FH Dortmund

- Fachbereich für Informationstechnik -

Fachhochschule
Dortmund
University of Applied Sciences and Arts

Masterarbeit

Integration eines Hardwarebeschleunigers für Maschinelles Lernen in einen RISC-V RV32IM Prozessor über Memory-Mapped Register

Fabian Brünger

betreut von

Prof. Dr. Michael Karagounis
M. Sc. Alexander Walsemann

24. August 2020

Danksagung

Mein Dank für die Unterstützung bei dieser Arbeit gilt Prof. Dr. Michael Karagounis, der nicht nur mit fachlicher, sondern auch mit zwischenmenschlicher Kompetenz immer für eine positive und produktive Arbeitsatmosphäre gesorgt hat. Auch der Anspruch nach qualitativ hochwertiger Arbeit, sowie die Analyse von den vermeidlich kleinen Details waren zum Ende meines Studiums Lektionen, die ich zu schätzen weiß. Weiterer Dank geht an Alexander Walsemann. Gerade zu Beginn meiner Laborzeit waren seine Kompetenzen unersetzlich.
Für die Unterstützung im privaten Umfeld danke ich meiner Partnerin Cristina Rosado-Alberdi, die mir in den schweren Stunden zur Seite stand und steht. Zuletzt möchte ich meinen Eltern danken. Ohne sie wäre mein Studium und mein Lebensweg in dieser Form nicht möglich gewesen. Für die Toleranz und Geduld mir gegenüber bin ich unglaublich dankbar.

„*Chicken in the corn*

Say the corn can't grow, mama“

Brushy One String

Kurzreferat

Im Rahmen dieser Arbeit wurde eine Analyse auf Register Transfer Level (RTL) Ebene des vom Fraunhofer IMS in Verilog entwickelten RV32IM RISC-V Prozessors durchgeführt und der Configurable Accelerator Engine for Convolution Operations (Caeco) als Hardware-Beschleuniger für Maschinelles Lernen (ML) integriert. Das Design wurde speziell auf das Lesen von Caecodaten und auf das Interrupt-Verhalten getestet und verifiziert. Das Schreiben von Caecodaten wurde zwar auf RTL Ebene simuliert, allerdings nicht auf dem Field Programmable Gate Arrays (FPGA) verifiziert. Durch einen erarbeiteten Hardware- und Software-Entwicklungsfluss werden beide Stränge optimiert und parallelisiert. Die Hardware-Entwicklung wurde in eine Gitlab Development and Operations (DevOps) Umgebung integriert, wodurch das Design im Project Batch Flow Modus der Vivado 2020.1 IDE automatisiert simuliert, synthetisiert und auf der Entwicklungsplatine Nexys4 DDR implementiert wird. Die Verifizierungsgrundlage bildet der entwickelte Programm-Code, der für die RTL Simulation, für die Simulation im Instruktionssimulator riscvOVPsim der Firma Imperas und dem Debugging des Designs auf dem FPGA genutzt wird. Letzteres wurde in der Eclipse IDE durchgeführt, wobei der JTAG Olimex ARM-USB-Tiny-H Adapter als Debug-Schnittstelle eingesetzt worden ist. Die Schnittstelle der beiden Entwicklungsstränge bilden zwei eigens geschriebene Rust Programme und das Xilinx Programm data2mem, durch die die kompilierten ELF Dateien in xilinx-kompatible MEM bzw. COE Dateien umgewandelt werden.

Stichworte: RISC-V RV32IM, FPGA, Vivado 2020.1, Gitlab CI/CD, RISC-V Software-Entwicklung

Abstract

Within the scope of this thesis, an analysis on RTL level of the RV32IM RISC-V processor developed by Fraunhofer IMS in Verilog was performed and the caeco was integrated as hardware accelerator for ML. The design has been tested and verified especially for the reading of caecodata and for the interrupt behaviour. The writing of caecodata was simulated on RTL level, but not verified on the FPGA. Both strands are optimized and parallelized by an elaborated hardware and software development flow. The hardware development was integrated into a Gitlab DevOps environment, whereby the design is automatically simulated, synthesized and implemented on the development board Nexys4 DDR in the Project Batch Flow mode of the Vivado 2020.1 IDE. The basis for verification is the developed program code, which is used for RTL simulation, for simulation in the instructional simulator riscvOVPsim from Imperas and for debugging the design on the FPGA. The latter was done in the Eclipse IDE, using the JTAG Olimex ARM-USB-Tiny-H Adapter as debug interface. The interface between the two lines of development are two specially written Rust programs and the Xilinx program data2mem, which converts the compiled ELF files into xilinx-compatible MEM or COE files.

Keywords: RISC-V RV32IM, FPGA, Vivado 2020.1, Gitlab CI/CD, RISC-V Softwaredevelopment

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Dortmund, den 24. August 2020

Inhaltsverzeichnis

Abkürzungsverzeichnis

MPU	Micro Processor Unit
CPU	Central Processing Unit
IC	Integrated Circuit
SoC	System-on-a-Chip
GPP	General Purpose Processor
DSP	Digital Signal Processor
MAC	Multiply-Accumulate
PC	Personal Computer
EEPROM	Electrically Erasable Programmable Read-Only Memory
CU	Control Unit
ISA	Instruction Set Architecture
RISC	Reduced Instruction Set Architecture
CISC	Complex Instruction Set Computers
MCU	Micro Controller Unit
CPU	Central Processing Unit
ALU	Arithmetic Logic Unit
ISA	Instruction Set Architecture
HDL	Hardware Description Language
GP	General Purpose
CSR	Control and Status Register
AHB	Advanced High-Performance Bus
AMBA	Advanced Microcontroller Bus Architecture

HASTI	Highly Advanced System Transport Interface
UART	Universal Asynchronous Receiver Transmitter
JTAG	Joint Test Action Group
DM	Debug Module
DMI	Debug Module Interface
TAP	Test Access Port
DTM	Debug Transport Module
IR	Instruktionsregister
DR	Datenregister
MSB	Most Significant Bit
LSB	Least Significant Bit
IP	Intellectual Property
RAM	Random Access Memory
FPGA	Field Programmable Gate Arrays
IF	Instruction Fetch
DX	Decode and Execution
WB	Write Back
ELF	Executable and Linkable Format
LTU	LookUp Table
DUT	Device under test
BSS	Basic Service Set
DevOps	Development and Operations
CI	Continuous Integration
CD	Continuous Delivery
ML	Maschinelles Lernen
IP	Intellectual Property
WSL	Windows Subsystem for Linux

RTL	Register Transfer Level
CFI	Call Frame Information
COE	Coefficient
MEM	Memory
ISR	Interrupt Service Routine
DUT	Device Under Test
UTF	UCS Transformation Format
ASIC	Application Specific Integrated Circuit
KI	Künstlichen Intelligenz
UI	User Interface
GPIO	General Purpose Input/Output
IDE	Integrated Development Environment
Caeco	Configurable Accelerator Engine for Convolution Operations

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einleitung

Der Entwurf von anwendungsspezifischen integrierten Schaltungen (engl. *Application Specific Integrated Circuits (ASICs)*) für Anwendungen der Künstlichen Intelligenz (KI) und des ML verflogt einerseits das Ziel, hohe Klassifizierungsgenauigkeiten durch aufwändige numerische Rechenverfahren zu erreichen und anderseits den Energieeffizienzanforderungen von eingebetteten Systemen gerecht zu werden [1]. Im Rahmen dieser Arbeit wird der Caeco als ML-Intellectual Property (IP) Kern für die Auswertung von EKG Daten im Bereich der Medizintechnik in den Raifes RISC-V RV32IM Prozessor integriert, um dadurch ein flexibles System zu entwerfen, das neben spezifischen Anforderungen aus dem ML Bereich auch herkömmlichen Software-Anforderungen nachkommen kann. Dabei geht es vor allem um die Aufgabe, Daten für den Caeco parallel zum generellen Prozessorbetrieb über eine Joint Test Action Group (JTAG) Schnittstelle zu schreiben und bei vorliegen eines Ergebnisses die Daten zu speichern bzw. per Universal Asynchronous Receiver Transmitter (UART) Schnittstelle bereitzustellen. Hierfür wird die vom Fraunhofer IMS entwickelte Raifes RV32IM Plattform auf RTL Ebene analysiert und um caeco-spezifischen Funktionen erweitert. Das erstellte Design wird auf RTL Ebene simuliert und auf der Nexys4 DDR FPGA Entwicklungsplatine implementiert und getestet. Die gesamte Hardware-Entwicklung wird durch die Anwendung der Gitlab DevOps Umgebung automatisiert.
Zusätzlich zur Hardware-Entwicklung wird in dieser Arbeit ebenfalls die passende Software in der Programmiersprache C und dem RISC-V Assembler entwickelt. Außerdem werden weitere Dateien, die zur Programmierung eingebetteter Systeme notwendig sind, analysiert und angepasst. Die Ziel-Software dient zum einen als Grundlage für die RTL Simulation, zum anderen wird sie durch den RISC-V Instruktionssimulator riscvOVPsim der Firma Imperas getestet, um so den Vergleich zu einem Referenzmodell zu ermöglichen. Als letzter Schritt wird das implementierte Design auf dem FPGA in Betrieb genommen, getestet und durch die Entwicklungsumgebung Eclipse IDE debuggt. Für das Debugging wird das Design so angepasst, dass das Interrupt-Verhalten des Caecos durch einen manuellen Schalter der Entwicklungsplatine nachgestellt wird, da zum aktuellen Zeitpunkt das Schreiben der EKG Daten noch nicht realisiert werden kann.
Die Hard- und Software-Entwicklung des Zielsystems muss parallel stattfinden. Zum einen gibt es Hardware-Voraussetzungen, die bei der Software-Entwicklung zu beachten sind, zum anderen kann die Hardware nur mit einem lauffähigen Programm getestet werden. Eine Prozessoptimierung wird durch zwei zusätzliche eigens geschriebene Hilfsprogramme erzielt, die im Zusammenspiel mit dem Xilinxtool data2mem die kompilierten Executable and Linkable Format (ELF) Dateien in Coefficient (COE) Dateien umwandeln, durch die xilinx-spezifischen Speicher initialisiert werden können.

2 RISC und RISC-V Prozessorgrundlagen

Dieses Kapitel bietet eine Übersicht auf die Reduced Instruction Set Architecture (RISC) Prozessorgrundlagen, ausgewählte RISC-V Spezifikationen und den JTAG Standard. In Abschnitt 2.1 wird in die Geschichte und Aufbau der RISC Prozessoren eingeführt. Ein Überblick über Teile der RISC-V, insbesondere der Debug-Spezifikation, ist in Abschnitt 2.2 gegeben. Abschließend wird in Abschnitt 2.3 der JTAG Standard beschrieben.

2.1 RISC Prozessoren

RISC Prozessoren sind das Produkt einer Entwicklung, die Ende der 70er Jahre im IBM-Forschungszentrum in Yorktown Heights mit dem IBM-801-Projekt ihren Anfang nahm. Anfang der 80er folgten das MIPS-Projekt der Stanford University und das RISC-Projekt der UC Berkeley. Das Ziel war bei allen Projekten die Entwicklung einer Prozessorarchitektur, die durch ein reduzierten und vereinfachten Befehlssatz und somit ein kleineres Steuerwerk, die Rechenleistung steigert und gleichzeitig den Entwicklungs- und Produktionspreis senkt [2]. RISC Architekturen sind ein Gegenentwurf zu den bis zu diesem Zeitpunkt üblichen Complex Instruction Set Computers (CISC) Prozessoren, die eine Fülle von Befehlen unterschiedlicher Komplexität bieten, jedoch durch die Mikroprogrammierung algorithmisch abgearbeitet werden, welches ein umfangreiches Steuerwerk voraussetzt. Bei RISC Prozessoren gibt es im Gegenteil nur wenige Maschinenbefehle und dadurch eine Hardware-Einheit mit reduzierter Komplexität, die Befehle direkt umsetzen kann. Somit entfallen komplexe Maschinenbefehle, die bei CISC Prozessoren vorgesehen sind. [3].
Die Effektivitätssteigerung der RISC Prozessoren wird größtenteils durch die Nutzung des Pipelining-Prinzips erzielt. Durch die Nutzung von Pipeline-Architekturen ist es möglich pro Prozessortakt einen Maschinenbefehl auszuführen. Ist ein Prozessor dazu befähigt, spricht man von einem skalaren Prozessor. Kann der Prozessor mehr als einen Maschinenbefehl pro Prozessortakt abarbeiten, ist von einem superskalaren Prozessor die Rede. Zusätzlich zu der fehlenden Mikroprogrammierung und der angestrebten Skalarität sind unter anderem dedizierte Lade- und Speicherbefehl für den Hauptspeicherzugriff vorgesehen, wodurch der aufwändige Datentransport zwischen Speicher und Prozessor minimiert wird. Durch eine höhere Anzahl prozessornaher Speicher, sogenannter Register, werden mehr Daten schneller zugänglich. Dadurch minimiert sich ebenfalls die Anzahl der Zugriffe auf den Datenhauptspeicher. [3]. RISC Prozessoren nutzen üblicherweise einen Zwischenspeicher für Programmdaten, der sich sehr nahe am Prozessor befindet und separat vom Datenhauptspeicher gehalten wird. Aus diesem Grund werden RISC Prozessoren meist als Harvard-Architektur

klassifiziert. Von-Neumann-Architekturen hingegen verwenden einen gemeinsamen Speicher sowohl für das Programm als auf für anderweitige Daten, welche auch über einen gemeinsamen Bus zum Prozessor transportiert werden [4]. Der schematische Aufbau eines RISC Prozessors mit einer Harvard-Architektur wird in 2.1.1 beschrieben beschrieben. In verschiedenen aktuellen Prozessoren verschwimmen die klaren Grenzen der Prozessor-Architekturen mehr und mehr, sodass man häufig von Hybridsystemen spricht, die vor allem im Bereich der Personal Computer (PC) vorzufinden sind.

2.1.1 Mikroprozessoraufbau

Eine Micro Processor Unit (MPU) ist die zentrale Rechnereinheit (engl. *Central Processing Unit (CPU)*) in einem Rechnersystem, die Maschinenbefehle ausführt. Die MPU enthält verschiedene Komponenten, die über Busse Daten und Steuerungsbefehle austauschen. Die Hauptkomponenten sind das Rechenwerk, welches ebenfalls als Operationswerk (engl. *Arithmetic Logic Unit (ALU)*) bezeichnet wird, und das Steuerwerk. Diese Komponenten fasst man meist als den Prozessorkern zusammen [2]. Weiterhin gibt es einen festen Registersatz als schnellen Zwischenspeicher für Operationen des Rechenwerks und ein Adresswerk zur Adressierung von Daten im Speicher. Ein Mikrorechner in Harvard-Architektur wird durch einen Datenspeicher, einen Programmspeicher und Schnittstellen zur externen Peripherie vervollständigt.

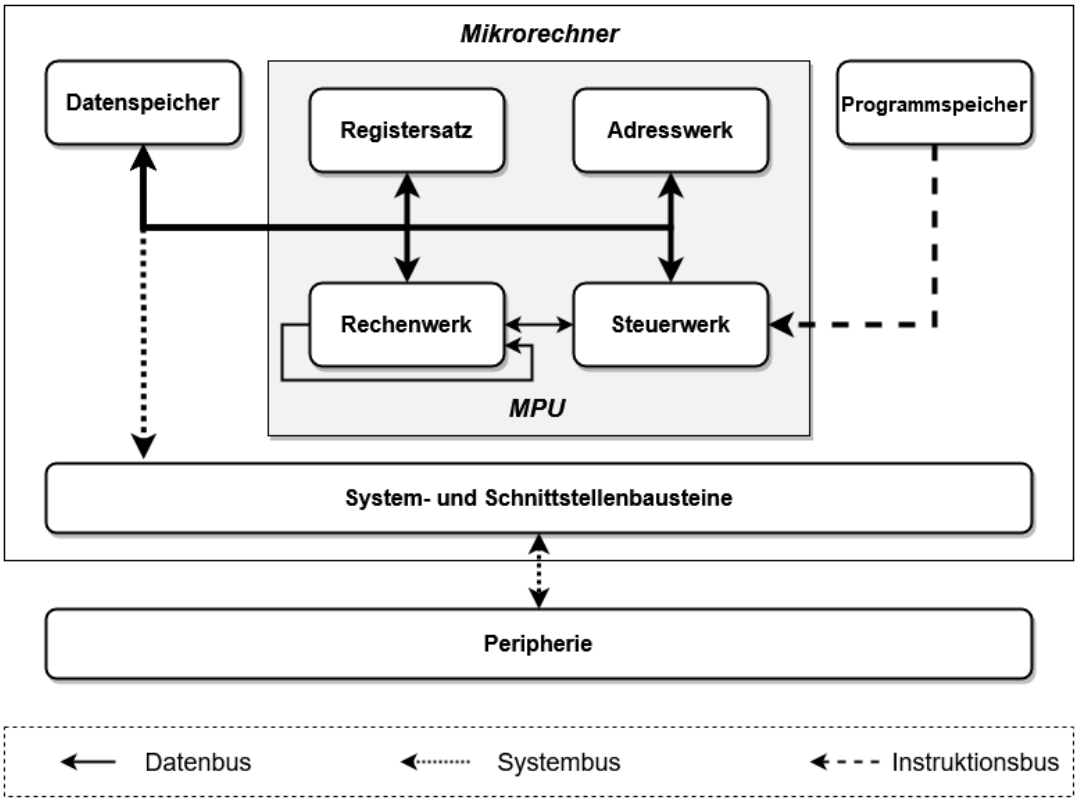

Abbildung 2.1: Mikroprozessoraufbau und Mikrorechneraufbau mit Harvardarchitektur (vgl. S. 89 [3] und S. 2 [4])

In Abbildung 2.1 ist dieser Aufbau dargestellt. Moderne Prozessoren verwenden oft einen oder mehrere Zwischenspeicher, sogenannte Caches. Diese sind kleiner als der Datenhauptspeicher, wodurch es möglich ist, sie auf dem gleichen Chip zusammen mit dem Prozessorkern zu integrieren. Auf Grund der örtlichen Nähe sind die Daten in den Caches schneller abrufbar, idealerweise in ein bis zwei Taktzyklen. Verfügt der Chip über mehrere Caches, sind diese in einer Speicherhierarchie angeordnet. Der prozessornähste Cache ist der First-Level-Cache, wobei die Anzahl der verschiedenen Stufen variieren kann. In sogenannten Pseudo-Harvard-Architekturen gibt es zwei First-Level-Caches, einen für die Speicherung der Maschinenbefehle und einen weiteren für Daten [5].
Die Aufgabe der MPU ist die Ausführung des Maschinen-Codes, welcher aus einer Aneinanderreihung verschiedener Maschinenbefehle besteht, die das eigentliche Programm darstellen. Das Programm wird von Programmiererinnen und Programmierern entwickelt und vor der Ausführung in den Programmspeicher geladen. Die Ausführung der Maschinenbefehle folgt einem Automatismus, der zuerst die gespeicherten Maschinenbefehle ließt (engl. *Fetch*) und danach dekodiert. Ein Maschinenbefehl besteht mindestens aus einem OpCode und einem oder mehreren Operanden. Der OpCode definiert die Art des auszuführenden Befehls. Operanden werden entweder aus den Registern ausgelesen oder im Falle von Konstanten dem Maschinenbefehl entnommen. Welcher Maschinenbefehl als nächstes gelesen werden soll, wird durch den Programmzähler bestimmt. Dieser startet bei einer Basisadresse und wird nach einem Lesebefehl inkrementiert. Bei einem Sprungbefehl wird der Programmzähler nicht auf die folgende Adresse inkrementiert, sondern zu einer angegebenen Adresse geändert [3].

Steuerwerk:

Das Steuerwerk (engl. *Control Unit (CU)*) ist für die Dekodierung der Maschinenbefehle und für die Steuerung sämtlicher Komponenten zuständig. Bei den RISC Prozessoren werden die Steuerwerke durch explizite kombinatorische Schaltlogik implementiert. Man spricht demnach auch von einem "fest verdrahtetem"(S. 2 [4]) Steuerwerk, welches ohne algorithmischen Ansatz genau auf einzelne Befehle reagiert. Somit minimiert man die Dauer eines Befehlszyklus, welcher der Zeit zur Abarbeitung eines Maschinenbefehls entspricht [3].

Rechenwerk:

Die zentrale Einheit des Rechenwerks ist die ALU. Die ALU führt die von dem Steuerwerk dekodierten Befehle als arithmetische und logische Operationen durch. Das Ergebnis einer Operation mag Einfluss auf die Ausführung der nachfolgenden Instruktionen haben. Dafür werden die Ergebnisse in verschieden Zustände klassifiziert und in Statusregistern bzw. Zustandsregistern gespeichert [3].

Programmspeicher:

Der Programmspeicher ist das Hauptmerkmal einer Harvard-Architektur und enthält den eigentlichen Programm-Code. Durch die Existenz eines separaten Programmspeichers kann gleichzeitig sowohl auf die Programmdaten als auch auf den Datenspeicher zugegriffen werden. Dieser Speicher ist oft nicht flüchtig, sodass die gespeicherte Programmierung auch nach dem Ausfall der Versorgungsspannung erhalten bleibt. Speichertechnologien, die in diesem Zusammenhang oft verwendet werden, sind die Folgende: elektrisch löschbarer programmierbarer Nur-Lese-Speicher (engl. *Electrically Erasable Programmable Read-Only Memory (EEPROM)*), oder auch als einmal programmierbare EPROMs (engl. *One Time Programmable Erasable Programmable Read-Only Memory (OTP-EPROM)*), oder auch als Flash-EEPROM [3].

Adresswerk:

Das Adresswerk übernimmt das Ansprechen der verschiedenen Speicherbereiche wie den Registern und dem Hauptdatenspeicher. Oft werden Speicheradressen als Kombination einer Basisadresse und eines Offsets, um welchen die Basisadresse erhöht wird, angegeben. Diese adressbezogenen Berechnungen werden meist von einem zusätzlichen Adressrechner durchgeführt, um das Rechenwerk zu entlasten [3].

Registersatz:

Die Register sind Speicherplätze, die im inneren des Prozessors liegen, und somit schnell zu erreichen sind. Sie können grundsätzlich in Universalregister und Spezialregister unterschiedenen werden. Die Universalregister dienen der freien Verfügung für verschiedene Inhalte und Maschinenbefehle. Die Spezialregister sind unter anderem der Programmzähler, der Stackpointer, das Basisregister, das Indexregister, das Statusregister und das Steuerregister. Dabei ist das Steuerregister dafür zuständig, Unterbrechung des Programms zu verwalten [3].

Zusammengefasst organisiert der Mikroprozessor den Datenaustausch in einer Mikrorechnerstruktur. Im Datenspeicher liegen alle anderen Daten, die nicht mit der eigentlichen Prozessorprogrammierung korrelieren. Über System- und Schnittstellenbausteine kann mit externer Peripherie kommuniziert werden. Wie diese Komponenten eines Prozessors genau spezifiziert sind und wie genau diese zusammenarbeiten, wird durch die Befehlssatzarchitektur definiert. In Kapitel 2.1.2 wird das Grundkonzept einer RISC Befehlssatzarchitektur beschrieben. Prozessoren können zwar grob als ein RISC Prozessor klassifiziert werden, allerdings gibt es je nach Hersteller und Entwickler unterschiedliche architektonische Ansätze, sowohl in Bezug auf die Hardware, als auch für die Organisation des Befehlssatzes.

2.1.2 Befehlssatzarchitekturen

Eine Befehlssatzarchitektur (engl. *Instruction Set Architecture (ISA)*) definiert den Befehlssatz und die Befehlsformate einer MPU. Sie ist die Schnittstelle zwischen der Software und der Hardware, da durch Sie genau definiert wird, auf welcher Weise ein bestimmter Befehlssatz vom Prozessor ausgeführt wird. Obwohl gewisse Grundfunktionen Bestandteil eines jeden Prozessors darstellen, wie zum Beispiel arithmetische Operationen, gibt es jedoch von verschiedenen Herstellern unterschiedliche ISA Ansätze. Die aktuell kommerziell erfolgreichste RISC Architektur stammt von der Firma ARM Limited. Es werden jedes Jahr über 30 Milliarden Chips mit einer ARM Architektur produziert. Dabei ist ARM selbst kein Chipproduzent, sondern bietet Chipproduzenten die Porzessorarchitektur samt fertiger Modelle als Lizenz an [3].
Der Befehlssatz ist einer der Hauptmerkmale, durch den eine Architektur charakterisiert wird. Dabei geht es um die genaue Beschreibung, wie Assamblerbefehle in Maschinensprache übersetzt werden. Grundbefehle sind: Transport-, Arithmetische-, Bitweise Logische-, Schiebe-, Rotations-, Einzelbit-, Sprung- und Prozessorsteuerungsbefehle. Bei dem Befehlsformat wird ebenfalls strikt festgelegt, wie OpCodes aufgebaut sein müssen, damit der Maschinenbefehl dekodiert werden kann. Dabei wird die Bitbreite festgelegt, die ein Befehl besitzen muss. Außerdem wird jedem einzelnen Bit in einem vorgegebenen Bitmuster eine Bedeutung zugesprochen, die bei jeder Decodierung in gleicher Weise interpretiert wird.

2.2 RISC-V

Die 2015 gegründete RISC-V Foundation definiert vier verschiedene Befehlssätze, die frei und kostenlos zu Verfügung stehen. Der Ursprung der Initiative ist dem Electrical Engineering and Computer Science Departement an der Universität von California in Berkeley entsprungen. Das Ziel ist es, durch eine offene Kollaboration im Rahmen eines Open-Source Ansatzes die Entwicklung der ISAs voranzutreiben und sowohl für die Wissenschaft als auch für die Industrie frei zugänglich zu machen. Momentan gibt es verschiedene Mitglieder der RISC-V Foundation, die bereits Prozessorkerne und SoCs entwickelt haben und diese sowohl als Hardware als auch als IP Core in verschiedenen Hardware Description Languages (HDLs) zu freien Verfügung stellen [6]. Unterabschnitt 2.2.1 gibt eine Einführung in die RISC-V ISA Thematik und stellt die grundlegenden Spezifikationen der zwei Basis ISAs und deren Erweiterungen kompakt dar. In Unterabschnitt 2.2.2 wird gezielt auf die RV32I Basis ISA sowie auf die Erweiterung für die Multiplikations- und Divisionsbefehle eingegangen. Abschließend wird in Unterabschnitt 2.2.3 die Debug Spezifikation erläutert.

2.2.1 Befehlssätze

Die vier RISC-V Basis ISAs werden durch die folgenden Merkmale charakterisiert: Die Anzahl und die Breite der Integer Register und die Größe des Adressspeichers. Die zwei Hauptarchitekturen sind

die RV32I und die RV64I. Dabei stehen die Buchstaben “RV“ als Kürzel in der Namensgebung für die RISC-V Architektur. Die 32 beziehungsweise 64 steht für die Größe der Register, welche demnach 32 bit bzw. 64 bit entspricht. Das “I“ steht für Integer und bezeichnet die Basis Architektur, auf welche die anderen ISAs aufbauen. Diese wird in Unterabschnitt 2.2.2 ausführlich beschrieben.

Tabelle 2.1: Übersicht der Befehlserweiterungen

Kürzel	**Bestimmung**
I	Integer Instructions: Basisbefehlssatz für Integer Operationen
Zifencei	Instruction-Fetch Fence: Verwaltung und Synchronisation von Datenzugriff und Befehlszugriff
M	Integer Multiplication and Division: Befehle für die Multiplikation und Division von zwei Integern in 2 Registern
A	Atomic Instructions: Atomare Befehle für die Synchronisation bei Multithreading
Zicsr	Control and Status Register: Seperater Adressraum für Control und Status Register
F	Single-Precision Floating-Point: Standard Befehlssatz für die Einfache-Genauigkeit von Gleitkomma Operationen; nach dem IEEE-754-2008 Standard; benötigt Zicsr Erweiterungen
D	Double-Precision Floating-Point: Standard Befehlssatz für die doppelte Genauigkeit von Gleitkomma Operationen; nach dem IEEE-754-2008 Standard; benötigt F Erweiterungen
Q	Quad-Precision Floating-Point: Standard Befehlssatz für die vierfache Genauigkeit von Gleitkomma Operationen; nach dem IEEE-754-2008 Standard; benötigt D Erweiterungen
L	Decimal Floating-Point: Standard Befehlssatz für dezimale Gleitkomma Operationen; nach dem IEEE-754-2008 Standard
C	Compressed Instructions: Erlaubt die Verwendung von 16-bit Befehlen
B	Bit Manipulation: Momentaner Platzhalter für Befehle, welche die Manipulation einzelner Bits durchführen
L	Dynamic Translated Languages: Momentaner Platzhalter für die Unterstützung von dynamischen Programmiersprachen
T	Transactional Memory: Momentaner Platzhalter für die Unterstützung von transaktionalen Speichern
P	Packed-SIMD Instructions: Momentaner Platzhalter für Befehle, die bei Aufruf die gleiche Operation auf viele unterschiedliche Operanden ausführen. (Single Instruction Multiple Data)
V	Vector Operations: Momentaner Platzhalter für die Unterstützung von Vektorbefehlen
N	User-Level Interrupts: Interrupt Handhabung auf User-Level Ebene
Zam	Misaligned Atomics: Unterstützung von falsch gerichteten atomaren Befehlen; benötigt A Erweiterung
Ztso	Total Store Ordering: Erweiterung zum Speichermanagement

Die Basis ISAs verwenden das Zweierkomplement für die Darstellung negativer Integer Zahlen. Grundsätzlich sind die RISC-V Basis ISAs vergleichbar mit denen herkömmlicher RISC Prozessoren,

wobei es zwei wesentliche Unterschiede zu diesen Architekturen gibt. Zum einen sind keine Branch Delay Slots (*Warteplatz*) vorhanden. Zum anderen wird in den RISC-V ISAs die Decodierung von Befehlen unterschiedlicher Bitlänge zugelassen. Die Basis ISAs und deren Befehlssätze sind drauf ausgelegt je nach Anwendungszweck erweitert zu werden. Eine Übersicht der verschiedenen bereits definierten Erweiterungen des RISC-V Basisbefehlssatzes ist in Tabelle 2.1 gegeben. Dabei sind die Standardbefehle von der RISC-V Foundation mit der Zusage definiert worden, dass diese unter allen Umständen unverändert bleiben. Darüber hinaus gibt es Befehle, die von der Foundation reserviert worden sind, allerdings nicht zu dem Standardbefehlssatz gehören. Diese reservierten Bitkombinationen können Nutzer bei Bedarf für die Implementierung individueller Befehle verwenden. Die RV32I stellt die grundlegende Befehlssatzarchitektur dar. Darüber hinaus sind jedoch weitere Zusatzbefehlsformate durch RISC-V definiert, die durch Kürzel gekennzeichnet sind und als Erweiterung der Bezeichnung der RV32I angehängt werden. Ein Beispiel stellt der Befehlssatz RV32IM dar. Das “M“ steht in diesem Fall für die Befehlssatzerweiterung, welche Befehle für die Multiplikation und Division definiert.

2.2.2 Basisbefehlssatz RV32I

In diesem Abschnitt wird der Basisbefehlssatz RV32I und die Befehlssatzerweiterung M beschrieben. Zu beginn wird der Registersatz erläutert und darauf folgend die definierten Befehle und ihr Format beschrieben.

Allzweckregister

Die Register des RV32I Basisbefehlssatz besitzen eine 32-Bit Breite. Es existieren 32 Register, die durch Zahl x beziffert sind.

Tabelle 2.2: Übersicht der Register (siehe [6], S. 137)

Reg	Verwendung	Reg	Verwendung	Reg	Verwendung
x0	Zero	x11	fun. arguments/return values	x22	saved reg
x1	return addr.	x12	fun. arguments	x23	saved reg
x2	stack pointer	x13	fun. arguments	x24	saved reg
x3	global pointer	x14	fun. arguments	x25	saved reg
x4	thread pointer	x15	fun. arguments	x26	saved reg
x5	alternate link	x16	fun. arguments	x27	saved reg
x6	temporaries	x17	fun. arguments	x28	temporaries
x7	temporaries	x18	saved reg	x29	temporaries
x8	saved reg/frame pointer	x19	saved reg	x30	temporaries
x9	saved reg	x20	saved registers	x31	temporaries
x10	fun. arguments/return values	x21	saved reg	pc	temporaries

Zusätzlich ist der Programmzähler als Spezialregister definiert. In Tabelle 2.2 sind die Register und deren bevorzugte Verwendung aufgelistet. Von den 33 Registern ist nur das Register x0 und der programm counter (*Programmzähler*) mit einer festen Funktion versehen. Die restlichen Register können als Allzweckregister genutzt werden. Das Register x0 hat den festen Wert Null, da alle Bits zu jeder Zeit auf Null gesetzt sind. Als Konvention für eine bessere Software-Kompatibilität haben sich die in Tabelle 2.2 angegeben Funktionen für alle weiteren Register ergeben.

Befehlsformate

Der Basisbefehlssatz definiert sechs verschiedene Befehlsformate, die eine feste Kodierung für alle 32 Bits festlegen. Es werden Befehlsformate vom Typ R-, I-, S-, B-, U- und J unterschieden. Dazu kommen Speicherorganisations- und System-Befehle. Im Folgenden werden die Formate und deren Kodierung beschrieben. Dabei werden einzelne Bits bzw. Bit-Vektoren mit inst[Bit bzw. Bit:Bit] gekennzeichnet. Obwohl die sechs Formate verschieden kodiert werden, besitzen alle Formate einen OpCode. Der OpCode befindet sich immer in den niederwertigsten sieben Bits und beinhaltet die Information, welche Operation ausgeführt werden soll. Dabei sind die zwei niederwertigsten Bits des Basisbefehlssatzes immer auf den Wert eins bzw. inst[1:0] = 11 gesetzt. Die restlichen fünf Bits entsprechen jeweils einer bestimmten Operation. So entspricht beispielsweise der Lade-Befehl LOAD dem OpCode inst[6:0]=0000011.

Im R-Typ Format werden Operationen und Berechnungen definiert, welche zwei Register als Quelle für Operanden verwenden. In Abbildung 2.2 ist das Befehlsformat R-Typ dargestellt.

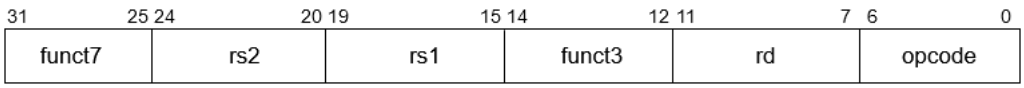

Abbildung 2.2: R-Type Befehlsformat

Tabelle 2.3: R-Typ Befehlsübersicht

Mnemonik	**Bedeutung**	**Operation**	**funct3**	**funct7**
ADD	Addition	rd:=$rs1 + rs2$	000	0000000
SLT	Signed Compare	rd:=1, if $rs1 < rs2$	010	0000000
SLTU	Unsigned Compare	rd:=1, if $rs1 < rs2$	011	0000000
AND	Bitwise Logical	rd:=$rs1 \& rs2$	111	0000000
OR	Bitwise Logical	rd:=$rs1 \mid rs2$	110	0000000
XOR	Bitwise Logical	rd:=$rs1$ xor $rs2$	100	0000000
SLL	Logical Left Shift	rd:=$rs1 <<$(value of)$rs2$	001	0000000
SRL	Logical Right Shift	rd:=$rs1 >>$(value of)$rs2$	101	0000000
SUB	Subtraction	rd:=$rs1 - rs2$	000	0100000
SRA	Arithmetic Right Shift	rd:=signed $rs1 >>$(value of)$rs2$	101	0100000

Der OpCode für R-Typ Befehle ist auf den Wert OP=0110011 festgelegt. Daneben besitzen die Bitfelder funct7=[31:25] und funct3=[14:12] die Aufgabe verschiedene Operationen zu definieren. Die Register, aus denen die Operanden entnommen werden, sind in den Bitfeldern rs2=[24:20] und rs1=[19:15] festgelegt. Das Zielregister wird im Bitfeld rd=[11:7] festgelegt. Tabelle 2.3 gibt eine Übersicht auf R-Typ Befehle in Korrelation zu den entsprechenden mnemonischen Codes. Bei diesen Operationen wird ein Overflow (*Überlauf*) nicht beachtet. Bei der Addition und der Subtraktion werden die untersten 32 bits des Ergebnisses in das Zielregister geschrieben. Eine Besonderheit ist die Vergleichsoperation SLTU, die ohne Beachtung des Vorzeichens durchgeführt wird. Wenn Mithilfe dieser Operation das Nullregister x0 mit einem zweiten Register rs2 verglichen wird, wird eine logische Eins in das Zielregister geschrieben, wenn rs2 ungleich Null ist. Ist andererseits rs2 gleich Null, dann wird das Zielregister ebenfalls auf Null gesetzt.

Der I-Typ definiert Integer Operationen zwischen einer Konstanten und einem Register rs1. Dabei umfasst die Größe der Konstanten zwölf Bits inst[31:20]. Die Operationen sind den R-Typ Operationen sehr ähnlich, mit der einzigen Ausnahme, dass bei den I-Typ Operationen statt auf ein zweites Register, auf eine Konstante als Operand zurück gegriffen wird. In Abbildung 2.3 ist das I-Typ Befehlsformat ohne Verschiebungsoperationen dargestellt. Für Schiebeoperationen wird ein eigenes Befehlsformat verwendet, welches starke Ähnlichkeit zu den R-Typ Operationen besitzt (siehe Abbildung 2.4).

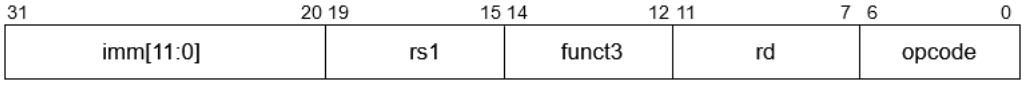

Abbildung 2.3: I-Typ Befehlsformat für Integer Berechnungen aus Register und Konstante

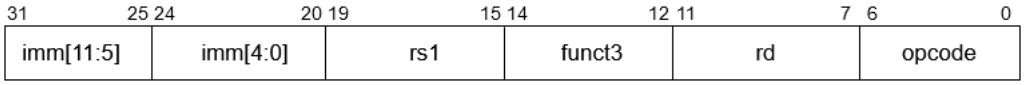

Abbildung 2.4: I-Typ Befehlsformat für Shift-Operationen aus Register und Konstante

Der OpCode für I-Typ Operationen ist auf den Wert 0010011 festgelegt. Die Bitfelder rs1, funct3, rd und OpCode besitzen dieselbe Funktion wie bei R-Typ Operationen. Die Konstante umfasst zwölf Bits imm[11:0]. Bei Shift-Operationen werden die niederwertigsten fünf Bits imm[4:0] für die Anzahl der Verschiebungen genutzt. Während die niederwertigsten sieben Bits imm[11:5] einem Bitmuster entsprechen, welches für die jeweilige Operation fest definiert ist. Die Operation ADDI ignoriert analog zur ADD Operation den Überlauf einer Berechnung zwischen der Konstanten und dem Registerwert rs1. Dabei werden in gleicher Weise nur die 32 niederwertigsten Bits des Resultats im Zielregister rd abgespeichert. Eine Besonderheit dieses Befehls besteht darin, dass der Befehl der Datenverschiebung (Mnemonisch: MV) von einem Register in ein anderes eine virtuelle Instruktion darstellt, welche durch den Befehl ADDI rd, rs implementiert wird. Die Nulloperation

(Mnemonisch: NOP) stellt eine weitere virtuelle Instruktion dar, welche ebenfalls mit Hilfe des ADDI Befehls in der Form ADDI x0, x0, 0 implementiert wird. Eine weitere Besonderheit ist im Zusammenhang mit Logik-Operationen zu finden. Die Inversion eines Registers ist als virtueller Instruktion NOT möglich, welche dem Befehl XORI rd, rs1, -1 entspricht. Befehle für das Laden von Daten aus dem Speicher (Mnemonisch: LOAD) sind ebenfalls im I-Typ Format definiert.

Tabelle 2.4: I-Typ Befehlsübersicht

Mnemonik	**Bedeutung**	**Operation**	**funct3**	**funct7**
ADDI	Addition	rd:=$rs1 + imm$	000	-
SLTI	Signed Compare	rd:=1, if $rs1 < imm$	010	imm
SLTIU	Unsigned Compare	rd:=1, if $rs1 < imm$	011	imm
ANDI	Bitwise Logical	rd:=$rs1 \& imm$	111	imm
ORI	Bitwise Logical	rd:=$rs1 \mid imm$	110	imm
XORI	Bitwise Logical	rd:=$rs1$ xor imm	100	imm
SLLI	Logical Left Shift	rd:=$rs1 <<$(value of)imm	001	0000000
SRLI	Logical Right Shift	rd:=$rs1 >>$(value of)imm	101	0000000
SRAI	Arithmetic Right Shift	rd:=signed $rs1 >>$(value of)imm	101	0100000
NOP	Zero operation	=ADDI x0, x0, 0	-	-
LW	Load word	rd:=32-bit-value at address of rs1	010	imm
LH	Load halfword	rd:=16-bit-value at address of rs1	001	imm
LB	LoadByte	rd:=8-bit-value at address of rs1	000	imm
LBU	Load word	rd:=32-bit-value at address of rs1	100	imm
LHU	Load word	rd:=32-bit-value at address of rs1	101	imm
JALR	Jump and link register	*siehe Beschreibung S.14*	000	imm

Im Gegensatz zu Speicherbefehlen, die mit dem S-Typ ein eigenes Format besitzen und in einem separaten Abschnitt beschrieben werden. Es sind unterschiedliche Ladebefehle definiert, welche alle einen OpCode mit dem Wert 0000011 besitzen. Bei Ladebefehlen gibt das Register rs1 die Speicheradresse an, welche mit einen vorzeichenbehafteten 12-Bit Offset imm[11:0] verknüpft werden kann. Beim LW Befehl wird ein 32-Bit Datenwort von der entsprechenden Speicheradresse geladen und in das Zielregister rd geschrieben. Bei den Befehlen LH und LB werden jeweils die niederwertigsten 16-Bits bzw. 8-Bits geladen. Der LHU Befehl lädt ein 16-Bit Halbwort und führt durch eine Auffüllung des Registers mit Nullen eine Erweiterung auf die volle 32 Bit Datenwortbreite durch. In gleicher Weise wird bei Nutzung des LBU Befehls das ausgelesende Byte Nullen auf die volle Datenwortbreite aufgefüllt. Der JALR Befehl entspricht einem unbedingten Sprungbefehl, welcher durch den OpCode 1101011 kenntlich gemacht wird. Die Adresse wird durch die folgende Rechenvorschrift festgelegt. Der Befehls-Code des Sprungbefehls enthält eine vorzeichenbehaftete Konstante in einer Zweierkomplement Darstellung. Diese Konstante wird mit dem Wert aus Register rs1 addiert und anschließend das niederwertigste Bit auf Null gesetzt. Das Resultat wird als Sprungadresse des nächsten Befehls in das Zielregister rd geschrieben.

In der RISC-V Architektur werden drei verschiedene Speicherbefehle definiert, welche im S-Typ

formatiert sind. Der OpCode dieser Befehle entspricht dem Wert 0100011. In Abbildung 2.5 ist das Befehlsformat abgebildet. Das Register rs2 enthält dabei den Wert, der in den Datenspeicher geschrieben werden soll. Dabei wird aus dem Wert in Register rs1 und dem vorzeichenbehafteten Offset imm[11:5]+imm[4:0] die Zieladresse ermittelt. In die adressierte Speicherstelle wird je nach verwendetem Befehl entweder ein word, halfword oder Byte geschrieben.

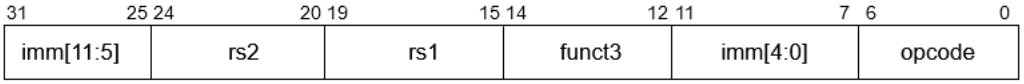

Abbildung 2.5: S-Type Befehlsformat

Tabelle 2.5: S-Typ Befehlsübersicht

Mnemonik	**Bedeutung**	**Operation**	**funct3**
SW	Store word	Store rs2 in Address(rs1+Offset)	010
SH	Store halfword	Store rs2[15:0] in Address(rs1+Offset)	001
SB	Store Byte	Store rs2[7:0] in Address(rs1+Offset)	000

Dabei werden immer die niederwertigsten Bits des in rs2 festgelegten Registers gespeichert. In Tabelle 2.5 wird ein Überblick auf verschiedene Adressierungsbefehle und ihre Eigenschaften gegeben.

Bei den Befehlen des B-Typ handelt es sich um bedingte Sprungbefehle. In Abbildung 2.6 ist das Befehlsformat dargestellt, wobei die definierten Bitfelder, den folgenden Einfluss auf die Ausführung des Sprunges besitzen.

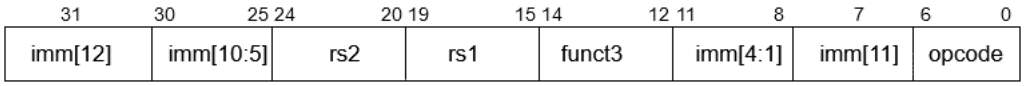

Abbildung 2.6: B-Typ Befehlsformat für bedingte Sprungbefehle

Tabelle 2.6: B-Typ Befehlsübersicht

Mnemonik	**Bedeutung**	**Operation**	**funct3**
SW	Store word	Store rs2 in Address(rs1+Offset)	010
SH	Store halfword	Store rs2[15:0] in Address(rs1+Offset)	001
SB	Store Byte	Store rs2[7:0] in Address(rs1+Offset)	000

Der OpCode für bedingte Sprungbefehle entspricht dem Wert 1100011. Die vorzeichenbehaftete Konstante *imm* mit 12-Bit Breite wird zur aktuellen Adresse des Sprungbefehls addiert, um die Zieladresse zu erhalten. Als Bedingung für die Ausführung eines Sprungbefehls werden die in den Bitfeldern rs1 und rs2 festgelegten Register miteinander verglichen. Das Ergebnis des Vergleiches

legt fest, ob der Sprungbefehl ausgeführt wird oder nicht. Es existieren sechs verschiedene bedingte Sprungbefehle, welche in Tabelle 2.6 gelistet und beschrieben werden. Der Befehl BEQ führt einen Sprungbefehl durch, wenn rs1 und rs2 gleich sind. Bei dem Befehl BNE wird ein Sprungbefehl ausgeführt, wenn dies nicht der Fall ist. Daneben bieten die Befehle BLT bzw. BLTU noch die Möglichkeit zu prüfen, ob rs1 kleiner als rs2 ist, und zwar unter Berücksichtigung bzw. ohne Berücksichtigung des Vorzeichens. Weiterhin kann auch die gegenteilige Überprüfung durchgeführt werden. Dies geschieht unter Verwendung der BGE und BGEU Befehlen.

U-Typ Befehle werden für die Bildung von Variablen verwendet. Im Vergleich zu den anderen Typen existieren in diesem Format viel weniger Befehle. In Abbildung 2.7 ist das Befehlsformat abgebildet und in Tabelle 2.7 wird eine Übersicht über gängige U-Typ Befehle gegeben.

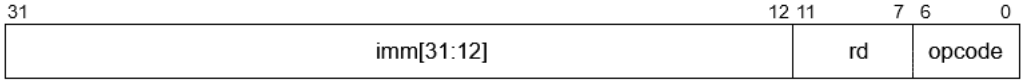

Abbildung 2.7: U-Typ Befehlsformat für das Laden von Konstanten

Tabelle 2.7: U-Typ Befehlsübersicht

Mnemonik	**Bedeutung**	**Operation**	**OpCode**
LUI	Load upper immediate	rd:=$imm[31:12] + [(11:0) = 0]$	0110111
AUIPC	Add upper immediate to pc	rd:=$(imm[31:12] + [(11:0) = 0]) + pc$	0010111

Der LUI Befehl kreiert eine 32-Bit Konstante. Dabei entsprechen die 20 hochwertigsten Bits des Befehlsformats imm[31:12] Konstanten, welche mit der gleichen Wertigkeit in das Zielregister rd geschrieben werden. Die niederwertigsten 12 Bits werden mit Nullen befüllt. Der AUIPC Befehl ist für die Verarbeitung des Programmzählers verantwortlich. Analog zum LUI Befehl wird das gleiche Bitfeld des Befehlsformats als Konstante verwendet. Allerdings entspricht diese Konstante einem Offset, um den die aktuelle Adresse des Programmzählers geändert und anschließend in das Zielregister rd geschrieben wird.

Das Befehlsformat J-Typ definiert einen unbedingten Sprungbefehl JAL. Das Befehlsformat ist in Abbildung 2.8 dargestellt. Neben dem JAL Befehl gibt es auch den JALR als unbedingten Sprungbefehl, der allerdings im I-Typ Format definiert ist. Die im Befehl definierte 12 bit Konstante ist im Zweierkomplement codiert und wird auf die momentane Befehlsadresse addiert, um zur gewünschten Zieladresse zu gelangen. Gleichzeitig wird im Zielregister rd die Rücksprungadresse (pc + 4) gespeichert. In Tabelle 2.8 ist dieser Befehl noch einmal zusammengefasst.

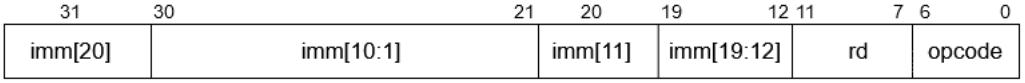

Abbildung 2.8: J-Typ Befehlsformat für unbedingte Sprungbefehl

Tabelle 2.8: J-Typ Befehlsübersicht

Mnemonik	Bedeutung	Operation	OpCode
JAL	Jump and link	rd:=$pc + 4$	1101111

Die Systembefehle verwalten den Zugang zu Systemfunktionen und sind ebenfalls im I-Typ definiert. Dabei werden zwei verschiedene Kategorien differenziert: Zum einen die Befehle, die automatisch die Status- und Kontrollregister schreiben und alle anderen exklusiven Befehlen, wie beispielsweise ECALL und EBREAK. Die Control and Status Register (CSR)-Befehle sind Teil des Zicsr Zusatzformates. Die beiden genannten Befehle sind in Tabelle 2.9 dargestellt.

Tabelle 2.9: System-Befehlsübersicht

Mnemonik	Bedeutung	funct3
ECALL	service request to execution environment	000
EBREAK	return control to debug environment	000

Der OpCode für diese Befehle besitzt den Wert 1110011. Die beiden Bitfelder rs1 und rd werden dabei mit Nullen beschrieben, genau wie das Bitfeld funct3=000. Der einzige Unterschied zwischen den beiden Befehlen liegt im Bitfeld imm[11:0], das beim ECALL Befehl den Wert 000000000000, und beim EBREAK den Wert 000000000001 annimmt. Der ECALL Befehl bewirkt einen Austausch zwischen der ausführenden Umgebung des Programms. Der EBREAK Befehl wird zu Debugging-Zwecken verwendet.

Befehlssatzerweiterung M

Die Befehlssatzerweiterung M umfasst Multiplikations- und Divisionsbefehle. Die Zielarchitektur RV32IM besteht aus dem Basisbefehlssatz und dieser Befehlserweiterung. Dabei werden zwei Register rs1 und rs2 als Operanden verwendet, während das Ergebnis in das Zielregister rd geschrieben wird. In Abbildung 2.9 ist das Befehlsformat dargestellt und in Tabelle 2.10 sind die Multiplikations- und Divisionsbefehle gelistet.

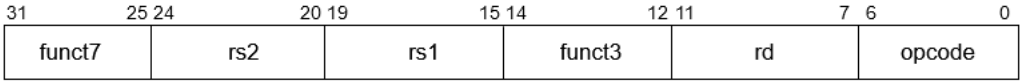

Abbildung 2.9: M-Befehlsformat für Multiplikation und Division

Die Multiplikation von zwei 32-Bit breiten Registern führt zu einen 64-Bit breiten Datenwort. Für die Speicherung der niederwertigsten 32 bit wird der Befehl MUL verwendet. Alle weiteren Befehle Speichern die höherwertigen 32 bit in das Zielregister. Dabei wird unterschieden, ob die Datenworte in den beiden Registern vorzeichenbehaftet sind. Dies wird in Tabelle 2.10 verdeutlicht.

Tabelle 2.10: M-Befehlsübersicht

Mnemonik	**Bedeutung**	**Operation**	**funct3**
MUL	Multiplication Low	rd:=$Lower32Bits of(rs1 x rs2)$	000
MULH	Multiplication High signedxsigned	rd:=$High32Bits of(rs1 x rs2)$	001
MULHU	Multiplication High unsignedxunsigned	rd:=$High32Bits of(rs1 x rs2)$	011
MULHSU	Multiplication High signedxunsigned	rd:=$High32Bits of(rs1 x rs2)$	010
DIV	Division signed	rd:=$r1/r2$	100
DIVU	Division unsigned	rd:=$r1/r2$	101
REM	Reminder signed	rd:=$r1/r2$	110
REMU	Reminder unsigned	rd:=$r1/r2$	111

Bei der Division ist das Register rs1 der Dividend und rs2 der Divisor, wobei das Ergebnis abgerundet und in das Zielregister rd geschrieben wird. Die hochwertigsten sieben Bits besitzen bei allen Befehlen den Wert funct7=0000001. Der OpCode ist ebenfalls bei allen Befehlen dieser Erweiterung identisch und besitzt den Wert 0110011. Durch die drei Bits des funct3 Bitfeldes werden die Befehle unterschieden. Bei der Division zweier Registerwerte mit Speicherung des Quotienten wird für vorzeichenbehaftete Registerwerte der Befehl DIV und für nicht vorzeichenbehaftete Registerwerte der Befehl DIVU verwendet. Die Befehle REM und REMU speichern den Divisionsrest derselben Operation. Sind beide Werte für eine Division als Ergebnis gewünscht, müssen beide Instruktionen nacheinander ausgeführt werden. Sonderfälle bei Divisionsbefehlen entstehen beispielsweise bei einer Division durch Null. Wird ein Register durch Null dividiert, entspricht der Quotient dem Wert 0xffffffff und der Rest entspricht dem Dividenden. Desweiteren kann ein Overflow auftreten, wenn bei der vorzeichenbehafteten Division die kleinst möglichste Zahl durch -1 dividiert wird. Dies entspricht bei 32-Bit Registerbreite der Operation: $(-2147483648)/(-1)$. Bei dieser Division entspricht der Quotient dem Dividenden, während der Rest Null beträgt.

2.2.3 Debug Spezifikation

Über die Debugschnittstelle können alle Register gelesen, die schreibbaren Register beschrieben und der Speicherinhalt des RISC-V Prozessors abgerufen werden. Der genaue Aufbau einer Debug Kette ist in Abbildung 2.10 dargestellt. Das Debug System besteht aus drei verschiedenen Hardware-Einheiten. Die erste Komponente ist die implementierte RISC-V Hardware. Desweiteren wird ein JTAG Adapter als Schnittstelle zwischen dem Debug Anwender und der RISC-V Hardware eingesetzt. Der Debug Anwender nutzt eine Rechnereinheit, auf der zwei Software-Komponenten ausgeführt werden. Zum einen den Debugger, der die Debug Informationen des Nutzers auswertet, und zum anderen den Debug Übersetzer, der die Debug-Informationen an den JTAG Adapter sendet. Das Debug Transport Module (DTM) Modul entspricht der JTAG Schnittstelle und ist im folgendem Abschnitt näher beschrieben. Dieses Modul kommuniziert mit dem Debug Module Interface (DMI), welches gemäß der RISC-V Spezifikation notwendige und optionale Funktionalitäten vorweisen muss und in Abschnitt 3.1.4 beschrieben ist.

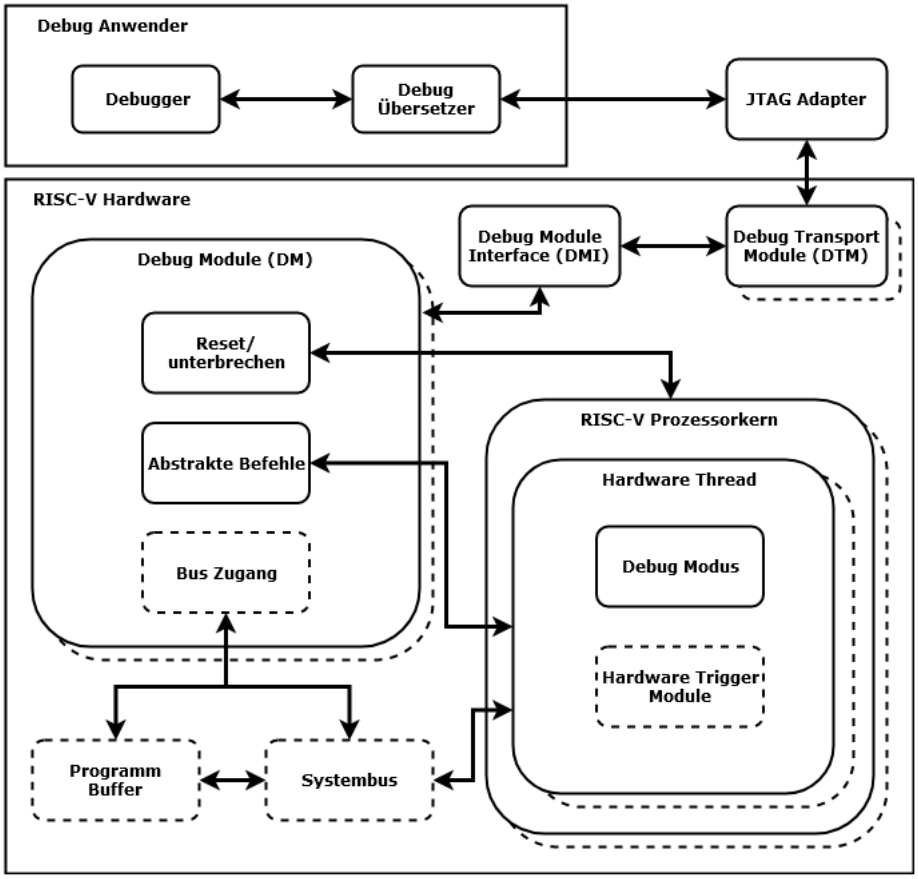

Abbildung 2.10: RISC-V Debug System Überblick [7]

Im vorliegenden Design wird das DMI und das Debug Module (DM) des Prozessorkerns im raifes_debug_modul zusammengefasst. Das DM verfügt über Zugriff auf alle Registers des Prozessors. Ein Zugriff auf die Busstruktur ist optional und für das vorliegende Design nicht implementiert.

Debug Transport Modul

Das DTM Modul ist der Knotenpunkt zwischen dem externen JTAG Adapter und dem DMI und entspricht dem in Abschnitt 2.3 beschrieben JTAG Test Access Port (TAP). Pro Prozessorkern bzw. pro DM sollte nur ein DTM implementiert sein. Ein paralleler Betrieb von mehreren DTMs ist nicht zulässig. Im vorliegenden Design wird bei einem Zugriff zuerst das Instruktionsregister (IR) mit einer Zieladresse des zu lesenden bzw. zu schreibenden Datenregister (DR) beschrieben. Dabei hat das IR eine Breite von 5-Bits. Die aktuell implementierten DRs und deren Eigenschaften sind in Tabelle 2.11 gelistet. Dabei wird momentan nur das Register DMI genutzt.

Tabelle 2.11: DTM DR Register

Adresse	Breite	Name	Beschreibung
0x01	32	IDCODE	Identifikationsnummer des Geräts
0x10	32	DTM	Steuerung des DTMs
0x11	40	DMI	Daten für das DMI

Das IDCODE Register wird gesetzt, wenn der JTAG TAP zurückgesetzt wird und ist ein reines Leseregister. Mit dem Register DTM wird das DTM Modul gesteuert. Eine Übersicht über dieses Register ist in Abbildung 2.11 gegeben. Die höherwertigen 14 Bits [31:18] und das Bit 15 haben den Wert Null. Das Bit 17 dmihardreset, wird auf eins gesetzt, falls das DTM hart zurückgesetzt und nachfolgende DMI Übertragungen verworfen werden sollen. Dieses Bit wird nur dann gesetzt, wenn sicher ist, dass die angesprochenen DMI Transaktionen nicht beendet werden. Das Bit 16 dmireset hebt bei einem Wert von eins einen Fehlerzustand auf und versucht die letzte Transaktion des DMIs zu wiederholen.

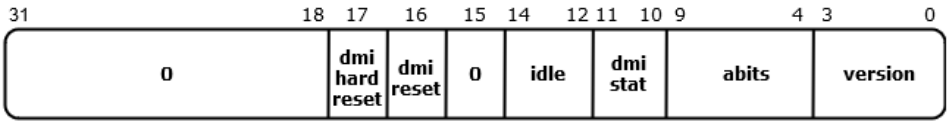

Abbildung 2.11: DTM DTM Register [7]

Das Bitfeld idle [14:12] gibt die minimale Anzahl an Takten an, in denen der Debugger im Zustand Run-Test/Idle des JTAG TAPs verharren soll, ohne einen Zustands-Code busy mit dem Wert eins in das Bitfeld dmistat zu schreiben. Wenn der Wert Null im Register steht, wird angezeigt, dass der Zustand Run-Test/Idle nicht angenommen werden soll. Bei einem Wert von eins wird der Zustand Run-Test/Idle für einen Takt angenommen und danach wieder verlassen. Bei einem Wert von 0x2 wird der Zustand angenommen und für einen weiteren Takt beibehalten. Eine Inkrementierung des Wertes bedeutet eine Erhöhung der Verweildauer im Run-Test/Idle Zustand. Das idle Bitfeld ist ein reines Lesebitfeld und nimmt bei einem Reset die Voreinstellung an.
Das Bitfeld dmistat ist ein Schreibregister und kann vier verschiedene Werte annehmen. Bei einem Reset nimmt es den Wert 0x0 an. Dieser Wert signalisiert den fehlerfreien Zustand des DMI. Die Werte 0x1 und 0x2 geben Fehlerzustände an, bei denen Operationen nicht erfolgreich waren. Der Wert 0x3 gibt an, das ein Versuch unternommen wurde, obwohl das DMI noch in Benutzung war. Das Bitfeld abits ist ebenfalls ein Schreibregister und gibt die Bitbreite der DMI Adressen an, die im vorliegenden Design sieben Bit beträgt. Beim Zurücksetzen wird das Bitfeld mit dem Wert sieben beschrieben, wobei nur das Bitfeld dmistat[11:10] auf den Wert Null gesetzt werden muss. Das letzte Bitfeld version[3:0] kann drei verschiedene Werte annehmen, welche Auskunft über die Version der Implementierung des DTMs geben. Der Wert eins steht für die Implementierung nach der RISC-V Spezifikation 0.11 und der Wert zwei für die Implementierung nach der Version 0.13, welche momentan die aktuellste ist. Bei einem Wert von 0xF entspricht die implementierte Version keiner der offiziellen Spezifikationen.

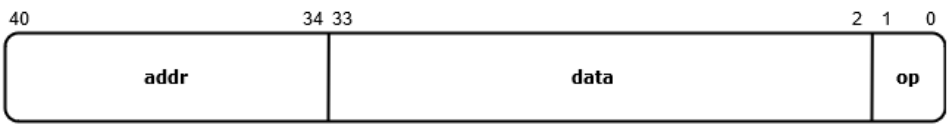

Abbildung 2.12: DTM DMI Register [7]

Das Debug Module Interface Access Register DMI ermöglicht dem DTM den Zugang zum DMI und wird für den Test des Design verwendet. Eine Übersicht des Registers ist in Abbildung 2.12 aufgeführt. Bei einem Reset werden alle Bits auf den Wert Null gesetzt. Das Bitfeld op [1:0] gibt an, ob ein Lese- oder Schreibzugriff ausgeführt wird. Das Bitfeld data [33:2] entspricht den Daten, die an das DMI weitergeleitet bzw. von dem DMI gelesen werden sollen. Das Bitfeld address [abits+33:34] gibt die Adresse des DMI Zielregisters an und hat im vorliegenden Design eine Bitbreite von sieben Bits. Wie genau die Schaltungslogik für das vorliegende Design aufgebaut ist, wird in Abschnitt 3.1.4 beschrieben.

2.3 JTAG IEEE Standard 1149.1-2001

Der IEEE Standard 1149.1-2001 definiert die JTAG Funktionalität, durch deren Implementierung das Lesen und Schreiben von Daten einer zu testenden Hardware-Logik-Einheit möglich macht. Dazu werden zwei spezielle Register implementiert, die als Schnittstelle zwischen externer JTAG Hardware und der zu untersuchenden Hardware dienen. Diese Register sind zum einen das Befehls- bzw. IR und zum anderen das Daten- bzw. DR Register.

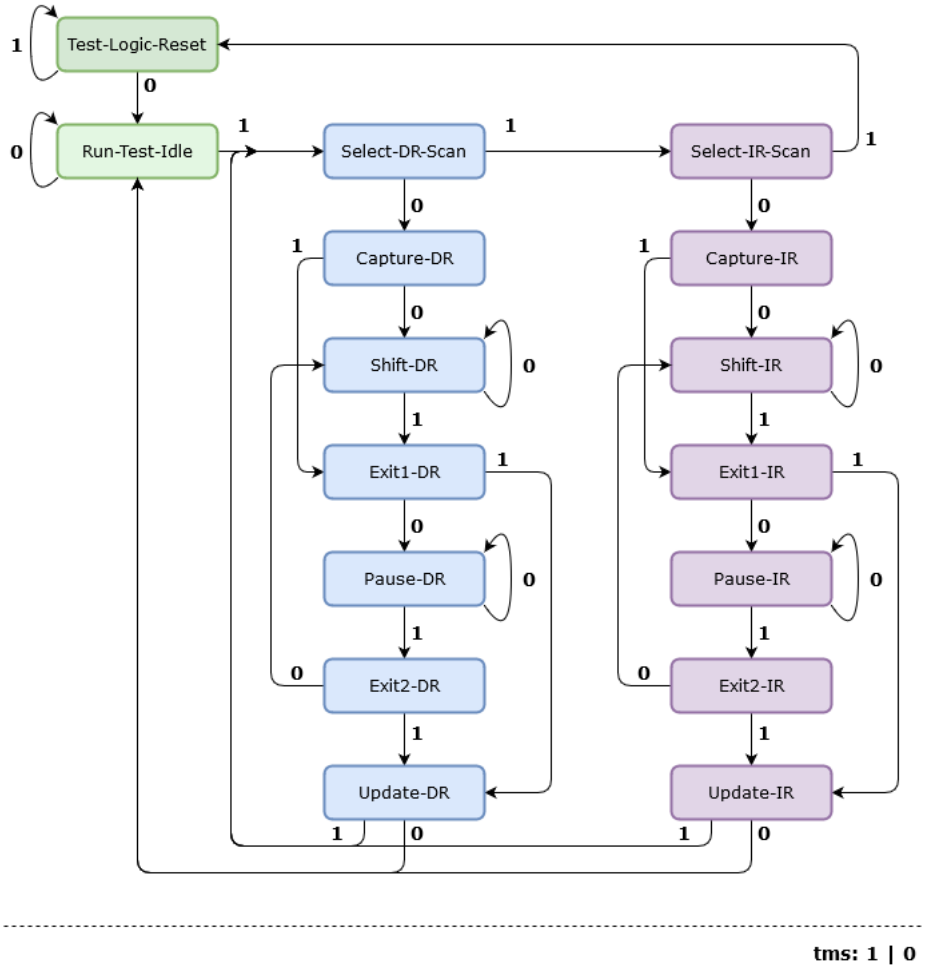

Abbildung 2.13: JTAG TAP Zustandsmaschine nach IEEE 1149.1-2001 [8]

Der Zugriff auf diese Register wird durch den TAP gesteuert. Dieser ist in Abbildung 2.13 als Zustandsmaschine dargestellt. Dabei beinhaltet das IR Steuerinformationen für den TAP und das DR Lese- oder Schreibdaten von bzw. für die Test-Hardware. Der TAP wird über die Signale tck, tms, tdi und tdo angesteuert. Hierbei entspricht das Signal tck dem Prüftakt, mit welchem die anderen Signale synchronisiert werden. Durch das Signal tms, der Testmodusauswahl, wird der TAP gesteuert und ist in Abbildung 2.13 als Dezimalzahl dargestellt. Über das Ein- bzw Ausgangssignal, tdi bzw. tdo, werden die Daten seriell geschrieben bzw. gelesen.
Um Daten in das IR zu schreiben, muss die Zustandsmaschine in den Zustand Shift-DR gesetzt werden. Da sich der TAP initial im Zustand Run-Test-Idle befindet, muss pro JTAG Takt tck das Signal tms nacheinander auf folgende Wertekombination gesetzt werden: 0,1,0. Solang das Signal tms nicht auf den Wert eins gesetzt wird, ändert sich der Zustand nicht und Daten können entweder über das Signal tdi geschrieben oder über das Signal tdo gelesen werden. Nachdem die Daten geschrieben worden sind, muss durch die Kombination 1011 das Register aktualisiert werden. Danach kann entweder wieder in den Anfangszustand oder in den Zustand Select-DR-Scan gesprungen werden [8].

3 RISC-V RV32IM auf RTL Ebene

In diesem Kapitel wird der Raifes Kern und die Implementierung des Caecos auf RTL Ebene analysiert und beschrieben. Das Verständnis der Architektur des Prozessors ist für die Implementierung notwendig, da der Caeco die eigentliche Prozessorstruktur nicht einschränken, sondern ergänzen soll. Dafür werden besondere Mechanismen, wie z.B. die Interrupt-Logik oder der Datentransport zu den Peripheriegeräten untersucht. Im Abschnitt 3.1 wird die Architektur des Prozessors anhand der HDL Dateien beschrieben. Im darauf folgenden Abschnitt 3.2 wird die Implementierung des Caecos im Detail anhand von Code-Ausschnitten erläutert.

3.1 RISC-V RV32IM Prozessorarchitektur

Der Raifes Kern basiert auf dem vScale-Project, einem frei verfügbaren RISC-V RV32IM Prozessorkern in Verilog. Das vScale Projekt ist als Verilog-Version des in Berkeley entwickelten zScale RV32IM Prozessors mit dem Ziel entstanden, als kleiner und leistungsfähiger Prozessor dem ARM Cortex-M0 Konkurrenz zu machen. Die Bus-Architektur basiert auf der von ARM entwickelten Advanced Microcontroller Bus Architecture (AMBA) und Advanced High-Performance Bus (AHB)-Lite Spezifikation, die allerdings unter RISC-V als Highly Advanced System Transport Interface (HASTI) bezeichnet werden [9]. Die Prozessor- bzw. Systemarchitektur ist in Abbildung 3.1 grafisch dargestellt.

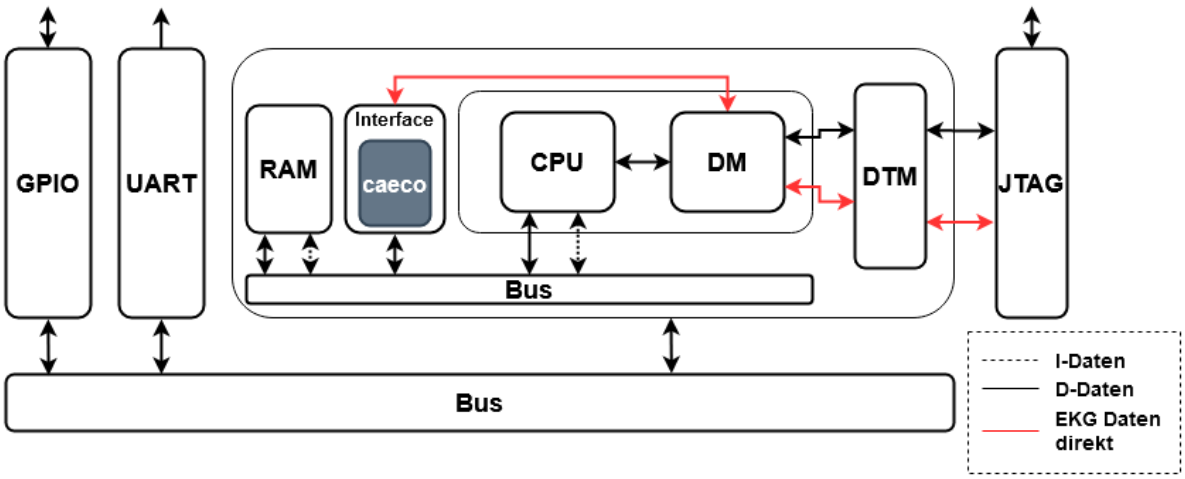

Abbildung 3.1: RV32IM Systemarchitektur

Der Prozessor verfügt über eine JTAG- und eine UART Schnittstelle und diverse General Purpose Input/Outputs (GPIOs). Die UART Schnittstelle ist unidirektional und wird zum Senden von Daten verwendet. Die GPIOs umfassen im Zuge dieser Arbeit nur eine Signal-LED, durch die die Aktivität des Caecos angezeigt wird. Über die bidirektionale JTAG Schnittstelle werden einerseits sowohl die Programmdaten des Prozessors, als auch die EKG Daten für den Caeco geschrieben, andererseits wird die Schnittstelle zum Testen and damit zum Auslesen von Registerwerten verwendet. Um die Verbindung nutzen zu können, ist das DTM notwendig, welches den JTAG TAP beschreibt. Wie bereits in Abschnitt 2.2.3 beschrieben, beinhaltet das DM das RISC-V DMI und das Debug Modul für den Prozessorkern, wodurch der Datenaustausch organisiert wird. Der Caeco wird über das Caecointerface instantiiert und als weiterer Peripheriebaustein in die Architektur hinzugefügt. Als Speicher für die Instruktionen und der Daten dient ein Xilinx IP True Dual Port Random Access Memory (RAM). Ein Überblick über alle HDL Dateien und deren Gliederung ist in Abbildung 3.2 gegeben. Die in der Beschreibung verwendeten Signal- und Registernamen entsprechen den Bezeichnungen aus dem Verilog-Design.

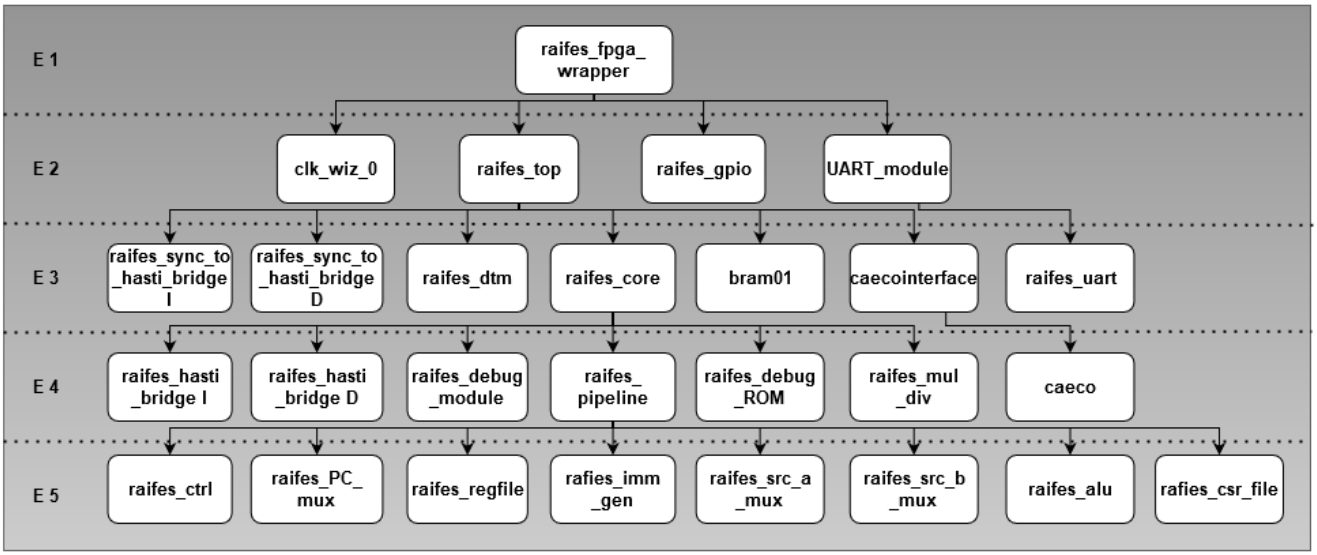

Abbildung 3.2: Gesamtprojektübersicht der Verilogmodule

Das Top-Modul raifes_fpga_wrapper bildet die oberste Hierarchie Ebene (E 1) und instantiiert vier Untermodule. Dabei wird der Systemtakt von 100 MHz des externen Oszillator durch den Xilinx IP Clock Wizard auf 25 MHz geteilt. Die Module UART_module und raifes_uart sind für die Modellierung der UART Schnittstelle zuständig und werden in Unterabschnitt 3.1.1 detailliert beschrieben. Das Modul raifes_top stellt den Hauptbestandteil des Entwurfs dar und instantiiert den größten Teil der restlichen Module. Auf der Ebene E 3 wird der Prozessorkern als Instanz des Moduls raifes_core eingebunden. Auf der gleichen Ebene befindet sich noch eine Instanz des Speichers bram01, welche in Unterabschnitt 3.1.3 beschrieben wird, das DTM als Instanz des Moduls raifes_dtm, das in Unterabschnitt 3.1.4 beschrieben wird, ein Teil der Busstruktur in den Instanzen der Module raifes_sync_to hasti_bridge_I/D, die in Unterabschnitt 3.1.2 erläutert werden, und die Caeco-Schnittstelle als Instanz des Moduls caecointerface, welches in dieser Arbeit

entwickelt wird und den Caeco instantiiert. In den Ebenen E 4 und E 5 befinden sich ausschließlich Komponenten des Prozessorkerns und dessen interner Busstruktur, was in Unterabschnitt 3.1.5 genauer beschrieben wird. Eine Übersicht über die gesamte Prozessorstruktur auf RTL Ebene ist im Anhang B zu finden.

3.1.1 Die UART Schnittstelle

Die UART Schnittstelle dient der seriellen Übertragung von Daten, die vom Prozessor an ein externes Geräten gesendet werden. Dabei beträgt die Übertragungsgeschwindigkeit bei einem Systemtakt von 25 MHz 115200 Baud. Es werden ein Startbit, acht Datenbits und zwei Stop-Bits übertragen. Die Signalübersicht bezüglich des übergeordneten Moduls UART_module und des instantiierten Moduls raifes_uart sind in Abbildung 3.3 gegeben.

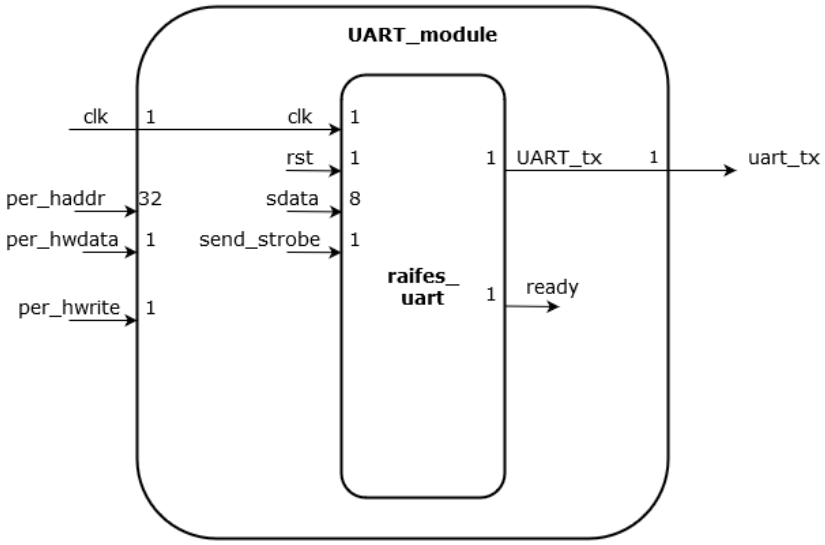

Abbildung 3.3: UART Module

Die Funktion des Moduls UART_module besteht darin, bei Ausführung eines Schreibbefehls und bei einer anliegenden Adresse von 0xc0000000 des Signals per_haddr und des gesetzten Signals per_hwrite die niederwertigsten zehn Bits des per_hwdata Signals synchron in das Register UART_reg zu schreiben. Die eigentliche Datentransmission wird durch eine Zustandsmaschine im Untermodul raifes_uart gesteuert und ist in Abbildung 3.4 abgebildet. Dabei sind die niederwertigsten acht Bits des Registers UART_reg die zu sendenden Daten, welche über das Signal sdata dem Untermodul übergeben werden. Bit neun aus dem Register UART_reg entspricht dem Signal send_strobe. Initial befindet sich die Zustandsmaschine im Zustand ready und erwartet den Wert eins für das Signal send_strobe, wobei das Ausgangssignal UART_TX auf eins gehalten wird. Sobald das Signal send_strobe gesetzt ist, werden die zu sendenden Daten in das Datenlatch txData an die Stelle neun bis zwei geschrieben. Dabei ist das MSB auf den Wert eins und die beiden niederwertigsten Bits auf null gesetzt, welche dem Startbit und den beiden Stopbits entsprechen.

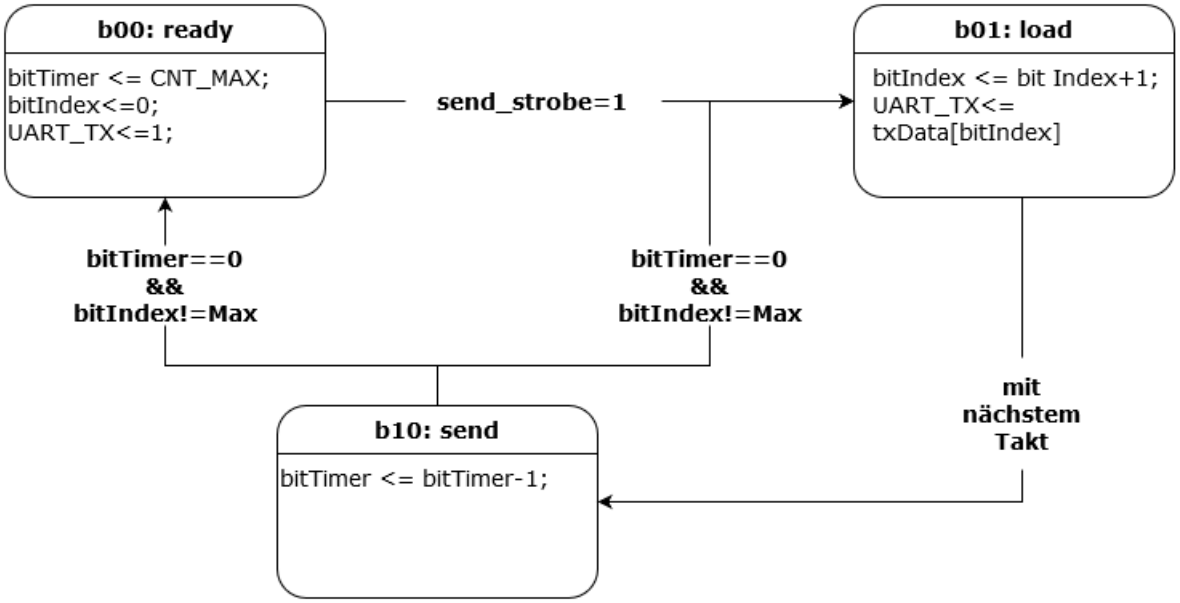

Abbildung 3.4: UART Zustandsmaschine

Außerdem springt die Zustandsmaschine für einen Takt in den Zustand load. In diesem Zustand wird dem Ausgangssignal der Wert des Bits aus dem Datenlatch zugewiesen, der durch den Wert des Registers bitIndex angegeben wird. Zusätzlich wird das Register bitIndex um den Wert eins inkrementiert und die Zustandsmaschine springt in den nächsten Zustand send. In diesem Zustand wird solang verweilt, bis ein Zähler von dem Wert 0xd8 bzw. CNT_MAX auf den Wert null dekrementiert wurde. Ist der Wert des Registers bitIndex ungleich zehn, wird wieder in den Zustand load gesprungen, sodass das nächste Bit übertragen werden kann. Wurden alle zehn Bits übertragen, springt die Zustandsmaschine wieder in den Zustand ready und erwartet ein erneutes Startsignal durch das Signal send_strobe.

3.1.2 Bussystem

Das HASTI Bussystem trennt die Daten (D) und Instruktionen (I) und wird durch die Module raifes_hasti_bridge I/D bzw. raifes_sync_to_hasti_bridge I/D beschrieben. Dabei bestehen die ersten beiden Module aus rein kombinatorischen Signalzuweisungen, während letztere beiden Module unter anderem getaktete Speicher verwenden. Alle Ausgangssignale, bis auf core_mem_rdata und core_mem_wait, der ersten beiden Module stellen die Eingangssignale der letzten beiden Module dar. In Abbildung D.5 bis Abbildung D.7 wird eine Übersicht auf die Signale der vier Module gegeben.
Die Datenadresse wird über das Signal core_mem_addr transportiert und befindet sich im Datenlatch dmem_addr_r_reg der Pipeline. Die Adresse wird im getakteten Register core_haddr_r des HASTI D-Bus gespeichert und über das Ausgangssignal dev_haddr an die Peripherie weitergeleitet. Dabei wird bei einem Speicherbefehl die Adresse aus dem Latch dmem_addr_r_reg entnommen, andernfalls wird das Eingangssignal core_mem_addr weitergeleitet. Ein Speicherbefehl liegt

dann vor, wenn die Signale core_mem_en und core_mem_wen den Wert eins haben. Die zu speichernden Daten werden über die Signale core_mem_wdata_delayed und dev_hwdata von der Pipeline an die Peripherie geleitet. Ob sich der Prozessor in der Ausführung eines Schreibbefehls befindet, wird über die Signale core_mem_wait und dev_hwrite signalisiert, wobei ersteres ein Steuerungssignal für den Prozessorkern ist und letzteres an die Peripherie geleitet wird. Über das Signal core_mem_size wird die Datenbreite für den Speicherzugriff bestimmt. Diese Information wird parallel zu der Adresse bei einem Schreibbefehl in das Register core_hsize_r geschrieben. Über das Signal dev_hsize wird die Information dann an die Peripherie weitergeleitet. Über die Signale dev_hrdata und core_mem_rdata wird das Datenwort aus dem Block-RAM an den Prozessorkern geleitet.
Instruktionen werden analog zu den Daten verarbeitet, da für beide Busse Instanzen der gleichen Verilogmodule verwendet werden. Allerdings ist für Instruktionen keine Schreibfunktion vorgesehen. Die Adresse der Instruktionen wird über den Programmzähler ermittelt und liegt als Signal core_mem_addr am Bus an. Diese wird synchron über das Signal dev_haddr an den Block-RAM geleitet. Die Instruktionen werden aus dem Block-RAM über das Signal dev_hrdata and den Prozessorkern weitergeleitet.

3.1.3 Block RAM und Adressraum

Der verwendete Dual-Port Speicher ist als Xilinx IP instantiiert und dient gleichzeitig als Daten- und Instruktionsspeicher, wobei Port A als D-Bus Schnittstelle und Port B als I-Bus Schnittstelle dient. Die Adressbreite und die Datenwortbreite betragen für beide Ports 32 bit. Der mögliche Adressraum bei einer 32 bit Architektur liegt zwischen 0x00000000 und 0xffffffff und ist in Abbildung 3.5 dargestellt.

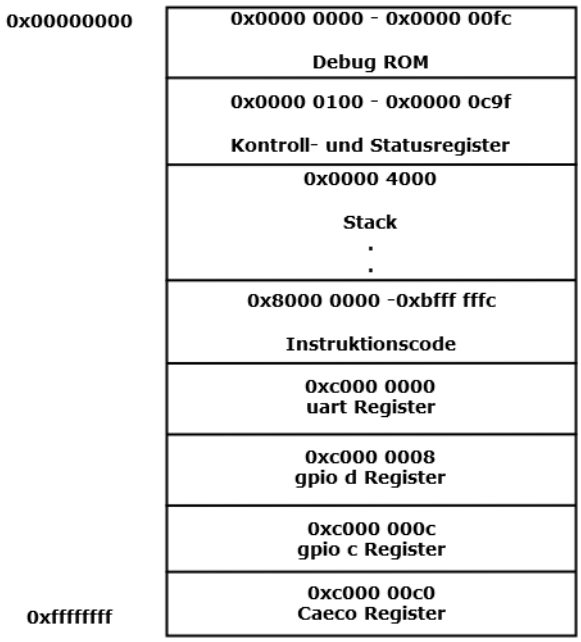

Abbildung 3.5: Adressraum

Die Größe des instantiierten Block-RAMs beträgt 262 kB. Der heap (*dynamischer Speicher*) beginnt nach der .bss Sektion und hat daher keine feste Adresse. Die zwei höherwertigen Bits der Datenwörter für die Adressen addra und addrb des Dual-Ports werden auf den Wert null gesetzt. Der Grund dafür ist, dass bei der JTAG Kommunikation das Most Significant Bit (MSB) der Adresse als Steuerungsinformation genutzt wird. Dies wird durch die Konkatenation der Adresse am Block-RAM Eingangsport, bei der die zwei höherwertigsten Bits auf den Wert null gesetzt werden, rückgängig gemacht, da ansonsten eine falsche Adresse angegeben werden würde. Über das Eingangssignal des Dual-Ports wea wird angegeben, mit welcher Datenwortbreite ein Speicherzugriff erfolgt. Besitzt das Signal wea den Wert 0xf, wird auf 32 bit, bei einem Wert von 0x3 auf 16 bit und bei einem Wert von 0x1 auf 8 bit zugegriffen.

3.1.4 JTAG und Debug Schnittstelle

Wie bereits in Unterabschnitt 2.2.3 beschrieben, folgt die im DTM implementierte Debugschnittstelle der RISC-V Hardware dem IEEE Std 1149.1-2013 Standard. Darauf wird in Abschnitt 3.1.4 noch einmal eingegangen. Danach wird die Umsetzung der RISC-V spezifischen Debug-Module genauer beschrieben, da die Debug-Spezifikation gewisse Freiheiten bei der Implementierung zulässt.

Debug Transport Module

Das Debug-Transport-Modul raifes_dtm ist die Schnittstelle zu der externen Debugger Hardware und umfasst ein JTAG TAP, über welchen die Daten des DR DMI bidirektional mit dem Pozessorkern ausgetauscht werden. Das Modul besitzt sechs Eingänge und fünf Ausgänge, die in Abbildung D.8 gelistet sind. Die Signale tck, tdi, tms und tdo, welche zur Schnittstelle des JTAG Debugger gehören, sind mit den Pins des FPGAs verbunden.
Durch das Signal tms wird die JTAG Zustandsmaschine (siehe Abbildung 2.13) gesteuert, die entweder auf das IR IR oder auf das DR zugreift. Üblicherweise wird erst das IR mit der Adresse für das DR beschrieben. Danach werden die Daten in das vorher ausgewählte DR geschrieben. In nachfolgenden Tests bezüglich des vorliegenden Designs werden die Daten nur in das DMI Register geschrieben. Bei Erreichung des Zustands UPDATE_DR werden die Ausgangssignale des Moduls dmi_addr, dmi_wdata, dmi_wen und dmi_en den Datenworten DMI[40:34], DMI[33:2], DMI[1] und DMI[0] des DR zugewiesen und der Prozessor wird beschrieben. Für die Auslesung von Daten aus dem Prozessorkern wird der Zustand CAPTURE_DR eingestellt, wodurch unter anderem das Datenwort dmi_rdata in das DR an die Stelle DMI[33:2] geschrieben wird. Dabei wird auch das Signal dmi_error an die beiden niederwertigsten Bits des DRs geschrieben. Das Signal dmi_dm_busy ist momentan nicht implementiert.

Debug Modul Interface und Debug Modul

Das Debug Modul Interface und das Debug Modul sind im Modul raifes_debug_module, innerhalb des Prozessorkerns durch zwei parallele Zustandsmaschinen zusammengefasst. Folgend wird der Begriff DM für das gesamte Modul verwendet. Das DM wird über das DTM angesteuert und kann auf den Prozessorkern zugreifen. Das Modul verfügt über elf Eingänge und 14 Ausgänge, die in Abbildung D.9 gelistet sind. Die Zustandsmaschine, die das DMI implementiert und in Abbildung 3.6 dargestellt ist, umfasst vier Zustände: Idle, Read, Write und Waitend, die im Register dmi_state gespeichert werden.

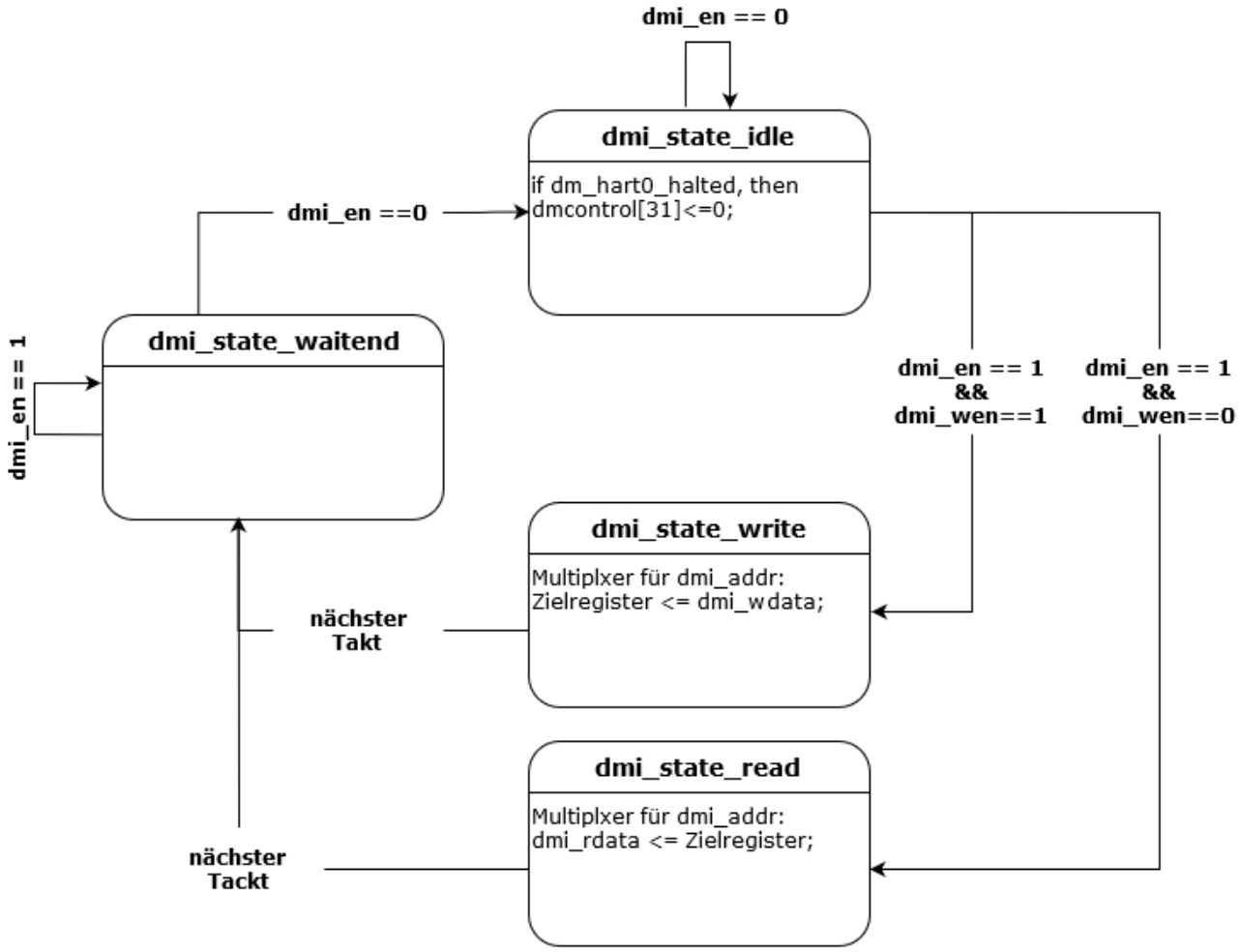

Abbildung 3.6: Debug Modul Zustandsmaschine DMI

Die Steuersignale sind hierbei dmi_en und dmi_wen. Da über das DMI die Daten bidirektional zwischen DTM und dem Prozessorkern ausgetauscht werden, können Daten aus bestimmten Registern entweder gelesen oder geschrieben werden. Initial befindet sich die Zustandsmaschine im Zustand idle, in dem keine weitere Aktion bezüglich des DMIs ausgeführt wird. Liegt an dem Eingangssignal dmi_en der Wert eins an, so wird der Zustand Read angenommen. In diesem Zustand wird, abhängig von der angegebenen Adresse im Register dmi_addr, das Leseregister dmi_rdata durch einen Multiplexer beschrieben, welches initial den Wert 0xdeadead besitzt. Bei

einem gesetzten dmi_wen Eingangssignal wird der Zustand write angenommen. In diesem Zustand wird ebenfalls abhängig von der Adresse im Register dmi_addr diverse Register des DM mit dem Wert aus dem Register dmi_wdata beschrieben. Sollten beide Signale nicht gesetzt sein, wird im Zustand Idle verharrt.

```
1 assign  dm_command = command[31:24];
2 assign dm_size = command[22:20];
3 assign  dm_size_invalid = dm_size[2] | dm_size[0];
4 assign dm_postexec = command[18];
5 assign dm_transfer = command[17];
6 assign dm_write = command[16];
7 assign  dm_regfile_access = command[12];
8 assign  dm_csr_access = ~command[12];
9 assign dm_regno = command[15:0];
```

Auflistung 3.1: DM Steuerregister command (raifes_debug_module.v Z. 306)

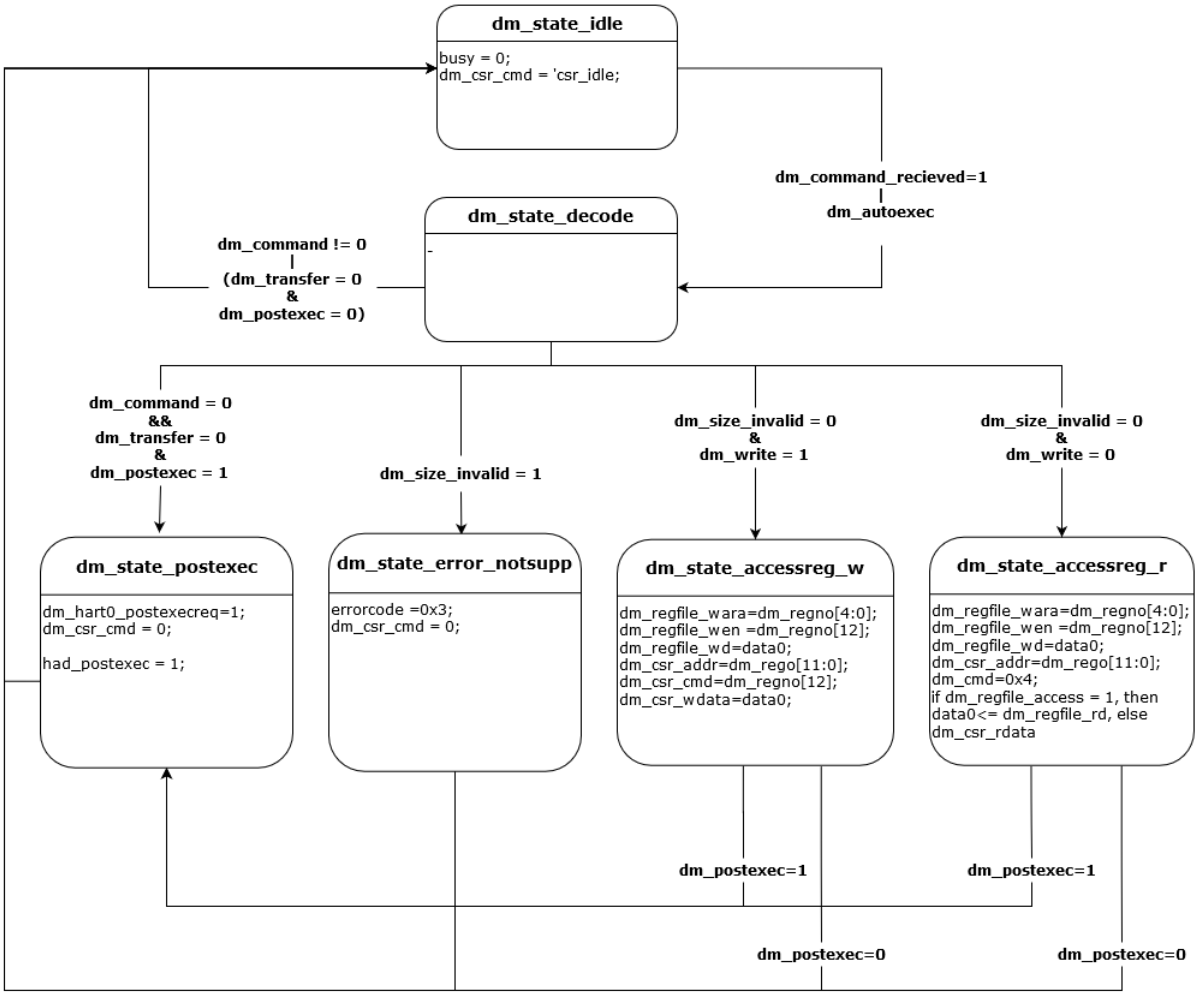

Abbildung 3.7: Debug Modul Zustandsmaschine DM

Bei beiden Zuständen Read und Write wird im nächsten Takt der Zustand Waitend angenommen, in dem solange gewartet wird, bis das Eingangssignal dmi_en wieder den Wert null hat, woraufhin der Zustand idle erneut erreicht wird.

Die Zustände der Zustandsmaschine für die Beschreibung der Debugfunktion werden analog zu der DMI Zustandsmaschine im Register dm_state gespeichert. Es können zehn verschiedene Zustände angenommen werden, die durch die Header-Datei raifes_dmi_constants.vh definiert sind. Die DM Zustandsmaschine wird in Abbildung 3.7 beschrieben. Hierbei ist command das zentrale Steueregister, das vorher über eine JTAG Befehl geschrieben werden kann (siehe Auflistung 3.1). Initial oder nach einem Reset befindet sich die Zustandsmaschine im Zustand Idle. Dieser Zustand ändert sich in den Zustand Decode, wenn das Signal dm_command_received oder das Signal dm_autoexec den Wert eins haben. Dabei wird das Signal dm_command_received auf den Wert eins gesetzt, wenn das Register dmi_state den Wert 0x4 und das Register dmi_addr den Wert 0x17 hat. Dies ist der Fall, wenn das DM von dem DMI Daten bekommt, wobei das Register command mit dem Datenwort aus dem Register dmi_wdata beschrieben wird. Andererseits wird das Signal dm_autoexec durch eine zweistellige Und-Verknüpfung gesetzt (siehe Auflistung 3.2).

```
1 assign dm_autoexec = ((dmi_state == `DMI_STATE_WRITE) || (dmi_state ==
    `DMI_STATE_READ)) &&
2         (((dmi_addr_r == `DMI_ADDR_DATA0) && (abstractauto[0] == 1'b1)) ||
3       ((dmi_addr_r == `DMI_ADDR_PROGBUF0) && (abstractauto[16] == 1'b1)) ||
4       ((dmi_addr_r == `DMI_ADDR_PROGBUF1) && (abstractauto[17] == 1'b1)));
```

Auflistung 3.2: dm_autoexec Zuweisung (raifes_debug_module.v Z. 197)

Dabei ist die erste Aussage in eine Oder-Verknüpfung unterteilt, die auf den Wert eins gesetzt wird, wenn das Register dmi_state den Wert 0x2 für Read oder den Wert 0x4 für Write hat. Die zweite Aussage wird durch eine dreistellige Oder-Verknüpfung beschrieben, deren Aussagen wiederum durch jeweils zweistellige Und-Verknüpfungen beschrieben werden. Damit wird die zweite Aussage der übergeordneten Und-Verknüpfung wahr, wenn entweder das Register dmi_addr_r den Wert 0x4 und das niederwertigste Bit des Registers abstractauto den Wert eins hat oder wenn das Register dmi_addr_r den Wert 0x20 und Bit 16 des Registers abstractauto den Wert eins hat oder wenn das Register dmi_addr_r den Wert 0x21 und Bit 17 des Registers abstractauto den Wert eins hat.

Bezüglich der Zustandsmaschinen ist zusammenfassend festzuhalten, dass die Steuerung des DMIs über die Signale dmi_en und dmi_wen erfolgt. Soll vom DM ein Datenwort über das Register dmi_rdata gelesen werden, muss das Signal dmi_en auf den Wert eins gesetzt werden. Soll ein Datenwort über das Register dmi_wdata geschrieben werden, muss das Signal dmi_wen auf den Wert eins gesetzt sein. In beiden Fällen wird über das Adressregister dmi_addr entschieden, welche Daten entweder gelesen oder in welches Register die Daten geschrieben werden. Neben

den Registern bezüglich der DMI Funktion gibt es weitere Register, die mit den Modulen für die Kontroll- und Statusregister, dem Debug-ROM und den Allzweckregistern Daten austauschen.

3.1.5 Prozessorkern und Pipeline

Der Prozessor weist eine 3-stufige Pipelinearchitektur auf, welche im Modul raifes_pipeline implementiert wird. Dabei gibt es drei Stufen: das Laden des nächsten Befehls-Code (Instruction Fetch (IF)), die Dekodierung und Ausführung des Befehls (Decode and Execution (DX)) und das Zurückschreiben in die Register bzw. Speicher (Write Back (WB)). Die Ein- und Ausgangssignale sind in Abbildung D.10 gelistet. In der Pipeline werden hauptsächlich mehrere getaktete Speicher genutzt, um die verschiedenen Pipeline Stufen parallel abarbeiten zu können (siehe Auflistung 3.3).

```
always @(posedge clk) begin
    if (reset) begin
        PC_WB <= 0;
        store_data_WB <= 0;
        alu_out_WB <= 0;
        csr_rdata_WB <= 0;
        dmem_type_WB <= 0;
        pcpi_rd_WB <= 0;
    end else if (~stall_WB) begin
        PC_WB <= PC_DX;
        store_data_WB <= rs2_data_bypassed;
        alu_out_WB <= alu_out;
        csr_rdata_WB <= csr_rdata;
        dmem_type_WB <= dmem_type;
        pcpi_rd_WB <= pcpi_rd;
    end
end
```

Auflistung 3.3: Register für WB Stufe (raifes_pipeline.v Z. 406)

Die Daten des D-Busses werden über das Signal dmem_rdata der Pipeline zugeführt und die Adresse der Daten wird im Datenlatch dmem_addr_r gespeichert. Über die Funktion load_data wird je nach Ladebefehl das Datenwort bearbeitet und um bestimmte Bits verschoben. Hat das Register wb_src_sel_WB den Wert eins, so wird das geladene Datenwort in das Register wb_data_WB und damit in eines der Allzweckregister geschrieben (siehe Auflistung 3.15). Die Instruktionsdaten imem_rdata werden synchron in das Register inst_DX geschrieben, vorausgesetzt, dass keine Art von Störung vorliegt und die Signale stall_DX und kill_IF den Wert null besitzen. Zeitgleich wird das Register PC_DX mit dem Wert aus dem Register PC_IF beschrieben, wobei PC_IF dem Programmzähler entspricht, der auf die nächste Instruktion zeigt. Beim Zurücksetzen oder

einem Störfall wird die Startadresse 0x80000000 in das Register PC_DX und der Wert für eine Nulloperation in das Register inst_DX geschrieben (dargestellt in Auflistung 3.4).

```
always @(posedge clk) begin
    if (reset) begin
    PC_DX <= `START_ADDRESS;
    inst_DX <= `RV_NOP;
    end else if (~stall_DX) begin
    if (kill_IF) begin
    inst_DX <= `RV_NOP;
    end else begin
    PC_DX <= PC_IF;
    inst_DX <= imem_rdata;
    end
    end
end
```

Auflistung 3.4: DX PC und Instruktion (raifes_pipeline.v Z. 305)

Die Daten aus dem Coprozessor pcpi_rd werden im Register pcpi_rd_WB gespeichert und bei einem Registerwert von drei des Registers wb_src_sel_WB an die Allzweckregister übergeben. Die zu schreibenden Daten sind im Register dmem_wdata_delayed gespeichert. Durch die Funktion store_data wird ermittelt, welcher Schreibbefehl ausgeführt werden soll. Abhängig von der aktuellen Instruktion und das angegebene Zieldatenwort wird entweder das Datenwort aus einem angegebenen Allzweckregister, aus den CSRs oder aus dem Coprozessor geladen (aufgezeigt in Auflistung 3.5).

```
function [`XPR_LEN-1:0] store_data;
    input [`XPR_LEN-1:0]            addr;
    input [`XPR_LEN-1:0]            data;
    input [`MEM_TYPE_WIDTH-1:0]     mem_type;
    begin
        case (mem_type)
            `MEM_TYPE_SB : store_data = {4{data[7:0]}};
            `MEM_TYPE_SH : store_data = {2{data[15:0]}};
            default : store_data = data;
        endcase
    end
endfunction
assign dmem_wdata_delayed =
   store_data(alu_out_WB,store_data_WB,dmem_type_WB);
```

Auflistung 3.5: Speicherfunktion (raifes_pipeline.v Z. 65)

3.1.6 Konstanten Generation

Das Modul raifes_imm_gen generiert die in RISC-V vorgesehenen Konstanten, den sogenannten Immidiates. Es ist ein Untermodul der Pipeline auf der untersten Hierarchieebene E-5, welches innerhalb eines Prozesses rein kombinatorische Signalzuweisungen implementiert, die durch eine Case-Anweisung gesteuert werden (dargestellt in Auflistung 3.6). Die Ein- und Ausgangssignale des Moduls sind in Tabelle 3.1 gelistet.

Tabelle 3.1: Konstanten Generation Signalübersicht

Signalname	I/O	Bits	Beschreibung	Verbindung
inst	I	32	Aktueller Instruktions-Code	debug_rom, bram
imm_type	I	2	Aktuelles Befehlsformat	raifes_ctrl
imm	O	32	generierte Konstante	raifes_src_b_mux

Basierend auf dem Befehlsformat imm_type des aktuellen Instruktions-Codes inst generiert das Modul durch Konkatenation einen Wert für das Register imm, welches dem Wert der Konstanten entspricht. Das Signal imm_type ist die Bedingung für die Case-Anweisung, wobei der Wert des Signals initial bei null liegt, welcher dem I-Befehlsformat entspricht. Dabei entsprechen die höherwertigen 21 Bits des Registers imm dem Wert des MSB des Signals inst. Die restlichen Bits des Signals inst werden den Bits 30 bis 20 des Registers imm zugewiesen. Bei einem Wert inst_type von eins, welcher dem S-Befehlsformat entspricht, werden die höherwertigen 21 Bits des Registers imm ebenfalls dem MSB des Signals inst zugewiesen. Die folgenden sechs Bits des Registers imm stimmen mit den Bits 30 bis 25 des Signals inst überein. Die darauf folgenden fünf Bits des Registers imm werden den Bits elf bis sieben zugewiesen.

```
1 always @(*) begin
2     case (imm_type)
3         `IMM_I : imm = { {21{inst[31]}}, inst[30:25], inst[24:21],
    inst[20] };
4         `IMM_S : imm = { {21{inst[31]}}, inst[30:25], inst[11:8], inst[7]
    };
5         `IMM_U : imm = { inst[31], inst[30:20], inst[19:12], 12'b0 };
6         `IMM_J : imm = { {12{inst[31]}}, inst[19:12], inst[20],
    inst[30:25], inst[24:21], 1'b0 };
7     endcase
8 end
```

Auflistung 3.6: Immidiates Konkatenation

Bei einem Wert von zwei des Registers imm_type, liegt das U-Befehlsformat vor. In diesem Fall entsprechen die höherwertigen 20 Bits des Registers imm den höherwertigen 20 Bits des Signals

inst. Die restlichen Bits de Registers imm werden auf null gesetzt. Das J-Befehlsformat liegt vor, wenn der Wert von imm_type drei ist. In diesem Fall werden die höherwertigen zwölf Bits des Registers imm auf den Wert des MSB des Signals inst gesetzt. Die folgenden acht Bits des Registers imm entsprechen den Bits 19 bis 12 des Signals inst. Bit zwölf entspricht Bit 20, Bit elf bis zwei entsprechen Bit 30 bis 21 des Signals inst. Das niederwertigste Bit wird auf null gesetzt.

3.1.7 Allzweckregister

Der Allzweckregistersatz wird durch das Modul raifes_regfile implementiert. Das Modul verfügt über neun Ein- und drei Ausgänge. Eine Modulübersicht ist in Abbildung D.11 gegeben. Die Anzahl der Register wird durch die Befehlssatzarchitektur bestimmt. Bei einer Basisbefehlssatzarchitektur ohne Erweiterung liegt die Anzahl bei 32. Liegt die Befehlssatzerweiterung E vor, so beläuft sich die Anzahl der Register auf 16. Die Registerinhalte werden im Normalbetrieb über die Pipeline bestimmt, können aber auch über den Debug-Zugriff gelesen und beschrieben werden. Bei einem Lesezugriff im Normalbetrieb geben die Signale ra1 und ra2 die Adresse der Register an, aus denen die Datenwörter in die Ausgangssignale rd1 und rd2 geschrieben werden.

```
1 assign rd1 = |ra1 ? data[ra1] : 0;
2 assign rd2 = |ra2 ? data[ra2] : 0;
3 assign dm_rd = |dm_wara ? data[dm_wara] : 0;
4
5 always @(posedge clk) begin
6 if(dm_wen) begin
7     data[dm_wara] <= dm_wd;
8 end else if(wen) begin
9  data[wa] <= wd;
10 end
11 end
```

Auflistung 3.7: Registerzugriff (raifes_regfile.v Z. 29)

Dabei entsprechen ra1 und ra2 den Bits 19 bis 15 bzw. 24 bis 20 des aktuellen Instruktions-Codes. Für einen Schreibzugriff im Normalbetrieb wird das Signal wen auf den Wert eins gesetzt, wobei das Signal wa die Adresse des Zielregisters und wd das zuschreibende Datenwort angibt. Bei einem Debug-Lesezugriff wird über das Signal dm_wara die Registeradresse angegeben, und das Datenwort aus dem angegebenen Register wird in das Ausgangssignal dm_rd geschrieben. Bei einem Schreibzugriff wird ebenfalls über das Signal dm_wara die Registeradresse angegeben. Dabei muss das Signal dm_wen auf eins gesetzt sein, damit das Datenwort aus dm_wd in das Zielregister geschrieben wird.

3.1.8 Rechenwerk

Das Rechenwerk setzt sich aus den Modulen raifes_src_a_mux, raifes_src_b_mux und raifes_alu zusammen. Der erste Operand des Rechenwerks wird durch den vorgeschalteten Multiplexer raifes_src_a_mux bestimmt. Dieses Modul besitzt drei Eingänge und einen Ausgang, welche in Tabelle 3.2 beschrieben werden. Durch eine Case-Anweisung wird entsprechend des Wertes in Signal src_a_sel, entweder das Datenwort aus PC_DX oder rs1_data in das Register alu_src_a geschrieben. Initial liegt der Wert des Registers alu_src_a bei null.

Tabelle 3.2: Rechenwerk Eingang A Signalübersicht

Signalname	**I/O**	**Bits**	**Beschreibung**	**Verbindung**
PC_DX	I	32	Aktueller Programmzähler	raifes_PC_mux
rs1_data	I	32	Datenwort aus Pipeline	raifes_regfile, raifes_alu, raifes_mul_div, raifes_csr_file
src_a_sel	I	2	Adressregister	raifes_ctrl
alu_src_a	O	32	ALU Eingang 1	raifes_alu

Hat das Signal src_a_sel den Wert null, wird das Datenwort aus Signal rs1_data in das Register alu_src_a geschrieben. Bei einem Wert von eins im Register src_a_sel wird das Datenwort aus PC_DX und bei einem Wert von null das Datenwort aus rs1_data in das Zielregister geschrieben. Der zweite Operand des Rechenwerks wird durch den anderen vorgeschalteten Multiplexer raifes_src_b_mux bestimmt. Dieses Modul besitzt ebenfalls drei Eingänge und einen Ausgang, welche in Tabelle 3.3 beschrieben werden. Analog zum Modul raifes_src_a_mux ist die Zuweisung des Wertes für Register alu_src_b vom Wert in src_b_sel abhängig. Initial liegt der Wert des Registers alu_src_b bei null.

Tabelle 3.3: Rechenwerk Eingang B Signalübersicht

Signalname	**I/O**	**Bits**	**Beschreibung**	**Verbindung**
imm	I	32	Konstante	raifes_imm_gen
rs2_data	I	32	Datenwort aus Pipeline	raifes_regfile, raifes_alu, raifes_mul_div, raifes_csr_file
src_b_sel	I	2	Adressregister	raifes_ctrl
alu_src_b	O	32	ALU Eingang 1	raifes_alu

Hat das src_b_sel den Wert null, wird das Datenwort aus rs2_data in das Register alu_src_b geschrieben. Bei einem Wert von eins im src_b_sel wird das Datenwort aus Register imm in das Zielregister geschrieben. Liegt ein Wert von zwei vor, so wird ein Wert von 0x4 in das Zielregister geschrieben.

ALU

Das Rechenwerk raifes_alu ist zu den folgenden 14 verschiedenen arithmetisch-logischen Operationen fähig: Addition mit Vorzeichen, Bitweise Verschiebung links und rechts, Bitweise exklusiv Oder, Bitweise Oder, Bitweise Und, Vergleich gleich und ungleich, Subtraktion, vorzeichenbehaftete rechte Schiebeoperation, Vergleich größer gleich vorzeichenbehaftet und nicht vorzeichenbehaftet und Vergleich kleiner gleich vorzeichenbehaftet und nicht vorzeichenbehaftet. Das Modul besitzt drei Eingänge und einen Ausgang, welche in Tabelle 3.4 gelistet sind.

Tabelle 3.4: Rechenwerk

Signalname	I/O	Bits	Beschreibung	Verbindung
op	I	4	Operations-Code	raifes_ctrl
in1	I	32	Datenwort 1	raifes_src_a_mux
in2	I	32	Datenwort 2	raifes_src_b_mux
out	O	32	Rechenergebnis	Multiplexer Pipeline

Die Operation wird durch eine Case-Anweisung mit dem Wert aus dem Signal op bestimmt. Ziel einer jeden Operation ist die Zuweisung des Registers out. Für eine Schiebeopration wird aus den niederwertigen fünf Bits aus in2 der Wert für die Verschiebung gebildet. Wird eine Vergleichsoperation ausgeführt, wird das Least Significant Bit (LSB) des Registers out bei einem wahren Ergebnis auf den Wert eins und bei einem unwahren Ergebnis auf den Wert null gesetzt, wobei in beiden Fällen die restlichen 31 Bits auf den Wert null gesetzt werden.

3.1.9 Steuerwerk

Das Steuerwerk raifes_ctrl stellt eine der komplexesten Strukturen des gesamten Prozessors dar und verfügt über zahlreiche Ein- und Ausgangssignale, die in Abbildung D.12 dargestellt sind. Die Kernaufgaben liegen in der Dekodierung des aktuellen Instruktions-Codes und in der Verarbeitung von Exceptions (*Störungen*) und Interrupts (*Unterbrechung*), wobei die Ereignisse aus allen drei Pipeline Stufen parallel gehandhabt werden.

IF Stufe

Bezüglich der IF Stufe werden im Steuerwerk nur Störungsfälle und Sprungbefehle behandelt, wobei die zwei Ausgangssignale stall_IF und kill_IF auf den Wert eins gesetzt werden, falls die aktuelle Instruktion der IF Stufe gehalten oder zurückgesetzt werden soll. Für den Fall, dass stall_IF den Wert eins hat, wird kill_IF ebenfalls auf den Wert eins gesetzt (siehe Auflistung 3.8). Ein zentrales Steuersignal hierbei ist interrupt_taken, das in Unterabschnitt 3.1.10 beschrieben werden.

```
assign kill_IF = stall_IF || ex_IF || ex_DX || ex_WB || redirect ||
    replay_IF || interrupt_taken;
assign stall_IF = stall_DX || ((imem_wait && !redirect) && !(ex_WB ||
    interrupt_taken));
```

Auflistung 3.8: Handling der IF Stufe (raifes_ctrl.v Z. 143)

Liegt ein Interrupt vor, so ist dieses Signal gesetzt und die IF Stufe wird unterbrochen, damit der Trap-Handler als nächstes geladen werden kann. Tritt eine Exception in einer der drei Stufen auf, wird kill_IF ebenfalls gesetzt.

DX Stufe

Der Großteil der Schaltungslogik im Steuerwerk wird für die Instruktiondecodierung in der DX Stufe und die Behandlung zugehöriger Störfälle verwendet. Die Register PC_src_sel, imm_type, src_a_sel, src_b_sel, alu_op und csr_imm_sel und die Signale dmem_size, dmem_type werden direkt aus bestimmten Bitfeldern der aktuellen Instruktion inst_DX zugewiesen (siehe raifes_ctrl.v Z. 216 - 382). Besondere Instruktionen sind Lade- und Speicherbefehle. Wird ein Speicherbefehl ausgeführt, wird das Signal dmem_en auf den Wert eins gesetzt. Bei einem Ladebefehl wird zusätzlich das Signal dmem_wen auf eins gesetzt. Die Signale imem_wait und dmem_wait geben beide an, ob aktuell ein Speicherzugriff ausgeführt wird. Dabei ist das Signal imem_wait hart auf den Wert null verdrahtet, da es nicht vorgesehen ist, dass Instruktionen in den Speicher geschrieben werden. Das Signal dmem_wait nimmt immer dann den Wert null an, wenn das MSB der Daten-Adresse den Wert null besitzt. Andernfalls entspricht der Wert von dmem_wait dem Wert von core_mem_wait (siehe Unterabschnitt 3.1.2). Durch das Signal csr_cmd wird angegeben, ob ein Lese- oder Schreibzugriff auf die Status- und Kontrollregister stattfindet. Das Signal cmp_true entspricht dem Ergebnis einer Sprungbefehlbedingung. Hat das Signal den Wert eins, ist die Sprungbedingung erfüllt und das Register PC_src_sel wird auf den Wert eins gesetzt (siehe Auflistung 3.9).

```
always @(*) begin
    if (exception || interrupt_taken) begin
    PC_src_sel = `PC_HANDLER;
    end else if (replay_IF || (stall_IF && !imem_wait)) begin
    PC_src_sel = `PC_REPLAY;
    end else if (eret) begin
    PC_src_sel = `PC_EPC;
    end else if (dret) begin
    PC_src_sel = `PC_DPC;
    end else if (branch_taken) begin
    PC_src_sel = `PC_BRANCH_TARGET;
```

```
12      end else if (jal) begin
13      PC_src_sel = `PC_JAL_TARGET;
14      end else if (jalr) begin
15      PC_src_sel = `PC_JALR_TARGET;
16      end else begin
17      PC_src_sel = `PC_PLUS_FOUR;
18      end
19  end
```

Auflistung 3.9: Handling des Programmzähler (raifes_ctrl.v Z. 398)

Andernfalls wird das Register auf den Wert vier gesetzt, wodurch der aktuelle Programmzähler gehalten wird. Bei einem unbedingten Sprungbefehl wird das Signal redirect ebenfalls auf den Wert eins gesetzt. Das Signal wr_reg_WB wird auf den Wert eins gesetzt, wenn ein ecall oder ebreak Befehl vorliegt. Das Signal eret wird auf den Wert eins gesetzt, wenn die Instruktion mret vorliegt, aber das Previlegienlevel nicht der Maschinenebene entspricht. Analog dazu wird das Signal dret auf den Wert eins gesetzt, wenn eine dret Instruktion vorliegt. Beide Instruktion entsprechen der Behandlung von Traps. Liegt eine Trap vor, so wird die DX Stufe ebenfalls unterbrochen und das Signal kill_DX gesetzt (siehe Auflistung 3.10).

```
1 assign kill_DX = stall_DX || ex_DX || ex_WB || interrupt_taken;
```

Auflistung 3.10: Handling der DX Stufe (raifes_ctrl.v Z. 178)

Das Signal illegal_csr_access gibt an, ob ein ungültiger Zugriff auf die Status- und Kontrollregister vorliegt. Bei einem Wert von eins liegt ein ungültiger Zugriff vor und das Ausgangssignal kill_DX wird ebenfalls auf den Wert eins gesetzt. Über das Signal interrupt_pending wird angegeben, ob im nächsten Takt ein Interrupt erfolgt oder nicht. Das Signal interrupt_taken setzt einerseits ebenfalls die Signale kill_IF und kill_DX. Andererseits wird gleichzeitig das Register PC_src_sel auf den Wert fünf gesetzt. Dies hat zu Folge, dass der Programmzähler auf die von den Status- und Kontrollregistern vorgegebene Adresse zeigt.
Bei einer Single-Step Debug-Anfrage hat das Signal stepmode einen Wert von eins (dargestellt in Auflistung 3.11). Ist dies der Fall, so hat das Signal exception_code_WB den Wert drei, wodurch ein Breakpoint signalisiert wird und der Programmzähler an die Debugadresse null springt.

```
1 always @(*) begin
2     ex_code_WB = prev_ex_code_WB;
3     if (!had_ex_WB) begin
4         if (dmem_access_exception) begin
5         ex_code_WB = wr_reg_unkilled_WB ? `MCAUSE_LOAD_ADDR_MISALIGNED :
    `MCAUSE_STORE_AMO_ADDR_MISALIGNED;
```

```
6           end else
7       if (stepmode) begin
8       ex_code_WB = `MCAUSE_BREAKPOINT;
9       end
10     end
11 end
```

Auflistung 3.11: Handling der DX Stufe Stepmode (raifes_ctrl.v Z. 458)

Das Signal dmode_WB gibt an, ob sich der Prozessor im Debug-Modus befindet. Weitere Steuersignale sind die Signale bypass_rs1 bzw bypass_rs2. Bei einem Wert von eins wird direkt auf das Datenwort des angegebenen Allzweckregister zugegriffen und der Programmzähler gesetzt.

WB Stufe

Die Steuerung der letzten Pipeline Stufe umfasst die Signale bzw. Register stall_WB, kill_WB, reg_to_wr_WB, wb_src_sel_WB und exception_WB. Die ersten beiden Signale bewirken analog zu den Signalen für die anderen beiden Pipeline Stufen den Stillstand und die Beendung der Stufe WB. Das Register reg_to_wr_WB gibt die Allzweckregisteradresse an. Diese wird direkt aus der aktuellen Instruktion entnommen. Die Adresse ist zulässig, wenn in der DX Stufe keine Unterbrechung oder Ausnahme auftreten. Bei Störungen, welche die WB Stufe betreffen, wird das Signal exception_WB auf den Wert eins gesetzt.

3.1.10 Status- und Kontrollregister

Die CSR (*Status- und Kontrollregister*) verarbeiten die Systembefehle und werden durch das Modul raifes_csr_file implementiert. Eine Signal- und Registerübersicht für dieses Modul ist in Abbildung D.13 gegeben. Der potentielle Adressraum liegt bei 0x000 bis 0xFFF und bietet somit Raum für 4096 Register. Dabei ist ein Register entweder nur lesbar oder auch schreibbar. Ein Zugriffsversuch auf nicht definierte oder nicht freigegebene Register oder ein Schreibversuch auf ein Leseregister erzeugt eine Unterbrechung, die an den Trap-Handler übergeben wird.

Debugmodus und Traps

Wie bereits erläutert, wird über das MSB des 32-Bit Datenworts des Debuggers signalisiert, ob das Signal debug_haltreq auf den Wert eins gesetzt und der Prozessorkern in den Debugmodus übergehen soll. Wenn sich der Prozessor nicht bereits im Debug-Modus befindet, wird das Signal dinterrupt, welches ein zentrales Steuersignal ist, auf den Wert eins gesetzt (dargestellt in Auflistung 3.12).

```
1 assign masked_interrupt = ext_interrupts & {20{mip[11]}};
2 assign minterrupt = interrupt_pending && ie;
3 assign dinterrupt = (debug_haltreq && !dmode) || (stepmode && stepdone_r);
4 assign interrupt_pending = |mip || dinterrupt;
```

Auflistung 3.12: Setzen der Debug- und Interruptsignale (raifes_csr.v Z. 207)

So wird zum einen das Ausgangssignal interrupt_pending auf den Wert eins gesetzt, wenn ein Bit aus dem Register mip ([10], Abs. 3.1.9) oder dinterrupt den Wert eins besitzt. Der Debug-Interrupt wird auch für beide implementierten Betriebsmodi akzeptiert, sodass das Register interrupt_taken auf den Wert eins gesetzt wird. Desweiteren wird das Register handler_PC auf null gesetzt, wodurch auf die Debug Startadresse 0x0 des Debug-ROMs gezeigt wird. Weiterhin werden die niederwertigsten zwei Bits und die Bits sechs bis acht des Debug-Mode Registers dcsr auf den Wert des aktuellen Betriebsmodus bzw. auf den Wert drei gesetzt, wodurch angegeben wird, dass der Prozessorkern gestoppt werden soll. Das Register dmode gibt an, ob sich der Prozessorkern im Debug Modus befindet. Dieses Register wird ebenfalls auf den Wert eins gesetzt, wobei das Ausgangssignal dmode_WB dem Wert des Registers dmode entspricht.
In gleicher Weise wird eine Ausnahme behandelt, die durch das Signal exception repräsentiert wird und ebenfalls ein zentrales Steuersignal darstellt. Dabei werden die Bits sechs bis acht des Registers dcsr auf den Wert zwei gesetzt. Dies entspricht einem Breakpoint beim Debuggen. Desweiteren wird das Datenwort aus dem Register exception_PC in das Register mepc geschrieben ([10], Abs. 3.1.15), wobei die zwei niederwertigsten Bits auf den Wert null gesetzt werden. Auf diese Weise wird die Adresse der Instruktion, welche die Ausnahme ausgelöst hat, zwischen gespeichert. Der Wert des Registers epc entspricht aktuell noch dem Wert des Registers mepc. Über das Register exception_code wird angegeben, um welche Unterbrechung es sich handelt. Aktuell sind allerdings nur die Ursachen für eine nicht korrekt ausgerichtete Instruktionsspeicherung (Instruction address misaligned) oder für einen Breakpoint implementiert. Die Interrupt-Handhabung wird in Unterabschnitt 3.2.5 näher analysiert.

Zugriff auf CSRs

Das Signal addr stellt die Adresse des Zielregisters dar und entspricht den Bits 31 bis 20 der aktuellen Instruktion. Dabei geben die beiden höherwertigen Bits mit einem Wert von 0x3 an, ob das Zielregister ein Nur-Leseregister ist. Das Signal cmd entspricht dem Zugriffsbefehl auf die Status- und Kontrollregister. Ist das MSB auf den Wert eins gesetzt, liegt ein Lesezugriff vor (siehe Auflistung 3.13). Ist einer der beiden niederwertigen Bits auf den Wert eins gesetzt, so liegt ein Schreibzugriff vor. Das Hauptdatenwort wdata entspricht den Daten, die in das durch das Signal addr adressierte Register geschrieben werden sollen. Wdata wird entweder direkt aus der aktuellen Instruktion entnommen oder entspricht dem Datenwort aus aus einem Allzweckregister.

```
1 assign system_en = cmd[2];
2 assign system_wen = cmd[1] || cmd[0];
```

Auflistung 3.13: CSR Zugriff (raifes_csr.v Z. 160)

Analog dazu wird das Datenwort aus dem Zielregister in das Register rdata geschrieben. Liegt eine Ausnahme im Schritt-für-Schritt Debug Modus vor, so wird das Datenwort aus dem Register exception_load_addr in das register rdata geschrieben. Liegt ein Zugriff auf ein undefiniertes Register oder ein Schreibzugriff auf ein Leseregister vor, so wird das Signal illegal_access auf den Wert eins gesetzt. Liegt ein fehlerhafter Schreibzugriff über den Debug Modus vor, wird das Signal illegal_access_debug auf den Wert eins gesetzt. Durch das Signal stepmode wird angegeben, ob sich der Prozessorkern im Schritt-für-Schritt Debug Modus befindet. Stepmode wird gesetzt, wenn Bit zwei aus dcsr gesetzt ist und dmode den Wert null hat.
Der aktuelle Betriebsmodus des Prozessors entspricht dem Wert des Signals prv. Initial startet der Prozessorkern im Betriebsmodus M und wird nur geändert, wenn das Signal dret den Wert eins hat. Die Interaktion mit dem Debugger wird mit Hilfe der Signale dm_csr_addr, dm_csr_cmd, dm_csr_wdata und dm_csr_rdata durchgeführt. Die Schaltlogik dieser Signale entspricht der Schaltlogik der Signale addr, cmd, wdata und rdata. Der Programmzähler für den Debug Modus wird durch den Wert in Register dpc angegeben.

3.2 Integration des ML IP-Kerns Caeco

Der Caeco ist ein ML IP-Kern, welcher einen EKG Datensatz eines Patienten entgegen nimmt und auswertet. Das Klassifikationsergebnis, welches durch das Ergebnisregister ausgegeben wird, soll darüber Auskunft geben, ob der EKG Datensatz Vorhofflimmern Symptome aufweist und damit ein Indiz für eine Herzerkrankung des Patienten darstellt. Die Beschreibung des RTL Entwurfs des Caecos ist dabei kein Bestandteil dieser Arbeit, sondern allein die Integration des Beschleunigers in den RISC-V Kern. Das Schreiben und Lesen der Daten kann entweder über dezidierte Schreibbefehle oder über einen direkten JTAG Zugriff stattfinden. Für die Einbindung des Caecos in das schon bestehende RISC-V Design wird in dieser Arbeit eine Schnittstelle das Caecointerface (siehe Unterabschnitt 3.2.1) entwickelt, welche die Ansteuerung des Caecos übernimmt. Dazu sind Erweiterungen im Prozessorkern notwendig, auf die in 3.2.2 eingegangen wird. Gesondert wird in 3.2.2 die DM Erweiterung für den direkten Transport von Daten über die JTAG Schnittstelle zum Caeco beschrieben. Der Caeco verfügt über sieben Eingangs- und drei Ausgangssignale. Die Ansteuerung der Eingangssignale für das Schreiben von Daten in den Caeco ist in Abbildung 3.8 dargestellt. Das Ziel ist es, die EKG Daten über das Signal DIN seriell in 16 Bit breiten Teilpacketen in die Kanäle null und eins des Caecos zu schreiben. Dabei haben die EKG Datensätze immer dieselbe Größe von 246 428 Byte Vor einer neuen Datenübertragung wird das Signal CMD für einen Takt auf den Wert eins gesetzt.

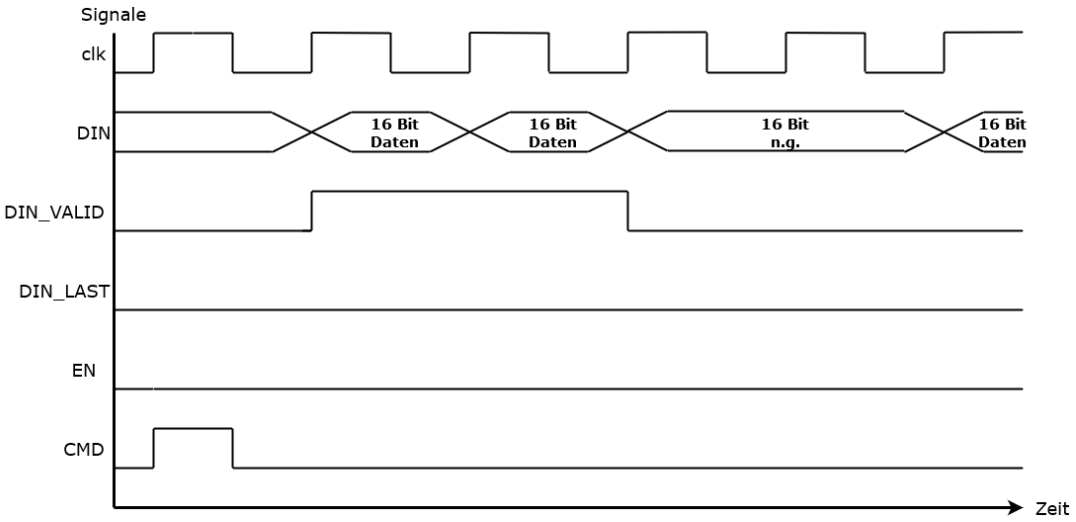

Abbildung 3.8: Caeco Schnittstelle Eingang

Danach können die Daten geschrieben werden (siehe Unterabschnitt 3.2.3 und Unterabschnitt 3.2.4). Pro Takt wird ein Datenwort gelesen. Dazu muss bei einem gültigen Datenwort das Signal DIN_VALID auf den Wert eins gesetzt werden. Während der gesamten Übertragung sind die Signale EN und DIN_LAST nicht gesetzt. Sollte ein weiterer Datensatz übertragen werden, muss zuerst das Signal CMD erneut für einen Takt auf den Wert eins gesetzt werden.

3.2.1 Caecointerface: Bus- und JTAG-Anbindung des Caecos

Das Modul caecointerface verfügt über neun Eingangs- und drei Ausgangssignale und ist in Abbildung 3.9 abgebildet. Der gesamte Verilog-Code des Moduls ist in Anhang B.1.1 und ein Blockschaltbild zur Ansteuerung und Signalübersicht in Abbildung D.14 einsehbar. Das Ausgangssignal rdata mit einer Breite von 32 Bit ist das Ergebnissregister des Caeco, das durch einen dezidierten Schreibbefehl von der Adresse 0xc00000c0 oder über einen JTAG Befehl ausgelesen werden kann. Das Ausgangssignal res_inter wird bei einem gültigen Ergebnis für einen Takt gesetzt, wodurch ein Interrupt ausgelöst wird. Über das Signal led wird signalisiert, ob sich der Caeco in einer Berechnung befindet. Aktuell wird es über das Modul fpga_wrapper als GPIO eingebunden.
Bei den Eingangssignalen wird neben dem Systemtakt clk und dem Resetsignal rst zwischen Signalen vom DM Modul und Signalen vom Prozessorkern unterschieden. Bezüglich der Eingangssignalen des Caecointerfaces ist einerseits zwischen den Prozessorsteuersignalen en, wen, wdata und addr und den DM-Steuersignalen dm_wdata, dm_wen und dm_cmd zu unterscheiden. Die Steuerung bzw. das Schreiben der Daten in den Caeco können entweder über die ersten vier Signale oder über letztere drei Signale erfolgen, allerdings nie gleichzeitig. Die Steuerungslogik ist durch die Moore Zustandsmaschine in Abbildung 3.10 dargestellt [11]. Die Zustandsmaschine startet im Zustand Idle, in welchen alle Steuersignale auf den Wert null gesetzt werden. Außerdem wird das Eingangssignal wdata in das Register data_r geschrieben. Ausgehend von dem Idle Zustand können drei verschiedenen Folgezustände angesteuert werden.

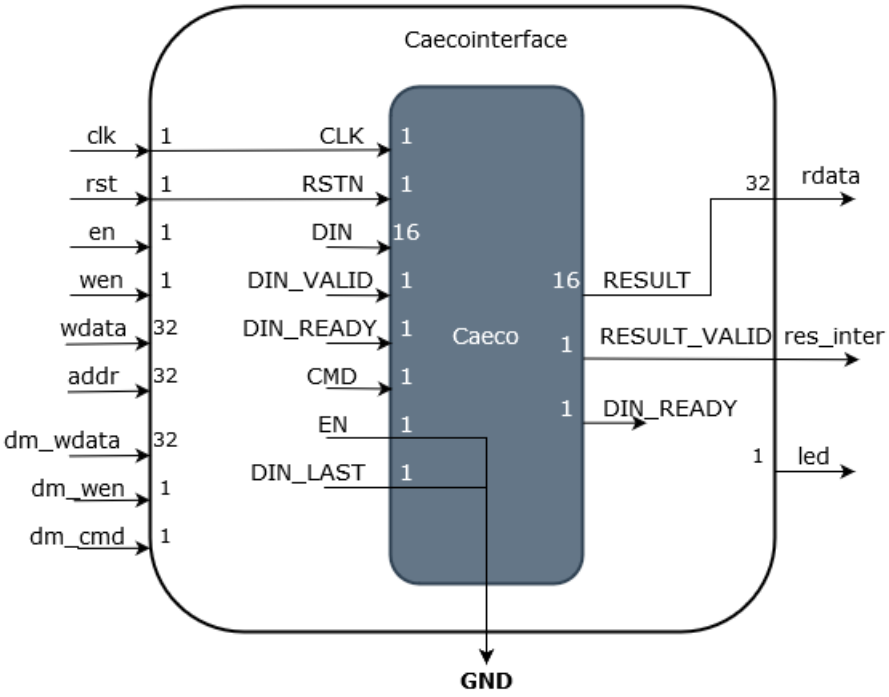

Abbildung 3.9: Caecointerface und Caeco Modulübersicht

Wird das Signal dm_cmd auf den Wert eins gesetzt, folgt der Zustand CTRL_DM. Dieser Zustand entspricht dem Setzen des Caeco Signals CMD über die JTAG Schnittstelle. Im Folgetakt wird der Zustand WAIT_DM angenommen. Dieser Zustand wird beibehalten, solange entweder das Signal dm_cmd oder dm_wen den Wert eins hat. Der Übergang vom Zustand CTRL_DM zum Zustand WAIT_DM nach einem Takt dient dazu, die Eingangssignale des Caecos CMD und IN_VALID nur für jeweils einen Takt lang zu aktivieren auch wenn die anliegen Steuersignale eine längere Signaldauer besitzen. Im Zustand WAIT_DM werden dementsprechend die Steuersignale des Caecos wieder auf den Wert null gesetzt. Außerdem wird das Datenregister data_r aktualisiert. Der Grund hierfür ist, dass in diesem Zustand ein dezidierter Schreibbefehl erfolgen kann, sodass in den Zustand WDA0 gesprungen wird, um das Datenwort zu verarbeiten. Dies ist der Fall, wenn die Signale en und wen den Wert eins haben und die Adresse dem Wert 0xc0000010 entspricht. Allerdings ist dieser Fall für den eigentlichen Betrieb nicht vorgesehen.
Im Normalfall wird zurück in den Idle Zustand gesprungen. Wie zuvor beschrieben, haben bei einem dezidierten Schreibbefehl die Signale en und wen den Wert eins und die Adresse den Wert 0xc0000010. In diesem Fall, oder wenn das Signal dm_wen den Wert eins hat, wird der Idle Zustand verlassen und der WDA0 Zustand wird für einen Takt angenommen.
In diesem Zustand findet die Datenverarbeitung statt. So wird zum einen das Signal invalid_r auf den Wert eins gesetzt und das Register datahalf_r mit den 16 Bit des Messwertes von Kanal null aus dem Datenregister data_r beschrieben. Mit dem nächsten Takt wird in den Zustand WDA1 gesprungen, in dem die anderen 16 Bit des Kanals eins in den Caeco geschrieben werden. Das letzte 32 Bit Datenwort ist nach 61 509 Zyklen geschrieben. Danach wird wieder in den Zustand WAIT_DM gesprungen.

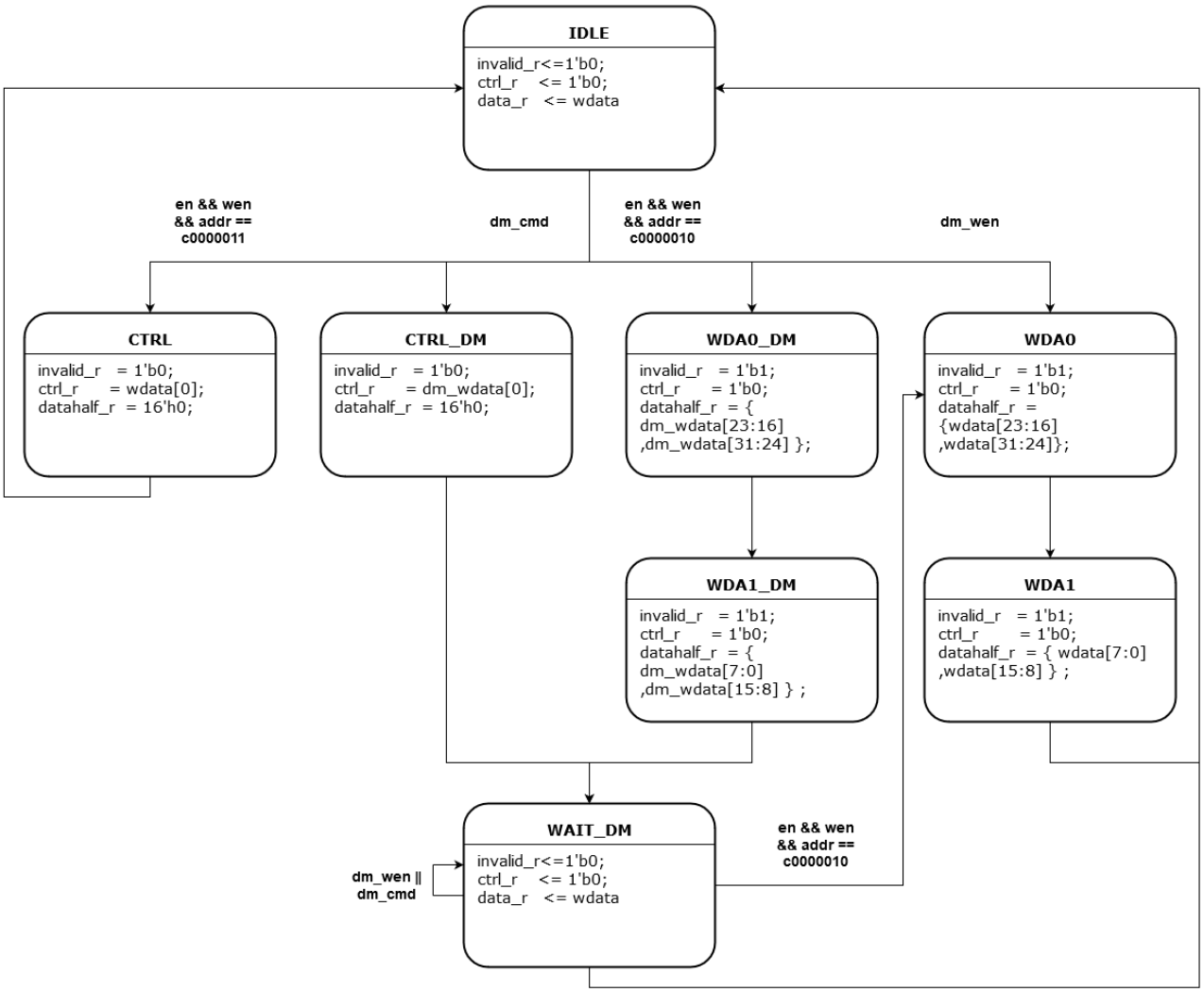

Abbildung 3.10: Caecointerface Zustandsmaschine

Wird das CMD Signal des Caecos ebenfalls durch einen dezidierten Schreibbefehl gesetzt, haben die Signale en und wen den Wert eins und die Adresse beträgt in diesem Fall 0xc0000011, sodass in den Zustand CTRL gesprungen wird. In diesem Zustand wird das Register ctrl_r auf den Wert des LSB des Datenwortes gesetzt. Danach wird wieder der Ausgangszustand Idle angenommen. Im folgenden Abschnitt sind die notwendigen Anpassungen innerhalb des Prozessors beschrieben, um den Caeco in das bestehende Design zu integrieren.

3.2.2 Adressmultiplexer

Zusätzlich zu dem Caecointerface sind Erweiterungen des Prozessorkern notwendig, um die Integration zu vervollständigen. So wird zum einen der Multiplexer, der die zu lesenden Peripheriedaten anhand der aktuellen Adresse auswählt, ergänzt. Dies ist in 3.14 dargestellt und entspricht einem Code-Ausschnitt aus dem Modul raifes_top.v. Anhand der aktuellen Adresse wird entweder das Datenwort aus dem Register per_hrdata, caeco_hrdata oder blockram_hrdata in das Signal

```
assign hrdata_muxed = (dmem_haddr_r == `GPIO_BASE_ADDR) ? per_hrdata :
   (dmem_haddr_r == `CAECO_reg) ? caeco_hrdata : blockram_hrdata;
```

Auflistung 3.14: Multiplexer für Lesen der Daten über Datenbus (raifes_top.v Z. 154)

hrdata_muxed geschrieben und bei einem Lesebefehl in ein angegebenes Allzweckregister des Prozessors geschrieben. Ein solcher Multiplexer existiert nur für den Datenbus, da die Instruktionen statisch aus dem RAM ausgelesen werden.

3.2.3 Lese- und Schreibzugriff über den Prozessor

Der Caeco kann durch dezidierte Schreib- und Ladebefehle des Prozessors entweder beschrieben oder ausgelesen werden. Hierfür sind drei Adressen reserviert, über die entweder das Signal CMD des Caecos gesetzt werden kann, die EKG Daten geschrieben werden können oder das Ergebnissregister des Caecos gelesen werden kann. Eine Übersicht ist in Tabelle 3.5 gegeben.

Tabelle 3.5: Adressraum des Caecos für dezidierte Schreib- und Lesebefehle

Adresse	Funktion
0xc0000010	EKG Daten schreiben
0xc0000011	CMD Schreiben; 0x1
0xc00000c0	Ergebnis Lesen

Für das Schreiben der EKG Daten kann der Caeco mit der Adresse 0xc000010 beschrieben werden. Die Daten werden unverändert als 32 Bit Datenworte geschrieben. Die korrekte Ausführung des Schreibvorgangs, also die Aufteilung der Kanäle und das Byte-Shifting, wird durch das Caecointerface durchgeführt. Auch das Signal CMD kann über einen Schreibbefehl gesetzt werden. Dazu dient die Adresse 0xc000011, wobei hier das zuschreibende 32 Bit Datenwort den Wert eins haben muss. Die Auslesung des Caecos ist über die Adresse 0xc0000c0 möglich. Diese Adresse bedarf einem besonderen Format, bei dem die beiden LSB nicht gesetzt sind. Grund dafür ist die in 3.15 dargestellte Ladefunktion load_data im Modul raifes_pipeline.v. Durch diese Funktion wird anhand des aktuellen Ladebefehls das Datenwort data, welches dem Wert des Registers hrdata_muxed aus 3.14 entspricht, bearbeitet und zurückgegeben.

```
function [`XPR_LEN-1:0] load_data;
  input [`XPR_LEN-1:0]                addr;
  input [`XPR_LEN-1:0]                data;
  input [`MEM_TYPE_WIDTH-1:0]         mem_type;
  reg [`XPR_LEN-1:0]                  shifted_data;
  reg [`XPR_LEN-1:0]                  b_extend;
  reg [`XPR_LEN-1:0]                  h_extend;
```

```
8   begin
9     shifted_data = data >> {addr[1:0],3'b0};
10    b_extend = {{24{shifted_data[7]}},8'b0};
11    h_extend = {{16{shifted_data[15]}},16'b0};
12    case (mem_type)
13        //gekuerzt
14    default : load_data = shifted_data;
15    endcase
16  end
17endfunction
```

Auflistung 3.15: Ladefunktion (raifes_pipeline.v Z. 81)

Wie in Zeile neun zu sehen ist, wird bei dem Standardladebefehl LW das Datenwort data anhand der zwei niederwärtigsten Bits verschoben. Dies würde bei einer Leseadresse des Caecos, die die zwei LSBs gesetzt hat, zu einem falschen Ladevorgang des Datenwortes führen.
Da das Schreiben der Daten über den Prozessor zeitaufwändig, energieintensiv und nicht kompatibel mit einem parallelen Betrieb ist, wird ein direkter Zugriff des Caecos durch das DM umgesetzt, welcher im folgenden Abschnitt beschrieben wird.

3.2.4 Direkter Lese- und Schreibzugriff über JTAG und das DM

Wie bereits in Abschnitt 3.1.4 beschrieben, ist für die Ausführung der JTAG Befehle das DM zuständig, in dem sowohl die RISC-V DMI- als auch die DM Funktionalität abgedeckt ist. Der direkte Zugriff auf den Caeco über das DM erfolgt über spezielle JTAG Befehle, die allerdings nur in die DMI Logik eingreifen, da für den Caeco keine Debug-Funktion vorgesehen ist. Dazu werden die DMI Zustandsmaschinen mit caeco-spezifischen Zuständen erweitert, dargestellt in Tabelle 3.6.

Tabelle 3.6: DMI Adressen des Caecos für direkten Schreib- und Lesezugriff

Adresse	Funktion
0x22	EKG Daten schreiben
0x23	CMD Schreiben
0x24	Ergebnis Lesen

Mit der DMI Adresse 0x22 wird das 32 Bit Datenwort des DTM DMI Registers über das Caecointerface in den Caeco geschrieben. Über die Adresse 0x23 kann das CMD Signal des Caecos gesetzt werden, wobei das eigentliche Datenwort einen beliebigen Wert haben kann. Zusätzlich kann das Ergebnissregister über die Adresse 0x24 ausgelesen werden. Die Implementierung des Schreib- und Lesezugriffs im Verilog-Code ist in Auflistung 3.16 bzw. in Auflistung 3.17 komprimiert

dargestellt. Bei einem Schreibbefehl über die JTAG Schnittstelle mit der DMI Adresse 0x22 wird die Zustandsmaschine in den Zustand ‘DMI_ADDR_CAECOWDATA gesetzt. Dabei wird das zu übertragende Datenwort aus dem Register wdata_r dem Signal caeco_wdata_r bzw. dm_wdata zugewiesen.

```
always @(posedge clk)
begin
 if(reset) begin
    // gekuerzt
  if(dmi_state == `DMI_STATE_WRITE) begin
   case (dmi_addr_r)
                // gekuerzt
    `DMI_ADDR_CAECOWDATA : begin caeco_wdata_r <= wdata_r; caeco_wen_r <= 1'b1; end
    `DMI_ADDR_CAECOCMD : begin caeco_wdata_r <= wdata_r; caeco_cmd_r <= 1'b1; end
   endcase
  end
    // gekuerzt
end
```

Auflistung 3.16: DMI Multiplexererweiterung für Schreiben des Caecos (raifes_debug_module.v Z. 413)

Außerdem wird das Signal caeco_cmd auf den Wert eins gesetzt, welches dem Signal dm_wen entspricht. Wird über das DTM die Adresse 0x23, welche ‘DMI_ADDR_CAECO-CMD entspricht, angesprochen, wird wieder das Zieldatenwort über caeco_wdata übertragen, welches bei diesem Befehl immer den Wert eins hat. Zeitgleich wird das Signal Signal caeco_cmd, welches dem Signal dm_cmd entspricht, auf den Wert eins gesetzt.
Für die Ausführbarkeit eines direkten Lesebefehls über die JTAG Schnittstelle wird die Multiplexerschaltung in Auflistung 3.17 ebenso mit den caeco-spezifischen Adressen erweitert. So wird bei einer Adressierung mit der DMI-Adresse vom Wert 0x24, welche ‘DMI_ADDR_CAECORDATA entspricht, das Ergebnisseregister des Caecointerface Moduls direkt ausgelesen.

```
always @*
begin
 rdata_r = 32'hdeaddead;
 case(dmi_addr_r)
        ...
  `DMI_ADDR_CAECOWDATA       : begin rdata_r = `XPR_LEN'hdeadbeef; end
  `DMI_ADDR_CAECOCMD         : begin rdata_r = `XPR_LEN'hdeadbeef; end
```

```
8   `DMI_ADDR_CAECORDATA       : begin rdata_r = caeco_rdata; end
9  endcase
10 end
```

Auflistung 3.17: DMI Multiplexererweiterung für Lesen des Caecos (raifes_debug_module.v Z. 257)

Die in diesem Abschnitt beschriebenen Anpassungen des bestehenden Designs entsprechen nicht mehr den RISC-V Spezifikationen. Für die Analyse und die Verifizierung des Prozessors ist daher in Bezug auf die Implementation des Caecos keine Verifizierung durch eine Referenzmodell möglich, sondern kann nur anhand von einer Simulation geprüft werden.

3.2.5 Trap-Auslösung durch Caeco

Unter RISC-V werden Interrupts und Exceptions (*Ausnahmen*) gemeinsam als Traps (*Fallen*) behandelt und sind daher auch bei der Implementierung eng verknüpft. Folgend wird allerdings nur die Implementierung bzgl. der Interrupts beschrieben, da bei einem gültigen Ergebnis des Caecos ein Hardware-Interrupt ausgelöst wird.
Für externe bzw. zusätzliche Interrupt-Quellen liegt der 24 bit breiter Eingangsport ext_interrupts am Modul raifes_crs_file.v vor. Dieser Port war vor der Caeco Implementierung hart auf null verdrahtet, wurde im Modul raifes_top.v allerdings so angepasst, dass das Signal caeco_interrupt dem LSB des Signals ext_interrupts entspricht (dargestellt in Auflistung 3.18).

```
1 raifes_core raifes(
2     .reset(reset),
3     .clk(clk),
4     .ext_interrupts({23'h0,caeco_interrupt}),
```

Auflistung 3.18: Caeco Interrupt Signal (raifes_top.v Z. 190)

Innerhalb des Moduls raifes_csr_file.v hat das Setzen des LSB des Signals ext_interrupts unterschiedliche Konsequenzen. Zum einen wird das Register mip, welches Informationen zu ausstehenden Interrupts enthält, geschrieben (siehe Auflistung 3.19).

```
1 assign mip = {|ext_interrupts,3'b0,mtip,3'b0,msip,3'b0};
```

Auflistung 3.19: Caeco Interrupt schreibt mip (raifes_csr_file.v Z. 319)

Desweiteren wird das Signal masked_interrupt gesetzt, dargestellt in Auflistung 3.12. Dieses Signal gibt an, ob es sich um einen Interrupt handelt, der bearbeitet werden kann. Ist das Signal gesetzt, so kann der vorliegende Interrupt behandelt werden. Dadurch dass eines der Bits im Register mip

gesetzt ist, wird ebenfalls das Signal interrupt_pending gesetzt. Im Falle, dass das Signal ie für das freischalten der Interrupts gesetzt ist, wird abschließend auch das Signal minterrupt gesetzt. All diese Signale werden für die folgende Fallunterscheidung der Sprungadresse bei einem Interrupt verwendet.

Ermittlung der Sprungadresse bei einer Trap

Das Register mtvec ist ein Lese- und Schreibregister, in das die Sprungadresse zum Trap-Handler geschrieben wird. Der Trap-Handler ist ein Software-Bestandteil und enthält ebenfalls Sprungbefehle, die ausgehend von der Adresse im Register mtvec die entsprechenden Routinen aufrufen. Die Implementierung dieser Fallunterscheidung ist in im Modul raifes_crs_file.v beschrieben (siehe Auflistung 3.20).

```
1  always @*
2  begin
3      if(dinterrupt || (exception_code == `MCAUSE_BREAKPOINT)) handler_PC <=
       `DEBUG_ADDRESS;
4      else if (minterrupt) begin
5          case(masked_interrupt)
6              24'h1: handler_PC <= {mtvec[31:2],2'b00}+4;
7              default: handler_PC <= {mtvec[31:2],2'b00}+8;
8          endcase
9      end
10     else handler_PC <= {mtvec[31:2],2'b00};
11 end
```

Auflistung 3.20: Sprungadresse zum Trap-Handler (raifes_csr_file.v Z. 138)

Der Multiplexer unterscheidet aktuell vier Fälle einer auftretenden Trap. Bei jedem der Fälle wird die Sprungadresse für die nächste Instruktion in das Register handler_PC geschrieben. Zeile drei und vier beschreiben einen Debug-Zugriff. Dabei wird die Adresse 0x0 in das Register handler_PC geschrieben. Die Zuweisung bei einem Interrupt wird in den Zeilen vier bis neun beschrieben, wobei Interrupts des Caecos in Zeile sechs verarbeitet werden. Bei einem durch den Caeco ausgelösten Maschinen-Interrupt ist das Signal minterrupt gesetzt und das Register masked_interrupt wird ausgewertet. Dieses Register enthält die Informationen über den Ursprung des Interrupts. Dadurch, dass das LSB dieses Register bei einem Caeco-Interrupt gesetzt wird, erhält das Register handler_PC die um den Wert vier inkrementierte Adresse aus dem Register mtvec. In allen anderen Fällen wird die Adresse im Register mtvec um den Wert acht inkrementiert und in das Register handler_PC geschrieben. Bei einer Exception entspricht die Sprungadresse genau der Adresse aus dem Register mtvec, welche bei der Software-Initialisierung geschrieben werden kann

Ermittlung der Rücksprungadresse bei einer Trap

Bei Auftreten einer Trap muss nicht nur die Sprungadresse, sondern auch die Rücksprungadresse ermittelt und gespeichert werden. Die Rücksprungadresse hängt vom aktuellen Befehl und der Pipeline-Stufe zusammen. Sprungbefehle stellen hier eine Besonderheit dar, da sie zwei Takte zur Bearbeitung benötigen. Fällt ein Interrupt genau zwischen die zwei Takte, kommt es darauf an, in welcher Pipeline-Stufe sich der Sprungbefehl befindet und ob der Sprung genommen werden soll. Die Ermittlung der Rücksprungadresse ist in Auflistung 3.21 dargestellt.

```
always @(posedge clk) begin
    if (reset) begin
        mepc <= 'XPR_LEN'h0;
    end
    else begin
        if (interrupt_taken & ~dinterrupt) begin
         if(redirect_WB) mepc <= PC_IF;
      else mepc <= (exception_PC & {{30{1'b1}},2'b0}) + 'XPR_LEN'h4;
     end
 end
end
```

Auflistung 3.21: Rücksprungadresse zum Trap-Handler (raifes_csr_file.v Z. 331)

Bei einem Interrupt gibt es zwei Fallunterscheidungen, um die Rücksprungadresse zu speichern. Ein Interrupt liegt vor, wenn das Signal interrupt_taken und das Signal dinterrupt nicht gesetzt ist. Hierbei wird das Signal interrupt_taken gesetzt, wenn das Signal minterrupt gesetzt ist. Im Normalfall wird dann die Adresse aus dem Register exception_PC, welche die Adresse der Instruktion der WB Stufe ist, um den Wert vier inkrementiert und gespeichert, sodass die Rücksprung auf die Instruktion der IF Stufe zeigt. Liegt ein auszuführender Sprungbefehl in der WB Stufe vor, entspricht die zu speichernde Adresse dem Wert in Register PC_IF, welches die berechnete Adresse für die Instruktion der IF Stufe angibt. Hierbei wird das Signal redirect_WB immer dann gesetzt, wenn ein gültiger Sprungbefehl vorliegt.

3.2.6 Manueller Interrupt zur Verifizierung auf dem FPGA

Für die Verifizierung des Design auf dem FPGA kann der Interrupt nicht durch den Caeco ausgelöst werden, da die Datenschnittstelle aktuell nicht vorliegt. Somit wird der Interrupt des Caecos über einen Schalter auf der Entwicklungsplatine ausgelöst, bei dem das Ergibnisregister des Caecos ausgelesen wird. Hierzu sind weitere Implementierung des Design innerhalb des Moduls raifes_fpga_wrapper.v notwendig (dargestellt in Auflistung 3.22).

```
1  `define          INT_IDLE    4'h0
2  `define          INT_ON      4'h1
3  `define          INT_OFF     4'h2
4  `define          INT_WAIT    4'h3
5  wire             interrupt;
6  reg      [3:0]   state, next_state;
7  reg              interrupt_r;
8  always @(posedge CLKout or posedge RESET)
9  begin
10     if(RESET) begin
11         state<=`INT_IDLE; end
12     else begin
13         state <= next_state; end
14 end
15 always@(*)
16 begin
17     case(state)
18         `INT_IDLE: begin
19             if (ext_inter) begin
20                 next_state = `INT_ON; end
21             else begin
22                 next_state = `INT_IDLE; end
23         end
24         `INT_ON: begin next_state = `INT_WAIT; end
25         `INT_WAIT: begin if (!ext_inter) begin next_state = `INT_IDLE; end
26             else begin next_state = `INT_WAIT; end
27         end
28         default: next_state =`INT_IDLE;
29     endcase
30 end
31 always@(state)
32 begin
33     interrupt_r = 1'b0;
34     case(state)
35         `INT_IDLE: interrupt_r = 1'b0;
36         `INT_ON: interrupt_r = 1'b1;
37         `INT_WAIT: interrupt_r = 1'b0;
38     endcase
39 end
40 assign interrupt = interrupt_r;
```

Auflistung 3.22: Manueller Interrupt (raifes_fpga_wrapper.v Z. 56)

Die dargestellte Zustandsmaschine reagiert auf das Signal ext_inter, welches dem Schalter des FPGAs entspricht. Wird der Schalter betätigt, hat dieser den Wert eins, wodurch die Zustandsmaschine aus dem Anfangszustand INT_IDLE in den Zustand INT_ON springt und das Signal interrupt wird für einen Takt lang gesetzt. Dies entspricht dem Verhalten des Caecos bei einem gültigen Ergebnis. Mit dem nächsten Takt springt die Zustandsmaschine dann in den Zustand INT_WAIT. Wird der Schalter auf den Ausgangswert gesetzt, so springt die Zustandsmaschine in den Anfangszustand INT_IDLE und durch das erneute Betätigen des Schalters kann ein weiterer Interrupt ausgelöst werden. Für die Verifizierung auf dem FPGA wird das Signal interrupt an das Modul raifes_top.v weitergeleitet und ersetzt das Signal caeco_interrupt bei den externen Interrupts (dargestellt in Auflistung 3.23).

```
1 raifes_core raifes(
2     .reset(reset),
3     .clk(clk),
4     //.ext_interrupts({23'h0,caeco_interrupt}),
5     .ext_interrupts({23'h0,ext_inter}),
```

Auflistung 3.23: Manuelles Interrupt Signal (raifes_top.v Z. 190)

Abschließend ist festzuhalten, dass die Implementierung des Caecos an zahlreichen Stellen Anpassungen erfordert. Die Integration in das Gesamt-Design impliziert einen Datenaustausch des Caecos mit dem Prozessor und der JTAG Schnittstelle. Um Energie und Zeit zu sparen, wurde ein direkter Zugriff auf den Caeco über die JTAG Schnittstelle implementiert. So können die EKG Daten direkt in den Caeco geschrieben werden. Weiterhin kann der Caeco aber auch über den Prozessor durch dezidierte Lese- und Schreibbefehle erreicht werden. Dies ist für die Auswertung eines Ergebnisses im realen Betrieb notwendig. Bezüglich der Verifizierung sind zwei separate Tests notwendig, da das Beschreiben des Caecos mit Daten über JTAG Schnittstelle aktuell nicht auf dem FPGA getestet werden kann. Daher wird einerseits das Verhalten des Caecos auf RTL Ebene simuliert und verifiziert, anderseits wird das Interrupt-Verhalten des Prozessors durch einen Schalter ausgelösten Interrupt getestet. Beide Tests haben allerdings ein und denselben Programm-Code gemeinsam. Die Software-Entwicklung bzgl. des Programms ist in den folgenden Abschnitten erläutert.

4 Hard- und Software-Entwicklungsfluss

In diesem Abschnitt wird der Entwicklungsfluss der Zielhard- und Software mit den eingangs erwähnten Werkzeugen beschrieben. Der Ablauf für die beiden Entwicklungszweige ist in Abbildung 4.1 dargestellt. Das Ziel auf Seite der Software-Entwicklung ist die Erstellung eines C-Programms (Quell-Code), welches Grundfunktionen des RISC-V Prozessors und des ML-IP-Kerns prüft. Dieses Programm wird für die Simulation in riscvOVPsim, für die RTL Simulation und im Anschluss auf der eigentlichen Ziel-Hardware genutzt, sodass die Simulationsergibnisse vergleichbar sind und Verifikationsrückschlüsse bzgl. des Hardware-Designs getroffen werden können.

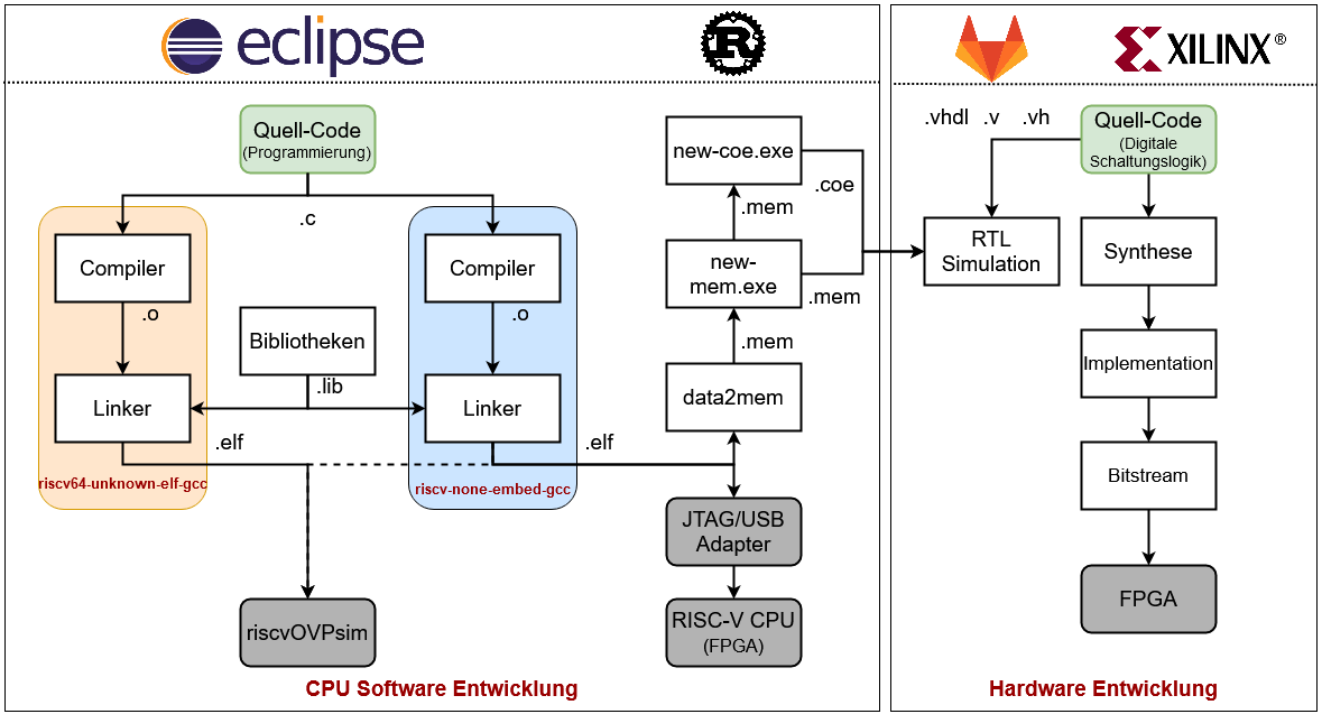

Abbildung 4.1: Zielhard- und Software-Entwicklungsablauf

Der CPU Software-Entwicklungsfluss wird eingesetzt, um die C Quell-Code-Dateien zu kompilieren und für die Simulation bzw. Debugging vorzubereiten. Für die Erstellung von den ELF Dateien werden zwei Toolchains genutzt, einmal die SiFive riscv64-unknown-elf-gcc 8.3.0-2019.08.0 Toolchain und der xPack riscv-none-embed-gcc v8.3.0 Toolchain. Letztere wird innerhalb der Entwicklungsumgebung Eclipse IDE verwendet. Die kompilierte ELF Datei der SiFive Toolchain wird für die Simulation mit dem Instruktionssimulator riscvOVPsim herangezogen. Zwar kann auch die ELF Datei der xPack Toolchain simuliert werden, allerdings werden dabei die eigens definierten system calls (*Systemaufrufe*) genutzt, die nicht das Betriebssystem, sondern die Ziel-Hardware

adressieren. Dadurch werden die Ausgaben des zum Debugging genutzten *printf()* Befehls nicht ausgegeben, wodurch eine erste Verifizierung durch den Simulator erschwert wird.
Die ELF Datei wird für das Debugging des implementierten FPGAs und der RTL Simulation verwendet. Für die Umwandlung in das richtige Format, das für die RTL Simulation benötigt wird, werden die drei Programme data2mem (Xilinx), new-mem (Rust) und new-coe (Rust) genutzt. Dabei wird die ELF Datei durch data2mem im ersten Schritt in eine Memory (MEM) Datei umgewandelt. Diese Datei entspricht allerdings nicht dem von Xilinx geforderten MEM Format, sodass im nächsten Schritt durch das selbst geschriebene Programm new-mem das Format überarbeitet wird. Zusätzlich kann diese Datei durch new-coe in eine COE Datei umgewandelt werden, mit der die Speicherbausteine unter Vivado initialisiert werden können.
Bei der Hardware-Entwicklung werden die HDL Dateien in einer RTL Simulation geprüft, um anschließend synthetisiert und für die Nexys4 DDR Entwicklungsplatine implementiert zu werden. Hierfür wird das Programm Vivado 2020.1 von Xilinx genutzt. Zusätzlich ist die gesamte Entwicklung durch Gitlab DevOps automatisiert. Dabei wird jeder in Abbildung 4.1 Schritt des Hardware-Entwicklungsflusses ausgeführt und bereits ausgewertet, wodurch eine frühzeitige Fehlererkennung möglich ist.

4.1 Ziel-Hardware

In diesem Abschnitt wird das Nexys 4 DDR Board als Ziel-Hardware für den Test des entwickelten Designs und der JTAG Adapter Olimex ARM-USB Tiny-H als Programmier- und Debugschnittstelle beschrieben. Im Unterabschnitt 4.1.1 wird dabei genauer auf die Schnittstellen und die Komponenten der Entwicklungsplatine eingegangen. Im Unterabschnitt 4.1.2 wird der JTAG Adapter und die Verbindung zum DDR4 Board erläutert. Für die Programmierung des FPGAs wird die IDE Vivado 2020.1 Entwicklungsumgebung verwendet, welche zu der Xilinx Unified Software Platform 2020.1 gehört (siehe Unterabschnitt 4.4.2). Um das implementierte Design zu Debuggen bzw. den Prozessor zu programmieren, wird die Entwicklungsumgebung Eclipse IDE genutzt (siehe Abschnitt 4.4.1).

4.1.1 Digilent Nexys 4 DDR Entwicklungsplatine

Die Entwicklungsplatine verfügt über eine Vielzahl von ansteuerbaren Schnittstellen und ist in Abbildung 4.2 dargestellt. Das zentrale Bauteil ist der Artix XC7A100T-CSG324 FPGA von Xilinx. Der generierte Takt für die Platine liegt bei 100 MHz. Für die Implementierung unter Vivado wird die spezifische XDC Datei des Boards benötigt, die über das Digilent Github Repository heruntergeladen und dem lokalen Installationsordner hinzugefügt werden kann [12].
Die Stromversorgung, die Programmierung des implementierten Designs über die Vivado Entwicklungsumgebung und die Abfrage des Host-PCs über die UART Schnittstelle ist über den Micro-USB Port J6 realisiert.

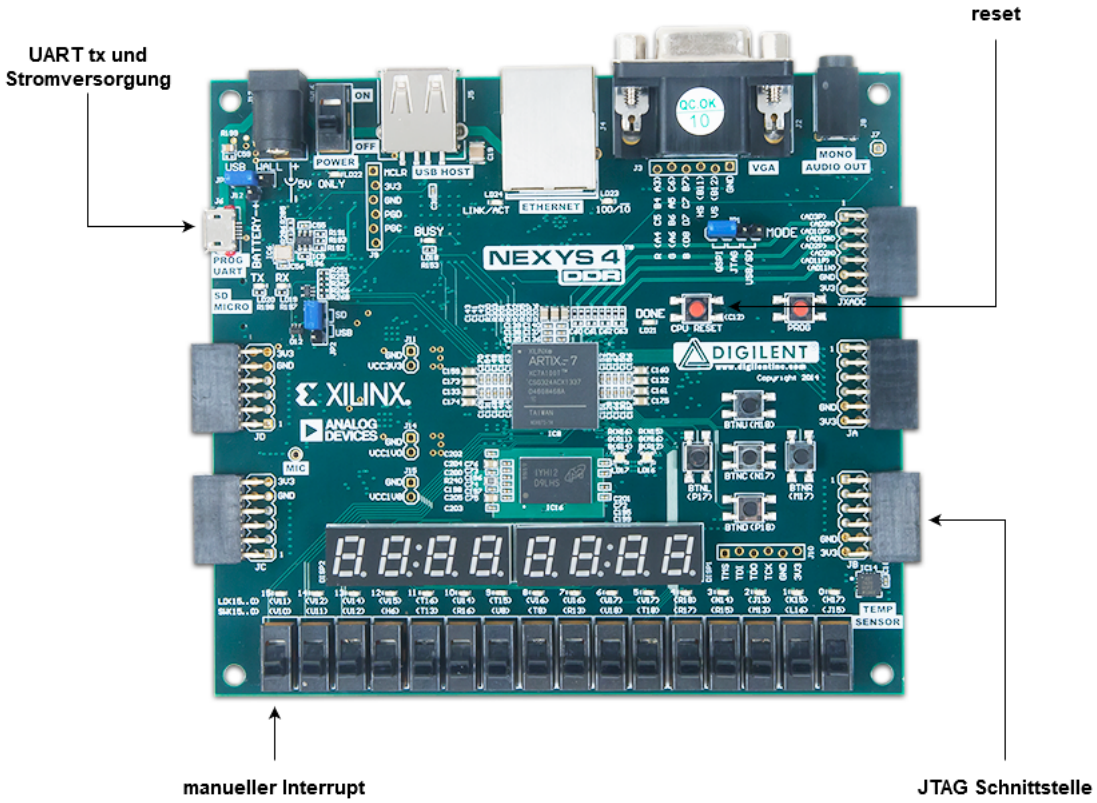

Abbildung 4.2: Digilent Nexys 4 DDR Entwicklungsplatine [13]

Der JTAG Olimex ARM-USB-Tiny-H Adapter wird über den Pmod Port JA mit dem FPGA verbunden. Ein Zurücksetzen des Prozessors kann über den CPU Reset-Knopf erfolgen. Für die Auslösung eines Interrupts wird bei der Hardware-Verifizierung der Schiebeschalter Nummer 16 genutzt. Im folgenden Abschnitt wird näher auf den JTAG Adapter und die Verbindung über die Pmod JA Schnittstelle eingegangen.

4.1.2 Olimex ARM-USB Tiny-H JTAG Adapter

Der Olimex ARM-USB Tiny-H JTAG Adapter wird in Kombination mit der OpenOCD Software für das Debugging des RISC-V Prozessors verwendet. Die Taktgeschwindigkeit für die Übertragung der Daten des Adapters kann bis zu 30 MHz betragen und wird in der späteren RTL Simulation mit 20 MHz bei einem Systemtakt von 25 MHz getestet. Damit der Adapter unter der lokalen Entwicklungsmaschine verwendet werden kann, müssen die entsprechenden Windows Treiber WinUSB(libUSB) über die Hilfs-Software Zadig installiert werden [14]. Die Verbindung zum Nexys4 DDR Board wird über die Pmod Schnittstelle JA aufgebaut. Die Pinbelegung ist in Abbildung 4.3 beschrieben. Bei der Steckerbeschaltung wird einer der beiden VCC Pins des Nexys4 mit dem Pin Vref des Olimex Adapters verbunden. Die Referenzspannung liegt hierbei bei 3,3 V. Ebenfalls wird einer der beiden GND Pins mit einem der GND des Adapters verbunden.

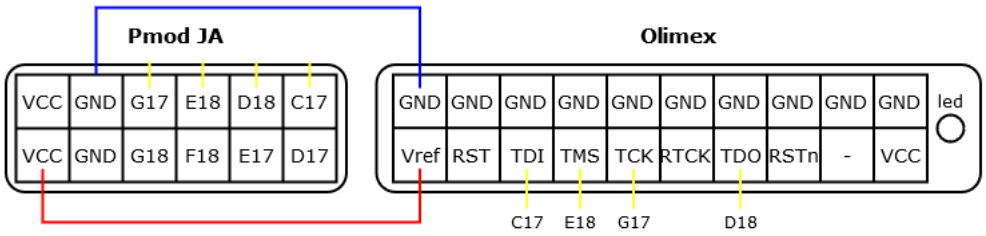

Abbildung 4.3: Hardware-Verbindung von Olimex Adapter und Nexys4 DDR

Die Signalleitung TDI wird mit dem Pin C17, TMS mit E18, TCK mit G17 und TDO mit D18 verbunden. Über USB wird der Adapter mit dem lokalen Entwicklungs-PC verbunden.

4.2 Automatisierungswerkzeuge

In diesem Abschnitt wird eine Einführung und Beschreibung der Entwicklungswerkzeuge für eine effiziente und nachhaltige Automatisierung bzgl. der Hardware-Entwicklung in der Vivado 2020.1 Integrated Development Environment (IDE) gegeben. Im Unterabschnitt 4.2.1 wird der Begriff DevOps erläutert. Dabei wird ebenfalls auf Die Begriffe Continuous Integration (CI) und Continuous Delivery (CD) eingegangen, die einen hohen Stellenwert in den DevOps besitzen. Im Abschnitt 4.2.2 wird speziell auf die DevOps Entwicklung in Gitlab eingegangen. Abschließend wird in Abschnitt 4.3 genau beschrieben, wie die DevOps für diese Arbeit aufgebaut wurde. Dabei wird auch konkret auf die Auslegung der .gitlab-ci.yml Datei eingegangen (siehe Unterabschnitt 4.3.2).

4.2.1 DevOps

Der Begriff DevOps entstammt der Software-Entwicklung und ist aus den Begriffen Development und IT Operations zusammengesetzt. Beim DevOps Ansatz geht es darum, eine Projektentwicklung, welche von mehreren Parteien durchgeführt wird, weitestgehend automatisiert, standardisiert und transparent durchzuführen und damit die Effizienz des Entwicklungsprozesses zu steigern und eine nachhaltige Qualitätssicherung zu erzielen. Es gibt zahlreiche Werkzeuge, welche in unterschiedlichen Bereichen im Zusammenhang mit DevOps verwendet werden. Darunter fallen z.B. Versionsverwaltungswerkzeuge wie Git, Testwerkzeuge wie JUnit oder auch Kontainer wie Docker. Die zentralen Elemente von DevOps sind CI und CD. Dabei umfasst das Schlagwort CI die Verwaltung des vollständigen Programm-Codes in einem Versionierungssystem. Außerdem findet bei einer Code-Aktualisierung eine automatische Integration und Verifikation statt, wobei der gesamte Prozess für jeden Anwender transparent ist. Der CD Begriff wird anschließende die automatisierte Veröffentlichung und Bereitstellung der aktuellsten und getesteten Version [15, 16].

4.2.2 Projektmanagement mit Gitlab

Für die Versionskontrolle des Quell-Codes und die Automatisierung der Verifikation der Entwürfe wird in dieser Arbeit die Plattform Gitlab genutzt. Gitlab basiert auf dem Versionsverwaltungssystem Git, bietet aber zusätzliche DevOps Funktionen, durch die die Nutzung zusätzlicher Plattformen nicht notwendig ist. Mit Hilfe der integrierten CI/CD Funktionen werden in Gitlab sogenannte Pipelines verwaltet, anhand derer die Automatisierung durchgeführt wird. Die Ausführung einer Pipeline wird durch eine .gitlab-ci.yml Datei definiert, zusammen mit dem Quell-Code im gleichen Verzeichnis abgelegt wird [17]. Der Aufbau der in dieser Arbeit verwendeten Datei wird in Unterabschnitt 4.3.2 detailliert beschrieben. Desweiteren wird die Ziel-Hardware, auf der die Pipeline ausgeführt werden soll, durch die Angabe des zu verwendeten Runners bestimmt. Ein Runner ist eine notwendige Instanz, welche auf der Ziel-Hardware installiert sein muss, um die Pipeline ausführen zu können. Es ist allerdings auch durchaus möglich, verschiedene Unterprozesse der Pipeline auf verschiedene Zielgeräte zu verteilen. Im Zuge dieser Arbeit wird eine Automatisierung auf einer lokalen Entwicklungsmaschine beschrieben. verwendet wird dabei eine 64-Bit Windows 10 Pro Maschine in der Version 1909, welche über einen Runner verfügt, der durch die Tags “local“ und “windows“ aktiviert wird.

4.3 Hardware-Entwicklung mit DevOps

Im Rahmen dieser Arbeit zielt die Hardware-Entwicklung auf die Implementierung des Design auf einem FPGA ab. Dafür werden alle HDL Quelldateien in einem Vivado 2020.1 Projekt synthetisiert, implementiert und der FPGA konfiguriert. Zusätzlich wird eine RTL Simulation durchgeführt. Um die Arbeit mit den RTL Dateien und den Entwicklungsfluss unter Vivado 2020.1 zu optimieren, wird die Hardware-Entwicklung durch die Nutzung einer Gitlabpipeline automatisiert, wobei im Vivado Project Batch Flow Modus (*Projekt-Stapel Modus*) gearbeitet wird.
Grundlage für die Durchführung einer Automatisierung in Gitlab ist ein aktives Nutzerkonto, ein Projekt, das die in Abbildung 4.4 dargestellte Projektstruktur aufweist und ein installierter bzw. zugänglicher Gitlab-Runner (siehe Abschnitt 4.3.1).

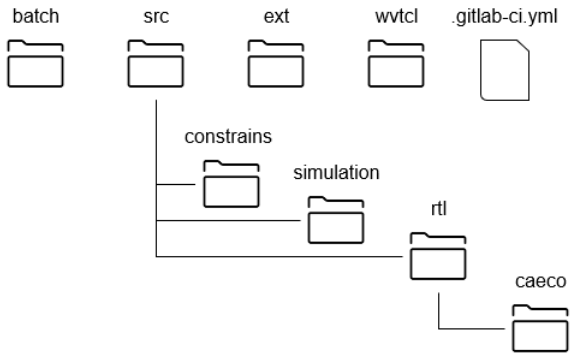

Abbildung 4.4: Gitlab Projektstruktur

Die Programmierung der Automatisierung findet sich in der .gitlab-ci.yml Datei wieder und ist in Abschnitt 4.3.2 beschrieben. Im Ordner batch sind Batchskripte enthalten, die punktuell eingesetzt werden, um den Pipelinerverlauf zu verifizieren. Der Ordner src hat vier Unterordner auf die sich alle notwendigen HDL Dateien verteilen. Im Ordner ext befinden sich die EKG Daten, sowie der Programm-Code als MEM und COE Datei. Die Vivado Batchskripte liegen im Ordner wvtcl.
Jede Aktualisierung der Quelldateien im Gitlabverzeichnis startet die Pipeline. Eine visuelle Übersicht über die Pipeline ist in Abbildung 4.5 gegeben. Es werden fünf aufeinander aufbauende Stages (*Stufen*) durchlaufen. Stages werden seriell abgearbeitet und können mehrere parallele Jobs (*Arbeitspakete*) enthalten. Ist eine Stage fehlerhaft und wird dadurch nicht erfolgreich beendet, kann an dieser Stelle die Pipeline bereits gestoppt und die Fehlersuche gestartet werden. In dieser Arbeit werden die Stages Prepare, Build, Synthesis, Implement und Simulate durchgeführt. Hierbei ist die Stage Build abhängig von der Stage Prepare, die Stages Synthesis und Simulate abhängig von der Stage Build und die Stage Implement abhängig von Synthesis.

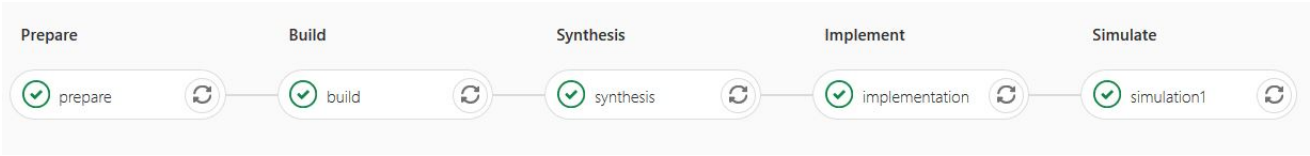

Abbildung 4.5: Gitlab Pipeline

In der Stage Prepare wird der Job prepare aufgerufen. Dieser dient der Vorbereitung des Speicherordners build für das zu erstellende Vivado Projekt und der Initialisierung der Log-Dateien. Ist dieser Schritt erfolgreich, wird in der nächsten Stage Build durch den Job build ein neues Vivado 2020.1 Projekt masterthesis erstellt. Alle neu erstellten Dateien werden als Artefakte behandelt, die für eine bestimmte Dauer auf der ausführenden Maschine unter dem Installationsverzeichnes des Gitlab-Runners und auf dem Gitlab Server gespeichert werden. Somit ändert sich lokal die Projektstruktur und wird mit den neu erstellen Dateien ergänzt, wie in Abbildung 4.6 zu sehen ist.

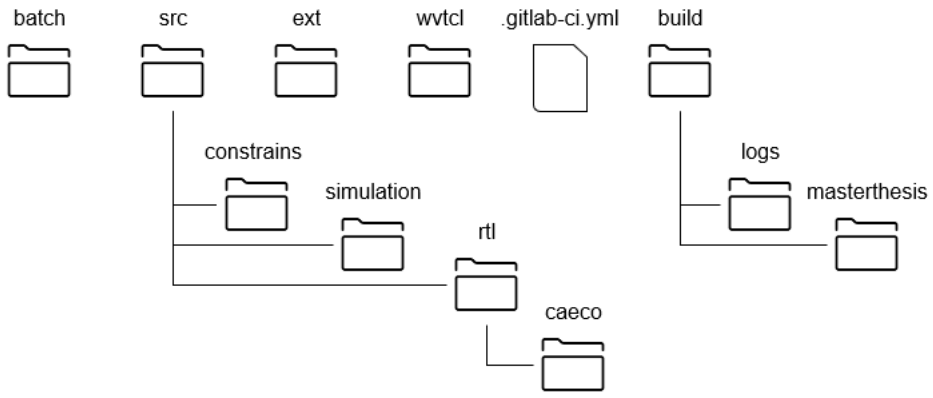

Abbildung 4.6: Gitlab Projektstruktur nach erfolgreicher Pipeline-Ausführung

Hierbei befinden sich die gespeicherten oder exportierten Log-Dateien im Ordner logs und das neu erstellte Vivadoprojekt im Ordner masterthesis. Durch die Nutzung von DevOps wird somit zum einen eine Zeitersparnis erzielt, da die Pipeline automatisiert und die aufwendigen Schritte der Projektverwaltung in Vivado automatisch abgearbeitet werden und bietet zum anderen die Möglichkeit auf Änderungen zu reagieren und Fehlerquellen frühzeitig zu erkennen. Im folgenden Abschnitt ist der Gitlab-Runner genauer beschrieben.

4.3.1 Gitlab-Runner

Der Gitlab-Runner ist ein Prozess, der den Programm-Code aus der .gitlab-ci.yml Datei auf einer Zielmaschine ausführt. Für diese Arbeit ist auf der lokalen Entwicklungsmaschine unter Windows 10 ein projektspezifischer shell Gitlab-Runner installiert [18]. Ob der installiert Runner aktiv ist und mit welchen tags (*Stichworte*) dieser genutzt werden kann, ist unter den Gitlab Projekteinstellungen unter CI/CD einsehbar (siehe Anhang C.1.5, Abbildung C.5). Der installierte Runner wird durch die tags “local“ und “windows“ angesprochen. Der Grund hierfür ist, dass die Entwicklungsplatine mit der lokalen Entwicklungsmaschine verbunden ist und so die nach der Implementierung erstellte Binärdatei zur Programmierung auch lokal zugänglich sein muss. Für Prozesse, welche die Rechenleistung einer privaten Maschine übersteigen würden, bietet Gitlab die Nutzung von shared Runners an, die teilweise auf potenteren Maschinen laufen. Allerdings muss darauf geachtet werden, dass die genutzten Programme auf der Maschine des angesprochenen Runners verfügbar sind, da ansonsten die Pipeline scheitert. Im nächsten Abschnitt wird die Programmierung eines Runners durch die .gitlab-ci.yml Datei beschrieben.

4.3.2 .gitlab-ci.yml Datei

Die .gitlab-ci.yml YAML Datei definiert die Pipelinestruktur und enthält den Programm-Code, der von einem oder mehreren Gitlab-Runner ausgeführt wird. Hauptsächlich werden bestimmte Stages und Jobs definiert, die Programm-Code ausführen und je nach Ergebnis zu einem Erfolg oder Misserfolg der Pipeline führen [19]. In Auflistung 4.1 ist die genutzte .gitlab-ci.yml Datei dargestellt. Wie bereits erwähnt, werden die fünf Stages bzw. Jobs definiert und miteinander verknüpft. Dazu werden zuerst Job-Vorlagen programmiert und diese im Anschluss abgerufen. So bleibt der Aufruf der verschiedenen Jobs übersichtlich und kann je nach Bedarfsfall angepasst werden.

```
1 stages:
2   - prepare
3   - build
4   - synthesis
5   - implement
6   - simulate
```

```
7
8 .job_template: &template_base
9   tags:
10     - local,windows
11   before_script:
12     - dir
13     - P:\Xilinx\Vivado\2020.1\.settings64-Vivado.bat
14
15 .job_template: &template_prepare
16   <<: *template_base
17   stage: prepare
18   script:
19     - dir
20     - mkdir build\logs
21     - New-Item -Path "build\logs\" -Name "project.log" -ItemType "file"
      -Value ""
22     - dir
23   artifacts:
24     when: on_success
25     name: "%CI_JOB_NAME%-%CI_JOB_STAGE%-%CI_COMMIT_REF_NAME%"
26     paths:
27       - .\build
28     expire_in: 60 minutes
29
30 .job_template: &template_build
31   <<: *template_base
32   stage: build
33   dependencies:
34     - prepare
35   script:
36     - dir
37     - vivado -mode batch -source wvtcl\create.tcl
38   artifacts:
39     when: on_success
40     name: "%CI_JOB_NAME%-%CI_JOB_STAGE%-%CI_COMMIT_REF_NAME%"
41     paths:
42       - .\build
43     expire_in: 60 minutes
44
45 .job_template: &template_synthesis
46   <<: *template_base
47   stage: synthesis
```

```
48   dependencies:
49     - build
50   script:
51     - vivado -mode batch -source wvtcl\synthesis.tcl
52     - batch\check.bat
53   after_script:
54       - copy build\masterthesis\masterthesis.runs\synth_1\runme.log
      build\logs\project_synthesis.log
55   artifacts:
56     when: on_success
57     name: "%CI_JOB_NAME%-%CI_JOB_STAGE%-%CI_COMMIT_REF_NAME%"
58     paths:
59       - .\build
60     expire_in: 1 day #5 minutes
61
62 .job_template: &template_implementation
63   <<: *template_base
64   stage: implement
65   dependencies:
66     - synthesis
67   script:
68     - vivado -mode batch -source wvtcl\implementation.tcl
69     - batch\check.bat
70   after_script:
71       - copy build\masterthesis\masterthesis.runs\impl_1\runme.log
      build\logs\project_implementation.log
72   artifacts:
73     when: on_success
74     name: "%CI_JOB_NAME%-%CI_JOB_STAGE%-%CI_COMMIT_REF_NAME%"
75     paths:
76       - .\build
77     expire_in: 15 minutes
78
79 .job_template: &template_simulation1
80   <<: *template_base
81   stage: simulate
82   dependencies:
83     - build
84   script:
85     - vivado -mode batch -source wvtcl\simulation.tcl
86   artifacts:
87     when: on_success
```

```
88      name: "%CI_JOB_NAME%-%CI_JOB_STAGE%-%CI_COMMIT_REF_NAME%"
89      paths:
90        - .\build
91      expire_in: 15 minutes
92
93  # Jobs
94  prepare:
95    <<: *template_prepare
96  build:
97    <<: *template_build
98  synthesis:
99    <<: *template_synthesis
100 implementation:
101   <<: *template_implementation
102 simulation1:
103   <<: *template_simulation1
```

Auflistung 4.1: .gitlab-ci.yml

In den Zeilen eins bis sechs werden die Stages definiert, welche zu einem späteren Zeitpunkt ausgewählt werden können. Danach wird eine Basisvorlage definiert, die in jedem der folgenden Jobs genutzt wird. Durch die Auswahl der tags in Zeile zehn wird der lokale Runner aufgerufen und in den Zeilen 11 bis 13 wird die Vivadokonfiguration geladen. Letzteres ist notwendig, da ansonsten der Start von Vivado durch den Gitlab-Runner nicht möglich ist. Da der Aufruf durch einen before_script Befehl stattfindet, wird die Pipeline bei einem fehlerhaften Aufruf fehlschlagen. Von Zeile 15 bis 28 wird die Jobvorlage für den ersten Job, der dann in Zeile 94 aufgerufen wird, definiert. Die Basisvorlage wird, wie bei allen anderen Jobs auch, zuerst aufgerufen. In Zeile 17 wird die Vorlage der Stage Prepare zugeordnet. Darauf folgend werden mit dem Aufruf script die Systembefehle definiert, bei denen die beiden Zielordner logs und build angelegt werden. Außerdem wird die Datei project.log erstellt, die dazu dient, spezielle Nachrichten zu speichern. Der erstellte Ordner build wird rekursiv durch die Zeilen 23 bis 28 als Artefakt definiert, sodass die Daten für die folgenden Jobs zugänglich sind und gespeichert werden können.
Mit den Zeilen 30 bis 43 erfolgt die dritte Jobdefinition, die der vorherigen ähnelt. Hier wird in Zeile 37 allerdings das erste mal Vivado im Batchmodus aufgerufen und das Batchskript create.tcl abgearbeitet (siehe Anhang A.2.1). Durch diesen Job werden alle HDL Quelldateien und die Constraints Datei dem Projekt hinzugefügt. Außerdem wird der Takt durch den Clock-Wizard und der Blockram durch den Block-RAM-Generator von Vivado generiert.
Die anderen Jobvorlagen folgen dem gleichen Format und werden demnach nicht im Einzelnen erläutert. Die Jobs werden in den Zeilen 93 bis 103 aufgerufen. Das Design wird nach dem Job build synthetisiert und danach implementiert. Am Ende erfolgt die Simulation des Designs.

4.4 Software-Werkzeuge

In diesem Abschnitt werden die notwendigen Software-Werkzeuge zur Entwicklung des Programm-Codes und der HDL Dateien beschrieben. Die Software-Entwicklung wird mit Hilfe der Entwicklungsumgebung Eclipse 2019-09 IDE durchgeführt. Die IDE benötigt allerdings für eine erfolgreiche Entwicklung angepasste RISC-V Erweiterungen, die in Unterabschnitt 4.4.1 beschrieben werden. Für die Hardware-Entwicklung werden Xilinxwerkzeuge genutzt, die gebündelt über die Xilinx Unified Software Platform 2020.1 installiert werden und in Abschnitt 4.4.2 erläutert werden.

4.4.1 GNU MCU Eclipse IDE v4.6.1

Auf dem lokalen Entwicklungs-PC ist die Entwicklungsumgebung Eclipse 2019-09 IDE zusammen mit dem GNU MCU Eclipse Plug-in v4.6.1 installiert. Die Entwicklungsumgebung bietet über einen Editor die Möglichkeit zur C Programm-Code-Eingabe und nutzt die GNU-Toolchain zur Kompilierung des Quell-Codes und zur Programmierung und anschließendem Debugging der Ziel-Hardware. Die verwendete Toolchain ist das in Absatz 4.4.1 beschriebene riscv-none-embed-gcc v8.3.0 Paket. Die SiFive riscv64-unknown-elf-gcc 8.3.0-2019.08.0 Toolchain wird ebenfalls installiert, da sie für die Kompilierung des Programm-Codes bei Ausführung im Simulator verwendet wird [20]. Die Toolchain wird allerdings nicht innerhalb der Entwicklungsumgebung genutzt, sondern durch direkte Terminalbefehle bedient, die in Abschnitt 4.6 beschrieben werden.

RISC-V Toolchain riscv-none-embed-gcc v8.3.0

Die Erweiterung GNU RISC-V Embedded GCC v8.3.0 wird über xPacks installiert und ist die genutzte Toolchain für die Kompilierung des Programm-Codes, welcher auf der Ziel-Hardware ausgeführt werden soll. Diese Toolchain leitet sich aus der SiFive riscv64-unknown-elf-gcc Toolchain ab, wobei auf die Einbindung der libgloss C Bibliothek und damit auf die ecall Instruktion verzichtet wird, da diese Instruktionen zu Problemen bei Bare-Metal Anwendung führt [21]. Für die Verwendung der Toolchain ist die Definitionen von eigenen System Calls (*Systemaufrufe*) notwendig, da diese in Bare-Metal Anwendung hardware-spezifisch sind.

Tabelle 4.1: Eclipse Toolchain Einstellungen

Einstellungspfad	**Attribut**
Target processor → Architecture	RV32I (-march=rv32i*)
Target processor → Multiply extension	RVM
Target processor → Integer ABI	ILP32(-mabi=ilp32*)
Debugging → Debug Level	-g3
GNU RISC-V Cross C Linker → General	Linkerscript angeben
GNU RISC-V Cross C Linker → Miscellaneous	Cross reference und use newlib-nano auswählen

Unter Project → Properties → MCU wird die Toolchain nach der Installation ausgewählt (siehe Anhang C.1.4). Zusätzlich werden unter Project → Properties → C/C++ Build → Settings → Tool Settings Einstellungen aus Tabelle 4.1 vorgenommen, welche für die korrekte Kompilierung des Programm-Codes notwendig sind.

GNU MCU Eclipse Windows Build Tools

Zusätzlich zur IDE werden die xPack Erweiterung Windows Build Tools v2.12.2 installiert. Diese Erweiterung beinhaltet Werkzeuge wie z.B. make, welche die Verskriptung des Kompliervorgangs vereinfachen. Die installierten Tools werden unter Project → Properties → MCU referenziert (siehe Anhang C.1.3).

xPack OpenOCD v0.10.0-13

Als Debugger wird OpenOCD v0.10.0-13 verwendet. Dieses Erweiterungspaket wird ebenfalls über xPacks installiert und kann danach in der Eclipse Entwicklungsumgebung über die Debug Bedienschnittstelle genutzt werden. Unter Project → Properties → MCU wird der Debugger nach der Installation ausgewählt (siehe Anhang C.1.1). Damit der Debugger mit dem Prozessor eine Verbindung aufbauen kann, ist das Hinzufügen einer Konfigurationsdatei in den Quellordner des Debuggers notwendig. Die Datei airi5c.cfg wird dem lokalen Installationsverzeichnis unter \xPacks\xpack-dev-tools\openocd\0.10.0-13.1\.content\scripts\target hinzugefügt. In Auflistung C.1 ist die Konfigurationsdatei dargestellt. Zeile eins bis fünf sind für jede Konfigurationsdatei notwendig und definieren den Chip Namen, welcher in diesem Fall der Bezeichnung airi5c entspricht. In Zeile acht wird der TAP für die JTAG Schnittstelle definiert, welcher mit den Definitionen des DTM übereinstimmen sollte. Zuletzt wird in Zeile neun die Ziel-CPU definiert [22]. Nach der Installation und Einfügung der Konfigurationsdatei werden im letzten Schritt einige zusätzliche Einstellungen unter den Debug Configurations → GDB OpenOCD Debugging → Debugger durchgeführt (siehe Anhang C.1.2). Dadurch wird beim Start des Debuggers die neue Konfigurationsdatei geladen, wodurch der Prozessor über den Debugger ansprechbar ist.

4.4.2 Xilinx Unified Software Platform 2020.1

Für die Entwicklung der Hard- und Software werden Tools der Firma Xilinx verwendet. Durch das installieren der Xilinx Unified Software Platform 2020.1 werden fast alle notwendigen Software-Entwicklungswerkzeuge installiert. Darunter ist die Vivado 2020.1 IDE, die für die Verarbeitung von HDL Dateien genutzt wird. Desweiteren wird das Xilinx Software Command Line Tool für die Generierung von MEM Dateien verwendet. Das Programm data2mem ist bei der neuen Version des Software-Pakets jedoch nicht mehr enthalten und muss daher separat heruntergeladen und dem Xilinx Installationsverzeichnis unter \Xilinx\Vitis\2020.1\bin\unwrapped\win64.o hinzugefügt werden [23].

4.5 Software-Entwicklung

Dieser Abschnitt umfasst die Software-Entwicklung in C und dem RISC-V Assembler. Ziel ist ein lauffähiger Programm-Code in Form einer ELF Datei. Es wird auf den C Quell-Code, die C Startdatei crt0.S und das Linker-Skript eingegangen.

4.5.1 Quell-Code in C

Ein zentraler Bestandteil der Software-Entwicklung ist der Programm-Code in der Hochsprache C. In diesem Abschnitt wird die main.c Datei erläutert, die das Hauptprogramm beinhaltet. Dieses Programm unterteilt sich hauptsächlich in zwei Funktionen. Hierbei handelt es sich zum Einen um die *main()* Funktion, mit der Speicherplatz reserviert und geprüft wird, ob der Speicher geschrieben worden ist. Des weiteren ist eine Interrupt Service Routine (ISR) (*Unterbrechungsroutine*) definiert worden, die ein gültiges Ergebnis des Caecos in den reservierten Speicherbereich schreibt. Der komplette Programm-Code ist in Auflistung 4.2 abgebildet.

```
1#include <stdio.h>
2#include <stdlib.h>
3#include <stdbool.h>
4#include "syscalls.h"
5
6int *cur_res = NULL;
7int *out_res = NULL;
8int *storage = NULL;
9
10void exception_handler(void) __attribute((interrupt));
11void exception_handler(void){
12 // do nothing
13}
14
15void l_1_caeco_interrupt_handler(void) __attribute((interrupt));
16void l_1_caeco_interrupt_handler(void){
17 int* pointer = (int *) 0xc00000c0;
18 int value = *pointer;
19 *cur_res = value;
20 ++cur_res;
21}
22
23int main (void)
24{
```

```
25 storage = (int *) malloc(1000 * sizeof(int));
26 if(storage != NULL) {
27  printf("\n ok\n");
28 }else {
29  printf("\n fail\n");
30 }
31 cur_res = storage;
32 out_res = storage;
33 printf("\n run!\n");
34 while (1){
35  if (out_res != cur_res){
36   printf("\n result: %p\n", *out_res);
37   ++out_res;
38  }
39 }
40}
```

Auflistung 4.2: main.c

In Zeile eins bis vier werden drei Standardbibliotheken und die Systemaufrufe eingebunden. Die Definitionen der Systemaufrufe in syscalls.h sind für die Nutzung der newlib-nano Bibliothek notwendig und entsprechen bis auf die Funktion *outbyte()* den vorgeschlagenen Standard Funktionen (siehe [24], S. 319). Danach folgen drei Definitionen von globalen Zeigern. An dieser Stelle muss den Zeigern ein initialer Wert zugewiesen werden, sodass diese nach dem Linkervorgang in der initialisierten Datensektion gespeichert werden [25]. Andernfalls wird der Programm-Code mit dem vorhanden Linkerskript nicht korrekt ausgeführt, da die globalen Variablen nicht geladen werden. In den Zeilen zehn bis 13 ist der Exception-Handler *exception_handler()* definiert. Diese Funktion wird im Falle einer Exception aufgerufen, führt allerdings keinen Programm-Code aus, da das Verhalten einer Exception nicht Gegenstand dieser Arbeit ist. Durch den Zusatz in Zeile zehn wird die Funktion auch für das Linkerskript als Interrupt gekennzeichnet, wodurch keine normaler Rücksprung ret zur zuletzt gespeicherten Rücksprungadresse, sondern ein Maschinenrücksprung mret zur im Register epc gespeicherten Adresse stattfindet.
Der Interrupt-Handler *l_1_caeco_interrupt_handler()* ist von Zeile 15 bis 21 definiert und wird bei einem durch den Caeco ausgelösten Interrupt aufgerufen. Dabei wird in Zeile 17 die Adresse des Caecos in den Zeiger pointer gespeichert. Anschließend wird der Wert an der Speicheradresse, welcher dem Ergebnisregisters des Caecos entspricht, in der Variablen value gespeichert. Das Ergebnis wird danach an eine Adresse eines Speicherbereichs geschrieben, der in der *main()* zuvor reserviert wurde, woraufhin die Adresse um den Wert vier inkrementiert wird.
Die *main()* Funktion umfasst die Zeilen 23 bis 40. Hier wird in Zeile 25 zu aller erst ein Speicherbereich von 4 kB reserviert. Ob die Zuteilung erfolgreich war, wird in Zeile 26 bis 30 geprüft. Bei einer gültigen Speicherplatzreservierung besitzt die Variable storage die Startadresse des Speicherraums

und wird durch das Senden der Zeichenkette “ok“ quittiert. Andernfalls wird die Zeichenkette “fail“ per UART Schnittstelle gesendet. Für die Feststellung, ob zwischenzeitlich ein Interrupt aufgetreten ist und neue Daten in den Speicher geschrieben worden sind, werden initial die Zeiger cur_res und out_res auf die Startadresse des Speicherraums gesetzt und die Endlosschleife wird gestartet. Innerhalb der Schleife wird geprüft, ob sich cur_res verändert hat, was auf das Erscheinen eines Interrupts hindeutet würde. Ist dies der Fall, dann wird das Ergebnis per UART ausgegeben und out_res angeglichen.
Für die Umwandlung des Programm-Codes in eine ELF Datei, die letztendlich in den Programmspeicher geladen werden kann, sind weitere Dateien notwendig. Diese zusätzlichen Dateien sind im folgenden Abschnitt genauer beschrieben.

4.5.2 Generierung der ELF Datei

Für die Generierung der notwendigen ELF Datei ist eine RISC-V spezifische Toolchain notwendig. Im folgenden wird der Kompilliervorgang des C Quell-Codes mit der aktuellen riscv-none-embed-gcc 8.3.0-1.1 Toolchain beschrieben. Diese leitet sich aus der SiFive riscv64-unknown-elf-gcc Toolchain ab, wobei allerdings die libgloss Bibliothek nicht mehr standardmäßig beim Linkprozess standardmäßig mit eingebunden wird, da es Probleme mit dem Aufruf von ecalls bei bare metal Anwendung gibt. Am Ende dieses Abschnittes wird auf die Unterschiede in den kompilierten ELF Dateien beider Toolchains eingegangen. Die für die Kompilierung notwendigen Dateien sind in Tabelle 4.2 gelistet.

Tabelle 4.2: Notwendige Dateien für Kompilierung

Dateiname	**Funktion**
main.c	Enthält Programm-Code
crt0.S	C runtime Startdatei
link.ld	Linker Skript
syscalls.h	Systemaufrufe
syscalls.c	Systemaufrufe

Die main.c Datei enthält den C-Programm-Code und und wurde in Abschnitt 4.5.1 beschrieben. Bei der link.ld Datei handelt es sich um das für den Linkprozess notwendige Linker Skript, welches der AT&T's Link Editor Command Language Syntax folgt [26]. Die crt0.S Datei ist die C runtime Startdatei, die den Programmablauf vor der Ausführung der eigentlichen *main()* Funktion festlegt [27]. Dies ist In den folgenden Abschnitten werden die einzelnen Dateien genauer beschrieben. Für einen erfolgreichen Kompiliervorgang und die weitere Nutzung der ELF Datei ist darauf zu achten, dass die Dateien untereinander stimmig sind und einem genauen Format folgen. Dabei ist entscheidend, dass der Programmeinstieg nicht bei der *main()* stattfindet, sondern bei der crt0.S Datei. Dies ist ein entscheidender Unterschied zwischen der Software-Entwicklung, bei der bereits ein Betriebssystem aktiv ist und der Bare-Metal Entwicklung.

Die crt0.S Datei

Die crt0.S Datei enthält RISC-V Assembler Befehle und hat unter anderem die Aufgabe, den Programmablauf vor dem Aufruf der *main()* Funktion zu beschreiben. Die Dateien main.c, crt0.S und link.ld sind durch wechselseitige Definitionen eng miteinander verknüpft. Die komplette crt0.S Datei, die für diese Arbeit verwendet wird, ist in 4.3 dargestellt.

```
1  .section .text
2  .align 2
3  .globl _start
4  .type _start, @function
5  _start:
6   .cfi_startproc
7   .cfi_undefined ra
8   .option push
9   .option norelax
10  la  gp, __global_pointer$
11  .option pop
12  la  sp, __stack
13  .globl exception_handler
14  .type exception_handler, @function
15  .globl l_1_caeco_interrupt_handler
16  .type l_1_caeco_interrupt_handler, @function
17  csrwi mstatus, 0x1F
18  la  x29, trap_handler
19  csrw mtvec, x29
20  mv  x1, x0
21  mv  x2, x1
22  mv  x3, x1
23  mv  x4, x1
24  mv  x5, x1
25  mv  x6, x1
26  mv  x7, x1
27  mv  x8, x1
28  mv  x9, x1
29  mv  x10, x1
30  mv  x11, x1
31  mv  x12, x1
32  mv  x13, x1
33  mv  x14, x1
34  mv  x15, x1
35  mv  x16, x1
```

```
36 mv  x17, x1
37 mv  x18, x1
38 mv  x19, x1
39 mv  x20, x1
40 mv  x21, x1
41 mv  x22, x1
42 mv  x23, x1
43 mv  x24, x1
44 mv  x25, x1
45 mv  x26, x1
46 mv  x27, x1
47 mv  x28, x1
48 mv  x29, x1
49 mv  x30, x1
50 mv  x31, x1
51 la x26, _bss_start
52 la x27, _bss_end
53 bge x26, x27, zero_loop_end
54 zero_loop:
55  sw x0, 0(x26)
56  addi x26, x26, 4
57  ble x26, x27, zero_loop
58 zero_loop_end:
59 call main
60.cfi_endproc
61 1:
62 j  1b
63trap_handler:
64 j exception_handler
65 j l_1_caeco_interrupt_handler
66 mret
```

Auflistung 4.3: crt0.S

In Zeile eins wird die Sektion .text markiert und für das Linker Skript sichtbar gemacht. Diese Sektion markiert den Bereich des ausführbaren Programm-Codes und muss in der Datei crt0.S angegeben werden. Alle Befehle nach dieser Zeile gehören demnach aus Linker-Sicht zu dem ausführbaren Programm-Code. In Zeile zwei wird die Speicherausrichtung definiert. Hierbei entspricht die Anzahl der Bytes dem Potenzwert von vier, berechnet aus der Basis zwei und dem angegebenen Exponenten, in diesem Fall der Wert zwei [28]. In den Zeilen drei bis fünf wird der Eintrittspunkt für das Linker Skript definiert, welcher mit Zeile 13 des Linker Skriptes link.ld korreliert. In Zeile sechs und sieben werden Call Frame Information (CFI) Instruktionen aufgerufen, die erweiterte

Informationen zu einer Funktion generieren. Diese Informationen sind für Hochsprachen notwendig, die Exceptions oder Debug-Prozesse verarbeiten müssen. Hierbei handelt es sich um keine Prozessor Instruktionen. Die CFI Informationen werden automatisch separat in der Sektion .eh_frame gespeichert. Zeile sieben gibt an, dass der Inhalt, den das Registers ra vor Ausführung der _start Funktion besessen hat, nämlich die Rücksprungadresse nicht mehr benötigt und dementsprechend auch nicht mehr wiederhergestellt werden braucht und damit gelöscht werden kann [29, 30].
Die Zeilen acht bis elf können als Befehlspaket gesehen werden, die für die RISC-V Plattformen notwendig sind. Durch den Befehl können .option RISC-V spezifische Assembler Optionen angegeben werden. Mit dem Argument push bzw. pop werden aktuelle .option Konfigurationen gespeichert bzw. wiederhergestellt. Diese Methoden können z.b. dann verwendet werden, wenn eine Option nur temporär gesetzt werden soll. Dies ist in Zeile neun und zehn der Fall. Hier wird die Assmebler- bzw. Linker Option norelax in Bezug auf die Assembler Instruktion in Zeile zehn gesetzt. Dadurch soll verhindert werden, dass der Assembler die Adressenangabe im la Befehl durch eine Offsetangabe zum aktuellen globalen Zeigerwert ersetzt und in Folge verkürzt. Diese Relaxationsvorgang wird normalerweise genutzt, um eine Adresskonstante mit nur einer einziger Instruktion laden zu können, was aber bei der ersten Initialisierung des globalen Zeigerwertes noch nicht möglich ist. (siehe [29], Kapitel 9.38.2 RISC-V Directives).
In Zeile zwölf befindet sich nun die erste Prozessor Instruktion, die nach der Kompilierung an der Adresse 0x80000000 steht. Hier wird die Adresse des Stacks 0x4000, der in dem Linkerskript definiert wird, in das Register sp geladen. Danach wird in Zeile 13 und 14 der Exception-Handler bzw. in Zeile 15 und 16 der Interrupt-Handler der main.c als globale Funktion deklariert und dem Linker so zugänglich gemacht. Bei diesen beiden Aufrufen handelt es sich ebenfalls nicht um Prozessor Instruktionen. Zeile 17 entspricht einem Schreibbefehl auf das CSR mstatus, durch den die Interrupts aktiviert werden. Danach wird in Zeile 18 die Adresse der in Zeile 63 definiert Trap-Handler Funktion in das register 29 geladen, woraufhin in Zeile 17 diese Adresse in das CSR mtvec geschrieben wird. Hierdurch wird bei einem Interrupt die Adresse der Funktion l_0_caeco_interrupt_handler geladen und diese Funktion ausgeführt.
In den Zeilen 20 bis 50 werden alle Allzweckregister auf den Wert null gesetzt. In den Zeilen 51 bis 58 wird das .bss Speichersegment, in dem statische Variablen abgelegt werden, auf den Wert 0 initialisiert. Dafür werden die Start- und Stopadresse der Sektion geladen und die Startadresse solange inkrementiert, bis diese der Stopadresse gleicht. Daraufhin wird die *main()* aus der main.c Datei aufgerufen. Sollte die Hauptfunktion verlassen werden, so wird in die Zeile 62 gesprungen, wodurch der Prozessor immer wieder auf den selben Befehl zurückspringt und nur noch durch einen Hardware-Reset zurückgesetzt werden kann.
Am Schluss der Datei, in Zeile 63 bis 66 ist der Trap-Handler definiert. Hier findet sich nun die bereits in Auflistung 3.20 beschriebene Implementierung des Hardware-Trap-Handlers wieder. Hierbei zeigt die Adresse, welche in Zeile 19 gespeichert wird, auf den Programm-Code an Zeile 63. Bei einem Caeco Interrupt wird somit die Adresse geladen, die dem Programm-Code an Zeile 65 entspricht. An dieser Stelle befindet sich ein Sprungbefehl zu der in der main.c definierten ISR. Analog dazu wird bei einer Exception zu dem in der main.c definierten Exception-Handler

gesprungen. Da beide Funktionen mit dem Attribut eines Interrupts gekennzeichnet sind, findet nach der Ausführung ein Rücksprung zu der im Register epc gespeicherten Adresse statt.
Im folgenden Abschnitt wird auf die Struktur des Linkerskriptes eingegangen, welches mit den Angaben in der main.c und der crt0.S übereinstimmen muss, damit einer erfolgreiche Kompilierung und Ausführung des Programm-Codes möglich ist.

Das Linker Skript link.ld

Im Linker Skript werden eine Vielzahl von Direktiven definiert, welche hauptsächlich dazu dienen, den Programm-Code für die Ausführung hardware-spezifisch zu ordnen und den Speicher zu organisieren. Dabei ist der Programm-Code in Sektionen unterteilt [26]. Mit dem Befehl in Auflistung 4.4 kann das Standard-Linker Skript der Toolchain aus dem Verzeichnis der Toolchain zur besseren Lesbarkeit in ein Zielverzeichnis abgelegt werden (siehe Unterabschnitt A.1.1). Diese Standarddatei gibt eine gute Übersicht auf ein beispielhaftes komplettes Linkerskript.

```
"absoluter Pfad der Toolchain"\bin\riscv-none-embed-ld --verbose >
   "absoluter Pfad des Zielverzeichnis"\link.ld
```

Auflistung 4.4: Kopieren des Standardlinkerskriptes

Da es für den Raifes Core Voreinstellungen gibt, die nicht mit dem des Standardlinkerskriptes übereinstimmen, wird ein neues Linkerskript erstellt. Dieses Skript wird für die Kompilierung verwendet und ist in 4.5 dargestellt.

```
OUTPUT_FORMAT("elf32-littleriscv", "elf32-littleriscv",
        "elf32-littleriscv")
OUTPUT_ARCH(riscv)
MEMORY
{
   RAM (rwx)   : ORIGIN = 0x80000000, LENGTH = 128K
}
ENTRY(_start)
SECTIONS
{
   .text : {
  . = ALIGN(4);
  _stext = .;
  *(.text)
  }
 __global_pointer$ = .;
 .data : {
```

```
18   . = ALIGN(4);
19   *(.data)
20   }
21  .rodata : { *(.rodata) }
22  .bss : {
23   . = ALIGN(4);
24   _bss_start = .;
25   *(.bss)
26   *(.bss.*)
27   *(.sbss)
28   *(.sbss.*)
29   _bss_end = .;
30   }
31  .bss : {
32   . = ALIGN(4);
33   ___end = .;
34  }
35  ___stack = ALIGN(4) + 0x4000;
36  _uart_dreg = 0xc0000000;
37  _gpio_dreg = 0xc0000008;
38  _gpio_creg = 0xc000000C;
39 }
```

Auflistung 4.5: link.ld

In den ersten drei Zeilen wird das Ausgabeformat und die Zielarchitektur der ELF Datei angegeben. Die Zeilen vier bis sieben geben die Speicherregion an. Der Speicher beginnt bei der Zieladresse 0x80000000 und ist 128 kB groß. Die angegebene Speicherkapazität sollte mit der in der Hardware vorhandenen Größe übereinstimmen. Andernfalls muss das Linkerskript an dieser Stelle angepasst werden. Die erste Linkerinstruktion findet sich in Zeile acht wieder. Hier wird der Start für den Linkprozess angegeben. Danach folgt die Angabe der Sektionen, also der unterschiedlichen Programmabschnitte. Diese werden dadurch für die Ausgabestruktur geordnet. Die erste Sektion wird in Zeile elf angegeben. In der .text Sektion befindet sich der eigentliche Programm-Code. In Zeile 16 wird der global pointer (*globaler Zeiger*) auf die Adresse nach der .text Sektion gesetzt. Darauf folgen die Sektionen .data und .rodata, die initialisierte Daten und nur lesbare initialisierte Daten beinhalten.
Zum Schluss wird die .bss Sektion geladen, die mit Nullwerten initialisierte Daten enthält. Nach dieser Sektion folgt der Heap, der durch die Funktion _sbrk in den Systemaufrufen angegeben wird. Dafür existiert die Angabe in Zeile 33, da diese Variable in der genannten Funktion aufgerufen wird, um das Ende der .bss Sektion und den Start des Heaps zu markieren.
Am Ende des Skriptes wird der Beginn des Stacks auf die Adresse 0x4000 gesetzt und hardwarespezifische Adressen angegeben. Somit ergibt sich der finale Adressraum. Dabei sind einige Angaben

statisch, wie z.B. die Stackadresse, andere Angaben sind dynamisch, da die Größe einiger Sektionen und somit die genauen Adressen vor der Kompilierung nicht bekannt sind [25, 27].

4.5.3 Generierung einer MEM Datei mit Xilinx data2mem

Die Umwandlung der ELF Datei in eine simulierbare MEM Datei erfolgt in zwei Schritten. Der erste Schritt beinhaltet dabei die Verwendung des Xilinx Tools data2mem, während in einem zweiten Schritt eine Weiterverarbeitung der generierten Datei durch ein selbst geschriebenes Programm erfolgt. Für den Aufruf des Xilinx Tools data2mem muss zuerst das Xilinx Software Command Line Tool auf der Entwicklungsmaschine geöffnet werden. Außerdem wird dann über den in Auflistung 4.6 angegebenen Befehl die Umwandlung durchgeführt.

```
data2mem -bd "absoluter Pfad der ELF Datei" -d e -o m "absoluter
    Ausgabepfad mit Dateiname der MEM Datei"
```

Auflistung 4.6: Nutzung von data2mem für MEM Generierung

Dem Aufruf von data2mem folgen spezifische Parameter, welche dafür sorgen, dass die Ausgangsdatei gelesen und die Zieldatei generiert wird. Dabei wird mit -bd die ELF Datei übergeben, die umgewandelt werden soll. Die Parameter -d e geben das Level der Depaketierung an. Durch die Parameter -o m wird der Ausgangspfad der MEM Datei angegeben.

```
// MEM file.
//
// Release 14.6 - Data2MEM P.20131013, build 3.0.10 Apr 3, 2013
// Copyright (c) 1995-2020 Xilinx, Inc.  All rights reserved.
//
// Command: data2mem.exe -bd "absoluter Pfad der ELF Datei" -d e -o m
    "absoluter Ausgabepfad mit Dateiname der MEM Datei"
//
//    "absoluter Ausgabepfad mit Dateiname der MEM Datei"


// Program header record #0, Size = 0x1FF0, at 0x80000000 to 0x80001FEF.

@80000000
    97 01 00 00 93 81 C1 0E 17 61 00 00 13 01 01 1F 73 D0 0F 30 97 0E 00
    00 93 8E 0E 0B 73 90 5E 30
    ...
```

Auflistung 4.7: MEM Datei data2mem

Wie genau die Umwandlung der ELF Datei durch data2mem stattfindet, ist aufgrund fehlenden Quell-Codes des Programms nicht nachvollziehbar. Allerdings weisen die MEM Ausgangsdateien immer dieselbe Struktur auf (siehe Auflistung 4.7). Diese Struktur besteht aus einem Header, der Zeile eins bis 13 umfasst. Hierbei findet sich in Zeile sechs der Aufruf von data2mem, der zuvor durchgeführt wurde, wieder. In Zeile zwölf wird die Größe des Programm-Codes angegeben. Die Startadressenangabe steht in Zeile 14, worauf ab Zeile 15 die eigentlichen Programminstruktionen folgen. Hierbei sind die Instruktionen 32 bit groß, beginnend mit dem LSB. Diese Struktur entspricht allerdings nicht den Xilinxvorgaben für MEM Dateien, die für das Einlesen von Programm-Code geeignet sind [31]. Das notwendige Format für eine MEM Datei besteht aus einer Startadresse, so wie sie in Zeile 14 gekennzeichnet wird, gefolgt von Instruktionen als Hexadezimalzahlen mit der Bitwertigkeit LSB 0. Hierzu sind in den folgenden Abschnitten Hilfsprogramme beschrieben, welche die Umwandlung zur simulierbaren MEM und COE Dateien durchführen.

4.5.4 Anpassung der MEM Datei für RTL Simulation

Der folgende Abschnitt beschreibt die Anpassung der durch data2mem generierten MEM Datei mit Hilfe eines selbst geschriebenen Hilfsprogramms in der Programmiersprache Rust. Dieser Schritt ist auch im CPU Software Entwicklungsablauf dargestellt und entspricht dem Schritt new-mem.exe in Abbildung 4.1. Für die Verwendung des $readmem Befehls innerhalb der verilog Testbench (*Prüfstand*) muss die MEM Datei immer das am Ende dieses Abschnittes dargestellte Format aufweisen. Der vollständige Programm-Code von new-mem.exe ist in Anhang A.6 beschrieben. Das Programm wird unter der Windows Eingabeaufforderung cmd.exe durch den Befehl in Auflistung 4.8 genutzt.

```
1 "absoluter Pfad der .exe Datei"\new-mem.exe "absoluter Pfad der
    Ausgangs-MEM Datei" "absoluter Pfad der finalen MEM Datei"
```

Auflistung 4.8: Generierung der finalen MEM Datei

Das Programm nimmt zwei Argumente entgegen. Durch das erste Argument wird die umzuwandelnde MEM Datei übergeben, die vorher aus data2mem generiert wurde. Das zweite Argument gibt den Speicherpfad und den Dateinamen der Zieldatei an, welche für die RTL Simulation genutzt wird.

```
1 @00000000
2 00000197
3 0EC18193
4 00006117
5 1F010113
```

Auflistung 4.9: Ersten 5 Zeilen der finalen MEM Datei

Die ersten fünf Zeilen der finalen MEM Datei sind in Auflistung 4.9 dargestellt. Das Programm wandelt die MEM Datei in das korrekte Format um, sodass die Zieldatei in der Verilog Testbench.

4.5.5 Generierung einer COE Initialisierungsdatei für Block-RAM

Die generierte MEM Datei wird vor allem für die Simulation und die Verifizierung des Ladevorgangs der Programmdaten in den Block-RAM über die JTAG Schnittstelle genutzt. Da dies nicht für Simulationen mit einem anderen Schwerpunkt benötigt wird, kann der Block-RAM über die IP Einstellungen direkt mit dem Programm-Code initialisiert werden. Dies erspart die zeitaufwändige Simulation bei Programm-Code mit größerem Umfang. Hierfür findet new-coe.exe als zweites selbst geschriebenes Hilfsprogramm Anwendung. Dieses Programm wandelt die durch das in Abschnitt 4.5.4 beschriebene Hilfsprogramm new-mem.exe generierte MEM Datei in eine lauffähige COE Datei um. Der Aufruf von new-coe.exe ist in Auflistung 4.10 dargestellt. Der gesamte Code ist in Anhang A.7 einsehbar.

```
"absoluter Pfad der .exe Datei"\new-coe.exe "absoluter Pfad der
    Ausgangs-COE Datei" "absoluter Pfad der finalen COE Datei"
```

Auflistung 4.10: Generierung der finalen COE Datei

Das new-coe.exe Programm nimmt zwei Argumente entgegen. Dabei entspricht das erste Argument dem absoluten Dateipfad der MEM Datei und das zweite Argument dem Speicherort der neu genierten COE Datei als absoluten Dateipfad. Die COE Datei besitzt ein ähnliches Format, wie die MEM Dateien [32]. Die ersten fünf Zeilen und die letzten beiden Zeilen der COE Datei werden in Auflistung 4.11 gezeigt. In Zeile eins und zwei befinden sich notwendige Angaben de COE Dateiformats wie radix und vector. Der radix Faktor, der in der ersten Zeile definiert ist, gibt an, um welches Datenformat es sich bei den zu ladenden Instruktionen handelt. In diesem Fall liegen die Instruktionen als Hexadezimalzahlen vor, somit hat der memory_initialization_radix den Wert 16. In Zeile zwei ist ein Vektor angegeben, der die durch Kommata getrennten Instruktionen enthält. Zeile sechs steht für den ausgeschnitten Programm-Code zwischen Zeile fünf und sieben.

```
memory_initialization_radix=16;
memory_initialization_vector=
00000197,
0ec18193,
00006117,
...
00000000
;
```

Auflistung 4.11: Ersten 5 und letzten 2 Zeilen der finalen COE Datei

Eine Besonderheit liegt am Ende des Vektors vor, da auf die letzte Instruktion ein Semikolon folgen muss. Auf diese Datei kann wie bereits gesagt unter den IP Einstellungen des RAM Blocks verwiesen werden, sodass die Datei zur Initialisierung des Speichers genutzt wird (siehe C.6). Sollte die Datei fehlerhaft sein, wird dies augenblicklich markiert, indem sich der angegebene Pfad der Datei rot färbt.

4.6 Inbetriebnahme riscvOVPsim

Der RISC-V Instruktionssimulator riscvOVPsim der Firma Imperas ist frei verfügbar und wird in dieser Arbeit beschrieben und in Betrieb genommen. Ziel der riscvOVPsim Simulation ist eine erste Verifizierung des Programm-Codes, durch die ermittelt wird, ob das kompilierte Programm lauffähig ist. Dieser Ansatz bietet eine Früherkennung potentieller Fehler und stellt eine Optimierung des Software-Entwicklungsvorgangs dar. Der Simulator bietet zahlreiche Einstellungsmöglichkeiten, wie z.B. die Simulationsdauer gemessen in Instruktionen oder in Zeit, oder Tracings (*Rückverfolgung*) ausgeführter Instruktionen oder Registerwerten. Der Simulator unterstützt alle Prozessormodelle der aktuellen RISC-V Spezifikation [33], wobei individuelle Prozessorarchitekturen bzw. Simulationsmodelle bei Imperas angefragt werden können. Als Simulationsgrundlage wird der kompilierte Programm-Code in Form von einer ELF Datei verwendet. Da die riscv-none-embed-gcc v8.3.0 Toolchain die Definition von eigenen system calls voraussetzt, wird der Programm-Code für die Nutzung des Simulators parallel mit der riscv64-unknown-elf-gcc Toolchain kompiliert. Dadurch wird bei einem *printf()* Aufruf nicht die UART Schnittstelle des Prozessors, sondern das Betriebssystem adressiert, wodurch ein erstes Debugging erleichtert wird. Ist die Simulation erfolgreich, kann das mit der riscv-none-embed-gcc Toolchain kompilierte Programm ebenfalls getestet werden.

Nach der Installation von riscvOVPsim wird im Ordner riscv-ovpsim\examples ein weiterer Ordner erstellt, der die Dateien aus Tabelle 4.3 enthält. Die ersten drei Dateien sind identisch mit denen aus Unterabschnitt 4.5.2, wobei die Zeile vier in der main.c auskommentiert sein muss, damit die angepassten system calls nicht verwendet werden. Die Datei masterthesis.elf entspricht dem mit der riscv64-unknown-elf-gcc 8.3.0-2019.08.0 Toolchain kompilierten Programm-Code und die mastersim.bat Datei enthält die Steuerungsbefehle für den Simulatoraufruf.

Tabelle 4.3: Notwendige Dateien für riscvOVPsim Test

Dateiname	**Funktion**
main.c	Enthält Programm-Code
crt0.S	C runtime Startdatei
link.ld	Linker Skript
masterthesis.elf	Kompilierter Programm-Code
mastersim.bat	riscvOVPsim Konfigurationsdatei

Die Kompilierung des Programm-Codes aus der main.c Datei erfolgt über den Windows Subsystem for Linux (WSL) Terminalaufruf aus Auflistung 4.12. Bei dem Toolchainaufruf werden zuerst zwei Argumente bzgl. der Zielarchitektur übergeben. Desweiteren wird mit dem dritten Aufruf das Laden der Standardstartdateien verhindert, sodass die Datei crt0.S für die Kompilierung verwendet wird. Das vierte Argument entspricht der Verwendung der newlib-nano Bibliothek. Das letzte übergebene Argument Xlinker ist notwendig, damit die Standardbibliotheken ebenfalls beim Linkerprozess genutzt werden.

```
riscv64-unknown-elf-gcc -mabi=ilp32 -march=rv32im -nostartfiles
  --specs=nano.specs crt0.S main.c -Xlinker link.ld -o masterthesis.elf
```

Auflistung 4.12: riscv64-unknown-elf-gcc 8.3.0-2019.08.0 Aufruf

Der mit dieser Toolchain kompilierte Programm-Code entspricht nicht genau dem kompilierten Programm-Code der GNU Embedded Toolchain, weshalb an dieser Stelle keine komplett vergleichbare Verifizierung möglich ist. Die Simulation dient dazu, den Programm-Code einem ersten Test zu unterziehen, um so frühzeitig Fehler zu erkennen und den Code gegebenenfalls zu korrigieren. Die Durchführung des Tests ist im folgenden Unterabschnitt erklärt.

4.6.1 Konfigurationsdatei

Das Batchskript masterthesis.bat enthält diverse Basiskonfigurationen bzgl. des simulierten Prozessors und ist in Auflistung 4.13 dargestellt. Eine Übersicht über alle Konfigurationsoptionen sind im mitinstallierten Nutzerhandbuch unter riscv-ovpsim\doc einzusehen. Die Einstellung des Prozessors erfolgt in den Zeilen 7 bis 15, in denen angegeben wird, mit welchen Argumenten die riscvOVPsim.exe aufgerufen wird.

```
1  @echo off
2
3  ;rem move into the Example Directory
4  set BATCHDIR=%~dp0%
5  cd /d %BATCHDIR%
6
7  ..\..\bin\Windows64\riscvOVPsim.exe ^
8      --variant RVB32I ^
9      --override riscvOVPsim/cpu/add_Extensions=M ^
10     --finishafter 1000000 ^
11     --program masterthesis.elf ^
12     --tracefile trace-masterthesis.txt ^
13     --t ^
```

```
14      --traceregsafter ^
15      --logfile masterthesis.sig.run.log ^
16      %*
17
18  if not defined calledscript ( pause )
```

Auflistung 4.13: riscvOVPsim Konfiguration

Die ersten beiden Parameter legen eine RISC-V RV32IM Prozessorarchitektur fest, wie sie dem entwickelten RISC-V Kern entspricht. Danach wird eine optionale Grenze von 1 Mio. Instruktionen definiert, nach der die Simulation stoppen soll. In Zeile 11 wird angegeben, dass der zuvor kompilierte Programm-Code masterthesis.elf simuliert werden soll. In den Zeilen zwölf bis 14 wird das Tracing eingestellt und definiert, dass die ausgeführten Instruktionen und die durch eine Instruktion geänderten Registerinhalte in die Datei trace-masterthesis.txt geschrieben werden sollen. Alle weiteren Informationen über den Ausgang der Simulation werden in die Logdatei masterthesis.sig.run.log geschrieben. Um die Simulation zu starten, ruft man das Batchskript durch einen Terminalaufruf auf.

5 Simualtionsergebnisse und Debugging

Dieses Kapitel ist in drei Abschnitte unterteilt, in denen die Simulations- und Debugergebnisse beschrieben werden. In Abschnitt 5.1 werden die Ergebnisse der riscvOVPsim Simulation vorgestellt. Dabei geht es hauptsächlich, um eine fehlerfreie Ausführung des Programmcodes und eine Analyse, wie der Simulator für zukünftige Verifikationsarbeiten genutzt werden kann.
Im Abschnitt 5.2 werden die RTL Simulationsergebnisse vorgestellt. Dabei geht es zum einen um die Simulations des Ziel-Design, in dem der Caeco als Interrupt-Quelle implementiert ist, zum anderen wird aber auch eine Hardware-Version getestet, bei der der Interrupt durch einen Schalter auf der Entwicklungsplatine ausgelöst wird.
Im letzten Abschnitt wird über die Eclipse IDE das auf dem FPGA implementierte Design gedebbugt. An dieser Stelle kann die Verifizierung des Programmcodes erneut nur bis Ausführung der Endlosschleife stattfinden, da im Debugmodus des Prozessors keine Interrupts ausgelöst werden können.

5.1 Simulationsergebnisse riscvOVPsim

Mit dem Instruktionssimulator riscvOVPsim ist eine Simulation über 1 Mio. Instruktionen erfolgreich durchgeführt worden. Ein Ausschnitt aus diesen Simulationsergebnissen ist in Auflistung 5.1 dargestellt. Für die Auswertung entscheidend sind allerdings nur die Informationen der Zeilen 28 bis 52 der Originaldatei, die hier den Zeilen 1 bis 25 entsprechen. Dabei geben die Zeilen eins bis vier Auskunft über das simulierte Programm bzgl. der Startadresse und der Größe, die in diesem Fall der Adresse 0x7ffff000 und ca. 366 kB entsprechen.

```
1 Info (OR_OF) Target 'riscvOVPsim/cpu' has object file read from
    'master-unknown.elf'
2 Info (OR_PH) Program Headers:
3 Info (OR_PH) Type           Offset     VirtAddr   PhysAddr   FileSiz
    MemSiz     Flags Align
4 Info (OR_PD) LOAD           0x00000000 0x7ffff000 0x7ffff000 0x00016608
    0x00016654 RWE   1000
5
6  ok
7
8  run!
```

```
9 Info
10 Info ------------------------------------------------------
11 Info CPU 'riscvOVPsim/cpu' STATISTICS
12 Info    Type                  : riscv (RVB32I+M)
13 Info    Nominal MIPS          : 100
14 Info    Final program counter : 0x800001d8
15 Info    Simulated instructions: 1,000,000
16 Info    Simulated MIPS        : 0.1
17 Info ------------------------------------------------------
18 Info
19 Info ------------------------------------------------------
20 Info SIMULATION TIME STATISTICS
21 Info    Simulated time        : 0.01 seconds
22 Info    User time             : 8.06 seconds
23 Info    System time           : 0.77 seconds
24 Info    Elapsed time          : 8.94 seconds
25 Info ------------------------------------------------------
```

Auflistung 5.1: Simulationslog riscvOVPsim riscv64-unknown-elf-gcc Toolchain

In Zeile sechs und acht sind die Ausgabewerte der genutzten *printf()* Funktion der main.c gezeigt. Damit ist erkennbar, dass der Programm-Code korrekt ausgeführt wird und die Reservierung des nötigen Speicherplatzes für dieses Prozessormodel erfolgreich ist. Hierbei muss allerdings darauf geachtet werden, dass der angegebene Speicher des Modells mit der Speichergröße des später genutzten Speichers übereinstimmen. Bei Beendigung der Simulation hat der Programmzähler den Wert 0x800001d8, welches einer Instruktion der Endlosschleife innerhalb der *main()* Funktion entspricht. Zeile 15 gibt die Anzahl der durchgeführten Instruktionen an, die der Vorgabe aus der Konfigurationsdatei entspricht.
In Auflistung 5.2 ist der Anfang des Tracelogs dargestellt. Durch den Log können die ausgeführten Instruktionen ausgewertet werden. In diesem Fall sind die ersten fünf Instruktionen aus der crt0.S einsehbar. Durch diesen erweiterten Log kann im Falle einer fehlerhaften Simulation nach der Ursache geforscht werden, indem die gescheiterte Instruktion zurückverfolgt und analysiert werden.

```
1 nfo 1: 'riscvOVPsim/cpu', 0x0000000080000000(_start): Machine 00015197
     auipc   gp,0x15
2 Info    gp 00000000 -> 80015000
3 Info 2: 'riscvOVPsim/cpu', 0x0000000080000004(_start+4): Machine 44018193
     addi    gp,gp,1088
4 Info    gp 80015000 -> 80015440
5 Info 3: 'riscvOVPsim/cpu', 0x0000000080000008(_start+8): Machine 80004117
     auipc   sp,0x80004
6 Info    sp 00000000 -> 00004008
```

```
Info 4: 'riscvOVPsim/cpu', 0x000000008000000c(_start+c): Machine ff810113
   addi    sp,sp,-8
Info    sp 00004008 -> 00004000
Info 5: 'riscvOVPsim/cpu', 0x0000000080000010(_start+10): Machine 300fd073
   csrwi    mstatus,31
Info    mstatus 00001800 -> 00001808
```

Auflistung 5.2: Tracelog riscvOVPsim riscv64-unknown-elf-gcc Toolchain Toolchain

Vergleicht man die Simulation ELF-Datei, die mit der riscv64-unknown-elf-gcc Toolchain generiert wurde, mit der Simulation der ELF-Datei die mit der riscv-none-embed-gcc Toolchain kompiliert wurde und in Auflistung 5.3 dargestellt ist, ist zu erkennen, dass zum einen die Startadresse der gewünschten Adresse von 0x80000000 entspricht. Außerdem ist der Programm-Code mit ca. 34 kB deutlich kleiner als ersterer. Insgesamt verläuft die Simulation ebenfalls fehlerfrei, allerdings werden die *printf()* Befehle nicht ausgegeben, was eine schnelle Verifikation des Codes erschwert.

```
Info (OR_OF) Target 'riscvOVPsim/cpu' has object file read from
   'master-embed.elf'
Info (OR_PH) Program Headers:
Info (OR_PH) Type           Offset     VirtAddr   PhysAddr   FileSiz
   MemSiz     Flags Align
Info (OR_PD) LOAD           0x00001000 0x80000000 0x80000000 0x000021e0
   0x000021fc RWE   1000
Info
Info ------------------------------------------------------------
Info CPU 'riscvOVPsim/cpu' STATISTICS
Info   Type                  : riscv (RVB32I+M)
Info   Nominal MIPS          : 100
Info   Final program counter : 0x8000043c
Info   Simulated instructions: 1,000,000
Info   Simulated MIPS        : 0.1
Info ------------------------------------------------------------
Info
Info ------------------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 0.01 seconds
Info   User time             : 7.95 seconds
Info   System time           : 1.09 seconds
Info   Elapsed time          : 9.17 seconds
Info ------------------------------------------------------------
```

Auflistung 5.3: Simulationslog riscvOVPsim riscv-none-embed-gcc Toolchain

Nach 1 Mio. Instruktionen befindet sich der Programmzähler an der Adresse 0x8000043c, welche sich innerhalb der Endlosschleife der *main()* Funktion befindet. Abschließend kann festgehalten werden, dass beide Programm-Codes auf einem spezifikationsgerechtem RV32IM Prozessor lauffähig sind. Allerdings wurde bei beiden nicht die ISR getestet, wodurch keine vollwertige Verifizierung des Programm-Codes erreicht wird. Die Nutzung des Simulators im Zuge dieser Arbeit dient zum einen als Einarbeitung in die Handhabung des Simulators und anderseits zur ersten Überprüfung des Programms innerhalb des Software-Entwicklungsfluss. Bei einer fehlerhaften Simulation kann das Programm untersucht und verbessert werden, sodass die Software-Entwicklung effizienter gestalten wird. In Bezug auf den Code, der mit der riscv-none-embed-gcc Toolchain kompiliert wird, ist eine erste Verifizierung nicht möglich, da die Ausgabe der *printf()* Funktion an die UART Schnittstelle gesendet wird.

5.2 RTL Simulation in Vivado

Für die Verifizierung des Designs auf RTL Ebene wird das Schreiben der EKG Daten über JTAG in Vivado simuliert und das Interrupt-Verhalten ausgewertet. Hierbei wird der Blockram bereits mit dem Programm-Code initialisiert (siehe C Abbildung C.6). Die EKG Daten eines kranken Patienten liegen als .ecg Datei vor und werden über die in Auflistung 5.4 dargestellte Testbench zwei mal hintereinander in den Caeco geschrieben.

```
1 `timescale 1ns / 1ns
2 `include "raifes_hasti_constants.vh"
3
4 module masterthesis_tb();
5 //------------------------------------------ signals, registers
6 // mandatory:
7 reg             CLK, RESET, tck, tms, tdi, ext_inter;
8 reg     [31:0]  result;
9 integer         testcase, i;
10 wire            nRESET, caeco_led;
11 // optional
12 // writing .ecg data through dmi
13 reg     [63:0]  memimg_caeco_dmi[30805-1:0];
14 reg     [63:0]  buff_caeco_dmi;
15 reg     [31:0]  sample_caeco_dmi;
16 integer         mcd_caeco_dmi;
17
18 //------------------------------------------ DUT
19 raifes_fpga_wrapper DUT(
20         .clk_raw(CLK),
```

```
21          .nRESET(nRESET),
22          .TCK(tck),
23          .TDI(tdi),
24          .TDO(tdo),
25          .TMS(tms),
26          .caeco_led(caeco_led),
27          .uart_tx(uart_tx),
28          .led(ledout),
29          .sw(8'b00000000),
30          .btnu(1'b0),
31          .btnd(1'b0),
32          .btnl(1'b0),
33          .btnr(1'b0),
34          .btnc(1'b0)
35  );
36  //------------------------------------------------ mandatory
37  assign nRESET = ~RESET;
38  // clock setting on 100 MHz for board clock -> 25 MHz mcu clock
39  always
40  begin
41      CLK = 1'b1;
42      #5;
43      CLK = ~CLK;
44      #5;
45  end
46
47  //`define CLK_PERIOD 50
48  // speed up JTAG -> 20 MHz
49  `define CLK_PERIOD 10
50  `include "jtag_tasks.vh"
51  //------------------------------------------------ caeco dmi
52  task caeco_write_dmi;
53  input   reg[7:0]        testnum;
54  input   reg[255*8:1]    filename;
55  input   reg[15:0]        length;
56  output  reg[31:0]       result;
57  begin
58      $write("JTAG Task Start: write Caeco through dmi! \n");
59      // set cmd signal directly with dmi address of 23
60      $write("Set the caemo signal cmd \n");
61      jtag_write_caeco(6'h23,32'h00000001,result);
62      // write the data
```

```
63      $write("Read ecg file for real testcase\n");
64      mcd_caeco_dmi = $fopen(filename, "rb");
65      $fread(memimg_caeco_dmi, mcd_caeco_dmi);
66      // write the data via jtag directly through the dm
67      i = 0;
68      for (i = 49; i < length; i = i + 1) begin
69          // 64 - Bit = 2 Datasamples
70          // get Sample 1 = 32 MSB
71          buff_caeco_dmi = memimg_caeco_dmi[i];
72          sample_caeco_dmi = buff_caeco_dmi[63:32];
73          // check if value is not zero (for last sample)
74          if (sample_caeco_dmi != 32'h0) begin
75              $write("writing sample:
    ");$write(sample_caeco_dmi);$write("\n");
76              jtag_write_caeco(6'h22,sample_caeco_dmi,result);
77          end
78          // get Sample 1 = 32 MSB
79          sample_caeco_dmi = buff_caeco_dmi[31:0];
80          if (sample_caeco_dmi != 32'h0)begin
81              $write("writing sample 2:
    ");$write(sample_caeco_dmi);$write("\n");
82              jtag_write_caeco(6'h22,sample_caeco_dmi,result);
83          end
84      end
85      $write("JTAG Task End: write Caeco through dmi! \n");
86  end
87  endtask
88  //------------------------------------------ Testbench start!
89  initial begin
90      $write("Testbench is starting! \n");
91      RESET <= 1'b1; tms <= 1'b0; tdi <= 1'b0; tck <= 1'b0; i <= 32'h0;
92      // reset the design for 2 ms to wait for the xilinx ip bram
93      $write("Wait for BRAM \n");
94      #2000000;
95      RESET <= 1'b0;
96      $write("JTAG TAP: reset..\n");
97      jtag_tap_reset;
98      $write("Initializing finished! Now starting with the testcases \n");
99
100     // if COE file is used, let the CPU run for initialization!
101     #200000;
102
```

```
103     caeco_write_dmi(1,"00019fb0-6b6a-4ccf-b818-b52221ec524c.ecg",1116,
    result);
104     #105000000;
105
106     caeco_write_dmi(1,"00019fb0-6b6a-4ccf-b818-b52221ec524c.ecg",1116,
    result);
107     #105000000;
108
109     $finish();
110 end
111 endmodule
```

Auflistung 5.4: Testbench für Caeco Interrupt

Die Zeilen 1 bis 35 entsprechen den Standardangaben einer Verilog Testbench, bei der zuerst die Zeitskalierung angegeben wird (Z. 1), benötigte Header-Dateien referenziert werden (Z. 2), notwendige Signaldeklaration stattfinden (Z. 7 - 16) und das Device Under Test (DUT) (*Testmodul*) instantiiert wird. Die Taktfrequenz wird in Zeile 39 bis 45 auf 100 MHz gesetzt. Dies entspricht dem Takt der Entwicklungsplatine und wird innerhalb des Moduls raifes_fpga_wrapper auf 25 MHz reduziert. Die JTAG Frequenz wird in Zeile 49 auf 20 MHz gesetzt.
Der zentrale JTAG Task caeco_write_dmi, durch den die EKG Daten eingelesen und an das DTM gesendet werden, ist in den Zeilen 53 bis 87 beschrieben. Dieser Task nimmt die drei Funktionsargumente testnum, filename und length entgegen und gibt result zurück, wobei das Argument testnum nur für Debug-Zwecke verwendet wird. Durch die Variable filename wird die EKG Datei referenziert. Diese kann entweder mit einem absoluten Systempfad angegeben werden, oder dem Vivado Projekt unter Simulationsdateien hinzugefügt und daraufhin allein mit dem Dateinamen referenziert werden. Über die Variable length wird angegeben, wie viele Zeilen der EKG Datei, bestehend aus 128 bit, eingelesen und geschrieben werden sollen. Der erste Schreibbefehl an den Caeco erfolgt in Zeile 61. Hier wird das Signal CMD des Caecos gesetzt und somit der Start des Datenschreibprozesses markiert. Mit dem Task jtag_write_caeco, welcher in der Datei jtag_tasks.vh definiert und in Auflistung 5.5 dargestellt ist, wird an die DMI Adresse 0x23 der Wert 1 gesendet.

```
1 task jtag_write_caeco;
2 input [5:0] dmiaddr;
3 input [31:0] wdata;
4 output reg [31:0] result;
5 begin
6  jtag_dmi_write(dmiaddr,wdata,2'h2,result);
7 end
8 endtask
```

Auflistung 5.5: JTAG Task jtag_write_caeco

Der aufgerufene Task jtag_dmi_write steuert direkt den JTAG TAP durch den das DMI Register beschrieben wird und die Daten an das DTM gesendet werden (siehe Anhang B, Auflistung B.3). Danach wird die EKG Datei eingelesen. Die notwendigen Befehle finden sich in Zeile 64 und 65 wieder. Dabei wird die Datei zuerst mit dem Befehl $fopen als lesbare Binärdatei („rb “) in der Variable mcd_caeco_dmi gespeichert und danach in den 64 bit breiten Buffer memimg_caeco_dmi geschrieben. Durch die For-Schleife in den Zeilen 68 bis 86 werden nun die Daten als 32 bit Datenworte in den Caeco geschrieben. Dazu werden zuerst die höherwertigsten 32 bit des Buffers zwischengespeichert und mit dem Task jtag_write_caeco and die DMI Adresse 0x22 gesendet (Z. 76). Gleiches wird mit den niederwärtigsten 32 bit des Buffers durchgeführt (Z. 82). Die For-Schleife endet mit der Erreichung des übergebenen Wertes aus length.
Die Zeilen 89 bis 110 beschreiben den eigentlichen Testablauf. Dabei werden zu Anfang alle Steuersignale, bis auf das Signal RESET, auf den Wert null gesetzt (Z. 91). Daraufhin wird für 2 ms gewartet. Diese Zeit benötigt der Blockram für die Initialisierung. In Zeile 95 wird das Signal RESET ebenfalls auf den Wert null gesetzt und der Prozessor beginnt mit der Arbeit. Danach wird der JTAG TAP zurückgesetzt und der Prozessor für 200 µs ausgeführt, damit die Initialisierung beendet werden kann. Zum Abschluss wird zweimal der Task caeco_write_dmi aufgerufen, um die EKG Daten in den Caeco zu schreiben.

5.2.1 Programmzähler bei Initialisierung nach Reset

Durch die Testbench wird anfangs das Reset-Signal für 2 ms auf den Wert null gesetzt, danach beginnt der Prozessor mit der Abarbeitung der Instruktionen. Durch den Initialisierung-Code der crt0.S Datei und der main.c werden wichtige Größen wie der global pointer, stack pointer und der Speicherplatz für die Caecoergebnisse geladen und reserviert. Dieses Verhalten des Prozessors ist nach einem Zurücksetzen zu beobachten.

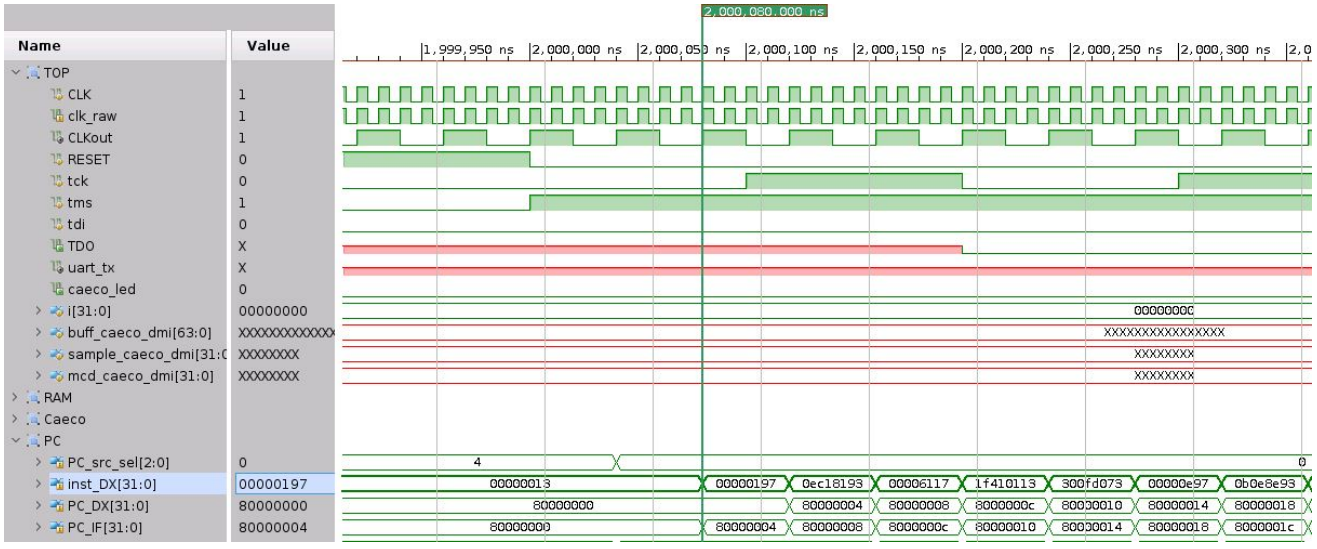

Abbildung 5.1: Simulationsüberblick und FPGA Ports

Dabei ist darauf zu achten, welche Instruktionen als erstes geladen und ausgeführt werden. In Abbildung 5.1 ist der Zeitpunkt nach dem Zurücksetzen des Prozessors abgebildet. Bei der markierten Zeit von 2 000 080 000 ns hat der aktuelle Programmzähler PC_DX den Wert 0x80000000 und lädt die Instruktion 00000197. Die nächste zu ladende Instruktion wird durch das Signal PC_IF angegeben und zeigt auf die Adresse 0x80000004, welche auf die Instruktion 0ec18193 zeigt. Vergleicht man diese Instruktionen mit denen aus dem kompilierten Programm-Code (vgl. Auflistung A.8 Z. 8 und 9), ist festzustellen, dass die erste Instruktion des Programmcodes korrekt geladen wird. Gleiches gilt für die folgenden Instruktionen.

5.2.2 Abgeschlossene Initialisierung „run!“

Die Initialisierung des Programm-Codes ist ab dem Zeitpunkt abgeschlossen, an dem innerhalb des C Programms die Endlosschleife aufgerufen wird (siehe Auflistung 4.2 Z. 34). Davor wird allerdings die Zeichenkette „run!“ per UART ausgegeben, wodurch das Ende der Initialisierung auch in der Simulation erkennbar gemacht wird. Das Zeichen “r“ wird ab der 30 237,48 µs übertragen, dargestellt in Abbildung 5.2.

Abbildung 5.2: Übertragung des Zeichens r per UART

Zuerst wird das Startbit übertragen, indem das Signal uart_tx auf den Wert null gesetzt wird. Folgend werden acht Datenbits gesendet, wobei mit dem LSB begonnen wird. Daraus resultiert das Byte mit dem Wert 01110010. Dies entspricht dem Dezimalwert von 114 und somit dem Buchstaben r des UCS Transformation Format (UTF)-8 Standard. Das Senden der gesamten Zeichenkette und damit der Abschluss der Initialisierung ist bzgl. einer erfolgreichen Speicherplatzreservierung bei 105 419,92 µs erreicht.

5.2.3 Schreiben der EKG Daten über das DMI

Nach der Initialisierung wartet der Prozessor auf einen durch den Caeco ausgelösten Interrupt. Wie bereits erläutert, resultiert der Interrupt aus einer abgeschlossenen Berechnung des Caecos. Dafür wird während der Initialisierung der Task caeco_write_dmi aufgerufen, sodass die EKG Daten in den Caecos geschrieben werden. Vor dem Schreiben der Daten erfolgt allerdings der Startbefehl, indem das Signal cmd des Caecos auf eins gesetzt wird. Dies beginnt ab 2207,35 µs abgebildet in Abbildung 5.3. Es ist zu erkennen, dass die Eingangssignale dmi_addr und dmi_wdata die Steuerwerte 0x23 und 0x1 haben. Damit wird auch das Ausgangssignal caeco_cmd im nächsten Takt auf den Wert eins gesetzt. Somit wird die Ansteuerung des DM über die JTAG Schnittstelle korrekt

umgesetzt. Das caeco_cmd Signal wird an das Caecointerface weitergeleitet (siehe Abbildung 5.4). Durch das Setzen des Eingangssignals dm_cmd des Caecointerfaces springt die Zustandsmaschine in den Zustand CTRL_DM und das Eingangssignal cmd des Caecos wird für einen Takt gesetzt.

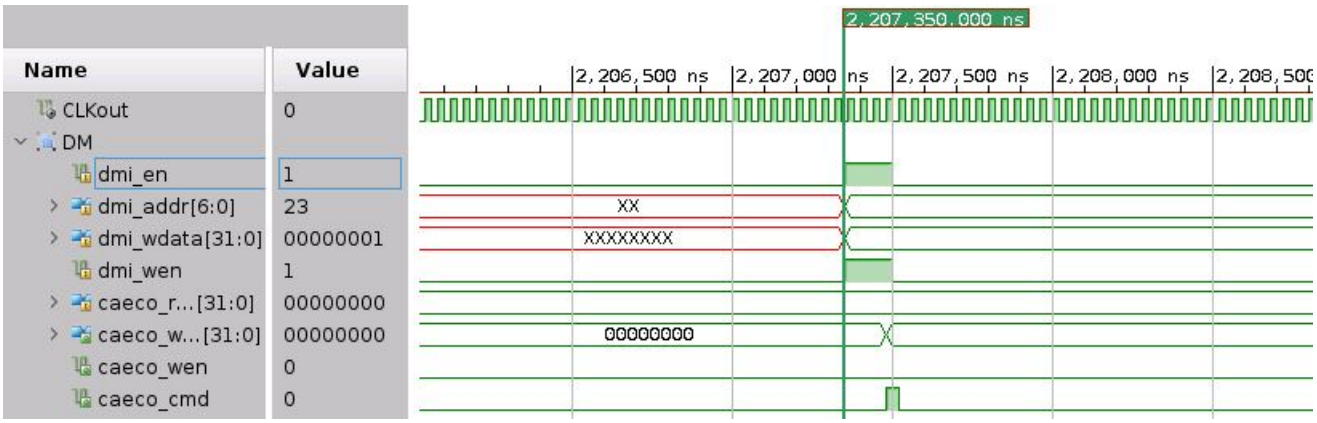

Abbildung 5.3: DMI Signale bei Schreiben des Caecos cmd

Einen Takt später wird der Beginn des Datenschreibens durch das Setzen des Signals led signalisiert. Nachdem das Startsignal erfolgreich an den Caeco gesendet wurde, kann mit dem Schreiben der Daten begonnen werden.

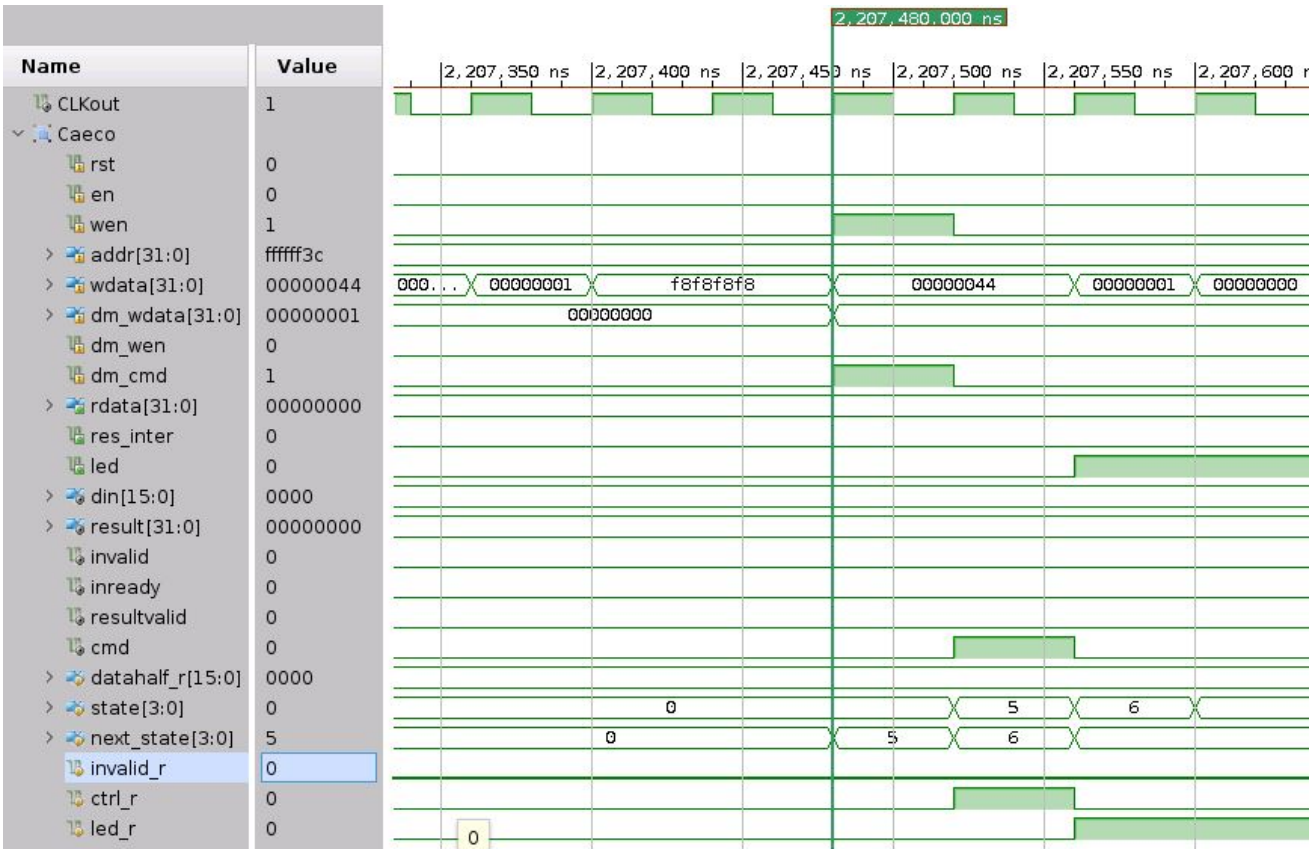

Abbildung 5.4: Caecointerface Signale bei Schreiben des Caecos cmd

Dabei bleibt das Signal led solange auf den Wert eins, bis der Caeco seine Berechnung abgeschlossen hat und somit das Signal resultvalid des Caecos gesetzt wird. In Abbildung 5.5 ist der Zeitpunkt vom Schreiben der ersten Daten bei 2213,4 µs abgebildet.
Zum markierten Zeitpunkt wird das DM Signal caeco_wen gesetzt, da die DMI Adresse dmi_addr den Wert 0x22 hat und damit Daten in den caeco geschrieben werden sollen. Die zu schreibenden Daten dmi_wdata haben den Wert 0xef08f307. Hierbei sind die höherwertigen 16 bits die Daten des ersten Kanals und die niederwertigsten 16 bits die Daten des zweiten Kanals. Die Reihenfolge der Bytes beider Datenwörter ist vertauscht, sodass sich die folgenden zuschreibenden Datenworte ergeben:

- Kanal 0: 0x08ef
- Kanal 1: 0x07f3

Durch das Setzen des Signals caeco_wen bzw. dm_wen springt die Zustandsmaschine des Caecointerfaces in den Zustand WDA0_DM, um die Daten des ersten Kanals zu schreiben. Dafür wird der Wert 0x08ef in das Register datahalf_r und geschrieben. Dadurch liegen die Daten am Eingangssignal din des Caecos an. Außerdem wird das Signal invalid gesetzt, um die Daten am Eingang des Caecos tatsächlich zu schreiben. Im nächsten Takt springt die Zustandsmaschine in den Zustand WDA1_DM und das zweite Datenwort 0x07f3 wird geschrieben.

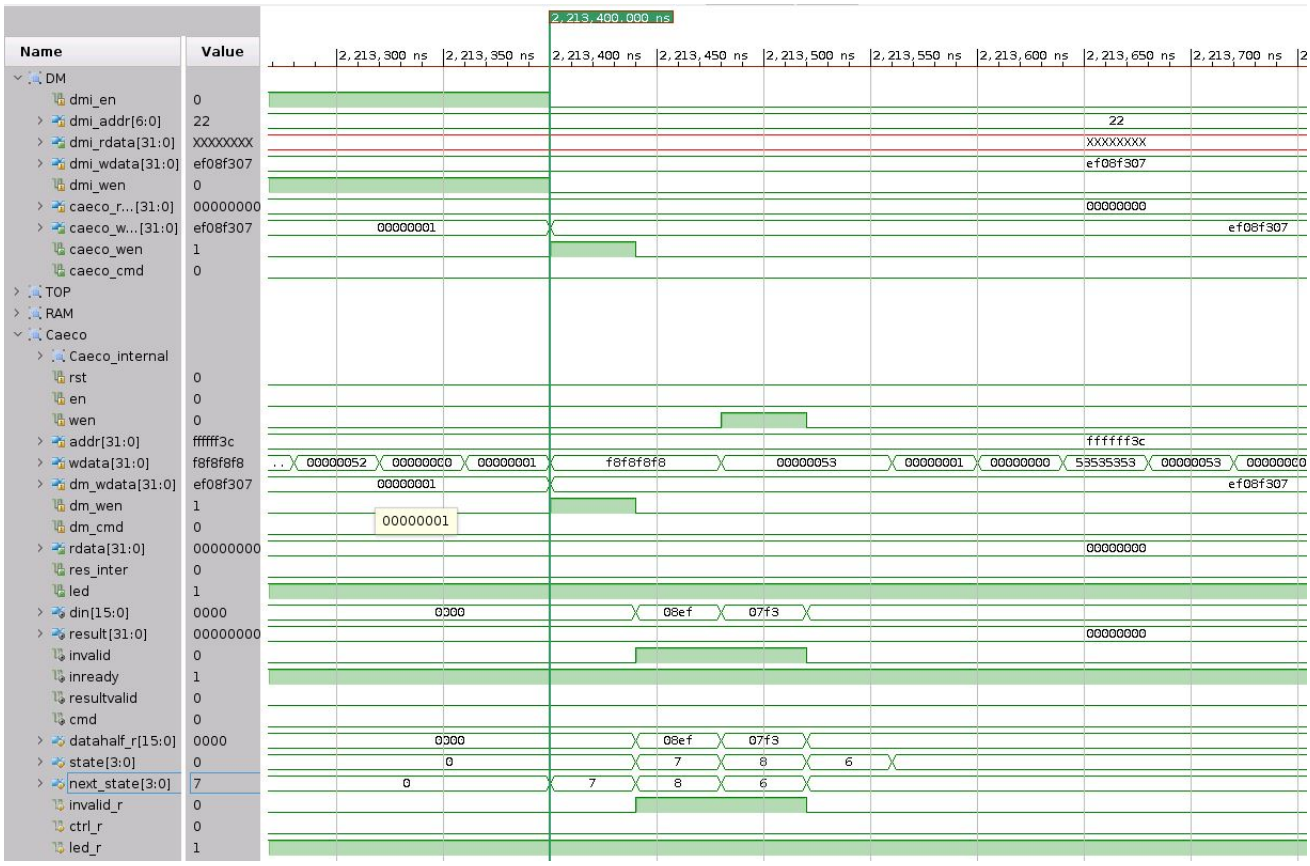

Abbildung 5.5: DMI und Caecointerface Signale bei Schreiben der Daten

Danach wird das Signal invalid auf den Wert null gesetzt. Damit ist der Schreibvorgang des ersten Datenwortes abgeschlossen. Durch eine Optimierung innerhalb des Caecos benötigt dieser aktuell nicht den kompletten Datensatz, um eine Berechnung durchzuführen. So werden für die vorliegende Testbench 68 288 bits geschrieben. Dies ist nach 14,7981 ms abgeschlossen.

5.2.4 Interrupt-Verhalten bei gültigem Ergebnis des Caecos

Solange der Caeco kein gültiges Ergebnis vorliegen hat, arbeitet der Prozessor im Stromsparmodus und überprüft, ob der Ergebnisspeicher beschrieben wurde. Bei einem gültigen Ergebnis setzt der Caeco das Signal resultvalid und ein Interrupt wird ausgelöst. Dadurch wird die ISR *l_1_caeco_interrupt_handler()* abgearbeitet und das Ergebnis in den reservierter Speicher geschrieben. Der Zeitpunkt des Interrupts ist demnach genauer zu untersuchen. Dabei ist entscheidend, ob die Sprungadresse korrekt gewählt, die Rücksprungadresse korrekt gespeichert und das Ergebnis per UART ausgegeben wird. In Abbildung 5.6 ist der Zeitpunkt der Ergebnisausgabe bzw. des Interrupt bei 53,245 32 ms mit den Signalen des Caecointerfaces abgebildet.

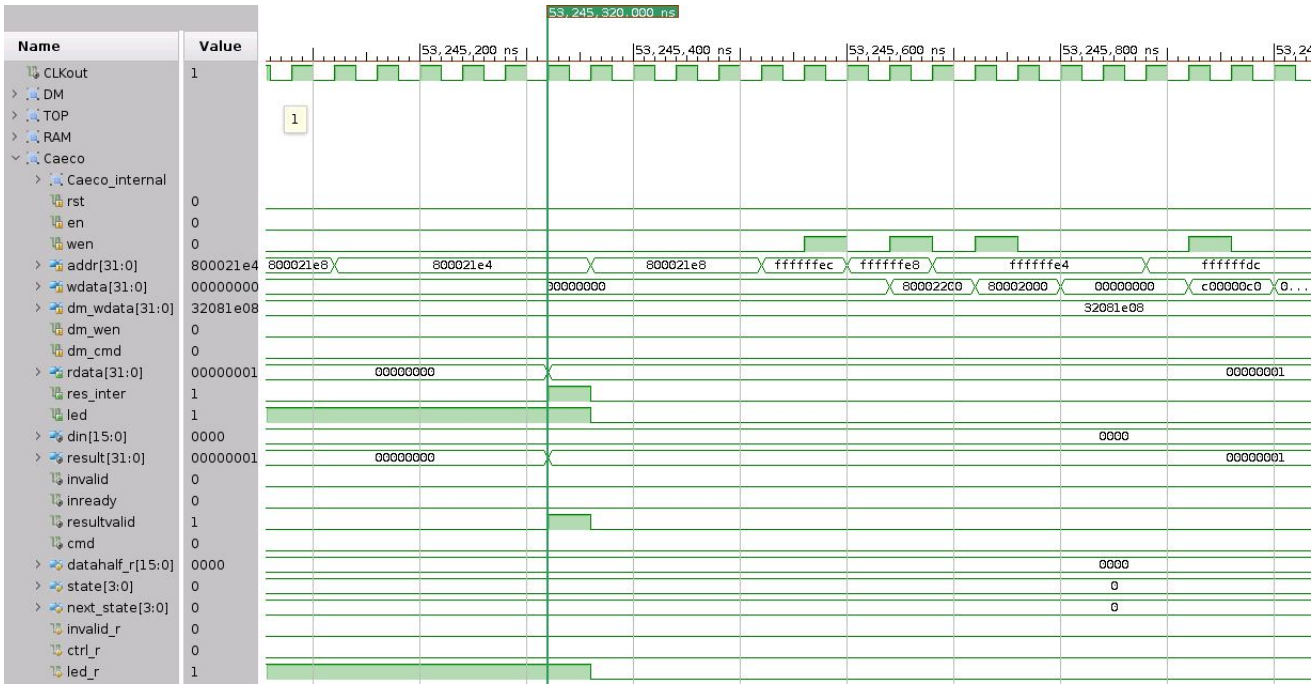

Abbildung 5.6: Caecointerfacesignale bei Ergebnis

Nach Abschluss der Berechnung wird das Ausgangssignal resultvalid auf den Wert eins gesetzt und das Ergebnis in das Register rdata geschrieben. Mit dem nächsten Takt wird das Signal led auf den Wert null gesetzt, um das Ende der Rechnung zu signalisieren. Um den Interrupt auszulösen, wird das Signal resultvalid über das Signal res_inter an den Prozessor weitergeleitet. Die Signale des Prozessors zum selben Zeitpunkt sind in Abbildung 5.7 dargestellt.

Der Interrupt des Caecos ist an den Signalen ext_interrupts, interrupt_pending und interrupt_taken zu erkennen. Durch den Interrupt wird das LSB des Signals ext_interrupts gesetzt. Dadurch wird der Interrupt ausgelöst und die Signale interrupt_pending und interrupt_taken werden ebenfalls gesetzt. Der Prozessor ermittelt daraufhin die Sprung- und Rücksprungadresse. Für die Ermittlung der Sprungadresse wird die bereits in Auflistung 3.20 erläuterte Fallunterscheidung vollzogen. Die Adresse des Trap-Handler wurde vorher in das Register mtvec geschrieben und hat den Wert 0x800000c4 (siehe Anhang A.3.3, Auflistung A.8 Z. 62). Für den vorliegenden Interrupt ist die Zieladresse zum *l_1_caeco_interrupt_handler()* somit 0x800000c. Diese wird einen Takt später korrekt als neuer Programmzähler für die nächste Instruktion gesetzt. Wiederum einen Takt später liegt die Adresse im aktuellen Programmzähler an und der Sprungbefehl 0x27c0006f zum *l_1_caeco_interrupt_handler()* wird ausgeführt. Dadurch wird als nächstes die Instruktion an der Adresse 0x80000344 geladen, welche den Beginn des *l_1_caeco_interrupt_handler()* markiert (siehe Anhang A.3.3, Auflistung A.8 Z. 69).

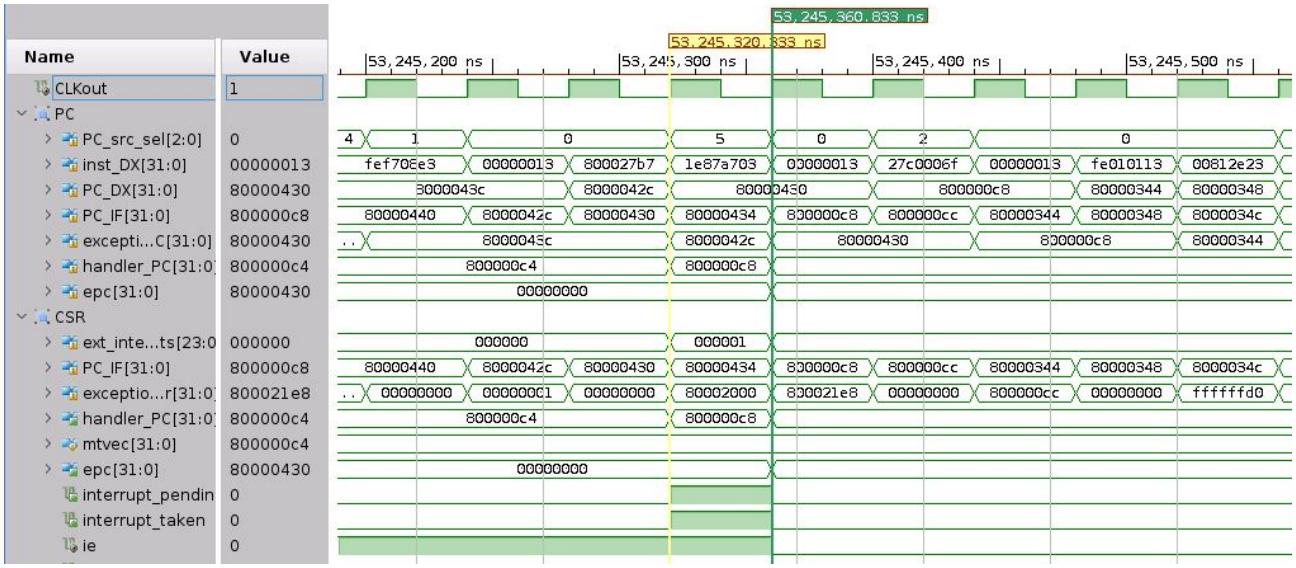

Abbildung 5.7: Prozessorsignale bei Ergebnis

Die Rücksprungadresse, zu der nach dem Ende des Interrupts gesprungen wird, wird im Register epc gespeichert und hat den Wert 0x80000430. Zum Zeitpunkt des Interrupts ist dies die Adresse, bei der Instruktion 0x1e87a703. Diese Instruktion wird allerdings nicht ausgeführt, sodass nach dem Interrupt der Programm-Code an dieser Stelle weitergeführt werden muss. Damit wird auch die Rücksprungadresse korrekt gespeichert.
Der Rücksprung zu der *main()* Funktion erfolgt nach 1680 ns. Da nun ein Ergebnis in den Speicher geschrieben worden ist, wird die If-Bedingung der *main()* erfüllt und das Ergebnis wird über die UART Schnittstelle ausgegeben (siehe Auflistung 4.2, Z. 35). In Abbildung 5.8 ist erkennbar, dass das Signal uart_tx kurz nach dem Interrupt anfängt zu arbeiten und insgesamt 14 Zeichen ausgibt. Diese Zeichen entsprechen der Zeichenkette der *main()* Funktion „\n result: 1\n“.

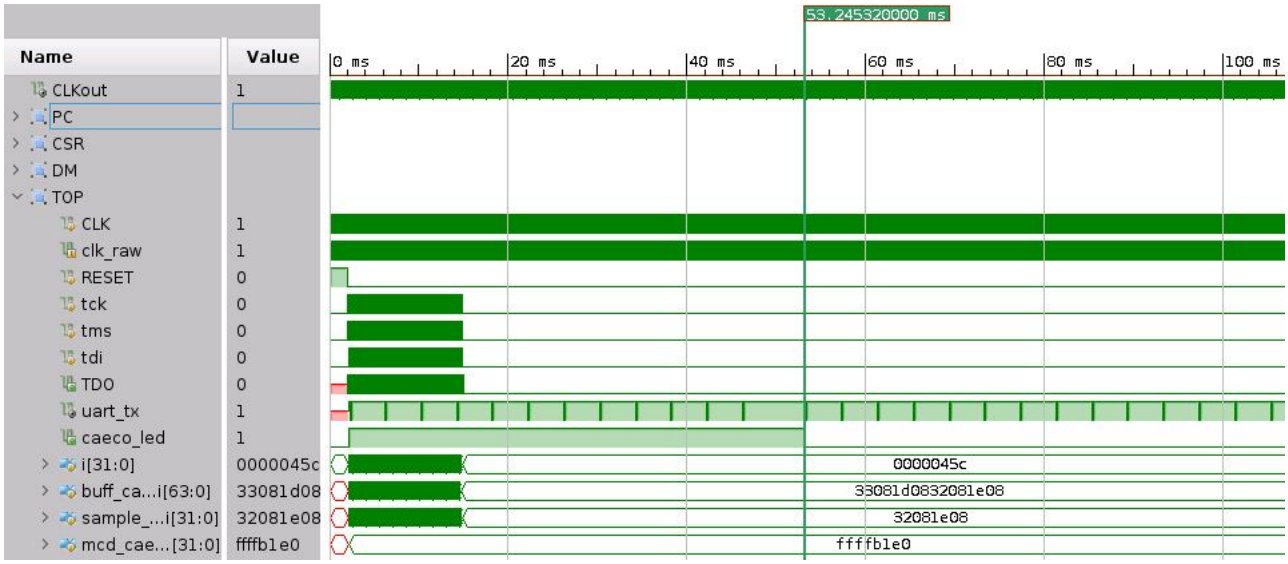

Abbildung 5.8: UART Verhalten bei Ergebnis

Dadurch ist ebenfalls verifiziert, dass die ISR korrekt ausgeführt und das Ergebnis über die UART Schnittstelle übertragen wird. Dies kann unter anderem dazu genutzt werden, um die Ergebnisse des Caecos außerhalb des Prozessors zu speichern oder weiter zu verarbeiten. Abschließend ist festzuhalten das sowohl der Caeco als auch der Prozessor das erwartete Verhalten aufzeigen und dadurch das Zusammenspiel von Hard- und Software gegeben ist. Die Ansteuerung und das Beschreiben des Caecos mit Daten über die JTAG Schnittstelle ist erfolgreich implementiert. Bei Fertigstellung eines Rechenergebnis durch den Caeco wird wie gewünscht ein Hardware-Interrupt ausgelöst, wodurch in erster Instanz der Trap-Handler aufgerufen wird, der wiederum die ISR aufruft. Hierbei wird die korrekte Rücksprungadresse gespeichert und der Prozessor arbeitet nach dem Interrupt in der *main()* weiter. Die Ausgabe des Ergebnisses über die UART Schnittstelle funktioniert ebenfalls für zwei simulierte Interrupts.
Im folgenden Abschnitt wird das implementierte Design auf dem FPGA geprüft. Hierbei wird der Interrupt allerdings nicht durch den Caeco, sondern durch einen Schalter auf der Entwicklungsplatine ausgelöst, damit das Interrupt-Verhalten verifiziert werden kann.

5.3 Hardware-Verifizierung FPGA Design

Zusätzlich zu der RTL Simulation wird das Design auf dem FPGA implementiert und getestet. Hierbei werden vor der Implementierung die Änderungen aus Abschnitt 3.2.6 eingefügt, sodass ein manueller Interrupt durch den Schalter 16 ausgelöst werden kann.
Im ersten Schritt wird das Design durch eine RTL Simulation geprüft. Hierbei entspricht die Testbench in großen Teilen der Testbench aus Auflistung 5.4. Allerdings werden die beiden Tasks zum Schreiben des Caecos durch eine For-Schleife ersetzt, durch die das Signal ext_inter gesetzt wird und so ein manueller Interrupt mehrfach simuliert wird. Die Änderungen der Testbench sind

in Auflistung 5.6 dargestellt. Zusätzlich wird dem DUT der zusätzliche Port ext_inter hinzugefügt. Nach der Zurücksetzung des Prozessors für 2 ms wird für 105 ms gewartet, damit die Initialisierung abgeschlossen werden kann. Danach erfolgt die Simulation des manuellen Interrupts durch die For-Schleife.

```
1  //------------------------------------------ Testbench start!
2  initial begin
3      $write("Testbench is starting! \n");
4      RESET <= 1'b1; tms <= 1'b0; tdi <= 1'b0; tck <= 1'b0; i <= 32'h0;
5      // reset the design for 2 ms to wait for the xilinx ip bram
6      $write("Wait for BRAM \n");
7      #2000000;
8      RESET <= 1'b0;
9      $write("JTAG TAP: reset..\n");
10     jtag_tap_reset;
11     $write("Initializing finished! Now starting with the testcases \n");
12     #60000000;
13
14     for (i = 0 ; i < 5 ; i = i +1) begin
15         ext_inter<=1'b1;
16         #500;
17         ext_inter<=1'b0;
18         #60000000;
19     end
20     $finish();
21 end
22 endmodule
```

Auflistung 5.6: Testbench für manuellen Interrupt

Das Ergebnis der Simulation ist in Abbildung 5.9 dargestellt. Hierbei wird das genaue Verhalten des Prozessors nicht erneut analysiert, da sich weder etwas an dem Programm-Code noch an der Implementierung bzgl. des Interrupts geändert hat. Hauptsächlich wird geprüft, ob der manuelle Interrupt durch die Betätigung des Schalters ausgelöst wird. Gleich zu Beginn werden analog zu der Simulation aus dem vorherigen Abschnitt Daten über die UART Schnittstelle gesendet. Dies entspricht dem Initialisierungsvorgang und ist nach ca. 46 ms abgeschlossen. Danach wird das Signal ext_inter gesetzt, die ISR wird aufgerufen und über die UART Schnittstelle wird das Ergebnis ausgegeben. Da der Caeco nicht arbeitet, entspricht das Ergebnis in dieser Simulation immer dem Wert null, da dies der Wert ist, mit dem das Register des Caecos initialisiert wird. Nach der Simulation wird das Design unter Vivado synthetisiert und für die Entwicklungsplatine Nexys4 DDR implementiert.

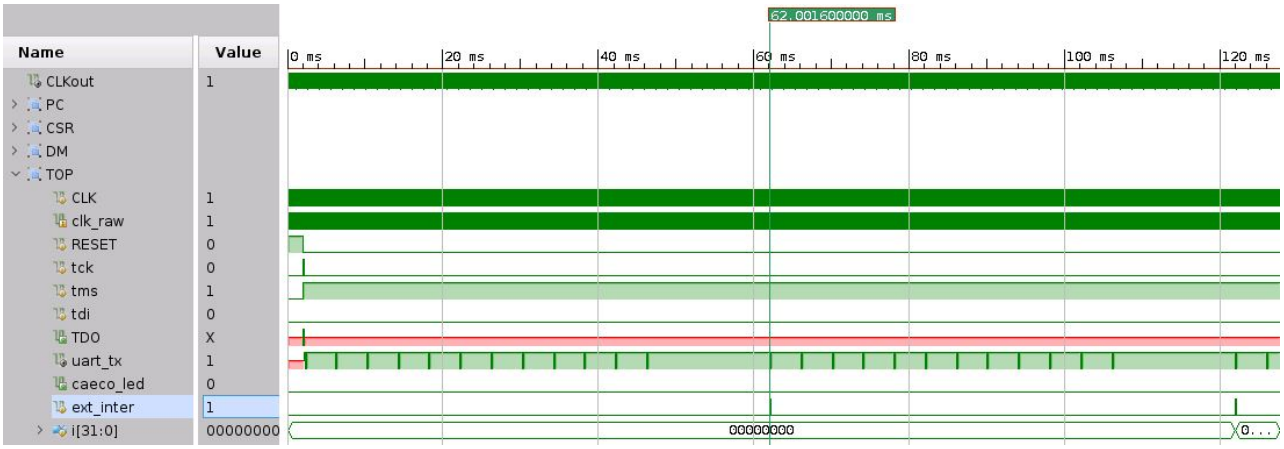

Abbildung 5.9: Signale FPGA bei manuellem Interrupt

Damit kann der FPGA programmiert und auf das Design auf der Hardware getestet werden. Für den Empfang der Daten per UART wird das Programm HTerm 0.8.5 verwendet. In Abbildung 5.10 ist der Empfang der Daten über HTerm abgebildet. Hierbei ist der UART Port der Entwicklungsplatine bei der lokalen Entwicklungsmaschine als COM5 Port angeschlossen. Die Baudrate beträgt 115200 baud und es werden achten Daten- und zwei Stopbits gesendet.

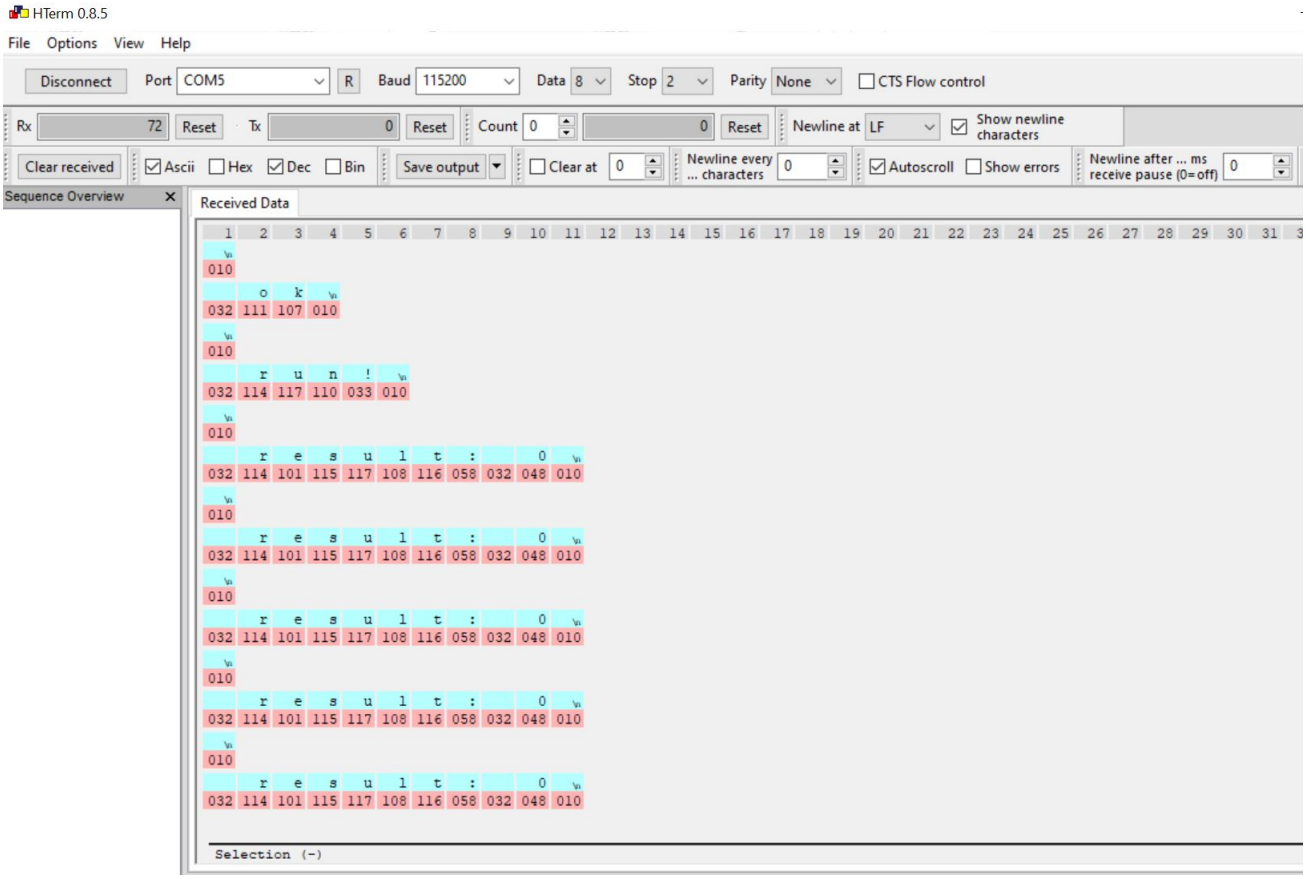

Abbildung 5.10: Empfang der Daten beim Hardwaretest

Nach einem Zurücksetzen des Prozessors werden die Zeichenketten „\n ok\n“ und „\n run!\n“ direkt übertragen (siehe dazu Auflistung 4.2 Z. 27 und Z. 33). Nach jeder Auslösung des Schalters wird die ISR aufgerufen, durch die das Ergebnisregister des Caecos ausgelesen und und die Zeichenkette „\n result: 0\n“ über die UART Schnittstelle gesendet wird.

Zusammenfassend ist festzustellen, dass das Interrupt-Verhalten des Prozessors erfolgreich geprüft worden ist. Bei der Initialisierung werden die erwarteten Zeichenketten gesendet. Ein Interrupt lässt sich durch die Betätigung des Schalters mehrfach hintereinander erzeugen und führt immer zum selben Ergebnis. Auch nach der erneuten Zurücksetzung des Prozessors läuft der Betrieb weiterhin stabil. In den folgendem Abschnitt wird auf das Debugging des Prozessors unter Eclipse als zusätzlicher Verifizierungsschritt eingegangen.

5.4 Debugging unter Eclipse

Das Debugging unter Eclipse erlaubt zum einen die Nutzung genereller Debug-Möglichkeiten des Prozessors bzgl. des stepmode (*Schritt-für-Schritt*) Debugging und das Setzen von breakpoints (*Haltepunkte*). Zusätzlich dazu kann die Initialisierung genauer beobachtet werden. Die Möglichkeit den Prozessor während eines Interrupts zu beobachten ist nicht gegeben, da im Debug-Modus keine Interrupts abgearbeitet werden. Dazu wird der FPGA unter Vivado 2020.1 zuerst mit dem implementierten Zieldesign programmiert. Für das Debugging ist die Entwicklungsplatine zum einen durch die UART Schnittstelle, zum anderen durch den JTAG Adapter mit der lokalen Entwicklungsmaschine verbunden. Während des Debuggings ist das Programm HTerm aktiv, um die per UART Schnittstelle übertragenen Daten zu empfangen. Sind alle Bedingungen erfüllt, wird in Eclipse IDE das Debugging gestartet. Der erste breakpoint wird in Zeile 45 der main.c gesetzt, da in dieser Zeile der erste *printf()* Befehl aufgerufen wird, der bei einer erfolgreichen Speicherplatzreservierung ausgeführt wird (siehe Abbildung 5.11). Die linke Spalte zeigt den Programm-Code in C, die rechte Spalte die aktuellen Instruktionen. Dadurch dass die Speichereservierung erfolgreich war, wird der *printf()* Aufruf in Zeile 45 ausgeführt. Aufgrund des gesetzten breakpoints stoppt die Programmausführung in Zeile 45. Bis zu diesem Zeitpunkt sind keine Daten per UART Schnittstelle geschickt bzw. empfangen worden. Lässt man dann den Programm-Code weiterlaufen, wird die Zeichenkette ausgegeben. Neben der resume (*fortsetzen*) Funktion für das Debugging, durch die der Programm-Code bis zum nächsten breakpoint ausgeführt wird, existiert noch die step-into (*Schritt-für-Schritt*) Funktion. Dadurch kann von Instruktion zu Instruktion gesprungen werden, was ein genaues Debbuging ermöglicht. Ist der Befehl in Gänze abgearbeitet worden, so wird auch die Zeichenkette übertragen und unter HTerm empfangen (siehe Abbildung 5.12).
Damit ist nachgewiesen, dass das Debugging des Prozessors möglich ist. Die Grenze des Debugging liegt allerdings darin, dass der Aufruf und die Ausführung der ISR nicht beobachtet werden kann, ähnlich wie bei der Simulation durch den riscvOVPsim Instruktionssimulator.

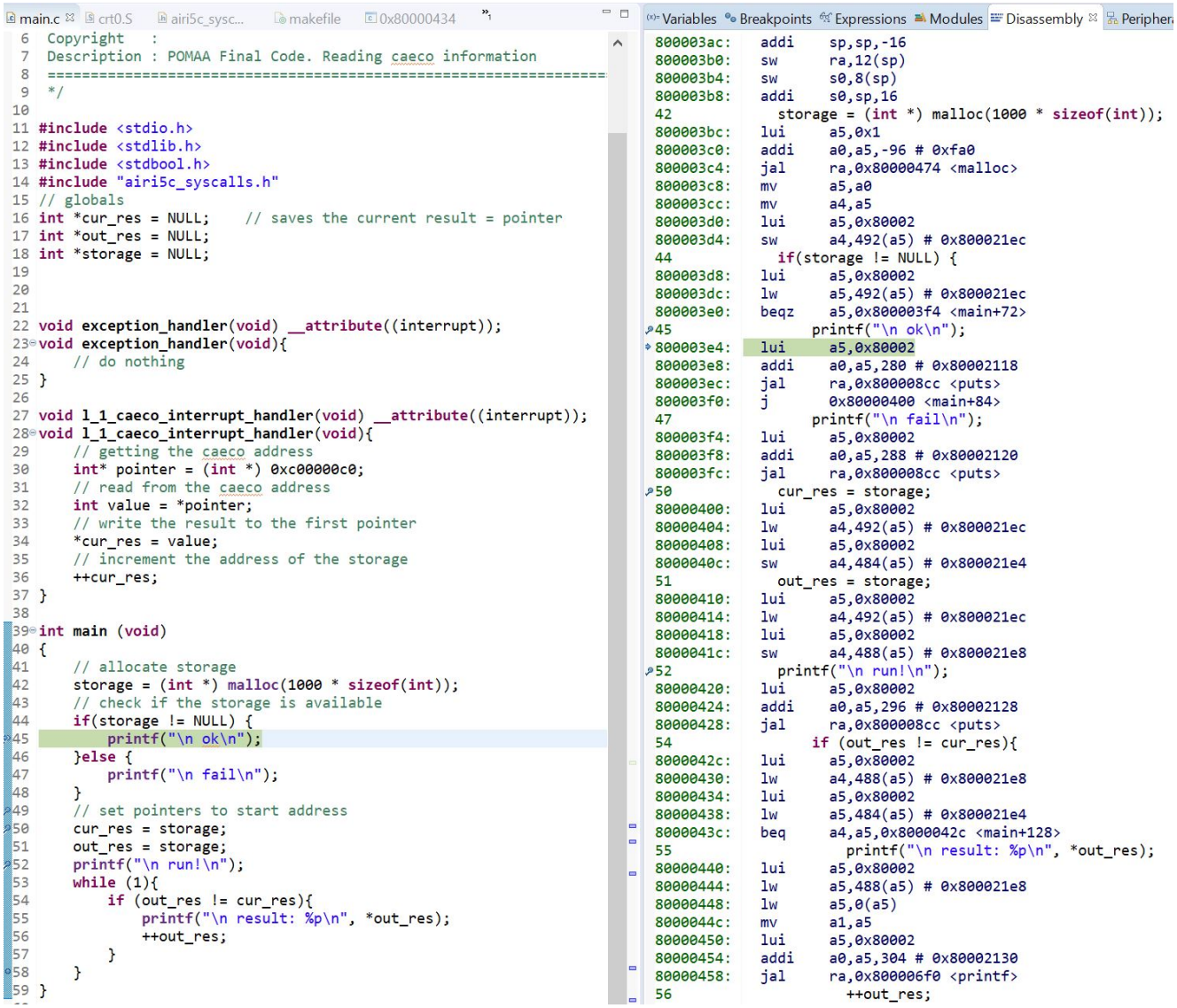

Abbildung 5.11: Breakpoint nach Speicherplatzreservierung

Abbildung 5.12: HTerm Ausgabe der ersten Zeichenkette

Dies liegt daran, dass ein Debug-Zugriff immer eine höhere Priorität als ein Interrupt besitzt und damit nicht auf Interrupts reagiert. Im Zuge dieser Arbeit wurde dies dadurch getestet, dass das

Design aus Abschnitt 5.3 implementiert worden ist, und während des Debbuging eine manueller Interrupt ausgelöst wurde. Die ISR wurde dadurch allerdings nicht ausgelöst. Trotzdem ist die Debug-Funktion für die Verifizierung des Prozessors ein notwendiges Werkzeug und kann für die punktuelle Verifizierung der Funktionalität und des Programm-Codes genutzt werden.

6 Zusammenfassung und zukünftige Arbeiten

Abschließend kann festgehalten werden, dass die Integration des Caecos in den Raifes RV32IM erfolgreich simuliert und auf der Hardware verifiziert wurde. Dazu wurde das Design mit dem implementierten Caeco inklusive der Übertragung von Daten über die JTAG Schnittstelle auf RTL Ebene simuliert. Die Ergebnisse zeigen, dass das Schreiben der EKG Daten parallel zum eigentlichen Prozessorbetrieb durchgeführt werden kann. Bei einem gültigen Ergebnis des Caecos wird der Trap-Handler aufgerufen und damit auch die ISR. Danach setzt der Prozessor seine Arbeit an gewünschter Stelle fort. Für die Verifizierung des Design auf dem FPGA wurde der Caeco Interrupt durch einen manuellen Schalter der Entwicklungsplatine ersetzt und mehrmals hintereinander ausgelöst. Wie gewünscht ist das der Simulation beobachtete Verhalten zu erkennen gewesen. Die gesamte Hardware-Entwicklung wurde durch die Gitlab DevOps automatisiert. Dabei sind die fünf Pipeline Stages erfolgreich durchlaufen worden, bei denen die HDL Dateien durch Vivado 2020.1 im Batch Project Flow Modus zu einem Projekt erstellt, synthetisiert, implementiert und anschließend simuliert wurden. Dadurch wird der Entwicklungsprozess optimiert und kann für spätere Projekte ausgebaut und verfeinert werden. Das gesamte Design umfasst 5015 LTUs, von denen 1052 zum Caeco gehören.
Ein lauffähiger Programm-Code, der in C und im RISC-V Assembler entwickelt wurde, ist durch den Instruktionssimulator riscvOVPsim getestet und daraufhin in der RTL Simulation verwendet worden. Das Programm erwartet einen 256 kB großen Speicher, der durch das Linkerskript angepasst werden kann. Es werden bei der Initialisierung 4 kB für die Ergebnisse des Caecos reserviert. Für die Kompilierung des Programms in der Eclipse IDE wird die xPack riscv-none-embed-gcc v8.3.0 Toolchain verwendet. Um die ELF Datei für die Xilinx Speicher nutzbar zu machen, wurden zwei Rust Programme entwickelt, durch die der Code in eine MEM bzw COE Datei umgewandelt werden kann. Auch das Debugging wurde unter Eclipse erfolgreich durchgeführt, wobei das Setzen von breakpoints und der step-into Modus ebenfalls getestet wurde.

Für zukünftige Arbeiten der Test der EKG Datenübertragung über die JTAG Schnittstelle und das dazugehörige Verhalten des Caecos notwendig. Dies wurde in dieser Arbeit auf der FPGA Ebene nicht verifiziert. Außerdem wird das Debugging dadurch erschwert, dass während eines Debugzugriffs externe Interrupts, wie die des Caecos, nicht berücksichtigt werden und dadurch das Verhalten bei einer Trap nicht untersucht werden kann. Hier muss einerseits natürlich an die RISC-V Spezifikation berücksichtigt werden, allerdings wäre eine Debug-Möglichkeit für Interrupts eine hilfreiche Erweiterung für spätere Implementierungen, da einen wesentlicher Bestandteil von Software-Lösungen in eingebetteten Systemen darstellen. Ein weiterer Punkt ist die Verbesserung der Gitlab CI/CD Struktur. Hier können zusätzliche Möglichkeiten der Verifizierung implementiert

werden, z.B. eine automatisierte Auswertung der Simulationsergebnisse. Parallel dazu kann die Testbench so erweitert werden, das vor allem die Werte der Allzweckregister bei jeder Instruktion geloggt werden, sodass eine Vergleichsgrundlage zu anderen Simulationsmodellen geschaffen wird. Abschließend besteht die Möglichkeit am Simulationsansatz über riscvOVPsim anzuknüpfen und die Verifizierung weiter auszubauen, indem z.B. vorhandene Debug-Funktionen genutzt werden. Es wäre auch möglich, die Prozessorarchitektur des Raifes RV32IM in den Simulator einzubinden, sodass genauere Untersuchungen stattfinden können. Im Vergleich dazu könnte man andere Simulatoren wie z.B. QEMO testen und Unterschiede in der Performance oder in den vorhandenen Möglichkeiten aufzeigen.

Literaturverzeichnis

[1] Kirchner, Dr. F.: *POMAA: Pareto-Optimaler MAchine Learning ASIC*. https://robotik.dfki-bremen.de/de/forschung/projekte/pomaa/. – Letzter Zugriff: 21.07.2020

[2] Brinkschulte, Uwe ; Ungerer, Theo: *Mikrocontroller und Mikroprozessoren*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2010. http://dx.doi.org/10.1007/978-3-642-05398-6. http://dx.doi.org/10.1007/978-3-642-05398-6

[3] Wüst, Klaus (Hrsg.): *Mikroprozessortechnik: Grundlagen, Architekturen, Schaltungstechnik und Betrieb von Mikroprozessoren und Mikrocontrollern*. 4., aktualisierte und erweiterte Auflage. Wiesbaden : Vieweg Teubner Verlag / Springer Fachmedien Wiesbaden GmbH Wiesbaden, 2011 (Studium). http://dx.doi.org/10.1007/978-3-8348-9881-4. http://dx.doi.org/10.1007/978-3-8348-9881-4. – ISBN 978–3–8348–0906–3

[4] Bähring, Helmut (Hrsg.): *Anwendungsorientierte Mikroprozessoren: Mikrocontroller und Digitale Signalprozessoren*. 4., vollst. überarb. Aufl. Berlin : Springer, 2010 (EXamen.press). – ISBN 978–3–642–12291–0

[5] Böttcher, Axel (Hrsg.): *Rechneraufbau und Rechnerarchitektur*. Berlin, Heidelberg : Springer-Verlag Berlin Heidelberg, 2006 (EXamen.press). – ISBN 10 3–540–20979–4

[6] Andrew Waterman ; Krste Asanovic ; SiFi Inc. (Hrsg.): *The RISC-V Instruction Set Manual: Volume I: Unprivileged ISA*. Berkely, 2019 (20190608). http://www-inst.eecs.berkeley.edu/~cs152/sp19/handouts/sp19/riscv-spec-rvv-v0p4.pdf

[7] Tim Newsome ; Megan Wachs ; SiFive Inc. (Hrsg.): *RISC-V External Debug Support Version 0.13.2*. Berkely : RISC-V, 2019 https://riscv.org/specifications/debug-specification/. – Letzter Zugriff: 13.01.2020

[8] IEEE Standard Test Access Port and Boundary Scan Architecture. In: *IEEE Std 1149.1-2001* (2001), S. 1–212

[9] Yunsup, Lee ; Albert, Ou ; Albert, Magyar ; EECS Barkeley (Hrsg.): *Z-scale: Tiny 32-bit RISC-V Systems: With Updates to the Rocket Chip Generator*. California, 2015

[10] Andrew Waterman ; Krste Asanovic ; SiFi Inc. (Hrsg.): *The RISC-V Instruction Set Manual: Volume II: Privileged Architecture*. Berkely, 2019 (2019060). https://riscv.org/specifications/privileged-isa/

[11] Engineering, UC Berkeley C.: *EECS150: Finite State Machines in Verilog*. https://inst.eecs.berkeley.edu/~cs150/sp12/resources/FSM.pdf

[12] Digilent: *Nexys 4 DDR XDC*. https://github.com/Digilent/digilent-xdc/. – Letzter Zugriff: 23.07.2020

[13] Digilent: *Nexys 4 DDR*. https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/start. – Letzter Zugriff: 23.07.2020

[14] Batard, Pete: *Zadig*. https://zadig.akeo.ie/. – Letzter Zugriff: 20.07.2020

[15] Vadapalli, Sricharan: *DevOps: Continuous Delivery, Integration, and Deployment with DevOps : Dive Into the Core DevOps Strategies*. "Packt Publishing", 2018. – ISBN 9781789131253

[16] Soni, Mitesh: *DevOps Bootcamp*. Packt Publishing, 2017 https://widgets.ebscohost.com/prod/customerspecific/s9218820/vpn/vpn_fhdo.php?url=http://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=1528138&lang=de&site=eds-live&scope=site. – ISBN 9781787285965

[17] Gitlab: *GitLab CI/CD Pipeline Configuration Reference*. https://docs.gitlab.com/ee/ci/yaml/. https://docs.gitlab.com/ee/ci/yaml/. Version: 2020. – Letzter Zugriff: 05.05.2020

[18] Gitlab: *Configuring GitLab Runners*. https://gitlab.com/help/ci/runners/README. – Letzter Zugriff: 21.07.2020

[19] Gitlab: *GitLab CI/CD pipeline configuration reference*. https://docs.gitlab.com/ee/ci/yaml/. – Letzter Zugriff: 21.07.2020

[20] SiFive: *GNU Embedded Toolchain — v2019.08.0*. https://www.sifive.com/boards. – Letzter Zugriff: 25.07.2020

[21] Ionescu, Liviu: *data2mem download*. https://xpack.github.io/riscv-none-embed-gcc/#install. – Letzter Zugriff: 16.07.2020

[22] Project, The O. ; The OpenOCD Project (Hrsg.): *Open On-Chip Debugger: OpenOCD User's Guide*. 2019

[23] Inc., Xilinx®: *data2mem download*. https://www.pconlife.com/viewfileinfo/data2mem-exe/. – Letzter Zugriff: 20.07.2020

[24] Chamberlain, Steve ; Pesch, Roland ; Johnston, Jeff: *The Red Hat newlib C Library - Full Configuration*. ftp://sources.redhat.com/pub/newlib/libc.pdf. Version: 2016

[25] Chamberlain, Steve ; Taylor, Ian L.: *The GNU linker*, 2010. – Version 2.19.51

[26] Chamberlain, Steve: *Using ld*. https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_toc.html. Version: Nov 1998

[27] Gu, Changyi: *Building Embedded Systems - Programmable Hardware*. https://www.apress.com/gp/book/9781484219188

[28] Free Software Foundation, Inc.: *GNU Free Documentation License Version 1.3*. https://sourceware.org/binutils/docs-2.31/as/index.html#Top, 2008

[29] Foundation, Free S.: *GNU assembler as (GNU Binutils) version 2.34 -CFI directives*. https://sourceware.org/binutils/docs/as/CFI-directives.html. Version: Nov 2008

[30] Darwin, Ian F. ; Vixie, Paul: *Chapter 8. Exception Frames*. https://refspecs.linuxfoundation.org/LSB_3.0.0/LSB-PDA/LSB-PDA/ehframechpt.html. Version: 2004

[31] Inc., Xilinx®: *UG898: Vivado Design Suite User Guide: Embedded Processor Hardware Design*. v2019.1. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug898-vivado-embedded-design.pdf, Juny 2019. – Letzter Zugriff: 19.07.2020

[32] Inc., Xilinx®: *COE File Syntax*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/cgn_r_coe_file_syntax.htm. – Letzter Zugriff: 20.07.2020

[33] Impreas: *riscvOVPsim*. https://github.com/riscv/riscv-ovpsim. – Letzter Zugriff: 23.07.2020

A Dateien

A.1 Linker-Skripte

A.1.1 Standard Linker Skript der riscv-none-embed-gcc 8.3.0-1.1 Toolchain

```
GNU ld (xPack GNU RISC-V Embedded GCC, 64-bit) 2.32
  Supported emulations:
   elf32lriscv
   elf64lriscv
using internal linker script:
==================================================
/* Script for -z combreloc: combine and sort reloc sections */
/* Copyright (C) 2014-2019 Free Software Foundation, Inc.
   Copying and distribution of this script, with or without modification,
   are permitted in any medium without royalty provided the copyright
   notice and this notice are preserved.  */
OUTPUT_FORMAT("elf32-littleriscv", "elf32-littleriscv",
      "elf32-littleriscv")
OUTPUT_ARCH(riscv)
ENTRY(_start)
SEARCH_DIR("=/Host/home/ilg/Work/riscv-none-embed-gcc-8.3.0-1.1/linux-x64/
    install/riscv-none-embed-gcc/riscv-none-embed/lib");
    SEARCH_DIR("=/usr/local/lib"); SEARCH_DIR("=/lib");
    SEARCH_DIR("=/usr/lib");
SECTIONS
{
  /* Read-only sections, merged into text segment: */
  PROVIDE (__executable_start = SEGMENT_START("text-segment", 0x10000)); .
    = SEGMENT_START("text-segment", 0x10000) + SIZEOF_HEADERS;
  .interp         : { *(.interp) }
  .note.gnu.build-id  : { *(.note.gnu.build-id) }
  .hash           : { *(.hash) }
  .gnu.hash       : { *(.gnu.hash) }
  .dynsym         : { *(.dynsym) }
  .dynstr         : { *(.dynstr) }
  .gnu.version    : { *(.gnu.version) }
```

```
28  .gnu.version_d   : { *(.gnu.version_d) }
29  .gnu.version_r   : { *(.gnu.version_r) }
30  .rela.dyn        :
31    {
32      *(.rela.init)
33      *(.rela.text .rela.text.* .rela.gnu.linkonce.t.*)
34      *(.rela.fini)
35      *(.rela.rodata .rela.rodata.* .rela.gnu.linkonce.r.*)
36      *(.rela.data .rela.data.* .rela.gnu.linkonce.d.*)
37      *(.rela.tdata .rela.tdata.* .rela.gnu.linkonce.td.*)
38      *(.rela.tbss .rela.tbss.* .rela.gnu.linkonce.tb.*)
39      *(.rela.ctors)
40      *(.rela.dtors)
41      *(.rela.got)
42      *(.rela.sdata .rela.sdata.* .rela.gnu.linkonce.s.*)
43      *(.rela.sbss .rela.sbss.* .rela.gnu.linkonce.sb.*)
44      *(.rela.sdata2 .rela.sdata2.* .rela.gnu.linkonce.s2.*)
45      *(.rela.sbss2 .rela.sbss2.* .rela.gnu.linkonce.sb2.*)
46      *(.rela.bss .rela.bss.* .rela.gnu.linkonce.b.*)
47      PROVIDE_HIDDEN (__rela_iplt_start = .);
48      *(.rela.iplt)
49      PROVIDE_HIDDEN (__rela_iplt_end = .);
50    }
51  .rela.plt        :
52    {
53      *(.rela.plt)
54    }
55  .init            :
56  {
57    KEEP (*(SORT_NONE(.init)))
58  }
59  .plt             : { *(.plt) }
60  .iplt            : { *(.iplt) }
61  .text            :
62  {
63    *(.text.unlikely .text.*_unlikely .text.unlikely.*)
64    *(.text.exit .text.exit.*)
65    *(.text.startup .text.startup.*)
66    *(.text.hot .text.hot.*)
67    *(.text .stub .text.* .gnu.linkonce.t.*)
68    /* .gnu.warning sections are handled specially by elf32.em.  */
69    *(.gnu.warning)
```

```
70  }
71  .fini           :
72  {
73    KEEP (*(SORT_NONE(.fini)))
74  }
75  PROVIDE (__etext = .);
76  PROVIDE (_etext = .);
77  PROVIDE (etext = .);
78  .rodata         : { *(.rodata .rodata.* .gnu.linkonce.r.*) }
79  .rodata1        : { *(.rodata1) }
80  .sdata2         :
81  {
82    *(.sdata2 .sdata2.* .gnu.linkonce.s2.*)
83  }
84  .sbss2          : { *(.sbss2 .sbss2.* .gnu.linkonce.sb2.*) }
85  .eh_frame_hdr   : { *(.eh_frame_hdr) *(.eh_frame_entry
     .eh_frame_entry.*) }
86  .eh_frame       : ONLY_IF_RO { KEEP (*(.eh_frame)) *(.eh_frame.*) }
87  .gcc_except_table   : ONLY_IF_RO { *(.gcc_except_table
     .gcc_except_table.*) }
88  .gnu_extab   : ONLY_IF_RO { *(.gnu_extab*) }
89  /* These sections are generated by the Sun/Oracle C++ compiler.  */
90  .exception_ranges   : ONLY_IF_RO { *(.exception_ranges*) }
91  /* Adjust the address for the data segment.  We want to adjust up to
92     the same address within the page on the next page up.  */
93  . = DATA_SEGMENT_ALIGN (CONSTANT (MAXPAGESIZE), CONSTANT
     (COMMONPAGESIZE));
94  /* Exception handling  */
95  .eh_frame       : ONLY_IF_RW { KEEP (*(.eh_frame)) *(.eh_frame.*) }
96  .gnu_extab      : ONLY_IF_RW { *(.gnu_extab) }
97  .gcc_except_table   : ONLY_IF_RW { *(.gcc_except_table
     .gcc_except_table.*) }
98  .exception_ranges   : ONLY_IF_RW { *(.exception_ranges*) }
99  /* Thread Local Storage sections  */
100 .tdata    :
101  {
102    PROVIDE_HIDDEN (__tdata_start = .);
103    *(.tdata .tdata.* .gnu.linkonce.td.*)
104  }
105 .tbss     : { *(.tbss .tbss.* .gnu.linkonce.tb.*) *(.tcommon) }
106 .preinit_array     :
107 {
```

```
108     PROVIDE_HIDDEN (__preinit_array_start = .);
109     KEEP (*(.preinit_array))
110     PROVIDE_HIDDEN (__preinit_array_end = .);
111   }
112   .init_array    :
113   {
114     PROVIDE_HIDDEN (__init_array_start = .);
115     KEEP (*(SORT_BY_INIT_PRIORITY(.init_array.*)
      SORT_BY_INIT_PRIORITY(.ctors.*)))
116     KEEP (*(.init_array EXCLUDE_FILE (*crtbegin.o *crtbegin?.o *crtend.o
      *crtend?.o ) .ctors))
117     PROVIDE_HIDDEN (__init_array_end = .);
118   }
119   .fini_array    :
120   {
121     PROVIDE_HIDDEN (__fini_array_start = .);
122     KEEP (*(SORT_BY_INIT_PRIORITY(.fini_array.*)
      SORT_BY_INIT_PRIORITY(.dtors.*)))
123     KEEP (*(.fini_array EXCLUDE_FILE (*crtbegin.o *crtbegin?.o *crtend.o
      *crtend?.o ) .dtors))
124     PROVIDE_HIDDEN (__fini_array_end = .);
125   }
126   .ctors          :
127   {
128     /* gcc uses crtbegin.o to find the start of
129        the constructors, so we make sure it is
130        first.  Because this is a wildcard, it
131        doesn't matter if the user does not
132        actually link against crtbegin.o; the
133        linker won't look for a file to match a
134        wildcard.  The wildcard also means that it
135        doesn't matter which directory crtbegin.o
136        is in.  */
137     KEEP (*crtbegin.o(.ctors))
138     KEEP (*crtbegin?.o(.ctors))
139     /* We don't want to include the .ctor section from
140        the crtend.o file until after the sorted ctors.
141        The .ctor section from the crtend file contains the
142        end of ctors marker and it must be last */
143     KEEP (*(EXCLUDE_FILE (*crtend.o *crtend?.o ) .ctors))
144     KEEP (*(SORT(.ctors.*)))
145     KEEP (*(.ctors))
```

```
146  }
147  .dtors          :
148  {
149    KEEP (*crtbegin.o(.dtors))
150    KEEP (*crtbegin?.o(.dtors))
151    KEEP (*(EXCLUDE_FILE (*crtend.o *crtend?.o ) .dtors))
152    KEEP (*(SORT(.dtors.*)))
153    KEEP (*(.dtors))
154  }
155  .jcr            : { KEEP (*(.jcr)) }
156  .data.rel.ro : { *(.data.rel.ro.local* .gnu.linkonce.d.rel.ro.local.*)
      *(.data.rel.ro .data.rel.ro.* .gnu.linkonce.d.rel.ro.*) }
157  .dynamic        : { *(.dynamic) }
158  . = DATA_SEGMENT_RELRO_END (0, .);
159  .data           :
160  {
161    __DATA_BEGIN__ = .;
162    *(.data .data.* .gnu.linkonce.d.*)
163    SORT(CONSTRUCTORS)
164  }
165  .data1          : { *(.data1) }
166  .got            : { *(.got.plt) *(.igot.plt) *(.got) *(.igot) }
167  /* We want the small data sections together, so single-instruction
      offsets
168     can access them all, and initialized data all before uninitialized, so
169     we can shorten the on-disk segment size.  */
170  .sdata          :
171  {
172    __SDATA_BEGIN__ = .;
173    *(.srodata.cst16) *(.srodata.cst8) *(.srodata.cst4) *(.srodata.cst2)
     *(.srodata .srodata.*)
174    *(.sdata .sdata.* .gnu.linkonce.s.*)
175  }
176  _edata = .; PROVIDE (edata = .);
177  . = .;
178  __bss_start = .;
179  .sbss           :
180  {
181    *(.dynsbss)
182    *(.sbss .sbss.* .gnu.linkonce.sb.*)
183    *(.scommon)
184  }
```

```
185  .bss            :
186  {
187   *(.dynbss)
188   *(.bss .bss.* .gnu.linkonce.b.*)
189   *(COMMON)
190   /* Align here to ensure that the .bss section occupies space up to
191      _end.  Align after .bss to ensure correct alignment even if the
192      .bss section disappears because there are no input sections.
193      FIXME: Why do we need it? When there is no .bss section, we do not
194      pad the .data section.  */
195   . = ALIGN(. != 0 ? 32 / 8 : 1);
196  }
197  . = ALIGN(32 / 8);
198  . = SEGMENT_START("ldata-segment", .);
199  . = ALIGN(32 / 8);
200 __BSS_END__ = .;
201    __global_pointer$ = MIN(__SDATA_BEGIN__ + 0x800,
202             MAX(__DATA_BEGIN__ + 0x800, __BSS_END__ - 0x800));
203 _end = .; PROVIDE (end = .);
204  . = DATA_SEGMENT_END (.);
205  /* Stabs debugging sections.  */
206  .stab          0 : { *(.stab) }
207  .stabstr       0 : { *(.stabstr) }
208  .stab.excl     0 : { *(.stab.excl) }
209  .stab.exclstr  0 : { *(.stab.exclstr) }
210  .stab.index    0 : { *(.stab.index) }
211  .stab.indexstr 0 : { *(.stab.indexstr) }
212  .comment       0 : { *(.comment) }
213  .gnu.build.attributes : { *(.gnu.build.attributes
      .gnu.build.attributes.*) }
214  /* DWARF debug sections.
215     Symbols in the DWARF debugging sections are relative to the beginning
216     of the section so we begin them at 0.  */
217  /* DWARF 1 */
218  .debug          0 : { *(.debug) }
219  .line           0 : { *(.line) }
220  /* GNU DWARF 1 extensions */
221  .debug_srcinfo  0 : { *(.debug_srcinfo) }
222  .debug_sfnames  0 : { *(.debug_sfnames) }
223  /* DWARF 1.1 and DWARF 2 */
224  .debug_aranges  0 : { *(.debug_aranges) }
225  .debug_pubnames 0 : { *(.debug_pubnames) }
```

```
226   /* DWARF 2 */
227   .debug_info     0 : { *(.debug_info .gnu.linkonce.wi.*) }
228   .debug_abbrev   0 : { *(.debug_abbrev) }
229   .debug_line     0 : { *(.debug_line .debug_line.* .debug_line_end) }
230   .debug_frame    0 : { *(.debug_frame) }
231   .debug_str      0 : { *(.debug_str) }
232   .debug_loc      0 : { *(.debug_loc) }
233   .debug_macinfo  0 : { *(.debug_macinfo) }
234   /* SGI/MIPS DWARF 2 extensions */
235   .debug_weaknames 0 : { *(.debug_weaknames) }
236   .debug_funcnames 0 : { *(.debug_funcnames) }
237   .debug_typenames 0 : { *(.debug_typenames) }
238   .debug_varnames  0 : { *(.debug_varnames) }
239   /* DWARF 3 */
240   .debug_pubtypes 0 : { *(.debug_pubtypes) }
241   .debug_ranges   0 : { *(.debug_ranges) }
242   /* DWARF Extension.  */
243   .debug_macro    0 : { *(.debug_macro) }
244   .debug_addr     0 : { *(.debug_addr) }
245   .gnu.attributes 0 : { KEEP (*(.gnu.attributes)) }
246   /DISCARD/ : { *(.note.GNU-stack) *(.gnu_debuglink) *(.gnu.lto_*) }
247 }
248
249
250 ================================================================
```

Auflistung A.1: Standard Linker Skript der riscv-none-embed-gcc Toolchain link.ld

A.2 Batchskripte

A.2.1 create.tcl

```
# This is the first vivado tcl script. It is creating a project for
    further usage with all the source files for the bram project and the
    board nexys video.
set outputDir build
set log build/logs/project.log
file mkdir $outputDir
# Project create with FPGA from Nyxes Video
create_project masterthesis build/masterthesis -part xc7a100tcsg324-1
    -force

# set board properties
# set_property board_part digilentinc.com:nexys4_ddr:part0:1.1
    [current_project]

# Set Verliog as preferred language
set_property target_language Verilog [current_project]

# Adding the required files
add_files -norecurse -scan_for_includes [ glob src/rtl/* ]
update_compile_order -fileset sources_1

# creating clock
create_ip -name clk_wiz -vendor xilinx.com -library ip -version 6.0
    -module_name clk_wiz_0
set_property -dict [list CONFIG.USE_PHASE_ALIGNMENT {false}
    CONFIG.CLKOUT1_REQUESTED_OUT_FREQ {25.000} CONFIG.USE_LOCKED {false}
    CONFIG.SECONDARY_SOURCE {Single_ended_clock_capable_pin}
    CONFIG.CLKOUT1_DRIVES {BUFG} CONFIG.CLKOUT2_DRIVES {BUFG}
    CONFIG.CLKOUT3_DRIVES {BUFG} CONFIG.CLKOUT4_DRIVES {BUFG}
    CONFIG.CLKOUT5_DRIVES {BUFG} CONFIG.CLKOUT6_DRIVES {BUFG}
    CONFIG.CLKOUT7_DRIVES {BUFG} CONFIG.MMCM_CLKFBOUT_MULT_F {9.125}
    CONFIG.MMCM_CLKOUT0_DIVIDE_F {36.500} CONFIG.CLKOUT1_JITTER {181.828}
    CONFIG.CLKOUT1_PHASE_ERROR {104.359}] [get_ips clk_wiz_0]
generate_target {instantiation_template} [get_files
    build/masterthesis/masterthesis.srcs/sources_1/ip/clk_wiz_0/clk_wiz_0
    .xci]
update_compile_order -fileset sources_1
```

```
generate_target all [get_files
    build/masterthesis/masterthesis.srcs/sources_1/ip/clk_wiz_0/clk_wiz_0
    .xci]
catch { config_ip_cache -export [get_ips -all clk_wiz_0] }
export_ip_user_files -of_objects [get_files
    build/masterthesis/masterthesis.srcs/sources_1/ip/clk_wiz_0/clk_wiz_0
    .xci] -no_script -sync -force -quiet
create_ip_run [get_files -of_objects [get_fileset sources_1]
    build/masterthesis/masterthesis.srcs/sources_1/ip/clk_wiz_0/clk_wiz_0
    .xci]
launch_runs clk_wiz_0_synth_1 -jobs 8
export_simulation -of_objects [get_files
    build/masterthesis/masterthesis.srcs/sources_1/ip/clk_wiz_0/clk_wiz_0
    .xci] -directory
    build/masterthesis/masterthesis.ip_user_files/sim_scripts
    -ip_user_files_dir build/masterthesis/masterthesis.ip_user_files
    -ipstatic_source_dir
    build/masterthesis/masterthesis.ip_user_files/ipstatic -lib_map_path
    [list
    {modelsim=build/masterthesis/masterthesis.cache/compile_simlib/modelsim}
    {questa=build/masterthesis/masterthesis.cache/compile_simlib/questa}
    {ies=build/masterthesis/masterthesis.cache/compile_simlib/ies}
    {xcelium=build/masterthesis/masterthesis.cache/compile_simlib/xcelium}
    {vcs=build/masterthesis/masterthesis.cache/compile_simlib/vcs}
    {riviera=build/masterthesis/masterthesis.cache/compile_simlib/riviera}]
    -use_ip_compiled_libs -force -quiet

#creating bram
create_ip -name blk_mem_gen -vendor xilinx.com -library ip -version 8.4
    -module_name blk_mem_gen_0
set_property -dict [list CONFIG.Memory_Type {True_Dual_Port_RAM}
    CONFIG.Enable_32bit_Address {true} CONFIG.Use_Byte_Write_Enable {true}
    CONFIG.Byte_Size {8} CONFIG.Assume_Synchronous_Clk {true}
    CONFIG.Write_Width_A {32} CONFIG.Write_Depth_A {65536}
    CONFIG.Read_Width_A {32} CONFIG.Enable_A {Always_Enabled}
    CONFIG.Write_Width_B {32} CONFIG.Read_Width_B {32} CONFIG.Enable_B
    {Always_Enabled} CONFIG.Register_PortA_Output_of_Memory_Primitives
    {false} CONFIG.Register_PortB_Output_of_Memory_Primitives {false}
    CONFIG.Load_Init_File {true} CONFIG.Coe_File
    {../../../../../../ext/riscv-caeco-master.coe}
    CONFIG.Fill_Remaining_Memory_Locations {true} CONFIG.Use_RSTA_Pin
    {false} CONFIG.Use_RSTB_Pin {false} CONFIG.Port_B_Clock {100}
```

```
    CONFIG.Port_B_Write_Rate {50} CONFIG.Port_B_Enable_Rate {100}
    CONFIG.EN_SAFETY_CKT {false}] [get_ips blk_mem_gen_0]
generate_target {instantiation_template} [get_files
    build/masterthesis/masterthesis.srcs/sources_1/ip/blk_mem_gen_0/
    blk_mem_gen_0.xci]
update_compile_order -fileset sources_1
generate_target all [get_files
    build/masterthesis/masterthesis.srcs/sources_1/ip/blk_mem_gen_0/
    blk_mem_gen_0.xci]
catch { config_ip_cache -export [get_ips -all blk_mem_gen_0] }
export_ip_user_files -of_objects [get_files
    build/masterthesis/masterthesis.srcs/sources_1/ip/blk_mem_gen_0/
    blk_mem_gen_0.xci] -no_script -sync -force -quiet
create_ip_run [get_files -of_objects [get_fileset sources_1]
    build/masterthesis/masterthesis.srcs/sources_1/ip/blk_mem_gen_0/
    blk_mem_gen_0.xci]
launch_runs blk_mem_gen_0_synth_1 -jobs 8
export_simulation -of_objects [get_files
    build/masterthesis/masterthesis.srcs/sources_1/ip/blk_mem_gen_0/
    blk_mem_gen_0.xci] -directory
    build/masterthesis/masterthesis.ip_user_files/sim_scripts
    -ip_user_files_dir build/masterthesis/masterthesis.ip_user_files
    -ipstatic_source_dir
    build/masterthesis/masterthesis.ip_user_files/ipstatic -lib_map_path
    [list
    {modelsim=build/masterthesis/masterthesis.cache/compile_simlib/modelsim}
    {questa=build/masterthesis/masterthesis.cache/compile_simlib/questa}
    {ies=build/masterthesis/masterthesis.cache/compile_simlib/ies}
    {xcelium=build/masterthesis/masterthesis.cache/compile_simlib/xcelium}
    {vcs=build/masterthesis/masterthesis.cache/compile_simlib/vcs}
    {riviera=build/masterthesis/masterthesis.cache/compile_simlib/riviera}]
    -use_ip_compiled_libs -force -quiet

# constrains:
add_files -fileset constrs_1 -norecurse
    src/constrains/Nexys-4-DDR-Master.xdc

set_property SOURCE_SET sources_1 [get_filesets sim_1]
add_files -fileset sim_1 -norecurse -scan_for_includes [glob
    src/simulation/masterthesis_tb.v]
update_compile_order -fileset sim_1

```

```
close_project
```

Auflistung A.2: create.tcl

A.2.2 synthesis.tcl

```
# This is the second TCL script for opening the project, adding the
    simulation files and performing a simulation
set outputDir build
set log build/logs/project.log
file mkdir $outputDir

open_project build/masterthesis/masterthesis.xpr
# Run Synthesis. Final Step for this Job
launch_runs synth_1 -jobs 8
wait_on_run synth_1
# Writes in log, when Synthesis is successfull.
if {[get_property STATUS [get_runs synth_1]] == {synth_design Complete!}}
    then {set fp [open $log a]; puts $fp "Synth. Successful: 1"; close $fp}
    else {set fp [open $log a]; puts $fp "Synth. Successful: 0"; close $fp}

close_project
```

Auflistung A.3: synthesis.tcl

A.2.3 implementation.tcl

```
# This is the second TCL script for opening the project, adding the
    simulation files and performing a simulation
set outputDir build
set log build/logs/project.log
file mkdir $outputDir
#
open_project build/masterthesis/masterthesis.xpr
#
# launch implementation and writing the bitstream
launch_runs impl_1 -jobs 8
wait_on_run impl_1
if {[get_property STATUS [get_runs impl_1]] == {route_design Complete!}}
    then {set fp [open $log a]; puts $fp "Impl. Successful: 1"; close $fp}
    else {set fp [open $log a]; puts $fp "Impl. Successful: 0"; close $fp}
# Write Bitstream
```

```
#
launch_runs impl_1 -to_step write_bitstream -jobs 8
wait_on_run impl_1
#
close_project
```

Auflistung A.4: implementation.tcl

A.2.4 simulation.tcl

```
# This is the second TCL script for opening the project, adding the
    simulation files and performing a simulation
open_project build/masterthesis/masterthesis.xpr
#Simulation Start
#
launch_simulation
# open_wave_config src/simulation/pomaa.wcfg
source
    build/masterthesis/masterthesis.sim/sim_1/behav/xsim/masterthesis_tb.tcl
log_wave /masterthesis_tb/DUT/*
#
run 1 us

close_project
```

Auflistung A.5: simulation.tcl

A.3 Programm-Codes

A.3.1 new-mem.exe Rust Hilfsprogramm

```
use std::env;
use std::io::Read;
use std::io::Write;
use std::str;

fn main() {
    // getting input argument, which is the path to the selected mem file
    let args: Vec<String> = env::args().collect();
    let path = &args[1].to_string();
    let path_exp = &args[2].to_string();
    // opens the file
    let mut file = std::fs::File::open(path).unwrap();
    let mut contents = String::new();
    file.read_to_string(&mut contents).unwrap();

    let b = contents.as_bytes();
     println!("{:?}", b);
    // get the @ Adress, since header can variate
    let pos = b.iter().position(|&x| x == 64).unwrap();
    println!("{:?}", pos);
    // iterate through the vector
    let filter : Vec<u8> = b
                                        .iter()
                                        .clone()
                                        .skip(pos + 10)
                                        .filter(|x| {
                                            // 0-9
                                            **x == 48 ||
                                            **x == 49 ||
                                            **x == 50 ||
                                            **x == 51 ||
                                            **x == 52 ||
                                            **x == 53 ||
                                            **x == 54 ||
                                            **x == 55 ||
                                            **x == 56 ||
                                            **x == 57 ||
                                            // a-f
```

```
                                                **x == 65 ||
                                                **x == 66 ||
                                                **x == 67 ||
                                                **x == 68 ||
                                                **x == 69 ||
                                                **x == 70} )
                                            .map(|x| *x)
                                            .collect();

    // creates a new file
    let mut file_create = std::fs::File::create(path_exp).expect("create
    failed");
    // write start address @00000000
    file_create.write_all("@00000000".as_bytes()).expect("write failed");
    // getting all instructions
    // for i in  0..instructions_num{
    for _i in  (0..filter.len()).step_by(8){
        let mut empty: Vec<u8> = vec![0;8];
        // creating the vector new for every instruction
        for j in 0..8{
            empty[7-j] = filter[j+_i];
        }
        // swaping bytes
        let mut instruction:Vec<u8> = vec![0;8];
        instruction[0] = empty[1];
        instruction[1] = empty[0];
        instruction[2] = empty[3];
        instruction[3] = empty[2];
        instruction[4] = empty[5];
        instruction[5] = empty[4];
        instruction[6] = empty[7];
        instruction[7] = empty[6];

        file_create.write_all("\n".as_bytes()).expect("write failed");
        file_create.write_all(&instruction).expect("write failed");
    }
}
```

Auflistung A.6: Programm-Code für new-mem.exe

A.3.2 new-coe.exe Rust Hilfsprogramm

```
use std::env;
use std::io::Read;
use std::io::Write;
use std::str;

fn main() {
    // getting input argument, which is the path to the selected mem file
    let args: Vec<String> = env::args().collect();
    // input file
    let path = &args[1].to_string();
    // output file
    let path_exp = &args[2].to_string();
    // opens the file
    let mut file = std::fs::File::open(path).unwrap();
    let mut contents = String::new();
    file.read_to_string(&mut contents).unwrap();

    let b = contents.as_bytes();

    // write the header for new file
    let mut file_create = std::fs::File::create(path_exp).expect("create failed");
    file_create.write_all("memory_initialization_radix=16;\n".as_bytes()).expect("write failed");
    file_create.write_all("memory_initialization_vector=\n".as_bytes()).expect("write failed");


    // iterate through the vector
    let mut filter : Vec<u8> = b
                                        .into_iter()
                                        .clone()
                                        .skip(10)
                                        .map(|mut x| {
                                            if *x == 65 {x = &97}
                                            else if *x == 66 {x = &98}
                                            else if *x == 67 {x = &99}
                                            else if *x == 68 {x = &100}
                                            else if *x == 69 {x = &101}
                                            else if *x == 70 {x = &102}
                                            else if *x == 10 {x = &44}
```

```
39                                                  else {x = x};
40                                                  *x
41                                              })
42                                              .collect();
43      // push a last sign for iterating
44      filter.push(59);
45      println!("Number of instructions {:?}", (filter.len() / 9));
46
47      // getting all instructions
48      for i in  (0..filter.len()).step_by(9){
49          // create the new instruction with colon
50          let mut instruction:Vec<u8> = vec![0;9];
51          instruction[0] = filter[i+0];
52          instruction[1] = filter[i+1];
53          instruction[2] = filter[i+2];
54          instruction[3] = filter[i+3];
55          instruction[4] = filter[i+4];
56          instruction[5] = filter[i+5];
57          instruction[6] = filter[i+6];
58          instruction[7] = filter[i+7];
59          instruction[8] = filter[i+8];
60
61          file_create.write_all(&instruction).expect("write failed");
62          file_create.write_all("\n".as_bytes()).expect("write failed");
63      }
64  }
```

Auflistung A.7: Programm-Code für new-coe.exe

A.3.3 Kompilierter C Programm-Code Initialisierung

```
1
2C:\Users\FabianBruenger\Documents\Masterthesis\Zieldateien\Test3.elf:
    file format elf32-littleriscv
3
4
5Disassembly of section .text:
6
780000000 <_start>:
880000000: 00000197          auipc gp,0x0
980000004: 0ec18193          addi gp,gp,236 # 800000ec <__global_pointer$>
1080000008: 00006117         auipc sp,0x6
118000000c: 1f410113         addi sp,sp,500 # 800061fc <__stack>
1280000010: 300fd073         csrwi mstatus,31
1380000014: 00000e97         auipc t4,0x0
1480000018: 0b0e8e93         addi t4,t4,176 # 800000c4 <trap_handler>
158000001c: 305e9073         csrw mtvec,t4
1680000020: 00000093         li ra,0
1780000024: 00008113         mv sp,ra
1880000028: 00008193         mv gp,ra
198000002c: 00008213         mv tp,ra
2080000030: 00008293         mv t0,ra
2180000034: 00008313         mv t1,ra
2280000038: 00008393         mv t2,ra
238000003c: 00008413         mv s0,ra
2480000040: 00008493         mv s1,ra
2580000044: 00008513         mv a0,ra
2680000048: 00008593         mv a1,ra
278000004c: 00008613         mv a2,ra
2880000050: 00008693         mv a3,ra
2980000054: 00008713         mv a4,ra
3080000058: 00008793         mv a5,ra
318000005c: 00008813         mv a6,ra
3280000060: 00008893         mv a7,ra
3380000064: 00008913         mv s2,ra
3480000068: 00008993         mv s3,ra
358000006c: 00008a13         mv s4,ra
3680000070: 00008a93         mv s5,ra
3780000074: 00008b13         mv s6,ra
3880000078: 00008b93         mv s7,ra
398000007c: 00008c13         mv s8,ra
4080000080: 00008c93         mv s9,ra
```

```
80000084: 00008d13          mv s10,ra
80000088: 00008d93          mv s11,ra
8000008c: 00008e13          mv t3,ra
80000090: 00008e93          mv t4,ra
80000094: 00008f13          mv t5,ra
80000098: 00008f93          mv t6,ra
8000009c: 00002d17          auipc s10,0x2
800000a0: 144d0d13          addi s10,s10,324 # 800021e0 <_bss_start>
800000a4: 00002d97          auipc s11,0x2
800000a8: 154d8d93          addi s11,s11,340 # 800021f8 <errno>
800000ac: 01bd5863          bge s10,s11,800000bc <zero_loop_end>

800000b0 <zero_loop>:
800000b0: 000d2023          sw zero,0(s10)
800000b4: 004d0d13          addi s10,s10,4
800000b8: ffaddce3          bge s11,s10,800000b0 <zero_loop>

800000bc <zero_loop_end>:
800000bc: 2f0000ef          jal ra,800003ac <main>
800000c0: 0000006f          j 800000c0 <zero_loop_end+0x4>

800000c4 <trap_handler>:
800000c4: 2640006f          j 80000328 <exception_handler>
800000c8: 27c0006f          j 80000344 <l_1_caeco_interrupt_handler>
800000cc: 30200073          mret

...

80000344 <l_1_caeco_interrupt_handler>:
80000344: fe010113          addi sp,sp,-32
80000348: 00812e23          sw s0,28(sp)
8000034c: 00e12c23          sw a4,24(sp)
80000350: 00f12a23          sw a5,20(sp)
80000354: 02010413          addi s0,sp,32
80000358: c00007b7          lui a5,0xc0000
8000035c: 0c078793          addi a5,a5,192 # c00000c0 <_gpio_creg+0xb4>
....
```

Auflistung A.8: Kompilierter C Programm-Code Initialisierung

```
/*
 * airi5c_syscalls.h
 *
 *  Created on: 11.11.2019
 *      Author: stanitzk
 */

#ifndef AIRI5C_SYSCALLS_H_
#define AIRI5C_SYSCALLS_H_

#include <sys/stat.h>
#include <sys/types.h>
#include <sys/fcntl.h>
#include <sys/times.h>
#include <sys/errno.h>
#include <sys/time.h>
#include <stdio.h>


void _exit();
int close(int file);
int execve(char *name, char **argv, char **env);
int fork();
int fstat(int file, struct stat *st);
int getpid();
int isatty(int file);
int kill(int pid, int sig);
int link(char *old, char *new);
int lseek(int file, int ptr, int dir);
int open(const char *name, int flags, ...);
int read(int file, char *ptr, int len);
caddr_t sbrk(int incr);
int stat(const char *file, struct stat *st);
clock_t times(struct tms *buf);
int unlink(char *name);
int wait(int *status);
int write(int file, char *ptr, int len);

#endif /* AIRI5C_SYSCALLS_H_ */
```

Auflistung A.9: syscalls.h

```c
/*
 * airi5c_syscalls.c
 *
 *  Created on: 11.11.2019
 *      Author: stanitzk
 *
 *  This is a non-reentrant implementation of the 13 syscalls required by
 *  newlib.
 */

#include <sys/stat.h>
#include <errno.h>
#include "airi5c_syscalls.h"

#undef errno
extern int errno; // linker script defines address for this


char *__env[1] = { 0 };
char **environ = __env;

void _exit(int i) {
 while(1);  // park loop
}

int _close(int file) {
 return(-1);  // the only file is stdout, which cannot be closed
}

int _execve(char *name, char **argv, char **env) {
 errno = ENOMEM;
 return -1;
}


int _fork() {
 errno = EAGAIN;
 return(-1);
}

int _fstat(int file, struct stat *st) {
 st->st_mode = S_IFCHR;
```

```
43  return 0;
44 }
45
46 int _getpid(void) {
47  return 1;
48 }
49
50 int _isatty(int file) {
51  return 1;
52 }
53
54 int _kill(int pid, int sig) {
55  errno = EINVAL;
56  return -1;
57 }
58
59 int _link(char *old, char *new) {
60  errno = EMLINK;
61  return -1;
62 }
63
64 int _lseek(int file, int ptr, int dir) {
65  return 0;
66 }
67
68 int _open(const char *name, int flags, ...) {
69  return -1;
70 }
71
72 int _read(int file, char *ptr, int len) {
73  return 0;
74 }
75
76 caddr_t _sbrk(int incr) {
77  extern int __end;
78  static void *heap_end;
79  void *prev_heap_end;
80
81  register void* stack_ptr asm("sp");
82  if(heap_end == NULL)
83   heap_end = (void*)&__end;
84  prev_heap_end = heap_end;
```

```
85  if((void*)(heap_end + incr) > stack_ptr) {
86   write(1, "Heap and stack collision\n",25);
87   while(1);
88  }
89  heap_end += incr;
90  return (caddr_t) prev_heap_end;
91 }
92
93 int _stat(const char *file, struct stat *st) {
94  st->st_mode = S_IFCHR;
95  return 0;
96 }
97
98 clock_t _times(struct tms *buf) {
99  errno = EACCES;
100  return -1;
101 }
102 int _unlink(char *name) {
103  errno = ENOENT;
104  return -1;
105 }
106
107 int _wait(int *status) {
108  errno = ECHILD;
109  return -1;
110 }
111
112 void outbyte(char payload)
113 {
114  int i;
115  volatile extern int _uart_dreg;
116  _uart_dreg = 0x100 | payload;
117  _uart_dreg = 0x000;
118  for(i = 0; i < 10000; i++);
119 }
120
121 int _write(int file, char *ptr, int len) {
122  int i;
123  for(i = 0; i < len; i++)
124  {
125   outbyte(ptr[i]);
126  }
```

```
127 return len;
128 }
```

Auflistung A.10: syscalls.c

B HDL Code

B.1 Verilog

B.1.1 Caecointerface

```
1 `timescale 1ns / 1ps
2 `include "POMAA_constants.vh"
3 //------------------------------------  States
    ------------------------------------------------------//
4
5 module caecointerface(
6     input clk,
7     input rst,
8     //---------------------------------- Signals from processor
    ----------------------------------------//
9     // enable signal for perepherie
10    input           en,
11    // write signal
12    input           wen,
13    // adresse
14    input   [31:0]  addr,
15    // data
16    input   [31:0]  wdata,
17    //---------------------------------- Signals from dmi module direct
    --------------------------------//
18    // data from debug module direct. Should be 0x00000001, if cmd is 1
19    input   [31:0]  dm_wdata,
20    // write enable from dm
21    input           dm_wen,
22    // command set from dm
23    input           dm_cmd,
24    //---------------------------------- SOut always through the processor
    ------------------------//
25    // final data out
26    output  [31:0]  rdata,
27    // interrupt
```

```
28      output          res_inter,
29      // led for debugging
30      output          led
31  );
32  //------------------------------------  Signale
        ------------------------------------------------------//
33  wire    [15:0]  din;
34  wire    [31:0]  result;
35  wire            invalid, inready, resultvalid, cmd;
36  reg     [15:0]  datahalf_r;
37  reg     [3:0]   state, next_state;
38  reg             invalid_r, ctrl_r, led_r;
39
40  //------------------------------------  Instantiierung Caemo
        ---------------------------------------//
41  caeco caeco_inst(
42      // data: in 16-Bit
43      .DIN(din),
44      // control signal to valid dai: in 1-Bit
45      .DIN_VALID(invalid),
46      // if true, data can be written: out 1-Bit
47      .DIN_READY(inready),
48      // has to be set, if last data is written: in 1-Bit
49      .DIN_LAST(1'b0),
50      //-----
51      // result register: out
52      .RESULT(result),
53      // valid result : out 1-Bit
54      .RESULT_VALID(resultvalid),
55      // cmd: in 1-Bit
56      .CMD(cmd),
57      // en: in 1-Bit
58      .EN(1'b0),
59      //-----
60      .RSTN(~rst),
61      .CLK(clk)
62  );
63  //------------------------------------  FSM signal processing
        Inputs-------------------------------------------//
64  //-------------------------------------------------- LED
65  always@(posedge clk or posedge rst)
66  begin
```

```
67      if(rst) begin
68      led_r <= 1'b0;
69      end
70      else begin
71          if(cmd) begin
72          led_r <= 1'b1;
73          end
74          else if (resultvalid) begin
75          led_r <= 1'b0;
76          end
77      end
78  end
79  //---------------------------------------------------- 1
80  always @(posedge clk or posedge rst)
81  begin
82      if(rst) begin
83          state<=`IDLE;
84      end
85      else begin
86          state <= next_state;
87      end
88  end
89  //---------------------------------------------------- 2
90  always@(*)
91  begin
92          case(state)
93              `IDLE: begin
94              //------------------------------------ States for the direct
    signals from cpu
95              // If adress is c0..10 -> write process start
96                  if (en && wen && addr==32'hc0000010) begin
97                      next_state = `WDA0;
98                  end
99              // If address is c0 .. 11 -> control signal is set for one
    clock
100                 else if (en && wen && addr==32'hc0000011) begin
101                     next_state = `CTRL;
102                 end
103             //------------------------------------ States for the direct
    signals from dm
104                 else if (dm_cmd) begin
105                     next_state = `CTRL_DM;
```

```
106                 end
107                 else if (dm_wen) begin
108                     next_state = `WDA0_DM;
109                 end
110                 else begin
111                     next_state = `IDLE;
112                 end
113             end
114             `WDA0: begin
115                 next_state = `WDA1;
116             end
117             `WDA1: begin
118                 next_state = `IDLE;
119             end
120             `WDA0_DM: begin
121                 next_state = `WDA1_DM;
122             end
123             `WDA1_DM: begin
124                 next_state = `WAIT_DM;
125             end
126             `CTRL:      begin
127                 next_state = `IDLE;
128             end
129             `CTRL_DM:   begin
130                 next_state = `WAIT_DM;
131             end
132             `WAIT_DM:   begin
133                 next_state = (dm_cmd || dm_wen) ? `WAIT_DM :
134                 (en && wen && addr==32'hc0000010) ?  `WDA0 :
135                 `IDLE;
136             end
137 
138             default: next_state = `IDLE;
139         endcase
140 end
141 //------------------------------------------------------ 3
142 always@(*)
143 begin
144     invalid_r   = 1'b0;
145     ctrl_r      = 1'b0;
146     datahalf_r  = 16'h0;
147     case(state)
```

```
148         `IDLE: begin
149             invalid_r   = 1'b0;
150             ctrl_r      = 1'b0;
151             datahalf_r  = 16'h0;
152         end
153         `WDA0: begin
154             invalid_r   = 1'b1;
155             ctrl_r      = 1'b0;
156             datahalf_r  = { wdata[23:16] ,wdata[31:24] };
157         end
158         `WDA1: begin
159             invalid_r   = 1'b1;
160             ctrl_r      = 1'b0;
161             datahalf_r  = { wdata[7:0] ,wdata[15:8] } ;
162         end
163         `WDA0_DM: begin
164             invalid_r   = 1'b1;
165             ctrl_r      = 1'b0;
166             datahalf_r  = { dm_wdata[23:16] ,dm_wdata[31:24] };
167         end
168         `WDA1_DM: begin
169             invalid_r   = 1'b1;
170             ctrl_r      = 1'b0;
171             datahalf_r  = { dm_wdata[7:0] ,dm_wdata[15:8] } ;
172         end
173         `CTRL: begin
174             invalid_r   = 1'b0;
175             ctrl_r      = wdata[0];
176             datahalf_r  = 16'h0;
177         end
178         `CTRL_DM: begin
179             invalid_r   = 1'b0;
180             ctrl_r      = dm_wdata[0];
181             datahalf_r  = 16'h0;
182         end
183         `WAIT_DM: begin
184             invalid_r   = 1'b0;
185             ctrl_r      = 1'b0;
186             datahalf_r  = 16'h0;
187         end
188     endcase
189 end
```

```
190 //---------- Assign
191 assign cmd          = ctrl_r;
192 assign din          = datahalf_r;
193 assign invalid      = invalid_r;
194 assign rdata        = result;
195 assign res_inter    = resultvalid;
196 assign led          = led_r;
197
198 endmodule
```

Auflistung B.1: Caecointerfave.v

B.1.2 POMAA_constants.vh

```
1 // POMAA constant definitions
2
3 // 1: states for FSM which handles the data from core
4 `define FSM_STATE_WIDTH 5
5
6 `define IDLE     `FSM_STATE_WIDTH'h0
7 `define WDA0     `FSM_STATE_WIDTH'h1
8 `define WDA1     `FSM_STATE_WIDTH'h2
9 `define CTRL     `FSM_STATE_WIDTH'h3
10 `define READ    `FSM_STATE_WIDTH'h4
11
12 // 2. States for data from dm directly
13 `define CTRL_DM `FSM_STATE_WIDTH'h5
14 `define WAIT_DM `FSM_STATE_WIDTH'h6
15 `define WDA0_DM `FSM_STATE_WIDTH'h7
16 `define WDA1_DM `FSM_STATE_WIDTH'h8
```

Auflistung B.2: POMAA_constants.vh

B.1.3 JTAG Task jtag_dmi_write

```
task jtag_dmi_write;
input [5:0] addr;
input [31:0] data;
input [1:0] command;

output reg[31:0] result;

begin
 // DEBUG if(command == 2'h2) $display("dmi: write to %h : %h",addr, data);
 // goto Shift-IR state
 tdi <= 1'b0;
 tms <= 1'b0; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b0;
 tms <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b0;
 tms <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b0;
 tms <= 1'b0; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b0;
 tms <= 1'b0; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b0;
 // Shift in address of DMI register (LSB to MSB)
 tms <= 1'b0;
 tdi <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b0;
 tdi <= 1'b0; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b0;
 tdi <= 1'b0; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b0;
 tdi <= 1'b0; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b0;
 tms <= 1'b1;
 tdi <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b0;
 // goto Update-IR state
 tdi <= 1'b0;
 tms <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b0;
 // goto Shift-DR state
 tdi <= 1'b0;
 tms <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b0;
 tms <= 1'b0; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b0;
 tms <= 1'b0; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b0;
 // shift in DMI address (0x10), data (0x80000000) and write command (0x2)
 tms <= 1'b0;
 tdi <= command[0]; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <=
   1'b0;
 tdi <= command[1]; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <=
   1'b0;

 tdi <= data[0]; #(5*`CLK_PERIOD) result[0] <= tdo; tck <= 1'b1;
   #(5*`CLK_PERIOD) tck <= 1'b0;
```

```
39 tdi <= data[1]; #(5*`CLK_PERIOD) result[1] <= tdo; tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
40 tdi <= data[2]; #(5*`CLK_PERIOD) result[2] <= tdo; tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
41 tdi <= data[3]; #(5*`CLK_PERIOD) result[3] <= tdo; tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
42
43 tdi <= data[4]; #(5*`CLK_PERIOD) result[4] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
44 tdi <= data[5]; #(5*`CLK_PERIOD) result[5] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
45 tdi <= data[6]; #(5*`CLK_PERIOD) result[6] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
46 tdi <= data[7]; #(5*`CLK_PERIOD) result[7] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
47
48 tdi <= data[8]; #(5*`CLK_PERIOD) result[8] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
49 tdi <= data[9]; #(5*`CLK_PERIOD) result[9] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
50 tdi <= data[10]; #(5*`CLK_PERIOD) result[10] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
51 tdi <= data[11]; #(5*`CLK_PERIOD) result[11] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
52
53 tdi <= data[12]; #(5*`CLK_PERIOD) result[12] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
54 tdi <= data[13]; #(5*`CLK_PERIOD) result[13] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
55 tdi <= data[14]; #(5*`CLK_PERIOD) result[14] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
56 tdi <= data[15]; #(5*`CLK_PERIOD) result[15] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
57
58 tdi <= data[16]; #(5*`CLK_PERIOD) result[16] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
59 tdi <= data[17]; #(5*`CLK_PERIOD) result[17] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
60 tdi <= data[18]; #(5*`CLK_PERIOD) result[18] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
61 tdi <= data[19]; #(5*`CLK_PERIOD) result[19] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
```

```
62
63 tdi <= data[20]; #(5*`CLK_PERIOD) result[20] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
64 tdi <= data[21]; #(5*`CLK_PERIOD) result[21] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
65 tdi <= data[22]; #(5*`CLK_PERIOD) result[22] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
66 tdi <= data[23]; #(5*`CLK_PERIOD) result[23] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
67
68 tdi <= data[24]; #(5*`CLK_PERIOD) result[24] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
69 tdi <= data[25]; #(5*`CLK_PERIOD) result[25] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
70 tdi <= data[26]; #(5*`CLK_PERIOD) result[26] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
71 tdi <= data[27]; #(5*`CLK_PERIOD) result[27] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
72
73 tdi <= data[28]; #(5*`CLK_PERIOD) result[28] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
74 tdi <= data[29]; #(5*`CLK_PERIOD) result[29] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
75 tdi <= data[30]; #(5*`CLK_PERIOD) result[30] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
76 tdi <= data[31]; #(5*`CLK_PERIOD) result[31] <= tdo;  tck <= 1'b1;
     #(5*`CLK_PERIOD) tck <= 1'b0;
77
78 tdi <= addr[0]; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <=
     1'b0;
79 tdi <= addr[1]; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <=
     1'b0;
80 tdi <= addr[2]; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <=
     1'b0;
81 tdi <= addr[3]; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <=
     1'b0;
82 tdi <= addr[4]; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <=
     1'b0;
83 tdi <= addr[5]; #(5*`CLK_PERIOD) tck <= 1'b1; #(5*`CLK_PERIOD) tck <=
     1'b0;
84 tms <= 1'b1;
85 tdi <= 1'b0; #(10*`CLK_PERIOD) tck <= 1'b1; #(10*`CLK_PERIOD) tck <= 1'b0;
```

```
86  // goto Update-DR state
87  tdi <= 1'b0;
88  tms <= 1'b1; #(5*`CLK_PERIOD) tck <= 1'b1; #(10*`CLK_PERIOD) tck <= 1'b0;
89  // goto RUN_TEST_IDLE state
90  tms <= 1'b0; #(5*`CLK_PERIOD) tck <= 1'b1; #(10*`CLK_PERIOD) tck <= 1'b0;
91 end
92 endtask
```

Auflistung B.3: jtag_dmi_write.vh

C Software-Bedienung

C.1 Eclipse

C.1.1 OpenOCD Pfadeinstellung

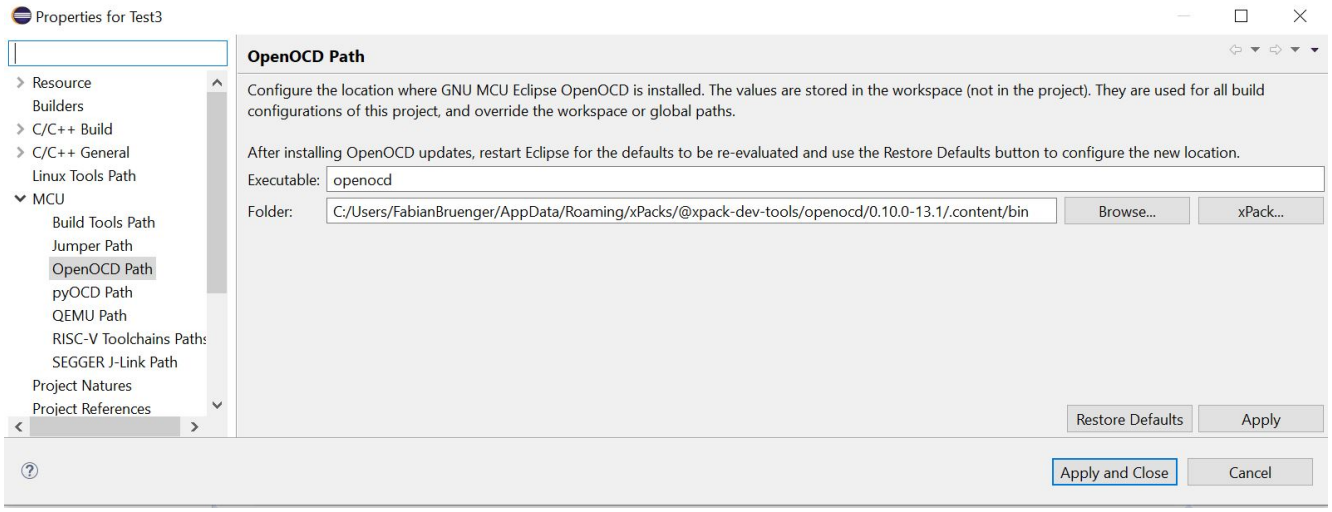

Abbildung C.1: Einstellungen für OpenOCD Pfad

C.1.2 Debugger Einstellung

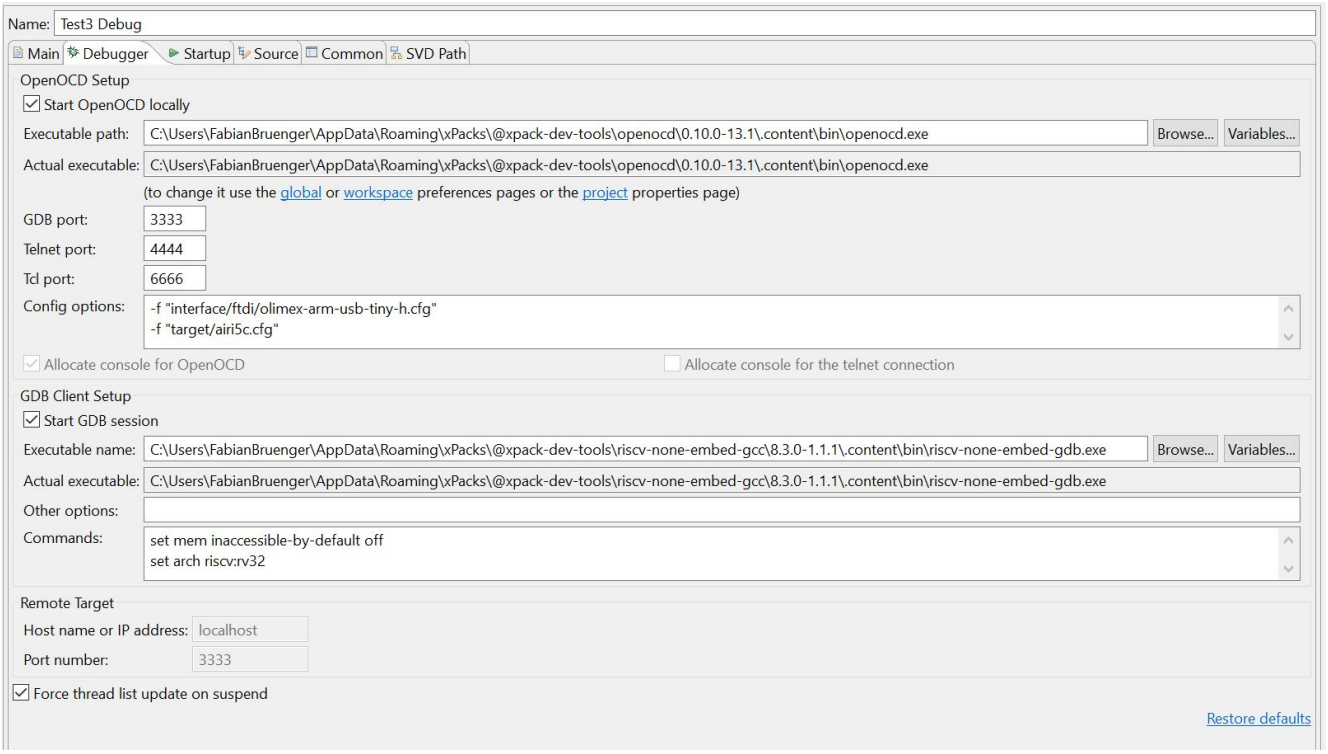

Abbildung C.2: Einstellungen für Debugging

```
if { [info exists CHIPNAME] } {
 set _CHIPNAME $CHIPNAME
} else {
 set _CHIPNAME airi5c
}
reset_config none
adapter_khz 1000
jtag newtap $_CHIPNAME tap -irlen 5 -ircapture 0x01 -expected-id 0x10001001
target create airi5ctarget riscv -chain-position airi5c.tap
```

Auflistung C.1: airi5c.cfg

C.1.3 Building Tools Pfadeinstellung

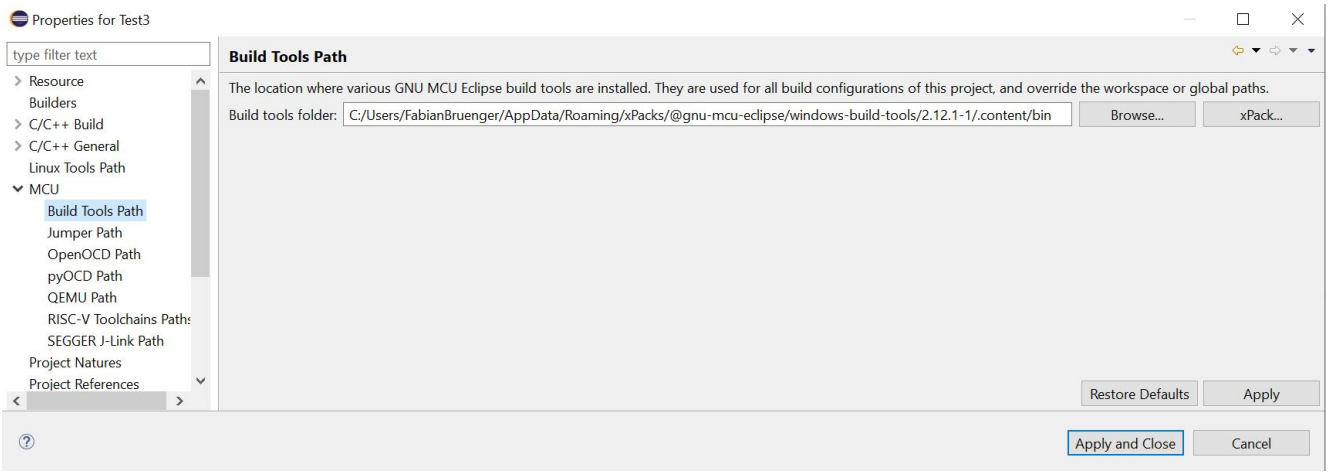

Abbildung C.3: Einstellungen für Building Tools

C.1.4 Toolchain Pfadeinstellung

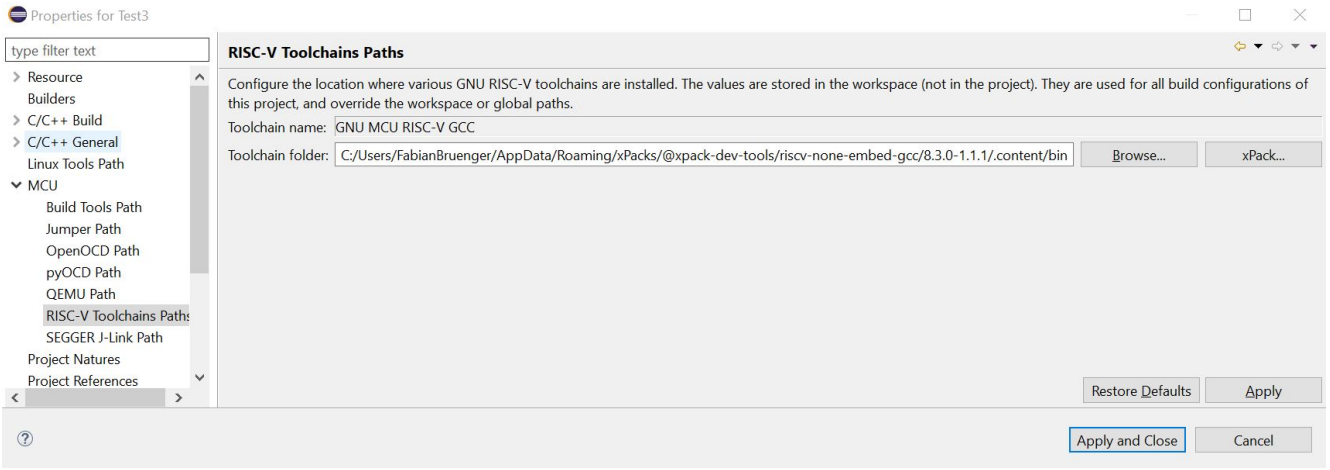

Abbildung C.4: Toolchain Pfadeinstellung

C.1.5 Gitlab

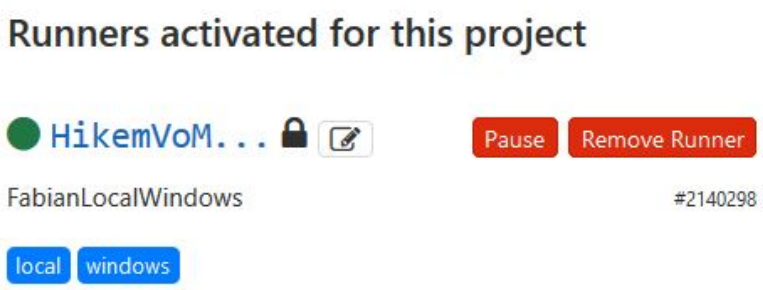

Abbildung C.5: Gitlab-Runner

C.2 Vivado

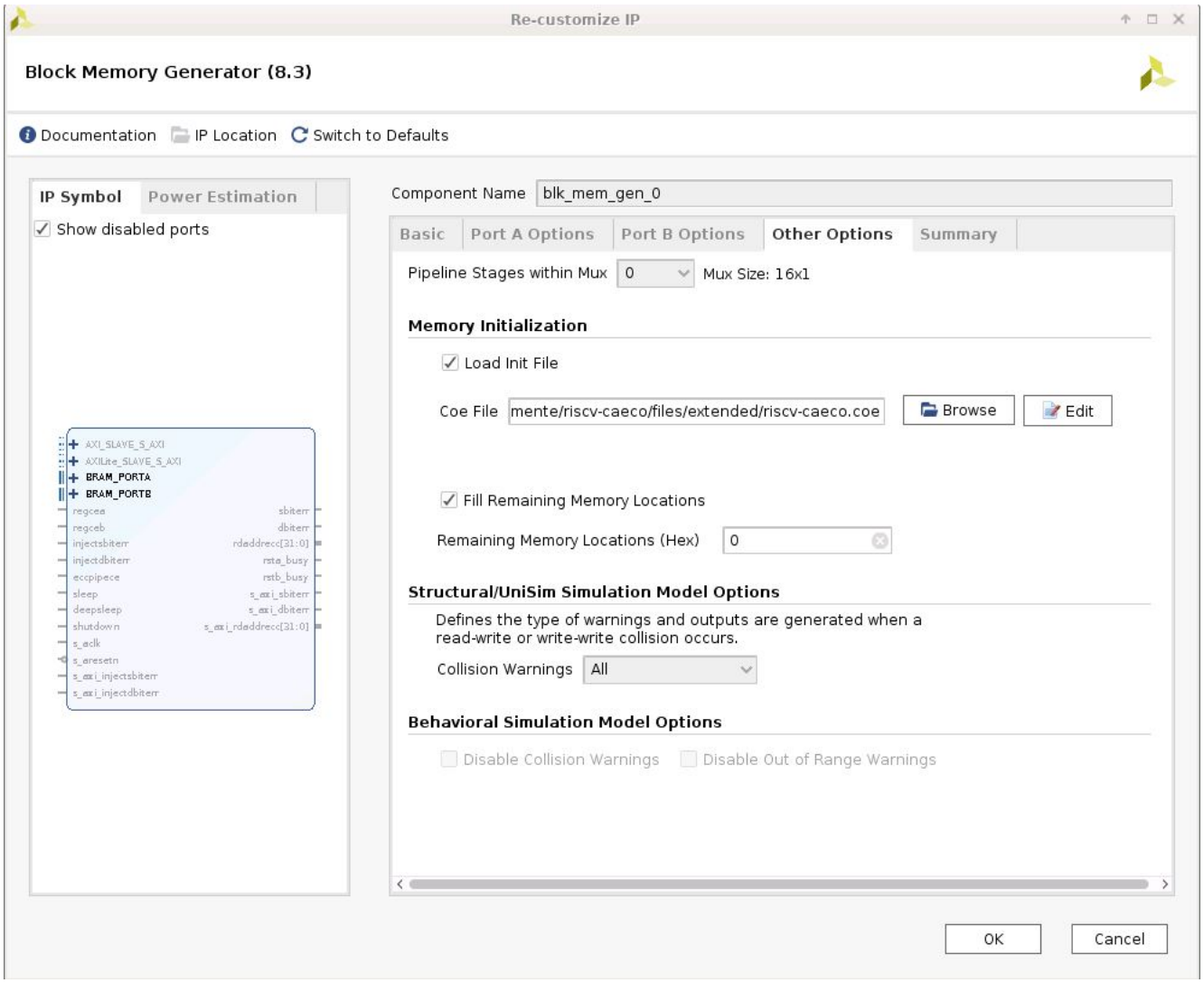

Abbildung C.6: Einstellungen zum Laden des COE Files für BRAM

D Verilog Module und Blockschaltbilder

D.1 Verilogmodule

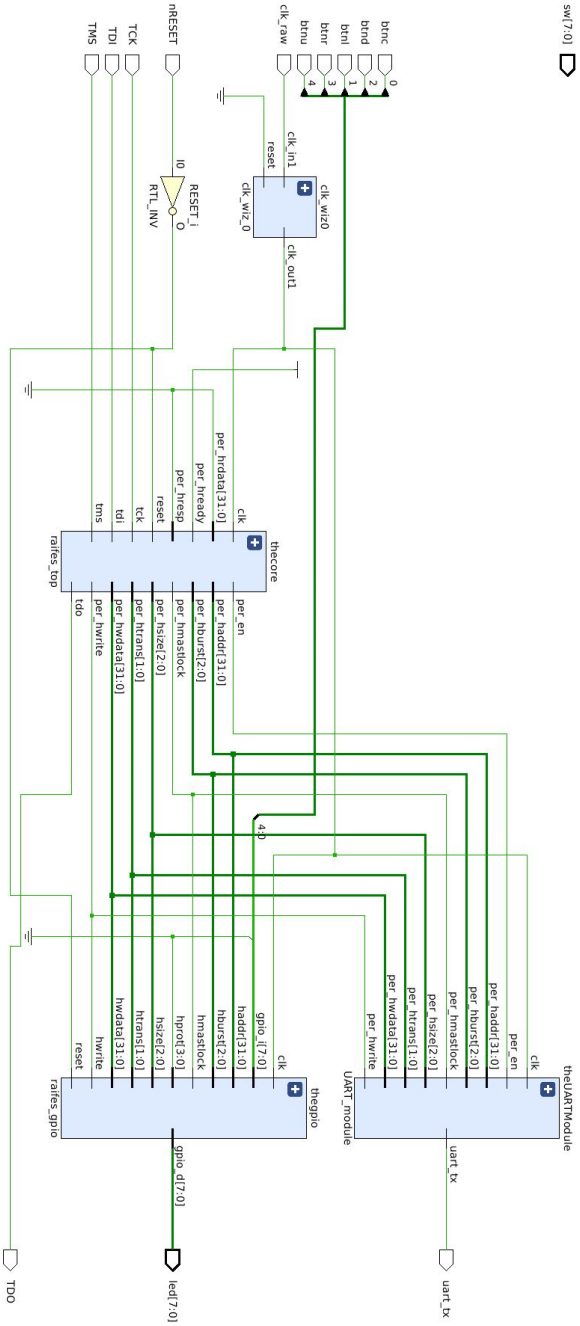

Abbildung D.1: FPGA Wrapper Modul

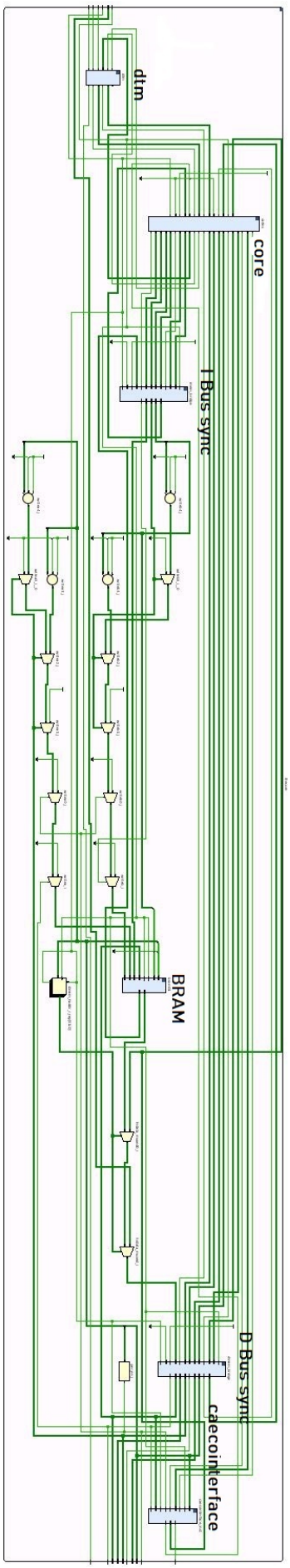

Abbildung D.2: Raifes Top Modul

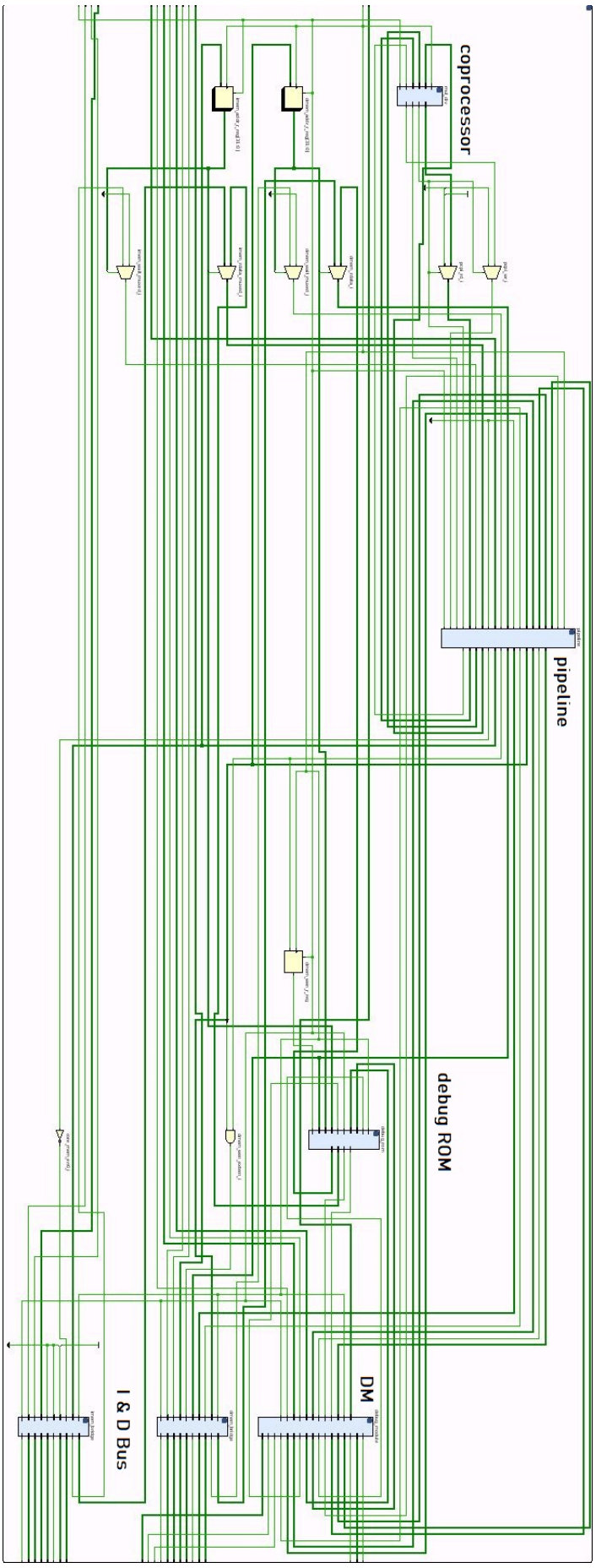

Abbildung D.3: Raifes Core Modul

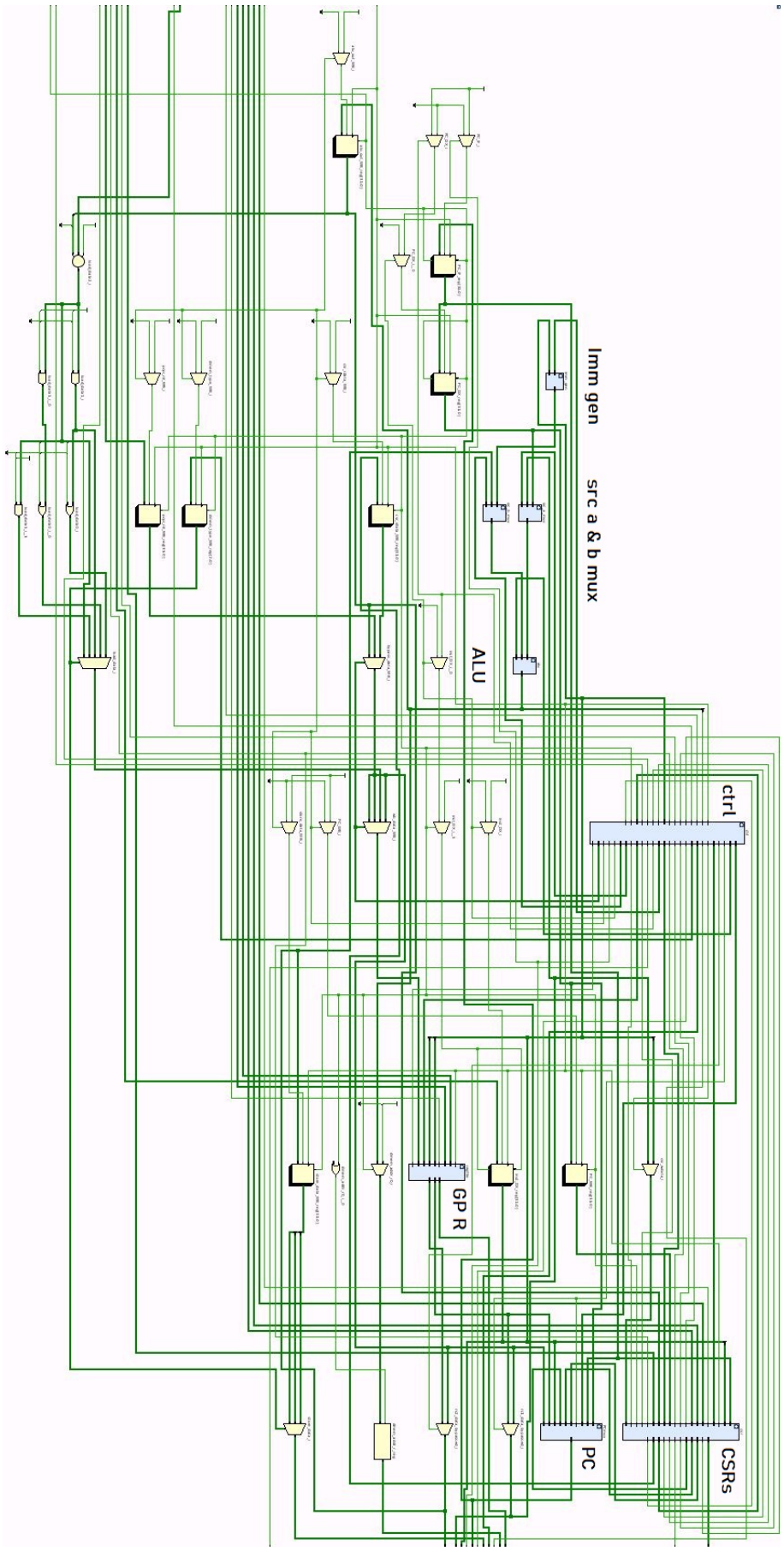

Abbildung D.4: Pipeline Modul

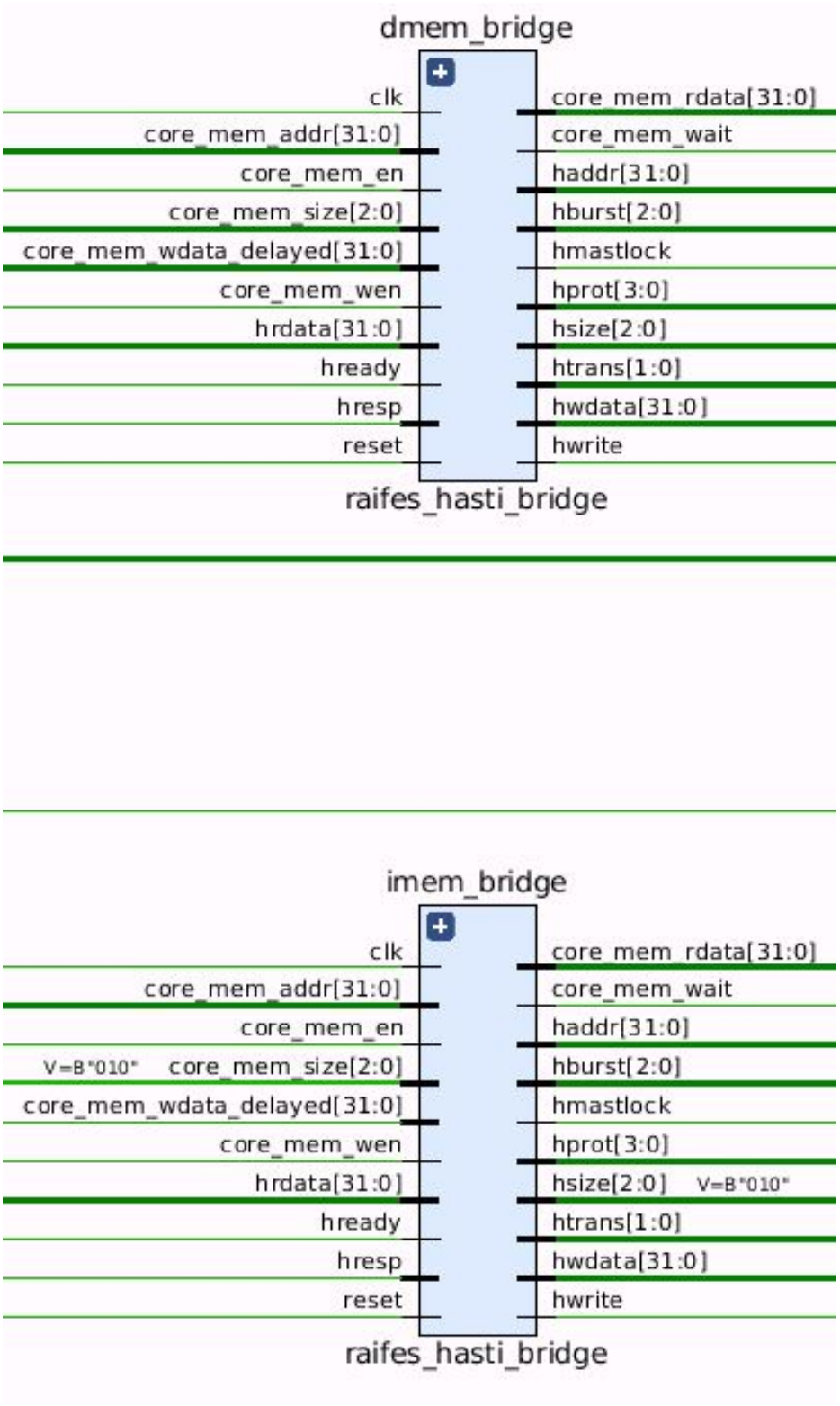

Abbildung D.5: I und D Bus

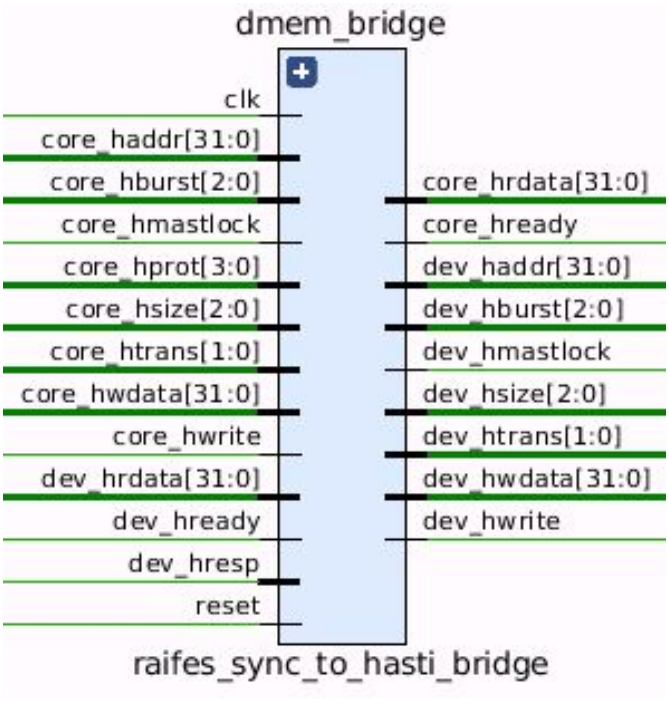

Abbildung D.6: D Bus sync

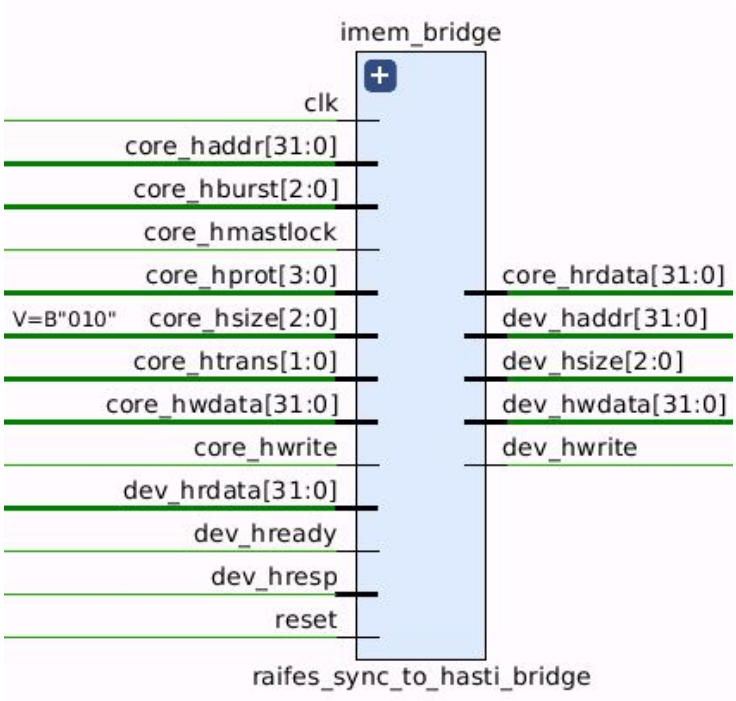

Abbildung D.7: I Bus sync

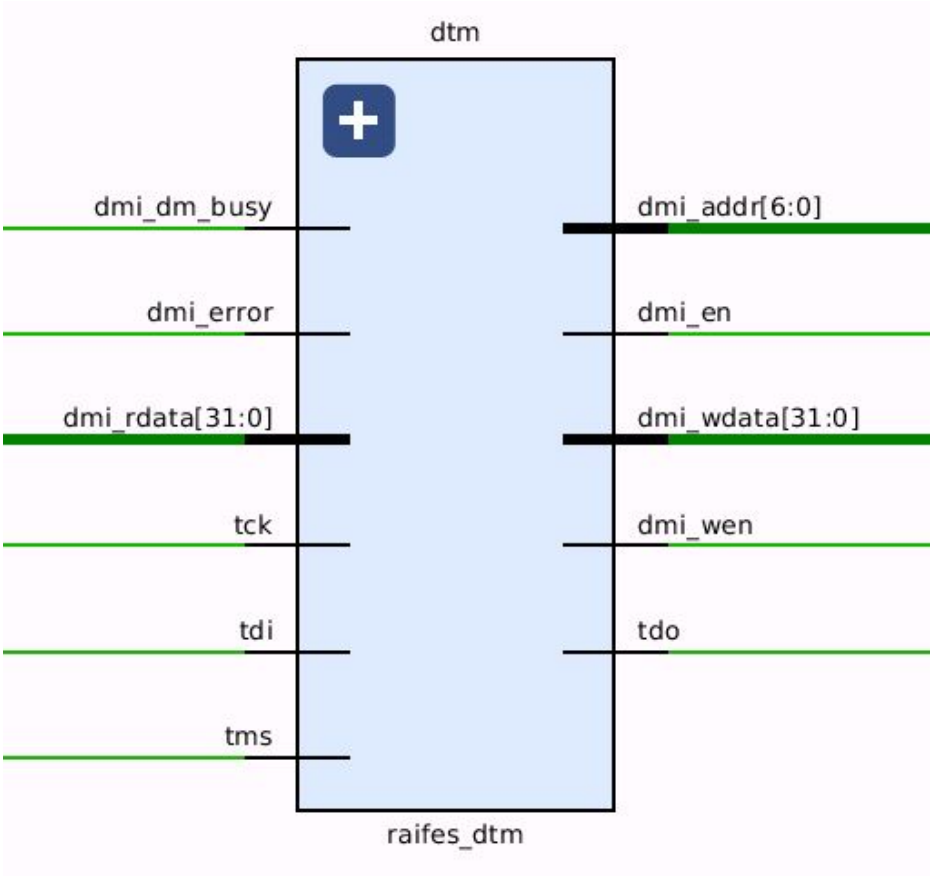

Abbildung D.8: DTM

Abbildung D.9: DM

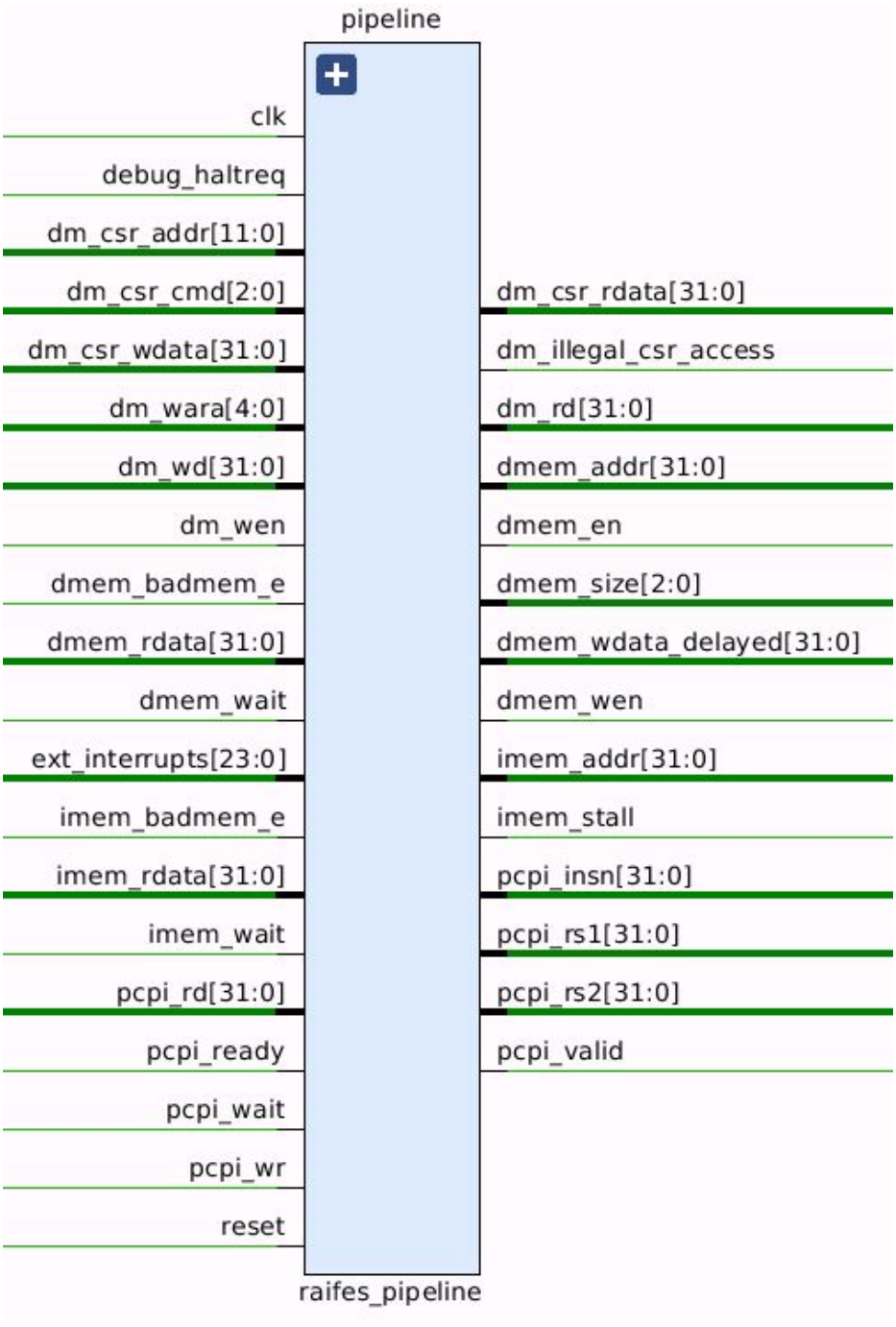

Abbildung D.10: Pipeline

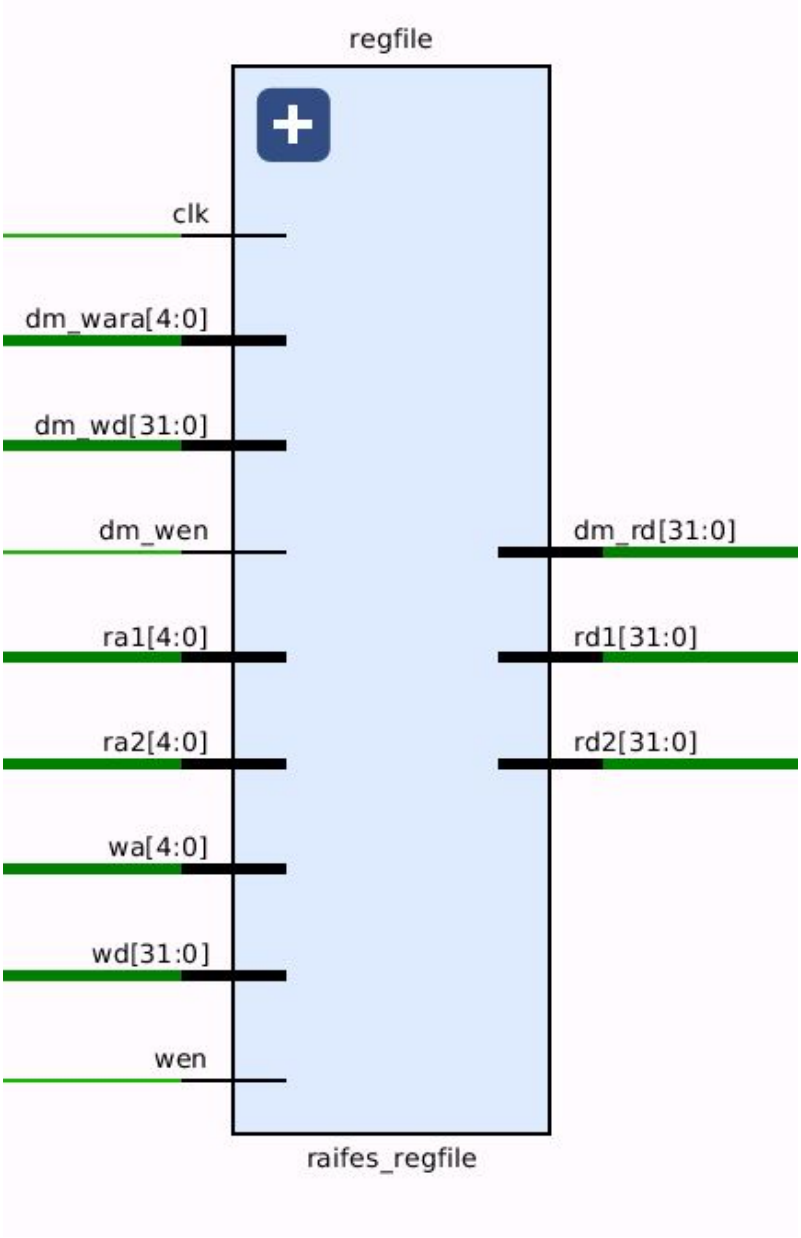

Abbildung D.11: Regfile

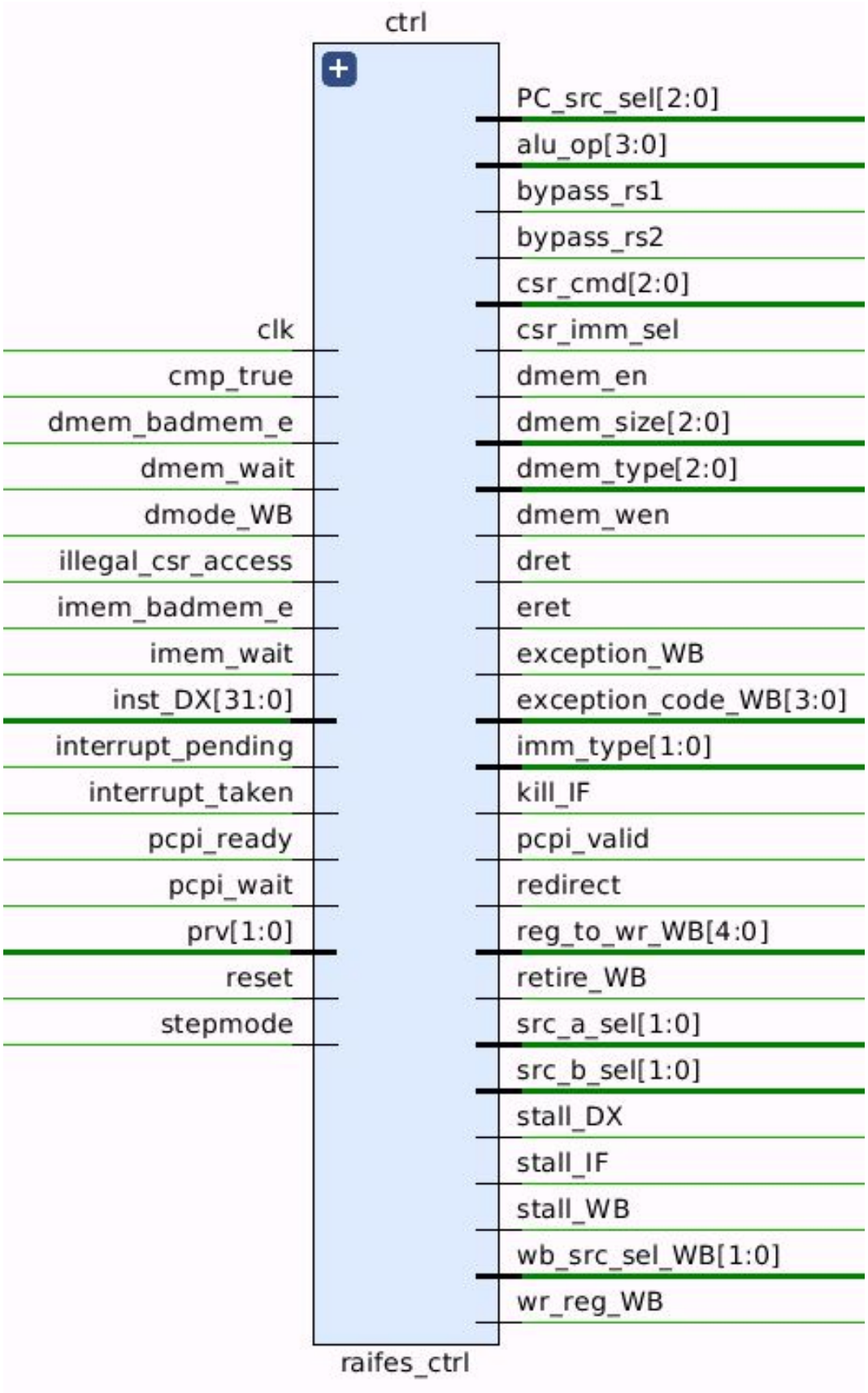

Abbildung D.12: Steuerwerk

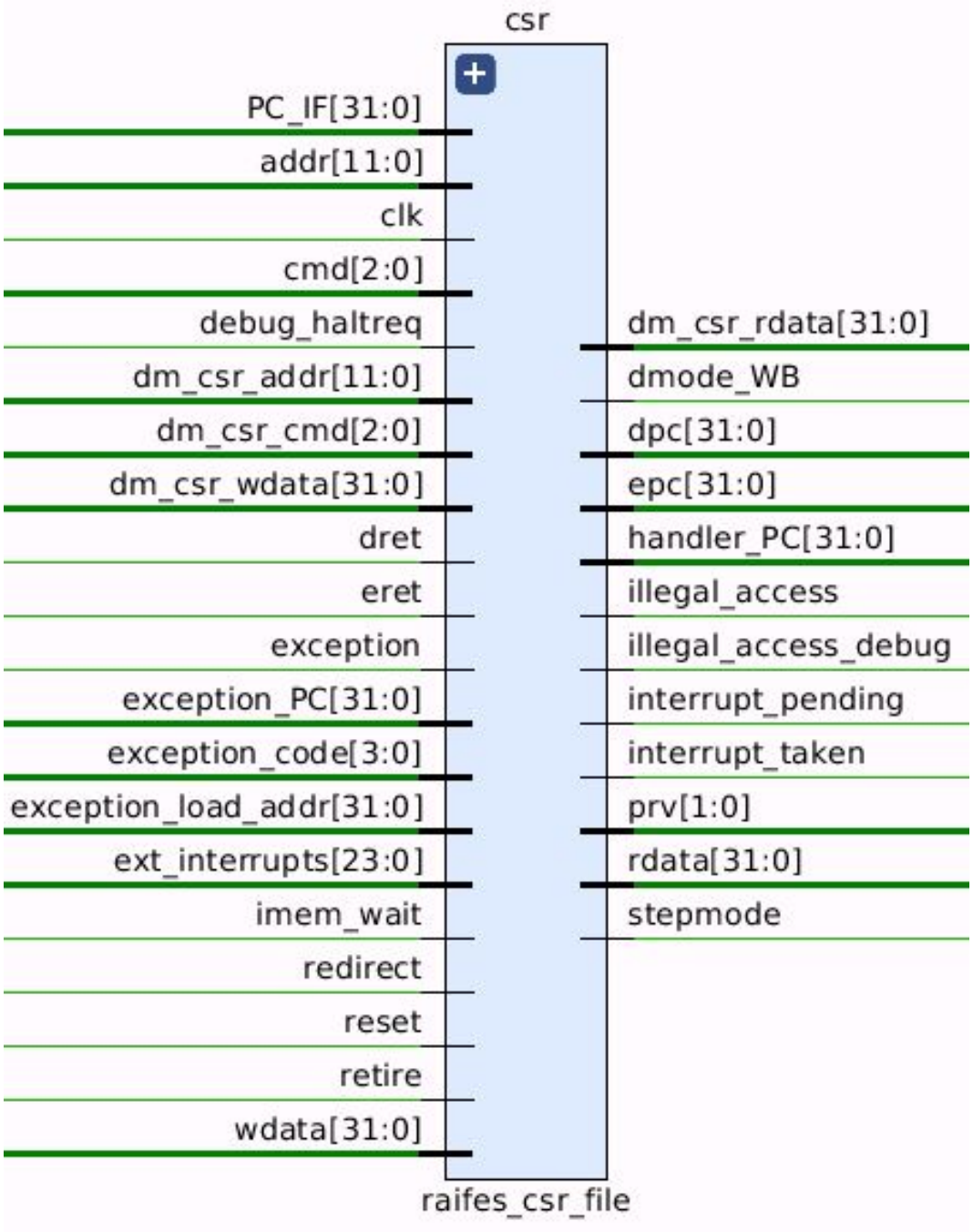

Abbildung D.13: CSRs

D.2 Blockschaltbilder

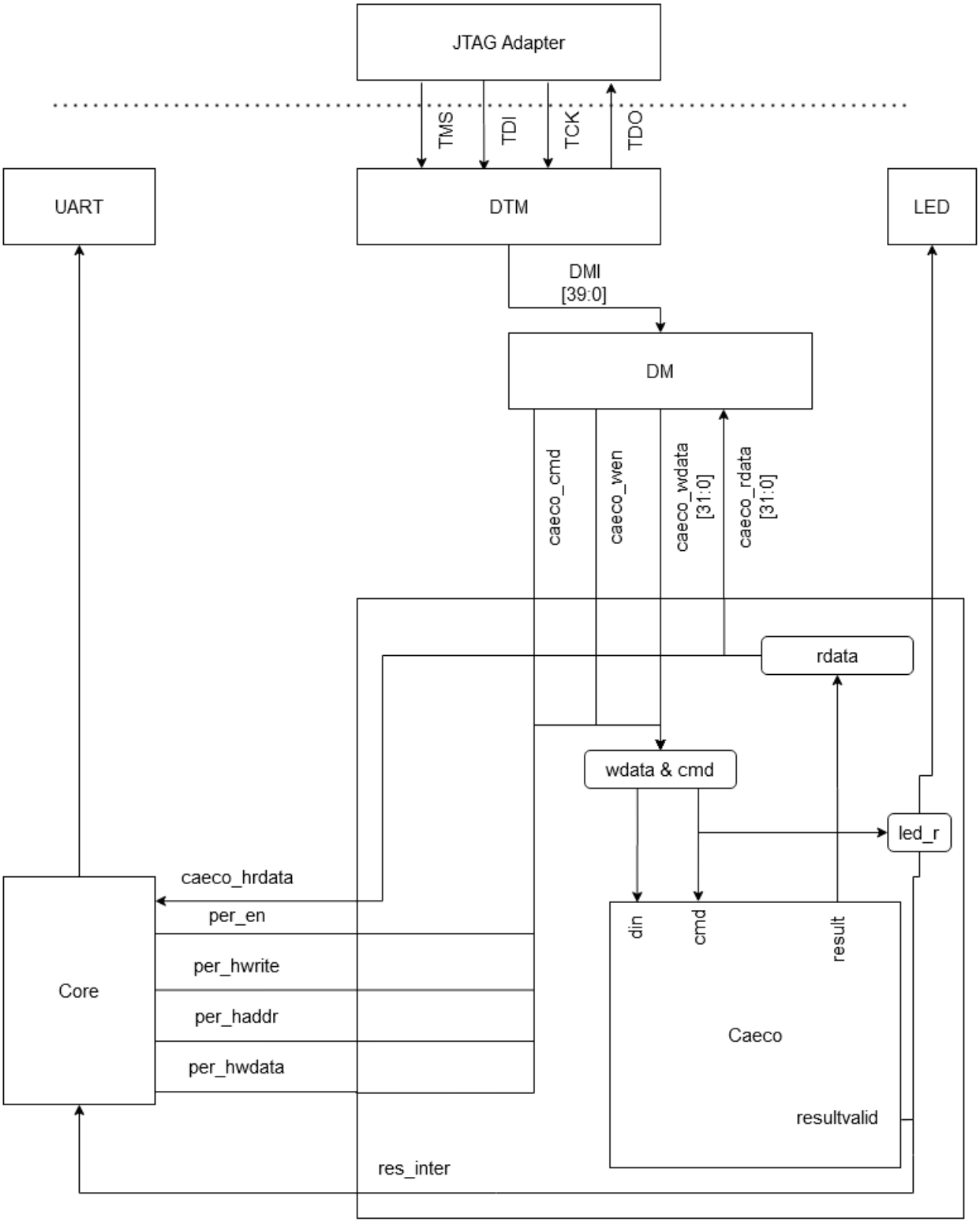

Abbildung D.14: Caecointerface BSB