DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

# WIP: WebAssembly-basierter Intel 8080 Emulator

## Studienarbeit

für die Prüfung zum

Bachelor of Science

des Studienganges Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

**Felix Hirschel, Moritz Gutfleisch und Nico Thomas**

Abgabedatum 1. April 2090

| | |
|---|---|
| Bearbeitungszeitraum | 30 Wochen |
| Matrikelnummer | 4711 |
| Kurs | Tinf19B1, Tinf19B4 |
| Ausbildungsfirma | SAP SE, Siemens AG |
| Gutachter der Studienakademie | Prof. Dr. Kai Becher |

## Erklärung

Wir versichern hiermit, dass wir unsere Studienarbeit mit dem Thema: „WIP: WebAssembly-basierter Intel 8080 Emulator“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Wir versicheren zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

| | |
|---|---|
| Ort Datum | Unterschrift |
| Ort Datum | Unterschrift |
| Ort Datum | Unterschrift |

**Zusammenfassung**

Die vorliegende Studienarbeit befasst sich mit der Emulation des 8-Bit-Mikroprozessors Intel 8080 in Rust. Neben einer Simulation des Verhaltens der Hardware gab es den Anspruch an die Bereitstellung einer Entwicklungsumgebung, mit welcher der Prozessor benutzt werden kann. Im Zuge dessen wurden neben dem Emulator eine Weboberfläche, bestehend aus Code-Editor und Visualisierung des Speichers entwickelt. Die Verbindung zwischen geschriebenem Code und dem Emulator bildet ein zusätzlich entwickelter Assembler, der entsprechenden Bytecode generiert.

Wissenschaftlicher Anspruch dieser Arbeit ist eine erstmalige programmatische Umsetzung des Intel 8080 in Kombination von Rust und WebAssembly. Der Inhalt bezieht sich maßgeblich auf den Entwicklungsprozess und aufgetretene Schwierigkeiten. Neben der Entwicklung ist vor allem die Performanz des entstandenen Emulators interessant, die im Vergleich zu herkömmlichen, Web-basierten Emulatoren ??? ausfällt.

**Abstract**

# Inhaltsverzeichnis

# Tabellenverzeichnis

# Quellcodeverzeichnis

# Abkürzungsverzeichnis

**Wasm** WebAssembly

**SPA** Single-Page-Applikation

**SP** Stack Pointer

**RAM** Arbeitsspeicher

**PC** Program Counter

**EBNF** Erweiterte Backus-Naur-Form

**CRC** Cyclic Redundancy Check

**CPU** Central Processing Unit

**CP/M** Control Program for Microcomputers

**BDOS** Basic Disk Operating System

**API** Application Programming Interface

**ALU** Arithmetic Logic Unit

# 1 Einleitung

Zu sagen Maschinensprache sei allgegenwärtig ist für nicht-Informatiker eine Aussage, mit der vermutlich zuerst nicht viel angefangen werden kann. Schließlich erfolgt der Kontakt mit den meisten elektronischen Geräten im Alltag entweder durch ein Touch Panel oder unterschiedliche Formen von Schaltern und Knöpfen, keine Eingabe von Nullen und Einsen. Wird sich länger mit der Thematik befasst, muss man feststellen, dass letzten Endes alle diese Interfaces nur eine Abstraktion darstellen. Keins steuert unmittelbar das Verhalten des darunterliegenden Systems insofern, dass ein System PC wüsste, was bspw. „drucke Dokument“ bedeutet.

Damit Steuerungsanweisungen für elektronische Systeme verständlich werden, müssen diese in Bits und Bytes vorliegen, zusätzlich müssen sie interpretiert werden. Schließlich handelt es sich um keine natürliche Form der Information. Die Interpretation geschieht seither mithilfe einer Central Processing Unit (CPU). Diese ist dazu in der Lage numerische Daten in Befehle umzusetzen und ein System mittels der Belegung von Pins, an die eventuelle Ein- und Ausgabegeräte angeschlossen sind, zu steuern.

In der heutigen Zeit gibt es nur noch für wenige Entwickler den Bedarf direkt auf die CPU zuzugreifen, höherlevelige Sprachen vereinfachen den Zugriff soweit, dass die Informationsdarstellung dem Menschen in den meisten Fällen verständlich bleibt. Um ein tieferes Verständnis von der grundlegenden Funktionsweise digitaler Systeme zu erhaletn, bietet es sich trotzdem an, einmal mit Assembler, der Sprache am nähesten an Maschinensprache, zu arbeiten.

Durch die Leistung heutiger PCs ist zur Ausführung von Maschinencode (zu bspw. Lernzwecken) keine zusätzliche CPU in Hardware mehr notwendig. Stattdessen lässt sich deren Verhalten, je nach benötigter Leistung des gewünschten Systems, mit modernen Prozessoren simulieren. Die vorliegende Arbeit befasst sich mit eben dieser Simulation eines älteren Mikroprozessors, dem Intel 8080.

Die Wahl fiel deshalb auf den Intel 8080, weil dieser als „Einsteigerprozessor“ bekannt ist, sich also einerseits für Entwickler eignet, die erste Berührpunk-

te mit der Entwicklung auf Systemebene suchen, andererseits für die, die sich nocht nicht in großem Maß mit der Emulation solcher Systeme befasst haben. Der Ruf rührt vor allem daher, dass eine umfangreiche Dokumentation vorhanden ist und es sich um ein 8-Bit-System handelt, das eine überschaubare Menge an Befehlen unterstützt. Die genaue Funktionsweise und Eigenheiten werden in Kapitel 2.3 näher ausgeführt.

Wie später im Kapitel „Verwandte Arbeiten“ (3.2) dargestellt, gibt es auf dem Gebiet der Emulation bereits eine Vielzahl von Arbeiten, umgesetzt in den verschiedensten digitalen Ökosystemen. Die Simulation ist vor allem unter dem Aspekt interessant, einen tieferen Einblick in die Funktionsweise solcher Systeme, die Vorreiter unserer heutigen Generation von PCs sind, zu erhalten. Deshalb soll nicht nur der Prozessor als solcher emuliert werden, es soll auch eine Oberfläche entwickelt werden, die dem Anwender den Systemzustand zeigt und es ermöglicht eigenen Assemblercode zu schreiben, der so nah am Maschinencode wie möglich ist. Ein systematischer Aufbau der kompletten Anwendung ist in Kapitel 4 zu finden. Auf diesem basiert der Schwerpunkt der Arbeit, die Implementierung in Kapitel 5.

# 2 Grundlagen

## 2.1 WebAssembly

"WebAssembly (Wasm) is a safe, portable, low-level code format designed for efficient execution and compact representation"[10]. Im Endeffekt handelt es sich bei Wasm also um eine low-level Bytecode-Sprache, die von Browsern ausgeführt werden kann. Diese Sprache soll ähnlich performant sein, wie die Ausführung naives Maschinen-Codes. Das Paper, in dem Wasm eingeführt wird, berichtet eine 10% Performance-Diskrepanz zwischen Wasm und naivem Assembly[3]. Durch Kompilation nach Wasm ist es möglich Programme auf Seite des Clienten laufen zu lassen, die sonst vom Server ausgeführt werden müssten. Für viele Programmiersprachen gibt es entsprechende Compiler, die es ermöglichen nach Wasm zu übersetzen (bspw. C/C++, Rust, usw.).

In der entsprechenden Sprache muss explizit die Schnittstelle zu JavaScript deklariert werden, um festzulegen welche Funktionalitäten dem Frontend zur Verfügung stehen.

## 2.2 Rust

Rust ist eine moderne, performante, speichersichere Programmiersprache[1]. Rust's Sprachmodell verhindert die meisten Laufzeitfehler schon zur Compilezeit, wodurch die Entwicklung deutlich angenehmer wird als bei vergleichbaren Sprachen (bspw. C/C++). Außerdem eignet sich Rust aufgrund der umfassenden Dokumentation sehr gut für Wasm-Anwendungen[2].

In diesem Abschnitt werden einige grundlegende Sprachkonstrukte/-konzepte erläutert, die nützlich zum Verständnis dieser Arbeit sind. Diese Informationen stammen direkt aus der offiziellen Dokumentation: Dem Rust Book [7] und der Dokumentation der Standard-Library [11].

[1] siehe `https://www.rust-lang.org/` für mehr

[2] siehe `https://rustwasm.github.io/docs/book/`

### 2.2.1 Structs

Klassen der Objektorientierung werden in Rust mithilfe von sog. „Structs“ (Structures) realisiert. Dabei handelt es sich um eigens definierbare Datentypen die über einen Namen mehrere Attribute zusammenfassen. Die Definition eines solchen Structs ist im Folgenden zu sehen. Sie sind Grundlage für die Entwicklung in der vorliegenden Arbeit, die primär auf objektorientierten Ansätzen aufbaut.

```
struct Example {
    name: String,
        valid: bool,
}
let ex = Example { name: String::from("Name"), valid: true};
```

Wie im Beispielcode zu sehen, befinden sich in der Definition des Structs `Example` keine Methoden. Das liegt daran, dass diese nicht an dieser Stelle stehen dürfen. Stattdessen geschieht dies mittels einem eigenen „Implementation Block“. Dieser lässt sich mittels `impl Example { }` definieren. In diesem Block können nach belieben Methoden implementiert werden, die dann vom entsprechenden Struct zur Verfügung gestellt werden.

Bei ihrer Implementierung sind Methoden implizit von privatem Scope, können also nur innerhalb der `.rs`-Datei genutzt werden, in der sie beschrieben sind. Das Schlüsselwort `pub` erlaubt es auch in anderen Dateien entsprechende Methoden zu verwenden. Methoden die innerhalb eines mit `impl` geöffneten Blocks stehen beziehen sich immer auf einen Struct und können mittels dem `&self`-Objekt auf die entsprechende Instanz und deren Felder zugreifen.

### 2.2.2 Traits

Mithilfe von Traits lässt sich in Rust das Verhalten von Structs (oder allgemeiner: Typen) definieren. Die Analogie zu herkömmlichen Sprachen ist das *Interface*. Genau wie diese definieren Structs eine oder mehrere Funktionalitäten, die der Compiler bei einem Objekt erwarten kann. Ihre Definition sieht ebenfalls ähnlich der von Interfaces aus:

```
pub trait Behaviour {
    fn print(&self) -> {
        print_ln!("Not yet implemented!");
    }
}
```

Das Beispiel beschränkt sich auf eine Methode, `print`. Traits können jedoch beliebig viele Methoden definieren. Ein entscheidender Unterschied zu herkömmlichen Interfaces ist, wie hier zu sehen, die Möglichkeit einen Metho-

denkörper anzugeben. So ermöglicht Rust Standardimplementierungen, auf die zurückgegriffen wird, sollte ein Typ die Methode nicht selber implementieren. Wird diese Implementierung weggelassen, also nur der Methodenkopf angegeben, setzt Rusts Compiler voraus, dass alle, den Trait implementierenden Typen, die entsprechenden Methoden implementieren.

Um das Verhalten eines Typen mittels Trait zu beschreiben wird, wie im vorigen Kapitel ein Implementation Block genutzt. Der einzige Unterschied ist, dass im Anfang des Blocks der jeweilige Trait zu stehen hat:

```
impl Behaviour for Example
```

Innerhalb des so geöffneten Blocks ist es dann möglich, bzw. erforderlich, die relevanten Methoden zu implementieren. Sollte lediglich die Standardimplementierung genutzt werden wollen, ist es erlaubt den Block leer zu lassen.

Sobald ein Typ einen Trait definiert, lässt er sich an allen Stellen benutzen, die diesen Trait voraussetzen, abermals ähnlich zu Interfaces. Die Syntax dafür lautet

```
fn use_type_with_trait(item: &impl Behaviour)
```

### 2.2.3 Das `match`-Statement

Das `match`-Statement ist die Rust Alternative zum klassischen `switch-case`-Statement. Bis auf die Syntax funktioniert es sehr ähnlich:

```
match x {
    1 => /* x == 1... */,
    2..=5 => /* x in [2,3,4,5] */,
    ...
    _ => // default case
}
```

Rust garantiert zu Compilezeit, dass die match-Arme alle möglichen Fälle abdecken (nur relevant wenn kein default-Case vorhanden ist).

### 2.2.4 Result und Option

In Rust gibt es keinen `null`, stattdessen existiert der `Option<T>` Typ. Option ist wie folgt definiert:

```
enum Option<T> {
    Some(T),
    None
}
```

`T` ist ein generischer Typ Parameter, durch Angabe eines Typen in den Spitzen Klammern, wird der im Option enthaltende Typ festgelegt: Ein Element vom Typ `Option<i32>` enthält also entweder nichts (None), oder oder eine 32-Bit Integer (Some(i32)). Um an den enthaltenden Wert zu kommen, muss eine Fallunterscheidung durchgeführt werden, bspw. durch ein `match`-Statement. Dies garantiert, dass keine ungewollten null-References möglich sind (wie z. B. ein Methodenaufruf auf `null`).

`Option<T>` wird verwendet, wenn es akzeptabel ist, dass kein Wert vorhanden ist. Andernfalls sollte `Result<T, E>` verwendet werden.

```
enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

Ein Result enthält entweder einen Wert des entsprechenden Typs, oder einen Error mit einer Error-Message des entsprechenden Typs (oft ein String).

### 2.2.5 Ownership und Moving

Die Verwaltung von Variablen und deren Lebensdauer, sowie Variablenzugriffe fallen unter Rusts Konzept von „Ownership“ (Besitz). Es handelt sich um ein System von Regeln, die der Compiler durchsetzt um eine konsistente Garbage Collection zu ermöglichen.

Grundsatz dieses Konzepts sind die sog. „Owner“(Besitzer) von Variablen. In Rust hat jeder Werte zu jeder Zeit (nur) einen Besitzer, eine Variable. Sollte dieser Besitzer außerhalb des Gültigkeitsbereiches liegen, wird der entsprechende Wert aus dem Speicher entfernt. Mit jeder schließenden, geschwungenen Klammer endet ein solches Scope und darin verwendete Variablen werden gelöscht. In dem Moment, in dem eine Variable von außerhalb in einen anderen Gültigkeitsbereich gegeben wird, wird diese in den neuen Bereich „gemoved“. Das hat zur Folge, dass eine Variable, die als Parameter an eine Methode übergeben wird, anschließend nicht mehr gültig ist. Bestimmte Typen (unter anderem `i32` oder `bool`) implementieren den Trait `Copy`, durch den beim Moven einer Variable in ein neues Scope eine Kopie anstelle des eigentlichen Besitzers übergeben wird. Das funktioniert, weil der Speicherbedarf dieser Typen bereits zur Compilezeit bestimmbar ist.

Die Einschränkung des Movens würde offensichtlich auf Dauer zum Problem werden, gäbe es keine Möglichkeit Variablen innerhalb mehrerer Scopes zu verwenden. Eine Möglichkeit wäre natürlich sie davor zu klonen oder jedes Mal als zusätzlichen Rückgabewert zurückzuliefern, was allerdings ebenso unpraktisch ist. Dafür bietet Rust die Möglichkeit zum „Borrowing“ mittels „Referencing“. Das geschieht mittels vorangestelltem Et-Zeichen (&).

So wird anstelle der Variablen (dem eigentlichen Besitzer) lediglich ein Verweis auf diesen übergeben.

Soll nun der Wert einer Variablen, für den ein Gültigkeitsbereich nur eine Referenz besitzt, verändert werden, bietet Rust das „Dereferencing“ an. Dazu wird der Variablen ein Sternchen (*) vorangestellt. Der beispielhafte Quellcode in 2.2.5 soll diese Konzepte veranschaulichen. Dabei existieren zwei Methoden von denen beim Aufrufen der ersten ein „Move“ auftritt, die zweite Methode arbeitet mit einer Referenz. Sollte nach dem Ausführen des Codes ein lesender Zugriff auf die Variable `txt` erfolgen, gäbe es einen Fehler. Im Fall von `num` erhält man den Wert 20.

**Quellcode 2.1** Darstellung einiger Phänomene von Rusts Ownership

```
fn take_ownrshp(var: String) {
}   // var goes out of scope and is dropped

fn inc_val(number: &i32) {
    *number = number + 10;
    // the value of number may be manipulated by dereferencing
}

let txt = String::from("text");
let num = 10;

take_ownrshp(txt); // ownership of txt moves into the method
inc_val(&num); // inc_val receives a reference to the value of num
```

### 2.2.6 Collections

Um mehrere Objekte eines Typens zu speichern wurde in der Arbeit und der später dargestellten Implementierung maßgeblich Datentypen genutzt, die in der Standardbibliothek von Rust enthalten sind. Deshalb wird die Rede von Maps und Vektoren sein:

Vektoren dienen dem Speichern einzelner Werte desselben Typs T. So entsprechen sie der Form `Vec<T>`. Sie liegen auf dem Heap und das entsprechende T muss deshalb zur Compilezeit implizit oder explizit festgelegt sein. Die Länge eines Vektors ist variabel und kann im Verlauf der Programmausführung durch typische Methoden wie `pop` und `push` manipuliert werden. Die Initialisierung eines Vektors kann als leere Menge erfolgen, oder auch mittels Makro `vec![]` und einer kommaseparierten Liste von Elementen. Darüber hinaus ist es möglich einen Vektor zu initialisieren, der aus einer Anzahl von gleichen Werten besteht: `vec![0; 10]` erzeugt einen Vektor, der zehn Einträge besitzt, die alle null sind.

Wenn in der Arbeit von einer Map die Rede ist, bezieht sich das konkret auf den Typ `HashMap<K, V>`. Die Funktionsweise ist analog zu der in anderen bekannten Programmiersprachen: Ein Wert vom beliebigen Typ V wird über einen Wert vom beliebigen Typ K identifiziert. Bei ihrer Verwendung ist in Rust zu beachten, dass auch hier das Konzept der Ownership gilt. Wenn also nicht bewusst Referenzen übergeben werden, werden alle Werte, bzw. ihr Besitz, beim Einfügen in die Map gemoved. Das Lesen von Elementen in einer Map geschieht mittels Schlüssel, wobei das Ergebnis im Fall von Rust `Option<&V>` ist: Wenn der Schlüssel in der Map existiert, antwortet die entsprechende Methode mit einem Verweis auf den Wert, gewrappt in `Some()`, ansonsten ist das Ergebnis vom Typ `None`.

### 2.2.7 Tests

Während das Testen selbst kein expliziter Bestandteil der schriftlichen Ausarbeitung sein wird, war das Schreiben von Tests bei der Entwicklung unerlässlich. Deshalb soll einmal in kurzen Zügen erläutert werden, auf welchen Funktionalitäten von Rust die geschriebenen Tests basieren.

Testen ist in Rust ein fester Bestandteil der Standardbibliothek und einfache Tests sind ohne die Verwendung zusätzlicher Bibliotheken oder Frameworks nötig. Der Übersicht halber bietet es sich für Unittests einer Datei an, ein eigenes Modul zu nutzen. Die Annotation `#[cfg(test)]` teilt Rust mit, wo die jeweiligen Tests zu finden sind. Außerdem kann der Compiler die Unittests dadurch für das spätere Bauen und Deployen (bzw. im Fall der Arbeit Kompilierung nach Wasm) der Anwendung vernachlässigen, da diese dort nicht mehr gebraucht werden.

Innerhalb des Test-Moduls können beliebig Tests definiert werden, wobei sich ihr Aufbau und Definition nicht grundlegend von anderen Sprachen unterscheidet: Die Testmethoden besitzen in der Regel keinen Rückgabewert und werden, sofern es sich um einen auszuführenden Test handelt, mit einer Annotation (`#[test]`) versehen. Mittels verschiedenen Methoden der Standardbibliothek lassen sich dann diverse Assertions (bspw. `assert_eq!()`) durchführen, wobei deren Scope nicht auf Testmethoden beschränkt ist.

## 2.3 Intel 8080

Das Folgende Kapitel ist eine allgemeine Übersicht über die Architektur und Funktionsweise einer Intel 8080 CPU. Die Informationen sind dem offiziellen Datenblatt[6] und dem offiziellen Programmierhandbuch[5] entnommen. Einige Aspekte — z. B. die Taktung des Intel 8080 — welche nicht emuliert werden müssen, werden bewusst nicht erwähnt. Die Übersicht soll ausschließlich Details erläutern, die relevant zum Verständnis des Emulators sind.

#### RAM und Stack

Im Arbeitsspeicher (RAM) liegt sowohl der Programmcode, als auch der Stack. Der RAM des 8080 wird über 16-Bit Addressen angesprochen, hat also maximal 65536 (0x10000) verfügbare Addressen.

Während der Ausführung eines Programmes zeigt der Program Counter (PC) auf die zunächst auszuführende Instruktion im RAM und der Stack Pointer (SP) auf die Spitze des Stacks. Der Stack des 8080 wächst allerdings nach unten (Addressen sinken bei größerem Stack). Es ist nicht festgelegt, wo der Stack anfängt, der Programmierer muss den SP per Programm setzen. Der PC ist initial 0, außer der Entwickler setzt den Startpunkt manuell.

Beim Stack handelt es sich um eine Last In-First Out (LIFO) Datenstruktur im RAM, die 2 zugreifende Operationen unterstützt: Pop und Push. Pop entfernt das oberste Element und lädt es in das angegebene Register, Push legt ein angegebenes Element auf den Stack. Die Operationen in-/dekrementieren automatisch den SP entsprechend.

### 2.3.1 Register

Register sind kleine Speichereinheiten auf dem Prozessorchip. Auf diese kann aufgrund der Nähe zum Prozessor schnell zugegriffen werden. Der 8080 hat 8 solcher Register. 6 dieser Register können über Assembly angesteuert werden. Jedes Register speichert einen 8-Bit Wert, zudem können die Register paarweise angesprochen werden (als ein 16-Bit Wert).

Tabelle 2.1: Intel 8080 Register, benachbarte Register können paarweise angesprochen werden

| | |
|---|---|
| W | Z |
| B | C |
| D | E |
| H | L |

Außerdem gibt es ein weiteres Register, den Akkumulator, welches für arithmetische Operationen verwendet wird. Die Abkürzung PSW (Processor Status Word), die im Bezug auf bestimmte Instruktionen verwendet wird, bezieht sich auf den Akkumulator kombiniert mit den Flags (siehe unten).

### 2.3.2 Flags

Die CPU muss Informationen über die Ergebnisse arithmetischer Operationen speichern (bspw. ob die letzte Operation 0 ergeben hat), dafür gibt sogenannte Flags. Diese werden in Hardware als 5 Flip-Flops[3] realisiert, im

[3] flüchtiger 1-Bit Datenspeicher

Endeffekt handelt es sich einfach um Booleans. Es gibt die folgenden Flags:

**Zero** Letzte Operation hat 0 ergeben

**Carry** Bei der letzten Operation gab es einen Übertrag

**Sign** Das Ergebnis der letzten Operation war negativ

**Parity** Die Anzahl der Einsen im Ergebnis (Basis 2) war gerade

**Auxiliary Carry** Überlauf im unteren Halbbyte[4]

### 2.3.3 Opcodes

Wie jede CPU ist der Intel 8080 in der Lage Maschinensprache auszuführen. Diese besteht aus sogenannten Opcodes (Instruktionen). Dabei handelt es sich um jeweils 1 bis 3 Bytes, wobei das erste zur Identifikation der Instruktion gilt und die anderen Bytes ggf. Parameter enthalten.

### 2.3.4 Assembly

Programme in Maschinensprache sind für Menschen schlecht lesbar, daher ist die typische Abstraktion über der Maschinensprache eines Prozessors die entsprechende Assembly-Syntax. Es folgt eine kurze Einführung in Intel 8080 Assembly.

#### Notation

In den folgenden Sektionen werden einige Instruktionen aufgelistet und erklärt. Für diese Erklärungen wird eine einfache Notation verwendet, um wichtige Konzepte darzustellen. Buchstaben repräsentieren die entsprechenden Register (Buchstabenpaare analog die Registerpaare), eckige Klammern bedeuten, dass das innere Register(paar) als Addresse interpretiert wird und der dortige Wert gemeint ist. Eine kurze Übersicht ist in Tabelle 2.2 auffindbar.

Tabelle 2.2: Notation zur Beschreibung der Assembly-Instruktionen

| | |
|---|---|
| A | Wert im Akkumulator |
| B | Wert in Register B |
| BC | Wert in Registerpaar BC |
| $[HL]$ | Wert an Addresse HL |
| 10X | 10 in Zahlensystem X (H: Hex, D: Dez, O/Q: Oct, B: Bin) |

Zahlenliterale werden durch Suffixe den entsprechenden Zahlensystemen zugewiesen — H für Hexadezimal (Basis 16), D für Dezimal (Basis 10), O/Q für Octal (Basis 8) und B für Binär (Basis 2).

[4] Siehe: `https://de.wikipedia.org/wiki/Hilfs%C3%BCbertragsflag`

### Registerzugriff

Tabelle 2.3: Beispielhafte Befehle zum Registerzugriff

| | |
|---|---|
| `MOV B, A` | Setze B auf Wert in A |
| `MOV M, A` | Setze $[HL]$ auf A |
| `MOV A, M` | Setze A auf $[HL]$ |
| `MVI B, 0FFH` | Setze B auf 0FFH |
| `LXI B, 1234H` | Setze BC auf 1234H |

### Verzweigungen

Bei Assembly gibt es grundlegend 2 Verzweigungstypen: Jumps und Calls. Bei Jumps handelt es sich um direkte Sprünge zu einer Addresse im RAM. Die äquivalente in Programmiersprachen wie C ist der `goto` Befehl.

Calls hingegen entsprechen grob dem Funktionsaufruf aus higher-level Sprachen. Beim Aufruf einer Call-Instruktion wird die momentane Position im Code (der PC) auf dem Stack abgelegt, dies ist die sogenannte Return-Addresse, und anschließend ein Sprung zur angegebenen Addresse (zum Funktions-Code) ausgeführt. Die Return-Instruktion stellt das Ende einer Funktion dar, sie setzt den PC auf das oberste Element des Stacks, bei korrekter Verwendung ist dies die Return-Addresse.

Tabelle 2.4: Beispielhafte Befehle für Verzweigungen

| | |
|---|---|
| `JMP 0FFH` | Setze PC auf 0FFH |
| `JZ 0FFH` | Setze PC auf 0FFH falls Zero-Flag gesetzt ist |
| `CALL 0FFH` | Speichere PC auf dem Stack und setze ihn auf FFH |
| `RET` | Setze PC auf oberstes Stack Element (und entferne es) |

### Arithmetik

Der Intel 8080 implementiert eine 8-Bit Arithmetik. Dafür werden die Bytes als Zahlen im Zweierkomplement interpretiert. Die Ergebnisse der Operationen werden hauptsächlich im Akkumulator gespeichert — Ausnahmen sind z. B. Inkrementierungen der Register. Viele der Operationen bearbeiten auch die Flags, um Informationen über das Ergebnis zurückzugeben (siehe Abschnitt 2.3.2 für Details). Tabelle 2.5 zeigt einige arithmetische Anweisungen mit einer kurzen Erklärung.

### Labels

Einen integralen Bestandteil der Programmierung mit Assembly machen „Label“ aus. Es handelt sich um Sprungmarken, mittels welcher die Reihenfolge der Ausführung der Statements beeinflusst werden kann. Außerdem

Tabelle 2.5: Beispielhafte Befehle für arithmetische Operationen

| | |
|---|---|
| `ADD B` | Addiere B auf Akkumulator |
| `CMP B` | Setze Zero-Flag falls $A = B$, sonst setze Zero-Flag zurück |
| `INR B` | Inkrementiere B um 1 |

enthalten Labels bestimmte Werte, welche für arithmetische Operationen verwendet werden können.

Die Deklaration eines Labels folgt diesem Aufbau: `label: Opcode`, wobei das Label selbst optional ist. Zusätzlich kann ein Label allein in einer einzelnen Zeile stehen, in diesem Fall bezieht es sich auf die nächste vom Assembler angetroffene Instruktion. Dadurch ergibt sich auch die Möglichkeit mehrere Label auf dasselbe Byte zu beziehen, was legal ist. In all diesen Fällen referenziert ein Label den Index des Bytes im Bytecode, vor welchem es deklariert wurde. Die alleinige Deklaration eines Labels hat noch keinen Einfluss auf das vorliegende Programm.

Der Name eines Labels muss mit einem Buchstaben des Alphabets, einem „@“ oder „?“ beginnen, besteht aus einem bis fünf Zeichen und muss mit einem „:“ enden. Nicht erlaubt ist die Verwendung reservierter Namen (beispielsweise Namen der Register) oder mehrfache Verwendung desselben Namens. Sofern der Name eines Labels länger als fünf Zeichen ist, wird er auf die ersten fünf gekürzt.

### Pseudo-Instruktionen

Neben den eigentlichen Instruktionen bietet der Intel 8080 dem Entwickler diverse Befehle, die keinen Bytecode erzeugen, weshalb es sich um sogenannte „Pseudo-Instruktionen“ handelt. Sie spielen bei der Entwicklung auf Assembly-Level eine zentrale Rolle, insofern dass sie es erlauben Konzepte wie If-Verzweigungen oder Variablen umzusetzen.

Eine Pseudo-Instruktion folgt einem ähnlichen Aufbau wie herkömmliche Befehle, manche legen allerdings fest, ob das vorangestellte Namensfeld eines Befehls vorhanden sein muss, beziehungsweise darf. Bis auf einige Ausnahmen können auch hier Labels, Sprungmarken die das nachfolgende Byte referenzieren, definiert werden. Dem Entwickler stehen insgesamt fünf Pseudo-Instruktionen zur Verfügung, deren Aufbau und Funktion nun folgt:

*Origin* erlaubt es die Adresse, in der das nächste Byte assembled wird, festzulegen. Die Syntax dafür lautet `ORG exp`, wobei es sich beim zweiten Teil um eine 16-Bit-Adresse, beziehungsweise deren Repräsentation als mathematischer Ausdruck handelt. Nur Instruktionen, die auf diesen Befehl folgen, sind von ihm betroffen.

*Equate* erlaubt die Definition einer Konstanten mittels `name EQU exp`. Jedes Vorkommnis von „name“ nach dessen Definition wird vom Assembler mit dem entsprechenden Ausdruck ersetzt. Eine so definierte Variable darf nicht erneut definiert werden.

*Set* funktioniert identisch zu Equate (`name SET exp`), erlaubt allerdings das wiederholte Definieren von Variablen.

*End Of Assembly* definiert das (physikalische) Ende des Programms mittels `END`. Dieser Befehl muss vorkommen, allerdings nicht mehr als einmal.

*Conditional Assembly* dient der Programmierung mit Bedingungen und funktioniert identisch zu den bekannten If-Statements. Die Definition beginnt mit `IF exp` beginnt und endet mit `ENDIF`. Der Code, der sich zwischen diesen beiden Zeilen befindet, wird aufgerufen sofern der mathematische Ausdruck, der auf das `IF` folgt, als wahr (ungleich 0) interpretiert wird.

*Macro Definition* ist die umfangreichste Pseudo-Instruktion die der Intel 8080 bietet. Mittels `name MACRO list` und `ENDM` lassen sich Methoden ohne Rückgabewert realisieren. Ein Makro besteht aus einem Namen, einer Menge von Befehlen und optional einer Menge von Ausdrücken, die als Parameter dienen (`list`). Sofern im Assembler nun der definierte Name als Instruktion angetroffen wird, wird an dieser Stelle der Inhalt des Makros' (gegebenenfalls unter Beachtung der Parameter), eingesetzt und in Bytecode übersetzt.

### 2.3.5 Interrupts

Durch sogenannte Interrupts kann die normale Ausführung der CPU unterbrochen werden. Interrupts sind externe Signale (bspw. von Peripheriegeräten), bei denen es sich um eine einzelne Instruktion handelt. Diese Instruktion wird ausgeführt, anschließend wird die normale Ausführung fortgesetzt. Meistens handelt es sich bei dieser Instruktion um eine `RST`-Instruktion, eine Gruppe von Instruktionen, die `CALL`s auf fixe Addressen realisieren. Dadurch kann der Entwickler den entsprechenden Code um die Interrupts zu bearbeiten an diesen Addressen ablegen, sodass er somit ausgeführt werden kann.

Der Entwickler kann entscheiden ob Interrupts möglich sind, mithilfe der `EI`-Anweisung (Enable Interrupts) bzw. `DI`-Anweisung (Disable Interrupts). Diese setzen eine interne Flag, die bestimmt ob Interrupts zugelassen sind. Interrupts während der Wert des Flags `false` ist werden ignoriert. Bei Programmstart ist dies der Fall.

### 2.3.6 Peripherie

Über sogenannte Ports regelt der Intel 8080 die Datenübergabe zwischen dem Chip und den angeschlossenen Geräten. Es gibt 256 verfügbare Ports, über die jeweils ein Byte entweder eingelesen oder ausgegeben werden kann.

Über die `IN` und `OUT` Instruktionen kann dies vom Programmierer gesteuert werden. Ihnen wird als Parameter der gewünschte Port übergeben. `IN` liest dann den Wert des Ports in den Akkumulator, `OUT` schreibt den Wert des Akkumulators in den Port.

So kann bspw. ein Display an den 8080 angeschlossen werden, dem über eine Interrupt-Routine die aktuellen Pixelwerte mitgeteilt werden, wenn das Display ein Interrupt-Signal sendet.

## 2.4 Angular

Angular ist ein quelloffenes TypeScript Framework zur Entwicklung von Single-Page-Applikationen (SPA) im Web. Eine SPA ist eine Webanwendung, die nur aus einer einzigen Webseite besteht und mithilfe von JavaScript Code Navigation ermöglicht und interaktive Bedienelemente anbietet. Nachfolgend werden einige Aspekte von Angular erläutert, die für die Entwicklung der Weboberfläche des Emulators von Bedeutung sind.

### 2.4.1 Components

Einer der Hauptbestandteile von Angular sind die *Components*. Components sind wiederverwendbare Bausteine, aus denen die finale Webanwendung aufgebaut wird. Das kann beispielsweise ein Knopf sein, den der Nutzer drückt, oder ein Eingabefeld. Vergleichbar ist dieses Konzept mit den eingebauten Tags, die HTML anbietet. Allerdings können bei Angular die Tags selbst erstellt werden und mit Attributen und Events ausgestattet werden.

```
@Component({
    selector: 'example',
    templateUrl: './example.component.html'
})
export class ExampleComponent {
    public heading: string = 'Hello World!';

    public onButtonClicked(): void {
        console.log('Button was clicked!');
    }
}
```

In diesem Codeabschnitt wird eine Beispiel-Komponente mithilfe von TypeScript definiert. Eine Komponente besteht aus einer Klasse, die mit der `@Component` Annotation versehen wird. Die Annotation verlangt einige Parameter: Der `selector` bestimmt wie das HTML-Element heißt, durch das die Komponente am Ende auf der Webseite eingefügt werden kann. Die Eigenschaft `templateUrl` gibt den Pfad zu einer HTML-Vorlage an, die be-

stimmt, wie die Komponente schließlich auf der Webseite dargestellt wird. Innerhalb der Klasse wird im Beispiel eine öffentliche Variable mit dem Namen `heading` definiert, die den Wert `'Hello World'` hat. Außerdem wird eine öffentliche Methode mit dem Namen `onButtonClicked()` definiert, die eine Nachricht in der Konsole des Browsers ausgibt.

Das Template der Komponente sieht folgendermaßen aus:

```
<div>
    <h1>{{ heading }}</h1>
    <button (click)="onButtonClicked()">Click me</button>
</div>
```

Dargestellt wird die Komponente als HTML-Block. Innerhalb des Blocks befindet sich eine Überschrift. Im `<h1>`-Tag sieht man nun ein weiteres Feature von Angular. Hier wird eine sogenannte **Text interpolation** definiert, die dafür sorgt, dass der Wert der Variable `heading` auf der Webseite angezeigt wird. Unterhalb der Überschrift wird nun ein `<button>` angezeigt. Auch hier kommt ein Feature von Angular zum Einsatz, ein sogenanntes **Event binding**. An das `click` Event des Knopfes wird die in der Klasse der Komponente definierte Methode `onButtonClicked()` gebunden, die nun bei jedem Klick auf den Knopf aufgerufen wird.

### 2.4.2 Services

Viele Komponenten einer Anwendung greifen auf gemeinsame Daten und Funktionen zu. Für diesen Anwendungsfall stellt Angular *Services* zur Verfügung. Das sind TypeScript Klassen, die mithilfe von Dependency Injection in anderen Services oder auch Komponenten genutzt werden können, um beispielsweise auf ein geteiltes Datenrepository zuzugreifen.

```
@Injectable({
    providedIn: 'root'
})
export class ExampleService {
    private counter: number = 0;

    public increaseCounter(): void {
        this.counter += 1;
    }

    public getCounter(): number {
        return this.counter;
    }
}
```

In diesem Codebeispiel sieht man wie ein Service aufgebaut ist. Er besteht lediglich aus einer TypeScript Klasse, die mit der `@Injectable`-Annotation versehen ist. Die Eigenschaft `providedIn: 'root'` gibt hier an, dass der Service als Singleton zur Verfügung gestellt wird und sich schließlich alle Komponenten, die den Service nutzen, die selbe Instanz teilen.

Um den Service zu nutzen muss eine Komponente diesen einfach nur in dessen Konstruktor anfordern:

```
@Component({
    selector: 'example',
    templateUrl: './example.component.html'
})
export class ExampleComponent {

    private readonly exampleService: ExampleService;

    constructor(exampleService: ExampleService) {
        this.exampleService = exampleService;
    }

    public onButtonClicked(): void {
        this.exampleService.increaseCounter();
    }
}
```

Das Dependency Injection Framework von Angular sorgt nun bei der Erstellung der Komponente dafür, dass der Service korrekt injiziert wird.

# 3 Analyse

## 3.1 Zielstellung

Das Ziel ist es einen Emulator zu entwickeln, welcher die vollständige Intel 8080 Spezifikation[6] unterstützt. Dabei sind die zentralen Aspekte wie folgt:

- Vollständige 8080 Assembly Unterstützung
- Simulierte Schnittstelle zu Ein-/Ausgabegeräten
- Korrekte Behandlung von Hardware-Interrupts

Außerdem soll ein entsprechendes Web-Frontend entwickelt werden, um den Emulator zu bedienen. Dieses soll einen Editor beinhalten, um Assembly Programme zu schreiben, die Ausführung dieser Programme ermöglichen, den Zustand des Emulators während der Ausführung darstellen und Auswahl zwischen verschiedenen Peripherie-Geräten ermöglichen (Pixel-Display, Eingabefeld, o.ä.).

Es soll sowohl möglich sein Schritt für Schritt durch ein Programm zu gehen, als auch das Programm automatisch laufen zu lassen.

## 3.2 Verwandte Arbeiten

Im folgenden sollen einige öffentlich verfügbare Projekte, die sich gleichermaßen mit der Emulation eines Intel 8080 Prozessors befasst haben, betrachtet werden. Sie sind als Beispiele zu verstehen um eine anschließende Abgrenzung dieser Arbeit zu ermöglichen. Weil es sich bei dem gewählten Prozessor um ein sehr altes, simples System handelt, existiert eine Vielzahl von Arbeiten, die im Umfang dieser Arbeit nicht alle berücksichtigt werden können.

### 3.2.1 Intel 8080 CPU Emulator (Stefan Tramm)

Bei dieser Implementierung handelt es sich um eine vollständige Emulation des Intel 8080. Der Emulator unterstützt vier Laufwerke und 13 unterschiedliche I/O-Ports. Das System realisiert ein Control Program for Microcom-

puters (CP/M), ein Betriebssystem, unter anderem für Intel 8080-basierte System entwickelt [4]. Es wird durch ein VT100 Terminal im Browser bedient. Die von den Laufwerken gelesenen Daten werden innerhalb einer lokalen Web-SQL Datenbank gespeichert. Die Emulation erfolgt in nativem JavaScript [14].

### 3.2.2 8080 (superzazu)

Diese Version des Emulators wurde in C99, einem Sprachdialekt von C, entwickelt. Ziele bei der Entwicklung waren Korrektheit (überprüft mittels Test-Roms), Lesbarkeit und Portierbarkeit für andere Maschinen. Der Emulator erlaubt unter anderem die Emulation des Arcade-Klassikers „Space Invaders“ (der eine zusätzliche Visualisierung benötigt)[12, 13].

### 3.2.3 js-8080-sim (Eli Bendersky)

Diese letzte Arbeit kommt der vorliegenden Aufgabe vermutlich am nähesten im Bezug auf das Ziel: Die Emulation des Intel 8080 und eine Darstellung im Webbrowser. Neben dieser Darstellung ist auch eine Ausführung über ein beliebiges Command Line Interface möglich. Das Projekt basiert auf einer JavaScript-Implementierung des Intel 8080 [8], das an einigen Stellen, gemäß Bedarf, angepasst wurde. Neben der Simulation des Maschinencodes mittels Fremdbibliothek wurde ein eigener Assembler und eine Weboberfläche entwickelt, die ebenfalls auf JavaScript basieren [1].

### 3.2.4 Abgrenzung

Wie die herangezogenen Beispiele zeigen und auch bei weiterer Recherche festgestellt wurde, basieren viele der verfügbaren Emulatoren auf JavaScript (vor allem wenn diese im Web verfügbar sein sollten) oder (Sprachdialekten von) C. Daneben existieren vereinzelt Projekte, die auf anderen Sprachen basieren, darunter ist auch Rust vertreten [9].

Ein maßgeblicher Unterschied zur bisherigen Entwicklung in diesem Feld ist die Verwendung von Wasm, die bisher bei keinem Projekt festgestellt werden konnte. Unter Verwendung dieser Schnittstelle sollen die zwei vorherrschenden Aspekte der Performanz und Bedienbarkeit optimal genutzt werden: Ein schneller Emulator, geschrieben in Rust anstelle von JavaScript und eine Weboberfläche, die die Ausführung von Programmcode auf Client-Seite ermöglicht.

## 3.3 Beitrag

Unser Beitrag ist WIP, ein in Rust geschriebener Intel 8080 Emulator mit Web-Frontend. WIP erfüllt die oben vorgestellten Ziele vollständig.

Dadurch, dass unser Emulator nach Web-Assembly kompiliert wird, läuft der Emulator naiv im Browser des Clienten. Durch eine explizit definierte Schnitstelle, kann der Emulator mittels JavaScript-Code bedient werden. Außerdem findet eine Übersetzung von Assembly- zu Bytecode mittels einem eigens definierten Assembler statt, der eine vollständige Unterstützung aller Opcodes und Pseudo-Instruktionen implementiert.

Der Editor unterstützt Syntax-Highlighting sowie automatische Vervollständigung von Code und verfügt über Knöpfe um Kompilation und schrittweise, bzw. automatische, Ausführung des Programmes zu ermöglichen. Es gibt Anzeigen für den Zustand der CPU (Register, Flags, etc.), für den Arbeitsspeicher und für die Peripherie-Geräte.

# 4 Design

## 4.1 Emulator

Der Emulator umfasst alles, was zur Simulation der Intel 8080 CPU beiträgt. Dabei handelt es sich hauptsächlich um das Ausführen von Instruktionen, das Interrupt-Handling sowie die Anbindung von Ein- und Ausgabegeräten.

### 4.1.1 Zentrale Struktur

**Quellcode 4.1** Zentrale Emulator Struktur

```
struct Emulator {
    pc: u16,
    sp: u16,
    ram: RAM,
    reg: RegisterArray,
    input_devices: [InputDevice; 256],
    output_devices: [OutputDevice; 256],
    running: bool,
    interrupts_enabled: bool
}
```

Den Kern des Emulators bildet eine Struktur, welche zuständig für die Ausführung der Maschinencode-Programme ist. Sie gruppiert alle notwendigen Komponenten eines Intel 8080 Systems. Der Aufbau der Struktur ist in Listing 4.1 illustriert.

Diese Komponenten wurden in Kapitel 2 bereits erklärt: `pc` und `sp` sind 2 16-Bit-Zahlen, die den Program Counter (PC) und den Stack Pointer (SP) repräsentieren. `ram` ist der Arbeitsspeicher und `reg` simuliert die Register (inklusive Flags und Akkumulator). Die Ports für I/O-Geräte werden durch 2 Arrays mit jeweils 256 Elementen repräsentiert. Darauf folgen zwei Boolean, einer sagt aus, ob der Emulator am Laufen ist und der andere gibt an, ob Interrupts erlaubt sind.

### 4.1.2 Modularität

Der Intel 8080 ist lediglich die CPU, RAM und I/O-Geräte arbeiten prinzipiell unabhängig. Diese müssen zwar eine entsprechende Schnittstelle bereitstellen um angeschlossen werden zu können, aber können beliebig implementiert sein. Unsere Implementierung ermöglicht verschiedene Implementierungen für RAM und Input/Output-Devices zu haben. Es handelt sich bei diesen Typen jedoch nicht um Interfaces, da Rust diese nicht unterstützt. Wie genau das in Rust umgesetzt ist, wird in Kapitel 5 erläutert. Prinzipiell ist die Funktionsweise identisch zu der klassischer Interfaces, aber ihre Implementierungen sind beliebig.

### 4.1.3 Ausführung

Die `Emulator::execute_next()` Methode führt die Instruktion an der Adresse im PC aus. Der Opcode wird über ein enormes `match`-Statement auf die entsprechende Funktion delegiert, die den Opcode ausführt. Diese Funktionen sind zuständig, den PC entsprechend zu erhöhen. Abhängig vom ausgeführten Opcode muss dieser um 1, 2 oder 3 erhöht werden.

Der Rückgabetyp der Methode ist `Result<(), &str>`, dadurch können entsprechende Fehlermeldungen nach außen propagiert werden. Dies ist wünschenswert, damit auf dem Frontend entsprechende Fehlermeldungen angezeigt werden können, um dem Benutzer den Entwicklungsprozess zu erleichtern.

#### Instruktionen

Um zu großen Dateien vorzubeugen, sind die Implementierungen der Instruktionen aufgeteilt in verschiedene Module. Sie sind logisch gruppiert in Arithmetik, Kontrollfluss, Logik, Speicherzugriff, Verschiebung und Speziell. Obwohl die Funktionen in unterschiedlichen Dateien/Modulen deklariert sind, sind sie Methoden der `Emulator`-Struktur. Die verschiedenen Funktionen werden dann im Code von `Emulator::execute_next()` aufgerufen. Auch diese Funktionen geben häufig `Result`s zurück, sofern die Ausführung in einem Fehler resultieren kann.

### 4.1.4 Interrupts

Über den `interrupts_enabled`-Boolean wird geregelt, ob es erlaubt ist, Interrupts an den Emulator zu senden. Die Folgende Methode der zentralen Struktur wird verwendet, um Interrupts auszulösen:

```
pub fn interrupt(&mut self, opcode: u8) -> EResult<usize> {
    if self.interrupts_enabled {
        self.interrupts_enabled = false;
```

```
            return self.execute_instruction(opcode);
        }
        Err("Interrupts disabled")
    }
```

Wenn Interrupts erlaubt sind, wird der übergebene Opcode ausgeführt und der Boolean geflipped, andernfalls wird eine Errormeldung zurückgegeben. `interrupts_enabled` ist initial `false` und muss durch die entsprechende Instruktion (`EI`) gesetzt werden. Prinzipiell kann jede 1 Byte große Instruktion (parameterlose Instruktion) als Interrupt ausgeführt werden, meistens wird jedoch eine der `RST`-Instruktionen ausgeführt. Diese sind `CALL`-Instruktionen, die zu einer fixen Speicheradresse springen. An diesen fixen Adressen kann der Programmierer die entsprechenden Interrupt-Routinen platzieren.

## 4.2 Assembler

Für den Emulator und andere Konsumenten des Assemblers soll dieser eine einzelne, geschlossene Schnittstelle sein. Dabei vereint er unterschiedliche Funktionalitäten, die in drei Modulen realisiert werden. Es findet eine funktionelle Aufteilung in die folgenden Dateien statt:

- Der eigentliche Assembler zum Übersetzen von Assemblycode und als öffentliche Schnittstelle
- Ein Präprozessor zur Behandlung von Pseudo-Instruktionen
- Ein Parser zur Auswertung numerischer Werte in unterschiedlichen Formaten

Um seine Funktionalität vollständig zu erfüllen soll der Assembler lediglich den vom Nutzer geschriebenen Code benötigen. Darauf aufbauend delegiert er dessen Verarbeitung intern mit Methodenaufrufen des Präprozessors.

Der Präprozessor ist eine „Pure Fabrication“ für den Assembler. Zwar gibt es in Rust keine Klassen, die eigentlich solche reinen Erfindungen sind, in diesem Fall wird das Konzept von einer Datei umgesetzt. Dabei beinhaltet diese diverse Methoden zum Verarbeiten von Pseudo-Instruktionen, wie sie in Kapitel 2.3.4 erläutert sind. Einzeln angewandt, sind die Methoden nur bedingt zu gebrauchen, da sie auf unterschiedlich weit verarbeitetem Code basieren. Deshalb übernimmt der Präprozessor die komplette Vorverarbeitung und bietet dafür die Methode `get_preprocessed_code()` nach außen hin an. Auf dem an dieser Stelle überlieferten Quellcode basierend, erstellt die Methode einen Vektor von Instruktionen, die der Assembler in den entsprechenden Bytecode umwandeln kann.

Zusätzlich zur Verarbeitung von Pseudo-Befehlen, soll der Präprozessor das Erstellen einer Map ermöglichen, die einen Zusammenhang zwischen den

erzeugten Bytes und den Zeilen, in denen sich der entsprechende Befehl befindet. Für diese Funktion bildet der Assembler entsprechend der Idee, die einzige Schnittstelle zu sein, einen Adapter für die Map, wodurch die interne Repräsentation und Implementierung unabhängig von Anforderungen bezüglich des Formats im Frontend wird.

Gemäß der Spezifikation erlaubt der Intel 8080 die Definition numerischer Werte in verschiedensten Formaten, unter anderem als mathematische Ausdrücke oder auch in Binärdarstellung. Um eine einheitliche Darstellung innerhalb von Rust zu gewährleisten und das Auslesen entsprechender Werte zu zentralisieren, nutzen sowohl Assembler als auch Präprozessor einen Parser. Ähnlich der zweiten Komponente handelt es sich hier um eine Pure Fabrication. Der entwickelte Parser, nimmt einen mathematischen Ausdruck als String entgegen und erzeugt davon ausgehend eine Menge Token, die stückweise abgearbeitet wird.

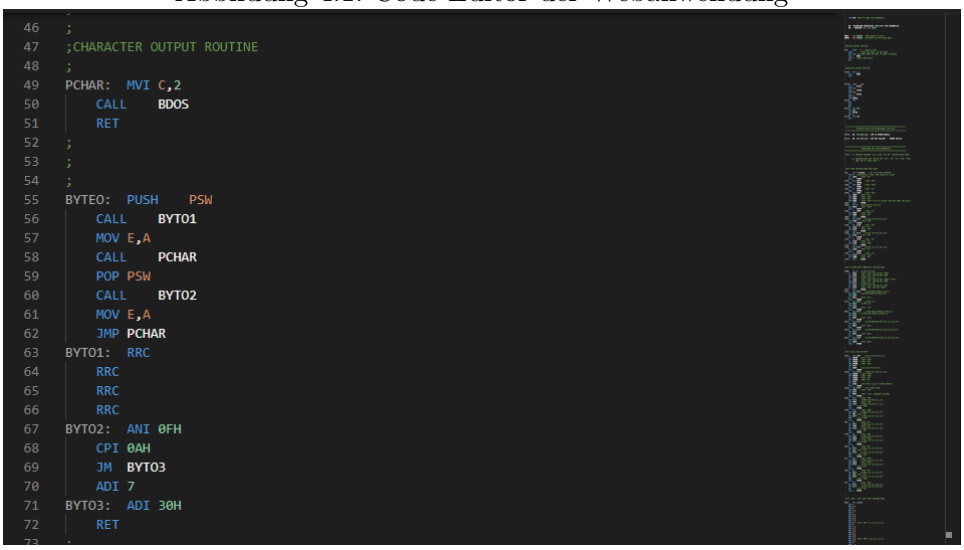
## 4.3 Disassembler

Ein Disassembler übersetzt Maschinencode zurück nach Assembly. Damit kann er zum Beispiel verwendet werden, um fremde Programme verständlich zu machen. Die Funktionsweise des Disassemblers ist sehr simpel: Die Eingabe wird Byte für Byte durchlaufen, durch ein großes Match-Statement wird der Opcode unterschieden und entsprechend des Opcodes werden 0, 1 oder 2 Bytes an Parametern gelesen. Der Opcode und die Parameter werden dann benutzt, um die entsprechende Assembly Instruktion als String zu erzeugen.

```
fn decode_next(&mut self) -> Result<String, &'static str> {
    // Get next Opcode
    let instr = self.read_byte();
    match instr {
        0x00 => Ok(String::from("NOP")),
        0x01 => Ok(format!(
            "LXI B,{}",
            Disassembler::fmt_hex::<u16>(self.read_addr())
        )),
        // ...
}
```

Der Disassembler liefert einen Iterator über `Result<String, &'static str>`, wobei die Results die einzelnen Zeilen des Assembly Programmes enthalten. Der Iterator läuft bis die Eingabe vollständig abgearbeitet ist.

Abbildung 4.1: Code-Editor der Webanwendung




## 4.4 Frontend

Die Webanwendung, mit der Nutzer den Emulator schließlich nutzen können, stellt zwei wesentliche Funktionalitäten zur Verfügung, die nachfolgend vorgestellt werden.

### 4.4.1 Code Editor

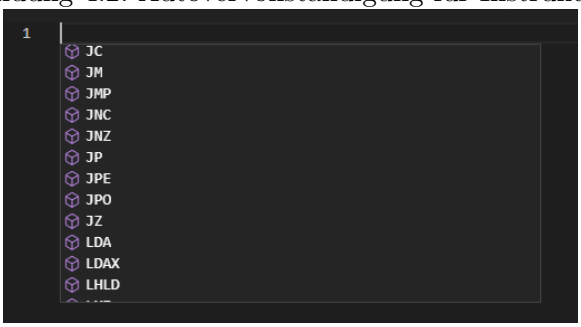
Mit einem eingebauten Code Editor können Nutzer direkt in der Webapplikation eigene Assembly-Programme schreiben und direkt ausführen, ohne beispielsweise vorher selbst den Code assemblen zu müssen und dann manuell in den Emulator zu laden.

Für den Code-Editor wird eine quelloffene Bibliothek von Microsoft genutzt, der sogenannte *Monaco Editor*. Monaco ist ein browser-basierter Editor, der praktische Funktionalitäten zur Verfügung stellt, wie zum Beispiel Autovervollständigung oder Syntax-Highlighting. Die Bibliothek wird unter anderem auch in dem weit verbreiteten und ebenfalls quelloffenen Code-Editor *Visual Studio Code* genutzt.

In Abbildung 4.1 sieht man den Code-Editor in Aktion. Die einzelnen Bestandteile des Assembly-Codes, die Labels, die Instruktionen und die Argumente, sowie Kommentare sind alle unterschiedlich eingefärbt.

In Abbildung 4.2 sieht man außerdem, wie die Autovervollständigung des Code-Editors funktioniert. Basierend auf der Eingabe des Nutzers und der Position im Code wird automatisch erkannt, ob eine Instruktion oder ein

Abbildung 4.2: Autovervollständigung für Instruktionen

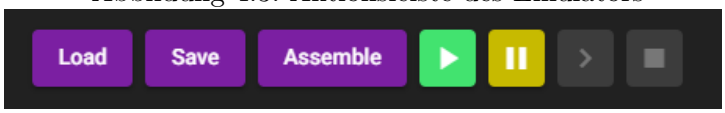



Argument vorgeschlagen werden soll und welche in Frage kommen.

### 4.4.2 Emulator-Zustand

Hat der Nutzer nun ein Assembly-Programm mithilfe des Code-Editors erstellt, kann er es ausführen. Hierfür wird der in Rust entwickelte Emulator genutzt, der mithilfe der Wasm-Schnittstelle in die Webapplikation eingebunden wird. Die Interaktion mit dem Emulator erfolgt mithilfe verschiedener Bedienelemente in einer Aktionsleiste am oberen Rand der Anwendung (siehe Abbildung 4.3).

Abbildung 4.3: Aktionsleiste des Emulators




Die ersten beiden Schaltflächen der Aktionsleiste, *Load* und *Save*, ermöglichen dem Nutzer, Assemblycode aus Dateien von seinem Endgerät zu laden oder sie dort zu speichern. *Load* kann außerdem auch bereits übersetzten

Abbildung 4.4: *Load*-Dialog

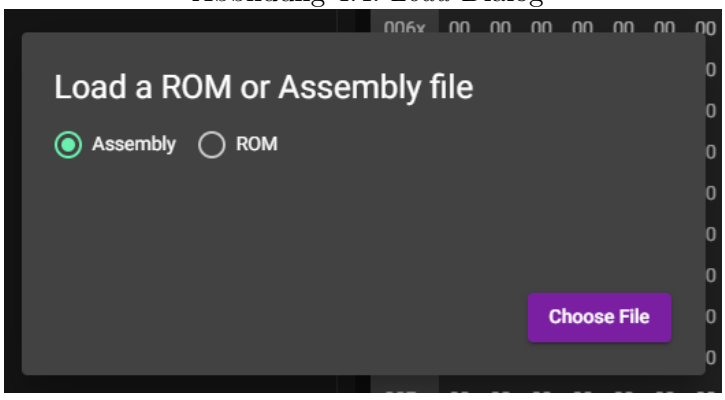
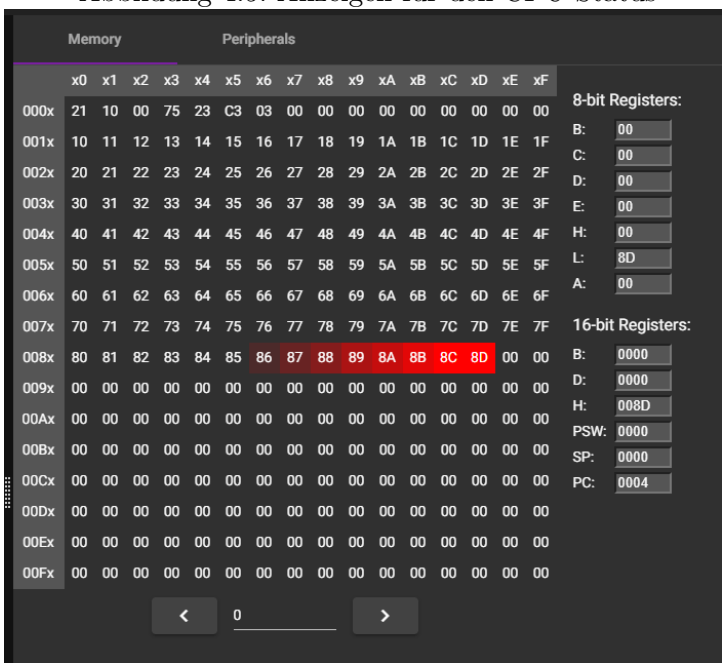


Abbildung 4.5: Anzeigen für den CPU-Status




Code direkt in den Speicher des Emulators laden. Die Auswahl erfolgt mithilfe eines eigenen Auswahldialogs, der in Abbildung 4.4 zu sehen ist. Wird ein fertiges Programm geladen, kann außerdem bestimmt werden, an welcher Stelle es im Arbeitsspeicher platziert werden soll. Dies ist notwendig, da viele Programme beispielsweise erwarten, dass sich der Einstiegspunkt an der Speicheradresse 256 befindet.

Um den Code, den der Nutzer geschrieben hat, nun zu assemblen, muss dieser in der Aktionsleiste die Funktion *Assemble* nutzen. Ist der Vorgang erfolgreich, werden die Bytes, die der Assembler erzeugt, in den Hauptspeicher des Emulators geschrieben. Die Emulation kann jetzt mithilfe der grünen Schaltfläche gestartet werden. Da die CPU keinen festgelegten Endzustand hat, läuft dieser solange weiter, bis der Nutzer die Emulation wieder beendet, was mit dem roten Stop-Knopf möglich ist.

Um den Ablauf des Codes nachvollziehen zu können, bietet die Applikation die Möglichkeit, jederzeit den internen Status des Prozessors einzusehen. Hierzu existiert auf der rechten Hälfte der Webseite eine Matrix, die den Arbeitsspeicher repräsentiert, sowie eine Anzeige für jedes der Register (siehe Abbildung 4.5).

Die RAM-Matrix stellt in jeder Zeile 16 Addressen dar, was die einfache Zuordnung von Zeile zu Spalte mithilfe der letzten Hexadezimalstelle erlaubt. Ändert sich im Arbeitsspeicher durch eine Instruktion des Prozessors ein

Wert, wird dieser für eine kurze Zeit rot hinterlegt, um einfacher zu sehen, was sich geändert hat. Da der Speicher in der Regel deutlich größer ist, als die dargestellten 16x16 Bytes, nämlich bis zu 64KB, ist es möglich, den angezeigten Speicherbereich zu verschieben.

Die Registeranzeigen rechts von der RAM-Matrix sind in zwei Kategorien gegliedert: Die 8-Bit Register (B, C, D, E, H, L, A) und die 16-Bit Register (BC, DE, HL, PSW, SP, PC).

Die aktuell ausgeführte Instruktion des Prozessors (basierend auf dem Program Counter) wird im Code-Editor außerdem rot hinterlegt. Für das Debugging von Programmen gibt es die Möglichkeit, die Ausführung des Emulators mithilfe der gelben Pause-Schaltfläche in der Aktionsleiste zu pausieren. Im pausierten Zustand kann nun jede Instruktion Schritt für Schritt ausgeführt werden.

# 5 Implementierung

## 5.1 Emulator

### 5.1.1 Registerarray

Im folgenden werden drei Aspekte der Implementierungen des Registerarrays gezeigt: Die Repräsentation der Register, der Zugriff auf die Register und der Zugriff auf die Flaggen.

#### Datentyp

Naive Implementierungen eines Registerarrays würden die Register einzeln implementieren und für Registerpaare die entsprechenden Inhalte konkatenieren. Dies ist jedoch unnötig umständlich. Der effizientere Ansatz ist die Register in Paaren zu speichern (als 16-Bit Unsigned Integer) und die Möglichkeit beizubehalten, die beiden Bytes individuell anzusprechen. In einer Sprache wie C ist dies mit Pointer-Arithmetik gut lösbar, in Rust ist es sinnvoller ein Union zu verwenden.

```
#[repr(C)]
union Register {
    bytes: (u8, u8),
    value: u16,
}
```

Ein Union wird ähnlich wie ein Struct deklariert, jedoch teilen alle Felder den gleichen Speicherplatz[1]. Das bedeutet man kann den Wert eines solchen `Register` entweder durch `Register::bytes` als Tupel aus 2 Bytes oder durch `Register::value` als 16-Bit-Wert auslesen. Dadurch ist keinerlei Konkatenation der Registerwerte notwendig.

[1] ausführliche Erklärung: `https://doc.rust-lang.org/reference/items/unions.html`

#### Registerzugriff

Der Registerzugriff ist eine sehr häufig verwendete Operation, da ein großer Teil der zu implementierenden Instruktionen sie benötigt. Um dies möglichst einfach zu machen, wurde Indizierung für den Registerarray implementiert. Über String-Indizierung — `reg["bc"] // Registerpaar BC` — ist Zugriff auf Registerpaare geregelt, über Character-Indizierung — `reg['b'] // Register B` — der normale Zugriff.

#### Flaggenzugriff

Die Flags sind bekannterweise Teil des PSW-Registerpaars, sprich sie sind als einzelnes Byte gespeichert. Um die Werte der einzelnen Flaggen zu erhalten, werden Bitmasken verwendet. Um bspw. herauszufinden, ob das Bit mit dem höchsten Stellenwert gesetzt ist, muss der Ausdruck `byte & 0x80 != 0` berechnet werden. Wenn dieser `true` ist, ist das Bit gesetzt[2].

### 5.1.2 Ein-/Ausgabegeräte

Wie in Kapitel 4 zu sehen ist, beinhaltet die zentrale Struktur zwei Arrays aus 256 Elementen, um Ein- und Ausgabegeräte zu realisieren. Der Übersichtlichkeit halber wurde der Datentyp dieser Arrays vereinfacht. Bei der Umsetzung als Array ist es problematisch, dass initial keine Geräte registriert sind, der Array also leer wäre. Da es in Rust kein `null` gibt, muss der Array mit dem `Option`-Typ befüllt werden:

```
[Option<InputDevice>; 256]
```

Diese Implementierung ist jedoch noch immer ungenügend, weil verschiedene Geräte registriert werden können. In klassischen objektorientierten Sprachen wären deshalb Input- bzw. OutputDevice als Interfaces realisiert, analog dazu gibt es in Rust Traits (siehe Kapitel 2). Da die Größe von Trait-Objekten nicht zur Compilezeit bestimmt werden kann, kann ein solches Objekt nicht auf dem Stack gelagert werden, was für die Initialisierung der eigentlichen Arrays aber erforderlich ist. Deshalb muss ein Pointer verwendet werden. Rust verwendet sogenannte „Smartpointer“ um die klassischen Probleme bei Verwendung von Pointern zu vermeiden. Meistens wird der Typ `Box` genutzt, der es allerdings verhindert Daten mehrfach zu referenzieren. Diese Möglichkeit ist jedoch notwendig, da sowohl von außen, als auch von innen mit den Geräten kommuniziert werden muss. Durch Kombination von `Rc`, einem referenzzählendem Pointer, mit `RefCell`, einer schreibbaren Speicherregion, kann das gewünschte Verhalten erreicht werden. Die Syntax um einen derartigen Array zu deklarieren ist wie folgt:

```
[Option<Rc<RefCell<dyn InputDevice>>>; 256]
```

[2] `0x80 == 0b10000000`

Solange das Programm als einzelner Thread ausgeführt wird, kann nun jederzeit mit einer mutable Referenz auf ein solches InputDevice gearbeitet werden. Außerdem dürfen beliebig viele non-mutable Referenzen existieren, sofern im aktuellen Scope keine mutable Referenz auf das Gerät existiert.

## 5.2 Assembler

Die Implementierung des Assemblers besteht aus drei maßgeblichen Komponenten: Dem eigentlichen Assembler, dem im Code sogenannten „preprocessor“ zur Vorverarbeitung und einem Parser für numerische Eingaben des Nutzers. In den folgenden Abschnitten wird auf verschiedene Besonderheiten und Herausforderungen der drei Komponenten eingegangen, was mittels Code-Beispielen illustriert wird. Generell gilt, dass der führende und schließende Whitespace einer Zeile des Programmcodes vor deren Behandlung durch die Methode `.trim()` entfernt und die Zeile gegebenenfalls übergangen wird, sofern eine Leerzeile entsteht. Der Übersichtlichkeit halber ist dieser Teil des Codes an vielen Stellen ausgelassen.

### 5.2.1 Assembler.rs

Der in der Klasse `assembler.rs` definierte Struct

```
pub struct Assembler {
    code: Vec<String>,
}
```

bildet die Schnittstelle für den eigentlichen Emulator und das Wasm-Interface. Der Emulator benötigt den zu Bytecode übersetzten Input des Nutzers und liefert dem Assembler dafür bei der Initialisierung ein String-Slice, das diesen Input repräsentiert. Basierend darauf erzeugt der Assembler das Objekt `code`, wobei je Zeile ein neuer String erzeugt wird.

Bereits zu diesem Zeitpunkt können Kommentare, also alles in einer Zeile nach einem „;“, entfernt werden. Dabei ist zu beachten, dass nur der Text der Kommentare und das Semikolon entfernt werden darf, die dadurch entstehenden Leerzeilen müssen erhalten bleiben (siehe 5.2.2). Obiges Struct implementiert neben der Methode `assemble()` zur Generierung des Bytecodes eine Methode, die diese Bytes auf den Index der jeweiligen Zeile mappt. Das nutzt die Web-Schnittstelle um bei schrittweiser Ausführung die aktuelle Zeile ersichtlich zu machen.

Der Assembler selbst beschränkt sich nach außen hin auf diese beiden Methoden, wobei zunächst `assemble` zu betrachten ist, auf das Mapping wird in 5.2.2 genauer eingegangen.

In seiner einfachsten Form besitzt `assemble()` den Aufbau aus Listing 5.2.1.

**Quellcode 5.1** Grundlegender Aufbau der Methode `assemble`

```
pub fn assemble(&self) -> Result<Vec<u8>, &'static str> {
        let ppc = get_preprocessed_code(&self.code)?;
        let mut byte_code: Vec<u8> = Vec::new();
    for line in ppc {
        if !line.contains("ORG ") {
                let bytes = to_machine_code(line)?;
            byte_code.extend(bytes);
        }
    }
    Ok(byte_code)
}
```

Diese Implementierung erzeugt aus dem, dem Assembler bei Initialisierung übergebenen Code, einen Vektor von positiven 8-Bit-Werten. Da es sich beim ursprünglichen Code um Nutzereingaben handelt, besteht die Gefahr, dass Syntaxfehler eine Übersetzung fehlschlagen lassen. Deshalb ist die Rückgabe vom Typ `Result`, womit bei einem Fehlschlag eine entsprechende Fehlermeldung zurückgegeben werden kann, anstatt dass das Programm abstürzt.

Die grundlegende Funktionsweise ist das Iterieren über alle Zeilen des vom Präprozessor behandelten Codes und ein Übersetzen der jeweiligen Zeile in Bytecode. Zeilen, in denen eine nutzerdefinierte Speicheradresse (vergleiche Origins, 2.3.4) deklariert ist, werden durch die Vorverarbeitung mittels Präprozessor (Zeile 2) nicht entfernt, weshalb diese Zeilen noch einmal gesondert übergangen werden müssen. Die Deklarationen der Origins müssen erhalten werden, da der Assembler die vom Nutzer deklarierten Speicheradressen selber ausliest, dafür allerdings den vorverarbeiteten Code nutzt.

Während der Entwicklung ist aufgefallen, dass es inhaltlich sinnvoll wäre, wenn der Assembler, der die Origins ohnehin ausliest, diese auch direkt auswertet. Deshalb wurde das if-Statement im Quellcode 5.2.1 um folgende Schleifen erweitert:

**Quellcode 5.2** Beachtung von Origins im Assembler

```
let origins = self.get_origins();
for (origin_index, next_address) in &origins {
        if current_byte_index == usize::from(*origin_index) {
        current_address = *next_address;
    }
}
while byte_code.len() < current_address.into() {
    byte_code.push(0);
}
```

Zunächst werden alle Origins bestimmt, welche sich als Kombination aus Byte-Index und Speicheradresse verstehen lassen. Das bedeutet der Assembler erstellt in der ersten Zeile einen Vektor vom Typ `Vec<(u16, u16)>`, bei dem der erste Wert der Index eines Bytes des Bytecodes ist, der zweite Wert ist die vom Nutzer definierte Speicheradresse, in die geschrieben werden soll, sobald das entsprechende Byte vom Emulator erreicht wurde. Um Bezug auf diese Angaben zu nehmen muss je Zeile überprüft werden, ob das aktuelle Byte (`current_byte_index`) ein Byte ist, für das es eine explizite Speicheradresse gibt. Sofern dies der Fall ist wird die aktuelle Adresse auf den Wert der neuen Speicheradresse gesetzt.

Damit der Vektor nun an der richtigen Stelle fortgeführt wird, werden die Adressen zwischen der zuletzt beschriebenen und der neuen aktuellen Adresse mit dem Wert 0 aufgefüllt. Wichtig ist, dass dieser Wert ein Platzhalter ist und nicht als „0“ verstanden werden darf. Auch der Emulator muss an dieser Stelle darauf achten, dass er diese Werte nur an den Stellen als solche versteht, an denen ein entsprechender Wert erwartet wird (beispielsweise bei Befehlen wie `LXI`, denen eine Zahl folgt).

Aufgrund dieser Erweiterung werden die generierten Bytes im Quellcode 5.2.1 in einer separaten Variable gespeichert, da der Index des aktuellen Bytes davon abhängt und entsprechend mittels

`current_byte_index += bytes.len();` aktuell gehalten wird.

### 5.2.2 Vorverarbeitung

Unter Vorverarbeitung versteht sich im Bezug auf diese Arbeit und den Assembler alles was zwar vom Assembler übersetzt wird, allerdings keine Verarbeitung eines Opcodes ist. Dabei wird der vom Nutzer eingegebene Code (ohne Kommentare) auf die Zeilen reduziert, die in den eigentlichen Bytecode übersetzt werden. Schritte der Vorverarbeitung sind:

1. Bestimmung eines korrekten Programmendes
2. Ersetzen von Makros
3. Ersetzen von Variablen
4. Bestimmung der Labels
5. Ersetzen von Referenzen auf den Programmzähler
6. Ersetzen von Referenzen mittels Label
7. Entfernen von evaluierten Pseudo-Instruktionen

Im Quellcode werden diese Aufgaben von der Klasse `preprocessor.rs` übernommen. Dabei handelt es sich um eine reine Erfindung innerhalb des As-

semblers, die außerhalb der Dokumentation, beziehungsweise dem eigentlichen Aufbau des originalen Intel 8080s existiert. Bezüglich der Struktur der folgenden Abschnitte sei angemerkt, dass das Ersetzen von Makros, des Umfangs wegen, erst in 5.2.3 erläutert wird, beim Assemblen aber wie oben gezeigt noch vor der Behandlung von Variablen stattfindet.

Neben der Vorverarbeitung bietet der „Präprozessor“ eine Methode zum Mappen von Zeilen. Diese bietet das Assembler-Objekt als eine Art Adapter nach außen hin an. Auf sie wird am Ende dieses Kapitels näher eingegangen. Zunächst sind die maßgeblichen Schritte der eigentlichen Vorverarbeitung zu betrachten. Die nachfolgende Sammlung von Vorgängen (als eigene Methoden realisiert) fasst der Präprozessor in einer einzelnen, öffentlichen Methode `get_preprocessed_code(code: &Vec<String>)` zusammen, die dem Assembler zur Verfügung gestellt wird.

Generell gilt, dass die meisten dieser Methoden dem Schema

```
fn method(code: &Vec<String>) ->
```

```
 Result<Vec<String>, &'static str>
```

folgen. Das bedeutet, dass sie auf der Referenz eines Vektors aus Strings operieren und einen neuen Vektor aus Strings zurückgeben. Diesem Vektor werden, je nach Auswertung, entsprechende Zeilen angehängt. Der Rückgabewert wird im folgenden auch als „Ergebnisvektor“ einer Methode bezeichnet und beinhaltet eine veränderte Form des ursprünglichen Programmcodes.

### Bestimmung des Programmendes

Weil dieser Schritt eindeutig und essentiell bezüglich der Korrektheit eines Assembler-Programms ist, ist es der erste Schritt in der Vorverarbeitung. Sinn dieses Schrittes ist es festzustellen, ob ein Programm ordnungsgemäß mit dem `END`-Befehl abgeschlossen wurde. Sollte dies nicht der Fall sein ist jeder folgende Schritt überflüssig und das Übersetzen kann mit einer Fehlermeldung abgebrochen werden.

Zur Bestimmung genügt es über alle Zeilen des Codes zu iterieren und festzustellen, ob eine einzelne Zeile `END` lautet. Dabei ist wichtig, dass tatsächlich alle Codezeilen untersucht werden und nicht nach dem ersten `END` ein `true` geliefert wird. Deshalb bringt es auch keine nennenswerten Performanceersparnisse, sollte die Iteration am Ende des Codes beginnen.

Der erste Quellcode in 5.2.2 zeigt die Implementierung der Überprüfung nach dem `END`. Es sind drei Fälle zu beachten, in denen das Programm inkorrekt ist und der vorliegende Code mit einem `false` antworten muss. Der einfachste Fall ist die Abwesenheit des Befehls, wobei die entsprechende Variable nie `true` wird und als solche zurückgegeben wird. Im zweiten Fall folgt auf das `END` eine nicht-leere Zeile, was vom letzten If-Statement ge-

**Quellcode 5.3** Überprüfung auf ein END-Statement

```
let mut has_end = false;
for line in code {
    if line.is_empty() {
        continue;
    }
    if line.trim().eq("END") {
        if has_end {
            return false;
        }
        has_end = true;
        continue;
    }
    if has_end {
    return false;
    }
}
has_end
```

deckt wird, welches ein `false` zurückgibt, sobald ein END gefunden wurde, eine danach folgende Zeile allerdings nicht leer ist. Der letzte Fall ist das Vorhandensein mehrerer ENDs, was mit einer verschachtelten If-Klausel in Zeile 6 abgedeckt wird.

### Ersetzen von Variablen

Damit Instruktionen wie If-Verzweigungen korrekt funktionieren und ausgewertet werden können sind nun Variablen durch ihre konstanten Werte zu ersetzen. Dadurch, dass Referenzen auf Makros bereits ersetzt und deren lokale Variablen gegebenenfalls umbenannt wurden (5.2.3), ist dieser Schritt relativ trivial, weil alle Variablennamen gleich zu behandeln sind.

Somit genügt es den Code Zeile für Zeile zu durchlaufen, wobei das Bestimmen gültiger Variablen sowie deren Ersetzung parallel erfolgen. Das schließt von vornherein aus, dass verwendete Variablen, die noch nicht deklariert wurden, fälschlicherweise ersetzt werden. Für Variablen, die mittels SET deklariert wurden, sieht das Ganze, auf seine grundlegende Logik reduziert, wie folgt aus:

Die genaue Funktionsweise der Methode `replace_names` aus Zeile zwei ist im Quellcode in 5.2.3 näher beschrieben. Letztlich beläuft sie sich darauf, die Vorkommen der Schlüssel der angegebenen Map in einem String mit den entsprechenden Werten zu ersetzen.

Neben dem Ersetzen von Variablen gilt es die zugehörigen Werte auszule-

**Quellcode 5.4** Bestimmen und Ersetzen von mittels SET deklarierter Variablen

```
for (key, value) in &set_assignments {
    line = replace_names(&line, assignment_map);
}
if line.contains(" SET ") {
    let (name, exp) = line.split_once(" SET ").unwrap();
    set_assignments.insert(
        name.to_string(),
        eval_str(exp.to_string())
    );
}
```

sen, was mit einer simplen Überprüfung nach den entsprechenden Pseudo-Befehlen geschieht. Sofern ein solcher vorhanden ist, wird die Zeile aufgeteilt und in eine Map vom Typ `HashMap<(String, u16)>` aufgenommen. Das geschieht unter der Bedingung, dass der Name dem vom Intel 8080 vorgegebenen Format entspricht, wofür getestet wird, dass er keinem reservierten Namen (Opcodes, Pseudo-Instruktionen etc.) entspricht und einem vorschriftsgemäßen Aufbau folgt. Diese Überprüfung ist einerseits das Nicht-Vorhandensein in einem Array aus reservierten Namen, andererseits das Entsprechen eines Regex.

Aus Gründen der Übersichtlichkeit ist die namentliche Überprüfung, sowie das Verfahren beim Befehl EQU nicht mitaufgenommen. Letzteres funktioniert beinah analog, wobei es sich zu SET insofern anders verhält, dass in dem Fall, sollte sich ein Variablenname bereits in der Map befinden, der zugehörige Wert nicht überschrieben, sondern eine Fehlermeldung propagiert wird.

Damit beim Iterieren über den Code Variablenzuweisungen, die innerhalb eines If-Blocks stehen, entsprechend behandelt werden, bedarf es einer weiteren Überprüfung des Zeileninhalts. Dazu werden zwei boolsche Variablen `in_conditional` und `condition` definiert, die den aktuellen Zustand bei der Verarbeitung repräsentieren. Sobald ein If-Statement passiert wird, wird die entsprechende Variable auf `true` gesetzt. Zusätzlich wird mittels Parser der Wert des Ausdrucks der Bedingung bestimmt und in der zweiten Variable angegeben, ob dieser ungleich „0“ ist. Sollte eine Zeile nun eine Variablenzuweisung beinhalten, sich aber in einem als `false` evaluierten If-Block befinden, wird die Zuweisung einfach übergangen.

### Bestimmung von Labels

Zum jetzigen Zeitpunkt besteht der Quellcode bereits nur noch aus Opcodes, Labels, Referenzen auf den Programmzähler und den Pseudo-Anweisungen

`IF`, sowie `EQU` und `SET`.

Im nächsten Schritt sollen die im Kapitel 2.3.4 vorgestellten Labels ausgelesen werden. Hier geht es nur um die Bestimmung deklarierter Label, ein Ersetzen der Referenzen durch die deklarierten Werte erfolgt an anderer Stelle. Weil Labels sich nicht auf einen Zeilenindex, sondern Byteindizes beziehen, müssen auch an dieser Stelle Origins beachtet werden, genauso wie der Inhalt der eigentlichen Zeile. Allerdings hat die Vorverarbeitung keinen Zugriff auf die Methode `to_machine_code` des Assemblers zur Überführung einer Zeile in ihren Bytecode.

Deshalb wurde eine Hilfsmethode

`get_byte_amount_of_line(opcode: &String) -> u16` eingeführt, die einen gegebenen Opcode in seine entsprechende Anzahl an Bytes im Speicher überführt. Wegen ihrer trivialen Funktionsweise (Prüfung ob sich der gegebene Opcode in einem bestimmten Vektor befindet) sei hier lediglich auf die Existenz und Verwendung der Methode hingewiesen.

Ähnlich der vorhergehenden Schritte wird zeilenweise über den Code iteriert. In einer Variable `mem_address` wird der aktuelle Byteindex gespeichert und aktualisiert, sofern eine Zeile ein `ORG`-Statement oder einen Opcode beinhaltet. In der resultierenden `HashMap<String, u16>` verweist dann ein gefundenes Label auf den Wert der zum Zeitpunkt ihres Funds in der `mem_address` steht.

Das Identifizieren von Labels erfolgt mittels folgendem Regex:

$\widehat{}(\Lambda[a-zA-Z@?][a-zA-Z@?0-9]\Lambda :)$, was die in den Grundlagen vorgestellte Spezifikation abbildet. Es sei zu beachten, dass dieser Ausdruck alle Namen matcht, auch die mit einer Länge von mehr als fünf Zeichen. Deshalb muss jeder so gefundene Name, bevor er weiterverarbeitet wird, entsprechend gekürzt werden. Sofern die Bezeichnung des Labels valide ist und überschüssige Zeichen gekürzt wurden wird das Label einem Vektor für „temporäre“ Label hinzugefügt.

Weil eine beliebige Menge Label auf dasselbe Byte verweisen kann wird prinzipiell davon ausgegangen, dass ein Label nicht unmittelbar vor einem Opcode steht. Stattdessen werden sie in diesem separaten, „temporären“ Vektor gesammelt und in dem Fall, dass ein Opcode angetroffen wird, alle in diesem Vektor gesammelten Label dem eigentlichen Ergebnis beigefügt. Der Vektor hat einen weiteren Vorteil: In dem Fall, dass ein Label auf nichts verweist (man stelle sich vor die letzte Zeile eines Programms lautet `instr:`) ist der Vektor zum Schleifenende hin nicht leer. Das kann überprüft werden und gegebenenfalls ein entsprechender Fehler zurückgeliefert werden.

#### Weitere Behandlung

Zum jetzigen Zeitpunkt benötigt der Präprozessor einen letzten Durchlauf über den Code. In diesem letzten Schritt werden Referenzen auf den Programmzähler (ausgedrückt als „$“) und Verwendungen von Labels ersetzt, die Deklarationen letzterer entfernt, übrige Pseudo-Instruktionen (`EQU`, `SET`) übergangen und If-Blöcke entsprechend evaluiert.

All diese Aufgaben belaufen sich auf simples Überprüfen von Strings auf gewisse Inhalte (bspw. `line.contains("\$")`) oder die Verwendung an anderer Stelle vorgestellter Konzepte und Methoden. So werden Referenzen auf Label mittels dem Quellcode in 5.2.2 ersetzt. Der Wert für den Programmzähler wird, entsprechend dem Vorgehen bei der Bestimmung der Labels mitgeführt und an den jeweiligen Stellen eingefügt. Die Evaluierung von bedingten Anweisungen erfordert ebenfalls keine neue Logik, sondern geschieht nach dem in 5.2.2 erläuterten Konzept (auch hier gibt es Optimierungspotential, das aus zeitlichen Gründen noch nicht erschöpft wurde).

Nach diesem letzten Durchlauf wurden alle Pseudo-Befehle aus dem ursprünglichen Code entfernt. Das Ergebnis ist ein Vektor von Strings, die entweder leer sind, oder einen Opcode beinhalten. Somit sind alle Zeilen eindeutig in ihre entsprechende Repräsentation aus Bytes übersetzbar, was der eigentliche Assembler übernimmt.

### 5.2.3 Makros

Im Programmcode gelten Makros zwar als Teil der Vorverarbeitung, ihre Mächtigkeit und damit einhergehende Komplexität setzt aber eine differenziertere Betrachtung voraus. Die Verarbeitung von Makros geschieht in drei unterschiedlichen Schritten, die nachfolgend erläutert werden. Zum Zeitpunkt der Makroverarbeitung wurden im Code bereits alle Entwicklerkommentare entfernt worden und es wurde festgestellt, dass das Programm ein korrektes Ende besitzt.

#### Deklarationen

Bevor ein Makro aufgerufen werden kann muss es deklariert werden. Die Syntax dafür wurde bereits in Kapitel 2.3.4 vorgestellt. Die Bestimmung dieser Deklarationen macht den ersten Schritt der Verarbeitung von Makros aus. Das Ergebnis ist ein Tupel von zwei `HashMap<String, Vec<String>>`s, wobei einmal der Name des Makros auf die enthaltenen Instruktionen und das andere Mal auf seine Liste von Parametern gemappt wird.

Ähnlich anderer Vorverarbeitungsschritte muss auch an dieser Stelle über den gesamten Code iteriert werden. Besonderer Behandlung bedarf es dabei vor allem der Zeilen in denen der Start eines Makros steht (`name MACRO list`)

und denen in welchen ein Makro abgeschlossen wird (ENDM). Das gesamte Verhalten während des Schleifenablaufs wird maßgeblich von der boolschen Variable `in_macro` beeinflusst. Sie ist als `false` initialisiert und wird in Abhängigkeit von Makrostart und -ende entsprechend verändert.

Sollte ein Makrobeginn festgestellt werden, während die Variable wahr ist, kann bereits an dieser Stelle abgebrochen werden, da der Intel 8080 es nicht erlaubt ein Makro innerhalb eines Makros zu definieren. Auch aufkommende ENDM-Befehle, die keinen entsprechenden Anfang besitzen können mittels dieser Variable identifiziert werden. Bis auf die Ausnahme von Makrodeklarationen werden alle Zeilen, sofern `in_macro == true` gilt in einen separaten Vektor aufgenommen, auf den später der Makroname gemappt wird. In jedem anderen Fall handelt es sich um eine Zeile, die außerhalb eines Makros steht und für den weiteren Verlauf bezüglich der Makros nicht mehr relevant ist.

Der restliche Arbeitsaufwand für das Finden von Makrodeklarationen beläuft sich auf das Auftrennen von Strings und eine geeignete Speicherung all dieser Werte. Der Quellcode in 5.2.3 illustriert das Ganze beispielhaft.

**Quellcode 5.5** Auftrennen einer Definition eines Makros

```
let split: Vec<&str> = line.split("MACRO").collect();
macro_name = split[0].to_string();
if macro_name.is_empty() {
    return Err("Cannot define macro without name");
}
for par in split[1].split(",") {
    if !par.is_empty() {
        parameters.push(par.trim().to_string());
    }
}
```

Dabei wird eine Zeile, sofern sie das Schlüsselwort „MACRO“ enthält, an eben diesem aufgetrennt. Das Ergebnis ist der Einfachheit halber ein Vektor, dessen erstes Element der Name des Makros ist, das zweite eine mögliche Liste von Parametern als String. Darauf aufbauend kann der Name (an dieser Stelle vereinfacht) validiert werden und der String der Parameter mittels einem wiederholten Auftrennen in einen Vektor aus den einzelnen Parameternamen überführt werden.

Sowohl die Parameter, als auch der Code eines Makros werden, solange kein Ende gefunden wurde, in separaten Variablen, die nicht dem Rückgabewert der Methode entsprechen, gespeichert. Ihre Inhalte werden zum Endergebnis hinzugefügt und die Variablen geleert, sobald ein ENDM aufkommt.

### Referenzierungen

Nach dem der Assembler eine Liste aller existierenden Makros bestimmt hat und ihre Namen entsprechenden Parametern sowie einer Liste an Befehlen zuordnen kann, ist der nächste Schritt die Stellen, an denen ein Makro referenziert wird, durch das entsprechende Makro zu ersetzen. Außerdem kann nun der Code, mittels welchem die Makros als solche deklariert wurden, entfernt werden. Das Konzept des Entfernens entspricht dem der vorhergehenden Implementierung mittels boolscher Variable (Bestimmung ob in einer Makrodeklaration, wenn ja Zeile nicht an Ergebnis anheften), weshalb im Folgenden vor allem auf die Herausforderung, die Parameter ordentlich zu ersetzen, eingegangen wird.

Die Stellen, an denen eine Makroreferenz durch seine Instruktionen ersetzt wird, werden in zwei unterschiedliche Strings gekapselt, sodass im Ergebnisvektor vor den Befehlen der String `"MACRO_START"` und nach den Befehlen `"MACRO_END"` steht. Auf den genauen Grund für diese Implementierung wird noch in 5.2.3 eingegangen werden.

Bei der Erkennung von Makros wird je Zeile überprüft, ob ihr Inhalt einem Schlüssel in der, im vorherigen Schritt generierten, Map entspricht. In diesem Fall müssen zuerst eventuelle Parameter ermittelt werden. Das ist auf dieselbe Art und Weise implementiert, wie im obigen Quellcode 5.2.3 vorgestellt. Damit die Variablen innerhalb eines Makros durch ihre eigentlichen Werte ersetzbar sind, müssen diese entsprechend gemappt werden, was im Quellcode in 5.2.3 zu sehen ist.

**Quellcode 5.6** Mapping von Nutzereingaben auf Parameter

```
let params = macro_params.get(macro_name).unwrap();
for (index, parameter) in params.iter().enumerate() {
        let value = if index >= inputs.len() {
            String::new()
    } else {
            inputs[index].to_string()
    };
    input_map.insert(parameter.to_string(), value);
}
```

Dazu wird über die entsprechende Menge von Parametern, die für das Makro definiert wurden, iteriert. An dieser Stelle ist wichtig, dass über eine Aufzählung der eigentlichen Werte iteriert wird, weil der Index der Parameter über die Zuordnung bestimmt. Das ist in Zeile 6 ersichtlich, in der der beim Makroaufruf angegebene Wert ausgelesen wird. Gemäß der Spezifikation des Assemblers ist es zulässig nicht alle definierten Parameter anzugeben. In dem Fall muss der Assembler davon ausgehen, dass nur die ersten Para-

meter (links beginnend) übergeben wurden und ersetzt die übrigen Werte mit einem leeren String. Im Code wird dies durch einen Vergleich zwischen Index und Anzahl aller beim Aufruf übergebenen Parameter realisiert, wobei eben jene Parameter fehlen, für die es keinen Index gibt.

Anschließend wird jede Instruktion des Makros an den Ergebnisvektor angehängt. Bei diesem Vorgang werden entsprechende Vorkommnisse von Parametern in den Befehlen ersetzt. Weil der Nutzer Variablennamen als Parameter angeben kann, könnten möglicherweise unerwartete Seiteneffekte auftreten, sobald ein Parameter Substring eines anderen ist, weshalb das genaue Vorgehen an der vereinfachten Fassung im Quellcode in 5.2.3 zu sehen ist.

**Quellcode 5.7** Ersetzung von Variablen in Makros einer einzelnen Instruktion (vereinfacht)

```
while let Some(reg_match) = var_regex.find(&line) {
    let last_match =  line.get(reg_match.end()..);
    let start = match first_match {
        " " | "," | "+" | "-" | "*" | "/" => reg_match.start()+1,
        _ => reg_match.start()
    };
    let end = match last_match {
        " " | "," | "+" | "-" | "*" | "/" => reg_match.end() - 2,
        _ => reg_match.end() -1
    };
let value_string = &format!("{}{}", &value, substr_prot);
line.replace_range(start..end - 1, value_string);
// identischer Ablauf für letztes Aufkommen eines Namens
line.replace(replacement_protection, "").trim().to_string()
```

Der Quellcode basiert auf den Regex `var_regex` und `end_regex`, die mit den Strings

`&format!(r"[ ,]{}[ ,+\-*/,].", variable)`, bzw.

`&format!(r"[ ,]{} ?\$", variable)` definiert sind. Sie matchen Vorkommnisse von Parameternamen in einer Zeile und Verhindern, dass nur ein Substring des eigentlichen Namens gefunden wird. Mittels dem ersten der beiden Regex werden alle Variablennamen, die nicht am Ende der Zeile stehen gefunden, das zweite Regex findet alle Variablen die am Ende einer Zeile stehen. Im Fall eines Treffers gibt die Methode `find()` (Zeile 1) ein `Option<Range>`-Objekt zurück dessen Inhalt, sofern vorhanden, in `reg_match` gespeichert wird. Der darin enthaltene Bereich läuft vom ersten Index des Treffers bis zum Letzten. Aus diesem Bereich lassen sich das erste und letzte Zeichen des gematchten Strings bestimmen, welche in den Varia-

blen `first_match` und `last_match` gespeichert sind. Die Bestimmung von letzterem ist als Referenz in Zeile 2 zu sehen.

Abhängig davon, ob sich vor dem ersten (bzw. nach dem letzten) Symbol des verglichenen Namens ein weiteres Symbol (hier sind nur Sonderzeichen möglich, die kein Teil von Variablennamen sein dürfen) befindet, muss der Index angepasst werden. Das geschieht in den Zeilen 3 bis 10. Sie definieren in Abhängigkeit des ersten und letzten Zeichens einen weiteren Index. Die beiden Indizes `start` und `end` verweisen somit auf den ersten und letzten Buchstaben des gefunden Namens im gesamten String `line`.

Nach einer erfolgreichen Bestimmung kann der so ermittelte Bereich durch den eigentlichen Wert ersetzt werden. Die erforderliche Methode stellt Rust dafür standardmäßig bereit. Allerdings wird der Wert mit einem Platzhalter (`substr_prot`) erweitert. Bei ihm handelt es sich um einen String, der keiner legalen Nutzereingabe entspricht. Er dient dazu bereits ersetzte Namen kein weiteres Mal zu ersetzen.

Zur Verdeutlichung ist in Tabelle 5.1 ein Beispiel illustriert. Für das Beispiel sei das Makro `mac` angenommen, das die zwei Parameter `FooBar` und `Foo` (in der Reihenfolge) besitzt. Der Inhalt des Makros ist der Befehl `MOV FooBar, Foo` (kein legaler Ausdruck, dient ausschließlich der Veranschaulichung). Außerdem wurde `Foo` vorher im Programm als Variable initialisiert.

Tabelle 5.1: Ersetzung ohne Platzhalter

| | |
|---|---|
| `mac Foo, A` | Referenz von `mac` wobei FooBar = Foo, Foo = A |
| `MOV FooBar, Foo` | Befehl vor Ersetzung |
| `MOV Foo, Foo` | FooBar mit seinem Wert ersetzt |
| `MOV A, Foo` | Erster Aufkommen des Parameters Foo ersetzt |
| `MOV A, A` | Zweites Aufkommen des Parameters Foo ersetzt |

Es wird schnell ersichtlich, dass dieses Ergebnis nicht das ist, was der Programmierer erwarten würde. Durch das Hinzufügen eines Werts, den der Nutzer nicht benutzen kann, bzw. darf, wird diesem Verhalten vorgebeugt. Denn dann wird in der dritten Zeile aus `FooBar` „Foo@ %“, wobei hier als Platzhalter der String „@ %“ gewählt wurde. Nun matcht der Inhalt des zweiten Parameters nur noch seinem eigentlichen Aufkommen am Ende der Zeile. Sobald alle Parameter auf diese Weise ersetzt wurden, können die Platzhalter einfach mit einem leeren String ersetzt werden. Hier ist es wichtig, dass der Platzhalter so gewählt wird, dass er nicht zufällig in seinem letzten Zeichen dem ersten Zeichen eines Namens entspricht, weshalb das „%“ gewählt wurde.

Das Ergebnis dieses gesamten Vorgangs ist der ursprüngliche Assemblycode, dessen Referenzen von Makros durch deren eigentlichen Inhalt ersetzt wur-

den. Der Inhalt ist von zwei, im Quellcode dafür definierten Strings, umgeben. Außerdem sind die Parameter innerhalb der Makros durch entsprechende Werte oder Variablen ersetzt.

### Lokale Referenzen

Der letzte Schritt bezüglich dem Ersetzen von Makros behandelt das Referenzverhalten von Variablen innerhalb der Makroaufrufe. Weil bei der gewählten Implementierung zuerst alle Makroreferenzen ersetzt werden, ohne dass das Lokalverhalten beachtet wird, muss an dieser Stelle noch erkennbar sein, wo sich Aufrufe von Makros befinden. Dafür dienen die zu Beginn des letzten Abschnitt erwähnten Strings. Diese Stelle bietet für zukünftige Arbeiten Platz zur Optimierung insofern, dass der kommende Schritt mit vorherigem kombiniert wird, aus Zeitgründen ist das jedoch vernachlässigt worden.

Um das Ersetzen der Makros abzuschließen müssen folgende Dinge getan werden:

- Nicht global definierte Label müssen, je Referenz, durch einen einzigartigen Namen ersetzt werden
- Global definierte Label müssen zu „normalen“ Label umgewandelt werden
- Variablen die mittels `SET` deklariert wurden und keinen globalen Kontext besitzen müssen einen einzigartigen Namen erhalten

Die entsprechende Methode beginnt in einer ersten Iteration über den Code damit alle per `EQU` definierten Variablen und Labels, die sich außerhalb von Makroaufrufen befinden, zu finden und die jeweiligen Namen in Vektoren zu speichern.

Nun beginnt der eigentliche Arbeitsvorgang. Dieser produziert einen Ergebnisvektor, der den ursprünglichen Code enthält, mit dem Unterschied, dass alle Makros entsprechend verarbeitet wurden. Es ist somit nicht mehr (direkt) ersichtlich ob eine Zeile Code als solche geschrieben oder mittels Makro generiert wurde. Um das zu bewerkstelligen wird der Quellcode Zeile für Zeile betrachtet und folgende Dinge getan:

1. Es wird, anhand der vorher genannten Strings, geprüft ob es sich um den Beginn / das Ende eines durch Makro generierten Code handelt. Dieser Zustand wird in der Variablen `in_macro` gespeichert.
2. Falls die Zeile aus einer Variablenzuweisung per `SET` besteht und diese außerhalb eines Makros steht, wird der Variablenname in einem eigenen Vektor abgelegt.

3. Es findet eine Unterscheidung statt, ob sich die momentan betrachtete Zeile innerhalb eines Makros befindet. Ist das nicht der Fall wird sie als solche an den Ergebnisvektor angehängt. Ansonsten treffen die nachfolgenden Schritte zu.
4. Die Zeile wird auf lokale Label untersucht, für die ein eigener Name generiert wird. Das so entstandene Paar wird in einer Map hinterlegt.
5. Sollte die Zeile ein global definiertes Label beinhalten, wird dieses zu einem lokalen Label umgeformt.
6. Wenn es sich um eine Zuweisung mittels `EQU` handelt, wird ähnlich zu den lokalen Labeln ein neuer Name generiert und das Paar in einer Map gespeichert.
7. Entsprechend der beiden Maps werden Variablen- und Labelnamen durch ihren synthetischen Partner ersetzt.
8. Ist die Zeile eine Zuweisung mittels `SET`, so wird zuerst geprüft, ob der entsprechende Name bereits im zweiten Schritt dieser Liste gefunden wurde. Wenn das nicht der Fall ist, wird abermals ein neuer Name generiert und das Paar gespeichert.
9. Variablennamen in der Map aus generierten Namen von `SET`-Statements werden ausgetauscht.
10. Die Zeile wird dem Ergebnisvektor angehängt

**Quellcode 5.8** Schleife zur Generierung synthetischer Namen

```
loop {
    let label_char = char::from_u32(
        (*generated_label_count / 10000) as u32
            + 'A' as u32)
        .unwrap();
    let label_num = (*generated_label_count % 10000) as u32;

    if label_char.eq(&'[') {
        return Err("Exceeded maximum amount of local labels!")
    }
    let new_label = format!("{}{}", label_char, label_num);
    if !taken_names.contains(&new_label) {
        return Ok(new_label)
    }
    *generated_label_count += 1;
}
```

Das Generieren eines Namens geschieht nach dem Ablauf, der in 5.2.3 zu

sehen ist. Er basiert auf einem Vektor vergebener Namen (`taken_names`) und einer Anzahl bereits generierter Namen (`generated_label_count`), die als Methodenparameter übergeben werden. Bei der Implementierung wurde sich dafür entschieden Namen zu generieren, die (gemäß Spezifikation) mit einem Großbuchstaben, bzw. dem „A“ beginnen und von einer bis zu vierstelligen Zahl gefolgt werden.

Sollten auf diese Weise über 10000 Namen generiert werden, wird der ASCII-Wert der den Buchstaben bestimmt, um eins erhöht. Der Zahlwert für den Namen entspricht dem Rest bei einer Division der Anzahl generierter Label durch 10000. Damit kein Name generiert wird, der vom Programmierer zufällig selber benutzt wird, muss sichergestellt werden, dass der generierte Name kein Teil der Liste bereits belegter Namen entspricht. Dieser Prozess der Generierung findet solange statt, bis ein geeigneter Name gefunden wurde, wobei die Anzahl generierter Namen mit jedem Schleifendurchlauf entsprechend inkrementiert wird.

Diese Implementierung ist insofern beschränkt, dass maximal $26 * 10000 = 260000$ Namen generiert werden können. Sollte der Quellcode nutzerdefinierte Label enthalten, die diesem Schema entsprechen, ist die Zahl entsprechend geringer. Allerdings gilt diese Umsetzung als genügend, da der Speicher des Intel 8080 auf 16384 Byte beschränkt ist und deshalb ohnehin keine solche Menge an Namen speichern könnte.

### 5.2.4 Mapping von Bytes

Neben dem Übersetzen von Assembly- in Bytecode bietet der Assembler eine weitere Funktionalität: Das Mappen der entstehenden Bytes auf ihre zugehörige Zeile. Diese dient ausschließlich dem Frontend, in dem bei einer schrittweisen Programmausführung angezeigt werden soll, in welcher Zeile sich das Programm momentan befindet. Weil die spätere Ausführung mittels Emulator auf Bytes basiert und nicht zeilenweise geschieht, bedarf es eben dieser Funktionalität, da ein triviales „führe Zeile X aus“ nicht möglich ist. Das Ergebnis der hier vorgestellten Methodik ist vom Typ `HashMap<u16, usize>`, wobei jeweils der Index eines Byte als Schlüssel dient, der auf den Index der Zeile verweist, in der sich der ursprüngliche Code befindet.

Damit es möglich ist eine Beziehung zwischen dem aktuellen Byte und seiner zugehörigen Zeile herzustellen, muss auf zwei Dinge besonders geachtet werden:

- Leerzeilen und konditionale Strukturen, die als `false` evaluiert sind, erhöhen den Zeilenindex unabhängig vom Byteindex
- Referenzen von Makros zeigen auf die Stelle, an der das Makro definiert wurde

Generell verhält sich die Methode ähnlich der anderen Schritte in der Vorverarbeitung, nämlich so, dass sie den Code Zeile für Zeile betrachtet, nachdem sie mittels dem in Kapitel 5.2.2 erläuterten Verfahren Variablen ersetzt hat. Anstatt dass nur der Zeileninhalt betrachtet wird, muss der Code an dieser Stelle aufgezählt werden, wodurch der Index der Zeile zur Verfügung steht:

```
for (index, line) in code.iter().enumerate() { }
```

Weil der Code an dieser Stelle noch keiner Vorverarbeitung unterzogen wurde (diese würde Zeilenindizes verfälschen) müssen Deklarationen von Labels, die an dieser Stelle nicht evaluiert werden, bewusst entfernt, bzw. übergangen werden. Außerdem ist es relevant ob sich die aktuelle Zeile innerhalb eines nicht auszuführenden If-Blocks oder einer Makrodefinition befindet. Diese Informationen werden entsprechend dem bereits mehrfach erläuterten Verfahren mittels boolscher Variable und einfacher Überprüfung des String-Inhalts bestimmt.

Für jede Zeile, auf die eine der beiden Eigenschaften zutrifft, genügt es den Zeilenindex um eins zu erhöhen und zur nächsten Zeile überzugehen. Ansonsten wird versucht die Zeile in Opcode und gegebenenfalls den Rest aufzuspalten. Für den nachfolgenden Code ist ausschließlich der Opcode interessant. Die Implementierung dieses Verhaltens lässt sich in Rust, ähnlich anderer Sprachen, mittels einer bedingten Zuweisung implementieren. Sie ist im Quellcode in 5.2.4 zu sehen.

**Quellcode 5.9** Bedingte Zuweisung

```
let operand: &str = if line.contains(" ") {
    line.split_once(" ").unwrap().0
} else {
    &line
};
```

Anschließend wird geprüft, ob der Inhalt der Variable `operand` in einem von drei Vektoren vorhanden ist. Diese Vektoren beinhalten alle Opcodes des Intel 8080, sortiert nach deren Anzahl Bytes. Zeilen, die keinen Opcode beinhalten, werden durch diese Implementierung automatisch gefiltert: Beispielweise wird `var SET 5` zu „var“, was keinem Opcode entspricht und somit übergangen wird.

Sofern es sich beim Variableninhalt tatsächlich um einen Opcode handelt wird die Map, die das Ergebnis repräsentiert erweitert. Das geschieht nach dem Muster, dass je nachdem wie vielen Byte der Opcode entspricht, ein bis drei Einträge nach dem Muster `(byte_index, line_index)` in der Map erstellt werden. Dabei wird der erste Index entsprechend erhöht, sollte es sich um mehrere Byte handeln. Sobald die Einträge in der Map erstellt wurden, werden die beiden Indizes um die entsprechenden Werte erhöht und zur

nächsten Zeile übergegangen.

Während bisherige Implementierung den Großteil aller auftretenden Fälle abdeckt fehlt ein entscheidender und bereits zu Beginn des Kapitels erwähnter Teil: Das Mappen von Zeilen innerhalb von Makros. Dafür wird eine Map vom Typ `HashMap<String, HashMap<u16, usize>>` benötigt. In dieser verweist der Name des Makros auf seine zugehörige Map. In letzterer befinden sich die Indizes der einzelnen Byte, die auf die jeweilige Zeile verweisen.

Für jede Zeile des eigentlichen Codes kann dann, sofern es sich um die Referenz auf ein Makro handelt, der entsprechende Eintrag an das finale Ergebnis angehängt werden. Dabei muss darauf geachtet werden, dass die Byteindizes abhängig vom Index des letzten Bytes vor der Referenz bestimmt werden. Die Zeilen hingegen sind statisch, weil die Definition des Makros nur an einer bestimmten Stelle steht.

Zur Bestimmung der eigentlichen Maps wurde ein rekursiver Methodenaufruf verwendet. Das ist möglich weil der Intel 8080 keine verschachtelten Makrodeklarationen erlaubt, weshalb es keiner Abbruchbedingung bedarf und höchstens ein Rekursionsabstieg stattfindet. Dafür werden die Instruktionen des Makros in einem separaten Vektor an die Methode übergeben, die den Inhalt nun als eigenständiges Programm versteht. Dementsprechend beginnen die Werte der so erstellten Map `local_map` bei 0. Mit dem folgenden Code werden die Zeilenindizes so korrigiert, dass sie dem „wirklichen“ Programm entsprechen:

```
*value += line_index + 1
```

Die so entstehende Map kann an die Map aller Makros angehängt und bei Bedarf entsprechend referenziert werden. Im Fall einer Referenz wird für jeden Eintrag der Makro-Map ein neuer Eintrag im Endergebnis erstellt. Die Einträge entsprechen dem Muster `(local_byte + byte_index, *line)`, bei dem jeweils der Byteindex um den aktuellen Wert erweitert, die Zeile als solche übernommen wird.

### 5.2.5 Parser

Für die Verarbeitung arithmetischer Ausdrücke haben wir einen Präzedenzparser geschrieben, der Edsger Dijkstra's Rangierbahnhof-Algorithmus implementiert. Im folgenden ist eine formale Grammatik in Erweiterte Backus-Naur-Form (EBNF) gegeben, welche die Sprache der akzeptierten arithmetischen Ausdrücke darstellt.

In der Grammatik ist Operator-Präzedenz bereits sichtbar — die Präzedenz steigt von oben nach unten.

⟨*expression*⟩ ::= ⟨*disjunctive*⟩

⟨*disjunctive*⟩ ::= ⟨*conjunctive*⟩ ( ( 'OR' | 'XOR' ) ⟨*conjunctive*⟩ ) *

⟨*conjunctive*⟩ ::= ⟨*additive*⟩ ( 'AND' ⟨*additive*⟩ ) *

⟨*additive*⟩ ::= ⟨*multiplicative*⟩ ( ( '+' | '-' ) ⟨*multiplicative*⟩ ) *

⟨*multiplicative*⟩ ::= ⟨*primary*⟩ ( ( '*' | '/' | 'MOD' | 'SHL' ) ⟨*primary*⟩ ) *

⟨*primary*⟩ ::= '(' ⟨*expression*⟩ ')' | NUMBER | ( '-' | 'NOT' ) ⟨*primary*⟩

### Tokenizer

Der Tokenizer ist als Iterator über den Eingabestring implementiert. Der Eingabestring wird Buchstabe für Buchstabe konsumiert, und entsprechend in sogenannte Tokens umgewandelt. Dieser Tokenstream dient dann als Eingabe für den Parsing-Algorithmus. Abschnitt 5.2.5 zeigt das Token-Enum und die dazugehörigen Operator-Enums.

**Quellcode 5.10** Token- und Operator-Enums

```
pub enum Token {
    Number(i32),
    Operator(Op),
    Parenthesis(char),
    Unary(UnOp)
}

pub enum Op {
    Add, Sub, Mul, // ...
}

pub enum UnOp {
    Minus, Not
}
```

Für die Operatoren ist eine Funktion implementiert, die die entsprechende Präzedenz zurückgibt und eine Funktion, die den Operator auf 2 Parameter anwendet.

### Rangierbahnhof-Algorithmus

Dijkstra's Rangierbahnhof-Algorithmus ist ein Algorithmus zur Umwandlung eines arithmetischen Ausdrucks in Postfixnotation oder in einen Abstract-Syntax-Tree. Er kann leicht angepasst werden um stattdessen das Ergebnis eines solchen Ausdrucks zu berechnen — siehe Algorithmus 1 für entsprechenden Pseudocode.

Der Algorithmus verwendet 2 Stacks, wobei auf einem (im Pseudocode einfach Stack genannt) Operatoren und Klammern abgelegt werden und auf dem anderen (im Pseudocode Ablage genannt) werden Zwischenergebnisse abgelegt. Der Algorithmus geht die Tokens nacheinander durch, wobei 3 Tokentypen unterschieden werden. Bei einer Zahl, wird diese sofort auf die Ablage gelegt. Bei einem Operator werden solange die Operatoren auf dem Stack abgearbeitet, bis der aktuelle Token höhere Präzedenz hat, als die Stackspitze. Anschließend wird der Token auf den Stack gelegt. Im klassischen Algorithmus werden Operatoren abgearbeitet, indem sie auf die Ablage verschoben werden. Dadurch entsteht auf dieser der Ausdruck in Postfixnotation. In unserem Fall jedoch, wenden wir die Operatoren sofort an, da wir auf der Ablage das Ergebnis des Ausdrucks erzeugen wollen. Wie in der Postfixnotation, sind die obersten 2 Elemente der Ablage stets die Parameter der als nächstes auszuführenden Operation. Folglich nehmen wir für jeden Operator der vom Stack entfernt wird diese 2 Elemente von der Ablage, wenden den Operator auf sie an (bspw. Berechnung der Summe) und legen das Ergebnis der Rechnung auf die Ablage. Der dritte Tokentyp ist die Klammer. Eine öffnende Klammer wird auf den Stack gelegt, wodurch erkennbar wird, welche Operatoren innerhalb der Klammer sind. Wenn die dazugehörige schließende Klammer vom Tokenstream gelesen wird, werden solange Operatoren vom Stack angewendet, bis die öffnende Klammer die Stackspitze bildet. Diese wird vom Stack entfernt und der nächste Token kann gelesen werden. Wenn alle Tokens abgearbeitet sind und der Stack vollständig geleert ist, verbleibt nur eine Zahl auf der Ablage: das Ergebnis des Ausdrucks.

**Algorithmus 1** Angepasster Rangierbahnhof-Algorithmus

```
Stack und Ablage sind leere Stapel
for all t ∈ tokens do
    if t ist Zahl then
        Lege t in Ablage
    else if t ist Operator then
        while Stack ist nicht leer do
            if Präzedenz(t) ist kleiner-gleich Präzedenz(Stackspitze) then
                Wende die Operation an der Stackspitze
                  auf die obersten 2 Elemente der Ablage an
                Ersetze oberste 2 Elemente mit Ergebnis
                Entferne Stackspitze von Stack
            else
                Gehe zum Schleifenende
            end if
        end while
        Lege t auf Stack
    else if t ist öffnende-Klammer then
        Lege t auf Stack
    else if t ist schließende-Klammer then
        while Stackspitze ist nicht öffnende-Klammer do
            Wende die Operation an der Stackspitze
              auf die obersten 2 Elemente der Ablage an
            Ersetze oberste 2 Elemente mit Ergebnis
            Entferne Stackspitze von Stack
        end while
    end if
end for
while Stack ist nicht leer do
    Wende die Operation an der Stackspitze
      auf die obersten 2 Elemente der Ablage an
    Ersetze oberste 2 Elemente mit Ergebnis
    Entferne Stackspitze von Stack
end while
return letztes Element in Ablage
```

### Komplexität

Unser Parser, inklusive Tokenizer, hat lineare Zeitkomplexität ($\mathcal{O}(n)$) und muss die Eingabe nur einmal durchlaufen. Die Reduktion auf einen Durchlauf ist durch Implementierung des Tokenizers als Iterator möglich, sonst müssten die Eingabe einmal von Tokenizer und einmal vom Parser durchlaufen werden.

## 5.3 WebAssembly API

Die Schnittstelle zwischen Frontend und Emulator wird mithilfe einer Wasm Application Programming Interface (API) umgesetzt. Hierzu wurde eine Rust-Bibliothek namens *wasm-bindgen* genutzt, die es erlaubt Methoden und Strukturen zwischen JavaScript und Rust zu teilen.

Um in Rust eine Methode oder Struktur zugänglich zu machen, ist es lediglich erforderlich das Attribut `#[wasm_bindgen]` hinzuzufügen. In der Datei *lib.rs* werden damit folgende zustandslose Helfermethoden definiert, die von der Web-Applikation aufgerufen werden können.

#### assemble

Die Funktion `assemble` nimmt Source-Code entgegen, assembled ihn, und liefert eine Liste von Bytes zurück.

#### get_linemap

`get_linemap` erstellt ein Mapping von Program Counter auf Quellcodezeile, das von der Webapplikation dazu genutzt wird, um die aktuell ausgeführte Codezeile rot zu hinterlegen.

#### disassemble

Diese Funktion ruft den Disassembler auf und wandelt eine Liste von Bytes wieder in Assembly-Code um.

#### createEmulator

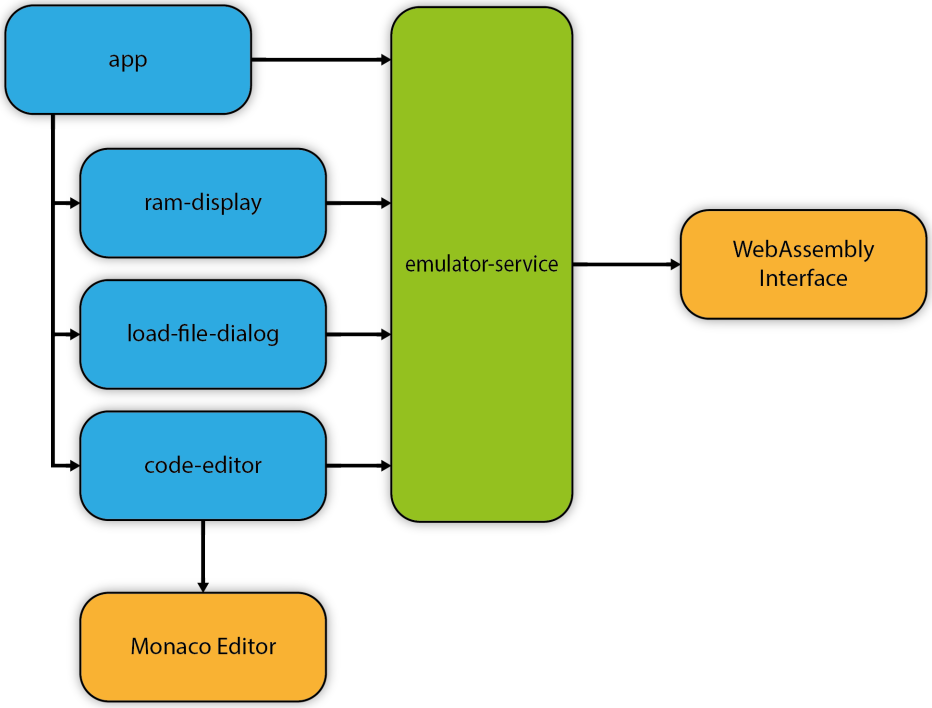
`createEmulator` erstellt eine neue Instanz des Emulators, setzt den Speicher auf den übergebenen Wert, und gibt die Instanz zurück.

Neben diesen Helfermethoden wurden auch die Felder der Emulator-Struktur, Program Counter und Stack Pointer, zugänglich gemacht, sowie die Methode für das Ausführen der nächsten Instruktion, `execute_next()`.

Die Schnittstelle sorgt allerdings nicht nur dafür, dass Rust-Code von JavaScript aus ausführbar ist. Auch die JavaScript-Funktion `console.log` wird für den Rust-Code zugänglich gemacht, um so Fehlermeldungen anzuzeigen:

```rust
#[wasm_bindgen]
extern "C" {
    // JS: console.log(msg)
    #[wasm_bindgen(js_namespace = console)]
    fn log(msg: &str);
}
```

Abbildung 5.1: Architektur der Webanwendung




## 5.4 Frontend

Die Implementierung der Web-Applikation erfolgt in Form einer Angular-Anwendung. Dazu wurden mehrere Komponenten und ein Service erstellt, die die einzelnen Funktionen der Anwendung abdecken.

In Abbildung 5.1 ist die Architektur der Applikation veranschaulicht. Die blauen Objekte stellen Komponenten dar, das grüne Objekt den Service und die orangenen Objekte externe Abhängigkeiten.

### 5.4.1 Komponenten

#### App-Component

Die App-Component bildet den zentralen Zugangspunkt der meisten Angular-Applikationen. Von hier aus werden die anderen Komponenten der Anwendung eingebunden und miteinander verknüpft. Das HTML-Template der App-Component umfasst in diesem Fall das komplette Skelett der Anwendung, welches eine Titelleiste und die Einteilung in eine linke und eine rechte Hälfte definiert. In dieses Skelett werden dann die anderen Komponenten eingebunden.

#### RAM-Display

Die RAM-Display-Component stellt den Arbeitsspeicher des Emulators in Form einer 16x16 Matrix dar. Genutzt wird dafür eine einfache HTML-Tabelle, die mithilfe von Stylesheets entsprechend angepasst wird. Der Zugriff auf den Arbeitsspeicher erfolgt durch direkten Zugriff auf den Speicherbereich der Wasm-Objekte im Browser über einen Pointer.

Neben der Arbeitsspeicher-Matrix kümmert sich diese Komponente auch um die Anzeige der verschiedenen Register.

#### Load-File-Dialog

Diese Komponente wird nicht von Anfang an auf der Seite eingebunden, sondern wird erst dann in Form eines modalen Dialogs angezeigt, wenn der Nutzer auf den *Load*-Button in der Aktionsleiste drückt. Der Dialog zeigt dann ein simples HTML-Formular an, das mit einem `<input type="file"/>` eine Dateiauswahl ermöglicht.

#### Code-Editor

Die Code-Editor-Component ermöglicht der Anwendung die Nutzung der Monaco Editor Bibliothek. Dazu erfolgt eine umfangreiche Konfiguration des Editors, um beispielsweise die Tokens und das Farbschema für das Syntaxhighlighting festzulegen oder um die Autovervollständigung zu ermöglichen.

### 5.4.2 Emulator-Service

Alle Komponenten greifen über das Dependency-Injection-Framework von Angular auf den sogenannten Emulator-Service zu. Dieser Service verwaltet den Emulator, der mithilfe des in Kapitel 5.3 definierten Wasm-Interfaces erstellt wird.

Der Service bietet den Komponenten verschiedene Funktionen an, wie zum Beispiel, Quellcode zu assemblen, was vom *Assemble*-Button in der Aktionsleiste genutzt wird. Zur Ausführung von Programmen wird im Service außerdem die Hauptschleife des Emulators implementiert.

```
this._loop = window.setInterval(() => {
  for (let i = 0; i < this._stepsPerInterval; i++) {
    this.cpuStep();
  }
}, this._interval);
```

Statt einer `while`-Schleife kommt hier die Funktion `window.setInterval()` des Browsers zum Einsatz. Diese Funktion führt eine gegebene andere Funk-

tion periodisch immer wieder mit einem bestimmten Intervall auf. Dies ermöglicht der Webanwendung trotz der „Endlossschleife“ responsiv zu bleiben. Da das Intervall nicht beliebig klein sein kann, gibt es außerdem die Möglichkeit, in jedem Schleifendurchlauf mehrere Prozessorschritte durchzuführen. So kann der Emulator trotzdem so schnell laufen wie es nötig ist.

Die `cpuStep()`-Methode arbeitet nun verschiedene Schritte ab: Hauptaufgabe der Methode ist das Ausführen von `emulator.execute_next()`, was dafür sorgt, dass der Emulator die nächste Instruktion ausführt und den Program Counter gegebenenfalls erhöht. Nach dem Ausführen des Emulatorschritts wird der Arbeitsspeicher mit dem vorherigen Zustand verglichen, um festzustellen ob es eine Änderung gab. Dies wird für das Einfärben der entsprechenden Stelle in der RAM-Matrix genutzt.

Eine weitere Aufgabe der Methode ist die Emulation von sogenannten *CP/M Syscalls*.

### 5.4.3 CP/M Emulation

Um den Emulator zu testen, wurden einige frei verfügbare Programme genutzt. Diese wurden allerdings für das CP/M-Betriebssystem geschrieben, welches ab 1974 unter anderem für den Intel 8080 entwickelt wurde und auf diversen Rechnern für Endanwender zum Einsatz kam.

Ein wesentliches Feature, was diese Programme nutzen, sind die Basic Disk Operating System (BDOS) Syscalls, die auf einem CP/M System das Ausgeben von Text in einem Terminal ermöglichen [2].

```
LD DE   ;parameter
LD C    ;function
CALL 5  ;syscall routine
```

Für den Aufruf eines Syscalls muss die Funktion, die aufgerufen wird, in das Register C geschrieben werden. Mögliche Parameter können im Register DE abgelegt werden. Der Syscall kann nun durch einen Sprung auf Adresse 5 ausgeführt werden.

Relevant für die Programme, die zum Testen genutzt werden, sind lediglich die Syscalls 2 (Zeichen-Ausgabe) und 9 (String-Ausgabe). Die Emulation dieser Syscalls ist vergleichsweise simpel:

```
if (this._cpmMode) {
  // Patch memory for BDOS syscalls
  this._emulatorMemory[5] = 0xC9; // RET
  this._emulator.pc = 0x100;
}
```

Beim Start eines Programms, was im CP/M Modus ausgeführt wird, wird der Speicher an der Stelle 5 mit einer Return-Instruktion überschrieben. Das sorgt dafür, dass das Programm bei jedem Syscall sofort wieder an die ursprüngliche Stelle zurückspringt. Zusätzlich wird der Program Counter beim Start direkt auf 0x100 gesetzt, da CP/M davon ausgeht, dass Nutzerprogramme an dieser Stelle beginnen.

Im `cpuStep()` wird nun folgender Code ausgeführt:

```
if (this.emulator.pc == 0x05) {
    // Emulate CP/M BDOS syscalls
    const syscall = registers.c;
    switch(syscall) {
        case 2: {
            // BDOS function 2 (C_WRITE) - Console output
            const char = registers.e;
            console.log(String.fromCharCode(char));
            break;
        }
        case 9: {
            // BDOS function 9 (C_WRITESTR) - Output string
            let address = largeRegisters.d;
            let currentChar = "";
            let fullString = "";
            while (currentChar != "$") {
                fullString += currentChar;
                const charCode = emulatorMemory[address++];
                currentChar = String.fromCharCode(charCode);
            }
            console.log(fullString);
            break;
        }
    }
}
```

Sobald der Program Counter den Wert 5 hat, das Programm also an die Syscall Adresse gesprungen ist, wird der entsprechende Syscall emuliert. Beim Syscall 2 wird das Zeichen, was sich in Register E befindet, ausgegeben. Syscall 9 gibt den String, der sich an der Speicheradresse aus Register DE befindet, so lange aus, bis das Zeichen „$“ auftritt.

# 6 Auswertung

## 6.1 Herausforderungen

Zum besseren Verständnis der nachfolgenden Bewertung der Ergebnisse werden noch einmal die Herausforderungen betrachtet. Dabei geht es um Schwierigkeiten und Probleme die während der Entwicklung aufgetreten sind. Sie sind maßgeblich für die Evaluierung des gewählten Vorgehens und sollen eventuelle Hindernisse für zukünftige Arbeiten auf diesem Gebiet aufzeigen.

### 6.1.1 Emulator

Der Großteil der Entwicklung des Emulators bildet die Implementierung der einzelnen Instruktionen. Diese sind alle relativ simpel und daher einfach umzusetzen. Jedoch ist es leicht möglich, dass sich kleine Fehler einschleichen, die von den Testfällen verpasst werden. Solche Fehler fallen ggf. erst zu einem deutlich späteren Zeitpunkt auf und können dann oft schwer auf konkrete Fehlerquellen zurückgeführt werden. Wie solche Fehler in unserem Emulator bemerkt und gefunden wurden wird in Abschnitt 6.1.4 im Detail ausgeführt.

### 6.1.2 Assembler

Beim Assembler haben sich zwei Faktoren besonders bemerkbar gemacht. Der Erste ist, dass es sich um eine **Schnittstelle zum Benutzer** handelt. Sie ermöglicht es beinahe beliebigen Input in Form einer Texteingabe zu übermitteln. Typischerweise bergen Nutzereingaben vor allem in sicherheitskritischen Applikationen, bei denen Anwender möglicherweise unbefugt Daten abfangen oder manipulieren, ein erhöhtes Risiko. Weil der Emulator keine solcher Daten beinhaltet bleibt dieser Aspekt zwar aus, nichtsdestotrotz muss eine korrekte Übersetzung des vom Intel 8080 beschriebenen Assemblercodes in Maschinencode erfolgen.

Grundsätzlich ist das auch möglich, aber es kommt zu möglichen Problemen, wenn es dem Anwender gestattet ist Formate zu benutzen, die zwar einer Form folgen, jedoch nicht eindeutig sind. Dazu gehören beispielsweise

Namen von Variablen oder mathematische Ausdrücke. Es kann zweifelsfrei festgestellt werden, ob diese eine zugelassene Form besitzen, erschweren eine Feststellung der Korrektheit jedoch ungemein, worauf in Kapitel 5.2.3 detailliert eingegangen wurde. Während hier die Verwendung von Regex' hilft, sind nicht alle Eck- und Kantenfälle direkt ersichtlich, die auf jeden Fall in Tests umgesetzt werden müssen, sobald sie identifiziert werden konnten.

Neben dem schwierigen Umgang mit Nutzereingaben war die **Verarbeitung von Pseudo-Instruktionen** eine weitere Herausforderung bezüglich der Umsetzung. Nicht nur sind vor allem diese Grund für die Möglichkeit zur Verwendung von benutzerdefinierten Namen, auch die Reihenfolge der Abarbeitung ist entscheidend. Dabei stachen vor allem Makros heraus, was sich in deren großem Anteil an Kapitel 5.2.2 widerspiegelt. Aufwendig waren hier vor allem die Beachtung verschiedenen Gültigkeitsbereiche und das korrekte Bestimmen von Variablen zu bestimmten Zeitpunkten. Letzteres erschwerte die Modularisierung der Vorverarbeitung maßgeblich, weil mehrere Funktionalitäten, die zwar inhaltlich getrennt werden können, nicht losgelöst voneinander durchführbar sind.

### 6.1.3 WebAssembly-Interface

Bei der Implementierung des Wasm-Interfaces gab es einige Herausforderungen, die aufgrund fehlender Funktionalität der *wasm-bindgen* Bibliothek entstanden.

#### Fehlende Union Types

Die Register der CPU wurden im Rust Code mithilfe von Union Types implementiert, die dafür sorgen, dass die 16-Bit Register und die dazugehörigen 8-Bit Register im gleichen Speicherbereich liegen. Diese Union Types können nicht auf die Wasm-Bindings abgebildet werden

#### Fehlende dynamische Trait Referenzen

Da der Arbeitsspeicher nur durch einen Trait definiert wird, wird die Referenz auf dieses Objekt in der Emulator Struktur durch eine dynamische Box repräsentiert. Diese Referenz kann ebenfalls nicht auf das Wasm-Binding abgebildet werden.

#### Lösung

Beider Probleme konnten durch einen direkten Speicherzugriff im Browser gelöst werden. Mithilfe der `WebAssembly.Memory` Klasse besteht die Möglichkeit, auf den gesamten allokierten Speicher einer Wasm-Instanz zuzugreifen. Über die Wasm-Schnittstelle werden nun lediglich Pointer übertragen,

die auf die entsprechende Stelle der Register und des RAM in diesem Speicher zeigen. Über manuell festgelegte Offsets kann nun der gewünschte Wert direkt aus dem Speicher gelesen werden.

### 6.1.4 Fehlersuche

Ziel dieser Arbeit ist es, die komplette Spezifikation des Intel 8080 zu implementieren. Fehler in der Implementierung von einzelnen Instruktionen zu finden hat sich als besonders schwierig herausgestellt, da diese häufig nur in besonderen Fällen auftreten und sich lediglich dadurch äußern, dass einzelne Flaggen des Prozessors einen falschen Wert haben.

Um dennoch die Fehlerfreiheit zu garantieren, wurden alle Instruktionen mithilfe eines Testprogramms getestet und der Status der CPU mit einem anderen, fehlerfreien Emulator verglichen, der das gleiche Programm ausführt. Sobald der Zustand des Prozessors nach dem Ausführen einer Instruktion nicht mehr identisch ist, ist eine fehlerhafte Implementierung der jeweiligen Instruktion wahrscheinlich.

## 6.2 Prozessor Tests

Um zu überprüfen, ob die Implementierung der Spezifikation des Intel 8080 tatsächlich vollständig gelungen ist, müssen Tests durchgeführt werden. Hierfür kommen frei verfügbare [1] CPU Testprogramme zum Einsatz, die die Funktionsweise verschiedener Instruktionen überprüfen und mit Sollzuständen vergleichen.

Die Ausgabe eines solchen Tests sieht wie folgt aus:

```
8080 instruction exerciser

dad <b,d,h,sp>.............
  PASS! crc is:
14474ba6

aluop nn....................
  PASS! crc is:
9e922f9e

aluop <b,c,d,e,h,l,m,a>.......
  PASS! crc is:
cf762c86

[...]
```

[1] https://altairclone.com/downloads/cpu_tests/

```
stax <b,d>...................
  PASS! crc is:
2b0471e9

Tests complete
```

Es werden unterschiedliche Instruktionen, unter anderem alle Arithmetic Logic Unit (ALU) Operationen, mit vielen verschiedenen Eingabewerten getestet. Der Zustand der CPU wird nach jeder getesteten Instruktion mit einem laufenden Cyclic Redundancy Check (CRC)-Wert verrechnet. Nach jeder Gruppe von Instruktionen wird diese Prüfsumme mit einem vorgegebenen Richtwert verglichen und wenn dieser übereinstimmt, ist der Test bestanden.

Da der Emulator alle Tests erfolgreich besteht, ist davon auszugehen, dass die Spezifikation des Prozessors vollständig erfüllt wurde.

## 6.3 Benchmarks

Eine interessante Kennzahl zur Auswertung des Emulators ist außerdem die mögliche Performance bei der Ausführung von Programmen. Dabei ist einerseits ein Vergleich mit der realen CPU von Interesse. Zusätzlich stellt sich die Frage, inwiefern die Wasm-Schnittstelle und die Ausführung des Codes im Browser einen Einfluss auf die Performance des Emulators hat.

Als Benchmark-Programm kommen hier erneut die Testprogramme aus Kapitel 6.2 zum Einsatz. Die Tests brauchen teilweise mehrere Minuten für einen Durchlauf, wodurch sich ein guter Mittelwert für die Rechenzeit ausrechnen lässt, der frei von zufälligen Schwankungen ist.

Die Ergebnisse der Tests werden in der folgenden Tabelle veranschaulicht.

| Testprogramm | Nativ | Browser | Vergleich |
|---|---|---|---|
| 8080EXM.COM | 314.5s | 593.9s | 188% |
| 8080PRE.COM | 0.17ms | 1.1ms | 647% |
| CPUTEST.COM | 4.1s | 8.7s | 212% |
| TST8080.COM | 0.14ms | 0.7ms | 500% |
| Taktrate | 75MHz | 40MHz | 53% |

Da die beiden Tests `8080PRE.COM` und `TST8080.COM` nur einige Tausend Instruktionen ausführen und demnach nur Bruchteile einer Millisekunde benötigen, sind die Tests hier vergleichsweise ungenau. Bei den anderen beiden Tests, die mehrere Sekunden bis Minuten laufen, ist der Wert allerdings deutlich aussagekräfter. Der Emulator braucht hier im Browser durchschnittlich doppelt so lange einen Test auszuführen wie der direkt auf dem Rechner

ausgeführte Emulator.

Mögliche Gründe hierfür könnte einerseits die geringere Effizienz von Wasm gegenüber nativem Maschinencode sein. Andererseits kann die Implementierung des Loops auf JavaScript-Ebene ein limitierender Faktor sein, da dadurch bei jeder Instruktion von JavaScript auf die Wasm Schnittstelle zugegriffen werden muss, was möglicherweise einen Overhead mit sich bringt.

Die Taktraten des Emulators sind mit 75MHz, beziehungsweise 40MHz im Browser, deutlich schneller als die 2MHz, die der echte Intel 8080 leistet.

Insgesamt kann man also sagen, dass die Performance des Emulators im Browser zwar nur halb so gut ist wie ein nativer Emulator sein könnte, allerdings trotzdem zufriedenstellend ist. Da die mögliche Taktrate von 40MHz immer noch weit über dem Wert des realen Prozessors liegt, ist eine realitätsnahe Emulation mit echter Geschwindigkeit sehr einfach möglich.

## 6.4 Fazit

# 7 Ausblick

Dieses Kapitel behandelt Ideen zur Erweiterung von WIP. Dabei geht es sowohl um mögliche Erweiterungen der bestehenden Spezifikation, als auch Anpassungen, die entsprechend dieser vorgenommen werden sollten, bzw. gemusst hätten, aber primär aus zeitlichen Gründen nicht mehr umzusetzen waren.

## 7.1 Dynamic Recompilation

Bei Dynamic Recompilation handelt es sich um eine Optimierungsstrategie, bei der Maschinencode des emulierten Systems auf Maschinencode des ausführenden Systems übersetzt wird. In unserem Fall würde dabei Intel 8080 Maschinencode zu WebAssembly kompiliert werden. Diese Rekompilation findet zur Laufzeit statt.

Das Ziel ist, Blöcke aus Intel 8080 Assembly zu Blöcken aus nativem Assembly (in unserem Fall wäre das WebAssembly Bytecode) zu kompilieren und diese Blöcke in einem Cache zu speichern. Dies sähe ungefähr so aus[1]:

```
fn run() {
    let addr = getBlock(PC);
    // Execute instructions at addr in memory
    execute(addr);
}

fn getBlock(pc: u16) -> u64 {
    match cache[pc] {
        Some(addr) => addr,
        None => {
            let addr = recompileBlock(pc);
            cache[pc] = Some(addr);
            addr
```

[1]Dieses Beispiel ist orientiert an folgendem Blogpost: `https://wiki.pcsx2.net/PCSX2_Documentation/Introduction_to_Dynamic_Recompilation`

```
        }
    }
}

fn recompileBlock(start_pc: u16) -> u64 {
    // Memory location the block will be written to
    let start_ptr = emitter.get_ptr();
    let pc = start_pc;
    loop {
        match opcodes[pc] {
            // Call emitter functions
            // Branching instructions will break the loop
        }
    }
    start_ptr
}
```

Um den obigen Code zu realisieren, werden 2 Komponenten benötigt: der Code-Emitter und der Cache.

### 7.1.1 Code-Emitter

Der Code-Emitter hat die Funktion, die entsprechenden nativen Instruktionen auf Grundlage der 8080 Opcodes zu generieren. Dafür speichert er eine Adresse (`emitter.get_ptr()` im obigen Code), wobei es sich um die Speicheradresse handelt, an die die nächste Instruktion geschrieben wird.

Der Emitter übersetzt im simpelsten Fall jeden Opcode in eine äquivalente Instruktion in nativen Assembly. Hier können allerdings auch Optimierungen stattfinden, die durch Verwendung eines moderneren Instruktionssatzes oder durch gesammelte Laufzeitinformationen möglich sind. Beispiele hierfür wären:

- Zusammenfassung mehrerer 8/16-Bit Instruktionen als einzelne 32/64-Bit Instruktionen
- Verwendung von Instruktionen, die komplexere Prozesse (bspw. Stringkopie) realisieren
- Schleifenoptimierung

Der Emitter schreibt diese nativen Instruktionen an die gespeicherte Adresse, welche in den Cache zeigt.

### 7.1.2 Cache

Beim bereits erwähnten Cache handelt es sich lediglich um eine Speicherregion, die mit dem PC indiziert werden kann und entweder `None` zurückgibt, wenn der entsprechende Maschinencode noch nicht generiert wurde, oder die nächste Instruktion zurückgibt.

## 7.2 Intel Hex

Momentan gibt der Assembler einen Vektor aus Bytes zurück, welcher in den RAM des Emulators geladen wird, um ein Programm zu laden. Ein alternatives Format für die Assembler-Ausgabe wäre das Intel Hex-Format, ein Dateiformat um Binärdaten im ASCII-Format zu speichern. In Intel Hex repräsentiert jede Zeile einen Datensatz, der eine konsekutive Bytefolge enthält. Ein solcher Datensatz enthält 6 Felder:

1. Satzbeginn: ein ASCII Doppelpunkt am Anfang der Zeile
2. Anzahl an Bytes: Wie viele Datenbytes enthalten sind
3. Speicheradresse: 16-Bit Adresse im Speicher, an der der Datenblock beginnt
4. Datensatztyp: 00..05
5. Daten: n Bytes (als 2n Hex-Zeichen kodiert)
6. Prüfsumme: 2 Hex-Zeichen große Prüfsumme über den Datensatz

Dieses Format zu verwenden hat den Vorteil, leere Regionen zwischen Assembly Instruktionen nicht abspeichern zu müssen. Außerdem sind ROMs für den 8080 häufig in diesem Format vorliegend, daher wäre es sinnvoll solche Dateien einlesen zu können.

## 7.3 Optimierung und Restrukturierung des Assemblers

Die Entwicklung des Assemblers folgte einem iterativ, inkrementellen Ansatz. Dabei wurden schrittweise neue Funktionen hinzugefügt, wobei mit den einfachen Opcodes angefangen, dann zu Labels und später den Pseudo-Instruktionen übergegangen wurde. Was für die Entwicklung unproblematisch war, machte sich jedoch vor allem zum Ende der Implementierung und bei der Dokumentation der Arbeit bemerkbar:

Die Struktur des Assemblers ist suboptimal. Zwar sind viele Funktionen in einzelnen Methoden gekapselt, allerdings gibt es potentielle Fehler in deren aktueller Anordnung. So ist es zum Zeitpunkt der Beendigung der Arbeit

beispielsweise nicht möglich, ein Label als Bedingung für einen If-Block zu verwenden. Zwar könnte der Code assembled werden, allerdings schlägt das Mapping von Zeilen fehl, weil für diese Makros nicht ersetzt werden, was aber notwendig für die Bestimmung von Labels ist.

In diesem Fall wurde sich dazu entschlossen den Fehler in Kauf zu nehmen. Begründet wird dies vor allem mit dem Verhältnis von Zeitaufwand und möglichem Gewinn: Zwar handelt es sich um einen kritischen Fehler, allerdings ist die Verwendung von Labels in konditionalen Anweisungen wenig sinnvoll. Schließlich ist nur das Label, das noch vor dem ersten Opcode steht, sofern es überhaupt existiert, null. In allen anderen Fällen sind Label ungleich null und damit konstante Werte, die die If-Bedingung erfüllen. Es erscheint, obwohl erlaubt, nur wenig sinnvoll Labels an einer solchen Stelle zu positionieren, weshalb der Fehler toleriert wird.

Neben solchen, durch Eckfälle bedingte Fehler, bietet der Assembler auch Raum für Optimierung. Das gilt bezüglich der Zeit- sowie der Codekomplexität. An Stellen, an denen dies bereits zum Zeitpunkt des Arbeitsabschlusses ersichtlich war, wurde das bereits in Kapitel 5.2 erwähnt. Vor allem die zeitliche Abfolge zum Auslesen und Ersetzen bestimmter Zeileninhalte ist verbesserbar. Allerdings ist die Performanz des Assemblers bisher mehr als ausreichend, weswegen der Fokus nicht auf solchen Verbesserungen lag.

## 7.4 Weitere Features für die Webanwendung

Die Webanwendung für den Emulator bietet noch reichlich Spielraum für weitere nützliche Features, die die Entwicklung von Programmen für den Intel 8080 vereinfachen würden.

### 7.4.1 Konfigurationsmenü

Die Anwendung hat bereits jetzt zahlreiche Optionen, wie beispielsweise die CP/M-Emulation, die Geschwindigkeit des Emulators oder die Startadresse, die vom User Interface aus nicht zugänglich sind, sondern lediglich im Code festgelegt werden. Durch Hinzufügen eines dedizierten Konfigurationsmenüs könnten diese Optionen dem Nutzer verfügbar gemacht werden.

### 7.4.2 Peripheriegeräte

Die Emulation des Prozessors umfasst unter anderem die `IN` und `OUT` Instruktionen, mit denen die CPU Daten mit externen Geräten austauschen kann. Für die Webanwendung wäre es denkbar, einige Peripheriegeräte hinzuzufügen. Beispiele wären:

- Schalter

- LED Displays, Matrizen, 7-Segment-Displays
- Keypads
- Datei-Interface
- Audioausgabe
- Logger

Um die Peripheriegeräte anzusteuern, wäre es notwendig, diese einem Port zuzuweisen. Hierfür müsste es die Möglichkeit geben, ein Mapping zu konfigurieren.

### 7.4.3 Breakpoints

Breakpoints sind ein Feature, das nahezu alle integrierten Entwicklungsumgebungen für klassische Programmiersprachen anbieten. Dabei wird die Ausführung des Programms automatisch angehalten, wenn der Prozessor bei einer mit einem Breakpoint markierten Instruktion angelangt. So können Nutzer bestimmte Codebereiche sehr einfach untersuchen und so mögliche Fehler finden.

Die Implementierung würde dank der Monaco Editor Bibliothek vermutlich relativ simpel werden, da Visual Studio Code, was auf dem selben Editor basiert, dies auch unterstützt.

### 7.4.4 Interrupt Status

Aktuell wird der komplette Arbeitsspeicher des Prozessors und alle Register angezeigt. Lediglich der interne Status, ob aktuell Interrupts ausgelöst werden können, ist im User Interface nicht einsehbar. Hierfür könnte noch eine Anzeige hinzugefügt werden.

# A Anhang

Jeglicher Code, der im Verlauf dieser Arbeit entwickelt und vorgestellt wurde, ist öffentlich auf GitHub zugänglich:

`https://github.com/moorts/name_is_wip`

Zum Zeitpunkt der Veröffentlichung wird mittels GitHub Pages die aktuellste Version des Emulators auf einer zusätzlichen Webseite deployed:

`https://moorts.github.io/name_is_wip/`

Die Weboberfläche basiert zu großen Teil auf zwei Open Source Projekten:

- Angular: `https://angular.io/`
- Monaco Editor: `https://microsoft.github.io/monaco-editor/`

# Literatur


[1] Eli Bendersky. *An Intel 8080 assembler and online simulator*. 2020. URL: https://eli.thegreenplace.net/2020/an-intel-8080-assembler-and-online-simulator/.

[2] John Elliott. *BDOS system calls*. 2012. URL: https://www.seasip.info/Cpm/bdos.html.

[3] Andreas Haas u. a. „Bringing the Web up to Speed with WebAssembly“. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, S. 185–200. ISBN: 9781450349888. DOI: 10.1145/3062341.3062363. URL: https://doi.org/10.1145/3062341.3062363.

[4] Denis Howe. *Control Program for Microcomputers*. The Free On-line Dicitonary of Computing. 2003. URL: https://encyclopedia2.thefreedictionary.com/Control+Program+for+Microcomputers.

[5] *intel 8080 Assembly Language Programming Manual*. INTEL CORPORATION. Santa Clara, CA, 2000.

[6] *intel 8080 Microcomputer Systems User's Manual*. INTEL CORPORATION. Santa Clara, CA, 2000.

[7] Steve Klabnik und Carol Nichols. *The Rust Programming Language*. San Francisco, CA: no starch press, 2018.

[8] Martin Malý. *8080js*. 2019. URL: https://github.com/maly/8080js.

[9] Aurélien Richez. *Intel 8080 emu*. 2018. URL: https://docs.rs/crate/intel-8080-emu/latest.

[10] *WebAssembly Core Specification*. https://webassembly.github.io/spec/core/_download/WebAssembly.pdf. W3C, 5. Dez. 2019. URL: https://www.w3.org/TR/wasm-core-1/.

[11] *Rust std library documentation*. https://doc.rust-lang.org/std/index.html. Accessed: 2022-03-15. 2022.

[12] superzazu. *8080*. 2020. URL: https://github.com/superzazu/8080.

[13] superzazu. *Space Invaders*. 2021. URL: https://github.com/superzazu/invaders.

[14] Stefan Tramm. *Intel 8080 Cpu Emulator*. 2010. URL: https://www.tramm.li/i8080/index.html.