UNIVERSITY OF TRIESTE

Department of Engineering and Architecture

Bachelor's degree in Computer Engineering

Restoration and development of Arm's Java-based LEGv8 ISA simulator

August 24, 2024

Graduating student
Simone Deiana

Supervisor
Prof. Alberto Carini

Academic Year 2023/2024

"I have only made this letter longer
because I have not had the time to make it shorter."
— Blaise Pascal – Letter 16, 1657 —

Summary

LEGv8 is an ARMv8-inspired ISA developed by D.A. Patterson and J.L. Hennessy for their Computer Architectures textbook [1]. Being an academic ISA for undergraduate teaching, emulators or simulators are needed to provide real life examples of LEGv8 code being executed. Amongst them, a Java-based simulator [2] published by Arm [3] has been chosen as the subject of this thesis. This simulator distinguishes itself for its comprehensible and structured codebase and for being deployed as a cross-platform web application. Its major drawbacks are a lack of documentation on how to build and develop the simulator, ageing libraries and tools, an incomplete implementation of the ISA, and a series of critical bugs impeding the normal functioning of the simulation.

The work presented in this thesis can be divided into five parts:

1. Understanding and documenting the current state of the codebase and the ways to build and deploy it as was intended by the developers.

2. Modernizing the libraries used in the project and integrating it with a modern build automation system [4] to make development and deployment of the simulator as simple as possible and decouple it from the tools it was originally developed with.

3. Fixing the critical bugs that impeded the normal functioning of the simulator, namely the inability to perform correct comparisons and subroutine calls.

4. Implementing the entire single cycle LEGv8 ISA inside the simulator by adding the remaining integer arithmetical instructions and integrating the necessary logic to perform floating-point operations both in single and double precision.

5. Updating the web interface of the simulator to provide a real time visualization of the stack memory, the newly introduced integer and floating point instructions, and the floating point registers. In addition, a layout change to better present the simulator information to the user.

Sommario

LEGv8 è un'ISA ispirata ad ARMv8 e sviluppata da D.A. Patterson e J.L. Hennessy per il loro libro di testo di Architetture dei Sistemi Digitali [1]. Essendo un'ISA accademica per l'insegnamento universitario, sono necessari emulatori o simulatori per fornire esempi tangibili di esecuzione di codice scritto per essa. Tra questi, è stato scelto come oggetto di questa tesi un simulatore basato su Java [2] pubblicato da Arm [3]. Esso si distingue per il suo codice ben strutturato e la sua distribuzione come applicativo web. I suoi principali svantaggi sono la mancanza di documentazione su come impostare e compilare il simulatore, librerie e strumenti obsoleti, un'implementazione incompleta dell'ISA, e una serie di bug critici che impediscono il normale funzionamento della simulazione.

Il lavoro presentato in questa tesi può essere diviso in cinque parti:

1. Comprendere e documentare lo stato attuale della base di codice e i modi per compilarla e distribuirla come previsto dagli sviluppatori.

2. Modernizzare le librerie utilizzate nel progetto e integrarlo con un moderno sistema di build automation [4] per rendere lo sviluppo e la distribuzione del simulatore il più semplice possibile e renderlo indipendente dagli strumenti con cui è stato originariamente sviluppato.

3. Correggere i bug critici che impedivano il normale funzionamento del simulatore, ovvero l'incapacità di eseguire confronti tra numeri e chiamate a subroutine.

4. Implementare l'intera ISA LEGv8 a ciclo singolo, aggiungendo le mancanti istruzioni aritmetiche intere e integrando la logica necessaria per le operazioni in virgola mobile, sia a singola che doppia precisione.

5. Aggiornare l'interfaccia web del simulatore per fornire una visualizzazione in tempo reale dello stack, delle nuove istruzioni intere e in virgola mobile e dei registri in virgola mobile. Inoltre, riorganizzare il layout per presentare meglio le informazioni.

Contents

Introduction

> "The purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise."
>
> *Edsger W. Dijkstra* – The Humble Programmer (1972)

The work presented in this thesis concerns the restoration and development of a Java-based LEGv8 simulator.

Some context

LEGv8 is an instruction set architecture (which from now on will be referred to as ISA) designed by D.A. Patterson and J.L. Hennessy to serve as a pedagogical tool in their textbook *Computer Organization and Design ARM Edition* [1]. An ISA is a set of specifications that provides a logical description of a processor-based system. It offers a layer of abstraction that both the hardware and the software need to conform to in order to ensure proper interoperability, meaning that code written for a certain ISA is sure to work on any processor designed following the same specification and vice versa. As the title of the book suggests, the text uses the ARM ISA (more specifically, the v8 version) as a real life showcase of the modern approaches to the design of computer systems. However, in some of the chapters, a simpler architecture is needed to better present some of the material, and for this reason the authors identified a simpler and more coherent subset of the ARMv8 ISA which they named LEGv8. Being a fictional ISA created to serve the pedagogical purposes of a single textbook, there are currently no commercially available hardware implementations of the architecture defined by LEGv8, meaning that any code written for it can only be executed by the means of software simulation.

Being a relatively popular textbook in undergraduate Computer Architeture courses, an ecosystem of simulators has grown over the years mostly from the enterprise of students and professors, either because of personal interests or because of course requirements. These simulators can be divided in two categories: hardware simulators and software simulators. The former concern themselves with simulating the inner workings of a hypothetical physical implementation of the LEGv8 ISA, while the latter are more interested in simulating the logical execution of the code from the point of view of an external user. Regardless of the type, none of them achieve a complete simulation of the entire LEGv8 ISA for one reason or another, especially when dealing with floating point operations. This might not be a issue when wanting to execute, test or showcase code limited to only a few of the available instructions, as would be the case in a university course with well defined requirements, but problems arise when a student or educator might want to do so using the full spectrum of operations specified by LEGv8. For this reason, a simulator capable of achieving a complete and correct coverage of the entire ISA would be a welcomed addition in the current LEGv8 software landscape.

Among the software simulators available online, one is being developed and published by Arm itself [2] and is the subject of this thesis' work.
Arm's LEGv8 simulator is almost entirely written in Java and uses the GWT framework [5] to distribute the software in the form of a web application. Its codebase is well structured and its use of a platform-agnostic and high level language allows for easy development and extension. Furthermore, the use of GWT gives it the ability to be run on any device running a modern-ish version of any HTML5-compatible web browser. Due to GWT's integration with the Eclipse IDE for Java, this development environment was chosen for the project. These design choices, however, present some criticalities which will be discussed later.
Other issues arise regarding the execution of the simulator, which presents critical bugs that impede any kind of conditional statements, loops, and that don't allow for the execution of user defined functions. It is clear then that a simulator only capable of executing branchless code without support for subroutines is not adequate for the simulation of the vast majority of computer code.
As mentioned in the previous paragraph, none of the publicly available simulators cover the entire LEGv8 ISA, and Arm's one is no exception. This, combined with the presence of the aforementioned simulation-breaking bugs, might discourage its usage and even question its utility. Nonetheless, because of its popularity and the nature of its publisher, it has been deemed the highest impact project when deciding which one to work on for this thesis.

Presentation of the thesis' work

The work presented in this thesis has tackled different parts both internal and external to the simulator and can be divided into 5 logical phases, not necessarily in chronological order.

0. Study of the simulator. The initial part of the work has consisted in analyzing the structure and inner workings of the simulator in order to identify the developers' intentions with its design and the source of its breaking bugs. An important part of this work has been understanding the codebase's undocumented build process and dependencies, documenting them through a comprehensive tutorial, and making it publicly available to allow for future development and collaboration.

An overview of the LEGv8 ISA, a survey of the available simulators, and an overview of Arm's codebase together with its set-up shortcomings and the work done fixing them will be the subject of Chapter 1.

1. Modernization of the development environment. A few of the design choices made by the original developers have had some unfortunate consequences on the current state of the project. First among them has been the lack of documentation regarding the build process for the simulator, which has made it impossible to effectively develop fixes and enhancements to the codebase and be able to deploy them and share them. Secondly, the choice of GWT as a framework upon which to base the project has caused the developers to depend on an outdated version of the Eclipse IDE for development. Furthermore, there have been some changes to newer versions of GWT that updated how the library is deployed and break the original Eclipse's workflow.
The work done in this phase has been to update GWT to its latest iteration and to adapt the codebase to accomodate the new build environment. This was achieved by integrating Maven [4] into the project, thus bringing it to a modern standard of build automation and dependency management. Thanks to this work, the simulator's development environment has been decoupled from Eclipse and can now be set-up in a few simple steps both from the terminal or via any Maven-compatible Java IDE. This allows for an almost automatic set-up process and opens the doors to future development and collaboration thanks to the ability to utilize different and more featureful IDEs, new language features, and take advantage of the automations provided by Maven such as code testing and plugins.

A more extensive overview of the project's structure, its libraries, configuration, and the steps taken to modernize it will be included in Chapter

2.

2. Critical bug fixes. As mentioned before, the simulator presented some breaking bugs that impeded the expected functioning of the code execution. The first bug consisted in an error in the logic of the simulator that incorrectly evaluated comparisons between integers. Since integer comparisons are at the foundation of conditional instructions and loops, this caused branches to not be taken and loops to be exited prematurely.
The second bug was instead the result of a problem in setting the correct return position when calling a user defined function. This meant that when a function was executed, once the final instruction was reached, instead of returning to the main program where the function was called, the program entered the function again in an infinite loop.

The nature of these bugs and the steps taken to fix them will be discussed in Chapter 3.

3. Completing the integer arithmetic and integrating floating point support. From the survey of available simulators, Arm's didn't distinguish itself for its coverage of the LEGv8 instruction set. This is both in the case of the integer instructions (having implemented only a subset) and the floating point ones (none of which have been implemented).
Regarding the addition of the missing integer instructions, the structure of the codebase allowed for an easy implementation, although some limitation of the Java language necessitated alternative solutions. None of the instructions fundamentally changed the inner workings of the simulator, thus all that was neeeded was to imitate the already implemented logic.
The floating point logic, on the other hand, introduced new elements to the design of the simulator since it required separate places in which to store the values being operated with, and didn't allow the intermixing of integer and floating point values and instructions. In order to implement these features, some previously untouched parts of the simulator have been extended with the necessary logic.

4. Updating the simulator's UI. All the changes and additions mentioned in the previous paragraph needed to be presented to the user through an update of the original UI.
An initial addition, independent from all the changes already mentioned, was the introduction of the visualization of the main memory of the simulated computer. This allows the user to see the complete state of of the memory in real time, both for didactic purposes and to check for errors in the code.
Secondly, all the new floating point logic required the addition of new elements in the UI to visualize the current floating point values in the working

memory divided between single and double precision and with the correct dotted decimal notation.
Lastly, a general reorganization of the UI elements was performed to make the simulator fit better on an average computer monitor or projector and present more information without navigating the page.

A complete overview and explanation of all the introduced changes and additions, both to the internal logic of the simulator and its UI, will be provided in Chapter 4.

The work presented in this thesis only tackled a subset of the problems and shortcomings of the simulator, as they are numerous and would require a great amount of additional work as many of them are intrinsic to the design choices of the original developers. The concluding chapter will include a final overview of the work done and a discussion the possible further developments of the simulator and how they might be achieved.

Chapter 1

The what, and the why

> "Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better."
>
> *Edsger W. Dijkstra* – EWD896

In this chapter we will provide a brief introduction to the LEGv8 ISA, together with a survey of its publicly available simulators. After that, we will focus on the simulator chosen for this thesis' work, namely Arm's, by giving an overview of its codebase, design choices, advantages, and shortcomings. The chapter will end with a summary of the motivations for choosing to work on this specific simulator.

1.1 What is an ISA?

A computer is a device which is capable of processing, storing, and displaying information [6]. It is clear from this definition that, when deciding how to design and build a computer, one must at least take into consideration the way data is stored and organized (i.e., the memory), and the mechanisms through which the computer is able to manipulate said data (i.e., the processor). As per the definition, the concept of a computer does not impose a certain technological choice to its physical realization. Nonetheless, the vast majority of computers nowadays are built through the assembly of digital components and thus natively speak the language of the binary number system. As such, just like a user operating a mechanical device needs to interact with the physical parts of the system, dealing with a computer at this physical level would require the programmer to manually insert ones and zeros into the right places for it to perform its computations. Such an

operation would require an intimate knowledge of the physical realization of the computer, and even minimal changes to its internal components might jeopardize the correctness of any program written for an earlier version.

Early on in the history of computers it was understood that an additional layer of abstraction was needed in order to separate the hardware from the software and give more freedom both to the designers of the computers, and the programmers operating on them. This layer of abstraction is called an Instruction Set Architecture, which from now on will be called ISA for short. An ISA provides a logical specification of how a computer manages its memory and what the instructions that is capable of performing are. This forms the layer through which all software must interface with in order to interact with the hardware.

1.2 What is LEGv8?

The ISA focus of this thesis is the LEGv8 ISA, an ARM-inspired architecture created by David A. Patterson and John L. Hennessy and designed to serve as a teaching tool in their book *Computer Organization and Design (ARM Edition)* [1]. Quoting the authors:

"The choice of instruction set architecture is clearly critical to the pedagogy of a computer architecture textbook. We didn't want an instruction set that required describing unnecessary baroque features for someone's first instruction set, no matter how popular it is. Given the growing popularity and the simple elegance of the MIPS instruction set, we switched to it for the first edition of this book and to later editions of the other book. MIPS has served us and our readers well. The incredible popularity of the ARM instruction set led some instructors to ask for a version of the book based on ARM. Although ARMv8 is much, much larger than MIPS we found a subset of ARMv8 instructions that is similar in size and nature to the MIPS core used in prior editions, which we call LEGv8 to avoid confusion. Hence, we wrote this ARMv8 edition." pp. xvi-xvii

Although the authors refer to LEGv8 as a "subset" of ARMv8, it is important to note that, in fact, some additions and changes have been made to the instruction set to simplify it further and make it more coherent for pedagogical purposes. This results in LEGv8 code not being executable on an ARM-compatible computer but instead requiring some manual translation. This, together with the purely academic nature of the ISA, means that no LEGv8-compatible computers are commercially available and all LEGv8 code must be executed through a software simulation or somehow converted to the ARMv8 equivalent.

1.3 Overview of the LEGv8 ISA

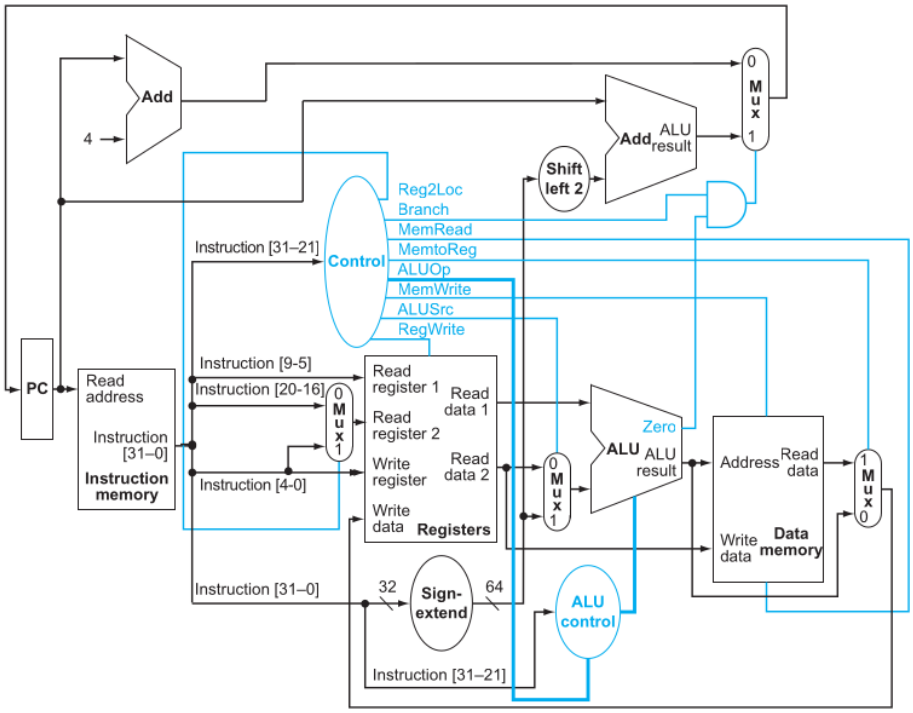


Figure 1.1: The logical scheme of the LEGv8 architecture. From *Computer Organization and Design (ARM Edition)* [1] p. 277

In the book [1], the LEGv8 ISA is specified with two different levels of abstraction. Throughout the whole text, a high level description of the ISA is provided and utilized, and it will be the one summarized in the rest of this section. This specification contains the entire LEGv8 instruction set, but doesn't concern itself with the lower level design needed to create a working implementation. In Chapter 4 of the book, the authors aim to provide a closer description of a model LEGv8 processor following the scheme depicted in Figure 1.1. Although the chapter goes into much more detail than the figure shown, providing a description of the entire ISA at this level would require an entire book of its own. For this reason, the authors limited themselves to a subset of LEGv8 containing only a handful of integer instructions. The distinction between these two specifications will become relevant in the next sections and chapters.

1.3.1 Architecture type

As Figure 1.1 makes apparent, the LEGv8 ISA follows the Harvard architecture model [7]. This means that the memory is logically divided in two: the *instruction memory* to store the code of the program, and the *data memory* to store the data necessary to the program during its execution. These two memory sections are addressed separately.

1.3.2 Registers

Registers are the memory components that the processor uses to temporarily store the data it needs to perform its operations. They are few, fast, and expensive, and are independent from the main memory.
LEGv8 defines 32 64-bit `X` registers for storing integer values and memory addresses and 32 64-bit `D` registers for storing double precision floating point values. This makes LEGv8 a 64-bit ISA. There are also 32 32-bit `S` registers dedicated to single precision floating point values, albeit being purely logical and simply occupying the lower 32 bits of the `D` registers. Unlike ARMv8, the presence of 32-bit `W` integer registers is not contemplated.
LEGv8 also defines a convention for using the registers, which, while not being entirely enforced by the hardware, is to be followed when writing any non-trivial program. It also specifies some alternative names with which to refer to certain registers for ease of readability. Figure 1.2 shows the convention for integer registers, but there are analogous conventions for the floating point ones too.

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
X0 – X7	0-7	Arguments / Results	No
X8	8	Indirect result location register	No
X9 – X15	9-15	Temporaries	No
X16 (IP0)	16	May be used by linker as a scratch register; other times used as temporary register	No
X17 (IP1)	17	May be used by linker as a scratch register; other times used as temporary register	No
X18	18	Platform register for platform independent code; otherwise a temporary register	No
X19-X27	19-27	Saved	Yes
X28 (SP)	28	Stack Pointer	Yes
X29 (FP)	29	Frame Pointer	Yes
X30 (LR)	30	Return Address	Yes
XZR	31	The Constant Value 0	N.A.

Figure 1.2: Integer registers usage convention. From *Computer Organization and Design (ARM Edition)* [1].

In addition to the normal registers directly accessible by the programmer, more exist to store the *program counter (PC)* (i.e. the address of the current instruction to be executed) and various flags that keep track of overflows or carry bits in arithmetic operations and comparisons.

Condition flags The registers responsible for holding the condition flags are of vital importance when needing to perform comparisons both in the case of signed and unsigned integer values. Here are the conditions that the four flag registers follow, as explained in Patterson's text [1] on page 97:

- negative (N) – the result that set the condition code had a 1 in the most significant bit;
- zero (Z) – the result that set the condition code was 0;
- overflow (V) – the result that set the condition code overflowed; and
- carry (C) – the result that set the condition code had a carry out of the most significant bit or a borrow into the most significant bit.

1.3.3 Memory

While in practice LEGv8 follows the Harvard model by addressing each section separately, its two memories can be logically stitched together as

a single array of bytes, with each memory address referring to one of said bytes. This unified logical representation of the memory is divided into 5 parts as depicted by Figure 1.3: a *reserved* segment for internal usage, a *text* segment containing the program code (and from which the program counter starts), a *static data* segment containing the constants defined at compile time, and a *dynamic data* and *stack* segments sharing the same location of memory. The *stack* memory starts at the address defined by the *stack pointer (SP)* and grows downwards. It is the memory that gets allocated when calling functions, and it contains all the data therein utilized for the sole duration of their execution. Each function allocates its portion of the stack, called *stack frame*, and it should be the only one it accesses. The *dynamic data*, or *heap*, begins immediately after the *static data* segment and grows upwards. It contains the dynamically allocated data of the program, thus its size depends on a particular execution of the program and cannot be predicted beforehand. This segment contains the shared data of the program, which can be accessed from everywhere.

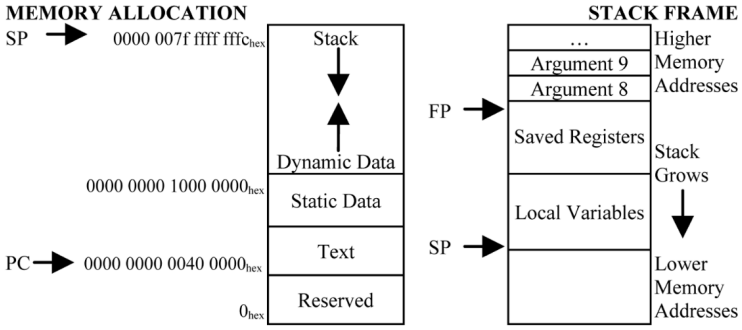


Figure 1.3: Logical division of the memory. From *Computer Organization and Design (ARM Edition)* [1].

1.3.4 ALU

The ALU is the component responsible for all the arithmetical operations inside of the processor. The LEGv8 ALU is capable of performing 64-bit integer operations and both single and double precision floating point operations. Hybrid operations between different types of data or registers are not possible. As shown in Figure 1.1, the type of operation that the ALU is expected to perform and some additional configuration is provided by the control unit through an ALUop code.

1.3.5 Pipeline

Among the eight great ideas in computer architecture compiled by the authors in the book, a particular kind of parallelism, called *pipelining*, is

mentioned. *Pipelining* refers to the ability of a computer architecture to organize its instructions and its structure in a way that makes it possible to overlap their execution without waiting for the completion of each to begin the next. The execution of only one instruction at a time is called *single cycle* execution. LEGv8 achieves pipelining by splitting the execution of an instruction into 5 stages: *fetch*, *decode*, *execute*, *data access*, and *write back*, with the order shown in Figure 1.4.
As the names suggest, the *fetch* stage is responsible for acquiring instructions from the instruction memory, the *decode* stage decodes the instructions, reads the registers involved in the operation, and configures the control unit accordingly, the *execute* stage performs the actual computation through the ALU, the *data access* stage is responsible for accessing the data memory, and the *write back* stage writes the result of the operation into the registers.

Figure 1.4: The 5 pipeline stages

As an example, suppose that two instructions do not make use of shared resources. If one is being decoded why can't the next one get fetched in the meantime? And if one is accessing the memory, why can't the other perform its computation via the ALU at the same time? This is, of course, a simplistic example, and in many situations subsequent instructions get in the way of each other by making use of the same registers or memory locations, and as such create an implicit order of execution that needs to be followed if we want to obtain correct results. Not all instructions make use of all the pipeline stages and this is taken into consideration when optimizing the execution flow. These situations in which instructions conflict with one another are called *hazards*, and need to be dealt with by either strategically rearranging the instructions in the code to avoid such conflicts, or by making additions to the architecture in order to create "shortcuts" that bypass the pipeline and provide the correct data to the other stages. If all else fails, which is the case for some combinations of instructions, the processor can simply stall part of the pipeline to let an instruction go through the stages without creating dependency problems. This is, of course, the least desirable option.

1.3.6 Instructions

As previously mentioned, LEGv8 can be considered a subset of ARMv8, but with a few caveats. Many higher level instructions performing aggregate or complex operations have been omitted altogether in order to keep the ISA

as minimal as possible, and of the ones that have been kept, many have been revisited to make them clearer in their scope. For example, in ARMv8 the `ADD` instruction can be used with both 32 and 64 bit integer registers, and with both register-based and immediate-based (i.e., defined directly in the program code) values. This of course allows the ARMv8 programmer to remember a single mnemonic (i.e., the name of an instruction) and use it in all sorts of operations, but it obscures some important underlying design differences that might be valuable to computer architecture students. In LEGv8 instead, it has been decided to split the `ADD` instruction into `ADD` and `ADDI`, for register and immediate values usage respectively. Similarly, in ARMv8 the `FADD` instruction is capable of performing additions both in the case of single and double precision registers, whereas in LEGv8 the instruction has been split into `FADDS` and `FADDD` for performing the operation only on single precision or double precision registers respectively. Just like ARMv8, LEGv8 too uses the IEEE-754 standard [8] for the representation of floating point values and the way its instructions operate on them.
Figure 1.5 provides a full list of all the instructions defined by LEGv8 together with their encoding.

Instruction		Opcode		Shamt	11-bit Opcode Range (1)	
Mnemonic	Format	Width (bits)	Binary	Binary	Start (Hex)	End (Hex)
B	B	6	000101		0A0	0BF
FMULS	R	11	00011110001	000010	0F1	
FDIVS	R	11	00011110001	000110	0F1	
FCMPS	R	11	00011110001	001000	0F1	
FADDS	R	11	00011110001	001010	0F1	
FSUBS	R	11	00011110001	001110	0F1	
FMULD	R	11	00011110011	000010	0F3	
FDIVD	R	11	00011110011	000110	0F3	
FCMPD	R	11	00011110011	001000	0F3	
FADDD	R	11	00011110011	001010	0F3	
FSUBD	R	11	00011110011	001110	0F3	
STURB	D	11	00111000000		1C0	
LDURB	D	11	00111000010		1C2	
B.cond	CB	8	01010100		2A0	2A7
STURH	D	11	01111000000		3C0	
LDURH	D	11	01111000010		3C2	
AND	R	11	10001010000		450	
ADD	R	11	10001011000		458	
ADDI	I	10	1001000100		488	489
ANDI	I	10	1001001000		490	491
BL	B	6	100101		4A0	4BF
SDIV	R	11	10011010110	000010	4D6	
UDIV	R	11	10011010110	000011	4D6	
MUL	R	11	10011011000	011111	4D8	
SMULH	R	11	10011011010		4DA	
UMULH	R	11	10011011110		4DE	
ORR	R	11	10101010000		550	

Instruction		Opcode		Shamt	11-bit Opcode Range (1)	
Mnemonic	Format	Width (bits)	Binary	Binary	Start (Hex)	End (Hex)
ADDS	R	11	10101011000		558	
ADDIS	I	10	1011000100		588	589
ORRI	I	10	1011001000		590	591
CBZ	CB	8	10110100		5A0	5A7
CBNZ	CB	8	10110101		5A8	5AF
STURW	D	11	10111000000		5C0	
LDURSW	D	11	10111000100		5C4	
STURS	R	11	10111100000		5E0	
LDURS	R	11	10111100010		5E2	
STXR	D	11	11001000000		640	
LDXR	D	11	11001000010		642	
EOR	R	11	11001010000		650	
SUB	R	11	11001011000		658	
SUBI	I	10	1101000100		688	689
EORI	I	10	1101001000		690	691
MOVZ	IM	9	110100101		694	697
LSR	R	11	11010011010		69A	
LSL	R	11	11010011011		69B	
BR	R	11	11010110000		6B0	
ANDS	R	11	11101010000		750	
SUBS	R	11	11101011000		758	
SUBIS	I	10	1111000100		788	789
ANDIS	I	10	1111001000		790	791
MOVK	IM	9	111100101		794	797
STUR	D	11	11111000000		7C0	
LDUR	D	11	11111000010		7C2	
STURD	R	11	11111100000		7E0	
LDURD	R	11	11111100010		7E2	

Figure 1.5: The complete LEGv8 instruction list. Adapted from *Computer Organization and Design (ARM Edition)* [1].

As Figure 1.6 shows, the instructions are encoded with the same length of 32 bits in order to fetch and decode them more efficiently. They are also grouped into 5 instruction formats to give a more homogeneous encoding to operations performing similar steps and increase their decoding speed. The `R`-type instructions perform operations solely on registers, the `I`-type instructions make use of immediate values, the `D`-type instructions access the

memory, the `B`-type and `CB` perform unconditional and conditional branching respectively, and the `IW`-type instructions to perform MOV instructions (i.e., instructions where data is simply moved around) with wider immediate values.

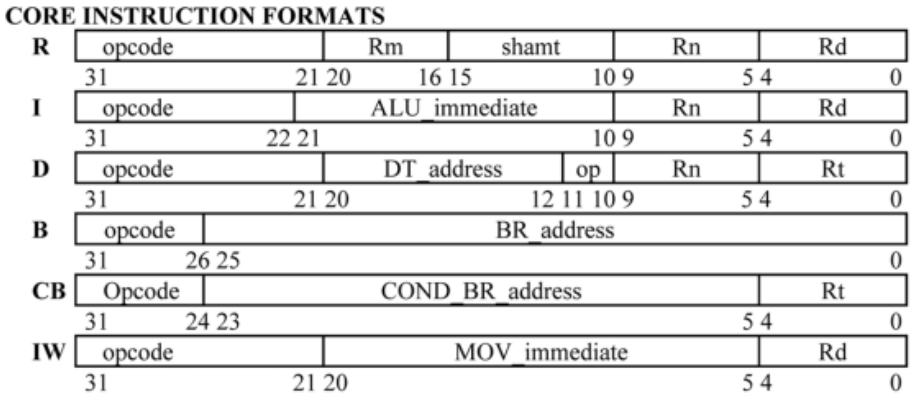


Figure 1.6: The 5 formats of LEGv8 instructions with their encoding pattern. From *Computer Organization and Design (ARM Edition)* [1].

1.3.7 Control unit

While everything we have talked about until now can be considered part of the *datapath* of the processor, meaning the series of components the data passes through in order to complete a computation, this datapath needs an additional component to coordinate its functioning. The control unit is the component responsible for coordinating the single cycle and the pipelined datapath, configuring the various components to perform the desired operations in the correct order using the correct parameters and avoiding hazards.

1.3.8 Other discrepancies between LEGv8 and ARMv8

There are ulterior design choices that differentiate LEGv8 from ARMv8 and make them even more incompatible with one another. We will report them for completeness, but are merely tangential to this thesis' work. Citing the authors of the book [1] at pages 174 and 175:

"In the full ARMv8 instruction set, register 31 is XZR in most instructions but the stack pointer (SP) in others. We think it is confusing, so register 31 is always XZR and SP is always register 28 in LEGv8. If you stick to using XZR and SP for register names when using the assembler and simulator, it should not be a problem, but this subtle distinction might show up in visualizations of the state of the processor, for example."

"The immediate fields for ANDI, ORRI, and EORI of the full ARMv8 instruction set are not simple 12-bit immediates that we assume in LEGv8. ARMv8 has an algorithm for encoding immediate values as repeating patterns. This means that some small constants (e.g., 1, 2, 3, 4, and 6) are valid, while others (e.g., 0, 5) are not. Thus, the assembler may insert more instructions than you might expect to create simple constants, or fewer if you pick a lucky large constant. The official definition is the bit pattern can be viewed "as a vector of identical elements of size e = 2, 4, 8, 16, 32, or 64 bits. Each element contains the same sub-pattern, that is a single run of 1 to (e − 1) nonzero bits from bit 0 followed by zero bits, then rotated by 0 to (e − 1) bits." Don't try to figure this out yourself; just leave it to the assembler!"

1.4 The LEGv8 simulators landscape

We have already mentioned that LEGv8 is a purely academic ISA designed to serve the purposes of a single undergraduate textbook. Although more complex academic architectures have been physically realized by research groups and even gained commercial traction [9], there are currently no publicly available implementations of LEGv8 to be used to run its native instructions. This means that, for any LEGv8 code to be successfully executed, a software simulator is currently needed. Being an ISA taught at a number of universities, various simulators have been created both by students and educators to aid in their teaching or as homework requirements. For this reason, the vast majority of simulators are limited in their scope and aim to implement only a subset of the LEGv8 ISA.

1.4.1 The survey

The current offering of publicly available LEGv8 simulators can be divided into two categories: simulators that aim to reproduce the lower level design presented in the textbook [1], in Chapter 4, and the simulators providing a high level simulation of the instruction set as described in the previous section.

Methodology GitHub [10] was chosen as the public repository in which to perform the survey. The keywords used to identify possible simulators were "LEGv8" and "simulator". Simulators with lacking documentation or which were described by the author(s) as partially done or not working were excluded. The simulator chosen for this thesis was also excluded to better highlight its features in the following sections.

Repository	Language	Integer Support	Pipelined	Registers view	Stack view	Floating Point Support
https://github.com/lcpckp/leg-cpu-sim	Java	Partial	No	Yes	Yes	No
https://github.com/chrwoods/legv8-emul	C/C++	Partial	Yes	Yes	Yes	No
https://github.com/mtalyat/LEGv8Day	C#	Partial	No	Yes	Yes	No
https://github.com/eaxworthy/LegV8Interpreter	Python	Partial	No	Yes	Yes	No
https://github.com/AdinAck/LEGv8-Simulator	Swift	Partial	No	Yes	Yes	No
https://github.com/anvitha305/legv8sim	Python	Partial	No	Yes	Yes	Double precision only
https://github.com/dangbandy/LegV8-Simulator	C++	Partial	No	Yes	Yes	No
https://github.com/schang412/LEGv8-PyEmu	Python	Partial	No	No	No	No
https://github.com/GeorgePerreault/LEGV8_Interpreter	Python	Partial	No	Yes	Yes	No

Table 1.1: The surveyed software simulators

Repository	Language	Integer Support	Pipelined	Floating Point Support
https://github.com/nxbyte/ARM-LEGv8	Verilog	Partial	Yes	No
https://github.com/phillbush/legv8	Verilog	Partial	Yes	No
https://github.com/ronitrex/ARMLEG	Verilog	Partial	Yes	No
https://github.com/mattco98/LEGv8-Processor	Verilog	Partial	Yes	Partial
https://github.com/amaurilopez90/LEGv8-CPU	Verilog	Partial	Yes	No
https://github.com/miguelangelo78/LEGv8-ISA	Verilog	Partial	Yes	No
https://github.com/brianworts/LEGv8_SingleCycle_Processor	Verilog	Partial	Yes	No
https://github.com/egflo/LEGv8	Verilog	Partial	Yes	No
https://github.com/ad153153/LegV8	Verilog	Partial	Yes	No

Table 1.2: The surveyed hardware simulators

1.4.2 Software simulators

The surveyed software simulators in Table 1.1 are in line with the previously made remarks. They are written in high level languages such as Java, C++, C#, Swift, and Python, and take the high level approach to LEGv8 as described near the beginning of the chapter. They do not implement the entirety of the integer arithmetic, although on average they include more instructions than hardware simulators. The languages they are written with allow most of them to provide a visualization of the registers and the stack to the user, in some cases through a GUI. Only one of them simulates the LEGv8 pipeline, and only one includes floating point arithmetic, although just the double precision one.

1.4.3 Hardware simulators

These simulators concern themselves with implementing the lower level description of LEGv8, as such all of them use a hardware description language. Of all of the surveyed ones, Verilog is the only one used. Because of this, they cannot provide any human-readable representation of the internal state of the simulation, thus visualization options weren't included in the fields of the table. Because they closely follow the design as presented in the book, they only implement a few integer instructions total, but on the other hand all of them are pipelined. Only one furthers its design by including some floating point operations.

1.4.4 Conclusions

It is clear from this brief survey that the LEGv8 simulators space lacks any desirable candidates for code execution and inspection, as the software simulators are incomplete and platform-dependant, and the hardware ones are extremely limited in their scope, and lack interactivity and comprehensive visual output capabilities. Furthermore, most of them are finished projects and are not currently being maintained or expanded.

1.5 Arm's LEGv8 simulator

Among the surveyed simulators, an important entry is missing. Indeed, when searching for a "LEGv8 simulator" both on GitHub and through various search engines, the first results all refer to a project [2] developed and released officially by Arm. From this metric we can infer that this is in fact the most prominent publicly available LEGv8 simulator, at least regarding authority and sheer exposure. In this section we will provide an overview of the simulator from the user's and the developer's perspectives, and in both cases detailing its strong points and shortcomings.

1.5.1 Obtaining the simulator

The project is currently hosted on a public GitHub repository [11] published by the account Arm Education. The source code and the resulting application are not clearly separated into different packages. Instead, both users and developers can download the entire project either by cloning the repository through `git` [12] or by downloading a prepackaged release from the Releases section of the web page. This version differs from the source code only by some minor updates to the project's license and `README` files. Nested inside of the downloaded folder, the source code is placed under `/src`, and the executable web application under `/war`.

1.5.2 An outside look

Here we will present the contents of the UI of the simulator together with a typical workflow from the user. At the end, a critical look of this experience will be provided.

The look and feel

The simulator presents itself as a web application. It is launched by opening the `LEGv8_Simulator.html` file with a web browser and the user is presented with the UI shown in figure 1.7.

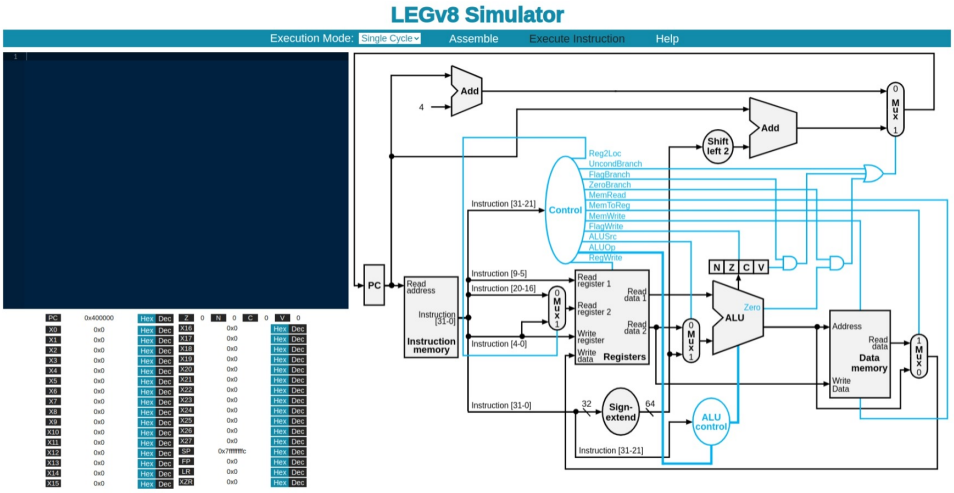


Figure 1.7: The simulator's home page

At the top of the page stands a menu bar consisting of four elements. The first three pertain to the simulator's code execution and their effect is limited to the current page, while the last one, bearing the label *Help*, redirects the user to a page containing both a tutorial for the application and a brief overview of LEGv8.

The *Execution Mode* selector allows the user to choose between a single cycle simulation and a pipeline simulation. These two modes present slightly different UIs, with the former being shown in Figure 1.7 and the latter in Figure 1.8.

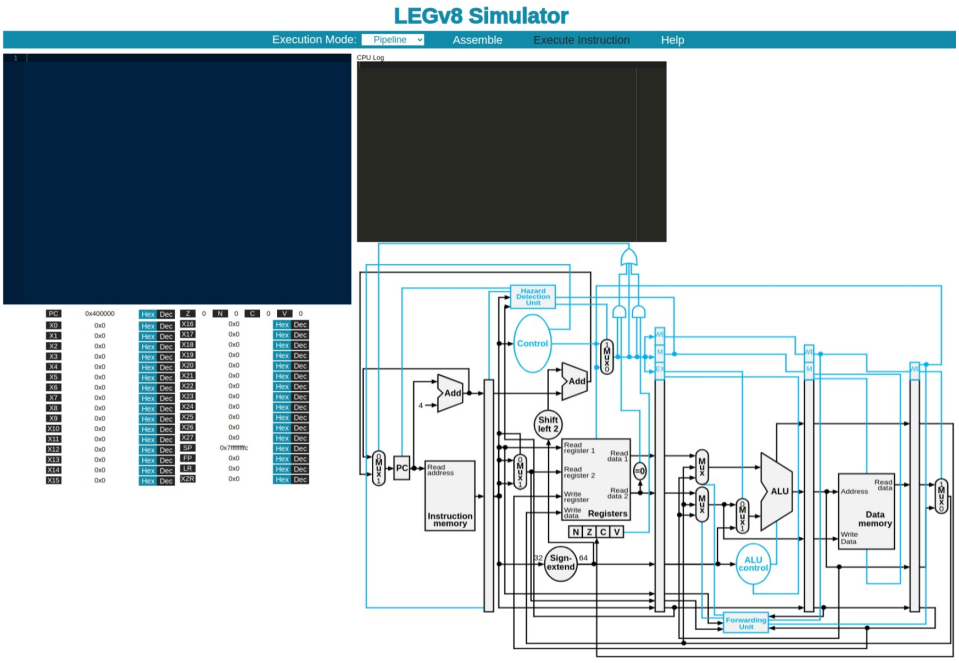


Figure 1.8: The pipeline view

The single cycle view is comprised of a text editor on the upper left, a visualization of the `PC` register, flag registers, and `X` registers just below it, and a reproduction of the LEGv8 logical diagram as previously reported in Figure 1.1. The text editor is the UI component responsible for the input of LEGv8 code. It also functions as a way to present errors to the user by showing tooltips next to the lines containing invalid code. The visualization of the registers, with the exception of the flags, allows the user to switch between a hexadecimal and a signed integer representation of the values therein stored. Lastly, the visualization of the LEGv8's logical scheme highlights in red the sections of the datapath that are being used by an instruction when it gets executed. It also shows, on the bottom left, which instruction is being executed and with what arguments, and its encoding. The pipeline visualization differs from the single cycle one by substituting the datapath diagram with a more complex one, showing more internal details of the pipeline, and by adding a read-only console above it that provides a textual representation of the instructions being executed and their position inside the pipeline. The *Assemble* button assembles the code written in the editor and automatically formats it with consistent indentations. If the code contains some syntac-

tical mistakes, the assembly operation fails and the aforementioned tooltips get displayed. The editor is quite lenient and allows both uppercase and lowercase notation, and an unrestricted use of spaces and tabs. If the code assembly is successful, the last button, *Execute Instruction*, gets enabled and allows the user to manually execute each instruction one by one until the end. As previously described, at each step the visualization of the registers, of the datapath, and of the pipeline is updated accordingly.

The good part

It is clear from this description and the attached figures, that the simulator presents itself with a nice and functional UI, and provides additional feedback to the user both in the way of error management and by visualizing the datapath just like in the textbook [1]. It is also extremely easy to execute, being a web application, and only necessitates a relatively modern web browser. This allows it to be run on a variety of devices, such as desktop computers, but also mobiles phones and tablets. Furthermore, it can be even self hosted and accessed remotely.

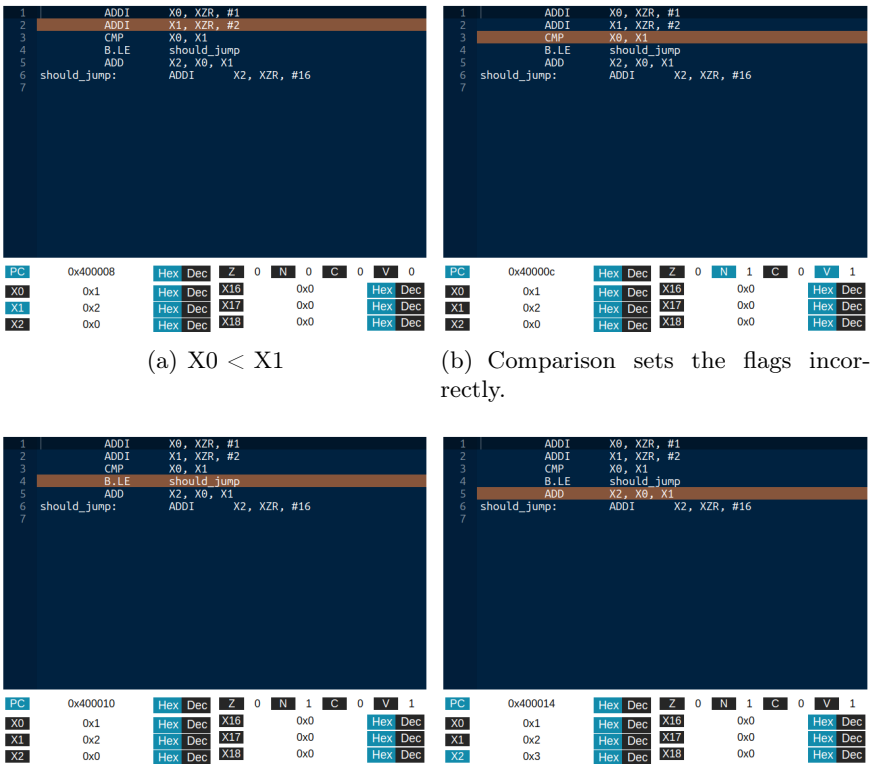
The shortcomings

Being the most popular LEGv8 simulator available, and being published by Arm, one would expect it to offer the best and most functional simulation. However, some glaring problems become evident after a cursory glance at its presentation and some simple code executions.

The visual problems Although the choice of compatible devices is vast, more restrictive is the list of devices in which the web page correctly renders. The application is clearly constructed to work best on a widescreen computer monitor, and any attempts at enlarging the page or viewing it from a smaller device results in an incorrect scaling and arrangement of the various UI elements. A UI element that instead is lacking, is a visualization of the current state of the stack memory. Even though most of the results needed to show a correct execution of the code are evident from the values inside of the registers, being able to check how the stack is accessed is crucial for debugging and to better understand the inner workings of LEGv8.

The simulation problems Trying to load or store values to the stack immediately presents a problem: the starting point of the stack pointer is not quadword aligned, and as such the program fails to assemble. Other than that, the simulator manages to work fairly well when executing a simple sequence of instructions, meaning without conditional jumps (conditional branches) nor function calls (branch and link), in single cycle mode. In pipeline mode, some combinations of instructions cause the simulation to

stall, and sometimes instructions are skipped and not properly visualized. When trying to introduce code using comparisons, loops, and user defined functions, the simulator breaks and causes a number of problems. In the case of comparisons, the simulator sets the wrong flags (which are used to determine the outcome), and the processor ends up choosing the wrong branch. In the case of loops, this means exiting them prematurely. An example of this behavior can be seen in Figure 1.9. The code is designed to force the simulator to take the jump, which doesn't happen.



(a) X0 < X1

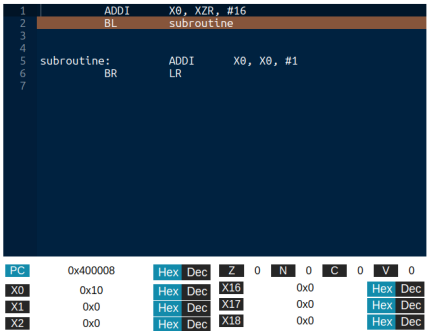
(b) Comparison sets the flags incorrectly.

(c) Less-or-equals jump doesn't happen.

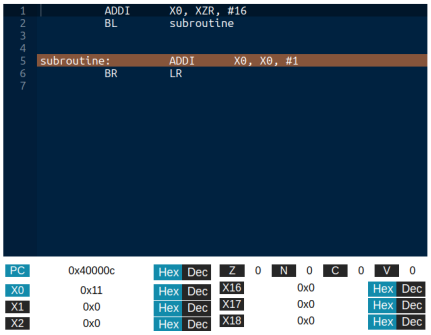
(d) Wrong instruction executed.

Figure 1.9: Comparisons do not set the correct flags and thus fail.

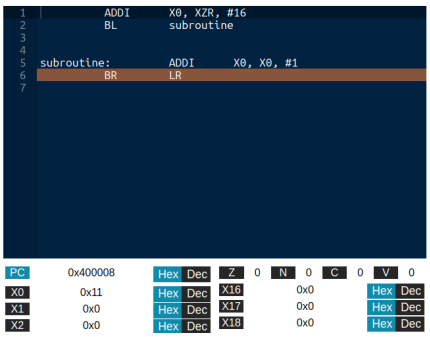
The mechanism through which functions or subroutines are executed is called a *branch and link* procedure. The way it works is that, when reaching the point of the code where the function is to be called, the processor saves in a certain register (the `LR` register in this case) the current value of the program counter. This way, when the function has finished executing, the processor knows where to come back to resume its steps. As is evident from Figure 1.10, the simulator writes the incorrect address to the `LR` register, and instead of going back to where the function was called, the simulator enters the function again creating an infinite loop.



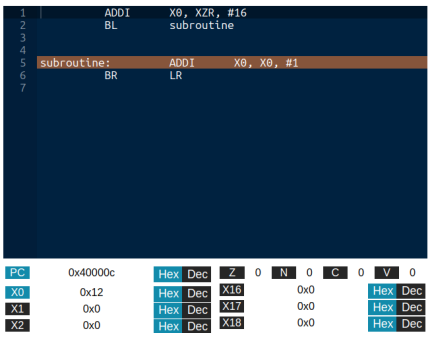
(a) BL instruction writes the incorrect address to the return register (LR).



(b) Jumps to the subroutine and increments X0.



(c) Reads wrong address from LR.



(d) Returns to the start of the subroutine instead of the main program.

Figure 1.10: Branch returns to the wrong instruction, making it execute the branch in a loop.

Although the deficiencies in the UI could cause some annoyance, it is evident that the bugs in the simulator's logic regarding conditional branching and branch and link operations greatly limit its ability to run even the most basic examples of code.

1.5.3 An inside look

Having showcased the features and criticalities of the simulator from the point of view of a casual user, it is now the time to analyze its internal logic and design choices as a developer willing to study and contribute to the project.

Some design choices Albeit deployed as a web application, the simulator is not natively written for the web, at least not its internal logic. The simulator is mostly written in Java, and uses a framework originally developed by Google called GWT [5] to build from it a self-contained web application.

GWT is a library that allows the creation of dynamic web applications from Java, both self-contained or using a client-server architecture. For its static components it uses normal HTML and CSS files placed in the appropriate folders, whereas for the client-side code it emulates a subset of the JVM [13] with JavaScript. This allows the programmer to write normal Java code which will then be translated to its JavaScript equivalent and incorporated into the final web pages. The project uses version 2.7 of the library, as can be deduced from the contents of the `LEGv8_Simulator.gwt.xml` file in the source directories, and, as can be seen by the release notes [14], the latest Java syntax supported is Java 7.

At the time of the simulator's development, a straightforward way to work with GWT was through the use of the Eclipse IDE [15] enhanced with the official GWT plugin [16]. As such, and due to the presence of Eclipse's `.project` files, it is clear that the project is specifically tailored to this IDE and plugin combination.

As we have seen from the overview of the UI, the simulator offers a featureful text editor embedded into the web page. This editor has not been developed from scratch, but has been incorporated from an existing library called AceGWT [17]. "Ace" in this case refers to an embeddable code editor for the web bearing the same name [18]. AceGWT is thus a Java interface built around the JavaScript-based Ace editor following the conventions specified by GWT. The end result is a GWT component that contains the Ace editor and that can be embedded into any GWT project. AceGWT in turn uses GWT version 2.8.2.

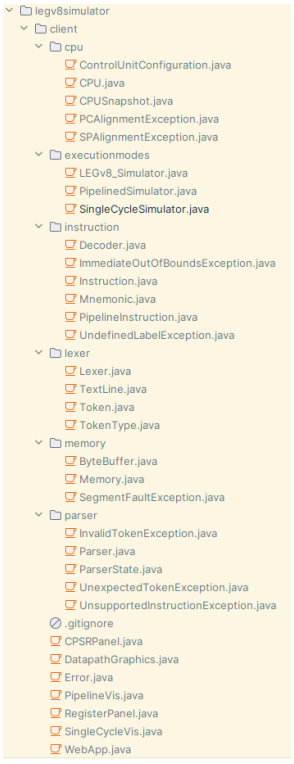


Figure 1.11: The codebase structure

The project uses the Eclipse's GWT plugin's way of building and deploying the web application. In this case, it all gets compiled into the `/war` folder. For this reason, a separate folder containing the static resources of the website (i.e., images, HTML, and CSS files) is directly included in `/war`.

The original developers of the project have not deemed it necessary to provide a tutorial on how to build the codebase. Many things can be inferred

from the structure of the project and the content of some files, but no official documentation is present anywhere in the repository.

The simulator's internals The analysis of the internal structure of the simulator will only include the Java codebase, and not the choices made regarding the web pages' style or files. Of the Java code, exception classes will not be included as they are self-describing. The source code of the project is divided into six packages, with some classes being found inside the parent `client` package, and the GWT module configuration file being located in the root `legv8simulator` packages, as can be seen in Figure 1.11.

The outer classes deal with the main web application. They specify the content of the web pages and create all the GWT components that need to be rendered on the screen. Specifically:

- `WebApp.java` : This class contains all the code to tie together the elements of the web UI. It creates all the visual componentes and inserts them into the page. It is the entry class of the application, and the one GWT looks for the building process.
- `SingleCycleVis.java` and `PipelineVis.java`: These two classes generate the visualization for the single cycle and pipeline datapaths. This is done by `DatapathGraphics.java`, a library used to generate schematic diagrams with HTML5.
- `RegisterPanel.java` and `CPSRPanel.java`: These two classes model the visualizations for the registers, program counter, and flag panels respectively.
- `Error.java`: Models the error tooltips displayed by the text editor.

The classes inside the `cpu` package are responsible for modeling the behavior of the processor, meaning the operational parts of the datapath.

- `CPU.java` and `CPUSnapshot.java`: The `CPU.java` class contains all the registers of the processor, and the ALU, implementing all the operation it is capable of performing. It fetches and decodes the instructions from the memory, communicates with it to store and load values, and performs the write back into the registers. The `CPUSnapshot.java` class provides a deep copy of the `CPU.java` state for use in the pipeline simulation.
- `ControlUnitConfiguration.java`: Models the configuration of the control unit for every type of instruction.

The `memory` package is dedicated to the modelization of the main memory of the processor.

- `ByteBuffer.java`: A utility class used to implement a byte buffer of client specified length, used to store data in big-endian format. As specified by the author, this class has been written because emulation for `java.nio.ByteBuffer` was not supported by GWT.
- `Memory.java`: This is the class used to implement the address space of LEGv8. It follows closely Patterson's and Hennessy's specification as presented in Figure 1.3 , and as such it appears like a single array of bytes. In this case it's implemented via a `HashMap<Long, Byte>` and offers various methods to read and write different amounts of bytes to it.

The `instruction` packages contains all the classes necessary to model the concept of an instruction as described by Patterson and Hennessy.

- `Instruction.java` and `PipelineInstruction.java`: These two classes model an instruction. An instruction contains its mnemonic, its arguments, its control signals for the control unit, and its position inside of the source code. The `PipelineInstruction.java` class is a wrapper for `Instruction.java` and is used in the pipelined execution.
- `Mnemonic.java`: An enumerator class containing the name, OPcode, type, and ALUcode of all of the implemented instructions.
- `Decoder.java`: This class is responsible for taking the tokenized source code from the parser, and converting each line into a valid instruction.

The `lexer` and the `parser` packages, although separate, can be considered as a single entity. Their purpose is to take the code written inside the web editor and perform various steps to verify its correctness and produce a tokenized version to be used later for the decoding of the instructions.

- `Lexer.java`: The class responsible for scanning the code line by line and tokenizing it. The `TextLine.java` class models a single line of code, the `Token.java` models a token, and the `TokenType.java` is an enum class containing the corresponding regular expressions to tokenize each part of the line of code.
- `Parser.java`: The class responsible for checking the validity of LEGv8 code by passing each line's tokens obtained by the lexer through a finite state machine defined by `ParserState.java`, whose structure can be seen in Figure 1.12.

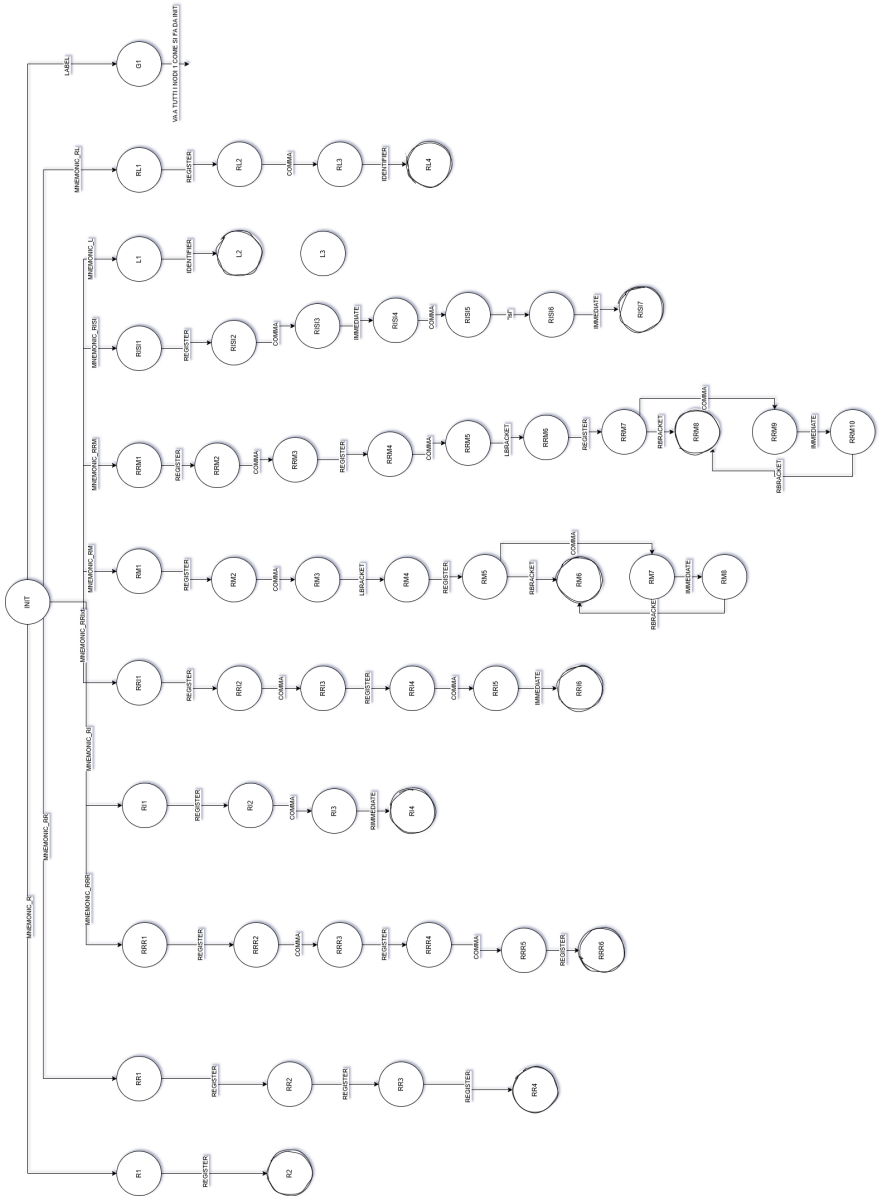


Figure 1.12: Diagram of the parser's states

Lastly, the `executionmodes` package contains the logic necessary to initialize the simulator and being the execution through either a single cycle or pipelined mode.

- `LEGv8_Simulator.java`: The abstract class responsible for setting up the simulation, initializing the various components of the processor, and signalling the CPU when to perform the various stages of the LEGv8 execution pipeline.
- `SingleCycleSimulator.java` and `PipelinedSimulator.java`: These two classes extend `LEGv8_simulator.java` and are responsible for making the proper adaptations to the code to fit the respective execution mode.

Now that this brief overview of the codebase has come to an end and some context has been provided, we will conclude the chapter by discussing the positives and negatives of the internal design choices of the simulator, and in the end explaining the motivations for choosing it as the subject of this thesis' work.

The good The simulator is written in Java, which makes it easy to structure, extend, and fix thanks to its high level design, and offers quality of life improvements in the form of the Java library to expedite development. Being deployed as a web application, it makes it compatible with most devices currently available and executable in a few clicks with most browsers. Furthermore, GWT allows the developers to focus on a single programming language without worrying too much about the complications of web programming. The codebase is fairly well structured with extensibility in mind, and most functions of the simulator reside in the correct packages and classes. With the exceptions of the two bugs previously mentioned, the instructions are executed correctly and the simulator works properly.

The bad The bugs plaguing the current iteration of the simulator can be easily spotted inside the codebase, but without a way to compile it into a working application they remain unfixed. The simulator does not implement the entirety of the LEGv8 ISA, and as such, even though its structure is apt for extension, the logic is designed to work with integer registers, values, and instructions. Eclipse, although featureful, is not the most advanced IDE to work with and doesn't offer any real advantages over the others. The pipelined execution mode has not been completed and is not well structured, which causes problems in identifying its issues and extending it with new features.

And the ugly GWT is an extremely old library for creating web applications. It has been abandoned by Google and is now maintained by the community through infrequent updates. The version used in the project uses an outdated build system that is being abandoned by the newer versions of the library, and strongly limits the Java features that can be used in the project. Similarly, AceGWT is a one-off library that is not being actively updated with newer versions of the Ace editor nor GWT. It is a crucial dependency of the project and would require parallel efforts to bring it up to current standards. Because of GWT and its outdated build process, the project is deeply coupled with the Eclipse IDE and its configuration has been slightly customized from the default one since it's a client-only application. Furthermore, newer releases of the Eclipse IDE (newer than 2023-09) break the installation of the GWT plug-in because of a change in libraries, making it impossible to develop with it. Certain features of the codebase, such as the datapath visualization for example, are complex and very sparsely documented, making it difficult to follow their logic. The single cycle mode was developed before the pipeline version, with the latter being adapted on top of the former later on. For this reason, it doesn't feel like the simulator is truly executing the pipeline stages individually, and this contributes to the instability of this execution mode.

1.6 Motivations for choosing Arm's simulator

Although the simulator presents some criticalities both from the point of view of the end user and the developer that leave it in a barely working state, we feel that its general structure, presentation, and prominence in the LEGv8 landscape make it stand out as the most impactful project to work on, and, when considering the work to be done, give it the prospect of being the most feature complete LEGv8 simulator available.

Not all the issues raised in this final section of the chapter have been tackled by this thesis' work. For example, the pipelined mode and the general structure of the codebase have been mostly left untouched, and, save for a few enhancements to the UI, improvements to the HTML and CSS have not been taken into consideration. Nonetheless, in the following chapters we will present the work done to restore the simulator to a working state and to enhance it to the point of providing a full simulation of the LEGv8 ISA, at least where single cycle mode is concerned.

Chapter 2

Building and modernizing the project

> "Thank goodness we don't have only serious problems, but ridiculous ones as well."
>
> *Edsger W. Dijkstra* – EWD475

In this chapter we will discursively lay out the steps that have been initially taken to get the simulator to build, as intended by the original developers. After this, we will go through the changes made to the project to update its dependencies and integrate it with the Maven [4] build automation system. Complete step-by-step guides for setting up the original development environment and the newly improved one will be provided in Appendix A and Appendix B respectively. Although the modernization of the codebase has been the last thing to be achieved in this thesis' work, it is a step that can be considered independent from the development efforts described in the next chapters, and as such it was decided to discuss it in conjunction with the original build system to offer a more consistent exposition.

2.1 Reconstructing the original set-up process

As was mentioned in Section 1.5, an initial look at the repository's source code and `README` makes evident a lack of documentation over how to set-up the development environment to build and deploy the project. Additionally, exploring the Issues section of the GitHub's repository [11] makes it clear that there have been some failed attempts at getting the codebase to compile, with some users even pointing out bugs in the code but being unable to build the project themselves and produce a fixed version. It can be safely asserted

then, that the lack of documentation has been at least partially responsible for the project sitting dormant throughout the years and not receiving needed improvements which would have solved some of the issues discussed in this thesis. Thus, this initial effort of understanding the set-up process for the simulator has been of crucial importance both for this thesis' work and to enable future external contributions.

Through an initial phase of trial and error when looking at the project's structure, the presence of a `.project` file indicates that the simulator has been at least partially developed using the Eclipse IDE [15], so this will be our starting point to get things back to a working state.

2.1.1 Installing the Eclipse IDE and JDK

The Eclipse IDE can be downloaded from its official repository [15]. At the time these steps were originally taken, circa the end of 2023, the current version of Eclipse was 2023-09. Due to some breaking changes introduced to the IDE in later updates, it is recommended to download the 2023-09 version of Eclipse for working with this specific project. Once the download is done, the IDE for Java can be installed normally following the recommended process.

After this, it is time to choose an appropriate JDK version. For the time being, JDK version 8 is the safest bet when working with older projects such as this one. One can choose to download any distribution of JDK 8. The choice made for this thesis has been Amazon's Corretto 8 [19]. Once installed or extracted, we can tell Eclipse to use this version of the JDK as default.

Once the project has been imported to Eclipse, the IDE will complain about some components missing. These will be our next clues.

2.1.2 GWT, AceGWT, and the final set-up

After the project has been imported, Eclipse will automatically recognize the presence of some GWT code and will suggest installing the GWT plug-in [16] from the Eclipse Marketplace. Although the automatic procedure presents some problems and should be canceled, we can manually install the plug-in ourselves.

After the plug-in installation, the IDE presents us with some configuration errors, and the place where missing dependencies and configuration problems are displayed is the Build Path section of the project's settings. By going there, a few things are made obvious: the project is missing a GWT SDK library and it requires an additional project, AceGWT, to work.

We can download GWT from the official website [5]. We might be tempted to download the latest version, but with some intuition about the project's age and from a cursory glance at the `LEGv8_Simulator.gwt.xml` file, we see that the version needed is 2.7 [14]. Having completed the download, we can tell Eclipse to use this library for the GWT SDK, and, after a short configuration, the dependency will be satisfied.

Similarly, we need to download the AceGWT library version 1.0.0 (currently, the only one available) from the Releases section of its GitHub repository [17]. After doing so, we can import the sources into the project, which will appear as an additional Java package inside of the project's tree. Again, the dependency will be satisfied and the IDE will drop its error.

GWT uses an XML file for configuration. This file requires a custom XML schema to specify its syntax in the form of a DTD file. This file is specified at the beginning of the XML document, and can either be a link to an online resource or the path to a local file on the developer's file system. Since the project uses GWT 2.7, and AceGWT uses GWT 2.8.2, two versions of this schema are required. The schema in both cases is specified as an URL, so we can either enable Eclipse's automatic downloads from the program's settings, or we can download the DTD files manually and point to their path inside the XML files. Both options are valid, although the second one is recommended, since we cannot be sure of the future availability of those online resources. In any case, Eclipse will drop the remaining issues and the project can be finally built.

2.1.3 Building and deploying the project

After installing the GWT plug-in, a new button will have appeared on the toolbar of the IDE, bearing the GWT logo. In order to build the project, we can either click that button or right click on the project's root, navigate to the GWT option, and start the compile process. The build process can be customized with differing levels of obfuscation for the final JavaScript code. In our case, the most verbose option is recommended. After compilation has ended, the old version of the simulator inside the `/war` folder will be overwritten with the new one.

It is clear from this description of the original set-up, and even more evident in Appendix A, that this process is quite long, fragile, and dependent on many user interactions and the visual appearance of the Eclipse IDE. In the next sections we will discuss the improvements that have been made to it, following some steps inspired by the newer features of the latest versions of GWT and AceGWT.

2.2 Updating GWT, AceGWT, and the JDK

By browsing the changelogs of more recent versions of GWT, and by taking a look at the tutorial page [20], we learn that the library has moved from its original build process to Maven. The `README` for AceGWT also mentions a switch to Maven, although only for its source code version and not the prepackaged 1.0.0 release. Because of these changes, and because of the aims of this thesis, integrating the simulator's codebase with Maven became the natural path to take.

What is Maven? Maven [4] is a build automation system developed by the Apache Software Foundation [21]. It is a tool to simplify and automate the management of dependencies, configuration, compilation, and deployment of Java-based applications.

The JDK

We start with GWT. Its latest version, at the time of writing, is version 2.11. As can be seen from its changelog [22], this version deprecates the old method for building GWT applications, introduces compatibility with JDK 21, and adds emulation for some JDK 11 features. This allows us to upgrade the JDK all the way to the latest long term version, and unlock some new Java syntax to use in the simulator. We can thus begin with downloading JDK 21 LTS from our preferred vendor and instruct Eclipse to use it.

AceGWT's GWT

Since AceGWT uses GWT internally and has moved to Maven in its latest version, we can attempt to build an updated version. It is only necessary to change GWT's XML configuration to point it to the newest version, and then update Maven's `pom.xml` file to use the latest repository of the library. This is necessary not only because of the newer version number, but also because, in the meantime, the project has been abandoned by Google and left in the hands of the community. In fact, both the simulator and AceGWT were developed before this change happened. Because of this, GWT's package and repository have been stripped of the original Google denomination and taken on a new name.

Having performed these small changes, and following the new Maven's procedure to build GWT projects, this updated version of AceGWT can be compiled into a working library. This means that, unfortunately, we cannot use AceGWT's original repository anymore and have instead to rely on this custom version. In order to make these steps more reproducible, a more detailed description of this operation can be found in Appendix B.

Everything is now ready to integrate Maven into the main project.

2.3 Porting the simulator to Maven

Now that all the dependencies have been properly upgraded to their latest version using Maven, integrating them into the simulator becomes much more straightforward. All that is needed is to follow GWT's new tutorial to build a GWT project using Maven, and configure the `pom.xml` files accordingly.

The source of the project is separated into two layers. An external one containing the license, contribution guides, `README`, etc., and an internal layer containing the source of the project itself. Since we want to keep the structure as close as the original as possible, we will need two `pom.xml` files. One, the "father", to describe the project externally, and one, the "child", to represent the project's code. The father `pom.xml` is configured as a standard Maven project, as the heavy lifting will be done by the other one. The `LEGv8_Simulator` folder will be the root of the actual GWT application. It contains the project's source code under `/src/main` following Maven's convention, a local Maven repository, and the project's `pom.xml` file.

Where to put AceGWT? The local repository exists to store the AceGWT library. This is because there is currently no AceGWT module in Maven's online central repository, and as such, for the time being, a local one is needed. It could be possible, through the use of Maven plug-ins, to treat a GitHub repository as a Maven one. Unfortunately, since the original AceGWT repository does not contain the additions described earlier in the document and is not properly configured, this is currently not an option. It is important to note that using a local Maven repository is heavily discouraged, and one day might even become deprecated.

How to organize the source code? There is a step during the GWT build process that was overlooked when discussing the original method for compiling the project. As we have pointed out in Chapter 1, in addition to the Java code, the project also contains the HTML, CSS, and images necessary to give the application its structure and style. These files, however, do not get automatically generated nor copied to the folder where the simulator is compiled to. Instead, they must be manually added. The reason why this wasn't a problem until now, is because they were present in the `/war` folder from the beginning, so when the build was complete they were already in the correct place. For this reason, the contents of `/main/src` have been divided into the Java source, and the static resources needed for the website, which have been copied from the `/war` folder to an appropriately named one.

How to configure the pom.xml? Everything has been put in place to create Maven's configuration file. We start by specifying its parent is the `pom.xml` file in the root of the project. After that, we use the same syntax as with AceGWT to specify that the project has GWT as a dependency and is a GWT application. Then, we inform Maven that there is a local repository containing a library in the `.jar` format, and to include it in the build process. We end by including and configuring a plug-in that automatically copies the static web files to the output folder once the build has terminated. This way there is no need to provide them beforehand outside of the sources folder, and it becomes possible to make changes to them that get reflected automatically to the output folder.

The configuration is now complete. We will end the chapter with a brief demonstration of the new workflow for setting up and building the simulator. A more detailed account of what has been described in this section is provided in Appendix B.

2.4 The updated workflow

By integrating Maven into the project, we not only managed to include updated versions of the various libraries, but we have automatically enhanced the codebase with all the benefits that Maven is able to bring. Among them, we can now set-up, develop, build, and deploy the project entirely through the terminal with just a couple of commands, and without relying on the Eclipse IDE. Not only that, it is now possible to use other Java IDEs that offer Maven compatibility, which opens up the doors to more modern and featureful tools. Tutorials for the most popular Java IDEs, including versions of Eclipse newer than 2023-09, will be provided in Appendix B, as in the next paragraph we will showcase only the headless approach.

First, obtain the project and put it in a folder of your choosing and install Maven on your operating system. After moving to the project's folder, run the `mvn package` command inside the `LEGv8_Simulator` subdirectory. The build process will begin, and, once terminated, the folder containing the compiled simulator will appear under `LEGv8_Simulator/target/` instead of `/war`. The specific name of the output folder will depend on how the `pom.xml` file is configured.

2.5 Conclusions

In this chapter we have showcased the undocumented, and probably unoptimized, process of setting up and building the original codebase. What

followed was a process of modernization and simplification of the entire procedure by integrating Maven in the project and its dependencies. The final result is now a much simpler series of steps that offer not only a high level of automation and customization, but completely decouple the project from its outdated IDE choice and opens it up for seamless collaboration from developers using different tools and settings. The next chapter will discuss the final touches done to bring the simulator to an acceptably working state.

Chapter 3

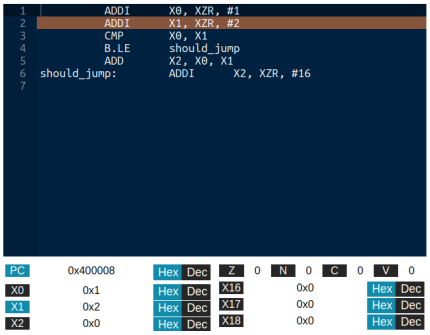
Solving the initial issues

> "Much of the excitement we get out of our work is that we don't really know what we are doing"
>
> *Edsger W. Dijkstra*

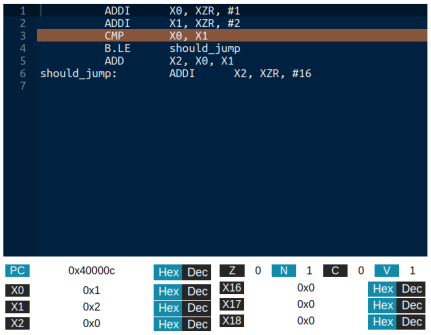
In this short chapter we will present the fixes for the initial bugs in the logic of the simulator discussed in Section 1.5.2 that were performed in order to bring it to an acceptably working state. In the end, an additional fix will be presented concerning the datapath visualization. This last fix was performed near the end of the thesis' work, but, since it doesn't per se represent an addition to the simulator, it was decided to include it in this chapter. Both in this and the following chapters, the source code will not be presented in its original form, but slightly cleaned up and simplified to highlight only the essential components.

3.1 Fixing the comparison behavior

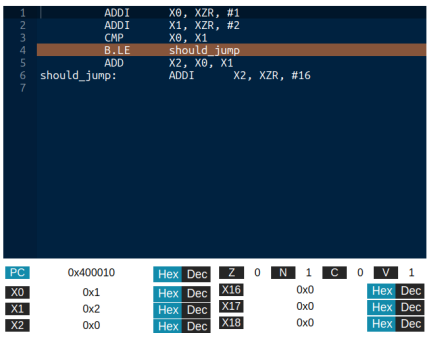
As we have showcased in Section 1.5.2, the simulator seems unable to perform comparisons in the correct way, and ignores a conditional jump that it should have instead followed. Figure 3.1 shows again the aforementioned execution flow.



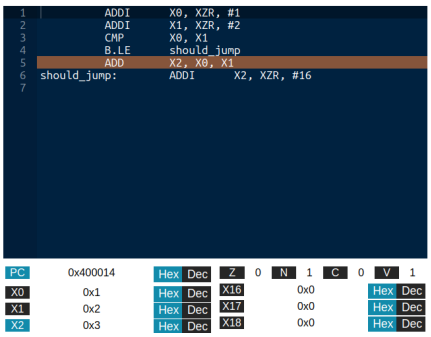
(a) X0 < X1



(b) Comparison sets the flags incorrectly.



(c) Less-or-equals jump doesn't happen.



(d) Wrong instruction executed.

Figure 3.1: Comparisons do not set the correct flags and thus fail.

First thing to note is that `CMP` is a *pseudo-instruction*. This term refers to virtual instructions that have their own mnemonic and syntax, but under the hood are implemented using one or more "real" instructions. In short, they are useful aliases for the developers. The overview of the codebase in Section 1.5.3 directs us to look into the `Decoder.java` class. Here we find that the developers implemented the `CMP` instruction following the way it's described in the LEGv8 specification, namely as an alias for the `SUBS` instruction. This last instruction performs the integer subtraction between two registers, writes the result to a third one, and sets the flag registers according to the criteria listed in Section 1.3.2. Since what we are interested in is not the actual subtraction, but to compare the two registers, `CMP` only has two arguments instead of three, and it gets translated to a `SUBS` instruction in which the destination register of the subtraction is `XZR` (i.e. `X31`, the constant zero register), as shown in Listing 3.1. After performing the comparison through `SUBS`, no registers are written to, and the flags are set. At this point, to determine the outcome of a conditional jump instruction (such as in this case `B.LE`, meaning branch if less or equal), the flags are read and interpreted, and the jump follows accordingly. Figure 3.2 shows the

```
case CMP :
    return new Instruction(Mnemonic.SUBS, ...);
...
private static int[] decodeCMPArgs(ArrayList<String> args) {
  int[] operands = new int[3];
  operands[0] = decodeRegister("XZR");
  operands[1] = decodeRegister(args.get(0));
  operands[2] = decodeRegister(args.get(1));
  return operands;
}
```

Listing 3.1: The CMP mnemonic being translated into a SUBS instruction

	Signed numbers		Unsigned numbers	
Comparison	Instruction	CC Test	Instruction	CC Test
=	B.EQ	Z=1	B.EQ	Z=1
≠	B.NE	Z=0	B.NE	Z=0
<	B.LT	N!=V	B.LO	C=0
≤	B.LE	~(Z=0 & N=V)	B.LS	~(Z=0 & C=1)
>	B.GT	(Z=0 & N=V)	B.HI	(Z=0 & C=1)
≥	B.GE	N=V	B.HS	C=1

Figure 3.2: Conditions for each conditional jump instruction in the case of subtraction. From *Computer Organization and Design (ARM Edition)* [1] p. 98

jump conditions when the flags are set by a subtraction. From the previous description, two options present themselves: either `SUBS` sets the wrong flags, or `B.LE` interprets them incorrectly. Let's look at `SUBS` in Listing 3.2. The subtraction seems to work fine, but the actual flag-setting is being performed by a helper function `SUBSetFlags`, shown in Listing 3.3. As we can see, the function itself makes use of `ADDSetFlags`, a function responsible for setting the flags for the addition instructions. This is strange, because `ADDSetFlags` is being called with the same arguments as `SUBSetFlags` even though the

```
private void SUBS(int destReg, int op1Reg, int op2Reg) {
    registerFile[destReg] = registerFile[op1Reg] - registerFile
    [op2Reg];
    SUBSetFlags(result, registerFile[op1Reg], registerFile[
    op2Reg]);
}
```

Listing 3.2: SUBS' implementation

```
1 private void SUBSetFlags(long result, long op1, long op2) {
2   ADDSetFlags(result, op1, op2);
3 }
```

Listing 3.3: Function that sets SUBS' flags

```
1 private void SUBSetFlags(long result, long op1, long op2) {
2     ADDSetFlags(result, op1, (~(op2)+1));
3 }
```

Listing 3.4: The fixed function

criteria for setting the flags in the case of an addition are different than with subtraction. We remember, though, that we can interpret a subtraction as an addition and vice versa, so all that is needed to fix this behavior is to call `ADDSetFlags` with the two's complement of `op2` as depicted in Listing 3.4, thus eliminating a need for two implementations. Testing the various jump conditions with `CMP` after this change gives positive results, proving that `ADDSetFlags` was implemented correctly and this was the source of the issue.

3.2 Fixing the branch and link behavior

The other bug that was discussed, concerned an incorrect behavior when performing function calls. The purpose of the branch and link instruction `BL` is to jump to a label representing a function or subroutine, and then, through a branch return `BR` instruction, go back to where the function was originally called and resume execution. To achieve this, as opposed to a normal branch, two additional steps have to be performed. Firstly, the program counter pointing to the original function call needs to be saved somewhere. This is performed automatically by the `BL` instruction by saving the address to the appropriately named *link register* `LR` (i.e., `X30`). Secondly, the `BR` instruction needs to be told to jump back to the aforementioned address. Unlike with `BL`, `BR` takes a register as an argument and thus doesn't automatically assume it needs to be `LR`, meaning we can in theory tell it to jump to user-defined locations in the source code. The usual case, though, is to tell the `BR` to jump to the instruction pointed by `LR`. As was the case in the last section, either the `BL` instruction sets the wrong return address, or the `BR` instruction somehow interprets it incorrectly. We can see in Listing 3.5 how `BL` is implemented in `CPU.java`, and immediately we can spot the problem. `instructionIndex` first gets updated with the address of the function, and *then* gets written to the `LR` register, meaning that `LR` contains the address of the function's label and not the current value of the program counter. Other than that, the code is correct, and all that is needed is to invert the order of operation as shown

```
1 private void BL(int branchIndex) {
2     instructionIndex = branchIndex;
3     registerFile[LR] = instructionIndex * INSTRUCTION_SIZE +
      Memory.TEXT_SEGMENT_OFFSET;
4 }
```

Listing 3.5: BL's implementation

```
1 private void BL(int branchIndex) {
2     registerFile[LR] = instructionIndex * INSTRUCTION_SIZE +
      Memory.TEXT_SEGMENT_OFFSET;
3     instructionIndex = branchIndex;
4 }
```

Listing 3.6: BL's correct implementation

in Listing 3.6.

3.3 Realigning the stack

As mentioned in Chapter 1, the stack grows from the top of the memory following an address called the *stack pointer* (SP). The authors of the textbook [1] specify that its starting value must be *quadword aligned*, meaning that, in LEGv8, it must be a multiple of 128. For some reason though, the authors set the stack pointer's initial value to $\texttt{7fffffffc}_2$, which is not quadword aligned. It has thus been chosen to adjust this value to $\texttt{8000000000}_2$, fixing the problem.

3.4 Fixing a datapath visualization bug

A bug that was addressed much later during this thesis' work, was first pointed out on the project's GitHub page [11] under the Issues section by user `aaarbk`. The bug consisted in the `Memread` and `Memwrite` signals being given the wrong values in the datapath visualization for the single cycle execution mode. Namely, they appear to behave oppositely than expected, with `Memwrite` being 0 when writing to the memory and `Memread` being 1. The code responsible, shown in Listing 3.7, is divided between `SingleCycleVis.java` and `ControlUnitConfiguration.java`. Due to the behavior of the bug, it's easy to see that in `ControlUnitConfiguration.java`, the `Memwrite` and `Memread` signals (namely, the 5th and 7th argument) for `RM_LOAD` type instructions are swapped and, similarly, in `SingleCycleVis.java` `c.memRead` and `c.memWrite` are swapped too. Listing 3.8 shows the correct code. This concludes the discussion of the original bugs present in the simulator that have been fixed during this thesis' work.

```
ctx.fillText(c.memRead, ...);
ctx.fillText(c.memWrite, ...);
...
/**
* @param reg2Loc    the value of the Reg2Loc control signal
* @param uncondBranch the value of the UncondBranch control
    signal
* @param flagBranch the value of the FlagBranch control signal
* @param zeroBranch the value of the ZeroBranch control signal
* @param memRead    the value of the MemRead control signal
* @param memToReg   the value of the MemToReg control signal
* @param memWrite   the value of the MemWrite control signal
* @param flagWrite    the value of the FlagWrite control signal
* @param aluSrc   the value of the ALUSrc control signal
* @param aluOp      the value of the ALUOp control signal
* @param regWrite   the value of the RegWrite control signal
*/
...
RM_LOAD(..., false, true, true, ...),
```

Listing 3.7: The old Memread signal logic

```
ctx.fillText(c.memWrite, ...);
ctx.fillText(c.memRead, ...);
...
RM_LOAD(..., true, true, false, ...),
```

Listing 3.8: The fixed Memread signal logic

3.5 Conclusions

Fortunately, the bugs discussed, at least the first two, were as critical as they were simple to fix. Both of them were probably a result of the developer's distraction, but greatly impacted the viability of the simulator. While this chapter talked about changes of the original code that simply fixed undesired behavior, the next chapter will present all the notable additions that have been made to the simulator, expanding on its original design.

Chapter 4

Introducing new capabilities

> "If debugging is the process of removing software bugs, then programming must be the process of putting them in."
>
> *Edsger W. Dijkstra*

In this chapter we will describe all the major additions that have been made to Arm's simulator. We will focus on discussing the changes that have introduced new functionalities, whereas the ones that were limited to refactoring the code without modifying its logic will only be briefly mentioned. This chapter can be logically divided into three parts: the work done to finish introducing all of the integer-based instructions, the implementation of the floating-point logic, and the necessary updates to the UI of the web application.

4.1 Finishing the integer job

As was briefly mentioned in Chapter 1.5.3, the simulator's logic is actually quite well structured and extensible when dealing with integer instructions, since the developers tailored everything, from the format of the instructions to the data structure of the registers, to work using integers. From this aspect, adding the remaining integer-based instructions to the simulator has been a simple work of extending the logic with new mnemonics and branches. However, some issues presented themselves due to some quirks of the Java language, which made implementing some instructions not as straightforward as it could have been.

```
public enum Mnemonic {
  ADD("ADD", "add", TokenType.MNEMONIC_RRR, "10001011000"),
  ADDS("ADDS", "adds", TokenType.MNEMONIC_RRR, "10101011000"),
  ADDI("ADDI", "addi", TokenType.MNEMONIC_RRI, "1001000100"),
  ...
}
```

Listing 4.1: How mnemonics are defined

```
public static Instruction getInstruction(...) {
  switch (mnemonic) {
    case ADD :
      return new Instruction(mnemonic, ...);
    case ADDS :
      return new Instruction(mnemonic, ...);
    case ADDI :
      return new Instruction(mnemonic, ...);
    ...
}
```

Listing 4.2: The creation of instructions

4.1.1 The logic around integer instructions

Before exploring the individual instructions, we first need to explain the work necessary to introduce them to the pre-existing logic. The simulator has a few places where instructions are read, parsed, decoded, created and executed, and as such, when adding a new instruction, they have to be extended.

First of all, to introduce a new instruction we must first define its representation, meaning its mnemonic, encoding, and type. This is done inside the `Mnemonic.java` class. As we can see from an excerpt in Listing 4.1, we need to provide both the uppercase and the lowercase mnemonic for the instruction, its type (in this example, `RRR` for instructions that operate on three registers, `RRI` for ones that operate on two registers and one immediate value), and its encoding (usually the one provided in the textbook [1] as shown in Figure 1.5).

It is now the turn of the `Decoder.java` class. As was detailed in the overview of the codebase, this is the class responsible for taking a line of parsed LEGv8 code and creating the actual instruction as defined in `Instruction.java`. As we can see from Listing 4.2, this is done through a switch case by analyzing the mnemonic and instantiating the instruction with the given parameters.

```
public enum TokenType {
  // The order in which these enums are defined is important to
     get the longest RegExp match
  LBRACKET("\\[", 1, "["),
  RBRACKET("\\]", 2, "]"),
  ...
  REGISTER("[Xx][12][0-9]|[Xx]30|[Xx][0-9]|XZR|xzr|SP|sp|LR|lr|
    FP|fp|IP[01]|ip[01]", 5, "REGISTER"),
  ...
  MNEMONIC_RRI("ADDIS?[ \t]+|SUBIS?[ \t]+|..."),
  MNEMONIC_RRR("ADDS?[ \t]+|SUBS?[ \t]+|..."),
  ...
}
```

Listing 4.3: Validating the syntax with regular expressions

```
private void execute(Instruction ins, Memory memory) {
  switch (ins.getMnemonic()) {
    case ADD :
      ADD(args[0], args[1], args[2]);
      break;
  }
}
private void ADD(int destReg, int op1Reg, int op2Reg) {
  registerFile[destReg] = registerFile[op1Reg] + registerFile[
    op2Reg];
}
```

Listing 4.4: How an instruction is executed

`TokenType.java` contains the definition of the regular expression required to correctly tokenize the LEGv8 syntax. We can see from Listing 4.3, that the mnemonic of the valid instruction needs to be reported both in uppercase and lowercase in the appropriate enum case and added to the regular expression string.

Finally, it comes time to implement the instruction's logic into `CPU.java`. As we can see from an example in Listing 4.4, this is done by adding a new method with the appropriate arguments, and adding a case to the switch governing which instructions get executed.

When implementing integer-based instructions, these are the necessary surrounding changes that need to be performed. Since the process has been now explained, from now on, in this section, we will focus only on the methods added to `CPU.java`.

```
1 private void MUL(int destReg, int op1Reg, int op2Reg) {
2   registerFile[destReg] = registerFile[op1Reg] * registerFile[
      op2Reg];
3 }
```

Listing 4.5: Implementation of the MUL instruction

4.1.2 MUL

Multiplicative operations between two 64-bit numbers can end up giving a 128-bit result ($2^{64} \cdot 2^{64} = 2^{64+64} = 2^{128}$). Since LEGv8 doesn't support 128-bit registers, how is multiplication handled? The `MUL` instruction performs the product between two integers and simply writes the lower 64-bits of the multiplication to the destination register. In this case, the multiplication algorithm works the same for signed and unsigned values: if the result stays inside the 64-bit bounds, then no 128-bit extension is required and thus the sign is maintained, and if the result gets truncated then its sign is meaningless without the context of the higher 64 bits. Since Java automatically truncates the multiplication between integers the same way, it is easily implemented as can be seen from Listing 4.5.

4.1.3 SMULH

This instruction, again, multiplies together the values inside two `X` registers, but in this case writes to the destination the higher 64 bits of the result, interpreting them as a signed multiplication. When multiplying two 64-bit numbers, if we care about the higher 64 bits, they have to be both extended to 128 bits, and this extension operation needs to be aware if they are signed or unsigned for it to be performed correctly. In this case, we specify that the two values are signed. Here we encounter a limitation of Java's primitive integer types. Namely, they can be at most 64 bits long and their multiplication gets truncated just like with `MUL`. For this reason, we need a way to perform products with signed integer values bigger than 64 bits. Here, Java's `BigInteger.java` library and GWT's support for it come to the rescue, as they allow us to create arbitrarily wide signed integer numbers. As we can see from Listing 4.6, we just need to convert the two 64-bit `long` values into a `BigInteger` and then use the provided `multiply()` method to obtain the result. After this is done, we shift the higher 64 bits to the lower ones, and we convert it to a `long` value to be written to the destination register.

4.1.4 UMULH

Following the prototype of the `SMULH` instruction, `UMULH` performs the same operation but interpreting the multiplication as unsigned. The logic is largely unchanged, except for another limitation of the Java language. Java does not

```
1 private void SMULH(int destReg, int op1Reg, int op2Reg) {
2   BigInteger fullResult = BigInteger.valueOf(registerFile[
      op1Reg].multiply(BigInteger.valueOf(registerFile[op2Reg]));
3   BigInteger shiftedResult = fullResult.bitLength() > 64 ?
      fullResult.shiftRight(64) : BigInteger.valueOf(0);
4   registerFile[destReg] = shiftedResult.longValue();
5 }
```

Listing 4.6: Implementation of the SMULH instruction

```
1 BigInteger UNSIGNED_LONG_MASK = BigInteger.ONE.shiftLeft(Long.
    SIZE).subtract(BigInteger.ONE);
2 ...
3 private void UMULH(int destReg, int op1Reg, int op2Reg) {
4   BigInteger fullResult = BigInteger.valueOf(registerFile[
      op1Reg].and(UNSIGNED_LONG_MASK).multiply(BigInteger.valueOf
      (registerFile[op2Reg]).and(UNSIGNED_LONG_MASK));
5   BigInteger shiftedResult = fullResult.bitLength() > 64 ?
      fullResult.shiftRight(64) : BigInteger.valueOf(0);
6   registerFile[destReg] = shiftedResult.longValue();
7 }
```

Listing 4.7: Implementation of the UMULH instruction

have dedicated types for unsigned integers, both in the case of `BigInteger` and of primitive types. This means that to perform an unsigned multiplication, we need a workaround to represent unsigned values as signed ones in such a way to obtain the correct results using the provided signed operations. This can be achieved, in this case, by applying a particular bitmask [23] as defined in Listing 4.7. This bitmask is used to convert 64-bit unsigned values into 65-bit signed ones in order to perform the correct multiplication. Even though the bits inside the register have no inherent interpretation, Java can only deal with signed `long` values, and as such tricks such as these are needed to make it work. The rest of the operations are analogous to `SMULH` case.

4.1.5 SDIV

This instruction performs the signed integer division between two 64-bit values. Since Java natively works using signed values and it doesn't automatically cast divisions between integers as floating-point, Listing 4.8 shows the elementary implementation of the instruction.

4.1.6 UDIV

This instruction performs the unsigned integer division between two 64-bit values. As has already been mentioned, because of Java's limitations regarding unsigned integer values, we have to, again, use the bitmask in

```
1 private void SDIV(int destReg, int op1Reg, int op2Reg) {
2   registerFile[destReg] = registerFile[op1Reg] / registerFile[
      op2Reg];
3 }
```

Listing 4.8: Implementation of the SDIV instruction

```
1 private void UDIV(int destReg, int op1Reg, int op2Reg) {
2   BigInteger dividend = BigInteger.valueOf(registerFile[op1Reg]
      .and(UNSIGNED_LONG_MASK);
3   BigInteger divisor = BigInteger.valueOf(registerFile[op2Reg].
      and(UNSIGNED_LONG_MASK);
4   BigInteger quotient = dividend.divide(divisor);
5   registerFile[destReg] = quotient.longValue();
6 }
```

Listing 4.9: Implementation of the UDIV instruction

Listing 4.7 to represent unsigned values as signed. Thankfully, `BigInteger` already provides a method for performing signed division, and Listing 4.9 shows the simple steps needed after the conversion is performed.

4.1.7 LDA

The last integer-based instruction that needs to be implemented is `LDA`, which is actually a pseudo-instruction. Its purpose is to copy the address pointed by a label into an `X` register. Since the instruction follows the same format as `CBZ` and `CBNZ` (i.e., its arguments are an `X` register and a label), it was added as an `RL`-type instruction. We should note that the LEGv8 documentation seems to be inconsistent in its description of this instruction. If we go by how it is used in the textbook's [1] examples, it appears to do what we have described above. Instead, when looking at the LEGv8 Quick Reference Guide, it's described as `R[Rd] = R[Rn] + DTAddress`, implying that it actually operates by taking the value of a register `Rn`, adding it to the address `DTAddress` specified by the label, and then writing the result into the `Rd` destination address. Furthermore, this instruction is present in just a couple of LEGv8 examples and never explained in detail, so we decided to implement it following the code examples in the book instead of its specification, as can be seen in Listing 4.10.

```
1 private void LDA(int destReg, long addressToLoad) {
2   registerFile[destReg] = addressToLoad;
3 }
```

Listing 4.10: Implementation of the LDA pseudo-instruction

4.2 Introducing floating-point capabilities

Both Java and LEGv8 support single-precision and double-precision operations following the IEEE-754 standard [8], making it easy to transpose them from one another. Because of the simplicity of the code that will be presented when showcasing the new instructions, it has been chosen to group them together by type and precision to make the exposition less verbose. More complicated, though, are the steps that were needed to adapt the original codebase to make use of floating-point values and functions. This will our next topic of discussion.

4.2.1 Setting the stage for the floating-point logic

In the previous section, we have given a brief overview of the places in the codebase where integer instructions are dealth with, and as such where one needs to look for when trying to add new ones. Although similar steps need to be taken with floating-point instructions, some changes need to be made to previously untouched parts of the simulator to open the doors to floating-point instructions.

Floating-point instructions need floating-point registers, namely `S` and `D` registers. Fortunately, `S` registers do not exist hardware-wise, as they are simply a way of referring to the lower 32 bits of the `D` registers. Nonetheless, these additions require a change in how registers are represented. If before, inside of `CPU.java`, they were a simply array of `long`s, now a new type is needed to encapsulate their logic. For this, the `Register.java` class and the `RegisterType.java` enum class have been created, their content being presented in Listing 4.11 and 4.12 respectively. As can be seen, registers are now written to and read from using dedicated method calls, which also allows to transparently write 32-bit values to the lower bits.

After performing this change, `CPU.java` now needs to hold two arrays, one of `X` registers, and one of `D` registers (with `S` registers being included for free). This is shown in Listing 4.13. We can also see that a new way of obtaining the values of the registers needed to be introduced to separate their type. In Listing 4.14 we can see an example of how the rest of the code was refactored following these changes.

The `Decoder.java` class does not deal with the type of operations being performed. Its only interest is the amount of arguments each instruction takes. Since floating-point instructions follow the same structure as integer ones, adding them to this class followed the same procedure previously described. The only addition has been creating a new method shown in Listing 4.15. Unlike with integer comparisons, floating-point ones are not an alias

```
public final class Register {
  private final RegisterType type;
  private final int size;
  private long content;
  public Register(RegisterType type) {
    this.type = type;
    switch(type) {
      case X:
      case D: this.size = Memory.DOUBLEWORD_SIZE;
        break;
      case S: this.size = Memory.WORD_SIZE;
        break;
      default: this.size = Memory.DOUBLEWORD_SIZE;
  }
    content = 0L;
  }
  public void writeDoubleWord(long bits) {
    this.content = bits;
  }
  public long readDoubleWord() {
    return this.content;
  }
  public void writeWord(int bits) {
    this.content = (this.content & 0xffffffff00000000L) OR (
    bits & 0x00000000ffffffffL);
  }
  public int readWord() {
    return (int) this.content;
  }
  public RegisterType getType() {
    return type;
  }
  public int getSize() {
    return size;
  }
}
```

Listing 4.11: The register class

```
public enum RegisterType {
  X, S, D
}
```

Listing 4.12: The RegisterType class

```
...
private Register[]  XRegisterFile;
private Register[]  DRegisterFile;
...
/**
 * @param index the register whose value to return, an integer
    in the range 0-31
 * @return    the value stored in the register <code>index</
    code>
 */
public long getRegister(RegisterType type, int index) {
  switch(type) {
    case X: return XRegisterFile[index].readDoubleWord();
    case D: return DRegisterFile[index].readDoubleWord();
    case S: return DRegisterFile[index].readDoubleWord() & 0
    x00000000ffffffffL;
    default: return 0L;
    }
}
```

Listing 4.13: The new registers

```
private void ADD(int destReg, int op1Reg, int op2Reg) {
  XRegisterFile[destReg].writeDoubleWord(XRegisterFile[op1Reg].
    readDoubleWord() + XRegisterFile[op2Reg].readDoubleWord());
}
```

Listing 4.14: The refactored register access

```
private static int[] decodeRRArgs(ArrayList<String> args) {
  int[] operands = new int[2];
  operands[0] = decodeRegister(args.get(0));
  operands[1] = decodeRegister(args.get(1));
  return operands;
}
```

Listing 4.15: New decoder for floating-point comparisons

```
public enum TokenType {
  LBRACKET("\\[", 1, "["),
  RBRACKET("\\]", 2, "]"),
  ...
  XREGISTER("[Xx][12][0-9]|[Xx]30|[Xx][0-9]|XZR|xzr|..."),
  SREGISTER("[Ss][12][0-9]|[Ss]30|[Ss][0-9]+", 6, "SREGISTER"),
  DREGISTER("[Dd][12][0-9]|[Dd]30|[Dd][0-9]+", 7, "DREGISTER"),
  ...
  XMNEMONIC_RRR("ADDS?[ \t]+|SUBS?[ \t]+|..."),
  SMNEMONIC_RRR("FADDS[ \t]+|FSUBS[ \t]+|..."),
  DMNEMONIC_RRR("FADDD[ \t]+|FSUBD[ \t]+|..."),
  ...
}
```

Listing 4.16: Refactoring of TokenType

```
public enum Mnemonic {
  ADD("ADD", "add", TokenType.XMNEMONIC_RRR, ...),
  ...
  FADDS("FADDS", "fadds", TokenType.SMNEMONIC_RRR, ...),
  FADDD("FADDD", "faddd", TokenType.DMNEMONIC_RRR, ...),
  ...
}
```

Listing 4.17: Updated Mnemonic.java

for the `SUBS` or `SUBIS` instructions. For this reason they are the only ones to truly operate on just two registers. This case needed to be added to the decoder.

Major changes needed to be performed on the `TokenType.java` class. This is because, since LEGv8 prohibits hybrid instructions, it was necessary to impose that certain instructions only deal with certain registers and they cannot be mixed together. This made it necessary to create new types of TokenTypes for each register, and for each combination of instruction type and instruction format. An excerpt of this can be seen in Listing 4.16 and can be compared with Listing 4.3. Adding floating-point mnemonics is now analogous to adding integer ones, except they need to be put in the correct dedicated regular expression.

The change to `TokenType.java` needed to be propagated to other classes making use of it. Firstly, `Mnemonic.java` needed to be updated by refactoring all the integer instruction to follow the new `TokenTypes`. Regarding the addition of floating-point instructions, the procedure is now analogous to the integer ones, except they need to be given their dedicated argument type, as shown in Listing 4.17.

Secondly, the finite state machine in `ParserState.java`, used to parse the tokens, needed to be integrated with all the new transitions it could take. This was achieved by mostly copying and adapting the existing transitions and adding them to the automata. Since hybrid instructions are not allowed, this meant simply adding new branches from the starting state, without making the logic more complex. Due to the verbosity of this class, a listing showcasing an excerpt of the changes is not provided.

Lastly, we haven't touched upon memory operations. This is because, even though the memory was designed with integer values in mind – using `long` or `int` values – this choice does not in fact hinder the introduction of floating-point values. Single and double-precision floating-point numbers are just 32 and 64-bit values which are given a certain interpretation. Since the memory does only care about storing raw bytes, we can keep using the existing logic by simply taking `float` bits and turning them into `int` bits when writing to the memory, and vice versa when reading. Same can be done between `double` bits and `long` bits. Java offers precisely these functionalities with the `Float.floatToIntBits()` and `Float.intBitsToFloat()`, and `Double.doubleToLongBits()` and `Double.longBitsToDouble()` respectively, and GWT implements them in its emulator. This also allows floating-point bits in the memory to be loaded into `X` registers and be interpreted as integers and vice versa with `S` and `D` registers, which is part of the intended behavior. `Float.floatToRawIntBits()` and `Double.doubleToRawLongBits()` should be the preferred methods, but they aren't yet supported by GWT and they don't seem to cause any problems.

This concludes the changes inside the logic of the simulator that were needed to be performed in order to enable the addition and execution of the new floating-point instructions.

4.2.2 FADDS, FDIVS, FMULS, FSUBS

As was previously mentioned, implementing LEGv8's floating-point arithmetic in Java does not present the same problems as with integer operations. This is because they both follow IEEE-754's [8] specification and as such can be easily translated between each other. Listing 4.18 provides the code for `FADDS`, `FDIVS`, `FMULS`, `FSUBS`.

4.2.3 FADDD, FDVID, FMULD, FSUBD

The same that has been said for single-precision arithmetic can be repeated verbatim in the case of double-precision. Listing 4.19 presents the new double-precision instructions, following extremely close what was done in the single-precision case.

```
private void FADDS(int destReg, int op1Reg, int op2Reg) {
  DRegisterFile[destReg].writeWord(Float.floatToIntBits(
    Float.intBitsToFloat(DRegisterFile[op1Reg].readWord()) +
    Float.intBitsToFloat(DRegisterFile[op2Reg].readWord())
  ));
}
private void FSUBS(int destReg, int op1Reg, int op2Reg) {
DRegisterFile[destReg].writeWord(Float.floatToIntBits(
  Float.intBitsToFloat(DRegisterFile[op1Reg].readWord()) -
  Float.intBitsToFloat(DRegisterFile[op2Reg].readWord())
  ));
}
private void FMULS(int destReg, int op1Reg, int op2Reg) {
  DRegisterFile[destReg].writeWord(Float.floatToIntBits(
    Float.intBitsToFloat(DRegisterFile[op1Reg].readWord()) *
    Float.intBitsToFloat(DRegisterFile[op2Reg].readWord())
  ));
}
private void FDIVS(int destReg, int op1Reg, int op2Reg) {
  DRegisterFile[destReg].writeWord(Float.floatToIntBits(
    Float.intBitsToFloat(DRegisterFile[op1Reg].readWord()) /
    Float.intBitsToFloat(DRegisterFile[op2Reg].readWord())
  ));
}
```

Listing 4.18: New single-precision arithmetic instructions

4.2.4 FCMPS, FCMPD

When dealing with comparisons, the IEEE-754 standard [8] provides all the criteria for all the possible value combinations. The results must then be translated into LEGv8's control flags. The book [1] does not include a table for floating-point flag-setting conventions like with integers in Figure 3.2. As such, we have chosen to use ARMv8's criteria [24] which are summarized in Figure 4.1.

IEEE-754 Relationship	ARM APSR Flags			
	N	Z	C	V
Equal	0	1	1	0
Less Than	1	0	0	0
Greater Than	0	0	1	0
Unordered *(At least one argument was NaN.)*	0	0	1	1

Figure 4.1: ARMv8 floating-point condition flags. From *Arm community blogs* [24]

As we can see, both in the case of single and double-precision, the outcomes relate to the same flag values. For this reason a helper function called

```
1  private void FADDD(int destReg, int op1Reg, int op2Reg) {
2    DRegisterFile[destReg].writeDoubleWord(Double.
       doubleToLongBits(
3      Double.longBitsToDouble(DRegisterFile[op1Reg].
       readDoubleWord()) +
4      Double.longBitsToDouble(DRegisterFile[op2Reg].
       readDoubleWord())
5    ));
6  }
7  private void FSUBD(int destReg, int op1Reg, int op2Reg) {
8    DRegisterFile[destReg].writeDoubleWord(Double.
       doubleToLongBits(
9      Double.longBitsToDouble(DRegisterFile[op1Reg].
       readDoubleWord()) -
10     Double.longBitsToDouble(DRegisterFile[op2Reg].
       readDoubleWord())
11   ));
12 }
13 private void FMULD(int destReg, int op1Reg, int op2Reg) {
14   DRegisterFile[destReg].writeDoubleWord(Double.
       doubleToLongBits(
15     Double.longBitsToDouble(DRegisterFile[op1Reg].
       readDoubleWord()) *
16     Double.longBitsToDouble(DRegisterFile[op2Reg].
       readDoubleWord())
17   ));
18 }
19 private void FDIVD(int destReg, int op1Reg, int op2Reg) {
20   DRegisterFile[destReg].writeDoubleWord(Double.
       doubleToLongBits(
21     Double.longBitsToDouble(DRegisterFile[op1Reg].
       readDoubleWord()) /
22     Double.longBitsToDouble(DRegisterFile[op2Reg].
       readDoubleWord())
23   ));
24 }
```

Listing 4.19: New double-precision arithmetic instructions

```
1  private void FCMPSetFlags(int comparisonResult, boolean isNaN)
       {
2    setNflag(comparisonResult < 0 && !isNaN);
3    setZflag(comparisonResult == 0 && !isNaN);
4    setCflag(comparisonResult >= 0  isNaN);
5    setVflag(isNaN);
6  }
7  ...
8  private void FCMPS(int op1Reg, int op2Reg) {
9    float op1f = Float.intBitsToFloat(DRegisterFile[op1Reg].
       readWord());
10   float op2f = Float.intBitsToFloat(DRegisterFile[op2Reg].
       readWord());
11   FCMPSetFlags(Float.compare(op1f, op2f), Float.isNaN(op1f)
       Float.isNaN(op1f));
12 }
13 private void FCMPD(int op1Reg, int op2Reg) {
14   double op1d = Double.longBitsToDouble(DRegisterFile[op1Reg].
       readDoubleWord());
15   double op2d = Double.longBitsToDouble(DRegisterFile[op2Reg].
       readDoubleWord());
16   FCMPSetFlags(Double.compare(op1d, op2d), Double.isNaN(op1d)
       Double.isNaN(op2d));
17 }
```

Listing 4.20: Floating-point comparisons implementations

`FCMPSetFlags` was created to work in both cases. As Listing 4.20 shows, both `FCMPS` and `FCMPD` use Java's built in floating-point comparison utilities and then pass the results to the helper function which sets the flags accordingly.

4.2.5 STURS, LDURS

Since the memory is type-agnostic, writing and reading from it can be done using `int`s as representations for raw 32 bits, as can be seen in Listing 4.21. Other than reading the lower 32 bits of `D` registers, the logic is analogous to the integer load and store instructions.

4.2.6 STURD, LDURD

Similarly, double-precision load and store operations use `long`s as a neutral representation of raw 64 bits. They function the same way as single-precision and integer ones, as seen in Listing 4.22.

This concludes the discussion of the changes made to the internal logic and behavior of the simulator. The following section will focus on the changes made to the UI and their motivations.

```
1 private void STURS(int valReg, int baseAddressReg, int offset,
      Memory memory) {
2   memory.storeWord(XRegisterFile[baseAddressReg].readDoubleWord
      ()+offset, DRegisterFile[valReg].readDoubleWord());
3   clearExclusiveAccessTag(XRegisterFile[baseAddressReg].
      readDoubleWord()+offset, Memory.DOUBLEWORD_SIZE);
4 }
5 private void LDURS(int destReg, int baseAddressReg, int offset,
       Memory memory) {
6   DRegisterFile[destReg].writeWord((int) memory.loadDoubleword(
      XRegisterFile[baseAddressReg].readDoubleWord()+offset));
7 }
```

Listing 4.21: Single-precision load and store instructions

```
1 private void STURD(int valReg, int baseAddressReg, int offset,
      Memory memory) {
2   memory.storeDoubleword(XRegisterFile[baseAddressReg].
      readDoubleWord()+offset, DRegisterFile[valReg].
      readDoubleWord());
3   clearExclusiveAccessTag(XRegisterFile[baseAddressReg].
      readDoubleWord()+offset, Memory.DOUBLEWORD_SIZE);
4 }
5 private void LDURD(int destReg, int baseAddressReg, int offset,
       Memory memory) {
6   DRegisterFile[destReg].writeDoubleWord(memory.loadDoubleword(
      XRegisterFile[baseAddressReg].readDoubleWord()+offset));
7 }
```

Listing 4.22: Double-precision load and store instructions

4.3 UI additions and refinements

It is clear from everything that has been discussed in this chapter and in Chapter 1.5.2, that the needs to undergo a few changes in order to incorporate all the new additions to the floating point logic, and to fix some of its shortcomings. This section will discuss the addition of the stack visualization, floating point registers, and the final reorganization of the UI elements.

4.3.1 Visualizing the stack memory

Having laid the work for the simulation of more complex programs with different types of data, being able to keep track of the state of the stack is of vital importance both for learning how the LEGv8 ISA works, and to debug more complex programs whose final results might not be enough to show the presence of bugs or errors.

If we take inspiration from how registers are visualized, we can see that they have a label, a signed integer or hexadecimal representation of their content, and a button to switch between the two. Even though the memory is addressed byte by byte, if we consider the fact that, usually, entire 64-bit values are loaded and stored, and the stack pointer only moves in quadword increments, it is more useful and intuitive to visualize the stack with panels each representing a doubleword worth of bytes, i.e. 8 of them in LEGv8's case. The address referring to each doubleword can be considered the stack cell's label. The content, unlike with registers, does not have a preferred interpretation, therefore we can keep the raw hexadecimal one. Having done these considerations, it becomes obvious that an easy way to implement the visualization of the stack is to copy how it's done with the registers. For this reason, all that was needed was to take `RegisterPanel.java` and slightly modify it to obtain `StackPanel.java`, a representation of a single stack memory cell. In the `WebApp.java` class, we just need to add an array of 32 `StackPanel`s and group it into two columns just like with the registers, obtaining the end result shown in Figure 4.2.

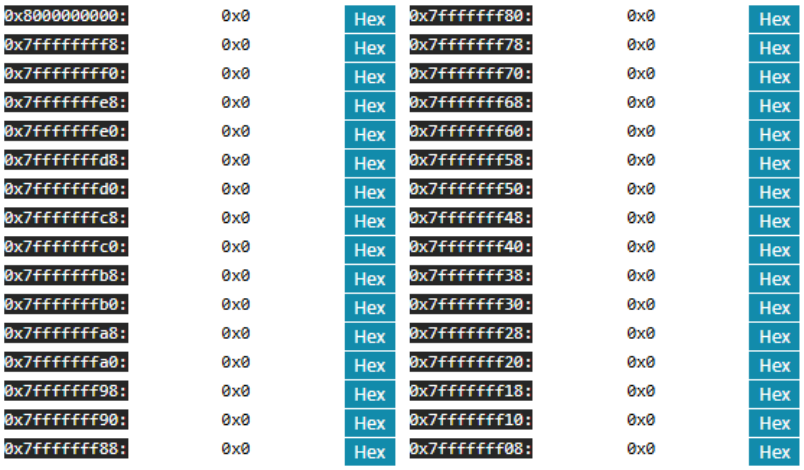


Figure 4.2: The new stack visualization

The values inside, instead of being updated with what's inside the registers, get updated at each cycle with the corresponding values in the stack. The code excerpt in Listing 4.23 shows how the stack panels are initialized and updated.

4.3.2 Showing the floating-point registers

Similarly, once floating-point capabilities were introduced, the respective visualization for the new floating-point registers needed to be added. In this case the parallels with integer registers are even stronger, and as such not even a new class was needed to implement this functionality. The only real difference, apart from the names on the labels, is the fact that floating-point values can be represented with the raw hexadecimal or with the decimal dotted notation. For this reason, the only changes made to how they work, as can be seen in Listing 4.24, were regarding their labels and the functionality of their button. The final result can be seen in Figure 4.3.

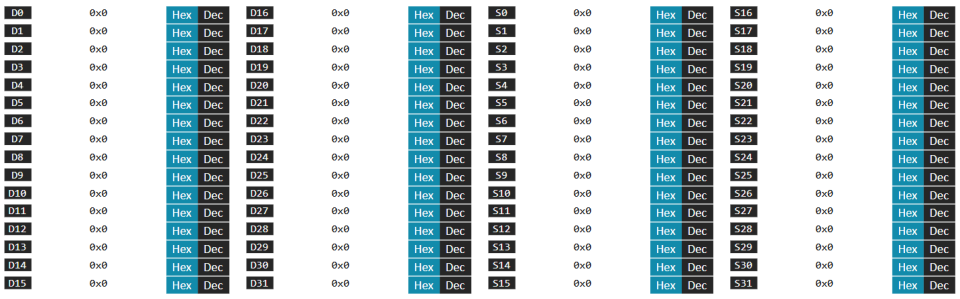


Figure 4.3: The final floating point register panels visualization

```
1  private void initStackPanel() {
2    HorizontalPanel stackFile = new HorizontalPanel();
3    VerticalPanel leftStackPanel = new VerticalPanel();
4    VerticalPanel rightStackPanel = new VerticalPanel();
5    rightStackPanel.setHorizontalAlignment(HasHorizontalAlignment
       .ALIGN_CENTER);
6    for (int i = 0; i < 32; i++) {
7      stackPanels[i] = new StackPanel(Memory.STACK_BASE - i * 8);
8      stackPanels[i].setStyleName("individualReg");
9      if (i < 16) {
10       leftStackPanel.add(stackPanels[i]);
11     } else {
12       rightStackPanel.add(stackPanels[i]);
13     }
14     }
15     stackFile.add(leftStackPanel);
16     stackFile.add(rightStackPanel);
17     stackPanel = new VerticalPanel();
18     stackPanel.addStyleName("registerPanel");
19     stackPanel.add(stackFile);
20     }
21 ...
22 private void updateStackLabels(LEGv8_Simulator sim) {
23   for (int i = 0; i < stackPanels.length; i++) {
24     try {
25       stackPanels[i].update(sim.getMemoryState().loadDoubleword
       (Memory.STACK_BASE - i * 8));
26     } catch (SegmentFaultException e) {
27         continue;
28       }
29     }
30 }
```

Listing 4.23: Stack panels' logic

```
1  private void updateRegisterLabels(...) {
2    for (int i = 0; i < XRegPanels.length; i++) {
3      XRegPanels[i].update(getCPURegister(RegisterType.X, i));
4    }
5    for (int i = 0; i < DRegPanels.length; i++) {
6      DRegPanels[i].update(getCPURegister(RegisterType.D, i));
7    }
8  for (int i = 0; i < SRegPanels.length; i++) {
9    SRegPanels[i].update(getCPURegister(RegisterType.S, i));
10   }
11   pcPanel.update(sim.getPC());
12 }
13 ...
14 private String convertToDecimal() {
15   switch(this.regType) {
16     case X: return Long.toString(regValue);
17     case D: return Double.toString(Double.longBitsToDouble(
       regValue));
18     case S: return Float.toString(Float.intBitsToFloat((int)
       regValue));
19     default: return "";
20   }
21 }
```

Listing 4.24: Logic of the floating-point registers' visualization

4.3.3 Reorganizing the UI

The additions of the new UI elements discussed in the previous sections naturally entice a partial change in the design of the web page. Furthermore, as was pointed out in Chapter 1.5.2, one of the shortcomings of the UI is that it reacts poorly to changes in page size. Due to the complexity of GWT and the components used (such as AceGWT), it was chosen to simply freeze the UI and make it inert to resizing, meaning the page simply zooms in without rearranging anything. Although this fixed position approach might make things a bit more cumbersome for small screen users, at least it gives them the ability to move around the page or even zoom out while maintaining the correct proportions and element positions.

The first page to be reorganized is the Single Cycle visualization. To accommodate the newly introduced stack and floating-point register visualizations, they have been arranged as to fill the main page. The datapah visualization, although useful, is secondary to the simulation and occupied a lot of space. It has been chosen to put it below the main simulator interface, still reachable with a slight scroll down. Lastly, the CPU log panel, being there just for debugging purposes, has been put on the bottom of the page. Figure 4.4 presents this new look, which can be compared with the original in Figure 1.7.

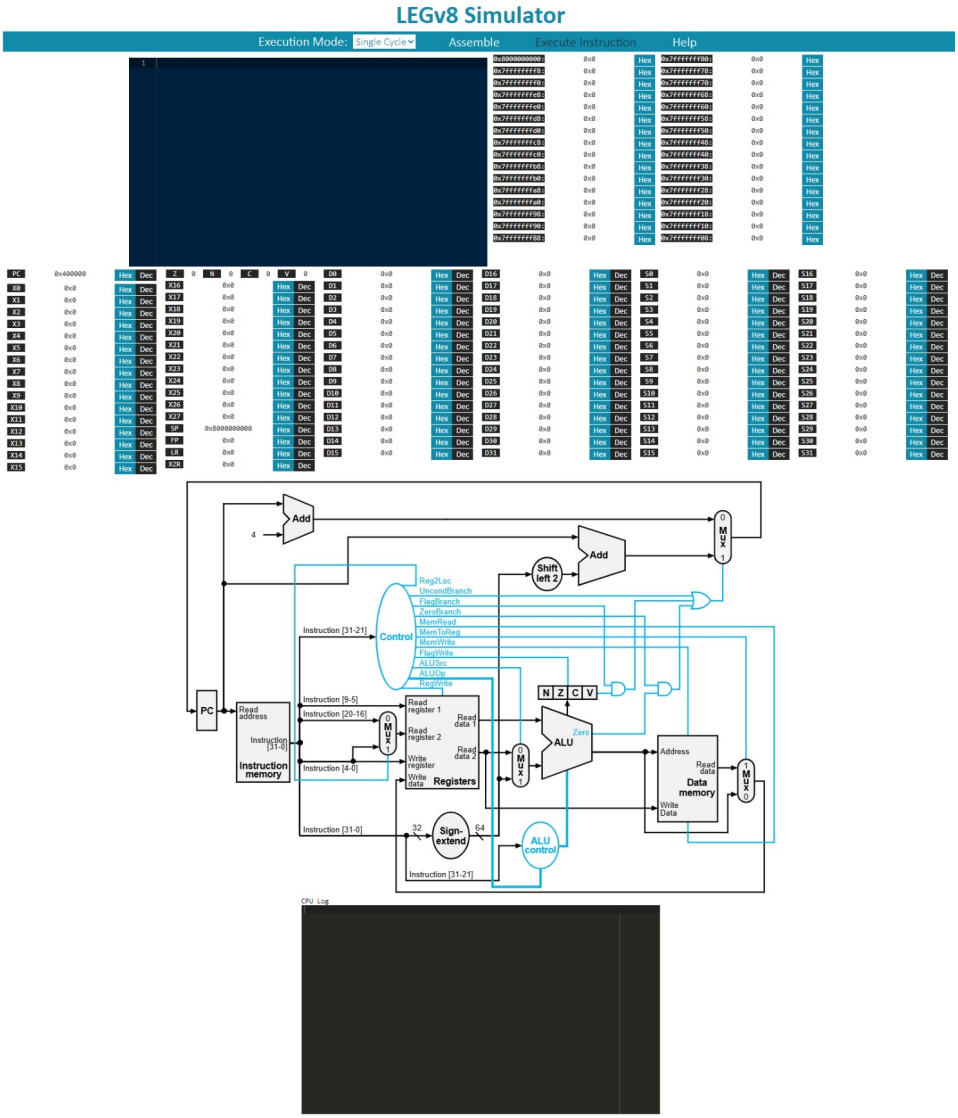


Figure 4.4: The updated Single Cycle visualization

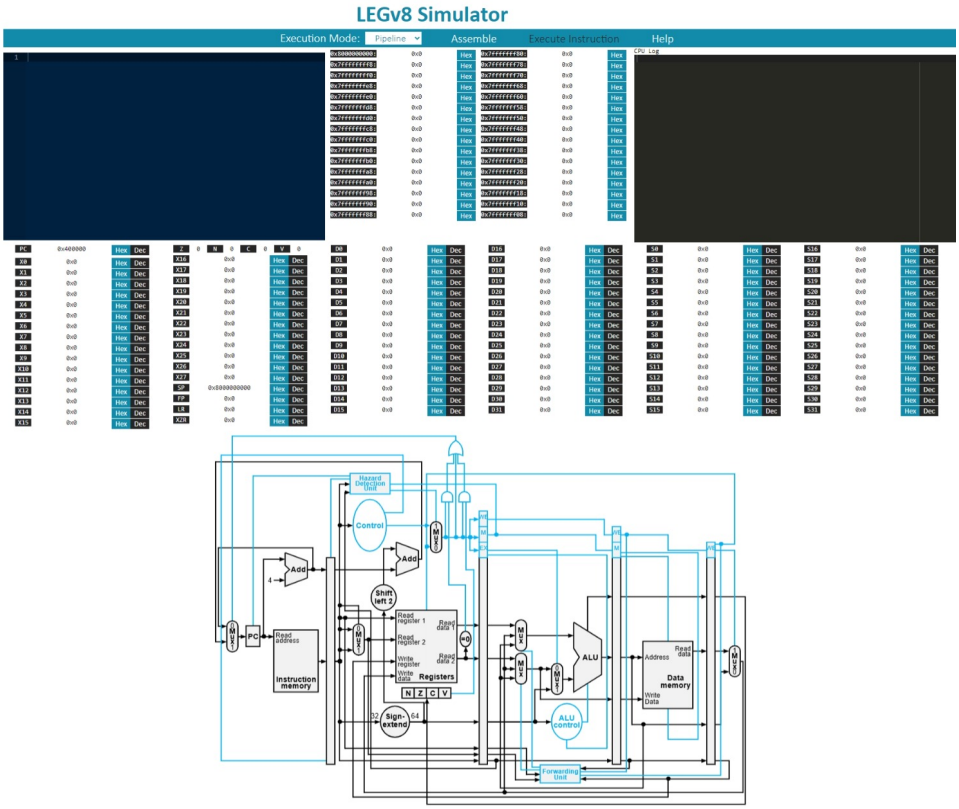


Figure 4.5: The updated pipeline execution mode

Although the pipeline execution was not touched upon in this thesis' work, similar actions have been taken to enhance its look. When compared with Figure 1.8, the updated UI puts, again, the datapath visualization on the bottom, but moves the CPU log to the right of the stack panel. This is because, in the pipelined mode, the CPU log panel provides critical updates to the state of the pipeline, which is very important to keep visible in the main UI. Figure 4.5 showcases this new arrangement.

4.4 Conclusions

In this chapter we have showcased the work done to fix, improve and implement new elements into the original simulator's web UI. Due to the design of the codebase, the process has been straightforward, in spite of GWT's quirky behavior. The end result is a more compact, featureful, informative UI which maintains the same visual language and appearence as the original. This concludes the work of this thesis. The next chapter will provide a summary of our efforts and an outlook on the future of the simulator.

Chapter 5

Concluding remarks

> "Probably I am very naive, but I also think I prefer to remain so, at least for the time being and perhaps for the rest of my life."
>
> *Edsger W. Dijkstra* – EWD923A

This thesis' work has consisted in studying Arm's official LEGv8 simulator, reconstructing and documenting its build process, bringing it up to an acceptable working state by fixing critical bugs in the codebase, and extending it by implementing the entirety of the LEGv8 ISA in single cycle mode and improving its functionalities and UI. As of this date, this is the only publicly available simulator to offer a complete implementation of LEGv8 in single cycle mode and a comprehensive visualization of the stack, of all the registers, and the datapath, as shown in the Patterson's and Hennessy's book [1].

Not every part of the simulator has been fixed or touched upon. Because of the undergraduate nature of this thesis, there wasn't enough time to tackle all the problems still present in the codebase. In this concluding chapter, we will provide a comprehensive list of issues and missing features that can be still worked on without fundamentally changing the nature of the project. In the end, we will discuss the more structural problems that would require a similar amount of effort as this thesis' to fix, and which could be considered suggestions for future developments.

5.1 Current shortcomings and missing features

In no particular order, we present a few observations about some features, both of the original simulator and the ones stemmed from this thesis' work, that could definitely be improved in incremental iterations.

5.1.1 Making full use of Maven

The introduction of Maven into the project has already given it a significant boost in configurability, system agnosticism, and ability to be automated in all of its aspects. Maven is a complicated and state of the art build system for Java, and its configuration is not straight forward. Many steps of of its integration in the project might have been performed incorrectly in this thesis, or without following the best practices. For these reasons, improving upon the configuration of Maven, by making use of plug-ins or by cleaning up the code, would make it even easier to deal with the set-up and build processes.

5.1.2 Updating AceGWT and giving it a new home

The updated version of AceGWT used in the new version of the project doesn't currently reside in the Maven central repositories nor in the official GitHub repository. An effort of informing and helping the original authors to officially integrate this new update, would allow for the removal of a manual, local installation of the library in the user's filesystem. Of course, updating the Ace editor itself to a newer version and creating the new bindings to GWT would be a welcome change, although it would probably require as much work as the original authors'.

5.1.3 Refactoring the codebase

Even though it has been repeated many times in this dissertation that the Java codebase is quite well structured, its code in many places still resembles more of a C-like design. Making too much use of primitive types and not really making use of all the OOP capabilities of Java makes things more complicated than necessary. Since GWT 2.11 started supporting Java 11 features, reorganizing the code to try and make use of these new tools might make future development less cumbersome and the code more extensible.

5.1.4 Creating more documentation

In this dissertation we have tried to include as much information as possible about the simulator, but the ultimate arbiter of truth remains the implemented code. For this reason, understanding the inner workings of

the simulator and providing additional documentation in the form of design documents, comments, and Javadoc, might make life easier for future developers. This thesis could be taken as a starting point for this effort.

5.1.5 Further improvements to the UI

The current state of the UI and UX is quite functional, but, although GWT is an old and outdated framework for creating web applications, it is still possible to make some improvements, such as adding a true responsive design to the web page, especially to make it more usable with mobile devices. Also, a rethinking of the UI elements might allow for a better visual design, such as giving different registers different colors or providing the state of the pipeline with a visual representation, instead of the textual one in the CPU log. Lastly, the stack could be made dynamic by allowing the user to manually browse through the entire memory, either by searching for a specific address or moving around using navigational arrows or scroll bars.

5.1.6 Improving the LEGv8 development experience

Ace is a powerful web editor, and as such it can be thoroughly configured. Things such as LEGv8 syntax highlighting could be added by specifying the LEGv8 grammar in an appropriate format and feeding it to the editor. Another shortcoming is that, between page reloads, the contents of the editor get flushed. Introducing permanence to the written code or the ability to upload LEGv8 assembly via a text file, would make resuming development or showcasing examples much easier. Also, being able to choose to run the code until completion – either instantly or using a custom clock speed – instead of manually going over each instruction, would help in this aspect.

Testing the logic

Maven makes it easy to implement a testing suite for the simulator. Creating an automatic battery of manual or parameterized tests to be ran each time new changes are made, would make it not only simpler to test the current coverage and correctness of the implemented instructions, but would allow for faster development going forward.

5.2 Structural problems and future developments

The points that will be discussed now are more fundamental and deeply embedded into the original design choices of the developers, and would require much more work to be implemented than the ones listed in the last section.

5.2.1 Extending the LEGv8 ISA

As was discussed in Chapter 1, LEGv8 is defined in two ways throughout Patterson's and Hennessy's book [1]. The more detailed one is only reserved to the minimal working example presented in their Chapter 4, while the more abstract one is the one used in the simulator to implement the instructions. Since Arm's simulator provides such a comprehensive representation of the instruction and the datapath, extending the detailed specification of LEGv8 to cover the entire ISA would be of great pedagogical help. This operation would require the developer to make informed decisions in order to be consistent with the textbook's exposition. Something like this was already done, in a much smaller scale, in the work presented in this thesis, when talking about the jump conditions for floating-point operations, where we decided to use ARMv8's convention.

5.2.2 Expanding and fixing the pipeline

The pipeline execution mode was not touched upon in this dissertation. Even when limiting ourselves to integer-based instructions, there are many cases in which it stalls or the execution behaves in unexpected ways. Furthermore, it's completely unequipped to deal with the new floating-point logic. Unfortunately, the `CPU.java` class is not designed as a pipeline, as it's not logically divided into the proper datapath components that make up the processor. Restructuring the simulator, especially with Java's OOP capabilities, to more closely resemble the five stages, would create a much simpler design that would automatically work in pipeline and single cycle mode. This operation would require major changes to many classes inside the project, and it should take inspiration from the implementations done in the hardware simulators showcased in Chapter 1.

5.2.3 Farewell, GWT?

As Appendix A shows, dealing with GWT can be cumbersome. Even though, from the build process point of view, this has been solved in Chapter 2, the entrenched obsolescence of GWT remains. The web is a place of constant innovation, and much better frameworks for creating web applications have been developed since the days of GWT. For this reason, especially when thinking about longer timescales, new solutions should be considered, starting with new libraries for Java. If none are considered suitable, perhaps other languages could be considered, in which case a complete rewrite and redesign would need to happen. It is clear that this last suggestion creates a sort of ship of Theseus situation, and should be the last one to be considered for future developments.

Appendix A

Walkthrough of the original project set-up

In this appendix, we will lay out a more detailed and visual retelling of the steps needed to build the original codebase using the original build methods. After the original build process was reconstructed, a 50-page document, complete with screenshots, was compiled and published inside a pull request to Arm's repository [25], both in PDF and HTML format. This appendix is an attempt to provide a more compact version of this tutorial, in order to embed it into the thesis without requiring access to external documents.

A.1 Downloading the resources

This section will contain all the links to the resources needed before the set-up process can begin. We will focus on downloading as much as possible locally, as these web resources might one day be taken offline. In case that happens, a directory containing all the resources is all that will be needed to reproduce the steps. This tutorial will suppose all the downloads are extracted and put inside a `/sources` folder somewhere in the user's filesystem.

A.1.1 JDK 8

As was mentioned in Chapter 2, JDK 8 is our best option, even though, as the changelogs for version GWT 2.7 [14] show, Java 8 syntax was supported only starting from version 2.8. The choice of JDK vendor is not really relevant, as the features emulated by GWT are the elementary ones of the language. In this case, it was chosen to use Amazon's Corretto 8 JDK [19]. It can be either installed on the operating system or downloaded as a compressed folder, with the latter being recommended for this procedure.

A.1.2 Eclipse IDE

Due to some library changes to newer versions of Eclipse, the last one compatible with the GWT plug-in used in this thesis's work is currently version 2023-09 [26]. Download and extract the `Eclipse IDE for Java Developers` version.

A.1.3 GWT plug-in for Eclipse

The GWT 4.0.0 plug-in repository can be downloaded and extracted locally from the GWT's plug-ins GitHub Releases page [27].

A.1.4 GWT 2.7 and DTD 2.7

From the releases page on GWT's website [28], we can download and extract the 2.7 version. We must also download the corresponding DTD file [29].

A.1.5 AceGWT and DTD 2.8.2

Similarly, we can download GWT 2.8.2's DTD [30]. To download AceGWT we can simply go to the Releases page on their GitHub repository [17] and download the 1.0.0 release.

A.1.6 Arm's LEGv8 simulator

The last component we need is the simulator itself. Like AceGWT, it can be downloaded from the Releases section of its GitHub repository [11] and extracted.

A.2 Setting everything up

Now everything is in place inside the `/sources` folder to begin the set-up process.

A.2.1 Running Eclipse

The Eclipse version that has been downloaded is in executable form, without an installation. We can run it and, when asked, we can choose our preferred folder for the Eclipse workspace.

A.2.2 Pointing to the JDK

Now we can tell Eclipse to use the downloaded JDK by navigating to the top taskbar and selecting `Window` $\longrightarrow$ `Preferences` $\longrightarrow$ `Java` $\longrightarrow$ `Installed JREs`. Now we can press the `Add...` button and tell Eclipse

it's a `Standard VM`. After pressing `Next`, under the `JRE home` label, we select `Directory...` and put the root folder where the JDK we have downloaded in present. Since we have downloaded the JDK as a compressed file, some additional folders might have been created. Start from the outer one and go ony by one until the `JRE system libraries` list gets filled with elements. Now we can `Finish`, select the newly added JDK from the list, and `Apply and Close`.

A.2.3 Installing the GWT plug-in

On the top taskbar, go to `Help` ⟶ `Install New Software...` and press `Add...`. Under the `Name` label, press `Local...` and point to the extracted repository folder and press `Add...`. Select the `GWT Eclipse Plugin` entry and press `Next >`. Continue with the installation, accept the license, and finish. A pop-up window will appear asking to trust the following content. Press `Select All` and `Trust Selected`. The same window will appear again, but this time asking to trust the artifacts. Do the same procedure, and restart Eclipse.

A.2.4 Importing the simulator

We can now add the simulator's sources to the workspace by going to the top taskbar and navigating to `File` ⟶
`Open Projects from File System...`. Under the `Import source` label, press on `Directory...` and select the folder where the simulator has been extracted to. Two elements will appear on the list, select only the one with `Eclipse project` under the `Import as` column and press `Finish`. If Eclipse asks about marketplace solutions or project natures, decline.

A.2.5 Configuring GWT 2.7

The IDE will complain about a lack of resources. First among them, the GWT SDK. Right click on the `LEGv8_Simulator` folder inside the package explorer on the left, and navigate to `Build Path` ⟶
`Configure Build Path...`. Now go to the `Libraries` tab and press
`Add Library...`. Choose GWT as the type of library to be added, and in the next window select `Configure SDKs...`. Press `Add...`, then under the `Installation directory` label press `Browse...` and select the folder where GWT 2.7 was extracted, and press `OK`, `Apply and Close`, and `Finish`. If asked about synchronization, refuse. At this point we will find ourselves back to the Libraries tab, and we can remove the old missing GWT SDK by selecting it and pressing `Remove`.

A.2.6 Adding AceGWT

The last step we need to perform is to include the AceGWT library inside the project in the correct location. While still in the `Build Path` window, go to the Projects tab, select the AceGWT entry and remove it. Go to the Sources tab, select the missing `LEGv8_Simulator/test` entry, and remove it as well. Now press `Apply and Close`. Right click on the `/src` folder inside the package explorer and go to `Import...` $\longrightarrow$ `General` $\longrightarrow$ `File System` and press `Next`. Now, under the `From directory` label, press `Browse...` and select the `/src` folder inside of the AceGWT folder inside of where AceGWT's release was extracted. Select the newly added `src` entry in the left panel of the window, and press `Finish`.

A.2.7 Pointing to the DTDs

It is now time to make use of the DTD files we have downloaded for both GWT 2.7 and 2.8.2. First, open with a text editor the `LEGv8_Simulator.gwt.xml` file inside the `/src/com.arm.legv8simulator` package in the package explorer, and substitute the `http://gwtproject.org/doctype/2.7.0/gwt-module.dtd` string inside the `DOCTYPE` definition with the absolute path of the 2.7 DTD in your filesystem. Now open the `AceGWT.gwt.xml` file in the `edu.ycp.cs.dh.acegwt` package. Since this xml file lacks the `DOCTYPE` definition, simply copy and paste the one from the other file in the same location, and substitute the path with the one for the 2.8.2 DTD.

A.2.8 Compiling the project

To compile the project we can navigate to the second taskbar from the top, and press the newly added GWT button (the red one bearing the GWT logo) and select `GWT Compile Project...`. Under the `Project` label we press `Browse...`, select the simulator's project, and press `OK`. We can now choose our preferred options for the `Log level` and `Output style`. The entry point for GWT will already be selected, as there is only one. If we press `Compile`, the build process will begin, with the resulting output being put inside the `/war` folder that will have appeared in the package explorer.

A.3 New developments

During the writing of the thesis, in the second half of 2024, the GWT community has developed an updated version of the GWT plug-in that fixes the aforementioned compatibility problems with newer versions of Eclipse. This 4.1.0 version is not yet stable and can be found in the same GitHub repository [31] in the Releases section. This update of the plug-in includes

the latest 2.11 version of GWT but removes the inclusion of version 2.7, making some changes to the naming conventions and other smaller details. For these reasons, even though the set-up process of the original codebase could probably now be performed in an even more expedite manner, some changes would need to be made to the tutorial, and things might break in new, subtle ways. Thus, it was decided to keep this version of the tutorial, since it serves an archival purpose and the new build process using Maven should be used anyway.

Appendix B

Guide for the new Maven-based set-up

In this brief appendix, we will provide the tools to build the updated version of AceGWT and some tutorials on how to use the updated build system on a variety of Java IDEs supporting Maven.

B.1 Maintaining AceGWT

As was touched upon in Chapter 2, in order to update AceGWT to the latest GWT version, some changes had to be made. Unfortunately, they are currently not present in the official GitHub repository [17], and thus a custom version of the library needs to be built and used. Some additional documentation about the project is available on the original repository's wiki.

Creating parent pom.xml First of all, to aid the build process, it is useful to create a pom.xml file in the root of the project. This is reported in Listing B.1.

Updating children pom.xml files Due to the new ownership of the GWT project, new artifact names are needed to include GWT 2.11. Listing B.2 shows the changes required for both the AceGWT and AceGWTDemo modules.

Updating module.gwt.xml The `module.gwt.xml` file, in both children modules, needs only an update to the link of the DTD, as shown in Listing B.3.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>edu.ycp.cs.dh</groupId>
  <artifactId>acegwt-2.11.0</artifactId>
  <version>1.3.3</version>
  <packaging>pom</packaging>
  <name>AceGWT-2.11.0</name>
  <description>
    Ace is an embeddable code editor written in JavaScript. It
    matches the features and performance of native editors such
     as Sublime, Vim and TextMate. It can be easily embedded in
     any web page and JavaScript application. Ace is maintained
     as the primary editor for Cloud9 IDE and is the successor
    of the Mozilla Skywriter (Bespin) project. AceGWT is an
    integration of Ace into GWT.
  </description>
  <modules>
    <module>AceGWT</module>
    <!-- <module>AceGWTDemo</module> -->
  </modules>
</project>
```

Listing B.1: The new parent pom.xml

```
...
<parent>
  <groupId>edu.ycp.cs.dh</groupId>
  <artifactId>acegwt-2.11.0</artifactId>
  <version>1.3.3</version>
</parent>
...
<gwt.version>2.11.0</gwt.version>
...
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.gwtproject</groupId>
        <artifactId>gwt</artifactId>
        <version>${gwt.version}</version>
        <type>pom</type>
...
<dependencies>
  <dependency>
    <groupId>org.gwtproject</groupId>
    <artifactId>gwt-user</artifactId>
  </dependency>
  <dependency>
    <groupId>org.gwtproject</groupId>
    <artifactId>gwt-dev</artifactId>
  </dependency>
  <dependency>
    <artifactId>gwt-codeserver</artifactId>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.gwtproject</groupId>
    <artifactId>gwt-servlet</artifactId>
    <scope>runtime</scope>
  </dependency>
...
```

Listing B.2: The changes to the children's pom.xml

```
<!DOCTYPE module PUBLIC "-//Google Inc.//DTD Google Web Toolkit
    2.11.0//EN"
       "http://gwtproject.org/doctype/2.11.0/gwt-module.dtd">
```

Listing B.3: Updated DTD version

Building the library Now the library has to be built, in our case, into a .jar file. This can be achieved by running
`mvn clean gwt:generate-module compile gwt:package-lib` into the `AceGWT` folder inside the project. This will create a folder called `target`. The contents of this folder are the ones that need to be copied in the local Maven repo folder discussed in Chapter 2.

B.2 Set-up tutorials for various Java IDEs

We will present written descriptions of the steps needed to set-up the project in the most popular Java IDEs that support Maven. It is possible to develop the simulator even in environments without Maven support, such as a normal text editor, by editing the code and running Maven via the terminal as showcased in the end of Chapter 2.

B.2.1 IntelliJ IDEA

Choose the `Get from VCS` option and copy paste this repository's git link. The Maven project will be cloned into your workspace and the dependencies will be automatically downloaded and configured. After making your changes you can build and package the simulator by going to the Maven panel on the right, navigate to `Graphical-Micro-Architecture-Simulator` $\longrightarrow$ `LEGv8_Simulator` $\longrightarrow$ `Lifecycle` $\longrightarrow$ `package`. The folder containing the simulator will appear under `LEGv8_Simulator/target/`. IntelliJ Ultimate offers a GWT plugin that warns the programmer when using methods, syntax and classes not implemented by GWT and can generate compile reports. Be warned that in some cases it might show bogous errors (such as missing css directives), but this shouldn't affect the compilation which is done with GWT. If you use IntelliJ without this plugin the errors will disappear but you will not have access to said features.

B.2.2 Eclipse

Choose `Import projects...`, select
`Git/Projects from Git (with smart import)` $\longrightarrow$ `Clone URI` and copy paste this repository's git link. Keep pressing `Next` until it has finished the procedure. It is recommended to go to `Window` $\longrightarrow$ `Preferences` $\longrightarrow$ `XML (Wild Web Developer)` and enable the download of external resources. Unlike IntelliJ, Eclipse doesn't show directly the package action but has to be added manually by right clicking on the `LEGv8_Simulator` folder and going to `Run As` $\longrightarrow$ `Run Configurations...` and then double click on `Maven Build`. This will create a new configuration for you to edit: give it the name package, select `Workspace...` $\longrightarrow$ `LEGv8_Simulator` as the `Base Directory` and write `package` inside the `Goals` text box. Now you can `Apply` and run

it. To run it again just go to the `Run As` menu as before and it should have been added there. The folder containing the simulator will appear under `LEGv8_Simulator/target/` (press F5 in Eclipse to refresh the folders). Eclipse offers a GWT plugin (which has installation problems with Eclipse versions newer than the 2023-09) that makes compilation easier but, unlike the Maven package action, deploys the compiled sources into a `/war` folder, and doesn't automatically copy-paste the web resources needed to launch the web page. That has to be done manually. This plugin uses the older build method and is not recommended.

B.2.3 Apache NetBeans

Clone the repository to a location of your choosing. In NetBeans go to `File` ⟶ `Open Project...` and select the cloned repository folder. To access the `LEGv8_Simulator` files in the IDE go to `Graphical-Micro-Architecture-Simulator` ⟶ `Modules` and double click on `LEGv8_Simulator`. This will open the module in the project browser. In order to build and package the simulator, right click on `LEGv8_Simulator` ⟶ `Run Maven` ⟶ `Goals...` and write package into the `Goals` text field and press `OK`. The folder containing the simulator will appear under `LEGv8_Simulator/target/`.

Bibliography

[1] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design ARM Edition: The Hardware Software Interface*. ISSN. Elsevier Science, 2016.

[2] Arm Limited. Graphical micro-architecture simulator. https://www.arm.com/resources/education/education-kits/legv8.

[3] Arm Holdings plc. https://www.arm.com/.

[4] The Apache Software Foundation. The apache maven project. https://maven.apache.org/.

[5] The GWT Project. https://www.gwtproject.org.

[6] Hemmendinger, David, Freiberger, Paul A., Pottenger, William Morton, Swaine, and Michael R. . "computer". https://www.britannica.com/technology/computer, 2024.

[7] The harvard architecture. https://en.wikipedia.org/wiki/Harvard_architecture.

[8] IEEE Standards Association. The ieee-754-2019 specification. https://standards.ieee.org/ieee/754/6210/.

[9] RISC-V International. History of risc-v. https://riscv.org/about/history/.

[10] GitHub. https://github.com.

[11] Arm Limited. Graphical micro-architecture simulator. https://github.com/arm-university/Graphical-Micro-Architecture-Simulator.

[12] Git. https://git-scm.com/.

[13] Wikipedia. Java virtual machine. https://en.wikipedia.org/wiki/Java_virtual_machine.

[14] The GWT Project. Gwt 2.7 release notes. https://www.gwtproject.org/release-notes.html#Release_Notes_2_7_0.

[15] The Eclipse Foundation. https://eclipseide.org/.

[16] The GWT Project. https://www.gwtproject.org/usingeclipse.html.

[17] daveho. Acegwt. https://github.com/daveho/AceGWT.

[18] Ace. https://ace.c9.io/.

[19] Amazon. https://aws.amazon.com/it/corretto/.

[20] The GWT Project. https://www.gwtproject.org/gettingstarted-v2.html.

[21] The Apace Software Foundation. https://apache.org/.

[22] The GWT Project. Gwt 2.11 release notes. https://www.gwtproject.org/release-notes.html#Release_Notes_2_11_0.

[23] Maarten Bodewes. https://stackoverflow.com/a/55752928.

[24] Jacob Bramley. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/condition-codes-4-floating-point-comparisons-using-vfp.

[25] Simone Deiana. https://github.com/arm-university/Graphical-Micro-Architecture-Simulator/pull/7#issue-2031433977.

[26] Eclipse Foundation. https://www.eclipse.org/downloads/packages/release/2023-09/r.

[27] The GWT Project. https://github.com/gwt-plugins/gwt-eclipse-plugin/releases/tag/v4.0.0.

[28] The GWT Project. https://www.gwtproject.org/versions.html.

[29] The GWT Project. http://gwtproject.org/doctype/2.7.0/gwt-module.dtd.

[30] The GWT Project. http://gwtproject.org/doctype/2.8.2/gwt-module.dtd.

[31] The GWT Project. https://github.com/gwt-plugins/gwt-eclipse-plugin.

I thank my family for tolerating my long journey.
I thank Beatrice G. for believing in me.