Technische Hochschule Augsburg

Bachelorarbeit

Studiengang
Technische Informatik

Implementierung einer Debug-Infrastruktur für den PicoNut/RISC-V-Prozessor auf FPGA-Hardware

Verfasser:
Johannes Hofmann
johannes.hofmann1@tha.de

im Fachgebiet Effiziente Eingebettete Systeme

Technische Hochschule
Augsburg
An der Hochschule 1
86161 Augsburg
Telefon: +49 (0)821-5586-0
Fax: +49 (0)821-5586-3222
info@tha.de

Verfasser: Johannes Hofmann

Prüfer: Prof. Gundolf Kiefer
Zweitprüfer: Prof. Hubert Högl

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

1.1 Motivation

Die Ausführung zunehmend komplexer Software stellt hohe Anforderungen an Transparenz, Testbarkeit und Fehlersuche, insbesondere wenn auf Systemen, wie dem PicoNut-System nicht nur leichtgewichtige Programme, sondern vollständige Systeme wie Linux und sogar anspruchsvolle Demonstratoren wie Doom ausgeführt werden sollen. Gerade auf FPGA-Hardware, wo sich Hardware- und Softwarefehlerbilder oft überlagern und sich die Architektur iterativ weiterentwickelt, ist eine leistungsfähige Debug-Infrastruktur entscheidend, um Funktionalität zuverlässig zu verifizieren und Probleme reproduzierbar einzugrenzen.

Gleichzeitig soll der PicoNut-Prozessor als Lernplattform dienen, auf der Studierende und Entwickelnde die inneren Abläufe eines Prozessors nachvollziehen können: Register- und Speicherzugriffe, Interrupts und das Zusammenspiel mit Peripherie werden erst durch Einblick in die internen Zustände des Chips wirklich verständlich. Eine gut integrierte Debug-Lösung schafft damit nicht nur die Grundlage, komplexe Software-Stacks stabil auszuführen, sondern macht den PicoNut-Prozessor auch didaktisch wertvoller, weil sie die „Blackbox“ Prozessor in der Ausführung öffnet und systematisches Debugging als zentrale Kompetenz im Hardware-/Software-Codesign vermittelt.

Um eingebettete Systeme, wie den PicoNut, zu debuggen benötigt das Zielsystem eine entsprechende Hardwareunterstützung.

1.2 Ziel der Arbeit

Das Ziel dieser Arbeit ist die Entwicklung einer Debug-Infrastruktur für das PicoNut-System. Diese soll vollständig in C++/SystemC implementiert und auf echter FPGA-Hardware lauffähig sein. Die Implementierung erfolgt auf Basis der *RISC-V Debug Specification* [9] in der Version 1.0. Durch die Möglichkeit, Programme zu debuggen, die auf echter FPGA-Hardware ausgeführt werden, können komplexe Hard- und Softwarekomponenten effizienter analysiert werden.

Der interne Block-RAM der Membrana wird derzeit während der Synthese mit dem gewünschten Programm initialisiert. Um anschließend ein anderes Programm auszuführen, muss der PicoNut jedoch resynthetisiert werden. Da schon die Synthese eines einfachen PicoNut-Systems mit wenigen Minuten Laufzeit schon sehr zeitaufwendig ist, soll eine Möglichkeit geschaffen werden, Programme effizient zur Laufzeit auszutauschen. Dadurch wird die Softwareentwicklung auf dem PicoNut deutlich effizienter, da schneller direkt auf echter FPGA-Hardware getestet werden kann.

Ein weiterer Vorteil des Ladens über die Debug-Infrastruktur besteht darin, dass jeder Speicher im Adressraum des Prozessors beschrieben werden kann. Dadurch ist es möglich, auch Speicher außerhalb des FPGAs zu initialisieren. Da der Block-RAM von FPGAs stark begrenzt ist, werden große Programme (z.B. Linux) künftig in externem Speicher liegen. Mit der Debug-Infrastruktur lassen sich diese Inhalte in den externen Speicher laden, was mit einer Resynthese nicht möglich ist.

1.3 Aufbau der Arbeit

Zu Beginn der Arbeit wird in Kapitel 2 eine Einführung in das PicoNut-Projekt, sowie in den Aufbau des PicoNut-Prozessors gegeben. Anschließend folgt eine Vorstellung der Debug-Kette und deren Tools: GDB, OpenOCD und JTAG. Daraufhin wird das Debugging eines RISC-V-Prozessors erläutert.

In Kapitel 3 werden vorhandene Debug-Infrastrukturen vorgestellt. Dazu zählt eine nicht synthetisierbare Variante für den PicoNut-Prozessor und ein anderer RISC-V-Prozessor mit einer synthetisierbaren Version.

In Kapitel 4 werden zum einen die Anforderungen der Debug-Infrastruktur definiert, zum anderen die Anforderungen, die ein RISC-V-Prozessor erfüllen muss, um mit dieser Kompatibel zu sein.

Danach wird in Kapitel 5 die Implementierung der Debug-Infrastruktur im Detail betrachtet, insbesondere der Debug-Prozessormodus, die Erweiterung des Prozessors und die Debug-Peripherie.

In Kapitel 6 werden die Ergebnisse der Arbeit gezeigt. Dazu gehört zum einen ein Synthesebericht über den Hardwareverbrauch der Debug-Infrastruktur zum anderen die Auswertung des Zeitaufwandes Programme im Hauptspeicher auszutauschen.

Abschließend wird in Kapitel 7 die Arbeit zusammengefasst und ein Ausblick auf zukünftige Entwicklungen gegeben.

2 Grundlagen

2.1 Einführung in das PicoNut-Projekt

2.1.1 Projektziele

Das PicoNut-Projekt der Technischen Hochschule Augsburg, das im Rahmen der EES-Arbeitsgruppe durchgeführt wird [13, 3], verfolgt seit Beginn des Jahres 2024 das Ziel, einen modularen RISC-V-Prozessor für die Forschung und Lehre zu entwickeln. Der PicoNut-Prozessor soll auf gängiger FPGA-Hardware lauffähig sein und einen Simulator bereitstellen. Die Teilkomponenten sind zudem flexibel austauschbar und ermöglichen so ein konfigurierbares und modulares System. Die Hardware wird mit C++/SystemC modelliert [1]. Für die Synthese wird *ICSC* [8] zusammen mit der *Yosys*-Toolkette verwendet [15].

Eine Besonderheit des PicoNut-Projekts ist die Einführung sogenannter *Soft-Hardware*. Dabei handelt es sich um Hardwaremodule, die vollständig in C++ implementiert sind und selbst nicht synthetisierbar sind. Sie weisen jedoch deutlich bessere Simulationszeiten auf als äquivalente simulierte, synthetisierbare Hardware. Dadurch können effizient schnelle Simulatoren gebaut werden.

Zu den Entwicklungszielen des PicoNut-Projekts zählen die Unterstützung von Betriebssystemen wie FreeRTOS und Linux sowie die Integration von Hardwarebeschleunigern für KI-Anwendungen. Das PicoNut-Projekt findet bereits Anwendung in der Lehre und Forschung, an der Technischen Hochschule Augsburg. [12]

2.1.2 Gesamtsystem

Das PicoNut-System besteht aus einem Prozessor, dem Systembus sowie den an diesen angeschlossenen Peripheriemodulen. Eine schematische Übersicht des Gesamtsystems ist in Abbildung 2.1 dargestellt.

Der Prozessor setzt sich aus dem Nucleus und der Membrana zusammen. Der Nucleus ist der Rechenkern und die Membrana dient als Speicher- und Systembusschnittstelle. Der Nucleus und die Membrana kommunizieren über eine performante IPort-DPort-Schnittstelle. Der Nucleus ist zum Zeitpunkt der Erstellung dieser Arbeit als Einkerner implementiert und unterstützt den RISC-V-Befehlssatz RV32-IA. Die Membrana ermöglicht den Zugriff auf die Register der Peripherien, sowie das Laden von Instruktionen über den Systembus. Darüber hinaus verfügt sie über einen

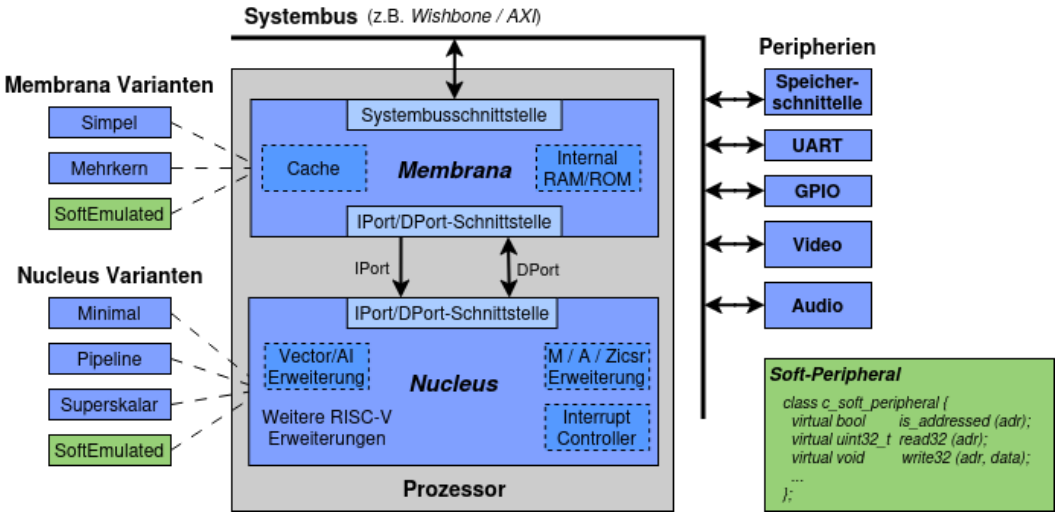

Abbildung 2.1: Übersicht des PicoNut-Systems. Bild in Anlehnung an Strukturbild von Gundolf Kiefer [4]

internen, als Block-RAM realisierten Speicher. Zum Zeitpunkt der Erstellung dieser Arbeit wird das Hauptprogramm, sowie zugehörige Handler-Routinen während der Synthese in den internen Speicher der Membrana geladen. [4]

Der Systembus ist derzeit als Wishbone-Bus mit einer Adressbreite von 32 Bit und einer Datenbreite von ebenfalls 32 Bit ausgeführt. Zum Zeitpunkt dieser Arbeit dient die Membrana als einziger Bus-Master und die Peripherien als entsprechende Slaves. [4]

Der Nucleus, die Membrana oder Peripherien können beliebig mit einer äquivalenten *Soft-Hardware*-Variante ausgetauscht werden, um Simulationszeiten zu verbessern.

2.2 On-Chip-Debugging

2.2.1 GDB (GNU Debugger)

GDB (GNU Debugger) ist ein weit verbreitetes Open-Source-Debugging-Werkzeug, das Entwickelnden ermöglicht, Programme schrittweise auszuführen, Breakpoints zu setzen, Register- und Speicherinhalte auszulesen oder zu verändern, sowie Variablen und Abläufe einer Anwendung detailliert zu untersuchen. Insbesondere bei der Entwicklung komplexerer Software ist Debugging ein zentrales Hilfsmittel, um Fehler effizient zu identifizieren und zu beheben. In dieser Arbeit wurde GDB Version 16.3.90 verwendet. [5]

Darüber hinaus hat GDB auch die Funktion Dateien, z.B. Binärdateien, direkt in den Speicher zu schreiben. Dies ist in der Embedded-Entwicklung relevant, bei der Programme auf Mikrocontrollern oder eigener Hardware ausgeführt werden. Für das Debugging ist Speicherzugriff notwendig. Über die vorhandene Infrastruktur können Programme in den Speicher des eingebetten Systems geladen werden. [5]

Für die Kommunikation mit externer eingebetteter Hardware verwendet GDB eine Schnittstelle zu OpenOCD, der als GDB-Server fungiert. Die Verbindung zwischen GDB und OpenOCD erfolgt nach entsprechender Konfiguration von GDB, über das standardisierte GDB Remote Serial Protocol (RSP). [5] GDB sendet dabei seine Befehle, wie zum Setzen eines Breakpoints oder zum Lesen oder Schreiben von Speicher, über eine TCP/IP-Verbindung an OpenOCD, standardmäßig auf Port 3333. OpenOCD übersetzt diese Remote-Protokoll-Kommandos anschließend in die hardwareabhängigen Signale, führt sie auf dem Zielsystem aus und übermittelt die Ergebnisse über dieselbe RSP-Verbindung zurück [14]. Für GDB erscheint die Zielhardware dadurch wie eine normale, ferngesteuerte Debug-Umgebung. Eine vollständige Debug-Kette für RISC-V-Systeme ist in Abbildung 2.3 dargestellt, wobei GDB der Debugger ist. [9]

Um ein RISC-V-System korrekt debuggen zu können, muss GDB so konfiguriert werden, dass nur die tatsächlich vom Zielsystem implementierten General-Purpose- und Control-and-Status-Register verwendet werden. Ohne die Konfiguration liest GDB, alle standardisierten RISC-V-Register aus. Nicht implementierte Register werden dann mit dem Wert `0` dargestellt, was die Ausgabe durch die Anzahl an Registern schnell unübersichtlich macht. Ansonsten benötigt GDB nur noch die mit Debug-Symbolen kompilierte Binärdatei, die auch auf dem Zielsystem ausgeführt wird, und es kann, wie bei nativ ausgeführter Software, debuggt werden.

2.2.2 OpenOCD

OpenOCD (Open On-Chip Debugger) ist ein weit verbreitetes Open-Source-Werkzeug für das Hardware-Debugging und die Programmierung von Mikrocontrollern sowie selbst entwickelter Hardware. Es dient als Vermittlungsinstanz zwischen dem Debugger, wie GDB, auf dem Hostrechner und der Zielhardware. [14] In dieser Arbeit wurde OpenOCD Version 0.12.0 verwendet.

Es stellt dazu zwei zentrale Schnittstellen bereit: eine *GDB-Server-Schnittstelle* für die Kommunikation mit GDB über das RSP, sowie Hardware-Transportschnittstellen wie JTAG, über die OpenOCD direkten Zugriff auf das Zielsystem erhält. Um mit dem Host zu kommunizieren wird ein JTAG-Adapter benötigt, welcher eine USB-Schnittstelle hat und auf JTAG-Signale

übersetzt. Eine vollständige Debug-Kette für RISC-V-Systeme ist in Abbildung 2.3 dargestellt, wobei OpenOCD als Debug Übersetzer fungiert. Das Projekt wurde ursprünglich von Dominic Rath an der Technischen Hochschule Augsburg initiiert und hat sich seitdem zu einem weltweit eingesetzten Standardwerkzeug entwickelt. [14]

Damit OpenOCD korrekt mit RISC-V-Systemen interagieren kann, muss OpenOCD entsprechend konfiguriert werden. Dabei sind insbesondere folgende Punkte relevant:

- **Zielsystem-Typ:** Es muss angegeben werden, dass es sich um ein RISC-V-System handelt.
- **Debug-Schnittstelle:** Es muss festgelegt werden, dass JTAG verwendet wird, sowie die Taktfrequenz und weitere Parameter des konkreten Adapters.
- **Debug-Zugriffsmechanismus:** Es muss definiert werden, wie der Zugriff auf Speicher erfolgt, z. B. über abstrakte Befehle des RISC-V Debug Modules.
- **TAP-Identifikation:** Der Identifikationscode des TAP muss angegeben werden, damit OpenOCD das richtige Gerät anspricht. Der Identifikationscode ist im TAP der Zielhardware hinterlegt (siehe Kapitel 2.2.3.3).

2.2.3 JTAG

2.2.3.1 Übersicht

Der Begriff JTAG wird häufig synonym für den IEEE-Standard 1149.1 [7] verwendet. JTAG ist ein Standard zum Testen und Debuggen von Integrierten Schaltungen. Eine JTAG-fähige Hardware enthält spezielle Module, die im Normalbetrieb keinen Einfluss auf die restliche Schaltung haben. Erst durch die Aktivierung des sogenannten Test Access Port (TAP) wird der Zugriff auf interne Signale ermöglicht, wodurch Test- und Debug-Funktionen ausgeführt werden können.

Um von einem Computer aus mit der JTAG-Schnittstelle kommunizieren zu können, wird ein JTAG-Adapter benötigt. Dieser dient als Schnittstelle zwischen der Hardware des Zielsystems und OpenOCD auf dem Hostrechner. Der Adapter wandelt dabei die Signale von gängigen Kommunikationsschnittstellen (z. B. USB) in die entsprechenden JTAG-Signale um. Eine vollständige Debug-Kette für RISC-V-Systeme ist in Abbildung 2.3 dargestellt, wobei der JTAG-Adapter die Debug-Transport-Hardware ist.

Der TAP besteht aus mehreren Komponenten: den Pins, die mit dem JTAG-Adapter verbunden sind, verschiedenen Schieberegistern zum Empfangen und Senden von

Daten, sowie dem TAP-Automaten, der die interne Steuerung des TAP übernimmt. [7]

2.2.3.2 TAP-Schnittstellen

Die Pins für die JTAG-Schnittstelle dienen ausschließlich der Steuerung und Kommunikation mit dem JTAG-Adapter. Sie dürfen daher nicht für andere Zwecke verwendet werden. In Tabelle 2.1 sind die einzelnen Signale des TAP aufgeführt und beschrieben.

Name	Beschreibung
`tck`	Test Clock, Taktleitung des TAP, bestimmt die Synchronisation der internen Zustandsübergänge.
`tms`	Test Mode Select, Steuerleitung für den TAP-Automaten, Pegel bei einer steigenden Flanke von `tck` legt den nächsten Zustand des TAP-Automat fest.
`tdi`	Test Data Input, Serieller Dateneingang, über den Daten in Schieberegister geladen werden.
`tdo`	Test Data Output, Serieller Datenausgang, über den Daten aus den Registern ausgelesen werden.
`trst_n`	Test Reset, Optionales Signal zum asynchronen Zurücksetzen des TAP-Automaten (aktiv low).

Tabelle 2.1: Signale der JTAG-Schnittstelle

2.2.3.3 TAP-Register

Der TAP verfügt über mehrere standardisierte Register, die über die JTAG-Schnittstelle zugänglich sind. Es gibt mehrere Data-Register und ein Instruction-Register, hier mit Breite von fünf Bit. Die Standarddatenregister IDCODE und BYPASS sind in Tabelle 2.2 beschrieben. Je nach Anwendung können weitere Data-Register implementiert werden um spezielle Anforderungen zu erfüllen.

Es wird ausgewählt, auf welches Data-Register zugegriffen wird, indem zuerst die entsprechende Adresse des gewünschten Datenregisters in das Instruction-Register geschrieben wird.

Name	Breite	Adresse	Beschreibung
IDCODE	32 Bit	0x0	Enthält eine eindeutige Kennung des Bausteins.
BYPASS	1 Bit	0x1	Einfaches Ein-Bit-Register, das das betreffende Gerät im Datenpfad überbrückt um auf weitere JTAG-Geräte in der Kette zuzugreifen.

Tabelle 2.2: Standard Data-Register des TAP

2.2.3.4 TAP-Automat

Der TAP-Automat steuert den Ablauf sämtlicher JTAG-Operationen. Er definiert, in welchem Zustand sich der TAP befindet, und legt fest, welches Register (z. B. Instruction- oder Data-Register) ausgewählt ist. Der Automat wird durch die Signale `tck` und `tms` gesteuert und besteht aus 16 Zuständen.

Zu den Aufgaben des TAP-Automaten gehören:

- die Auswahl, ob das Instruction-Register oder ein Data-Register angesteuert wird,
- das Schieben von Daten durch das ausgewählte Register,
- das Aktualisieren von Registern, sobald eine neue Instruktion oder ein Datensatz übernommen werden soll,
- sowie das Zurücksetzen in einen definierten Ausgangszustand.

Die Zustandsübergänge erfolgen zum Takt `tck` und werden allein durch den Pegel des Steuersignals `tms` bestimmt. Dadurch kann der TAP deterministisch durch jeden Zustand der Maschine geführt werden. Der vollständige TAP-Automat ist in Abbildung 2.2 dargestellt.

Um auf ein Data-Register zuzugreifen, muss im ersten Schritt die Adresse des Data-Registers in das Instruction-Register geladen werden. Dazu wird der TAP-Automat in den Zustand *Shift-IR* überführt. Anschließend werden die Daten über `tdi` bei jeder steigenden Flanke von `tck` sequenziell in das Instruction-Register geschoben.

Der Zugriff auf ausgewählte Data-Register erfolgt analog im Zustand *Shift-DR*. Die zu schreibenden Daten werden dabei ebenfalls sequenziell über das Signal `tdi` bei jeder steigenden Flanke von `tck` in das ausgewählte Data-Register geschoben. Das Signal `tdo` führt stets den Pegel des 0-ten Bits des Registers. Während eines Schiebevorgangs kann so der vorherige Registerinhalt ausgelesen werden.

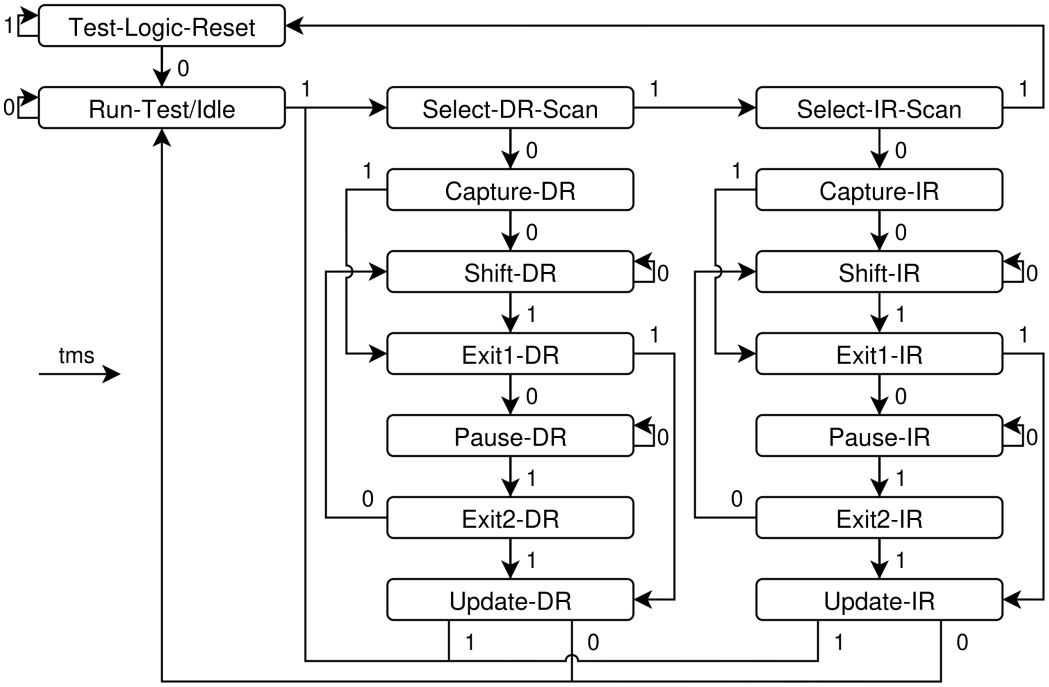

Abbildung 2.2: Zustände des TAP-Automaten. Bild in Anlehnung an [7]

In Tabelle 2.3 sind Beispielsequenzen der JTAG-Signale aufgelistet. Die Sequenzen starten aus dem Ausgangszustand *Test-Logic-Reset*.

Signal	**Reset**	**Zugriff Instruction-Register**	**Zugriff Data-Register (Bsp.: IDCODE)**
`tck`	↑ ↑ ↑ ↑ ↑	↑ ↑ ↑ ↑ ↑ | ↑ ↑ ↑ ↑ ↑ |	↑ ↑ ↑ ↑ ↑ | 32x ↑ |
`tms`	1 1 1 1 1	0 1 1 0 0 | 0 0 0 0 0 |	0 1 0 0 0 | 32x 0 |
`tdi`	– – – – –	– – – – – | *Neuer Wert* |	– – – – – | *Neuer Wert* |
`tdo`	x x x x x	x x x x x | *Alter Wert* |	x x x x x | *Alter Wert* |

Tabelle 2.3: Beispielsignalsequenzen für Zugriffe auf TAP-Register

Im Folgenden werden die einzelnen Zustände beschrieben. Wenn nicht anders erwähnt bleibt bei `tms`=0 der Zustand unverändert.

Im Zustand *Test-Logic-Reset* werden alle internen Register und Zustände zurückgesetzt. Durch Halten von `tms`=1 über mindestens fünf Taktzyklen kann dieser Zustand jederzeit erreicht werden, siehe Tabelle 2.3. Bei `tms`=0 erfolgt der Übergang in den Zustand *Run-Test/Idle*.

Im Zustand *Run-Test/Idle* ist der TAP inaktiv. Dieser Zustand dient dem Warten zwischen zwei Registeroperationen. Bei `tms`=1 wird in *Select-DR-Scan* gewechselt.

Im Zustand *Select-DR-Scan* entscheidet der TAP, ob der Datenpfad oder Instruktionspfad aktiviert wird. Bei `tms`=0 erfolgt der Übergang zu *Capture-DR*, bei `tms`=1 zu *Select-IR-Scan*.

Im Zustand *Capture-DR* werden aktuelle Daten aus der Schaltung in das aktive Datenregister geladen. Dies ermöglicht den anschließenden Zugriff auf die Daten über die serielle Schnittstelle. Bei `tms`=0 erfolgt der Übergang zu *Shift-DR*, bei `tms`=1 zu *Exit1-DR*.

Im Zustand *Shift-DR* können Daten seriell über `tdi` in das aktive Register eingeschoben und über `tdo` ausgelesen werden. Bei `tms`=0 bleibt der TAP in diesem Zustand, um weitere Bits zu schieben. Mit `tms`=1 erfolgt der Übergang zu *Exit1-DR*.

Im Zustand *Exit1-DR* wird entschieden, ob der Schiebevorgang beendet oder unterbrochen werden soll. Bei `tms`=0 erfolgt der Übergang zu *Pause-DR*, bei `tms`=1 zu *Update-DR*.

Im Zustand *Pause-DR* kann der Datenfluss temporär angehalten werden, ohne den Inhalt der Register zu verlieren. Bei `tms`=1 erfolgt der Übergang zu *Exit2-DR*.

Im Zustand *Exit2-DR* wird festgelegt, ob der Schiebevorgang fortgesetzt oder abgeschlossen wird. Bei `tms`=0 kehrt der TAP-Automat zu *Shift-DR* zurück, bei `tms`=1 zu *Update-DR*.

Im Zustand *Update-DR* werden die zuvor eingeschobenen Daten in das Zielregister übernommen und entsprechende Operationen die die Daten auslösen werden ausgeführt. Anschließend kann bei `tms`=0 in *Run-Test/Idle* gewechselt oder bei `tms`=1 in *Select-DR-Scan* fortgefahren werden.

Im Zustand *Select-IR-Scan* wird der Instruktionspfad vorbereitet. Bei `tms`=0 erfolgt der Übergang zu *Capture-IR*, bei `tms`=1 kehrt der TAP in *Test-Logic-Reset* zurück.

Im Zustand *Capture-IR* wird das Instruction-Register zurückgesetzt. Bei `tms`=0 erfolgt der Übergang zu *Shift-IR*, bei `tms`=1 zu *Exit1-IR*.

Im Zustand *Shift-IR* werden neue Instruktionen seriell über `tdi` eingeschoben und über `tdo` ausgelesen. Bei `tms`=1 erfolgt der Übergang zu *Exit1-IR*.

Im Zustand *Exit1-IR* wird entschieden, ob der Schiebevorgang beendet oder unterbrochen werden soll. Bei `tms`=0 erfolgt der Übergang zu *Pause-IR*, bei `tms`=1 zu *Update-IR*.

Im Zustand *Pause-IR* kann der Schiebevorgang temporär angehalten werden. Bei `tms`=1 erfolgt der Übergang zu *Exit2-IR*.

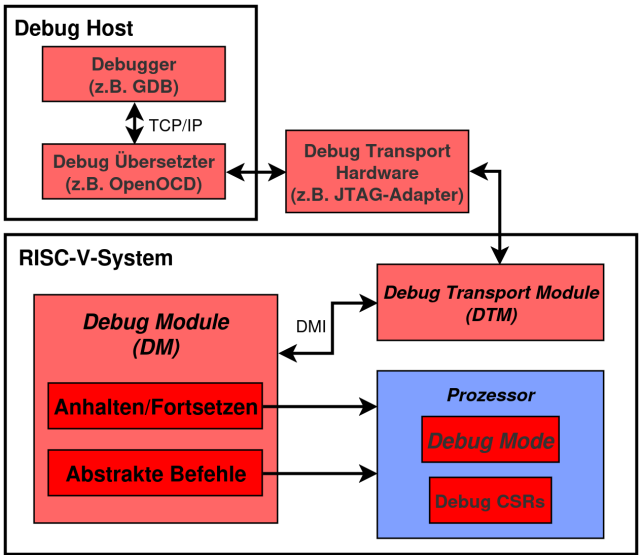

Abbildung 2.3: Übersicht einer Debug-Infrastruktur für ein RISC-V-System

Im Zustand *Exit2-IR* wird entschieden, ob der Schiebevorgang fortgesetzt oder abgeschlossen wird. Bei `tms`=0 kehrt der TAP zu *Shift-IR* zurück, bei `tms`=1 zu *Update-IR*.

Im Zustand *Update-IR* wird der Inhalt des Schieberegisters in das Instruction-Register übernommen. Anschließend kann bei `tms`=0 in *Run-Test/Idle* gewechselt oder bei `tms`=1 in *Select-DR-Scan* fortgefahren werden.

Einzelheiten sind aus [7] entnommen.

2.3 Die RISC-V Debug-Spezifikation

2.3.1 Übersicht

Die *RISC-V Debug Specification* [9] definiert einen standardisierten Mechanismus zum Debugging von RISC-V-Systemen über externe Schnittstellen wie JTAG. Sie ermöglicht das Setzen von Breakpoints, das Anhalten und Fortsetzen des Programmes, sowie eine Einzelschrittausführung. Außerdem wird das Auslesen und Schreiben von Registern und Speicher unterstützt. Die dafür notwendige Infrastruktur basiert auf einer dedizierten Debug-Peripherie, einem Prozessormodus, den Debug-Modus, und zusätzlichen CSRs, die für diverse Debugging-Funktionen genutzt werden. In Abbildung 2.3 ist eine Übersicht über den Aufbau einer Debug-Infrastruktur (rot) für ein RISC-V-System dargestellt.

2.3.2 Debug-Modus

Um einen Prozessor zu debuggen muss er angehalten werden. Das heißt, das Hauptprogramm bleibt stehen und Debugging ist aktiv. Ein Prozessor wird durch folgende Ereignisse angehalten:

- Erreichen eines Breakpoints, der sogenannte `EBREAK`-Befehl.
- Asynchrones Anhalten durch den Debugger.
- Bei aktiver Einzelschrittausführung nach der Ausführung eines Befehls.

Ist der Prozessor angehalten befindet er sich im Debug-Modus. Im Debug-Modus wird als Erstes der aktuelle Wert des Programmzählers in das Debug Program Counter (DPC) CSR geschrieben. Jetzt können Debug Befehle des Debuggers ausgeführt werden wie Zugriffe auf Register und Speicher. Wenn das Hauptprogramm fortgesetzt werden soll, wird der Programmzähler auf den Wert im DPC-Register gesetzt. Jetzt wird das Hauptprogramm mit dem gleichen Kontext fortgesetzt wie vor Betreten des Debug-Modus. Wie der Debug-Modus konkret implementiert wird ist durch die Spezifikation nicht vorgegeben.

2.3.3 Debug Control and Status Register

Der Prozessor erhält für das Debugging vier neue CSRs. Diese sind in Tabelle 2.4 aufgelistet.

Name	Adresse	Beschreibung
DCSR	0x7B0	Debug Control and Status, Speichert den Zustand des Prozessors vor dem Debug-Modus, sowie Informationen für den Debugger über den Prozessor.
DPC	0x7B1	Debug Program Counter, Speichert den PC vor dem Debug-Modus.
DSCRATCH[0..1]	[0x7B2..0x7B3]	Debug Scratch, Speicherregister für Operationen im Debug-Modus.

Tabelle 2.4: Debug CSRs

Das DCSR-Register speichert den Zustand des Prozessors vor dem Betreten des Debug-Modus. Dazu zählt das Privilege-Level und der Grund des Betretens des Debug-Modus. Gründe sind hier ein asynchroner Halt, das Erreichen eines Breakpoints und Einzelschrittausführung. Außerdem werden hier Informationen über den

Prozessor abgespeichert die für den Debugger relevant sind, dazu zählt das Verhalten von Breakpoints in den verschiedenen Privilege-Leveln. Zudem speichert der Debugger im `step`-Bit ab, ob die Einzelschrittausführung aktiv ist.

Das DPC-Register speichert den letzten Wert des Programmzählers vor dem Betreten des Debug-Modus ab. Diese wird beim Verlassen des Debug-Modus wieder in den Programmzähler geladen. Somit wird das Hauptprogramm an derselben Stelle wieder fortgesetzt. Der Debugger kann den Programmzählers des Hauptprogramms im Debug-Modus durch dieses Register lesen.

Die Register DSCRATCH[0..1] können im Debug-Modus benutzt werden um Daten zwischen zu speichern.

2.3.4 Debug-Peripherie

2.3.4.1 Übersicht

Die Debug-Peripherie besteht aus dem Debug Transport Module (DTM) und dem Debug Module (DM). Sie bildet die Brücke zwischen dem RISC-V-Prozessor und der externen Debug-Transport-Hardware. Das DTM bildet die Schnittstelle zwischen der Debug-Transport-Hardware und dem DM. Das DM ist das Herzstück der Debug-Peripherie und beinhaltet die eigentliche Logik. Die Kommunikation zwischen DTM und DM funktioniert über eine nicht spezifizierte Schnittstelle, das Debug Module Interface (DMI). Es ist ein Bussystem bei dem DTM als Master und dem DM als Slave fungiert.

2.3.4.2 Debug Transport Module

Das DTM bildet eine Abstraktionsschicht zwischen Debug-Transport-Hardware und dem DM. Der Standard nach außen zur Debug-Transport-Hardware ist eine JTAG-Schnittstelle. Es ist aber erlaubt DTMs mit anderen Schnittstellen wie SWD, Ethernet, usw. zu entwickeln. Ein JTAG-basiertes DTM hat als Schnittstellen zum einen eine DMI-Schnittstelle Richtung DM, und zum anderen nach außen eine JTAG-Schnittstelle Richtung Debug-Transport-Hardware. Es implementiert grundlegend einen TAP, dieser ist bereits in Kapitel 2.2.3 beschrieben. Zusätzlich werden zwei neue TAP-Register eingeführt, die zum einen Informationen über die Debug-Peripherie beinhalten und zum anderen für die Kommunikation über die DMI-Schnittstelle zuständig sind. Die zusätzlichen Register sind in Tabelle 2.5 aufgeführt.

Name	Breite	Adresse	Beschreibung
DTMCS	32 Bit	0x10	DTM Control and Status, Enthält Statusinformationen über die Debug-Infrastruktur.
DMI	34 Bit+ DMI-Adressbreite	0x11	Debug Module Interface Access, Register für Lese- und Schreibzugriffe auf die DMI-Schnittstelle.

Tabelle 2.5: Zusätzliche TAP-Register für das Debugging

Das DTMCS-Register speichert Informationen über die Debug-Peripherie. Hier wird die Version der Spezifikation festgehalten, nach der die Debug-Peripherie implementiert ist. Außerdem wird die Breite des DMI-Adresssignals hier abgespeichert.

Das DMI-Register ist für den Zugriff auf die DMI-Schnittelle verantwortlich. Durch dieses Register kann der Debugger somit auf Register des DMs zugreifen. Es hält folgende Informationen:

- Ob geschrieben oder gelesen werden soll.
- Die Daten die geschrieben werden sollen oder die gelesenen Daten.
- Die Adresse auf die im DMI-Adressraum zugegriffen werden soll.

2.3.4.3 Debug Module

Das DM bildet das zentrale Steuerelement der Debug-Peripherie. Es dient als Übersetzungsschicht zwischen abstrakten Befehlen des Debuggers, wie den Prozessor anhalten, und der konkreten Ausführung der dieser Befehle. Dazu hat das DM eine DMI-Schnittstelle, über die der Debugger via DTM auf Register des DMs zugreifen kann. Folgend sind wichtige Operationen aufgelistet, die das DM unterstützen kann um Debugging zu ermöglichen:

- Prozessor asynchron Anhalten und Fortsetzen
- Einzelschrittausführung
- General Purpose Register (GPR) lesen und schreiben
- Control and Status Register (CSR) lesen und schreiben
- Speicher lesen und schreiben
- Weitere Operationen wie ein direkter Systembuszugriff

Die Spezifikation macht dabei keine Vorgaben zur konkreten Implementierung der Operationen.

Register an der DMI-Schnittstelle

Alle für diese Arbeit relevanten Register, auf die der Debugger über die DMI-Schnittstelle zugreifen kann, sind in Tabelle 2.6 aufgelistet. Die Register an der DMI-Schnittstelle sind 32 Bit breit.

Name	Adresse	Beschreibung
DATA[0..11]	0x04 - 0x0F	Abstract Data, Speicherregister für Daten.
DMCONTROL	0x10	Debug Module Control, Kontrollregister für das DM.
DMSTATUS	0x11	Debug Module Status, Statusregister für den Zustand des DMs.
ABSTRACTCS	0x16	Abstract Control and Status, Kontroll- und Statusregister für die abstrakten Befehle.
COMMAND	0x17	Abstract Command, Speicherregister für einen abstrakten Befehl.
ABSTRACTAUTO	0x18	Abstract Command Autoexec, Hilft bei der Beschleunigung von Speicherzugriffen.

Tabelle 2.6: Register an der DMI-Schnittstelle

Die Register DATA[0..11] sind für die Speicherung von Daten verantwortlich. Hier werden Register- oder Speicherinhalte für das Schreiben vom Debugger abgelegt und bei Lesezugriffen legt das DM entsprechend die gelesenen Daten hier ab, damit der Debugger auf sie zugreifen kann. Außerdem werden Argumente für abstrakte Befehle hier abgespeichert.

Das DMCONTROL-Register steuert zum einen das DM, als auch den Prozessor. Der Debugger schaltet das DM über dieses Register ein. Außerdem wird über das `haltreq`-Bit der Prozessor asynchron Angehalten und in den Debug-Modus überführt. Mittels des `resumereq`-Bit wird Angefragt den Debug-Modus wieder verlassen und das Hauptprogramm wird fortgesetzt.

Das DMSTATUS-Register speichert zum einen den Status des DMs, als auch den des Prozessors. Der Debugger kann hier lesen nach welcher Version der Spezifikation

das DM implementiert wurde. Außerdem wird hier gespeichert, ob der Prozessor aktuell angehalten ist durch das `allhalted`-Bit, ob der Prozessor gerade das Hauptprgramm ausführt durch das `allrunning`-Bit oder ob die Fortsetzanfrage, durch das `resumereq`-Bit aus dem DMCONTROL-Register, angenommen wurde durch das `allresumeack`-Bit.

Das ABSTRACTCS-Registers, speichert Informationen über die abstrakten Befehle. Dazu zählt wie viele DATA-Register vorhanden sind, sowie ein Error-Code durch das `cmderr`-Feld, der entsprechend gesetzt wird, falls ein abstrakter Befehl fehlschlägt oder der Debugger einen neuen abstrakten Befehl ausführen möchte, obwohl die Ausführung des letzten noch nicht beendet ist. Damit der Debugger keine Fehler übersieht kann das `cmderr`-Feld nur von ihm gelöscht werden.

Das COMMAND-Register speichert den abstrakten Befehl. Der Debugger schreibt dieses Register nachdem die nötigen Argumente zuvor in die DATA[0..11]-Register geschrieben wurden. Das schreiben dieses Registers löst die automatisch Ausführung des abstrakten Befehls aus.

Das ABSTRACTAUTO-Register speichert, ob der letzte abstrakte Befehl automatisch nochmal ausgeführt werden soll, nachdem auf ein entsprechendes DATA[0..11]-Register durch den Debugger zugegriffen wurde. Dies steigert die Effizienz bei Blockzugriffen, da die Adresse und die Daten in den DATA[0..11]-Registern gespeichert werden, muss für jeden Zugriff nicht derselbe abstrakte Befehl in das COMMAND-Register übertragen werden, sondern nur die Adresse oder die Daten in die DATA[0..11]-Registern geladen werden.

Abstrakte Befehle

Damit der Debugger auf GPRs, CSRs und Speicher zugreifen kann werden folgende Mechansmen vorgestellt: Systembuszugriff, Program Buffer und abstrakte Befehle. Abstrakte Befehle sind Befehle die der Debugger an das DM übermittelt um auf Speicher, GPRs oder CSRs zuzugreifen. Die Befehle werden dabei in abstrakter Form an das DM übermittelt (z.B. x8-GPR Lesen, DCSR-CSR Schreiben). Das DM ist verantwortlich wie diese Ausgeführt werden. In dieser Arbeit werden die abstrakten Befehle verwendet, da sie einen guten Kompromiss zwischen Hardwareverbrauch und Performance, im Vergleich zum Systembus und Program Buffer, bieten. Dabei wird die Operation in abstrakter Form in das COMMAND-Register geschrieben. Wie die Operation ausgeführt wird, ist von der Spezifikation nicht vorgegeben. Argumente für die abstrakten Befehle werden vom Debugger vor dem Beschreiben des

COMMAND-Registers in die DATA[0..15]-Register geladen. In Tabelle 2.7 sind die möglichen Zugriffsarten durch abstrakte Befehle aufgelistet.

Zugriff auf	Lesen	Schreiben
Speicher	Ja	Ja
GPR	Ja	Ja
CSR	Ja	Ja

Tabelle 2.7: Zugriffsarten von abstrakte Befehlen

Für den Registerzugriff lädt der Debugger die Registernummer und die Zugriffsart in das COMMAND-Register. Bei einem Schreibzugriff werden die zu schreibenden Daten zuerst in das DATA0-Register geladen. Bei einem Lesezugriff wird der Registerinhalt in das DATA0-Register geladen und kann, nach der Ausführung des abstrakten Befehls, dort vom Debugger ausgelesen werden.

Für den Speicherzugriff lädt der Debugger zuerst die Speicheradresse in das DATA1-Register. Bei einem Schreibzugriff werden die zu schreibenden Daten in das DATA0-Register geladen. Anschließend lädt der Debugger die Zugriffsart in das COMMAND-Register. Nach der Ausführung wird bei einem Lesezugriff der Speicherinhalt in das DATA0-Register geladen und kann von dort vom Debugger ausgelesen werden. Für Blockzugriffe kann im COMMAND-Register das `aampostincrement`-Bit gesetzt werden. Nach der Ausführung eines abstrakten Befehls wird die Adresse im DATA1-Register um eins inkrementiert. Somit muss der Debugger für das nächste Speicherfeld die Adresse nicht übertragen.

Einzelschrittausführung

Um die Einzelschrittausführung zu aktivieren muss der Prozessor als erstes in den Debug-Modus überführt werden. Als nächstes setzt der Debugger im DCSR-Register das `step`-Bit. Jetzt setzt der Debugger das `resumereq`-Bit und das Hauptprogramm wird fortgesetzt. Der Prozessor muss sich nach dem Ausführen eines Befehls von alleine in den Debug-Modus überführen.

Asynchroner Halt

Wie der asynchrone Halt implementiert werden soll, ist nicht genauer spezifiziert. Es muss dafür gesorgt werden, dass bei Setzen `haltreq`-Bits durch den Debugger der

Prozessor, nach Beendigung des aktuellen Befehls, in den Debug-Modus überführt wird.

3 Vorausgegange Arbeiten

3.1 Soft-Debug-Infrastruktur des PicoNut-Systems

Das PicoNut-System verfügt bereits über eine Debug-Infrastruktur, die auch auf der *RISC-V Debug Specification* [9] basiert. Diese wurde als Soft-Hardware-Modul implementiert, ist also nicht synthetisierbar und ausschließlich simulierbar. Damit können RISC-V-kompilierte Programme, die auf dem PicoNut simuliert ausgeführt werden, debuggt werden.

Die Soft-Debug-Infrastruktur unterstützt folgende Funktionen:

- **Programmflusskontrolle:** Das Programm kann asynchron angehalten und fortgesetzt werden. Zudem werden Breakpoints und die Einzelschritt-Ausführung unterstützt.
- **Registerzugriff:** Lese- und Schreibzugriffe auf die GPRs sowie auf die CSRs sind möglich.
- **Speicherzugriff:** Sowohl lesende als auch schreibende Zugriffe auf den Speicher werden unterstützt.

Da die Infrastruktur ausschließlich simulierbar ist, ist das Debugging auf FPGA-Hardware nicht möglich. Das hat einige Nachteile wie, dass Fehler in Programmen die nur auf dem FPGA auftreten nicht gefunden werden können. Außerdem ist das Debugging von Programmen, die auf externe Hardware zugreifen, für die kein Emulator existiert, nicht durchgeführt werden.

Abgesehen von der effizienten Speicherbeschreibung werden alle Debug-Funktionen bereitgestellt, die auch von der in dieser Arbeit entwickelten Debug-Infrastruktur unterstützt werden [6].

3.2 Debug-Infrastruktur des ParaNut-Prozessor

Der ParaNut-Prozessor ist ein an der Technischen Hochschule Augsburg von Studierenden entwickelter Prozessor mit RISC-V-Architektur. Er verfügt über eine synthetisierbare Debug-Infrastruktur, die auf der *RISC-V Debug Specification* [9] basiert. Obwohl der ParaNut als Parallelprozessor konzipiert wurde, konnte in der bisherigen Implementierung der Debug-Infrastruktur nur ein Kern aktiv zu debuggen werden. In dieser Hinsicht zeigt das Projekt eine starke Ähnlichkeit zum PicoNut, da auch hier die Debugging-Funktionalität auf einen einzelnen Kern beschränkt ist. [2]

3.3 CVA6 RISC-V CPU

Der CVA6 ist ein konfigurierbarer 6-Phasen Pipeline RISC-V Prozessor welcher RV64 und RV32 implementiert. Des weiteren Unterstützt er die I, M, A und C RISC-V Erweiterungen. Das Projekt wird von der OpenHW Community entwickelt und gepflegt. Als Modellierungssprache wird SystemVerilog verwendet [10]. Das PULP-Projekt hat unteranderem für diesen Prozessor eine synthetisierbare Debug-Infrastruktur entwickelt. Sie ist konform zur *RISC-V Debug Specification* [9] Version 0.13.1. Als Debug-Hardwareschnittstelle wird JTAG verwendet. [11]

4 Anforderungen

Für den PicoNut soll eine Debug-Infrastruktur entwickelt werden, die vollständig konform zur *RISC-V Debug Specification* [9] in der Version 1.0 ist. Ziel ist es, Debugging direkt auf der FPGA-Hardware mithilfe von OpenOCD und GDB zu ermöglichen. Darüber hinaus soll die Infrastruktur das Laden und Ausführen von Programmen in den Hauptspeicher erlauben, ohne dass der PicoNut neu synthetisiert werden muss. Das Laden von Programmen in den Hauptspeicher soll dazu im Vergleich zu einer Resynthese schneller ablaufen. Damit soll die Softwareentwicklung auf dem PicoNut mit FPGA-Hardware genauso effizient wie mit konventionellen Microcontrollern möglich sein.

Als Hardwareschnittstelle soll die Debug-Infrastruktur JTAG bieten. Als Zugriffsmechnismus auf GPRs, CSRs und Speicher sollen die entsprechende abstrakten Befehle unterstützt werden. Um Debugging zu ermöglichen muss die Debug-Infrastruktur außerdem folgenden Funktionalitäten bereitstellen:

- Anhalten und Fortsetzen der Programmausführung
- Setzen von Breakpoints
- Einzelschrittausführung
- Lesen und Schreiben der GPRs
- Lesen und Schreiben der CSRs
- Lesen und effizientes Schreiben des Speichers

Der Prozessor muss hierfür folgende Anforderungen erfüllen:

- Debug CSRs Implementieren: DCSR, DPC, DSCRATCH[0..1].
- Ist ein externes Hardwaresignal gesetzt, den Programmzähler ändern.
- Beim Dekodieren des `EBREAK`-Befehls den Programmzähler ändern.
- Beim Ausführen eines Befehls den Programmzähler ändern, falls das `step`-Bit des DSCR-CSRs gesetzt ist.
- Beim Dekodieren des `dret`-Befehls, den Programmzähler ändern.

5 Implementierung

5.1 Übersicht

In diesem Kapitel wird die in dieser Arbeit entwickelte Implementierung der Debug-Infrastruktur erläutert. Die Debug-Infrastruktur für einen RISC-V-Prozessor besteht gemäß der *RISC-V Debug Specification* [9] in Version 1.0 aus drei Hauptkomponenten: einer Debug-Peripherie, die das Debugging sowie die Kommunikation mit dem Debugger steuert, einem neuen Prozessor Modus, den Debug-Modus, indem sich der Prozessor während des Debugging befindet, und einer Erweiterung des Prozessors

einschließlich zusätzlicher CSRs. Für die Synthese wurden folgende Tools verwendet: *ICSC* Version 1.6.36, *Yosys* Version 0.57 und nextpnr-ecp5 Version 0.9.

Zunächst wird in Kapitel 5.4 auf den Debug-Modus eingegangen. Dieser stellt einen speziellen Prozessormodus dar, in dem sich der Prozessor befindet, wenn er durch einen Debugger angehalten wurde.

Anschließend wird in Kapitel 5.2 beschrieben, welche Erweiterungen am Prozessor erforderlich waren, um Debugging zu ermöglichen. Dazu zählt die Verarbeitung des Eintritts in den Debug-Modus durch den Prozessor, sowie die Implementierung der zusätzlichen CSRs.

Als Nächstes wird in Kapitel 5.3 die Implementierung der Debug-Peripherie beschrieben. Es werden die Submodule: DTM und DM behandelt, sowie die Verbindung über die DMI-Schnittstelle.

5.2 Erweiterung des Prozessors

Um das Debugging zu unterstützen, musste der Prozessor entsprechend erweitert werden. Die notwendigen Anpassungen beschränkten sich dabei ausschließlich auf den Nucleus. Konkret wurden das CSR-Modul, das Controller-Modul sowie das Programmzähler-Modul angepasst.

Die Schnittstelle zur erstellten Debug-Peripherie bestehen aus zwei Teilen. Zum einen aus zwei Signalen `debug_haltrequest` und `debug_haltrequest_ack`, zwischen der Debug-Perippherie und dem Nucleus, und zum anderen aus einer Sprungadresse, die auf die Debug-Handler Routine zeigt. Die Debug-Handler Routine, ein kleines Programm, das der Prozessor ausführt solange er sich im Debug-Modus befindet.

Damit der Prozessor in den Debug-Modus überführt werden kann, wurde das Controller-Modul so angepasst, dass die Ereignisse asynchrones Halten, Breakpoints und Einzelschrittausführung verarbeitet werden können. Für das asynchrone Halten setzt das DM das `debug_haltrequest`-Signal, welches in den Controller des Nucleus geleitet wird, wechselt der Controller in einen extra Zustand der den Prozessor in den Debug-Modus überführt. Anschließend wird das `debug_haltrequest_ack`-Signal gesetzt. Dieses teilt dem DM mit, dass der Prozessor erfolgreich in den Debug-Modus überführt wurde und das `debug_haltrequest`-Signal zurückgenommen werden kann. Der Controller verbleibt in einem Wartezustand bis das Signal zurückgenommen wurde. Bleibt das Signal gesetzt würde der Prozessor versuchen mehrfach den Debug-Modus zu betreten.

Da ein Breakpoint durch den `EBREAK`-Befehl ausgelöst wird, wurde hierfür ein eigener Zustand eingeführt, der den Eintritt in den Debug-Modus aufgrund des Dekodierens des `EBREAK`-Befehls auslöst. Ist die Einzelschrittausführung aktiviert, also das `step`-Bit im DCSR-CSR gesetzt, so wird nach Abschluss der Ausführung eines Befehls unmittelbar in einen Zustand gewechselt, der den erneuten Eintritt in den Debug-Modus auslöst. Die Zustände für das Eintreten in den Debug-Modus teilen dem CSR-Modul den Grund des Eintritts via jeweiligem Signal mit. Das CSR-Modul speichert den Grund im DCSR-CSR ab. Dort kann er vom Debugger ausgelesen werden.

Zum Verlassen des Debug-Modus dient der `dret`-Befehl. Beim Dekodieren dieses Befehls wird in einem neuen Zustand dem Programmzähler-Modul mitgeteilt, den im DPC-CSR gespeicherten Rücksprungwert zu laden, sodass die Ausführung des Hauptprogramms fortgesetzt wird.

Das CSR-Modul wurde um die Debug-spezifischen Control and Status Register gemäß Kapitel 2.3.3 erweitert. Alle hinzugefügten CSRs sind über die standardisierten RISC-V-CSR-Befehle zugreifbar. Beim Eintritt in den Debug-Modus wird der aktuelle Wert des Programmzählers im DPC-CSR gespeichert. Das DCSR-CSR wird beim Reset auf definierte Standardwerte gesetzt. Dazu gehört unter anderem das Privileg-Level, welches auf Machine-Mode initialisiert wird, sowie die Versionsangabe der Debug-Implementierung, hier 1.0. Zusätzlich wird der Grund für das Betreten des Debug-Modus entsprechend des auslösenden Ereignisses gesetzt.

Das Programmzähler-Modul wurde so erweitert, dass beim Eintritt in den Debug-Modus der Programmzähler auf die Startadresse der Debug-Handler Routine gesetzt wird. Beim Verlassen des Debug-Modus wird der Programmzähler auf den im DPC-CSR gespeicherten Wert gesetzt. Das jeweilige Überschreiben des Programmzählers wird dabei durch entsprechende Steuersignale aus dem Controller-Modul ausgelöst.

5.3 Debug-Peripherie

5.3.1 Übersicht

In diesem Kapitel ist die Implementierung der Debug-Peripherie beschrieben. Die Implementierung orientiert sich an der Spezifikation aus Kapitel 2.3.4.1. Der grundsätzliche Aufbau der Debug-Infrastruktur im PicoNut ist in Abbildung 5.1 dargestellt.

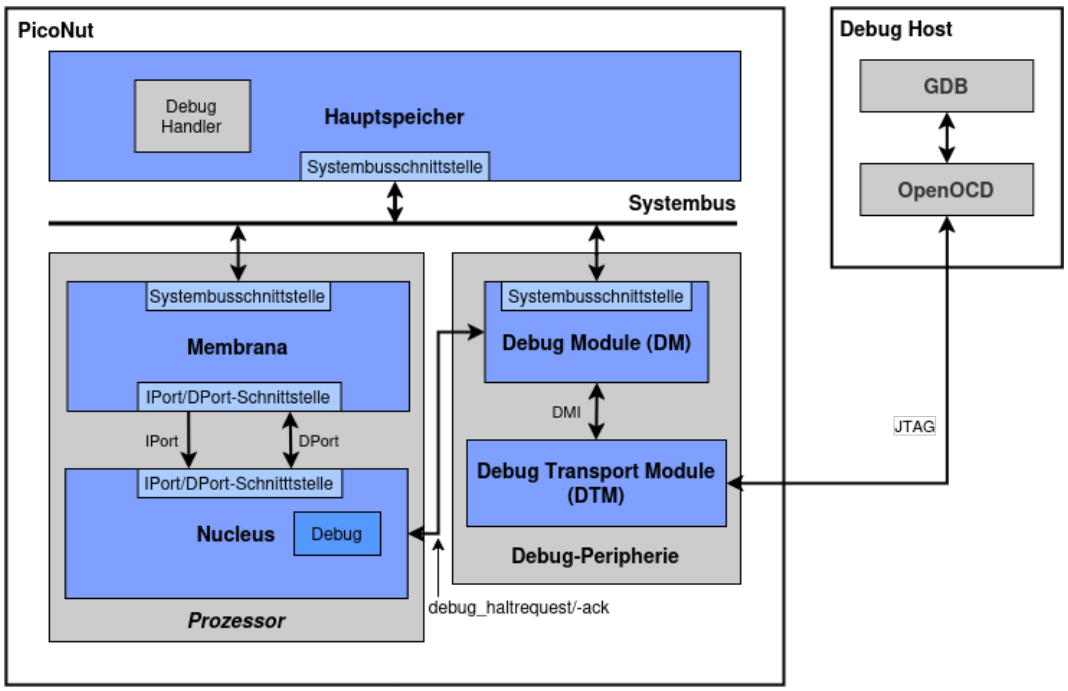

Abbildung 5.1: Übersicht der Debug-Infrastruktur im PicoNut

Die Spezifizierung der DMI-Schnittstelle, welche die Kommunikation zwischen Debug Transport Module (DTM) und Debug Module (DM) ist in Kapitel 5.3.2 beschrieben.

Die Implementierung des DTMs mit JTAG-Schnittstelle ist in Kapitel 5.3.4 beschrieben. Die JTAG-Signale werden vor der Verwendung im DTM mit dem Systemtakt synchronisiert und aufbereitet. Dieser Prozess ist Kapitel 5.3.3 beschrieben.

Das Herzstück der Debug-Peripherie bildet das Debug Module (DM). Dessen Implementierung ist in Kapitel 5.3.5 detailliert dargestellt.

5.3.2 Debug Module Interface (DMI)

Das Debug Module Interface (DMI) ist die Schnittstelle zwischen DTM und DM. Es handelt sich um eine Master-Slave-Schnittstelle, wobei das DTM dem Master entspricht und entsprechend das DM dem Slave. Alle Lese- und Schreibzugriffe auf Register des DMs werden vom DTM gesteuert. Es wurde darauf geachtet, die Schnittstelle möglichst einfach und hardwaresparend zu gestalten. Die Signale der Schnittstelle sind in Tabelle 5.1 aufgelistet.

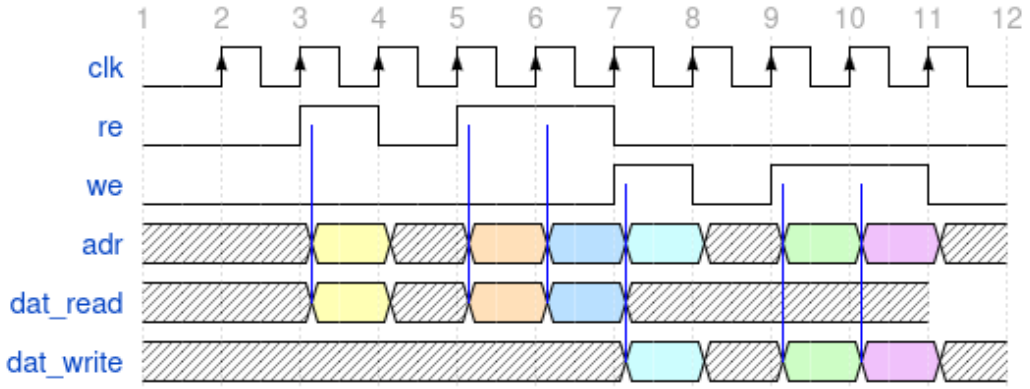

Abbildung 5.2: Signalverlauf der DMI-Schnittstelle

Name	Bits	Beschreibung
`re`	1	Read-Enable: 1, wenn Daten gelesen werden sollen, sonst 0.
`we`	1	Write-Enable: 1, wenn Daten geschrieben werden sollen, sonst 0.
`adr`	6	Adresse des Registers, auf welches zugegriffen werden soll.
`dat_read`	32	Daten, die gelesen werden sollen.
`dat_write`	32	Daten, die geschrieben werden sollen.

Tabelle 5.1: Signale der DMI-Schnittstelle

Die Datenleitungen haben wie die Register im DM eine Breite von 32 Bit. Somit kann in einem Takt ein komplettes Register gelesen bzw. geschrieben werden. Um die Schnittstelle einfach zu halten wurde auf ein Acknowledge-Signal verzichtet und es wird davon ausgegangen, dass alle Zugriffe erfolgreich sind.

Der Signalverlauf der Lese- und Schreibzugriffe ist in Abbildung 5.2 dargestellt. Für einen Lesezugriff legt der Master die Adresse des Zielregisters an das `adr`-Signal an und das `re`-Signal wird gesetzt. Im gleichen Takt legt der Slave die Daten an `dat_read` an. Für einen Schreibzugriff legt der Master ebenfalls die Adresse des Zielregisters an das `adr`-Signal an, sowie die Daten an das `dat_write`-Signal an und setzt das `we`-Signal. Der Slave muss jetzt die Daten in das Zielregister übernehmen.

5.3.3 Synchronisierung der JTAG-Eingangssignale

Die JTAG-Signale `tck`, `tms` und `tdi` werden vom Debug-Adapter erzeugt. Dieser hat eine eigene Taktdomäne, welche vom Systemtakt des FPGAs und somit des DTMs abweicht. Der Systemtakt wird durch eine externe Clock auf der Platine des ULX3S-FPGA-Boards erzeugt und beträgt `25 MHz`.

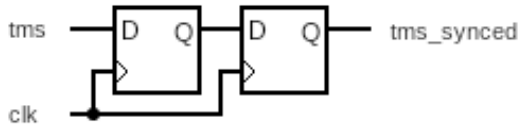

Abbildung 5.3: Synchronisierung mit Flip-Flop-Kette

Das DTM muss diese Signale verarbeiten, dabei gibt es beim Übergang zwischen diesen beiden Taktdomänen das Problem, dass die Eingangssignale nicht mit dem Systemtakt abgestimmt sind. Änderungen an den JTAG-Signalen können genau in dem Moment auftreten, in dem das DTM sie einliest. Dadurch besteht die Gefahr von Metastabilität in den Flip-Flops, was zu unvorhersehbarem Verhalten führen kann.

Um dieses Problem zu vermeiden, müssen die Eingangssignale zunächst mit dem Systemtakt synchronisiert werden. Da der JTAG-Takt mit `5 MHz` im Vergleich zum Systemtakt langsamer ist, genügt hierfür eine einfache Kette aus zwei hintereinandergeschalteten Flip-Flops. Auf diese Weise werden die Signale stabil in die Systemtaktdomäne übernommen, ohne dass Informationen verloren gehen.

Ein implementierte Synchronisierungskette ist in Abbildung 5.3 dargestellt. Dabei ist links das Signal `tms` das Eingangssignal und rechts `tms_synced` das Ausgangssignal. Ein beispielhafter Signalverlauf ist für das `tms`-Signal in Abbildung 5.5 abgebildet. Das Signal `tdi` wird analog zu `tms` synchronisiert.

Damit die Signale im DTM korrekt verarbeitet werden, muss zudem sichergestellt sein, dass das JTAG-Taktsignal `tck` nur für einen einzigen Taktzyklus aktiv ist. Wäre das Signal über mehrere Takte hinweg gesetzt, könnten die Anliegenden Daten mehrfach ausgewertet werden. Dadurch könnte der TAP-Automat mehrere Zustandsübergänge ausführen, obwohl nur einer beabsichtigt ist oder es könnte mehrfach in ein Register geschrieben werden.

Die Synchronisierungskette sorgt zwar für die zeitliche Abstimmung, ändert aber wenig an der Dauer der Signalpegel. Daher wird hinter die Synchronisierungskette eine Flankenerkennung geschaltet. Sie erzeugt aus jedem Signalwechsel einen kurzen Impuls, der nur einen Taktzyklus lang aktiv ist. Dadurch kann das DTM mit Hilfe des Impulsartigen `tck`-Signals die Daten eindeutig verarbeiten.

Ein beispielhafter Signalverlauf für das `tck`-Signal ist in Abbildung 5.5 dargestellt. Die implementierte Synchronisierungskette mit Flankenerkennung für das `tck`-Signal ist in Abbildung 5.5 gezeigt. Das resultierende Impuls Signal heißt `tck_synced_edge`.

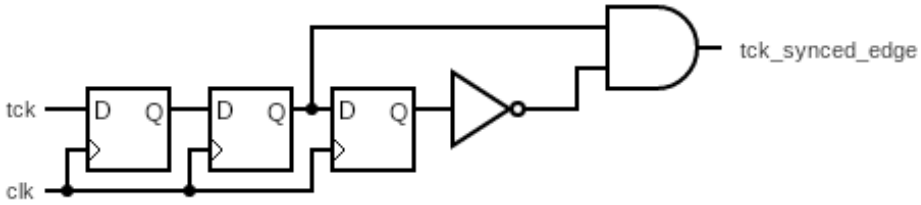

Abbildung 5.4: Synchronisierung mit Flankenerkennung mit Flip-Flop-Kette

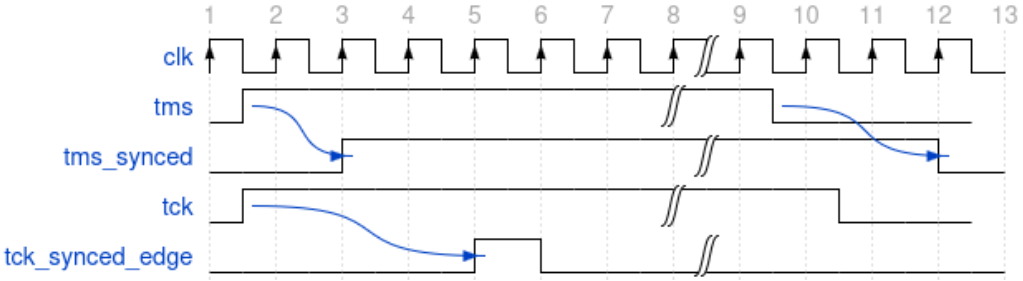

Abbildung 5.5: Signalverlauf der JTAG-Signalsynchronisation

5.3.4 Debug Transport Module (DTM)

Das DTM bildet die Schnittstelle zwischen der externen Debug-Transport-Hardware und dem internen Debug Module (DM) und setzt hierfür eine JTAG-basierte Anbindung gemäß der *RISC-V Debug Specification* [9] um.

Die Eingangssignale der JTAG-Schnittstelle werden vom externen JTAG-Adapter in einer anderen Takt-Domäne erzeugt als der Systemtakt, mit dem das DTM betrieben wird. Um einen zuverlässigen Betrieb zu gewährleisten, müssen diese Signale vor ihrer Weiterverarbeitung zunächst in die Systemtakt-Domäne synchronisiert werden. Die hierfür eingesetzte Synchronisationslogik wird in Kapitel 5.3.3 detailliert beschrieben. In diesem Kapitel wird vorausgesetzt, dass die Signale synchronisiert vorliegen.

Das implementierte DTM besteht im Wesentlichen aus einem Zustandsautomaten zur Realisierung des JTAG Test Access Ports (TAP) sowie den dazugehörigen Registern. Hierzu zählen sowohl die standardisierten TAP-Register gemäß Kapitel 2.2.3.3 als auch die zusätzlichen Register für das RISC-V-Debugging, die in Kapitel 2.3.4.2 beschrieben wurden. Darüber hinaus enthält das DTM eine Steuerlogik zur Anbindung der DMI-Schnittstelle an das Debug Module. Der Aufbau des DTMs ist in Abbildung 5.6 dargestellt.

Der TAP-Automat wurde als Moore-Zustandsmaschine realisiert. Zustandsübergänge erfolgen ausschließlich bei einem aktiven `tck`-Signal, welches hi-

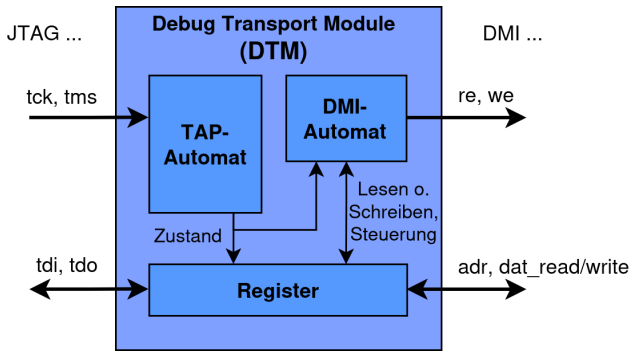

Abbildung 5.6: Aufbau des implementierten DTMs

erbei als Enable-Signal für den Zustandswechsel dient. Die implementierten Zustände und Übergänge entsprechen dem in Kapitel 2.2.3.4 beschriebenen TAP-Zustandsdiagramm gemäß IEEE 1149.1.

Sämtliche beschriebenen Datenregister sowie das Instruction-Register wurden vollständig implementiert. Der serielle Zugriff über die Signale `tdi` und `tdo` erfolgt über entsprechende Multiplexer, die abhängig vom aktuellen Zustand des TAP-Automaten sowie vom Inhalt des Instruction-Registers das jeweils aktive Register auswählen.

Für die Ansteuerung der DMI-Schnittstelle wurde ein separater, einfacher Zustandsautomat implementiert. Dieser führt Lese- und Schreiboperationen auf der DMI-Schnittstelle im Zustand *Capture-DR* des TAP-Automaten aus, sofern im Instruction-Register das DMI-Register ausgewählt ist. Wie in Kapitel 2.3.4.2 sind sämtliche Informationen im DMI-Register gespeichrt. Dieses wird vom Automaten ausgelesen, ob gelesen bzw. geschrieben wird. Die Adresse und bei einem Schreibzugriff die zu schreibenden Daten werden vom Automaten entsprechend gesetzt. Bei einem Lesezugriff wird über ein Steuersignal das DMI-Register zur richtigen Zeit Informiert, die Daten an der DMI-Schnittstelle zu übernehmen. Nach einem Schreibvorgang wird das DMI-Register gelöscht. Das zeigt dem Debugger, dass die Daten verarbeitet wurden.

Da in dieser Implementierung davon ausgegangen wird, dass sämtliche DMI-Zugriffe innerhalb eines Takts abgeschlossen werden, kann auf eine Mehrzyklische-Handshake-Logik verzichtet werden. Entsprechend konnten Lese- und Schreibzugriffe vollständig innerhalb eines einzelnen TAP-Zustands ausgeführt.

5.3.5 Debug Module (DM)

In diesem Kapitel wird die Implementierung des Debug Modules (DM) beschrieben. Das DM bildet das Herzstück der Debug-Peripherie. Es beinhaltet die Logik um die Debug-Operationen, wie Anhalten, Fortsetzen oder einen Speicherzugriff, in die konkrete Ausführung zu bringen. In Abbildung 5.7 ist der komplette Aufbau des implementierten DMs zu sehen.

Das DM hat insgesamt drei Ports, die in Abbildung 5.7 je oben angeordnet sind: den DMI-Slave-Port, Wishbone-Slave-Port, sowie den Debug-Port. Die Ports bilden die Schnittstelle nach außen und werden in Kapitel 5.3.5.1 behandelt.

Es werden eine Reihe von Registern vom DM implementiert. Dazu zählen die schon beschriebenen Register aus Kapitel 2.3.4.3, auf die der Debugger zugreift. Sowie neu eingeführte Register mit Systembusanbindung, auf die der Prozessor Zugriff hat. In Abbildung 5.7 sind die Register mittig vertikal untereinander angeordnet. Die Implementierung der Register ist in Kapitel 5.3.5.2 beschrieben.

Um abstrakte Befehle aus Kapitel 2.3.4.3 ausführen zu können werden sie in dieser Implementierung in RISC-V-Assembler-Befehle Befehle übersetzt und anschließend vom Prozessor ausgeführt. Die Übersetzung der Befehle wird in Kapitel 5.3.5.3 behandelt.

Das Steuerwerk des DMs ist der Controller. Dieser reagiert auf Änderungen in den Registern durch Debugger und Prozessor und steuert entsprechend die internen Zugriffe auf die Register. Der Controller ist in Abbildung 5.7 unten angeordnet. Die Funktionsweise wird in Kapitel 5.3.5.3 beschrieben.

5.3.5.1 Ports

Der DMI-Slave-Port bildet den Zugang zur DMI-Schnittstelle und somit zum DTM. Darüber sind die Register aus Kapitel 2.3.4.3 erreichbar. Der Debugger kann somit auf die Register zugreifen.

Über den Wishbone-Slave-Port kann der Prozessor im Debug-Modus auf die neu eingeführten Register HARTCONTROL, HARTSTATUS, ABSTRACT[0..7] und die schon beschriebenen Register DATA[0..1] zugreifen.

Der Debug-Port besteht aus zwei Signalen: `debug_haltrequest` und `debug_haltrequest_ack`. Das erste wird direkt in den Controller des Prozessors geleitet. Wird es durch das DM gesetzt löst dort ein anhalten, also ein Überführen in den Debug-Modus, des Prozessors aus. Der Controller meldet die erfolgreiche Überführung durch

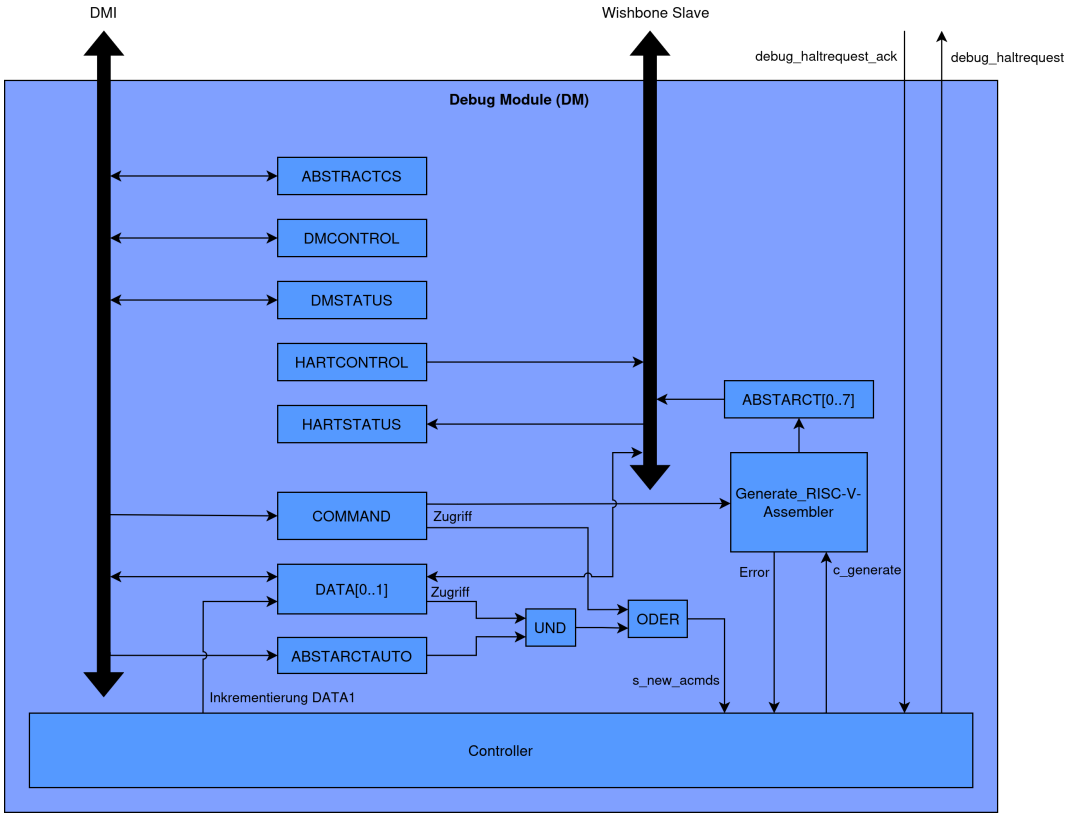

Abbildung 5.7: Aufbau des implementierten DMs

setzen des zweiten Signals. Über diesen Mechanismus kann der Prozessor asynchron vom Debugger angehalten werden. Wie das Anhalten im Prozessor funktioniert ist in Kapitel 5.2 beschrieben.

5.3.5.2 Register

Das Debug Module (DM) implementiert die in der Spezifikation beschriebenen Register gemäß Kapitel 2.3.4.3. Diese Register sind über die DMI-Schnittstelle erreichbar. Zusätzlich werden eigene Register zur Steuerung und Statusabfrage des Prozessors im Debug-Modus eingeführt, die über den Systembus zugänglich sind. Darüber hinaus existieren Speicherregister für generierte RISC-V-Assembler-Befehle, die im Debug-Modus ausgeführt werden können. Eine detaillierte Beschreibung der generierten RISC-V-Assembler-Befehle ist in Kapitel 5.3.5.3 enthalten. Die Register sind in Abbildung 5.7 mittig und vertikal angeordnet dargestellt.

Die Register an der DMI-Schnittstelle gemäß Kapitel 2.3.4.3 wurden wie beschrieben implementiert. Zur Reduzierung des Hardwareaufwands wurden jedoch nur zwei der DATA[0..11]-Register realisiert. Für die aktuell implementierte Debug-Infrastruktur ist diese Anzahl an DATA-Registern ausreichend. Die DATA-Register sind zudem auch über den Systembus dem Prozessor zugänglich.

Die Register mit Systembusanbindung sind in Tabelle 5.2 aufgelistet. Alle aufgelisteten Register sind 32 Bit breit.

Name	Speicheradresse	Beschreibung
DATA[0..1]	0x00 - 0x04	Speicherregister für Daten. Siehe Kapitel 2.3.4.3.
ABSTARCT[0..7]	0x14 - 0x30	Speicherregister für generierte Assembler-Befehle.
HARTCONTROL	0x34	Kontrollregister für den Prozessor im Debug-Modus.
HARTSTATUS	0x38	Statusregister für den Prozessor im Debug-Modus.

Tabelle 5.2: Register mit Systembusanbindung im DM

Das HARTCONTROL-Register dient zur Steuerung des Prozessors im Debug-Modus. Der Aufbau des HARTCONTROL-Registers ist in Abbildung 5.8 mit den einzelnen Feldern gezeigt.

- `resume_req`: Gibt an, ob das Hauptprogramm fortgesetzt werden soll.

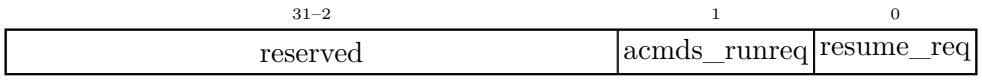

Abbildung 5.8: Aufbau des HARTCONTROL-Registers

- `acmds_runreq`: Gibt an, ob die generierten RISC-V-Assembler-Befehle ausgeführt werden sollen.

Das HARTSTATUS-Register dient als Statusregister des Prozessors. Der Aufbau des HARTSTATUS-Registers ist in Abbildung 5.9 mit den einzelnen Feldern gezeigt.

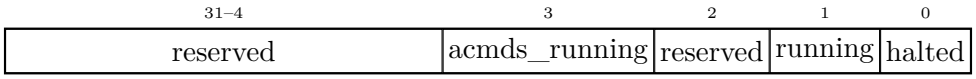

Abbildung 5.9: Aufbau des HARTSTATUS-Registers

- `halted`: Gibt an, ob der Prozessor im Debug-Modus ist.
- `running`: Gibt an, ob der Prozessor das Hauptprogramm ausführt.
- `acmds_running`: Gibt an, ob der Prozessor die generierten RISC-V-Assembler-Befehle ausführt.

Die Register ABSTRACT[0..7] sind Speicherregister für die generierten RISC-V-Assembler-Befehle. Was die generierten RISC-V-Assembler-Befehle sind wird in Kapitel 5.3.5.3 beschrieben.

5.3.5.3 Abstrakte Befehle

In Kapitel 2.3.4.3 wird das Konzept der abstrakten Befehle eingeführt. Es gibt zwei verschiedene Operationen von abstrakten Befehlen: den Registerzugriff und den Speicherzugriff. Wie die Zugriffe realisiert werden wird nicht spezifiziert.

Um abstrakte Befehle auszuführen, wurde in dieser Implementierung das Verfahren gewählt den abstrakten Befehl, welcher vom Debugger in das COMMAND-Register geladen wird, in mehrere RISC-V-Assembler-Befehle zu übersetzen. Der Prozessor führt anschließend die generierten Befehle aus, die in den Registern ABSTRACT[0..7] abgelegt sind. Die Erzeugung von RISC-V-Befehlen spart Chipfläche und Komplexität, da nicht für jede Art eines abstrakten Befehls die Logik im gesamten System erstellt werden muss, sondern auf vorhanden Zugriffsmechanismen durch den Prozessor zurückgegriffen wird. Lediglich die Logik für die Erzeugung der RISC-V-Assembler-Befehle fällt an.

Damit RISC-V Befehle generiert werden muss eine der folgenden Bedingungen erfüllt werden:

- Schreiben in des COMMAND-Register vom Debugger.
- Schreiben in das DATA1-Register vom Debugger und im ABSTRACTAUTO-Register. ist das Bit für das DATA1-Register gesetzt.

In Abbildung 5.7 ist diese Logik durch das UND- und das ODER-Gatter unten mittig dargestellt. Aus dieser Logik folgt das `s_new_acmds`-Signal, welches in den Controller geleitet wird. Dieser entscheidet, ob die Befehle erzeugt werden und setzt des `c_generate`-Signal, welches als Enable-Signal für die Generierung dient. Die Funktionsweise des Controllers, sowie die Steuerung für Generierung der Befehle wird in Kapitel 5.3.5.4 behandelt.

Die Generierung der Befehle erfolgt in einem eigenen Submodul des DMs: dem Generate_RISC-V-Assembler-Modul. Es hat als Eingänge das COMMAND-Register und das Enable-Signal `c_generate`. Die Ausgänge sind die erzeugten Befehle in den ABSTRACT-Registern und ein Errorsignal, falls ein Fehler bei der Erzeugung auftritt. Ein Fehler entsteht beispielsweise durch einen fehlerhaften Wert der COMMAND-Registers. Falls ein Fehler auftritt wird das `cmderr`-Feld des ABSTRACTCS-Registers auf den Wert `2` (*Not Supported*) gesetzt. Die Erzeugung erfolgt rein kombinatorisch. Die Schritte der Kombinatorik sind in Abbildung 5.10 dargestellt.

Im ersten Schritt wird überprüft, ob das `c_generate`-Signal gestetzt ist und das COMMAND-Register verarbeitet werden soll. Im Falle, dass es gesetzt ist wird im zweiten Schritt überprüft, ob es sich bei dem abstrakten Befehl um einen Register- oder Speicherzugriff handelt. Falls keines der beiden Zutrifft wird das Errorsignal gesetzt und abgebrochen. Ansonsten wird im dritten Schritt überprüft, ob der abstrakte Befehl auf je 32 Bit zugreifen möchte. Da der PicoNut zum Zeitpunkt dieser Arbeit ein reines 32 Bit System ist. Werden andere Zugriffsfenster nicht unterstützt. Für den Speicherzugriffspfad werden im vierten Schritt die RISC-V-Assembler-Befehle nach Tabelle 5.3 erzeugt. Für den Registerpfad wird im vierten Schritt zuerst noch überprüft, ob auf ein CSR oder GPR zugegriffen werden soll. Im fünften Schritt werde die RISC-V-Assembler-Befehle für den entsprechenden Registerzugriff erzeugt. Die Befehle für den Registerzugriff sind in Tabelle 5.3 zu sehen. Als letztes wird ein `EBREAK`-Befehl angehängt. Dadurch wird nach der Ausführung der Befehle automatisch wieder in die Debug-Handler Routine gesprungen.

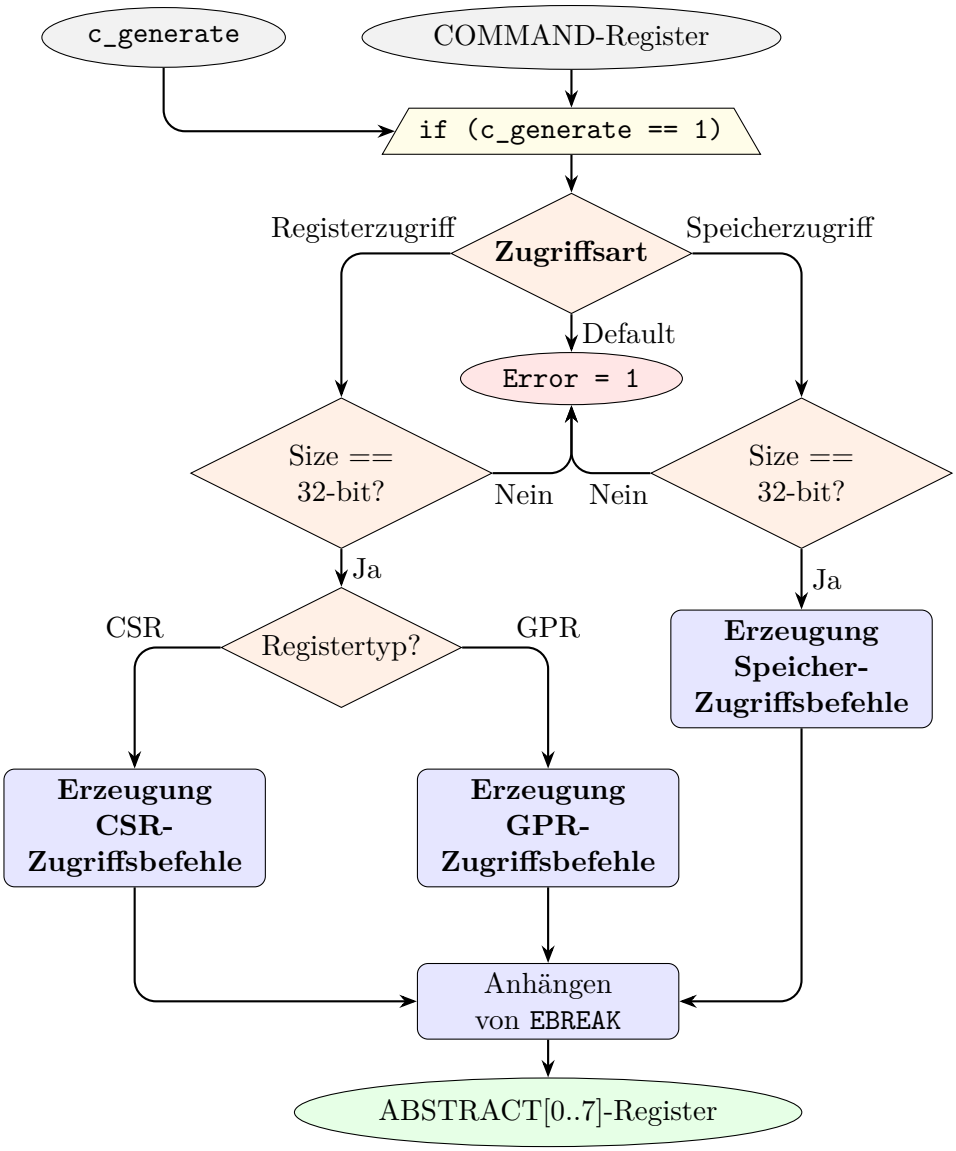

Abbildung 5.10: Flussdiagramm zur Erzeugung der RISC-V-Befehle

Zugriff auf	Schreiben	Lesen
Speicher	lw t5, DATA0(0) sw t5, 0(t6)	lw t6, 0(t6) sw t6, DATA0(0)
GPR	lw GPR-Num., DATA0(0)	sw GPR-Num., DATA0(0)
CSR	lw t6, DATA0(0) csrrw 0, CSR-Num., t6	csrrs t6, CSR-Num., 0 sw t6, DATA0(0)

Tabelle 5.3: Erzeugte RISC-V-Assembler-Befehle der Zugriffsarten

5.3.5.4 Controller

Der Controller bildet das Steuerwerk des DMs. Er ist als endlicher Zustandsautomat realisiert und besitzt gemäß Abbildung 5.11 insgesamt elf Zustände. Der Controller fungiert als Vermittlungsinstanz zwischen den Registern, auf die der Debugger zugreift, und den Registern, die vom Prozessor verwendet werden. Das DM beschreibt und verändert Register ausschließlich über den Controller. Dadurch wird sichergestellt, dass der Controller zu jedem Zeitpunkt den aktuellen Zustand des DMs sowie des Prozessors kennt. Der Controller greift in der Implementierung über Steuer- und Statussignale auf die Register zu. Zur Übersichtlichkeit werden, um die Funktionsweise des Controllers zu erklären, direkt die Bits der Register verwendet.

In Abbildung 5.11 ist das Zustandsdiagramm des Controllers gezeigt. Folgend werden die Zustände und Zustandsübergange beschrieben. Diese bilden die Funktionsweise des gesamten DMs ab. Es wird davon Ausgegangen, dass nach einem Systemreset das Hauptprogramm ausgeführt wird. Deshalb ist der Resetzustand der RUNNING-Zustand.

Im RUNNING-Zustand führt der Prozessor das Hauptprogramm aus. Es wird das `allrunning`-Bit des DMSTATUS-Registers gesetzt. Der Debugger sieht, dass der Prozessor das Hauptprogramm ausführt. Setzt der Debugger das `haltreq`-Bit des DMCONTROL-Registers, wird der Controller in den HALTREQ-Zustand überführt. Setzt der Prozessor das `halted`-Bit des HARTSTATUS-Registers, durch das Erreichen eines Breakpoints oder die Einzelschrittausführung, wird der Controller direkt in den HALTED-Zustand überführt.

Im HALTREQ-Zustand wird der Prozessor durch das Setzen des `debug_haltrequest`-Signals angefragt anzuhalten bzw. den Debug-Modus zu betreten. Ist der Prozessor angehalten, dann setzt er das `debug_haltrequest_ack`-Signal. Ist dieses oder das `halted`-Bit des HARTSTATUS-Registers gesetzt ist der Prozessor im Debug-Modus. Der Controller wird in den HALTED-Zustand überführt.

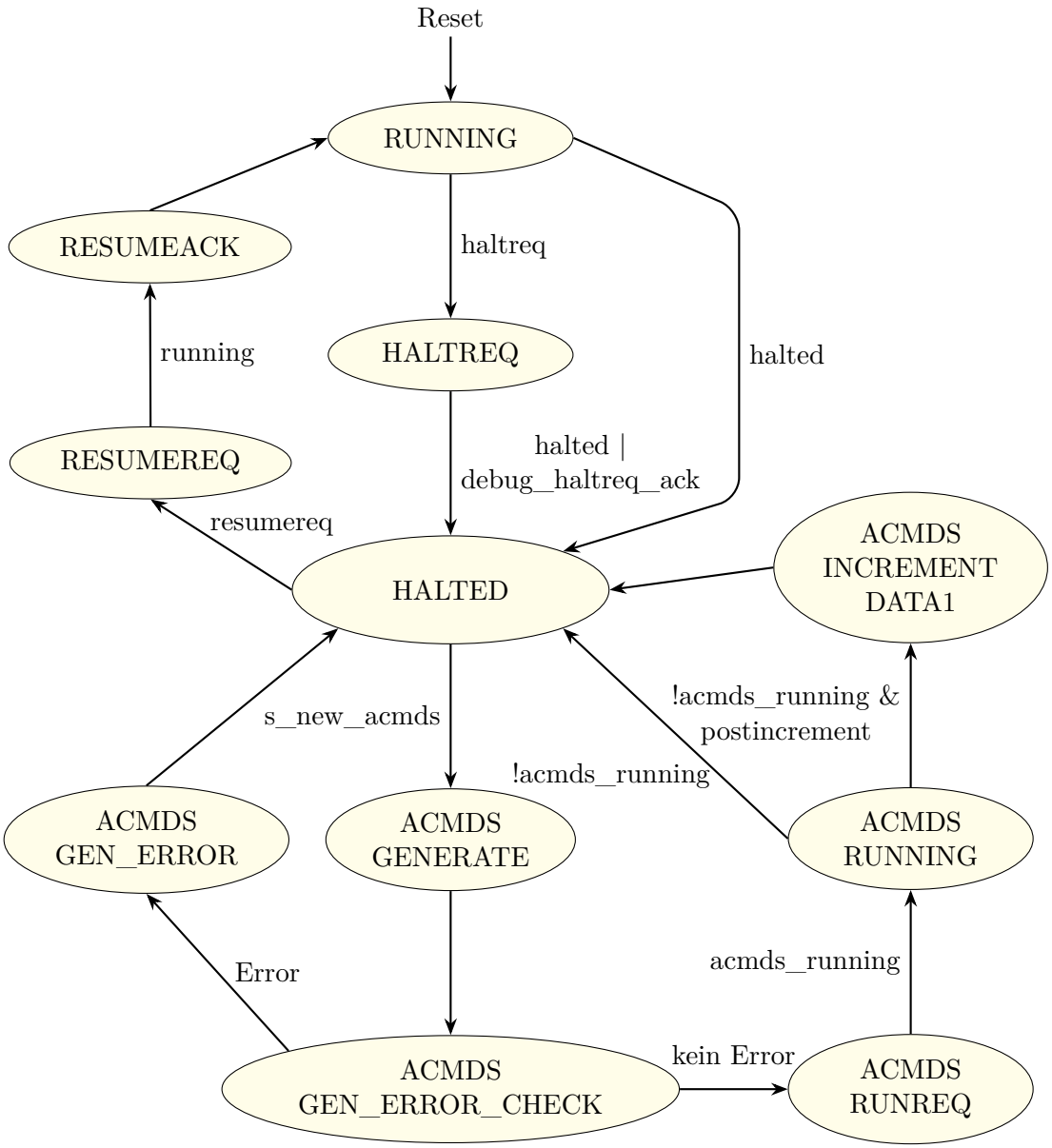

Abbildung 5.11: Zustandsdiagramm des Controllers im DM

Im HALTED-Zustand befindet sich der Prozessor im Debug-Modus. Das Hauptprogramm ist angehalten. Das `allhalted`-Bit des DMSTATUS-Registers wird gesetzt. Der Debugger sieht, dass der Prozessor im Debug-Modus ist. Setzt der Debugger das `resumereq`-Bit des DMCONTROL-Registers, dann wird der Controller in den RESUMEREQ-Zustand überführt. Ist das `s_new_acmds`-Signal gesetzt, das `acmds_running`-Bit des HARTSTATUS-Registers ist nicht gesetzt und das `cmderr`-Feld des ABSTRACTCS-Registers keinen Fehler hält wird der Controller in den ACMDS_GENERATE-Zustand überführt.

Im RESUMEREQ-Zustand wird der Prozessor angefragt den Debug-Modus zu verlassen und das Hauptprogramm fortzusetzen. Dazu wird das `resumereq`-Bit des HARTCONTROL-Registers gesetzt. Der Prozessor reagiert im Debug-Modus darauf und setzt das `running`-Bit des HARTCONTROL-Registers gesetzt hat, wird der Controller in den RESUMEACK-Zustand überführt.

Im RESUMEACK-Zustand hat der Prozessor bereits gemeldet, das das Hauptprogramm wieder fortgesetzt wurde. Dem Debugger wird über das Setzen des `allresumeack`-Bits des DMSTATUS-Registers gezeigt, dass das Hauptprogramm fortgesetzt wurde. Anschließend wird ohne Bedingung in den RUNNING-Zustand übergegangen.

Im ACMDS_GENERATE-Zustand werden die RISC-V-Assembler-Befehle aus dem COMMAND-Register nach Kapitel 5.3.5.3 generiert. Dafür wird das `c_generate`-Signal gesetzt um die Generierungskombinatorik zu starten. Der Controller wird ohne Bedingung in den ACMDS_GEN_ERROR_CHECK-Zustand überführt.

Im ACMDS_GEN_ERROR_CHECK-Zustand ist die Kombinatorik der Befehlserzeugung durchgelaufen. Es wird überprüft, ob es einen Fehler bei der Generierung gab. Falls das Errorsignal gesetzt ist wird der Controller in den ACMDS_GEN_ERROR-Zustand überführt. Ansonsten wird Controller in den ACMDS_RUNREQ-Zustands überführt.

Im ACMDS_GEN_ERROR-Zustand gab es bei der Generierung einen Fehler. Um dem Debugger das mitzuteilen, wird das `cmderr`-Feld des ABSTRACTCS-Registers auf den Wert `2` (*Not Supported*) gesetzt. Der Debugger sieht, dass der abstrakte Befehl von der Implementierung nicht unterstützt wird. Der Controller wird ohne Bedingung in den HALTED-Zustand überführt.

Im ACMDS_RUNREQ-Zustand war die Generierung erfolgreich, die Befehle liegen in den ABSTRACT[0..7]-Registern vor und können ausgeführt werden. Damit der Prozessor die Befehle ausführt, wird das `acmds_runreq`-Bit des HARTCONTROL-Registers gesetzt, der Prozessor im Debug-Modus führt daraufhin die generierten

Befehle aus. Er setzt davor das `acmds_running`-Bit des HARTCONTROL-Registers gesetzt hat, wird der Controller in den ACMDS_RUNNING-Zustand überführt.

Im ACMDS_RUNNING-Zustand führt der Prozessor die generierten RISC-V-Assembler-Befehle aus. Sobald er die Ausführung beendet hat, setzt der Prozessor das `acmds_running`-Bit zurück. Wenn das der letzte abstrakte Befehl ein Speicherzugriff war und das `aampostincrement`-Bit des COMMAND-Registers gesetzt ist. Wird der Controller in den ACMDS_INCREMENT_DATA1-Zustand überführt. Ist das `aampostincrement`-Bit nicht gesetzt wird er in den HALTED-Zustand überführt.

Im ACMDS_INCREMENT_DATA1-Zustand wird der Wert im DATA1-Register um ein Wort erhöht. Bei einem Blockzugriff auf den Speicher muss so die Adresse für den nächsten Speicherinhalt nicht übertragen werden. Anschließend wird der Controller in den HALTED-Zustand überführt.

5.4 Debug-Modus

Der Debug-Modus ist ein Prozessormodus, in welchem sich der Prozessor befindet, während Debugging aktiv ist. Wann der Debug-Modus aktiv ist wird in Kapitel 2.3.2 beschrieben.

Solange sich der Prozessor im Debug-Modus befindet führt er anstelle des Hauptprogramms die sogennante Debug-Handler Routine aus. Der Debug-Handler ist dabei im Hauptspeicher abgelegt und wird mit dem Hauptprogramm in den Speicher geladen. Die Routine wurde in der Datei *startup.S* abgelegt und liegt, nach dem Kompilieren eines RISC-V Programms für das PicoNut-Prozessor, entsprechend im Hauptspeicher vor.

Der Debug-Handler sorgt prinzipiell dafür, dass das Hauptprogramm angehalten bleibt, das Hauptprogramm fortgesetzt werden kann und abstrakte Befehle durch die generierten RISC-V-Assembler-Befehle ausgeführt werden können. Die generierten RISC-V-Assembler-Befehle werden in Kapitel 5.3.5.3 behandelt. In der Implementierung wird für abstrakte Befehle die Abkürzung acmds (abstract commands) verwendet.

Beim Betreten des Debug-Modus, wird als erstes der Programmzähler in das DPC-CSR übernommen. Im zweiten Schritt wird in den Debug-Handler gesprungen, indem der Programmzähler auf die erste Befehlsadresse der Routine gesetzt wird.

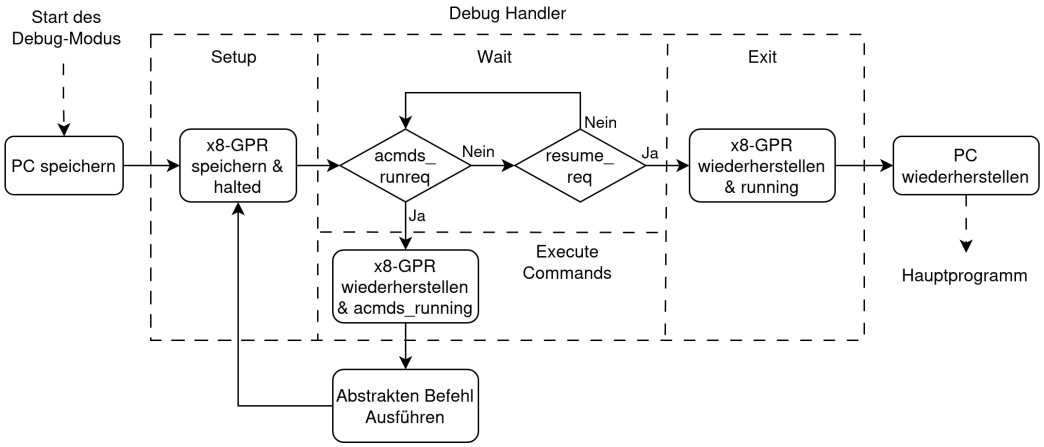

Abbildung 5.12: Ablauf im Debug-Modus und Debug-Handler

Die Routine hat vier Phasen. Das Ablaufdiagramm im Debug-Modus ist in Abbildung 5.12 zu sehen. Dabei passieren die Schritte in der gestrichelten Box im Debug-Handler.

Setup

- Abspeichern `x8`-GPRs in das DSCRATCH0-CSR, da dieses der Debug-Handler für die interne Funktionalität benötigt.
- Setzen des *halted*-Bits im HARTSTATUS-Register.

Wait

- Warten, bis das *resume_req*-Bit oder das *acmds_runreq*-Bit im *HARTCONTROL*-Register gesetzt ist.
- Ist das *resume_req*-Bit gesetzt, erfolgt ein Sprung in den Abschnitt *Exit*.
- Ist das *acmds_runreq*-Bit gesetzt, erfolgt ein Sprung in den Abschnitt *Execute Commands*.

Execute Commands

- Setzen des *acmds_running*-Bits im HARTSTATUS-Register.
- Wiederherstellen des `x8`-GPRs aus DSCRATCH0.
- Der Programmzähler wird auf die Adresse des ABSTRACT0-Registers gesetzt. Dieses Register enthält den ersten auszuführenden abstrakten Befehl, siehe dazu Kapitel 5.3.5.2.

Exit

- Setzen des *running*-Bits im HARTSTATUS-Register.
- Wiederherstellen des `x8`-GPRs aus DSCRATCH0.
- Verlassen des Debug-Modus mit der `dret`-Instruktion.

Der Debug-Handler wurde in RISC-V-Assembler geschrieben und umfasst 16 Instruktionen. Alle Befehle des Debug-Handlers sind in Anhang A.2 aufgelistet.

6 Ergebnisse

In diesem Kapitel werden die Ergebnisse dieser Arbeit dargestellt. Es wird behandelt, welche Funktionen erfolgreich implementiert wurden, sowie wie diese validiert wurden. Des Weiteren wird das Syntheseergebnis der Debug-Infrastruktur vorgestellt und analysiert. Zuletzt wird die erreichbare Ladegeschwindigkeit von Programmen über die Debug-Infrastruktur in den Hauptspeicher dargestellt und bewertet, ob diese Übertragungsrate für den praktischen Einsatz in der realen Entwicklung ausreichend ist.

6.1 Funktionen

Zur Validierung wurde die Debug-Kette mit einem JTAG-Adapter (Olimex ARM-USB-TINY-H), OpenOCD und GDB aufgebaut. Auf dem FPGA lief ein Referenzdesign inklusive Debug-Infrastruktur.

Das Referentzdesign besteht aus einem Nucleus, einer Membrana und einem UART-Peripheriemodul. Der Nucleus ist ein skalarer Kern mit In-Order-Ausführung und implementiert RV32IA. Die Membrana ist eine einfache Membrana mit integrierterm Block-RAM für die Software und Systembuszugriff. Der Block-RAM der Membrana ist 256 kB groß und besteht aus Dual-Port Block-RAM Zellen. Das UART-Peripheriemodul hat eine Sende-, sowie Empfangsfunktion. Beide Wege sind mit einem FIFO-Puffer mit je einer Tiefe von acht Wörten mit je acht Bit. Als Testprogramm diente das Programm aus Anhang A.3. Anschließend wurden die einzelnen Funktionen aus Kapitel 4 mit GDB geprüft und manuell erfolgreich validiert.

6.2 Syntheseergebnisse

In diesem Kapitel wird das Syntheseergebnis der Debug-Infrastruktur gezeigt. Es wird der Hardwareverbrauch des Referenzdesigns mit Debug-Infrastruktur mit einem Referenzdesign ohne Debug-Infrastruktur nach dem Platzieren und Verdrahten (Place-and-Route). Das Referenzdesign besteht aus dem Referenznucleus, einer einfachen Membrana und einem UART-Modul. Außerdem wird gezeigt wie viel Flip-Flops und Logikblöcke die Debug-Peripherie und die Erweiterung im Prozessor benötigen. Hier wird das Ergebnis der Netzliste gezeigt.

Um die SystemC-Hardwarebeschreibung des minimalen Nucleus zu synthetisieren, wurde diese zuerst mit *ICSC* in äquivalenten SystemVerilog-Code konvertiert. Die Synthese wurde mit *Yosys* und *nextpnr* für das Lattice ECP5-85k FPGA durchgeführt.

6.2.1 Vergleich des Referenzdesigns mit Debug-Infrastruktur mit dem Referenzdesign

Die Tabelle 6.1 zeigt eine Statistik der durch das Synthesetool gewählten FPGA-Zelltypen für das Referenzdesign. In Tabelle 6.2 ist die Statistik für das Referenzdesign mit Debug-Infrastruktur gezeigt. Die Werte beziehen sich auf die Ergebnisse nach dem Platzieren und Verdrahten. Es wurde folgender Commit verwendet: `f174215fed7b286e19b120133c3f6e0f19166229`

Zelltyp	Belegt	Verfügbar	Auslastung
TRELLIS_IO	4	365	1%
DP16KD	128	208	61%
MULT18X18D	0	156	0%
ALU54B	0	78	0%
TRELLIS_FF	1892	83640	2%
TRELLIS_COMB	8569	83640	10%
TRELLIS_RAMW	0	10 455	0%

Tabelle 6.1: Statistik des Referenzdesigns für das Lattice ECP5-85k FPGA

Zelltyp	Belegt	Verfügbar	Auslastung
TRELLIS_IO	4	365	1%
DP16KD	128	208	61%
MULT18X18D	0	156	0%
ALU54B	0	78	0%
TRELLIS_FF	2474	83640	2%
TRELLIS_COMB	11546	83640	13%
TRELLIS_RAMW	0	10 455	0%

Tabelle 6.2: Statistik des Referenzdesigns mit Debug-Infrastruktur für das Lattice ECP5-85k FPGA

Damit ergibt sich ein Overhead von 582 TRELLIS_FF und 2977 TRELLIS_COMB Zellen. Block-RAM-Zellen, hier DP16KD, werden in der Debug-Infrastruktur nur indirekt verwendet, da der Debug-Handler sich im Hauptspeicher befindet, hat sich die Anzahl nicht verändert.

Nun werden die Netzlisten der Debug-Peripherie und der Erweiterung des Prozessors analysiert. In Tabelle 6.3 ist die Statistik aus der Netzliste für die Debug-Peripherie dargestellt. In Tabelle 6.4 die Statistik für die Erweiterung des Prozessors.

Zelltyp	Anzahl
TRELLIS_FF	472
LUT4	2439

Tabelle 6.3: Statistik der Debug-Peripherie aus der Netzliste

Die Debug-Peripherie umfasst gemessen 472 Flip-Flops. Die theoretische Anzahl der Flip-Flops des DTMs ist 82. Die theoretische Anzahl der Flip-Flops des DMs beträgt 387. Insgesamt braucht die Debug-Peripherie theoretisch 469 Flip-Flops. Damit ist das gemessene Ergebnis plausibel.

Variante	TRELLIS_FF	LUT4
Mit Erweiterung	1 607	7 682
Ohne Erweiterung	1 497	7 323
Differenz	**+110**	**+359**

Tabelle 6.4: Statistik des Nucleus aus der Netzliste

Der Nucleus wächst gemessen um 110 Flip-Flops. Die neu Eingeführten Debug-CSRs, DCSR, DPC, DSCRATCH[0..1], sind jeweils 32 Bit breit. Damit ergibt sich

ein theoretischer Overhead von 128 Flip-Flops. Das DCSR-CSR hat allerdings nur zehn Bits die nicht konstant sind. Somit ergibt sich eine theoretischer Overhead von 106 Flip-Flops. Die gemessene Differenz von 110 Flip-Flops ist dementsprechend ein plausibles Ergebnis.

6.3 Laden von Programmen

Ein weiteres Ziel dieser Arbeit ist der Austausch von Programmen ohne erneute Resynthese des FPGA-Designs. Für die Messung wurde das Testprogramm aus Anhang A.3 mit der RISC-V GNU Toolchain (Version 2025.12.18) kompiliert. Das Testprogramm hat eine Größe von ca. 73 kB. Es wurden die folgenden Compileroptionen verwendet: `-march=rv32i_zicsr -mabi=ilp32 -O3`. Es wurde folgender Commit verwendet: `f174215fed7b286e19b120133c3f6e0f19166229`

Zum Vergleich wurde auch eine Resynthese durchgeführt, bei der ausschließlich das Testprogramm geändert wurde. Der überwiegende Anteil der Zeit entfällt dabei auf das Platzieren und Verdrahten. Die Resynthese dauerte `110 s`, siehe Anhang A.1. Die Durchführung der Messung der Resynthesezeit ist in Anhang A.1 dokumentiert. Die Spezifikation des verwendeten PCs ist in Anhang A.4 dokumentiert.

Für das Laden des Programms in den Hauptspeicher wurde die Debug-Kette mit JTAG-Adapter (Olimex ARM-USB-TINY-H), OpenOCD und GDB aufgebaut. Die JTAG-Taktfrequenz wurde auf `5 MHz` eingestellt. GDB wurde mit dem Zielprogramm gestartet und das Laden erfolgte über den Befehl `load`. GDB meldete dabei eine Übertragungsrate von `12 KB/s`. Das Testprogramm wurde in `6,3 s` erfolgreich übertragen.

Der größte Zeitanteil entfällt dabei auf Platzieren und Verdrahten. Die Gesamtdauer steigt durch ein größeres Programm nur begrenzt, da lediglich der Kompilationsschritt des Programms aufwendiger wird. Das Laden über die Debug-Infrastruktur ist dagegen stark von der Programgröße abhängig, weil hier die Datenübertragung selbst den Hauptfaktor darstellt, während sie bei der Resynthese konstant bleibt.

7 Fazit

7.1 Zusammenfassung

Im Rahmen dieser Bachelorarbeit wurde eine vollständig synthetisierbare Debug-Infrastruktur für den PicoNut/RISC-V-Prozessor entwickelt und erfolgreich auf FPGA-Hardware implementiert. Ziel war es, Debugging direkt auf realer Hardware zu ermöglichen und damit die bisherige Einschränkung auf eine rein simulationsbasierte Debug-Lösung zu überwinden.

Die entwickelte Infrastruktur orientiert sich konsequent an der *RISC-V Debug Specification* [9] in Version 1.0 und umfasst sämtliche zentralen Komponenten dieser Architektur: einen Debug-Modus im Prozessor, die notwendigen Debug-CSRs sowie eine Debug-Peripherie bestehend aus Debug Transport Module (DTM) und Debug Module (DM). Die Kommunikation mit externen Debug-Werkzeugen erfolgt standardkonform über eine JTAG-basierte Anbindung und ermöglicht die Integration von OpenOCD und GDB. Die Schnittstelle die der Nucleus mitbringen muss ist zudem schmall gehalten und umfasst ledeglich zwei Hardwaresignale, sowie eine Sprungadresse, welche ihm bekannt sein muss.

Die Implementierung erlaubt das Anhalten und Fortsetzen der Programmausführung, das Setzen von Breakpoints, die Einzelschrittausführung, sowie Lese- und Schreibzugriffe auf Register und Speicher. Darüber hinaus wurde eine effiziente Möglichkeit geschaffen, Programme zur Laufzeit in den Hauptspeicher des PicoNuts zu laden. Dadurch konnte die Entwicklungs- und Testzeit von Software auf FPGA-Hardware signifikant reduziert werden.

Die Synthese- und Implementierungsergebnisse zeigen, dass der zusätzliche Ressourcenbedarf der Debug-Infrastruktur moderat ausfällt und in einem angemessenen Verhältnis zum gewonnenen Funktionsumfang steht. Insgesamt konnte damit gezeigt werden, dass ein leistungsfähiges, standardkonformes Debugging für den PicoNut auch unter realen Hardwarebedingungen praktikabel umsetzbar ist.

7.2 Ausblick

Zukünftige Arbeiten könnten die Unterstützung mehrerer Kerne ermöglichen, um den Debug-Support auf Mehrkern-Varianten des Prozessors auszuweiten.

Um den Hardwareverbrauch zu verringern könnten zukünfitg die ABSTRACT[0..7]-Register durch virtuelle Register ersetzt werden. Die Generierung der RISC-V-Assembler-Befehle würde in so einem Design On-Demand stattfinden.

In dieser Arbeit wurden Register- und Speicherzugriffe ausschließlich über abstrakte Befehle implementiert. Da diese nur Register- und Speicherzugriffe implementieren ist das Ausführen von anderen Befehlen nicht Möglich. Dadurch ist beispielsweise die Ausführung des `fence`-Befehls, welcher dafür sorgt, dass alle Speicheroperationen vor diesem Befehl abgeschlossen sind, nicht möglich. Sobald im Nucleus eine Out-of-Order-Architektur implementiert wird, kann dies zu Inkonsistenzen führen. In diesem Fall muss der Program Buffer gemäß der *RISC-V Debug Specification* [9] unterstützt werden.

Eine weitere interessante Perspektive stellt die Unterstützung alternativer Debug-Transportmechanismen neben JTAG dar, etwa über serielle oder netzwerkbasierte Schnittstellen.

Nicht zuletzt eröffnet die nun verfügbare Debug-Infrastruktur neue Möglichkeiten für den Einsatz des PicoNut in Lehre und Forschung. Insbesondere das Debugging von Betriebssystemen, hardware-naher Software sowie externer Peripherie auf echter FPGA-Hardware wird dadurch erheblich erleichtert und erweitert den praktischen Nutzen des PicoNut-Prozessors nachhaltig.

Literaturverzeichnis

[1] Accellera Systems Initiative: *Webseite SystemC*. 2026. URL: https://systemc.org/ (siehe S. 3).

[2] Bauer, Lukas: "Weiterentwicklung der Debug-Infrastruktur des ParaNut-Prozessors". Bachelorarb. TH Augsburg, 2023 (siehe S. 19).

[3] Forschungsgruppe EES: *Effiziente Eingebettete Systeme*. TH Augsburg. Mai 2025. URL: https://ees.tha.de/ (siehe S. 3).

[4] Forschungsgruppe EES: *PicoNut Manual*. Mai 2025. URL: https://ees.tha.de/piconut/manual (siehe S. 4).

[5] GDB developers: *GDB: The GNU Project Debugger*. 2025. URL: www.sourceware.org/gdb/ (besucht am 10. 04. 2025) (siehe S. 4, 5).

[6] Hofmann, Johannes: "Implementierung einer Debug-Infrastruktur für einen RISC-V-Prozessor in Simulationsumgebung". Techn. Ber. TH Augsburg, Mai 2025 (siehe S. 18).

[7] IEEE: *IEEE Standard Test Access Port and Boundary-Scan Architecture*. 1990. DOI: 10.1109/IEEESTD.1990.114395. URL: https://ieeexplore.ieee.org/document/211226 (siehe S. 6, 7, 9, 11).

[8] Moiseev, Mikhail: *ICSC Repository*. 2026. URL: https://github.com/intel/systemc-compiler (besucht am 11. 01. 2026) (siehe S. 3).

[9] Newsome, Tim; Donahue, Paul: *The RISC-V Debug Specification*. 1.0. Feb. 2025. URL: https://drive.google.com/file/d/1h_f9NgB_8m2fS6uCnKP1Oho-3x1MpBEl/view?usp=drive_link (siehe S. 1, 5, 11, 18–20, 26, 43, 44).

[10] OpenHW Group: *CVA6 RISC-V-Prozessor*. Feb. 2026. URL: https://github.com/openhwgroup/cva6 (siehe S. 19).

[11] PULP platform: *PULP Debug Support*. Feb. 2026. URL: https://github.com/pulp-platform/riscv-dbg (siehe S. 19).

[12] Schäfterling, Michael: *PicoNut*. Mai 2025. URL: https://ees.tha.de/piconut (besucht am 27. 05. 2025) (siehe S. 3).

[13] TH Augsburg: *Webseite TH Augsburg*. 2026. URL: https://www.tha.de/ (siehe S. 3).

[14] The OpenOCD Project: *Open On-Chip Debugger: OpenOCD User's Guide*. 0.12.0. The OpenOCD Project. 2022. URL: https://openocd.org/doc-release/pdf/openocd.pdf (siehe S. 5, 6).

[15] YosysHQ: *Webseite YosysHQ*. 2026. URL: https://yosyshq.net/ (siehe S. 3).

Ich, Johannes Hofmann, versichere hiermit, dass ich die vorliegende Arbeit mit dem Thema

> *Implementierung einer Debug-Infrastruktur für den PicoNut/RISC-V-Prozessor auf FPGA-Hardware*

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Während der Vorbereitung dieser Thesis nutzte der Autor ChatGPT zur Rechtschreib- und Grammatikprüfung. Der Autor überprüfte die vom Tool vorgeschlagenen Änderungen und passte den Text bei Bedarf an. Er übernimmt die volle Verantwortung für den Inhalt dieser Thesis.

Augsburg, den 13. Mai 2026

Johannes Hofmann

A Anhang

A.1 Laufzeitmessung der Synthese

Für die Evaluierung der Synthesezeiten wurde das PicoNut Referenzdesign für den Lattice ECP5-85k FPGA fünfmal mit identischen Parametern resynthetisiert. Es wurde jeweils von einer Veränderung des Testprogramms künstlich erzeugt. Es wurde folgender Commit verwendet: `f174215fed7b286e19b120133c3f6e0f19166229`

#	1	2	3	4	5	Ø
Zeit [s]	138,18	137,38	137,79	138,93	138,25	138,11

Tabelle A.1: Ergebnis der Laufzeitmessung der Synthese

In Tabelle A.2 ist das Syntheseergebnis des Designs dargestellt, welches während der Synthesezeitmessung erzeugt wurde.

Zelltyp	Belegt	Verfügbar	Auslastung
TRELLIS_IO	4	365	1%
DP16KD	128	208	61%
MULT18X18D	0	156	0%
ALU54B	0	78	0%
TRELLIS_FF	1892	83640	2%
TRELLIS_COMB	8569	83640	10%
TRELLIS_RAMW	0	10 455	0%

Tabelle A.2: Statistik des bei der Synthesezeitmessung erstellten Designs für das Lattice ECP5-85k FPGA

A.2 Debug-Handler

Listing A.1: Debug-Handler im RISC-V Assembler

```
1 debug_handler:
2    csrrw x0,      dscratch0, x8
3    addi  x8,      x0,        1
```

```
4     sw      x8,       0x38(x0)        /* set HARTSTATUS halted bit
          */
5
6   loop:
7     lw      x8,       0x34(x0)
8     andi    x8,       x8,         3
9     beq     x0,       x8,         loop
10
11  run_commands:
12    andi    x8,       x8,         2
13    beq     x8,       x0,         resume
14    addi    x8,       x0,         9
15    sw      x8,       0x38(x0)        /* set HARTSTATUS
          acmds_running bit */
16    csrrw   x8,       dscratch0,  x0
17    jalr    x0,       0x14(x0)
18
19  resume:
20    addi    x8,       x0,         2
21    sw      x8,       0x38(x0)        /* set HARTSTATUS running bit
          */
22    csrrw   x8,       dscratch0,  x0
23    .word 0x7b200073                  /* dret */
```

A.3 Testprogramm

Listing A.2: Testprogramm zur Messung der Ladegeschwindigkeit

```
1   #include <stdio.h>
2
3   #define DO_LOOP 0
4
5   #define KNRM "\x1B[0m"
6   #define KRED "\x1B[31m"
7   #define KGRN "\x1B[32m"
8   #define KYEL "\x1B[33m"
9   #define KBLU "\x1B[34m"
10  #define KMAG "\x1B[35m"
11  #define KCYN "\x1B[36m"
12  #define KWHT "\x1B[37m"
13
14  int main()
15  {
16    do
```

```
17   {
18     printf("%s______ _            _    _        _        %s__%
         s_____ _____ _____ _____        %s_    _ \n", KMAG, KNRM,
         KBLU, KYEL);
19     printf("%s| ___ (_)          | \\ | |      | |      %s/ /%s
         ___ \\_    _/  ___/  __ \\     %s| | | |\n", KMAG, KNRM,
         KBLU, KYEL);
20     printf("%s| |_/ /_  ___ ___ |  \\| |_    _| |_    %s/ /%s| |_
         / / | | \\ `--.| /  \\/%s_____| | | |\n", KMAG, KNRM,
         KBLU, KYEL);
21     printf("%s|  __/| |/ __/ _ \\| . ` | | | | __| %s/ /%s |
           /  | |  `--. \\ |  %s|______| | | |\n", KMAG, KNRM,
         KBLU, KYEL);
22     printf("%s| |   | | (_| (_) | |\\  | |_| | |_ %s/ /%s  |
         |\\ \\ _| |_/\\__/ / \\__/\\     %s\\ \\_/ /\n", KMAG,
         KNRM, KBLU, KYEL);
23     printf("%s\\_|   |_|\\___\\___/\\_| \\_/\\__,_|\\__%s/_/%s
           \\_| \\_|\\___/\\____/ \\____/      %s\\___/%s \n\n\n"
         , KMAG, KNRM, KBLU, KYEL, KNRM);
24   } while(DO_LOOP);
25
26   return DO_LOOP;
27 }
```

A.4 Systemdetails

Nachfolgend sind die Systemdetails aufgeführt, auf dem die Messungen durchgeführt wurden.

```
Hardware:   ASUSTeK VivoBook TP420IA
CPU:        AMD Ryzen 7 4700U (8 Threads)
RAM:        16 GiB
GPU:        AMD Radeon Graphics
Disk:       512 GB NVMe

OS:         Debian GNU/Linux 13 (trixie)
GNOME:      Version 48 (Wayland)
Kernel:     Linux 6.12.48
Firmware:   TP420IA.305
```