UNIVERSITY OF TRIESTE

Department of Engineering and Architecture

Bachelor's degree in Computer Engineering

Restoration and development of a Java-based LEGv8 ISA simulator

July 14, 2024

Graduating student
Simone Deiana

Supervisor
Prof. Alberto Carini

Academic Year 2023/2024

Summary

In this thesis I will be reporting my work done developing upon a Java-based LEGv8 ISA simulator.

In the Introduction I will provide a brief overview of the LEGv8 ISA together with the reasons for choosing this thesis project in the context of the Digital Architectures course.

In Chapter 1 I will provide a short summary of the current landscape of software simulators available online for the LEGv8 ISA. I will end the chapter with a focus on the simulator chosen for this thesis' project, namely the LEGv8 simulator developed and distributed by Arm Holdings plc. I will give an overview of its working state, functionality and structure prior to my development efforts.

In Chapter 2 I will present the work done to decouple the project from the Eclipse IDE and migrate it to a modern build automation system, namely Maven.

In the Chapter 3 I will showcase the bugs that have been fixed and I will introduce all of the functionalities that have been added to the simulator and the structural changes by them entailed.

In Chapter 4 I will talk about the shortcomings of the simulator and the work that can be done to further improve it.

Contents

Introduction

"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

What is an ISA?

A computer is a device which is capable of acquiring data, performing calculations upon it, and making the results available for use at a later date. 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 (the memory) and the mechanisms through which the computer is able to manipulate said data (the processor). Computers are an abstract concept and do not impose a certain technological choice to their 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 when using a mechanical device an operator needs to interact with the physical parts of the system, operating a computer at this level would require the user to manually insert ones and zeros into the right places for it to perform its calculations. It is clear that such an operation would require an intimate knowledge of the physical implementation of the computer, and even minimal changes to its digital circuitry might jeopardize the correctness of any sequences of bits written for an earlier model.

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 circuit designers and the programmers. 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 it's capable of performing are. This forms the layer through which all software must interface with in order to interact with the hardware.

What is the LEGv8 ISA?

The ISA focus of this thesis is the LEGv8 ISA, an ARM-inspired architecture created by David A. Patterson and John L. Hennessy designed to serve as a teaching tool in their book *Computer Organization and Design (ARM Edition)*. As the title suggests, the book is actually about the ARMv8 ISA, whose first iteration was originally released in 1983 by Acorn Computers and which is now developed by Arm Holdings plc. The authors, however, have introduced a few changes and simplifications to the ARMv8 ISA to make it friendlier to students and emphasize certain design concepts. As such, this ISA is used in the sections of the book dedicated to the design of a model processor and its programming, and it's these sections upon which the LEGv8 simulator subject of this thesis is based.

Overview of the LEGv8 ISA

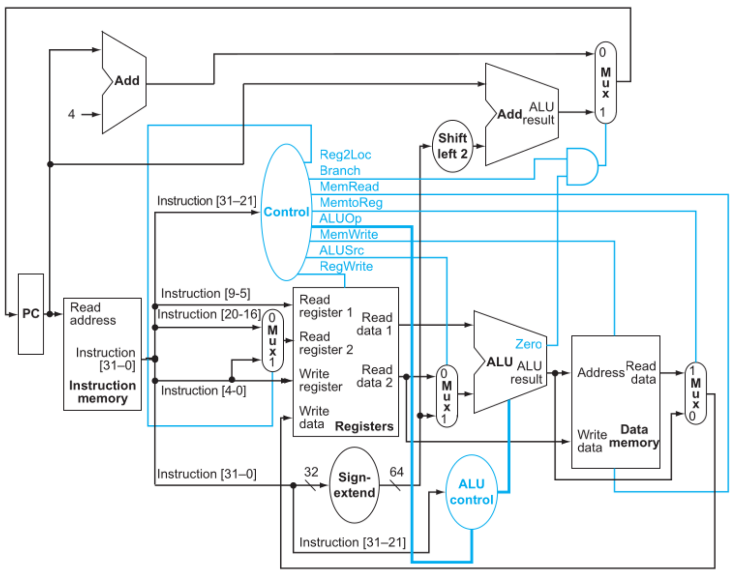


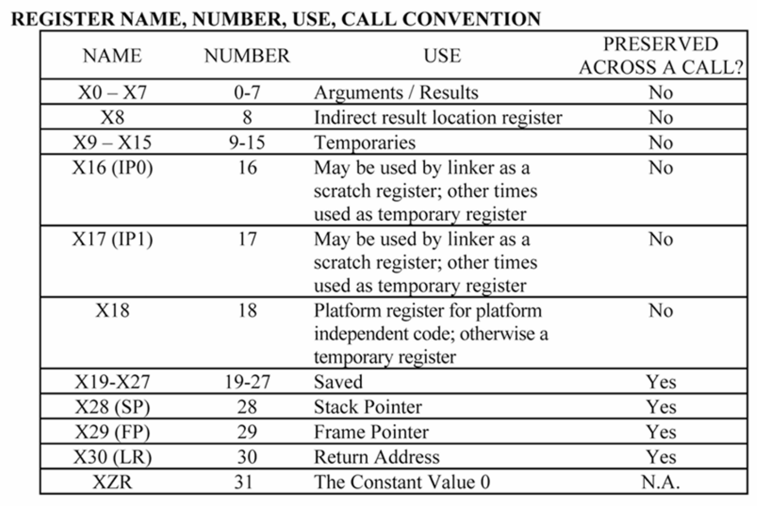
Figure 1: The logical scheme of the LEGv8 architecture

Architecture type

LEGv8 follows the Von Neumann architecture paradigm and thus contemplates the existence of a single memory containing both the instructions and the program data. It is a 64-bit architecture and is specifically designed for pipelined execution.

Registers

LEGv8 defines 32 64-bit `X` registers for storing integer values and 32 64-bit `D` registers for storing double precision floating point values. 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. Registers are also used following a certain convention that is defined by the ISA but not enforced by the processor, and some can be addressed using alternative names for readibility purposes. There are analogous conventions for floating point registers 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 2: Integer registers usage convention

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

Memory

The memory contains both the program code and the data. It is logically divided into a *reserved* segment, a *text* segment containing the program code, a *static data* segment containing the constants defined at compile time, and a *dynamic data* and *stack* segments occupying the same location of memory and respectively growing upwards from the *static data* segment and downwards from the stack pointer. This section of the memory is the one containing the data defined at execution time.

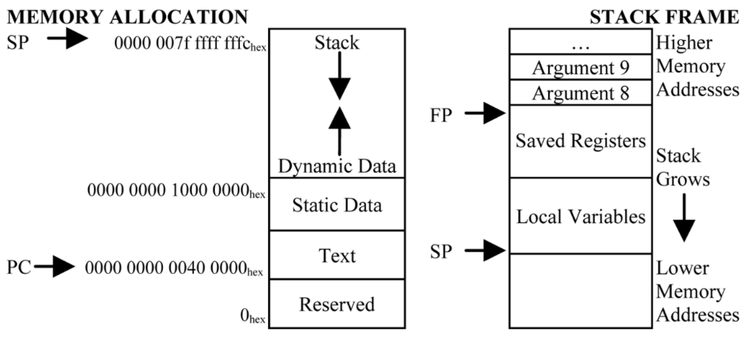


Figure 3: Logical division of the memory

Control unit

The control unit is the component responsible for coordinating the pipeline execution flow and configuring the various components to perform the desired operations in the correct order using the correct parameters.

ALU

The LEGv8 ALU is capable of performing 64-bit integer operations and both single and double precision floating point operations. The operation to perform at any given moment is configured through an ALUop code provided by the control unit.

Pipeline

The LEGv8 pipeline is comprised of 5 stages: *fetch*, *decode*, *execute*, *data access*, and *write back*. As the names suggest, the *fetch* stage is responsible for acquiring instructions from the text segment of the 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 calculation through the ALU, the *data access* stage is responsible for accessing the the memory, and the *write back* stage finally writes the result into the registers. Of course not all instructions make use of all the pipeline stages and this is taken into consideration when optimizing the execution flow.

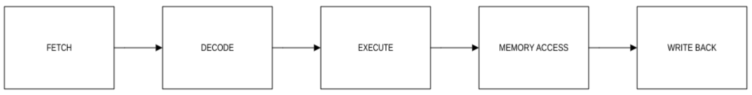
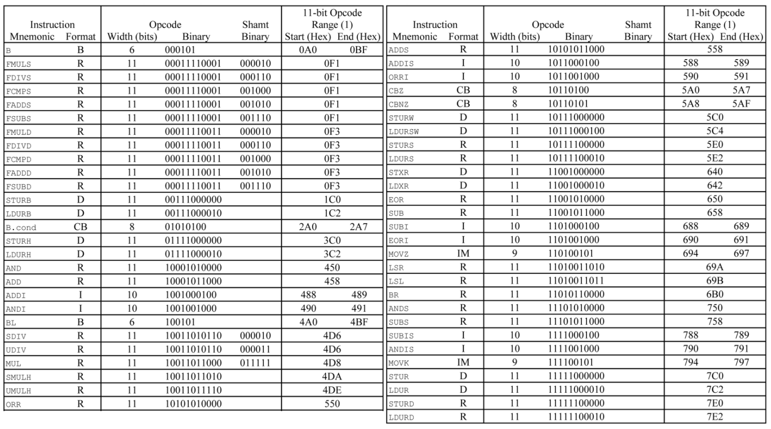


Figure 4: The 5 pipeline stages

Instructions

LEGv8 can be considered a subset of ARMv8, but with a few caveats. Many higher level instructions have been omitted altogether in order to keep the ISA as minimal as possible, and many of the ones that have been kept 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 both with 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 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` or 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.



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 5: The complete LEGv8 ISA

All 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 with wider immediate values.

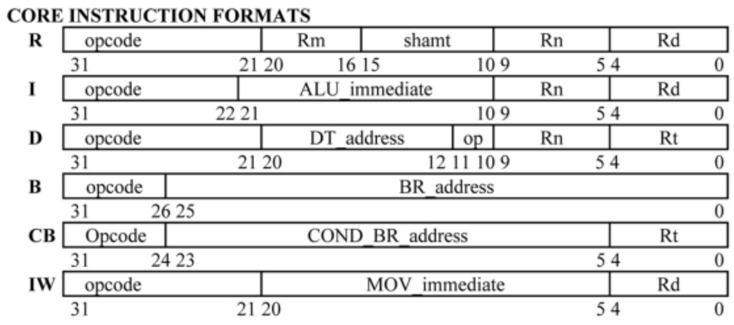


Figure 6: The 5 formats of LEGv8 instructions with their encoding pattern

Motivations for choosing LEGv8

The LEGv8 ISA, being presented and defined in one of the major computer architecture undergraduate textbooks, is taught in many university courses around the world, including the Digital Systems Architecture course held by Prof. Carini at UniTS. In spite of its popularity, no real hardware has been made to run its instruction set natively, and the simulator landscape is almost equally lacking in viable options. This in turn makes it impossible for educators and students alike to show working examples of LEGv8 code, depriving them of teaching and learning opportunities. For these reasons I have chosen to work on an already existing and partially working LEGv8 simulator provided by Arm Holdings plc. to expand upon its functionalities to include a complete simulation of the ISA.

Chapter 1

The LEGv8 simulators landscape

"It used to be the program's purpose to instruct our computers; it became the computer's purpose to execute our programs."

Edsger W. Dijkstra

The current landscape of publicly available LEGv8 simulators can be divided into two categories: simulators that aim to reproduce the logical design presented in the textbook in chapter 4, and the simulators providing a high level simulation of the instruction set as defined in the book. The survey was performed on GitHub using "LEGv8" and "simulator" as keywords and only those in a reasonably working state (as per the author) have been considered.

Software simulators

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

They utilize high level languages such as C++, Python, Swift, TypeScript and Java. Some of them offer a graphical interface, pipelined execution and none of them implement the LEGv8 ISA in its entirety.

Hardware 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

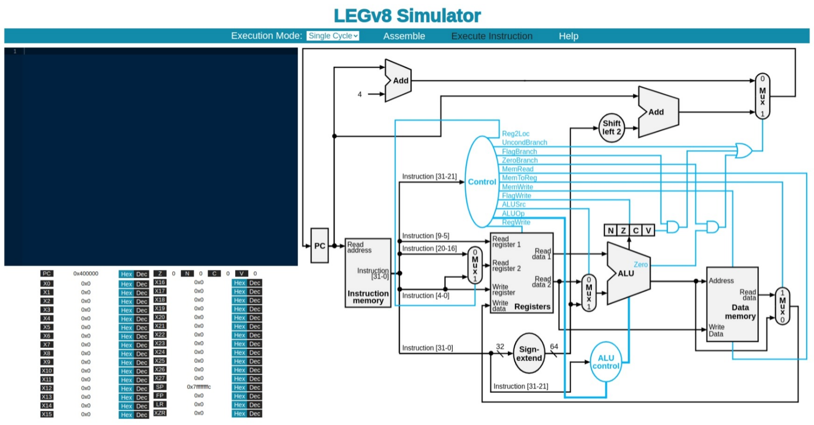
They use mostly Verilog as their hardware description language and implement an incomplete subset of the LEGv8 ISA. Some of them follow closely the design of the textbook while others expand upon it adding more executable instructions. None of them offer a graphical interface nor implement the ISA in its entirety.

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 lack interactivity and comprehensive visual output capabilities.

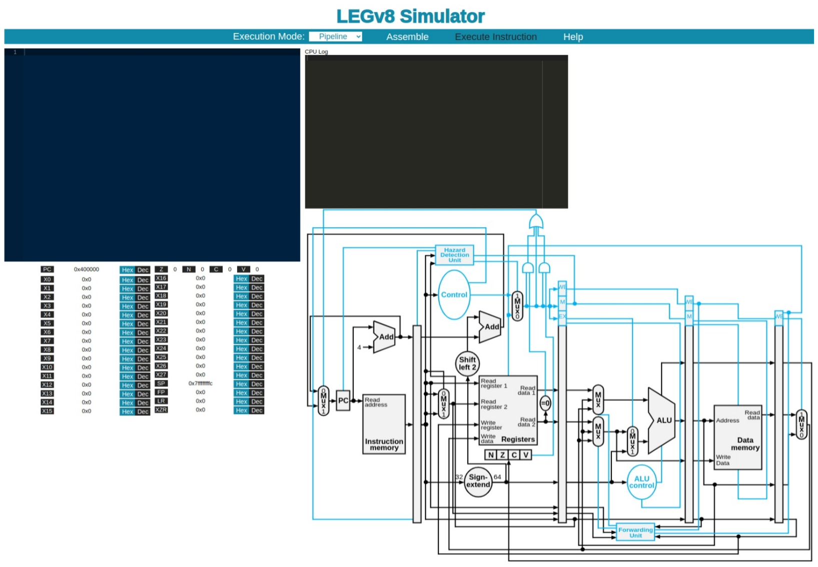
ARM's LEGv8 simulator [1]

This is the simulator officially provided by ARM Education and is the subject of this thesis' work. It is written in Java 8 and uses Google's GWT framework to transpile the code into native JavaScript to allow the simulator to be executed inside a web browser as a normal web application. It provides a comprehensive user interface displaying an interactive text editor (provided by AceGWT) to input LEGv8 code and to display errors, and a visualization of the state of the `X` registers. When selecting the single-cycle execution mode, a visualizaton of the logical scheme of the LEGv8 ISA is presented and for each step of the execution various components change color to indicate the current stage of the pipeline. For the pipelined execution mode, the visualization is slightly modified to include pipeline-specific information such as pipeline registers, the hazard detection unit and the forwarding unit. An additional textual representation of the pipeline is provided to see the stage occupied by each instruction at any given moment.

[1] https://github.com/arm-university/Graphical-Micro-Architecture-Simulator



(a) Single cycle



(b) Pipeline

Figure 1.1: The simulator's main page with the two different execution modes.

Features

This simulator presents many favorable characteristics:

- Written in Java (platform agnostic, extensible).
- Compiled as a web application (platform agnostic and easily deployable).
- Embedded text editor to input code and display errors to.

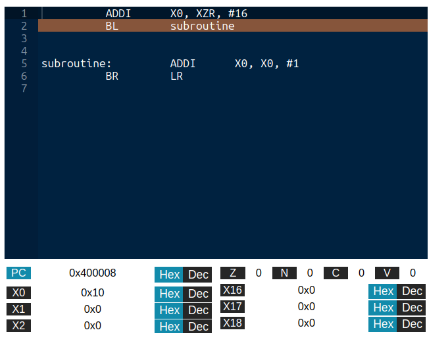
- Clear and rich visualization of the `X` and flag registers and the datapath of the CPU thanks to the web-based interface.
- Almost all of the integer arithmetic is already implemented.
- All types of integer `LOAD` and `STORE` instructions are already implemented, including `STXUR` and `LDXUR`.
- Officially distributed by ARM Education (biggest support and discoverability).

Problems

Unfortunately many problems present themselves when trying to run or develop the simulator:

- Absence of any documentation on how to build the project and design choices behind it.
- Executable version distributed in automatically-generated web page form.
- Pipeline execution is incomplete.
- The mechanism for calling subroutines is broken and results in infinite loops, making it impossible to delegate code to other functions.
- The mechanism for performing comparisons is broken and results in the wrong branches being taken, making it impossible to perform conditional operations and loops.
- The project is heavily dependent on the Eclipse Java IDE with an old GWT plugin to perform the build process.
- The project depends on the outdated and barely supported GWT library to deploy the simulator as a web application. This restricts the developers from using newer Java features or better web frameworks.

I present below a demonstration of the bugs regarding the subroutine calls and number comparisons:

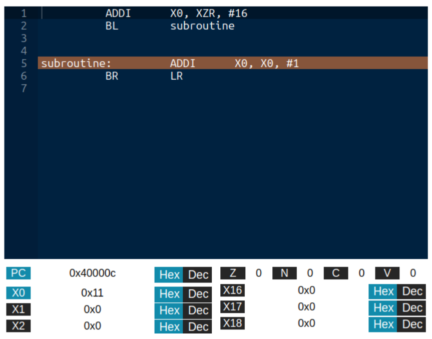


```
            ADDI        X0, XZR, #16
            BL          subroutine


subroutine:             ADDI        X0, X0, #1
            BR          LR
```

```
PC   0x400008     Z 0   N 0   C 0   V 0
X0   0x10         X16   0x0
X1   0x0          X17   0x0
X2   0x0          X18   0x0
```

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

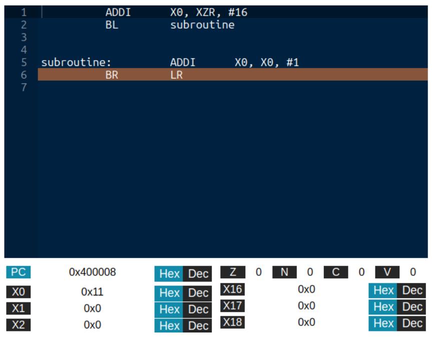


```
            ADDI        X0, XZR, #16
            BL          subroutine


subroutine:             ADDI        X0, X0, #1
            BR          LR
```

```
PC   0x40000c     Z 0   N 0   C 0   V 0
X0   0x11         X16   0x0
X1   0x0          X17   0x0
X2   0x0          X18   0x0
```

(b) Jumps to the subroutine and increments X0

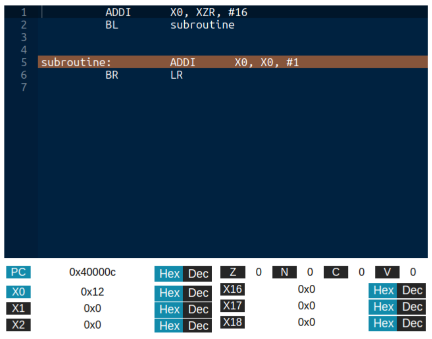


```
            ADDI        X0, XZR, #16
            BL          subroutine


subroutine:             ADDI        X0, X0, #1
            BR          LR
```

```
PC   0x400008     Z 0   N 0   C 0   V 0
X0   0x11         X16   0x0
X1   0x0          X17   0x0
X2   0x0          X18   0x0
```

(c) Reads wrong address from LR



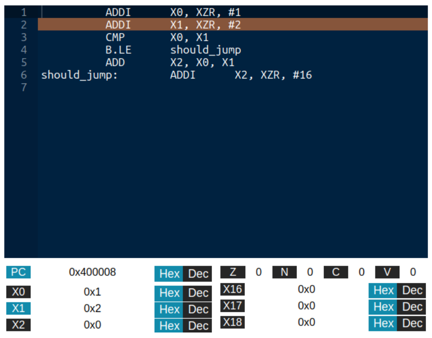
```
            ADDI        X0, XZR, #16
            BL          subroutine


subroutine:             ADDI        X0, X0, #1
            BR          LR
```

```
PC   0x40000c     Z 0   N 0   C 0   V 0
X0   0x12         X16   0x0
X1   0x0          X17   0x0
X2   0x0          X18   0x0
```

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

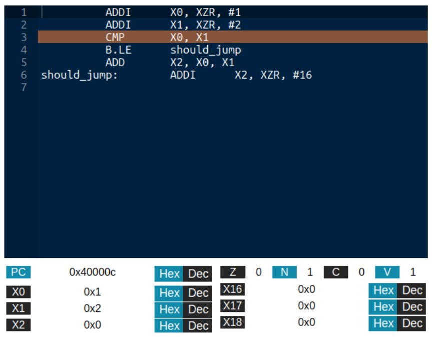
Figure 1.2: Branch returns to the wrong instruction, making it execute the branch in a loop



```
            ADDI        X0, XZR, #1
            ADDI        X1, XZR, #2
            CMP         X0, X1
            B.LE        should_jump
            ADD         X2, X0, X1
should_jump:            ADDI        X2, XZR, #16
```

```
PC   0x400008   Hex Dec   Z 0   N 0   C 0   V 0
X0   0x1        Hex Dec   X16   0x0   Hex Dec
X1   0x2        Hex Dec   X17   0x0   Hex Dec
X2   0x0        Hex Dec   X18   0x0   Hex Dec
```

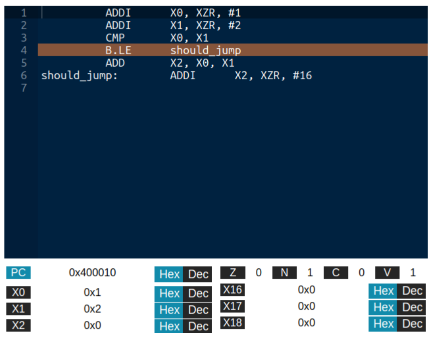
(a) X0 < X1



```
            ADDI        X0, XZR, #1
            ADDI        X1, XZR, #2
            CMP         X0, X1
            B.LE        should_jump
            ADD         X2, X0, X1
should_jump:            ADDI        X2, XZR, #16
```

```
PC   0x40000c   Hex Dec   Z 0   N 1   C 0   V 1
X0   0x1        Hex Dec   X16   0x0   Hex Dec
X1   0x2        Hex Dec   X17   0x0   Hex Dec
X2   0x0        Hex Dec   X18   0x0   Hex Dec
```

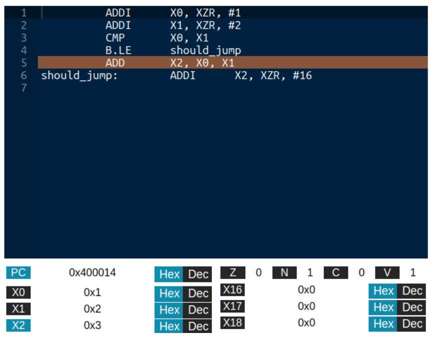
(b) Comparison sets the flags incorrectly



```
            ADDI        X0, XZR, #1
            ADDI        X1, XZR, #2
            CMP         X0, X1
            B.LE        should_jump
            ADD         X2, X0, X1
should_jump:            ADDI        X2, XZR, #16
```

```
PC   0x400010   Hex Dec   Z 0   N 1   C 0   V 1
X0   0x1        Hex Dec   X16   0x0   Hex Dec
X1   0x2        Hex Dec   X17   0x0   Hex Dec
X2   0x0        Hex Dec   X18   0x0   Hex Dec
```

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



```
            ADDI        X0, XZR, #1
            ADDI        X1, XZR, #2
            CMP         X0, X1
            B.LE        should_jump
            ADD         X2, X0, X1
should_jump:            ADDI        X2, XZR, #16
```

```
PC   0x400014   Hex Dec   Z 0   N 1   C 0   V 1
X0   0x1        Hex Dec   X16   0x0   Hex Dec
X1   0x2        Hex Dec   X17   0x0   Hex Dec
X2   0x3        Hex Dec   X18   0x0   Hex Dec
```

(d) Wrong instruction executed

Figure 1.3: Comparisons do not set the correct flags and thus fail

Motivations

For these reasons, this simulator was chosen as the subject of my thesis:

– Maximize the impact of my work by fixing and improving the most popular simulator available

– Provide the first complete implementation of the LEGv8 instruction set

– Allow the Digital Systems Architecture course at UniTS and other courses in general to have a working LEGv8 simulator for more effective teaching

– Opportunity to work on a real Java code base

The simulator's inner workings

Chapter 2

Building and modernizing the code base

"Much of the excitement we get out of our work is that we don't really know what we are doing"

Edsger W. Dijkstra

Getting the project to compile

As was pointed out in Chapter 1, the project's documentation lacks any kind of indications of how to successfully build it [1]. The presence of a `.project` file indicates that at some point it was developed using the Eclipse IDE [2]. Furthermore the existence of a `.gwt.xml` file [3] makes it clear that the GWT framework [4] is being used to generate the web application. Its contents tell us that the JDK version to use is Java 8, since that's the latest GWT v2.7 (partially) supports [5]. By reading the file we also discover that the project uses a module called AceGWT [6], a port of an older version of the ACE editor [7] that implement GWT bindings and allows the web application to embed a

[1]https://raw.githubusercontent.com/arm-university/Graphical-Micro-Architecture-Simulator/main/README.md

[2]https://www.eclipse.org/

[3]https://raw.githubusercontent.com/arm-university/Graphical-Micro-Architecture-Simulator/main/LEGv8_Simulator/src/com/arm/legv8simulator/LEGv8_Simulator.gwt.xml

[4]https://www.gwtproject.org/

[5]As we can see, until v2.8, GWT didn't even support basic Java constructs such as Map, Arrays, BigInteger, Stream, etc. https://www.gwtproject.org/release-notes.html#Release_Notes_2_8_0

[6]https://github.com/daveho/AceGWT

[7]https://ace.c9.io/

text editor. The project uses the 1.0.0 release of AceGWT [8] which predates its integration with Maven [9].
By piecing together these clues I cloned the repository, downloaded both the GWT v2.7 and AceGWT's 1.0.0 releases and imported the project into Eclipse. GWT's website also suggests using GWT's Eclipse plugin [10], so that was installed as well.
The project expected to have access to some files and libraries in certain places, so by configuring correctly the build path of the project I was able to get it to finally compile [11].
This set-up allowed me to do most of the work presented in this thesis, but presented a few glaring problems when thinking about the future maintainability of the software:

- Changes to the Eclipse IDE introduced after version 2023-09 have made it impossible to install the GWT plugin. This means that any future development would need to happen on an old version of the IDE unless an official fix was provided.
- Both AceGWT and GWT have switched to Maven in their latest releases, making the importing, dependency management, and building of the code base automatic.
- The project uses an old version of GWT and could make use of the new features implemented in the newer releases.
- Downloading the dependencies and manually setting up the project from a non-working state each time is a tedious and finnicky process that cannot be depended upon in case something changes to the IDE.
- The project is forever bounded to the Eclipse IDE, meaning it cannot be automatically built headlessly through a script or developed using more modern and featureful IDEs.
- The building process is not well configured. For example, it's not possible to change the directory where the web application is compiled and all the web resources need ot be already present in the output folder otherwise the web application cannot be launched.

Thus, my aim was to make the project as agnostic as possible and turn the set-up into a 1-click process to make it viable for future developers to get

[8] https://github.com/daveho/AceGWT/releases/tag/1.0.0
[9] Maven is a build automation system that allows to automatically fetch and import libraries to your Java project and compile and deploy it: https://maven.apache.org
[10] https://www.gwtproject.org/usingeclipse.html
[11] The entire process is available as a PDF file or static web page: https://github.com/arm-university/Graphical-Micro-Architecture-Simulator/pull/7

started collaborating without any roadblocks. This has been mostly achieved by porting the project to Maven, and in the process making a few updates to the environment.

Modernizing the project and porting to Maven

This part of my work progressed through much trial and error. After reading through the Maven and GWT documentation and creating empty GWT projects using their newest tools, I figured out how to configure Maven's `pom.xml` and GWT's `.gwt.xml` files to correctly import the latest version of GWT and make it recognize the project as a GWT web application. As part of the modernization, I created a local Maven repository in which I built a custom version of AceGWT using the latest version of GWT. Lastly, even though GWT still doesn't support the entirety of Java 8, it is possible to use JDK 21 to build the project and utilize some newer Java features in the code.
After all of this was done, downloading, configuring, and building the project was reduced to running `git clone` and `mvn package` inside the project's directory when using the command line. This also made it possible to import and develop the project on any Java IDE that supports Maven by doing the same steps using the IDE's graphical workflow.

Chapter 3

Bug fixing and new features

"If debugging is the process of removing software bugs, then programming must be the process of putting them in."

Edsger W. Dijkstra

Getting the project to a working state

The flag setting bug In LEGv8, CMP and CMPI are pseudoinstructions, meaning that under the hood they actually make use of the SUBS and SUBIS instructions respectively to set the compare flags. The fact that the former instructions failed, pointed at a problem in the latter ones, which was proven to be correct. The simulator first implements the function responsible for setting the flags of the addition operations and when setting the flags for the subtraction operations it simply calls the same function with the same arguments.

```java
private void ADDSetFlags(long result, long op1, long op2) {
  setNflag(result < 0);
  setZflag(result == 0);
  setCflag(result, op1, op2);
  setVflag(result, op1, op2);
}
```

Listing 3.1: The adddition flag-setting code

```java
private void SUBSetFlags(long result, long op1, long op2) {
  ADDSetFlags(result, op1, op2);
}
```

Listing 3.2: The buggy subtraction flag-setting code

As we can see, this presents a problem since subtraction and addition set their flags in a different way. The fix was simply to call the same function but with the 2-complement of the second operand.

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

Listing 3.3: The fixed subtraction flag-setting code

The branch return bug For this bug, inspecting the LR register showed that the BL instruction was not writing the register with the address of the current instruction, but with the subroutine's one instead. This created an infinite loop since, when the subroutine returned to the LR, the program would jump back to the beginning of the subroutine all over again.

```
private void BL(int branchIndex) {
  instructionIndex = branchIndex;
  XRegisterFile[LR].writeDoubleWord(instructionIndex *
 INSTRUCTION_SIZE + Memory.TEXT_SEGMENT_OFFSET);
  ...
}
```

Listing 3.4: The buggy address writing

As we can see, the instructionIndex is updated too soon and thus the LR register gets written with the address of the branch.

```
private void BL(int branchIndex) {
  XRegisterFile[LR].writeDoubleWord(instructionIndex *
 INSTRUCTION_SIZE + Memory.TEXT_SEGMENT_OFFSET);
  instructionIndex = branchIndex;
  ...
}
```

Listing 3.5: The fixed address writing

The datapath visualization bug An issue that was raised on GitHub [1] complained about erroneous values of the MemWrite and MemRead signals from the control unit. This was a problem in the configuration.

```
...
ctx.fillText(ControlUnitConfiguration.toString(c.memRead),
  DATA_MEM_COORDS[0]+DATA_MEM_DIMENSIONS[0]/2-t.getWidth()-1,
   DATA_MEM_COORDS[1]-3);
ctx.fillText(ControlUnitConfiguration.toString(c.memToReg),
  MUX_READ_DATA_MEM_COORDS[0]+MUX_READ_DATA_MEM_DIMENSIONS
  [0]/2-t.getWidth()-1, MUX_READ_DATA_MEM_COORDS[1]-3);
```

[1]https://github.com/arm-university/Graphical-Micro-Architecture-Simulator/issues/8

```
ctx.fillText(ControlUnitConfiguration.toString(c.memRead),
  DATA_MEM_COORDS[0]+DATA_MEM_DIMENSIONS[0]/2-t.getWidth()-1,
   DATA_MEM_COORDS[1]+DATA_MEM_DIMENSIONS[1]+10);
...
```

Listing 3.6: Buggy SingleCycleVis.java

```
...
ctx.fillText(ControlUnitConfiguration.toString(c.memWrite),
  DATA_MEM_COORDS[0]+DATA_MEM_DIMENSIONS[0]/2-t.getWidth()-1,
   DATA_MEM_COORDS[1]-3);
ctx.fillText(ControlUnitConfiguration.toString(c.memToReg),
  MUX_READ_DATA_MEM_COORDS[0]+MUX_READ_DATA_MEM_DIMENSIONS
  [0]/2-t.getWidth()-1, MUX_READ_DATA_MEM_COORDS[1]-3);
ctx.fillText(ControlUnitConfiguration.toString(c.memRead),
  DATA_MEM_COORDS[0]+DATA_MEM_DIMENSIONS[0]/2-t.getWidth()-1,
   DATA_MEM_COORDS[1]+DATA_MEM_DIMENSIONS[1]+10);
...
```

Listing 3.7: Fixed SingleCycleVis.java

```
...
RM_LOAD(null, false, false, false, false, true, true, false,
  true, 0, true),
...
```

Listing 3.8: BuggyControlUnitConfiguration.java

```
...
RM_LOAD(null, false, false, false, true, true, false, false,
  true, 0, true),
...
```

Listing 3.9: Fixed ControlUnitConfiguration.java

Adding new features

Refactoring the memory The `ByteBuffer.java` and `Memory.java` classes have mostly been left untouched, although their methods and variables presented some Java-centric names and have thus been replaced with more apt LEGv8 names such as `getDoubleWord` instead of `getLong`. The base address of the stack, defined in the textbook as `0x7ffffffffc`, was not quadword-aligned, leading to a design contradiction. I chose to change it to the compatible address `0x8000000000`.

Completing the integer arithmetic The integer-related instructions missing from the simulator were: `MUL`, `SMULH`, `UMULH`, `SDIV`, and `UDIV`. In order to implement these new instructions a few changes to the code had to be made. First of all they had been added to `Mnemonic.java`.

```
...
...
```

```
...
MUL("MUL", "mul", TokenType.XMNEMONIC_RRR, "10011011000", "
  0010"),
SMULH("SMULH", "smulh", TokenType.XMNEMONIC_RRR, "10011011010
  ", "0010"),
UMULH("UMULH", "umulh", TokenType.XMNEMONIC_RRR, "10011011110
  ", "0010"),
SDIV("SDIV", "sdiv", TokenType.XMNEMONIC_RRR, "10011010110",
  "0010")
UDIV("UDIV", "udiv", TokenType.XMNEMONIC_RRR, "10011010110",
  "0010"),
...
```

Listing 3.10: Added mnemonics

Then to Decoder.java

```
...
case MUL :
return new Instruction(mnemonic, decodeRRRArgs(args),
  lineNumber, ControlUnitConfiguration.RRR);
case UMULH :
return new Instruction(mnemonic, decodeRRRArgs(args),
  lineNumber, ControlUnitConfiguration.RRR);
case SMULH :
return new Instruction(mnemonic, decodeRRRArgs(args),
  lineNumber, ControlUnitConfiguration.RRR);
case UDIV :
return new Instruction(mnemonic, decodeRRRArgs(args),
  lineNumber, ControlUnitConfiguration.RRR);
case SDIV :
return new Instruction(mnemonic, decodeRRRArgs(args),
  lineNumber, ControlUnitConfiguration.RRR);
...
```

Listing 3.11: Added instructions to the decoder

Then they had to be added to `TokenType.java` to use with the parser

```
...
 MNEMONIC_RRR("ADDS?[ \t]+|SUBS?[ \t]+|ANDS?[ \t]+|MUL[ \t
  ]+|SMULH[ \t]+|UMULH[ \t]+|SDIV[ \t]+|UDIV[ \t]+|ORR[ \t]+|
  EOR[ \t]+|adds?[ \t]+|subs?[ \t]+|ands?[ \t]+|mul[ \t]+|
  smulh[ \t]+|umulh[ \t]+|sdiv[ \t]+|udiv[ \t]+|orr[ \t]+|eor
  [ \t]+", 15, "MNEMONIC"),
...
```

Listing 3.12: Addition to the parser

And finally they had to be implemented inside `CPU.java` to execute the operations

```java
...
private void MUL(int destReg, int op1Reg, int op2Reg) {

    XRegisterFile[destReg].writeDoubleWord(XRegisterFile[
  op1Reg].readDoubleWord() * XRegisterFile[op2Reg].
  readDoubleWord());
   }
}
private void SDIV(int destReg, int op1Reg, int op2Reg) {
    XRegisterFile[destReg].writeDoubleWord(XRegisterFile[
  op1Reg].readDoubleWord() / XRegisterFile[op2Reg].
  readDoubleWord());
}

private void UDIV(int destReg, int op1Reg, int op2Reg) {
    BigInteger dividend = BigInteger.valueOf(XRegisterFile[
  op1Reg].readDoubleWord()).and(UNSIGNED_LONG_MASK);
    BigInteger divisor = BigInteger.valueOf(XRegisterFile[
  op2Reg].readDoubleWord()).and(UNSIGNED_LONG_MASK);
    BigInteger quotient = dividend.divide(divisor);
    XRegisterFile[destReg].writeDoubleWord(quotient.longValue
  ());
}

private void SMULH(int destReg, int op1Reg, int op2Reg) {
    BigInteger fullResult = BigInteger.valueOf(XRegisterFile[
  op1Reg].readDoubleWord()).multiply(BigInteger.valueOf(
  XRegisterFile[op2Reg].readDoubleWord()));
    BigInteger shiftedResult = fullResult.bitLength() > 64 ?
  fullResult.shiftRight(64) : BigInteger.valueOf(0);
    XRegisterFile[destReg].writeDoubleWord(shiftedResult.
  longValue());;

}

private void UMULH(int destReg, int op1Reg, int op2Reg) {
    BigInteger fullResult = BigInteger.valueOf(XRegisterFile[
  op1Reg].readDoubleWord()).and(UNSIGNED_LONG_MASK).multiply(
  BigInteger.valueOf(XRegisterFile[op2Reg].readDoubleWord()).
  and(UNSIGNED_LONG_MASK));
    BigInteger shiftedResult = fullResult.bitLength() > 64 ?
  fullResult.shiftRight(64) : BigInteger.valueOf(0);
    XRegisterFile[destReg].writeDoubleWord(shiftedResult.
  longValue());;
}
...
```

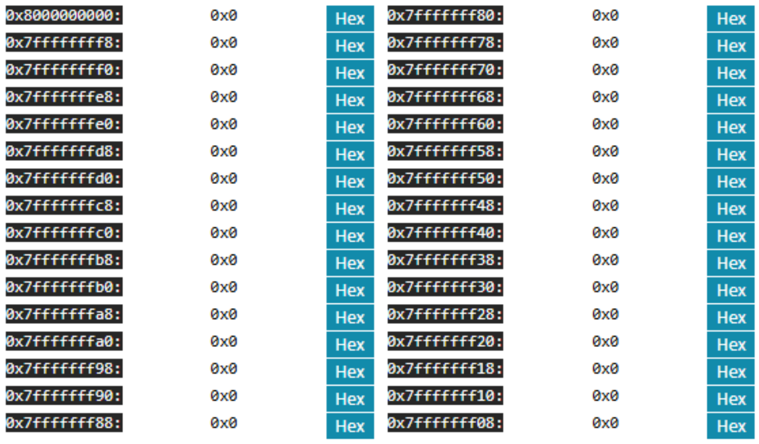
Of particular interest are the `UDIV`, `UMULH` and `SMULH` instructions as they make use of the `BigInteger` class.

– Java does not support unsigned integers. This means that `UDIV` needs to artificially represent them with 65 bit signed numbers through the use of a bit mask. This way it's able to perform the division and return a native 64 bit signed integer.

– The `*MULH` instructions perform a 128-bit multiplication between two 64-bit integers and retain the higher 64 bits. To perform such a calculation Java needs to go beyond its primitive types and make use of `BigInteger`.

Of course this could have been done in more primitive ways through the use of arrays, but GWT supported the `BigInteger` type and allowed to solve the problem quickly.

Visualizing the stack After finishing implementing the integer arithmetic it was time to make the stack visible inside the web interface. This was done by reutilizing the same structure of `RegisterPanel.java`. As it's evident from `StackPanel.java`, the stack visualization includes the address of the double word stored and only shows hexadecimal values, unlike with `X` registers where you can convert between hex and decimal signed representation.



```
0x8000000000:    0x0    Hex    0x7fffffff80:    0x0    Hex
0x7ffffffff8:    0x0    Hex    0x7fffffff78:    0x0    Hex
0x7ffffffff0:    0x0    Hex    0x7fffffff70:    0x0    Hex
0x7fffffffe8:    0x0    Hex    0x7fffffff68:    0x0    Hex
0x7fffffffe0:    0x0    Hex    0x7fffffff60:    0x0    Hex
0x7fffffffd8:    0x0    Hex    0x7fffffff58:    0x0    Hex
0x7fffffffd0:    0x0    Hex    0x7fffffff50:    0x0    Hex
0x7fffffffc8:    0x0    Hex    0x7fffffff48:    0x0    Hex
0x7fffffffc0:    0x0    Hex    0x7fffffff40:    0x0    Hex
0x7fffffffb8:    0x0    Hex    0x7fffffff38:    0x0    Hex
0x7fffffffb0:    0x0    Hex    0x7fffffff30:    0x0    Hex
0x7fffffffa8:    0x0    Hex    0x7fffffff28:    0x0    Hex
0x7fffffffa0:    0x0    Hex    0x7fffffff20:    0x0    Hex
0x7fffffff98:    0x0    Hex    0x7fffffff18:    0x0    Hex
0x7fffffff90:    0x0    Hex    0x7fffffff10:    0x0    Hex
0x7fffffff88:    0x0    Hex    0x7fffffff08:    0x0    Hex
```

Figure 3.1: The newly introduced stack visualization

Implementing floating point arithmetic Since the simulator was built with integer arithmetic in mind, all the additions made until now did not require any change to the underlying logic and structure of the code base. Introducing floating point operations on the other hand, required updating other parts of the logic that were previously left untouched.

Adding floating point registers The introduction of two new types of registers required the creation of a new `Register.java` type. This new class includes the *type* of the register, its *content* (i.e. the bits stored inside) and the methods to write and read from it. I have chosen to memorize the bits inside of the registers as a `long` value regardless of the register type.

This is because Java offers utility methods to convert `float`s into `int` bits, `double`s into `long` bits, and vice versa, I simply applied these conversions when reading and writing to the registers. This way, the `long` and `int` values used throughout the simulator become just sequences of bits without an implicit interpretation. Although this choice might not be obvious when reading the code since no binary type was introduced, one of its advantages has been that the `Memory.java` class has not required any changes to keep working correctly.

Adding floating point operations Having added the new registers to `CPU.java`, it was now possible to implement all the floating point operations. Just like before, different classes had to be modified, but right now I will focus on the operations done by the CPU.

```
 ...
 DRegisterFile[destReg].writeWord(Float.floatToIntBits(
 Float.intBitsToFloat(DRegisterFile[op1Reg].readWord()) +
 Float.intBitsToFloat(DRegisterFile[op2Reg].readWord())
 ));
 ...

```

Listing 3.13: FADDS

```
 ...
 DRegisterFile[destReg].writeDoubleWord(Double.
doubleToLongBits(
 Double.longBitsToDouble(DRegisterFile[op1Reg].
readDoubleWord()) +
 Double.longBitsToDouble(DRegisterFile[op2Reg].
readDoubleWord())
 ));
 ...

```

Listing 3.14: FADDD

```
 ...
 float op1f = Float.intBitsToFloat(DRegisterFile[op1Reg].
readWord());
 float op2f = Float.intBitsToFloat(DRegisterFile[op2Reg].
readWord());
 FCMPSetFlags(Float.compare(op1f, op2f), Float.isNaN(op1f)
|| Float.isNaN(op1f));
 ...

```

Listing 3.15: FCMPS

```
 ...
```

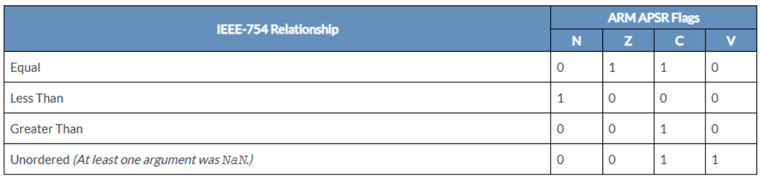
```
    double op1d = Double.longBitsToDouble(DRegisterFile[op1Reg
   ].readDoubleWord());
    double op2d = Double.longBitsToDouble(DRegisterFile[op2Reg
   ].readDoubleWord());
    FCMPSetFlags(Double.compare(op1d, op2d), Double.isNaN(op1d)
     || Double.isNaN(op2d));
    ...
```

Listing 3.16: FCMPD

```
    private void FCMPSetFlags(int comparisonResult, boolean
   isNaN) {
      setNflag(comparisonResult < 0 && !isNaN);
      setZflag(comparisonResult == 0 && !isNaN);
      setCflag(comparisonResult >= 0 || isNaN);
      setVflag(isNaN);
    }
```

Listing 3.17: Function for setting floating point comparison flags

It's important to note that the LEGv8 specification does not say how flags should be set in case of floating point comparison. As a guideline my supervisor Prof. Carini decided to use ARM's official documentation regarding IEEE 754 [2]. Since Java offers the same comparison utility methods and criteria for single and double precision floating point numbers, a single method needed to be written.



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 3.2: The flag setting convention for floating point comparisons.

```
    ...
    DRegisterFile[destReg].writeWord(Float.floatToIntBits(
    Float.intBitsToFloat(DRegisterFile[op1Reg].readWord()) /
    Float.intBitsToFloat(DRegisterFile[op2Reg].readWord())
    ));
    ...
```

Listing 3.18: FDIVS

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

```
    ...
    DRegisterFile[destReg].writeDoubleWord(Double.
   doubleToLongBits(
    Double.longBitsToDouble(DRegisterFile[op1Reg].
   readDoubleWord()) /
    Double.longBitsToDouble(DRegisterFile[op2Reg].
   readDoubleWord())
    ));
    ...
```

Listing 3.19: FDIVD

```
    ...
    DRegisterFile[destReg].writeWord(Float.floatToIntBits(
    Float.intBitsToFloat(DRegisterFile[op1Reg].readWord()) *
    Float.intBitsToFloat(DRegisterFile[op2Reg].readWord())
    ));
    ...
```

Listing 3.20: FMULS

```
    ...
    DRegisterFile[destReg].writeDoubleWord(Double.
   doubleToLongBits(
    Double.longBitsToDouble(DRegisterFile[op1Reg].
   readDoubleWord()) *
    Double.longBitsToDouble(DRegisterFile[op2Reg].
   readDoubleWord())
    ));
    ...
```

Listing 3.21: FMULD

```
    ...
    DRegisterFile[destReg].writeWord(Float.floatToIntBits(
    Float.intBitsToFloat(DRegisterFile[op1Reg].readWord()) -
    Float.intBitsToFloat(DRegisterFile[op2Reg].readWord())
    ));
    ...
```

Listing 3.22: FSUBS

```
    ...
    DRegisterFile[destReg].writeDoubleWord(Double.
   doubleToLongBits(
    Double.longBitsToDouble(DRegisterFile[op1Reg].
   readDoubleWord()) -
    Double.longBitsToDouble(DRegisterFile[op2Reg].
   readDoubleWord())
```

```
    ));
    ...

```

Listing 3.23: FSUBD

As we can see from the last four instructions, the floating point registers, just like their integer counterpart, write and read values from the memory in the exact same way without any change to the logic. In fact, these methods could be refactored and grouped together to reduce code duplication.

```
    ...
    DRegisterFile[destReg].writeWord((int) memory.
   loadDoubleword(XRegisterFile[baseAddressReg].readDoubleWord
   ()+offset));
    ...

```

Listing 3.24: LDURS

```
    ...
    DRegisterFile[destReg].writeDoubleWord(memory.
   loadDoubleword(XRegisterFile[baseAddressReg].readDoubleWord
   ()+offset));
    ...

```

Listing 3.25: LDURD

```
    ...
    memory.storeWord(XRegisterFile[baseAddressReg].
   readDoubleWord()+offset, DRegisterFile[valReg].
   readDoubleWord());
    ...

```

Listing 3.26: STURS

```
    ...
    memory.storeDoubleword(XRegisterFile[baseAddressReg].
   readDoubleWord()+offset, DRegisterFile[valReg].
   readDoubleWord());
    ...

```

Listing 3.27: STURD

Refactoring the parser LEGv8 does not allow instructions to operate on different types of registers. For this reason code like `ADD X0, S4, D20` or `FADDS X0, X1, X2` is not valid. This in turn means that the assembler (in this case, the parser) needs to be made aware that certain instructions work

only with certain types of registers.
The parser works by implementing a finite state machine through the use of enumerators. It takes each line and scans it step by step to check if its syntax is correct. The integer parser has the following diagram.

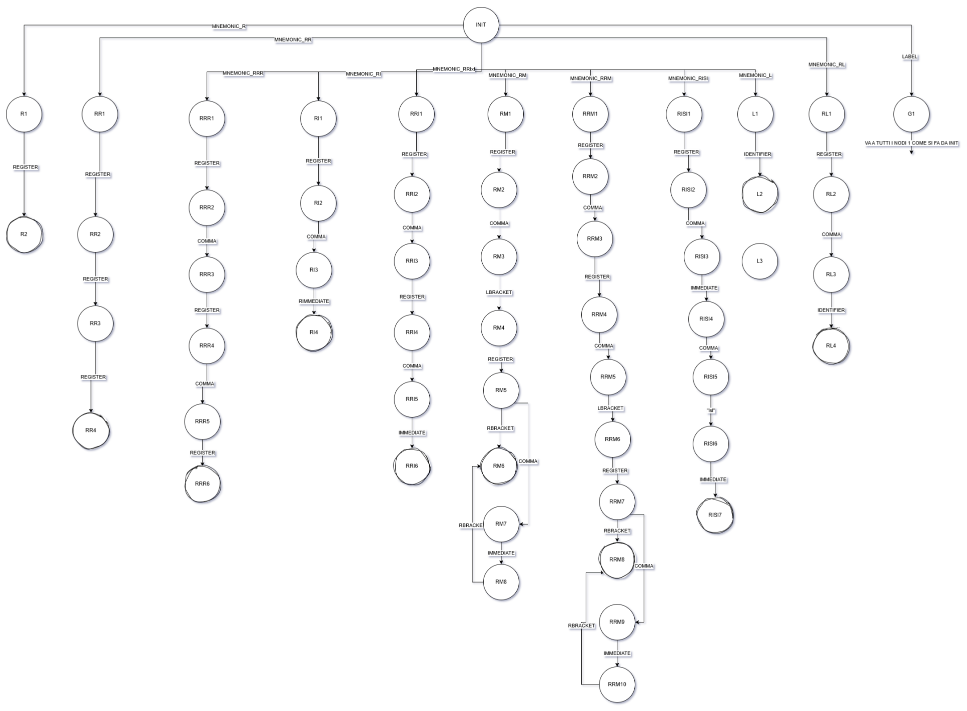


Figure 3.3: The old integer-instructions-only parser.

The structure of the parser allows it to be extended quite easily. This was done by recategorizing all the mnemonics into `X`, `S`, and `D` type mnemonics. For example, what once was a `MNEMONIC_RRR` (i.e. an instruction that operates on 3 registers), now is split into `XMNEMONIC_RRR`, `SMNEMONIC_RRR`, and `DMNEMONIC_RRR`, each of them being able to oeprate only on the compatible registers. By doing this, the parser now recognizes invalid hybrid instructions and refuses to run the program.

Visualizing the new registers Adding the visualization for the new registers required similar work as what was done with the displaying of the stack. A new `FloatRegisterPanel.java` class was created following the same structure as the integer registers'. The only real change besides the registers' labels was to correctly display the decimal representation of the floating point number stored inside the register.

Visualizing the data path The only thing left was to refactor the `SingleCycleVis.java` class to make it recognize the new types of mnemonics and thus provide a correct visualization for the kinds of operations they were performing.

Chapter 4

Current problems and further development

"Perfecting oneself is as much unlearning as it is learning."

Edsger W. Dijkstra

Concluding remarks

“The effort of using machines to mimic the human mind has always struck me as rather silly. I would rather use them to mimic something better.”

Edsger W. Dijkstra

Bibliography

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

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