.[34]

Diese Implementierung soll zum Austesten des Burst-Modes auf Seiten des Processing-Systems dienen, da dieses ein AXI4 (full)-Interface besitzt. Der AXI-GPIO-Core unterstützt aufgrund seines AXI4-Lite-Interfaces keine Burst-Transaktionen. Im Folgenden Programmcode wird ein Burst mit der Gesamtlänge 16 Byte, bestehend aus vier Integern (jeweils 4 Byte), zum AXI-GPIO-Core gesendet. Da die Kanäle des AXI-GPIOs nur 32 Bit groß sind, werden sie jeweils sofort überschrieben. Dieses Programm wird nur ausgeführt, um das Verhalten des Processing-Systems zu untersuchen und führt nicht zu validen Daten am AXI-Subordinate.

Zu Beginn des Programms wird zunächst der AXI-GPIO-Core initialisiert und die Datenrichtung festgelegt (siehe Abb. 6.9, Zeile 15 ff).

Um die Datenübertragung beim Lesen und Schreiben jeweils im System-Speicher durchzuführen, wird mit der Funktion malloc() Platz im Speicher, in der Größe der zu übertragenden Daten, reserviert. Der Schreib-Buffer wird mit vier Integern gefüllt, die innerhalb eines Bursts übertragen werden sollen (siehe Abb. 6.10).

Vor der Übertragung wird mit der Funktion Xil_DcacheEnable() der Daten-Cache akti-

```
14    //Configuration of AXI-GPIO
15    XGpio output;
16    XGpio input;
17
18    XGpio_Initialize(&output, XPAR_AXI_GPIO_0_DEVICE_ID);
19    XGpio_Initialize(&input, XPAR_AXI_GPIO_0_DEVICE_ID);
20
21    XGpio_SetDataDirection(&output, 1, 0);
22    XGpio_SetDataDirection(&input, 2, 1);
23
```

Abb. 6.9. Konfiguration des AXI-GPIO 0 für eine Burst-Transaktion

```
25    //Memory allocation for write and read data
26    int* write_buffer = (int*)malloc(4*sizeof(int));
27    if (write_buffer == NULL) {
28        printf("Failed to allocate memory.\n");
29        return -1;
30    }
31
32    int* read_buffer = (int*)malloc(4*sizeof(int));
33    if (read_buffer == NULL) {
34        printf("Failed to allocate memory.\n");
35        return -1;
36    }
37
38    //Filling the write_buffer with example data
39    write_buffer[0] = 1;
40    write_buffer[1] = 2;
41    write_buffer[2] = 3;
42    write_buffer[3] = 4;
```

Abb. 6.10. Reservierung von Speicher für Schreib- und Lese-Buffer für eine Burst-Transaktion

viert.[58] Die Datenübertragung geschieht durch das Kopieren von Daten aus einem Buffer in einen anderen mithilfe der Funktion memcpy()[33] (siehe Abb. 6.11). Diese erwartet drei Parameter. Der erste Parameter ist ein Pointer auf die Zieladresse, der zweite Parameter ist ein Pointer auf die Adresse der Quelle und der dritte Parameter gibt die Anzahl der insgesamt zu kopierenden Bytes an. Für das Schreiben von Daten wird der Schreib-Buffer als Quelle und der AXI-GPIO-Core als Ziel angegeben. Für das Lesen von Daten wird der AXI-GPIO-Core als Quelle und der Lese-Buffer als Ziel angegeben. In diesem Programm sollen vier Integer mit jeweils 32 Bit in einem Burst übertragen werden. Daher wird als dritter Parameter 4*4Byte =16 Byte angegeben.

```
45    //Enabling Data Cache
46    Xil_DCacheEnable();
47
48    //Write operation
49    int* src_addr_write = write_buffer;
50    int* dst_addr_write = (int*)(XPAR_AXI_GPIO_0_BASEADDR);
51    memcpy(dst_addr_write, src_addr_write, 16);
52
53    //Read operation
54    /*int* src_addr_read = (int*)(XPAR_AXI_GPIO_0_BASEADDR+ XGPIO_CHAN_OFFSET*1);
55    int* dst_addr_read = read_buffer;
56    memcpy(dst_addr_read, src_addr_read, 16);
```

Abb. 6.11. Datenübertragung mit der Funktion memcpy() für eine Burst-Transaktion

6.6. Programmierung des Processing-Systems mit Linux-Betriebssystem

Durch das Einstecken der zuvor erstellten SD-Karte mit dem Boot-Image und Anschluss des Boards an die Stromversorgung bootet der Mercury XU5 automatisch in Linux. Im Linux-Datei-System liegen die Applikationen und Hardwaresysteme vor. Die AXI-GPIO-Cores befinden sich im Ordner /sys/class/gpio. Dort liegen sie als Gpio-Chips vor. Jeder Gpio-Chip besitzt n Gpio-Signale, wobei n die Breite des AXI-GPIO-Kanals ist. Um die GPIO-Signale beschreiben und auslesen zu können, müssen diese in das Filesystem sysfs exportiert werden. Dabei handelt es sich um ein Filesystem, welches dazu dient, Devices und ihre Attribute in den User-Space zu exportieren. Dazu wird die Nummer n eines Gpio-Signals in die Datei export des Ordners /sys/class/gpio geschrieben. Im gleichen Ordner erscheint daraufhin ein weiterer Ordner für jedes exportierte Gpio-Signal. Dieser Ordner enthält Dateien für die Datenrichtung und den Wert des GPIO-Signals. Diese können beschrieben und ausgelesen werden. So kann mithilfe des sysfs über den User-Space auf die AXI-GPIO-Cores zugegriffen werden.[59–61]
Wie in Abbildung 6.12 zu sehen kann die Zugehörigkeit der einzelnen gpiochips zu den jeweiligen AXI-GPIO-Cores über die Datei label in jedem gpiochip-Ordner ermittelt werden. Es ist außerdem zu erkennen, dass die Nummern der vier gpiochips jeweils um 32 auseinander liegen. Dazwischen liegen jeweils 32 GPIO-Signale, die die einzelnen Bits repräsentieren.

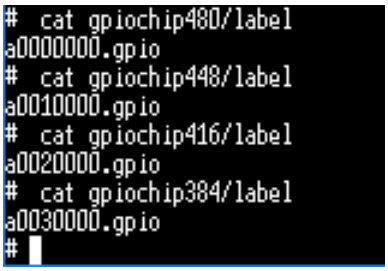

Abb. 6.12. gpiochips und die zugehörigen Basisadressen der AXI-GPIO-Cores

Für das zuvor beschriebene Beschreiben und Auslesen der AXI-GPIO-Cores wird mithilfe der Xilinx Developer Tools eine ausführbare Datei erstellt. Dazu wird in Vitis auf Basis der Hardware-Specification-Datei ein Platform-Projekt erstellt. Dabei wird im Feld Operating System Linux anstatt standalone ausgewählt. Die Erstellung des Application-Projekts erfolgt identisch wie bei der Implementierung ohne Betriebssystem.
Im Programmcode muss zunächst die zuvor beschriebene Exportierung der GPIO-Signale in das sysfs-Filesystem vorgenommen werden (siehe Abb. 6.13, Zeile 22 f.). Daraufhin wird für jedes Signal die Datenrichtung festgelegt (siehe Abb. 6.13, Zeile 25 ff).
Zum Auslesen eines Wertes, der sich aus mehreren Bits, also mehreren GPIO-Signalen, zusammensetzt, wird die Funktion get_gpio_value() (siehe Abb. 6.14) genutzt. Diese bekommt als ersten Parameter die Nummer des Gpio-Chips übergeben, welche gleichzeitig

```c
int main(){

    char system_command_export[50];
    char system_command_direction[50];

    //iteration through 4 AXI-GPIO-Cores = 4*32Bit
    for(int i = 384; i <= 511; i ++){
        //exporting the gpio signals into sysfs
        sprintf(system_command_export, "echo %i > /sys/class/gpio/export",i);
        system(system_command_export);

        if(i < 448){ //AXI-GPIO 2 and 3 are inputs
            sprintf(system_command_direction, "echo in > /sys/class/gpio/gpio%i/direction",i);
        }
        else{ //AXI-GPIO 0 and 1 are outputs
            sprintf(system_command_direction, "echo out > /sys/class/gpio/gpio%i/direction",i);
        }
        system(system_command_direction);
    }

    return 0;
}
```

Abb. 6.13. Konfigurierung der AXI-GPIO-Cores im Linux-Filesystem

die Nummer des ersten GPIO-Signals ist. Die Anzahl der auszulesenden Signale wird als zweiter Parameter übergeben.

Innerhalb einer for-Schleife, die durch die angegebenen GPIO-Signale durchiteriert, wird die Datei value des jeweiligen Signals geöffnet, ausgelesen und der Wert in ein Array gespeichert. So ergibt sich insgesamt aus den einzelnen Einträgen eine Binärzahl, die dem Wert des auszulesenden AXI-Gpio-Kanals entspricht.

```c
static int get_gpio_value_print(int gpio_base, int ngpio){

    char gpio_val_file[128];
    int val_fd=0;
    int gpio_max;
    char val_str[4];
    int value = 0;
    int c;
    int i;
    int j=0;
    int intarray[ngpio];

    gpio_max = gpio_base + ngpio; //all gpio signals

    for(c = gpio_max-1; c >= gpio_base; c--) {
        sprintf(gpio_val_file, "/sys/class/gpio/gpio%d/value",c);
        val_fd=open(gpio_val_file, O_RDWR);

        if (val_fd < 0) {
            fprintf(stderr, "Cannot open GPIO to export %d\n", c);
            return -1;
        }
        read(val_fd, val_str, sizeof(val_str));
        value = atoi(val_str);
        //writing every bit of each signal to array to get whole 32 bit value
        intarray[j] = value;
        j++;

        close(val_fd);
    }

    for(i = 0; i<32;i++){
        printf("%i",intarray[i]);
    }

    return value;
}
```

Abb. 6.14. Funktion get_gpio_value() zum Auslesen aller GPIO-Signale eines GPIO-Chips[62]

6.7. Messung der Funktionalität und Performance des AXI-Interface

Zur Validierung der Funktionalität des AXI-Interface wird der Integrated-Logic-Analyser-IP-Core von Xilinx genutzt. Auf der Strecke des Datentransfers zwischen Processing System und Programable-Logic besteht eine AXI-Kommunikation zwischen dem Processing System und dem SmartConnect und zwischen dem SmartConnect und dem AXI-GPIO-Core. Um die gesamte Datenkommunikation mit dem ILA-Core erfassen zu können, wird jeweils eine dieser Verbindungen an einen Slot des ILA-Cores angeschlossen (siehe Abb. 6.15). Da in dem Design vier AXI-GPIO-Cores enthalten sind, werden fünf ILA-Slots benötigt. Ein Slot für die AXI-Kommunikation zwischen Processing-System und Smart-Connect und vier Slots für die AXI-Kommunikation zwischen dem SmartConnect und dem jeweiligen AXI-GPIO-Core.

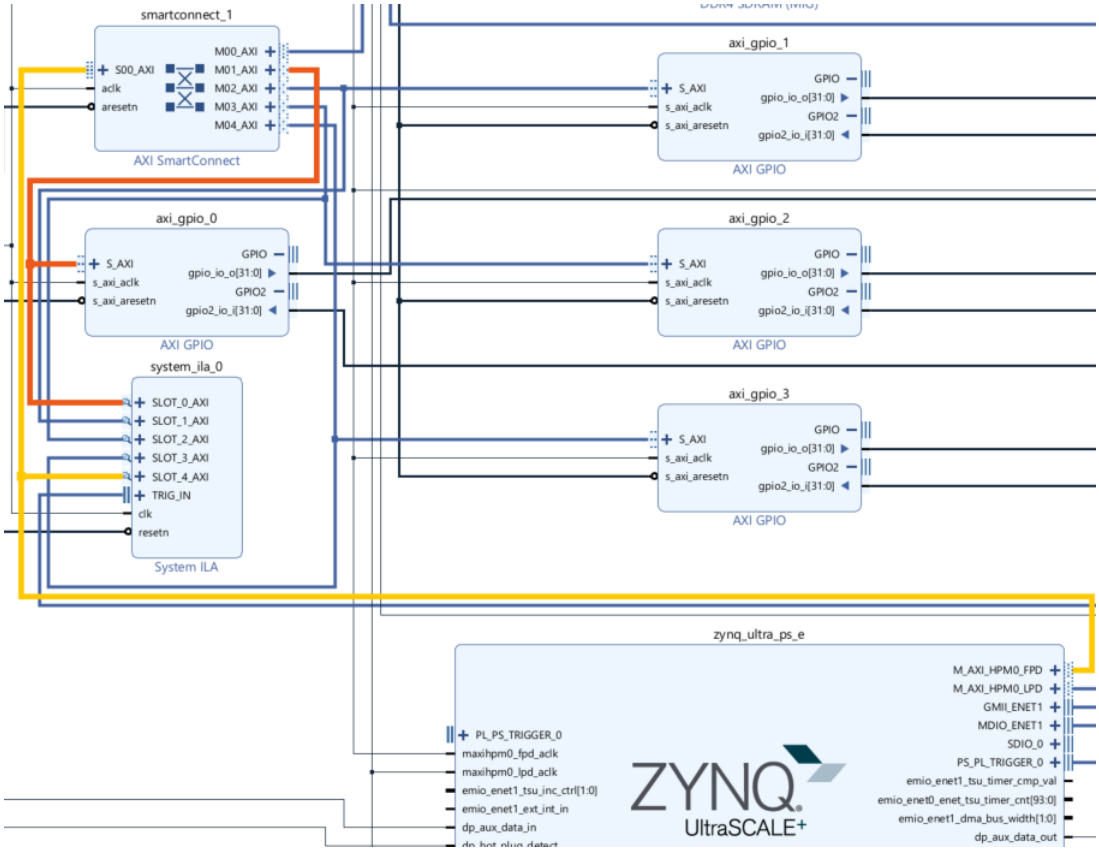

Abb. 6.15. Ausschnitt des Blockdiagramms mit ILA-Slots an den Kommunikationsstrecken zwischen Processing-System und SmartConnect (gelb) und zwischen SmartConnect und AXI-GPIO 0 (orange)

Der PS-UltraScale+-Block muss für die Messung angepasst werden, da der Datenfransfer vom Processing System gestartet wird. Daher wird dem PS-UltraScale+-Block ein PS_PL_Trigger_Output hinzugefügt, welcher mit dem TRIG_IN-Port des ILA-Cores verbunden wird. Im Debug-Setup der Vitis-Applikation muss zudem die Option Cross-Triggering aktiviert werden. Die Messung wird in Vitis gestartet, indem die Applikation im Debug-Modus ausgeführt wird. Daraufhin wird in Vivado der Hardware-Manager geöffnet und eine Verbindung mit dem Target, dem Mercury XU5 geöffnet.
Es öffnet sich ein Hardware-Fenster, in dem der ILA-Core ausgewählt wird. In diesem Fenster befindet sich ein Waveform-Fenster und weitere Fenster zur Einstellung und zum Status der Messung. Im Settings-Fenster wird als Trigger TRIG_IN_ONLY gewählt und der Capture-Modus wird von ALWAYS auf BASIC umgestellt, damit später nicht immer gemessen wird, sondern bestimmte Momente der AXI-Kommunikation aufgefangen werden. Dazu wird im Capture-Setup je nach Messung eine Bedingung für ein bestimmtes AXI-Signal eingestellt. Zur Validierung der Funktionalität des Interfaces wird dort das Signal WDATA genutzt und dessen Übereinstimmung mit dem Wert, der im Schreibbefehl in Vitis festgelegt wurde, überprüft.[52] Im Trigger-Setup wird kein Trigger eingestellt, da dieser softwareseitig passiert. Dazu wird in Vitis an der Stelle, an der die aufzunehmende Transaktion beginnt, ein Breakpoint im Code gesetzt.

6.7.1. Messung der Performance der AXI-Kommunikation

Nach der Überprüfung der Funktionalität soll auch die Erfüllung der Echtzeitanforderung an die Datenkommunikation untersucht werden. Die Zeitmessung der Datenkommunikation erfolgt ebenfalls mit dem ILA-Core. Die Skalierung der X-Achse in dessen Waveform-Fenster entspricht den Taktzyklen des Taktsignals, mit dem der ILA-IP-Core versorgt ist. Somit kann die genaue Zeit einer AXI-Transaktion gemessen werden.

6.8. Messung des zeitlichen Unterschieds beim Hardwarezugriff mit und ohne Linux-Betriebssystem

Die zeitliche Performance, die im vorherigen Kapitel gemessen wurde, betrifft lediglich die Datenkommunikation innerhalb des FPGAs. Ein weiterer Faktor, der die Performance der Datenkommunikation beeinflusst, ist der Zugriff von der Software auf die Hardware. Die Art des Hardwarezugriffs und damit einhergehend die Dauer dieses unterscheidet sich bei der Anwendung mit und ohne Betriebssystem. Um die Differenz des zeitlichen Einflusses des Hardwarezugriffs mit und ohne Betriebssystem zu ermitteln, wird ein im FPGA-Design implementierter Counter genutzt, der im Takt des Logik-Moduls nach oben zählt (siehe Abb. 6.16). In einem Zyklus der Datenkommunikation zwischen PS und PL gibt es genau zwei Hardwarezugriffe. Ein Zugriff geschieht beim Auslesen des neuen

Datensatzes vom FPGA und der zweite Zugriff geschieht beim Schreiben des Datensatzes nach der Datenverarbeitung auf dem Prozessorkern.

```
23  module CommunicationSV(
24
25      input clk,
26
27      output logic [31:0] data_out_1 = 0,
28      output logic [31:0] data_out_2,
29      output logic status_out_1,
30      output logic [31:0] counter = 0,
31
32      input logic [31:0] data_in_1,
33      input logic [31:0] data_in_2,
34      input logic status_in_1
35      );
36
37
38      always @(posedge clk)
39      begin
40
41          if(data_in_1 == 1) begin
42              counter <= counter + 1;
43          end
44
45          if(data_in_1 == 0) begin
46              counter <= counter;
47          end
48
```

Abb. 6.16. Implementierumng des Counters im Logik-Modul

Zunächst sind an der Zählaktion des Counters drei Hardwarezugriffe beteiligt. Ein Schreibbefehl startet den Counter. Dann folgt ein Lesebefehl als erster zu messender Hardwarezugriff und dann folgt wieder ein Schreibbefehl, der den Counter stoppt. Da der Counter erst mit dem vom ersten Schreibbefehl gesendeten Signal gestartet wird, geht die Zeit des ersten Hardwarezugriffs nicht in die Zählung mit ein. Erst wenn der erste Hardwarezugriff vollständig abgeschlossen ist und der geschriebene Wert im Register des AXI-GPIO-Cores gespeichert und von dort aus an das Logik-Modul weitergeleitet wurde, startet der Counter. Während der Counter aktiv ist, geschehen der Lesebefehl und der zweite Schreibbefehl. Dieser sendet den Wert 0 und stoppt damit den Counter, sobald die Daten ins Register des AXI-GPIO-Cores geschrieben wurden. Nachdem der Counter gestoppt wurde, wird dessen Wert mit einem weiteren Lesebefehl ausgelesen und über das Processing-System ausgegeben.

7. Auswertung

Mit den Ergebnissen der zuvor beschriebenen Messungen soll die entwickelte Datenkommunikation nun im Bezug auf die Anforderungen ausgewertet werden. Dazu soll zunächst die Funktionalität des AXI-Interfaces verifiziert und die zeitliche Performance der Datenkommunikation untersucht werden. Daraufhin wird der Einsatz des Linux-Betriebssystems beurteilt.

7.1. Funktionalität des AXI-Interface

Zur Überprüfung der Funktionalität des AXI-Interfaces wird ein Wert vom Processing System zum AXI-GPIO-Core gesendet, dieser wird durch den GPIO-Core des Moduls an das Logik-Modul weitergeleitet und dort mit zwei multipliziert (siehe Abschnitt 5.3). Dieses Register wird vom AXI-Manager wiederum ausgelesen, um das korrekte Abspeichern und Auslesen des Wertes aus dem Register zu verifizieren. Diese beiden AXI-Transaktionen werden im Waveform-Fenster des ILA-Cores dargestellt.

7.1.1. AXI-Kommunikation über das AXI-SmartConnect

Die Transaktionen teilen sich jeweils in zwei Kommunikationsstrecken auf, da das AXI-Manager-Interface des Processing-Systems mit dem AXI-Subordinate-Interface des AXI-GPIO-Cores durch ein AXI-SmartConnect miteinander verbunden ist. In Abbildung 7.1 sind daher zwei Slots zu sehen. Slot_0 misst die AXI-Kommunikation zwischen SmartConnect und AXI-GPIO und Slot_1 misst die AXI-Kommunikation zwischen AXI-Manager und SmartConnect. Die Namen auf der linken Seite des Waveform-Fensters setzen sich aus der Slot-Nummer und dem AXI-Manager-Interface der jeweiligen Kommunikationsstrecke zusammen. Für Slot_0 ist das Manager-Interface das SmartConnect, da dieses die Befehle des Processing-Systems an den AXI-GPIO weiterleitet. Für Slot_1 ist das Manager-Interface das Processing-System.
Auf der linken Seite sind für beide Slots jeweils die fünf AXI-Kanäle zu sehen. Außerdem besitzt jeder Slot zwei weitere Signale, die die gesamte Dauer der Schreib- bzw. Lesetransaktion anzeigen. Wie in Abbildung 7.1 zu erkennen, beginnt die Transaktion von Slot_1 früher und dauert wesentlich länger als die Transaktion von Slot_0, da das Processing System der Ursprung des Schreibbefehls ist. Dieser wird dort gestartet und die finale Schreibantwort des AXI-Subordinate muss dort eingehen, damit die gesamte

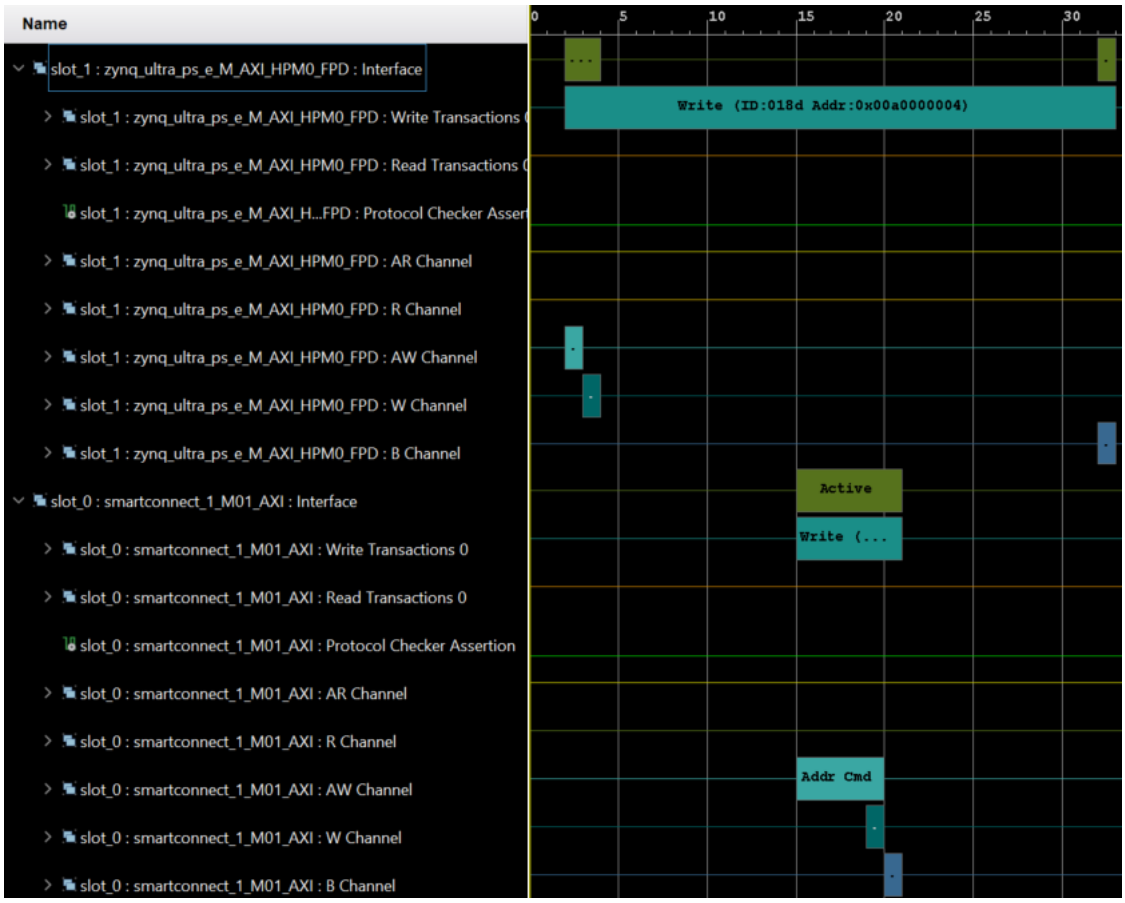

Abb. 7.1. Schreibtransaktion zwischen AXI-Manager, AXI-SmartConnect und AXI-Subordinate

Schreibtransaktion abgeschlossen ist. Der Balken des Signals in Slot_1 zeigt daher die Dauer der gesamten Transaktion an. In der Abbildung kann beobachtet werden, dass die Transaktion zunächst mit dem AW-Kanal startet und kurz darauf ein Transfer im W-Kanal stattfindet. Die nächste Aktion passiert dann im Slot_0, wo das SmartConnect wiederum ein Request über den Adresskanal (AW) sendet und danach ein Transfer auf dem Datenkanal geschieht. Im Slot_0 folgt einen Takt später auch die Schreibantwort. Damit ist die Transaktion im Slot_0 also zwischen SmartConnect und AXI-GPIO abgeschlossen. Als nächstes folgt im Slot_1 die Schreibantwort, die vom SmartConnect an den AXI-Manager weitergeleitet wird. Dieses Signal beendet die Schreibtransaktion.

7.1.2. Adressierung zwischen Processing-System und SmartConnect

Bei einer Transaktion findet zunächst der Transfer auf dem Adressierungskanal AW/AR statt. Dabei wird unter anderem die Adresse des zu adressierenden AXI-Subordinate und des zu beschreibenden Registers übertragen. In Abbildung 7.2 ist der AW-Kanal bei einer Schreibtransaktion zu sehen. Im Signal *Write Transaction* ist die ID und die Adresse der Transaktion zu erkennen. Die Adresse lautet 0x00a0000004. Dabei handelt es sich um die Basisadresse des AXI-GPIO-Cores 0x00a0000000 plus den Adressoffset

des GPIO_TRI_Registers des ersten AXI-GPIO-Signals. An dieser Stelle würde jedoch zunächst die Adresse des ersten GPIO_DATA-Registers erwartet werden. Mit dem Signal AWADDR wird die jeweils anzusprechende Adresse übertragen. Wie in Abbildung 7.2 zu sehen ist, wird zunächst die Adresse des GPIO_TRI-Registers übertragen und danach die Adresse 0x00a0000000, welche die Adresse des ersten GPIO_Data-Registers ist. Daraus kann geschlossen werden, dass der AXI-Manager zunächst die im Programm vollzogene Konfigurierung des ersten GPIO-Kanals an das AXI-GPIO überträgt, bevor eine Adress- bzw. Datenübertragung stattfindet.

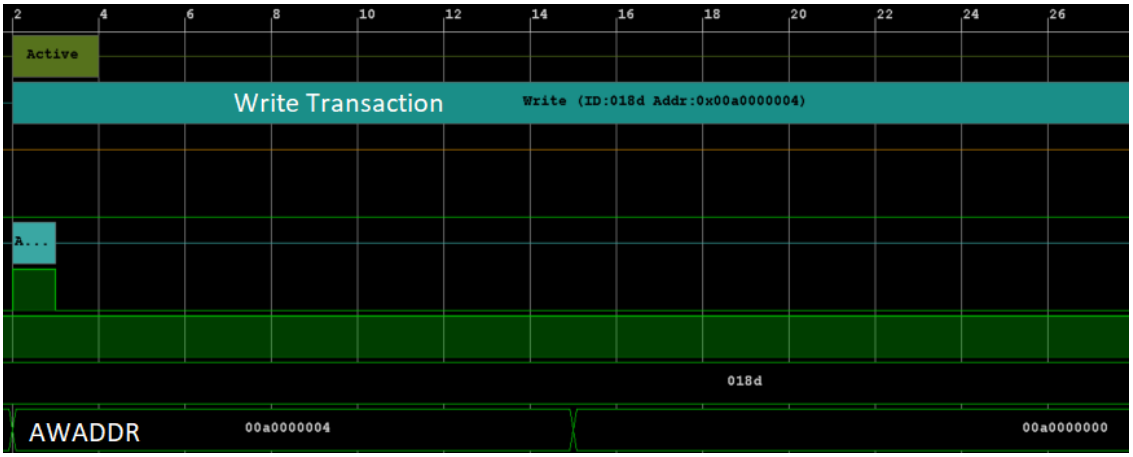

Abb. 7.2. Adressierung des SmartConnects vom Processing-System bei einer Schreibtransaktion mit AWADDR-Signal

In Abbildung 7.3 ist die Lesetransaktion zu erkennen. Das Signal Read Transaction zeigt wieder die Basisadresse 0x00a0000000 des AXI-GPIO-Cores plus einen Adressoffset. Der Adressoffset beträgt 0x08, dabei handelt es sich um den Offset des zweiten GPIO_Data-Registers.

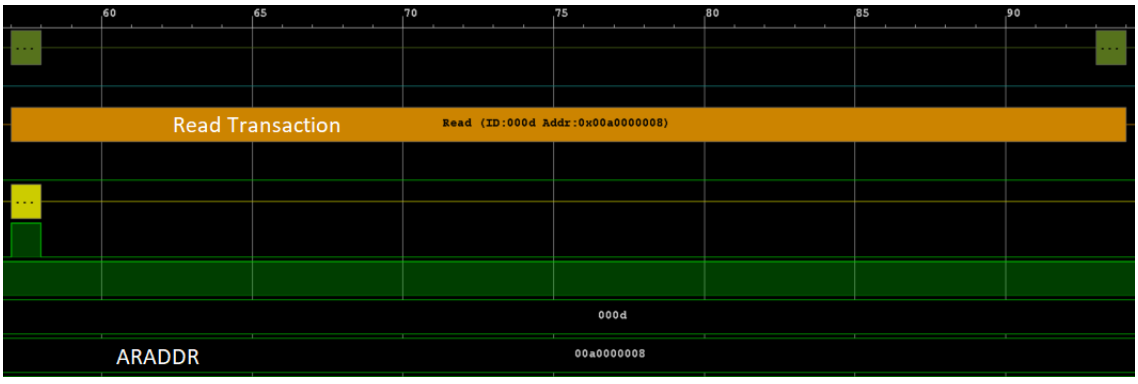

Abb. 7.3. Adressierung des SmartConnects vom Processing-System bei einer Lesetransaktion

Das Processing System soll Daten zum Output des AXI-GPIO-Cores schreiben, welcher mit dem Logik-Modul verbunden ist. Der Input des AXI-GPIO erlangt die Daten vom Logik-Modul, welche dann vom Processing-System ausgelesen werden können. Dazu wurde der erste Kanal des AXI-GPIO-Cores als Output und der zweite Kanal als Input kon-

figuriert. Die korrekte Adressierung wurde somit verifiziert, da der AXI-Manager zum Schreiben von Daten den ersten AXI-GPIO-Kanal mit der Adresse 0x00a0000000 adressiert und zum Lesen den zweiten Kanal mit der Adresse 0x00a0000008.

7.1.3. Adressierung zwischen SmartConnect und AXI-GPIO

Die zuvor beschriebenen Adressen, die der AXI-Manager aussendet, werden zunächst nur an das AXI-SmartConnect gesendet. Dieses führt eine eigene Transaktion mit den AXI-Subordinate aus. Diese Transaktion wird, wie in Abbildung 7.4 zu sehen, ebenfalls mit dem AW-Kanal gestartet. Die übertragene Adresse ist 0x000, diese entspricht dem Adressoffset des ersten Datenkanals des AXI-GPIO .

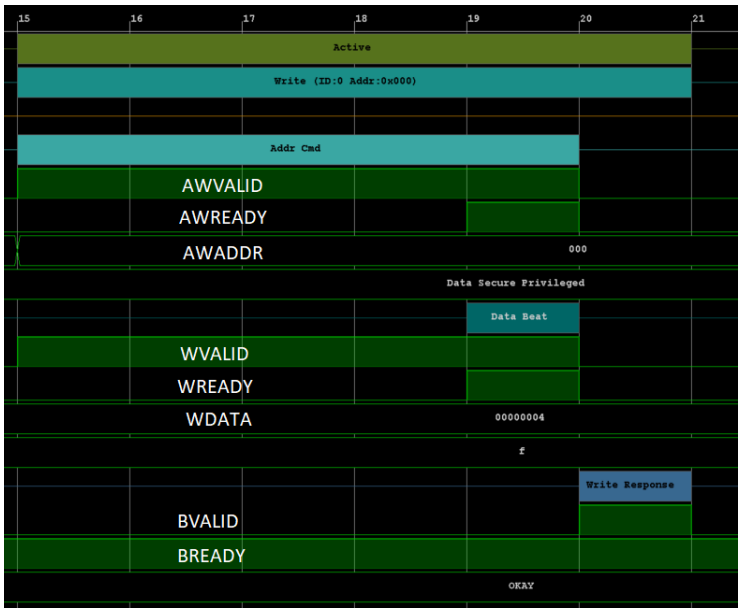

Abb. 7.4. Adressierung des AXI-Subordinate vom SmartConnect bei einer Schreibtransaktion

Die Adresse für die Lesetransaktion beträgt 0x008 (siehe Abb. 7.5), was dem Adressoffset des zweiten Datenkanals des AXI-GPIO entspricht.

Abb. 7.5. Adressierung des AXI-Subordinate vom SmartConnect bei einer Lesetransaktion

Bei der Datenkommunikation zwischen SmartConnect und AXI-Subordinate, welche die Befehle des AXI-Managers an das Subordinate weitergeleitet, wird also die Adresse des Subordinate selbst nicht mehr übertragen, sondern nur noch die Offsets für die zu beschreibenden und auszulesenden Register.

7.1.4. Validierung des AXI-Handshakes

Die Transaktion besteht aus Transfers, die jeweils auf den Kanälen des AXI-Interface stattfinden. Ein Transfer besteht aus einem VALID-READY-Handshake. Diese sind in Abbildung 7.4 jeweils auf dem Adress (AW)-, Daten (W)- und Antwortkanal (B) zu sehen. Zunächst setzt die Quelle, also in diesem Fall das SmartConnect, das AWVALID-Signal auf HIGH. Sobald das Ziel, also der AXI-GPIO, das AWREADY-Signal auch auf eins zieht, findet der Transfer statt und es werden Informationen übertragen. Auf dem Daten-Kanal ist ein ebensolcher Transfer zu erkennen. Das Signal WDATA überträgt dabei den Wert 4.

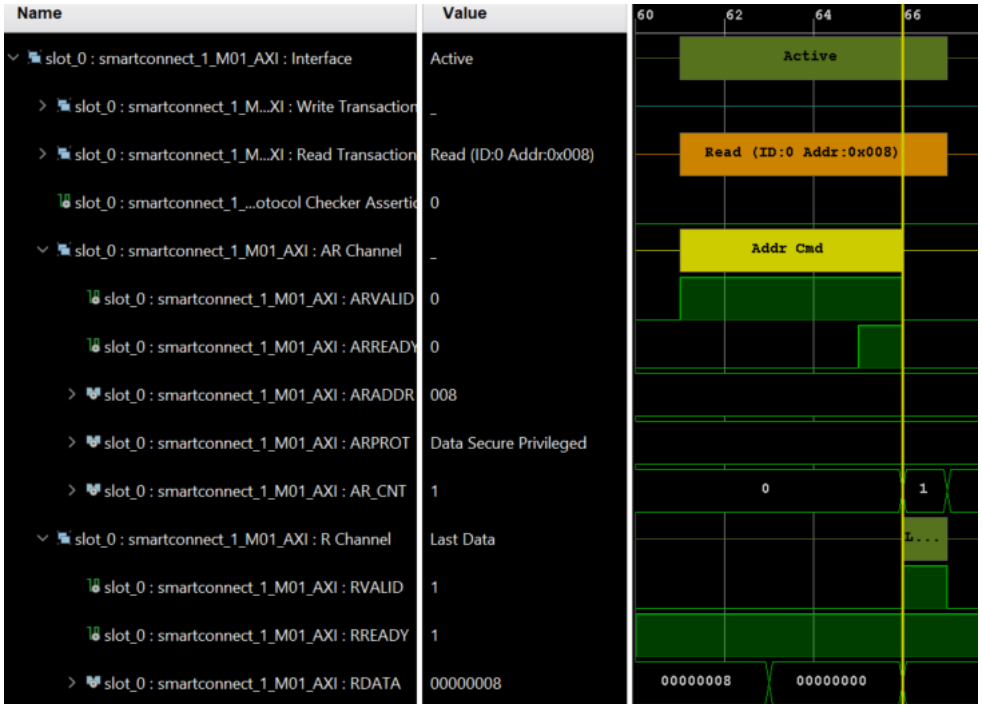

Abb. 7.6. Lesetransaktion mit AXI-Handshakes, ARADDR-Signal und RDATA-Signal zwischen SmartConnect und AXI-Subordinate

In Abbildung 7.6 ist die Lesetransaktion zu sehen. Im Datenkanal (R) des Slot_0 ist das Signal RDATA zu sehen. Dieses besitzt am Anfang den korrekten Wert. Der AXI-Manager, also das SmartConnect, signalisiert die ganze Zeit die Bereitschaft zum Empfang der Daten mit dem RREADY-Signal. Das RDATA-Signal wechselt zwischendurch den Wert, daher zieht das AXI-Subordinate das RVALID-Signal noch nicht nach oben. Sobald wieder die korrekten Daten vorliegen, wird das RVALID-Signal auf eins gesetzt und es kommt zum Datentransfer. Das Processing-System liest den Wert acht aus dem

AXI-GPIO-Register aus. Dies entspricht dem zuvor geschriebenen Wert mit zwei multipliziert.
Dies bestätigt die Funktionalität der Datenkommunikationskette. Ein Wert kann vom Processing-System an das AXI-GPIO gesendet werden, über die GPIO-Register gelangt der Wert in das Logik-Modul, passiert die dort implementierte Logik und wird wieder in die GPIO-Register geschrieben. Das Processing-System kann den Wert von dort wieder auslesen.

7.2. Performance der AXI-Kommunikation

Nachdem die Funktionalität der AXI-Kommunikation gezeigt wurde, soll nun die Performance der aufgestellten Datenkommunikation untersucht werden. Diese ist vor dem Hintergrund der Echtzeitanforderung an die Datenkommunikation im Falle einer teilweisen Auslagerung der Regelung auf den FPGA zu bewerten. Der größte Einflussfaktor beim Einhalten der Echtzeitanforderung ist das zu übertragende Datenvolumen. Abbildung 7.7 zeigt eine vollständige Datenkommunikation zwischen Processing-System und FPGA, also eine Lese- und eine Schreibtransaktion nacheinander.

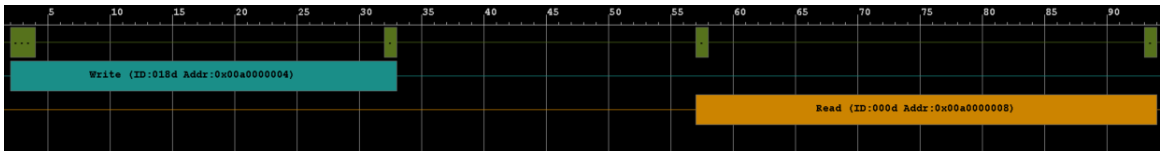

Abb. 7.7. Gesamtlänge einer Schreib- und Lesetransaktion

Die Gesamtdauer für diese beiden Transaktionen beträgt 92 Taktzyklen. Bei einem Takt des AXI-GPIO-Cores von 266,5 MHz, beträgt die Dauer der zwei Transaktionen insgesamt

$$\frac{92}{266{,}5\ \text{MHz}} = 0{,}345\ \mu\text{s}. \tag{7.1}$$

Zur Beobachtung der Datenkommunikation bei der Übertragung eines größeren Datenvolumens wurden mehrere Transaktionen nacheinander ausgeführt, da für die spätere Funktionalität mehrere Daten in einem Regelungszyklus übertragen werden müssen. Dazu wurden vom Processing-System vier Schreibbefehle und vier Lesebefehle zu vier separaten AXI-GPIO-Cores gesendet. Zur Zeitmessung wird an dieser Stelle das ILA-Tool benötigt. Mit diesem soll die Dauer von mehreren Schreib- und Lesetransaktionen hintereinander gemessen werden. Jedoch kann das ILA-Tool mehrere schnell aufeinander folgende Schreibtransaktionen nicht auffangen. Um trotzdem die Zeitmessung durchführen zu können, werden zwischen den einzelnen Schreibbefehlen künstliche Pausen erzeugt. Dies verfälscht jedoch zunächst die gemessene Zeit für die Transaktionen. Um

die Messung trotzdem nutzen zu können, wird im Folgenden der Einfluss der for-Schleife auf die Transaktionsdauer berechnet.

7.2.1. Lösung des Overflow-Problems des ILA-Cores

Der ILA-Core gerät bei mehreren AXI-Schreibtransaktionen in einen Overflow-Status (siehe Abb. 7.8). Dieser tritt auf, sobald mehr als zwei Schreibbefehle direkt hintereinander gesendet werden. Daher wird für die Performance-Messung eine leere for-Schleife hinter jeden zweiten Schreibbefehl eingefügt. Diese zählt bis 50 und führt keine Anweisungen durch. So entsteht eine kurze Pause, damit der ILA-Core alle Schreibtransaktionen auffangen kann. Dies führt jedoch zu einer Verfälschung des zeitlichen Verlaufs der Transaktion.

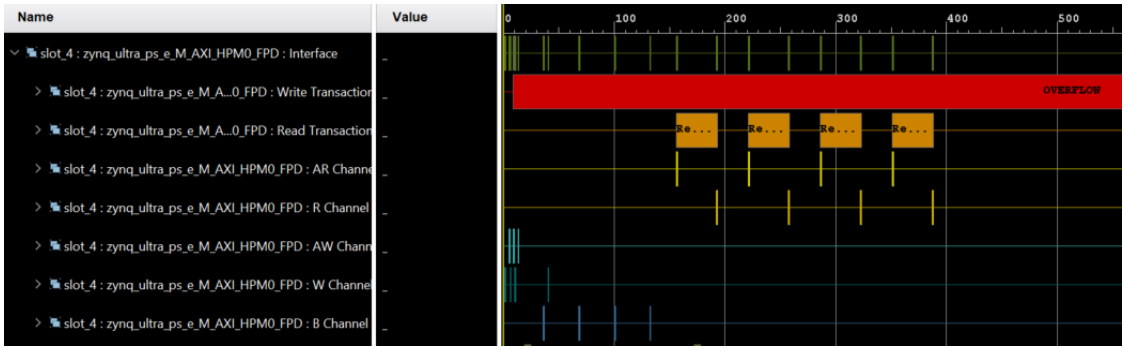

Abb. 7.8. Overflow des ILA-Cores bei zu schnell aufeinanderfolgenden Schreibbefehlen

Um den zeitlichen Einfluss der for-Schleife zu messen, wird diese in das Counter-Programm zur Bemessung des zeitlichen Einflusses des Processing-Systems (siehe Abschnitt 6.8) eingefügt. Bei der Ausführung des Programms ergeben sich 208 Taktzyklen. Davon wird nun die Dauer des Hardwarezugriffs ohne for-Schleife, welche 127 Takte beträgt (siehe Abschnitt 7.3.1), abgezogen. Damit ergeben sich für die for-Schleife $208 - 127 = 81$ Taktzyklen.
Es werden insgesamt 4 Schreibbefehle gesendet. Zwischen dem zweiten und dritten Befehl befindet sich die for-Schleife. Es lässt sich in Abbildung 7.9 erkennen, dass die zweite Schreibtransaktion, hinter welchem die for-Schleife stattfindet, länger dauert als die anderen drei.

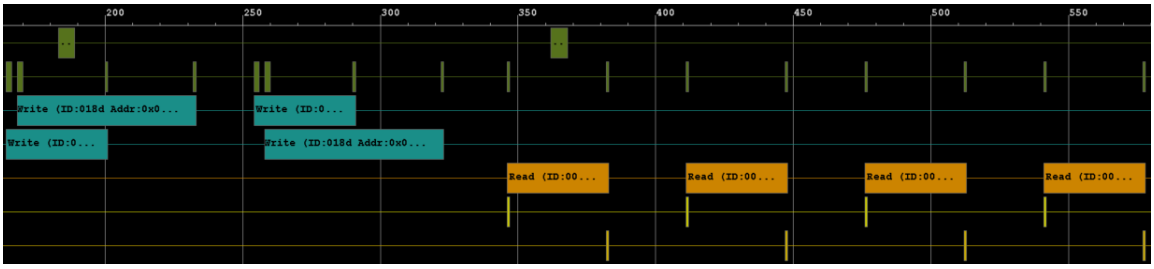

Abb. 7.9. Messung von Schreibbefehlen mit eingebauter for-Schleife

Der zweite Schreibbefehl beginnt 4 Taktzyklen später als der erste. Wenn davon ausgegangen wird, dass dies die Standarddauer ist, die das Processing-System abwartet, um den nächsten Schreibbefehl zu senden, müsste der Beginn des dritten Schreibbefehls vier Takte nach dem Beginn des zweiten Schreibbefehls liegen. Werden nun die 81 Taktzyklen der for-Schleife zum berechneten Start von 94 des dritten Schreibbefehls hinzugerechnet, ergibt sich ein Wert von $94 + 81 = 175$, dies entspricht genau dem Startwert des dritten Schreibbefehls. Die dritte Schreibtransaktion startet 81 Taktzyklen, also genau die Dauer der for-Schleife später. Somit kann davon ausgegangen werden, dass die korrekte Dauer der vier Schreibtransaktionen berechnet werden kann, indem 81 Taktzyklen von der gemessenen Dauer abgezogen werden. Die Dauer der vier Schreibtransaktionen mit der manipulierten ILA-Messung beträgt 158 Takte. Für die vier Schreibtransaktionen ergibt sich nach dem Herausrechnen der Dauer für die for-Schleife eine Dauer von insgesamt $158 - 81 = 77$ Taktzyklen.
Wenn nicht anders angegeben, wurde bei den im Folgenden angegeben Werten die Dauer der for-Schleife bereits abgezogen.

7.2.2. Single-Data-Transfer-Transaktion

Mit der Anpassung des Programmcodes für das Processing-System lässt sich der zeitliche Verlauf mehrfacher Transaktionen hintereinander beobachten. Abbildung 7.9 zeigt vier Schreibtransaktionen und darauffolgend vier Lesetransaktionen zwischen Processing-System und SmartConnect. Die Schreibtransaktionen laufen teilweise parallel ab und dauern insgesamt 77 Taktzyklen. Die Lesetransaktionen laufen nicht parallel ab und dauern daher 232 Taktzyklen. Die Schreibtransaktionen beinhalten das Schreiben der Daten vom Processing-System zum FPGA. Die Lesetransaktionen beinhalten das Schreiben von Daten vom FPGA zum Processing-System. Dies zeigt, dass das Processing-System dazu in der Lage ist, Schreibtransaktionen parallel zu starten, der AXI-GPIO-Core jedoch nicht. Daher dauert das Lesen des Datensatzes länger als das Schreiben. Dabei ist jedoch zu beachten, dass jede gestartete Schreibtransaktion beendet wird, bevor die nächste Transaktion beendet werden kann. Insgesamt dauert das Lesen und Schreiben von vier Datenworten 332 Taktzyklen. Ein Viertel von dieser Zeit sind 83 Taktzyklen. Dieses Ergebnis befindet sich grob in der Größenordnung der Dauer von 92 Taktzyklen für eine Schreib- und Lesetransaktion. Die Dauer der Übertragung ist also annähernd proportional zur Anzahl der zu übertragenden Daten.
Sowohl bei einem Schreib-/Lesezyklus als auch bei der Übertragung der vier Datenworte muss jedoch beachtet werden, dass sich die gemessene Gesamtdauer nicht nur aus der AXI-Kommunikation ergibt. Die Gesamtdauer enthält außerdem die Zeit für den Buszugriff vom Processing-System und weitere Verzögerungen durch die genutzten Geräte wie den PC und das USB-Kabel.
Die 332 Taktzyklen für vier Schreib- und Lesetransaktionen entsprechen bei einer Takt-

frequenz von 266,5 MHz einer Zeit von

$$\frac{332}{266{,}5\ \text{MHz}} = 1{,}246\ \mu\text{s}. \tag{7.2}$$

Die Dauer dieser vier Schreib- und Lesetransaktionen ist damit bereits höher als die in den Anforderungen festgelegten 0,38 µs. Aufgrund des zuvor festgestellten proportionalen Zusammenhangs zwischen Anzahl der zu übertragenden Daten und Dauer der Übertragung kann davon ausgegangen werden, dass bei der Übertragung des in den Anforderungen festgelegten Datenvolumens die Echtzeitanforderung nicht eingehalten wird.

7.2.3. Burst-Transaktion

Neben dem Hardwarezugriff über die XGpio.h-Bibliothek, welche nur einzelne Datentransfers pro Transaktion zulässt, wurde PS-seitig auch eine Übertragung im Burst-Modus implementiert und mit dem ILA-Core ausgemessen. Zunächst wurden vier Integer in einem Burst vom Processing-System versandt. In Abbildung 7.10 ist das Waveform-Fenster des ILA-Cores sehen, in welchem die Schreibtransaktion in Slot_4 dargestellt ist.

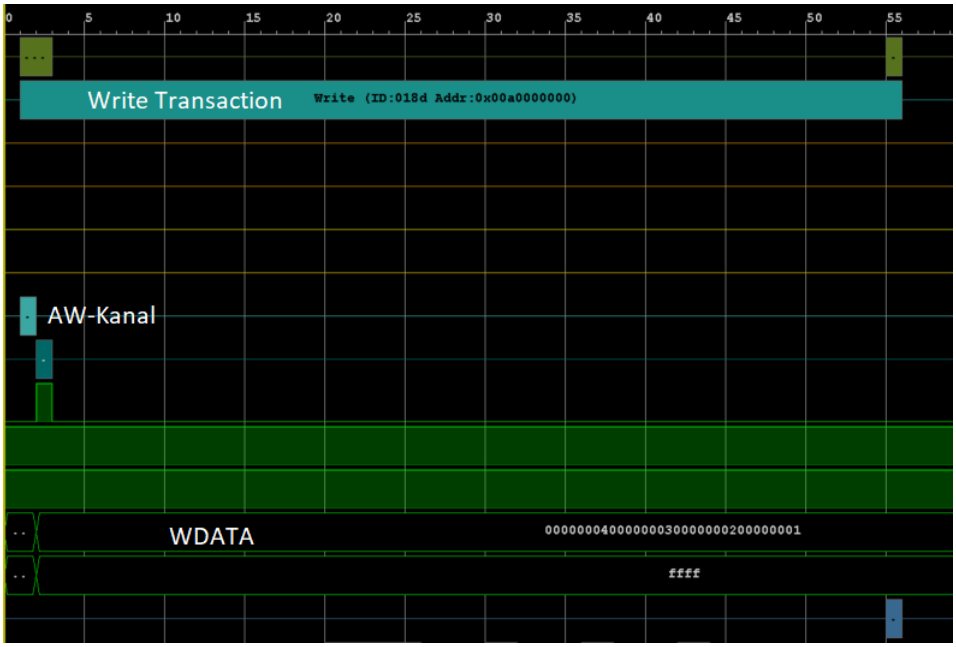

Abb. 7.10. Übertragung von einer Adressphase und vier Datenworten innerhalb einer Burst-Transaktion vom Processing-System

Slot_4 misst die AXI-Kommunikation zwischen Processing-System und SmartConnect. Im Signal *Write Transaction* ist die gesamte Transaktion zu erkennen. Die angezeigte Adresse 0x00a0000000 stimmt mit der Basis-Adresse des angeschriebenen AXI-GPIOs überein. Im Write-Kanal sind auf dem Signal WDATA die zu übertragenden Daten zu

erkennen. Vor der Übertragung der vier Integer gibt es nur einen Adresstransfer. Dies verifiziert, dass es sich bei dieser Übertragung um einen Burst handelt.
Die Gesamtlänge der Transaktion zur Übertragung von vier Datenworten beträgt 55 Taktzyklen. Die Übertragung von vier Datenworten mit der XGpio.h-Bibliothek hat eine Länge von 77 Taktzyklen (siehe Abschnitt 7.2.1). Damit ist die Burst-Übertragung bereits um

$$\frac{77-55}{266{,}5\ \text{MHz}} = 0{,}083\ \mu\text{s} \tag{7.3}$$

schneller. Daher soll außerdem die Übertragung eines großen Datensatzes getestet werden. Das in den Anforderungen festgelegte Datenvolumen (siehe Tabelle 4.1) lässt sich auf 15 mal 32 Bit zusammenfassen. Dies entspräche 60 Byte. Daher wird eine Übertragung von 64 Byte vorgenommen, welche in vier Bursts von jeweils vier 32 Bit-Datenworten verpackt wird. In Abbildung 7.11 ist im oberen Teil die Übertragung von vier Bursts vom Processing-System aus zu sehen. Für jede Transaktion ist jeweils ein Adresstransfer zu erkennen. Nach diesem folgen vier 32-Bit-Datenworte. Im unteren Teil ist die Übertragung zwischen SmartConnect und AXI-GPIO dargestellt. Dort sind für jede dieser Transaktionen vier einzelne Adress- und Datentransfers zu sehen. Dies liegt an der Größenbeschränkung der AXI-GPIO-Signale auf 32 Bit. Das SmartConnect splittet die Datentransfers daher in vier einzelne auf.[33]

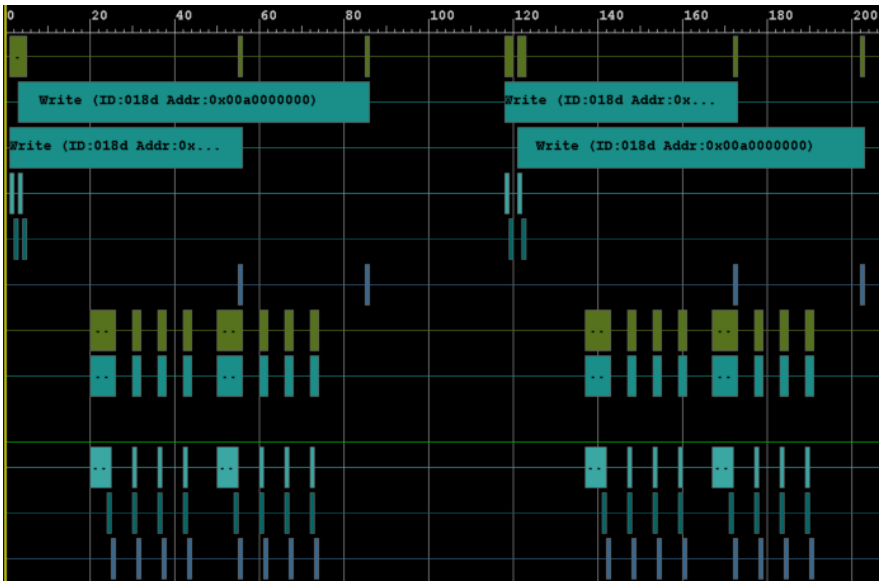

Abb. 7.11. Übertragung zwischen Processing-System und SmartConnect (oben) und SmartConnect und Subordinate (unten) während vier Burst-Transaktionen

Die vier Transaktionen führen im ILA-Core wieder zu einem Overflow. Es wird daher zwischen den zweiten und dritten Burst erneut eine for-Schleife eingefügt. Die Dauer dieser wurde mit dem Counter-Programm vermessen und beträgt 80 Taktzyklen (siehe Abschnitt 7.2.1). Die Gesamtdauer der vier Bursts beträgt 203 Takte (siehe Abb. 7.11).

Nach der Subtraktion der Dauer für die for-Schleife ergibt sich eine Länge von 123 Taktzyklen. Dies entspricht einer Dauer von

$$\frac{123}{266{,}5\ \text{MHz}} = 0{,}462\ \mu\text{s} \tag{7.4}$$

für die Übertragung von 16 32-Bit-Datenworten. Durch die Burst-Fähigkeit des AXI-Interfaces des Processing-Systems kann die Übertragungszeit deutlich gesenkt werden, jedoch kann die Echtzeitanforderung von 0,38 µs knapp nicht eingehalten werden. Dies ist darin begründet, dass die Kanäle des AXI-Subordinate, welches die Daten empfängt, auf 32 Bit beschränkt sind. Außerdem unterstützt es keine Burst-Übertragungen. Das Aufteilen der Bursts in einzelne Transfers mit jeweils zusätzlichem Adresstransfer verlangsamt die Datenkommunikation.

Anhand der somit berechneten Übertragungszeit kann außerdem die Bandbreite der Datenkommunikation berechnet werden. Dabei wird nach dem im Paper *Novel Performance Evaluation Approach of AMBA AXI-Based SoC Design*[63] vorgeschlagenen Ansatz vorgegangen. Das Paper betreibt den selben Messaufbau, ein AXI-Manager sendet einen Burst von vier mal vier Byte zu einem AXI-Subordinate. Zur Evaluation schlägt das Paper die Berechnung der Bandbreite nach folgender Formel vor

$$\text{Gemessene Bandbreite} = \frac{\text{Anzahl der Transaktionen} \cdot \text{Datenbreite}}{\text{Übertragungszeit}}. \tag{7.5}$$

Für die aus Abbildung 7.10 bestimmte Übertragungszeit ergibt sich für einen Burst mit vier Transfers von je vier Byte eine Bandbreite von

$$\text{Gemessene Bandbreite} = \frac{1 \cdot 16\ \text{Byte}}{0{,}206\ \mu\text{s}} = 77{,}670\ \frac{\text{MByte}}{\text{s}}. \tag{7.6}$$

Die ideale Bandbreite lässt sich laut dem Paper folgendermaßen bestimmen

$$\text{Ideale Bandbreite} = \text{Taktfrequenz} \cdot \text{Datenbreite} \tag{7.7}$$

$$= 266{,}5\ \text{MHz} \cdot 16\ \text{Byte} = 4264\ \frac{\text{MByte}}{\text{s}}. \tag{7.8}$$

Aus der Gemessenen und der idealen Bandbreite lässt sich nun die Bandbreitennutzung berechnen.

$$\text{Bandbreitennutzung} = \frac{\text{Gemessene Bandbreite}}{\text{Ideale Bandbreite}} = 1{,}8\% \tag{7.9}$$

Die erhaltene Bandbreitennutzung ist sehr gering, da durch den AXI-GPIO-Core die Übertragungszeit erhöht wird. Die Messungen im Paper ergaben bei einer Taktfrequenz

von 200 MHz eine Bandbreite 3193 $\frac{\text{MByte}}{\text{s}}$. Dies lässt vermuten, dass die Übertragung bei bei Nutzung eines geeigneteren AXI-Subordinate noch um ein Vielfaches schneller sein kann.

7.3. Der Einsatz des Linux-Betriebssystems

Der Entwurf, die Implementierung und die Anwendung der Datenkommunikation wurde jeweils einmal mit und einmal ohne Linux-Betriebssystem durchgeführt. Dabei konnte auch mit Linux-Betriebsystem erfolgreich die Datenkommunikation aufgebaut werden. In Abbildung 7.12 ist die Ausgabe der Linux-Anwendung zum Beschreiben und Auslesen der AXI-GPIO-Cores zu sehen. Auch hier ist eine Verdopplung der Eingangswerte zu sehen, was den vollständigen Ablauf Datenkommunikation bestätigt.

```
# ./Linux_AXI_application.elf
first data transfer:00000000000000000000000000000010
 second data transfer:00000000000000000000000000000100
```

Abb. 7.12. Ausführung der Linux-Applikation und Ausgabe der ausgelesenen Werte auf dem terminal

Die Implementierung der Datenkommunikation wird durch die Xilinx Developer Tools unterstützt, indem für den Hardwarezugriff Pattformdateien erstellt und Bibliotheken zu Verfügung gestellt werden. Das Zufügen von Hardware zum Reference-Design erfordert keinen Mehraufwand. Es muss lediglich die entsprechende Xilinx-Bibliothek eingebunden werden, um auf die neue Hardware zugreifen zu können. Bei der Verwendung des Linux-Betriebssystems gestaltet sich das Hinzufügen von Hardware wesentlich komplexer. Es muss ein neues Boot-Image erstellt und die Device-Tree-Datei umgeschrieben werden. Jedoch stellt Xilinx viele Linux-Driver zur Verfügung. Für die spätere Funktion soll ein OPCUA-Server auf dem Processing-System implementiert werden. Dieser benötigt einen Internetzugang, um Daten mit Client-Geräten auf dem DESY-Gelände auszutauschen. Ein Ethernet-Driver ist im Boot-Image der Enclustra Build Environment enthalten und kann sofort nach dem Boot genutzt werden. Ohne Linux-Betriebssystem müsste dieses Modul zuerst von Hand implementiert und konfiguriert werden. Diese Merkmale spielen jedoch nur bei der Implementierung der Datenkommunikation eine Rolle und beeinflussen die Performance der Datenkommunikation nicht. Daher ist das wichtigste zu untersuchende Merkmal die Dauer des Zugriffs vom Processing-System mit Linux auf die Hardware.

7.3.1. Zeitlicher Einfluss des Hardwarezugriffs

Zur Messung des Anteils des Hardwarezugriffs an der zeitlichen Performance der Datenkommunikation, wurde im Logik-Modul des FPGA-Designs ein Counter implementiert.

Dieser zählt die Taktzyklen die während einer Lese- und einer Schreibtransaktion vergehen. Der Counter wird durch Software-Befehle vom Processing-System aus gestartet und gestoppt. Die Zeit, die das Processing-System benötigt, um auf die Hardware zuzugreifen, um diese Befehle zu senden, ist also in der gemessene Zeit enthalten. Bei der Ausführung des Programms auf einem Processing-System ohne Linux-Betriebssystem gibt der Counter 127 Taktzyklen als Ergebnis aus. Dies entspricht bei einer Taktfrequenz von f=266,5 MHz einer Zeit von 0,477 µs. Die Zeit für eine AXI-Schreib- und eine AXI-Lesetransaktion beträgt 0,345 µs. In dieser Zeit ist jedoch auch zum Teil ein Hardwarezugriff enthalten, der den Lesebefehl sendet. Wird die Zeit für die AXI-Kommunikation von der Zeit der Gesamtkommunikation subtrahiert, ergibt sich eine Zeit von 0,477 µs – 0,345 µs = 0,132 µs. Es lässt sich sagen, dass dies die Zeit ist, die der Hardwarezugriff mindestens benötigt. Bei einer Durchführung des Programms auf dem Processing-System mit Linux-Betriebssystem ergibt sich ein gemitteltes Ergebnis von 730993 Taktzyklen. Dies entspricht bei der erwähnten Taktfrequenz einer Zeit von 2,743 ms. Bei dieser Zeitmessung kann keine Aussage darüber getroffen werden, wie viel Zeit der Hardwarezugriff selbst benötigt, denn die zuvor angegebenen Zeiten enthalten auch die Zeit für die AXI-Kommunikation. Die Zeit der Gesamtkommunikation inklusive Hardwarezugriff von einem Processing-System mit Linux-Betriebssystem ist jedoch um einige Größenordnungen höher als die gemessene Zeit für eine Schreib- und Lesetransaktion, sodass dieser Umstand vernachlässigt werden kann. Somit lässt sich sagen, dass der Hardwarezugriff mit Linux-Betriebssystem im Millisekunden-Bereich liegt und die Echtzeitanforderungen damit um Größenordnungen überschreitet. Die Zeit für den Hardwarezugriff ohne Linux-Betriebssystem befindet sich in derselben Größenordnung wie die Zeit für die AXI-Kommunikation selbst. Sie muss bei der Beurteilung beachtet werden, ist aber kein Ausschlusskriterium für die Anwendung.

8. Fazit

Das Ziel dieser Arbeit war die Entwicklung und Validierung einer Datenkommunikation zwischen Processing-System und FPGA auf einem MPSoC von Xilinx für die Implementierung einer Magnetstromregelung. Dazu wurde mit dem AXI4-Interface eine Kommunikation zwischen den beiden Komponenten erstellt. Für die Interaktion mit dem Logik-Modul, in welchem später die Regelung implementiert werden kann, wurde der Xilinx-IP-Core AXI-GPIO genutzt. Dabei wurde eine Kommunikationsart nur auf dem AXI4-Lite-Interface basierend und eine zweite Kommunikation PS-Seitig mit Burst-Transaktionen aufgebaut.

Die Signale des AXI-Interface wurden mit dem Xilinx-Integrated-Logic-Analyzer aufgenommen. Das Schreiben und Lesen von Daten zum bzw. aus dem korrekten Register des AXI-GPIO konnte damit verifiziert werden. Außerdem konnte die Dauer der AXI-Transaktionen in Taktzyklen gemessen werden. Dabei stellte sich heraus, dass das konstante Senden der Adresse vor dem Senden der Daten in einer Single-Data-Transfer-Transaktion die Übertragung stark verlangsamt. Schon bei der Übertragung von vier 32-Bit-Datenworten, also ca. 26% des angeforderten Datenvolumens wurde die Echtzeitanforderung mit einer Dauer von 1,246 µs überschritten. Daher ist das AXI4-Lite-Interface und damit einhergehend der AXI-GPIO-IP-Core für die angeforderte Datenkommunikation ungeeignet.

Mit dem vollen AXI4-Manager-Interface des Processing-Systems war es jedoch möglich die Performance der Datenkommunikation zu erhöhen. Zunächst kann dieses Transaktionen teilweise parallel durchführen, in dem es nach dem Versenden eines Schreibbefehls bereits die Adresse des nächsten zu beschreibenden Subordinates versendet. Das Processing-System kann außerdem Daten als Bursts versenden. Dabei konnten insgesamt 16 Byte an Datenworten in einer Transaktion versandt werden. Durch das einmalige Senden der Adresse wurde die Datenkommunikation bereits um einiges schneller, als die Kommunikation ohne Bursts. Mit vier Burts aus jeweils 16 Byte konnte das gesamte angeforderte Datenvolumen in 0,466 µs versendet werden. Die Echtzeitanforderung wurde damit jedoch trotzdem knapp überschritten. Außerdem werden die Bursts vor dem Empfang beim AXI-GPIO durch das SmartConnect aufgeteilt, da dieser mit dem AXI4-Lite-Interface keine Bursts empfangen kann und dessen Datenkanäle zu klein sind. Dies deutet darauf hin, dass die Performance der Datenkommunikation noch er-

höht werden könnte, wenn ein AXI-Subordinate verwendet würde, dass das vollständige AXI4-Interface unterstützt.

Die Überprüfung der Echtzeitanforderung erforderte außerdem die Messung des zeitlichen Einflusses des Zugriffs vom Processing-System auf die Hardware. Dieser wurde mit einem in Hardware implementierten Counter gemessen. Es stellte sich heraus, dass eine einzelne Schreib- und Lese-Transaktion mit Zugriff vom Processing-System mit Linux-Betriebssystem 2,743 ms benötigt. Dahingegen benötigt die selbe Transaktion (kein Burst) mit Zugriff vom Processing-System ohne Betriebssystem 0,477 µs. Davon ausgehend kann abgeleitet werden, dass diese Datenkommunikation zur Auslagerung eines Teils der Regelung auf den FPGA nicht geeignet ist. Für die Übertragung der Daten über den OPCUA-Server an externe Rechner stellt die Dauer das Hardwarezugriffs kein Problem dar, da diese nicht unter der strengen Echtzeitanforderung der Regelung liegt.

Im Allgemeinen konnte im Rahmen dieser Arbeit die Plattform Mercury XU5 von Enclustra in Betrieb genommen werden. Die Xilinx Developer Tools boten dabei Unterstützung für den Designfluss vom FPGA-Design bis zur Software-Applikation. Mit dem Tool Enclustra Build Environment konnte erfolgreich ein Boot-Image für die Plattfrom Mercury XU5 generiert werden und an das erstelle FPGA-Design angepasst werden. Die Linuxversion enthält bereits die Konfigurierung für den Ethernet-Port, sodass ein zukünftig auf dem Prozessorkern implementierter OPCUA-Server über Ethernet mit Client-Programmen kommunizieren kann.

Die Xilinx-Developer-Tools bieten die für die Entwicklung nötige Felxibilität, sodass die Anforderung die Sprachen C++ und SystemVerilog zu nutzen, erfüllt werden konnte. Jedoch sind die meisten Xilinx-IP-Cores in VHDL geschrieben und für die Synthese des SystemVerilog-Moduls wird ein Verilog-Wrapper benötigt, sodass das Projekt nun aus drei Sprachen besteht. Daher sollte bei der zukünftigen Nutzung der Xilinx-Umgebung die Anforderung die Sprache SystemVerilog zu nutzen eventuell neu evaluiert werden.

Insgesamt konnte mit den Xilinx-Developer-Tools erfolgreich eine Datenkommunikation aufgestellt werden. Das AXI4-Lite-Interface und der AXI-GPIO-Core eignen sich für die Anfforderungen dieses Projekts nicht. Jedoch wurde mit dem PS-seitigen Burst bereits die Hälfte einer Datenkommunikation aufgebaut, die sich durch die Ergänzung des richtigen AXI-Subordinates zu einer Kommunikation ausbauen lässt, die für die Implementierung der neuen Magnetstromregelung für PETRA IV geeignet ist.

9. Ausblick

Eine Haupterkenntnis dieser Arbeit war die Möglichkeit zur Performance-Erhöhung durch die Nutzung des Burst-Interfaces. Auf der Seite des Processing-System konnte dies bereits umgesetzt werden. Die von dort aus gesandten Bursts wurden jedoch vom Smart-Connect aufgeteilt, da das gewählte AXI-Subordinate, der AXI-GPIO-IP-Core, die Burst-Transktion nicht unterstützt. Die Untersuchung passender AXI-Subordinate-Komponenten ist daher ein interessantes Thema für die weitere Entwicklung der Datenkommunikation.

An dieser Stelle könnte eine solche Komponente selbst geschrieben werden um die zwei größten Nachteile des AXI-GPIO-Cores zu verbessern: die Beschränkung der Datenkanäle auf 32 Bit und die fehlende Unterstützung von Burst-Transaktionen. Die neue Komponente würde die RTL-Logik enthalten und außerdem einen Wrapper um diese, die das vollständige AXI4-Interface implementiert.

Für eine weitere Erhöhung der Performance könnten außerdem Strukturen implementiert werden, die die Schreib- und Lesedaten verwalten. Burst-Transaktionen können erfordern das Managen von großen Datenmengen in kurzer Zeit. Dazu könnte der AXI-DMA-Ansatz, der bereits im Konzept.Kapitel besprochen wurde implementiert werden.Der AXI-DMA könnte dann die Datenübertragung zwischen dem Systemspeicher und dem Logik-Modul übernehmen. Für den Empfang der Datenmengen beim AXI-Subordinate könnte zusätzlich noch ein FiFo implementiert werden, der diese in kurzer Zeit geschriebenen Datenmengen auffängt.

Ein Thema, das in dieser Arbeit bisher nicht behandelt wurde, ist die Korrektheit der übertragenden Daten. Ein Umstand der diese beeinflusst, ist der Zeitpunkt, zu welchem die Daten ausgelesen werden. An dieser Stelle könnten PL-Seitig Interrupts implementiert werden, die signalisieren wann ein vollständiger Datensatz für den aktuellen Regelungszyklus bereit steht. Der Zynq UltraScale enthält dazu 16 Interrupt-Signale von der PL zum PS.[13]

Um die Kommunikationskette weiter auszubauen, kann auf Basis dieser Arbeit außerdem der OPCUA-Server auf dem Processing-System implementiert werden. Im Rahmen dieser Arbeit wurde dort bereits ein Linux-Betriebssystem installiert und ein Zugriff von

diesem auf das AXI-Interface geschaffen. Nun könnte dort der Code für den Server eingebettet und die pro Taktzyklus vom FPGA an das Processing-System übertragenen Daten an Client-Programme auf externen Rechnern übertragen werden.

10. Literaturverzeichnis

[1] DESY: *PETRA IV - Upgrade of PETRA III to the Ultimate 3D X-ray Microscope - Conceptual Design Report*. https://bib-pubdb1.desy.de/record/426140/files/DESY-PETRAIV-Conceptual-Design-Report.pdf

[2] WINFRIED GEHRKE, Marco W.: *Digitaltechnik - Grundlagen, VHDL, FPGAs, Mikrocontroller*. Springer Vieweg, 2022

[3] KESEL, Frank: *FPGA Hardware-Entwurf*. De Gruyter, 2018

[4] AMDXILINX: *Field Programmable Gate Array (FPGA)*. https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html

[5] AMDXILINX: *AMD Adaptive SoCs*. https://www.xilinx.com/products/silicon-devices/soc.html

[6] INTEL: *Intel® FPGAs and SoC FPGAs*. https://www.intel.com/content/www/us/en/products/details/fpga.html

[7] AMDXILINX: *PRODUCT BRIEF Zynq™ UltraScale+™ MPSoC*. https://www.xilinx.com/content/dam/xilinx/support/documents/product-briefs/zynq-ultrascale-plus-product-brief.pdf

[8] ENCLUSTRA: *Mercury XU5 - Xilinx® Zynq® UltraScale+ MPSoC Module*. https://www.enclustra.com/en/products/system-on-chip-modules/mercury-xu5/

[9] ENCLUSTRA: *Mercury+ ST1Base Board for Mercury|Mercury+ FPGA Modules*. https://www.enclustra.com/en/products/base-boards/mercury-st1/

[10] *Block Diagramm Zynq UltraScale+ EG*. https://www.xilinx.com/content/dam/xilinx/imgs/products/zynq/zynq-eg-block.PNG

[11] AMDXILINX: *Zynq® UltraScale+™ MPSoC Data Sheet: Overview (DS891)*. https://docs.xilinx.com/v/u/en-US/ds891-zynq-ultrascale-plus-overview. Version: November 2022

[12] ENCLUSTRA: *Mercury XU5 SoC ModuleUser Manual*. Februar 2021

[13] AMDXILINX: *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*. https://docs.xilinx.com/r/en-US/ug1085-zynq-ultrascale-trm/Introduction-to-the-UltraScale-Architecture

[14] AMDXILINX: *Adaptive SoCs & FPGA Design Tools*. https://www.xilinx.com/products/design-tools.html

[15] AMDXILINX: *Vivado Design Suite User Guide: Design Flows Overview (UG892)*. https://docs.xilinx.com/r/en-US/ug892-vivado-design-flows-overview/Vivado-System-Level-Design-Flows

[16] AMDXILINX: *Intellectual Property*. https://www.xilinx.com/products/intellectual-property.html

[17] AMDXILINX: *Designing IP Subsystems Using IP Integrator*. https://docs.xilinx.com/v/u/2020.1-English/ug994-vivado-ip-subsystems. Version: Juni 2020

[18] AMDXILINX: *Vivado Design Suite User Guide - Synthesis*. https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2022_2/ug901-vivado-synthesis.pdf. Version: November 2022

[19] AMDXILINX: *Vivado Design Suite UserGuideImplementation*. https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2021_2/ug904-vivado-implementation.pdf. Version: November 2021

[20] XILINX: *FPGA Bitstream*. https://www.xilinx.com/htmldocs/xilinx2018_1/SDK_Doc/SDK_concepts/concept_fpgabitstream.html. Version: April 2018

[21] AMDXILINX: *Vitis Unified Software Platform Documentation: Embedded Software Development (UG1400)*. https://docs.xilinx.com/r/en-US/ug1400-vitis-embedded/Getting-Started-with-Vitis

[22] AMDXILINX: *Zynq UltraScale+ MPSoC Software Developer Guide*. https://docs.xilinx.com/r/2021.2-English/ug1137-zynq-ultrascale-mpsoc-swdev/System-Boot-and-Configuration. Version: 2021

[23] *Das U-Boot*. https://u-boot.readthedocs.io/en/latest/index.html#

[24] *Xilinx / u-boot-xlnx*. https://github.com/Xilinx/u-boot-xlnx

[25] AMDXILINX: *Zynq UltraScale+ MPSoC Embedded Design Tutorial - Boot and Configuration*. https://xilinx.github.io/Embedded-Design-Tutorials/docs/2021.1/build/html/docs/Introduction/ZynqMPSoC-EDT/8-boot-and-configuration.html

[26] AMDXILINX: *Zynq-7000 Embedded Design Tutorial*. https://xilinx.github.io/Embedded-Design-Tutorials/docs/2021.1/build/html/docs/Introduction/Zynq7000-EDT/7-linux-booting-debug.html#boot-methods

[27] *Linux and the Devicetree*. https://www.kernel.org/doc/html/latest/devicetree/usage-model.html

[28] *Devicetree Specification*. https://github.com/devicetree-org

[29] ARM: *Learn the architecture - An introduction to AMBA AXI - What is AMBA, and why use it?* https://developer.arm.com/documentation/102202/0300/What-is-AMBA--and-why-use-it-

[30] ARM: *Learn the architecture - An introduction to AMBA AXI - AXI protocol overview*. https://developer.arm.com/documentation/102202/0300/AXI-protocol-overview

[31] *AMBA® AXI Protocol Specification*. https://developer.arm.com/documentation/ihi0022/latest. Version: März 2023

[32] ARM: *Learn the architecture - An introduction to AMBA AXI - Channel transfers and transactions*. https://developer.arm.com/documentation/102202/0300/Channel-transfers-and-transactions

[33] AMDXILINX: *Vivado Design Suite AXI Reference Guide*. https://docs.xilinx.com/v/u/en-US/ug1037-vivado-axi-reference-guide. Version: 2017

[34] XILINX: *Inferring Burst Transfer from/to Global Memory*. https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/xsi1504034302722.html

[35] *LogiCORE IP AXI4-Lite IPIF*. https://docs.xilinx.com/v/u/en-US/axi_lite_ipif_ds765. Version: Januar 2012

[36] AMDXILINX: *LogiCORE IP AXI GPIO (v1.01.b)*. https://docs.xilinx.com/v/u/1.01b-English/ds744_axi_gpio. Version: 2012

[37] BASTOS, Miguel C. ; FERNQVIST, Gunnar ; HUDSON, Gregory ; PETT, John ; CANTONE, Andrea ; POWER, Francis ; SAAB, Alfredo ; HALVARSSON, Björn ; PICKERING, John: High accuracy current measurement in the main power converters of the large hadron collider: tutorial 53. In: *IEEE Instrumentation & Measurement Magazine* 17 (2014), Nr. 1, S. 66–73. http://dx.doi.org/10.1109/MIM.2014.6783001. – DOI 10.1109/MIM.2014.6783001

[38] WDWD: *Abwärtswandler Schaltskizze*. https://commons.wikimedia.org/wiki/File:Buck_converter.svg#/media/Datei:Buck_converter.svg. Version: 2010

[39] INTEL: *Terasic DE10-Nano Kit*. https://www.intel.com/content/www/us/en/developer/topic-technology/edge-5g/hardware/fpga-de10-nano.html

[40] *open62541 Documentation*. https://www.open62541.org/doc/open62541-master.pdf. Version: März 2023

[41] ALTERA: *MAX+PLUS® II*. https://www.intel.com/content/dam/support/us/en/programmable/support-resources/fpga-wiki/asset01/altera-ahdl-language-reference.pdf. Version: 1995

[42] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. In: *IEEE* (2013). https://standards.ieee.org/ieee/1800/4934/

[43] *WinSCP*. https://winscp.net/eng/index.php

[44] AMDXILINX: *AXI DMA LogiCORE IP Product Guide (PG021)*. https://docs.xilinx.com/r/en-US/pg021_axi_dma/AXI-DMA-v7.1-LogiCORE-IP-Product-Guide

[45] AMDXILINX: *Example Design - Zynq-based FFT co-processor using the AXI DMA*. https://support.xilinx.com/s/article/58582?language=en_US. Version: 2017

[46] AMDXILINX: *PetaLinux Tools*. https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html

[47] ENCLUSTRA: *Linux Build Environment*. https://www.enclustra.com/en/products/tools/linux-build-environment/

[48] *Enclustra BSP / bsp-xilinx*. https://github.com/enclustra-bsp/bsp-xilinx

[49] ENCLUSTRA: *Mercury XU5 ST1 Reference Design*. https://github.com/enclustra/Mercury_XU5_ST1_Reference_Design

[50] XILINX: *UltraFast Embedded Design Methodology Guid*. https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/ug1046-ultrafast-design-methodology-guide.pdf#nameddest=ConfigurationAndBootDevices

[51] AMDXILINX: *Integrated Logic Analyzer v2.0 Data Sheet (DS875)*. https://docs.xilinx.com/v/u/en-US/ds875-ila. Version: Juli 2012

[52] XILINX: *Integrated Logic Analyzer v6.2 Product Guide (PG172)*. https://docs.xilinx.com/v/u/en-US/pg172-ila. Version: Oktober 2016

[53] AMDXILIN: *AXI GPIO v2.0 LogiCORE IP Product Guide*. https://docs.xilinx.com/v/u/en-US/pg144-axi-gpio

[54] AMDXILINX: *SmartConnect (PG247)*. https://docs.xilinx.com/r/en-US/pg247-smartconnect

[55] *Buildroot*. https://buildroot.org/

[56] *Specifying GPIO information for devices*. https://www.kernel.org/doc/Documentation/devicetree/bindings/gpio/gpio.txt#:~:text=AGPIObankisaninstanceofa,thesameIPblockafewtimesover.

[57] AMDXILINX: *EDK Concepts, Tools, and Techniques (UG683)*. https://docs.xilinx.com/v/u/en-US/edk_ctt

[58] AMDXILINX: *Standalone Library Documentation: BSP and Libraries Document Collection (UG643)*. https://docs.xilinx.com/r/en-US/oslib_rm/Xil_DCacheEnable?tocId=Zfgz10AprmDJYJndcfDhnw

[59] *The Userspace I/O HOWTO*. https://www.kernel.org/doc/html/v4.12/driver-api/uio-howto.html

[60] PATRICK MOCHEL, Mike M.: *sysfs - The filesystem for exporting kernel objects*. https://www.kernel.org/doc/html/latest/filesystems/sysfs.html. Version: 2011

[61] *GPIO Driver Interface*. https://github.com/Xilinx/linux-xlnx/blob/master/Documentation/driver-api/gpio/driver.rst

[62] *Embedded-Design-Tutorials*. https://github.com/Xilinx/Embedded-Design-Tutorials/blob/master/docs/Introduction/ZynqMPSoC-EDT/ref_files/design1/ps_pl_linux_app.c

[63] TAN, Tuy N. ; DUONG-NGOC, Phap ; PHAM, Thang X. ; LEE, Hanho: Novel Performance Evaluation Approach of AMBA AXI-Based SoC Design. In: *18th International SoC Design Conference (ISOCC)*, 2021, S. 403–404

A. Appendix

Der Anhang zur Arbeit befindet sich auf CD und kann beim Erstgutachter eingesehen werden.

```
gpio@a0000000 {
    #gpio-cells = <0x02>;
    compatible = "xlnx,axi-gpio-2.0\0xlnx,xps-gpio-1.00.a";
    gpio-controller;
    reg = <0x00 0xa0000000 0x00 0x10000>;
    xlnx,all-inputs = <0x00>;
    xlnx,all-inputs-2 = <0x00>;
    xlnx,all-outputs = <0x01>;
    xlnx,all-outputs-2 = <0x00>;
    xlnx,dout-default = <0x00>;
        xlnx,dout-default-2 = <0x00>;
        xlnx,gpio-width = <0x20>;
        xlnx,gpio2-width = <0x20>;
        xlnx,is-dual = <0x00>;
        xlnx,tri-default = <0xffffffff>;
        xlnx,tri-default-2 = <0xffffffff>;
};

gpio@a0010000 {
    #gpio-cells = <0x02>;
    compatible = "xlnx,axi-gpio-2.0\0xlnx,xps-gpio-1.00.a";
    gpio-controller;
    reg = <0x00 0xa0010000 0x00 0x10000>;
    xlnx,all-inputs = <0x00>;
    xlnx,all-inputs-2 = <0x00>;
    xlnx,all-outputs = <0x01>;
    xlnx,all-outputs-2 = <0x00>;
    xlnx,dout-default = <0x00>;
        xlnx,dout-default-2 = <0x00>;
        xlnx,gpio-width = <0x20>;
        xlnx,gpio2-width = <0x20>;
        xlnx,is-dual = <0x00>;
        xlnx,tri-default = <0xffffffff>;
        xlnx,tri-default-2 = <0xffffffff>;
};
```

Abb. A.1. Devicetree-Code - Nodes für vier AXI-GPIO-Cores - Teil 1

```
gpio@a0020000 {
    #gpio-cells = <0x02>;
    compatible = "xlnx,axi-gpio-2.0\0xlnx,xps-gpio-1.00.a";
    gpio-controller;
    reg = <0x00 0xa0020000 0x00 0x10000>;
    xlnx,all-inputs = <0x01>;
    xlnx,all-inputs-2 = <0x00>;
    xlnx,all-outputs = <0x00>;
    xlnx,all-outputs-2 = <0x00>;
    xlnx,dout-default = <0x00>;
        xlnx,dout-default-2 = <0x00>;
        xlnx,gpio-width = <0x20>;
        xlnx,gpio2-width = <0x20>;
        xlnx,is-dual = <0x00>;
        xlnx,tri-default = <0xffffffff>;
        xlnx,tri-default-2 = <0xffffffff>;
};

gpio@a0030000 {
    #gpio-cells = <0x02>;
    compatible = "xlnx,axi-gpio-2.0\0xlnx,xps-gpio-1.00.a";
    gpio-controller;
    reg = <0x00 0xa0030000 0x00 0x10000>;
    xlnx,all-inputs = <0x01>;
    xlnx,all-inputs-2 = <0x00>;
    xlnx,all-outputs = <0x00>;
    xlnx,all-outputs-2 = <0x00>;
    xlnx,dout-default = <0x00>;
        xlnx,dout-default-2 = <0x00>;
        xlnx,gpio-width = <0x20>;
        xlnx,gpio2-width = <0x20>;
        xlnx,is-dual = <0x00>;
        xlnx,tri-default = <0xffffffff>;
        xlnx,tri-default-2 = <0xffffffff>;
};
```

Abb. A.2. Devicetree-Code - Nodes für vier AXI-GPIO-Cores - Teil 2

```c
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpio.h"
#include "xparameters.h"
#include <string.h>


int main()
{
    init_platform();

    XGpio input_0;
    XGpio output_0;

    XGpio input_1;
    XGpio output_1;

    XGpio input_2;
    XGpio output_2;

    XGpio input_3;
    XGpio output_3;

    int axi_0 = 0;
    int axi_1 = 0;
    int axi_2 = 0;
    int axi_3 = 0;


    //AXI-GPIO 0
    XGpio_Initialize(&input_0, XPAR_AXI_GPIO_0_DEVICE_ID);
    XGpio_Initialize(&output_0, XPAR_AXI_GPIO_0_DEVICE_ID);

    XGpio_SetDataDirection(&input_0,2,1);//instance, channel, direction(1:input, 0:output)
    XGpio_SetDataDirection(&output_0,1,0);

    //AXI-GPIO 1
    XGpio_Initialize(&input_1, XPAR_AXI_GPIO_1_DEVICE_ID);
    XGpio_Initialize(&output_1, XPAR_AXI_GPIO_1_DEVICE_ID);

    XGpio_SetDataDirection(&input_1,2,1);
    XGpio_SetDataDirection(&output_1,1,0);

```

Abb. A.3. PS-Code - Konfigurierung, Schreiben und Auslesen von vier AXI-GPIO-Cores - Teil 1

```
45      //AXI-GPIO 2
46      XGpio_Initialize(&input_2, XPAR_AXI_GPIO_2_DEVICE_ID);
47      XGpio_Initialize(&output_2, XPAR_AXI_GPIO_2_DEVICE_ID);
48
49      XGpio_SetDataDirection(&input_2,2,1);
50      XGpio_SetDataDirection(&output_2,1,0);
51
52      //AXI-GPIO 3
53      XGpio_Initialize(&input_3, XPAR_AXI_GPIO_3_DEVICE_ID);
54      XGpio_Initialize(&output_3, XPAR_AXI_GPIO_3_DEVICE_ID);
55
56      XGpio_SetDataDirection(&input_3,2,1);
57      XGpio_SetDataDirection(&output_3,1,0);
58
59
60      //write
61      XGpio_DiscreteWrite(&output_0, 1, 1);
62      XGpio_DiscreteWrite(&output_1, 1, 1);
63      for(int i=0; i<50; i++){}
64      XGpio_DiscreteWrite(&output_2, 1, 1);
65      XGpio_DiscreteWrite(&output_3, 1, 1);
66
67      //read
68      axi_0=XGpio_DiscreteRead(&input_0,2);
69      axi_1=XGpio_DiscreteRead(&input_1,2);
70      axi_2=XGpio_DiscreteRead(&input_2,2);
71      axi_3=XGpio_DiscreteRead(&input_3,2);
72
73      printf("axi_0: %i\n",axi_0);
74      printf("axi_1: %i\n",axi_1);
75      printf("axi_2: %i\n",axi_2);
76      printf("axi_3: %i\n",axi_3);
77
78      cleanup_platform();
79      return 0;
80
81  }
```

Abb. A.4. PS-Code - Konfigurierung, Schreiben und Auslesen von vier AXI-GPIO-Cores - Teil 2

B. Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig angefertigt und keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle Stellen, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem einzelnen Fall unter genauer Angabe der Quelle deutlich als Entlehnung kenntlich gemacht. Dies gilt auch für alle Informationen, die dem Internet oder anderer elektronischer Datensammlungen entnommen wurden.

Hamburg, 19. Mai 2023

Jana Miericke