Universität Hamburg
Fachbereich Informatik

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

# Entwurf und Implementation eines Mikrorechners

Projekt Bericht

## Entwurf, Realisierung und Programmierung eines Mikrorechners

Benedikt Ostendorf, Anran Wang, Jasper Schwarzwald
Matr.Nr. 6981708, 6923546, 6948085
{7ostendo, 6awang, 6schwarz}@informatik.uni-hamburg.de

25. August 2020

# Inhaltsverzeichnis

## Abstract

In diesem Bericht diskutieren wir die Struktur einer einfachen zentralen Recheneinheit mit mehreren Registern und einer MIPS (Microprocessor without Interlocked Pipelined Stages)-ähnlichen Architektur. Danach implementieren wir die oben genannte Architektur in VHDL (Very High Speed Integrated Circuit Hardware Description Language) sowie einen Befehlssatzemulator in Haskell.

In this essay, we discuss the structure of a simple central computing unit, with multiple registers and MIPS (Microprocessor without Interlocked Pipelined Stages)-like architecture. We also propose an implementation of the aforementioned architecture in VHDL (Very High Speed Integrated Circuit Hardware Description Language) as well as an instruction emulator in Haskell.

# Einleitung

Durch die Entwicklung von Computer-Technologien wird auch der Technologie von SoC (Source of Computing) immer mehr Relevanz zugeschrieben. Um in diesem Bereich einsteigen zu können, muss man zunächst simple Architekturen verstehen.

In diesem Bericht untersuchen wir zunächst eine einfache Struktur eines Computers und diskutieren dann die klassische Fetch-Decode-Execute Schleife, bevor wir einige Optimierungsmöglichkeiten in einer CPU vorstellen.

Im zweiten Teil schlagen wir eine Implementierung eines solchen Systems vor, die auf der MIPS (Microprocessor without Interlocked Pipelined Stages) Architektur basiert. Wir zeigen unser Design im Detail mit dem Schaltplan, stellen dann die, für unsere Implementierung verwendete, Sprache VHDL (Very High Speed Integrated Circuit Hardware Description Language) vor und zeigen schließlich die Struktur unserer Implementierung. Der Quellcode ist in einem offenen Git-Repository verfügbar. [1]

Im dritten Teil stellen wir zunächst eine auf RISC-V basierende ISA (Instruction Set Architecture) vor. Wir geben dann eine kurze Einführung in die Sprache Haskell, mit welcher wir einen Emulator implementieren. Schließlich zeigen wir die Implementierung eines Befehlssatzemulators im Detail. Der gesamte Quellcode ist in einem offenen Git-Repository verfügbar. [2]

Abschließend fassen wir den Bericht zusammen und diskutieren mögliche Future Work.

[1] https://git.mafiasi.de/16awang/pj-vhdl-impl.git
[2] https://gitlab.com/rnxpyke/riscvemu

## Ein simpler Aufbau eines Computers

Die erste formale Definition eines Computers, die uns, wie wahrscheinlich den meisten anderen Studierenden, im Studium begegnet ist, nennt sich Turingmaschine. Sie ist ein mathematisches Modell, welches eine einfache Sammlung von Zuständen und Zustandsübergängen und ein lineares Speichermodell, ein sogenanntes Tape besitzt. Die Einfachheit dieses Modells macht es für die theoretische Informatik sehr zugänglich. Besonders die Eigenschaft, dass jeder Zustandsübergang vergleichbar mit einem einfachen Tabellen-lookup ist, der einfach in elektronische Schaltwerke übersetzbar wäre, macht sie zur Analyse von Zeitkomplexität zu einem hilfreichen Werkzeug. Ein Problem der Turingmaschine ist jedoch, dass selbst für moderat komplexe Probleme die Anzahl der Zustände stark zunimmt, sodass es unpraktisch ist diese alle aufzuschreiben, geschweige denn sie in Hardware zu überführen. Oftmals wird in Argumenten über abstraktere Problemklassen von Berechnungsmodellen gesprochen, die sich selbst simulieren können. Beim Lambda-Kalkül von (Church [1941] 1985) ist dies relativ simpel zu erreichen (Mogensen 1992). Allerdings ist in unseren Vorlesungen so eine Repräsentation noch nie für eine Turing Maschine durchgeführt worden. Ein anderes Problem ist, dass ohne eine solche Selbst-Interpretation für jedes Problem ein andere Menge von Zuständen bzw. Zustandsübergängen benötigt wird, was für ein generelles und physikalisches Berechnungsmodell ungeeignet ist.

Als Alternative zur Turingmaschine als Berechnungsmodell wird für physikalische Hardware die Von-Neumann Architektur (Neumann 1993) verwendet. Im Unterschied zur Turingmaschine hat dieses Modell einen adressierbaren Speicher, sowie ein Kontroll- und ein Rechenwerk. Im Speicher des Von-Neumann Rechners liegen sowohl die zu verarbeitenden Daten als auch die Instruktionen, wie der Rechner diese verarbeiten soll. Vorstellen kann man sich das Kontrollwerk wie eine Zustandsübergangsfunktion einer Turing Maschine (TM), die eine beliebige andere TM simuliert. Im Rechenwerk sind dann einfache Subroutinen wie das Addieren oder Subtrahieren von Zahlen eingebaut, sodass für elementare Operationen nur noch eine Instruktion statt 11 Zustände einer TM gebraucht werden.

Damit das Von-Neumann Modell genauso mächtig wie eine TM ist, kann man sich leicht überlegen, dass man den aktuellen TM Zustand und die aktuelle Zelle des Tapes in Zellen im Von-Neumann Speicher übersetzen muss. Abhängig vom Zustand und Datenzelleninhalt muss also der neue Zustand und die neue Datenzelle ermittelt, sowie die aktuelle Zelle überschrieben werden können. Auf welche Art und Weise dieser Übergang konkret passiert, hängt von der jeweiligen Implementation des Rechners ab.

Moderne Computer besitzen natürlich noch viel mehr als nur diese drei Bausteine, aber Speicher, Kontrolleinheit und Rechenwerk lassen sich in noch so ziemlich jedem modernen Rechner wiederfinden. Verschiedene Optimierungen und Architekturänderungen können zusätzlich schnellere Ausführung von Instruktionen

und Lese sowie Schreiboperationen ermöglichen. Zusätzliche Geschwindigkeit wird durch nebenläufige Ausführung von verschiedenen Programmen auf einem Computer ermöglicht.

Typischerweise werden Instruktionen zum Lesen beziehungsweise Schreiben von Werten in Speicher, zum Addieren/Subtrahieren, Vergleichen von Werten und zum Wechseln zu anderen Zuständen bzw. Instruktionen benötigt.

## N-Register Maschinen

Die meisten Architekturen speichern Zwischenwerte in Registern. Instruktionen werden abhängig davon kategorisiert, wie viele Register gleichzeitig referenziert werden können. Dabei spricht man typischerweise von 0 bis 3-Register Instruktionen.

Eine 3-Register Instruktion kann zwei Argumente für beispielsweise eine Addition entgegennehmen, und an ein anderes Ziel schreiben. Eine 2-Register Instruktion überschreibt typischerweise eines der Argumente. Eine 1-Register Instruktion arbeitet mit einem speziellen Akkumulator Register. Es können Werte in das Akkumulator Register geladen werden, oder daraus gelesen werden. Eine Operation wie eine Addition wird mit dem Register aus der Instruktion und den Akkumulator ausgeführt, und das Ergebnis zurück in den Akkumulator geschrieben. Eine 0-Register Maschine arbeitet stackbasiert. Eine Operation nimmt die ersten beiden Werte vom Stack und schreibt das Ergebnis wieder auf den Stack drauf.

## Fetch - Decode - Execute Loop

Eine herkömmliche Art, eine CPU zu strukturieren ist es, sich das aktuelle Codewort in einem speziellem Register, dem Instruction Pointer, zu speichern. Ein regelmäßiges Signal, auch Clock genannt, treibt die CPU dazu, die verschiedenen Rechenschritte auszuführen. Dabei wird der semantische Effekt einer Instruktion in kleinere Schritte unterteilt.

- Im Fetch-Step wird zuerst die aktuelle Instruktion aus dem Speicherelement geholt, auf das der Instruction Pointer zeigt.
- Im Decode-Step werden die Bits der Instruktion analysiert und festgestellt, welche Register gelesen werden sollen oder ob es Interaktionen mit dem Speicher gibt oder welche Operation im Rechenwerk ausgeführt werden soll.
- Im Execute-Step werden die von der Instruktion codierten Befehle dann ausgeführt und der Instruction Pointer auf die nächste Instruktion gesetzt.

# Optimierungen von sequenziellen Rechnern

## Pipelining

In einer CPU wird das Ausführen einer Instruktion also in kleinere Schritte, wie zum Beispiel das Fetch - Decode - Execute zerlegt. Für jeden dieser Schritte gibt es einen dedizierten Hardwarebaustein, der die Funktionalität abbildet. Wenn Instruktionen nun sequenziell ausgeführt werden, ist immer nur einer dieser Bausteine aktiv. Zuerst der Baustein der die Instruktion holt, dann einer der sie dekodiert, und so weiter. Die meisten Bausteine in unserer Fetch - Decode - Execute Pipeline sind also untätig. Sie warten nur darauf, dass die anderen mit ihrer Aufgabe fertig sind.

Oftmals steht aber schon fest welche Instruktionen als nächstes ausgeführt werden. Dies ist z.B. immer dann der Fall, wenn kein Sprung zu einer anderen Instruktion ausgeführt wird, also der Instruktion Pointer nur inkrementiert wird. Wenn dies der Fall ist muss der Fetch-Baustein aber nicht untätig sein. Er kann noch während die aktuelle Instruktion dekodiert wird die nächste holen. Ähnlich können auch die Decode und Execute Schritte gleichzeitig ausgeführt werden. Wenn aufeinander folgende Instruktionen keine gemeinsamen Register oder ähnliches verwenden können also nach einer kleinen Anlaufphase alle Bausteine in unserer Pipeline gleichzeitig arbeiten. Mit den drei Schritten Fetch - Decode - Execute verdreifacht sich damit im Optimalfall die Berechnungsgeschwindigkeit der CPU.

# Implementation in VHDL

## Unser Design

Wie in Abbildung 1 zu sehen ist, kann unser Entwurf grob in vier Teilen zerlegt werden: der Controller, der Datenpfad, der Speicher für Befehlssätze, und der Speicher für Daten.

### Controller

Im Controller-Teil liegen ein Main-Decoder (**DCD(main)**) und ein ALU-Decoder (**DCD(alu)**). Der Main-Decoder ist die zentrale Steuerungseinheit des CPUs. Er nimmt die 26-31 Bits (inklusiv) vom Befehlssatz als Input und gibt Steuerungsinformation aus, insbesondere was der ALU-Decoder dekodieren soll, ob man im Speicher schreiben oder lesen soll, zu welchem Register man schreiben soll, ob es sich jetzt um ein Springen oder Abspalten handelt, usw.

Dem ALU-Decoder wird ein `aluop` vom Main-Decoder, sowie die 0-5 Bits vom Befehlssatz zugeteilt. Als Output gibt der ALU-Decoder `alucrtl` an ALU, damit diese entscheiden kann, ob sie eine Addition/Subtraktion, oder eine logische Operationen durchführen soll.

### Datenpfad

Im Datenpfad findet man die ALU, den Register (**REG**), einen Flopper (**FLOPR**), mehrere Multiplexer (**MUX**) und Adder (**ADD**), sowie andere Recheneinheiten (**SIGNEXT**, **SL2**).

Das Register wird durch ein Clock `clk` getaktet. Die Adressen der zu ladenden Register sind von den 21-25, beziehungsweise 16-20 Bits vom Befehlssatz vorgegeben. Das Zielregister wird durch `writereg` angegeben, und deren zu schreibenden Daten wird durch `result` angegeben, welches entweder aus ALU Output oder dem Speicher stammt.

Der erste Input von der ALU `srca` wird immer vom Register geladen. `srcb` kommt entweder aus dem Register, oder aus dem Immediate-Wert. Dies wird durch den Main-Decoder gesteuert. Das Rechenergebnis wird gegebenenfalls im Register beziehungsweise in den Speicher geschrieben.

### Speicher für Befehlssätze

Die Adresse des nächsten Befehlssatzes ist durch `pc` gegeben, dementsprechend wird die 32-stellige Instruktion geladen und zugeteilt. Der Flopper, der die neue Adresse ausgibt, wird ebenfalls durch eine Clock gesteuert, und somit wird nur maximal ein Befehlssatz in einem Clock-Cycle geladen.

### Speicher für Daten

Der Datenspeicher wird vom Controller informiert, ob jetzt ein Schreiben stattfindet. Falls ja, wird die Adresse wo die Daten gespeichert werden soll durch ALU Output angegeben, und die Daten selbst werden vom Register gelesen.

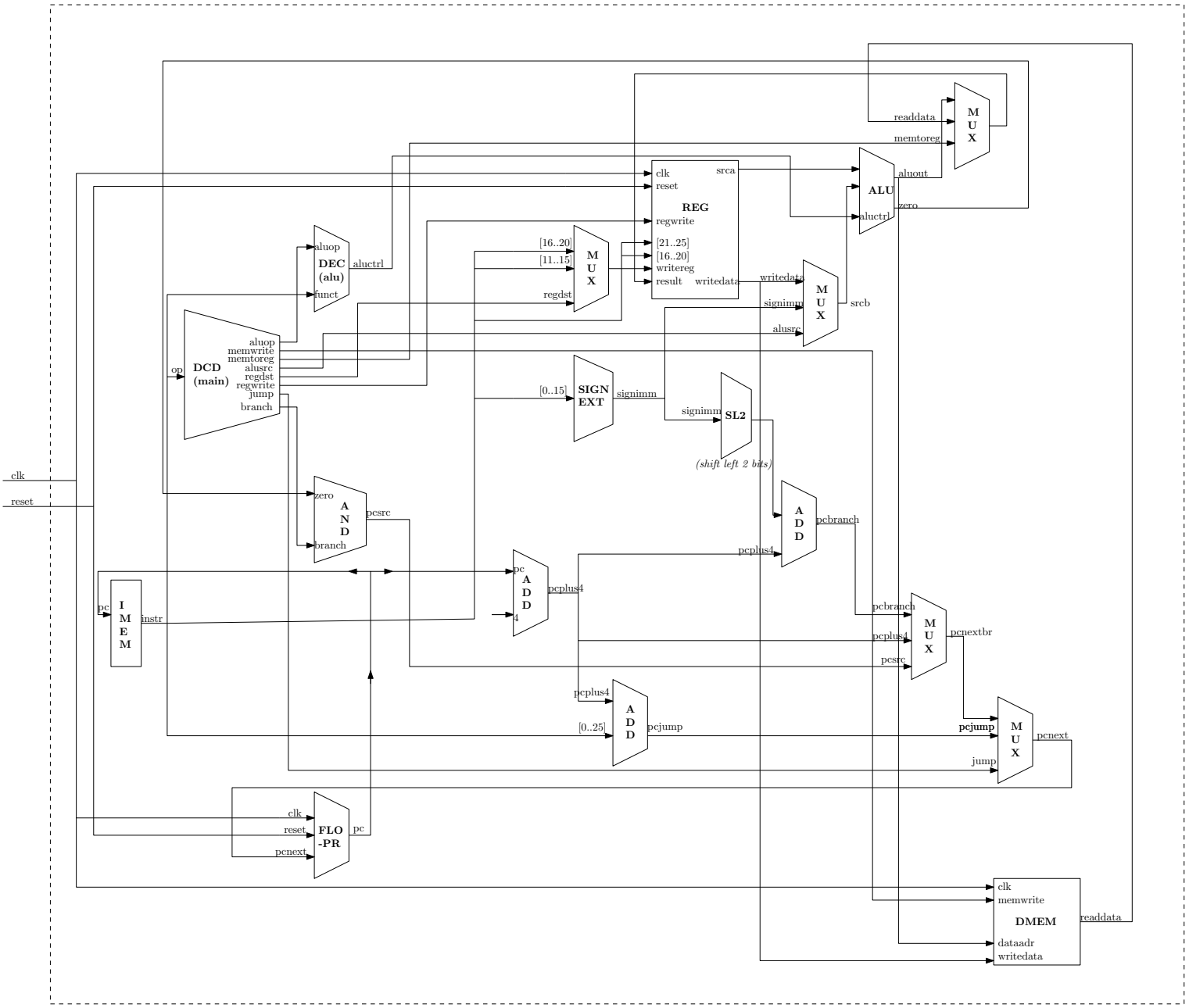


Abbildung 1: Schaltplan unseres Designs

## Die Sprache VHDL

**VHDL** (VHSIC-HDL, Very High Speed Integrated Circuit Hardware Description Language) ist eine bekannte Programmiersprache zur Beschreibung der Hardwa-

restruktur. VHDL hat 5 Design-Einheiten: *Entity*, *Architecture*, *Configuration*, *Package Declaration*, *Package Body*. (Eduvance, o. J.)

In der Entity-Deklaration werden die Schnittstellen (Ports) der Entity aufgelistet. Architektur beschreibt das Verhalten, und Configuration ermöglicht die Definition mehrerer Architekturen von einer Entity. Die Packages fassen unter anderem Funktionen und Datentypen zusammen, die mehrmals in verschiedenen Dateien eines VHDL-Projektes erscheinen.

Eine VHDL-Datei entspricht einem Teil der zu beschreibenden Hardware, beispielsweise ein Adder, ein ALU (arithmetic-logic unit), oder eine komplette CPU, je nach Stil des Designs: strukturell, nach Datenfluss, oder nach Verhalten. Beim strukturellen Design beischreibt man in einer VHDL-Datei ein Teil des Systems, und baut somit das ganze System auf. Komponenten eines Teils werden mit dem Schlüsselwort *component* gekennzeichnet, und die entsprechenden Ports werden abgebildet. Bei einem Datenfluss-orientiertem Design fokussiert man naturgemäß eher auf Daten. Die beiden Stile sind gut für kleinere Systeme geeignet, wo die internen Verbindungen ohne zu viel Aufwand klar und eindeutig dargestellt werden können. Für gößere und komplexere Systeme ist ein Verhalten-orientiertes Design sinnvoller. Man beschreibt in diesem Fall keine detaillierten Systemteile, sondern die Funktionalität des Systemteils selbst. Als ein Beispiel nehmen wir an, dass wir ein XOR-Gate modellieren möchten. Angenommen wir haben OR-Gates, AND-Gates, und NAND-Gates zur Verfügung. Ein XOR(a,b) wird durch AND(OR(a,b), NAND(a,b)) dargestellt.

In struktureller Art werden OR-, NAND-, AND-Gates als *component* abgebildet, und mit *port map* verbunden, als folgendes:

```
library ieee;
use ieee.std_logic_1164.all ;

entity XOR_Gate is
    port(a_xor, b_xor:  in STD_LOGIC_VECTOR(31 downto 0);
              out_xor:  out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture structure of XOR_Gate is
  component OR_Gate
    port (a_or, b_or:   in STD_LOGIC_VECTOR(31 downto 0);
             out_or:    out STD_LOGIC_VECTOR(31 downto 0));
  end component;

  component NAND_Gate
    port (a_nand, b_nand:   in STD_LOGIC_VECTOR(31 downto 0);
             out_nand:      out STD_LOGIC_VECTOR(31 downto 0));
  end component;

  component AND_Gate
```

```
    port (a_and, b_and:     in STD_LOGIC_VECTOR(31 downto 0);
            out_and:        out STD_LOGIC_VECTOR(31 downto 0));
  end component;

  orgate: OR_Gate port map(a_xor, b_xor, out_or);
  nandgate: NAND_Gate port map(a_xor, b_xor, out_nand);
  andgate: AND_Gate port map(out_or, out_nand, out_xor);
end;
```

Zwar ist die Struktur des NAND-Gates klar, jedoch benötigt man viele Zeilen an Codes um dies zu beschreiben.

Alternativ modelliert man das XOR-Gate Verhalten-orientiert. In diesem Fall ist die Struktur des NAND-Gates weniger relevant, die Funktionalität vom XOR-Gate ist hier kurz und präzise beschrieben:

```
library ieee;
use ieee.std_logic_1164.all ;

entity XOR_Gate is
    port(a_xor, b_xor:  in STD_LOGIC_VECTOR(31 downto 0);
            out_xor:    out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behavior of XOR_Gate is
begin
    out_xor <= a_xor XOR b_xor;
end;
```

### Implementation von unserem System

Bei unserer Implementation haben wir mehrere Design-Stile angenommen. Beispielsweise ist die ALU (alu.vhd) in Verhalten-orientierter Sicht realisiert, und der Multiplexer (mux.vhd) strukturell entworfen.

Die CPU wird grob in zwei Teile geteilt: Speicher, und MIPS-Architektur. Im Datenspeicher und Instruktionsspeicher werden Daten und Befehlssätze gespeichert und geladen. Die MIPS-Architektur kann in zwei Hälfte untergeteilt: Datenpfad und Steuerungseinheit. In der Steuerungseinheit (controller.vhd) werden Instruktionen gelesen und weiter bearbeitet, und der Datenfluss wird in datapath.vhd beschrieben. Die Struktur der Implementation ist in Abbildung 2 gezeigt, wobei die Komponenten einzelner Teile mit Pfeilen gekennzeichnet sind.

Der detaillierte Struktur und Port-Verbindungen im System sieht man in Abbildung 1.

Die komplette Implementation findet man im offenen Git-Repository (Schwarzwald, Ostendorf, und Wang 2020).

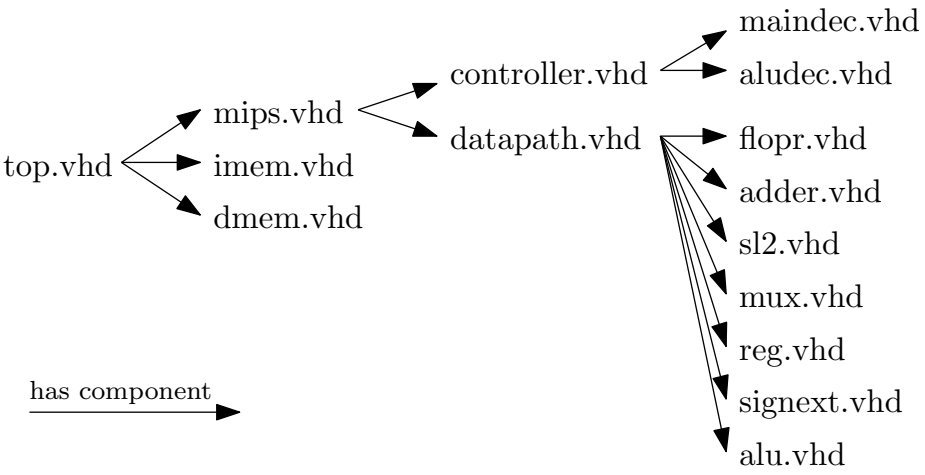


Abbildung 2: Struktur unserer Implementation

# Die Instruction Set Architecture

Zu Beginn des Projekts haben wir uns für eine herkömmliche Architektur mit einem reduced instruction set entschieden. Dabei kam der Vorschlag auf, sich an dem RISC-V Instruction Set zu orientieren.

RISC-V ist eine offene Instruction Set Architektur, deren Entwicklung in 2010 an der Berkeley Universität begann. Im zugrunde liegenden Handbuch (Waterman u. a. 2016) werden viele verschiedene Versionen und Erweiterungen des Befehlssatzes spezifiziert. Beispielsweise werden verschiedene Wortbreiten von 32 über 64 bis 128 Bit spezifiziert, und es gibt Extensions für Multiplikationen, Floating Point Operations und vieles mehr. Es gibt Extensions für Multiplikationen, Floating Point Operations und vieles mehr.

Wir beschränken uns jedoch auf eine Wortbreite von 32 Bit und ein minimales Subset von Instruktionen. Der Grundaufbau der CPU bleibt allerdings erhalten. Das Maschinenmodell der RISC-V Architektur liegt einer Registermaschine mit 32 Registern zugrunde. Davon lassen sich 31 beliebig nutzen, ein spezielles Register `x0` liefert immer den Wert 0 wenn es gelesen wird. Instruktionen werden im 3-Register-Stil definiert. So muss beispielsweise keine Instruktion zum Kopieren von einem Register in ein anderes spezifiziert werden, da das Addieren von `x0` auf einen Wert und das Speichern in einem anderen Register schon den selben Effekt hat.

## Base Instruction Format

Die Instruktionen in der RISC-V Architektur sind alle ähnlich codiert. Die untersten 7 Bit stehen für den Opcode. Anhand dessen lässt sich der Instruktionstyp herleiten. Die Codierung der vier Typen R/I/S/U ist in Abbildung 3 gezeigt. Bei

der Wahl der Codierung ist auf eine einfache Implementation in der Hardware geachtet. Die Felder für Register Argumente `rs1` und `rs2` und das Zielregister `rd` sind in allen Typen an der selben Position, sofern sie vorhanden sind.

Auch das höchstwertigste Bit des Immediate Arguments steht immer an der selben Stelle in der Instruktion. So kann einfach eine Vorzeichenerweiterung des Wertes vorgenommen werden. Die I und S Instruktionstypen setzen immer die unteren 12 Bit. Zusammen mit einer U Instruktion, die nur die oberen 20 Bits setzt kann so innerhalb von nur zwei Instruktionen ein kompletter 32 Bit Immediate Wert geladen werden.

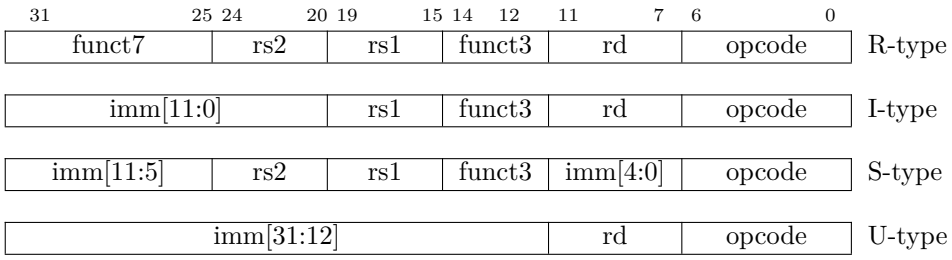


Abbildung 3: RISC-V Unprivileged ISA: Base Instruction Formats

Zusätzlich gibt es noch B und J Instruktionstypen, die ähnlich zu den S und U Typen sind, aber für Branches beziehungsweise Jumps gedacht sind, und desshalb das 0te Bit des Immediate weglassen. Dies wird für Jumps und Branches nicht benötigt, da die kleinsten Instruktionen in der Compact Instruction Extension 16 Bit aligned sind. Da der Speicher byteweise adressiert wird, kann hier ein Bit eingespart werden. Das Layout der B und J Instruktionen ist in Abbildung 4 gezeigt.

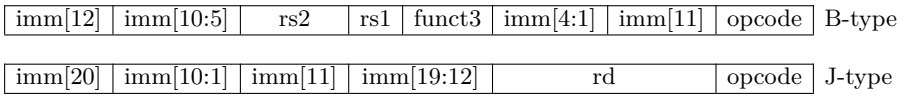


Abbildung 4: RISC-V base instruction formats showing immediate variants.

## Opcode Map

Das RISC-V Instruction Set Manual definiert ein Mapping von verschiedenen Opcodes zu Instruktionen. Dabei ist ein Opcode immer sieben Bit breit. Opcodes, deren erste zwei Bits nicht beide auf eins gesetzt sind werden für kompakte Instruktionen mit nur 16 Bits reserviert. Opcodes, bei denen die ersten fünf Bit gesetzt sind werden hingegen für Instruktionen mit mehr als 32 Bit reserviert.

Die Tabelle 1 ist dem RISCV ISA Manual entnommen, und zeigt die für uns

| inst[4:2] / inst[6:5] | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 ($> 32b$) |
|---|---|---|---|---|---|---|---|---|
| 00 | LOAD | LOAD-F | *custom-0* | MISC-MEM | OP-IMM | AUIPC | OP-IMM-32 | $48b$ |
| 01 | STORE | STORE-F | *custom-1* | AMO | OP | LUI | OP-32 | $64b$ |
| 10 | MADD | MSUB | NMSUB | NMADD | OP-FP | *reserved* | *custom-2* | $48b$ |
| 11 | BRANCH | JALR | *reserved* | JAL | SYSTEM | *reserved* | *custom-3* | $\geq 80b$ |

Tabelle 1: RISC-V base opcode map, inst[1:0]=11

relevanten Opcodes tabellarisch auf. Die dunkelgrau markierten Zellen enthalten für unsere Implementation nicht relevante Instruktionsklassen.

Übrig bleiben die bekannten LOAD und STORE Instruktionen, eine Klasse von BRANCH Instruktionen mit verschiedenen Vergleichen wie BEQ, BNEQ, den Jump-and-Link (-Register) Instruktionen JAL und JALR, die Instruktionen LUI und AUIPC, die die oberen Immediate Bits enthalten, sowie die normalen Operationsklassen OP und OP-IMM für Additionen und ähnliches.

Wie die Instruktionen und gegebenenfalls Immediates genau kodiert sind, ist in Tabelle 2 dargestellt. Sie ist dem RISC-V Instruction Set Manual entnommen und nach der Opcode Map gruppiert. Die in der Opcode Map ausgegrauten Felder sind der Kürze halber in Tabelle 2 ausgelassen. Dazu gehören insbesondere die AMO Instruktionen, gewisse Wortbreiten bei LOAD und Store, sowie FENCE sowie die ECALL und EBREAK Instruktionen.

**RV32I Base Instruction (Sub-) Set**

Upper Immediate / Jumps

| | | | | | |
|---|---|---|---|---|---|
| imm[31:12] | | | rd | 0110111 | LUI |
| imm[31:12] | | | rd | 0010111 | AUIPC |
| imm[20\|10:1\|11\|19:12] | | | rd | 1101111 | JAL |
| imm[11:0] | rs1 | 000 | rd | 1100111 | JALR |

Branches

| | | | | | | |
|---|---|---|---|---|---|---|
| imm[12\|10:5] | rs2 | rs1 | 000 | imm[4:1\|11] | 1100011 | BEQ |
| imm[12\|10:5] | rs2 | rs1 | 001 | imm[4:1\|11] | 1100011 | BNE |
| imm[12\|10:5] | rs2 | rs1 | 100 | imm[4:1\|11] | 1100011 | BLT |
| imm[12\|10:5] | rs2 | rs1 | 101 | imm[4:1\|11] | 1100011 | BGE |
| imm[12\|10:5] | rs2 | rs1 | 110 | imm[4:1\|11] | 1100011 | BLTU |
| imm[12\|10:5] | rs2 | rs1 | 111 | imm[4:1\|11] | 1100011 | BGEU |

Load / Store

| | | | | | |
|---|---|---|---|---|---|
| imm[11:0] | | rs1 | 000 | rd | 0000011 | LB |
| imm[11:0] | | rs1 | 010 | rd | 0000011 | LW |
| imm[11:0] | | rs1 | 100 | rd | 0000011 | LBU |
| imm[11:5] | rs2 | rs1 | 000 | imm[4:0] | 0100011 | SB |
| imm[11:5] | rs2 | rs1 | 010 | imm[4:0] | 0100011 | SW |

OP-IMM

| | | | | | |
|---|---|---|---|---|---|
| imm[11:0] | | rs1 | 000 | rd | 0010011 | ADDI |
| imm[11:0] | | rs1 | 010 | rd | 0010011 | SLTI |
| imm[11:0] | | rs1 | 011 | rd | 0010011 | SLTIU |
| imm[11:0] | | rs1 | 100 | rd | 0010011 | XORI |
| imm[11:0] | | rs1 | 110 | rd | 0010011 | ORI |
| imm[11:0] | | rs1 | 111 | rd | 0010011 | ANDI |
| 0000000 | shamt | rs1 | 001 | rd | 0010011 | SLLI |
| 0000000 | shamt | rs1 | 101 | rd | 0010011 | SRLI |
| 0100000 | shamt | rs1 | 101 | rd | 0010011 | SRAI |

OP

| | | | | | | |
|---|---|---|---|---|---|---|
| 0000000 | rs2 | rs1 | 000 | rd | 0110011 | ADD |
| 0100000 | rs2 | rs1 | 000 | rd | 0110011 | SUB |
| 0000000 | rs2 | rs1 | 001 | rd | 0110011 | SLL |
| 0000000 | rs2 | rs1 | 010 | rd | 0110011 | SLT |
| 0000000 | rs2 | rs1 | 011 | rd | 0110011 | SLTU |
| 0000000 | rs2 | rs1 | 100 | rd | 0110011 | XOR |
| 0000000 | rs2 | rs1 | 101 | rd | 0110011 | SRL |
| 0100000 | rs2 | rs1 | 101 | rd | 0110011 | SRA |
| 0000000 | rs2 | rs1 | 110 | rd | 0110011 | OR |
| 0000000 | rs2 | rs1 | 111 | rd | 0110011 | AND |

Tabelle 2: Reduced Instruction listing for RISC-V

# Software Simulation

Zum Simulieren unseres Computers wird die Programmiersprache Haskell (Marlow und others 2010) verwendet. Haskell ist eine pure, stark getypte, funktionale Programmiersprache. Das bedeutet insbesondere, dass es in Haskell typischerweise nur Funktionen im mathematischen Sinne gibt. Wenn eine Funktion also den gleichen Input zweimal erhält, so produziert sie auch den selben Output.

Die Simulation eines Computers ist eine inherent zustandsbehaftete Aufgabe. Wie die Problematik von Seiteneffekten wie dem Ändern eines Zustandes in einer pur funktionalen Sprache gelöst wird ist gut untersucht und wird beispielweise in den Papern `Imperative functional Programming` (Jones und Wadler 1993) und `State in Haskell` (Launchbury und Jones 1995) anschaulich behandelt.

Ein kurzer Überblick über die Grundlagen von Haskell sowie die Implementation von simpler Zustandsbehandlung wird in diesem Kapitel dennoch gegeben, um die Implementation des Emulators zumindest oberflächlich nachvollziehen zu können.

## Haskell Grundlagen

Ein Haskell Programm besteht aus mehreren Deklarationen. Eine Deklaration definiert einen Wert wie eine Funktion, oder eine Typklasse. Jedes ausführbare Haskell Programm enthält die `main` Deklaration Funktion, die beim Starten des Programms ausgeführt wird.

Ein Haskell Programm, das `42!` berechnet:

```haskell
dieAntwortAufAlles :: Int
dieAntwortAufAlles = 42

fac :: (Num a) => a -> a
fac 0 = 1
fac n = n * fac (n-1)

main = print (fac dieAntwortAufAlles)
```

Ein Typ wird in Haskell immer groß geschrieben. Die Fakultätsfunktion wurde hier mit Pattern-Matching auf dem ersten Argument implementiert. Wird `fac 0` ausgeführt, wird `1` zurückgegeben. Ansonsten wird `n * fac (n-1)` berechnet. Zum Aufrufen einer Funktion werden außerdem keine Klammern benötigt. Der Ausdruck `f g a` wird nach links assoziiert. Es wird also `(f g) a` berechnet. Da im Beispiel oben aber nicht die `fac` Funktion selbst, sondern das Ergebnis geprinted werden soll, werden dort explizit Klammern gesetzt.

Jeder Ausdruck in Haskell hat einen Typ. Dieser kann wie bei `fac` explizit angegeben sein. Eine solche Typdeklaration ist aber nicht notwendig, wenn der Haskell Compiler den Typen wie in bei `main` selbst herausfinden kann. `print` ist eine Funktion mit dem Typ `(Show a) => a -> IO ()`. Die Funktion nimmt

einen beliebigen Typen `a`, der die `Show` Klasse implementiert, und gibt ein `IO ()` zurück. `main` hat also den Typen `IO ()`.

### Effekte

Um in Haskell trotzdem nützliche Programme schreiben zu können, ist es hilfreich, Seiteneffekte als pure Funktion darstellen zu können. Es gibt viele Arten von Seiteneffekten. Beispielsweise kann eine Prozedur fehlschlagen, ein Zustand verändert oder ein Log geschrieben werden. Viele dieser Seiteneffekte können wieder in eine pure Funktion umgewandelt werden. Betrachten wir die Funktion `head`, die das erste Element einer Liste auswählt:

```
# die Definiton einer verketteten Liste.
# Eine Liste ist entweder Nil oder ein Element und eine Restliste
data List a = Nil | Cons a (List a)

head :: List a -> a
head (Cons a _) = a

# ein Datentyp, der Fehler oder das Nichtvorhandensein modelliert.
# ein 'Maybe Int' kann z.B. ein 'Just 1' sein,
#wenn eine Zahl vorhanden ist,  oder 'Nothing', wenn nicht.
data Maybe a = Just a | Nothing

safeHead :: List a -> Maybe a
safeHead Nil = Nothing
safeHead (Cons a _) = a
```

Die normale Definition von `head` schlägt fehl, wenn die Liste leer ist. Da Haskell im Allgemeinen nicht prüfen kann, ob alle Fälle durch Pattern Matching abgedeckt sind, bricht das Programm in diesem Fall ab. Man kann durch das Einführen eines `Maybe` Typs, der die Abwesenheit eines sinnvollen Wertes modelliert, diesen Fehlerfall explizit behandeln.

Ähnlich kann eine Prozedur, die Seiteneffekte zum Lesen und Schreiben eines Zustandes benutzt, in eine pure Funktion umgewandelt werden, bei welcher der Zustand explizit übergeben wird. Eine Prozedur `p :: a --> b`, die solche Art von Seiteneffekten nutzt, kann als pure Funktion `f :: (a,s) -> (b,s)` aufgefasst werden, bei der `s` der Typ des Zustands ist. Durch Currying erhält man `f' :: a -> s -> (b,s)`. Mit dem Typalias `type State s v = s -> (v,s)` lässt sich die Funktion schreiben als `f :: a -> State s b`. Diese Art von Seiteneffekt lässt sich also als Funktion nach `State s b` ausdrücken.

Ein `State s b` lässt sich selbst also als Effekt auffassen, den man mit einem initialen Zustand ausführen kann.

Allgemeine Ein- und Ausgabe wie das Lesen vom Dateisystem wird in Haskell auch mit einem speziellen Typ implementiert. Die Typsignaturen von typischen

I/O Effekten sehen so `getChar :: IO Char` oder so `putChar :: Char -> IO ()` aus.

## Monaden

Nun besteht ein Programm aber typischerweise aus mehreren Effekten. Aus einem `State s a` wird ja gelesen und geschrieben. Dafür müssen Effekte wie `f :: a -> Maybe b` und `g :: b -> Maybe c` hintereinander ausgeführt werden können. Die normale Funktionskomposition kann an dieser Stelle aber nicht benutzt werden, da `Maybe b` und `b` nicht der selbe Typ sind.

Wenn nun über den konkreten Effekt, sei es nun `Maybe`, `State`, oder `IO` abstrahiert wird, dann ist eine Prozedur `p :: a --> b` unter dem Effekt `M` eine pure Funktion `f :: a -> M b`. Ein Monad ist ein Effekt `M`, der die Komposition unterstützt. In der Haskell Standardbibliothek wird dies durch die Monad Typklasse realisiert.

```haskell
{- A type @f@ is a Functor if it provides a function @fmap@ which,
given any types @a@ and @b@ lets you apply any function from @(a -> b)@
to turn @f a@ into an @f b@, preserving the structure of @f@.-}
class Functor f where
    fmap :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
    -- | lifts a value
    pure :: a -> f a
    -- | sequential application
    (<*>) :: f (a -> b) -> f a -> f b

class Applicative m => Monad m where
    return = pure
    -- | Sequentially compose two actions,
    -- passing any value produced by the first
    -- as an argument to the second
    (>>=) :: m a -> (a -> m b) -> m b

    -- composition of actions
    (>=>) :: (a -> M b) -> (b -> M c) -> (a -> M c)
    f >=> g = \x -> f x >>= g

{- Monad Implementation für Maybe -}
instance Monad Maybe where
    return = Just

    Just x >>= f = f x
    Nothing >>= f = Nothing

{- Monad Implementation für State -}
newtype State s a = State { runState :: s -> (a,s) }
```

```haskell
instance Monad (State s) where
    return a = State $ \s -> (a, s)
    State x >>= f = State $ \s ->
            let (v, s') = x s
            in runState (f v) s'
```

Monaden sind in Haskell besonders nützlich, da sie nicht nur die sequenzielle Ausführung von Effekten modellieren, sondern auch eine eigene Syntax haben, die das Schreiben von imperativ aussehendem Code ermöglicht. Ein einfaches Beispiel ist das interaktive Grüßen des Nutzers auf der Console. Die IO Aktionen `putStr :: String -> IO ()` und `getLine :: IO String` tun genau das, was man vom Namen her erwarten würde. Zusammen mit der `do`-Notation ist das Programm `greet` einfach zu schreiben.

```haskell
greet :: IO ()
greet = do
    putStr "What is your first name? "
    firstName <- getLine
    putStr "and your last name? "
    lastName <- getLine
    let name = firstName ++ " " ++ lastName
    putStr ("Welcome, " ++ name)
```

Natürlich lassen sich nicht nur IO Aktionen auf diese Weise benutzen. Jeder Effekt, der die Monad Klasse implementiert, kann von der `do`-Notation Gebrauch machen.

## Monad Transformer

Mit Monaden ist es sehr einfach, Effekte des selben Typs mithilfe von `>>=` oder der `do`-Notation zu nutzen. Umständlich wird es hingegen, wenn man verschiedene Effekttypen nutzen möchte. Um verschiedene Monaden zusammenzusetzen werden daher Monad Transformer genutzt. So kann zu einem beliebigen Grundmonad `m` ein Zustandseffekt hinzugefügt werden, wenn der `StateT` Monad Transformer genutzt wird. `StateT` ist prinzipiell der selbe Typ wie der oben definierte State, mit dem Unterschied, dass das Ergebnis in einem beliebigen Monad `m` zurückgegeben werden kann. Die Operationen `put` und `get`, die häufig in der `do`-Notation genutzt werden sind nun in der `MonadState` Klasse gekapselt und können in allen Monaden genutzt werden, die einen State Transformer nutzen.

```haskell
newtype StateT s m a = StateT {runStateT :: s -> m (a,s) }

instance (Monad m) => Monad (StateT s m) where
    return a = StateT $ \s -> return (a,s)
    (StateT x) >>= f = StateT $ \s -> do
        (v,s') <- x s
```

```
        runState (f v) s'

class (Monad m ) => MonadState s m | m -> s where
    put :: s -> m ()
    get :: m s

instance (Monad m) => MonadState s (StateT s m) where
    put s = StateT $ \_ -> ((),s)
    get   = StateT $ \s -> (s ,s)
    -- many more instances omitted
```

## Die Emulator Monad Implementation

Nach der langen Einleitung kann nun endlich gesagt werden: Alles was man braucht, um einen Computer zu simulieren ist ein aufgepepptes State Monad. Ein Computer enthält mehrere Arten von Speicher. Dazu gehört der Ram, die Register, der Programm Counter oder ähnliches.

Fangen wir mit dem Hauptspeicher an. Gäbe es nur eine Speicherzelle, könnte man diese mit einem normalen State Monad modellieren, und mit den schon definierten `put` und `get` Funktionen manipulieren. Der Speicher hat allerdings mehrere Zellen, die alle adressierbar sind. Wünschenswert wäre folgende Schnittstelle:

```
type Address = Integer
type Word = Integer

class (Monad m) => MonadRam m where
    load  :: Address         -> m Word
    store :: Address -> Word -> m ()
```

Dieselbe Schnittstelle kann man dann für Register und Program Counter replizieren, und die Effekte `loadPc` und `loadReg` nennen. Da aber Register, PC und RAM alle dieselbe Art von Daten speichern, kann hier die Schnittstelle vereinheitlicht werden, indem man einen allgemeineren Adresstyp definiert.

```
-- verallgemeinerte Adresse
data MemRef = Pc | Reg Int5 | Ram Int32

class (Monad m) => MonadEmulator where
    load  :: MemRef         -> m Word
    store :: MemRef -> Word -> m ()
```

Konkret implementiert wird diese Klasse mithilfe des `ST` Monads, welches mehrere Referenzen und Arrays unterstützt und dasselbe Interface wie IO Referenzen benutzt, sowie dem `Reader` Monad, um die Referenzen zu speichern.

```
-- Emulator Memory
data Memory s = Memory
    { pc :: STRef s Word32
```

```
    , reg :: STArray s Word5 Word32
    , ram :: STArray s Word32 Word32 }

loadMem :: Memory s -> MemRef -> ST s Word32
loadMem Memory{pc}   Pc     = readSTRef pc
loadMem Memory{reg} (Reg x) = readArray reg x
loadMem Memory{ram} (Ram x) = readArray ram x

storeMem :: Memory s -> MemRef -> Word32 -> ST s ()
storeMem Memory{pc}   Pc     v = writeSTRef pc v
storeMem Memory{reg} (Reg 0) v = pure ()
storeMem Memory{reg} (Reg x) v = writeArray reg x v
storeMem Memory{ram} (Ram x) v = writeArray ram x v

newtype EmulatorT s m a = EmulatorT
    { runEmulatorT :: ReaderT (Memory s) (ExceptT [Error] m) a
    } deriving ({-...-})

newtype STEmulator s a
    = STEmulator { runSTEmulator :: EmulatorT s (ST s) a }
    deriving ({-...-})

newtype IOEmulator a
    = IOEmulator { runIOEmulator :: EmulatorT RealWorld IO a }
    deriving ({-...-})
```

Eine simple Implementation von einem fetch-decode-execute Kreislauf sieht in diesem Paradigma folgendermaßen aus:

```
type Decode inst m = Word32 -> m inst
type Execute inst m b = inst -> m b

cycle :: MonadEmulator m
    => Decode  i m
    -> Execute i m b
    -> m b
cycle decode execute = do
    pc <- load Pc
    bits <- load (Ram pc)
    store Pc (pc + 4)
    instr <- decode bits
    execute instr
```

Die Implementation des `MonadEmulators` nutzt die `ask` Funktion des ReaderT, um an die Referenzen zum Speicher zu kommen und die `load-` sowie `storeMem` Funktionen zum Wählen der konkreten Referenz. Die letzte Funktion baut aus Aktionen zum Dekodieren und Ausführen einer Instruktion eine Emulator Aktion,

die einen einzelnen Rechenzyklus durchläuft.

Noch sind im Emulator keine unaligned Zugriffe implementiert. Für Instruktionen ist dies allerdings nicht so schlimm, da wir keine komprimierten Instruktionen implementieren, und so der Programm Counter immer Word-aligned ist. Eine LOAD oder STORE Instruktion kann allerdings nur auf Adressen zugreifen, bei denen die letzten zwei Bits 0 sind. Ein unaligned Zugriff führt zu einer CPU Exception.

## Execute that Instruction!

Für einen funktionierenden Emulator fehlt nun nur noch ein Instruktionsdatentyp `Instr` und Funktionen `decode :: Word32 -> Inst` und `execute :: MonadEmulator m => Inst -> m ()`. Hierbei bedienen wir uns der `risc-isa` Haskell Bibliothek („Haskell representation of the RISC-V instruction set architecture“ 2016), die einen `Instr` Typ und entsprechende `encode` und `decode` Funktionen bereitstellt. Die Bibliothek unterscheidet beispielsweise in `BranchInstr`, `JumpInstr`, `MemoryInstr`, `RRInstr` und `RIInstr`. Für all diese Typen wird eine Instanz der `Executable` Typklasse bereitgestellt.

```
class Executable a where
    execute ::
        ( MonadEmulator m
        , MonadError [Error] m
        ) => a -> m ()

instance Executable RegisterRegisterInstr where
    execute (RInstr func rs2 rs1 rd) = do
        a1 <- loadReg rs1
        a2 <- loadReg rs2
        op <- f func
        storeReg rd (op a1 a2)
        where f ADD = pure (+)
            {- ... -}

instance Executable RegisterImmediateInstr where
    execute (IInstr func imm rs1 rd) = do
        a1 <- loadReg rs1
        op <- f func
        let a2 = ext imm
        storeReg rd (op a1 a2)
        where f ADDI = pure (+)
            {- ... -}

instance Executable BranchInstr where
    execute (Branch offset cond rs2 rs1) = do
        a1 <- loadReg rs1
```

```
        a2 <- loadReg rs2
        op <- f cond
        if op a1 a2
        then do
            pc <- load Pc
            store Pc (pc + branchExt offset - 4)
        else return ()
        where f BEQ = pure (==)
            {- ... -}

instance Executable JumpInstr where
    exectue (JAL off rd) = do
        pc <- load Pc
        storeReg rd pc
        store Pc (pc + jumpExt off - 4)
    {- ... -}

instance Executable MemoryInstr where
    execute (Store Word offset rsrsc base) = do
        addr <- loadReg base
        val <- loadReg rsrc
        store (Ram $ addr + ext offset) val
    {- ... -}
```

Mithilfe dieser `execute` Definitionen und der `decode` Funktion aus `riscv-isa` lässt sich eine Single-Step Semantik für die Maschine bauen:

`singleStep = cycle decode execute`.

## Emulator Initialization

Zum Initialisieren des Speichers wird ein Dateiformat bereitgestellt, welches ermöglicht, den Programm Counter zu setzen, und Hexwerte oder Instruktionen in Haskell Syntax an bestimmte Stellen im RAM zu legen. Das Format sieht in etwa so aus:

```
4
0:  0x00000000
4:  0x00a00093
8:  0x00100293
12: RRInstr (RInstr ADD X10 X0 X1)
16: 0xfe4078e3
```

Eine elegantere Lösung wäre ein direktes Parsen von einer ELF (TIS Committee 2001) Datei. Diese brächte allerdings zusätzliche Komplikationen wie Sektionen und Labels mit sich, welche von dem bisherigem Emulator Framework nicht unterstützt werden.

Assembly Code kann dennoch recht einfach in das oben beschriebene Dateiformat überführt werden, die `as` und `objdump` Werkzeuge verwendet. Dies wird anhand des Beispielprogramms `sum.as` demonstriert, welches die Summe von 0 bis n aus Register 4 berechnet.

```
start:
fence
addi    x1,x0,8     # n = 0

init:
mv      x4,x1       # get n
li      x5,0        # sum = 0

loop:
bgeu    x0,x4,start # if n == 0 goto start
add     x5,x5,x4    # sum = sum + n
addi    x4,x4,-1    # n--
j       loop
```

Das Mapping von Adressen auf Werte erhält man folgendermaßen:

```
riscv64-linux-gnu-as --march=rv32i sum.as
riscv64-linux-gnu-objdump -D a.out

a.out:     file format elf32-littleriscv


Disassembly of section .text:

00000000 <start>:
   0:   0ff0000f            fence
   4:   00800093            li  ra,8

00000008 <init>:
   8:   00008213            mv  tp,ra
   c:   00000293            li  t0,0

00000010 <loop>:
  10:   fe4078e3            bgeu    zero,tp,0 <start>
  14:   004282b3            add     t0,t0,tp
  18:   fff20213            addi    tp,tp,-1
  1c:   ff5ff06f            j       10 <loop>
```

## Emulator Terminal User Interface

Mithilfe der Haskell `Brick` Bibliothek ist eine TUI implementiert, um den Zustand der CPU und des RAMs einsehen zu können. Eine Abwandlung des

oben gezeigten Programms sieht in diesem User Interface folgendermaßen aus:

```
0 :0  |         0 :8  |         0 :16 |         0 :24 |
0 :1  |         0 :9  |         0 :17 |         0 :25 |
0 :2  |         0 :10 |         0 :18 |         0 :26 |
0 :3  |         0 :11 |         0 :19 |         0 :27 |
0 :4  |         0 :12 |         0 :20 |         0 :28 |
0 :5  |         0 :13 |         0 :21 |         0 :29 |
0 :6  |         0 :14 |         0 :22 |         0 :30 |
0 :7  |         0 :15 |         0 :23 |         0 :31 |
4 :PC

                                                      0 :0
       RIInstr (IInstr ADDI (Word12 10) X0 X1)    10485907 :4
        RIInstr (IInstr ADDI (Word12 0) X1 X4)       33299 :8
        RIInstr (IInstr ADDI (Word12 1) X0 X5)     1049235 :12
BranchInstr (Branch (Word12 4088) BGEU X4 X0)  4265638115 :16
              RRInstr (RInstr ADD X4 X5 X5)     4358835 :20
    RIInstr (IInstr ADDI (Word12 4095) X4 X4)  4294050323 :24
          JumpInstr (JAL (Word20 1048570) X0)  4284477551 :28
                                                      0 :32
                                                      0 :36
                                                      0 :40
```

Die Werte und Adressen werden noch jeweils in Dezimalzahlen kodiert angegeben. Falls ein Wort im RAM eine valide Instruktion enthält, wird diese zusätzlich zum Wert angezeigt. Durch das Drücken der `n`-Taste kann ein einzelner Schritt des Emulators ausgeführt werden. Die Instruktion, auf die der Program Counter im Bild oben zeigt wird ausgeführt. Daraufhin wird der neue Zustand angezeigt, wobei der Program Counter wieder auf die nächste Instruktion zeigt, die ausgeführt wird.

# Future Work und Abschluss

## Future Work

Ein kompletter RISC-V Chip macht natürlich noch viel mehr, als im User-Space Handbuch spezifiziert wird. Zusätzlich zu Quality-of-Live Änderungen wie dem unterstützten von ELF Dateien würde eine komplette Implementation Hardware Interrupts richtig behandeln.

Dazu gehört auch, dass invalide Instruktionen nicht wie aktuell einfach übersprungen werden, sondern dass der Programmierer die Möglichkeit erhält, eine Interrupt Handler Routine zu schreiben, die im Falle eines Fehlers oder externen Interrupts ausgeführt wird.

Auf der anderen Seite hat ein normaler Rechner natürlich einen nützlichen Effekt.

Dies geschieht durch periphäre Geräte, die am Rechner angeschlossen sind. Ein Programm kommuniziert mit solchen Geräten meist indem bestimmte Stellen im Speicher Hardwarefunktionen zugewiesen wird.

Eine solche Memory Map müsste folglicherweise auch vom dem Emulator implementiert werden. Vorstellbar ist zum Beispiel, dass ein externes Gerät als anderer Prozess modelliert wird, der alle Lese- und Schreibzugriffe in dem gemappten Speicherbereich abfängt.

## Zusammenfassung

In diesem Projekt haben wir die grundlegende Struktur eines Mikrorechners kennengelernt, selbst ein Modell für einen solchen entworfen und dieses dann in VHDL implementiert. In diesem Prozess haben wir ein tieferes Verständnis über die MIPS Architektur gewonnen, und eine neue Programmiersprache gelernt. Außerdem haben wir RISC-V Instruction Set Architecture im Detail angeschaut und einen Emulator in Haskell implementiert. Wir haben Erfahrungen mit dem Design und der Implementierung eines Mikroprozessors gesammelt, mit mehreren Versionen von Architektur experimentiert und Wissen über Haskell genutzt.

Wir hoffen, dass dieser Bericht als eine Dokumentation unserer Erfahrungen aus diesem Projekt, sowie als eine Einführung für alle dient, die sich für dieses Gebiet interessieren oder verwandte Projekte annehmen.

# Referenzen

Der komplette Sourcecode ist in öffentlichen Git Repositories(Ostendorf 2020) (Schwarzwald, Ostendorf, und Wang 2020) aufzufinden.

Church, Alonzo. (1941) 1985. *The Calculi of Lambda Conversion. (AM-6)*. Princeton University Press. https://doi.org/10.1515/9781400881932.

Eduvance. o. J. „VHDL Lecture 1 VHDL Basics“. https://www.youtube.com/watch?v=BDq8-QDXmek&t=8s.

„Haskell representation of the RISC-V instruction set architecture“. 2016. https://hackage.haskell.org/package/riscv-isa-0.1.0.0/.

Jones, Simon L. Peyton, und Philip Wadler. 1993. „Imperative functional programming“. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL 93*. ACM Press. https://doi.org/10.1145/158511.158524.

Launchbury, John, und Simon L Peyton Jones. 1995. „State in haskell“. *Lisp and symbolic computation* 8 (4): 293–341.

Marlow, Simon, und others. 2010. „Haskell 2010 language report“. https://haskell.org/definition/haskell2010.pdf.

Mogensen, Torben Æ. 1992. „Efficient self-interpretation in lambda calculus“. *Journal of Functional Programming* 2 (3): 345–64. https://doi.org/10.1017/s0956796800000423.

Neumann, J. von. 1993. „First draft of a report on the EDVAC“. *IEEE Annals of the History of Computing* 15 (4): 27–75. https://doi.org/10.1109/85.238389.

Ostendorf, Benedikt. 2020. „riscvemu“. GitLab. https://gitlab.com/rnxpyke/riscvemu.

Schwarzwald, Jasper, Ostendorf Benedikt, und Anran Wang. 2020. „Implementation in VHDL - Projekt Mikrorechner“. Git. https://git.mafiasi.de/16awang/pj-vhdl-impl.git.

TIS Committee. 2001. „Executable and linkable format (elf)“. *Specification, Unix System Laboratories*.

Waterman, Andrew, Yunsup Lee, David A Patterson, und Krste Asanović. 2016. „The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1“. https://github.com/riscv/riscv-isa-manual.