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

Inhaltsverzeichnis

Tabellenverzeichnis

Quellcode Verzeichnis

1 Einleitung

2 Grundlagen

2.1 Entwicklungsumgebung

2.2 WebAssembly

"WebAssembly (Wasm) is a safe, portable, low-level code format designed for efficient execution and compact representation"[5]. 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[1]. 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.

2.3 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 ist Rust gut für Wasm-Anwendungen geeignet, da gute Dokumentation existiert[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

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

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

[4] und der Dokumentation der Standard-Library [6].

2.3.1 Traits

2.3.2 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).

2.3.3 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).

2.4 Intel 8080

Das Folgende Kapitel ist eine allgemeine Übersicht über die Architektur und Funktionsweise einer Intel 8080 CPU. Die Informationen sind dem offiziellen Datenblatt[3] und dem offizielen Programmierhandbuch[2] 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 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.

2.4.1 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 Sta-

tus Word), die im Bezug auf bestimmte Instruktionen verwendet wird, bezieht sich auf den Akkumulator kombiniert mit den Flags (siehe unten).

2.4.2 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 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 Übertrag im vierten Bit

2.4.3 Assembly

Wie jede CPU ist der Intel 8080 in der Lage Maschinensprache auszuführen. Programme in Maschinensprache sind für Menschen jedoch 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)

[3] flüchtiger 1-Bit Datenspeicher

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).

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 2.4.2 für Details). Tabelle 2.5 zeigt einige arithmetische Anweisungen mit einer kurzen Erklärung.

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

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 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.

2.4.4 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.

2.4.5 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.

2.5 Angular

Angular ist ein quelloffenes TypeScript Framework zur Entwicklung von Single-Page-Applikationen im Web. Eine sogenannte 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 des Emulators von Bedeutung sind.

2.5.1 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-Component mithilfe von TypeScript definiert. Eine Component 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 bestimmt, wie die Component 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 Component sieht folgendermaßen aus:

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

Dargestellt wird die Component 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 Component definierte Methode `onButtonClicked()` gebunden, die nun bei jedem Klick auf den Knopf aufgerufen wird.

2.5.2 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 Components 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 Components, die den Service nutzen, die selbe Instanz teilen.

Um den Service zu nutzen muss eine Component 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 Component 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[3] 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 Beitrag

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

Dadurch, dass unser Emulator nach Web-Assembly kompiliert wird, läuft der Emulator nativ im Browser des Clienten. Durch eine explizit definierte Schnittstelle, kann der Emulator mittels JavaScript Code bedient werden.

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

In Kapitel 4 wird unser Design für den Emulator, die API und das Frontend erläutert. Darauf folgend werden in Kapitel 5

4 Design

4.1 Emulator

4.1.1 Zentrale Struktur

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

Quellcode 4.1: Zentrale Emulator Struktur

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

Diese Komponenten wurden in Kapitel 2 bereits erklärt: `pc` und `sp` sind 2 16-Bit-Zahlen, die den Program Counter und den Stack Pointer 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 folgt ein Boolean, die aussagt ob der Emulator am Laufen ist und der Boolean die anzeigt 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 imple-

mentiert 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.

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.

4.2 Assembler

4.3 Disassembler

4.4 WebAssembly API

4.5 Frontend

Die Webanwendung, mit der Nutzer den Emulator schließlich nutzen können, stellt zwei wesentliche Funktionalitäten zur Verfügung:

4.5.1 Code Editor

Mit einem eingebauten Code Editor können Nutzer direkt in der Webapplikation eigene Assembly-Programme schreiben und direkt ausführen, ohne

Abbildung 4.1: Code-Editor der Webanwendung

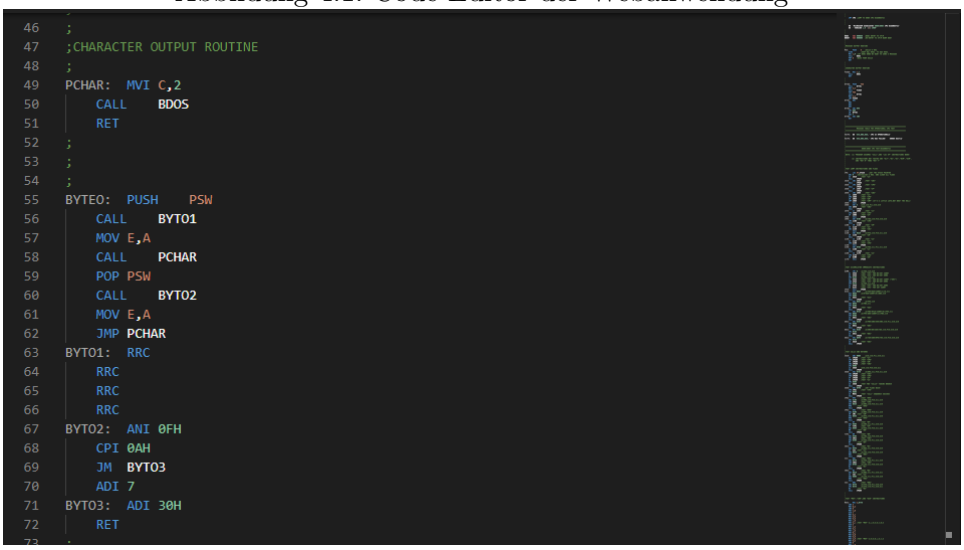
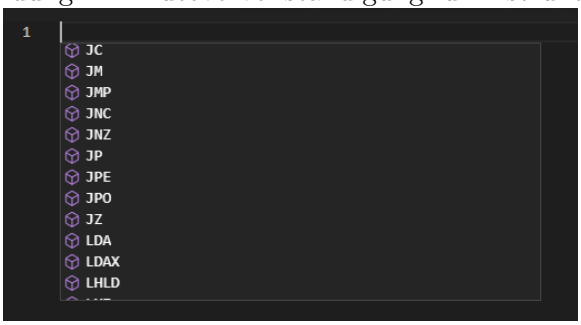


Abbildung 4.2: Autovervollständigung für Instruktionen

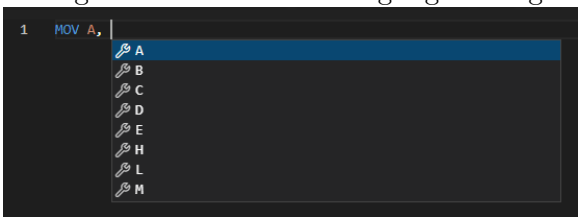


beispielsweise vorher selbst den Code assemblen zu müssen und dann manuell in den Emulator zu laden.

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 Verfügugung 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, der ebenfalls von Microsoft entwickelt wird.

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.

Abbildung 4.3: Autovervollständigung für Argumente

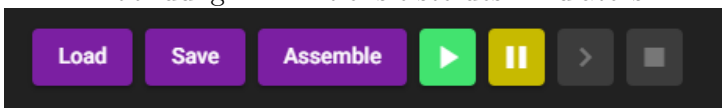


In den Abbildungen 4.2 und 4.3 sieht man außerdem, wie die Autovervollständigung des Code-Editors funktioniert. Basierend auf der Eingabe des Nutzers und der Position im Code wird automatisch erkannt, ob eine Instruktion oder ein Argument vorgeschlagen wird und welche in Frage kommen.

4.5.2 Emulator-Zustand

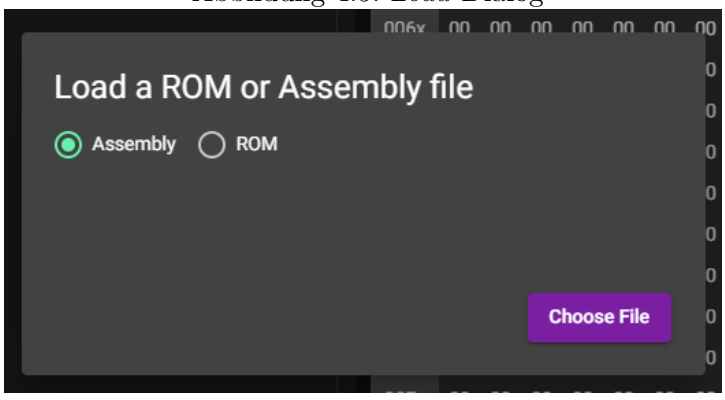
Hat der Nutzer nun ein Assembly-Programm mithilfe des Code-Editors erstellt, kann er es auch ausführen lassen. Hierfür wird der in Rust entwickelte Emulator genutzt, der mithilfe der WebAssembly-Schnittstelle in die Webapplikation eingebunden wird. Die Interaktion mit dem Emulator erfolgt mithilfe verschiedener Bedienelemente in einer Aktionsleiste am oberen Rand der Anwendung (siehe Abbildung 4.4).

Abbildung 4.4: Aktionsleiste des Emulators



Die ersten beiden Schaltflächen der Aktionsleiste, *Load* und *Save*, ermöglichen dem Nutzer, Assemblycode aus Dateien auf seinem Endgerät zu laden

Abbildung 4.5: *Load*-Dialog



oder sie dort zu speichern. *Load* kann außerdem auch bereits assemblierten Code direkt in den Speicher des Emulators laden. Die Auswahl erfolgt mithilfe eines eigenen Auswahldialogs, der in Abbildung 4.5 zu sehen ist.

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.

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 beispielhafter Code-Beispiele illustriert wird. Generell gilt, dass alle Zeilen des Programmcodes vor ihrer Behandlung durch eine Methode mittels `.trim()` zugeschnitten und gegebenenfalls übergangen werden, 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 eine 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 (mittels Regex) entfernt werden. Dabei ist zu beachten, dass nur der Text der Kommentare 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.

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

```
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)
}
```

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 letztlich 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, 2.4.3) deklariert ist, werden durch die Vorverarbeitung mittels Präprozessor (Zeile 2) nicht entfernt, weshalb diese Zeilen noch einmal gesondert übergangen werden müssen. Die Origindeklarationen 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 Listing 5.2.1 um folgende Schleifen erweitert:

```
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 `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. Es sei zu beachten, 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).

Wegen dieser Erweiterung werden die generierten Bytes im Listing 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 eingegebenen Code (ohne Kommentare) auf die Zeilen reduziert, die in den eigentlichen Bytecode übersetzt werden. Dazu zählen unter anderem das Ersetzen von Labels oder auch If-Verzweigungen. Im Quellcode wird diese Aufgabe von der Klasse `preprocessor.rs` übernommen. Dabei handelt es sich um eine reine Konstruktion innerhalb des Assemblers, die außerhalb der Dokumentation beziehungsweise dem eigentlichen Aufbau des originalen Intel 8080s existiert.

Neben der Vorverarbeitung bietet die Hilfsklasse eine Methode zum Mappen von Zeilen an, die der Assembler als eine Art Adapter nach außen hin bietet. 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 einzelne Methoden realisiert) fasst der Präprozessor in einer einzelnen, öffentlichen Methode `get_preprocessed_code(code: &Vec<String>)` zusammen, die dem Assembler zur Verfügung gestellt wird. Man beachte das der Parametertyp einer Referenz auf den des im Assembler-Struct gespeicherten Codes entspricht.

Bestimmung des Programmendes

Weil dieser Schritt eindeutig und essentiell bezüglich der Korrektheit eines Assembler-Programms ist, ist es der erste Schritt in der Vorverarbeitung.

```
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
```

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.

Das Listing 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 Absenz 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 (An dieser Stelle dürfen keine Kommentare im Code vorkommen), was vom letzten If-Statement gedeckt wird, welches mit einem `false` antwortet, sobald ein END gefunden wurde, die aktuell betrachtete 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 im nächsten Schritt Variablen durch ihre konstanten Werte zu ersetzen. Entsprechend dem besonderen Verhalten von

Makros im Bezug auf Variablen (siehe Kapitel 2.4.3) sind zum aktuellen Zeitpunkt bereits alle Werte bestimmbar.

Auch für diesen Schritt wird einmal der Code einmal Zeile für Zeile 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:

```
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())
    );
}
```

Es sei zu beachten, dass eine Variable in vier unterschiedlichen Konstellationen auftreten kann. Im obigen Ausschnitt sind lediglich die Fälle behandelt, dass eine Variable entweder frei im Programm steht oder am Ende einer Zeile. Das vorangestellte und folgende Leerzeichen stellt sicher, dass nur Variablen ersetzt werden, deren Name vollständig entsprechend ist. Sollten diese Leerzeichen fehlen würde folgendes Beispiel nicht erwartungsgemäß funktionieren: Der Entwickler deklariert eine Variable namens „Foo“ und eine zweite Variable namens „FooBa“. In dem Fall, dass Foo als erstes in der Map `set_assignments` auftaucht, wird auch ein Aufkommen von FooBa durch Foo ersetzt, weil nicht überprüft werden würde, ob sich die beiden Namen vollständig entsprechen. Die übrigen Fälle, die hier fehlen, funktionieren analog, allerdings mit dem Unterschied, dass auch unmittelbar vor einer Variablen ein Komma stehen darf, auf das entsprechend geprüft wird.

Für den Fall, dass sich der hintere Teil zweier Namen entspricht wird entsprechend der zwei vorangehenden Zeilen Code das Ende überprüft. Der möglicherweise intuitive Ansatz eines Entwicklers zur schöneren Implementierung mittels Regex ist an dieser Stelle nicht möglich, weil das entsprechende Crate in Rust zum Zeitpunkt der Arbeit nicht den, zur Verbesserung notwendigen, Lookahead und -behind anbietet. Diese sind notwendig da ansonsten möglicherweise wichtige Kommata und Leerzeichen verschwinden könnten.

Neben dem Ersetzen von Variablen gilt es die entsprechenden Werte auszulesen, was mit einer simplen Überprüfung nach den entsprechenden Pseudo-Befehlen geschieht. Sofern ein solcher vorhanden ist, wird die Zeile aufge-

teilt 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 der Name 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 bei `EQU` nicht mitaufgenommen. Letzteres funktioniert beinah analog wobei der einzige Unterschied zu `SET` ist, dass in dem Fall, dass ein Variablenname bereits in der Map liegt der zugehörige Wert nicht überschrieben, sondern eine Fehlermeldung propagiert wird.

Bestimmung von Labels

Im nächsten Schritt sollen die im Kapitel **??** vorgestellten Labels ausgelesen werden. Weil diese 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 sind an dieser Stelle einmal alle Opcodes, in Abhängigkeit von ihrer produzierten Anzahl von Bytes (ein bis drei) in Arrays verfügbar gemacht. An dieser Stelle geht es nur um die Bestimmung deklarierter Label, ein Ersetzen der Werte erfolgt an anderer Stelle.

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 Opcdoe beinhaltet. In der resultierenden `HashMap<String, u16>` verweist dann ein gefundenes Label auf den Wert der Variable, zum Zeitpunkt ihres Funds.

Das Identifizieren von Labels erfolgt mittels folgendem Regex:

`"^( *[a-zA-Z@?][a-zA-Z@?0-9]*:)"`, 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 einem separaten Vektor gesammelt

und in dem Fall, dass die Methode auf einen Opcode trifft, dem eigentlichen Ergebnis beigefügt. Dieser 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.

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 Variablen durch ihre zugewiesenen Werte ersetzt. Außerdem sind 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 2.4.3 vorgestellt. Damit der Assembler eine Makroreferenz ersetzen kann, muss auch er entsprechend davor ermitteln welche Makros überhaupt deklariert wurden, was den ersten Schritt der Verarbeitung von Makros ausmacht. 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 Variablen `in_macro: bool` beeinflusst. Sie ist als `false` initialisiert und wird in Abhängigkeit von Makrostart und -ende entsprechend verändert.

Sollte ein weiterer 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. Alle anderen Zeilen werden, 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 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. Listing 5.2.3 illustriert das Ganze beispielhaft.

```
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());
    }
}
```

Quellcode 5.1: Auftrennen einer Definition eines Makros

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 temporären Variablen gespeichert. Ihre Inhalte werden in das Endergebnis geschrieben und die Variablen geleert, sobald ein ENDM aufkommt.

Referenzierungen

Sobald der Assembler eine Liste aller existierenden Makros besitzt und die 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 entspricht dem der vorhergehenden Implementierung mittels boolscher Variable, weshalb im Folgenden vor allem auf die Herausforderung, die Parameter ordentlich zu ersetzen, eingegangen wird.

Bei der Erkennung von Makros wird je Zeile überprüft, ob ihr Inhalt einem Eintrag in der, im vorherigen Schritt generierten, Map vorhanden ist. In diesem Fall müssen zuerst eventuelle Parameter ermittelt werden. Das ist auf dieselbe Art und Weise implementiert, wie bereits in Listing 5.2.3 vorgestellt.

Damit die Variablen innerhalb eines Makros durch ihre eigentlichen Werte ersetzbar sind, müssen diese entsprechend gemappt werden, was in Listing 5.2.3 zu sehen ist.

```
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);
}
```

Quellcode 5.2: Mapping von Nutzereingaben auf Parameter

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 (links beginnend) Parameter 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 resultierenden Vektor (`Vec<String>`) angehängt. Bei diesem Vorgang werden entsprechende Vorkommnisse von Parametern in den Befehlen ersetzt. Im Gegensatz zum Ersetzen von Variablen ist die hier geforderte Implementierung komplexer, weil der Nutzer wiederum Variablennamen als Parameter angeben kann. Das führt möglicherweise zu unerwarteten Seiteneffekten sobald ein Parameter Substring eines anderen ist, weshalb das genaue Vorgehen an der vereinfachen Fassung in Listing 5.2.3 zu sehen ist.

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 Namen, die nicht am Ende der Zeile stehen gefunden. Im Fall eines Treffers gibt die Methode `find()` ein `Option<Range>`-Objekt zu-

```
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()
```

Quellcode 5.3: Ersetzung von Variablen in Makros einer einzelnen Instruktion (vereinfacht)

rück dessen Inhalt, sofern vorhanden, in `reg_match` gespeichert wird. Die darin enthaltene Range läuft vom ersten Index des Matches bis zum letzten. Aus dieser Range lassen sich das erste und letzte Zeichen des gematchten Strings bestimmen, welche in den Variablen `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 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`. Außerdem wurde `Foo` vorher im Programm als Variable initialisiert.

Es wird schnell ersichtlich, dass dieses Ergebnis nicht das ist, was der Programmierer erwarten würde. Durch das Hinzufügen eines Werts, den der

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

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.

6 Auswertung

Abkürzungsverzeichnis

Wasm WebAssembly

RAM Arbeitsspeicher

Literatur

[1] 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.

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

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

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

[5] *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/.

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