

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

Summary	i
Introduction	iii
1 The LEGv8 simulators landscape	1
2 Building and modernizing the code base	7
3 Bug fixing and new features	8
4 Current pitfalls and suggestions for the future	9
Conclusions	10

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 instruc-

tions 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

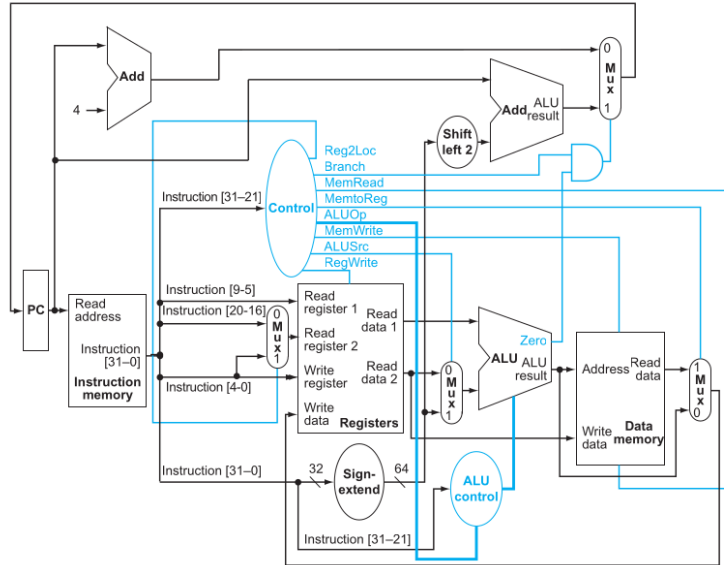


Figure 1: The logical scheme of the LEGv8 architecture

Architecture type

LEGv8 follows the Von Neumann architecture paradigm and thus contemplates the existence of a single memory containing both the instructions and

INTRODUCTION

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 readability 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

INTRODUCTION

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.

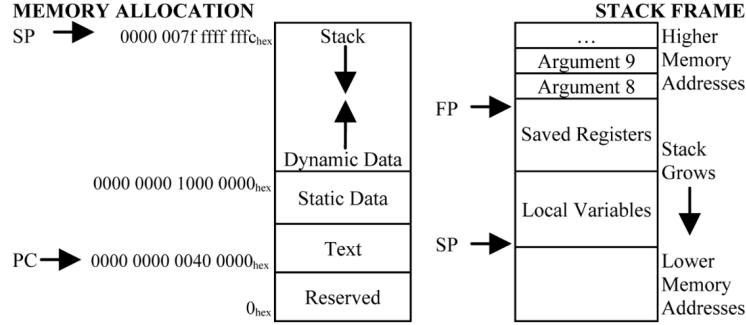


Figure 3: Logical division of the memory

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.

INTRODUCTION



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 *FADD* for performing the operation only on single precision or double precision registers respectively.

Instruction Mnemonic	Format	Opcode Width (bits)	Opcode Binary	Shamt Binary	11-bit Opcode Range (1) Start (Hex) End (Hex)	Instruction Mnemonic	Format	Opcode Width (bits)	Opcode Binary	Shamt Binary	11-bit Opcode Range (1) Start (Hex) End (Hex)
B	B	6	000101		0A0 0BF	ADD	R	11	10101011000		558
PHBLE	R	11	00011110001	000010	0F1	ADDI	I	10	1011000100		588 589
PDIVS	R	11	00011110001	000110	0F1	ORR	I	10	1011001000		590 591
PCMPS	R	11	00011110001	001000	0F1	CBE	CB	8	10110100		5A0 5A7
FADDS	R	11	00011110001	001010	0F1	CBZE	CB	8	10110101		5A8 5AF
PSUBS	R	11	00011110001	001110	0F1	STURW	D	11	10111000000		5C0
PHBLD	R	11	00011110011	000010	0F3	LDRSW	D	11	10111000100		5C4
PDIVD	R	11	00011110011	000110	0F3	STURB	R	11	10111100000		5E0
PCMPD	R	11	00011110011	001000	0F3	LDRBS	R	11	10111100010		5E2
FADD	R	11	00011110011	001010	0F3	STXR	D	11	11001000000		640
PSUBD	R	11	00011110011	001110	0F3	LDR	D	11	11001000010		642
STURB	D	11	00111000000		1C0	EOB	R	11	11001010000		650
LDRB	D	11	00111000010		1C2	SUB	R	11	11001011000		658
B-COND	CB	8	01010100		2A0 2A7	SUBI	I	10	1101000100		688 689
STURB	D	11	01111000000		3C0	EOPI	I	10	1101001000		690 691
LDRB	D	11	01111000010		3C2	MOVZ	IM	9	110100101		694 697
AND	R	11	10001010000		450	LSR	R	11	11010011010		69A
ADD	R	11	10001011000		458	LSL	R	11	11010011011		69B
ADDI	I	10	10010001000		488 489	RR	R	11	110100110000		6B0
ANDI	I	10	10010010000		490 491	ANDS	R	11	11101010000		750
BL	B	6	100101		4A0 4BF	SHBS	R	11	11101011000		758
SDIV	R	11	10011010110	000010	4D6	SHBS	I	10	1111000100		788 789
UDIV	R	11	10011010110	000011	4D6	ANDIS	I	10	1111001000		790 791
MUL	R	11	10011011000	011111	4D8	MOVK	IM	9	111100101		794 797
SMULB	R	11	10011011010		4DA	STUR	D	11	11111000000		7C0
UMULB	R	11	10011011110		4DE	LDR	D	11	11111000010		7C2
ORR	R	11	10101010000		550	STURD	R	11	11111100000		7E0
						LDRD	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

INTRODUCTION

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.

CORE INSTRUCTION FORMATS									
R	opcode		Rm		shamt		Rn		Rd
	31	21	20	16	15	10	9	5	4
I	opcode		ALU immediate				Rn		Rd
	31	22	21	10	9	5	4		0
D	opcode		DT address			op	Rn		Rt
	31	21	20	12	11	10	9	5	4
B	opcode		BR address						
	31	26	25						0
CB	Opcode		COND BR address						Rt
	31	24	23					5	4
IW	opcode		MOV immediate					Rd	
	31	21	20					5	4

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/earworthy/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/philbush/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/adi53153/LegV8	Verilog	Partial	Yes	No

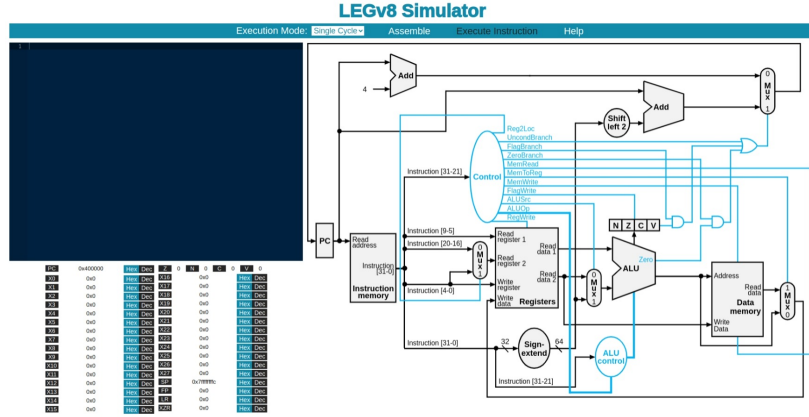
Table 1.2: The surveyed hardware simulators

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.

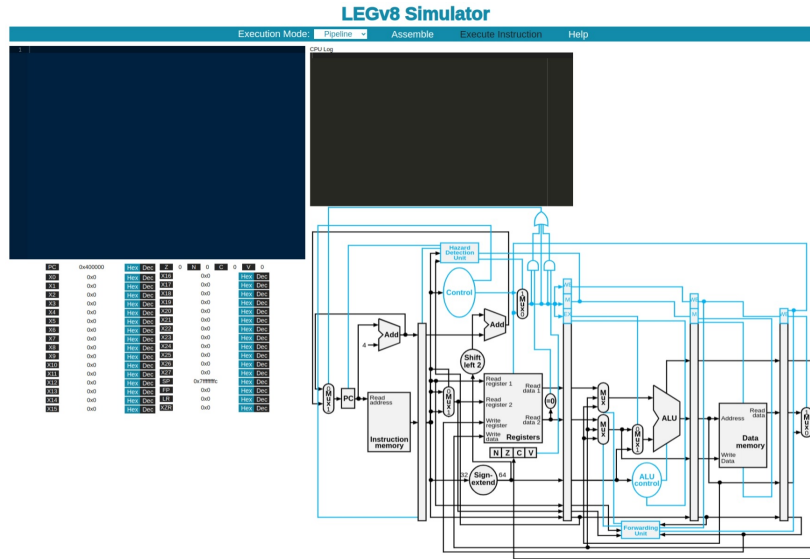
It is clear from this brief survey that the LEGv8 simulators space lacks any desirable candidates for code execution and inspection, as the software simulators are incomplete and platform-dependant, and the hardware ones lack interactivity and comprehensive visual output capabilities.

ARM’s LEGv8 simulator

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 visualization 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.



(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 with, and error display

- Clear and rich visualization of the X and flag registers and the data-path of the CPU
- 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:

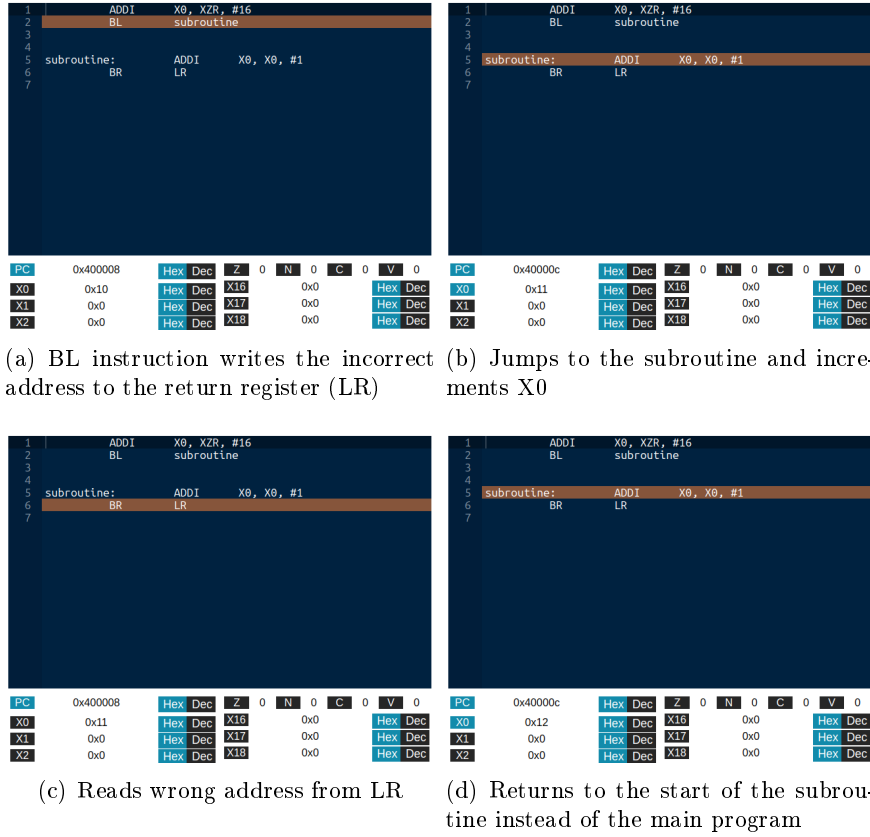


Figure 1.2: Branch returns to the wrong instruction, making it execute the branch in a loop

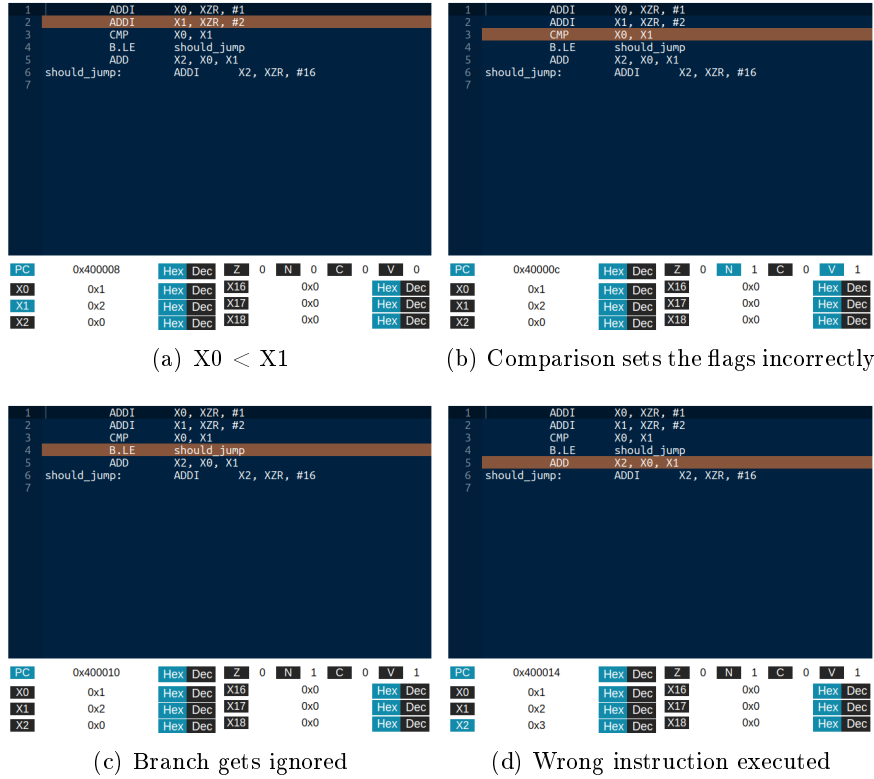


Figure 1.3: Comparisons do not set the correct flags and thus fail

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

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

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

Chapter 4

Current pitfalls and suggestions for the future

“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.