Hochschule
Augsburg University of
Applied Sciences

Fakultät für
Informatik

Bachelorarbeit

Studienrichtung
Technische Informatik

Weiterentwicklung der Debug-Infrastruktur des ParaNut-Prozessors

im Fachgebiet Effizente Eingebettete Systeme

Prüfer: Professor Gundolf Kiefer
Zweitpüfer: Professor Hubert Högl

Verfasser:
Lukas Bauer
Selibertstraße 6
82276 Adelshofen
+49 1578 3696662
lukas.bauer@hs-augsburg.de
Matrikelnr.: 2073233

Hochschule für angewandte Wissenschaften Augsburg
An der Hochschule 1
86161 Augsburg
Telefon: +49 (0)821-5586-0
Fax: +49 (0)821-5586-3222
info@hs-augsburg.de

Inhaltsverzeichnis

Abkürzungsverzeichnis

AXI	Advanced eXtensible Interface
CePU	Centreral Processing Unit
CoPU	Co Processing Unit
CSR	Control and Status Register
DM	Debug Module
DMI	Debug Module Interface
DR	test data-register
DTM	Debug Transport Module
dtmcs	DTM Controle and Status
EXU	Execution Unit
FIFO	First in First out
FPGA	Field Programable Gate Array
GDB	GNU Debugger
GPR	General Purpose Register
hart	hardware thread
HLS	High Level Synthese
ICSC	Intel Compiler for SystemC
IR	instruction register
JTAG	Joint Test Action Group
LSB	Least Significant Bit
MSB	Most Significant Bit
OpenOCD	Open On-Chip Debugger
RBB	Remote Bitbang
TAP	Test Access Port
TCK	Test Clock Input
TDI	Test Data Input
TDO	Test Data Output

TMS	Test Mode Select Input
TRST	Test Reset Input
tui	text user interface
UART	Universal Asynchronous Receiver Transmitter
VHDL	Very High Speed Integrated Circuit Hardware Description Language

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

1.1 Motivation

Software-Debugger wie der GNU Debugger (GDB) erlauben es dem Entwickler, das Verhalten seiner Anwendung zu analysieren, indem er unter anderem die Möglichkeit erhält, das Programm anzuhalten, Daten auszulesen und zu verändern [3]. So können Fehler in Programmen gefunden und behoben werden. Für die Fehlersuche bei Software, die auf einem Betriebssystem läuft, können Programme viele Aufgaben übernehmen, jedoch kann es vorkommen, dass Unterstützung durch die Hardware benötigt wird. Wenn ein System jedoch ohne Betriebssystem arbeitet, ist Hardwareunterstützung umso wichtiger [11]. Die Hardware bildet mit der Software die Debug-Infrastruktur.

Im Fall des ParaNut besteht diese aus einem, dem RISC-V Standard entsprechendem, „Debug Module“ (DM), einem „Debug Module Interface“ (DMI), einem „Debug Transport Module“ (DTM), welches eine „Joint Test Action Group“ (JTAG) Schnittstelle bereitstellt. Bei DM, DMI und DTM handelt es sich um Hardwaremodule(Beschreibung in Kapitel 2.5.3). Weiterhin besteht die Infrastruktur aus einem JTAG Adapter zur Kommunikation mit dem Host-System, eine Interface Software, hier „Open On Chip Debugger“ (OpenOCD), und einen Software-Debugger. Beim ParaNut handelt es sich um das Programm GDB. Außerdem wird eine „Universal Asynchronous Receiver Transmitter“ (UART)-Schnittstelle benötigt, um Daten mit dem ParaNut-System auszutauschen.

Eine Überarbeitung der Komponenten der Debug-Infrastruktur wird sicherstellen, dass diese zuverlässig funktioniert. Dazu werden die einzelnen Teile analysiert, um in Erfahrung zu bringen, welche Teile erweitert oder verbessert werden können und diese in die Komponenten einzubauen. Damit wird sichergestellt, dass Fehler bei der Entwicklung von Programmen für den ParaNut gefunden werden können.

1.2 Ziel der Arbeit

Das Ziel dieser Bachelorarbeit ist es, die Nutzbarkeit der Debug-Infrastruktur des ParaNuts, durch die Überarbeitung der einzelnen Komponenten, sicherzustellen und zu erweitern, um das Debuggen von Software wie zum Beispiel Linux zu ermöglichen. Auch soll die Stabilität der Debug-Infrastruktur verbessert werden, sodass diese, wenn sie benötigt wird funktioniert. Außerdem soll die Abhängigkeit von der Firma Xilinx, die die im ParaNut-Projekt verwendeten „Field Programable Gate Array“ (FPGA) Boards herstellt, minimiert werden.

Um dies zu erreichen, soll die aktuell im ParaNut-Projekt verwendete Version von GDB, die nur über eine Kommandozeile verfügt und zudem nicht mehr aktualisiert wird, um eine Version ergänzt werden die Updates erhält und über ein „text user interface“ verfügt. Damit soll der Entwickler einen besseren Überblick über, die Abläufe bei Ausführung eines Programms erhalten. Was beim Debuggen von komplexer Software die Arbeit erleichtert. Auch ist so möglich, Fehler in GDB durch Aktualisierungen zu beheben, so kann die Zuverlässigkeit des Debuggers verbessert werden.

Weiterhin wird die bereits existierende Speicherzugriffsart des DM's, durch eine in der Theorie schnellere Art des Speicherzugriffs ergänzt. So wird es in Zukunft möglich, den Wert einer Variable, wie einem String, schneller zu verändern. So kann beim Debuggen Zeit gespart werden.

Die Testinfrastruktur der Debug-Infrastruktur ist bisher unzureichend. Sie besteht nur aus Testbenches für die Hardwaremodule DM und DTM. Um den Zustand zu verbessern, werden zum einen die Testbenches erweitert, zum anderen wird eine Möglichkeit geschaffen, um die Debug-Infrastruktur als Komplettsystem zu testen. Außerdem wird ein Testkonzept für die Debug-Infrastruktur erstellt. So wird sichergestellt, dass Probleme an der Infrastruktur schnell entdeckt und behoben werden können, um die Fähigkeit zum Debuggen zu garantieren.

Für die Übertragung von Daten auf die verwendeten FPGA-Boards, von Xilinx, wird bisher der darauf verbaute ARM-Prozessor benötigt, der die UART-Kommunikation mit dem ParaNut übernimmt. Um die Abhängigkeit von dem Hersteller der FPGAs zu reduzieren, wird eine eigene UART-Schnittstelle in den ParaNut eingebaut und ein Bootloader erstellt, der die Kommunikation mit dem Computer des Entwicklers übernimmt.

1.3 Aufbau der Arbeit

Im folgenden Kapitel 2 „Grundlagen“ wird die RISC-V Befehlssatzarchitektur und der „External Debug Standard“ vorgestellt, darüber hinaus werden benötigte Technologien, wie UART und JTAG beschrieben. Darüber hinaus wird die ParaNut-Architektur beschrieben. Im darauffolgenden Kapitel 3 „Kompatibilität mit aktueller Version von GDB“ wird beschrieben, wie die Kompatibilität mit einer „text user interface“ kompatiblen Version von GDB hergestellt wurde. Kapitel 4 „Beschleunigter Speicherzugriff des DM's“ beschreibt die Implementierung und den Test einer neuen Speicherzugriffsart. Darüber hinaus wird erläutert, wie die Kompatibilität mit OpenOCD dabei sichergestellt werden konnte. Das Kapitel 5 „Testinfrastruktur der Debug-Infrastruktur“ behandelt, wie die Testinfrastruktur für die zum Debuggen benötigten Hardware-Module verbessert wurde. Wie die Implementierung des neuen UART-Moduls und des Bootloaders für Software durchgeführt wurde, wird im Kapitel 6 „Austausch der Kommunikationsschnittstelle des ParaNuts“ beschrieben. Das Kapitel 7 „Ergebnisse“ beinhaltet einen experimentellen Vergleich der verschiedenen Speicherzugriffsarten. Im letzten Kapitel 8 „Fazit“, wird die Arbeit noch einmal zusammengefasst und ein Ausblick auf mögliche weitere Schritte gegeben.

2 Grundlagen

2.1 UART

Der Begriff UART steht für „Universal Asynchronous Receiver Transmitter“ und ist eine einfache serielle Schnittstelle. Für die Übertragung von Daten benötigt diese 2 Leitungen. Eine wird für das Senden von Daten benutzt und trägt den Namen „TX“. Die Andere ist für das Empfangen von Daten zuständig und heißt „RX“ [1]. Beide Leitungen sind im Ruhezustand auf „high“ gezogen. Damit die Datenübertragung zwischen zwei UART-Schnittstellen funktioniert, braucht es keine Leitung, die ein Takt zwischen beiden überträgt, denn bei diesem Protokoll wird der Takt auf beiden Seiten der Übertragung generiert. Die Anzahl der Bits, die pro Sekunde übertragen werden, wird bei UART Baudrate genannt. Es gibt weitere Einstellungen, die bei beiden Interfaces gleich sein müssen. Die Anzahl an Bits pro Übertragung, wird Zeichen genannt und muss zwischen den UART-Schnittstellen gleich sein. Auch muss

die gleiche Einstellung für das Paritätsbit gewählt werden und die Anzahl der Stopbits übereinstimmen. Eine Datenübertragung läuft wie folgt ab.

Das Interface, das die Daten senden will, zieht seine „TX“ für einen Takt auf „low“, was den Start einer Übertragung signalisiert und Startbit genannt wird. Danach werden die Datenbits übertragen, wobei das „Least Significant Bit“ LSB zuerst gesendet wird. Daraufhin wird ein Paritätsbit übertragen, wenn die Parität aktiv ist. Zuletzt folgt noch die Anzahl der konfigurierten Stopbits. Dafür wird die „TX“ Leitung auf „high“ gezogen. Damit ist die Übertragung beendet und eine neue kann gestartet werden [9].

2.1.1 Der TI16C750-Chip

Ein Chip, der das im Kapitel 2.1 beschriebene UART-Protokoll implementiert, ist der TL16C750 von Texas Instruments. Dieser Chip erlaubt dem Benutzer eine Vielzahl von Konfigurationsmöglichkeiten. Die Konfiguration erfolgt über adressierbare Register. Es können UART spezifische Einstellungen getroffen werden, wie zum Beispiel die Anzahl der Datenbits pro Zeichen, die Parität oder die Anzahl der Stopbits. Darüber hinaus verfügt der Chip über einen programmierbaren Baudrate Generator, mit dem der 16C750 die Taktrate selbst generieren kann. Diese Baudrate wird über die beiden „Devisor Latch“ Register konfiguriert. Somit können Baudraten von bis zu einem Mega Baud generiert werden. Außerdem lassen sich für den Sender und den Empfänger „First In First Out“ (FIFO)-Speicher aktivieren. Damit ist es möglich, bis zu 64 Zeichen, mit einer Länge von 8 Bit, zwischenzuspeichern. Ist der Chip fertig konfiguriert, können die zu sendenden Zeichen auf den Datenbus des Chips geschrieben werden. Der 16C750 sendet die geschriebenen Zeichen automatisch. Empfangene Daten werden, bei aktivierter FIFO, zwischengespeichert und sind dann durch ein Lesen am Datenbus abrufbar [17].

2.2 JTAG

2.2.1 Überblick

Die Bezeichnung JTAG ist ein anderer Name für die IEEE Spezifikation 1149.1. In diesem Standard wird das JTAG „Test Access Port“ (TAP) definiert [16, 11]. Mit der JTAG-Schnittstelle wird es möglich, Verbindungen zwischen Chips oder die Chips selbst zu testen. Außerdem ermöglicht die Schnittstelle das Beobachten und Verändern eines Chips während des normalen Betriebs [5].

2.2.2 JTAG-Signale

Die TAP-Schnittstelle besteht dabei aus bis zu fünf Leitungen:

- „`Test Clock Input" (TCK)`: Der Takt für den TAP-Controller, ist unabhängig vom Takt des beobachteten Systems
- „`Test Mode Select Input" (TMS)`: Steuert den TAP-Controller
- „`Test Data Input" (TDI)`: Dateneingang den TAP-Controller
- „`Test Data Output" (TDO)`: Datenausgang den TAP-Controller
- „`Test Reset Input" (TRST)`: Initialisieren den TAP-Controller und des beobachteten Systems

[5]

2.2.3 JTAG-Register

Der Standard definiert außerdem Register, die in dem, in Abbildung 2.1 zu sehenden Zustandsautomaten, benötigt werden. Dabei handelt es sich um die Datenregister (DR), deren Daten über Befehle geändert werden können. Des Weiteren wird noch das Befehlsregister (IR) benötigt, um das Datenregister auszuwählen [5].

2.2.4 JTAG-Befehle

Es gibt bereits vordefinierte Befehle, die in das Befehlsregister geladen werden können. Einer davon ist der Befehl „BYPASS“, welcher ein 1-Bit breites Schieberegister als Datenregister zur Verfügung stellt. Ein weiterer Befehl ist „IDCODE“, der den Zugriff auf das Datenregister mit den Herstellerangaben bereitstellt. Außerdem können eigene Befehle in der TAP-Controller eingebaut werden.[5].

2.2.5 TAP-Zustandsmaschine

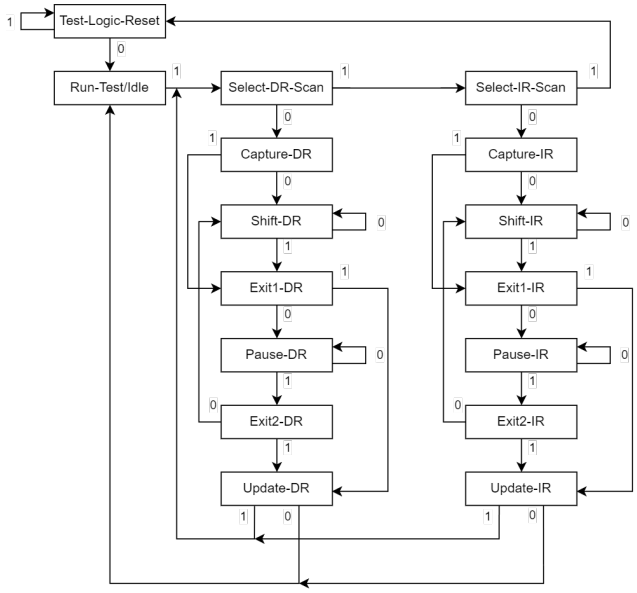

Abbildung 2.1: JTAG Zustandsmaschine (angelehnt an [5])

Anhand der Abbildung 2.1 soll nun die Funktionsweise der des JTAG TAP-Controllers erklärt werden. Hierbei wird nur auf die für diese Arbeit benötigten Zustände eingegangen. Die Zustandsübergänge erfolgen zur steigenden Taktflanke des Signals `TCK`. Dabei wird der nächste Zustand von dem Signal `TMS` bestimmt [5].

Nachdem der Controller mit Strom versorgt wird, befindet sich dieser im Zustand „Test-Logic-Reset“, der einen uneingeschränkten Betrieb des beobachteten Systems zulässt. Außerdem wird der Befehl im IR, auf definierten Wert, zurückgesetzt. Wenn in diesem Zustand `TMS` aktiviert wird, wechselt der TAP-Controller in den Zustand „Run-Test/Idle“ [5].

Dort wird der Inhalt der DR- und IR-Register nicht verändert. Der Zustand wird gewechselt, wenn `TMS` aktiv ist [5].

Dann wird in den Zustand „Select-DR-Scan“ gewechselt, der keine Änderungen an DR oder IR durchführt. Er bietet die Möglichkeit in den Zustand „Select-IR-Scan“, wenn `TMS` den Wert „1“ oder in den Zustand „Capture-DR“ zu wechseln, wenn `TMS` den Wert „0“ hat [5].

Im Zustand „Select-IR-Scan“ besteht die Möglichkeit zurück in den Zustand „Test-Logic-Reset“ zu wechseln, wenn `TMS` aktiv ist, oder in den Zustand „Capture-IR“ zu wechseln, wenn die Leitung `TMS` deaktiviert ist. Dadurch werden die Register nicht geändert [5].

Im Zustand „Capture-DR“ werden Daten in das DR Register, abhängig von dem in IR stehenden Befehl, in das DR Schieberegister geladen. Wenn `TMS` gleich „1“ ist, wird in den „Exit1-DR“ Zustand gewechselt, wenn es „0“ ist, wird der „Shift-DR“ Zustand betreten [5].

In „Shift-DR“ liegt das „Least Significant Bit“ (LSB) des DR Registers an `TDO` an. Am „Most Significant Bit“ (MSB) liegt `TDI` an. Wenn in diesem Zustand `TMS` den Wert „0“ hat, wird der Inhalt des DR Registers nach rechts geschoben. Somit wird der Wert des LSB auf `TDO` ausgegeben und der Wert von `TDI` in das MSB übernommen. Wenn `TMS` den Wert „1“ hat, wird in den Zustand „Exit1-DR“ gewechselt [5]. So können in diesem Zustand die Daten bitweise aus dem Datenregister gelesen oder in dieses geschrieben werden.

In diesem Zustand kann entschieden werden, ob der nächste Zustand „Pause-DR“ oder ob von hier aus in den Zustand „Update-DR“, wenn `TMS` aktiv ist, gewechselt wird [5].

In „Update-DR“ werden die in DR liegenden Daten abgespeichert. Von diesem Zustand aus kann wieder in die Zustände „Run-Test/Idle“ oder „Select-DR-Scan“ gewechselt werden [5].

Wenn vom „Select-IR-Scan“ Zustand zu „Capture-IR“ gewechselt wurde, wird der aktuelle Befehl in das IR Schieberegister geladen. Aus diesem Zustand kann analog zum DR in die Zustände „Shift-IR“ und „Exit1-IR“ gewechselt werden [5].

Im Zustand „Shift-IR“ werden analog zu Zustand des DR `TDO` mit dem LSB und `TDI` mit dem MSB des IR Schieberegisters verbunden. So können analog zum DR Daten in das IR Register geschrieben oder gelesen werden. Durch das Setzen von `TMS` auf den Wert „1“ kann in den Zustand „Exit1-IR“ gewechselt werden [5]. So können in diesem Zustand die Daten bitweise aus dem Befehlsregister gelesen oder in dieses geschrieben werden.

Dieser Zustand hat die selbe Funktionalität wie „Exit1-DR“. Von diesem Zustand kann entweder in „Pause-IR“ oder „Update-IR“ gewechselt werden [5].

Dieser Zustand übernimmt die Daten im IR Schieberegister als neuen Befehl. Analog zum Zustand „Update-DR“ können die Zustände „Run-Test/Idle“ oder „Select-DR-Scan“ erreicht werden [5].

2.3 GDB

Bei GDB handelt es sich um einen Debugger. Er erlaubt es dem Entwickler, das Verhalten von Anwendungen zu analysieren. Um die Ausführung des Programms zu beeinflussen, werden nach dem Start des Debuggers, Befehle in GDBs Kommandozeile eingegeben. Die Anwendung kann dabei lokal auf dem Rechner des Programmierers untersucht werden oder auch auf einem entfernten System. Die Option des Remote-Debuggings wird dann verwendet, wenn der Kernel eines Betriebssystems oder ein System ohne Betriebssystem untersucht werden sollen. Dazu wird mit Hilfe einer seriellen Schnittstelle oder Netzwerkschnittstelle eine Verbindung mit dem zu untersuchenden System aufgebaut. Danach kann mit Hilfe eines von GDB spezifizierten Protokolls die Software, die auf dem entfernten System ausgeführt wird, gedebuggt werden.

Wenn GDB mit eingebetteten Systemen betrieben wird, muss darauf geachtet werden, dass die Eigenheiten des Systems dem Debugger bekannt sind. Um dieses Problem zu umgehen, erlaubt GDB es, dass ein Zielsystem seine Konfiguration, über das von GDB definierte Protokoll für entfernte Systeme mitzuteilen. Außerdem erlaubt es GDB auch, die Beschreibung des Zielsystems aus einer Datei zu laden. GDB verfügt über einen tui Modus, der eine Benutzeroberfläche im Terminal bereitstellt. Dieses „text user interface“ ermöglicht die Anzeige von Quellcode, Assembler-Befehlen und Registerinhalten. Er wird nur auf Systemen verfügbar, die die „curses“ Bibliothek unterstützen.

Weiter ist es möglich, ein sogenanntes „command file“ für das Debuggen mit GDB zu benutzen. Dabei handelt es sich um eine Textdatei die GDB-Befehle beinhaltet. Es ist möglich eine solche Datei mit Hilfe des Schalters `-x <Pfad zur Datei>` beim Start von GDB zu laden, wodurch die darin enthaltenen Befehle ausgeführt werden [15].

2.4 OpenOCD

OpenOCD erlaubt es einem Entwickler, mit Hilfe eines „debug adapters“, ein eingebettetes System zu debuggen, Programme auf Hardware zu flashen und „boundary scans“ durchzuführen. Wenn OpenOCD gestartet wird, übernimmt es die Kommunikation mit dem über den „debug adapter“ angeschlossenem System. Dabei kann mit Hilfe des Schalters `-f`, der Pfad zu einer Konfigurationsdatei angegeben werden. Diese Datei beinhaltet die Einstellungen des „debug adapters“ und Informationen über die Konfiguration des angeschlossenen Systems.

Um Einfluss auf das System zu nehmen, öffnet OpenOCD mehrere Netzwerkschnittstellen mit verschiedenen Protokollen. Dazu zählen eine Telnet-Schnittstelle, über welche Befehle an OpenOCD geschickt werden können und eine gdbserver-Schnittstelle, welcher die Verbindung mit GDB ermöglicht [19, 20].

2.4.1 Remote Bitbang-Protokoll

Bei Remote Bitbang (RBB) handelt es sich um einen Treiber für OpenOCD, der das JTAG-Protokoll für die Verwendung mit Prozessen bereitstellt. Der Aufbau kann für das Debuggen in Simulatoren benutzt werden, um Befehle, die in ASCII kodiert sind, an den Prozess mit der RBB-Schnittstelle zu schicken. Diese bilden das Verhalten einer JTAG-Schnittstelle nach. Mit dem Zeichen „R“ wird der Wert von `tdo` zurückgegeben. Mit den Befehlen „0“ bis „7“ können alle Kombinationen von `TCK`, `TMS` und `TDI` dargestellt werden. Eine Auflistung der Befehle ist in der Tabelle 2.1 zu sehen [18].

Befehl	TCK	TMS	TDI
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Tabelle 2.1: Schreibbefehle von Remote Bitbang

Darüber hinaus gibt es noch weitere Befehle, die jedoch nicht für die Arbeit benötigt werden [18].

2.5 RISC-V

2.5.1 RISC-V Befehlssatzarchitektur

RISC-V ist der Name einer freien, offenen und modularen Befehlssatzarchitektur, dessen Entwicklung 2010 an der UC Berkeley begann. Seit dem Jahre 2015 ist die RISC-V Foundation für die Weiterentwicklung zuständig [22].

Die Architektur besteht aus zwei Teilen, dem privilegierten und dem unprivilegierten Teil [vgl. 21, 22]. Befehle, die zum unprivilegierten Teil der Befehlssatzarchitektur

gehören, können in sämtlichen privilegierten Modi verwendet werden. Im Vergleich dazu können privilegierte Befehle nur mit den erforderlichen Rechten ausgeführt werden. [22].

Darüber hinaus besteht die Befehlssatzarchitektur aus einem Basisbefehlssatz, der in verschiedenen Bitbreiten verfügbar ist. Die Grundkonfiguration kann mit Befehlssatzerweiterungen ergänzt werden, die zum Beispiel die Multiplikation und Division ermöglichen [22]. In der RISC-V Befehlssatzarchitektur gibt es des Weiteren zwei Arten von „Control and Status Register“ (CSR), „standard“ und „custom“. Register vom Typ „standard“ sind in der Spezifikation definiert, wohingegen Register des Typs „custom“ durch die spezifische Implementierung definiert werden [21].

In der RISC-V-Befehlssatzarchitektur kann ein Kern aus einem oder mehrere „hardware threads“ (harts) bestehen [11].

2.5.2 RISC-V Privilege Levels

In der Spezifikation für den privilegierten Teil der RISC-V Befehlssatzarchitektur wird erklärt, wie verschiedene „privelege-level“ umgesetzt werden müssen. Im aktuellen Standard gibt es drei verschiedene Berechtigungsstufen. Das höchste „privileg-level“ ist das „Machine“ Level (M-Mode), was eine umfangreiche Kontrolle der Hardware ermöglicht. Darüber hinaus gibt es noch die „privileg-level“ „Supervisor“ (S-Mode) und „User“ (U-Mode). Dabei ist das Level „Supervisor“ für Betriebssystem gedacht, wobei das „User“ Level für normale Anwendungen konzipiert wurde. Jede Implementierung der Befehlssatzarchitektur kann zwischen einem und drei der genannten „privileg-level“ umsetzen. Dabei ist nur die Umsetzung des „Machine“ Levels verpflichtend, um den Zugriff auf das komplette System zu ermöglichen. Jeder Level verfügt über optionale Erweiterungen und eigene CSR-Register. Zusätzlich zu den bereits genannten „privileg-level“ gibt es noch den „Debug Mode“ (D-Mode), der wie ein viertes „privileg-level“ gesehen werden kann. Der D-Mode hat dabei erweiterte Zugriffsrechte im Vergleich zum M-Mode, die weitere CSR-Register und das Reservieren von Teilen des physischen Speichers beinhalten [21].

2.5.3 RISC-V Debug Unterstützung

In der RISC-V Befehlssatzarchitektur (ISA) gibt es die Spezifikation „RISC-V External Debug Support“. In dieser wird ein Standard für die Implementierung der „External Debug Support“ Architektur für RISC-V Systeme beschrieben, um sicherzustellen, dass verschiedene System von Tools wie GDB unterstützt werden können

und trotzdem mehrere Wege der Implementierung für die zum Debuggen benötigten Hardware zu erlauben [11].

Zu den Funktionen zählen unter anderem:

- Zugriff auf alle Register eines harts
- Zugriff auf den Speicher
- Das Setzen von Breakpoints

[11]

Eine genauere Erklärung findet in Kapitel 2.7 statt. Dort wird ebenfalls auf die Implementierung im ParaNut eingegangen.

2.6 ParaNut-Architektur

2.6.1 ParaNut-Allgemein

Der Begriff ParaNut beschreibt eine, von der Forschungsgruppe Effiziente Eingebettete Systeme (EES) der Hochschule Augsburg entwickelte, Prozessorarchitektur. Diese basiert auf der freien Befehlssatzarchitektur RISC-V und soll eine skalierbare und offene Plattform für FPGA-Systeme darstellen. [8, 14].

Die ParaNut-Architektur verfolgt einen eigenen Ansatz bei der Parallelisierung. Dabei liegt der Fokus auf Parallelität auf Thread- und Daten-Ebene. Um Platz auf dem FPGA, zu sparen werden für die Parallelisierung nicht benötigte Teile abgeschalten. Dazu verfügt die Architektur über 4 Modi, in denen ein Kern betrieben werden kann. Der Modus 3 stellt einen Kern mit allen Funktionen dar, er wird auch als „Central Processing Unit“ (CePU) bezeichnet. Im Modus 2, auch „thread mode“ genannt, fehlt dem Kern die Hardware, welche Interrupts und Exceptions behandeln kann. Diese Aufgabe wird dann von der CePU übernommen. Im Modus 1, welcher auch als „linked mode“ bezeichnet wird, sind noch mehr Funktionen als im 2. Modus abgeschaltet. Der Kern kann selbst keine Befehle laden, sondern bekommt diese von der CePU. Im Modus 0 ist der betroffe Kern abgeschaltet [8].

Die Implementierung des ParaNut-Prozessors erfolgt in SystemC und VHDL, wobei VHDL nur für die Modellierung von zeitkritischen Modulen verwendet wird. Dadurch ist es möglich, den gleichen Code für die Synthese von Hardware zu nutzen oder daraus einen Simulator zu generieren. Dieser Simulator kann für die Entwicklung von Software und Hardware genutzt werden. Dazu verfügt er über die Funktion

VCD Dateien zu erzeugen, womit die internen Abläufe nachvollzogen werden können. Außerdem besitzt der Simulator eine Remote Bitbang Schnittstelle, die das Debuggen mit Hilfe von OpenOCD ermöglicht. Um OpenOCD nutzen zu können, verfügt der ParaNut über eines, dem „External Debug Support“ von RISC-V entsprechendes, Debug-Modul [8, 14]. Dessen Überarbeitung ist ein Ziel dieser Arbeit.

2.6.2 Kommunikation mit dem ParaNut

Um Programme für den ParaNut in den Speicher, des in Abbildung 2.2 gezeigten Systems, zu übertragen und Konsolenausgaben durchzuführen, wird der auf dem verwendeten Zynq FPGA-Board verbaute ARM-Prozessor verwendet [8].

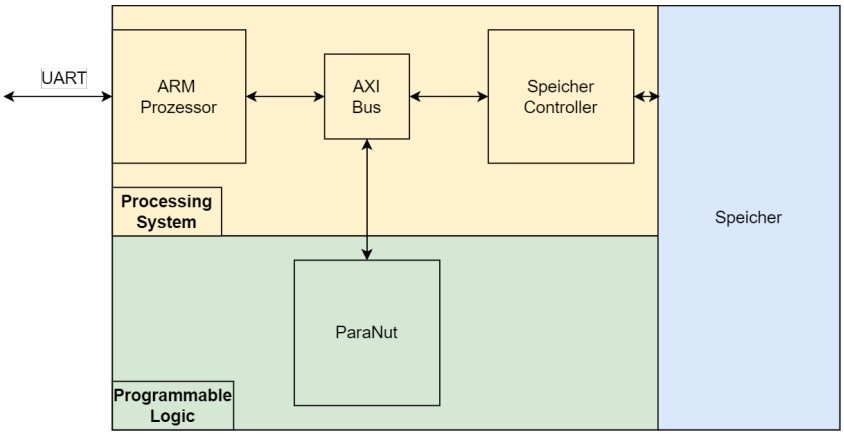

Abbildung 2.2: Blockschaltbild des Systems auf Zynq Hardware

Dazu werden die Daten mittels UART an den ARM-Chip gesendet, der diese dann in den Speicher schreibt. Auch bei der Konsolenausgaben des ParaNut werden die zu übertragenden Zeichen zuerst in den Speicher geschrieben, bevor diese mit Hilfe des ARM-Chips an das Host-System übertragen werden [8]. Ein Nachteil dieser Implementierung ist, dass für den ParaNut nur FPGA-Boards in Fragen kommen, die auch über einen zusätzlichen ARM-Prozessor verfügen. Dies begrenzt die Auswahl an verwendbaren FPGA-Boards.

Um diese Abhängigkeit aufzulösen, kann eine eigene Datenschnittstelle in der programmierbaren Logik implementiert werden. Dabei kann eine eigens entwickelte UART-Schnittstelle zu Einsatz kommen, die die Aufgaben des ARM-Chip übernimmt. Dabei wäre vorteilhaft, dass am Tool „pn-flash“, das für die Übertragung von Software auf den ParaNut zuständig ist, nur geringe Anpassungen durchgeführt werden müssen. So wäre die Verwendung der alten Schnittstelle weiterhin möglich.

2.7 RISC-V „External Debug Support“ Architektur

Ein Beispiel für den Aufbau einer „External Debug Support“ Architektur wird in Abbildung 2.3 gezeigt.

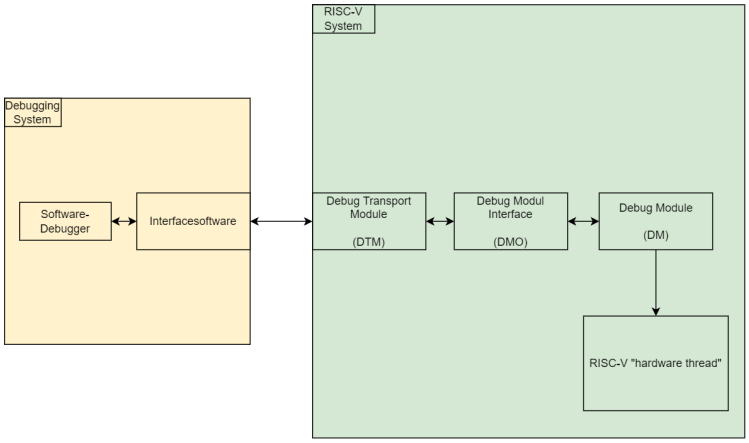

Abbildung 2.3: Strukturbild einer „External Debug Support“ Architektur

Der orange Bereich der Abbildung stellt das System dar, auf dem der Software-Debugger, im Fall des ParaNut GDB und die Interface-Software, im Fall des ParaNut OpenOCD, betrieben wird. Dieses System kann beispielsweise der Computer des Entwicklers sein. Dieses Teilsystem ist über einen Debug-Adapter, bei dem ParaNut-Prozessor ein JTAG- oder Remote Bitbang Adapter, mit dem RISC-V-System verbunden [11].

Die Komponenten der „External Debug Support“ Architektur im RISC-V-System aus Abbildung 2.3 bestehen, wie beim ParaNut-Prozessor, aus ein DM, ein DMI und ein DTM. Eine genauere Beschreibung erfolgt in den nachstehenden Abschnitten.

2.7.1 Debug Transport Modul

Das DTM ist dafür zuständig, den Zugriff auf das DM über eine Schnittstelle, wie zum Beispiel, JTAG bereitzustellen. Dabei ist es möglich, mehrere DTMs in ein System einzubauen, die verschiedene Schnittstellen zur Verfügung stellen, wobei immer nur ein DTM gleichzeitig verwendet werden kann [11].

In „External Debug Support“-Standard wird die Spezifikation für ein DTM mit JTAG Schnittstelle definiert. Dieses orientiert sich an der Definiton eines TAP im JTAG Standards, der den Zugriff auf eigens definierte JTAG-Register ermöglicht [11].

Dafür sind vier Register mit dazugehörigen Befehlen definiert. Das „IDCODE“ Register, das der Spezifikation aus dem JTAG Standard entspricht. Der Zugriff darauf erfolgt durch ein Schreiben von 0x01 in das Befehlsregister. Es ist außerdem das Register, dessen Befehl nach dem Zurücksetzen der TAP-Schnittstelle im Befehlsregister steht. Das „BYPASS“ Register wird ebenfalls wie im JTAG Standard implementiert und hat den Befehl 0x1F. Mit dem Register „DTM Status and Control“ (dtmcs) kann der Status des DTM abgefragt und das DMI zurückgesetzt werden. Es hat den Befehl 0x10. Das letzte Register ist das „Debug Module Interface Access“ (dmi) mit dem Befehl 0x11. Es erlaubt den Zugriff auf das DMI und somit auf die einzelnen DM's. In Abbildung 2.4 ist der Aufbau des Registers dargestellt [11].

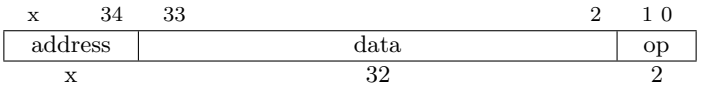

Abbildung 2.4: Aufbau von „dr“ im ParaNut (Adapiert von [11])

- `address`: Die Adresse des Registers auf dem DMI Bus. (Die Breite hängt von der Implementierung ab.)
- `data`: Die gelesenen oder zu schreibenden Daten
- `op`: Gibt an, ob Daten geschrieben oder gelesen werden sollen.
  - „01“: Zum Lesen von Daten
  - „10“: Zum Schreiben von Daten

[11]

2.7.1.1 Das DTM im ParaNut

In diesem Kapitel werden die Besonderheiten des DTM im ParaNut-Prozessors erläutert. Der Aufbau des DTM Moduls im ParaNut-Prozessor wird in Abbildung 2.5 dargestellt.

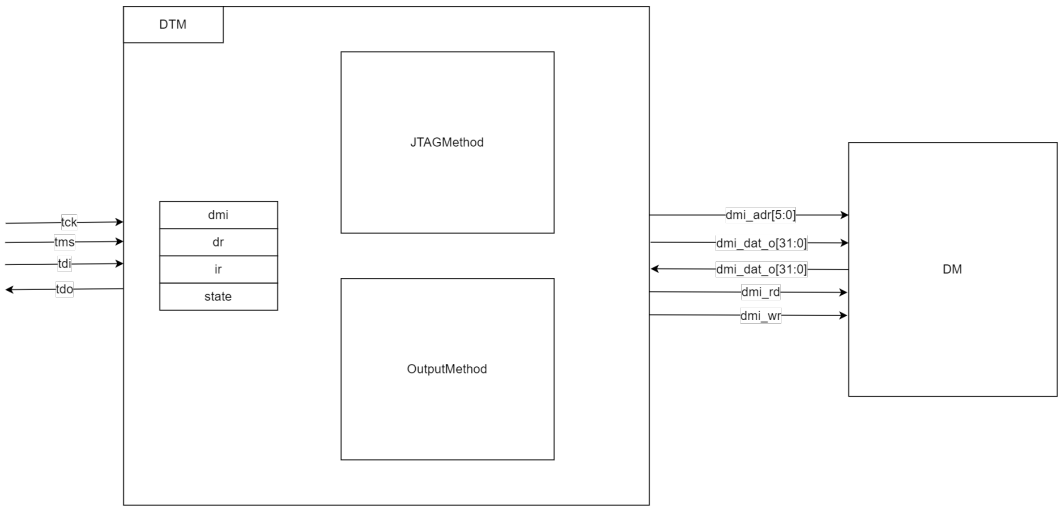

Abbildung 2.5: Aufbau des DTM im ParaNut

Das DTM im ParaNut-Prozessor besitzt die in Kapitel 2.7.1 beschreiben Befehle für das Befehlsregister (IR) und die benötigten Datenregister (DR). Dabei besitzt die Implementierung das Register „Debug Module Interface Access“ 6-Adressbits, womit das Datenregister eine Gesamtlänge von 40-Bit hat.

Das DTM in der Hardware

In der Hardware bietet das DTM eine JTAG-Schnittstelle, die über den PMOD JD der verwendeten Zybo FPGA-Boards verfügbar ist. Die Belegung des Steckers kann dem ParaNut-Manual entnommen werden [7].

Das DTM in der Simulation

In der Simulation werden die Signale der JTAG-Schnittstelle nicht verwendet. Hier wird mit Hilfe von Funktionen die Ansteuerung über die „remote_bitbang“ Schnittstelle des Simulators realisiert. Auch wird hierbei eine für das Remote Bitbang Protokoll optimierte Version des JTAG Zustandsautomaten verwendet.

2.7.2 Debug Module Interface

Das DMI ist ein Bus zwischen den im System vorhandenen DM's und DTM's, wobei die DTM's die Busmaster darstellen. Dabei kann der Bus über nur ein DM und DTM verfügen oder mehrere Teilnehmer unterstützen. Der Bus unterstützt Lese-

und Schreibzugriffe und erlaubt die Adressierung der DM's über bis zu 32 Adressleitungen. Wobei das erste und meist einzige DM an das untere Ende des Adressraums gelegt wird [11].

2.7.2.1 Das DMI im ParaNut

In diesem Kapitel werden die Besonderheiten des DMI im ParaNut-Prozessors erläutert.

Der DMI Bus im ParaNut-Prozessor hat nur zwei Busteilnehmer, da der Prozessor über ein DTM und ein DM verfügt. Die Leitungen des Busses sind in Abbildung 2.5 zwischen DM und DTM zu erkennen.

2.7.3 Debug Module

Ein DM setzt die Anweisungen, die es vom DMI erhält, in Anweisungen um, mit denen implementierte Debug-Hardware arbeiten kann. Dazu muss es die folgenden Anfragen behandeln können:

- Dem Debugger Informationen über die Implementierung der Hardware geben
- Das Anhalten und Starten von harts
- Auskunft darüber geben, welche harts angehalten sind
- Das Lesen und Schreiben von „General Purpose Register“ (GPR)
- Die Möglichkeit ab der ersten Assembler-Anweisung zu Debuggen

Punkte wie zum Beispiel das Anhalten von harts werden über das Schreiben oder Lesen von Registern des DM's realisiert. Der Zugriff auf diese Register erfolgt über den DMI Bus. Außerdem gibt es noch abstrakte Befehle (abstract commands), die unter anderem für den Zugriff auf GPR-Register verwendet werden und Register aus der Spezifikation benötigt. Zum einen eine oder mehrere „data“ Register, die Argumente für die abstrakten Befehle beinhalten. Das können zum Beispiel Speicheradressen, oder Daten sein. Zum anderen wird das „command“ Register benötigt, in das der zu bearbeitende abstrakte Befehl geschrieben wird [11].

2.7.3.1 Speicherzugriff

Zusätzlich zu diesen Anweisungen muss auch mindestens eine der drei, im Standard beschriebenen Möglichkeiten, zum Speicherzugriff geschaffen werden [11].

Die erste Möglichkeit ist der Zugriff mit Hilfe eines implementierten Programmspeichers. Dabei schreibt das Debug-System, über das DTM, Assembler Befehle, die den gewollten Speicherinhalt in ein GPR schreiben oder aus einem GPR lesen. Damit dies funktioniert, muss der abstrakte Befehl für Registerzugriff mit Programmspeicherregistern implementiert sein. Dafür wird der abstrakte Befehl zweimal aufgerufen. Einmal um die Befehle im Programmspeicher auszuführen und ein zweites Mal um Daten oder die Speicheradresse zwischen den Registern des DM's und den GPR-Registern auszutauschen [11]. Der Vorteil dieser Methode ist der geringe Aufwand für die Implementierung in Hardware, da dafür nur der abstrakte Befehl für den Registerzugriff erweitert werden muss. Jedoch muss das Debug-System die benötigten Assembler Befehle an das DM übertragen, was eine gewisse Zeit in Anspruch nimmt.

Die zweite Methode für den Speicherzugriff kann durch die Implementierung des im RISC-V Standard definierten, abstrakten Befehls für den Speicherzugriff realisiert werden. Hier muss das Debug-System nur die Speicheradresse, die Daten und den abstrakten Befehl an das DM übertragen. Der Zugriff wird dann eigenständig vom DM erledigt [11]. Ein Vorteil dieser Zugriffsart ist, dass der Zugriff ohne die Generierung von Assembler-Befehlen auf dem Debug-System auskommt, womit weniger Daten an das DM übertragen werden müssen. Auch können Teile der für den abstrakten Befehl für Registerzugriff wiederverwendet werden.

Die dritte Möglichkeit ist der Zugriff über den Systembus. Dafür muss im DM zusätzliche Hardware für den Zugriff auf den Systembus implementiert werden. Unter anderem sind zusätzliche Daten- und Adressregister für den Buszugriff nötig [11]. Ein Vorteil dieser Zugriffsart ist, dass der hart nicht für den Speicherzugriff benötigt wird. Jedoch ist dann sicherzustellen, dass die Daten mit denen aus der Sicht des hart übereinstimmen. Dafür ist bei dieser Art des Zugriffs das Debug-System verantwortlich.

Anzumerken ist, dass die Reihenfolge der Zugriffsarten, bei einer Verwendung von OpenOCD, erst ab der Version 0.12.0 der Software verwendet werden kann. Da die Option dafür erst mit dieser Version hinzugefügt wurde [19][20, S.159f].

2.7.3.2 Das DM im ParaNut

In diesem Kapitel werden die Besonderheiten des DM im ParaNut-Prozessors erläutert. Der Aufbau des Debug Moduls im ParaNut-Prozessor wird in Abbildung 2.6 dargestellt.

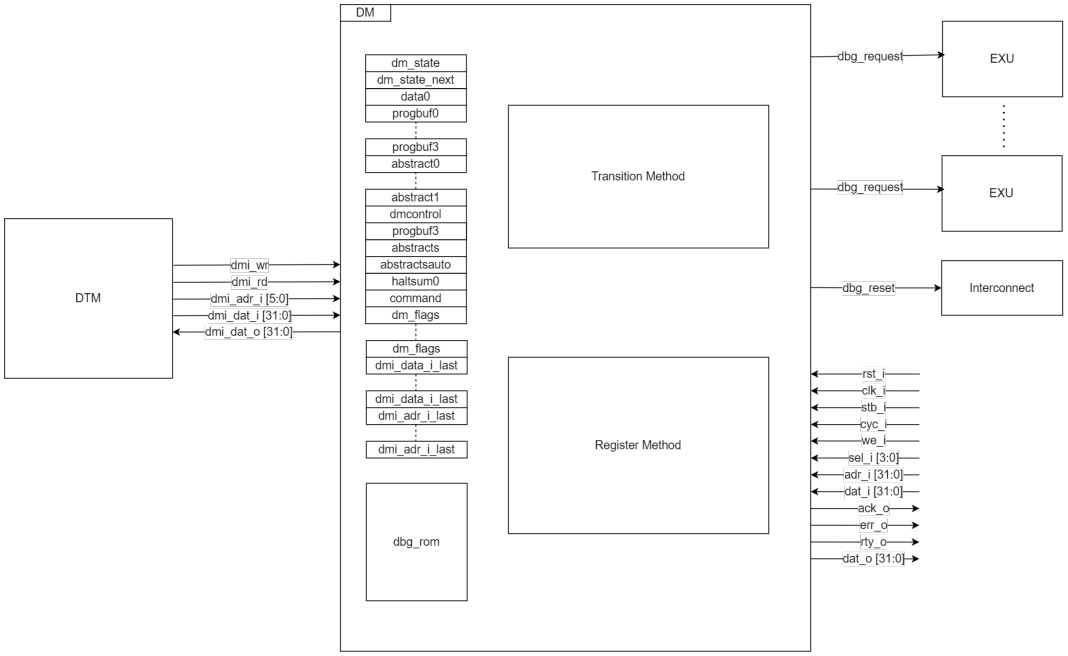

Abbildung 2.6: Aufbau des Debug Moduls

Register

Das DM des ParaNut-Prozessors besaß zum Begin der Arbeit ein „data“-Register und drei Programmspeicherregister.

In der Tabelle 2.2 werden die Register aufgelistet, die zusätzlich zu den Registern der „External Debug Support“ Spezifikation im Debug Module des ParaNut definiert wurden.

Name	Beschreibung
dm_state	Beinhaltet den aktuellen Zustand für den Zustandsautomaten, der für die Ausführung der abstrakten Befehle zuständig ist.
dm_state_next	Beinhaltet den nächsten Zustand für den Zustandsautomaten, der für die Ausführung der abstrakten Befehle zuständig ist.
abstract0 - abstract1	Beinhalteten Assembler Anweisungen, die für die Ausführung von abstrakten Befehlen im Zustandsautomaten generiert wurden
dm_flag	Beinhaltet die zum Debuggen benötigten Flags eine genauere Beschreibung ist in Abbildung 2.7 zu finden.
dmi_data_i_last	Diese Register beinhalten die Daten vom Port „dmi_dat_i“ des DMI Busses und dienen zur Synchronisation der Clockdomainen von DTM und DM
dmi_adr_i_last	Diese Register beinhalten die Daten vom Port „dmi_adr_i“ des DMI Busses und dienen zur Synchronisation der Clockdomainen von DTM und DM

Tabelle 2.2: Register des Debug Moduls

dm_flags:

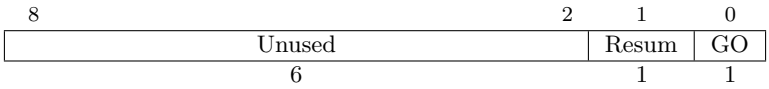

Abbildung 2.7: dm_flags Register

Für jeden Prozessorkern des ParaNut gibt es ein „dm_flags“ Register, wie in Abbildung 2.7 gezeigt, welche während der Ausführung von abstrakten Befehlen folgende Optionen beinhalten.

- Das „GO“ Flag des Registers löst die Ausführung der, durch den Zustandsautomaten, in den „abstract“ Registern gespeicherten Assembler Befehle auf dem betroffenen Prozessorkern aus.
- Das „resume“ Flag bringt den betroffen Prozessorkern dazu, das Debuggen zu beenden und mit der Ausführung des Programms fortzufahren.

Im DM des ParaNut ist es möglich, auf „data“, Programmspeicher, „abstract“ und „dm_flag“ Register über eine Wishbone Schnittstelle zuzugreifen. Dafür verfügen diese Register über eine Adresse im Adressraum des ParaNut.

Debug ROM

Der Debug ROM beinhaltet ein Assembler Programm, das geladen wird, wenn ein Prozessorkern des ParaNut vom DM angehalten wird. Er überprüft in einer Schleife, ob das „GO“ oder das „resume“ Flag des zum Prozessorkern gehörenden „dm_flags“ Registers gesetzt wurde.

Wenn das „GO“ Flag gesetzt wurde, springt er zum ersten „abstract“ Register des DMs und führt die darin enthaltenen Assembler Befehle aus. Ist das „resume“ Flag gesetzt, wird der Assembler Befehl „dret“ ausgeführt. Damit wird die Ausführung des Programms fortgesetzt.

Ein Zugriff auf den Inhalt des Debug ROM erfolgt über den Wishbone Bus des ParaNuts.

Abstrakter Befehl für den Registerzugriff

Für die Durchführung des abstrakten Befehls kommt ein Zustandsautomat zum Einsatz, der bei der Ausführung Assembler Anweisungen generiert, die er in die dafür vorgesehenen „abstract“ Register schreibt und darauf das „GO“-Flag im „dm_-flags“ Register des angesprochenen Prozessorkerns setzt. Um die Ausführung der Assembler Befehle auf dem Kern anzustoßen.

Speicherzugriff

Der ParaNut verwendet die in Kapitel 2.7.3.1 beschriebene erste Methode, die mit Hilfe der Programmspeicherregister des abstrakten Befehls für den Registerzugriff auf den Systemspeicher zugreift.

3 Kompatibilität mit aktueller Version von GDB

Die aktuell im ParaNut-Projekt verwendete Version von GDB wird um eine neue Version ergänzt. Die bisher verwendete Version von GDB aus der freedom-tools Toolchain[1] wurde seit dem Dezember 2020 nicht aktualisiert. Durch die fehlenden Aktualisierungen konnten seitdem in GDB behobenen Fehler nicht in die verwendete Version einfließen. Außerdem verfügt die bisher verwendete Version nicht über ein „text user interface“, das die Übersicht, vor allem bei Softwareprojekten wie zum Beispiel Linux, verbessert.

3.1 Kompatibilität mit aktueller Version von GDB herstellen

Dieses Kapitel behandelt die Schritte, die nötig sind, um eine aktuelle Version von GDB mit dem ParaNut kompatibel zu machen, die über ein „text user interface“ verfügt. Um dieses Ziel zu erreichen, waren die folgenden Teilschritte durchzuführen:

1. Installation einer RISC-V kompatiblen GDB Version
2. Test der neuen GDB Version

[1] https://github.com/sifive/freedom-tools/releases

3. Beheben von Fehlermeldungen beim Registerzugriff
4. Zugriff auf ParaNut spezifische „Control and Status Register“(CSR)

3.1.1 Installation einer RISC-V kompatiblen GDB Version

Da die bisher verwendete Version von GDB nicht mehr aktualisiert wird, wurde nach einer neuen Quelle für GDB gesucht. Dabei kamen mehrere mögliche Quellen infrage. Die Paketverwaltung des Betriebssystems oder das Bauen einer eigenen Version aus dem Quellcode.

Die Verwendung einer im Paketmanager verfügbaren Version ist dabei erstrebenswert, da die Installation und das Aktualisieren der Software durch das Tool übernommen werden. Die im ParaNut-Projekt verwendete Linux Distribution Debian in der Version 11, stellt zum aktuellen Zeitpunkt jedoch keine mit RISC-V Systemen kompatible Version von GDB in seinem Paketmanager bereit.

So musste eine aktuelle Version von GDB aus dem Quellcode gebaut werden. Dafür wird die „riscv-gnu-toolchain“ von riscv-collab verwendet, welche auf GitHub[2] zu finden ist. Hierbei handelt es sich wie bei der „freedom-toolchain“ um ein Projekt, welches eine komplette Toolchain für RISC-V zur Verfügung stellt. Somit ist es möglich, die bis dato verwendete Toolchain zu ersetzten, das jedoch nicht Teil dieser Arbeit ist. Bei der Installation ist es wichtig, dass das Paket `libncurses-dev` bereits auf dem System vorhanden ist. Dieses wird für das „text user interface“ benötigt, welches nicht durch eine Konfigurationsoption des Build-Systems, sondern alleine durch das Vorhandensein des Pakets aktiviert wird. Eine Anleitung für die Installation ist im Kapitel A.1 des Anhangs zu finden.

3.1.2 Test der neuen GDB Variante im Simulator

Um die neue Version des Debuggers zu testen, ist der ParaNut Simulator verwendet worden. Dafür wurde das im ParaNut-Projekt zur Verfügung gestellte Programm „hello_newlib“ verwendet, das ein von eins bis zehn durchnummeriertes „<x>.Hello World!“ ausgibt. Dafür muss zuerst das Makefile der Software modifiziert werden, indem zu den `CFLAGS`, den Parametern für den Compiler GCC, der Schalter `-g` hinzugefügt wird. Das bewirkt, dass Debug-Symbole erstellt werden, die GDB nutzen kann [4]. Das angepasste Programm kann nun gebaut werden. Bevor der Test durchgeführt wird, muss noch sichergestellt sein, dass die neue Version von GDB in der `PATH` Variable des Systems eingetragen wurde. Erst dann kann

[2] https://github.com/riscv-collab/riscv-gnu-toolchain

der Test gestartet werden. Dazu muss eine Debug-Session, wie im Kapitel A.2 des Anhangs beschrieben, gestartet werden. Im Anschluss lässt sich mit dem Kommando `(gdb) layout regs`, wie in Abbildung 3.1 ersichtlich, das „text user interface“, der neuen GDB Version, aktivieren, welches den Inhalt der Register und einen Codeausschnitt, zusätzlich zur GDB Kommandozeile, anzeigt.

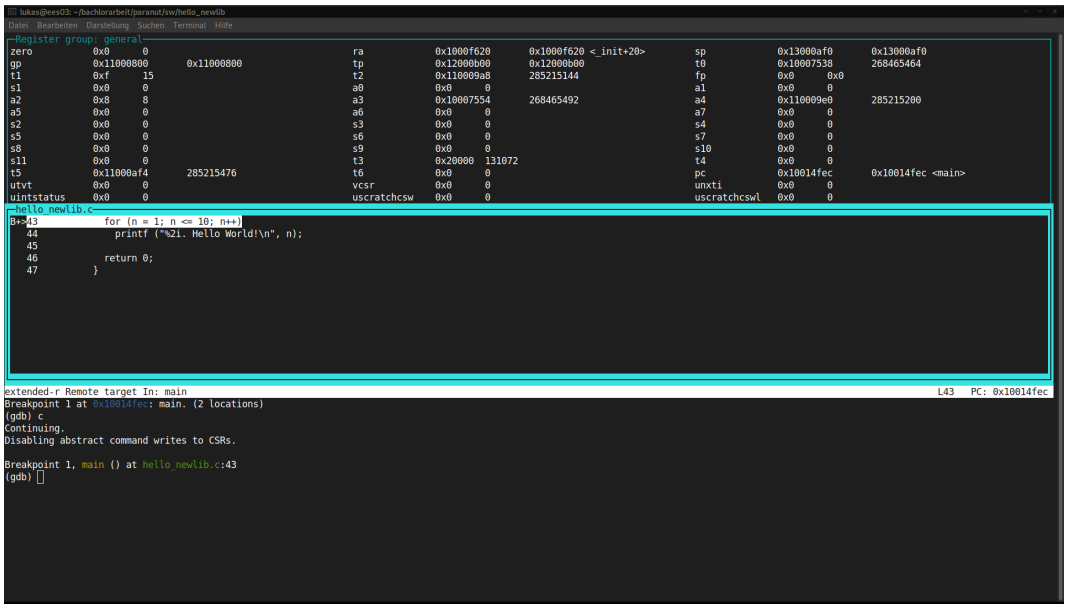

Abbildung 3.1: GDB mit „text user interface“

In dem so vorbereiteten GDB können übliche Befehle wie `break`, zum Erstellen eines Breakpoints, eingegeben und somit damit begonnen werden, Funktionen wie das Anhalten von Programmen, das Lesen bzw. Schreiben von Speicher und Registern zu testen. Diese Funktionen können ohne Probleme ausgeführt werden.

3.1.3 Zugriffsfehler auf CSR-Registern mit „text user interface“

GDB versucht beim Verwenden des „text user interface“s auf Control and Status Register zuzugreifen, die nicht vorhanden sind. Dies äussert sich in Warnungen wie dieser `... GetCSR: Read access to unkown CSR ...`, die in der Konsole des ParaNuts in großer Menge ausgegeben werden, wenn folgende Befehle in GDB eingegeben wurden. Bei diesen Befehlen handelt es sich um `step` und das Abbrechen von `continue`. Da zu diesem Zeitpunkt die Werte der Register, die im „text user interface“ zu sehen sind, aktualisiert werden. Dieses Problem ist bei der Durchführung der in Kapitel 3.1.2 beschriebenen Tests aufgefallen, wobei es durch die große Anzahl der Warnungen nicht mehr möglich war, die Konsolen-Ausgaben des ParaNuts nachzuvollziehen. Ein Beispiel dafür ist in der Abbildung 3.2 zu sehen.

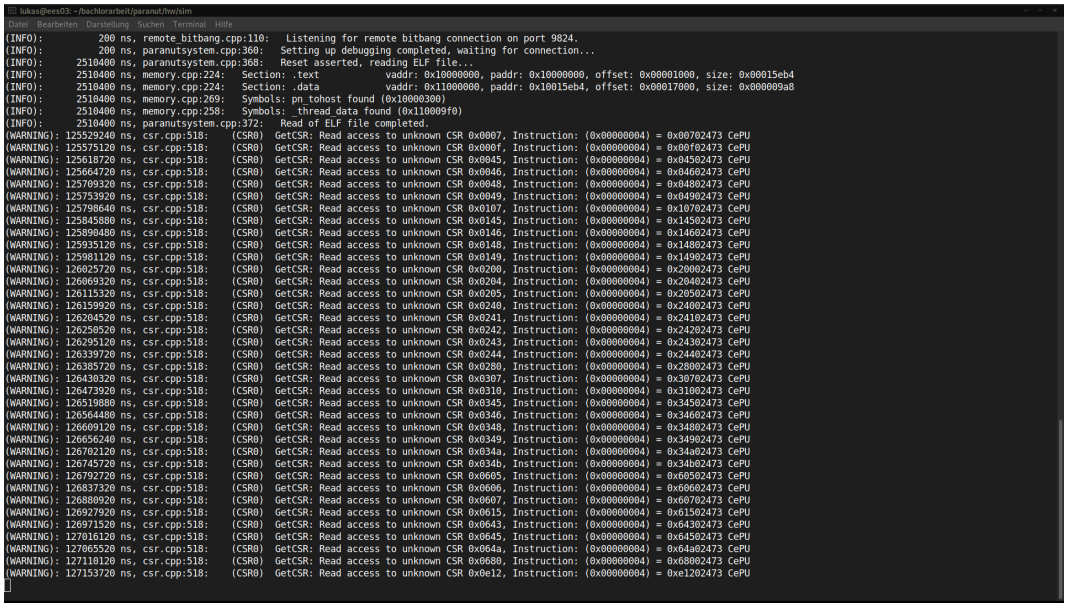

Abbildung 3.2: Fehlermeldungen in der ParaNut-Konsole

Durch weitere Untersuchung der in der Konsole auftretenden Fehlermeldungen, konnten mehrere Ursachen für die Meldungen ausfindig gemacht werden:

- Fehlende CSR-Register in ParaNut
- Falsche Konfiguration von GDB durch OpenOCD

In den folgenden Abschnitten wird beschrieben, wie die Ursachen für die Warnungen behoben wurden.

Fehlende CSR-Register im ParaNut

Der ParaNut-Prozessorarchitektur fehlen CSR-Register, die in den Spezifikationen für den privilegierten und den unprivilegierten Teil der RISC-V Befehlssatzarchitektur verlangt werden [22, 21]. Diese Diskrepanz fiel beim Vergleich der RISC-V Spezifikation mit dem Quellcode des CSR Moduls im ParaNut auf. In der Tabelle 3.1 werden die fehlenden Register mit ihren Adressen aufgelistet. Dabei ist anzumerken, dass Register des U-Mode in der SystemC Implementierung das Prefix „u“ erhalten. Dies ist nötig, um eine Kollision der Funktion „time“, in C, mit dem CSR-Register „time“ zu verhindern. Damit die Benennung bei den U-Mode-Registern in SystemC einheitlich bleibt, wurde entschieden, das Prefix „u“ bei allen User-Mode-Registern anzufügen. Damit ist eine einheitliche Namensgebung der User-Mode Register in SystemC sichergestellt.

Adresse	Registername
M-Mode Register	
0xF15	mconfigptr
0x306	mcounteren
0x310	mstatush
0xB09 - 0xB1F	mhpmcounter9 - mhpmcounter31
0xB89 - 0x9F	mhpmcounter9h - mhpmcounter31h
0x320	mcounterinhibit
0x323 - 0x33F	mhpmevent3 - mhpmevent31
0x30A	menvcfg
0x31A	menvcfgh
U-Mode Register	
0xC01	time
0xC02	instret
0xC03 - 0xC1F	hpmcounter3 - hpmcounter31
0xC81	timeh
0xC82	instreth
0xC83 - 0xC9F	hpmcounter3h - hpmcounter31h
S-Mode Registers	
0x106	scounteren
0x10A	senvcfg
0x5A8	scontext

Tabelle 3.1: Liste der fehlenden CSR-Register

Um die Register mit ihren Adressen in den ParaNut einzufügen, mussten diese zuerst der Enumeration aller „Control and Status Register“ hinzugefügt werden, die sich in der Datei `csr.h` im Ordner `hw/sysc` des ParaNut-Repositories befindet. So kann bei der Implementierung der Lese- und Schreibzugriffe der Name des CSR-Registers verwendet werden, statt die Adresse angeben zu müssen. Was die Lesbarkeit des SystemC-Codes erhöht.

Um einen Lesezugriff für ein neues Register einzubauen, musste dieses zum „switch-case“ der „CSRReadMethod“ hinzugefügt werden. In dem Listing 3.1 wird am Beispiel des Registers „mstatush“ gezeigt, wie das Einfügen eines Lesezugriffs funktioniert.

```
1                    switch(csradr){
2                        ⋮
3                        case mstatush:
4                            rdata = 0x0;
```

```
5                           break;
6                       ⋮
7                   }
```

Listing 3.1: Lesezugriff in der CSRReadMethod

Zum Hinzufügen eines Schreibzugriffs für ein Register, musste in das „switch-case“ in der „CSRWriteMethod“ ein neuer „case“ eingefügt werden. Hier wird erneut wie beim Lesezugriff am Beispiel des Registers „mstatush“ gezeigt, wie dies funktioniert. Was im Listing 3.2 zu sehen ist.

```
1                   switch(csradr){
2                       ⋮
3                       case mstatush:
4                           break;
5                       ⋮
6                   }
```

Listing 3.2: Schreibzugriff in der CSRReadMethod

Für jedes CSR-Register, das hinzugefügt wurde, musste außerdem noch beachtet werden, dass es nur dann verfügbar ist, wenn der ParaNut mit dem für das CSR benötigte „privilege-level“ konfiguriert ist. Durch die Konfigurierbarkeit des ParaNut ist es möglich, „privilege“ Modi hinzu- oder wegzuschalten. Um das sicherzustellen wurden CSR-Register des U- und des S-Modes mit Hilfe von Pre-Prozessor `#if` Anweisungen abschaltbar gemacht. Zudem wurden die in Tabelle 3.1 nur für den CePU-Kern der ParaNut-Architektur implementiert, da zu diesem Zeitpunkt nicht ersichtlich ist, ob diese CSR-Register in Zukunft von den CoPU-Kernen unterstützt werden sollen.

Durch dies Änderungen konnte die Anzahl der Fehlermeldungen in der ParaNut-Konsole reduziert werden.

Falsche Konfiguration von GDB durch OpenOCD

Beim Aufbau der Verbindung zwischen GDB und OpenOCD, wie in Kapitel A.2, nutzt OpenOCD, die Option, die Konfiguration des Zielsystems über die TCP/IP Verbindung an GDB zu übertragen [20]. Aufgrund der nach wie vor auftreten Fehlermeldungen in der ParaNut-Konsole nach dem Hinzufügen der fehlenden CSR-Register zur ParaNut-Architektur, musste davon ausgegangen werden, dass eine weitere Ursache für die Meldungen existiert. Mithilfe des GDB Befehls
`(gdb) info registers csr`, welcher die Werte aller CSR-Register des Systems

ausgibt, wurde es ersichtlich, dass GDB CSR-Register abfragt, die zu optionalen Erweiterungen der RISC-V Befehlssatzarchitektur gehören. Zum Beispiel wurde das Register „hip“ abgefragt, das zu optionalen Hypervisor-Erweiterung der Befehlssatzarchitektur gehört [21].

Aufgrund der Kenntnis des Verhalten von OpenOCD und GDB wurde versucht, eine eigene Systemkonfiguration in den Debugger zu laden, die keine CSR-Register beinhaltet, um herauszufinden, ob dies die Ursache der Meldungen in der Konsole ist. Um eine gültige Beschreibung für ein RISC-V Zielsystem zu erhalten, wurde mit Hilfe des Programms „Wireshark“, welches es ermöglicht Netzwerkpakte mitzuschreiben, die von OpenOCD und GDB gesendete Konfiguration, im XML Format, beim Verbindungsaufbau mitgeschrieben. Der Inhalt dieser Mitschrift ist im Kapitel A.3 des Anhangs abgebildet.

Aus dieser Konfiguration wurden für einen Test alle CSR-Register entfernt, um zu untersuchen, ob diese Konfiguration die Ursache ist. Dadurch traten die Fehlermeldungen bei der Durchführung des Tests nicht mehr auf. Dies deutet darauf hin, dass dies der Grund für die restlichen Fehlermeldungen in der ParaNut-Konsole war.

Somit muss eine eigene „Target-Description-Datei“ erzeugt werden, die bei der Verwendung von GDB in diesen geladen wird. Es muss beachtet werden, dass jeder „privilege-level“ aufgrund der unterschiedlichen CSR-Register eine eigene Datei benötigt. Bei der Erstellung der Dateien mussten nicht nur die nicht benötigten Register aus den Dateien entfernt, sondern auch Register hinzugefügt werden, da manche ParaNut vorhanden CSR-Register nicht in der Konfiguration von OpenOCD auftauchten. Beim Eintragen der zusätzlichen Register ist aufgefallen, dass die Nummern der Register in den XML-Dateien, einen Offset von 65 zu den Adressen der RISC-V Spezifikation haben. Beim Eintragen der zusätzlichen CSR-Register wurde dieser Offset addiert, um eine korrekte Konfiguration zu erhalten. Die so erstellten Konfigurationsdateien können nun mit Hilfe des GDB-Befehls
`(gdb) set tdesc filename <Pfad>`, nach dem Verbinden mit OpenOCD in den Debugger geladen werden. Dadurch wurden die Fehlermeldungen in der ParaNut-Konsole beseitigt.

Die zur Konfiguration hinzugefügten Register hatten beim Versuch, diese mit GDB auszulesen, keine Werte. Zum Beheben dieses Problems musste zusätzlich die Konfigurationsdateien für OpenOCD angepasst werden. Dazu wurde die Option „riscv expose_csrs“ verwendet, die CSR-Register zu den standardmäßig in OpenOCD vorhanden Registern hinzufügt [20]. Es müssen nach der Option die Adressen aus der RISC-V Spezifikation angegeben werden, wie in folgenden Beispiel anhand des CSR Registers „medeleg“, mit der Adresse 0x302 sichtbar wird.

```
riscv expose_csrs 770,...
```

Es ist anzumerken, dass die Adresse in der Konfigurationsdatei als Dezimalzahl angegeben werden muss. Dabei können weitere Adresse, getrennt durch ein Komma, hinzugefügt werden. Für jedes „privileg-level“ wurde eine solche Konfigurationsoption den OpenOCD Konfigurationsdatei, für Simulation und Hardware, eingebaut.

Somit ist es nun möglich, alle in der RISC-V Befehlssatzarchitektur definierten CSR Register, in GDB zu verwenden.

3.1.4 Zugriff auf CSR-Register der ParaNut-Architektur

Die Architektur spezifischen CSR-Register des ParaNut können nicht gelesen werden. Da diese Register nicht in der RISC-V Befehlssatzarchitektur definiert sind, können OpenOCD und GDB die Register nicht erkennen. Für diesen Fall bietet OpenOCD die Konfigurationsoption `riscv expose_csr` an, womit, wie bereits in Kapitel 3.1.3 beschrieben, zusätzliche CSR-Register zur Konfiguration des Systems hinzugefügt werden können [20]. Um die ParaNut spezifischen Register ansprechen zu können, werden diese analog zu Kapitel 3.1.3 den OpenOCD Konfigurationsdateien hinzugefügt. Außerdem müssen sie in die „Target-Description“ Dateien von GDB eingefügt werden, um im Debugger aufzutauchen. Somit ist es nun möglich ParaNut spezifische CSR-Register in GDB zu lesen.

3.2 Kompatibilität mit GDB aus der ursprünglichen Toolchain sicherstellen

Es sollte sichergestellt werden, dass die in Kapitel 3.1 durchgeführten Änderungen für das „text user interface“, der aktuellen GDB Version, nicht die Kompatibilität mit der in der „freedom-tools“-Toolchain[3] enthaltenen Version von GDB zunichtegemacht haben. Durch das Durchführen der im Kapitel 3.1.2 beschriebenen Tests, mit der ursprünglich im ParaNut-Projekt verwendeten Version von GDB, wurde überprüft, ob diese noch immer wie erwartet funktioniert. Hierbei konnten keine Inkompatibilitäten festgestellt werden. Lediglich der Befehl zum Aktivieren des „tui“ konnte, wie erwartet, nicht in dieser Version von GDB ausgeführt werden.

[3] https://github.com/sifive/freedom-tools/releases

4 Beschleunigter Speicherzugriff des DM's

Der Speicherzugriff für das Debug-Modul wird durch eine weitere Zugriffsart beschleunigt. Für die aktuell verwendete Art des Speicherzugriffs über den Programmspeicher das Übertragen von Assembler-Befehlen und das zweifache Ausführen des abstrakten Registerzugriffsbefehls benötigen, siehe Kapitel 2.7.3.1. Andere Zugriffsarten benötigen solche Vorgänge nicht [11], sodass sie in der Theorie schneller sind als der bisher verwendete Speicherzugriff. Ein schneller Zugriff erlaubt es, den Inhalt von Variablen, wie Strings, schneller zu modifizieren, wodurch beim Debuggen Zeit gespart werden kann.

4.1 Auswahl der Speicherzugriffsart

Es musste die Entscheidung getroffen werden, welche weitere Speicherzugriffsart im DM des ParaNut implementiert wird, um den Speicherzugriff des DMs zu beschleunigen. Dabei wurde der abstrakte Speicherzugriffsbefehl implementiert, da bei diesem auf die bereits bestehende Hardware für den abstrakten Befehl für den Registerzugriff zurückgegriffen werden kann. Aufgrund der Reduzierung von Übertragungen über die JTAG-Schnittstelle des DTM's ist dies in der Theorie auch schneller. Das wird im folgenden Kapitel 4.2 näher erläutert.

4.2 Vergleich der Speicherzugriffsarten

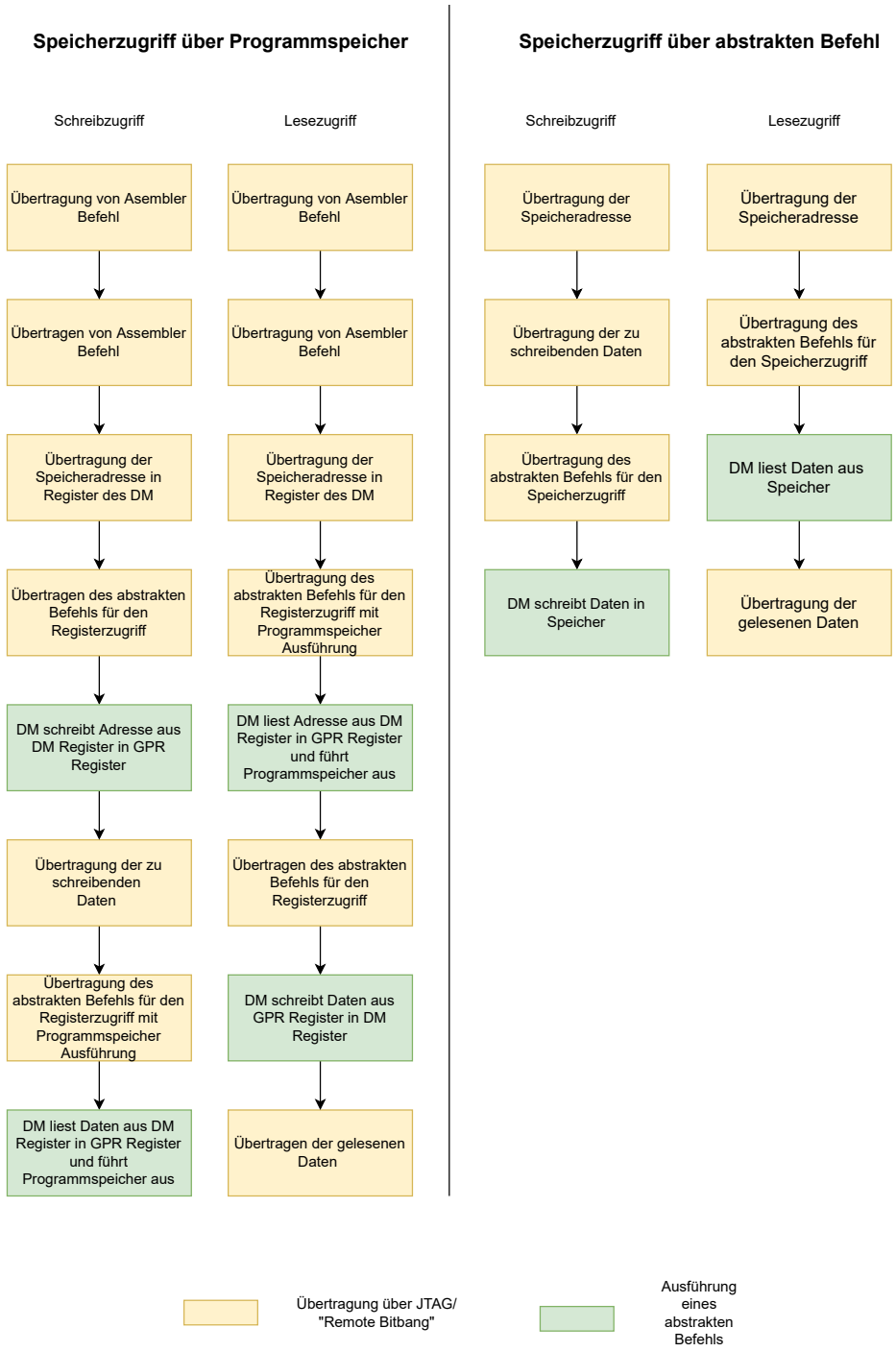

Abbildung 4.1: Vergleich der Speicherzugriffsarten

Wie in Abbildung 4.1 zu erkennen ist, benötigen Lese- und Schreibzugriffe mit dem abstrakten Befehl nur vier statt sechs Zugriffe über die JTAG-Schnittstelle beziehungsweise „Remote Bitbang"-Schnittstelle des DTM auf die Register des DM. Diese Zugriffe sind langsam, da die Daten bitweise über den TAP-Controller, der in Kapitel 2.2.5 beschrieben wurde, in die Register übertragen werden müssen. Außerdem ist in der Abbildung 4.1 auch der doppelte Aufruf des abstrakten Befehls für den Registerzugriff bei dem Zugriff über den Programmspeicher zu erkennen, wohingegen der Zugriff über den abstrakten Befehl für den Speicher nur einmal aufgerufen werden muss.

4.3 Implementieren des Abstrakten Speicherzugriffs

Es wird der abstrakte Befehl für den Speicherzugriff implementiert. Dafür ist es nötig, den Zustandsautomat im Debug Module um weitere Zustände zu erweitern. In Abbildung 4.2 sind die hinzugefügten Befehle des abstrakten Speicherzugriffs orange eingefärbt. Dabei funktioniert der Zustandsautomat wie folgt. Wenn ein abstrakter Befehl an das DM übertragen wird und kein Fehlerzustand im DM anliegt, ist die Bedingung „Ü1" erfüllt und ein Wechsel von „Idle" nach „CMD" wird durchgeführt. Dort wird entschieden, ob ein abstrakter Befehlt für den Registerzugriff auf ein GPR-Register mit 32-Bit Breite anliegt und der Prozessorkern angehalten ist. Dann ist die Bedingung „Ü3" erfüllt und die Ausführung des Befehls beginnt. Wenn es sich um einen abstrakten Befehl für den Speicherzugriff handelt und der Prozessorkern angehalten ist, ist die Bedingung „Ü4" erfüllt und die Ausführung beginnt. Wenn keine dieser Bedingungen zutrifft ist, „Ü2" erfüllt und es wird zurück in den Zustand „Idle" gewechselt, wobei ebenfalls ein Fehlerzustand in DM ausgelöst wird.

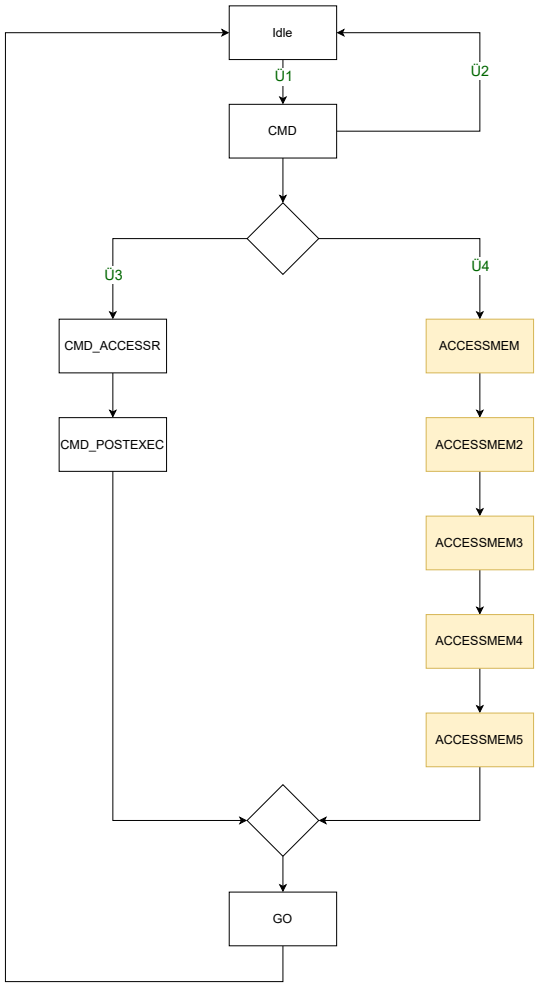

Abbildung 4.2: Zustandsautomat des Debug Modules

In den Tabellen 4.2 und 4.1 sind die Assembler Befehle aufgelistet, die bei der Ausführung eines abstrakten Befehls für den Speicherzugriff, analog zum abstrakten Registerzugriffsbefehl, durch die Zustände im Zustandsautomat in die dafür bestimmten „abstract“ Register schreibt. Die Auswahl, ob die Befehle zum Schreiben oder zum Lesen von Speicher verwendet werden, wird dem abstrakten Befehl entnommen. Dabei sind in Tabelle 4.1 die Befehle für das Lesen von Speicher dargestellt und in Tabelle 4.1 die für das Schreiben.

Zustand	Assembler Befehl	Beschreibung
ACCESSMEM	`lw x31, data1(x0)`	Lädt die Adresse in, die die Daten geschrieben werden sollen aus dem Register „data1“, des DMs über dessen Wishbone Schnittstelle, in das Register „x31“ des Prozessors.
ACCESSMEM2	`l[b/h/w] x30, data0(x0)`	Lädt die zu schreibenden Daten aus dem Register „data0“, des DM über dessen Wishbone Schnittstelle, und schreibt diese in das Register „x30“ des Prozessor
ACCESSMEM3	`s[b/h/w] x30, 0(x31)`	Schreibt die Daten die aus dem Register „x30“ an die Speicheradresse, die in „x31“ hinterlegt ist.
ACCESSMEM4	`sw zero, INCRE(zero)`	Setzt das Increment Flag im „dm_flags“ zurück
ACCESSMEM5	`ebreak`	Hält den hart nach der Ausführung der Assembler Befehle wieder an

Tabelle 4.1: Assembler Befehle für den Schreibzugriff

Nummer	Assembler Befehl	Beschreibung
ACCESSMEM	`lw x31, data1(x0)`	Lädt die Adresse in, die die Daten geschrieben werden sollen aus dem Register „data1“, des DMs über dessen Wishbone Schnittstelle, in das Register „x31“ des Prozessors.
ACCESSMEM2	`l[b/h/w] x31, data0(x31)`	Lädt die zu lesenden Daten von der Speicheradresse, die in „x31“ definiert ist in das Register „x31 “
ACCESSMEM3	`s[b/h/w] x31, 0(data0)`	Schreibt die Daten aus dem „x31“ Register in das „data0“ Register des DMs über dessen Wishbone Schnittstelle
ACCESSMEM4	`sw zero, INCRE(zero)`	Setzt das Increment Flag im „dm_flags“ zurück
ACCESSMEM5	`ebreak`	Hält den hart nach der Ausführung der Assembler Befehle wieder an

Tabelle 4.2: Assembler Befehle für den Lesezugriff

Da bei der Ausführung des abstrakten Befehls für den Registerzugriff nur zwei Assembler Befehle generiert werden, reichten zwei „abstract“ Register aus. Bei der Durchführung des abstrakten Speicherzugriffsbefehls werden fünf Assembler Befehle generiert, das eine Erhöhung der Anzahl an „abstract“-Register auf mindestens fünf erforderlich machte. Außerdem wurde ein weiteres „data“-Register erforderlich, da in der „External Debug Support“ Spezifikation gefordert wird, dass es separate „data“-Register für die Speicheradresse und Daten gibt [11].

Bei einem Test des abstrakten Speicherzugriffsbefehls konnte beobachtet werden, dass es zu Fehlermeldungen beim Zugriff auf Variablen kommt, die kleiner als 32-Bit breit sind, wenn der abstrakte Speicherzugriffsbefehl, wie der abstrakte Befehl für den Registerzugriff, nur den Zugriff auf 32-Bit breite Datenworte erlaubt. Zum Beheben dieses Problems musste der abstrakte Befehl für den Speicherzugriff um eine Möglichkeit für den Zugriff auf 8- und 16-Bit breite Datenworte, erweitert werden.

Der abstrakte Befehl für den Speicherzugriff bietet die Möglichkeit, die Speicheradresse nach dem Zugriff automatisch zu inkrementieren. Dazu muss die Speicheradresse nach einem Zugriff um die Anzahl an Bits erhöht werden, die der Breite

des gelesenem Datenworts entspricht [11]. Zum Beispiel muss die Adresse nach dem Lesen eines 32-Bit breiten Datenworts um 4 Byte inkrementiert werden. Hierzu ist es nötig, dass das DM informiert wird, wenn der Speicherzugriff abgeschlossen ist und um wie viele Bytes die Speicheradresse erhöht werden muss. Dazu wurden zwei weitere Einträge zu „dm_flags“ hinzugefügt, wie in Abbildung 4.3 zu sehen ist.

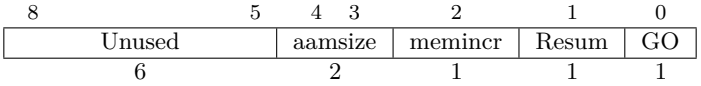

Abbildung 4.3: dm_flags Register

Die Bedeutung der neu hinzugefügten „Flags“ soll im Folgenden erklärt werden:

- `memincr`: Gibt an, dass die Speicheradresse nach dem Zugriff inkrementiert wird.
- `aamsize`: Gibt an, um wie viele Byte die Speicheradresse inkrementiert wird. Er entspricht dem Wert von „aamsize“ aus dem Abstrakten Befehl für den Speicherzugriff [11].
  - `0`: Inkrementieren um 1 Byte
  - `1`: Inkrementieren um 2 Byte
  - `2`: Inkrementieren um 4 Byte

Für das automatische Inkrementieren der Speicheradresse, wird zusätzlich ein Register zu der Wishbone Schnittstelle des Debug Moduls hinzugefügt. Wenn dieses mit dem Wert „0“ beschreiben wird, werden die von Zustandsautomaten in das „dm_-flags“ Register des Prozessorkerns geschriebenen Flags „memincr“ und „aamsize“ ausgewertet und wenn nötig die Speicheradresse im „data“ Register inkrementiert. Auch werden die Einträge im „dm_flags“ zurückgesetzt. Das Schreiben des Registers geschieht bei der Ausführung des im Zustand „ACCESSMEM4“ geschriebenen Assembler Befehls.

Somit ist der abstrakte Befehl für den Speicherzugriff in das Debug Module des ParaNut implementiert.

4.4 Kompatibilität mit OpenOCD sicherstellen

Um die neu implementierte Speicherzugriffsart zu nutzen, muss sichergestellt sein, dass die restlichen Komponenten der Debug-Infrastruktur die Speicherzugriffsart auch unterstützen. Die Analyse der Komponenten, die in der Abbildung 2.3 zu sehen

sind, zeigt, dass nur eine für die Funktion des neuen abstrakten Befehls modifiziert werden muss. Dabei handelt es sich um die Interface-Software, OpenOCD. Diese muss die Fähigkeit haben, bei Anfragen des Debuggers einen abstrakten Befehl für den Speicherzugriff, anstatt den Program-Buffer zu verwenden. Dazu muss es eine Konfigurationsoption für die Interface-Software geben. Diese Option wurde in der Version 0.12.0 hinzugefügt [19, 20].

Dafür wurde analog zum Kapitel 3.1.1 versucht, die Version 0.12.0 über den Paketmanager des verwendeten Betriebssystems zu beziehen. Der Paketmanager der im ParaNut-Projekt verwendeten Linux Distribution Debian, in der Version 11, verfügt zum aktuellen Zeitpunkt nicht über OpenOCD in der Version 0.12.0.

Somit muss die Version 0.12.0 aus dem Quellcode kompiliert werden. Die durchzuführenden Schritte werden im Kapitel A.4, des Anhangs, beschrieben.

Um nun die Art des Speicherzugriffs umzustellen, mussten die Konfigurationsdateien der Simulation und der Hardware von OpenOCD angepasst werden. Dazu wurde die Konfigurationsoption `riscv set_mem_access abstract progbuf` in die Dateien geschrieben. Diese Option bewirkt, dass der Speicherzugriff zuerst über den abstrakten Befehl versucht wird und nur wenn dies fehlschlägt, der Speicherzugriff über den Program-Buffer verwendet wird [20].

Mit der so angepassten OpenOCD Version kann der Speicherzugriff über den abstrakten Befehl verwendet werden.

5 Testinfrastruktur der Debug-Infrastruktur

Die Testinfrastruktur der Debug-Infrastruktur soll verbessert werden. Da diese bisher nur aus den Testbenches für das Debug Module und das Debug Transport Module besteht. Dafür wird ein Testkonzept für die komplette Debug-Infrastruktur erstellt. Weiterhin werden die bereits bestehenden Hardware Testbenches erweitert. Zudem wird eine Möglichkeit geschaffen, die Infrastruktur als Komplettsystem zu testen. So kann sichergestellt werden, dass Fehler in der Debug-Infrastruktur schnell gefunden und behoben werden können, um die Funktion dieser zu garantieren.

5.1 Testkonzept für die Debug-Infrastruktur

Es wurde ein Testkonzept erstellt, dass die Debug-Infrastruktur des ParaNut-Prozessors in der Simulation und in der Hardware mit verschiedenen Konfigurationsoptionen des Prozessors überprüft.

In der Simulation kommen unter anderem die bereits vorhandenen Testbenches für die Hardwaremodule zum Einsatz. Diese wurden um weitere Tests erweitert, um die Funktionen der Hardwaremodule umfänglicher zu überprüfen. Auch wurde eine Möglichkeit entwickelt, die es erlaubt, das Komplettsystem der Debug-Infrastruktur, die in Abbildung 2.3 dargestellt ist, als ganzes zu überprüfen. Dieser Test läuft nach dem Start automatisch ab und liefert als Ergebnis zurück, ob beim Test Probleme aufgetreten sind. So ist es möglich Fehler, die nur bei einer Interaktion der einzelnen Komponenten auftreten, zu identifizieren. Dazu werden GDB „command files“ und Shell-Skripte verwendet.

Für Tests auf der Hardware wird die Methode zum Überprüfen des Komplettsystems aus der Simulation angepasst, sodass dieser auch bei der Verwendung eines FPGA-Boards zur Anwendung kommen kann. Dieser Teil des Konzepts konnte aufgrund von Zeitmangel nicht in dieser Arbeit umgesetzt werden, soll aber in Zukunft noch entwickelt werden.

Bei den Tests muss außerdem auf die verschiedenen Konfigurationsoptionen des Prozessors eingegangen werden. So muss bei den Tests berücksichtigt werden, dass sich die Anzahl der „Control and Status“-Register mit dem Wechsel des „privilege-level“ ändert. Dies kommt bei den Tests des Gesamtsystems zu tragen und wird durch einen Schalter zur Auswahl des „privilege-level“ mit in den Test einbezogen. Außerdem ist es nötigt, die zwei im ParaNut-Prozessor verfügbaren Speicherzugriffsart einzeln testen zu können. Dies konnte aufgrund von fehlender Zeit nicht mehr Implementiert werden. Dies soll in Zukunft analog zu dem Schalter für „privilege-level“ umgesetzt werden. Auch müssen in Zukunft hinzukommende Optionen, die zum Beispiel Speicherzugriffsarten am Debug-Modul abschalten, um Fläche auf dem FPGA zu sparen, einen Test erhalten.

5.2 Erweiterte Testbenches

Um die Funktion von DM und DTM zu überprüfen, sind die SystemC-Testbenches beider Module überarbeitet worden, da bei der ersten Durchsicht der Dateien auffiel, dass die Tests nicht die kompletten Hardwaremodule überprüften. In den folgenden Kapiteln wird darauf genauer eingegangen.

5.2.1 DM-Testbench

Um die Hardware des Debug Moduls besser zu testen, mussten die abstrakten Befehle, die in der Version des ParaNut implementiert sind, in der Testbench hinzugefügt werden, da bisher nur eine Überprüfung der Zugriffe auf die internen Register des DM stattfand. Zum einen wurden Zugriffe über die DMI-Schnittstelle, die einen Zugriff über das DTM darstellen, getestet. Zum anderen ein Buszugriff über das implementierte Wishbone Interface des DM's. Die im DM vorhandene Hardware für abstrakte Befehle wurden nicht überprüft. Hierfür fand eine Analyse des Aufbaus der abstrakten Befehle statt, um zu ermitteln, wie die Tests aufzubauen sind. Dabei wurde ermittelt, dass für den abstrakten Befehl für den Registerzugriff ein kompletter Zugriff auf ein Register getestet werden kann. Bei dem abstrakten Befehl für den Speicherzugriff wurden Testfälle entwickelt, die das Schreiben und Lesen bei allen Datenwortbreiten überprüft. Außerdem sind Testfälle erstellt worden, die überprüfen, ob ein fehlerhafter abstrakter Befehl auch einen Fehlerzustand im Debug Module auslöst. Weitere Informationen zu den erstellten Testfällen in der Testbench des DM sind im Kapitel A.5, des Anhangs, zu finden.

5.2.2 DTM-Testbench

Um beide Implementierungen des DTM besser zu testen, musste die Testbench für die Hardware erweitert und eine neue Testbench für die Simulation erstellt werden.

Testbench für die Hardware

Um die Tests für die Hardware Version des DTM zu erweitern wurde überprüft, ob ein Zugriffsbefehl für ein Register des DM richtig von dem DTM bearbeitet wird, da die Testbench bisher nur die OpenOCD Testsequenz überprüft. Dazu wird in das Register „Debug Module Interface Access“ ein gültiger Befehl geschrieben und überprüft, ob diese richtig an die DMI Bus Schnittstelle des DTM angelegt werden. Eine tiefere Beschreibung des Tests kann dem Kapitel A.7, des Anhangs entnommen werden.

Testbench für die Simulation

Es wurde eine neue Testbench für die Simulationsversion des DTM erstellt, da es bisher keine Testbench gab, die die Remote Bitbang Version des DTM überprüfte.

Dazu wurde die Testbench für die Hardware Implementierung kopiert und das Makefile für die Simulation angepasst. Um diese für die Simulationsversion des DTM zu verwenden, musste der Generator für das `TCK`-Signal angepasst werden. Damit dieser mit den in Kapitel 2.7.1.1 erwähnten Funktionen für die Ansteuerung des JTAG Zustandsautomaten funktionierten. Hierfür wurde mit Hilfe einer in den Quellcode, des DTM, eingebauten Funktion für die Fehlersuche das Verhalten des `TCK`-Signals beobachtet. Dies wurde mit Hilfe einer zusätzlichen Funktion zum Generieren des `TCK`-Signals nachgebildet. Eine detailiertere Beschreibung des Vorgehens kann dem Kapitel A.8 des Anhangs entnommen werden.

5.3 Testskripten für die Debug-Infrastruktur

Für den Test der kompletten Debug-Infrastruktur wurden GDB „command files“ verwendet. Da ein Test der Hardware von DM und DTM mit Hilfe von Testbenches nur die Hardware der einzelnen Module testet, jedoch nicht das Komplettsystem. Wodurch es passieren konnte, dass die einzelnen Komponenten funktionieren, es aber bei der Interaktion der Module zu Fehlern kommt. Das dafür erstellt „command file“ testet die Funktionen, die von der Debug-Infrastruktur unterstützt werden.

- Erstellen eines Haltepunkts
- Lesen/Schreiben von (CSR-)Registern
- Lesen/Schreiben von Speicher
- Die Fortsetzung der Ausführung

Um die erstellte Datei für einen automatisierten Test zu verwenden, wurde ein Shell-Skript geschrieben, das den Simulator und das für den Test verwendete Programm „hello_newlib“ kompiliert, die im „command file“ enthaltenen Befehle auf dem Simulator ausführt und die Ergebnisse mit der Erwartung vergleicht.

Die Software „hello_newlib“ wurde für die Verwendung bei den Tests modifiziert, um das Lesen und Schreiben von Speicher zu testen. Hierzu ist der auszugebende String in eine eigene Variable ausgelagert worden.

Dem Skript kann mit einer Konfigurationsoption der zu testende „privilege-level“ übergeben werden. Auch kann der Pfad zu einer anderen Version von OpenOCD angegeben werden, wenn die Version der `PATH`-Variable des Systems nicht für den Test verwendet werden soll.

Für die Ausführung des Tests werden im Shell-Skript die im Kapitel A.2 des Anhangs, beschriebenen Befehle benutzt. Wobei der Befehl für den Start von GDB

modifiziert wurde, sodass der Debugger die Befehle aus dem „command file“ bearbeite und die Ergebnisse in eine Datei speichert.

Die Ergebnisse werden mit Hilfe des Programms `diff` mit der Erwartung verglichen. Dabei musste darauf geachtet werden, dass der Inhalt von CSR-Registern wie `cycle`, die beim Test abgefragt werden, nicht in den Vergleich einbezogen werden, da diese bei jeder Ausführung des Skripts einen anderen Wert haben.

Somit kann in Simulation die komplette Debug-Infrastruktur getestet werden.

5.4 Peer-Test der Änderungen an der Debug-Infrastruktur

Um weitere Fehler in der veränderten Debug-Infrastruktur zu finden, wurde ein Peer-Test erstellt, der vom ParaNut-Team durchgeführt wurde. So konnten Fehler ausfindig gemacht werden, die bei eigenen Tests nicht bemerkt wurden. Eine korrigierte Version der Anleitung des Peer-Tests ist in Kapitel A.10 des Anhangs, hinterlegt.

6 Austausch der Kommunikationsschnittstelle des ParaNuts

Wie bereits in Kapitel 2.6.2 beschrieben, wird zum Schreiben von Programmen in den Speicher der in den FPGA-Boards von Xilinx integrierte ARM-Prozessor benötigt, was das ParaNut-Projekt von diesen FPGA-Boards abhängig macht. Um diese Abhängig aufzulösen, wurde eine eigene UART Schnittstelle in den ParaNut integrierte. Die Veränderungen am Gesamtsystems des ParaNut werden beim Vergleich der Abbildungen 2.2 und 6.1 sichtbar.

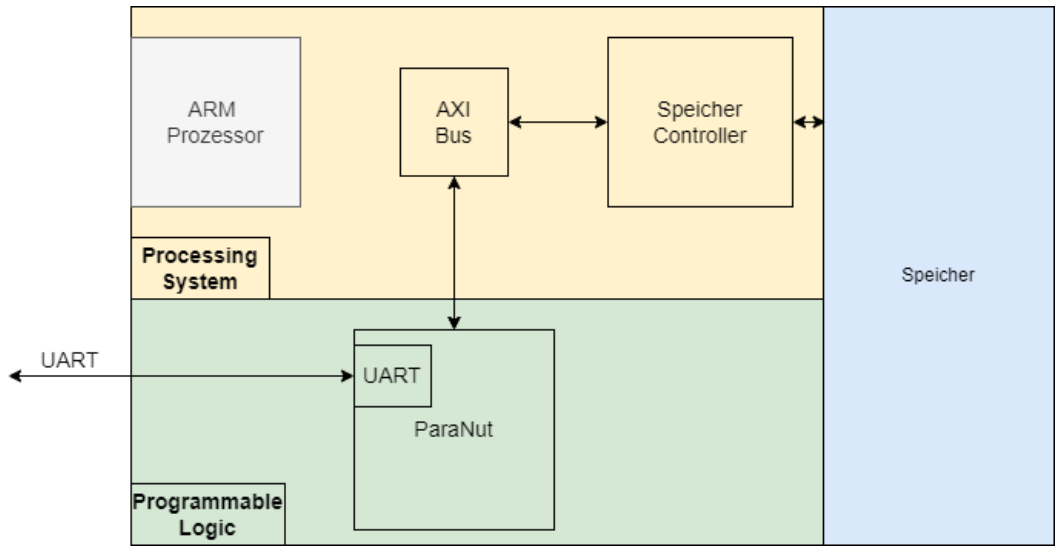

Abbildung 6.1: ParaNut System mit eigenem UART

Der eingebaute UART kann nicht nur für die Übertragung von Programme auf den ParaNut-Prozessor und die Konsolenausgabe verwendet werden, sondern auch als Standard Kommunikationsschnittstelle, für zum Beispiel externe Hardware, fungieren. Diese Funktion wird von Elias Schuler zum Ansteuern eines Bluetooth Moduls in seiner Bachelorarbeit verwendet, weshalb das Konvertieren eines bestehenden Hardwaremoduls in Zusammenarbeit mit ihm durchgeführt wurde [13]. Die Entscheidung ein bereits bestehendes Modul mit dem ParaNut kompatibel zu machen, anstatt die Implementierung selbst zu übernehmen, wurde aufgrund der bei einer Recherche gefundenen Anzahl an bereits implementierten UART Modulen getroffen. Bei der Auswahl des UART-Moduls wurde außerdem darauf geachtet, dass dieses über einen Linux Treiber verfügt, um das Modul auch nachdem die Kompatibilität von Linux mit dem ParaNut-Prozessor hergestellt ist, dort zu verwenden.

6.1 ParaNut kompatibles UART-Moduls

6.1.1 Auswahl eines UART-Moduls

Bei der Auswahl eines UART-Moduls musste darauf geachtet werden, dass bestimmte Voraussetzungen erfüllt werden. Zum einen sollte das Modul über eine Schnittstelle für einen Wishbone Bus verfügen, damit es möglich ist, das erstellte Hardwaremodul an den Bus des ParaNut-Prozessors anzuschließen. Darüber hinaus sollte das Modul in SystemC modelliert sein, dass es möglich ist, das Modul mit den bereits vorhandenen Tool des ParaNut-Projekts zu synthetisieren. Außerdem wurde eine Implementierung gesucht, die mit bereits vorhandenen Linux Treibern kompatibel

ist, um das Modul, nach dem Herstellen der Kompatibilität des ParaNut mit Linux, auch dort verwenden zu können.

Für die Auswahl eines Moduls wurden zwei Implementierungen analysiert. Zum einen den „wbuart32“, welcher auf GitHub[4] bereitgestellt wird. Er verfügt über eine Wishbone Schnittstelle mit 32 Bit Breite, ist jedoch nicht in SystemC, sondern in Verilog implementiert, was eine Konvertierung nach SystemC erfordern würde. Außerdem gibt es zum aktuellen Zeitpunkt keinen Linux Treiber für das Hardwaremodul.

Die andere Implementierung heißt „wb_uart“ und wird auf opencores[5] bereitgestellt. Sie verfügt wie das erste Modul über eine Wishbone Schnittstelle, die jedoch nur 8-Bit breit ist, wodurch eine Anpassung erforderlich wird und ist ebenfalls nicht in SystemC, sondern in VHDL implementiert. Jedoch verfügt dieses Hardwaremodul über einen bereits vorhandenen Linux Treiber, da es den Chip 16C750 von Texas Instruments nachempfunden wurde. Außerdem befand sich der Quellcode des Hardwaremoduls bereits im Repository des ParaNuts. Aufgrund des bereits vorhandenen Linux Treibers wurde entschieden, dass das Modul „wb_uart“ implementiert wird.

6.1.2 Konvertierung zu SystemC

Für den Einbau in den ParaNut musste das Hardwaremodul von VHDL in SystemC übersetzt werden. Dazu mussten die bereits im ParaNut-Projekt vorhandenen Dateien, bis auf Dateien, die nur für Testbenches nötig sind, bearbeitet werden. Hierbei fielen mehrere Unterschiede zwischen VHDL und SystemC auf. Dadurch wurden weitere Schritte erforderlich. Zum einen gibt es in SystemC das Konzept von „Generics“ nicht, das es in VHDL erlaubt, mehrere Instanzen eines Hardwaremoduls mit unterschiedlicher Konfiguration zu erzeugen. Dies soll anhand des im „wb_uart“ verwendeten Moduls „slib_fifo“ weiter erläutert werden. Wie in Listing 6.1 zu erkennen, besitzt die „entity“ Definitison des Moduls einen Bereich „generic“, in dem die Breite der Speicherstellen der FIFO, mit `WIDTH`, und der Exponent für die Anzahl der Speicherstellen, mit `SIZE_E` angegeben werden. Bei den Werten „6“ und „8“ handelt es sich um Standardwerte, die dann verwendet werden, wenn bei der Instanziierung kein Wert übergeben wird. Die „Generics“ werden, unter anderem dazu verwendet, die Breite der Eingangs- und Ausgangsbusse zu definieren [12].

```
29 entity slib_fifo is
30     generic (
31         WIDTH       : integer := 8;                    -- FIFO width
32         SIZE_E      : integer := 6                     -- FIFO size (2^SIZE_E)
33     );
34     port (
```

[4] https://github.com/ZipCPU/wbuart32
[5] https://opencores.org/projects/wb_uart

```
35          CLK             : in  std_logic;                                -- Clock
36          RST             : in  std_logic;                                -- Reset
37          CLEAR           : in  std_logic;                                -- Clear FIFO
38          WRITE           : in  std_logic;                                -- Write to FIFO
39          READ            : in  std_logic;                                -- Read from FIFO
40          D               : in  std_logic_vector(WIDTH-1 downto 0);       -- FIFO input
41          Q               : out std_logic_vector(WIDTH-1 downto 0);       -- FIFO output
42          EMPTY           : out std_logic;                                -- FIFO is empty
43          FULL            : out std_logic;                                -- FIFO is full
44          USAGE           : out std_logic_vector(SIZE_E-1 downto 0)       -- FIFO usage
45      );
46  end slib_fifo;
```

Listing 6.1: Beispiel eines „entitys“ mit Generics

Ein Beispiel einer Instanziierung wird im Listing 6.2 gezeigt. Hier werden mit Hilfe einer „Generic Map“ die Werte der einzelnen „Generics“ gesetzt. So kann aus dem Modul `slib_fifo` eine FIFO erzeugt werden, die wie in diesem Fall, 11 Bit Breite Speicherstellen hat.

```
810     UART_RXFF: slib_fifo generic map (WIDTH => 11, SIZE_E => 6)
811                          port map (CLK       => CLK,
812                                    RST       => RST,
813                                    CLEAR     => iRXFIFOClear,
814                                    WRITE     => iRXFIFOWrite,
815                                    READ      => iRXFIFORead,
816                                    D         => iRXFIFOD,
817                                    Q         => iRXFIFOQ,
818                                    EMPTY     => iRXFIFOEmpty,
819                                    FULL      => iRXFIFO64Full,
820                                    USAGE     => iRXFIFOUsage
821                                   );
```

Listing 6.2: Beispiel einer Generic Map

Es wurde versucht, dieses Konzept in SystemC mit Template Variablen nachzubilden. Es kam jedoch zu Compiler Fehlermeldungen, wodurch es nötig wurde, zwei Versionen des konvertierten Quellcodes zu erstellen, um die Receiver FIFO mit 11 Bit breiten Speicherstellen und die Transmitter FIFO mit 8 Bit Breiten Speicherstellen zu modellieren.

Neben den fehlenden „Generics“ fiel ein weiterer Unterschied zwischen VHDL und SystemC auf. Wie im Listing 6.3, der „port map“ des Interrupt Controllers zu sehen, war es nötig, Teile von Registern an Submodule weiterzugeben.

```
469     -- Interrupt control and IIR
470     UART_IIC: uart_interrupt port map (CLK => CLK,
471                                        RST => RST,
472                                        IER => iIER(3 downto 0),
473                                        LSR => iLSR(4 downto 0),
474                                        THI => iTHRInterrupt,
475                                        RDA => iRDAInterrupt,
476                                        CTI => iCharTimeout,
477                                        AFE => iMCR_AFE,
478                                        MSR => iMSR(3 downto 0),
479                                        IIR => iIIR(3 downto 0),
480                                        INT => INT
481                                       );
```

Listing 6.3: Verbindung von teilen eines Registers

Was in SystemC nicht möglich ist, da ein „sc_signal“ über keinen Operator zum Ansprechen von Teilen des Signals verfügt [6]. Um das Problem zu umgehen, wurden zusätzlich Signale mit den benötigten Breiten angelegt, die mit dem Modul des Interrupt Controllers verbunden wurden. Die Teilsignale von „iLSR“ und „iMSR“ werden mit Hilfe der „WriteMethod“ mit Werten befüllt, was im Listing 6.4 zu sehen ist.

```
482 void Wb8Uart16750::WriteMethod(){
483
484     sc_uint<8> iLSR_var = iLSR.read();
485     sc_uint<8> iMSR_var = iMSR.read();
486
487     iLSR_4_0 = iLSR_var(4,0);
488     iMSR_3_0 = iMSR_var(3,0);
489 }
```

Listing 6.4: WriteMethod

Wohingegen die Teilsignale von „iIIR“ und „iIER“ mit Hilfe von sogenannten „CombinationMethods“ zu den kompletten Registern zusammengesetzt werden. Ein Beispiel wird in Listing 6.5 gezeigt werden soll.

```
1604 void Wb8Uart16750::IIRCombinationMethod(){
1605
1606     sc_uint<8> iIIR_var = iIIR.read();
1607
1608     iIIR_var(3,0) = iIIR_3_0.read();
1609     iIIR_var(7,4) = iIIR_7_4.read();
1610
1611     iIIR = iIIR_var;
1612 }
```

Listing 6.5: IIRCombinationMethod

Beim Umschreiben des VHDL Codes fielen noch weitere Unterschiede auf, diese werden in der Arbeit von Elias Schuler behandelt [13].

6.1.3 Erstellen von Testbenches

Das Erstellen von Testbenches wurde durch Elias Schuler durchgeführt und wird somit in seiner Arbeit behandelt [13].

6.1.4 High-Level-Synthese des UART-Moduls

Um den SystemC Code in die programmierbare Logik des FPGAs zu bringen, muss dieser mit Hilfe eines „High Level Synthesis“(HLS) Tools in eine Hardwarebeschreibungssprache umgewandelt werden. Dazu standen zwei Tools zur Auswahl.

HLS mit Vivado

Es wurde versucht, die HLS des SystemC Codes mit Hilfe des bereits im ParaNut-Projekt verwendeten „Vivado HLS“ Programms durchzuführen. Hierbei kam es jedoch zu Problemen bei der Initialisierung der Submodule des UART. Eine genaue Beschreibung dieses Vorgangs ist in der Arbeit von Elias Schuler zu finden [13].

HLS mit dem „Intel Compiler for SystemC“

Für eine erfolgreich HLS des UART-Moduls wurde das „Intel Compiler for SystemC“ ICSC HLS Tool benutzt. Aufgrund der bei der HLS Synthese mit Vivado aufgetretenen Probleme bei der Initialisierung von Submodulen des UART, wurde das Tool ICSC verwendet, da es aufgrund der Bachelorarbeit von Marco Milenkovic im ParaNut-Projekt bereits verwendet wird [10]. Das Tool kann mit Hilfe der Anleitung von GitHub[6] auf dem System des Entwicklers installiert werden. Dabei ist es wichtig, die beigelegte `setenv.sh` nicht in `.bashrc` hinzuzufügen, da dadurch das Kompilieren von Testbenches im ParaNut-Projekt nicht mehr möglich ist.

Um die HLS durchzuführen, musste eine „CMakeLists.txt“ Datei erstellt werden. Dafür wurde eine Vorlage von Marco Milenkovic verwendet, die für das UART-Modul angepasst werden musste [10]. Zum einen mussten alle `.cpp` Quelldateien in die „CMakeLists.txt“ eingetragen werden. Zum anderen wurde ein weiterer „Includepfad“ eingefügt, um die `base.h` aus dem ParaNut-Projekt zu verwenden. Auch musste der Name des Moduls in der Datei angepasst werden.

Bei der Instanziierung von Submodulen war zu beachten, dass diese nicht, wie in der Testbench, mit Hilfe des `new` Operators geschieht, da dies zu Fehlermeldungen bei der HLS führt.

Außerdem musste die Dateien `base.cpp` und `base.h` angepasst werden. Zum einen musste die Deklaration für die Synthese der Funktion „pn_GetTraceName“ von der `h` in die `cpp` Datei verschoben werden, da sonst Fehlermeldungen über eine fehlende Funktionsdekaration entsteht. Eine weitere Anpassung musste an der Klasse „CPerfMonCPU“, für die Synthese, durchgeführt werden. Dort wurde die Funktion „Init“ in der `h` hinzugefügt, da diese in der `cpp` Datei aufgerufen wird.

Durch diese Anpassungen war es möglich, das UART-Modul mit Hilfe des ICSC Tools von SystemC in SystemVerilog zu konvertieren. Dabei fiel auf, dass die Fehlermeldungen von ICSC umfangreicher sind als die von „Vivado_HLS“. Zum einen

[6] https://github.com/intel/systemc-compiler/wiki/Getting-started

bricht ICSC die Synthese ab, wenn es fehlenden Signale in der Sensitivitätsliste erkennt, sodass diese nicht übersehen werden können. Des Weiteren gibt es Meldungen, die darauf hinweisen, dass Latches entstehen könnten. Aufgrund dieser Meldungen konnten Fehler in der SystemC Implementierung behoben werden, die bei dem Synthese Versuch mit „Vivado_HLS“ nicht auffielen.

Um den synthetisieren SystemVerilog Code in den ParaNut einzufügen, musste dieser noch mit Hilfe des Programms „sv2v“[7] zu Verilog umgewandelt werden. Um das UART-Modul des ParaNuts auch für Entwickler ohne eine Installation von ICSC nutzbar zu machen, wurde entschieden, die von ICSC und sv2v erzeugte Verilog Datei im Repository des ParaNut-Projekts zu hinterlegen.

Zum Aktualisieren, nach Änderungen des SystemC Codes, wurde ein Makefile Target erstellt, das es dem Entwickler erlaubt, die bereits vorhandene Verilog Datei zu erneuern. Somit ist eine Installation von ICSC und sv2v nur bei Änderungen UART-Modul nötig.

So konnte die High Level Synthese des UART-Modul abgeschlossen werden.

6.1.5 Einbau am AXI Bus des Systems

Um das UART-Modul zu testen, wurde dieses an den in Abbildung 6.1 zu sehenden „Advanced eXtensible Interface“(AXI) Bus des Systems angeschlossen. Hierfür musste das Modul analog zum ParaNut in einen IP-Core umgewandelt werden. Dazu musste das `swb2maxi` Modul, was für „Slave Wishbone to Master AXI“ steht, mit dem UART-Modul verbunden werden. Dieses übersetzt die Wishbone Bus Schnittstelle des UART-Moduls in eine AXI Schnittstelle, die mit dem AXI Bus des Systems verbunden werden kann. Hierfür musste das Modul `swb2maxi` angepasst werden, was Elias Schuler in seiner Arbeit behandelt [13].

Auch das Erstellen eines Top-Level VHDL Moduls, das die Verilog Datei der HLS und den Buskonverter miteinander verbindet und sich am Top-Level Modul des ParaNut IP-Cores orientiert, wurde von Elias Schuler übernommen [13].

Um den IP-Core zu erstellen, wurde das für Vivado benötigte `tcl`-Skript des ParaNut-Projekts an das UART-Modul angepasst. Hierzu mussten die für den IP-Core des UART-Moduls benötigten Dateien in das Skript eingetragen und der Name des Moduls geändert werden. Außerdem wurde ein Makefile Target erstellt, dass das Erzeugen des IP-Cores automatisiert.

[7] https://github.com/zachjs/sv2v

Bei der Ausführung ist es aufgrund eines Fehlers in Vivado, der es erfordert auf Computern mit drei oder mehr Bildschirmen ein Patch zu installiert, um einen Absturz von Vivado beim Öffnen zu verhindert. Hierfür ist es nötig geworden, das `tcl`-Skript zum Erstellen des IP-Cores nochmals zu modifizieren. Eine tiefer gehende Beschreibung kann dem Kapitel A.11 des Anhangs, entnommen werden.

Der erstellte IP-Core wurde in ein ParaNut System eingebaut. Dazu wurde der erstellte UART IP-Core zu einem bereits synthetisieren ParaNut-„refdesign“-Systems hinzugefügt.

Hierfür wurde der IP-Core in das „refdesign“-System kopiert und das Blockdesign der `xpr` Datei des Systems geöffnet, um dort das UART-Modul hinzuzufügen. In Abbildung 6.2 ist das modifizierte Blockdesign des ParaNut-Systems zu erkennen. Das besteht unter anderem aus dem „processing_system7_0“ (PS7), welches dem orangen Bereich aus der Abbildung 6.1 entspricht. Darin ist unter anderem der ARM-Prozessor und der Speichercontroller enthalten. Der „AXI Interconnect“ verbindet die einzelnen Teilnehmer des AXI Busses miteinander, wie in Abbildung 6.2 zu sehen ist. Der Baustein mit dem Namen „Paranut_0“ stellt den IP-Core des ParaNut-Prozessors dar und verfügt über Takt, Reset und AXI Bus Schnittstellen. Er ist außerdem noch mit einer JTAG-Schnittstelle versehen. „Uart_0“ beinhaltet den IP-Core des erstellten UART-Moduls.

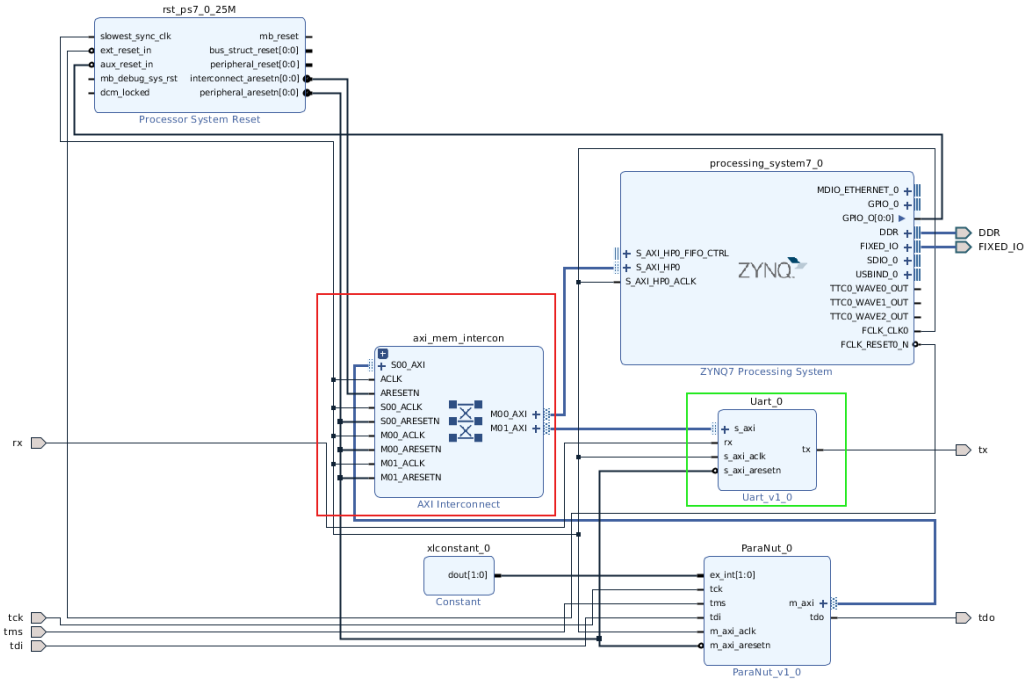

Abbildung 6.2: Einbau des UART-Moduls

Dem rot markierte Block des „AXI Interconnects“ wurde eine weitere Master Schnittstelle hinzugefügt, um das UART-Modul dort anzuschließen. Außerdem wurde das grün markierte Modul „Uart_0“ in das Blockdesign eingebaut und mit dem „AXI Interconnect“ verbunden. Des Weiteren wurden die Signale „tx“ und „rx“ als externe Signale definiert. Dafür wurde ein Constraint File angelegt, das angibt, an welchen Pins eines PMOD Stecker die Signale verfügbar sind. Beim UART wurden diese auf den PMOD JE gelegt. Das Signal „tx“ liegt dabei auf dem Pin 1 und „rx“ auf dem Pin 7. Zusätzlich musste noch eine Adresse im Adressraum des AXI Bus für den UART definiert werden. Dazu wurde die Startadresse und Größe im Adressraum im Adresseditor von Vivado festgelegt.

Die erste Ansteuerung erfolgte mit einem Programm für den ARM-Prozessor des PS7, was in der Arbeit von Elias Schuler behandelt wird [13].

Das Programm des ARM-Prozessors wurde in eines für den ParaNut-Prozessors umgeschrieben. Dazu wurde das Programm „hello_newlib“ als Basis verwendet. Dieses wurde kopiert, der Code des ARM-Prozessors eingefügt und der Xilinx spezifische Code entfernt. Außerdem wurden Defines für die Adressen der Register des UART-Moduls angelegt, um diese mit einem Namen statt der Adresse ansprechen zu können. Auch wurde das Makefile an das neue Programm angepasst. Das erstellte Programm kann in das refdesign des ParaNut geladen und auf diesen übertragen werden. Bei den Tests fielen Probleme mit dem AXI Bus auf. Diese werden in der Arbeit von Elias Schuler behandelt [13]. Da es sich bei der Ansteuerung am AXI Bus um einen Test handelte, wurde das UART Modul in den Wishbone Bus des ParaNuts integriert.

6.1.6 Einbau am Wishbone Bus des Systems

Das UART-Modul wurde an den Wishbone Bus des ParaNuts angeschlossen. Dazu wurden die aus der HLS stammende Verilog Datei in den Ordner der VHDL Dateien des ParaNut-Prozessors gelegt.

Außerdem mussten weitere Dateien angelegt werden. In einer Datei wird ein VHDL „package“ erzeugt. VHDL „packages“ können Deklarationen und Implementierungen enthalten [2]. Im Falle des UART-Moduls ist in der Datei die Definition eine VHDL „component“ für das Modul „uart_wrapper“ enthalten. In einer weiteren Datei ist das „entity“ des „uart_wrapper“ Moduls enthalten. Dort werden außerdem die Signale, des Verilog Moduls aus der HLS, mit dem VHDL „uart_wrapper“ entity verbunden. s Für den Einbau mussten auch bereits existierende Dateien erweitert werden. Hierbei wurden Möglichkeiten zur Konfiguration des Moduls geschaffen. Zum einen kann die Synthese des UART-Moduls abgeschaltet werden, um Platz auf

dem FPGA zu sparen. Weiterhin gibt es eine Option, die Basisadresse im Adressraum des ParaNut zu verändern.

Um beide Funktionen zu realisieren, mussten Anpassungen am Toplevel-Modul des ParaNut-Prozessors durchgeführt werden. Dieses Modul ist für das Verbinden der Submodule des ParaNut-Prozessors zuständig. Beim Einfügen der „port map“ wurde diese in einen „if generate“ Block eingebaut. Dieses „if generate“ erlaubt es, dass das UART-Modul nur dann mit dem ParaNut synthetisiert wird, wenn die Bedingung erfüllt ist [12].

Um die Basisadresse des UART-Moduls im Adressraum zu verschieben, musste bei der Verbindung des Moduls mit dem Wishbone Bus, eine Abfrage für die Basisadresse hinzugefügt werden, mit welcher die Signale des Wishbone Buses nur dann mit dem UART-Modul verbunden werden, wenn dieses adressiert ist.

Außerdem mussten der „entity“ Definition, des Toplevel-Moduls im ParaNut-Prozessor, die Signale „rx“ und „tx“ des UART Moduls hinzugefügt werden. Das Gleiche musste in der Toplevel-Datei des ParaNut IP-Cores geschehen, um diese Leitungen mit den PMOD Anschlüssen des FPGA-Boards verbinden zu können.

Um die implementierten Konfigurationsoptionen nutzen zu können, müssen diese in mehrere Konfigurationsdateien des Paranut-Projekts eingefügt werden. Für die Konfiguration durch den Nutzer wurden sie zur zentralen Konfigurationsdatei `config.mk` des Projekts hinzugefügt. Um die Optionen in VHDL und SystemC nutzbar zu machen, kommt die Makefile Infrastruktur des ParaNuts zum Einsatz, die die Konfigurationsoptionen mit Hilfe des Programms `sed` in die benötigten Dateien schreibt. Für die VHDL Konfigurationsdatei musste die Basisadresse des UART-Moduls in das Hexadezimalformat von VHDL gebracht werden.

Bei den Tests der Konfiguration der Basisadresse fiel auf, dass das UART-Modul nur dann ordnungsgemäß funktioniert, wenn die Adresse gerade ist. Das bedeutet, dass die Adresse 0x60000000 funktioniert, die Adresse 0x70000000 jedoch nicht. Durch weitere Versuche wird vermutet, dass das Problem in der MemU des ParaNut zu verorten ist. Dies wurde jedoch nicht weiter untersucht, aber bei den Konfigurationsoptionen in der Datei `config.mk` dokumentiert.

Um das UART-Modul im refdesign des ParaNut verwenden zu können, wurden die Optionen zum Aktivieren des UART-Moduls und zum Ändern der Basisadresse in die Konfigurationsdateien der unterstützten FPGA-Boards eingefügt.

Weiterhin wurde ein Schalter zum Makefile des refdesigns hinzugefügt, der es dem Entwickler erlaubt, die „rx“ und „tx“ Leitungen des UART-Moduls entweder am

PMOD JE des FPGA Board zu nutzen oder dies über den USB-Anschluss des „Processing System 7“ (PS7) an den Computer des Entwicklers weiterzuleiten. Die zweite Option ist zum aktuellen Zeitpunkt noch nicht funktional, soll aber in Zukunft in Kombination mit einem Bootloader das Übertragen von Programmen in den Speicher des ParaNut von dem ARM-Prozessor des PS7 übernehmen. Aktuell werden die Leitungen „rx“ und „tx“ bei dieser Option offen gelassen.

Um diese Funktionalität zu implementierten, mussten die Makefiles der verwendbaren FPGA-Boards modifiziert und die bestehenden Boardfiles modifiziert werden. Um zwischen den beiden Optionen wechseln zu können, musste das bereits existierende Boardfile des FPGA Typs kopiert und umbenannt werden. Somit konnte die Konfiguration, die zukünftig für das Übertragen von Programmen verwendet wird, erzeugt werden. Um die Konfiguration zu erzeugen, bei der „rx“ und „tx“ an einem PMOD Anschluss anliegen, wurde die Datei `xpr` des ParaNut-Systems im refdesign, mit Vivado geöffnet. Dort wurden, wie in der Abbildung 6.3 zu sehen, die Leitungen, im „Board Design“ auf externe Ports gelegt und das modifizierte Boardfile abgespeichert. Im Vergleich mit der Abbildung 6.2 ist zu erkennen, dass kein eigener Block für das UART-Modul mehr existiert und die „rx“ und „tx “ Schnittstellen jetzt vom IP-Core des ParaNut-Prozessors ausgehen.

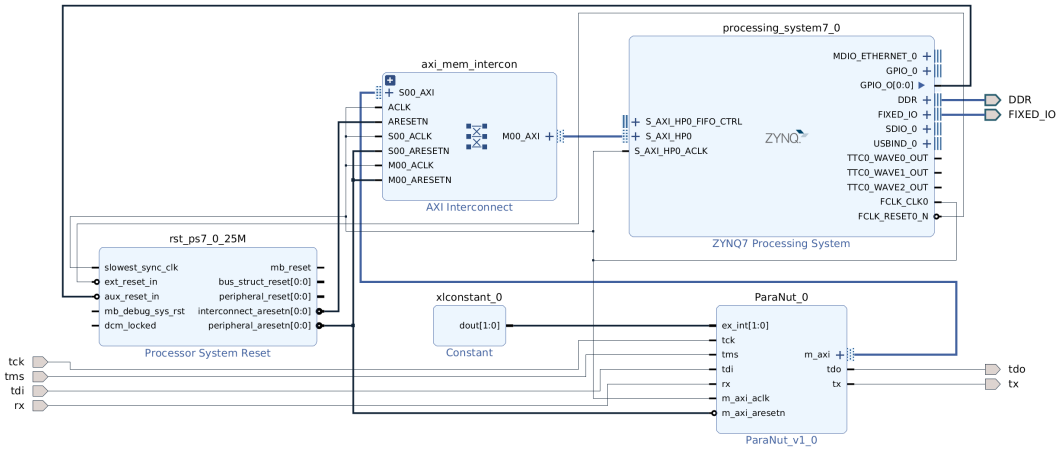

Abbildung 6.3: Verbindung von „rx“ und „tx“

Außerdem wurden noch, wie in Kapitel 6.1.5 beschreiben, Constraint Files für beide FPGA Typen angelegt, um „rx“ und „tx“ extern nutzen zu können.

Um zwischen den beiden Boardfiles wechseln zu können, mussten die Makefiles, die das Xilinx Projekt für die Synthese generieren, modifiziert werden. Hierfür wurde der Schalter aus dem Makefile des refdesign verwendet, mit dessen Hilfe das Boardfile und die für die Synthese benötigten Constraint Files, für den verwendeten FPGA Typ, ausgewählt werden.

Somit ein konfigurierbares UART-Modul im ParaNut verfügbar.

6.1.7 Peer-Test des UART-Moduls

Um beim Einbau des UART-Moduls entstandene Fehler zu finden, wurde in Zusammenarbeit mit Elias Schuler [13], ein Peer-Test erstellt, der vom ParaNut-Team durchgeführt wurde. So konnten Fehler ausfindig gemacht werden, die bei eigenen Tests nicht bemerkt wurden. Eine korrigierte Version der Anleitung des Peer-Tests ist in Kapitel A.12 des Anhangs, hinterlegt.

6.1.8 UART API

Von Elias Schuler wurde eine API für das UART-Modul erstellt. Weitere Informationen dazu sind in seiner Arbeit zu finden [13].

6.2 Bootloaders für das Übertragen von Programmen

Aufgrund von fehlender Zeit konnte der Bootloader nicht mehr implementiert werden. Dieser sollte die Aufgaben der Firmware des ARM-Prozessors übernehmen. Dabei sollte das Protokoll, das über die UART Schnittstelle gesprochen wird, gleich bleiben, um die Kompatibilität mit dem bereits vorhandenen Tool „pn-flash“ sicherzustellen. Der Bootloader, der vom ParaNut ausgeführt wird, würde mit Hilfe der neue implementierten UART Schnittstelle die Programmdaten in den Speicher schreiben und danach an die Startadresse des gerade übertragenen Programms springen, um die Aufführung zu starten. Für die Übertragung der Konsolenausgaben des ParaNut müsste der Bootloader nicht verwendet werden. Dafür sollte die Adresse von `PN_TOHOST` auf die Adresse des eingebauten UARTs verändert werden womit, die Zeichen darüber an den Computer des Entwicklers übertragen werden.

7 Ergebnisse

7.1 Geschwindigkeitsverleich der Speicherzugriffsarten

In diesem Kapitel wird beschrieben, wie der Vergleich zwischen dem Speicherzugriff über den Program-Buffer und dem abstrakten Befehl für den Speicherzugriff durchgeführt wurde.

Versuchsaufbau und -beschreibung

Verwendete Komponenten:

- Computer im Labor:
  - CPU: „Intel Xeon E3-1220 v3 @ 3.30GHz“
  - RAM: 32 GB
  - GPU: Nvidia GT 740
- Aktuelle Version des Simulators „pnsim“
- OpenOCD in der Version 0.12.0
- GDB aus der riscv-collab Toolchain
- Kompilierte Version von „hello_newlib“ mit CFLAG `-g`

Für den Test wurde wie beim Test des Komplettsystems, der in Kapitel 5.3 beschrieben wird, GDB „command files“ und Shell-Skripte verwendet.

Das „command file“ springt mit Hilfe eines Breakpoints zur „printf“ Funktion im verwendeten Programm „hello_newlib“. Worauf es 1000 Lese- und Schreibzugriffe auf den im Programm enthaltenen CString durchführt.

Das Shell-Skript benutzt zum Ausführen des Tests, die in A.2 des Anhangs, beschriebenen Befehle. Wobei der Befehl zum Starten von GDB modifiziert wurde, sodass der Debugger die Befehle aus dem „command file“ bearbeitet. Auch wird der Befehl zum Starten des Debuggers mit dem Programm `time` aufgerufen, das die Zeit zurückgibt, wie lange GDB für Ausführen des „command files“ benötigt hat. Diese Zeit wird in einer Datei abgespeichert. Das Skript führt den beschriebenen Test zehnmal aus, bevor es sich beendet.

Um bei dem Test sicherzustellen, dass nur die zu testende Speicherzugriffsart verwendet wird, muss die Konfiguration von OpenOCD angepasst werden. Dafür wird die Option `riscv set_mem_access` für den Test des abstrakten Speicherzugriffsbefehls auf „abstract“ und für den Test des Speicherzugriffs über den Programmspeicher auf „progbuf“ gesetzt.

Um den Test durchzuführen, wird das Shell-Skript mit der richtigen Einstellung in der OpenOCD Konfiguration ausgeführt. Dies geschah für beide Speicherzugriffsarten.

Ergebnisse

Bei den Testläufen konnten die in Tabelle 7.1 zu sehenden Daten erhoben werden. Dabei handelt es sich um die Zeit die laut GDB für die Ausführung des „command files“ pro Durchlauf benötigt hat.

Durchlauf	Program-Buffer	Abstrakter Befehl
	Dauer[mm:ss]	Dauer[mm:ss]
1	3:50,388	2:13,272
2	3:53,685	2:13,013
3	3:53,971	2:13,922
4	3:49,863	2:13,233
5	3:49,162	2:13,757
6	3:53,279	2:12,554
7	3:55,988	2:12,603
8	3:49,695	2:11,914
9	3:54,154	2:10,631
10	3:48,877	2:15,286

Tabelle 7.1: Messergebnisse

Für das Diagramm in Abbildung 7.1 wurden die Ergebnisse aus Tabelle 7.1 in Minuten konvertiert und der Mittelwert für beide Zugriffsarten gebildet.

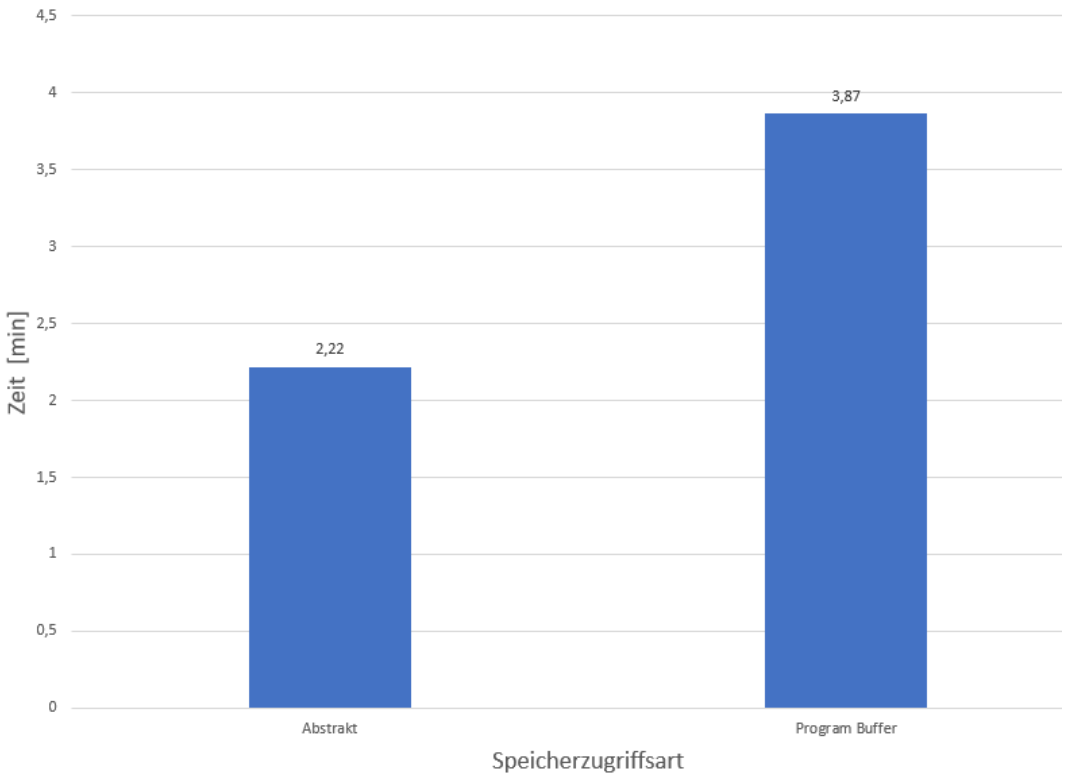

Abbildung 7.1: Vergleich der Zugriffgeschwindikkeiten

Schlussfolgerung

Wie Tabelle 7.1 und Abbildung 7.1 entnommen werden kann, ist der Zugriff auf den Speicher unter Verwendung des abstrakten Befehls in allen Durchläufen bei 1000 Lese- und Speicherzugriff um ca. 1,65 Minuten schneller. Was einer Beschleunigung von ca. $42,64$ % gegenüber dem Zugriff über den Program-Buffer entspricht. Dies deutet darauf hin, dass die Reduzierung der Zugriffe über die JTAG-Schnittstelle des DTMs, beim Zugriff über den abstrakten Befehl, die in den Daten zu sehende Beschleunigung des Zugriffs verursachen.

8 Fazit

8.1 Zusammenfassung

Im Laufe dieser Arbeit ist die Debug-Infrastruktur des ParaNuts überarbeitet worden. Der ParaNut verfügt nun über eine aktuelle und „text user interface“ fähige Version von GDB. Bei der Implementierung dieser sind fehlende CSR-Register aufgefallen, die der Architektur hinzugefügt wurden. Dabei blieb die Kompatibilität mit der bis dahin verwendeten Version von GDB bestehen.

Weiterhin ist der Speicherzugriff des Debuggers über das DM schneller als zum Beginn der Arbeit. Dies konnte durch den Einbau des Speicherzugriffs über einen eigenen abstrakten Befehl bewerkstelligt werden. Eine Vergleich der zwei Speicherzugriffsarten des ParaNuts ergab, dass der Zugriff über den abstrakten Befehl um ca. 43 % schneller ist als der über den Program-Buffer.

Auch die Tests für die Hardware der Debug-Infrastruktur erfuhren eine Verbesserung. Zum einen durch das Erstellen eines Testkonzepts und die Erweiterung der Testbenches von DM und DTM. Zum anderen durch die Schaffung eines Tests für die komplette Debug-Infrastruktur. Hierbei kommen „command files“ und Shell-Skripte zur Verwendung.

Um die Abhängigkeit vom FPGA-Hersteller Xilinx zu reduzieren, wurde ein eigenes UART-Modul entwickelt, das in Zukunft die Verwendung des ARM-Prozessors auf dem FPGA-Board überflüssig machen wird. Auch stehlt dieses eine Standard Kommunikationsschnittstelle dar. Die zum Beispiel für externe Hardware benutzt werden kann. Hierzu ist eine VHDL Version eines UART 16750 Chips, die ursprünglich von Texas Instruments stammt, in SystemC übersetzt und in den ParaNut eingebaut worden [17]. Um die Übertragung von Programmen über den implementierten UART durchzuführen, sind noch weitere Arbeiten erforderlich.

8.2 Ausblick

In Zukunft soll mit Hilfe der eigenen UART-Schnittstelle das Flashen von Programmen in den Speicher des ParaNut übernommen werden. Dazu wird ein Bootloader

implementiert, der die Funktionalität der ARM-Firmware übernehmen wird. Hierzu soll der USB-Anschluss, des FPGA-Boards, benutzt werden.

Auch sollen in Zukunft die Tests für die komplette Debug-Infrastruktur für eine Verwendung auf der Hardware angepasst werden. Außerdem werden Konfigurationsoptionen in das Skript für Überprüfung des Komplettsystems eingefügt.

Literaturverzeichnis

[**1**] Catsoulis, John: *Designing embedded hardware*. Beijing [u.a.] : O'Reilly, 2002. – ISBN 9780596003623 2.1

[**2**] Gazi, Orhan: *A Tutorial Introduction to VHDL Programming*. 1. Singapore : Springer Singapore, 2019 https://doi.org/10.1007/978-981-13-2309-6. – ISBN 9789811323096 6.1.6

[**3**] GDB Developers: *GDB: The GNU Project Debugger*. https://www.sourceware.org/gdb/. Version: Dezember 2022. – Abgerufen am: 14.01.2023 1.1

[**4**] Hagen, William: *The Definitive Guide to GCC*. Second Edition. Berkeley, CA : Apress, 2006 http://dx.doi.org/10.1007/978-1-4302-0219-6. – ISBN 9781430202196. – Includes index 3.1.2

[**5**] IEEE: IEEE Standard Test Access Port and Boundary-Scan Architecture. In: *IEEE Std 1149.1-1990* (1990), S. 1–139. http://dx.doi.org/10.1109/IEEESTD.1990.114395. – DOI 10.1109/IEEESTD.1990.114395 (document), 2.2.1, 2.2.2, 2.2.3, 2.2.4, 2.1, 2.2.5

[**6**] IEEE: IEEE Standard for Standard SystemC Language Reference Manual. In: *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)* (2012), S. 1–638. http://dx.doi.org/10.1109/IEEESTD.2012.6134619. – DOI 10.1109/IEEESTD.2012.6134619 6.1.2

[**7**] Kiefer, Gundolf ; Bahle, Alexander ; Meyer, Christian H. ; Wagner, Felix ; Borgsmüller, Nico: *The ParaNut Processor*. 1.0-114, Februar 2023. – Abgerufen am: 19.02.2023 2.7.1.1

[**8**] Kiefer, Gundolf ; Bahle, Alexander ; Pfützner, Anna K. ; Vollbracht, Lutz: The ParaNut/RISC-V Processor - An Open, Parallel and Highly Scaable Processor Architecture for FPGA-based Sytems. In: *EmbeddedWorld* (2020), 1-8. https://ees.hs-augsburg.de/paranut/paranut-paper-ew2020.pdf. – Abgerufen: 10.12.2022 2.6.1, 2.6.2, 2.6.2

[**9**] Lacamera, Daniele: *Embedded systems architecture*. Birmingham : Packt Publishing, 2018. – ISBN 9781788830287 2.1

[**10**] Milenkovic, Marco: *Synthese von SystemC Code mit Open-Source-Tools*. Februar 2023. – Laufende Bachelorarbeit, Vorrausischtliche Abgabe: 21.03.2023 6.1.4

[**11**] Newsome, Tim ; Wachs, Megan: *RISC-V External Debug Support*. https://riscv.org/wp-content/uploads/2019/03/riscv-debug-release.pdf. Version: März 2019. – Abgerufen am: 02.01.2023 (document), 1.1, 2.2.1, 2.5.1, 2.5.3, 2.7, 2.7.1, 2.4, 2.7.1, 2.7.2, 2.7.3, 2.7.3.1, 4, 4.3, 4.3, A.1, A.5, A.2

[**12**] Reichardt, Jürgen: *Digitaltechnik und digitale Systeme*. Berlin, Boston : De Gruyter, 2021. http://dx.doi.org/doi:10.1515/9783110706970. http://dx.doi.org/doi:10.1515/9783110706970. – ISBN 9783110706970 6.1.2, 6.1.6

[**13**] Schuler, Elias: *Entwurf eines Demonstrators für intelligente Tier-Implantate auf Basis eines ParaNut/RISC-V-Prozessors*. März 2023. – Laufende Bachelorarbeit, Vorrausischtliche Abgabe: 20.03.2023 6, 6.1.2, 6.1.3, 6.1.4, 6.1.5, 6.1.5, 6.1.7, 6.1.8, A.12

[**14**] Schäferling, Michael: *Der ParaNut-Prozessor*. https://ees.hs-augsburg.de/paranut/index.html. Version: Januar 2023. – Aufgerufen: 13.01.2023 2.6.1

[**15**] Stallman, Richard ; Pesch, Roland ; Shebs, Stan ; al. et: *Debugging with GDB*. https://sourceware.org/gdb/current/onlinedocs/gdb.pdf. Version: 2023. – Abgerufen am 28.01.2023 2.3

[**16**] Stollon, Neal: *On-chip Instrumentation Design And Debug For Systems On Chip*. Springer, 2010. – ISBN 9781441975621 2.2.1

[**17**] Texas Instruments: Asynchronous Communications Element With 64-Byte FIFOs And AutoFlow Control datasheet (Rev. C). (1997), Dezember. https://www.ti.com/lit/gpn/tl16c750. – Abgerufen am 06.01.2023 2.1.1, 8.1

[**18**] The OpenOCD Project: *remote_bitbangpage OpenOCD Developer's Guide*. GitHub. https://raw.githubusercontent.com/openocd-org/openocd/master/doc/manual/jtag/drivers/remote_bitbang.txt. – Abgerufen am: 02.02.2022 2.4.1, 2.4.1

[**19**] The OpenOCD Project: *Open On-Chip Debugger:OpenOCD User's Guide*, Januar 2023. – Version: 0.11.0-rc2 2.4, 2.7.3.1, 4.4

[**20**] The OpenOCD Project: *Open On-Chip Debugger:OpenOCD User's Guide*, Januar 2023. – Version: 0.12.0 2.4, 2.7.3.1, 3.1.3, 3.1.4, 4.4

[**21**] Waterman, Andrew ; Asanovic, Krste ; Hauser, John ; SiFive Inc: *The RISC-V Instruction Set ManualVolume II: Privileged Architecture*. https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf. Version: Dezember 2021. – Abgerufen am: 22.01.2023 2.5.1, 2.5.2, 3.1.3, 3.1.3

[**22**] Waterman, Andrew ; Asanoviéc, Krste: *The RISC-V Instruction Set ManualVolume I: Unprivileged ISA*. https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf. Version: Dezember 2019. – Abgerufen am: 01.01.2023 2.5.1, 3.1.3

[**23**] Xilinx: 72614 - 2018.3 Vivado - Cannot start the vivado GUI Multiple Display Issue: IllegalArgumentException: Window must not be zero. (2021). https://support.xilinx.com/s/article/72614?language=en_US. – Abgerufen am: 05.02.2023 A.11

A Anhang

A.1 Installation von GDB

Vorraussetzungen:

- Die Vorraussetzungen welche auf GitHub aufgelistet sind[8]
- libncurses-dev
- Ein symbolischer Link der python auf python3 verlinkt

Informationen:

- Zeit die zum Bauen mit der Option -j4 benötigt wird: 22 Minuten
- Größe des Git Repositories mit Build-Artefakten: 11 GB
- Größe der Installierten Toolchain: 1,4 GB
- Installiert eine komplette riscv-gnu-toolchain inklusive einer „multilib“ Bibliothek

In diesem Kapitel wird beschreiben, wie eine Version von GDB mit aktiviertem „text user interface“ gebaut wird. Dazu werden die durchzuführenden Schritte hier aufgelistet.

Der erste Schritt ist es, die vorausgesetzten Pakete zu installieren. Dazu werden zum einen die Pakete von GitHub[8] zum System hinzugefügt und zusätzlich das Paket `libncurses-dev` mit Hilfe des Paketmanagers installiert. Bei Debian wird dazu der folgende Befehl verwendet:

```
$ sudo apt install libncurses-dev
```

Wenn auf dem System `python` nicht definiert ist, muss ein symbolischer Link erstellt werden, der den Namen `python` trägt und auf `python3` zeigt.

[8] https://github.com/riscv-collab/riscv-gnu-toolchain

Nachdem alle benötigten Pakete installiert sind, kann der Programmcode von GitHub, mit folgenden Befehl geholt werden.

```
$ git clone https://github.com/riscv-collab/riscv-gnu-toolchain
```

Wenn das geschehen ist, wird in den Ordner des gerade geclonten Git Repositories gewechselt.

```
$cd riscv-gnu-toolchain
```

Danach muss der tag der Version 2023.01.04 ausgewählt werden.

```
$ git checkout 2023.01.04
```

Nun muss das Buildsystem noch konfiguriert werden. Das wird mit dem folgenden Befehl erreicht. Dazu muss der „\“ aus dem Befehl entfernt werden.

```
$ ./configure --prefix=<Installationsordner> \
--with-multilib-generator="rv32i-ilp32--;rv32im-ilp32--;rv32ima-
ilp32- -"
```

Nach der Konfiguration kann die Toolchain kompiliert und installiert werden. Das erfolgt mit dem folgenden Befehl:

```
$ make
```

Um die in dieser Toolchain enthaltene Version von GDB nutzen zu können, muss die bereits in der „freedom-tools“ Toolchain, vorhandene Version von GDB, ersetzt werden. Dazu wird im Ordner der „freedom-tool“ Toolchain die ausführbare Datei von GDB umbenannt. Danach wird diese durch einen Symlink auf die neue Version ersetzt.

A.2 Verbinden des Debuggers in der Simulation

Vorraussetzungen:

- Ein Programm welches mit Debug-Flags gebaut wurde
- Ein bereits gebauter pn-sim

Um ein Programm im Simulator des ParaNut zu debuggen, müssen die folgenden Schritte durchgeführt werden.
Zuerst muss das Programm welches getestet werden soll, mit dem Simulator gestartet

werden. Dazu wird aus dem Stammverzeichnis eines lokalen ParaNut-Repositories in den Ordner des Simulators gewechselt. Dieser befindet sich im Ordner `hw/sim`. Danach wird der Simulator mit dem folgenden Befehl gestartet.

```
$ ./pn-sim -d <Pfad zur Ausführbaren Datei>
```

Das `-d` bewirkt, dass der Simulator auf eine Verbindung mit OpenOCD wartet.

Nachdem der Simulator gestartet wurde, muss OpenOCD gestartet werden. Dazu wird eine weitere Konsole im Ordner des lokalen ParaNut-Repositories geöffnet und OpenOCD mit dem folgenden Befehl gestartet.

```
$ openocd -f tools/etc/openocd-sim.cfg
```

Der Schalter `-f` gibt an, dass der nachfolgende Pfad auf die zu verwendende Konfigurationsdatei zeigt.

Nun kann GDB gestartet werden. Dazu wird erneute eine Konsole geöffnet. Diese wird jedoch im Ordner geöffnet, in welchem sich die ausführbare Datei des zu testenden Programms befindet. Dort wird zum Starten dann folgender Befehl eingegeben.

```
$ riscv64-unknown-elf-gdb <Pfad zur Ausführbaren Datei>
```

Als letzter Schritt muss GDB noch mit OpenOCD verbunden werden. Dazu wird der folgende Befehl in die Kommandozeile von GDB eingegeben.

```
(gdb)target remote localhost:3333
```

A.3 Listing der GDB Target-Description

```
<?xml version="1.0"?>
<!DOCTYPE target SYSTEM "gdb-target.dtd">
<target version="1.0">
<feature name="org.gnu.gdb.riscv.cpu">
<reg name="zero" bitsize="32" regnum="0" save-restore="yes" type="int" group="general"/>
<reg name="ra" bitsize="32" regnum="1" save-restore="yes" type="int" group="general"/>
<reg name="sp" bitsize="32" regnum="2" save-restore="yes" type="int" group="general"/>
<reg name="gp" bitsize="32" regnum="3" save-restore="yes" type="int" group="general"/>
<reg name="tp" bitsize="32" regnum="4" save-restore="yes" type="int" group="general"/>
<reg name="t0" bitsize="32" regnum="5" save-restore="yes" type="int" group="general"/>
<reg name="t1" bitsize="32" regnum="6" save-restore="yes" type="int" group="general"/>
<reg name="t2" bitsize="32" regnum="7" save-restore="yes" type="int" group="general"/>
<reg name="fp" bitsize="32" regnum="8" save-restore="yes" type="int" group="general"/>
<reg name="s1" bitsize="32" regnum="9" save-restore="yes" type="int" group="general"/>
<reg name="a0" bitsize="32" regnum="10" save-restore="yes" type="int" group="general"/>
<reg name="a1" bitsize="32" regnum="11" save-restore="yes" type="int" group="general"/>
<reg name="a2" bitsize="32" regnum="12" save-restore="yes" type="int" group="general"/>
<reg name="a3" bitsize="32" regnum="13" save-restore="yes" type="int" group="general"/>
<reg name="a4" bitsize="32" regnum="14" save-restore="yes" type="int" group="general"/>
<reg name="a5" bitsize="32" regnum="15" save-restore="yes" type="int" group="general"/>
<reg name="a6" bitsize="32" regnum="16" save-restore="yes" type="int" group="general"/>
<reg name="a7" bitsize="32" regnum="17" save-restore="yes" type="int" group="general"/>
<reg name="s2" bitsize="32" regnum="18" save-restore="yes" type="int" group="general"/>
```

```
24 <reg name="s3" bitsize="32" regnum="19" save-restore="yes" type="int" group="general"/>
25 <reg name="s4" bitsize="32" regnum="20" save-restore="yes" type="int" group="general"/>
26 <reg name="s5" bitsize="32" regnum="21" save-restore="yes" type="int" group="general"/>
27 <reg name="s6" bitsize="32" regnum="22" save-restore="yes" type="int" group="general"/>
28 <reg name="s7" bitsize="32" regnum="23" save-restore="yes" type="int" group="general"/>
29 <reg name="s8" bitsize="32" regnum="24" save-restore="yes" type="int" group="general"/>
30 <reg name="s9" bitsize="32" regnum="25" save-restore="yes" type="int" group="general"/>
31 <reg name="s10" bitsize="32" regnum="26" save-restore="yes" type="int" group="general"/>
32 <reg name="s11" bitsize="32" regnum="27" save-restore="yes" type="int" group="general"/>
33 <reg name="t3" bitsize="32" regnum="28" save-restore="yes" type="int" group="general"/>
34 <reg name="t4" bitsize="32" regnum="29" save-restore="yes" type="int" group="general"/>
35 <reg name="t5" bitsize="32" regnum="30" save-restore="yes" type="int" group="general"/>
36 <reg name="t6" bitsize="32" regnum="31" save-restore="yes" type="int" group="general"/>
37 <reg name="pc" bitsize="32" regnum="32" save-restore="yes" type="int" group="general"/>
38 </feature>
39 <feature name="org.gnu.gdb.riscv.csr">
40 <reg name="ustatus" bitsize="32" regnum="65" save-restore="no" type="int" group="csr"/>
41 <reg name="uie" bitsize="32" regnum="69" save-restore="no" type="int" group="csr"/>
42 <reg name="utvec" bitsize="32" regnum="70" save-restore="no" type="int" group="csr"/>
43 <reg name="utvt" bitsize="32" regnum="72" save-restore="no" type="int" group="csr"/>
44 <reg name="vcsr" bitsize="32" regnum="80" save-restore="no" type="int" group="csr"/>
45 <reg name="uscratch" bitsize="32" regnum="129" save-restore="no" type="int" group="csr"/>
46 <reg name="uepc" bitsize="32" regnum="130" save-restore="no" type="int" group="csr"/>
47 <reg name="ucause" bitsize="32" regnum="131" save-restore="no" type="int" group="csr"/>
48 <reg name="utval" bitsize="32" regnum="132" save-restore="no" type="int" group="csr"/>
49 <reg name="uip" bitsize="32" regnum="133" save-restore="no" type="int" group="csr"/>
50 <reg name="unxti" bitsize="32" regnum="134" save-restore="no" type="int" group="csr"/>
51 <reg name="uintstatus" bitsize="32" regnum="135" save-restore="no" type="int" group="csr"/
       >
52 <reg name="uscratchcsw" bitsize="32" regnum="137" save-restore="no" type="int" group="csr"
       />
53 <reg name="uscratchcswl" bitsize="32" regnum="138" save-restore="no" type="int" group="csr
       "/>
54 <reg name="sedeleg" bitsize="32" regnum="323" save-restore="no" type="int" group="csr"/>
55 <reg name="sideleg" bitsize="32" regnum="324" save-restore="no" type="int" group="csr"/>
56 <reg name="stvt" bitsize="32" regnum="328" save-restore="no" type="int" group="csr"/>
57 <reg name="snxti" bitsize="32" regnum="390" save-restore="no" type="int" group="csr"/>
58 <reg name="sintstatus" bitsize="32" regnum="391" save-restore="no" type="int" group="csr"/
       >
59 <reg name="sscratchcsw" bitsize="32" regnum="393" save-restore="no" type="int" group="csr"
       />
60 <reg name="sscratchcswl" bitsize="32" regnum="394" save-restore="no" type="int" group="csr
       "/>
61 <reg name="vsstatus" bitsize="32" regnum="577" save-restore="no" type="int" group="csr"/>
62 <reg name="vsie" bitsize="32" regnum="581" save-restore="no" type="int" group="csr"/>
63 <reg name="vstvec" bitsize="32" regnum="582" save-restore="no" type="int" group="csr"/>
64 <reg name="vsscratch" bitsize="32" regnum="641" save-restore="no" type="int" group="csr"/>
65 <reg name="vsepc" bitsize="32" regnum="642" save-restore="no" type="int" group="csr"/>
66 <reg name="vscause" bitsize="32" regnum="643" save-restore="no" type="int" group="csr"/>
67 <reg name="vstval" bitsize="32" regnum="644" save-restore="no" type="int" group="csr"/>
68 <reg name="vsip" bitsize="32" regnum="645" save-restore="no" type="int" group="csr"/>
69 <reg name="vsatp" bitsize="32" regnum="705" save-restore="no" type="int" group="csr"/>
70 <reg name="mstatus" bitsize="32" regnum="833" save-restore="no" type="int" group="csr"/>
71 <reg name="misa" bitsize="32" regnum="834" save-restore="no" type="int" group="csr"/>
72 <reg name="mie" bitsize="32" regnum="837" save-restore="no" type="int" group="csr"/>
73 <reg name="mtvec" bitsize="32" regnum="838" save-restore="no" type="int" group="csr"/>
74 <reg name="mcounteren" bitsize="32" regnum="839" save-restore="no" type="int" group="csr"/
       >
75 <reg name="mtvt" bitsize="32" regnum="840" save-restore="no" type="int" group="csr"/>
76 <reg name="mstatush" bitsize="32" regnum="849" save-restore="no" type="int" group="csr"/>
77 <reg name="mcountinhibit" bitsize="32" regnum="865" save-restore="no" type="int" group="
       csr"/>
78 <reg name="mhpmevent3" bitsize="32" regnum="868" save-restore="no" type="int" group="csr"/
       >
79 <reg name="mhpmevent4" bitsize="32" regnum="869" save-restore="no" type="int" group="csr"/
       >
80 <reg name="mhpmevent5" bitsize="32" regnum="870" save-restore="no" type="int" group="csr"/
       >
81 <reg name="mhpmevent6" bitsize="32" regnum="871" save-restore="no" type="int" group="csr"/
       >
82 <reg name="mhpmevent7" bitsize="32" regnum="872" save-restore="no" type="int" group="csr"/
       >
83 <reg name="mhpmevent8" bitsize="32" regnum="873" save-restore="no" type="int" group="csr"/
       >
84 <reg name="mhpmevent9" bitsize="32" regnum="874" save-restore="no" type="int" group="csr"/
       >
```

```
85 <reg name="mhpmevent10" bitsize="32" regnum="875" save-restore="no" type="int" group="csr"
       />
86 <reg name="mhpmevent11" bitsize="32" regnum="876" save-restore="no" type="int" group="csr"
       />
87 <reg name="mhpmevent12" bitsize="32" regnum="877" save-restore="no" type="int" group="csr"
       />
88 <reg name="mhpmevent13" bitsize="32" regnum="878" save-restore="no" type="int" group="csr"
       />
89 <reg name="mhpmevent14" bitsize="32" regnum="879" save-restore="no" type="int" group="csr"
       />
90 <reg name="mhpmevent15" bitsize="32" regnum="880" save-restore="no" type="int" group="csr"
       />
91 <reg name="mhpmevent16" bitsize="32" regnum="881" save-restore="no" type="int" group="csr"
       />
92 <reg name="mhpmevent17" bitsize="32" regnum="882" save-restore="no" type="int" group="csr"
       />
93 <reg name="mhpmevent18" bitsize="32" regnum="883" save-restore="no" type="int" group="csr"
       />
94 <reg name="mhpmevent19" bitsize="32" regnum="884" save-restore="no" type="int" group="csr"
       />
95 <reg name="mhpmevent20" bitsize="32" regnum="885" save-restore="no" type="int" group="csr"
       />
96 <reg name="mhpmevent21" bitsize="32" regnum="886" save-restore="no" type="int" group="csr"
       />
97 <reg name="mhpmevent22" bitsize="32" regnum="887" save-restore="no" type="int" group="csr"
       />
98 <reg name="mhpmevent23" bitsize="32" regnum="888" save-restore="no" type="int" group="csr"
       />
99 <reg name="mhpmevent24" bitsize="32" regnum="889" save-restore="no" type="int" group="csr"
       />
100 <reg name="mhpmevent25" bitsize="32" regnum="890" save-restore="no" type="int" group="csr"
       />
101 <reg name="mhpmevent26" bitsize="32" regnum="891" save-restore="no" type="int" group="csr"
       />
102 <reg name="mhpmevent27" bitsize="32" regnum="892" save-restore="no" type="int" group="csr"
       />
103 <reg name="mhpmevent28" bitsize="32" regnum="893" save-restore="no" type="int" group="csr"
       />
104 <reg name="mhpmevent29" bitsize="32" regnum="894" save-restore="no" type="int" group="csr"
       />
105 <reg name="mhpmevent30" bitsize="32" regnum="895" save-restore="no" type="int" group="csr"
       />
106 <reg name="mhpmevent31" bitsize="32" regnum="896" save-restore="no" type="int" group="csr"
       />
107 <reg name="mscratch" bitsize="32" regnum="897" save-restore="no" type="int" group="csr"/>
108 <reg name="mepc" bitsize="32" regnum="898" save-restore="no" type="int" group="csr"/>
109 <reg name="mcause" bitsize="32" regnum="899" save-restore="no" type="int" group="csr"/>
110 <reg name="mtval" bitsize="32" regnum="900" save-restore="no" type="int" group="csr"/>
111 <reg name="mip" bitsize="32" regnum="901" save-restore="no" type="int" group="csr"/>
112 <reg name="mnxti" bitsize="32" regnum="902" save-restore="no" type="int" group="csr"/>
113 <reg name="mintstatus" bitsize="32" regnum="903" save-restore="no" type="int" group="csr"/
       >
114 <reg name="mscratchcsw" bitsize="32" regnum="905" save-restore="no" type="int" group="csr"
       />
115 <reg name="mscratchcswl" bitsize="32" regnum="906" save-restore="no" type="int" group="csr
       "/>
116 <reg name="mtinst" bitsize="32" regnum="907" save-restore="no" type="int" group="csr"/>
117 <reg name="mtval2" bitsize="32" regnum="908" save-restore="no" type="int" group="csr"/>
118 <reg name="pmpcfg0" bitsize="32" regnum="993" save-restore="no" type="int" group="csr"/>
119 <reg name="pmpcfg1" bitsize="32" regnum="994" save-restore="no" type="int" group="csr"/>
120 <reg name="pmpcfg2" bitsize="32" regnum="995" save-restore="no" type="int" group="csr"/>
121 <reg name="pmpcfg3" bitsize="32" regnum="996" save-restore="no" type="int" group="csr"/>
122 <reg name="pmpaddr0" bitsize="32" regnum="1009" save-restore="no" type="int" group="csr"/>
123 <reg name="pmpaddr1" bitsize="32" regnum="1010" save-restore="no" type="int" group="csr"/>
124 <reg name="pmpaddr2" bitsize="32" regnum="1011" save-restore="no" type="int" group="csr"/>
125 <reg name="pmpaddr3" bitsize="32" regnum="1012" save-restore="no" type="int" group="csr"/>
126 <reg name="pmpaddr4" bitsize="32" regnum="1013" save-restore="no" type="int" group="csr"/>
127 <reg name="pmpaddr5" bitsize="32" regnum="1014" save-restore="no" type="int" group="csr"/>
128 <reg name="pmpaddr6" bitsize="32" regnum="1015" save-restore="no" type="int" group="csr"/>
129 <reg name="pmpaddr7" bitsize="32" regnum="1016" save-restore="no" type="int" group="csr"/>
130 <reg name="pmpaddr8" bitsize="32" regnum="1017" save-restore="no" type="int" group="csr"/>
131 <reg name="pmpaddr9" bitsize="32" regnum="1018" save-restore="no" type="int" group="csr"/>
132 <reg name="pmpaddr10" bitsize="32" regnum="1019" save-restore="no" type="int" group="csr"/
       >
133 <reg name="pmpaddr11" bitsize="32" regnum="1020" save-restore="no" type="int" group="csr"/
       >
```

```
134 <reg name="pmpaddr12" bitsize="32" regnum="1021" save-restore="no" type="int" group="csr"/
        >
135 <reg name="pmpaddr13" bitsize="32" regnum="1022" save-restore="no" type="int" group="csr"/
        >
136 <reg name="pmpaddr14" bitsize="32" regnum="1023" save-restore="no" type="int" group="csr"/
        >
137 <reg name="pmpaddr15" bitsize="32" regnum="1024" save-restore="no" type="int" group="csr"/
        >
138 <reg name="hstatus" bitsize="32" regnum="1601" save-restore="no" type="int" group="csr"/>
139 <reg name="hedeleg" bitsize="32" regnum="1603" save-restore="no" type="int" group="csr"/>
140 <reg name="hideleg" bitsize="32" regnum="1604" save-restore="no" type="int" group="csr"/>
141 <reg name="hie" bitsize="32" regnum="1605" save-restore="no" type="int" group="csr"/>
142 <reg name="htimedelta" bitsize="32" regnum="1606" save-restore="no" type="int" group="csr"
        />
143 <reg name="hcounteren" bitsize="32" regnum="1607" save-restore="no" type="int" group="csr"
        />
144 <reg name="hgeie" bitsize="32" regnum="1608" save-restore="no" type="int" group="csr"/>
145 <reg name="htimedeltah" bitsize="32" regnum="1622" save-restore="no" type="int" group="csr
        "/>
146 <reg name="htval" bitsize="32" regnum="1668" save-restore="no" type="int" group="csr"/>
147 <reg name="hip" bitsize="32" regnum="1669" save-restore="no" type="int" group="csr"/>
148 <reg name="hvip" bitsize="32" regnum="1670" save-restore="no" type="int" group="csr"/>
149 <reg name="htinst" bitsize="32" regnum="1675" save-restore="no" type="int" group="csr"/>
150 <reg name="hgatp" bitsize="32" regnum="1729" save-restore="no" type="int" group="csr"/>
151 <reg name="tselect" bitsize="32" regnum="2017" save-restore="no" type="int" group="csr"/>
152 <reg name="tdata1" bitsize="32" regnum="2018" save-restore="no" type="int" group="csr"/>
153 <reg name="tdata2" bitsize="32" regnum="2019" save-restore="no" type="int" group="csr"/>
154 <reg name="tdata3" bitsize="32" regnum="2020" save-restore="no" type="int" group="csr"/>
155 <reg name="dcsr" bitsize="32" regnum="2033" save-restore="no" type="int" group="csr"/>
156 <reg name="dpc" bitsize="32" regnum="2034" save-restore="no" type="int" group="csr"/>
157 <reg name="dscratch0" bitsize="32" regnum="2035" save-restore="no" type="int" group="csr"/
        >
158 <reg name="dscratch1" bitsize="32" regnum="2036" save-restore="no" type="int" group="csr"/
        >
159 <reg name="mcycle" bitsize="32" regnum="2881" save-restore="no" type="int" group="csr"/>
160 <reg name="minstret" bitsize="32" regnum="2883" save-restore="no" type="int" group="csr"/>
161 <reg name="mhpmcounter3" bitsize="32" regnum="2884" save-restore="no" type="int" group="
        csr"/>
162 <reg name="mhpmcounter4" bitsize="32" regnum="2885" save-restore="no" type="int" group="
        csr"/>
163 <reg name="mhpmcounter5" bitsize="32" regnum="2886" save-restore="no" type="int" group="
        csr"/>
164 <reg name="mhpmcounter6" bitsize="32" regnum="2887" save-restore="no" type="int" group="
        csr"/>
165 <reg name="mhpmcounter7" bitsize="32" regnum="2888" save-restore="no" type="int" group="
        csr"/>
166 <reg name="mhpmcounter8" bitsize="32" regnum="2889" save-restore="no" type="int" group="
        csr"/>
167 <reg name="mhpmcounter9" bitsize="32" regnum="2890" save-restore="no" type="int" group="
        csr"/>
168 <reg name="mhpmcounter10" bitsize="32" regnum="2891" save-restore="no" type="int" group="
        csr"/>
169 <reg name="mhpmcounter11" bitsize="32" regnum="2892" save-restore="no" type="int" group="
        csr"/>
170 <reg name="mhpmcounter12" bitsize="32" regnum="2893" save-restore="no" type="int" group="
        csr"/>
171 <reg name="mhpmcounter13" bitsize="32" regnum="2894" save-restore="no" type="int" group="
        csr"/>
172 <reg name="mhpmcounter14" bitsize="32" regnum="2895" save-restore="no" type="int" group="
        csr"/>
173 <reg name="mhpmcounter15" bitsize="32" regnum="2896" save-restore="no" type="int" group="
        csr"/>
174 <reg name="mhpmcounter16" bitsize="32" regnum="2897" save-restore="no" type="int" group="
        csr"/>
175 <reg name="mhpmcounter17" bitsize="32" regnum="2898" save-restore="no" type="int" group="
        csr"/>
176 <reg name="mhpmcounter18" bitsize="32" regnum="2899" save-restore="no" type="int" group="
        csr"/>
177 <reg name="mhpmcounter19" bitsize="32" regnum="2900" save-restore="no" type="int" group="
        csr"/>
178 <reg name="mhpmcounter20" bitsize="32" regnum="2901" save-restore="no" type="int" group="
        csr"/>
179 <reg name="mhpmcounter21" bitsize="32" regnum="2902" save-restore="no" type="int" group="
        csr"/>
180 <reg name="mhpmcounter22" bitsize="32" regnum="2903" save-restore="no" type="int" group="
        csr"/>
```

```
181 <reg name="mhpmcounter23" bitsize="32" regnum="2904" save-restore="no" type="int" group="
        csr"/>
182 <reg name="mhpmcounter24" bitsize="32" regnum="2905" save-restore="no" type="int" group="
        csr"/>
183 <reg name="mhpmcounter25" bitsize="32" regnum="2906" save-restore="no" type="int" group="
        csr"/>
184 <reg name="mhpmcounter26" bitsize="32" regnum="2907" save-restore="no" type="int" group="
        csr"/>
185 <reg name="mhpmcounter27" bitsize="32" regnum="2908" save-restore="no" type="int" group="
        csr"/>
186 <reg name="mhpmcounter28" bitsize="32" regnum="2909" save-restore="no" type="int" group="
        csr"/>
187 <reg name="mhpmcounter29" bitsize="32" regnum="2910" save-restore="no" type="int" group="
        csr"/>
188 <reg name="mhpmcounter30" bitsize="32" regnum="2911" save-restore="no" type="int" group="
        csr"/>
189 <reg name="mhpmcounter31" bitsize="32" regnum="2912" save-restore="no" type="int" group="
        csr"/>
190 <reg name="mcycleh" bitsize="32" regnum="3009" save-restore="no" type="int" group="csr"/>
191 <reg name="minstreth" bitsize="32" regnum="3011" save-restore="no" type="int" group="csr"/
        >
192 <reg name="mhpmcounter3h" bitsize="32" regnum="3012" save-restore="no" type="int" group="
        csr"/>
193 <reg name="mhpmcounter4h" bitsize="32" regnum="3013" save-restore="no" type="int" group="
        csr"/>
194 <reg name="mhpmcounter5h" bitsize="32" regnum="3014" save-restore="no" type="int" group="
        csr"/>
195 <reg name="mhpmcounter6h" bitsize="32" regnum="3015" save-restore="no" type="int" group="
        csr"/>
196 <reg name="mhpmcounter7h" bitsize="32" regnum="3016" save-restore="no" type="int" group="
        csr"/>
197 <reg name="mhpmcounter8h" bitsize="32" regnum="3017" save-restore="no" type="int" group="
        csr"/>
198 <reg name="mhpmcounter9h" bitsize="32" regnum="3018" save-restore="no" type="int" group="
        csr"/>
199 <reg name="mhpmcounter10h" bitsize="32" regnum="3019" save-restore="no" type="int" group="
        csr"/>
200 <reg name="mhpmcounter11h" bitsize="32" regnum="3020" save-restore="no" type="int" group="
        csr"/>
201 <reg name="mhpmcounter12h" bitsize="32" regnum="3021" save-restore="no" type="int" group="
        csr"/>
202 <reg name="mhpmcounter13h" bitsize="32" regnum="3022" save-restore="no" type="int" group="
        csr"/>
203 <reg name="mhpmcounter14h" bitsize="32" regnum="3023" save-restore="no" type="int" group="
        csr"/>
204 <reg name="mhpmcounter15h" bitsize="32" regnum="3024" save-restore="no" type="int" group="
        csr"/>
205 <reg name="mhpmcounter16h" bitsize="32" regnum="3025" save-restore="no" type="int" group="
        csr"/>
206 <reg name="mhpmcounter17h" bitsize="32" regnum="3026" save-restore="no" type="int" group="
        csr"/>
207 <reg name="mhpmcounter18h" bitsize="32" regnum="3027" save-restore="no" type="int" group="
        csr"/>
208 <reg name="mhpmcounter19h" bitsize="32" regnum="3028" save-restore="no" type="int" group="
        csr"/>
209 <reg name="mhpmcounter20h" bitsize="32" regnum="3029" save-restore="no" type="int" group="
        csr"/>
210 <reg name="mhpmcounter21h" bitsize="32" regnum="3030" save-restore="no" type="int" group="
        csr"/>
211 <reg name="mhpmcounter22h" bitsize="32" regnum="3031" save-restore="no" type="int" group="
        csr"/>
212 <reg name="mhpmcounter23h" bitsize="32" regnum="3032" save-restore="no" type="int" group="
        csr"/>
213 <reg name="mhpmcounter24h" bitsize="32" regnum="3033" save-restore="no" type="int" group="
        csr"/>
214 <reg name="mhpmcounter25h" bitsize="32" regnum="3034" save-restore="no" type="int" group="
        csr"/>
215 <reg name="mhpmcounter26h" bitsize="32" regnum="3035" save-restore="no" type="int" group="
        csr"/>
216 <reg name="mhpmcounter27h" bitsize="32" regnum="3036" save-restore="no" type="int" group="
        csr"/>
217 <reg name="mhpmcounter28h" bitsize="32" regnum="3037" save-restore="no" type="int" group="
        csr"/>
218 <reg name="mhpmcounter29h" bitsize="32" regnum="3038" save-restore="no" type="int" group="
        csr"/>
219 <reg name="mhpmcounter30h" bitsize="32" regnum="3039" save-restore="no" type="int" group="
        csr"/>
```

```
220 <reg name="mhpmcounter31h" bitsize="32" regnum="3040" save-restore="no" type="int" group="
        csr"/>
221 <reg name="cycle" bitsize="32" regnum="3137" save-restore="no" type="int" group="csr"/>
222 <reg name="time" bitsize="32" regnum="3138" save-restore="no" type="int" group="csr"/>
223 <reg name="instret" bitsize="32" regnum="3139" save-restore="no" type="int" group="csr"/>
224 <reg name="hpmcounter3" bitsize="32" regnum="3140" save-restore="no" type="int" group="csr
        "/>
225 <reg name="hpmcounter4" bitsize="32" regnum="3141" save-restore="no" type="int" group="csr
        "/>
226 <reg name="hpmcounter5" bitsize="32" regnum="3142" save-restore="no" type="int" group="csr
        "/>
227 <reg name="hpmcounter6" bitsize="32" regnum="3143" save-restore="no" type="int" group="csr
        "/>
228 <reg name="hpmcounter7" bitsize="32" regnum="3144" save-restore="no" type="int" group="csr
        "/>
229 <reg name="hpmcounter8" bitsize="32" regnum="3145" save-restore="no" type="int" group="csr
        "/>
230 <reg name="hpmcounter9" bitsize="32" regnum="3146" save-restore="no" type="int" group="csr
        "/>
231 <reg name="hpmcounter10" bitsize="32" regnum="3147" save-restore="no" type="int" group="
        csr"/>
232 <reg name="hpmcounter11" bitsize="32" regnum="3148" save-restore="no" type="int" group="
        csr"/>
233 <reg name="hpmcounter12" bitsize="32" regnum="3149" save-restore="no" type="int" group="
        csr"/>
234 <reg name="hpmcounter13" bitsize="32" regnum="3150" save-restore="no" type="int" group="
        csr"/>
235 <reg name="hpmcounter14" bitsize="32" regnum="3151" save-restore="no" type="int" group="
        csr"/>
236 <reg name="hpmcounter15" bitsize="32" regnum="3152" save-restore="no" type="int" group="
        csr"/>
237 <reg name="hpmcounter16" bitsize="32" regnum="3153" save-restore="no" type="int" group="
        csr"/>
238 <reg name="hpmcounter17" bitsize="32" regnum="3154" save-restore="no" type="int" group="
        csr"/>
239 <reg name="hpmcounter18" bitsize="32" regnum="3155" save-restore="no" type="int" group="
        csr"/>
240 <reg name="hpmcounter19" bitsize="32" regnum="3156" save-restore="no" type="int" group="
        csr"/>
241 <reg name="hpmcounter20" bitsize="32" regnum="3157" save-restore="no" type="int" group="
        csr"/>
242 <reg name="hpmcounter21" bitsize="32" regnum="3158" save-restore="no" type="int" group="
        csr"/>
243 <reg name="hpmcounter22" bitsize="32" regnum="3159" save-restore="no" type="int" group="
        csr"/>
244 <reg name="hpmcounter23" bitsize="32" regnum="3160" save-restore="no" type="int" group="
        csr"/>
245 <reg name="hpmcounter24" bitsize="32" regnum="3161" save-restore="no" type="int" group="
        csr"/>
246 <reg name="hpmcounter25" bitsize="32" regnum="3162" save-restore="no" type="int" group="
        csr"/>
247 <reg name="hpmcounter26" bitsize="32" regnum="3163" save-restore="no" type="int" group="
        csr"/>
248 <reg name="hpmcounter27" bitsize="32" regnum="3164" save-restore="no" type="int" group="
        csr"/>
249 <reg name="hpmcounter28" bitsize="32" regnum="3165" save-restore="no" type="int" group="
        csr"/>
250 <reg name="hpmcounter29" bitsize="32" regnum="3166" save-restore="no" type="int" group="
        csr"/>
251 <reg name="hpmcounter30" bitsize="32" regnum="3167" save-restore="no" type="int" group="
        csr"/>
252 <reg name="hpmcounter31" bitsize="32" regnum="3168" save-restore="no" type="int" group="
        csr"/>
253 <reg name="cycleh" bitsize="32" regnum="3265" save-restore="no" type="int" group="csr"/>
254 <reg name="timeh" bitsize="32" regnum="3266" save-restore="no" type="int" group="csr"/>
255 <reg name="instreth" bitsize="32" regnum="3267" save-restore="no" type="int" group="csr"/>
256 <reg name="hpmcounter3h" bitsize="32" regnum="3268" save-restore="no" type="int" group="
        csr"/>
257 <reg name="hpmcounter4h" bitsize="32" regnum="3269" save-restore="no" type="int" group="
        csr"/>
258 <reg name="hpmcounter5h" bitsize="32" regnum="3270" save-restore="no" type="int" group="
        csr"/>
259 <reg name="hpmcounter6h" bitsize="32" regnum="3271" save-restore="no" type="int" group="
        csr"/>
260 <reg name="hpmcounter7h" bitsize="32" regnum="3272" save-restore="no" type="int" group="
        csr"/>
```

```
261 <reg name="hpmcounter8h" bitsize="32" regnum="3273" save-restore="no" type="int" group="
        csr"/>
262 <reg name="hpmcounter9h" bitsize="32" regnum="3274" save-restore="no" type="int" group="
        csr"/>
263 <reg name="hpmcounter10h" bitsize="32" regnum="3275" save-restore="no" type="int" group="
        csr"/>
264 <reg name="hpmcounter11h" bitsize="32" regnum="3276" save-restore="no" type="int" group="
        csr"/>
265 <reg name="hpmcounter12h" bitsize="32" regnum="3277" save-restore="no" type="int" group="
        csr"/>
266 <reg name="hpmcounter13h" bitsize="32" regnum="3278" save-restore="no" type="int" group="
        csr"/>
267 <reg name="hpmcounter14h" bitsize="32" regnum="3279" save-restore="no" type="int" group="
        csr"/>
268 <reg name="hpmcounter15h" bitsize="32" regnum="3280" save-restore="no" type="int" group="
        csr"/>
269 <reg name="hpmcounter16h" bitsize="32" regnum="3281" save-restore="no" type="int" group="
        csr"/>
270 <reg name="hpmcounter17h" bitsize="32" regnum="3282" save-restore="no" type="int" group="
        csr"/>
271 <reg name="hpmcounter18h" bitsize="32" regnum="3283" save-restore="no" type="int" group="
        csr"/>
272 <reg name="hpmcounter19h" bitsize="32" regnum="3284" save-restore="no" type="int" group="
        csr"/>
273 <reg name="hpmcounter20h" bitsize="32" regnum="3285" save-restore="no" type="int" group="
        csr"/>
274 <reg name="hpmcounter21h" bitsize="32" regnum="3286" save-restore="no" type="int" group="
        csr"/>
275 <reg name="hpmcounter22h" bitsize="32" regnum="3287" save-restore="no" type="int" group="
        csr"/>
276 <reg name="hpmcounter23h" bitsize="32" regnum="3288" save-restore="no" type="int" group="
        csr"/>
277 <reg name="hpmcounter24h" bitsize="32" regnum="3289" save-restore="no" type="int" group="
        csr"/>
278 <reg name="hpmcounter25h" bitsize="32" regnum="3290" save-restore="no" type="int" group="
        csr"/>
279 <reg name="hpmcounter26h" bitsize="32" regnum="3291" save-restore="no" type="int" group="
        csr"/>
280 <reg name="hpmcounter27h" bitsize="32" regnum="3292" save-restore="no" type="int" group="
        csr"/>
281 <reg name="hpmcounter28h" bitsize="32" regnum="3293" save-restore="no" type="int" group="
        csr"/>
282 <reg name="hpmcounter29h" bitsize="32" regnum="3294" save-restore="no" type="int" group="
        csr"/>
283 <reg name="hpmcounter30h" bitsize="32" regnum="3295" save-restore="no" type="int" group="
        csr"/>
284 <reg name="hpmcounter31h" bitsize="32" regnum="3296" save-restore="no" type="int" group="
        csr"/>
285 <reg name="hgeip" bitsize="32" regnum="3667" save-restore="no" type="int" group="csr"/>
286 <reg name="mvendorid" bitsize="32" regnum="3922" save-restore="no" type="int" group="csr"/
      >
287 <reg name="marchid" bitsize="32" regnum="3923" save-restore="no" type="int" group="csr"/>
288 <reg name="mimpid" bitsize="32" regnum="3924" save-restore="no" type="int" group="csr"/>
289 <reg name="mhartid" bitsize="32" regnum="3925" save-restore="no" type="int" group="csr"/>
290 </feature>
291 <feature name="org.gnu.gdb.riscv.virtual">
292 <reg name="priv" bitsize="8" regnum="4161" save-restore="no" type="int" group="general"/>
293 </feature>
294 </target>
```

Listing A.1: Listing der mit Wireshark gelesen Target-Description

A.4 Installation von OpenOCD

In diesem Kapitel wird beschrieben wie, eine Version von OpenOCD installiert werden kann, die alle Speicherzugriffsarten der RISC-V Spezifikation unterstützt.

Informationen:

- Zeit die zum Bauen mit der Option -j4 benötigt wird: 1 Minuten
- Größe des Git Repositories mit Build-Artefakten: 227 MB
- Größe des Installierten OpenOCD: 23 MB
- Installiert OpenOCD in der Version 0.12.0

Der erste Schritt ist es den Programmcode von dem dazugehörigen Git Repository herunterzuladen.

```
$ git clone ttps://git.code.sf.net/p/openocd/code openocd-code
```

Wenn dieser Schritt abgeschlossen ist, wird in den Ordner des Git Repositories gewechselt

```
$ cd openocd-code
```

Danach muss der tag der Version 0.12.0 ausgewählt werden.

```
$ git checkout v0.12.0
```

Nachdem die richtige Version ausgewählt wurde, müssen weitere benötigte Komponenten geladen werden.

```
$ ./bootstrap
```

Danach das Buildsystem von OpenOCD konfiguriert werden.

```
$ ./configure -prefix=<Installationsordner> --enable-remote-bitbang
```

Nun ist es möglich OpenOCD mit den folgenden Befehlen zu kompilieren und zu installieren.

```
$ make
```

```
$ make install
```

Um die neu instalierte Version nutzen zu können, muss der Pfad zum Ordner der ausführbaren Datei von OpenOCD zur `PATH` Variable hinzugefügt werden.

```
$ export PATH=«Pfad zum Ordner>:$PATH"
```

A.5 Beschreibung von Tests in der Testbench des DM

In diesem Kapitel werden die Tests für die im ParaNut-Prozessor implementieren abstrakten Befehle genauer geschrieben. Es wurden die Tests des abstrakten Befehls für den Registerzugriff erweitert. Wofür der Aufbau des abstrakten Befehls, der in Abbildung A.1 zu sehen ist, analysiert wurde um herauszufinden welche Tests durchgeführt werden können.

31 24	23	22 20	19	18	17	16	15 0
cmdtype	0	aarsize	aarpostincement	postexec	transfer	write	regno
8	1	3	1	1	1	1	16

Abbildung A.1: Abstrakter Register Befehl (aus [11])

Es ist möglich, das Feld „cmdtype“ zu testen. Dieses gibt an, um welchen abstrakten Befehl es sich handelt. Dazu kann ein Wert größer als „2“ zum Beispiel „3“ getestet werden, da es keine abstrakten Befehle mit einer größeren Nummer gibt [11]. Der Aufbau dieses Tests soll mit Hilfe des Kapitles A.6 im Anhang verdeutlicht werden. Dort ist zu sehen, dass zuerst das Register „command“ des DM's mit einem abstrakten Befehl im Format aus Abbildung A.1 befüllt wird. Es wurde nur das Feld „cmdtype“ verändert, worauf mit der Funktion „RunCycle“ ein Takt simuliert wird. Nachfolgend wurde das Register „abstracts“ auf den erwarteten „Not Supported“ Fehler überprüft. Um einen weiteren Test durchführen zu können, muss der Fehlerzustand durch das Schreiben des Wertes „0x0000700“ in das „abstracts“ Register zurückgesetzt werden.

Analog dazu kann ein Test für das Feld „arrsize“ durchgeführt werden, das angibt wie groß ein zu lesendes Register ist. Im ParaNut gibt es nur 32-Bit GPR-Register. Das bedeutet, dass das Feld nur den Wert „2“ akzeptiert. Im Test wurde ein anderer Wert eingegeben und überprüft, ob der „Not Supported“ Fehlerzustand gesetzt wird. Außerdem kann getestet werden, ob der „Not Supported“ Zustand bestehen bleibt, wenn ein weiterer, valider, abstrakter Befehl an das DM gesendet wird. Weiterhin kann getestet werden, ob ein „haltresume“ Fehler auftritt, wenn ein valider abstrakter Befehl an das DM gesendet wird, wenn der hart nicht angehalten ist. Tests von weiteren Feldern des abstrakten Registerzugriffsbefehls können nur getestet werden, wenn der hart angehalten ist. Dazu muss in der Testbench an den Offset `DBG_HALTED_OFFSET` der Wert „1“ geschrieben werden. Somit können die restlichen Funktionen, wie das Schreiben und Lesen von Registern, getestet werden. Dazu werden Befehle die aus Kombinationen der Felder „postexec“, was einen Sprung zu den „Program-Buffer“ Registern ausführt, „transfer“, was einen Lese- oder Schreibzugriff für die Register auslöst und „write“, welches angibt, ob das Register gelesen oder geschrieben werden soll, ausgeführt. Nach dem Senden eines solchen Befehls

werden die Inhalte der „abstract“ Register mit Sollwerten verglichen. Für das Feld „aarpostincement“ kann kein Test erstellt werden, da dieses, wie im Debug Standard beschrieben, auch nicht implementiert sein muss und dadurch keinen Einfluss auf die Ausführung hat [11]. Auch kann noch überprüft werden, ob das „GO“ Flag des „dm_flags“ Registers nach dem Erzeugen der Assembler Befehle gesetzt wurde.

Der abstrakte Befehl für den Speicherzugriff wurde analog zum abstrakten Befehl für Registerzugriff analysiert. Dafür wird der Aufbau des Speicherzugriffs Befehls in Abbildung A.2 dargestellt.

31 24	23	22 20	19	18 17	16	15 14	13 0
cmdtype	aamvirtual	aamsize	aampostincement	0	write	target-specific	0
8	1	3	1	2	1	2	14

Abbildung A.2: Abstrakter Speicher Befehl (aus [11])

Beim abstrakten Speicherzugriff werden die Felder „aamvirtual“ und „aamsize“ einzeln getestet. Dazu werden analog zum abstrakten Registerzugriff Befehle mit ungültigen Werten auf dem DM ausgeführt und überprüft ob, der „NotSupported“ Fehlerzustand erreicht wird. Das Schreiben und Lesen von Speicher muss für jeden unterstützten Wert von „aamsize“ überprüft werden. Hierfür wurde eine Schleife geschrieben, die für jeden Wert von „aamsize“ jeweils einen Lese- und Schreibzugriff mit oder ohne gesetzten „aampostincrement“ durchführt. Somit werden die abstrakten Befehle nun der Testbench getestet.

A.6 Listing eines Tests in „dm_tb“

```
346     // abstract command type test
347     // --------------------
348     PN_INFO("abstract command type test:");
349     // write abstract command to dm command register
350     CompleteDMIWrite(command, (sc_uint<8> (4), // invalid command type
351                                sc_uint<1> (0),
352                                sc_uint<3> (2), // register size
353                                sc_uint<1> (0), // no postincrement
354                                sc_uint<1> (0), // no postexec
355                                sc_uint<1> (1), // transfer register
356                                sc_uint<1> (1), // write to register
357                                sc_uint<16>(0x1000))); // register no.
358
359     RunCycle(1);
360
361     // read abstracts register
362     ret = CompleteDMIRead(abstracts);
363     PN_INFOF(("abstracts: \t0x%08x", ret));
```

```
364     // check for not supported error
365     PN_ASSERTM(ret == (sc_uint<3>  (0),
366                        sc_uint<5>  (DBG_NUM_PROGBUF),
367                        sc_uint<11> (0),
368                        sc_uint<1>  (0),
369                        sc_uint<1>  (0),
370                        sc_uint<3>  (CMDERR_NOTSUP),
371                        sc_uint<4>  (0),
372                        sc_uint<4>  (DBG_NUM_DATA)),
373                        "cmd error \"not supported\" not raised");
374
375     // clearing error in abstracts reg
376     ret = ret &= (0x0000700);
377     CompleteDMIWrite(abstracts, ret);
```

Listing A.2: Test des „cmdtype“ Feldes

A.7 Beschreibung eines Tests für die Testbench des DTM

Für den Test musste zuerst der „dmi“ Befehl in das IR Register geladen werden. Dafür musste mit Hilfe der bereits implementieren Funktion „JtagClock“, die es erlaubt, einen JTAG Takt mit vorgegebenen Werten für `TMS` und `TDI` auszuführen, in den „Shift-IR“ Zustand des JTAG Zustandsautomaten gewechselt werden. Woraufhin der „dmi“ mit der Funktion „JtagRdWR“, die eine angegebene Anzahl von Bits in das IR Schieberegister schiebt, geschrieben werden konnte. Woraufhin in den Zustand „DR-Shift“ gewechselt, wurde, um den Zugriffsbefehl in das „DR“ Register zu schreiben. Nach dem Zugriff musste mit Hilfe von Asserts überprüft werden ob die Leitungen der DMI Schnittstelle, „dmi_adr“, „dmi_dat_o“ und „dmi_rd“, die erwarteten Werte hatten. So konnte der Test der Hardware Version des DTM erweitert werden.

A.8 Gernerator für Taktsignal der Simulations Testbench des DTM

Durch das Aktivieren des `#if` in Zeile 56 der Datei „jtag_dtm.cpp“ konnte das Verhalten des `TCK`-Signals beobachtet werden. Das `#if` bewirkt, dass zusätzliche Informationen auf der Simulator Konsole ausgegeben werden. Ein Ausschnitt davon ist in Kapitel A.9 des Anhangs zu sehen. Dort kann in den Zeilen 3 bis 4 und 11 bis 12 gesehen werden. Dort wechselt das Taktsignal nicht von „0“ auf „1“, es bleibt

dort auf „0“. Das gleiche Verhalten lässt sich ebenfalls in den Zeilen 99 bis 100 beobachten. So liessen sich folgende Regeln aufstellen.

Wenn der JTAG-Zustandsautomat von „1“, was dem Zustand „Run-Test/Idle“ entspricht, auf „2“, was „Select-DR-Scan“ entspricht, wechselt, bleibt das Taktsignal im Zustand „2“ auf „0“. Das Gleiche gilt für die zweite Taktperiode in, der der Zustandsautomat den Wert „4“ annimmt, was dem Zustand „Shift-DR“ entspricht. Was auch für die zweite Taktperiode des Zustands „Shift-IR“ zutrifft, das dem Wert „11“ entspricht.

Um dieses Verhalten in der Testbench abzubilden, wurde die Funktion „JtagClockOne“ hinzugefügt, die nur eine halbe Taktperiode simuliert und die Reihenfolge des Taktes in einer Taktperiode von „0“, „1“ auf „1“, „0“ wechselt und bei einem erneuten Aufruf dies wieder umkehrt. Auch die Funktion „JtagClock“ musste angepasst werden, sodass diese nach einem Aufruf von „JtagClockOne“ die Reihenfolge der Takte einer Periode, wie bereits beschreiben, tauscht. Das Verhalten bei den Zuständen „Shift-DR“ und „Shift-IR“ wurde der Funktionen „JtagRd“ und „JtagRdWr“ hinzugefügt. Bei den Wechseln von „Run-Test/Idle“ zu „Select-DR-Scan“ wurde jeweils ein zusätzliches „JtagClockOne“ eingefügt. Mit diesen Änderungen konnte das Verhalten des Taktsignals nachgebildet werden, wodurch die Testbench ohne Fehler ausgeführt werden konnte.

A.9 Listing des der Simulations Verhaltens des DTM

```
1  state= 1, tdi=0, tdo=1, tms=0, tck=0, ir=0x11, dr=0x0
2  state= 1, tdi=0, tdo=1, tms=0, tck=1, ir=0x11, dr=0x0
3  state= 1, tdi=0, tdo=1, tms=0, tck=0, ir=0x11, dr=0x0
4  state= 2, tdi=0, tdo=1, tms=1, tck=0, ir=0x11, dr=0x0
5  state= 2, tdi=0, tdo=1, tms=1, tck=1, ir=0x11, dr=0x0
6  Capture DR; IR=0x11, DR=0x0 (40 bits)
7  state= 3, tdi=0, tdo=1, tms=0, tck=0, ir=0x11, dr=0x0
8  state= 3, tdi=0, tdo=1, tms=0, tck=1, ir=0x11, dr=0x0
9  state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x0
10 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x0
11 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x0
12 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x0
13 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x0
14 state= 4, tdi=1, tdo=0, tms=0, tck=0, ir=0x11, dr=0x8000000000
15 state= 4, tdi=1, tdo=0, tms=0, tck=1, ir=0x11, dr=0x8000000000
16 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x4000000000
17 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x4000000000
18 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x2000000000
19 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x2000000000
20 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x1000000000
```

```
21 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x1000000000
22 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x800000000
23 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x800000000
24 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x400000000
25 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x400000000
26 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x200000000
27 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x200000000
28 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x100000000
29 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x100000000
30 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x80000000
31 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x80000000
32 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x40000000
33 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x40000000
34 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x20000000
35 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x20000000
36 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x10000000
37 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x10000000
38 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x8000000
39 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x8000000
40 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x4000000
41 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x4000000
42 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x2000000
43 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x2000000
44 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x1000000
45 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x1000000
46 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x800000
47 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x800000
48 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x400000
49 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x400000
50 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x200000
51 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x200000
52 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x100000
53 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x100000
54 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x80000
55 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x80000
56 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x40000
57 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x40000
58 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x20000
59 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x20000
60 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x10000
61 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x10000
62 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x8000
63 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x8000
64 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x4000
65 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x4000
66 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x2000
67 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x2000
68 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x1000
69 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x1000
```

```
70 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x800
71 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x800
72 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x400
73 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x400
74 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x200
75 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x200
76 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x100
77 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x100
78 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x80
79 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x80
80 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x40
81 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x40
82 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x20
83 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x20
84 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x10
85 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x10
86 state= 4, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x8
87 state= 4, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x8
88 state= 4, tdi=1, tdo=0, tms=0, tck=0, ir=0x11, dr=0x8000000004
89 state= 4, tdi=1, tdo=0, tms=0, tck=1, ir=0x11, dr=0x8000000004
90 state= 5, tdi=0, tdo=0, tms=1, tck=0, ir=0x11, dr=0x4000000002
91 state= 5, tdi=0, tdo=0, tms=1, tck=1, ir=0x11, dr=0x4000000002
92 Update DR; IR=0x11, DR=0x4000000002 (40 bits)
93 dmi_write=(0x10) 0x0
94 dmi=0x4000000000
95 state= 8, tdi=0, tdo=0, tms=1, tck=0, ir=0x11, dr=0x4000000002
96 state= 8, tdi=0, tdo=0, tms=1, tck=1, ir=0x11, dr=0x4000000002
97 state= 1, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x4000000002
98 state= 1, tdi=0, tdo=0, tms=0, tck=1, ir=0x11, dr=0x4000000002
99 state= 1, tdi=0, tdo=0, tms=0, tck=0, ir=0x11, dr=0x4000000002
100 state= 2, tdi=0, tdo=0, tms=1, tck=0, ir=0x11, dr=0x4000000002
101 state= 2, tdi=0, tdo=0, tms=1, tck=1, ir=0x11, dr=0x4000000002
```

Listing A.3: Auszug des Simulator beim Test

A.10 Peer-Test der Debug-Infrastruktur

In diesem Kapitel ist eine korrigierte Version der Anleitung des im Rahmen der Bachelorarbeit durchgeführten Peer-Tests, der Debug-Infrastruktur, abgelegt.

A.10.1 Einrichtung der Testumgebung

Benötigte Voraussetzungen:

- Linux (Debian/Ubuntu)

Auszuführende Schritte:

Klonen des ParaNut-Repositories und Wechsel in den Branch „dev_debugger“:

```
$ git clone https://ti-build.informatik.hs-augsburg.de:8443/
paranut_developers/paranut.git
```

```
$ cd paranut
```

```
$ git checkout dev_debugger
```

A.10.2 Installieren von GDB mit „text user interface“

Auszuführende Schritte:

- Der Anleitung zur Installation von GDB aus dem Kapitel A.1 des Anhangs, folgen.

Erwartete Ergebnisse:

- Die Installation wurde erfolgreich abgeschlossen.

A.10.3 Installieren von OpenOCD in der Version 0.12.0

Auszuführende Schritte:

- Der Anleitung zur Installation von GDB aus dem Kapitel A.4 des Anhangs, folgen.

Erwartete Ergebnisse:

- Die Installation wurde erfolgreich abgeschlossen.

A.10.4 Test der Testbenches

Auszuführende Schritte:

1. Öffnen einer Konsole in dem vorhin geklonten `paranut` Ordner.
2. Ausführen der DM Testbench:

```
$ cd hw/sysc/tb/dm
```

```
$ make run
```

3. Ausführen der JTAG DTM Testbench für die Hardware:

```
$ cd ../jtag_dtm/hw
```

```
$ make run
```

4. Ausführen der JTAG DTM Testbench für die Simulation:

```
$ cd ../sim
```

```
$ make run
```

Erwartete Ergebnisse:

- Alle Testbenches laufen ohne fehlgeschlagene Asserts

A.10.5 Test der GDB „command file“ Tests

Auszuführende Schritte:

1. Für die Ausführung dieses Tests muss die neu kompilierte Version von OpenOCD in der PATH Variable des Systems eingetragen sein. Wie im Kapitel A.4 des Anhangs, beschrieben.
2. Um den abstrakten Speicherzugriff für den Test zu aktivieren, müssen folgende Schritte ausgeführt werden:

- Öffnen der Datei `openocd-sim.cfg`, die sich im Ordner `tools/etc` des ParaNut-Repositories befindet.
- Entfernen des Zeichen „#“ in der Zeile die mit `riscv set_mem_access ...` beginnt

3. Ausführen der Schritte in der `README` Datei, die sich im Ordner `tools/gdbtest` des ParaNut-Repositories befindet. Diese Schritte werden für die 3 „priveleg-level“ des ParaNut-Prozessors ausgeführt.

Erwartete Ergebnisse:

- Alle drei Tests sind ohne Fehler durchgelaufen.

A.10.6 Manueller Test der Debug-Verbindung mit abstrakten Speicherzugriff

Auszuführende Schritte:

1. Im Wurzelverzeichnis des ParaNut Repositories den Simulator, mit folgenden Befehl, bauen.

```
$ make
```

2. Bauen des Programms, indem Sie im Ordner `sw/hello_newlib` des ParaNut-Repositories, den folgenden Befehl ausführen

```
$ make
```

3. Ausführen der Schritte zu Aufbau einer Debug-Verbindung aus Kapitel A.2 des Anhangs.

4. Laden der Target Description von GDB, für den zu testenden „privilege-level“. Das „x“ wird duch die Nummer des „privilege-level“ aus der Datei `config.mk` ersetzt. Dabei ist es wichtig, dass GDB wie in Kapitel A.1 des Anhangs installiert wurde.

```
(gdb) set tdesc filename
<Pfad zum ParaNut-Repository>/tools/etc/gdb_csr_priv<x>.xml
```

5. Setzen eines Breakpoint in Zeile 45 von „hello_newlib“ und Sprung zu diesem Breakpoint

```
(gdb) break hello_newlib.c:45
```

```
(gdb) continue
```

6. Aktivieren des „text user interface“

```
(gdb) layout regs
```

7. Nun können normale GDB-Befehle genutzt werden, um den Debugger zu testen:

   - Zum Lesen von Speicher:

     ```
     (gdb) print <Variablenname> z.B: n, hello
     ```

     s

   - Zum Lesen von Registern:

     ```
     (gdb) print $<Registername> z.B: $misa, $t1, usw.
     ```

   - Zum Schreiben von Speicher:

     ```
     (gdb) set var <Variablenname> = <neuer Wert> z.B: n, hello
     ```

   - Zum Schreiben von Registern:

     ```
     (gdb) set <Registername> = <neuer Wert> z.B: $t1, usw.
     ```

   - Zum Setzen von Breakpoints:

     ```
     (gdb) break <Zeile/Funktionsname>
     ```

   - Zum Step um eine Zeile:

     ```
     (gdb) step
     ```

- Zum Ausführen bis zum nächsten Breakpoint bzw. Ende des Programms

```
(gdb) continue
```

8. Wenn der Test abgeschlossen ist, können GDB, OpenOCD und pn-sim beendet werden.

Erwartete Ergebnisse:

- Die vom Nutzer ausgeführten Befehle laufen ohne Probleme
- Der Debugger (GDB) konnte ohne Probleme mit dem ParaNut verbunden werden

A.10.7 Manueller Test der Debug-Verbindung mit Programmspeicher Speicherzugriff

Auszuführende Schritte:

1. Editieren der Datei `openocd-sim.cfg` im Ordner `tools/etc` des ParaNut-Repositories
   - Ändern der Zeile 58 `riscv set_mem_access abstract progbuf` wie folgt

```
riscv set_mem_access progbuf
```

2. Ausführen der Schritte zu Verbindung einer Debug-Verbindung aus Kapitel A.2 des Anhangs.
3. Laden der Target Description von GDB, für den zu testenden „privilege-level“. Das „x“ wird duch die Nummer des „privilege-level“ aus der Datei `config.mk` ersetzt. Dabei ist es wichtig das GDB wie in Kapitel A.1 des Anhangs installiert wurde.

```
(gdb) set tdesc filename
<Pfad zum ParaNut-Repository>/tools/etc/gdb_csr_priv<x>.xml
```

4. Setzen eines Breakpoint in Zeile 45 von „hello_newlib“ und Sprung zu diesem Breakpoint

```
(gdb) break hello_newlib.c:45
```

```
(gdb) continue
```

5. Aktivieren des „text user interface“

```
(gdb) layout regs
```

6. Nun können normale GDB-Befehle genutzt werden, um den Debugger zu testen:

   - Zum Lesen von Speicher:

     ```
     (gdb) print <Variablenname> z.B: n, hello
     ```

   - Zum Lesen von Registern:

     ```
     (gdb) print $<Registername> z.B: $misa, $t1, usw.
     ```

   - Zum Schreiben von Speicher:

     ```
     (gdb) set var <Variablenname> = <neuer Wert> z.B: n, hello
     ```

   - Zum Schreiben von Registern:

     ```
     (gdb) set <Registername> = <neuer Wert> z.B: $t1, usw.
     ```

   - Zum Setzen von Breakpoints:

     ```
     (gdb) break <Zeile/Funktionsname>
     ```

   - Zum Step um eine Zeile:

     ```
     (gdb) step
     ```

   - Zum Ausführen bis zum nächsten Breakpoint bzw. Ende des Programms

     ```
     (gdb) continue
     ```

7. Wenn der Test abgeschlossen ist, können GDB, OpenOCD und pn-sim beendet werden.

8. Rückgängig machen der Änderungen in der Zeile 58 der Datei `openocd-sim.cfg`.

Erwartete Ergebnisse:

- Die vom Nutzer ausgeführten Befehle laufen ohne Probleme
- Der Debugger (GDB) konnte ohne Probleme mit dem ParaNut verbunden werden

A.11 Patch für die Verwendung von drei Monitoren mit Vivado

Dieser Patch fügt die Optionen `-jvm` und `-patch-module=java.desktop=<path-to-jar>` der Startdatei von Vivado hinzu, die an die Datei `uart.tcl` als Übergabeparameter weitergegeben werden [23], deren Abfrage die neuen Parameter nicht bekannt sind, was eine Fehlermeldungen auslöst. Deshalb musste die Abfrage der Übergabeparameter, wie im Listing A.4 zu sehen ist, um die Zeilen 79 und 80 erweitert werden.

```
73 if { $::argc > 0 } {
74   for {set i 0} {$i < $::argc} {incr i} {
75     set option [string trim [lindex $::argv $i]]
76     switch -regexp -- $option {
77     "--projdir" { incr i; set projdir [lindex $::argv $i]; puts  [
           lindex $::argv $i] }
78         "--origin_dir" { incr i; set origin_dir [lindex $::argv $i];
               puts  [lindex $::argv $i] }
79         "-jvm" {}
80   "-patch-module" {}
81   default {
82         if { [regexp {^-} $option] } {
83           puts "ERROR: Unknown option '$option' specified, please type
                '$script_file -tclargs --help' for usage info.\n"
84           return 1
85         }
86       }
87     }
88   }
89 }
```

Listing A.4: Behandlung der Übergabeparameter in der `uart.tcl`

Diese Zeilen bewirken, dass die neu hinzugefügten Übergabeparameter beim Ausführen der Datei ignoriert werden.

A.12 Peer-Test des UART-Moduls

In diesem Kapitel ist eine korrigierte Version der Anleitung des mit Elias Schulers entwickelten Peer-Tests für das UART-Modul des ParaNut-Prozessor abgelegt [13].

A.12.1 Einrichtung der Testumgebung

Benötigte Voraussetzungen:

- Linux (Debian/Ubuntu)
- USB to UART Adapter
- ein FPGA Board Zybo 7000 / Zybo Z7 7020
- Picocom oder ein ähnliches Terminalprogramm für serielle Kommunikation
- Vivado Suite in der Version 2019.1

Installationsanleitung für die folgenden Tools ist im Test enthalten

- ICSC (für High-Level-Synthese Test)
- sv2v (für High-Level-Synthese Test)

Auszuführende Schritte:

Klonen des ParaNut-Repositories und Wechsel in den Branch „dev_uart16750“:

```
$ git clone https://ti-build.informatik.hs-augsburg.de:8443/
paranut_developers/paranut.git
```

```
$ cd paranut
```

```
$ git checkout dev_uart16750
```

Installation von Picocom (unter Debian/Ubuntu):

```
$ sudo apt install picocom
```

A.12.2 Test der Testbenches

Auszuführende Schritte:

In den Ordner der Testbenches wechseln und die Testbenches ausführen. Der Befehl zum Wechsel geht davon aus, dass die Konsole im Wurzelverzeichnis des ParaNut-Verzeichnisses geöffnet wurde.

```
$ cd hw/sysc/tb/uart
```

```
$ make run
```

Erwartete Ergebnisse:

- Keine Fehler beim Kompilieren der „libuart“ und der Testbenches
- Alle Testbenches laufen ohne fehlgeschlagenen Asserts

A.12.3 Test auf Hardware: Interner UART

Auszuführende Schritte:

1. Aktivieren des UART-Moduls für das entsprechende Board.
   - In `system/refdesign/hardware/boards/<Boardtyp>/config.mk` im ParaNut-Repository den Parameter `CFG_UART_ENABLE` auf `true` setzen
2. Bauen eines ParaNuts mit internen UART:
   - Sourcen der `settings64.sh` von Vivado

     ```
     $ source /<Pfad zu Xilinx Vivado>/Xilinx/Vivado/2019.1/settings64.sh
     ```

   - Sourcen der Datei `settings.sh` im Wurzelverzeichnis des ParaNut-Repositories

     ```
     $ source settings.sh
     ```

- In den Ordner des „refdesign“ des ParaNut-Repositories wechseln und die Synthese starten

```
$ cd systems/refdesign
```

```
$ make build UART_EXT=0
```

3. Überprüfung des gebauten ParaNuts:

- Bauen des Programm „uart_test“ im Software-Ordner des „refdesign“

```
$ make software SOFTWARE_SRC=uart_test
```

- Anschließen des USB to UART Adapter, wie in der `README` im Ordner `sw/uart_test` des ParaNut-Repositories beschrieben.
- Öffnen von Picocom oder ähnlicher Software in einer neuen Konsole. Hierbei zu beachten, dass der richtigen USB-Port auswählen wird. In „/dev/ttyUSB*“ muss das Symbol „*“ durch die Zahl des USB Geräts ersetzt werden.

```
$ sudo picocom -b 115200 /dev/ttyUSB*
```

- Ausführen des Programm auf der Hardware:

```
$ make run
```

Nun ist zu sehen, dass über Picocom, wie erwartet, keine Ausgabe erfolgt. Siehe Screenshot in Abbildung A.3

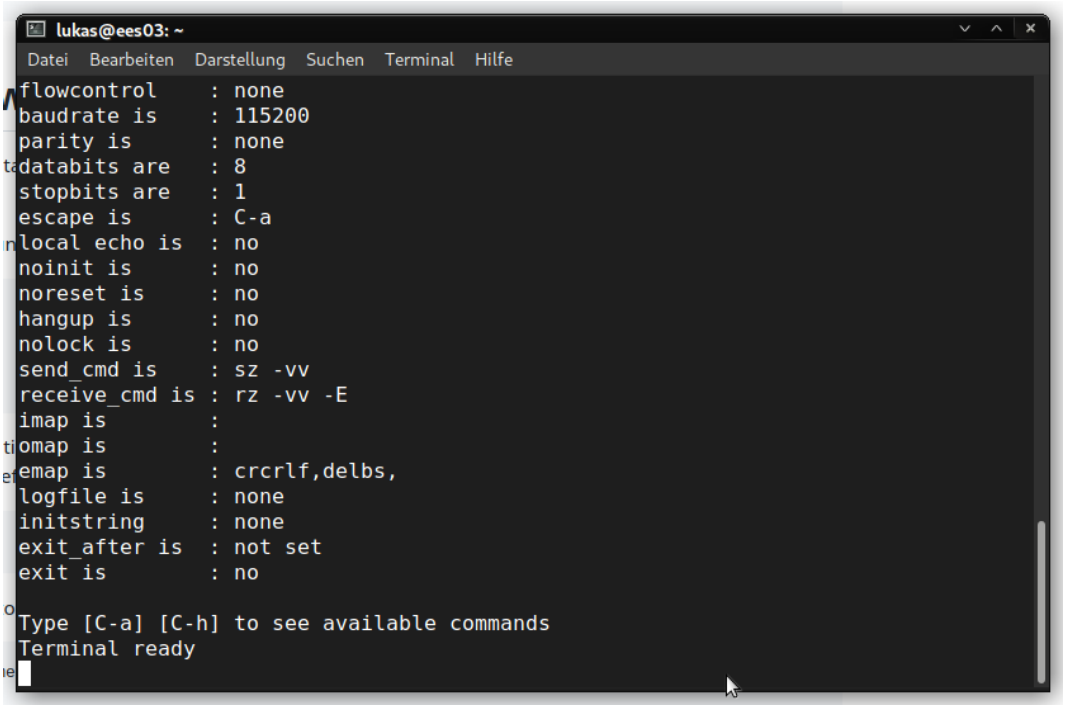

Abbildung A.3: Picocom Ausgabe bei Internen UART

4. Überprüfen des gebauten UART in Vivado:

   - Wechseln in den Ordner mit der `system.xpr` vom Wurzelverzeichnis des ParaNut-Repositories

     ```
     $ cd systems/refdesign/hardware/build
     ```

   - Öffnen der Datei `system.xpr`

     ```
     $ vivado system.xpr
     ```

   - Auf „Open Block Design“ drücken wie in Abbildung A.5 sichtbar

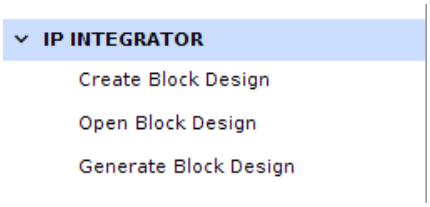

Abbildung A.4: Öffnen des Blockdesigns

- Dieses sollte wie in Abbildung A.5 aussehen. Zu sehen ist, dass die Leitungen „rx“ und „tx“ des ParaNut Blocks nicht verbunden sind. Wie es in diesem Testabschnitt geschehen soll.

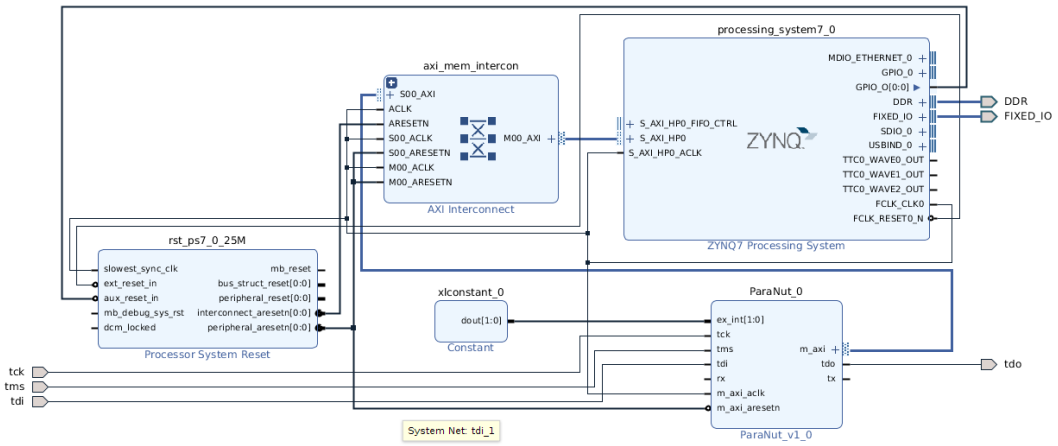

Abbildung A.5: Blockdesign für Internen UART

Erwartete Ergebnisse:

- Synthese des ParaNut konnte erfolgreich durchgeführt werden
- Keine Ausgabe in Picocom und der ParaNut Konsole
- UART ist in Vivado nicht extern verbunden

A.12.4 Test auf Hardware: Externer UART

Auszuführende Schritte:

1. Aktivieren des UART-Moduls für das entsprechende Board. Wenn dies noch nicht in Kapitel A.12.3 geschehen ist.
   - In `system/refdesign/hardware/boards/<Boardtyp>/config.mk` im ParaNut-Repository den Parameter `CFG_UART_ENABLE` auf `true` setzen
2. Bauen eines ParaNuts mit externen UART:
   - Sourcen der `settings64.sh` von Vivado, wenn dies noch nicht in Kapitel A.12.3 geschehen ist.

```
$ source /<Pfad zu Xilinx Vivado>/Xilinx/Vivado/2019.1/settings64.sh
```

- Sourcen der Datei `settings.sh` im Wurzelverzeichnis des ParaNut-Repositories, wenn dies noch nicht in Kapitel A.12.3 geschehen ist

```
$ source settings.sh
```

- Wenn der Test in Kapitel A.12.3 bereits durchgeführt wurde, muss die synthetisierte Hardware gelöscht werden. Dazu muss in den Ordner des „refdesign“ im ParaNut Repository gewechselt werden.

```
$ cd systems/refdesign
```

```
$ make veryclean
```

- Die Synthese für den externen UART starten.

```
$ make build UART_EXT=1
```

3. Überprüfung des gebauten ParaNuts:

- Bauen des Programm „uart_test“ im Software-Ordner des „refdesign“

```
$ make software SOFTWARE_SRC=uart_test
```

- Anschließen des USB to UART Adapter, wie in der `README` im Ordner `sw/uart_test` des ParaNut-Repositories beschrieben, wenn dies noch nicht in Kapitel A.12.3 geschehen ist.
- Öffnen von Picocom oder ähnlicher Software in einer neuen Konsole, wenn dies noch nicht in Kapitel A.12.3 geschehen ist. Hierbei zu beachten, dass der richtigen USB-Port auswählen wird. In „/dev/ttyUSB*“ muss das Symbol „*“ durch die Zahl des USB Geräts ersetzt werden.
- Ausführen des Programm auf der Hardware:

```
$ make run
```

Nun ist zu sehen, dass Picocom eine Ausgabe des Alphabets in Großbuchstaben ausführt. Siehe Screenshot in Abbildung A.6

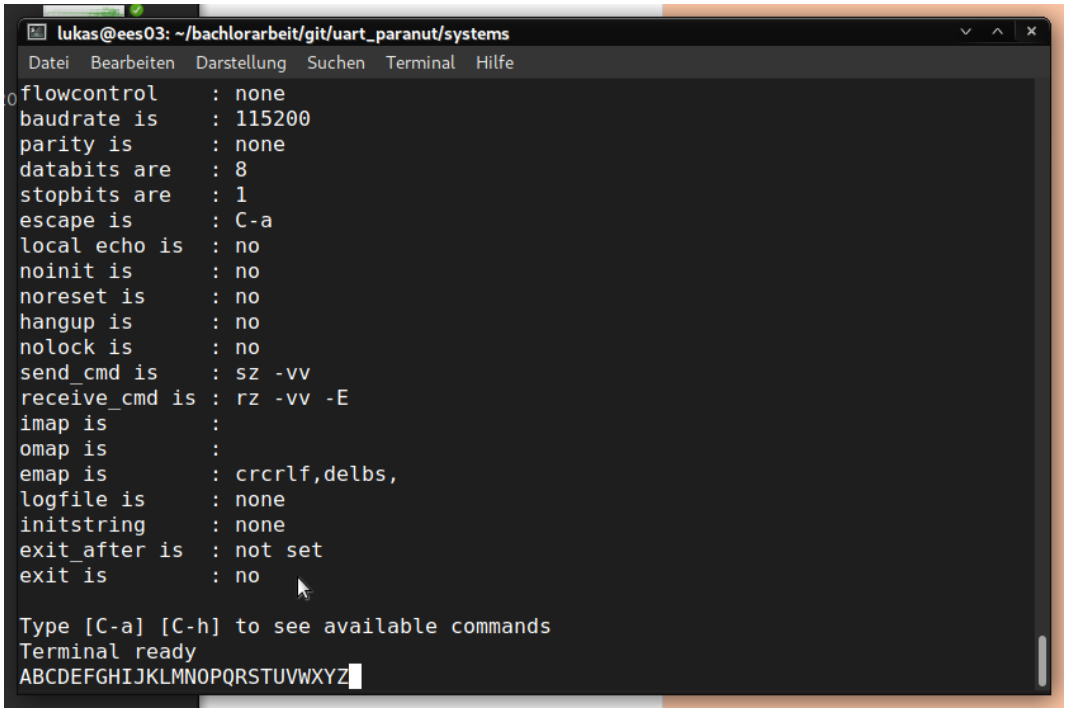

Abbildung A.6: Picocom Ausgabe bei Externen UART

- Wenn nun Text in Picocom eingegeben wird, ist dort und in der ParaNut Konsole eine Ausgabe zu sehen. Dies ist in den Abbildungen A.7 und A.8 anhand der Eingabe „Hallo“ zu sehen.

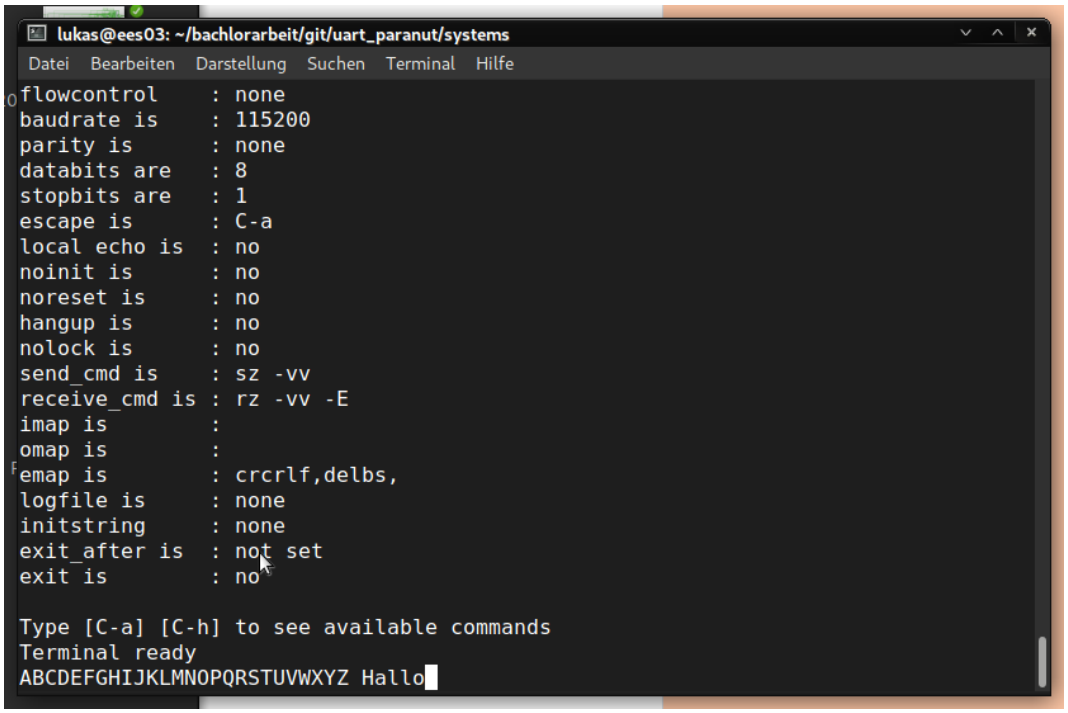

Abbildung A.7: Picocom Ausgabe nach Eingabe von Text bei Externen UART

Abbildung A.8: ParaNut-Konsole nach Eingabe von Text bei Externen UART

4. Überprüfen des gebauten UART in Vivado:

   - Wechseln in den Ordner mit der `system.xpr` vom Wurzelverzeichnis des ParaNut-Repositories

     ```
     $ cd systems/refdesign/hardware/build
     ```

   - Öffnen der Datei `system.xpr`

     ```
     $ vivado system.xpr
     ```

   - Auf „Open Block Design“ drücken wie in Abbildung A.12 sichtbar

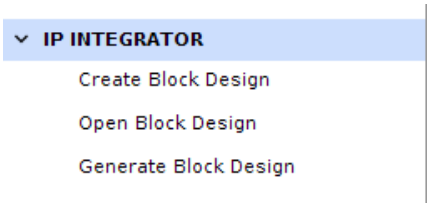

Abbildung A.9: Öffnen des Blockdesigns

   - Dieses sollte wie in Abbildung A.12 aussehen. Dort ist zu sehen, dass die Leitungen „rx“ und „tx“ des ParaNut Blocks verbunden sind. Wie es in diesem Testabschnitt geschehen soll.

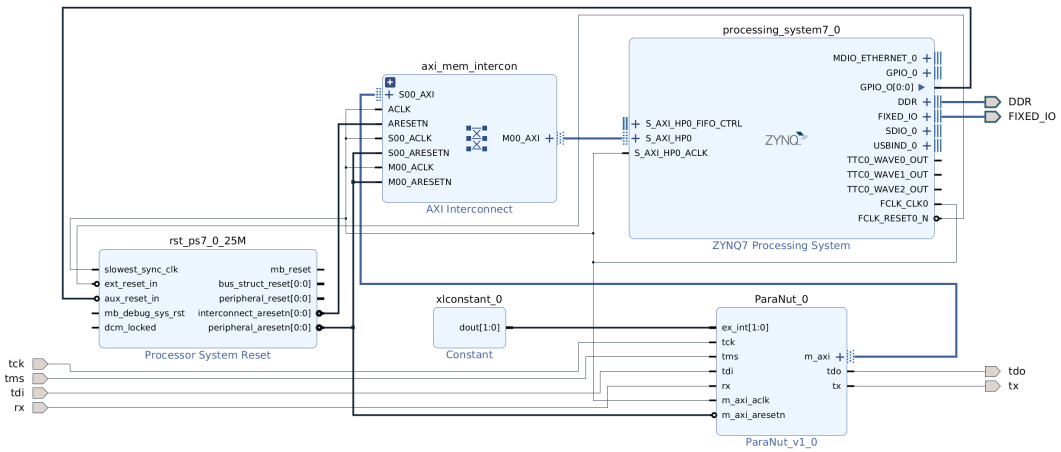

Abbildung A.10: Blockdesign für Externen UART

5. Überprüfen ob UART-Modul im IP-Core eingebaut ist.

   - Navigation zurück in den „refdesign“ Ordner des ParaNut-Repositories. Wenn sich die Konsole noch im Ordner `systems/refdesign/hardware/build` des ParaNut Repositories befindet, ist dies mit dem folgenden Befehl

     ```
     $ cd ../../
     ```

   - Anschließend wird mit dem folgenden Befehl in den IP-Core des ParaNut navigiert

     ```
     $ cd viavao_cores/PARANUT
     ```

   - Öffnen des „Elaborated Design“, wie in Abbildung A.11 zu sehen.

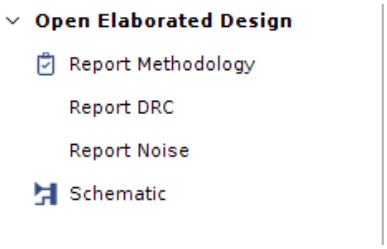

Abbildung A.11: Öffnen des Elaborated Design

   - Öffnen des Blocks „paranut_inst“ und Suche nach dem Signal „rx“, am linken Rand. Dieses sollte wie in Abbildung A.12 gezeigt intern verbunden sein.

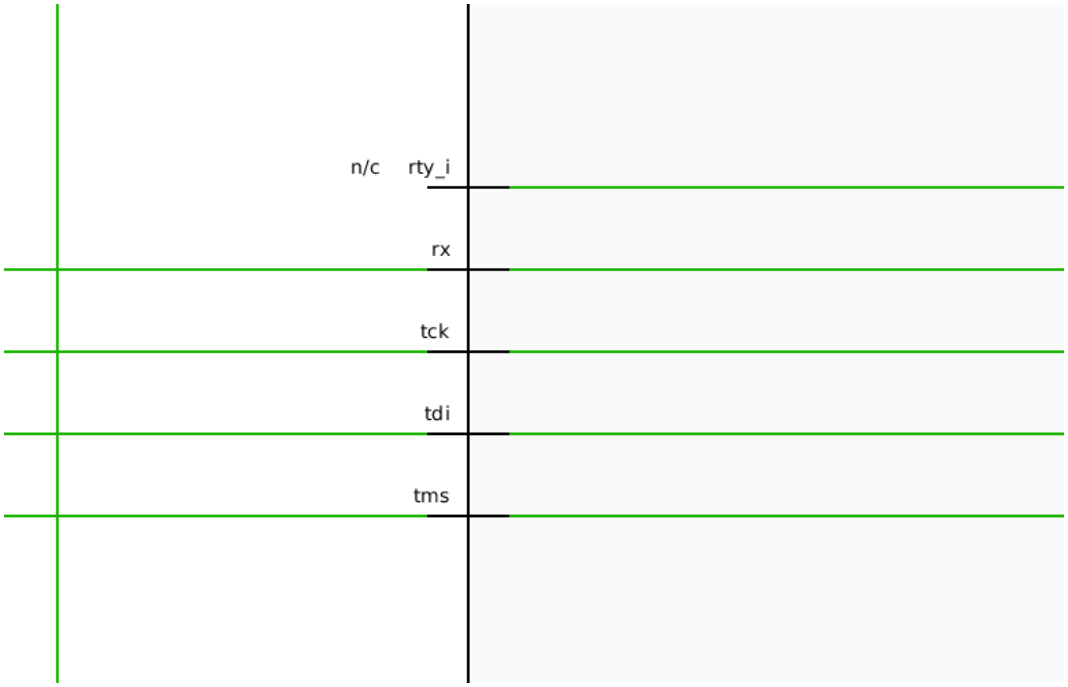

Abbildung A.12: Elaborated Design für aktiven Externen UART

Erwartete Ergebnisse:

- Synthese des ParaNut konnte erfolgreich durchgeführt werden
- Ausgabe in Picocom und der ParaNut Konsole
- Beim Tippen in Picocom werden die Zeichen zurückgegeben und in der ParaNut Konsole angezeigt
- UART ist in Vivado extern verbunden
- Die RX Leitung ist im IP-Core des ParaNut verbunden

A.12.5 Bauen eines ParaNut Cores ohne UART

Auszuführende Schritte:

1. Deaktiveren des UART-Moduls für das entsprechende Board.
   - In `system/refdesign/hardware/boards/<Boardtyp>/config.mk` im ParaNut-Repository den Parameter `CFG_UART_ENABLE` auf `false` setzen
2. Bauen eines ParaNut IP-Cores:

- Sourcen der `settings64.sh` von Vivado, wenn dies noch nicht in Kapitel A.12.4 geschehen ist.

```
$ source /<Pfad zu Xilinx Vivado>/Xilinx/Vivado/2019.1/settings64.sh
```

- Sourcen der Datei `settings.sh` im Wurzelverzeichnis des ParaNut-Repositories, wenn dies noch nicht in Kapitel A.12.4
- Navigieren in den refdesign Ordner des ParaNut Repositories

```
$ cd systems/refdesign
```

- Bauen eines IP-Cores des ParaNut

```
$ make paranut_core
```

3. Überprüfen der Ergebnisse:

- Wechsel in den IP-Core des ParaNut mit dem folgenden Befehl

```
$ cd viavao_cores/PARANUT
```

- Öffnen des „Elaborated Design“ wie in Abbildung A.13 zu sehen.

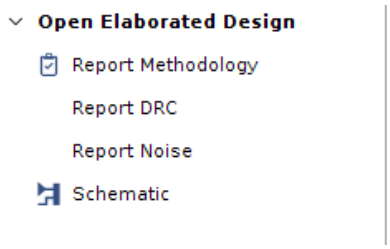

Abbildung A.13: Öffnen des Elaborated Design

- Öffnen des Block „paranut_inst“ und Suche nach dem Signal „rx“, am linken Rand. Dieses sollte wie in Abbildung A.14 gezeigt intern nicht verbunden sein.

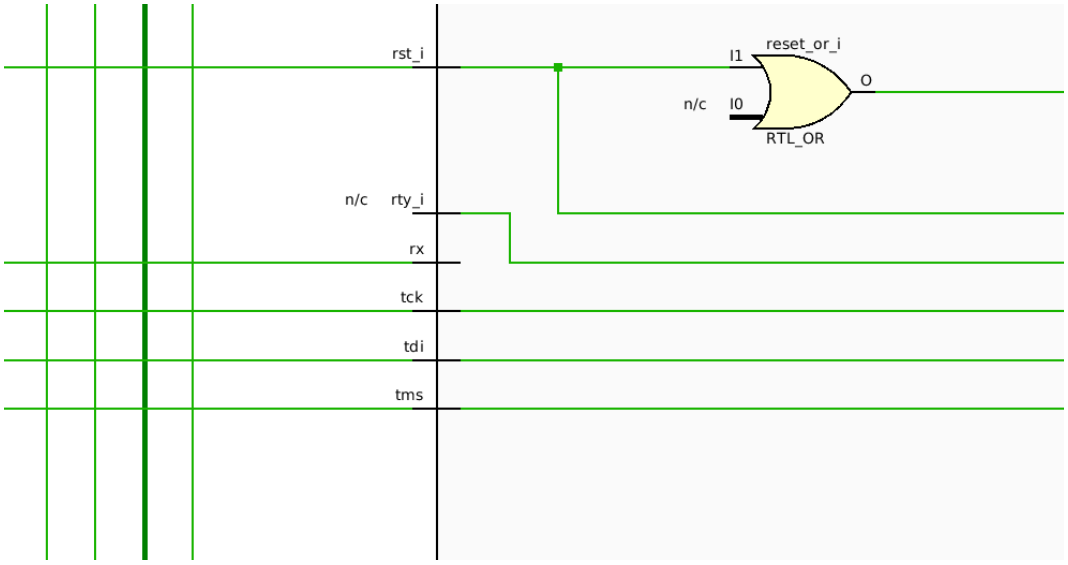

Abbildung A.14: Elaborated Design für deaktivierten Externen UART

Erwartete Ergebnisse:

- Die „rx“ Leitung ist im IP-Core des ParaNut nicht intern verbunden

A.12.6 High-Level-Synthese mit ICSC

Auszuführende Schritte:

1. Installation des ICSC Tools und sv2v:
   - Damit die Synthese mit dem ICSC Tool möglich ist, muss dieses gedownloadet werden und gebaut werden. Die `README` hierfür ist im Ordner `hw/sysc/uart` des ParaNut Repositories zu finden.
   - Auch sv2v muss wie in der `README` im Ordner `hw/sysc/uart` des ParaNut Repositories beschrieben, installiert werden.
2. Ausführen der ICSC HLS:
   - Es ist sicherzustellen, dass die Datei `settings.sh` aus dem Wurzelverzeichnis des ParaNut Repositories gesourced ist. Im Wurzelverzeichnis wird dazu der folgenden Befehl ausgeführt

     ```
     $ source settings.sh
     ```

- Wechseln in den Ordner `hw/sysc/uart` des ParaNut Repositories

```
$ cd hw/sysc/uart
```

- Ausführen der HLS mit dem folgenden Befehl

```
$ make update_uart
```

3. Betrachten des Ergebnisse:
   - Überprüfen ob im Ordner `hw/sysc/uart` des ParaNut Repositories, der Ordner `build` generiert wurde.
   - Auch sollte die Datei `UARTModule.v` im Ordner `hw/vhdl/paranut` aktualisiert worden sein.

Erwartete Ergebnisse:

- Die Synthese ist ohne Abbruch durchgelaufen
- Die `UARTModule.v` Datei ist vorhanden und aktualisiert