uc3m | Universidad Carlos III de Madrid

Bachelor's degree in Computer Science and Engineering

Bachelor Thesis

"Analysis, Design and Implementation of a Didactic and Generic Assembly Language Simulator"

Author

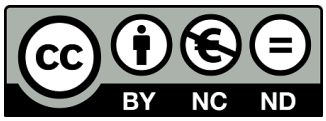
Luis Daniel Casais Mezquida

Tutor

Alejandro Calderón Mateos

Leganés, Madrid, Spain

June 2024



Nor again is there anyone who loves or pursues or desires to obtain pain of itself, because it is pain, but because occasionally circumstances occur in which toil and pain can procure him some great pleasure.

— Cicero, *de Finibus Bonorum et Malorum*

ACKNOWLEDGEMENTS

I would like to start by thanking my parents. It is thanks to them that I managed to be here right now, writing my bachelor's thesis. Not only because they "paid for the damned thing" (and more than they should've, thanks to Algebra and Physics), but because they supported, educated, and guided me through my whole life. I wouldn't be the person that I am today without them.

The rest of the support has come from my friends, and specially my girlfriend. Here, I make a rather liberal use of the word 'support', as it has mainly consisted of me yapping about "computer stuff" and them nodding politely. I will probably never know what (low) percentage of the things I have told them are still in their memory, but I'm thankful to them for entertaining my love for teaching and sharing knowledge. I know they love me, and I love them too.

A special consideration is deserved for the two people that helped make this project (me finally graduating) possible: my tutor, Alex, and Jose Antonio.

I had to 'stalk' Alex to meet with him the first time (and some subsequent times), but he was generous enough not only to accept me, but also to propose ideas for the thesis (as I had none). Despite his *very* full schedule, he accepted scheduling weekly tutoring sessions for the report, as the deadline was rapidly approaching, and I was very lost.

Jose Antonio has been the greatest help for me, despite being a junior student. He has helped me navigate the C++ language, which is not an easy task, and was a great partner to share the frustrations of this great language. He also contributed some ideas to the project, such as using a Lisp-like language to define the instructions. I think we can all expect great things from him.

I'd also like to mention professor Jose Daniel, who introduced me to the world of C++ and pointed me to Alex for project ideas, after I initially asked him.

Another group of people that deserves an acknowledgement is *el* GUL. I entered the association only two years ago, but I am now its president. I have to be thankful not only of the people (Danié, Banga, Alex, Jose Antonio, Jorge, Cía, and the rest of members) for teaching me a lot of extremely useful stuff, transforming me into a Linux believer, and keeping me company; but also for offering me the opportunity of organizing and presenting talks and workshops, and sharing my knowledge.

There are many other people that got me through the degree, even if their contributions were small. This goes for all of my excellent professors, who gave me the tools that I will surely make use of in the future, and my colleagues, who helped and inspired

me throughout the years. This is getting too long, so I hope they'll forgive me for not explicitly mentioning them.

Finally, I'd like to thank the open-source community. Most of my involvement with them may have been passive, using excellent software and reading documentation, tutorials, and GitHub issues, but I have to admire their dedication for making the world a better place, one line of code at a time. I'll be sure to follow their steps.

ABSTRACT

There are not many didactic and generic assembly language simulators that can be used to introduce people to assembly language programming and Instruction Set Architecture (ISA) design.

Due to the current geopolitical and commercial climate, countries and corporations have started investing in the manufacturation of their own chips, and open and modular standards, such RISC-V, act as a base for designing and implementing new ISAs.

Our proposal aims to foment ISA design and assembly programming, by providing a simple and easy to understand tool for anyone to start exploring different instruction sets, while also aiding people with the sometimes difficult process of learning assembly language programming.

This project presents a simple, open-source, and intuitive simulator, focused on providing the user an intuitive knowledge of ISA design and assembly programming. Our simulator allows users to create their own ISAs, by defining instructions in a simple Lisp-like language, and to interpret any assembly-like program using that definition.

Keywords: Assembly • Simulation • ISA • Interpreter

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

This first chapter briefly presents the project: why it came to be (Section 1.1, *Motivation*), the objectives it strives to fulfill (Section 1.2, *Objectives*), and a description of the overall document structure (Section 1.3, *Document Structure*).

1.1. Motivation

Programming is the process of expressing abstract ideas and models within a language. These languages can be either high-level languages, with more abstractions, or low-level languages, with few abstractions; but for any language to work for a specific processor, that processor must be able to interpret and execute it. Low-level languages are built on a set of instructions, each one executing a single 'atomic' function on several (or none) defined data, providing the 'benefit' of having no context, which simplifies the electronics. This also makes them more flexible and robust, which is why high-level languages are built upon these low-level languages, by being compiled (translated) from high-level to low-level.

Specifically, the languages that processors interpret are called 'assembly languages'. In order to get the maximum performance from a processor, using them effectively is essential, as they give the programmer the maximum control over the hardware. Due to the fact that, as stated before, high-level languages are built on top of assembly languages, understanding the latter is also necessary for not only creating the former, but also for effectively utilizing them, as different abstractions might affect what the processor will actually do.

For all these reasons, amongst others, learning how assembly languages work is required for any computer science student, or any software engineer, given that it offers a better understanding of the underlying architecture, therefore enabling them to more effi-

ciently use that architecture, but also enabling them to better express those abstract ideas in terms that the processor will understand.

Moreover, it is not worth learning one specific assembly language for one specific architecture, not only because architectures change with time and the principles and concepts behind those specific languages don't, but because understanding those principles allows a person to abstract from the specific details of the implementation, transferring knowledge 'horizontally' to another assembly languages and 'vertically' to the high-level languages. As a professor from the University of Iowa put it almost forty years ago, "[a] student who learns to compare and contrast machines based on a generic model and who understands the relationship of higher level ideas to the generic machine will be better prepared to function for a longer period of time" [1].

Unfortunately, the act of learning assembly languages has been proven difficult, as the low-level instructions make it difficult for the abstract human brain to adapt to. There have been several approaches used in order to enhance the learning, such as self-explaining and the use of architecture diagrams [2], but here we propose the 'simulator' approach: allowing the user to execute instructions step by step and to see state of the different subcomponents of the virtual computer 'in real time'. This, added to the more 'generic' approach to understanding the language and its instructions, should allow the user to perfectly understand the current state of the processor (*what's happening*), and the effects of each instruction (*what happens next*). This is similar to some debugging techniques used in some high-level languages, which have been proven to improve programming skills [3].

Lastly, we are firm believers in Free Open Source Software (FOSS), due to the fact that anyone can see the source code, modify it, learn from it, and contribute in order to improve it. We also consider important that the implementation is flexible enough to be easily ported to many devices: desktop devices, embedded devices, web applications, etc.

In this thesis, we aim to create an open-source generic assembly language simulator focused on helping the user learn and understand assembly languages, not only as a programming language, but also enabling the user to easily define and try their own instructions and combinations of them. We also aim to implement a simulator that can be used in many situations, without sacrificing performance.

1.2. Objectives

The main objective of this project is to design and develop a simulator that, unlike existing ones, enables the user to define their own sets of assembly language instructions in a simple, generic, and easy to understand language, so the user can fully understand what each instruction does.

The secondary objectives, derived from the main objective, are as follows:

- **O1:** Simulate a simple and generic computer: memory, registers, and a processor that executes a simple programming language.
- **O2:** Define a simple programming language that allows to program the different components of the simulated computer.
- **O3:** Allow the user to display in each clock cycle the status of each component of the simulated computer.
- **O4:** Allow the user to execute any program using their defined assembly language.
- **O5:** Create an implementation that allows the simulator to be used in many environments, without sacrificing performance.

1.3. Document Structure

The document contains the following chapters:

- Chapter 1, *Introduction*, briefly presents the project, its motivations and objectives, and a description of the contents of the document.
- Chapter 2, *State of the Art*, includes a description of the current stage of assembly language simulators, providing examples and comparing them with our proposal.
- Chapter 3, *Analysis*, presents the project and establishes its requirements and use cases.
- Chapter 4, *Design*, details the design process and the system's architecture, including all of its components.
- Chapter 5, *Implementation and Deployment*, includes the implementation details of the main parts of the developed software, and the necessary steps for its deployment.
- Chapter 6, *Verification and Validation*, details the verification and validation processes of the system and its objectives, including test cases.
- Chapter 7, *Project Plan*, presents the concepts related to the followed planning, and breaks down all project costs. It also discusses the regulatory framework and socio-economic environment that applies to the project.
- Chapter 8, *Conclusions and Future Work*, highlights the contributions of the project, discusses the overall conclusions, and presents what future work could be done in order to improve the system.
- Appendix A, *User manual*, provides the user manual for the software.

CHAPTER 2

STATE OF THE ART

This chapter presents the current stage of the technologies related to the project, specifically the different kinds of simulators that are currently present in the market (Section 2.1, *Assembly Simulators*), both specific to one architecture (Subsection 2.1.1, *Specific Simulators*) and architecture-agnostic simulators (Subsection 2.1.2, *Generic Simulators*).

Finally, it includes a comparison between all the mentioned simulators and the proposed application (Section 2.2, *Comparison*).

2.1. Assembly Simulators

An 'assembly simulator' is a CPU simulator that enables the user to program it through an assembly language, and it usually has an educational purpose: either to aid the user in learning the language, or as an exercise for the programmer in order to give them a better understanding of CPU architecture and software development.

These simulators typically offer an interface that allows the user to execute the program step-by-step and see the current state of the simulated computer at each step.

2.1.1. Specific Simulators

The vast majority of simulators that can be found today focus on emulating a specific Instruction Set Architecture (ISA), which can range from simple 8-bit microprocessors like the Intel 8080 [4] to architectures that are used today, like ARM Thumb [5].

Here, two examples are provided: Kite and ARMLite.

Kite

Kite [6] is a simulator that models a five-stage pipeline RISC-V CPU, based on the model described in *Computer organization and design: the hardware/software interface (RISC-V edition)*, by D. Patterson and J. Hennessy [7], and implemented in C/C++. It was developed in 2019 in order to provide students of Yonsei University (Seoul, Korea) with an easy-to use simulator to accompany follow its Computer Architecture course.

It incorporates advanced features derived from a pipeline architecture, such as instruction dependency detection and pipeline stalls, among others. These 'under the hood' features offer a better understanding of the underlying concepts of computer architecture.

The simulator consists of an executable with a Command Line Interface (CLI) that takes three input files: the program's code, the register's state, and the data memory's state. It loads the state of registers and data memory, executes the program's code and saves the state to the specified files, after the execution, printing some statistics (Figure 2.1). It also implements a debug mode that prints the state of each instruction on the pipeline, each clock cycle.

```
$ ./kite program_code

************************************************************
* Kite: Architecture Simulator for RISC-V Instruction Set  *
* Developed by William J. Song                             *
* Computer Architecture and Systems Lab, Yonsei University *
* Version: 1.12                                            *
************************************************************

Start running ...
Done.

======== [Kite Pipeline Stats] =========
Total number of clock cycles = 48
Total number of stalled cycles = 6
Total number of executed instructions = 17
Cycles per instruction = 2.824

Data cache stats:
    Number of loads = 0
    Number of stores = 1
    Number of writebacks = 0
    Miss rate = 1.000 (1/1)

Register state:
x0 = 0
x1 = 0
x2 = 4096
x3 = 0
```

Fig. 2.1. Kite simulator's CLI

Its source code can be found in the Yonsei University Computer Architecture and Systems Lab's GitHub [8].

ARMLite

ARMLite [9] is a web-based simulator for a 32-bit 'ARM-like' processor. It includes a basic set of instructions described in *Assembly Language Programming*, by R. Pawson [10]. This simulator was developed for educational purposes, but specifically to target AQA [11]'s Assembly language instruction set for its A-level computer syllabus [12].

The simulator offers a Graphical User Interface (GUI) (Figure 2.2) that shows the

current state of the registers and the data memory, as well as the current instruction and the status bits. It also adds Input/Output (I/O) support (a text box and a display), and options to start, stop, pause, resume, and slow or speed up execution, as well as step-by-step execution. It allows for users to load, save, and edit their programs, as well as a box to output system information (errors, last action performed, etc.).

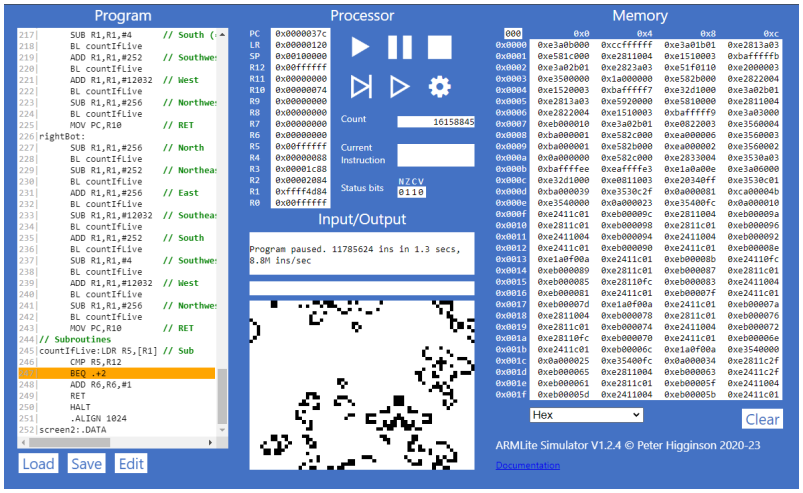


Fig. 2.2. ARMLite executing Conway's Game of Life [13]

While not being FOSS *per se* (no information about the license used by this simulator could be found), the simulator uses JavaScript, which means the browser interprets the webpages's source code locally. As the code is unobfuscated and is, to date, public and free to use, it is possible *could* consider this simulator FOSS.

2.1.2. Generic Simulators

Simulators that are able to define and execute multiple ISAs are considered 'generic simulators'. They add a significant complexity to those types of simulators, due to the fact that they must be able to dynamically adapt to the different architectures.

Two different approaches to creating such simulators can be found in two specific examples: Sail and CREATOR.

Sail

Sail [14] is an ISA definition language. It was developed by the Rigorous Engineering of Mainstream Systems (REMS) research group [15], from the University of Cambridge, as a tool to allow ISAs to be mathematically modeled and verified and formally proven [16].

This tool can not only type-check instruction and vector lengths for the ISA, generate theorem provers definitions of the architecture and tests, or automatically generate documentation, but, more importantly for this specific case, it can generate executable simulators in C or OCaml based on those ISAs.

CREATOR

CREATOR [17] is a web-based didactic and generic assembly language simulator. It was created by the ARCOS group from Universidad Carlos III de Madrid as a tool to teach students from the Computer Architecture and Computer Structure courses assembly language programming [18].

It is a fully-fledged simulator, with support for an assembly language editor with syntax highlighting and error detection, step-by-step execution, I/O, breakpoints, visualization of memory, registers and stack, floating point support, loading and saving files, adding libraries, detection of argument passing conventions, etc., all accessible through its GUI (Figure 2.3), or a CLI.

Furthermore, it allows the user to view, edit, and create the architecture in use, using JavaScript. CREATOR also allows the user to tweak a lot of the simulator's architecture and behavior. They can edit the memory layout, register file (including number of bits, names and alias, and properties such as enabling reading or writing, if its value is saved when jumping into a subroutine, or which one is the stack and global pointer), instructions and pseudo-instructions (clock cycles, number of words, fields, etc.), and assembly directives.

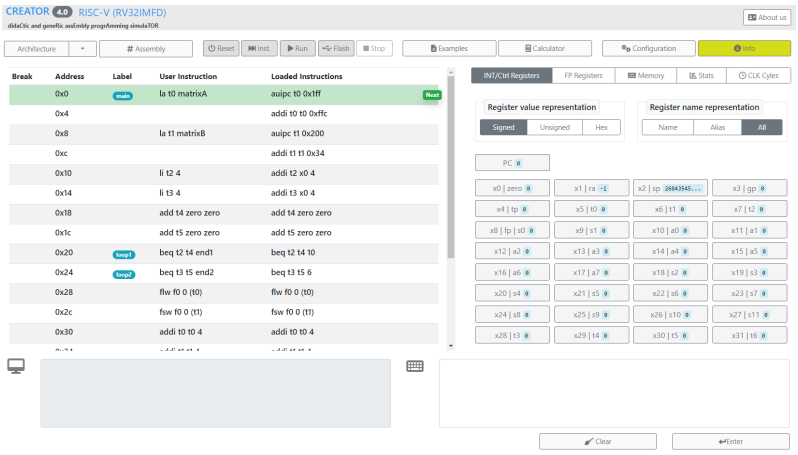


Fig. 2.3. CREATOR's main GUI

2.2. Comparison

This proposal aims to take the desirable features from each of the different simulators presented in order to create an application which fits our needs.

As it was justified before, the goal is to provide a *generic* simulator and, from the two discussed approaches, CREATOR's approach is more suitable for the proposal, as the ability to quickly modify the architecture instead of re-compiling the whole simulator encourages experimentation (less 'friction'). Another thing that must also be taken into account is that some features that make the simulator more generic might also create more

of this 'friction'; giving users a more fine-grained control over the architecture could discourage experimentation and add unnecessary complexity, if the goal is to prioritize for the user's understanding of assembly language over their understanding of computer architecture.

Table 2.1 provides a comparison of the different features of the current assembly simulators and the proposal.

The evaluated features are the following:

- **Language:** The main programming language the software is written in.
- **License:** The legal terms and conditions under which the software is licensed.
- **CLI:** Whether the application offers a CLI.
- **I/O:** Whether the application supports a I/O operations.
- **Step-by-step execution:** Whether the simulator can execute programs step-by-step (either instruction-by-instruction or cycle-by-cycle).
- **Simple architecture definition:** Whether the simulator's ISA is defined in a simple way.
- **Native execution:** Whether the simulator can be executed natively in hardware.
- **In-browser:** Whether the simulator can be executed in a browser.
- **Error checking:** Whether the simulator offers some type of checking of syntactic errors in the program.
- **Architecture validation support:** Whether the simulator offers support for validating the defined architecture, through the use of an external tool or through other means such as a REPL environment for the instruction definition language.

It is also possible to compare the mentioned simulators by plotting them on a two-dimension map (Figure 2.4), with one axis representing how 'generic' it is and the other one representing how 'didactic' it is.

TABLE 2.1

FEATURE COMPARISON OF CURRENT ASSEMBLY SIMULATORS

Simulator	Kite	ARMLite	Sail	CREATOR	Proposal
Language	C/C++	JavaScript	OCaml	JavaScript	ISO C++20
License	BSD-3-Clause	None	BSD-2-Clause	LGPLv2.1	GPLv3
CLI	✓			✓	✓
I/O		✓		✓	✓
Step-by-step execution				✓	✓
Simple architecture definition				✓	✓
Native execution	✓		✓		✓
In-browser		✓		✓	✓*
Error checking		✓		✓	✓
Architecture validation support					✓

* Future work

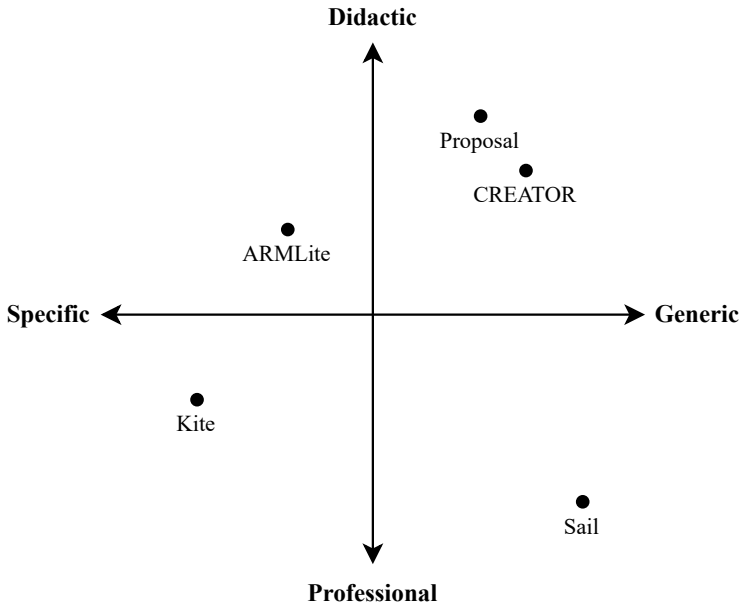


Fig. 2.4. Simulator comparison map

CHAPTER 3

ANALYSIS

This chapter describes the proposed solution by briefly recapping the project (Section 3.1, *Project Overview*), specifying the system's requirements (Section 3.2, *Requirements*), and specifying the different use cases for the system (Section 3.3, *Use Cases*).

3.1. Project Overview

The main objective of this project is to create an application that is capable of simulating an ISA defined by the user, with educational purposes. This will help the user learn assembly language programming by incentivizing experimentation and, in turn, giving the user a deeper understanding of assembly languages. Moreover, the instructions should be defined using a simple language that supports some form of validation of the instruction set, in order to facilitate ensuring that the ISA is correct even before executing any program.

As stated in Chapter 2, *State of the Art*, current simulators either don't offer this possibility, or offer an overwhelming amount of options and configurations, which focus on the hardware and architecture aspects of the simulation instead on the assembly language itself.

The aim is to provide the user with a more 'logical' approach to understanding assembly language programming, by abstracting the hardware details and providing a simple and generic architecture.

3.2. Requirements

This section provides a detailed description of the system's requirements. For the requirement specification task, the IEEE recommended practices [19] were followed. According to these practices, a good specification must address the software functionality, performance issues, external interfaces, other non-functional features and design or implementation constraints.

Moreover, the requirement specification must be:

1. **Complete:** The document reflects all significant software requirements.
2. **Consistent:** Requirements must not generate conflicts with each other.
3. **Correct:** Every requirement is one that the software shall meet according to the user needs.
4. **Modifiable:** The structure of the specification allows changes to the requirements in a simple, complete and consistent way.
5. **Ranked based on importance and stability:** Every requirement must indicate its importance and its stability.
6. **Traceable:** The origin of every requirement is clear, and it can be easily referenced in further stages.
7. **Unambiguous:** Every requirement has a single interpretation.
8. **Verifiable:** Every requirement must be verifiable, that is, there exists some process to verify that the software complies with every single requirement.

Starting from the user requirements (Subsection 3.2.1, *User Requirements*), which constitute an informal reference to the product the client expects, the software requirements (Subsection 3.2.2, *Software Requirements*) were derived. These requirements guided the design process with specific information on the functionality of the system, as well as any and other related characteristics.

3.2.1. User Requirements

This section provides a detailed description of the user's requirements for the project. These requirements indicate the main functionality and restrictions the developed system must fulfill.

The user requirements are divided into two distinct types:

- **Capacities:** Describe the expected system's functionality.

- **Restrictions:** Impose constraints or conditions that the system must fulfill.

Each user requirement is uniquely identified by an ID, which follows the format *UR-YY-XX*, where *YY* identifies the type of the requirement, either a capacity (*CA*) or a restriction (*RE*); and *XX* identifies the sequential number of the requirement within that type, starting at *01*.

Table 3.1 provides the template used for the specification of the requirements, including the description of each attribute.

TABLE 3.1
USER REQUIREMENT TEMPLATE

UR-YY-XX	
Description	Detailed description of the requirement.
Necessity	Priority of the requirement for the user (*Essential*, *Convenient* or *Optional*).
Priority	Priority of the requirement for the developer (*High*, *Medium* or *Low*).
Stability	Indicates the requirement's variability trough the development process (*Constant*, *Inconstant* or *Very unstable*).
Verifiability	Ability to test the validity of the requirement (*High*, *Medium* or *Low*).

TABLE 3.2
REQUIREMENT UR-CA-01

UR-CA-01	
Description	The system shall simulate a basic and generic computer architecture.
Necessity	Essential
Priority	High
Stability	Constant
Verifiability	Medium

TABLE 3.3
REQUIREMENT UR-CA-02

UR-CA-02	
Description	The system shall display information about all system components at each step of the simulation.
Necessity	Essential
Priority	High
Stability	Constant
Verifiability	High

TABLE 3.4
REQUIREMENT UR-CA-03

UR-CA-03	
Description	The system shall allow the user to define its own ISA.
Necessity	Essential
Priority	High
Stability	Constant
Verifiability	High

TABLE 3.5
REQUIREMENT UR-CA-04

UR-CA-04	
Description	The system shall be able to execute any program written in the defined instruction set, instruction by instruction.
Necessity	Essential
Priority	High
Stability	Constant
Verifiability	Medium

TABLE 3.6
REQUIREMENT UR-CA-05

UR-CA-05	
Description	The system shall have a CLI.
Necessity	Convenient
Priority	Medium
Stability	Inconstant
Verifiability	High

TABLE 3.7
REQUIREMENT UR-CA-06

UR-CA-06	
Description	The system shall notify the user of syntax errors in the program.
Necessity	Essential
Priority	High
Stability	Constant
Verifiability	High

TABLE 3.8
REQUIREMENT UR-CA-07

UR-CA-07	
Description	The data on the components shall be represented in various formats: signed integer, unsigned integer, and hexadecimal.
Necessity	Optional
Priority	Low
Stability	Inconstant
Verifiability	High

TABLE 3.9
REQUIREMENT UR-CA-08

UR-CA-08	
Description	The system shall be able to fully execute and stop the execution of any program.
Necessity	Convenient
Priority	Medium
Stability	Constant
Verifiability	High

TABLE 3.10
REQUIREMENT UR-CA-09

UR-CA-09	
Description	The system shall provide a REPL environment for the user to test the instruction definition language.
Necessity	Convenient
Priority	Medium
Stability	Constant
Verifiability	High

TABLE 3.11
REQUIREMENT UR-RE-01

UR-RE-01	
Description	The system shall be able to be executed natively, and support multiple platforms.
Necessity	Convenient
Priority	High
Stability	Inconstant
Verifiability	Medium

TABLE 3.12
REQUIREMENT UR-RE-02

UR-RE-02	
Description	The system shall be FOSS.
Necessity	Convenient
Priority	Medium
Stability	Inconstant
Verifiability	High

TABLE 3.13
REQUIREMENT UR-RE-03

UR-RE-03	
Description	The system shall be intuitive and easy to use.
Necessity	Convenient
Priority	Medium
Stability	Constant
Verifiability	Low

TABLE 3.14
REQUIREMENT UR-RE-04

UR-RE-04	
Description	The system shall provide a comprehensive user manual.
Necessity	Essential
Priority	Medium
Stability	Constant
Verifiability	High

TABLE 3.15
REQUIREMENT UR-RE-05

UR-RE-05	
Description	The ISA definition format shall be simple and easily modifiable.
Necessity	Convenient
Priority	Medium
Stability	Constant
Verifiability	Medium

TABLE 3.16
REQUIREMENT UR-RE-06

UR-RE-06	
Description	The instruction definition language shall be simple, easy to understand, and validatable by an external tool.
Necessity	Convenient
Priority	Medium
Stability	Constant
Verifiability	Low

3.2.2. Software Requirements

This section provides a detailed description of the system's software requirements for the project. These requirements are derived from the user requirements, defined in Subsection 3.2.1, and comprise the software specifications for the system.

The user requirements are divided into two distinct types:

- **Functional requirements:** Specify the software's functionalities and characteristics.
- **Non-functional requirements:** Specify other non-functional characteristics of the software.

Each software requirement is uniquely identified by an ID, which follows the format *SR-YY-XX*, where *YY* identifies the type of the requirement, either functional (*FC*) or non-functional (*NF*); and *XX* identifies the sequential number of the requirement within that type, starting at *01*.

Table 3.17 provides the template used for the specification of the requirements, including the description of each attribute.

TABLE 3.17
SOFTWARE REQUIREMENT TEMPLATE

SR-YY-XX	
Description	Detailed description of the requirement.
Necessity	Priority of the requirement for the user (*Essential*, *Convenient* or *Optional*).
Priority	Priority of the requirement for the developer (*High*, *Medium* or *Low*).
Stability	Indicates the requirement's variability trough the development process (*Constant*, *Inconstant* or *Very unstable*).
Verifiability	Ability to test the validity of the requirement (*High*, *Medium* or *Low*).
Origin	User requirements that derived this requirement.

TABLE 3.18
REQUIREMENT SR-FC-01

SR-FC-01	
Description	The system shall simulate the basic components of a computer: data memory, text memory, registers, Arithmetic Logic Unit (ALU) and I/O.
Necessity	Essential
Priority	High
Stability	Constant
Verifiability	High
Origin	UR-CA-01

TABLE 3.19
REQUIREMENT SR-FC-02

SR-FC-02	
Description	The system shall simulate all basic computer operations: arithmetic operations, logical operations, branching, memory manipulation, and I/O.
Necessity	Essential
Priority	High
Stability	Inconstant
Verifiability	Medium
Origin	UR-CA-01, UR-CA-03

TABLE 3.20
REQUIREMENT SR-FC-03

SR-FC-03	
Description	The system shall display the contents and state of the components of the current ISA at each step of the simulation.
Necessity	Essential
Priority	High
Stability	Constant
Verifiability	High
Origin	UR-CA-02

TABLE 3.21
REQUIREMENT SR-FC-04

SR-FC-04	
Description	The user shall be able to configure the different components of the simulator.
Necessity	Essential
Priority	High
Stability	Constant
Verifiability	High
Origin	UR-CA-01, UR-CA-03

TABLE 3.22
REQUIREMENT SR-FC-05

SR-FC-05	
Description	The system shall be able to interpret instructions defined in a simple language.
Necessity	Essential
Priority	High
Stability	Constant
Verifiability	High
Origin	UR-CA-03

TABLE 3.23
REQUIREMENT SR-FC-06

SR-FC-06	
Description	The system shall be able to transform assembly-like instructions, defined in the ISA, into instructions in a simple language that the system can interpret.
Necessity	Essential
Priority	High
Stability	Constant
Verifiability	High
Origin	UR-CA-04

TABLE 3.24
REQUIREMENT SR-FC-07

SR-FC-07	
Description	The system shall be able to execute programs instruction by instruction.
Necessity	Essential
Priority	High
Stability	Constant
Verifiability	High
Origin	UR-CA-04

TABLE 3.25
REQUIREMENT SR-FC-08

SR-FC-08	
Description	The user shall be able to load programs and ISA definitions in the simulator.
Necessity	Essential
Priority	High
Stability	Constant
Verifiability	High
Origin	UR-CA-04

TABLE 3.26
REQUIREMENT SR-FC-09

SR-FC-09	
Description	The user shall be able to interact with the system through a CLI.
Necessity	Essential
Priority	High
Stability	Constant
Verifiability	High
Origin	UR-CA-05

TABLE 3.27
REQUIREMENT SR-FC-10

SR-FC-10	
Description	The system shall detect syntactical errors on the executed program.
Necessity	Essential
Priority	High
Stability	Constant
Verifiability	High
Origin	UR-CA-06

TABLE 3.28
REQUIREMENT SR-FC-11

SR-FC-11	
Description	The user shall be able to change the representation of the data on the components: either signed integer, unsigned integer, or hexadecimal.
Necessity	Essential
Priority	High
Stability	Constant
Verifiability	High
Origin	UR-CA-07

TABLE 3.29
REQUIREMENT SR-FC-12

SR-FC-12	
Description	The user shall allow the user to start executing the rest of the program automatically, and stop the execution of a current program.
Necessity	Convenient
Priority	Medium
Stability	Inconstant
Verifiability	High
Origin	UR-CA-08

TABLE 3.30
REQUIREMENT SR-FC-13

SR-FC-13	
Description	The system shall provide a REPL environment for the instruction definition language.
Necessity	Convenient
Priority	Medium
Stability	Inconstant
Verifiability	High
Origin	UR-CA-09

TABLE 3.31
REQUIREMENT SR-NF-01

SR-NF-01	
Description	The system shall be able to be executed natively.
Necessity	Convenient
Priority	High
Stability	Constant
Verifiability	High
Origin	UR-RE-01

TABLE 3.32
REQUIREMENT SR-NF-02

SR-NF-02	
Description	The system shall support multiple platforms.
Necessity	Convenient
Priority	High
Stability	Constant
Verifiability	High
Origin	UR-RE-01

TABLE 3.33
REQUIREMENT SR-NF-03

SR-NF-03	
Description	The system shall be licensed under a FOSS license.
Necessity	Convenient
Priority	Medium
Stability	Constant
Verifiability	High
Origin	UR-RE-02

TABLE 3.34
REQUIREMENT SR-NF-04

SR-NF-04	
Description	The system's source code shall be publicly available.
Necessity	Optional
Priority	Low
Stability	Constant
Verifiability	High
Origin	UR-RE-02

TABLE 3.35
REQUIREMENT SR-NF-05

SR-NF-05	
Description	The system shall have an intuitive User Interface (UI).
Necessity	Convenient
Priority	Medium
Stability	Constant
Verifiability	Low
Origin	UR-RE-03

TABLE 3.36
REQUIREMENT SR-NF-06

SR-NF-06	
Description	The system shall provide a comprehensive user manual.
Necessity	Convenient
Priority	Medium
Stability	Constant
Verifiability	Low
Origin	UR-RE-04

TABLE 3.37
REQUIREMENT SR-NF-07

SR-NF-07	
Description	The format used to define the ISA shall be simple and easily modifiable.
Necessity	Convenient
Priority	Low
Stability	Inconstant
Verifiability	Medium
Origin	UR-RE-05

TABLE 3.38
REQUIREMENT SR-NF-08

SR-NF-08	
Description	The language used to define the instructions shall be easily validatable by and external tool.
Necessity	Convenient
Priority	Low
Stability	Inconstant
Verifiability	Medium
Origin	UR-RE-06

TABLE 3.39
REQUIREMENT SR-NF-09

SR-NF-09	
Description	The language used to shall be simple and easy for the user to use and understand.
Necessity	Convenient
Priority	Medium
Stability	Inconstant
Verifiability	Low
Origin	UR-RE-06

3.2.3. Traceability

A traceability matrix verifies if software requirements cover all user requirements. All functional requirements cover the capabilities (Table 3.40) and all non-functional requirements cover the restrictions (Table 3.41), proving the analysis was correct.

TABLE 3.40
TRACEABILITY BETWEEN CAPACITIES AND FUNCTIONAL REQUIREMENTS

	UR-CA-01	UR-CA-02	UR-CA-03	UR-CA-04	UR-CA-05	UR-CA-06	UR-CA-07	UR-CA-08	UR-CA-09
SR-FC-01	•								
SR-FC-02	•		•						
SR-FC-03		•							
SR-FC-04	•		•						
SR-FC-05			•						
SR-FC-06				•					
SR-FC-07				•					
SR-FC-08				•					
SR-FC-09					•				
SR-FC-10						•			
SR-FC-11							•		
SR-FC-12								•	
SR-FC-13									•

TABLE 3.41
TRACEABILITY BETWEEN RESTRICTIONS AND NON-FUNCTIONAL REQUIREMENTS

	UR-RE-01	UR-RE-02	UR-RE-03	UR-RE-04	UR-RE-05	UR-RE-06
SR-NF-01	•					
SR-NF-02	•					
SR-NF-03		•				
SR-NF-04		•				
SR-NF-05			•			
SR-NF-06				•		
SR-NF-07					•	
SR-NF-08						•
SR-NF-09						•

3.3. Use Cases

The UML use case model [20] (Figure 3.1) represents the typical usage of the system, allowing developer to visualize the different interactions between the user and the functionalities the system offers. The use case model includes all the different use cases for the system.

Each use case represents the process an external agent (actor) should follow in order to execute a specific functionality, and is uniquely identified by an ID. The ID follows the format *UC-XX*, where *XX* identifies the sequential number of the use case, starting at *01*.

Table 3.42 provides the template used for the specification of the use case, including the description of each attribute.

TABLE 3.42
USE CASE TEMPLATE

UC-XX	
Name	Brief description of the use case.
Actors	External agent that executes the use case.
Objetive	The use case's purpose.
Description	Steps that the external agent must take to execute the use case.
Pre-condition	Conditions that must be fulfilled *before* executing the use case.
Post-condition	Conditions that must be fulfilled *after* executing the use case.

TABLE 3.43
USE CASE UC-01

UC-01	
Name	Load ISA
Actors	User
Objetive	Loading the ISA on the simulator from a file specified by the user.
Description	1. User provides the route to the ISA file through an executable argument.
Pre-condition	N/A
Post-condition	The ISA is loaded in the simulator, and the main CLI starts.

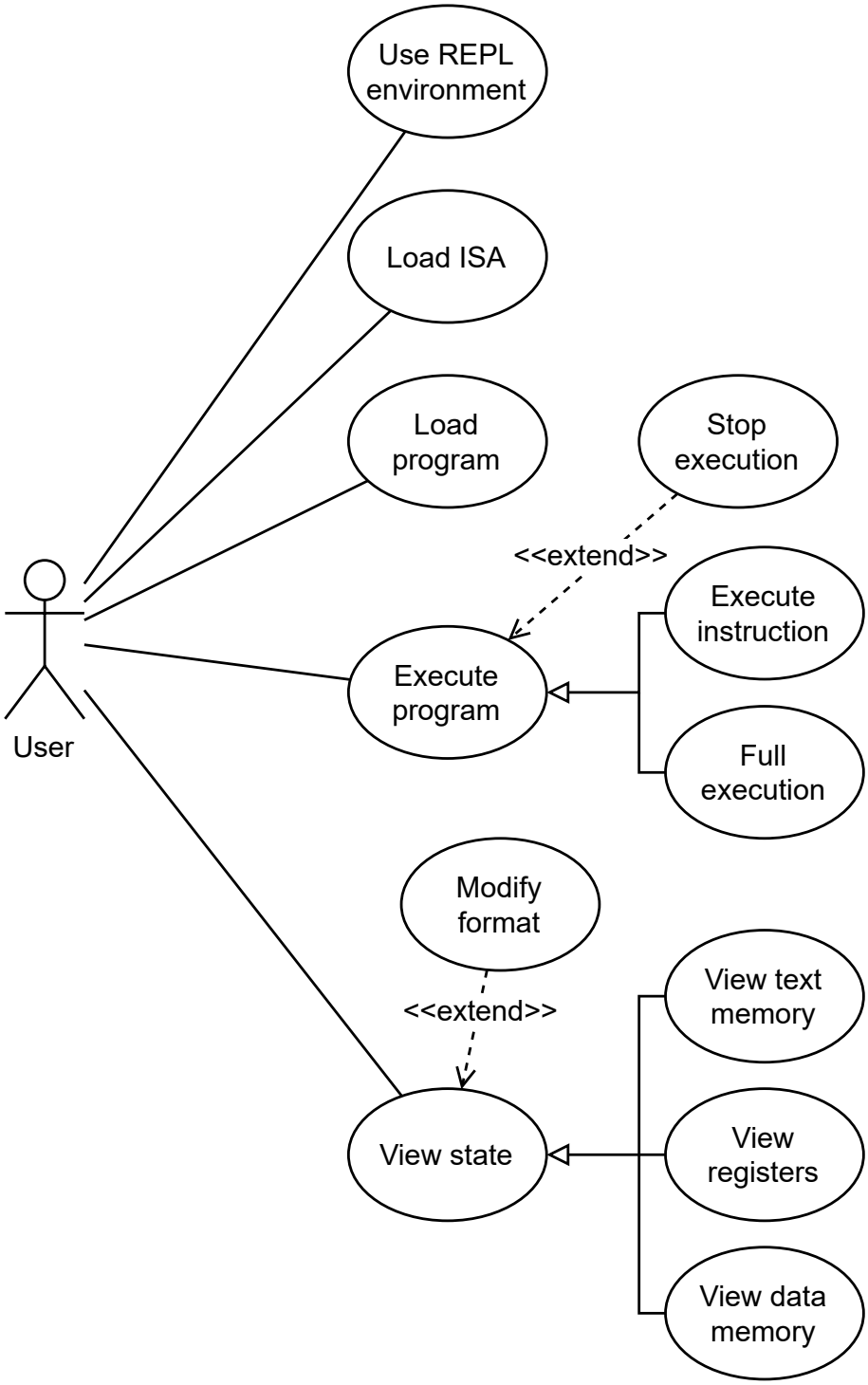


Fig. 3.1. Use case model

TABLE 3.44
USE CASE UC-02

UC-02	
Name	Load program
Actors	User
Objetive	Loading a program on the simulator from a file specified by the user.
Description	1. User selects the 'load program' option in the CLI. 2. User specifies the route to the program file.
Pre-condition	The ISA is loaded in the simulator.
Post-condition	The program is loaded in the simulator.

TABLE 3.45
USE CASE UC-03

UC-03	
Name	Execute instruction
Actors	User
Objetive	Executing the next instruction of the program on the simulator.
Description	1. User selects the 'next instruction' option in the CLI.
Pre-condition	The ISA and program are loaded in the simulator.
Post-condition	The simulator executes one instruction of the program.

TABLE 3.46
USE CASE UC-04

UC-04	
Name	Full program execution
Actors	User
Objetive	Fully executing the current program.
Description	1. User selects the 'full execution' option in the CLI.
Pre-condition	The ISA and program are loaded in the simulator.
Post-condition	The simulator starts/continues executing the program automatically.

TABLE 3.47
USE CASE UC-05

UC-05	
Name	Stop program execution
Actors	User
Objetive	Stopping the current program execution.
Description	1. User selects the 'stop execution' option in the CLI.
Pre-condition	The ISA and program are loaded in the simulator, and the program is executing.
Post-condition	The simulator stops executing the program.

TABLE 3.48
USE CASE UC-06

UC-06	
Name	Execute instruction
Actors	User
Objetive	Executing the next instruction of the program on the simulator.
Description	1. User selects the 'next instruction' option in the CLI.
Pre-condition	The ISA and program are loaded in the simulator.
Post-condition	The simulator executes one instruction of the program.

TABLE 3.49
USE CASE UC-07

UC-07	
Name	Use REPL environment
Actors	User
Objetive	Evaluating instruction definition Language expressions.
Description	1. User selects the 'REPL' option in the CLI. 2. User types the expression to evaluate.
Pre-condition	The ISA and program are loaded in the simulator.
Post-condition	The simulator evaluates the expressions.

CHAPTER 4

DESIGN

This chapter provides a full description of the proposed solution. It details the system's design process, by discussing the different alternatives (Section 4.1, *Study of the Solution*), and describes the proposed architecture (Section 4.2, *System Architecture*), including all components and design decisions.

4.1. Study of the Solution

After the elicitation of the requirements (Section 3.2, *Requirements*) and use cases (Section 3.3, *Use Cases*) of the system, the tools to build the application must be discussed, as the design will depend on them.

4.1.1. Instruction Definition Language

Requirement SR-FC-05 calls for the creation of a language to encode the functionality of the different instructions on the ISA. This language is the one that the simulator will actually execute. This language, as per requirement SR-NF-09, should be simple and easy to understand.

One approach is to use a generic assembly language, such as the one described in the IEEE Standard for Microprocessor Assembly Language [21]. This is a set of common set of instructions for all microprocessors, and *should* cover all ISA possibilities. The main problem with this solution is that it wouldn't solve the main problem, to teach the user assembly language programming, as the instructions would be encoded in yet another assembly language. Therefore, some other type of simple and generic language that's easier to understand must be used.

There are scripting languages with good integration with other programming lan-

guages, such as Lua [22]. Being an embeddable language, it can function as a simple language to interact with an application. Lua is currently used in several projects, such as Neovim [23] and LuaTEX [24]. The problem with this approach is requirement SR-NF-08; Lua is not easily validatable due to the complexity of the language.

The final approach is to create a domain-specific language. This language should be simple, but able to execute any computation, and abstract enough to be easy to understand. The solution is to implement a language in the LISP family, a concept similar to WebAssembly's Text Format [25]. LISP [26] is a simple recursive language whose operations are based on symbolic expressions. Expressions are lists of elements, where the first element is the operator and the rest of elements the operand. Elements can be atomic (numbers, operators, etc.), or other lists, which will be synthesized into atomic elements by the recursive pattern. By implementing in the language the basic operations described in requirement SR-FC-02, all of the goals for a simple language that supports any possible ISA are fulfilled.

4.1.2. Interpreter

Requirement SR-FC-06 states that the system must be able to execute the instruction definition language, and requirement SR-FC-07 states it should be able to do it instruction by instruction. Therefore, the best approach is to create an interpreter for the LISP-like language.

To implement an interpreter for any language, there are two main steps involved: processing the input into an Abstract Syntax Tree (AST) (*parsing*), and executing it with the help of the AST (*evaluating*) [27].

Nowadays, there are tools that, given a language definition as an input, generate parsers for those languages, simplifying the process of creating an interpreter. Such is the case of Lex and yacc [28] (or its modern counterpart, Flex and bison [29]), and ANTLR [30]. These parser generator tools generate libraries that parse the inputs into ASTs that can be later used in the evaluation phase. The main problem with this approach is that you need to re-generate the parser libraries each time you modify the language, which gives less flexibility to add new features (floating point operations, etc.).

The final approach is to build the interpreter from scratch. Being a LISP-like language comes with the advantage of simplifying the interpreter's design. Make-A-Lisp [31] is a guide for implementing a generic LISP interpreter, and outlines the basic functionalities and architecture.

4.1.3. ISA Definition Format

Requirement SR-NF-07 specifies that the format used to define and store the ISA must be simple and easy to modify. The best approach is to use a text-based format, as they are

human-readable and can be modified by any basic text editor, without the need for any specific tools. Furthermore, it allows us to encode with text both the defined assembly languages and instruction definition language, simplifying the system's implementation.

The best approach is to use a structured data format. There are many formats, but the most widely used are those who store data in attribute-value pairs and arrays, as is the case of the JavaScript Object Notation (JSON) [32] and the YAML Ain't Markup Language (YAML) [33] formats. While both formats offer libraries for parsing and validating files in many programming languages, the JSON format is a 'friendlier' format for HyperText Transfer Protocol (HTTP) due to the fact that it ignores whitespace, unlike YAML. Using JSON over YAML offers many possibilities for the use of the system in other environments, such as web services.

4.1.4. Programming Language

As per requirements SR-NF-01, the programming language must allow the system to be executed natively and on multiple platforms. Native execution has the advantage of producing more performant code, as there is no overhead derived from virtual machines or interpreters. Popular natively compiled languages include C [34], C++ [35], Rust [36], and Go [37] [38].

Furthermore, requirement SR-NF-02 states that the system shall also support multiple platforms, and this includes browsers. Through the use of WebAssembly [39], many programming languages can be compiled and executed 'natively' in the browser, including all previously mentioned languages.

From the mentioned languages, C++ (specifically the current standard at the time of writing, ISO C++20 [40]) was chosen for several reasons:

- It's a mature language, with support for modern features and multiple programming paradigms.
- It's strongly typed, memory safe, and high-performant, with zero-cost abstractions.
- It's an ISO standard language.

4.2. System Architecture

The system's architecture can be divided in four main elements:

- **Compiler:** Translates the assembly-like programs into the interpreter's language, using the instruction definitions.
- **Memory:** Stores the instructions and program memory.

- **Central Processing Unit (CPU):** Fetches, decodes, and executes the instructions (in the interpreter's language), and stores the registers.
- **CLI:** Provides the UI and coordinates all the other elements.

Figure 4.1 details the UML component diagram [20] of the system, including all of its components. Table 4.1 provides the template used for the specification of the components, including the description of each attribute. The successive tables describe each of the components.

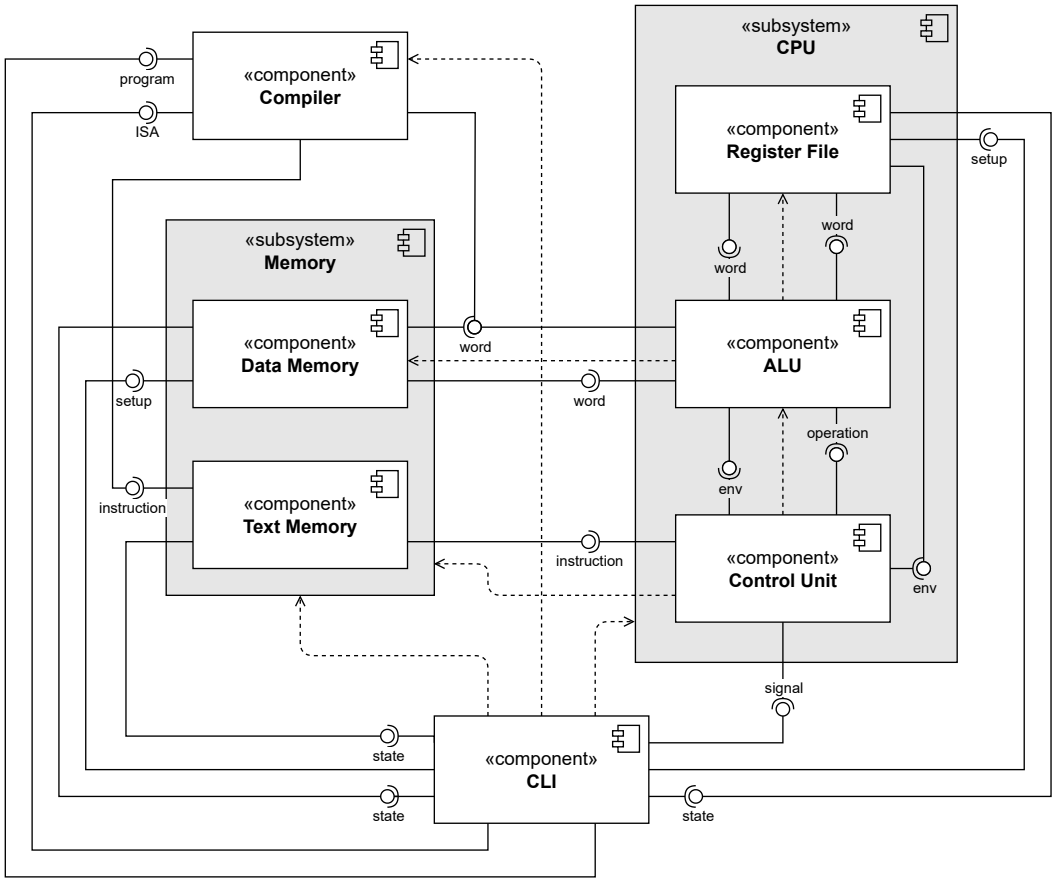


Fig. 4.1. Component diagram of the proposed solution

TABLE 4.1
COMPONENT TEMPLATE

Identifier	
Role	Component's function in the system.
Dependencies	Components that depend on this component.
Description	Explanation of the functioning of the component.
Data	Input (*in*) and output (*out*) data of the component.
Origin	Software requirements that derived the component.

TABLE 4.2
COMPONENT 'COMPILER'

Compiler	
Role	Compiles the program to execute into the interpreter's language
Dependencies	Memory subsystem (Data Memory, Text Memory), CPU subsystem (Register File, ALU, Control Unit), CLI
Description	The compiler translates the assembly-like program to execute into the interpreter's language, according to the ISA, loading the text memory. It also preloads the data memory segment if defined in the program.
Data	• **in:** Program to compile, ISA definition • **out:** Data Memory words to store (as defined in the program), program to execute.
Origin	SR-FC-06

TABLE 4.3
COMPONENT 'DATA MEMORY'

Data Memory	
Role	Stores the program's data
Dependencies	N/A
Description	Data memory stores words and maps them to specific addresses. It allows reading and writing to those bits, given a specified address. The start and end address is configurable.
Data	• **in:** Memory words to store. • **out:** Memory words to provide, state.
Origin	SR-FC-01, SR-FC-04, SR-FC-08

TABLE 4.4
COMPONENT 'TEXT MEMORY'

Text Memory	
Role	Stores the program's instructions
Dependencies	N/A
Description	Text memory stores the instructions to be executed by the Control Unit. Each instruction is composed of an address and the instruction data itself. This component allows for adding new instructions and reading any instruction, given its address.
Data	• **in:** Instructions to store, setup configuration. • **out:** Instructions to provide, state.
Origin	SR-FC-01, SR-FC-07

TABLE 4.5
COMPONENT 'REGISTER FILE'

Register File	
Role	Stores the register's data
Dependencies	N/A
Description	The register file holds all the registers of the ISA, and its contents, words. It also holds the program counter. The number of registers is configurable.
Data	• **in:** Register data to store, setup configuration • **out:** Register data to provide, state, environment (register names).
Origin	SR-FC-01, SR-FC-04, SR-FC-08

TABLE 4.6
COMPONENT 'ALU'

ALU	
Role	Executes the operations in the instructions
Dependencies	Data Memory, Register File words
Description	The ALU interacts with the data memory, registers and I/O in order to execute the operations required by the instructions.
Data	• **in:** Data Memory words to read, Register File words to read, environment (available operations). • **out:** Data Memory words to write, Register File words to write, operation to execute.
Origin	SR-FC-01, SR-FC-02

TABLE 4.7
COMPONENT 'CONTROL UNIT'

Control Unit	
Role	Decodes and executes the instructions
Dependencies	ALU, Memory subsystem (Data Memory, Text Memory)
Description	
Data	• **in:** Text Memory instructions, CLI signal, ALU environment, Register File environment. • **out:** Operation to execute.
Origin	SR-FC-05, SR-FC-07, SR-FC-10, SR-FC-12, SR-FC-13

TABLE 4.8
COMPONENT 'CLI'

CLI	
Role	Interfaces with the user and coordinates the system
Dependencies	Memory subsystem (Data Memory, Text Memory)
Description	The CLI is the main component of the system. It's in charge of executing its main functionalities: validating the input files, executing the program (by signaling the Control Unit) and viewing the state of the memory and register file.
Data	• **in:** Register File state, Data Memory state, Text Memory state. • **out:** Control Unit signals, Data Memory setup configuration, Register File setup configuration, program to execute, ISA.
Origin	SR-FC-03, SR-FC-04, SR-FC-07, SR-FC-08, SR-FC-09, SR-FC-10, SR-FC-11, SR-FC-12, SR-FC-13

Traceability

A traceability matrix was generated (Table 4.9) in order to validate that all the software requirements, defined in Subsection 3.2.2, were satisfied in the components of the architecture.

TABLE 4.9
TRACEABILITY BETWEEN FUNCTIONAL REQUIREMENTS AND COMPONENTS

	SR-FC-01	SR-FC-02	SR-FC-03	SR-FC-04	SR-FC-05	SR-FC-06	SR-FC-07	SR-FC-08	SR-FC-09	SR-FC-10	SR-FC-11	SR-FC-12	SR-FC-13
ALU	•	•											
CLI			•	•			•	•	•	•	•	•	•
Compiler						•							
Control Unit					•		•			•		•	•
Data Memory	•			•				•					
Register File	•			•				•					
Text Memory	•						•						

4.2.1. Instruction Definition Language

As stated in Subsection 4.1.1, the language used to define the instructions is a language from the LISP family. The developed language is LUISP-DA (*assembLy analogoUs lISP DiAlect*).

Being a LISP-like language, instructions are defined as LISP expressions, comprised of lists of symbols and/or other lists, delimited by parenthesis. Inner expressions are evaluated first, recursively. To simplify the language, there are no variables, macros, or functions, as they are not necessary for defining basic instructions.

There are three data types:

- $\langle NUM \rangle$, integer numbers.
- $\langle BOOL \rangle$, boolean values.
- $\langle NIL \rangle$, 'none' type.

The first element of each expression is called the *operator*, and takes the rest of the elements of the expression the values the operator is applied to. There are several predefined operators, which can be categorized by type they evaluate to:

- Operators that evaluate to $\langle NUM \rangle$:

  - *add*: Adds two values.
  - *sub*: Subtracts two values.
  - *neg*: Negates a value.
  - *mul*: Multiplies two values.
  - *div*: Divides two values.
  - *mod*: Module of two values.
  - *get-reg*: Gets the contents of a register.
  - *set-reg*: Stores a value in a register. Evaluates to the value.
  - *get-pc*: Gets the current program counter.
  - *set-pc*: Stores a value in the program counter. Evaluates to the value.
  - *get-mem*: Gets the value in a memory address.
  - *set-mem*: Stores a value in a memory address. Evaluates to the value.

- Operators that evaluate to $\langle BOOL \rangle$:
  - *lt*: Evaluates if a value is less than other value.
  - *le*: Evaluates if a value is less or equal than other value.
  - *gt*: Evaluates if a value is greater than other value.
  - *ge*: Evaluates if a value is greater or equal than other value.
  - *eq*: Evaluates if a value is equal to other value.
  - *ne*: Evaluates if a value is not equal to other value.
  - *not*: Negates a value.

There are other 'special' operators, as well:

- Blocks ($\langle block \rangle$) allow the user to evaluate an infinite number of expressions, one after the other, the whole expression evaluating to $\langle NIL \rangle$.
- Conditionals ($\langle conditional \rangle$) enable branching, by using an *if-then-else* structure. If the first element evaluates to `true`, the the conditional evaluates to the second element; and if the condition evaluates to `false` (or $\langle NIL \rangle$), it evaluates to the third element instead (if the third element doesn't exist, it evaluates to $\langle NIL \rangle$).
- System calls ($\langle call \rangle$) evaluates the first element, known as the *opcode*, and executes the specified *call* with its arguments. The *opcodes* are defined in the ISA. There defined *calls* are:
  - *print-int*: Prints a value as an integer to Standard Output (STDOUT).
  - *read-int*: Stores a value from Standard Input (STDIN) in a register.
  - *print-char*: Prints a value as an ASCII character to STDOUT.

- *read-char*: Stores an ASCII character from STDIN in a register.
- *exit*: Ends the execution of the program.

The full grammar's representation, in Backus-Naur Form [41], is the following:

$$
\begin{array}{lcll}
\langle \textit{expression} \rangle & ::= & \langle \textit{ATOMIC} \rangle & \\
& | & \langle \textit{b-expression} \rangle & \\
& | & \langle \textit{i-expression} \rangle & \\
& | & \langle \textit{block} \rangle & \\
& | & \langle \textit{conditional} \rangle & \\
& | & \langle \textit{call} \rangle & \\
\\
\langle \textit{b-expression} \rangle & ::= & \langle \textit{BOOL} \rangle & \\
& | & \langle L \rangle \text{`<'} \ \langle \textit{i-expression} \rangle \ \langle \textit{i-expression} \rangle \langle R \rangle & \textit{lt} \\
& | & \langle L \rangle \text{`<='} \ \langle \textit{i-expression} \rangle \ \langle \textit{i-expression} \rangle \langle R \rangle & \textit{le} \\
& | & \langle L \rangle \text{`>'} \ \langle \textit{i-expression} \rangle \ \langle \textit{i-expression} \rangle \langle R \rangle & \textit{gt} \\
& | & \langle L \rangle \text{`>='} \ \langle \textit{i-expression} \rangle \ \langle \textit{i-expression} \rangle \langle R \rangle & \textit{ge} \\
& | & \langle L \rangle \text{`==`} \ \langle \textit{i-expression} \rangle \ \langle \textit{i-expression} \rangle \langle R \rangle & \textit{eq} \\
& | & \langle L \rangle \text{`!=`} \ \langle \textit{i-expression} \rangle \ \langle \textit{i-expression} \rangle \langle R \rangle & \textit{ne} \\
& | & \langle L \rangle \text{`!`} \ \langle \textit{b-expression} \rangle \langle L \rangle & \textit{not} \\
\\
\langle \textit{i-expression} \rangle & ::= & \langle \textit{NUM} \rangle & \\
& | & \langle L \rangle \text{`+'} \ \langle \textit{i-expression} \rangle \ \langle \textit{i-expression} \rangle \langle R \rangle & \textit{add} \\
& | & \langle L \rangle \text{`-'} \ \langle \textit{i-expression} \rangle \ \langle \textit{i-expression} \rangle \langle R \rangle & \textit{sub} \\
& | & \langle L \rangle \text{`-'} \ \langle \textit{i-expression} \rangle \langle R \rangle & \textit{neg} \\
& | & \langle L \rangle \text{`*'} \ \langle \textit{i-expression} \rangle \ \langle \textit{i-expression} \rangle \langle R \rangle & \textit{mul} \\
& | & \langle L \rangle \text{`/'} \ \langle \textit{i-expression} \rangle \ \langle \textit{i-expression} \rangle \langle R \rangle & \textit{div} \\
& | & \langle L \rangle \text{`\%'} \ \langle \textit{i-expression} \rangle \ \langle \textit{i-expression} \rangle \langle R \rangle & \textit{mod} \\
& | & \langle L \rangle \text{`reg'} \ \langle \textit{REG} \rangle \langle R \rangle & \textit{get-reg} \\
& | & \langle L \rangle \text{`reg!'} \ \langle \textit{REG} \rangle \ \langle \textit{i-expression} \rangle \langle R \rangle & \textit{set-reg} \\
& | & \langle L \rangle \text{`pc'} & \textit{get-pc} \\
& | & \langle L \rangle \text{`pc!'} \ \langle \textit{i-expression} \rangle \langle R \rangle & \textit{set-pc} \\
& | & \langle L \rangle \text{`mem'} \ \langle \textit{i-expression} \rangle \langle R \rangle & \textit{get-mem} \\
& | & \langle L \rangle \text{`mem!'} \ \langle \textit{i-expression} \rangle \ \langle \textit{i-expression} \rangle \langle R \rangle & \textit{set-mem} \\
\\
\langle \textit{block} \rangle & ::= & \langle L \rangle \text{`do'} \ \langle \textit{expr-block} \rangle \langle R \rangle & \\
\\
\langle \textit{expr-block} \rangle & ::= & \langle \textit{expression} \rangle & \\
& | & \langle \textit{expression} \rangle \ \langle \textit{expr-block} \rangle & \\
\\
\langle \textit{conditional} \rangle & ::= & \langle L \rangle \text{`if'} \ \langle \textit{b-expression} \rangle \ \langle \textit{expression} \rangle \ \langle \textit{expression} \rangle \langle R \rangle & \\
\\
\langle \textit{call} \rangle & ::= & \langle L \rangle \text{`call'} \ \langle \textit{NUM} \rangle \ \langle \textit{i-expression} \rangle \langle R \rangle & \textit{print-int / print-char} \\
& | & \langle L \rangle \text{`call'} \ \langle \textit{NUM} \rangle \ \langle \textit{REG} \rangle \langle R \rangle & \textit{read-int / read-char}
\end{array}
$$

	| ⟨*L*⟩'call' ⟨*NUM*⟩⟨*R*⟩	*exit*
⟨*L*⟩	::= '('	
⟨*R*⟩	::= ')'	
⟨*ATOMIC*⟩	::= ⟨*NUM*⟩ | ⟨*BOOL*⟩ | ⟨*NIL*⟩	
⟨*NUM*⟩	::= ⟨*DEC*⟩ | '0x'⟨*HEX*⟩	
⟨*DEC*⟩	::= ⟨*I-DIGIT*⟩ | ⟨*I-DIGIT*⟩⟨*DEC*⟩	
⟨*HEX*⟩	::= ⟨*H-DIGIT*⟩ | ⟨*H-DIGIT*⟩⟨*HEX*⟩	
⟨*I-DIGIT*⟩	::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'	
⟨*H-DIGIT*⟩	::= ⟨*I-DIGIT*⟩ | 'a' | 'b' | 'c' | 'd' | 'e' | 'f'	
⟨*BOOL*⟩	::= 'true' | 'false'	
⟨*NIL*⟩	::= 'nil'	
⟨*REG*⟩	::= *The set of register names defined in the ISA*	

For more information about the use of the language, see Appendix A, *User manual*.

4.2.2. Control Unit

The LUISP-DA interpreter is implemented in the Control Unit component. As stated in Subsection 4.1.2, the architecture of the interpreter will be based on the Make-A-Lisp [31] architecture. Nevertheless, as it is only needed to model a simple assembly-like language, a subset of the original architecture will be used, modified in order to fit the project's purposes.

Figure 4.2 shows an overview of the Control Unit. As stated in Subsection 4.1.2, the interpretation is done in two stages:

1. **READ:** The instruction, a string, is parsed into an AST.

2. **EVAL:** The AST is evaluated recursively.

Finally, after each instruction (except the ones inside a ⟨*block*⟩), the program counter is incremented.

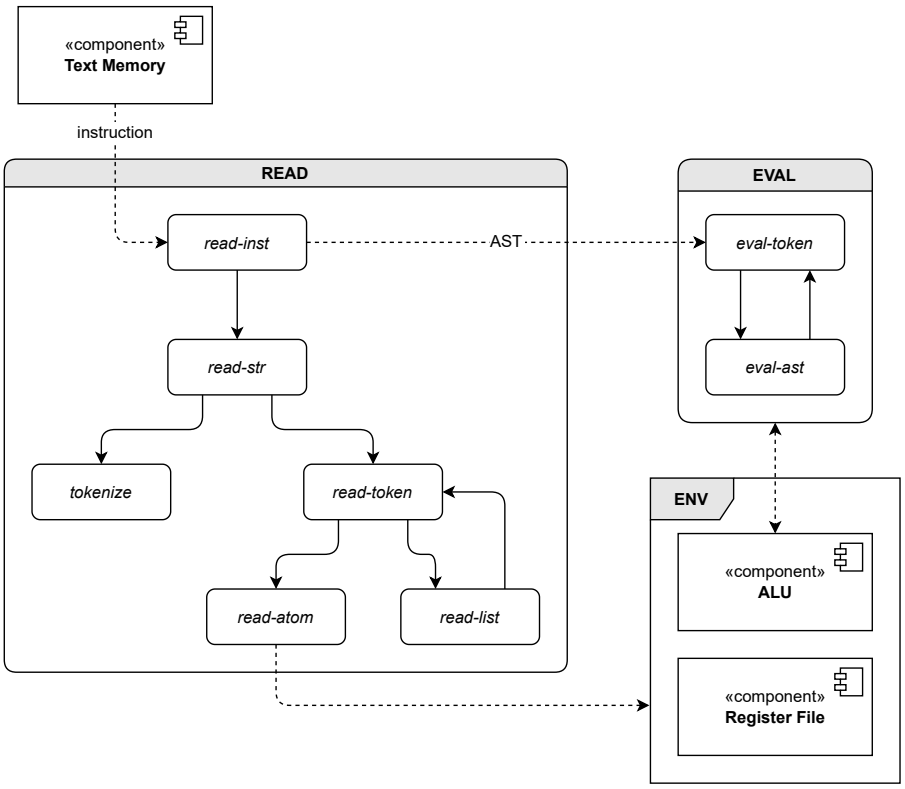


Fig. 4.2. Control Unit architecture

READ

In this stage, the interpreter first reads the instruction from the Text Memory (*read-inst*), then parses the string (*read-str*) and extracts (and stores) the tokens (*tokenize*). After that, it starts generating the AST by reading and evaluating the first token. This is done through the use of three functions:

- *read-token*: Checks if the current token is the start of an ⟨*L*⟩ token. If it is, it calls *read-list* on the next token; otherwise it calls *read-atom* on the current token. It returns the generated AST.

- *read-list*: Creates a new AST node, and iteratively calls *read-token* on all the following tokens until it reaches an ⟨*R*⟩ token, adding the resulting nodes as children of the current node. Finally, it returns the new node.

- *read-atom*: Evaluates the current token's type (by searching the token in the ENV), and returns a new AST node.

EVAL

After *read-inst* returns the AST of the instruction, it is evaluated through the use of two functions:

- *eval-token*: Reads the node's token. If it's an $\langle R \rangle$ token, it calls *eval-ast* on all the arguments (all children nodes except the first one), executes on the ALU the operator (first child) with the results as arguments, and returns the result. If it has no children, it returns a $\langle NIL \rangle$ node, and if it has only one child, it returns the child node. In any other case, it calls *eval-ast* on the node.

- *eval-ast*: Reads the node's token. If it's an $\langle R \rangle$ token, it creates a new node with an $\langle R \rangle$ token, adds as children of that node the result of calling *eval-token* on all the original node's children, and returns the new node. Otherwise, it returns the current node.

The first symbol of the AST is passed to *eval-token* to start the evaluation.

4.2.3. Program Files

Requirement SR-FC-08 states that the system must be able to load assembly-like programs.

Assembly programs allow the user to load both text memory and data memory before the execution, by dividing the program in two segments: a data segment and a text segment. For the simulator, the user will be allowed to load data in the form of strings or words[1]. The simulator will also support the use of tags, references to instruction addresses, in order to simplify the branching syntax. Finally, there will be support for comments.

For more information about the use of the program files, see Appendix A, *User manual*.

4.2.4. Compiler

According to requirement SR-FC-06, the system must be able to transform assembly-like instructions into interpreter instructions.

The compiler will go through the program file, line by line (ignoring empty lines), searching for the data and text segment keywords, and applying different procedures for each segment:

- **Data segment:** The compiler will read the tag, type and value of each datum to store, map the tags to the memory references, and store the data in data memory.

[1] For more information, see Subsection 4.2.5, *ISA Definition*.

- **Text segment:** The compiler will go instruction by instruction, translating them to LUISP-DA instructions, and temporarily with the locations of tags, if any. Afterwards, the compiler will translate the tags in the instructions by the addresses they represent, and load the finished instructions in the text memory.

4.2.5. ISA Definition

As stated in Subsection 4.1.3, the ISA will be defined in a JSON format, which will also serve as the configuration file for the simulator. For the configuration, several things need to be defined:

- The name of the architecture.
- The start addresses of data memory and text memory, as well as the end address of the data memory.
- The registers.
- The system calls' *opcodes*.
- The instruction set, mapping the assembly-like instruction syntax, the instruction code and the arguments, into the LUISP-DA language.
- The keywords for the different data types.
- The comment character.

For more information about the use of the ISA definition files, see Appendix A, *User manual*.

CHAPTER 5

IMPLEMENTATION AND DEPLOYMENT

This chapter summarizes the main decisions taken when implementing the design of the system, including the file structure of the source code (Section 5.1, *Implementation*), and describes the specifications and steps for deploying that implementation (Section 5.2, *Deployment*).

5.1. Implementation

As stated in Subsection 4.1.4, *Programming Language*, the chosen programming language for the project is ISO C++20 [40]. The chosen build system is CMake [42] plus ninja-build [43], as it provides a FOSS, cross-platform, compiler-independent build platform.

The system was implemented as described in Section 4.2, *System Architecture*. Each component was implemented as an object in a separate library, with both a header (`.hpp`) and source (`.cpp`) file, and with each component importing the other components they were dependent on. The Control Unit component was divided into three sub-objects: the AST object, a reader object in order to keep track of the current word read, and the control unit object itself.

Some extra libraries were added in order to add exception handling (*Exceptions* library) and other auxiliary functions (*Lib* library). In order to parse the JSON architecture definition files, a third-party library, *JSON for Modern C++* [44] was used.

Figure 5.1 shows the file structure of the project, relating the directories and files to the described components.

```
/
├── src/
│   ├── compiler/
│   │   ├── compiler.hpp / compiler.cpp ..... Compiler
│   │   └── CMakeLists.txt ..... Compiler library
│   ├── cpu/ ..... CPU subsystem
│   │   ├── alu.hpp / alu.cpp ..... ALU
│   │   ├── register_file.hpp / register_file.cpp ..... Register File
│   │   ├── control_unit/ ..... Control Unit
│   │   │   ├── ast.hpp / ast.cpp ..... AST definition
│   │   │   ├── reader.hpp / reader.cpp ..... read functions
│   │   │   └── control_unit.hpp / control_unit.cpp
│   │   └── CMakeLists.txt ..... CPU library
│   ├── memory/ ..... Memory subsystem
│   │   ├── data.hpp / data.cpp ..... Data Memory
│   │   ├── text.hpp / text.cpp ..... Text Memory
│   │   └── CMakeLists.txt ..... Memory library
│   ├── exceptions/ ..... Exception definitions
│   │   ├── exceptions.hpp
│   │   └── CMakeLists.txt ..... Exceptions library
│   ├── lib/ ..... Auxiliary functions
│   │   ├── string_manipulation.hpp / string_manipulation.cpp
│   │   └── CMakeLists.txt ..... Lib library
│   └── CMakeLists.txt ..... CMake configuration
```

Fig. 5.1. Project file structure

The full source code can be found in https://github.com/ldcas-uc3m/TFG.

5.2. Deployment

The technical specifications recommended for the final user to obtain the best experience from the application are:

- **Operating System:** Ubuntu 24.04 LTS (Linux distribution) / Windows 11.
- **Processor:** Intel® Core™ i3 CPU 6300 @3.8GHz or higher.
- **Random Access Memory (RAM):** 2GB or higher.
- **Storage:** 100MB or free space or higher.

- **Network:** An Internet connection is required to download the dependencies when compiling the software.
- **Software:** The following software must be installed in order to run the application:
  - CMake [42] version 3.22 or compatible.
  - Ninja-build [43] version 1.11.1 or compatible.
  - GCC [45] version 11.4 or compatible.

In order to compile and install the software, the following steps must be followed:

1. Download and decompress the source code from the repository.
2. Build and compile the source code with CMake [42] and Ninja-build [43].
3. Execute the simulator with an input ISA file.

For more information about the installation and use of the simulator, see Appendix A, *User manual*.

CHAPTER 6

VERIFICATION AND VALIDATION

The main objective of this chapter is to verify that all the requirements, specified in Chapter 3, *Analysis*, have been fulfilled.

In software engineering, verification and validation are the processes of checking that a software system meets its specifications and fulfills its intended purpose [46]. Figure 6.1 summarizes the process of software validation and verification.

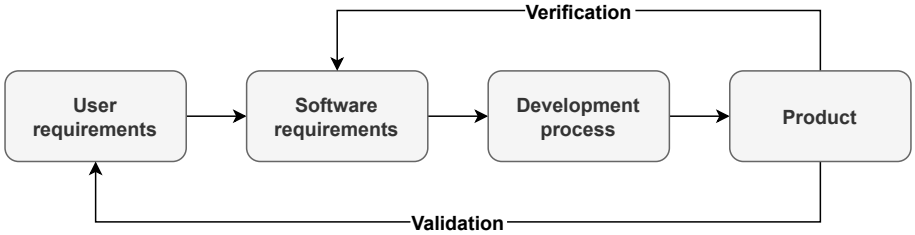


Fig. 6.1. Software verification and validation process overview

As stated in Chapter 3, *Analysis*, the customer initially sets the requirements desired for the final product, the user requirements. From there, analysts derive and specify the software requirements.

Software verification (Section 6.1, *Verification Tests*) is the process of evaluating work-products (not the actual final product) of a development phase to determine whether they meet the specified requirements for that phase (the software requirements) [46]. Software validation (Section 6.2, *Validation Tests*) is the process of evaluating the final product at the end of the development process to determine whether it satisfies the requirements specified by the user at the beginning of the project [46].

For both verification and validation, several tests were designed in order to evaluate the corresponding requirements. Each test is uniquely identified by an ID. The ID follows the format *YYY-XX*, where *XX* identifies the sequential number of the test case, starting at *01*, and *YYY* represents the type, either *VET* (verification) or *VAT* (validation). Table 6.1

provides the template used for the specification of the test case, including the description of each attribute.

TABLE 6.1
TEST TEMPLATE

YYY-XX	
Description	Test description.
Preconditions	Conditions that must be fulfilled in order to perform the test.
Procedure	Description of the steps to take in order to perform the test.
Postconditions	Conditions that must be fulfilled after performing the test in order to pass.
Origin	Requirements that originated this test.
Evaluation	Result of the test (*OK* or *Error*).

6.1. Verification Tests

In order to perform the verification tests, a dynamic process during the development phase of the software was followed. With these tests the aim is to answer the question "*Are we building the product correctly?*".

Table 6.11 shows the traceability between the software requirements and the verification tests.

[2] As LUISP-DA is a language from the LISP family, and its formal definition was provided (see Subsection 4.2.1, *Instruction Definition Language*), it is a validatable language.

TABLE 6.2
TEST VET-01

VET-01	
Description	Verify that the system is able to simulate a predefined ISA.
Preconditions	The software is installed.
Procedure	1. Implement a predefined ISA in an architecture definition file, including the configuration of the different components. 2. Write a program that uses that ISA, and implements all basic operations: arithmetic, logical, branching, memory manipulation, and I/O. 3. Load the architecture file into the simulator. 4. Load and fully execute the program.
Postconditions	The results from the execution of the program are the same as the theoretical expected results.
Origin	SR-FC-01, SR-FC-02, SR-FC-04, SR-FC-06, SR-FC-08, SR-NF-07, SR-FC-09
Evaluation	OK

TABLE 6.3
TEST VET-02

VET-02	
Description	Verify that the system is able to detect syntactical errors.
Preconditions	The software is installed, and a ISA is loaded.
Procedure	1. Write a program that uses the loaded ISA, with syntactical errors. 2. Load and fully execute the program.
Postconditions	The software stops the execution and warns the user of the error.
Origin	SR-FC-10
Evaluation	OK

TABLE 6.4
TEST VET-03

VET-03	
Description	Verify that the system allows for a step-by-step execution.
Preconditions	The software is installed, and a ISA is loaded.
Procedure	1. Write a program that uses the loaded ISA, with operations on registers and memory. 2. Load and the program. 3. View the state of the registers and memory. 4. Change the representation format. 5. Execute an instruction. 6. View the state registers and memory. 7. Repeat until the program finishes.
Postconditions	The state of the registers and memory is updated correctly after each instruction, and the program is fully executed as expected.
Origin	SR-FC-03, SR-FC-06, SR-FC-07, SR-FC-09, SR-FC-11
Evaluation	OK

TABLE 6.5
TEST VET-04

VET-04	
Description	Verify that the system allows to stop the execution and load a new program.
Preconditions	The software is installed, and a ISA and program are loaded.
Procedure	1. Execute an instruction. 2. Stop the execution of the program. 3. Load and execute a new program.
Postconditions	The new program is loaded and executed as expected, and the simulator is reset.
Origin	SR-FC-08, SR-FC-09, SR-FC-12
Evaluation	OK

TABLE 6.6
TEST VET-05

VET-05	
Description	Verify that the system offers a REPL environment for the LUISP-DA language.
Preconditions	The software is installed, and a ISA is loaded.
Procedure	1. Enter the REPL mode. 2. Write and evaluate some LUISP-DA expressions, including register and memory manipulation.
Postconditions	The expressions are evaluated correctly, and the state is modified correctly.
Origin	SR-FC-13, SR-FC-05
Evaluation	OK

TABLE 6.7
TEST VET-06

VET-06	
Description	Verify that the software can be executed natively in many platforms.
Preconditions	The software's source code is downloaded.
Procedure	1. Compile the source code for multiple platforms. 2. Execute the simulator in those platforms.
Postconditions	The expressions are evaluated correctly, and the state is modified correctly.
Origin	SR-NF-01, SR-NF-02
Evaluation	OK

TABLE 6.8
TEST VET-07

VET-07	
Description	Verify that the software is FOSS and publicly available.
Preconditions	N/A
Procedure	1. Access the software's source code website. 2. Verify the software's license.
Postconditions	The source code is publicly available, and it uses a FOSS license.
Origin	SR-NF-03, SR-NF-04
Evaluation	OK

TABLE 6.9
TEST VET-08

VET-08	
Description	Verify that the software is easy to use.
Preconditions	The software is installed.
Procedure	1. Read the software's user manual. 2. Execute all basic functionalities of the software: create an ISA and program, execute it fully, instruction by instruction, stop and change programs and use the REPL.
Postconditions	All actions were successfully performed.
Origin	SR-NF-05, SR-NF-06, SR-NF-07, SR-NF-09
Evaluation	OK

TABLE 6.10
TEST VET-09

VET-09	
Description	Verify that the instruction definition language is validatable.
Preconditions	N/A
Procedure	1. Validate LUISP-DA expressions.
Postconditions	The language is validatable.
Origin	SR-NF-08
Evaluation	OK[2]

TABLE 6.11

TRACEABILITY BETWEEN SOFTWARE REQUIREMENTS AND VERIFICATION TESTS

	SR-FC-01	SR-FC-02	SR-FC-03	SR-FC-04	SR-FC-05	SR-FC-06	SR-FC-07	SR-FC-08	SR-FC-09	SR-FC-10	SR-FC-11	SR-FC-12	SR-FC-13	SR-NF-01	SR-NF-02	SR-NF-03	SR-NF-04	SR-NF-05	SR-NF-06	SR-NF-07	SR-NF-08	SR-NF-09
VET-01	•	•		•		•		•	•											•		
VET-02										•												
VET-03			•			•	•		•		•											
VET-04								•	•			•										
VET-05					•								•									
VET-06														•	•							
VET-07																•	•					
VET-08																		•	•	•		•
VET-09																					•	

6.2. Validation Tests

To perform the validation tests, the final software was tested, comparing it against the user's needs specified in Subsection 3.2.1, *User Requirements*. With these tests the aim it to answer the question "*Have we built the right product?*".

Table 6.19 shows the traceability between the software requirements and the verification tests.

TABLE 6.12
TEST VAT-01

VAT-01	
Description	Validate that the system can execute any program defined in any ISA, through the CLI.
Preconditions	An ISA is designed.
Procedure	1. Define an ISA. 2. Execute the program with the ISA definition.
Postconditions	The program is executed correctly.
Origin	UR-CA-01, UR-CA-02, UR-CA-03, UR-CA-04, UR-CA-05
Evaluation	OK

TABLE 6.13
TEST VAT-02

VAT-02	
Description	Validate that the system recognizes errors.
Preconditions	An ISA is designed.
Procedure	1. Define a program with syntactical errors. 2. Try to execute the program.
Postconditions	The system notifies the user of the errors.
Origin	UR-CA-06, UR-CA-07
Evaluation	OK

TABLE 6.14
TEST VAT-03

VAT-03	
Description	Validate that the system can be executed either instruction by instruction, or fully.
Preconditions	Design an ISA and a program.
Procedure	1. Execute the program instruction by instruction. 2. Change the representation. 3. Stop it and reset the program. 4. Execute the program fully.
Postconditions	The program is executed correctly.
Origin	UR-CA-04, UR-CA-07, UR-CA-08
Evaluation	OK

TABLE 6.15
TEST VAT-04

VAT-04	
Description	Validate that the system contains a REPL environment for the LUISP-DA language.
Preconditions	Define an ISA.
Procedure	1. Test LUISP-DA expressions in the REPL environment.
Postconditions	The expressions are evaluated as expected.
Origin	UR-CA-09
Evaluation	OK

TABLE 6.16
TEST VAT-05

VAT-05	
Description	Validate that the software is FOSS .
Preconditions	N/A.
Procedure	1. See the source code's license.
Postconditions	The license is FOSS.
Origin	UR-RE-02
Evaluation	OK

TABLE 6.17
TEST VAT-06

VAT-06	
Description	Validate that the system can be executed natively.
Preconditions	The source code is downloaded.
Procedure	1. Compile the software for multiple architectures. 2. Execute all basic functionalities.
Postconditions	All actions were successfully performed.
Origin	UR-RE-01
Evaluation	OK

TABLE 6.18
TEST VAT-07

VAT-07	
Description	Validate that the system is easy to use.
Preconditions	The system is installed.
Procedure	1. Read the software's user manual. 2. Execute all basic functionalities.
Postconditions	All actions were successfully and easily performed.
Origin	UR-RE-03, UR-RE-04, UR-RE-05, UR-RE-06
Evaluation	OK

TABLE 6.19

TRACEABILITY BETWEEN USER REQUIREMENTS AND VALIDATION TESTS

	UR-CA-01	UR-CA-02	UR-CA-03	UR-CA-04	UR-CA-05	UR-CA-06	UR-CA-07	UR-CA-08	UR-CA-09	UR-RE-01	UR-RE-02	UR-RE-03	UR-RE-04	UR-RE-05	UR-RE-06
VAT-01	•	•	•	•	•										
VAT-02						•	•								
VAT-03				•			•	•							
VAT-04									•						
VAT-05										•					
VAT-06															
VAT-07											•	•	•	•	

CHAPTER 7

PROJECT PLAN

This chapter presents an overview of the development and logistics of the project. The planning of the project (Section 7.1, *Planning*) is detailed, analyzing its budget and overall cost (Section 7.2, *Budget*). Furthermore, it analyzes the different legislation and regulations that may apply to the project (Section 7.3, *Regulatory Framework*), and discusses the socio-economic environment in which it was carried out. (Section 7.4, *Socio-Economic Environment*).

7.1. Planning

This section details the project's planning, by describing the followed methodology and detailing the duration of each part.

7.1.1. Methodology

Due to the characteristics of the design, the development process was divided into four iterations:

I. **Memory and Register File**. This iteration's goal was to implement the Text Memory, Data Memory, and Register File components separately.

II. **Control Unit and ALU**. This phase consists on implementing the Control Unit and ALU components, and connecting them to all the previous components. The goal of this iteration is to be able to simulate the full CPU and memory, and interpret LUISP-DA instructions.

III. **Compiler**. This iteration consists on implementing the Compiler component, with the purpose of being able to load ISAs and interpret full programs.

IV. **CLI**. The final iteration implements the CLI component, adding features such as the REPL environment and the whole UI.

An iterative methodology was chosen, based on Bohem's spiral model [47], in order to ensure that, before implementing a component, all components that it depends on are correctly implemented. It also simplifies the development life cycle and makes it more flexible, as it allows the developer to go back and modify previous elements, and encourages prototyping.

The life cycle development process of this model (Figure 7.1) has four phases, which are repeated during the different iterations of the model. These phases are:

1. **Planning:** The user requirements are gathered, and the iteration's objectives are determined.

2. **Analysis:** An analysis of the user requirements is performed in order to identify potential risks, and the test cases are designed.

3. **Development and testing:** The architecture is designed, implemented, and the tests are performed.

4. **Evaluation:** The software is evaluated with the client, in order to provide feedback. In this specific case, the tutor acted as the client. This is the critical task of the life cycle, as the iteration isn't finished until the software is approved.

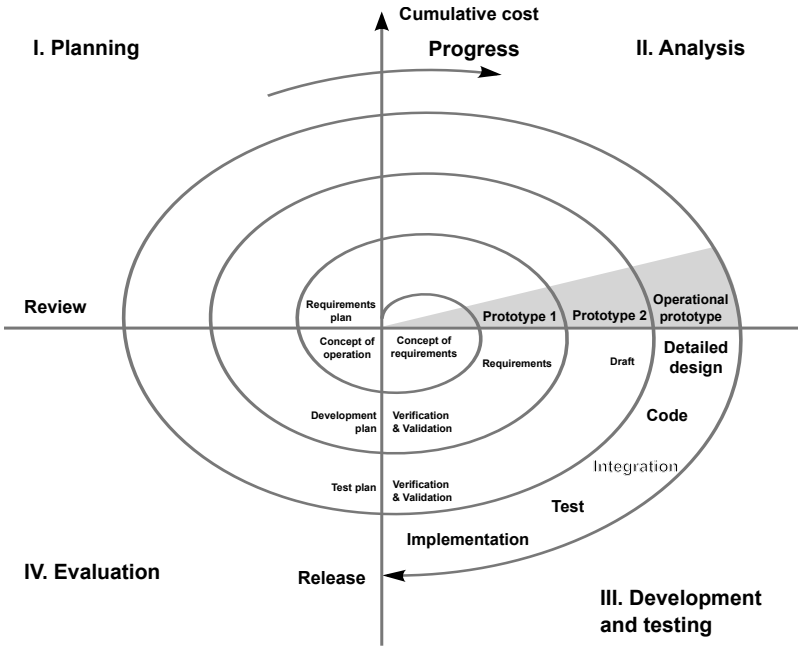


Fig. 7.1. Spiral model life cycle

7.1.2. Time Estimation

The time estimation of the project was designed with the use of a Gantt chart [48] (Figure 7.2). This diagram shows all the performed tasks in each iteration of the life cycle, plus an extra documentation task in each iteration for the drafting of this thesis. A final 'Report' task was added to represent the time spent finishing this report.

The project had a total duration of 9 months, averaging 35 hours per month, around 10 hours per week (without taking into account days off, holidays, sick days, and other drawbacks). Therefore, the total time dedicated to this project, excluding the tutor's work, is 315 hours.

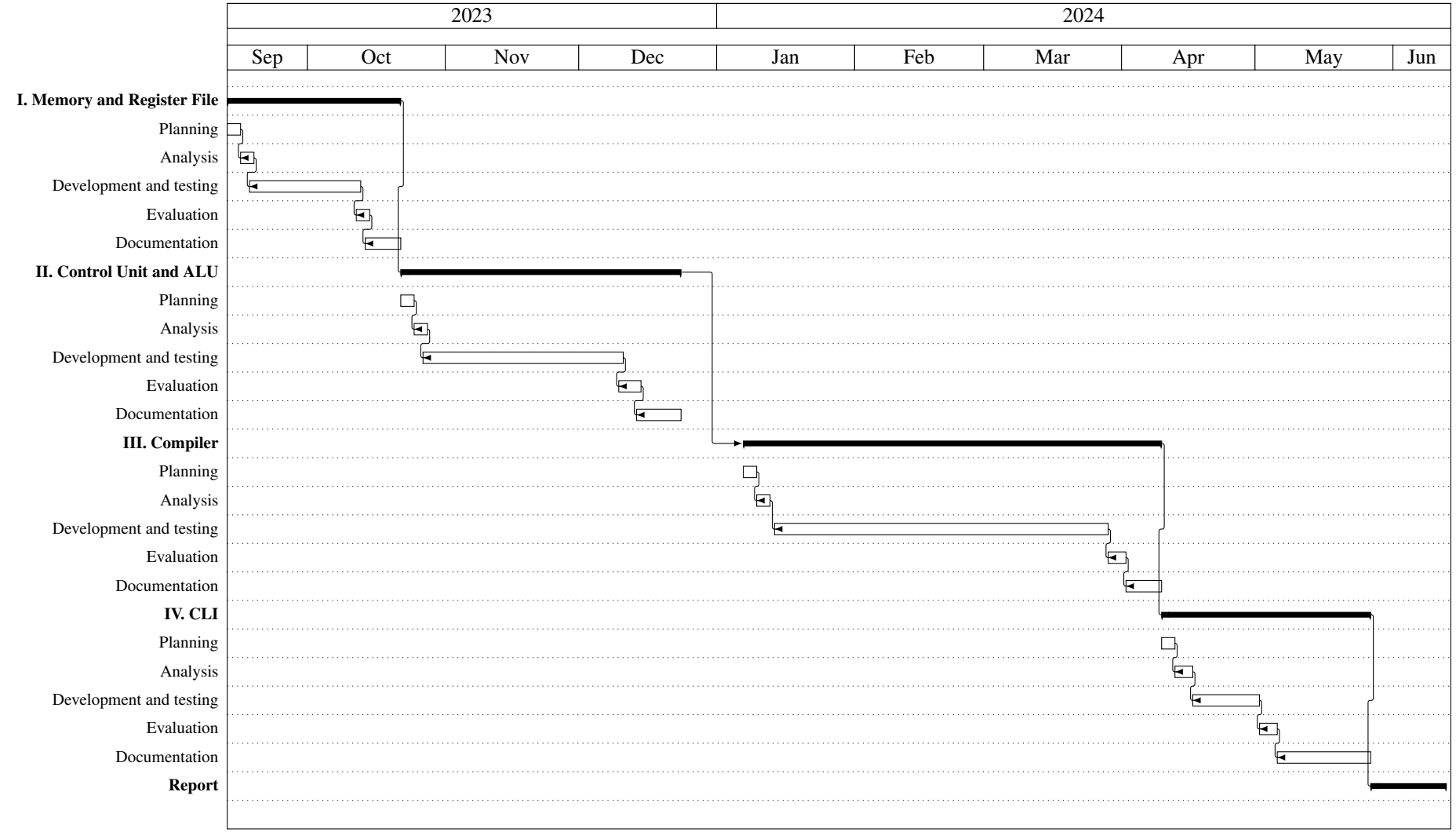


Fig. 7.2. Gantt chart

7.2. Budget

This section details the project's budget, based on the time estimation and planning described in Section 7.1, *Planning*. Table 7.1 summarizes the main characteristics of the project, including the total budget.

TABLE 7.1
PROJECT INFORMATION

Title	*Analysis, Design and Implementation of a Didactic and Generic Assembly Language Simulator*
Author	Luis Daniel Casais Mezquida
Department	Departamento de Informática
Start date	13th of September of 2023
End date	13th of June of 2024
Duration	9 months
Total budget	12.161,84 €

The different costs will be divided in two parts: direct costs (those associated with personnel and equipment), and indirect costs (those with an indirect influence to the project). These costs don't include taxes, which will be included in the costs summary on Subsection 7.2.3.

7.2.1. Direct Costs

Direct costs are those that are directly related with the development of the project. These can be divided into two groups:

- **Personnel costs:** These vary in relation to the qualifications, experience, and geographical location of each member.
- **Equipment costs:** These are all the costs associated to the tools required for developing the software, mainly hardware and software tools.

Personnel costs can be divided between four different roles:

- **Project manager:** Manages the project's schedule and provides feedback.
- **Analyst:** Analyzes the user's requirements, designs the architecture of the system, and writes documentation.
- **Programmer:** Implements the required functionalities.

- **Tester:** Designs and performs the tests for the different functionalities.

The tutor played the role of project manager, while the student played the rest of the roles. Table 7.2 shows the direct costs caused by each of the roles, and the total personnel costs.

TABLE 7.2
PERSONNEL COST

Role	Hours	Cost per hour	Total
Project Manager	40 h	65,00 €/h	2.600,00 €
Analyst	80 h	35,00 €/h	2.800,00 €
Programmer	175 h	30,00 €/h	5.250,00 €
Tester	60 h	20,00 €/h	1.200,00 €
Total	385 h		**11.850,00 €**

Equipment costs refer to those caused by equipment acquisition and usage. For software, all the software tools used for this project were FOSS. The chargeable cost, C, for each item is calculated using Equation (7.1), where:

- c is the cost of the item.
- d is the time the item has been used.
- u is the percentage of the total time the item was used for the project.
- D is the item's depreciation period.

$$C = \frac{c \cdot u \cdot d}{D} \tag{7.1}$$

Table 7.3 shows the chargeable costs of each equipment item, and the total equipment costs.

7.2.2. Indirect Costs

Indirect costs are those that are present during the development process, but cannot be assigned directly to any product.

For the energy consumption, it is assumed that the laptop's energy usage averages 80W, and the monitor and mouse average 20W, and that they were used during the whole development process, totaling 360h. Therefore, the total energy consumed is $100W \cdot 360h = 3.600Wh$. The internet plan includes a 600mb optic fiber connection, 31,80 €/month, which is shared between three people, two of them not included in the project, therefore the cost applicable to the project is a third of that.

TABLE 7.3
EQUIPMENT COST

Item	Cost (c)	Usage (u)	Dedication (d)	Depreciation (D)	Chargeable cost (C)
Laptop	649,00 €	60 %	9 months	72 months	48,68 €
Monitor	124,99 €	40 %	9 months	60 months	74,99 €
Mouse	79,95 €	50 %	9 months	36 months	9,99 €
Adapter	39,90 €	70 %	9 months	48 months	5,24 €
Video cable	5,75 €	50 %	9 months	48 months	0,54 €
Software	0,00 €	40 %	9 months	120 months	0,00 €
Total	899,59 €				**139,44 €**

TABLE 7.4
INDIRECT COSTS

Resource	Unitary cost	Units	Total
Electricity	0.14 €/kWh	3.600 Wh	0,50 €
Internet	10,60 €/month	9 months	95,40 €
Transportation	8 €/month	9 months	72,00 €
Total			**172,40 €**

7.2.3. Costs Summary

Table 7.5 includes a summary of the costs of the whole project.

TABLE 7.5
COSTS SUMMARY

Personnel	11.850,00 €
Equipment	139,44 €
Indirect costs	172,40 €
Total	**12.161,84 €**

Therefore, the total cost of the project is **12.161,84 € (TWELVE THOUSAND ONE HUNDRED SIXTY-ONE EURO AND EIGHTY-FOUR CENTS)**.

7.2.4. Project Offer Proposal

Table 7.6 details an offer proposal for the project. This proposal includes the estimated risks, expected benefits, and taxes. After applying all these concepts, the final cost of this project, in case it is presented to a third-party client, is **20.307,84 € (TWENTY THOUSAND THREE HUNDRED SEVEN EURO AND EIGHTY-FOUR CENTS)**.

TABLE 7.6
OFFER PROPOSAL

Concept	Increment	Partial cost	Aggregated cost
Project cost	–	12.161,84 €	12.161,84 €
Risk	20 %	2.432,37 €	14.594,21 €
Benefits	15 %	2.189,13 €	16.783,34 €
Tax	21 %	3.524,50 €	20.307,84 €
Total	67 %		**20.307,84 €**

7.3. Regulatory Framework

This section details and discusses the different regulation that may apply to the project.

7.3.1. Applicable Legislation

As the software is executed locally, and it does not transmit or use private data from the users, no data protection laws apply. The software is also used for educational purposes, meaning there are no risks involved in the execution of the software and no other regulatory compliance is required.

7.3.2. Technical Standards

The software makes use of two technical standards:

- ISO/IEC 14882:2020 [40], the standard for the C++20 programming language, which is the one the software is implemented in.
- ISO/IEC 21778:2017 [32], the standard for the JSON data format, which is used to store the ISA definition files.

7.3.3. Licenses

The software uses and redistributes one third-party library, *JSON for Modern C++* [44]. This library is licensed under the MIT License [49], which allows permission to use, copy, modify, or redistribute source code without restrictions or limitations.

The software itself is licensed under the GNU General Public License, version 3 (GPLv3) [50], in order to allow free use and modification of the source code, while ensuring it will always remain that way. Furthermore, the software is publicly available at https://github.com/ldcas-uc3m/TFG.

7.4. Socio-Economic Environment

In 2010, RISC-V appears as an open, modular ISA [51]. This comes at a time when the commercial war between the USA and China has just started, affecting the exportation of microchips. This forces countries and corporations to start investing in manufacturing their own chips, and an open standard such as the mentioned RISC-V can act as a base for designing and implementing their own ISAs.

Furthermore, recent technological advances in cloud computing, Internet of Things (IoT), and Artificial Inteligence (AI) greatly benefit from hardware acceleration, which pushes for the design of new, more specific ISAs that need to be tested and validated.

The recent advancements (and overall push) in computing has also put a great focus in cybersecurity, as our increasing dependence on computing resources creates greater security risks. Many of the vulnerabilities and exploits occur at the assembly level which, together with the constant search for efficiency, creates a high demand for people with deep knowledge of assembly language programming.

As discussed in Chapter 2, *State of the Art*, there are not many didactic and generic assembly language simulators that can be used to introduce people to assembly language programming and ISAs design. Our proposal aims to foment these areas of computer science by providing a simple and easy to understand tool for anyone to start exploring different instruction sets, while also aiding people with the sometimes difficult process of learning assembly language programming.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

This chapter presents both the conclusions of the project Section 8.1, *Project Conclusions*, and personal conclusions Section 8.2, *Personal Conclusions*. It also highlights some of the contributions that were made during this project that are not directly related to it Section 8.3, *Additional Contributions*, and discusses what future work could be done to the project Section 8.4, *Future Work*.

8.1. Project Conclusions

This document has described the analysis, design, and implementation of a generic and didactic assembly language simulator. This project presents a simple, open-source, and intuitive simulator, focused on providing the user an intuitive knowledge of ISA design and assembly language programming.

As stated in Section 1.2, *Objectives*, the main objective in this project was to create a simulator that, unlike existing ones, allows the user to define its own ISAs, defining instructions through a simple and easy to understand language, and execute any assembly program.

All secondary objectives were also fulfilled:

- **O1:** The software simulates a simple and generic computer, with registers, memory, and a processor that executes LUISP-DA, a simple domain specific language.
- **O2:** LUISP-DA is a simple programming language that allows to program all the components of the simulated computer.
- **O3:** The simulator allows the user to view the state of the computer of the at each clock cycle.

- **O4:** The user can execute any assembly language program in the defined ISA.
- **O5:** The software can be executed natively in many platforms.

One of the main obstacles encountered while developing this project, as shown on the Gantt chart (Figure 7.2), was the design and implementation of the Control Unit (LUISP-DA interpreter). As stated in Subsection 4.1.2, *Interpreter*, several design options were considered and explored, until reaching the final design. This design also took considerable time, as it required a good understanding of the programming language, C++20, in order to implement. Greater knowledge and experience in the field of compilers and C++20 would have decreased greatly development time, and may have resulted in an even better end product.

8.2. Personal Conclusions

This project has been a great opportunity for me personally, as it has allowed not only to make use of all the knowledge I have gained throughout this degree, but also to gain more of it.

The main sources of knowledge for this project have been the *Computer Structure*, *Computer Architecture*, and *Computer Organization* courses, which coincidentally are also amongst my favorites. These subjects taught me about how computers are made, about ISAs, and about assembly language. Other source of knowledge would have been the *Language Processors* course, focused on compilers, but unfortunately due to my academic plan, I wasn't able to attend it. It is also worth it to mention other courses, such as *Programming*, *Data Structure and Algorithms*, *Functional Programming*, and *Automata Theory and Formal Language*, whose knowledge also helped be during this process.

A lot of knowledge was gained from the implementation of the software and the writing of this document. I started this project with very little knowledge and experience on C++, and learned the language through the course of it. The same goes for CMake, and the overall organization of C++ projects. Writing this document in LaTeX has also offered me the chance to greatly deepen my knowledge of this tool, something that I'm sure will be useful in the future.

8.3. Additional Contributions

As mentioned before, this document was created and written in LaTeX [52]. Not only the source code of this report is publicly available at https://github.com/ldcas-uc3m/TFG, but the code was structured in such a way that it is easily reusable, through the use of a LaTeX class template file that contains the configuration for the formatting of the document. This template was based on the template document provided by the Universidad

Carlos III de Madrid's library [53], and follows the bachelor's thesis guidelines for this university [54].

The class template, along with a template for the organization of the whole report, and documentation, was made publicly available at https://github.com/ldcas-uc3m/thesis-template, in order to provide a better and easier to use bachelor thesis template for future engineering students.

The source code for this document also makes use of a fork (modified version) of Javier López Gómez's SRS package [55]. This package automates the process of generating tables for software engineering tasks, such as requirements, use cases, traceability matrices, etc. This fork changes the style of the generated tables and adds several features such as support for the English language, generation of template tables, automatic captions, and hyperlinks on traceability matrices. The source code for the fork is also publicly available at https://github.com/ldcas-uc3m/srs-latex, in order to aid computer science students to perform these tasks in LaTeX.

8.4. Future Work

This project provided a basic simulator, but there are several lines of work that could be explored, in order to improve the system:

- As mentioned in Subsection 4.1.4, *Programming Language*, C++ allows to compile the source code to WebAssembly [39]. Further research in this topic could allow the simulator to be executed in browsers, making it even more accessible.
- The simulator could also offer support for more types of operations: floating point, bitwise, etc.
- Support for even more assembly language features could be added, such as non-aligned memory, multiple names for the same registers, and inline comments.
- Many other simulators implement ease-of-use features, such as breakpoints, and save states. These features and others, such as not requiring to reload the whole simulator to change the ISA, or more formatting options for the display, would be appreciated by the users.
- The base system is robust enough for it to be built upon, specially as it's implemented in C++. It currently uses a CLI, but it could also include a GUI implemented in Qt [56]. It could also be converted into a web service, executing on a server instead of on the client. The system could also be converted into a Python extension module [57], opening even more possibilities.

Some aspects of the implementation could also be improved:

- Currently, the LUISP-DA interpreter stores all of its data (tokens) in `std::string`, due to the fact that a token can hold either strings, integers, or boolean values. This forces static castings (from and to string) for any operation, which creates a great overhead. One possible solution is the use of `std::variant` [58], a data structure that can hold many types.
- Instead of C-style libraries, the implementation could make use of C++20 modules [59][3].
- The C++23 ISO standard [60] is in the approval phase, and the system could benefit from some new features the standard provides.

[3] At the time of writing, this feature is not fully implemented in any of the C++ compilers.

REFERENCES

[1] W. F. Decker, "A modern approach to teaching computer organization and assembly language programming," *SIGCSE bulletin*, vol. 17, no. 4, pp. 38–44, 1985. DOI: 10.1145/126445.126457.

[2] Y.-C. Hung, "Combining Self-Explaining With Computer Architecture Diagrams to Enhance the Learning of Assembly Language Programming," *IEEE transactions on education*, vol. 55, no. 4, pp. 546–551, 2012. DOI: 10.1109/TE.2012.219517.

[3] D. Klahr and S. McCoy Carver, "Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer," *Cognitive Psychology*, vol. 20, no. 3, pp. 362–404, 1988. DOI: 10.1016/0010-0285(88)90004-7.

[4] S. Tramm. "Intel 8080 CPU Emulator." (2010), [Online]. Available: https://st.sdf-eu.org/i8080/index.html (visited on 04/22/2024).

[5] S. Barrachina Mir, G. Fabregat Llueca, J. C. Fernández Fernández, and G. León Navarro, "Utilizando ARMSim y QtARMSim para la docencia de Arquitectura de Computadores," *ReVisión*, vol. 8, no. 3, pp. 17–35, 2015.

[6] W. J. Song, *Kite: An Architecture Simulator for RISC-V Instruction Set*, Yonsei University, [Online]. Available: https://casl.yonsei.ac.kr/kite, May 2019.

[7] D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface (RISC-V edition)* (Issn Series), First edition. 2018.

[8] Computer Architecture and Systems Lab, Yonsei University. "Kite: Architecture Simulator for RISC-V Instruction Set." (2019), [Online]. Available: https://github.com/yonseicasl/Kite (visited on 04/24/2024).

[9] P. Higginson. "ARMLite Simulator." (2020), [Online]. Available: https://peterhigginson.co.uk/ARMlite/ (visited on 04/24/2024).

[10] R. Pawson and P. Higginson, *Assembly Language Programming* (Computer Science from the Metal Up), V1.0.0. Metal Up, 2020. [Online]. Available: https://metalup.org/al/description.html.

[11] "AQA – education charity providing GCSEs, A-levels and support." (2012), [Online]. Available: https://www.aqa.org.uk/ (visited on 04/25/2024).

[12] "Assembly language instruction set: AS and A-level Paper 2." (2021), [Online]. Available: https://filestore.aqa.org.uk/resources/computing/AQA-75162-75172-ALI.PDF (visited on 04/25/2024).

[13] M. Gardner, "The fantastic combinations of John Conway's new solitaire game "life"," *Scientific American*, vol. 223, pp. 120–123, 1970. DOI: 10.1038/scientificamerican1070-120.

[14] REMS Project. "Sail architecture definition language." (2019), [Online]. Available: https://github.com/rems-project/sail/ (visited on 04/29/2024).

[15] P. Sewell. "REMS." (2013), [Online]. Available: https://www.cl.cam.ac.uk/~pes20/rems/ (visited on 04/29/2024).

[16] A. Armstrong *et al.*, "ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS," *Proceedings of the ACM on programming languages*, vol. 3, no. POPL, pp. 1–31, Jan. 2019. DOI: 10.1145/3290384.

[17] D. Camarmas Alonso, F. García Carballeira, E. Del Pozo Puñal, and A. Calderón Mateos. "CREATOR – didaCtic and geneRic assEmbly prograMming simulaTOR." (2019), [Online]. Available: https://creatorsim.github.io/ (visited on 04/29/2024).

[18] D. Camarmas Alonso, F. García Carballeira, E. Del Pozo Puñal, and A. Calderón Mateos, "CREATOR: Simulador didáctico y genérico para la programación en ensamblador," in *XXXI Jornadas de Paralelismo*, Sociedad de Arquitectura y Tecnología de Computadores, Málaga, Spain: Zenodo, Jul. 2021. DOI: 10.5281/zenodo.5130302.

[19] "IEEE Guide for Software Requirements Specifications," Institute of Electrical and Electronics Engineers, IEEE Std. 830-1984, 1984. DOI: 10.1109/IEEESTD.1984.119205.

[20] S. Cook *et al.*, "Unified Modeling Language specification," Object Management Group, Standard, version 2.5.1, Dec. 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1.

[21] "IEEE Standard for Microprocessor Assembly Language," Institute of Electrical and Electronics Engineers, IEEE Std. 694-1985, 1985. DOI: 10.1109/IEEESTD.1985.81632.

[22] LuaLab. "The Programming Language Lua." (1993), [Online]. Available: https://www.lua.org/ (visited on 05/20/2024).

[23] Neovim team. "Lua Guide – Neovim docs." (2016), [Online]. Available: https://neovim.io/doc/user/lua-guide.html (visited on 05/20/2024).

[24] LuaTeX team. "LuaTeX." (2007), [Online]. Available: https://www.luatex.org/ (visited on 05/20/2024).

[25] A. Rossberg, Ed. "WebAssembly text format." (2022), [Online]. Available: https://webassembly.github.io/spec/core/text/ (visited on 05/29/2024).

[26] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part I," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, Apr. 1960. DOI: 10.1145/367177.367199.

[27] T. Æ. Mogensen, *Basics of compiler design*, Anniversary edition. Torben Ægidius Mogensen, 2009.

[28] J. R. Levine, D. Brown, and T. Mason, *Lex & yacc*, 2nd ed. O'Reilly, 1992.

[29] J. R. Levine, *Flex & bison*, 1st ed. O'Reilly Media, 2009.

[30] T. J. Parr, *Language implementation patterns: create your own domain-specific and general programming languages* (Pragmatic programmers), 1st edition. 2010.

[31] J. Martin. "Make-A-Lisp." (2015), [Online]. Available: https://github.com/kanaka/mal (visited on 10/05/2023).

[32] "Information technology – The JSON data interchange syntax," International Organization for Standardization, International Standard ISO/IEC 21778:2017, Mar. 2017.

[33] YAML Language Development Team. "YAML Ain't Markup Language (YAML) version 1.2." (Nov. 2021), [Online]. Available: https://yaml.org/spec/1.2.2/ (visited on 05/22/2023).

[34] B. W. Kernighan and D. M. Ritchie, *The C programming language* (Prentice-Hall software series), 2nd ed. Prentice-Hall, 1988.

[35] B. Stroustrup, *The C++ programming language*, 4th ed. Addison-Wesley, 2013.

[36] S. Klabnik and C. Nichols, *The Rust programming language*, Second edition. No Starch Press, 2023.

[37] A. Donovan, *The Go Programming Language*, 1st edition. 2015.

[38] L. S. Vailshery. "Most used programming languages among developers worldwide as of 2023." (2023), [Online]. Available: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/ (visited on 05/23/2024).

[39] A. Haas *et al.*, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017, vol. 52, Barcelona, Spain: Association for Computing Machinery, 2017, pp. 185–200. DOI: 10.1145/3140587.3062363.

[40] "Programming Languages – C++," International Organization for Standardization, International Standard ISO/IEC 14882:2020, Mar. 2020.

[41] J. W. Backus *et al.*, "Revised report on the algorithmic language ALGOL 60," *Communications of the ACM*, vol. 6, no. 1, pp. 1–17, Jan. 1963. DOI: 10.1145/366193.366201.

[42] Kitware. "CMake." (2000), [Online]. Available: https://cmake.org/ (visited on 06/03/2024).

[43] Ninja-build. "Ninja." (2013), [Online]. Available: https://ninja-build.org/ (visited on 06/03/2024).

[44] N. Lohmann. "JSON for Modern C++." (2015), [Online]. Available: https://json.nlohmann.me/ (visited on 06/03/2024).

[45] Free Software Foundation. "GCC, the GNU Compiler Collection." (1987), [Online]. Available: https://gcc.gnu.org/ (visited on 06/03/2024).

[46] "IEEE Standard for System and Software Verification and Validation," Institute of Electrical and Electronics Engineers, IEEE Std. 1012-2012, 1984. DOI: 10.1109/IEEESTD.2012.6204026.

[47] B. W. Boehm, "A spiral model of software development and enhancement," *IEEE Computer*, vol. 21, no. 5, pp. 61–72, 1988. DOI: 10.1109/2.59.

[48] W. Clark, W. Polakov, and F. Trabold, *The Gantt Chart: A Working Tool of Management* (Ronald manufacturing management and administration series). Ronald Press Company, 1922.

[49] Open Source Initiative. "The MIT License." (1998), [Online]. Available: https://opensource.org/license/MIT (visited on 06/08/2024).

[50] Free Software Foundation. "GNU General Public License." (2007), [Online]. Available: https://www.gnu.org/licenses/gpl-3.0.en.html (visited on 12/26/2023).

[51] RISC-V International. "History of RISC-V." (2020), [Online]. Available: https://riscv.org/about/history/ (visited on 06/08/2024).

[52] L. Lamport, *LATEX: A Document Preparation System*. Addison-Wesley, 1986.

[53] Biblioteca UC3M. "Bachelor thesis template - UC3M - IEEE Style." (2021), [Online]. Available: https://www.overleaf.com/latex/templates/bachelor-thesis-template-uc3m-ieee-style/rtmtnzvxjnwt (visited on 02/07/2024).

[54] Biblioteca UC3M. "Bachelor Thesis UC3M: Writing your TFG." (2021), [Online]. Available: https://uc3m.libguides.com/en/TFG/writing (visited on 05/27/2024).

[55] J. López Gómez. "SRS-latex-uc3m." (2019), [Online]. Available: https://github.com/jalopezg-git/SRS-latex-uc3m (visited on 10/15/2023).

[56] QT Group. "QT framework." (1995), [Online]. Available: https://www.qt.io/product/framework (visited on 06/10/2023).

[57] The Python Software Foundation. "Extending python with C or C++." (2009), [Online]. Available: https://docs.python.org/3/extending/extending.html (visited on 06/10/2023).

[58] A. Naumann, "Variant: A type-safe union for C++17," The C++ Standards Committee, C++ standard feature proposal P0088R3, version 8, Jun. 2016. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0088r3.html.

[59] R. Smith, "Merging modules," The C++ Standards Committee, C++ standard feature proposal P1103R3, Feb. 2019. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1103r3.pdf.

[60] "Programming Languages – C++," International Organization for Standardization, International Standard Draft ISO/IEC PRF 14882, 2023.

GLOSSARY

A

AI

A technology that enables computers to simulate human intelligence and problem-solving capabilities. 79

ALU

The part of the processor that carries out the different operations. 20

arithmetic operation

A mathematical operation applied to numeric values. 20

assembly directive

Directions for the assembler to perform some specific action or change a specific setting. 8

assembly language

A type of low-level programming language intended to communicate directly with the computer's hardware. 1–3, 5, 8, 9, 11, 22, 37, 39, 41, 48, 50, 51, 79, 81–83

assembly simulator

A piece of software that simulates a computer that executes an assembly language see. 5

AST

A data structure used to represent the structure of a program or instruction. 38

B

branch

The act of switching execution to a different instruction sequence as a result of evaluating an expression. 20, 46, 50, 59

build system

A computer program that orchestrates the compilation of a software system. 53

C

CLI

A way of interacting with a computer program by inputting lines of text. 6

clock cycle

A single increment of a processor's clock, during which the smallest unit of activity is carried out. 3, 6, 8, 81

cloud computing

The on-demand availability of computer system resources without direct active management by the user. 79

compilation

The process of translating computer programs from one programming language into another, typically from a high into a low-level language. 1, 8, 41, 53

computer

A machine that can be programmed to automatically carry out sequences of arithmetic or logical operations. 2, 3, 5

computer science

The study of computation, information, and automation. 1

CPU

The main processor in a computer, in charge of decoding and executing instructions. 40

D

data memory

A specialized type of memory used to store exclusively program data. 20, 41–43, 50, 51, *see* memory

debugging

The process of finding the cause of, and solving, undesired behavior in computer programs. 2, 6

desktop device

A personal computer designed for regular use. 2

E

embedded device

An independent device responsible for executing a specific task within a larger system. 2

F

FOSS

A type of software that is free to use and has its source code public. 2

G

GUI

A form of user interface that allows users to interact with electronic devices through graphical icons and visual indicators. 6

H

high-level language

A programming language with higher level abstractions, closer to the human language. 1, 2, *see* programming language

HTTP

An protocol designed to transfer information between networked devices. 39

I

I/O

The communication between a computer and the outside world. 7

instruction

An order given to a processor that makes the computer take some action. 8, 37–40

instruction dependency

A situation in which an instruction refers to the data of a preceding statement. 6, *see* instruction

interpreter

A computer program that directly executes instructions written in a programming or scripting language. 38, 39, 48, 49

IoT

A network of interrelated devices that connect and exchange data with other IoT devices and the cloud. 79

ISA

The set of instructions that a computer processor can understand and execute, as well as an abstract model of the architecture of a processor. 5

L

logical operation

A mathematical operation applied to logical values. 20

low-level language

A programming language with low level abstractions, closer to machine instructions see. 1

M

memory

A digital storage device used to store information. 3, 6–8, 20, 60, 61, 71, 74, 81

O

obfuscate

The act of creating source or machine code that is difficult for humans or computers to understand. 7

P

pipeline

A technique for implementing simultaneous execution of a sequence of instructions within a single processor. 6

pipeline stalls

A delay in execution of an instruction in order to resolve a hazard. 6, *see* pipeline

port

A connection endpoint to direct data to a specific service or application. 2

processor

An electrical component that executes instructions and performs operations on external data sources. 1–3

program

A sequence or set of instructions for a computer to execute. *see* programming

program counter

A processor register that points to the next instruction to be executed. 43, 46, 49

programmer

A person who programs. 1, *see* programming

programming

The process of expressing abstract ideas and models within a language. 1, 3

programming language

A system of notation for expressing computer programs. 2, 3, 9, 39, 53, 81, *see* program

programming paradigm

A method to solve a problem using tools and techniques that are available to us following some abstract approach. 39

pseudo-instruction

In an ISA, an instruction that is internally divided into several instructions by the processor. 8, *see* instruction

R

RAM

A form of computer memory that can be read and changed in any order, but requires power to retain the data. 54

register

A small data storage inside a computer's processor, used for storing data to operate. 3, 7, 8, 20, 40, 43, 44, 46–48, 51, 60, 61, 71, 74, 81

REPL environment

A simple interactive computer programming environment that reads a user input, executes it, and returns the result to the user. 9, 16, 26, 36, 61, 67, 72

S

software engineering

An engineering approach to software development, which involves the definition, implementation, testing, management and maintenance of software systems. 1, 57, 83

stack

A portion of memory in which the information or item stored last is retrieved first, typically used to store the local variables of a subroutine when it is called. 8

STDIN

The default input device that the program uses to read data. 46

STDOUT

The default file descriptor where a process can write output. 46

subroutine

A callable unit of software logic that has a well-defined interface and behavior and can be invoked multiple times. 8

system call

The programmatic way in which a computer program requests a service from the operating system on which it is executed. 46, 51

T

text memory

A specialized type of memory used to store exclusively program instructions. 20, 41, 42, 50, 51, *see* memory

theorem prover

A computer program or tool used in the field of logic and mathematics to automatically verify the validity of mathematical statements, propositions, or theorems. 7

U

UI

The space where interactions between humans and machines occur. 27

W

web application

An application software that is accessed using a web browser. 2

web service

A service offered by an electronic device to another electronic device, communicating with each other via the Internet. 39, 83

word

A fixed-sized datum handled as a unit by the instruction set. 42, 43, 50

ACRONYMS

A

AI

Artificial Inteligence. 79, *Glossary:* AI

ALU

Arithmetic Logic Unit. 20, 43, 71, 74, *Glossary:* ALU

AST

Abstract Syntax Tree. 38, 48–50, 53, 54, *Glossary:* AST

C

CLI

Command Line Interface. 6, 8–10, 15, 24, 32, 34–36, 40, 44, 72, 74, 83, *Glossary:* CLI

CPU

Central Processing Unit. 40, 71, *Glossary:* CPU

F

FOSS

Free Open Source Software. 2, 7, 17, 27, 53, 62, 67, 76, *Glossary:* FOSS

G

GPLv3

GNU General Public License, version 3. 78

GUI

Graphical User Interface. 6, 8, 83, *Glossary:* GUI

H

HTTP

HyperText Transfer Protocol. 39, *Glossary:* HTTP

I

I/O

Input/Output. 7–10, 20, 43, 59, *Glossary:* I/O

IoT

Internet of Things. 79, *Glossary:* IoT

ISA

Instruction Set Architecture. , 5, 7, 9, 11, 14, 18, 21–23, 28, 32, 34–38, 41, 43, 44, 46, 48, 51, 55, 59–61, 63, 65–67, 71, 78, 79, 81–83, *Glossary:* ISA

J

JSON

JavaScript Object Notation. , 39, 51, 53, 78

R

RAM

Random Access Memory. 54, *Glossary:* RAM

RISC

Reduced Instruction Set Computing. *Glossary:* RISC

S

STDIN

Standard Input. 46, 47, *Glossary:* STDIN

STDOUT

Standard Output. 46, *Glossary:* STDOUT

U

UI

User Interface. 27, 40, 72, *Glossary:* UI

Y

YAML

YAML Ain't Markup Language. 39

APPENDIX A

USER MANUAL

This annex presents a user manual for the developed simulator. The simulator allows for the definition of different computer architectures, with assembly-like instruction sets, and executing assembly-like programs using those instruction sets.

Instalation and execution

1. Install CMake[1] 3.22, Ninja-build[2] 1.11.1 and GCC[3] 11.4, or compatible versions.
2. Download and decompress the source code from the repository[4].
3. Move to the `build` directory and build and compile the source code[5]:

```
cd build/
cmake -G Ninja ..
cmake --build .
```

4. Execute the simulator (which can be found in `build/src/`) with an input ISA file. E.g.:

```
./src/tfg --isa=../architectures/RISC-V.json
```

[1] https://cmake.org/
[2] https://ninja-build.org/
[3] https://gcc.gnu.org/
[4] https://github.com/ldcas-uc3m/TFG
[5] You will need an Internet connection for CMake to download the dependencies.

Usage

When launching the simulator, you must specify the architecture definition file, through the `-isa` argument, which starts the simulator with that architecture. In order to change architecture, you must reload the simulator.

Once inside the simulator, the CLI offers several options, specified by entering the action letter:

- **`p` – load program:** Prompts the user to specify a program file, and loads the program.
- **`s` – stop program:** Stops the execution of the current program.
- **`n` – execute next instruction:** Executes the next instruction
- **`e` – enter REPL environment:** Enters the REPL environment, that allows the user to evaluate LUISP-DA expressions.
- **`f` – full run:** Executes the current program until it exits.
- **`r` – show registers:** Displays the registers and its contents (including the program counter).
- **`t` – show text memory:** Displays text memory.
- **`d` – show data:** Displays data memory.
- **`c` – change format:** Changes the display format, between the options it prompts the user: signed decimal, unsigned decimal, and hexadecimal.
- **`x` – exit:** Exits the simulator.
- **`h` – help:** Shows the different actions to perform.

LUISP-DA

LUISP-DA stands for *assembLy analogoUs lISP DiAlect*. It's the language used to define the instructions of the architecture.

It's a Lisp-like language, where instructions are defined as a list ($\langle LIST \rangle$) of symbols and/or other lists.

The immediate symbols implemented are:

- $\langle NUM \rangle$: Integer numbers.
- $\langle BOOL \rangle$: Booleans, `true` or `false`.

- $\langle NIL \rangle$: Undefined token.
- $\langle REG \rangle$: Registers defined in the architecture file.
- $\langle OP \rangle$: Defined operators.

A list is always in between parenthesis:

```
(+ 1 2)
```

Its first element is called the operand symbol, which will be applied to the rest of the elements (immediate symbols or lists) of the list. In the previous example, the operator + is applied to `1` and `2`, giving as a result `3`.

A list can be an element of another list. Inner lists are computed first, and they return the evaluation of the list. In the example:

```
(+ 1 (+ 2 2))
```

The operator + is applied to `2` and `2`, giving as a result `4`, and transforming the expression into:

```
(+ 1 4)
```

A list of one element is evaluated as the element. A list with no elements is evaluated as a $\langle NIL \rangle$.

An operand symbol can be:

- An operator $\langle OP \rangle$, which takes the child elements as arguments. The implemented operators (with its operands) are:
  - `+ A B`: Adds two $\langle NUM \rangle$ `A` and `A`.
  - `- A B`: Subtracts two $\langle NUM \rangle$ `A` and `A`.
    - `- A`: Negates the $\langle NUM \rangle$ `A`.
  - `* A B`: Multiplies two $\langle NUM \rangle$ `A` and `A`.
  - `/ A B`: Divides two $\langle NUM \rangle$ `A` and `A`.
  - `% A B`: Computes the modulo two $\langle NUM \rangle$ `A` and `A`.
  - `< A B`: For two $\langle NUM \rangle$ `A` and `A`, checks if `A` is less than `A`, returning a $\langle BOOL \rangle$.
  - `> A B`: For two $\langle NUM \rangle$ `A` and `A`, checks if `A` is bigger than `A`, returning a $\langle BOOL \rangle$.
  - `<= A B`: For two $\langle NUM \rangle$ `A` and `A`, checks if `A` is less or equal than `A`, returning a $\langle BOOL \rangle$.

  - `>= A B`: For two ⟨*NUM*⟩ `A` and `A`, checks if `A` is bigger or equal than `A`, returning a ⟨*BOOL*⟩.
  - `== A B`: For two ⟨*NUM*⟩ `A` and `A`, checks if `A` is equal to `A`, returning a ⟨*BOOL*⟩.
  - `!= A B`: For two ⟨*NUM*⟩ `A` and `A`, checks if `A` is not equal to `A`, returning a ⟨*BOOL*⟩.
  - `! A`: Negates the ⟨*BOOL*⟩ `A`.
  - `reg RA`: Returns the value stored in register (⟨*REG*⟩) `RA`.
  - `reg! RA B`: Stores the ⟨*NUM*⟩ `A` in register (⟨*REG*⟩) `RA`, and returns `A`.
  - `mem A`: Returns the value stored in memory address (⟨*NUM*⟩) `A`.
  - `mem! A B`: Stores the ⟨*NUM*⟩ `A` in memory address (⟨*NUM*⟩) `A`, and returns `A`.
  - `pc`: Returns the value stored in the program counter.
  - `pc! A`: Stores the ⟨*NUM*⟩ `A` in the program counter, and returns `A`.

- A block ⟨*BLK*⟩. It sequentially executes multiple instructions (⟨*LIST*⟩), and returns the result of the last one (PC is not updated for those instructions). In the example:

```
(do (+ 1 2) (+ -1 1))
```

  That would return the result of `(+ -1 1)`, `0`.

- A conditional ⟨*CND*⟩. It checks the condition (first element) (which must evaluate to a ⟨*BOOL*⟩), if it's other than ⟨*NIL*⟩ or `false`, it evaluates and returns the second element. Else, it evaluates and returns the third element (if it doesn't exist, it returns ⟨*NIL*⟩). In the example:

```
(if (> 2 1) (+ 1 1) (+ 2 2))
```

  That would return the result of `(+ 1 1)`, `2`.

- A system call ⟨*CALL*⟩. It checks the *opcode* (first element) (which must evaluate to a ⟨*NUM*⟩), and executes the specified system call with its arguments.
  - *print_int* – `call N A`: Prints the value of the ⟨*NUM*⟩ 'A'.
  - *print_char* – `call N A`: Prints the value of the ⟨*NUM*⟩ 'A' as an ascii character.
  - *read_int* – `call N RA`: Reads the value of a ⟨*NUM*⟩ and saves it to the register `RA`.
  - *read_char* – `call N RA`: Reads the value of an ascii character and saves it as a ⟨*NUM*⟩ to the register `RA`.

- *exit* – `call N`: Stops the execution of the program.

The definition of the *opcodes* is defined in the architecture definition file.

After each instruction, the PC is updated.

Architecture Definition File

The structure of the JSON files is the following:

- `name`: Name of the architecture.
- `memory`: Memory configuration.
  - `text`: Text memory configuration.
    - `start_addr`: Memory start address (`string` - HEX).
  - `data`: Data memory configuration.
    - `start_addr`: Memory start address (`string` - HEX).
    - `end_addr`: Memory end address (`string` - HEX).
- `registers`: Register names (`array[string]`).
- `syscalls`: System call map (`object[string, string]`).
  - <id>: `"print_int"|"print_char"|"read_int"|"read_char"|"exit"`
- `instruction_set`: Set of architecture instructions.
  Each instruction is a JSON object with the following format:
  - <instruction_name>
    - `args`: Function arguments. They must all start by `$`. (`array[string]`)
    - `def`: LUISP-DA definition (`string`).
- `data_types`: Map of memory data types and their mappings. Supported data types are:
  - `str`: zero-terminated strings
  - `word`: 32-bit words
- `comment_char`: Character that starts a comment line (`string`).

You can find examples of instruction sets in the `architectures/` folder.

Assembly Program File

This text file will contain the instructions and data for the program to be executed. Empty lines and extra whitespace is ignored. The simulator only allows line comments, and they must start by the defined `comment_char`. You can find examples of programs in the `examples/` folder.

The program file is divided into two segments: the data and the text segment.

Data segment

Here you can specify how to preload the data memory of the computer. The segment start is marked by the use of the `.data` keyword.

They are divided into three parts, separated by spaces: label (`<name>:`), type (those defined in `data_types`), and value, which can be a string (`"<data>"`), a hexadecimal value (`0x<number>`), or an integer value.

Text segment

Here you can define the different instructions and labels. The segment start is marked by the use of the `.text` keyword.

Instructions must occupy only one line. Labels, with a '`<name>:`' format, must be defined in a single line, and are mark to the next instruction.

In your instruction you can use any of the defined label, including those from the data segment. The only required label is `main`, which marks the entry point of the program.