Patrick Seebacher, BSc

Evaluierung von Inter-Core Kommunikations-Konzepten

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

Masterstudium Telematik

eingereicht an der

Technischen Universität Graz

Betreuer

Univ.-Prof. Dipl.-Inf. Univ. Dr.rer.nat. Marcel Carsten Baunach
Mitwirkender: Dipl.-Ing. Fabian Mauroner, BSc

Institut für Technische Informatik

Graz, Dezember 2017

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Datum Unterschrift

Danksagung

Diese Diplomarbeit wurde im Studienjahr 2016/2017 am Institut für Technische Informatik der Technischen Universität Graz durchgeführt.

Mein herzlicher Dank gilt allen, die mich bei der Erstellung dieser Arbeit unterstützt haben. Besonders danken möchte ich meinen Betreuern vom Institut für Technische Informatik. Zum einen Herrn Univ.-Prof Marcel Baunach und zum anderen Herrn Dipl.-Ing. Fabian Mauroner. Ich danke ihnen sehr für dieses interessante Thema und dafür, dass ich auf ihre ausgezeichnete Unterstützung vertrauen konnte. Die Arbeit profitierte sehr durch die stets verfügbare Diskussionsbereitschaft und die interessanten Ideen meiner Betreuer.

Weiters gilt mein Dank meiner großartigen Familie. Meine Eltern und meine Schwester haben mich während meiner gesamten Studienzeit unterstützt und die Zeit in meiner Heimat konnte ich immer zur Erholung nutzen.

Ein großes Dankeschön gebührt noch meinen Freunden, welche immer für mich da waren und dafür gesorgt haben, dass ich meine Freizeit ausgiebig nutze. Ihnen und all meinen Studienkollegen danke ich für die schöne Zeit an der Universität.

Abschließend möchte ich mich bei meiner wunderbaren Freundin bedanken. Sie war in unserer gemeinsamen Zeit immer eine starke Stütze und konnte meine Motivation stets hoch halten.

Graz, Patrick Seebacher

Kurzfassung

Diese Masterarbeit ist gegliedert in drei Teile. Ausgangspunkt ist die Evaluierung des Open-Source Prozessors „J2 Core“. Dieser Prozessor ist eine Reinraum-Implementierung[1] des SuperH-2 Befehlssatzes mit Erweiterungen und ist seit 2015 als Softcore[2] verfügbar.

Features wie zum Beispiel die hohe Codedichte, die Open-Source Implementierung und die Verfügbarkeit von günstiger Hardware, machen die SuperH Architektur interessant.
In der Evaluierung soll die Brauchbarkeit des J2 Core für diese und zukünftige Arbeiten untersucht werden und gegebenenfalls soll ein anderer Prozessor ausgewählt werden. Die RISC-V Architektur dient in diesem Fall als Alternative.
Relevant ist die Qualität und die Lesbarkeit der Logik, sowie die nötige Toolchain und der Umfang der Dokumentation.

Der nächste Teil der Arbeit umfasst die Evaluierung von Inter-Core-Bussen wie AMBA, CoreConnect, Avalon, STBus und Wishbone. Ziel ist es, einen passenden Bus für den ausgewählten Prozessor und die geplante Implementierung zu finden.
Der optimale Bus sollte leicht integriert werden können und die Wiederverwendbarkeit aller entwickelten Komponenten gewährleisten. Zusätzlich soll der Bus dynamische Prioritäten unterstützen oder um diese erweitert werden können. Weitere interessante Aspekte der Inter-Core Busse werden im Bezug auf zukünftige Arbeiten evaluiert.

Analyse zu Entwickelung und Test des Multi-Master Systems bildet den letzten Teil dieser Arbeit. Die Informationen der Prozessorevaluierung und der Busevaluierung wurden genutzt, um das System zu entwerfen. Zum Beispiel, wie die Bus-Topologie umgesetzt wird und die Hardware mit dynamischen Prioritäten des Betriebssystems arbeitet. Weiterer Fokus liegt auf der Erweiterbarkeit und Wiederverwendbarkeit des Systems, sowie die Frage, wie das System mit zusätzlichen Komponenten skaliert.

[1] Entwickelt ohne den Einsatz der originalen Implementierung

[2] Ein Prozessor der als Quellcode oder in Form einer Netzliste vorliegt und in programmierbarer Logik implementiert werden kann

Abstract

This master thesis is divided into three parts. Starting point of the thesis is the evaluation of the open-source “J2 core“. This processor is a cleanroom reimplementation of the SuperH-2 ISA with extensions and is available as soft core since 2015.
Features like the high code density, patent and royalty free (BSD licensed) implementation and availability of low cost hardware development platforms make this architecture interesting.
The evaluation is intended to assess the usability of the J2 core for this and future work. Furthermore, the processor should be replaced if necessary. The RISC-V architecture serves as alternative in this case. Relevant for the evaluation is the code quality, readability of the code, complexity of the required tool chain and quality of the documentation.

The next part of the thesis deals with the comparison of inter-core communication busses like AMBA, CoreConnect, Avalon, STBus and Wishbone. The goal is to find a suitable bus for the selected processor and the intended implementation.
The optimal bus should be easy to integrate and ensure reusability for every component. Additionally, the bus should provide support for dynamic task priorities. Other interesting aspects of the inter-core bus will be evaluated in reference to future work.

Analysis of the development and test of the multi-master system is the last part of this work. The collected information of the core- and bus-evaluation is used to design specific elements of the system, like the bus interconnection and how the hardware bus works with dynamic priorities set by the operating system on the processor core. Another focus is the extendability of the system and how the resource requirements scale with more components.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ACE AXI Coherency Extensions

AHB Advanced High-performance Bus

AMBA Advanced Microcontroller Bus Architecture

APB Advanced Peripheral Bus

ASB Advanced System Bus

ASIC Application-Specific Integrated Circuit

AXI Advanced eXtensible Interface

BSD Berkeley Software Distribution

CDMA Code Division Multiple Access

CISC Complex Instruction Set Computer

CLB Configurable Logic Block

CMPXCHG Compare and Exchange

CPU Central Processing Unit

CSR Control and Status Register

DCR Device Control Register Bus

DSP Digital Signal Processor

EAS Embedded Automotive Systems

ELC Embedded Linux Conference

ESA European Space Agency

FPGA Field Programmable Gate Array

FPU Floating Point Unit

GPIO General Purpose Input/Output

HASTI Highly Advanced System Transport Interface

IP-Core Intellectual Property Core

ISA Instruction Set Architecture

LPDDR Low Power Double Data Rate

LUT Look-Up Table

MMU Memory Management Unit

NASTI Not A STandard Interface

NoC Network-on-Chip

OPB On-Chip Peripheral Bus

OPF Open Processor Foundation

PLB Processor Local Bus

POCI Peripheral Oriented Connection Interface

RAM Random-Access-Memory

RMW Read-Modify-Write

ROM Read-Only-Memory

RISC Reduced Instruction Set Computer

RTL Register-Transfer-Level

SDK Software Development Kit

SIMD Single Instruction Multiple Data

SoC System-on-a-Chip

TCB Task Control Block

TDMA Time Division Multiple Access

UCB University of California, Berkeley

UART Universal Asynchronous Receiver Transmitter

VHDL Very High Speed Integrated Circuit Hardware Description Language

Kapitel 1

Einleitung

Nach Auswahl eines Prozessors und eines passenden Bus-Systems beschäftigt sich diese Arbeit mit der Entwicklung eines eingebetteten Multi-Core-Systems. Spezieller Fokus bei diesem System liegt auf der Unterstützung von dynamischen Prioritäten innerhalb des Bus-Verbindungsnetzwerkes.
Dieses Einleitungskapitel umfasst eine Motivation, die Zielsetzung für diese Arbeit und den Aufbau dieser Arbeit.

1.1 Motivation

Eingebettete Systeme und System-on-a-Chip (SoC) werden heutzutage immer komplexer. Neben der Leistungsfähigkeit von einzelnen Komponenten steigt vor allem die Anzahl individueller Komponenten innerhalb eines Systems. Herkömmliche Designs mit einem Prozessor und dazugehöriger Peripherie werden viel öfter durch Mehrkern- oder Mehrprozessorsysteme abgelöst.
Neben der Komplexität interner Busverbindungsnetzwerke, steigt auch die Bandbreite, die zur Verfügung gestellt werden muss. Dementsprechend müssen Systemdesigner, in Bezug auf die gesamte Entwicklungszeit, mehr Aufmerksamkeit auf die Entwicklung dieser Verbindungsnetzwerke legen.

Hauptmotivation dieser Arbeit ist daher die Entwicklung eines Bus Verbindungsnetzwerkes für ein Multi-Core-System. Um das System besser prädizierbar zu machen und somit attraktiv für Echtzeitanwendungen, soll das Bus-System dynamische Task-Prioritäten unterstützen. Diese Prioritäten werden vom Betriebssystem gesetzt und über die Bus-Schnittstelle an das System weitergegeben.

Ein weiterer wichtiger Faktor bei der Entwicklung von eingebetteten Systemen oder SoCs, ist die Wiederverwendbarkeit von bereits entwickelten Komponenten. Bei komplexen Systemen kann der Einsatz von bereits vorhandenen

Komponenten die Entwicklungszeit massiv verkürzen. Die Erweiterbarkeit von Systemen, beziehungsweise der mögliche Einsatz von einzelnen Komponenten in den unterschiedlichsten Systemen, sollte daher von jedem Systemdesigner berücksichtigt werden.

1.2 Zielsetzung

Diese Arbeit umfasst drei aufeinanderfolgende Ziele. Anfänglich soll der Open-Source Softcore „J2 Core“ evaluiert werden. Die Evaluierung soll die Brauchbarkeit des J2 Core für diese und zukünftige Arbeiten am Institut für Technische Informatik untersuchen. Wenn dieser Softcore nicht für diese Arbeit eingesetzt wird, so sollte eine passende Alternative gefunden werden.

Parallel zur Prozessorauswahl soll ein passendes Bus-System für die geplante Implementierung dieser Arbeit gefunden werden. Für diesen Zweck sollen Inter-Core Bus-System wie Advanced Microcontroller Bus Architecture (AMBA), Core-Connect, Avalon, STBus und Wishbone im Bezug auf den gewählten Prozessor und den Anforderungen der Implementierung analysiert werden.

Basierend auf den beiden Evaluierungen soll nun ein Multi-Master System entwickelt und getestet werden. Folgende Punkte sollen bei diesem System besonders beachtet werden:

- Einfache Erweiterbarkeit des Gesamtsystems
- Wiederverwendbarkeit der einzelnen Komponenten / Intellectual Property Cores (IP-Cores)
- Dynamisch erweiterbares Bussystem / Bus-Verbindungsnetzwerk
- Arbitrierung über dynamische Prioritäten, des laufenden Task, gesetzt durch das Betriebssystem (Task-Awareness). Das Bussystem soll anhand der Priorität entscheiden welche Verbindung priorisiert werden soll.
- Verilog[3] als bevorzugte Hardwarebeschreibungssprache aufgrund von existierenden Komponenten in Verilog

Das fertige System soll anschließend simuliert und auf einem Field Programmable Gate Array (FPGA) getestet werden. Hierfür wird das Nexys4 Board[4] von Digilent, basierend auf einem Arktix-7-FPGA[5] von Xilinx eingesetzt.

[3] http://ieeexplore.ieee.org/document/1620780/
[4] https://reference.digilentinc.com/reference/programmable-logic/nexys-4/start
[5] https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html

1.3 Aufbau der Arbeit

Kapitel 2 befasst sich mit der Prozessorauswahl. Der initial gewählte Prozessor wird genauer beleuchtet und Vorteile sowie Nachteile analysiert. In weiterer Folge wird eine alternative Architektur untersucht und eine passende Implementierung ausgewählt.

Kapitel 3 beschreibt die Auswahl eines geeigneten Bus-Systems in Bezug auf den ausgewählten Prozessor. Diese Evaluierung wurde im Gegensatz zu dieser Gliederung parallel zur Prozessorevaluierung durchgeführt. Es wird analysiert, welches Bus-System am besten für die Implementierung dieser Arbeit geeignet ist.
Kapitel 4 behandelt ähnliche Arbeiten im Bezug auf die Busevaluierung und die Implementierung dieser Masterarbeit.

Kapitel 5 beschäftigt sich mit der Entwicklung des Multi-Core-Systems. Zu Beginn wird die Anpassung des Prozessors sowie die Implementierung des dynamischen Crossbar-Switches erläutert. Weiters folgt der Aufbau des Gesamtsystems und die Änderungen, die im Betriebssystem nötig sind um Task-spezifische Prioritäten an das Bus-System zu übertragen.

In Kapitel 6 wird zuerst die Skalierbarkeit, also der Ressourcenverbrauch der Hardwarekomponenten, analysiert. In weiterer Folge werden Ergebnisse der Simulation sowie Messergebnisse bei unterschiedlichen Testprogrammen demonstriert.

Den Abschluss dieser Arbeit bildet Kapitel 7 mit einem Ausblick auf mögliche Weiterentwicklungen und einer Zusammenfassung.

Kapitel 2

Prozessorevaluation

Zu Beginn der Arbeit wurde der J2 open processor, mit der SuperH[6] Instruction Set Architecture (ISA), von der Open Processor Foundation (OPF) [10] als Prozessor für die weitere Bus-Evaluierung, sowie für die Implementierung festgelegt. Da der Prozessor noch nicht im *mosart*MCU-Projekt[7] verwendet wurde, befasste sich der erste Teil der Arbeit mit der Evaluierung dieses Prozessors.

Hauptaugenmerk der Evaluierung lag auf der Kompatibilität zum *mosart*MCU-Projekt. Das Projekt läuft unter der Arbeitsgruppe Embedded Automotive Systems (EAS) am Institut für Technische Informatik der Technischen Universität Graz.
Schlussendlich wurde im Verlauf der Arbeit der verwendete Prozessor gewechselt. Als Alternative wurde V-scale[8] gewählt. V-scale ist eine Implementierung der RISC-V [11] Open Instruction Set Architecture (ISA) von der University of California, Berkeley (UCB).

In diesem Kapitel werden nun beide Prozessoren genauer beschrieben. Die Features der Prozessoren sind im Abschnitt 2.1.3 beziehungsweise im Abschnitt 2.2.2 zu finden. Die Zusammenfassung am Ende soll klären, warum der RISC-V Prozessor verwendet wurde.

[6] https://www.renesas.com/en-us/products/microcontrollers-microprocessors/superh.html
[7] Multi-Core Operating-System-Aware Real-Time Microcontroller
[8] https://github.com/ucb-bar/vscale

2.1 J2-Core

2.1.1 SuperH (SH)-Architektur

Die SuperH-Architektur ist ein japanisches Reduced Instruction Set Computer (RISC) Central Processing Unit (CPU)-Design. Entwickelt wurde die Architektur ursprünglich von Hitachi in den späten 90er-Jahren. Mittlerweile führt Renesas Electronics, Zusammenschluss der Halbleiterbereiche von Hitachi, Mitsubishi Electric und NEC, die Produktion der SuperH fort.

Hitachi hat fünf Generationen der SuperH-Architektur entwickelt. SH-1 und SH-2 wurden zum Beispiel in der Videospielkonsole Sega Saturn von 1994 verwendet. Zwei SH-2 Prozessoren wurden hier als Hauptprozessor für das System eingesetzt. Interessanterweise wurde von vielen Spiele-Entwicklern, aufgrund einer unzureichenden Entwicklungsumgebung (SDK) nur ein Prozessor verwendet. Die volle Leistung der Konsole wurde folglich nur selten ausgenützt.

Beim SH-3 wurde die maximale Taktfrequenz stark erhöht und zusätzlich eine Memory Management Unit (MMU) verbaut. Der Befehlssatz (ISA) wurde wie bei allen folgenden Generationen angepasst und um zusätzliche Instruktionen ergänzt. Spezielle Formen der SuperH-Architektur enthalten oft noch weitere, spezialisierte Instruktionen. Zum Beispiel ist der SH3-DSP eine Spezialform, der noch heute als digitaler Signalprozessor im Musikbereich eingesetzt wird.

Wie seine Vorgänger ist der SH-4 ebenfalls ein 32-Bit-Mikroprozessor und arbeitet in der Videospielkonsole Sega Dreamcast als Hauptprozessor. Wie die anderen SuperH-Generationen wird auch der SH-4 häufig in der Automobilindustrie eingesetzt. Die kleineren Mikroprozessoren (SH-2) werden in Steuergeräten (Motorsteuergerät) verwendet, wohingegen SH-4-Mikroprozessoren primär für Multimediasysteme konzipiert sind.

Eine Besonderheit der SuperH-Architektur ist die hohe Codedichte die durch 16-Bit breite Instruktionen ermöglicht wird. Die Codedichte gibt an, wie viel Speicherbedarf durchschnittlich von den Instruktionen benötigt wird. Damals war es üblich, dass ein 32-Bit-RISC-Prozessor auch 32-Bit Instruktionen verwendet. Das Design der SuperH-Architektur verband somit die Vorteile von RISC-Prozessoren mit der höheren Codedichte einer Complex Instruction Set Computer (CISC)-Architektur.

Weaver und McKee [1] untersuchten die Codedichte verschiedenster Architekturen und SuperH konnte fast mit x86 mithalten. x86 setzt den Maßstab für einen Befehlssatz im Bezug auf dichten Assemblercode.

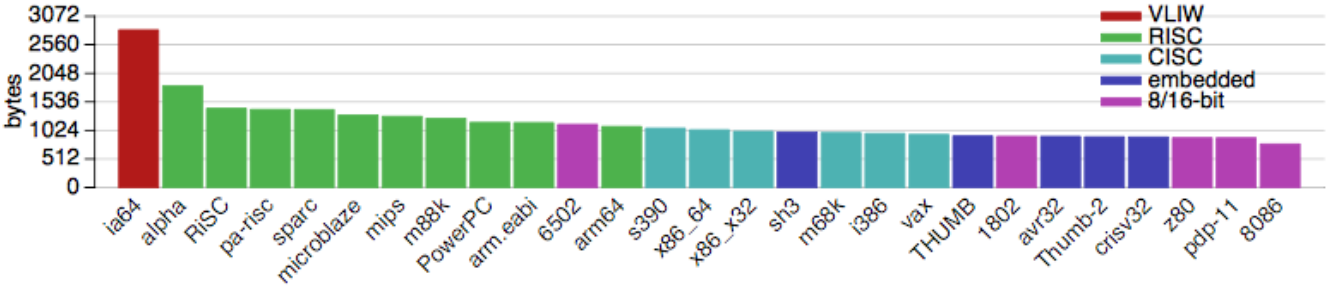

Abbildung 2.1: Gesamtgröße der Benchmarks von Weaver und McKee [1].

Abbildung 2.1 zeigt den Vergleich der Codedichte. Zu beachten ist, dass auch plattformspezifischer Code enthalten ist. Daher zeigt der Graph nicht unbedingt die genaue Codedichte der einzelnen Architekturen. Eine Architektur mit höherer Codedichte ist zum Beispiel die von ARM-Thumb-CPUs. Dieser Befehlssatz wurde aber auf Basis einiger SuperH-Patente entwickelt.

Zu Beginn der SuperH-Architektur war Speicher kostbar und die Produktion somit günstiger wenn weniger Speicher verbaut werden musste. Aufgrund dieser Eigenschaft werden SuperH-Mikroprozessoren, trotz der alten Architektur, auch heute noch weltweit eingesetzt. Das Alter und die damit auslaufenden Patente ermöglichen die Entwicklung von Open-Source-Varianten der SuperH-Architektur.

2.1.2 Überblick zur J2-Entwicklung

Die letzten SH-2 Patente sind im Oktober 2014 ausgelaufen. Anschließend auf der Linuxcon Tokio im April 2015 präsentierten die Entwickler zum ersten Mal ihr Reinraumdesign der SH-2 ISA mit Erweiterungen.

Da der Name "SuperH" noch durch Trademarks geschützt ist, wird für die Open-Source-Varianten der Name "J-Core" verwendet. Im Juni 2015 haben die J-Core Entwickler dann die erste Register-Transfer-Level (RTL)-Version des J2-Core veröffentlicht.

Folgend auf der Embedded Linux Conference (ELC) 2016 [12] wurde ein zweiter Release des J2-Core präsentiert. Zusätzlich zeigten die Entwickler mehr Details zur Entstehung ihres Designs und eine Roadmap für die weiteren J-Core Entwicklungen.

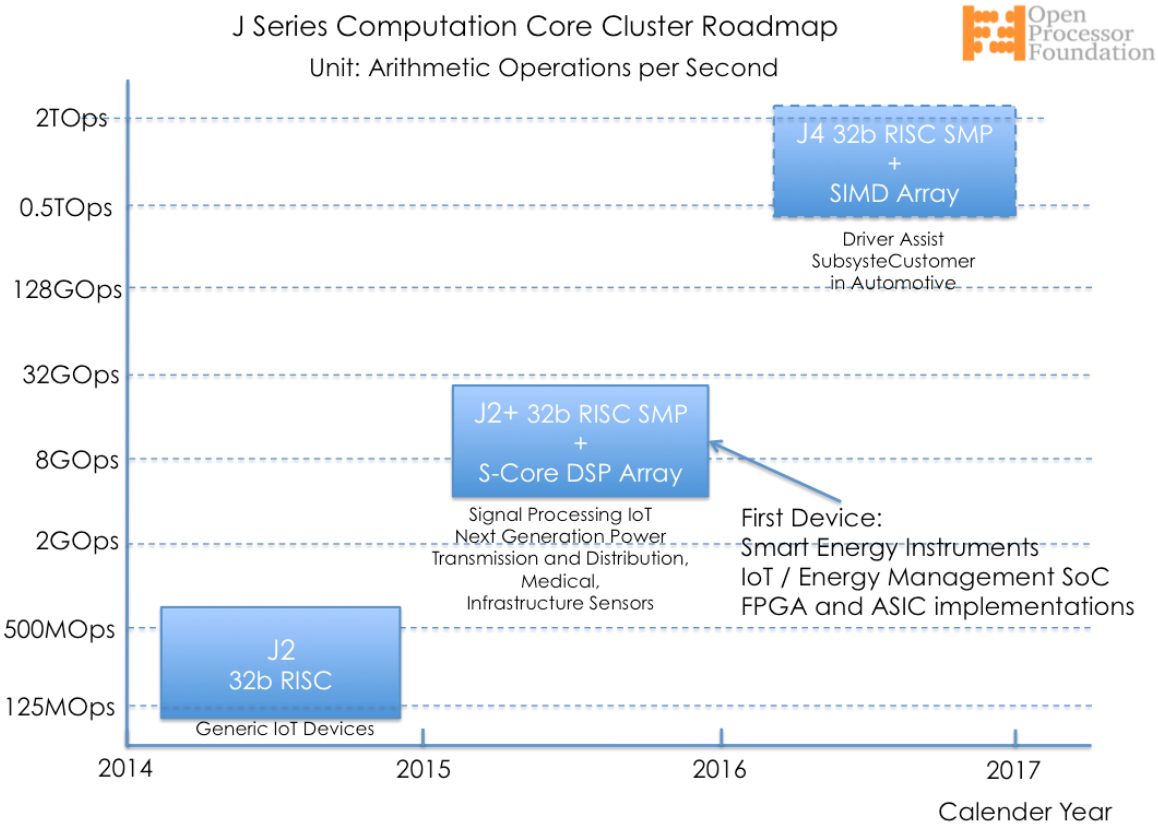

Abbildung 2.2: Ursprüngliche Roadmap für die J-Core Entwicklung [9]

Im selben Zeitraum starteten die Entwickler auch ihre Website[10].
Da die Website der OPF seit dem initialen RTL-Release des J2-Core nicht mehr aktualisiert wurde, gehe ich davon aus, dass die OPF in diesem Sinne nicht mehr aktiv ist. Das Entwicklerteam konzentriert sich nur auf die J-Core Entwicklung und hat im Zuge dessen auch die neue Website gestartet.

Die Aktivitäten auf der aktuellen J-Core Website enden Mitte 2016 ebenfalls. Jedoch wurde die Roadmap für die Entwicklung bereits angepasst:

- J1 wurde für Ende 2016 hinzugefügt. Dieser Prozessor soll eine reduzierte Version des J2-Core werden, wo Hardware-Multiplizierer sowie ein Großteil der Speicherlogik entfernt werden.

- Für 2017 wurde der J3-Core eingeplant. Die letzten Patente für den SH-3 sind im Dezember 2015 ausgelaufen. Dieser Prozessor beinhaltet bereits eine MMU und eine 64-Bit Floating Point Unit (FPU).

[9] http://0pf.org/j-core.html
[10] http://j-core.org

- J4, das Äquivalent zum weit verbreiteten SH-4 Prozessor, sollte 2017 (bzw. laut der aktuellen Website[11] in 2018) veröffentlicht werden. Der Prozessor wurde primär für Multimedia-Anwendungen konzipiert und beinhaltet neben neuen Instruktionen, eine verbesserte FPU. Die größte Verbesserung beim J4 ist der Umbau zu einem superskalaren Prozessor. Superskalarität bedeutet, dass mehrere Instruktionen pro Taktzyklus ausgeführt werden können.
- SH-5 wäre die letzte SuperH-Generation, die dann noch nicht als J-Core umgesetzt wurde. Die Entwickler planen jedoch einen J64. Bei diesem Prozessor wollen die Entwickler ihren eigenen Ansatz eines 64-Bit Prozessors auf Basis des J4 umsetzen.

2.1.3 J2 Features

Die erste J-Core-Generation, der J2, ist kompatibel mit der SH-2-ISA. Genau wie ein SH-2-Prozessor ist auch der J2 ein NOMMU[12]-Prozessor. Implementiert ist der Prozessor als Harvard Architektur (getrennter Instruktion- und Daten-Bus) mit einer 5-stufigen Pipeline.

Alle SH-2-Instruktionen wurden übernommen. Neben einer Compare and Exchange (CMPXCHG)-Instruktion, wurden noch zusätzlich zwei Instruktionen vom SH-3 portiert, um Bit-Shifts in C effizient abarbeiten zu können. Unterstützt wird der Prozessor durch einen Speicherkontroller mit bis zu 256 Megabyte Low Power Double Data Rate (LPDDR)-Speicher.

Der Aufbau des J2 wurde so einfach wie möglich umgesetzt. Laut den J-Core-Entwicklern war der Instruktions-Decoder auch das einzig komplexe Element der gesamten Architektur. Um eine Anpassung dieses Instruktions-Decoders zu erleichtern, stellen die Entwickler einen eigenen Generator zur Verfügung. Geschrieben wurde dieser in der Programmiersprache Clojure[13] und es wird zusätzlich Java und eine spezielle Java Bibliothek namens vMAGIC[14] benötigt.

[11] http://j-core.org/
[12] NOMMU bedeutet das jede Adresse im System eine physikalische Adresse ist
[13] https://clojure.org/index
[14] http://sourceforge.net/projects/vmagic/

Ausgangsmaterial für den Generator bildet eine Tabelle mit den J2 Instruktionen. Eine Readme-Datei informiert zwar darüber was installiert werden muss und wie der Generator gestartet wird, allerdings gibt es keine Informationen zur Tabelle mit den Instruktionen. Einträge in dieser Tabelle sind schwer nachvollziehbar und bedürfen zusätzlicher Einlesearbeit. Der Befehlssatz[15] vom SuperH-2, sollte für diesen Zweck nützlich sein.

Implementiert wurde der J2 in VHDL. Die Entwickler begründen ihre Entscheidung darin, dass VHDL auch auf einem höheren Abstraktions-Level arbeitet als Verilog. Dies soll die Implementierung des J2 leicher lesbar machen. Um dies noch besser zu bewerkstelligen, wurde eine "Two-Process"-Design-Methode verwendet, die auch als Standard bei der European Space Agency (ESA) eingesetzt wird [12].

2.2 V-scale / RISC-V

2.2.1 RISC-V-Architektur

RISC-V ist eine freie und offene ISA [11], [9]. Ursprünglich startete das Projekt 2010 in der Computer Science Division an der University of California, Berkeley. Ziel war es, die Forschung von Computerarchitekturen und die Lehre zu unterstützen. Mittlerweile etabliert sich die offene RISC-V-Architektur als Standard auch in der Industrie. Kontrolliert wird die RISC-V-Entwicklung durch die RISC-V Foundation [16].

RISC-V verbreitet sich dank seiner großen Community. Mitwirkende stammen aus den verschiedensten Anwendungsbereichen. Aus diesem Grund wurde der RISC-V-Befehlssatz auch sehr modular ausgelegt. [13]

Der Basis-Integer-Befehlssatz unterstützt eine Bitbreite von 32-, 64- und 128-bit. Ihre Erfahrung hat den Entwicklern gezeigt, dass ein unbehebbarer Fehler in der Befehlssatzentwicklung ein zu kleiner Adressraum ist. Daher wurde die 128-bit Version eingeplant, auch wenn diese bis heute noch nicht definiert wurde. Diese Basisversion wird je nach Bitbreite RV32I, RV64I oder RV128I genannt.

[15] http://www.shared-ptr.com/sh_insns.html
[16] https://riscv.org/

Mit den folgenden Erweiterungen kann der RISC-V-Befehlssatz dann für spezielle Anwendungen erweitert werden:

- "M" Integer-Multiplikation und Division
- "A" Atomare Instruktionen
- "F" Single-Precision Floating-Point Operationen
- "D" Double-Precision Floating-Point Operationen
- "C" Komprimierte 16-bit Instruktionen

Die ersten vier Erweiterungen zusammen mit der Basis-Integer-Erweiterung (IMAFD) werden mit der Bezeichnung "**G**" abgekürzt [14].
Für verbesserte Leistung, Codegröße und Energieeffizienz kann die Erweiterung für komprimierte Instruktionen ("**C**") verwendet werden. Im Gegenzug erhöht diese Erweiterung die Hardwarekomplexität. Die Entwickler erklären in Kapitel 14 [11], dass typisch 50%-60% der Instruktionen auf 16-bit komprimiert werden können, und somit 25%-30% Code gespart werden kann. Dieses Konzept ist vergleichbar mit dem Befehlssatz der SuperH-Architektur.

Die restlichen Erweiterungen sind spezieller und eignen sich wohl eher für spezielle Anwendungsfälle:

- "Q" Quad-Precision Floating-Point Operationen
- "L" Dezimal Floating-Point Operationen
- "V" Vektor Operationen
- "B" Bitmanipulationen
- "T" Transaktionaler Speicher
- "P" Packed-Single Instruction Multiple Data (SIMD) Operationen

Bei neu entwickelten Befehlssätzen (ISA) mangelt es oftmals an CPU-Designs und passender Software. Dieses Problem verhindert eine leichte Verbreitung des Befehlssatzes. Letztlich können private und auch kommerzielle Projekte, oft nicht von Grund auf neu entwickelt werden.
Frei zur Verfügung stehende Problemlösungen, erleichtern somit die ersten Schritte mit einem neuen Befehlssatz. Wie in Tabelle 2.1 zu sehen, kann man bei der RISC-V-Architektur auf eine Vielzahl an Implementierungen zurückgreifen.

Core	Hauptentwickler	Lizenz	Details
Rocket	UCB BAR[17], SiFive	BSD	- RV32G & RV64G - Chisel[18] - 5-stage In-Order pipeline - UserSpec 2.0, Privileged 1.9.1
BOOM	UCB BAR	BSD	- RV64G - Chisel - Superscalar Out-of-Order Machine - optimiert für ASICs
Sodor	UCB BAR	BSD	- RV32I - Chisel - 1/2/3/5-stage pipeline - "bus" based micro-coded version - UserSpec 2.0, Privileged 1.7
Z/V-scale	UCB BAR	BSD	- RV32IM - Chisel & Verilog - für Mikrocontroller entwickelt - 3-stage Single-Issue In-Order - Machine & User Privilege Modes
PicoRV32[19]	Clifford Wolf	ICS	- RV32IMC - Verilog - auf Größe optimiert
ORCA[20]	VectorBlox	BSD	- RV32IM - VHDL - optimiert für FPGAs - Bus-Typ und Pipeline konfigurierbar
PULPino[21]	ETH Zürich, Università di Bologna	SHL	- RV32IMC - SystemVerilog - für Signalverarbeitung (DSP-Erweiterungen) - Independent 4-stage pipeline
mriscv[22]	OnChipUIS	MIT	- RV32I - Verilog

Tabelle 2.1: Verschiedene RISC-V Implementierungen

[17] https://github.com/ucb-bar Berkeley Architecture Research
[18] https://chisel.eecs.berkeley.edu/
[19] https://github.com/cliffordwolf/picorv32
[20] https://github.com/vectorblox/orca
[21] https://github.com/pulp-platform/pulpino
[22] https://github.com/onchipuis/mriscv

Von diesen RISC-V-Implementierungen wurde **V-scale** für die Arbeit ausgewählt. V-scale ist eine sehr einfache CPU (3-stage) und wurde speziell für den Einsatz als Mikrocontroller entwickelt. Der einfache Aufbau erlaubt es, die Implementierung ohne große Schwierigkeiten anzupassen.

Ein weiterer wichtiger Punkt ist die verwendete Sprache für die Implementierung. Verilog wurde bereits im *mosart*MCU-Projekt verwendet und war daher die am besten geeignete Hardwarebeschreibungssprache. Bereits vorhandene Arbeiten mit dem V-scale im *mosart*MCU-Projekt legten es nahe, speziell diesen Softcore zu verwenden.

Die verschiedenen Sodor-Cores von UCB BAR wären auch eine gute Alternative gewesen. Sodor wurde sogar speziell als Lehrplattform entwickelt, jedoch versucht die UCB BAR hiermit, ihre eigene Hardwarebeschreibungssprache Chisel weiter zu verbreiten.

Die Implementierungen Rocket und BOOM sind ebenfalls nur in dieser speziellen Hardwarebeschreibungssprache verfügbar. Darüber hinaus sind diese Implementierungen und auch PULPino etwas komplexer als die Übrigen und somit schwieriger erweiterbar. Infolgedessen wurden nur kleinere Implementierungen für diese Arbeit berücksichtigt.

Im Verlauf der Bus-Evaluierung (Kapitel 3) wurde Wishbone als Bus-System für diese Arbeit gewählt. Entgegen dieser Auswahl wurde aber nicht die RISC-V Implementierung ORCA oder PicoRV32 verwendet. Bei diesen beiden Implementierungen kann die Bus-Schnittstelle konfiguriert werden und Wishbone wäre eine konfigurierbare Option.
Des Weiteren entfällt somit mriscv als Implementierung. Dieses Projekt nutzt Advanced eXtensible Interface (AXI) als Busanbindung und ist darauf ausgelegt als vollständiger Mikrocontroller verwendet zu werden. Dementsprechend sind auch die Konfigurationsmöglichkeiten des Prozessors nur begrenzt.

V-scale wurde aber gewählt, da diese Implementierung in der bevorzugten Hardwarebeschreibungssprache Verilog geschrieben wurde. Außerdem ermöglichte es die Entwicklung der Wishbone-Schnittstelle inklusive der benutzerdefinierten Signale, sich mit dem Wishbone-Bus vertraut zu machen.
Zusätzlich konnte ein Betriebssystem genutzt werden, welches im Verlauf des *mosart*MCU-Projekts für die RISC-V Architektur entwickelt wurde. Gleichwohl könnten die V-scale-Kerne im System ohne viel Aufwand durch ORCA oder PicoRV32 Kerne ersetzt werden.

2.2.2 V-scale Features

Wie bereits erwähnt, wurde Z-scale beziehungsweise die Verilog-Version V-scale, speziell für Mikrocontroller und eingebettete Systeme entwickelt. Die RISC-V-Architektur wurde dementsprechend minimalistisch implementiert.

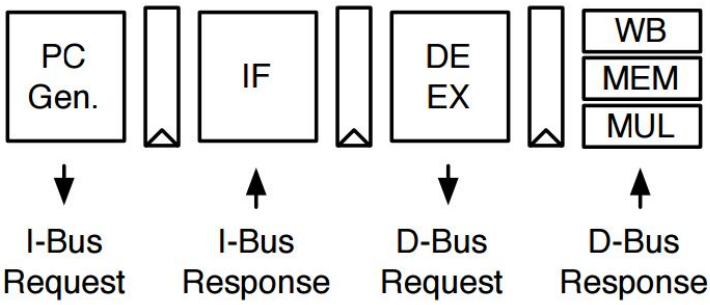

Abbildung 2.3: V-scale Pipeline [2]

Ein V-scale-Softcore besitzt eine 3-stufige Single-Issue-In-Order-Pipeline mit getrennten Insktruktions- und Datenbus (Abbildung 2.3). Diese beiden Bus-Anbindungen sind 32-Bit breite Highly Advanced System Transport Interface (HASTI)-Busse. HASTI ist die Advanced High-performance Bus (AHB)-Lite Implementierung der University of California, Berkeley (UCB). AHB-Lite ist einer von mehreren Bussen innerhalb der AMBA-Spezifikation [15]. Weitere Informationen zu dieser Architektur sind in Kapitel 3.2.1 zu finden.
Komplexere RISC-V Implementierungen von UCB, unterstützen auch noch weitere Busse von AMBA. Not A STandard Interface (NASTI) entspricht AXI und der kleinste Bus Peripheral Oriented Connection Interface (POCI) entspricht dem Advanced Peripheral Bus (APB).

2.3 Zusammenfassung der Prozessor Auswahl

Der J2-Core mit SuperH-Architektur wurde zuerst für diese Arbeit evaluiert. Es wurde geklärt, ob der Softcore sich leicht im *mosart*MCU-Projekt integrieren lässt und wie gut Anpassungen für die weiteren Arbeiten möglich sind.

Genauere Informationen zum J2-Core sind im vorherigen Abschnitt 2.1 zu finden. Die Liste im Anschluss stellt eine, für diese Arbeit interessante, Spezifikation des J2-Core dar:

- 32-Bit RISC-Mikroprozessor mit 16-Bit Instruktionen. (Die hohe Codedichte minimiert den Speicherverbrauch)
- Komplexer Instruktionsdekodierer
- 5-stufige Pipeline
- DDR Speicherkontroller
- Open Source Berkeley Software Distribution (BSD) Lizenz für den VHDL J2-Core. (Getestet auf Xilinx FPGAs)
- Existierende Compiler und Betriebssystem Unterstützung (uClinux, QNX ..)
- Nur wenig bis gar keine Dokumentation

Einige von diesen Eigenschaften waren problematisch. Die Architektur, aufgrund der 5-stufigen Pipeline und dem komplexen Instruktions-Dekoder, war vergleichsweise schwer zugänglich. Hier kommt noch die umfangreiche Toolchain hinzu, die allein durch den zusätzlichen ISA-Generator nötig ist.

Das größte Problem war jedoch die fehlende Dokumentation des gesamten Softcores. Alle Funktionen zu verstehen, beziehungsweise den J2 an eigene Bedürfnisse anzupassen, stellte sich als zu aufwendig heraus. Auch das verwendete Bussystem ist nicht dokumentiert. Genau am Bussystem musste ich meine Arbeit jedoch ansetzen. Aus diesen Gründen wurde der Prozessor für die Arbeit gewechselt.

Als Alternative wurde **V-scale** mit der RISC-V-Architektur gewählt. Folgende Liste zeigt auch für diesen Softcore die Spezifikationen, die für diese Arbeit entscheidend sind:

- 32-Bit RISC-Mikroprozessor
- 3-stufige Single-Issue In-Order Pipeline
- Machine & User Privilege Modes
- I-Bus und D-Bus sind 32-Bit HASTI (AHB-Lite) Busse
- Open Source BSD Lizenz für Chisel und Verilog Core
- Existierende Compiler, Software Tools und Betriebssystem Unterstützung (Linux, FreeBSD, NetBSD)
- Kompakte Implementierung der Architektur
- Bereits im *mosart*MCU-Projekt in Verwendung

Im Gegensatz zu den Problemen mit dem J2-Core, gab es mit V-scale keine großen Schwierigkeiten. Zum einen ist die komplette RISC-V-Spezifikation ausführlich dokumentiert und zum anderen wurde die Implementierung sehr übersichtlich umgesetzt und mit Kommentaren versehen. Außerdem ist die 3-stufige Pipeline weniger komplex im Vergleich zur 5-stufigen Pipeline des J2-Core.
Weitere Vorteile des V-scale sind die verwendete Hardwarebeschreibungssprache (Verilog), die einfache Busanbindung sowie die freien Bereiche in der RISC-V-Spezifikation, um eigene Erweiterungen zu implementieren.

Kapitel 3

Bus Evaluation

Der zweite Teil der Arbeit behandelt die Evaluierung von unterschiedlichen Bussystemen zum Verbinden mehrerer Prozessorkerne in einem SoC. Mehrere Faktoren sind für die folgende Evaluierung entscheidend. Das Bus-System sollte möglichst leicht implementiert werden können und auch die Möglichkeit bieten, den Bus zu erweitern, um eine Arbitrierung mittels dynamischer Prioritäten zu ermöglichen.
Die Implementierung dieser Arbeit beschränkt sich auf eine Topologie, den Standard-Übertragungsmodus sowie eine Arbitrierung abhängig von der Priorität eines Tasks. Für zukünftige Arbeiten sind jedoch alle unterstützten Topologien, unterschiedliche Übertragungsmodi sowie die Arbitrierung von Interesse.

Weiters wird noch Network-on-Chip (NoC) als moderne Alternative gegenüber herkömmlichen Bus-Architekturen behandelt. Ein NoC bringt viele Vorteile mit sich, sinnvoll ist der Einsatz jedoch erst ab einer gewissen Komplexität des Gesamtsystems. Zum Beispiel bei einer hohen Anzahl an Komponenten oder wenn sich die Komponenten oftmals ändern und das Verbindungsnetzwerk nicht immer angepasst werden kann. Für die Implementierung dieser Arbeit wird eine herkömmliche Busarchitektur bevorzugt.

Das Kapitel beginnt mit allgemeinen Informationen zu herkömmlichen Bussystemen und einem Abschnitt über NoCs. Die nächsten Abschnitte beschreiben unterschiedliche Topologien, Strategien zur Arbitrierung sowie unterschiedliche Übertragungsmodi von Bussystemen. Abschließend werden ausgewählte Bus-Systeme beschrieben, verglichen und die Auswahl diskutiert.

3.1 Allgemeines

Die Elektronikindustrie entwickelt seit Jahren immer kleinere und leistungsfähigere Schaltkreise. Diese Miniaturisierung ermöglicht es, komplexe Systeme, bestehend aus verschiedensten Komponenten, zusammen auf einem Chip unterzubringen. Ein sogenanntes SoC besteht also häufig nicht nur aus einem Prozessor mit dazugehöriger Peripherie, sondern aus einer großen Anzahl an Rechenkernen. Somit werden bei SoC-Designs die chip-internen Verbindungsnetzwerke immer wichtiger und bringen einige Herausforderungen für die Chip-Designer mit sich.

Die hohe Anzahl an Komponenten macht es unmöglich, jede Verbindung im System, einzeln zu implementieren. Zusätzlich ist es notwendig, immer mehr Komponenten wiederzuverwenden, um eine wettbewerbsfähige Entwicklungszeit für das gesamte SoC zu ermöglichen.
Die Verbindung der Komponenten erfolgt daher über standardisierte Bus-Systeme. Hersteller von IP-Cores veröffentlichen ihre Produkte häufig schon mit verschiedenen Standard-Bus-Schnittstellen, was eine Implementierung im eigenen SoC vereinfacht.

Eine weitere Entwicklung in diesem Gebiet ist der Einsatz von mehreren Bussystemen in einem SoC. Zum Beispiel könnte ein Hochleistungsbus mit niedriger Latenz die Prozessorkerne miteinander verbinden, ein Bus mit hoher Bandbreite den Speicher im System anbinden und ein zusätzlicher Bus für die gesamte Peripherie eingesetzt werden. Diese Methode kann Ressourcen sparen, da nicht alles mit einem komplexen (breiten) Bus angebunden werden muss. Andererseits müssen zusätzliche Brückenkomponenten implementiert werden, die Signale von einem Bussystem ins andere übersetzen.

Network on Chip

NoCs sind der nächste Evolutionsschritt für noch komplexere SoC Designs. Diese Systeme definieren sich durch eine Trennung der Verbindungen in einzelne Schichten. Beispielsweise gibt es eine rein physische Schicht, eine Transportschicht und eine darüberliegende Transaktionsschicht. Der Aufbau orientiert sich hier stark am Open Systems Interconnection (OSI)-Modell von Computernetzwerken.

Im Endeffekt wird die gesamte "Bus"-Logik von den einzelnen IP-Core, in das Verbindungsnetzwerk selbst verschoben. Systemkomponenten müssen nur noch ihre Daten in standardisierte Pakete bestehend aus Daten und Header zusammenfassen und an die standardisierte Schnittstelle übergeben. Die physische Schicht kümmert sich um die Übertragung der einzelnen Bits und

Bytes entsprechend der lokal verfügbaren Verbindung. Übertragung und Integrität der einzelnen Pakete werden von der Transportschicht behandelt, während die Transaktionsschicht ganze Datenströme über Protokolle an die Transportschicht weiter gibt.

Gegenüber herkömmlichen Bussystemen bietet diese Architektur viele Vorteile. Die Flexibilität und Erweiterbarkeit ist hoch, da die Anbindung von Komponenten nur über standardisierte Schnittstellen erfolgt. Auch spezielle Anforderungen wie unterschiedliche Clocks im System können von den Paketen überwunden werden. Zusätzliche Fehlerbehandlung oder Sicherheitsaspekte können ebenso in die Netzwerkkomponenten integriert werden.

Auch wenn herkömmliche Bussysteme stetig weiterentwickelt werden, so konzentriert sich der Großteil der aktuellen Forschung auf NoCs. Im Zuge dessen wird seit dem Jahr 2007 regelmäßig das IEEE International Symposium on Networks-on-Chip abgehalten.
Die Forschungen verfolgen verschiedenste Ansätze um bessere NoCs zu entwickeln oder um NoCs in immer mehr Anwendungsgebieten einzusetzen. Zum Beispiel behandelt [16] die Energieeffizienz beim Einsatz auf einem FPGA. In der Arbeit [17] wird der spezielle Ressourcenbedarf von Soft-Core-NoCs und Hard-Core-NoCs miteinander verglichen.

3.1.1 Bustopologien

On-Chip-Bussysteme können, bezogen auf die Topologie, in unterschiedliche Klassen eingeteilt werden:

Geteilter Bus [18] - Die einfachste Möglichkeit, ein Bussystem aufzubauen, ist die Verbindung von allen Komponenten über eine gemeinsame Bus-Leitung. Da eine aktive Verbindung den restlichen Bus für alle anderen Komponenten im System blockiert, ist eine Arbitrierung erforderlich. Folgende Vorteile und Nachteile sind bei dieser Topologie zu erwarten:

+ Einfache Erweiterbarkeit, Aufbau und Implementierung
+ Günstig da wenig Leitungen (Fläche) benötigt wird
+ Bis auf die Arbitrierung keine aktiven Buskomponenten notwendig
- Hohe Auslastung pro Bus-Leitung
- Hohe Latenz
- Niedrige Bandbreite
- Hoher Energieverbrauch

Die hohe Latenz und die niedrige Bandbreite wird ab einer großen Anzahl an Komponenten problematisch. Speziell bei der Latenz ist die Art der Arbitrierung entscheidend. Ein Überblick folgt am Ende dieses Abschnitts.

Hierarchischer Bus [18] - Diese Architektur ist, wie im allgemeinen Abschnitt bereits erwähnt, die Antwort auf immer komplexer werdende Systeme. Mehrere geteilte Busarchitekturen werden hier mit Bridge-Komponenten zu einer Hierarchie zusammengeschlossen. Jede Komponente wird im Bezug auf die benötigte Bandbreite auf geeignetem Level innerhalb der Hierarchie platziert. Zum Beispiel werden alle langsamen SoC-Komponenten auf einem gemeinsamen Bus platziert, welcher über eine Bridge mit einem schnelleren Bus verbunden ist. Schnelle Komponenten auf diesem werden somit nicht von den langsamen Komponenten blockiert. Allerdings blockiert eine Transaktion über eine Bridge hinweg beide Bus-Level.
Kommerzielle Bussysteme mit dieser Architektur sind zum Beispiel AMBA [15] und CoreConnect [19].

+ Hohe Bandbreite im Vergleich zum geteilten Bus aufgrund:
    * Geringe Auslastung pro Bus-Level/Bus-Leitung
    * Mögliche parallele Transaktionen auf unterschiedlichen Bus-Leveln
    * Mehrere Transaktionen zwischen Bus-Level mittels Pipelining
- Overhead bei Transaktionen zwischen zwei Bus-Leveln

Ringbus / **Daisy Chain** [20] - Eine weitere gängige Architektur für die Kommunikation innerhalb eines SoC ist die Zusammenschaltung der Komponenten zu einem Ring. Üblicherweise wird dies über ein Tokenweitergabe-Protokoll implementiert. Ein Token (Datenpaket) bewegt sich bei diesem Protokoll im Ring. Dieses spezielle Datenpaket berechtigt eine Komponente, eine neue Transaktion zu initiieren. Der Token wird an die nächste Komponente im Ring weitergegeben, sobald die Transaktion abgeschlossen wurde.

Crossbar-Switch [20] - Ein Crossbar-Switch ist bereits die Weiterentwicklung einer einfachen Bustopologie. Es werden je nach Bedarf Punkt-zu-Punkt-Verbindungen zwischen zwei Komponenten aufgebaut. Die eigentliche Kommunikation über den Bus kann somit ohne weitere Störungen erfolgen. Wie bei anderen Bustopologien entscheidet ein Arbiter, im Bezug auf die Eingangssignale, welche Verbindungen bevorzugt werden.
Ein Crossbar-Switch ist somit das Bindeglied zwischen einer herkömmlichen Bus-Architektur und einem NoC. Die gesamte Bus-bezogene Logik wurde von den einzelnen IP-Cores entfernt und wird vom aktiven Crossbar-Switch umgesetzt. Im Falle eines NoC übernehmen die Router diese Funktion.

Pro Crossbar-Switch wird nur eine Bus-Architektur unterstützt, da die Komponenten wie bei einem herkömmlichen Bus angebunden sind. Aus diesem Grund kann das System nicht an die benötigte Bandbreite jeder Komponente angepasst werden.

+ Hohe Bandbreite
+ Parallele Verbindungen möglich
- Hoher Bedarf an Chipfläche

3.1.2 Arbitrierung

Die Art der Arbitrierung definiert, wie der Zugriff auf einen geteilten Bus erfolgt. Notwendig ist dies zum Beispiel bei simultanen Zugriffen auf Ressourcen im System. Zu diesem Zeitpunkt muss entschieden werden, welche Verbindung zuerst aktiv geschaltet werden soll. Wird die Arbitrierung zentral von einem eigenen Modul gesteuert dann ist dieses Modul ein sogenannter Arbiter.
Bei dezentraler Arbitrierung wird kein eigenständiges Modul benötigt, sondern es kümmert sich jede Komponente selbst um den richtigen Zugriff. Der folgende Abschnitt beschreibt einige gängige Arbitrierungsmethoden:

Statische Priorität [21] - Ein zentraler Arbiter empfängt Anfragen von jedem Master im System und erlaubt dem Master mit der höchsten Priorität seine Transaktion durchzuführen. Je nach Implementierung kann eine neue Anfrage mit höherer Priorität eine aktive Transaktion unterbrechen.

Zeitmultiplexverfahren - Time Division Multiple Access (TDMA) [22] - Beim Zeitmultiplexverfahren ist für jeden Master im System ein Zeitslot in einem periodischen Zeitfenster festgelegt. Ein Master kann also in jeder Periode in genau seinem Zeitslot eine Transaktion durchführen. Zusätzliche Techniken ermöglichen es unbenutzte Zeitslots zum Teil trotzdem zu verwenden.

Lotterie [21] - Eine zentrale Lotterie sammelt spezielle Lotterie-Tickets von Mastern, die eine Transaktion durchführen möchten und wählt daraus ein Ticket aus. Jeder Master besitzt eine, statisch oder dynamisch zugewiesene, Anzahl an diesen Lotterie-Tickets. Ein Master mit vielen Tickets erhält also viel wahrscheinlicher die Erlaubnis, den Bus zu benutzen.

Tokenweitergabe [23] - Diese Methode wird bei Ringbussen eingesetzt. Wie bereits in Abschnitt 3.1.1 beschrieben, wird hier ein spezielles Token eingesetzt. Wenn eine Komponente das Token erhält, besitzt es die Erlaubnis, eine Transaktion durchzuführen. Danach wird der Token an die nächste Komponente im Ring weitergereicht.

Codemultiplexverfahren - Code Division Multiple Access (CDMA) [24] - Beim Codemultiplexverfahren werden mehrere Datenströme zusammen übertragen. Diese Datenströme belegen eigene Bereiche im Spektrum und stören sich dadurch nicht gegenseitig. Dieses Verfahren ist schwierig zu implementieren, da bei den Bus-Anbindungen komplexe Systeme notwendig sind, um die Daten zu Demodulieren beziehungsweise um den gemeinsamen Datenstrom zu modulieren.

3.1.3 Übertragungsmodi

Bussysteme unterstützen zumeist mehrere Übertragungsmodi. Diese bestimmen wie Daten über den Bus übermittelt werden. Die Komplexität vom Bus und der benötigten Buskomponenten, werden stark von den unterstützten Übertragungsmodi beeinflusst. Dementsprechend beschränken sich einfache Busse, die zum Beispiel nur zum Anbinden von Peripherie eingesetzt werden, oft nur auf die simpleren Modi.

Datenflusssteuerung / **Handshaking** [18] - Diese grundlegende Technik beschreibt die Verwendung von Kontrollsignalen zwischen Master und Slave um den Datenfluss zu steuern. Beispiele für diese Technik sind das Bestätigen (Acknowledge) bei erfolgreichem Empfang von Daten, eine bestimmte Antwort wenn Daten erneut benötigt werden oder ein Slave der signalisiert wenn zurzeit keine Daten verarbeitet werden können.

Burstmodus / **Block-Transfer** [4] - Dieser Modus ermöglicht es eine große Menge an Daten möglichst effizient zu übertragen. Um dies zu bewerkstelligen wird ein ununterbrochener Strom an Datenpaketen übertragen, ohne dass für jedes Datenpaket der Bus initialisiert werden muss. Bei großen Datenmengen kann diese Einsparung bei jedem Datenpaket, die resultierende Bandbreite stark erhöhen. Im Allgemeinen ist der Burstmodus ein Spezialfall eines Block-Transfers. Zum Beispiel ist dieser Modus möglich wenn die Zieladressen der einzelnen Pakete in einer Reihe liegen. Ein Slave kann die benötigten Adressen selbst berechnen und somit müssen diese nicht für jedes Paket übertragen werden.

Parallelverarbeitung / **Pipelining** [4] - Bei vielen Bussystemen besteht eine Datenübertragung aus einer Adressphase gefolgt von einer Datenphase. Wenn die Adresse in der Datenphase nicht mehr benötigt wird, kann während die Daten übertragen werden, bereits eine neue Adressphase gestartet werden. Eine Datenübertragung startet somit noch bevor die vorangegangene Übertragung abgeschlossen wurde. Diese quasi parallele Übertragung von Daten erhöht ebenfalls die erzielbare Bandbreite des Systems.

Split-Transfer [25] - Bei Split-Transfer Bussen werden Datenübertragungen immer in mehreren Schritten durchgeführt. In der ersten Übertragung wird nur die Anfrage an den entsprechenden Slave übertragen. Die zweite Übertragung wird dann vom Slave aus gestartet und liefert die Daten (Antwort).
Zwischen den Übertragungen ist der Bus frei und kann für andere Übertragungen genutzt werden. Auch die Reihenfolge der Antworten muss nicht mit der Reihenfolge der Anfragen übereinstimmen und erhöht somit die Flexibilität. Nachteilig ist zu erwähnen, dass die Arbitrierung komplexer wird, da in diesen Systemen auch

die Slaves Übertragungen starten können.

Read-Modify-Write [4] - Mit diesem Begriff wird ein atomarer Prozessorbefehl bezeichnet, der Daten aus dem Speicher lädt (read), die Daten verändert (modify) und die veränderten Daten wieder an derselben Stelle in den Speicher schreibt (write). Bei diesem atomaren Befehl ist gewährleistet, dass keine anderen Komponenten auf diese Speicherstelle zugreifen.
Für Bussysteme existiert dieses Prinzip ebenfalls. Eine Read-Modify-Write (RMW)-Übertragung gewährleistet, dass in der Zwischenzeit keine anderen Buskomponenten auf die Zieladresse zugreifen können.

3.2 Bussysteme

3.2.1 AMBA - AHB, AXI, ASB, APB

Die erste AMBA-Spezifikation [15] wurde 1996 von ARM veröffentlicht. AMBA ist weit verbreitet, da es sich um einen offenen Standard handelt und somit keine Abgaben für eine Implementierung notwendig sind. In der Spezifikation sind mehrere Busprotokolle definiert, da AMBA auf eine hierarchische Bustopologie setzt. In der ersten Version bestand AMBA nur aus einem Systembus und einem Bus zur Anbindung der Peripherie. Über die Jahre sind weitere Protokolle zur Spezifikation hinzugefügt worden, um weitere Anwendungen für AMBA zu ermöglichen.
Die Spezifikation definiert die Protokolle, die Anbindung von Komponenten sowie die Brücken zwischen den Bus-Ebenen. Die Arbitrierung wird jedoch von der Spezifikation nicht vorgegeben. Nachfolgend ein Beispielaufbau sowie die Beschreibung der meisten AMBA Bus-Protokolle:

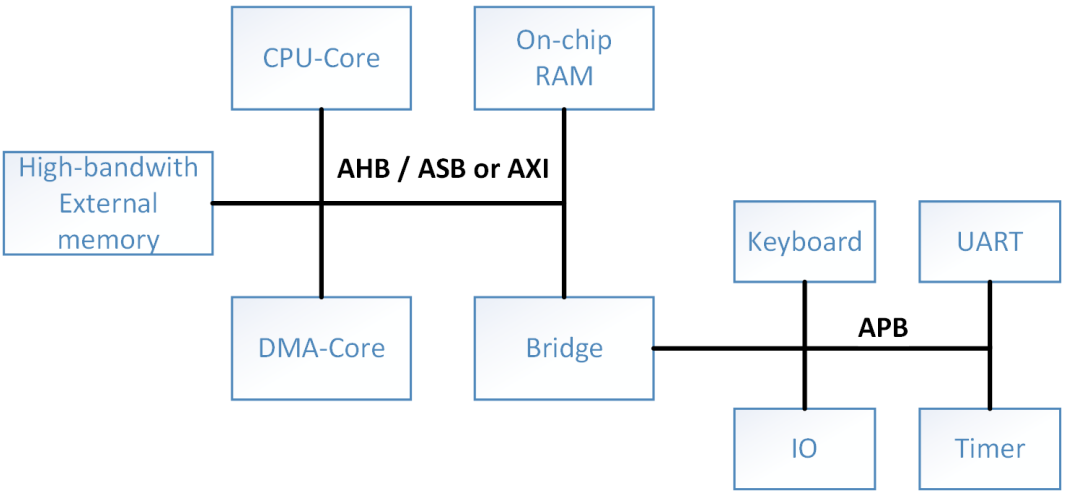

Abbildung 3.1: Typisches AMBA System

- Advanced System Bus (ASB) - Einfacher kostengünstiger Bus für mehrere Master, welcher einen Burstmodus und Pipelined-Transfer unterstützt.
- Advanced High-performance Bus (AHB) - Im Gegensatz zu ASB wurde AHB für Hochleistungssysteme entwickelt. Der Bus unterstützt hohe Taktfrequenzen, große Bus-Breiten und verschiedenste Übertragungsmodi. Es werden ebenfalls mehrere Master unterstützt und mit der dritten Version der Spezifikation wurde Multi-Layer-AHB eingeführt. AHB-Lite ist eine zusätzliche Alternative zu den anderen Protokollen und ist eine vereinfachte Version von AHB, für den Einsatz mit nur einem Master.
- Advanced eXtensible Interface (AXI) - Spezieller Bus für hohe Performance mit hoher Taktfrequenz. AXI arbeitet mit Punkt-zu-Punkt Verbindungen. In der vierten Version der AMBA Spezifikation wurde AXI Coherency Extensions (ACE) eingeführt. Eine Erweiterung von AXI um systemweite Speicherkohärenz zu gewährleisten.
- Advanced Peripheral Bus (APB) - Dieser Bus ist ausgelegt für Komponenten mit wenig Leistung und wenig Bandbreite. Die Bus-Bridge zu den leistungsfähigeren Bussen ist immer der Bus-Master und alle angeschlossenen Komponenten sind die Slaves.

Die ausführliche Dokumentation der einzelnen Protokolle trägt stark zur Verbreitung von AMBA bei. Genauere Informationen zu AMBA sind in den Referenzen [15, 26, 27, 28] zu finden.

3.2.2 CoreConnect

CoreConnect [19] ist ein hierarchisch aufgebautes on-chip Bussystem von IBM. Wie auch bei anderen Bussystemen wurde Wert darauf gelegt, dass Komponenten wiederverwendet werden können und somit die Entwicklungszeit stark verkürzt wird. IBM stellt CoreConnect ohne Gebühr für Entwickler von Komponenten oder SoCs zur Verfügung. Nachfolgend ein Beispielaufbau eines CoreConnect Systems und die Beschreibung der drei unterschiedlichen Busse von CoreConnect:

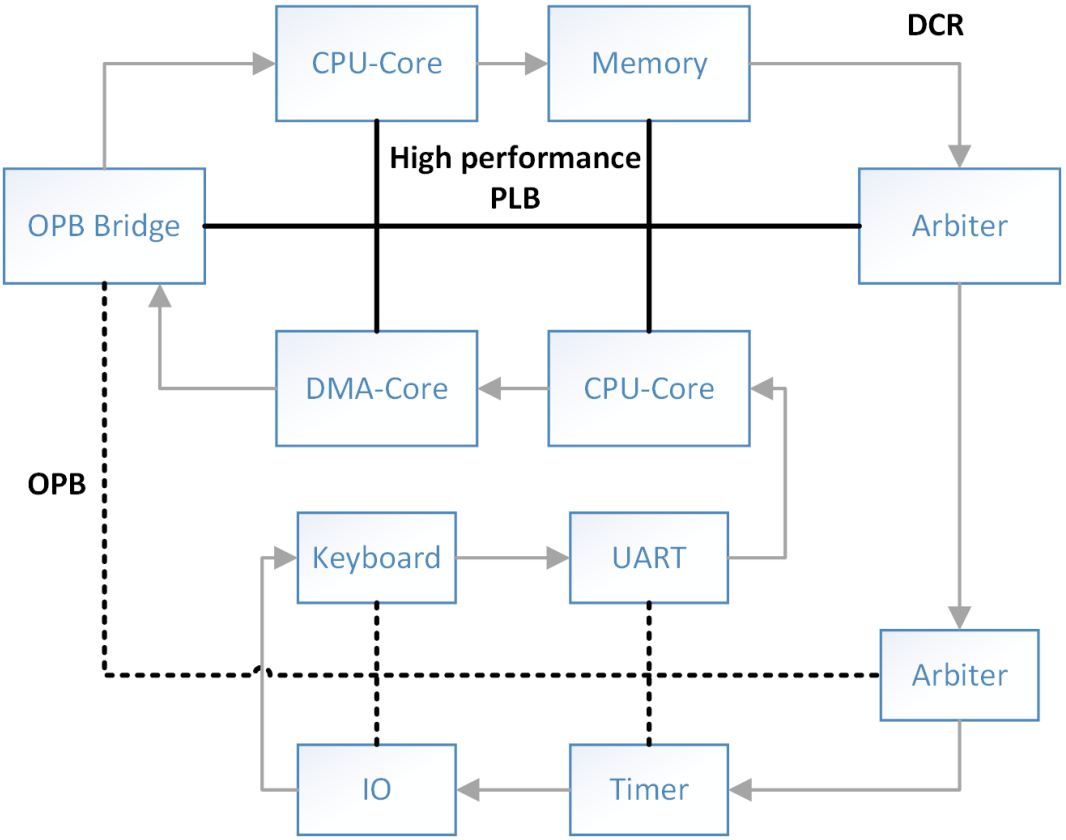

Abbildung 3.2: CoreConnect Bus-Architektur

Processor Local Bus (PLB) - Das ist der Systembus in der CoreConnect Architektur. Der Bus ist synchron, unterstützt mehrere Master und eine zentrale Arbitrierung regelt den Zugriff. Ausgelegt ist der Bus auf eine hohe Bandbreite und niedrige Latenzen. Getrennte Lese- und Schreib-Datenbusse ermöglichen simultanes Lesen und Schreiben.

On-Chip Peripheral Bus (OPB) - Speziell für langsame Peripherie die nur eine geringe Bandbreite benötigt, steht dieser Bus zur Verfügung. Die OPB Bridge arbeitet als Slave im PLB und als Master im lokalen OPB. Features vom OPB sind die synchrone Übertragung, dynamische Busbreiten, getrennte Adress- und Datenbusse, multiple Master etc.

Device Control Register Bus (DCR) - Dieser synchrone Ringbus arbeitet mit nur einem Master und ist ein alternativer Weg, um Statusinformationen beziehungsweise Konfigurationen an Komponenten weiterzuleiten. Der Bus ist relativ langsam, besteht aus einem 10-Bit Adressbus und einem 32-Bit Datenbus.

3.2.3 Avalon

Der Avalon-Bus [29, 30] wurde für FPGA-SoC-Designs auf Basis eines Nios Prozessors [31] von Altera entwickelt. Systeme auf Nios Basis sind die einzigen, auf denen Avalon eingesetzt werden kann. Diese System werden über den sogenannten SOPC-Builder (Sysem-On-a-Programmable-Chip) zusammengestellt.

Der Avalon-Bus ist synchron und unterstützt multiple Master. Adress-, Daten und Kontrollleitungen sind voneinander getrennt ausgeführt. Die Arbitrierung erfolgt im Bezug auf die Slaves und wird innerhalb des Avalon Busmoduls durchgeführt. Komponenten müssen nichts über die Arbitrierung wissen und können somit in Systemen mit einem Master oder in Systemen mit mehreren Mastern eingesetzt werden. Dementsprechend sind auch mehrere simultan aktive Verbindungen innerhalb Switch-Fabric möglich.
Dar SOPC-Builder von Altera generiert das Switch-Fabric automatisch mit der entsprechenden Logik, um alle von Avalon unterstützten Techniken zu ermöglichen. Zum Beispiel Datenpfad-Multiplexing[23], Dynamische Busbreiten, Adressdekodierung, Read and Write Streams etc.

[23] Die Datenübertragung von mehreren Quellen an einen Empfänger, oder umgekehrt.

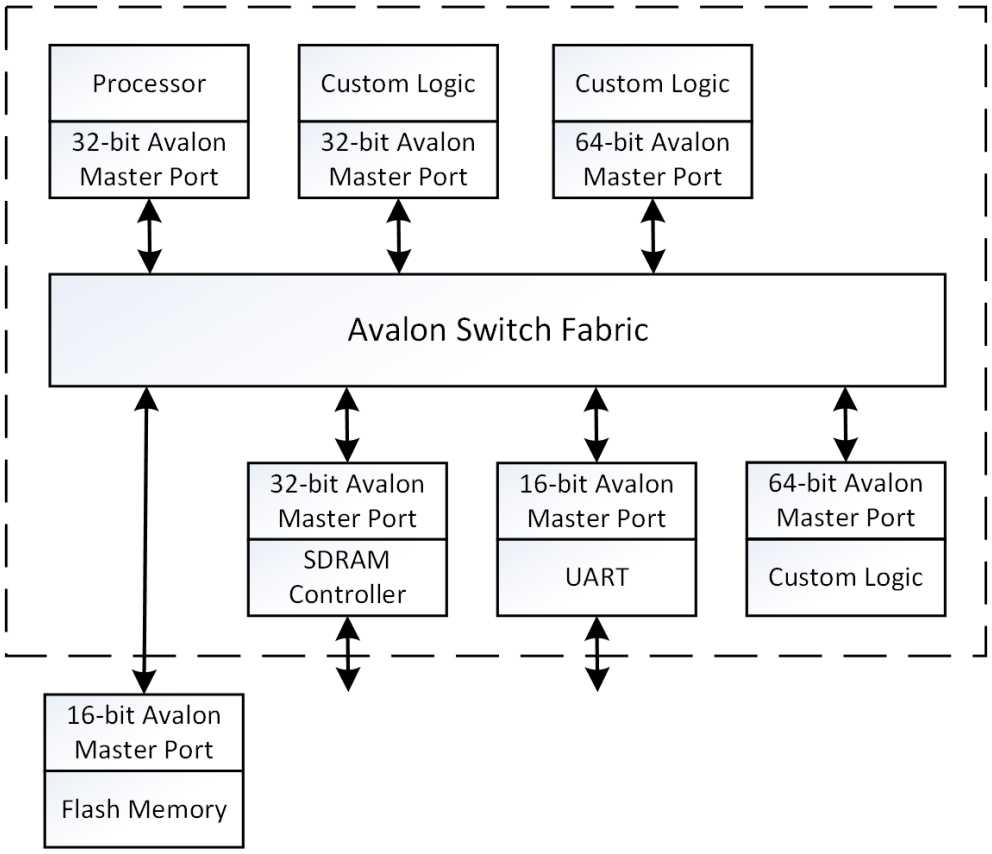

Abbildung 3.3: Avalon Bus-Architektur

3.2.4 STBus

STBus [3] ist ein von STMicroelectronics entwickeltes Kommunikationssystem für SoCs. Es beinhaltet Protkolle, Interfaces und Spezifikationen für das Implementieren des Verbindungsnetzwerkes. STBus verwendet die folgenden drei Busprotokolle:

- Typ I (Peripheral) - Ein einfaches synchrones Handshake Protokoll. Geeignet, um auf langsame Peripherie oder auf Register zuzugreifen.
- Typ II (Basic) - Effizienter als Typ I, da bereits Split-Transactions und Pipelining unterstützt wird. Lese- und Schreib-Operationen arbeiten mit unterschiedlichen Breiten bis zu 64 Bytes und unterstützten spezielle Operationen wie zB. Read-Modify-Write. Typischerweise wird dieses Protokoll beim Senden von Daten an externe Speichercontroller eingesetzt.
- Typ III (Advanced) - Das leistungsfähigste Protokoll unterstützt zusätzlich Out-Of-Order-Transaktionen sowie asymmetrische Kommunikation (Anzahl der Anfragen kann von der Zahl an Antworten abweichen).

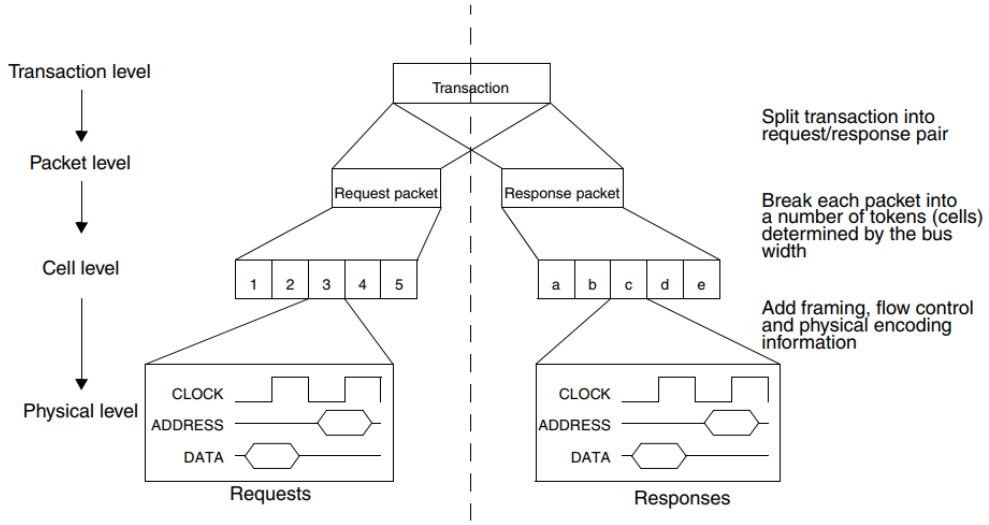

Abbildung 3.4: STBus Protokoll Ebenen [3]

Der Bus ist flexibel und erlaubt es, dass Master und Slaves mit unterschiedlichen Protokollen und Datenbreiten miteinander kommunizieren. Abbildung 3.4 zeigt die Ebenen über die diese Kommunikation verläuft. Ein gesendetes Paket resultiert in einem passenden Antwortpaket. Protokollkonverter in den Komponenten ermöglichen die Verwendung der unterschiedlichen Bus-Protokolle.
Verschiedenste Arbitrierungsmethoden können in einem STBus eingesetzt werden.

Neben den Mastern und Slaves im System gibt es weitere Komponenten im Verbindungsnetzwerk, die in 3 Arten eingeteilt werden können:

Switch / **Node** - Diese Komponente erledigt die Arbitrierung im System und leitet Anfragen und Antworten weiter.
Converter or Bridge - Konvertierung von einem Protokoll in ein anderes.
Size Converter - Konvertierung von Daten innerhalb eines Protokolls. Zusätzliche Buffer ermöglichen das Senden von Daten an eine Komponente mit geringerer Bandbreite.

Ein STBus-System kann mit unterschiedlichen Bustopologien aufgebaut werden (Geteilter Bus, Crossbar-Switch, ...)

3.2.5 Wishbone

Der Wishbone-Bus ist ein Open-Source-On-Chip-Bus zur Vernetzung von IP-Cores. Ursprünglich wurde Wishbone von der Silicore Corporation entwickelt. Im Jahr 2002 wurde Wishbone der Öffentlichkeit zur Verfügung gestellt und seither von OpenCores[24] weiterentwickelt.
Ziel von OpenCores ist es, Open-Source Hardwaredesigns frei zur Verfügung zu stellen. Neben der Wishbone Spezifikation wurde bereits eine Vielzahl an Prozessoren und IP-Cores mit Wishbone Anbindung entwickelt.

Wishbone wurde als logischer Bus konzipiert. Es wurden keine elektrischen Signale oder Bustopologien spezifiziert. Stattdessen umfasst die Spezifikation nur Buszyklen und beschreibt wann welche Signale ihren Pegel ändern. Diese offene Spezifikation ist gewünscht, um Entwicklern möglichst viel Freiheit bei der Implementierung zu bieten.

Alle Wishbone-Signale arbeiten synchron und die Busbreite kann beliebig gewählt werden. Die Spezifikation berücksichtigt zusätzlich sogenannte Tag-Signale. Diese ermöglichen es, Zusatzinformationen zu Adresse, Daten oder dem aktuellen Bus-Zyklus zu übertragen. Mit diesen Signalen ist es möglich, den Bus an die eigenen Anforderungen anzupassen ohne dafür von der Spezifikation abzuweichen.

Die Topologie des Wishbone Verbindungsnetzes kann in verschiedenster Form implementiert werden. Punkt-zu-Punkt, Geteilter Bus, Data Flow, Hierarchisch oder Crossbar-Switch. Ebenso ist die Arbitrierung nicht spezifiziert und kann je nach Anforderung integriert werden. Die folgenden Abbildungen 3.5, 3.6 und 3.7 aus der Spezifikation [4] zeigen drei Beispiele für mögliche Bustopologien:

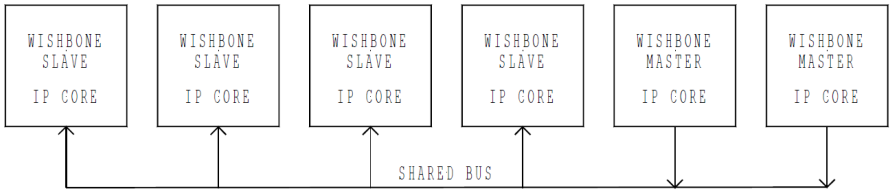

Abbildung 3.5: Wishbone - geteilter Bus [4]

[24] www.opencores.org

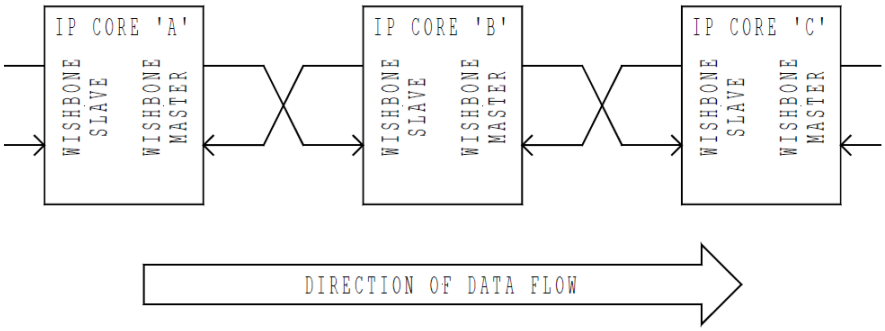

Abbildung 3.6: Wishbone - Data Flow Topologie [4]

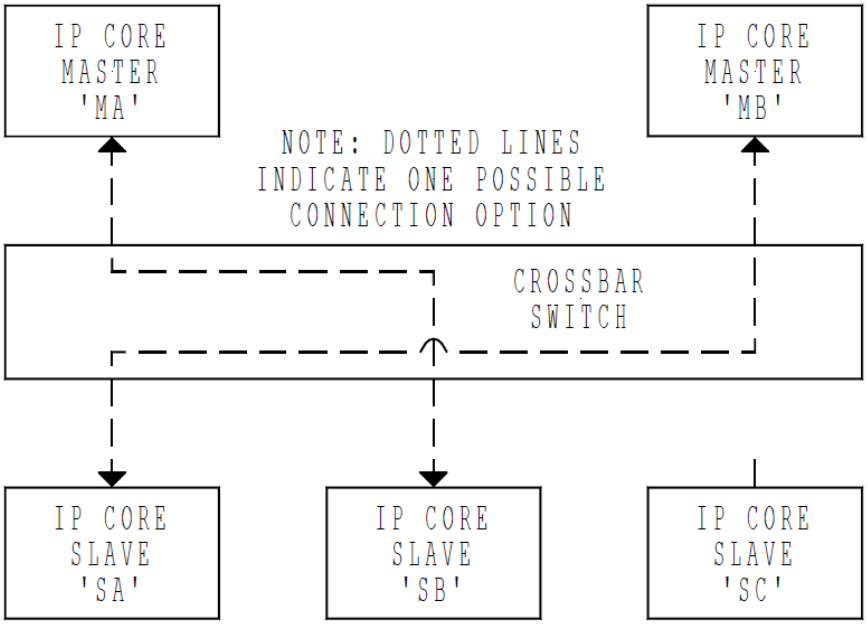

Abbildung 3.7: Wishbone - Crossbar-Switch Topologie [4]

Weitere Informationen zu Wishbone sind in Abschnitt 3.4 zu finden.

3.3 Bus Auswahl

Für die Auswahl eines geeigneten On-Chip Bus-Systems für die Implementierung dieser Arbeit, wurde der ausgewählte Prozessor berücksichtigt. Wie in Kapitel 2 bereits geschildert, fiel die Wahl auf V-scale als CPU im System. Diese Implementierung der RISC-V-Architektur steht in Verilog zur Verfügung und besitzt bereits eine HASTI / AHB-Lite Bus-Anbindung.

Infolgedessen wäre ein AMBA-System naheliegend, da der Prozessor ohne weitere Probleme ins System integriert werden könnte. HASTI bzw. AHB-Lite haben jedoch den Nachteil, dass nur ein Master unterstützt wird. Der Einsatz von dynamischen Prioritäten im System ergibt jedoch nur in einem MultiCore-System bestehend aus mehreren aktiven Mastern Sinn.

Um dieses Problem zu umgehen, müsste entweder ein Bus eingesetzt werden, der multiple Master unterstützt oder die Bustopologie müsste auf einen Crossbar-Switch eingeschränkt werden. Die einzelnen Implementierungen dieser Arbeit sollten jedoch wiederverwendbar sein. Dementsprechend wird ein anderes Bussystem gewählt, um alternative Topologien in Zukunft nicht auszuschließen.

Schlussendlich wurde **Wishbone** in Form eines **Crossbar-Switches** als Bussystem für die Implementierung ausgewählt. Die folgenden Punkte im Bezug auf Wishbone führten zu dieser Entscheidung:

- Wishbone ist open-source und eine Vielzahl an IP-Cores mit Wishbone Schnittstelle stehen frei zur Verfügung.
- Die Wishbone-Spezifikation unterstützt benutzerdefinierte Anpassungen. Ideal, um die geplanten Task-Prioritäten innerhalb des Bussystems umzusetzen.
- Wishbone unterstützt Multi-Master Systeme
- Die Arbitrierung kann bei Wishbone frei implementiert werden.
- Für die Topologie des Bussystems gibt es ebenfalls keine Einschränkungen. Alle Komponenten mit einer Wishbone-Schnittstelle zu versehen, ermöglicht es somit auch jederzeit, die Topologie zu wechseln oder Komponenten in anderen Systemen mit anderer Topologie einzusetzen.

Das resultierende Bussystem inklusive Arbitrierung, basierend auf den Task-Prioritäten, sollte einfach wiederverwendet werden und an beliebige Systeme angepasst werden können. Für diesen Zweck wird ein Crossbar-Switch gewählt. Die gesamte Buslogik und die Arbitrierung können komplett im Crossbar-Switch Modul untergebracht werden. Dieses eigenständige Modul kann dann auch in anderen Systemen eingesetzt werden.
Im Vergleich zu anderen Topologien ist ein Crossbar-Switch jedoch schwieriger an eine unterschiedliche Anzahl an Komponenten anpassbar. Dieser Nachteil wird gemindert, indem ein **dynamischer Crossbar-Switch** entwickelt wurde. Dynamisch bedeutet, dass nur die Anzahl der Komponenten spezifiziert werden muss und der Crossbar-Switch dann die nötigen Signale automatisch generiert.

		AMBA	Core-Connect	Avalon	Wishbone
Verfügbarkeit	Freies Bussystem	X	X	X	X
	Registrierung	X	X	X	
	Lizenz			X	
Topologie	Punkt-zu-Punkt			X	X
	Geteilter Bus				X
	Hierarchisch	X	X[1]		
	Ringbus		X[1]		X
	Crossbar-Switch				X
Arbitrierung	Statische Prioritäten	X[2]	X	X[3]	X[4]
	TDMA	X[2]		X[3]	X[4]
	Lotterie	X[2]		X[3]	X[4]
	Round-Robin	X[2]		X[3]	X[4]
	Tokenweitergabe	X[2]		X[3]	X[4]
	CDMA	X[2]		X[3]	X[4]
Übertragungs-Modi	Handshaking	X	X		X
	Burst-Modus	X	X	X	X
	Pipelining	X	X	X	X
	Split-Transfer	X	X		
	Read-Modify-Write				X
Busbreiten (Bit)	Datenbus	8-256[5]	8-256[6]	1-128	8,16,32,64
	Adressbus	32	10-32[7]	1-32	1-64

Tabelle 3.1: Eigenschaften der unterschiedlichen Bussysteme
Ergänzungen zur Tabelle:

[1] Kontrollbus ist ein Ringbus und die Datenbusse sind geteilte Busse
[2] Benutzerdefiniert mit Ausnahme des APB
[3] Slave-seitige Arbitrierung
[4] Benutzerdefiniert
[5] 32, 64, 128 oder 256 Bit für AHB und ASB. 8, 16, 32 Bit für APB
[6] 32, 64, 128 oder 256 Bit für PLB. 8, 16 oder 32 Bit für OPB. 32 Bit für DCR
[7] 32 Bit für PLB und OPB. 10 Bit für DCR

Nicht ausgewählte Busse

Aufgrund der Flexibilität, der freien Verfügbarkeit und der ausführlichen Spezifikation konnte Wishbone sich gegen alle anderen Bussysteme durchsetzen. Nachfolgend noch ein Vergleich mit den zuvor erwähnten Bussystemen und welche Eigenschaften für diese Arbeit von Nachteil sind:

- **AMBA** - Trotz vorhandener AHB-Lite-Schnittstelle wurde AMBA nicht ausgewählt. Zum einen entsprach AHB-Lite nicht den Anforderungen für die Implementierung und zum anderen beschränkt sich AMBA auf die hierarchische Bustopologie und erlaubt Anpassungen nicht so einfach wie die Wishbone Spezifikation. Aufgrund der ausführlichen Dokumentation war ein AMBA-System jedoch die zweite Wahl.
- **CoreConnect** - Wie bei anderen Bus-Systemen wird auch hier die hierarchische Bus-Topologie vorgegeben. Ein weiterer Aufwand sind die Kontrollsignale, die als ringförmiger Bus im System implementiert werden. Größter Nachteil im Bezug auf die Implementierung dieser Arbeit ist, dass CoreConnect zur Arbitrierung nur statische Prioritäten verwendet.
- **Avalon** - Avalon unterstützt nur Punkt-zu-Punkt-Verbindungen über einen Crossbar-Switch. Auch wenn dies der gewählten Topologie entspricht ist es wieder eine Einschränkung für die Wiederverwendbarkeit der einzelnen Module. Die entscheidende Einschränkung ist jedoch, dass Avalon nur auf einem Nios-Prozessor [31] von Altera eingesetzt werden kann.
- **STBus** - Der STBus trennt die Übertragung von Daten in mehrere Schichten und geht dadurch stark in Richtung NoC. STBus ist jedoch noch kein vollwertiges NoC, da fix festgelegte Protokolle für die Übertragung der Daten eingesetzt werden. Die Trennung der einzelnen Schichten erfolgt also nicht komplett. Wie auch bei einem NoC macht der Aufbau solch eines Netzwerkes nur dann Sinn, wenn das Gesamtsystem ausreichende Komplexität aufweist. Für die Implementierung dieser Arbeit wird STBus daher nicht weiter in Betracht gezogen.

Die Bussysteme, die in dieser Evaluierung behandelt wurden, wurden im Bezug auf Bekanntheit und Umfang der verfügbaren Dokumentation ausgewählt. Es existieren auch noch weitere SoC-Busse, die mehr oder weniger für diese Arbeit geeignet wären.

3.4 Weitere Details zu Wishbone

Der folgende Abschnitt enthält zusätzliche Details zur Wishbone-Spezifikation. Abbildung 3.8 zeigt eine Punkt-zu-Punkt-Wishbone-Verbindung die in der Spezifikation als Beispielaufbau verwendet wird.
Bei dieser Verbindung werden nicht alle Wishbone-Signale benötigt. Der Grund dafür ist, dass generell viele Signale in der Spezifikation nur als optional definiert sind. Diese werden nur bei speziellen Übertragungsmodi oder anderen erweiterten Busfeatures benötigt. Eine detaillierte Auflistung aller Signale inklusive der optionalen Signale kann in der Spezifikation ab Seite 27 gefunden werden.

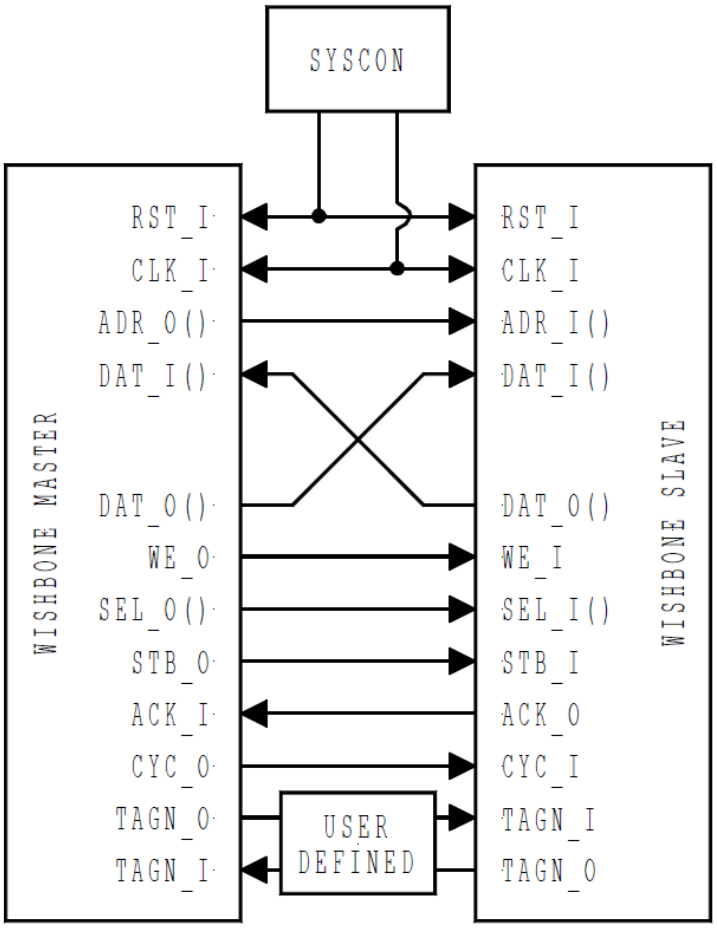

Abbildung 3.8: Wishbone Standard Verbindung/Signale [4]

Nachfolgend werden alle Signale aus Abbildung 3.8 näher beschrieben:
`RST_I` und `CLK_I` werden vom Verbindungsnetzwerk an alle Master und alle Slaves im System weitergegeben. Ersteres dient zum Zurücksetzen der Wishbone-Schnittstelle. Das Taktsignal `CLK_I` synchronisiert alle Komponenten miteinander.

ADR_O ist der Adressausgang, DAT_I und DAT_O sind die Datenleitungen eines Masters. Slaves besitzen hingegen einen Adresseingang ADR_I neben den beiden Datenleitungen und ein Acknowledgement ACK Signal um Busoperationen zu bestätigen. WE signalisiert ob eine Lese- oder Schreiboperation durchgeführt wird.

Das Signal SEL wird benötigt wenn Datenpakete mit unterschiedlichen Breiten übertragen werden. Mit Hilfe von SEL kann bestimmt werden welche Daten auf den Datenleitungen gültig sind. Bei einem 32-Bit breiten Bus, der minimal ein Byte übertragen kann, könnten die vier einzelnen Bytes individuell geschrieben werden. (SEL wäre in diesem Fall 4-Bit breit)

CYC (Zyklus) und STB (Strobe) kontrollieren den Ablauf einer Wishbone-Transaktion. CYC ist ein binäres Signal und ist immer aktiv wenn ein Buszyklus läuft. Andererseits ist STB immer dann aktiv wenn gerade eine Datenübertragung durchgeführt wird. Die beiden Signale unterscheiden sich erst wenn komplexere Transaktionen über den Bus abgewickelt werden. Beispielsweise werden bei einem BLOCK-Transfer mehrere Datenübertragungen durchgeführt. Über die gesamte Zeit wäre CYC aktiv um einen Buszyklus zu signalisieren. STB würde aber bei jeder einzelnen Datenübertragung aktiv sein und zwischen den Übertragungen nicht. Slaves bestätigen mit ihrem ACK oder anderen Signalen (Error, Retry) jede Aktivierung des STB Signals.

Für eine Wishbone-Verbindung werden alle erwähnten Signale benötigt. TAGN_O und TAGN_I stellen allerdings zwei benutzerdefinierte optionale TAG-Signale dar. Diese Signale können entweder der Adresse, den Daten oder dem Zyklus zugeordnet werden. Je nach Zuordnung wird somit festgelegt, zu welchen Zeitpunkten diese TAG-Signale gesetzt werden müssen. TAG-Signale die der Adressleitung zugeordnet sind, müssen zum Beispiel zu den gleichen Zeiten wie auch ADR_I oder ADR_O gesetzt werden. Folgende Tabelle zeigt die möglichen Zuordnungen dieser TAG-Signale:

	Master		Slave	
TAG INFO	TAG TYP	Zuordnung	TAG TYP	Zuordnung
Adress-Tag	TGA_O()	ADR_O()	TGA_I()	ADR_I()
Daten-Tag, IN	TGD_I()	DAT_I()	TGD_I()	DAT_I()
Daten-Tag, OUT	TGD_O()	DAT_O()	TGD_O()	DAT_O()
Zyklus-Tag	TGC_O()	Bus-Zyklus	TGC_I()	Bus-Zyklus

Tabelle 3.2: Wishbone TAG Typen

Wishbone unterstützt mehrere verbreitete Übertragungsmodi. Diese sind in Kapitel 3, 4 und 5 der Spezifikation [4] detailliert beschrieben. Neben einzelnen Lese- und Schreibzyklen sind Blockzyklen, ein RMW-Zyklus sowie ein Pipelined-Modus und ein Burst-Modus möglich. Eine Erklärung zu diesen Übertragungsmodi ist in Abschnitt 3.1.3 zu finden.

Abbildung 3.9 zeigt das Ablaufdiagramm der Wishbone-Signale bei einem einzelnen Lesezyklus. Die Wishbone-Signale im Bezug auf den Master müssen sich für diesen Lesezyklus an folgenden Ablauf halten:

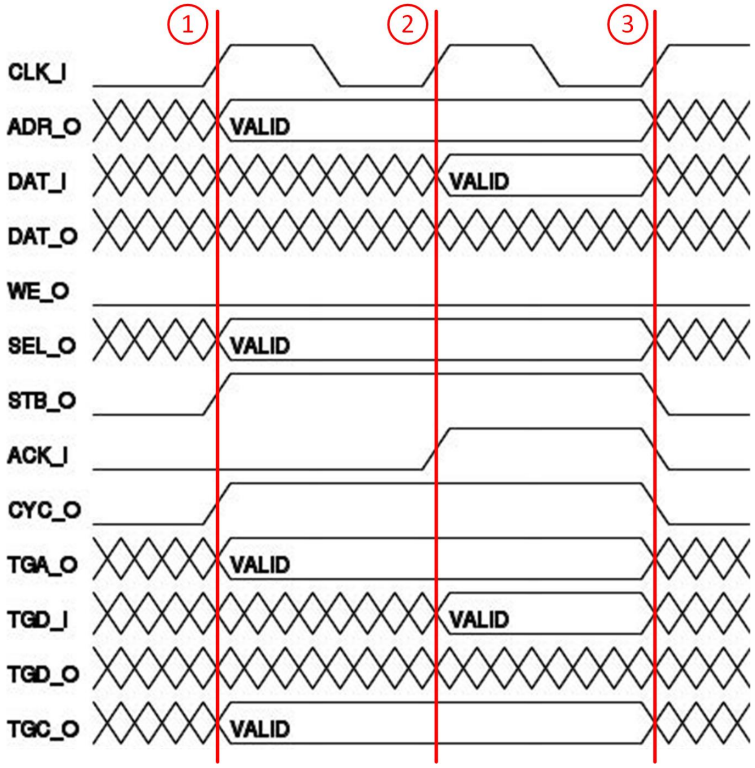

Abbildung 3.9: Einzelner Standard Lesezyklus bei Wishbone [4]

- Erste positive Taktflanke - (1)
  - Eine gültige Adresse muss auf `ADR_O` gesetzt werden. Inklusive dem dazugehörigen Tag-Signal `TGA_O`, falls dieses verwendet wird.
  - `WE_O` muss in diesem Fall negiert werden, um eine Lesezyklus zu kennzeichnen.
  - Die Bits in `SEL_O` müssen entsprechend der Daten gesetzt werden. `SEL_O` signalisiert wo im gesamten Datenarray gültige Daten erwartet werden können.

  - CYC_O und das dazugehörige TGC_O Signal müssen gesetzt werden, um den Start eines neuen Zyklus zu signalisieren.
  - Das setzen von STB_O startet die aktuelle Übertragungsphase. (Bei anderen Übertragungsmodi wie zum Beispiel Block Lesen/Schreiben besteht ein Bus-Zyklus aus mehreren Phasen. CYC_O würde aktiv bleiben und STB_O sowie ACK_I übernehmen das Handshaking).

- Zweite positive Taktflanke - (2)
  - Slave legt gültigte auf DAT_I und setzt das dazugehörige Signal TGD_I.
  - Slave setzt ACK_I als Antwort auf STB_O, um die anliegenden Daten als gültig zu erklären.
  - Der Master überwacht ACK_I und bereitet sich darauf vor, DAT_I und TGD_I zwischen zu speichern.

- Dritte positive Taktflanke - (3)
  - Die Daten auf DAT_I und TGD_I werden zwischengespeichert.
  - STB_O und CYC_O wird negiert, um das Ende der Phase und das Ende des Zyklus zu signalisieren.
  - Der Slave negiert ACK_I als Antwort auf die beendete Phase (STB_O negativ)

Um der Wishbone-Spezifikation konform zu sein, muss eine Implementierung ein sogenanntes Wishbone-Datenblatt beinhalten. Dieses Datenblatt enthält Informationen über die einzelnen Signale und eine genauere Erklärung zu benutzerdefinierten Signalen und wie diese eingesetzt werden. Anhand dieses Datenblattes wird die Wiederverwendbarkeit von Wishbone Komponenten gewährleistet.
Im Anhang B ist als Beispiel, das Wishbone-Datenblatt für die Wishbone-Schnittstelle des V-scale Prozessors zu finden.

Kapitel 4

Related Work

Zu Beginn beschäftigt sich diese Kapitel mit ähnlichen Arbeiten im Bezug auf die Busevaluierung und die Implementierung dieser Arbeit. Im Bereich der Busevaluierung werden andere Arbeiten gezeigt, die die Busevaluierung allgemeiner und ausführlicher umgesetzt haben. Zusätzlich wird eine Arbeit über ein modernes NoC vorgestellt. Herkömmliche SoC-Bussysteme werden von NoCs dieser Art immer häufiger abgelöst.
Im Bezug auf die dynamischen Task-Prioritäten, die in der Implementierung dieser Arbeit umgesetzt wurden, wird ein anderer Ansatz vorgestellt. Bei diesem Ansatz wurde eine Arbitrierung für SoCs entwickelt, die die nötige Bandbreite und die Einhaltung von Echtzeit-Bedingungen garantiert.

4.1 Related Work

4.1.1 SoC-Bussysteme

Evaluierungen von SoC-Bussystemen existieren in den verschiedensten Varianten. Ähnlich wie bei dieser Arbeit sind oft spezielle Anforderungen an ein Bussystem vorhanden. Eine frühe Arbeit ist ein von OpenCores durchgeführte Bewertungen von CoreConnect, AMBA und Wishbone [32]. Das Resultat dieser Arbeit führte dazu, dass OpenCores in den folgenden Jahren gezielt auf Wishbone als Bus gesetzt hat. Der Autor begründet diese Entscheidung mit der Vielseitigkeit von Wishbone. Alle Komponenten können mit der gleichen Wishbone-Schnittstelle ausgestattet werden und müssen nicht, wie beim hierarchischen AMBA Bussystem, in schnellere und langsamere Busse eingeteilt werden.
Darüber hinaus die intuitiven Signale die einfach an andere Schnittstellen angepasst werden können, ein Vorteil gegenüber den anderen Bussystemen. Dies Entscheidung wiederum resultiert in der hohen Verfügbarkeit an freien IP-Cores, die kompatibel zu Wishbone sind.

Eine weitere interessante Übersicht über SoC-Bussysteme liefern Mitić und Stojčev [33]. Neben der unterstützten Topologien, Arbitrierungsmethoden und Übertragungsmodi, wird auch die Verfügbarkeit behandelt. Also welche Lizenz nötig ist, um das Bussystem einzusetzen oder ob es sich um Open-Source-Systeme handelt. Alleinstellungsmerkmal für diese Arbeit ist jedoch die hohe Anzahl an Bussystemen, die für die Übersicht berücksichtigt werden.

Als letzte Arbeit in dieser Kategorie, ist noch „Wishbone Bus Architecture - A suvery and comparison“ [34] zu erwähnen. Diese verhältnismäßig junge Arbeit vergleicht Wishbone mit AMBA, CoreConnect und Avalon. Die Verfasser dieser Arbeit kommen, wie schon Rudolf Usselmann von OpenCores [32], zu dem Schluss, dass im Allgemeinen Wishbone eingesetzt werden sollte. Die Begründung ist in diesem Fall ähnlich und es werden zusätzlich noch Wishbone-Features wie RMW und der Support bzw. die Entwicklungstools positiv hervorgehoben.

4.1.2 NoC

Wie bereits in Kapitel 3 erwähnt, befasst sich die heutige Forschung für SoC-Kommunikationsnetzwerke, vorwiegend mit modernen NoCs. Das Unternehmen Arteris[25] wurde 2003 gegründet und ist auf On-Chip-Verbindungsnetzwerke spezialisiert, wie sie in SoCs genutzt werden. 2006 veröffentlichte Arteris das erste kommerzielle NoC-System.

Aus dieser Zeit stammt auch ein informativer Artikel von Arteris [35], in dem NoCs und traditionelle Bussysteme gegenübergestellt werden. Standardkriterien, wie die maximale Frequenz, Bandbreite und Latenz wurden ebenso herausgearbeitet wie zum Beispiel der Ressourcenverbrauch der Implementierungen und der Energiebedarf. Ergebnis des Artikels ist, dass NoCs die traditionellen Bus-Systeme in nahezu jedem Bereich übertreffen.

Sucht man ein NoC, das speziell auf die Besonderheiten von FPGAs angepasst ist, so könnte CONNECT[26] relevant sein. CONNECT steht für „CONfigurable NEtwork Creation Tool“ und wurde 2012 veröffentlicht [5]. In ihrer Publikation zur initialen Version des NoC beschreiben die Entwickler ihre Designentscheidungen im Bezug auf FPGAs als Zielhardware. Des Weiteren vergleichen sie Ressourcenverbrauch und Leistung mit einem frei verfügbaren NoC für Application-Specific Integrated Circuits (ASICs). Folgende Abbildung 4.1 zeigt die Ergebnisse hinsichtlich Ressourcenverbrauch:

[25] http://www.arteris.com/
[26] http://users.ece.cmu.edu/ mpapamic/connect/

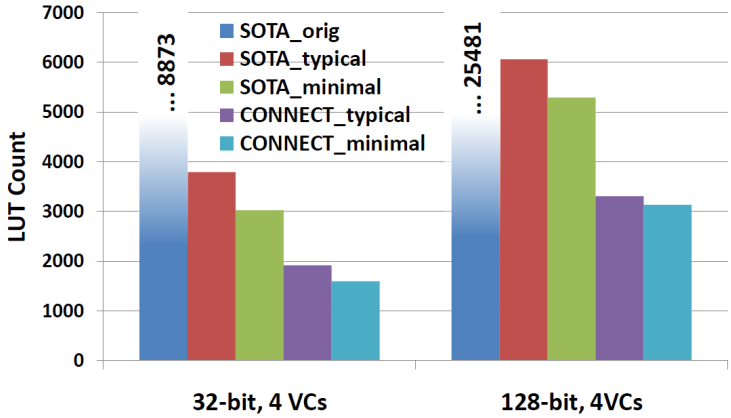

Abbildung 4.1: Ressourcenverbrauch von CONNECT im Vergleich [5]

`SOTA_orig` entspricht dem original State-of-the-art (SOTA) NoC welches als Referenz verwendet wird. Dieses moderne NoC arbeitet mit virtuellen Kanälen (VC). Diese virtuellen Kanäle stellen Punkt-zu-Punkt Verbindungen innerhalb des Netzwerkes dar und sind entscheidend dafür, wie groß die Buffer im System sein müssen. Mehr virtuelle Kanäle erhöhen den Ressourcenbedarf eines NoC daher spürbar.
In weiterer Folge haben die Entwickler die Implementierung dieses NoC für FPGAs angepasst (`SOTA_typical`).
Außerdem wurden noch beide NoCs (`SOTA_minimal` und `CONNECT_minimal`) auf minimalen Ressourcenverbrauch hin parametriert, um ein unteres Limit für den Ressourcenverbrauch zu erhalten.

4.1.3 Echtzeit-Arbitrierung

Die Implementierung dieser Arbeit (Kapitel 5) hat sich damit befasst, ein Multi-Core-System mit Task-Awareness zu entwickeln. In diesem System ist es möglich, dass jeder einzelne Master dringliche Daten mit entsprechend hoher Priorität versenden kann. Die Software, die auf den Prozessorkernen läuft, entscheidet somit über die Auslastung des gesamten Kommunikationsnetzwerkes. Echtzeitanforderungen sowie Bandbreitenanforderungen können mit diesem System über die Software gesteuert werden.

Entwickler von SoC-Kommunikationssystemen wählen zumeist einen anderen Ansatz. Dies ist notwendig, da die eingesetzte Software für die Entwickler des SoC oftmals nicht bekannt ist. Eine in den unterschiedlichsten Varianten entwickelte Alternative ist daher eine Arbitrierung, die unabhängig von den Daten diese Anforderungen erfüllen kann.

„RT_lottery“ (**R** für Real-time und **T** für Tuned weight) [8] ist zum Beispiel ein Arbitrierungs-Algorithmus für SoCs, der Echtzeit- und Bandbreitenanforderungen garantiert. Im Paper werden anfangs herkömmliche Arbitrierungsmethoden untersucht, wie sie auch in dieser Arbeit in Kapitel 3.1.2 untersucht wurden. Darüber hinaus behandeln die Verfasser zweistufige Arbitrierung, um Probleme mit einfachen Algorithmen zu umgehen. Um zum Beispiel nicht genutzte Zeitslots bei einem TDMA Algorithmus neu zu vergeben, könnte ein zweiter Algorithmus genau diesen Zeitslot an andere wartende Master übergeben.

RT_lottery arbeitet ebenfalls mit zwei Stufen. Vorab muss der Systemdesigner einige notwendige Faktoren für jeden Master bestimmen. Zum einen die benötigte Bandbreite und zum anderen Informationen über das Verhalten des Datenverkehrs (Zyklen in denen eine Echtzeitanforderung erfüllt werden muss, Intervall zwischen zwei aufeinanderfolgenden Übertragungen, etc.). Die erste Stufe kümmert sich um diese Echtzeitbedingungen indem die Übertragungen, mit Hilfe der zusätzlichen Informationen, in optimaler Reihenfolge durchgeführt werden. Wenn alle harten Echtzeitbedingungen erfüllt wurden und der Bus frei ist, entscheidet ein Lottery-Algorithmus. Das besondere an diesem ist eine automatische Gewichtsanpassung, um Bandbreite zwischen den Mastern besser zu verteilen.

RT_lottery wurde von den Entwicklern in einem System mit sechs Mastern getestet und drei anderen Arbitrierungs-Methoden gegenübergestellt. Das Resultat dieser Tests ist durchaus positiv und ist in Tabelle 4.1 zusammengefasst:

Arbitrierungs-Methode	Echtzeitfähigkeit	Bandbreitenauslastung
RT_lottery	Harte Echtzeit	Am besten
TDM + Lotterie	Nicht bei kritischen Fällen	Gut aber benötigt Gewichtsanpassung
Lotterie	Nicht berücksichtigt	Gut aber benötigt Gewichtsanpassung
Statische Prioritäten	Nicht berücksichtigt	Schlecht

Tabelle 4.1: RT_lottery im Vergleich zu anderen Arbitrierungs-Methoden [8]

Kapitel 5

Implementierung

Die Implementierung für diese Arbeit umfasst ein Multicore-System in Verilog, bestehend aus mehreren V-scale Softcores, einem Verbindungsnetzwerk für den Wishbone Bus und Peripherie. Kapitel 2 erläutert warum V-scale als passender Prozessor für diese Implementierung gewählt wurde. Information zum gewählten Wishbone-Bus sind im vorherigen Kapitel 3 zu finden.

Hauptaugenmerk bei der Implementierung liegt auf der Unterstützung von taskabhängigen Prioritäten im gesamten Multicore-System. Überdies wurde das Bussystem auf einfache Erweiterbarkeit ausgelegt. Zusätzliche Prozessoren oder Komponenten sollten ohne großen Aufwand ins System integriert werden können.

Der folgende Abschnitt 5.1 behandelt die notwendigen Anpassungen am V-scale-Prozessor. Abschnitt 5.2 beschreibt die Entwicklung des dynamischen Wishbone-Verbindungsnetzwerkes (Crossbar-Switch).

Im nächsten Abschnitt 5.3 ist ein Überblick über das komplette System am FPGA zu finden. In diesem werden alle weiteren Wishbone Komponenten beschrieben, die für diese Arbeit entwickelt wurden.

Am Ende dieses Kapitels, in Abschnitt 5.4, wird das verwendete Betriebssystem *mosart*MCU-OS beschrieben, sowie die durchgeführte Anpassung für die Task-Prioritäten. Der Aufbau der Testprogramme wird in diesem Abschnitt ebenfalls erläutert.

5.1 V-scale Integration

Der V-scale-Soft-Core wurde zu Beginn der Implementierung vom Github-Repository[27] von UCB Architecture Research bezogen. Eine Busanbindung ist standardmäßig bereits in Form einer HASTI/AHB-Lite-Schnittstelle vorhanden. Diese Schnittstelle musste durch eine Wishbone-Anbindung ersetzt werden.

Weiters musste die Möglichkeit geschaffen werden, die Priorität des laufenden Tasks für die Wishbone-Schnittstelle zur Verfügung zu stellen. Im Zuge dessen wurde auch noch ein Feature in die Gegenrichtung implementiert, welches es erlaubt, von der Hardware aus, einen Core-Index an das Betriebssystem zu übermitteln. Dieser Core-Index wird benötigt, um den Ablauf der Testsoftware für jeden Prozessorkern unabhängig zu steuern. In Abschnitt 5.1.2 und Abschnitt 5.4.2 wird diese Funktion detaillierter betrachtet.

[27] https://github.com/ucb-bar/vscale

5.1.1 Wishbone-Anbindung

Für die Wishbone-Anbindung wurde ein eigenes Top-Modul (wb_vscale_riscV/hdl/vscale_core_wb.v) entwickelt, welches das Originalmodul aus dem V-scale-Repository (vscale_core.v) ersetzt.

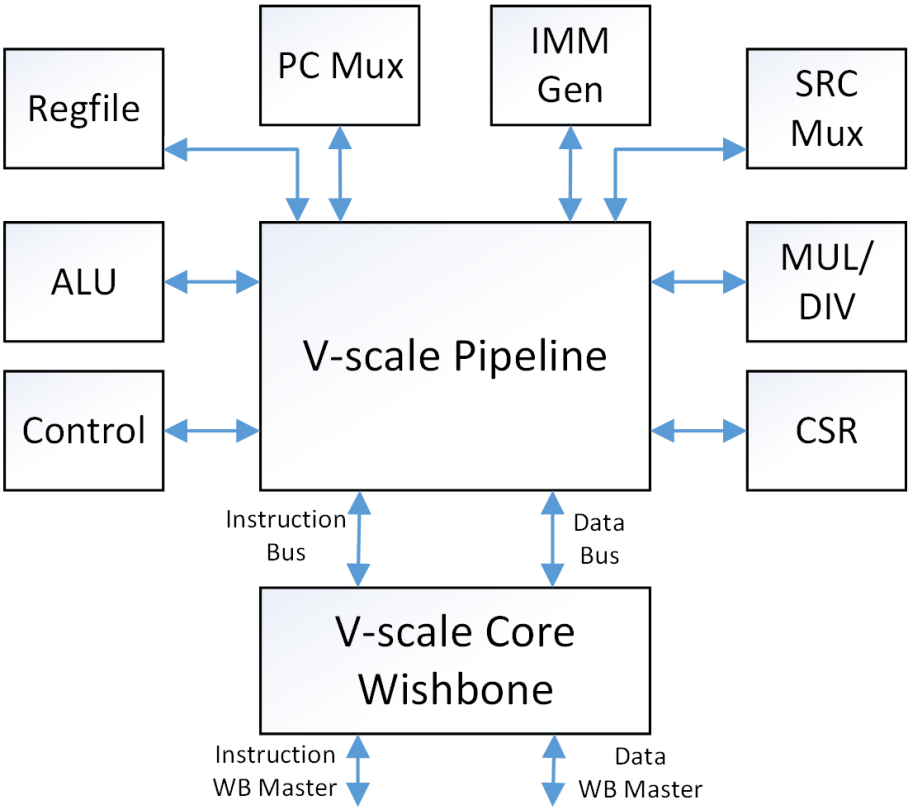

Abbildung 5.1: V-scale Module inklusive Top-Modul mit Wishbone-Schnittstelle

Abbildung 5.1 zeigt den Aufbau von einem V-scale Kern in Verilog. Das V-scale Pipeline-Modul ist im Grunde das Herzstück des Prozessors. Das Wishbone-Modul setzt sich direkt darüber und übersetzt den Instruktions- und Datenbus in zwei getrennte Wishbone-Schnittstellen.

Üblicherweise würde ein Prozessorkern auch nur einen Master im System darstellen. Aus mehreren Gründen wurde jedoch eine getrennte Busanbindung implementiert:

- Instruktions- und Datenbus sind aufgrund der Prozessorarchitektur bereits getrennt vorhanden.
- Eine getrennte Bus-Anbindung ergibt zwei Wishbone-Master pro V-scale-Kern. Möglichst viele Master im Multicore-System sind von Vorteil, um die

Robustheit der Implementierungen bzw. des Verbindungsnetzwerkes zu testen.

- Ein bereits implementierter Dual-Port-Speicher vom *mosart*MCU-Projekt (angepasst an Wishbone), kann wie zuvor eingesetzt werden.

Implementiert wird die Wishbone-Anbindung über zwei Zustandsautomaten. Zum einen ein rein lesender Zustandsautomat für die Instruktionen und für die Daten ein Zustandsautomat, der Lese- und Schreiboperationen unterstützt. Ähnliche Implementierungen sind auch online zu finden[28]. Abweichende Wishbone-Signale sowie Anpassungen am V-scale und Probleme mit fertigen Implementierungen machen jedoch eine eigene Ausführung notwendig. Nachfolgende Abbildung 5.2 zeigt den Zustandsautomaten zuständig für den Instruktions-Bus:

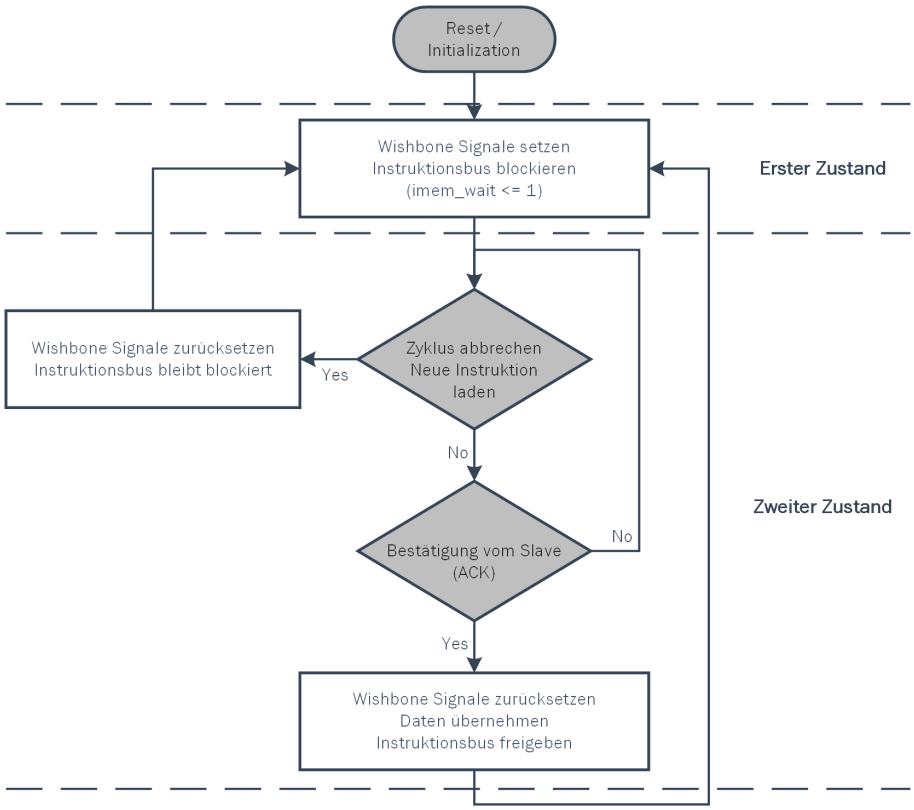

Abbildung 5.2: V-scale-Instruktion-Bus Zustandsautomat für die Wishbone-Schnittstelle

Im Gegensatz zum Zustandsautomaten für den Datenbus gibt es beim Instruktions-Bus eine Besonderheit, nämlich die Möglichkeit den Wishbone-Zyklus zurück zu setzen. Notwendig ist dies, da der Prozessor zum Beispiel in eine Ausnahme laufen oder ein Interrupt auftreten kann, während ein Wishbone-Zyklus aktiv ist.

[28] https://github.com/heshamelmatary/wb_riscvscale

Auch bei einem Sprung innerhalb des Programmablaufs muss der Wishbone-Zyklus abgebrochen werden. Aufgrund der Unterbrechung benötigt der Prozessor die Daten für die ursprüngliche Instruktion nicht mehr. In diesem Fall muss die Instruktion erneut geladen und ein neuer Wishbone-Zyklus gestartet werden. Dies ermöglicht dem V-scale Prozessor, alle auftretenden Ereignisse abzuarbeiten.

Zusätzlich zum Top-Modul muss daher ein Signal implementiert werden, welches der Wishbone Schnittstelle mitteilt, dass der aktuelle Zyklus zurückgesetzt werden muss. Listing 5.1 zeigt die Implementierung dieses `wishbone_replay_IF` Signals im Steuerwerk (Control Unit):

```
1   // IF stage ctrl
2   always @(posedge clk) begin
3      if (reset) begin
4         wishbone_replay_IF <= 1'b1;
5         replay_IF <= 1'b1;
6      end else begin
7         replay_IF <= (redirect && imem_wait) ||
8         (fence_i && store_in_WB);
9         // Wishbone Instruktions-Bus wird bei
10        // diesen Signalen unterbrochen
11        wishbone_replay_IF <= (redirect && imem_wait) ||
12        (fence_i && store_in_WB) || (ex_DX && imem_wait);
13     end
14  end
```

Listing 5.1: Signal wishbone_replay_IF zum Zurücksetzen der Wishbone-Schnittstelle (Control Modul)

`replay_IF` wird vom Prozessor intern verwendet und signalisiert das erneute Laden einer Instruktion (Instruction Fetch). Ausgelöst wird dieses Signal bei einem Adresssprung (`redirect`) oder im Falle einer Speicherbarriere[29] (`fence_i` Instruktion). `wishbone_replay_IF` reagiert zusätzlich auf `ex_DX`, welches eine Ausnahme in der Ausführung signalisiert. Sobald eine dieser drei Kriterien erfüllt ist, wird der laufende Wishbone-Zyklus zurückgesetzt.

[29]Garantiert das laufende Speicheroperationen abgearbeitet werden bevor neue Speicheroperationen behandelt werden.

5.1.2 Task-Priorität und Core-Index in Hardware

Damit das Wishbone-Verbindungsnetzwerk mit den Prioritäten der laufenden Tasks arbeiten kann, muss diese Information vom Betriebssystem an die Hardware übergeben werden. Die RISC-V-Spezifikation ermöglicht es einfach, solch eine Benutzer-Erweiterung umzusetzen.

Die RISC-V-ISA definiert einen 12-Bit breiten Adressbereich für bis zu 4096 Control and Status Register (CSR). Diese Register können mit privilegierten Funktionen atomar beschrieben und ausgelesen werden. Bereits definierte CSRs sind zum Beispiel Timer (Systemuhr), Counter, Interrupt-Handling und dergleichen. Der komplette CSR-Adressbereich ist definiert in freie und für den Standard reservierte Bereiche. Die freien (Non-standard) Bereiche, werden auch in Zukunft nicht von Standard-Erweiterungen belegt werden.

CSR Address			Hex	Use and Accessibility
[11:10]	[9:8]	[7:6]		
User CSRs				
00	00	XX	0x000-0x0FF	Standard read/write
01	00	XX	0x400-0x4FF	Standard read/write
10	00	XX	0x800-0x8FF	Non-standard read/write
11	00	00-10	0xC00-0xCBF	Standard read-only
11	00	11	0xCC0-0xCFF	Non-standard read-only

Tabelle 5.1: RISC-V-User-CSR-Adressbereiche aus [9], Seite 6.

Die Spezifikation definiert auch noch weitere Adressbereiche für Supervisor-, Hypervisor- und Machine-Level. Für die V-scale Implementierung ist nur User- und Machine-Level interessant, da nur diese zwei Modi unterstützt werden.

```
1  `define CSR_ADDR_PRIORITY  12'h800
2  `define CSR_ADDR_CORE_IDX  12'hFC0
```

Listing 5.2: Hinzugefügte CSR-Definitionen in vscale_csr_addr_map.vh

Das CSR für die Task-Priorität wurde im ersten freien Bereich bei den User CSRs definiert (Grau hinterlegt in Tabelle 5.1). Verbunden ist dieses Register direkt mit dem Signal **task_priority** welches an die Wishbone-Schnittstelle übergeben wird. **TASK_PRIO_LEN** wurde auf 8-Bit festgelegt und definiert somit die breite der benötigten Signale sowie eine maximale Task-Priorität von 255 (0xFF). Dieser Wert stellt auch die höchste Priorität in einem System dar. Initialisiert wird die Task-Priorität bei jedem V-scale mit dem Wert **0x7F**, also gerundet in der Mitte des Wertebereichs.

Ein V-scale-Kern, der sich gerade initialisiert, hat also immer Vorrang im Vergleich zu einem Kern, der gerade im Idle-Modus (Priorität = 0) ist. Höhere Prioritäten als 0x7F ermöglichen es, zeitkritischen Tasks auch vor initialisierenden Prozessorkernen bevorzugt zu werden.

Wie in Listing 5.2 zu erkennen, wird die Adresse für ein weiteres CSR definiert (**Core-Index**). Dieses dient zur hardwareseitigen Identifizierung der einzelnen V-scale-Kerne im Multi-Core-System. Zum einen wird dieses Register in der Adressberechnung der Wishbone-Schnittstelle genutzt (Listing 5.8). Adressen die für das Random-Access-Memory (RAM)-Modul bestimmt sind, werden mit Hilfe des Registers für den richtigen RAM umgerechnet.
Zum anderen wird der Core-Index genutzt um den richtigen Programmcode in der Testsoftware auszuführen. In Abschnitt 5.4.2 wird diese Funktionalität näher beschrieben. Das Register Core-Index wird bei der Initialisierung des V-scale-Kerns über einen Parameter gesetzt.

Spezielle CSR-Instruktionen sind notwendig, um mit diesen beiden CSRs zu arbeiten. In Abschnitt 5.4, wo das Betriebssystem näher erklärt wird, sind auch diese Funktionen beschrieben.

5.2 Wishbone Crossbar-Switch

Für das Multicore-System musste ein Bussystem entwickelt werden, welches alle Komponenten miteinander vernetzt. Hier fiel die Wahl auf einen Crossbar-Switch für den gewählten Inter-Core-Bus Wishbone.

OpenCore[30], die Entwickler von Wishbone, haben für diese Zwecke bereits einen Generator (Wishbone Builder[31]) zur Verfügung gestellt. Dieser wurde für diese Arbeit allerdings nicht verwendet. Der Wishbone Builder unterstützt noch keine Verilog-Ausgabe (zurzeit nur Very High Speed Integrated Circuit Hardware Description Language (VHDL)) und der erzeugte Code ist schwer leserlich. Die Signale sowie die Arbitrierung müssten auch noch manuell angepasst werden.

Aus diesen Gründen wurde ein eigener Wishbone-Crossbar-Switch implementiert. Dieser sollte jedoch trotzdem die Vorteile eines Generators besitzen und mit einer dynamischen Anzahl an Knoten arbeiten. Da diese Flexibilität im Verilog-Code direkt implementiert wurde, ist auch kein externer Generator nötig, um das Gesamtsystem anzupassen. Diese Inline-Generierung macht Anpassungen einfach und der Crossbar-Switch lässt sich schnell in andere Projekte integrieren.

5.2.1 Implementierung

Der dynamische Wishbone Crossbar-Switch kann über folgende Parameter konfiguriert werden:

```
1     parameter MASTERCOUNT   = 2,
2     parameter SLAVECOUNT    = 2,
3     parameter WB_DATA_WIDTH = 32,
4     parameter WB_ADDR_WIDTH = 32,
5     parameter WB_PRIO_WIDTH = 8
```

Listing 5.3: Konfigurierbare Parameter beim Wishbone Crossbar-Switch

Die Anzahl der Master sowie die Anzahl der angeschlossenen Slaves definieren die generierte Größe des Crossbar-Switches. Zusätzlich kann auch noch die Adressbreite sowie die Datenbreite angepasst werden.

Der letzte Parameter definiert die Breite des Benutzer-Signales (Adress Tag - TGA_i). Dieses dient zur Übergabe der Task Priorität an den Crossbar-Switch, und ist daher nur als Eingangssignal von den Mastern implementiert.

[30] http://opencores.org
[31] http://opencores.org/project,wb_builder

Ein weiterer Teil der Konfiguration ist die Angabe von Adressen und Masken:

```
1     // address & mask for slave mapping
2     input wire [SLAVECOUNT-1:0] [WB_ADDR_WIDTH-1:0]   address,
3     input wire [SLAVECOUNT-1:0] [WB_ADDR_WIDTH-1:0]   mask
```

Listing 5.4: Adressen und Adressmasken der Slaves (Crossbar)

Für jeden Slave muss hier eine Adresse und eine dazugehörige Adressmaske definiert werden. Diese Adressbereiche können sich auch überschneiden. Der anfragende Master wird dann immer mit dem Slave verbunden, der einen niedrigeren Index in der Crossbar-Switch besitzt.
Der Crossbar-Switch unterteilt sich intern in einen synchronen Block und kombinatorische Logik:

Synchroner Block

Der synchrone Block kümmert sich um die Anfragen der Master. Die angefragten Adressen werden maskiert und den richtigen Slaves zugeordnet. Danach wird noch überprüft, ob der betreffende Master oder der angefragte Slave noch mit einem aktiven Zyklus beschäftigt ist.

Wenn eine gültige Adresse angefragt wurde und Master sowie Slave bereit für eine Verbindung sind, folgt noch die Arbitrierung. Diese ist spezifisch für das geplante Gesamtsystem und wählt den nächsten Master anhand der übermittelten Task-Priorität. Zusätzlich wird darauf geachtet, dass immer nur ein Master mit einem Slave verbunden wird. Mehrere dieser Verbindungen können aber parallel aktiv sein und somit die Performance des Gesamtsystems erhöhen.
Übermitteln mehrere Master die gleiche Task-Priorität, dann wird der Master mit dem niedrigeren Index ausgewählt.

Mit der Arbitrierung hat der synchrone Block seine Arbeit vollendet und wird erst bei der nächsten steigenden Clock-Flanke erneut ausgeführt. Eine aktive Verbindung im Crossbar-Switch bleibt daher immer mindestens einen Clock-Zyklus aktiv.

Kombinatorische Logik

Die kombinatorische Logik hat die Aufgabe, alle Signale einer aktiven Verbindung zwischen Master und Slave durchzuschleifen. Der Programmcode für diese Aufgabe stellt den größten Teil eines Crossbar-Switches dar und hier muss am meisten geändert werden wenn, wenn Signale modifiziert werden müssen.

Allerdings wird dieser kombinatorische Teil in diesem dynamischen Crossbar-Switch automatisch generiert. Möglich macht dies der Verilog-Befehl **generate**. Mit Hilfe dieses Befehls kann Verilog-Code direkt in einer Verilog-Datei mehrmals erzeugt werden. Zum einen bleibt der Crossbar-Switch somit übersichtlich und zum anderen sind Anpassungen an den Signalen leicht möglich.

```
1  // Kombinatorische Logik welche die ausgewaehlten Signale von
2  // Slave zu Master durchschleift
3  genvar mst_gen;
4  generate
5    for (mst_gen = 0; mst_gen < MASTERCOUNT; mst_gen = mst_gen + 1)
6      begin
7        always @(*) begin
8          wb_s2m_ack_o[mst_gen] = 0;
9          wb_s2m_err_o[mst_gen] = cycle_high_bad_address[mst_gen];
10         wb_s2m_rty_o[mst_gen] = 0;
11         wb_s2m_dat_o[mst_gen] = {WB_DATA_WIDTH{1'b0}};
12
13         for (slave = 0; slave < SLAVECOUNT; slave = slave + 1)
14           if (active_connections[mst_gen][slave])
15             begin
16               wb_s2m_ack_o[mst_gen] = wb_s2m_ack_i[slave];
17               wb_s2m_err_o[mst_gen] = wb_s2m_err_i[slave];
18               wb_s2m_rty_o[mst_gen] = wb_s2m_rty_i[slave];
19               wb_s2m_dat_o[mst_gen] = wb_s2m_dat_i[slave];
20             end
21       end
22     end
23 endgenerate
```

Listing 5.5: Generierung des Codes der die Slave-Signale weiterleitet. (Crossbar Modul)

Listing 5.5 zeigt die Generierung der kombinatorischen Master-Logik. Die spezielle Generator-Variable **mst_gen** gibt an, wie oft der folgende Programmcode generiert werden soll. Falls eine aktive Verbindung besteht, leitet der resultierende Programmcode, alle Slave-Signale an den entsprechenden Master.

5.2.2 Arrays in Verilog

Bei der Integration des Crossbar-Switches in ein Verilog Modul muss eine Einschränkung von Verilog berücksichtigt werden. Die Unterstützung von mehrdimensionalen Arrays ist nämlich nur bedingt gegeben.

Es ist zwar möglich, mit mehrdimensionalen Arrays zu arbeiten, aber es können keine Input- oder Output-Ports als mehrdimensional definiert werden. Hier müssen immer **packed** Arrays übergeben werden, also alle Bits in einer Reihe.
Nehmen wir als Beispiel die Adress-Signale aller Master zum Crossbar-Switch:

```
1 // Master Ports (master --> crossbar)
2 input wire [MASTERCOUNT-1:0] [WB_ADDR_WIDTH-1:0] wb_m2s_adr_i
```

Listing 5.6: Adress-Input Array von allen Mastern (Packed 2D-Array)

Im Crossbar-Switch kann einzeln auf die Wishbone-Adressen der Master zugegriffen werden. Das Verilog Modul darüber sieht jedoch nur ein langes eindimensionales Array mit der Länge **MASTERCOUNT * WB_ADDR_WIDTH**, und kann daher nicht auf die richtigen Elemente zugreifen.

Um dieses Problem zu beheben, müssen die **packed** Arrays in **unpacked** Arrays umgewandelt werden. Diese befinden sich dann nicht mehr in einer Reihe im Speicher und der Zugriff ist möglich. Implementiert wird diese Umwandlung wieder mit dem übersichtlichen Verilog-Befehl **generate**. Nachfolgendes Listing 5.7 zeigt die Umwandlung aller zweidimensionalen Signale zwischen Crossbar-Switch und den Mastern:

```
1  // Konvetriert packed in unpacked 2D Arrays
2  ////////////////////////////////////////////////////////////
3  generate
4  genvar m;
5  for(m = 0; m < MASTERCOUNT; m = m + 1)
6  begin
7    assign intercon_m2s_adr_i[m*WBADDRWIDTH+:WBADDRWIDTH]
8       = unpacked_m2s_adr_i[m];
9    assign intercon_m2s_sel_i[m*4+:4]
10      = unpacked_m2s_sel_i[m];
11   assign intercon_m2s_cti_i[m*3+:3]
12      = unpacked_m2s_cti_i[m];
13   assign intercon_m2s_bte_i[m*2+:2]
14      = unpacked_m2s_bte_i[m];
15   assign intercon_m2s_dat_i[m*WBDATAWIDTH+:WBDATAWIDTH]
16      = unpacked_m2s_dat_i[m];
17   assign intercon_m2s_tga_i[m*WBPRIORWIDTH+:WBPRIORWIDTH]
18      = unpacked_m2s_tga_i[m];
19
20   assign unpacked_s2m_dat_o[m]
21      = intercon_s2m_dat_o[m*WBDATAWIDTH+:WBDATAWIDTH];
22 end
23 endgenerate
```

Listing 5.7: Umwandlung der zweidimensionalen Master-Crossbar-Arrays (Nexys4 Modul)

5.3 Gesamtsystem

Mehrere Systeme wurden für das Nexys4-Board von Digilent, basierend auf einem Arktix-7-FPGA von Xilinx entwickelt. In diesem Abschnitt wird das größte Gesamtsystem, im Bezug auf die Anzahl der V-scale-Prozessoren, näher beschrieben.

Zuvor befasst sich dieser Abschnitt noch kurz mit den zusätzlichen Komponenten die an den Wishbone-Bus angepasst wurden. Es sind zwar eine Vielzahl an Wishbone-Komponenten frei verfügbar, jedoch wurden Komponenten vom *mosart*MCU-Projekt angepasst um auch diese mit Wishbone verwenden zu können.

GPIO und UART

General Purpose Input/Output (GPIO) sowie Universal Asynchronous Receiver Transmitter (UART) wurden als Wishbone-Versionen umgesetzt. Diese beiden Komponenten dienen zur Ausgabe von Informationen über die Hardware und sind somit für Tests mit der Hardware geeignet. Bei beiden Komponenten können die Signale direkt in Wishbone-Signale umgewandelt werden. Es muss lediglich ein geeignetes Acknowledge-Signal (**wbs_ack_o**) generiert werden, um zum Wishbone-Bus kompatibel zu sein.

Dual-Port-Speicher

Eine essentielle Komponente für die V-scale Prozessoren ist der Speicher. Hierfür wird ein synchroner Dual-Port-Speicher als ROM im gesamten System verwendet. Der spezielle Aufbau der Zielhardware wird somit verwertet, da der gesamte Block-RAM [6] mit jeweils zwei unabhängigen Ports versehen ist. Die folgende Abbildung 5.3 zeigt den Aufbau eines 36KB RAM Blocks.

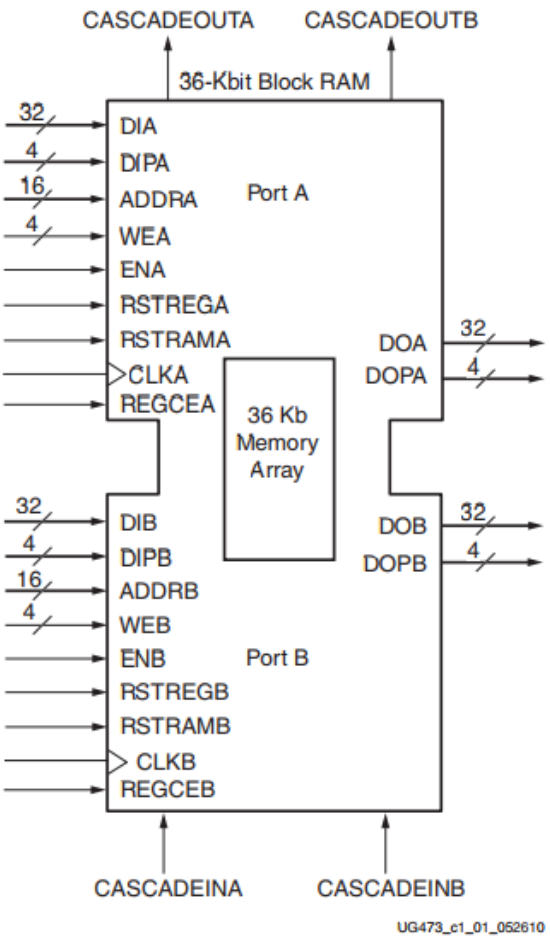

Abbildung 5.3: 36-KB Block RAM eines Artix-7-FPGA laut [6] Seite 16

Im Gesamtsystem ist nur ein Read-Only-Memory (ROM) für alle V-scale Prozessoren vorhanden. Der zweite ROM-Port ermöglicht es jedoch, dass der Instruktions-Bus eines Prozessors und der Daten-Bus eines anderen V-scale Prozessors synchron auf den ROM zugreifen können. Aufgrund der vergebenen Adressen im Gesamtsystem ist kein synchroner Zugriff von zwei Instruktions-Bussen möglich. Hierzu wäre eine Erweiterung der Wishbone-Crossbar nötig, welches es erlaubt, dass zwei Master auf den Adressraum der ROM zugreifen können wenn noch ein Zugriffsport unbenutzt ist.

Als RAM wird zwar die gleiche Komponente eingesetzt, aber jeweils nur ein Zugriffsport wird genutzt. Zudem besitzt jeder V-scale Prozessor im System seinen eigenen RAM. Der freie RAM-Port erhöht die Erweiterbarkeit des Systems. Hier könnte zum Beispiel ein Debug-Interface integriert werden, welches alle nötigen Informationen aus dem RAM lesen könnte, ohne das laufende System zu beeinträchtigen.

5.3.1 Aufbau des Gesamtsystems

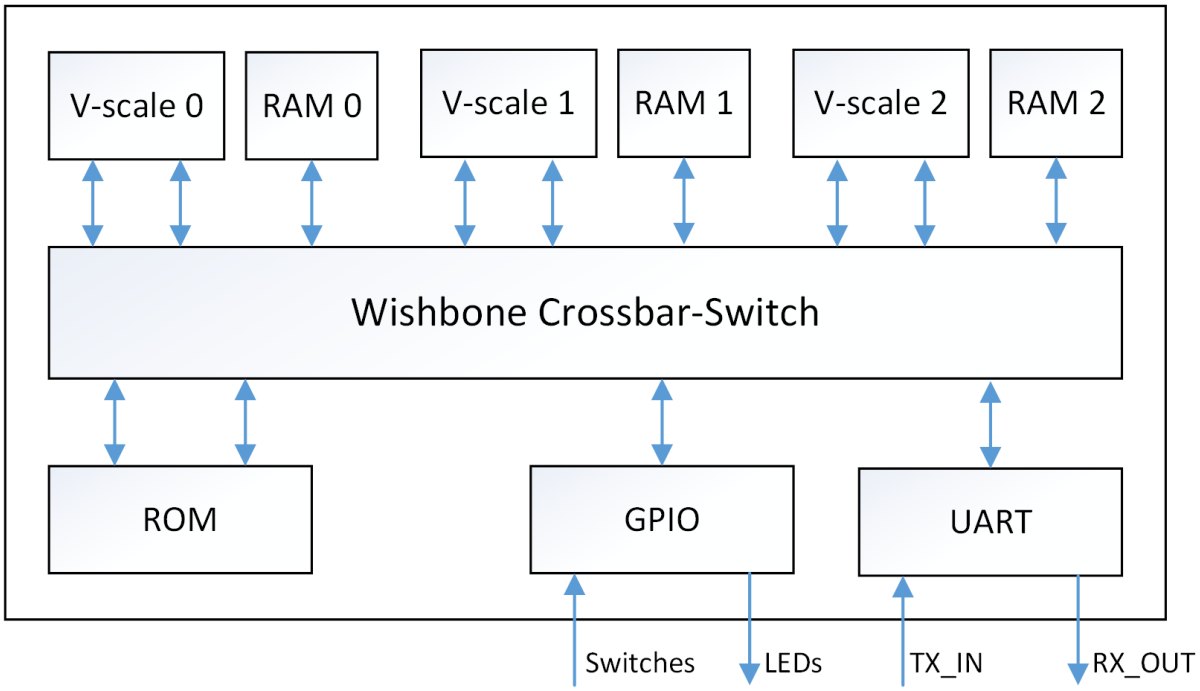

Abbildung 5.4: Gesamtsystem bestehend aus drei V-scale Kernen

Abbildung 5.4 zeigt den Aufbau des Gesamtsystems für drei V-scale-Prozessoren. Jeder V-scale-Prozessor besitzt einen dazugehörigen RAM, der aber global über den Wishbone-Bus angebunden ist. Alle Komponenten am Bus sind in einem linearen Adressraum angesiedelt. Theoretisch kann ein V-scale also auch auf den Speicher der anderen Prozessoren zugreifen.
In einem produktiven System würden dedizierte RAM-Module möglichst direkt an die Prozessoren angebunden werden. In diesem System wurde die Anbindung über den Bus realisiert, um bei den Tests den Crossbar-Switch noch zusätzlich auszulasten.
Das zuvor in Abschnitt 5.1.2 erwähnte Register Core-Index wird eingesetzt, um auf den richtigen RAM zuzugreifen.

Eine weitere Besonderheit bei der Adressberechnung für den Daten-Bus ist der zweite Zugriffsport des ROMs. Hier mussten zwei zusätzliche Parameter eingeführt werden, da der Prozessor nicht direkt mit dem ROM verbunden ist, sondern über den Wishbone Crossbar-Switch. Für das ROM ist daher nur eine Adresse möglich.

Um den zweiten Zugriffsport nutzbar zu machen, wird bei allen Adressen, die im Bereich des ROMs liegen (**DUAL_PORT_MEM_RANGE**) ein Offset addiert. Dieser zusätzliche Adressbereich wird im Crossbar-Switch ebenfalls zum ROM geleitet, und dort wird der Offset wieder subtrahiert, um die richtige Adresse zu erhalten.

```
1 parameter DUAL_PORT_MEM_RANGE = 'hFFFF8000,
2 parameter SECOND_PORT_OFFSET = 'h80000,
3 parameter CORE_INDEX = 'h00
4
5 dwbm_riscv_adr <= ((dmem_addr & DUAL_PORT_MEM_RANGE) == 0) ?
6                   dmem_addr + SECOND_PORT_OFFSET :
7                   dmem_addr + (CORE_INDEX - 1) * 'h8000;
```

Listing 5.8: V-scale-Parameter und Daten-Bus-Adressberechnung (V-scale Core Modul)

In Listing 5.8 sind die zusätzlichen Parameter jedes V-scale-Prozessors sowie die Adressberechnung für den Datenbus zu finden. Zeile 7 nutzt den Parameter **CORE_INDEX**, um die Adressen für den jeweils zugehörigen RAM zu berechnen.

Das Gesamtsystem besitzt neben den angegebenen Ausgängen und Eingängen auch noch weitere Ausgänge. Diese sind über Pins auf der Hardware zu erreichen und können für Messungen eingesetzt werden.

5.4 *mosart*MCU-OS und Testprogramme

Um das Gesamtsystem zu testen, wird ein einfaches Betriebssystem auf den RISC-V Prozessorkernen eingesetzt. Dieses ***mosart*MCU-OS** (**riscV_os**) wurde im Verlauf des mosartMCU-Projektes am Institut für Technische Informatik entwickelt.

Das Betriebssystem besitzt einen Scheduler mit dazugehöriger Task- und Ressourcenverwaltung. Auch unterschiedliche Prioritäten bei den laufenden Tasks werden bereits unterstützt. Einige Syscalls ermöglichen es, mit Tasks, Events, Timern und anderen Funktionen zu arbeiten. Implementiert ist das Betriebssystem in C und Assembler speziell für die RISC-V-Architektur.

Nachfolgend wird die Erweiterung des Betriebssystems erläutert, welche es ermöglicht, die Task-Prioritäten an die Hardware zu übergeben.

Am Ende dieses Abschnitts wird noch die Struktur der entwickelten Testprogramme, im Bezug auf das Betriebssystem, gezeigt.

5.4.1 *mosart*MCU-OS Erweiterung

Wie schon erwähnt, arbeitet das Betriebssystem bereits mit Task Prioritäten. Um diese nun an die Hardware zu übergeben, wird die Methode für den Taskwechsel modifiziert. Diese Methode wird auch bei einem Wechsel in den Idle-Task aufgerufen und eignet sich daher am besten für unsere Erweiterung.

Wie in Abschnitt 5.1.2 beschrieben, wird ein CSR-Register in V-scale integriert, um die Task-Priorität für die Wishbone-Schnittstelle zwischenzuspeichern. Auf alle CSRs kann mit speziellen atomaren Instruktionen zugegriffen werden.

Diese Instruktionen sind in der RISC-V-Spezifikation [11] definiert. $\mathbf{CSRR}\ rd,\ csr$ dient zum atomaren Lesen eines CSR und $\mathbf{CSRW}\ csr,\ rs1$ ist die dazugehörige Instruktion zum Schreiben eines dieser Register.

```
1   // CSR task_priority wird hier gesetzt
2   // immer wenn der Prozessor in den User-Mode wechselt
3   // Struktur des Task Control Blocks:
4   // 0(tp)  = the stack
5   // 4(tp)  = the priority
6   // 8(tp)  = base_priority
7   // 12(tp) = member_list
8
9   REG_L t1, 4(tp)           // Task Prio ins Register t1 laden
10  csrw 0x800, t1            // Ergebnis ins CSR task_priority
11                            // auf 0x800 schreiben
```

Listing 5.9: Setzen der Task-Priorität in **kernel_entry_uepilogue()**

Listing 5.9 zeigt die Umsetzung der Betriebssystemerweiterung. Bei einem Taskwechsel wird über den Taskpointer (**tp**) die Task-Priorität in ein Register geladen und dann mit dem Befehl csrw ins CSR geschrieben.
Der gezeigte Ausschnitt ist in der Methode **kernel_entry_uepilogue** enthalten, welche den Kontextwechsel durchführt.
Des Weiteren ist in den Kommentaren (Zeile 3 bis 6), der Aufbau des Task Control Block (TCB) angegeben. Diese Struktur befindet sich im Speicher und kann mit Hilfe des Taskpointer (**tp**) gefunden werden. Jeder Task im Betriebssystem besitzt einen TCB und dieser enthält notwendige Informationen über den jeweiligen Task. Bei einem Taskwechsel muss der nötige TCB aus dem Speicher geladen werden. Aus diesem Grund wurde das Setzen des CSRs für die Task-Priorität, an dieser Stelle implementiert.

5.4.2 Struktur der Testprogramme

Alle RISC-V-Prozessoren in einem System beziehen die gleiche Testsoftware vom ROM. Der Core-Index jedes Prozessors bestimmt dann, welcher Teil der Software ausgeführt wird. Diese Struktur gestaltet die Verteilung der Testprogramme im Multi-Core-System möglichst einfach.

```
1  #include <mosart_os.h>
2
3  int task0_core0(void);
4  int task0_core1(void);
5  int task0_core2(void);
6
7  void main(void)
8  {
9    uint32_t core_idx = 0;
10   asm volatile (" csrr %0, 0xFC0" : "=r" (core_idx));
11
12   switch (core_idx)
13   {
14     case 1:
15       os_register_task(task0_core0, 8*80, 50);
16       break;
17
18     case 2:
19       os_register_task(task0_core1, 8*80, 150);
20       break;
21
22     case 3:
23       os_register_task(task0_core2, 8*80, 250);
24       break;
25
26     default:
27       break;
28   }
29   os_run(0);
30 }
```

Listing 5.10: Struktur der Main-Funktion aller Testprogramme

Listing 5.10 zeigt die Struktur der Main-Funktion, wie sie in den Testprogrammen verwendet wird. Zu Beginn der Test-Routine wird mittels Inline-Assembler der Core-Index aus dem dazugehörigen CSR geladen. Inline-Assembler wird bei hardwarenaher Programmierung häufig benötigt, da nur so spezielle Instruktionen eines Prozessors ausgeführt werden können. In C wird ein Assembler-Befehl mit den Schlüsselwörtern **asm volatile** eingeleitet.

Mit der zuvor erklärten CSR Instruktion $csrr$ wird der Core-Index in die Variable **core_idx** geladen. Diese entscheidet in der darauffolgenden Switch-Anweisung, welcher Prozessor welchen Task für sich registriert. Die Parameter für **os_register_task** sind ein Pointer der Methode beziehungsweise des auszuführenden Tasks, die Größe des Stacks in Byte $(8 * 80)$ und die Priorität des Tasks $(50, 150, 250)$.

os_run(os_priority_t idlePrio) startet dann die Ausführung des Betriebssystems. Im Zuge dessen wird ein weiterer Task mit der Priorität **idlePrio** initialisiert, welcher den Idle-Task des Systems darstellt. Dieser Idle-Task läuft immer dann wenn kein anderer Task Rechenzeit benötigt. Um eine korrekte Funktionsweise zu ermöglichen, muss dieser Idle-Task in jedem Prozessorkern ausgeführt werden.

Kapitel 6

Messungen und Simulation

Dieses Kapitel behandelt die genauere Analyse der entwickelten Hardware und Software. Der Ressourcenverbrauch der gesamten Implementierung wird zuerst untersucht. In weiterer Folge wird der Ressourcenverbrauch des Wishbone-Crossbar-Switches einzeln betrachtet. In beiden Fällen wird untersucht, wie sich die Anzahl der Logikblöcke bei unterschiedlicher Anzahl an Komponenten verändert.

Im zweiten Teil dieses Kapitels werden Simulationen und Testergebnisse der Implementierung vorgestellt. Im Verlauf der Arbeit sind einige Testbenches und auch unterschiedlichste Testsoftware für die V-scale-Prozessoren entstanden. Die Ergebnisse in diesem Kapitel spiegeln somit nur einen Teil aller Tests wieder und sollen die korrekte Funktion der Implementierung demonstrieren.

6.1 Ressourcenverbrauch

Bei der Implementierung dieser Arbeit wurde spezieller Wert auf die Erweiterbarkeit des Gesamtsystems gelegt. Die einzelnen Komponenten wurden dahingehend entwickelt, dass diese möglichst einfach konfiguriert und wiederverwendet werden können. Der Ressourcenverbrauch bei unterschiedlichen Konfigurationen ist bei solch einem System von großem Interesse.

Als Referenz für diese Untersuchung dient das Multi-Core-System mit drei V-scale-Prozessorkernen. Der Aufbau von diesem kann in Abschnitt 5.3 des vorherigen Kapitels nachgeschlagen werden. Die zentrale Komponente im System ist der dynamische Wishbone Crossbar-Switch. Dieser wird genauer betrachtet, da die Konfiguration des Gesamtsystems direkten Einfluss auf den Ressourcenverbrauch dieses Moduls nimmt.

Unter Verwendung eines Synthesetools kann der Ressourcenverbrauch im Bezug auf das verwendete FPGA ermittelt werden. Für diese Arbeit wurde ein Artix-7-FPGA von Xilinx eingesetzt. Demzufolge wurde Vivado[32] als Synthesetool verwendet.

Nach der Synthese ist es mit Vivado möglich, einen sogenannten „Utilization report“ zu generieren. Dieser Bericht zerlegt das synthetisierte Design in die unterschiedlichen Ressourcentypen und stellt das Ergebnis in Summe oder hierarchisch dar. Tabelle 6.1 zeigt das Ergebnis für das Referenzsystem:

Modul	Slice LUTs (63400)	Flip-Flops (126800)	Muxes (31700)	Block RAM (135)	IO (210)	BUFG (32)
Nexys4_wb	16,62% 10536	3,04% 3856	0,93% 296	35,56% 48	20,95% 44	6,25% 2
wb_vscale0	2888	1091	47	0	0	0
wb_vscale2	2662	1091	47	0	0	0
wb_vscale1	2482	1091	47	0	0	0
crossbar	1517	48	155	0	0	0
rom	264	82	0	8	0	0
gpio_0	173	214	0	0	0	0
uart_0	171	115	0	0	0	0
ram_2	133	41	0	16	0	0
ram_1	133	41	0	16	0	0
ram_0	132	41	0	8	0	0

Tabelle 6.1: Utilization Report für Referenzsystem mit drei V-scale Kernen

Die Hauptlogikelemente eines Artix-7 FPGA sind sogenannte Configurable Logic Blocks (CLBs) [7]. Jedes dieser CLBs ist innerhalb des FPGA mit der Switch-Matrix verbunden und besteht aus zwei Slices. Ein Slice besteht wiederum aus vier 6-Input Look-Up Tables (LUTs) und deren acht Flip-Flops sowie Multiplexer und Carry-Logik. Ungefähr ein Drittel aller Slices sind SLICEM, welche ihre LUTs auch als 64-bit RAM oder 32-bit Schieberegister einsetzen können. Abbildung 6.1 zeigt die innere Struktur eines CLB:

[32] https://www.xilinx.com/products/design-tools/vivado.html

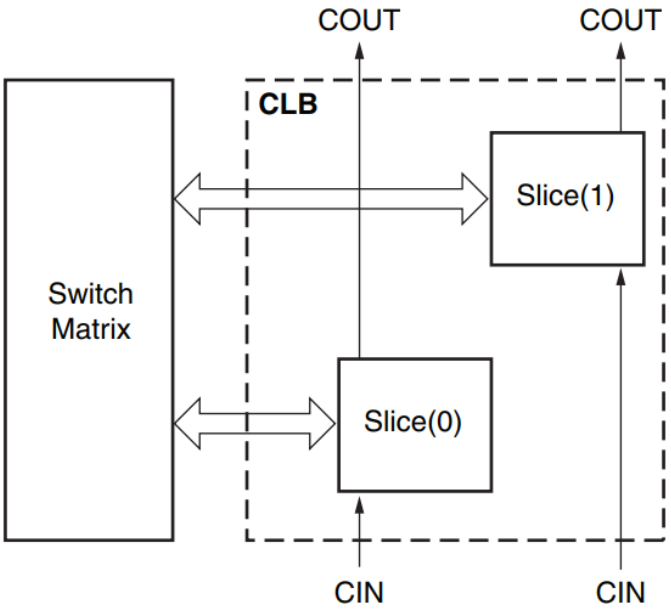

Abbildung 6.1: Struktur der Slices innerhalb eines CLB [7]

Das vom Nexys4-Board eingesetzte Artix-7-FPGA ist das Modell 7A100T. Die einzelnen Modelle unterscheiden sich in der Größe, also bei der Anzahl ihrer CLBs. Tabelle 6.1 zeigt in der Kopfzeile, in Klammer, jeweils die Anzahl der vorhandenen Ressourcentypen für das Modell 7A100T. Diese Werte bilden auch die Referenz für die erste Zeile (`Nexys4_wb`) in der Tabelle, um den Ressourcenverbrauch in Prozent angeben zu können.

`BUFG` [33] sind die am häufigsten verwendeten Takt-Ressourcen. Diese sind globale Taktgeber und können mit jedem Taktpunkt innerhalb des FPGAs verbunden werden.

Die zweite Zeile `Nexys4_wb` in der Tabelle zeigt den Ressourcenverbrauch des Systems in konkreten Zahlen. Gefolgt von den einzelnen Modulen, aus denen das System zusammengesetzt ist, sortiert nach der Anzahl der verwendeten Slice-LUTs. Naturgemäß benötigen die drei V-scale-Prozessorkerne am meisten Ressourcen. Gefolgt vom Wishbone-Crossbar-Switch.

Die drei V-scale-Prozessorkerne sind identisch parametrisiert. Der unterschiedliche Ressourcenverbrauch ergibt sich aufgrund der Optimierung vom Synthesetool.

[33] https://www.xilinx.com/support/documentation/user_guides/ug472_7Series_Clocking.pdf

6.1.1 Vergleich mit Single- und Dual-Core-System

Drei unterschiedliche Konfigurationen wurden für die folgende Untersuchung herangezogen. Neben dem Multi-Core-System mit drei V-scale-Prozessorkernen, ein Dual-Core-System und eines mit nur einem V-scale-Kern. Aufgrund des Aufbaus reduziert dies auch die Anzahl an RAM-Einheiten im System.

	Ein V-scale Kern		Zwei V-scale Kerne		Drei V-scale Kerne	
		Crossbar		Crossbar		Crossbar
#Master		2		4		6
#Slave		5		6		7
Flip-Flop	1556	12	2704	28	3856	48
LUT	3937	**179**	7630	**977**	11510	**1517**
MUXFX	53	0	94	0	296	155
CARRY	320	2	617	2	926	2
BMEM	16	0	32	0	48	0
DMEM	128	0	224	0	320	0
NETS	44	351	44	1238	44	1753
Clk Inst	1603	12	2795	28	3991	48
Slack (ns)	8,429		8,429		8,388	
Max. Freq. (MHz)	86,4		86,4		86,1	

Tabelle 6.2: Ressourcenverbrauch bei unterschiedlicher Anzahl an V-scale-Prozessorkernen

Tabelle 6.2 vergleicht den Ressourcenverbrauch der Systeme und des jeweiligen Crossbar-Switches. Zusätzlich zu den Statistiken in der vorherigen Tabelle gibt es noch weitere Informationen. `CARRY` ist die benötigte arithmetische Carry-Logik, Distributed-RAM (`DMEM`) ist im Vergleich zum Block-RAM (`BMEM`), der über LUT realisierte verteile Speicher im System. `NETS` kommen von den Net Boundary Statistics und geben Verbindungen an, die über eine Komponente hinausgehen. Anhand der Crossbar-Switches ist dies leicht zu erkennen, da diese eine hohe Anzahl an `NETS` besitzen, um alle Komponenten im System miteinander zu Verbinden. `Clk Inst` gibt an, wie viele Taktinstanzen vom jeweiligen Modul benötigt werden.

Ein wichtiger Punkt, der von dieser Tabelle abzuleiten ist, ist das nichtlineare Wachstum des Crossbar-Switches. Die erwartete Komplexität liegt bei $O(M \cdot S)$, wobei M für die Anzahl der Master und S für die Anzahl der Slaves steht. Die internen zweidimensionalen Arrays des Crossbar-Switches sind hauptverantwortlich für den resultierenden Ressourcenverbrauch.

Im implementierten Gesamtsystem steigt bei jedem weiteren V-scale-Kern die Anzahl der Komponenten, die ein Crossbar-Switch handhaben muss, um 3. Zwei Master-Anbindungen des V-scale-Kerns und eine zusätzliche Slave-Anbindung für den benötigten RAM.

Die letzten beiden Einträge der Tabelle gehen auf das Timing der Systeme ein. Mit Hilfe des Synthesetools kann die Slack-Zeit bestimmt werden. Bei den Werten in der Tabelle handelt es sich um den "Worst Negative Slack (WNS)" im Bezug auf die Frequenz des Systems. Für diese Werte wird die Zeit gemessen die ein Signal von einem Logik-Element zu einem anderen benötigt. Entscheidend ist also die längste Verbindung auf dem gerouteten FPGA.

Die Frequenz wurde bei allen Systemen mit $50MHz$ festgelegt. Dies entspricht einer Periodendauer (t_p) von $20ns$. Die positiven Slack-Zeiten (t_s) bedeuten nun das die Timing-Bedingungen auch noch erfüllt werden wenn dieser Wert zur längsten Signallaufzeit addiert wird. Die längste Signallaufzeit ist daher die Periodendauer minus den Slack-Zeiten in der Tabelle. Mit diesem Ergebnis kann die maximale theoretische Frequenz (f_{max}) der Systeme ermittelt werden:

$$f_{max} = \frac{1}{(t_p - t_s)} \tag{6.1}$$

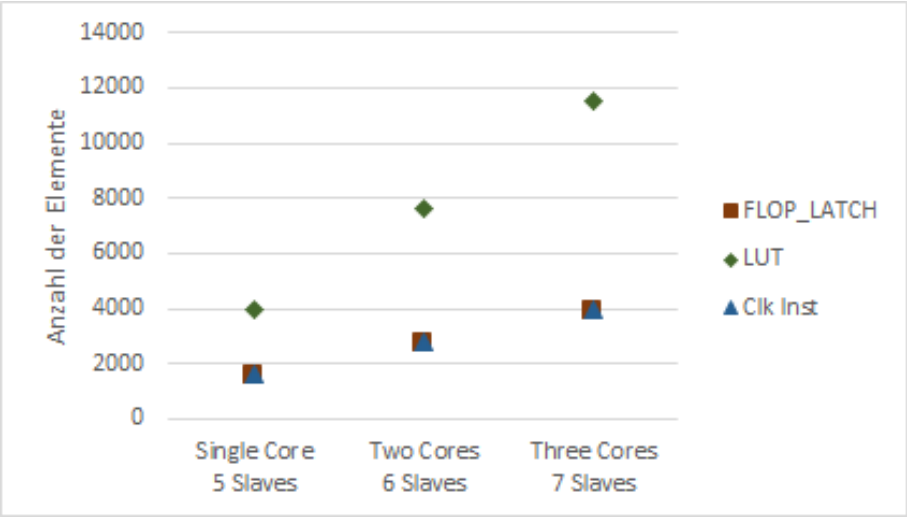

Abbildung 6.2: Ressourcenverbrauch der unterschiedlichen Systeme

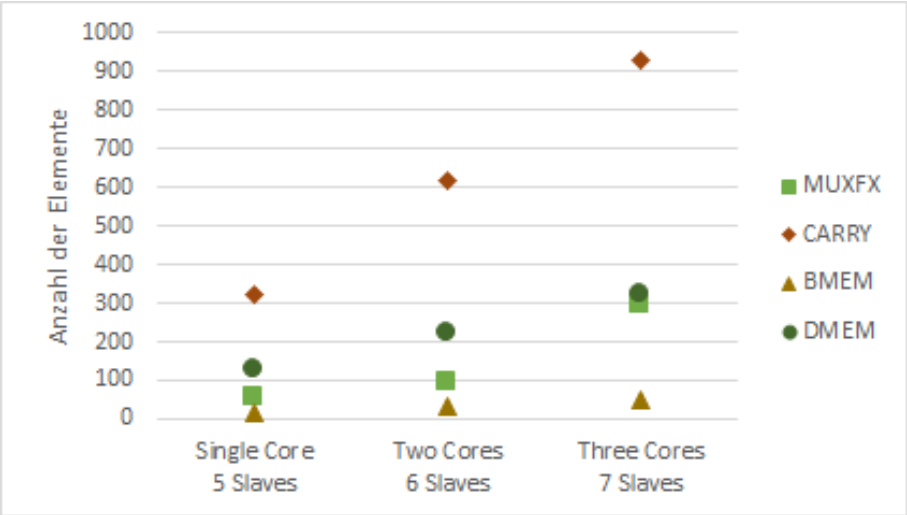

Abbildung 6.3: Ressourcenverbrauch der unterschiedlichen Systeme

Abbildung 6.2 und Abbildung 6.3 zeigen nochmals die Ergebnisse von Tabelle 6.2. Aufgrund der unterschiedlichen Wertebereiche, wurden die Elemente in zwei Diagramme aufgeteilt. Da sich die Anzahl der `NETS` für die drei Gesamtsysteme nicht ändert, wurde dieses Element nicht mit in die Diagramme aufgenommen. Der Verlauf der einzelnen Elemente gibt Aufschluss darüber, in welchem Maß sich der Ressourcenverbrauch verändert, sobald ein weiterer Prozessor hinzugefügt wird.

6.1.2 Auswertung

Anhand der vorherigen beiden Tabellen ist der Ressourcenverbrauch der entwickelten System zu erkennen. Mit Ausnahme des Crossbar-Switches benötigen alle anderen Komponenten nur eine festgelegte Anzahl an Logikblöcken. Der Crossbar-Switch wächst jedoch mit jeder neuen Komponente im System nichtlinear weiter. Bei dem größten System mit drei V-scale-Kernen benötigt der Crossbar-Switch bereits $14,4\%$ aller LUTs und die Hälfte aller Multiplexer im System.

Bei zukünftigen Systemen mit einer höheren Anzahl an Komponenten, müsste der Sinn des Crossbar-Switches hinterfragt werden. Die benötigten Ressourcen würden hauptsächlich vom Crossbar-Switch beansprucht werden. In diesem Fall wäre eine andere Bus-Topologie auf Kosten der Bandbreite des Systems zu empfehlen. Wie in Kapitel 3 bereits vorgestellt, würden sich hier moderne NoCs besonders gut eignen.

Tabelle 6.3 zeigt die Auswertung des Ressourcenverbrauchs bei unterschiedlicher Datenbreite der Task-Priorität. Die Anzahl der benötigten Multiplexer wird am stärksten durch die Task-Priorität beeinflusst. Dies lässt sich damit begründen, dass die Task-Priorität im Crossbar-Switch zur Arbitrierung genutzt wird.

	Task-Priorität (Bit)		
	2	8	32
Flip-Flop	3838	3856	3928
LUT	11214	11510	11510
MUXFX	129	296	368
CARRY	891	926	926
Clk Inst	3973	3991	4063

Tabelle 6.3: Ressourcenverbrauch bei unterschiedlicher Breite der Task-Priorität im System mit drei V-scale-Kernen

6.2 Messungen und Simulation der Implementierung

Parallel zur Implementierung wurden stetig Simulationen durchgeführt und Testsoftware entwickelt, um die korrekte Funktion der Implementierung zu überprüfen. Abschließend wurden dann Messungen an der Hardware durchgeführt, um die fertiggestellten Gesamtsysteme zu testen.
Simulationen machten daher den größten Teil der Funktionstests aus und wurden mit Vivado Simulator 2016.2 durchgeführt. Visualisiert wurden die Simulationen mit GTKWave[34], einem sogenannten Waveform-Viewer der für alle gängigen Betriebssysteme frei zur Verfügung steht.

6.2.1 Wishbone Crossbar-Switch-Testbench

In der ersten Phase der Implementierung wurde der Wishbone-Crossbar-Switch entwickelt. Um die Funktion zu überprüfen und den Aufbau von den größeren Systemen zu ermöglichen, wurde eine simple Testbench entwickelt. Diese Testbench simuliert den Zugriff von mehreren Mastern auf einen Slave über den Crossbar-Switch.

[34] http://gtkwave.sourceforge.net/

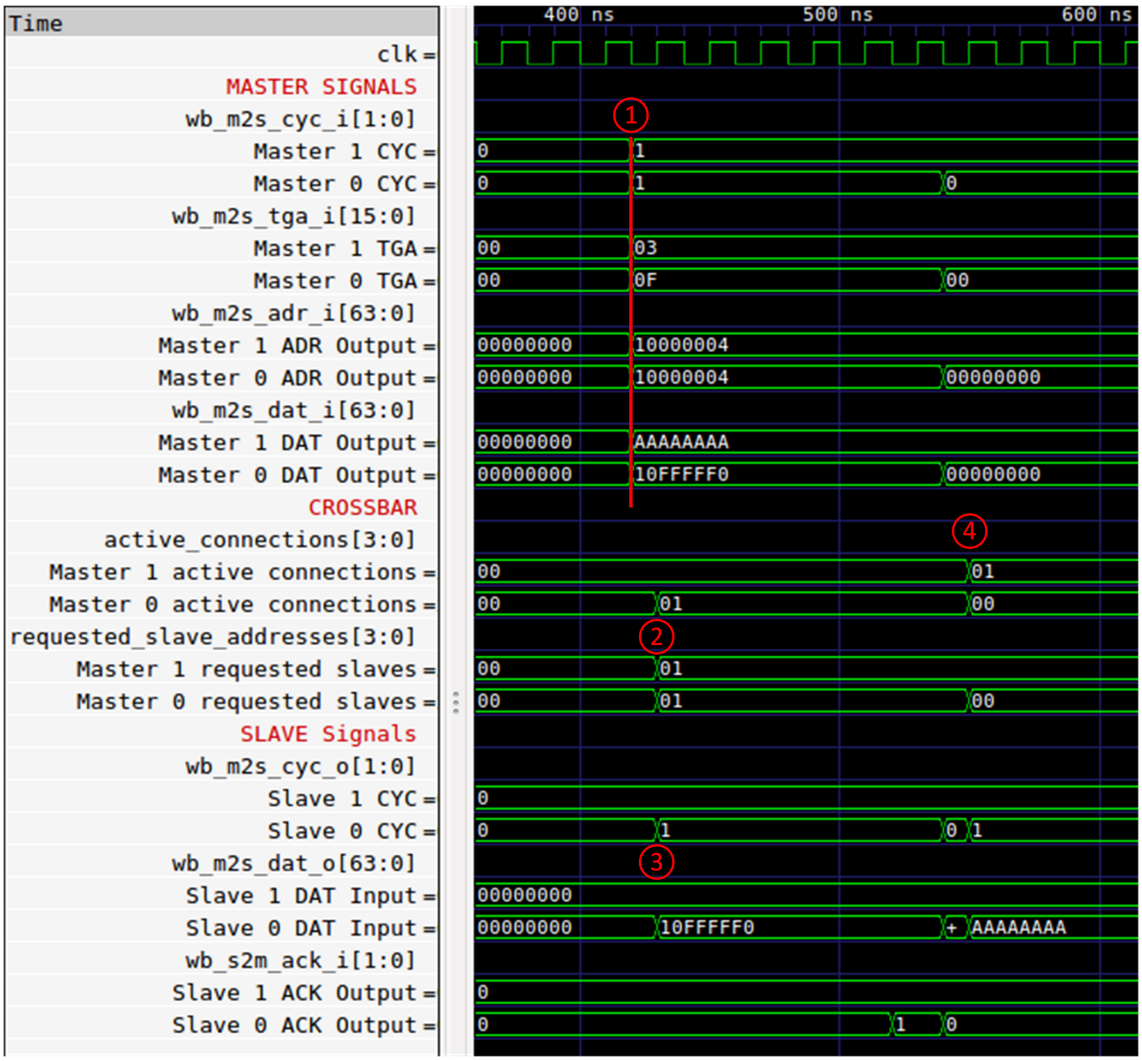

Abbildung 6.4: Ergebnisse der Crossbar-Switch Testbench

Abbildung 6.4 zeigt einen Ausschnitt aus den Ergebnissen der Testbench für zwei Master und zwei Slaves. Im Speziellen zeigt dieser Ausschnitt den simultanen Zugriff von zwei Mastern auf ein und denselben Slave.
Um die Übersichtlichkeit in der Darstellung zu wahren, wurden die zweidimensionalen Arrays je Master aufgespalten. Die Einträge ohne Signalverlauf sind die nicht aufgespaltenen Originale der entsprechenden Signale.

Die Signale im ersten Abschnitt werden von den beiden Mastern generiert. `CYC` signalisiert, wenn ein Wishbone-Zyklus aktiv ist. `TGA` ist das TAG-Signal, das der Adresse zugeordnet ist und welches die Task-Prioritäten überträgt. `ADR` ist die Zieladresse für den aktiven Wishbone-Zyklus. `10000004` entspricht der Adresse des zweiten Slaves in diesem System.

Im zweiten Abschnitt sind die internen Signale des Crossbar-Switches abgebildet. `requested_slave_addresses` fasst alle Anfragen der verbundenen Master auf ein Array zusammen. Für jeden Master ist ein Bit für jeden möglichen Slave vorhanden und der Index entscheidet darüber welcher Slave ausgewählt wurde. Dementsprechend bedeutet ein gesetztes Bit, dass die Adresse die der Master mit `wb_m2s_adr_i` übermittelt, für den betreffenden Slave bestimmt ist. `active_connections` zeigt dann die Verbindungen, die vom Crossbar-Switch ausgewählt wurden und somit Daten übertragen können.

Bei den Signalen der Master (1) ist zu erkennen, dass diese simultan einen neuen Zyklus starten. Auch die Adresse des Zielslaves ist bei beiden gleich. Der Master mit dem niedrigeren Index weist jedoch eine höhere Priorität (höherer Wert) auf (`TGA`). Die Verbindung von diesem Master wird daher auf aktiv gesetzt (2).

Dem entsprechenden Slave mit dem Index 0 werden die Signale vom anfragenden Master weitergeleitet (3). Die Verbindung bleibt solange bestehen bis der Master das `CYC` Signal zurücksetzt.
In diesem Beispiel wartet der zweite Master und der Zyklus (`CYC`) bleibt aktiv. Im Anschluss an die Transaktion des ersten Masters wird die Verbindung des zweiten Masters aktiv gesetzt (4).

Im Gegensatz zu allen anderen Implementierungen für diese Arbeit, wurde die Crossbar-Switch-Testbench in SystemVerilog entwickelt. Dies war zu Beginn dieser Arbeit nützlich, da Verilog Probleme mit zweidimensionalen Arrays hat, die in SystemVerilog jedoch ohne großen Aufwand behandelt werden können. In Kapitel 5.2.2 sind die Probleme mit Arrays in Verilog kurz zusammengefasst. Ungeachtet dessen benutzt die Testbench nur Verilog kompatible Befehle.

6.2.2 Zugriffszeit über den Wishbone-Crossbar-Switch

Abbildung 6.5 zeigt den Zugriff eines V-scale-Prozessors auf eine Speicherstelle im ROM. Anhand dieser Abbildung kann die Zugriffszeit über den Crossbar-Switch analysiert werden.

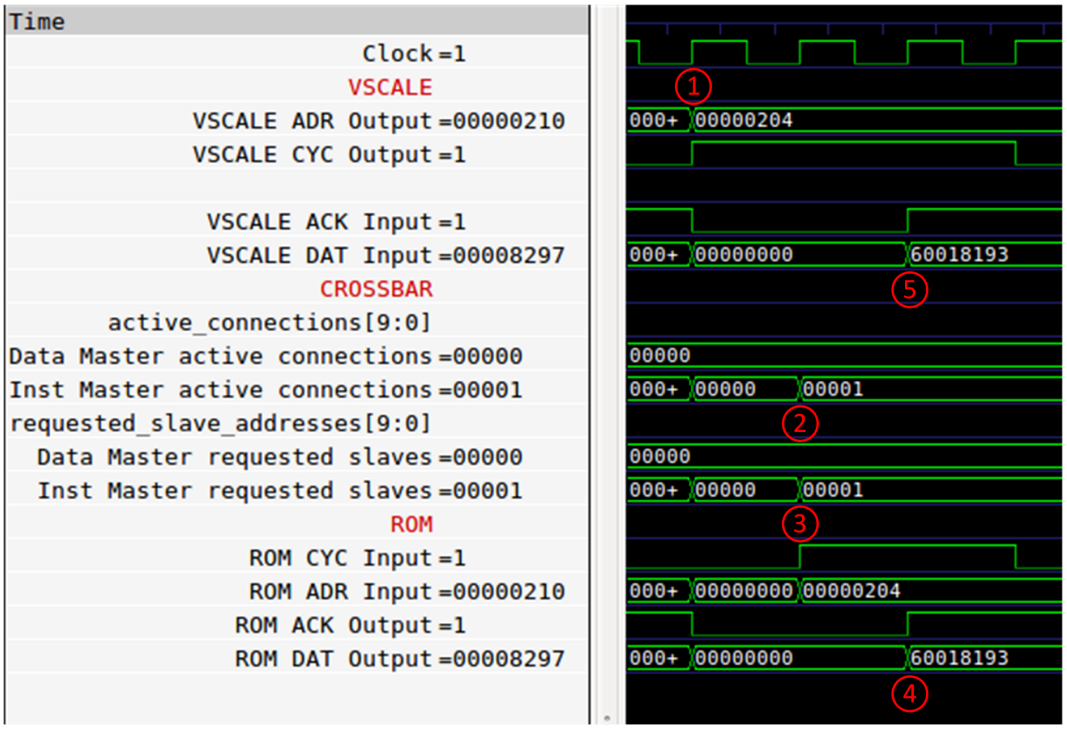

Abbildung 6.5: Zugriff auf den ROM über den Wishbone-Crossbar-Switch

Nachdem die Adresse (`ADR`) gesetzt und der Wishbone-Zyklus (`CYC`) gestartet wurde (1), benötigt der Crossbar-Switch einen Taktzyklus, um die Verbindung herzustellen (2).
Neben den beiden V-scale-Mastern (Instruktionen und Daten) gibt es fünf Slaves in diesem System. Das ROM mit zwei Schnittstellen (Dual-Port), RAM, GPIO und UART. Demzufolge ist das zweidimensionale Array, welches im Crossbar-Switch die aktiven Verbindungen speichert, 10 Bit breit. Das letzte gesetzt Bit (`00001`) zeigt also die Verbindung zwischen dem Master mit Index 0 (Instruktions-Bus) und dem Slave mit Index 0 (Port A vom ROM).

Nachdem die Signale vom Master an das ROM-Modul weitergeleitet wurden (3), benötigt die Wishbone-Schnittstelle im ROM einen Taktzyklus, um die Daten bereitzustellen und das Bestätigungssignal (`ACK`) zu setzen (4).

Der Speicherzugriff über den Crossbar-Switch benötigt zwei Taktzyklen. Die Wishbone-Schnittstelle im V-scale-Prozessor braucht jedoch einen weiteren Taktzyklus, um die Daten vom Bus zu übernehmen und zu Beginn einen weiteren Takt, um die Daten an den Bus zu übergeben. Folglich werden insgesamt **vier Taktzyklen** für einen kompletten Speicherzugriff benötigt, wenn der Slave für eine Verbindung verfügbar ist.

6.2.3 Gesamtsystem: Simulation und Messung an der Hardware

Mit der Fertigstellung der Wishbone-Anbindung für den V-scale-Prozessor, konnte das erste Gesamtsystem aufgebaut werden. Zum Testen des Systems wurde eine neue Testbench entwickelt, die den Bootvorgang für alle vorhandenen V-scale-Kerne übernimmt. Die Testbench wurde im Verlauf der Arbeit dann nicht mehr modifiziert. Über unterschiedliche Testsoftware, die über die Testbench geladen wird, können die unterschiedlichsten Tests durchgeführt werden. Die Struktur der Testprogramme wurde bereits in Kapitel 5.4.2 behandelt.

Neben den Simulationen wurden auch Messungen direkt an der Testhardware durchgeführt. Zu diesem Zweck wurde das digitale Oszilloskop PicoScope 2205 MSO[35] verwendet. Das Oszilloskop bietet 2 analoge und 16 digitale Eingänge, die simultan arbeiten. Für diese Arbeit wurden nur die digitalen Eingänge benötigt, die eine maximale Input-Frequenz von 100Mhz unterstützen. Ausgewertet wurden die Signale über die entsprechende Software PicoScope 6.

[35] https://www.picotech.com/products/oscilloscope

Abbildung 6.6: Anschluss des PicoScopes an das Nexys4-Board

Übersicht über des Gesamtsystems mit drei V-scale-Prozessoren

Das komplexeste System, welches für diese Arbeit entwickelt wurde, besteht aus drei V-scale-Prozessoren. Wie in Kapitel 5.3 genauer beschrieben, beinhaltet das System zusätzlich ein GPIO- und ein UART-Modul.
Jeder V-scale ist mit zwei Wishbone-Schnittstellen ausgestattet, um Instruktionen und Daten unabhängig voneinander zu bearbeiten. Dementsprechend gibt es in diesem System sechs aktive Wishbone-Master. Wishbone-Slaves gibt es sieben, diese sind der ROM mit zwei Ports, GPIO, UART und die drei RAMs für jeden V-scale.

Die verwendete Testsoftware ist in Anhang A zu finden. Da auch die ROM-Zugriffe über den Crossbar-Switch erfolgen, konnte dieses Modul besonders belastet und getestet werden. Gleichzeitig sollten auch das GPIO- und das UART-Modul verwendet werden. Zum einen, um die Ausgaben dieser beiden Module für die Messungen zu verwenden und zum anderen, um auch die Wishbone-Schnittstelle dieser beiden Module zu testen.

Die drei V-scale-Prozessoren übernehmen folgende Aufgaben:

- Core 0 (Priorität **0x32**) - schreibt Daten über die UART-Schnittstelle
- Core 1 (Priorität **0x96**) - erzeugt ein Lauflicht, das die 8 rechten Leuchtdioden (Index 0 bis Index 7) des Nexys4-Boards verwendet. Dies wird mit Hilfe des GPIO-Moduls realisiert.
- Core 2 (Priorität **0xFA**) - arbeitet gleich wie Core 1, beschreibt jedoch nur die 8 linken Leuchtdioden (Index 8 bis Index 15).

Die Prioritäten für die unterschiedlichen Tasks wurden demonstrativ im gesamten 8-Bit Wertebereich verteilt. Die Boot-Priorität der V-scale-Prozessoren liegt genau in der Mitte des Wertebereichs also bei **0x7F**.

Simulation mit Fokus auf GPIO und UART

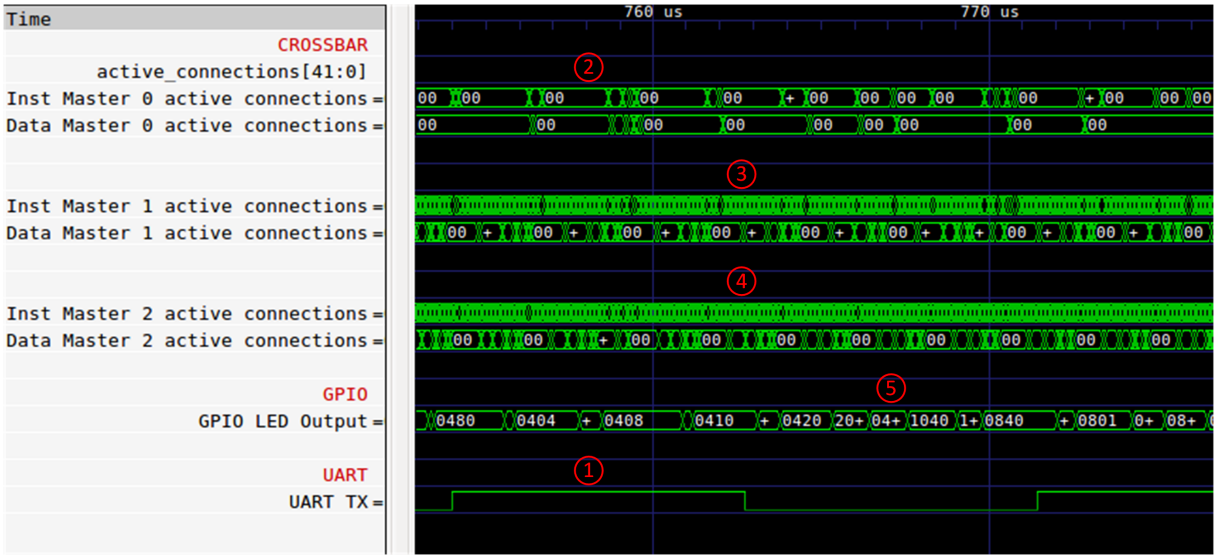

Abbildung 6.7: Simulation mit drei V-scale-Kernen (Fokus auf GPIO und UART)

Abbildung 6.7 zeigt die Simulation des Systems bei laufender Testsoftware mit Fokus auf dem GPIO- und dem UART-Modul.
Die Abbildung soll die Aktivität der drei V-scale-Prozessoren aufzeigen. Im ersten Abschnitt wurde das zweidimensionale Array `active_connections` aufgespalten, um die aktiven Wishbone-Verbindungen je Master zu zeigen. Im unteren Bereich sind die Ausgänge der beiden Module dargestellt.

Der Task, der auf Prozessorkern 0 arbeitet, schreibt Daten über die UART-Schnittstelle (1). Für UART-Übertragungen müssen bestimmte Zeitvorgaben eingehalten werden, um die eingestellte Baudrate sicherzustellen. Das UART-Modul benötigt die Daten daher nicht so schnell wie Prozessorkern 0 diese zur Verfügung stellt.
Als Resultat muss nicht immer eine Wishbone-Verbindung hergestellt werden. Im entsprechenden Array (2) ist somit häufiger eine Null zu finden im Vergleich zu den Arrays der beiden anderen Prozessorkernen.

Die beiden anderen Prozessorkerne, die jeweils 8 Leuchtdioden über das GPIO-Modul beschreiben, benötigen weit häufiger eine Verbindung über den Bus. Dies ist leicht an der erhöhten Aktivität im `active_connections` Array zu erkennen ((3), (4)).
Im Ausgangssignal des GPIO-Moduls (5), ist zu erkennen, wie die beiden Prozessorkerne dieses Signal beeinflussen. Die 8 niederwertigen Leuchtdioden (0x01 bis 0x80) sowie die 8 höherwertigen (0x100 bis 0x8000) werden abwechselnd gesetzt.

Mit Hilfe dieser Simulation, konnte die korrekte Funktion der beiden Module (GPIO und UART) über den Crossbar-Switch bestätigt werden. Darüber hinaus konnte die parallele Arbeit von mehreren Prozessorkernen untersucht werden.
In einer alternativen Testsoftware wurden Wartezeiten integriert, um die Änderungen der Leuchtdioden in regelmäßigen Abstand durchzuführen und mit freiem Auge erkennbar zu machen.

Simulation mit Fokus auf den Crossbar-Switch

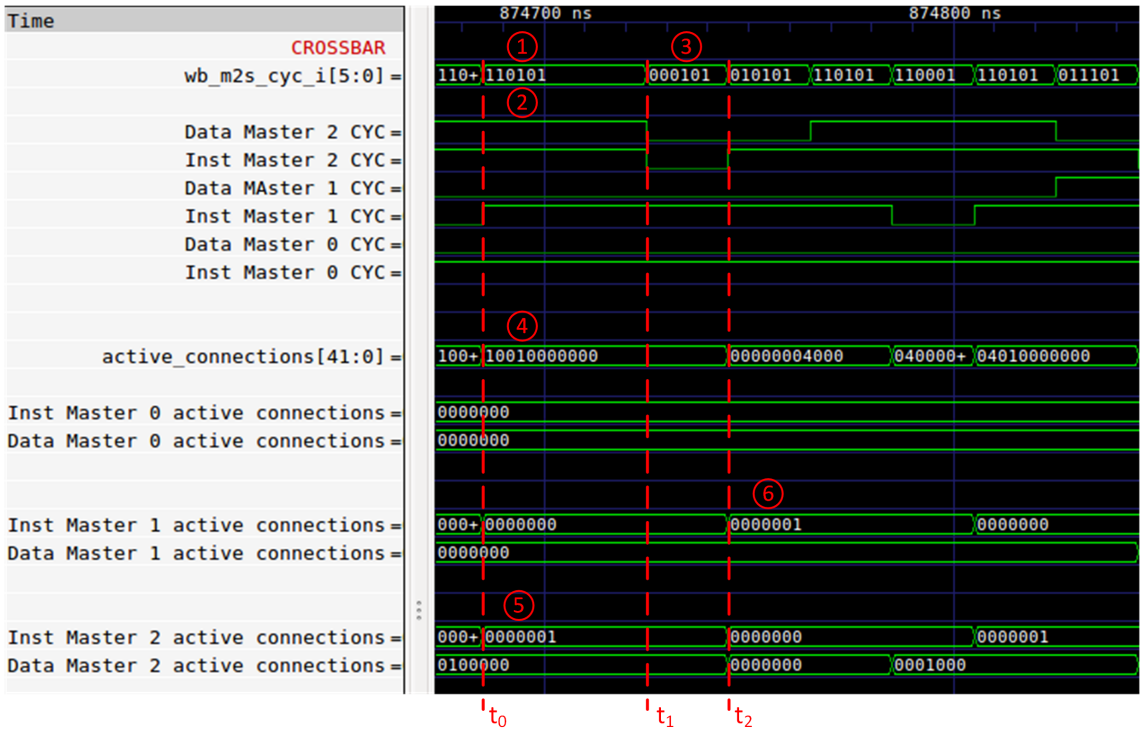

Abbildung 6.8: Simulation mit drei V-scale-Kernen (Fokus auf Crossbar-Switch)

Abbildung 6.8 ist Teil der selben Simulation wie zuvor, fokussiert sich aber mehr auf die Aktivitäten im Crossbar-Switch. Änderungen beim GPIO- oder UART-Modul sind in dieser Ansicht nicht mehr zu erkennen.

Im ersten Abschnitt dieser Abbildung ist das Array mit den Zyklussignalen (`wb_m2s_cyc_i`) dargestellt (1). An diesen Werten ist zu erkennen, falls ein Master Zugriff über den Bus benötigt.
Das zweidimensionale Array `active_connections` (4), ist die Matrix aller Master im System mit allen Slaves in der zweiten Dimension. Bei diesem System besteht `active_connections` daher aus sieben Teilen zu je sechs Bits. Sechs Master-Schnittstellen kommen zustande da ein V-scale-Prozessorkern für Instruktionen und Daten jeweils eine Schnittstelle besitzt. An den Signalen je Master, ist gut zu erkennen, welche Verbindung im Endeffekt aktiv geschaltet wurde.

Anhand von (1) ist zu erkennen, dass zum Zeitpunkt t_0 vier Wishbone-Master gleichzeitig auf einen Slave zugreifen möchten. In diesem Fall besteht die Verbindung der beiden Master vom Prozessorkern 0 bereits. Dies ist daran zu erkennen, dass die Zyklussignale `CYC` bereits länger aktiv sind (2).

Die Funktionsweise des Crossbar-Switches ist gut bei (1) bzw (4) zu sehen. Von den vier Mastern, die eine Verbindung benötigen, sind nur die zwei Verbindungen vom Prozessorkern 2 aktiv (5).

Sobald diese beiden Verbindungen zum Zeitpunkt t_1 abgearbeitet wurden, sind die beiden immer noch aktiven Zyklussignale der anderen Prozessorkerne an der Reihe (3). Da Prozessorkern 0 und Prozessorkern 1 eine Verbindung zum selben Slave herstellen möchten, Prozessorkern 1 jedoch eine höhere Priorität besitzt, wird die Verbindung von diesem nach einem Taktzyklus (t_2) aktiv geschalten (6).

Ein Slave darf immer nur einmal aktiv sein und in den sieben Bits je Master darf auch immer nur ein Bit gesetzt sein. Das heißt, auf einen Slave kann nicht doppelt zugegriffen werden und ein Master kann nicht auf zwei Slaves zugreifen.

Simulation mit Fokus auf die Task-Prioritäten

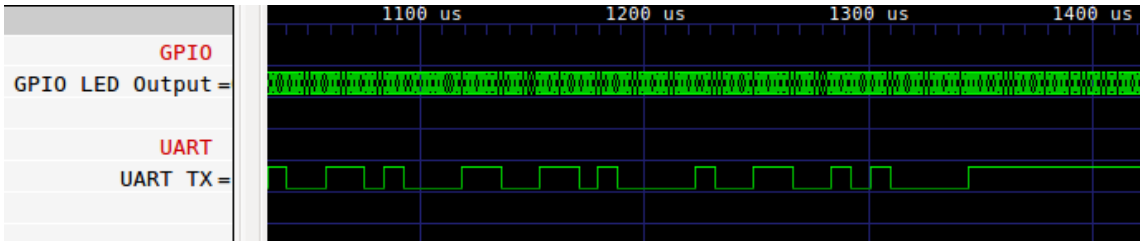

Abbildung 6.9: Simulation mit drei V-scale-Kernen (UART-Task mit niedrigster Priorität)

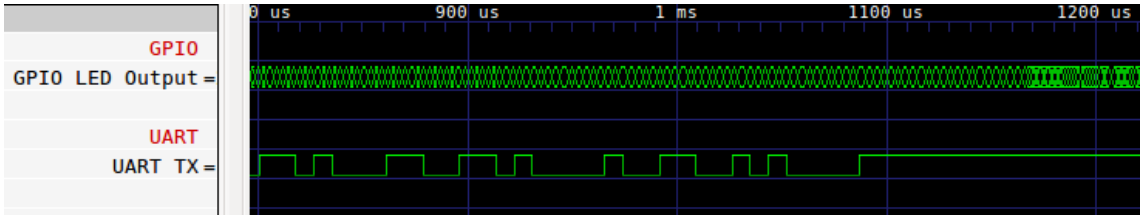

Abbildung 6.10: Simulation mit drei V-scale-Kernen (UART-Task mit höchster Priorität)

Abbildung 6.9 und Abbildung 6.10 zeigen wieder eine ähnliche Simulation wie zuvor. In diesem Fall wurde das Testprogramm für das UART-Modul so verändert, dass nur eine kurze Zeichenkette ausgegeben wird. Dadurch kann das Ende dieser Aufgabe problemlos bestimmt werden.

Durchgeführt wurde diese Simulation in zwei Durchgängen. Im Ersten wurde die Priorität des UART-Tasks niedriger als bei den beiden anderen Prozessorkernen festgelegt. Im zweiten Durchgang wurde dann die höchste Priorität gewählt.

Bedingt durch die höhere Priorität, kann das Senden der Zeichenkette über die UART-Schnittstelle, schneller abgeschlossen werden. Mit niedriger Priorität werden **1344µs** benötigt und bei höchster Priorität nur **1087µs**. Vergleicht man die GPIO-Ausgangssignale, so kann man auch hier die unterschiedliche Auslastung erkennen. Naturgemäß wird der GPIO-Ausgang weniger oft verändert, wenn der UART-Task mit der höchsten Priorität arbeitet.

Messung an der Hardware mit drei V-scale-Kernen

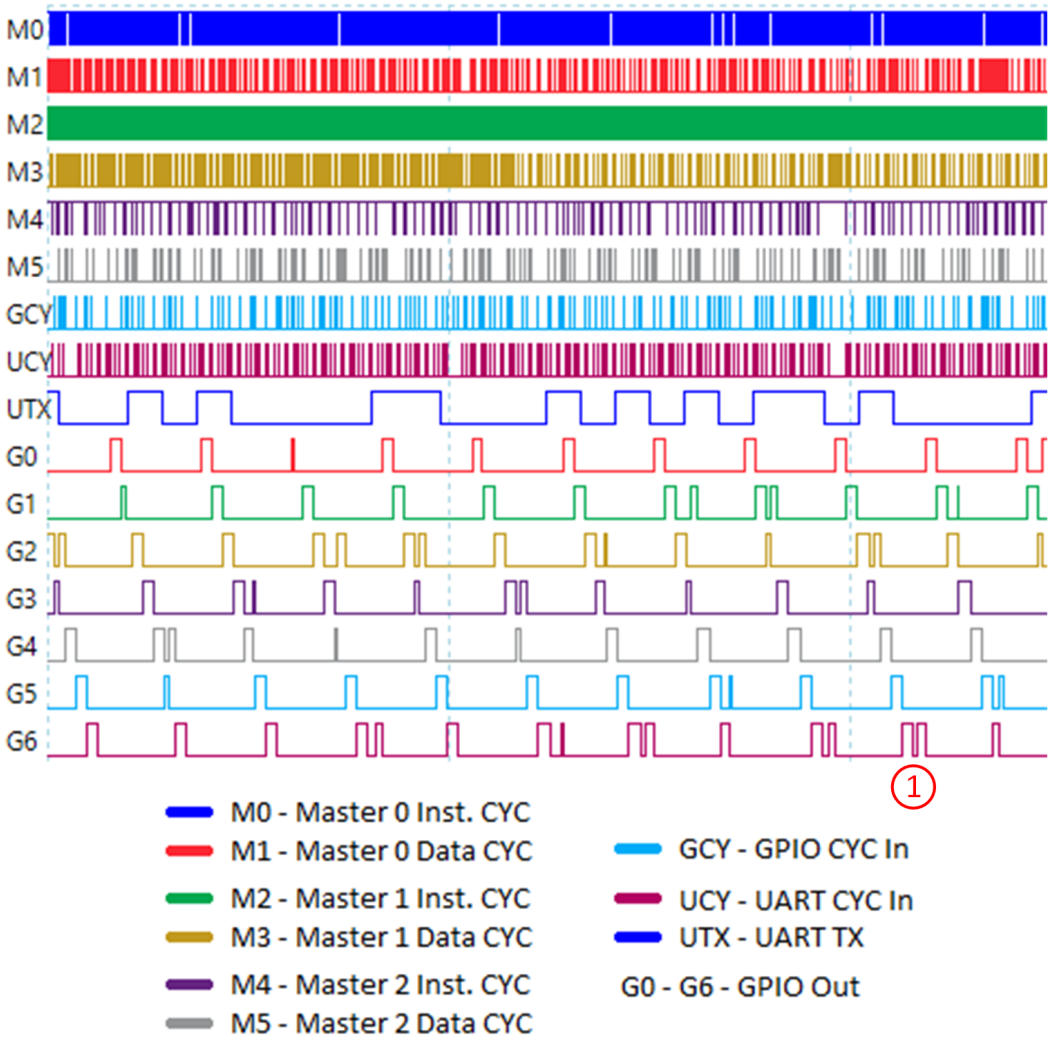

Abbildung 6.11: Messung der CYC-Signale mit drei V-scale-Kernen

Die Messung der Testsoftware an der Hardware wird in Abbildung 6.11 dargestellt. Es wurde die selbe Testsoftware wie bei den Simulationen zuvor verwendet (Abbildung 6.7 und Abbildung 6.8). Aufgrund der begrenzten Anzahl an digitalen Eingängen am Oszilloskop wurde nur eine kleine Auswahl an Signalen gemessen.

Die ersten sechs Signale repräsentieren die Instruktions- und Datenbusse der drei V-scale-Prozessoren. Hierzu wurden die Wishbone-Zyklus-Signale (`CYC`) für die Messung herangezogen. Wie in Kapitel 3.4 erläutert, ist ein Zyklus-Signal immer aktiv, wenn eine Wishbone-Transaktion durchgeführt wird.

Die nächsten beiden Signale (GCY und UCY) sind die Wishbone-Zyklus Signale vom GPIO-Modul und vom UART-Modul.

Die Aktivität der Komponenten lässt sich an diesen Signalen einfach erkennen. Würde ein Prozessor einen Fehler verursachen oder hängen bleiben, dann würden auch die Zyklus-Signale der betroffenen Komponenten ständig im gleichen Zustand bleiben. Zum Beispiel würde das auch passieren, wenn ein Slave nie eine Bestätigung (ACK) an den Master zurücksendet. Instruktionen müssen von den Prozessoren jedoch in jedem Takt aus dem ROM geladen werden.
In dieser Testsoftware arbeitet der Prozessorkern 1 mit der höchsten Priorität, gefolgt von Prozessorkern 0 mit mittlerer Priorität und Prozessorkern 2 mit der niedrigsten Priorität. In der Abbildung ist dies ebenfalls zu erkennen, da das Zyklus-Signal von Prozessorkern 2 (M4) weniger oft aktiv ist, im Vergleich zum Zyklus-Signal von Prozessorkern 1 (M2).

Das Signal UTX ist das Ausgangssignal des UART-Moduls. Zusätzlich entsprechen die letzten sieben Signale (G0 - G6) den Leuchtdioden, die von Prozessorkern 1 beschrieben werden. Von G0 bis G6 werden die Leuchtdioden nacheinander geschaltet. Dies entspricht einem Lauflicht, das von der Testsoftware korrekterweise generiert wird.
Da Prozessorkern 1 und Prozessorkern 2 asynchron arbeiten und kein Locking[36] implementiert wurde, können Leuchtdioden fälschlicherweise zu oft aktiv geschaltet werden. In der Abbildung ist dies anhand von kurz unterbrochenen Aktivphasen zu erkennen (1).

Zusammenfassend kann man mit Hilfe der Simulationen und Messungen die korrekte Funktionsweise der einzelnen Komponenten des Systems bestätigen. Die geringe Anzahl an digitalen Eingängen bei der Messung, konnte durch die optische Kontrolle der Leuchtdioden und der Ausgabe über die UART-Schnittstelle kompensiert werden.

[36] Sperren die einen exklusiven Zugriff auf geteilte Ressourcen ermöglichen.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

Im Verlauf der Prozessorevaluation wurde der ursprünglich ausgewählte J2 Core nicht für die weiteren Teile dieser Arbeit ausgewählt. Dies ist maßgeblich damit zu begründen, dass so gut wie keine Dokumentation für diesen Prozessor zur Verfügung steht. Zusätzlich werden viele spezielle Tools benötigt, um mit dem Prozessor arbeiten zu können. Die Integration und Anpassung des Prozessors benötigt somit einen erhöhten Arbeitsaufwand.

Als Alternative wurde V-scale, ein Prozessor basierend auf der RISC-V-Architektur, ausgewählt. Im Gegensatz zum J2 Core ist V-scale übersichtlicher aufgebaut und eine ausführliche Dokumentation über die Architektur steht zur Verfügung. Wegen Letzterem sind RISC-V-Architekturen weit verbreitet und es existieren die unterschiedlichsten Prozessordesigns.

Die Busevaluierung führte zu dem Ergebnis, dass Wishbone am geeignetsten für diese Arbeit ist. Neben der Vielzahl an freien Wishbone-Komponenten und der Spezifikation die benutzerdefinierte Erweiterung bereits berücksichtigt, lässt sich Wishbone außerdem frei und uneingeschränkt nutzen.

Für die Simulationen und Messungen dieser Arbeit wurde ein Multi-Master System bestehend aus drei V-scale Prozessoren plus Peripherie aufgebaut. Alle diese Komponenten wurden mit einer Wishbone-Schnittstelle ausgestattet und mit dem eigens entwickelten Wishbone-Crossbar-Switch untereinander zu einem Gesamtsystem vernetzt.
Im Gegensatz zu vielen anderen Arbeiten in diesem Bereich erfolgt die Arbitrierung im gesamten System über Task-Prioritäten, die von jedem laufenden Task an das System übertragen werden. Für diesen Zweck wurde das Gesamtsystem und das Betriebssystem dahingehend erweitert.

Des Weiteren wurde besonderer Wert auf eine einfache Erweiterbarkeit des gesamten Systems gelegt. Um dies zu erreichen wurde das Crossbar-Switch-Modul dynamisch implementiert. Das resultierende Modul kann über wenige Parameter an Systeme angepasst werden und generiert dynamisch alle nötigen Verbindungen.
Bei vergleichbaren Implementierungen wird das Verbindungsnetzwerk oftmals nur statisch implementiert. Für Wishbone und andere Bussysteme sind zwar Generatoren verfügbar, diese sind aber externe Tools und ein gewisser Arbeitsaufwand ist bei jeder kleinen Systemanpassung notwendig.

Die Simulationen und die dazugehörige Messung an der Hardware zeigen die korrekte Funktion der implementierten Komponenten. Zudem ist es mit den Task-Prioritäten möglich, die Ausführung für zeitkritische Tasks anzupassen.
Der Ressourcenverbrauch unterschiedlich zusammengesetzter Systeme wurde ebenfalls untersucht. Besonderes Interesse lag hier auf verschiedenen Konfigurationen des Wishbone-Crossbar-Switches.

7.2 Ausblick

Im Verlauf dieser Arbeit sind einige Ideen aufgetaucht, die nicht mehr umgesetzt werden konnten. Folgende Punkte zeigen mögliche Erweiterungen:

Wishbone-Schnittstellen: Der Crossbar-Switch und die implementierten Wishbone-Schnittstellen könnten um zusätzliche optionale Signale erweitert werden. Zum Beispiel könnten somit Übertragungen im Burstmodus ermöglicht werden, oder Pipelining genutzt werden. In der Spezifikation [4] sind Beschreibungen und Beispiele zu diesen optionalen Features zu finden.

V-scale: Beim implementierten V-scale-Prozessor könnte eine alternative Wishbone-Schnittstelle implementiert werden, die den Instruktions-Bus und den Datenbus auf einen Wishbone-Master zusammenfasst.

Erweiterte Tests oder anders zusammengesetzte Systeme: Das Gesamtsystem könnte mit weiteren Komponenten ausgebaut werden und in unterschiedlicher Zusammenstellung getestet werden. Da alle Komponenten der Wishbone-Spezifikation folgen, könnten auch Systeme mit alternativen Wishbone-fähigen Prozessoren aufgebaut werden. Ein weiterer Test wäre die Untersuchung der Echtzeitfähigkeit in Bezug auf Übertragungen am Bus.

Darüber hinaus wäre die Implementierung eines **NoC** ein weiterer Schritt für diese Arbeit. Die Frage, wie sich Task-spezifische Prioritäten in einem NoC umsetzten lassen, wäre von großem Interesse.

Appendix

A Testsoftware

Der folgende Abschnitt zeigt die Testsoftware zum Testen des Systems mit drei V-scale Prozessorkernen. Der Prozessorkern mit dem Index 0 schreibt Daten über das UART-Modul. Die beiden anderen Prozessorkerne arbeiten mit dem GPIO-Modul und greifen auf jeweils auf die eine Hälfte der verfügbaren Leuchtdioden zu.

```
1  #include <mosart_os.h>
2
3  #define GPIO_OUT         *((volatile uint32_t *)0x80001004ul)
4  #define UART_CTRL        *((volatile uint32_t *)0x80002000ul)
5  #define UART_STATUS      *((volatile uint32_t *)0x80002004ul)
6  #define UART_DATA        *((volatile uint32_t *)0x80002008ul)
7
8  #define BAUD_DIV         (50000000/115200 -1 )
9
10 int core0_task0(void);
11 int core1_task0(void);
12 int core2_task0(void);
13
14 int send_char(int data) {
15     while(UART_STATUS & 0x0200);
16     UART_DATA = (unsigned char)data;
17     return 1;
18 }
19
20 void main(void)
21 {
22   uint32_t core_idx = 0;
23
24   asm volatile (" csrr %0, 0xFC0" : "=r" (core_idx));
25
26   switch (core_idx)
27   {
```

```
28      case 1:
29        os_register_task(core0_task0, 8*80, 50);
30        os_run(0);
31        break;
32
33      case 2:
34        os_register_task(core1_task0, 8*80, 150);
35        os_run(0);
36        break;
37
38      case 3:
39        os_register_task(core2_task0, 8*80, 250);
40        os_run(0);
41        break;
42
43      default:
44        break;
45    }
46  }
47
48  int core0_task0(void)
49  {
50    UART_CTRL =  BAUD_DIV << 16;
51    printfSetStreamwriter(send_char);
52    printfx("Task 0 started!\n");
53    uint32_t i = 0;
54
55    while(1)
56    {
57      printfx("Task 0 is at iteration #%d!\n", i);
58      i++;
59    }
60  }
61
62  int core1_task0(void)
63  {
64    while(1)
65    {
66      for(uint32_t i = 0; i < 8; i++)
67      {
68        GPIO_OUT =  (GPIO_OUT & 0xFF00) | (0x1 << i);
69
70        // sleep(1000000); // Wird nur benoetigt um das Schalten
71        // der LEDs mit freiem Auge erkennen zu koennen
72      }
73    }
```

```
74 }
75
76 int core2_task0(void)
77 {
78   while(1)
79   {
80     for(uint32_t i = 15; i > 7; i--)
81     {
82       GPIO_OUT = (GPIO_OUT & 0x00FF) | (0x1 << i);
83
84       // sleep(1000000); // Wird nur benoetigt um das Schalten
85       // der LEDs mit freiem Auge erkennen zu koennen
86     }
87   }
88 }
```

B Wishbone-Datenblatt

Folgendes Datenblatt beschreibt die für diese Arbeit entwickelte Wishbone Schnittstelle des V-scale (RISC-V) Prozessors.

```
1  ****************************************************************
2  * WISHBONE DATASHEET (2 Wishbone Interfaces | Instruction & Data)
3  *
4  *  1. Revision Level          B.4
5  *  2. Type of interface       Master / Master
6  *  3. Defined signal names    RST_I => rst
7  *                             CLK_I => clk
8  *                             DAT_I => iwbm_dat_i / dwbm_dat_i
9  *                             DAT_O => iwbm_dat_o / dwbm_dat_o
10 *
11 *                             ADR => iwbm_adr_o / dwbm_adr_o
12 *                             CYC => iwbm_cyc_o / dwbm_cyc_o
13 *                             SEL => iwbm_sel_o / dwbm_sel_o
14 *                             STB => iwbm_stb_o / dwbm_stb_o
15 *                              WE => iwbm_we_o  / dwbm_we_o
16 *                             CTI => iwbm_cti_o / dwbm_cti_o
17 *                             BTE => iwbm_bte_o / dwbm_bte_o
18 *                             TGA => iwbm_tga_o / dwbm_tga_o
19 *
20 *                             ACK => iwbm_ack_i / dwbm_ack_i
21 *                             ERR => iwbm_err_i / dwbm_err_i
22 *                             RTY => iwbm_rty_i / dwbm_rty_i
23 *
24 *  4. Master ERR_I            Supported - Forwards error signal
25 *                                   to V-scale control unit
26 *  5. Master RTY_I            Unsupported
27 *  6. Tag signals             TGA - Address tag associated with
28 *                                   ADR (wb_m2s_adr_i)
29 *                                   Specifies the task priority
30 *                                   (arbitration information)
31 *
32 *  7. Port size                     32-bit
33 *  8. Port granularity              8-bit
34 *  9. Data transfer ordering        Little Endian
35 * 10. Maximum operand size          32-bit
36 * 11. Data transfer sequencing      Undefined
37 * 12. Constraints on the CLK_i      None
38 ****************************************************************
```

Listing 1: V-scale Wishbone-Datenblatt

Die generellen Anforderungen für dieses Datenblatt sind in der Wishbone-Spezifikation [4] unter 2.1.1 zu finden.

Begonnen wird das Datenblatt mit der Angabe der verwendeten Spezifikation. In **Punkt 2** muss angegeben werden ob es sich um eine Master- oder Slave-Schnittstelle handelt. Beim Beispieldatenblatt wurde `Master / Master` angegeben, da sich die Schnittstelle für Instruktionen und die für den Datenbus im selben Verilog-Modul befindet.
Die darauffolgende Liste (**Punkt 3**) referenziert jedes verwendete Wishbone-Signal auf die Signalnamen, die in der Spezifikation definiert sind.

Punkt 4 und **Punkt 5** geben an, ob die optionalen Fehlersignale (Error) oder Wiederholungssignale (Retry) unterstützt werden. Ist dies der Fall, muss spezifiziert werden wie das Modul auf diese Signale reagiert.
Im gleichen Sinne behandelt **Punkt 6** die optionalen Tag-Signale. Bei diesen muss der Typ des eingesetzten Tag-Signales sowie die Zugehörigkeit angegeben werden.

Punkt 7 gibt die Datenbreite der Wishbone-Schnittstelle an. **Punkt 8** definiert die Granularität und somit die minimale Größe eines Datenpaketes. Um ein solches Datenpaket zu übertragen, welches nicht die komplette Busbreite ausnutzt, werden zwei weitere Informationen benötigt. Zum einen die Anordnung der Daten, die in **Punkt 9** definiert ist und zum anderen das Signal `SEL`. Mithilfe dieses Signals kann dem Empfänger die Position der Daten innerhalb des gesamten Datensignals übermittelt werden.

Punkt 10 gibt die maximale Größe von zusammenhängenden Daten an, die übertragen werden können. Falls der Wert höher ist als die Datenbreite in Punkt 7, muss unter **Punkt 11** die Reihenfolge der entsprechenden Datenpakete angegeben werden.

Als Abschluss muss unter **Punkt 12** angegeben werden, wenn bestimmte Bedingungen vom Taktsignal (`CLK_I`) erfüllt werden müssen. Zum Beispiel könnte hier eine Taktfrequenz vorgegeben werden.

Literaturverzeichnis

[1] Sally A. McKee Vincent M. Weaver. Code Density Concerns for New Architectures. *2009 IEEE International Conference on Computer Design*, Oct 2009.

[2] Yunsup Lee, Albert Ou, and Albert Magyar. Z-scale: Tiny 32-bit RISC-V Systems. 2015. URL: https://riscv.org/wp-content/uploads/2015/06/riscv-zscale-workshop-june2015.pdf.

[3] STMicroelectronics. *STBus Communication System Concepts and Definitions*, 2007. URL: http://www.st.com/content/ccc/resource/technical/document/user_manual/39/81/fa/c8/2e/4d/41/f5/CD00176920.pdf/files/CD00176920.pdf/jcr:content/translations/en.CD00176920.pdf.

[4] Wade D. Peterson. *Wishbone B4, WISHBONE System-on-Chip (SoC)Interconnection Architecturefor Portable IP Cores*. Silicore Corp., 2010. URL: http://cdn.opencores.org/downloads/wbspec_b4.pdf.

[5] Mohandeep Sharma and Dilip Kumar. CONNECT: Re-examining Conventional Wisdom for Designing Nocs in the Context of FPGAs. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Feb 2012.

[6] Xilinx. *7 Series FPGAs Memory Resources*. Xilinx, ug473 (v1.12) edition, Sep 2016. URL: https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf.

[7] Xilinx. *7 Series FPGAs Configurable Logic Block*. Xilinx, ug474 (v1.8) edition, Sep 2016. URL: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf.

[8] Chien-Hua Chen, Geeng-Wei Lee, Juinn-Dar Huang, and Jing-Yang Jou. A real-time and bandwidth guaranteed arbitration algorithm for SoC bus communication. Jan 2006.

[9] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A. Patterson, and Krste Asanovi. The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9.1. Technical Report UCB/EECS-2016-161, EECS Department, University of California, Berkeley, Nov 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-161.html.

[10] Open Processor Foundation. Dedicated to the advancement of open core processors, 2015. URL: http://0pf.org/.

[11] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovi. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1. Technical Report UCB/EECS-2016-118, EECS Department, University of California, Berkeley, May 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html.

[12] Jeff Dionne Rob Landley. Building a CPU from Scratch: j-core Design Walkthrough. Embedded Linux Conference 2016, Apr 2016. URL: http://j-core.org/talks/ELC-2016.pdf.

[13] Krste Asanovi and David A. Patterson. Instruction Sets Should Be Free: The Case For RISC-V. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley, Aug 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html.

[14] Tony Chen and David A. Patterson. RISC-V Geneology. Technical Report UCB/EECS-2016-6, EECS Department, University of California, Berkeley, Jan 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-6.html.

[15] ARM Ltd. *AMBA Specification Rev 2.0*, 1999. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0011a/index.html.

[16] Mohamed S. Abdelfattah and Vaughn Betz. The Power of Communication: Energy-Efficient NoCs for FPGAs. *IEEE International Conference on Field-Programmable Logic and Applications*, Sep 2013.

[17] Mohamed S. Abdelfattah and Vaughn Betz. Design Tradeoffs for Hard and Soft FPGA-based Networks-on-Chip. *IEEE International Conference on Field-Programmable Technology*, Dec 2012.

[18] Dietmar Moeller. Rechnerstrukturen: Grundlagen der Technischen Informatik. Technical Report 978-364-255-898-6, 2013.

[19] IBM. *The CoreConnect™ Bus Architecture*, 1999. URL: http://www.scarpaz.com/2100-papers/SystemOnChip/ibm_core_connect_whitepaper.pdf.

[20] Andrew S. Tanenbaum. Structured Computer Organization (5th Edition). Technical Report 013-148-521-0, 2005.

[21] Dr. Preeti Bajaj and Dinesh Padole. Arbitration Schemes for Multiprocessor Shared Bus. Technical Report 978-953-307-517-4, 2011.

[22] Jr Philip J. Koopman and Bhargav P. Upender. Time Division Multiple Access Without a Bus Master. Technical Report RR-9500470, United Technologies Research Center, Connecticut, June 1995. URL: https://users.ece.cmu.edu/~koopman/networks/koopman95_tdma.pdf.

[23] Dieter Conrads. Datenkommunikation: Verfahren - Netze - Dienste. Technical Report 978-332-291-973-1, 2013.

[24] Kamil Sh. Zigangirov. Theory of Code Division Multiple Access Communication. Technical Report 978-0-471-65548-0, April 2004.

[25] Sudeep Pasricha Nikil Dutt. On-Chip Communication Architectures. Technical Report 9780123738929, 2008.

[26] ARM Ltd. *AMBA Multi-layer AHB Technical Overview 2.0*, 2004. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dvi0045b/index.html.

[27] ARM Ltd. *AMBA 3 AHB-Lite Protocol v1.0*, 2006. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0033a/index.html.

[28] ARM Ltd. *AMBA AXI Protocol v1.0*, 2004. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022b/index.html.

[29] Altera. *Avalon Interface Specifications*, 2004. URL: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl_avalon_spec_1_3.pdf.

[30] Altera. *Avalon Bus Specifications*, 2003. URL: http://www.ee.ryerson.ca/~courses/coe718/Data-Sheets/sopc/mnl_avalon_bus.pdf.

[31] Altera. Nios II Processor. URL: https://www.altera.com/products/processors/overview.html.

[32] Rudolf Usselmann. OpenCores SoC Bus Review, Jan 2006. URL: https://opencores.org/cdn/downloads/soc_bus_comparison.pdf.

[33] Milica Miti and Mile Stojcev. An Overview of SoC Buses. Dez 2006.

[34] Mohandeep Sharma and Dilip Kumar. Wishbone Bus Architecture - A survey and comparison. *International Journal of VLSI design and Communication Systems*, April 2012.

[35] Arteris. A comparison of Network-on-Chip and Busses, 2005. URL: https://www.design-reuse.com/articles/10496/a-comparison-of-network-on-chip-and-busses.html.

[36] Pico Technology. *PicoScope 2205 MSO Mixed Signal Oscilloscope*, 2016. URL: https://www.picotech.com/download/datasheets/PicoScope2205MSODatasheet-en.pdf.

[37] Altera. *Avalon Interface Specifications*, 2015. URL: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf.

[38] Andreas Traber. RI5CY Core: Datasheet. Technical report, ETH Zürich, Feb 2016. URL: http://www.pulp-platform.org/wp-content/uploads/2016/02/datasheet_RI5CY.pdf.

[39] VectorBlox. ORCA FPGA-Optimized RISC-V. http://riscv.org/wp-content/uploads/2016/01/Wed1200-2016-01-05-VectorBlox-ORCA-RISC-V-DEMO.pdf, 2016.

[40] Andreas Traber, Florian Zaruba, Sven Stucki, Antonio Pullini, Germain Haugou, Eric Flamand, Frank K. Gürkayank, and Luca Benini. PULPino: A small single-core RISC-V SoC. http://iis-projects.ee.ethz.ch/images/d/d0/Pulpino_poster_riscv2015.pdf, 2015.