Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Bachelorarbeit

Interleaved Multithreading für RISC-V Softcores

Leipzig, September 2020

vorgelegt von
Till Mahlburg
Studiengang Informatik

Betreuender Hochschullehrer:

Prof. Dr. Martin Bogdan
Fakultät für Mathematik und Informatik
Technische Informatik

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Prozessoren kommen heute in den unterschiedlichsten Größen, Formen und Leistungsklassen. Sie stecken in Geräten, in denen man sie erwartet, etwa einem PC, einem Notebook, der Spielekonsole oder auch dem Smartphone. Längst stecken sie aber auch in Geräten, in denen man es vielleicht nicht erwarten würde: Im Kühlschrank oder im Toaster, sogar in manchen Lichtschaltern. Gleichzeitig steigen in jedem dieser Geräte die Leistungsanforderungen, da immer mehr Funktionen unterstützt werden sollen. Um diese Anforderungen auch angesichts der Verlangsamung der Erhöhung der Transistordichte erfüllen zu können, wird heute immer mehr auf Parallelismus gesetzt. Genau auf die parallele Ausführung zur Steigerung der Leistung zielt diese Arbeit ab.

Die Idee ist, mit möglichst geringem zusätzlichen Hardwareaufwand einem bestehenden Prozessor die Fähigkeit zu geben, mehrere unabhängige Aufgaben effizient zu erfüllen. Als Methode hierfür soll *Interleaved Multithreading* dienen. Dieses verspricht eine bessere Auslastung der sowieso vorhandenen Hardware und die Ausführung mehrerer *Threads*, die dann jeweils an unterschiedlichen Aufgaben arbeiten können [21, S. 223-226].

Ein anderer Trend ist die Hinwendung auch großer Unternehmen zu Open-Source-Software. Viele Firmen bieten heute Software zu den Bedingungen offener Lizenzen an [30, S. 7 - 32]. Open-Source-Hardware dagegen nimmt bisher nur langsam Fahrt auf. Ein wichtiger Schritt in diese Richtung war die Entwicklung des RISC-V-Befehlssatzes. Dieser steht unter einer offenen Lizenz und kann von jeder bzw. jedem implementiert werden [28, S. 2]. Dadurch liegt der Entwicklung effizienter, offener Prozessoren nun ein Stein weniger im Weg, denn die Befehlssatzarchitektur ist ein grundlegendes Element eines Prozessors und RISC-V zunehmend populär, wodurch die Softwareunterstützung stetig zunimmt.

Dieser Entwicklung hilft auch die Verfügbarkeit erschwinglicher FPGA-Entwicklungsboards. Durch sie können solche offenen Prozessoren auf echter Hardware getestet und überprüft werden. Das ermöglicht es Privatpersonen oder Gemeinschaften, benutzbare Prozessoren zu entwickeln, ohne einen teuren Fertigungsprozess bezahlen zu müssen.

Die Frage, die diese Arbeit zu beantworten versucht, ist also, wie sich die Implementierung von *Interleaved Multithreading* für ein solches offenes RISC-V-Prozessordesign auf Leistung und Ressourcenverbrauch auswirkt. Das Verfahren verspricht dabei, einen deutlichen Geschwindigkeitszuwachs bei geringem zusätzlichen Hardwareaufwand zu ermöglichen.

Die Arbeit stellt zur Beantwortung dieser Frage zuerst die theoretischen *Grundlagen* vor. Darauf folgt die Untersuchung der Entwicklungen und Forschungen in den relevanten Gebieten im Kapitel *Stand der Technik*. Hierauf aufbauend wird im nächsten Kapitel der *Entwurf* des Designs und danach dessen *Implementierung* vorgestellt. Diese wird dann getestet. Die Ergebnisse daraus werden im Kapitel *Auswertung* analysiert. Den Schluss bildet das Kapitel *Zusammenfassung und Ausblick*. In dieser Arbeit werden mehrere Abbildungen gezeigt. Sind diese nicht anderweitig gekennzeichnet, handelt es sich um vom Verfasser dieser Arbeit selbst erstellte Darstellungen.

2 Grundlagen

Zuerst werden die grundlegenden Begriffe und Konzepte vorgestellt, die für diese Arbeit relevant sind. Dazu werden die entsprechenden Technologien kurz eingeführt und deren Funktionsweisen erklärt.

2.1 Pipelining

In modernen Prozessoren ist einer der zentralen Mechanismen zur Leistungssteigerung das Pipelining. Hierbei werden die typischen Schritte, die ein Prozessor zur Ausführung jeder Instruktion durchläuft, überlappend ausgeführt. Angenommen, der Prozessor, der Pipelining implementiert, durchläuft die fünf typischen Phasen *Instruction Fetch*, *Instruction Decode / Register Fetch*, *Execution*, *Memory Access* und *Write-back*, dann würden die Instruktionen im besten Fall wie in Tabelle 2.1 ausgeführt werden [21, S C-5 - C6].

In diesem Beispiel ist mit I_n die Instruktion gemeint, die an der fiktiven Speicheradresse n gespeichert ist. Außerdem wird vorausgesetzt, dass jede dieser Ausführungsphasen genau einen Takt lang dauert. Es wird deutlich, dass beim Pipelining jede Instruktion für sich genau so viel Zeit zur Ausführung benötigt, wie bei einem Prozessor, der kein Pipelining unterstützt. Die Latenz einer Instruktion bleibt also gleich. Da aber ab Takt 5 in jedem Takt eine Instruktion fertig abgearbeitet wird, ist der Durchsatz im Idealfall um die Anzahl der isolierten Ausführungsphasen, auch als *Pipelinestufen* bezeichnet, schneller. Es gilt nach [21, S. C-2 - C-3]:

$$\text{idealer Speedup} = \frac{\frac{\text{Zeit}}{\text{Instruktion}} \text{ auf einer Maschine ohne Pipelining}}{\text{Anzahl Pipelinestufen}} \tag{2.1}$$

Wie bereits erwähnt, handelt es sich hier um den besten Fall. Es werden beim Pipelining drei Arten von Problemen unterschieden, die als *Hazards* bezeichnet werden. Durch diese kann die ideale Ausführung erschwert oder verhindert werden [26, S. 210 - 211]:

	Ausführungsphase				
Takt	IF	ID / RF	EX	MA	WB
1	I_0	-	-	-	-
2	I_1	I_0	-	-	-
3	I_2	I_1	I_0	-	-
4	I_3	I_2	I_1	I_0	-
5	I_4	I_3	I_2	I_1	I_0
6	I_5	I_4	I_3	I_2	I_1
7	I_6	I_5	I_4	I_3	I_2
...					

Tabelle 2.1: Beispielhafter Ablauf einer Pipeline

1. **Structural Hazards**: Ressourcenkonflikte, wenn aus zwei Ausführungsphasen gleichzeitig auf dieselbe Hardware zugegriffen werden muss.
2. **Data Hazards**: Eine Instruktion hängt von dem Ergebnis einer vorherigen Instruktion ab, die noch nicht zu Ende abgearbeitet wurde. Im Beispiel könnte etwa I_2 im Takt 4 das Ergebnis von I_0 in einem Register erwarten, obwohl I_0 ihr Ergebnis noch nicht in dieses Register geschrieben hat.
3. **Control Hazards**: Bestimmte Instruktionen ändern den Ablauf des Programms, sodass die Adresse der nachfolgende Instruktion nicht sofort eindeutig ist. Das betrifft zum Beispiel Verzweigungen und Sprünge.

Für all diese Probleme gibt es verschiedene, unterschiedlich komplexe Lösungen, auf die hier aber nicht weiter eingegangen werden soll, da sie für diese Arbeit nur wenig Relevanz besitzen [22, S. 392 - 393]. Wichtig ist hier in erster Linie, dass diese Probleme den Ablauf einer Pipeline stören können.

Pipelining ist eine Form der Parallelität auf Instruktionsebene und als solche für die Programmiererin oder den Programmierer im Allgemeinen unsichtbar [21, S. C-2 - C-3].

2.2 Multithreading

Eine weitere Methode zur Leistungssteigerung in Prozessoren ist das Multithreading. Hierbei werden mehrere Threads überlappend oder auch gleichzeitig von demselben Prozessor ausgeführt. Es handelt sich dementsprechend um Parallelität auf Threadebene. Der Programmierer muss also das Multithreading

explizit ausnutzen, um dessen volles Potenzial ausschöpfen zu können. Unterschieden werden drei Formen des Multithreadings, wobei sich die konkreten Bezeichnungen unterscheiden, aber dasselbe meinen. In der Arbeit werden die Bezeichnungen aus [35, S. 33 - 35] übernommen. Die entsprechenden Bezeichnungen aus [21, S. 224 - 225] sind hier zur Information zusätzlich mit angegeben, werden aber im Verlauf der Arbeit nicht mehr verwendet.

1. Interleaved oder Fine-grained Multithreading (IMT)
2. Blocked oder Coarse-grained Multithreading (BMT)
3. Simultaneous Multithreading (SMT)

Die verschiedenen Varianten des Multithreadings können weiterhin noch danach kategorisiert werden, wie viele Threads pro Takt die Arbeit an einer neuen Instruktion beginnen können. IMT und BMT erlauben maximal einem Thread eine neue Instruktion, SMT kennt diese Beschränkung nicht. Die Unterscheidung zwischen BMT und IMT liegt darin, dass bei IMT üblicherweise die jeweils nächste Instruktion aus einem anderen Thread stammt als die vorherige, bei BMT hingegen solange derselbe Thread ausgeführt wird, bis ein bestimmtes Ereignis eintritt, das einen Threadwechsel einleitet. Das kann beispielweise ein langandauernder Speicherzugriff sein [35, S. 33]. SMT funktioniert ähnlich wie IMT, wird aber auf superskalaren Prozessoren ausgeführt, also Prozessoren, die mehrere Funktionseinheiten und/oder mehrere Pipelines haben. Diese werden mithilfe mehrerer Threads ausgelastet, wodurch eine höhere Auslastung der Prozessorhardware möglich ist. Dazu braucht es auch ein zusätzliches Scheduling, das bestimmt, welche Threads als nächste ausgeführt werden sollen [21, S. 224 - 225].

2.3 Interleaved Multithreading

In dieser Arbeit wird es speziell um Interleaved Multithreading (IMT) gehen. Im Vergleich zu BMT verhält sich IMT vorhersagbarer und teilt die vorhandenen Ressourcen gleichmäßig zwischen den Threads auf, da jeder Thread dieselbe Ausführungszeit erhält. Aus diesen Gründen soll hier noch einmal konkreter auf die Arbeitsweise desselben eingegangen werden. Ähnlich der Erklärung zum Pipelining, folgt zum besseren Verständnis eine tabellarische Übersicht. Dabei gelten dieselben Annahmen wie zur Tabelle 2.1, allerdings werden Instruktionen fünf verschiedener Threads ausgeführt, wobei $T_n I_m$ für die m-te Instruktion steht, die von Thread n ausgeführt werden soll.

	Ausführungsphase				
Takt	IF	ID / RF	EX	MA	WB
1	T_0I_0	-	-	-	-
2	T_1I_0	T_0I_0	-	-	-
3	T_2I_0	T_1I_0	T_0I_0	-	-
4	T_3I_0	T_2I_0	T_1I_0	T_0I_0	-
5	T_4I_0	T_3I_0	T_2I_0	T_1I_0	T_0I_0
6	T_0I_1	T_4I_0	T_3I_0	T_2I_0	T_1I_0
7	T_1I_1	T_0I_1	T_4I_0	T_3I_0	T_2I_0
8	T_2I_1	T_1I_1	T_0I_1	T_4I_0	T_3I_0
9	T_3I_1	T_2I_1	T_1I_1	T_0I_1	T_4I_0
10	T_4I_1	T_3I_1	T_2I_1	T_1I_1	T_0I_1
11	T_0I_2	T_4I_1	T_3I_1	T_2I_1	T_1I_1
...					

Tabelle 2.2: Beispielhafter Ablauf von Interleaved Multithreading

In Tabelle 2.2 wird deutlich, dass innerhalb eines Threads jede Instruktion vollständig abgearbeitet wird, bevor die Arbeit an der nächsten begonnen wird. Deutlich wird das etwa daran, dass T_0I_0 in Takt 5 fertig abgearbeitet wird und erst danach, in Takt 6, die nächste Instruktion, T_0I_1, bearbeitet wird. Die Ausführung eines einzelnen Threads ist daher gegenüber einem Prozessor, der weder Pipelining noch IMT unterstützt, nicht beschleunigt, die Latenz eines Threads bleibt also gleich. Dafür steigt bei mehreren Threads die Auslastung bzw. der Durchsatz, da auch hier ab Takt 5 in jedem Takt eine Instruktion, wenn auch aus unterschiedlichen Threads, abgeschlossen wird.

Gegenüber dem Pipelining wird beim IMT also nicht der einzelne Thread beschleunigt, sondern nur der Durchsatz bei der Ausführung mehrerer Threads erhöht. Andererseits geht man auf diese Weise einigen der Hazards aus dem Weg, die beim Pipelining auftreten. Während *Structural Hazards* immer noch auftreten können, sind *Control Hazards* ausgeschlossen, da die Berechnung der Adresse der nächsten Instruktion eines Threads garantiert abgeschlossen ist, bevor diese bearbeitet wird. Auch *Data Hazards* sind in der bekannten Form ausgeschlossen, da auch hier alle Ergebnisse der vorherigen Instruktion bereits feststehen, wenn die Arbeit an einer neuen begonnen wird. Data Hazards können nur noch zwischen Threads auftreten, nicht mehr innerhalb dieser. Die Verhinderung dieser Art Data Hazards ist aber Aufgabe des Programmierers bzw. der Programmiererin und nicht der Hardware.

Durch dieses Design sollte also dank der minimierten Hazards mithilfe von relativ wenig Hardwareaufwand der Durchsatz eines Prozessors ohne Pipeline gesteigert werden können. Das gilt besonders im Vergleich mit einer ähnlich hazardfreien Pipelineimplementierung. Speziell die effiziente Ausführung von Software, die auf viele schwer vorherzusagende Verzweigungen angewiesen ist, erfordert bei Prozessoren mit Pipeline einen hohen zusätzlichen Hardwareaufwand, während IMT-Prozessoren mit solcher im Allgemeinen keine Probleme haben. Dafür wird durch IMT wiederum die Programmierbarkeit erschwert, da nun zusätzlicher Aufwand zur Verwaltung und Synchronisierung der Threads notwendig wird. Außerdem muss die auszuführende Aufgabe hinreichend parallelisierbar oder mehrere, möglichst unabhängige Aufgaben gleichzeitig ausführbar sein. Dadurch wird der mögliche Leistungsgewinn auch durch die auszuführende Software begrenzt.

2.4 RISC-V

Das Feld der Prozessoren wird aktuell vor allem von zwei Befehlssatzarchitekturen dominiert [21, S. K-2]. Auf der einen Seite gibt es die x86- bzw. x86_64-Architektur. Diese sind besonders in Desktop-PCs verbreitet, aber auch in Laptops und in Servern. Auf der anderen Seite gibt es die verschiedenen Versionen der ARM-Architektur. Diese ist vor allem in eingebetteten Systemen dominant, von kleinsten Mikrocontrollern bis zu Smartphones, aber vereinzelt auch in Laptops oder Servern zu finden. Neben diesen beiden Architekturen gibt natürlich noch weitere, die dann meist eine eher nischenhafte Verbreitung finden, bspw. wird IBMs z/Architecture ausschließlich in Mainframes verwendet.

RISC-V ist der Versuch, eine neue offene Befehlssatzarchitektur zu etablieren, die ohne Altlasten auskommt und aus den Fehlern und Erfolgen der Vergangenheit gelernt hat [28, S. 12]. Die Architektur wurde an der University of California enwickelt und im Jahr 2010 vorgestellt. Wie am Namen zu erkennen ist, handelt es sich um eine *RISC*-Architektur. Das steht für *Reduced Instruction Set Computer*. Dieser Ansatz zum Prozessordesign ermöglicht eine hohe Leistung durch einen Fokus auf Cache und Parallelismus auf Instruktionsebene [21, S. 2]. Dabei wird letzteres meist durch Pipelining ermöglicht und ist im Allgemeinen für den Entwickler unsichtbar. Um die Implementierung einer Pipeline zu unterstützen, haben verschiedene RISC-Architekturen oft einige Gemeinsamkeiten [25, S. 13 - 14]:

- feste Instruktionslängen
- Speicherzugriff nur über load- bzw. store-Instruktionen

- gleichbleibender Aufbau der Operanden in allen Instruktionen: ein bis zwei Quelloperanden (entweder aus Registern oder Konstanten) und ein Zielregister
- feste Ausführungszeiten für jede Instruktion, die nicht auf den Speicher zugreift
- simple Kontrollstrukturen

An den jeweiligen Beispielen in den entsprechenden Abschnitten ist zu erkennen, dass sich Pipelining und IMT konzeptionell ähnlich sind. Dadurch sind viele der Bemühungen, die Implementierung von Pipelines zu vereinfachen, ebenso geeignet, IMT einfacher implementierbar zu machen.

Die Anwendung dieser Konventionen und die dadurch möglichen Pipelines führten nicht nur zu höherer Leistung, sondern auch zu günstigeren Herstellungskosten im Vergleich zu früheren CISC-Architekturen. Aufgrund der Vorteile sind auch Architekturen wie x86_64, die früher als CISC implementiert waren, heute intern RISC [21, S. 2].

Basierend auf diesen Erfahrungen mit früheren RISC-Designs wurde RISC-V entwickelt. Dabei war es ein Ziel, eine *universelle* Befehlssatzarchitektur zu schaffen. Daher soll sie folgenden Anforderungen genügen [28, S. 3]:

- implementierbar mit Prozessoren aller Leistungsklassen und Größen
- Kompatibilität zu populären Programmiersprachen und Softwarestacks
- Unterstützung aller Implementierungstechnologien: FPGAs, ASICs, herkömmliche Mikroprozessoren oder auch zukünftige Technologien
- effiziente Umsetzbarkeit von Mikroarchitekturen aller Art
- Unterstützung umfassender Spezialisierung als Voraussetzung zur Implementierung von Rechenbeschleunigern
- Stabilität grundlegender Teile der Architektur

Diese Punkte führen zu einer starken Trennung der Architektur von der Implementierung bzw. der Befehlssatzarchitektur von der Mikroarchitektur. Darüber hinaus ist RISC-V eine modulare Architektur, wobei das »Basismodul«, RV32I bzw. RV64I, nur ganzzahlige Operationen in Hardware unterstützt. Weitere Module sind beispielsweise RV32M, das Multiplikationsinstruktionen hinzufügt, oder RV32C, womit kompaktere Instruktionen (nur 16 statt 32 Bit) unterstützt werden. Diese Module können von der Prozessordesignerin oder vom Prozessordesigner beliebig je nach Anforderung zusammengestellt werden und sogar durch eigene Module erweitert werden. Ein 32-Bit-Prozessor, der beispiels-

Name	Beschreibung	Klassifikation	Status
RV32I	32 Bit, ganzzahlige Operationen	Basismodul	ratifiziert
RV64I	64 Bit, ganzzahlige Operationen	Basismodul	ratifiziert
RV32E	32 Bit, ganzzahlige Operationen, nur 16 Register	Basismodul	Entwurf
RV128I	128 Bit, ganzzahlige Op.	Basismodul	Entwurf
M	ganzzahlige Multiplikation u. Division	Erweiterung	ratifiziert
A	atomare Operationen	Erweiterung	ratifiziert
F	Gleitkomma-Op., einfache Genauigkeit	Erweiterung	ratifitiert
D	Gleitkomma-Op., doppelte Genauigkeit	Erweiterung	ratifiziert
Q	Gleitkomma-Op. vierfache Genauigkeit	Erweiterung	ratifiziert
C	auf 16 Bit komprimierte Instr.	Erweiterung	ratifiziert
Counters	Zähler	Erweiterung	Entwurf
L	dezimale Gleitkomma-Operationen	Erweiterung	Entwurf
B	Operationen zur Manipulation von Bits	Erweiterung	Entwurf
J	effiziente Ausführung dynamischer Sprachen	Erweiterung	Entwurf
T	Operationen für transaktionalen Speicher	Erweiterung	Entwurf
P	Packed-SIMD-Operationen	Erweiterung	Entwurf
V	Vektor-Operationen	Erweiterung	Entwurf
Zicsr	Zugriff auf Control-and-Status-Register	Erweiterung	ratifiziert
Zifencei	fence-Instruktionen	Erweiterung	ratifiziert
Zam	falsch ausgerichtete atomare Operationen	Erweiterung	Entwurf
Ztso	total-store-Speicherordnung	Erweiterung	eingefroren

Tabelle 2.3: Module des RISC-V-Befehlssatzes

weise die I, M und F Erweiterungen unterstützt, hätte den Befehlssatz RV32IMF [28, S. 4 - 5]. In Tabelle 2.3 sind alle offiziell ratifizierten, sich in Entwicklung befindenden oder geplanten Module dargestellt [32, S. i; S. 152].

Eine weitere Besonderheit von RISC-V ist die Offenheit der Architektur. Jede und jeder darf ohne Lizenzgebühren zu bezahlen Prozessoren mit diesem Befehlssatz entwickeln. Die dabei entstehenden Designs können ebenso offen sein wie die Architektur, dürfen aber auch proprietär sein. Da die RISC-V-Foundation die Architektur entwickelt, ist die weitere Zukunft von RISC-V nicht vom Schicksal eines oder weniger Unternehmen abhängig [28, S. 2]. Dank dieser Struktur hat sich eine rege Community aus Einzelpersonen und Unternehmen entwickelt, die sich mit dem Design offener RISC-V-Prozessoren beschäftigt. Auf einen Teil der Ergebnisse dieser Community wird im Kapitel *Stand der Technik* noch weiter eingegangen.

2.5 Softcores

Ein Softcore ist ein IP-Kern, der in einem FPGA realisiert wird. Ein Softcore-Prozessor ist ein Prozessor, der vollständig in einer Hardwarebeschreibungssprache, oder *HDL*, modelliert ist und auf programmierbarer Hardware, z.B. FPGAs, synthetisiert werden kann [29, S. 1]. Diese Form der Prozessorentwicklung bietet einige Vorteile: Es ermöglicht die Koexistenz verschiedener spezialisierter Logikschaltungen mit einem oder sogar mehreren unterschiedlichen Prozessoren auf einem einzigen Chip [31, S. 695]. Außerdem sind angepasste Prozessordesigns in niedrigeren Stückzahlen billiger auf FPGAs zu realisieren als in einem herkömmlichen Produktionsprozess [27, S. 1]. Darüber hinaus sind die Prozessoren auf diese Weise leichter zu verändern und flexibler auf echter Hardware zu testen, da z.B. nur ein neuer Bitstream auf den FPGA übertragen werden muss, um das neue Design zu testen, anstatt neue Hardware zu produzieren [24, S. 36].

Diese Eigenschaften, insbesondere die niedrigen Kosten bei geringen Stückzahlen, unterstützen die Entwicklung offener Prozessordesigns für FPGAs, da es für Einzelpersonen oder Gruppen erschwinglich ist, Designs auf echter Hardware zu testen. Dadurch können auch Freiwillige Hardware entwickeln, denen die finanziellen Mittel eines Unternehmens fehlen. Außerdem können Interessierte ein offenes Design wie bei Open-Source-Software an ihre eigenen Bedürfnisse anpassen und die veränderte Version ebenso kostengünstig mit FPGAs verwenden. Hier kann also echte Open-Source-Hardware entwickelt werden, die tatsächlich von ihren Anwenderinnen und Anwendern verändert und kontrolliert werden kann, da keine teure Produktion erforderlich ist. Außerdem integriert sich die Entwicklung mit HDLs ohne Probleme in übliche Arbeitsabläufe der Open-Source-Community, etwa die Nutzung von *git* als Quelltextverwaltung und Plattformen wie `https://github.com`, `https://gitlab.com` oder auch `https://opencores.org` als Kollaborationswerkzeuge.

3 Stand der Technik

In diesem Kapitel soll es um die aktuelle Situation von IMT und RISC-V im Allgemeinen, und IMT in Verbindung mit RISC-V im Speziellen gehen. Außerdem soll das Angebot an offenen Prozessordesigns, die für die Zwecke dieser Arbeit in Frage kommen, überblicksweise dargestellt werden.

3.1 Interleaved Multithreading und RISC-V

Die Entwicklung des Interleaved Multithreading geht bis in die 60er- und 70er-Jahre zurück. Bereits in der *CDC 6600*, einem Großrechner von 1964, wurde in den Peripher- und Kontrollprozessoren eine Form dieser Technik angewandt [34, S. 34-35]. Heute ist Hardware mit Unterstützung für Multithreading meist auf die bessere Auslastung der großzügig zu Verfügung stehenden Ressourcen ausgelegt. Hier wird dann meist SMT genutzt, um die mehrfach vorhandenen Ausführungseinheiten in Verbindung mit spekulativer oder auch Out-of-Order-Ausführung effizienter auszunutzen [23, S. 1].

Neben einigen Ausnahmen wie der Architektur der *Cray MTA*, die IMT mit einem VLIW Befehlssatz verbindet [35, S. 45], nimmt in neuerer Zeit die Relevanz des IMT für Hochleistungsarchitekturen ab. Gleichzeitig wird die Effektivität des Verfahrens für eingebettete Systeme und deren Prozessoren zunehmend deutlich. So implementiert der *JackKnife* Mikrocontroller den Befehlssatz AVR und »Dynamic Interleaving« [23, S. 3]. Damit ist eine Form des IMT gemeint, bei dem eine dynamische Threadauswahl erfolgt. Dabei wird einbezogen, welche Ressourcen verfügbar und welche Threads bereit sind, um den nächsten auszuführenden Thread zu wählen. Hervorgehoben wird u. A. der geringe Mehraufwand an Hardware, ein deutlich höherer Durchsatz gegenüber anderen AVR-Designs und die Möglichkeit der sehr schnellen Kontextwechsel [23]. Auch verschiedene Netzwerkprozessoren nutzen IMT, um Latenzen einzelner Threads durch die Ausführung anderer Threads zu überbrücken [35, S. 48].

Im Gegensatz zu IMT ist RISC-V eine sehr junge Entwicklung. Begonnen im Jahr 2010 aus den Erfahrungen früherer RISC-Architekturen [32, S.156], sind inzwischen eine 32- und eine 64-Bit-Variante und verschiedene Erweiterungen

von der RISC-V Foundation ratifiziert worden, während andere bisher erst als Entwurf vorliegen [32, S. i]. Trotz des vergleichsweise niedrigen Alters von zehn Jahren gibt es bereits eine beachtliche Anzahl an Prozessoren und SoCs [9]. Dabei geht das Feld von sehr kleinen Softcores [36] bis zu superskalaren Prozessoren, wie dem *Xuantie-910* mit 16 Kernen, einem Takt von bis zu 2,5 GHz, Out-of-Order-Ausführung und 64-Bit-Unterstützung [19].

Hardwareseitiges Multithreading ist bei den bisher entwickelten RISC-V-Prozessoren die Ausnahme. So ist die Prozessorfamilie *Klessydra* die bisher einzige offen dokumentierte und zugängliche Implementierung von IMT für einen Prozessor mit dem RISC-V-Befehlssatz [17]. Dieser zielt auf die Anwendung in »IoT end-nodes« [17, S. 1] ab und wird an der Sapienza-Universität in Rom entwickelt. Bei diesen »IoT end-nodes« handelt es sich um eingebettete Systeme in vernetzten Teilnehmern des *Internet of Things*, wie z.B. einem per WLAN steuerbaren Lichtschalter. Die *Klessydra*-Prozessoren sind Softcores, ein Design für die Fertigung als integrierter Schaltkreis war 2017 bereits in Arbeit [17, S. 2]. In der Prozessorfamilie gibt es verschiedene Ausführungen mit unterschiedlich vielen maximal möglichen und minimal nötigen Threads. Dabei werden beim kleinsten Minimum zwei Threads, und beim größten Maximum vier Threads ausgeführt [17, S. 2-3]. Entwickelt sind die Prozessoren in der Sprache VHDL [5]. Inzwischen existiert auch eine Variante, welche die Vektorerweiterung von RISC-V unterstützt [18], sowie eine fehlertolerante Version [4].

Wie im letzten Abschnitt deutlich wird, ist die Anwendung von IMT bei RISC-V-Prozessoren ein bisher noch wenig bearbeitetes Feld. Obwohl im *Klessydra*-Projekt schon erste erfolgreiche Versuche gemacht worden sind, ist die Anwendung bisher doch recht begrenzt. Außerdem wurde mindestens bei RISC-V noch nicht öffentlich untersucht, wie sich die Implementierung von IMT auf ein bestehendes Design auswirkt, da die *Klessydra*-Prozessoren sämtlich Neuentwicklungen sind.

3.2 Offene RISC-V-Designs

Im Vorfeld des Entwurfs soll hier ein Überblick über aktuell bestehende RISC-V-Designs gegeben werden. Aus diesen soll später eines ausgewählt werden, um als Grundlage für die Implementierung von IMT zu dienen. Daher müssen alle aufgeführten Prozessoren bestimmte Eigenschaften aufweisen, die das möglich machen sollen:

Lizenz Das Design muss unter einer **offenen Lizenz** veröffentlicht sein, die Veränderung und Veröffentlichung desselben ermöglicht. Nur auf diese Weise kann eine Erweiterung ebenfalls veröffentlicht werden.

Befehlssatz Mindestens der **RV32I**-Befehlssatz sollte unterstützt werden. Im Idealfall werden auch keine zusätzlichen Erweiterungen unterstützt, um den Prozessor einfach zu halten. Dies soll aber kein Ausschlusskriterium sein.

HDL Da der Verfasser dieser Arbeit zwar Kenntnisse in Verilog besitzt, aber nicht in anderen HDLs, sollte das Design entweder in **Verilog oder SystemVerilog** entwickelt sein, um den Fokus vollständig auf die Entwicklung der IMT-Erweiterung legen zu können.

Dokumentation Es sollte ein Mindestmaß an Dokumentation in entweder **deutscher oder englischer Sprache** vorliegen, da der Verfasser dieser Arbeit andere Sprachen nicht ausreichend beherrscht.

Ausgehend von der bereits vorher genannten Auflistung bestehender RISC-V-Designs [9] sind nach obigen Kriterien relevante Prozessoren in Tabelle 3.1 aufgeführt. Dabei wird zuerst der offizielle Name angegeben. Ist dieser nicht klar, wird der Name des entsprechenden Quelltextrepositorys verwendet. Darüber hinaus wird die verwendete HDL, die Lizenz und der Befehlssatz (ISA) angegeben. Bei Letzterem handelt es sich um die maximal unterstützten RISC-V-Erweiterungen. Einige Prozessoren unterstützen die Deaktivierung bestimmter Erweiterungen zur Verringerung der benötigten Hardwareressourcen auf Kosten der Funktionalität. In der Spalte *Pipeline* wird angegeben, in wie viele Ausführungssphasen die Pipeline, sofern vorhanden, aufgeteilt ist. Ist es unter Besonderheiten nicht anders vermerkt, so sind die Prozessoren nicht superskalar, können also in jedem Taktzyklus maximal eine Instruktion ausführen und die Ausführung geschieht in-order. Zuletzt wird in der Spalte *Link* noch auf die Internetadresse des entsprechenden Quelltextrepositorys verwiesen. Das verwendete Design *PicoRV32* ist in der Tabelle hervorgehoben. Der Auswahlprozess wird im Kapitel *Entwurf* dargestellt.

Die Auflistung der insgesamt 14 relevanten Prozessoren zeigt einerseits, wie aktiv die RISC-V-Community ist, und andererseits, welche Bandbreite an Anwendungsfällen schon durch diese Designs abgedeckt werden kann. Allerdings handelt es sich bei allen Designs um Prozessoren, die in erster Linie für den Einsatz in eingebetteten Systemen ausgelegt sind. Das liegt auch am Zielmedium FPGA. Denn während herkömmlich produzierte Mikroprozessoren ausschließlich auf ihre konkrete Funktion optimiert sind, sind FPGAs in der Lage, unterschiedlichste logische Funktionen auszuführen, können also nicht im selben Maß optimiert sein. Daher befindet sich der Höchsttakt der aufgeführten

Prozessor	entwickelt in	Lizenz	ISA	Pipeline	Besonderheiten	Link
biRISC-V	Verilog	Apache	RV32IM	6 - 7 St.	superskalar	[15]
Clarvi	SystemVerilog	BSD	RV32I	6 St.		[14]
CV32E40P	SystemVerilog	SHL	RV32IMFC	4 St.		[8]
DarkRISCV	Verilog	BSD	RV32I	2 St.		[3]
Ibex	Verilog	Apache	RV32IMC	2 St.		[6]
mriscvcore	Verilog	MIT	RV32IM	3 St.		[7]
PicoRV32	**Verilog**	**ISC**	**RV32IMC**	**keine**	**ohne Pipelining**	**[36]**
RISC-V	Verilog	BSD	RV32IM	5 St.		[16]
RV32CPU	SystemVerilog	ISC	RV32I	5 St.		[1]
RSD	SystemVerilog	BSD	RV32IM	11 - 13 St.	OoO, supersk.	[11]
SCR1	SystemVerilog	SHL	RV32IMC	2 - 4 St.		[13]
SSRV	SystemVerilog	Apache	RV32IMC	4 - 5 St.	OoO, supersk.	[10]
SweRV EH1	SystemVerilog	Apache	RV32IMC	9 St.	superskalar	[2]
Taiga	SystemVerilog	Apache	RV32IMA	3 St.		[12]

Tabelle 3.1: Übersicht relevanter RISC-V-Designs

Designs in aller Regel unterhalb der 1-GHz-Grenze. Aufgrund der daher eher niedrigen maximalen Leistung sind sie in erster Linie für den Einsatz in eingebetteten Systemen geeignet. Auch die Möglichkeit, zusätzliche, spezialisierte Logik auf demselben Gerät bzw. FPGA zu implementieren, um besonders rechenintensive Aufgaben zu beschleunigen, trägt zu einer besonderen Eignung für diese Systeme bei.

Deutlich wird auch, dass kein Prozessor aufgeführt ist, der irgendeine Form von Multithreading auf Hardwareebene unterstützt. Ein Pipelining wird hingegen in verschiedenen Varianten von 13 der Prozessoren in unterstützt. Hier zeigt sich, dass die große Mehrheit der Designs weiterhin auf Pipelining setzt, trotz der oben genannten Erkenntnisse über den Einsatz von IMT in eingebetteten Systemen. Eine Erklärungsmöglichkeit dafür ist die bereits genannte Erschwerung der Programmierbarkeit von Prozessoren mit Multithreading.

4 Entwurf

Der Entwurf der IMT-Implementierung hängt eng mit dem Design des Prozessors zusammen, für den diese erfolgen soll. Daher geht dieses Kapitel neben dem Entwurf der IMT-Implementierung auch auf den Auswahlprozess und den Aufbau des gewählten Ausgangsdesigns ein.

4.1 Idee und Ziel

Das Ziel des Entwurfs ist es, ein bestehendes Prozessordesign mit der Fähigkeit auszustatten, statt einem, mehrere Threads auszuführen. Dabei sollen die Threads nach dem Verfahren des Interleaved Multithreading abwechselnd ausgeführt werden.

Besonders wurde beim Entwurf versucht, den zusätzlich benötigten Hardwareaufwand in Grenzen zu halten. Aufgrund der begrenzten Zeit für den Abschluss dieser Arbeit ist das Design konzeptionell so einfach wie möglich gehalten und implementiert nur die notwendigsten Funktionen.

Wie bereits im Kapitel Grundlagen erwähnt, ist IMT auf Hardwareebene nur von den *Structural Hazards* betroffen, bei denen Probleme bei gleichzeitigem Zugriff auf dieselbe Hardware oder Ressource auftreten. Diese werden durch das Design von vornherein ausgeschlossen. Dies kann etwa durch Multiplikation von Registern, Vervielfachung von Datenpfaden oder ähnliche Maßnahmen geschehen.

Auf diese Weise soll die Eignung von IMT als beschleunigende Erweiterung eines bestehenden Prozessors bei überschaubarem zusätzlichem Aufwand überprüft werden.

4.2 Auswahl eines geeigneten Ausgangsdesigns

Im vorherigen Kapitel Stand der Technik wurden in Tabelle 3.1 bereits alle prinzipiell in Frage kommenden Designs aufgeführt. Für die Auswahl eines geeigne-

ten Prozessors gelten folgende Punkte als Leitlinie und wurden gegeneinander abgewogen:

Komplexität Der wohl wichtigste Punkt bei der Auswahl eines passenden Designs ist dessen Komplexität. Da bei geringer Komplexität insgesamt weniger Funktionen unterstützt werden müssen, ist die Erwartung, dass die Erweiterung eines einfachen Designs weniger Entwicklungsaufwand erfordert.

Speicheranbindung bzw. -organisation Um die bereits erwähnten *Structural Hazards* zu unterbinden, ist vor allem ein gleichzeitiger Zugriff auf Instruktionen und Daten wichtig, da kein Thread auf eine neue Instruktion warten müssen soll, nur weil ein anderer Thread gleichzeitig auf Daten zugreift. Hierfür ist ein getrennter Daten- und Instruktionspfad oder sogar ein getrennter Daten- und Instruktionsspeicher nötig. So kann gleichzeitig eine Datenoperation ausgeführt und eine neue Instruktion geholt werden. Ist dies nicht der Fall, kann eine solche Speicheranbindung allerdings auch mit überschaubarem Aufwand implementiert werden.

Pipelineorganisation Um zu zeigen, wie hoch der Aufwand von IMT gegenüber einem Prozessor ohne Pipelining ist, sind Designs ohne Pipelining als Basis ideal. Sollte Pipelining doch unterstützt werden, so sollte es sich um eine möglichst einfache Implementierung handeln, die bspw. ohne Sprungvorhersagen funktioniert, um möglichst wenig unterstützte Funktionalität aus dem Design entfernen zu müssen. Außerdem kann der Prozessor nicht superskalar sein, da sonst nicht IMT, sondern SMT der Ansatz wäre, der am meisten Erfolg verspricht.

SoC-Implementierung Neben dem eigentlichen Prozessorkern gibt es noch weitere Merkmale, welche die Entscheidung beeinflussen. Dazu gehört, ob das Design bereits in einem passenden SoC implementiert wurde. Dies ist Voraussetzung für das tatsächliche Testen des Designs, da hier zum Beispiel der Speicher, aber auch Zugriff auf externe Schnittstellen wie UARTs implementiert und geregelt wird. Auch für das sinnvolle Testen auf echter Hardware ist ein SoC unerlässlich. Auch hier ist eine Implementierung mit überschaubarem Aufwand möglich.

Tests und Benchmarks Sollten dem Design bereits Benchmarks oder andere Tests beiliegen, spart das späteren Portierungsaufwand.

Community und Dokumentation Eine große Community und möglichst umfassende Dokumentation des Prozessors sind eine große Hilfe bei der Entwicklung. Die jeweilige Qualität kann aber leider nicht immer korrekt vorhergesagt werden.

Nach Abwägung der Punkte, stellte sich der *PicoRV32* als die beste Wahl dar. Optimiert auf geringe Größe und mit vielen deaktivierbaren Funktionen, zeichnet sich das Design durch geringe Komplexität aus. Die Speicheranbindung muss allerdings noch überarbeitet werden. Der Prozessor ist nicht superskalar und unterstützt kein Pipelining im in Kapitel 2 beschriebenen Sinne. Es werden allerdings trotzdem einige Aufgaben überlappend ausgeführt, etwa das Schreiben in Register und das Holen einer neuen Instruktion. Es besteht eine SoC-Implementierung (PicoSoC) und es existieren einige Tests und eine Portierung des *Dhrystone-Benchmarks*. Der Umfang der Dokumentation ist im Gegensatz zu anderen Designs im mittleren Bereich, enthält aber die wichtigsten Informationen, wenn auch auf unstrukturierte Art und Weise. Ein weiterer Vorteil ist die im Vergleich zu vielen der anderen Designs aktive Community. Auch wenn die Punkte nicht perfekt erfüllt sind, so stellt sich der *PicoRV32* doch als solide Ausgangsbasis für die Arbeit dar.

4.3 Aufbau des Ausgangsdesigns

Da der *PicoRV32* verschieden konfigurierbar ist, ist hier kurz diejenige Konfiguration beschrieben, die als Ausgangsbasis für die Erweiterung dient. Jegliche Unterstützung für Instruktionen abseits der im RV32I-Befehlssatz enthaltenen ist deaktiviert. Das heißt, dass der volle Registersatz zur Verfügung steht, weder Multiplikation noch Division unterstützt werden und die Instruktionen die üblichen 32 Bit lang sein müssen. Auch die eigenen Instruktionen zur Behandlung von Interrupts sind nicht aktiv. Das Design bietet über das *Pico Co-Processor Interface*, kurz *PCPI*, die Möglichkeit zusätzliche Instruktionen in externer Logik zu implementieren. Auch diese Funktionalität ist deaktiviert. Ebenfalls nicht aktiv sind die verschiedenen Mechanismen, Operationen auf zwei Takte zu strecken, um einen höheren Maximaltakt zu ermöglichen. All diese Deaktivierungen dienen der Vereinfachung des Designs, in dem es auf die Funktionalität reduziert wird, die zur Beantwortung der Forschungsfrage nötig ist.

Aktiviert ist die Verwendung von zwei Registerports, durch die zwei Register gleichzeitig gelesen, oder eines beschrieben und eines gelesen werden kann. Außerdem ist der Barrel-Shifter aktiv, der die Berechnung des Ergebnisses von Instruktionen zum Verschieben auf einen Takt verkürzt [37]. Dadurch wird in beiden Fällen eine Ausführungsphase des Prozessors überflüssig (`cpu_state_ld_rs2` und `cpu_state_shift`), dienen also auf diese Weise ebenfalls der Vereinfachung des Designs.

Der Aufbau eines so konfigurierten *PicoRV32* kann wie in Abbildung 4.1 vereinfacht dargestellt werden. Dabei sind parallel arbeitende Elemente übereinander

positioniert. Gezeigt sind außerdem die wichtigsten Module und Verbindungen. Interessant für das Thema dieser Arbeit ist aber insbesondere der Ablauf und die Überlappung der Ausführungsphasen, weswegen auf eine detailliertere Beschreibung der Funktionsweise verzichtet wurde.

Während sich der Prozessor in der Phase `cpu_state_fetch` befindet, wird nicht nur die nächste Instruktion geholt, sondern auch dekodiert. Außerdem wird die Adresse der nächsten Instruktion bestimmt und das Ergebnis der vorherigen Instruktion in die Register geschrieben.

In der Phase `cpu_state_ld_rs1` werden je nach auszuführender Operation die jeweiligen Operatoren aus dem Register gelesen.

Zuletzt wird eine der drei Ausführungsphasen durchlaufen. Je nach Operation geht der Prozessor in die Phase `cpu_state_ldmem` oder `cpu_state_stmem` im Falle einer Speicheroperation oder in die Phase `cpu_state_execute` für eine ALU-Operation, also eine Rechenoperation.

Durch das situationsabhängige Vorladen von Instruktionen und anderen ähnlichen Techniken, die hier aus Gründen der Übersichtlichkeit nicht dargestellt sind, liegt die durchschnittliche CPI nach Angabe der Dokumentation in einer vergleichbaren Konfiguration bei vier Taktzyklen pro Instruktion. Dabei werden mindestens drei Takte und höchstens sechs Takte pro Instruktion benötigt [37].

4.4 Aufbau der Erweiterung

Ausgehend von der Analyse des bestehenden Aufbaus im vorigen Abschnitt, können nun konkrete Änderungen gezeigt werden, die zur Umsetzung des IMT führen.

1. Die Implementierung einer zusätzlichen, nur lesenden Schnittstelle zum Speicher für Instruktionen, die Multiplikation der Registersätze und der internen Ergebnisregister, sowie ein Mechanismus zur Rotation der aktiven Threads.
2. Techniken wie das Vorladen von Instruktionen müssen entfernt werden, da hier auf zuvor nicht gleichzeitig verwendete Hardwareressourcen zugegriffen wird, die allerdings nun von einem anderen Thread belegt sind.
3. Die Kommunikation mit der Software muss geregelt werden. Dafür wird eine zusätzliche Instruktion implementiert, über welche die Identifikationsnummer des aktuell ausgeführten Threads zurückgegeben wird. Diese

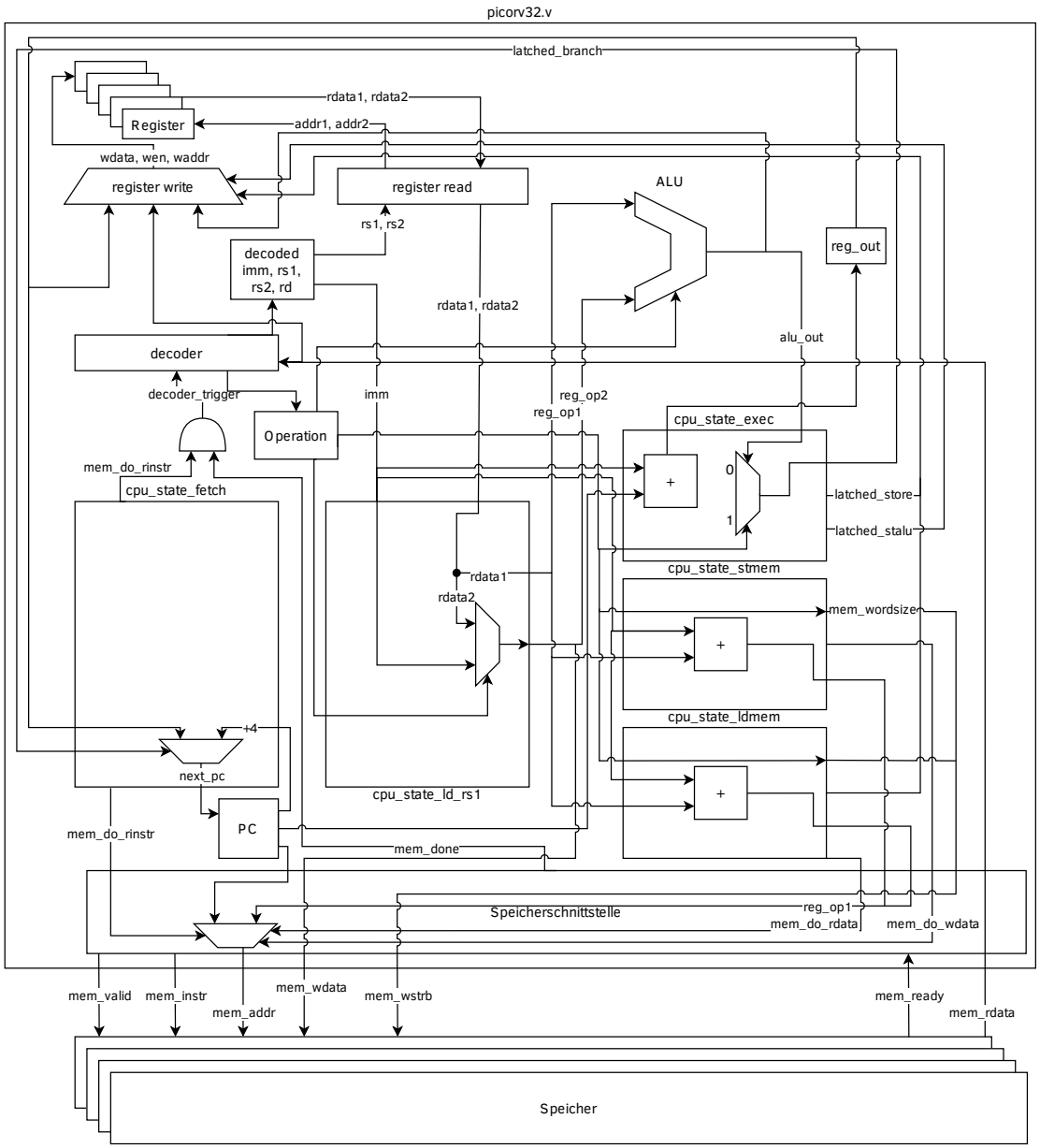

Abbildung 4.1: Vereinfachter Aufbau des PicoRV32 ohne Unterstützung für IMT

Instruktion ist eine bereits vorhandene RISC-V-Operation, bei der die sogenannten *Control Status Register* ausgelesen werden. Diese sind im Ausgangsdesign aber nicht vorhanden. Für die Erweiterung wird daher ausschließlich das Auslesen derjenigen Registeradresse unterstützt werden, in der die erwähnte Identifikationsnummer gespeichert ist.

Diese Veränderungen sollen anhand des vereinfachten Schaubilds verdeutlicht werden. Daher zeigt Abbildung 4.2 den vereinfachten *PicoRV32* mit den genannten Veränderungen. Diese Änderungen sind die Grundlage für die Implementierung der IMT-Unterstützung, die im gleichnamigen Kapitel 5 dokumentiert ist. Dieser angepasste Prozessor trägt den Namen *PicoRV32-imt*.

Im Schaubild werden alle Zwischenspeicher mehrfach dargestellt. Außerdem gibt es nun mehrere Registersätze. Im Beispiel handelt es sich um drei Threads, diese Anzahl ist aber nur zur Veranschaulichung gewählt und kann noch geändert werden. Die Speicherschnittstelle ist nun aufgeteilt in eine Instruktionsschnittstelle und eine Datenschnittstelle.

Ganz oben ist eine grobe Übersicht über die geplante Threadsteuerung dargestellt. Für jeden Thread, oder *hart* wie es in der RISC-V-Dokumentation genannt wird [32, S. 2], wird der aktuelle Status gespeichert. Damit ist gemeint, für welche Ausführungsphase der jeweilige Thread bereit ist. Dies wird in `hart_ready` gespeichert. Darüber hinaus wird für jede der Ausführungsphasen gespeichert, welcher Thread dieser momentan zugewiesen ist. Die Zuordnung der bereiten Threads zu der jeweiligen Phase erfolgt über die Threadauswahl, wobei der `hart_counter` sicherstellt, dass die richtige Reihenfolge eingehalten wird. Zu bemerken ist hier, dass die Threadsteuerung etwas dynamischer ablaufen wird, als in herkömmlichen IMT-Prozessoren. Da die jeweiligen Ausführungsphasen bei dem gewählten Design nicht gleich lang sind, wird der auszuführende Thread nicht in jedem Takt gewechselt, sondern immer dann, wenn derjenige Thread bereit wird, der als nächstes an der Reihe ist. Die Ausführungsreihenfolge bleibt also weiterhin fest, allerdings ist die Ausführungsdauer je nach Instruktion veränderlich.

4.5 Aufbau des dazugehörigen SoC

Da für die Verwendung des Prozessors neben dem bereits dargestellten Rechenkern Speicher und Ein- und Ausgaben benötigt werden, ist zusätzlich der Entwurf eines passenden SoCs notwendig. Wie bereits erwähnt, liefert das *PicoRV32*-Projekt u.A. eine SoC-Implementierung. Dieser *PicoSoC* bindet über die Speicherschnittstelle des *PicoRV32* einen SRAM-Arbeitsspeicher, einen UART-

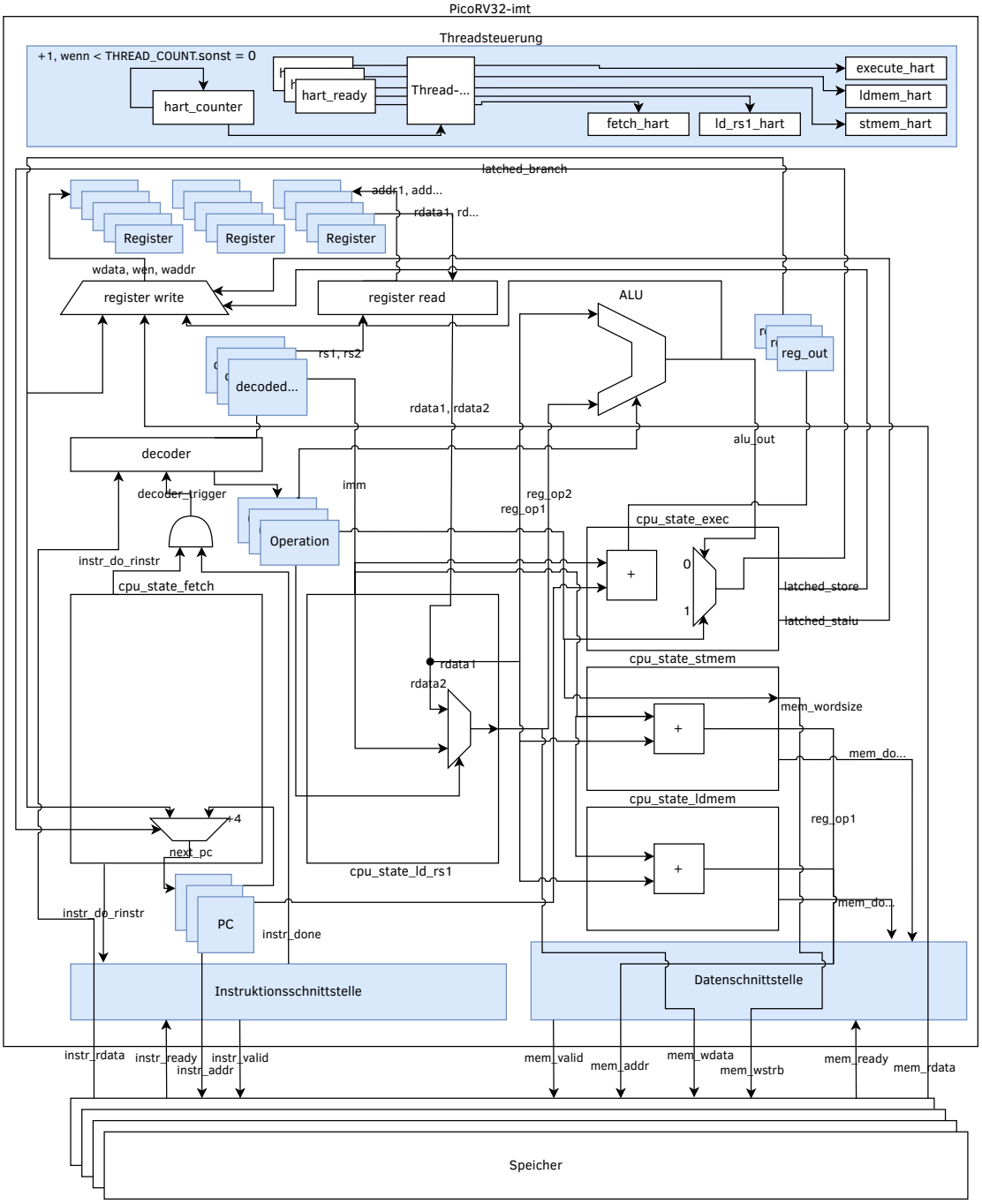

Abbildung 4.2: Vereinfachter Aufbau des PicoRV32 mit Unterstützung für IMT. Dabei sind die Änderungen gegenüber dem Ursprungsdesign eingefärbt.

Controller und einen Quad-SPI-Flash-Controller an. Dieser Flash-Controller verbindet den Prozessor mit einem entsprechenden nicht-flüchtigen Speicher, auf dem das auszuführende Programm erwartet wird. Um diese im Vergleich zum Arbeitsspeicher langsamen Zugriffe zu umgehen und das Design insbesondere im Hinblick auf die getrennten Daten- und Instruktionspfade zu vereinfachen, soll ein neuer SoC auf Grundlage des bestehenden entworfen werden, der Programme direkt aus dem Arbeitsspeicher heraus ausführt. Außerdem muss der Entwurf des Speichers überarbeitet werden, sodass Instruktionen und Daten zeitgleich übertragen werden können. Die UART-Anbindung soll beibehalten werden, um eine Möglichkeit der Ein- und Ausgabe zu behalten.

Um auch diesen Entwurf zu verdeutlichen, gibt Abbildung 4.3 einen Überblick über den Aufbau des neuen *PicoRAMSoC*. Die grundlegende Funktionsweise ist dabei soweit wie möglich mit dem *PicoSoC* kompatibel und unterscheidet sich nur an den notwendigen Stellen. Um eine bessere Übersichtlichkeit zu erhalten, sind die einzelnen Leitungen der Daten- und Instruktionspfade nur zwischen dem `PicoRV32-imt`- und dem `Weiterleitung nach Speicheradresse`-Block vollständig ausgeschrieben. Sonst werden diese als bidirektionale Verbindungen mit der Beschriftung `data` für den Datenpfad bzw. `instr` für den Instruktionspfad dargestellt. In dem genannten `Weiterleitung nach Speicheradresse`-Block wird anhand der vom Prozessor gegebenen Zieladresse entschieden, welches der Module auf der linken Seite für die Anfrage verantwortlich ist. Danach leitet der Block die Daten und die Adresse an das entsprechende Modul weiter. Noch anzumerken ist, dass die Datenverbindung zum UART-Controller vereinfacht dargestellt ist, da die konkrete Implementierung für diese Arbeit nur eine geringe Relevanz besitzt.

Dementsprechend wurde auch die Möglichkeit des *PicoSoC* übernommen, über Speicheradressen, die über den Adressbereich des internen Speichers hinausgehen, zusätzliche Ein- und Ausgabe zu ermöglichen. Dafür wird die Speicherschnittstelle des *PicoRV32* weitergeleitet und kann damit z.B., wie in der Referenzimplementierung, LEDs als Speicheradresse ansteuern und schalten. Auch der verwendete UART-Controller `simpleuart.v` wurde aus dem *PicoSoC* übernommen.

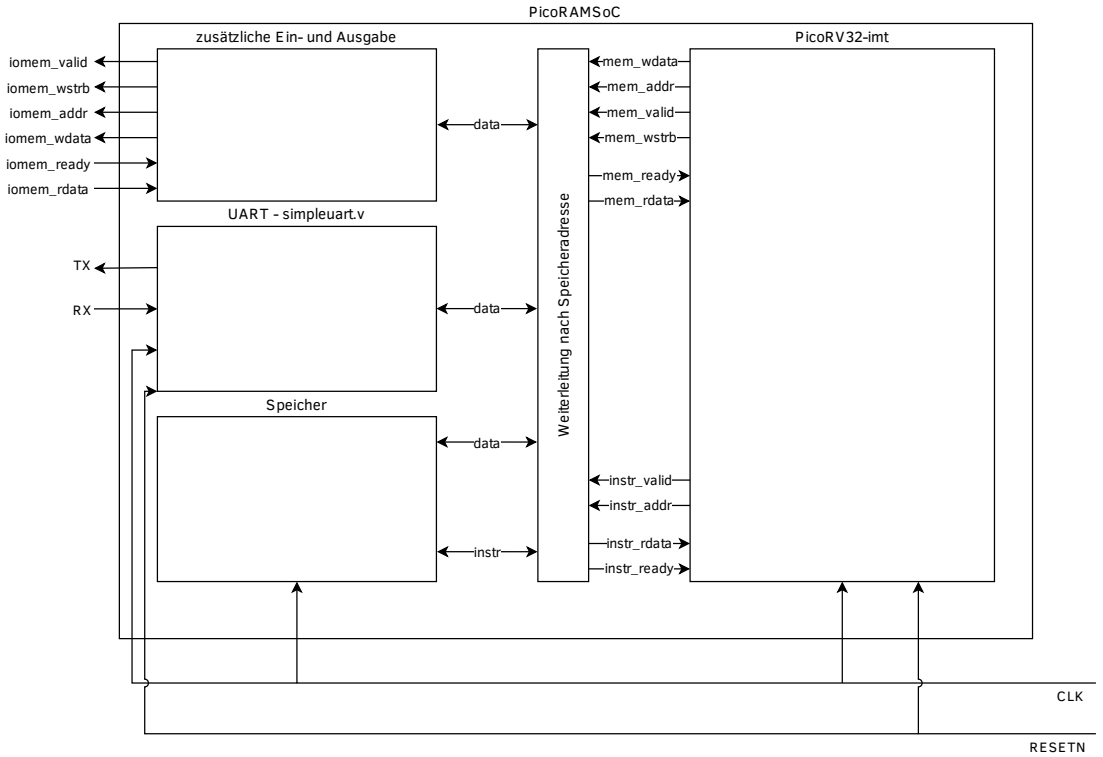

Abbildung 4.3: Aufbau des PicoRAMSoC

5 Implementierung

Nachdem der generelle Entwurf dargestellt ist, geht es in diesem Kapitel um den konkreten Prozess der Implementierung. Um das gewünschte Ergebnis zu erzielen, sind drei Probleme bzw. Aufgaben zu lösen.

Portierung Der *PicoSoC* wurde für das PicoRV32-Projekt auf verschiedene FPGA-Plattformen portiert. Für diese Arbeit steht keine dieser Plattformen zur Verfügung, dafür aber der *Digilent Basys 3*. Dieser ist mit einem *Xilinx Artix-7* ausgestattet. Deswegen wurde zuerst sicher gestellt, dass der *PicoRV32* überhaupt auf der *Artix-7*-Architektur implementiert werden kann.

Implementierung einer geeigneten SoC-IP Wie bereits im Kapitel *Entwurf* angesprochen, muss der bestehende SoC angepasst werden.

Implementierung des Interleaved Multithreading Hier findet die Lösung der Hauptaufgabe statt: Die Implementierung des IMT für den *PicoRV32*.

Am Ende des Kapitels soll zudem kurz auf Probleme und Grenzen der Implementierung eingegangen werden.

5.1 Portierung

Dass die Portierung des unveränderten *PicoRV32* auf den *Basys 3* als erster Schritt unternommen wurde, hat mehrere Gründe: Einerseits wird hierdurch sicher gestellt, dass das Design überhaupt auf dieser Architektur implementierbar ist. Dadurch ist bei späteren Implementierungsproblemen klar, dass sie erst durch die Veränderung zustande kamen und nicht von Anfang an vorhanden waren. Andererseits kann hier bereits ein grundlegendes Verständnis der Arbeitsweise des *PicoSoC* und des *PicoRV32* erarbeitet werden, ohne diese bereits verändern zu müssen. Darüber hinaus können auch die vorgesehene Verwendung dieser Projekte nachvollzogen und die Toolchains eingerichtet werden, sowohl für die HDL-Entwicklung, als auch für die Entwicklung von Software, die auf dem Prozessor ausgeführt wird.

Die bestehenden Referenzimplementierungen des *PicoRV32* sind alle für FPGAs der Firma Lattice bestimmt. Da die Toolchain für diese nicht kompatibel mit der für Xilinx-FPGAs ist, war die erste Aufgabe also die Einbindung des Projekts in die Xilinx-Toolchain. Dafür wurde ein entsprechendes Projekt in der Xilinx Entwicklungsumgebung *Vivado* erstellt. Daraufhin wurde ein Wrapper entwickelt, der die Ein- und Ausgabeverbindungen des *PicoSoCs* mit der entsprechenden Hardware des *Basys 3* verbindet. Das betrifft die UART-Schnittstelle und die Verbindung zum Quad-SPI-Flashspeicher des FPGA-Boards, aber auch die Ansteuerung der LEDs und den Taktgeber.

Außerdem musste die Firmware, also die auszuführende Software, so angepasst werden, dass die Besonderheiten des *Basys 3* berücksichtigt werden. Zuletzt wurden noch die wichtigsten Teile des Build-Prozesses kompatibel zum gewünschten FPGA-Board gemacht.

Am Ende der Entwicklung funktionierte die Übertragung der Firmware in den Flashspeicher, die Ausführung des Programms auf dem Prozessor und die Kommunikation über ein UART-Terminal, das an einem per USB verbundenen PC emuliert wurde.

5.2 Implementierung einer geeigneten SoC-IP

Dem Kapitel *Entwurf* folgend, wurde nun der SoC angepasst. Auch hier musste die Funktionsweise des Prozessorkerns selbst noch nicht angepasst werden. In diesem Schritt konnte also das Verständnis der Architektur weiter ausgebaut werden, bevor diese verändert wird.

Aufbauend auf der bestehenden Implementierung wurden zuerst nicht mehr benötigte Funktionen entfernt. Dabei geht es um alles, was mit dem Flashspeicher zusammenhängt. Auf jeder Ebene wurde also die Unterstützung dafür entfernt: Im SoC, im gerätespezifischen Wrapper und ebenso in der Firmware.

Dafür wurde der interne Speicher auf nun 4096 statt nur 256 Speicherworte vergrößert. Das entspricht 32 KiB. Außerdem wird die zu verwendende Firmware nun während der Synthese in das Design integriert und dann am Ende als Teil des Bitstreams auf den FPGA übertragen. Um das zu erreichen, wurde das Verilogmodul `picoramsoc_mem` um einen nur zu Beginn der Synthese auszuführenden `initial`-Block ergänzt. Dieser ist im Listing 5.1 gezeigt. Die Datei `firmware.hex` ist eine Textdatei, bei der in jeder Zeile genau eine achtstellige Hexadezimalzahl steht. Da jede Hexadezimalziffer 4 Bit belegt, enthält jede Zeile also 32 Bit an Informationen, also bspw. genau eine RV32-Instruktion, es können aber auch Daten gespeichert sein. Wichtig ist, dass an der Einsprung-

adresse des Prozessors, die über den Parameter `PROGADDR_RESET` festgelegt ist, die erste Instruktion des auszuführenden Programms gespeichert ist. Das heißt, wenn die Einsprungadresse z.B. auf `0x10`, also 16, konfiguriert ist, muss auch in der 16. Zeile der Datei `firmware.hex` die erste auszuführende Instruktion stehen.

```
module picoramsoc_mem #(
    parameter integer WORDS = 4096
    ) (
    input clk,
    input [3:0] wen,
    input [21:0] addr1,
    input [21:0] addr2,
    input [31:0] wdata,
    output reg [31:0] rdata1,
    output reg [31:0] rdata2
    );

    reg [31:0] mem [0:WORDS-1];

    /* Der Inhalt der Datei "firmware.hex" wird in den erzeugten
     * Speicher "mem" kopiert.
     */
    initial
        $readmemh("firmware.hex", mem);


    always @(posedge clk) begin
        rdata1 <= mem[addr1];
        if (wen[0]) mem[addr1][ 7: 0] <= wdata[ 7: 0];
        if (wen[1]) mem[addr1][15: 8] <= wdata[15: 8];
        if (wen[2]) mem[addr1][23:16] <= wdata[23:16];
        if (wen[3]) mem[addr1][31:24] <= wdata[31:24];
        rdata2 <= mem[addr2];
    end
endmodule
```

Listing 5.1: Speicherimplementierung des neuen SoCs

Neben dem direkten Laden der Firmware in den Speicher wurde dieser außerdem um einen zusätzlichen Lesezugang ergänzt, über den später Instruktionen im selben Takt gelesen werden sollen, wie Daten geschrieben oder gelesen werden. Dafür wurden ein Eingang `addr2` und ein Ausgang `rdata2` hinzugefügt und analog zu dem bereits bestehenden Lesezugang im `always`-Block hinzugefügt.

Nach der Implementierung erfolgte auch hier die erfolgreiche Ausführung eines Programms auf der Hardware des *Basys 3*.

5.3 Implementierung des Interleaved Multithreading

Der Hauptteil der praktischen Entwicklungsarbeit dieser Arbeit ist die Implementierung des Interleaved Multithreading für den *PicoRV32*. Dabei wurde nur so wenig Funktionalität des *PicoRV32* unterstützt, wie nötig ist, um das Thema dieser Arbeit zu beleuchten.

5.3.1 Bestimmung und Entfernung nicht benötigter Funktionalität

Im ersten Schritt wurden diejenigen Funktionen des Prozessors bestimmt, die nicht zur Klärung der Forschungsfrage nötig sind. Dabei handelt es sich in erster Linie um das bereits erwähnte *PCPI*; damit zusammenhängend auch die Unterstützung für die RV32M-Erweiterung, die intern über das PCPI implementiert wurde, und die Unterstützung für die RV32C-Erweiterung, durch welche sowohl die Speicherschnittstelle als auch die Dekodierung der Instruktionen sehr unübersichtlich wurden. Kleinere Vereinfachungen betrafen die Möglichkeit kürzerer Registersätze und des Auslagerns der Registermodule in eine externe Datei. Beide Funktionen wurden entfernt. Auch die recht umfangreichen Möglichkeiten für das Debugging wurden aus Zeitgründen nur teilweise portiert. Die meisten dieser Änderungen wurden am Anfang entschieden, aber erst im Verlauf der Arbeit durchgeführt, um zu verhindern, dass versehentlich wesentliche Funktionalität entfernt wird.

Nicht entfernt wurden die Möglichkeiten, die verschiedenen Ausführungsphasen in zwei Takte aufzuteilen und die Unterstützung der *PicoRV32*-eigenen Interruptinstruktionen. Diese Funktionen sind wünschenswert, da sie im ersten Fall den Ablauf des IMT flexibilisieren und im zweiten Fall den Prozessor für Anwendungen deutlich interessanter machen. Die Anpassung und das Testen dieser Elemente wäre allerdings über den Fokus dieser Arbeit hinaus gegangen, daher sind sie nicht funktionsfähig.

5.3.2 Erweiterung der Speicherschnittstelle

Um gleichzeitig Daten und Instruktionen aus dem Speicher lesen bzw. schreiben zu können, muss ein gleichzeitiger Zugriff auf den Speicher möglich sein. Dies wurde über die Implementierung einer neuen Speicherschnittstelle realisiert, die ausschließlich für das Abrufen neuer Instruktionen verantwortlich

ist. Die entsprechende Unterstützung durch den Speicher besteht bereits, allerdings muss diese auch noch in der Verbindungslogik des SoC implementiert werden. Hier ist aber nur eine Kopie der bestehenden Speicherlogik notwendig, weswegen darauf nicht weiter eingegangen wird.

Für die Implementierung im Prozessorkern selbst wurde auf Basis der bestehenden Schnittstelle eine neue entwickelt, die bestehende auf seine verbleibenden Funktionen reduziert und der Rest des Prozessors auf die Verwendung der jeweils korrekten angepasst.

Im Listing 5.2 ist dargestellt, wie die neue Schnittstelle arbeitet. Dabei wurde die Funktion minimal vereinfacht, da der Fall nicht abgedeckt wird, in dem der Prozessor zurückgesetzt wird (das Signal `resetn` wird auf niedriges Potenzial geschaltet).

```
output instr_valid;
output [31:0] instr_addr;

input instr_ready;
input [31:0] instr_rdata;

/* ... */

reg instr_state;
reg [31:0] instr_rdata_q;
reg instr_do_rinst;

wire instr_xfer = instr_valid && instr_ready;
wire [31:0] instr_rdata_latched = instr_xfer ? instr_rdata :
   instr_rdata_q;
wire instr_done = instr_xfer && instr_state && instr_do_rinst;
assign instr_valid = instr_do_rinst;
assign instr_addr = next_pc[31:2], 2'b00};

always @(posedge clk)
    if (instr_xfer)
        instr_rdata_q <= instr_rdata;

always @(posedge clk)
    case (instr_state)
        0: if (instr_do_rinst)
            instr_state <= 1;
        1: if (instr_xfer)
            instr_state <= 0;
    endcase
```

Listing 5.2: Speicherschnittstelle für Instruktionen

Das Holen einer neuen Instruktion wird über das Register `instr_do_rinstr` gesteuert. Unabhängig davon, ob das bereits erfolgte, wird aber bereits die Adresse dieser Instruktion aus dem Register `next_pc` in `instr_addr` geladen. Gibt der Prozessor nun in der `fetch`-Phase die Anweisung `instr_do_rinstr`, wird dem Speicher gemeldet, dass diese Adresse nun gültig ist, indem `instr_valid` auf `1` gesetzt wird. Dadurch ist bereits eine der Bedingungen erfüllt, durch die `instr_xfer` den Wert `1` annimmt. Dieses Signal gibt an, ob die Speicheroperation abgeschlossen ist. Die Schnittstelle befindet sich daraufhin im folgenden Taktzyklus Zustand `1`. In diesem wird auf die Vollendung einer Speicheroperation gewartet, während im Zustand `0` auf den Beginn einer solchen gewartet wird.

Sobald der Speicher die angeforderte Instruktion in `instr_rdata` zur Verfügung stellt, gibt er das Signal `instr_ready`. Dadurch ist nun `instr_xfer` auf den Wert 1 gesetzt, die Operation also erfolgreich abgeschlossen. Deswegen kann im folgenden Taktzyklus die Instruktion, die vom Speicher in `instr_rdata` übermittelt wird, im Register `instr_rdata_q` gespeichert werden. Der `instr_state` kehrt daraufhin auf Zustand `0` zurück, wartet also auf den Beginn der nächsten Operation. Die eben gelesene Instruktion wird in `instr_rdata_latched` aufbewahrt.

Da die vereinfachte Datenschnittstelle ganz ähnlich arbeitet und sich im Wesentlichen nur durch die Schreibmöglichkeit und die Herkunft der Zieladresse unterscheidet, wird auch hierauf nicht weiter eingegangen.

Auch dieser Implementierungsschritt konnte erfolgreich auf dem FPGA ausgeführt und getestet werden.

5.3.3 Threadsteuerung und Kommunikation mit der Software

Wie bereits erwähnt, musste die Threadsteuerung ein wenig anders implementiert werden, als bei IMT üblich. Dies liegt daran, dass das Design des *PicoRV32* nicht auf Pipelining ausgelegt ist und daher die verschiedenen Ausführungsphasen nicht gleich viele Taktzyklen benötigen.

Für die Threadsteuerung muss zuerst festgelegt werden, wie viele Threads der Prozessor ausführen soll. Dafür wurde ein neuer Parameter `THREADS` eingeführt, über den dies eingestellt werden kann. Darauf baut die gesamte Steuerung auf und bleibt dadurch dynamisch für unterschiedlich viele Threads konfigurierbar. Da der genannte Parameter allerdings momentan nur drei Bit groß ist, sind aktuell maximal sieben Threads theoretisch möglich. Warum nicht acht wird weiter unten erläutert. Dieser Wert lässt sich über die Vergrößerung des Pa-

rameters `THREADS` noch ändern. Wie sinnvoll das allerdings ist, wird im Kapitel *Auswertung* noch einmal angesprochen.

Die Steuerungslogik basiert darauf, dass jeder Ausführungssphase maximal ein Thread zugewiesen ist. Außerdem wird für jeden Thread gespeichert, welche Phase dieser als nächstes ausführen soll. Entsprechend der Bezeichnungen der Phasen im Originaldesign ergibt sich folgender Aufbau:

```
1 localparam [2:0] no_hart = 3'b111;
2 reg [2:0] trap_hart = no_hart;
3 reg [2:0] fetch_hart = no_hart;
4 reg [2:0] ld_rs1_hart = no_hart;
5 reg [2:0] exec_hart = no_hart;
6 reg [2:0] stmem_hart = no_hart;
7 reg [2:0] ldmem_hart = no_hart;
8
9 reg [7:0] hart_ready [0:THREADS-1];
```

Listing 5.3: Grundlage der Threadsteuerung

Jeder Thread besitzt eine implizite Identifikation oder `hart_id`, die durch die Position im `hart_ready`-Array gegeben ist. Diese ID wird dann den jeweiligen Phasen zugewiesen, also bspw. `fetch_hart`. Dadurch kann überall im Prozessor auf die Register und Ergebnisse des Threads, der gerade in der `fetch`-Phase beschäftigt ist, zugegriffen werden. Das liegt daran, dass diese Register und Ergebnisse entsprechend der Reihenfolge des `hart_ready`-Arrays aufgebaut sind. Die ID `3'b111`, also Sieben, ist für den `no_hart` reserviert, der einer Phase zugewiesen wird, wenn kein Thread diese aktuell bearbeitet. Die Belegung dieser ID ist der Grund, warum trotz der Größe des `THREADS`-Parameters von drei Bit nur sieben Threads unterstützt werden.

Der Übergang der bereiten Threads in die freien Phasen erfolgt über einen zweiteiligen Mechanismus, wie man im Listing 5.4 nachvollziehen kann. Im ersten Teil wird mit Hilfe des Registers `hart_counter` gesteuert, welcher Thread als nächstes, sobald er bereit ist, in die `fetch`-Phase übergehen darf. Dafür wird überprüft, ob der Thread mit der entsprechenden ID für diese Phase bereit ist und ob die Phase gerade von keinem anderem Thread belegt ist. Dann wird im `hart_ready`-Array vermerkt, dass dieser Thread nun `busy`, also beschäftigt ist, der Thread wird der `fetch`-Phase zugeordnet und der `hart_counter` erhöht, sodass der nächste Thread danach an der Reihe ist. Übersteigt der `hart_counter` die Anzahl der Threads, wird er auf `0` zurück gesetzt.

Im zweiten Teil werden die möglichen Kombinationen aus Thread und Ausführungsphase in Reihenfolge aktiv. Dabei sind die drei Phasen `exec`, `ldmem` und `stmem` jeweils exklusiv, da dies jene Phasen sind, in denen die in der Instruktion geforderte Operation ausgeführt wird.

```
if (hart_ready[hart_counter] == cpu_state_fetch &&
    fetch_hart == no_hart) begin

    hart_ready[hart_counter] = cpu_state_busy;
    fetch_hart = hart_counter;
    hart_counter = hart_counter + 1;
end

if (hart_counter >= THREADS)
    hart_counter = 0;

for (k = 0; k < THREADS; k = k + 1)
    case (hart_ready[k])
        cpu_state_ld_rs1:
            if (ld_rs1_hart == no_hart) begin
                hart_ready[k] = cpu_state_busy;
                ld_rs1_hart = k;
            end
        cpu_state_exec:
            if (exec_hart == no_hart &&
                stmem_hart == no_hart &&
                ldmem_hart == no_hart) begin

                hart_ready[k] = cpu_state_busy;
                exec_hart = k;
            end
        cpu_state_stmem:
            /* ... */
        cpu_state_ldmem:
            /* ... */
    endcase
```

Listing 5.4: Logik der Threadsteuerung

Freigegeben werden Thread und Phase entsprechend nach erledigter Arbeit durch folgende Zeilen:

```
hart_ready[exec_hart] = cpu_state_fetch;
exec_hart = no_hart;
```

Listing 5.5: Freigabe von Thread und Phase

Im Beispiel handelt es sich dabei um die `exec`-Phase. Nach dieser ist der Thread nun wieder für die `fetch`-Phase bereit und die `exec`-Phase für einen neuen Thread.

Nun musste auch die Abfrage der ID eines Threads via Software ermöglicht werden. Dafür wurde auf die `csrrs`-Instruktion des RV32-Befehlssatzes zurückgegriffen. Da der Prozessor die *Control Status Register*, auf die diese Instruktion zugreift, nicht unterstützt, wurde ausschließlich das für diese Operation

notwendige Register mit der Nummer `0xF14` [33, S. 10] implementiert. Anfragen an andere Register führen zu Fehlern. Die Implementierung selbst stellte kein aufwendiges Problem dar, es mussten nur die Dekodierung und die `exec`-Phase minimal erweitert werden, die aktuelle ID selbst konnte dadurch direkt aus dem `exec_hart`-Register gelesen werden.

5.3.4 Multiplikation der Register und Zwischenspeicher

Nachdem bereits der erste Schritt zur Vermeidung von *Structural Hazards* mit der Entwicklung einer zusätzlichen Speicherschnittstelle unternommen wurde, folgte jetzt der zweite. Auf Basis des Parameters `THREADS` wurden nun zuerst die Prozessorregister in der festgelegten Anzahl vervielfacht. Dafür wurde über einen `generate`-Block das vorhandene Modul `picorv32_regs` mehrfach initialisiert. Listing 5.6 zeigt dieses Vorgehen und stellt auch exemplarisch dar, wie andere Register oder Verbindungen anhand des Parameters `THREADS` multipliziert wurden.

```
reg cpuregs_write [0:THREADS-1];
reg [31:0] cpuregs_wrdata [0:THREADS-1];

wire [31:0] cpuregs_rdata1 [0:THREADS-1];
wire [31:0] cpuregs_rdata2 [0:THREADS-1];

wire [5:0] cpuregs_waddr [0:THREADS-1];
wire [5:0] cpuregs_raddr1 [0:THREADS-1];
wire [5:0] cpuregs_raddr2 [0:THREADS-1];

genvar i;
generate
    for (i = 0; i < THREADS; i = i + 1) begin : cpuregs
        picorv32_regs cpuregs (
            .clk(clk),
            .wen(cpuregs_write[i] && latched_rd[i]),
            .waddr(cpuregs_waddr[i]),
            .raddr1(cpuregs_raddr1[i]),
            .raddr2(cpuregs_raddr2[i]),
            .wdata(cpuregs_wrdata[i]),
            .rdata1(cpuregs_rdata1[i]),
            .rdata2(cpuregs_rdata2[i])
        );
    end
endgenerate
```

Listing 5.6: Register

Es mussten auch alle anderen Register vervielfacht werden, die threadspezifische Informationen zwischenspeichern. Dabei wurde nach demselben Muster

vorgegangen, wie eben beschrieben. Auch die jeweiligen Zuweisungen mussten z.T. multipliziert werden, außerdem musste bestimmt werden, dass das jeweils korrekte Register für den gewünschten Thread beschrieben oder ausgelesen wird. Speziell betraf diese Vervielfachung Ergebnisregister, wie die der ALU, Informationen zu der auszuführenden Instruktion, also die Operation, die Operanden und das Ziel, und besonders auch den Program Counter, in dem jeweils die Adresse der aktuellen bzw. nächsten Instruktion gespeichert ist.

Dieser Schritt stellte sich als der komplizierteste und zeitaufwendigste heraus. Der Grund dafür liegt beim Aufbau des *PicoRV32*. Dadurch, dass dieser ohne Pipeline entwickelt wurde, ist bei vielen Elementen nur schwer ersichtlich, zu welcher Ausführungsphase diese gehören, wodurch häufig erst erarbeitet werden musste, welchem Thread ein Element zugewiesen werden muss. Die vielen kleineren Optimierungen, wie das Vorladen von Instruktionen, vermischen die Phasen noch mehr. Ebenso stellte der komplexe und nicht dokumentierte Aufbau des Programmzählers häufig ein Problem dar, da es dadurch sehr aufwendig war, zu ermitteln welches Element des Zählers für welche Aufgaben gedacht ist und an welcher Stelle die gespeicherte Adresse jeweils wechselt. Darüber hinaus wurden durch den strengen Fokus auf wenig Ressourcen des Ausgangsdesigns auch andere Elemente untypisch implementiert. All das und die in Teilen recht sparsame Dokumentation führten dazu, dass die Entwicklung deutlich mehr Zeit in Anspruch nehmen musste, als ursprünglich geplant. Einige der daraus folgenden Versäumnisse werden im nächsten Abschnitt erläutert. Ideen zur Weiterarbeit werden noch einmal im Kapitel *Auswertung* angesprochen.

5.4 Probleme und Grenzen der Implementierung

Bei den letzten beiden Unterpunkten fehlt der Hinweis auf die erfolgreiche Synthese und Ausführung auf dem *Basys 3*. Das liegt daran, dass diese bisher nicht erfolgt sind. Hier zeigt sich das offensichtlichste Problem der Implementierung. Die Synthese zu ermöglichen wäre Aufgabe einer zukünftigen Arbeit sein.

Das andere Problem ist die unterschiedliche Länge der einzelnen Ausführungsphasen, durch die eine vollständig überlappende Ausführung nicht immer möglich ist. Es liegen in allen Konfigurationen immer wieder Ressourcen brach, wie in Tabelle 5.1 für den schlechtesten Fall gezeigt ist. Hier sind ab Takt vier durchschnittlich nur knapp 56% der Phasen ausgelastet. Dadurch ist die potentiell mögliche Beschleunigung begrenzt. Durch Unterstützung der Mechanismen zur Verlängerung bestimmer Ausführungsphasen, zusätzlicher Optimierung der bestehenden Phasen und besonders der Aufteilung der längeren Phasen in

	Ausführungsphase		
Takt	IF / ID / WB	RF	EX / MA
1	$T_0 I_0$	-	-
2	$T_0 I_0$	-	-
3	$T_0 I_0$	-	-
4	$T_1 I_0$	$T_0 I_0$	-
5	$T_1 I_0$	-	$T_0 I_0$
6	$T_1 I_0$	-	-
7	$T_0 I_1$	$T_1 I_0$	-
8	$T_0 I_1$	-	$T_1 I_0$
9	$T_0 I_1$	-	-
10	$T_1 I_1$	$T_0 I_1$	-
...			

Tabelle 5.1: Beispielhafter Programmablauf des *PicoRV32-imt* mit zwei Threads, wobei keine `load`- oder `store`-Instruktionen ausgeführt werden.

mehrere unabhängige, ist allerdings eine Minimierung dieser nicht genutzten Ressourcen denkbar.

6 Auswertung

Nach der Implementierung folgen in diesem Kapitel die Benchmarks der Erweiterung und des ursprünglichen Designs. Da die Synthese noch aussteht, können keine Messungen zur Taktbarkeit durchgeführt werden. Dadurch sind die Ergebnisse z.T. nur begrenzt relevant für den Einsatz in tatsächlichen Anwendungen, da nur die Anzahl der benötigten Takte allein nicht reicht, um die gesamte Ausführungszeit zu bestimmen. So ist nach [28, S. 49] die »cpu time« ein geeignetes Maß zur Messung der Prozessorleistung. Die Formel zur Berechnung derselben benötigt aber die Einbeziehung der Taktrate.

$$\text{cpu time} = \frac{\text{benötigte Taktzyklen für das Programm}}{\text{Taktrate}} \tag{6.1}$$

Aus diesem Grund wurde auch nicht der im Kapitel *Entwurf* angesprochene *Dhrystone-Benchmark* portiert, da der Aufwand dafür angesichts der ohnehin nur begrenzten Aussagekaft nicht gerechtfertigt wäre.

Dafür ist die Anzahl der benötigten Elemente auf dem FPGA eine relevante Größe, da sich durch diese exakt bestimmen lässt, auf welchen FPGAs diese Architektur implementierbar wäre und wie viel Platz noch für andere Module vorhanden wäre.

6.1 Auswahl der Benchmarks

Die Benchmarks und Tests werden zu zwei Bereichen ausgewählt: Leistung und Nutzung von Hardwareressourcen. Dabei wird die Leistung weniger intensiv getestet, als ursprünglich geplant, da die Ergebnisse ohne die Ausführung auf realer Hardware nur wenig Aussagekraft hätten. Stattdessen soll die benötigte Anzahl an Taktzyklen für ein sehr gut parallelisierbares Problem als besten Fall und die sequenzielle Lösung desselben Problems als schlechtesten Fall gemessen und mit dem Ausgangsdesign verglichen werden. Dafür werden der Prozessor und die Ausführung des entsprechenden Benchmarks mithilfe von *Vivado* simuliert. Außerdem wurden nur Konfigurationen mit bis zu sechs

Threads getestet, da sich bereits bei dieser Anzahl keine relevanten Änderungen mehr ergeben haben.

Das effektiv parallelisierbare Problem ist dabei die Matrix-Vektor-Multiplikation, nach dem Algorithmus, der in [20, S. 785] beschrieben wird. Hierbei wird eine $n \times n$-Matrix $A = (a_{ij})$ mit einem n-Vektor $x = (x_j)$ multipliziert. Der Ergebnisvektor $y = (y_i)$ ist hierbei von der folgenden Formel gegeben [20, s. 785]:

$$y_i = \sum_{j=1}^{n} a_{ij} x_j \tag{6.2}$$

Hierbei liegt i zwischen 1 und n.

Im dort gezeigten Beispiel wird eine parallele `for`-Schleife genutzt und deren Ausformulierung dem Compiler überlassen. Hier wird diese explizit implementiert. Der Quelltext ist Teil des digitalen *Anhangs* und in den Dateien `benchmark.c`, `benchmark_data.h` und `benchmark_data.c` zu finden. In letzterer ist auch die Matrix und der Vektor gezeigt, mit denen getestet wurde. Im Listing 6.1 ist beispielhaft gezeigt, wie die Threads ihre jeweilige Aufgaben verarbeiten. Der Thread im Beispiel hat die ID `0`. Jeder Thread berechnet die jeweils n-te Stelle des Ergebnisvektors, wobei n die Anzahl der Threads des Prozessors repräsentiert. Um alle Konfigurationen an derselben Matrix und demselben Vektor gleichberechtigt zu testen, wurde als Seitenlänge der Matrix 30 gewählt, da jede gültige Anzahl an Threads diese Zahl ohne Rest teilt. Die Ausnahme ist hier die Konfiguration mit vier Threads. Hier müsste eine 60×60-Matrix mit einem 60-Vektor multipliziert werden. Davon wurde hier abgesehen, da bereits die Simulation der gewählten Größe sehr viel Zeit in Anspruch nahm und außerdem ein viel größerer Speicher notwendig wäre. Schließlich ist die dann zu multiplizierende Matrix viermal so groß. Die Konfiguration mit sieben Threads hätte dies noch verstärkt.

```
done[0] = 0;
// DIMxDIM-Matrix
for (a = 0; a < DIM; a += THREADS) {
    result_vector[b] = 0;
    for (i = 0; i < DIM; i++) {
        result_vector[a] += matrix[i+a*DIM] * vector[i];
    }
}
done[0] = 1;
```

Listing 6.1: Ausschnitt des Benchmarkprogramms

Nachdem Thread `0` mit seiner Arbeit fertig ist, überprüft dieser, ob alle anderen Threads ebenfalls ihre Arbeit beendet haben und speichert in diesem Fall die Hexadezimalzahl `0xffff` im LED-Register des Prozessors. Das erfolgreiche Schreiben dieses Signals soll als Ende des Benchmarks gelten. Als Beginn soll das Setzen des `resetn`-Signals auf `0` dienen. Die Anzahl der zwischen diesen Ereignissen verstrichenen Taktzyklen ist das Ergebnis des Benchmarks.

Die Variante für nur einen Thread dieses Programms ist nach demselben Konzept aufgebaut. Dabei ist allerdings die parallele `for`-Schleife durch eine sequenzielle ersetzt. Auch hier ist das konkrete Programm im *Anhang* zu finden, diesmal aber in der Datei `benchmark_single.c`. Diese Version wird nicht nur auf dem Ursprungsdesign getestet, sondern auch nochmal auf den verschiedenen Konfigurationen des *PicoRV32-imt*, um deren Leistung bei Auslastung von nur einem einzelnen Thread zu zeigen. Dafür führen alle Threads außer Thread `0` nur eine Schleife von `nop`-Instruktionen aus, während Thread `0` die Aufgabe löst. Dies soll zeigen, wie viele Taktzyklen im schlechtesten Fall zusätzlich benötigt werden. Der schlechteste Fall ist hierbei derjenige, in dem nur ein Thread beschäftigt wird.

Alle Konfigurationen des angepassten Designs, sowie das Original in einer vergleichbaren Konfiguration (siehe auch Listing 6.2) werden mit dem *PicoRAM-SoC* und dem entsprechenden Wrapper für das *Basys 3* simuliert.

Um die verwendeten Hardwareressourcen zu vergleichen, wird ein Benchmark verwendet, der vom *PicoRV32*-Projekt entwickelt wurde [38]. Dabei wird in der bestehenden Fassung der Prozessor je einmal in seiner Standardkonfiguration, einer minimalen und zum Schluss einer maximalen Konfiguration mithilfe von *Vivado* synthetisiert und es werden die benutzten Ressourcen angezeigt. Für die Messung des Originaldesigns wurde hier nur die Standardkonfiguration so angepasst, dass sie etwa der Konfiguration des *PicoRV32-imt* entspricht. Die genaue Konfiguration ist im Listing 6.2 dokumentiert.

```
1  ENABLE_COUNTERS = 1,
2  ENABLE_COUNTERS64 = 1,
3  ENABLE_REGS_16_31 = 1,
4  ENABLE_REGS_DUALPORT = 1,
5  LATCHED_MEM_RDATA = 0,
6  TWO_STAGE_SHIFT = 0,
7  BARREL_SHIFTER = 1,
8  TWO_CYCLE_COMPARE = 0,
9  TWO_CYCLE_ALU = 0,
10 COMPRESSED_ISA = 0,
11 CATCH_MISALIGN = 1,
12 CATCH_ILLINSN = 1,
13 ENABLE_PCPI = 0,
14 ENABLE_MUL = 0,
```

```
15 ENABLE_FAST_MUL = 0,
16 ENABLE_DIV = 0,
17 ENABLE_IRQ = 0,
18 ENABLE_IRQ_QREGS = 0,
19 ENABLE_IRQ_TIMER = 0,
20 ENABLE_TRACE = 0,
21 REGS_INIT_ZERO = 0,
22 MASKED_IRQ = 32'h 0000_0000,
23 LATCHED_IRQ = 32'h ffff_ffff,
24 PROGADDR_RESET = 32'h 0000_0000,
25 PROGADDR_IRQ = 32'h 0000_0010,
26 STACKADDR = 32'h ffff_ffff
```

Listing 6.2: Konfiguration des *PicoRV32*

Derselbe Benchmark wurde für den *PicoRV32-imt* minimal angepasst, um diesen vergleichbar zu testen. Dabei wurde die Anzahl der Threads parametrisiert, sodass die benötigten Ressourcen für verschiedene Threadkonfigurationen gemessen werden können.

6.2 Ergebnisse

Die Ergebnisse der Benchmarks sind im Folgenden aufbereitet dargestellt. Die vollständigen unbearbeiteten Messdaten finden sich im *Anhang*.

6.2.1 Ressourcen

Die Abbildung 6.1 zeigt den Verbrauch der FPGA-Ressourcen des Originaldesigns und der verschiedenen Konfigurationen des angepassten Prozessors. Für diese wird jeweils die Anzahl der benötigten LUTs, Register und Multiplexer dargestellt. Deutlich ist dabei, dass die IMT-Version einen deutlich höheren Ressourcenverbrauch hat. Zwischen den verschiedenen Threadkonfigurationen ist der zusätzliche Aufwand hingegen spätestens bei solchen mit mehr als vier Threads recht gering.

So steigt die Summe der benötigen Ressourcen vom *PicoRV32* zur IMT-Version mit zwei Threads um etwa 38%. Sind sechs Threads konfiguriert, steigt der Verbrauch nur noch um rund 7%, also insgesamt um etwa 47% im Vergleich zum Original.

Die durchschnittlichen Kosten je zusätzlichem Thread belaufen sich auf eine Steigerung um etwa $1,6\%$. Lässt man die Konfiguration mit vier Threads außen

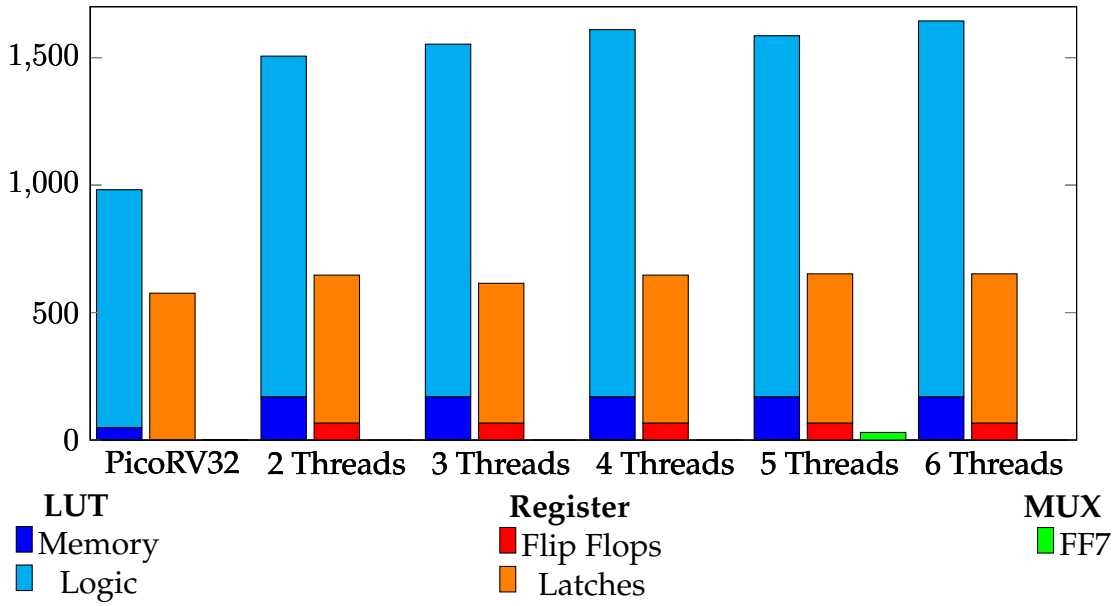

Abbildung 6.1: Verwendete Ressourcen

vor, die mit einem Zuwachs um ca. 4% aus der Reihe fällt, wächst der Prozessor sogar nur um rund $0,8\%$ für jeden zusätzlichen Thread.

Gründe für die hohen Zusatzkosten gegenüber dem *PicoRV32* sind z.B. die zusätzliche Speicherschnittstelle und die Threadsteuerung. Auch die mangelnde Optimierung spielt eine Rolle. Um die genauen Gründe und ihren Anteil an den Ergebnissen herauszufinden, sind aber noch weitere Nachforschungen und Tests nötig. Das gilt auch für die vergleichsweise hohen Kosten der Konfiguration mit vier Threads.

Zu bemerken ist, dass die Summe aller Ressourcen keine Vergleichsgröße ohne Probleme ist, da von den verschiedenen Elementen nicht immer die gleiche Anzahl verfügbar ist und nicht alle gleich universell einsetzbar sind, die Anzahl der Ressourcen also nur begrenzt auf diese Weise verglichen werden können. Dies kann die Summe hier nicht abbilden, auch weil sich die Anzahl und das Verhältnis der zur Verfügung stehenden Ressourcen auf verschiedenen FPGAs unterscheiden und daher keine universelle Angabe möglich ist.

6.2.2 Leistung

Wie bereits erwähnt, sollen für die Einschätzung der Leistung einer der besten und einer der schlechtesten Fälle getestet werden. Im besten Fall wird ein Programm ausgeführt, das sehr gut parallelisierbar ist. Diesen sieht man in Abbil-

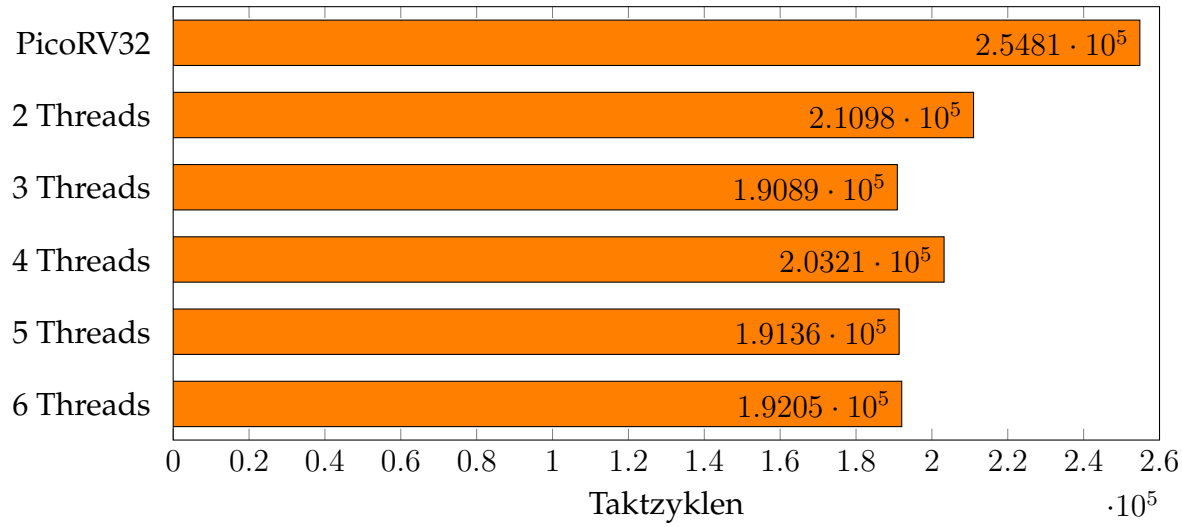

Abbildung 6.2: Benötigte Taktzyklen zur Multiplikation einer 30×30-Matrix mit einem 30-Vektor, alle Threads arbeiten

dung 6.2. Hier wird deutlich, dass jede Threadkonfiguration weniger Taktzyklen benötigt als der *PicoRV32*.

Sind zwei Threads verfügbar, benötigt das neue Design nur etwa 83% der Taktzyklen des Originals. Im Idealfall würden nur noch 50% benötigt werden. Dass es mehr sind, kann verschiedene Gründe haben. So gibt es einen Mehraufwand in der Software, durch die Zuteilung der Arbeit an verschiedene Threads. Außerdem gibt es auch bei dem für die gute Parallelisierbarkeit gewählten Problem noch einen kleinen sequenziellen Teil, der nicht parallel ausgeführt wird, wodurch in diesem kein Vorteil durch die Verfügbarkeit mehrerer Threads vorhanden ist. Darüber hinaus hat das Originaldesign Optimierungen, wie das Laden der nächsten Instruktion, bevor der Prozessor die *fetch*-Phase erreicht. Diese Optimierungen verringern die insgesamt nötigen Taktzyklen, sind aber, wie im Kapitel *Entwurf* bereits dargestellt, nicht mit IMT zu verbinden. Der nächste Punkt schließt sich daran an, denn das angepasste Design ist aus Zeitgründen nur begrenzt optimiert. Hier könnten wahrscheinlich noch weniger Taktzyklen erreicht werden.

Dass das Potential des *PicoRV32-imt* bei zwei Threads noch nicht optimal ausgenutzt wird, zeigt die Konfiguration mit drei Threads. Hier werden im Vergleich zum Ausgangsdesign nur noch rund 75% der Taktzyklen benötigt. Daran zeigt sich, dass der Prozessor häufig an drei Instruktionen gleichzeitig arbeiten kann, also alle drei Ausführungsphasen ausgelastet sind. Betrachtet man die Konfigurationen mit mehr Threads, wird auch deutlich, dass hier die maximale Leistung erreicht wird. Mit fünf oder sechs Threads wird der Prozessor nicht mehr schnel-

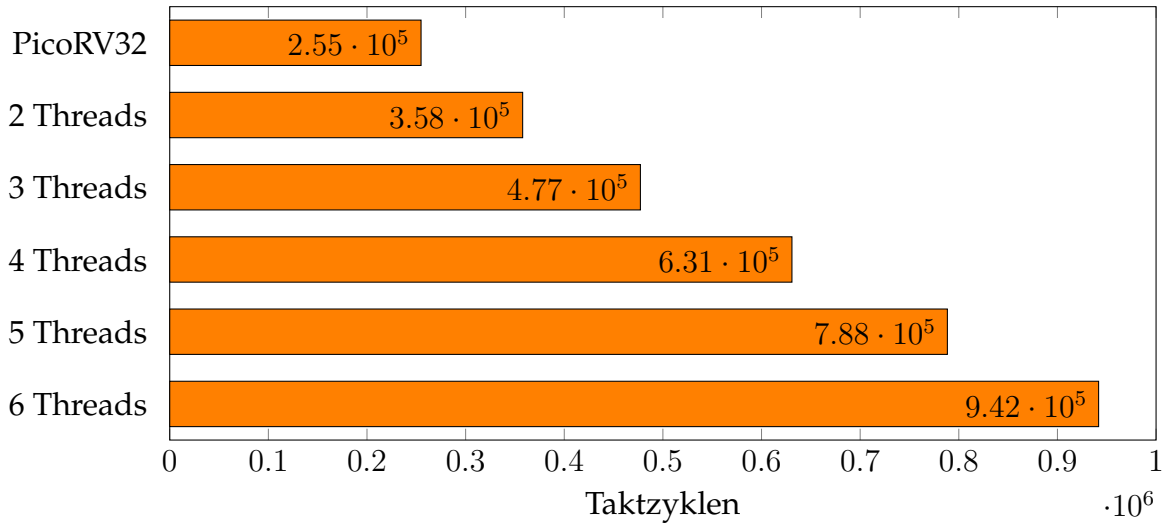

Abbildung 6.3: Benötigte Taktzyklen zur Multiplikation einer 30×30-Matrix mit einem 30-Vektor, nur ein Thread arbeitet

ler, sondern im Gegenteil minimal langsamer. Das kann durch den zusätzlichen Software- und Hardwareaufwand erklärt werden, der für die Koordination der neuen Threads nötig ist. Dieses Ergebnis entspricht etwa den Erwartungen, da der Prozessor nur drei Ausführungsphasen besitzt und daher maximal drei Threads zur gleichen Zeit bearbeiten kann. Jeder zusätzliche Thread muss dann also warten.

Das Ergebnis der Konfiguration mit vier Threads lässt sich dadurch erklären, dass die gewählte Dimension der Matrix und des Vektors nicht ohne Rest durch die Anzahl der Threads geteilt werden können. Dadurch müssen Threads bereits warten, während andere noch arbeiten. Würde man eine 60×60-Matrix mit einem 60-Vektor multiplizieren, ist eine im Vergleich zu drei Threads etwas höhere Anzahl an Taktzyklen zu erwarten.

In der Abbildung 6.3 ist das Ergebnis einer der schlechtesten Fälle für den Prozessor zu sehen. Hierbei wurde ebenfalls eine Matrix mit einem Vektor multipliziert, allerdings wurde die Berechnung immer nur von einem Thread ausgeführt. Alle anderen Threads führen solange nur eine Schleife von `nop`-Instruktionen, also solche ohne Funktion, aus, bis der arbeitende Thread fertig ist.

Hier zeigt sich, wie jeder zusätzliche Thread die Anzahl benötigter Taktzyklen vermehrt. Bereits die Konfiguration mit zwei Threads erhöht die Taktzyklen um etwa 40%. Durchschnittlich erhöht jeder zusätzliche Thread die Ausführungsdauer um ca. 30%. Dabei benötigt die Konfiguration mit sechs Threads fast 3,7 mal mehr Taktzyklen als das Ausgangsdesign.

Eine perfekte IMT-Implementierung für einen Prozessor ohne irgendwelche Formen des Parallelismus auf Instruktionsebene wäre hier in einer Konfiguration mit genauso vielen Threads wie Ausführungsphasen nicht langsamer als der ursprüngliche Prozessor. Da hier allerdings, wie im vorigen Kapitel gezeigt, teilweise Ressourcen brach liegen und auch der Ausgangsprozessor bereits gewisse Aufgaben überlappend ausgeführt hat, zeigt sich hier eine schwache Seite des *PicoRV32-imt*.

Wie am Anfang dieses Kapitels beschrieben, reichen diese Tests nicht aus, um eine abschließende Aussage über die Leistung treffen zu können. Erst der Test auf der echten Hardware unter Beachtung der möglichen Taktraten ermöglicht es, tatsächlich Aussagen hierüber zu treffen. Die bisherigen Ergebnisse können aber Ausgangspunkt sein, die Entwicklung weiterzuführen, um das tatsächliche Potenzial der Erweiterung bewerten und dann auch anwenden zu können.

7 Zusammenfassung und Ausblick

Wie wirkt sich die Implementierung von Interleaved Multithreading auf ein bestehendes, offenes RISC-V-Prozessordesign in Bezug auf Leistung und Ressourcenverbrauch aus? Diese Frage zu beantworten war Ziel dieser Arbeit. Dafür wurden zuerst die technischen Grundlagen erklärt. Besprochen wurde, was Pipelining und (Interleaved) Multithreading sind, und auf welche Weise sie sich ähneln und unterscheiden. Die Befehlssatzarchitektur RISC-V wurde vorgestellt und es gab eine Erklärung was Softcores genau sind. Danach folgte ein tieferer Einstieg in das Thema, der aktuelle Stand bei RISC-V, IMT und ihrer Kombination. Außerdem wurden die Designs vorgestellt, aus denen das in der Forschungsfrage erwähnte »bestehende, offene RISC-V-Prozessordesign« gewählt wurde. Darauf folgte die Beschreibung dieses Designs, der Erweiterung und des dazugehörigen SoCs. Die Implementierungsschritte wurden erklärt und daraufhin die entstandene Erweiterung eingeordnet, getestet und ausgewertet.

Diese Auswertung zeigt sowohl Erfolge als auch Probleme der IMT-Erweiterung. Es hat sich gezeigt, dass offene Prozessordesigns wie der *PicoRV32* tatsächlich geeignet sind, als Ausgangsbasis für weitgehende Veränderungen der Funktionalität. Es ist gelungen Interleaved Multithreading in einem Prozessor zu implementieren, der dies vorher nicht beherrschte. Hierbei ist die dynamisch konfigurierbare Anzahl von Threads hervorzuheben, die eine flexible Ausgangsbasis für zukünftige Erweiterungen darstellt. Darüber hinaus konnte nachgewiesen werden, dass das erweiterte Design für die Lösung geeigneter Probleme weniger Taktzyklen benötigt als das ursprüngliche.

Zu den Problemen gehört, dass die Implementierung in ihrer gegenwärtigen Form nicht erfolgreich synthetisiert und auf einem FPGA ausgeführt wurde. Daraus leitet sich ein weiteres Problem ab: Die tatsächliche Leistung des Designs kann nicht eingeschätzt werden, da dafür der Höchsttakt bekannt sein muss. Nur so ist das Design realistisch einzuschätzen. Die in der *Einleitung* geäußerte Erwartung eines geringen zusätzlichen Hardwareaufwands wurde nicht erfüllt. Die Änderungen in ihrer aktuellen Form erhöhen auch im besten Fall die verwendeten Ressourcen mehr, als sie die benötigten Taktzyklen verringern. Zuletzt ist auch die deutliche Erhöhung der benötigten Taktzyklen im sequentiellen Fall ein Ergebnis, das hinter dem theoretischen Potenzial einer IMT-Implementierung zurück bleibt.

Viele dieser Probleme lassen sich auf die Auswahl des *PicoRV32* als Ausgangsdesign zurückführen. Entgegen der Erwartung, dass ein besonders kompaktes Design ohne Pipeline den Entwicklungsaufwand klein halten würde, sorgten besonders der atypische Aufbau der Ausführungsphasen und das Fehlen einer zweiten Speicherschnittstelle für eine aufwendige Implementierung. Da keine entsprechenden Designdokumente existieren, wurden diese Probleme erst im Verlauf der Arbeit deutlich. Ein Prozessor, der bereits eine einfache Pipeline und einen getrennten Bus für Daten und Instruktionen besitzt, wäre im Rückblick die erfolgversprechendere Wahl gewesen.

Trotz der Probleme stellt der *PicoRV32-imt* in seiner jetzigen Form eine gute Basis für zukünftige Arbeiten dar. Neben der Behebung der genannten Probleme, ist auch neue Funktionalität denkbar. Die folgende Übersicht stellt eine kleine Auswahl von möglichen zukünftigen Anknüpfungspunkten zusammen.

Ausführung auf realer Hardware Um den Prozessor tatsächlich produktiv nutzen zu können, muss der Grund für das Fehlschlagen der Ausführung auf realer Hardware ermittelt werden. Dies ist der wichtigste Punkt und sollte der erste Anknüpfungspunkt sein, damit auf einer guten Basis weitergearbeitet werden kann. Ein Anfang wäre dafür Register zu untersuchen, die in der Simulation ungültige Werte enthalten, deren Ablauf aber nicht behindern. Hier könnten Unterschiede zwischen der erfolgreichen Simulation und der erfolglosen Synthese aufgedeckt werden.

Stärkere Aufspaltung der Ausführungsphasen Die *fetch*-Phase des Prozessors übernimmt mehrere Funktionen gleichzeitig. Hier könnte etwa das Dekodieren der Befehle in eine eigene Phase ausgelagert werden, um eine höhere Überlappung der Threads zu erreichen. Hier ist ein bessere Auslastung der Ausführungsphasen zu erwarten.

Reimplementierung entfernter Funktionen Vorstellbar ist zum Beispiel, RV32C und RV32E wieder zu unterstützen. Besonders interessant ist daneben die Reimplementierung der IRQ-Instruktionen, da hier eine vielseitige Anwendbarkeit zu erwarten ist. Außerdem ist die Wiedereinführung der Möglichkeiten zur Streckung einiger Ausführungsphasen vielversprechend, da hierdurch eine flexiblere Hardwareauslastung möglich wäre und unter Umständen auch der Maximaltakt steigt. Besonders interessant wäre es hierbei, die Kombination verschieden vieler Threads mit den unterschiedlichen Konfigurationen der Phasenlänge zu untersuchen, um die höchste Leistung, den niedrigsten zusätzlichen Aufwand oder das effizienteste Verhältnis der beiden Faktoren zu ermitteln.

Untersuchung des anormalen Ressourcenverbrauchs bei vier Threads Im Kapitel *Auswertung* wurde deutlich, dass die Konfiguration mit vier

Threads zu einer ungewöhnlich hohen Ressourcennutzung führt. Die Untersuchung der Gründe hierfür steht noch aus. Denkbar wäre, dass bei mehr als vier Threads Hardwaremodule platziert werden können, die mehr Signale pro Element verarbeiten. Hilfreich wäre es auch, die Ergebnisse anderer Synthesewerkzeuge zu ermitteln und zu verlgeichen.

Allgemeine Optimierung Dem Prozessor fehlt die abschließende Optimierung. Es gibt noch Register oder andere Elemente, die keinen Zweck mehr erfüllen, obwohl schon viel nicht mehr Benötigtes entfernt wurde. Auch Speicherzugriffe bieten noch Potenzial zur Optimierung.

Erweiterung des Threadmanagements Aktuell wird streng ein Thread nach dem anderen in die *fetch*-Phase übergeben. Hier wäre denkbar, beschäftigte Threads zu überspringen und bereiten Threads den Vorzug zu geben. Dadurch könnten Speicherlatenzen noch besser überbrückt werden. Hierbei würde also das aktuell statische Threadmanagement dynamisiert. Zudem wäre es denkbar, die Ausführung einer ALU-Operation gleichzeitig mit der Ausführung einer Speicheroperation zu erlauben. Durch diese Maßnahmen würde die Leistung steigen, die Threadsteuerung jedoch an Vorhersagbarkeit verlieren. Es würde sich dann nicht mehr um IMT handeln, sondern um einen dynamischeren Ansatz.

In der *Einleitung* wurde offene Hardware angesprochen. Daran soll sich auch diese Arbeit orientieren, weswegen im Verlauf drei Quelltextrepositorys veröffentlicht wurden, in denen die Ergebnisse für die Allgemeinheit unter den Bedingungen der ISC-Lizenz zur Verfügung gestellt werden. Es handelt sich dabei, neben dem Hauptprojekt (`https://github.com/tmahlburg/picorv32-imt`), um die Portierung des *PicoSoC* auf den *Basys 3* (`https://github.com/tmahlburg/picosoc-basys3`) und den *PicoRAMSoC* (`https://github.com/tmahlburg/picoramsoc`). Damit soll ein Beitrag zur Unterstützung der Entwicklung zukünftiger offener Prozessoren und Hardware im Allgemeinen geleistet werden. Diese Stelle soll auch genutzt werden, um *Claire Wolf* für die Veröffentlichung und Entwicklung des *PicoRV32* unter einer offenen Lizenz zu danken, die diese Erweiterung überhaupt erst ermöglicht hat.

Wie wirkt sich die Implementierung von Interleaved Multithreading auf ein bestehendes, offenes RISC-V-Prozessordesign in Bezug auf Leistung und Ressourcenverbrauch aus? Eine abschließende Antwort darauf kann diese Arbeit nicht geben. Die präsentierten Messergebnisse selbst bleiben hinter den Erwartungen zurück, die aufgeführten Anknüpfungspunkte deuten aber darauf hin, dass es sich hierbei nicht um endgültige Ergebnisse handeln muss. Vielmehr zeigen sie, dass in der Flexibilität des Ansatzes ein großes Potenzial liegt.

Glossar

Build-Prozess Der gesamte Prozess des Zusammenbauens einer Software. Hier wird damit außerdem der Prozess der Erstellung eines Bitstreams aus dem HDL-Quelltext zur Übertragung und Ausführung auf einem FPGA so bezeichnet. Dieser Fall liegt auf gewisse Weise zwischen Hardware und Software.

In-Order-Ausführung Ausführung eines Programms in der vom Programmierer oder Compiler vorgesehenen Reihenfolge.

Latenz Latenz bezeichnet hier den Zeitabstand zwischen dem Anfang und dem Ende einer Operation. Zum Beispiel die Zeit, die vom Laden einer Instruktion bis zum Schreiben der Ergebnisse in ein Register vergeht.

Out-of-Order-Ausführung Auführung eines Programms abweichend von der gegebenen Reihenfolge. Dies kann bspw. genutzt werden, um in Wartezeiten eines Prozessors trotzdem davon unabhängige Instruktionen ausführen zu können, obwohl diese eigentlich noch nicht an der Reihe sind.

Repository Ein verwaltetes Archiv, hier für Software bzw. Quelltext. Dieses kann meist über das Web aufgerufen werden.

Scheduling Die Zuteilung von Ressourcen nach einem bestimmten Plan.

Thread Unabhängige Ausführungszusammenhänge in einem Prozessor. Diese teilen sich denselben Speicher, sind aber sonst von einander vollständig isoliert und können nur über den Speicher kommunizieren.

Toolchain Sammlung von Programmen, die in ihrer Gesamtheit den Build-Prozess durchführen.

Abkürzungsverzeichnis

ALU Arithmetic Logic Unit

BMT Blocked Multithreading

CISC Complex Instruction Set Computer

CPI Cycles per Instruction

CSR Control and Status Register

FPGA Field Programmable Gate Array

HDL Hardware Description Language

IMT Interleaved Multithreading

ISA Instruction Set Architecture

LUT Look-up Table

OoO Out-of-Order

RISC Reduced Instruction Set Computer

SMT Simultaneous Multithreading

SoC System-on-a-Chip

SPI Serial Peripheral Interface

SRAM Static Random Access Memory

VLIW Very Long Instruction Word

UART Universal Asynchronous Reciever Transmitter

Anhang

Neben diesen Anhängen ist dieser Arbeit noch eine CD als digitaler Anhang beigelegt. Die dort zu findenden Dateien entsprechen denen der im Kapitel *Zusammenfassung und Ausblick* genannten Quelltextrepositorys, sowie die Ursprungstabelle der Benchmarkergebnisse.

Anzahl Module	PicoRV32	2 Thr.	3 Thr.	4 Thr.	5 Thr.	6 Thr.
Slice LUTs	982	1.506	1.553	1.610	1.586	1.644
LUT as Logic	934	1.337	1.384	1.441	1.417	1.475
LUT as Memory	48	169	169	169	169	169
LUT as Distr. RAM	48	169	169	169	169	169
LUT as Shift Reg.	0	0	0	0	0	0
Slice Reg.	576	647	615	647	652	652
Reg. as Flip Flop	576	581	549	581	586	586
Reg. as Latch	0	66	66	66	66	66
F7 Muxes	1	0	0	0	30	0
F8 Muxes	0	0	0	0	0	0
Summe	1.559	2.153	2.168	2.257	2.268	2.296
Abs. Änderung	0	594	609	698	709	737
Rel. Änderung	0%	38,10%	39,06%	44,77%	45,48%	47,27%
Veränd. zu 2 Thr.	-	0%	0,70%	4,83%	5,34%	6,64%
Veränd. zu -1 Thr.	-	-	0,70%	4,11%	0,49%	1,23%
Durchschnitt	-	-	1,6325%			
∅ ohne 4 Thr.	-	-	0,81%			

Tabelle 7.1: Ergebnisse des Benchmarks zur Verwendung von Ressourcen

	PicoRV32	2 Thr.	3 Thr.	4 Thr.	5 Thr.	6 Thr.
Startzeit (ns)	6.445	6.445	6.445	6.445	6.445	6445
Endzeit (ns)	25.487.645	21.104.145	19.095.445	20.327.545	19.142.745	19.210.945
Dauer (ns)	25.481.200	21.097.700	19.089.000	20.321.100	19.136.300	19.204.500
Taktperiode (ns)	100	100	100	100	100	100
Dauer (Takte)	254.812	210.977	190.890	203.211	191.363	192.045
Rel. Änderung	100%	82,80%	74,91%	79,75%	75,10%	75,37%

Tabelle 7.2: Ergebnisse des Benchmarks zur Leistung im besten Fall

	PicoRV32	2 Thr.	3 Thr.	4 Thr.	5 Thr.	6 Thr.
Startzeit (ns)	6.445	6.445	6.445	6.445	6.445	6445
Endzeit (ns)	25.487.645	35.788.745	47.725.645	63.089.945	78.848.545	94.180.445
Dauer (ns)	25.481.200	35.782.300	47.719.200	63.083.500	78.842.100	94.174.000
Taktperiode (ns)	100	100	100	100	100	100
Dauer (Takte)	254.812	357.823	477.192	630.835	788.421	941.740
Rel. Änderung	100%	140,43%	187,27%	247,57%	309,41%	369,58%
Steig. zu -1 Thr.	-	40,43%	33,36%	32,20%	24,98%	19,45%
Durchschnitt	-	30,08%				

Tabelle 7.3: Ergebnisse des Benchmarks zur Leistung im schlechtesten Fall

Literaturverzeichnis

[1] *bwitherspoon/rv32cpu*. https://github.com/bwitherspoon/rv32cpu. Aufgerufen am 05.09.2020. 14

[2] *chipsalliance/Cores-SweRV*. https://github.com/chipsalliance/Cores-SweRV. Aufgerufen am 05.09.2020. 14

[3] *darklife/darkriscv*. https://github.com/darklife/darkriscv. Aufgerufen am 05.09.2020. 14

[4] *klessydra/F03x*. https://github.com/klessydra/F03x. Aufgerufen am 04.09.2020. 12

[5] *klessydra/T02x*. https://github.com/klessydra/T02x/tree/master/klessydra-t0-2th. Aufgerufen am 04.09.2020. 12

[6] *lowRISC/ibex*. https://github.com/lowRISC/ibex. Aufgerufen am 05.09.2020. 14

[7] *onchipuis/mriscvcore*. https://github.com/onchipuis/mriscvcore. Aufgerufen am 05.09.2020. 14

[8] *openhwgroup/cv32e40p*. https://github.com/openhwgroup/cv32e40p. Aufgerufen am 05.09.2020. 14

[9] *RISC-V Exchange: Cores and SoCs*. https://riscv.org/exchange/cores-socs/. Aufgerufen am 03.09.2020. 12, 13

[10] *risclite/SuperScalar-RISCV-CPU*. https://github.com/risclite/SuperScalar-RISCV-CPU. Aufgerufen am 05.09.2020. 14

[11] *rsd-devel/rsd*. https://github.com/rsd-devel/rsd. Aufgerufen am 05.09.2020. 14

[12] *sfu-rcl/Taiga*. https://gitlab.com/sfu-rcl/Taiga. Aufgerufen am 05.09.2020. 14

[13] *syntacore/scr1*. https://github.com/syntacore/scr1. Aufgerufen am 05.09.2020. 14

[14] *ucam-comparch/clarvi*. `https://github.com/ucam-comparch/clarvi`. Aufgerufen am 05.09.2020. 14

[15] *ultraembedded/biriscv*. `https://github.com/ultraembedded/biriscv`. Aufgerufen am 05.09.2020. 14

[16] *ultraembedded/riscv*. `https://github.com/ultraembedded/riscv`. Aufgerufen am 05.09.2020. 14

[17] Cheikh, A., G. Cerutti, A. Mastrandrea, F. Menichelli und M. Olivieri: *The microarchitecture of a multi-threaded RISC-V compliant processing core family for IoT end-nodes*. CoRR, abs/1712.04902, 2017. `http://arxiv.org/abs/1712.04902`. 12

[18] Cheikh, A., S. Sordillo, A. Mastrandrea, F. Menichelli, G. Scotti und M. Olivieri: *Klessydra-T: Designing Vector Coprocessors for Multi-Threaded Edge-Computing Cores*, 2020. 12

[19] Chen, C., X. Xiang, C. Liu, Y. Shang, R. Guo, D. Liu, Y. Lu, Z. Hao, J. Luo, Z. Chen, C. Li, Y. Pu, J. Meng, X. Yan, Y. Xie und X. Qi: *Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-bit High Performance RISC-V Processor with Vector Extension : Industrial Product*. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, S. 52–64, 2020. 12

[20] Cormen, T. H.: *Introduction to Algorithms, 3rd Edition* -. MIT Press, Cambridge, 2009, ISBN 978-0-262-03384-8. 38

[21] Hennessy, J. und D. Petterson: *Computer architecture : a quantitative approach*. Morgan Kaufmann, Waltham, MA, 2012, ISBN 9780123838728. 1, 3, 4, 5, 7, 8

[22] Hoffmann, D.: *Grundlagen der Technischen Informatik*. Carl Hanser, München, 2016, ISBN 9783446448674. 4

[23] Hoover, G., F. Brewer und T. Sherwood: *A case study of multi-threading in the embedded space*. S. 357–367, Jan. 2006. 11

[24] Kamaraju, M., B. Sk und C. J. Prakash: *Development of Soft-Core Processor System on FPGA*. 2014. 10

[25] Machanick, P.: *RISC Revisited: the RISC-V approach*. 2019. 7

[26] Märtin, C.: *Rechnerarchitekturen : CPUs, Systeme, Software-Schnittstellen ; mit 10 Tabellen, 42 Beispielen*. Fachbuchverl. im Carl-Hanser-Verl, Leipzig, 2001, ISBN 9783446214750. 3

[27] Olivieri, M., A. Cheikh, G. Cerutti, A. Mastrandrea und F. Menichelli: *Investigation on the Optimal Pipeline Organization in RISC-V Multi-threaded Soft Processor Cores*. S. 45–48, Sep. 2017. 10

[28] Patterson, D. und A. Waterman: *The RISC-V reader an open architecture atlas*. Strawberry Canyon LLC, San Francisco, CA, 2017, ISBN 9780999249116. 1, 7, 8, 9, 37

[29] Plavec, F., B. Fort, Z. G. Vranesic und S. D. Brown: *Experiences with soft-core processor design*. In: *19th IEEE International Parallel and Distributed Processing Symposium*, S. 4 pp.–, 2005. 10

[30] Schrape, J. F.: *Open Source Softwareprojekte zwischen Passion und Kalkül*, Bd. 2015-02 d. Reihe *Research contributions to organizational sociology and innovation studies / Stuttgarter Beiträge zur Organisations- und Innovationssoziologie : SOI discussion paper*. Universität Stuttgart, Fak. 10 Wirtschafts- und Sozialwissenschaften, Institut für Sozialwissenschaften Abt. VI Organisations- und Innovationssoziologie, Stuttgart, 2015. 1

[31] Sheldon, D., R. Kumar, F. Vahid, D. Tullsen und R. Lysecky: *Conjoining Soft-Core FPGA Processors*. In: *Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '06, S. 694–701, New York, NY, USA, 2006. Association for Computing Machinery, ISBN 1595933891. https://doi.org/10.1145/1233501.1233645. 10

[32] SiFive Inc., CS Division, EECS Department, University of California, Berkeley: *The RISC-V Instruction Set Manual - Volume I: Unprivileged ISA*, 2019. 9, 11, 12, 20

[33] SiFive Inc., CS Division, EECS Department, University of California, Berkeley: *The RISC-V Instruction Set Manual - Volume II: Privileged Architecture*, 2019. 33

[34] Thornton, J. E.: *Parallel Operation in the Control Data 6600*. In: *Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems*, AFIPS '64 (Fall, part II), S. 33–40, New York, NY, USA, 1964. Association for Computing Machinery, ISBN 9781450378888. https://doi.org/10.1145/1464039.1464045. 11

[35] Ungerer, T., B. Robič und J. Šilc: *A survey of processors with explicit multithreading*. ACM Computing Surveys (CSUR), 35(1):29–63, 2003. 5, 11

[36] Wolf, C.: *cliffordwolf/picorv32*. https://github.com/cliffordwolf/picorv32. Aufgerufen am 03.09.2020. 12, 14

[37] Wolf, C.: *PicoRV32 - A Size-Optimized RISC-V CPU*. https://github.com/cliffordwolf/picorv32/blob/master/README.md. Aufge-

rufen am 10.09.2020. 17, 18

[38] Wolf, C.: *picorv32/scripts/vivado*. `https://github.com/cliffordwolf/picorv32/tree/master/scripts/vivado`. Aufgerufen am 15.09.2020. 39

Erklärung

„Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann“.

Ort Datum Unterschrift