Design und Implementierung eines Debuggers für Mikro-Assembler-Programme

Studienarbeit

des Studiengangs Angewandte Informatik
der Dualen Hochschule Baden-Württemberg Stuttgart

von

Christian Rösch

Oktober 2011 - Mai 2012

Bearbeitungszeitraum	Oktober 2011 - Mai 2012
Bearbeitungsdauer	300 Stunden
Matrikelnummer	0487930
Kurs	TIT09AIC
Ausbildungsfirma	icon Systemhaus GmbH – Stuttgart
Gutachter der Studienakademie	Prof. Dr. Karl Stroetmann

Zusammenfassung

In dieser Arbeit stelle ich den *micro-debug* und die *micro-debug-gui* vor. Der *micro-debug* ist ein konsolenbasierter Debugger für die *Mic-1*, die *Tanenbaum* in [TG98] beschreibt. Die *micro-debug-gui* ist eine GUI (grafische Benutzerschnittstelle) für den *micro-debug*.

Sowohl für den *micro-debug* als auch die *micro-debug-gui* beschreibe ich, wie sie bedient werden. Ich zeige ein Tutorial, wie mit dem *micro-debug* ein Fehler im Assembler-Code gefunden werden kann und weise auf die Unterschiede zwischen dem *micro-debug* und der *micro-debug-gui* hin.

Im zweiten Teil der Arbeit konzentriere ich mich auf die Entwicklung des *micro-debug* und der *micro-debug-gui*. Die Werkzeuge, die ich zur Entwicklung genutzt habe, stelle ich vor und zeige, wie ein Außenstehender sich an der Entwicklung des *micro-debug* oder der *micro-debug-gui* beteiligen kann. Auch den Aufbau des Codes beschreibe ich und beschreibe, welche Funktionalität des *micro-debug* wo implementiert ist.

Abstract

In this seminar paper I present the *micro-debug* and the *micro-debug-gui*. The *micro-debug* is a console based debugger for the *Mic-1*, described by *Tanenbaum* in [TG98]. The *micro-debug-gui* is a GUI for the *micro-debug*.

For the *micro-debug* and the *micro-debug-gui* I describe how to use them. Also there is a tutorial where I explain, how you can find bugs in an assembler code with the *micro-debug*. Furthermore I show the differences between the *micro-debug* and the *micro-debug-gui*.

In the second part I concentrate on the development of the *micro-debug* and the *micro-debug-gui*. I describe the tools I've used for development and explain how you can use them to participate in development of the *micro-debug* and the *micro-debug-gui*. Also I describe the structure of the code and where which functionality of the *micro-debug* has been implemented.

Inhaltsverzeichnis

Glossar

Ijvm Ein von *Tanenbaum* entwickelter Befehlssatz für die *Mic-1*. 1, 6

ANT Ein Build-Werkzeug. 35

git Ein Versionskontrollsystem. 31, 32, 34, 35

Java Eine objektorientierte Programmiersprache. 4, 35, 40, 41

JUnit Framework zur Automatisierung von Tests für Java-Anwendungen. 49

Make Ein Build-Werkzeug. 35

Maven Ein Build-Management-Werkzeug. 35, 36, 40

Subversion Ein Versionskontrollsystem von *Apache*. 31, 32

Abkürzungsverzeichnis

ALU Arithmetisch-logische Einheit (Arithmetic Logic Unit). 40, 41

API Programmierschnittstelle (Application Programming Interface). 49

AUT Application Under Test. 49

CISC Complex Instruction Set Computing. 1

EDT Event Dispatch Thread. 26, 48, 49, 51

FEST Fixtures for Easy Software Testing. 36, 49

GUI grafische Benutzerschnittstelle (Graphical User Interface). i, 1, 2, 24–26, 36, 49, 50, 52

JVM Java Virtual Machine. 7

MPC Micro program counter. 40, 42

RISC Reduced Instruction Set Computing. 1

XML Extensible Markup Language. 36

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1. Einleitung

Mit dieser Arbeit stelle ich *micro-debug* vor, einen Debugger für die *Mic-1*. Die *Mic-1* ist ein Prozessor und wird von *Tanenbaum* in [TG98] vorgestellt – sie ist ein Prozessor der CISC (Complex Instruction Set Computing) Architektur.

Im Gegensatz zur RISC (Reduced Instruction Set Computing) Architektur haben CISC-Prozessoren einen komplexeren Befehlssatz. Bei der *Mic-1* wird dies durch einen Mikro-Assembler-Code realisiert, der für die begrenzte Hardware einen komplexen Befehlssatz bereitstellt. Dieser Mikro-Assembler-Code ermöglicht, Maschinenbefehle auf einem hohen Abstraktionsniveau zu definieren und ohne Veränderung der Hardware anzupassen.

Aufgabenstellung

Im Skript der Vorlesung Rechnertechnik [Str09, Kapitel 8] wird von *Stroetmann* ein Mikro-Assembler für die *Mic-1* vorgestellt. Ziel dieser Arbeit ist der Entwurf und die Implementierung eines Debuggers für diesen Mikro-Assembler. Der Debugger soll die Entwicklung von Mikro-Assembler- und Assembler-Code für die *Mic-1* erleichtern.

Um entsprechenden Mikro-Assembler- und Assembler-Code debuggen zu können, muss der Debugger die *Mic-1* simulieren können. In seiner Studienarbeit [Kut04] beschreibt *Kutzer* einen Simulator für die *Mic-1*. Der hier entwickelte Debugger soll diesen Simulator ersetzen und um Debug-Funktionen erweitern.

In dieser Arbeit sollen dabei hauptsächlich zwei Teilaufgaben bearbeitet werden.

Debugger für die Konsole

Zunächst soll eine Version des Debuggers entwickelt werden, die über die Konsole bedient wird. Das Ziel ist die Implementierung des Debuggers unabhängig von der Darstellung der Ein- und Ausgabe.

Der Debugger soll in der Lage sein zwei Binärdateien einzulesen: eine Datei, die den Mikro-Assembler- und eine, die den Assembler-Code enthält. Der Assembler-Code sollte grundsätzlich nur vom Mikro-Assembler-Code abhängig und durch den Benutzer frei definierbar sein; als Referenz für diese Arbeit kann aber der IJVM-Assembler dienen.

Der Benutzer muss Breakpoints (Haltepunkte) definieren können, die die Simulation des Mikro-Assembler-Code unterbrechen. Der Debugger ermöglicht dem Benutzer dabei, Breakpoints für bestimmte Mikro-Assembler-Instruktionen aber auch für bestimmte Assembler-Instruktionen zu definieren.

Neben den Werten der Register kann der Benutzer sich auch den Inhalt des Stacks, die gesetzten Breakpoints, den Mikro-Assembler- und den Assembler-Code anzeigen lassen. Dem Benutzer soll es möglich sein, lokale Variablen, Register, den ausgeführten Mikro-Assembler- sowie Assembler-Code zu beobachten.

GUI für den Debugger

Im zweiten Teil der Aufgabe wird für den Debugger eine GUI entwickelt werden. Die GUI soll sowohl den Mikro-Assembler- als auch den Assembler-Code anzeigen; Auch die Register mit ihren Werten und der Stack sollen sichtbar sein.

Die GUI ermöglicht dem Benutzer, Breakpoints zu setzen und zeigt an, welche Mikro-Assembler- und Assembler-Instruktion gerade abgearbeitet wird.

Da in dieser Arbeit ein allumfassender Debugger weder geschaffen werden kann noch soll, ist besonders darauf zu achten, die Wartung und Erweiterung des Debuggers außenstehenden Personen zu ermöglichen. Besonders der Code des Debuggers soll daher gut dokumentiert werden; es sollen sich andere Studenten in den Code einarbeiten und neue Funktionalitäten implementieren oder bestehende Funktionalitäten anpassen können.

Aufbau der Arbeit

Die Arbeit ist in zwei Teile geteilt: Im ersten Teil erläutere ich die Bedienung und im zweiten Teil die Entwicklung des *micro-debug*. Ich beschreibe also eher den derzeitigen Zustand des *micro-debug*, als den Weg der Entstehung des *micro-debug*. Denn diese Arbeit soll vor allem dazu dienen, dass Außenstehende sich in den *micro-debug* einarbeiten können und den jetzigen Zustand nachvollziehen können.

Ich werde im Folgenden *micro-debug* für die Konsolenvariante des Debuggers verwenden und *micro-debug-gui* für die GUI. Diese beiden Bezeichnungen sind gleichzeitig die Namen der Code-Projekte – so möchte ich eine saubere Trennung zwischen den Projekten erreichen. Die Bezeichnung *Debugger* werde ich verwenden, um allgemein über den *micro-debug* und die *micro-debug-gui* zu sprechen; oder für Erklärungen, die beide Projekte betreffen.

Bedienung

Im Kapitel 2 beantworte ich einige allgemeine Fragen über den Debugger: Welche Systemvoraussetzungen gibt es? Wie konfiguriert man den Debugger? Wie kann man den Debugger in andere Sprachen übersetzen? Produziert der Debugger Log-Ausgaben? Wenn ja, kann man diese konfigurieren?

Die Bedienung des *micro-debug* beschreibe ich im Kapitel 3; dort erkläre ich auch die Parameter des *micro-debug*. Außerdem erläutere ich die verschiedenen Befehle und zeige ein Beispiel, wie mit dem *micro-debug* ein Fehler im Assembler-Code gefunden werden kann.

Im Kapitel 4 beschreibe ich die Bedienung der *micro-debug-gui*; dort nenne ich die Unterschiede zum *micro-debug* und erkläre die verschiedenen Oberflächenelemente.

Entwicklung

In Kapitel 5 beschreibe ich einige Werkzeuge, die ich zur Entwicklung des Debuggers genutzt habe und deren Verständnis nötig ist, um selbst Änderungen am Projekt vorzunehmen. In Kapitel 6 gehe ich kurz auf die Notwendigkeit der automatisierten Tests ein.

In den nachfolgenden Kapiteln beschreibe ich, welche Funktionalität des Debuggers sich in welchen Klassen befindet. Kapitel 7 nutze ich, um zu beschreiben, wie die *Mic-1* simuliert wird und wie ich die verschiedenen Hardwarekomponenten implementiert habe. Im Kapitel 8 erkläre ich den Code des *micro-debug* und in Kapitel 9 den Code der *micro-debug-gui*.

Teil I.

Bedienung

2. Allgemein

Der Debugger – sowohl der *micro-debug* als auch die *micro-debug-gui* – benötigt keine Installation; er kann als `.zip`-Archiv heruntergeladen werden und ist direkt nach dem Entpacken nutzbar. Die Datei heißt üblicherweise `micro-debug-version.zip`[1] und enthält mehrere Dateien:

micro-debug-version.jar enthält den Code des *micro-debug* und kann mit dem Befehl `java -jar` ausgeführt werden. Diese Datei ist auch in der *micro-debug-gui* enthalten, dort gibt es aber zusätzlich noch die Datei `micro-debug-gui-version.jar`, die den Code der *micro-debug-gui* enthält.

micro-debug.sh und micro-debug.bat sind die Startskripte des *micro-debug* für Windows und Linux[2]. Der Debugger kann durch Ausführen dieser Skripte gestartet werden – der Benutzer muss sich dadurch nicht um die Konfiguration des Klassenpfades kümmern. Ich habe die Startskripte so geschrieben, dass sie von beliebigen Orten ausgeführt werden können: Der Benutzer kann den Debugger von vielen Arbeitsverzeichnissen aus aufrufen und hat die Konfigurationsdateien nur an einer Stelle zu pflegen.

config/micro-debug.properties ist die Konfigurationsdatei für den *micro-debug* und die *micro-debug-gui*, hier kann beispielsweise die Größe des Hauptspeichers oder Tastenkombinationen konfiguriert werden – siehe Abschnitt 2.2.

config/logging.properties ist die Konfigurationsdatei für das Logging des Debuggers – hier kann der Benutzer definieren, ob und welche Ausgaben auf der Konsole oder in einer Datei erscheinen sollen – siehe Abschnitt 2.2.2.

config/lang/ enthält die Textressourcen des Debuggers – siehe Abschnitt 2.3.

lib/ enthält die verwendeten Bibliotheken des Debuggers; zur Zeit nur in der *micro-debug-gui* verwendet.

2.1. Systemvoraussetzungen

Der Debugger ist in Java geschrieben und daher prinzipiell an keine spezielle Plattform gebunden. Voraussetzung für die Nutzung des *micro-debug* ist lediglich Java 5.

In der Praxis differenzieren sich die verschiedenen Plattformen, auf denen Java verfügbar ist, in einigen Feinheiten. Daher ist es prinzipiell möglich, dass sich der Debugger auf gewissen Plattformen unerwünscht verhält. Wie anhand der Startskripte zu erkennen ist, habe ich den Debugger im Blick auf *Linux* und *Windows* entwickelt und auf diesen Plattformen[3] getestet. Um den Debugger auf einer weiteren Plattform zu nutzen, muss zumindest das Startskript eigenständig geschrieben werden.

Der *micro-debug* nutzt bislang einen Thread, die *micro-debug-gui* hingegen mindestens zwei (wie ich in Abschnitt 9.1.3 näher beschreibe), die allerdings selten parallel arbeiten. Für die Ausführungsgeschwindigkeit des Debuggers ist die Anzahl der verfügbaren Prozessorkerne daher irrelevant.

[1] Die Datei der *micro-debug-gui* heißt üblicherweise `micro-debug-gui-version.zip`.

[2] Auch hier ist die Namensgebung für die Startskripte der *micro-debug-gui* entsprechend: `micro-debug-gui.sh` und `micro-debug-gui.bat`

[3] Ausführlich habe ich mit *Windows 7* (in der 64-Bit-Variante) und *Ubuntu 12.04* (auch 64-Bit) getestet.

2.2. Konfiguration

Die Konfiguration des Debuggers findet in `.properties`-Dateien statt. Die Konfigurationsdatei `config/micro-debug.properties` enthält die Konfigurationen sowohl für den *micro-debug* als auch die *micro-debug-gui*. Wie in `.properties`-Dateien üblich, werden dort Schlüsselwert-Paare hinterlegt.

```
1 # default value for the register CPP
2 mic1.register.cpp.defval = 0x4000
```

Listing 2.1: Eintrag in `conf/micro-debug.properties`

Listing 2.1 zeigt einen Eintrag aus der Konfigurationsdatei: Den Startwert für das Register `CPP`. Diese Datei kann für jeden Schlüssel genau einen Wert enthalten. Ist für einen Schlüssel kein Wert konfiguriert oder ist ein ungültiger Wert konfiguriert, nutzt der Debugger den internen Standardwert für den jeweiligen Schlüssel.

Dadurch ist es möglich ein Update des Debuggers durchzuführen, und weiterhin die alte Konfiguration zu nutzen. Ersetzt der Benutzer die `.jar`-Datei durch eine neuere Version, die zusätzliche Konfigurationsoptionen benötigt, nutzt der Debugger die Standardwerte dieser Konfigurationsoptionen.

Der Benutzer kann seine eigene Konfiguration dadurch von Version zu Version des Debuggers behalten; er muss geänderte Konfigurationsoptionen nicht mühsam nachpflegen.

2.2.1. Zahlenformat

In Listing 2.1 ist zu sehen, wie dem Register `CPP` der Standardwert `0x4000` zugewiesen wird – hier hexadezimal notiert. Andere Konfigurationseinträge sind dezimal eingetragen; welches Zahlenformat ist nun wo anzuwenden?

Für den Debugger ist es irrelevant, welches Zahlenformat bei welcher Konfigurationsoption verwendet wird. Er liest die gegebene Zahl und wandelt sie in einen *Integer* um; dadurch kann der Benutzer für jeden Eintrag das passende Zahlenformat wählen. Nicht nur bei Konfigurationsoptionen, sondern bei allen eingegebenen Zahlen des Benutzers ist das Format variabel.

Welche Zahlenformate gibt es außer dezimal und hexadezimal? Der Debugger unterstützt alle Zahlensysteme mit der Grundzahl von $b = 2$ bis $b = 36$. Generell werden die Zahlen in der Art `ZAHL_BASIS` angegeben – `ZAHL` wird in dem jeweiligen Zahlensystem und `BASIS` im Dezimalsystem angegeben. In Tabelle 2.1 sind verschiedene Eingabemöglichkeiten für die Zahl 10 gegeben.

Zahlensystem	**Darstellung**
Dualsystem	`1010_2`
Ternärsystem	`101_3`
Dezimalsystem	`10_10`
Hexadezimalsystem	`A_16`

Tabelle 2.1.: Auswahl möglicher Eingabearten der Zahl 10 im Debugger

Für die meiner Meinung am häufigsten verwendeten Zahlensysteme gibt es die in Tabelle 2.2 dargestellten Vereinfachungen. Der Benutzer *muss* diese Vereinfachungen aber nicht nutzen, er kann sowohl `A_16` als auch `0xA` eingeben – oder aber `10`.

Zahlensystem	ausführlich	vereinfacht
Dualsystem	`1010_2`	`0b1010`
Oktalsystem	`12_8`	`0o12`
Dezimalsystem	`10_10`	`10`
Hexadezimalsystem	`A_16`	`0xA`

Tabelle 2.2.: Vereinfachungen typischer Zahlenformate im Debugger am Beispiel der Zahl 10

2.2.2. Logs

Sucht der Benutzer die Ursache für einen Fehler, den er oder der Debugger begangen hat, sind Log-Ausgaben ein guter Ansatzpunkt. Denn dort sind die vermeintlich wichtigen Vorgänge protokolliert und im Hinblick auf solche Analysen geschrieben.

Der Debugger wird im Startskript so gestartet, dass die Log-Konfiguration aus der Datei `conf/logging.properties` genutzt wird. Da die Syntax und Semantik der Konfigurationseinträge von *Oracle* [Ora04] und *Vogel* [Vog12] detailliert beschrieben ist, lasse ich sie hier aus.

Die Startskripte sind so konfiguriert, dass der Debugger alle Log-Ausgaben mit dem Log-Level `INFO` oder höher in Dateien schreibt. Diese Dateien liegen im *home*-Verzeichnis des Benutzers und sind von der Form `micro-debugX.log` – wobei `X` ein Laufindex ist. Es existieren neben dem genannten noch weitere Log-Level; je nach Situation kann der Debugger beispielsweise für ausführlichere Log-Ausgaben konfiguriert werden. Die verschiedenen Log-Level sind mit absteigender Wichtigkeit: `SEVERE`, `WARNING`, `INFO`, `CONFIG`, `FINE`, `FINER` und `FINEST`.

2.2.3. ijvm.conf

Ich habe den Debugger so konstruiert, dass er prinzipiell für jeden beliebigen Assembler-Code genutzt werden kann. Allerdings habe ich den Debugger nur mit dem IJVM-Assembler getestet, da der Debugger in der Praxis wohl vorwiegend mit dem IJVM-Assembler genutzt wird.

Um den Assembler-Code disassemblieren zu können, gibt es eine Konfigurationsdatei – die `ijvm.conf`. Damit der Debugger einen gewissen Standardsatz an IJVM-Befehlen versteht, wird diese Datei in der `.jar`-Datei des Debuggers mit ausgeliefert.

Möchte der Benutzer die Datei `ijvm.conf` anpassen, genügt es, die Datei im Verzeichnis `conf/` abzulegen. Durch das Startskript wird sie dann auf den Klassenpfad geladen; vor der internen `ijvm.conf`-Datei in der `.jar`-Datei und ersetzt diese somit.

Die Datei `ijvm.conf` ist zeilenweise zu lesen, wobei eine Zeile nach folgendem Format aufgebaut ist:

⟨*line*⟩ ::= ⟨*comment*⟩
| ⟨*address*⟩ ⟨*white*⟩+ ⟨*identifier*⟩ ⟨*white*⟩* ⟨*argumentlist*⟩ ⟨*comment*⟩?

⟨*comment*⟩ ::= '//' ⟨*text*⟩

⟨*address*⟩ ::= '0x' ⟨*hexchar*⟩ ⟨*hexchar*⟩

⟨*hexchar*⟩ ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'

⟨*identifier*⟩ ::= ⟨*word*⟩

⟨*argumentlist*⟩ ::= (⟨*white*⟩ ⟨*argument*⟩ ⟨*white*⟩*)*

⟨*argument*⟩ ::= 'byte' | 'const' | 'index' | 'label' | 'offset' | 'varnum'

⟨*white*⟩ ::= ' ' | '\t'

2.3. Lokalisierung

Der Debugger wurde in Englisch geschrieben – weitere Sprachen können über das Verzeichnis `conf/lang/` hinzugefügt werden.

In diesem Verzeichnis liegen verschiedene `.xml`-Dateien, die dazu dienen, dem Benutzer Ausgaben in seiner Sprache anzuzeigen. Die Sprache, die der Benutzer angezeigt bekommt, hängt vom *Locale* (siehe dazu ausführlich [Ora10]) der JVM (Java Virtual Machine) ab.

Der Debugger sucht für diese Sprache folgende vier Dateien, die Schlüsselwert-Paare für die Textkonstanten enthalten:

text_lang_CT_var1.xml wobei **lang** die Sprache, **CT** die Länderkennung und **var1** die Variante ist, die vom *Locale* gegeben sind. Diese Datei ist die spezifischste und wird sofern sie existiert zuerst geladen. Schlüssel, die hier nicht gefunden werden, werden in den folgenden Dateien gesucht.

text_lang_CT.xml in der Praxis existieren selten Informationen über die Variante des *Locale*, daher kann `_var1` weggelassen werden.

text_lang.xml häufig existieren zwar verschiedene Länderkennungen für die gleiche Sprache, aber meist unterscheiden sich die Übersetzungsdateien für die Länderkennungen nicht. Daher werden in der Praxis viele Dateien in dieser Form angelegt werden.

text.xml die Basis-Datei für die Lokalisierung. Wenn ein Schlüssel in den anderen Dateien nicht gefunden wurde oder die anderen Dateien nicht existieren, dann wird er in dieser Datei gesucht.

Wie ich gerade angedeutet habe, ist der Aufbau dieser vier Dateien hierarchisch: ein Schlüssel, der in der spezifischsten Datei gefunden wurde, wird aus den anderen Dateien nicht mehr ausgewertet.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
3 <properties>
4     <entry key="border">++++++++++</entry>
5     <entry key="border2">---</entry>
6 </properties>
```

Listing 2.2: Beispiel für Datei `text_de_DE.xml`

Anhand eines Beispiels möchte ich dieses Verhalten erläutern: Gegeben seien die drei Dateien, die in Listing 2.2, Listing 2.3 und Listing 2.4 aufgeführt sind. Die Datei der Form `text_lang_CT_var1.xml` ist in diesem Beispiel nicht vorhanden.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
3 <properties>
4     <entry key="file-not-found">Datei nicht gefunden {0}</entry>
5     <entry key="version">Micro-Debug - Version {0}</entry>
6 </properties>
```

Listing 2.3: Beispiel für Datei `text_de.xml`

Listing 2.5 zeigt das Ergebnis nach der Auswertung dieser drei Dateien am Beispiel des deutschen *Locale*. Es ist zu sehen, wie die Einträge sich überschreiben – ein Benutzer mit englischem *Locale* würde in diesem Beispiel den Eintrag für `border2` nicht erhalten und hätte dafür keinen definierten Wert.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <entry key="version">Micro-Debug: Version {0}</entry>
    <entry key="file-not-found">file not found {0}</entry>
    <entry key="border">----------------------------------------</entry>
</properties>
```

Listing 2.4: Beispiel für Datei `text.xml`

Die Reihenfolge der Schlüsselwert-Paare ist willkürlich und hat für den Programmverlauf keine Auswirkung.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <entry key="version">Micro-Debug - Version {0}</entry>
    <entry key="file-not-found">Datei nicht gefunden {0}</entry>
    <entry key="border">++++++++++</entry>
    <entry key="border2">---</entry>
</properties>
```

Listing 2.5: Beispiel für Ergebnis nach Verarbeitung der Lokalisierungsdateien

Dieses System ermöglicht das einfache Hinzufügen neuer Sprachen. Dieser Vorgang ist zwar bisher nicht für den Benutzer vorgesehen, aber prinzipiell kann er in diesen Dateien Anpassungen vornehmen und sich dadurch die Ausgaben des Debuggers an seine Bedürfnisse anpassen. Auch ist denkbar, dass Übersetzungen des Debuggers von Dritten angeboten wird – der Benutzer kann dann die Sprachdateien für seine Sprache herunterladen und in dem Verzeichnis `conf/lang/` ablegen.

Im Gegensatz zur Konfigurationsdatei gibt es bei den Lokalisierungsdateien keinen individuellen Fallback für jeden Schlüssel, sodass fehlende Werte mit einem Fehlertext belegt werden. Die Lokalisierungsdateien sind daher auch von einem Update des Debuggers betroffen – mögliche Änderungen des Benutzers gehen verloren oder müssen in die neuen Dateien übernommen werden.

Einige Einträge in den Lokalisierungsdateien können Platzhalter enthalten. Platzhalter gelten nur für ein bestimmtes Schlüsselwert-Paar und sind von 0 aufsteigend nummeriert in der Form `0` zu verwenden. Der Inhalt mit dem der jeweilige Platzhalter gefüllt wird kann in den Kommentaren in der Datei nachgelesen werden.

Zusätzlich zu den `text...xml`-Dateien gibt es bei der *micro-debug-gui* `text-gui...xml`-Dateien. Die Lokalisierungsdateien des *micro-debug* und der *micro-debug-gui* sind also voneinander unabhängig.

Die Log-Ausgaben aus Abschnitt 2.2.2 sind immer auf Englisch und können über die `.xml`-Dateien nicht verändert werden.

3. Interaktion per Konsole

Wie in Kapitel 2 beschrieben, muss der *micro-debug* nicht installiert werden: Vom Herunterladen bis zum Starten des *micro-debug* genügen folgende Schritte.

1. Die Datei `micro-debug-version.zip` von der Projektseite [Rö12a] herunterladen
2. Die `.zip`-Datei in ein beliebiges Verzeichnis entpacken (beispielsweise `/opt/micro-debug/`)
3. Das Verzeichnis des *micro-debug* dem `PATH` hinzufügen
4. Den *micro-debug* starten – mit `micro-debug.sh --help`

Der *micro-debug* kann durch einige Parameter gesteuert werden und wird nach dem Start durch Befehle bedient. Ich werde daher zunächst die verschiedenen Parameter des *micro-debug* und dann die Befehle zur Bedienung des *micro-debug* erklären. Am Ende des Kapitels zeige ich in einem Tutorial, wie mit dem *micro-debug* Fehler im Assembler-Code gefunden werden können.

3.1. Parameter

Der Standardaufruf für den *micro-debug* ist in Listing 3.1 zu sehen. Es gibt zwei verpflichtende Parameter: Die Pfade zur Mikro-Assembler- und zur Assembler-Code-Datei. Die beiden Pfade können sowohl relativ als auch absolut angegeben werden, wichtig ist allerdings, dass zuerst der Pfad zur Mikro-Assembler- und dann der Pfad zur Assembler-Code-Datei angegeben wird. Werden die beiden Pfade vertauscht, so startet der *micro-debug* nicht und bricht mit einer Fehlermeldung ab.

```
1 micro-debug.sh [PARAMETER]... MIC1 IJVM
```

Listing 3.1: Aufruf des *micro-debug*

Neben den beiden Dateipfaden gibt es im Folgenden erklärte optionale Parameter. Jeder Parameter kann sowohl in der langen (mit doppeltem Minus) als auch in der kurzen (einfaches Minus gefolgt von einem Zeichen) Variante angegeben werden.

-h, --help ist dieser Parameter gegeben, so wird die Hilfe angezeigt, die neben den verschiedenen Aufrufmöglichkeiten die möglichen Parameter erklärt. Zusätzlich werden noch einige andere Informationen angezeigt, wie beispielsweise Kontaktmöglichkeiten oder ein Hinweis, wo Fehler berichtet werden können.

-o, --output-file FILE wenn dieser Parameter angegeben ist, so wird die Ausgabe der *Mic-1* (nicht des *micro-debug*) in eine Datei umgelenkt. Normalerweise wird die Ausgabe der *Mic-1* auf der Konsole ausgegeben.

Das Argument `FILE` ist der Pfad zur Datei, in die die Ausgabe geschrieben werden soll. Wenn die Datei bereits existiert, wird die Ausgabe an das Ende der Datei angehängt.

Unter Linux kann man dann in einer zweiten Konsole diese Datei beispielsweise mit `tail -f` anzeigen und die Ausgabe der *Mic-1* komfortabel von der Ausgabe des *micro-debug* trennen. In diesem Szenario ist auch der Parameter `--unbuffered-output` sinnvoll.

-u, --unbuffered-output verhindert die Pufferung der Ausgabe der *Mic-1*. Normalerweise gibt der *micro-debug* die Ausgabe der *Mic-1* zeilenweise aus, also erst bei der Ausgabe eines Zeilenumbruchs. Verwendet man den Parameter `--output-file` oder möchte man aus sonstigen Gründen jedes ausgegebene Zeichen der *Mic-1* direkt auf der Konsole sehen, so ist dieser Parameter die Lösung.

Vorsicht! Wird dieser Parameter ohne `--output-file` genutzt, so kann ist es schwer, die ausgegebenen Zeichen der *Mic-1* ausfindig zu machen, da sie in der Menge der Ausgaben des *micro-debug* untergehen.

-v, --version gibt die Version des *micro-debug* aus.

Wenn einer der Parameter `--help` oder `--version` angegeben wurde, startet der *micro-debug* nicht. Dies kann genutzt werden, um ohne vorhandene Bytecode-Dateien Informationen über den *micro-debug* anzeigen zu können. Listing 3.2 zeigt, wie der *micro-debug* daher ohne Bytecode-Dateien aufgerufen werden kann.

```
1 micro-debug.sh --help
2 micro-debug.sh --version
```

Listing 3.2: Aufruf des *micro-debug* ohne Start

Bei der Abarbeitung der gegebenen Parameter hängt die Reihenfolge der Parameter nicht von der Reihenfolge ab, in der sie dem *micro-debug* als Parameter übergeben wurden. Wichtig ist nur, dass die beiden Bytecode-Dateien die letzten beiden Parameter und in der richtigen Reihenfolge aufgeführt sind.

3.2. Befehle

Bei der Bedienung des *micro-debug* muss der Benutzer zwischen den verschiedenen Befehlen auch Zeichen für die *Mic-1* eingeben. Damit der Benutzer sieht, ob die Eingabe für den *micro-debug* oder die *Mic-1* sind, gibt der *micro-debug* '`micro-debug>` ' und die *Mic-1* '`mic1>` ' aus, bevor eine Eingabe erwartet wird. Wenn die *Mic-1* eine Eingabe erwartet, kann der Benutzer mehrere Zeichen auf einmal eingeben – an die *Mic-1* werden die eingegebenen Zeichen plus ein Zeilenumbruch gesendet. Sollte der Assembler-Code einzelne Zeichen in einer Schleife einlesen, sollte die gesamte Zeile daher auf einmal eingeben werden.

Ist der *micro-debug* gestartet lässt er sich durch verschiedene Befehle bedienen, die ich jetzt ausführlich beschreiben werde.

Die verschiedenen Befehle können ein oder mehrere Argumente benötigen. Die Argumente haben einen der folgenden Datentypen:

Register ist der Name eines Registers, also `CPP`, `H`, `LV`, `MAR`, `MBR`, `MBRU`, `MDR`, `OPC`, `PC`, `SP` oder `TOS`.

Zahl ist eine Zahl im Wertebereich eines *Integers*. Die Formate für die Eingabe der Zahl habe ich in Abschnitt 2.2.1 beschrieben.

Je nach Befehl kann der zulässige Wertebereich eingeschränkt sein – eine solche Einschränkung ergibt sich aus der Beschreibung des jeweiligen Befehls.

break *Register [Zahl]*

setzt einen Breakpoint für das gegebene *Register*: Sobald das Register den Wert *Zahl* erhält, hält der *micro-debug* an. Der *micro-debug* hält, **nachdem** das Register den Wert erhalten hat – da dies unter Umständen zu spät ist, gibt es die Möglichkeit, das Argument *Zahl* wegzulassen. Wird nur ein Register (ohne Wert) angegeben, hält der *micro-debug* **bevor** das Register einen neuen Wert zugewiesen bekommt.

exit
beendet den *micro-debug*.

help
zeigt die verfügbaren Befehle mit einer kurzen Beschreibung an. Zeigt des weiteren einige Zusatzinformationen über den *micro-debug* an.

ls-break
zeigt alle Breakpoints mit der jeweiligen Bedingung an. Jeder Breakpoint erhält eine Identifikationsnummer, die bei anderen Operationen, wie dem Entfernen, angegeben werden muss.

ls-macro-code *[Zahl1 [Zahl2]]*

zeigt den disassemblierten Assembler-Code an. Dabei gibt es drei mögliche Konstellationen der Parameter:

- Wird **kein Parameter** gegeben, wird der vollständige Assembler-Code angezeigt.
- Wird **ein Parameter** *Zahl1* gegeben, wird die angegebene Anzahl an Zeilen vor und nach der aktuellen Zeile angezeigt.
- Werden **zwei Parameter** gegeben, wird der Assembler-Code von Zeile *Zahl1* bis zur Zeile *Zahl2* (inklusive) angezeigt.

Da der *micro-debug* nur den Bytecode kennt, gibt es eigentlich keine Zeilen. Der *micro-debug* schreibt pro Zeile einen Befehl (inklusive Argumente), somit existieren Zeilennummern, die für den *micro-debug* als Referenz dienen.

ls-micro-code *[Zahl1 [Zahl2]]*

zeigt den disassemblierten Mikro-Assembler-Code an und arbeitet wie `ls-macro-code`. Es gibt drei mögliche Konstellationen der Parameter:

- Wird **kein Parameter** gegeben, wird der vollständige Mikro-Assembler-Code angezeigt.
- Wird **ein Parameter** *Zahl1* gegeben, wird die angegebene Anzahl an Zeilen vor und nach der aktuellen Zeile angezeigt.
- Werden **beide Parameter** gegeben, wird der Mikro-Assembler-Code von Zeile *Zahl1* bis zur Zeile *Zahl2* (inklusive) angezeigt.

Wie bei `ls-macro-code` schreibt der *micro-debug* pro Zeile eine Mikro-Assembler-Instruktion (die 36 *Bit*, die die Instruktion spezifizieren), somit existieren Zeilennummern, die für den *micro-debug* als Referenz dienen.

ls-mem *Zahl1 Zahl2*

zeigt den Inhalt des Hauptspeichers zwischen den Adressen *Zahl1* und *Zahl2* (inklusive) an. Die Adressen des Hauptspeichers sind Wortadressen – jedes Wort hat eine Größe von 32 *Bit*.

ls-reg *[Register]*

zeigt den Wert von dem gegebenen *Register* an; wird das optionale Argument weggelassen, werden alle Register und deren Werte angezeigt.

ls-stack
zeigt den aktuellen Stack an.

Hinweis: Dieser Befehl wird durch die Konfigurationsoption `stack.elements.to.hide` beeinflusst. Normalerweise wird der Stack von dem initialen Stackpointer bis zum aktuellen

Stackpointer ausgegeben. Da dies unter Umständen mehr Elemente liefert, als der Stack tatsächlich enthält, gibt es die Möglichkeit über die Konfiguration die Ausgabe der ersten Elemente zu unterdrücken.

Möchte man den realen (im Speicher vorhandenen) Stack sehen, sollte man sicherstellen, dass `stack.elements.to.hide = 0` konfiguriert ist.

macro-break *Zahl*

fügt einen Breakpoint hinzu, der den *micro-debug* anhält, sobald der Assembler-Code an der Adresse *Zahl* ausgeführt werden soll. *Adresse* muss Element aus der Menge der von `ls-macro-code` angezeigten Zeilennummern sein.

micro-break *Zahl*

fügt einen Breakpoint hinzu, der den *micro-debug* anhält, sobald der Mikro-Assembler-Code an der Adresse *Zahl* ausgeführt werden soll. *Adresse* muss Element aus der Menge der von `ls-micro-code` angezeigten Zeilennummern sein.

micro-step *[Zahl]*

führt die nächsten *Zahl* Mikro-Assembler-Instruktionen aus; wird kein Argument gegeben, so wird eine Mikro-Assembler-Instruktion ausgeführt.

reset

die *Mic-1* wird in den Anfangszustand zurückgesetzt: Der Hauptspeicher und die Register werden auf die initialen Werte zurückgesetzt. Auch die Ein- und Ausgabe der *Mic-1* wird geleert. Die Informationen des *micro-debug*, vor allem die Breakpoints, bleiben allerdings erhalten und müssen vom Benutzer nicht erneut gesetzt werden.

rm-break *Zahl*

entfernt den Breakpoint mit der Nummer *Zahl*. Die Nummer des Breakpoints ist die Identifikationsnummer, die mit `ls-break` angezeigt wird.

run

führt alle Instruktionen bis zum Programmende oder bis zum nächsten Breakpoint aus.

Hinweis: Unter Umständen und ungünstigem Programmcode kann das zu debuggende Programm in eine Schleife geraten, welche ohne Breakpoints nur durch den Programmabbruch verlassen werden kann.

set *Register Zahl*

weist dem *Register* den Wert *Zahl* zu.

set-mem *Zahl1 Zahl2*

schreibt den Wert *Zahl2* an die Wortadresse *Zahl1* im Hauptspeicher.

Hinweis: Auch wenn dieser Befehl offensichtlich dazu genutzt werden könnte den Assembler-Code zu manipulieren, ist er für diesen Zweck nicht vorgesehen.

step *[Zahl]*

führt die nächsten *Zahl* Assembler-Instruktionen aus; wird kein Argument gegeben, so wird eine Assembler-Instruktion ausgeführt.

trace-mac

der Assembler-Code wird nun beobachtet. Dadurch wird jede Assembler-Instruktion angezeigt, nachdem sie ausgeführt wurde.

trace-mic
der Mikro-Assembler-Code wird nun beobachtet. Dadurch wird jede Mikro-Assembler-Instruktion angezeigt, nachdem sie ausgeführt wurde.

trace-reg *[Register]*

das gegebene *Register* wird nun beobachtet. Dadurch wird der Wert des Registers angezeigt, wenn er sich ändert. Wird das optionale Argument weggelassen, werden alle *Register* beobachtet.

trace-var *Zahl*

die Variable *Zahl* wird nun beobachtet. Dadurch wird der Inhalt der Variable angezeigt, wenn er sich ändert.

Zahl ist die Nummer der lokalen Variable. Wird eine Methode im Assembler-Code aufgerufen, ändert sich der Zeiger `LV` und damit auch die Identität der lokalen Variablen. Wird also Variable Nummer 1 in Methode X beobachtet und führt der *micro-debug* gerade Methode Y aus, wird eine Änderung der jetzigen lokalen Variable 1 nicht ausgegeben (sofern diese nicht auch beobachtet wird).

untrace-mac
beendet das Beobachten des Assembler-Code. Dadurch werden ausgeführte Assembler-Instruktion nun nicht mehr ausgegeben.

untrace-mic
beendet das Beobachten des Mikro-Assembler-Code. Dadurch werden ausgeführte Mikro-Assembler-Instruktion nun nicht mehr ausgegeben.

untrace-reg *[Register]*

beendet das Beobachten des angegebenen *Register*s. Wird das optionale Argument weggelassen, wird nun kein Register mehr beobachtet.

untrace-var *Zahl*

beendet das Beobachten der lokalen Variable Nummer *Zahl*.

3.3. Tutorial

In diesem Abschnitt möchte ich ein Tutorial beschreiben, um in das Arbeiten mit dem *micro-debug* einzuführen. Ich beschreibe das Debuggen eines Assembler-Programms: ein Programm zum Einlesen von Binärzahlen. Listing 3.3 zeigt den Code für das Programm in C und soll hier als Verständnis des Algorithmus dienen.

```
1  int main() {
2    int character = 0;
3    int result = 0;
4    while(1) {
5      character = getchar();
6      if( c == '\n' ) {
7        return result;
8      }
9      c = c - '0';
10     result = 2 * result + c;
11   }
12 }
```

Listing 3.3: C-Programm zum Einlesen einer Binärzahl

Der entsprechende Assembler-Code ist in Listing 3.4 aufgeführt. Dieses Programm muss nun kompiliert werden – *Ontko* stellt dafür in [Ont99] einige Programme bereit: `mic1asm` zum Kompilieren des Mikro-Assembler-Code und `ijvmasm` zum Kompilieren des Assembler-Code. In [Ont99] ist auch ein Mikro-Assembler-Code zu finden, der für dieses Tutorial ausreichend ist; außerdem gibt es dort die zum Mikro-Assembler-Code passende `ijvm.conf`-Datei.

```
1  .main
2  .var
3      c
4      result
5  .end-var
6      bipush      0
7      istore      result
8  loop:
9      in
10     istore      c
11     iload       c
12     bipush      10
13     if_icmpeq   finish
14     iinc        c       -48
15     iload       result
16     dup
17     iadd
18     iload       c
19     iadd
20     goto        loop
21 finish:
22     iload       result
23     halt
24 .end-main
```

Listing 3.4: IJVM-Assembler zum Einlesen einer Binärzahl

Zum Debuggen des Mikro-Assembler- und Assembler-Code empfehle ich, die `ijvm.conf`-Datei zu verwenden, die zum Kompilieren der `.mic1`-Datei genutzt wurde. Diese kann entweder in das `conf/` Verzeichnis des *micro-debug* gelegt werden, oder jeweils in das Verzeichnis, von dem aus der *micro-debug* ausgeführt wird. Wie erkenne ich, ob ich die korrekte `ijvm.conf`-Datei verwende? Beim Ausführen des Befehls `ls-macro-code`: zeigt der *micro-debug* unbekannte Assembler-Instruktionen, enthält der Assembler-Code Instruktionen, die in der `ijvm.conf` nicht oder mit abweichender Adresse definiert sind.

Vor dem Start des *micro-debug* empfehle ich außerdem, die Konfigurationsoption `mic1.micro.address.ijvm` zu überprüfen. Diese Option enthält die Adresse der Mikro-Assembler-Instruktion, die von allen Mikro-Assembler-Code-Methoden angesprungen wird, um die nächste Mikro-Instruktion zu *laden*. Ist diese Option falsch konfiguriert, funktioniert später der Befehl `step` nicht wie erwartet – der *micro-debug* hält nicht oder an falscher Stelle.

Die in Listing 3.4 aufgeführte Methode soll später eine eigene Methode werden und gibt daher keinen Wert aus, sondern legt das Ergebnis am Ende auf den Stack. Listing 3.5 zeigt in Zeile 1 den Befehl, um den *micro-debug* zu starten – im aktuellen Verzeichnis liegen die Dateien `mic1ijvm.mic1`, `binary-read.ijvm` und `ijvm.conf`.

```
1  micro-debug.sh mic1ijvm.mic1 binary-read.ijvm
2  MicroDebug - Copyright (C) 2011-2012 Christian Roesch AND 1999 Prentice-Hall, Inc.
3  Welcome! Please type 'help' for a list of valid commands
4  ----------------------------------------
5  micro-debug>
```

Listing 3.5: Start des *micro-debug*

Ab Zeile 2 steht die Willkommensnachricht des *micro-debug*, gefolgt von der Zeile 5, die anzeigt, dass der *micro-debug* nun einen Befehl erwartet. Mit dem Befehl `help` kann jederzeit eine ausführliche Beschreibung der verfügbaren Befehle anzeigt werden lassen.

Mit dem Befehl `ls-macro-code` erhält man den disassemblierten Assembler-Code– in Listing 3.6 ist die Ausgabe des Befehls zu sehen. Die Ausgabe zeigt zunächst pro Zeile die Assembler-Code-Zeile, dann die Adresse Instruktion im Mikro-Assembler-Code und anschließend den Namen des Befehls mit seinen Argumenten. Die Ausgabe ist nicht identisch mit dem Quellcode, der kompiliert wurde – aus Listing 3.4 – in der disassemblierten Variante fehlen Informationen wie Kommentare, Variablennamen und Sprungmarkennamen. Beispielsweise zeigt Zeile 7 einen bedingten Sprung zur Zeile `0x1B`.

```
1      0x0: [ 0x10] BIPUSH  0x0
2      0x2: [ 0x36] ISTORE  1
3      0x4: [ 0xFC] IN
4      0x5: [ 0x36] ISTORE  0
5      0x7: [ 0x15] ILOAD  0
6      0x9: [ 0x10] BIPUSH  0xA
7      0xB: [ 0x9F] IF_ICMPEQ  0x1B
8      0xE: [ 0x84] IINC  0 0xD0
9     0x11: [ 0x15] ILOAD  1
10    0x13: [ 0x59] DUP
11    0x14: [ 0x60] IADD
12    0x15: [ 0x15] ILOAD  0
13    0x17: [ 0x60] IADD
14    0x18: [ 0xA7] GOTO  0x4
15    0x1B: [ 0x15] ILOAD  1
16    0x1D: [ 0xFF] HALT
```

Listing 3.6: Disassemblierter Assembler-Code

Das erwartete Ergebnis eines Programmlaufs ist, dass die eingegebene Binärzahl am Ende auf dem Stack und damit im Register `TOS` vorliegt – wenn ich 1010 eingebe, soll `TOS` nach einem Programmdurchlauf den Wert 10 enthalten.

Mit dem Befehl `run` lasse ich das Programm nun zunächst ohne Breakpoints laufen. Nachdem das Programm gestartet ist, wird die *Mic-1* in Zeile 3 aus Listing 3.6 mit dem Befehl `IN` Zeichen einlesen. Das erkennt man auf der Konsole an der Zeile 2 aus Listing 3.7 – statt `micro-debug>` steht hier nun `mic1>`.

```
1 micro-debug> run
2 mic1>
```

Listing 3.7: *Mic-1* erwartet Eingabe

Ich gebe nun `1010` ein und bestätige die Eingabe mit `ENTER` – dadurch werden fünf Zeichen im *micro-debug* gepuffert: Die vier Zeichen, die ich eingegeben habe plus ein Zeilenumbruch. Jedes Mal, wenn die *Mic-1* nun ein Zeichen benötigt, wird aus diesem Puffer gelesen. Erst wenn dieser leer ist, erscheint erneut die Eingabeaufforderung für den Benutzer.

```
1 Processor executed 441 ticks.
2 micro-debug>
```

Listing 3.8: *micro-debug* gibt Anzahl ausgeführter Zyklen aus

Nachdem ich nun die Zahl eingegeben habe erscheint die Ausgabe aus Listing 3.8, die anzeigt, dass die *Mic-1* insgesamt 441 Zyklen ausgeführt hat. Da mein Programm keine Ausgabe macht, sondern das Ergebnis auf den Stack (und damit in dem Register `TOS`) ablegt, überprüfe ich das nun wie folgt. Mit dem Befehl `ls-reg` lasse ich mir den Inhalt des Registers `TOS` anzeigen, das ist in Listing 3.9 zu sehen.

Das Register `TOS` enthält den Wert 0 und damit einen falschen Wert. Ich kann nun noch den Stack ansehen, um zu überprüfen, ob dort der erwartete Wert 10 abgelegt ist. Den Stack sehe ich mit dem Befehl `ls-stack` an, was die in Listing 3.10 gezeigte Ausgabe liefert.

```
micro-debug> ls-reg TOS
Register TOS : 0x0
micro-debug>
```

Listing 3.9: *micro-debug* gibt Anzahl ausgeführter Zyklen aus

Der erwartete Wert 10 liegt auch nicht auf dem Stack. Wäre der Wert auf dem Stack, aber nicht im Register `TOS` wäre das ein Hinweis auf einen Fehler im Mikro-Assembler-Code. Denn der Mikro-Assembler-Code ist für die Einhaltung der Regel zuständig, dass das Register `TOS` stets den Wert des obersten Elements des Stacks enthalten muss.

```
Stack value #1 [  0xC001]: 0x1
Stack value #2 [  0xC002]: 0x0
Stack value #3 [  0xC003]: 0x1
Stack value #4 [  0xC004]: 0x0
Stack value #5 [  0xC005]: 0x0
micro-debug>
```

Listing 3.10: Inhalt des Stacks nach der Ausführung des Assembler-Code

Damit das Programm für weitere Analysen erneut ablaufen gelassen werden kann, muss die *Mic-1* auf ihren Startzustand zurückgesetzt werden; mit dem Befehl `reset`. Woran könnte das Problem liegen? Womöglich werden falsche Werte eingelesen – mit einem Breakpoint auf dem `IN`-Befehl überprüfe ich diese Vermutung. In Listing 3.11 ist aufgeführt, wie ich einen Breakpoint in der Zeile `0x4` setze und anschließend überprüfe, welche Breakpoints gesetzt sind.

```
micro-debug> macro-break 4
micro-debug> ls-break
Breakpoint #1: at macro code line 0x4
micro-debug>
```

Listing 3.11: Setzen eines Breakpoints im Assembler-Code

Nachdem der Breakpoint gesetzt ist, kann ich das Programm mit dem Befehl `run` ausführen. An der Ausgabe erkennt man, dass die *Mic-1* nicht das gesamte Programm ausgeführt hat, sondern nur 16 Zyklen. Ich werde nun den `IN`-Befehl überprüfen und dazu einzeln die Mikro-Assembler-Instruktion des Befehls ausführen. Damit ich nicht jedes Mal den Mikro-Assembler-Code anzeigen lassen muss, beobachte ich den Mikro-Assembler-Code mit dem Befehl `trace-mic`.

```
micro-debug> micro-step
Executed: fetch;goto 0x4
Processor executed 1 ticks.
micro-debug> micro-step
Executed: PC=PC+1;goto 0x5
Processor executed 1 ticks.
micro-debug> micro-step
Executed: goto (MBR)
Processor executed 1 ticks.
```

Listing 3.12: Ausgeführte Mikro-Assembler-Instruktionen bis zum Sprungbefehl `goto (MBR)`

Je nach Mikro-Assembler-Code kann es mehrere Mikro-Assembler-Instruktionen dauern, bis man zur Ausführung des `IN`-Befehls gelangt. Wichtig ist die Zeile, die `goto (MBR)` ausführt – daher führe ich nun so lange den Befehl `micro-step` aus, bis dieser Sprungbefehl ausgeführt wird. Bei mir werden von der Stelle, an der der *micro-debug* gehalten hat, bis zu dem Sprungbefehl drei Mikro-Assembler-Instruktionen ausgeführt, wie in Listing 3.12 zu sehen ist.

```
1 0xFC: H=OPC=-1;goto 0x6A
2 0x6A: OPC=H+OPC;goto 0x6B
3 0x6B: MAR=H+OPC;rd;goto 0x6C
4 0x6C: SP=MAR=SP+1;goto 0x6D
5 0x6D: TOS=MDR;wr;goto 0x3
```

Listing 3.13: Mikro-Assembler-Code des Befehls `IN`

Bei meiner Implementierung des Mikro-Assembler-Code liegt der Befehl `IN` an der Adresse `0xFC`. Listing 3.13 enthält den Mikro-Assembler-Code für den Befehl `IN`; bis zur Zeile `0x6B` wird die Adresse für *memory mapped IO* erzeugt: -3. In Zeile `0x6B` (Zeile 3 in Listing 3.13) wird der `rd`-Befehl ausgeführt, der letztendlich das Zeichen über den Hauptspeicher einliest.

```
1  micro-debug> micro-step 3
2  Executed: H=OPC=-1;goto 0x6A
3  Executed: OPC=H+OPC;goto 0x6B
4  mic1> 1010
5  Executed: MAR=H+OPC;rd;goto 0x6C
6  Processor executed 3 ticks.
7  micro-debug> micro-step 1
8  Executed: SP=MAR=SP+1;goto 0x6D
9  Processor executed 1 ticks.
10 micro-debug> ls-reg MDR
11 Register MDR : 0x31
```

Listing 3.14: Überprüfung des `rd`-Befehls durch den Inhalt des Registers `MDR`

In Listing 3.14 ist zu sehen, wie die *Mic-1* die drei Mikro-Assembler-Instruktionen zum Aufbau der Hauptspeicheradresse und zum Einlesen des Zeichens ausführt. Während der Ausführung fragt der *micro-debug* im Auftrag der *Mic-1* nach einer Eingabe, hier gebe ich wieder `1010` ein und bestätige mit `ENTER`.

Die *Mic-1* hat nun Zeile `0x6B` ausgeführt und das erste Zeichen gelesen. Dieses Zeichen wird allerdings erst im nächsten Zyklus in das Register `MDR` geschrieben – daher habe ich wie in Zeile 7 zu sehen eine weiteren Mikro-Assembler-Instruktion ausgeführt. Jetzt muss das gelesene Zeichen im Register `MDR` angekommen sein, was ich mit dem Befehl `ls-reg MDR` überprüfe. Wie ab Zeile Zeile 10 zu sehen, enthält das Register `MDR` den Wert $0x31 = 49$ – der *ASCII*-Code für das Zeichen `1`.

Der Fehler scheint demnach nicht im Mikro-Assembler-Code zu liegen – zumindest nicht, im `IN`-Befehl. Deswegen konzentriere ich mich jetzt auf den Assembler-Code. Mit dem Befehl `step` führe ich die letzte Mikro-Assembler-Instruktion des Befehls `IN` aus. Die *Mic-1* befindet sich nun in Zeile `0x5` des Assembler-Code– der Zeile 4 aus Listing 3.6.

Damit ich die Veränderungen der lokalen Variablen direkt erkenne, beobachte ich sie. In Listing 3.4 ist zu sehen, wie viele lokale Variablen vorhanden sind: zwei. Ich beobachte beide, um zu sehen, ob der *ASCII*-Code korrekt konvertiert wird und welche Werte das Ergebnis annimmt. In Listing 3.15 ist zu sehen, wie ich dazu den Befehl `trace-var` nutze.

```
1 micro-debug> trace-var 0
2 micro-debug> trace-var 1
```

Listing 3.15: Beobachten beider lokaler Variablen

Ich werde das Programm nun schrittweise ausführen – da ich auf die Korrektheit des Mikro-Assembler-Code vertraue, führe ich jeweils Assembler-Instruktionen aus. Dann überprüfe ich, ob das erwartete Verhalten eingetreten ist und suche so schrittweise die Ursache für den Fehler. Zur Erinnerung, die *Mic-1* hat gerade eine `1` eingelesen.

Mit dem Befehl `step` wird die nächste Assembler-Instruktion ausgeführt. In Listing 3.16 ist die Ausgabe dieses Befehls zu sehen – da ich den Mikro-Assembler-Code noch beobachte, ist diese Ausgabe unübersichtlich. Ich kann aber in Zeile 8 sehen, dass die lokale Variable `0` den korrekten Wert erhalten hat.

```
1  micro-debug> step
2  Executed: fetch;goto 0x4
3  Executed: PC=PC+1;goto 0x5
4  Executed: goto (MBR)
5  Executed: H=LV;fetch;goto 0x1F
6  Executed: MDR=TOS;goto 0x20
7  Executed: MAR=H+MBRU;wr;goto 0x21
8  Local variable 0: 49
9  Executed: SP=MAR=SP-1;rd;goto 0x22
10 Executed: PC=PC+1;goto 0x23
11 Executed: TOS=MDR;goto 0x3
12 Processor executed 9 ticks.
```

Listing 3.16: Ausführen einer Assembler-Instruktion bei Beobachten des Mikro-Assembler-Code

Damit ich im Folgenden übersichtlichere Ausgaben erhalte, beende ich das Beobachten des Mikro-Assembler-Code durch den Befehl `untrace-mic`. Zusätzlich Beobachten ich nun den Assembler-Code, um bei jeder ausgeführten Assembler-Instruktion überprüfen zu können, was gerade ausgeführt wurde. Nachdem ich den Befehl `trace-mac` eingegeben habe, kann ich das Programm weiter schrittweise ausführen.

```
1  micro-debug> step
2  Executed:      0x7: [ 0x15] ILOAD  0
3  Processor executed 8 ticks.
4  micro-debug> ls-stack
5  Stack value #1 [  0xC001]: 0x31
6  micro-debug> step
7  Executed:      0x9: [ 0x10] BIPUSH  0xA
8  Processor executed 6 ticks.
9  micro-debug> ls-stack
10 Stack value #1 [  0xC001]: 0x31
11 Stack value #2 [  0xC002]: 0xA
12 micro-debug> step
13 Executed:      0xB: [ 0x9F] IF_ICMPEQ  0x1B
14 Processor executed 11 ticks.
15 micro-debug> ls-stack
16 Stack doesn't contain any elements, nothing to display.
```

Listing 3.17: Ausführen der Assembler-Instruktionen zum Vergleich zweier lokaler Variablen

Listing 3.17 enthält die Konsolenausgabe nach der Ausführung der nächsten drei Assembler-Instruktionen. Da diese Assembler-Instruktionen auf dem Stack operieren, habe ich zusätzlich nach jedem Befehl den Stack angesehen. Das Programm prüft hier, ob die eingegebene Zahl ein Zeilenumbruch ist. Dafür legt es die eingegebene Zahl und einen Zeilenumbruch auf den Stack und ruft die Instruktion `IF_ICMPEQ` auf, die bei Gleichheit der beiden obersten Zahlen auf dem Stack an die angegebene Adresse springt.

An den Konsolenausgaben ist zu erkennen, dass zunächst beide Zahlen korrekt auf den Stack gelegt werden. Der Vergleich sollte die beiden Werte vom Stack entfernen und sie dann vergleichen; in Zeile 16 in Listing 3.17 sieht man, dass der Stack korrekterweise keine Elemente enthält. Wenn ich nun die nächste Assembler-Instruktion ausführe, kann ich an der Adresse der Assembler-Instruktion erkennen, ob der bedingte Sprung ausgeführt wurde.

```
1  micro-debug> step
2  Executed:      0xE: [ 0x84] IINC  0 0xD0
3  Local variable 0: 1
4  Processor executed 9 ticks.
5  micro-debug> step
6  Executed:     0x11: [ 0x15] ILOAD  1
7  Processor executed 8 ticks.
8  micro-debug> step
9  Executed:     0x13: [ 0x59] DUP
10 Processor executed 5 ticks.
11 micro-debug> step
12 Executed:     0x14: [ 0x60] IADD
13 Processor executed 6 ticks.
14 micro-debug> step
15 Executed:     0x15: [ 0x15] ILOAD  0
16 Processor executed 8 ticks.
17 micro-debug> step
18 Executed:     0x17: [ 0x60] IADD
19 Processor executed 6 ticks.
20 micro-debug> ls-stack
21 Stack value #1 [  0xC001]: 0x1
```

Listing 3.18: Ausführen der Assembler-Instruktionen zur Berechnung des temporären Ergebnisses

Der bedingte Sprung wurde nicht ausgeführt! Stattdessen wurde nun der *ASCII*-Code dekodiert und die lokale Variable enthält den Wert `1`. Im Folgenden wird das Programm das aktuelle Ergebnis – das derzeit `0` ist – mit zwei multiplizieren und die gerade gelesene Zahl dazu addieren.

Listing 3.18 zeigt die gerade beschriebenen Schritte: Zunächst wird das derzeitige Ergebnis geladen, auf dem Stack dupliziert und dann aufaddiert, was der Multiplikation mit zwei entspricht. Anschließend wird die gelesene Zahl auf den Stack gelegt und dazu addiert und sollte das aktuelle Ergebnis ergeben. Mit dem Befehl `ls-stack` kann ich nun sehen, dass diese Annahme korrekt ist: es liegt der Wert `1` auf dem Stack, wie in Zeile 21 in Listing 3.18 zu sehen.

Bisher scheint der Assembler-Code auch korrekt zu sein. In Listing 3.19 ist die Konsolenausgabe nach der Ausführung der nächsten Assembler-Instruktionen zu sehen. Hier führt die *Mic-1* den `GOTO`-Befehl aus und springt zu Adresse `0x4`, an der der `IN`-Befehl liegt.

```
1  micro-debug> step
2  Executed:     0x18: [ 0xA7] GOTO  0x4
3  Processor executed 8 ticks.
```

Listing 3.19: Ausführen des fehlerhaften Schritts

Vermutlich liegt der Fehler des Programms darin, das berechnete Ergebnis nicht in der lokalen Variable zu speichern. Wenn diese Vermutung stimmt, sollte der Stack pro Schleifendurchlauf wachsen, da das Ergebnis am Ende der Schleife auf dem Stack liegt, aber nicht mehr gelesen wird. Dies lässt sich durch die Beobachtung aus Listing 3.10 zumindest nicht widerlegen: der Stack enthielt am Ende des Programmlaufs viele Elemente.

Diese Vermutung soll nun näher überprüft werden. Dazu setze ich noch einen Breakpoint auf Zeile `0x18` des Assembler-Code, dem Schleifenende. Da die Zeile `0x4` unmittelbar nach der Zeile `0x18` ausgeführt wird, kann ich den Breakpoint in Zeile `0x4` entfernen.

Listing 3.20 zeigt das Setzen des Breakpoints in Zeile `0x18`; anschließend ist die Anzeige aller Breakpoints zu sehen. In Zeile 5 wird die lokale Variable `0` beobachtet, die das gelesene Zeichen enthält. Der Vermutung nach wird die lokale Variable `1`, die das Ergebnis enthalten sollte, nie beschrieben; um meine Vermutung eventuell verwerfen zu können, beobachte ich auch diese in Zeile Zeile 6.

Anschließend entferne ich den alten Breakpoint aus Zeile `0x4` – die Nummer, die ich dabei angebe, ist die Nummer des Breakpoints, die der Befehl `ls-break` liefert. Nachdem nun die verschiedenen Werte beobachtet werden und ein neuer Breakpoint gesetzt ist, kann mit dem `RUN`-Befehl jeweils ein Schleifendurchlauf bis zum Ende des Programms ausgeführt werden.

```
micro-debug> macro-break 0x18
micro-debug> ls-break
Breakpoint #1: at macro code line 0x4
Breakpoint #2: at macro code line 0x18
micro-debug> trace-var 0
micro-debug> trace-var 1
micro-debug> rm-break 1
micro-debug> ls-break
Breakpoint #2: at macro code line 0x18
```

Listing 3.20: Setzen eines Breakpoints und Beobachten der lokalen Variablen

```
1  micro-debug> run
2  Executed:      0x4: [ 0xFC] IN
3  Executed:      0x5: [ 0x36] ISTORE  0
4  Local variable 0: 48
5  Executed:      0x7: [ 0x15] ILOAD  0
6  Executed:      0x9: [ 0x10] BIPUSH  0xA
7  Executed:      0xB: [ 0x9F] IF_ICMPEQ  0x1B
8  Executed:      0xE: [ 0x84] IINC  0 0xD0
9  Local variable 0: 0
10 Executed:     0x11: [ 0x15] ILOAD  1
11 Executed:     0x13: [ 0x59] DUP
12 Executed:     0x14: [ 0x60] IADD
13 Executed:     0x15: [ 0x15] ILOAD  0
14 Executed:     0x17: [ 0x60] IADD
15 Processor executed 84 ticks.
```

Listing 3.21: Schleifendurchlauf Nummer 2

Listing 3.21 enthält die Konsolenausgabe des nächsten – dem zweiten – Schleifendurchlauf. In Zeile 4 gibt der *micro-debug* aus, dass der Wert `48` korrekt der lokalen Variable `0` zugewiesen wurde. Die `48` ist der *ASCII*-Code für `0`, was das zweite Zeichen meiner Eingabe `1010` war.

Ab Zeile 5 wird überprüft, ob das gelesene Zeichen ein Zeilenumbruch ist. Auch hier ist korrekt, dass ab Zeile 8 das gelesene Zeichen in den Zahlenwert konvertiert wird und ab Zeile 10 das Ergebnis berechnet wird.

```
micro-debug> ls-stack
Stack value #1 [  0xC001]: 0x1
Stack value #2 [  0xC002]: 0x0
```

Listing 3.22: Stack nach Schleifendurchlauf Nummer 2

In Listing 3.22 ist der aktuelle Stack nach dem jetzigen zweiten Schleifendurchlauf zu sehen. Wie erwartet ist er um ein Element angewachsen, dass nun `0` ist. Da das Ergebnis nie in der lokalen Variable abgelegt wird, ist in der Schleife das alte Ergebnis `0`, welches zur Berechnung des aktuellen Ergebnisses verwendet wird. Wenn das bisherige Ergebnis `0` ist, dann entspricht das aktuelle Ergebnis gerade der gelesenen Zahl.

Aus der Vermutung über die Fehlerursache kann man den Schluss ziehen, dass am Ende der Stack exakt die eingelesenen Ziffern enthalten sollte. Auch diese Schlussfolgerung konnte bereits in Listing 3.10 beobachtet werden. Daher akzeptiere ich nun meine Vermutung und versuche den Assembler-Code zu korrigieren; Listing 3.23 zeigt die korrigierte Version des Assembler-Code.

Die Korrektur besteht darin die Zeile Zeile 20 einzufügen – das Speichern des aktuellen Ergebnisses in der lokalen Variable.

Nachdem die korrigierte Version des Assembler-Code kompiliert ist, kann ich sie debuggen und überprüfen, ob die Korrektur erfolgreich war. Der *micro-debug* kann den Mikro-Assembler- und Assembler-Code nicht automatisch aktualisieren und muss daher mit dem Befehl `EXIT` beendet werden. Anschließend kann er mit dem korrigierten Kompilat gestartet werden.

```
1  .main
2  .var
3      c
4      result
5  .end-var
6      bipush      0
7      istore      result
8  loop:
9      in
10     istore      c
11     iload       c
12     bipush      10
13     if_icmpeq   finish
14     iinc        c       -48
15     iload       result
16     dup
17     iadd
18     iload       c
19     iadd
20     istore      result
21     goto        loop
22 finish:
23     iload       result
24     halt
25 .end-main
```

Listing 3.23: IJVM-Assembler zum Einlesen einer Binärzahl (korrigiert)

Mit dem Befehl `ls-macro-code` kann nun überprüft werden, ob der Assembler-Code korrekt aktualisiert wurde. In Listing 3.24 ist die Ausgabe des Befehls zu sehen – Zeile 15 zeigt die neu eingefügte Zeile.

```
1  micro-debug> ls-macro-code
2       0x0: [ 0x10] BIPUSH  0x0
3       0x2: [ 0x36] ISTORE  1
4       0x4: [ 0xFC] IN
5       0x5: [ 0x36] ISTORE  0
6       0x7: [ 0x15] ILOAD  0
7       0x9: [ 0x10] BIPUSH  0xA
8       0xB: [ 0x9F] IF_ICMPEQ  0x1D
9       0xE: [ 0x84] IINC  0 0xD0
10     0x11: [ 0x15] ILOAD  1
11     0x13: [ 0x59] DUP
12     0x14: [ 0x60] IADD
13     0x15: [ 0x15] ILOAD  0
14     0x17: [ 0x60] IADD
15     0x18: [ 0x36] ISTORE  1
16     0x1A: [ 0xA7] GOTO  0x4
17     0x1D: [ 0x15] ILOAD  1
18     0x1F: [ 0xFF] HALT
```

Listing 3.24: Disassemblierter Assembler-Code (korrigiert)

Ich habe nun überprüft, dass der Assembler-Code korrekt aktualisiert wurde, daher kann ich jetzt überprüfen, ob die Vermutung korrekt war und der Fehler im Assembler-Code nun behoben ist.

```
1  micro-debug> trace-mac
2  micro-debug> trace-var 0
3  micro-debug> trace-var 1
```

Listing 3.25: Beobachten des Assembler-Code und der lokalen Variablen

Beim Beenden des *micro-debug* werden Breakpoints sowie beobachtete Register, Variablen oder Assembler-Code verworfen – zur leichteren Übersicht habe ich diese daher nochmal aktiviert. Die Befehle dazu sind in Listing 3.25 aufgeführt. Ein Breakpoint werde ich jetzt nicht

mehr benötigen; ich werde das Programm ausführen und die Ausgaben analysieren. Listing 3.26 und Listing 3.27 enthalten die Konsolenausgaben bei der Ausführung des Assembler-Code; wieder habe ich `1010` eingegeben.

```
micro-debug> run
Executed:      0x0: [ 0x10] BIPUSH  0x0
Executed:      0x2: [ 0x36] ISTORE  1
Executed:      0x4: [ 0xFC] IN
mic1> 1010
Executed:      0x5: [ 0x36] ISTORE  0
Local variable 0: 49
Executed:      0x7: [ 0x15] ILOAD  0
Executed:      0x9: [ 0x10] BIPUSH  0xA
Executed:      0xB: [ 0x9F] IF_ICMPEQ  0x1D
Executed:      0xE: [ 0x84] IINC  0 0xD0
Local variable 0: 1
Executed:     0x11: [ 0x15] ILOAD  1
Executed:     0x13: [ 0x59] DUP
Executed:     0x14: [ 0x60] IADD
Executed:     0x15: [ 0x15] ILOAD  0
Executed:     0x17: [ 0x60] IADD
Executed:     0x18: [ 0x36] ISTORE  1
Local variable 1: 1
Executed:     0x1A: [ 0xA7] GOTO  0x4
Executed:      0x4: [ 0xFC] IN
Executed:      0x5: [ 0x36] ISTORE  0
Local variable 0: 48
Executed:      0x7: [ 0x15] ILOAD  0
Executed:      0x9: [ 0x10] BIPUSH  0xA
Executed:      0xB: [ 0x9F] IF_ICMPEQ  0x1D
Executed:      0xE: [ 0x84] IINC  0 0xD0
Local variable 0: 0
Executed:     0x11: [ 0x15] ILOAD  1
Executed:     0x13: [ 0x59] DUP
Executed:     0x14: [ 0x60] IADD
Executed:     0x15: [ 0x15] ILOAD  0
Executed:     0x17: [ 0x60] IADD
Executed:     0x18: [ 0x36] ISTORE  1
Local variable 1: 2
Executed:     0x1A: [ 0xA7] GOTO  0x4
Executed:      0x4: [ 0xFC] IN
Executed:      0x5: [ 0x36] ISTORE  0
Local variable 0: 49
Executed:      0x7: [ 0x15] ILOAD  0
Executed:      0x9: [ 0x10] BIPUSH  0xA
Executed:      0xB: [ 0x9F] IF_ICMPEQ  0x1D
Executed:      0xE: [ 0x84] IINC  0 0xD0
Local variable 0: 1
Executed:     0x11: [ 0x15] ILOAD  1
Executed:     0x13: [ 0x59] DUP
Executed:     0x14: [ 0x60] IADD
Executed:     0x15: [ 0x15] ILOAD  0
Executed:     0x17: [ 0x60] IADD
Executed:     0x18: [ 0x36] ISTORE  1
Local variable 1: 5
Executed:     0x1A: [ 0xA7] GOTO  0x4
Executed:      0x4: [ 0xFC] IN
Executed:      0x5: [ 0x36] ISTORE  0
Local variable 0: 48
Executed:      0x7: [ 0x15] ILOAD  0
Executed:      0x9: [ 0x10] BIPUSH  0xA
Executed:      0xB: [ 0x9F] IF_ICMPEQ  0x1D
Executed:      0xE: [ 0x84] IINC  0 0xD0
Local variable 0: 0
Executed:     0x11: [ 0x15] ILOAD  1
Executed:     0x13: [ 0x59] DUP
Executed:     0x14: [ 0x60] IADD
Executed:     0x15: [ 0x15] ILOAD  0
Executed:     0x17: [ 0x60] IADD
Executed:     0x18: [ 0x36] ISTORE  1
Local variable 1: 10
```

Listing 3.26: Konsolenausgabe der Ausführung des korrekten Assembler-Codes – Teil 1

Der Assembler-Code ist nun korrekt: Der Stack und das Register `TOS` enthalten am Ende das Ergebnis – `0xA`.

```
Executed:     0x1A: [ 0xA7] GOTO  0x4
Executed:      0x4: [ 0xFC] IN
Executed:      0x5: [ 0x36] ISTORE  0
Local variable 0: 10
Executed:      0x7: [ 0x15] ILOAD  0
Executed:      0x9: [ 0x10] BIPUSH  0xA
Executed:      0xB: [ 0x9F] IF_ICMPEQ  0x1D
Executed:     0x1D: [ 0x15] ILOAD  1
Executed:     0x1F: [ 0xFF] HALT
Processor executed 477 ticks.
micro-debug> ls-stack
Stack value #1 [  0xC001]: 0xA
micro-debug> ls-reg TOS
Register TOS : 0xA
```

Listing 3.27: Konsolenausgabe der Ausführung des korrekten Assembler-Codes – Teil 2

4. Interaktion per GUI

Im zweiten Teil der Aufgabe sollte eine GUI für den *micro-debug* entwickelt werden – die *micro-debug-gui*. Wie wird diese bedient? Worauf ist zu achten? Was unterscheidet sich die *micro-debug-gui* vom *micro-debug*? Diese Fragen möchte ich in diesem Kapitel beantworten.

Auch die *micro-debug-gui* erfordert keine Installation – die *micro-debug-gui* ist nach dem Entpacken der `.zip`-Datei unmittelbar ausführbar. Die Schritte vom Herunterladen bis zum Starten der *micro-debug-gui* sind dabei analog zum *micro-debug*:

1. Die Datei `micro-debug-gui-version.zip` von der Projektseite [Rö12b] herunterladen
2. Die `.zip`-Datei in ein beliebiges Verzeichnis entpacken (beispielsweise `/opt/micro-debug-gui/`)
3. Das Verzeichnis der *micro-debug-gui* dem `PATH` hinzufügen
4. Die *micro-debug-gui* starten – mit `micro-debug-gui.sh --help`

Die *micro-debug-gui* ist technisch eine eigenständige Anwendung, die den *micro-debug* benutzt – wie ich in Kapitel 9 erläutere. Der Code des *micro-debug* ist daher in der *micro-debug-gui* enthalten und kann mit wenigen Anpassungen ohne GUI benutzt werden. Dies möchte ich nun kurz genauer erklären.

4.1. Nutzung der Konsolenvariante

Nach dem Entpacken der `.zip`-Datei entsteht eine ähnliche Verzeichnis-Struktur, wie beim *micro-debug*. Allerdings existiert nun das Verzeichnis `lib/`, das unter anderem eine `.jar`-Datei beinhaltet, die den Code des *micro-debug* enthält – die Datei `micro-debug-version.jar`.

Ich möchte nun zeigen, wie man die *micro-debug-gui* so ergänzt, dass sowohl die *micro-debug-gui* als auch der *micro-debug* aufrufbar sind. Da dieses Verhalten für Windows und Linux analog ist, zeige ich es am Beispiel von Linux. Listing 4.1 enthält die Kommandos, die für diese Ergänzung auf der Konsole nötig sind.

```
1 cd micro-debug-gui-version/
2 cp micro-debug-gui.sh micro-debug.sh
```

Listing 4.1: Parallele Nutzung des *micro-debug* und der *micro-debug-gui*

Als Basis für die parallele Nutzung kann das Startskript der *micro-debug-gui* kopiert werden; diese Kopie (`micro-debug.sh`) musst aber noch angepasst werden:

1. Der Text `"$DIR/micro-debug-gui-versiong.jar":"$DIR/lib/*"` muss durch `"$DIR/lib/micro-debug-versionk.jar"` ersetzt werden – *versiong* ist dabei die Version der *micro-debug-gui* und *versionk* die Version des *micro-debug*.

   Hierdurch wird die Bibliothek geändert, die den auszuführenden Programmcode enthält.
2. Auch die Startklasse `com.github.croesch.micro_debug.gui.MicroDebug` muss durch `com.github.croesch.micro_debug.MicroDebug` ersetzt werden.

Nach diesen Schritten sind zwei Startskripte vorhanden: Eines zum Start der *micro-debug-gui* und eines zum Start des *micro-debug*. Beide können nun unabhängig voneinander ausgeführt werden.

4.2. Unterschiede zur Konsolenvariante

Die Struktur der *micro-debug-gui* ist ähnlich zur Struktur des *micro-debug*, aber nicht identisch. Auch beim Aufruf und bei der Bedienung gibt es einige nennenswerte Unterschiede, die ich nun aufzeigen möchte.

4.2.1. Parameter

Bereits der Aufruf der *micro-debug-gui* unterscheidet sich von dem des *micro-debug*: Es gibt weniger Parameter und die *micro-debug-gui* kann auch ohne Parameter aufgerufen werden. Alle der im Folgenden aufgeführten Parameter sind optional.

-h, --help mit diesem Parameter kann, wie beim *micro-debug*, die Hilfe angezeigt werden lassen, die neben den verschiedenen Aufrufmöglichkeiten die möglichen Parameter erklärt. Zusätzlich werden auch hier noch einige ergänzende Informationen angezeigt.

-v, --version gibt die Version der *micro-debug-gui* und des benutzten *micro-debug* aus.

Wie der *micro-debug* startet auch die *micro-debug-gui* nicht, wenn die Hilfe oder Version angezeigt werden soll, sondern beendet sich nach Ausgabe der Informationen.

Anders als der *micro-debug* benötigt die *micro-debug-gui* die Pfade zu den Bytecode-Dateien nicht; diese werden direkt zu Beginn über die GUI eingegeben. Dies erkläre ich nochmal genauer im Abschnitt 4.3.1.

4.2.2. Verfügbare Funktionen

Die *micro-debug-gui* benutzt den *micro-debug* und hat daher prinzipiell Zugriff auf alle Funktionen, die der *micro-debug* dem Benutzer bereitstellt. Da die *micro-debug-gui* dem Benutzer aber fortlaufend die Werte der Register und des Hauptspeichers, den Mikro-Assembler- sowie den Assembler-Code anzeigt, ist es nicht nötig diese Werte beobachten zu können. Befehle wie `trace-reg` sind daher in der *micro-debug-gui* nicht wiederzufinden. Auch die Ausgabe des Stacks gibt es nicht mehr, denn dieser kann im Hauptspeicher angesehen werden.

Die Breakpoints werden dem Benutzer fortlaufend angezeigt und können über Mausklicks aktiviert und deaktiviert werden, daher fallen Befehle wie `ls-break` auch weg.

Die *micro-debug-gui* bietet demzufolge konstruktionsbedingt manche Funktionen nicht explizit an. Es gibt aber auch manche Funktionen, die bisher noch nicht über die *micro-debug-gui* abgebildet sind, diese sind:

- Einen Breakpoint für einen bestimmten Wert eines Registers zu setzen, wie beispielsweise per `break Register Wert`.
- eine gewisse Anzahl an (Mikro-)Instruktionen auszuführen, wie per `micro-step Zahl`.
- den Wert eines Registers zu verändern, wie per `set Register Zahl`.
- den Wert eines Wortes im Hauptspeicher zu setzen, wie per `set-mem Zahl1 Zahl2`.

Diese Funktionen habe ich bisher aus Zeitgründen noch nicht implementiert, werde ich aber nach und nach ergänzen.

Allerdings bietet die *micro-debug-gui* dem Benutzer im Gegensatz zum *micro-debug* die Möglichkeit, die laufende *Mic-1* zu unterbrechen. Wie ich in Abschnitt 9.1.3 erkläre, nutzt die

micro-debug-gui zur Ausführung der *Mic-1* einen eigenen *Thread*. Über die GUI kann dieser Thread dann mit Hilfe des EDT (Event Dispatch Thread) unterbrochen werden – prinzipiell ist diese Funktion wie ein unbedingter Breakpoint.

4.2.3. Geschwindigkeit

Die *micro-debug-gui* nutzt die Funktionen des *micro-debug*, um die *Mic-1* simulieren und debuggen zu können. Folglich kann die *micro-debug-gui* prinzipiell nicht schneller sein als der *micro-debug*, sondern eher langsamer.

Diese Vermutung bestätigte sich bei ersten Tests der *micro-debug-gui*. Bei einfachen Befehlen, wie eine (Mikro-)Instruktion auszuführen oder die *Mic-1* zurückzusetzen war kein Geschwindigkeitsunterschied beobachtbar. Aber besonders bei dem `run`-Befehl war deutlich zu beobachten, dass die *micro-debug-gui* mehr Zeit benötigte.

Was machte die *micro-debug-gui* besonders langsam? Wie konnte dieses Problem gelöst werden? Die Ursache der langsamen Ausführung war die Aktualisierung der GUI: Die Werte der Register und des sichtbaren Hauptspeichers, sowie die aktuell ausgeführte Zeile wurden ständig aktualisiert.

Ständig bedeutet, dass nach jedem Zyklus der *Mic-1* die GUI aktualisiert wurde. Ich habe daraufhin eine Konfigurationsoption – `gui.update.after.each.tick` – eingeführt; wird diese Option auf `true` gesetzt, aktualisiert die *micro-debug-gui* die GUI nach jedem Zyklus der *Mic-1*, eine deutlich längere Ausführungszeit kann beobachtet werden. Daher ist diese Option standardmäßig auf `false` gesetzt.

4.3. Oberflächenelemente

Nachdem ich nun einige Eigenschaften der *micro-debug-gui* beschrieben habe, möchte ich jetzt einen detaillierteren Einblick in die *micro-debug-gui* geben. Dazu werde ich die verschiedenen Hauptkomponenten der *micro-debug-gui* zeigen und deren Funktionen erklären.

4.3.1. Startfenster

Wie ich in Abschnitt 4.2.1 beschrieben habe, werden der *micro-debug-gui* die Pfade der Bytecode-Dateien nicht als Parameter übergeben – es gibt ein extra Fenster dafür. Dieses Fenster erscheint unmittelbar nach dem Start der *micro-debug-gui* und ist in Abbildung 4.1 zu sehen.

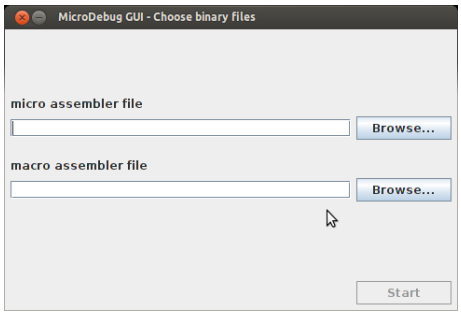

Abbildung 4.1.: Screenshot des Start-Fensters

Wenn zwei Pfade eingegeben sind, kann die *Mic-1* dann über den Start-Button initialisiert werden und die *micro-debug-gui* ist funktionsbereit. Bei der Initialisierung der *Mic-1* werden die beiden Bytecode-Dateien auf ihre Gültigkeit geprüft; zuerst die Mikro-Assembler-Datei und dann die Assembler-Datei. Falls eine der Dateien ungültig ist, wird statt dem Hauptfenster der

micro-debug-gui erneut das Start-Fenster gezeigt. In diesem Fall ist dann das Textfeld mit der ungültigen Datei leer, wie beispielsweise in Abbildung 4.2 zu sehen.

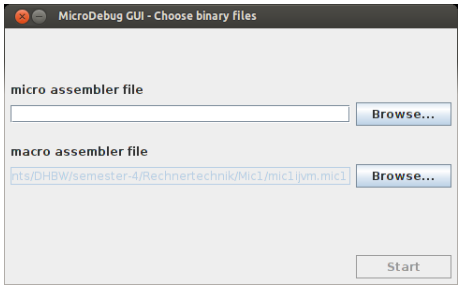

Abbildung 4.2.: Screenshot des Start-Fensters – die angegebenen Dateien waren *beide* ungültig

In diesem Beispiel waren beide Dateien ungültig, aber nur ein Textfeld ist leer. Warum? Die *micro-debug-gui* prüft die Dateien nacheinander, ist die erste Datei bereits ungültig, wird die zweite nicht geprüft und zunächst als gültig angenommen. Das Textfeld der zweiten Datei ist dann inaktiv, kann aber mit einem Doppelklick wieder aktiviert und dadurch editiert werden. In dem genannten Beispiel habe ich das getan und anschließend zwei korrekte Dateien eingetragen, wie in Abbildung 4.3 abgebildet ist.

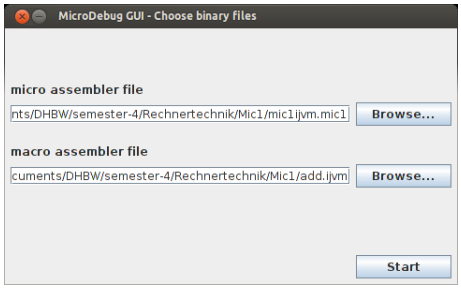

Abbildung 4.3.: Screenshot des Start-Fensters mit zwei eingetragenen Dateien

Wurden zwei gültige Dateien eingetragen, startet die *micro-debug-gui* und öffnet das Hauptfenster aus Abbildung 4.4.

4.3.2. Register

In Abbildung 4.4 sind links oben die Register mit ihren Werten zu sehen; die gleiche Ansicht ist in Abbildung 4.6 nochmal größer dargestellt.

Die Checkboxen repräsentieren die Breakpoints für die Register, in Abbildung 4.6 hält die *Mic-1* demnach an, wenn `MBR` oder `MDR` im nächsten Zyklus geschrieben wird. Über die Radiobuttons unten kann das Zahlenformat der Registerwerte geändert werden: auf binär, dezimal oder hexadezimal. So kann der Benutzer für jeden Zweck die optimale Darstellung der Registerwerte wählen.

4.3.3. Hauptspeicher

Unter der Registeransicht befindet sich im Hauptfenster die Ansicht des Hauptspeichers. Hier kann wie bei der Registeransicht das Zahlenformat der Hauptspeicherwerte jederzeit geändert werden.

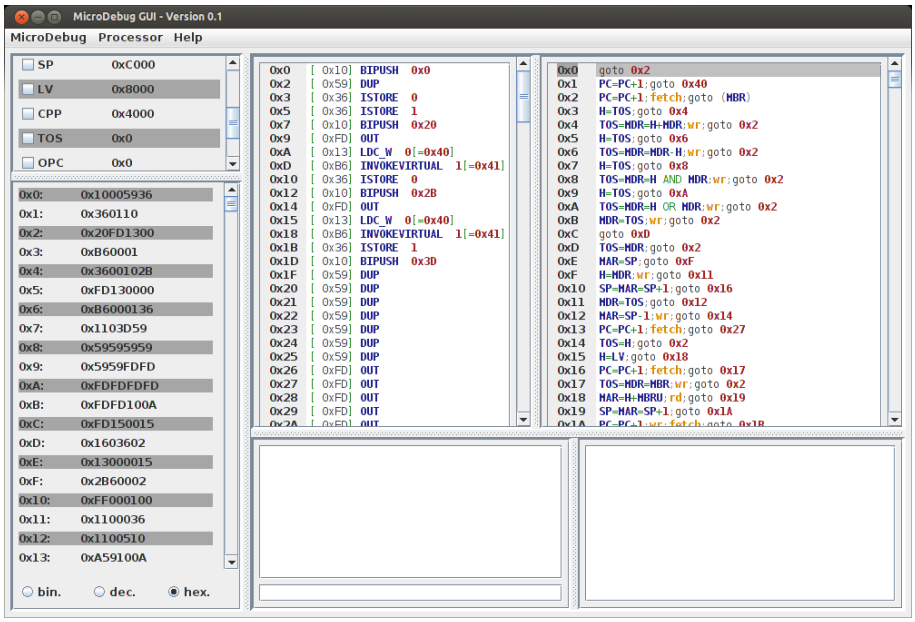

Abbildung 4.4.: Hauptfenster der *micro-debug-gui* zu Beginn

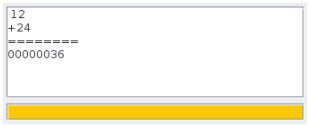

Abbildung 4.5.: Textkomponenten zur Ein- und Ausgabe der *Mic-1* im Hauptfenster der *micro-debug-gui*

In der Ansicht des Hauptspeichers sind nur eine feste Anzahl an Einträgen zu sehen, die sich nicht dynamisch an die Größe des sichtbaren Bereichs anpassen. Dies habe ich wegen der besseren Performance so gelöst.

Die Ansicht zeigt den gesamten Hauptspeicher – neben den lokalen Variablen, dem Stack und den Konstanten kann dadurch auch der disassemblierten Assembler-Code in Bit-Form betrachtet werden.

4.3.4. Text-Ein- und -Ausgabe

Rechts neben der Hauptspeicheransicht sind im Hauptfenster unten verschiedene Textkomponenten zu sehen. Deren Funktion sollte spätestens im laufenden Betrieb der *micro-debug-gui* deutlich werden: Links unten ist die Textkomponente, um der *Mic-1* Text bereitzustellen. Darüber befindet sich die Textkomponente, die die Ausgaben der *Mic-1* anzeigt und im rechten Bereich ist die Textkomponente, die Informationen des *micro-debug* anzeigt, die sonst auf der Konsole ausgegeben würden und bisher noch keinen Platz in der Oberfläche bekommen haben.

Abbildung 4.5 zeigt, wie sich das Eingabefeld für die *Mic-1* verhält, wenn die *Mic-1* per `IN` ein Zeichen liest, aber der Puffer des *micro-debug* keine Zeichen mehr enthält: es wird orange gefärbt. In diesem Fall sollte der nächste Text für die *Mic-1* eingegeben und mit *ENTER* bestätigt werden. Dadurch werden die eingegebenen Zeichen inklusive Zeilenumbruch im *micro-debug* gepuffert, bis sie die *Mic-1* einliest.

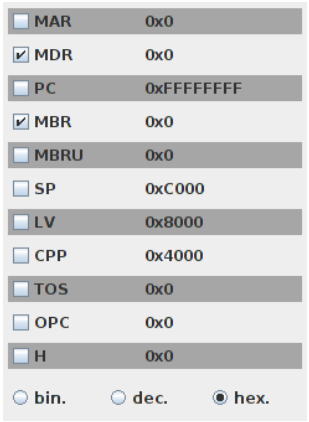

Abbildung 4.6.: Registeransicht im Hauptfenster der *micro-debug-gui*

Es können durch diese Textkomponente auch Eingaben gemacht werden, wenn der Puffer des *micro-debug* noch nicht leer ist. In diesem Fall wird der Text, der eingegeben wird, dem Puffer des *micro-debug* angehängt. So können schon zu Beginn alle nötigen Texteingaben gemacht werden und anschließend der Assembler-Code ohne Unterbrechung ausgeführt werden.

4.3.5. Code

Die wohl wichtigsten zwei Komponenten befinden sich in Abbildung 4.4 rechts oben: Der disassemblierte Mikro-Assembler- und Assembler-Code. Die beiden Ansichten sind von der Funktion identisch aufgebaut und zeigen den disassemblierten Code mit Syntaxhervorhebung, Zeilennummern und den dazugehörigen Breakpoints an. Zusätzlich wird die Zeile des Codes hervorgehoben, die als nächstes ausgeführt wird.

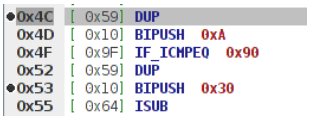

Abbildung 4.7.: Ausschnitt des Assembler-Code im Hauptfenster der *micro-debug-gui*

Abbildung 4.7 zeigt einen Ausschnitt der Code-Ansicht. Hier ist zu sehen, dass als nächstes die Zeile `0x4C` abgearbeitet werden wird und dass in den Zeilen `0x4C` und `0x53` Breakpoints gesetzt sind. Die Breakpoints werden durch einen Doppelklick gesetzt und durch einen Doppelklick können sie wieder entfernt werden. Der Doppelklick muss an der Stelle ausgeführt werden, an der die Breakpoints gezeichnet werden – links neben den Zeilennummern.

Teil II.

Implementierung

5. Werkzeuge

In diesem Teil der Arbeit möchte ich besonders eine Frage beantworten: Wie sind die bisher vorgestellten Funktionen des Debuggers implementiert? Mein Ziel ist, die Implementierung und Struktur des Debuggers so nachvollziehbar zu beschreiben, dass Leser mit Programmiererfahrung in der Lage sind, den Debugger weiterzuentwickeln.

Der Debugger soll auch nach Ende dieser Arbeit weiterentwickelt werden können. Daher stelle ich in diesem Kapitel vor, welche Werkzeuge ich zur Entwicklung des Debuggers nutze und wie diese zur Mitarbeit genutzt werden können. Dies ist nicht als Bedienungsanleitung zu verstehen! Ich bin aber der Ansicht, dass es das Verständnis für die Implementierung und meine Arbeit steigert, wenn geklärt ist, welche Werkzeuge ich wie nutze.

5.1. Versionskontrollsystem

Die Projekte *micro-debug* und *micro-debug-gui* sind bei *GitHub* (siehe [Rö12a, Rö12b]) gehostet und sind somit mit git versioniert.

git wird beispielsweise in [Ver] beschrieben und ist demnach:

effizient Auch bei großen Projekten zeigen Vergleiche, dass git insgesamt schneller ist, als beispielsweise Subversion.

verteilt Jeder Entwickler erhält ein lokales Repository und kann damit arbeiten. Es wird kein zentraler Server benötigt, alle Funktionen des Systems und alle Versionen stehen lokal zur Verfügung.

sicher Bei git wird jeder Commit gehasht. Es ist praktisch nicht möglich einen Commit zu manipulieren, da dies die Hash-Summe verändern würde. Auch wenn theoretisch Kollisionen der Hash-Summe möglich sind, ist bisher eine solche Kollision noch nicht konstruierbar – bei einem Code-Projekt müsste eine solche Kollision zudem kompilierbarer Code sein, was als hinreichend unwahrscheinlich gilt.

Ich habe git gewählt, da dadurch später leicht *forks*[1] gebildet werden können und der Debugger durch Außenstehende weiterentwickelt werden kann. Das dezentrale Entwickeln mit git ermöglicht es zusätzlich, dass später viele Entwickler am Debugger mitentwickeln.

Wie checkt man mit git den Quelltext der Arbeit aus? Wie unterstützt mich git, wenn ich das Projekt weiterentwickeln möchte? Im Folgenden möchte ich auf diese Fragen antworten. Ich möchte keine umfassende Bedienungsanleitung für git geben, sondern die Grundlagen für die alltägliche Arbeit mit git vermitteln. git am *micro-debug* erklären, für *micro-debug-gui* ist das Vorgehen analog, in den meisten Fällen muss lediglich ein `-gui` eingefügt werden.

5.1.1. Repository klonen und Code auschecken

Die Adresse des öffentlichen Repositorys lautet:
`https://github.com/croesch/micro-debug.git`

[1] Zu deutsch Abspaltung, ist ein Entwicklungszweig, nachdem sich ein Software-Projekt in zwei oder mehr Teile geteilt hat. Diese Teile werden meist unabhängig voneinander weiterentwickelt. Im täglichen Arbeitsablauf mit git ist ein fork aber auch die Möglichkeit Neuerungen für ein Projekt zu Entwickeln und den entstandenen Entwicklungszweig nach Fertigstellung der Neuerung in das Ursprungsprojekt einzupflegen.

```
1 git clone git://github.com/croesch/micro-debug.git
2 cd micro-debug
```

Listing 5.1: *micro-debug* mit git klonen

Da git ein verteiltes Versionskontrollsystem ist, muss das entfernte Repository zunächst lokal angelegt werden – das wird klonen genannt. Auschecken bezeichnet den Vorgang, den Inhalt aus dem lokalen Repository im Arbeitsverzeichnis zu realisieren, wo er bearbeitet werden kann.

Klont man ein entferntes Repository, beispielsweise das Repository des *micro-debug*, wird im aktuellen Verzeichnis ein Verzeichnis `micro-debug` angelegt. In diesem Verzeichnis wird standardmäßig die Kopie des entfernten Repositorys abgelegt – im Verzeichnis `micro-debug/.git`. Außerdem wird der Inhalt des Repositorys ausgecheckt und befindet sich anschließend im Verzeichnis `micro-debug`. Listing 5.1 zeigt, welcher Befehl zum Klonen (und implizit Auschecken) des *micro-debug*-Repositorys verwendet wird – in Zeile 2 wird in das erzeugte Verzeichnis gewechselt, das Arbeitsverzeichnis.

Im Gegensatz zu Subversion erstellt git nur im Wurzelverzeichnis eines Projekts ein Verzeichnis `.git`. Direkt nach dem Klonen eines anderen Repositorys enthält es beispielsweise einen Verweis (*origin*) auf dieses Repository. Sobald das geklonte Repository neuere Commits enthält, kann dieser Verweis genutzt werden, um wie in Listing 5.2 die neuen Commits in das lokale Repository zu übernehmen.

```
1 git pull origin
```

Listing 5.2: Mit git *pull* auf Originalrepository ausführen

Mit dem Befehl `pull` werden die neuen Commits heruntergeladen, in das lokale Repository übernommen und ausgecheckt.

Dieses Vorgehen ist aufgrund der Änderung des aktuellen Arbeitsverzeichnisses häufig unerwünscht, daher kann man den Befehl aus Listing 5.2 wie in Listing 5.3 in zwei Befehle aufteilen, um zu regeln, ob und welcher Branch die Neuerungen erhält.

```
1 git fetch origin
2 git merge origin/master
```

Listing 5.3: *pull* in zwei Befehlen manuell ausführen

Mit *fetch* in Zeile 1 werden die Aktualisierungen heruntergeladen und zunächst im lokalen Repository gespeichert. Erst der Befehl *merge* in Zeile 2 verändert die Dateien im Arbeitsverzeichnis.

Es gibt bei git auch einen expliziten `checkout`-Befehl, der verwendet wird, um beispielsweise zwischen verschiedenen Branches zu wechseln.

5.1.2. Code beitragen

Hat man ein Repository geklont und möchte Codeänderungen dem Originalrepository zuführen, gibt es zwei Möglichkeiten: Man benutzt den Standard-Branch oder so genannte Feature-Branches.

```
1 git add relativer-pfad-zur-datei
2 git commit -m "Commit-Nachricht"
```

Listing 5.4: Eine Datei mit git committen

Den Standard-Branch (`master`) zu nutzen, empfehle ich nur, wenn man alleine an einem Projekt entwickelt. Listing 5.4 zeigt die Befehle, um eine veränderte Datei zu committen. Dieser Commit wirkt sich auf den aktuellen Branch aus und kann mit dem `push`-Befehl in das geklonte Repository transferiert werden. Das entfernte Repository muss dazu den gleichen Stand haben, wie das lokale Repository ohne den zu veröffentlichenden Commit.

Da dies insbesondere bei mehrköpfigen Entwicklungsteams eine Herausforderung ist, empfehle ich Feature-Branches zu nutzen. Durch folgende Schritte kann jeder Leser selbst Code zum *micro-debug* beitragen und diesen veröffentlichen.

1. Unter `https://github.com/croesch/micro-debug/issues` sind die Bugs und offenen Aufgaben zu finden. Möchte jemand Code beitragen, vermerkt er dies an dem jeweiligen Thema, oder öffnet ein neues, falls kein passendes Thema gefunden werden kann.

2. Im lokalen Repository wird nun ein so genannter Feature-Branch angelegt, in dem der Code geändert werden wird. Listing 5.5 enthält den Befehl, mit dem ein neuer Branch im lokalen Repository angelegt und ausgecheckt wird – eine Konvention ist, dass die Nummer des Branches der Nummer des Themas entspricht.

```
1 git checkout -b 100-themen-titel
```

Listing 5.5: Einen Branch mit git erstellen

3. Der entsprechende Code kann nun geändert und wie in Listing 5.4 committet werden – die Commits betreffen den neu erstellten Branch.

4. Regelmäßig oder zumindest am Ende der Änderungen sollte der aktuelle Stand des Originalrepositorys geladen werden. Dazu empfehle ich den Branch `master` auszuchecken und die aktuellen Commits des Originalrepositorys dort einzupflegen – wie in Listing 5.6 zu sehen.

```
1 git checkout master
2 git pull origin master
```

Listing 5.6: Mit git den Branch `master` auschecken und aktualisieren

5. Nun sollte der neu erstellte Branch so verändert werden, dass seine Commits auf dem aktuellen Stand des Repositorys aufsetzen. Dies kann mit dem Befehl `rebase` erledigt werden, wie in Listing 5.7 beschrieben. Das auschecken des neuen Branches ist nötig, da `rebase` den aktuellen Branch so verändert, dass er auf dem angegebenen Branch aufsetzt; in diesem Fall `master`.

```
1 git checkout 100-themen-titel
2 git rebase master
```

Listing 5.7: Mit git ein `rebase` auf einen Branch ausführen

6. Zum Schluss ist das lokale Repository in einem Zustand, der veröffentlicht werden kann. Listing 5.8 zeigt, wie nun der neu erstellte Branch an das Originalrepository gesendet werden kann.

```
1 git push origin 100-themen-titel
```

Listing 5.8: Mit git einen lokalen Branch an das Originalrepository senden

Die Entwickler des Originalrepositorys können den neuen Branch nun einsehen, überprüfen und in den Branch `master` einpflegen. Dadurch ist er offiziell Teil des Produkts geworden – diese Vorgehensweise ermöglicht sehr verteiltes Arbeiten bei gleichzeitig zentraler Verwaltung und Entscheidung, welche Änderungen tatsächlich dem Projekt zugeführt werden.

Bei dem Arbeitsfluss, den ich gerade beschrieben habe, ist das lokale Repository ein Klon des Originalrepositorys. In der Regel hat aber nur ein oder wenige Entwickler Schreibrechte auf dem Originalrepository, das alle lesen können. Daher ist in der Praxis noch ein zusätzlicher Schritt nötig, den ich im Folgenden vorstellen möchte.

5.1.3. Unterstützung durch GitHub

Die Idee ist, dass jeder Entwickler sein eigenen öffentlichen Klon des Originalrepositorys besitzt. Dieser öffentliche Klon wird von dem jeweiligen Entwickler lokal geklont, und daran gearbeitet, wie oben beschrieben. Zur Aktualisierung des lokalen Repositorys muss zusätzlich das öffentliche Repository des Entwicklers aktualisiert werden und nach Fertigstellung der Arbeit kann das Ergebnis per `push`-Befehl in dem Repository des Entwicklers veröffentlicht werden. Der oder die Entwickler des Originalrepository können die veröffentlichten Änderungen dann in das Originalprojekt einfließen lassen.

Für diesen Arbeitsablauf ist *GitHub* sehr gut geeignet – daher möchte ich diesen Arbeitsablauf am Beispiel von *GitHub* verdeutlichen.

1. Bei *GitHub* ist eine Registrierung notwendig, um Repositorys veröffentlichen zu können.
2. Ist man nun angemeldet, erscheint auf der Projektseite [Rö12a] der *fork*-Button. Dadurch klont *GitHub* das Projekt für den Benutzer, wo es nun unter `https://github.com/benutzername/micro-debug` öffentlich verfügbar ist.
3. Anschließend kann wie in Listing 5.9 das öffentliche Repository geklont werden. Der Unterschied zu Listing 5.1 ist die Adresse des geklonten Repositorys und das Protokoll – das git-Protokoll erlaubt Schreib- und Leseoperationen, bei entsprechenden Rechten auf dem Server.

```
1 git clone git://github.com/ihr-name/micro-debug.git
2 cd micro-debug
```

Listing 5.9: *micro-debug* mit git vom Benutzerrepository klonen

4. Um später Aktualisierungen des Originalrepository zu erhalten, sollte dieses noch im lokalen Repository als `remote` hinzugefügt werden, wie in Listing 5.10 zu sehen.

```
1 git remote add original-projekt git://github.com/croesch/micro-debug.git
```

Listing 5.10: Originalrepository des *micro-debug* dem lokalen Repository als `remote` hinzufügen

5. Im lokalen Repository wird nun ein Feature-Branch angelegt, wie in Listing 5.5 beschrieben.
6. Der entsprechende Code kann nun geändert und wie in Listing 5.4 committet werden – die Commits betreffen den neu erstellten Branch.

7. Da sich das öffentliche Repository des Benutzers nicht ändert, können Aktualisierungen nicht wie in Listing 5.6 beschrieben in das lokale Repository geholt werden. Daher benötigt das lokale Repository die Referenz auf das Originalrepository, wie in Listing 5.10 erstellt.

```
1 git checkout master
2 git pull original-projekt master
```

Listing 5.11: Mit git den Branch `master` auschecken und aktualisieren

   Listing 5.11 enthält die Befehle, um das lokale Repository mit den Informationen aus dem Originalrepository zu aktualisieren.

8. Abschließend sollte der neue Branch wie in Listing 5.7 beschrieben neu auf den `master`-Branch aufgesetzt werden und dann wie in Listing 5.8 gezeigt, in dem öffentlichen Repository des Benutzers veröffentlicht werden.

9. *GitHub* bietet nun die Möglichkeit die Entwickler des Originalprojekts über den neuen Branch im öffentlichen Repository des Entwicklers zu informieren. Dazu befindet sich auf der Projektseite des Benutzers ein Button *Pull-Request*. Wenn die Entwickler des Originalrepositorys entscheiden die Änderungen in das Projekt aufzunehmen, sind diese Informationen später über die in Listing 5.11 beschriebene Aktualisierung im Branch `master` zu sehen.

Meine Ausführungen über git sollten es ermöglichen, den Code des *micro-debug* auszuchecken, zu ändern und die Änderungen zu veröffentlichen. Im Folgenden möchte ich nun das Build-Management-Werkzeug erläutern, dass ich zum Entwickeln nutze: Maven.

5.2. Build-Management

Maven ist ein Werkzeug zum Bauen von Java-Anwendungen. Meines Erachtens gibt es keine signifikanten Gründe für oder gegen die Nutzung von Maven; stattdessen hätte ich auch ANT oder Make nutzen können.

In diesem Abschnitt möchte ich die grundlegenden Befehle für die Arbeit mit Maven erläutern.

Ich verwende Maven, da hier viel implizites Wissen eingesetzt wird; dadurch wird der Konfigurationsaufwand geringer – solange man den Konventionen der Maven-Entwickler folgt. Zu dem impliziten Wissen gehört unter anderem:

- Java-Klassen befinden sich in `src/main/java/`
- Java Test-Klassen befinden sich in `src/test/java/`
- Ressourcen befinden sich in `src/main/resources/`
- Ressourcen, die nur für die Tests benötigt werden, befinden sich in `src/test/resources/`
- Kompilate und von Maven generierte Klassen befinden sich in `target/`
- Die explizite Konfiguration steht in der Datei `pom.xml`

Die Datei `pom.xml` enthält aus Maven-Sicht alle nötigen Informationen über das Projekt: Die Version, der Name und benötigte Bibliotheken sind einige dieser Informationen.

Das wohl Wichtigste an einer Build-Management-Software ist, den Code kompilieren zu können. Listing 5.12 zeigt den Befehl, mit dem der Code eines Maven-Projekts wie dem *micro-debug* kompiliert werden kann. Testklassen werden dadurch nicht kompiliert und auch nicht ausgeführt.

```
mvn compile
```

Listing 5.12: Den Code eines Maven-Projekts kompilieren

Maven liest die Abhängigkeiten für ein Projekt aus der `pom.xml` Datei und lädt die benötigten Bibliotheken selbstständig herunter. Auch für die Ausführung von Maven werden gewisse Bibliotheken, dies führt bei der ersten Ausführung von Maven dazu, dass zunächst einige Bibliotheken heruntergeladen werden.

Bei der Ausführung von Maven gibt es verschiedene Ziele, die aufeinander aufbauen. So ist beispielsweise für das Packetieren eines Projekts notwendig zunächst alle Klassen zu kompilieren und die Tests auszuführen, bevor das Projekt packetiert wird. Das Projekt wird mit dem Befehl aus Listing 5.13 packetiert und im Verzeichnis `target/` abgelegt.

```
mvn package
```

Listing 5.13: Den Code eines Maven-Projekts packetieren

Wenn man Änderungen am Code vornimmt, ist es besonders hilfreich zu wissen, ob die automatisierten Tests fehlschlagen. Möchte man mit Maven die Tests ausführen, kann dazu der in Listing 5.14 gezeigte Befehl genutzt werden.

```
mvn test
```

Listing 5.14: Die automatisierten Tests eines Maven-Projekts ausführen

Die in der Datei `pom.xml` definierten abhängigen Projekte sind in XML (Extensible Markup Language) definiert, wie beispielsweise in Listing 5.15 am Beispiel von FEST (Fixtures for Easy Software Testing) zu sehen, das vom Debugger genutzte GUI-Test-Framework.

```
<dependency>
    <groupId>org.easytesting</groupId>
    <artifactId>fest-swing</artifactId>
    <version>1.2.1</version>
    <scope>test</scope>
</dependency>
```

Listing 5.15: Konfiguration eines abhängigen Projekts in Maven

Dort kann nun die Version von FEST geändert werden, die der Debugger nutzen soll.

Die hier gezeigten Befehle sind keine umfassende Beschreibung von Maven, genügen aber, um den Debugger zu bauen. Für das Kompilieren können auch andere Werkzeuge verwendet werden, allerdings muss der Entwickler sich dann um die Bibliotheken selbst kümmern. Aber prinzipiell ist Maven durch jedes beliebige andere Werkzeug ersetzbar.

6. Automatisierte Tests

Nachdem ich nun die Werkzeuge vorgestellt habe, die für das Entwickeln am Debugger benötigt werden, möchte ich noch einen wichtigen Aspekt ansprechen – automatisierte Tests in dieser Arbeit.

Bei der Implementierung dieser Arbeit sollte besonderen Wert auf die Erweiterbarkeit gelegt werden. Andere Studenten sollen sich in den Code einarbeiten, zusätzliche Funktionen implementieren und bestehende Funktionen abändern können.

Obwohl ich in meinem Studium gelernt habe, wie man ein Softwareprojekt plant und dadurch die optimale Struktur erhält, ist es mir bei der Entwicklung dieser Arbeit nicht gelungen vorher die optimale Struktur zu entwerfen. Immer wieder bin ich auf Probleme gestoßen, die es erforderten, den vorhandenen Code zu refaktorieren – die Struktur anzupassen:

> "**Refactoring** (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior." — [Fow99, S. 53]

Ich gehe davon aus, dass Refaktorierung am *micro-debug* und der *micro-debug-gui* auch in Zukunft eine wichtige Rolle spielen wird; besonders, wenn weitere Studenten mitentwickeln. *Fowler* schreibt an anderer Stelle nochmal deutlich, dass die Refaktorierung die Qualität des Codes nicht durch das Ändern des Verhaltens verbessert:

> "So, we want programs that are easy to read, that have all logic specified in one and only one place, that do not allow changes to endanger existing behavior, and that allow conditional logic to be expressed as simply as possible.
>
> Refactoring is the process of taking a running program and adding to its value, not by changing its behavior but by giving it more of these qualities that enable us to continue developing at speed." — [Fow99, S. 60]

Wie stelle ich sicher, dass ich durch Refaktorierung das Verhalten des Codes nicht manipuliere? Ich schreibe automatisierte Tests: Ausführliche automatisierte Tests geben mir die nötige Sicherheit für die Refaktorierung, die wiederum die Lesbarkeit und Erweiterbarkeit des Codes steigert – die zwei wichtigsten Ziele in dieser Arbeit.

Für den Debugger habe ich für nahezu jede Klasse automatisierte Tests geschrieben, die im Verzeichnis `src/test/java/` zu finden sind. Jede Testklasse liegt in diesem Verzeichnis im selben Package wie die zu testende Klasse und trägt den selben Namen, allerdings um das Suffix *Test* erweitert.

Die Tests sind nicht nur indirekt über die Refaktorierung eine gute Möglichkeit, den Code besser zu verstehen: Möchte man das Verhalten oder die Funktion einer Klasse verstehen, sind die automatisierten Tests der entsprechenden Testklasse ein guter Anlaufpunkt. Denn in jedem Test werden für verschiedene Eingaben die erwarteten Ausgaben beschrieben und dadurch gezeigt, wie sich die Klasse verhalten soll.

Dazu möchte ich ein kleines Beispiel geben: Listing 6.1 zeigt den Code, der analysiert werden soll. Die beiden entsprechenden Testmethoden sind in Listing 6.2 und Listing 6.3 zu sehen. Ohne die JavaDoc-Kommentare, die hier nicht aufgeführt sind, ist es nicht ganz einfach den Code aus Listing 6.1 zu verstehen.

```java
@Nullable
public Integer parse(final String toParse) {
  if (toParse == null) {
    return null;
  }

  try {
    // try parsing and if this works, it was a valid number
    return Integer.valueOf(toParse);
  } catch (final NumberFormatException nfe) {
    // number might be 0x.. or ..._.., so split the number in radix and the number
    // convert aliases (0x,0b,..) to the correct notation
    final String[] num = convertAliases(toParse).split("_");
    if (num.length != 2) {
      // not a valid number - no idea what to do
      return null;
    }

    // number is a special number
    try {
      // try to parse the number with the specified radix
      return Integer.valueOf(num[0], Integer.parseInt(num[1]));
    } catch (final NumberFormatException nfe2) {
      // didn't work -> invalid number
      return null;
    }
  }
}

@NotNull
private String convertAliases(final String num) {
  if (num.length() > 1 && num.charAt(0) == '0') {
    switch (Character.toLowerCase(num.charAt(1))) {
      case 'b': return num.substring(2) + "_2";
      case 'o': return num.substring(2) + "_8";
      case 'x': return num.substring(2) + "_16";
      default:  return num;
    }
  }
  return num;
}
```

Listing 6.1: `IntegerParser.java` – relevante Methoden

Wenn man aber den Testcode aus Listing 6.2 anschaut, dann erhält man schnell eine Ahnung, was die Klasse macht. Eine mögliche Vermutung kann nun an Listing 6.3 noch überprüft werden. Handelt es sich hier um die Funktionalität, die in Abschnitt 2.2.1 vorgestellt wurde?

```java
@Test
public void testNumber_Valid() {
  IntegerParser parser = new IntegerParser();
  assertThat(parser.parse("12")).isEqualTo(12);
  assertThat(parser.parse("0x12")).isEqualTo(0x12);
  assertThat(parser.parse("12_16")).isEqualTo(0x12);
  assertThat(parser.parse("1234_10")).isEqualTo(1234);
  assertThat(parser.parse("1010_2")).isEqualTo(10);
  assertThat(parser.parse("0b1010")).isEqualTo(10);
  assertThat(parser.parse("11010_002")).isEqualTo(26);
  assertThat(parser.parse("1g_17")).isEqualTo(33);
  assertThat(parser.parse("-1g_17")).isEqualTo(-33);
  assertThat(parser.parse("-1G_17")).isEqualTo(-33);
  assertThat(parser.parse("-011")).isEqualTo(-11);
  assertThat(parser.parse("-0000")).isEqualTo(0);
  assertThat(parser.parse("0o0")).isEqualTo(0);
  assertThat(parser.parse("0o27")).isEqualTo(23);
  assertThat(parser.parse("27_8")).isEqualTo(23);
  assertThat(parser.parse("0B0")).isEqualTo(0);
  assertThat(parser.parse("0O0")).isEqualTo(0);
  assertThat(parser.parse("0X0")).isEqualTo(0);
}
```

Listing 6.2: `IntegerParser.java` – Test auf Validität

Es ist zu sehen, dass der zu testenden Klasse Zeichenketten übergeben werden und geprüft wird, ob das Ergebnis die erwartete Zahl ist. Die Vermutung war also korrekt! Außerdem ist zu sehen, welche Werte die Komponente zurück gibt, wenn ungültige Zeichenketten übergeben werden: `null`. Durch Analyse des Tests hat man also erstens eine gute Vorstellung davon, was die Klasse macht und zweitens durch den automatisierten Test gesichert, dass bei künftigen Refaktorierungen oder sonstigen Änderungen das Verhalten nicht manipuliert wird.

```
1  @Test
2  public void testNumber_Invalid() {
3    IntegerParser parser = new IntegerParser();
4    assertThat(parser.parse("")).isNull();
5    assertThat(parser.parse(" ")).isNull();
6    assertThat(parser.parse("\t")).isNull();
7    assertThat(parser.parse("A")).isNull();
8    assertThat(parser.parse("1010_a")).isNull();
9    assertThat(parser.parse("2010_2")).isNull();
10   assertThat(parser.parse("a010_2")).isNull();
11 }
```

Listing 6.3: `IntegerParser.java` – Test auf Invalidität

Zusammenfassend lässt sich sagen, dass für mich die ausführlichen automatisierten Tests das Erreichen der beiden wichtigsten Ziele dieser Arbeit sichern: leichte Erweiterbarkeit und gute Lesbarkeit des Codes. Sicherlich habe ich in das Schreiben ausführlicher Tests viel Zeit investiert, wodurch ich weniger Zeit für die Entwicklung von Programmcode hatte. Ich denke aber, dass durch diese anfängliche Zeiteinbuße und bei kontinuierlicher Pflege der Tests dauerhaft viel Zeit gespart werden kann: in der Praxis habe ich schon oft erfahren, wie viel Zeit verloren gehen kann, wenn ungenügend getestet wird.

7. Implementierung der Mic-1

Nachdem ich nun die Vorteile der automatisierten Tests genannt habe, möchte ich den Code fokussieren. Ich werde kann hier aus Zeit- und Platzgründen nur gewisse Eckpunkte der Implementierung beleuchten, die verschiedenen Klassen und Methoden sind jeweils gut dokumentiert und sollten nach dem Lesen der folgenden Kapitel schnell verstanden werden können.

Der erste Schritt bei der Implementierung des Debuggers ist die Simulation der *Mic-1*, erst wenn dies zufriedenstellend funktioniert, können die Debug-Funktionen entwickelt werden. Ich möchte daher in diesem Kapitel die Implementierung des Prozessors anschauen und zeigen, welche Besonderheiten es gibt.

Für die Implementierung der *Mic-1* ist es nötig, sich ein Abstraktionsniveau auszusuchen, auf dem implementiert wird. In den folgenden Abschnitten ist zu sehen, dass ich mich für eine relativ hardwarenahe Implementierung entschieden habe. Daher habe ich zunächst folgende zu implementierende Komponenten identifiziert: die ALU (Arithmetisch-logische Einheit) inklusive Shifter, die Register, den Mikro-Code-Speicher inklusive der darin abgelegten Instruktionen und der Hauptspeicher.

Aufgrund der Maven-Architektur liegt der Java-Code in dem Verzeichnis `src/main/java/` und die entsprechenden Tests dazu in `src/test/java/`. In diesen Verzeichnissen ist eine Package-Struktur erkennbar, die unter dem Package `com.github.croesch.micro_debug` beginnt, alle Klassen des Debuggers sind in diesem Package oder in Subpackages zu finden.

Auch der Code für die *Mic-1* befindet sich in einem solchen Subpackage: `com.github.croesch.micro_debug.mic1`. In diesem Verzeichnis gibt es die Klasse `Mic1.java`, die ich in Abschnitt 7.6 genauer beschreiben werde. Alle weiteren Klassen, die den *Mic-1* bilden sind in folgenden Subpackages zu finden:

alu bildet die ALU ab und ist in Abschnitt 7.1 beschrieben.

api enthält einige Interfaces, um die Implementierung der *Mic-1* unabhängig gestalten zu können.

controlstore bildet den Mikro-Assembler-Code-Speicher und die Mikro-Assembler-Instruktionen ab und ist in Abschnitt 7.3 beschrieben.

io bildet die Ein- und Ausgabe der *Mic-1* ab. Die Klasse `Input.java` dient zum Einlesen einzelner Zeichen und enthält einen Puffer, der die noch nicht von der *Mic-1* gelesenen Zeichen enthält, da vom Benutzer nur zeilenweise Eingaben gemacht werden können.

Die Klasse `Output.java` dient zur Ausgabe einzelner Zeichen und puffert, falls die Ausgabe gepuffert erfolgen soll, die ausgegebenen Zeichen bis ein Zeilenumbruch ausgegeben wird.

mem bildet den Hauptspeicher ab und ist in Abschnitt 7.5 beschrieben.

mpc bildet die Komponenten ab, die zusammen für die Berechnung des nächsten MPC (*micro program counter*) verantwortlich sind und ist in Abschnitt 7.4 beschrieben.

register bildet die Register der *Mic-1* ab und ist in Abschnitt 7.2 beschrieben.

shifter bildet den Shifter ab und ist in Abschnitt 7.1 beschrieben.

Die Package-Struktur sollte bereits einen groben Überblick geben, wo welche Komponenten implementiert sind. Zum besseren Verständnis möchte ich nun nochmal die Hardware-Komponenten der *Mic-1* betrachten und ihre Implementierungen beschreiben.

7.1. ALU

Die ALU der *Mic-1* setzt sich aus 32 Ein-Bit-ALUs zusammen – so ist die ALU auch implementiert. Die Klasse `Alu.java` benutzt 32 Instanzen der Klasse `OneBitAlu.java`, zur Berechnung der Ausgabewerte – damit ist die ALU die Komponente, die am hardwarenahesten implementiert ist.

Die Klasse besitzt eine Methode `calculate()`, in der aus den Eingangssignalen das Ausgangssignal der ALU berechnet wird; dieses Signal wird an den Shifter weitergeleitet. Die Klasse `Shifter.java` im gleichnamigen Package bildet den Shifter ab und besitzt ebenso eine Methode `calculate()`, um die Verarbeitung der Signale anzustoßen.

Die Berechnung der ALU und des Shifters werden bei jedem Zyklus der *Mic-1* angestoßen. Durch die sehr hardwarenahe Implementierung bilden sie daher vermutlich ein Performance-Engpass zur Laufzeit des *micro-debug*.

7.2. Register

Die Dateneingänge der ALU werden mit den Inhalten zweier Register gefüllt und das Ergebnis des Shifters wird wiederum in verschiedene Register geschrieben. Die Register sind im *micro-debug* als Enumeration `Register.java` im gleichnamigen Package implementiert. Sie erfüllen lediglich die Aufgabe einer 32 Bit Variable.

Das Register `MBR` kann im *Mic-1* mit oder ohne Vorzeichenerweiterung gelesen werden. Dieses Verhalten ist im *micro-debug* dadurch realisiert, dass es ein Register `MBR` und ein Register `MBRU` gibt. `MBRU` enthält den Wert ohne Vorzeichenerweiterung und das Register `MBR` enthält den Wert mit Vorzeichenerweiterung. Bei der Vorzeichenerweiterung wird wie bei der *Mic-1* davon ausgegangen, dass der ursprüngliche Wert in 8 Bit vorlag.

Dass die Register als Enumeration implementiert sind hat den Vorteil, dass man von allen Klassen leicht auf die Register zugreifen kann, aber den Nachteil, dass allein durch die Code-Struktur nicht deutlich wird, zu welcher logischen Einheit die Register gehören.

7.3. Mic1-Instruktionen

Welche Berechnung mit welchen Registerwerten ausgeführt wird und in welches Register das Ergebnis geschrieben wird, regeln die Signale der Mikro-Assembler-Instruktionen. Die Mikro-Assembler-Instruktionen (`MicroInstruction.java`) sind im Mikro-Assembler-Code-Speicher (`MicroControlStore.java`) abgelegt.

Zur besseren Lesbarkeit des Codes nutzt die Implementierung der Mikro-Assembler-Instruktion Unterklassen von `SignalSet.java`, um die verschiedenen Signale zu gruppieren. Die Darstellung der Mikro-Assembler-Instruktion für den Benutzer ist in der Klasse `MicroInstructionDecoder.java` implementiert. Wie wird eine Mikro-Assembler-Instruktion erzeugt? Der Mikro-Assembler-Code-Speicher ist in der `.mic1`-Datei definiert; aus dieser Datei kann die Klasse `MicroInstructionReader.java` einzelne Mikro-Assembler-Instruktionen erzeugen.

Meine Implementierung der Mikro-Assembler-Instruktion, der Darstellung einer Mikro-Assembler-Instruktion und dem Einlesen einer Mikro-Assembler-Instruktion basieren auf der Implementierung von *Ontko*. *Ontko* hat in [Ont99] einen Simulator für die *Mic-1* in Java entwickelt, der bereits Komponenten zur Darstellung und zum Einlesen von Mikro-Assembler-Instruktionen implementiert hat.

7.4. MPC-Berechnung

Der MPC bestimmt, welche Mikro-Assembler-Instruktion im nächsten Zyklus ausgeführt werden soll. In der *Mic-1* sind mehrere Komponenten gemeinsam für die Berechnung des MPC verantwortlich. Diese verschiedenen Komponenten habe ich in der Klasse `NextMPCCalculator.java` zusammengefügt; auch hier gibt es eine Methode `calculate()` zum Verarbeiten der Eingangssignale.

7.5. Hauptspeicher

Der Stack, der Assembler-Code und die Konstanten liegen im Hauptspeicher; mit diesem kommuniziert die *Mic-1* über die Register `MAR`, `MDR`, `MBR` und `PC`. Der Hauptspeicher ist in der Klasse `Memory.java` implementiert; der Speicher selbst ist in einem *int*-Array abgebildet, weswegen der Hauptspeicher im *micro-debug* wortweise adressiert wird.

Die Assembler-Instruktionen im Hauptspeicher sind nicht so sauber implementiert wie die Mikro-Assembler-Instruktionen: sie sind im *int*-Array abgelegt. Lediglich für das disassemblieren werden die Zahlenwerte anhand der Datei `ijvm.conf` durch den `IJVMConfigReader.java` in Instruktionsobjekte (`IJVMCommand.java` inklusive `IJVMCommandArgument.java`) gepackt.

7.6. Zusammensetzung der Komponenten

Die einzelnen Komponenten werden in der Klasse `Mic1.java` zusammengeführt und nach außen als ein Prozessor dargestellt. Lediglich für gewisse Debug-Funktionen ist der Zugriff auf einzelne Komponenten, wie beispielsweise die Register, zugelassen.

Die Klasse `Mic1.java` bekommt im Konstruktor zwei Objekte der Klasse `InputStream.java` – eines enthält den Mikro-Assembler- und das andere den Assembler-Code. Mit diesen beiden Objekten wird dann der Mikro-Assembler-Code-Speicher und der Hauptspeicher erzeugt, die jeweils selbst für das Lesen des Bytecodes verantwortlich sind. Stimmt eine *magic number* nicht, oder sind die Eingabedateien aus anderen Gründen ungültig, wird eine *Exception* geworfen.

Auf die einzelnen Methoden möchte ich hier nicht eingehen, bis auf eine: In der Methode `doTick()` wird ein einzelner Zyklus der *Mic-1* ausgeführt. Diese Methode ist nur für Testzwecke sichtbar und wird von außen über die Methoden `run()`, `step()` und `microStep()` aufgerufen, die unterschiedlich viele Zyklen auf einmal abarbeiten lassen.

Aufgrund der gesammelten Funktionalität ist die Klasse `Mic1.java` eine der größten und damit komplexesten im *micro-debug*.

8. Implementierung der Konsolenvariante

Nachdem ich nun kurz die Simulation der *Mic-1* beschrieben habe, möchte ich jetzt die Implementierung des *micro-debug* vorstellen. In den folgenden Abschnitten möchte ich in Abschnitt 8.1 zunächst skizzieren, welche Klassen im Programmablauf des *micro-debug* welche Rolle spielen. Danach möchte ich in Abschnitt 8.1.4 erläutern, wie die Konfigurationsdateien gelesen werden und in Abschnitt 8.1.5 wie die Lokalisierungsdateien interpretiert werden. Am Ende dieses Kapitels möchte ich in Abschnitt 8.2 die einzelnen Packages beschreiben und damit einen vollständigen Überblick über den Code des *micro-debug* geben.

8.1. Programmablauf

Der Programmablauf des *micro-debug* wurden nun insbesondere in Abschnitt 3.3 einige Male beschrieben, daher wahrscheinlich der einfachste Einstieg in den Code des *micro-debug* am Programmablauf zu erklären. Ich möchte deshalb in diesem Abschnitt einige Schritte im Programmablauf aufgreifen und die beteiligten Klassen erwähnen.

8.1.1. Verarbeitung der Argumente

Wie an den Startskripten erkennbar ist, ist die `main()`-Methode in der Klasse `MicroDebug.java` enthalten. In dieser Klasse wird zunächst geprüft, wie viele Argumente der Benutzer angegeben hat und dann entsprechende Aktionen eingeleitet.

Angenommen der Benutzer möchte den *micro-debug* starten und hat daher mindestens zwei Argumente angegeben; in diesem Fall verarbeitet die Methode `createArgumentList(String[])` in der Klasse `AArgument.java` die Argumente. Diese Klasse ist auch gleichzeitig die Basisklasse für alle verfügbaren Argumente. Die Argumente, die jeweils noch Parameter besitzen können, werden dann in der Methode `executeTheArguments(Map<AArgument, String[]>)` nacheinander ausgeführt.

Hier wird die Methode `execute(String ...)` an jedem Argument mit den dazugehörigen Parametern ausgeführt. Diese Methode gibt einen Wahrheitswert zurück, der bestimmt, ob der *micro-debug* nach Ausführung des Arguments starten darf oder nicht.

8.1.2. Aufbau der Debugumgebung

Nachdem die Argumente abgearbeitet wurden und der *micro-debug* starten darf, werden die beiden Bytecode-Dateipfade eingelesen und in Objekte der Klasse `InputStream.java` verwandelt. Dann wird, wie im vorherigen Kapitel beschrieben, mit diesen beiden Objekten die *Mic-1* erzeugt. Hierbei kann es sein, dass bereits im Konstruktor eine *Exception* auftritt, wenn eine der Dateien nicht das erwartete Format hat. Daher ist in der Klasse `MicroDebug.java` auch die Fehlerbehandlung für diesen Fall definiert.

Ist die *Mic-1* erzeugt, kann die Debug-Umgebung gestartet werden. Hier ist die zentrale Klasse die `Debugger.java`: in dieser Klasse befindet sich die Schleife zum Einlesen der Benutzerinstruktionen, aber dazu im Abschnitt 8.1.3 mehr. In der Klasse `Debugger.java` wird auch eine Instanz der Klasse `Mic1Interpreter.java` erzeugt, die die eigentlichen Debug-Funktionen sammelt und delegiert. Sie dient als Abstraktionsebene zwischen Benutzerinstruktionen und dem Prozessor *Mic-1*, so dass die *Mic-1* nicht mit Debug-Funktionalität gefüllt wird.

Die Klasse `Mic1Interpreter.java` delegiert einige Funktionalität an zwei wichtige Klassen, die hier erzeugt werden: `TraceManager.java`, zum Beobachten von Prozessorvariablen und `MemoryInterpreter.java`, der eine Abstraktionsebene über der Klasse `Memory.java` darstellt und den Assembler-Code disassemblieren kann.

8.1.3. Verarbeitung der Benutzerinstruktionen

Wenn die *Mic-1* erzeugt wurde und die Debug-Umgebung initialisiert ist, wird in der Klasse `MicroDebug.java` die Methode `run()` an der Klasse `Debugger` aufgerufen und damit die Schleife für die Bearbeitung der Benutzerinstruktionen gestartet. Das Bearbeiten der Benutzerinstruktionen funktioniert ähnlich, wie das Bearbeiten der Argumente: die Methode `of(String)` der Klasse `UserInstruction.java` wandelt einen *String* in eine Benutzerinstruktion um.

Anschließend wird diese in der Methode `execute(Mic1Interpreter,String ...)` mit den eventuellen Parametern ausgeführt. Der Unterschied zu der Verarbeitung der Argumente ist, dass die Benutzerinstruktionen keine Klassenhierarchie bilden, sondern in einer Enumeration implementiert sind. Demzufolge enthält die Klasse `UserInstruction.java` die Implementierung aller Benutzerinstruktionen – was sie zu einer der größten und komplexesten Klassen im *micro-debug* macht.

Auch die Methode `execute(Mic1Interpreter,String ...)` gibt einen Wahrheitswert zurück – dieser gibt an, ob der *micro-debug* nach Ausführung der Benutzerinstruktion beendet werden soll oder nicht. Beendet der Benutzer den Debugger mit dem Befehl `EXIT`, dann terminiert die Methode `run()` in `Debugger.java` und damit auch am Ende die `main()`-Methode in `MicroDebug.java`.

8.1.4. Verarbeitung der Konfiguration

Der Benutzer hat (wie in Abschnitt 2.2 beschrieben) die Möglichkeit, den *micro-debug* zu konfigurieren. Das Einlesen der Konfiguration geschieht implizit, das heißt zu keinem definierten Zeitpunkt, sondern sobald die erste Konfigurationsoption benötigt wird.

Das Package `settings` enthält verschiedene Klassen, die alle einen anderen Typ von Konfigurationsoptionen enthalten. Diese Klassen sind als Enumerationen implementiert und nutzen meist die Klasse `PropertiesProvider.java` – die Klasse, die Zugriff auf die tatsächlichen Dateien hat. Dieser Mechanismus ist etwas komplex, daher möchte ich hier die einzelnen Schritte erklären, die beim Abfragen der ersten Konfigurationsoption ausgeführt werden:

1. An der Klasse `PropertiesProvider.java` wird die Methode `getInstance()` aufgerufen und damit die einzige Instanz der Klasse erzeugt.
2. An diesem Objekt wird die Methode `get(String,String)` aufgerufen, die einen Dateinamen und den Schlüssel der Konfigurationsoption erhält.
3. Das Objekt ruft an sich selbst `getProperties(String)` mit dem Dateinamen auf.
4. Das Objekt hält eine *Map* mit einem *Properties*-Objekt für jeden Dateinamen. Da diese *Map* noch leer ist, wird die Methode `createNewProperties(String)` mit dem Namen der Datei aufgerufen. Diese Methode wird von jeder Unterklasse überschrieben und liest die Datei ein und erzeugt ein *Properties*-Objekt, welches in die *Map* gelegt wird, um bei späteren Aufrufen darauf zugreifen zu können.
5. Dieses *Properties*-Objekt wird nun einige Methodenaufrufe zurück gereicht und daran wird dann die Methode *getProperty(String)* aufgerufen, um den Wert der Konfigurationsoption zu erhalten.

6. Der erhaltene Wert wird nun bis in den Konstruktor der Enumeration zurückgegeben und dort auf Validität überprüft. Stellt sich heraus, dass der Wert ungültig ist, wird die Konfigurationsoption mit einem Standardwert belegt. Das Ergebnis steht dem Benutzer dann zur Verfügung.

Beim der zweiten Konfigurationsoption aus einer Datei, wird der gerade beschriebene Mechanismus in Punkt 4 verkürzt und besteht nur aus einer Wertabfrage an eine *Map*.

Die Klasse `PropertiesProvider.java` erbt von `APropertiesProvider.java`, die einige der gerade beschriebenen Funktionalität enthält. Es gibt nämlich weitere Unterklassen, die beispielsweise Konfigurationen aus `.xml`-Dateien lesen.

8.1.5. Verarbeitung der Lokalisierung

Die Verarbeitung der Lokalisierung geschieht analog zur im Abschnitt 8.1.4 beschriebenen Verarbeitung der Konfiguration.

Allerdings werden die Lokalisierungsdateien aus `.xml`-Dateien gelesen und deshalb die Klasse `XMLPropertiesProvider.java` genutzt. Der einzige Unterschied besteht in Punkt 4 des Ablaufs in Abschnitt 8.1.4: Statt einem gewöhnlichen *Properties*-Objekt wird hier ein Objekt der Klasse `XMLI18nProperties.java` erzeugt. Diese Klasse ist für das in Abschnitt 2.3 beschriebene Verhalten verantwortlich: Im Konstruktor liest sie die Dateien von allgemein nach spezifisch und ergänzt so die erzeugten Schlüsselwert-Paare der allgemeinen Dateien mit den Informationen aus den spezifischen Dateien..

8.2. Packages

Ich habe nun die Hauptklassen des *micro-debug* vorgestellt und einen groben Überblick über die verschiedenen Klassen gegeben. Ich möchte in diesem Abschnitt nochmal alle Packages des *micro-debug* erwähnen und kurz ihren Inhalt beschreiben. Danach sollten Leser für die meisten Eigenschaften des *micro-debug* ein Gefühl haben, wo die Implementierung zu finden ist. Der *micro-debug* besteht aus folgenden Packages:

annotation dieses Package enthält nur Annotationen. Zur Zeit `@Nullable` und `@NotNull`, die dazu genutzt werden, um Methoden zu markieren, ob sie `null` zurückgeben oder nicht. Auch Variablen können damit markiert werden, um zu definieren, ob sie den Wert `null` annehmen können oder nicht.

Diese Annotationen helfen bei der Navigation durch den Code, denn einmal analysiert und markiert erspart man sich beim nächsten Betrachten der Methode die Analyse.

argument enthält die Klasse `AArgument.java` und ihre Unterklassen – somit alle möglichen Argumente des *micro-debug*.

commons enthält Klassen, die nicht zugeordnet werden konnten. Beispielsweise `Reader.java` und `Printer.java`, die für die Ein- und Ausgabe des *micro-debug* verantwortlich sind. Auch die Klasse `Utils.java` befindet sich hier – sie enthält einige Methoden, die keinem genauen Objekt und keiner Klasse zugeordnet werden konnten, wie beispielsweise die Methode `toBinaryString(int)`, die eine Zahl in eine Zeichenkette wandelt, die die Binärrepräsentation der Zahl darstellt.

console enthält die Debug-Klassen, die speziell für die Benutzung per Konsole konzipiert sind. Hier sind die Klassen `Debugger.java`, `UserInstruction.java` und `Mic1Interpreter.java` zu finden.

debug enthält Klassen, die mit Breakpoints zu tun haben. Hier sind sowohl der `BreakpointManager.java` zu finden, als auch die verschiedenen Unterklassen von `Breakpoint.java`, die die verschiedenen Breakpoints darstellen. Diese Unterklassen von `Breakpoint.java` müssen nur in dem Package sichtbar sein – für den `BreakpointManager.java`.

error enthält eigene *Exceptions*.

i18n enthält die Enumeration `Text.java`, die die Klasse `XMLPropertiesProvider.java` nutzt, um Textkonstanten aus den Lokalisierungsdateien zu lesen und im ganzen Programm verfügbar zu machen.

mic1 wie in Kapitel 7 beschrieben.

parser enthält das Interface `IParser.java` und dessen Unterklassen: `IntegerParser.java` und `RegisterParser.java`, die aus Zeichenketten das entsprechende Objekt parsen. `IntegerParser.java` liest beispielsweise Zahlen nach dem in Abschnitt 2.2.1 beschriebenen Zahlenformat ein und gibt ein *Integer*-Objekt zurück.

properties enthält die Logik, wie Konfigurationsdateien einzulesen und zu speichern sind. Hier sind beispielsweise die Klassen `APropertiesProvider.java`, `PropertiesProvider.java`, `XMLPropertiesProvider.java` und `XMLI18nProperties.java` zu finden.

settings enthält verschiedene Einstellungs-Enumerationen, zur Zeit eine für intern und eine für vom Benutzer zu ändernde Konfigurationsoptionen. Angedacht ist, dass künftig in der Datei `micro-debug.properties` nicht nur ganzzahlige Einstellungen benutzt werden, sondern auch Wahrheitswerte oder Farbeinstellungen. Jeder Datentyp hätte dann in diesem Package eine Klasse, die alle auf die selbe Datei zugreifen. So wäre intern die Typsicherheit gewährleistet und der Benutzer hätte in einer einzigen Datei alle Einstellungen, die er je nach Konfigurationsoption mit Zahlenwerten, Wahrheitswerten oder Sonstigem angibt.

Für jeden Datentyp kann hier eine eigene Enumeration genutzt werden, die eine Unterklasse von `PropertiesProvider.java` nutzt, da diese Klasse wie in Abschnitt 8.1.4 beschrieben die Dateien in einem Zwischenspeicher hält. Dadurch wird jede Konfigurationsdatei nur einmal gelesen und kann dann von den verschiedenen Einstellungs-Enumerationen ausgewertet werden – jede Enumeration nutzt nur die Werte aus der Datei, die sie selbst benötigt.

Ich habe nun eine Übersicht über den *micro-debug* und seine Implementierung gegeben. Nochmal wiederholt, die wichtigsten Klassen des *micro-debug* sind: die Startklasse `MicroDebug.java`, die Enumeration der Benutzerinstruktionen `UserInstruction.java`, die *Mic-1* `Mic1.java` und die Schleife zum Einlesen der Benutzerinstruktionen in `Debugger.java`.

9. Implementierung der GUI

In diesem Kapitel möchte ich vorstellen, wie die *micro-debug-gui* den *micro-debug* nutzt und welche Klassen hier welche Rolle einnehmen. Ähnlich wie in Kapitel 8 möchte ich im Abschnitt 9.1 die Komponenten anhand des Programmablaufs der *micro-debug-gui* vorstellen. In Abschnitt 9.2 möchte ich die verwendeten Bibliotheken vorstellen, die der Benutzer am Ende erhält und deren Bedeutung für die *micro-debug-gui* erläutern. Am Ende dieses Kapitels möchte ich auch hier nochmal alle Packages aufführen und beschreiben, welche Funktionalität dahinter steckt und eventuell einzelne Klassen erwähnen.

Die *micro-debug-gui* ist ein eigenständiges Projekt und vom Code her ein Benutzer des *micro-debug*. Alle Implementierungen der *micro-debug-gui* haben also keinerlei Auswirkung auf den *micro-debug*, umgekehrt allerdings schon. Wie in Abschnitt 4.1 beschrieben ist es daher möglich, dass bei vorliegender *micro-debug-gui* auch der *micro-debug* eigenständig genutzt werden kann.

Im Unterschied zur Implementierung des *micro-debug* befinden sich in der *micro-debug-gui* alle Klassen im Package `com.github.croesch.micro_debug.gui` oder in Subpackages davon. Das verhindert, dass gleichnamige Packages oder Klassen zwischen *micro-debug* und *micro-debug-gui* zu Problemen führen. Außerdem ist je nach Packetierung die Struktur der packetieren *micro-debug-gui* übersichtlicher.

9.1. Programmablauf

Wie in Abschnitt 8.1 möchte ich zunächst den Start der *micro-debug-gui* betrachten und zeigen, welche Klassen an welchen Aktionen beteiligt sind. Gelegentlich werde ich erwähnen, wo Zugriffe auf den *micro-debug* stattfinden oder wann der Benutzer gefragt ist.

9.1.1. Verarbeitung der Argumente

Analog zum *micro-debug* gibt es auch in der *micro-debug-gui* eine Klasse `MicroDebug.java`, die die `main()`-Methode enthält. Allerdings ist es hier irrelevant, wie viele Argumente der Benutzer eingegeben hat, oder ob er überhaupt welche eingegeben hat.

Die *micro-debug-gui* nutzt auch den Mechanismus der Klasse `AArgument.java`. Da die *micro-debug-gui* allerdings andere Argumente als der *micro-debug* besitzt, müssen die neuen Argumente zunächst an der Klasse `AArgument.java` registriert werden. Die Argumente der *micro-debug-gui* sind im Package `argument` zu finden; registriert werden diese in der Methode `createListOfPossibleArguments()` der Klasse `MicroDebug.java`.

Nachdem die textuellen Argumente in Objekte vom Typ `AArgument.java` konvertiert wurden, werden diese ausgeführt und geben auch jeweils einen Wahrheitswert zurück, ob die *micro-debug-gui* starten darf oder nicht.

9.1.2. Aufbau der grafischen Oberfläche

Darf die *micro-debug-gui* starten, so wird eine Instanz der Klasse `Mic1Starter.java` erzeugt, die die *micro-debug-gui* starten kann. Zunächst erzeugt die Klasse jedoch das Startfenster aus Abbildung 4.1, welches in der Klasse `StartFrame.java` definiert ist, und zeigt dieses an. Das Startfenster enthält einige Aktionsmöglichkeiten – hat der Benutzer nun die richtigen Dateien

ausgewählt und den Button zum Start gedrückt, wird an der Klasse `Mic1Starter.java` die Methode `create(String,String)` mit den zwei angegebenen Dateipfaden ausgeführt.

Wie in Abschnitt 8.1.2 wird nun anhand dieser Dateipfade versucht die *Mic-1* zu erzeugen. Gelingt dies nicht, wird das Startfenster erneut gezeigt. Hat der Benutzer zwei korrekte Dateien angegeben, wird das Hauptfenster aus Abbildung 4.4 aufgebaut und angezeigt, was in der Klasse `MainFrame.java` definiert ist.

In der Klasse `MainFrame.java` werden drei erwähnenswerte Objekte erzeugt: die *View*, der *Controller* und die *Actions*. Die *View* bildet die Oberfläche und ist zunächst in der Klasse `MainView.java` definiert, die wiederum weitere Klassen zur Darstellung der einzelnen Bereiche nutzt. Ein ähnliches Konzept nutzt der *Controller*, der in der Klasse `MainController.java` definiert ist: Für die einzelnen kleinen *Views* werden entsprechende *Controller* erzeugt, die dadurch jeweils nur einen kleinen Aufgabenbereich erhalten. Das dritte Objekt ist von der Klasse `ActionProvider.java` und erzeugt die *Actions*, die der Benutzer über die Oberfläche ausführen kann. Dieser *ActionProvider* hält die Referenzen auf die *Actions*, so dass nur dieses eine Objekt weitergereicht werden muss anstatt dutzender *Actions*.

Diese *Actions* werden dann in der `MainMenuBar.java` in Menüeinträgen visualisiert und mit Tastenkombinationen versehen, die der Benutzer konfigurieren kann.

9.1.3. Verarbeitung von Benutzeraktionen

Nachdem die Oberfläche aufgebaut ist, sind die *Actions* der Ausgangspunkt von weiteren Code-Ausführungen. Bis der Benutzer die `EXIT`-Aktion ausführt oder das Fenster schließt, läuft die *micro-debug-gui*.

In der Klasse `ActionProvider.java` werden die Referenzen auf die *Actions* gehalten und unter den Schlüsseln abgelegt, die die Enumeration `Actions.java` bietet.

Bis auf einige Ausnahmen werden die verschiedenen *Actions* auf dem EDT ausgeführt. Bei einigen *Actions* wäre dies von Nachteil: spätestens wenn die *Mic-1* eine Eingabe erwarten würde, käme es zu einem Deadlock. Denn die Eingabe für die *Mic-1* wird in der *micro-debug-gui* über ein Textfeld ausgeführt. Liest die *Mic-1* nun auf dem EDT ein Zeichen aus diesem Textfeld, obwohl dort noch keines eingegeben wurde, dann blockiert der aktuelle Thread (in diesem Fall der EDT) bis der Benutzer eine Eingabe gemacht hat. Eine Eingabe kann aber nur über den EDT ausgeführt werden, der noch blockiert ist – ein Deadlock.

Aus diesem Grund gibt es die Klasse `AbstractExecuteOnWorkerThreadAction.java`. Jede *Action*, die davon ableitet, wird nicht auf dem EDT ausgeführt sondern auf einer Instanz der Klasse `WorkerThread.java`. Dieser Thread läuft und arbeitet kontinuierlich Objekte der Klasse `Runnable.java` ab und wird hier genutzt, um solche Deadlocks zu umgehen.

9.1.4. Konfiguration und Lokalisierung

Das Lesen der Konfiguration und der Textkonstanten funktioniert analog zu den Ausführungen in Abschnitt 8.1.4 und Abschnitt 8.1.5.

Allerdings nutzt die *micro-debug-gui* eigene Enumerations-Klassen: die Klasse `GuiText.java` für die Textkonstanten und die Klassen im Package `settings` zum Verarbeiten der Konfiguration. Somit können Komponenten der *micro-debug-gui* sowohl auf die Konfiguration des *micro-debug* als auch der *micro-debug-gui* zugreifen und auch auf Textkonstanten aus beiden Projekten.

Wie in Abschnitt 2.3 bereits erwähnt, nutzt die *micro-debug-gui* zur Lokalisierung eine eigene Datei-Hierarchie: Alle `.xml`-Dateien, die mit `text-gui` beginnen, enthalten Textkonstanten für die *micro-debug-gui*. Bei der Konfiguration wird allerdings die selbe Datei verwendet, die der *micro-debug* auch nutzt. Das hat für den Benutzer den Vorteil, dass er nur eine Datei zu pflegen hat.

Für die Pflege des Projekts *micro-debug-gui* bedeutet dies aber, dass sowohl die `.xml`-Dateien, die nur mit `text` beginnen, als auch die Konfigurationsoptionen des *micro-debug* in

der Datei `micro-debug.properties` aktualisiert werden müssen, wenn die Version des benutzten *micro-debug* geändert wird. Denn diese Dateien werden im Verzeichnis `java/main/ressources/` gepflegt und sind eigenständig, das heißt, sie werden nicht automatisch aktualisiert, wenn sich die entsprechenden Dateien im *micro-debug* ändern. Da der *micro-debug* und die *micro-debug-gui* aber derzeit beide zusammen entwickelt werden, ist dies noch kein Problem.

9.2. Bibliotheken

Wie gerade angemerkt, wird der *micro-debug* von der *micro-debug-gui* benutzt. Der Code des *micro-debug* muss also der *micro-debug-gui* vorliegen; derzeit wird der *micro-debug* der *micro-debug-gui* als `.jar`-Datei übergeben und dem Klassenpfad hinzugefügt. Dies ermöglicht es dem Benutzer, die verwendete Version des *micro-debug* auszutauschen.

Die *micro-debug-gui* enthält nur die GUI, das komplette Wissen über die *Mic-1* ist in der Bibliothek – dem *micro-debug*– enthalten. Diese Bibliothek ist also sehr wichtig und in der Regel nicht vorgesehen, um vom Benutzer durch neuere Versionen ersetzt zu werden.

Eine weitere Bibliothek, die der Benutzer erhält, ist das miglayout (siehe dazu ausführlich [AB11]). Diese Bibliothek wird benötigt, um die Oberflächenelemente zu positionieren. Das miglayout bietet eine komfortable und mächtige Schnittstelle, um Komponenten zu positionieren.

Die Bibliothek miglayout wird zwar auch zwingend benötigt, kann aber durchaus durch neuere Versionen ersetzt werden. Auch wenn prinzipiell das Risiko der Inkompatibilität zu künftigen Versionen besteht, ist das Risiko hier geringer, da nur sehr wenige Schnittstellen genutzt werden.

An dieser Stelle möchte ich auch das Testframework FEST erwähnen: Die dazugehörigen Bibliotheken werden zwar nicht an den Benutzer ausgeliefert, werden aber zur Entwicklung benötigt. Beim *micro-debug* wird dieses Framework auch schon verwendet, allerdings nur um eine flexiblere API (Programmierschnittstelle) als JUnit 4 nutzen zu können. Bei der *micro-debug-gui* wird FEST genutzt, um GUI-Tests zu schreiben.

Die GUI-Tests laufen in der Regel stabil, allerdings neigen sie häufiger zum Fehlschlagen, ohne tatsächlichen Fehler in der AUT (Application Under Test) oder dem Test selbst. Daher sollte beim Ausführen der Tests damit gerechnet werden, dass gelegentlich der ein oder andere GUI-Test fehlschlägt.

Da FEST-Tests die realen Ressourcen des Computers nutzen, kann dieser während die Tests laufen nicht benutzt werden, sonst schlagen die Tests fehl. Diese Eigenschaft verhindert außerdem, dass mehrere Tests parallel ausgeführt werden können – die Tests würden sich gegenseitig behindern. [Rö11, Abschnitt 3.2]

9.3. Packages

Ich habe nun die wichtigsten Klassen der *micro-debug-gui* vorgestellt. Um aber einen ausführlicheren Überblick über die *micro-debug-gui* zu bieten, möchte ich auch hier nochmal alle Packages erwähnen und erklären, welche Funktionalität die darin enthaltenen Klassen erfüllen. Die Packages sind im Einzelnen:

actions enthält alle *Actions*, die in der *micro-debug-gui* ausgeführt werden können. Hier ist auch die Klasse `AbstractExecuteOnWorkerThreadAction.java` enthalten, deren Unterklassen nicht auf dem EDT ausgeführt werden. In diesem Package gibt es noch ein Subpackage: `api`, das enthält einige Interfaces, um zyklische Abhängigkeiten zu verhindern.

argument enthält die Unterklassen von `AArgument.java`, die die gültigen Argumente der *micro-debug-gui* darstellen. Diese müssen an der Klasse `AArgument.java` registriert werden, um dem Benutzer zur Verfügung zu stehen.

commons enthält bisher nur die Klasse `WorkerThread.java`, also wie beim *micro-debug* Klassen, die zu keinem anderen Package zugeordnet werden konnten.

components enthält in den Subpackages alle Oberflächenelemente und das Hauptfenster – die Klasse `MainFrame.java`.

components.about enthält die Klasse `AboutFrame.java` und damit alle Komponenten, um den *Über*-Dialog anzuzeigen.

components.api enthält einige Interfaces, um zyklische Abhängigkeiten zu verhindern.

components.basic ist ein sehr großes Package mit vielen Klassen. Diese haben aber selten eigene Logik sondern bilden Unterklassen der bekannten Swing-Komponenten, von denen dann alle Komponenten in der *micro-debug-gui* ableiten können.

Diese Klassen haben beispielsweise erweiterte Konstruktoren, so dass jedes Oberflächenelement standardmäßig das `name`-Attribut gesetzt hat. Das `name`-Attribut ist für die automatisierten GUI-Tests wichtig.

components.code enthält die Komponenten, die zur Darstellung des Mikro-Assembler- und Assembler-Code benötigt werden:

ACodeArea.java eine abstrakte Klasse der Textkomponente, die den Code enthält. Hiervon leiten `MicroCodeArea.java` und `MacroCodeArea.java` ab, deren Namen bereits ausdrücken, welche Aufgabe sie erfüllen – den Code anzeigen.

ACodeFormatter.java eine abstrakte Klasse, die für die Syntaxhervorhebung im Code verantwortlich ist. Auch hier gibt es zwei Ableitungen: `MicroCodeFormatter.java` und `MacroCodeFormatter.java`.

LineNumberLabel.java die Komponente zur Darstellung der Zeilennummerierung.

Ruler.java die Komponente zur Darstellung der Breakpoints.

components.controller enthält die Klasse `MainController.java` und die entsprechenden *Controller* mit kleinerem Verantwortungsbereich, beispielsweise `RegisterController.java`.

components.start enthält die Komponenten, die im Programmablauf der *micro-debug-gui* vor der Klasse `MainFrame.java` aktiv sind: `Mic1Starter.java` und den `StartFrame.java`.

components.view enthält die Klasse `MainView.java` und die entsprechenden *Views*, die nur einen gewissen Teil darstellen, beispielsweise die `MicroCodeView.java`. Hier sind auch Komponenten, wie die `MainMenuBar.java` und `NumberStyleSwitcher.java` enthalten.

debug enthält die Klassen, die die Breakpoints behandeln. Hier gibt es eine zusätzliche Abstraktionsschicht zwischen der Klasse `BreakpointManager.java` aus dem *micro-debug* und beispielsweise `Ruler.java`, die diese *Handler* benutzt. Diese Schicht ist nötig, um auf die korrekten Zeilennummern zu schließen, was mit Hilfe der Klasse `LineNumberMapper.java` erreicht wird.

Die Klasse `LineNumberMapper.java` wird dazu genutzt, um zumindest für den Assembler-Code von den fortlaufenden Zeilennummern der Klasse `Ruler.java` auf die angezeigten Zeilennummern für den Benutzer schließen zu können.

Zusätzlich gibt es die Klasse `MicroLineBreakpointHandler.java` und `MacroLineBreakpointHandler.java`, außer zur Korrektur der Zeilennummern werden diese genutzt, um die korrekten Methoden an der Klasse `BreakpointManager.java` aufzurufen, der die Informationen über gesetzte Breakpoints enthält und eine Klasse aus dem *micro-debug* ist.

i18n enthält die Klasse `GuiText.java`, die die Textkonstanten für die *micro-debug-gui* bereitstellt.

listener enthält verschiedene *Listener*, die in der *micro-debug-gui* genutzt werden.

settings enthält Einstellungs-Enumerationen, die die Konfigurationsoptionen für die *micro-debug-gui* bereit stellen. Zusätzlich zu den aus dem *micro-debug* bekannten Klassen `IntegerSettings.java` und `InternalSettings.java` gibt es nun auch eine Klasse `KeyStrokes.java`, die Tastenkombinationen aus der Konfiguration ausliest.

Im Gegensatz zum *micro-debug* gibt es bei der *micro-debug-gui* keine wichtige Klasse, hier ist vielmehr das Zusammenspiel der verschiedenen Klassen wichtig zu verstehen. Am komplexesten ist dabei wahrscheinlich die Ausführung der *Actions* auf den zwei Threads, dem EDT und dem *WorkerThread*. Aber auch die Darstellung des Mikro-Assembler- und Assembler-Code und das Zusammenspiel zwischen Zeilennummern, Breakpoints, Zeilenhervorhebung und Code ist etwas komplexer.

10. Zusammenfassung

Ich habe in der vorliegenden Arbeit einen Einblick in den *micro-debug* und die *micro-debug-gui* gegeben. Das Ziel einen Debugger für die *Mic-1* zu entwerfen und implementieren konnte ich erreichen; in Abschnitt 3.3 habe ich ein mögliches Szenario gezeigt, wie der *micro-debug* eingesetzt werden kann.

Auch die GUI für den *micro-debug*– die *micro-debug-gui*– ist funktionsfähig und performant ausführbar. Einige Funktionen aus dem *micro-debug* sind in der *micro-debug-gui* nicht vorhanden, da diese nicht mehr benötigt werden, wie beispielsweise das Anzeigen des Mikro-Assembler- oder Assembler-Code. Andere Funktionen fehlen noch gänzlich, wie beispielsweise das setzen eines Wertes im Hauptspeicher oder das Verändern der Werte von Registern. Dafür bietet die *micro-debug-gui* allerdings einige Vorteile, zunächst bietet sie eine gute Übersicht – jederzeit ist der Code, die Register und der Hauptspeicher zu sehen. Aber auch ein funktionaler Vorteil ist nennenswert: die Möglichkeit den laufenden Prozessor zu unterbrechen.

Die beiden Aufgabenteile wurden als eigenständige Projekte entwickelt. Veränderungen an der GUI haben demnach keinen Einfluss auf den *micro-debug*. Zwei wichtige Ziele waren die Erweiterbarkeit und Lesbarkeit des Codes, damit andere Personen sich einarbeiten können und den Debugger weiterentwickeln können. Ich habe gezeigt, dass auch dieses Ziel erreicht wurde, unter anderem durch einen hohen Testaufwand: die Codeabdeckung der *micro-debug-gui* liegt über 93%[1], des *micro-debug* sogar über 97%[2].

Der Umfang der entwickelten Funktionen erfüllt nicht alle Anforderungen, die während der Entwicklung entstanden. Der Debugger ist aber nun eine gute Basis, um eben solche Funktionalitäten nach und nach zu entwickeln. Diese Arbeit und die Dokumentation im Code sollte ausreichen, dass auch Außenstehende künftig solche Funktionalitäten implementieren.

Ich hoffe nun auf viele Nutzer und Rückmeldungen, was am Debugger verbessert werden kann. Oder welche Funktionalität gut wäre, wenn sie der Debugger hätte.

[1] Gemessen mit *sonar* (siehe [son12]) am 05. Juni 2012.
[2] Gemessen mit *sonar* (siehe [son12]) am 02. Juni 2012.

Literaturverzeichnis

[AB11] AB, MiG I.: *MiG Layout Java Layout Manager for Swing and SWT*. `http://miglayout.com`. Version: 2011

[Fow99] *Refactoring: improving the design of existing code*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1999. – ISBN 0–201–48567–2

[Kut04] Kutzer, Thomas ; Berufsakademie Stuttgart (Hrsg.): *Mic1-Simulator*. : Berufsakademie Stuttgart, April 2004. – Studienarbeit

[Ont99] Ontko, Ray: *mic1*. `http://www.ontko.com/mic1/`. Version: 1999

[Ora04] Oracle: *JDK 5.0 Logging-related APIs & Developer Guides – from Sun Microsystems*. `http://docs.oracle.com/javase/1.5.0/docs/guide/logging/`. Version: 2004

[Ora10] Oracle: *Internationalization: Understanding Locale in the Java Platform*. `http://java.sun.com/developer/technicalArticles/J2SE/locale/`. Version: 2010

[Rö11] Rösch, Christian: *Automatisierte Tests von grafischen Benutzeroberflächen: Testerstellung und -management mit FEST*. 2011

[Rö12a] Rösch, Christian: *croesch/micro-debug – GitHub*. `https://github.com/croesch/micro-debug`. Version: 2012

[Rö12b] Rösch, Christian: *croesch/micro-debug-gui – GitHub*. `https://github.com/croesch/micro-debug-gui`. Version: 2012

[son12] sonar: *Sonar > Documentation*. `http://sonar.codehaus.org/documentation`. Version: 2012

[Str09] Stroetmann, Karl: *Rechner-Technik – Wintersemester 2010 / Sommersemester 2011*. Vorlesungsskript, September 2009

[TG98] Tanenbaum, Andrew S. ; Goodman, James R.: *Structured Computer Organization*. 4th. Upper Saddle River, NJ, USA : Prentice Hall PTR, 1998. – ISBN 0130959901

[Ver] Verfasser, Ohne: *Git - About*. `http://git-scm.com/about`

[Vog12] Vogel, Lars: *Java Logging API – Tutorial*. `http://www.vogella.com/articles/Logging/article.html`. Version: 2012

Erklärung

gemäß §5 (2) der „Studien- und Prüfungsordnung DHBW Technik“ vom 18. Mai 2009.
Ich versichere hiermit, dass ich die vorliegende Arbeit mit dem Thema

Design und Implementierung eines Debuggers für Mikro-Assembler-Programme

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Stuttgart, 10. Juni 2012
Ort, Datum

Christian Rösch