Extension of a Simulator for a Microprogrammable Processor with Automatic Tests

Bachelor's Thesis

by

Loïc Duval Lekeuagni Nguegang

born in

Bamendou II

Group for Computer Networks

Janine Golov

Heinrich-Heine-University Düsseldorf

14 September 2023

Supervisor:

Janine Golov

Abstract

This thesis presents an extension of a simulator of a microprogrammable processor, originally implemented in Java 8 to fulfill the requirements of a bachelor's degree program [Loe23], with automatic tests. The problem statement highlights the challenges associated with the development of the simulator, including the absence of systematic testing and outdated technology.

The project, based on Andrew Tanenbaum's Mic-1 microprocessor design [Tan01], underwent a series of transformative changes. These changes encompassed structural reorganization into a Gradle-based project, upgrade java language-level, and the introduction of dependency injection and design patterns to enhance code modularity and testability.

Moreover, the project was segmented into distinct packages, facilitating a more logical and efficient organization. The key innovation was the creation of a central `State.java` class to manage the program's state effectively. In addition, the absence of interfaces was rectified, enabling more effective unit testing practices.

This thesis outlines the motivation, challenges, and the methodology employed to achieve these enhancements. Furthermore, it explores the implications of these changes, offering a comparative analysis of the project before and after the improvements. The document concludes by summarizing the achievements, and suggesting potential future enhancements for the Mic-1 Simulator.

Overall, this thesis provides valuable insights into the extension of the MIC-1 simulator and the use of automatic tests to ensure that the classes and methods composing the software do what they're expected to do.

Acknowledgments

I would like to express my heartfelt gratitude to my supervisor, Frau Janine Golov, for her continuous support throughout my work. I would also like to extend my thanks to my co-referee, Herr Dr. Markus Brenneis.

Special thanks go to my parents for their unwavering support in difficult times, not forgetting my siblings, other family members and friends for their encouraging words.

Contents

List of Figures

List of Tables

Chapter 1.

Introduction

In the previous semester a software called **Mic1-Simulator** was developed and described by a student [Loe23], intended for educational purposes in the module ‘computer architecture’ at Heinrich Heine University. However, this simulator, designed for the Mic-1 microprocessor described by Andrew Tanenbaum in his book “Structured Computer Organization” [Tan01], exhibits certain limitations, such as the inability to easily add new microprograms and the absence of automated tests to validate critical method correctness. These aforementioned aspects will constitute the central emphasis of this thesis.

1.1. Motivation, Background and Problem Statement

The Mic-1 simulator, developed with pedagogical objectives in mind, should be employed to facilitate the teaching and learning of microprocessor architecture. However, it’s crucial for future end-users to rely on the accuracy of its output, as any errors in its functionality can lead to misunderstandings and hinder effective learning. So, the motivation was rooted in ensuring that future users can place complete trust in the simulator’s output.

The simulator presented an opportunity for enhancement on multiple fronts. The absence of systematic testing made it challenging to determine if the software contained errors, while the use of outdated Java 8 technology and the lack of proper code structuring hindered its maintainability. Consequently, the necessity for a tested Mic-1 simulator became apparent, serving as the core challenge that this thesis aims to address.

1.2. Structure of the Thesis

The subsequent chapters of this thesis present detailed steps undertaken to extend the Mic-1 simulator. Chapter 2 deals with the background, setting the scene by exploring the field of software development and unit testing. Chapter 3 outlines the methodology adopted, detailing the steps involved in refactoring, and enhancing the simulator. The results and analysis of changes made, are expounded upon in Chapter 4.

The thesis culminates in Chapter 5, which summarises the achievements and contributions of this work and provides directions and ongoing exploration.

Chapter 2.

Background

To better understand the next chapters of this thesis, it is important to understand the initial state of the software that forms the basis of this work and to understand the notion of automated software testing. The initial part of this chapter will provide a concise introduction to software development and testing. It will highlight the importance of software testing and the consequences of inadequately tested software. Subsequently, the second segment will delve into the prior work.

2.1. Overview of Software Development and Testing

Software testing is an evaluation and verification process to ensure a software product or application performs its intended functions [Doo17]. Testing brings several advantages, including bug prevention, decreased development expenses, and enhanced performance. Besides, delivering bug-free software, software testing helps to improve the functionalities and usability of applications. In the field of software engineering, tests are categorized into three main types: unit testing, integration testing, and end-to-end testing. In this thesis, the focus will be on unit testing.

2.1.1. Importance of Unit Testing

Unit tests form the foundation of the test automation pyramid. The unit test layer is where much of the testing will take place and is normally written to examine every individual components of code in isolation. This approach will allow us to perform designed test for each individual critical method that could comprise the Mic1 Simulator. Let's explore some advantages of adopting a unit testing approach:

Well-crafted tests also function as dynamic documentation, providing developers with a comprehensive understanding of the logic within each module and the system overall [Kat]. This, in turn promotes better code design and organisation. For example, at the start of this project the received software lacked any comments or documentation describing the function of the various methods. This made it difficult to join the project and understand the responsibility and functionality of each method or component code. This probably wouldn't have been the case if the tests had already been written, as having a look at the inputs and assertions would help to understand the requirements of the methods and the output they generate.

Another advantage of unit testing is its precision [Kat]. By concentrating on discrete code units, developers can meticulously examine the smallest functional building blocks of a software system. This micro-level enables the detection of errors, inconsistencies, and unintended behaviours at an early stage, preventing the propagation of defects to higher levels of the codebase.

Unit testing also provides an immediate feedback loop for developers [AU19]. As units are tested in isolation, results are obtained swiftly, allowing developers to identify and rectify issues promptly. This rapid feedback accelerates the development process, fosters agile iterations, and supports continuous integration (CI). It also empowers developers with the confidence to refactor and modify code without fear of breaking existing functionality [MT04]. A robust suite of tests acts as a safety net, ensuring that any changes made do not introduce regressions. This confidence in refactoring encourages code improvement, promotes architectural enhancements, and contributes to long-term code maintainability.

2.1.2. Consequences of not tested software

The consequences of not testing software can be significant and far-reaching, affecting both developers and end-users. Testing plays a crucial role in ensuring the quality, reliability, and functionality of software applications. When software is not adequately tested, several negative consequences can arise.

- Bugs and Errors [Lit79]: Untested software is more likely to contain bugs, errors, and defects. These can range from minor issues that are inconvenient for users to critical errors that cause crashes or data corruption. Bugs can lead to unexpected behaviour, security vulnerabilities, and system failures.

- Reduced Reliability: Without testing, software's reliability and stability could be compromised without raising any red flags. For example, let's imagine a method of the simulator responsible for parsing integer to short and let's imagine that this calculation has a loss value, which should be considered for the following calculation, but isn't. This type of code failure is common in the software development and can be tricky to detect. Until the converted value would be smaller or equal `Short.MAX_VALUE`, the method would seem to work perfectly.

- User Dissatisfaction: Software that lacks proper testing might not meet user expectations or requirements. This can result in poor user experiences, leading to dissatisfaction. According to the last point, a user, who would like to use only big numbers would either always get wrong results or may abandon the software in favour of alternatives that offer better functionality and reliability.

- Increased Maintenance Costs [IBM]: Identifying and fixing issues in software that was not thoroughly tested can be time-consuming and costly. As bugs surface after deployment, developers must allocate resources to diagnose and rectify the problems. The longer it takes to identify issues, the more expensive the fixes become.

After concluding the overview of software development and testing methodologies, as well as exploring the ramifications of untested software, attention will shift to the examination of the efforts made by the previous developer of the Mic-1 simulator.

2.2. State of the Art

In summary, the software possesses the capabilities of the MIC-1 microprocessor designed by Andrew S. Tanenbaum in his book "Structured Computer Organization" [Tan01]. This means that a user can use it to write IJVM- and MAL-code and execute it step by step using the simulator, while also having access to the value of each register after every clock cycle.

2.2.1. Former Project Structure

The project [Loe23] was organized into two distinct packages (see Listing 2.1). The first package, labelled backend, encompassed a collection of classes intricately tied to the program's logic and state management. Within this package, essential classes were located, including `InstructionParser.java`, `MalParser.java`, `CodeParser.java`, `CPU.java`, `Alu.java`, `MicroinstructionParser.java`, and several other classes, each contributing to the core functionality of the application.

The second package was named sample and served as a repository for the classes responsible for user interaction and control. Here, `Main.java` held a pivotal role, accompanied by an assembly of Controller classes, collectively forming the bridge between user inputs and the underlying program operations. This well-structured division ensured a clear separation of concerns within the project as publishes in [Loe23].

```

   /* Old Project Structure */

./root
|-- LICENSE
|-- MIC-1-Simulator.iml
|-- README.md
|-- resources
|__ src
   |-- backend
   |   |-- ALU.java
   |   |-- BitValues.java
   |   |-- ByteMemoryLine.java
   |   |-- CodeExample.java
   |   |-- CodeParserException.java
   |   |-- CodeParser.java
   |   |-- CPU.java
   |   |-- FileParser.java
   |   |-- HighBit.java
```

```
|   |-- InstructionParser.java
|   |-- MALParserException.java
|   |-- MALParser.java
|   |-- Memory.java
|   |-- MemoryLine.java
|   |-- MicroinstructionParser.java
|   |-- NumericFactory.java
|   |-- ObservableResourceFactory.java
|   |-- O.java
|   |-- Register.java
|   |__ Shifter.java
|__ sample
    |-- AboutController.java
    |-- ContactController.java
    |-- Controller.java
    |-- HelpController.java
    |-- Host.java
    |__ Main.java
```

Listing 2.1: Old Project Structure

2.2.2. Role of Distinct Classes

This subsection will provide concise descriptions of the key classes mentioned earlier and explain their individual roles and contributions to the functionality of the proceeding project [Loe23]. This will help readers gain a clearer understanding of how these classes interact and collaborate to achieve the project's objectives.

- ***Alu.java:*** The `Alu.java` class holds a pivotal role within the proceeding project [Loe23], playing a vital part in executing Arithmetic Logic Unit (ALU) calculations. Central to its functionality is the method `calculate(...)`, a cornerstone method that orchestrates the intricate process of ALU operations. This method takes inputs, processes them in accordance with the defined logic, and generates results that are crucial to the overall operation of the microprogrammable processor.

  Beyond its mathematical significance, the `Alu.java` class also carries the responsibility of flag management. As part of its operation, it meticulously sets flags to reflect the outcome of the ALU calculation. These flags serve as indicators of specific conditions or results, providing valuable insights to subsequent program flow and decision-making processes.

In essence, this class encapsulates the essence of microprocessing intricacies, offering a platform for ALU calculations and flag manipulation. Through its method calculate, it embodies the core essence of the Arithmetic Logic Unit, a foundational component within the microprogrammable processor's operation.

- ***NumericFactory.java*:** The `NumericFactory.java` class serves as a versatile utility for converting and formatting numeric values across different bases, including decimal, hexadecimal, and binary. It offers a collection of static methods that enable seamless transformation of numbers while maintaining flexibility and readability.

- ***InstructionParser.java*:** The `InstructionParser.java` class encapsulates the functionality of parsing and managing assembly instructions used in microprogrammed processors. It offers methods to interact with and extract relevant information from instructions, making it an essential component for decoding and handling assembly code. The methods in this class have some particular roles. For example: `getInstructionBytesCount(...)` calculates the total number of bytes required for a given instruction, `addInstruction(...)` adds a new assembly instruction to the supported instructions map along with its corresponding opcode and custom parameter value.

- ***CodeParser.java*:** Together with `InstructionParser.java`, the `CodeParser.java` class plays a pivotal role in translating human-readable assembly code into machine code that can be executed by the microprocessor. One of the most important methods of this class is `parseCode(...)`, which accepts a multi-line string of assembly code and processes it to generate an array of short integers representing the machine code. It meticulously handles various aspects of instruction parsing, such as labels, operands, mnemonics, and arguments. The method `purifyCode(...)` has the role of cleaning the assembly code lines by removing comments, extraneous spaces, and labels, while calculating memory addresses based on instruction byte counts.

- ***MicroinstructionParser.java*:** The `MicroinstructionParser.java` class is responsible calculating microinstructions based on a given MAL (Microprogramming Assembly Language) instruction. It offers methods for processing various parts of the instruction and calculating the corresponding microinstruction value.

- ***MALParser.java*:** This class serves as the main component for parsing and processing Microprogramming Assembly Language (MAL) code. It offers methods for retrieving various properties of the parsed code. The `parseMAL(...)` method is likely the core of the class, responsible for processing the MAL code and generating control memory values.

- ***Memory.java*:** this class represents the memory of the system with stack memory and method area memory. It provides methods for initializing memory regions and counters, checking readiness, reading and setting memory addresses and data, as well as methods for reading and writing memory contents. The class uses `ObservableList` to manage and track changes to memory lines.

- ***CPU.java*:** Overall, `CUP.java` class is the most important class. It simulates the internal operations of a CPU, handling memory access, instruction decoding, ALU operations, and various other functionalities involved in executing machine instructions.

Chapter 3.

Task definition, Objective and Methodology

By enhancing the Mic-1 microprocessor simulator and ensuring its reliability, a structured approach was undertaken. This chapter delves into the methodology employed to transform the existing software into a more refined and dependable system. The chapter is divided into three primary sections.

The initial section of this document is dedicated to identifying gaps and challenges, followed by the second section, where the details of refactoring and optimizing the code structure will be explored. this section will also delve into the reasons behind the reorganization of the project's layout and the adoption of design patterns

The third section closely examines the process of implementing unit tests. Recognizing the fundamental importance of unit testing, this section guides us through the decisions behind test selection, used frameworks, and the strategic approaches employed to ensure comprehensive test coverage. Together, these sections provide the systematic steps taken to enhance the project's structure and reliability.

3.1. Identifying Gaps and Challenges

The analysis of the previous project revealed several notable gaps and challenges that prompted the need for comprehensive improvements. These gaps encompassed aspects such as the absence of effective testing strategies, project structure and code quality.

3.1.1. Absence of Comprehensive Testing: Impeding Bug Detection

One of the most challenges was the lack of any meaningful testing framework in the previous project. The absence of unit tests, integration tests, or any automated testing strategy compromised the reliability of the system (e.g. For example, it was possible to replace the instruction of any "MAL" program simply by selecting a wrong address. see 3.1.1). Without proper tests, it was not easy to validate the correctness of individual components or ensure the software's behaviour as a whole.

The lack of unit tests, or any other automated testing framework meant that bugs and errors could persist within the system without detection. Minor code changes could introduce unintended side effects that could only surfaced during runtime. This deficiency hindered iterative development, as changes could inadvertently disrupt existing functionality, necessitating cautious coding practices to avoid introducing regressions. Without automated tests to validate behaviour across scenarios, the impacts of changes can be unpredictable. This could not only lead to complicate debugging but, could also make challenging to identify root causes. Let's illustrates this with an example.

- Consider a method responsible for adding a microprogram within the microprocessor simulation. Without unit tests, this method appeared functional. However, specific input (see MAL program 3.1) conditions triggered an unanticipated bug, causing the method to overwrite an existing MAL instruction (see listing 3.2). The third line `0x0 MDR = TOS = MDR + H; wr; goto Main1` (listing 3.1) has been chosen such that the MAL microprogram of the IJVM command "NOP" will be modificated at the address *"0x0"*. Here it's important to note that the simulator starts with a predefined state. This means there are default entries in the control memory, and IJVM commands such as `NOP, BIPUSH, ISTORE...` already exist in the control store. Additionally, the address "0x0" is automatically assigned to the IJVM command `NOP`. The

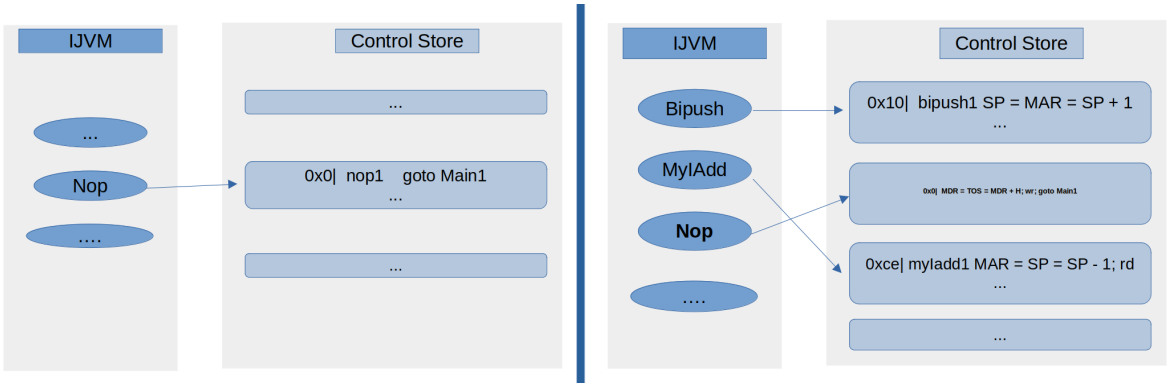
following shows he state of the control memory before and after adding the MAL program (see listing 3.1).

```
1       myIadd1 MAR = SP = SP - 1; rd
2       myIadd2 H = TOS
3       0x0     MDR = TOS = MDR + H; wr; goto Main1
4
```

Listing 3.1: Add a Microprogram with fixed Address. (See last line of the MAL program)

- Before adding the MAL program (listing 3.1) the address *"0x0"* in the control store is the one, which had been chosen per default to store the microinstruction of microprogram responsible for the "NOP" command. This microprogram has only one instruction (`0x0: goto Main1;`) and should do nothing. and shouldn't change the value of data registers like `LV, CPP, TOS, OPC, H, MAR` or `MDR`. now what happened, when the microprogram mentioned above will be added?

- After adding the microprogram, it was noticeable that the address *"0x0"* of the control store has also been modified. The old microinstruction residing in this address has been overwritten and replaced by the microinstruction `0x0: MDR = TOS = MDR + H; wr; goto Main1`. In the first hand, this would work correctly until the opcode "NOP" will appear in the IJVM program. – *Just for clarification: After adding the new microprogram, the address responsible for "NOP" IJVM command has been overwritten, this command is still referring to that address. and the new microprogram is also referring to the same address (see figure 3.1).* – Now we're going to see how this can break the reliability of the software.



(a) (b)

Figure 3.1.: The picture 3.1a shows the first address in the control store of the microprogram indexed by the IJVM command before adding the microprogram `MyIAdd` (see listing 3.1). and picture 3.1b shows the same but after adding it.

- Now, it's possible to write a designed IJVM code such that the simulator will give wrong results. After the execution of the IJVM code (listing 3.2), the value "0x8" will be at the top of the stack instead of "0x5". The table (3.1) shows the state of the stack pointer after the execution of the IJVM code referenced above.

```
1
2       bipush 2 // push 2 onto the stack
3       bipush 3 // push 3 onto the stack
4       myIadd // sum 2 + 3 and push result (the value ''5'') onto the stack.
5       NOP // Here, this command will sum 5 + 3 again and push 8 onto the stack
    because the value 3 is in the H resgister, and 5 is in the MDR. And, the command
    ''NOP'' executes MDR = TOS = MDR + H; wr; goto Main1
6
```

Listing 3.2: Manipulated IJVM code.

Address	Value
...	...
132096	0x0
132097	0x8
132098	0x3
132099	0x0
...	...

Table 3.1.: State of the stack after running IJVM program (listing 3.2).

The visualization illustrates the path taken by the faulty methods, resulting in unexpected outcomes that couldn't have been foreseen without proper testing. This bug remained dormant within the codebase until it was unintentionally triggered during a user simulation, underscoring the need for systematic testing.

3.1.2. Project Structure, Modularity and Documentation

The initial project's structure lacked clear modularization, leading to code entanglement and difficulties in understanding component interactions. A flat package hierarchy hindered the separation of concerns and made it challenging to maintain or extend the system. The absence of a logical and organized architecture created confusion attempting to grasp the codebase. The inadequacy of the original structure impeded collaboration and made it hard to isolate issues or implement improvements efficiently.

The project's architecture also exhibited fragility in the face of changes. Modifications in one part of the codebase sometimes resulted in unintended consequences in other areas. This fragile architecture was exacerbated by the lack of comprehensive documentation that could guide developers through the code and explain critical design decisions. Without proper documentation, the process of understanding, and extending the system was impeded.

3.2. Refactoring and Code Structure

To enhance the Mic-1 microprocessor simulator and lay the groundwork for future development, a deliberate effort was invested in refactoring the existing codebase. This involved an analysis of the project's structure, recognizing areas that could be optimized for clarity, maintainability, and extensibility. This section sheds light on the refactoring undertaken and the principles that guided the restructuring process.

3.2.1. Rationalizing the Project Structure

The initial project structure, while functional, exhibited certain complexities that hindered its testability and its potential for growth. Recognizing the significance of a coherent and modular architecture, the decision was made to reorganize the project into more distinct components. The primary goals were to enhance the clarity of code relationships, and simplify future updates.

Reorganizing Class Hierarchies for Clarity and Coherence.

In the prior iteration (according to listing 2.1), classes were confined to two overarching packages, which at times hindered the ease of navigation and maintainability. To address this, a twofold approach was adopted. Firstly, the main class was strategically relocated to the root package, aligning with a more intuitive entry point for the simulator. This adjustment not only facilitated a clearer project organization but also served as a stepping stone for further restructuring. And secondly, the existing monolithic packages were deconstructed into smaller, more focused subpackages: `exceptionHandler`, `parser` and `state`.

Classes with shared responsibilities and functionalities were logically grouped together, fostering a more granular and manageable architecture. This reorganization improved the clarity of class relationships. The restructured project (see listing 3.3) now boasts a hierarchy of distinct subpackages, each serving a unique and pivotal role in the functionality of the microprocessor simulator.

```

      /* New Project Structure */


./root
|-- build.gradle
|-- gradle
|-- gradlew
|-- gradlew.bat
|-- LICENSE
|-- README.md
|-- settings.gradle
|__ src
   |__ main
       |-- java
       |   |-- de
       |   |   |__ hhu
       |   |       |__ MIC_1_Simulator
       |   |           |-- backend
       |   |           |   |-- AluInterface.java /* New Class */
       |   |           |   |-- ALU.java
       |   |           |   |-- CodeExample.java
       |   |           |   |-- CPU.java
       |   |           |   |-- NumericFactory.java
       |   |           |   |__ ObservableResourceFactory.java
       |   |           |-- exceptionHandler
       |   |           |   |-- CodeParserException.java
       |   |           |   |__ MALParserException.java
       |   |           |-- Main.java
       |   |           |-- parser
       |   |           |   |-- CodeParser.java
       |   |           |   |-- FileParser.java
       |   |           |   |-- InstructionParser.java
       |   |           |   |-- MalDto.java  /* New Class */
       |   |           |   |-- MALParser.java
       |   |           |   |__ MicroinstructionParser.java
       |   |           |-- sample
       |   |           |   |-- AboutController.java
       |   |           |   |-- ContactController.java
       |   |           |   |-- Controller.java
       |   |           |   |-- HelpController.java
       |   |           |   |-- Host.java
       |   |           |   |-- ImagesMapping.java  /* New Class */
       |   |           |   |__ NammigMapping.java  /* New Class */
       |   |           |__ state
       |   |               |-- BitValues.java
```

```
47      |   |                |-- ByteMemoryLine.java
48      |   |                |-- HighBit.java
49      |   |                |-- Memory.java
50      |   |                |-- MemoryLine.java
51      |   |                |-- O.java
52      |   |                |-- Register.java
53      |   |                |-- Shifter.java
54      |   |                |-- Statable.java  /* New Class */
55      |   |                |__ State.java  /* New Class */
56      |   |__ module-info.java  /* New Class */
57      |__ resources
```

Listing 3.3: New Project Structure

As the package and class names in the listing (3.3) indicate, the state package encapsulates classes responsible for maintaining the state of the simulator during execution, while the backend package houses components that form the backbone of the simulator's operation, including the execution engine and relevant functionalities. Furthermore, the `exceptionHandler` package handles exception scenarios, when the package parser contains classes dedicated to parsing and interpreting instructions, and the last package (sample) for user interaction and interface management. The outcome of these changes was twofold: a more intuitive and streamlined project entry point and a meticulously organized class hierarchy that promoted both cohesion[1] and comprehensibility.

Enhancing Scalability through Gradle

Another important step to fortify the project's architecture was the introduction of a gradle[2] build file (according to listing A.1). This addition empowered the project with a modern and dynamic build automation tool, enabling efficient management of dependencies, compilation, and testing processes. The integration of gradle not only streamlined the development workflow but also laid the foundation for future scalability. With gradle's intuitive configuration capabilities, the project's dependencies could be elegantly managed, eliminating potential conflicts and version mismatches. This translated into a more stable and manageable environment, wherein the addition of new features or libraries seamlessly integrated without disrupting the existing codebase.

[1] "Cohesion is an attribute of a software unit or module that refers to the 'relatedness' of module components." [BO94] A module with high cohesion serves a singular, indivisible purpose, making it intricate to separate its components.

[2] Gradle is an open-source build automation tool flexible enough to build almost any type of software. (See `https://docs.gradle.org/current/userguide/what_is_gradle.html`)

Furthermore, incorporating the gradle build system facilitated a modular approach to dependency management. Modules and libraries crucial to the simulator's functionality could be clearly defined, ensuring a fine-grained control over the project's external components. This modularization not only simplified the inclusion of external resources such as test dependencies, but also mitigated the risk of dependency-related issues. The gradle build file encapsulated the project's configuration and build settings, contributing to a more organized and cohesive codebase. This centralization simplified the onboarding process for new developers, as they could readily access the project's configuration details without navigating through an intricate maze of files. Another inherent advantage of gradle lies in its compatibility with various testing frameworks. By incorporating gradle, the project embraced a seamless integration of testing procedures into the development cycle. Unit tests, essential for validating the correctness of individual components, could be effortlessly executed as part of the build process.

3.2.2. Design Patterns and Dependency Injection

To create a more modular and maintainable microprocessor simulator, a restructuring was undertaken to address architectural challenges that hindered extensibility and testability. Key elements of this initiative included the integration of dependency injection and the introduction of interfaces. This section explores how these fundamental changes contributed to the enhancement of project structure and overall development efficiency.

The first action undertaken was the externalization of the state: By taking a look at the previous `CPU.java` class, it's immediately noticeable that this class is not only responsible for executing a MAL program, but also for managing the state of a part of the simulator. This makes the class complicated to understand and to test its functionality. So, one of the biggest changes made was to extract the state in each of these classes (like the `CPU.java` class) and put them together in an external package called state and containing classes for storing the state of the simulator during its lifecycle. The purpose of this action was to simplify the project and make it easier to understand and write the tests.

Another issue of the original project was the tight coupling of components, resulting in limited flexibility and hindered testability. The introduction of dependency injection marked a transformative shift by externalizing the creation and provision of dependencies. With this

approach, components were no longer solely responsible for instantiating their dependencies, but rather relied on external mechanisms to provide these dependencies. This architectural adjustment not only reduced coupling but also enabled easier substitution of components during testing. For example: the state of the simulator could be mocked, allowing precise behaviour testing.

A notable absence in the previous simulator was the lack of interfaces, which limited the ability to create mock implementations for testing purposes. In the new architecture, interfaces were introduced to abstract the interactions between components. By defining clear contracts through interfaces, components could now be tested independently, with mock implementations facilitating comprehensive unit testing. For example (see listing 3.4), we want to test the behaviour of the method `runSubCycle()` of the class `CPU.java`, when the clock value is "0". The problem here is that the clock's value change after every subcycle and there's no setter for this attribute and normally this value should not be changed from outside of the cpu class. This means, it should not be possible to do something like this: `cpu.setClockState(int_value)`. Another problem it that once the `runSubCycle()` method has been executed, this triggers the execution of several other methods, which, once certain values have been checked, may raise exceptions. But, since the aim of the test is just to check whether the `runFirstSubCycle()` method will be executed once, it's a necessary to find a clean and simple way of performing this type of test while avoiding creating a very large state object and having to initialize all its variable to prevent exceptions. And with the help of interfaces and dependency injection, it's possible to take charge of this kind of case and test correctly, while modifying the behaviour of certain methods according to the problem being tested, using mocks. This decoupling of interfaces from concrete implementations facilitated modular development and systematic testing, leading to higher code quality and maintainability.

```
@Test
@DisplayName("when state.getClock is 0, runSunCycle will call runFirstSubCycle")
void runSubCycle_will_call_runFirstSubCycle_when_state_getClock_is_0(){
  // Arrange
  when(state.getClock()).thenReturn(new SimpleIntegerProperty(0));
  willDoNothing().given(cpu).runFirstSubCycle();
  // Act
  cpu.runSubCycle();
  // Verify
  verify(cpu, times(1)).runFirstSubCycle();
  verify(cpu, times(0)).runSecondSubCycle();
  verify(cpu, times(0)).runThirdSubCycle();
  verify(cpu, times(0)).runFourthSubCycle();
```

```
14 }
```

Listing 3.4: Example Usage of Mock and State Interface

The combined impact of dependency injection and interface-driven development has been substantial. The new architecture not only addresses the shortcomings of the previous design but also positions the simulator for future scalability and innovation. By adopting these principles, the project has gained an increased ability to adapt to evolving requirements, facilitate comprehensive testing, and promote a more organized development process.

3.3. Implementation of Unit Tests

The main part of the project was the implementation of unit test. All the changes described so far were made to facilitate and enable this implementation (e.g. changing the access of the method `purifyCode(...)` of the class `CodeParser.java` from private to public or refactoring the *long* `parseCode(...)` to more smaller methods). The previous sections gave a global overview of these changes. In this section, the focus will be on examining the implementation of the unit tests.

The aim of implementing the unit tests was to develop a comprehensive suite of test cases for rigorous evaluation of each class's functionality. The behaviour of each class was tested against various scenarios to evaluate its correctness, responsiveness, and adherence to its specifications. This was achieved by utilizing essential dependencies as outlined in the table (3.2).

3.3.1. Testing the `CodeParser.java` Class

This class is responsible for testing the three fundamental methods (`purifyCode`, verifyParametersCount and `parseCode`) of the class `CodeParser.java`. In the first hand, the various tests for the `purifyCode` method allowed us to check that the comments and blank spaces in an IJVM code submitted by a user are correctly removed, and also to ensure that the method throws an exception if a user submits a code with recurring labels (e.g. see listing A.2).

Dependency's Name	Version	Role in the project
Assertj	3.24.2	Using AssertJ, assertions that closely resemble the natural language, could be constructed, which made the tests more intuitive and easier to understand
Mockito	5.3.1	This Dependency allowed us to create mock objects that mimic the behaviour of real classes and interfaces. (e.g. the state.java has been mocked to verify its interactions with the class `CPU.java`)
Junit	5.9.2	Junit makes it easy for us to organise testing by using annotations to define test cases.

Table 3.2.: Table of Helpful Dependencies used for Testing the Simulator

In the second hand, some tests have also been implemented to check that the number of bytes parameter for each Mnemonic code is correct. The verifyParametersCount(...) method of the `CodeParser.java` class was extracted from the method `purifyCode(...)`, then updated so that it can be used to check if the number of parameters entered by the user is valid (see listing A.3).

The third and last method tested in this class was `parseCode(...)`, which is responsible for translating a given IJVM code into machine code. It is one of the most critical methods of the simulator. For this method, it was verified that an exception will be raised when a user tries to run a really big IJVM code (with resulting machine code length bigger than 65535 bits). And then, it was also verified that the IJVM code is correctly parsed (see listing. A.4).

3.3.2. Testing the `MALParser.java` Class

The purpose of this test class is to test the behaviour and correctness of the `MALParser.java` class, which has the responsibility for adding new customised MAL program. The two relevant methods to test in this class were `purifyCode(...)` and `parseMal(...)`.

The tests for the `purifyCode()` method included detecting and raising exceptions for scenarios such as recurring labels, mixed known and unknown labels, improper addresses, missing labels or addresses, incorrect jump distances, and more. The tests also verified the

correctness of code purification by comparing the actual purified code, label/address maps, and the expected outcomes (see listing A.5).

However, the `parseMAL(...)` method was tested to ensure correct parsing of microinstructions with various combinations of labels, addresses, and comments. Tests covered scenarios with different addresses, labels, and binary values, ensuring that the method correctly generates the corresponding control memory array with the appropriate values and addresses (see listing A.6).

3.3.3. Testing the `CPU.java` Class

The `CPUTest.java` class serves as a test suite designed to test the functionality and behaviour of the `CPU.java` class. The primary purpose of the `CPUTest.java` class is to ensure that the `CPU.java` class executes instructions correctly and maintains the expected state of registers, memory, and other components throughout its execution. By simulating the execution of a specific bytecode program (see listing 3.5) and carefully comparing expected and actual states, these tests ensure that the CPU operates as expected and maintains the correct state throughout its execution.

Example: In this example (listing 3.6), we write a byte program resulting from the IJVM code (listing 3.5) in the memory methodArea and then execute the method `runCycle()` 19 times and check if the memory and the stack contain the expected values. In the CPUTest class, various aspects of the CPU's behaviour were tested to ensure its correctness. The tests cover a range of scenarios and conditions, examining different components and behaviours of the CPU. Here are some aspects of the CPU's behaviour that were tested:

```
1   BIPUSH 21
2   DUP
3   IADD
4   ISTORE L1
```

Listing 3.5: Example IJVM Program

```
1 private void arrange_cpu_test_Bipush_Dup_Iadd() throws CodeParserException {
2   // let's assume that codeParser is working. because this class has already been tested.
3   CodeParser codeParser = CodeParser.getInstance();
4   short[] methodArea = codeParser.parseCode("""
```

```
                BIPUSH 21
                DUP
                IADD
                ISTORE L1
        """);
  Memory memory = new Memory();
  alu = new ALU();
  state = new State(memory);
  cpu = new CPU(state, alu);
  cpu.writeMethodArea(methodArea);
}

@Test
@DisplayName("Test a Bipush_Dup_Iadd programm and check if after the nineteenth runCycle,
   the memory is correct")
void cpu_Iadd_test_runCycle_19() throws CodeParserException {
  // Arrange
  arrange_cpu_test_Bipush_Dup_Iadd();

  int expected_pc = 7;
  int expected_sp = 132096;
  int expected_lv = 65536;
  int expected_cpp = 0;
  int expected_tos = 0;
  int expected_opc = 0;
  int expected_h = 65536;

  int expected_mpc = 0x100;
  long expected_mir = 70582801L;
  int expected_mbr = Short.MAX_VALUE;
  int expected_mar = 132096;
  int expected_mdr = 0;

  // Act
  IntStream.iterate(0, i -> i + 1).limit(19).forEach(i -> cpu.runCycle());

      // Verify
  memory_assertion(expected_pc,
        expected_sp,
        expected_lv,
        expected_cpp,
        expected_tos,
        expected_opc,
        expected_h,
        expected_mpc,
        expected_mir,
        expected_mbr,
        expected_mar,
        expected_mdr);

  assertThat(state.getClockCounter().get()).isEqualTo(19);
  // at this step, the program should be finished.
  assertThat(cpu.isEndOfExecution()).isTrue();
```

```
57    assertThat(cpu.getStack().get(expected_lv).getValue()).isEqualTo(0x42);
58    assertThat(cpu.getStack().get(expected_sp + 1).getValue()).isEqualTo(0x42);
59    assertThat(cpu.getStack().get(expected_sp + 2).getValue()).isEqualTo(0x21);
60    // this assertion is to check if the stack has exactly two not null elements.
61    assertThat(cpu.getStack().stream().filter(register -> register.getValue() != 0).count()).
       isEqualTo(3);
62  }
```

Listing 3.6: Example Test CPU-Memory Management Lifecycle

Instruction Execution The primary focus of the tests is to verify that the CPU correctly executes a specific bytecode program. This includes the proper execution of instructions like `BIPUSH, DUP` and `IADD`. The tests confirm that the instructions are executed in the correct order and produce the expected results.

Memory Interaction The tests examine the interaction between the CPU and memory. They verify that the CPU reads and writes to memory locations as expected, and that the Memory Address Register (MAR) and Memory Data Register (MDR) are used appropriately.

Clock Counting The tests monitor the clock cycles and validate that the clock counter is incremented appropriately as the CPU executes instructions.

Verification of Memory Contents Throughout the tests, the memory contents at specific memory addresses are verified to match the expected values. This ensures that memory is being updated and accessed properly during execution.

Register State The tests verify the state of various registers within the CPU, such as the program Counter (PC), Stack Pointer (SP), Local Variable (LV), Constant Pool Pointer (CPP), Operand Stack (TOS), and more. The tests ensure that these registers are updated correctly during the execution of the program.

3.3.4. Testing the `ALU.java` Class

The `ALUTest.java` class serves as a suite of tests for the ALU implementation. It validates various aspects of the ALU's behaviour, including output values, flag settings, and specific calculations across a range of input scenarios. Some specific calculation scenarios can be

found in series of tests named `testCalculate...()`, which cover different types of ALU calculations and verify that the output, `Nbit`, and `Zbit` are as expected. These tests evaluate various arithmetic, and logical operations, including addition, subtraction, bitwise AND and OR. For example , the following (example 3.7) shows a test scenario involving the subtraction operation `B - A`, where B is 5 and A is 3 (here the binary representation of the control bits for subtraction is `0d111111`). This assertion checks if the output of the ALU calculation is as expected and check also that the `NBit` and `Zbit` are not set after the subtraction.

```
@Test
public void testCalculateSubtraction_B_Minus_A() {
      alu.calculate((byte) 0b111111, 3, 5);
      assertEquals(2, alu.getOutput());
      assertFalse(alu.getNBit());
      assertFalse(alu.getZBit());
}

@Test // this is a second Test where the value 'true' for Nbit is asserted.
public void testCalculate_Minus_A() {
    alu.calculate((byte) 0b111011, 0b0100, 0b10010);
    assertEquals(alu.getOutput(), (byte)(-4));
    assertTrue(alu.getNBit());
    assertFalse(alu.getZBit());
}
```

Listing 3.7: Example ALU Test Calculate

3.3.5. Testing Other Classes

During the project, some other classes were also tested including `NumericFactory.java` and `MicroinstructionParser.java`. While the class `NumericFactoryTest.java` contains a collection of unit tests designed to assess the functionality and correctness of the `NumericFactory.java` class, the class `MicroinstructionParserTest.java` otherwise was designed to test the functionality of the `MicroinstructionParser.java` class.

Together, these tests evaluate various aspects of the NumericFactory's and the MicroinstructionParser's behaviour related to parsing and formatting numeric values across different radixes (binary, decimal, hexadecimal), and validating the calculation of B-bus values, C-bus values, memory actions, next addresses, jump bits, and ALU calculations.

Chapter 4.

Results and Discussion

This chapter elucidates the outcomes of the efforts made to enhance the Mic-1 Simulator project. A summary of the changes implemented will be presented, highlighting key adjustments across the software. An analysis between the initial and refined states of the project provides insight into the practical improvements achieved followed by a discussion of the implications of refactoring and testing.

4.1. Summary of Changes Made

During the project realization a lot of changes have been made on the old Mic1-Simulator. All those changes had the same purpose, which was to facilitate the understanding of the project and to enable us perform appropriate unit testing for every important class. Firstly, the architectural structure of the project has been changed, in order to have classes with same purpose grouped in small packages. Then the project has been turned into a modular project and tailored gradles to facilitate the dependency management. Finally, every class was cleaned up by removing inappropriate and reorganizing them in newly defined classes, i.e. the `controlMemory`, `memory` objects have been removed from the `cpu.java` class implemented in a new class (`State.java`). The class `State.java`, residing in a new package named state, is now dedicated to manage the state of the simulator during the life-cycle of the application. This package includes some other related classes as well.

Another big change made was in the class `Malparser.java`. In the previous project, it

was only possible to add one microprogram and the address of the first microinstruction was fixed (0xce). This method presented a range of bugs as it was possible to overwrite existing microinstruction in the control store or to carefully choose microinstruction addresses for the microprogram, which is going to be added, so that one line will be overwritten by the other line (see listing 4.1).At first glance, the MAL-code seems to be correct. But a closer look at the control memory (see figure A.1) shows, that the instruction at the address "0x1ce" has been overwritten and replaced by `myIadd4 H = TOS`. Normally it should be `H = TOS; if (z) goto myIadd4; else goto myIadd5` instead. Those bugs have been fixed and the `Malparser.java` class has been updated to ensure that multiple microprograms may be added at same time.

```
1  0x21 MAR = SP = SP - 1; rd
2  myIadd2 H = TOS
3  myIadd3 MDR = TOS = MDR + H; wr;
4  0x1ce H = TOS; if (z) goto myIadd4; else goto myIadd5
5  myIadd4 H = TOS
6  myIadd5 MDR = TOS; wr; goto Main1
```

Listing 4.1: Example Test PurifyCode

Another change concerns the simplification of many methods by means of refactoring, the utilisation of higher order functions, and applaying an updated version of java to make methods compact and more readable.

4.2. Adaptation of the Simulator

In the prior version of the simulator it was only possible to add one MAL program at a time. By the necessity of adding a second MAL program, the user shall first remove the previously added one. A notable disadvantage of this was that there was no way for a user to add two or more microprograms and simulate their execution.

The new version of the simulator enhances the simulators functionality in such a way, that many microprograms may be added at the same time. Instead of a fixed address a dynamic procedure was introduced. For this purpose, a method of the `Malparser.java` class checks for available addresses and dynamically assigns an address to every instruction depending on their syntax. The following passage describes how these method works based on some newly developed microprograms in combination with some microprograms introduced

in ([Loe23] section: “Appendix A. Program”). For clarification of achieved improvements some figures of the control store are also added.

When adding a new “MAL” program the first address of the program is not anymore hard coded like in the previous state of the simulator (the first address of a MAL program was always “0xce”). Instead, free addresses of the control store table are provided by a designed algorithm and allocated to microinstructions. With this implementation, the limit of just one microprogram of maximun 50 lines starting from “0xce” to “0xff” is overcome, and replaced by the possibility to add many microprograms in all free addresses from “0x0” to “0x1ff”. The method `getMemoryAvailableAddresses(...)` in following code snippet 4.2 is the method responsible for finding all the available addresses.

```
1
2  function FindNextAvailableAddressesWithDistance0x100(availableAddresses, labels, distance):
3    availableAddresses512 = FindAvailableAddresses512(labels)
4    for addr in availableAddresses:
5        if (addr + distance) is in availableAddresses and (addr + distance) is in
       availableAddresses512:
6            return addr
7
8  // this method take the values of goto, then check if they're labels. If yes, then it
       calculates their addresses.
9  // if they are addresses, it checks, if they have the distance of 0x100.
10 function CalculateFalseJumpAddress(trueAddress, falseAddress, labels, availableAddresses):
11   labelsCopy = CopyLabelsToLowercase(labels)
12   if IsNotHex(trueAddress) and IsNotHex(falseAddress):
13       if IsNotInLabels(labelsCopy, trueAddress) and IsNotInLabels(labelsCopy, falseAddress):
14           if trueAddress equals falseAddress:
15               throw MALParserException("=jump-error")
16           nextAvailableAddress = FindNextAvailableAddressesWithDistance0x100(
       availableAddresses_512, labelsCopy, 0x100)
17           if nextAvailableAddress is not null:
18               return nextAvailableAddress
19   else if IsHex(trueAddress) and IsHex(falseAddress):
20       if IsNotInCorrectDistanceHex(trueAddress, falseAddress):
21           throw MALParserException("=jump-error")
22   return null
23
24 function getMemoryAvailableAddresses(labels, codeLines):
25   addressesSelectedByTheUser = ExtractAddressesSelectedByUser(codeLines)
26   usedAddresses = ExtractUsedAddresses(labels, addressesSelectedByTheUser)
27   availableAddresses = FindAvailableAddresses(usedAddresses)
28   for each line in codeLines:
29     if line is blank or does not contain space, continue
30     if line contains "else":
31         trueAddress, falseAddress = FindTrueFalseAddresses(line)
```

```
32          falseJumpAddress = CalculateFalseJumpAddress(trueAddress, falseAddress, labels,
    availableAddresses)
33          if falseJumpAddress is not null:
34              trueJumpAddress = CalculateTrueJumpAddress(falseJumpAddress)
35              UpdateLabels(labels, trueAddress, trueJumpAddress, falseAddress,
    falseJumpAddress)
36              RemoveUsedAddresses(availableAddresses, trueJumpAddress, falseJumpAddress)
37  return availableAddresses
```

Listing 4.2: Function Responsible for finding available Addresses

The method `getMemoryAvailableAddresses(...)` (of listing 4.2) comprises the following functions:

- First it takes a list of addresses from 0x0 to 0x1ff
- Then, it removes all the addresses already used by other microprograms from this list.
- At this step, the available addresses can now be used for the new program and the hard-coded addresses will be reserved.
- Now, with the help of the method `calcFalseJumpAddr(...)` and the method `FindNextAvailableAddressesWithDistance0x100()`, the jump addresses for goto labels will be calculated. This step will be skipped if there's any If-condition with a label. The calculated addresses will be also reserved and removed from the list of available addresses.
- The rest of available addresses will now be returned to the calling method and will be attributed to the rest of the microinstructions in the ascending order.

The functionality of the method (cf. listing 4.2), shall be explained by adding the "MAL" microprogram (cf. listing 4.3) to the control store. This program was designed for memory allocation demonstration, only. Absolute program correctness was not intended. The length of the program was chosen for best demonstration of microprogram allocation in view of existing If-conditions (see lines `myTest3...` `myTest13...`) and, three instructions with chosen addresses (0x5, 0x6 and 0x20).

Description of the microprogram (listing 4.3)

In view of the MAL program, it should be mentioned that

- The lines `myTest3...` and `myTest13...` have each "goto" references to two addresses of the instructions starting with labels `myTest8, myTest9, myTest20` and `myTest18`.
- The addresses 0x5, 0x6 and 0x20 have been chosen by the user for some reason.
- `myTest8` should have the exact distance 0x100 to `myTest9` and `myTest20` should also have the exact distance 0x100 to `myTest18`
- Except for the three points listed up, the rest of the instructions can take any free address.

```
1
2   myTest1 MAR = SP - 1;
3   myTest2 H = TOS
4
5   // Has a condition
6   myTest3 SP = SP - 1; rd; if (Z) goto myTest8; else goto myTest9
7
8   myTest4 H = TOS + 1
9   myTest5 MAR = SP = TOS;
10  myTest6 H = 1
11  myTest7 MAR = SP = SP - 1; wr
12  myTest8 H = TOS
13  myTest9 MAR = SP = SP - 1; fetch
14  myTest10 H = MDR
15  myTest11 MAR = SP = SP - 1; rd
16  myTest12 TOS = H
17
18  // Has a condition
19  myTest13 Z = OPC - H; if (Z) goto myTest20; else goto myTest18
20
21  // Fixed chosen Address
22  0x20 MAR = SP = MDR = SP - 1; rd
23
24  myTest14 H = TOS = MDR
25  myTest15 SP = MDR; wr
26  myTest16 H = TOS = 1; rd
27  myTest17 MAR = 1; rd
28  myTest18 H = TOS = TOS -1
29
```

```
30  // Fixed chosen Addresses
31  0x5 MAR = SP = SP +1
32  0x6 H = TOS = TOS +1
33
34  myTest20 SP = SP; fetch
35  myTest22 MAR = TOS
36  myTest23 MAR = SP = ; rd
37  myTest24 H = TOS = MDR
38  myTest25 MAR = SP = MDR; wr
39  myTest26 H = TOS; goto Main1
```

Listing 4.3: Long MAL Program

There are three pictures provided to visualize how addresses can be dynamically selected for every microinstruction:

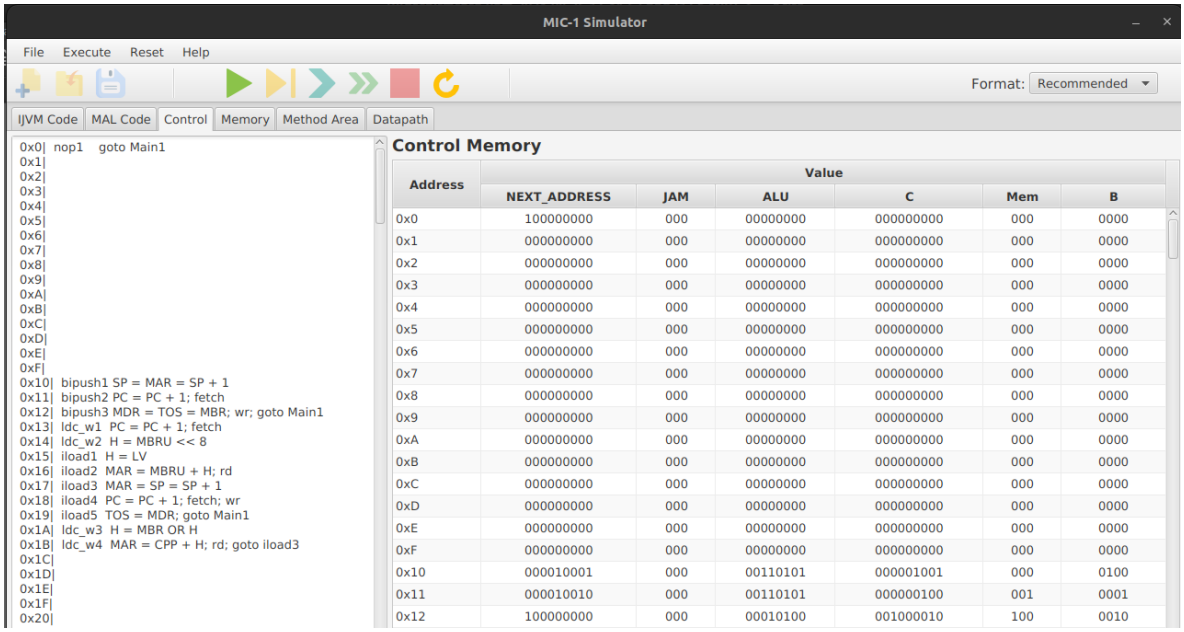
- The first picture (figure 4.1a) shows the state of the control store with hard coded initial MAL programs.
- The second picture (figure 4.1b) shows the state of the control store after adding the MAL program (listing 4.3).
- The red areas indicate the microinstructions with jump addresses, the green areas show the microinstructions with hard coded addresses and the yellow areas represent the rest of the microinstructions.

4.3. Overview of Test Coverage

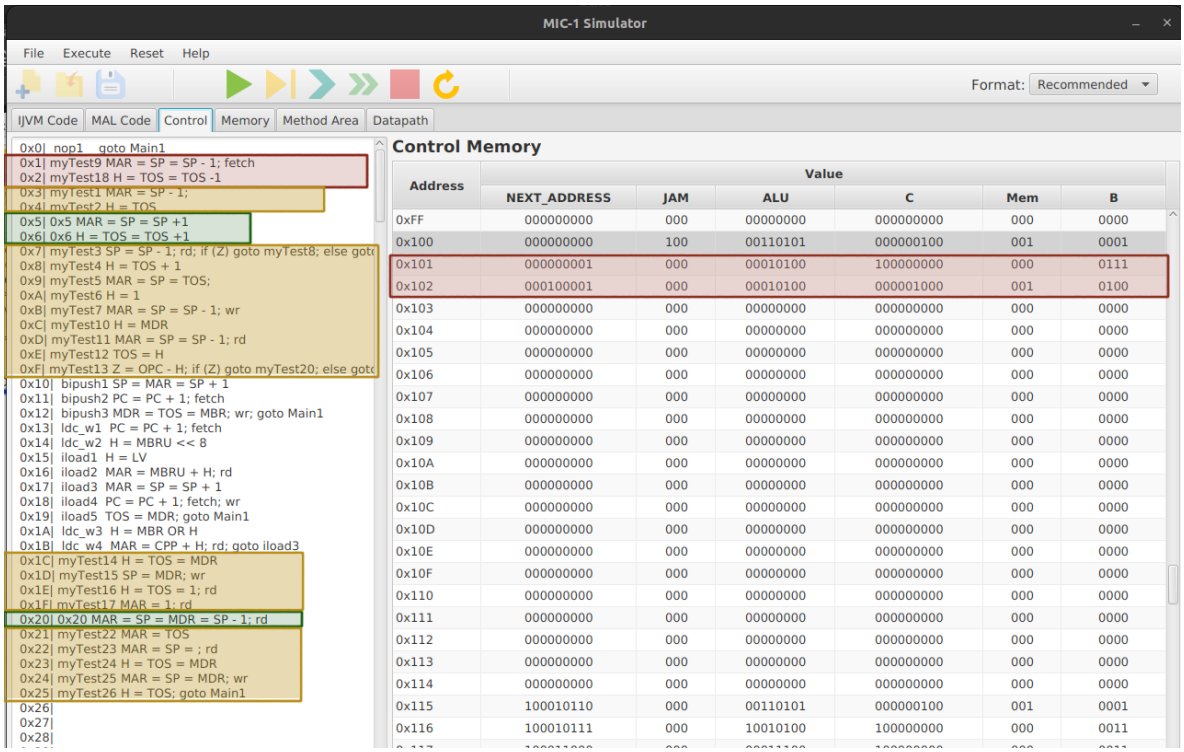
Since the main purpose of the project was the implementation of automatic tests, a summary of the test's coverage shall be given.

As refactoring of the prior simulator version in view of some methods were carried out and some bugs were solved before testing, the performance of the original version can't be tracked anymore.

The scope of tests executed with the new version comprised 228 runs, thereof two failed and two were ignored. The failed methods belong to the class `MalParserTest.java`. The first method `purifyCode_04()` failed because of the fact that the parser can parse



(a) Initial Value of the Control Store before Adding Self-Microprogam



(b) Add a Microprogam with If-Conditions and fixed chosen Addresses

Figure 4.1.: Initial State of Control Store 4.1a and State after adding the microprogram (listing 4.3).

the eguality sign **"="** as label instead of throwing exceptions. And the second method `parseMal_04()` failed because of the wrong interpretation of the microinstruction `Z = OPC - H;`.

The two ignored methods are caused by the fact that there are small mismatches in the code logic making it not easy to perform some tests correctly, i.e. the method `check_if_getStringValue32_returns_correct_values()` from the class `NumericFactoryTest.java` (see listing 4.4) was skipped because this class is a static class. Changing the radix value before performing that test would affect other methods. A solution, for this would be either to change the radix value in every single test or refactor the class such that the radix is handed over as parameter for every method call.

The following tables show respectively the percentage of test coverage for the packages (table 4.1) and for the classes (table 4.3). The noticable difference of test percentage results from the fact that some packages weren't tested, e.g. the package sample in the table (4.1), as the main focus was on the uni tests and not on the UI & integration tests. Furthermore, methods of classes `CUP.java` (package backend) or `MalParser.java` (package parser) are less than 80% covered. This effect results from the fact that some methods aren't testable or in doesn't make much sense to test them. For example, the method `toString()` and multiples `String get...Tooltip()` methods residing in the class `CPU.java` serve to describe the registers and don't have any effect on the state of the simulator. It is also noticable that the methods of state package aren't 100% covered. This also results from the fact that some existing data classes in this package don't neccessarily require any tests as they don't have any logic. And the table (4.3) shows a summary with a number of tests done per classes.

Result of Test Coverage. (Package Overview)			
Package Name	Class, %	Method, %	Line ,%
backend	50%	62%	49%
exceptionHandler	100%	100%	100%
parser	83%	84%	79%
sample	14%	0%	0%
state	100%	71%	83%

Table 4.1.: Package Overview

Result of Test Coverage. (Classes Overview)				
Package	Name	Class, %	Method, %	Line ,%
backend	Alu	100%	100%	100%
backend	AluInterface	100%	100%	100%
backend	CPU	100%	62%	41%
backend	NumericFactory	100%	80%	81%
parser	CoderParser	100%	93%	79%
parser	InstructionParser	100%	89%	88%
parser	MalParser	100%	77%	84%
parser	MicroinstructionParser	100%	100%	95%
state	BitValues	100%	100%	100%
state	Memory	100%	87%	92%
state	State	100%	86%	90%
state	Stateable	100%	100%	100%

Table 4.2.: Classes Overview

Test Summary: Numer of Tests Per Classes				
Class	Tests	Failures	Ignored	Success rate, %
AluTest	23	0	0	100%
CPUTest	39	0	0	100%
NumericFactoryTest	10	0	1	100%
CoderParserTest	67	0	0	100%
MalParserTest	22	2	0	90%
MicroinstructionParserTest	60	0	1	100%
MemoryTest	1	0	0	100%
ShifterTest	5	0	0	100%

Table 4.3.: Test Summary

```
1
2  /* Disabled Tests */
3
4  @DisplayName("Test if the method calcNextAddress return the correct value for the
       microinstruction with goto100")
5  @Disabled("If I change the address value to 0x100, the test will fail. Because the addresses
       of the map are not with ''0x''")
6  @Test
7  void calcNextAddress0x100() throws MALParserException {
8      // Arrange
9      String microinstruction = "pc=pc+1;goto100;";
10     long expected = 0x800000000L;
11     // Act
12     calcNextAddressAssertion(microinstruction, expected);
13 }
14
15
16 @Disabled("This in not working because the class is static and the Set radix in other Test
       cases affect the radix of this test")
17 @Test
18 void check_if_getStringValue32_returns_correct_values() {
19   // verify
20   assertThat(NumericFactory.getStringValue32(10)).isEqualTo("0xA");
21   assertThat(NumericFactory.getStringValue32(15)).isEqualTo("0xF");
22   assertThat(NumericFactory.getStringValue32(31)).isEqualTo("0x1F");
23   assertThat(NumericFactory.getStringValue32(63)).isEqualTo("0x3F");
24   assertThat(NumericFactory.getStringValue32(123)).isEqualTo("0x7B");
25   assertThat(NumericFactory.getStringValue32(255)).isEqualTo("0xFF");
26   assertThat(NumericFactory.getStringValue32(1023)).isEqualTo("0x3FF");
27 }
28
29 /* Failed Tests */
30
31 @DisplayName("purifyCode should throw an exception when when the label or address is missing
       ")
32 @Test
33 void purifyCode_04() {
34   // setup
35   String code = """
36             = SP = SP - 1; rd
37           """;
38   String exception = "=missing-label";
39   // verify
40   assertPurifyCodeThrowsException(exception, code);
41 }
42
43 @DisplayName("parseMal should return array of long values when code is correct, contains
       labels and addresses are selected correctly")
44 @Test
45 void parseMal_04() throws MALParserException {
46   // setup
47   List<String[]> micro_instructions = generateMicroInstructionWithBinaryValue();
48   String code =
```

```
49    "0x20 " + micro_instructions.get(0)[0] +" \n"       // SP = MAR = SP + 1
50      + "label1 " + micro_instructions.get(1)[0] +" \n" // PC = PC + 1; fetch
51      + "label2 " + micro_instructions.get(8)[0] +" \n" // H = MBR OR H
52      + "label3 " + micro_instructions.get(9)[0] +" \n" // Z = OPC - H;
53      + "0x22 " + micro_instructions.get(10)[0] +" \n"; // MAR = CPP + H; rd; goto iload3
54  int addrIload3 = 0x17;
55  Map<Integer, Long> expectedArrayPart = new HashMap<>();
56  expectedArrayPart.put(0x20, toLongAddrValue(0x1) + Long.parseLong(micro_instructions.get
      (0)[1], 2)); // 000 00110101 000001001 000 0100
57  expectedArrayPart.put(0x1, toLongAddrValue(0x2) + Long.parseLong(micro_instructions.get(1)
      [1], 2)); // 000 00110101 000000100 001 0001
58  expectedArrayPart.put(0x2, toLongAddrValue(0x3) + Long.parseLong(micro_instructions.get(8)
      [1], 2)); // 000 00011100 100000000 000 0010
59  expectedArrayPart.put(0x3, toLongAddrValue(0x22) + Long.parseLong(micro_instructions.get
      (9)[1], 2)); // 001 00111111 000000000 000 1000
60  expectedArrayPart.put(0x22, toLongAddrValue(addrIload3) + Long.parseLong(
      micro_instructions.get(10)[1], 2)); // 000 00111100 000000001 010 0110
61
62  // verify
63  assertParseMAL(code, expectedArrayPart);
64  }
```

Listing 4.4: Failed and Disabled Tests

Chapter 5.

Conclusion

This project aimed to expand a simulator for a microprogrammable processor by integrating automated testing functionalities. Throughout the work, several critical aspects that were integral to achieving the project's objectives have been addressed. Starting with refactoring the existing codebase, then transforming its prior structure to a more modular and maintainable architecture. The purpose of this refactoring was to facilitate the code comprehension, and to provide a good approach to implement uni-testing.

5.1. Recap of Achievements

The most significant contribution of this project lies in the implementation of unit tests. By introducing a suite of tests for essential methods, we not only ensured the correctness of the simulator's behavior but also established a foundation for future developments. Furthermore, the introduction of dependency injection and the adoption of design patterns demonstrated the commitment to contemporary software design paradigms.

5.2. Future Directions and Further Enhancements

Compared with the previous state, the software at this stage has taken a big step forward with the restructuring of classes, the implementation of automatic tests and the extension of the

number of microprograms that can be added at once to the `control memory`. Nevertheless, there are some desirable features that couldn't be incorporated due to time constraints during this project, but they may be added in the future.

A first somewhat more elaborate extension would be to call methods via `INVOKEVIRTUAL` and return via `IRETURN`.

A second future implementation could be to allow a user to remove a desired microprogram from the list of added microprogram without the need of restarting the simulator.

Another future extension could be the option of saving the user added microprograms and restoring them each time the application is started, the need to rewrite or recompile them again.

Appendix A.

Additional Illustrations

A.1. Figures

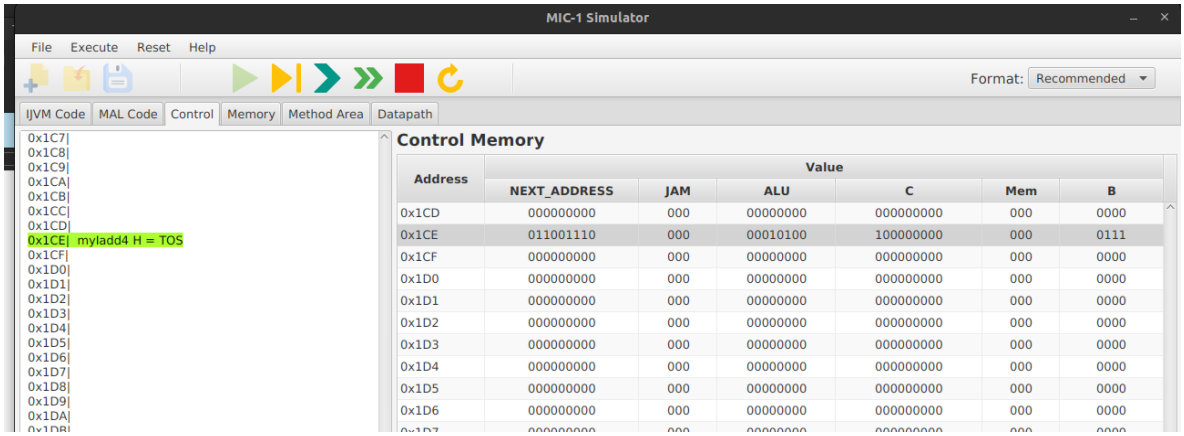


Figure A.1.: Control Memory: Instruction *H = TOS; if (z) goto myIadd4; else goto myIadd5* replaced by *myIadd4 H = TOS* at line 0x1ce.

A.2. Code Snippets and Examples

```
1
2 plugins {
3     id 'java'
4     id 'application'
5     id 'org.openjfx.javafxplugin' version '0.0.13'
6 }
7
```

```
group 'de.hhu'
version '1.0-SNAPSHOT'

java {
    sourceCompatibility = '17'
    targetCompatibility = '17'
}

repositories {
    mavenCentral()
}

ext {
    junitVersion = '5.9.2'
}

tasks.withType(JavaCompile).configureEach {
    options.encoding = 'UTF-8'
}

application {
    mainModule = 'de.hhu.MIC_1_Simulator'
    mainClass = 'de.hhu.MIC_1_Simulator.Main'
}

javafx {
    version = '18.0.2'
    modules = ['javafx.controls', 'javafx.fxml']
}

dependencies {

    implementation 'org.openjfx:javafx-controls:21-ea+17'
    implementation 'org.apache.logging.log4j:log4j-api:2.20.0'
    implementation 'org.apache.logging.log4j:log4j-core:2.20.0'

    testImplementation("org.assertj:assertj-core:3.24.2")
    testImplementation("org.mockito:mockito-core:5.3.1")

    testImplementation("org.junit.jupiter:junit-jupiter-api:${junitVersion}")
    testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine:${junitVersion}")
}

test {
    useJUnitPlatform()

    testLogging {
        events "passed"
    }
}

jar {
    manifest {
```

```
        attributes("Implementation-Title": 'MIC_1_Simulator',
                "Implementation-Version": '1.0-SNAPSHOT',
                "Main-Class": 'de/hhu/MIC_1_Simulator/Main')
    }
}

run {
    doFirst {
        jvmArgs = [
                '--module-path', classpath.asPath,
                '--add-modules', 'javafx.controls',
                '--add-modules', 'javafx.fxml',
                '--add-modules', 'javafx.graphics',
        ]
    }
}
```

Listing A.1: build.gradle file

```
@Test
@DisplayName("Purify code should add labels in the labels Map")
void purifyCode_4() throws CodeParserException {
    // Arrange
    Map<String, Short> labels = new HashMap<>();
    String code = """
            // this comment should be removed and the method purifycode should return an array one empty string.
            L1: BIPUSH               21
            """;
    String[] expected = {"", "BIPUSH 21"};

    // Act
    String[] actual = codeParser.purifyCode(code, labels);

    //Assert
    assertArrayEquals(actual, expected);
    assertThat(labels).containsKey("L1");
}

@Test
@DisplayName("Purify code should throw CodeParserException when a label is used many times")
void purifyCode_5() {
    // Arrange
    Map<String, Short> labels = new HashMap<>();
    String code = """
            L1: BIPUSH               21
            L1: IADD
            """;
    String expectedMessage = "2=recurring-label";

    //Assert
    assertThrows(CodeParserException.class, () -> codeParser.purifyCode(code, labels), expectedMessage);
```

```
}
```

Listing A.2: Exemple Test PurifyCode CodeParser.java

```
    @Test
    @DisplayName("Iload should return missing-arg when no parameter is passed")
    void verifyParametersCount_Iload_1() {
        // Arrange
        assertMissingParameterException(new String[]{"ILOAD"});
    }

    @Test
    @DisplayName("Iload should return too-many-args when too many parameters are passed")
    void verifyParametersCount_Iload_2() {
        // Arrange
        assertTooManyParameterException(new String[]{"ILOAD", "1", "2"});
    }

    @Test
    @DisplayName("Iload should return null when the correct number of parameters is passed")
    void verifyParametersCount_Iload_3() {
        // Arrange
        assertNull(new String[]{"ILOAD", "1"});
    }
```

Listing A.3: Exemple Test verifyParametersCount

```
@Test
@DisplayName("the method parseCode should correctly parse Mnemonics instruction with back
    jumps")
void parseCode_10() throws CodeParserException {
    // Arrange
    String code = """
        bipush 0
        istore 2
        bipush 0
        istore 3
        l0: bipush 0
        istore 1
        iload 3
        bipush 1
        iadd
        istore 3
        iload 3
        bipush 5
        if_icmpeq l2
        l1: iload 2
        bipush 1
        iadd
        istore 2
        iload 1
        bipush 1
        iadd
```

```
        istore 1
        iload 1
        bipush 5
        if_icmpeq l0
        goto l1
        l2:
        """;

    List<Short> expected = List.of(
        (short) 0x10, (short) 0x00, (short) 0x36, (short) 0x02, (short) 0x10,
        (short) 0x00, (short) 0x36, (short) 0x03, (short) 0x10, (short) 0x00,
        (short) 0x36, (short) 0x01, (short) 0x15, (short) 0x03, (short) 0x10,
        (short) 0x01, (short) 0x60, (short) 0x36, (short) 0x03, (short) 0x15,
        (short) 0x03, (short) 0x10, (short) 0x05, (short) 0x9f, (short) 0x00,
        (short) 0x1b, (short) 0x15, (short) 0x02, (short) 0x10, (short) 0x01,
        (short) 0x60, (short) 0x36, (short) 0x02, (short) 0x15, (short) 0x01,
        (short) 0x10, (short) 0x01, (short) 0x60, (short) 0x36, (short) 0x01,
        (short) 0x15, (short) 0x01, (short) 0x10, (short) 0x05, (short) 0x9f,
        (short)(byte) 0xff, (short)(byte) 0xdc, (short) 0xa7, (short)(byte) 0xff, (short)(
    byte) 0xeb);

    //Assert
    assertThat(codeParser.parseCode(code)).containsExactly(parseMachineCodeListToByteArray(
    expected));
}
```

Listing A.4: Exemple Test parseCode

```
@DisplayName("purifyCode should throw =jump-error Exception when distance between address
    is not 0x100 and first address is higher than second address")
@Test
void purifyCode_06() {
    // setup
    String code = """
            0x1 H = TOS; if (z) goto 0x20; else goto 0x120
            """;
    String exception = "=jump-error";
    // verify
    assertPurifyCodeThrowsException(exception, code);
}

@DisplayName("purifyCode should return array of code when code is valid and there are valid
     labels")
@Test
void purifyCode_14() throws MALParserException {
     // setup
     String code = """
             label1 MAR = SP = SP - 1; rd
             label2 MAR = SP = SP - 1; rd
             """;
     String[] expectedCode = new String[]{"label1 MAR=SP=SP-1;rd", "label2 MAR=SP=SP-1;rd"};
     Map<String, Integer> expectedAddresses = new HashMap<>();
     expectedAddresses.put("label1", 0x1);
     expectedAddresses.put("label2", 0x2);
```

```
25      Map<String, String> expectedNextLine = new HashMap<>();
26      expectedNextLine.put("first", "label1");
27      expectedNextLine.put("label1", "label2");
28      Map<String, String> expectedLabels = new HashMap<>(labels);
29      expectedLabels.put("label1", "1");
30      expectedLabels.put("label2", "2");
31      // verify
32      assertPurifyCode(code, expectedCode, expectedAddresses, expectedNextLine,
    expectedLabels);
33  }
```

Listing A.5: Exemple Test PurifyCode Malparser.java

```
1   List<String[]> generateMicroInstructionWithBinaryValue() {
2       return Arrays.stream("""
3               0x10 | SP = MAR = SP + 1 | 000 00110101 000001001 000 0100
4               0x11 | PC = PC + 1; fetch | 000 00110101 000000100 001 0001
5               0x12 | MDR = TOS = MBR; wr; goto Main1 | 000 00010100 001000010 100 0010
6               0x14 | H = MBRU << 8 | 000 10010100 100000000 000 0011
7               0x15 | H = LV | 000 00010100 100000000 000 0101
8               0x16 | MAR = MBRU + H; rd | 000 00111100 000000001 010 0011
9               0x17 | MAR = SP = SP + 1 | 000 00110101 000001001 000 0100
10              0x19 | TOS = MDR; goto Main1 | 000 00010100 001000000 000 0000
11              0x1a | H = MBR OR H | 000 00011100 100000000 000 0010
12              0xa6 | Z = OPC - H; if (Z) goto T; else goto F | 001 00111111 000000000 000
    1000
13              0x1b | MAR = CPP + H; rd; goto iload3 | 000 00111100 000000001 010 0110
14              """.split("\n"))
15              .map(line -> line.split("\\|"))
16              .map(line -> new String[]{line[1].trim(), (line[2].replaceAll("\\s+", ""))})
    .toList();
17  }
18
19  private void assertParseMAL(String code, Map<Integer, Long> expectedCode) throws
    MALParserException {
20      // exercise
21      long[] controlMemory = FileParser.getControlMemory();
22      expectedCode.keySet().forEach(key -> controlMemory[key] = expectedCode.get(key));
23
24      // verify
25      assertArrayEquals(controlMemory, malParser.parseMAL(code));
26  }
27
28  @DisplayName("parseMal should return array of long values when code is correct, contains
     labels and addresses are selected correctly")
29  @Test
30  void parseMal_04() throws MALParserException {
31      // setup
32      List<String[]> micro_instructions = generateMicroInstructionWithBinaryValue();
33      String code =
34              "0x20 " + micro_instructions.get(0)[0] +" \n"
35                      + "label1 " + micro_instructions.get(1)[0] +" \n"
36                      + "label2 " + micro_instructions.get(8)[0] +" \n"
37                      + "label3 " + micro_instructions.get(9)[0] +" \n"
```

```
38                          + "0x22 " + micro_instructions.get(10)[0] +" \n";
39          int addrIload3 = 0x17;
40          Map<Integer, Long> expectedArrayPart = new HashMap<>();
41          expectedArrayPart.put(0x20, toLongAddrValue(0x1) + Long.parseLong(micro_instructions
    .get(0)[1], 2));
42          expectedArrayPart.put(0x1, toLongAddrValue(0x2) + Long.parseLong(micro_instructions.
    get(1)[1], 2));
43          expectedArrayPart.put(0x2, toLongAddrValue(0x3) + Long.parseLong(micro_instructions.
    get(8)[1], 2));
44          expectedArrayPart.put(0x3, toLongAddrValue(0x22) + Long.parseLong(micro_instructions
    .get(9)[1], 2));
45          expectedArrayPart.put(0x22, toLongAddrValue(addrIload3) + Long.parseLong(
    micro_instructions.get(10)[1], 2));
46
47          // verify
48          assertParseMAL(code, expectedArrayPart);
49      }
```

Listing A.6: Exemple Test parSeMal

A.3. Prompts

Since I had the assistance of software during the course of this thesis, including Deepl writing for automatic grammar correction, Duden Mentor for grammatical correction, and chat GPT for guidance and assistance with sentence restructuring or word choice, I deemed it unnecessary to include all of the prompts. I have only provided some examples of such prompts, as shown in the tables below.

Prompt	Result
what is the definition of Mockito rewrite this:The purpose of our project involved extending a simulator for a microprogrammable processor by incorporating automated testing capabilities	Long text to define Mockito and exemples about how to use it. Our project aimed to expand a simulator for a microprogrammable processor by integrating automated testing functionalities.

Table A.1.: Chat GPT: Helping Tool use to get some help and definitions.

Prompt	Result
e.g. it was possible to replace the instruction of any "MAL" program just by choosing	For example, it was possible to replace the instruction of any "MAL" program simply by selecting a wrong address.

Table A.2.: Deepl Write: Helping Tool to improve my writing.

Prompt	Result
Au terme de notre tour d'horizon des méthodologies de développement et de test de logiciels, nous allons maintenant nous concentrer sur l'examen des efforts effectués par les précédents developpeurs du Mic-1 simulateur	Having completed our overview of software development and testing methodologies, we will now focus on examining the efforts made by the previous developers of the Mic-1 simulator.

Table A.3.: Deepl Translator: Helping Tool to translate my text from French to English

Bibliography

[AU19] A Anand and A Uddin. “Importance of software testing in the process of software development”. In: *International Journal for Scientfic Research and Development* 12.6 (2019).

[BO94] J.M. Bieman and L.M. Ott. “Measuring functional cohesion”. In: *IEEE Transactions on Software Engineering* 20.8 (1994), pp. 644–657. DOI: 10.1109/32.310673.

[Doo17] John F Dooley. *Software Development, Design and Coding With Patterns, Debugging, Unit Testing, and Refactoring*. Springer, 2017.

[IBM] IBM. *What is the cost of not testing?* https://www.ibm.com/docs/en/wsfz-and-o/1.1?topic=wsim-what-is-cost-not-testing. [Online; accessed 04-Sept-2023].

[Kat] Katalon. *Unit Testing vs. Functional Testing: An In-Depth Comparison*. https://katalon.com/resources-center/blog/unit-testing-vs-functional-testing. [Online; accessed 03-Sept-2023].

[Lit79] Bev Littlewood. “How to Measure Software Reliability and How Not To”. In: *IEEE Transactions on Reliability* R-28.2 (1979), pp. 103–110. DOI: 10.1109/TR.1979.5220510.

[Loe23] Annika Loettgen. *Simulation eines mikroprogrammierbarenProzessors*. 2023.

[MT04] T. Mens and T. Tourwe. “A survey of software refactoring”. In: *IEEE Transactions on Software Engineering* 30.2 (2004), pp. 126–139. DOI: 10.1109/TSE.2004.1265817.

[Tan01] Andrew S Tanenbaum. *Structured Computer Organization*. 2001.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 14. September 2023 Loïc Duval Lekeuagni Nguegang