

## Bachelorarbeit

# Design und Implementierung einer Prozessor/FPGA-Datenschnittstelle auf einem System on Chip

eingereicht von:

**Simon Patrick Liebig**

Matrikelnummer: 3048744

Studiengang: Bachelor Elektro- und Informationstechnik

OTH Regensburg

betreut durch:

Prof. Dipl.-Ing. Dieter Kohlert

OTH Regensburg

Regensburg, den 10. März 2020





OSTBAYERISCHE  
TECHNISCHE HOCHSCHULE  
REGENSBURG

## ERKLÄRUNG ZUR BACHELORARBEIT VON

Name: **Liebig**

Vorname: **Simon Patrick**

Studiengang: **Elektro- und Informationstechnik**

1. Mir ist bekannt, dass dieses Exemplar der Bachelorarbeit als Prüfungsleistung in das Eigentum der Ostbayerischen Technischen Hochschule Regensburg übergeht.
  
2. Ich erkläre hiermit, dass ich diese Bachelorarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Regensburg, den

.....  
**Unterschrift**

*Diese Erklärung ist mit der Bachelorarbeit (eingeheftet) abzugeben.*

Stand: 21.09.2018/Abt. III



---

# Abstract

Diese Bachelorarbeit beschäftigt sich mit dem Design und der Implementierung einer Datentransferschnittstelle zwischen Field Programmable Gate Array (FPGA) und Processing System (PS). Zielsetzung ist, einen performanten Austausch von Daten zwischen den Systemen auf einem System on Chip (SoC) zu ermöglichen. Durch die verbesserte Zusammenarbeit der Systeme können die jeweiligen Vorteile dieser nutzbringender verwendet werden.

Als Basis dient das „MicroZed“ Entwicklungsboard aufgebaut auf der Zynq-Architektur von Xilinx. Auf dem PS wird ein Embedded-Linux verwendet.

Die konkrete Problemstellung hat sich aus einem Projekt von Herrn Professor Kohlert ergeben. Eine sehr langsame Übertragung soll verbessert werden. Der von einer C-Applikation auf dem PS angestoßene Datentransfer zu einem Custom-Block auf der Programmable Logic (PL) basierend auf einem Block-RAM (BRAM) hat nur ca. 4.8 MB/s erreicht.

Um die Übertragungsrate zu verbessern, ist zuerst die Zynq-Architektur auf mögliche Übertragungswege untersucht worden. Für den Datenfluss vom Double Data Rate (DDR)-Random-Access Memory (RAM) zu einem BRAM ist der High Performance (HP)-Port aufgrund der internen Architektur und der möglichen Datenbreite von 64 Bit gut geeignet.

Für das Management der Transfers ist eine Direct Memory Access (DMA)-Lösung gesucht worden. Mit dem ausgewählten Central Direct Memory Access (CDMA)-Block ist ein Hardware-Design in Vivado aufgebaut worden. Dieses ermöglicht Memory-Mapped Datenübertragungen über den Advanced eXtensible Interface (AXI)-Bus. Über die dazu passend implementierte Applikation sind Datenraten von bis zu 305 MB/s möglich geworden.

Eine solche Übertragung ist auf Kanal-Ebene mit Hilfe des Integrated Logic Analyzers (ILAs) genauer analysiert worden. Dabei werden die zuvor präsentierten Grundlagen über den AXI-Bus veranschaulicht und die Auswirkung der Design Parameter auf die Datenrate beschrieben.

Neben der Memory-Mapped-Variante wird auch die Verwendung von Stream-Daten diskutiert. Durch Daten-Streams kann man auf BRAMs in der PL verzichten und Operationen praktisch hintereinander verkettet durchführen. Für die Verwaltung wird dann der AXI-DMA-Block eingesetzt.

Während der Bearbeitung des Themas hat sich ein Zukunftsziel gezeigt, bei dem eine Schnittstelle mit einfacher Application Programming Interface (API) einen DMA-Linux-Treiber versteckt, sodass Datentransfers ohne tiefer reichendes Wissen ermöglicht werden.

Um diesem Ziel näherzukommen, werden in dieser Bachelorarbeit viele explizite Workflowbeschreibungen und Grundlagen (besonders für PetaLinux) dargestellt. Außerdem wird die Verwaltung von DMA-Prozessen über Linux-Treiber vorgestellt. Damit wird es möglich, einen soliden Stand zu reproduzieren, von dem aus weiterführende Projekte gestartet werden können. Ein Ausblick auf solche vertiefenden Themen runden die Bachelorarbeit ab.



# Inhaltsverzeichnis

<b>Erklärung</b>	<b>I</b>
<b>Abstract</b>	<b>II</b>
<b>Inhaltsverzeichnis</b>	<b>III</b>
<b>Abkürzungsverzeichnis</b>	<b>VI</b>
<b>Abbildungsverzeichnis</b>	<b>VII</b>
<b>1. Einleitung</b>	<b>1</b>
1.1. Motivation . . . . .	1
1.2. Aufgabenstellung . . . . .	3
1.3. Überblick . . . . .	3
<b>2. Voraussetzungen</b>	<b>4</b>
2.1. Programme . . . . .	4
2.1.1. Betriebssystem . . . . .	4
2.1.2. Vivado . . . . .	4
2.1.3. Xilinx SDK . . . . .	4
2.1.4. PetaLinux Tools . . . . .	4
2.1.5. Verwendung/Setup . . . . .	4
2.2. Hardware . . . . .	5
<b>3. Grundlagen</b>	<b>6</b>
3.1. AXI Bus . . . . .	6
3.1.1. AXI Transfer . . . . .	6
3.1.2. Burst Adressierung . . . . .	6
3.1.3. AXI-Lite . . . . .	8
3.1.4. AXI-Stream . . . . .	8
3.2. Embedded Linux Grundlagen . . . . .	9
3.2.1. Linux . . . . .	9
3.2.2. Boot Prozess . . . . .	9
3.2.3. Root File System . . . . .	10
3.2.4. Device Drivers . . . . .	10
3.2.5. Device Tree . . . . .	10
3.3. Direct Memory Access . . . . .	11
3.3.1. Scatter Gather Mode . . . . .	11

<b>4. Datentransfer Schnittstelle</b>	<b>12</b>
4.1. Diskussion der Datentransfer Schnittstellen . . . . .	12
4.1.1. ACP . . . . .	13
4.1.2. GP-Interfaces . . . . .	13
4.1.3. HP-Interfaces . . . . .	14
4.2. Vorstellung der Datentransfer Management Einheiten . . . . .	14
4.2.1. DMAC (PS-DMA) . . . . .	14
4.2.2. AXI-Datamover . . . . .	14
4.2.3. AXI-Streaming-FIFO . . . . .	15
4.2.4. AXI-DMA . . . . .	15
4.2.5. AXI-VDMA . . . . .	17
4.2.6. AXI-CDMA . . . . .	17
4.3. Auswahl und Begründung . . . . .	17
<b>5. Erstellung des Ausgangssystems</b>	<b>18</b>
5.1. Grundsystem Petalinux . . . . .	18
5.2. SDK Workflow . . . . .	22
<b>6. Realisierung</b>	<b>24</b>
6.1. Hardwaredesign . . . . .	24
6.1.1. Funktionsweise fir_memo_top_v1_0 . . . . .	26
6.1.2. Erstellung des Designs . . . . .	27
6.2. Linux-App für Memory-Mapped PS-PL Zugriffe . . . . .	32
6.2.1. mmap Befehl . . . . .	32
6.2.2. CDMA Programmier-Sequenz . . . . .	33
6.2.3. Reserved-Memory . . . . .	34
6.2.4. Einbau ins PetaLinux . . . . .	35
6.3. Linux Treiber . . . . .	36
6.3.1. Treiberentwicklung . . . . .	36
6.3.2. Überblick von DMA-Treiberlösungen . . . . .	36
6.3.3. Treiber SetUp in Petalinux - Am Beispiel bperez77 axidma-driver . . . . .	37
6.3.4. Verwendung von Treibern im Embedded-Linux . . . . .	41
6.3.5. Beispiel Anwendungen . . . . .	43
6.3.6. User-Space minimal Beispiel - Lauri . . . . .	44

<b>7. Analyse der Übertragung</b>	<b>46</b>
7.1. Controle-Phase . . . . .	46
7.2. Datenübertragungs-Phase . . . . .	48
<b>8. Analyse der Performance</b>	<b>50</b>
8.1. Was beeinflusst die Performance . . . . .	50
8.1.1. Datenbreite . . . . .	50
8.1.2. Burst Size . . . . .	51
8.2. Übertragungsrate . . . . .	51
<b>9. Fazit</b>	<b>53</b>
<b>10. Ausblick</b>	<b>54</b>
<b>Literaturverzeichnis</b>	<b>VIII</b>
<b>Anhang</b>	<b>XII</b>
<b>A. Beiliegender Datenträger</b>	<b>XII</b>
<b>B. CDMA Design</b>	<b>XIII</b>
<b>C. C-Code</b>	<b>XIV</b>
<b>D. AXI-DMA Design</b>	<b>XXI</b>

# Abkürzungsverzeichnis

ACP	Accelerator Coherency Port	HP	High Performance
ADC	Analog to Digital Converter	HW	Hardware
API	Application Programming Interface	ILA	Integrated Logic Analyzer
ASIC	Application Specific Integrated Circuit	IP	Intellectual Property
AXI	Advanced eXtensible Interface	JTAG	Joint Test Action Group
BRAM	Block-RAM	LSB	Least significant Bit
BSP	Board Support Package	LTS	Long Term Support
CDMA	Central Direct Memory Access	MM2S	Memory Mapped to Stream
CMA	Contiguous Memory Allocator	PCIe	Peripheral Component Interconnect Express
DAC	Digital to Analog Converter	PL	Programmable Logic
DDR	Double Data Rate	PS	Processing System
DMA	Direct Memory Access	RAM	Random-Access Memory
DMAC	Direct Memory Access Controller	S2MM	Stream to Memory Mapped
DSP	Digital Signal Processor	SDK	Software Design Kit
DTB	Device Tree Blob	SG	Scatter/Gather
DTC	Device Tree Compiler	SoC	System on Chip
DTS	Device Tree Source	SPI	Serial Peripheral Interface
FFT	Fast Fourier Transformation	SSH	Secure Shell
FIFO	First In First Out	SW	Software
FPGA	Field Programmable Gate Array	TCL	Tool command language
FS	File System	UART	Universal Asynchronous Receiver Transmitter
FSBL	First-Stage Boot Loader	VDMA	Video Direct Memory Access
FTP	File Transfer Protocol	VHDL	Very High Speed Integrated Circuit Hardware Description Language
GP	General Purpose		
GPIO	General Purpose Input Output		
HDF	Hardware Description File		

# Abbildungsverzeichnis

1.1. HW/SW-Co-Design . . . . .	1
2.1. Hardware Aufbau . . . . .	5
3.1. AXI read transaction . . . . .	7
3.2. AXI write transaction . . . . .	7
3.3. AXI-Stream von zwei Paketen . . . . .	8
3.4. MicroZed Boot-Mode Jumperstellungen . . . . .	9
4.1. Topologie . . . . .	12
4.2. Interconnect Block Diagram . . . . .	13
4.3. AXI-HP Path to DDR . . . . .	14
4.4. AXI Stream FIFO IP-Block . . . . .	15
4.5. AXI Stream Data FIFO IP-Block . . . . .	15
4.6. AXI-DMA IP-Block Datenfluss . . . . .	16
4.7. Example Design - Zynq-based FFT co-processor using the AXI DMA . . . . .	16
4.8. AXI-CDMA IP-Block Datenfluss . . . . .	17
5.1. Netzwerkeinstellung am PC . . . . .	21
5.2. Neue Linux Application im SDK . . . . .	22
5.3. Debugg Configuration . . . . .	23
6.1. Hardware-Design . . . . .	25
6.2. fir_memo_top Design . . . . .	26
6.3. Ausgabe des vorbelegten Speichers . . . . .	27
6.4. Re-customize IP - ZYNQ PS - Preset . . . . .	28
6.5. Address Editor-CDMA . . . . .	30
6.6. Signalformen: (a) 1kHz Dreieck, (b) 1kHz Sinus . . . . .	33
6.7. DMA-Driver-Stack . . . . .	37
7.1. Schreiben auf BRAM - AXI-Übertragung Übersicht . . . . .	46
7.2. Schreiben auf BRAM - Controle-Phase . . . . .	47
7.3. Schreiben auf BRAM - Controle-Phase Read Statusregister . . . . .	47
7.4. Schreiben auf BRAM - CDMA-Befehl . . . . .	48
7.5. Schreiben auf BRAM - Daten . . . . .	49
8.1. Datenbreite im CDMA-Design . . . . .	50
8.2. Schreiben auf BRAM AXI-Übertragung Übersicht . . . . .	51



# 1. Einleitung

## 1.1. Motivation

Früher erfolgte der Entwurf von Systemen unterteilt in Software- und Hardware-Abschnitte, die unabhängig voneinander entwickelt werden konnten. Die heutigen Elektroniksysteme werden jedoch immer komplexer und auch der Stromverbrauch spielt zunehmend eine wichtigere Rolle. Durch ein neues Konzept, indem Hardware und Software parallel - „Hand in Hand“ - entwickelt werden, wird es möglich, Leistungsaufnahme-Optimierungen oder auch Performance-Verbesserungen auf Systemebene auszuführen. Eine neue Methode ist das „Hardware-Software-Co-Design“, welche besonders auf System on Chips (SoCs) abzielt. SoCs integrieren Allzweck-Mikroprozessoren, Digital Signal Processor (DSP)-Strukturen, Field Programmable Gate Arrays (FPGAs), Application Specific Integrated Circuit (ASIC)-Cores, Memory-Block-Peripheriegeräte und deren Verbindungsbusse auf einem Chip. Die Integration durch SoCs ermöglicht die Nutzung der jeweiligen Vorteile der einzelnen Komponenten. Beispielsweise können Aufgaben speziell auf Performance oder bessere Leistungsaufnahme auf einer Hardware, wie einem FPGA, optimiert werden, während andere Aufgaben die Wartbarkeit und Änderungsflexibilität von Software nutzen. [15]

Außerdem gibt es Bedarf an Software-Beschleunigern, welche genutzt werden, um Funktionen von normalen Prozessorsystemen auszulagern. Diese Accelerator sind dann in Hardware speziell auf ihren Anwendungsfall ausgelegt. Mögliche Einsatzgebiete sind die „Big Data“ verwandten Themenfelder wie Machine-learning und künstliche Intelligenz in Branchen wie Cybersecurity, Genforschung, Autonomes Fahren und bei Finanzanalysen. [11]

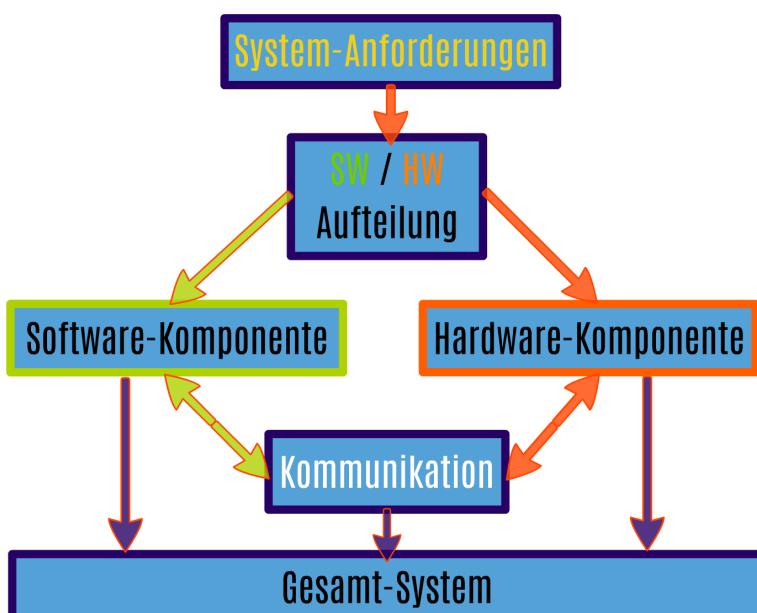


Abbildung 1.1.: Hardware-Software-Co-Design

Wie man in Abbildung 1.1 sieht, ist die Kommunikation ein zentrales Bindeglied im Gesamtdesign. Die Zusammenarbeit verschiedener Komponenten kann folglich nur mit einer funktionierenden Kommunikation gelingen. Auch bei den Hardware-Beschleunigern darf die Datenübertragung nicht den Flaschenhals/die Engstelle bilden.

Die Herausforderung, das Bindeglied „Kommunikation“ für die richtige Ausnutzung von SoCs besser zugänglich zu machen und zu verstehen, motiviert die Bearbeitung der Thematik.

So beschäftigt sich die vorliegende Arbeit mit der Prozessor-/FPGA-Kommunikation. Dabei wird festgestellt, dass an einer performanten Datenübertragung noch viel mehr hängt (z.B. Memory Management) und die Umsetzung deshalb nicht trivial ist. Zudem werden Potenzial und Möglichkeiten identifiziert, mit denen zum Beispiel Cache-Coherente Datenaustausches vorgenommen werden können und dabei noch weit höhere Übertragungsraten und mehr Abstraktion möglich gemacht werden können, als im Rahmen der Arbeit erreicht wird.

Die Arbeit hat den Anspruch einen guten Einblick in die Thematik zu geben. Außerdem wird bewusst, dass die Datenübertragung zwischen Systemen ein Thema für sich darstellt. Ziel muss es sein, dass die Technik eines Tages beim Komponenten-Entwurf „einfach nur“ verwendet werden kann. Um das Hardware-Software-Co-Design zu erleichtern, sollte eine Lösung entstehen, die den einfachen Einsatz ermöglicht und die Technik hinter einer dokumentierten Schnittstelle versteckt.

Das Ergebnis der Arbeit zeigt einige Methoden, Bausteine und Lösungen auf und setzt einen Weg passend zur Aufgabenstellung um.

## 1.2. Aufgabenstellung

Ausgangspunkt zur Entwicklung des Themas für diese Arbeit war eine sehr langsame Datenübertragung in einem Projekt von Professor Kohlert.

Nach einigen Analysen während der Durchführung verfeinerte sich die Aufgabe dann zu einer memory-mapped Kommunikation zwischen Double Data Rate (DDR)-Speicher und einem Block-RAM (BRAM). Die Rahmenbedingungen ergaben sich aus dem verwendeten Entwickler-Board „MicroZed“, was auf einem „Xilinx Zynq®-7000 All Programmable SoC“ basiert und einem Ziel-Hardware-Block „Fir\_Memo\_Top\_v1\_0“ - entwickelt von Professor Kohlert.

Hierbei war mit dem „Fir\_Memo\_Top\_v1\_0“ und der vorhandenen Applikation eine gute Beispieldatenwendung gegeben, mit der Übertragungen einfach getestet werden konnten.

Die Umsetzung sollte auf einem Embedded-Linux erfolgen.

Außerdem gehörte zur Aufgabe das Bestehende zu analysieren, gegebenenfalls zu überarbeiten und mit einer neuen Technik zu beschleunigen.

Notwendige Basisarbeit war die sehr komplexe Toolchain für die SoC-Entwicklung zu verstehen, aufzusetzen und nutzerfreundlich ausführbar zu machen.

Zusammenfassend befasst sich die Arbeit mit der Realisierung der Prozessor-/FPGA-Kommunikation auf dem „MicroZed-Board“.

## 1.3. Überblick

Zu Beginn der Arbeit erfolgt eine Beschreibung der Toolchain und des Systemaufbaus - beginnend nach der Installation aller Programme (Kapitel 2). Darauf folgt die Beschreibung der Grundlagen dieser Arbeit (Kapitel 3). In Kapitel 4 wird diskutiert, wie die Datenschnittstelle umgesetzt werden soll, bevor sie dann aufbauend auf ein erstelltes Ausgangssystem (Kapitel 5) in Kapitel 6 realisiert wird. Abschließend wird das Ergebnis noch analysiert und bewertet (Kapitel 7, 8).

## 2. Voraussetzungen

Bevor an der eigentlichen Aufgabe gearbeitet werden kann, muss man einige Vorarbeiten erledigen. Hierzu gehören vor allem das Aufsetzen der Toolchain mit allen wichtigen Programmen und einige Grundlagen zu den verwendeten Systemen sowie der Hardwareaufbau zum Testen.

### 2.1. Programme

Im Folgenden werden die wichtigsten Tools für den Workflow aufgeführt und kurz beschrieben.

#### 2.1.1. Betriebssystem

Es ist sinnvoll alle Arbeiten auf einem System auszuführen, da der Workflow so nicht durch einen Systemwechsel unterbrochen werden muss. Die PetaLinux Tools unterstützen nur Linux Hostsysteme. Somit bietet sich Ubuntu in einer Long Term Support (LTS) Version an. Als Basissystem dient deshalb ein Ubuntu 18.04.

#### 2.1.2. Vivado

Vivado wird benötigt, um einerseits Hardware im FPGA Teil des Zynq Chips zu implementieren, jedoch auch um die Prozessor- und Peripheriemodule zu aktivieren und zu konfigurieren. Als Endprodukt erstellt Vivado aus dem Bitstream für das FPGA-Modul und den unterschiedlichen Konfigurationen ein Hardware Description File (HDF).

#### 2.1.3. Xilinx SDK

Das Xilinx Software Design Kit (SDK) basiert auf Eclipse und dient zur Softwareentwicklung für eingebettete Systeme.

#### 2.1.4. PetaLinux Tools

Die PetaLinux Tools stellen ein Embedded-Linux Entwicklungskit - speziell für SoCs mit FPGA - zur Verfügung. Sie vereinen mehrere Programme in einem gemeinsamen Interface. Damit wird dem Anwender die Erstellung seines eigenen Linux-Betriebssystems erleichtert und ein linearer Entwicklungsablauf ermöglicht. [36]

#### 2.1.5. Verwendung/Setup

Im Folgenden wird davon ausgegangen, dass die Programme alle richtig in der Version 2017.4 installiert sind und dass die Tools durch die jeweilige Ausführung des „settings.sh“ Skripts initialisiert sind. Die dafür notwendigen source Befehle wurden zur Automatisierung in den Shellstart des Ubuntu Operating Systems eingebaut (.bashrc unter /home). Zusätzlich muss der Linux Command Interpreter einmalig von „dash“ auf „bash“ umkonfiguriert werden. [36]

```
1 $ sudo dpkg-reconfigure dash
```

## 2.2. Hardware

Es wird ausschließlich das MicroZed Evaluation Board verwendet. Dabei handelt es sich um ein kostengünstiges Board basierend auf Xilinxs Zynq®-7000 All Programmable SoC. Die Architektur besteht aus zwei ARM Cortex-A9 Prozessoren auf der Processing System (PS)-Seite und einem sehr energieeffizienten FPGA auf der Programmable Logic (PL)-Seite. [4]

Das MicroZed Board wird auf die MicroZed I/O Carrier Card aufgesteckt verwendet. Es ist über den JTAG-Programmer und die USB-UART sowie über den Ethernetport mit dem Basisrechner verbunden. Für spätere Testzwecke ist ein Pmod DA3 16-bit Digital-to-Analog Konverter (AD5541A inside) an der Carrier Card JA 7-12 angeschlossen. In Abbildung 2.1 wird der schematische Aufbau dargestellt.

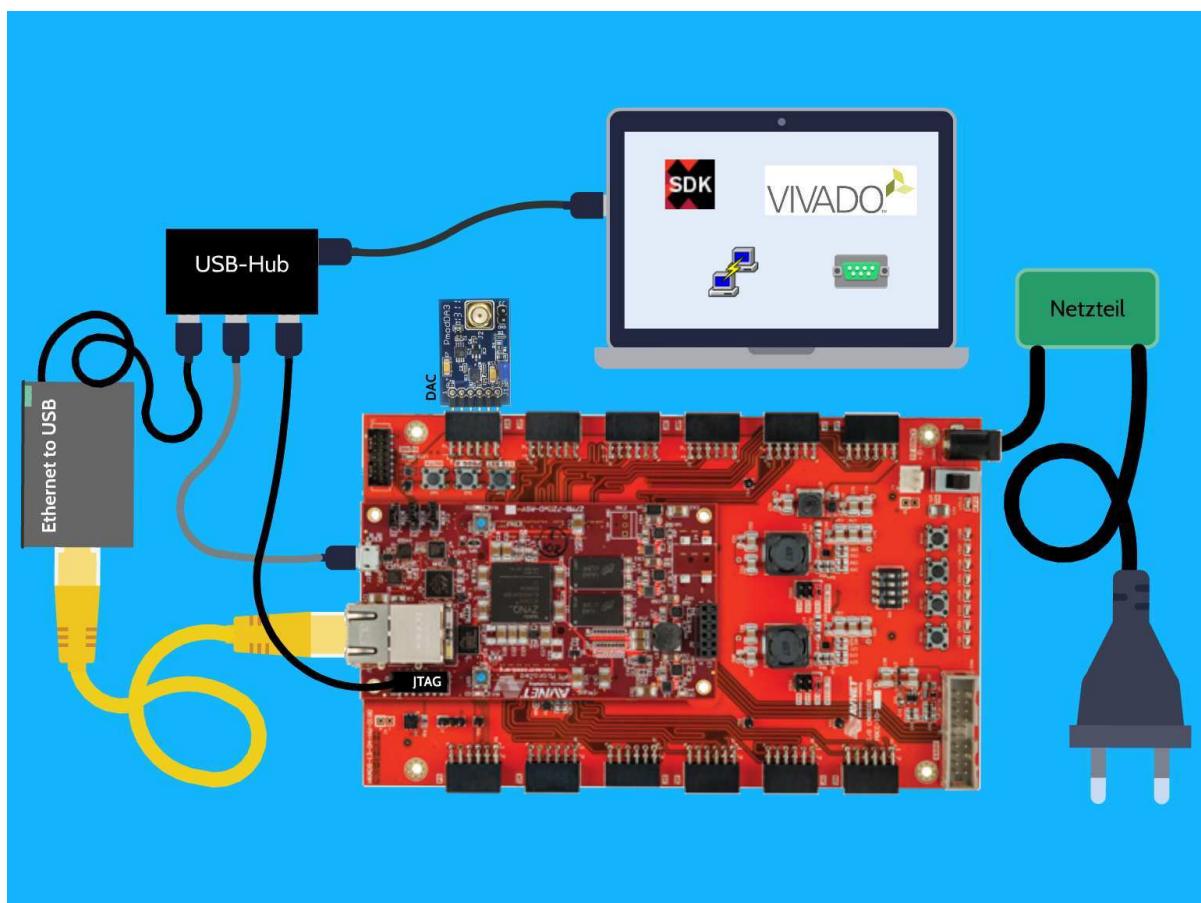


Abbildung 2.1.: Hardware Aufbau

# 3. Grundlagen

## 3.1. AXI Bus

Das Advanced eXtensible Interface (AXI) ist ein high-Performance Bus zwischen Master- und Slave-Einheiten. Er ist besonders gut für die On-Chip Kommunikation.

Es werden fünf Kanäle zwischen Master und Slave benötigt - zwei Kanäle Read, drei Kanäle Write (siehe Abbildung 3.1 und 3.2). Für eine Übertragung findet über die Adress-Kanäle eine Adress- und Kontrollphase statt, bevor der eigentliche Datenaustausch ausgeführt wird.

Da sowohl die Write- als auch die Read-Address-Channel sehr ähnlich sind, werden beide zusammen erklärt und einfach der 'Ax'-Präfix ('x' für 'R' oder 'W') verwendet, um auf ein Adress-Signal zu verweisen.

### 3.1.1. AXI Transfer

Für eine Übertragung auf dem AXI-Bus wird zuerst ein Handshake-Mechanismus ausgeführt. Das **AxVALID**-Signal wird von der Quelle gesendet, um die Empfängereinheit darüber zu informieren, dass die Daten auf dem Kanal gültig sind und von diesem Taktzyklus an gelesen werden können. Der Empfänger antwortet mit dem **AxREADY**-Signal, dass er bereit dafür ist.

Verschiedene andere Werte, die mit diesem Request verbunden sind, geben Auskunft über die Größe und Länge der angeforderten Transfers. [10, 17]

- **AxSIZE:** Ist ein 3-Bit Wert, mit dem ausgewählt wird, wie breit eine Dateneinheit für die Datenübertragung ist. Man kann maximal 128 Byte auswählen.
- **AxLEN:** Mit einer einzigen Anfrage auf dem Adresskanal kann man zwischen einem und 256 Werten anfordern, indem man AxLEN auf einen Wert zwischen 0 und 255 setzt. Das ist vielleicht die wichtigste Eigenschaft des AXI-Busses.
- **AxBURST:** Ist ein 2-Bit Wert, mit dem die Adressierungsart ausgewählt wird.

### 3.1.2. Burst Adressierung

Es gibt drei Typen für Burst Adressierung: FIXED, INCREMENT und WRAP.

Bei **fester Adressierung** wird die erste Adresse über AxADDR übergeben und verändert sich nie. Dies ist der perfekte Adressierungsmodus, um aus einem First In First Out (FIFO) zu lesen, da er immer mit einer einzigen Adresse dargestellt wird.

**Inkrementelle Adressierung** ist die üblichere Adressierungsart, die zum Beispiel auch für `memcpy` verwendet wird. Die erste Adresse wird mit der Größe AxSIZE weiter gezählt. Dies kann sehr komplex werden, denn AxSIZE muss laut AXI-Spezifikation nicht der Bus-Size entsprechen.

Die Grundidee des **Wrappings** ist, dass die Adressbits sich so verhalten, als ob nur einige Bits

inkrementieren und andere nicht. Dies findet bei Cache-Zugriffen Anwendung, ist sehr komplex und wird im Folgenden nicht weiter betrachtet. [10, 27]

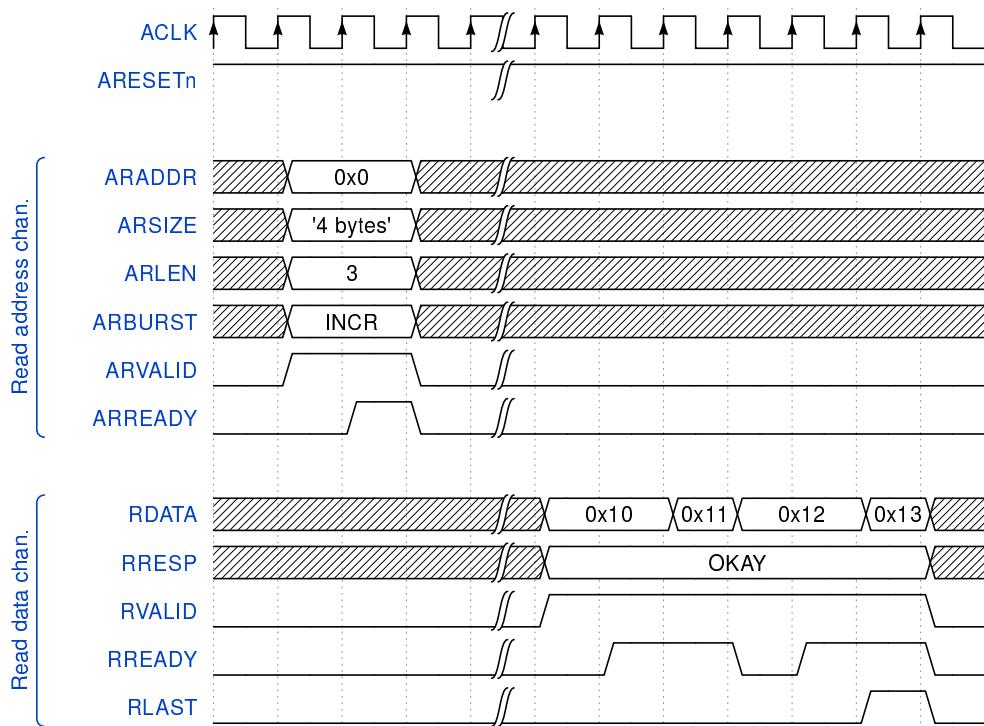


Abbildung 3.1.: AXI read transaction [20]

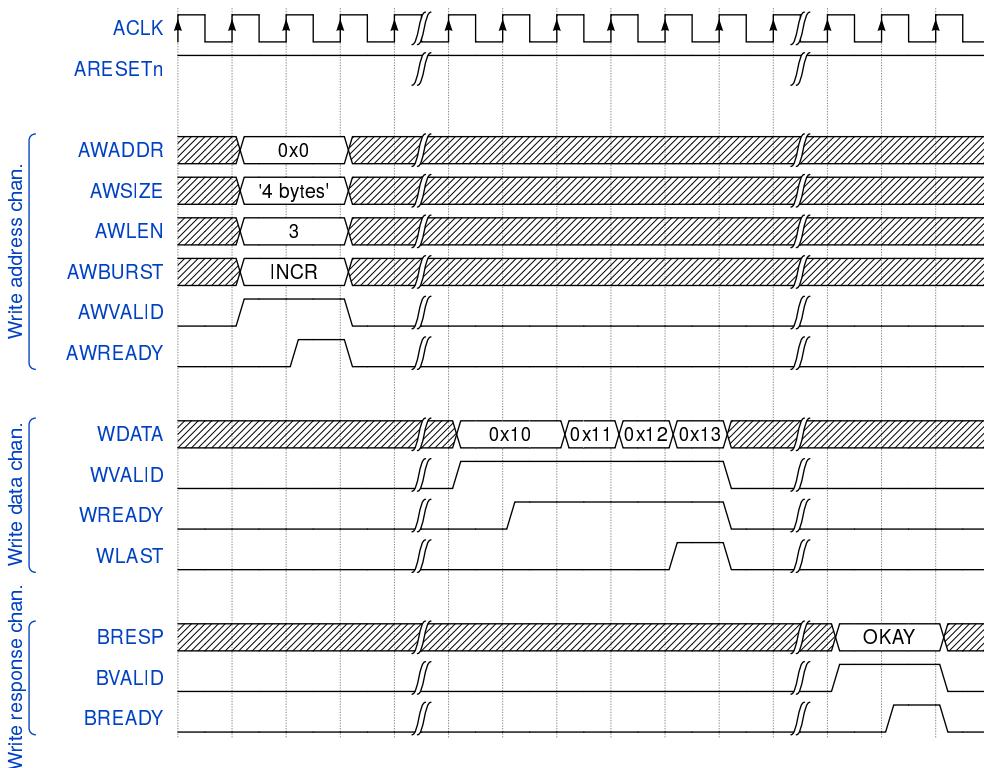


Abbildung 3.2.: AXI write transaction [20]

### 3.1.3. AXI-Lite

Die einfachere AXI Variante ist in der Funktionalität stark eingeschränkt. Der wichtigste Unterschied ist, dass immer nur eine Übertragung pro Burst zugelassen ist. [27]

### 3.1.4. AXI-Stream

AXI4-Stream ist ein Protokoll zum Transport beliebiger unidirektionaler Datenströme. Die Besonderheit ist, dass auf die Adressierung verzichtet wird und damit der Overhead reduziert ist. Dies ist möglich, da Streaming-Verbindungen immer Punkt-zu-Punkt-Verbindungen sind.

Beim Streaming werden Daten von einem Block zum anderen gesendet. Hinter Streaming-Geräten steckt die Idee, einen gleichmäßigen Fluss von Hochgeschwindigkeitsdaten zu gewährleisten, so dass normalerweise bei jedem Takt ein neuer Datenblock übertragen wird.

Beispiele für Streaming-Schnittstellen sind der Anschluss von Digital to Analog Converter (DAC) und Analog to Digital Converter (ADC), Videobussen, etc.

In der Abbildung 3.3 aus dem AXI-Referenzhandbuch von Xilinx werden zwei Transaktionen dargestellt. Die erste Transaktion besteht aus vier Datenzyklen, D0 bis D3. Der letzte Datentakt wird durch das **TLAST**-Signal markiert. Die zweite Transaktion hat 'n' Datenzyklen.

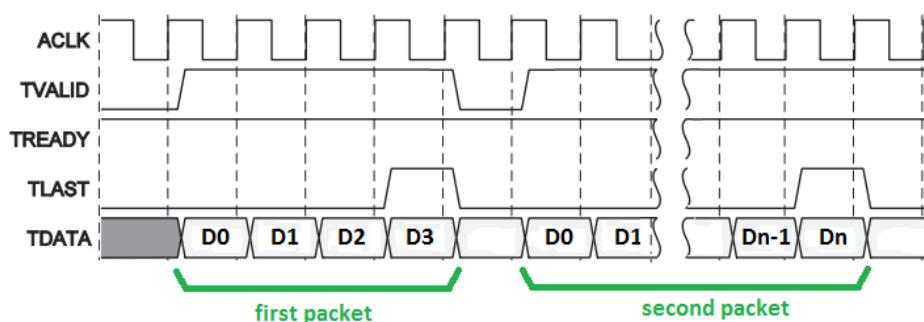


Abbildung 3.3.: AXI-Stream von zwei Paketen [27]

**TVALID** und **TREADY** werden für die Flusssteuerung verwendet. Sie müssen während der Transaktion immer 'high' sein. Wenn der Sender nicht bereit ist, weitere Daten zu senden, geht das TVALID-Signal auf 'low'. Ähnlich verhält es sich, wenn die Empfangsseite eines AXI-Streams nicht bereit ist, mehr Daten zu empfangen, dann wird das TREADY-Signal deaktiviert.

Ein typisches Szenario für die Aufhebung des TVALID-Signals ist, wenn der FIFO für die Sendedaten leer ist. Wenn hingegen der Empfangsdaten-FIFO voll ist, wechselt TREADY auf 'low'. [6, 27]

## 3.2. Embedded Linux Grundlagen

Für die Konfiguration und Verwendung des Embedded Linux Systems müssen einige Grundbestandteile und Prozesse verstanden werden. In der folgenden Sammlung gibt es einen kurzen Überblick über die wichtigsten Themen, die später auch verwendet werden.

### 3.2.1. Linux

Die Verwendung von Linux ist kostenlos und besteht aus open-source Software. Außerdem ist es reich an Features, die dem Nutzer viele Arbeiten erleichtern. Das System bietet unter anderem an: symmetric multiprocessing, preemptive multitasking, shared libraries, device drivers, memory management, IP networking und support for multiple file systems. Der Linux Kernel ist demzufolge auch sehr stark konfigurierbar.

An dieser Stelle sollte auch kurz der Unterschied zwischen Kernel Space und User Space erwähnt werden. [3]

- Im Kernel-Space laufen kritische Operationen wie Device Driver und Kernel Code. Diese haben uneingeschränkten Zugriff auf die darunterliegende Hardware. Außerdem werden Interrupts vom Kernel verarbeitet.
- Im User-Space laufen Anwendungen, die ihren eigenen Speicher über ein virtuelles Memory System anlegen. Physikalischer Speicher kann nur über Kernel-APIs erreicht werden. Das verkompliziert den Aufbau der zu realisierenden Datenverbindung.

### 3.2.2. Boot Prozess

Der Boot Prozess besteht im Wesentlichen aus vier Stufen: [3]

1. BootROM: der Boot-Modus (NAND, QSPI, JTAG) wird beim MicroZed durch Jumper-Stellungen ausgewählt (Abbildung 3.4). Vom gewählten Ort wird dann der First-Stage Boot Loader (FSBL) geladen.
2. FSBL: initialisiert externe Speicher und System Clocks.
3. U-Boot: lädt den Linux Kernel und gibt den Device Tree weiter
4. Linux Kernel: initialisiert die System HW und mountet das root-File System (FS)

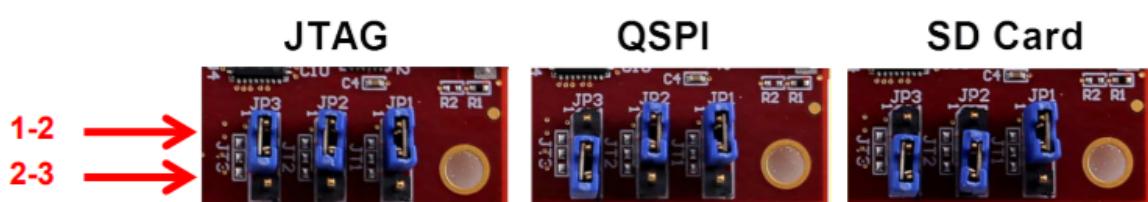


Abbildung 3.4.: MicroZed Boot-Mode Jumperstellungen [44]

### 3.2.3. Root File System

Das Root-FS beinhaltet die Anwendungen, Konfigurationsfiles, Bibliotheken und weitere Daten. [3]

- /dev System Devices
- /root Aufbewahrung von Super User Daten
- /mnt Mount Punkt für weitere FSs
- /lib System Bibliotheken
- /sys /proc Virtuelle FSs (wie Registry)
- /usr Programme

### 3.2.4. Device Drivers

Device Drivers abstrahieren Hardware (HW) Details weg von der Software (SW)-Schicht. Also ist der Treiber ein System-Call-Interface, das Zugriff auf physikalische Hardware Ressourcen ermöglicht. Vereinfacht ausgedrückt lässt der Device Driver das User-Space-Programm denken, dass ein Device einfach nur ein File unter /dev ist. Man kann Treiber fest in das System einbauen oder als ladbares Modul einsetzen.

Die Kommunikation mit den Treibern erfolgt über /dev und /sys.

Vereinfacht betrachtet wird /dev für den Datenzugriff verwendet. Der Zugriff auf das Gerät. /sys hingegen wird eher für die Feinabstimmung verwendet - die Verwaltung des Geräts. [16]

### 3.2.5. Device Tree

Ein Device Tree ist eine Baumdatenstruktur mit Knoten, die die physischen Geräte in einem System beschreibt. Die Datenstruktur wird beim Bootprozess an den Kernel weitergegeben. Der Device Tree wird in einer textbasierten Form als Device Tree Source (DTS) File erstellt (.dtsi Files sind Include Dateien). Mit dem im PetaLinux integrierten Device Tree Compiler (DTC) wird dieses dann in ein Binary (Device Tree Blob (DTB)) kompiliert.

Der Baum startet mit einem Wurzelknoten. Busse sind Zweige. Geräte, Speicher und Peripherie sind Knoten. [18]

Die wichtigsten Parameter der Knoten: [5]

- compatible: wird verwendet, um ein Gerät mit einem Gerätetreiber zu verbinden.
- interrupts: enthält die vom Gerät verwendete Interrupt-Nummer.
- reg: enthält den vom Gerät verwendeten Speicherbereich.

Mit einem '&' Zeichen kann man existierende Knoten referenzieren.

Auf dem Linux System kann man sich den Inhalt des Device Trees mit folgendem Befehl anschauen: [18]

```
$ ls -l /sys/firmware/devicetree/base/
```

### 3.3. Direct Memory Access

Der Direct Memory Access (DMA)-Zugriff zielt darauf ab, den Prozessor zu entlasten und eine schnelle Datenübertragung zu ermöglichen. Angeschlossen über das Bussystem erlaubt die Technik Kommunikation zwischen Speichern und Peripheriegeräten.

Die Zync-Architektur bietet verschiedene DMA-Varianten an, die in Abschnitt 4.2 genauer betrachtet werden. Alle Umsetzungen brauchen irgend eine Art von Steuerung. Im Rahmen der Arbeit werden die DMA-Bausteine nur im Direct Register Mode verwendet. Dabei werden die Register der Baugruppe geschrieben und gelesen um die Übertragungen anzustoßen, verwalten und konfigurieren zu können. Für die Transaktionen werden meist zusammenhängende Buffer benötigt. Für fragmentierte Zugriffe wird ein spezieller Mode bereitgestellt. Im Folgenden findet diese Variante Erwähnung, da sie für reale Konzepte wichtig ist.

#### 3.3.1. Scatter Gather Mode

Der Scatter/Gather (SG) Modus scheint eine interessante Möglichkeit zu sein, auch ohne zusammenhängende Buffer effizient Daten zu übertragen.

Der SG-Modus bietet dem Benutzer mehr Flexibilität, er ist jedoch auch viel komplexer zu verwenden als der Direct Register Mode. Wenn der Modus aktiviert ist (Tickbox im DMA Block), wird eine separate AXI4-Schnittstelle freigelegt, die zum Abrufen von Buffer-Deskriptoren verwendet wird. Buffer-Deskriptoren sind Knoten in einer ringförmig verketteten Liste (auch als Buffer-Deskriptor-Ring bezeichnet). Jeder Knoten enthält einen Zeiger auf die tatsächliche Daten-Payload. Die Software ist für die Erstellung und Pflege der Buffer-Deskriptoren verantwortlich. [23]

Mit dem SG-Modus ist es möglich, Daten aus einem nicht zusammenhängenden Speicherblock in einen einzigen Datenstrom zu schreiben, oder Daten aus einem Datenstrom zu lesen und in mehrere Puffer zu schreiben, wie in den Deskriptoren vorher festgelegt. [40, 26]

Diese Funktion erscheint sinnvoll, da es nicht immer einfach ist, einen zusammenhängenden Puffer für eine große Übertragung bereitzustellen. Damit können fragmentierte Stücke verwendet werden - da, wo eben gerade Platz ist. Im Rahmen dieser Arbeit wurde die Funktion nicht getestet, da die Deskriptor-Verwaltung in diesem Rahmen zu komplex ist.

# 4. Datentransfer Schnittstelle

Um alle Vorteile der Zynq Plattform zu nutzen, muss man Daten zwischen der eigenen Hardware (PL) und dem PS übertragen können - und das möglichst effizient.

Es stellt sich die Frage: „Welches ist die beste Art und Weise Daten mit hohem Durchsatz von einer eigenen Peripherie zum DDR-Speicher zu übertragen?“

## 4.1. Diskussion der Datentransfer Schnittstellen

Beim Betrachten des Systemblockdiagramms für das Zynq-Gerät erkennt man drei Hauptschnittstellen zwischen der PL und dem PS: General Purpose (GP)-, High Performance (HP)- und Accelerator Coherency Port (ACP)-Ports.

Abbildung 4.1 zeigt eine vereinfachte Topologie mit den Interfaces in Anwendung.

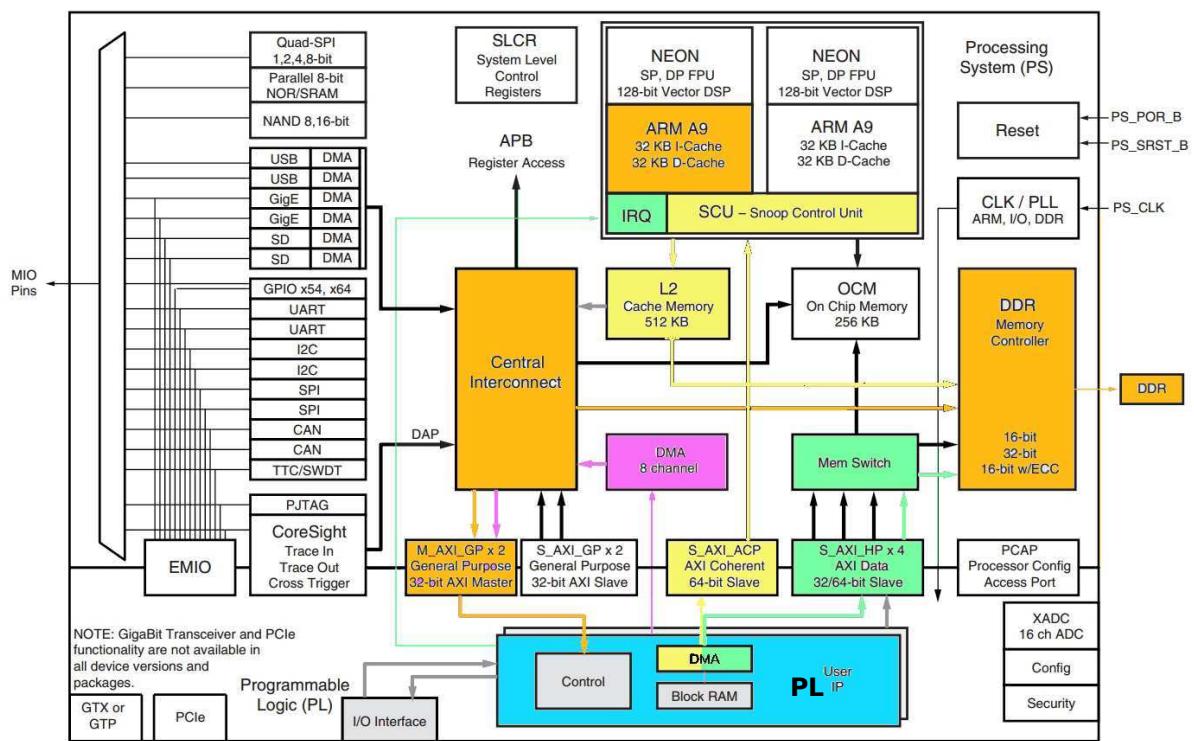
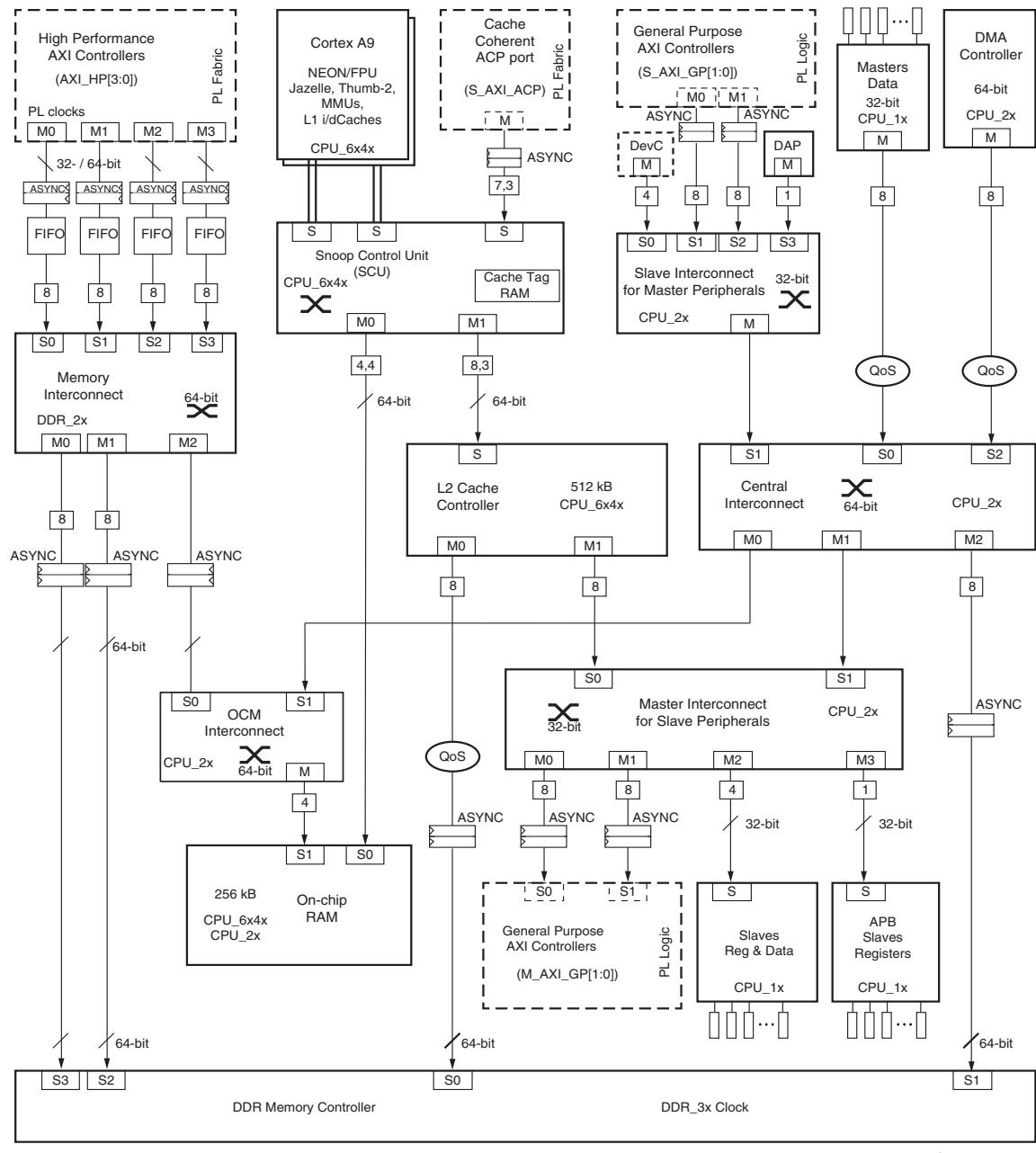


Abbildung 4.1.: Topologie: ACP gelb, HP mit DMA grün, DMAC pink, GP/mehrfach benutzt orange [43]

Der interne Datenpfad, aus dem sich die Vor- und Nachteile der Interfaces ableiten lassen, ist in Abbildung 4.2 gezeigt.



UG585\_c5\_01\_120813

Abbildung 4.2.: Interconnect Block Diagramm [43]

### 4.1.1. ACP

Der Accelerator Coherency Port (ACP) ist über den Snoop Controller direkt mit dem L1 und L2 Cash des Prozessors verbunden. Damit sind sehr geringe Latenzen für Daten, die die CPU verwendet, möglich. Die Zugriffe können cash-coherent passieren. Auf den ACP-Port wird auf Grund der Komplexität nicht weiter eingegangen.

### 4.1.2. GP-Interfaces

Bei den General Purpose (GP)-Ports handelt es sich um ein Interface zum Standard AXI-Bus. Es gibt ein paar Engpässe auf dem Pfad von den GP-Ports zum DDR-Speicher. Die Verbindung führt durch

zwei Interconnects, außerdem ist die Busbreite immer auf 32-Bit begrenzt (siehe Abbildung 4.2). Diese Schnittstelle ist folglich nur für den allgemeinen Gebrauch und nicht für hohe Leistung gedacht.

### 4.1.3. HP-Interfaces

Der High Performance (HP)-Port hingegen unterstützt auch 64-Bit Busbreite. Außerdem sind FIFOs zum Datenzwischenspeichern verbaut, was die Performance verbessert. Des Weiteren gibt es zwei DDR-Memory-Ports, zu denen verbunden werden kann. Wenn man mehrere HP-Ports zur Verbindung zum DDR-Speicher verwenden möchte, sollte man jeweils einen Port aus 0/1 und 2/3 auswählen, da sich sonst die Zusammenlegung des DDR-Interconnects auf die Performance auswirkt (siehe Abbildung 4.3).

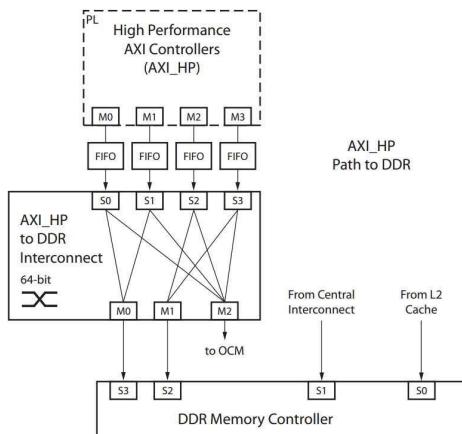


Abbildung 4.3.: AXI-HP Path to DDR [43]

## 4.2. Vorstellung der Datentransfer Management Einheiten

An diesem Punkt ist es klar, dass die HP-Ports zu verwenden sind, um die höchste Leistung zu erzielen. Da eine ausschließlich vom Prozessor überwachte Datenübertragung sehr ineffizient ist, soll ein extra Baustein diese Aufgabe übernehmen. Xilinx bietet hierfür verschiedene (DMA)-Möglichkeiten an.

### 4.2.1. DMAC (PS-DMA)

Bei einem ersten Überblick über die Architektur findet man den PS DMA Controller (auch mit PL330 oder Direct Memory Access Controller (DMAC) bezeichnet). Der Einsatz des DMACs spart PL-Ressourcen, weil er sich auf der PS-Seite befindet. Da der DMAC jedoch keinen Zugriff auf die HP-Ports hat und die Handhabung aufwendig sein soll, muss ein anderer Baustein gefunden werden. [2]

### 4.2.2. AXI-Datamover

Der AXI-Datamover wird für Anwendungen verwendet, die über den DMA-Zugriff noch zusätzlichen Zugriff auf die Hardware benötigen. Die Verwendung ist sehr aufwendig in der System Planung - also auch nicht die richtige Haupteinheit.

### 4.2.3. AXI-Streaming-FIFO

Das AXI-Streaming-FIFO ist einfach ein FIFO mit einer AXI Stream-Schnittstelle auf der einen Seite und einer AXI (oder AXI Lite) Schnittstelle auf der anderen Seite (siehe Abbildung 4.4). Damit ermöglicht er Memory-Mapped-Zugriff auf eine AXI Streaming-Schnittstelle. Außerdem kann man FIFOs zur Clock Conversion einsetzen. Für eine performante Übertragung kommt der FIFO jedoch nicht infrage, da alle Anfragen über Software angestoßen werden müssen.

Der AXI Stream FIFO sollte nicht mit dem AXI Stream Data FIFO verwechselt werden, der in Abbildung 4.5 gezeigt ist. Dieser stellt nur einen FIFO zwischen zwei Streams dar. [28]

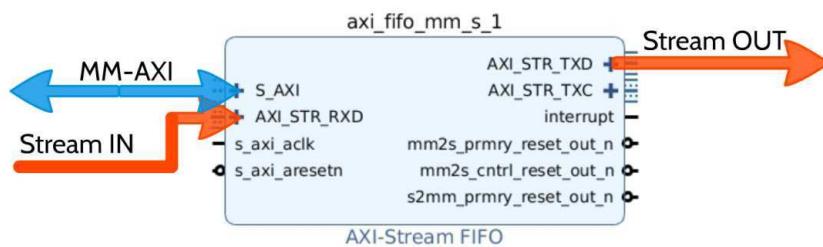


Abbildung 4.4.: AXI Stream FIFO IP-Block

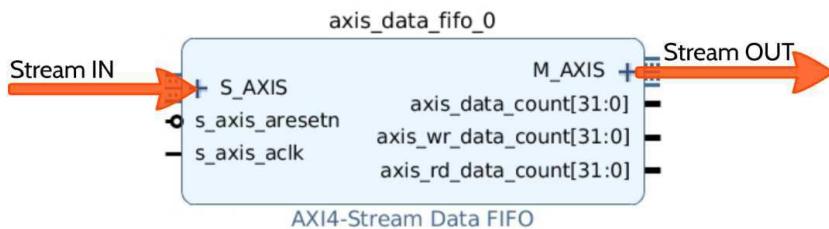


Abbildung 4.5.: AXI Stream Data FIFO IP-Block

### 4.2.4. AXI-DMA

Der AXI-DMA ist in Anwendungen mit großen Datenmengen vom Typ Streaming wie z.B. ADC/DAC-Schnittstellen, Ethernet usw. einzusetzen. Damit ist der AXI-DMA-Intellectual Property (IP)-Block der wahrscheinlich nützlichste Block für die meisten high-performance Anwendungen. Außerdem ist der Block ein Konverter von Memory Mapped Zugriffen zu Stream und umgekehrt. [26]

In Abbildung 4.6 ist der Datenfluss im IP-Block angedeutet. In einem normalen Anwendungsfall besitzt der Block drei Master- und zwei Slave-Ports. Es ist sehr hilfreich, die Benennungs-Konvention zu verstehen. Der erste Buchstabe unterscheidet zwischen **M**aster und **S**lave. Dahinter ist die Bus-Art angegeben (**AXI**, **AXI-Lite** oder **AXIS** für Stream). Am Ende wird die „Richtung“ mit Memory Mapped to Stream (**MM2S**) oder Stream to Memory Mapped (**S2MM**) angegeben. Der daraus resultierende Datenfluss wird hier farblich verdeutlicht.

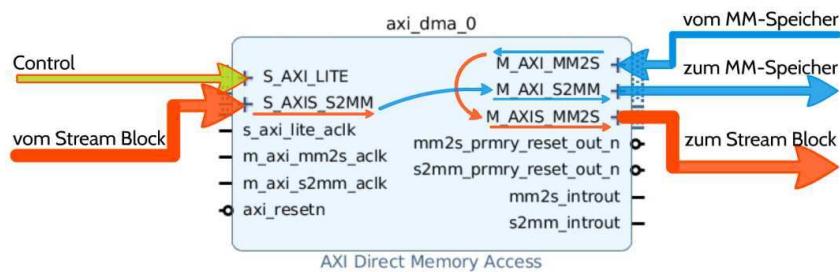


Abbildung 4.6.: AXI-DMA IP-Block Datenfluss

### AXI-DMA Accelerator Example

Xilinx stellt unter <https://www.xilinx.com/support/answers/58582.html> ein Beispiel Design vor, bei dem der AXI-DMA für eine Accelerator-Anwendung eingesetzt wird.

Beim typischen Accelerator-Anwendungsfall stammen die Daten aus dem ARM-Prozessorsystem, von wo sie an einen „maßgeschneiderten“ Co-Prozessor in der PL gesendet und schließlich zur Endverarbeitung an den Prozessor zurückgeschickt werden.

Die Stimulusdaten im Beispiel (eine Linearkombination aus mehreren Cosinuswellen bei verschiedenen Frequenzen) werden über den AXI-DMA aus dem Hauptspeicher geholt und an den Xilinx Fast Fourier Transformation (FFT) Core gesendet. Sobald der Core fertig mit der Berechnung der FFT ist, sendet er die Daten zurück an einen anderen Buffer im Hauptspeicher. Schließlich gibt die DMA Engine einen Interrupt aus, um dem ARM Prozessor mitzuteilen, dass die FFT-Daten dem Prozessor zur Verfügung stehen. Ein AXI General Purpose Input Output (GPIO)-Core wird verwendet, um dem Prozessor zu ermöglichen, den FFT-Block mit verschiedenen Parametern zu konfigurieren. Das vereinfachte Blockdesign kann man in Abbildung 4.7 sehen. [24]

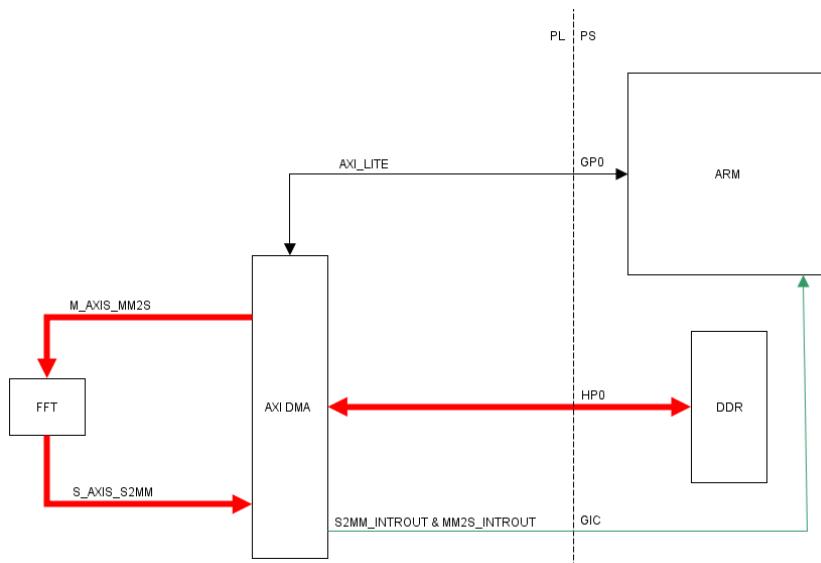


Abbildung 4.7.: Example Design - Zynq-based FFT co-processor using the AXI DMA [24]

Das Beispieldesign enthält alle notwendigen Daten und Dokumentationen. Dadurch ist die Bare-Metal-Application und das Design nach einigen Änderungen auch auf dem MicroZed lauffähig zu bekommen (Projekt auf Datenträger beigelegt).

#### 4.2.5. AXI-VDMA

Der AXI-Video Direct Memory Access (VDMA) ist dem AXI-DMA sehr ähnlich. Wenn eine Übertragung von Video- oder Bilddaten notwendig ist, findet dieser IP-Block Anwendung. Er erlaubt die einfache Übertragung von zweidimensionalen Daten.

#### 4.2.6. AXI-CDMA

Der AXI-Central Direct Memory Access (CDMA) bietet direkten Speicherzugriff mit hoher Bandbreite zwischen einer dem Speicher zugeordneten Quell- und Zieladresse unter Verwendung des AXI4-Protokolls an (Memory Mapped Zugriffe).

Der Zugriff auf Initialisierungs-, Status- und Steuerregister erfolgt über eine AXI4-Lite-Slave-Schnittstelle.

Abbildung 4.8 soll klarstellen, dass der CDMA über seinen M\_AXI Port alle Geräte erreichen kann, zu denen er Zugang über den angeschlossenen Interconnect- oder Smartconnect-Block hat.

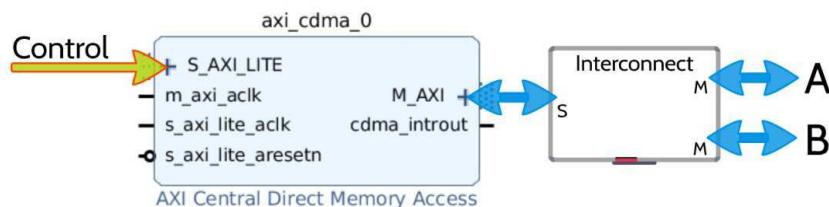


Abbildung 4.8.: AXI-CDMA IP-Block Datenfluss

Der CDMA wird z.B. häufig in Peripheral Component Interconnect Express (PCIe)-Systemen verwendet, da die PCIe to AXI Bridge ein AXI Interface zur Verfügung stellt. [25]

### 4.3. Auswahl und Begründung

Da zu den Randbedingungen mit einer Custom Logik als Memory-Mapped Baustein (fir\_memo\_top) nur eine Memory-Mapped DMA-Lösung passt, wurde der AXI-CDMA ausgewählt. Bei Memory-Mapped Systemen besitzt jede Speicherzelle eine Adresse. Bei den Übertragungen ist dann immer genau festgelegt, von welcher Quelladresse zu welcher Zieladresse übertragen wird. Der verallgemeinerte Anwendungsfall für den AXI-CDMA, an dem im Folgenden gearbeitet wird, ist der Memory-Mapped Datenaustausch zwischen Hauptspeicher (DDR-Random-Access Memory (RAM)) und BRAM.

Ohne die Randbedingung mit dem Memory-Mapped Zielbaustein sind aus jetziger Sicht Stream-Übertragungen zwischen PS und PL am sinnvollsten. Dies lässt sich mit den allgemeinen Anwendungsfällen begründen. Die PL kann als Co-Prozessor (Accelerator) verwendet werden (siehe Abschnitt 4.2.4) oder wird für Datenerfassung oder Ausgabe gebraucht. Jedes Mal handelt es sich um kontinuierliche Daten, die nicht unbedingt einzeln adressiert werden müssen. Hierbei kommt dann der AXI-DMA zum Einsatz. Damit Stream-Übertragungen möglich sind, müssen für die HW-Logik entsprechende Blöcke mit der gewünschten Funktion erstellt oder ausgewählt werden.

# 5. Erstellung des Ausgangssystems

## 5.1. Grundsystem Petalinux

Im Folgenden werden die notwendigen Schritte erläutert, um ein Basissystem für das MicroZed Board zu erstellen, welches von einer SD-Karte gebootet wird.

### SD-Karte vorbereiten

Auf der SD-Karte müssen zwei Partitionen erstellt werden. Hierzu kann man das graphische Tool GParted verwenden. Die erste Partition wird mit 4 MB vorangehendem Abstand als FAT32 mit mindestens 100 MB formatiert. Sie wird später den Bootloader und Kernel Images enthalten, deshalb wird sie sinnvollerweise BOOT genannt. Auf der zweiten Partition wird das Root-FS gespeichert. Diese Partition sollte als EXT4 mit dem restlichen Speicherplatz formatiert werden - Label `rootfs`. [36]

### Hardware Plattform

Um ein Embedded-Linux mit den PetaLinux Tools zu erstellen, benötigt man zuerst das Hardware Design. Die benötigten Daten werden in Vivado über 'generate Bitstream' und anschließenden 'File->Export->Export Hardware' (include Bitstream) erstellt. Das Minimaldesign enthält bloß den Prozessorsystem Block.

### Projekt erstellen

Ein neues PetaLinux Projekt kann man nun vom Template 'zynq' erzeugen. Nach dem Befehl `petalinux-create` müssen alle weiteren Anweisungen im Projektordner ausgeführt werden. Als Nächstes sollte dann die Hardwarebeschreibung zum Projekt hinzugefügt werden. Hierzu ist der Pfad zum Ordner, der die exportierten Daten enthält, anzugeben. Im Normalfall ist es der `<vivado-projekt>.sdk` Ordner im Vivado-Projekt-Verzeichnis.

```
1 $ petalinux-create --type project --template zynq --name <projekt-name>
2 $ cd <projekt-name>
3 $ petalinux-config --get-hw-description=/<PATH to Vivado Projekt>/<
  vivado-projektname>.sdk
```

### PetaLinux aus BSP

Alternativ gibt es die Möglichkeit ein vorgefertigtes Board Support Package (BSP) zu verwenden. Dieses enthält alle notwendigen Design- und Konfigurationsdateien um ein lauffähiges System zu erstellen. Für einen ersten Test kann das passende BSP für die gewählte Software Version und das MicroZed Board von [http://zedboard.org/sites/default/files/design/mz7020\\_fmccc\\_2017\\_4.zip](http://zedboard.org/sites/default/files/design/mz7020_fmccc_2017_4.zip) heruntergeladen und entpackt werden.

Zuerst wechselt man dann mittels 'cd /PFAD/' in das Verzeichnis, indem das PetaLinux-Projekt erstellt werden soll. Dort wird dann das Projekt kreiert und erhält den Namen des BSP-Files. [12]

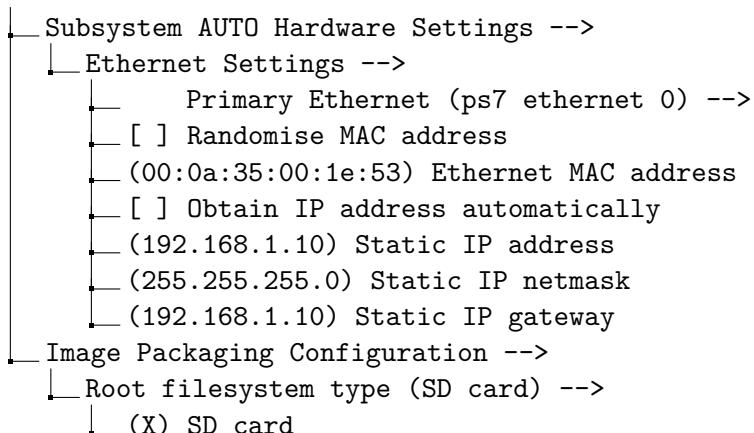
```
1 $ petalinux-create -t project -s /<Pfad to BSP>/file-name.bsp
```

## PetaLinux-Config

```
1 $ petalinux-config # Weitere Einstellungen vornehmen
```

In der PetaLinux Config können einige Einstellungen geändert werden, die spätere manuelle Schritte einsparen. Zum einen sollte man die IP-Adresse statisch konfigurieren, indem man 'Obtain IP address automatically' ausmarkiert (kein \* mehr zwischen [ ]) und dann die gewünschte statische Einstellung setzt.

Außerdem ist es praktisch, wenn das Root-FS direkt auf die SD-Karte geschrieben wird und sich automatisch einhängt. So bleiben Änderungen im FS auch erhalten. Die SD-Karte muss dementsprechend vorbereitet werden (siehe Abschnitt 5.1). [37]



In der `rootfs-config` können Apps und Module aktiviert werden. Die `kernel-config` ermöglicht es Treiber zu installieren <\*> oder nur als modulares Feature hinzuzufügen <M>. Diese Einstellungen sind für das Grundsystem noch nicht relevant und werden erst bei der Verwendung der Device Driver notwendig (Abschnitt 6.3).

```
1 $ petalinux-config -c rootfs
2 $ petalinux-config -c kernel
```

## Projekt Builden

Die Änderungen werden durch das Bauen des Projekts angewendet. Außerdem muss man noch das Boot Image erzeugen, welches den FSBL, den FPGA Bitstream und das U-Boot enthält.

```
1 $ petalinux-build
2 $ petalinux-package --boot --force --fsbl ./images/linux/zynq_fsbl.elf
   --fpga ./images/linux/BitFileName.bit --u-boot
```

## Kopieren auf die SD-Karte

Dann kann man die Dateien: BOOT.BIN, image.ub und rootfs.tar.gz auf die SD-Karte kopieren. Um den Kopierprozess auf die SD-Karte zu automatisieren, wurde ein kleines Shell-Skript geschrieben, welches die gesamte Arbeit übernimmt (Code 5.1).

Code 5.1: SD-Card-Config-Skript.sh

```
1 #!/bin/bash
2 echo -e "Kopieren der Dateien auf die SD Karte\n"
3 echo -e "man muss sich im <plnx-proj-root> Verzeichnis befinden\n"
4 DIRECTORY=""
5 # ins petalinux-projekt wechseln
6 while [ ! -d "$DIRECTORY" ]
7 do
8   read -e -p "<plnx-proj-root> Pfad angeben: " DIRECTORY
9 done
10 cd $DIRECTORY
11 # altes System entfernen
12 sudo chown -R $USER:$USER /media/$USER/rootfs/
13 rm -r /media/$USER/BOOT/*
14 rm -rf /media/$USER/rootfs/*
15 # Dateien kopieren
16 cp images/linux/BOOT.BIN /media/$USER/BOOT/
17 cp images/linux/image.ub /media/$USER/BOOT/
18 cp images/linux/rootfs.tar.gz /media/$USER/rootfs/
19 # rootfs entpacken
20 cd /media/$USER/rootfs
21 sudo tar xvf rootfs.tar.gz
22 # SD-Karte aushängen
23 sudo umount /media/$USER/BOOT
24 sudo umount /media/$USER/rootfs
```

Nach der Ausführung kann die SD-Karte wieder ins MicroZed Board eingebaut werden.

## Verbindung zum PC

Vom PC aus nutzt man drei verschiedene Verbindungen zur Kommunikation mit dem Board.

- Serielle Verbindung über Gtkterm (/dey/ttyUSB1 115200-8-N-1)
- SSH über Putty (192.168.1.10 Port 22)
- FTP über Dateimanager Krusader (192.168.1.10 Port 21)

In allen Verbindungen muss man sich mit Benutzername **root** und Passwort **root** anmelden.

Die serielle Verbindung ist die einfachste Möglichkeit auf das Board zuzugreifen. Sie wird meist nur zu Anfang benötigt, um die Netzwerkverbindung zum Laufen zu bringen. Im Embedded Linux muss dafür noch die Netzwerkadresse gesetzt werden (entfällt bei erweiterter Petalinux-Configuration nach Abschnitt 5.1).

Für die Netzwerkverbindung ist ein Netzwerkadapter Ethernet-LAN/USB am PC praktisch, da die normale Verbindung zum Internet nicht verändert wird. Die USB-Netzwerkverbindung muss dann noch, wie in Abbildung 5.1 zu sehen ist, konfiguriert werden.



Abbildung 5.1.: Netzwerkeinstellung am PC.

Die Verbindung kann man mit dem Ping Befehl testen.

```

1 $ ifconfig eth0 192.168.1.10    #setzt statische IP
2 $ ping 192.168.1.100           #Pingtest vom Zynq aus
3 $ ping 192.168.1.10            #Pingtest vom Rechner aus

```

## Verwendung des Systems

Um eigene Applikationen auf dem Board auszuführen, kompiliert man diese am besten in der SDK und kopiert das 'application.elf' File mit dem Dateimanager Krusader über die File Transfer Protocol (FTP)-Verbindung auf das Board. Über Secure Shell (SSH) muss die Applikation dann noch ausführbar gemacht werden. Dann kann man das Programm ausführen.

```

1 $ chmod u+x application # ausfuerbar machen
2 $ ./application          # starten

```

Alternativ kann auch über die SDK das Programm direkt gestartet oder debuggt werden.

## 5.2. SDK Workflow

### Neues Projekt

Um eine neue Linux Applikation zu erstellen, wählt man unter File-> New-> Other-> Xilinx-> Application Projekt, vergibt einen Namen und stellt die OS Plattform auf 'linux' (siehe Abbildung 5.2).

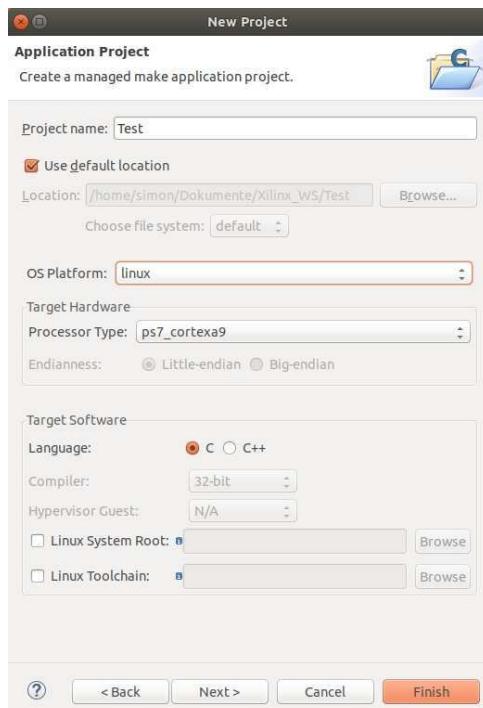


Abbildung 5.2.: Neue Linux Application im SDK

Mit Rechtsklick im Projekt Explorer auf die Applikation kann man unter Import-> General-> File System Dateien dem Projekt hinzufügen. Das Projekt wird automatisch mit dem Speichern gebaut.

## Debugging mit dem System Debugger

In der SDK wählt man zuerst die Applikation aus, die debuggt werden soll. Über Run-> Debug Configurations Doppelklick auf Xilinx C/C++ application (System Debugger) wird eine neue Konfiguration erstellt. Unter dem Reiter Target Setup kann man dieser nun einen neuen Namen geben und den Debug Type auf 'Linux Application Debug' umstellen. Die Connection stellt man mit Port 1534 und der IP-Adresse des Hosts ein (192.168.1.10). Unter dem Reiter Application wählt man das Projekt aus und muss gegebenenfalls die Pfade noch ergänzen (siehe Abbildung 5.3).

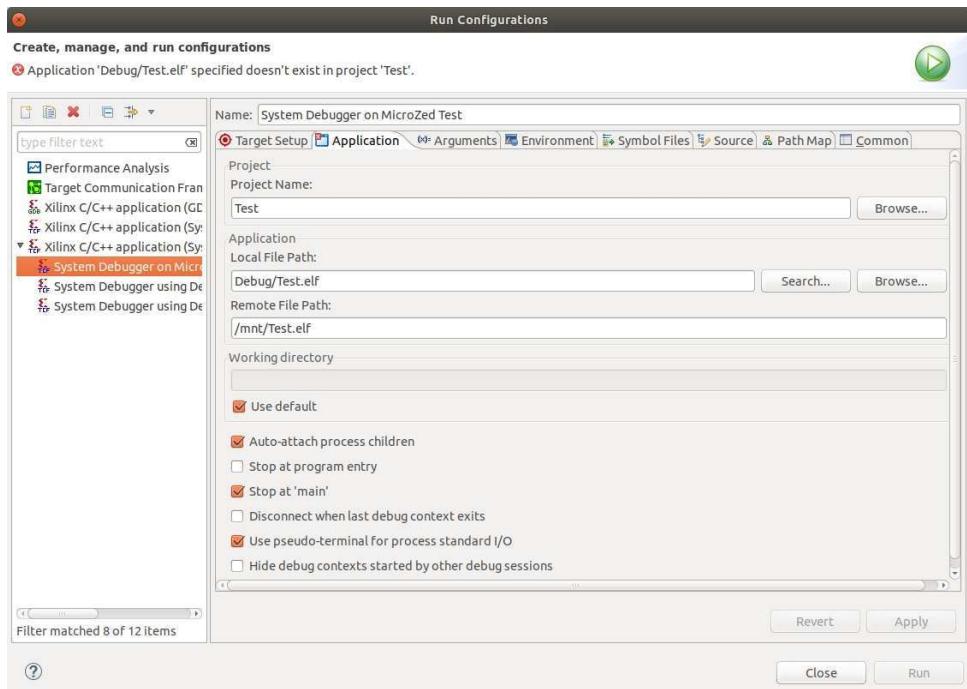


Abbildung 5.3.: Einstellung für den Remote System Debugger

Nach der Einstellung können dann die üblichen Debug und Run Buttons der Umgebung verwendet werden. [32]

Achtung: Hin und wieder kommt es zu Problemen, wenn eine Ausführung nicht zum Ende kommt. Die Verbindung zeigt dann: „Cannot download programm file; Text file busy“ und es wird eventuell eine alte Version des Programms ausgeführt. Diese Verklemmung kann durch einfaches Löschen des ELF-Files auf dem Hostsystem über `$ rm /mnt/Test.elf` behoben werden.

# 6. Realisierung

Für den Aufbau einer performanten Datenschnittstelle zwischen PS und PL wird ein solides Ausgangssystem benötigt. Dieses setzt sich vor allem aus einem lauffähigen Embedded-Linux und einer durchdachten Hardware Plattform zusammen.

Das in Abschnitt 5.1 beschriebene PetaLinux Grundsystem kann erst mal für einfache Anwendungen, die nicht auf den Device Tree zurückgreifen müssen, verwendet werden.

Nachdem das Konzept für das Hardwaredesign (Kommunikation vom DDR-RAM über HP-Ports via CDMA zu einem BRAM) fertiggestellt war, wurde eine Übertragung über eine Linux-Applikation mittels Registerzugriffen implementiert (Abschnitt 6.2). Diese stellt die Lösung zur Aufgabenstellung dar. Im Laufe der Entwicklung wurde jedoch auch klar, dass eine Realisierung mittels Treiber eine sauberere und performantere Umsetzung darstellen könnte. Mit der Thematik setzt sich Abschnitt 6.3 auseinander, jedoch nicht mehr in Bezug auf die Zielhardware.

## 6.1. Hardwaredesign

Bevor man im Vivado starten kann, ein Projekt für das MicroZed-Board zu entwerfen, sollte man das „MicroZed board definition file“ von <http://zedboard.org/support/documentation/1519> unter „MicroZed Board Definition Install for Vivado 2015.3 through 2017.4“ herunterladen und an die richtige Stelle in die Vivado-Installation entpacken (/opt/Xilinx/Vivado/2017.4/data/boards/board\_files/). Durch die Definitions-Files kann man das MicroZed Board bei der Erstellung eines neuen Projekts auswählen.

Für die Realisierung der Datenübertragung wurde in Kapitel 4 der CDMA-IP-Core ausgewählt. Mit seiner Hilfe sollen Daten vom DDR-RAM zur Zielhardware dem 'fir\_memo\_top\_v1\_0' Block und zurück übertragen werden können. Die Zielhardware kann man als BRAM abstrahieren, der zyklisch ausgelesen wird, und für Testzwecke die Daten über einen SPI-DAC analog ausgibt. Wenn erkennbare Signalformen eingespeichert werden, ist die Funktion mit dem Oszilloskop testbar.

Abbildung 6.1 zeigt das fertige Design mit den hervorgehobenen wichtigen Blöcken: PS, CDMA, BRAM-Controller und fir\_memo\_top.

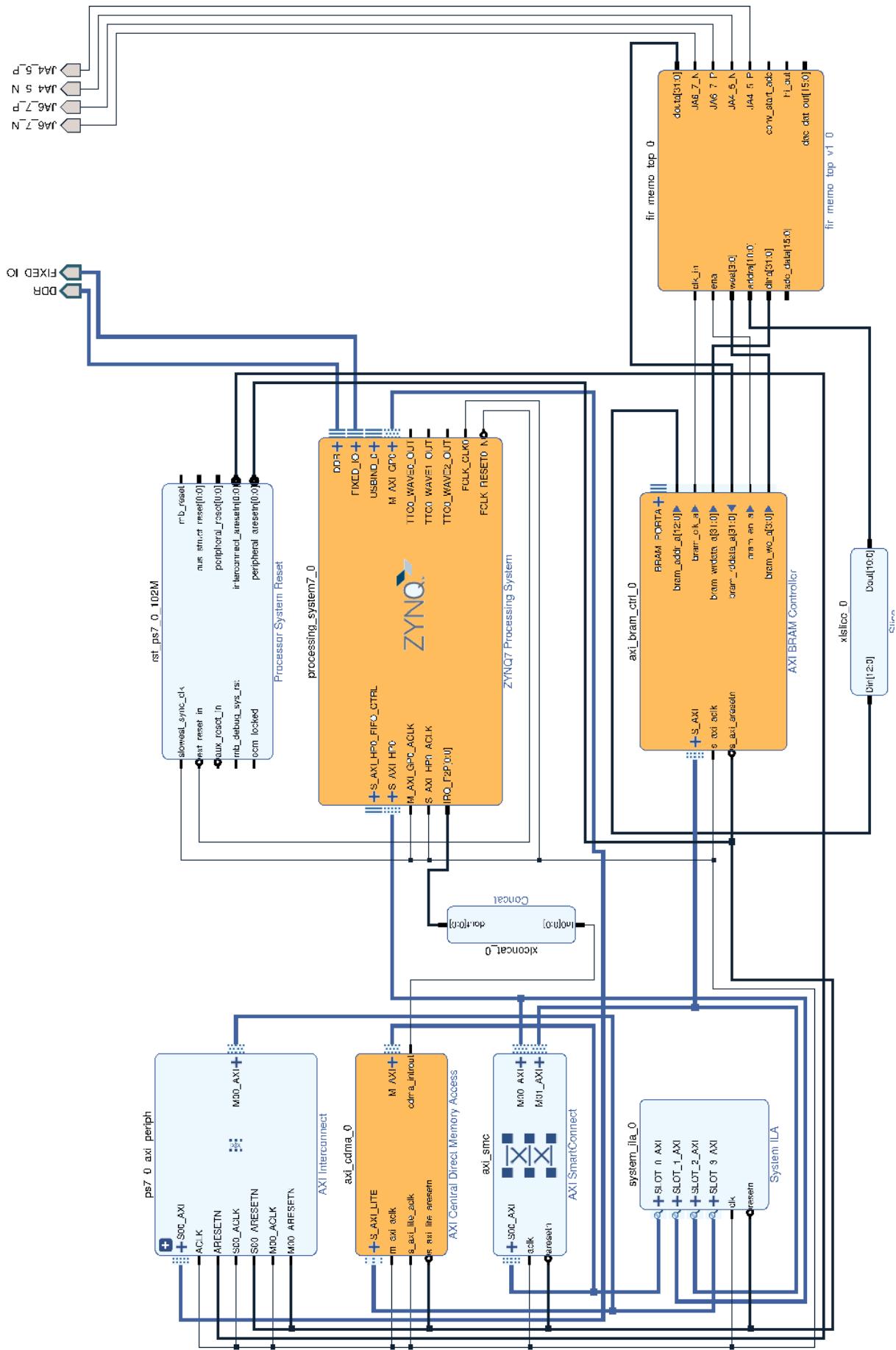


Abbildung 6.1.: CDMA-Hardware-Design - wichtige Blöcke orange eingefärbt

### 6.1.1. Funktionsweise fir\_memo\_top\_v1\_0

Um die Zielhardware richtig anzusprechen, lohnt sich ein Blick in das Design (Abbildung 6.2), um die Randbedingungen zu verstehen.

Besonders wichtig ist hier die Adressierung, die im Folgenden genauer betrachtet wird.

Mit den 11 (10 downto 0) eingehenden Adressleitungen können 2048 Byte adressiert werden. Der Speicher ist aus vier einzelnen 1 Byte Blöcken aufgebaut. Dadurch, dass der Prozessor immer mit 32 Bit (4 Byte) arbeitet, wird der Speicher mit seiner Word-Adresse angesprochen (immer 4 Byte auf einmal - also jede 4. Adresse). Im mem\_top bekommt jeder Byte-Block die Word-Adresse, indem die Adresse nur als '10 downto 2' übergeben wird. Die unteren 2 Bit, die immer 0,1,2,3 zählen, fallen einfach weg. Mit jeder vierten Adresse werden 4 Byte gelesen/geschrieben, indem die Daten der einzelnen Byte-Blöcke zu einem 32-Bit-Vektor 'din' bzw. 'dout' verkettet werden (zu sehen in Abbildung 6.2).

Die DAC-Ausgabe erfolgt durch periodisches Auslesen des Speichers. Ein Counter zählt mit einem um 100 herunter geteilten Takt die Adressen 0 bis 1023 durch. Der DAC wandelt immer 16 Bit Werte in einen Analogwert. Der Speicher ist jedoch 32 Bit breit - zwei Datenwerte liegen immer hintereinander.

Beispiel: 0x7f5f7ebe → Wert 1: 0x7f5f      Wert 2: 0x7ebe

Während mit den Counter-Bits '9 downto 1' die 512 mal 32 Bit ausgewählt werden, wird mit dem Least significant Bit (LSB) zwischen den Daten '31 downto 16' und '15 downto 0' gewechselt.

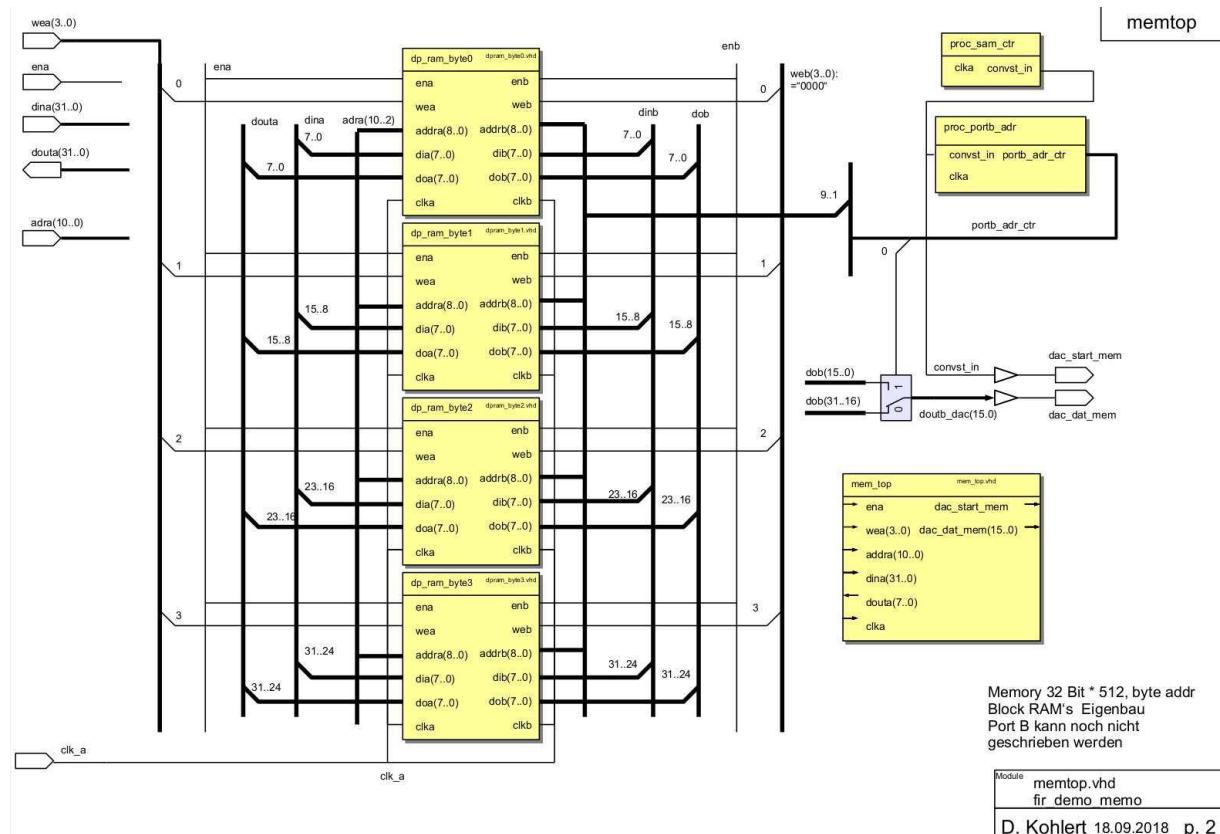


Abbildung 6.2.: fir\_memo\_top Design [7]

Der Speicher ist mit einem Sinus auf den ersten 1024 Byte vorbelegt, der Rest besitzt den Wert 0. Dies ergibt eine Signalform zusammengesetzt aus einem Sinus und anschließender gleichlanger Null-Phase (siehe Abbildung 6.3). Die PL wird mit einer 100 MHz Clock versorgt. Durch den Taktteiler 100 und insgesamt 1024 16-Bit-Werten ergibt sich eine mit 976 Hz wiederholende Form.

$$\frac{\text{PL-Clock}}{\text{Teiler}} = \frac{100 \text{ MHz}}{100} = \frac{1}{1024} = 976 \text{ Hz}$$

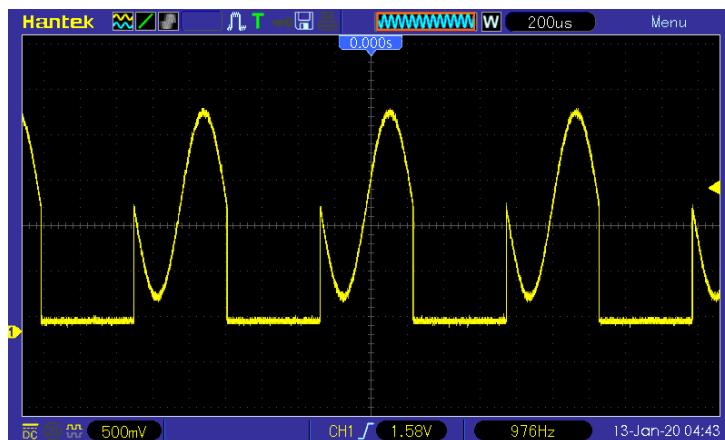


Abbildung 6.3.: Ausgabe des vorbelegten Speichers

### 6.1.2. Erstellung des Designs

Das vollständige Design ist in Anhang B zu sehen. Es folgt die Beschreibung der Erstellung mit einigen weiteren Informationen über die Design-Blöcke und den Workflow.

#### ZYNQ7 Processing System

Das Hardwaredesign besteht als Grundlage immer aus dem ZYNQ7 Processing System - dem Block, der die PS-Seite repräsentiert und konfigurierbar macht.

Dieser IP-Block muss als Erstes ins Block Design eingefügt werden. Anschließend kann man ein Tool command language (TCL)-Skript von AVNET nutzen, um einige sinnvolle Voreinstellungen vorzunehmen. Das Skript findet man auch auf der <http://zedboard.org/support/documentation/1519> Webseite unter „MicroZed Zynq PS Preset for Vivado 20XX.X (TCL)“. Über den Aufruf von `source` in der TCL-Console wird das Skript ausgeführt und setzt die Voreinstellungen für das MicroZed-Board.

```
1 $ source /link/to/MicroZed_PS_properties_v03.tcl
```

Für das CDMA-Design müssen noch einige Ports, Interrupts und weitere Eigenschaften aktiviert und umgestellt werden. Diese Einstellungen wurden über den 'Re-customize IP' Dialog (siehe Abbildung 6.4) vorgenommen und als 'Preset' gespeichert (ein TCL-Skript wird erstellt). Im gleichen Dialog kann man gespeicherte Presets auch wieder aktivieren.

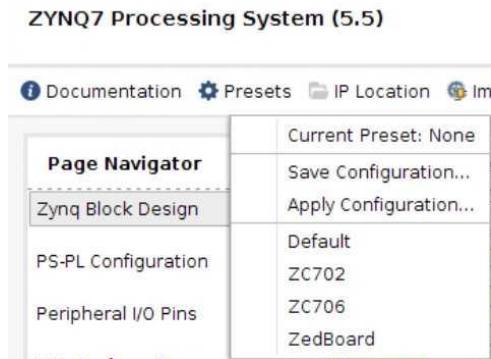


Abbildung 6.4.: Re-customize IP - ZYNQ PS - Preset

### **Fir\_Memo\_Top\_v1\_0**

Da der 'fir\_memo\_top\_v1\_0'-Block eine Eigenentwicklung von Professor Kohlert ist, muss er über den IP-Catalog hinzugefügt werden. Der Block hat sein eigenes Vivado-Projekt, in dem man die Very High Speed Integrated Circuit Hardware Description Language (VHDL)-Dateien bearbeiten kann. Nach einer Änderung wird man dann automatisch in den Projekten, die den Block nutzen, aufgefordert die Bibliothek upzudaten. In der im Folgenden genutzten Variante wurde der Adresscounter 'portb\_adr\_ctr' (9 downto 0) so abgeändert, dass er nicht nur die Hälfte, sondern alle Speicherzellen ausliest. Das geänderte Projekt ist auf dem der Arbeit beigefügten Datenträger abgespeichert. Der Block besitzt nicht nur die Memory-Funktion, sondern auch einen FIR-Filter sowie ADC-Datenanschlüsse. Diese Funktionen werden jedoch nicht verwendet.

### Weitere verwendete Blöcke

Als Nächstes fügt man alle weiteren benötigten IP-Blocks ein:

- AXI Central Direct Memory Access
- AXI BRAM Controller
- Concat
- Slice

Einige weitere Blöcke werden automatisch hinzugefügt. Die folgende Auflistung beschreibt die Funktion aller von Xilinx verwendeten IP-Cores. [29]

- **ZYNQ Processing System:** ist die Software-Schnittstelle rund um das Zynq-7000 Processing System.
- **AXI Central Direct Memory Access:** bietet direkten Speicherzugriff mit hoher Bandbreite zwischen einer Quell- und Zieladresse unter Verwendung des AXI4-Protokolls an.
- **AXI BRAM Controller:** kommuniziert mit einem lokalen BRAM.
- **Processor System Reset:** bietet individuelle Resets für ein vollständiges Prozessorsystem, einschließlich des Prozessors, der Interconnects und der Peripheriegeräte.

- **AXI Interconnect:** verbindet ein oder mehrere AXI memory-mapped Master-Geräte mit einem oder mehreren memory-mapped Slave-Geräten.
- **AXI SmartConnect:** wie Interconnect - konvertiert automatisch zwischen Schnittstellen mit unterschiedlichen Datenbreiten.
- **System ILA:** ist ein Logikanalysator, den man zur Überwachung der internen Signale und Schnittstellen eines Designs verwenden kann.
- **Concat:** dient zur Verkettung von Bussignalen unterschiedlicher Breite.
- **Slice:** wird verwendet, um Bits aus einem Busnetz zu entfernen.

Im Folgenden wird die Verbindung der Blocks Schritt für Schritt angefangen bei der Zielhardware beschrieben:

### DAC-Anschluss

Als erstes erstellt man die Ports für den Anschluss des DAC über Rechtsklick 'Create Port' im Blockdesign. Für die Nutzung muss die Belegung über das Constrain-File (<http://zedboard.org/support/documentation/1522>) Z7010 or Z7020 MicroZed with MBCC-IO-PCB-D\_v2.xdc noch festgelegt werden. In diesem File brauchen nur die entsprechenden Pins (JA Pmod - Bank 34: Pin 7-10) und die bank voltage für Bank 34 aktiviert sein.

### Verbindung BRAM-Controller - fir\_memo\_top\_v1\_0

Der fir\_memo\_top\_v1\_0-Baustein wird wie ein Block RAM angesteuert. Zuerst verringert man die Anzahl der BRAM-Interfaces in der Controller-Konfiguration auf eins. Für die Verbindung klappt man dann den BRAM\_PORTA aus und verbindet beide Bausteine per Hand. Hierbei gibt es eine Besonderheit: Der BRAM-Controller kann minimal 8 kB adressieren ('12 downto 0' Adressleitungen). Der memo\_top-Baustein hat jedoch nur 2 kB ('10 downto 0' Adressleitungen). Damit es zu keinen Warnungen kommt, kann man diese Verbindung mit dem Slice-Block so konfigurieren, dass nur die unteren 11 Leitungen verbunden werden.

Die restlichen Verbindungen sind logisch und in Abbildung 6.1 zu sehen.

### Verbindung PS - CDMA

Das PS besitzt durch die vorigen Einstellungen schon alle benötigten Ports und Konfigurationen. Im Anpassungsdialog vom CDMA-Block ist die Scatter Gather Tickbox auszumarkieren und die Adressbreite auf 32 einzustellen. Die Burst Size und die Datenbreite haben Auswirkungen auf die Übertragung und werden in Kapitel 8 - Analyse der Performance - genauer beschrieben.

Der Concat-Block wird zur Verkettung von Interrupt-Signalen verwendet. In diesem Design befindet sich nur ein Interrupt, weshalb die Anzahl der Ports am Concat auf eins reduziert werden muss. Das **dout** wird dann mit dem **IRQ\_F2P** des PS-Blocks verbunden und der **cdma\_intout** mit dem **In0**. Für die folgenden Verbindungen ist die Designer Assistance sehr hilfreich. Dort werden Verbindungen vorgeschlagen und Interconnect, SmartConnect und Reset Bausteine automatisch eingefügt. Achtung

jedoch, der erste Vorschlag ist nicht unbedingt der, den man erreichen möchte.

Gesteuert wird der CDMA über die **S\_AXI\_LITE**-Schnittstelle, die mit dem **M\_AXI\_GP0** des PS über ein AXI-Interconnect verbunden wird.

### Verbindung des Datenflusses

Um eine Datenverbindung zwischen DDR-RAM und BRAM mittels CDMA herzustellen, wird der **M\_AXI**-Port des CDMA über ein SmartConnect mit dem **S\_AXI\_HP0**-Port des PS und dem **S\_AXI**-Port des BRAM-Controllers verbunden.

### Address Editor

Im Addresseditor legt man noch die Adressbereiche der Komponenten fest. Diese müssen später mit denen im Programm übereinstimmen. In Abbildung 6.5 ist zu erkennen, dass der CDMA an Adresse 0x7E20000 liegt und dort dann mit dem entsprechenden Offset die Register des Blocks beschrieben werden können.

Der BRAM-Controller mit dem dahinter hängenden fir\_memo\_top muss auf die im VHDL-Code hinterlegte Adresse gesetzt werden. Den Range kann man nicht kleiner 8 kB wählen, was jedoch kein Problem verursacht, da einfach nur die vorhandenen 2 kB verbunden werden.

Dem HP-Port sollte ein Adressbereich des DDR-RAM zugeordnet werden. Der Bereich ist im Reference Manual UG585 in Tabelle 4-1 angegeben. Für das aktuelle System wurde der HP-Slave-Port 0 nur mit einem kleinen Stück (8 kB) von 0x20000000 bis 0x20001FFF konfiguriert. Diese DDR-Speicherstelle des Systems dient als Quell-/Zielpufferspeicher für den CDMA zum Lesen/Schreiben der Daten. Es schadet jedoch auch nicht, hier den ganzen Adressraum von 0x00000000 bis 0x3FFFFFFF anzugeben. Dann kann man im Programm eine beliebige Adresse verwenden, aber auch Memory-Konflikte mit dem Betriebssystem durch Doppelbelegung bekommen. Der hier verwendete Speicherbereich wurde extra im Device Tree reserviert (siehe Unterabschnitt 6.2.3). [42]

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
axi_cdma_0					
Data (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Mem0	0x4200_0000	8K	0x4200_1FFF
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x2000_0000	8K	0x2000_1FFF
processing_system7_0					
Data (32 address bits : 0x40000000 [ 1G ])					
axi_cdma_0	S_AXI_LITE	Reg	0x7E20_0000	64K	0x7E20_FFFF

Abbildung 6.5.: Address Editor-CDMA

## Fertigstellung

Zum Schluss sollte man mit 'Validate Design' (F6) den Aufbau testen. Anschließend kann mittels 'Generate Bitstream' die Synthese und Implementierung gestartet werden.

Der System Integrated Logic Analyzer (ILA) wird für die eigentliche Funktion nicht benötigt und braucht sehr viele Ressourcen. Er war aber sehr hilfreich für das Verständnis und zum Debuggen. Für eine finale Anwendung sollte er jedoch entfernt werden.

## TCL Skript

Um das Design später nochmal von Grund auf zu erzeugen, ist ein TCL-Skript sehr hilfreich. Alle durchgeführten Schritte werden im Hintergrund als TCL-Befehle ausgeführt. Die Befehle sieht man in der TCL-Console im Vivado. Von dort kann man alles am Ende kopieren und mit einiger Nacharbeit und Verallgemeinerung von Pfaden etc. ein eigenes TCL-Skript für den Aufbau erstellen. Ein solches Skript wurde erstellt. In den ersten Zeilen kann man Name und Verzeichnis anpassen. Eventuell sollte man auch die letzte Zeile auskommentieren, wenn nicht direkt der Bitstream erzeugt werden soll. Aufgerufen wird das Skript mit dem `source` Befehl im Welcome-Screen von Vivado.

```
1 $ source /link/to/Create-CDMA-Projekt.tcl
```

## 6.2. Linux-App für Memory-Mapped PS-PL Zugriffe

Auf dem Linux-System kann man den CDMA über eine C-Applikation ansprechen. Der Quellcode ist im Anhang C nachzulesen.

Die Entwicklung der Applikation verfolgte das Ziel, im PS-generierte Signaldaten mit Hilfe des CDMA auf einen Speicher im PL zu schreiben oder zurück zu lesen. Das Programm ist über Konsolen-Eingaben bedienbar, in denen verschiedene Signalformen für die Übertragung ausgewählt werden können. Die wichtigsten Schritte im Programm sind die Initialisierung des CDMA, die Bereitstellung von zusammenhängenden Buffern für die Übertragung und die Befüllung der Buffer mit generierten oder abgerufenen Daten.

Die Schwierigkeit hierbei ist die Unterscheidung von physikalischen und virtuellen Adressen. Vom User-Space aus arbeitet man immer mit den virtuellen Adressen. Der CDMA benötigt jedoch die 'echten' Hardware-Adressen.

In der im Folgenden beschriebenen Applikation wird unter Verwendung des `mmap` Befehls Speicher in den User-Space gebracht. Den übergebenen Hardware-Adressen wird somit eine virtuelle Adresse zugewiesen, mit der gearbeitet werden kann.

### 6.2.1. mmap Befehl

Die Datei `/dev/mem` bildet den Arbeitsspeicher des Systems ab. Diese muss zuerst geöffnet werden, um unter Angabe des Filepointers den Speicherbereich in die Datei zu mappen.

Mit dem `mmap` Befehl wird ein neues Mapping im virtuellen Adressraum des aufrufenden Prozesses erzeugt. Die Startadresse für das neue Mapping wird in `addr` angegeben. Das Argument `length` spezifiziert die Länge des Mappings. Mit `offset` gibt man den Offset des Datenbereichs an. Das ist der Bereich, der vom Anfang der Datei ausgelassen werden soll. In diesem Fall muss hier die physikalische Adresse angegeben werden. [14]

```
1 void* mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

Der Systemaufruf `munmap` löscht die Mappings für den angegebenen Adressbereich und bewirkt, dass weitere Referenzen auf Adressen innerhalb des Bereichs ungültige Speicherreferenzen erzeugen. Die Region wird auch automatisch unmapppt, wenn der Prozess beendet wird. [14]

```
1 int munmap(void *addr, size_t length);
```

Für das Bereitstellen des CDMA-Adressraums wurden die Funktionen: `void* openDEV(int *memfd)` und `void closeDEV(int *memfd, void* device)` implementiert.

### 6.2.2. CDMA Programmier-Sequenz

Wenn der Adressbereich des CDMA an eine virtuelle Adresse gemappt ist, kann man seine Register beschreiben. Hierfür wurden am Anfang des Programms diverse Offsets und Bit-Masken definiert. Die Register werden in Xilinx's Produkt Guide 'pg034' unter Tabelle 2-6 beschrieben. [25]

Für die Initialisierung wurde eine Funktion geschrieben, in der ein Reset der Baugruppe ausgeführt wird, die Interrupts aktiviert werden und der Modus auf Simple-Mode gestellt wird.

```
1 void initCDMA(void * device)
```

Um verschiedene Signalformen auf dem DAC ausgeben zu können existieren die Funktionen:

```
1 void ex_triang(int32_t *Array)
2 void ex_restore_sin(int32_t *Array)
3 void ex_clear(int32_t *Array)
```

Sie belegen das übergebene Array mit Werten für ein Dreieck, Sinus oder mit Nullen, um den Speicher zu 'löschen'. Die dadurch später ausgegebenen Signalformen sind in Abbildung 6.6 zu sehen.

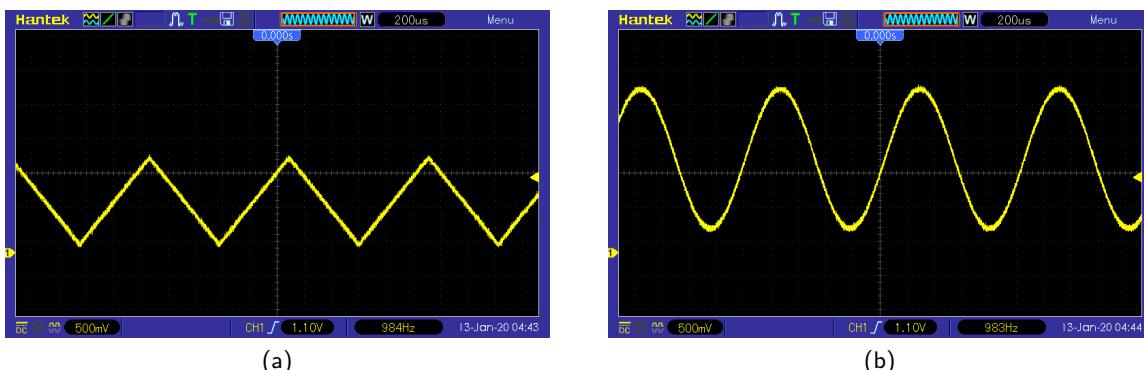


Abbildung 6.6.: Signalformen: (a) 1kHz Dreieck, (b) 1kHz Sinus

Dieses Array wird über die Funktion `copytoDDR` dann in den Speicherbereich im DDR-RAM kopiert. Der Speicherbereich ist der, der im Adresseditor für den HP-Port freigegeben wurde. Er dient im Programm als Source- bzw. Destination-Buffer.

```
1 void copytoDDR(int32_t *Array)
```

Um an diese Stelle kopieren zu können, muss der Speicherbereich natürlich wieder über `mmap` erreichbar gemacht werden. Nach dem `memcpy` wird der Speicher wieder mit `munmap` ausgehängt.

Die Datentransfer-Funktion prüft als Erstes den 'Idle' Zustand des Busses über eine Registerabfrage. Anschließend werden die übergebenen Quell- und Zieladressen gesetzt. Mit dem Schreiben der Transferlänge ins ByteToTransfer Register wird die Transaktion angestoßen.

```
1 void datatransfer(void * device, unsigned long source, unsigned long
destination)
```

Da es sehr schwierig ist einen Interrupt-Händler in einer User-Space Applikation zu implementieren, wird das Statusregister gepollt. Bei einer Änderung wird die entsprechende Interrupt-Flagge ausgewertet und zurückgesetzt.

Der Polling-Modus ist ideal für kleine Testfälle und zum Lernen, aber er ist zu verschwenderisch für praktische Systeme und verschlechtert die gesamte Systemleistung, weil der Prozessor seine Zyklen zum Lesen der DMA-Statusregister verschwendet, wenn er etwas anderes Nützlicheres tun könnte. Dies ist ein Grund für den Einsatz eines Device-Drivers, der im Kernel-Space läuft (Abschnitt 6.3).

Im Main-Programm wird anfangs der CDMA initialisiert. Danach erfolgt die Abfrage, was übertragen werden soll, in einer `while`-Schleife. Nach dem Verlassen der Schleife wird der CDMA-Adressbereich entmappt und `/dev/mem` geschlossen.

### 6.2.3. Reserved-Memory

Für die CDMA-Anwendung wurde über den Device Tree ein kleiner Speicherbereich vom System abgetrennt und reserviert. Dies stellt sicher, dass das Linux System den Bereich nicht anderweitig verwendet und dadurch nicht mit den statischen Adressen aus dem C-Programm in Konflikt tritt. [22] Der Eintrag für benutzerdefinierte Device Tree Einträge muss in das `system-user.dtsi` Device Tree Include File eingetragen werden.

```
1 /<petalinux-projekt-dir>/project-spec/meta-user/recipes-bsp/device-tree/
  files/system-user.dtsi
```

```
1 reserved-memory {
2   #address-cells = <1>;
3   #size-cells = <1>;
4   ranges;
5
6   reserved: buffer@0x20000000 {
7     reg = <0x20000000 0x2000>;
8   };
9 }
```

Die mit den Befehlen `#address-cells` und `#size-cells` gesetzten Werte bestimmen das Adress- und Datenalignment des reservierten Bereiches. Die Zahlen sind mit 32 Bit zu multiplizieren und stellen somit im obigen Kontext 32 Bit dar.

Der Kinder-Knoten „reserved“ legt mit `reg` den genauen Speicherbereich fest, der reserviert wird. Dieser beginnt bei der Adresse `0x20000000` und ist `0x2000` (8 kB) groß und befindet sich im DDR-RAM. Aus Xilinx's UG585 Table 29-1: findet man den DDR Adressbereich heraus.[9, 18, 43]

Um ein rebuild des Device-Trees auszuführen, nutzt man den Befehl:

```
1 $ petalinux-build -c device-tree
```

#### 6.2.4. Einbau ins PetaLinux

Nach der Entwicklung in der SDK wurde die Applikation in das PetaLinux-Projekt eingebaut. Hierzu wird eine neue C-Applikation dem Projekt hinzugefügt. Durch das angehängte '-enable' wird die App auch direkt in der rootfs-Konfiguration aktiviert und dem Filesystem hinzugefügt. [41]

```
1 $ petalinux-create -t apps --template c --name App-Name --enable
```

Nach der Erstellung können die C- und H-Files in den für die Applikation erstellten Ordner unter <plnx-proj-root>/project-spec/meta-user/recipes-apps/App-Name/files kopiert werden. Die Dateien müssen dann noch im darüber liegenden .bb-File eingetragen werden, bevor kompiliert werden kann. Achtung: der Compiler wirft schneller Warnungen und Fehler als die SDK. [36, 38]

```
1 $ petalinux-build -c App-Name
```

Wenn das komplette Projekt wieder neu gebildet von der SD-Karte gestartet wird, kann man das Programm am folgenden Ort finden und auf dem Embedded-Linux ausführen:

```
1 $ cd /user/bin
2 $ ./App-Name
```

#### Applikation entfernen

Wenn man eine Applikation aus dem PetaLinux-Projekt entfernen möchte, muss man in der Datei: <plnx-proj-root>/project-spec/meta-user/recipes-core/images/petalinux-image.bbappend den Verweis auf das Programm entfernen und den zugehörigen Ordner löschen.

## 6.3. Linux Treiber

Wenn man eine Baugruppe betreibt, ist es meistens sinnvoll einen Treiber bereitzustellen, mit dem der allgemeine Zugriff hinter Application Programming Interfaces (APIs) versteckt wird. Das erleichtert dem Benutzer den Umgang mit der Baugruppe. In einem Linux System sind Zugriffe auf Hardware nur aus dem Kernel-Space vorgesehen. Deshalb werden Treiber dort ausgeführt. Die Hauptaufgaben von einem DMA Device Driver sind die DMA-Device Verwaltung und die Speicherreservierung mit Cache-Kontrolle.

Ziel dieses Abschnitts soll sein, die DMA-Treiber-Möglichkeiten aufzuzeigen. Dabei wird besonders darauf eingegangen, wie man Treiber in ein Petalinux-Projekt integriert, da sich dies als große Einstiegshürde herausstellte.

### 6.3.1. Treiberentwicklung

Für die Treiberentwicklung gibt es hilfreiche Präsentationen.

In fünf Teilen wird von „Linux Device Treiber Grundlagen“ bis zur „Verwendung aus dem User-Space“ ein Überblick für die DMA-Treiberentwicklung gegeben. Diese haben jedoch nicht den Anspruch den Leser zur Treiberentwicklung zu befähigen.

1. Introduction to Linux Device Drivers - The Basics [30]
2. Introduction to Linux Device Drivers - Platform and Character Drivers [31]
3. Linux User Space Device Drivers [35]
4. Linux DMA in Device Drivers [34]
5. Linux DMA from User Space [33]

Mit Hilfe der Präsentationen kann der Code des 'dma-proxy'-Projekts (Abschnitt 6.3.3) nachvollzogen werden.

Teil 1 und 2 zeigen die Struktur eines Treibers auf. Dabei werden die Mechanismen von `_init`, `_exit`, `probe()`, `remove()`, `irq()` und der Zusammenhang mit dem Device Tree beschrieben.

Im dritten Teil wird eine Alternative zur normalen Treiberentwicklung im Kernel-Space genauer betrachtet. Mit dem UIO-Driver ist es möglich, vom User-Space aus über Memory-Mapping Geräte zu verwenden. Damit ist es eine ähnliche aber sicherere Variante als die in Abschnitt 6.2 beschriebene Linux-App.

Der vierte und fünfte Teil spezialisiert die Treibergrundlagen von eins und zwei mit der Umsetzung von einem einfachen DMA-Treiber (dma-proxy).

### 6.3.2. Überblick von DMA-Treiberlösungen

Für die Verwendung von DMAs existieren unter Linux verschiedene Treiber-Lösungen, die nicht ausreichend gut dokumentiert erscheinen.

Basis ist meist das DMA Engine Framework (DMA-Engine), welches allgemeinen Support für verschiedenste DMAs bereitstellt. An diese Infrastruktur knüpfen dann spezifischere Device-Driver unter Verwendung der Standard API an.

Xilinx stellt für die DMA-IP-Cores (AXI-DMA, AXI-CDMA, AXI-VDMA) einen Treiber bereit, den 'Xilinx\_dma'. Dieser ist unter [github.com/Xilinx/linux-xlnx/blob/master/drivers/dma/xilinx/xilinx\\_dma.c](https://github.com/Xilinx/linux-xlnx/blob/master/drivers/dma/xilinx/xilinx_dma.c) zu finden. Für einen sinnvollen Zugriff muss auf diesem sehr komplexen Treiber jedoch noch ein Custom-Treiber aufgesetzt werden. Alle Treiber laufen im Kernel-Space und können somit nicht direkt von einer einfachen C-Applikation angesprochen werden.

Im Xilinx-Forum ist die Problematik: „Verwendung von DMAs unter Linux“ ein kontrovers diskutiertes Thema in verschiedenen Threads. Es wird dabei zumeist der AXI-DMA-IP-Core diskutiert, weshalb es auch fast nur zu diesem Lösungsansätzen gibt.

Der vielversprechendste Ansatz ist ein Proxy Treiber, der es möglich macht, DMA-Transaktion aus dem User-Space anzustoßen. Der Grundaufbau wird in Abbildung 6.7 beschrieben. [19]

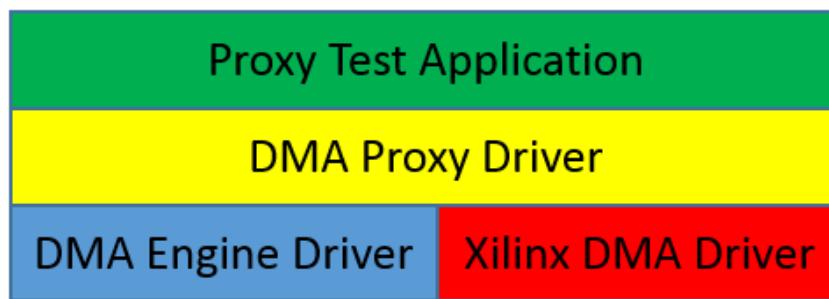


Abbildung 6.7.: DMA-Driver-Stack: grün: Userspace-Application, gelb: Custom-Proxy-Treiber (Kernel-Space), blau/rot: Xilinx/Linux Treiber (Kernel-Space) [19]

Die Treiber sind alle sehr verschachtelt aus vielen Strukturen aufgebaut. Das erschwert das Verständnis. Es wurde versucht den Proxy-Treiber für den CDMA umzuschreiben. Dies gelang auf Grund der fehlenden Dokumentation und Erfahrung an Treiberentwicklung in der verfügbaren Zeit der Bachelorarbeit nicht.

Im Folgenden werden verschiedene Treiber für den AXI-DMA vorgestellt und dabei einige wichtige Mechanismen für die Handhabung von Treibern im PetaLinux verdeutlicht.

Hierfür wurde ein neues PetaLinux Projekt ('AXI-DMA') erstellt, was ein Hardware Design mit einem AXI-DMA in Loopback verwendet. Das Design ist unter Anhang D zu sehen.

### 6.3.3. Treiber SetUp in Petalinux - Am Beispiel bperez77 axidma-driver

Ein Custom-Treiber für den AXI-DMA von bperez77 ([https://github.com/bperez77/xilinx\\_axidma](https://github.com/bperez77/xilinx_axidma)) besitzt eine User-Space-Interface-Library und Testanwendungen. Mit Hilfe von diesem Treiber als Beispiel wird im Folgenden der Einbau in ein PetaLinux-Projekt beschrieben.

## Modul erstellen

Als Erstes erstellt man im PetaLinux-Projekt ein neues Modul für den Custom-Treiber.

```
1 $ petalinux-create -t modules -n axidma-driver --enable
```

Der Befehl legt unter <petalinux-projekt>/project-spec/meta-user/recipes-modules/axidma-driver/files eine C-Datei an, die für die Treiberentwicklung verwendet werden könnte. Diese entfernt man und kopiert alle .c und .h Files des Treibers dort hin.

Das Modul muss in der rootfs-config aktiviert sein, das geschieht durch das '-enable' im Creat-Befehl oder in der Rootfs-Konfiguration selbst.

```
1 $ petalinux-config -c rootfs # "Modules --->" submenu
```

## Makefile und Abhängigkeiten anpassen

Als Nächstes muss das Makefile angepasst werden. In den ersten Zeilen müssen die .o Dateinamen des Treibers angegeben werden.

```
1 DRIVER_NAME = axidma-driver
2 $(DRIVER_NAME)-objs = axi_dma.o axidma_chrdev.o axidma_dma.o axidma_of.o
3 obj-m := $(DRIVER_NAME).o
```

Außerdem müssen alle C- und H-Files in das axidma-driver.bb Dokument im Ordner /axidma-driver/ ergänzt werden.

```
1 SRC_URI = "file://Makefile \
2           file://axi_dma.c \
3           file://axidma_chrdev.c \
4           file://axidma_dma.c \
5           file://axidma_of.c \
6           file://axidma.h \
7           file://axidma_ioctl.h \
8           file://COPYING \
9           "
```

## Kompilieren

Dann kann das Modul gebaut werden, um zu sehen, ob es richtig kompiliert.

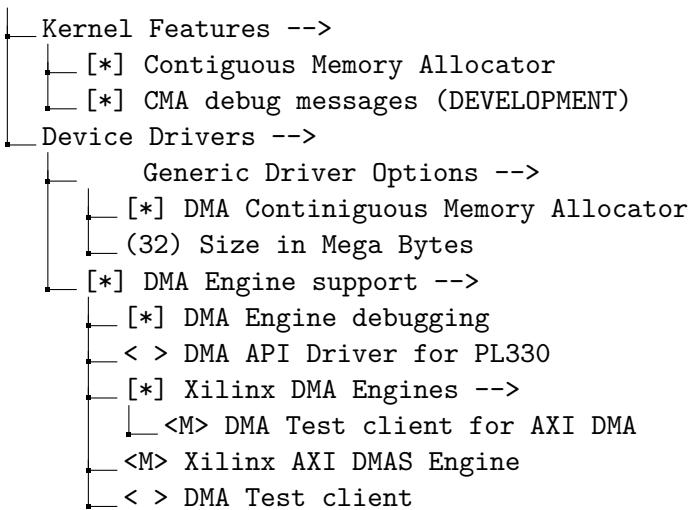
```
1 $ petalinux-build -c axidma-driver
```

## Kernel-Konfiguration

In der Kernel-Konfiguration müssen einige Änderungen vorgenommen werden, um alle Komponenten des Treiber-Stack zu aktivieren. Verschiedene Treiber müssen aktiviert sein, andere (der PL330) müssen herausgenommen werden. Das meiste wird als Modul hinzugefügt, um im System verschiedene Konstellationen testen zu können. Da der Custom Treiber auf dem 'Xilinx\_dma' basiert, wird dieser aktiviert. Der Xilinx-Treiber wird dann automatisch über PetaLinux von GIT gezogen und verwaltet.

```
1 $ petalinux-config -c kernel # Treiber aktivierung
```

Die dazu notwendige Konfiguration wird im folgenden Baumdiagramm dargestellt.



Der Contiguous Memory Allocator (CMA) wird vom Treiber für die Buffer-Bereitstellung genutzt. Die Größe muss angepasst werden, damit auch Übertragungen im Megabyte Bereich möglich sind. Die DMA Engine ist Basis aller Treiber. Damit es nicht zu Konflikten mit dem DMA des PS kommt (DMAC-PL330), muss dieser deaktiviert werden. Die Xilinx AXI DMAS Engine ist der Device Treiber für den PL-Baustein, der vom Custom Treiber genutzt wird.

## Device Tree Einträge

Im Device Tree müssen die entsprechenden Einträge für den Treiber stehen. Da der Tree automatisch aus dem HDF generiert wird und dabei nicht alle Einträge zum Treiber passen, müssen diese abgeändert werden.

Die generierten Device Tree Dateien findet man im unten aufgeführten Ordner. Der für den Treiber interessante Eintrag steht in der 'pl.dtsi', in der die PL-Block-Informationen zu finden sind.

```
1 <plnx-proj-root>/components/plnx_workspace/device-tree/
  device-tree-generation
```

Für benutzerdefinierte Änderungen gibt es das 'system-user.dtsi' Dokument.

```
1 /plnx-proj-root>/project-spec/meta-user/recipes-bsp/device-tree/files/
  system-user.dtsi
```

Die Anpassungen des Files sind in Code 6.1 zu sehen. Der erste Eintrag ist speziell für den Treiber. Der zweite korrigiert die 'device-id' des Stream to Memory Mapped (S2MM)-Channels.

Code 6.1: system-user.dtsi

```
1 /include/ "system-conf.dtsi"
2 /
3 /* Eintrag für den bperez77 Treiber */
4     axidma_chrdev: axidma_chrdev@0 {
5         compatible = "xlnx,axidma-chrdev";
6         dmas = <&axi_dma_0 0 &axi_dma_0 1>;
7         dma-names = "tx_channel", "rx_channel";
8     };
9 /* Ueberschreibung des pl.dtsi Eintrag AXI-DMA mit richtiger ID */
10    amba_pl: amba_pl {
11        #address-cells = <1>;
12        #size-cells = <1>;
13        compatible = "simple-bus";
14        ranges ;
15        axi_dma_0: dma@40400000 {
16            #dma-cells = <1>;
17            clock-names = "s_axi_lite_aclk", "m_axi_sg_aclk", "m_axi_mm2s_aclk", "m_axi_s2mm_aclk";
18            clocks = <&clkc 15>, <&clkc 15>, <&clkc 15>, <&clkc 15>;
19            compatible = "xlnx,axi-dma-1.00.a";
20            interrupt-parent = <&intc>;
21            interrupts = <0 29 4 0 30 4>;
22            reg = <0x40400000 0x10000>;
23            xlnx,addrwidth = <0x20>;
24            xlnx,include-sg ;
25            dma-channel@40400000 {
26                compatible = "xlnx,axi-dma-mm2s-channel";
27                dma-channels = <0x1>;
28                interrupts = <0 29 4>;
29                xlnx,datawidth = <0x20>;
30                xlnx,device-id = <0x0>;
31            };
32            dma-channel@40400030 {
33                compatible = "xlnx,axi-dma-s2mm-channel";
34                dma-channels = <0x1>;
35                interrupts = <0 30 4>;
36                xlnx,datawidth = <0x20>;
37                xlnx,device-id = <0x1>;
38            };
39        };
40    };
41 };
```

## Projekt Bauen

Zum Schluss muss das ganze Projekt gebaut und das Boot Image aktualisiert werden. Dann kann das Projekt wieder auf die SD-Karte kopiert werden.

```
1 $ petalinux-build
2 $ petalinux-package --boot --force --fsbl ./images/linux/zynq_fsbl.elf
   --fpga ./images/linux/microzed_axi_dma_wrapper.bit --u-boot
```

Im PetaLinux-Projekt wird das gebaute Kernel Modul im Ordner unter <plnx-proj-root>/build/tmp/sysroots/plnx\_arm/lib/modules/<KernelVersion>/extra/axidma-driver.ko abgelegt. Das Modul wird dadurch auch gleichzeitig mit ins Filesystem gepackt. Im Embedded-Linux findet man das Kernel Objekt File dann unter /lib/modules/<KernelVersion>/extra/axidma-driver.ko.

## Module entfernen

Um ein Modul aus einem Projekt zu entfernen, muss der zugehörige Ordner gelöscht werden und der Eintrag IMAGE\_INSTALL\_append= "mymodule" aus <plnx-proj-root>/project-spec/meta-user/recipes-core/images/petalinux-image.bb entfernt werden. Danach führt man noch einmal die rootfs-config aus.

```
1 $ petalinux-config -c rootfs
```

## Einbau des dma-proxy Treibers

Analog zum vorigen Abschnitt wurde der 'dma-proxy' Treiber dem PetaLinux hinzugefügt. Der zugehörige Device Tree Eintrag muss auch dem 'system-user.dtsi' File hinzugefügt werden.

```
1 /* Eintrag für den Proxy Treiber */
2 dma_proxy {
3     compatible = "xlnx,dma_proxy";
4     dmas = <&axi_dma_0 0 &axi_dma_0 1>;
5     dma-names = "dma_proxy_tx", "dma_proxy_rx";
6 };
```

Im Makefile muss noch das Linker-Flag '-lpthread' ergänzt werden. Alle anderen Schritte sind identisch oder einfacher.

### 6.3.4. Verwendung von Treibern im Embedded-Linux

#### xilinx\_dma

Beim Start des Linux-Systems sieht man schon die Meldung, dass der 'Xilinx\_dma' aktiviert wurde.

```
1 xilinx-vdma 40400000.dma: Xilinx AXI DMA Engine Driver Probed!!
```

Wenn der 'Xilinx\_dma' Treiber aktiviert ist, kann man sich die zur Verfügung stehenden Channel anzeigen lassen. [21]

```
1 $ ls -1 /sys/class/dma/
```

### **axidmatest**

Durch die Kernel-Konfiguration wurde der Test-Client als Modul hinzugefügt. Dieses Kernel-Modul kann zur Verifikation, dass der Xilinx-Treiber funktioniert und die richtige Hardware findet, ausgeführt werden.

```
1 $ cd /lib/modules/4.9.0-xilinx-v2017.4/kernel/drivers/dma/xilinx/
2 $ modprobe axidmatest.ko
```

Der Test sollte wie folgt antworten:

```
1 dmatest: Started 1 threads using dma0chan0 dma0chan1
2 dma0chan0-dma0c: terminating after 5 tests, 0 failures (status 0)
```

Durch die aktivierten Debug-Meldungen kann man sich die Kernel-Ausgaben anzeigen lassen.

```
1 $ dmesg -c      # löscht den Kernel Ring Buffer zur Übersichtlichkeit
2 $ dmesg        # Ausgabe
```

Der Test sollte danach wieder entfernt werden um die Channel freizugeben.

```
1 $ rmmod axidmatest.ko
```

Durch den erfolgreichen Test kann nun auch einer der Custom-Treiber aktiviert werden. Wichtig ist, dass immer nur einer gleichzeitig laufen kann, da beide die gleichen Ressourcen benötigen.

### **dma-proxy**

```
1 $ cd /lib/modules/4.9.0-xilinx-v2017.4/extr/
2 $ insmod dma-proxy.ko
```

Der Treiber sollte Memory für TX- und RX-Buffer reservieren.

### **axidma-driver**

```
1 $ cd /lib/modules/4.9.0-xilinx-v2017.4/extr/
2 $ insmod axidma-driver.ko
```

Der Treiber sollte einen Transmit-Channel und einen Receive-Channel für den AXI-DMA finden.

### 6.3.5. Beispiel Anwendungen

Die Funktion kann mit verschiedenen Applikationen veranschaulicht werden. Die Apps sind auch in das PetaLinux-Projekt eingebaut und können von dort oder extern über das entsprechende SDK-Projekt ausgeführt werden. Entweder man aktiviert den 'axidma-driver' und nutzt axidma\_benchmark, axidma\_transfer, oder der 'dma-proxy' muss aktiviert sein für das Testen der dma-proxy-test Applikation.

#### **axidma\_benchmark - axidma-driver**

Der Benchmark Test führt mehrere Übertragungen aus und prüft das empfangene Pattern gegen das gesendete. Dabei wird die Übertragungsrate durch Zeitmessung ermittelt.

```
1 $ cd /usr/bin/
2 $ ./axidma-benchmark
```

Es wird eine Geschwindigkeit von ca. 275.64 MiB/s in eine Richtung erreicht.

#### **axidma\_transfer - axidma-driver**

Die Transfer-Applikation überträgt den Inhalt einer Datei über das Loopback Design. Die Quell- und Ziel-Datei müssen als Parameter angegeben werden.

```
1 $ cd /usr/bin/
2 $ ./axidma-transfer /home/root/input.txt /home/root/output.txt
```

Für Testzwecke mit verschiedenen Größen kann auf dem Ubuntu-System einfach eine zufällige Datei mit einer festen Größe erzeugt werden. Diese kann man dann über die FTP-Verbindung auf das Embedded-Linux übertragen.

```
$ base64 /dev/urandom | head -c 1000000 > file.txt #für 1MB große Datei
```

Für eine Übertragung mit einer Datei von 90 Byte wurde eine ILA-Aufzeichnung vorgenommen. Bei der Analyse erkennt man die Designschwächen. Es wird z.B. nur ein HP-Port verwendet und die Datenbreiten sind nur auf 32 Bit konfiguriert. Außerdem ist zu erkennen, dass durch die Verwendung von Treibern nicht unbedingt eine größere Übertragungsrate erreicht wird. Die 90 Byte benötigen für das Loopback ca. 800 Takte. Trotzdem ist die Verwendung von Treibern empfohlen, da das Memorymanagement, das Reservieren von Buffern und die Kommunikation zum Kernel-Space viel sauberer und sicherer ablaufen.

Diese Anwendung kann auch mit einem entsprechenden Hardware-Design zu einem Accelerator-Tester werden. Anstelle des Data-FIFO kann z.B. ein FFT-Block eingebaut werden. Dieser wird dann mit der Input-Datei gefüttert und gibt sein Ergebnis in die Output-Datei aus.

### dma-proxy-test - dma-proxy

Der Test führt eine Übertragung unter Verwendung des Proxy-Treibers durch. Dabei wird mit verschiedenen Threads gearbeitet.

```
1 $ cd /usr/bin/
2 $ ./axidma-transfer 1 512 # Arg1: number of transactions Arg2: test_size
```

### 6.3.6. User-Space minimal Beispiel - Lauri

Für das Verständnis des AXI-DMA gibt es auch wieder eine einfache Anwendung - nur aus dem User-Space. Dabei wird wieder über /dev/mem auf die Hardware zugegriffen und ein Memory Copy ausgeführt.

Dieses Beispiel benötigt wiederum ein neues Hardware-Design, in dem der SG-Modus deaktiviert ist. Außerdem darf im Embedded-Linux kein Treiber aktiviert sein. Es kann jedoch das AXI-DMA-Projekt verwendet werden. Dabei muss dann nur der 'xilinx-axidma' Treiber deaktiviert werden. Danach muss noch das Bit-File vom modifizierten Hardware-Design geladen werden.

Um den AXI-DMA Block zu verwenden müssen, vier minimale Schritte ausgeführt werden: [13]

- 1. Starten der DMA Kanäle (Memory Mapped to Stream (MM2S), S2MM oder beide), indem eine 1 ins Control Register geschrieben wird.
- 2. Die Start-/Ziel-Adressen müssen in die entsprechenden Register geschrieben werden.
- 3. Die Transferlänge muss in das entsprechende Register eingetragen werden, um die Übertragung zu starten.
- 4. Das Status Register muss auf die Inerrupt-flags überwacht werden.

Eine ausführliche Anleitung ist unter <https://lauri.vosandi.com/hdl/zynq/xilinx-dma.html> zu finden.

```
1 $ rmmod /lib/modules/4.9.0-xilinx-v2017.4/kernel/drivers/dma/xilinx/
      xilinx_dna.ko
2 $ cat /home/root/AXI-DMA-Lauri.bit > /dev/xdevcfg
3 $ cd /usr/bin/
4 $ ./Memcopy-Lauri
```

## Befehl Übersicht

Die folgende Übersicht beinhaltet einige Befehle für den Gebrauch von Treibern auf dem Embedded-Linux.

```
1 $ modprobe      # läd automatisch aus /lib/modules/*
2 $ insmod </path-to-module/module.ko> # laden
3 $ rmmod </path-to-module/module.ko>  # entfernen
4 $ dmesg | grep -i schlüsselwort    # Ausgabe Kernel-Massages
5 $ dmesg -c           # Ausgabebuffer clear
6 $ cat Bitstreamfile.bit > /dev/xdevcfg
```

# 7. Analyse der Übertragung

Die Übertragung eines Schreibprozesses über den CDMA wird im Folgenden mit Hilfe von ILA-Aufzeichnungen genauer analysiert. Der ILA ist mit den folgenden Leitungen verbunden:

- CDMA-Master-Port als Slot-0
- Anschluss des HP-Ports als Slot-1
- Anschluss des BRAM als Slot-2
- CDMA-Controle-Port als Slot-3

Der Trigger wird über den CDMA-Controle-Port (Slot-3) 'ARVALID' oder 'AWVALID' aktiviert und zeichnet ab Takt 2 auf. Es werden dann 1024 Samples aufgenommen.

Für die einfachere Erklärung wird die Übertragung in eine Controle-Phase (Abschnitt 7.1) und die Datenübertragungs-Phase (Abschnitt 7.2) aufgeteilt. Der Überblick über die ganze Übertragung ist in Abbildung 7.1 gezeigt.

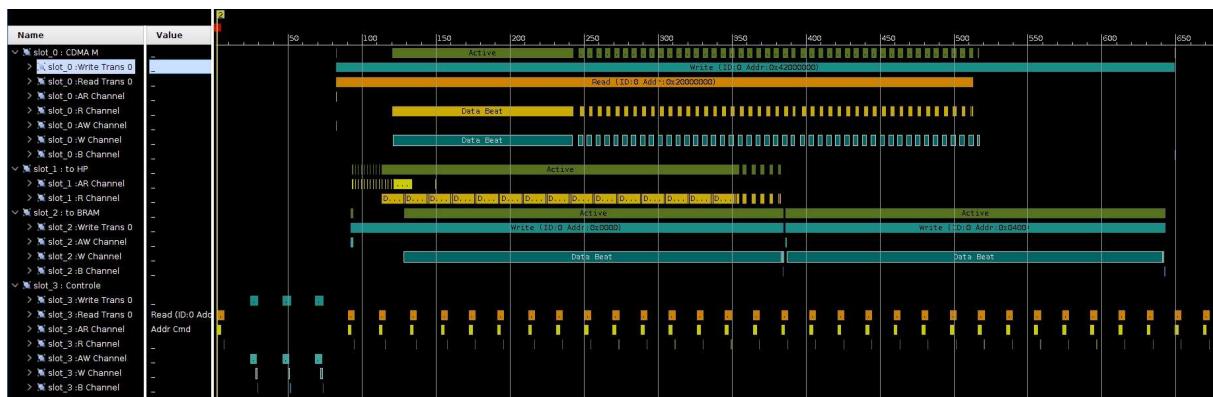


Abbildung 7.1.: Schreiben auf BRAM - AXI-Übertragung Übersicht

## 7.1. Controle-Phase

Der Ablauf zeigt die Ausführung der Programmsequenz der `datentransfer()`-Funktion aus Abschnitt 6.2 auf dem Bus.

In der Controle-Phase werden die Register des CDMA über die AXI-Lite-Schnittstelle (Slot-3) beschrieben (Abbildung 7.2) und überprüft (Abbildung 7.3).

1. Das C-Programm liest zuerst das Statusregister (Offset 0x04), ob der Bus idle (0x02) ist. (Takte 2-7)
2. Danach wird das Source-Register (Offset 0x10) mit der Quell-Adresse beschrieben (DDR: 0x20000000). (Takte 24-30)

3. Ebenso folgt das Beschreiben des Destination-Registers (Offset 0x20) mit der Ziel-Adresse (BRAM: 0x42000000). (Takte 46-52)
4. Zuletzt wird noch das Bytes-to-transfer-Register (Offset 0x28) gesetzt (2048 Byte = 0x800) und damit die Übertragung des CDMA gestartet. (Takte 68-74)
5. In Takt 82 sieht man schon den Anfang des CDMA-Transfers.
6. Danach fängt das C-Programm an, das Statusregister auf die Interrupt-Flags zu überprüfen (Abbildung 7.3).
7. Die Übertragung ist beendet, sobald im Statusregister der Completion-Interrupt (0x1000) zu lesen ist, gleichzeitig ist der Bus auch wieder idle (0x02). (Takt 673)

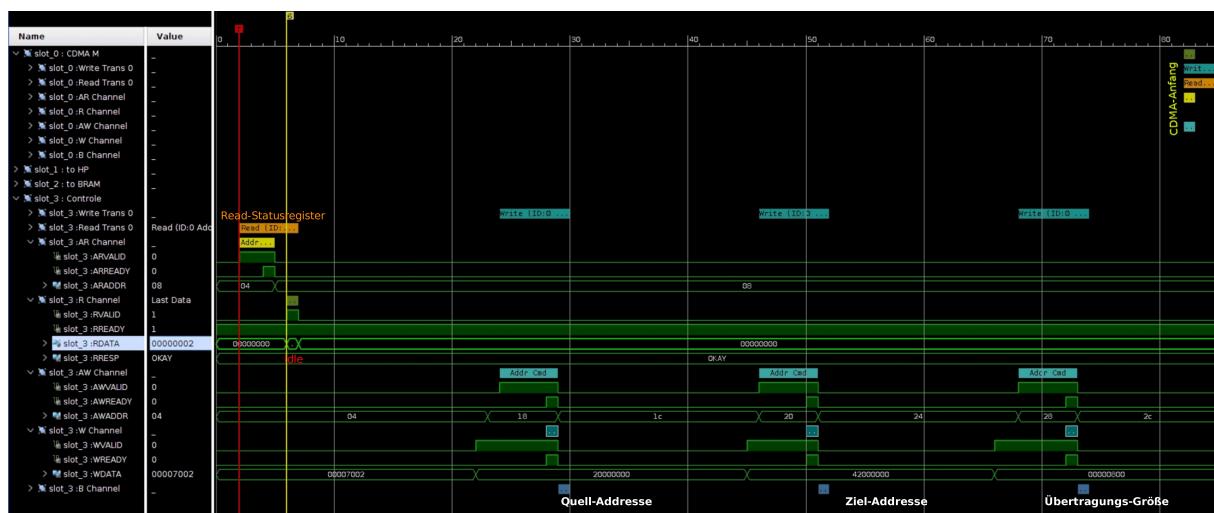


Abbildung 7.2.: Schreiben auf BRAM - Controle-Phase

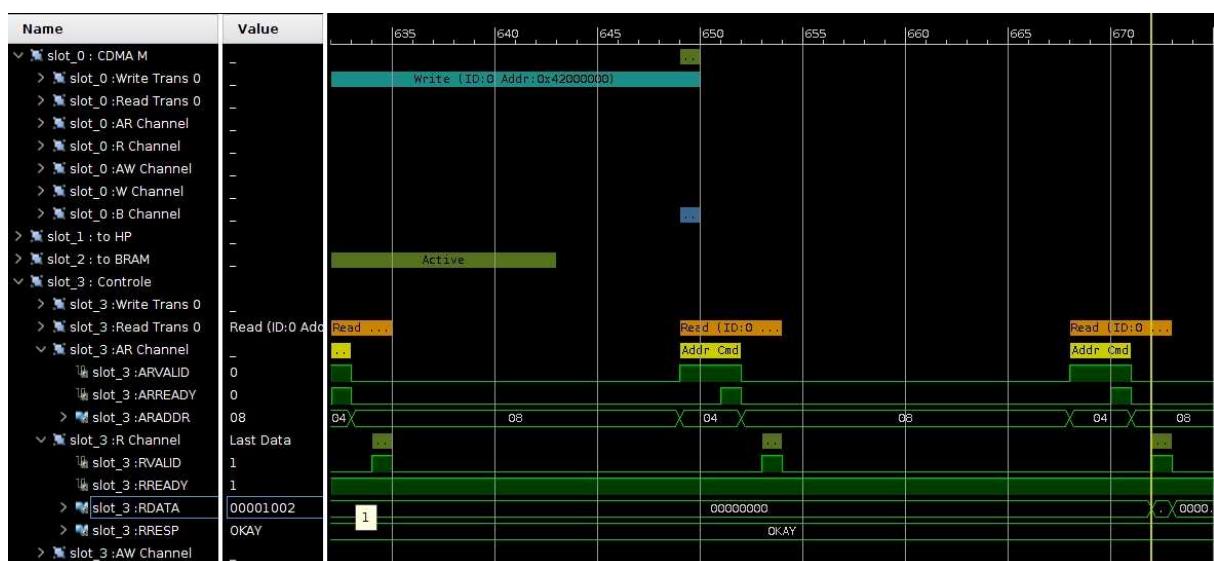


Abbildung 7.3.: Schreiben auf BRAM - Controle-Phase Read Statusregister

## 7.2. Datenübertragungs-Phase

In Takt 82 sendet der CDMA seinen Read- und Write-Adresskanälen die Informationen für die Memory-to-Memory Übertragung (Abbildung 7.4). Aus Axlen = 0xFF = 255 und Axsiz (8 Byte) ergibt sich die gesamte Datenmenge (2048 Byte).

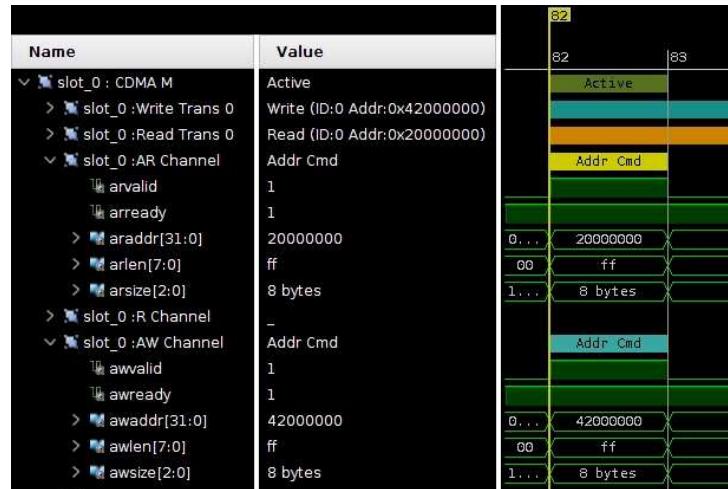


Abbildung 7.4.: Schreiben auf BRAM - CDMA-Befehl

In Abbildung 7.5 werden die Daten-Beats aufgetragen. Auch wenn in dieser Abbildung sehr viel sehr klein dargestellt ist, kann man erkennen, wie die Kanäle unterschiedlich schnell sind und aufeinander warten müssen.

Slot-1, der HP-Port arbeitet mit einer Burst-Länge von 16. Es finden auch 16 Daten-Beats statt. Insgesamt werden somit die 2048 Byte in 16 Beats mit jeweils  $16 * 8$  Byte gelesen. Der HP-Kanal ist damit schon ab Takt 383 wieder frei.

Der CDMA-Master Port (Slot-0) überwacht die ganze Übertragung. Etwas versetzt zum HP-Port erkennt man den Read- und Write-Data-Beat. Dort ist auch gut zu erkennen, dass die Beats durch fehlende (auf 'low' gehende) wready- und rready-Signale unterbrochen werden.

Das Schreiben auf den BRAM (Slot-2) dauert auf Grund der halben Datenbreite (32 Bit) doppelt so lange. Dafür ist hier die Burst-Länge mit 256 am größten und die Übertragung muss nur einmal für einen zweiten Adress-Befehl unterbrochen werden. Dieser Bruch ist in Takt 387 zu sehen, ab dem die zweite Hälfte des BRAM ab Adresse 1024 = 0x400 beschrieben wird.

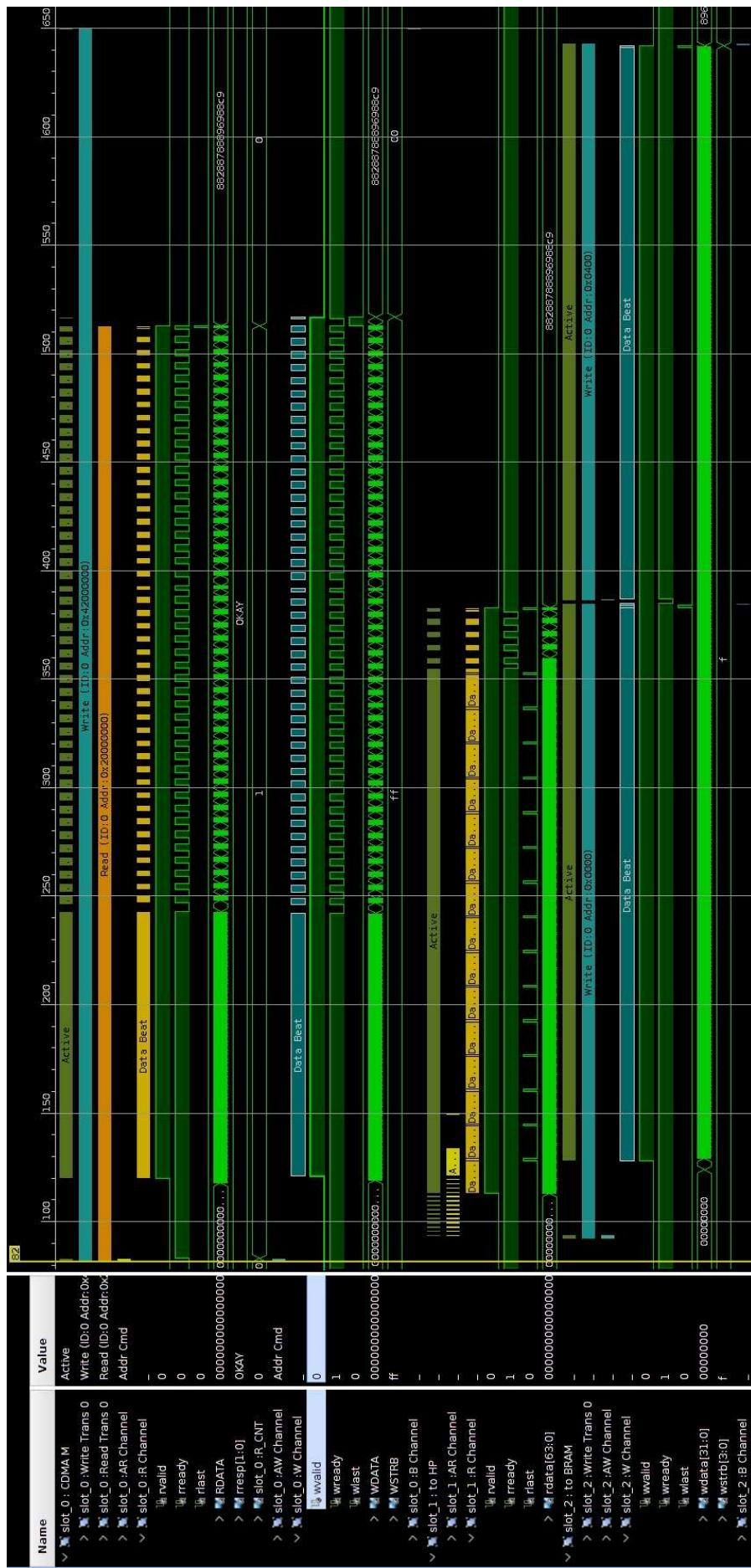


Abbildung 7.5.: Schreiben auf BRAM - Daten

# 8. Analyse der Performance

Um eine Aussage über die Performance zu treffen, wurde die Übertragungsrate mit Hilfe der benötigten Takte und der übertragenen Datenmenge bestimmt.

## 8.1. Was beeinflusst die Performance

Die Haupteinflussfaktoren auf die absolute Datenübertragungsrate bilden der PL-Takt und die Datenbreite. Denn wenn die Übertragung über den DMA angestoßen ist, wird pro Takt ein Datenpaket übertragen.

### 8.1.1. Datenbreite

Der Datenfluss wird mit den entsprechenden Datenkanal-Breiten in Abbildung 8.1 zur Übersicht nochmals dargestellt. Das Bottleneck bildet im verwendeten Design der BRAM mit einer Datenbreite von nur 32 Bit. Der HP-Port bietet eine Datenbreite von 64 Bit, was somit auch die maximale sinnvolle Breite für eine Übertragung zwischen PS und PL darstellt. Die Datenbreite ist generell immer eine Entscheidung zwischen zeitlicher Bus-Auslastung und Platzverbrauch.

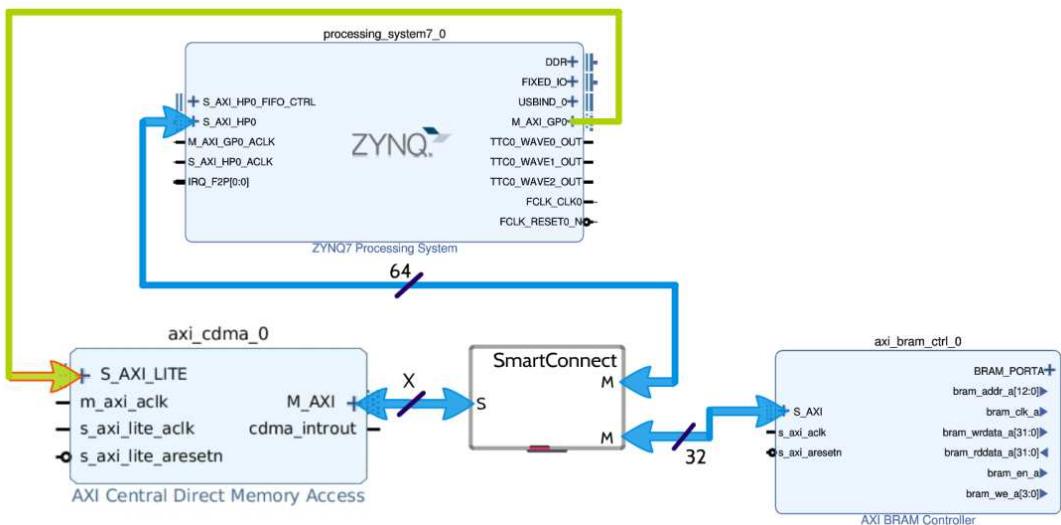


Abbildung 8.1.: Datenbreite im CDMA-Design

Die unterschiedlichen Datenbreiten werden vom SmartConnect richtig verbunden, ohne dass der Nutzer etwas davon mitbekommt. [39]

Wie in der Übertragung in Abbildung 8.2 zu sehen ist, werden der HP- (Slot-1) und BRAM-Kanal (Slot-2) unterschiedlich schnell wieder frei auf Grund der unterschiedlichen Kanal-Breiten. Daten müssen deshalb im Verbindungsblock zwischen-gepuffert werden. Hierfür benötigt man mehr Flip-Flops. Um diese Ressourcen zu sparen, sollten die Datenbreiten im Design sinnvoll zusammenpassend gewählt werden.

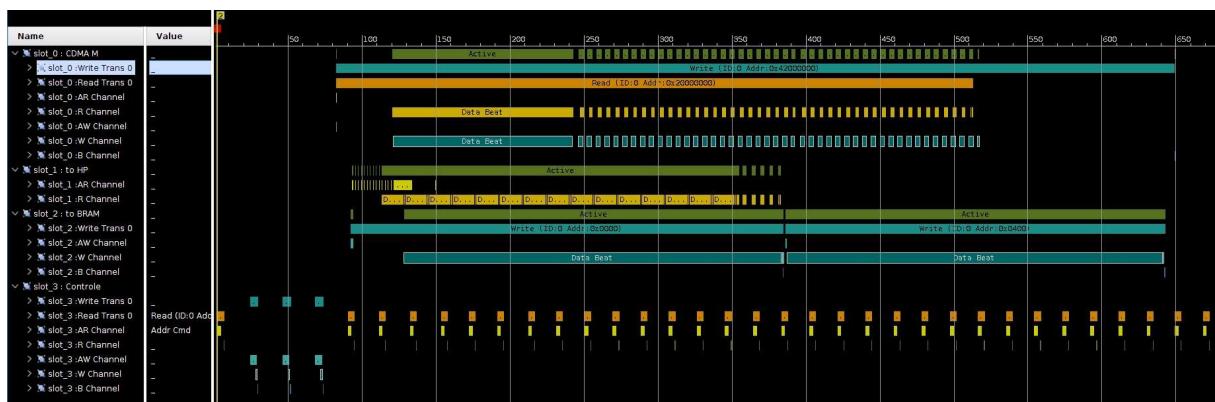


Abbildung 8.2.: Schreiben auf BRAM AXI-Übertragung Übersicht

In der IP-Konfiguration des CDMA-Blocks können die „Write/Read Data Width“ und die „Write/Read Burst Size“ ausgewählt werden. Die Datenbreite spielt hier keine besondere Rolle, solange man sie nicht kleiner als die Quell- und Ziel-Breiten auswählt.

### 8.1.2. Burst Size

Die Burst Size gibt an, wie viele Datenpakete in einer Transaktion übertragen werden. Das Maximum ist durch das AXI4-Protokoll begrenzt auf 256. Je höher die Burst Size, desto besser ist der Durchsatz. Wenn die Burst Size klein gewählt wird, entstehen mehr Controle-Phasen und der Daten-Kanal ist ein paar Takte nicht genutzt.

## 8.2. Übertragungsrate

Xilinx gibt für den HP-Port bei einer Frequenz von 150 MHz eine Bandbreite von 1200 MB/s an (UG585, Tab 22-2). Diese ergibt sich aus der Datenbreite (64 Bit = 8 Byte), die in einem Takt übertragen werden kann. [43]

$$\frac{\text{Datenmenge}}{\frac{1}{\text{PL-Clock}}} = \frac{8 \text{ B}}{\frac{1}{150 \text{ MHz}}} \approx 1200 \text{ MB/s}$$

Mit 100 MHz können also maximal 800 MB/s über den HP-Port übertragen werden.

Da für die gesamte Übertragung von Start bis Ziel der BRAM mit nur 32 Bit Breite das Bottleneck bildet, kann die maximale Übertragungsgeschwindigkeit nur bei 400 MB/s liegen. Hierzu kommt dann noch der Overhead der Übertragung. Besonders ausschlaggebend sind hier die langsame Registerbeschreibung für die Konfiguration des CDMA und die (jedoch viel kleinere) Unterbrechung zwischen den Bursts.

Für die Design-Parameter:

- PL-Clock: 100 MHz
- CDMA-Burst Size: 256
- CDMA-Data Width: 64
- HP-Data Width: 64 (fix)
- mem\_top-Data Width: 32 (fix)

und einer Übertragung von 2048 Byte ergibt sich dann eine Übertragungsrate von

$$\text{Datenmenge} / \frac{\text{benötigte Takte}}{\text{PL-Clock}} = 2048 \text{ Byte} / \frac{671}{100 \text{ MHz}} \approx 305 \text{ MB/s}$$

In der Anzahl der Takte wird der ganze Overhead berücksichtigt - von der ersten Controle-Übertragung (Trigger bei Takt 2) auf dem AXI-Lite-Port bis das Statusregister mit der Interrupt-Flag gelesen wurde (Takt 672).

Die Ausgangssituation, die verbessert werden sollte, erreichte eine Datenübertragungsrate von etwa 4,8 MB/s. Dabei musste für jeden Speicherzugriff jede einzelne Adresse vom Prozessor gelesen/geschrieben werden. Dies erfolgte durch eine For-Schleife mit inkrementierender Adresse (`firdemo_ilा.c`). [8]

Der Performancegewinn von 4,8 MB/s auf 305 MB/s stellt das erreichte Ziel der Bachelorarbeit dar. Die Aufgabe wurde durch das Design mittels CDMA und zugehöriger C-Applikation gelöst.

## 9. Fazit

Diese Bachelorarbeit hat sich mit der Datenübertragung zwischen Processing System (PS) und Programmable Logic (PL) beschäftigt. Hierbei wurde die Zynq Architektur auf ihre internen Datenaustauschmöglichkeiten untersucht. Die Analyse führte zu dem Ergebnis, dass das High Performance (HP)-Interface für den gegebenen Rahmen die beste Lösung darstellt. Aufgrund der möglichen Datenbreite von 64 Bit und der internen Architektur sind mit den HP-Ports performante Übertragungen möglich.

Bei der Auswahl der Datentransfer-Management-Einheit überzeugten die Direct Memory Access (DMA) Blöcke AXI-DMA für Streaming Daten und AXI-Central Direct Memory Access (CDMA) für memory-mapped Zugriffe.

Unter Verwendung vom CDMA entstand ein Design mit dem Custom-Block 'fir\_memo\_top' für Übertragungen zwischen dem Hauptspeicher und dem Block-RAM (BRAM) im 'memo\_top'.

Zur Steuerung der Übertragungen wurde auf einem Embedded-Linux eine Applikation unter Verwendung von `mmap` entwickelt. Die hierfür benötigte Erstellung eines PetaLinux wurde genauer erklärt, sodass der Einstieg für weitere Projekte erleichtert wird.

Mit der Applikation wird eine Performance von 305 MB/s erreicht. Dies ist eine deutliche Verbesserung zur Ausgangslage von nur etwa 4,8 MB/s. Höhere Geschwindigkeiten können durch ein Design optimiert auf die 64 Bit Datenbreite erreicht werden.

Durch anschließende Analysen der Daten-Transfers auf dem AXI-Bus und vorherige Grundlagenarbeit wurde ein Einblick in die Funktionsweise des AXI-Busses gegeben.

Während der Durchführung stellte sich heraus, dass für saubere, native Hardwarezugriffe aus einem Embedded-Linux Device-Treiber benötigt werden. Speziell ausgerichtet auf den AXI-DMA wurden bestehende Treiber-Lösungen herausgesucht. Es folgte eine genauere Beschreibung des Einbaus und der Verwendung der Treiber in einem PetaLinux-Projekt.

Zusätzlich zur schriftlichen Dokumentation der Entscheidungen, Erkenntnisse und Anleitungen entstand ein umfassendes Paket an Projektverzeichnissen und anderen Materialien, die einen Großteil der Arbeit darstellen.

Das Thema der OnChip-Kommunikation zwischen verschiedenen Komponenten wird auch in Zukunft eine sehr wichtige Thematik darstellen. Durch die untersuchten Beispiele in der Bachelorarbeit wurde bewiesen, dass es viele Möglichkeiten und Potenzial auf diesem Gebiet gibt. Außerdem wurde aufgezeigt, dass auch individuell entwickelte Blöcke mit einer standardisierten Technik gut eingebunden werden können. Es bleibt interessant, was auf dem Gebiet der System on Chip (SoC)-Entwicklung passiert.

# 10. Ausblick

In der Arbeit fanden schon einige Techniken Erwähnung, die für zukünftige Recherchen und Projekte interessant sein können.

Für eine noch engere Zusammenarbeit von Prozessor und Logik ist Cache-Kohärenz notwendig, damit beide Einheiten mit den gleichen konsistenten Daten arbeiten. Der Accelerator Coherency Port (ACP) bietet diese Möglichkeit, indem er über die Snoop Control Unit die PL-Seite direkt mit dem L1 und L2 Cache des Prozessors verbindet. Für die Verwendung wird auf PS- sowie PL-Seite tieferes Wissen aus dem Memory-Management benötigt.

Eine weitere Technik aus der DMA-Verwaltung ist der Scatter/Gather (SG)-Modus. Hierbei werden die Transaktionen mittels Deskriptoren verwaltet. Deskriptoren können für zyklische Übertragungen fest auf einem Block-RAM abgelegt werden. Dadurch können DMA-Blöcke performanter auf die Informationen zugreifen. Außerdem können zu übertragende Daten fragmentiert vorliegen und über Deskriptoren zu einem zusammenhängenden Stream zusammengefügt werden. Die Deskriptor-Verwaltung stellt hier das genauer zu untersuchende Thema dar.

Bei der Auswahl der DMA-Blöcke wurde schon der AXI-DMA-Block favorisiert. Für die Verwendung des Blocks müssen jedoch Funktionsblöcke mit der AXI-Stream Schnittstelle entworfen werden. Das Design solcher Logik-Blöcke ist sehr interessant und bietet viel Potenzial für weiterführende Arbeiten.

Im Bereich Treiberentwicklung bleibt der Wunsch nach einem einfachen Zugriff aus dem User-Space auf die Hardware. Mit dem Proxy-Treiber ist hier ein Ansatz gegeben. Da dieser jedoch nicht den Anspruch an einen performanten Treiber hat, steckt hier noch viel Verbesserungspotenzial für einen Treiberentwickler drin. Die ganze DMA-Thematik versteckt hinter einer einfachen und dokumentierten API eines Treibers wäre ein riesiger Gewinn für das Hardware-Software-Co-Design.

# Literaturverzeichnis

- [1] Adam Taylor, *The MicroZed Chronicles - Using the Zynq 101: Complete First Year*, Englisch, First Edition. CreateSpace Independent Publishing Platform, Aug. 2015, ISBN: 978-1-5153-3288-6.
- [2] ARM Limited, *PrimeCell DMA Controller (PL330) Technical Reference Manual*. Adresse: [http://infocenter.arm.com/help/topic/com.arm.doc.ddi0424a/DDI0424A\\_dmac\\_pl330\\_r0p0\\_trm.pdf](http://infocenter.arm.com/help/topic/com.arm.doc.ddi0424a/DDI0424A_dmac_pl330_r0p0_trm.pdf) (besucht am 27.11.2019).
- [3] Avnet Electronics Marketing, *SpeedWay Design Workshops™ - Implementing Linux on the Zynq®-7000*. Adresse: <http://microzed.org/course/implementing-linux-zynq%C2%AE-7000-all-programmable-soc-ise-142> (besucht am 05.11.2019).
- [4] Avnet Reach Further, *MicroZed / Zedboard*. Adresse: <http://zedboard.org/product/microzed> (besucht am 14.12.2019).
- [5] Avnet Reach Further, *SpeedWay Workshop - Integrating Sensors on MiniZed with PetaLinux 2017.4*. Adresse: <http://microzed.org/course/integrating-sensors-minized-petalinux-20171-and-20174?sid=167866> (besucht am 05.11.2019).
- [6] Claudio Avi Chami, *Xilinx AXI Stream tutorial - Part 1*. Adresse: <http://fpgasite.blogspot.com/2017/07/xilinx-axi-stream-tutorial-part-1.html> (besucht am 02.01.2020).
- [7] Dieter Kohlert, *fir\_memo\_demo Block-Design - fir\_memo\_demo.odg*.
- [8] Dieter Kohlert, *Hard/Softwaredesign auf MicroZed-Board mit XILINX-ZYNQ-7020-Baustein - ZYNQ Demo 2018 Präsentation*.
- [9] eLinux.org, *Device Tree Usage - eLinux.org*. Adresse: [https://elinux.org/Device\\_Tree\\_Usage](https://elinux.org/Device_Tree_Usage) (besucht am 23.01.2020).
- [10] Gisselquist Technology's, *Understanding AXI Addressing*. Adresse: <https://zipcpu.com/blog/2019/04/27/axi-addr.html> (besucht am 02.01.2020).
- [11] Golem.de, *Programmable Acceleration Card: Intel bringt FPGA-Beschleuniger mit DDR4-Slots* - Golem.de, de-DE. Adresse: <https://www.golem.de/news/programmable-acceleration-card-intel-bringt-fpga-beschleuniger-mit-ddr4-slots-1809-136818-all.html> (besucht am 31.01.2020).
- [12] Instructables, *Getting Started With PetaLinux*, en. Adresse: <https://www.instructables.com/id/Getting-Started-With-PetaLinux/> (besucht am 04.12.2019).
- [13] Lauri Vosandi, *Lauri's blog / AXI Direct Memory Access*. Adresse: <https://lauri.xn--vsandi-pxa.com/hdl/zynq/xilinx-dma.html> (besucht am 23.01.2020).

- [14] Michael Kerrisk, *Linux Programmer's Manual - mmap(2)*. Adresse: <http://man7.org/linux/man-pages/man2/mmap.2.html> (besucht am 23.01.2020).
- [15] ScienceDirect, *Hardware-Software Codesign - an overview / ScienceDirect Topics*. Adresse: <https://www.sciencedirect.com/topics/engineering/hardware-software-codesign> (besucht am 31.01.2020).
- [16] stackexchange.com, *Difference between /dev and /sys*. Adresse: <https://unix.stackexchange.com/questions/176215/difference-between-dev-and-sys> (besucht am 02.01.2020).
- [17] Stephen St. Michael, *Introduction to the Advanced Extensible Interface (AXI) - Technical Articles*, en. Adresse: <https://www.allaboutcircuits.com/technical-articles/introduction-to-the-advanced-extensible-interface-axi/> (besucht am 02.01.2020).
- [18] Thomas Petazzoni, *Device Tree for Dummies*. Adresse: [https://elinux.org/images/f/f9/Petazzoni-device-tree-dummies\\_0.pdf](https://elinux.org/images/f/f9/Petazzoni-device-tree-dummies_0.pdf) (besucht am 02.01.2020).
- [19] X. Wiki, *Linux DMA From User Space - Xilinx Wiki - Confluence*. Adresse: <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842418/Linux+DMA+From+User+Space> (besucht am 04.02.2020).
- [20] Wikipedia, *Advanced eXtensible Interface*, en. Adresse: [https://en.wikipedia.org/w/index.php?title=Advanced\\_eXtensible\\_Interface&oldid=932804926](https://en.wikipedia.org/w/index.php?title=Advanced_eXtensible_Interface&oldid=932804926) (besucht am 02.01.2020).
- [21] www.kernel.org, *DMA Test Guide — The Linux Kernel documentation*. Adresse: <https://www.kernel.org/doc/html/latest/driver-api/dmaengine/dmatest.html> (besucht am 08.02.2020).
- [22] Xilinx Wiki, *Linux Reserved Memory - Xilinx Wiki - Confluence*. Adresse: <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841683/Linux+Reserved+Memory#LinuxReservedMemory-ReservedmemorythroughDMAAPI> (besucht am 04.02.2020).
- [23] Xilinx, Inc., *AR# 58080: Example Design - Using the AXI DMA in scatter gather mode to transfer data to memory*, en. Adresse: <https://www.xilinx.com/support/answers/58080.html> (besucht am 16.01.2020).
- [24] Xilinx, Inc., *AR# 58582: Example Design - Zynq-based FFT co-processor using the AXI DMA*, en. Adresse: <https://www.xilinx.com/support/answers/58582.html> (besucht am 16.01.2020).
- [25] Xilinx, Inc., *AXI Central Direct Memory Access v4.1 LogiCORE IP Product Guide (PG034)*. Adresse: [https://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_cdma/v4\\_1/pg034-axi-cdma.pdf](https://www.xilinx.com/support/documentation/ip_documentation/axi_cdma/v4_1/pg034-axi-cdma.pdf) (besucht am 24.12.2019).

- [26] Xilinx, Inc., *AXI Direct Memory Access v7.1 LogiCORE IP Product Guide (PG021)*. Adresse: [https://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_dma/v7\\_1/pg021\\_axi\\_dma.pdf](https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf) (besucht am 28.11.2019).
- [27] Xilinx, Inc., *AXI Reference Guide (UG761)*, en, 2012. Adresse: [https://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_ref\\_guide/v13\\_4/ug761\\_axi\\_reference\\_guide.pdf](https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_axi_reference_guide.pdf) (besucht am 08.11.2019).
- [28] Xilinx, Inc., *AXI4-Stream FIFO v4.1 LogiCORE IP Product Guide (PG080)*. Adresse: [https://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_fifo\\_mm\\_s/v4\\_1/pg080-axi-fifo-mm-s.pdf](https://www.xilinx.com/support/documentation/ip_documentation/axi_fifo_mm_s/v4_1/pg080-axi-fifo-mm-s.pdf) (besucht am 16.01.2020).
- [29] Xilinx, Inc., *Intellectual Property*, en. Adresse: <https://www.xilinx.com/products/intellectual-property.html> (besucht am 23.01.2020).
- [30] Xilinx, Inc., *Introduction to Linux Device Drivers - Part 1 The Basics*, en. Adresse: <https://www.xilinx.com/video/soc/linux-device-drivers-part-1-the-basics.html> (besucht am 12.02.2020).
- [31] Xilinx, Inc., *Introduction to Linux Device Drivers - Part 2 Platform and Character Drivers*, en. Adresse: <https://www.xilinx.com/video/soc/linux-device-drivers-part-2-platform-character-drivers.html> (besucht am 12.02.2020).
- [32] Xilinx, Inc., *Linux Application Debugging with System Debugger*. Adresse: [https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx14\\_7/SDK\\_Doc/tasks/sdk\\_t\\_linux\\_application\\_debugging\\_system\\_debugger.htm](https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/SDK_Doc/tasks/sdk_t_linux_application_debugging_system_debugger.htm) (besucht am 26.12.2019).
- [33] Xilinx, Inc., *Linux DMA from User Space*, en. Adresse: <https://www.xilinx.com/video/soc/linux-dma-from-user-space.html> (besucht am 12.02.2020).
- [34] Xilinx, Inc., *Linux DMA In Device Drivers*, en. Adresse: <https://www.xilinx.com/video/soc/linux-dma-in-device-drivers.html> (besucht am 12.02.2020).
- [35] Xilinx, Inc., *Linux User Space Device Drivers*, en. Adresse: <https://www.xilinx.com/video/soc/linux-user-space-device-drivers.html> (besucht am 12.02.2020).
- [36] Xilinx, Inc., *PetaLinux Tools Documentation: Reference Guide (UG1144)*. Adresse: [https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2017\\_4/ug1144-petalinux-tools-reference-guide.pdf](https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1144-petalinux-tools-reference-guide.pdf) (besucht am 07.11.2019).
- [37] Xilinx, Inc., *PetaLinux Tools Documentation: Workflow Tutorial (UG1156)*. Adresse: [https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2017\\_3/ug1156-petalinux-tools-workflow-tutorial.pdf](https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug1156-petalinux-tools-workflow-tutorial.pdf) (besucht am 03.11.2019).
- [38] Xilinx, Inc., *PetaLinux Tools User Guide: Application Development Guide (UG981)*. Adresse: [https://www.xilinx.com/support/documentation/sw\\_manuals\\_j/petalinux2014\\_2/ug981-petalinux-application-development-debug.pdf](https://www.xilinx.com/support/documentation/sw_manuals_j/petalinux2014_2/ug981-petalinux-application-development-debug.pdf) (besucht am 23.11.2019).

- [39] Xilinx, Inc., *Vivado Design Suite: AXI Reference Guide (UG1037)*. Adresse: [https://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_ref\\_guide/latest/ug1037-vivado-axi-reference-guide.pdf](https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf) (besucht am 10.09.2019).
- [40] Xilinx, Inc., *Xilinx DS416 Direct Memory Access and Scatter Gather (v2.01a), Data Sheet*. Adresse: [https://www.xilinx.com/support/documentation/ip\\_documentation/dma\\_sg.pdf](https://www.xilinx.com/support/documentation/ip_documentation/dma_sg.pdf) (besucht am 28.11.2019).
- [41] Xilinx, Inc., *Zynq-7000 All Programmable SoC Software Developers Guide (UG821)*. Adresse: [https://www.xilinx.com/support/documentation/user\\_guides/ug821-zynq-7000-swdev.pdf](https://www.xilinx.com/support/documentation/user_guides/ug821-zynq-7000-swdev.pdf) (besucht am 23.11.2019).
- [42] Xilinx, Inc., *Zynq-7000 All Programmable SoC: Embedded Design Tutorial (UG1165)*. Adresse: [https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2017\\_4/ug1165-zynq-embedded-design-tutorial.pdf](https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1165-zynq-embedded-design-tutorial.pdf) (besucht am 07.11.2019).
- [43] Xilinx, Inc., *Zynq-7000 SoC Technical Reference Manual (UG585)*, en, 2018. Adresse: [https://www.xilinx.com/support/documentation/user\\_guides/ug585-Zynq-7000-TRM.pdf](https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf) (besucht am 10.09.2019).
- [44] Yasura's Workshop, *Capture.png Grafik*. Adresse: [https://3.bp.blogspot.com/-9IiqE\\_LgIR8/WezHKv4tjAI/AAAAAAAABFM/H8UUUiE9\\_iACmwq7UCLVHGZ7ykigQ2xCACLcBGAs/s1600/Capture.PNG](https://3.bp.blogspot.com/-9IiqE_LgIR8/WezHKv4tjAI/AAAAAAAABFM/H8UUUiE9_iACmwq7UCLVHGZ7ykigQ2xCACLcBGAs/s1600/Capture.PNG) (besucht am 25.11.2019).

# Anhang

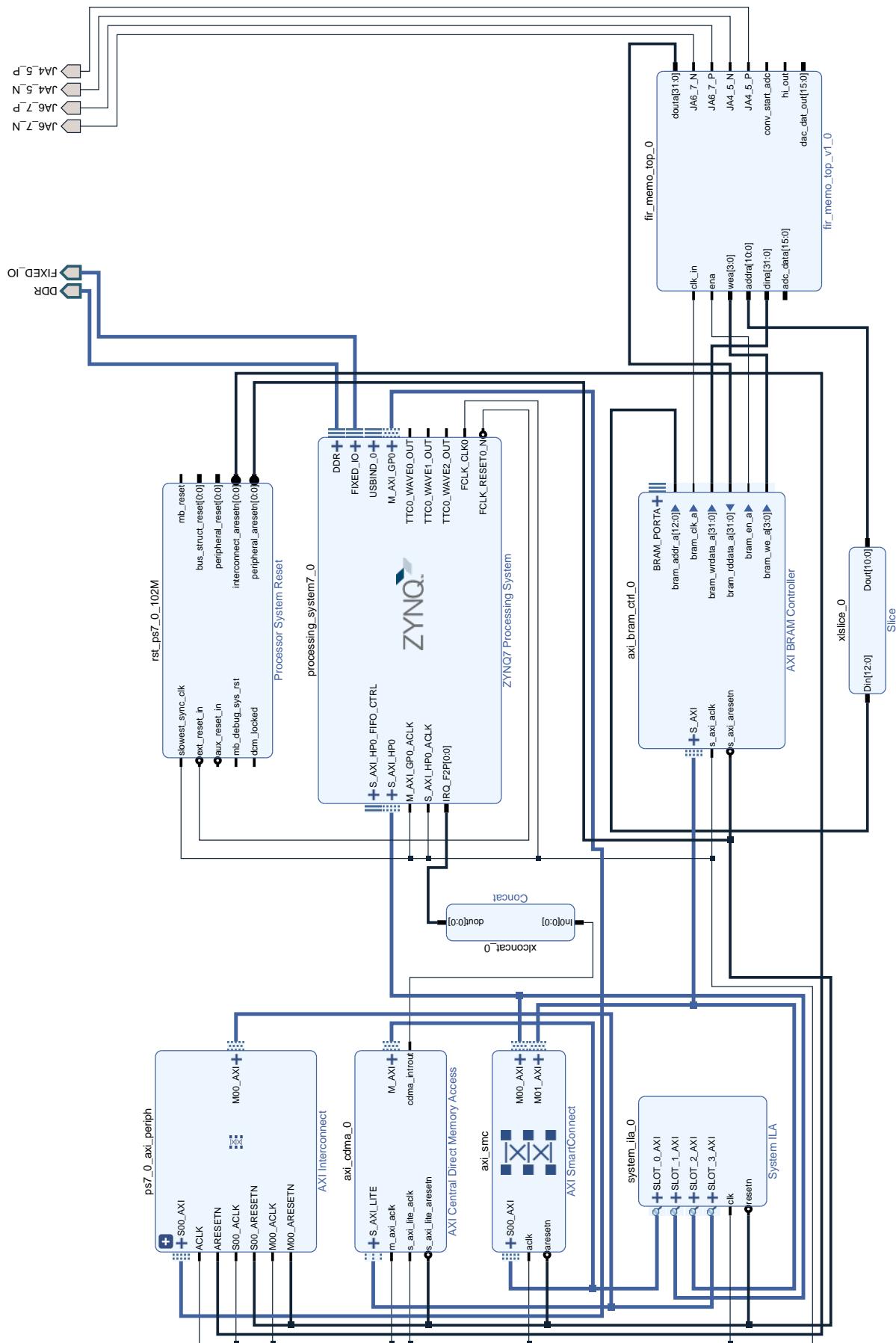
## A. Beiliegender Datenträger

Der Bachelorarbeit liegt zusätzlich ein Datenträger bei. Auf diesem befinden sich die schriftliche Ausarbeitung und die Präsentation in digitaler Form. Außerdem sind weitere wichtige Dokumente, Skripte, Programm-Projekte und Files beigelegt. Die nachfolgende Abbildung zeigt die Ordnerstruktur des Datenträgers und erläutert kurz die enthaltenen Dateien.

01_Bachelorarbeit	
02_Hardware-Designs .....	Archivierte Vivado-Projekte und Block-Designs als PDF.
└── AXI-CDMA	
└── AXI-DMA	
└── AXI-DMA-Lauri-Design	
└── FFT-Example	
03_PetaLinux-Projekte .....	Die PetaLinux-Projekte archiviert und die Dateien, die man für ein lauffähiges Projekt auf einer SD-Karte benötigt.
└── AXI-DMA-petalinux-projekt.zip	
└── CDMA-petalinux-projekt.zip	
└── Files-for-Home	
└── Files-for-SD-Image	
04_SDK-Projekte .....	Alle Source-Codes als C-Programme für den Import in die SDK.
└── axidma_benchmark	
└── axidma_transfer	
└── dma-prox-test	
└── linux_cdma_app_rw	
└── Memcopy_LaurisBlog	
└── Export-of-all-Projekts.zip	
05ILA-Aufzeichnungen .....	Aufzeichnungen von verschiedenen Übertragungen für die Analyse der Übertragungskanäle.
└── axidma-transfer.ila	
└── CDMA-controle.ila	
└── CDMA-sin.ila	
└── memcopy-Lauri.ila	
06_Weiteres .....	Weitere Dateien, die in der Arbeit Erwähnung finden, darunter das Skript um eine SD-Karte mit dem Embedded Linux aufzusetzen.
└── BSP	
└── for HW-Design	
└── SD-Card-Config-Skript.sh	
└── TCL .....	Das TCL-Skript für das Erstellen des Vivado-Projekts und die dafür benötigten Daten. Das IP-repro enthält den aktualisierten 'fir_memo_top'.
└── Create-CDMA-Projekt.tcl	
└── Ressourcen	
└── IP_repro	
07_nuetzliche-Dokumente	

Die Verwendung der Dateien und einige genauere Informationen sind in diversen `ReadMe.txt` Files beschrieben.

## B. CDMA Design



## C. C-Code

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/stat.h>
5 #include <fcntl.h>
6 #include <sys/mman.h>
7 #include <unistd.h>
8 #include <string.h>
9 #include <sys/time.h>           // Timing functions and definitions
10 #include "conversion.h"        // Miscellaneous conversion utilities
11 #include "sin_dat.h"          // Sinus Werte mit 2 Werten hintereinander
12
13 // The purpose this test is to show that users can get to devices in user
14 // mode .This is not to say this should replace a kernel driver, but does
15 // provide some short term solutions sometimes
16 // or a debug solution that can be helpful.
17
18 #define CDMA_BASE_ADDRESS      0x7E200000
19
20 #define DDR_BASE_ADDRESS_READ   0x20000000
21
22 #define DDR_BASE_ADDRESS_WRITE  0x20000800 //800 entspricht 2048 Byte weiter
23
24 #define BRAM_BASE_ADDRESS       0x42000000 //fir_memo_top
25
26 /* CDMACR Register Offsets
27 * Tabelle 2-7
28 */
29 #define XAXICDMA_CR_OFFSET     0x00000000 /*< Control register */
30 #define XAXICDMA_SR_OFFSET      0x00000004 /*< Status register */
31 #define XAXICDMA_CDESC_OFFSET   0x00000008 /*< Current descriptor pointer */
32 #define XAXICDMA_TDESC_OFFSET   0x00000010 /*< Tail descriptor pointer */
33 #define XAXICDMA_SRCADDR_OFFSET 0x00000018 /*< Source address register */
34 #define XAXICDMA_DSTADDR_OFFSET 0x00000020 /*< Destination address register */
35 #define XAXICDMA_BTT_OFFSET     0x00000028 /*< Bytes to transfer */
36
37 /* Bitmasks of XAXICDMA_CR_OFFSET register
38 */
39 #define XAXICDMA_CR_RESET_MASK  0x00000004 /*< Reset DMA engine */
40 #define XAXICDMA_CR_SGMODE_MASK 0x00000008 /*< Scatter gather mode */
41
42 /* Bitmasks for interrupts for XAXICDMA_CR_OFFSET and XAXICDMA_SR_OFFSET
43 */
44 #define XAXICDMA_XR_IRQ_IOC_MASK 0x00001000 /*< Completion interrupt */
45 #define XAXICDMA_XR_IRQ_DELAY_MASK 0x00002000 /*< Delay interrupt */
46 #define XAXICDMA_XR_IRQ_ERROR_MASK 0x00004000 /*< Error interrupt */
47 #define XAXICDMA_XR_IRQ_ALL_MASK  0x00007000 /*< All interrupts */
48 #define XAXICDMA_XR_IRQ_SIMPLE_ALL_MASK 0x00005000 /*< All interrupts for
49                                         simple only mode */
50
51 /* Bitmasks of XAXICDMA_SR_OFFSET register
52 * This register reports status of a DMA channel
53 * Tabelle 2-8
54 */
54 #define XAXICDMA_SR_IDLE_MASK    0x00000002 /*< DMA channel idle */
55 #define XAXICDMA_SR_SGINCLD_MASK 0x00000008 /*< Hybrid build */
56 #define XAXICDMA_SR_ERR_INTERNAL_MASK 0x00000010 /*< Datamover internal err */
57 #define XAXICDMA_SR_ERR_SLAVE_MASK 0x00000020 /*< Datamover slave err */
58 #define XAXICDMA_SR_ERR_DECODE_MASK 0x00000040 /*< Datamover decode err */
59 #define XAXICDMA_SR_ERR_SG_INT_MASK 0x00000100 /*< SG internal err */
60 #define XAXICDMA_SR_ERR_SG_SLV_MASK 0x00000200 /*< SG slave err */

```

- 1 -

```

61 #define XAXICDMA_SR_ERR_SG_DEC_MASK 0x00000400 /*< SG decode err */
62 #define XAXICDMA_SR_ERR_ALL_MASK 0x00000770 /*< All errors */
63
64 //Größe für CDMA mmap
65 #define MAP_SIZE 4096UL
66 #define MAP_MASK (MAP_SIZE - 1)
67 //Größe für DDR mmap
68 #define DDR_MAP_SIZE 0x2000 //in bytes in Hex 2000 entspricht 8kB
69 #define DDR_MAP_MASK (DDR_MAP_SIZE - 1)
70 /*
71 * Parameter für die Übertragung
72 */
73 #define BUFFER_BYTESIZE 2048 // Length of the buffers for DMA transfer in byte
74
75 #define max_adr 512 //Addressen des Speichers in der PL
76
77 //*****
78 /*
79 * Array mit Dreieck vorbelegen
80 */
81 void ex_triang(int32_t *Array)
82 {
83     int16_t increment, value;
84     int32_t loword, hiword, outval;
85     value = 0;
86     increment = 64;
87     for (int i = 0; i < max_adr/2; i++)
88     {
89         value += increment;
90         loword = value;
91         value += increment;
92         hiword = (value << 16) & 0xffff0000;
93         outval = hiword | (loword & 0x0000ffff);
94         Array[i] = outval;
95     }
96     for (int i = max_adr/2; i < max_adr; i++)
97     {
98         value -= increment;
99         loword = value;
100        value -= increment;
101        hiword = (value << 16) & 0xffff0000;
102        outval = hiword | (loword & 0x0000ffff);
103        Array[i] = outval;
104    }
105    Array[510]= Array[510] & 0xffffffff; //damit im mem mode bleibt
106    printf("Dreiecksignal\n");
107}
108
109 /*
110 * Array mit Sinus vorbelegen
111 */
112 void ex_restore_sin(int32_t *Array)
113 {
114     for (int i = 0; i < max_adr; i++)
115     {
116         Array[i] = sin_dat[i];
117     }
118     Array[510]= Array[510] & 0xffffffff; //damit man im memory-mode bleibt
119     printf("Sinus -> Speicher\n");

```

- 2 -

```

120 }
121 /*
122 * Array leeren
123 */
124 void ex_clear(int32_t *Array)
125 {
126     for (int i = 0; i < max_adr; i++ )
127     {
128         Array[i] = 0x00000000;
129     }
130     printf("Speicher wird geloescht\n");
131 }
132 /*
133 * holt DDR in den userspace und kopiert Array hinein und unmaped DDR wieder
134 */
135 void copytoDDR(int32_t *Array)
136 {
137     //Source Speicher im DDR
138     int memfd_1;
139     void *mapped_dev_src;
140     void *mapped_dev_base_1;
141     off_t dev_base_1 = DDR_BASE_ADDRESS_READ;
142
143     //map DDR to userspace
144     memfd_1 = open("/dev/mem", O_RDWR | O_SYNC);
145     if (memfd_1 == -1)
146     {
147         printf("Can't open /dev/mem.\n");
148         exit(0);
149     }
150     printf("/dev/mem opened. DDR\n");
151     mapped_dev_src = mmap(0, DDR_MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, memfd_1, dev_base_1 & ~DDR_MAP_MASK);
152     if (mapped_dev_base_1 == (void *) -1)
153     {
154         printf("Can't map the memory to user space.\n");
155         exit(0);
156     }
157     mapped_dev_base_1 = mapped_dev_src + (dev_base_1 & DDR_MAP_MASK);
158     printf("Memory mapped DDR at address %p.\n", mapped_dev_base_1);
159     //copy Array to DDR
160     memcpy(mapped_dev_base_1, Array, (BUFFER_BYTESIZE));
161     // Unmap DDR vrom userspace
162     if (munmap(mapped_dev_base_1, DDR_MAP_SIZE) == -1)
163     {
164         printf("Can't unmap memory from user space.\n");
165         exit(0);
166     }
167     close(memfd_1);
168     printf("/dev/mem closed. DDR\n");
169 }
170 /*
171 * öffnet dev/mem und macht device im userspace verfügbar
172 * gibt die dev_adresse zurück
173 */
174 void* openDEV(int *memfd)
175 {

```

- 3 -

```

179     void* device;
180     off_t dev_base = CDMA_BASE_ADDRESS;
181
182     *memfd = open("/dev/mem", O_RDWR | O_SYNC);
183     if (*memfd == -1)
184     {
185         printf("Can't open /dev/mem.\n");
186         exit(0);
187     }
188     printf("/dev/mem opened CDMA.\n");
189
190     // Map one page of memory into user space such that the device is in that
191     // page, but it may not
192     // be at the start of the page.
193     device = mmap(0, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, *memfd,
194     dev_base & ~MAP_MASK);
195     if (device == (void *) -1)
196     {
197         printf("Can't map the memory to user space.\n");
198         exit(0);
199     }
200     return device;
201 }
202 */
203 * Un-map Device from User layer
204 */
205 void closeDEV(int *memfd, void* device)
206 {
207     if (munmap(device, MAP_SIZE) == -1)
208     {
209         printf("Can't unmap memory from user space.\n");
210         exit(0);
211     }
212     if (close(*memfd) == 0)
213     {
214         printf("/dev/mem closed Device.\n");
215     }
216     else
217     {
218         printf("Error closing Device\n");
219     }
220 }
221 */
222 * Init CDMA
223 * Reset und Aktivierung der Interrupts
224 */
225 void initCDMA(void * device)
226 {
227     printf("CDMA init\n");
228     unsigned int ResetMask;
229     unsigned int RegValue;
230     unsigned int TimeOut =5;
231     //Reset CDMA
232     do{
233         ResetMask = (unsigned long)XAXICDMA_CR_RESET_MASK;
234         *((volatile unsigned long *) (device + XAXICDMA_CR_OFFSET)) = (unsigned
235         long)ResetMask;

```

- 4 -

```

236     /* If the reset bit is still high, then reset is not done */
237     ResetMask = *((volatile unsigned long *) (device + XAXICDMA_CR_OFFSET));
238     if(!(ResetMask & XAXICDMA_CR_RESET_MASK))
239     {
240         break;
241     }
242     TimeOut -= 1;
243 }while (TimeOut);
244 //enable Interrupt
245 RegValue = *((volatile unsigned long *) (device + XAXICDMA_CR_OFFSET));
246 RegValue = (unsigned long)(RegValue | XAXICDMA_XR_IRQ_ALL_MASK );
247 *((volatile unsigned long *) (device + XAXICDMA_CR_OFFSET)) = (unsigned long)RegValue;
248 // Check the DMA Mode and switch it to simple mode
249 RegValue = *((volatile unsigned long *) (device + XAXICDMA_CR_OFFSET));
250 if(RegValue & XAXICDMA_CR_SGMODE_MASK)
251 {
252     RegValue = (unsigned long)(RegValue & (~XAXICDMA_CR_SGMODE_MASK));
253     printf("Reading \n \r");
254     *((volatile unsigned long *) (device + XAXICDMA_CR_OFFSET)) = (unsigned long)RegValue ;
255 }
256 }
257
258 /*
259 * Datentransfer Übergabe von source- und destination- Adresse
260 */
261 void datatransfer(void * device, unsigned long source, unsigned long destination)
262 {
263     printf("transfer\n");
264     unsigned int RegValue;
265
266     // Checking for the Bus Idle
267     RegValue = *((volatile unsigned long *) (device + XAXICDMA_SR_OFFSET));
268     if(!(RegValue & XAXICDMA_SR_IDLE_MASK))
269     {
270         printf("BUS IS BUSY Error Condition \n\r");
271         exit(1);
272     }
273
274     //Set the Source Address
275     *((volatile unsigned long *) (device + XAXICDMA_SRCADDR_OFFSET)) = source;
276     //Set the Destination Address
277     *((volatile unsigned long *) (device + XAXICDMA_DSTADDR_OFFSET)) = destination;
278     RegValue = (unsigned long)(BUFFER_BYTESIZE); //länge
279     // write Byte to Transfer -> startet transfer
280     *((volatile unsigned long *) (device + XAXICDMA_BTT_OFFSET)) = (unsigned long)RegValue;
281
282     //Heißes Warten auf einen Interrupt über den transfer Status
283     do
284     {
285         RegValue = *((volatile unsigned long *) (device + XAXICDMA_SR_OFFSET));
286     }while(!(RegValue & XAXICDMA_XR_IRQ_ALL_MASK));
287     //Interrupt auswerten
288     if((RegValue & XAXICDMA_XR_IRQ_IOC_MASK))
289     {
290         printf("Transfer Completed \n\r ");
291     }
292

```

```

293     }
294     if((RegValue & XAXICDMA_XR_IRQ_DELAY_MASK))
295     {
296         printf("IRQ Delay Interrupt\n\r ");
297     }
298     if((RegValue & XAXICDMA_XR_IRQ_ERROR_MASK))
299     {
300         printf(" Transfer Error Interrupt\n\r ");
301     }
302     //Clear Interrupt Flag
303     RegValue = *((volatile unsigned long *) (device + XAXICDMA_SR_OFFSET));
304     RegValue = (unsigned long)(RegValue | XAXICDMA_XR_IRQ_ALL_MASK );
305     *((volatile unsigned long *) (device + XAXICDMA_SR_OFFSET)) = (unsigned long)RegValue;
306 }
307 */
308 /* Main
309 */
310 int main()
311 {
312     printf("sizeof(int)=%i\n",sizeof(int));
313     printf("sizeof(int16_t)=%i\n",sizeof(int16_t));
314     printf("sizeof(unsigned long)=%i\n",sizeof(unsigned long));
315     printf("getpagesize()=%i\n",getpagesize());
316
317     //CDMA
318     int CDMA_fd =0;
319     void *mapped_dev_base_CDMA = NULL;
320
321     int32_t SrcArray[BUFFER_BYTESIZE/4 ];
322     int key = 0; //für Tastatureingabe
323     int leave =0; //für Tastatureingabe
324
325     //make CDMA Accesable
326     mapped_dev_base_CDMA = openDEV(&CDMA_fd);
327     initCDMA(mapped_dev_base_CDMA);
328
329     printf("Taste druecken ! \n");
330     while(leave != 1)
331     {
332         if (key != 0xa)
333         {
334             printf("\n0 : Speichern \n1 : rekonstruieren \n2: Dreieck \n3: Leeren \n4: Sinus ");
335             printf("\n6: Exit \n");
336             printf("Taste druecken ! \n");
337             }
338             scanf("%d", &key);
339             switch (key)
340             {
341                 case 0: //Store
342
343                     datatransfer(mapped_dev_base_CDMA, BRAM_BASE_ADDRESS, DDR_BASE_ADDRESS_WRITE);
344                     break;
345                 case 1: //Restore
346
347                     datatransfer(mapped_dev_base_CDMA, DDR_BASE_ADDRESS_WRITE, BRAM_BASE_ADDRESS);
348             }
349     }
350 }

```

```

347     break;
348     case 2: //Dreieck
349         ex_triang(SrcArray);
350         copytoDDR(SrcArray); //Daten im DDR ablegen
351             datatransfer(mapped_dev_base_CDMA,DDR_BASE_ADDRESS_READ,BRAM_BASE_ADDRESS);
352             );
353             break;
354             case 3: //Löschen
355                 ex_clear(SrcArray);
356                 copytoDDR(SrcArray); //Daten im DDR ablegen
357                     datatransfer(mapped_dev_base_CDMA,DDR_BASE_ADDRESS_READ,BRAM_BASE_ADDRESS);
358                     );
359                     break;
360                     case 4: //Sinus
361                         ex_restore_sin(SrcArray);
362                         copytoDDR(SrcArray); //Daten im DDR ablegen
363                             datatransfer(mapped_dev_base_CDMA,DDR_BASE_ADDRESS_READ,BRAM_BASE_ADDRESS);
364                             );
365                             break;
366                             case 5: //not used
367                                 break;
368                                 case 6: //Schließen
369                                     printf("exit\n");
370                                     leave =1;//exit(0);
371                                     break;
372                                     case 0xa:
373                                         break;
374                                         default:
375                                             printf("key = %x \n",key);
376                                             break;
377                                         }//end switch
378 } //end while 1
379 // Unmap the CDMA-Device
380 closeDEV(&CDMA_fd, mapped_dev_base_CDMA);
381 printf("ende\n");
382 return 0;
383 }
384

```

# D. AXI-DMA Design

