HAW HAMBURG

Bachelorarbeit

Lenard Heß

Leistungsvergleich von RISC-V und ARM Prozessoren in Motor Control Anwendungen

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Computer Science and Engineering
Department Information and Electrical Engineering

Lenard Heß

Leistungsvergleich von RISC-V und ARM Prozessoren in Motor Control Anwendungen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Elektro- und Informationstechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Karl-Ragmar Riemschneider
Zweitgutachter: Prof. Dr.-Ing. Lutz Leutelt

Eingereicht am: 05. April 2019

Lenard Heß

Thema der Arbeit

Leistungsvergleich von RISC-V und ARM Prozessoren in Motor Control Anwendungen

Stichworte

ARM, RISC-V, Motorsteuerung, Leistungsvergleich

Kurzzusammenfassung

In dieser Arbeit wird eine auf RISC-V basierende eingebettete Applikation für die Motorsteuerung entwickelt. Zusätzlich werden Designrichtlinien für die Schnittstelle zwischen Hard- und Software emtwocleöt und in Kooperation mit Hardwareentwicklern in der eingebetteten Applikation umgesetzt. In einem Vergleich wird das entwickelte System mit einem existerenden System, welches auf einem ARM-Prozessor basiert, verglichen und Defizite identifiziert.

Lenard Heß

Title of Thesis

Performance comparison of RISC-V and ARM processors in Motor control applications

Keywords

ARM, RISC-V, Motor control, Benchmark

Abstract

In this thesis, an embedded application based on a RISC-V Processor specialised for motor control is developed. For the interfacing of Hard- and Software multiple design guidelines are formulated and subsequently implemented in cooperation with hardware developers. A comparison between the developed embedded application and a different application based on an ARM processor is made in order to identify deficits of the new application.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Stand der Technik . . . 1
1.2 Motivation und Zielsetzung . . . 2
1.3 Strukturierung der Arbeit . . . 3

2 Theoretische Grundlagen **4**
2.1 Struktur von Mikrocontrollern . . . 4
2.1.1 Instruction Set Architecture . . . 4
2.1.2 Mikroarchitektur von Prozessoren . . . 5
2.2 Anwendung von Prozessoren als eingebettete Mikrocontroller . . . 7
2.2.1 Anwendungskonzept einer eingebetteten Applikation . . . 7
2.2.2 Prinzip der Ablaufsteuerung durch Interrupts . . . 7
2.3 Aufbau der verwendeten Architekturen . . . 10
2.3.1 ARM . . . 10
2.3.2 RISC-V . . . 11
2.4 Bewertung der Leistungsfähigkeit von eingebetteten Mikrocontrollern . . . 13
2.4.1 Leistungsfähigkeit des Prozessors . . . 13
2.4.2 Latenz . . . 13

3 Implementation eines eingebetteten Systems **15**
3.1 Aktueller Stand des Projektes . . . 15
3.1.1 Vorliegende Hardware . . . 15
3.1.2 Existierende Softwarebeispiele . . . 16
3.1.3 Verfügbare Toolchain . . . 16
3.2 Anforderungen für das Gesamtprojekt . . . 17
3.2.1 Geplante Kommunikationsschnittstellen . . . 17
3.2.2 Spezialisierte Hardware für die Motorsteuerung . . . 18
3.2.3 Allgemeine Prozessorkomponenten . . . 18
3.2.4 Zusammenfassung der Anforderungen in ein Designkonzept . . . 19

3.3 Designüberlegungen zu Beginn des Projektes . . . 19
3.3.1 Designrichtlinien für die Software . . . 20
3.3.2 Designrichtlinien für die Hardware . . . 22
3.4 Realisierung der Implementation . . . 22
3.4.1 Strukturierung der Software . . . 22
3.4.2 Strukturierung der Hardware . . . 24
3.4.3 Implementation der Kommunikationsschnittstellen . . . 26
3.4.4 Implementation von Interrupts und Exceptions . . . 30
3.4.5 Implementation allgemeiner Prozessorkomponenten . . . 35
3.5 Funktionsverifikation . . . 37
3.5.1 RISC-V Prozessor . . . 37
3.5.2 Kommunikationsschnittstellen . . . 39

4 Vorbereitung des Prozessorvergleichs 41
4.1 Definition von Messgrößen für den Prozessorvergleich . . . 41
4.2 Entwicklung von Messmechanismen für benötigte Taktzyklen . . . 42
4.2.1 Messmechanismen für RISC-V . . . 42
4.2.2 Messmechanismen für ARM . . . 45
4.2.3 Theoretischer Vergleich der Prozessorimplementationen . . . 46
4.3 Definition der zu vergleichenden Parameter . . . 47
4.3.1 Ablaufsteuerung . . . 47
4.3.2 Bitlogik . . . 47
4.3.3 Regelungsalgorithmus . . . 47
4.3.4 Kommunikation mit anderem System . . . 48
4.4 Auswahl der Vergleichsalgorithmen . . . 48
4.4.1 Zyklische Redundanzprüfung als Softwareimplementation . . . 48
4.4.2 Field Oriented Control . . . 48
4.4.3 Algorithmen nach Clark und Park . . . 49
4.4.4 Trinamic Motion Control Language . . . 49

5 Durchführung des Prozessorvergleichs 50
5.1 Benchmark: Zyklische Redundanzprüfung . . . 50
5.1.1 Der Algorithmus . . . 50
5.1.2 Die Implementation . . . 53
5.1.3 Betrachtete Bewertungskriterien . . . 54
5.1.4 Messergebnisse . . . 55

5.1.5 Auswertung der Messergebnisse . . . 56
5.2 Field Oriented Control . . . 64
5.2.1 Der Algorithmus . . . 64
5.2.2 Die Implementation . . . 65
5.2.3 Betrachtete Bewertungskriterien . . . 66
5.2.4 Messergebnisse . . . 66
5.2.5 Auswertung . . . 67
5.3 Algorithmen nach Clark und Park . . . 69
5.3.1 Die Algorithmen . . . 69
5.3.2 Die Implementation der Algorithmen . . . 70
5.3.3 Messergebnisse . . . 70
5.3.4 Auswertung der Messergebnisse . . . 70
5.4 Trinamic Motion Control Language . . . 73
5.4.1 Aufbau des Protokolls . . . 73
5.4.2 Vergleich der Implementationen . . . 73
5.4.3 Messaufbau . . . 74
5.4.4 Durchführung der Messung . . . 74
5.4.5 Auswertung der Messergebnisse . . . 76
5.5 Zusammenfassung der Messergebnisse . . . 78
5.5.1 Leistungsfähigkeit des Prozessors . . . 78
5.5.2 Leistungsfähigkeit des Gesamtsystems . . . 80

6 Fazit **81**
6.1 Umsetzung der Aufgabenstellung . . . 81
6.2 Erkenntnisse über RISC-V gegenüber ARM . . . 81
6.3 Weitere Schritte für das Projekt ROCINANTE . . . 82

Abbildungsverzeichnis **83**

Tabellenverzeichnis **84**

Codeverzeichnis **85**

Literaturverzeichnis **86**

Selbstständigkeitserklärung **89**

1 Einleitung

Mikrocontroller sind in der Industrie weit verbreitet. Man findet sie als Embedded Systeme in den meisten modernen digitalen Systemen. Die kleinen Prozessoren erlauben eine flexible Anpassung auf spezielle Aufgaben aller Art. Software für einen Mikrocontroller erlaubt eine deutlich schnellere Produktentwicklung als das Design von Logik in Hardware. Heutzutage basiert ein Großteil von verwendeten Embedded Mikrocontrollern auf der ARM-Prozessorarchitektur [12].

In dieser Arbeit soll ein auf Motorsteuerung spezialisierter Mikrocontroller - unter dem Projektnamen *ROCINANTE* - auf Basis der Prozessorarchitektur RISC-V bearbeitet werden. Dies umfasst die Implementation von Software und das Design von Hardwareschnittstellen - das Entwickeln der Hardware ist nicht enthalten. Das entstehende Produkt soll mit der Landungsbrücke [18], einem auf ARM basierendem, existerenden Projekt von Trinamic Motion Control verglichen werden. Ziel des Vergleiches ist es, die Unterschiede zwischen Projekten auf ARM- bzw. RISC-V-Basis zu identifizieren, um eine Bewertung der Markttauglichkeit zu ermöglichen.

1.1 Stand der Technik

Trinamic Motion Control entwickelt Hardware für die Steuerung von Elektromotoren. Mit zahlreichen Features für die Motorsteuerung kann ein digitaler Mikrocontroller das analoge System - den Motor - leicht bedienen. Auch die Ansteuerung durch den Mikrocontroller findet bei Trinamic statt. Die sogenannten Evaluation Kits, eine Reihe an Referenzdesigns für die Verwendung der Motor- und Motion Controllers, bieten ein fertiges einsatzbereites System für die Motorsteuerung [19]. Sie sind modular aufgebaut und bestehen aus einem Mikrocontrollerboard, welches ein Motion-Controller-Board oder ein Driver-Board ansteuert. Als Mikrocontrollerboard sind die *Startrampe* [21] und die *Landungsbrücke* [18] verfügbar - zwei auf ARM Cortex-M basierende Mikrocontroller.

ARM Ltd. dominiert mit 46.2% Marktanteil im Jahr 2017 den Markt für Design-IP [12]. Ihre Mikroprozessoren basieren auf der ARM-Architektur und sind in Smartphones, Tablets sowie eingebetteten Systemen verwendet.

RISC-V ist eine frei verfügbare öffentliche Prozessorarchitektur mit dem Ziel, ein freies, erweiterbares Ökosystem an Hard- und Software für kommerzielle und akademische Zwecke zu ermöglichen [13]. Die *RISC-V Foundation* ist für die Weiterentwicklung der RISC-V Architektur zuständig.

1.2 Motivation und Zielsetzung

Der Einstieg von RISC-V in den Mikrocontrollermarkt bietet Firmen neue Perspektiven. Begonnen in 2010 und offen entwickelt, erlaubt RISC-V die Vermeidung von Komplikationen durch historische Designentscheidungen (*Tech Debt*), die in älteren Architekturen zu finden sind. Durch ein modulares Design erlaubt die RISC-V-Architektur eine Anpassung an spezifische Nutzungszwecke in einem Produkt. Jedoch ist das relativ geringe Alter von RISC-V auch eine Schwäche - ARM ist mit einem großen Ökosystem an existierenden Produkten, Entwicklertools sowie Entwicklern im Markt etabliert. Durch den Wechsel zu einer Open-Source ISA können jedoch die Lizenzgebühren für die Verwendung von ARM eingespart werden [10]. Ein Einstieg in RISC-V bedeutet für Firmen somit die Einsparung von Lizenzgebühren zu dem Preis, einen Rückschlag in existierender Entwicklungsinfrastruktur zu erfahren. Allerdings bietet RISC-V langfristig das Potenzial, in der Zukunft ARM einzuholen oder sogar zu überholen.

Durch die neuen Möglichkeiten von RISC-V will Trinamic seine bestehenden Produkte für die Motorsteuerung mit den benötigten Mikrocontrollern für die Ansteuerung kombinieren. Auf Basis der miteinander vereinten Systeme können Prototypen sowie fertige Applikationen einfacher und schneller entwickelt werden. Die engere Kopplung von Mikrocontroller und Hardware für Motorsteuerung eliminiert die Notwendigkeit für die Verbindung zweier getrennter Systeme und der daraus resultierenden Nachteile. Der Wechsel auf die öffentliche RISC-V-Architektur kann dabei ein potenzielles Hindernis für langlebige Produkte entfernen - die Bindung an ARM Ltd. durch die benötigten Lizenzen.

1.3 Strukturierung der Arbeit

Die Arbeit ist unterteilt in zwei separate Aufgaben:

In Kapitel 3 soll das auf RISC-V basierte Projekt bearbeitet werden. Dies beinhaltet die Entwicklung von Software für den Prozessor, das Design von Hardwareschnittstellen sowie die Funktionsverifikation der Hardware. Die Entwicklung der Hardware ist nicht Teil dieser Arbeit und wird von anderen Entwicklern durchgeführt.

Für den Vergleich des entwickelten Projektes mit der Landungsbrücke sollen in Kapitel 4 die Unterschiede der beiden Produkte identifiziert, passende Messkriterien definiert und Messverfahren implementiert werden. In Kapitel 5 sollen die Messungen durchgeführt und analysiert werden.

2 Theoretische Grundlagen

Dieses Kapitel umfasst Grundlagen für den Vergleich von verschiedenen Mikrocontrollern. Es werden die einzelnen Bestandteile sowie Anwendungsprinzipien für eingebettete Applikationen dargestellt sowie das ein Grundprinzip für das Vergleichen verschiedener Prozessoren erläutert.

2.1 Struktur von Mikrocontrollern

Im Kern einer eingebettete Applikationen steckt der Prozessor - die zentrale Recheneinheit, die ein Programm abarbeitet. In diesem Abschnitt soll auf den typischen Aufbau eines solchen Prozessors eingegangen werden.

2.1.1 Instruction Set Architecture - die Sprache eines Prozessors

Eine Instruction Set Architecture (ISA) definiert die Schnittstelle zwischen Software und Hardware eines Prozessors. Die ISA basiert auf dem grundlegenden Prozessormodell, beispielsweise eine *Load-Store-Architektur*, die Speicheroperationen von Rechenoperationen trennt, oder eine *Register-Memory-Architektur*, wo diese Operationen kombiniert sein können. Des weiteren definiert die ISA den verwendbaren Befehlssatz den die Software verwenden kann. Teilweise sind noch weitere Eigenschaften definiert, wie die Strukturierung der Speicheraddressen, dem *Addressraum*, das genaue Verhalten des Speichers (*Memory model*) oder die Interaktion mit der Außenwelt über verschiedene I/O-Mechanismen.

Für die Definition von Befehlen gibt es mehrere Designansätze. Die in dieser Arbeit verwendeten Prozessoren sind sogenannte *Reduced Instruction Set Computer* (RISC). Hierbei zielen die Befehlssätze darauf, mit einer geringen Anzahl an Befehlen auszukommen. Komplexe Operationen werden in Software durch mehrere simple Befehlen gebildet.

Als Resultat ist eine Implementation der Hardware leichter umzusetzen und zu optimieren. Im Gegensatz zu diesem Designansatz stehen *Complex Instruction Set Computer* (CISC) mit vielen Befehlen für komplexere Aufgaben. Solche Prozessoren benötigen in der Hardwareimplementation mehr Logik für die Implementation der komplexen Befehle [16].

2.1.2 Mikroarchitektur von Prozessoren

Als Mikroarchitektur wird eine Implementation einer ISA in Hardware bezeichnet. Designentscheidungen dieser Implementationen sind für die Leistungsfähigkeit eines Prozessors maßgeblich entscheidend.

Pipeline

Die Abarbeitung von Prozessorinstruktionen erfolgt in mehreren Schritten. Fundamental sind drei Schritte notwenig:

- *Fetch*: Das Laden einer Instruktion aus dem Speicher
- *Decode*: Die Übersetzung der Instruktion in Steuersignale für die Prozessorhardware
- *Execute*: Die Ausführung eines Befehls.

Ein Prozessor, der jeden Schritt separat bearbeitet benötigt somit mindestens drei Takte pro Instruktion (*Sub-Skalarität*). Für einen effizienten Programmablauf werden diese drei Schritte in Form einer *Pipeline* durchgeführt. Hierbei können mehrere Schritte parallel durchgeführt werden - während eine Instruktion in der *Execute*-Phase ist, kann die nächste Instruktion in der *Decode*-Phase bearbeitet werden. Eine simple Pipeline hat für jede der drei fundamentalen Schritte eine Stufe und besteht somit aus 3 Stufen. Solch eine Pipeline erlaubt die Ausführung einer Instruktion pro Takt (*Skalarität*). Um die Leistungsstärke eines Prozessors weiter zu erhöhen kann eine Pipeline in mehrere kleinere Stufen unterteilt werden, um den kritischen Pfad der Digitallogik zu verkürzen und somit eine höhere Taktgeschwindigkeit zu erlauben. Des weiteren kann eine komplexere Strukturierung der Pipeline mit mehr Stufen genutzt werden, um mehrere Instruktionen parallel auszuführen (*Super-Skalarität*) [15].

Durch die Verwendung einer Pipeline entstehen neue Probleme bei der Programmausführung. Wenn die Ausführung (*execute*-Phase) eines Befehls in einem Takt nicht fertiggestellt werden kann, so müssen die vorherigen Pipelinestufen pausieren, bis der Befehl durchgeführt wurde. Ursache für solche Verzögerungen können komplexere Befehle wie eine Division oder Befehle, die mit anderen Teilen des Systemes interagieren wie Speicherzugriffe sein. Dieses Problem wird als *Pipeline Stall* bezeichnet - die Pipeline muss Teilschritte stoppen. In komplexeren Pipeline-Designs können diese Pausen durch die Verarbeitung anderer Befehle des Programmes gefüllt werden. Hierbei erkennt die Hardware voneinander unabhängige Befehle und kann dynamisch die Ausführungsreihenfolge umsortieren. Dies wird als *Out-of-Order-Execution* bezeichnet, ist allerdings in den in dieser Arbeit behandelten Prozessoren nicht implementiert.

Ein weiteres Problem durch Pipelines sind Datenkonflikte. Diese entstehen, wenn eine Instruktion das Ergebnis der vorherigen Instruktion als Operand verwendet (*Pipeline Hazard*). Um den korrekten Wert zu verwenden muss je nach Design der Pipeline entweder gewartet werden (*Pipeline Stall*), bis der korrekte Wert zur Verfügung steht oder das Ergebnis der vorherigen Instruktion muss durch einen separaten Datenpfad vorzeitig für die nächste Instruktion zur Verfügung gestellt werden (*Operand Forwarding*).

Ablaufsteuerung

Während der Ausführung eines Programmes bearbeitet der Prozessor die Instruktionen des Programmes der Reihe nach. Die Addresse der auszuführenden Instruktion ist in dem Programmzähler hinterlegt und zählt während der Programmbearbeitung hoch. Befehle für die Ablaufsteuerung oder externe Ereignisse (Interrupts) sorgen für einen Sprung im Programmablauf. Für solch einen Vorgang muss der Prozessor den Programmzähler auf eine neue Addresse setzen und die Programmausführung von dieser Addresse aus fortführen.

Bei der Verwendung einer Pipeline führt dies zu einem sogenannten Steuerkonflikt - während die Instruktion zur Ablaufsteuerung sich in der finalen Stufe (*Execute*) befindet, sind die vorherigen Stufen mit den darauf folgenden Instruktionen gefüllt. Durch den Sprung im Programmablauf müssen diese unvollständig bearbeiteten Instruktionen verworfen werden und die Arbeit an der neuen Programmaddresse muss begonnen werden. Diese Leerung der Pipeline wird als *Pipeline Flush* bezeichnet. Das erneute Füllen der Pipeline beginnt in der ersten Stufe und resultiert somit in einer Verzögerung bis zur endgültigen

Ausführung der nächsten Instruktion in der *Execute-Phase*. Je länger eine Pipeline ist, desto größer ist diese Verzögerung.

2.2 Anwendung von Prozessoren als eingebettete Mikrocontroller

2.2.1 Anwendungskonzept einer eingebetteten Applikation

Für die Verwendung von Prozessoren als Mikrocontroller werden Peripherien für verschiedene Aufgaben verwendet. Schnittstellen über verschiedene Bussysteme erlauben es, mit anderen Systemen zu kommunizieren, Ein- und Ausgänge für digitale und analoge Signale ermöglichen die Interaktion mit elektronischen Schaltungen.

Die Verwendung von Sensoren und Aktoren für die Interaktion mit der Außenwelt bilden eine einbettende Umgebung, in die ein Mikrocontroller eingebettet werden kann. Das daraus resultierende System - die eingebettete Applikation - erlaubt es, Vorgänge zu überwachen oder zu steuern. Eine eingebettete Applikation kann über Sensorik über Ereignisse informiert werden und darauf reagieren. Bei der Überwachung eines Systemes können Messwerte gespeichert werden oder das Überschreiten von Schwellwerten gemeldet werden. Gemessene Eingangswerte können verarbeitet werden - beispielsweise in einem Regelkreis - und entsprechende Ausgangswerte produziert werden, um Vorgänge in der Außenwelt zu steuern. Eingebettete Applikation sind typischerweise auf eine bestimmte Aufgabe spezialisiert, für die sie spezifisch entwickelt werden.

2.2.2 Prinzip der Ablaufsteuerung durch Interrupts

In einer eingebetteten Applikation treten Ereignisse außerhalb des Prozessors auf, die eine Bearbeitung durch den Prozessor benötigen. Beispielsweise kann ein anderes System über einen Bus Daten an die Applikation senden oder ein Sensor einen Schwellwert überschreiten. Es gibt zwei Möglichkeiten für die Bearbeitung dieser externen Ereignisse. Ein Programm kann den Status solcher Ereignisse abfragen und beim Auftreten bestimmte Funktionen bearbeiten. Dies wird als *Polling* bezeichnet und hat mehrere Nachteile. Für ein Echtzeitsystem ist eine Einhaltung der Zeitlimits essentiell - dies kann mit Polling nur mit erheblichen Komplikationen und Limitationen umgesetzt werden. Die Alternative zu *Polling* ist, das aktuelle Programm zu unterbrechen und das Ereignis zu bearbeiten. Ist

dies abgeschlossen, kann das Programm wieder fortgeführt werden. Dieser Mechanismus wird als *Interrupt* bezeichnet - die Hardware kann laufende Software unterbrechen (*Interrupt Request*, kurz *IRQ*), um ein Ereignis zu bearbeiten (*Interrupt Service Routine*, kurz *ISR*).

Interruptquellen

Ein Interrupt kann von verschiedenen Stellen des Systems aus erzeugt werden:

- Außerhalb des Prozessors kann Hardware einen Interrupt für die Bearbeitung eines externen Ereignisses anfordern. So kann beispielsweise eine Flanke an einem Pin einen Interrupt auslösen.
- Ein Prozessor kann den Interrupt-Mechanismus auslösen, wenn gewisse Ereignisse auftreten, die eine gesondere Behandlung erfordern. Beispielsweise kann das Laden einer ungültigen Instruktion oder der Zugriff auf eine nichtexistente Speicheraddresse zu einer Unterbrechung des Programmablaufes führen. Solche Fälle werden auch als *Exceptions* bezeichnet und die Funktion zur Bearbeitung wird als *Exception Service Routine*, kurz *ESR* bezeichnet.
- Software kann durch eine Instruktion einen Interrupt auslösen. Dieser Mechanismus findet oft in Betriebssystemen Verwendung, wo der Wechsel in eine *Interrupt Service Routine* als Aufruf von Betriebssystemfunktionen genutzt wird.

Implementation von Interrupts

Für die Implementation von Interrupts sind mehrere Schritte nötig:

1. Eine oder mehrere *Interrupt Requests* werden ausgelöst.
2. Aus allen aktiven *Interrupt Requests*, wird ein Interrupt zur Bearbeitung ausgesucht, da der Prozessor nur eine *ISR* auf einmal ausführen kann. Die Auswahl wird über ein Prioritätsverfahren gesteuert. Mögliche Verfahren reichen von sequentieller Bearbeitung entsprechend einer Nummerierung hin zu in Software definierten Prioritätsstufen.
3. Der Kontext des aktuellen Programmes wird gespeichert. Dies beinhaltet die aktuelle Programmaddresse und Werte der Prozessorregister.

4. Die *Interrupt Service Routine* wird aufgerufen. Die Bearbeitung des Ereignisses ist hiermit abgeschlossen

5. Der gespeicherte Kontext wird wiederhergestellt. Dies lässt das ursprünglich laufende Programm fortsetzen.

Alle Schritte können komplett in der Prozessorhardware durchgeführt werden, eine Verlagerung in Software ist teilweise auch möglich. Die Auswahl durch das Prioritätsverfahren kann in Software durchgeführt werden, allerdings muss dieser Schritt dann erst nach dem Speichern des Kontexts geschehen. Das Speichern und Wiederherstellen des Kontexts ist ebenfalls über Software lösbar.

Ein erweitertes Prioritätsverfahren kann die Implementation von verschachtelten Interrupts erlauben. Hierbei kann ein Interrupt höherer Priorität sofort bearbeitet werden, während ein Interrupt niedrigerer Priorität bereits in Bearbeitung ist. Ein solches System erlaubt es, eine Echtzeitgarantie mithilfe eines Interrupts hoher Priorität einzuhalten und trotzdem andere Interrupts niedrigerer Priorität mit langsamen Routinen zu verwenden.

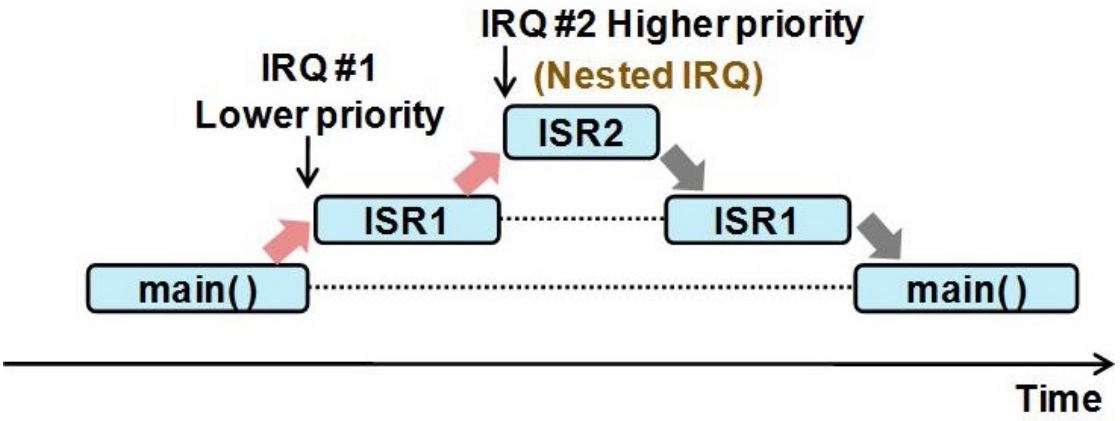

Abbildung 2.1: Verschachtelung von Interrupts [26]

Vor- und Nachteile von Interrupts

Die Verwendung von Interrupts erlaubt es, von der Hauptschleife eines Programmes unabhängige Routinen zu benutzen. Insbesondere bei Aufgaben, die vom Rest des Programmes getrennt sind, erlaubt diese Unabhängigkeit eine simple Implementation. Jedoch ist die Interaktion zwischen Interrupt-Routinen und Hauptprogramm sehr fehleranfällig bei

der Programmierung - eine gute Implementation von Applikationssoftware ist nötig. Typischerweise erfolgt diese Interaktion über zwischen den Routinen geteilten Variablen im Hauptspeicher. Die Modifikation einer Variable durch verschiedene Interrupt-Routinen und Hauptprogramm hat das Potenzial durch sogenannte *Race Conditions* Fehler zu generieren.

2.3 Aufbau der verwendeten Architekturen

2.3.1 ARM

Die ARM-ISA hat sich über drei Jahrzehnte hin entwickelt. Begonnen hat die Architektur im Jahre 1985 mit der ersten Version (*ARMv1*), die neueste Version (*ARMv8*) wurde 2011 herausgebracht. Moderne ARM-Prozessoren sind in drei sogenannte Cortex-Familien und drei weitere spezialisierte Familien unterteilt [8]. Die Cortex-Familien sind unterteilt in Cortex-M für eingebettete Applikationen, Cortex-R für Echtzeitapplikationen und Cortex-A für Applikationen mit hohen Leistungsansprüchen. Als spezialisierte Familien bietet ARM die Neoverse-Familie für Verwendung in Netzwerkinfrastruktur und Cloud-Servern, die SecurCor-Familie für sicherheitskritische Applikationen wie elektronische Zahlungsmethoden und die Machine-Learning-Familie für die Verwendung in Machine-Learning-Applikationen.

ARM-Prozessoren verfügen über 16 Register von denen die letzten 3 Register als Stack Pointer, Link Register und Program Counter definiert sind [1]. Der Stack Pointer enthält die aktuelle Addresse des Stacks, das Link Register enthält die Rücksprungaddresse[1] und der Program Counter enthält die aktuelle Addresse, von der aus Instruktionen geladen werden.

Der originale ARM-Befehlssatz verwendet 32-Bit Instruktionen. Um die Codegröße zu reduzieren, wurde später der sogenannte Thumb-Befehlssatz mit 16-Bit Instruktionen ergänzt. Die Thumb-Instruktionen entsprechen einer Teilmenge des ARM-Befehlssatzes und erlauben so ein Ersparnis in Codegröße auf Kosten der Ausführungsgeschwindigkeit. Der Wechsel zwischen Thumb- und ARM-Instruktionen kann nur nach einem Sprung im

[1] Die Rücksprungaddresse ist eine Programmaddresse, an die nach Abschluss eines Funktionsaufrufes hin zurückgesprungen wird. Beim Aufruf einer Funktion durch die *Branch-and-link*-Instruktion wird das Link Register gesetzt.

Ablauf geschehen. Um einige dieser Nachteile zu kompensieren, wurde 2003 der Thumb-2-Befehlssatz veröffentlicht. Dieser erweitert die alten Thumb-Instruktionen und fügt einige 32-Bit Instruktionen hinzu.

2.3.2 RISC-V

Die RISC-V-ISA setzt sich aus mehreren modularen Teilen zusammen. Jede Implementation nutzt eins Basismodule. Diese umfassen die drei Standard-Integer-Basismodule mit 32 Registern für 32-, 64- oder 128-Bit Prozessoren (bezeichnet als RV32I, RV64I bzw. RV128I) oder das Embedded Basismodul mit nur 16 Registern und einem 32-Bit Prozessor (bezeichnet als RV32E) [14]. Zu dem Basismodul können optional mehrere Erweiterungen ergänzt werden. Mögliche Erweiterungen sind unter anderem Integer Multiplikation (M), atomare Operationen (A), komprimierte Instruktionen (C) oder Gleitkommaarithmetik für Single-Precision (F), Double-Precision (D) oder Quad-Precision(Q). Einige der Module sind noch in der Entwicklung, während die fertiggestellten Erweiterungen standardisiert sind und nicht mehr geändert werden (*frozen*).

Base	**Name**	**Frozen**
32 Bit Integer	RV32I	Y
64 Bit Integer	RV64I	Y
128 Bit Integer	RV128I	N
32 Bit Embedded	RV32E	N
Extension	**Name**	**Frozen**
Integer Multiplication and Division	M	Y
Atomics	A	Y
Single-Precision Floating-Point	F	Y
Double-Precision Floating-Point	D	Y
Quad-Precision Floating-Point	Q	Y
Decimal Floating-Point	L	N
16-bit Compressed Instructions	C	Y
Bit Manipulation	B	N
Dynamic Languages	J	N
Transactional Memory	T	N
Packed-SIMD Extensions	P	N
Vector Extensions	V	N
User-Level Interrupts	N	N

Tabelle 2.1: Liste der RISC-V ISA Erweiterungen [14]

Fast alle Module von RISC-V nutzen 32-Bit Instruktionen. Die einzige Ausnahme bildet die C-Erweiterung, welche komprimierte 16-Bit Befehle implementiert. Jeder komprimierte Befehl ist als 32-Bit Befehl darstellbar. Dies erlaubt eine simple Implementation in Hardware - die Dekodierung der Befehle kann jeden 16-Bit Befehl in einen 32-Bit Befehl umwandeln bevor dieser weiter bearbeitet wird. Instruktionen, die länger als 32 Bit sind, werden von der ISA jedoch unterstützt - Kodierungen für größere Instruktionen bis zu 192 Bit sind definiert und für noch größere Instruktionen ist Raum in der Kodierung reserviert [14].

2.4 Bewertung der Leistungsfähigkeit von eingebetteten Mikrocontrollern

Für die Bewertung der Leistungsfähigkeit von eingebetteten Systemem sind mehrere Teile des Gesamtsystems relevant.

2.4.1 Leistungsfähigkeit des Prozessors

Wenn man die Leistungsfähigkeit verschiedener Prozessoren vergleichen will, so gilt immer die sogenannte „Iron Law“ [11] [25]. Die Zeit, die ein gegebenes Programm benötigt wird hier durch drei Faktoren bestimmt:

$$\frac{1}{Performance} = \frac{Time}{Program} = \frac{Instructions}{Program} \cdot \frac{ClockCycles}{Instruction} \cdot \frac{Time}{ClockCycle}$$

Jeder dieser drei Faktoren setzt sich zusammen aus mehreren beeinflussenden Komponenten des Gesamtsystems:

1. Die Instruktionen pro Programm („Instruction Count“) hängen von der implementieren ISA sowie dem verwendeten Compiler ab. Je besser der Compiler und je komplexer die Instruktionen einer ISA, desto weniger Instruktionen werden für ein gegebenes Programm benötigt.

2. Die Taktzyklen pro Instruktion hängen von der Komplexität einzelner Instruktionen sowie der Mikroarchitektur der vorliegenden Prozessorimplementation ab. Für diesen Term wird oft die Inverse verwendet - Instruktionen pro Takt („Instructions per Cycle“, kurz „IPC“).

3. Die Zeit pro Takt entspricht der Taktfrequenz eines Prozessors. Diese ist limitiert durch eine fundamentale physikalische Grenze - die Ausbreitungsgeschwindigkeit elektrischer Signale. Eine erhöhte Taktrate führt außerdem zu erhöhter Leistungsaufnahme und Wärmebildung.

2.4.2 Latenz

Die Verzögerung zwischen einem externen Ereignis und der Bearbeitung durch den Prozessor ist ein wichtiger Faktor für die schnelle Reaktion eines Systems. Bei Echtzeitsyste-

men ist dies besonders wichtig, um die gesetzten Echtzeitgarantien einzuhalten. Bei der Verwendung von Interrupts zur Bearbeitung von externen Ereignissen entspricht diese Verzögerung der Interrupt-Latenz - der Zeit zwischen dem Ereignis und der Bearbeitung durch einen Interrupt-Handler.

3 Design eines eingebetteten Systemes auf Basis eines RISC-V Mikroprozessors

In diesem Kapitel wird die Arbeit an dem Projekt ROCINANTE beschrieben, einem RISC-V-Mikrocontroller für Motorsteuerungs-Applikationen. Es werden zunächst die bereits existierende Hard- und Software sowie verfügbare Tools des vorliegenden RISC-V-Prozessors analysiert und anschließend das Konzept für die weitere Entwicklung definiert. Dies umfasst die Festlegung der Anforderungen, die das fertige System erfüllen soll und welche Hard- und Softwarekomponenten hierfür benötigt werden.

Bevor die Realisierung des entwickelten Konzeptes beginnt sollen allgemeine Designrichtlinien für Hard- und Software festgelegt werden. Zudem sollen die bereits verfügbaren Software-Beispiele auf ihren Wert zur Weiterverarbeitung geprüft werden.

3.1 Aktueller Stand des Projektes

3.1.1 Vorliegende Hardware

Der RISC-V Prozessor liegt als Soft-Core auf einem FPGA vor. Dieser wurde von einer externen Firma als Soft-IP-Core lizensiert. Auf dem verwendeten FPGA steht als Hauptspeicher nur SRAM zur Verfügung. Somit ist es nicht möglich, ein Programm über den Verlust der Betriebsspannung hinweg zu erhalten. An Peripherie-Modulen für den Prozessor liegen eine UART-Schnittstelle vom Prozessorhersteller, eine extern lizensierte CAN-Schnittstelle sowie eine intern entwickelte USB-Schnittstelle vor. Des weiteren sind acht LEDs des FPGA-Boards als GPIO-Ausgänge verfügbar. Zusätzlich ist der Motion Controller TMC4671 [22] - auch bekannt unter dem Projektnamen Weasel - als Peripherie des Prozessors integriert.

Die existierende Hardware wird fast vollständig für das Projekt ROCINANTE weiter verwendet. Nur die UART-Schnittstelle soll durch eine intern entwickelte UART-Schnittstelle ersetzt werden.

3.1.2 Existierende Softwarebeispiele

Es liegt ein Beispielprojekt vor, welches eine simple Implementation eines SysTicks nutzt, um eine der acht LEDs blinken zu lassen. Dieses Projekt enthält zusätzlich auch Code für das Empfangen und Ausführen einiger TMCL[1]-Befehle über die CAN-Schnittstelle, jedoch funktioniert dieser Code aufgrund von Änderungen in der Hardware nicht mehr. Zusätzlich ist ein weiteres Projekt für die Kommunikation über die USB-Schnittstelle vorhanden, aber dies ist aufgrund von Fehlern in der USB Hardware zu diesem Zeitpunkt nicht verwendbar. Beide Beispielprojekte nutzen eine in Assembler programmierte C-Runtime für den Programmstart sowie die Nutzung von Interrupts.

Zu Beginn der Entwicklung wird das erste Beispielprojekt mit LED-Status und CAN Kommunikation verwendet. Das Beispielprojekt für USB wird zu einem späteren Zeitpunkt nur noch als Referenz für die Integration von USB in das aktiv entwickelte Projekt verwendet. Die C-Runtime wird zunächst verwendet, zu einem späteren Zeitpunkt soll diese aber überarbeitet werden.

3.1.3 Verfügbare Toolchain

Zusammen mit dem RISC-V-Prozessor wurde eine Toolchain mit einem C-Compiler, Simulatoren für den Prozessor sowie ein auf GDB basierter Debugger mitgeliefert. Das Flashen des Prozessors erfolgt über den Debugger, welcher das Programm über eine JTAG-Schnittstelle hochlädt.

Die Simulatoren des Prozessors finden keine Verwendung in der Entwicklung des Projektes. Der C-Compiler wird zu Beginn verwendet und wird im späteren Projektverlauf aufgrund eines Compilerfehlers sowie unerwünschter spezieller Erweiterungen durch den Standard RISC-V-Compiler ersetzt. Der Debugger findet während des gesamten Projektes weitere Verwendung. Eine Verwendung von frei verfügbaren Tools ist zwar bevorzugt, jedoch hier zu aufwändig.

[1] Trinamic Motion Control Language - eine von Trinamic entwickelte Skriptsprache [20]

3.2 Anforderungen für das Gesamtprojekt

Das Ziel des Projektes ROCINANTE ist es, die präzisen Kontrollmöglichkeiten eines Motion Controllers mit der Flexibilität eines Mikrocontrollers zu kombinieren. Durch die Verbindung dieser beiden Komponenten in einem Hardwaremodul soll es möglich sein, eine einsatzbereite Embedded Applikation zur Motorsteuerung mit deutlich reduziertem Entwicklungsaufwand zu liefern.

Der Fokus des Projektes liegt darauf, die Komplexität von Motorsteuerung in Hardware zu verlagern, sodass der Prozessor nur noch die leistungsstarken Hardwareelemente verwalten muss. So ist zum Beispiel eine vollständige FOC-Regelung durch das Weasel-Modul in Hardware verfügbar. Durch diese Designentscheidungen muss der Prozessor zeitkritische Aufgaben nur noch verwalten anstatt diese selbst zu bearbeiten. Dies führt zu einer Entlastung des Prozessors sowie einer Vereinfachung der benötigten Software.

Die Anforderungen an die ROCINANTE sind zum Zeitpunkt der Arbeit schon definiert, sollen hier jedoch zusammenfassend beschrieben werden.

3.2.1 Geplante Kommunikationsschnittstellen

Um eine Vielzahl an Applikationen zu ermöglichen, soll der Mikrocontroller eine große Auswahl an Kommunikationsschnittstellen bieten. Diese können für die Integration verschiedener Sensoren, Aktoren oder anderen Controller verwendet werden. Zur Kommunikation soll daher eine Vielzahl an verschiedenen Schnittstellen in Hardware zur Verfügung stehen:

- USB 2.0
- CAN
- I2C
- SPI
- QSPI
- UART

3.2.2 Spezialisierte Hardware für die Motorsteuerung

Als ein Mikrocontroller für Motorsteuerungsapplikationen, soll das ROCINANTE-Projekt möglichst viele Features in Hardware zur Verfügung stellen:

- Ein Weasel-Modul für die Steuerung von einem DC-, BLDC- oder Schrittmotor ist direkt integriert.
- Die SPI Kommunikationsschnittstelle stellt spezialisierte Betriebsmodi für die Verbindung zu einem Positionsgeber (*Encoder*) mit SPI-Schnittstelle zur Verfügung.
- Ein StepDir Interface[2] erlaubt die direkte Kontrolle eines weiteren Driver-Moduls.
- Die Timer bieten Betriebsmodi zur direkten Zeitmessung von Pulsformen eines ABN Encoders.

Neben den integrierten Möglichkeiten zur Motorsteuerung erlauben die zahlreichen Schnittstellen, weitere Module zur Motorsteuerung in das Gesamtsystem einzubauen.

3.2.3 Allgemeine Prozessorkomponenten

Neben den Schnittstellen und den für Motorsteuerung spezialisieren Komponenten, sollen auch für Mikrocontroller klassische Komponenten verfügbar sein:

- GPIO Pins für die Interaktion mit elektronischen Schaltungen
- General-Purpose-Timer für Echtzeitmessungen, periodische Vorgänge oder Generierung von PWM Signalen
- Eine JTAG-Schnittstelle zum Hochladen und Debuggen eines Programmes
- ADCs für die Verarbeitung von Analogsignalen

[2] *StepDir* erlaubt die Steuerung eines Schrittmotors über zwei Pins. Bei jedem Puls des *Step*-Pins bewegt sich der Motor um einen Schritt. Die Richtung hängt vom Level des *Dir*-Pins ab.

3.2.4 Zusammenfassung der Anforderungen in ein Designkonzept

Die einzelnen Komponenten des Systemes sind in einem Blockschaltbild zusammengefasst:

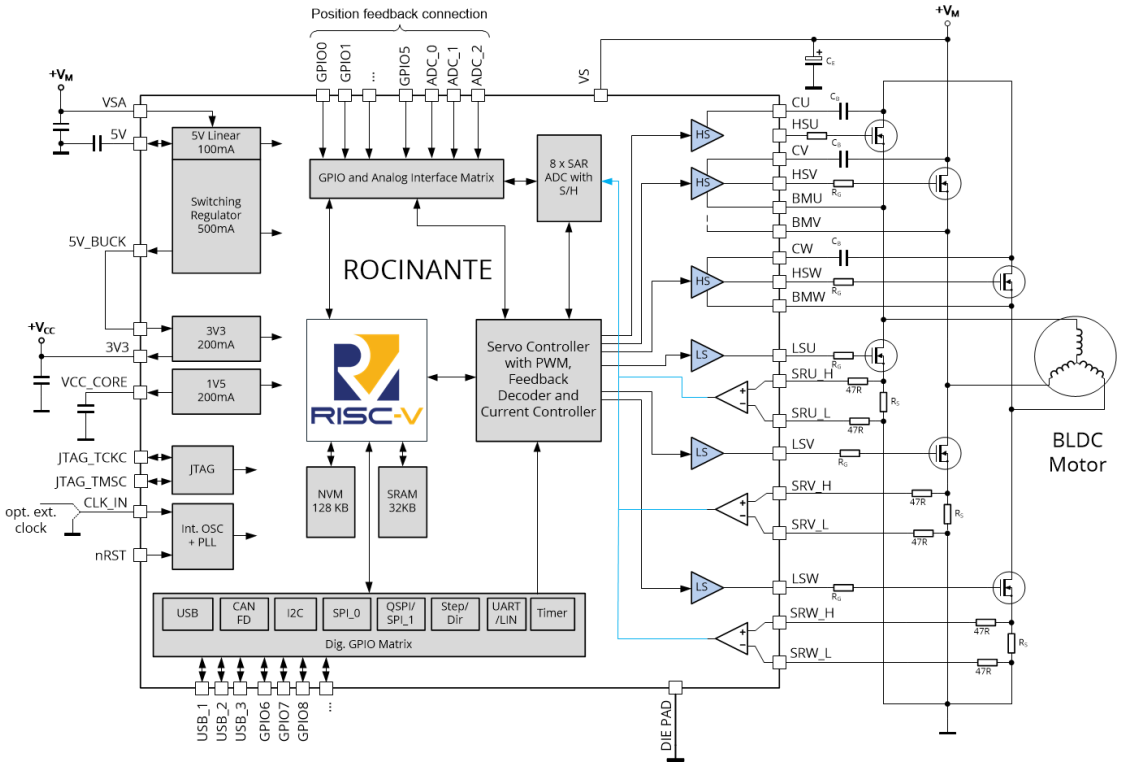

Abbildung 3.1: Designkonzept des Projektes ROCINANTE [9]

Es ist noch nicht festgelegt, in welchen Mengen einzelne Module mehrfach zur Verfügung stehen werden. Dies wird abhängig vom Hardwareaufwand und von genaueren Systemanforderungen gegen Ende der Hardwareentwicklung entschieden und ist dementsprechend nicht Teil dieser Arbeit.

3.3 Designüberlegungen zu Beginn des Projektes

Für die Implementation von Software und Hardware werden zu Projektbeginn mehrere Designrichtlinien definiert. Ziele dieser Richtlinien sind:

- Zusätzlicher Aufwand in Form von Prozessorrechenleistung durch ineffizientes Design sollen vermieden werden. Hierfür müssen die einzelnen Komponenten des Systems aufeinander abgestimmt sein, damit kein zusätzlicher Aufwand durch Verknüpfungen unterschiedlicher Systeme entsteht.
- Die Verwendung von Komponenten soll keine Überraschungen für einen Applikationsentwickler enthalten. Um dies zu erreichen soll jedes Feature genau die Funktion erfüllen, die zu erwarten ist. Weitere unerwartete Nebeneffekte sollen vermieden werden.
- Die Komponenten sollen leicht verständlich sein. Während hierfür eine qualitativ hochwertige Dokumentation essentiell ist, welche nicht Teil dieser Arbeit ist, so ist durch geschicktes Design ein System leichter verständlich. So kann zum Beispiel die Namensgebung für die Dokumentation so gewählt werden, wie ein Entwickler es auch von anderen Produkten kennen könnte.

Die Umsetzung dieser Ziele in Richtlinien geschieht auf empirischer Basis.

3.3.1 Designrichtlinien für die Software

Vermeidung impliziter Abhängigkeiten

Durch die Verwendung von Header-Dateien lässt sich ein Programm in funktional unabhängige Blöcke trennen. Dies erlaubt es, ein Projekt modular zu organisieren. Durch die Präprozessordirektive *include* können dann für eine gegebenene Quelldatei die benötigten Abhängigkeiten in Form von Header-dateien eingefügt werden.

Beim Einfügen zwei (oder mehrerer) Header-Dateien hintereinander stehen die Inhalte der ersten Header-Datei der zweiten Header-Datei zur Verfügung. Während der Programmentwicklung kann es leicht passieren, dass diese verfügbaren Inhalte aus der ersten Header-Datei in der zweiten Header-Dateien verwendet werden, ohne die daraus resultierende Abhängigkeit explizit durch eine *#include*-Direktive in der zweiten Header-Datei zu kennzeichnen. Wird daraufhin in einem Programm die zweite Header-Datei verwendet, ohne die erste zuvor per *#include*-Direktive einzufügen, entstehen Fehler bei der Kompilierung.

Im Rahmen dieses Projektes sollen diese impliziten Abhängigkeiten vermieden werden. Dies erfordert das sorgfältige Entwickeln von Header-Dateien.

Strukturierung der Hardwareimplementation

Für die Implementation der Hardwarefunktionalität soll die Software einen hohen Grad an Modularität haben. Dementsprechend soll eine Quelldatei nur die benötigte Funktionalität an deklarierten Konstanten, Makros und Funktionen einbinden (*#include*), die benötigt wird.

Die Definitionen der Hardware-Register (Memory Mapping) wird in separate Header Dateien getrennt - für jedes Modul eine Datei. Für die häufigsten Verwendungszwecke jedes Hardwaremoduls werden Implementationen programmiert, welche über weitere Header Dateien zur Verfügung gestellt werden. Dies erlaubt es einer Applikation, nur die fertige Implementation zu verwenden, ohne die Hardwareinformationen laden zu müssen. Dies reduziert die Menge an vordefinierten Funktionen, Konstanten und Makros auf die für eine Applikation nötige Hardware.

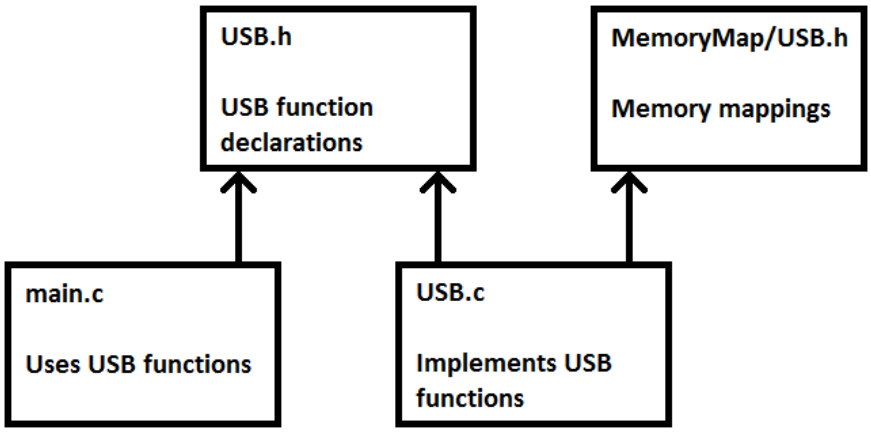

Abbildung 3.2: Strukturierung der Header-Dateien für Hardwarefunktionen

Konfigurationsmöglichkeiten

Bei der Entwicklung von Hardwarefunktionen können oft verschiedene Ansätze für die Implementationen gewählt werden, welche verschiedene Stärken und Schwächen haben. Im Rahmen dieser Arbeit soll für solche Fälle ein Konfigurationsmechanismus entwickelt werden, durch den zwischen verschiedenen Implementationsvarianten gewechselt werden kann ohne weitere Laufzeitkosten zu generieren.

3.3.2 Designrichtlinien für die Hardware

Für den einfachen Betrieb von Hardwaremodulen werden mehrere Ansätze verfolgt, um die Bedienung zu erleichtern. Hierbei soll darauf geachtet werden, dass für einen Entwickler die inkrementelle Inbetriebnahme - der erste Test, Aktivierung weiterer Features etc. - möglichst einfach gestaltet wird:

- Beim Systemstart soll ein Minimum an Konfiguration erforderlich sein für die Verwendung.
- Die Konfiguration zur Aktivierung von Funktionalität soll für einen Entwickler möglichst leicht und übersichtlich sein.
- Typische Nutzungsfälle (Use Cases) eines Moduls sollen mit möglichst wenig Aufwand konfigurierbar sein.

3.4 Realisierung der Implementation

3.4.1 Strukturierung der Software

Implementation von Hardwaredefinitionen

Oft werden für einen Mikrocontroller sämtliche Hardwaredefinitionen für die Software in eine einzige Header-Datei zusammengefasst. Dieser Ansatz ermöglicht es, mit nur einer Präprozessordirektiven (*#include*) die Definitionen sämtlicher Hardwaremodule bereitzustellen. Für dieses Projekt wird jedoch ein anderer Ansatz gewählt - jedes Hardwaremodul bekommt seine eigene Headerdatei für seine Hardwaredefinitionen. Während dies zwar in vielen einzelnen Dateien resultiert, erlaubt es jedoch, in einer Quelldatei exakt nur die Hardwaredefinitionen zu laden, die auch benötigt werden.

Die Hardwaredefinitionen sind als *memory mapped structs* implementiert. Alle Register eines Hardwaremoduls sind in einer *struct* aufgelistet und die Startaddresse (*Base address*) des Moduls ist über ein einziges Makro festgelegt. Zusätzlich ist für jedes Register ein Makro für den direkten Zugriff angelegt.

Innerhalb eines Hardwareregisters sind oft mehrere Bitfelder nebeneinander platziert. Die Programmiersprache C hat einen Mechanismus für Bitfelder, jedoch ist dieser nicht für die

Repräsentation von Hardware brauchbar. Da die Anordnung mehrerer Bitfelder von der gegebenen Implementation der Sprache C definiert wird [5] (*implementation defined*), ist es nicht möglich, Bitfelder portabel zu verwenden. Für die Verwendung innerhalb eines Projektes ist es zwar möglich, der Implementation entsprechend korrekt Bitfelder zu verwenden, jedoch hat dies ein hohes Fehlerpotenzial und erschwert eine Portierung auf eine andere Plattform. Als Alternative werden sogenannte Mask-Shift-Konstanten[3] verwendet. Diese erlauben eine portable Definition der Struktur der Hardwareregister.

Konfigurationsmechanismen zur Anpassung der Software

Um die Verwendung mehrerer Implementationen mit verschiedenen Vor- und Nachteilen zu ermöglichen, wird ein Konfigurationsmechanismus verwendet. Dieser basiert auf dem Präprozessor, um Änderungen am Programmablauf vornehmen zu können, ohne zusätzliche Kosten während der Laufzeit zu verursachen.

Deartige Konfigurationsmechanismen verwenden oft *#ifdef*-Statements innerhalb von Code verwendet. Die hier implementierten Konfigurationen nutzen dies nicht, da solch ein Ansatz mehrere Nachteile mit sich bringt:

- Verwendung von *#ifdef* in Code führt oft zu vielen kleinen Blöcken, die durch den Präprozessor aktiviert oder deaktiviert werden. Dies kann sehr schnell zu unübersichtlichem Code führen - insbesondere wenn mehrere Optionen einen Codeabschnitt an mehreren Stellen beeinflussen [4].
- Bei Ausschluss eines Codeabschnittes durch den Präprozessor wird der Abschnitt entfernt, bevor der Compiler ihn verarbeitet. Als Resultat kann es beispielsweise bei der Umbenennung von einer Variable dazu kommen, dass durch den Präprozessor nicht kompilierter Code übersehen wird. Wenn anschließend nicht alle Konfigurationsoptionen getestet werden, können Fehler entstehen.

Um diese Nachteile zu vermeiden, wird das *#ifdef*-Statement nur in Kombination mit anderen Präprozessordirektiven wie *#define* verwendet. Innerhalb von Codeabschnitten wird statt *#ifdef* eine *if*-Anweisung verwendet. Die Bearbeitung des Präprozessors resultiert dann in *if(1)* oder *if(0)* im Code. Dies erlaubt es dem Compiler, einen inaktiven

[3] Ein Bitfeld ist über Mask-Shift Konstanten verwendbar. Die Maske wird genutzt, das Bitfeld aus dem Gesamtwert mittels Und-Verknüpfung zu extrahieren. Die Shift-Konstante gibt an, wieviele Bitpositionen versetzt das Bitfeld positioniert ist.

Codepfad auf Fehler zu überprüfen und anschließend durch Optimierungen einen inaktiven Codepfad zu eliminieren (*Dead Code Elimination*).

3.4.2 Strukturierung der Hardware

Die Steuerung der Hardware durch Software erfolgt durch sogenannten *Memory Mapped I/O*. Hierbei liegen Kontrollregister zur Ansteuerung der Hardware im Addressraum des Hauptspeichers vor. Lesen oder Schreiben von solchen Hardwareregistern wird somit mit den gleichen Befehlen unternommen, wie Lesen oder Schreiben des Hauptspeichers.

Implementation der Hardwareregister

Durch die verfügbaren Zugriffsmöglichkeiten des Prozessors sind mehrere Limitationen für die Manipulation von Speicher gegeben:

- Speicherzugriffe erfolgen in Blöcken von 1, 2 oder 4 Bytes.
- Speicherzugriffe müssen ihrer Zugriffsgröße entsprechend ausgerichtet sein. Hierfür muss die Speicheraddresse durch die Anzahl an Bytes, auf die zugegriffen wird, teilbar sein.
- Ein Speicherzugriff erfolgt atomar, er kann nicht durch einen Interrupt unterbrochen werden.
- Zahlenwerte, die größer als ein Byte sind, werden im *Little-Endian-Format* gespeichert. So wird beispielsweise bei einem Zwei-Byte-Wert das niederwertige Byte im Speicher vor dem höherwertigen Byte positioniert.
- Um einen Wert im Speicher zu modifizieren, muss er erst gelesen werden, dann die Modifikation innerhalb des Prozessors durchgeführt werden und anschließend der Wert zurück in den Speicher geschrieben werden (*Read-Modify-Write*). Dieser Vorgang erfordert mindestens drei Instruktionen und ist daher nicht atomar.
- Eine Modifikation in den unteren 11 Bit eines Wertes durch die Bitoperationen *AND*, *OR* und *XOR* kann mit nur einer Instruktion durchgeführt werden. Die anderen Bits benötigen mehr Instruktionen. In Kombination mit den Instruktionen für das Laden und Speichern können somit die unteren 11 Bits mit nur drei Instruktionen verändert werden (*Read-Modify-Write*-Operation).

- Zum Extrahieren eines Bitfeldes innerhalb eines gelesenen Wertes, müssen zwei Instruktionen verwendet werden. Falls ein Bitfeld weniger als 12 Bits hat und an der ersten Bitposition des umschließenden Wertes steht, kann es mit nur einer Instruktion extrahiert werden.

Bei der Implementation der Interfaces zwischen Hardware und Software - den Hardwareregistern - werden die Limitationen des Prozessors berücksichtigt, um eine möglichst effiziente Steuerung der Hardwaremodule zu ermöglichen:

- Hardwareregister sind typischerweise 4 Bytes groß. Kleinere Hardwareregister liegen nur vor, wenn mehrere gleich große solcher Register folgen. So werden zum Beispiel 4 Register von 1 Byte Größe hintereinander im Addressraum positioniert.
- Hardwareregister sind immer korrekt im Addressraum ausgerichtet.
- Durch Zugriff auf Hardwareregister ausgelöste Funktionen werden entsprechend der Möglichkeiten des Prozessors für atomare Zugriffe entwickelt.
- Zahlenwerte in Hardwareregistern, die größer als ein Byte sind, werden dem *Litte-Endian-Format* entsprechend positioniert.
- Wenn nötig werden Hardwarefunktionen für typische *Read-Modify-Write*-Operationen eingebaut. Durch zusätzliche Hardwareregister können dann einzelne Bits atomar gesetzt, gelöscht oder invertiert werden.
- Enthält ein Register mehrere Bitfelder, so soll das am häufigsten verwendete Bitfeld an die unterste Bitposition im Register platziert werden, um den Shift-Vorgang beim Mask-Shift-Verfahren einzusparen.

Strukturierung der Hardwarekonfiguration

Um eine einfache Bedienung der Hardware zu ermöglichen, sollen die Konfigurationsmöglichkeiten entsprechend der vorher formulierten Designrichtlinien entwickelt werden. Aus diesen Richtlinien werden folgende Prinzipien gebildet:

- Beim Systemstart sind alle optionalen Funktionalitäten deaktiviert. Dies entspricht typischerweise einem Bitwert von 0 für die entsprechende Option in den Konfigurationsregistern.

- Die Benennung von Hardwareregistern soll aussagekräftige Namen verwenden - typischerweise die gleichen Namen, die auch in anderen Mikrocontrollern verwendet wurden.
- Ein Modul soll, wenn es für seine Verwendung konfiguriert wurde, möglichst wenig neue Konfigurationsschritte benötigen. So soll beispielsweise eine Kommunikationsschnittstelle beim Lesen oder Schreiben von Daten möglichst keine weitere Konfigurationszustände beeinflussen.

3.4.3 Implementation der Kommunikationsschnittstellen

Für das Design der Kommunikationsschnittstellen werden die einzelnen Funktionen der Schnittstellen in drei Kategorien eingestuft:

- Essentielle Funktionalität: Hierzu zählt die eigentliche Kommunikation sowie die Unterstützungen für spezialisierte Anwendungsbereiche. Für diese Features wird die Hard- und Software so ausgelegt, dass die Verwendung so effizient wie möglich abläuft.
- Erweiterte Funktionalität: Optionale Features, welche nicht immer verwendet werden. Diese sollen solange sie deaktiviert sind keinerlei zusätzliche Kosten für die essentielle Funktionalität verursachen.
- Nebensächliche Funktionalität: Hierzu zählen die Features, die als Nebenprodukt der Entwicklung entstanden sind, jedoch nicht als vorgesehende Verwendungszwecke konzipiert sind. Ein Beispiel hierfür sind untypisch lange Datenpackete für Bussysteme, die keine vorgegebenen Maximaldatenlängen haben. Solche Features haben keine direkte Unterstützung und können so auch einen hohen Softwareaufwand bei der Verwendung mit sich bringen.

Die vorliegendende Arbeit fokussiert sich auf die essentielle Funktionalität. Zusätzliche Funktionalität wird als Teil des Entwicklungsprozesses getestet, jedoch nicht immer in der entwickelten Software verwendet. Die nebensächliche Funktionalität wird weder implementiert noch verifiziert.

USB 2.0

Für die Kommunikation über USB implementiert die Hardware einen Endpoint der Kategorie *Commmunications Class Device* (*CDC*). Für das Senden und Empfangen von Daten stehen zwei FIFO Buffer von 64 Byte Größe zur Verfügung. Dies ist für die verwendete Applikation - das TMCL-Protokoll - ausreichend, um keine FIFO-Buffer in der Softwareimplementation zu benötigen. Als Resultat besteht die Software für USB nur aus drei simplen Funktionen:

- Eine Konfigurationsfunktion setzt *Vendor ID* und *Product ID* und aktiviert die Kommunikationsschnittstelle.
- Eine Lesefunktion erlaubt das Lesen mehrerer Bytes in einen als Parameter übergebenen Buffer. Sind nicht genug Daten verfügbar, wird ein Fehler zurückgegeben
- Eine Schreibfunktion erlaubt das Schreiben von mehreren Bytes. Es können nicht mehr Bytes auf einmal geschickt werden, als Raum im FIFO der USB-Hardware ist. Dies ist für die Implementation des TMCL-Protokolls ausreichend, da die Datagramme nur 9 Bytes groß sind.

CAN FD

Die extern entwickelte CAN-Schnittstelle kann im Design nicht an die Designrichtlinien angepasst werden. So erfordert das wiederhole Senden gleichartiger Datagramme bei jedem Sendevorgang das Setzen von 6 verschiedenen Konfigurationseinstellungen. Diese Einstellungen erfolgen durch mehrere Bits, die zusammen mit den Daten in eine Buffer-Region im Speicher geschrieben werden. Um die zu schreibenden Daten zu generieren, müssen die einzelnen Optionen in verschiedene Bitfelder gefüllt werden. Um diesen Vorgang nicht immer wieder neu durchzuführen, werden in der Software alle Konfigurationseinstellungen als Zwischenergebnis gespeichert. Als Resultat muss die Sendefunktion nicht mehr die Konfigurationseinstellungen setzen, sondern nur noch die vorkonfigurierten Zwischenergebnisse zusammen mit den zu sendenden Daten an das Hardwaremodul schicken.

```
1 void CAN_configure(CanFrameConfig *config)
2 {
3     TBUF[0]  = (config->timeStampEnable)?           CAN_TBUF_0_TTSEN_MASK:0;
4     TBUF[1]  = (config->identifierExtension)?       CAN_TBUF_1_IDE_MASK  :0;
5     TBUF[1] |= (config->remoteTransmissionRequest)? CAN_TBUF_1_RTR_MASK  :0;
```

```
6       TBUF[1] |= (config->extendedDataLength)?        CAN_TBUF_1_EDL_MASK  :0;
7       TBUF[1] |= (config->switchBitRate)?             CAN_TBUF_1_BRS_MASK  :0;
8       TBUF[1] |= CAN_TBUF_1_DLC(config->dataLengthCode);
9   }
```

Listing 3.1: Zwischenspeicherung der CAN-Schnittstellenkonfiguration

SPI

Die Implementation der SPI Schnittstelle ist komplexer, als die anderen Schnittstellen, da die existierende Hardware noch nicht den Designprinzipien entsprechend angepasst ist. Die nötigen Anpassungen sind fertig geplant, jedoch noch nicht in der Hardware umgesetzt.

Beispielsweise gibt es je nach Betriebsmodi unterschiedliche Wege, den Sendevorgang zu starten. Ein Betriebsmodus erlaubt es, ein Datagramm automatisch immer wieder zu senden. Das Senden wird in diesem Modus durch das Setzen eines Bits in einem Konfigurationregister aktiviert und durch Löschen des Bits deaktiviert. Im Betriebsmodus für das Senden eines einzelnen Datagrammes wird dieses Bit ignoriert. Stattdessen wird das Senden gestartet, sobald das erste der vier Datenregister beschrieben wird. Dies fordert das Füllen des Datenbuffers in invertierter Reihenfolge. Um die Bedienung zu erleichtern, soll das Senden immer durch das explizite Konfigurationsbit gestartet werden. Dies erlaubt es, den Datenbuffer in beliebiger Reihenfolge zu füllen. Für den Betriebsmodus, der nur ein Datagram sendet, wird das Konfigurationsbit für das Senden durch die Hardware zurückgesetzt, sobald der Sendevorgang abgeschlossen ist. Somit stellt das Konfigurationsbit gleichzeitig ein Status da - ob gerade gesendet wird, oder nicht.

UART

Die UART-Schnittstelle hat keinen eingebauten FIFO-Buffer für das Senden oder Empfangen von Daten. Für die Übertragung mehrerer Datenpakete muss daher die Software regelmäßig neue Daten an das UART-Modul übergeben. Da eine UART-Übertragung mit typischen Baudraten von 9600 bis 115200 im Vergleich zum Prozessor mit 40 MHz relativ langsam ist, sollen Daten asynchron übertragen werden. Hierfür speichert die Software zu sendende Daten in einem Ringbuffer, der durch einen Interrupt über die UART-Schnittstelle sendet. Empfangene Daten werden in einen zweiten Ringbuffer platziert, wo die Software diese einlesen kann.

Neben der Größe der Ringbuffer kann außerdem das Verhalten des Sendebuffers konfiguriert werden. Gibt die Software mehr Daten, als in den Buffer passen, kann entweder gewartet werden, bis ausreichend Platz im Buffer ist, oder die Software verwirft die Daten, die nicht in den Buffer passen, um das Warten zu vermeiden.

I2C

Die I2C-Schnittstelle stand erst gegen Ende der Arbeit als Erstversion in Hardware zur Verfügung. Die Strukturierung der Hardwareregister wurde anschließend komplett überarbeitet.

Die Erstversion basierte darauf, jedes Byte des Datagrams einzeln von der Software an das Hardwaremodul zu übergeben und dazu eine Reihe an Anweisungen mit zu übergeben. Einige Beispiele für diese Anweisungen sind „Sende ein Startbit“, „Sende die Busaddresse“, „Sende das Datum“. Dies sollte das Senden von beliebig großen Datagrammen ermöglichen. Als Folge dieser Designentscheidung wurde das Senden typischer, kleiner Datagramme sehr kompliziert für die Software, da einzelne Schritte für das Bussystem von Software aus gesteuert werden mussten.

Bei der Überarbeitung der Erstversion wurde dieses Verfahren umgeändert, entsprechend der Kategorisierung von verschiedenen Funktionalitäten. Die neue geplante Schnittstelle unterstützt kurze Datagramme von bis zu 8 Bytes auf eine einfache Weise. Das Senden von Start- und Stopp-Bits sowie der Busaddresse muss nicht mehr über die Software gesteuert werden. Es ist noch nicht entschieden, ob das Senden größerer Datagramme unterstützt werden soll, oder nicht. In die neue Schnittstelle würde dies über einen Interrupt werden, der weitere Daten zum Senden von der Software anfordert.

Weitere Kommunikationsschnittstellen

Während der Entwicklung sind USB und CAN aktiv verwendet worden, um über das TMCL-Protokoll mit einem PC zu kommunizieren. Die Implementationen von SPI und UART sind teilweise implementiert, eine Vervollständigung der Implementation ist durch Hardwarefehler zum Ende dieser Arbeit hin nicht möglich. Für die I2C-Schnittstelle ist ein Design fertiggestellt, die Hardware jedoch nicht rechtzeitig für eine Implementation der Software verfügbar. Die QSPI Schnittstelle stand ebenfalls nicht rechtzeitig zur Verfügung.

3.4.4 Implementation von Interrupts und Exceptions

Hardwareunterstützung für Interrupts und Exceptions

Der verwendete RISC-V-Prozessor verfügt über drei Kategorien von Interrupts:

- *Timer Interrupts* werden durch einen eingebauten Echtzeitzähler ausgelöst. Sofern dieser Zähler über einem dazugehörigen Schwellwert ist, wird ein Interrupt ausgelöst.
- *External Interrupts* werden durch Hardwareperipherien ausgelöst. Alle Kommunikationsschnittstellen sowie Timer, GPIOs finden über diesen Interrupt-Typ statt.
- *Software Interrupts* werden durch eine Instruktion in der Software ausgelöst.

Bei folgendem Fehlverhalten wird eine *Exception* ausgelöst:

- Unausgerichteter Zugriff auf den Speicher. Dies kann beim Laden von Instruktionen oder beim Laden oder Speichern durch das Programm geschehen.
- Nicht zulässige Speicheraddresse. Dies kann beim Laden von Instruktionen oder beim Laden oder Speichern durch das Programm geschehen.
- Eine inkorrekte Instruktion wurde geladen.

Der Prozessor unterstützt zwei Optionen für das Verhalten bei Interrupts und Exceptions. Beim direkten Modus wird unabhängig davon, welches dieser Ereignisse aufgetreten ist an eine konfigurierbare Programmaddresse gesprungen. Beim Vektor-Modus wird an eine von dem Ereignistyp abhängige Position in einem Vektor gesprungen. Ein Programm kann an Sprunganweisungen an diese Positionen schreiben, um verschiedene Interrupt-Handler oder den Exception-Handler zu erreichen.

Der *Timer Interrupt* wird nur für die Erstimplementation eines Systicks verwendet. Der *Software Interrupt* wird in der Applikation nicht verwendet. Der *External Interrupt* wird von einem *Programmable Interrupt Controller* (kurz *PIC*) ausgelöst, über den alle Interruptquellen von verschiedenen Modulen zusammengeschlossen werden. Über den PIC können einzelne Interruptquellen aktiviert oder deaktiviert werden. Ein Statusregister im PIC zeigt alle aktiven Interrupt-quellen an. Nach Bearbeitung eines Interrupts muss im PIC der aktive Interrupt deaktiviert werden.

Vorliegende Implementation

Zu Beginn des Projektes ist ein einfacher Interrupt-Mechanismus bereits implementiert. Der direkte Modus wird verwendet und ein zentraler sogenannter *trap handler*, der alle Register auf dem Stack abspeichert, je nach Ereignis eine *Interrupt service routine* (*ISR*) oder eine *Exception service routine* (*ESR*) aufruft und anschließend die Register wieder vom Stack wiederherstellt und zum Hauptprogramm zurückkehrt. Keine der beiden aufgerufenen Routinen hat Funktionalität implementiert.

Optimierung der Sicherung von Registern für Interrupts

Bei der Behandlung eines Interrupts müssen alle von der ISR verwendeten Register vorher gesichert und danach wiederhergestellt werden. Für dieses Verhalten gibt es für andere Prozessorarchitekturen durch den Compiler die Möglichkeit, eine C-Funktion als Interrupt-Funktion zu deklarieren. Als Resultat werden alle Register, die diese Funktion verwendet zu Beginn der Funktion gesichert und am Ende wiederhergestellt. Jedoch ist bei dem verwendetem Compiler - GCC - für RISC-V diese Funktionalität noch nicht in der verwendeten Compilerversion verfügbar[4]. Die Interrupt-Funktionen verhalten sich daher der RISC-V-Aufrufkonvention entsprechend. Diese unterteilt die Register in 16 Register, die vor einem Funktionsaufruf gesichert werden müssen (*Caller-saved*), und 15 Register, die eine aufgerufene Funktion selbst sichern muss (*Callee-saved*). Somit muss vor dem Aufruf einer C-Funktion zur Bearbeitung von Interrupts der *trap handler* die 16 *Caller-saved* Register speichern.

Wechsel des Prozessormodus für die Interruptbehandlung Für den direkten Interrupt- und Exceptionmodus muss in der Software ein spezielles, sogenanntes *Control and Status Register* (kurz *CSR*) ausgewertet werden, um die korrekte Handler-Funktion aufzurufen. Diese Auswertung kostet ca. 10-15 Zyklen, die durch die Verwendung des Vektor-Modus gespart werden können. Für diesen Modus erhält jeder der drei Interrupt-Kategorien sowie Exceptions einen separaten Handler. Jedoch muss jeder dieser vier Handler das Sichern und Wiederherstellen von Registern implementieren - je 2-Byte-Instruktionen für Sichern und Wiederherstellen eines Registers, für 16 Register zusätzliche dreimal implementiert resultiert in ca. 192 zusätzlichen Bytes.

[4] Die Unterstützung dieser Funktionalität ist bereits entwickelt [24], jedoch zum Zeitpunkt der Arbeit noch nicht in der verwendeten *stable* Version von GCC dabei.

Auswahl des Handlers für externe Interrupts

Da sämtliche Hardwareperipherien über den externen Interrupt implementiert sind, soll der Handler für externe Interrupt die korrekte Funktion für jede mögliche externe Interruptquelle aufrufen. Die zu bearbeitenden Interrupts sind über den PIC verfügbar - jede Quelle korrespondiert zu einem Bit in einem gemeinsamen Register (*Interrupt Flag Register*, kurz *IFR*).

Für jede Interruptquelle wird ein Funktionszeiger in einem Array platziert. Die Indexe der Funktionen entsprechen der Bitpositionen im IFR. Somit muss der Interrupt-Handler die Bitpositionen der auf Eins gesetzen Bits bestimmen. Dieser Vorgang erfordert Bitshift-Operationen in einer Schleife, um die gesetzten Bits zu finden. Die niedrigste Bitposition wird so zuerst, die höchste Bitposition zuletzt geprüft. Dieser Ansatz muss für jedes Bit eine Schleifeniteration abarbeiten, unabhängig davon, ob das Bit gesetzt bzw. der Interrupt aktiv ist oder nicht. Als Folge davon haben die Interrupts mit höheren Bitpositionen eine höhere Latenz.

```
1  void IRQ_External()
2  {
3      uint32 interrupts = PIC_IFR;
4      uint32 mask = 1;
5
6      // Iterate through the interrupt handlers
7      for(uint8 i = 0; i < ARRAY_SIZE(IRQ_Vector); i++, mask<<=1)
8      {
9          if(interrupts & mask)
10         {
11             // Call the handler
12             IRQ_Vector[i]();
13             // Clear the interrupt
14             PIC_IFRC = mask;
15         }
16     }
17 }
```

Listing 3.2: Erste Implementation der Auswahl der Interrupt-Handler

Um die Auswahl der Interrupt-Handler weiter zu verbessern, wird im PIC ein weiteres Register ergänzt. Dieses Register enthält die Bitposition des untersten auf Eins gesetzen Bits. Der Wert dieses Registers kann direkt als Index für den Array aus Funktionszeigern verwendet werden. Dies reduziert die benötigte Zeit für die Auswahl aktiver Interrupts.

Des weiteren wird durch diesen Mechanismus nicht mehr über alle Bits, sondern nur noch über alle auf Eins gesetzen Bits iteriert.

```
1  void IRQ_External()
2  {
3      while(PIC_IFR)
4      {
5          // Get the index of the next active bit
6          uint32 index = PIC_IF_INDEX;
7          // Call the corresponding interrupt handler
8          IRQ_Vector[index]();
9          // Clear the interrupt in the PIC
10         PIC_IFR = 1 << index;
11     }
12 }
```

Listing 3.3: Verbesserte Implementation der Auswahl der Interrupt-Handler durch Hardwareunterstützung

Weitere Schritte für Interrupts

Für die Interrupts ist ein konfigurierbares Prioritätenverfahren in Hardware geplant. Mit vorraussichtlich 8 verschiedenen Prioritätsstufen soll so die genauere Kontrolle über die Abarbeitung mehrerer Interrupts möglich sein. Des weiteren soll die Verschachtelung mehrerer Interrupts durch Software ermöglicht werden. Durch diese beiden Erweiterungen sollen geringe Latenzen für Aufgaben hoher Priorität, wie Echtzeitsysteme ermöglicht werden.

Implementation von Exceptions

Im normalen Programmablauf sollte eine Exception nicht auftreten. Inkorrekte Instruktionen oder Speicheraddressen deuten typischerweise auf einen Fehler im Programmablauf hin, der nicht während der Laufzeit lösbar ist. Für diese Fälle wird das Programm in einer Endlosschleife gehalten, um eine Fehleranalyse durch den Debugger zu ermöglichen.

Exceptions durch unausgerichtete Speicherzugriffe können hingegen während der Laufzeit behoben werden. Durch byteweise Speicherzugriffe können die Daten einzeln gelesen oder

geschrieben werden, und das Verhalten des unausgerichteten Speicherzugriffes zu emulieren. Um dies zu implementieren, muss der Exception Handler mit dem gespeicherten Kontext - den Registern des unterbrochenen Programmcodes - interagieren.

Die Umsetzung der Emulation unausgerichteter Speicherzugriffe erfolgt folgendermaßen:

1. Zu Beginn des Exception-Handlers im Assemblercode werden alle 31 Register gespeichert.
2. Die C-Funktion für Exceptions wird aufgerufen. Als Parameter erhält sie eine Nummer, die den Exception-Typ angibt, die Addresse, an der das Programm eine Exception verursacht hat, und einen Zeiger auf die gespeicherten Register.
3. Ist der Exception-Typ ein unausgerichteter Speicherzugriff, werden 16 Bits von der Programmaddresse geladen, die die Exception verursacht hat.
4. Es wird geprüft, ob eine komprimierte Instruktion vorliegt. Wenn nicht, werden weitere 16 Bit geladen. Das Laden der Instruktion muss in zwei Schritten geschehen, da sonst eine 32-Bit Instruktion unausgerichtet geladen werden könnte - Instruktionen müssen nur entlang 16-Bit-Grenzen im Speicher ausgerichtet sein, selbst wenn es 32-Bit Instruktionen sind.
5. Die von der Instruktion verwendete Speicheraddresse wird berechnet,
6. Für Lesebefehle wird byteweise von der Speicheraddresse gelesen und der Wert in das passende Register in den gespeicherten Registern geschrieben.Für Schreibbefehle wird der Wert byteweise gespeichert.
7. Die C-Funktion meldet mittels des Rückgabewertes, ob eine komprimierte oder normale Instruktion emuliert wurde und ob die gespeicherten Register modifiziert wurden.
8. Im Assembler-Handler wird der Programmzähler verschoben - 2 Byte für eine komprimierte und 4 Byte für eine normale Instruktion.
9. Die gespeicherten Registerwerte werden wieder in die Register geladen. Die *Callee-saved* Register werden nur geladen, wenn die C-Funktion eine Modifikation der gespeicherten Register gemeldet hat.

Die Emulation des unausgerichteten Speicherzugriffes benötigt ca. 160 Zyklen, während ein normaler Speicherzugriff nur 1-2 Zyklen benötigt. Neben den Geschwindigkeitseinbußen ist ein unausgerichteter Speicherzugriff auch nicht mehr atomar.

3.4.5 Implementation allgemeiner Prozessorkomponenten

GPIOs

Die Implementation für die GPIOs ist simpel - wie auch in vielen anderen Mikrocontrollern gibt es für jeden Pin je ein Bit in mehreren Registern für verschieden Operationen. Diese Register sind 4 Byte groß und umfassen demendsprechend 32 Pins:

- Ein Register legt fest, ob ein Pin ein Input oder Output ist.
- Ein Register erlaubt es, den Pinstatus zu Lesen oder zu Schreiben.
- Ein Register erlaubt das atomare Aktivieren von Pins.
- Ein Register erlaubt das atomare Deaktivieren von Pins.
- Ein Register erlaubt das atomare Umschalten von Pins.
- Ein Register legt fest, welche Pins ein Interrupt aktivieren.
- Ein Register legt fest, welche Polarität einen Interrupt produziert.
- Ein Register zeigt an, welche Pins einen Interrupt ausgelöst haben.

Die einzige geplante Änderung sieht vor, die Nutzung von Interrupts zu verbessern. Es sollen auch Flanken als Auslöser für einen Interrupt nutzbar sein. Des weiteren sollen mehrere Pin-Gruppen gebildet werden, die zusammen einen gemeinsamen Interrupt-Handler nutzen. Dies soll die Anzahl an Pins pro Interrupt-Handler reduzieren.

Systick

Eine simple Implementation des Systicks ist zu Beginn des Projekts bereits vorhanden. Diese nutzt den Echtzeitzähler des Prozessors, welcher einen Mechanismus zum Generieren von Interrupts eingebaut hat. Hiermit wird jede Millisekunde ein Interrupt generiert, der eine Systick-Variable hochzählt.

Durch die Verwendung eines Echtzeittimers sorgt ein Stopp des Systems durch einen Debugger dafür, dass bei Fortsetzung des Programmes durch den Debugger das Programm wiederholt den Systick-Interrupt aufruft. Dies geschieht solange, bis alle verpassten Millisekundeninvervalle aufgeholt wurden. Als Resultat ist die Verwendung eines Debuggers, insbesondere beim manuellen Durchlaufen durch die Instruktionen (*Single stepping*) stark eingeschränkt.

Um die Nutzung eines Debuggers mit aktiven Systick zu erleichtern, wurde daher eine andere Systick-Implementation entwickelt, die nicht auf Interrupts basiert. Stattdessen wird der Wert des Echtzeittimers genommen und per Division von seiner Frequenz von 40 MHz herunter auf 1 kHz herunterskaliert. Dies ergibt den gewünschten Systickwert mit Millisekundengenauigkeit. Diese Implementation hat keine negative Auswirkung für den Debugger und entfernt gleichzeitig die durch den kontinuierlichen Interrupt generierte Systemlast. Allerdings benötigt das Ermitteln des aktuellen Systickwertes durch die Divison des 64 Bit großen Timerwertes mehr Rechenleistung.

Die Auswahl der Implementation - mit oder ohne Interrupts - geschieht über den entwickelten Konfigurationsmechanismus. Des weiteren kann für die Version ohne Interrupts statt dem Echtzeittimer auch ein Prozessortimer verwendet werden - dieser hat dieselbe Frequenz, jedoch kein Echtzeitverhalten. Dies die Zeitreferenz des Systicks beim Stopp des Prozessors durch einen Debugger ebenfalls anhalten.

General-Purpose-Timer

Für die Timer sind zwei Varianten verfügbar. Eine simpler Timer (*Basic Timer*) mit wenig Funktionalität soll für einfache Aufgaben verwendbar sein während ein komplexer Timer (*Advanced Timer*) für komplexere Aufgaben zur Verfügung steht. Die Erstimplementationen beider Timer hatten ihre eigenen unterschiedlichen Hardwareregisterstrukturen.

Um die Verwendung der Timer zu erleichtern, soll der komplexe Timer eine leistungsstärkere Version des einfachen Timers sein. Um dies zu erreichen, müssen die Strukturen der Hardwareregister miteinander kompatibel sein. Als Resultat kann eine Implementation, die einen einfachen Timer verwendet, auch einen komplexen Timer verwenden, ohne dass sich das Verhalten des Programmes ändert. Des weiteren muss ein Entwickler nicht zwei verschiedene konzeptionelle Modelle für verschiedene Timer verwenden, sondern kann das die Timer als das gleiche konzeptionelle Modell mit optionaler Zusatzfunktionalität betrachten.

Um dieses Ziel umzusetzen, wird im Verlauf von mehreren Iterationen die Hardwareregisterstrukturen beider Timer immer weiter aneinander angeglichen. In der finalen Iterationen bildet die Hardwareregisterstruktur des einfachen Timers eine Teilmenge von den Registern des komplexen Timers. Des weiteren werden die zahlreichen Features des komplexen Timers modifiziert, um eine leicht verständliche Dokumentation zu erleichtern.

3.5 Funktionsverifikation

3.5.1 RISC-V Prozessor

Im Laufe der Softwareimplementation wurden einige Fehler auf inkorrekte Funktionsweise des Prozessorkerns zurückgeführt. Dieser Abschnitt beschreibt die entdeckten Fehler des Prozessors. Eine vollständige Verifikation des Prozessorkerns ist nicht Teil dieser Arbeit.

Vollständigkeit der Prozessorinstruktionen

Während der Implementation eines Systick-Mechanismus wurden mehrere Mechanismen für die Zeitmessung ausprobiert. Bei Verwendung der *RDTIME*- und *RDTIMEH*-Instruktionen - für das Auslesen eines 64-Bit Echtzeit-Timers - wurde festgestellt, dass der Prozessor diese beiden Instruktionen nicht implementiert hat. Benutzung einer der Instruktionen resultierte in einer *Illegal Instruction Exception*. Da die beiden Instruktionen Teil der Grundspezifikation von RISC-V (RV32I) sind, müssten diese eigendlich verfügbar sein.

Aufgrund dieses Prozessorfehlers wurden alle für die vorliegenden Prozessorimplementation (RV32IMC) definierten Instruktionen geprüft. Hierfür wurde per Skript die bisher entwickelte Software für den Prozessor in eine Liste aller darin vorkommenden Befehle umgewandelt und mit der Liste an definierten Instruktionen verglichen. Die fehlenden Instruktionen wurden anschließend manuell über Assemblerprogrammierung getestet. Neben den oben genannten zwei Instruktionen wurden keine weiteren fehlenden Instruktionen gefunden.

Der Fehler wurde an die Firma, die den Prozessor entwickelt hat, weitergeleitet. Der für die Instruktionen nötige Echtzeitzähler existiert bereits als Hardwaremodul und kann per *Memory Mapped IO* verwendet werden. Eine Emulation der Instruktion wäre somit zwar möglich, würde bei der aktuellen Implementation des Exception-Handlers jedoch ca. 150 Zyklen kosten.

Inkorrektes Laden von Instruktionen

Im Verlaufe des Projektes wurden viele Hardwarerevisionen verwendet, da die Hardwaremodule sich laufend geändert haben. Beim Wechsel auf eine neuere Revision begann der Prozessor *Illegal Instruction Exceptions* zu generieren. Bei dieser Prozessorausnahme stellt der Prozessor die Addresse der verbotenen Instruktion sowie die aus dem Speicher geladene Instruktion zur Verfügung. Bei Betrachtung dieser Informationen wurden nicht die Bytes als Instruktion gemeldet, die an der entsprechenden Speicheraddresse gespeichert waren. Stattdessen wurde teilweise der Wert 0, teilweise ein korrektes Byte gefolgt von Null-Bytes. Diese Symptome deuten auf ein Problem mit dem Laden der Instruktionen aus dem Speicher hin.

Da der Fehler reproduzierbar auftrat, wurde versucht, die auslösenden Umstände zu isolieren. Hierfür wurden die Position und Ausrichtung der falsch geladenen Instruktion betrachtet. Der vorliegende Prozessortyp (RV32IMC) erlaubt Instruktionen von 2 oder 4 Bytes Größe, welche an 2-Byte-Grenzen im Speicher positioniert sein müssen. Jedoch konnte der Fehler nicht durch Anpassung der Instruktionen - Ausrichtung an 4-Byte-Grenzen, keine Verwendung von komprimierten 2-Byte Instruktionen - eliminiert werden.

Zur Fehlerbehebung wurde die Laufzeit des Programmes bis zum Auftreten des Fehlers verkürzt, indem Stück für Stück andere Codeabschnitte deaktiviert wurden. Durch

eine kürzere Laufzeit war es anschließend möglich, das Programm in einer Hardware-Simulation in angemessener Zeit laufen zu lassen. Dies ermöglichte es den Hardwareentwicklern, den Fehler zu beheben. Ursache des Fehlers war ein Timing-Problem bei der Kommunikation mit dem Hauptspeicher.

3.5.2 Kommunikationsschnittstellen

Die Features der Kommunikationsschnittstellen wurden einzeln auf Fehler geprüft. Aufgrund von Verzögerungen im Entwicklungszyklus bei der Zusammenarbeit mit Hardwareentwicklern wurden aus Zeitgründen nicht alle Prozessorkomponenten vollständig getestet. Mit allen vorhandenden Kommunikationsschnittstellen wurde die wichtigste Funktionalität zuerst getestet - das Senden und Empfangen von Daten. Im folgenden Abschnitt sind einige der Fehler, die bei der Verifikation der Schnittstellen aufgetreten sind, dokumentiert.

SPI

Das SPI-Modul verfügt über mehrere verschiedene Sendemodi. Hierbei traten mehrere Fehler auf:

- Im normalen Betriebsmodus konnte nach Absenden des ersten Datagrammes kein weiteres Datagramm mehr gesendet werden.
- Bei dem kontinuierlich wiederholtem Senden eines Datagrammes sollte das Senden durch das Setzen eines Bits in einem Konfigurationsregister gestoppt werden. Dies schlug jedoch zunächst fehl.
- Der kontinuierliche Modus erzeugte kein ChipSelect-Signal.
- Bei einem Betriebsmodus für das Senden von zwei Datagrammen in Folge fehlte das ChipSelect-Signal für das zweite Datagramm.
- Beim Invertieren der Bitreihenfolge für das Senden wurde ein Bit weniger, als konfiguriert gesendet.

UART

Auch beim UART-Modul traten mehrere Fehler auf:

- Der Prescaler für die Baudrate hatte falsche Divisionsfaktoren verwendet.
- Bei invertierter Polarität der Tx-Leitung wurde bei der ersten Übertragung keine Startbitflanke erzeugt.
- Die automatische Detektion der verwendeten Baudrate führte sehr selten zu falsch übertragenen Daten, wenn parallel empfangen und gesendet wurde. Bei ca. einer von 100000 Übertragungen wurden die gesendeten Daten um eine Bitposition nach links verschoben.

4 Vorbereitung des Prozessorvergleichs

In diesem Kapitel werden die Vorbereitungen für einen aussagekräftigen Vergleich zwischen Prozessoren auf Basis von RISC-V und ARM durchgeführt. Die zu vergleichenden Messgrößen werden definiert und Mechanismen zur Messung derselben werden implementiert. In einer theoretische Vorbetrachtung sollen Bewertungskriterien bestimmt werden, um die Stärken und Schwächen der Prozessoren zu zeigen. Für die Untersuchung der Bewertungskritierien sollen Algorithmen ausgesucht werden, die zu den Kriterien passend aussagekräftig sind.

4.1 Definition von Messgrößen für den Prozessorvergleich

Für den Vergleich der beiden Prozessoren wird die Leistungsfähigkeit anhand der „Iron Law“ betrachtet:

$$\frac{1}{Performance} = \frac{Time}{Program} = \frac{Instructions}{Program} \cdot \frac{ClockCycles}{Instruction} \cdot \frac{Time}{ClockCycle}$$

Als Vereinfachung wird die Betrachtung auf Basis der „Iron Law“ auf die ersten beiden Faktoren beschränkt. Der dritte Faktor - die Zeit pro Taktzyklen - bewertet die Leistungsfähigkeit einer gegebenen Prozessorimplementation. Durch die fundamentale physikalische Limitation der Lichtgeschwindigkeit ergibt eine maximale Taktrate. Hierbei ist der kritische Pfad - der längste Weg durch die Logikschaltung, den ein Signal innerhalb eines Taktes überwinden muss - entscheidend. Je länger der kritische Pfad, desto geringer ist die maximale Taktfrequenz. Zusätzlich hat die für die Implementation zugrunde liegende Technik - FPGA oder ASIC - Einfluss auf die Länge des kritischen Pfades. Da RISC-V und ARM eine identische grundlegende architekturelle Basis haben - eine Load/Store RISC Architektur - ist anzunehmen, dass Hardwareimplementationen

keine wesentlichen Unterschiede für diesen Faktor zeigen würden. Daher wird für den Vergleich der Prozessoren die Betrachtung auf die benötigten Taktzyklen für ein gegebenes Programm vereinfacht:

$$\frac{ClockCycles}{Program} = \frac{Instructions}{Program} \cdot \frac{ClockCycles}{Instruction}$$

Für die meisten Vergleiche wird daher die Anzahl an Takten als Messwert verwendet. Nur ein Vergleich, der das Gesamtsystem betrachtet, verwendet nicht die Taktzyklen als Messwert, sondern die Echtzeit.

4.2 Entwicklung von Messmechanismen für benötigte Taktzyklen

Für die Messung der Leistungsfähigkeit beider Prozessoren sollen auf beiden Prozessoren Codeblöcke zur präzisen Messung von CPU-Zyklen entwickelt werden. Da die Prozessorarchitekturen verschiedene Mechanismen für solche Messungen bieten, sind die jeweiligen Codeblöcke für beide Prozessoren unterschiedlich.

4.2.1 Messmechanismen für RISC-V

Zähler für Prozessorzyklen

Als Teil der Grundspezifikation von RISC-V sind drei 64-Bit Zähler definiert:

- „cycle“: Vom Prozessor ausgeführte Takte
- „instret“: Vom Prozessor ausgeführte Instruktionen
- „time“: Ein Echtzeitzähler („wall-clock real time“)

Für den Zugriff auf die Zähler sind je zwei Assembler-Befehle pro Zähler definiert:

RDCYCLE	Lese die unteren 32 Bit des „cycle“-Zählers
RDCYCLEH	Lese die oberen 32 Bit des „cycle“-Zählers
RDINSTRET	Lese die unteren 32 Bit des „instret“-Zählers
RDINSTRETH	Lese die oberen 32 Bit des „instret“-Zählers
RDTIME	Lese die unteren 32 Bit des „time“-Zählers
RDTIMEH	Lese die oberen 32 Bit des „time“-Zählers

Tabelle 4.1: Befehle für Zugriff auf die Performance-Counter von RISC-V

Der Echtzeitzähler hat keine Relevanz für die Messungen, da der Prozessor während des Messvorgangs nicht gestoppt wird. Die „cycle“- und „instret“-Zähler bieten die nötigen Informationen für die geplanten Performance-Messungen. Der Vergleich von ausgeführten Takten und ausgeführten Instruktionen kann hierbei Informationen über teurere Instruktionen wie Speicherzugriffe und Ablaufsprünge liefern.

Speicherung von Messwerten

Für die Vermessung der Dauer eines zu testenden Codeabschnittes werden der Start- und Endzeitpunkt benötigt. Es werden nur die unteren 32 Bit der Zähler verwendet, da dies einen ausreichend großen Wertebereich für die in dieser Arbeit durchgeführten Messungen liefert.

Der RISC-V-Prozessor verfügt über 32 Register, von denen eins für die Konstante 0 verwendet wird. Die übrigen 31 Register sind frei für Software nutzbar. In der RISC-V Spezifikation sind für einige dieser Register spezielle Rollen definiert. Unter anderem sind die Register *x3* und *x4* vordefiniert - das Register *x3/gp* dient als „Global Pointer“, ein Pointer auf ein für ein Programm global verfügbares Objekt und das Register *x4/tp* dient als „Thread Pointer“, ein Pointer auf ein für einen aktiv laufenden Softwarethread verfügbares Objekt. Diese Register finden in Betriebssystemen Verwendung, für die vorliegende Embedded Anwendung entfallen diese Nutzungszwecke. Vom Compiler generierter Code verwendet diese Register nicht, daher bieten diese sich für das Abspeichern der Messwerte an.

Implementation des Messmechanismus

Der Messmechanismus wird als Assemblercode implementiert. Um ihn innerhalb von C Code verwenden zu können, werden Inline-Assembler Befehle vom gcc-Compiler verwendet. Diese erlauben es, Assemblerbefehle direkt in C zu verwenden. Für die erleichterte Verwendung werden die einzelnen Messbefehle in Makros eingekapselt.

Um eine Messung zu starten, wird das Makro __measure_init verwendet. Dies liest den aktuellen Wert des „cycle“-Zählers in das Register *x3/tp* mittels des RDCYCLE-Befehls:

```
1 // Initialise the timing measurement by setting a cycle reference point into
2 // the tp register
3 #define __measure_init            \
4     asm volatile("RDCYCLE tp")
```

Listing 4.1: Beginn einer Zyklenmessung für RISC-V

Um eine Messung zu beenden, wird das Makro __measure verwendet. Es berechnet die verstrichenen Zyklen seit dem Aufruf von __measure_init und speichert dies in das Register *x4/gp*:

```
1 // Measure the elapsed cycles since __measure_init and store it in the gp
2 // register. Subtract 1 to compensate for the cycle between the two
3 // RDCYCLE instructions.
4 #define __measure                 \
5     asm volatile(                 \
6             "RDCYCLE gp\r\n"      \
7             "sub gp, gp, tp\r\n"  \
8             "addi gp, gp, -1"     \
9     )
```

Listing 4.2: Ende einer Zyklenmessung für RISC-V

Das Auslesen eines Messwertes erfolgt über das Makro __measure_read(). Der im Register *x4/gp* liegende Messwert wird in eine Variable geschrieben:

```
1 // Read out the gp value into a provided variable
2 #define __measure_read(time)      \
3     asm volatile(                 \
4             "add %0, x0, gp"      \
5             : "=rm"(time)         \
6     )                             \
```

Listing 4.3: Auslesen einer Zyklenmessung für RISC-V

Um zu verhindern, dass derselbe Messwert ungewollt doppelt gelesen wird kann zusätzlich mit dem Makro __measure_delete das Register x4 auf 0 gesetzt werden:

```
1 // Delete the measurement
2 #define __measure_delete          \
3     asm volatile("addi gp, x0, 0");
```

Listing 4.4: Löschung einer Zyklenmessung für RISC-V

Für die Messung von durchgeführten Instruktionen wird der Befehl *RDCYCLE* durch den Befehl *RDINSTRET* ausgetauscht.

4.2.2 Messmechanismen für ARM

Zähler für Prozessorzyklen

Im ARM Cortex M4 wird in der „Data Watchpoint and Trace Unit“ (DWT) ein Zähler für die Anzahl verwendeter CPU Zyklen zur Verfügung gestellt. Dieser Zähler ist 32 Bit groß.

Speicherung von Messwerten

Der ARM Prozessor hat weniger Register zur Verfügung, als der RISC-V Prozessor. Die Speicherung von Messwerten in zwei dedizierten Registern ist somit nicht möglich. Stattdessen müssen die Messwerte im Speicher abgelegt werden. Hierdurch wird der Messmechanismus komplizierter, da der Zugriff auf eine Variable im Speicher mehr Aufwand erfordert als bei einem Register.

Bei Verwendung von Start- und Endwert muss zum Messungsbeginn erst aus der *DWT*-Einheit gelesen und anschließend der gelesene Wert abgespeichert werden. Da der Lesezugriff den Startzeitpunkt festlegt, würde ein Teil des Startvorgangs in die Messung hineinfließen. Dies kann vermieden werden, indem stattdessen die Messung durch Zurücksetzen des Zählers auf 0 gestartet wird.

Implementation des Messmechanismus

Bei Verwendung von Inline-Assembler-Befehlen hat der Compiler die geschrieben Instruktionen in der Reihenfolge umsortiert. Als Resultat konnte der Messmechanismus nicht taktgenaue Werte liefern. Zur Vereinfachung der Implementation wurde daher auf die Verwendung von Inline-Assembler-Befehlen verzichtet und stattdessen C Code verwendet. Als Resultat können die gemessenen Werte um einzelne Zyklen schwanken, abhängig von der Anordnung an Befehlen, die der Compiler produziert. Diese Ungenauigkeit zeigte sich jedoch als irrelevant für das Messverfahren, da die gemessenen Werte ausreichend groß waren, um den potenziellen Fehler von wenigen Zyklen in der Messung zu vernachlässigen.

```
1 static uint32 measure_result = 0;
2
3 #define __measure_init         (DWT_CYCCNT = 0)
4 #define __measure              (measure_result = DWT_CYCCNT)
5 #define __measure_read(time)   (time = measure_result)
6 #define __measure_delete       (measure_result = 0)
```

Listing 4.5: Zyklenmessung für ARM

4.2.3 Theoretischer Vergleich der Prozessorimplementationen

Der RISC-V-Prozessor ist zusammengesetzt aus der 32 Bit Integer Grundarchitektur („RV32I“) sowie Erweiterungen für Integer Multiplikationen und Divisionen (Erweiterung „M") und komprimierte Instruktionen (Erweiterung „C“). Er verfügt über eine simple Pipeline mit drei Stufen. Der ARM-Prozessor ist ein Cortex M4 Prozessor - er verfügt ebenfalls über eine Pipeline mit 3 Stufen, kann allerdings bei Sprunganweisungen spekulativ Instruktionen im voraus laden (*Prefetching*). Für mehrere aufeinander folgende voneinander unabhängige Speicherzugriffe kann der ARM-Prozessor die einzelnen Operationen in der Pipeline verschachtelt ausführen, um Zyklen zu sparen. Der RISC-V-Prozessor hingegen kann bei einer ungeraden Anzahl an aufeinander folgenden Zugriffen nur einen einzigen Zyklus für den gesamten Block an Speicherzugriffen einsparen.

4.3 Definition der zu vergleichenden Parameter

Für den Vergleich der Prozessoren soll kompilierter C Code verwendet werden, der auf Codegröße hin optimiert wurde (Compileroption: „-Os“). Dies ist eine häufig verwendete Konfiguration außerhalb von spezialisierten hochoptimierten Codeabschnitten und soll daher als Vergleichsgrundlage verwendet werden. Als Compiler wird für beide Prozessoren *gcc* verwendet. Der verwendete RISC-V Compiler hat die Version 7.2.0-4-20180606, der verwendete ARM Compiler hat die Version 6.3.1.20170215.

Für den Vergleich werden mehrere Kriteren zur genaueren Betrachtung definiert. Die Wahl der Kriteren basiert auf grundlegenden Operationen des Prozessors sowie geplanten Verwendungszwecken für das Projekt ROCINANTE.

4.3.1 Ablaufsteuerung

Für Ablaufsteuerung eines Programmes werden immer verschiedene Konstrukte verwendet. In der C Sprache werden hierfür die Anweisungen *if/elseif/else*, *switch/case*, *for*, *while*, *do while* sowie Funktionsaufrufe verwendet. Auf Prozessorebene werden hierfür Instruktionen für bedingte, unbedingte und indirekte Sprünge verwendet. Diese Instruktionen sind sowohl auf ARM- als auch auf RISC-V-Prozessoren implementiert. Zusätzlich bietet ARM eine Instruktion zur konditionalen Ausführung von Folgeinstruktionen (If-Then: *IT*) und zwei Instruktionen für das Springen mithilfe eines Arrays aus Sprungaddressen (Table Branch Byte: *TBB* und Table Branch Halfword: *TBH*) [3].

4.3.2 Bitlogik

Zur Steuerung von Hardwareperipherien werden oft Bitfelder in den Hardwareregistern verwendet. Für die Interaktion mit diesen Bitfeldern werden verschiedene Bitoperationen benötigt.

4.3.3 Regelungsalgorithmus

In der Motorsteuerung können oft Regelkreise verwendet werden. Während das Projekt ROCINANTE darauf hin zielt, die nötigen Algorithmen in Hardware zu repräsentieren,

ist die Hardwareimplementation jedoch oft auf Softwareimplementationen als Prototyp gestützt.

4.3.4 Kommunikation mit anderem System

Die Kommunikation mit anderen Geräten ist für eingebettete Applikationen essentiell. Bei diesem Vorgang ist nicht nur der Prozessor involviert, sondern auch die Hardwareperipherien des Prozessors, das verwendete Bussystem, sowie das System, mit dem kommuniziert werden soll. Als Folge dessen ist eine auf die Taktzyklen basierte Messung unpassend. Daher soll stattdessen die benötigte Echtzeit hierfür vermessen werden.

4.4 Auswahl der Vergleichsalgorithmen

Auf Basis der gewünschten Vergleichskriterien werden mehrere Algorithmen bestimmt, mit denen die Vergleiche durchgeführt werden sollen:

4.4.1 Zyklische Redundanzprüfung als Softwareimplementation

Die zyklische Redundanzprüfung ist ein Algorithmus zur Fehlerprüfung von Daten. Sie ist sehr leicht in Hardware implementierbar und findet daher in verschiedenen Bussystemen wie USB oder CAN Verwendung. Bei einer Berechnung in Software ist eine bitweise Iteration über die zu prüfenden Daten notwendig. ALs Resultat liegt der Schwerpunkt des Algorithmus auf Bitlogik und Ablaufsteuerung.

Eine portable optimierte Implementierung für die Berechnung von zyklischen Redundanzprüfungen liegt bereits vor und kann daher leicht als Vergleichsalgorithmus auf mehreren Prozessoren verwendet werden.

4.4.2 Field Oriented Control

Für die Steuerung eines PMSM-Motors verwendet Trinamic die sogenannte Field Oriented Control (kurz: FOC). Während ältere Produkte diesen Regelkreis als Softwareimplementation realisierten, ist mittlerweile die komplette Regelung als Hardwareimplementation verfügbar. Zum Zweck des Prozessorvergleichs soll der alte Algorithmus für

FOC verwendet werden. Der Algorithmus kombiniert Regelkreise mit Matrixmultiplikationen und sollte daher gute Aussagen zur Leistungsfähigkeit der Datenverarbeitung ermöglichen.

4.4.3 Algorithmen nach Clark und Park

Teil des Algorithmus für die Field Oriented Control basiert auf den Algorithmen nach Clark und Park. Beide dieser Algorithmen basieren auf Matrixmultiplikationen und sollen als einzelne Blöcke genauer betrachtet werden. Hierfür wird eine neue Implementation verwendet werden, um die Algorithmen als isolierte Blöcke zu betrachten. Die Algorithmen in der existierenden FOC-Implementation sind für solch eine isolierte Betrachtung zu sehr in den Rest des FOC-Algorithmus integriert.

4.4.4 Trinamic Motion Control Language

Für die Kommunikation zwischen einem PC und Mikrocontrollern zur Motorsteuerung nutzt Trinamic die eignens entwickelte *Trinamic Motion Control Language*, kurz *TMCL* [20]. Es handelt sich um ein simples Protokoll zum Senden von Befehlen an das Mikrocontrollersystem. Prinzipiell kann es über eine beliebige Schnittstelle übertragen werden, typischerweise wird es über USB oder CAN verwendet. Für den Vergleich soll die Verbindung über USB genutzt werden. Da dieses Protokoll in dem Projekt ROCINANTE vorraussichtlich verwendet wird, soll die Leistungsfähigkeit des Protokolls getestet werden.

5 Durchführung des Prozessorvergleichs

5.1 Messung der Leistungsfähigkeit mittels der zyklischen Redundanzprüfung

Die Zyklische Redundanzprüfung (engl.: Cyclic Redundancy Check, kurz CRC) ist ein Algorithmus zur Erkennung von Datenfehlern. Eine Implementation in Hardware ist sehr einfach und eignet sich besonders für serielle Daten [6]. Er findet daher oft Verwendung in der Übertragung von Daten mittels Bussystemen. So findet er beispielsweise Verwendung in USB, CAN oder Ethernet.

Für die Berechnung eines CRC sind zwei Werte nötig - die verwendete Bitlänge und ein sogenanntes Generator-Polynom. Die Bitlänge gibt an, wie lang die resultierende Prüfsumme ist. Das Generatorpolynom eines CRC der Bitlänge N hat den folgenden Aufbau:

$$x^N + \sum_{n=0}^{N-1} a_n \cdot x^n, \quad a_i \in [0, 1]$$

Verschiedene Polynome sind unterschiedlich gut für die Datensicherung geeignet.

In der Praxis gibt es mehrere Varianten des CRC-Algorithmus, die sich durch die Bitordnung der zu prüfenden Daten unterscheiden. Normalerweise wird vom höchsten zum geringsten Bit hin gearbeitet - wird diese Reihenfolge umgedreht, beizeichnet man dies als reflektierten CRC. Zusätzlich können je ein Start- und Endwert defininiert werden, welche zusätzlich über Exklusiv-Oder-Verknüpfung in den Datenstrom gemischt werden.

5.1.1 Der Algorithmus

Die Berechnung eines CRCs besteht aus den folgenden Schritten:

1. Wandle das Polynom in eine binäre Representation um. Diese Repräsentation ist $N + 1$ Bit lang und hat immer eine 1 im höchsten Bit. Die darauf folgenden Bits entsprechen den Koeffizienten a_i des Polynoms. Beispielsweise ergibt das Polynom $x^5 + x^2 + 1$ dem Binärwert $100101_{bin} = 25_{hex}$.

2. Schreibe die zu prüfenden Daten in Binärform und ergänze N auf 0 gesetze Bits hinter den Daten.

3. Schreibe das Polynom in Binärform unter die Daten, sodass die erste 1 in den Daten auf gleicher Höhe ist, wie die erste 1 im Polynom.

4. Bilde die Verknüpfung mittels binärem Exklusiv-Oder zwischen Daten und Polynom.

5. Wenn alle Bits der Daten 0 sind, steht in den hinten ergänzten Bits das Ergebnis - die Prüfsumme. Die Berechnung ist dann abgeschlossen. Wenn dies noch nicht der Fall ist, wiederhole Schritt 3 und 4.

Beispielsweise ergeben die Daten 01000000_{bin} bei Verwendung des Generatorpolynoms $x^8 + x^2 + x + 1$ den CRC8-Wert 11000111_{bin}:

Daten:	01000000															
Polynom:	$x^8 + x^2 + x + 1 \rightarrow 100000111$															
	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		1	0	0	0	0	0	1	1	1						
		0	0	0	0	0	0	1	1	1						
								↓	↓	↓	↓	↓	↓	↓	↓	↓
								1	1	1	0	0	0	0	0	0
								1	0	0	0	0	0	1	1	1
								0	1	1	0	0	0	1	1	1
CRC:	11000111															

Tabelle 5.1: Beispiel für die Berechnung einer zyklischen Redundanzprüfung

Bei der Berechnung eines reflektierten CRCs werden erst die Daten vor Beginn der Berechnung und nach der Berechnung die Prüfsumme jeweils in der Bitreihenfolge umgedreht.

Anpassung des Algorithmus für Softwareimplementation

Um den CRC in Software zu implementieren, kann der Ablauf des Algorithmus angepasst werden. Um das höchste gesetze Bit in den Daten zu finden, können die Daten durch die Bitshift-Operation nach links geschoben werden. Anschließend kann an einer fixen Bitposition geprüft werden, ob ein Bit gesetzt ist. Wenn das geprüfte Bit gesetzt ist, wird das Polynom an der passenden Position per Exklusiv-Oder-Verknüpfung in die Daten gebracht. Dieser Vorgang muss so oft durchgeführt werden, wie es Datenbits zu prüfen gibt. Diese Modifikation des Algorithmus sorgt dafür, dass das Polynom nicht mehr an die richtige Position geschoben werden muss, da die Daten stattdessen an die richtige Position geschoben werden.

Für jeden Berechnungsschritt hat nur das vorderste Bit einen Einfluss auf die CRC-Berechnung. Die Bits dahinter haben zu diesem Zeitpunkt noch keine Wirkung. Diese Eigenschaft ermöglicht es, die zu prüfenden Daten in einzelne Blöcke zu unterteilen und jeden Block nacheinander zu bearbeiten. Werden zum Beispiel 16 Bit kontrolliert, so können zunächst nur die vorderen acht Bits bzw. das erste Byte verwendet werden. Während der Berechnung werden zunächst nicht die hinteren acht Bits nachgerückt, sondern 0-Bits. Sobald der Vorgang achtmal durchgeführt ist, wurde das vordere Byte komplett verarbeitet und der nächste Vorgang wird Bits des hinteren Bytes verwenden. Zu diesem Zeitpunkt kann dann das gesamte hintere Byte in die Daten importiert werden, indem es durch Exklusiv-Oder-Verknüpfung mit dem Zwischenergebnis kombiniert wird.

Da die Berechnung eines CRC sehr viele einzelne Bitoperationen erfordert, ist eine Berechnung in Software langsam. Die Einteilung in Blöcke erlaubt es jedoch, eine effizientere Software-Implementation zu bauen. Für jeden möglichen Byte-Wert kann der dazugehörige CRC-Wert einmalig im Vorraus berechnet werden und gespeichert werden. Dadurch vereinfacht sich der Algorithmus:

1. Das erste Byte der Daten wird eingelesen.
2. Für den aktuellen Datenwert wird der dazugehörige CRC-Wert aus der vorgenerierten Tabelle entnommen.
3. Wenn keine weiteren Bytes an Daten vorhanden sind, ist der aus der Tabelle entnommene Wert die CRC-Prüfsumme der gesamten Daten.
4. Ansonsten wird das nächste Byte der Daten genommen und mit dem aus der Tabelle gelesenen CRC-Wert mit Exklusiv-Oder verknüpft.

5. Wiederhole ab Schritt 2.

Eine solche vorgenerierte Tabelle ist nur für ein bestimmtes gegebenes Polynom und einen CRC-Modus (Normal, Reflektiert) gültig. Diese Tabelle wird durch den normalen Algorithmus generiert. Für eine Softwareimplementation kann dies zu Beginn des Programmes geschehen oder eine bereits generierte Tabelle kann in das Programm integriert werden.

5.1.2 Die Implementation

Für die Leistungsmessung wird eine existierende Implementation für das Generieren einer CRC8 Lookup-Tabelle verwendet. Hierbei wird für jeden Bytewert von 0 bis 255 der dazugehörige CRC-Wert mit einem gegebenen Polynom berechnet und in eine Tabelle eingetragen. Ein kompletter Durchlauf führt somit 256 CRC8 Berechnungen durch.

Optional kann die Tabelle auch für einen reflektierten CRC berechnet werden. Hierfür wird jeweils vor und nach der Berechnung des CRCs das Datenbyte umgedreht. Hierbei werden die 8 Bits des Datums in invertierter Reihenfolge genommen.

Die existierende Implementation ist optimiert worden - anstatt jeden Wert einzeln zu berechnen werden vier Werte auf einmal berechnet, indem vier einzelne Bytes in einer 32-Bit Variable zusammengepackt sind. Dies reduziert die Anzahl an Schleifendurchläufen um den Faktor 4 und erlaubt es, die Links-Shift-Operation auf 4 Bytes und somit 4 Berechnnungen gleichzeitig anzuwenden.

Die für den reflektierten CRC nötige Drehung der Datenbits erfolgt in einer Hilfsfunktion. Durch das Komprimieren von vier Datenwerten in eine 32-Bit Variable, müssen somit die Bits jedes Bytes separat gedreht werden.

Die Drehung der Bits erfolgt in drei Stufen. Zuerst werden benachbarte Nibble-Paare, anschließend 2-Bit-Paare und zuletzt Bit-Paare in ihren Position ausgetauscht [7]. In der Implementation geschieht jeder dieser Tauschvorgänge folgendermaßen:

1. Schiebe die oberen Hälften der Paare durch einen Bitshift an die Positionen der unteren Hälften. Die Anzahl an zu schiebenden Bits entspricht hierbei der Anzahl an Bits in den Einzelblöcken - Beim Tauschen der Nibble-Paare wird um 4 Bit, bei den 2-Bit-Paaren wird um 2 und bei den Bit-Paaren um 1 Bit verschoben.

2. Maskiere je das originale und das verschobene Datum mit einer Bitmaske, um die unteren Hälften der Paare zu extrahieren. Für die Nibble-Paare ist diese Bitmaske der Wert 0x0F0F0F0F, für die 2-Bit-Paare der Wert 0x33333333 und für die Bit-Paare der Wert 0x55555555.

3. Schiebe das maskierte originale Datum nach links, sodass die unteren Hälften der Paare an den Positionen der oberen Paare sind.

4. Verknüpfe die beiden Zwischenergebnisse.

5.1.3 Betrachtete Bewertungskriterien

Kontrollfluss

Bei der Berechnung des CRC wird für jedes Bit überprüft, ob das Polynom mit Exklusiv-Oder in die Daten gemischt werden soll, oder nicht. In der existierenden Implementation erfolgt diese Überprüfung mittels eines Branch-Befehls. Zusätzlich werden die Bits in einer Schleife abgearbeitet. Dies resultiert in zwei Branch-Befehlen pro berechnetem Bit oder 16 Branch-Befehlen pro berechnetem Byte.

Die Berechnung jedes möglichen Bytewertes resultiert zusätzlich darin, dass jede mögliche Kombination der einzelnen Programmverzweigungen für die bedingten Sprünge, die das Exklusiv-Oder mit dem Polynom steuern, unabhängig vom verwendeten Polynom vorkommt. So würde bei dem gegebenen Polynom $0x07$ der Bytewert $0x83$ beim ersten Branch springen und bei den folgenden sieben Branches nicht springen. Der Wert $0x00$ hingegen würde beim ersten Branch nicht springen und ebenfalls bei den folgenden sieben Branches nicht springen.

Bitlogik

Die Prüfung, ob das durch die Shift-Operation entfernte Bit eine 1 oder 0 ist, erfordert Bitlogik. Durch die Optimierung des Algorithmus werden Bits an 4 verschiedenen Positionen kontrolliert. Das oberste Bit des obersten Bytes wird durch die Shift-Operation entfernt und sein Zustand wird somit vor der Shift-Operation an der Position $0x80000000$ kontrolliert. Die obersten Bits der anderen 3 Bytes können nach der Shift-Operation kontrolliert werden - an den Bitpositionen $0x01000000$, $0x00010000$ und $0x00000100$.

Für das Berechnen eines Reflektiert-CRC werden die Bits eines Bytes vor und nach der CRC-Berechnung umgedreht

Durch eine Anpassung der Implementation können die Branches, die die Exklusiv-Oder Verknüpfung des Polynoms kontrollieren durch eine Und-Verknüpfung von Polynom und einem Maskenwert von 0x00000000 bzw. 0xFFFFFFFF ersetzt werden. Je nach gewählem Wert wird so entweder das Polynom oder dem Wert 0 mit Exklusiv-Oder verknüpft. Der Maskenwert 0 entspricht hierbei dem Überspringen der Exklusiv-Oder-Verknüpfung bei der Branch-Variante. Die Auswahl der korrekten Maske kann ebenfalls mit Bitlogik gelöst werden, sodass die Branches komplett entfernt werden können. Dies geschieht durch die Kombination der Links-Bitshift-Operation und der arithmetischen Rechts-Bitshift-Operation. Das zu kontrollierende Bit wird mithilfe von dem Links-Bitshift-Operator an die MSB-Position geschoben. Anschließend wird um 31 Bit arithmetisch nach rechts geschoben. Dies resultiert in dem Wert 0x00000000, wenn das kontrollierte Bit 0 war und dem Wert 0xFFFFFFFF, wenn das kontrollierte Bit 1 war. In C kann dies bei Verwendung von implementationsdefiniertem Verhalten implementiert werden:

```
1
2 // Check bit [n] in [data], set mask to 0xFFFFFFFF if it's 1, to 0 otherwise.
3 // Both mask and data are 32-Bit unsigned integers.
4 // Valid C code:
5 mask = (data & (1<<n))? 0xFFFFFFFF : 0x00000000; // Utilizing implementation
6 defined behaviour // (Right-Shift of signed integers sign-extends):
7 mask = ((int32_t) (data << (31-n))) >> 31;
```

5.1.4 Messergebnisse

Für alle Messungen wurde das Polynom $0x07$ verwendet.

CRC Variante	Implementation	RISC-V		ARM
		Zyklen	Instruktionen	Zyklen
Normal	Branching	12175	10166	11702
	Bitlogik	11593	10161	8683
Reflektiert	Branching	16015	13878	13935
	Bitlogik	15433	13873	10841

Tabelle 5.2: Messergebnisse der Zyklischen Redundanzprüfung

5.1.5 Auswertung der Messergebnisse

Analyse der Implementation

Die Implementation des CRC-Algorithmus verwendet zwei Schleifen. Die äußere Schleife iteriert über alle 256 Bytes, wobei durch Optimierungen vier Bytes pro Iteration verarbeitet werden. Somit hat die äußere Schleife 64 Iterationen. Die innere Schleife iteriert über die acht Bits eines Bytes - durch die Vernestung in der äußeren Schleife läuft die innere Schleife somit mit insgesamt 512 Iterationen.

Für einen reflektierten CRC wird zusätzlich zu Beginn und Ende der äußeren Schleife eine Hilfsfunktion zum Invertieren der Bitreihenfolge aufgerufen. Im Verlaufe des gesamten Algorithmus erfährt diese Hilfsfunktion somit 128 Aufrufe. Im Falle des normalen CRC werden die Aufrufe der Hilfsfunktion durch konditionale Sprünge ausgelassen.

Analyse des generierten Maschinencodes

Der erste Teil der Analyse befasst sich mit dem ersten Term der „Iron Law“ - den Instruktionen pro Programm. Hierfür wird der vom Compiler generierte Assemblercode für alle Implementationen ausgewertet und auf die Kriterien Compiler-Qualität und Möglichkeiten der ISA hin bewertet.

Die gesamte Testfunktion wird in 6 Bereiche unterteilt:

- Funktionsaufruf (N_{Call}): Die für den Aufruf der Funktion nötigen Instruktionen. Dies beinhaltet Präparation der Funktionsargumente sowie der Funktionsaufruf selbst.
- Prolog (N_{Prolog}): Der Anfang der CRC-Funktion. Hier werden Register gespeichert, Parameter auf Korrektheit geprüft und Vorbereitung für die darauf folgende Schleife durchgeführt.
- Äußere Schleife ($N_{outerLoop}$): Die äußere Schleife der CRC-Berechnung. Diese iteriert über alle 256 möglichen Bytewerte.
- Innere Schleife ($N_{innerLoop}$): Die innere Schleife der CRC-Berechnung. Diese iteriert über die 8 Bits eines Bytes und führt die CRC-Berechnung durch.

- Epilog (N_{Epilog}): Das Ende der CRC-Funktion. Hier werden im Prolog gespeicherte Register wiederhergestellt und von der Funktion zurückgekehrt.
- Hilfsfunktion (N_{helper}): Eine Hilfsfunktion für das Invertieren der Bits in jedem Byte eines 32-Bit Wertes. Diese wird für den reflektierten CRC benötigt.

Die Analyse der Instruktionszahlen in jedem Programmabschnitt ergibt folgende Werte:

Programmabschnitt	RISC-V		ARM	
	Branching	Bitlogik	Branching	Bitlogik
Funktionsaufruf	4	5	3	4
Prolog	35	31	17	17
Äußere Schleife	14, 16	14, 16	15, 17	15, 17
Innere Schleife	16-20	18	20	13
Epilog	15	13	2	2
Hilfsfunktion	28	28	16	16

Tabelle 5.3: Aufteilung der Instruktionen nach Programmabschnitt von der Zyklischen Redundanzprüfung

Für den normalen CRC sind bei allen Varianten in der äußeren Schleife zwei Instruktionen weniger nötig, als für den reflektierten CRC. Die innere Schleife der auf Branching basierten Implementation für RISC-V nutzt zwischen 16 und 20 Instruktionen. Aufgrund der Eigenschaften des CRC werden bei Berechnung aller 256 möglichen Datenwerte eines Bytes im Durchschnitt genau 18 Instruktionen benötigt.

Die Anzahl an Instruktionen zum Aufruf der CRC Testfunktion hängen primär von der Strukturierung um den Funktionsaufruf ab. Dies ist für den Vergleich irrelevant.

Wesentliche Unterschiede zwischen RISC-V und ARM sind im Prolog und Epilog sichtbar. Hier verwendet ARM die Instruktionen *Load Multiple (LDM)* und *Store Multiple (STM)* für das Speichern und Wiederherstellen von Registern mittels des Stacks. RISC-V muss hierfür pro Register je eine Instruktion für das Speichern und Wiederherstellen verwenden.

In der äußeren Schleife benötigt ARM eine Instruktion mehr für das Inkrementieren und anschließende Prüfen von der for-Schleifenvariable. Ansonsten ist dieser Codeabschnitt für beide Prozessoren nahezu identisch.

In der inneren Schleife variieren die Unterschiede je nach Implementationsart. Bei der Branching Implementation nutzt RISC-V konditionale Sprünge während ARM die Instruktion *If-Then (IT)* nutzt, um darauf folgende Instruktionen konditional auszuführen. Der konditionale Sprung von RISC-V überspringt somit Instruktionen während ARM die betroffenen Instruktionen als *NOP* behandelt. Dies resultiert darin, dass RISC-V durchschnittlich zwei Instruktionen weniger pro Iteration durchführt.

Bei der Bitlogik Implementation gewinnt ARM in der inneren Schleife enorm durch bessere Instruktionen für das Prüfen einzelner Bits. ARM kann hier das Generieren einer Maske abhängig von einem Bit mit einer einzigen Instruktion durchführen - *Signed Bitfield Extract (SBFX)*. Diese Instruktion nimmt ein Bitfeld aus einem Register und interpretiert diesen Wert als vorzeichenbehaftete Zahl im Zweierkomplement. Ein ein Bit breites Bitfeld kann somit die Werte 0 oder -1 annehmnen. Diese Bitfeldwerte resultieren in dem extrahierten Wert 0x00000000 bzw. 0xFFFFFFFF - welche genau den gewünschten Masken entsprechen. Hingegen muss RISC-V für das Generieren der Maske zwei Instruktionen verwenden. Erst wird das zu prüfende Bit durch einen Bitshift nach links in die MSB-Position verschoben. Danach wird ein Arithmetischer Bitshift um 31 Bit nach rechts verwendet, um die Maskenwerte abhängig vom Wert des Bits zu generieren. Zusätzlich kann ARM das Verschieben um ein Bit nach links in eine andere Instruktion integrieren. Dies geschieht durch die Verwendung des Barrel-Shifters. Zusammen mit der Einsparung je einer Instruktion relativ zu RISC-V bei der Bitprüfung ergibt sich die Differenz von 5 Instruktionen in der inneren Schleife.

Die Hilfsfunktion nutzt mehrere Bitmasken, um die Bitreihenfolgen zu invertieren. Das Austauschen benachbarter Nibbles nutzt die Maske 0x0F0F0F0F die 2-Bit Paare nutzt die Maske 0x33333333 und die 1-Bit Paare nutzen die Maske 0x55555555. Zum Generieren dieser Masken kann ARM die Instruktion *AND* direkt verwenden. Hierbei wird für den zweiten Operand eine 8-Bit Konstante viermal wiederholt, um die korrekte Maske direkt zu generieren [2]. Bei RISC-V sind die Konstanten für die *AND* Instruktion auf vorzeichenerweiterte 12-Bit Werte limitert. Dieses Verfahren erlaubt es nicht, die entsprechenden Bitmasken direkt zu generieren. Stattdessen müssen diese über das Instruktionspaar *Load upper immediate (LUI)* und *Add immediate value (ADDI)* generiert werden. Zusätzlich wird vom Compiler die Berechnung über zwei Masken pro zu tauschen-

den Paaren verwendet. Somit müssen für die drei Tauschpaare insgesamt sechs Masken mit je zwei Instruktionen pro Maske generiert werden. Dies resultiert in den 12 zusätzlichen Instruktionen für die RISC-V Implementation. Wenn der Compiler die Berechnung über eine Maske pro Paar verwenden, so würden nur 6 zusätzliche Instruktionen benötigt werden.

Aus den Instruktionszahlen lässt sich die Anzahl der durchgeführten Instruktionen für den normalen und reflektierten CRC berechnen:

$$N_{Total,Normal} = N_{Call} + N_{Prolog} + 64 \cdot N_{outerLoop} + 512 \cdot N_{innerLoop} + N_{Epilog}$$

$$N_{Total,Reflektiert} = N_{Call} + N_{Prolog} + 64 \cdot (N_{outerLoop} + 2 \cdot N_{helper}) + 512 \cdot N_{innerLoop} + N_{Epilog}$$

Diese Berechnungen ergeben folgende Instruktionszahlen:

CRC Variante	RISC-V		ARM	
	Branching	Bitlogik	Branching	Bitlogik
Normal	10166	10161	11222	7639
Reflektiert	13878	13873	13398	9815

Tabelle 5.4: Theoretisch berechnete Instruktionszahlen von der Zyklischen Redundanzprüfung

Der Vergleich der theoretisch berechneten Werte mit den für RISC-V gemessenen durchgeführten Instruktionen zeigt, dass die theoretische Betrachtung der Instruktionszahle dieselben Werte ergibt, wie die Messung. Somit bestätigt die Messung die theoretischen Berechnungen.

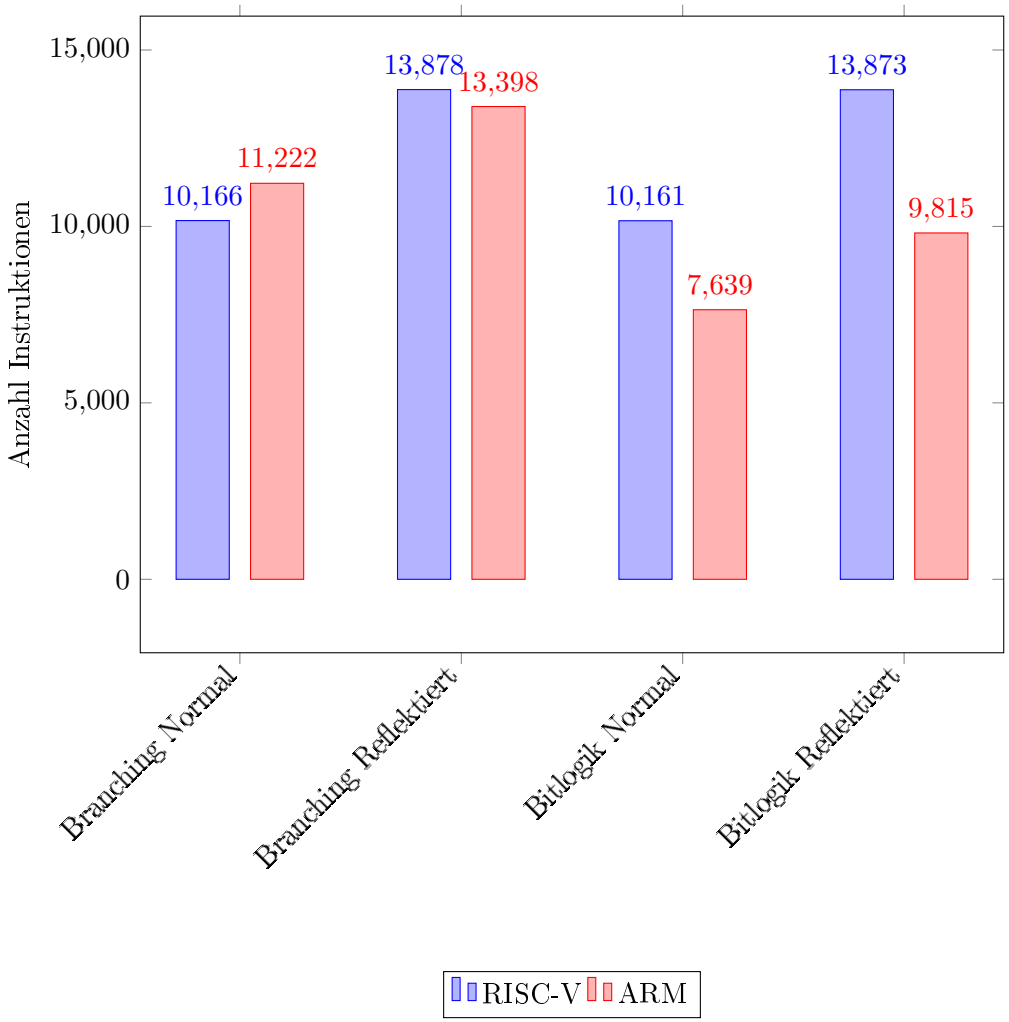

Abbildung 5.1: Illustration der Instruktionszahlen von der Zyklischen Redundanzprüfung

Bei Betrachtung der Instruktionszahlen liegt RISC-V für die Branching Implementation für den normalen CRC noch mit 9.5% weniger Instruktionen vor ARM. Beim Wechsel zum reflektierten CRC wird dieser Vorsprung von RISC-V jedoch schon durch die effizientere Hilfsfunktion von ARM ausgeglichen. RISC-V benötigt für einen reflektierten CRC im Vergleich zum normalen CRC genau 3712 weitere Instruktionen - eine Steigerung um 36.5% für Branching und Bitlogik. ARM benötigt hingegen nur 2176 weitere Instruktionen - 19.4% für die Branching-Implementation und 28.5% für die Bitlogik-Implementation. Als Resultat fällt RISC-V von 9.5% Vorsprung zurück auf 3.5% mehr Instruktionen als ARM - ein Unterschied von 12.0%.

Für die Bitlogik-Implementation gewinnt ARM enorm durch besser geeignete Befehle für die durchzuführenden Berechnungen in der inneren Programmschleife. Für einen normalen CRC benötigt RISC-V somit 33.0% mehr Instruktionen, für einen reflektierten CRC sogar 41.3% mehr Instruktionen.

Analyse der Laufzeit

Im zweiten Teil der Analyse wird der zweite Term der „Iron Law“ betrachtet - Die Zyklen pro Instruktion. Da keiner der beiden Prozessoren superskalar arbeitet, reduziert sich die Betrachtung auf Befehle, die nicht genau einen Zyklus benötigen. Für beide Prozessoren werden zusätzliche Zyklen für Speicherzugriffe sowie Instruktionen für die Programmablaufsteuerung benötigt. Zusätzlich kann ARM die *If-then-else (IT)* Instruktion in den Taktzyklus der vorherigen Instruktion integrieren, sofern diese eine 16-Bit Instruktion ist.

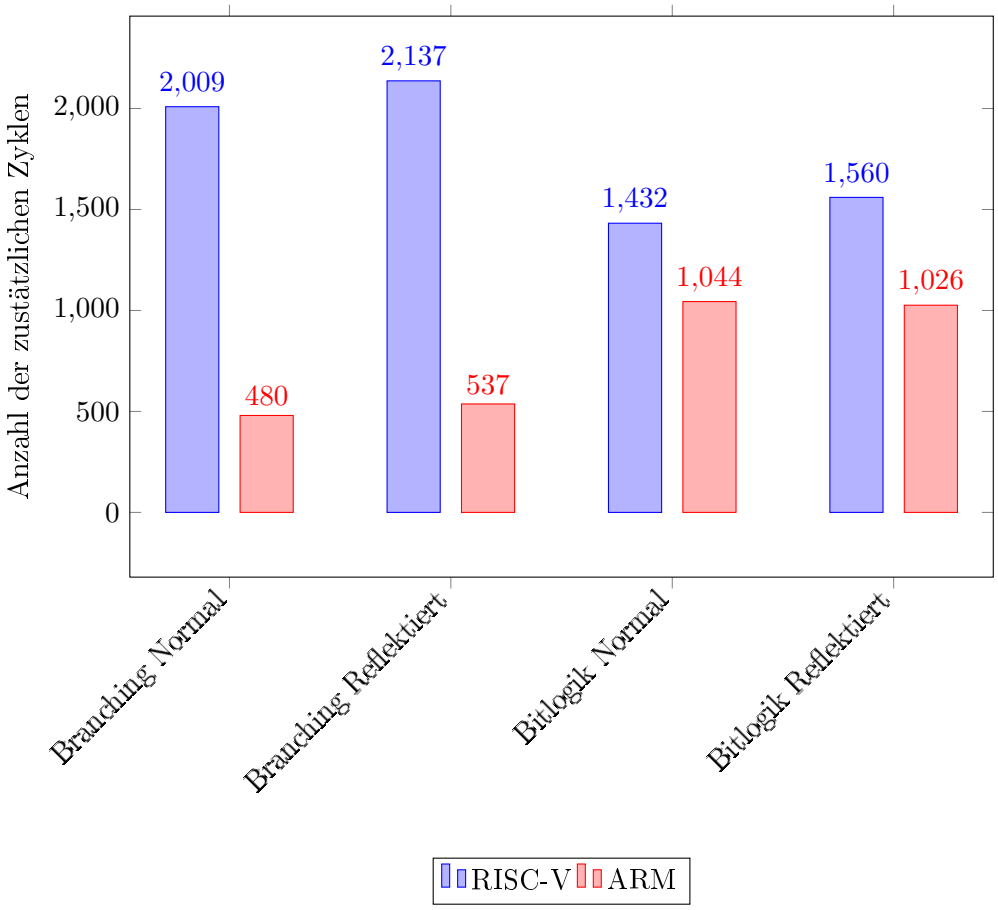

Abbildung 5.2: Illustration der zusätzlichen Taktzyklen für die Zyklische Redundanzprüfung

Es werden die zusätzlich benötigten Zyklen betrachtet - die Differenz von Zyklen und Instruktionen. Hier ist ein direkter Unterschied zwischen den Prozessoren sichtbar: Bei der Branching Implementation schneidet RISC-V schlechter ab mit ca. viermal so vielen zusätzlichen Zyklen wie ARM. Für die Bitlogik Implementation reduziert sich dieser Unterschied auf ca. 50% zusätzliche Zyklen für RISC-V im Vergleich zu ARM.

In der Branching Implementation verbraucht RISC-V viele Zyklen aufgrund von konditionalen Sprüngen in der inneren Schleife. Im Durchschnitt werden hier zwei Sprünge pro Iteration durchgeführt und somit 1024 zusätzliche Zyklen verwendet. Diese zusätzlichen Zyklen fallen in der Bitlogik Implementation nicht an. Jedoch verliert die Bitlogik Implementation 448 Zyklen dadurch, dass im Gegensatz zur Branching Implementation

der Beginn der inneren Schleife nicht an einer 32-Bit Grenze im Speicher positioniert ist und die erste Instruktion 32 Bit groß ist. Dies sorgt für einen weiteren verbrauchten Taktzyklus für jede Wiederholung der inneren Schleife.

ARM kann in der Branching Implementation den Zyklus einer *If-then-else (IT)* Instruktion in der inneren Schleife einsparen. Die Bitlogik Implementation verliert dieser Ersparnis. Die Kosten für Schleifenlogik und Speicherzugriffe bleiben zwischen den beiden Implementationen identisch. Als Resultat sind in der Bitlogik Implementation mehr zusätzliche Zyklen als in der Branching Implementation.

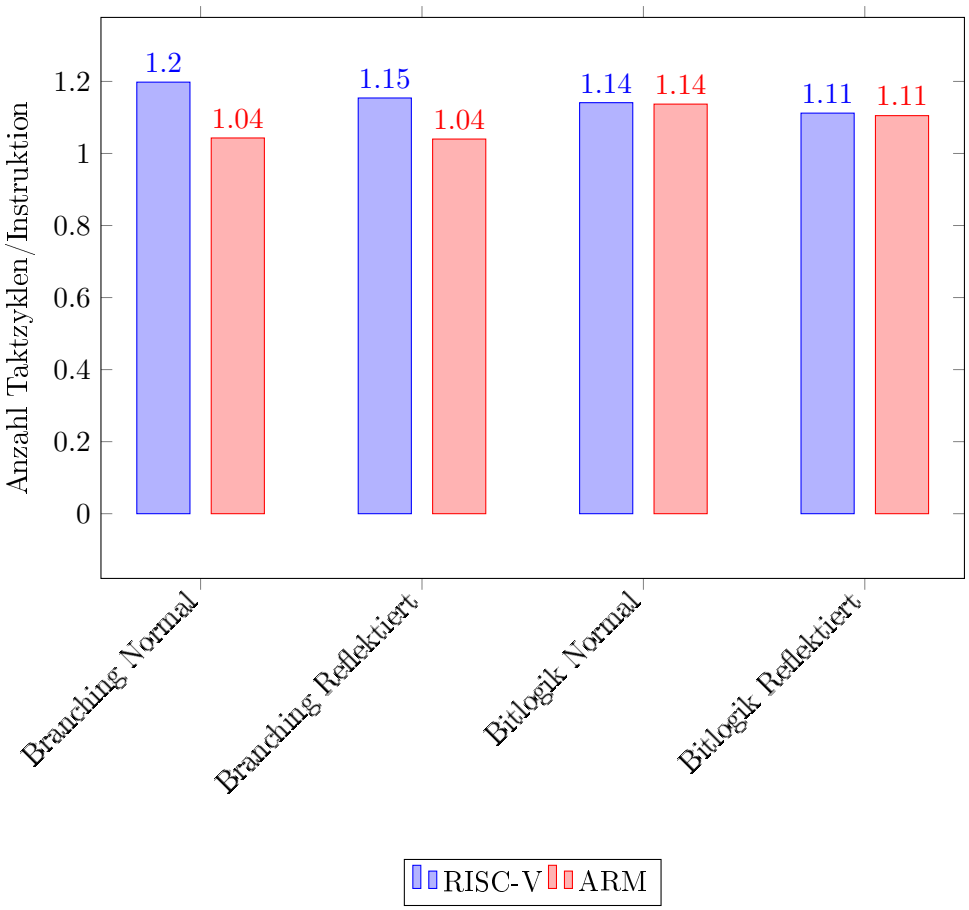

Abbildung 5.3: Durchschnittliche Takte pro Instruktion für die Zyklische Redundanzprüfung

Die Betrachtung der durchschnittlichen Taktzyklen pro Instruktion zeigt einen wesentlichen Unterschied zwischen RISC-V und ARM für die Branching Implementation. Hier

spart ARM durch das konditionale Ausführen von Instruktionen durch die *If-then-else (IT)* Instruktion zusätzliche Zykluskosten und kann teilweise diese Instruktion mit effektiv 0 Zyklen ausführen. Allerdings zeigt die Bitlogik Implementation, dass der ARM Prozessor trotz eines vorhandenen Branch Predictors keinen großen Vorsprung gegenüber dem RISC-V Prozessor für die Schleifenlogik hat.

5.2 Field Oriented Control

5.2.1 Der Algorithmus

Die *Field Oriented Control* (kurz *FOC*) ist ein Regelungsansatz für die Steuerung von dreiphasigen Elektromotoren. Bei diesen Motoren sind drei Spulen im äußeren Teil (Stator) des Motors fixiert - mit jeweils 120° Versatz zwischen ihnen. Die Krafteinwirkung jeder Spule auf die im Rotor verbauten Permanentmagneten sind somit ebenfalls um 120° versetzt. Die drei Spulen sind sternförmig miteinander verbunden. Als Resultat ist die Summe der drei Ströme 0. Dies ermöglicht es, nur zwei Ströme zu messen und den dritten zu berechnen.

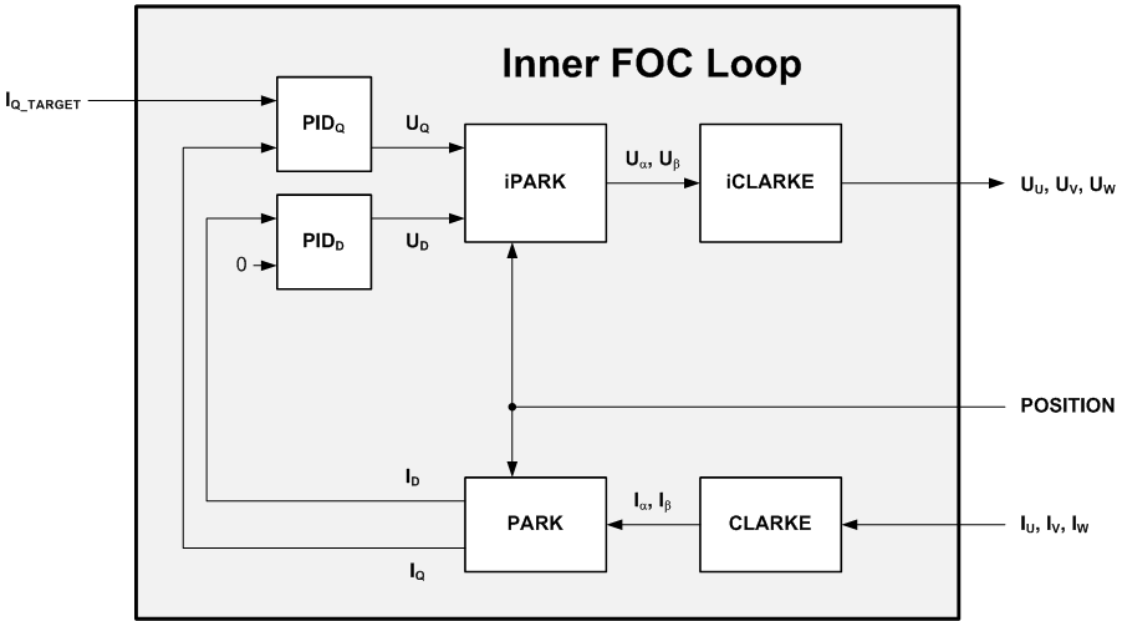

Abbildung 5.4: Blockschaltbild der Field Oriented Control [23]

Die drei Phasenströme (I_U, I_V, I_W) können durch die *Clark-Transformation* in zwei zueinander orthogonale Ströme (I_α, I_β) umgewandelt werden. Darstellung dieser Ströme in der komplexen Ebene resultiert in einem Drehzeiger, dessen Rotation der Motorrotation entspricht. Um das Wechselstromverhalten dieser Ströme in einer Regelung zu verwenden, wird die Park-Transformation angewendet. Diese Transformation dreht ein zweidimensionales Koordinatensystem um einen Winkel θ um dem Ursprung herum. Wird der Winkel kontinuierlich verändert, kann aus dem rotierenden Stromzeiger ein nicht rotierender Stromzeiger berechnet werden. Dieser Stromzeiger setzt sich aus zwei zueinander orthogonalen Strömen zusammen - der Strom I_d repräsentiert die magnetische Flussdichte und der Strom I_q repräsentiert das Drehmoment des Motors.

Die Ströme I_d und I_q werden durch zwei PI-Regler geregelt. Der Winkel θ kann von außen gesteuert werden oder den gemessenen Winkel des Motors verwenden. Um die aus der Regelung resultierenden Stromwerte in Phasenstromwerte umzuwandeln, werden die inverse Park- und inverse Clark-Transformation verwendet. Die resultierenden Phasenströme werden eine analoge Treiberstufe, typischerweise durch Pulsweitenmodulation, an den Motor geleitet.

5.2.2 Die Implementation

Für die Messung wird eine bereits existierende Implementation für den FOC-Algorithmus verwendet. Die Implememention ist in zwei Teile getrennt - die Geschwindigkeitsregelung und die Stromregelung. Die Geschwindigkeitsregelung erfolgt alle 10 Durchläufe, die Stromregelung erfolgt in jedem Durchlauf. Die Geschwindigkeitsregelung verfügt über *Position*-, *Velocity*- und *Torque*-Betriebsmodi - Position, Geschwindigkeit oder Drehmoment können als Zielparameter verwendet werden. Die Stromregelung verwendet die von der Geschwindigkeitsregelung generierte Geschwindigkeit, um die Berechnung der FOC durchzuführen. Der in der Park-Transformation verwendete Winkel θ kann von außen gesteuert oder von Hall- oder Encoder-Sensoren ermittelt werden. Die Phasenströme werden über Analog-Digital-Wandler gemessen.

Für den Leistungsvergleich der Prozessoren wird der *Velocity*-Modus verwendet. Da kein physikalischer Aufbau vorhanden ist, werden statt analoger Treiberstufe, Motor und Analog-Digital-Wandler die Ausgangswerte der FOC als Eingangswerte verwendet. Dies erlaubt es, das System ohne die vorhandene Leistungselektronik zu testen. Da die vorliegende Implementation Teil eines größeren Mikrocontroller-Projektes ist, wird die

Implementation für die portable Verwendung vereinfacht - neben den Hardwarezugriffe werden Funktionen für Debuggen während der Laufzeit sowie nicht portable Elemente der nicht verwendeten Betriebsmodi entfernt. Es werden Hilfsfunktionen für das Schreiben der Eingangswerte und Auslesen aller Zwischen- und Endergebnisse der Berechnungen eingebaut.

5.2.3 Betrachtete Bewertungskriterien

Bei dem Vergleich für diese Implementation soll im Gegensatz zu den vorherigen Vergleichen nicht eine vollständige Analyse der generierten Assembler-Instruktionen durchgeführt werden. Stattdessen sollen lokale Vergleiche für verschiedene Teile des Gesamtsystemes geschehen:

- Die Bearbeitung von mehreren Variablen in Folge - bei der Initialisierung.
- Die Umsetzung von Strukturen, dessen Elemente nicht 32 Bit - die Wortlänge beider Prozessoren - groß sind.
- Weitere Mängel bei der Codegenerierung des Compilers - sofern vorhanden.

Die Gesamtleistungsfähigkeit der Systeme soll somit zwar betrachtet, jedoch nicht vollständig analysiert werden.

5.2.4 Messergebnisse

Durchgeführte Regelung	RISC-V Cycles	ARM Cycles
Stromregelung	1498	889 - 935
Geschwindigeits- und Stromregelung	1953	1137 - 1157

Tabelle 5.5: Messergebnisse der Field Oriented Control

Wird nur die Stromregelung berechnet, ist RISC-V 60.2% bis 68.5% langsamer als ARM. Werden Geschwindigkeits- und Stromregelung berechnet, so ist RISC-V 68.8% bis 71.8% langsamer.

5.2.5 Auswertung

Als Teil der Auswertung sollen Verbesserungen des generierten Assembler-Codes betrachtet werden. Diese Verbesserungen haben nicht das Ziel, den bestmöglichen Code zu erhalten, sondern für einen Compiler realistische Verbesserungen zu identifizieren.

Initialisierung mehrerer Variablen

Bei der Initialisierung der FOC werden insgesamt 23 Variablen gesetzt. Davon werden 18 Variablen auf den Wert 0 und die restlichen 5 Variablen auf unterschiedliche Werte gesetzt. Die Variablen sind bei RISC-V in einem Speicherbereich von 292 Bytes positioniert. Bei ARM sind drei der Variablen separat von den anderen in einem Speicherbereich von 12 Bytes, der Rest der Variablen in einem Speicherbereich von 138 Bytes positioniert. Für diese Trennung ist kein Grund identifizierbar - alle Variablen sind in der gleichen C-Quelldatei deklariert.

Für RISC-V sind mehrere Mängel erkenntlich:

- Drei 64-Bit Variablen werden auf den Wert 0 gesetzt. Dies erfordert für RISC-V zwei 32-Bit Speichervorgänge. Der Compiler füllt hierfür zwei separate Register mit dem Wert 0 und nutzt das eine Register für die unteren 32 Bit und das andere Register für die oberen 32 Bit anstatt eins für beide zu verwenden.
- Der generierte Code verwendet sowohl komprimierte Speicherinstruktionen mit einem auf 0 gesetztem Register[1] als auch normale Speicherinstruktionen mit dem Nullregister x0. Optimal sollte nur eine der beiden Varianten verwendet werden, um entweder Codegröße oder Laufzeit zu sparen.
- Für einen Speichervorgang wird für einen normalen Speichervorgang der Wert 0 generiert, statt das Register x0 zu verwenden.
- Für fast alle Speichervorgänge werden die oberen Addressbits erneut generiert. Acht davon überschreiben dabei das Register mit demselben Wert. Zwei schreiben denselben Wert in ein anderes Register, ohne Grund - das alte Register wird nicht für andere Zwecke verwendet.

[1] Die komprimierte Speicherinstruktion kann nur die Register x8-15 als Quellregister verwenden.

- Es werden zwei verschiedene Werte als obere Addressbits benötigt. Der Funktion stehen genug Register zur Verfügung, um hierfür zwei Register zu verwenden. Alternativ könnte die Reihenfolge der Speicherzugriffe umsortiert werden, um mehrere Wechsel zwischen den beiden Addresswerten zu vermeiden - dies würde fünf Wechsel ersparen.

Der RISC-V Compiler fusioniert die Initialisierung auf den Wert 0 für zwei 16-Bit Werte einer *struct* in einen Speicherzugriff von 32 Bit zusammen. Bei zwei benachbarten 16-Bit Variablen, die auf 0 gesetzt werden, wird jedoch kein 32-Bit Speicherzugriff generiert. Der ARM-Compiler führt keine solche Fusionen von Speicherzugriffen durch - alle 16-Bit Variablen werden mit 16-Bit Zugriffen beschrieben.

Verwendung von Strukturen

Die Funktion für das Messen der drei Phasenströme gibt die Messwerte in Form einer Struktur aus drei vorzeichenbehafteten 16-Bit Werten zurück. RISC-V gibt diesen Wert auf die ersten beiden Argumentregister (a0 und a1) verteilt zurück. Da die Struktur größer als 32 Bit ist, wird bei ARM die Funktion um ein implizites Funktionsargument erweitert - eine Stackaddresse, wo die zurückzugebene Struktur hineingeschrieben wird. Derselbe Mechanismus wird bei RISC-V für Strukturen, die größer als 64 Bit sind, verwendet. Strukturen, die größer als 32 und kleiner als 64 Byte sind können somit bei RISC-V, aber nicht bei ARM den Speicherzugriff über den Stack vermeiden.

Da RISC-V im Gegensatz zu ARM zurzeit keine dedizierten Instruktionen für das Bearbeiten von Bitfeldern hat, müssen mehrere Struktur-Elemente innerhalb von 32 Bit durch Bitverschiebung, Und-Verknüpfung und Oder-Verknüpfung bearbeitet werden. ARM kann dies mit Instruktionen für die Bearbeitung von Bitfeldern durchführen.

Weitere Mängel

Bei Funktionen, die keinen Speicherplatz auf dem Stack benötigen modifiziert der RISC-V Compiler teilweise trotzdem dem Stack Pointer. Bei vier Funktionen wurden 16 oder 32 Byte reserviert und ohne eine einzige Verwendung des Stacks wieder freigegeben. Eine Ursache für diese zweckfreie Stack-Operation konnte nicht identifiziert werden. Dies hat keinen Einfluss auf die korrekte Funktionsweise des Programmes.

5.3 Algorithmen nach Clark und Park

Die Algorithmen nach Clark und Park können verwendet werden, um Phasenströme zwischen verschiedenen Koordinatensystemtypen zu transformieren. Die Clark-Transformation bringt einen Vektor aus einem Dreiphasensystem in ein zweiachsiges Koordinatensystem über. Dies erlaubt die Darstellung eines dreiphasigen Motorstroms als einen Zeiger in der komplexen Ebene. Durch die Park-Transformation wird dieses zweiachsige Koordinatensystem rotiert, um eine Betrachtung unabhängig von dem aktuellen Motorwinkel zu erlauben. Beide Transformationen verfügen über eine inverse Transformation.

5.3.1 Die Algorithmen

Beide Transformationen beruhen auf Matrixmultiplikationen. Die Park-Transformation verwendet eine zweidimensionale Drehmatrix, um den Betrachtungswinkel des Koordinatensystemes zu ändern [17]:

$$\begin{bmatrix} I_d \\ I_q \end{bmatrix} = \begin{bmatrix} cos(\theta) & sin(\theta) \\ -sin(\theta) & cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} I_\alpha \\ I_\beta \end{bmatrix}$$

Die inverse Park-Transformation verwendet dieselbe Drehmatrix, jedoch mit invertiertem Drehvektor θ [17]:

$$\begin{bmatrix} I_\alpha \\ I_\beta \end{bmatrix} = \begin{bmatrix} cos(-\theta) & sin(-\theta) \\ -sin(-\theta) & cos(-\theta) \end{bmatrix} \cdot \begin{bmatrix} I_d \\ I_q \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} I_d \\ I_q \end{bmatrix}$$

Die Clark-Transformation überführt die drei Phasenströme I_U, I_V und I_W in ein zweidimensionales Koordinatensystem mit den Strömen I_α und I_β [17]:

$$\begin{bmatrix} I_\alpha \\ I_\beta \end{bmatrix} = \frac{2}{3} \begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \cdot \begin{bmatrix} I_U \\ I_V \\ I_W \end{bmatrix}$$

Die inverse Clark-Transformation lautet:

$$\begin{bmatrix} I_U \\ I_V \\ I_W \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} \\ -\frac{1}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \cdot \begin{bmatrix} I_\alpha \\ I_\beta \end{bmatrix}$$

5.3.2 Die Implementation der Algorithmen

Für alle Zahlenwerte wird Fixpunkt-Arithmetik mit einer Vor- und 15 Nachkommastellen verwendet. Die Sinusfunktion besteht aus einer Look-Up-Table für den ersten Quadranten bzw. Winkel von $\theta \in [0, 1/2\pi)$ und Umrechnungen für die anderen drei Quadranten. Die Kosinusfunktion wird über die Sinusfunktion implementiert ($cos(\theta) = sin(\theta + \frac{1}{2}\pi)$). Des weiteren wird der Term $\frac{\sqrt{3}}{2}$ als vorberechneter Konstantwert verwendet.

5.3.3 Messergebnisse

Algorithmus	RISC-V		ARM
	Zyklen	Instruktionen	Zyklen
Clark Transformation	78	24	32
Inverse Clark Transformation	139	24	39
Park Transformation	99	64	81-85
Inverse Park Transformation	99	62	81-86

Tabelle 5.6: Messergebnisse der Clark- und Park-Transformationen

5.3.4 Auswertung der Messergebnisse

Analyse des generierten Maschinencodes

Für die Analyse des Maschinencodes werden die Instruktionszahlen aller vier Transformationen ermittelt. Dasselbe wird für die Hilfsfunktionen für Sinus und Kosinus durchgeführt - hierbei wird berücksichtigt, dass beide Funktionen dasselbe Winkelargument erhalten.

Algorithmus	RISC-V	ARM
Clark Transformation	19	15
Inverse Clark Transformation	18	18
Park Transformation	30	20
Inverse Park Transformation	29	20
Sinus/Kosinus-Paar	28-34	29

Tabelle 5.7: Instruktionszahlen der Clark- und Park-Transformationen

Die unterschiedlichen Instruktionszahlen basieren auf den folgenden Unterschieden zwischen den beiden Architekturen:

- Eine Multiplikation und eine Addition können mit dem ARM-Befehlssatz durch die Verwendung einer der *Multiply-Add*-Instruktionen - *SMLABB*[2] - zusammengefasst werden. RISC-V muss diese Operationen mit zwei separaten Instruktionen durchführen.
- Eine Multiplikation um den Faktor 3 kann durch eine Addition von dem Operanden und dem um eine Bitposition nach links geschobenen Operanden ersetzt werden. Dies kann durch die Verwendung des Barrel-Shifters, den ARM als Teil des zweiten Operandes vieler Instruktionen zur Verfügung stellt, mit nur einer Instruktion durchgeführt werden.
- Die im Voraus berechnete Konstante für den Term $\frac{\sqrt{3}}{2}$ kann der ARM-Befehlssatz mit nur einer Instruktion geladen werden, während der RISC-V-Befehlssatz dafür zwei Instruktionen benötigt.
- Für das Speichern und Wiederherstellen von Registern über den Stack nutzt ARM nur jeweils einen Befehl - *PUSH*[3] und *POP*[4]. RISC-V muss für jedes Register separate Load- (Load word - *LW*) und Store-Befehle (Store word - *SW*) sowie Addition und Subtraktion für die Änderung des Stack-Pointers vornehmen.

[2] *Signed multiply and accumulate, bottom by bottom*: Diese Instruktion multipliziert die unteren 16 Bits von Registern und addiert ein drittes Register

[3] PUSH ist ein Alias für die Instruktion „STMDB sp!, reglist“ (***St**ore **m**ultiple, **d**ecrease address **b**efore access*). Diese Instruktion dekrementiert den Stack Pointer und speichert an der neuen Addresse wiederholt, bis alle Register in *reglist* gespeichert sind

[4] POP ist ein Alias für die Instruktion „LDMIA sp!, reglist“ (***L**oad **m**ultiple, **i**ncrease address **a**fter access*). Diese Instruktion speichert an der Stack Pointer Addresse und inkrementiert den Stack Pointer wiederholt, bis alle Register in *reglist* gespeichert sind

- Für den Sprung zurück nach Abschluss einer Funktion kann die ARM-Architektur die Stackoperation *POP* nutzen, um den Programmzähler direkt zu aktualisieren. Dies erspart eine explizite *return*-Instruktion.
- Durch die höhere Registerzahl kann RISC-V mehr Register verwenden, ohne diese vorher auf dem Stack zu sichern. Dies erspart die Instruktionen für die Sicherung und Wiederherstellung auf dem Stack.

Analyse der Laufzeit

Für den RISC-V-Prozessor werden für die Laufzeitanalyse die ausgeführten Instruktionen in mehrere Kategorien sortiert. Die Kategorien unterscheiden sich durch die für die Instruktion benötigten Zyklen.

- Simple Instruktionen - in einem Zyklus ausführbar.
- Sprunganweisungen - 1 Zyklus für nicht durchgeführte bedingte Sprünge, 2-3 Zyklen für durchgeführte Sprünge. Dies beinhaltet die Sprünge durch Funktionsaufrufe und -rückkehr.
- Speicherzugriffe - 1-2 Zyklen, abhängig von benachbarten Instruktionen.
- Divisonen - unbekannte, von den verarbeiteten Daten abhängige Zyklenzeit

Die Kategorisierung des Maschinencodes der Funktionen ergibt folgende Werte:

Algorithmus	Befehlstyp			
	Simpel	Sprung	Speicherzugriff	Divison
Clark Transformation	16	2	4	1
Inverse Clark Transformation	13	3	5	3
Park Transformation	36-42	12	16	0
Inverse Park Transformation	35-41	11	16	0

Tabelle 5.8: Instruktionszahlen nach Befehlsart für die Clark- und Park-Transformation

Für die Betrachtung der Zyklenzeit für die Divisionen werden die Zyklenzeiten aller anderen Kategorien von der gesamten gemessenen Zyklenzeit abgezogen. Bei der Clark-Transformation ist der Divisor 2 und der Divident Teil der verarbeiteten Daten. Die

Anzahl an benötigten Zyklen für die Division der Clark-Transformation liegt zwischen 48 und 54. Diese Division nutzt hat den Wert 2 als Divisor und einen von den bearbeitenden Daten abhängigen Dividend. Die Divisonen der inversen Clark-Transformation benötigen zusammen eine Zyklenzeit zwischen 107 und 115. Alle drei Divisionen haben den Divisorwert 3 und einen variierenden Dividend. Vergleichsweise benötigt der ARM-Prozessor für eine Division nur 2-12 Zyklen [3].

5.4 Trinamic Motion Control Language

Die Trinamic Motion Control Language (kurz TMCL) ist ein von Trinamic Motion Control entwickeltes Protokoll zum Senden von Befehlen an einen Mikrocontroller. Es wird verwendet, um eine einheitliche Steuerung von Motoren über einen vordefinierten Befehlssatz zu ermöglichen [20].

5.4.1 Aufbau des Protokolls

Die Kommunikation über TMCL findet durch einzelne Befehls- und Antwort-Datenpakete statt. Für jedes Befehls-Datagramm wird ein Antwort-Datagramm zurückgesendet. Ein TMCL Befehl setzt sich zusammen aus einem Opcode (1 Byte), einem Typ-Spezifikator (1 Byte), einer Motornummer (1 Byte) und einem Datum (4 Byte). Je nach verwendetem Bus werden noch Host- und Modul-ID (je 1 Byte) für die korrekte Selektion der Kommunikationspartner und eine simple Prüfsumme (1 Byte) verwendet. Bei einer Verbindung über USB werden alle drei optionalen Parameter verwendet, bei einer Verbindung über CAN wird auf die Prüfsumme verzichtet und die ID des CAN-Datenpaketes ersetzt die Sender-ID - Modul-ID bei Befehlen, Host-ID bei Antworten. Die Antwort auf einen Befehl sendet ein Status-Byte (1 Byte), den bearbeiteten Befehl (1 Byte), ein Datum (4 Byte) zurück. Wie bei dem Befehl sind hier Host- und Modul-ID sowie Prüfsumme optional.

5.4.2 Vergleich der Implementationen

Für die Verarbeitung eines empfangenen Befehls verwenden beide Implementationen ein *switch/case*-Statement, um das Opcode-Feld auszuwerten. Die anderen Felder des Datenpaketes werden abhängig vom verwendeten Opcode ausgewertet.

Für das Empfangen der Befehle über USB nutzt die Landungsbrücke einen Interrupt. Das Senden von Antworten geschieht über eine periodisch aufgerufene Funktion, welche die zu sendenen Daten aus einem Ring-Buffer liest und an das Hardwaremodul weiterleitet.

Befehle werden bei ROCINANTE in einem FIFO-Buffer im USB-Modul zwischengespeichert, bis die Software diese abruft. Da die TMCL-Datagramme vollständig in den FIFO des USB-Moduls passen, muss kein Interrupt für das Leeren des FIFOs verwendet werden, solange die Bearbeitung der TMCL-Befehle schneller ist, als neue Befehle gesendet werden. Zum Senden von Daten werden die Daten in einen zweiten FIFO-Buffer im USB-Modul geschrieben.

5.4.3 Messaufbau

Als Messgröße werden die bearbeiteten Befehle pro Sekunde verwendet. Dies wird von einem PC aus gemessen, welcher Befehle sendet und die Antworten empfängt. Die Messung wird für zwei verschiedene TMCL-Befehle durchgeführt. Einer der Befehle verrichtet keine Rechenarbeit auf dem Mikrocontroller und der andere Befehl wartet 1000 Prozessortakte, um verrichtete Arbeit zu simulieren. Beide Befehle verwenden den gleichen Opcode, aber unterschiedliche Werte für das Type-Feld.

Für die Kommunikation vom PC aus wird ein Python-Skript verwendet, um wiederholt über einen COM-Port den TMCL-Befehl zu senden und die Antwort zu lesen. Eine komplette Messung besteht aus mehreren Runden - für jede Runde wird die benötigte Zeit gemessen, um mehrmals über TMCL zu kommunizieren. Für die Zeitmessung wird die Python-Funktion *time.perf_counter_ns()* verwendet. Diese Funktion gibt einen Zeitstempel mit Nanosekundenauflösung zurück, im Rahmen der Messung wird allerdings auf Millisekundengenauigkeit gerundet.

Die TMCL-Implementation auf den Mikrocontrollern wird erweitert, um den zeitlichen Ablauf der Befehlsverarbeitung auf einem Oszilloskop darstellen zu können. Hierfür werden an zwei Pins Pulse generiert - ein Puls stellt die Gesamtdauer der Befehlsverarbeitung da, der andere Puls stellt den Antwortvorgang nach der Befehlsverarbeitung dar.

5.4.4 Durchführung der Messung

Zur Messung der TMCL-Leistungsfähigkeit wurden 20 Runden mit jeweils 100000 gesendeten TMLC-Datagrammen vermessen. Die gemessenen Zeiten wurden gemittelt und die

bearbeiteten Befehle pro Sekunde errechnet:

Befehlstyp	RISC-V		ARM	
	Zeit in ms	Befehle / s	Zeit in ms	Befehle / s
Keine Arbeit	14429	6930	21495	4652
1000 Zyklen Arbeit	15941	6272	21660	4616

Tabelle 5.9: Messergebnisse der Trinamic Motion Control Language

Die Betrachtung der Zeitverläufe der Befehlsverarbeitung ergibt folgende Verläufe:

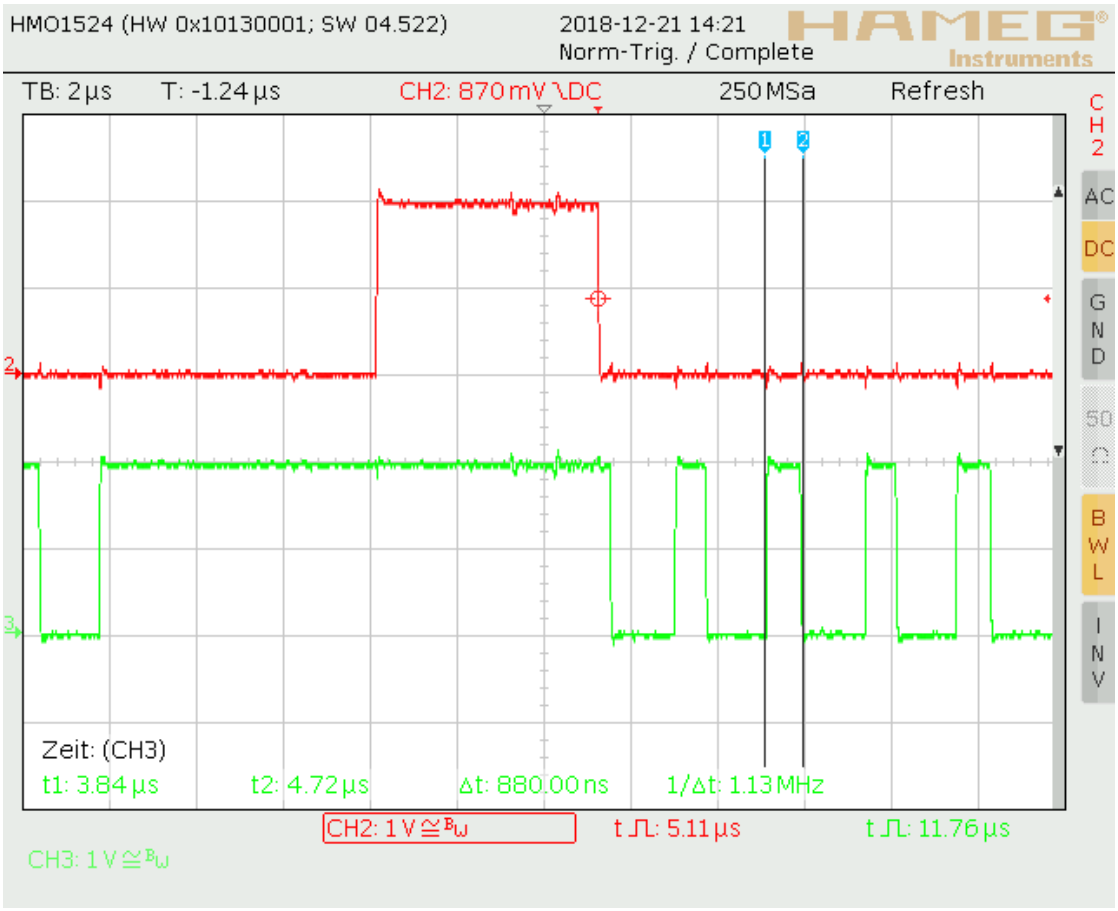

Abbildung 5.5: Dauer der Verarbeitung eines TMCL-Befehls auf der ROCINANTE. Grün: Abfrage der USB-Schnittstelle und ggf. Befehlsverarbeitung. Rot: Senden der Antwort

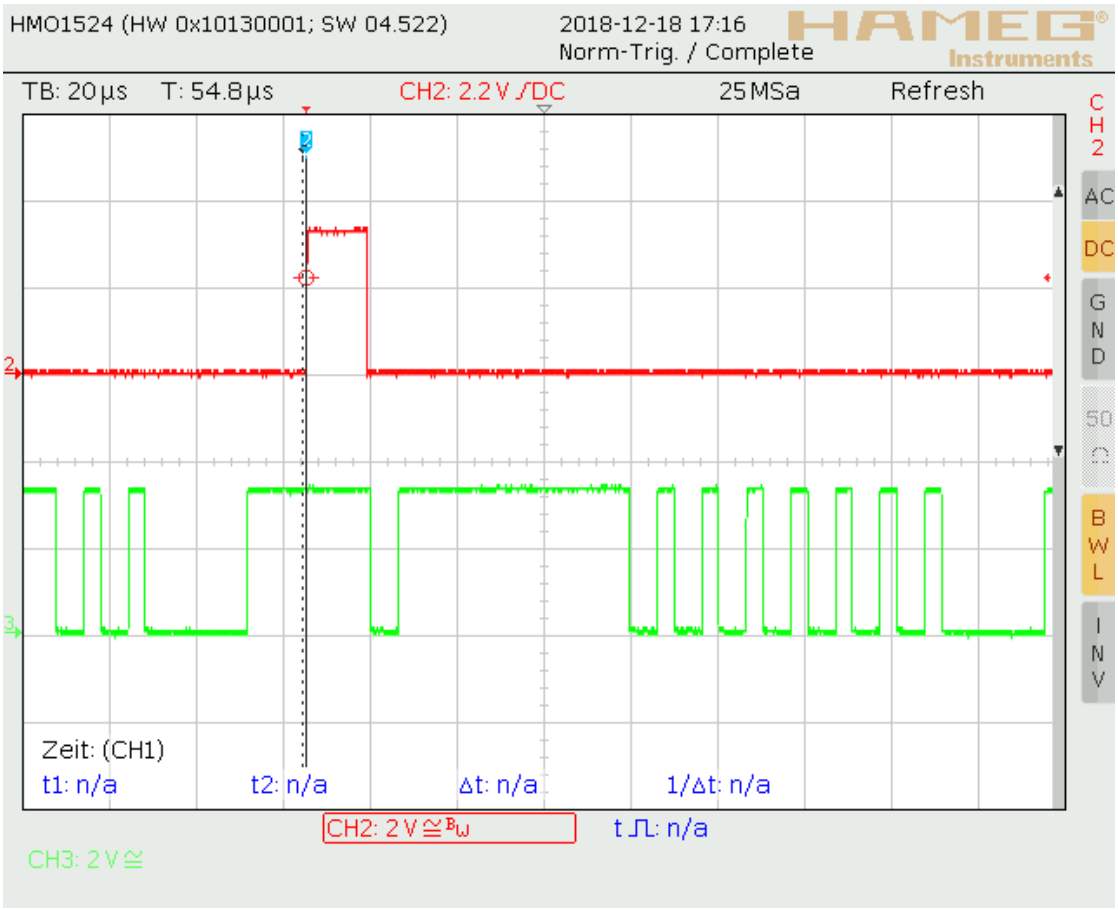

Abbildung 5.6: Dauer der Verarbeitung eines TMCL-Befehls auf der Landungsbrücke. Grün: Abfrage der USB-Schnittstelle und ggf. Befehlsverarbeitung. Rot: Senden der Antwort

5.4.5 Auswertung der Messergebnisse

Für die Bearbeitung von TMCL-Befehlen ist das Projekt ROCINANTE 50% schneller als die Landungsbrücke sofern keine weitere Arbeit durch den Befehl verursacht wird. Bei einer Arbeitslast von 1000 Zyklen ist ROCINANTE nur noch 36% schneller.

Zeitverläufe der Befehlsverarbeitung

Die Zeitverläufe zeigen die Befehlsschleife in Arbeit. Die kurzen grünen Pulse entstehen, wenn kein Befehl empfangen wurde. Die Bearbeitung eines Befehls zeigt sich durch den

längeren grünen Puls mit einem roten Puls - dem Senden der Antwort. Bei der Landungsbrücke zeigen sich zusätzlich noch weitere Effekte. Vor Bearbeitung des Befehls ist die Zeit zwischen den Befehlsschleifen länger durch den Empfangsinterrupt. Nach dem Senden der Antwort benötigt die nächste Befehlsschleife deutlich mehr Zeit, da die Leseanfrage an den USB-Stack die periodische Funktion für das Senden von Daten aktiviert. Diese periodische Funktion blockiert, bis die Daten gesendet wurden. Im Verlaufe der folgenden Analyse wird dieser Abschnitt als *Folgebefehl* bezeichnet.

Aus den Zeitverläufen werden folgende Zeiten für die einzelnen Programmabschnitte ausgelesen:

Programmabschnitt	ROCINANTE	Landungsbrücke
Empfangs-Interrupt	-	17.2 μs
Kein Befehl	0.76 μs	3.8 μs
Antwort senden	5.12 μs	14 μs
Kompletter Befehl	11.76 μs	30.4 μs
Folgebefehl	0.76 μs	53.2 μs

Tabelle 5.10: Zeitverläufe der Einzelschritte der Befehlsverarbeitung von TMCL

Die Verarbeitung eines TMCL-Befehls ohne Zusatzarbeit benötigt auf der Landungsbrücke durchschnittlich 70.66 μs mehr, als auf ROCINANTE. Die vollständige Bearbeitung eines Befehls dauert 18.76 μs länger. Der blockierende Aufruf der periodischen USB-Funktion bei der Landungsbrücke resultiert in zusätzlichen 49.4 μs für den Durchlauf der Befehlsschleife ohne zu bearbeitenden Befehl. Dies resultiert in 52.44 μs mehr, als bei ROCINANTE. Beide Effekte zusammen entsprechen etwa dem Gesamtunterschied von 70.66 μs zwischen den beiden Plattformen.

Effekte von Arbeitslast

Wird durch den Befehl eine Arbeit auf dem Prozessor - hier 1000 Zyklen Verzögerung - durchgeführt, so ändern sich die Bearbeitungszeiten beider Plattformen. Für ROCINANTE entsprechen die 1000 Zyklen bei 40 MHz Taktung in 25 μs pro Befehl bzw. 2.5s für 100000 Befehle. In der Messung erhöht sich die benötigte Gesamtzeit jedoch nur um

1.512s. Der Unterschied von etwa einer Sekunde entsteht vermutlich durch eine Überlappung von Prozessorrechenzeit mit anderen Verzögerungen innerhalb des Kommunikationsablaufes.

Weitere Betrachtungen

Durch die Verwendung eines Interrupts zum Empfangen von Daten auf der Landungsbrücke muss der Prozessor Arbeit verrichten, die bei ROCINANTE komplett in Hardware geschieht. Im Rahmen dieser Messung hat der Prozessor keine weitere Aufgabe zu bearbeiten, wodurch die zusätzliche Prozessorlast keine anderen Einflüsse hat. In einer Applikation würde diese Art der Implementation jedoch höhere Kosten an Rechenzeit mit sich bringen.

5.5 Zusammenfassung der Messergebnisse

5.5.1 Leistungsfähigkeit des Prozessors

Die Messungen zeigen, dass RISC-V gegenüber von ARM bei allen durchgeführten Prozessorvergleich schlechter abschneidet.

Algorithmus	Instruktionen		Mehraufwand für RISC-V
	RISC-V	ARM	
Normaler CRC, Branching	12175	11702	4.04%
Normaler CRC, Bitlogik	11593	8683	33.51%
Reflektierter CRC, Branching	16015	13935	14.93%
Reflektierter CRC, Bitlogik	15433	10841	42.36%
FOC, Stromregelung	1498	912	64.25%
FOC, Geschwindigkeits- und Stromregelung	1953	1147	70.27%
Clark Transformation	78	32	143.75%
Inverse Clark Transformation	139	39	256.41%
Park Transformation	99	83	19.28%
Inverse Park Transformation	99	84	17.86%

Tabelle 5.11: Zusammenfassung aller Messergebnisse des Prozessorvergleichs. Für die teilweise leicht schwankenden Instruktionszahlen von ARM wurde in dieser Zusammenfassung der Mittelwert gebildet.

Durch die Analyse haben sich mehrere Verbesserungsmöglichkeiten für den RISC-V Prozessor gezeigt:

- Der Compiler für RISC-V hat Raum für Verbesserung. Diese können unabhängig von der von Trinamic entwickelten Hardware in der Zukunft die Leistungsfähigkeit des Prozessors steigern.
- Die Prozessorimplementation für die Divisionsinstruktion ist relativ langsam. Da diese Verwendung in verschiedensten Algorithmen findet, bietet die Verbesserung dieser Instruktion weitere Leistungssteigerungen.
- Die Ergänzung der RISC-V-Erweiterungen für Bitlogik hat das Potenzial, viele verschieden Operationen - zum Beispiel Zugriffe auf Bitfelder in Hardwareregistern - zu verbessern. Da diese Erweiterung noch nicht fertig standardisiert ist, kann diese Verbesserung nicht in näherer Zukunft geschehen.

Des weiteren haben sich Unterschiede zwischen den beiden Architekturen gezeigt. Diese sind Teil der standardisierten ISAs und können somit nicht durch Verbesserung der Implementationen ausgeglichen werden:

- ARM bietet eine größere Vielfalt an Befehlen für die Ablaufsteuerung. Die Table-Sprungbefehle (*TBB*, *TBH*) ermöglichen eine leichte Umsetzung von *switch/case*-Statements. Die Vermeidung von Sprüngen durch bedingte Ausführung (*IT*) beliebiger Befehle erlaubt es, Verzögerungen durch den *Pipeline Flush* zu vermeiden.
- Durch den Barrel-Shifter kann ARM zusätzliche Bitshift-Befehle einsparen.
- Die Load- und Store-Multiple Befehle von ARM erlauben kompakten Code für das Sichern und Wiederherstellen von Registern zu Beginn bzw. Ende einer Funktion.
- Die höhere Registerzahl erlaubt es RISC-V, Interaktionen mit dem Stack einzusparen. Jedoch müssen dementsprechend auch mehr Register bei einem Interrupt gespeichert werden.

5.5.2 Leistungsfähigkeit des Gesamtsystems

Die TMCL-Messung zeigt, dass trotz dem Unterschied in der Leistungsfähigkeit zwischen den beiden Prozessoren das RISC-V-System schneller arbeiten kann. Durch effizientes Design der Hard- und Software bietet der leistungsschwächerere RISC-V Prozessor mit der ROCINANTE ein leistungsstärkeres Gesamtsystem, als die Landungsbrücke.

Die Messung zeigte des weiteren, dass für die Landungsbrücke die Softwareimplementation für die USB-Kommunikation Potenzial für Verbesserung hat. Da die existierende Implementation jedoch schon seit mehreren Jahren aktiv verwendet wird, deutet darauf hin, dass der aktuelle Stand der Software für Applikationen bereits ausreichend ist.

6 Fazit

Zuletzt werden die Resultate der Arbeit betrachtet. Die Bearbeitung der Aufgabenstellung, die identifizierten Vor- und Nachteile der verwendeten Prozessoren sowie des implementierten Systems sowie die Relevanz dieser Arbeit für das Projekt ROCINANTE werden zusammengefasst.

6.1 Umsetzung der Aufgabenstellung

Die vollständige Umsetzung des geplanten Systemes konnte in dieser Arbeit nicht beendet werden. Bei der Zusammenarbeit mit Hardwareentwicklern wurde ein hohes Maß an Qualität durch das gemeinsame Design von Hardwarefunktionalität und der Schnittstelle zwischen Hard- und Software erreicht. Dieses sorgfältige Vorgehen stand jedoch der vollständigen Umsetzung im Weg - hierfür wird mehr Zeit benötigt.

Der Vergleich der ROCINANTE mit der Landungsbrücke bestätigte alle Vermutungen. Der Wechsel des Prozessors von ARM auf RISC-V büßt Leistungsfähigkeit eines ausgereiften Systemes ein im Austausch für die Flexibilität und freie Verfügbarkeit von RISC-V. Die sorgfältige Designarbeit für die Hardware zeigt auch ihre Vorteile - eine schlanke, simple Softwareimplementation und ein schnelles Gesamtsystem rechtfertigen den hohen Zeitaufwand.

6.2 Erkenntnisse über RISC-V gegenüber ARM

Beim Vergleich von RISC-V mit ARM konnten mehrere Defizite von RISC-V identifiziert werden. Fast alle dieser Unterschiede gegenüber ARM basieren darauf, dass RISC-V jünger als ARM ist und somit noch mehr Raum für Verbesserungen bietet. So kann der Compiler für RISC-V noch verbessert und fehlende Erweiterungen der RISC-V ISA

können noch fertiggestellt werden. Einige weitere Unterschiede beruhen auf verschiedenen Designentscheidungen für die Architekturen. Hier sorgt die RISC-Designphilosophie bei RISC-V für weniger und simplere Instruktionen, als bei ARM - und zahlt dafür mit geringen Leistungseinbußen.

Mit weiterer Entwicklung von RISC-V ist daher zu erwarten, dass die Unterschiede in Leistungsfähigkeit immer geringer werden. Langfristig können die wirtschaftlichen Vorteile, die eine offene Architektur bietet, den immer geringer werdenden Leistungsunterschied zwischen den verschiedenen ISA-Designs für Unternehmen rechtfertigen.

6.3 Weitere Schritte für das Projekt ROCINANTE

Die aus dieser Arbeit gewonnenen Erkenntnisse und technische Realisierungen bieten ein guten Start für das Projekt ROCINANTE - und somit dem Einstieg in den Mikrocontrollermarkt von Trinamic Motion Control. Die Vorteile der Spezialisierung auf Steuerung eines Systemes mit integrierten Motoren waren bei dem Vergleich der Gesamtsysteme erkennbar.

Das hohe Maß an investierter Zeit in das Design der Hardwaremodule zu Beginn einer Projektreihe sorgt dafür, dass neue Produkte wenig bis gar keine neuen Revisionen erfordern. Die hohe Qualität, die durch die Zeitinvestitionen erreicht wird, kann durch die gesamte Produktreihe hindurch verwertet werden.

Für die Weiterführung des Projektes ROCIANTE sind nur noch wenige offene Punkte zu bearbeiten. Die Implementation der Interrupt-Mechanismen erfordert noch weitere Anpassungen von Hard- und Software. Ansonsten erfordert die weitere Entwicklung des Projektes ROCINANTE nur noch die Vervollständigung der Kommunikationsschnittstellen entsprechend der in dieser Arbeit konzipierten Designrichtlinien.

Abbildungsverzeichnis

2.1 Verschachtelung von Interrupts [26] 9

3.1 Designkonzept des Projektes ROCINANTE [9] 19
3.2 Strukturierung der Header-Dateien für Hardwarefunktionen 21

5.1 Illustration der Instruktionszahlen von der Zyklischen Redundanzprüfung 60
5.2 Illustration der zusätzlichen Taktzyklen für die Zyklische Redundanzprüfung 62
5.3 Durchschnittliche Takte pro Instruktion für die Zyklische Redundanzprüfung 63
5.4 Blockschaltbild der Field Oriented Control [23] 64
5.5 Dauer der Verarbeitung eines TMCL-Befehls auf der ROCINANTE 75
5.6 Dauer der Verarbeitung eines TMCL-Befehls auf der Landungsbrücke . . . 76

Tabellenverzeichnis

2.1 Liste der RISC-V ISA Erweiterungen [14] 12

4.1 Befehle für Zugriff auf die Performance-Counter von RISC-V 43

5.1 Beispiel für die Berechnung einer zyklischen Redundanzprüfung 51
5.2 Messergebnisse der Zyklischen Redundanzprüfung 55
5.3 Aufteilung der Instruktionen nach Programmabschnitt von der Zyklischen Redundanzprüfung 57
5.4 Theoretisch berechnete Instruktionszahlen von der Zyklischen Redundanzprüfung 59
5.5 Messergebnisse der Field Oriented Control 66
5.6 Messergebnisse der Clark- und Park-Transformationen 70
5.7 Instruktionszahlen der Clark- und Park-Transformationen 71
5.8 Instruktionszahlen nach Befehlsart für die Clark- und Park-Transformation 72
5.9 Messergebnisse der Trinamic Motion Control Language 75
5.10 Zeitverläufe der Einzelschritte der Befehlsverarbeitung von TMCL 77
5.11 Zusammenfassung aller Messergebnisse des Prozessorvergleichs 79

Codeverzeichnis

3.1 Zwischenspeicherung der CAN-Schnittstellenkonfiguration 27
3.2 Erste Implementation der Auswahl der Interrupt-Handler 32
3.3 Verbesserte Implementation der Auswahl der Interrupt-Handler durch Hardwareunterstützung 33

4.1 Beginn einer Zyklenmessung für RISC-V 44
4.2 Ende einer Zyklenmessung für RISC-V 44
4.3 Auslesen einer Zyklenmessung für RISC-V 44
4.4 Löschung einer Zyklenmessung für RISC-V 45
4.5 Zyklenmessung für ARM 46

Literaturverzeichnis

[1] *ARM registers*. – URL http://www.keil.com/support/man/docs/armasm/armasm_dom1359731128950.htm. – abgerufen am 04. April 2019

[2] *Assembler User Guide: Syntax of Operand2 as a constant*. – URL http://www.keil.com/support/man/docs/armasm/armasm_dom1361289851958.htm. – abgerufen am 04. April 2019

[3] *Cortex-M4 Technical Reference Manual: Cortex-M4 instructions*. – URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0439b/CHDDIGAC.html. – abgerufen am 04. April 2019

[4] *Archivierte Newsgruppen-Beiträge über die Verwendung von der Präprozessordirektive ifdef*. August 2001. – URL https://yarchive.net/comp/linux/ifdefs.html. – abgerufen am 04. April 2019

[5] *ISO/IEC 9899 - Committee Draft*. September 2007. – URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf. – abgerufen am 04. April 2019

[6] *Understanding and implementing CRC (Cyclic Redundancy Check) calculation*. Februar 2015. – URL http://www.sunshine2k.de/articles/coding/crc/understanding_crc.html. – abgerufen am 04. April 2019

[7] Anderson, Sean E.: *Bit Twiddling Hacks: Reverse an N-bit quantity in parallel in 5 * lg(N) operations*. – URL https://graphics.stanford.edu/~seander/bithacks.html#ReverseParallel. – abgerufen am 04. April 2019

[8] ARM Ltd.: *Arm Processors for the Widest Range of Devices - from Sensors to Servers*. – URL https://www.arm.com/products/silicon-ip-cpu. – abgerufen am 04. April 2019

[9] Bekkers, Corné: *Taking motion control to the next level with RISC-V*. November 2018. – URL https://blog.trinamic.com/2018/11/07/taking-motion-control-to-the-next-level-with-risc-v/. – abgerufen am 04. April 2019

[10] Dahad, Nitin: *Can Arm Survive RISC-V Challenge?* Februar 2019. – URL https://www.eetimes.com/author.asp?section_id=36&doc_id=1334306#. – abgerufen am 04. April 2019

[11] Eeckhout, L.: *Computer Architecture Performance Evaluation Methods*. Morgan & Claypool Publishers, 2010 (Synthesis lectures on computer architecture). – URL https://books.google.de/books?id=HtYuOaaNcL8C. – ISBN 9781608454679

[12] Nenni, Daniel: *Worldwide Design IP Revenue Grew 12.4% in 2017*. 2018. – URL https://www.semiwiki.com/forum/content/7450-worldwide-design-ip-revenue-grew-12-4-p-2017-a.html. – Zugriffsdatum: 2018-05-11. – abgerufen am 04. April 2019

[13] RISC-V Foundation: *RISC-V*. – URL https://riscv.org/. – abgerufen am 04. April 2019

[14] RISC-V Foundation: *The RISC-V Instruction Set Manual*. 2.2. – URL https://riscv.org/specifications/. – abgerufen am 04. April 2019

[15] Shimano, Crystal Chen; Greg Novick; K.: *pipelining*. – URL https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/pipelining/index.html. – abgerufen am 04. April 2019

[16] Shimano, Crystal Chen; Greg Novick; K.: *risc vs. cisc*. Dezember 2006. – URL https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/. – abgerufen am 04. April 2019

[17] Texas Instruments Europe: *Field Orientated Control of 3-Phase AC-Motors*. Februar 1998. – URL http://www.ti.com/lit/an/bpra073/bpra073.pdf. – abgerufen am 04. April 2019

[18] Trinamic Motion Control: *Landungsbrücke*. – URL https://www.trinamic.com/support/eval-kits/details/landungsbruecke/. – abgerufen am 04. April 2019

[19] Trinamic Motion Control: *Modular Evaluation System*. – URL https://www.trinamic.com/support/eval-kits/. – abgerufen am 04. April 2019

[20] Trinamic Motion Control: *Protocols*. – URL https://www.trinamic.com/technology/architecture/protocols/. – abgerufen am 04. April 2019

[21] Trinamic Motion Control: *Startrampe*. – URL https://www.trinamic.com/support/eval-kits/details/Startrampe/. – abgerufen am 04. April 2019

[22] Trinamic Motion Control: *TMC4671-ES*. – URL https://www.trinamic.com/products/integrated-circuits/details/tmc4671-es/. – abgerufen am 04. April 2019

[23] Trinamic Motion Control: *TMC4671 Datasheet*. 1.06, Februar 2019. – 47 S. – URL https://www.trinamic.com/fileadmin/assets/Products/ICs_Documents/TMC4671_datasheet_v1.06.pdf. – abgerufen am 04. April 2019

[24] Wilson, Jim: *RISC-V: Add interrupt attribute support*. Mai 2018. – URL https://github.com/riscv/riscv-gcc/commit/ae581c86a9848c323b60d7f5e4900ed4e473f279. – abgerufen am 04. April 2019

[25] Yadin, A.: *Computer Systems Architecture*. CRC Press, 2016 (Chapman & Hall/-CRC Textbooks in Computing). – URL https://books.google.de/books?id=KzeLDQAAQBAJ. – ISBN 9781315355924

[26] Yiu, Joseph: *A Beginner's Guide on Interrupt Latency - and Interrupt Latency of the Arm Cortex-M processors*. April 2016. – URL https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/beginner-guide-on-interrupt-latency-and-interrupt-latency-of-the-arm-cortex-m-processors. – abgerufen am 04. April 2019

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Ort	Datum	Unterschrift im Original