Diploma Thesis

Höhere Technische Bundeslehranstalt Leonding
Abteilung für Informatik

Visulation of the NoBeard Virtual Machine

Submitted by: **Egon Manya, 5AHIF**
Date: **April 4, 2018**
Supervisor: **Peter Bauer**

Declaration of Academic Honesty

Hereby, I declare that I have composed the presented paper independently on my own and without any other resources than the ones indicated. All thoughts taken directly or indirectly from external sources are properly denoted as such.

This paper has neither been previously submitted to another authority nor has it been published yet.

Leonding, April 4, 2018

Egon Manya

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorgelegte Diplomarbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Gedanken, die aus fremden Quellen direkt oder indirekt übernommen wurden, sind als solche gekennzeichnet.

Die Arbeit wurde bisher in gleicher oder ähnlicher Weise keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Leonding, am 4. April 2018

Egon Manya

Abstract

The target of this diploma thesis is to extend an available system programming environment which is called the NoBeard project. The project is developed to enable students to gain some experiences in the field of system programming by coding on assembler level with basic instructions. In addition, the reader of this thesis gets a basic knowledge about formal languages and their uses.

Initially, the project contained three main components:

- *The NoBeard Machine:* A stack based virtual machine with a small set of instructions. The main task of the machine is to execute NoBeard object files.
- *The NoBeard Assembler:* Provided to write programs for the NoBeard Machine.
- *The NoBeard Compiler:* Includes a dedicated programming language and the corresponding compiler to simplify the writing of codes for the NoBeard Machine.

The purpose of this thesis was to develop a proper concept of a graphical user interface for the virtual machine. The concept has to focus the didactic aims to enable users to explore the execution cycle of an assembler instruction, the execution of programs on assembler level, the monitoring of stack frames, the expression stack etc...

Furthermore, the system is extended with some debugger functions such as setting break points, stepping on assembler instruction level etc...

The initial version of the NoBeard project was developed in Java, therefore the graphical user interface had to be implemented with the Java FX framework. Primary, it was conceived with the NetBeans IDE on the ant build system, however the entire project was migrated to Maven and further developed using the IntelliJ IDEA environment.

Zusammenfassung

Das Ziel dieser Diplomarbeit ist es, eine verfügbare Systemprogrammierumgebung zu erweitern, die als NoBeard-Projekt bezeichnet wird. Das Projekt wurde entwickelt, um den Studierenden zu ermöglichen, einige Erfahrungen im Bereich der Systemprogrammierung zu sammeln, indem sie auf Assembler-Ebene mit grundlegenden Anweisungen kodieren. Darüber hinaus erhält der Leser dieser Arbeit ein grundlegendes Wissen über formale Sprachen und ihre Verwendung.

Ursprünglich enthielt bestand das Projekt aus drei Hauptkomponenten:

- *Die NoBeard Maschine:* Eine stack-basierte virtuelle Maschine mit einer kleinen Menge von Anweisungen. Die Hauptaufgabe der Maschine besteht darin, NoBeard-Objektdateien auszuführen.
- *Der NoBeard Assembler:* Zum Schreiben von Programmen für die NoBeard Maschine.
- *Der NoBeard Compiler:* Enthält eine spezielle Programmiersprache und den entsprechenden Compiler, um das Schreiben von Codes für die NoBeard Maschine zu vereinfachen.

Der Zweck dieser Arbeit war es, ein geeignetes Konzept für eine grafische Benutzeroberfläche für die virtuelle Maschine zu entwickeln. Das Konzept muss die didaktischen Ziele fokussieren, um Benutzern zu ermöglichen, den Ausführungszyklus eines Assemblerbefehl zu erkunden, die Ausführung von Programmen auf Assembler-Ebene, die Überwachung von Stack-Frames etc. . .

Darüber hinaus wurde das System um einige Debugger-Funktionen, wie das Setzen von Breakpoints, erweitert

Die ursprüngliche Version des NoBeard-Projekts wurde in Java entwickelt, daher musste die grafische Benutzeroberfläche mit dem Java FX-Framework implementiert werden. Primär wurde es mit der NetBeans-IDE auf dem ant-Build-System konzipiert, jedoch wurde das gesamte Projekt auf Maven migriert und unter Verwendung der IntelliJ IDEA-Umgebung weiterentwickelt.

Contents

1 Introduction **3**
- 1.1 Initial Situation . . . 3
- 1.2 Goals . . . 3
- 1.3 Structure of the Thesis . . . 3

2 Formal languages **4**
- 2.1 Overview . . . 4
- 2.2 Alphabets and Strings . . . 4
  - 2.2.1 Alphabet . . . 4
  - 2.2.2 String . . . 5
  - 2.2.3 Concatenation . . . 5
- 2.3 Production Rules and Grammars . . . 5
  - 2.3.1 Kleene Star . . . 5
  - 2.3.2 Production Rule . . . 6
  - 2.3.3 Definition of a Grammar . . . 6
  - 2.3.4 Syntax of formal grammars . . . 7
  - 2.3.5 Terminal and Non-Terminal Symbols . . . 7
  - 2.3.6 Derivation . . . 7
  - 2.3.7 Example . . . 8
- 2.4 Languages . . . 8
- 2.5 Syntax Notations . . . 9
  - 2.5.1 Classical . . . 9
  - 2.5.2 EBNF . . . 9
  - 2.5.3 Syntax Diagrams . . . 10

3 The NoBeard Machine Architecture **11**
- 3.1 The NoBeard Machine . . . 11
  - 3.1.1 Program Memory . . . 11
  - 3.1.2 Data Memory . . . 11
  - 3.1.3 Call Stack . . . 14
  - 3.1.4 Control Unit . . . 15
- 3.2 Binary File Format . . . 15
- 3.3 Instructions . . . 16

3.3.1 Load and Store Instructions . . . 16
3.3.2 Integer Instructions . . . 18
3.3.3 Control Flow Instructions . . . 19
3.3.4 IO-Instructions . . . 20
3.4 NoBeard Assembler . . . 20

4 User Manual **21**
4.1 Overview . . . 21
4.2 The UI Components . . . 21
4.2.1 Control Window . . . 21
4.2.2 Output Window . . . 22
4.2.3 Program Window . . . 23
4.2.4 Data Memory Window . . . 23
4.2.5 Input Window . . . 23
4.3 Loading and Running a Program . . . 23
4.4 Debugging . . . 23
4.5 Data Visualization . . . 24
4.5.1 Converting Data to Character . . . 25
4.5.2 Converting Data to Integer . . . 26
4.5.3 Multiple Conversion . . . 26

5 Implementation **29**
5.1 Used Technologies . . . 29
5.1.1 IntelliJ IDEA . . . 29
5.1.2 Maven . . . 29
5.1.3 JavaFX . . . 30
5.1.4 Scene Builder . . . 30
5.2 Supporting NoBeard Packages . . . 30
5.3 Architecture of the User Interface . . . 31
5.3.1 The View . . . 31
5.3.2 The Controller . . . 33
5.4 Synchronization of NoBeard Machine and GUI . . . 36
5.5 Handling of Breakpoints . . . 37
5.6 Visualisation of DataMemory . . . 39

A Additional Information **46**

B Individual Goals **47**

Chapter 1

Introduction

1.1 Initial Situation

The NoBeard environment is developed to support the courses of Theoretical Informatics conducted for third grade students of the Department of Informatics at the HTBLA Leonding. The project was introduced by Prof. DI Peter Bauer with the purpose that students get the possibility to gain some experience in the field of system programming.

The NoBeard tools consists of a stack based virtual machine, an assembler and a basic compiler with the corresponding language. The components here are developed in the simplest way to facilitate the operation of the user as much as possible.

1.2 Goals

The general goal of this thesis was to extend the already existing NoBeard project by a graphical user interface to make the use of the virtual machine more simple. It was the highest priority to meet all the usability requirements. Amongst other things, it includes a better and easier use of the virtual machine compared to command line interfaces. Through the advantages of a GUI users are able to get a better overview of an assembler program execution and its data memory state.

The other major goal of this continuation project was the introducing of a debugger into this system. This tool need to cover nearly all basic debugger functionalities for assembler programs such as toggling breakpoints or stepping through assembler instruction. Of course, the concept should support as well CLI as GUI devices.

1.3 Structure of the Thesis

1.3.1 Formal Languages

This chapter is intended to provide a rough overview of formal languages, their structure (grammars), and usefulness. It gives a short description about the syntax of a language.

Chapter 2

Formal languages

2.1 Overview

A formal language is an abstract language that, unlike concrete languages, often does not focus on communication but on mathematical use. A formal language consists of a certain set of symbol strings (generally strings) that can be assembled from a set of characters or symbols. Formal languages are used in linguistics, logic and theoretical computer science.

Formal languages are suitable for the (mathematically) precise description of the handling of character strings. For example, data formats or entire programming languages can be specified. Together with a formal semantics, the defined character strings have a meaning. With a programming language, a programming instruction (as part of the formal language) can be assigned a unique machine behaviour (as part of the semantics).

Based on formal languages, however, it is also possible to define logic calculi with which one can draw mathematical conclusions. In conjunction with formally defined programming languages, calculi can help to check programs for their correctness.

All major information here are from the book [**?**] written by Rechenberg Peter and Mössenböck Hanspeter. This book describes in detail how a compiler works, especially with the usage of formal grammars and syntax analysis

2.2 Alphabets and Strings

2.2.1 Alphabet

An alphabet (sometimes called a Vocabulary) is a non-empty and finite set of elements. Alphabets are often denoted by upper-case letters A or V and sometimes as upper-case Greek letters like Σ. Examples of common alphabets are e.g. the machine text alphabet ASCII or the binary bit 0 or 1.

2.2.2 String

A string (or word) is a finite sequence of elements, called symbols or letters selected from a particular finite set which is the alphabet. The length of a string ω is denoted as $|\omega|$ which gives the number of symbols in the string. The empty string is a special word which has no symbols and is denoted by ε or λ. This means a string ω is called an empty string if $|\omega| = 0$.

2.2.3 Concatenation

The basic operation of strings is concatenation, i.e., writing strings as a junction. Let ω_1 and ω_2 be two strings. Then the concatenation of ω_1 and ω_2 makes a new string α containing all the symbols of ω_1 in order followed by all symbols of ω_2 in order. This is usually written as $\alpha = \omega_1\omega_2$. Concatenations in the alphabet $\{x, y, z\}$ can be:

$\omega_1 = xxzyyx$	$\omega_2 = zxxz$	$\omega_1\omega_2 = xxzyyxzxxz$
$\omega_1 = xxzyyx$	$\omega_2 = \varepsilon$	$\omega_1\omega_2 = xxzyyx$
$\omega_1 = \varepsilon$	$\omega_2 = zxxz$	$\omega_1\omega_2 = zxxz$

The concatenation on an alphabet has two important properties:

- Assiciativity: $\omega_1(\omega_2\omega_3) = (\omega_1\omega_2)\omega_3$
- An identity element ε: i., e., $\omega\varepsilon = \varepsilon\omega = \omega$

In some cases string notation can be simplified by the following rules:

a^n	For strings containing n same symbols, e.g., $a^3 = aaa$
ε	The empty string containing null symbols
a^+	For strings with set of $(a^n : n \geq 1)$, e.g., $a^+ = \{a, aa, aaa, aaaa, \ldots\}$
a^*	For strings with set of $(a^n : n \geq 0)$, e.g., $a^* = \{\varepsilon, a, aa, aaa, aaaa, \ldots\}$

2.3 Production Rules and Grammars

2.3.1 Kleene Star

The Kleene star set Σ^* of an alphabet Σ is a language that contains all the words of the alphabet. It can be defined by means of structural induction. Firstly, in the beginning of the induction the empty word ε is defined. Then in the induction step it is defined that each string is being one element in the Kleene star set. To define the sets Σ_i recursively, we use the following components:

- **Basis Clause:** $\Sigma_0 = \{\varepsilon\}$
- **Inductive Clause:** $\Sigma_{i+1} = \{\omega v \mid \omega \in \Sigma_{i-1} \wedge v \in \Sigma\}$

The Kleene star is defined then by $\Sigma^* = \bigcup_{i \in \mathbb{N}_\nu} \Sigma_i$ In other words, the Kleene Star of an alphabet is the set of all strings that can be built out of an alphabet.

The Kleene star set L^* of a language L is the union of all its potential languages (repeated concatenation of languages):

$$L^* = \bigcup_{i \in \mathbb{N}_\nu} Ł^i$$

Where $L^0 = \{\varepsilon\}$ and $Ł^{n+1} = L^n L$.

2.3.2 Production Rule

A grammar rule has the form:

$$X \to \alpha \qquad \text{With} \quad X \in V \quad \text{and} \quad \alpha \in V^*$$

It is read "X is defined as α" or "X can be replaced by α" and means that in a string containing X, X can be replaced by α. Several rules with the same left side such as:

$X \to \alpha_1$
$X \to \alpha_2$
$X \to \alpha_3$
are usually grouped together by separating the alternatives with the character "|":

$$X \to \alpha_1 \mid \alpha_2 \mid \alpha_3$$

All symbols appearing on the left side of a production rule or generally a grammar are non-terminals V_N and all others are terminal symbols V_T. The rules describe substitutions of non-terminal symbols by strings. In the formal language theory these substitutions are called production rules. A grammar is defined by production rules that indicate which symbols can replace which other symbols. These rules can be used to generate strings or parse them.

Parsing is the process of recognizing an utterance (a string in natural languages) by breaking it up into a series of symbols and analysing them with the grammar of the language. Most languages have structured the meanings of their utterances according to their syntax.

To create a string in the language, we start with a string that consists only of a start symbol. This start symbol S appears on at least one left side of the production rules. All symbols on the left and right sides are the Vocabulary. From the starting symbol S, each symbol will be replaced repeatedly by strings, according to the rules of the grammar, until a string is created which contains only terminal symbols.

2.3.3 Definition of a Grammar

A grammar $G(S)$ is a finite, non-empty set of production rules. The rules describe how to build strings from the alphabet of the language that are valid according to the syntax of the language. A grammar does not describe the meaning of the strings or what can be done with them in any context, just their form.

2.3.4 Syntax of formal grammars

A formal grammar is represented by the 4-tuple $G = (N, \Sigma, P, S)$ wherein:

- N: a finite set of non-terminal symbols that is separated from the G formed strings.
- Σ: a subset of V, also called alphabet whose elements are called terminal symbols
- P: a finite set of production rules where each rule has the form: $(\Sigma \cup N)^* N (\Sigma \cup N)^* \to (\Sigma \cup N)^*$
- $S \in N$: the start symbol

The set $N = V \setminus T$ is the set of non-terminal symbols (also called meta-symbols), in particular the start symbol belongs to it. The word on the left side of the rule pairs may not consist exclusively of terminal characters, which can also be expressed by a concatenation: $(V^* \setminus T^*) = V^* N V^*$.

2.3.5 Terminal and Non-Terminal Symbols

The grammatical structure of a statement, a program or generally of a string is a tree, the so-called syntax tree. There are two types of symbols:

- **Terminal symbols:** these are the symbols of the sentence itself, i.e. the symbols that can not be further decomposed. In other words they are the "leaves" of the tree and are donated with lower-case letter.
- **Non-Terminal symbols:** these are all other symbols with upper-case letters.

With $V = V_N \cup V_T$ we denote the union of the alphabets of terminal and non-terminal symbols. Each tree contains an excellent non-terminal symbol, the start symbol or the root from which the whole tree originates. The allowed structures of syntax trees and thus sentences of a formal language are described by "grammars".

2.3.6 Derivation

Let a string α generating directly a string β, written $\alpha \Rightarrow \beta$. Given a formal grammar G with a rule $A \to \varphi$. If there are the strings ω_1 and ω_2 with the definition $\alpha = \omega_1 A \omega_2$, then we can obviously generate a new string $\beta = \omega_1 \varphi \omega_2$.

A sequence of direct generations is described with the symbols $\Rightarrow^+$ and $\Rightarrow^*$. When a string α produces another string β with the form $\alpha \Rightarrow^+ \beta$ then it is about a multiple string generation. A sequence of applications of the rules of a grammar that produces a finished string of terminals is mostly called as derivation.

$$\alpha = \omega_0 \Rightarrow \omega_1 \Rightarrow \omega_2 \ldots \Rightarrow \omega_n = \beta \qquad \text{with} \quad n \geq 1$$

In case, that $\alpha \Rightarrow^+ \beta$ or $\alpha = \beta$ we write $\alpha \Rightarrow^* \beta$. This means α generates or is equal to β.

2.3.7 Example

Given the grammar $G = (\{S, A, B\}, \{a, b, c\}, P, S)$ with the non-terminal symbols S, A, B, the terminal symbols a, b, c and the production rule set P:

$S \to Ac$

$A \to aB \mid BBb$

$B \to b \mid ab$

Now the following types of strings can be generated by derivation:

$$\begin{array}{llll} S \Rightarrow Ac & \Rightarrow aBc & \Rightarrow abc & \\ & & \Rightarrow aabc & \\ & \Rightarrow BBbc & \Rightarrow bBbc & \Rightarrow bbbc \\ & & & \Rightarrow babbc \\ & & \Rightarrow abBbc & \Rightarrow abbbc \\ & & & \Rightarrow ababbc \end{array}$$

Therefore it is $L(G) = \{\text{“}abc\text{”}, \text{“}aabc\text{”}, \text{“}bbbc\text{”}, \text{“}babbc\text{”}, \text{“}abbbc\text{”}, \text{“}ababbc\text{”}\}$

If $G(S)$ is a formal grammar with a start symbol S then the set

$$L(G(S)) = \{\alpha : S \Rightarrow^* \alpha \wedge \alpha \in V_T^*\}$$

is called the language of $G(S)$.

2.4 Languages

Formally, a language L is defined as a set of strings over some finite alphabet. Specific examples of languages are finite languages with a finite number of words.

For languages we use the usual set-theoretic notations like:

- $\subseteq$ (inclusion),
- $\supseteq$ (proper inclusion),
- $\cup$ (union),
- $\cap$ (intersection),

A formal language L over an alphabet Σ is an arbitrary set of words about Σ, i.e., $L \subseteq \Sigma^*$.

If a string ω belongs to a language L it is denoted by $\omega \in L$, as usual. There are also “negated” relations such as $\not\subseteq$, $\not\subset$ and $\notin$.

Examples of languages:

- Above the letter A-Z including umlauts and punctuation marks. The set of all grammatically correct German sentences
- The grammar G(Java) defining the programming language Java. L(G(Java)) is then the set of all syntactically correct Java programs
- Above the ASCII character set. The set of all C programs

2.5 Syntax Notations

Syntax notations are compact formal meta-languages for representing context-free grammars. This includes the syntax of common higher-level programming languages. It is also used for the notation of instruction sets and communication protocols.

There are essentially three methods for the syntax description and they will be explained in the following sections. They describe the syntax as a grammar in the form of so-called syntax rules. They form a well-understood formal system and find their justification in the theory of formal languages.

2.5.1 Classical

In computer science different spellings are used for grammar rules. Nevertheless, the previously used syntax has the following identifiers:

- Terminals in lower-case letters
- Non-terminals in upper-case letters
- Separator: $\rightarrow$
- Alternatives: |

The language of propositional logic is defined with the classical notation system of formal languages as follows.

$$\begin{array}{lcl} F & \rightarrow & I|\neg F|(F \wedge F)|(F \vee F) \\ I & \rightarrow & a|b|c \end{array}$$

2.5.2 EBNF

The extended Backus-Naur form meta-syntax notations is a very commonly used form for defining grammars that is equivalent to the syntax diagrams and substitution rules. It is very similar to the previously used spelling in the productions, but has the advantage that it is easier machine-readable. The EBNF has the following rules:

- Terminals under double quotes
- Non-terminals in meaningful words written in camel case
- Separator: =
- Alternatives: |
- Each rule ends with a period
- Options: $[A]$ means A or empty ϵ
- Repetition: $\{A\}$ means ϵ or A or AA or AAA ...

- Brackets for grouping

As an example we can take a look at the EBNF syntax of the expressions in the programming language Modula 2 from [**?**].

$$
\begin{array}{lcl}
expression & = & [\text{“}+\text{”}|\text{“}-\text{”}]term\{(\text{“}+\text{”}|\text{“}-\text{”})term\} \\
term & = & factor\{(\text{“}*\text{”}|\text{“}/\text{”})factor\} \\
factor & = & c|v|\text{“}(\text{“}expression\text{”})\text{”}
\end{array}
$$

2.5.3 Syntax Diagrams

A syntax diagram is used in theoretical computer science to graphically represent the syntax of a rule set. In particular, formal languages up to the class of context-free languages, and thus the syntax of programming languages due to the subset property, can be represented in a syntax diagram.

Each extended Backus-Naur form (EBNF) can be converted with the help of enclosed translation one on one in a syntax diagram. Figure 2.1 shows how the previous EBNF example could look like after a success translation into a syntax diagram.

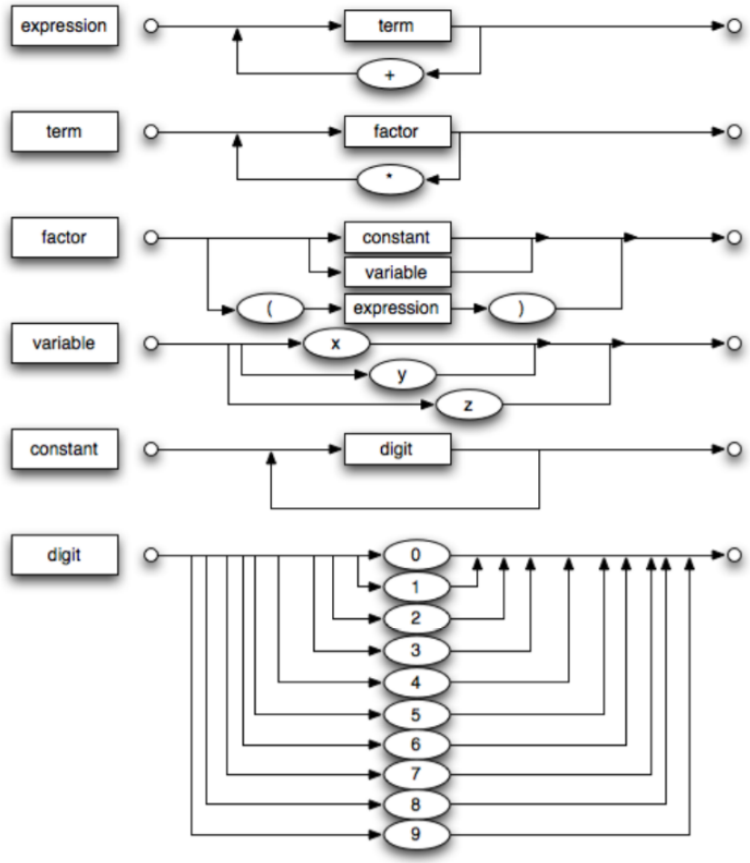

Figure 2.1: Example of a syntax diagram

Chapter 3

The NoBeard Machine Architecture

3.1 The NoBeard Machine

The NoBeard Machine is a virtual machine with an instruction set of 31 instructions which is pretty easy to understand compared to instruction sets of real life machines which have significant larger instruction sets. The machine is purely stack based such that the structure of each instruction is easy to grasp and to follow. The machine has a word width of four bytes and is being target for NoBeard programs. The NoBeard Machine consists of the following components:

- Program Memory
- Data Memory
- Call Stack
- Control Unit

Figure 3.1 shows these components and how they are related. The components will be described in the following sections.

3.1.1 Program Memory

The program memory stores instructions with their corresponding opcodes and operands. The memory is byte addressed with a specified maximum size. Memory access outside the valid range lead to a `ProgramAddressError`.

3.1.2 Data Memory

The data memory is a byte addressed storage and stores variables as follows:

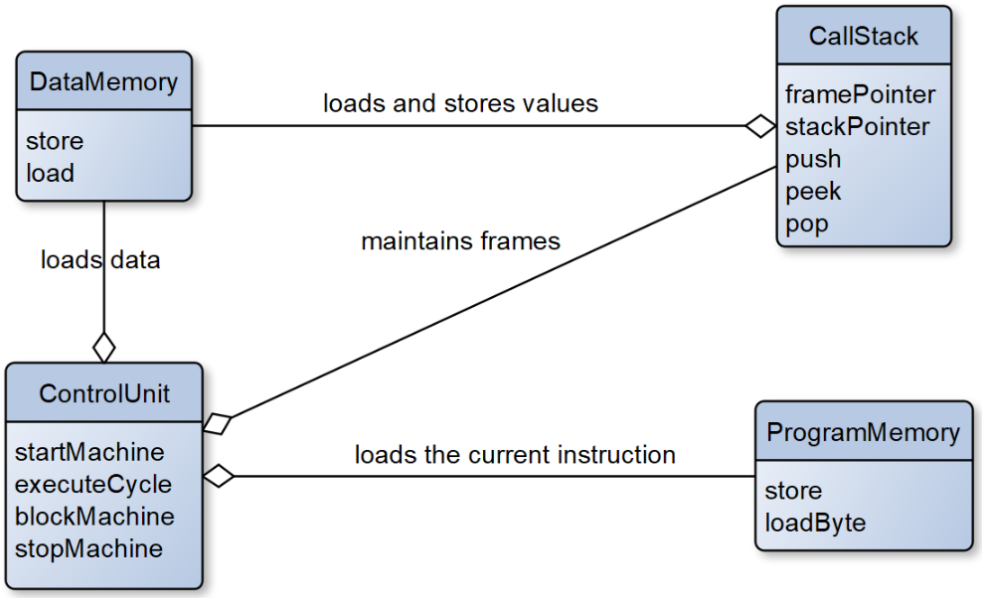

Figure 3.1: Components of the NoBeardMachine Architecture

- **Characters** are one-byte values and consume exactly one byte in memory, i.e., no alignment is done.
- **Integers** are four-byte values and stored in little endian order. Negative integer values are stored as Two's Complement (see [**?**]).
- **Booleans** are four-bytes values and are stored as the integer 0 for false and the integer 1 for true.

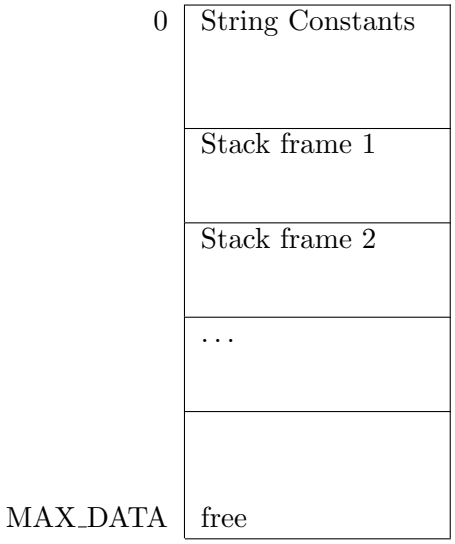

Figure 3.2: Data Memory of the NoBeard Machine

As figure 3.2 shows, the data memory is separated into two parts, string constants and to stack frames of the currently running functions.

String Constants

String constants are in the top segment of the data memory. It includes all strings that have to be printed on the console. The storing of string constants happens exactly after the opening of a binary file on the virtual machine.

Stack Frames

After the constant memory the stack frames are maintained. Whenever a function gets called a new frame is added. It holds data for the function arguments, local variables and its expression stack. As soon as a function ends, its frame is removed.

Address	Content	Remark
0	0	frame pointer of frame 0
	...	
32	13	local int in frame 0
36	0	static link to frame 0 (start of frame 1)
	...	
68	17	local int in frame 1
72	42	local int in frame 1
76	36	static link to frame 1 (start of frame 2)
	...	
108	'D'	
109	61	
MAX_DATA	free	

Figure 3.3: Snapshot of a call stack with three frames

Figure 3.3 shows a pretty good example from [**?**]. Here we can see that the memory is working with three frames. Frame 0 starts at address 0. The first 32 bytes of each frame are reserved for administrative data like the static link and the dynamic link to the surrounding frame, the return value, etc. Address 32 holds the value of a local variable in frame 0. At address 36 frame 1 starts with the address to its statically surrounding frame, i.e, the function (or unit or block) represented by frame 0 is defining the function (or block) represented by frame 1. Frame 1 defines two local values at addresses 68 and 72.

3.1.3 Call Stack

By structuring the data memory as a stack the call stack is needed as an abstraction to the data memory. With the help of different functions the call stack is able to add and remove frames from the stack and to maintain the expression stack. Data needed for each statement gets stored in the expression stack. It grows and shrinks as needed and is empty at the end of each NoBeard statement. The stack is addressed word-wise only. Functions like `push()`, `peek()` and `pop()` are provided for the maintenance of values on the stack. The call stack has the two major components:

- **Stack Pointer:** Address of the start of the last used word on the stack.
- **Frame Pointer:** Address of the first byte of the currently running function's stack frame.

3.1.4 Control Unit

The control unit is responsible for the program work flow. It executes one machine cycle in three steps, it fetches, decodes and operates the current instruction. Depending on some instruction, the control unit also affect the state of the machine. To achieve these steps it has to work with the following components:

- **Program Counter:** Start address of the next instruction to be executed.
- **Machine State:** The NoBeard machine has four different states and is always in one of them.
  - `running`: The machine runs
  - `stopped`: The machine stops. Usually when the end of program is reached.
  - `blocked`: The machine pauses. Mostly when a breakpoint is placed by the user.
  - `error`: Error state

As already mentioned the machine has a firmly defined execution cycle:

1. Fetch instruction
2. Decode instruction
3. Execute instruction

The very first instruction is fetched from a specified starting program counter which is provided as an argument when starting the program. From this point of time onwards the program is running until the machine state changes from run. There are two options to interrupt the machine from running state. First, if it gets interrupted by a breakpoint typically set by the debugger and second, if a `halt` instruction gets executed.

3.2 Binary File Format

The virtual machine runs only NoBeard object files with extension `.no` which can be generated by the NoBeard Assembler or NoBeard Compiler. As figure 3.4 shows NoBeard binaries are separated into three parts, a header part, a string storage and a program segment. The first six bytes are reserved for the header part which holds information about the file. This information includes the file identifier and the version of the file. After the header follows the string storage where a stream of constants are stored. Finally, the program segment which is organized like the string storage deals with the storing of machine instructions.

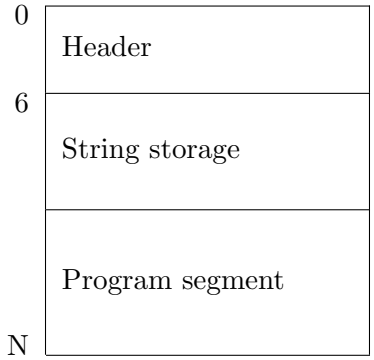

Figure 3.4: NoBeard Binary File Format

3.3 Instructions

NoBeard instructions are of a different length and each has an opcode and operands of an amount between 0 and 2. The first byte of all instructions is reserved for the opcode, which is the identifier used to identify the instruction on machine language level. The remaining bytes, if any, are assigned to the operand(s) of the instruction. Each of the following subsections explains these instructions in four categories. The underlined title is the shorthand that identifies the instruction on assembler level. Then follows a table showing the size of the instruction and which bytes carry which information. Finally, each one has also a short explanation in human language.

3.3.1 Load and Store Instructions

lit

Byte 0	Byte 1	Byte 2
0x01	Literal	

Operation: Pushes a value on the expression stack.

la

Byte 0	Byte 1	Byte 2	Byte 3
0x02	Displacement	DataAddress	

Operation: Loads an address on the stack.

lv

Byte 0	Byte 1	Byte 2	Byte 3
0x03	Displacement	DataAddress	

Operation: Loads a value on the stack.

lc

Byte 0	Byte 1	Byte 2	Byte 3
0x04	Displacement	DataAddress	

Operation: Loads a character on the stack.

lvi

Byte 0	Byte 1	Byte 2	Byte 3
0x05	Displacement	DataAddress	

Operation: Loads a value indirectly on the stack.

lci

Byte 0	Byte 1	Byte 2	Byte 3
0x06	Displacement	DataAddress	

Operation: Loads a character indirectly on the stack.

inc

Byte 0	Byte 1	Byte 2
0x1D	Size	

Operation: Increases the size of the stack frame by `Size`.

sto

Byte 0
0x07

Operation: Stores a value on an address.

stc

Byte 0
0x08

Operation: Stores a character on an address.

3.3.2 Integer Instructions

neg

Byte 0
0x0B

Operation: Negates the top of the stack.

add

Byte 0
0x0C

Operation: Adds the top two values of the stack.

sub

Byte 0
0x0D

Operation: Subtracts the top two values of the stack.

mul

Byte 0
0x0E

Operation: Multiplies the top two values of the stack.

div

Byte 0
0x0F

Operation: Divides the top two values of the stack.

mod

Byte 0
0x10

Operation: Calculates the remainder of the division of the top values of the stack.

not

Byte 0
0x11

Operation: Calculates the remainder of the division of the top values of the stack.

3.3.3 Control Flow Instructions

fjmp

Byte 0	Byte 1	Byte 2
0x16	NewPc	

Operation: Sets `pc` to `newPc` if stack top value is false.

tjmp

Byte 0	Byte 1	Byte 2
0x17	NewPc	

Operation: Sets `pc` to `newPc` if stack top value is true.

jmp

Byte 0	Byte 1	Byte 2
0x18	NewPc	

Operation: Unconditional jump: Sets `pc` to `newPc`.

break

Byte 0
0x20

Operation: Breaks the machine. Especially used for the debugging function.

halt

Byte 0
0x1F

Operation: Halts the machine.

3.3.4 IO-Instructions

in

Byte 0	Byte 1
0x19	Type

Operation: Reads data from the terminal. Depending on `Type` different data types are read:

- 0: An `int` is read and stored at the address on top of the stack. After execution the value 1 is pushed if an integer was read successfully, otherwise 0 is pushed.
- 1: A `char` is read and stored at the address on top of the stack. After execution the value 1 is pushed
- 2: a `string` with a specific length is read

out

Byte 0	Byte 1
0x1A	Type

Operation: Writes data to the terminal. Depending on `Type` different data types are printed:

- 0: An `int` with a specific column width is printed
- 1: A `char` with a specific column width is printed
- 2: a `string` with a specific column width is printed
- 3: a new line is printed

3.4 NoBeard Assembler

To write programs for the NoBeard machine an Assembler is provided. NoBeard Assembler files are separated in two blocks, which is called the string constants and the assembler program. The files have the extensions `.na` for NoBeard Assembler. The string constants are stored between two double quotes and has to be located at the beginning of the file. There is no way to address a single constant. So, if we use a string constant in the assembler program, we have to specify the starting address of the string constant and the length needed in the program. Assembler programs contain a sequence of assembler instructions like `lit` for load or `out` for print. After the opcode of the instruction follows the operands, if they are needed as already described in section 3.3.

Chapter 4

User Manual

4.1 Overview

This chapter describes how to use the graphical user interface of the NoBeard virtual machine and serves at the same time as a user manual. With the NoBeard Machine users are able to load and run NoBeard object files with just a view clicks. The integrated debugger of the virtual machine gives them also the possibility to debug these programs by setting breakpoints and stepping through the program in one by one assembler instructions. Another big feature is the data visualization which gives users a whole view of the data memory.

4.2 The UI Components

As shown in figure 4.1 the NoBeardMachine contains the following five windows:

- **Control Window:** A window with a set of buttons for user interactions.
- **Output Window:** A terminal showing outputs of programs and user inputs.
- **Program Window:** Shows the actually opened program and its running flow.
- **Data Memory Window:** A visualized call stack of a debugged program.
- **Input Window:** A single-line field for user inputs.

4.2.1 Control Window

This window consists of five buttons that gives the possibility to open, run, stop programs, step between instructions or continue execution from a breakpoints until another one.

- **Open file:** Opens a file chooser dialog where the target object file can be selected.
- **Run:** Executes the program from the first instruction.

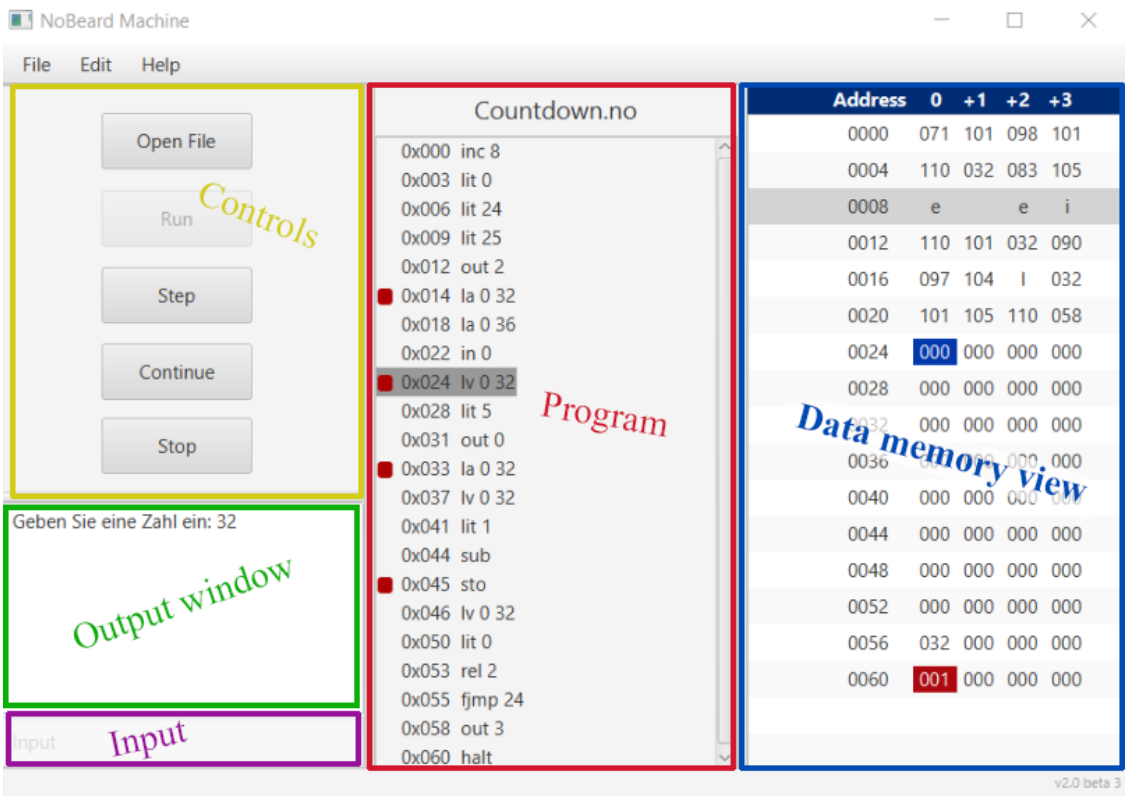

Figure 4.1: Components of the NoBeardMachine

- **Step:** Allows the user to step one single assembler instruction further in the program flow. This button is only enabled if the machine is in state `blocked`.
- **Continue:** When the program execution gets interrupted by a breakpoint this button enables users to continue the execution of the program until the next breakpoint or the end of the program. If there is no breakpoints left it continues the process until the end.
- **Stop:** Stops the execution and sets the machine into the `stopped` state.

4.2.2 Output Window

The output window is a non-editable text area which simulates a terminal. Program outputs that are coming from a NoBeard `out` instruction are shown here. Submitted inputs are also visible here after a successful submit.

4.2.3 Program Window

The program window shows the assembler instructions of the loaded program with the corresponding start address of each instruction. When clicking on the empty area in front of the instruction's address a breakpoint can be set or unset.

4.2.4 Data Memory Window

A ListView filled with raw data from the data memory. Every line of the ListView contains four byte data with the belonging addresses. The raw data can be converted into different formats like characters or integers to make it better readable to humans. Furthermore the frame pointer and the stack pointer of the currently running frame are highlighted. Each byte can be converted to a character and a whole line can be translated to four characters or to a single four byte integer. Data with blue background color highlights the frame pointer. Red background color stands for the stack pointer.

4.2.5 Input Window

The Input window is a TextField where inputs from users can be handled. It is only enabled when the machine is executing an `in` instruction. To submit an input, the "ENTER" key has to be pressed and then the entered text will be attached to the Output window.

4.3 Loading and Running a Program

After starting the NoBeardMachine, a NoBeard object file has to be loaded by a click on the "Open File" button. Than a file chooser dialog appears where the user can choose the desired file. Afterward the window shows the assembler code and the title of the opened program which is now ready for execution.

The Program window lists the assembler instructions with the belonging operators and addresses in decimal form. By hitting the "Run" button, the machine executes the loaded program. Output results can be seen in the Output window. If the program runs against an input instruction, the machine stops and requests the user for an input which can be done at the Input window. To submit an input, the Enter key has to be pressed. After the user pressed the Enter key the program continues its execution.

4.4 Debugging

To debug a program the user has to set breakpoints which interrupts the program flow and allows the user to inspect the current state of the running program. These breakpoints can be placed by clicking on the address or onto the empty space in front of the address with the instruction where the program flow has to be interrupted.

After an interruption caused by a breakpoint, the user is able to step one line further or continue execution until to the next breakpoint. Stepping is handled by the "Step"

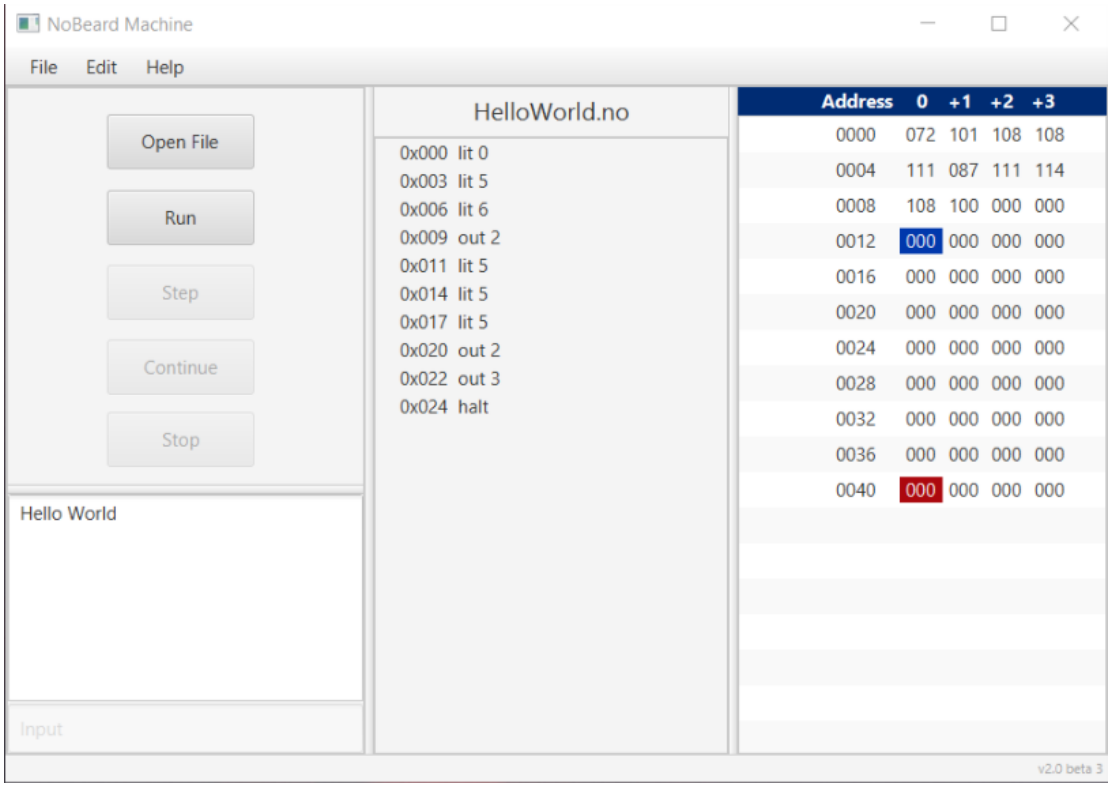

Figure 4.2: Executed "Hello World" program

button as shown in figure 4.3. By clicking the "Continue" button, the program runs from the current line until the next breakpoint. If there is not any breakpoint left from the current line, it runs until the end of the program. Optionally, the user is able to stop the program during the execution. This could be achieved by clicking the "Stop" button.

4.5 Data Visualization

On the right side of the window is the visualisation of the data memory in a ListView form. This window lists data byte-wise from the data memory of the machine. Each line of the ListView has a content of an address given in decimal notation followed by four bytes of raw data. The memory is separated in two parts. The list starts on the top with the string constants followed by stack frames of the currently running functions. While the frame pointer is highlighted with a blue background, the stack pointer is signed with a red background. The user has also the possibility to convert raw data to characters or integers. With a right click on the selected line of the list, a context menu opens. The context menu includes the following functions which will be described in the sections

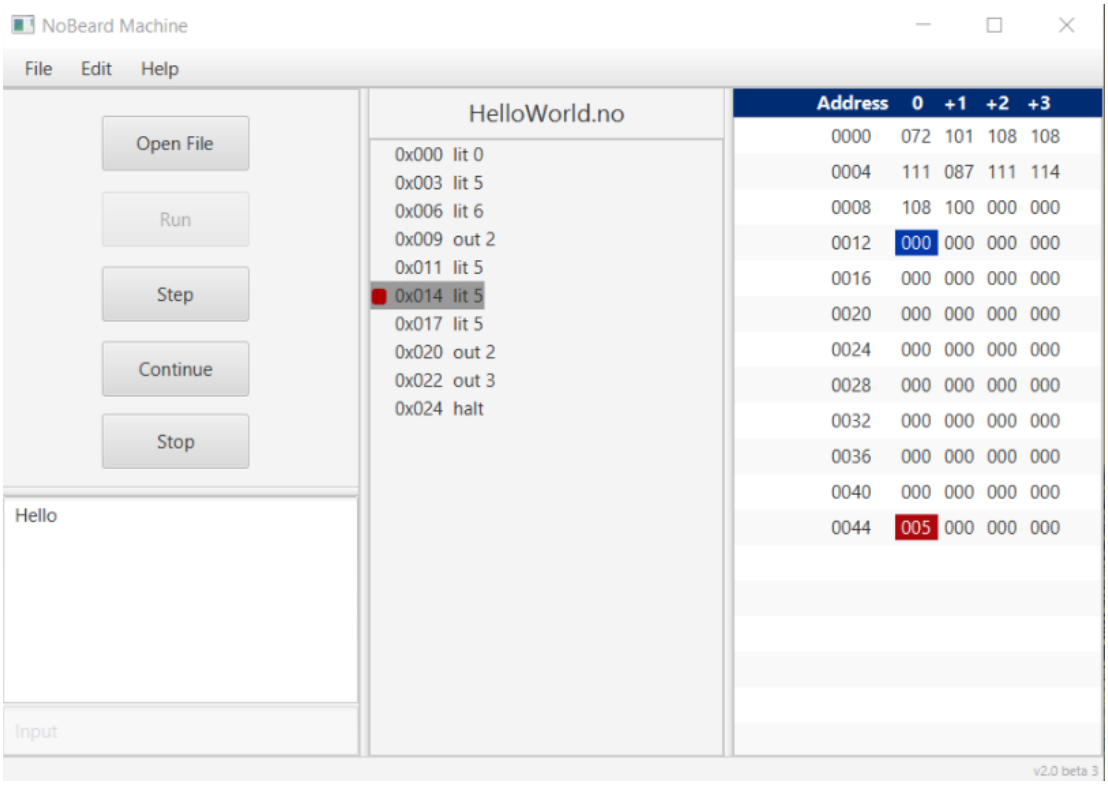

Figure 4.3: Debugging the "Hello World" program

thereafter:

- Converting a single byte to character or a whole line to four characters
- Converting data to integer
- Converting a line back to raw data.

4.5.1 Converting Data to Character

To view a character of a one-byte data as shown in figure 4.4, the following steps have to be done:

1. Select the line where the one-byte value is located
2. Right click on the value to be converted
3. Click on "View as Char"

Now the value at the given address is translated to an alphanumeric character. The conversion of an entire row of data to characters is done in the same way except that the user has to click "View characters" instead of "View as Char"

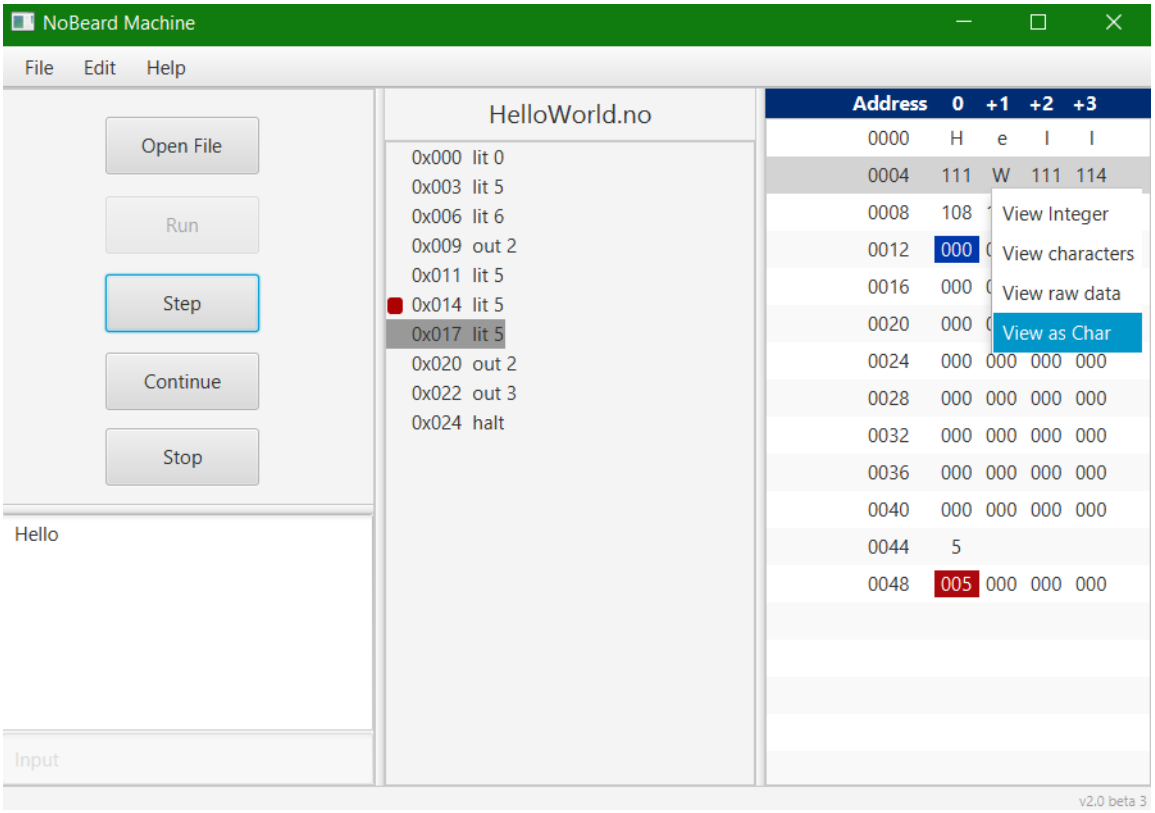

Figure 4.4: Converting data to a character

4.5.2 Converting Data to Integer

The translation of an integer takes four bytes that means, it takes four data cells from the desired start address and translates them to a single integer.

1. Select the data cell with the start address of the integer
2. Right click on the cell
3. Click on "View Integer"

After a successful conversion, the four data cells are being replaced by a single integer. Figure 4.5 shows an example of the translated integers. The first one is at address 54 which is not aligned in one row and the second one is on the same place as the stack pointer, precisely at the address 64.

4.5.3 Multiple Conversion

To facilitate the conversion from raw data to alphanumeric characters a multi selection function is available for the list of the data memory. To convert multiple data rows, the

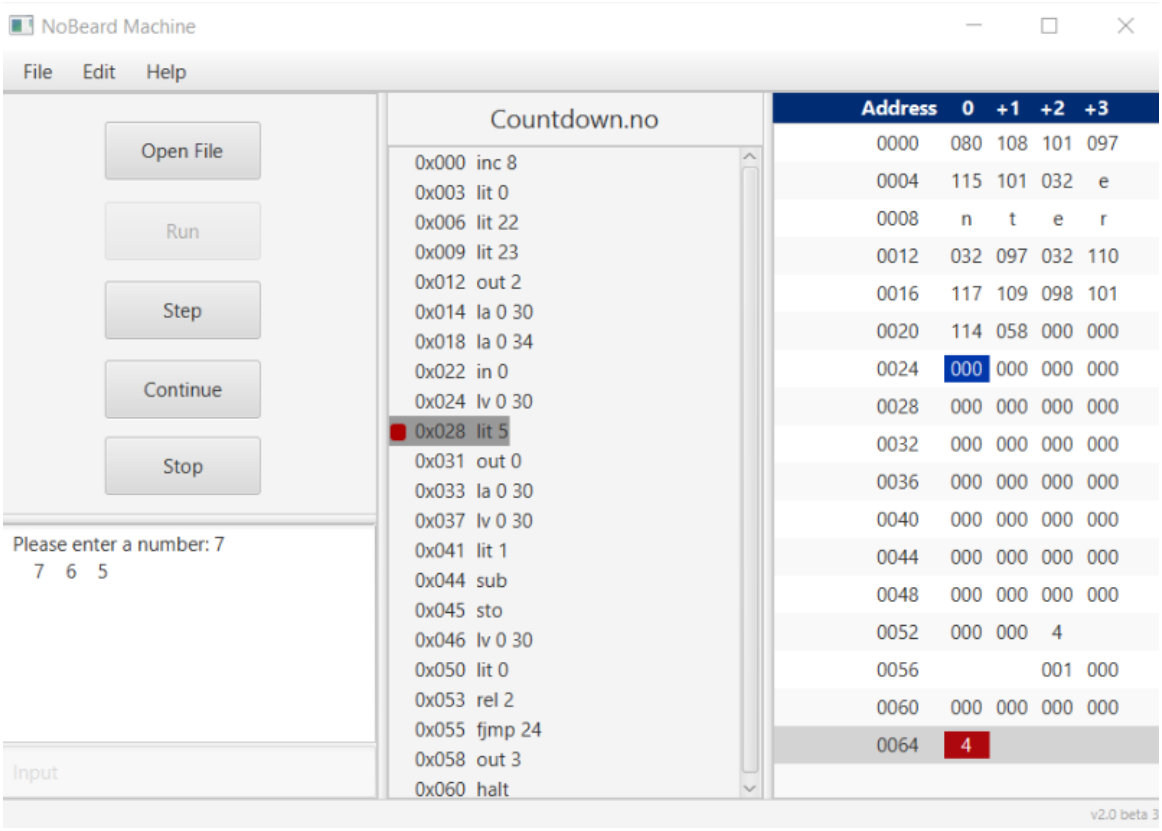

Figure 4.5: Converting four-byte data to a single integer

user has to hold the "Ctrl" key and select the specified rows with a left mouse click. Chosen lines will get highlighted with a light grey background color. As figure 4.6 shows, the selected rows with grey background are successfully concerted to some characters.

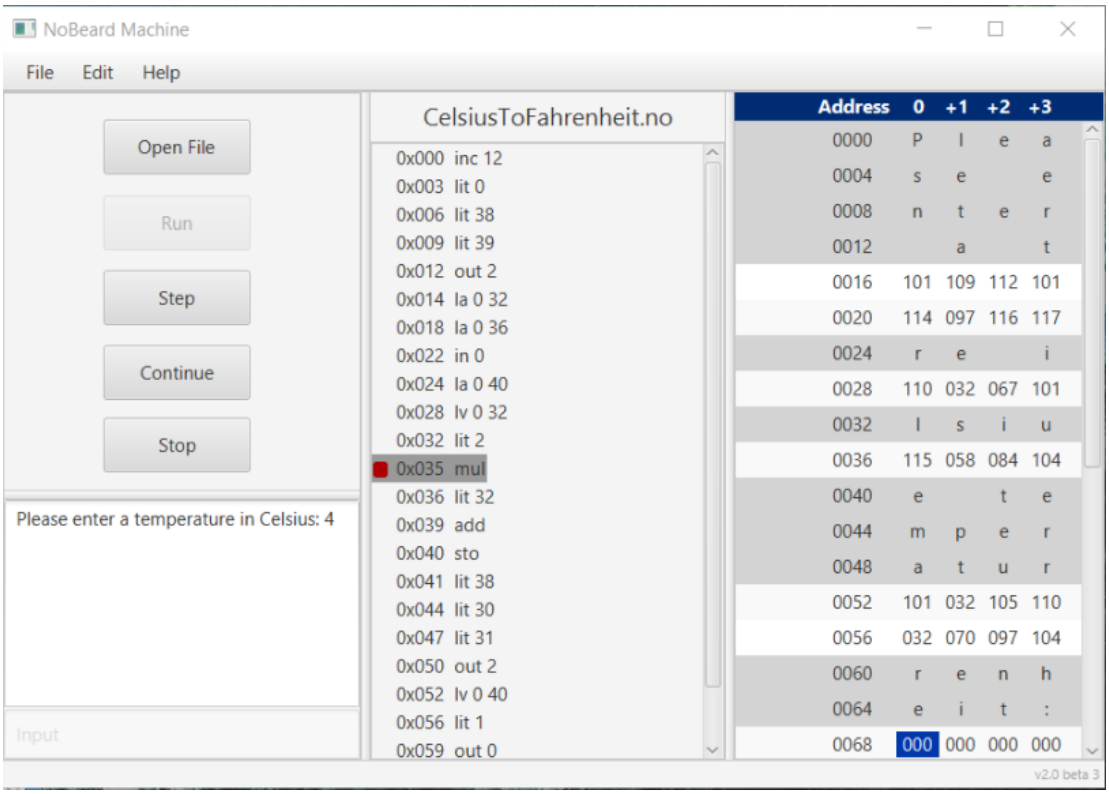

Figure 4.6: Multiple conversion of data

Chapter 5

Implementation

5.1 Used Technologies

5.1.1 IntelliJ IDEA

The implementation of the project is written in Java using the integrated development environment IntelliJ IDEA which is mainly for developing Java based software. Nevertheless, it supports other programming languages too like Scala, Groovy, Kotlin etc... It also support PHP, HTML, CSS3, JavaScript and TypeScript that could be very practicable for web development with the combination of different frameworks like AngularJs, Node.js, SQL Server etc... Another big advantage that IntelliJ IDEA has is the mobile development in Android or Cordova. This IDE support a huge variety of benefits such as:

- Different build systems (maven, gradle, ant, grunt, bower, etc ...)
- Version control systems (Git, Mercurial, Perforce, and SVN)
- Plugin ecosystem
- Test runner and coverage

More about this topic can be found on this site: [**?**]

5.1.2 Maven

This is a software management and build automation tool which is based on the concept of a project object model (POM). POM is fundamental unit of work in Maven. It is an XML file that resides in the base directory of the project as pom.xml. The POM contains information about the project and various configuration detail used by Maven to build the project(s). POM also contains the goals and plugins. While executing a task or goal, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information, and then executes the goal.

One of the biggest features in Maven is the dependency management. Maven automatically downloads the libraries and plug-ins declared in the POM file from the Maven central repository and stores them locally. In simple, when a developer builds a Maven project, all dependency files will be stored in a Maven local repository. So if Maven finds dependency libraries in the Maven local repository, it do not need to download them from the default online Maven central repository multiple time.

By default, the local Maven repository is the .m2 folder

- **Unix/Mac OS X:** `~/.m2`
- **Windows:** `C:\Users\{your-username}\.m2`

The Maven command `mvn install` builds a project and places its binaries in the local repository. Then other projects can utilize this project by specifying its coordinates in their POMs. Further information about Maven can be found on the official website: [**?**]

5.1.3 JavaFX

JavaFX is a framework which enables developers to design rich client applications that is able to run constantly on different platforms. It offers a wide range of APIs for web rendering, user interface styling and media streaming. JavaFX relies in particular on a scene graph, which manages the individual components of a GUI. It also provides a declarative description of XML-based graphical interfaces with FXML.

The newest JavaFX releases are fully integrated with the current Java SE Runtime Environment(JRE) and Java Development Kit(JDK), which is available for all main desktop platforms. More about this topic can be found on the following website: [**?**]

5.1.4 Scene Builder

JavaFX Scene Builder is a visual layout tool that generates FXML, an XML-based markup language that lets developers quickly design user interfaces, without any coding. Users just have to drag and drop UI components to the work area. By selecting components users can easily modify their properties or apply style sheets. The XML code for the layout is generated automatically in the background by the tool.

The resulting FXML file can be combined with a Java project by binding the UI to the application's logic. Every item of the layout view can be assigned with an fx:id to give the controller an easy access of components by the "@FXML" annotation. SceneBuilder is an external tool so it has to be downloaded from the official Oracle website and has to be integrated to a Java supporting IDE. More about this topic can be found on its website: [**?**]

5.2 Supporting NoBeard Packages

The NoBeard Machine is part of the existing NoBeard project. This project already consists of the following packages:

- **asm:** NoBeard Assembler to assemble .na files
- **compiler:** NoBeard Compiler to compile .nb source code files
- **config:** Hold information of the NoBeard project
- **error:** To handle errors that are occurring during compilation or assembly
- **io:** Responsible for reading source files and handling binary files
- **machine:** Implements a virtual machine with components like DataMemory, ProgramMemory, Instructions, etc...
- **parser:** Converts source code files to binary files **BAU:** This has to be done better after you have done the introductory chapters
- **scanner:** To analyze NoBeard source code files for keywords, operands or arguments **BAU:** see above
- **symbolTable:** Looks up for matching words based on the symbol table entry **BAU:** see above

However, the most important package for our work is `machine`. It defines the interface which the project described here is built upon.

5.3 Architecture of the User Interface

This is the model view controller pattern. The model is responsible for holding data. In our case this is not in a data base but the data is held in the virtual machine. The Model-View-Controller pattern does not require necessarily a database.

The GUI project is build up of three main components as shown in figure 5.1. The Controller and the Virtual Machine both of them are running on different threads and are accordingly synchronized. This will be described in detail in section 5.4. The relationship between the controller and view is for event actions and updating of view-components like data memory, output or program flow. The View allows users to call functions from the Controller by clicking on control buttons or by entering inputs.

5.3.1 The View

The view is basically a single `.fxml` file which holds the view components of the layout. As figure 5.2 shows it was created with an external tool called Scene Builder which is already described in section 5.1.4.

Each window is made up of an `AnchorPane` [**?**] and is separated by `SplitPanes`. An `AnchorPane` enables developers to build flexible user interfaces which can adapt automatically to different screen sizes and orientations. It provides the possibility to constrain the position and size of each element to its parent or siblings. So it becomes very comfortable and easy to size every window as the user wants. Very similar concepts are

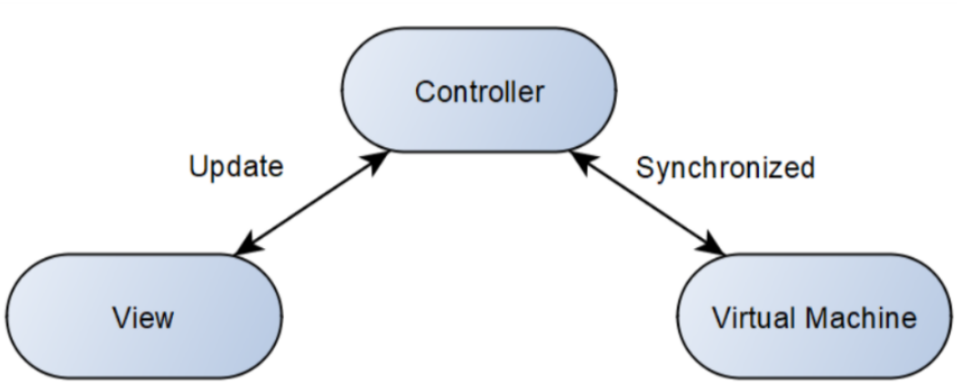

Figure 5.1: Main parts of the GUI implementation

available in other UI frameworks as Android (`LinearLayout` [?]) or iOS (`UIStackView` [?]).

Then each of these `AnchorPane`s holds the needed view-components like `Button`, `TextArea`, `TextField`, `ListView` or `ScrollPane`.

There are two types of designing and both of them is used in this project to develop a UI that meets all usability requirements.

Static Design

Static designing means that all view components are defined in the layout file and not in the Java code. With this type of designing it is pretty easy to build quickly a basic user interface. The developer has only to define the components in an fxml file or place them onto the view via drag and drop with the help of Scene Builder. This part of a project is mostly done by a designer because it does not require a lot of programming knowledge. Listing 5.1 shows a simplified version of how the control buttons were defined in the fxml file.

```
1 <AnchorPane>
2     <Button fx:id="openButton" onAction="#openFile" text="Open File"/>
3     <Button fx:id="startButton" onAction="#startProgram" text="Run"/>
4     <Button fx:id="stepButton" onAction="#step" text="Step"/>
5     <Button fx:id="continueButton" onAction="#continueToBreakpoint" text="Continue"/>
6     <Button fx:id="stopButton" onAction="#stopProgram" text="Stop"/>
7 </AnchorPane>
```

Listing 5.1: FXML example

Dynamic Design

In dynamic design the already existing view-components is being changed according to run time conditions or to user interactions. This means that the content of statically

placed controls changes e.g., by filling or updating a table view. Another typically dynamic designing is when the appearance of these items gets altered through enabling a button or changing its text. This is usually when a user or system triggers an event that matches a particular condition. In this case, the design code has to be mixed with the Java code and therefore programming skills from both sides are necessary.

Code snippet 5.2 shows the `fillProgramDataView` function which is called after a user opens a binary file. This function fills the program window with the content and additionally inserts a `CheckBox` to each line for maintaining the breakpoints.

```
1  private void fillProgramDataView(List<String> programDataList) {
2      VBox programData = new VBox();
3      for (String lineStr : programDataList)
4          addLineToProgramDataView(programData, lineStr);
5      programDataView.setContent(programData);
6  }
7
8  private void addLineToProgramDataView(VBox programData, String lineContent) {
9      CheckBox line = new CheckBox(lineContent);
10     line.setPadding(new Insets(1));
11     line.setOnAction((event) -> setClickEventToLine((CheckBox) event.getSource()));
12     programData.getChildren().add(line);
13     programDataMap.put(getAddressOfProgramLine(line.getText()), line);
14 }
15
16 private void setClickEventToLine(CheckBox breakpoint) {
17     if (breakpoint.isSelected())
18         machine.addBreakpoint(getAddressOfProgramLine(breakpoint.getText()));
19     else
20         machine.removeBreakpoint(getAddressOfProgramLine(breakpoint.getText()));
21 }
```

Listing 5.2: Implementation of program data view

5.3.2 The Controller

The controller is primarily responsible for the coordination and operation of the view and its components. As already described in section 5.1.4 the controller gets access to the view components via the binding of an fx id. To enable this access the view and the controller have to be connected by setting the `fx:controller` attribute of the root view element to the currently used controller. The user interface defined by an FXML file and its controller is loaded into memory by an `FXMLLoader`. When the `load()` method is called on the `FXMLLoader`, it:

1. Loads the FXML file
2. Creates an instance of the controller class specified by the `fx:controller` attribute, by calling its no-argument constructor

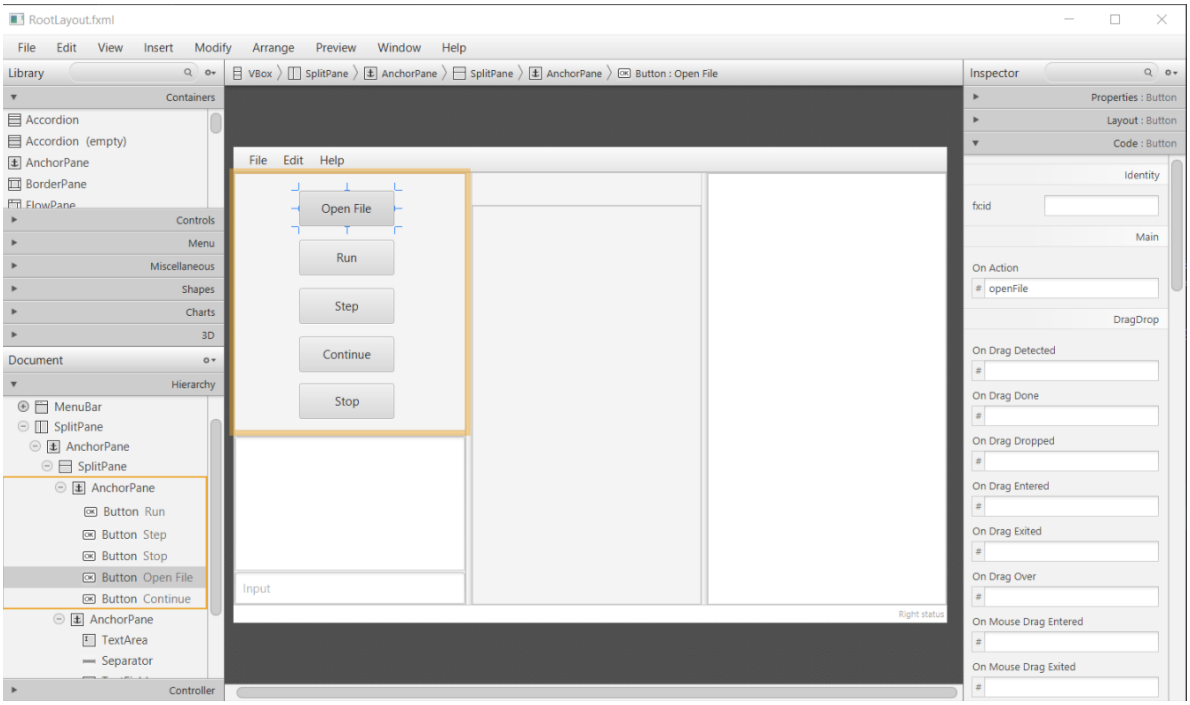

Figure 5.2: Development of the view with Scene Builder

3. Sets the value of any @FXML-annotated fields in the controller to the elements defined with matching fx:id attributes

4. Registers any event handlers mapping to methods in the controller

5. Calls the `initialize()` method on the controller, if there is one.

Through this advantage event actions like mouse clicks or key presses can be bound between the controller and the view. This could be achieved by setting a name for the "On Action" property in Scene Builder and then creating the related function in the controller. The function has to be declared with an `@FXML` annotation and the same name as it was set on the action property of the item.

These action methods accept a single argument of type `ActionEvent`. This can be used to get some further information of the fired event like which mouse button was pressed.

Those events have a defined order. The constructor is called before the `@FXML`-annotated fields are injected, but the `initialize()` method is called after. This means we can access (and configure) `@FXML`-annotated fields in the `initialize()` method, but not in the constructor. It is quite common (at least in simple applications) not to define any constructor in the controller classes.

The other task the controller has to handle is to update the model, i.e., in our case to change the state of the NoBeard machine according to the user commands. So it acts like a bridge to the machine and calls functions like `runProgram(int startPc)`, `step()`, `getCallStack()` etc...

To achieve this connection between the two components, the controller gets an instance of the NoBeard machine which is being target for NoBeard assembler programs. This happens directly in the `initialize()` method. As said before this method is called automatically and as first when the FX application starts. So every stuff that is needed for the initialization e.g., enabling or disabling controls on the view, is done in this `initialize()` method.

Since the initialization, all other methods can only be invoked by triggering an event like clicking on the "openFile" button. However, these event methods gets called under certain conditions that depend on the user. For example, the user has to stop a running program before opening another NoBeard object file. This conclusion gives the controller a defined cycle that the user can execute:

1. Open file
2. Disassemble file
3. Update view
4. Start program

The opening of a NoBeard object file is operated with the help of a BinaryFileHandler. After an object file is successfully read and program data is available as a binary byte stream it has to be disassembled i.e., that it has to be converted from its binary form to a human readable assembler form.

After this translation is finished the machine has to load the string constants and the program data from the object file into the corresponding memory. Program data is filled in a VBox where each line consists of a CheckBox and an Assembler statement. All of the CheckBoxes get an OnAction event which toggles breakpoints of the machine. This is already demonstrated with the listing 5.2.

When a program is started, a new external thread must be started in which the machine is running separately from the UI. The code 5.3 illustrates how the machine gets started on a background thread using a lambda expression.

```
1  @FXML
2  void startProgram(ActionEvent event) {
3      prepareGuiForProgramStart();
4      new Thread(() -> {
5          machine.runProgram(0);
6          Platform.runLater(() -> {
7              highlightNextInstructionToBeExecuted();
8              DataMemoryView.update(this, getRawDataMemoryList());
9      }).start();
10 }
```

Listing 5.3: Starting the machine on a new thread

The reason why and how they run on different threads will be explained in section 5.4. The machine executes step by step every instruction of the program until any interruption. It can be interrupted by a breakpoint, input request or by a `halt` instruction.

`Platform.runLater(java.lang.Runnable runnable)` is usually used for updating the GUI from a background thread. Java supports `Task` which is a similar concept and can be utilized for the same purpose.

The method `highlightNextInstructionToBeExecuted()` at line 7 signs the instruction which hast to be executed as next, e.g., if the user press the "Step" button then the highlighted assembler statement will be executed. In other words, it shows where the current program counter is.

The final statement in this code snippet is at line 8 which updates the data memory on the view after an interruption, but this will be described in section 5.6.

5.4 Synchronization of NoBeard Machine and GUI

As already mentioned, the NoBeard virtual machine has to run on a background thread. The main reason for this is that there are two different processes using common processing resources. For instance, if they run on the same thread then the user would not be able to make any interactions because the virtual machine locks the main thread. But, even when the virtual machine gets an extra thread there would be still a deadlock because the background thread has to synchronize with the UI thread. So e.g., every time the user has to operate an input, a switch to the UI thread is needed. Otherwise it would cause a critical section between the threads.

The synchronization of these two threads is implemented with the semaphore construction. A semaphore has a pretty simple usage to solve critical section problems and to achieve process synchronization in the multi processing environment. It is a variable that acts like a traffic light with its `acquire()` and `release()` functions.

The NoBeard Machine contains two interfaces to optimize outputs and inputs on the used device. As soon as the machine executes an input instruction `in`, it calls firstly a function from the input interface either `hasNextInt()` or `hasNext()`. Both do the same thing, checks whether there is an input by the user or not. The only different is that the one of them also checks if the string is numeric. These functions are overridden in the `FxInputDevice` class where also an instance of the controller is loaded by the constructor. So, by calling one of these `hasNext` function the machine thread should be paused as shown in the code snippet 5.4. To avoid the deadlock, the semaphore from the controller is acquired at this position i.e., at line 11.

```
1 @Override
2 public boolean hasNextInt() {
3     waitForInput();
4     return controller.getInput().chars().allMatch( Character::isDigit );
```

```
5 }
6
7 private void waitForInput() {
8     try {
9         Platform.runLater(() -> controller.enableInputView(true));
10        controller.getSemaphore().acquire();
11    } catch (InterruptedException e) {
12        e.printStackTrace();
13    }
14 }
```

Listing 5.4: Synchronisation with semaphore (Part 1)

Now the user can make an input on the UI thread and submit it. To get back to the machine thread, the semaphore hast to be released after the user fires the submit event by pressing the ENTER key. This is demonstrated with the code fragment 5.5 more precisely at line 12. Then the machine continues at the same position where the semaphore was acquired, at one of the has Next function and can analyse the provided input string.

```
1  private void makeInputViewReactOnReturn() {
2      inputView.setOnKeyPressed(event -> {
3          if (event.getCode() == KeyCode.ENTER && inputView != null &&
4                  !inputView.getText().isEmpty())
5              inputIsAvailable(inputView.getText());
6      });
7  }
8
9  private void inputIsAvailable(String providedInput) {
10     getOutputView().appendText(providedInput + "\n");
11     input = providedInput;
12     inputView.clear();
13     enableInputView(false);
14     getSemaphore().release();
15 }
```

Listing 5.5: Synchronisation with semaphore (Part 2)

5.5 Handling of Breakpoints

Machine internal handling of breakpoints is done as described in [**?**]. The implementation for the maintaining of breakpoints is coded with a simple observer pattern design to keep the best performance of the virtual machine.

The idea of this solution is pretty easy to understand. The machine runs only as long as it is in a `running` state. So, a new instruction is introduced, called `break` which sets the machine into a `blocked` state. This means at the same time that as soon as a user sets a breakpoint on a specified instruction, a change from the original to a `break` instruction is necessary.

The `ControlUnit` which is responsible for execution cycles in the virtual machine is going to be an observable class. Then a new Observer class is implemented which is called `BreakpointsHandler` that is holding all breakpoints in a `HashMap`.

```
private HashMap<Integer, Byte> breakpoints;
```

This `HashMap` stores the address and the instruction of a selected breakpoint. The `BreakpointsHandler` class contains in all four major functions as shown in the code snippet 5.6. Adding and removing breakpoints to or from the `HashMap` and replacing an instruction at a specified address from the program memory to a new one. The selection of a breakpoint on the UI calls the set or remove function from the debugger, as the case may be. As soon as a breakpoint is selected, it will be stored to the `HashMap` with its original instruction and at same time replaces the original instruction by the newly added `break` instruction.

```
1  void setBreakpoint(int atAddress) {
2      byte originalInstruction = programMemory.loadByte(atAddress);
3      if (originalInstruction != -1) {
4          breakpoints.put(atAddress, programMemory.loadByte(atAddress));
5          programMemory.replaceInstruction(atAddress, InstructionSet.Instruction.BREAK.getId());
6      }
7  }
8
9  void removeBreakpoint(int atAddress) {
10     if (breakpoints.containsKey(atAddress)) {
11         programMemory.replaceInstruction(atAddress, breakpoints.get(atAddress));
12         breakpoints.remove(atAddress);
13     } else {
14         // force program address error
15         programMemory.loadByte(-1);
16     }
17 }
18
19 void onStopAtBreakpoint(int atAddress) {
20     if (breakpoints.containsKey(atAddress)) {
21         programMemory.replaceInstruction(atAddress, breakpoints.get(atAddress));
22         programCurrentlyStoppedAtAddress = atAddress;
23     }
24 }
25
26 void onContinueFromBreakpoint() {
27     programMemory.replaceInstruction(programCurrentlyStoppedAtAddress,
28             InstructionSet.Instruction.BREAK.getId());
29 }
```

Listing 5.6: Implementation of the debugger

This `break` instruction sets the machine to a `blocked` state and notify the observer to change the `break` instruction back to the original one by calling the `onStopAtBreakpoint(int atAddress)` method.

Now the machine is blocked at a specified breakpoint and the user can take a look to the current memory state and go one step further in the program flow or continue the execution cycle to a next breakpoint. However, when the user wants to step further to execute the current instruction where the breakpoint is, of course again a switch back to the break instruction is needed after the original instruction completed. This is will be done by calling the `onContinueFromBreakpoint()` method.

5.6 Visualisation of DataMemory

The visualisation of the data memory gives an overview about string constants and stack frames. As figure 5.3 shows the view lists four byte of data on each row starting with the belonging address.

Logically, the list has to be updated after any interruption by a breakpoint. This is demonstrated with the code snippet 5.3, more precisely at line 7.

Address	0	+1	+2	+3
0000	072	101	108	108
0004	111	087	111	114
0008	108	100	101	114
0012	032	097	032	110
0016	117	109	098	101
0020	114	058	000	000
0024	000	000	000	000
0028	000	000	000	000
0032	000	000	000	000
0036	000	000	000	000
0040	000	000	000	000
0044	005	000	000	000
0048	005	000	000	000

Figure 5.3: Visualisation of data memory

For the visualisation of these data a `ListView` is defined on the view which is one of the best choice for this purpose. The updating of data memory view has defined a cycle. Firstly a function iterates through the memory until the current stack pointer and groups all data into four bytes and finally stores them into an observable collection of strings. Then this collection is assigned to `ListView` items that are visible on the view. Each row of the `ListView` contains now a line of string filled with the address and four pure data.

Since each item in a `ListView` is represented by an instance of the `ListCell` class, a row can be customized to get a nice structure like in figure 5.3. This customisation

of a cell can be achieved with the `cellFactory` API. The cell factory is called by the platform whenever it determines that a new cell needs to be created. To specialize the `Cell` used for the `ListView`, an implementation of the `cellFactory` callback function has to be provided.

```
static void update(Controller controller, ObservableList<String> content) {
    controller.getDataMemoryListView().setItems(content);
    controller.getDataMemoryListView().setCellFactory(list -> new ListCell<String>() {
        int framePointer = controller.getMachine().getCallStack().getFramePointer();
        int stackPointer = controller.getMachine().getCallStack().getStackPointer();
        static final int INDEX_OF_ADDRESS = 0;

        @Override
        protected void updateItem(String item, boolean empty) {
            super.updateItem(item, empty);
            if (item != null) {
                createDataLine(item);
            }
        }

        private void createDataLine(String item) {
            HBox line = new HBox();
            int firstAddressInLine = getIndex() * 4;
            convertStringToLabels(firstAddressInLine, item).forEach(line.getChildren()::add);
            setContextMenuToDataCells(line);
            setGraphic(line);
        }

        private List<Label> convertStringToLabels(int firstAddressInLine, String line) {
            String[] lineContent = splitDataLine(line);
            List<Label> result = createLabelForAddress(lineContent[INDEX_OF_ADDRESS]);
            int currentAddress = firstAddressInLine;
            for (int i = 1; i < lineContent.length; i++) {
                if (currentAddress == framePointer)
                    result.add(createHighlightedLabel(lineContent[i], "#0038AC"));
                else if (currentAddress == stackPointer)
                    result.add(createHighlightedLabel(lineContent[i], "#AC080E"));
                else
                    result.add(createNormalLabel(lineContent[i]));
                currentAddress++;
            }
            return result;
        }
        ...
```

Listing 5.7: Implementation of the data memory view using cell factory

In the code example 5.7 an anonymous inner class is created using lambda expression, that simply returns instances of `ListCell` overriding the `updateItem` method. This method is called whenever the item in the cell changes, for example when the user scrolls

the ListView or the NoBeard machine updates the data memory (and the cell is reused to represent some different item in the ListView). Because of this, there is no need to manage bindings - simply reacting to the change in items when this method occurs. In this example, whenever the item changes, we update the cell text property, and also modify the text fill to ensure that we get the correct visuals. So each row of the list view will be separated in five labels, one for the address and four with one byte data. Labels with a given id can be easily styled in a extra CSS file by setting paddings or backgroundcolor. The data labels get a click event to open up a context menu where the following four functions are available:

- Convert data to a single character
- Convert a line of data to four characters
- Convert a four byte of data to a single integer
- Convert a translated line of data back to raw data

A context menu is a pop-up window that appears in response to a mouse click. The context menu can contain one or more menu items. It is quite similar to a Menu and consists of items with types of MenuItem, CheckMenuItem, RadioMenuItem or SeparatorMenuItem. In our case, the context menu holds only items with the default type which is MenuItem. Each MenuItem gets an OnAction event where the conversion method is being called. So, when the user clicks on one of these MenuItem the translation will be started. Listing 5.8 shows how to add a ContextMenu to our ListView and display it when a user clicks with the right mouse button on an item from the ListView.

```
1 controller.getDataMemoryListView().setContextMenu(contextMenu);
2 controller.getDataMemoryListView().setOnContextMenuRequested(event -> {
3     contextMenu.show(controller.getDataMemoryListView(),
4      event.getScreenX(), event.getScreenY());
5     event.consume();
6 });
```

Listing 5.8: Context menu

With this solution we can only convert entire lines of the DataMemoryView, since only access to a selected index of a ListView is possible. For example, if a user wants to translate a cell into a single char or to view an integer, then the desired address is needed for the conversion. When translating a whole line, we get the address because every line begins with the starting address but when a user click on a random cell, we get only the address of the first byte from the row.

To solve this problem each Label (cell) has to react when a right mouse button is being clicked. With the setOnMouseClicked method an event can be assigned to a Label where the needed extra MenuItems will be added to the ContextMenu of the ListView. Each MenuItem get again an OnAction event where the translation method will be called but now the needed address or data can passed as an argument.

For the implementation of these conversation functions a separated `DataMemoryConverter` class is introduced. The conversion of a single byte to character is handled by translating the ASCII value which is the current byte to a char.

For integers it is a bit more complex because a single integer has to take four byte of data. If all the four bytes are not aligned in one row, the rest bytes has to be occupied from the next row. Then the first cell on the desired address is being changed to the translated integer and the next three cell has to be replaced by empty labels.

List of Figures

2.1 Example of a syntax diagram 10

3.1 Components of the NoBeardMachine Architecture 12
3.2 Data Memory of the NoBeard Machine 13
3.3 Snapshot of a call stack with three frames 14
3.4 NoBeard Binary File Format 16

4.1 Components of the NoBeardMachine 22
4.2 Executed "Hello Worldprogram 24
4.3 Debugging the "Hello Worldprogram 25
4.4 Converting data to a character 26
4.5 Converting four-byte data to a single integer 27
4.6 Multiple conversion of data 28

5.1 Main parts of the GUI implementation 32
5.2 Development of the view with Scene Builder 34
5.3 Visualisation of data memory 39

Listings

5.1 FXML example 32
5.2 Implementation of program data view 33
5.3 Starting the machine on a new thread 35
5.4 Synchronisation with semaphore (Part 1) 36
5.5 Synchronisation with semaphore (Part 2) 37
5.6 Implementation of the debugger 38
5.7 Implementation of the data memory view using cell factory 40
5.8 Context menu 41

Project Log Book

Date	Participants	Todos	Due

Appendix A

Additional Information

If needed the appendix is the place where additional information concerning your thesis goes. Examples could be:

- Source Code
- Test Protocols
- Project Proposal
- Project Plan
- Individual Goals
- ...

Again this has to be aligned with the supervisor.

Appendix B

Individual Goals

This is just another example to show what content could go into the appendix.