FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

eduARM: Web platform to support the teaching and learning of the ARM architecture

Maria Inês Fernandes Alves

Mestrado em Engenharia Informática e Computação

Supervisor: António Duarte Araújo

Second Supervisor: Bruno Lima

October 12, 2022

eduARM: Web platform to support the teaching and learning of the ARM architecture

Maria Inês Fernandes Alves

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

October 12, 2022

Abstract

Computer architecture is a prevalent topic of study in informatics and electrical engineering courses, though students' overall grasp of the concepts present in this subject is many times hampered, mainly due to the lack of educational tools that can intuitively represent the internal behaviour of a CPU. This already posed a problem in the teaching of the most commonly used processors in academic settings, and with the ever-growing evolution of the ARM architecture and its adoption in higher education institutions, the demand for this sort of tools has increased. Although a few works dedicated to this domain already exist, none dedicated to the ARM architecture and capable of effectively responding to this problem have been found. Nevertheless, certain tools contain valuable features which could present a good starting point for the platform to develop.

Taking this into account, this thesis aims to contribute towards solving this problem with the development of a practical and interactive web platform that simulates how a CPU functions, tailored specifically for the ARMv8 architecture. Since this tool's main purpose is to aid computer architecture students, contributing to an improvement in their learning experience, the platform must be modular and highly interactive, comprising varied concepts of computer architecture and organization in a simple and intuitive manner, such as showcasing the internal structure of a CPU, in both its unicycle and pipelined versions, and the effects of executing a set of instructions. This solution is thus a source of innovation, as educational tools for computer architecture, specifically the ARMv8 processor, are scarce and inadequate for what is necessary in an academic context.

Since this platform should encourage learning computer architecture through interactive experimentation, the developed solution is to be validated with a case study, along with the participation of students of the informatics and electrical engineering courses, which constitute the target group of this project.

Keywords: computer architecture teaching, ARM, simulation, learning resources

Resumo

Arquitetura de computadores é um tópico de estudo predominante nos cursos de Engenharia Informática e Eletrotécnica, embora a compreensão global dos conceitos presentes nesta unidade curricular seja muitas vezes dificultada, principalmente devido à falta de ferramentas educativas que possam representar intuitivamente o comportamento interno de um CPU. Este problema já se verificava no ensino dos processadores mais utilizados em ambientes académicos, e com a rápida evolução da arquitetura ARM e a sua adoção em diversas instituições do Ensino Superior, a procura por este tipo de ferramentas tem aumentado. Embora existam alguns trabalhos dedicados a este domínio, nenhum foi encontrado que fosse dedicado à arquitetura ARM e capaz de responder de forma eficiente ao problema. Apesar disto, certas ferramentas contêm funcionalidades vantajosas que podem ser um bom ponto de partida para o desenvolvimento da plataforma.

Tendo isto em conta, esta dissertação visa contribuir para a resolução deste problema através do desenvolvimento de uma plataforma web prática e interativa capaz de simular o funcionamento de um CPU, adaptada especificamente para a arquitetura ARMv8. Uma vez que o principal objetivo desta ferramenta é ajudar os estudantes de arquitetura de computadores, contribuindo para uma melhor experiência de aprendizagem, a plataforma deve ser modular e interativa, abordando vários conceitos de arquitetura e organização de computadores de uma forma simples e intuitiva, tais como mostrar a estrutura interna de um CPU, tanto nas suas versões uniciclo e pipeline, como mostrar os efeitos da execução de um conjunto de instruções. Esta solução assume-se assim como uma fonte de inovação, uma vez que as ferramentas educacionais dedicadas a arquitetura de computadores, especificamente para o processador ARMv8, são escassas e inadequadas para o que é necessário num contexto académico.

Uma vez que esta plataforma deve encorajar a aprendizagem de arquitetura de computadores através de experimentação interativa, a solução desenvolvida deve ser validada com um caso de estudo, contando com a participação de estudantes dos cursos de Engenharia Informática e Eletrotécnica, que constituem o público-alvo deste projeto.

Palavras-chave: ensino de arquitetura de computadores, ARM, simulação, recursos de aprendizagem

Acknowledgements

Gostaria de agradecer, antes de mais, ao meu orientador, o Professor António Duarte Araújo, e ao meu co-orientador, o Professor Bruno Lima, por me darem a oportunidade de desenvolver este trabalho, e por me terem acompanhado ao longo deste ano no desenvolvimento deste projeto, tendo sempre disponibilidade para me ajudar.

Queria agradecer, também, à minha família, especialmente aos meus pais, irmã e avó, por terem estado do meu lado ao longo destes anos, por sempre me motivarem e acreditarem nas minhas capacidades. Espero que me ver a concluir o curso vos traga um pouco de orgulho.

Daniela, obrigada especialmente a ti, por aturares os texugos que te mando constantemente como resposta, e pelas fotos dos (lindos!) gatos que foram a origem de grande parte da minha motivação. Obrigada também ao Miguel, rei do tech support.

Ao Francisco, com quem já é mais de uma década juntos, obrigada por todos estes anos, por me aturares desde a escola até aqui, e mal posso esperar para seres tu deste lado. Desculpa a minha ausência estes últimos meses, que mais pareceram anos.

João, juntos desde o primeiro dia, tenho muito orgulho do que conseguimos fazer. Adoro-te sempre.

Márcia, és um exemplo de esforço e dedicação, tenho imenso orgulho em ti, e quero que me tragas um bacalhau.

Henrique, sabes que sem ti isto não era possível. Já tinha ficado maluca. Adoro-te, e obrigada por tudo, mesmo.

A todos os restantes membros do SCTP, Bruno, Campos, Nunito, Pedrito, Vitor, obrigada por estes cinco anos, e que venham muitos mais.

Inês

"Have no fear of perfection - you'll never reach it."

Salvador Dalí

Contents

List of Figures

List of Tables

Listings

Abbreviations

ALU	Arithmetic Logic Unit
API	Application Programming Interface
CA	Computer Architecture
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
CSS	Cascading Style Sheets
FEUP	Faculty of Engineering of the University of Porto
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
ISA	Instruction Set Architecture
I/O	Input/Output
IDE	Integrated Development Environment
L.EIC	Bachelor in Informatics and Computing Engineering
MIPS	Microprocessor without Interlocked Pipeline Stages
PC	Personal Computer
RISC	Reduced Instruction Set Computer
VPN	Virtual Private Network

Chapter 1

Introduction

In this chapter, the main context and motivation for this thesis are introduced, as well as the goals it strives to achieve, finishing with a general overview of the document's structure.

1.1 Context and Motivation

Computer Architecture (CA) is a fundamental subject in higher education technology courses, namely Informatics and Electrical Engineering. Such is the case of the Bachelor in Informatics and Computing Engineering (L.EIC) on the Faculty of Engineering of the University of Porto (FEUP) and its CA course unit - among other topics, students can learn about the major internal subsystems of a computer, the general architecture of its platform, and assembly programming based on the ARMv8 64-bit architecture.

The Central Processing Unit (CPU) itself and its internal behaviour is a key study item, and one that students tend to exhibit difficulties in understanding. This problem is a result of the scarcity of intuitive educational tools that can graphically represent the CPU and the impact of executing a set of instructions [14].

Another problem is raised with the evolution and popularity of the ARM architecture, specifically ARMv8. This architecture is currently the most popular instruction set architecture (ISA) in the industry [17], that is widely used in smartphones, laptops, and other embedded systems. Consequently, this caused certain higher education institutions to adapt their syllabus to include the teaching of this architecture instead of the previously lectured MIPS or RISC-V architectures [16].

Similarly to what was already identified as a problem in the teaching of the most common processors [14], suitable platforms for teaching ARMv8 are scarce, which increases the need for developing an adequate software tool. Such a tool would encourage CA students to learn through interactive experimentation, allowing them, for example, to thoroughly consult the CPU's datapath, in both its unicycle and pipelined versions, for each instruction in a set written in assembly language.

To contribute towards solving this issue, this thesis aims to develop a practical and interactive web platform that simulates how a CPU works, built for ARMv8 education, that can thus be used in higher education with the objective of improving student comprehension and productivity.

1.2 Goals

This project was created with the interest of CA students in mind, anticipating that they can take maximum advantage of the developed tool and thus better comprehend the topics approached in classes. Taking this motivation into account, a set of goals for this thesis were defined as follows:

- Deliver a practical and intuitive web platform suitable for teaching CA, specifically the ARMv8 ISA. This platform should be capable of replicating, in a simple manner, the behaviour of the CPU, in both its unicycle and pipelined versions, displaying its components and their states during the execution of a given assembly program.

- Validate the solution with former CA students of both the Informatics and Electrical Engineering courses at FEUP, as well as collect feedback for further improvements.

- Analyse and discuss the results obtained in order to understand the value of the developed solution and how much of an impact it can have on student comprehension of the topics lectured in CA.

- Document the work developed in paper format, with the objective of being submitted to a conference

In order to achieve these goals, a work plan for this project was devised and is documented throughout the following chapters.

1.3 Document Structure

Following the present chapter, this document includes a review and discussion of state of the art in Chapter 2, where related work is analysed in detail. In Chapter 3, the requirements set for the project and an overall summary of the development process are provided. An in-depth explanation of the platform's backend is provided in Chapter 4, followed by its correspondent frontend in Chapter 5. Following these chapters on methodology, Chapter 6 features the results obtained from validation with students and their analysis. To finalize, conclusions and proposals for future work are disclosed in Chapter 7.

Chapter 2

State of the Art

In this chapter, previous work on tools for CA education is presented and analysed. An overview of the ARM architecture, necessary for a better understanding of the project and its requirements, is provided. For each of the projects presented, their various features are discussed, as well as their positive and negative aspects. Finally, in the last section, a comparison is made between the presented tools, also highlighting which features are desirable in the platform this thesis aims to develop and what contribution it can give to the field.

2.1 Background

In order to develop an adequate platform to teach the ARM architecture, educational tools already used in this domain should first be studied, as to better understand which features are most important to include, and how these tools' good practices can be adopted. This state of the art analysis, however, requires some prior contextualization on how an architecture such as ARMv8 works, which is provided in this section.

To understand how any computer architecture works, one must know what exactly an architecture is. An architecture is defined as an abstract model of a computer that defines the interface between the hardware and the lowest-level software. It encapsulates everything necessary to write a machine language program, including instructions, data types, registers, memory access and I/O.

Another important concept to understand is the CPU, or processor for short. As David A. Patterson and John L. Hennessy mentioned in their book Computer Organization and Design (Arm Edition) [17], the five standard components of a computer are input, output, memory, datapath, and control. The combination of the datapath and control units is commonly referred to as the processor, the active part of a computer capable of executing a set of instructions within a program.

A processor can have multiple possible implementations, and each of them entails different clock cycle times and number of clock cycles per instruction. These two variables, along with the number of program instructions, are what impact a computer's performance. Fundamentally, different CPU versions will perform in different ways, which is the case for the two approaches considered in this project: the unicycle and the pipeline.

2.1.1 Unicycle

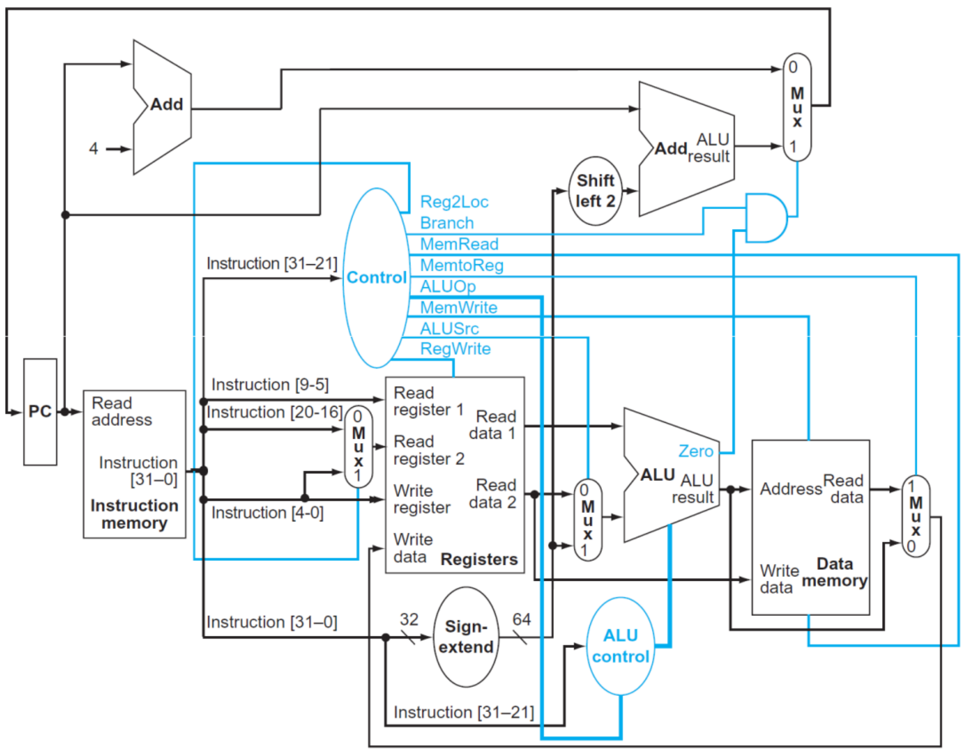


Figure 2.1: Unicycle CPU Datapath [17]

As seen in Figure 2.1, the CPU comprises various components connected by buses. Highlighted blue is the control unit, whose signals' values, depending on the instruction to execute, influence the datapath. Specifically, every component type has a different behaviour, sometimes changing according to the control values. Within the datapath, each connection carries important data from one component to another, and the sequential actions of each component make it possible to execute a given instruction.

In this CPU version, each instruction is executed in a single clock cycle. This means this implementation, albeit straightforward and easy to understand, is not very efficient for modern designs [17], which is why another technique, pipelining, was created to improve performance.

2.1.2 Pipeline

Although its components remain mostly the same, the behaviour of a pipelined CPU differs from its unicycle version. In this case, the datapath is divided into five stages, with each taking one clock cycle:

- IF (Instruction Fetch): Fetch the instruction from memory
- ID (Instruction Decode): Read registers and decode instruction
- EX (Execute): Execute the operation or calculate an address
- MEM (Memory access): Access an operand in data memory
- WB (Write back): Write the result back into a register

Dividing an execution's tasks into five steps makes it so that an instruction can only be present at one stage at a given time. This means that, for example, while an instruction is in its decoding phase, the next instruction is already being fetched from memory.

This overlapping of multiple instructions in a program's execution is, indeed, the technique called pipelining, used universally and capable of, under ideal circumstances, making a five-stage pipeline nearly five times faster than its unicycle version [17]. The reason behind this increase in performance is that pipelining increases instruction throughput, instead of decreasing the execution time of just one instruction - and the former offers a significant advantage in terms of total execution time.

The full pipeline CPU datapath is represented in Figure 2.2.

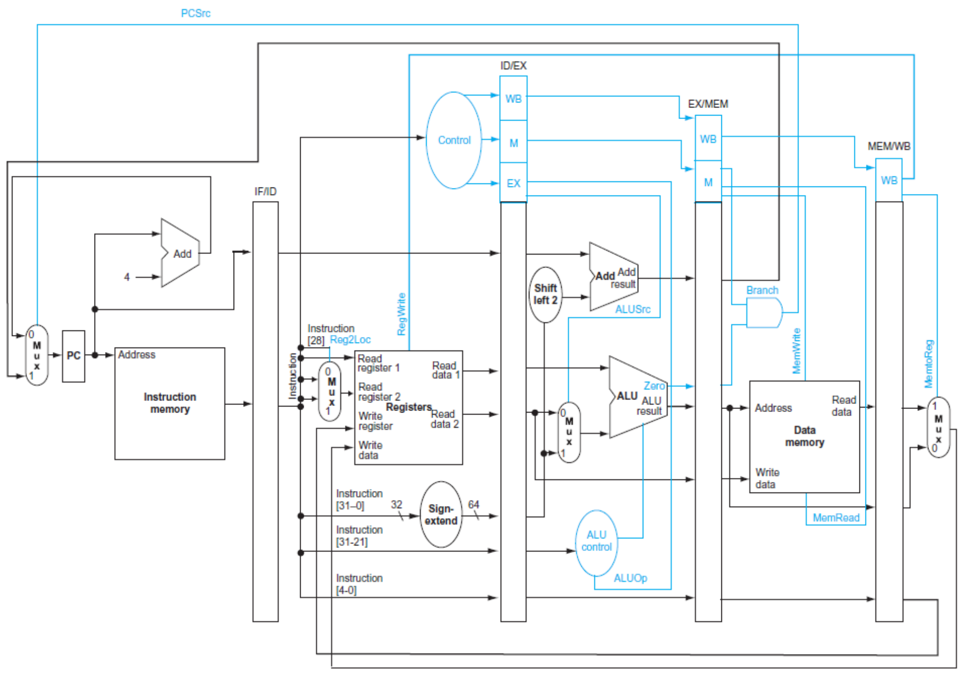


Figure 2.2: Pipeline CPU Datapath [17]

A CPU can also be classified as either CISC (Complex Instruction Set Computer) or RISC (Reduced Instruction Set Computer). RISC architectures, considering their simple instruction sets and fewer clock cycles, make up a much less complicated processor compared to CISC [10]. Taking this simplicity into account, CA courses such as FEUP's choose to adopt this ISA for teaching the CPU [16], thus this chapter focuses solely on tools dedicated to RISC architectures, namely MIPS, RISC-V and ARM. All of the tools presented are open-source, which means their code base is free for developers to expand for innovation.

2.2 MIPS

2.2.1 SPIM

SPIM is an open-source, self-contained MIPS simulator that runs assembly language programs. Developed by James Larus, it is considered the most widely known and used MIPS simulator [22], both for education and the industry. This program has an updated version, called QtSpim, that unlike the previous versions, runs on Microsoft Windows, Mac OS X, and Linux, and is the one that is currently actively maintained [12].

This platform's interface, as seen in Figure 2.3, contains a panel dedicated to register visualization, accompanied by an area that displays the text and data segments, and a console window on the lower part of the screen. *SPIM* does not provide an integrated code editor, which means users must write code externally and then load it. Additionally, this tool allows users to modify register and memory values, set breakpoints and execute on single step or multi step mode.

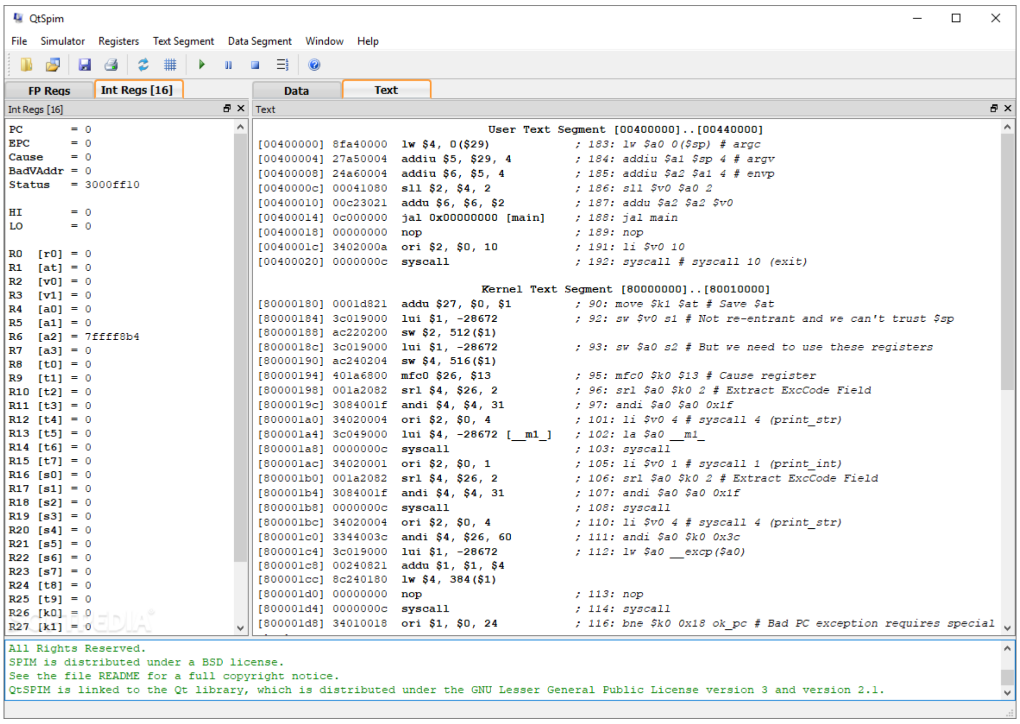
Although *SPIM* can be used for debugging assembly programs, in order for the tool to thoroughly support the teaching of CA, CPU datapath visualization is missing. This functionality is essential for understanding the internal behaviour of the CPU, as well as support for the pipelined version of the CPU.

2.2.2 WebMIPS

WebMIPS is an educational web-based MIPS simulation environment, developed by the University of Siena in Italy and written in the ASP language. This platform is designed for Web browsers, which means users can have access to the software regardless of their operating system.

More specifically, this tool is a five stage MIPS pipeline simulator (thus not including support for the unicycle version of the processor), conceived with the objective of teaching assembly programming and mastering pipeline, control and datapath design [6]. This means students can explore what type of hazards can occur in the system, and how the CPU detects and solves them in the pipeline, using data forwarding and stalls.

As seen in Figure 2.4, the interface includes an area for visualization of instructions, data memory and registers, accompanied by a display of which parts of the pipeline are stalled. The CPU datapath and its components are shown in the center of the screen, with the wires related to the data or control paths differentiated by color. The user can, in the commands panel, load any MIPS assembly file on a simple editor or use one of the built-in examples. Each program can then be executed all at once or step-by-step, where the student can explore what happens in each stage of the pipeline - by clicking on each component of the processor, along with a small description, data input and output data in each clock cycle is displayed.


QtSpim
File Simulator Registers Text Segment Data Segment Window Help
FP Regs
Int Regs [16]
Data
Text
Int Regs [16]
Text

```
PC       = 0
EPC      = 0
Cause    = 0
BadVAddr = 0
Status   = 3000ff10

HI       = 0
LO       = 0

R0  [r0] = 0
R1  [at] = 0
R2  [v0] = 0
R3  [v1] = 0
R4  [a0] = 0
R5  [a1] = 0
R6  [a2] = 7ffff8b4
R7  [a3] = 0
R8  [t0] = 0
R9  [t1] = 0
R10 [t2] = 0
R11 [t3] = 0
R12 [t4] = 0
R13 [t5] = 0
R14 [t6] = 0
R15 [t7] = 0
R16 [s0] = 0
R17 [s1] = 0
R18 [s2] = 0
R19 [s3] = 0
R20 [s4] = 0
R21 [s5] = 0
R22 [s6] = 0
R23 [s7] = 0
R24 [t8] = 0
R25 [t9] = 0
R26 [k0] = 0
R27 [k1] = 0
```

```
                              User Text Segment [00400000]..[00440000]
[00400000] 8fa40000  lw $4, 0($29)             ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29, 4          ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004  addiu $6, $5, 4           ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080  sll $2, $4, 2             ; 186: sll $v0 $a0 2
[00400010] 00c23021  addu $6, $6, $2           ; 187: addu $a2 $a2 $v0
[00400014] 0c000000  jal 0x00000000 [main]     ; 188: jal main
[00400018] 00000000  nop                       ; 189: nop
[0040001c] 3402000a  ori $2, $0, 10            ; 191: li $v0 10
[00400020] 0000000c  syscall                   ; 192: syscall # syscall 10 (exit)

                             Kernel Text Segment [80000000]..[80010000]
[80000180] 0001d821  addu $27, $0, $1          ; 90: move $k1 $at # Save $at
[80000184] 3c019000  lui $1, -28672            ; 92: sw $v0 s1 # Not re-entrant and we can't trust $sp
[80000188] ac220200  sw $2, 512($1)
[8000018c] 3c019000  lui $1, -28672            ; 93: sw $a0 s2 # But we need to use these registers
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6800  mfc0 $26, $13             ; 95: mfc0 $k0 $13 # Cause register
[80000198] 001a2082  srl $4, $26, 2            ; 96: srl $a0 $k0 2 # Extract ExcCode Field
[8000019c] 3084001f  andi $4, $4, 31           ; 97: andi $a0 $a0 0x1f
[800001a0] 34020004  ori $2, $0, 4             ; 101: li $v0 4 # syscall 4 (print_str)
[800001a4] 3c049000  lui $4, -28672 [__m1_]    ; 102: la $a0 __m1_
[800001a8] 0000000c  syscall                   ; 103: syscall
[800001ac] 34020001  ori $2, $0, 1             ; 105: li $v0 1 # syscall 1 (print_int)
[800001b0] 001a2082  srl $4, $26, 2            ; 106: srl $a0 $k0 2 # Extract ExcCode Field
[800001b4] 3084001f  andi $4, $4, 31           ; 107: andi $a0 $a0 0x1f
[800001b8] 0000000c  syscall                   ; 108: syscall
[800001bc] 34020004  ori $2, $0, 4             ; 110: li $v0 4 # syscall 4 (print_str)
[800001c0] 3344003c  andi $4, $26, 60          ; 111: andi $a0 $k0 0x3c
[800001c4] 3c019000  lui $1, -28672            ; 112: lw $a0 __excp($a0)
[800001c8] 00240821  addu $1, $1, $4
[800001cc] 8c240180  lw $4, 384($1)
[800001d0] 00000000  nop                       ; 113: nop
[800001d4] 0000000c  syscall                   ; 114: syscall
[800001d8] 34010018  ori $1, $0, 24            ; 116: bne $k0 0x18 ok_pc # Bad PC exception requires special
```

```
All Rights Reserved.
SPIM is distributed under a BSD license.
See the file README for a full copyright notice.
QtSPIM is linked to the Qt library, which is distributed under the GNU Lesser General Public License version 3 and version 2.1.
```

Figure 2.3: SPIM: A MIPS32 Simulator

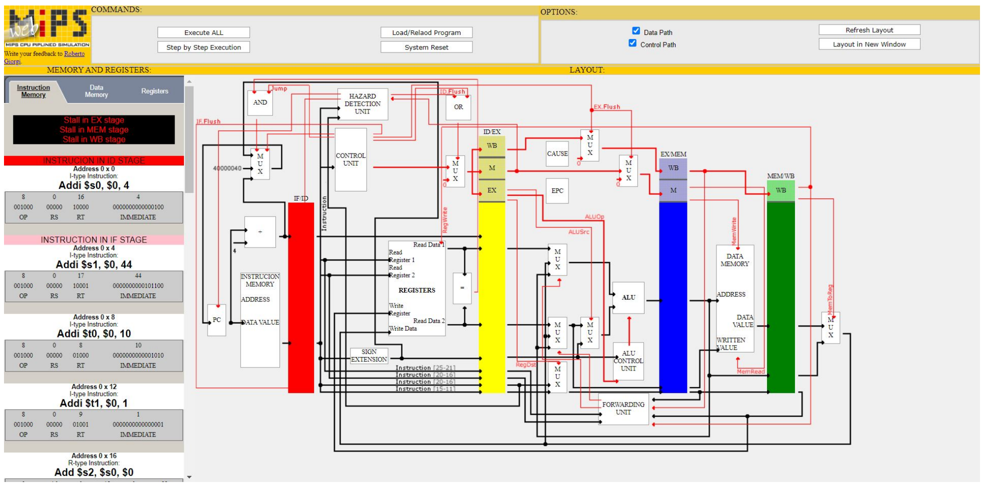


Figure 2.4: WebMIPS - Web-Based MIPS Simulation Environment

Similarly to the simulator previously presented, *WebMIPS* solely supports the MIPS architecture and, in this case, only the pipeline version of the CPU, which poses a great disadvantage, compared with other tools that can support both unicycle and pipeline. The visualization is mostly static, as the datapath seemingly does not change with each instruction (the user must click on each component in order to see the differences) and there is a need to refresh the layout to enable or disable the data and control paths. The platform can, therefore, be seen as outdated and easily surpassed by other similar, more recent tools.

2.2.3 MARS

MARS, the Mips Assembly and Runtime Simulator, was created by the Missouri State University, and its objective is to assemble and simulate the execution of MIPS assembly language programs. This tool can be used either from a command line or through its integrated development environment (IDE).

This platform is an extremely complete assembly debugger, used in various CA courses all around the world, containing, as of Release 4.0, "155 basic instructions of the MIPS-32 instruction set, approximately 370 pseudo-instructions or instruction variations, the 17 syscall functions mainly for console and file I/O defined by SPIM, and an additional 22 syscalls for other uses such as MIDI output, random number generation and more" [21].

As seen in Figure 2.5, *MARS* has a fairly simple and easy to use interface, which contains an integrated editor with multiple file-editing tabs and syntax highlighting dedicated to assembly programming, accompanied by a panel for showcasing register values, that can be either displayed in hexadecimal or decimal, and a console window on the lower part of the interface.

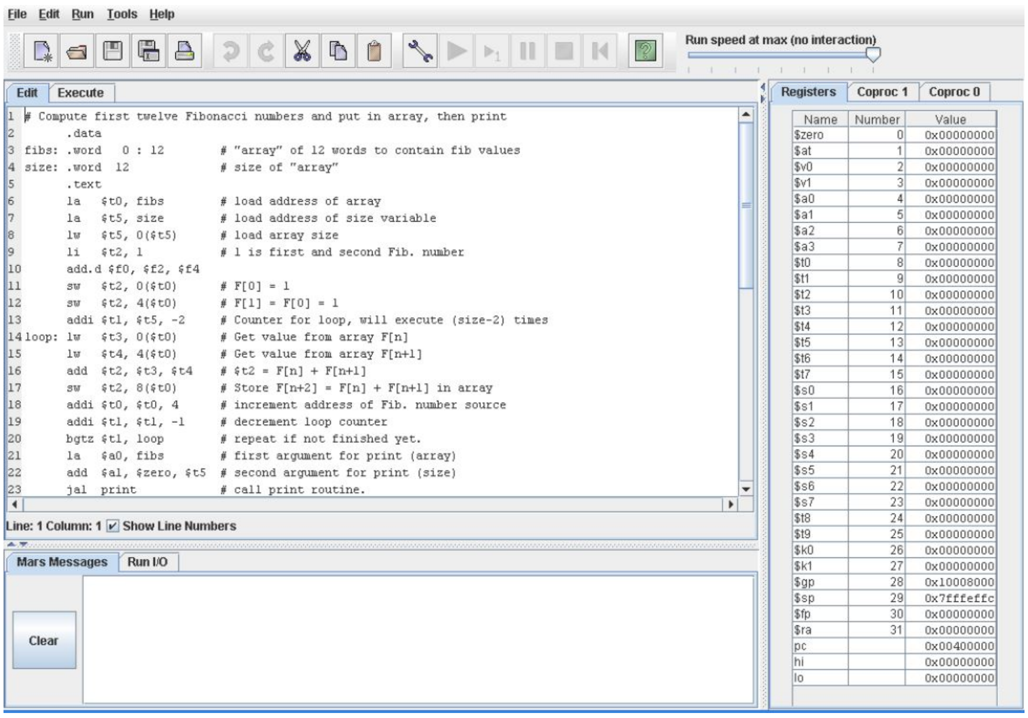
Each program can be executed all at once or step-by-step. During the execution, the user can see the compiled code and the data segment, as well as the resulting inputs and outputs. Users can also create and remove execution breakpoints and go back and forth through a set of instructions while viewing or editing register and memory values.

This assembly simulator was designed as an alternative to *SPIM*, tackling most of its shortcomings and greatly outperforming it.

Despite being a robust and useful tool for assembly debugging, *MARS* has no CPU datapath visualization, which, in order for students to thoroughly understand how the processor functions, is considered a crucial feature. Moreover, this tool executes the instructions according to the unicycle version of the CPU, with no support for pipelined architectures. Another disadvantage is the program not being available on the Web, as it must be downloaded and may not be available for every user's operating system.

2.2.4 DrMIPS

DrMIPS is an open-source graphical simulator of the MIPS processor, specifically designated for teaching and learning CA [14]. This project was developed in a Master's thesis at FEUP, having since been updated with various bug fixes and interface improvements.



```
# Compute first twelve Fibonacci numbers and put in array, then print
      .data
fibs: .word   0 : 12        # "array" of 12 words to contain fib values
size: .word  12             # size of "array"
      .text
      la   $t0, fibs        # load address of array
      la   $t5, size        # load address of size variable
      lw   $t5, 0($t5)      # load array size
      li   $t2, 1           # 1 is first and second Fib. number
      add.d $f0, $f2, $f4
      sw   $t2, 0($t0)      # F[0] = 1
      sw   $t2, 4($t0)      # F[1] = F[0] = 1
      addi $t1, $t5, -2     # Counter for loop, will execute (size-2) times
loop: lw   $t3, 0($t0)      # Get value from array F[n]
      lw   $t4, 4($t0)      # Get value from array F[n+1]
      add  $t2, $t3, $t4    # $t2 = F[n] + F[n+1]
      sw   $t2, 8($t0)      # Store F[n+2] = F[n] + F[n+1] in array
      addi $t0, $t0, 4      # increment address of Fib. number source
      addi $t1, $t1, -1     # decrement loop counter
      bgtz $t1, loop        # repeat if not finished yet.
      la   $a0, fibs        # first argument for print (array)
      add  $a1, $zero, $t5  # second argument for print (size)
      jal  print            # call print routine.
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

Figure 2.5: MARS, the Mips Assembly and Runtime Simulator

DrMIPS provides students with a robust and highly intuitive tool that includes fundamental principles lectured in CA, such as the components and behaviour of the CPU, on both its unicycle and pipelined versions, latencies and critical path, and assembly programming. Its educational focus and various functionalities make *DrMIPS* a greatly valuable tool, having it been used at FEUP to support students in CA classes.

The platform is available for free on both PCs and Android devices, namely tablets, which allows students to use the application wherever they want. In terms of functionalities, it includes step-by-step assembly instruction execution, accompanied by CPU datapath visualization. Users can thus create (or import) their own assembly programs and explore what happens within the CPU and its components while executing certain instructions, encouraging learning through interactive experimentation. Moreover, the CPU datapath is highly configurable, as CPU components can be created externally and imported into the simulator, allowing users to customize it according to their needs.

DrMIPS only supports instruction sets that the datapath itself can support, which, in turn, makes up for a limited set of instructions (for example, floating point and shift operations, syscalls, *jal* and *jr* are not supported) - besides these exceptions, the instruction sets used by the CPU are highly configurable.

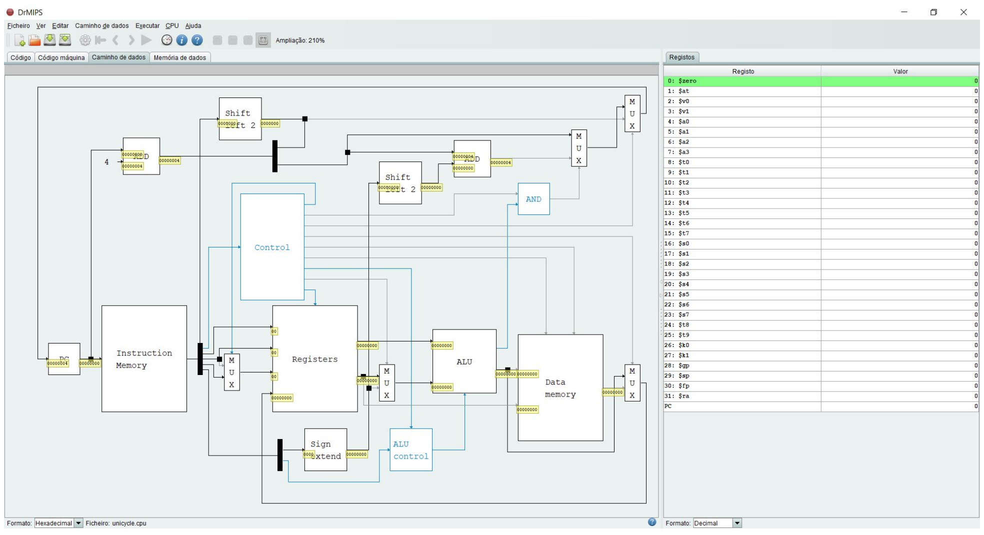
This tool displays the values of the registers in each step of execution, on the right side of the screen, so students can see, along with the CPU datapath, what influence certain instructions have on these registers. Besides this, DrMIPS also displays the assembled instructions in machine code and the register values on the data memory, which are also important concepts approached in CA classes.

In regards to further visualization, the datapath shows, for each execution of a set of instructions, the current values at input or output of every CPU component and which wires are relevant to the current instruction. Students can thus, at each step, understand how the datapath and its values change with a specific instruction.

Support for both unicycle and pipeline versions of the processor is provided, which allows users to experiment with both versions and, in the case of the pipeline CPU, learn what types of hazards can arise while using a pipeline and how the system solves them.

Moreover, the platform includes a "performance mode" where latency is simulated, allowing for visualization of the CPU critical path, in order to better understand how latencies can have an effect on it, as well as showcasing performance metrics, such as time of execution and clock cycles, that can highlight the impact of each execution on the CPU's performance.

Regarding its user interface, *DrMIPS* is simple and intuitive, providing useful features such as tooltips for data visualization, which allows students to interact with components and check their current status and values, support for multiple representation formats (binary, decimal and hexadecimal), and an integrated code editor with auto-complete and syntax highlighting. *DrMIPS* also has support for multiple languages, dark mode, and different possibilities for window layout.



Registo	Valor
0: $zero	0
1: $at	0
2: $v0	0
3: $v1	0
4: $a0	0
5: $a1	0
6: $a2	0
7: $a3	0
8: $t0	0
9: $t1	0
10: $t2	0
11: $t3	0
12: $t4	0
13: $t5	0
14: $t6	0
15: $t7	0
16: $s0	0
17: $s1	0
18: $s2	0
19: $s3	0
20: $s4	0
21: $s5	0
22: $s6	0
23: $s7	0
24: $t8	0
25: $t9	0
26: $k0	0
27: $k1	0
28: $gp	0
29: $sp	0
30: $fp	0
31: $ra	0
PC	0

Figure 2.6: DrMIPS - Educational MIPS simulator

Although this intuitive and educational tool, with its various features described above, seems ideal for supporting CA students in their studies, the fact that it exclusively supports the MIPS architecture impedes its adoption in more recent higher education courses that currently work with the ARMv8 processor.

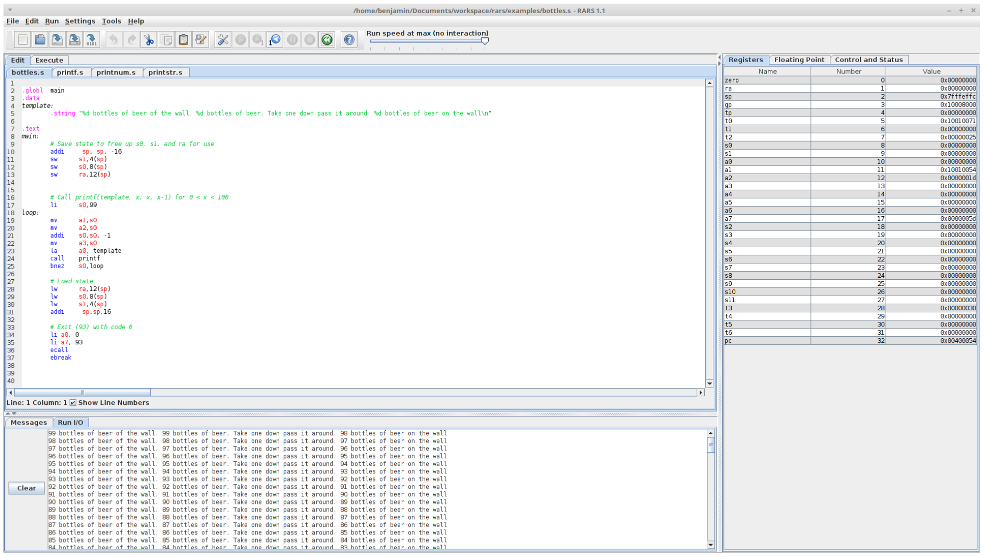
Another disadvantage arises considering that more recent simulators, as presented in the next section, can already be used on the Web with support for multiple browsers, making it even easier to have immediate access to these platforms, with no need to download them.

Nevertheless, its focus on education and multitude of functionalities and visualization, that help students learn the MIPS architecture, make DrMIPS an extremely valuable platform whose vision serves as a model for a new platform to develop, this time dedicated to the ARMv8 ISA.

2.3 RISC-V

2.3.1 RARS

RARS, or RISC-V Assembler and Runtime Simulator, is a direct port of the MIPS simulator *MARS*, described in subsection 2.2.3, for teaching the RISC-V architecture. This platform, created with the intent of delivering an intuitive development environment for those new to this architecture, uses *MARS*' source code and adapts it to RISC-V, maintaining its user interface, as seen in Figure 2.7.


/home/benjamin/Documents/workspace/rars/examples/bottles.s - RARS 1.1
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
bottles.s printf.s printnum.s printstr.s
Line: 1 Column: 1 Show Line Numbers
Messages Run I/O
Clear
Registers Floating Point Control and Status

```
.globl  main
.data
template:
        .string "%d bottles of beer of the wall. %d bottles of beer. Take one down pass it around. %d bottles of beer on the wall\n"

.text
main:
        # Save state to free up s0, s1, and ra for use
        addi     sp, sp, -16
        sw      s1,4(sp)
        sw      s0,8(sp)
        sw      ra,12(sp)


        # Call printf(template, x, x, x-1) for 0 < x < 100
        li      s0,99
loop:
        mv      a1,s0
        mv      a2,s0
        addi    s0,s0, -1
        mv      a3,s0
        la      a0, template
        call    printf
        bnez    s0,loop

        # Load state
        lw      ra,12(sp)
        lw      s0,8(sp)
        lw      s1,4(sp)
        addi     sp,sp,16

        # Exit (93) with code 0
        li a0, 0
        li a7, 93
        ecall
        ebreak
```

```
99 bottles of beer of the wall. 99 bottles of beer. Take one down pass it around. 98 bottles of beer on the wall
98 bottles of beer of the wall. 98 bottles of beer. Take one down pass it around. 97 bottles of beer on the wall
97 bottles of beer of the wall. 97 bottles of beer. Take one down pass it around. 96 bottles of beer on the wall
96 bottles of beer of the wall. 96 bottles of beer. Take one down pass it around. 95 bottles of beer on the wall
95 bottles of beer of the wall. 95 bottles of beer. Take one down pass it around. 94 bottles of beer on the wall
94 bottles of beer of the wall. 94 bottles of beer. Take one down pass it around. 93 bottles of beer on the wall
93 bottles of beer of the wall. 93 bottles of beer. Take one down pass it around. 92 bottles of beer on the wall
92 bottles of beer of the wall. 92 bottles of beer. Take one down pass it around. 91 bottles of beer on the wall
91 bottles of beer of the wall. 91 bottles of beer. Take one down pass it around. 90 bottles of beer on the wall
90 bottles of beer of the wall. 90 bottles of beer. Take one down pass it around. 89 bottles of beer on the wall
89 bottles of beer of the wall. 89 bottles of beer. Take one down pass it around. 88 bottles of beer on the wall
88 bottles of beer of the wall. 88 bottles of beer. Take one down pass it around. 87 bottles of beer on the wall
87 bottles of beer of the wall. 87 bottles of beer. Take one down pass it around. 86 bottles of beer on the wall
86 bottles of beer of the wall. 86 bottles of beer. Take one down pass it around. 85 bottles of beer on the wall
85 bottles of beer of the wall. 85 bottles of beer. Take one down pass it around. 84 bottles of beer on the wall
84 bottles of beer of the wall. 84 bottles of beer. Take one down pass it around. 83 bottles of beer on the wall
```

Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x7fffeffc
gp	3	0x10008000
tp	4	0x00000000
t0	5	0x10010071
t1	6	0x00000000
t2	7	0x00000025
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x00000000
a1	11	0x10010054
a2	12	0x0000001d
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000
a6	16	0x00000000
a7	17	0x0000005d
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000000
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x00000030
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000
pc	32	0x00400054

Figure 2.7: RARS, the RISC-V Assembler and Runtime Simulator

Besides the changes associated with switching from MIPS to RISC-V, this platform includes major internal restructuring and refactoring, such as removing small features present in *MARS* that were considered irrelevant or incompatible with RISC-V [11].

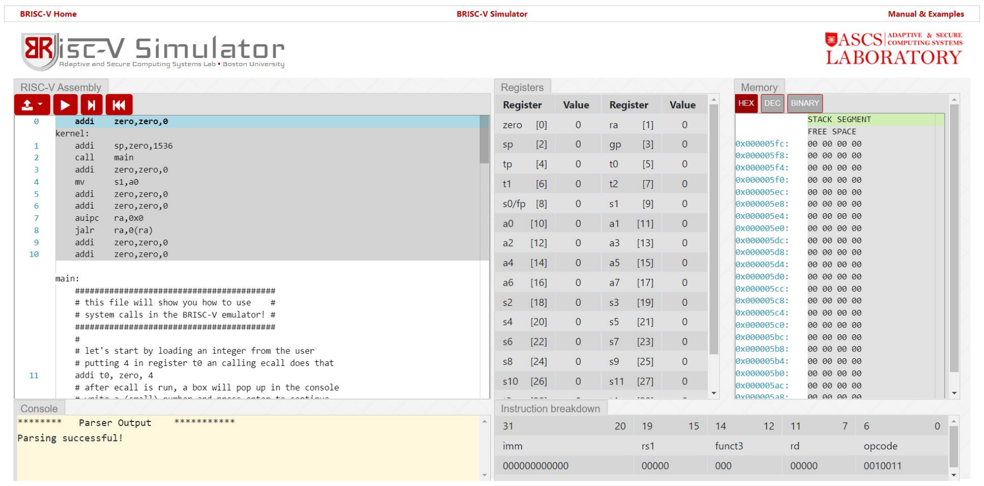
RARS, much like its MIPS counterpart, focuses intensively on assembly debugging, containing essentially the same features described in 2.2.3. As a consequence, this tool also lacks CPU datapath visualization and support for pipelined architectures, which are considered key features for teaching the CPU's structure and behaviour.

2.3.2 BRISC-V Simulator

The *BRISC-V Simulator* is a browser-based assembly programming simulator developed by the Adaptive and Secure Computing Systems Laboratory at Boston University. Together with *BRISC-*

V Explorer, it makes up the *BRISC-V Platform* [1]. This simulator's main objective is to provide an educational environment for writing, compiling, assembling and executing RISC-V code.

Students can learn about several low level CA concepts, such as calling conventions, memory allocation and the flow of execution of a set of instructions. As it is available on the web, users can easily access the simulator and study the RISC-V architecture regardless of their platform.



```
    addi    zero,zero,0
kernel:
    addi    sp,zero,1536
    call    main
    addi    zero,zero,0
    mv      s1,a0
    addi    zero,zero,0
    addi    zero,zero,0
    auipc   ra,0x0
    jalr    ra,0(ra)
    addi    zero,zero,0
    addi    zero,zero,0

main:
    #########################################
    # this file will show you how to use    #
    # system calls in the BRISC-V emulator! #
    #########################################
    #
    # let's start by loading an integer from the user
    # putting 4 in register t0 an calling ecall does that
    addi t0, zero, 4
    # after ecall is run, a box will pop up in the console
```

Register	Value	Register	Value
zero [0]	0	ra [1]	0
sp [2]	0	gp [3]	0
tp [4]	0	t0 [5]	0
t1 [6]	0	t2 [7]	0
s0/fp [8]	0	s1 [9]	0
a0 [10]	0	a1 [11]	0
a2 [12]	0	a3 [13]	0
a4 [14]	0	a5 [15]	0
a6 [16]	0	a7 [17]	0
s2 [18]	0	s3 [19]	0
s4 [20]	0	s5 [21]	0
s6 [22]	0	s7 [23]	0
s8 [24]	0	s9 [25]	0
s10 [26]	0	s11 [27]	0

```
            STACK SEGMENT
            FREE SPACE
0x000005fc: 00 00 00 00
0x000005f8: 00 00 00 00
0x000005f4: 00 00 00 00
0x000005f0: 00 00 00 00
0x000005ec: 00 00 00 00
0x000005e8: 00 00 00 00
0x000005e4: 00 00 00 00
0x000005e0: 00 00 00 00
0x000005dc: 00 00 00 00
0x000005d8: 00 00 00 00
0x000005d4: 00 00 00 00
0x000005d0: 00 00 00 00
0x000005cc: 00 00 00 00
0x000005c8: 00 00 00 00
0x000005c4: 00 00 00 00
0x000005c0: 00 00 00 00
0x000005bc: 00 00 00 00
0x000005b8: 00 00 00 00
0x000005b4: 00 00 00 00
0x000005b0: 00 00 00 00
0x000005ac: 00 00 00 00
```

```
********    Parser Output    ***********
Parsing successful!
```

31 20	19 15	14 12	11 7	6 0
imm	rs1	funct3	rd	opcode
000000000000	00000	000	00000	0010011

Figure 2.8: BRISC-V Assembly Simulator

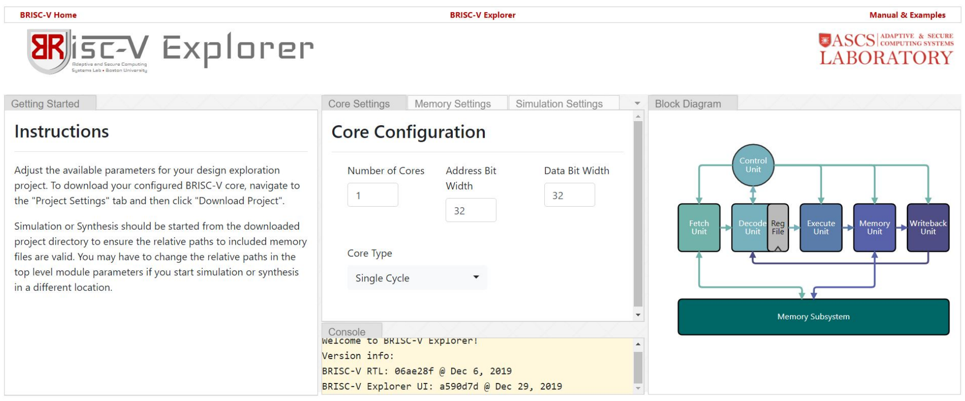
As seen in Figure 2.8, *BRISC-V* has a fairly simple and intuitive interface, with an area for RISC-V code assembly, where students can freely create and edit programs or load existing examples, as well as execute them step-by-step. Register and memory visualization is provided on the right side of the screen, with support for multiple value formats such as binary, hexadecimal and decimal. On the lower part of the interface, a console and instruction breakdown visualization are provided.

Similarly to several of the already presented educational platforms, *BRISC-V* is more focused on assembly debugging and does not include a display of the CPU datapath, as well as lacking support for a pipeline version of the CPU.

2.3.3 BRISC-V Explorer

The *BRISC-V Explorer* is an educational tool for exploring CPU design, allowing the creation of single or multi-core RISC-V processors. Students can learn and experiment with the various components of the CPU by freely configuring a hardware system, choosing from a selection of cores, caches, main memory and network-on-chip topologies, whose Verilog implementation can then be downloaded.

As seen in Figure 2.9, the platform contains multiple configuration panels that can be freely rearranged and resized within the interface. Users can load and save programs, select between a single cycle, five stage or seven stage pipelined core, as well as configure cache size, associativity and number of levels. Each hardware component is written in parametrized Verilog HDL modules,



Instructions

Adjust the available parameters for your design exploration project. To download your configured BRISC-V core, navigate to the "Project Settings" tab and then click "Download Project".

Simulation or Synthesis should be started from the downloaded project directory to ensure the relative paths to included memory files are valid. You may have to change the relative paths in the top level module parameters if you start simulation or synthesis in a different location.

Core Configuration

Number of Cores: 1
Address Bit Width: 32
Data Bit Width: 32
Core Type: Single Cycle

```
Welcome to BRISC-V Explorer!
Version info:
BRISC-V RTL: 06ae28f @ Dec 6, 2019
BRISC-V Explorer UI: a590d7d @ Dec 29, 2019
```

Figure 2.9: BRISC-V Explorer

whose parameters can be changed according to what users desire. Finally, a console panel can signal when a configuration is invalid. When the user finishes the processor design, a Verilog file with its configuration can be exported for future use.

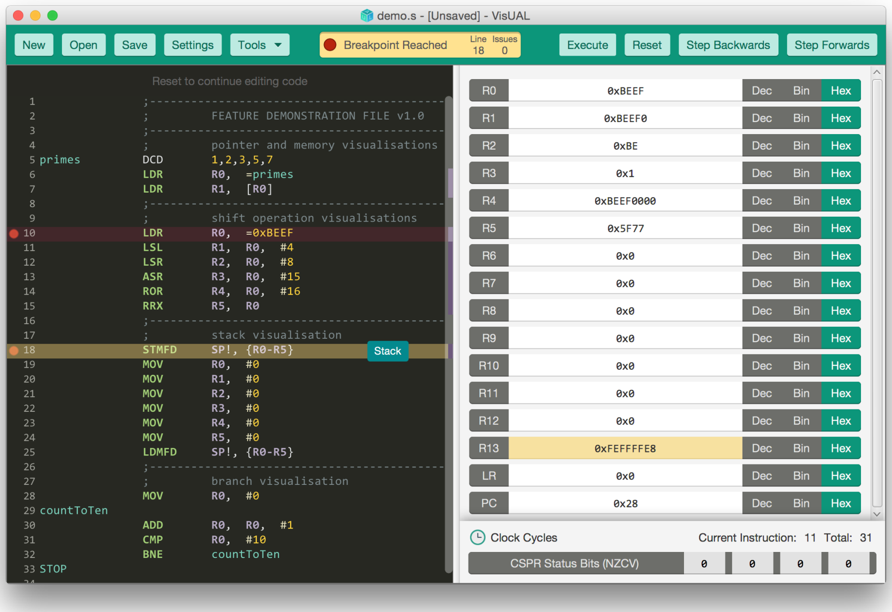
As the *BRISC-V Explorer* provides a platform for CPU architecture design, this can be helpful for students to understand its components on a deeper level, introducing more advanced concepts, such as cache configuration, in a simple manner. Since this tool is more focused on architectural design, it is better suited for more advanced CA classes [1]. Otherwise, having both the assembly simulator (*BRISC-V Simulator*) and CPU visualization running on the same space would be a preferable approach, similarly to other tools presented above, such as *DrMIPS*.

2.4 ARM

2.4.1 VisUAL

VisUAL is an ARM emulator developed as a cross-platform tool for ARM education, particularly ARMv7 [2]. Developed for the Imperial College London and tested by its students, it was designed to aid CA students in learning ARM assembly language. Although this tool is not as focused on the CPU and its internal behaviour, and more on assembly debugging, it includes multiple educational features that are useful for students and can be taken into consideration while designing a new tool.

Besides an assembly coding interface and display of registers and memory addresses, the emulator contains key visualization components of great value to students. Such is the case of pointer visualization, so the user can explore pointer behaviour when using a load/store instruction, and shift operation display, that can teach students how the shift works through an animation. The tool guides students, providing explanations and tips on using the interface and its features. This level of detail is crucial for an educational platform so students can easily explore and learn with it.



```
                ;----------------------------------------
                ;         FEATURE DEMONSTRATION FILE v1.0
                ;----------------------------------------
                ;         pointer and memory visualisations
primes          DCD       1,2,3,5,7
                LDR       R0,  =primes
                LDR       R1,  [R0]
                ;----------------------------------------
                ;         shift operation visualisations
                LDR       R0,  =0xBEEF
                LSL       R1,  R0,  #4
                LSR       R2,  R0,  #8
                ASR       R3,  R0,  #15
                ROR       R4,  R0,  #16
                RRX       R5,  R0
                ;----------------------------------------
                ;         stack visualisation
                STMFD     SP!, {R0-R5}
                MOV       R0,  #0
                MOV       R1,  #0
                MOV       R2,  #0
                MOV       R3,  #0
                MOV       R4,  #0
                MOV       R5,  #0
                LDMFD     SP!, {R0-R5}
                ;----------------------------------------
                ;         branch visualisation
                MOV       R0,  #0
countToTen
                ADD       R0,  R0,  #1
                CMP       R0,  #10
                BNE       countToTen
STOP
```

R0	0xBEEF
R1	0xBEEF0
R2	0xBE
R3	0x1
R4	0xBEEF0000
R5	0x5F77
R6	0x0
R7	0x0
R8	0x0
R9	0x0
R10	0x0
R11	0x0
R12	0x0
R13	0xFEFFFFE8
LR	0x0
PC	0x28

Current Instruction: 11 Total: 31

CSPR Status Bits (NZCV)	0	0	0	0

Figure 2.10: VisUAL - A highly visual ARM emulator

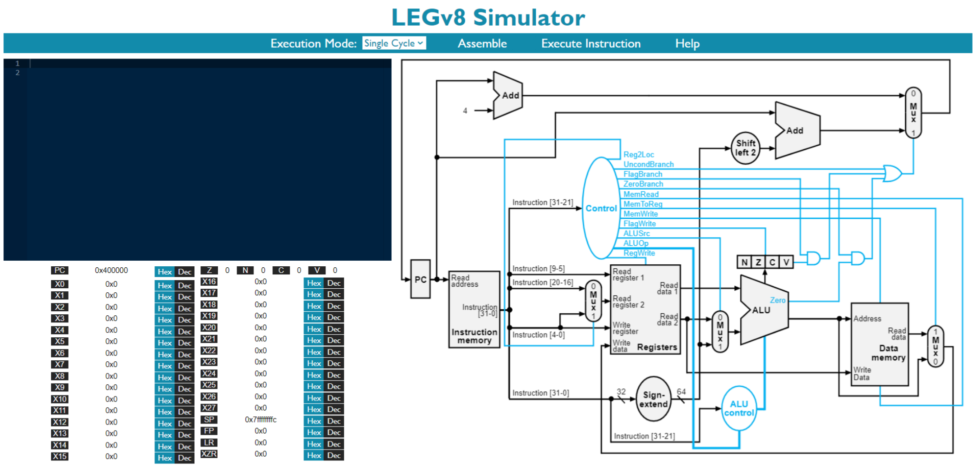
VisUAL acts as a valuable assembly code teaching tool, but its lack of emphasis on the CPU's components and its datapath hinders its adoption in CA courses with a focus on these subjects. Moreover, *VisUAL* only supports ARMv7, which was already replaced by ARMv8 in institutions such as FEUP [16], not being suitable for aiding those students any longer. Still, its educational qualities can be taken into consideration in the development of an adequate tool.

2.4.2 Graphical Micro-Architecture Simulator

During the development of this thesis, a new tool developed by ARM was discovered. The *Graphical Micro-Architecture Simulator*, also called simply *LEGv8 Simulator*, is a web based ARMv8-A ISA simulator, designed for education, with its first release on December 2021 [3]. This tool, albeit still in BETA version, delivers a complete and interactive environment for CPU visualization, both its unicycle and pipelined versions, whose datapaths are based on the Computer Organization and Design (Arm Edition) textbook by David A. Patterson and John L. Hennessy [17], widely used in CA courses.

Regarding its features, the *LEGv8 Simulator* provides a code editor, with support for multiple LEGv8 instructions, along with a register file, whose contents change with each execution. Both the single cycle and pipelined versions of the CPU are provided, and their datapaths are displayed on the right side of the screen. Each time an instruction is executed, its relevant components are highlighted red, which can help the user understand the flow of the program. For the pipelined CPU, specifically, hazards are correctly identified and solved, with a CPU log displaying the pipeline stages and when bubbles occur.

Despite being a very complete platform, this simulator, as a possible result of still being in its BETA version, lacks certain important features for education, such as a data memory display, so students can understand what happens in memory when load or store instructions are executed, visualization of input and output values in each CPU component - whose absence hinders the user's comprehension of what is happening inside each part of the processor - and a more complete register file, capable of displaying register values in the binary format, as well as allowing its values to be altered directly, and not just with assembly code. Component latencies and the critical path are also not included in the platform.



PC	0x400000	Hex Dec	Z 0 N 0 C 0 V 0		
X0	0x0	Hex Dec	X16	0x0	Hex Dec
X1	0x0	Hex Dec	X17	0x0	Hex Dec
X2	0x0	Hex Dec	X18	0x0	Hex Dec
X3	0x0	Hex Dec	X19	0x0	Hex Dec
X4	0x0	Hex Dec	X20	0x0	Hex Dec
X5	0x0	Hex Dec	X21	0x0	Hex Dec
X6	0x0	Hex Dec	X22	0x0	Hex Dec
X7	0x0	Hex Dec	X23	0x0	Hex Dec
X8	0x0	Hex Dec	X24	0x0	Hex Dec
X9	0x0	Hex Dec	X25	0x0	Hex Dec
X10	0x0	Hex Dec	X26	0x0	Hex Dec
X11	0x0	Hex Dec	X27	0x0	Hex Dec
X12	0x0	Hex Dec	SP	0x7ffffffffc	Hex Dec
X13	0x0	Hex Dec	FP	0x0	Hex Dec
X14	0x0	Hex Dec	LR	0x0	Hex Dec
X15	0x0	Hex Dec	XZR	0x0	Hex Dec

Figure 2.11: Graphical Micro-Architecture Simulator - LEGv8 Visual Simulator

2.5 Support for multiple architectures

2.5.1 WepSIM

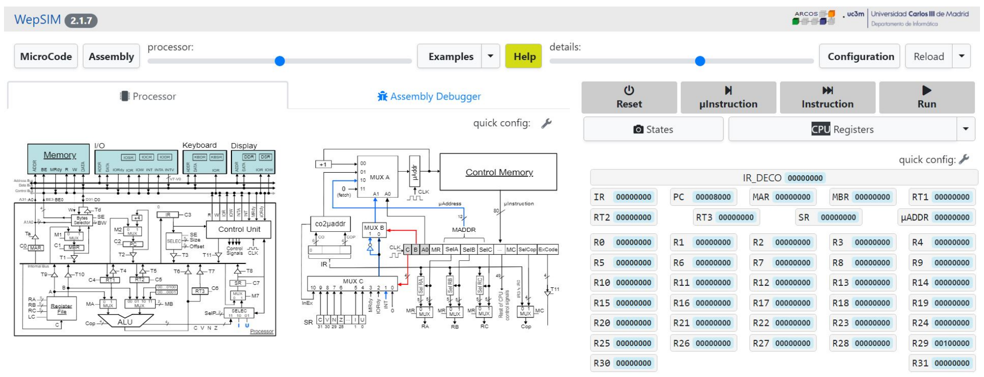
WepSIM is a modular and intuitive online educational simulator of the CPU, supporting both MIPS and RISC-V architectures [9]. It was developed by the Informatics Department at the University Carlos III of Madrid and tested by students of its Bachelor's Degree in Computer Science and Engineering.

The platform was conceived specifically for education, with the aim of being used not only by students, so they can better comprehend the topics lectured in CA classes, but also by professors, who can use this tool to teach complex concepts in a more intuitive way, encouraging learning through hands-on experimentation. It can be used to teach, for example, microprogramming and how the CPU and its components work, firmware (through microcode) for assembly, and how a set of instructions interacts both with the hardware and the operating system.

Being available on the web, the platform is highly portable and can be run from any web browser, making it accessible to everyone with Internet access and a desktop or laptop computer, smartphone or tablet, or from a text-based command line. Because of this, students can use the platform almost anywhere they want and learn on the go.

Its flexible interface allows users to quickly switch from one view to another (assembly, microcode and processor) and display information in multiple value formats. *WepSIM* is also highly interactive and configurable, as students can interact with the datapath and experience what would happen if they were to change a specific value. Instruction sets are highly customizable and new ones can be created. Performance metrics are also provided, such as the number of instructions executed and clock ticks.

Additionally, the platform provides students with already built examples on several topics approached in CA classes, such as interrupts, exceptions, system calls and threads. With this functionality, students can simply load them and learn alone through freely experimenting with the examples.



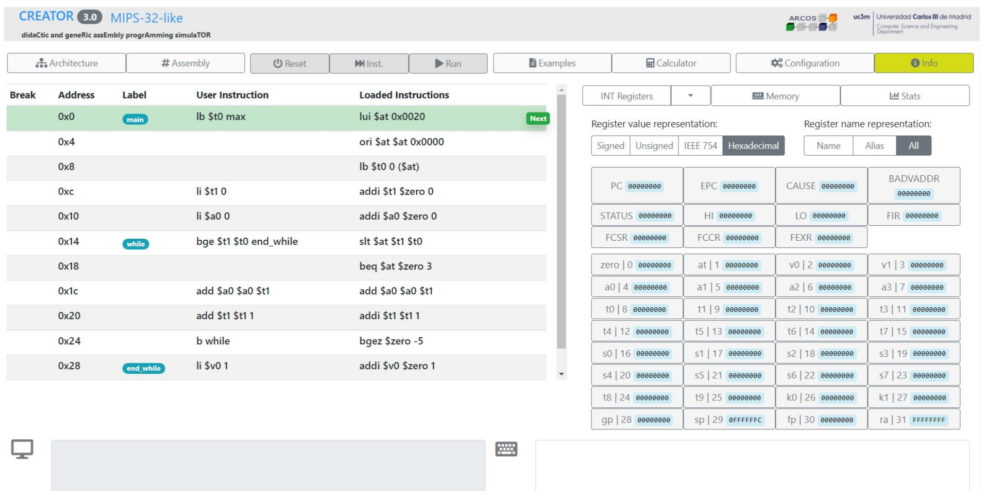
```
IR_DECO 00000000
IR 00000000   PC 00008000   MAR 00000000   MBR 00000000   RT1 00000000
RT2 00000000   RT3 00000000   SR 00000000   µADDR 00000000
R0 00000000   R1 00000000   R2 00000000   R3 00000000   R4 00000000
R5 00000000   R6 00000000   R7 00000000   R8 00000000   R9 00000000
R10 00000000  R11 00000000  R12 00000000  R13 00000000  R14 00000000
R15 00000000  R16 00000000  R17 00000000  R18 00000000  R19 00000000
R20 00000000  R21 00000000  R22 00000000  R23 00000000  R24 00000000
R25 00000000  R26 00000000  R27 00000000  R28 00000000  R29 00100000
R30 00000000                                            R31 00000000
```

Figure 2.12: WepSIM - The Web Elemental Processor SIMulator

WepSIM, albeit a complete platform, can be too dense or complex, possibly making it difficult for students to get familiar with the interface. Moreover, its microcode interface provides an additional complexity for the configuration that might not be necessary for every course unit. Finally, there is no support for a pipeline version of the CPU nor implementation of latencies and the critical path, which are crucial concepts in understanding the processor and how it functions.

2.5.2 CREATOR

CREATOR is a generic web-based simulator for assembly programming developed by the Computer Science and Engineering Department at the University Carlos III of Madrid, by the same authors as *WepSIM* [7]. Both these platforms are extremely similar, but *CREATOR* is solely focused on assembly programming, with a newer interface, as seen in Figure 2.13.


CREATOR 3.0 MIPS-32-like
didaCtic and geneRic assEmbly progrAmming simulaTOR
ARCOS
uc3m Universidad Carlos III de Madrid
Computer Science and Engineering Department
Architecture
# Assembly
Reset
Inst.
Run
Examples
Calculator
Configuration
Info
INT Registers
Memory
Stats
Next

Break	Address	Label	User Instruction	Loaded Instructions
	0x0	main	lb $t0 max	lui $at 0x0020
	0x4			ori $at $at 0x0000
	0x8			lb $t0 0 ($at)
	0xc		li $t1 0	addi $t1 $zero 0
	0x10		li $a0 0	addi $a0 $zero 0
	0x14	while	bge $t1 $t0 end_while	slt $at $t1 $t0
	0x18			beq $at $zero 3
	0x1c		add $a0 $a0 $t1	add $a0 $a0 $t1
	0x20		add $t1 $t1 1	addi $t1 $t1 1
	0x24		b while	bgez $zero -5
	0x28	end_while	li $v0 1	addi $v0 $zero 1

Register value representation: Signed Unsigned IEEE 754 [✓ Hexadecimal]

Register name representation: Name Alias [✓ All]

PC 00000000	EPC 00000000	CAUSE 00000000	BADVADDR 00000000
STATUS 00000000	HI 00000000	LO 00000000	FIR 00000000
FCSR 00000000	FCCR 00000000	FEXR 00000000	
zero | 0 00000000	at | 1 00000000	v0 | 2 00000000	v1 | 3 00000000
a0 | 4 00000000	a1 | 5 00000000	a2 | 6 00000000	a3 | 7 00000000
t0 | 8 00000000	t1 | 9 00000000	t2 | 10 00000000	t3 | 11 00000000
t4 | 12 00000000	t5 | 13 00000000	t6 | 14 00000000	t7 | 15 00000000
s0 | 16 00000000	s1 | 17 00000000	s2 | 18 00000000	s3 | 19 00000000
s4 | 20 00000000	s5 | 21 00000000	s6 | 22 00000000	s7 | 23 00000000
t8 | 24 00000000	t9 | 25 00000000	k0 | 26 00000000	k1 | 27 00000000
gp | 28 00000000	sp | 29 0FFFFFFC	fp | 30 00000000	ra | 31 FFFFFFFF

Figure 2.13: CREATOR - Generic assembly programming simulator

Similarly to *WepSIM*, this platform includes already built-in support for MIPS and RISC-V, with additional support for other architectures, as the user can freely edit and define other instruction sets as desired. A set of common examples taught in CA classes are provided, as well as a floating point calculator.

Unlike its older counterpart, *CREATOR* does not include a display of the CPU datapath, as it is a tool more focused on assembly programming instead of CPU visualization and configuration. Nevertheless, its light and intuitive interface, with several improvements over *WepSIM*, can be a great influence for a new platform to develop.

2.6 Discussion

Although, as seen in the previous sections, various CA simulators exist in the field, only some of them are focused on the CPU and its behaviour, and, within that group, no adequate platforms for studying the ARMv8 architecture were discovered. Educational tools that do exist are usually dedicated to other common architectures such as MIPS and RISC-V, thus not matching the needs of students whose institutions adopted ARM.

In the previous sections, a set of relevant projects were presented, analysing their strengths and weaknesses. Despite not entirely solving the problem identified previously, these tools contain important aspects that can be taken into consideration and used as a starting point for the development of a new and updated platform.

In order to efficiently analyse these existing tools and what the new platform should include, a few key requirements were established:

- Available on the web
- CPU datapath visualization

- Assembly code programming
- Both unicycle and pipeline CPU versions
- Latencies and critical path
- Support for ARMv8

To evaluate whether these tools would meet all the requirements established for a complete and suitable platform to teach the ARM architecture, table 2.1 is provided.

Table 2.1: Comparison of requirements met by the tools

	Web	**CPU datapath**	**Assembly**	**Unicycle & pipeline**	**Latency & critical path**	**ARMv8**
SPIM	No	No	Yes	No	No	No
WebMIPS	Yes	Yes	Yes	No	No	No
MARS	No	No	Yes	No	No	No
DrMIPS	No	Yes	Yes	Yes	Yes	No
RARS	No	No	Yes	No	No	No
BRISC-V	Yes	Yes	Yes	Yes	No	No
VisUAL	No	No	Yes	No	No	No
LEGv8Sim	Yes	Yes	Yes	Yes	No	Yes
WepSIM	Yes	Yes	Yes	No	Yes	No
CREATOR	Yes	No	Yes	No	Yes	No

Through the analysis of the table, it is clear that, even though each tool has its strengths and useful features already discussed in the previous sections, none can satisfy all the requisites proposed. More specifically, even though all of them can teach assembly programming, only a few of the works presented also focus on the CPU and its datapath. Most of them lack support for both the unicycle and pipeline versions of the processor, as well as latency and critical path implementation.

Out of all of them, *DrMIPS* and *LEGv8 Simulator* stand out from the others. *DrMIPS*, on the one hand, seems to be the most complete one, especially regarding the CPU datapath and what it can showcase to the user. On the other hand, it is lacking in not being web-based and not supporting ARMv8, the latter being a requirement only the *LEGv8 Simulator* could meet.

LEGv8 Simulator, compared with *DrMIPS*, presents the advantage of being a web-based platform and tailored for the ARMv8 architecture, but fails to match the level of detail of *DrMIPS*' CPU visualization. Besides having multiple detailed datapaths for both unicycle and pipelined versions of the CPU, the MIPS platform contains descriptions, inputs, and outputs for each of the processor's components, which greatly supports the comprehension of the flow of a program. This important feature is lacking in the *LEGv8 Simulator*, along with data memory representation, a way to change the register file directly, latencies, and critical path implementation.

Both these tools focus on education and have CPU datapath visualization, an assembly programming interface, and support for both unicycle and pipeline processors. Despite having their advantages and disadvantages, these are both valuable tools whose good practices can be adopted in a new solution.

Additionally, *WepSIM* is web-based, complete, and contains a highly configurable and intuitive interface, making it a good example to follow in terms of user interface design.

2.7 Concluding Remarks

Seeing as none of the works presented can completely respond to the problem identified, even more so with most of them lacking support for ARMv8, a platform that can aggregate the positive features of these tools and provide what they are lacking would be the ideal solution. It would also present a source of innovation in the field, which is what this thesis aims to achieve, and whose methodology is thoroughly explained in the next chapter.

Chapter 3

Methodology

In this chapter, the methodology behind the work developed in this thesis is described in detail, beginning with the requirements specification for the developed platform, following with an overview of the development process, and concluding with the tool's architecture and technologies.

3.1 Requirements Specification

The development of this educational platform must take several CA concepts into account, handling the multiple topics students learn in classes. After analysing the state of the art and the functionalities of existing educational tools, the requirements for a new platform were specified as follows:

1. Must be simple and intuitive, so students can easily interact with the platform and understand its concepts, avoiding unnecessary elements or visual clutter
2. Represent the CPU datapath in both its unicycle and pipelined versions, which the user can interchange accordingly
3. Provide an interface for assembly coding, so students can write their own instructions and explore what happens in the CPU with their execution, showing detailed information on all data and control signals, as well as display how these instructions are encoded in machine code
4. Execute a set of instructions provided step-by-step, allowing the user to forward or go back to a certain instruction and understand what its effects on the CPU are
5. Allow visualization of registers and data memory contents, as well as latencies and the CPU critical path

6. Be available on the Web, hence providing easy access to anyone, regardless of their operating system or computer specifications

Considering the state of the art analysis in Chapter 2, and the conclusion that none of the tools found can meet all requirements described above, a suitable platform should take these tools' good practices into account, while simultaneously providing the functionalities that were missing. Delivering a tool focused on ARMv8 that is able to run on the web, making it more accessible, and incorporating key concepts of CA that were not always present in the state of the art (such as unicycle and pipelined processors), aspires to both act as a source of innovation in the field and provide students with a robust and thorough studying tool.

Thus, and after their proper identification, each of the requirements above was taken into consideration while planning for the platform's development, ensuring that the finished product would accomplish all objectives and adequately respond to CA students' needs.

3.2 Development Process

After the platform's requirements specification, the first step of development was to ensure it would run on the web. A simple web app was created, and, within it, two distinct layers were conceived - the platform's frontend, which would encapsulate all user interface-related work and client-side operations, and its backend, which handles the server and all CPU simulation logic. These two segments would eventually connect, allowing the web platform to run in its entirety, with each request sent by a user's actions being received and handled server-side. The operations managed by each of these two domains are further described in Chapters 4 and 5.

3.2.1 Backend development

Having established the foundation for both layers, the program's backend was the first to be thoroughly implemented. This decision was based on the fact that the CPU simulation logic is the most significant part of the project, and, although a complete and intuitive user interface is crucial, the simulator must be working in its entirety for the platform to truly fulfill its objectives. Thus, implementation started with the simulation logic of the unicycle CPU version, accompanied by frequent testing to ensure every component was behaving as expected. The simulation's development is described in detail in Chapter 4.

3.2.2 Frontend development

After completion of the unicycle's backend logic, its frontend counterpart followed, with the creation of the unicycle CPU datapath, which would provide a visual representation of the CPU components previously defined, along with the connections between them. An area for assembly code was also incorporated in this stage, which makes coding a set of ARMv8 instructions possible, along with the register file that collects values inserted by the user.

By connecting both the simulation logic and the user interface, the platform is able to execute simple assembly programs and provide the corresponding datapath visualization. This includes the results of the assembly operations on the registers and the data memory, as well as the input and output values of each CPU component in that state. The user's interface features and development process is documented in Chapter 5.

3.2.3 Adding a pipeline CPU version

As for the pipeline version of the processor, the same method was applied. In this case, the backend - more specifically, the logic of each CPU component - remains essentially the same, with a few exceptions. Likewise, the user interface remains mostly unaltered, save for the pipelined datapath, which differs from the unicycle one.

The decision to first implement the entirety of the unicycle CPU, with both its logic and user interface, proved to be of great advantage. By doing this, when the time came to implement the pipeline version, only a few adjustments needed to be done, as most was already setup from the simpler one. This bought precious time to focus on fixing eventual bugs and ensuring the program was sturdy enough to be used by students.

3.2.4 Deployment and validation

After wrapping up its development phase, the platform, in order to be tested, had to be made available on the web for students to access freely. In order to achieve this, a virtual machine was used for its deployment, and the website became available through access to FEUP's VPN, at `eduarm.fe.up.pt:3000`.

Having setup the environment for testing, the solution was validated with the help of former CA students from the Informatics and Electrical Engineering courses at FEUP. This step was crucial in order to understand the students' perceptions of the course and how having access to this web tool would have helped in their studies and overall academic success. This validation is further explored in Chapter 6.

3.3 Architecture and Technologies

3.3.1 System architecture

Regarding the platform's general architecture, it essentially covers two main layers: the presentation layer, commonly referred to as frontend, and the data access layer, the backend.

The frontend handles everything client-side and represents the interface - the platform's website - that the user interacts with. Each user action that requires operating on the server (for example, assembling an instruction) generates an HTTP request. Once received, the backend, which covers all server and CPU simulation logic, will handle this request, proceed with the corresponding operations, and send the necessary data back to the frontend.

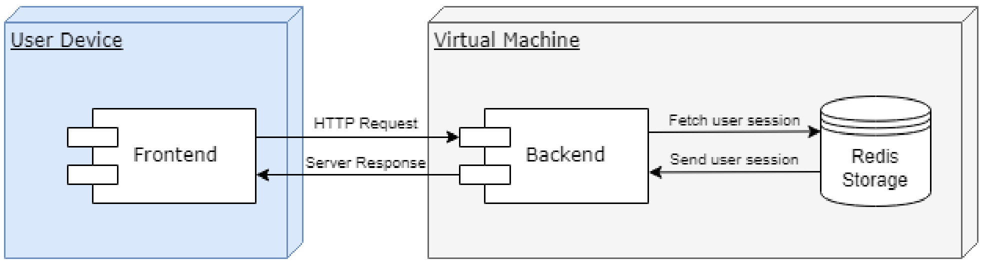


Figure 3.1: eduARM Architecture Diagram

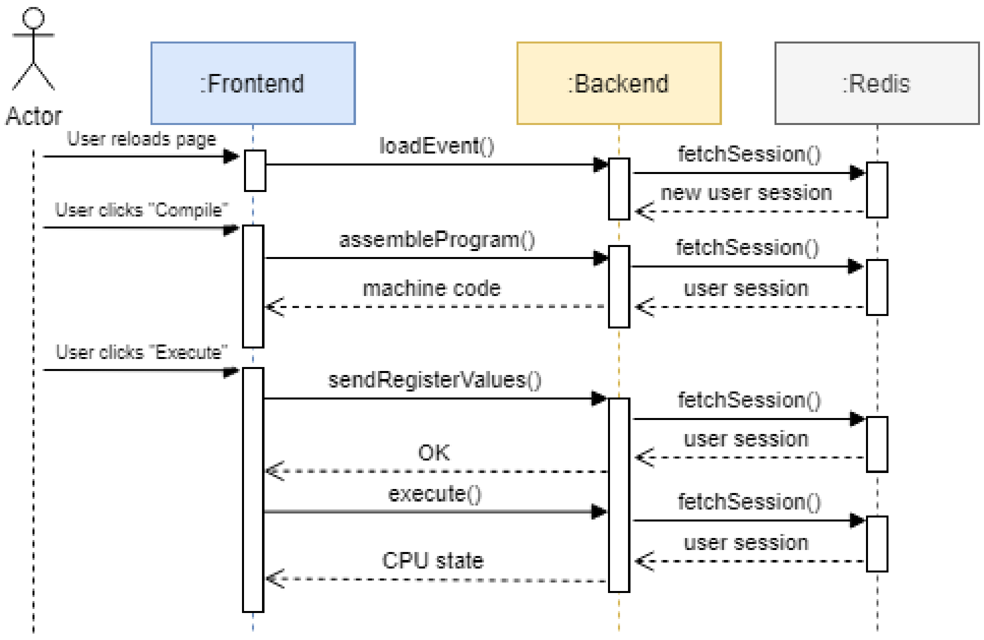


Figure 3.2: Sequence Diagram of the execution of a program

Taking into consideration that the platform must support multiple accesses at the same time, a data store had to be implemented into the backend. Each user, when first visiting the website, generates a user session that carries all information about the current condition of the program (such as CPU states, memory and register values), which will be kept in a Redis [20] data store. Using a session middleware ensures that multiple users can explore the platform at the same time, which avoids overlapping backend states when executing programs concurrently.

In the case of the previous example of assembling an instruction, the user writes a LEGv8 instruction and expects it to be converted to machine code. When this happens, the server will receive the instruction and do the required operations to convert it into binary, sending the result back to the client, which will display it on its designated area on the website. Additionally, in between each request to the server, the user's session is fetched from the Redis data store. If the user proceeds with the execution of the program, the current state of the register file is sent to the backend, and, when an acknowledgement is received, a request for execution is sent. After the simulation executes the necessary operations, the backend sends back the generated CPU state, which is displayed on the interface for the user to explore, in the form of a datapath. This process is illustrated in the sequence diagram of figure 3.2.

3.3.2 Choosing the appropriate technologies

Choosing the appropriate technologies is crucial for any software program, as making a wrong decision can greatly hinder the solution's final result. In the case of a web application, selecting the correct tech stack means making sure it offers sturdiness and scalability, with good support and documentation, favoring those considered the industry standard. Besides this, already being familiarized with the technology helps immensely in saving time and producing better quality software.

For this project, the main programming language chosen for development was JavaScript. Considering that the platform runs on the web, JavaScript is a logical choice, as it is widely used in the majority of websites. This client-sided language can be used in both the frontend and backend layers, is fast and simple to handle, and is supported by a multitude of powerful libraries that make the development of dynamic and interactive applications much easier.

Such is the case of React [19], chosen for this project, which is a component-based JavaScript frontend library, intuitive and easy to work with, and excellent for creating user interfaces. Using such a library instead of the traditional combination of HTML, CSS and JavaScript provides standard functionality to the application, saving time and effort for more challenging tasks, such as the CPU simulation in this particular project. Besides this, numerous useful packages for React are created by developers and updated regularly, and can be used to enhance the interface's overall quality.

The use of React was coupled with Bootstrap [5], a frontend toolkit able to greatly facilitate the development and customization of a responsive website. This framework contains multiple reusable components for the user interface, such as buttons and navigation bars, and its own grid system that can make the website adjustable to multiple screen sizes. Overall, Bootstrap helps in creating an interface that is more user friendly and easier to navigate.

Regarding the backend layer, Node.js [13], an open-source JavaScript runtime environment, was used. This platform makes it possible to run server-sided operations and generate dynamic page content, without the need of incorporating another programming language. Using JavaScript in both frontend and backend makes the application more consistent and easier to work with.

Finally, to connect the two layers, Node.js framework Express [8] was used in combination with Axios [4], a promise-based HTTP client. The former is used to create an API that listens and responds to HTTP requests, which Axios sends on the frontend layer. Express is also used in conjunction with a session middleware to create user sessions and save them in the Redis data store, as described in the subsection above.

Each of these packages were incorporated into the program using npm [15], a package manager for JavaScript projects.

3.4 Concluding Remarks

In this chapter, the requirements and general methodology for this thesis' project were described. Outlining and following a work plan that ensures every requirement is met was decisive for the project's success. Its development phase followed a steady pace and was backed by frequent testing, working towards a final solution of the best possible quality. While the present chapter merely provided an overview, the following chapters go further in detail on the methodology behind this project, starting with its backend on Chapter 4 and following with the frontend at Chapter 5.

Chapter 4

Backend and Simulation Logic

This chapter details the simulation logic of both the unicycle and pipelined CPUs, along with the backend server logic and how it communicates with the frontend. The chapter's content is laid out in a way that simulates the actual development process of the program's backend and in what order each problem was addressed, starting with the complete implementation of the simpler unicycle CPU and the addition of a server, finalizing with its pipelined version, which only required a few key changes compared to the unicycle.

4.1 Unicycle CPU Version

The unicycle version, already briefly described in Chapter 2, constitutes a fairly simple implementation of the processor. This version's program flow is linear and easy to understand, with each instruction in a program taking exactly one clock cycle to complete. Specifically, with each instruction executed, the state of the processor changes with it, which means there will not be two instructions being handled by the datapath simultaneously, which was the case for a pipeline implementation.

That being said, simulating the unicycle CPU is rather straightforward. For it to function correctly, the implementation must include at least a datapath with control, a register file, data memory and an assembler. Besides the essentials, this program must take the requirements set in Chapter 3 into account, adding support for latencies and the critical path.

4.1.1 Supported Instructions

eduARM implements a total of eight assembly instructions, part of a small subset of the AArch64 ISA called LEGv8. The selected instructions are considered sufficient to demonstrate how the architecture functions, and easy enough for students to understand. This subset is showcased in Table 4.1.

Table 4.1: LEGv8 instruction set used in this project

Instruction type	**Instruction**	**Name**	Meaning
R-type	ADD Rd, Rn, Rm	Add	Rd←Rn + Rm
	SUB Rd, Rn, Rm	Subtract	Rd←Rn - Rm
	AND Rd, Rn, Rm	And	Rd←Rn&Rm
	ORR Rd, Rn, Rm	Or	Rd←Rn | Rm
Memory access	STUR Rt, [Rn, address]	Store register	Memory[Rn + address] = Rt
	LDUR Rt, [Rn, address]	Load register	Rt = Memory[Rn + address]
Branch	CBZ Rt, label	Compare and branch on 0	if (Rt == 0) PC = label
	B label	Unconditional branch	PC = label

Upon execution of a program, each of these instructions is encoded and interpreted by the processor in a different way, according to their formats. As highlighted in Table 4.1, each instruction type has multiple fields with varying numbers of bits. The opcode is always present and determines the instruction's operation and overall format. The other fields can have different meanings depending on their instruction type:

- R-type instructions (ADD, SUB, AND, ORR): These have three register operands: Rn, Rm, and Rd, where Rn and Rm are sources and Rd is the result destination. The shamt field is used only for shifts and is thus considered irrelevant for this implementation.
- Memory access instructions (LDUR AND STUR): register operand Rn is the base register that is added to the 9-bit address field, forming the memory address. For a load instruction, whose opcode is 1986, Rt is the destination register for the loaded value. For a store, whose opcode is 1984, Rt is the source register whose value should be stored in memory.
- Branch instructions (CBZ and B): the conditional branch instruction has an opcode of 180. Rt is the source register that will be tested for zero, and the 19-bit address field is used to compute the branch target address. As for the unconditional branch, Rt no longer exists and its offset is 26-bits wide.

Understanding which different fields constitute an instruction's 32-bit code and their meaning for each instruction type is essential for the overall comprehension of the CPU behaviour, as these fields are used by various components' for their own calculations, as will be seen throughout this chapter.

4.1.2 Understanding the execution flow

The unicycle datapath, as seen in Figure 2.1 of Chapter 2, contains several components, each with its characteristics and behaviour.

The main program flow of the unicycle CPU is, in summary, explained below. The processor's behaviour for each supported instruction type is mostly the same, with memory access and jump actions showing some slight differences.

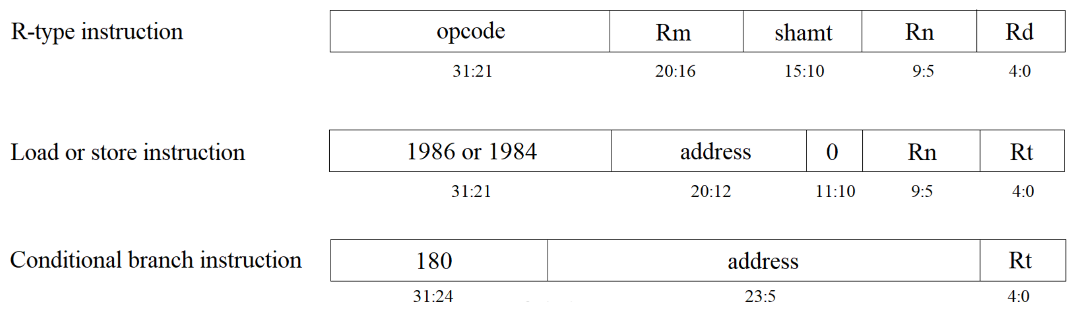


Figure 4.1: Types of instructions and their formats

- On the instruction memory component, the instruction to execute is fetched from the memory position of the value of the program counter. More specifically, if, for example, the PC value equals 8, the second instruction will be fetched, as PC values increment by 4 bytes (32 bits) each clock cycle;
- The fetched instruction value is sent both to the control unit and the distributor, which will then send the appropriate bits to the register bank and the sign extend component;
- The control unit breaks down the instruction code into the adequate control signals. These control outputs will later influence the behaviour of the components they connect to;
- The register bank reads the values of the registers involved in the operation, sending the results into the ALU;
- The ALU executed the appropriate operation. For example, if the instruction is an "add", it merely adds the two numbers;
- The result of the ALU operation is then sent to the data memory component, which will send the value back to the register bank;
- The register bank, to finalize, will alter the register file to include the new register value;
- Meanwhile, the PC value is again incremented by 4 to prepare for the next instruction.

The steps described above illustrate what essentially happens in an arithmetic-logical instruction. In the case of a memory access instruction, the ALU result will be used as the address in the data memory component. For a store instruction, the desired value will be saved in this memory address. In case of a load, the value will be read from that address and sent to the register file.

If the instruction to execute is a branch instruction, the program logic is mostly set on the upper side of the datapath. A new possible PC value, corresponding to the instruction to jump to, is calculated. If the instruction is an unconditional branch, in the next clock cycle, the PC value will change into that sum and the instruction will be fetched from memory in that position. In the case of a conditional branch, it will only perform this change if the value of the register equals zero.

4.1.3 CPU Components

The flow of the processor, as described in the previous subsection, highlights how every component has its own behaviour while executing a program. To accurately simulate the CPU, each component must first be simulated individually.

The approach used to simulate a component was to implement a JavaScript class, specifically a generic class named "Component". This class stores a component's list of input and output objects, its identifier and associated JSON file, and its latency.

The different component types will then extend this class, using its methods to store their respective inputs and outputs. Each component class has, besides the characteristics of the base "Component" class, its own distinct execution behaviour. A generic component's structure is illustrated below.

```
class GenericComponent extends Component {
  constructor(id, json) {
    //Assigns an ID and a JSON file to the component
    super(id, json);

    //GenericComponent input and output variables
    this.input1 = super.addInput(json.input, new Data(0, 0));
    this.output1 = super.addInput(json.output[0], new Data(0, 4));
    this.output2 = super.addOutput(json.output[1], new Data(0, 0));

    //Assigns a latency value to the component
    super.latency = json.latency;
  }
  //Execute the component's operations
  execute() {}

  //Log the component's values into the console
  printValues() {}
}
```

Listing 4.1: Example class for a component

As seen in the provided code example, each component has its own constructor, with an ID, which is the name used to identify it, and variables for inputs and outputs, that will logically differ from one component to another.

A component's inputs and outputs are also classes of their own. Each input or output object created in the "Component" class will have its own ID, along with an object to store its value, called “Data”. This is another JavaScript class that stores a value and a size, in bits.

The "Input" and "Output" classes function in mostly the same way, storing the input/output identifier, the "Data" object mentioned before, and the identifier of the component it relates to. Besides this, the "Output" class also contains a method "createConnection" to connect an output to an input.

Besides its inputs and outputs, each component has a specific "execute" method, which will run when that component is scheduled to operate. The execution logic for each component type is showcased in table 4.2.

Table 4.2: Table with the inputs, outputs, and execution behaviour of each component type

Component type	**Inputs**	**Outputs**	**Behaviour**
Adder	value1, value2	result	Adds the two inputs and stores the result in its output
ALU	input1, input2, controlOp	zero, aluResult	Depending on the "controlOp" value received, performs ADD, SUB, AND or OR on its inputs, storing the result in "aluResult". If the result is 0, output "zero" will take the value 1.
ALU Control	ALUOp0, ALUOp1, opcode	ctrlALU	Assigns a "ctrlALU" value, used to define the ALU's operation, depending on "ALUOp0", "ALUOp1" and "opcode" values. Follows the logic of Table 4.4.
And	value1, value2	result	Performs a logical AND between the two inputs, stores the result in its output
Control	controlInput	signalsArray	Depending on the value of the input's opcode field, assigns either 0 or 1 to each of the outputs, storing them in an array. Each instruction type's corresponding control signals are shown in Table 4.3.
Data Memory	address, writeData, memRead, memWrite	readData	For load operations, "memRead" is 1 and "readData" will be the value of memory position "address"/8 (each memory position occupies 8 bits). For a store, "memWrite" is 1, and "writeData" is written on memory position "address"/8 as well.
Distributor	input	outputArray	Takes the 32-bit value of "input" and divides it into sets of bits, depending on the type of instruction. The different bits are stored in an array and sent to different components.
Fork	input	output	Equals the output value to the received input value
Instruction Memory	address	instruction	Fetches the instruction on position "address"/4. This input is the PC value, so dividing it by 4 will return the memory position of the instruction that should be fetched.
Multiplexor	zero, one, selector	muxOut	If value of "selector" is true (1), "muxOut" equals the value of input "one". If "selector" is false (0), "muxOut" equals the value of "zero".
Or	value1, value2	result	Performs a logical OR between the two inputs, stores the result in its output
Program Counter	pcValue	updatedPC	Updates the output value with either pcValue + 4, or another number besides 4 if the instruction is a jump
Register Bank	readReg1, readReg2, writeReg, writeData, regWrite	readData1, readData2	"readData1" will take the value of register number "readReg1", doing the same for "readData2" and "readReg2". If "regWrite" is set to 1, the value of register number "writeReg" will take the value "writeData".
Shift Left	shiftInput	shiftOutput	Shifts "shiftInput" two bits to the left, storing the result in "shiftOutput".
Sign Extend	signExtendIn	signExtendOut	Does a 64-bit sign extend operation on the "signExtendIn", storing the result on "signExtendOut", maintaining the number's sign and value.

Having set the behaviour for each CPU component, the next step in development was to figure out how to assign the correct identifiers, inputs and outputs to these components. To make this possible, a JSON file with the unicycle CPU configuration was used. This file has an object for “cpuComponents” that encapsulates all the processor’s components.

In the code snippet below, one generic component of the JSON object is shown. The GenericComponentID (which is the component’s identifier) has its own associated component type "GenericComponent", followed by the names of their input and two outputs, and, finally, its latency value.

```
{
 "cpuComponents": {
   "GenericComponentID1": {
      "componentType": "GenericComponent",
      "input": "input1",
      "output": ["output1", "output2"],
      "latency": 50
   },
 ...
```

Listing 4.2: JSON object representing a CPU component

As for the value of each input and output, they are transferred from one component to another by buses. In order to simulate these connections, another object on the JSON file was added, called "cpuConnections".

```
 ...
  "cpuConnections": [
    {
     "origin": "GenericComponentID1", "dest": "GenericComponentID1",
     "output": "output1", "input": "input2",
     "rType": "true",
     "loadType": "true", "storeType": "true",
     "cBranchType": "true", "uncondBranchType": "true"
    },
  ]
}
```

Listing 4.3: JSON object representing a CPU connection

Each object is composed of IDs of their origin and destination components, the input (of the destination component) and output (of the origin component) that are being connected, and five boolean fields that indicate whether that specific connection is relevant (meaning it carries relevant data) or not for a certain instruction type.

The relevant lines of an instruction will be highlighted in the frontend, in order for the user to understand which buses carry relevant information in that execution. In this implementation, these lines are considered static for each instruction type, as they remain the same unless, for example, the user switches from a load to a store instruction.

4.1.4 Control Unit

A special case within the components of the CPU is the control unit. Given an instruction code, this element breaks it down into a total of 10 control signals, with 1 bit each, whose values depend on the type of instruction, as seen in Table 4.3.

The implementation method of the control component follows the same logic of reading a JSON file explained in the previous section. In this case, besides the JSON configuration file that assigns a type, inputs, outputs and latency, another JSON file was used to interpret the instruction code that reaches the control unit.

This file includes two JSON objects. The first, "controlUnit", matches the opcode field of the instruction code with its corresponding control signals, while "ALUControl" matches the value of the ALUOp signals (and opcode, if these carry the same value) to a CtrlALU value. CtrlALU is the name given for the control input of the ALU, which will later dictate what type of operation that component should make.

```
{
  "controlUnit": {
    "1984": {
      "reg2Loc": 1, "ALUSrc": 1, "memToReg": 0,
      "regWrite": 0, "memRead": 0, "memWrite": 1,
      "uncondBranch": 0, "branch": 0, "ALUOp1": 0, "ALUOp0": 0
      },
        ...
    },
    "ALUcontrol": {
      "control": [
             {"ALUOp0": 0, "ALUOp1": 0, "CtrlALU": 2},
             {"ALUOp0": 0, "ALUOp1": 1, "opcode": 1112, "CtrlALU": 2},
             {"ALUOp0": 0, "ALUOp1": 1, "opcode": 1624, "CtrlALU": 6},
             ...
             ]
    ...
```

Listing 4.4: JSON object defining control values

Considering an instruction's opcode field equals 1984, which is the case of a store instruction, the program will read the JSON file and match with the first line of the "controlUnit" object. The signals of the control unit will thus assume the values indicated inside.

As for the CtrlALU value of the ALU Control component, necessary to select an operation for the ALU, the ALUOp0 and ALUOp1 values match with the first line as well, which makes the CtrlALU for this specific instruction assume the value 2.

Table 4.4 showcases how the different combinations of an instruction's ALUOp and opcode values influence the ALU control input, and whose logic was used in the creation of the "ALU-control" JSON object of listing 4.4.

4.1.5 Computing the CPU Datapath

Finally, in order to fully connect the created components, completing the CPU datapath, a "Cpu" class was created. This class represents one version of a CPU, which is comprised of its own identifier, version (either "Unicycle" or "Pipeline"), and two arrays: one for the CPU's components and another for its connections, along with multiple relevant methods.

```
class CPU {
  constructor(id) {
    this.id = id;
    this.cpuComponents = [];
    this.connections = [];
    this.cpuVersion = "Unicycle";
  }
  //Initializes each CPU component
  initializeCPU(registers, memory) {}

  //Connects the CPU's components
  connectComponents() {}

  //Stores the program's instruction codes in the instruction memory
  setInsMemInstructions(instructions) {}

  //Starts the CPU's execution
  executeCPU(instructionType) {}

  //Resets the CPU completely
  resetCPU() {}

  //Returns the relevant lines of an instruction
  returnCPURelevantLines(instructionType) {}

  //Returns the critical path of an instruction
  returnCPURelevantLines(instructionType) {}
  ...
}
```

Listing 4.5: Class representing a CPU

The key methods for completing the CPU are "initializeCPU" and "connectComponents". The first method is what ultimately assembles every component, connecting them and building the CPU. It starts by reading the JSON configuration file and extracting each component's component

Table 4.3: Control signals' values according to their instruction type

Instruction	**Reg2Loc**	**ALUSrc**	**MemtoReg**	**RegWrite**	**MemRead**	**MemWrite**	**Branch**	**ALUOp1**	**ALUOp0**
R-format	0	0	0	1	0	0	0	1	0
LDUR	X	1	1	1	1	0	0	0	0
STUR	1	1	X	0	0	1	0	0	0
CBZ	1	0	X	0	0	0	1	0	1

Table 4.4: How the ALUOp and opcode influence the ALU control bits

Instruction	**ALUOp**	**Instruction operation**	**Opcode field**	**ALU action**	**ALU control input**
LDUR	00	load register	XXXXXXXXXXX	add	0010
STUR	00	store register	XXXXXXXXXXX	add	0010
CBZ	01	compare and branch on zero	XXXXXXXXXXX	pass input b	0111
R-type	10	ADD	10001011000	add	0010
R-type	10	SUB	11001011000	subtract	0110
R-type	10	AND	10001010000	AND	0000
R-type	10	ORR	10101010000	OR	0001

type, which will be matched with the name of the JavaScript classes created for the components. Following the usual example, the program detects that GenericComponentID1's type is GenericComponent and searches for a class with the same name. When class GenericComponent is found, it creates a new instance of that class with ID "GenericComponentID1" and associates this JSON object to it. This new component will then be added to the "Cpu" class' array of CPU components.

```
initializeCPU(registers, memory) {
   //Read JSON file and get its components
   let jsonFile = cpuJsonMap[this.cpuVersion];
   let jsonComponents = Object.entries(jsonFile.cpuComponents);

   //Get all component classes, stored in a "modules" object
   let componentClasses = Object.values(modules);

   //Initialize variables
   let componentType, component = null;

   for (let i = 0; i < jsonComponents.length; i++) {
     componentType = jsonComponents[i][1].componentType;

     //Find corresponding class
     for (let j = 0; j < componentClasses.length; j++) {
       if (componentType === componentClasses[j].name) {

         //Initialize component with the matching class' constructor
         component = new modules[Object.keys(modules)[j]](
           jsonComponents[i][0],
           jsonComponents[i][1]
         );

         //Add component to component array
         this.cpuComponents.push(component);
       }
     }
   }

   this.connectComponents();

   //Update register file and data memory in their respective components
   ...
 }
```

Listing 4.6: Snippet of the "initializeCPU" method

After every CPU component has been created, the "initializeCPU" method calls "connectComponents". This second method will create the connections between each component, matching inputs and outputs. Having read the JSON configuration file, for each of the "cpuConnections" in it, the method performs a search on the array of CPU components created previously, looking for a component that matches its "origin" and one that matches its "destination". Having found these components, it will then select their input and output that match the ones on the JSON, ultimately connecting the two with the method "createConnection" of the output class. Each new connection is added to the array of CPU connections.

```
connectComponents() {
   //Read JSON file and get its connections
   let jsonFile = cpuJsonMap[this.cpuVersion];
   let cpuConnections = jsonFile.cpuConnections;

   //Initialize variables
   let originComponent, destComponent, output, input = null;

   for (let i = 0; i < cpuConnections.length; i++) {
     //Find corresponding origin component
     originComponent = this.cpuComponents.find(
     (e) => e.id === cpuConnections[i].origin);

     //Find corresponding destination component
     destComponent = this.cpuComponents.find(
     (e) => e.id === cpuConnections[i].dest);

     //Get output of the origin and input of destination
     output = originComponent.outputs[cpuConnections[i].output];
     input = destComponent.inputs[cpuConnections[i].input];

     //Create the connection with the output class' method
     output.createConnection(input);

     //Add the object to the array of CPU connections
     this.connections.push([output, input]);
   }
 }
```

Listing 4.7: "connectComponents" method

Back on the "initializeCPU" method, to finalize the construction of the CPU, it will send the current register file information to the Register Bank component, and the data memory's to the Data Memory component.

Still regarding the "Cpu" class, the methods "returnCPURelevantLines" and "returnCPUCriticalPath" are also important, especially for the platform's frontend. These two will, for a specific instruction type, read the "cpuConnections" object of the JSON configuration file and check, for each connection, if its field for that instruction type is "true". If that is the case, it adds the connection to an array, ultimately returning said array of either relevant lines or the critical path.

For example, for an R-Type instruction, "returnCPURelevantLines" will check whether the "rType" field of a connection's JSON object is true, and, if it is, marks it as a relevant connection. The collected relevant connections will later be displayed on the frontend for the user to see which CPU buses carry important data for a specific instruction. This feature is explained in further detail in Chapter 5. As for the critical path, the logic is mostly the same, and is detailed in subsection 4.1.7.

When the processor is finally completed, with each component containing its logic, inputs and outputs, a program can be executed, which is accomplished with the "executeCPU" method of the "Cpu" class. This method calls, for each component in the array of components previously created, its "execute". As seen in the previous subsection, each component type has a different execution method, and each one is now called sequentially, following the standard CPU execution flow. To finalize, this method will return the "cpuComponents" array, its state now modified by the instruction given.

While still in development, and in order to test the program flow, a method "printValues" was used for logging into the console what was happening in the CPU. By giving the instruction memory a specific instruction code in binary, the program was able to execute the corresponding operations and store them in the register file, which was held by the Register Bank component, and, in the case of memory access instructions, alter the memory values held by the Data Memory component.

However, this is not what the platform is supposed to do, as the user should be able to write assembly code and still get the same results. In that current state, the CPU could only handle machine code, so there must be a way for an assembly instruction to be converted into a number the processor can understand, which is where an assembler comes in.

4.1.6 Assembler

An assembler is a unit capable of reading an assembly instruction and converting it into binary, so it can be interpreted by the CPU and used to execute a program.

The assembler's work, despite being done server-side, actually starts in the frontend of the program. When a user writes and compiles an instruction, said instruction will be broken down into parts. For example, the instruction `ADD X1, X2, X3` will become an object with the format ["ADD", "1", "2", "3"], which is what is sent to the server to turn into machine code. Further details on the frontend and how it connects to the backend are provided in Chapter 5.

Having received this object, the backend will interpret each field and compose a 32-bit number corresponding to that instruction's machine code.

Each instruction has a different encoding process depending on its type, as was previously explained in subsection 4.1.1. To assemble an instruction's 32-bit machine code, each of its assembly code's operands was matched to the appropriate instruction field.

Given the previous example of the assembly code `ADD X1, X2, X3` and the information on Figure 4.1, the necessary fields to assemble this instruction are the opcode, Rm, Rn and Rd (as was previously mentioned in subsection 4.1.1, the shamt field is irrelevant and thus considered "000000"). The opcode, as was seen in Table 4.4, equals "10001011000", while Rn will be

"00010" (the number 2 in binary, which is the number of the first register operand), Rm "00011" (3 in binary) and, finally, Rd "00001". Putting all these numbers together, the resulting machine code is "10001011000000110000000001000001".

After building up the machine code for the desired instruction, the server will send back this result to the frontend, which will, in turn, display it to the user and, at the start of execution, send it to the Instruction Memory component. Through this method, the CPU has access to the instruction in binary format, previously a line of assembly code written by the user.

4.1.7 Latencies and critical path

As was discussed previously in Chapter 3, the unicycle implementation of the CPU is not the most efficient in terms of performance, despite showcasing its behaviour in a simple and easily understood manner.

For students to understand exactly how much time the unicycle processor takes to execute a set of instructions, latencies were implemented into the platform. A component's innate latency is defined as the time required for it to complete its respective operation and is carried out to the components connected to it. This means the total latency of a component is actually the sum of its own (fixed) latency and the highest latency received as output from the other components connected to it.

For instance, assuming that the ALU has a latency of 120ps (picoseconds), and the Register Bank, multiplexor and ALU control connected to it have respectively 600ps, 530ps and 500ps, the actual value of the ALU's latency will be 120 plus 600, which equals 720ps.

These calculations are implemented into the simulation: each component was given, in the JSON configuration file, a default latency value which, on execution, is recalculated according to its outputs, as explained previously.

In addition to the implementation of latencies, the platform also highlights the critical path of each instruction. The critical path is defined as the sequence of components that, altogether, take the longest to execute. Considering the typical latency values of the CPU's components, this critical path, for each type of instruction, will always be the same unless any component's latency is significantly altered.

With this in mind, a few additions were made to the JSON configuration file, specifically in the "cpuComponents" object. For every connection that is part of an instruction type's critical path, a boolean value was added, similarly to the definition of the instruction's relevant lines.

```
    ...
{
  "origin": "GenericComponentID1", "dest": "GenericComponentID2",
  "output": "output1", "input": "input1",
  "rType": "true", "loadType": "false", "storeType": "true",
  "cBranchType": "true", "uncondBranchType": "false",
  "rTypeCP": "true", "storeTypeCP": "false",
  "loadTypeCP": "false", "cBranchTypeCP": "false",
  "uncondBranchTypeCP": "false"
},
   ...
```

Listing 4.8: JSON object of a connection belonging to an instruction's critical path

As seen in the code example, the connection between the two generic components now has five additional boolean values, in the format instruction type plus "CP", which indicates whether that connection is a part of the critical path for that specific instruction type or not. These will be used in the "returnCriticalPath" method of the "Cpu" class, mentioned in subsection 4.1.5, and its behaviour is the same as "returnCPURelevantLines", now taking the "CP" string into consideration. The obtained critical path is then used in the frontend, explained in Chapter 5.

Through these features, the user can understand just how much time the CPU takes to execute a program and how latencies are propagated through the execution flow. In this unicycle CPU version, a high latency component will significantly delay the total execution time, as the subsequent components will only operate when that component finishes its task. This causes a noticeable impact on the CPU's performance, which is why the pipelining technique was created.

4.2 Pipeline CPU Version

The pipeline implementation of the CPU, in regards to its datapath, is much similar to its unicycle version. Although execution in pipelining is divided into five stages, the CPU's components and connections remain mostly the same.

The most significant change in the datapath is the addition of pipeline registers. A pipeline register is an element that acts as an intermediary between two stages, meaning there are exactly four of them, whose names correspond to those stages. As was previously shown in Figure 2.2, the names of the four pipeline registers are:

- IF/ID: between instruction fetch and decoding
- ID/EX: between instruction decoding and execution
- EX/MEM: between execution and memory access
- MEM/WB: between memory access and write back

The behaviour of a pipeline register is rather straightforward. On execution, this component will receive, as input, the values of its first stage's operations, including its control values, and store them. In the next cycle, it will pass those values, as outputs, to its second pipeline stage. Essentially, each instruction will only be at one stage at the same time, passing on to the next one on a new clock cycle.

For example, for register IF/ID, on the execution's first clock cycle, it will store the results of the instruction fetch (IF) operation (in this case, the PC value and the instruction code are sent).

On the next clock cycle, these values will be sent as output to the instruction decode (ID) stage. Meanwhile, another instruction is already being fetched on the IF stage.

Regarding its implementation on the platform, a pipeline register is also its own JavaScript class, extending the “Component” class, much like the example given at listing 4.1. Its structure is essentially the same as the other CPU components, containing various inputs and outputs, which in this case, differ depending on the stages. Each register also implements its own "execute" method, which will simply transfer its inputs’ values to its outputs.

Despite transferring the values of one stage to another, this class does not handle the distribution of control values. Rather, for storing the control values, another component was created, called "PipelineControl". The control propagation logic was kept separate through this class in order to simulate the datapath illustrated in Figure 2.2, whose control elements are isolated atop each pipeline register.

Which control values need to be stored depends on which stage the pipeline is in. As seen in Figure 2.2, in pipeline register ID/EX, control signals related to EX, MEM and WB are stored, but in the next register, EX/MEM, only MEM and WB are, and in MEM/WB only WB is relevant. The control signals stored in each of these components are as follows:

- EX: ALUOp0, ALUOp1, ALUSrc
- MEM: memRead, memWrite, branch, uncondbranch
- WB: memToReg, regWrite

The addition of these new elements to the datapath implies a change in the CPU’s configuration file, as new components and their connections must be included. In order to separate the unicycle and the pipeline’s logic, another JSON file was created, following the same structure as the one created for the unicycle implementation, explained in subsection 4.1.3. This file’s content is mostly the same as the unicycle’s, solely adding the new components and connections.

Having defined the CPU’s components and their respective behaviours in this new JSON file, the rest of the implementation process follows the same flow as the unicycle version. In this case, however, the methods to initialize the CPU and start a program’s execution are already defined, as was explored in subsection 4.1.3. Because of this, one merely needs to change the JSON file that is read in the "Cpu" class to the pipeline version, and the pipeline CPU’s simulation is finally complete.

Through the method described in this section, it was possible to simulate the CPU’s pipeline version by making slight changes to the unicycle. The two versions, despite following the same implementation process and sharing components and methods, ultimately behave very differently. The pipelined version is now capable of working on five instructions at the same time, in a single clock cycle, significantly improving its performance in comparison to the unicycle.

4.2.1 Pipeline Hazards

Besides the addition of the pipeline registers, this CPU version implements two new components, dedicated to identifying and solving hazards that may occur throughout a program's execution. The developed platform is capable of handling a type of hazard that occurs when data necessary to execute an instruction is not yet available, called a data hazard.

Data hazards in the pipeline must be solved as effectively as possible, in order to avoid drastically impairing the pipeline's performance. In order to do so, forwarding, a method which retrieves the missing data from internal buffers instead of waiting for it to arrive from registers or memory, was implemented through the forwarding unit. In certain cases, however, such as when a load is followed by an operation that reads its result, this unit cannot solve the problem by itself. A hazard detection unit is necessary, operating in the ID stage, and capable of identifying such problems and solving them by stalling the pipeline.

Both these components, the forwarding unit and the hazard detection unit, were thus implemented into the platform. Their structure is the same as every other CPU component, having a JavaScript class of their own, and their execution methods follow a set of conditions that identify and solve hazards.

For instance, when an instruction, in the EX stage, tries to use a register that a previous instruction is writing to, the ALU will not have access to it yet, as the WB stage is still in progress for that previous instruction. The forwarding unit identifies this dependence of registers and forwards the result on the MEM stage to the ALU. This process prevents the pipeline from stalling and consequently worsening its performace.

The hazard detection unit is used when forwarding is not enough to solve a more complicated hazard. As mentioned previously, when a certain operation must use a register whose value is still being loaded from memory, the pipeline must be stalled in order for the hazard to be solved, as the register value is only available after the MEM stage and cannot be obtained by forwarding to the ALU. The hazard detection unit, which operates in the ID stage, is able to identify this problem and, in its execution method, sets the control unit's signals to zero, which inserts a stall between the load operation and the instruction dependent on it. As a consequence, the pipeline will do nothing for one cycle and the forwarding unit will, later, be able to handle the dependence and proceed with the program's execution.

With the implementation of hazard detection and forwarding logic into the simulator, the backend development of the pipeline is complete, and the platform is capable of simulating both this version and the unicycle CPU.

4.3 User Sessions

After the program is working correctly for one user, the platform must be extended to allow multiple users to access it at the same time. In order to achieve this, sessions had to be implemented into the application's backend.

A user session is saved as a JavaScript class that holds information about the user's current CPU states, instructions, register file and memory. Each user that accesses the website for the first time will have their own session created, which will be maintained throughout the time spent exploring the platform. This essentially means that two different users have each a different session assigned to them, and one's usage does not influence the other's program state.

```
class UserSession {
  constructor() {
    //Assigns a CPU object to the user session
    this.cpu = new CPU("basicCPU");

    //Stores the CPU states executed
    this.cpuStates = [];

    //Stores the group of instructions executed
    this.instructionGroup = [];

    //Register file of the session
    this.registers = [];

    //Memory contents of the session
    this.memory = new Array(15).fill(0);

    //Group with the type of the instructions executed
    this.instructionTypeGroup = [];

    //Relevant lines of the CPU states
    this.relevantLines = [];

    //Critical path of the CPU states
    this.criticalPath = [];
  }
  ...
}
```

Listing 4.9: "UserSession" contents

Were this not implemented, while a user was testing the platform, altering its CPU values, another user could start a new execution, overlapping the two backend states and resulting in the malfunction of the program.

All sessions currently in use are stored in memory using Redis. As explained in the Technologies section of Chapter 3, Redis is an in-memory data store, whose usage contributes to faster response times. A session is cleared after 15 minutes of inactivity, so unused sessions are not kept in memory indefinitely.

The implementation of this user session system enables access to the platform to multiple users at the same time, a feature crucial for this platform, as a significant number of students can thus explore the website simultaneously.

4.4 Server Logic

Having finished the CPU's simulation logic for both its unicycle and pipelined versions, the next logical step of implementation is developing an interface for the user to interact with, allowing the execution of a set of instructions, showcasing its effects on the datapath and each component's characteristics. This user interface would constitute the frontend layer of the platform.

However, for both the backend and frontend to work in accordance, they first need to be connected in some way. As the CPU simulation logic implemented is significantly heavy on operations, running its calculations client-side would overload the user's machine and thus be extremely inefficient. Having a dedicated web server that runs the simulation and handles all requests made on the frontend was the approach selected to solve this problem.

In order to achieve this, a new file, "main.js," was created, which acts as the main environment for all backend operations. The Node.js framework Express was then used to create a simple web server.

The purpose of this Express server is to be used as an application programming interface (API), so that it can communicate with the frontend layer and answer its requests, receiving, altering, and sending back data. In order to do this, API endpoints for each relevant backend operation had to be defined.

For example, the endpoint created for route "/execute" is shown in listing 4.10. Whenever the frontend layer, as a result of a user executing a program, makes an HTTP GET request to this route, the server will start execution and send back the results obtained as a JSON object.

```
app.get("/execute", (req, res) => {
  //Fetch the user session and initialize the CPU
  let userSession = userSessions[req.session.id];
  userSession.cpu.initializeCPU(userSession.registers, userSession.memory);

  //Send the assembled instructions to the instruction memory
  userSession.cpu.setInsMemInstructions(userSession.instructionGroup);

  let instructionFlow = [];

  //If pipeline, loop 4 more times because an instruction will go through each of
      the 5 stages
  let maxPC = userSession.instructionGroup.length + (userSession.cpu.cpuVersion ===
       "Pipeline" ? 4 : 0);

  //Iterating through the group of instructions to execute
  for (let i = 0; i < maxPC && instructionFlow.length <= 200;) {
    //Array of numbers representing the flow of the program ("1" is the first
        instruction, "2" the second, and so on)
    instructionFlow.push(i);

    //Creates a CPU state with the result of the "executeCPU" method, storing it
    in the user session
    let state = userSession.cpu.executeCPU(userSession.instructionTypeGroup[i]);

    //Stores obtained relevant lines and critical path in the user session
    userSession.relevantLines.push(userSession.cpu.returnCPURelevantLines(
        userSession.instructionTypeGroup[i]));
    userSession.criticalPath.push(userSession.cpu.returnCriticalPath(
    userSession.instructionTypeGroup[i]));

    //Checks the PC value for a jump instruction that could change the iterator's
        value
    i = state[0].updatedPC.data.value / 4;

    //Stores the CPU states in the user session
    userSession.cpuStates.push(JSON.parse(JSON.stringify(state)));
  }

  //Stores this session in memory
  userSessions[req.session.id] = userSession;

  //Sends the execution's results to the frontend
  res.send({
    cpuStates: userSession.cpuStates,
    instructionFlow: instructionFlow,
    relevantLines: userSession.relevantLines,
    criticalPath: userSession.criticalPath,
  });
});
```

Listing 4.10: Endpoint for route "/execute" that handles a program's execution

A complete list of the API's endpoints and their characteristics is presented in table 4.5.

On the backend side, all endpoints are defined, and the program is ready to handle user HTTP requests. How this is done on the client side is described in section 5.3 of the next chapter, dedicated to the project's frontend development.

4.5 Concluding Remarks

In this fourth chapter, the implementation process behind the platform's backend was laid out, starting with a simple overview of assembly instruction types and how a CPU executes a program, followed by an explanation of the unicycle CPU's simulation. This simulation included setting up the processor's multiple components and connections, defining their execution behaviour, and creating an assembler able to convert instructions to machine code, ultimately resulting in a simulator

Table 4.5: Table with the API's endpoints and their purpose

Route	**Method**	**Behaviour**	**Response**
/execute	GET	Calls for methods of the "Cpu" class ("initializeCPU" and "execute"), storing the results of execution as a CPU "state". Sends back the CPU states obtained, along with the instruction flow, relevant lines and critical path.	{cpuStates, instructionFlow, relevantLines, criticalPath}
/reset	GET	Resets the CPU, updating the one in the user's session to an empty state.	OK
/sendRegisters	POST	Receives the current register file from the frontend and stores it in the user's session.	OK
/changeCPUVersion	POST	Receives either "Unicycle" or "Pipeline", resets the CPU and changes the CPU version of the user's session.	OK
/readInstruction	POST	Receives one or more assembly instructions from the frontend and breaks them down into the appropriate fields, converting them to the binary format. Returns the generated machine code as response. More details about this process can be found in subsection 4.1.6.	{instructionCodes}

capable of executing a set of instructions correctly. Subsequently, the pipeline version's implementation was described, comparing its behaviour to the unicycle. The chapter closes with an explanation of the web server's logic, how it handles user requests and proceeds with a program's execution. How the user interacts with the platform and how these requests are sent, however, is described in the next chapter, dedicated to the frontend layer and its development process.

Chapter 5

User Interface

In this chapter, the frontend layer of the web application is described in detail. This includes the thought process behind the website's design and a highlight of its main features, implementation details on how the CPU datapath display was carried out and how each CPU state is managed, closing with an explanation of how this frontend layer connects with its backend counterpart, completing the implementation of the platform.

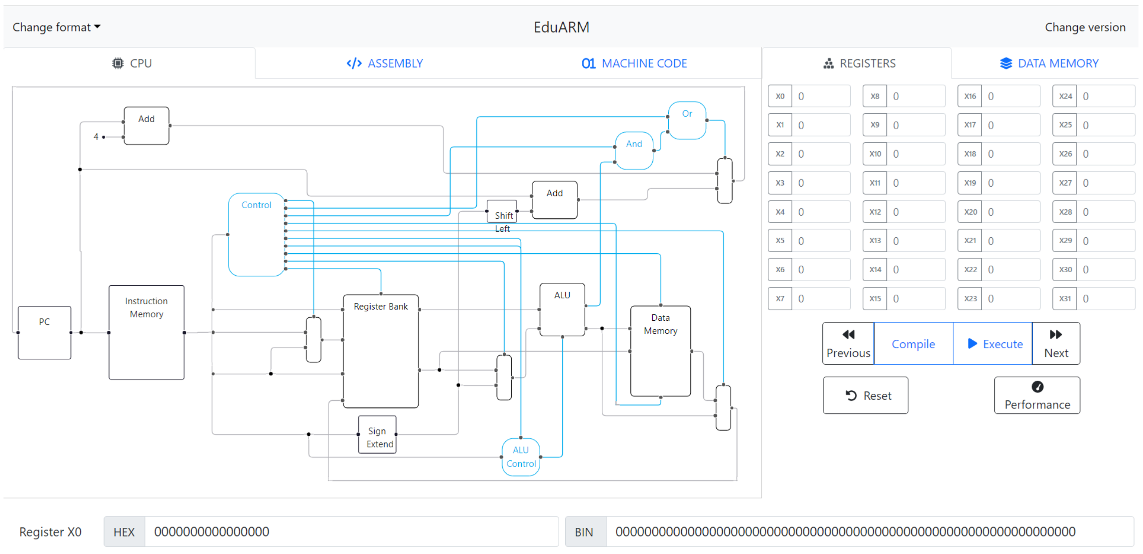
5.1 Interface Design and Features

The application's user interface was designed with the objective of providing students with a user friendly and interactive environment for studying. In order to achieve this, the interface was kept mostly simple, avoiding any visual clutter and the addition of too much information, which could end up confusing students and ultimately doing more harm than good. The finished product's homepage is shown in Figure 5.1.

Regarding its features' positions, the platform's most prominent element is the CPU datapath, on its left side, with its components and corresponding connections. In this visual representation, control lines and related components are colored blue.

On top of the diagram, two more tabs besides the CPU one can be seen: "Assembly" and "Machine Code". The former contains a grey text area, dedicated to writing assembly code, while the latter will display each instruction's machine code after compiling a program.

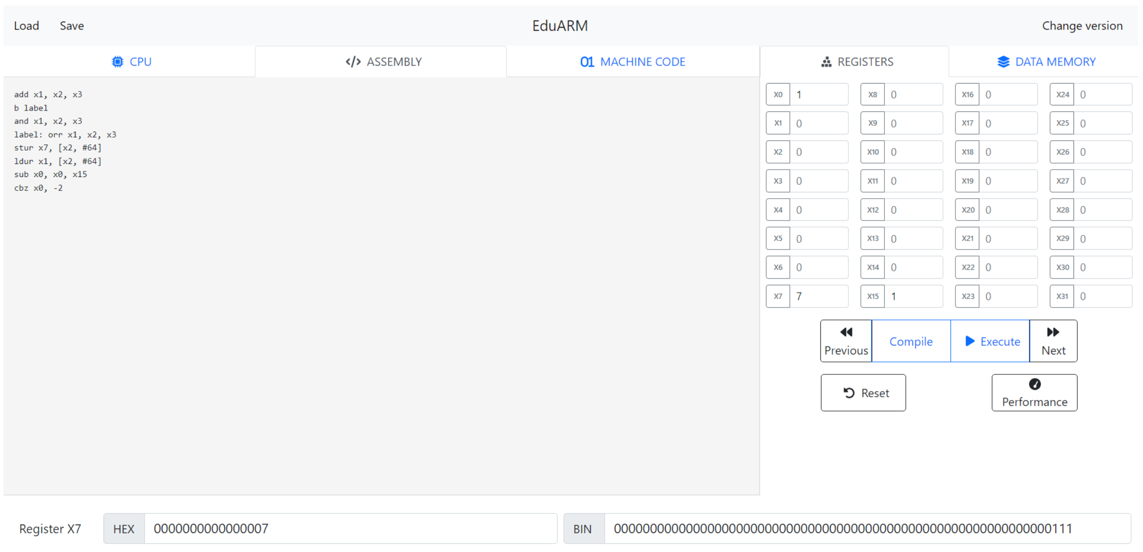
Besides its dedicated assembly coding area, the "Assembly" page also features two buttons on the top left corner of the screen, "Load" and "Save". If the user already has their assembly program written in an external text file, it can be imported into the program using the "Load" feature, which will open the computer's file explorer to select said file. Additionally, programs written on the "Assembly" text area can also be saved into the user's device by clicking on the "Save" option, which will automatically download a text file with the code.



X0	0	X8	0	X16	0	X24	0
X1	0	X9	0	X17	0	X25	0
X2	0	X10	0	X18	0	X26	0
X3	0	X11	0	X19	0	X27	0
X4	0	X12	0	X20	0	X28	0
X5	0	X13	0	X21	0	X29	0
X6	0	X14	0	X22	0	X30	0
X7	0	X15	0	X23	0	X31	0

Register X0 HEX 0000000000000000 BIN 0000000000000000000000000000000000000000000000000000000000000000

Figure 5.1: Homepage of eduARM



```
add x1, x2, x3
b label
and x1, x2, x3
label: orr x1, x2, x3
stur x7, [x2, #64]
ldur x1, [x2, #64]
sub x0, x0, x15
cbz x0, -2
```

X0	1	X8	0	X16	0	X24	0
X1	0	X9	0	X17	0	X25	0
X2	0	X10	0	X18	0	X26	0
X3	0	X11	0	X19	0	X27	0
X4	0	X12	0	X20	0	X28	0
X5	0	X13	0	X21	0	X29	0
X6	0	X14	0	X22	0	X30	0
X7	7	X15	1	X23	0	X31	0

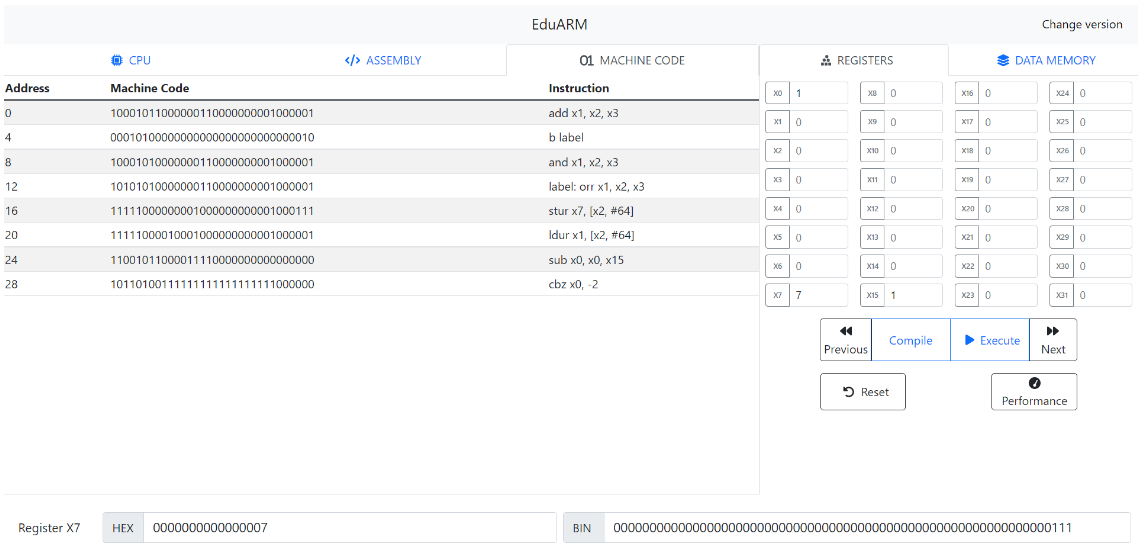
Register X7 HEX 0000000000000007 BIN 0000000000000000000000000000000000000000000000000000000000000111

Figure 5.2: Area dedicated to assembly programming and the register file

On the screen's rightmost area, the content of each register, from X0 to X31, is displayed. The register file's values can be changed by clicking on each input area, typing the desired number in its decimal format, and pressing enter. Through this process, the hexadecimal and binary representations of the register will also be displayed in the two fields at the bottom of the page, always on their 64-bit format. The content of these can also be filled in by the user, and the corresponding conversions will be made to the register file. For instance, if a user selects register "X0" and types "0c" on the hexadecimal input area, after pressing enter, the number's value in decimal and binary will show "12" and "1100" (extended to 64 bits), respectively.

In Figure 5.2, a simple assembly program is written in the "Assembly" text area, with a few registers already filled out and the hexadecimal and binary formats of register X7, which was selected, displayed in the bottom of the page.

After a user writes the code and changes the necessary registers, the program is ready to be executed. The area below the register file contains a variety of buttons. The user can press "Compile" and the machine code of each of the program's instructions will appear in the "Machine Code" page, along with their position on the instruction memory and their assembly code, as seen on Figure 5.3.



Address	Machine Code	Instruction
0	10001011000000110000000001000001	add x1, x2, x3
4	00010100000000000000000000000010	b label
8	10001010000000110000000001000001	and x1, x2, x3
12	10101010000000110000000001000001	label: orr x1, x2, x3
16	11111000000001000000000001000111	stur x7, [x2, #64]
20	11111000010001000000000001000001	ldur x1, [x2, #64]
24	11001011000011110000000000000000	sub x0, x0, x15
28	10110100111111111111111111000000	cbz x0, -2

X0	1	X8	0	X16	0	X24	0
X1	0	X9	0	X17	0	X25	0
X2	0	X10	0	X18	0	X26	0
X3	0	X11	0	X19	0	X27	0
X4	0	X12	0	X20	0	X28	0
X5	0	X13	0	X21	0	X29	0
X6	0	X14	0	X22	0	X30	0
X7	7	X15	1	X23	0	X31	0

Register X7 HEX 0000000000000007 BIN 0000000000000000000000000000000000000000000000000000000000000111

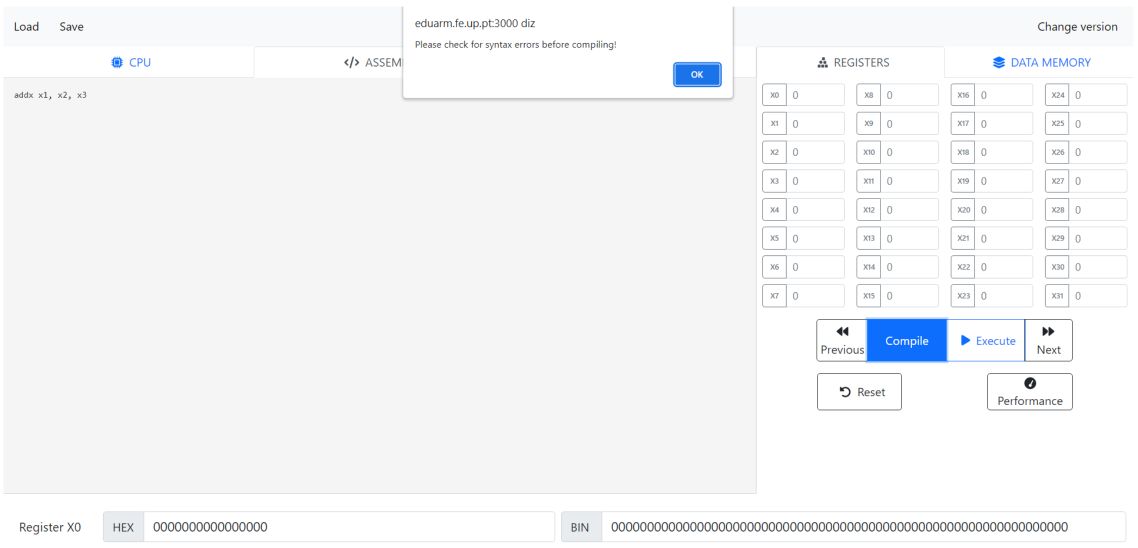
Figure 5.3: Machine code of the written assembly program

In the case of the platform identifying any compilation errors, these will appear as an alert on top of the page, and the program will not proceed unless these are fixed. The details of the error are logged into the browser's console. The following actions will ask the user to fix any errors:

- Not compiling the program before executing
- Trying to compile/execute an empty block of assembly code
- Using an instruction that does not belong to the set of supported instructions

- Writing an instruction whose register operands do not start with an "X"
- Writing an instruction using a register operand that does not exist

An example of an alert is given in Figure 5.4, where the user wrote an instruction that does not exist.



eduarm.fe.up.pt:3000 diz

Please check for syntax errors before compiling!

```
addx x1, x2, x3
```

X0	0	X8	0	X16	0	X24	0
X1	0	X9	0	X17	0	X25	0
X2	0	X10	0	X18	0	X26	0
X3	0	X11	0	X19	0	X27	0
X4	0	X12	0	X20	0	X28	0
X5	0	X13	0	X21	0	X29	0
X6	0	X14	0	X22	0	X30	0
X7	0	X15	0	X23	0	X31	0

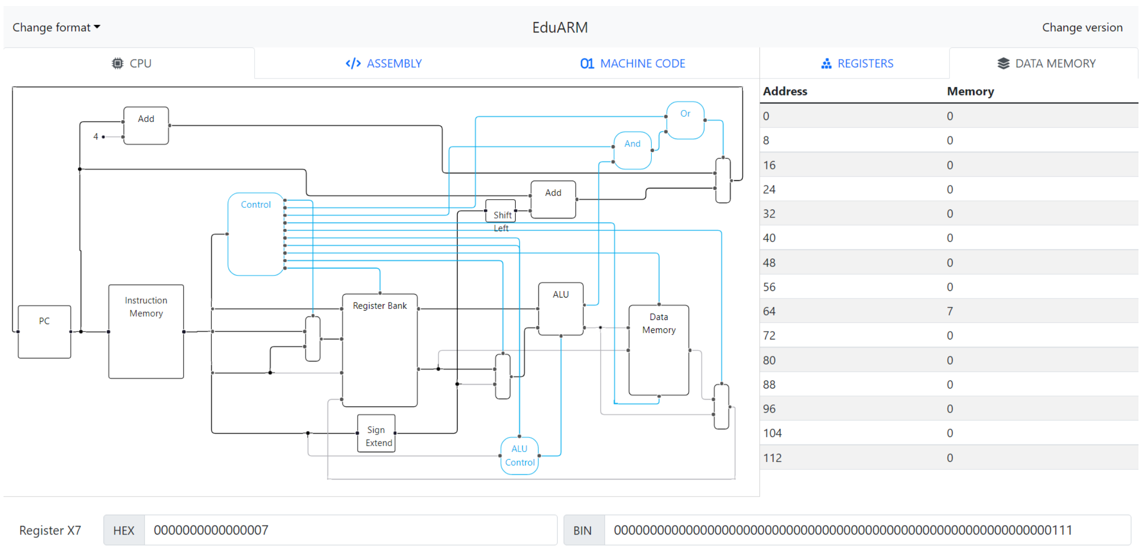
Register X0 HEX 0000000000000000 BIN 0000000000000000000000000000000000000000000000000000000000000000

Figure 5.4: Alert received after writing an unknown instruction

Having successfully compiled the program, the user can then proceed by clicking the "Execute" button. The program is thus finally executed, and the datapath will showcase, for each instruction, the internal behaviour of the CPU. Any changes to data memory, in the case of executing memory access instructions, can be checked in the "Data Memory" tab, next to "Registers", as shown in Figure 5.5. In the case of the used example, value "7" will appear on the 64th memory position, due to the store instruction with register X7. The register file also changes according to the instructions executed, as seen in Figure 5.6, where X0 took on the value "-1".

After execution, the datapath will display the state of the last instruction written. The user can freely cycle through instructions by using the "Previous" and "Next" buttons next to "Execute". Below this area, the current instruction's machine code and its different fields (following the logic of Figure 4.1) are presented, also updating when changing the current instruction. The instruction's assembly code is also displayed in this area and highlighted on the code written in the "Assembly" tab.

Connections that are considered relevant for each instruction, that is, that carry any sort of relevant data, are painted black. The various CPU components can be hovered over in order to see their IDs, latencies, inputs and outputs' current values. These values can also be converted to either their binary or hexadecimal formats by selecting the desired representation with the “Change format” button on the top left side of the screen.

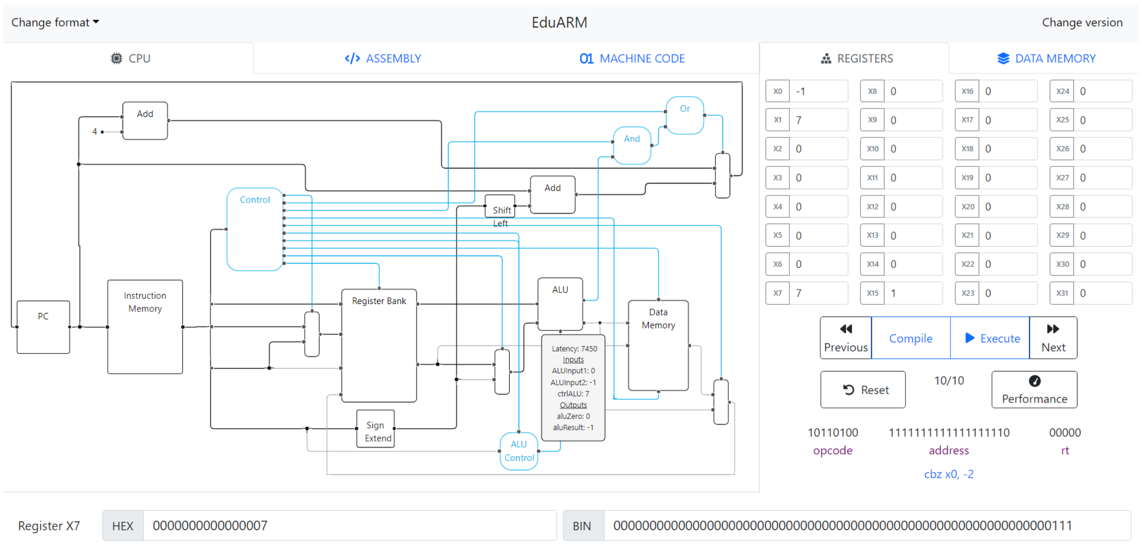


Address	Memory
0	0
8	0
16	0
24	0
32	0
40	0
48	0
56	0
64	7
72	0
80	0
88	0
96	0
104	0
112	0

Register X7 HEX 0000000000000007 BIN 0000000000000000000000000000000000000000000000000000000000000111

Figure 5.5: Datapath and memory after execution

These features are highlighted through Figure 5.6, representing the results of a conditional branch instruction, which is the last one to be executed in the example program written in 5.2. In the datapath area, a tooltip of the ALU's state is presented, which includes its current latency, inputs and outputs. The instruction's opcode, address and rt fields are displayed below the execution area, along with a counter indicating "10/10". This element represents the number of steps of execution of the program, which the user can cycle through freely and explore changes to the datapath, registers and memory.



Latency: 7450
Inputs
ALUInput1: 0
ALUInput2: -1
ctrlALU: 7
Outputs
aluZero: 0
aluResult: -1

X0	-1	X8	0	X16	0	X24	0
X1	7	X9	0	X17	0	X25	0
X2	0	X10	0	X18	0	X26	0
X3	0	X11	0	X19	0	X27	0
X4	0	X12	0	X20	0	X28	0
X5	0	X13	0	X21	0	X29	0
X6	0	X14	0	X22	0	X30	0
X7	7	X15	1	X23	0	X31	0

10/10

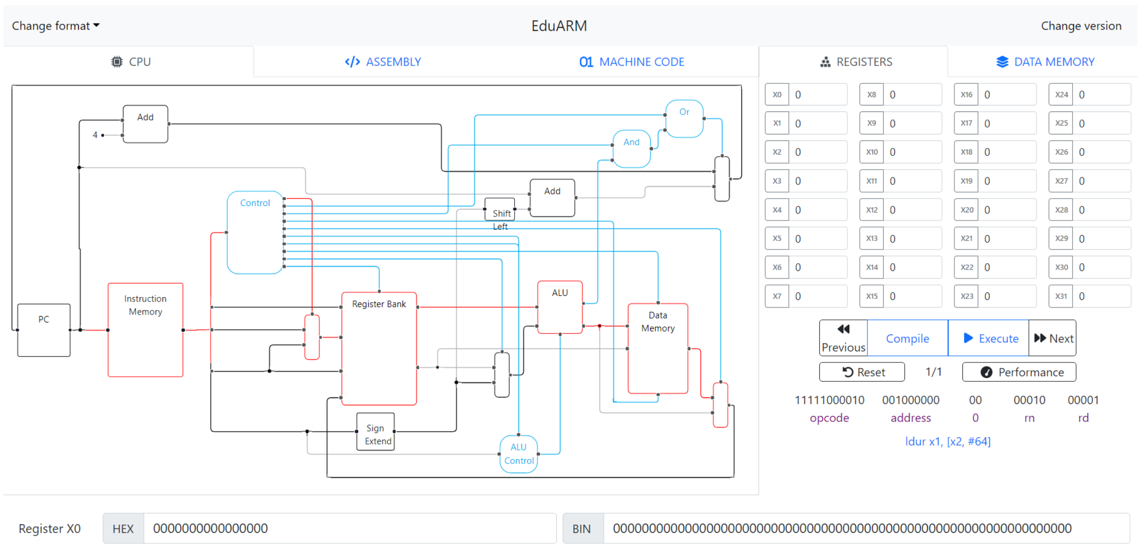
opcode	address	rt
10110100	1111111111111111110	00000

cbz x0, -2

Register X7 HEX 0000000000000007 BIN 0000000000000000000000000000000000000000000000000000000000000111

Figure 5.6: Interface after executing a program

Additionally, two buttons entitled "Reset" and "Performance" are located below the execution controls. The former will reset the program for the user, setting all registers and memory values to zero and reverting the datapath to its original state. "Performance" grants access to the CPU's critical path, whose connections and components will be highlighted red (much like the relevant lines painted black), also changing according to the instruction on display. The critical path will highlight the CPU elements that carry the highest latencies for that instruction, which in the case of the load operation are illustrated through Figure 5.7.



X0	0	X8	0	X16	0	X24	0
X1	0	X9	0	X17	0	X25	0
X2	0	X10	0	X18	0	X26	0
X3	0	X11	0	X19	0	X27	0
X4	0	X12	0	X20	0	X28	0
X5	0	X13	0	X21	0	X29	0
X6	0	X14	0	X22	0	X30	0
X7	0	X15	0	X23	0	X31	0

11111000010	001000000	00	00010	00001
opcode	address	0	rn	rd

ldur x1, [x2, #64]

Register X0 HEX 0000000000000000 BIN 0000000000000000000000000000000000000000000000000000000000000000

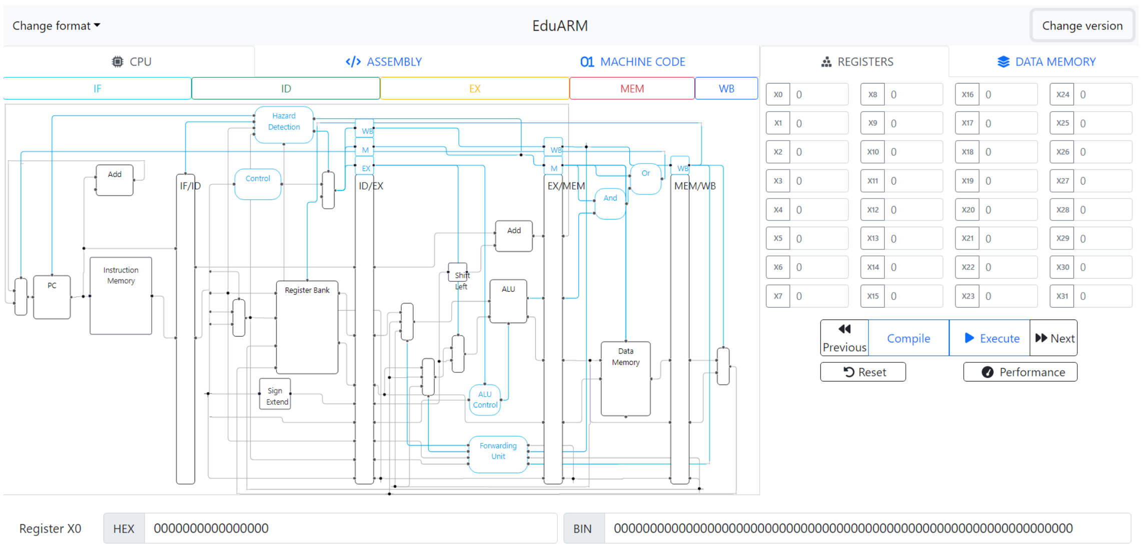
Figure 5.7: Critical path of a load instruction

The button "Change version" in the top right corner of the page can be used to change between the unicycle and pipelined versions of the CPU. While the unicycle is the one chosen as default, the user can press this button to switch to the pipeline, whose look is mostly the same, except for the datapath, that changes as presented in Figure 5.8. The datapath's tooltips, however, function in the exact same way as the unicycle, using the cursor to hover over the CPU components.

In this version, the five pipeline stages are represented with colors on top of the datapath. The process of executing an assembly program is the same as the unicycle's, but its results slightly differ.

Instead of representing one instruction, a CPU state will represent the five pipeline stages in a clock cycle, which could have five instructions at the same time. Because of this, the number of steps is now five times higher, and the "Previous" and "Next" buttons do not iterate through instructions, but rather through clock cycles. Which instruction is in each stage can be seen in the area below the execution buttons.

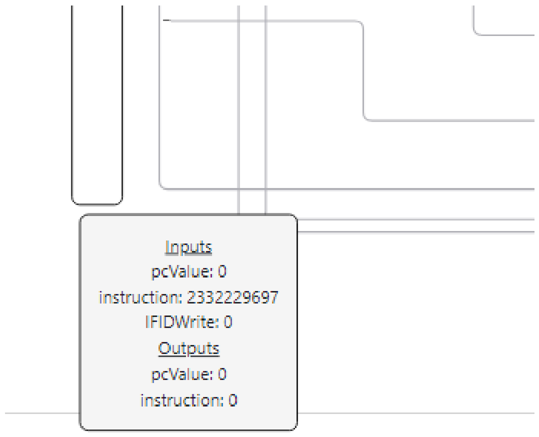
To demonstrate how the pipeline stages function, an example is provided in Figure 5.9. In the datapath, a tooltip of the IF/ID pipeline register is presented, where it is clear, in (a), that its contents are not immediately sent to the next stage, as they stay in as "inputs". By clicking "Next", however, the pipeline will advance one clock cycle and these values will already be propagated to the next stage, receiving another instruction as input, as seen in (b).



X0	0	X8	0	X16	0	X24	0
X1	0	X9	0	X17	0	X25	0
X2	0	X10	0	X18	0	X26	0
X3	0	X11	0	X19	0	X27	0
X4	0	X12	0	X20	0	X28	0
X5	0	X13	0	X21	0	X29	0
X6	0	X14	0	X22	0	X30	0
X7	0	X15	0	X23	0	X31	0

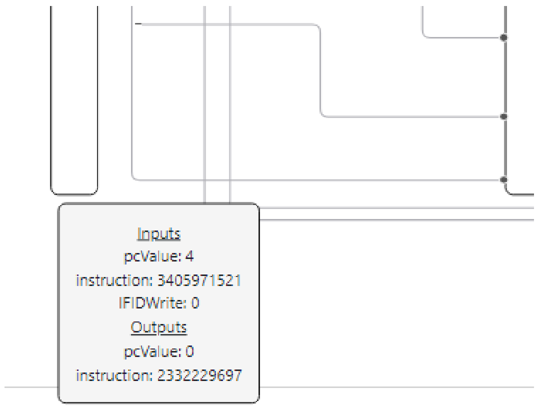
Register X0 HEX 0000000000000000 BIN 0000000000000000000000000000000000000000000000000000000000000000

Figure 5.8: Pipeline CPU version of the platform



Inputs
pcValue: 0
instruction: 2332229697
IFIDWrite: 0
Outputs
pcValue: 0
instruction: 0

(a) First clock cycle

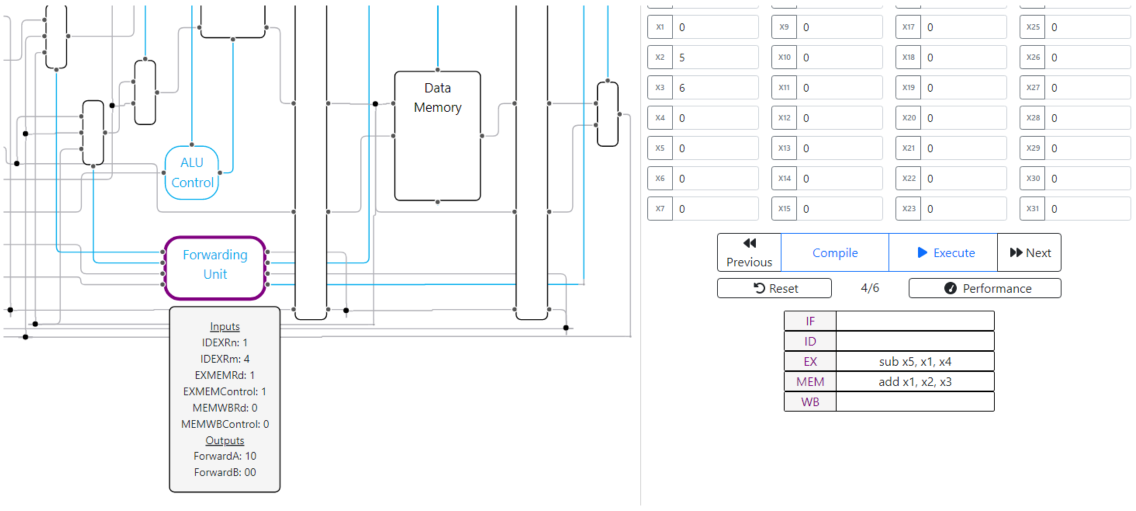


Inputs
pcValue: 4
instruction: 3405971521
IFIDWrite: 0
Outputs
pcValue: 0
instruction: 2332229697

(b) Second clock cycle

Figure 5.9: Close-up of a pipeline register in between two stages

Regarding forwarding and hazard detection, when either unit identifies a problem, their datapath component will be highlighted purple. For instance, Figure 5.10 shows the forwarding unit attempting to solve a register dependence, as the sub instruction uses register X1, which is still being modified by the add instruction. The appropriate control signal is set, as seen in the tooltip, and the forwarding unit will fetch the value from the MEM stage and send it to the ALU.



Inputs
IDEXRn: 1
IDEXRm: 4
EXMEMRd: 1
EXMEMControl: 1
MEMWBRd: 0
MEMWBControl: 0
Outputs
ForwardA: 10
ForwardB: 00

X1	0	X9	0	X17	0	X25	0
X2	5	X10	0	X18	0	X26	0
X3	6	X11	0	X19	0	X27	0
X4	0	X12	0	X20	0	X28	0
X5	0	X13	0	X21	0	X29	0
X6	0	X14	0	X22	0	X30	0
X7	0	X15	0	X23	0	X31	0

IF	
ID	
EX	sub x5, x1, x4
MEM	add x1, x2, x3
WB	

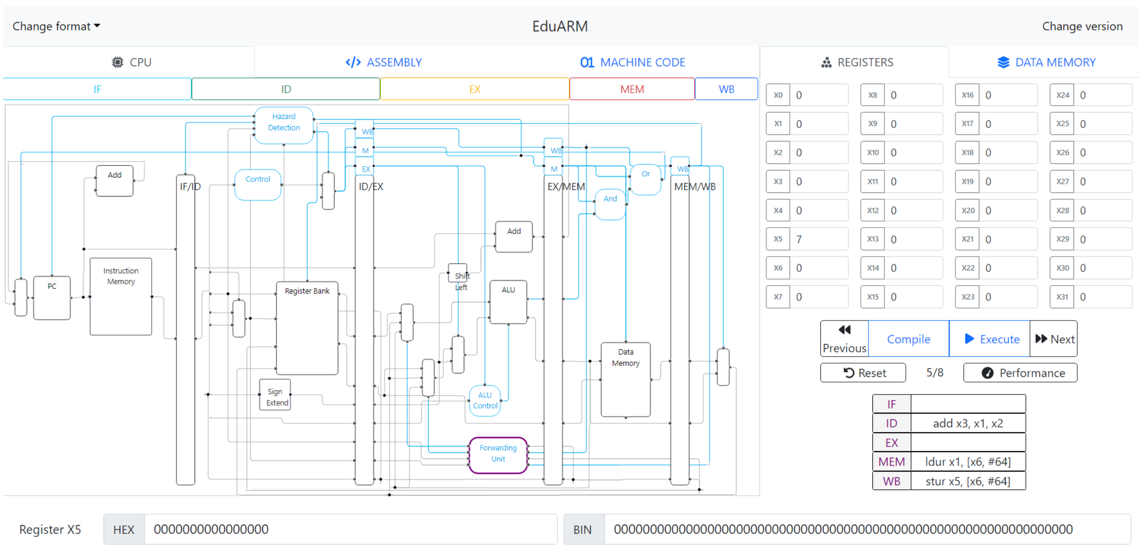
Figure 5.10: Forwarding in the pipeline to solve an instruction dependence

As for the hazard detection unit, Figure 5.11 illustrates a situation where an add instruction uses a register whose value is still being loaded from memory. As explained in subsection 4.2.1 of Chapter 4, such cases cannot be solved by forwarding alone, requiring the pipeline to be stalled. The hazard detection unit, highlighted purple, identifies this problem and sets the appropriate control signals for solving it, sent to the multiplexor connected to Control, to the Program Counter and to the IF/ID register. The PC value and the contents of IF/ID are preserved, while the multiplexor sets all control values to zero, stalling the pipeline by one clock cycle. During a stall, the processor will not do any relevant operations and a "bubble" is added to the instruction flow, which is represented in the area below the execution buttons through an empty field in the table.

5.2 Implementation Details

As previously mentioned in subsection 3.3.2 of Chapter 3, the technology chosen for the platform's frontend implementation was React. This JavaScript library allows its users to build complex user interfaces by combining independent and reusable pieces of code, called components. These components can store state variables, which are unique React variables capable of preserving values between function calls.

The main React component of the platform is called "App.js", which stores essentially every state variable related to CPU simulation, such as the state of the datapath, registers or memory. This component represents the main page of the platform, creating its interface by encapsulating



X0	0	X8	0	X16	0	X24	0
X1	0	X9	0	X17	0	X25	0
X2	0	X10	0	X18	0	X26	0
X3	0	X11	0	X19	0	X27	0
X4	0	X12	0	X20	0	X28	0
X5	7	X13	0	X21	0	X29	0
X6	0	X14	0	X22	0	X30	0
X7	0	X15	0	X23	0	X31	0

IF	
ID	add x3, x1, x2
EX	
MEM	ldur x1, [x6, #64]
WB	stur x5, [x6, #64]

Register X5 HEX 0000000000000000 BIN 0000000000000000000000000000000000000000000000000000000000000000

Figure 5.11: Results of a hazard on the instruction flow

other React components. The App component is also where HTTP requests related to a program's execution are made. Additionally, it computes the logic behind cycling through and representing the various CPU states obtained in those requests for execution, further explained in subsection 5.2.2.

5.2.1 Datapath Visualization

A key React component of the interface is called "Datapath". This is where the CPU datapath's visual representation is created, resulting in a collection of different nodes and edges, as seen in the platform's homepage.

The CPU datapaths, for both unicycle and pipelined versions, were created using a React library called ReactFlow, suitable for building node-based applications. Despite its main purpose of designing interactive diagrams and graphs, the library's high customizability made it possible to create a structure that accurately replicates the datapath designs presented in section 2.1 of Chapter 2, with nodes representing the CPU components, and edges the connections between them.

The ReactFlow library provides a React component of the same name, which can receive nodes and edges as props. In this implementation, the nodes and edges are arrays whose structures are exemplified in listing 5.1 and 5.2, respectively.

```
{
  id: "RegBank",
  data: { label: "Register Bank" },
  position: { x: 450, y: 285 },
  type: "regBankNode",
},
```

Listing 5.1: "RegBank" node created for representing the register bank

Each node contains an ID, which matches a CPU component's name, a data object that may include a label (in the case of a fork node, for instance, no labels are added) and a position. The "type" element represents the node's type, which can be "default" (contains exactly one input and one output), "input" (only one input), "output" (only one output) or custom. Each node that has more than one input or output had to be custom made.

```
{
  id: "readData1/ALUInput1",
  source: "RegBank",
  sourceHandle: "a",
  targetHandle: "a",
  target: "ALU",
  type: "smoothstep",
},
```

Listing 5.2: Edge connecting one of RegBank's outputs with an ALU input

An edge is defined by an ID, which is the names of its output and input, separated by "/", a source node, a target node, a type of line design, and, in the case of connecting with a node that has multiple inputs/outputs, a "sourceHandle" or "targetHandle" that indicates which output or input to connect with, respectively.

Both these objects were created with the intent of mirroring the backend's datapath, explained in subsection 4.1.3 of Chapter 4. Since each node was given the same ID as its respective component on the JSON configuration file, it was possible to match the two and display their current state to the user. This is achieved through method "showNodeInformation" of the Datapath component, capable of identifying when a user places the cursor on a component of the datapath after execution. When this event is triggered, the method will retrieve the node's ID (as defined on its array in DiagramUtils), and search through the current backend CPU state for a component that matches that ID. If this operation is successful, a tooltip will appear on the interface, next to the component, displaying the component's latency, inputs and outputs. The presented tooltips will disappear when the user stops hovering over the component.

Through this process, connecting the frontend datapath with the one previously defined in the backend was rather straightforward, allowing the platform's users to have access to the CPU's simulation values at any given time. Additionally, since the CPU states change whenever the user presses the "Next" or "Previous" buttons, these tooltips will also change accordingly, which means students can easily understand which components and connections have an impact in each step of the program.

5.2.2 CPU States and Execution

Upon execution, a method in the App component named "executeProgram" is called, whose structure is shown in listing 5.3.

```
const executeProgram = () => {
...
   setInstructionDisplayed(instructions[instructions.length - 1]);
   axios.post(BASE_URL + ":3001/sendRegisters", registerValues, {
       withCredentials: true, credentials: "include",
   }).then(() => {
       axios.get(BASE_URL + ":3001/execute", { withCredentials: true }).then(
           function (res) {
         setInstructionFlow(res.data.instructionFlow);
         setSavedCPUStates(res.data.cpuStates);
         setSavedRelevantLines(res.data.relevantLines);
         setSavedCriticalPath(res.data.criticalPath);
         setCpuState(res.data.cpuStates[res.data.cpuStates.length - 1]);
         setCpuIndex(res.data.cpuStates.length - 1);
         setRelevantLines(res.data.relevantLines[res.data.cpuStates.length-1]);
         setCriticalPath(res.data.criticalPath[res.data.cpuStates.length-1]);
         updateRegisters(res.data.cpuStates[res.data.cpuStates.length - 1]);
       });
   });
...
```

Listing 5.3: Snippet of method "executeProgram"

The method starts by setting the state variable "instructionDisplayed" to the last line of the executed assembly code, which will be displayed below the execution buttons' area. The current state of the register file is then sent to the server using the POST method for route "/sendRegisters". Upon receiving back an acknowledgement, the client will then send a GET request for route "/execute", asking for the resulting CPU states of executing the program.

The backend will, in turn, handle this request as explained in table 4.5, sending back a JSON object with the appropriate response, whose elements are stored each in their corresponding React state variables, as presented in table 5.1.

These variables are then sent as props to the React components that will use them for calculations or for displaying them on the platform, the most significant being the Datapath component, which will, as explained in the previous subsection, match the frontend CPU states with the backend's datapath.

The process of cycling through states, that is, switching from one CPU state to another with the "Next" and "Previous" buttons, is also implemented in the App component, through two methods entitled "getNext" and "getPrevious". These are triggered when the "Next" and "Previous" buttons, respectively, are pressed by the user.

Table 5.1: Table with the key state variables of the App component and their contents

State variable	Meaning
instructionDisplayed	String representing the executed program's instruction that is currently being displayed on the platform
instructionFlow	Array of numbers representing the order in which the instructions are executed
savedCPUStates	Array containing all CPU states originated from execution (one for each instruction)
savedRelevantLines	Array containing all relevant lines originated from execution (one for each instruction)
savedCriticalPath	Array containing all critical paths originated from execution (one for each instruction)
cpuState	Array of all CPU "Component" objects of an instruction, containing information about their inputs and outputs
cpuIndex	Index of the instruction currently being displayed on the platform. Always starts on the last instruction of the assembly code written
relevantLines	Array of all CPU connections considered relevant for a specific instruction
criticalPath	Array of all CPU connections that make up the critical path for a specific instruction
registerValues	Array representing the current state of the register file, where each element is an object with the register ID and its value.

The state variable "cpuIndex" is incremented in the "getNext" method and reduced in the "getPrevious", by exactly 1 for each button press. These methods also update the other relevant state variables (the instruction displayed, the CPU state, the registers, the relevant lines and critical path) according to this index, making sure that the interface is consistent.

5.3 Client-server communication

When the user interface's implementation is complete, the program is ready to connect with the backend, sending the requests necessary for execution. As was explained previously in section 4.4 of Chapter 4, a web server was created using the Node.js framework Express, along with an API with endpoints defined for each possible user request, shown in table 4.5.

Having this web server logic already set up, the frontend layer only needs to be able to fetch the desired data from the server. In order to do this, the client must communicate with the server through HTTP requests.

This connection between frontend and backend is accomplished by using Axios, a promise-based HTTP client for Node.js and the browser. In section 4.4 of the previous chapter, the way the server handles requests was explained, highlighting the importance of the Express framework. Axios and Express ultimately work together to fully connect client and server, with Axios handling the HTTP requests sent by the client.

An usage example of Axios is shown in listing 5.4, which represents a HTTP "POST" request for an instruction's machine code. The Axios method "post" is called with the server URL used for the request, BASE_URL + ":3001/readInstruction/", (the route is "/readInstructions" and the server runs on port 3001), sending along the instructions in an object. The server converts the instructions into machine code, sending them back as a JSON object, which is then stored in the state variable "machineCodes" using "setMachineCodes".

```
axios.post(BASE_URL + ":3001/readInstruction/", {
      instructions: tempIns,
    }).then((instructionCodes) => {
      props.setMachineCodes(instructionCodes.data);
    });
```

Listing 5.4: Axios HTTP "POST" request used in the assembly of instructions

Axios also supports the "GET" request method, whose structure is essentially the same as a "POST", except it does not send any data on the body of the request. An example of the Axios "get" method was provided in the previous subsection.

5.4 Concluding Remarks

Throughout this chapter, the platform's user interface was described, following the possible interactions of a student exploring the platform. In the first section, the platform's features were explained in detail, alongside images of the website, providing an example of both the unicycle and the pipelined CPU's executions. Afterward, the parts of the interface implementation process deemed the most relevant and complex were presented, including how the datapath's visual representation was conceived and connected with the backend, and how the CPU states are defined and explored by the user. Finally, an explanation of how the client and the server communicate through HTTP requests is provided. This chapter closes the process behind the implementation of the platform, which was then tested by a group of students through a case study, whose results and discussion are presented in the next chapter.

Chapter 6

Validation

In this chapter, the process behind validating the platform with its intended audience is described, starting, in the first section, with an overview of the method used, explaining why it was chosen and deemed appropriate for validation. The user survey sent to students is then described in detail, and, in the next section, a discussion of the results obtained is provided, analysing each question individually, closing with a summary of the qualitative feedback given by users.

6.1 User Testing

Throughout the previous chapters, the implementation of a platform capable of providing an interactive learning environment for CA students was described in detail. The developed platform, however, in order to completely fulfill its intended objectives, must be validated by its intended audience. This audience, in the case of *eduARM*, is comprised of Informatics and Electrical Engineering students who are learning CA. It is only possible to understand the true value of the platform when it is tested by students.

The development of the unicycle version of the platform was concluded in August, amidst the summer holidays, which completely excluded testing the platform with students currently learning CA as an option for validation. Alternatively, the platform can be validated by ex-students of CA, as these students had the course as well and are thus capable of measuring how useful the tool would be for their studying, had they access to it during the course.

This group of students was then chosen for testing the platform. Taking into consideration that validation would be done during student holidays, adherence is expected to be low. Another issue would arise if the testing process was very time consuming and unappealing, for instance, through face-to-face usability tests, as students would be much less likely to participate.

The method deemed the most appropriate was thus validating the platform through a survey, sent to students by e-mail. This form should take at most 10 minutes to complete, otherwise there is a chance of its participants losing interest halfway. Besides collecting user feedback about the platform's usability, the survey should present a small exercise for students to solve using the platform, allowing them to fully explore its features and essentially simulating how it could be used during classes.

A user survey was then created using Google Forms and sent to FEUP's Informatics and Electrical engineering students that previously frequented the CA course.

6.1.1 Survey

The form sent to the students, fully included in appendix A of this document, opens with a small contextualization of the work and explanation of the study's objectives, being followed by a total of 17 questions grouped into three different sections.

The first group of questions relates to the background and general vision of the students on the CA course. The purpose of this survey section was attempting to understand what difficulties students felt during the course, and how challenging the CPU and its datapath are to comprehend. The students were asked the following questions:

- Q1. On a scale of 1 to 10, how would you rate the difficulty of the AC course unit?
- Q2. On a scale of 1 to 10, how difficult was it to understand the CPU datapath, its components and overall behaviour?
- Q3. Do you think having access, during classes, to an interactive platform that simulates the CPU's behaviour, would have improved your understanding of these topics?

In the next survey section, the actual testing of the platform follows, starting with some small steps for the students to follow in order to efficiently use the platform, such as a reminder that the user must have access to FEUP's VPN, and a link to a small guide of the tool, included in the appendix A of this document, which was strongly suggested for the user to read. An assembly program incorporating all the instructions supported by the platform is provided, and the user is asked to paste it in the interface's assembly area, as well as setting specific values on the register file. A total of 8 simple questions about the program's execution are then presented, each one allowing the user to explore a specific feature of the tool:

- Q4. Compile the program. What is the machine code of the load instruction?
- Q5. Execute the program. What is the value of register X0?
- Q6. What position in memory has been filled, and by which value?
- Q7. What is the hexadecimal value of register X0?
- Q8. Click on the "CPU" tab and examine its components by hovering the cursor over them. Check the Control component. What is its input value on the conditional branch instruction, in hexadecimal?

- Q9. Check the PC component. By how much was the program counter incremented in the unconditional branch instruction?
- Q10. Which of these lines is NOT relevant to the add instruction?
- Q11. Check the load instruction. What is the critical path of the CPU?

The third and final part of the form consists of an area for students to offer their feedback, both quantitative and qualitative, on the tested platform. Students are thus asked to answer 5 questions:

- Q12. On a scale of 1 to 5, how would you qualify the platform, regarding how easy it is to use and understand?
- Q13. On a scale of 1 to 5, how much do you think having access to this platform during AC classes would have helped you to better understand the concepts lectured?
- Q14. On a scale of 1 to 5, how much do you think using this platform during AC classes would have made learning more enjoyable for you?
- Q15. On a scale of 1 to 5, how useful do you think the platform would be for your studying outside the class (for example, for exams)?
- Q16. On a scale of 1 to 5, how much do you think having access to this platform would have improved your final grade?
- Q17. Would you have used this platform, were it available when you had AC?

The form closes with a field for qualitative feedback, where students could freely write their opinions or any bugs they might have found. The results obtained in the survey, along with an analysis of each question, are provided in the next section.

6.2 Analysis of feedback

A total of 15 students participated in this study, with 9 of them also providing qualitative feedback and reporting bugs. Since the form was sent during summer break, and the study is already limited to a specific group of students, adherence was low. Nonetheless, the feedback received was valuable, and is reported along this section.

6.2.1 CA Background

The first question of the survey, Q1, attempts to understand how students perceive the CA course by asking them to measure its difficulty. The results obtained were mixed, with the majority of participants (53,4%) claiming the course has either a difficulty level of 6 and 7 out of 10, which implies the course unit is considered mildly challenging by students.

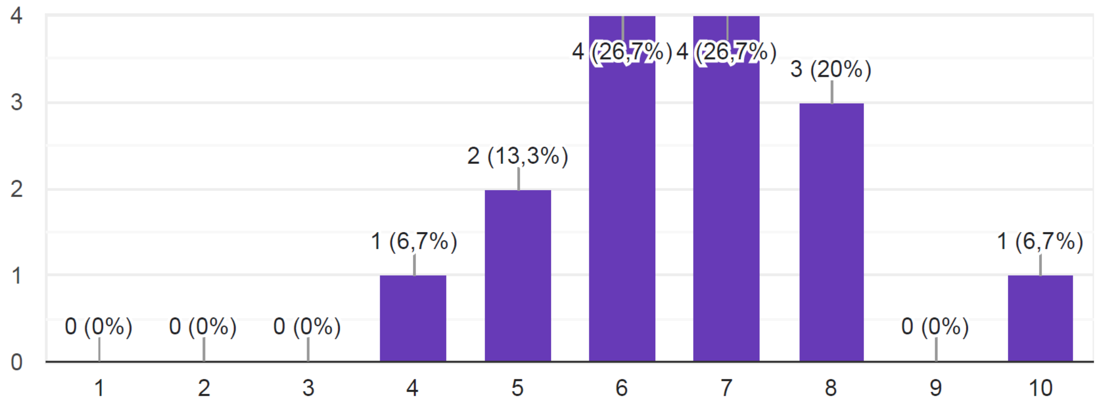


Figure 6.1: Results of Question 1 (Q1)

The second question, Q2, asks students to evaluate how difficult the CPU's concepts and behaviour are to understand. The results obtained were diverse, similarly to question 1, with most participants (80%) placing this subject's difficulty level between 6 and 8, thus implying the majority of CA students found the CPU rather difficult to understand.

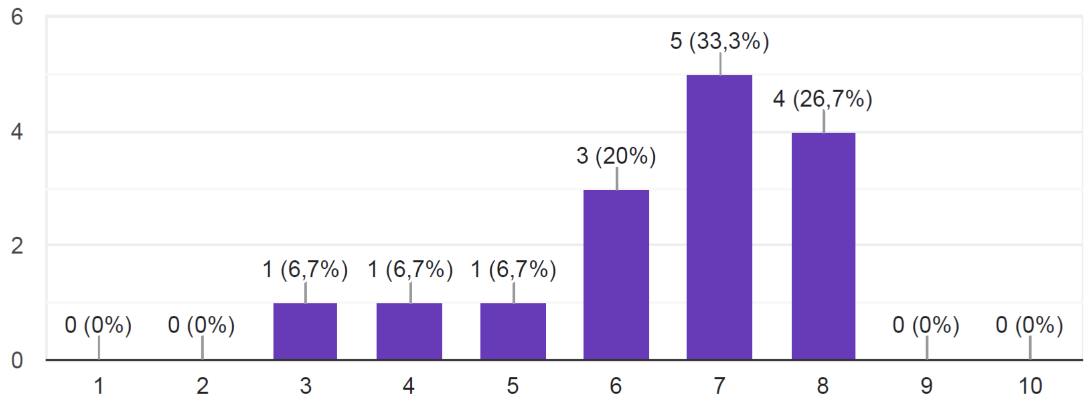


Figure 6.2: Results of Question 2 (Q2)

The final question in the form's first section is a multiple choice question, with 3 options, asking students whether they believe having access to a platform capable of simulating the CPU's behaviour would have helped them better understand these topics. Every participant agreed such a platform would be beneficial, with 100% choosing the option "Yes".

6.2.2 Platform testing with an exercise

The next section in the form is dedicated to a simple exercise with 8 questions that enables the participant to explore each key feature of the platform. The results obtained in this part are not as significant, since they mostly evaluate whether the student chose the right answer or not, which could depend on a variety of factors. For instance, a wrong answer could mean the user did not understand the question correctly, could not follow the steps necessary in the platform in order to get the result, or simply did not want to proceed with the exercise.

Nonetheless, some wrong answers can be related to bugs or parts of the interface that are not as clear. Some qualitative feedback given, and presented in subsection 6.2.4, could explain why some questions have worse results than others.

The first question of the exercise, which starts at question 4, asks the user to compile the program and select the correct machine code for the load instruction. In order to answer correctly, the student needs to input the appropriate assembly code and registers given in the form, and simply press the "Compile" button. The "Machine Code" tab will then display the instructions' machine codes, including the requested load instruction. This question is rather easy to follow, and almost every participant (86,7%) chose the correct answer.

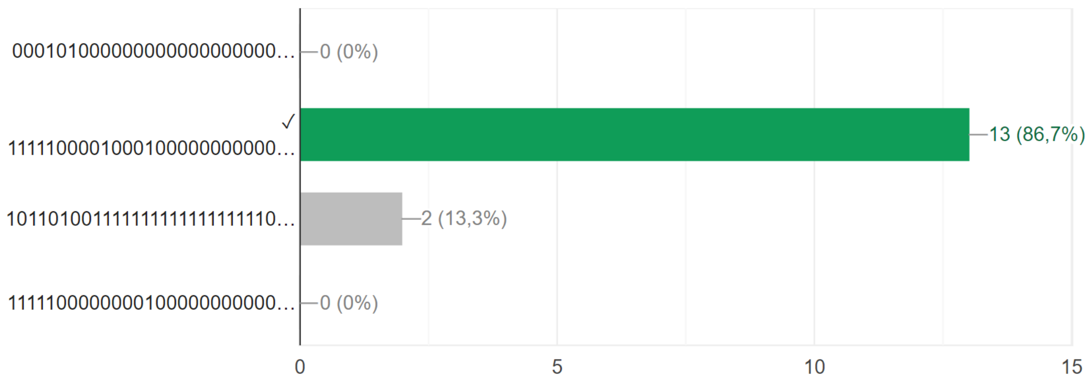


Figure 6.3: Results of Question 4 (Q4)

In order to answer question 5 correctly, the student solely needs to press the "Execute" button and look at the contents of register "X0" on the register file. These steps are rather straightforward and about 80% of students chose the right answer.

For question 6, the student is only required to select the "Data Memory" tab and look at the value on position 64, which was 7. The results show the majority (73,3%) of the participants chose the correct answer.

The next question, Q7, allows the user to see how the platform changes number formats between decimal, binary and hexadecimal. By selecting register "X0" and inspecting the "HEX" input area at the bottom of the screen, 80% of the students chose the right answer of "fffffffffffffff3".

Question 8 requires a few more steps when compared with the previous ones. The user is asked to click on the "CPU" tab in order to analyse the datapath's components. Specifically, students must change the number format to hexadecimal, through pressing the "Change format" button, and hover the cursor over the "Control" component, checking its input value. This question encourages the user to explore the CPU datapath and inspect its values, and 86,7% of participants chose the right answer.

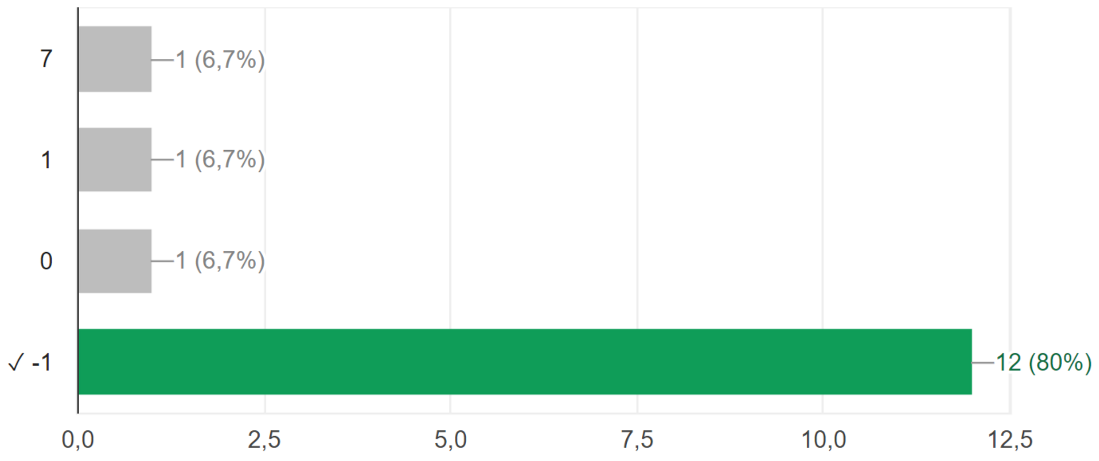


Figure 6.4: Results of Question 5 (Q5)

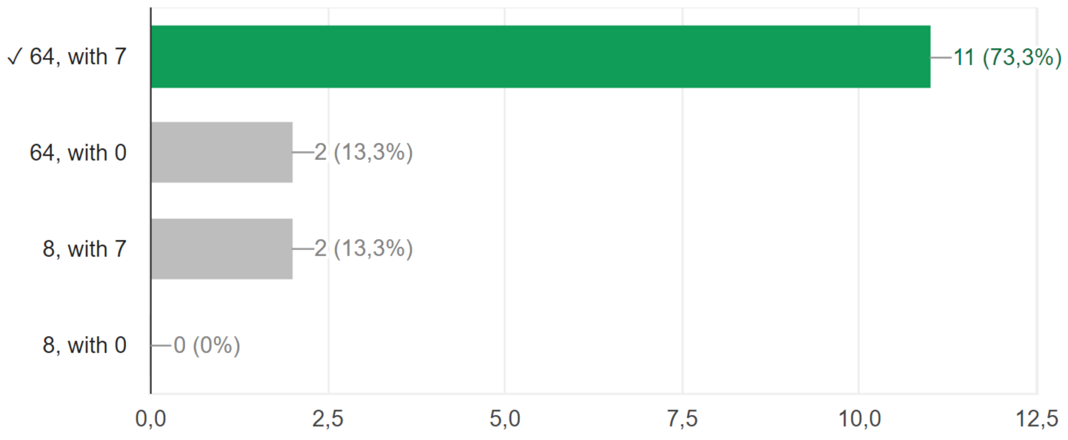


Figure 6.5: Results of Question 6 (Q6)

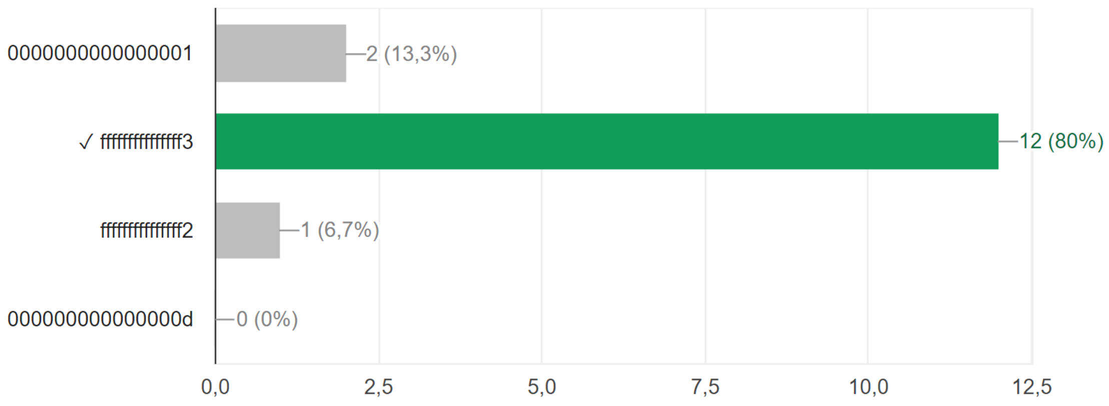


Figure 6.6: Results of Question 7 (Q7)

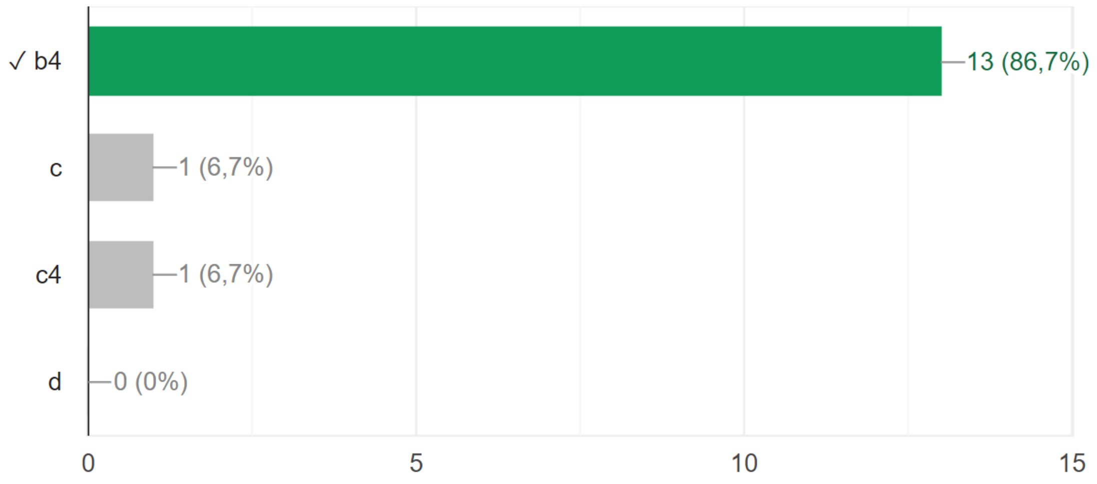


Figure 6.7: Results of Question 8 (Q8)

Question 9 also requires the user to analyse the CPU datapath, but, in this case, for another instruction that is not the one displayed on execution. In order to answer correctly, students must use the "Previous" button to reach the unconditional branch instruction and inspect the PC component to see how its value changes. For this specific program, the PC value was not incremented by 4, but rather by 8, since the current instruction is a jump. Answering correctly required cycling through CPU states, and 73,3% of participants managed to do so.

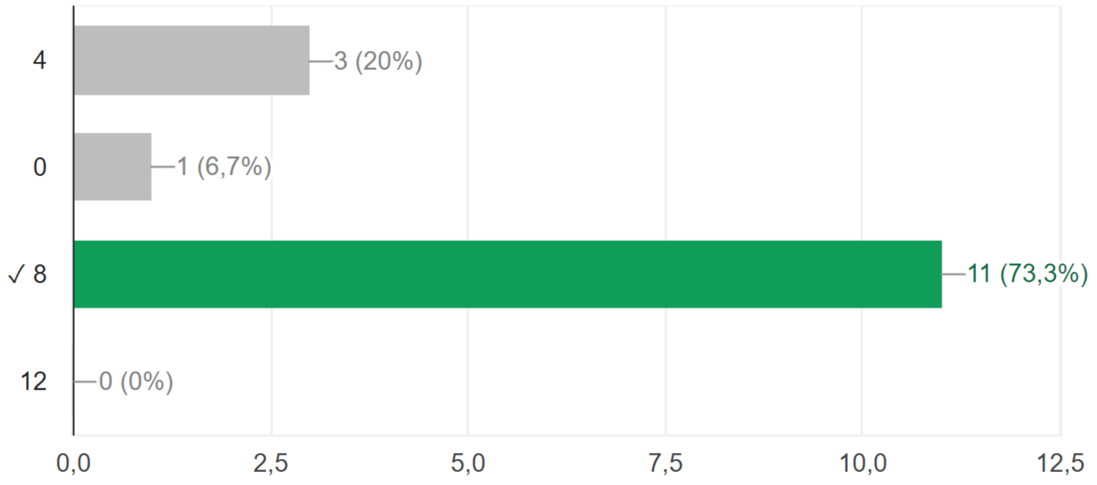


Figure 6.8: Results of Question 9 (Q9)

The next question, Q10, explores the "Relevant Lines" feature of the platform. The student must cycle through states again, using the "Previous" and "Next" buttons, and analyse the "add" instruction's relevant lines. The correct answer, which 73,3% of students selected, is the only option that represents a line not colored black, but rather grey, thus considered not relevant.

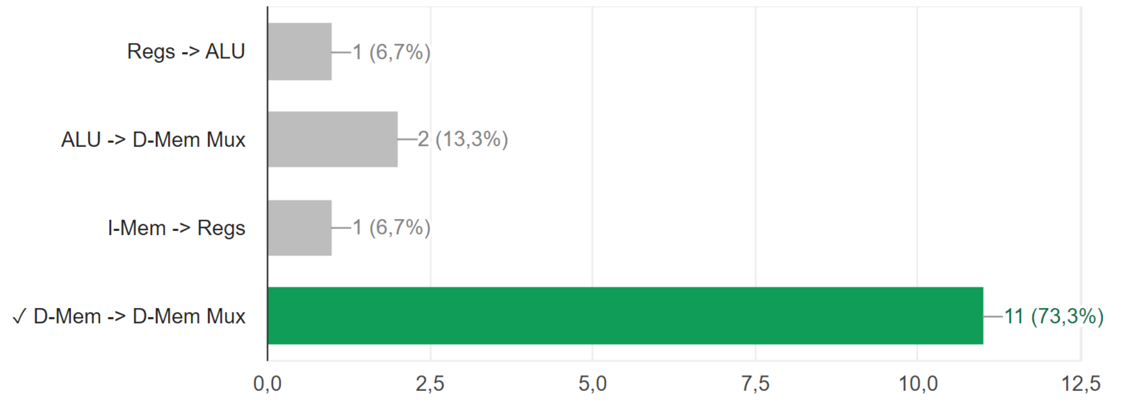


Figure 6.9: Results of Question 10 (Q10)

The final question of this exercise, Q11, concerns the critical path of an instruction. The user must, one more time, cycle through CPU stages and reach the "load" instruction. The correct critical path is found by clicking the "Performance" button and checking which lines are colored red, which 86,7% of the students could successfully accomplish.

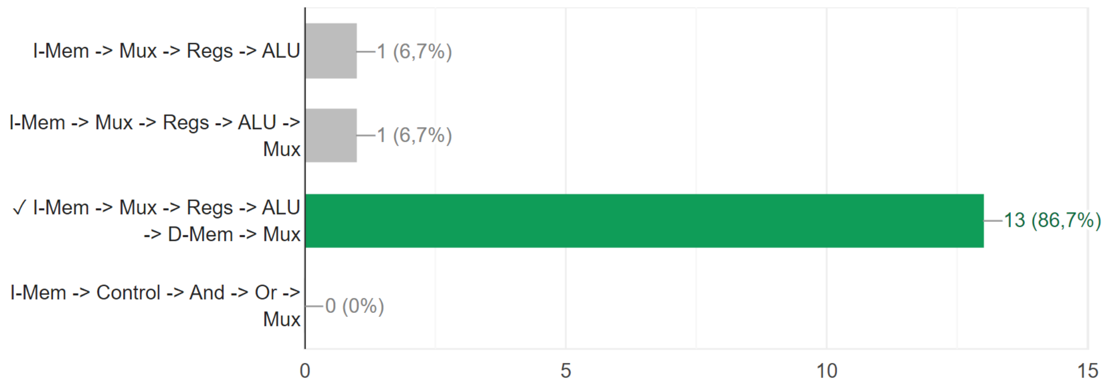


Figure 6.10: Results of Question 11 (Q11)

Through this exercise, the participants were given the opportunity to interact with the platform and study its features, with each correct answer requiring the user to follow specific steps on the interface. Overall, the majority of students were capable of doing so, with an average of 80% answering the questions correctly.

6.2.3 Quantitative feedback

The first question of the feedback section asks students to quality the platform regarding its usability and how intuitive it is. On a scale of 1 to 5, results were considered positive when equal to 4 or above, which was the case of Q12, with 80% of the participants giving a 4 or the maximum score of 5 to the interface. A few users reported, in the qualitative feedback, that the interface could be refined, which may justify the other 20%.

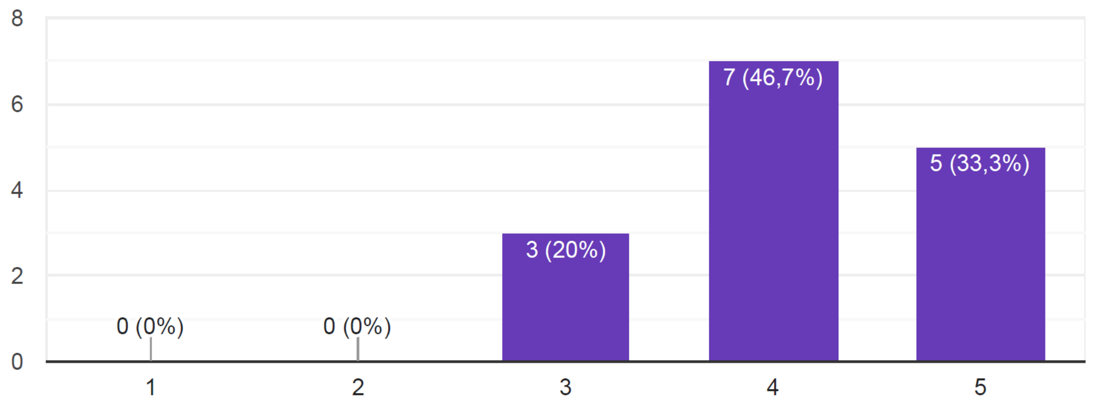


Figure 6.11: Results of Question 12 (Q12)

Question 13's purpose is to understand exactly how useful the platform is for CA classes, asking students to measure how much they think having access to the tool would have improved their comprehension of the lectured topics. The scores obtained were, once again, very positive, as 80% of the students chose the highest option.

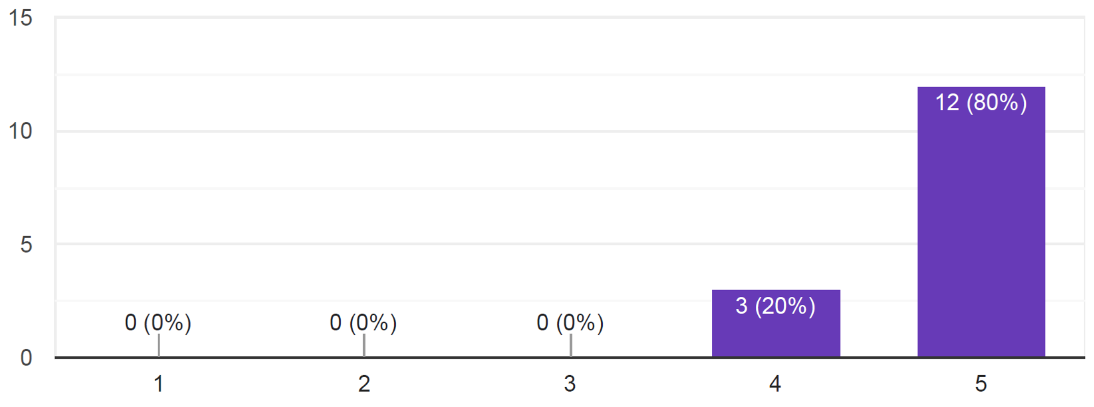


Figure 6.12: Results of Question 13 (Q13)

Question 14 aims to analyse how this platform would make learning more enjoyable for students. Most students (93,4% chose either option 4 or 5) seem to believe the platform would greatly improve their learning experience.

The next question, Q15, asks students to measure how useful the platform would be for their studying outside the class. For instance, the tool could be used by students while doing exercises at home or to study for an exam. The majority of the participants agree that having access to this platform would benefit their studying, with 86,7% selecting either option 4 or 5.

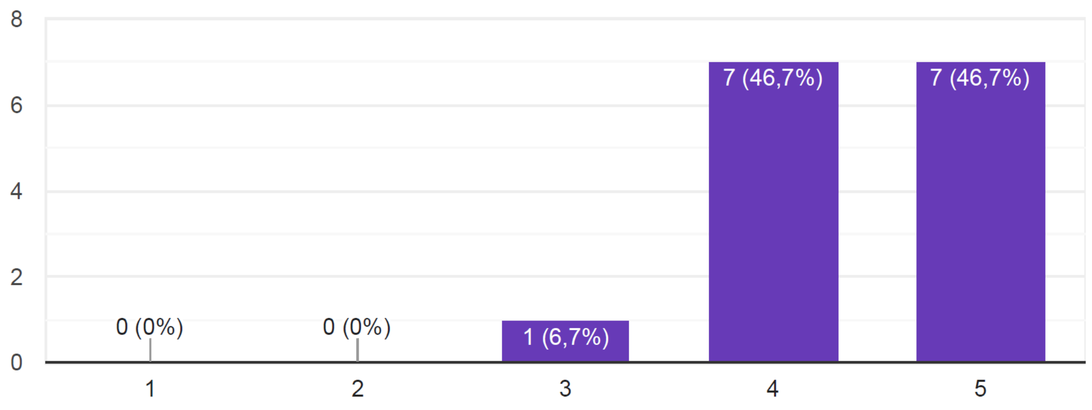


Figure 6.13: Results of Question 14 (Q14)

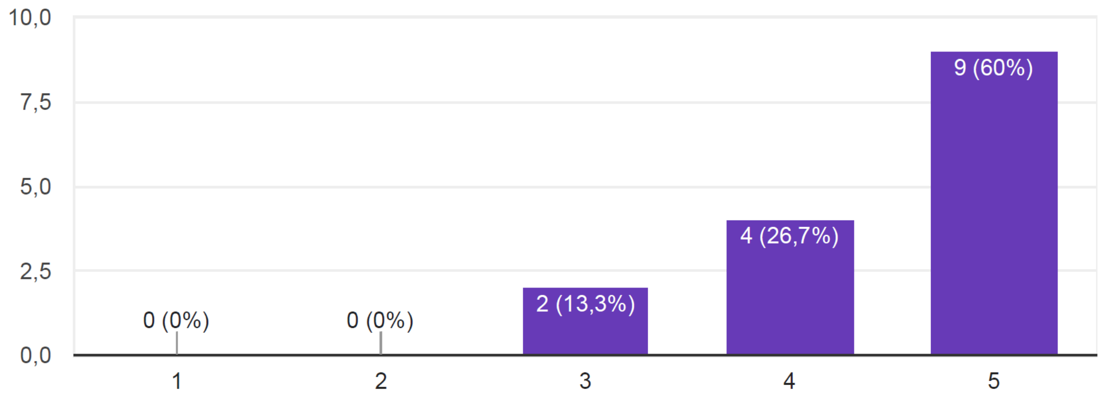


Figure 6.14: Results of Question 15 (Q15)

Question 16 inquires how much the students think having access to the platform would have improved their final CA grade, when they frequented the course. Each student's response will naturally be influenced by how low or high their final grade was - nevertheless, more than half (60%) of the answers were positive, with scores of either 4 or 5.

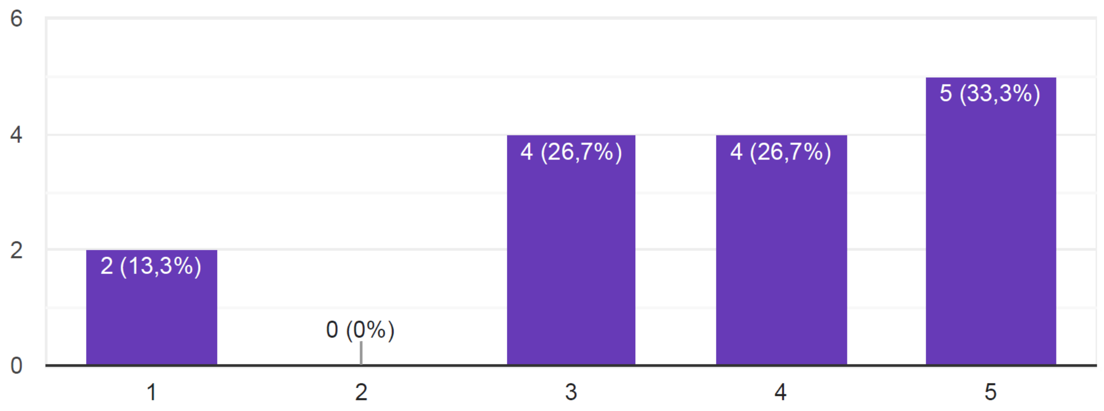


Figure 6.15: Results of Question 16 (Q16)

The final question of the survey is very straightforward, simply asking the participants whether they would use the tested platform, were it available when they had the CA course unit. Results were extremely positive, with 100% of the students selecting option "Yes".

6.2.4 Qualitative feedback

The user survey closes with a text area for students to provide any additional comments, such as positive or negative qualitative feedback, reporting bugs, or suggestions for improvements.

A total of nine comments were made, with most consisting of valuable feedback on aspects of the platform that could be improved. Among those, common observations include:

- Some critical paths were not clear and highlighted unnecessary parts of the datapath;
- The introduction of values to the register file could be improved, as it is necessary to press the "Enter" key to actually update the register, which is considered a bothersome extra step;
- The assembly code compilation and its debugging could be enhanced, making more evident which errors occured in the code

6.3 Concluding Remarks

In this chapter, how the developed solution's validation was carried out was described in detail, showcasing its obtained results. Overall, student feedback was highly positive, with the participants finding the platform adequate and useful for CA classes. Nevertheless, these results could be improved if student adherence to the survey was higher, whose low values were a consequence of the timing chosen for validation.

Another aspect that would reinforce the platform's validation is the inclusion of the pipelined CPU version. Due to this version not yet being completed at the time of user testing, the survey only contemplates the unicycle version of the CPU. Nonetheless, this brought forth valuable feedback on how to improve the interface, which helps towards refining the platform to become its best possible version.

Chapter 7

Conclusions

By developing the platform described throughout the previous chapters, this thesis aspires to be of use to both computer architecture teachers and students in the academic community, as well as presenting a source of innovation in the field of ARMv8 educational tools capable of simulating the CPU.

Considering the use of simulators is believed to greatly improve students' understanding [18], this work expects to aid in the comprehension of subjects that would otherwise be difficult to understand through more conventional teaching methods. Thus, this platform offers an interactive approach to learning the CPU's components and overall behaviour, for both its unicycle and pipelined versions. Students can freely explore the CPU datapath in each stage of execution and understand how a specific instruction impacts its components' values, write simple assembly programs and debug them by inspecting the simulator's results, and check component latencies and the critical path of an instruction.

Through the course of this document, the state of the art on tools for CA education could be analysed in detail, aiding in identifying their strong and weak points, which in turn made it possible to specify requirements for the platform to develop. The process behind its implementation was delineated, focusing first on the tool's backend layer, which handles the CPU simulation's logic and every server-side operation, and then on the client-sided frontend layer, representing the user interface. The solution was then validated through a study with the participation of past CA students, whose feedback aided in highlighting elements of the platform that can still be improved.

Additionally, a paper based on the work developed was written, which will soon be submitted to an international conference on education.

The successful adoption of this platform in computer architecture classes would further emphasize that the goals of this thesis were accomplished, and that CA students have access to an intuitive and adequate platform capable of helping them further comprehend the topics lectured in classes and, ultimately, positively contributing to their academic success.

7.1 Future Work

Despite what was accomplished with the development of this platform, several aspects of its interface could be improved and new, valuable features incorporated in the future. Thus a selection of items were considered for planning the next possible work steps:

- Improve the platform's overall usability, which includes the suggestions given by students described in subsection 6.2.4, such as improving the register values' input method, the highlighting of the critical path, and assembly code debugging.
- Provide additional functionality to the latencies feature of the unicycle CPU, allowing the user to edit a specific component's latency and explore what impact it has on the CPU's critical path. The platform's critical path calculations would thus become dynamic, changing according to the values set by students on each component, and allowing them to understand how altering their latencies can improve or worsen the CPU's performance.
- Validate the pipeline version of the CPU with CA students, through a case study similar to the described in Chapter 6.

Working on the presented aspects, in the future, would bring additional value to the platform, which, as a result, would be able to further contribute towards improving the learning experiences and comprehension of CA students all over the world.

References

[1] Rashmi Agrawal, Sahan Bandara, Alan Ehret, Mihailo Isakov, Miguel Mark, and Michel A Kinsy. The brisc-v platform: A practical teaching approach for computer architecture. In *Proceedings of the Workshop on Computer Architecture Education*, pages 1–8, 2019.

[2] Salman Arif. Visual - a highly visual arm emulator. https://salmanarif.bitbucket.io/visual/index.html, 2015. Accessed 2022-08-04.

[3] ARM. Graphical micro-architecture simulator. https://www.arm.com/zh-TW/resources/education/education-kits/legv8, 2021. Accessed 2022-09-02.

[4] Axios. Axios. https://axios-http.com/, 2022. Accessed 2022-09-12.

[5] Bootstrap. Bootstrap - the most popular html, css, and js library in the world. https://getbootstrap.com/, 2022. Accessed 2022-08-01.

[6] Irina Branovic, Roberto Giorgi, and Enrico Martinelli. Webmips: a new web-based mips simulation environment for computer architecture education. In *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, pages 19–es, 2004.

[7] Diego Camarmas-Alonso, Félix García-Carballeira, Elías Del-Pozo-Puñal, and Alejandro Calderón Mateos. A new generic simulator for the teaching of assembly programming. In *2021 XLVII Latin American Computing Conference (CLEI)*, pages 1–9. IEEE, 2021.

[8] Express. Express - node.js web application framework. https://expressjs.com/, 2022. Accessed 2022-08-04.

[9] Félix García-Carballeira, Alejandro Calderón-Mateos, Saúl Alonso-Monsalve, and Javier Prieto-Cepeda. Wepsim: an online interactive educational simulator integrating microdesign, microprogramming, and assembly language programming. *IEEE Transactions on Learning Technologies*, 13(1):211–218, 2019.

[10] Khushi Gupta and Tushar Sharma. Changing trends in computer architecture : A comprehensive analysis of arm and x86 processors. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, pages 619–631, 06 2021.

[11] Benjamin Landers. Rars - risc-v assembler and runtime simulator. https://github.com/TheThirdOne/rars, 2017. Accessed 2022-10-05.

[12] James R Larus. Spim s20: A mips r2000 simulator. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1990.

[13] Node.js. Node.js. https://nodejs.org/en/, 2022. Accessed 2022-08-04.

[14] Bruno Nova, Joao C Ferreira, and António Araújo. Tool to support computer architecture teaching and learning. In *2013 1st International Conference of the Portuguese Society for Engineering Education (CISPEE)*, pages 1–8. IEEE, 2013.

[15] npm. npm. `https://www.npmjs.com/`, 2022. Accessed 2022-08-01.

[16] Faculty of Engineering of the University of Porto. Computer architecture course syllabus. `https://sigarra.up.pt/feup/en/UCURR_GERAL.FICHA_UC_VIEW?pv_ocorrencia_id=484399`, 2021. Accessed 2022-08-04.

[17] David A Patterson and John L Hennessy. *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann, 2016.

[18] PWC Prasad, Abeer Alsadoon, Azam Beg, and Anthony Chan. Using simulators for teaching computer organization and architecture. *Computer Applications in Engineering Education*, 24(2):215–224, 2016.

[19] React. React - a javascript library for building user interfaces. `https://reactjs.org/`, 2022. Accessed 2022-08-04.

[20] Redis. Redis. `https://redis.io/`, 2022. Accessed 2022-09-11.

[21] Missouri State University. Mars - mips assembly and runtime simulator help contents. `http://courses.missouristate.edu/kenvollmar/mars/Help/MarsHelpIntro.html`, 2014. Accessed 2022-09-14.

[22] Kenneth Vollmar and Pete Sanderson. Mars: an education-oriented mips assembly language simulator. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 239–243, 2006.

Appendix A

Validation Material

This appendix includes the material used for validating the platform with users, including a manual of the platform, provided to help the participants with testing, followed by the complete survey, created with Google Forms.

A.1 User Manual

When you open the website, you will be shown a screen with two main divisions.

CPU

The CPU datapath is displayed on the left side, with its components and corresponding connections. In this visual representation, control lines and related components are colored blue. The diagram can also be repositioned and zoomed in and out as desired. On top of the diagram, two more tabs besides the CPU one can be selected: "Assembly" and "Machine Code".

Assembly

By clicking on the "Assembly" tab, a grey area will appear where you can write the assembly code to execute. You are free to write as many instructions as you want but should always separate them with a newline. If you already have code written in an external text file, you can import it to the program using the "Load" feature on the top left area of the screen. Code that you write on the text area can also be saved in your device: by clicking the "Save" option, next to "Load", a text file with your code will automatically be downloaded. Six basic instructions are supported by the program:

- Arithmetic-logical: add, sub, and, orr
- Memory: ldur, stur
- Branches: b, cbz

The written instruction set can then be compiled and executed - but first, any used registers can be initialized, unless you want their values to be zero.

Registers

On the screen's rightmost area, the content of each register, from X0 to X31, can be changed by clicking on each input area, typing the desired number in its decimal format, and pressing enter. When you press enter, the hexadecimal and binary representations of the register will be displayed in the two boxes at the bottom of the page. The content of these boxes can also be filled in by the user, and the corresponding conversions will be displayed.

After writing the code and changing the necessary registers, the program is ready to be executed. In order to do this, you can press the "Compile" button, followed by the "Execute" button below the register area. Next to them, you will also find the "Previous" and "Next" buttons, which allow you to cycle through instructions in a set to see each step and its changes to the CPU in more detail.

Machine Code

After compiling a set of instructions, the program will display each instruction's machine code in the "Machine Code" tab next to "Assembly". Each instruction type (arithmetic-logical, memory load and store, branches) has a specific number of bits representing the data the CPU requires for execution. After executing the program, this structure can be seen below the "Execute" button area.

Datapath

Now that the program has been compiled and executed, the datapath will showcase, for each instruction, the internal behavior of the CPU. Connections that are considered relevant for each instruction, that is, that carry any sort of relevant data, are painted black. You can hover over any components to see their inputs and outputs' current values. If you want to see these values in either their binary or hexadecimal formats, you can do this by selecting the desired representation with the "Change format" button on the top left side of the screen. You can also see the CPU's critical path if you click on the "Performance" button. These lines will be painted red.

Memory

In case you use any load or store instructions, its changes to the memory's contents can be seen in the "Memory" tab, next to "Registers".

A.2 User Survey

eduARM: Web platform to support the teaching and learning of the ARM architecture

eduARM is a web platform currently in development for a Master's thesis in Informatics and Computer Engineering. This tool's main goal is to give Computer Architecture students an intuitive way of learning the ARMv8 architecture lectured in classes.
This survey consists of three parts: first, some questions about your background in Computer Architecture (AC) will be asked, in order to understand how students perceive the subject. Then, you will be asked to complete a series of simple tasks using the platform. You will then be able to give your feedback in the final section.

*Obrigatório

Background

Please answer the following questions regarding your experience with the Computer Architecture course unit, and the difficulties you may have faced in it.

1. On a scale of 1 to 10, how would you rate the difficulty of the AC course unit? *

Marcar apenas uma oval.

	1	2	3	4	5	6	7	8	9	10	
Very easy	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	Very challenging

2. On a scale of 1 to 10, how difficult was it to understand the CPU datapath, its components and overall behaviour? *

Marcar apenas uma oval.

	1	2	3	4	5	6	7	8	9	10	
Very easy	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	Very challenging

3. Do you think having access, during classes, to an interactive platform that simulates the CPU's behaviour, would have improved your understanding of these topics? *

Marcar apenas uma oval.

- ◯ Yes
- ◯ No
- ◯ Maybe

Platform testing

Before exploring the platform, please take a look at its manual. Its key parts are marked in bold.

Once you're ready, please open the platform on:
eduarm.fe.up.pt:3000
Make sure you have FEUP's VPN on. The platform might not work in Google Chrome - if this happens, please try using Firefox.
If the CPU datapath is not correctly adjusted to your screen size, don't forget you can zoom in and out as desired.

You can now start testing the platform by using this example:

```
add x1, x2, x3
b label
and x1, x2, x3
label: orr x1, x2, x3
stur x7, [x2, #64]
ldur x1, [x2, #64]
sub x0, x0, x15
cbz x0, -2
```

You should paste this in the “Assembly” text area. For the program to work properly, please fill in the following registers (register -> value):
X7 -> 7, X15 -> 1, X0 -> 1.

!!! Don't forget to press enter while typing in the values !!!

IMPORTANT: If the program returns all 0s, this is most likely because you didn't enter the register values correctly! You must press enter each time you change one of the values.

4. Compile the program. What is the machine code of the load instruction? *

Marcar apenas uma oval.

- 00010100000000000000000000000010
- 11111000010001000000000001000001
- 10110100111111111111111111000000
- 11111000000001000000000001000111

5. Execute the program. What is the value of register X0? *

Marcar apenas uma oval.

- 7
- 1
- 0
- -1

6. What position in memory has been filled, and by which value? *

Marcar apenas uma oval.

- 64, with 7
- 64, with 0
- 8, with 7
- 8, with 0

7. What is the hexadecimal value of register X0? *

Marcar apenas uma oval.

- 0000000000000001
- fffffffffffffff3
- fffffffffffffff2
- 000000000000000d

8. Click on the “CPU” tab and examine its components by hovering the cursor over them. *
Check the Control component. What is its input value on the conditional branch instruction, in hexadecimal?

Marcar apenas uma oval.

- ⬭ b4
- ⬭ c
- ⬭ c4
- ⬭ d

9. Check the PC component. By how much was the program counter incremented in the unconditional branch instruction? *

Marcar apenas uma oval.

- ⬭ 4
- ⬭ 0
- ⬭ 8
- ⬭ 12

10. Which of these lines is NOT relevant to the add instruction? *

Marcar apenas uma oval.

- ⬭ Regs -> ALU
- ⬭ ALU -> D-Mem Mux
- ⬭ I-Mem -> Regs
- ⬭ D-Mem -> D-Mem Mux

11. Check the load instruction. What is the critical path of the CPU? *

Marcar apenas uma oval.

- ⬭ I-Mem -> Mux -> Regs -> ALU
- ⬭ I-Mem -> Mux -> Regs -> ALU -> Mux
- ⬭ I-Mem -> Mux -> Regs -> ALU -> D-Mem -> Mux
- ⬭ I-Mem -> Control -> And -> Or -> Mux

Feedback

Please give your honest feedback about the platform and your experience with it.

12. On a scale of 1 to 5, how would you qualify the platform, regarding how easy it is to use and understand? *

Marcar apenas uma oval.

	1	2	3	4	5	
Not intuitive at all	◯	◯	◯	◯	◯	Very intuitive

13. On a scale of 1 to 5, how much do you think having access to this platform during AC classes would have helped you to better understand the concepts lectured? *

Marcar apenas uma oval.

	1	2	3	4	5	
Not helpful at all	◯	◯	◯	◯	◯	Very helpful

14. On a scale of 1 to 5, how much do you think using this platform during AC classes would have made learning more enjoyable for you?

Marcar apenas uma oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	A lot

15. On a scale of 1 to 5, how useful do you think the platform would be for your studying outside the class (for example, for exams)? *

Marcar apenas uma oval.

	1	2	3	4	5	
Not useful at all	◯	◯	◯	◯	◯	Very useful

16. On a scale of 1 to 5, how much do you think having access to this platform would have improved your final grade? *

Marcar apenas uma oval.

	1	2	3	4	5	
No change at all	◯	◯	◯	◯	◯	Greatly improved

17. Would you have used this platform, were it available when you had AC? *

Marcar apenas uma oval.

◯ Yes
◯ No

18. Do you have any additional feedback on the platform?
You can also identify eventual bugs that you have found or give any suggestions for improvement.

Thank you for your response!