UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR TECHNISCHE INFORMATIK

RISC-V Assembler - Machbarkeitsstudie für die Lehre

RISC-V Assembler - Feasibility Study for Education

Bachelorarbeit

verfasst am
Institut für Technische Informatik

im Rahmen des Studiengangs
Medizinische Informatik
der Universität zu Lübeck

vorgelegt von
Philipp Rothmann

ausgegeben und betreut von
Dr.-Ing. Kristian Ehlers

Lübeck, den 4. Dezember 2019

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

_____

Philipp Rothmann

Zusammenfassung

RISC-V ist eine neue, offene und freie Befehlssatzarchitektur. Nicht nur in der Wirtschaft erreicht RISC-V große Aufmerksamkeit sondern auch in der akademischen Lehre wird es immer beliebter. Diese Arbeit zeigt, wie Teile des Lehrmoduls *Technische Grundlagen der Informatik 1* mit RISC-V Architektur umgesetzt werden können. Es erfolgt die beispielhafte Realisierung von Assemblerprogrammen für das Entwicklungsboard *Hifive1RevB* und ein Vergleich mit dem bisher verwendeten AVR Assembler auf dem *ATmega16*. Vor allem die Offenheit des Standards bietet viele Vorteile. So kann ein RISC-V Core speziell für die Lehre konzipiert und entwickelt werden. Insgesamt bietet RISC-V eine moderne, modulare und simple Architektur, die sich in den Lehrbetrieb einbringen lässt.

Abstract

RISC-V is a new, open and free Instruction Set Architecture. Not only in industry RISC-V attracts great attention, but also in academic education it becomes more and more popular. This work will show how parts of the class *Technical Foundations of Computer Science 1* can be realized in RISC-V Architecture. Therefore assembler programs are presented on the development board *Hifive1RevB* and compared to the so far used AVR assembler on the *ATmega16*. Above all, RISC-V being an open standard offers many advantages. For instance, a special on education designed RISC-V core can be used. Overall, RISC-V is a modern, modular and simple architecture that can be integrated into educational practice.

Inhaltsverzeichnis

1

Einleitung

Ein Kernelement des Informatikstudiums ist das Erlernen der grundlegenden Funktionsweise eines Computers. Dazu werden die Architektur von Computern (bzw. Prozessoren) und elektrotechnische Grundlagen im Lehrmodul *Technische Grundlagen der Informatik 1* (*TGI1*) gelehrt. Für die hardwarenahe Programmierung wird die Assemblersprache anhand der AVR Architektur unterrichtet und auf dem Mikrocontroller ATmega16 ausgeführt. Die Schnittstelle zwischen dem Prozessor und der Assemblersprache wird als Befehlssatzarchitektur (Instruction Set Architecture (ISA)) bezeichnet. Aktuelle Entwicklungen der University of California, Berkeley haben die neue ISA RISC-V hervorgebracht. Dieser Standard ist im Gegensatz zu weit verbreiteten ISAs (wie z. B. AVR, ARM, x86, usw.) frei verfügbar und offen. Dies bietet für die Lehre den großen Vorteil, dass Prozessoren von Grund auf, ohne Lizenzabkommen eingehen zu müssen, selbst entwickelt werden können. RISC-V konnte bereits in Prozessorimplementierungen durch bessere Platzeffizienz, Energieeffizienz und höherer Rechenleistung im Vergleich zu ARM Prozessoren überzeugen (Lee u. a., 2014, Sec. 2A). Die Modularität von RISC-V ermöglicht das Entwickeln von Prozessoren unterschiedlichster Größe mit der gleichen ISA. Das *Parallel Ultra Low Power* (*PULP*) opensource Projekt der ETH Zurich beinhaltet sowohl kleine, energiesparsame und platzeffiziente 32-Bit Mikrocontroller (Traber u. a., 2016), als auch leistungsstarke, Linux fähige 64-Bit Prozessoren (Zaruba und Benini, 2019).

1.1 Beiträge dieser Arbeit

Ziel dieser Arbeit ist es, die praktischen Übungen aus TGI1 auf die RISC-V Architektur und die RISC-V Assemblersprache zu portieren. Um das zu erreichen, wird entweder eine Simulationsumgebung, die einen RISC-V Core emuliert, oder ein Mikrocontroller, der einen RISC-V Core in Hardware verbaut hat, benötigt. Für die Lehre bietet es sich an, beide Möglichkeiten zu haben. So können Studierende zur Vorbereitung Programme im Simulator testen und in praktischen Übungen auf einem Mikrocontroller mit realer Hardware experimentieren. In dieser Arbeit werden mehrere Entwicklungsplattformen vorgestellt und verglichen. Hierbei wird das *Hifive1RevB* genauer betrachtet und anhand dessen die Grundlagen der Assemblerprogrammierung in RISC-V vorgestellt. Die Komplexität und die Eignung dieser Assemblersprache für die Lehre lässt sich in einem Vergleich mit Programmen in dem bisher verwendeten AVR Assembler abschätzen.

1.2 Verwandte Arbeiten

Einige Universitäten verwenden bereits RISC-V in ihren Lehrveranstaltungen. Die University of California, Berkeley unterrichtet den Kurs Computer Architecture[1] mit RISC-V. Auch die Technical University of Denmark[2], die University of Cornell [3] und das Massachusetts Institute of Technology[4] verwenden RISC-V in ihren Veranstaltungen. Allerdings werden in diesen Kursen lediglich Simulationsumgebungen genutzt und eine Programmierung auf realer Hardware findet nicht statt. Dabei kann der Umgang mit Hardware in praktischen Übungen, wie es in TGI1 bisher der Fall ist, für Studierende großes Lernpotenzial bieten. So können Fehler auftreten, die eine Softwaresimulation nicht abbilden kann. Beispielsweise das falsche Verbinden zweier Komponenten oder das Prellen von Schaltern. Da es hierzu in der akademischen Lehre noch kaum bis keine Ansätze auf RISC-V Basis gibt, fokussiert sich diese Arbeit auf eine Realisierung in Hardware.

Die Aktualität und das große Interesse an RISC-V werden ebenfalls durch verschiedene Förderprojekte deutlich. Zum Beispiel wird durch die *European Processor Initiative* ein high-performance Mikroprozessor auf RISC-V Basis entwickelt[5]. Auch in Deutschland werden *Zukunftsfähige Spezialprozessoren und Entwicklungsplattformen* (*ZuSE*)[6] gefördert. Das betrifft insbesondere solche, die auf opensource und lizenzfreien Konzepten basieren.

1.3 Aufbau dieser Arbeit

Das Kapitel 2 betrachtet zunächst grundlegend, was eine ISA ist, welche Arten es gibt und welche Gründe für eine Neuentwicklung sprechen. Es wird gezeigt, welche Neuerungen RISC-V mit sich bringt und wie der Standard aufgebaut ist. Aktuelle Entwicklungen im RISC-V Umfeld werden im Kapitel 3 präsentiert. Es folgt ein Vergleich von unterschiedlichen Simulationsumgebungen, Entwicklungsboards und Entwicklungsumgebungen. Anschließend thematisiert das Kapitel 4 die Programmierung des Entwicklungsboards *Hifive1RevB*. Neben den Grundlagen ist der Umgang mit Peripherie und die Verwendung von Timer- und externen Interrupts dargestellt. Ein Vergleich von AVR-Assembler und RISC-V Assembler dient in Kapitel 5 zur Untersuchung der Tauglichkeit von RISC-V in der Lehre. Dazu sind Konzepte aus der Assembler Programmierung anhand von Beispielen in beiden Assemblersprachen dargestellt.

[1] https://cs61c.org/, Oktober 2019
[2] https://github.com/schoeberl/cae-lab, Oktober 2019
[3] https://www.csl.cornell.edu/courses/ece4750/, Oktober 2019
[4] https://6004.mit.edu/, Oktober 2019
[5] https://www.european-processor-initiative.eu/, Oktober 2019
[6] https://www.bmbf.de/foerderungen/bekanntmachung-2422.html, Oktober 2019

2

RISC-V Instruction Set Architecture

2.1 Instruction Set Architecture (ISA)

Die ISA ist die Schnittstelle zwischen Hardware und low-level Software (Assembler). Dabei abstrahiert die ISA die Funktionalität eines Prozessors anhand von definierten, ausführbaren Befehlen. Das lässt die Art der Hardwareimplementierung frei, garantiert aber Kompatibilität der Software auf unterschiedlicher Hardware. Grundsätzlich unterscheidet man zwischen Complex Instruction Set Computer (CISC)- und Reduced Instruction Set Computer (RISC)-Architektur (Schiffmann, 2006, Kap. 4). Eine CISC-Architektur bietet dem Programmierer eine möglichst einfache Schnittstelle zum Prozessor. Dazu gibt es viele Befehle die zu simpleren Assemblerprogrammen führen. Für die Implementierung in Hardware wird allerdings ein komplexes Mikroprogrammsteuerwerk benötigt (Schiffmann, 2006, Kap. 5). Dem gegenüber steht die RISC-Architektur, die eine möglichst simple und effiziente Hardwareimplementierung bevorzugt. Dafür werden nur simple Adressierungsarten und Datenformate verwendet. Das führt als Konsequenz zu größeren Assemblerprogrammen. Moderne Compiler können aus Hochsprachen wie z. B. *C* sehr effizienten Assemblercode kompilieren (Schiffmann, 2006, Kap. 6).

2.2 RISC-V ISA

Die RISC-V ISA wurde 2010 von der University of California (UC), Berkeley entwickelt. RISC-V ist nach RISC-I, -II, SOAR und SPUR die fünfte RISC-ISA der UC Berkeley (Waterman, 2016, Kap. 3). Sie zielt darauf ab, als universelle ISA alle Größen von Prozessoren zu unterstützen. Durch den neuen Entwurf konnten Designfehler anderer ISAs vermieden werden. Die Offenheit des Standards, der von der nonprofit *RISC-V Foundation* verwaltet wird, ermöglicht das freie Entwickeln von Prozessoren ohne Lizenzabgaben. Das führt unter anderem dazu, dass RISC-V in den letzten Jahren viel Aufmerksamkeit bekommen hat. Das ist auch an den Mitgliedern der RISC-V Foundation zu sehen. Mitglieder sind große Firmen wie zum Beispiel Google, Huawai, IBM, Microsoft, Samsung , NXP, Western Digital und zahlreiche kleinere Firmen (Patterson und Waterman, 2017, Kap. 1).

Warum eine neue ISA?

Waterman vergleicht in seiner Doktorarbeit bestehende ISAs, um zu begründen, warum es einer Neuen bedarf (Waterman, 2016, Kap. 2). Das größte Problem der Etablierten ist, dass das Geschäftsmodell der entwickelnden Firmen auf Intellectual Property (IP) und Lizenzeinnahmen basiert. Möchte man seine eigene CPU entwickeln oder Änderungen an der ISA vornehmen, müssen Lizenzen eingehalten oder sogar erworben werden. Es gibt bereits zwei freie ISAs, diese nennen sich SPARC und OpenRisc. Beiden fehlt es allerdings an wichtigen Strukturen, die für eine moderne ISA fundamental sind. Dazu gehören *Compressed Instructions* (16-Bit breite Befehle), die Code speichereffizienter machen. Das Trennen von Privilege- und User-ISA erleichtert das Virtualisieren von Systemen. Ein weit verbreitetes Feature zum dynamischen Linken von Libraries ist *Position Independent Code*. Schließlich sollte eine moderne ISA mit Fließkommazahlen nach aktuellem Standard (IEEE754-2008) umgehen können. Als Beispiel nennt Waterman die *ARMv8* Architektur, die diese Anforderungen alle erfüllen würde, aber durch die proprietäre Lizenz nicht in Frage kommt. Weiter sind beliebte ISAs sehr komplex und viele Designentscheidungen historisch begründet. Das beste Beispiel dafür ist Intels *80x86* Architektur, die mittlerweile um die 1300 Instruktionen umfasst. Mit einem sauberen Neuanfang hofft Waterman, dies in RISC-V zu vermeiden. Problematisch bei einem solchen Neuanfang ist allerdings der Software-Stack, der für die neue ISA entwickelt werden muss.
Die RISC-V Spezifikation ist modular aufgebaut. Sie teilt sich in verschiedene Basisinstruktionssätze und Erweiterungen auf. Außerdem wird in Privileged- und Unprivileged Architecture unterschieden.

Unprivileged Architecure

Der Basisinstruktionssatz ist die Grundlage für eine RISC-V ISA und muss wie in der Spezifikation vorgegeben implementiert werden. Die Registerbreite in Bits wird mit dem Begriff XLEN beschrieben und ist maßgebend für die Basis-ISA. Das *Instruction Set Manual, Volume 1: Unprivileged ISA* beschreibt die Basisinstruktionssätze *RV32I* und *RV64I*. Dabei steht das **I** für Integer und umfasst den grundlegenden Befehlssatz, den jede RISC-V Implementierung benötigt. Die Basisinstruktionssätze *RV32E*, für Embedded-Devices mit nur 16 Registern, und *RV128I*, sowie viele Erweiterungen sind zu diesem Zeitpunkt noch nicht abgeschlossen. Erweiterungen ergänzen die Basis um weitere Befehle, Register oder Konventionen und können nahezu beliebig angehängt werden. Sie werden mit einem einzelnen Buchstaben abgekürzt, z. B. **C** für Compressed. In Tabelle 2.1 sind diese aufgelistet (Waterman und Asanovic, 2019a).

Erweiterungen, die eine bestimmte Erweiterung ergänzen, haben das Kürzel **Z** gefolgt von dem Kürzel der jeweiligen Erweiterung. Zum Beispiel erweitert *Zicsr* den **I**-Befehlssatz um Instruktionen zur Verwendung von Control Status Register (CSR). Diese sind Register, auf die nur mit bestimmten Berechtigungen zugegriffen werden kann. Sie geben Informationen über den Zustand des Prozessors und bieten u. a. Kontrollmöglichkeiten für Interrupts. Wie ein Zugriff auf diese Register erfolgen kann, wird in Abschnitt 4.3 beschrieben. Da sich Control Status Register (CSR) für Counter verwenden lassen, werden sie bereits in der Unprivileged Spezifikation als Erweiterung eingeführt, auch wenn sie in dieser Arbeit nur in der Privileged Architecure verwendet werden.

Tabelle 2.1: Basisinstruktionssätze und Erweiterungen des RISCV-Standards (Waterman und Asanovic, 2019a)

Basisinstruktionssatz	XLEN	Register	Status
RV32I	32	32	Ratifiziert
RV64I	64	32	Ratifiziert
RV32E	32	16	Entwurf
RV128I	128	32	Entwurf
RVWMO	Weak Memory Ordering Consistency-Model		Ratifiziert

Erweiterung	Bedeutung	Status
A	Atomic	Frozen
B	Bit Manipulation	Entwurf
C	Compressed	Ratifiziert
D	Double-Precision Floating-Point	Ratifiziert
F	Single-Precision Floating-Point	Ratifiziert
G	Überbegriff für I, M, A, D, Zifencei	Ratifiziert
J	Dynamically Translated Languages	Entwurf
L	Decimal Floating-Point	Entwurf
M	Integer Multiply, Divide	Ratifiziert
N	User-Level Interrupts	Entwurf
P	Packed-SIMD Instructions	Entwurf
Q	Quad-Precision Floating-Point	Ratifiziert
T	Transactional Memory	Entwurf
V	Vector Extensions	Entwurf
Zam	Misaligned Atomics	Entwurf
Zicsr	Control Status Register Access	Ratifiziert
Zifencei	Instruction-Fetch Fence	Ratifiziert
Ztso	Total Store Ordering Memory-Consistency-Model	Frozen
Z	Machine-level extension	Ratifiziert
S	Supervisor-level extension	Ratifiziert
H	Hypervisor-level extension	Entwurf
X	Non-standard extension	-

Privileged Architecture

Der zweite Teil *Volume II: Privileged Architecture* führt Strukturen und Befehle ein, die zur Kommunikation mit externen Systemen und für Betriebssysteme notwendig sind (Waterman und Asanovic, 2019b). Die Privileged Architecture sieht die Berechtigungsstufen User-, Supervisor- und Machine-Mode als Abstraktionsstufen zwischen Softwarekomponenten und direktem Hardwarezugriff vor (siehe Tabelle 2.1). Somit können in einer vertrauensvollen Umgebung unbekannte und potenziell nicht vertrauensvolle Anwendungen ausgeführt werden. Vor allem Betriebssysteme machen von diesem Konzept Gebrauch. In dieser Arbeit wird nur der Machine-Mode betrachtet, welcher die höchste Berechtigungsstufe ist. Dieser muss in einer Privileged Architecture immer implementiert sein und bietet den vollen Zugriff auf alle Hardwarekomponenten. Physical Memory Attributes und Physical Memory Protection werden ebenfalls in der Privileged Architecture beschrieben, sind jedoch nicht Teil dieser Arbeit. Weitere Spezifikationen wie der *External Debug Support* oder die *Processor Trace Specification* befinden sich in einem Entwurfsstatus.

3

Stand der Technik

Dieses Kapitel betrachtet aktuelle Entwicklungen aus dem RISC-V Umfeld, die für die Lehre relevant sind. Dazu werden Hardware- und Softwarekomponenten vorgestellt, die zur Programmierung eines RISC-V Prozessors nötig sind. Das Kernelement ist dabei der *Core*. Er beinhaltet den Prozessor und ist meistens in einem System on a Chip (SoC) mit weiterer Peripherie integriert. Um auf einen SoC leichter zugreifen zu können, werden so genannte *Entwicklungsboards* verwendet. Diese führen die Pins des SoC heraus und stellen unterstützende Komponenten, wie z. B. Spannungsregler, Programmier- und Debugschnittstellen, bereit. Eine *Entwicklungsumgebung* bietet für die Programmierung hilfreiche Tools und erleichtert den Umgang mit dem SoC. Liegt einem die Hardware nicht direkt vor, wie es beispielsweise bei Studierenden in der Vorbereitung auf eine praktische Übung der Fall ist, ist eine Simulationsumgebung von großem Vorteil. Diese stellt das Verhalten des Prozessors nach und ermöglicht das Testen von Programmen.

3.1 Simulation

Simulatoren werden in die beiden Gruppen funktionelle und Timing Simulatoren eingeteilt. Funktionelle Simulatoren implementieren nur die funktionelle Architektur und stellen das Verhalten eines Systems nach. Da sie meistens keine Mikroarchitektur implementieren, sind funktionelle Simulatoren oft schneller, können aber weniger Informationen über z. B. Timing oder Strombedarf geben. Neben dem Speicher, der Peripherie und dem Zustand eines Systems besteht ein funktioneller Simulator hauptsächlich aus einem Instruction Set Simulator (ISS). Dieser interpretiert die Instruktionen und speichert Inhalte von Registern (Akram und Sawalha, 2019, Sec. 2). Eine Möglichkeit zur Simulation eines Prozessors inklusive Mikroarchitektur ist die Softcoreimplementierung auf einem Field-Programmable Gate Array (FPGA). Dieser kann Clock-Cycle genau simulieren und bietet Zugriff auf Hardwareperipherie.

Für das Simulieren von Programmen zur Vorbereitung auf die Lehrveranstaltung genügt den Studierenden ein funktionales Modell, mit dem einfache Assemblerprogramme kompiliert und Schritt-für-Schritt ausgeführt werden können. Hilfreich ist dabei ein Graphical User Interface (GUI), das den Zustand des Prozessors, wie z. B. Register, Speicherinhalte usw., anzeigt.

Nachfolgend werden einige für die Lehre interessante Simulatoren aufgelistet. Eine

vollständige Liste kann auf den Seiten der *RISC-V Foundation* gefunden werden[7].

- **Spike** ist ein ISS der häufig als Referenzimplementierung der RISC-V Spezifikation gesehen wird. In Kombination mit dem RISC-V Proxy Kernel und einer Reihe an Instruction Tests wird Spike häufig als Testumgebung für neue Coreimplementierungen verwendet [8].
- **QEMU**, der Quick Emulator, ist bereits in anderen Architekturen weit verbreitet und unterstützt RISC-V in Fullsystem oder Usermode Emulation[9]. Zum Debuggen kann *GDB* verwendet werden[10]. Als graphische Oberfläche kann beispielsweise *GDBGUI* benutzt werden[11].
- **RARS** und **Jupiter** sind beides ISS mit einem GUI und basieren auf Java[12] [13].
- **RIPES** emuliert graphisch einen 5-Stage Pipeline Prozessor und stellt die einzelnen Phasen der Pipeline dar[14].
- **venus** und **briscv** sind Webbrowser basierte ISS. Sie bieten den großen Vorteil, dass Studierende keine Software installieren müssen und sehr leicht zu verwenden sind[15][16]

3.2 Entwicklungsboards

In der Lehrveranstaltung TGI1 sollen Studierende in praktischen Übungen Assemblerprogramme auf einem Mikrocontroller ausführen. Dabei soll unterschiedliche Peripherie, wie z. B. Light-Emitting Diodes (LEDs), Schalter, 7-Segment Anzeigen und ein Liquid Crystal Display (LCD), verwendet werden. Ein Entwicklungsboard hat meistens einen SoC und verschiedene Peripherie verbaut. Ein SoC vereinigt auf einem Chip den Prozessor (Core), den Bus, unterschiedliche Speicher und weitere Schnittstellen. Ist ein Core in Hardware realisiert, wird er Hardcore genannt. Andererseits kann er auch in Software (Softcore) auf einem FPGA ausgeführt werden (Brinkschulte und Ungerer, 2010, Kap. 3.6.1). Für die Lehre ist es dabei interessant, einen möglichst simplen und leicht verständlichen Core mit ausreichend Peripherie für verschiedene Experimente zur Verfügung zu stellen. Tabelle 3.1 zeigt eine Auswahl von aktuellen Entwicklungsboards mit einem RISC-V Core. Auf der Seite der RISC-V Foundation ist eine vollständige Liste zu finden[17].

[7] https://riscv.org/software-status/, September 2019
[8] https://github.com/riscv/riscv-tools, September 2019
[9] https://www.qemu.org/, September 2019
[10] https://www.gnu.org/software/gdb/, September 2019
[11] https://www.gdbgui.com/, September 2019
[12] https://github.com/thethirdone/rars, September 2019
[13] https://github.com/andrescv/Jupiter, September 2019
[14] https://github.com/mortbopet/Ripes, September 2019
[15] https://github.com/kvakil/venus, September 2019
[16] http://ascslab.org/research/briscv/simulator/simulator.html, September 2019
[17] https://riscv.org/risc-v-cores/, September 2019

Tabelle 3.1: Entwicklungsboards mit einem RISC-V Core.

Entwicklungsboard	SoC	Core	Architektur	Hersteller
-	Raven	PicoRV32	RV32IMAC	efabless
GAPUino	GAP-8	PULP, 9 verschiedene	RV32IMC	Greenwaves
Sispeed Maix	K210	2x K210	RV64IMAC	Sispeed, Kendryte
VegaBoard	RV32M1	RI5CY, Zero-RI5CY, 2x ARM	RV32IMC	NXP
Hifive1RevB	FE310-G002	E31	RV32IMAC	SiFive
Basys 3	Artix 7 (FPGA)	bel. Softcore	bel.	Digilent, Xilinx

Raven

Der *Raven* SoC [18] implementiert den PicoRV32 [19] Core. Der SoC ist relativ klein und hat eine gut überschaubare Architektur. Der größte Vorteil ist, dass beide Implementierungen open-source sind. Da es allerdings zu diesem Zeitpunkt kein Entwicklungsboard gibt, konnte der Raven SoC nicht getestet werden.

GAPUino

Das *GAPUino* Entwicklungsboard verbaut den GAP-8 SoC[20]. Neben einem Mikrocontroller hat der GAP-8 acht weitere Cores für parallele Berechnungen. Die Cores basieren auf der Parallel Ultra-Low-Power Processing-Platform (PULP)[21]. Durch die komplexere Architektur und eine sehr geringe Dokumentation ist der GAP-8 für eine low-level Programmierung mit Assembler nur geringfügig brauchbar.

Sipeed Maix

Das Entwicklungsboard *Sipeed Maix*[22] hat zusätzlich zu zwei K210 RISC-V Cores einen Neural Network Processor, Audio Processor und weitere Peripherie. Es ist zur Programmierung mit Python ausgerichtet und hat in der Dokumentation nicht ausreichend Informationen für eine Programmierung auf Assemblerebene.

VegaBoard

Das VegaBoard[23] hat gleich vier Cores verbaut. Neben zwei ARM Cores sind der Zero-RI5CY und der RI5CY Core des PULP Projekts verbaut. Obwohl der Zero-RI5CY eine sehr reduzierte und für die Lehre vielversprechende Architektur hat, bringt das VegaBoard zu viel Komplexität mit sich. Da es gleich vier Prozessoren unterstützt, fasst das Reference Manual mehr als 4400 Seiten. Ein einfaches C-Projekt besteht bereits aus drei Dateien für die Konfiguration des Boards. Der Buildablauf ist undurchschaubar und für die Verwendung der C-Library besteht die Dokumentation nur aus wenigen Beispielen.

[18] https://github.com/efabless/raven-picorv32, September 2019
[19] https://github.com/cliffordwolf/picorv32, September 2019
[20] https://greenwaves-technologies.com/manuals/BUILD/HOME/html/index.html, September 2019
[21] http://iis-projects.ee.ethz.ch/index.php/PULP, September 2019
[22] https://wiki.sipeed.com/en/maix/board/, September 2019
[23] https://open-isa.org/, September 2019

Hifive1RevB

Das Entwicklungsboard *Hifive1RevB*[24] hat den SoC FE310-G002 verbaut. Abbildung 3.2 zeigt eine Übersicht des Entwicklungsboards und Abbildung 3.3 enthält ein Blockdiagramm des FE310-G002. Der FE310-G002 unterstützt den Instruktionssatz *RV32IMAC* (32-Bit, Integer, Multiply, Atomic, Compressed) inklusive der Privileged Architecture mit dem *Machine* und *User Mode*. Der General-Purpose Input/Output (GPIO)-Complex unterstützt außerdem Universal Asynchronous Receiver Transmitter (UART), Pulse Width Modulation (PWM), Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C) und Joint Test Action Group (JTAG). Auf dem Hifive1RevB befindet sich zusätzlich ein ESP32[25], ein SPI-Flash und JLinks Universal Serial Bus (USB)-Debugger[26]. Im Gegensatz zu den anderen bisher betrachteten Entwicklungsboards hat dieses eine weniger komplexe Architektur. Mit nur einem Core und grundlegender Peripherie wirkt das Hifive1 am vielversprechendsten, weswegen es in dieser Arbeit auf Tauglichkeit untersucht wird. In Abschnitt 4.2 ist die Programmierung genauer beschrieben.

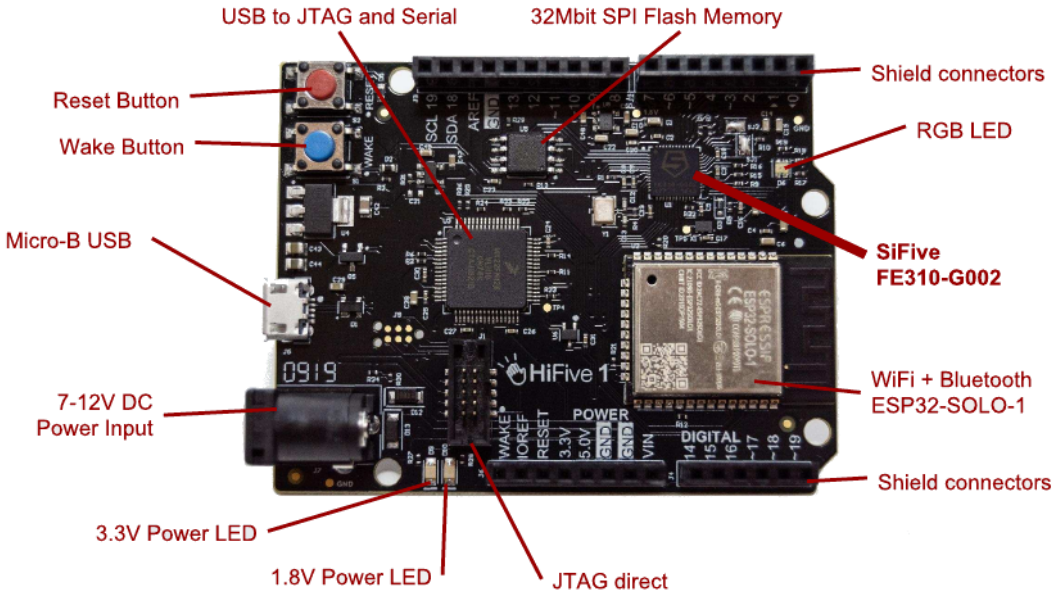

Abbildung 3.2: Übersicht des Hifive1RevB Entwicklungsboard (*SiFive HiFive1 Rev B Getting Started Guide* 2019)

[24] https://www.sifive.com/boards/hifive1-rev-b, September 2019
[25] Der ESP32 ist ein Mikrocontroller zur WiFi- und Bluetoothkommunikation und kann via SPI angesteuert werden. https://www.espressif.com/en/products/hardware/esp32/overview, November 2019
[26] https://www.segger.com/products/debug-probes/j-link/, November 2019

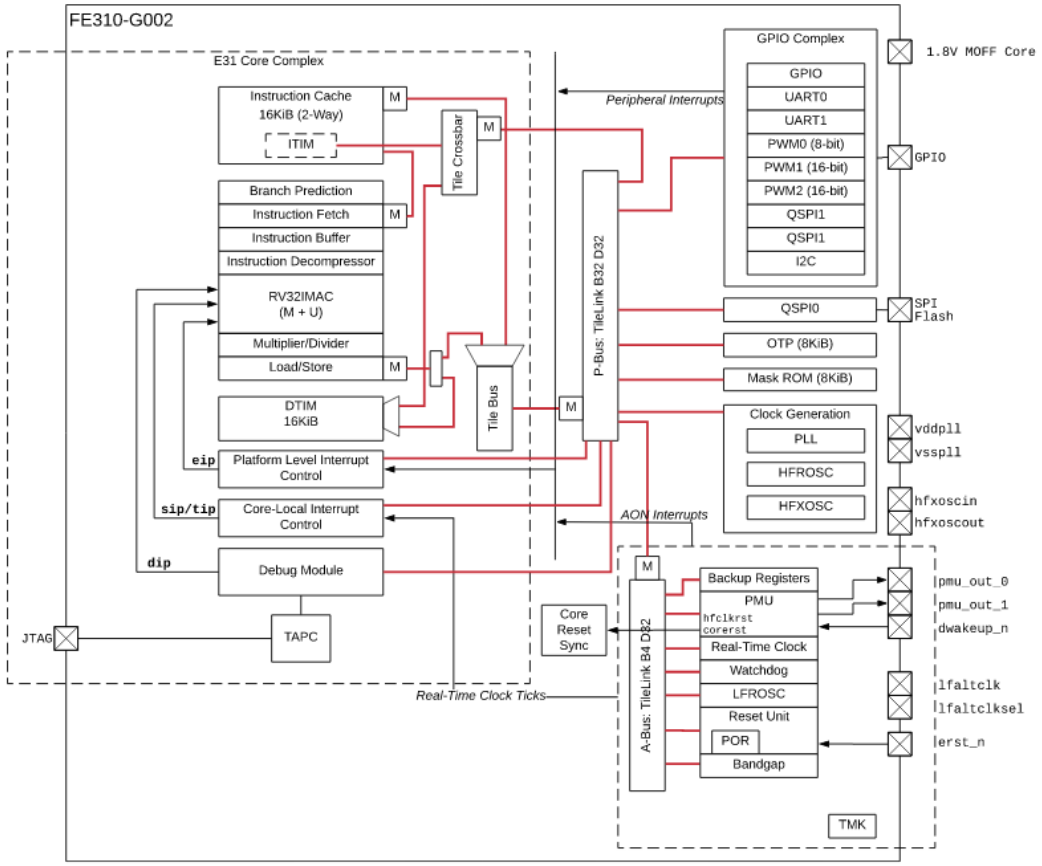

Abbildung 3.3: Blockdiagramm des FE310-G002 (*SiFive FE310-G002 Manual* 2019)

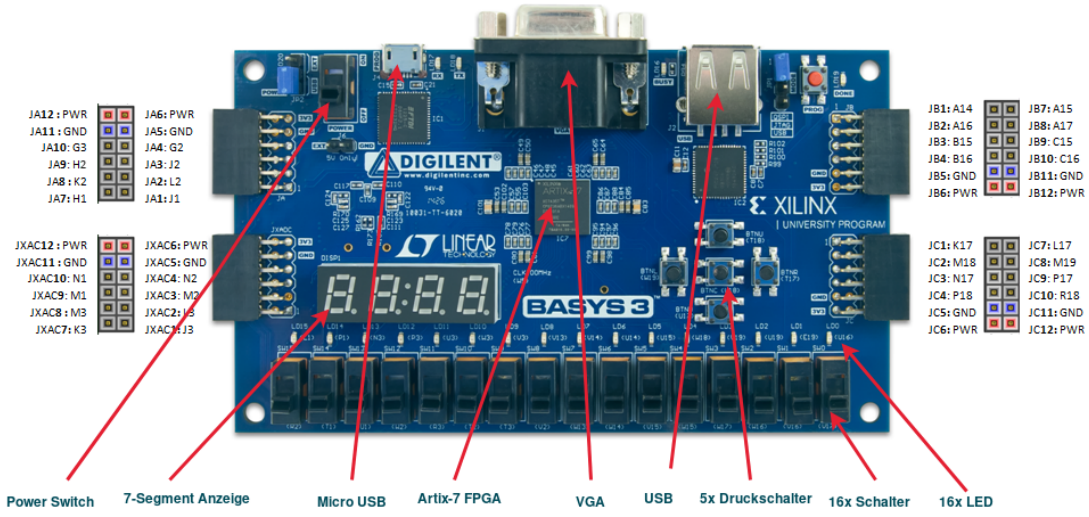

Abbildung 3.4: Übersicht des Basys3 Entwicklungsboard[27]

Basys 3

Das *Basys 3* ist ein Entwicklungsboard mit einem Xilinx Artix-7 FPGA[27]. Es hat unterschiedliche Peripherie direkt verbaut und kann mit kleineren Softcores programmiert werden, was eine flexible Alternative zu einem Hardcore darstellt. Diese Variante ist in Abschnitt 5.5 dargestellt. Abbildung 3.4 zeigt das Basys3 Board.

3.3 Entwicklungsumgebung

Eine Entwicklungsumgebung integriert verschiedene Tools, die dem Programmierer die Entwicklung von Software erleichtert.

Toolchain

Eine Toolchain ist eine Sammlung von Programmen zur Entwicklung von Software. In dieser Arbeit wurde die GNU Toolchain verwendet. Sie besteht aus der GNU Compiler Collection (gcc), den GNU Binutils (bestehend aus Assembler, Linker und weiteren Programmen zum Umgang mit Binärdateien), der GNU C Library (glibc), dem GNU Debugger (gdb) und weiteren Programmen für Buildabläufe und Ähnlichem[28]. Andere Compiler wie z. B. LLVM unterstützten ebenfalls bereits RISC-V[29].

Software Development Kit (SDK)

Ein Software Development Kit (SDK) beinhaltet eine Toolchain und für bestimmte Hardware angepasste Tools. Zur Entwicklung auf dem Hifive1RevB bietet SiFive die *Freedom-E-SDK*[30] an. Damit können Projekte für verschiedene Plattformen der Freedom-E Reihe entwickelt werden und mit Tools von JLink kann das Entwicklungsboard direkt über die USB-Schnittstelle programmiert und debuggt werden.

Integrated Development Environment (IDE)

Eine Integrated Development Environment (IDE) ist eine Oberfläche, die einen leichteren Zugriff auf das SDK bietet. Für das Hifive1 gibt es als grafische Entwicklungsumgebung eine Eclipse Portierung, das Freedomstudio[31]. Neben den Funktionen des SDK gibt es eine Integration für QEMU, womit Programme simuliert werden können. Da allerdings das Debugging im Freedomstudio teilweise fehlerhaft ist, bietet es sich an, den JLinkDebugger direkt mit Ozone[32] zu verwenden. Eine weitere Möglichkeit ist die IDE VSCode[33] mit dem Plugin PlatformIO[34], das ebenfalls eine Vielzahl von Entwicklungsboards unterstützt, zu nutzen.

[27] https://reference.digilentinc.com/reference/programmable-logic/basys-3/start, September 2019
[28] https://www.gnu.org/manual/blurbs.html, Oktober 2019
[29] https://riscv.org/software-status/#c-compilers-and-libraries, Oktober 2019
[30] https://github.com/sifive/freedom-e-sdk, Oktober 2019
[31] https://www.sifive.com/boards#software, Oktober 2019
[32] https://www.segger.com/products/debug-probes/j-link/, Oktober 2019
[33] https://code.visualstudio.com/, Oktober 2019
[34] https://platformio.org/, Oktober 2019

4

RISC-V Assembler

Der erste Abschnitt dieses Kapitels präsentiert plattformunabhängige Grundlagen der Assemblerprogrammierung aus dem ersten Teil der RISC-V Spezifikation. Danach folgt die Präsentation des Umgangs mit Peripherie beispielhaft am *FE310*, der auf dem Hifive1 Entwicklungsboard verbaut ist. Schließlich werden Timer- und Externe Interrupts vorgestellt. Diese sind grundlegend im zweiten Teil der RISC-V Spezifikation definiert und werden plattformspezifisch implementiert. Anleitungen zum Einrichten einer Entwicklungsumgebung und zur Programmierung des Boards, sowie Verweise zur Dokumentation und hilfreichen Internetseiten sind im Anhang A aufgeführt.

4.1 Grundlagen

Die Grundlagen für die Assemblerprogrammierung auf einer RISC-V Architektur werden hier anhand des im ersten Teil der RISC-V Spezifikation (Waterman und Asanovic, 2019a) vorgestellten *Base Integer Instruction Set RV32I* gezeigt. Um den Einstieg in RISC-V Assembler einfach zu halten, werden andere Erweiterungen vorerst nicht behandelt. Ein grundlegendes Element sind Register. Diese halten Werte, auf denen Operationen ausgeführt werden können, temporär fest. Weiterhin wird gezeigt, wie diese Werte aus dem Speicher in ein Register geladen oder aus einem Regster in den Speicher geschrieben werden können. Sprungbefehle kontrollieren den Programmfluss und können an Bedingungen geknüpft sein. Schließlich werden Mittel gezeigt, die die Lesbarkeit und die Struktur von Assemblerprogrammen verbessern. Dazu gehören Funktionalitäten des Compilers (Assemblerdirektiven, Präprozessor und Pseudoinstruktionen), sowie die Konzepte von Funktionen und des Stacks.

Register

Nach der RISC-V Spezifikation sind 32 General-purpose Register vorgesehen. Auf ihnen können alle grundlegenden Befehle ausgeführt werden. Die Register haben eine Breite von 32-Bit. Bezeichnet werden sie entweder mit *x0-x31* oder nach einer Binärschnittstellen-Konvention (Application Binary Interface (ABI)), die von einem Compiler übersetzt wird. Diese legt die Verwendungszwecke für die jeweiligen Register fest. In Tabelle 4.1 sind die durch die ABI spezifizierten Registerbezeichnungen aufgelistet. Hervorgehoben werden

soll das Register *x0*. Es gibt per Definition festverdrahtet den Wert *0* zurück. Dies kann in der Assemblerprogrammierung oft hilfreich sein und ist die Grundlage für viele Pseudoinstruktionen. Das Register *x1* wird in den meisten Fällen bei Sprungbefehlen zum Sichern der Rücksprungadresse verwendet. Der Stackpointer *x2* zeigt auf das Ende des Stacks, dahingegen zeigt der Framepointer (*x8*) auf den Beginn eines Stackframes. Der Globalpointer (*x3*) zeigt auf einen Speicherbereich für globale Variablen und der Threadpointer (*x4*) zeigt auf den Speicherplatz für lokale Variablen des (aktuellen) Threads. Der Program Counter (PC) ist ein zusätzliches Register und kann nur über Jump- oder Branch-Befehle verändert werden.

Tabelle 4.1: Register nach RV32I ABI. Einige Register werden von aufgerufenen Funktionen gesichert und haben nach einem Funktionsaufruf den selben Wert wie vorher.

Register	ABI	Verwendung nach Konvention	Gesichert?
x0	zero	fest verdrahtet 0, ignoriert Schreibzugriffe	-
x1	ra	Rücksprungadresse	Nein
x2	sp	Stackpointer	Ja
x3	gp	Globalpointer	-
x4	tp	Threadpointer	-
x5 - x7	t0 - t2	Temporäre Register 0 - 2	Nein
x8	s0 *oder* fp	Gesichertes Register 0 *oder* Frame Pointer	Ja
x9	s1	Gesichertes Register 1	Ja
x10	a0	Rückgabewert *oder* Funktionsargument 0	Nein
x11	a1	Rückgabewert *oder* Funktionsargument 1	Nein
x12 - x17	a2 - a7	Funktionsargumente 2 - 7	Nein
x18 - x27	s2 - s11	Gesicherte Register 2 - 11	Ja
x28 - x31	t3 - t6	Temporäre Register 3 - 6	Nein
pc	pc	Program Counter	-

Befehlstypen

Der Aufbau eines Assemblerbefehls ist abhängig von dem jeweiligen Befehlstyp. Es wird zwischen Register-Register, Register-Immediate, Store, Branch, Upper-Immediate und Jump Befehlen unterschieden. Die Kodierung der Befehlsformate wird hier nicht weiter behandelt. Tabelle 4.2 zeigt den für die Assemblerprogrammierung relevanten Befehlsaufbau. Hierbei wird das Destination Register mit *rd*, die Source Register mit *rs1*, *rs2* und Immediates (konstante numerische Werte) mit *imm*[*Anzahl Bits*] oder *offset*[*Anzahl Bits*] abgekürzt. Der Store Befehlstyp addiert den Offset auf das Source Register *rs1*. Dahingegen wird beim Jump Befehlstyp das PC Register als Grundlage verwendet.

Arithmetische, logische und Shift-Befehle

Der Basis Integer Instruktionssatz (RV32I) beinhaltet die in Tabelle 4.3 aufgelisteten Befehle für arithmetische, logische und Shift-Operationen mit Register-Register und Register-Immediate (direkter) Adressierung. Befehle mit Immediate Adressierung können Konstanten bis zu einer Größe von $2^{12} - 1$ verarbeiten. Da es nur einzeilige Instruktionen

Tabelle 4.2: Befehlstypen

Befehlstyp	Felder		
Register:	rd,	rs1,	rs2
Immediate:	rd,	rs1,	imm[12]
Store:	rs2,	offset[12](rs1)	
Branch:	rs1,	rs2,	offset[12]
Upper Immediate:	rd,	imm[20]	
Jump:	rd,	offset[20]	

gibt, ist für größere Immediates ein zusätzlicher Befehl vorgesehen. Mit dem Load Upper Immediate (`lui`) Befehl kann ein 20-Bit Wert in ein Register geladen und um 12-Bit nach links geshiftet werden. Danach können die unteren 12-Bit mit dem `addi` Befehl addiert werden. In der Summe ergibt sich ein bis zu 32-Bit großer Wert.

Bei Shift-Operationen unterscheidet man logische und arithmetische Shifts. Logische schieben eine Null an das frei werdende Bit, Arithmetische verwenden den Wert des alten Bits. Für den arithmetischen Rechtsshift gilt also, dass das alte Most Significant Bit (MSB) an die Stelle des neuen MSB eingefügt wird (Patterson und Hennessy, 2017, Kap. 2). Beispiel 4.4 zeigt exemplarisch eine Berechnung mit diesen Befehlen.

Tabelle 4.3: Logische, arithmetische und Shift-Befehle in RV32I

Instruktion					Bedeutung
Add	add	rd,	rs1,	rs2	$rd = rs1 + rs2$
Add Imm.	addi	rd,	rs1,	imm	$rd = rs1 + imm$
Sub	sub	rd,	rs1,	rs2	$rd = rs1 - rs2$
And	and	rd,	rs1,	rs2	$rd = rs1 \wedge rs2$
And Imm.	andi	rd,	rs1,	imm	$rd = rs1 \wedge imm$
Inklusiv or	or	rd,	rs1,	rs2	$rd = rs1 \vee rs2$
Inklusiv or Imm.	ori	rd,	rs1,	imm	$rd = rs1 \vee imm$
Exklusiv or	xor	rd,	rs1,	rs2	$rd = rs1 \oplus rs2$
Exklusiv or Imm.	xori	rd,	rs1,	imm	$rd = rs1 \oplus imm$
Logischer Linksshift	sll	rd,	rs1,	rs2	$rd = rs1 \ll rs2$
Logischer Linksshift Imm.	slli	rd,	rs2,	imm	$rd = rs1 \ll imm$
Logischer Rechtsshift	srl	rd,	rs1,	rs2	$rd = rs1 \gg rs2$
Logischer Rechtsshift Imm.	srli	rd,	rs2,	imm	$rd = rs1 \gg imm$
Arithmetischer Rechtsshift	sra	rd,	rs1,	rs2	$rd = rs1 \ggg rs2$
Arithmetischer Rechtsshift Imm.	srai	rd,	rs1,	imm	$rd = rs1 \ggg imm$
Set Less Than	slt	rd,	rs1,	rs2	$rd = rs1 < rs2$
Set Less Than Imm.	slti	rd,	rs1,	imm	$rd = rs1 < imm$
Set Less Than Unsigned	sltu	rd,	rs1,	rs2	$rd = rs1 < rs2$
Set Less Than Imm. Unsigned	sltiu	rd,	rs1,	imm	$rd = rs1 < imm$
Load Upper Imm.	lui	rd,	imm		$rd = imm \ll 12$

Beispiel 4.4: Beispielhafte Berechnung mit arithmetischen und logischen Befehlen

```
addi t0, zero, 42  // t0 = 0 + 42          (t0: 0b101010)
addi t1, zero, 19  // t1 = 0 + 19          (t1: 0b010011)
sub  a0, t0, t1    // a0 = t0 - t1         (a0: 0b010111)
andi a0, a0, 0b101 // a0 = a0 & 0b101      (a0: 0b000101)
add  a1, a0, zero  // a1 = a0 + 0          (a1: 0b000101)
slti a0, a1, 0b111 // a0 = (a1 < 0b111)    (a0: 0b000001)

lui  t0, 0x92345   // t0 = 0x92345 << 12   (t0: 0x9234 5000)
addi t0, 0x678     // t0 = t0 + 0x678      (t0: 0x9234 5678)
                   //                  (t0: 0b10010010 00110100 010101100 1111000)
srli t2, t0, 1     // t2 = t0 >> 1     (t2: 0b01001001 00011010 001010110 0111100)
srai t3, t0, 1     // t3 = t0 >>> 1    (t3: 0b11001001 00011010 001010110 0111100)
```

Tabelle 4.5: Befehle für Speicherzugriffe in RV32I

Instruktion				Bedeutung
Load {byte, half, word}	l{b,h,w}	rd,	offset(rs1)	$rd = memory[rs1 + imm]$
Load {byte, half} unsigned	l{b,h}u	rd,	offset(rs1)	$rd = memory[rs1 + imm]$
Store {byte, half, word}	s{b,h,w}	rs2,	offset(rs1)	$memory[rs1 + imm] = rs2$

Speicherzugriffe

Speicherzugriffe sind durch die Load- und Store-Befehle realisiert. Zuerst muss in ein Register (*rs1*) die Speicheradresse geladen werden. Dann kann durch eine Konstante ein Offset angegeben und von der resultierenden Adresse Werte im Speicher geladen oder geschrieben werden. Es ist möglich die Anzahl der zu schreibenden oder zu lesenden Bytes entsprechend Tabelle 4.5 über den Suffix des Befehls zu bestimmen: Byte **b** (8-Bit), Half **h** (16-Bit), Word **w** (32-Bit). Zu beachten ist, dass es sich um eine byteweise Adressierung handelt. Möchte man also das nächste Word adressieren, muss ein Offset von vier Byte gewählt werden. Die im Speicher verwendete Byte-Reihenfolge ist *little-endian*[35]. Bei einem load byte unsigned (`lbu`) oder half word (`lhu`) werden Bits des Registers, die höher als der geladene Wert sind, mit Nullen gefüllt. Standardmäßig erfolgt ein signed load, der diese mit dem MSB füllt. Beispiel 4.6 zeigt exemplarisch einen Speicherinhalt und verschiedene Zugriffe, sowie den resultierenden Wert der Register.

Sprungbefehle

Zur Kontrolle des Programmablaufs werden die so genannten Jump- und Branch-Befehle verwendet. Jumps verändern den PC ohne Bedingung. Dies geschieht entweder relativ zum PC mit einem Offset (`jal`) oder zu einer absoluten Adresse (`jalr`), indem zuvor in ein Register (*rs1*) die 32-Bit Adresse geladen wird. Ein Offset kann durch ein Symbol

[35] Bei einer little-endian Bytereihenfolge wird das niedrigste Byte an die Anfangsadresse und entsprechend das höchste Byte an die Endadresse geschrieben.

Beispiel 4.6: Beispiel für Speicherzugriffe

```
// Speicherinhalt:
// Addr    0    1    2    3
//  96: 0xaa 0xbb 0xcc 0xdd
// 100: 0xf0 0x00 0x00 0x00

addi t0, zero, 100
lb   t1, 0(t0)      //t1: 0xfffffff0 sign extended
lbu  t2, 0(t0)      //t2: 0x000000f0 unsigned
addi t0, zero, 96   //t0: 0x00000060
lw   t3, 0(t0)      //t3: 0xddccbbaa little-endian
lh   t4, 0(t0)      //t4: 0xffffbbaa sign extended
lh   t5, 2(t0)      //t5: 0xffffddcc 2 bytes offset
```

angegeben werden, welches der Compiler durch den korrekten Offset ersetzt. Im Zielregister (*rd*) kann der aktuelle PC gesichert werden. Dafür wird in der Regel das Register *ra* verwendet oder *zero*, falls die Adresse verworfen werden soll. Beim `jalr` ist zu beachten, dass das Least Significant Bit (LSB) verworfen wird, um nur 16-Bit (compressed Befehle) und 32-Bit Adressierungen zu erlauben. In Kombination mit einem vorherigen `auipc` kann `jalr` verwendet werden, um an eine absolute 32-Bit Adresse zu springen. Dabei wird ein anzugebendes 20-Bit Immediate um 12-Bit nach links geshiftet und mit dem aktuellen Wert des PCs addiert. Mit dem 12-Bit Offset im `jalr` Befehl ergeben sich insgesamt 32-Bit.

Branches überprüfen ob eine Bedingung, wie beispielsweise die Gleichheit zweier Register, gilt und addieren den PC mit dem 12-Bit Offset - sofern die Bedingung erfüllt ist. Andernfalls wird der PC um vier erhöht. Tabelle 4.7 listet diese Befehle auf und Beispiel 4.8 zeigt eine Multiplikationsschleife.

Tabelle 4.7: Sprungbefehle in RV32I

Instruktion					Bedeutung
Jump and Link	jal	rd,	offset[12]		rd=pc+4; pc=pc+offset
Jump and Link Register	jalr	rd,	rs1,	offset[12]	rd=pc+4; pc=(rs1+offset)$\wedge -2$
Branch Equal	beq	rs1,	rs2,	offset[12]	if rs1 == rs2 {pc=pc+offset}
Branch Not Equal	bne	rs1,	rs2,	offset[12]	if rs1 != rs2 {pc=pc+offset}
Branch Less Than	blt	rs1,	rs2,	offset[12]	if rs1 $<$ rs2 {pc=pc+offset}
Branch Greater Than Equal	bge	rs1,	rs2,	offset[12]	if rs1 $\geq$ rs2 {pc=pc+offset}
Branch Less Than Unsigned	bltu	rs1,	rs2,	offset[12]	if rs1 $<$ rs2 {pc=pc+offset}
Branch Greater Than Equal Unsigned	bgeu	rs1,	rs2,	offset[12]	if rs1 $\geq$ rs2 {pc=pc+offset}
Add Upper Immediate To PC	auipc	rd,	offset[20]		rd = pc + (offset $\ll$ 12)

Beispiel 4.8: Beispiel einer iterativen Multiplikation

```
addi a0, zero, 3    // a0 = 0 + 3
addi a1, zero, 2    // a1 = 0 + 2
loop:
  beq zero, a0, end // if(a0 == 0) {pc += offset(end)} else {pc += 4}
  add a2, a2, a1    // a2 = a2 + a1
  addi a0, a0, -1   // a0--
  jal zero, loop    // zero=pc; pc += offset(loop)
end:
```

Pseudoinstruktionen

Eine RISC-Architektur zeichnet sich durch eine geringe Menge an Befehlen aus. Das vereinfacht und optimiert die Implementierung in Hardware, führt aber oft zu komplexerer Software. Compiler und die Programmierung in Hochsprachen können die Komplexität auf Softwareseite sehr gut auffangen. Möchte man direkt in der Assemblersprache programmieren, bietet der Assembler hilfreiche Pseudoinstruktionen. Sie werden in eine oder mehrere echte Instruktionen übersetzt. Häufig verwendet sind folgende:

`li rd, imm[32]:`	Load Immediate, lädt den konstanten Wert in *rd*
`la rd, symbol:`	Load Address, lädt die Adresse des Symbols in *rd*
`j offset:`	Jump, springt zum PC + Offset und verwirft die Rücksprungadresse.
`call offset:`	Funktionsaufruf, sichert die Rücksprungadresse in *ra*.
`ret:`	Return, springt zu der Adresse im Register *ra* und verwirft die Rücksprungadresse.

Eine vollständige Auflistung und wie die Pseudoinstruktionen übersetzt werden befindet sich im Anhang B.

Präprozessor Direktiven

Bevor der Assembler (GNU-AS) ein Assemblerprogramm interpretiert und in Binärdaten konvertiert, kann es vom C-Präprozessor vorverarbeitet werden. Die Dateiendung '.S' indiziert dies. Dagegen werden Dateien mit '.s' direkt vom Assembler übersetzt. Der Präprozessor ermöglicht das Einbinden von Header-Dateien, die Verwendung von Makros und die bedingte Kompilierung. Der Befehl `#include "random.S"` bindet die Datei aus dem Dateipfad des kompilierten Programms ein. Mit der Notation `#include <gpio.h>` sucht der Präprozessor die Datei in Pfaden für Header-Dateien. Der Befehl `gcc -I pfad` kann einen Ordner zu diesem hinzufügen. Mit der `#define` Direktive werden Makros erstellt. Makros können die Lesbarkeit eines Assemblerprogrammes verbessern, indem Konstanten, Register oder komplexere Ausdrücke durch ein Label definiert werden. Um Fehler bei der Makroersetzung durch mehrfaches Einbinden eines Makros zu vermeiden, wird üblicherweise ein `#ifndef` mit `#endif` am Ende eingefügt. Der Präprozessor verarbeitet Befehle in diesem Block nur, wenn das Label noch nicht definiert wurde. Weitere

Direktiven können im *GNU-AS Manual* [36] nachgeschlagen werden.

Assembler Direktiven

Dem Assembler können verschiedene Direktiven erteilt werden. Unter anderem ermöglichen diese das Ablegen von Daten im Speicher. Abhängig von dem Verwendungszweck können dazu unterschiedliche Sektionen benutzt werden, die verschiedene Zugriffsrechte erlauben. Dabei ist eine Sektion ein Platzhalter, der durch den Linker an einen im Linkerskript definierten Speicherbereich gesetzt wird. Die Direktive `.section` leitet einen Befehl zur Auswahl einer Sektion ein und muss von einer dieser Direktiven gefolgt sein:

`.text`, Read-Only Sektion für ausführbaren Code
`.data`, Read-Write Sektion für globale oder statische Variablen
`.rodata`, Read-Only Sektion für konstante Variablen
`.bss`, Read-Write Sektion für uninitialisierte Variablen

Eine weitere Assembler Direktive ist `.align x`. Sie kann die Speicherausrichtung setzen, wobei x ein Vielfaches von 2 sein muss. Befehle und Daten, die auf die Direktive folgen, werden an Speicheradressen die ein Vielfaches von x sind abgelegt.

Die Direktiven `.byte`, `.half`, `.word` und `.dword` definieren einen Speicherbereich der entsprechenden Größe. Darauf können Daten durch Kommata separiert folgen. Um auf die Speicherstelle zugreifen zu können, kann mit einem Namen gefolgt von einem Doppelpunkt vor der Direktive ein Symbol definiert werden: `magic: .word 0x42`. Der Linker (GNU-LD)[37] ersetzt das Symbol dann durch die jeweilige Speicheradresse. Die Direktive `.globl` macht ein Symbol global sichtbar und kann somit von Programmen in anderen Dateien aufgerufen werden. Numerische Symbole können für lokale Referenzen verwendet werden. Beim Zugriff wird ein *f* für forward oder ein *b* für backward angehängt. Demnach sucht der Compiler vorwärts oder rückwärts nach der nächsten Übereinstimmung. Weitere Direktiven sind im Beispiel 4.9 aufgeführt.

Funktionen

Um die Einheitlichkeit von Funktionen zu gewährleisten, muss eine Calling Convention eingehalten werden. Nach dem Application Binary Interface müssen bestimmte Register vom Caller oder Callee vor Veränderung gesichert werden (siehe Tabelle 4.1). In der aufrufenden Funktion (Caller) müssen die temporären Register *t0-t6*, die Argumente *a0-a7* und die Rücksprungadresse *ra* gesichert werden. In der aufgerufenen Funktion (Callee) müssen die Register *s0-s11* und der Stackpointer *sp* gesichert werden. Funktionsparameter werden mit den Registern *a0-a7* übergeben. Weitere Argumente können auf dem Stack gespeichert werden. Eventuelle Rückgabewerte einer Funktion werden in *a0, a1* gesichert. Um aus einer Funktion zurückzukehren wird der *jalr* mit dem *ra* Register ausgeführt. Das Sichern von Registern passiert üblicherweise auf dem Stack. Dazu wird ein Stackframe erstellt, in dem die Anzahl der zu speichernden Bytes vom Stackpointer subtrahiert wird. Beispiel 4.10 zeigt diese Vorgehensweise.

[36] GNU-AS Manual: https://sourceware.org/binutils/docs/as/
[37] https://sourceware.org/binutils/docs/ld, September 2019

Beispiel 4.9: Beispielhafte Verwendung von Präprozessor- und Assemblerdirektiven

```
#define LOOPS 8                 // Ersetzt jedes Vorkommen von LOOPS durch 8
.section .data                  // Leitet die Datensektion ein.
  str_debug: .string "DEBUG-Nachricht"  // Legt den String unter dem Symbol ab.
  num_array: .byte 42, 23, 13, 37       // Erstellt ein Array.
.section .rodata                // Leitet die Readonly-Sektion ein.
  secret: .word 274354          // Legt den 32-Bit Wert ab.
.section .text                  // Leitet die Text-Sektion ein.
.globl main                     // Definiert das Symbol main global, damit
main:                           // main i.d.R. aus start.S aufgerufen werden kann.
  li a0, LOOPS                  // Load Immediate (a0 = 8)
  1:                            // Numerisches Symbol
  addi a0, a0, -1               // (a0 -= 1)
  bne zero, a0, 1b              // if(0 =! a0) PC=PC-4 (Offset zum ersten Vorkommen
                                //                      des Symbols "1" rückwärts)
  lui  a0, 0xaabbccdd >> 20     // liefert die oberen 20-Bits  (a0 = 0xaabbc000)
  addi a0, 0xaabbccdd & 0xfff // liefert die unteren 12-Bits (a0 = 0xaabbccdd)
  j main                        // jal zero, main
```

Beispiel 4.10: Beispiel von Funktionen. Die Funktion caller ruft callee auf, wobei nach Konvention bestimmte Register auf dem Stack gesichert werden müssen.

```
caller:
addi sp, sp, -4 // Stackframe
sw   ra, 0(sp)  // Sichern der Rücksprungadresse

li   a0, 3       // Übergabe der Argumente
li   a1, 4
jal  ra, callee  // call

lw   ra, 0(sp)  // Wiederherstellen der Rücksprungadresse
addi sp, sp, 4  // Zurücksetzen des Stackframes
jalr zero, ra   // return

callee:
addi sp, sp, -4 // Stackframe
sw   s0, 0(sp)  // Sichern des s0 Registers
// Berechnungen mit s0, a0, a1
lw   s0, 0(sp) // Wiederherstellen von s0
addi sp, sp, 4 // Zurücksetzen des Stackframes
jalr zero, ra  // return
```

4.2 Peripherie

Dieser Abschnitt zeigt den grundlegenden Umgang mit Peripheriefunktionen des FE310 auf dem Entwicklungsboard Hifive1. Die für diesen Teil relevanten Informationen wurden den Datenblättern entnommen (*SiFive FE310-G002 Manual* 2019; *SiFive HiFive1 Rev B Getting Started Guide* 2019). Einige Konzepte, wie z. B. ein memory-mapped GPIO, sind nicht selten auch auf anderen Plattformen in ähnlicher Art und Weise zu finden. Neben dem GPIO wird die Kommunikation mittels I2C gezeigt.

General-Purpose Input/Output (GPIO)

Der FE310 verfügt über 32 GPIO Pins. Diese sind memory-mapped und können mit den Load/Store Befehlen erreicht werden. Hierbei ist *GPIOBASE* die Basisspeicheradresse. Offsets im Abstand von 4 Byte bieten für den 32-Bit breiten GPIO verschiedene Funktionalitäten, zum Beispiel ob ein Pin als Eingang oder Ausgang konfiguriert sein soll. Weiterhin können Pull-Ups, Drive-Strength und Interrupts eingestellt werden. Beispiel 4.11 zeigt einige Funktionalitäten exemplarisch. Jedes Bit in einem 32-Bit breiten Offset korrespondiert hierbei mit einem Hardware Pin. Das Hifive Entwicklungsboard hat 20 Pins davon herausgeführt. Tabelle 4.12 bildet dieses Mapping ab. Um die Programmierung zu vereinfachen, wurde ein Headerfile mit Definitionen erstellt. Ein Ausschnitt ist in Beispiel 4.13 zu sehen.

Beispiel 4.11: Einfacher GPIO Zugriff

```
#include <gpio.h>

li   t0, GPIO_BASE          //Basis Speicheradresse
li   t1, (1 << GPIO_PIN5)   //Bitmaske
sw   t1, GPIO_OUT_EN(t0)    //Output Enable Offset
sw   t1, GPIO_OUT_VAL(t0)   //Output Value Offset
```

Ist ein Interrupt aktiviert, wird bei Anliegen eines passenden Zustands oder einer passenden Flanke das Bit des Pins im zugehörigen Interrupt Pending Register gesetzt. Auch können sogenannte IO-Functions ausgewählt und aktiviert werden. Damit kann die Kontrolle über einen Pin an eine Hardware Funktion, wie z. B. I2C oder UART, übergeben werden.

Inter-Integrated Circuit (I2C)

Mit externen Komponenten kann durch verschiedenste Protokolle, zum Beispiel I2C, kommuniziert werden. Eine Hardware Implementierung ist im FE310 vorhanden und kann durch Setzen der entsprechenden Bits im GPIO-Function Register aktiviert werden. Konfigurationsregister und Befehlsregister sind memory-mapped und müssen in vier Byte Abständen ab einer Basisadresse adressiert werden. Eine Library, die im Rahmen dieser Arbeit implementiert wurde, erleichtert den Zugriff auf die I2C Register. Sie bietet grundlegende Funktionen zur Initialisierung, sowie zum Senden und Empfangen von Daten auf

Tabelle 4.12: Pinout des Hifive1revB Entwicklungsboards

Pin	GPIO	Interrupt ID	IO-Function 0	IO-Function 1	Bemerkung
0	16	24	UART 0 RX		
1	17	25	UART 0 TX		
2	18	26	UART 1 TX		
3	19	27		PWM 1-1	LED
4	20	28		PWM 1-0	
5	21	29		PWM 1-2	LED
6	22	30		PWM 1-3	LED
7	23	31	UART 1 RX		
8	00	?		PWM 0-0	
9	01	09		PWM 0-1	
10	02	10	SPI 1 CS0	PWM 0-2	
11	03	11	SPI 1 MOSI	PWM 0-3	
12	04	12	SPI 1 MISO		
13	05	13	SPI 1 SCK		
14	-				Nicht Verbunden
15	09	17	SPI 1 CS2		
16	10	18	SPI 1 CS3	PWM 2-0	
17	11	19		PWM 2-1	
18	12	20	I2C SDA	PWM 2-2	
19	13	21	I2C SCL	PWN 2-3	

Beispiel 4.13: Ausschnitt der gpio.h Header-Datei

```
#define GPIO_BASE       0x10012000
#define GPIO_IN_VAL     0x00    // Pin Value
#define GPIO_IN_EN      0x04    // Pin Input Enable
#define GPIO_OUT_EN     0x08    // Pin Output Enable
#define GPIO_OUT_VAL    0x0C    // Output Value
#define GPIO_PUE        0x10    // Internal Pull-Up Enable
#define GPIO_DS         0x14    // Pin drive strength
#define GPIO_RISE_IE    0x18    // Rise Interrupt Enable
#define GPIO_RISE_IP    0x1C    // Rise Interrupt Pending
[...]
#define GPIO_PIN0  16
#define GPIO_PIN1  17
[...]
```

dem I2C-Bus. Beispiel 4.14 zeigt wie der Wert eines Registers von einem I2C Slave empfangen werden kann. Dazu wird erst die Slaveadresse mit einer 0 im LSB (Schreibzugriff) auf den Bus geschrieben, um danach das Register auszuwählen. Zum Auslesen wird die Slaveadresse mit einer 1 im LSB (Lesezugriff) auf den Bus geschrieben und die Antwort gelesen.

Beispiel 4.14: Beispiel einer I2C Kommunikation

```
#include "i2c.S"              // Einbinden der I2C Library
#define SLAVE_ADDR 0b1010101 // 7-Bit Addresse
#define SLAVE_REGISTER 0x3

call i2c_init                 // Initialisiert I2C Interface
li   a0, (SLAVE_ADDR << 1)    // LSB=0:Schreibzugriff
call i2c_start
li   a0,  SLAVE_REGISTER
call i2c_write
li   a0, (SLAVE_ADDR << 1) | I2C_READ // LSB=1: Lesezugriff
call i2c_rep_start
call i2c_readNaK              // Liest vom Bus und sendet NAck
// Rückgabewert befindet sich in  a0
```

4.3 Interrupts und Exceptions

Grundsätzlich werden die Begriffe *Trap, Exception und Interrupt* unterschieden. Dabei steht eine *Exception* für eine synchrone Unterbrechung des Programmablaufs durch ein außergewöhnliches Systemverhalten. Dies tritt beispielsweise bei einer unbekannten Instruktion oder beim Zugriff auf eine Adresse mit falschem Alignement auf. Dahingegen treten *Interrupts* asynchron bei normalem Verhalten durch Auslöser extern vom Core auf. Dies kann zum Beispiel durch einen Timer oder den GPIO auftreten. *Trap* ist der Überbegriff für eine Exception oder einen Interrupt, bei denen der Programmablauf unterbrochen und von einem Traphandler übernommen wird. Traps werden im zweiten Teil der RISC-V Spezifikation eingeführt (Waterman und Asanovic, 2019b). Es werden eigene Register (CSR) und Befehle vorgestellt. Da CSR auch für Zähler und Timer Verwendung finden, werden einige Register und Befehle bereits im ersten Teil der Spezifikation vorgestellt (Waterman und Asanovic, 2019a, Kap. 9). Timer und Software Interrupts werden im FE310 durch den so genanten Core-Local Interruptor (CLINT) verarbeitet, der in dem Datenblatt (*SiFive FE310-G002 Manual* 2019, Kap. 9) beschrieben ist. Die Verarbeitung externer Interrupts erfolgt durch einen Platform-Level Interrupt Controller (PLIC). Ein Spezifikationsentwurf dazu ist im RISC-V Repository zu finden [38].

[38] https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc, September 2019

Kontroll- und Statusregister

Die Control Status Register (CSR) bieten einen sicheren Zugriff auf Konfigurationen und Informationen des Prozessors. Dafür werden eigene Befehle verwendet, die Atomarität gewährleisten. Das heißt, dass der Zugriff nicht unterbrochen oder von Seiteneffekten beeinflusst werden kann. Die Privilege Architecture unterscheidet drei verschiedene Berechtigungsstufen. Machine-Mode (höchstes Level), Hypervisor-Mode und User-Mode.

Aufgeführt sind hier für diese Arbeit alle relevanten Befehle und Register zur Verwendung von Interrupts im Machine Mode.

`csrrw rd, csr, rs1`, Atomic Read-Write CSR
`csrrs rd, csr, rs1`, Atomic Read and Set Bits CSR
`csrrc rd, csr, rs1`, Atomic Read and Clear Bits CSR

Der `csrrw` Befehl schreibt den alten Wert des CSR nach *rd* und kopiert den Wert von *rs1* in das CSR. Wenn $rd = x0$ gilt, wird das CSR nicht nach *rd* geladen. Die `csrrs` und `csrrc` Befehle setzen bzw. löschen die in *rs1* gesetzten Bits in dem CSR. Für diese Befehle gilt bei $rs1 = x0$, dass nur gelesen und nichts geschrieben wird.

Machine Status Register (mstatus)

Das *mstatus* Register hält Kontrollbits zur globalen Aktivierung von Interrupts und das vor der Trap geltende Privilege Level.

Tabelle 4.15: Schematische Darstellung des *mstatus* Register. Bits gekennzeichnet mit '—' sind der Übersichtlichkeit halber ausgelassen worden.

Bit	31-13	12-11	10-8	7	6-4	3	2-0
Bezeichnung	—	MPP	—	MPIE	—	MIE	—

MIE [3], *Machine Interrupt Enable*, aktiviert Interrupts global

MPIE [7], *Machine Previous Interrupt enable*, wird bei einem Interrupt auf den Wert von mie gesetzt und ist hilfreich für verschachtelte Interrupts.

MPP [11 – 12], *Machine Previous Privilege*, hält das vor der Trap herrschende Privilege Level.

Machine Interrupt Enable Register (mie)

Das *mie* dient zur Aktivierung von Software, Timer oder externen Interrupts. Das dazu korrespondierende *Machine Interrupt Pending Register* (*mip*) hält an denselben Bits die Information, ob der Interrupt ausstehend (pending) ist.

Machine Trap-Vector Base-Address Register (mtvec)

Die Speicheradresse für die Trap Routine wird im *mtvec* Register gespeichert. Das Register kann nach Spezifikation eine Hardwired Read-Only Adresse beinhalten. Ist das Register

Tabelle 4.16: Schematische Darstellung des *mie* Register. Bits gekennzeichnet mit '—' sind der Übersichtlichkeit halber ausgelassen worden.

Bit	31-12	11	10-8	7	6-4	3	2-0
Bezeichnung	—	MEIE	—	MTIE	—	MSIE	—

MSIE [3], *Machine Software Interrupt Enable*, aktiviert Software Interrupts, die hauptsächlich zur Kommunikation zwischen mehreren HARDware Threads (HARTs) verwendet wird.

MTIE [7], *Machine Timer Interrupt Enable*, aktiviert den durch den CLINT augelösten Timer Interrupt

MEIE [11], *Machine External Interrupt Enable*, aktiviert die durch den PLIC gerouteten externen Interrupts.

auch schreibbar implementiert, beschreiben die Bits $[0-1]$ den *Mode* und die Bits $[2-31]$ die *Base*, die Adresse. Die *Base* muss dabei 4-Byte aligned sein, d. h. ein Vielfaches von vier.

Die Mode Bits sind wie folgt kodiert:

0 : Direct, alle Exceptions und Interrupts setzen PC = `mtvec.BASE`.

1 : Vectored, Exceptions setzen PC = `mtvec.BASE`, asynchrone Interrupts setzen den PC = `mtvec.BASE + 4 * cause`. Dabei ist cause der Interruptgrund, wie in *mcause* beschrieben.

Machine Exception Program Counter (mepc)

Bei dem Betreten einer Trap wird das *mepc* Register auf die Adresse der unterbrochenen Instruktion, also den Wert des PC, gesetzt. Im normalen Programmablauf wird das *mepc* nicht beschrieben. Dennoch kann es durch Software verändert werden, um z.B. nach einer Trap an eine andere Stelle zu springen.

Machine Cause Register (mcause)

Das *mcause* Register wird beim Auftreten einer Trap mit einem Exceptioncode beschrieben. Dieser gibt im MSB (das 31. Bit) an, ob es sich um einen Interrupt oder eine Exception handelt und kodiert im restlichen Register warum sie ausgelöst wurde. Da das MSB auch als Vorzeichen dient, kann mit einem einzelnen Branch (z.B. `blt zero, t0, exception`) überprüft werden, ob es sich um eine Exception oder einen Interrupt handelt. Die Kodierung des *mcause* wird in Tabelle 4.17 beschrieben.

Tabelle 4.17: Kodierung einiger Interruptauslöser im *mcause* Register. Interrupts haben negative und Exceptions positive Werte.

0x80000003	Machine Software Interrupt
0x80000007	Machine Timer Interrupt
0x8000000B	Machine External Interrupt
$0x0 \geq$ 0x80000010	Plattformabhängige Interrupts
$\geq$ 0x0	Exceptions

Machine-Mode Befehle

Mit dem mret Befehl kann von einer Trap zum normalen Programmablauf zurückgekehrt werden. Dabei wird der PC auf den gesicherten Wert im mepc gesetzt, das mstatus.mie Bit wird auf mstatus.mpie gesetzt. Der Privilege Mode wird auf den Wert von mstatus.mpp gesetzt.

Zusammenfassend ergibt sich Folgendes:

```
PC = mepc
mstatus.MIE = mstatus.MPIE
privMode = mstatus.MPP
```

Der Befehl Wait for Interrupt (wfi) kann Implementierungsabhängig den Prozessor anhalten oder eine nop Operation ausführen, bis ein Interrupt auftritt (mstatus.mip = 1). Ist das mstatus.mie-Bit gesetzt, wird zur im mtvec Register gespeicherten Adresse gesprungen, falls nicht wird die nächste Instruktion ausgeführt (pc = pc+4).

Timer Interrupts

Timer Interrupts werden im Core-Local Interruptor (CLINT) bereitgestellt. Dieser prüft ständig, ob ein im Register CLINT_MTIME_CMP gespeicherter Vergleichswert kleiner oder gleich dem Wert im CLINT_MTIME Register ist. Trifft dies zu und sind die mie.mtie und mstatus.mie-Bits gesetzt, wird ein Timer Interrupt ausgelöst. Beide Register sind 64-Bit breit und müssen daher bei einer RV32I Architektur mit zwei Speicherzugriffen gelesen als auch geschrieben werden. Das CLINT_MTIME wird durch eine Real Time Clock (RTC) mit einer Frequenz von 32768 kHz inkrementiert (im Fall des FE310). Soll beispielsweise der Timer Interrupt jede Sekunde auslösen, muss das CLINT_MTIME_CMP Register um den Wert der RTC Frequenz addiert werden. Die Laufzeit der Instruktionen kann vernachlässigt werden, da der letzte Vergleichswert als Referenz verwendet werden kann. Auch werden Instruktionen mit einer deutlich höheren Frequenz (zwischen 1.1MHz - 384MHz) als die der RTC ausgeführt.

Als Beispiel für einen Timer Interrupt soll eine LED jede Sekunde an bzw. ausgeschaltet werden. Das Programm wird hier in mehreren Abschnitten unterteilt und erklärt.

Als erstes werden Header-Dateien eingebunden und Definitionen vorgenommen.

```
#include "gpio.h"
#include "interrupts.h"
#define LED GPIO_PIN8
.section .text
.globl main
main:
```

Danach wird der GPIO für eine LED am PIN 8 als Ausgang definiert und initialisiert.

```
li  s0, GPIO_BASE
li  t1, (1 << LED)
```

```
sw  t1, GPIO_OUT_EN(s0)
sw  t1, GPIO_OUT_VAL(s0)
```

Dann erfolgt das Schreiben der Adresse des Traphandlers in das *mtvec* Register und das Setzen des *Timer Compare Register* initial auf 0.

```
la   t0, trap_handler  // Speicheradresse des Symbols "trap_handler"
csrw mtvec, t0         // Setzen des Traphandlers

li  t0, CLINT_MTIMECMP // Speicheradresse des Timer Compare Register
sw  zero, 4(t0)
sw  zero, 0(t0)
```

Im letzten Schritt der Initialisierung werden das *Machine Timer Interrupt Enable* (*mtie*) Bit im Register *mie* und das *Machine Interrupt Enable*-Bit im *mstatus* Register gesetzt.

```
li    t0, (1 << CSR_MIE_MTIE)
csrrs zero, mie, t0                       // setzt das MTIE-Bit
csrrsi t0, mstatus, (1 << CSR_MSTATUS_MIE) // setzt das MIE-Bit
```

Nun verharrt das Programm in einer Endlosschleife und führt bis zu einem Interrupt nichts aus.

```
loop:
wfi     // Wait for Interrupt
j loop
```

Löst ein Interrupt aus, so wird der PC auf die Adresse des *trap_handler* gesetzt. Erst wird der Interruptgrund gelesen und überprüft, ob es sich um eine Exception oder einen Interrupt handelt. Danach wird das MSB durch Shiften entfernt, um anschließend vergleichen zu können, ob es sich um einen Timer-Interrupt (MTI=7) handelt.

```
trap_handler:
csrr t0, mcause                 // csrrs t0, mcause, zero
bgez t0, trap_handler_exception // if (t0 >= 0) pc = pc + trap_handler_exception
slli t0, t0, 1
srli t0, t0, 1
li   t1, CSR_MCAUSE_MTI
bne  t0, t1, trap_handler_end  // if (t0 != 7) pc = pc + trap_handler_end
```

Falls es sich um einen Timer-Interrupt handelt, soll der Wert am GPIO invertiert werden.

```
lw  t1, GPIO_OUT_VAL(s0)
not t1, t1
sw  t1, GPIO_OUT_VAL(s0)
```

Um das Intervall bis zum nächsten Auslösen des Timer-Interrupts festzulegen, werden die 8 Byte des Compare Registers gelesen, der RTC-Frequenz Wert addiert und schließlich wieder gespeichert. Da es bei der Addition zu einem Overflow kommen kann, wird der Carry mit dem `sltu` Befehl generiert, wenn die Summe kleiner als der Summand ist.

```
li  t0, CLINT_MTIMECMP // t0 = 0x2004000
lw  t1, 0(t0)          // t1 = mem[0x2004000]
lw  t3, 4(t0)          // t3 = mem[0x2004004]
li  t2, RTC_FREQ       // t2 = 32768000
add t1, t1, t2         // t1 = t1 + t2
sltu t2, t1, t2        // t2 = (t1 < t2) Auf Carry prüfem
add t3, t3, t2         // t3 = t3 + t2   Carry addieren
sw  t1, 0(t0)          // mem[0x2004000] = t1
sw  t3, 4(t0)          // mem[0x2004004] = t3
```

Zuletzt wird der Interrupt beendet und ein Exceptioner-Handler definiert.

```
trap_handler_end:
mret

trap_handler_exception:
j trap_handler_exception
```

Externe Interrupts

Der Platform-Level Interrupt Controller verteilt Interrupts verschiedener Herkunft auf das externe Interrupt Bit (MEI). Damit können z. B. Interrupts vom GPIO, SPI, I2C, PWM und UART verwaltet werden. Dabei ermöglicht der PLIC auch das Setzen von Prioritäten und Thresholds. Zur Konfiguration und Aktivierung gibt es memory-mapped Register, die in der Header-Datei *interrupts.h* definiert sind. Zur Aktivierung eines Interrupts wird das Bit der vordefinierten Interrupt-ID im `PLIC_EN` gesetzt. Außerdem muss eine Priorität > 0 an die Adresse `PLIC_BASE+4*Interrupt_ID` geschrieben werden. Im Interrupt-Handler kann mit dem so genannten CLAIM Prozess der aktuell anliegende Interrupt mit der höchsten Priorität gelesen und beendet werden. Dazu muss von der Speicheradresse `PLIC_CLAIM` gelesen werden. Der Rückgabewert ist die Interrupt-ID. Schreibt man diese in das selbe Register, wird das Pending-Bit des Interrupts an der jeweiligen Stelle gelöscht und der Interrupt damit als beendet erklärt. Das Register gibt den Wert 0 zurück, wenn kein Interrupt anliegt. Die Interrupt-IDs und eine vollständige memory-map sind dem Datenblatt (*SiFive FE310-G002 Manual* 2019, Kap. 10) zu entnehmen.

In diesem Beispiel soll ein an einen GPIO Pin angeschlossener Drucktaster eine LED an- bzw. ausschalten. Da viele Konfigurationen ähnlich zum vorherigen Beispiel sind, soll hier nur auf die Besonderheiten des PLIC eingegangen werden.

In diesem Beispiel wird der GPIO als Eingang mit aktivierten Pull-Ups und Interrupts auf fallender Flanke konfiguriert. Register *s0* hält dabei wieder die *GPIO_BASE* Adresse.

```
li  t0, (1 << BTN)
sw  t0, GPIO_IN_EN(s0)   // Input Enable
sw  t0, GPIO_PUE(s0)     // Pull-Up Enable
sw  t0, GPIO_FALL_IE(s0) // Falling Edge Interrupt Enable
```

Die Interrupt-ID ist zuvor als *BTN_INTR* definiert worden. Eine Auflistung der GPIO Interrupt-IDs befindet sich in Tabelle 4.12. Zur Aktivierung im PLIC setzt man die höchste Priorität und das enable-Bit. Der Interrupt Priority Threshold wird deaktiviert. Alle anderen Interrupts werden deaktiviert um Seiteneffekte zu vermeiden.

```
li  t1, 7              // höhste Priorität
li  t0, PLIC_BASE      // Basisadresse des PLIC
sw  t1, BTN_INTR*4(t0) // Setzen der Interrupt Priority

li  t0, PLIC_IPTR      // Interrupt Priority Threshold
sw  zero, 0(t0)

li  t0, PLIC_EN_2      // Deaktivieren anderer Interrupts
sw  zero, 0(t0)
li  t0, PLIC_EN_1
li  t1, (1 << BTN_INTR)// Bitmaske für die GPIO Interrupt-ID
sw  t1, 0(t0)          // Aktivieren des GPIO Interrupts
```

Im Interrupt-Handler folgt nach der bekannten Überprüfung auf eine Exception die CLAIM-Prozedur. Dazu muss das CLAIM Register gelesen und überprüft werden, ob es sich um den richtigen Interrupt handelt. Ist dies der Fall, wird der Wert des GPIO Pins invertiert und das Pending Bit im GPIO zurückgesetzt. Das Beenden eines Interrupts erfolgt durch ein erneutes Schreiben der Interrupt-ID in das CLAIM Register.

```
trap_handler:
[...]
li   t0, PLIC_CLAIM       // Starten des CLAIM durch Laden
lw   t1, 0(t0)            // der aktuell anliegenden Interrupt-ID
li   t2, BTN_INTR
bne  t1, t2, otherInt     // Überprüfen, ob der Button-Interrupt ausgelöst hat

li   t2, (1 << BTN)       // Zurücksetzen des GPIO Pending-Bits
sw   t2, GPIO_FALL_IP(s0)

lw   t2, GPIO_OUT_VAL(s0) // Togglen der LED
xori t2, t2, (1 << LED)
sw   t2, GPIO_OUT_VAL(s0)

otherInt:
li   t0, PLIC_CLAIM       // Beenden des CLAIM durch Schreiben der Interrupt-ID
sw   t1, 0(t0)
mret
```

5

RISC-V in der Lehre

Die Veranstaltung *Technische Grundlagen der Informatik 1* (*TGI1*) beschäftigt sich mit fundamentalen Konzepten der Elektrotechnik und Computerarchitektur. Dazu gehört unter anderem die Programmierung in Assembler auf Atmels ATmega16. Dieser ist ein 8-Bit Mikrocontroller aus der AVR Reihe. Bei AVR handelt es sich wiederum um eine RISC Architektur, die besonders durch die Arduino Plattform große Beliebtheit gewonnen hat[39]. Alle den ATmega16 betreffenden Angaben sind dem Datenblatt entnommen (*Atmel ATmega16 Complete Datasheet* 2010). Im Rahmen des praktischen Übungsbetriebes sind von den Studierenden verschiedene Verhalten auf dem Atmega16 zu realisieren. Beispielhafte Lösungsansätze werden in dieser Arbeit nach RISC-V Assembler portiert. Anhand dieser Umsetzung wird die Eignung von RISC-V Assembler beispielhaft auf dem Hifive1RevB (FE310)(*SiFive FE310-G002 Manual* 2019) und dem EduCore-V (Aljnabi, 2019) für die Lehrtätigkeit untersucht.

Dieses Kapitel orientiert sich an dem inhaltlichen Aufbau von TGI1. Zu Beginn erfolgt die Betrachtung grundlegender GPIO Zugriffe anhand eines Lauflichts. Weiterhin werden Unterschiede in Funktionsaufrufen am Beispiel der rekursiven Fibonacci Funkion und Unterschiede in Speicherzugriffen anhand des Sortieralgorithmus Bubblesort gezeigt. Den Abschluss bildet die Untersuchung von Interrupts und weiterer Peripherie am Beispiel der Spaceinvaders Übung.

Alle Beispielprogramme sind aufgrund ihrer Länge nur ausschnittsweise dargestellt und sind im beiliegenden Ordner unter `hifive1revb/src` und `educorev/src` in vollem Umfang zu finden.

5.1 Grundlagen

Die erste praktische Übung fordert die Studierenden zur Realisierung eines Lauflichts, visualisiert durch LEDs, auf. In einem Ringshift wird jede der 12 LEDs nacheinander durchgeschaltet. Eine durch zwei Schalter kontrollierbare Wartezeit bestimmt dabei die Dauer einzelner Umschaltvorgänge. Eine Darstellung der Wartezeit soll auf einem 7-Segment Display erfolgen. Die Lösung dieser Aufgabe erfordert die Konzepte von arithmetischen Befehlen, Zugriffen auf den GPIO und bedingten Sprungbefehlen.

[39] `https://www.arduino.cc/en/Guide/Introduction`, September 2019

Input/Output

In einem ersten Schritt werden die GPIO-Pins als Ausgang für die LEDs und als Eingang für die Schalter konfiguriert. Die Beispiele 5.1 und 5.2 zeigen dieses Vorgehen.

Beispiel 5.1: GPIO Zugriff in AVR

```
[...]
ser r16         // r16 = 0b11111111
out DDRA, r16   // Ausgang für LED
out DDRB, r16   // Ausgang für 7-Seg
ldi r16, 0x0F   // r16 = 0b00001111
out DDRD, r16   // PD4, PD5 Schalter
                // restliche LED
[...]
```

Beispiel 5.2: GPIO Zugriff in RISC-V

```
[...]
li  s0, GPIO_BASE
li  t0, 0xff0e2f // Bitmaske LED, 7-Seg
sw  t0, GPIO_OUT_EN(s0)
[...]
li  t0, (1 << BTN_INC) | (1 << BTN_DEC)
sw  t0, GPIO_IN_EN(s0)
[...]
```

Port-Mapped, Memory-Mapped

Im Gegensatz zu dem memory-mapped Input Output (IO) der RISC-V Prozessoren arbeitet AVR mit einem port-mapped IO. Dieser hat einen vom Datenspeicher getrennten Bereich an Speicheradressen, auf die mit gesonderten Befehlen zugegriffen werden kann. Die Befehle `in` und `out` ersparen dabei das Laden der Adresse in ein Register. Es ist auch möglich, mit den normalen Load- und Store-Befehlen zuzugreifen. Beim memory-mapping wird derselbe Adressbereich für Datenspeicher und IO verwendet und mit den gleichen Befehlen zugegriffen. Nachteilig ist, dass Speicheradressen für den IO reserviert werden müssen und nicht für Datenspeicher zur Verfügung stehen. Da die 32-Bit und 64-Bit Architekturen aber genügend große Speicherbereiche haben, stellt dies kein Problem mehr dar. Daher verwenden viele moderne Systeme mittlerweile memory-mapped IO.

Eine Problemstellung dieser Aufgabe ist die Ansteuerung der 12 LEDs mit den acht Bit breiten Registern. So werden acht Bit des `PORTA` und die unteren vier Bit des `PORTD` verwendet. Das Hifive1 Board führt die GPIO Pins des FE310 anders heraus, als sie angesprochen werden. Beispielsweise ist das Bit 0 im GPIO Register der Pin 8 auf dem Entwicklungsboard. Daher können ebenso nicht direkt 12-Bit am Stück angesteuert werden, sondern nur einzelne Bereiche in dem 32-Bit Register.

Button Polling

Die Wartezeit zwischen dem Umschalten zweier LEDs soll variabel sein. Dazu sind Schalter mit dem GPIO zu verbinden. Das Abfragen der Schalterzustände erfolgt durch sogenanntes Polling. Dabei wird ständig der Eingabewert überprüft und im geschalteten Zustand die Wartezeit verändert. In diesem Beispiel liegt am Schalter im nicht geschalteten Zustand ein High-Pegel und im geschalteten ein Low-Pegel an. Mit dem `sbic` (skip if bit in IO-register is clear) Befehl bietet AVR eine praktische Bitoperation, die sehr platzsparend ist. Ist das zu überprüfende Bit in dem IO-Register gleich 0, wird die nächste Instruktion übersprungen (`pc+=2`), andernfalls wird die nächste Instruktion ausgeführt (`pc+=1`).

Beispiel 5.3 zeigt dies exemplarisch. Soll für einen Schalter jedoch mehr als nur eine Direktive ausgeführt werden, z. B. Test auf Überlauf, müssten einzelne Überprüfungen und Branches verwendet werden.

Der RV32I Befehlssatz beinhaltet keine Bitoperationen, allerdings gibt es die Bit Manipulation Erweiterung, die sich aber noch im Entwurfsstatus befindet. Beispiel 5.4 zeigt die Realisierung in RISC-V Assembler. Da der `andi` Befehl nur ein 12-Bit Immediate Feld hat, müssen größere Bitmasken vorher in ein Register geladen und mit dem `and` Befehl verwendet werden.

Beispiel 5.3: Button Polling in AVR

```
[...]
poll:
sbic PIND, PD4
subi delay, -5
sbic PIND, PD5
subi delay, 5




[...]
```

Beispiel 5.4: Button Polling in RISC-V

```
[...]
poll:
lw  t0, GPIO_IN_VAL(s0) // GPIO Wert laden
li  t1, (1 << BTN_INC)  // Bitmaske erstellen
and t2, t1, t0          // vergleichen
beq zero, t2, poll_increase
li  t1, (1 << BTN_DEC)
and t2, t1, t0
bne zero, t2, poll_decrease
j poll_end
poll_increase:
addi delay, delay, 5
[...]
```

Ringshift

Zustände der Airthmetic Logic Unit (ALU) werden in AVR in einem Status Register (SREG) gespeichert. Jedes Bit in diesem Register entspricht einem so genannten Flag. Zum Beispiel zeigt das Negative-Flag an, dass die letzte Operation der ALU ein negatives Ergebnis war. Weiterhin gibt es Flags für einen Carry, einen Overflow und einige Weitere. Diese werden von Branches verwendet und ermöglichen einige Tricks. Wie in Beispiel 5.5 zu sehen ist, kann der Inhalt aus einem Register in das Nächste mit dem Carry Bit geshiftet werden.

RISC-V kennt kein SREG in dieser Art, bietet aber dafür Befehle, die dies ersetzen können. So arbeiten die Branch Befehle direkt auf zwei zu vergleichenden Registern oder Befehle wie `sltu` (set if less then unsigned), der einen Carry (wie in Beispiel 5.17 verwendet) erzeugen kann.

Ein anderer Weg, um einen solchen Ringshift zu produzieren ist, zu prüfen, ob ein bestimmtes Bit erreicht wurde und entsprechend das Register auf das nachfolgende Bit zu setzen (Beispiel 5.6). Auch möglich, allerdings didaktisch an dieser Stelle verfrüht, ist ein Array, welches für jede LED die Pinnummer hält. Dabei muss nur über dieses Array iteriert werden und der aus dem Array gelesene Pin geschaltet werden.

Beispiel 5.5: LED shiften in AVR

```
[...]
mov  r16, LEDhigh
swap r16
lsl  r16     // generiert Carry
rol  LEDlow  // fügt Carry ins LSB
rol  LEDhigh
andi LEDhigh, 0x0F



[...]
```

Beispiel 5.6: LED shiften in RISC-V

```
[...]
andi t0, leds, (1 << 5)
bne  zero, t0, shift_bit_5
[...]
slli leds, leds, 1
j shift_end

shift_bit_5:
li   leds, (1 << 9)
j shift_end
[...]
```

Warteschleifen

Damit die LEDs für eine kontrollierbare Zeit leuchten, werden zwei Warteschleifen verschachtelt. Die Innere dauert eine feste Zeit und die Äußere ist durch einen Multiplikator steuerbar.

Die Dauer einer Instruktion wird mit der Anzahl an benötigten Clock-Cycles angegeben. Der ATmega16 führt eine Instruktion pro Clock-Cycle aus. Das wird durch eine 2-stage-pipeline mit parallelem *Instruction Fetch* und *Instruction Execute* erreicht. Das Datenblatt zeigt für jede Instruktion wie viele Clock-Cycles sie benötigt. Mit der Anzahl an Clock-Cycle und der Taktfrequenz (1-16MHz) kann die Dauer einer Warteschleife einfach errechnet werden.

Der FE310 verwendet eine 5-stage-pipeline, die aus den Phasen *Instruction Fetch*, *Instruction Decode*, *Execute*, *Data Memory Access* und *Register Writeback* besteht. Weitere Caches, wie z. B. zur Branchprediction, beschleunigen die Ausführungszeit, machen aber eine genaue Aussage über eben diese schwerer. So gibt es extra Performance-Counter die z. B. Cache-Hits zählen. Das Datenblatt des FE310 zeigt eine Spitzen-Ausführungsrate von einer Instruktion pro Clock-Cycle. Durch die erhöhte Komplexität lässt sich nur grob abschätzen, wie viel Zeit es benötigt um eine Warteschleife zu durchlaufen. Trotz alledem ist die Implementierung einer Warteschleife in beiden Assemblersprachen sehr ähnlich, wie es die Beispiele 5.7 und 5.8 zeigen.

Beispiel 5.7: Wartschleife in AVR

```
[...]
clr r17
delay_outer:
  inc  r17
  cp   r17, delay
  breq delay_outer_end
  ldi  r16, 250
  delay_inner:
    dec r16
    brne delay_inner
rjmp delay_outer
[...]
```

Beispiel 5.8: Warteschleife in RISC-V

```
[...]
mv t0, zero
delay_outer:
  addi t0, t0, 1
  beq  t0, delay, delay_outer_end
  li   t1, 200000

  delay_inner:
    addi t1, t1, -1
    bnez t1, delay_inner
j delay_outer
[...]
```

5.2 Funktionen

Assemblerprogramme lassen sich mit der Verwendung von Funktionen besser strukturieren, modularisieren und machen diese übersichtlicher. Eine Funktion kann mehrere Übergabeparameter (auch Argumente genannt) und einen bis zwei Rückgabewerte haben.

Calling Convention

RISC-V's Calling Convention wurde in Abschnitt 4.1 bereits erläutert und ist in Tabelle 5.9 noch einmal zusammengefasst. Wie auch bei RISC-V werden in AVR bei mehr Argumenten als zur Verfügung stehenden Argumentregister weitere Argumente auf dem Stack gespeichert. Ist der Rückgabewert in AVR größer als 8-Bit, können bis zu 64-Bit mit den Registern *r18-r25* zurückgegeben werden. Weitere Details sind im Reference Manual zu finden[40].

Tabelle 5.9: Vergleich der Calling Convention

	AVR	RISC-V
Argumente	r25 - r8	a0 - a7
Rückgabewerte	r24	a0, a1
Caller-Saved	r18 - r27, r30 - r31	t0 - t6, ra
Callee-Saved	r2 - r17, r28 - r29	s0 - s11

[40] https://www.microchip.com/webdoc/avrlibcreferencemanual/FAQ_1faq_reg_usage.html, September 2019

Rücksprungadresse

Bei einem Funktionsaufruf wird in AVR-Assembler die Rücksprungadresse automatisch auf dem Stack gesichert. Bei einem Return wird sie wieder in den PC geladen. RISC-V nutzt hierfür nach Konvention das Register x2: *Return Address* (*ra*). Mit dem `jal` (*jump and link*) Befehl wird dann bei einem Funktionsaufruf die Rücksprungadresse in diesem Register gesichert. Dies spart Speicherzugriffe, da sogenannte Leaf-Funktionen, die keinen weiteren Funktionsaufruf durchführen, nicht auf den Stack zugreifen müssen. Bei verschachtelten Aufrufen muss die Adresse nach wie vor auf dem Stack gesichert werden.

Beispiel Fibonacci

Die Fibonacci-Folge kann sehr leicht über eine Rekursion hergeleitet werden. Beispiel 5.10 zeigt dies beispielhaft in der C Programmiersprache. Wie klar zu sehen ist, ruft sich die Funktion zur Herleitung selbst auf.

Beispiel 5.10: Fibonacci in C

```
int fib(int n) {
    if (n == 0 || n == 1)
          return n;
    else
          return (fib(n-1) + fib(n-2));
}
```

Die Beispiele 5.11 und 5.12 zeigen die Implementierung der Fibonacci-Folge in Assembler. Bis auf das Initialisieren des Stackpointers fällt die Variante in AVR etwas kürzer aus. Das liegt hauptsächlich an den `push` und `pop` Befehlen. Diese verändern den Stackpointer und laden / speichern das jeweilige Register. Außerdem zwingen die Befehle den Programmierer streng nach dem Prinzip des Stacks zu arbeiten. Das heißt, es darf nur auf das oberste Element zugegriffen werden. Trotzdem ist es prinzipiell möglich, Lade- und Schreibzugriffe beliebiger Art mit dem Stackpointer durchzuführen.

Beispiel 5.11: Fibonacci in AVR

```
#inlcude "m16def.inc"
  main:
  ldi r16, high(RAMEND)
  out SPH, R16
  ldi r16, low(RAMEND)
  out SPL, R16

  loop:
  ldi r25, 10
  call fib
  rjmp loop

  fib:
  push r24




  clr  r24

  cpi r25, 2
  brlt fib_end

  push r25
  subi r25, 1
  call fib
  mov  r24, r25
  pop  r25
  subi r25, 2
  call fib
  add  r25, r24

  fib_end:



  pop r24
  ret
```

Beispiel 5.12: Fibonacci in RISC-V

```
.globl main
main:
li sp, 0x80000800



loop:
li a0, 10
call fib
j loop

fib:
addi sp, sp, -12
sw  ra, 0(sp)
sw  a0, 4(sp)
sw  a1, 8(sp)
li  a1, 0

li  t0, 1
ble a0, t0, fib_end

addi a0, a0, -1
call fib
mv a1, a0
lw a0, 4(sp)
addi a0, a0, -2
call fib
add a0, a0, a1


fib_end:
lw  a1, 8(sp)
lw  ra, 0(sp)
addi sp, sp, 12
ret
```

5.3 Speicherzugriffe

Dieser Abschnitt betrachtet Unterschiede in Zugriffen auf den Speicher. Dabei bietet AVR für unterschiedliche Speicher spezielle Befehle. Diese erleichtern den Zugriff, sorgen aber auch für mehr Komplexität. RISC-V verwendet die Load- und Store-Befehle für alle Zugriffe.

Direkte und Indirekte Speicherzugriffe

Der ATmega16 hat 16Kbyte Flash Speicher (ROM), 512Byte EEPROM und 1Kbyte SRAM. Die Speicher sind über verschiedene Daten-Busse angebunden. Der EEPROM ist wie auch der GPIO an den IO-Bus angegliedert und wird mit den `in` und `out` Befehlen angesprochen. Der SRAM sitzt am Datenbus und kann direkt oder indirekt mit sieben verschiedenen Befehlen adressiert werden. Der ROM kann nur lesend und nur indirekt mit dem Befehl `lpm` adressiert werden. Direkter Speicherzugriff auf den SRAM ist mit den Befehlen `sts` und `lds` möglich. Zur indirekten Adressierung kann eines der drei 16-Bit Register `X`, `Y`, `Z`, welche jeweiles Registerpärchen von `r26`–`r31` sind, als Pointer verwendet werden.

Im Gegensatz dazu kann in RISC-V durch das memory-mapping auf verschiedenste Speicher und IO-Komponenten mit denselben Befehlen zugegriffen werden. Es kann nur indirekt adressiert werden, dafür aber mit allen general-purpose Registern.

Insgesamt benötigt AVR für die verschiedenen Speicher (ROM, RAM, EEPROM) elf unterschiedliche Befehle für Speicherzugriffe. RISC-V kommt mit drei grundlegenden Befehlen aus[41].

Beispiel Bubblesort

Der Sortieralgorithmus Bubblesort iteriert mit zwei verschachtelten Schleifen über ein im Speicher abgelegtes Array. Dabei werden immer zwei benachbartete Werte verglichen. Falls das vordere Element größer ist, werden sie vertauscht. Beispiel 5.13 zeigt eine Implementierung in C (Knuth, 1997, Section 5.2.2).

Beispiel 5.13: Bubblesort in C

```
void bubblesort(int arr[], int len) {
  for(int n=len-1;n > 0; n--)
    for(int i=0; i < n; i++)
      if(arr[i] > arr[i+1]) {
        int tmp = arr[i+1];
        arr[i+1] = arr[i];
        arr[i] = tmp;
      }
}
```

[41] Genau genommen sind es acht Instruktionen, betrachtet man auch diejenigen für 32-, 16- und 8-Bit Zugriffe.

In beiden Assemblersprachen lässt sich dieser Algorithmus sehr ähnlich umsetzen. Dies zeigt sich an den Beispielen 5.14 und 5.15. Ein Unterschied besteht im Speicherzugriff, wobei in AVR mit dem Postinkrement des X-Pointers gearbeitet werden kann. Dafür kann mit RISC-V jedes Register als Pointer verwendet werden.

Beispiel 5.14: Bubblesort in AVR

```
[...] //r25: length, X: array pointer
bubblesort:
dec r25
OLOOP:
  clr r24
  ILOOP:
    inc r24
    ld  r23, X+
    ld  r22, X

    cp   r23, r22
    brcs INEXT
    subi XL, 1
    st   X+, r22
    st   X,  r23
  INEXT:
    cp   r24, r25
    brne ILOOP
ONEXT:
  dec r25
  cpi r25, 0
  brne OLOOP
ret
```

Beispiel 5.15: Bubblesort in RISC-V

```
[...] //a0: array pointer, a1: length
bubblesort:
addi a1, a1, -1 //length
outer:
  mv t0, zero
  inner:
    addi t0, t0, 1
    lw   t1, 0(a0)
    lw   t2, 4(a0)

    blt  t1, t2, 1f
    sw   t2, 0(a0)
    sw   t1, 4(a0)

    1:
    addi a0, a0, 4
    blt  t0, a1, inner



addi a1, a1, -1
bgt  a1, zero, outer
ret
```

5.4 Traps und Peripherie

Nachfolgende Beispiele sind der letzten und größten praktischen Übung entnommen. Es soll das Arcade-Spiel Spaceinvaders mit einem 2-Zeilen LCD realisiert werden. Dazu werden Timer zur Visualisierung und dem Zeitverlauf und ein externer Interrupt für einen Schalter verwendet. Weiterhin wird ein Potentiometer über einen Analog Digital Converter (ADC) ausgelesen und das LCD mit einem seriellen Protokoll angesteuert.

Interrupts

Beide Architekturen unterstützen interne und externe Interrupts. Der ATmega16 hat einen 16-Bit und zwei 8-Bit Timer, drei externe GPIO Interrupts und Weitere für andere Schnittstellen wie UART, ADC und SPI. Der FE310 hat 52 externe Interrupts, davon sind

32 GPIO Interrupts, und einen 64-Bit Timer. Es wird deutlich, dass sich auf diesem Gebiet beide Architekturen stark unterscheiden. Dennoch werden im Folgenden einige Gemeinsamkeiten und Unterschiede aufgezeigt:

Der zweite Teil der RISC-V Spezifikation (Waterman und Asanovic, 2019b) beschreibt die Basis für Interrupts. Da RISC-V Prozessoren in jeder Größe erstellt werden können sollen, wird Kompatibilität für größere Software-Stacks (Betriebssysteme, Hypervisor) bereitgestellt. Das führt unweigerlich zu einer höheren Komplexität, als sie ein einfacher Mikrocontroller aufweist. Außerdem lässt die Spezifikation Platz für plattformspezifische Implementierungen, wie z.B. den PLIC des FE310. Dies erhöht weiter die Komplexität, da auch hier Freiraum für größere und komplexere Systeme gelassen wird. Wie später noch in Abschnitt 5.5 ausführlicher beschrieben wird, kommt hier eine für die Lehre angepasste Implementierung mit deutlich geringerer Komplexität zur Geltung.

Traphandler

Traps bzw. Interrupts werden in AVR und RISC-V unterschiedlich abgehandelt. AVR hat an einer festen Adresse eine sogenannte *Interupt-Vector-Table*. Diese bietet für jeden Interrupt einen Eintrag mit ausreichend Platz für eine Instruktion, üblicherweise ein Jump zu einer Interrupt-Service-Routine (ISR). Löst ein Interrupt aus, wird im SREG das I-Flag gelöscht und damit das Auftreten eines weiteren Interrupts verhindert. In der ISR müssen verwendete Register und das SREG auf dem Stack gesichert werden, da es sonst zu Seiteneffekten kommen könnte. Beendet wird eine ISR mit dem `reti` (Return from Interrupt) Befehl, der das I-Flag wieder setzt und den PC auf die im Stack gesicherte Rücksprungadresse setzt.

RISC-V sieht ein eigenes Register vor, das die Adresse des Trap Handlers hält (`mtvec`). Es ist möglich Interrupts vektorisiert, ähnlich wie in AVR, an unterschiedliche Speicheradressen springen zu lassen, meist ist es aber einfacher den Interrupt Auslöser über das `mcause` Register direkt auszulesen. Gesichert werden müssen im Trap-Handler nur die Register, die verwendet werden, und das Register `ra`, falls eine Funktion aufgerufen wird. Der letzte Wert des PCs wird in einem eigenen Register (`mepc`) gesichert und Interrupts werden vorläufig durch das Löschen des `mstatus.mie`-Bits deaktiviert. Der Befehl `mret` beendet einen Interrupt.

Timer

Das Intervall in dem ein Timer in AVR einen Interrupt auslöst, kann über Prescaler (abhängig von der Systemclock) und Vergleichswerte kontrolliert werden. Ein Timer Interrupt wird entweder bei einem Overflow oder beim Erreichen eines Vergleichswerts ausgelöst. Je nach eingestelltem Modus zählt der Timer nach einem Interrupt weiter oder wird auf 0 zurückgesetzt. Die RISC-V Spezifikation sieht einen 64-Bit Timer vor, der von einer festen (oft Real-Time-) Clock gesteuert wird. Im Sinne der RISC-Philosophie sieht die Spezifikation keine weiteren Timer vor und befürwortet die Implementierung von Software-Timern, die den Hardware-Timer multiplexen. Weitere Timer könnten auf Plattform Level implementiert werden. Ein vermeintlicher Trend in eingebetteten Systemen geht weg von Programmen, die direkt mit starren Timern arbeiten, hin zu Real-Time Operating Systems (RTOS). Wie in Abschnitt 4.3 schon beschrieben, wird das Intervall bis zum nächsten Timer-Interrupt durch den Vergleichswert und die RTC-Frequenz bestimmt. So muss

im Trap-Handler stets der nächste Vergleichswert gesetzt werden.

Beispiel 5.17 zeigt, wie im RISC-V Trap Handler ein zweiter Timer durch Software abgebildet werden kann. In dem eigentlichen Timer, dessen Vergleichswert immer um 10ms inkrementiert wird, wird ein weiterer Vergleichswert (gespeichert in einer globalen Variable) überprüft und um eine Sekunde inkrementiert (RTC-Frequenz). Im Vergleich dazu wird in Beispiel 5.16 die Konfiguration zweier seperater Timer in AVR-Assembler gezeigt. Beide Programmtexte sind Ausschnitte des Programms Spaceinvaders und machen deutlich, wie unterschiedlich die Implementierung von Timern ist.

Beispiel 5.16: Timerkonfiguation in AVR

```
[...]
ldi  r16, (1 << WGM01)
       | (1 << CS00)
       | (1 << CS02)
out  TCCR0, r16
ldi  r16, 100
out  OCR0, r16

ldi  r16, (1 << WGM12)
       | (1 << CS10)
       | (1 << CS12)
out  TCCR1B, r16

ldi  r16, high(977)
out  OCR1AH, r16
ldi  r16, low(977)
out  OCR1AL, r16

ldi  r16, (1 << OCIE0)
       | (1 << OCIE1A)
out  TIMSK, r16


[...]
```

Beispiel 5.17: Ausschnitt Traphandler für Timer in RISC-V

```
[...]
trap_handler_timer:
  call timer0_isr

  la  s1, CLINT_MTIME // Lädt RTC Zeit
  lw  s2, 0(s1)
  lw  s3, 4(s1)
  la  s1, TIMER1_CMP // Globale Variable
  lw  s4, 0(s1)
  blt s2, s4, 1f  // Timer1 überprüfen
  li t0, RTC_FREQ // TIMER1_CMP += RTC_FREQ
  add  s4, s2, t0 // entspr. 1 Sekunde
  sw  s4, 0(s1)
  call timer1_isr

  1:
  li  s4, RTC_FREQ / 100 // entspr. 10ms
  add s4, s4, s2
  sltu s2, s4, s2 // Carry Bit generieren
  add s3, s3, s2
  la  s1, CLINT_MTIMECMP
  sw  s3, 4(s1)
  sw  s4, 0(s1)
[...]
```

Externe Interrupts

Externe Interrupts verlaufen in der AVR-Architektur sehr ähnlich zu den Timer Interrupts. Eine ISR wird in der Interupt-Vector-Table aufgerufen und in einem Konfigurationsregister wird festgelegt, auf welche Flanke der Interrupt auslöst. Der FE310 führt alle externen Interrupts durch den PLIC. Dieser bietet zwar mehr Funktionalitäten, aber die Konfiguration und Verwendung wird ebenso komplexer. Wie ein Button Interrupt konfiguriert wird, wurde bereits in Abschnitt 4.3 gezeigt und wird hier ausgespart. Die Bei-

spiele 5.18 und 5.19 zeigen den Unterschied zwischen Interrupt-Vector-Table und einem Trap Handler.

Beispiel 5.18: Externer Interrupt in AVR

```
[...] Interrupt Vector Tabelle
.org INT2addr
     jmp ISR_INT2
[...]
ldi  r16, (1 << INT2)
out  GICR, r16




[...]




ISR_INT2:
  push r16
  in   r16, SREG
  push r16
  [...]
  pop  r16
  out  SREG, r16
  pop  r16
  reti
```

Beispiel 5.19: Externer Interrupt in RISC-V

```
[...] Initialisierung
trap_handler:
[...] mcause auslesen usw.
  trap_handler_extern:
  li   s2, PLIC_CLAIM
  lw   s1, 0(s2)
  li   s4, BTN_FIRE_INTR
  bne  s1, s4,
      trap_handler_extern_end_claim
  call button_isr
  li   s2, GPIO_BASE
  li   s4, (1 << BTN_FIRE_GPIO)
  sw   s4, GPIO_FALL_IP(s2)
[...]
mret

button_isr:
addi  sp, sp, -4
sw    ra, 0(sp)
[...]
lw    ra, 0(sp)
addi  sp, sp, 4
ret
```

Analog Digital Converter (ADC)

Ein ADC wandelt analoge, kontinuierliche Spannungen proportional in digitale diskrete Werte um. Meist sind ADC seriell realisiert, wobei durch das sequentielle Vorgehen je nach Auflösung eine gewisse Umsetzungszeit benötigt wird (Schiffmann, 2006). Das Ende einer Umsetzung kann durch das Pollen eines Statusflags oder durch einen Interrupt bemerkt werden. Der ATmega16 hat einen 10 Bit ADC auf dem IO-Bus integriert. Im Programm Spaceinvaders wird ein Potentiometer, ausgelesen durch den ADC, zur Steuerung des Raumschiffes verwendet. Der FE310 hat keinen ADC verbaut. Um ein ähnliches Ergebnis zu erzielen, wurde ein externer ADC über den I2C-Bus verbunden. Der *ADS1115* (*ADS1115 I2C-Compatible, 16-Bit ADC Datasheet* 2018) eignet sich hierfür sehr gut. Allerdings stand für diesen noch keine Library zur Verfügung, weshalb diese implementiert werden musste. Zuerst wird eine I2C Kommunikation, wie in Abschnitt 4.2 beschrieben, aufgebaut. Im Anschluss müssen je nach Betriebsart verschiedene Bits im Konfigurationsregister des *ADS1115* gesetzt werden. Schließlich kann der gemessene Analogwert aus dem Conversion Register als 16-Bit Signed Wert gelesen werden.

Liquid Crystal Display (LCD)

Die Visualisierung der Spaceinvaders Übung erfolgt mit einem LCD. Das verwendete LCD hat den *Hitachi HD44780 controller* verbaut. Zur Kommunikation wird ein serielles Protokoll verwendet. Die AVR Implementierung konnte angepasst und auf RISC-V Assembler portiert werden. Weitere Details zum Protokoll können dem Datenblatt (*HD44780, Dot Matrix Liquid Crystal Display Controller/Driver Datasheet* 1998) entnommen werden. Dabei ist vor allem das Timing zu beachten. Viele AVR-Implementierungen verlassen sich hierbei auf die CPU-Frequenz und riskieren falsches oder ungenaues Timing bei unterschiedlichen Taktungen. Vorteilhaft am FE310 ist die Realtime-Clock, die unabhängig von der CPU-Frequenz ein zuverlässiger Zeitgeber für Verzögerungen ist. Ein einfaches blockierendes Delay ohne Timer kann wie in Beispiel 5.21 aussehen. Das AVR Beispiel 5.20 durchläuft $257 * (255 * 3 + 2)$ Takte. Bei einer Taktfrequenz von 1MHz entspricht das ungefähr 19.71ms, bei 16MHz sind es 1.23ms. Grundsätzlich wäre es natürlich auch möglich, eine externe RTC am ATmega16 zu verwenden, die allerdings zusätzlich verbaut werden müsste.

Beispiel 5.20: Blockierendes Delay in AVR Assembler, abhängig von der Clock

```
delay:
clr r16
clr r17

delay_:
  dec r16
  bne delay_
  dec r17
  brne delay_

ret
```

Beispiel 5.21: Blockierendes Delay in RISC-V, abhängig von einer RTC

```
delay_ms:
li  t0, RTC_FREQ / 1000
mul t0, t0, a0
li  t1, CLINT_MTIME
lw  t2, 0(t1)
add t0, t0, t2
1:
  li  t1, CLINT_MTIME
  lw  t2, 0(t1)
  ble t2, t0, 1b
ret
```

5.5 EduCore-V

Einer der größten Vorteile der RISC-V Architektur ist die Offenheit des Standards. Das heißt, jede*r die*der möchte, kann ihre*seine eigene Version eines RISC-V Cores implementieren. Dabei muss selbstverständlich die Spezifikation eingehalten werden, in einigen Bereichen ist die Umsetzung aber auch der*dem Implementierenden freigestellt. So hat auch Aljnabi vom Institut für Technische Informatik seinen eigenen RISC-V Core für den Lehrbetrieb entwickelt, den EduCore-V (Aljnabi, 2019). Dabei hat er eine leicht verständliche und nicht zu komplexe, aber dennoch moderne Architektur entwickelt, die sich für den Lehrbetrieb verwenden lässt. Der EduCore-V wird auf einem *Basys 3* Entwicklungsboard mit einem *Artix 7* FPGA simuliert. Neben LEDs, Schaltern, 7-Segment

Display und GPIO verfügt das *Basys 3* Board über eine Video Graphics Array (VGA)-Schnittstelle. Unterstützt wird der RV32I Basisinstruktionssatz, die CSR Erweiterung und einige Features aus der Privilege Spezifikation. Weitere Hinweise zur Verwendung und Programmierung des Cores sind im Anhang A abgelegt.

Die in diesem Kapitel beschriebenen Beispiele wurden auch für den EduCore-V angepasst. Es zeigt sich, dass durch die simple Architektur auch einige Assemblerprogramme weniger komplex sind. Das soll im Folgenden an Auszügen einiger Beispiele näher erläuter werden. Für die Portierung der Programme musste nur wenig abgeändert werden. Dabei handelte es sich hauptsächlich um Definitionen und die Interrupts. Programme die keine IO-Zugriffe haben, wie z. B. Bubble-Sort und Fibonacci konnten unverändert übernommen werden.

Lauflicht

Beispielsweise kann auf die verbauten LEDs direkt mit dem `LED` Offset vom `GPIO_BASE` zugegriffen werden. Für das in Abschnitt 5.1 beschriebene Lauflicht bedeutet dies, dass nur das 12. Bit geprüft werden muss und ansonsten ein logischer Linksshift durchgeführt wird. Die Warteschleife konnte unter Anpassung der Clock-Frequenz übernommen werden und auch für die Schalter musste nur vom `BUTTON` Offset gelesen werden. Beispiel 5.22 zeigt diese Implementierung.

Beispiel 5.22: Lauflicht auf dem EduCore-V

```
main:
    li   delay, 5
    li   s0, GPIO_BASE
    li   leds, 1
loop:
    mv   a0, delay
    call segment_print_hex
    sh   leds, LED(s0)
    slli leds, leds, 1
    li   a1, (1<<12)
    bne  leds, a1, 1f
    li   leds, 1
1:
delay:
  [...]
poll:
  [...]
```

Das 7-Segment Display, welches die Wartezeit ausgibt, muss segmentweise angesprochen werden. Der Vorteil gegenüber einem Binary-Coded Decimal (BCD) ist, dass beliebige Symbole angezeigt werden können. Beispiel 5.23 zeigt eine kleine Hilfsfunktion, die geschrieben wurde, um Hexadezimalwerte darzustellen. Die Hex-Symbole (0 bis F) liegen als Segmente kodiert in einem Array. Die Funktion iteriert über die vier 7-Segment Anzeigen. Zunächst wird das unterste Nibble der anzuzeigenden Zahl maskiert und der

Wert aus dem Array an diesem Index geladen. Dieser Wert wird an den Offset der 7-Segment Anzeige geschrieben und die Zahl um vier nach rechts (ein Byte) geshiftet.

Beispiel 5.23: Ausschnitt aus segment.S

```
.section .data
segment_nums: .byte 0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6d, 0x7d, 0x07,\
                    0x7f, 0x6f, 0x77, 0x7c, 0x39, 0x5e, 0x79, 0x71

.section .text
segment_print_hex:
li  t0, GPIO_BASE + SEG_0
li  t2, 4            // Loop index
1:                   // Loops 4 times
andi a1, a0, 0xf     // Mask lowest byte
la  t1, segment_nums // Load array base
add t1, t1, a1       // Add n to array index
lb  a1, 0(t1)        // Converts number to 7seg letter
sb  a1, 0(t0)        // Store on SEG_n
addi t0, t0, 1       // n++
srli a0, a0, 4       // Shift for next Byte
addi t2, t2, -1      // loop index
bne zero, t2, 1b
ret
```

Interrupts

Timer Interrupts sind, wie in der Privilege Spezifikation vorgegeben, umgesetzt und lassen sich genauso verwenden wie mit dem FE310. Mit einem externen Interrupt sind bisher nur die verbauten Knopfschalter versehen. Sie lassen sich über den Offset `BUTTON_CONF` konfigurieren und haben den Wert 16 im `mcause`. Im `BUTTON_LATCH` lässt sich auslesen, welche Button-Interrupts ausstehend sind. Um die jeweiligen Bits zu löschen, muss an den selben Offset geschrieben werden. Vergleicht man dies mit der Variante auf dem FE310 (Beispiel 5.19) zuzüglich der Initialisierung (Abschnitt 4.3), wird sehr deutlich, wie viel simpler diese Lösung (Beispiel 5.24) ist.

Schalter die über einen GPIO angeschlossen werden, benötigen entweder weitere Bauteile für eine Hardware Entprellung oder eine Software Entprellung. Da auf dem *Basys 3*-Board Schalter bereits verbaut sind, wurde eine Entprellung direkt in Hardware umgesetzt.

Peripherie

Das Board verfügt über einen integrierten ADC, der mit dem RISC-V Core über den GPIO-Offset `ANALOG_0` gelesen wird. Es gibt 24 GPIO-Pins, die sich auf drei Ports (A, B, C) verteilen. Kontrolliert wird ein Port über die Offsets `DDR_{A,B,C}` (Data Direction Register), `PIN_{A,B,C}` (Port Input Value) und `PORT_{A,B,C}` (Port Output Value). Da

Beispiel 5.24: Externer Interrupt auf dem EduCore-V

```
interrupt_init:
li  t0, 0b11111
sb  t0, BUTTON_LATCH(s0)          // Löschen des Button Latch
li  t0, (1 << BTN_FIRE)
sb  t0, BUTTON_N_CONF(s0)
[...]
trap_handler_external:
  lb   s1, BUTTON_LATCH(s0)       // Auslesen welcher Knopf gedrückt wurde.
  andi  s1, s1, (1 << BTN_FIRE)
  beqz  s1, trap_handler_otherInt
  call button_isr
  li   s4, (1 << BTN_FIRE)        // Zurücksetzen des Button Latch Bits.
  sb   s4, BUTTON_LATCH(s0)
[...]
```

der EduCore-V keine Unterstützung für die Multiplikations- und Divisions-Befehle bietet, müssen diese Befehle mit kleinen Hilfsfunktionen substituiert werden. Eine Division kann z. B. wie in Beispiel 5.25 ersetzt werden.

Beispiel 5.25: Division in Spaceinvaders (interrupts.S). Der Rückgabewert des ADC wird auf 14 mögliche Raumschiffpositionen auf dem LCD aufgeteilt.

```
li  s0, GPIO_BASE
lbu a1, ANALOG(s0)
li  t0, 255/13
li  a0, 0
1: // a0 = a1 / t0
blt a1, t0, 2f
sub a1, a1, t0
addi a0, a0, 1
j 1b
2:
la t0, SPACESHIP_POS
sb a0, 0(t0)
call display_spaceship
```

5.6 Fazit

Es stellte sich heraus, dass Programme ohne IO Zugriffe, wie es bei Fibonacci (Beispiel 5.11) und Bubblesort (Beispiel 5.14) der Fall ist, in beiden Assemblersprachen gleich komplex sind. Zwar bietet der AVR Insruktionssatz einige Befehle, die Codezeilen einsparen können, weist aber insgesamt deutlich mehr Instruktionen als der RV32I auf.

Timer sind in RISC-V sehr simpel gehalten und überlassen der Software oder der Plattform die Verwantwortung für größere Strukturen. In AVR sind diese näher an die Hardware angelehnt.

Bei externen Interrupts kommt es stark auf die Größe der Plattform an. Der FE310 bietet sehr viel IO und hat einen dementsprechend mächtigen, aber komplexen Interrupt Controller. Der ATmega16 hat eine weniger mächtige Interruptstruktur und insgesamt weniger externe Interrupts. Das führt auch zu einer simpleren Konfiguration und Verwendung. Der EduCore-V implementiert nur die für die Lehre benötigten Interrupts, momentan sind dies die Knopfschalter. Das reduziert die Komplexität deutlich.

Durch die 32-Bit Architektur sind in RISC-V Assembler beispielsweise Adressierungen einfacher, weil nur ein Register, statt zwei in AVR, gebraucht werden. Es ist allerdings auch weniger simpel, Adressen, Bitmasken o. ä. im Kopf zu berechnen. So kann eine 8-Bit Maske wie z. B. `0b11110000` noch sehr anschaulich sein. Dahingegen sind 32-Bit binär dargestellt weniger sinnvoll und die Darstellung in Hexadezimal sollte stattdessen verwendet werden. Um z. B. in einer Maske ein bestimmtes Bit zu setzen, sollten Bitmanipulationen `(1 << 23)` genutzt werden. Das erscheint zunächst komplizierter, ist aber deutlich lesbarer, weniger fehleranfällig und in der Programmierung von eingebetteten Systemen üblich. Darüberhinaus hat RISC-V eine größere Architektur und kann später einen leichteren Einstieg in Betriebssysteme ermöglichen. Der Umgang mit Immediates ist in RISC-V Assembler durch das Immediate Befehlsformat mit nur 12 Bit etwas komplizierter, weil für größere Werte ein zusätzlicher Befehl (`lui` oder `auipc`) verwendet werden muss. Durch das Verwenden von Pseudoinstruktionen kann diese Schwierigkeit allerdings umgangen werden.

Zwar ist die RISC-V Architektur und deren Implementierungen sehr modern und zukunftsträchtig, jedoch sind sie noch lange nicht so etabliert und gut unterstützt im Vergleich zu AVR. Es gibt bisher nur wenig Literatur und kaum Libraries für externe Komponenten. Teilweise sind während der Implementierung Fehler in Datenblättern aufgefallen oder fehlende Informationen konnten nur durch längere Recherche in den Hardware-Design-Files gefunden werden.

Durch die freie Lizenz der RISC-V Spezifikation, ist es für jeden möglich, seine eigene Version eines RISC-V Cores zu implementieren. Mit dem EduCore-V konnten viele der Schwierigkeiten behoben und ein RISC-V Core speziell für die Bedürfnisse der Lehre entwickelt werden. Alles in allem bietet RISC-V damit eine offene, moderne und flexible ISA, die sich mit akzeptablem Mehraufwand für den Lehrbetrieb anpassen lässt.

6

Zusammenfassung

Ziel dieser Arbeit war es, die auf AVR basierenden praktischen Übungen der Lehrveranstaltung *Technische Grundlagen der Informatik 1* (*TGI1*) in RISC-V Assembler umzusetzen. Die Freiheit, Offenheit, Modernität und Modularität des Standards bieten dabei die größten Vorteile. Aktuelle Entwicklungen bringen bereits eine Vielzahl an Simulatoren und Entwicklungsboards hervor. Davon sind allerdings nur wenige für die Lehre ausgelegt und haben eine für diesen Zweck zu komplexe Architektur. Es wurde der Basisinstruktionssatz *RV32I*, der Umgang mit Peripherie und die Interrupt Architektur auf dem Entwicklungsboard *Hifive1RevB* vorgestellt. In einem Vergleich mit der AVR ISA konnte gezeigt werden, dass Assemblerprogramme in RISC-V Architektur leicht komplexer sind. Der Instruktionssatz *RV32I_Zicsr* ist damit für die Lehrveranstaltung ausreichend. In einigen Punkten, wie z. B. der geringeren Anzahl an Befehlen und den weniger Adressierungsarten, zeigt sich RISC-V mehr als RISC Architektur, als es bei AVR der Fall ist. Des Weiter hat sich herausgestellt, dass im Umgang mit Peripherie und Interrupts die Komplexität stark von der Plattform abhängt. Programme auf dem *Hifive1RevB*, die Interrupts verwenden, sind deutlich komplexer und eignen sich nicht für die Lehre. So konnte der eigens für die Lehre entwickelte EduCore-V hier mit einer leicht verständlichen Architektur weniger komplexe Assemblerprogramme hervorbringen. Das verwendete *Basys 3* Entwicklungsboard bringt viel der benötigten Peripherie mit, was den Umgang weiter vereinfacht. Ein Nachteil gegenüber größeren Cores ist eine fehlende Debug-Schnittstelle, wie z. B. JTAG. Es können zwar Informationen über die UART oder die VGA Schnittstelle ausgegeben werden, aber Features wie Single-Step Debugging oder Breakpoints sind noch nicht implementiert. Eine Ergänzung dieses und weiterer Features könnten in der Zukunft umgesetzt werden.

Alles in allem lässt sich sagen, dass die Lehrveranstaltung *TGI1* ohne größere Probleme mit der RISC-V ISA umgesetzt werden kann.

Eben durch die Möglichkeit der Eigenimplementierung eines Prozessors, wird viel Potenzial für die Lehre gewonnen. Studierende können den Prozessor von Grund auf verstehen lernen, selber verändern oder ergänzen und mit Assembler oder einer Hochsprache programmieren. Dabei wird eine moderne und zukunftsträchtige ISA erlernt. Die Offenheit und Freiheit des RISC-V Standards fördert die open-source Community, die auch im Bereich der Lehre aktiv ist.

A

Tutorials

In diesem Abschnitt werden Hilfestellungen zum Umgang mit dem Entwicklungsboard *Hifive1RevB* und dem *EduCore-V* gegeben.

A.1 Hifive1RevB

Alle Informationen zum Einrichten des Boards können auch auf der Produktseite des Herstellers gefunden werden[42]. Weitere Hilfestellungen und Problemlösungen können in *SiFives* Forum gefunden werden[43].

Entwicklungsumgebung

Von der Herstellerseite kann das *Freedom-Studio*, ein modifiziertes Eclipse, bezogen werden[44]. Es enthält neben der IDE auch eine vorkompilierte *GNU-GCC* Toolchain, sowie das *Freedom-SDK*.

Programmierung

In diesem Abschnitt wird gezeigt, wie das Entwicklungsboard programmiert wird. Dies wird anhand eines Linux Betriebssystems gezeigt. Für andere Betriebssysteme kann eine Anleitung im *Getting-Started-Guide* gefunden werden (*SiFive HiFive1 Rev B Getting Started Guide* 2019, Kap. 6).

Zunächst muss festgestellt werden, ob die JLink Treiber für die serielle Schnittstelle funktionieren.

1. Das Entwicklungsboard mittels micro USB mit dem PC verbinden.
2. überprüfen, ob folgende Devices gefunden werden:
   - `/dev/ttyACM3` (Serial zum FE310)
   - `/dev/ttyACM4` (Serial zum ESP)
   - `/dev/sdb` (Flash speicher für Programme)

[42] `https://www.sifive.com/boards/hifive1-rev-b`, November 2019
[43] `https://forums.sifive.com/`, November 2019
[44] `https://www.sifive.com/boards`, November 2019

Wird kein passender Treiber gefunden, kann er wie in (*SiFive HiFive1 Rev B Getting Started Guide* 2019, Kap. 6) installiert werden.

Ein Weg zur Programmierung ist die direkte Verwendung des SDK.

1. In den Ordner der SDK wechseln
   `cd FreedomStudio20190802/SiFive/freedom-e-sdk-v201908/`
2. Im `software/` Ordner befinden sich Beispielprogramme. Nach derselben Struktur können so auch eigene Programme erstellt werden.
3. Mit `make` kann das Programm kompiliert, programmiert, debuggt und simuliert werden. Der Befehl `make help` erklärt dies.

Alternativ kann die IDE verwendet werden: `./FreedomStudio20190802/FreedomStudio`. Im FreedomStudio kann dann über den Dialog *File - New - Freedom E SDK Software Project* ein neues Projekt erstellt werden. In dem Projekt kann eine Assemblerdatei mit der Dateiendung `.S` erstellt werden. Zur Programmierung können die in Eclipse üblichen Schaltflächen verwendet werden. Möchte man externe Libraries benutzen, so müssen diese entweder dem Projektordner hinzugefügt werden oder dem Makefile muss ein `-I` mit dem entsprechenden Pfad hinzugefügt werden. Im beiliegenden Ordner `riscv-assembler/hifive1revb/src/include` sind die in dieser Arbeit erstellten Library Dateien zu finden.

Memorymap

Die Memorymap des FE310-G002 ist in Tabelle A.1 zu sehen.

Tabelle A.1: Memorymap des FE310-G002 (*SiFive FE310-G002 Manual* 2019, Kap. 4)

Startadresse	Endadresse	Beschreibung
0x00000000	0x00000FFF	Debug
0x00001000	0x00001FFF	Mode Select
0x00002000	0x00002FFF	Reserved
0x00003000	0x00003FFF	Error Device
0x00004000	0x0000FFFF	Reserved
0x00010000	0x00011FFF	Mask ROM (8 KiB)
0x00012000	0x0001FFFF	Reserved
0x00020000	0x00021FFF	OTP Memory Region
0x00022000	0x001FFFFF	Reserved
0x02000000	0x0200FFFF	CLINT
0x02010000	0x07FFFFFF	Reserved
0x08000000	0x08001FFF	E31 ITIM (8 KiB)
0x08002000	0x0BFFFFFF	Reserved
0x0C000000	0x0FFFFFFF	PLIC
0x10000000	0x10000FFF	AON
0x10001000	0x10007FFF	Reserved
0x10008000	0x10008FFF	PRCI
0x10009000	0x1000FFFF	Reserved
0x10010000	0x10010FFF	OTP Control
0x10011000	0x10011FFF	Reserved
0x10012000	0x10012FFF	GPIO
0x10013000	0x10013FFF	UART 0
0x10014000	0x10014FFF	QSPI 0
0x10015000	0x10015FFF	PWM 0
0x10016000	0x10016FFF	I2C 0
0x10017000	0x1002FFFF	Reserved
0x10023000	0x10023FFF	UART 1
0x10024000	0x10024FFF	SPI 1
0x10025000	0x10025FFF	PWM 1
0x10026000	0x10033FFF	Reserved
0x10034000	0x10034FFF	SPI 2
0x10035000	0x10035FFF	PWM 2
0x20000000	0x3FFFFFFF	QSPI 0 Flash (512MiB), Off-Chip Non-Volatile Memory
0x80000000	0x80003FFF	E31 DTIM (16KiB), On-Chip Volatile Memory

A.2 EduCore-V

Dieser Abschnitt zeigt die Einrichtung einer Entwicklungsumgebung für den EduCore-V (Aljnabi, 2019).

Voraussetzungen

Folgende Programm werden benötigt und sollten zuvor installiert sein.

- Bash / Shell, zur Verwendung der Shellskripte. Unter Windows kann die Git Bash verwendet werden.
- Vivado, zur Übertragung des Bitstreams.
- Java, zur Seriellen Programmierung des Cores.

Hochladen des Bitstreams

Damit der Bitstream des Cores auf das FPGA hochgeladen werden kann, muss Vivado installiert sein. Gegebenenfalls muss der Pfad der Vivado Installation in der Datei `riscv-assembler/educorev/core/upload.sh` angepasst werden. Ansonsten muss nur dieses Shellskript ausgeführt werden. Das erfolgreiche Hochladen ist an der Ausgabe `End of startup status: HIGH` zu erkennen.

Kompilierung und Programmierung

Die Compiler Toolchain kann entweder vorkompiliert heruntergeladen werden[45] oder selber kompiliert werden[46]. In beiden Fällen sollte die Toolchain zur *PATH* Umgebungsvariable hinzugefügt werden:

```
export PATH=$PATH:~/riscv-assembler/hifive1revb/FreedomStudio20190802/\
                    SiFive/riscv64-unknown-elf-gcc-8.3.0-2019.08.0/bin/
```

Um ein Assemblerprogramm maschinenlesbar zu machen, müssen folgende Schritte durchgeführt werden. Zuerst wird es mit gcc vorverarbeitet, kompiliert und gelinkt. Danach wird mit objcopy die Textsektion in eine Hexdatei extrahiert. Abschließend wird über die UART-Schnittstelle die Hexdatei dem Core übermittelt.

```
riscv64-unknown-linux-gnu-gcc src/01_Lauflicht/01_Lauflicht.S ./lib/start.S
    -nostartfiles
-march=rv32i -mabi=ilp32 -I./include -Wl,-T ./script.lds -static -o
    src/01_Lauflicht/bin/program.bin

riscv64-unknown-linux-gnu-objcopy src/01_Lauflicht/bin/program.bin
    --dump-section .text=src/01_Lauflicht/bin/program.hex

java -jar src/01_Lauflicht/bin/program.hex
```

[45] https://www.sifive.com/boards, November 2019
[46] https://github.com/riscv/riscv-gnu-toolchain, November 2019

Die verwendeten Compiler-Flags sind in Tabelle A.2 kurz erklärt. Das Shellskript `compile.sh` enthält diese Befehle und kann mit einem Ordner als Argument aufgerufen werden: `./compile.sh src/01_Lauflicht/`

Tabelle A.2: Verwendete GCC Compiler Flags

[...]/01_Lauflicht.S	Eine oder mehrere Dateien die kompiliert werden soll
./lib/start.S	Assemblerprogramm, das die entry Methode enthält und den Core in einen wohldefinierten Zustand bringt.
-nostartfiles	Unterbindet, dass die standard system startup Dateien verwendet werden
-march=rv32i	Wählt die RISC-V Architektur aus
-mabi=ilp32	Wählt das RISC-V Application Binary Interface aus
-I./include	Fügt den Ordner zur Liste der für Headerdateien durchsuchten Pfade hinzu.
-Wl,	Leitet einen Befehl an den Linker weiter
-T ./script.lds	Linkerbefehl zur Verwendung des Linkerscripts
-static	Verhindert dynamisches Linking
-o [...]/bin/program.bin	Definiert die Ausgabe Datei

Memorymap

Tabelle A.3 zeigt die Memorymap des EduCore-V.

Tabelle A.3: Memorymap des EduCore-V (Aljnabi, 2019)

		+0x0	+0x1	+0x2	+0x3
0x0000	RAM	.text segment			
		.data segment			
		Stack			
0x6F80	GPIO	LED		*unused*	
0x6F84		SWITCH		*unused*	
0x6F88		DDR_A	PIN_A	PORT_A	*unused*
0x6F8C		DDR_B	PIN_B	PORT_B	*unused*
0x6F90		DDR_C	PIN_C	PORT_C	*unused*
0x6F94		SEG_0	SEG_1	SEG_2	SEG_3
0x6F98		BUTTON	BUTTON_LATCH	BUTTON_P_CONF	BUTTON_N_CONF
0x6F9C		ANALOG	*unused*		
0x6FA0		MTIME_L			
0x6FA4		MTIME_H			
0x6FA8		MTIMECMP_L			
0x6FAC		MTIMECMP_H			
0x6FB0		UART_RX	UART_TX	*unused*	
0x6FB4		*unused*			
0x7000	VRAM	Tiles			
0x7800		Background Layer			
0x7B00		Background Color			
0x7E00		Sprites			
0x7FE0		Palette			
0x7FFF					

B

Befehlsübersicht

Die folgenden Tabellen B.1 und B.2 listen sämtliche Instruktionen des RV32I Basisinstruktionssatz, der Zicsr Erweiterung und in dieser Arbeit verwendete Befehle aus der Privileged Architecture auf.

Tabelle B.1: Befehle des Basisinstruktionssatzes RV32I und der Zicsr Erweiterung, entnommen der RISC-V Spezifikation (Waterman und Asanovic, 2019a, Kap. 2, 9, 25)

Instruction	Pseudo-code
lui rd, I	rd = I << 12
auipc rd, I	rd = pc + I << 12
jal rd, I	rd = pc + 4; pc = pc + I
jalr rd, rs1, I	rd = pc + 4; pc = rs1 + I
beq rs1, rs2, I	if (rs1 == rs2) pc = pc + I
bne rs1, rs2, I	if (rs1 != rs2) pc = pc + I
blt rs1, rs2, I	if (rs1 <s rs2) pc = pc + I
bge rs1, rs2, I	if (rs1 >=s rs2) pc = pc + I
bltu rs1, rs2, I	if (rs1 < rs2) pc = pc + I
bgeu rs1, rs2, I	if (rs1 >= rs2) pc = pc + I
lb rd, I(rs1)	rd = M[rs1+I]; rd[31:8] = rd[7]
lh rd, I(rs1)	rd = M[rs1+I]; rd[31:16] = rd[15]
lw rd, I(rs1)	rd = M[rs1+I]
lbu rd, I(rs1)	rd = M[rs1+I]; rd[31:8] = 0
lhu rd, I(rs1)	rd = M[rs1+I]; rd[31:16] = 0
sb rs2, I(rs1)	M[rs1+I] = rs2 & 0xFF
sh rs2, I(rs1)	M[rs1+I] = rs2 & 0xFFFF
sw rs2, I(rs1)	M[rs1+I] = rs2
addi rd, rs1, I	rd = rs1 + I
slti rd, rs1, I	rd = rs1 <s I
sltiu rd, rs1, I	rd = rs1 < I
slli rd, rs1, I	rd = rs1 << (I % 32)
srli rd, rs1, I	rd = rs1 >> (I % 32)
srai rd, rs1, I	rd = rs1 >>> (I % 32)
xori rd, rs1, I	rd = rs1 ^ I
ori rd, rs1, I	rd = rs1 | I
andi rd, rs1, I	rd = rs1 & I
add rd, rs1, rs2	rd = rs1 + rs2
sub rd, rs1, rs2	rd = rs1 − rs2
slt rd, rs1, rs2	rd = rs1 <s rs2
sltu rd, rs1, rs2	rd = rs1 < rs2
sll rd, rs1, rs2	rd = rs1 << (rs2 % 32)
srl rd, rs1, rs2	rd = rs1 >> (rs2 % 32)
sra rd, rs1, rs2	rd = rs1 >>> (rs2 % 32)
xor rd, rs1, rs2	rd = rs1 ^ rs2
or rd, rs1, rs2	rd = rs1 | rs2
and rd, rs1, rs2	rd = rs1 & rs2
csrrw rd, csr, rs1	rd = csr; csr = rs1
csrrs rd, csr, rs1	rd = csr; csr |= rs1
csrrc rd, csr, rs1	rd = csr; csr &= !rs1
csrrwi rd, csr, imm	rd = csr; csr = imm
csrrsi rd, csr, imm	rd = csr; csr |= imm
csrrci rd, csr, imm	rd = csr; csr &= !imm
wfi	Waits until an interrupt occurs
mret	Returns from a trap handler

Tabelle B.2: Auflistung der Pseudoinstruktionen aus der RISC-V Spezifikation (Waterman und Asanovic, 2019a, Kap. 26)

Pseudoinstruction	Base Instruction(s)	Meaning
la rd, symbol	auipc rd, delta[31:12] + delta[11] addi rd, rd, delta[11:0]	Load absolute address, where $delta = symbol - pc$
l{b|h|w} rd, symbol	auipc rd, delta[31:12] + delta[11] l{b|h|w} rd, delta[11:0](rd)	Load global where $delta = symbol - pc$
s{b|h|w} rd, symbol, rt	auipc rt, delta[31:12] + delta[11] s{b|h|w} rd, delta[11:0](rt)	Store global where $delta = symbol - pc$
nop	addi x0, x0, 0	No operation
li rd, imm	lui rd, imm[31:12] + imm[11] addi rd, rd, imm[11:0]	Load immediate
mv rd, rs	addi rd, rs, 0	Copy register
not rd, rs	xori rd, rs, −1	One's complement
neg rd, rs	sub rd, x0, rs	Two's complement
seqz rd, rs	sltiu rd, rs, 1	Set if = zero
snez rd, rs	sltu rd, x0, rs	Set if ≠ zero
sltz rd, rs	slt rd, rs, x0	Set if < zero
sgtz rd, rs	slt rd, x0, rs	Set if > zero
beqz rs, offset	beq rs, x0, offset	Branch if = zero
bnez rs, offset	bne rs, x0, offset	Branch if ≠ zero
blez rs, offset	bge x0, rs, offset	Branch if ≤ zero
bgez rs, offset	bge rs, x0, offset	Branch if ≥ zero
bltz rs, offset	blt rs, x0, offset	Branch if < zero
bgtz rs, offset	blt x0, rs, offset	Branch if > zero
bgt rs, rt, offset	blt rt, rs, offset	Branch if >
ble rs, rt, offset	bge rt, rs, offset	Branch if ≤
bgtu rs, rt, offset	bltu rt, rs, offset	Branch if >, unsigned
bleu rs, rt, offset	bgeu rt, rs, offset	Branch if ≤, unsigned
j offset	jal x0, offset	Jump
jal offset	jal x1, offset	Jump and link
jr rs	jalr x0, 0(rs)	Jump register
jalr rs	jalr x1, 0(rs)	Jump and link register
ret	jalr x0, 0(x1)	Return from subroutine
call offset	auipc x1, offset[31:12] + offset[11] jalr x1, offset[11:0](x1)	Call far-away subroutine
tail offset	auipc x6, offset[31:12] + offset[11] jalr x0, offset[11:0](x6)	Tail call far-away subroutine
csrr rd, csr	csrrs rd, csr, x0	
csrw csr, rs	csrrw x0, csr, rs	
csrs csr, rs	csrrs x0, csr, rs	
csrc csr, rs	csrrc x0, csr, rs	
csrwi csr, imm	csrrwi x0, csr, imm	
csrsi csr, imm	csrrsi x0, csr, imm	
csrci csr, imm	csrrci x0, csr, imm	

Glossar

ABI Application Binary Interface. 11–13, 18
ADC Analog Digital Converter. 37, 40, 43, 44
ALU Airthmetic Logic Unit. 31

BCD Binary-Coded Decimal. 42

CISC Complex Instruction Set Computer. 3
CLINT Core-Local Interruptor. 22, 24, 25
CSR Control Status Register. 4, 5, 22, 23

FPGA Field-Programmable Gate Array. 6, 7, 9, 42

GPIO General-Purpose Input/Output. 9, 20, 22, 25–30, 36–38, 42, 43
GUI Graphical User Interface. 6, 7

HART HARDware Thread. 24

I2C Inter-Integrated Circuit. 9, 20, 22, 27, 40
IDE Integrated Development Environment. 11, 47, 48
IO Input Output. 30, 42, 44, 45
ISA Instruction Set Architecture. 1–4, 45, 46
ISR Interrupt-Service-Routine. 38, 39
ISS Instruction Set Simulator. 6, 7

JTAG Joint Test Action Group. 9, 46

LCD Liquid Crystal Display. 7, 37, 41, 44
LED Light-Emitting Diode. 7, 29, 30
LSB Least Significant Bit. 16

MSB Most Significant Bit. 14, 15, 24

PC Program Counter. 13, 15, 16, 25, 26, 34, 38
PLIC Platform-Level Interrupt Controller. 22, 24, 27, 28, 38, 39
PWM Pulse Width Modulation. 9, 27

RISC Reduced Instruction Set Computer. 3, 17, 29, 38, 46
RTC Real Time Clock. 25, 27, 38, 39, 41
RTOS Real-Time Operating Systems. 38

SDK Software Development Kit. 11, 48
SoC System on a Chip. 6–8

SPI Serial Peripheral Interface. 9, 27, 37
SREG Status Register. 31, 38

UART Universal Asynchronous Receiver Transmitter. 9, 20, 27, 37, 46
USB Universal Serial Bus. 9, 11

VGA Video Graphics Array. 42, 46

Literatur

ADS1115 I2C-Compatible, 16-Bit ADC Datasheet (2018). Online erhätlich unter http://www.ti.com/lit/ds/symlink/ads1115.pdf; Zugegriffen am 24.09.2019.

Akram, A. und Sawalha, L. (2019). A Survey of Computer Architecture Simulation Techniques and Tools. IEEE Access.

Aljnabi, W. (2019). Entwicklung eines RISC-V Cores für die Lehre. Bachelorarbeit. Universität zu Lübeck.

Atmel ATmega16 Complete Datasheet (2010). Online erhätlich unter https://www.microchip.com/wwwproducts/en/ATmega16#datasheet-toggle; Zugegriffen am 24.09.2019.

Brinkschulte, U. und Ungerer, T. (2010). Mikrocontroller und Mikroprozessoren. Bd. 3. Springer.

HD44780, Dot Matrix Liquid Crystal Display Controller/Driver Datasheet (1998). Online erhätlich unter http://www.ti.com/lit/ds/symlink/ads1115.pdf; Zugegriffen am 24.09.2019. Hitachi.

Knuth, D.E. (1997). The art of computer programming. Bd. 3. Pearson Education.

Lee, Y., Waterman, A., Avizienis, R., Cook, H., Sun, C., Stojanović, V. und Asanović, K. (2014). A 45nm 1.3 GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators. In ESSCIRC 2014-40th European Solid State Circuits Conference (ESSCIRC), IEEE, S. 199–202.

Patterson, D. und Waterman, A. (2017). The RISC-V Reader: An Open Architecture Atlas. Strawberry Canyon.

Patterson, D.A. und Hennessy, J.L. (2017). Computer Organization and Design RISC-V Edition: The Hardware Software Interface. Morgan Kaufmann.

Schiffmann, W. (2006). Technische Informatik 2: Grundlagen der Computertechnik. Springer-Verlag.

SiFive FE310-G002 Manual (2019). Online erhätlich unter https://www.sifive.com/documentation; Zugegriffen am 12.09.2019.

SiFive HiFive1 Rev B Getting Started Guide (2019). Online erhätlich unter https://www.sifive.com/documentation; Zugegriffen am 12.09.2019.

Traber, A., Zaruba, F., Stucki, S., Pullini, A., Haugou, G., Flamand, E., Gurkaynak, F.K. und Benini, L. (2016). PULPino: A small single-core RISC-V SoC. In 3rd RISCV Workshop,

Waterman, A. und Asanovic, K. (2019a). *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Version 2.2*. Online erhätlich unter `https://riscv.org/specifications/`; Zugegriffen am 22.08.2019.

Waterman, A. und Asanovic, K. (2019b). *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 2.2*. Online erhätlich unter `https://riscv.org/specifications/`; Zugegriffen am 22.08.2019.

Waterman, A.S. (2016). Design of the RISC-V instruction set architecture. Diss. UC Berkeley.

Zaruba, F. und Benini, L. (2019). The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-ready 1.7 GHz 64bit RISC-V Core in 22nm FDSOI Technology. arXiv preprint arXiv:1904.05442.