

Implementierung eines RISC-V CPU mit

Teilen des RISC-V External Debug

Support

**Ein Bericht zur Auswahl, der Erstellung und dem Einsatz von
Tools und der Durchführung der Implementierung**

Seminararbeit zur Erlangung des akademischen Grades

Bachelor of Engineering

im Studiengang Elektro- und Informationstechnik
an der Fakultät Ingenieurwissenschaften und Informatik
der Technischen Hochschule Aschaffenburg

Vorgelegt von:
Wolfgang Bischoff
Matrikelnummer:
2228538

Prüfer/Prüferin:
Prof. Dr. Müller
Abgabedatum:
22. Juni 2025

Erklärung zur Abschlussarbeit

Hiermit versichere ich, dass die eingereichte Ausarbeitung von mir persönlich verfasst und meine eigene individuelle Prüfungsleistung ist.

Ich versichere, dass die Ausarbeitung weder vollständig noch in Teilen durch sogenannte künstliche Intelligenz (KI) dergestalt erstellt wurde, dass das KI-Werk bzw. KI-Werkeile meine eigene Prüfungsleistung ersetzen. Der Einsatz von Rechtschreibüberprüfprogrammen ist nicht als unzulässiger KI-Einsatz im vorgenannten Sinne zu verstehen. Ich versichere, KI allenfalls eingesetzt zu haben, um einen von KI für meine Aufgabenstellung ausgearbeiteten Lösungsvorschlag kritisch zu beurteilen und/oder einen Überblick über Aspekte zu erhalten, die für die von mir in Eigenleistung zu erbringende Prüfungsleistung relevant sein könnten. Soweit durch die Aufgabenstellung bzw. Hinweise der Prüfenden der Einsatz von KI vorgegeben ist oder KI von mir aus Eigenantrieb, bspw. als Formulierungshilfe oder Bild-, Grafikerstellungs-Tool, etc. eingesetzt wurde, sind die von KI erzeugten Werkeile von mir in der Arbeit entsprechend gekennzeichnet. Mir ist bewusst, dass die von KI erzeugten Werkeile auf ihre Validität zu überprüfen und nicht zitierfähig sind. Mir ist bewusst, dass diejenigen Teile der Arbeit, die automatisch oder halbautomatisch, insbesondere mit den Systemen der sogenannten KI, individuell generiert wurden (wie Einleitungen, Definitionen, Synopsen, etc.), vom Prüfer als Prüfungsleistung ausgeschlossen werden können.

Ebenso versichere ich, dass diese Arbeit oder Teile daraus weder von mir selbst noch von anderen als Leistungs nachweise andernorts eingereicht wurden.

Wörtliche oder sinngemäße Übernahmen aus anderen Schriften und Veröffentlichungen in gedruckter oder elektronischer Form sind gekennzeichnet. Sämtliche Literatur und sonstige Quellen sind nachgewiesen und im Literatur- und Quellenverzeichnis aufgeführt. Das Gleiche gilt für graphische Darstellungen und Bilder sowie für alle Internet-Quellen.

Ich bin ferner damit einverstanden, dass meine Arbeit zum Zwecke eines Plagiatsabgleichs in elektronischer Form anonymisiert versendet und gespeichert werden kann.

Ort, Datum

Unterschrift des Verfassers / der Verfasserin

Der Veröffentlichung der Master-/Bachelorarbeit in der Bibliothek der Technischen Hochschule Aschaffenburg wird [nicht] zugestimmt.

Ort, Datum

Unterschrift des Verfassers / der Verfasserin

Abstract

Das Ziel dieser Studienarbeit ist es, einen einfachen RISC-V CPU zu erstellen. Das Testen der CPU durch selbstentwickelte Software und das Debuggen der Software auf dem CPU ist ebenfalls Teil der gesetzten Aufgabe. Dazu soll nicht nur ein RISC-V CPU entwickelt werden, sondern es sollen auch Tools bereitgestellt werden, mit denen sich die obigen Aufgaben ausführen lassen. Die FPGA Entwicklungswerkzeuge werden vorgestellt, bewertet und schließlich ausgewählt.

Inhaltsverzeichnis

1. Zielsetzungen und Herausforderungen.....	1
2. Einleitung zu RISC-V.....	4
3. Auswahl der Hardware und Software.....	6
4. Erstellen des Assemblers.....	8
5. RISC-V External Debug Support (Debug Spezifikation).....	13
JTAG State Machine.....	16
Implementierung in Verilog.....	18
RISCV Abstract Commands.....	19
Zusammenfassendes Beispiel.....	19
Debug Probe und Python Skripte.....	20
6. Abschluss und Ausblick.....	21
7. Danksagung.....	22

Abbildungsverzeichnis

Abbildung 1: Inhaltsverzeichnis aktualisieren (1)	4
Abbildung 2: Inhaltsverzeichnis aktualisieren (2)	4
Abbildung 3: Überschrift (Ebene 1) in Kopfzeile automatisch anzeigen (1)	8
Abbildung 4: Überschrift (Ebene 1) in Kopfzeile automatisch anzeigen (2)	9
Abbildung 5: Abbildung beschriften (1)	10
Abbildung 6: Abbildung beschriften (2)	11
Abbildung 7: Formatvorlagen (1)	11
Abbildung 8: Formatvorlagen (2)	12
Abbildung 9: Abschnittsumbrüche	13
Abbildung 10: Formatierungssymbole anzeigen	13

Abkürzungsverzeichnis

Abkürzung	Bedeutung
DTM	Debug Transport Module
DM	Debug Module
DMI	Debug Module Interface

Formelverzeichnis

Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.

Tabellenverzeichnis

Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.

1. Zielsetzungen und Herausforderungen

Das Ziel dieser Studienarbeit ist es, einen einfachen RISC-V CPU zu erstellen. Das Testen der CPU durch selbstentwickelte Software und das Debuggen der Software auf dem CPU ist ebenfalls Teil der gesetzten Aufgabe. Dazu soll nicht nur ein RISC-V CPU entwickelt werden, sondern es sollen auch Tools bereitgestellt werden, mit denen sich die obigen Aufgaben ausführen lassen. Die FPGA Entwicklungswerkzeuge werden vorgestellt, bewertet und schließlich ausgewählt.

Die Implementierung einer RISC-V CPU stellt einen Studenten bzw. Anfänger vor mehrere handfeste Herausforderungen.

1. Auswahl der Hardware - Ein FPGA Entwicklungsboard kann von der Bildungseinrichtung beigestellt werden und die Auswahl ist damit vorgegeben. Es kann trotzdem sinnvoll sein sich nochmals über die Wahl des Entwicklungsboards Gedanken zu machen, da die Auswahl des Entwicklungsboards Einfluss auf die zu verwendende FPGA-Entwicklungstools hat. Die Verwendung bestimmter FPGA-Entwicklungstools haben selbst wiederum Konsequenzen und die Auswahl sollte auf einem sinnvollen Wissensstand erfolgen. Ein weiterer Punkt ist der Preis und die Verfügbarkeit der FPGA Boards. Es gibt Boards zu bereits niedrigen Kosten, die sich für das Erlernen der FPGA-Entwicklung sehr gut eignen und aufgrund ihrer Softwaretools sogar bestimmte Vorteile gegenüber sehr kostspieligen Boards haben.

-> Die Lösung in dieser Studienarbeit wird es sein, die Integration auf einem professionellen Board (Arty S7) ausgeführt, während schnelle Tests auf einem GOWIN Tang Nano 9k erfolgen.

2. Auswahl der FPGA-Entwicklungstools - Die FPGA Entwicklungstools können vom Hersteller des FPGA gestellt werden. Diese Herstellertools sind im Falle von Vivado (Vom Hersteller AMD für Artix, Kintex, Virtex, Spartan und Zynq 7000 series und auch UltraScale and UltraScale+ Familien) und Quartus Prime (Intel für Cyclone und MAX chips) sehr umfangreich. Vorteile von Intel und AMD sind die

Bereitstellung von IP-Bibliotheken und Generatoren. Nachteile sind die schiere Größe dieser Tools (mehrere zig GB an Festplatten für die Installation sind keine Seltenheit). Ein mittleres Angebot wird von kleineren Herstellern wie der Firma Lattice und der Firma GOWIN gemacht. Ihre FPGAs sind teilweise zusätzlich auch durch OpenSource Toolchains abgedeckt aber die Herstellereigneten FPGA-Entwicklungstools sind auch eine gute Option und bieten oft einen guten Funktionsumfang bei kleinerem Resourcenverbrauch. Des Weiteren geht die Bandbreite bis hin zu OpenSource Tools (Yosys, NextPNR) am unteren Ende des Resourcenverbrauchs aber leider oft auch der Funktionalität. Die Toolchains werden in einem späteren Abschnitt miteinander verglichen und eine Auswahl wird getroffen.

-> Die Lösung wird es sein, Vivado für das Endprodukt und die GOWIN FPGA Designer Software für schnelle Tests verwendet wird.

3. Erlernen der HDLs Verilog, System Verilog oder VHDL. Ausführen von Simulationen und übertragen der Designs auf den FPGA und dortigem Test. Es kann sein, dass Designs trotz funktionierender Simulation nicht auf dem FPGA funktionieren. Die Fehlersuche ist dann sehr schwer, da es keine "Debugger" für FPGAs und deren internen Abläufe gibt. Im schlimmsten Fall kann das Design überhaupt kein Lebenssignal von sich geben, obwohl es in der Simulation korrekt Signale ausgegeben hat. Von Vorteil ist es, wenn die Dauer eines Zyklus von der Änderung des Designs über die Synthese bis zum Upload des Bitstreams kurz sind. Das Verständnis vom Design Digitaler Systeme, also Signale (wires) und getakteter, Zustandsbehafteter Logik (Register mit Clock-Flanken) ist ein Erfahrungswert, den ein Anfänger nicht intuitiv besitzt. Die Fähigkeit ein digitales System zu erstellen weicht von der Fähigkeit in einer Programmiersprache oder Assembler zu programmieren ab. Das digitale System muss in den meisten Fällen als Gesamtkonstrukt betrachtet werden müssen, während ein Computerprogramm in einzelne Anweisungsfolgen aufteilbar ist. Der FPGA Entwickler muss einen großen Kontext gleichzeitig in seinem Denkprozess berücksichtigen, während der Anwendungsprogrammierer nach Möglichkeit das "Teile und Herrsche" Prinzip anwenden kann. Ohne Erfahrung verliert sich ein

Anfänger in der Komplexität und kann nicht mehr entscheiden woher ein Fehlverhalten kommt.

-> Die Lösung für dieses Problem ist es sich an das Lehrbuch von Harris & Harris zu halten.

Auch nachdem die Aufgabe des Designs eines CPU gemeistert ist, gibt es weitere Herausforderungen.

4. Auswahl eines Assemblers - Es gibt wenige Assembler, die RISC-V Code in Maschinenbefehle übersetzen und dabei möglichst wenige Annahmen treffen.

Das Problem liegt in einer Diskrepanz zwischen der RISC-V Spezifikation und Lehrbüchern zu RISC-V und den Compilern und Assemblern, die tatsächlich zur Erstellung von Software genutzt werden. Die RISC-V Spezifikation und Lehrbücher sprechen über RISC-V Instruktionen und deren Kodierung in Maschinenbefehle ohne dabei ein konkretes Betriebssystem als Zielumgebung zu adressieren. Wirkliche Compiler haben die Aufgabe Maschinencode auszugeben, der konkrete Speicherbereiche belegt und die Kombination von unterschiedlichen Objektdateien in ein kombiniertes Programm ermöglicht. Für einen Anfänger kann diese Diskrepanz zu einer zusätzlichen Herausforderung werden, für die er nicht die notwendige Zeit oder Motivation hat. Ein echter Compiler benötigt beispielsweise noch ein Linkerscript bevor er arbeitet. Das Produkt verwendet spezielle Quellcodeabschnitte, die bestimmte Datenstrukturen initialisieren. Wenn man eine RISC-V CPU entwirft, möchte man eventuell lediglich simple Testprogramme von Adresse 0x00 ausführen und legt keinen Wert auf Adressbereiche oder Initialisierung bestimmter Daten.

-> Die Lösung für dieses Problem wird es sein einen Assembler für 32-bit RISC-V bereitzustellen, mit dem schnell, unkompliziert und ohne Annahmen RISC-V Assemblercode in Maschinenbefehle für den Upload in den CPU übersetzt werden kann. Der erstellte Maschinen-Code führt das Assembler-Programm dann exakt aus, ohne jegliche Änderungen oder Annahmen für ein Zielsystem zu treffen.

5. Interaktion mit dem CPU - Wenn das Design über den Bitstream auf den FPGA bereitgestellt worden ist, wird der CPU ein Programm ausführen. Die Frage ist, wie man jetzt mit dem CPU interagiert. Diese Thema wird in den Lehrbüchern zu digitalem Design nicht adressiert. Das Ziel ist es nicht Peripherie wie Druckknöpfe, LEDs oder LCDs an den CPU anschließen, sondern mit dem CPU selbst zu kommunizieren. Nach Möglichkeit soll der CPU gestoppt und gestartet werden können. Der Zugriff auf Register und den Speicher soll möglich sein. Dazu gibt es die Debug-Spezifikation für RISC-V (TODO: Referenz auf eine bestimmte Version der Debug-Spezifikation hier einfügen! file:///C:/Users/laptop/dev/fpga/gowin_tang_nano/riscv_debug/doc/riscv-debug-release.pdf). Die Debug-Spezifikation beschreibt Komponenten um mit den Harts eines RISC-V CPU zu interagieren. Im RISC-V Umfeld wird eine Recheneinheit als Hart bezeichnet. Ein Hart führt einen Thread aus. Ein CPU kann beliebig viele Harts enthalten. Die Kommunikation kann über das JTAG-Protokoll ausgeführt werden.

-> Die Lösung in dieser Studienarbeit wird es sein einen Teil der RISC-V Debug Spezifikation implementiert, gerade genug um die oben geforderten Punkte über eine JTAG DebugProbe auszuführen. Die DebugProbe wird auf einem Arduino DUE ausgeführt, der direkt mit dem FPGA Entwicklungsboard verbunden ist.

Das Ihnen vorliegende Dokument dient als Vorlage für wissenschaftliche Seminar- und Abschlussarbeiten.

2. Einleitung zu RISC-V

RISC-V ist eine Instruction Set Architecture (ISA). Ein Instructionset Architecture ist eine Menge an Anweisungen, Registern und Datentypen. Damit beschreibt RISC-V keine konkrete CPU Architektur sondern gibt Befehle und Register vor, die auf einer konkreten Architektur ausführbar und vorhanden sein müssen, damit dieses System RISC-V konform ist und RISC-V Befehle ausführen kann. Laut der RISC-V homepage (Quelle: [> about > faq > Who uses an ISA?](https://riscv.org)) ist

eine ISA auch im Zuge der Leere einsetzbar damit Studenten die Wirkungsweiße eines Computers verstehen lernen. RISC-V wurde daher auch vor dem Hintergrund der Lehre erstellt.

Damit steht RISC-V in Konkurrenz mit der x86 ISA (Chips von Intel, AMD und weiteren), der ARM ISA (Chips von STM32, Renesas, NXP, Qualcomm und andere) und allen anderen definierten Instructionsets (AVR, Itanium (IA-64), MIPS, Z-80 und viele weitere). Die Besonderheit von RISC-V ist das Lizenzmodell. Das Lizenzmodell besteht aus der Erlaubnis zur freien Benutzung und damit der Abwesenheit jeglicher Lizenzkosten (Quelle: <https://riscv.org/about/faq/#:~:text=There%20is%20no%20fee%20to%20use%20the%20RISC%20V%20ISA>. In Google eingeben: <https://riscv.org> > about > faq > What is the license model).

RISC-V ist an der Universität UC Berkeley entwickelt worden. Dabei ist RISC-V aus dem Wissenschatz erstellt worden, den die Beteiligten im Zuge der vier Vorgängeprojekte (RISC-I bis RISC-IV) sammeln konnten und heißt damit sinnigerweise RISC-V (Quelle: <https://riscv.org> > about > faq > Why is it called RISC-V).

RISC-V hat in den letzten Jahren weitere Fortschritte gemacht. Das Triple-A Spiel "The Witcher 3" lässt sich mit Einschränkungen auf einem RISC-V System (<https://milkv.io/pioneer>) ausführen. RISC-V Mainline Support wurde Linux im Jahre 2022 hinzugefügt. Die 64 Bit Variante von RISC-V hielt im Jahre 2023 Einzug in den Linux-Kernel (Quelle: <https://en.wikipedia.org/wiki/RISC-V>)

Die RISC-V ISA ist keine monolithische Spezifikation. Das angewandte Prinzip ist, dass ein Kern von Funktionen für eine gewisse Bit-Breite (XLEN) vorhanden sein muss, der dann durch Extensions erweitert werden kann. Beispiele für Kerne sind RV32I, RV64I und RV128I die Integer-Operationen mit jeweiligen Registerbreiten von 32, 64 und 128 Bit unterstützen. Extensions fügen dem System Floating-Point Unterstützung, komprimierte Anweisungen, Vector-Mathematik, Kryptografie, Atomic-Operationen und viele weitere Funktionen hinzu.

In dieser Studienarbeit wird ein Teil des Kerns RV32I umgesetzt. Dieser Umfang kann durch weitere Entwicklungstätigkeit immer weiter bis zum vollen Umfang von RV32I ergänzt werden, falls notwendig. Des Weiteren wird ein Teil des RISC-V External Debug Support hinzugefügt.

3. Auswahl der Hardware und Software

Als Optionen für die Entwicklungsboards werden der Lattice iCEstick, das GOWIN Tang Nano 9k und das Digilent Arty S7 25T mit ihren jeweiligen Entwicklungsumgebungen betrachtet. Die Software soll auf einem Windows-Laptop ausführbar sein.

Die Kriterien zusammen mit ihren Gewichtungen zur Bewertung sind:

- Größe des FPGA (Gewichtungsfaktor 3) - Passt ein RISC-V Design auf den FPGA?
- Formfaktor (Gewichtungsfaktor 1) - (Kann man es leicht in einem Rucksack mitnehmen z.B. zwischen Hochschule und Wohnung)
- Peripherie (Gewichtungsfaktor 3) - Wieviel Peripherie hat es (PMODs, LEDs, Breakout Pins).
- Anschaffungskosten (Gewichtungsfaktor 1) - Sind die Anschaffungskosten für den Anwendungszweck gerechtfertigt.
- Freie Software (Gewichtungsfaktor 5) - Lässt sich der FPGA überhaupt mit Herstellersoftware programmieren
- Installationsplattenplatz - (Gewichtungsfaktor 2) - (100-te GB sind schlecht!)
- Geschwindigkeit der Synthese - (Gewichtungsfaktor 1) - allgemeine Performance des Tools
- Fehlererkennung der Synthese - (Gewichtungsfaktor 5) - Beispielsweise werden logische Loops werden von Yosys nicht erkannt! Von professionellen Tools aber schon)

- Editor-Qualität - (Gewichtungsfaktor 1) - welche Funktionen bietet der Editor

Es wird mit Punkten von 1 bis 5 bewertet, wobei mehr Punkte eine bessere Qualität bedeuten.

	iCEstick, Yosys Toolchain	GOWIN Tang Nano 9k, FPGA Designer	Digilent ARTY S7 T25, Vivado
Kapazität FPGA	1	2	3
Formfaktor	5	5	4
Peripherie	2	3	4
Preis Board	3	5	3
Freie Hersteller-Software	1	4	4
Plattenplatz	5	5	2
Synthesedauer	4	5	2
Fehlererkennung	1	3	5
Qualität Editor	3	4	3

Die Verwendung des Lattice iCEstick ist für einen Anfänger zu kompliziert, da kritische Fehler vom Werkzeug (Yosys) nicht automatisch erkannt werden tappt ein Anfänger zu leicht in die Falle Quellcode zu schreiben, der zwar synthetisiert wird dann aber auf dem FPGA dann nicht funktioniert. Das erschwert Erstellen von Speicher führt dazu, dass ein Anfänger daran scheitert einen RISC-V CPU einfach zu Erstellen, der auf den iCEstick passt. Eine detaillierte Einarbeitung in Yosys unter würde diese Situation wahrscheinlich entschärfen. Im Fernstudium steht diese Option allerdings nicht zur Wahl. Es wird Anfängern mit einem knappen Zeitrahmen nicht empfohlen, mit Yosys zu beginnen. Der iCEStick wird nach der Gewichtung in Summe mit 44 Punkten bewertet.

Für das Tang Nano 9k Board zeigt sich, dass die Arbeit mit dem GOWIN Tang Nano 9K Board am ehesten der Arbeit mit einem Arduino gleicht. Die IDE ist simpel und erste Verilog Module lassen sich schnell umsetzen und auf das Board laden. Die Synthese erfolgt sehr schnell, was die Wartezeiten gering hält. Dabei erkennt die Synthese viele relevante Fehler und gibt Warnungen und Fehler aus! Die IDE hat eine Bibliothek für IP-Cores, um z.B. einfach Speicher synthetisieren zu lassen. Es gibt einige gut dokumentierte Beispiele auf der Homepage des Produkts. Für einen Anfänger ist dieses Board gut geeignet, da er in der Lernphase sehr viele Fehler machen wird und das unkomplizierte und schnelle Deployment auf dem Tang Nano 9k damit einen großen Vorteil bietet. Die beschriebenen Designs lassen sich oft auf dem FPGA testen, ohne dass die Wartezeit ermüdend wirkt. Damit sinkt die Wahrscheinlichkeit, sich mit Designs und deren Simulation zu beschäftigen, die dann später nicht auf dem FPGA funktionieren. Das Tang Nano 9k Board wird in Summe mit 79 Punkten bewertet.

Das Digilent Arty S7 und die Vivado IDE sind im Vergleich zur GOWIN IDE langsamer, bieten dafür aber einen größeren Funktionsumfang. Vivado erkennt bereits bei der Synthese viele Fehler und warnt den Benutzer, bevor er Zeit mit einem Upload des Bit-Streams verschwendet. Die Auswahl an IP-Blöcken ist sehr umfangreich und deren Konfiguration wird durch eine Menüführung mit vielen Auswahloptionen unterstützt. Beispielsweise lassen sich damit Speicher und PLLs sehr einfach erstellen und werden im Design dann durch wirkliche Hardware im FPGA und nicht durch Logikblöcke ausgeführt. Das Digilent Arty S7 und Vivado werden in Summe mit 82 Punkten bewertet.

4. Erstellen des Assemblers

Um die korrekte funktionsweiße des RISC-V CPU zu testen, muss ein Programm im Instruction-Memory des CPU abgelegt werden, damit der CPU diese Anweisungen ausführen kann. Die Daten, die im Hauptspeicher erwartet werden sind die Anweisung der Maschinensprache. Die Umwandlung von

Assembler-Anweisungen in Maschinen Sprache wird Kodierung genannt. Ein Assembler hat die Aufgabe RISC-V Assembly Anweisungen zu Kodieren. Die RISC-V ISA definiert exakt, wie die Kodierung ablaufen muss.

Die Instruktionen des RV32I Kerns sind in sechs Gruppen unterteilt, die mit den Buchstaben R, I, S, B, U und J bezeichnet sind. Einige Instruktionen verwenden Register (Register, R), andere enthalten Parameter aus Daten, die direkt in die Instruction kodiert werden (Immediate, I). Andere Instruktionen sind für Verzweigungen auf Basis einer Bedingung (Branches, B) und Sprünge ohne Bedingung (Jumps, J) zuständig. Instruktionen, die in den Speicher schreiben, gehören der Store-Gruppe (S) an. Die U Instruktionen werden verwendet um 32 bit Werte in Register zu laden, was sich mit einem 32 Bit breiten Anweisungsformat gar nicht beschreiben lässt. U Instruktionen erlauben daher nur Immediate Wert aus 20 bit und füllen die fehlenden 12 bit während der Ausführung im CPU implizit mit 0 auf.

Um die Kodierung zu bewerkstelligen, wird im Zuge dieser Studienarbeit ein Assembler in Java erstellt. Dafür gibt es mehrere Gründe. Java lässt sich (nach der Installation eines JRE oder JDK) auf jedem Betriebssystem ausführen. Die Assembler und Compiler aus der GNU toolchain sind nicht ohne weiteres auf Windows ausführbar. Ein weiteres Argument ist, dass es ein simples Tool ohne viele Parameter geben sollte um Testprogramm schnell und einfach zu übersetzen. Echte Assembler sind komplizierter zu bedienen und verwenden immer auch ein spezifische Memory Layout für ein Zielsystem (Target) und fügen extra Instruktionen für den Target hinzu. Diese zusätzlichen Instruktionen werden von dem eigenen simplen CPU wahrscheinlich gar nicht unterstützt und das kodierte Programm kann dann nicht verwendet werden. Hier soll es lediglich um einen Assembler für einen sehr simplen RISC-V CPU gehen. Der Assembler kodiert die Anweisungen, bis auf leichte Optimierungen, so wie sie eingegeben worden sind.

Vor dem Entschluss, einen eigenen Assembler zu erstellen, wurde nach Alternativen gesucht. Es gibt einen Online Assembler unter <https://riscvasm.lucasteske.dev/>. Der Nachteil dieses Assembler ist es, dass er

64 bit RISC-V Instruktionen für RV64I ausgibt, ohne dass dies irgendwo auf der Seite erwähnt wird. Wenn man einen RISC-V CPU mit einer RV32I Kern erstellt, dann kann dieser Online-Assembler nur eingeschränkt verwendet werden. Wie bereits erwähnt, ist die RISC-V GNU Toolchain erst ab einem bestimmten Reifegrad des CPU sinnvoll. Damit wird der Anfang der Entwicklung durch die GNU Toolchain eher erschwert als unterstützt.

An den Assembler ist eine Emulator angeschlossen, der das kodierte Programm direkt ausführen kann. Damit kann die Gültigkeit des Programms erst lokal verifiziert werden, um zu verhindern, dass das Programm auf den CPU geladen wird, obwohl es an sich fehlerhaft ist. Wenn man ein fehlerhaftes Programm auf dem CPU ausführt, kann man in der Entwicklungsphase nicht unterscheiden, ob das fehlerhafte Verhalten von dem CPU oder von dem Programm stammt. In diesem Fall werden Fehler eventuell an Stellen gesucht, an denen es gar keine Fehler gibt. Diese Situation soll vermieden werden.

Der Assembler liest Dateien mit RISC-V Assembler Anweisungen ein. Ein Beispiel ist in Anlage A1 zu sehen.

Der Assembler benötigt das Label `_main` um den Haupteinsprungpunkt zu finden, damit er diese Adresse an den Emulator geben kann und der Emulator die Ausführung dort im Speicher beginnt. Für den RISC-V CPU auf dem FPGA sollte `_main` immer so gelegt werden, dass seine Adresse mit 0x00 zusammenfällt, also ganz am Anfang des Programms.

Die Assembler Instruktionen werden dann durch einen Parser basierend auf einer Grammatik für RISC-V Assembly geparsst. Der Weg über das Parsing wurde gewählt, da RISC-V Assembly ganze arithmetische Ausdrücke erlaubt. Ein Beispiel ist die Anweisung "jal ra, pc - 176;" Diese arithmetische Ausdrücke können beliebig kompliziert werden und sind nur schwer durch einen manuell entwickelten Parser zu verarbeiten. Erfahrungsgemäß wird der Parser dann letztendlich über lange Zeit fehlerbehaftet sein. Auf lange Sicht macht der Einsatz einer Grammatik Sinn und spart Zeit. Die Aufgabe wird an den Computer übergeben, der die Aufgabe besser erledigt als ein Mensch.

Das Ergebnis des Parsing ist ein Parse-Tree. Ein Visitor wird über den Parse-Tree geführt. Der Visitor (RISCASMExtractingOutputListener) befüllt einzelne Instanzen der Klasse AsmLine mit den Werten, die er aus den ParseTrees der RISC-V Assembler Zeilen ziehen kann. An dieser Stelle liegt jetzt die RISC-V Assembly Datei in Form einer Liste von AsmLine Objekten im Hauptspeicher vor.

Der nächste Schritt ist das Anwenden einer ersten einfachen Optimierung. Wenn sich im Quellcode lui und addi Anweisungen befinden, kann ein Paar dieser Anweisungen durch eine LI (Load Immediate) Pseudo Instruktion ersetzt werden. Dies erscheint zunächst kontraproduktiv, da ein Assembler eher Pseudoinstruktionen auflösen als erstellen soll. Es ergibt nur dann Sinn, wenn man weiß, dass zu einem späteren Zeitpunkt wieder Pseudoinstruktionen aufgelöst werden. Während der Auflösung hat der Assembler die Freiheit, die LI Instruktion zu optimieren und es wird eventuell anstelle des lui, addi Paars nur ein lui ausgegeben. Durch dieses Verhalten korrigiert der Assembler eventuell unnötig eingefügt Instruktionen durch den Programmierer. Diese Optimierung wurde übernommen, da andere Assembler das gleiche Verhalten haben und sich die Ergebnisse des eigenen Assemblers nur mit anderen Assemblern gegenprüfen lassen, wenn man diese Optimierung auch anwendet.

TODO: Füge eventuell Beispiel einer Optimierung ein:

Im Folgenden erstellt der Assembler eine Tabelle von .equ Konstanten mit deren Werten, damit er Konstanten in Parametern, Expressions und Labeln durch numerische Werte ersetzen kann.

TODO Füge eventuell .equ Beispiel ein, das die Ersetzung zeigt.

Nach dem Ersetzen der Konstanten werden Pseudo-Anweisungen aufgelöst. Beispielsweise wird ein NOP Pseudo-Anweisung durch ein ADDI Befehl ersetzt, wobei die ADDI Operation keinen Effekt haben darf. Es wird eine ADDI Instruktion ausgegeben, die den Wert 0 auf das ZERO Register addiert! Dieser Ablauf ändert absolut nichts am Zustand der CPU, außer, dass ein CPU Zyklus verbraucht wird und sich der Program Counter (PC) auf die folgende Anweisung verschiebt.

Nach dem Auflösen der anderen Pseudoinstruktionen wird nun auch LI aufgelöst. Dabei wird geprüft, ob die LI Anweisung ein Label als Immediate-Wert verwendet. Wenn ein Label verwendet wird, dann wird die Adresse des Labels ermittelt und die Differenz zwischen der aktuellen Adresse und der Label-Adresse gebildet. Wenn die Differenz mit einem simplen LUI geladen werden kann, dann wird nur ein LUI ausgegeben, ansonsten wird eine Kombination aus ADDI und LUI ausgegeben.

Jetzt wird der Call-Optimizer aufgerufen. Wenn eine Routine angesprungen wird, hat der Programmierer mehrere Möglichkeiten den Sprung zur Routine zu implementieren. Wenn die Routine innerhalb eines Megabyte relativ zur aktuellen Adresse zu finden ist (near call), dann reicht eine einzige JAL Instruktion. Wenn die Routine außerhalb des Megabyte zu finden ist (far call) dann wird eine Kombination aus AUIPC und JALR ausgegeben.

Nachdem alle Pseudo-Instruktionen durch echte Instruktionen ersetzt worden sind und alle Optimierer entschieden haben, ob sie ein oder zwei Instruktionen ausgeben möchten, liegt eine Liste mit ausschließlich "echten" Instruktionen vor. Alle Pseudo-Instruktionen wurden ersetzt. Auf dieser Liste werden jetzt relative Offsets ersetzt. Relative Offsets werden mit f und b und einem Label-Namen notiert. Der Assembler muss im Falle von f das erste folgende Label finden und im Falle von b das erste vorhergehende Label finden. Das gefundene Label wird dann durch seine Adresse ersetzt.

Siehe Beispiel aus Anlage A2 von
<https://marz.utk.edu/my-courses/cosc230/book/example-risc-v-assembly-programs/> in dem 1f und 1b verwendet wird. 1 ist hier ein Label, das doppelt verwendet wird, einmal in f-Richtung und einmal in b-Richtung aufgelöst werden muss.

Im Anschluss werden arithmetische Ausdrücke ausgerechnet und das Ergebnis in Form eines numerischen Werts in die Instruktion eingesetzt. Danach werden die Modifier HI() und LO() ersetzt, die das High-Byte oder das Low-Byte aus einem Wert herauslösen.

Der allerletzte Schritt ist es jetzt den Encoder auf jede AsmLine anzuwenden welcher MaschinenCode für die AsmLine ausgibt. Dieser Vorgang wird für alle AsmLines im Hauptspeicher ausgeführt. Damit hat der Assembler die Eingabe an den RISC-V CPU erzeugt. Der Maschinen-Code wird auf die Kommandozeile ausgegeben und kann von dort kopiert werden.

5. RISC-V External Debug Support (Debug Spezifikation)

Möchte man direkt über den Zustand des CPUs oder der an den Systembus angeschlossen Module wie Speicher oder UARTs informiert werden, sind einfache Tests über LEDs oder UART nicht detailliert genug.

Einen direkten Zugriff auf die Register und den Speicher erhält man über einen Debugger. Ein Debugger kann nur funktionieren, wenn die Zielhardware Funktionen bereitstellt, die der Debugger nutzen kann. Der Debugger muss darüber hinaus überhaupt erst über eine Infrastruktur mit dem CPU verbunden werden.

Die Spezifikation für den Debug Support heißt "RISC-V External Debug Support" [101] und wird im Folgenden Debug-Spezifikation genannt. Der RISC-V External Debug Support ist keine RISC-V Extension, sondern eine Richtlinie für eine Implementierung.

Um die Debug-Spezifikation zu verstehen, ist es wichtig, unterschiedliche Ebenen voneinander zu trennen, da sonst schnell Verwirrung entsteht. Die Ebenen, die zu trennen sind, sind die konkrete Implementierung der Infrastruktur (hier JTAG und Wishbone) und die Module, die spezifisch für RISC-V definiert werden (DTM, DMI, DM und Harts).

TODO: Hier Bild von Seite 6 der Debug_Spec darstellen.

Die Debugger-Software spricht mit einer Debug-Probe, welche die Signale auf das FPGA-Board überträgt, auf dem das RISC-V System läuft. Die RISC-V Debug

Spezifikation gibt nicht vor, welche Technologie zur Übertragung der Signale verwendet werden muss.

Innerhalb der RISC-V Platform wird die Verbindung zunächst am Debug Transport Module (DTM) terminiert. Es kann in einer RISC-V Platform mehrere DTM Implementierungen geben. Beispielsweise könnte ein DTM über USB sprechen, eine andere Implementierung eines DTM über JTAG.

Der DTM verwendet das Debug-Module-Interface (DMI), um mit einem oder mehreren Debug-Modulen (DM) zu sprechen. Das Debug-Module-Interface definiert Register und Befehle, die eine Kommunikation mit dem DM auslösen. Der Debugger muss diese Befehle schicken, um bestimmte Aktionen innerhalb der RISC-V CPU auszulösen.

Die RISC-V CPU besteht aus mindestens einem Hart. Fortschrittliche RISC-V CPUs können viele Harts enthalten um echtes Multithreading zu ermöglichen. Ein Debug-Module (DM) ist für einen oder mehrere Harts gleichzeitig zuständig. Eine RISC-V Platform kann eine oder mehrere DM enthalten. Über Befehle kann ein DTM die DMs anweisen, ihre Harts zu pausieren, zu reseten und weiterlaufen zu lassen. Es ist auch möglich die Register der Harts zu lesen, zu schreiben. Über den Systembus ist es möglich, mit dem Speicher (RAM für Instruktionen und Daten) zu interagieren und den Speicherinhalt zu lesen und zu schreiben.

Die RISC-V Platform, die im Zuge dieser Arbeit erstellt wird, besitzt einen einzigen Hart. Aus dem gleichen Grund wird nicht jeder Teil der Debug-Spezifikation implementiert. Das Zählen und das Auswählen der DMs wird nicht implementiert. Es wird implizit von einem einzigen Hart ausgegangen, der automatisch immer ausgewählt und damit immer das Ziel aller Befehle ist.

Die konkrete Implementierung zur Übertragung der Daten zwischen der Debug-Probe und dem DTM innerhalb der RISC-V Platform wird mit JTAG implementiert.

JTAG ist ein Protokoll, das zum Testen von Hardware eingesetzt wird. Es erlaubt neben dem Ausführen von Hardwaretest Abläufen auch das Schreiben in Register und in den Speicher und wird daher auch häufig eingesetzt, um

Mikrocontroller-Firmware zu flashen. Innerhalb der JTAG-Beschreibung gibt es den TAP. Der TAP befindet sich im Zielsystem und terminiert dort die JTAG Verbindung. Da diese Aufgabe beinahe deckungsgleich mit der des DTM aus der RISC-V Debug Spezifikation ist, sind in dieser Studienarbeit DTM und JTAG-TAP im selben Modul implementiert.

JTAG ist in der Lage, Register zu beschreiben. Die Interaktion zwischen dem JTAG Protokoll und dem RISC-V DTM besteht also darin, dass der JTAG-TAP in die Register des DTM schreibt. Die RISC-V Debug Spezifikation ist so ausgeführt, dass das Beschreiben eines Registers Aktionen direkt startet. Damit ist JTAG ein perfektes Protokoll, um die RISC-V Debug Spezifikation auszuführen. JTAG kann also dazu genutzt werden DMI-Befehle auszuführen, in dem das DMI Register des DTM über JTAG beschrieben wird.

Das Debug-Module-Interface (DMI) ist das Protokoll zwischen einem DTM und mehreren DMs. Die Frage ist, wie diese Datenübertragung in Hardware implementiert wird. Die RISC-V Debug Spezifikation überlässt dieses Detail der Implementierung und macht absolut keine Vorgaben. In der Debug-Spezifikation wird nirgends definiert, wie eine DMI-Transaktion gestartet oder gestoppt wird. Der verwendete Wortlaut ist absichtlich abstrakt und es wird von "Lesen" und "Schreiben" gesprochen. Im Zuge dieser Studienarbeit wird die Interaktion über den Wishbone Bus implementiert. Der Wishbone Bus ist gut dokumentiert (TODO: B4 Spec hier verlinken) und ohne Lizenzgebühren einsetzbar. Der Wishbone Bus ist von der Komplexität her minimal im Vergleich zu anderen Bussystemen (AMBA-Bus, AXI-Bus, ...).

Für einen einfachen Debugger sollen folgende Funktionen implementiert werden:

- Hart stoppen, reseten und starten und im Single Step Modus ausführen sowie die normale Ausführung wieder aufnehmen.
- Register lesen schreiben
- Speicher lesen schreiben

Mit diesen Funktionen kann ein Programm auf den RISC-V CPU geladen werden und das Programm kann im Single-Step Betrieb ausgeführt werden.

Um den Hart zu stoppen, hat der DM Kontroller über die Hardware Clock-Source, die den Hart mit einem Taktsignal versorgt. Wenn die Clock-Leitung unterbrochen wird, dann ist der Hart gestoppt. Zum Single-Step Betrieb generiert der DM ein eigenes Clock-Signal und führt dem Hart dieses Clock-Signal zu. Wenn der Hart fortgesetzt wird, dann wird die Hardware Clock-Source wieder zugeführt.

JTAG State Machine

JTAG ist ein sehr allgemeines System. Für JTAG besteht jedes Gerät aus Registern. JTAG definiert das Vorhandensein der Register IR (Instruction Register), BYPASS und weitere Datenregister, über die jedes Gerät selbst entscheiden kann. Jedes Gerät sollte jedoch zumindest ein IDCODE Datenregister haben. Jedem Register wird ein Code zugeordnet. Das IR-Register wird verwendet um die Codes aufzunehmen und damit das zum Code gehörige Register zum aktuellen Register zu machen. Zu jedem Zeitpunkt kann nur ein einziges Datenregister oder das BYPASS Register aktiviert sein. Das IR Register ist parallel dazu immer aktiv. Man kann immer Codes ins IR Register laden. JTAG schreibt vor, dass beim Start des Systems das IR Register so befüllt sein muss, dass das IDCODE Datenregister vorausgewählt wird. Das IDCODE Register enthält eine Beschreibung des Geräts.

JTAG ermöglicht es, mit lediglich vier Steuersignalen Register beliebiger Breite in einem Schreibzyklus zu beschreiben. Dazu findet eine Umsetzung der seriellen Signale zur Übertragung in eine parallele Verarbeitung der Signale zum tatsächlichen Schreiben der Register statt. Es wird zunächst ein Shift-Register seriell befüllt und dann das Schreiben in einem einzigen Schritt parallel ausgeführt. Die serielle Befüllung dauert mehrere Zyklen. Das Beschreiben geschieht in einem einzigen Zyklus.

Zunächst definiert JTAG eine Zustandsmaschine aus 16 Zuständen. Die Zustandsmaschine hat grundsätzlich zwei Arme. Ein Arm aktiviert das Shift-Register für IR und lädt Daten aus dem IR-Register in das IR-Shift-Register. Der andere Arm führt die entsprechenden Funktionen für das aktuell ausgewählte Datenregister oder BYPASS Register aus.

TODO: Bild der StateMachine

Jedes JTAG Gerät muss intern diese JTAG Zustandsmaschine ausführen. Die Zustandsmaschine ist im JTAG TAP enthalten. Ein Gerät kann einen oder mehrere JTAG TAPs (Test Access Port) aufweisen. Mehrere Geräte mit jeweils mehreren TAPs können Teil einer JTAG Verbindung sein. In JTAG wird nicht über eine Punkt-zu-Punkt-Architektur kommuniziert, es wird stattdessen eine Daisy-Chain aus JTAG TAPs gebildet. Dabei werden alle Shift-Register aller TAPs in einer langen Daisy-Chain aneinander gehängt! Alle Bits werden von Anfang bis Ende durch diese Daisy Chain geschiftet und durchlaufen dabei alle Shift-Register aller TAPs! Durch diesen minimalistischen Ansatz ist das System über nur vier Signale steuerbar.

TODO Bild der Daisy Chain aus Vivado

TODO: Bild Figure 2 von Medium einfügen.
(<https://medium.com/@aliaksandr.kavalchuk/diving-into-jtag-protocol-part-1-overview-fbdc428d3a16>)

Alle Zustandsmaschinen in allen JTAG TAPs der Daisy Chain befinden sich zu jedem Zeitpunkt im gleichen Zustand und führen synchron Transitionen aus. Um die synchronen Transaktionen zu ermöglichen sind die JTAG Steuersignale Test Clock (TCK) und Test Mode Select (TMS) parallel an alle JTAG TAPs angeschlossen. Die Steuersignale Test Data In (TDI) und Test Data Out (TDO) sind im Falle von TDO am Anfang der Daisy Chain und im Falle von TDI am Ende der Daisy Chain angeschlossen. Es ist damit möglich gleichzeitig ein Bit

einzushiften (TDI) und dabei gleichzeitig ein Bit zu lesen, das in TDO ausgeschiftet worden ist.

TODO eventuell einfügen Bild Seite 29 von
https://tu-dresden.de/ing/informatik/ti/vlsi/ressourcen/dateien/dateien_studium/dateien_lehstuhlseminar/vortraege_lehrstuhlseminar/hs_ws_0708/jtag-schnittstelle.pdf?lang=de

Mit JTAG ist es also möglich, Werte in das Register zu schreiben, zu warten und dann Werte aus Registern zu lesen. Oft werden Operationen in den Geräten angestoßen, sobald ein Register mit einem Wert beschrieben worden ist. Dabei wird ein bestimmter Wert in ein Shift-Register geschiftet. Wenn dann im Zustand UPDATE-DR der Wert aus dem Shift Register in das echte Datenregister zurückkopiert wird, findet die Ausführung der Operation statt. Die RISC-V Debug Spezifikation definiert Register und eine Reihe von Operationen, die durch das Schreiben bestimmter Register angestoßen werden. Damit ist JTAG eine passende Infrastruktur, um die RISC-V Debug Spezifikation zu implementieren.

Implementierung in Verilog

Die JTAG Zustandsmaschine ist in `jtag_tap.v` zu finden. Das Modul besitzt die Signale `jtag_clk`, `jtag_tms`, `jtag_tdi` und `jtag_tdo`. Ein Always-Block reagiert auf die negative Flanke der JTAG Clock und shifted ein Bit aus `jtag_tdo` aus. Ein anderer Always-Block reagiert auf die positive Flanke der JTAG Clock und bestimmt den nächsten Zustand der Zustandsmaschine. Dieser Block führt auch Aktionen aus, die für die jeweiligen Zustände ausgeführt werden sollen.

Wenn `CAPTURE_DR` betreten wird, wird abhängig vom IR Register das gewählte Datenregister in das entsprechende Shift-Register übertragen. Entsprechendes gilt für das IR-Shift- und IR-Datenregister, wenn `CAPTURE_IR` betreten wird.

Im `SHIFT_DR` Zustand wird zunächst das TDO Bit aus dem aktuellen Shift-Register für die folgende fallende Flanke gespeichert. Dann wird das Shift-Register um eine Position geshiftet und gleichzeitig das TDI-Bit in das

Register einfügt. Im UPDATE_DR Register wird das Shift-Register in das Datenregister zurück kopiert. Dabei wird dieser Zeitpunkt auch verwendet, um Aktionen anzustoßen. Beispielsweise wird im Falle des DTM.dmi Register eine DMI Transaktion über den Wishbone Bus mit dem DM aktiviert.

RISCV Abstract Commands

Dieser Abschnitt beschreibt die Implementierung der Befehle, mit denen der Speicher gelesen und geschrieben werden kann.

Die RISC-V Debug Spezifikation gibt an, dass es mehrere Möglichkeiten gibt, auf den Speicher zuzugreifen. Diese Möglichkeiten werden im Abschnitt B.2.7 Reading Memory aufgelistet. In dieser Studienarbeit wird der dritte Ansatz der abstrakten Kommandos ausgewählt.

Abstrakte Kommandos werden laut Debug Spezifikation mit den Privilegien des Harts ausgeführt. Dabei muss aber nicht zwangsweise der Hart selbst verwendet werden. Dabei wird kein ProgramBuffer verwendet. Da der simple Hart keine Privilegien implementiert, ist das System konform zur Debug Spezifikation, wenn der DM den Speicherzugriff einfach direkt selbst implementiert und ausführt. Es wird einfach direkt auf den Speicher zugegriffen. Dabei wird kein Bus zwischen DM und Speicher verwendet, da die simple RISC-V CPU keinen internen Bus besitzt.

Zusammenfassendes Beispiel

Das Schreiben in den Speicher über JTAG innerhalb der RISC-V Debug Spezifikation umfasst folgende Abfolge. Zunächst muss ein DMI Command abgesetzt werden, damit der DTM in den DM schreibt. Dazu wird der DMI Command in das DTM dmi Register geschrieben. Der DMI Command ist ein Schreibbefehl in das Control Register des DM. Der Wert, der in das DM Control

Register geschrieben wird, ist ein Abstract Command. Diese Abstract Command schreibt einen Wert in den Speicher. Der Wert, der in den Speicher geschrieben wird, wird im Data0 register erwartet, die Adresse an die geschrieben wird, wird im Data1 Register erwartet. Streng genommen muss also vor dem Ausführen des Abstract Commands also zunächst Data0 und Data1 geschrieben werden. Wenn der Abstract Command ausgeführt worden ist, gibt es keinen Rückgabewert!

Das Lesen aus dem Speicher erfolgt ebenfalls über einen Abstract Command. Dieser startet ebenfalls dadurch, dass ein DMI Command, welcher einen Wert in das DM Command Register schreibt, in das DTM dmi Register geschrieben wird. Damit wird der DMI Command in den DM schreiben. Die Adresse, von der gelesen werden soll, muss sich im Register Data1 befinden. Der aus dem Speicher gelesene Wert wird im Data0 Register zurückgegeben. Von dort kann er mit einem anderen Abstract Command abgefragt und an den Debugger über JTAG zurückgegeben werden.

Debug Probe und Python Skripte

Es gibt JTAG DebugProbes wie z.B. den Segger J-Link, den Jtagulator oder den simpleren GoodFet. Da die Implementierung in dieser Studienarbeit keine vollständige JTAG Implementierung liefert, wird eine einfache Umsetzung über einen Arduino DUE erstellt. Der Arduino Due kommuniziert über UART mit einem PC auf dem Python Skripte ausgeführt werden und setzt auf der anderen Seite Kommandos in die vier Signale JTAG_TMS, JTAG_CLK, JTAG_DTI und JTAG.DTO um.

Anbei ein kleiner Auszug, der zeigt, wie der Arduino die Zustandsmaschine antreibt, indem er das TMS eintaktet: Siehe Anlage A3.

Das Protokoll zwischen den Python-Skripten und dem Arduino definiert ein einfaches Datenformat und lediglich drei Kommandos. Das Ping Kommando wird mit einem Pong beantwortet und dient lediglich zum Test, ob der Arduino antwortet und folglich ob man das Protokollformat korrekt implementiert hat. Es

werden keine JTAG Daten aufgrund des Ping Pong übertragen. Das SEND_TMS Kommando kann verwendet werden, um die State Machines anzutreiben, ohne Daten über TDI zu schicken. Das letzte Kommando ist SHIFT_DATA. Hier wird eine Menge an Bits über TDI in alle State Machines geschiftet. Dabei kann angegeben werden, welches Signal an TMS angelegt werden soll. Dies ist notwendig um ein Register bis auf das letzte Bit zu füllen und dann zusammen mit dem letzten Bit auch ein TMS Signal zu schicken, mit dem die StateMachine dann angewiesen wird den Shift-Zustand zu verlassen, so dass das Register "versiegelt" wird und keine falschen Daten mehr in das Register gelangen! Register werden nämlich nur während des Shift Zustandes mit Daten aus TDI versorgt.

Innerhalb der Python-Skripte wird mit der Zustandsmaschine umgegangen, um bestimmte Register auszuwählen und Daten in diese Register zu shiften. Damit lassen sich die Abfolgen für abstrakte Kommandos ausführen.

6. Abschluss und Ausblick

Es wurde Hardware und Software verglichen, bewertet und ausgewählt um einen einfachen RISC-V Kern mit einer Pipeline zu implementieren. Das Design wurde beschrieben. Das RISC-V Systems wurde um eine Programmierschnittstelle per JTAG unter Verwendung der RISC-V Debug-Spezifikation erweitert. Damit ist das System um eigene Software erweiterbar. Die Software kann durch einen eigens erstellten Assembler für 32-Bit RISC-V übersetzt und durch einen eigens entwickelten Emulator vorgeprüft werden.

Damit ist das Ziel, einen Studenten eine Weg zur Implementierung eines RICS-V Systems aufzuzeigen, erreicht.

Es gibt Teile der Implementierung, die vervollständigt werden können. Es sind nicht alle RV32I Befehle im Kern ausführbar. Mehrere interessante Extensions können hinzugefügt werden.

Die Übertragung der JTAG Befehle über den Arduino ist funktionsfähig allerdings momentan sehr langsam getaktet, da die Always-Blöcke, die im FPGA auf die JTAG Clock reagieren oft Bits verpassen oder duplizieren, wenn der Takt angehoben wird. Das ist unverständlich, da die Verbindung langsamer aus ein UART mit Baudrate 115200 ist und damit eigentlich noch einfacher für einen FPGA handhabbar sein sollte. An dieser Stelle kann das Verfahren mit echter JTAG Hardware nachgeprüft und die Übertragungsrate angehoben werden. Momentan wurde lediglich die Machbarkeit gezeigt.

7. Danksagung

Ich bedanke mich sehr herzlich bei Dr. Christian Jakob für die Betreuung der Arbeit und seine Hilfestellung und Ratschläge beim Erstellen dieser Arbeit. FPGAs haben einen sehr hohen Stellenwert für mich und das Erstellen dieser Arbeit bringt mich meinem Ziel sehr viel näher. Nebenbei gesagt, hat die Arbeit mit dem RISC-V System und dem FPGA auch sehr viel Spaß gemacht. In einer Umgebung, die aus kleinsten Bausteinen besteht, kann mit Kreativität eine Vielzahl an Lösungen erstellt werden, auch wenn der Weg zur Lösung oft ein steiniger ist.

Literatur- und Quellenverzeichnis

Anlagenverzeichnis

Anlage A1:

```
--main:  
loop_start:  
    addi x5, x0, 0x0  
    addi x6, x0, 0x0  
    lui x7, 0  
    addi x7, x7, 2  
  
busy_loop_start:  
    beq x5, x7, 0xC          # if (x5 == x7) jump to loop_end (pc relative jump  
of +12 bytes)  
    addi x5, x5, 1  
    jal x0, busy_loop_start  # jal loop head (pc relative jump back -8 bytes)  
  
busy_loop_end:  
    lw x6, 52(x0)  
    xori x6, x6, 1  
    sw x6, 52(x0)  
  
    jal x0, loop_start
```

Anlage A2:

```
.section .text  
.global strlen  
strlen:  
    # a0 = const char *str  
    li      t0, 0           # i = 0  
1: # Start of for loop  
    add    t1, t0, a0        # Add the byte offset for str[i]  
    lb     t1, 0(t1)        # Dereference str[i]  
    beqz  t1, 1f            # if str[i] == 0, break for loop  
    addi   t0, t0, 1         # Add 1 to our iterator  
    j      1b                # Jump back to condition (1 backwards)
```

```
1: # End of for loop
    mv      a0, t0          # Move t0 into a0 to return
    ret                 # Return back via the return address register
```

Anlage A3:

```
for (size_t i = 0; i < len; i++) {
    // determine next bit to send out
    uint8_t bit = *in_data & 0x01;
    *in_data >>= 1;
    // set CLK low
    delay(delay_in_ms);
    digitalWrite(jtag_clk, LOW);
    jtag_clk_state = LOW;
    // place data on TMS
    digitalWrite(jtag_tms, tms_data);
    digitalWrite(jtag_tdo, bit);
    // set CLK high
    delay(delay_in_ms);
    digitalWrite(jtag_clk, HIGH);
    jtag_clk_state = HIGH;
    // cause negedge and read data
    delay(delay_in_ms);
    digitalWrite(jtag_clk, LOW);
    jtag_clk_state = LOW;
    // combine read bit into returned value
    delay(delay_in_ms);
    int val = digitalRead(jtag_tdi);
    *read_data >>= 1;
    *read_data |= (val << 7) << 24;
}
```

Anlage A4:

Anlage A5:

Anlage A6:

Anlage A7:

Anlage A8: