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

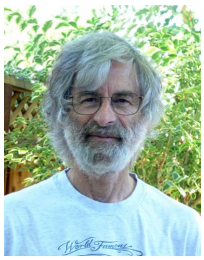
Here it is described what the thesis is all about. The abstract shall be brief and concise and its size shall not go beyond one page. Furthermore it has no chapters, sections etc. Paragraphs can be used to structure the abstract. If necessary one can also use bullet point lists but care must be taken that also in this part of the text full sentences and a clearly readable structure are required.

Concerning the content the following points shall be covered.

- *Definition of the project:* What do we currently know about the topic or on which results can the work be based? What is the goal of the project? Who can use the results of the project?
- *Implementation:* What are the tools and methods used to implement the project?
- *Results:* What is the final result of the project?

This list does not mean that the abstract must strictly follow this structure. Rather it should be understood in that way that these points shall be described such that the reader is animated to dig further into the thesis.

Finally it is required to add a representative image which describes your project best. The image here shows Leslie Lamport the inventor of LaTeX.



Zusammenfassung

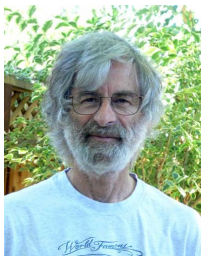
An dieser Stelle wird beschrieben, worum es in der Diplomarbeit geht. Die Zusammenfassung soll kurz und prägnant sein und den Umfang einer Seite nicht übersteigen. Weiters ist zu beachten, dass hier keine Kapitel oder Abschnitte zur Strukturierung verwendet werden. Die Verwendung von Absätzen ist zulässig. Wenn notwendig, können auch Aufzählungslisten verwendet werden. Dabei ist aber zu beachten, dass auch in der Zusammenfassung vollständige Sätze gefordert sind.

Bezüglich des Inhalts sollen folgende Punkte in der Zusammenfassung vorkommen:

- *Aufgabenstellung:* Von welchem Wissenstand kann man im Umfeld der Aufgabenstellung ausgehen? Was ist das Ziel des Projekts? Wer kann die Ergebnisse der Arbeit benutzen?
- *Umsetzung:* Welche fachtheoretischen oder -praktischen Methoden wurden bei der Umsetzung verwendet?
- *Ergebnisse:* Was ist das endgültige Ergebnis der Arbeit?

Diese Liste soll als Sammlung von inhaltlichen Punkten für die Zusammenfassung verstanden werden. Die konkrete Gliederung und Reihung der Punkte ist den Autoren überlassen. Zu beachten ist, dass der/die LeserIn beim Lesen dieses Teils Lust bekommt, diese Arbeit weiter zu lesen.

Abschließend soll die Zusammenfassung noch ein Foto zeigen, das das beschriebene Projekt am besten repräsentiert. Das folgende Bild zeigt Leslie Lamport, den Erfinder von LaTeX.



Acknowledgments

If you feel like saying thanks to your grandma and/or other relatives.

Contents

Chapter 1

Introduction

1.1 Initial Situation

Common word processors do not prepare print-like documents in so far as these programs do not reflect the rules of professional printing which have been grown over centuries. These rules contain clear requirements for balancing page layouts, the amount of white space on pages, font-handling, etc. Donald Knuth's TeX package (see citation removed to get rid of BibTeX warning) is a word processor which conforms to these printing rules. This package was enhanced by Leslie Lamport by providing more text structuring commands. He called his package LaTeX citation removed to get rid of BibTeX warning.

When preparing a thesis, we want not only to have our content on a top level, we also want to commit to a high level of formal criteria. Therefore, we request our students to use one of these professional printing production environments like TeX or LaTeX.

Furthermore students should train their scientific writing skills. This includes a clear and structured break-down of their ideas, a high-level and clear wording, and the training of transparent citations of ideas from other sources than from theirs. A good source for more information concerning technical and scientific writing can be found in citation removed to get rid of BibTeX warning.

1.2 Goals

The general goals and objectives of the project are described here. Care must be taken that the goals documented here are purely project goals and have nothing to do with individual goals of the team members. If individual goals should be part of the thesis they are listed in appendix B.

1.3 Overview

Details of the diploma thesis have to be aligned between student and supervisor. This should be a basic structure to facilitate the first steps when students start to write their

Figure 1.1: Don Knuth, the inventor of TEX

theses.

Never forget to add some illustrative images. Images must not be messed up with your normal text. They are encapsulated in floating bodies and referenced in your text. An example can be seen in figure 1.1. As you can see, figures are placed by default on top of the page nearby the place where they are referenced the first time. Furthermore you can see that a list of figures is maintained automatically which can be included easily by typing the command `\listoffigures` into your document.

1.4 Basic Terminology

As usual the very basic terminology is briefly explained here. Most probably the explanations here only scratch a surface level. More detailed explanations of terminology goes into chapter **??**.

1.5 Related Work and Projects

Here a survey of other work in and around the area of the thesis is given. The reader shall see that the authors of the thesis know their field well and understand the developments there. Furthermore here is a good place to show what relevance the thesis in its field has.

1.6 Structure of the Thesis

Finally the reader is given a brief description what (s)he can expect in the thesis. Each chapter is introduced with a paragraph roughly describing its content.

Chapter 2

Formal languages

Chapter 3

Compiler construction

Chapter 4

The NoBeard Machine Architecture

4.1 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 4.1 shows these components and how they are related. The components will be described in the following sections.

4.1.1 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`.

4.1.2 Data Memory

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

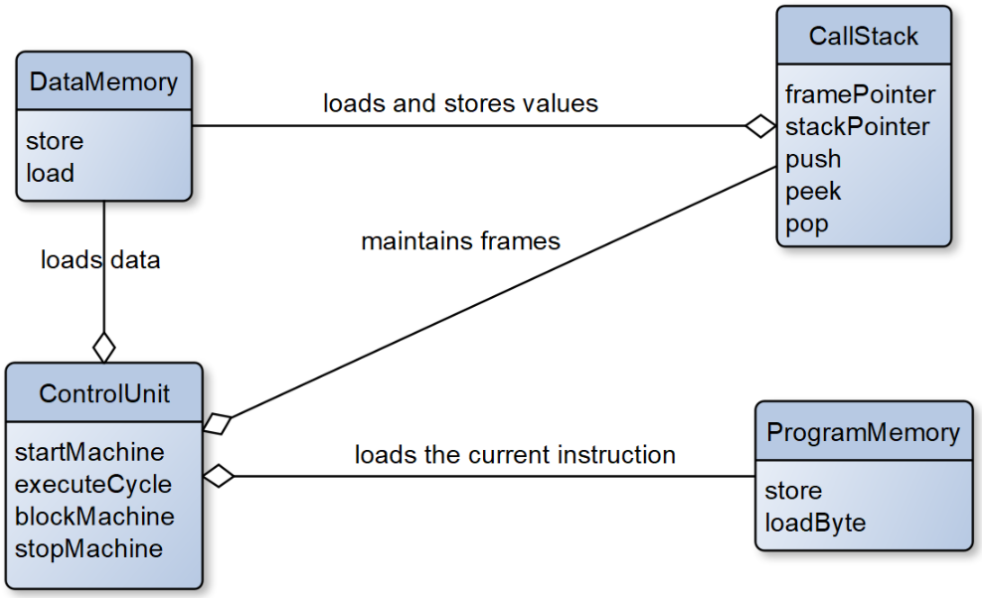


Figure 4.1: Components of the NoBeardMachine Architecture

- **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 [wik18]).
- **Booleans** are four-bytes values and are stored as the integer 0 for false and the integer 1 for true.

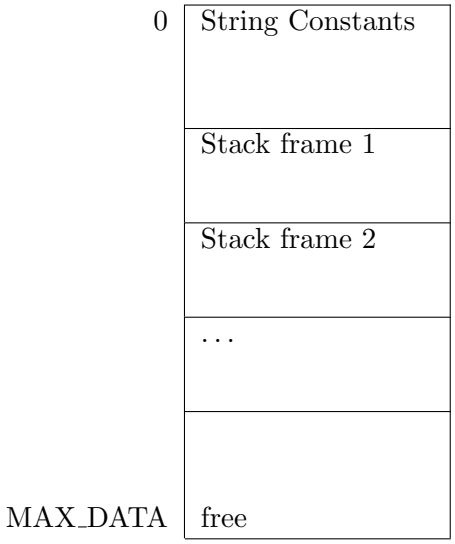


Figure 4.2: Data Memory of the NoBeard Machine

As figure 4.2 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 4.3: Snapshot of a call stack with three frames

Figure 4.3 shows a pretty good example from [Bau17]. 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.

4.1.3 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.

4.1.4 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.

4.2 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 4.4 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.

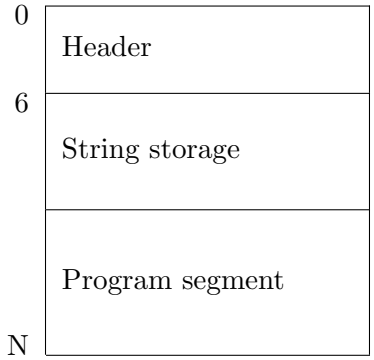


Figure 4.4: NoBeard Binary File Format

4.3 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.

4.3.1 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.

4.3.2 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.

4.3.3 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.

4.3.4 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

4.4 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 4.3.

Chapter 5

User Manual

5.1 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.

5.2 The UI Components

As shown in figure 5.1 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.

5.2.1 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.

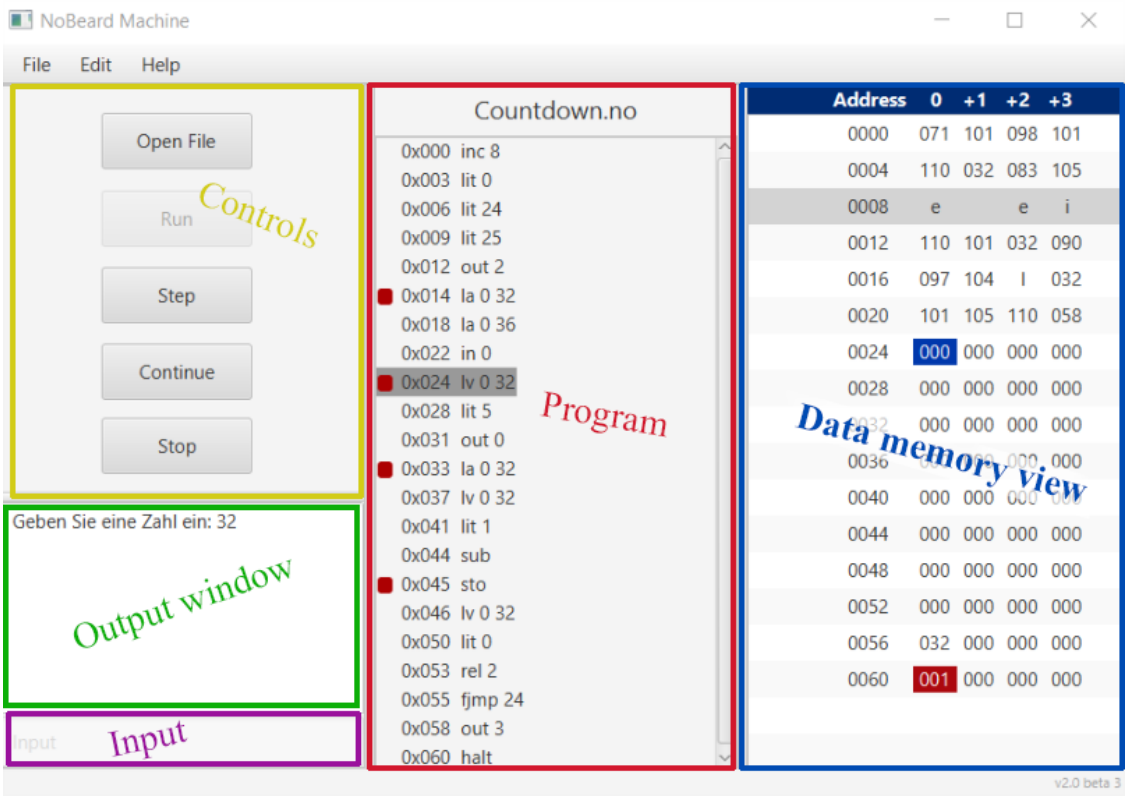


Figure 5.1: Components of the NoBeardMachine

- **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.

5.2.2 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.

5.2.3 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.

5.2.4 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.

5.2.5 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.

5.3 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.

5.4 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"

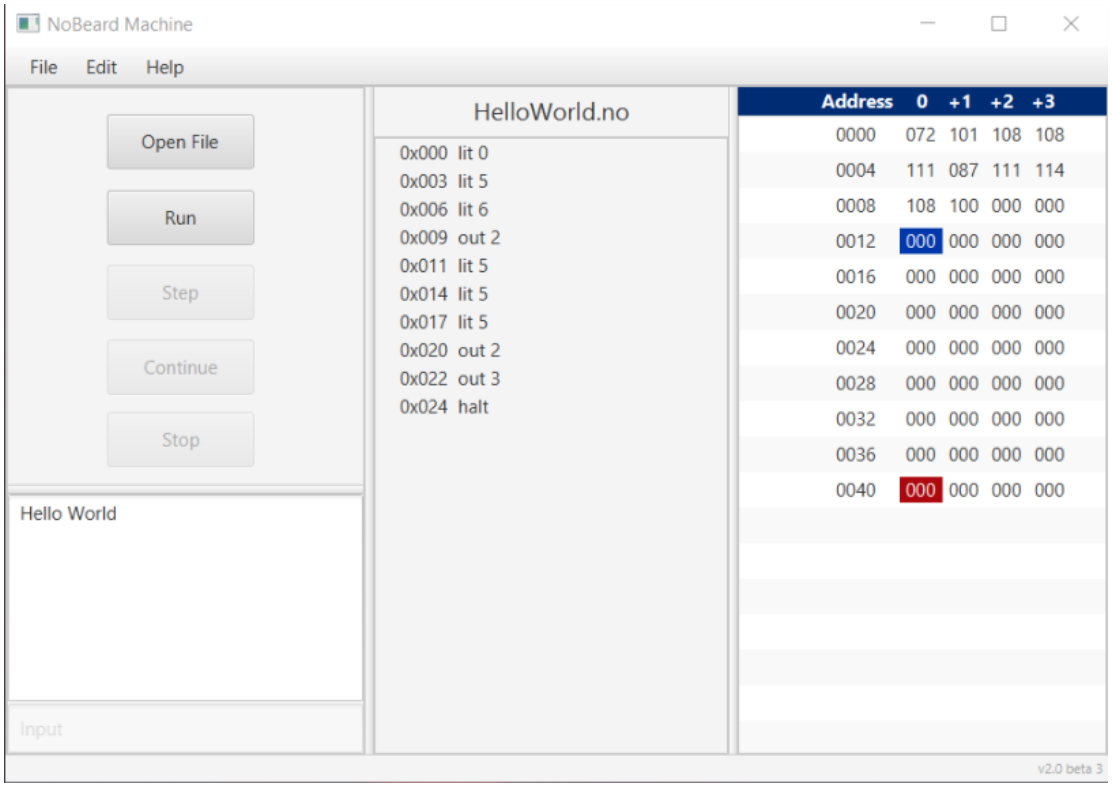


Figure 5.2: Executed "Hello World" program

button as shown in figure 5.3. 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.

5.5 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

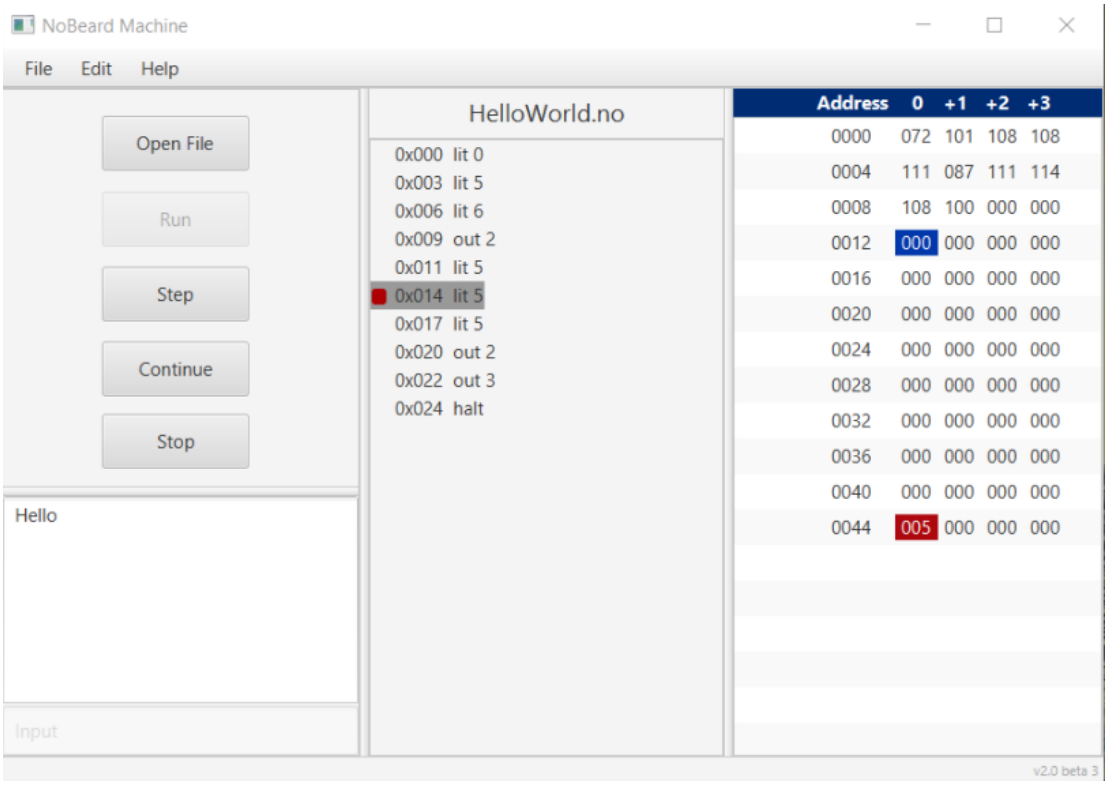


Figure 5.3: Debugging the "Hello World" program

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.

5.5.1 Converting Data to Character

To view a character of a one-byte data as shown in figure 5.4, 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"

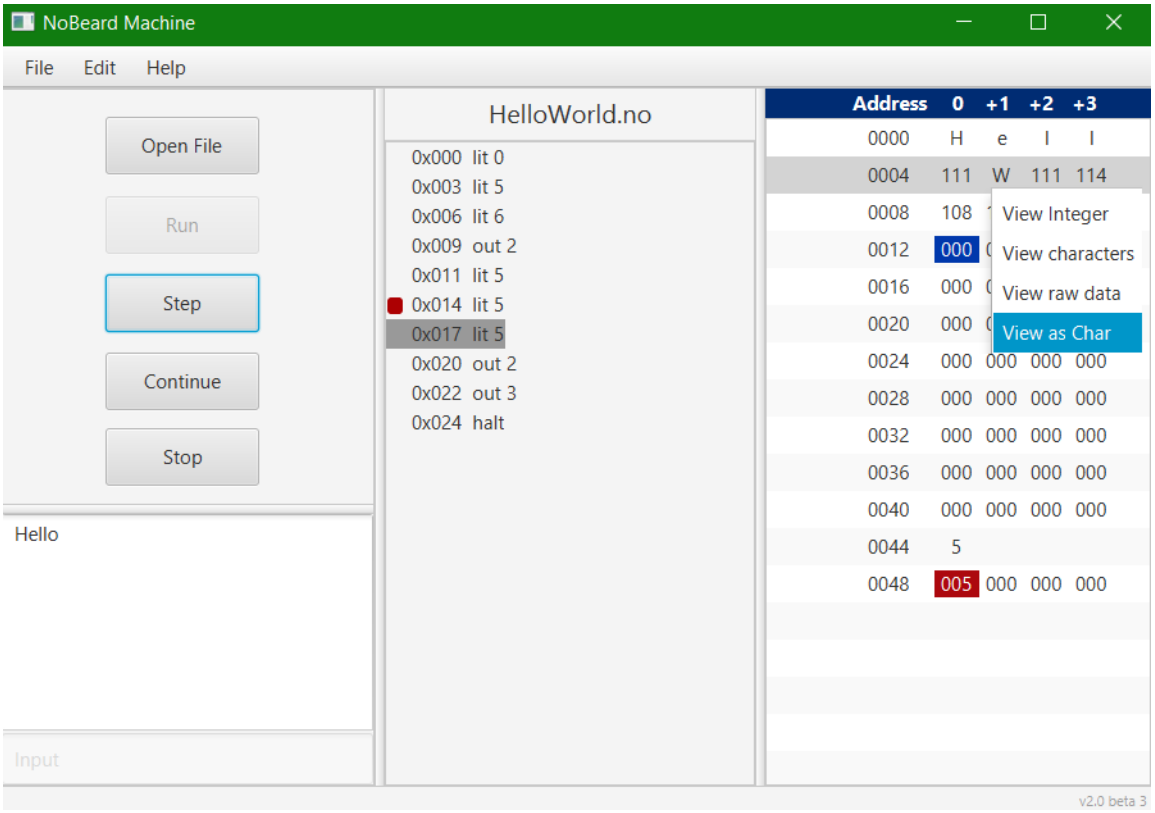


Figure 5.4: Converting data to a character

5.5.2 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 5.5 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.

5.5.3 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

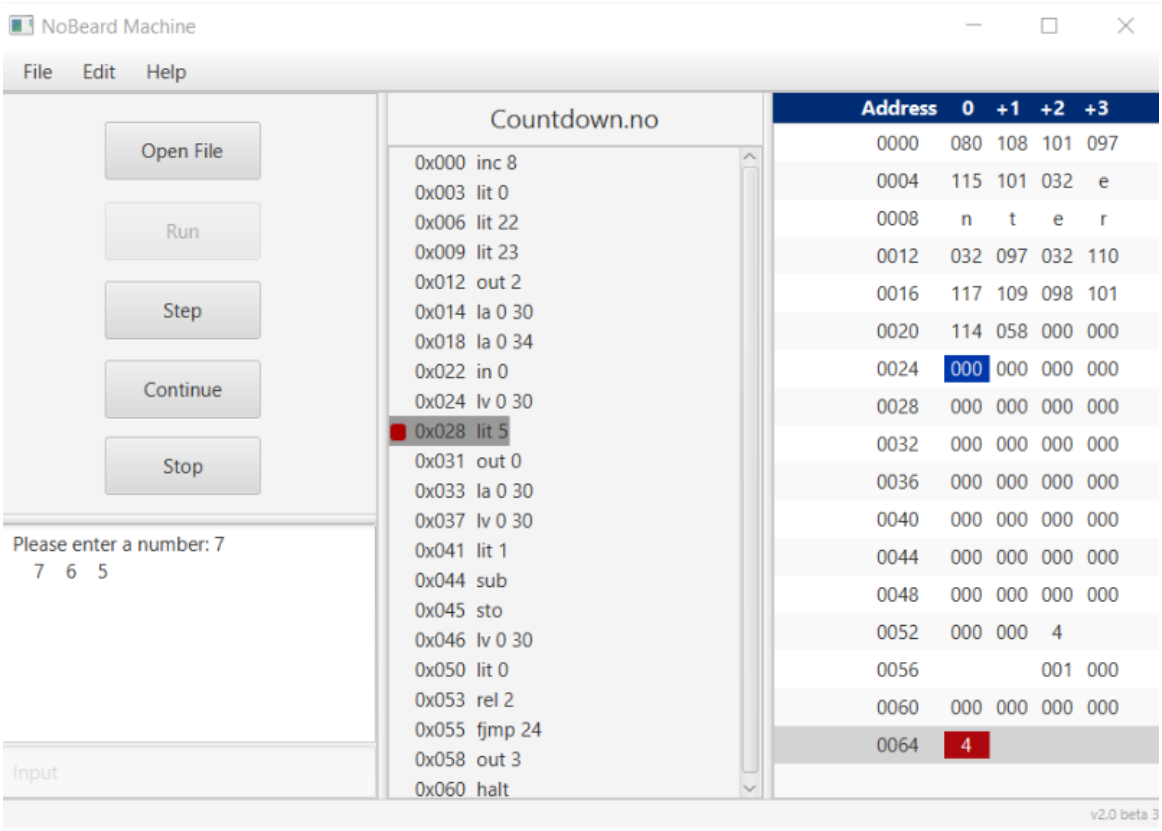


Figure 5.5: Converting four-byte data to a single integer

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 5.6 shows, the selected rows with grey background are successfully concerted to some characters.

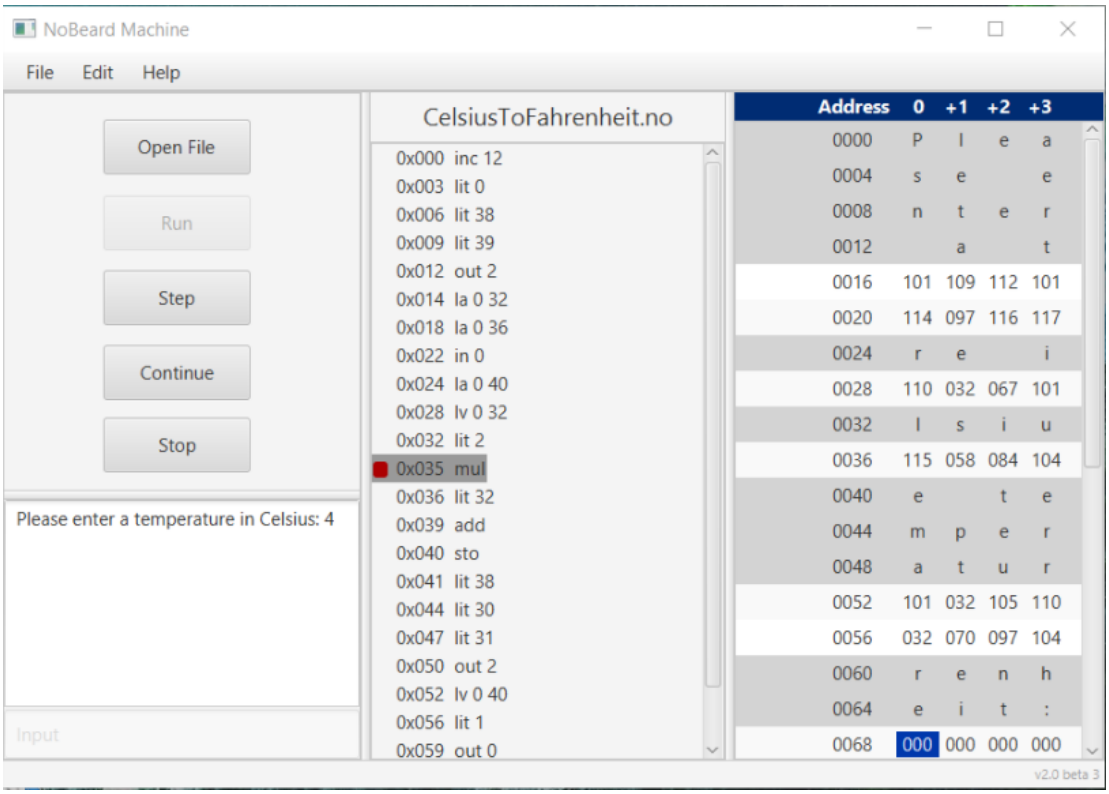


Figure 5.6: Multiple conversion of data

Chapter 6

Implementation

6.1 Used Technologies

6.1.1 IntelliJ IDEA

The Implementation of the project is written in Java on the integrated development environment IntelliJ which is mainly for developing Java based software. Nevertheless, it supports other programming languages too like Scala, Groovy and Kotlin. The IDE is also often used for mobile development in Android or Cordova. It could be very practicable for web development because of the different frameworks like JavaScript, AngularJs, Node.js, TypeScript etc... 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

6.1.2 Maven

This is a software management and build automation tool which is based on the concept of a project object modem (POM). One of the biggest feature in Maven is the dependency management. Maven automatically downloads in POM file declared libraries and plug-ins from a repository and stores them local. The local repository is a folder structure that is used as a centralized storage place for locally built artifacts and as a cache for downloaded dependencies. 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.

6.1.3 Scene Builder

JavaFX Scene Builder is a visual layout tool that generates FXML, an XML-based markup language that lets developers to 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 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 supportive IDE.

6.1.4 JavaFX

JavaFX 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. The newest JavaFX releases are fully integrated with the Java SE 7 Runtime Environment, which is available for all main desktop platforms. and the Java development Kit. So JavaFX application compiled to JDK 7 or later also run on all major desktop.

6.2 Surrounding 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 acquiring 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
- **scanner:** To analyze NoBeard source code files for keywords, operands or arguments
- **symbolTable:** Looks up for matching words based on the symbol table entry

However, one of the most important package is the “machine”. This is defined as the core of the whole project because it has the most significantly roles such as running a NoBeard program.

6.3 Architecture of the User Interface

The GUI is designed very similar to the Model-View-Controller pattern. However, this project is not depending on any database so it comes without any Model. The GUI project is build up of three main components as shown in figure 6.1. The Controller and the Virtual Machine both of them is running on different threads and are accordingly synchronized. This will be described in detail in section 6.4. 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.

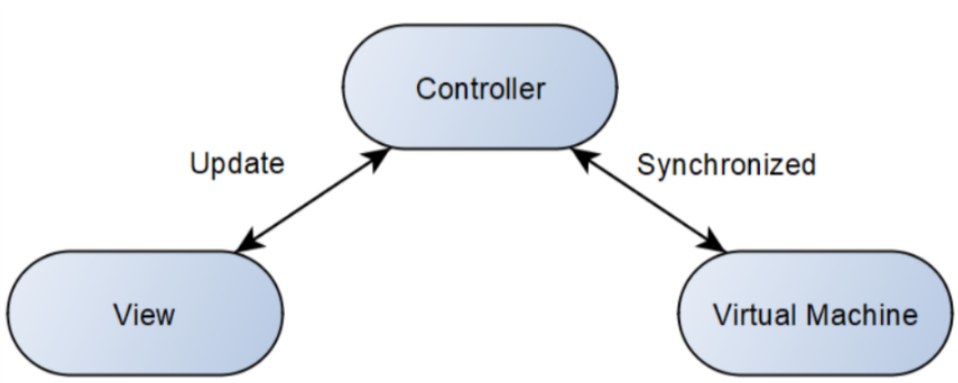


Figure 6.1: Main parts of the GUI implementation

6.3.1 The View

The view is basically a single `.fxml` file which holds the view components of the layout. As figure 6.2 shows it was created with an external tool called Scene Builder which is already described in section 6.1.3. Each window is made up of an AnchorPane and is separated by SplitPanes. AnchorPane enables developers to construct high rated dynamic user interfaces for desktops. It is a highly specialized layout that can be used for edge alignment. This benefit gives the possibility to fill the whole pane to the parent, which is a SplitPane, by setting anchor offsets to zero. So, it becomes very comfortable and easy to size every window as the user wants. Then each of these AnchorPanes 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 an UI that meats 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 a 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, just creativity. The code snippet 6.1 shows a simplified version of how the control buttons were defined in the fxml file.

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

Listing 6.1: FXML example

Dynamic Design

In dynamic design the already existing view is being changed according to run time or to user interactions. This means that the view changes only 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. The code snippet 6.2 shows a function which is called after a user opens a binary file. This function fills the program window with the content and additionally inserts to each line a CheckBox for the maintaining of breakpoints.

```
private void fillProgramDataView(List<String> programDataList) {
    VBox programData = new VBox();
    for (String lineStr : programDataList) {
        CheckBox line = new CheckBox(lineStr);
        line.setPadding(new Insets(1));
        line.setOnAction((event) -> {
            if (event.getSource() instanceof CheckBox) {
                CheckBox breakpoint = (CheckBox) event.getSource();
                if (breakpoint.isSelected())
                    machine.addBreakpoint(getAddressOfProgramLine(breakpoint.getText()));
                else
                    machine.removeBreakpoint(getAddressOfProgramLine(breakpoint.getText()));
            }
        });
        programData.getChildren().add(line);
        programDataMap.put(getAddressOfProgramLine(line.getText()), line);
    }
    programDataView.setContent(programData);
```

```
19 }
```

Listing 6.2: Implementation of program data view

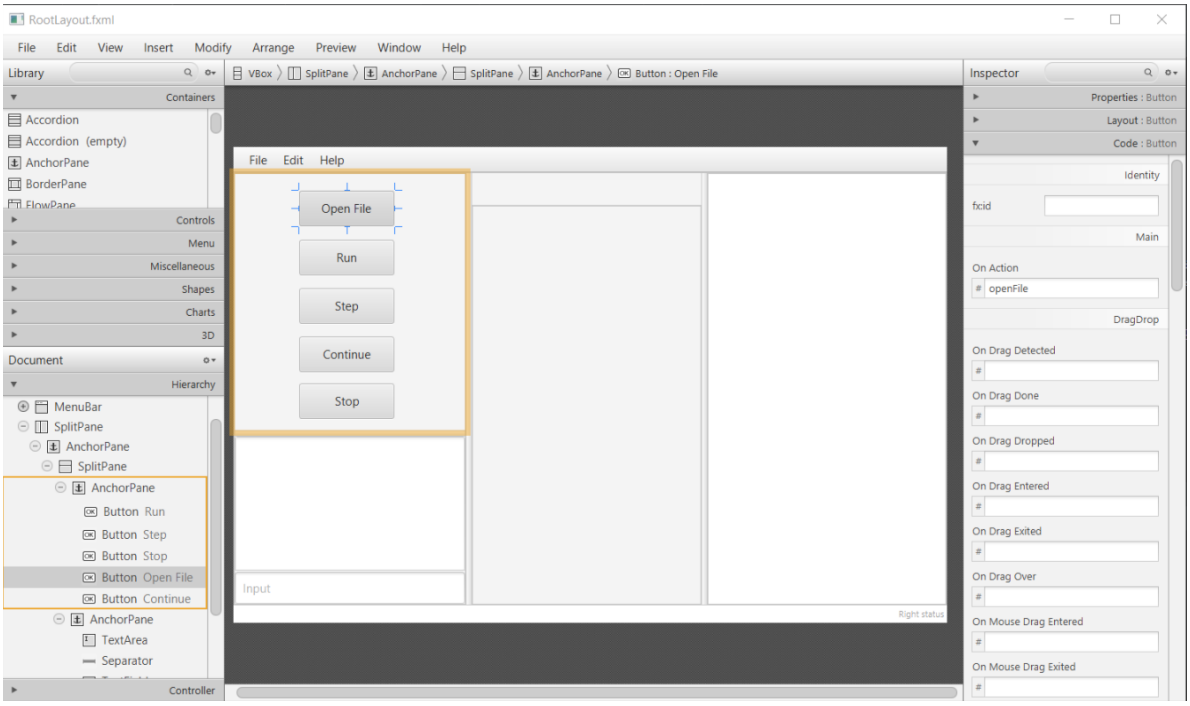


Figure 6.2: Development of the view with Scene Builder

6.3.2 The Controller

The controller is primarily responsible for the coordination and operation of the view and its components. As already described in section 6.1.3 the controller can have easy access to the view components via the binding of an fx id. But firstly, they have to be connected with each other by setting the `fx:controller` attribute of the root view element to the currently used controller. Through this advantage it does not need to now the resource file of the layout or to call any "get" functions to get the target items. Even event actions, like mouse click or key pressing, 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 the `@FXML` annotation and the same name as it was set on the action property of the item. Even functions like this comes only with a single argument which an ActionEvent. This parameter could be very helpful for getting some further information of the fired event like which mouse button was pressed. The other big task that the controller has to handle is to execute user commands on the virtual machine.

So it acts like a bridge to the machine and calls functions like run program, step one instruction further or view data etc...

Initially, the controller gets an instance of the NoBeard machine which is being target for NoBeard assembler programs. The opening of a NoBeard object file is operated with the help of a BinaryFileHandler. After a successful opening of the object file, it has to be disassembled. This means the disassembler has to convert the content of a binary file to primary program data where addresses, instructions and operands become visible on the GUI. By finishing the translation, the machine has to load the strings 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 gets an OnAction event which adds and removes breakpoints from the machine. By starting to run a program, a new external thread has to be started where the machine runs separate from the UI. The code 6.3 illustrates how the machine gets started on a background thread using lambda expression.

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

Listing 6.3: Starting the machine on a new thread

The reason why and how they run on different threads will be explained in section 6.4. The machine executes step by step every instruction of the program until any interruptions. It can be interrupted by a breakpoint, input request or by a `halt` instruction.

6.4 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 NoBeardMachine 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 6.4. 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          controller.enableInputView(true);
10         controller.getSemaphore().acquire();
11     } catch (InterruptedException e) {
12         e.printStackTrace();
13     }
14 }
```

Listing 6.4: 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 6.5 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  inputView.setOnKeyPressed(event -> {
2      if (event.getCode() == KeyCode.ENTER && inputView != null &&
3      !inputView.getText().isEmpty())
4          inputIsAvailable(inputView.getText());
5  });
6
7  private void inputIsAvailable(String providedInput) {
8      getOutputView().appendText(providedInput + "\n");
9      input = providedInput;
10     inputView.clear();
11     enableInputView(false);
12     getSemaphore().release();
13 }
```

Listing 6.5: Synchronisation with semaphore (Part 2)

6.5 Handling of Breakpoints

Machine internal handling of breakpoints is done as described in [Ben]. 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 Control-Unit 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 debugger 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 debugger class contains in all three major functions as shown in the code snippet 6.6. Adding, 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(Integer address, Byte instructionId) {
2      this.breakpoints.put(address, instructionId);
3      programMemory.replaceInstruction(address, InstructionSet.Instruction.BREAK.getId());
4  }
5
6  void removeBreakpoint(Integer address) {
7      programMemory.replaceInstruction(address, breakpoints.get(address));
8      this.breakpoints.remove(address);
9  }
10
11 public void replaceInstructionAtAddress(int address, InstructionSet.Instruction instruction) {
12     if (!breakpoints.isEmpty() && breakpoints.keySet().contains(address)) {
13         if (instruction != null)
14             programMemory.replaceInstruction(address, instruction.getId());
15         else
16             programMemory.replaceInstruction(address, breakpoints.get(address));
17     }
18 }
```

Listing 6.6: 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 which is stored in the HashMap. Now the machine is blocked at a specified breakpoint and the user can take a look to the current stack frames and go one step further in the program 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.

6.6 Visualisation of DataMemory

The visualisation of the data memory gives an overview about string constants and stack frames. As figure 6.3 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 6.3, more precisely at line 7. 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

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 6.3: Visualisation of data memory

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 6.3. 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.

```
1 static void update(Controller controller, ObservableList<String> content) {
2     controller.getDataMemoryListView().setItems(content);
```

```
3     controller.getDataMemoryListView().setCellFactory(list -> new ListCell<String>() {
4         int framePointer = controller.getMachine().getCallStack().getFramePointer();
5         int stackPointer = controller.getMachine().getCallStack().getStackPointer();
6         static final int INDEX_OF_ADDRESS = 0;
7
8         @Override
9         protected void updateItem(String item, boolean empty) {
10            super.updateItem(item, empty);
11            if (item != null) {
12                createDataLine(item);
13            }
14        }
15
16        private void createDataLine(String item) {
17            HBox line = new HBox();
18            int firstAddressInLine = getIndex() * 4;
19            convertStringToLabels(firstAddressInLine, item).forEach(line.getChildren()::add);
20            setContextMenuToDataCells(line);
21            setGraphic(line);
22        }
23
24        private List<Label> convertStringToLabels(int firstAddressInLine, String line) {
25            String[] lineContent = splitDataLine(line);
26            List<Label> result = createLabelForAddress(lineContent[INDEX_OF_ADDRESS]);
27            int currentAddress = firstAddressInLine;
28            for (int i = 1; i < lineContent.length; i++) {
29                if (currentAddress == framePointer)
30                    result.add(createHighlightedLabel(lineContent[i], "#0038AC"));
31                else if (currentAddress == stackPointer)
32                    result.add(createHighlightedLabel(lineContent[i], "#AC080E"));
33                else
34                    result.add(createNormalLabel(lineContent[i]));
35                currentAddress++;
36            }
37            return result;
38        }
39        ...
```

Listing 6.7: Implementation of the data memory view using cell factory

In the code example 6.7 an anonymous inner class is created using modern 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 react 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 four functions are available:

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

For the implementation of these 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. So if all the four bytes are not aligned in one row, the rest bytes has to be occupied from the next row. And integers are stored in little endian order, that means the lowest significant byte is stored first.

Chapter 7

Summary

Here you give a summary of your results and experiences. You can add also some design alternatives you considered, but kicked out later. Furthermore you might have some ideas how to drive the work you accomplished in further directions.

Bibliography

[Bau17] Peter Bauer. P. Bauer, "The NoBeard Report," HTBLA Leonding, Leonding, 2016. Technical report, June 2017. original-date: 2016-06-13T13:15:03Z. URL: https://github.com/Bauepete/no-beard/blob/master/doc/NoBeardReport/NoBeardReport.pdf.

[Ben] Eli Bendersky. *How debuggers work: Part 1 - Basics - Eli Bendersky's website*. URL: https://eli.thegreenplace.net/2011/01/23/how-debuggers-work-part-1/.

[wik18] Two's complement, July 2018. Page Version ID: 849910398. URL: https://en.wikipedia.org/w/index.php?title=Two%27s_complement&oldid=849910398.

List of Figures

Listings

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.