Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Porting of the μMPS3 Educational Emulator to RISC-V

Relatore:
Chiar.mo Prof.
Renzo Davoli

Presentata da:
Gianmaria Rovelli

I Sessione
Anno Accademico 2022/23

Sometimes it is the people
no one can imagine anything of
who do the things no one can imagine.
Alan Turing

Abstract

Teaching operating systems to students has always been a challenging task for lecturers. To address this issue, educational architectures, and emulators have been developed to offer students an easier entry point into the subject matter.

In this context, Virtual Square has created a simplified architecture named μMPS, along with its emulator. This simplified architecture serves as a starting point for students to familiarize themselves with the operating systems, allowing them to gain foundational knowledge and understanding in the field.

Virtual Square has developed a range of software tools that emulate the MIPS and ARM RISC architectures. In recent years, RISC-V has emerged as a new and interesting RISC architecture. Known for its open-source nature, RISC-V offers a promising instruction set architecture. One of its significant advantages is the ability for students and professionals to directly study and customize the architecture, making it a valuable resource for learning and exploration. This thesis aims to expand Virtual Square's software family by incorporating the new RISC-V architecture, inheriting the skeleton of the previous project μMPS3.

Sommario

Insegnare i sistemi operativi agli studenti è sempre stato un compito impegnativo per i docenti. Per affrontare questo problema, sono state sviluppate architetture didattiche ed emulatori con l'obiettivo di offrire agli studenti un approccio semplificato a questa materia complessa.

In questo contesto, Virtual Square ha sviluppato, insieme al suo emulatore, un'architettura semplificata chiamata μMPS. Questa architettura è stata progettata con l'obiettivo di fungere da punto di partenza per gli studenti, offrendo loro l'opportunità di familiarizzare con i sistemi operativi, di acquisire conoscenze e comprensione fondamentali in questa materia.

Virtual Square ha sviluppato una gamma di strumenti software che emulano le architetture di tipo RISC: MIPS e ARM. Negli ultimi anni è comparsa RISC-V, una nuova e interessante architettura RISC. Conosciuta per la sua natura open source, RISC-V offre una promettente architettura di istruzioni. Uno dei suoi punti di forza è la possibilità che da a studenti e professionisti di studiare e personalizzare direttamente l'architettura, rendendola una risorsa preziosa per l'apprendimento e l'esplorazione. Lo scopo di questa tesi è espandere questa famiglia di software sviluppata da Virtual Square incorporando la nuova architettura RISC-V, ereditando lo scheletro del precedente progetto μMPS3.

Contents

1 Introduction

This chapter presents a summary of the subject matter, providing the reader with adequate background knowledge to comprehend the Implementation section (see chapter 2). Furthermore, it aims to establish the objective of the thesis.

1.1 State of the Art

Educational emulators represent cutting-edge technology in the education field, providing immersive and interactive learning experiences that bridge the gap between theory and practice. These software applications emulate real-world environments, allowing students to explore and engage with various subjects in a safe and controlled virtual setting. With the advancement of technology, educational emulators have evolved to offer increasingly sophisticated and realistic simulations, revolutionizing the way we teach and learn across multiple disciplines.

Over the years, multiple projects have been initiated to educate individuals about the intricacies of computer architecture. Albeit many researchers have expressed their concern about the difficulty of that task [3], this effort has been acknowledged as a crucial undertaking, as it guarantees continuous advancement and exploration in the development and research realm. Projects worth to be cited are (three of them suggested by C. Yehezkel and W. Yurcik [36]) *EasyCPU* [36], *Little Man Computer* [36, 37], *RTLSim* [36], and *Nand2Tetris* [28]. *EasyCPU* [36] is a simulator designed to emulate a simplified version of the Intel X86 microprocessor. It offers users a user-friendly graphical user in-

terface (GUI) that allows them to edit, assemble, execute, and debug programs. *Little Man Computer* [36, 37] is a web-based simulation tool aimed at providing students with a comprehensive understanding of the internal operations of a CPU, including the fetch-execute cycle. On the other hand, *RTLSim* [36] simulates a non-pipelined processor similar to MIPS, where the user assumes the control unit (CU) role. The last cited project was incorporated as a component of the Computer Architecture examination in the Computer Science course (2020/2021) at the University of Bologna. *Nand2Tetris* [28] is a step-by-step course where participants acquire fundamental computer knowledge, starting from the basic logic gate *NAND* [20] and progressing to developing complex programs such as *Tetris* [29].

Over the years, the complexity of what such projects could achieve increased, even though keeping such works user-friendly and approachable to undergraduate students. For instance, *VisibleZ* [35] is a mainframe architecture emulator that helps students to understand how the mainframe technology works and its inner components, including storage and I/O channels. In *VisibleZ*, users can also define their instructions with custom semantics, broadening their understanding of how computer architectures work. Related to the RISC-V instruction set and this thesis, it is relevant to cite three recent projects: *EmulsiV* [31], *riscvOVPsim* [17], and *µMPS* [15]. *EmulsiV* [31] is a visual simulator based on the minimal [1] subset of the RISC-V instruction set called *Virgule* [31]. In *EmulsiV*, the users, albeit limited by design, can discover how a processor works, compiling a C program, linking it, and executing it, giving them a broad overview of such a process. In addition, *riscvOVPsim* [17] is a RISC-V Instruction Set Simulator (ISS) perfect entry point for learning computer architecture. It provides users with debugging facilities. The students' source code can be compiled easily using a cross-compiler, while the GUI helps the users to see memory and register values. One of the key features of *riscvOVPsim* is the possibility to simulate real Operating Systems (OS) and Real-time operating systems (RTOS). *µMPS* [15] provides users with a user-friendly GUI, debugging features, and virtual memory management, allowing students to develop

[1] "minimal" means only the base instructions generated from a simple C program

their operating system and run it using the emulator bundled with μMPS. μMPS is described in further detail in section 1.2.2.

1.2 Background

This section provides the reader with an understanding of the larger context and offers a concise description of the subjects that have been addressed in the thesis.

1.2.1 RISC-V

RISC-V is an *open standard instruction set architecture* (ISA). The name "RISC-V" stands for "Reduce Instruction Set Computer - Version 5", which means that RISC-V is designed based on the reduced instruction set computer (RISC) principles. RISC-V is available under royalty-free open-source licenses, which means that any firm or user can freely use, modify, distribute, and incorporate code to RISC-V without the need to pay royalties or licensing fees. The RISC-V ISA is a load-store architecture, meaning that the instructions are of two categories: memory access and arithmetic logic unit (ALU) operations.

The principal characteristics of the RISC-V ISA include:

- *Openness and Modularity*: RISC-V is an open standard, meaning anyone can access and contribute to its development. The design of RISC-V allows it to be modular, letting the user select specific instruction sets that best suit their needs;
- *Simplicity*: the instructions provided by the RISC-V ISA are minimal and easy to decode. The core instruction set provides the basic operations, while more complex instructions, such as vector processing is available thanks to the extensions;
- *Multiple Privilege Levels*: RISC-V supports a hierarchical privilege model with multiple privilege models (in μRISC-V just user and machine-level

are used). This allows the implementation of operating systems and virtualization techniques;

- *Scalability*: RISC-V is designed to be used either on embedded devices or high-performance computing systems. It supports 32-bit, 64-bit, and 128-bit words, although the latter is still "not frozen" [2].

- *Software Ecosystem*: due to the open nature of RISC-V, the ecosystem of compilers, development tools, and libraries for that architecture has grown considerably.

The RISC-V ISA has become popular among students and educational institutions as a valuable learning resource thanks several reasons. First of all, the openness of RISC-V allows students to approach it easily, while onlinea documentation fosters hands-on exploration. Although remaining a very expressive architecture, the simplified architecture makes RISC-V a perfect entry point for students new to computer architecture. For instance, on a RISC-V core can be executed Linux [18]. Over the years, the community has grown, and numerous educational content appeared on the Internet, such as courses, tutorials, and community-driven projects.

1.2.2 μMPS

μMPS is an educational computer system architecture that aims to be simple but remains realistic. The μMPS processor implements the MIPS I [3] instruction set, avoiding the need to develop a compiler for it, since the existing MIPS compilers work.

The architecture and its emulator are now at their third version (fourth if we consider MPS, see Preface in [13]): μMPS3, which is not backward compatible with μMPS2 (nor μMPS). The evolution of μMPS has been gradual:

[2] Frozen means that it is being reached a point in the development after which making changes to ISA has become strictier

[3] MIPS R2000/R3000

1. *MPS* [27]: it is the first attempt, but the virtual memory management was unsuitable for undergraduates;
2. *μMPS* [15]: implemented the virtual memory management that better matched the theory of textbooks;
3. *μMPS2* [19]: upgrade of the GUI, multiprocessor support implemented, and debugging features;
4. *μMPS3* [13, 1]: migrating from GNU Autotools [8] to CMake [4], changed exception handlers, migrating the GUI to Qt5, and introduced the new class of flash devices.

All these projects are part of the project family under the umbrella of VirtalSquare [12]. μARM [22] is another project from the same family, that shares the same goals with μMPS, but through a more popular architecture: ARM (used in almost every Android device).

1.2.3 Emulation

Emulation is a powerful technique that replicates the behavior and functionality of one system on another. It enables running software or operating systems designed for a specific platform on different hardware or software environments. Emulation can be achieved through software or hardware approaches.

Software emulation involves creating an emulator that interprets instructions intended for the original hardware and executes them on the host system. This method allows running legacy software on modern computers or virtual machines, extending the lifespan and accessibility of obsolete systems. The emulation of gaming consoles, such as the GameBoy, is a perfect example of software emulation.

Hardware emulation goes beyond software emulation by replicating both the software and physical hardware components of the target system. It requires building specialized hardware platforms that accurately mimic the architecture and behavior of the original system. Hardware emulation is particularly useful

for studying, debugging, and analyzing complex systems at a low level. For instance, the MiSTer project [4] is a popular hardware emulation platform based on FPGA (Field Programmable Gate Array) technology.

Emulation offers students valuable opportunities to explore and understand different computer systems, software, hardware, and programming concepts interactively. By emulating the behavior of different computer architectures, to say one, emulation provides a practical and immersive learning experience. By examining the behavior of emulated systems at the hardware level, students can gain insights into concepts such as instruction execution, memory management, and I/O operations. Emulation platforms enable them to experiment with different instruction sets and architectures, fostering a deeper understanding of computer organization and design, which is one of the goals of μRISCV.

1.2.4 Translation Lookaside Buffer (TLB)

The Translation Lookaside Buffer (TLB) is a relevant component in computer science systems that helps the efficiency of virtual memory management. It serves as a cache for memory address translations, reducing the time and resources required to access data from a physical address.

The TLB consists of an array of entries, with each storing a virtual-to-physical address mapping. Each entry typically contains the virtual page number, the corresponding page number, and other control bits, such as the dirty bit. The TLB is organized as a content-addressable memory (μRISCV uses this type of TLB) or a hash table, allowing for fast associative lookup based on virtual page numbers.

The TLB works besides the memory management unit (MMU) and the page table to translate virtual addresses into physical ones. When a program references a memory location, the TLB first check a match in its cache. If there is, the TLB retrieves the corresponding physical address directly, avoiding the

[4] `https://mister-devel.github.io/MkDocs_MiSTer`

need to access the page table. In the event of a TLB miss, where the requested virtual address is not found in the TLB, the MMU performs a page table walk to retrieve the information needed to: fill the TLB (TLB-Refill event in PandoOS plus [14]) and complete the translation.

1.2.5 Disassembler

A disassembler is a powerful tool that analyzes binary code and converts it into assembly language instructions. It plays a vital role in reverse engineering, software debugging, and understanding the inner workings of compiled programs. The disassembler's work is the symmetrical opposite of the assembler's one.

The disassemblers operate by reading the binary code of a program, analyzing the code, identifying the operation code (opcode), and translating it into their corresponding mnemonic representations. The disassembler's output provides details about the program's instructions, including the opcode, operands, registers, memory addresses, and control flow instructions.

One main application of a disassembler is reverse engineering, where it enables the examination of compiled software without having access to the source code. Reverse engineering can use the disassembled code to analyze the algorithm, identify vulnerabilities, understand program behavior, perform malware analysis, and extract valuable information such as data structures, function names/-calls, and control flow.

In addition, a disassembler plays a vital role in debugging and troubleshooting. When investing the reasons behind a program's crash, the disassembler allows the developers to trace the execution flow, and examine the registers and memory to identify potential bugs or errors.

Disassemblers can be found in various forms, ranging from standalone command-line tools to integrated development environment (IDE) plugins. Some disassemblers offer advanced features such as symbolic debugging (such as μRISCV), cross-referencing, and graph visualization to assist the code analysis.

1.2.6 Debugger

Debuggers are powerful tools, that allow the programmers to analyze the program execution. They assist developers in the laborious activity of bug finding and error fixing. Debuggers provide a wide range of features to aid in the debugging process, such:

- *Breakpoints*: debuggers allow programmers to set breakpoints at a specific line of code, which pause the normal execution flow when reached and allow a to examine the current status of the program, including the processor's state;
- *Variable inspection*: developers can inspect variables' value at specific points of the program, helping them to discover potential issues;
- *Call stack and Stack trace*: debuggers keep trace and display the call stack, which shows the sequence of function calls that led to the current point in the program. This feature helps the developer to understand the path, and which branches have been taken to reach a point of the execution;
- *Watchpoints*: watchpoints are similar to breakpoints, but they can be set only on variables and are triggered when such variables change their values. This helps track down unexpected modifications and avoid useless interruptions;
- *Stepping and flow control*: debuggers provide the functionalities to run programs step-by-step, to step over lines (bypassing checking), and to step out to return to the calling function;
- *Memory inspection*: debuggers allow developers to inspect the content of an arbitrary memory location, helping them to identify potential issues, such as buffer overflow or memory leaks.
- Debug symbols: debuggers work in conjunction with a compiler to incorporate debug symbols into the executable (see section 1.2.7).

Debuggers are essential tools for software development, not only do they help find issues, but they also accelerate the whole process. In the μRISCV project, a *debugging stub* (see section 2.6) has been implemented. A *debugging stub* is a component of the debugging system that resides on the target device. It works as a bridge between the debugger and the host machine. A debugging stub provides communication channels and services that enable the exchange of debugging information and commands between the target (e.g. μRISCV) and the debugger.

1.2.7 Symbol Table

The symbol table is a data structure used by compilers, linkers, and debuggers to store information about symbols (names) in a program. It serves as a database for mapping symbols to their attributes and provides efficient lookup.

The symbol table stores a wide range of information related to symbols, including variable names, function names, constants, labels, memory addresses, and their respective offsets. It also keeps track of data types, scope information, visibility, and other relevant attributes.

During the *compilation phase*, the symbol table is populated by the compiler as it scans the source code. Symbols encountered such functions and variables are added to the table. This allows, later on, the compiler to perform name resolution, type checking, and other semantic analysis tasks.

In the *linking phase*, the symbol table plays a relevant role, especially in resolving external symbols. Indeed, when multiple object files (programs, modules) are compiled together in a single executable or library, the linker can use symbols already resolved in one program on another one, reducing the linking time and avoiding issues related to redundancy.

Debuggers utilize the symbol table to facilitate program debugging. When debugging a program, the debugger can access the symbol table to display meaningful names and information about the variables and functions currently

involved in the debugging (this is the case of μRISCV). This helps programmers in understanding state, set breakpoints, and examine stack traces.

1.2.8 Toolchain

A toolchain is a collection of development tools that work together to facilitate the building, compilation, testing, and deployment of complex software. It includes tools like compilers, linkers, assemblers, debuggers, and code analyzers.

The streamline of the software development process made in the act by a toolchain usually begins with the compiler. A compiler is a tool that translates highlevel source code into machine code or an intermediate machine language. Compilers convert the code written in programming languages like C, C++, and Rust into executable or object files.

Once the source code is compiled, the *linker* comes to play. The role of the linker is to resolve symbols and references, combine object files and generate the final executable of library files. It ensures that all the dependencies are linked together and that the program can run.

Assemblers are a vital component of toolchains, especially when programming in low-level languages. They translate assembly language code into machine code, providing a bridge between human-readable instructions and the computer's processor.

Debuggers are powerful tools, that allow the programmers to analyze the program execution (see section 1.2.6).

Code analyzers and static analysis tools perform static code analysis. They examine the code for potential errors, security vulnerabilities (CVE) [25], code quality issues, and coding standards (e.g. naming convention, refactoring). These tools assist developers in writing cleaner, safe, and more maintainable code.

In addition to these tools, a toolchain may also include *profiles*, *performance analysis tools*, *documentation generators*, *build systems* (see section 1.2.9), and *version control systems*.

Toolchains can be customized based on the language, target platform, and development workflow. A particular type of toolchain called *cross-platform toolchains* allows the developer to write code on platform A to compile an executable targetted for platform B. It provides a unified development environment, allowing developers to write code once and deploy it across different platforms without significant modifications. To achieve the goal of cross-platform compatibility, toolchains often rely on inherently portable programming languages, such as C, C++, or Python. Cross-platform toolchains allow developers to develop the program without directly owning the target machine, reducing the cost for the programming environment in its complex. In general, they provide various advantages, including code reusability, faster development cycles, and reduced maintenance efforts.

In this dissertation, to compile the PandoOS plus project (see chapter 3), the RISC-V GNU cross-platform toolchain [5] has been used, since usually, the developer host's architecture is x86, while the project has to be compiled for the RISC-V architecture.

1.2.9 Build System

A *build system* is a software tool that automates the process of compiling source code, linking dependencies, and generating executables or object files. It provides a structured and repeatable approach to building software applications, simplifying the complex sequence of steps to perform to transform source code into a runnable application.

[5] The GNU cross-platform toolchain can be found on GitHub: `https://github.com/riscv-collab/riscv-gnu-toolchain`

The main goal of a build system is to provide efficiency, consistency, and reliability in the process of building an executable independently from the platform or development environment.

One of the striking features of a build system is the ability to track dependencies between source files. Moreover, it can manage the subsequent build, when a build is triggered, the system only recompiles the changed source, avoiding unnecessary recompilation and speeding up the whole process.

In addition, a build system is also capable of managing external dependencies, automatically downloading required libraries, and integrating them into the project. This feature assists developers to find and keep dependencies upgraded, especially among different platforms.

Another relevant feature of build systems is their integration with version control systems. Indeed they often support continuous integration (CI) and continuous delivery (CD) workflows. This allows to automatize the whole process of compiling, testing, and generating deployable artifacts.

Many are popular build systems, such as Make, Maven, Gradle, and Autotool, each with its advantages and features for specific programming languages.

Make is one of the most popular build systems. It is generally used to build software, but it has applications whenever arbitrary commands need to be executed. When executed, Make searches the current directory for a $makefile$, which is a well-formatted list of rules. Although Make is an incredible tool, it has problems when comes to executing a makefile on different platforms. For instance, a compiler on x86 might not accept the same options on RISC-V. To resolve this issue, tools like Cmake or Autoconf, have been developed.

Cmake is not a build system itself, indeed it only generates configuration files for other build systems like Make. It allows developers to specify project structure, dependencies, compiler options, and other build-related settings in a $CMakeLists.txt$ file. One of the key advantages of CMake is that developers can define different configurations based on, for instance, platform or compiler availability. In addition, it provides a modular and hierarchical approach to

organizing projects, allowing the creation of libraries, executables, and sub-projects, keeping a clear and structured representation of complex software.

1.3 Objective

The learning-by-doing principle is a widely adopted teaching method, where the students learn hands-on on a practical project: their source of knowledge is the direct experience [26]. μRISCV and all the versions of μMPS agreed with such a theory. This project aims to teach students how computer architecture and operating systems work, avoiding relatively useless and convoluted implementation details, that spread in those fields. Inner components had to be simplified, to achieve the goal of practical learning and maintain the project's realism. For instance, even though students do not directly manage external devices, they understand their functioning and how to interact with them.

The previous MIPS-based editions of μMPS (μMPS2, μMPS3) were optimal for teaching, but not ahead of the time, since nowadays its real-world applications are rare. On the other hand, RISC-V (see section 1.2.1), an emerging instruction set architecture, is gradually revolutionizing the entire market, thanks to its versatility ranging from embedded devices to high-performance computers. μRISCV provides an entry point for students to discover this practical instruction set.

The knowledge acquired through the utilization of μRISCV should not be limited to its specific environment, but rather it should be transferable and applicable in diverse contexts. For instance, the debugger feature section 1.2.6, which uses GDB [10], is a practical skill that students can reuse in future projects, thanks to the popularity of GDB.

To summarize, the objectives of μRISCV are: teaching students the fundamental concepts of computer architectures and operating systems in a straightforward yet realistic manner; introducing students to the RISC-V instruction set, which holds great potential for future advancements and innovations in the field of computer architecture; enabling students to acquire skills that extend

beyond the specific μRISCV environment, ensuring that the knowledge and competencies gained are applicable and transferrable to other contexts.

2 Implementation

This chapter covers the implementation of μRISCV, including all the additional and modified features. By the end of this chapter, the reader should have a clear understanding of how the processor, exception handling, and other components of the emulator function. The assumption is that the reader already possesses an overview of μMPS3, enabling them to comprehend extensively the enhancements made by this porting.

2.1 Processor

The processor is the heart of the computer and so of the emulation. It is responsible for operations like fetching, decoding, and executing the program's instructions, commonly called Fetch-Decode-Execute (FDE) cycle. The μRISCV's processor emulates the *RV32I* [33] version supporting the Integer Multiply/Divide (I) extension. Every machine cycle, the processor checks whether the *CPU TIMER* (see section 2.2) interrupt has to be signaled, then the previously fetched instruction is executed. If an exception occurs during the previous step, the exception handler is called while the various registers are correctly assigned to handle it. After incrementing the PC (Program Counter), the processor checks for the interrupts. If an interrupt occurs during this process, similarly to exceptions, the exception handler will tackle it. In the end, the processor fetches the next instruction, and whether necessary, the MMU translates its address from a virtual to a physical address. The executing behavior of the processor's steps heavily depends on the ISA. The various op-

codes defined by RISC-V, based on their composition, can be divided into six groups:

- *R type*: branching instructions (BEQ, BNE, BLT, BGE, BLTU, BGEU);
- *I type*: basic unsigned integer arithmetic and bitwise instructions (ADDI, SLLI, XORI, ANDI, etc.);
- *I2 type*: operations over CSR (see section 2.1.3) and break, ecall, return instructions (non-canonical);
- *S type*: store instructions;
- *U type*: sign extended unconditional jump instructions;
- *J type*: unconditional jump instructions;

The phases of Decoding and Execute of the instruction are combined to simplify the logic of the processor, while the Fetch of the new instruction is accomplished immediately after those two steps.

2.1.1 New Processor's State

A processor state is the collection of information and data that represents the current condition and context of the processor's execution. The processor's state constantly changes as instructions are fetched, executed, and results are stored in registers. During a context switch, the processor's state is saved to memory by a BIOS routine (see section 2.3), and another state previously saved in memory is executed.

Since the status register no more holds information about the interrupts enabled, a new register, called *mie* (see fig. 2.6), must to be saved in the processor's state.

Listing 2.1: Processor's state

```
typedef struct state {
  unsigned int entry_hi;
```

```
  unsigned int cause;
  unsigned int status;
  unsigned int pc_epc;
  unsigned int mie;
  unsigned int gpr[STATE_GPR_LEN];
} state_t;
```

Every field of the processor's state stores the corresponding register value (see section 2.1.3). For instance, *entry_hi* holds $CSR\_ENTRYHI$ value.

The *hi* and *lo* fields, previously dedicated to holding results from multiplication and division operations (see 2.1 of [13]), have been completely removed since unused in μRISCV.

2.1.2 Removed Coprocessor

The μMPS3 project included a coprocessor, which extends the processor functionality. Such a coprocessor had a series of special purpose registers and provided support for two operation modes, for exception handling, a processor Local Timer, and a Memory Management Unit (MMU) [13]. Unlike MIPS, which provides support for an ISA-level coprocessor, RISC-V does not. It would be possible to design a coprocessor specific to the project architecture, but since the aim of $\mu RISCV$ is educational, the idea has been discarded. Some of the registers of the MIPS processor supported the TLB handling, so those are the only ones that have to be ported in the new project. RISC-V architecture provides additional registers for custom purposes that have been used to cover the missing coprocessor (see section 2.1.3).

2.1.3 Control and Status Register (CSR)

The Control and Status Registers (CSR) are registers that store CPU data. RISC-V defines 4096 CSRs. A few of them are allocated by RISC-V [33], whilst the others are designed for custom purposes.

The relevant CSRs used in μRISCV are:

- *time*: real-time counter;
- *mtvec*: trap handler address (see section 2.2);
- *mepc*: address to return after an exception is handled;
- *mcause*: code indicating the event that caused the trap (see fig. 2.1);
- *mstatus*: keeps track of current processor state (see fig. 2.2);
- *mie*: interrupt enabled bits (see fig. 2.6);
- *mip*: interrupt pending bits (see fig. 2.7);

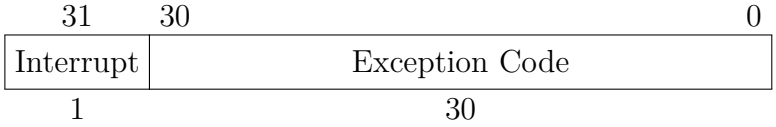

Figure 2.1: Machine cause register layout [33] (only relevant bits are named)

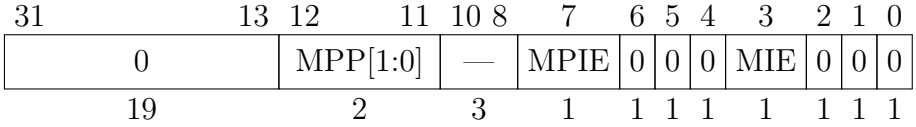

Figure 2.2: Machine status register layout [33] (only relevant bits are named)

Since there are no registers dedicated to the TLB, some from the custom CSRs area have been selected (see fig. 2.3). The names and semantics of these registers are inherited from the TLB handling of μMPS3 to keep compatibility with previous projects.

Macro	Value	Description
ENTRYLO	0x800	Higher part of a TLB entry
ENTRYHI	0x801	Lower part of a TLB entry
INDEX	0x802	Index of the TLB table
RANDOM	0x803	Random index of the TLB table
BADVADDR	0x804	Virtual address that generated an error

Figure 2.3: Custom CSRs

2.2 Exception Handling

Exceptions are events that change the normal execution flow of the instruction currently processed by the machine. There are two main types of exceptions: *interrupts* and *exceptions*. From the point of view of the hardware, the latter can be again divided into two categories: *User TLB exceptions* and *all the other ones*. The handling proposed in the RISC-V manual is quite different compared to the MIPS one. The RISC-V specs provide a list of interrupts and exceptions that a machine should handle (see fig. 2.4).

Interrupt	Exception code	Description
1	3	Machine software interrupt (Interval Timer)
1	7	Machine timer interrupt (CPU Timer)
1	≥16	Designed for platform use (Devices)
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store address misaligned
0	7	Store access fault
0	8	Environment call from U-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store page fault
0	24-31,48-63	Designed for custom use

Figure 2.4: Machine cause register (*mcause* see fig. 2.1) values after trap (see Table 3.6 in [33])

There are no predefined codes for handling TLB exceptions. Consequently, a portion of the *Designed for custom use* area has been used (see fig. 2.5) for this purpose.

Interrupt	Exception code	Description
0	24	TLB mod
0	25	TLB load fault
0	26	TLB store fault
0	27	User TLB load fault
0	28	User TLB store fault

Figure 2.5: Custom exceptions to handle TLB events

mtvec is a special register that defines how the machine should retrieve the correct subroutine to handle a specific event. This register has two fields: *BASE* (30-bit) and *MODE* (first 2-bit). Where *BASE* indicates the base address of the subroutine, *MODE* specifies how the processor must interpret the *BASE* field. There are two possible values for *mode*: *0* (direct mode) or *1* (vectorized mode). This project aims to let students develop their small kernel in a high-level language (e.g. C). Using the vectorized mode would mean introducing another layer represented by the assembly code to handle the various exceptions. So to maintain the logic of the processor simple, μRISCV implements the direct mode. Since the *mtvec* in the direct mode stores only one address, the procedure to determine whether the TLB-Refill or general exception handlers should be called, is now incorporated in the BIOS (see section 2.3).

2.2.1 Processor Behaviour

An interrupt breaks the normal flow of the machine, so when raised the processor state (i.e. registers) must be saved to continue the stopped execution after such an interrupt has been handled. The routine to handle exceptions/interrupts has not changed apart from the processor state field that has been modified (see section 2.1.1). Indeed since the *cause* register, replaced with

the CSR register *mcause* (see fig. 2.1), no more holds the information about which interrupts are enabled, a new CSR register, *mie* (see fig. 2.6), has been introduced to achieve that. In addition, a new CSR register, called *mip* (see fig. 2.7), holds the pending bits of all interrupts. Interrupt cause number i (see fig. 2.1, fig. 2.8) corresponds to the i-th bit in both *mip* and *mie*.

15 — 12	11	10	9	8	7	6	5	4	3	2	1	0
—	MEIE	0	0	0	MTIE	0	0	0	MSIE	0	0	0
4	1	1	1	1	1	1	1	1	1	1	1	1

Figure 2.6: Interrupt enable bits (mie 0:15) layout [33]

15 — 12	11	10	9	8	7	6	5	4	3	2	1	0
—	MEIP	0	0	0	MTIP	0	0	0	MSIP	0	0	0
4	1	1	1	1	1	1	1	1	1	1	1	1

Figure 2.7: Interrupt pending bits (mip 0:15) layout [33]

The full procedure is similar to the one described in [13] but slightly modified:

1. Load the Exception PC (EPC) CSR register with the current PC value;
2. Set the exception cause code in *mstatus* (see fig. 2.1, fig. 2.2, fig. 2.4)
3. Copy the value of the *mie* bit in *mstatus* CSR register into the *mpie* bit of *mstatus*, then reset the *mie* bit;
4. Copy the current machine mode into the *mpp* field of *mstatus*, then set the current machine mode to *Machine* (0x3) [33];
5. Load the PC with the value of the *mtvec* CSR register (see section 2.2).

Then whenever a *mret* instruction [33] is executed, the complementary operations are performed: the *mie* bit of *mstatus* is set to the value of *mpie* bit, and the current mode is set to the value of the *mpp* field of *mstatus*.

Interrupt Line	Description
16	Inter Processor Interrupt
17	Disk Device
18	Flash Device
19	Ethernet Device
20	Printer Device
21	Terminal Device

Figure 2.8: Custom interrupt lines

2.3 BIOS

The BIOS ROM code provides low-level services such as:

- *Bootstrap* function: then BIOS routine is the first code executed when μRISCV is turned on;
- *TLB-Refill* handler: routine invoked whenever a TLB-Refill event occurs;
- *Exceptions* handler: routine invoked whenever an exception occurs;
- *Library* services: wrapper function to access/modify CPU registers and to manage the TLB table;

2.3.1 Bootstrap

The bootstrap program initializes hardware facilities and starts execution. The routine code can be found in *coreboot.S*. The start-up of the emulator performs the following legacy operations:

- Setting the BIOS data page address;
- Setting the TLB-Refills and exception handlers to point to the Kernel Panic routine;
- Setting the next PC to the start of the kernel (fixed at $0x20001004$);

In addition, the address of the exception handler is loaded in the CSR *mtvec* (see section 2.2), as stated in the RISC-V manual.

2.3.2 Library Services

μRISCV is bundled with a library: *liburiscv*. That library provides a series of wrapper functions, that works as an abstraction for the user to read/write CPU registers or manipulate the TLB. The *liburiscv* functions can be divided into three categories:

- Wrappers to RISC-V assembly instructions, since it is impossible to use them in C (see fig. 2.10);
- Wrappers to CSR registers, since it is impossible to access them in C (see fig. 2.9);
- New instructions to extend the RISC-V architecture and useful for kernel authors;

Wrapper(s)	CSR Register
getINDEX(), setINDEX()	CSR_INDEX
getRANDOM()	CSR_RANDOM
getENTRYLO(), setENTRYLO()	CSR_ENTRYLO
getENTRYHI(), setENTRYHI()	CSR_ENTRYHI
getBADVADDR()	CSR_BADVADDR
getTIMER(), setTIMER()	TIMER
getSTATUS(), setSTATUS()	MSTATUS
getCAUSE(), setCAUSE()	MCAUSE
getEPC()	MEPC
getMIE(), setMIE()	MIE
getMIP()	MIP

Figure 2.9: Wrapper functions of CSR register

liburiscv provides C-language access to two RISC-V assembly instructions:

Wrapper(s)	Description	RISC-V instruction
WAIT()	Idle Processor	wfi
SYSCALL(a0,a1,a2,a3)	Syscall	ecall

Figure 2.10: Wrapper functions of RISC-V assembly instructions

In addition to providing "wrapper" functions to access various μRISCV registers and assembly instructions, *liburiscv* extends the RISC-V instruction set with the following services/instructions:

Wrapper(s)	Description	Ebreak code
LDCXT()	Load Context	0
LDST()	Load Processor State	1
PANIC()	Kernel Panic	2
HALT()	Halt Processor	3
TLBWR()	TLB-Write-Random	4
TLBWI()	TLB-Write-Index	5
TLBR()	TLB-Read	6
TLBP()	TLB-Probe	7
TLBCLR()	TLB Clear	8
STST()	Store Processor State	/

Figure 2.11: Wrapper functions of custom services

Excluded $STST$, all the instructions in fig. 2.11 are implemented using the *ebreak* assembly instruction. *ebreak* is an instruction used to return control to a debugging environment, but in that case, as μMPS3 did, μRISCV uses it to generate a trap and handle the custom instruction based on its code (e.g. 4 for $TLBWR$). The idea is to first set the $a0$ register with the command code value (see fig. 2.11) and then call *ebreak*. The emulator will check that register to decide which function has been requested. Such a trick is necessary because unlike MIPS (see [16]), RISC-V does not provide, at the ISA level,

instructions to manipulate the TLB. That is the reason why *liburiscv* has to expose specific functions.

Further details can be found in [13].

2.4 Devices

μRISCV supports five different classes of external devices: *disk*, *tape*, *network card*, *printer*, and *terminal*. Furthermore, μRISCV can support up to eight instances of each device type. Each single device is operated by a controller. Controllers exchange information with the processor via device registers; special memory locations. Further details can be found in [19].

2.4.1 Terminal

Since the graphical interface is optional in μRISCV, the terminals can no more send their output to the GUI as in μMPS3. The console where the emulator has been executed will log their output with this format:

```
[!] TERM (i) `Output from terminal i`
```

where "i" is the terminal number that printed the string "Output from terminal i". The controller of each terminal stores the characters received from the kernel/user processes in a buffer. When the controller reads a new line character (i.e. "\n"), the emulator prints the buffer into the console and clears it.

Note:
The terminals at the moment can not read the user input (see section 4.1.2).

2.5 A simple disassembler

A *disassembler* is a tool capable of translating the machine language (e.g. 000101010) to assembly language. The disassembler has no utility in the emulation itself. Nonetheless, when comes to debugging, the disassembler plays a vital role. Since disassembling is not the purpose of this thesis, just the basic operations are implemented. μRISCV disassembles the instructions when the processor executes them, so the disassembling process is linear with the program flow. In that way, it is possible to output, along with the assembly, the current value of the registers involved in the opcodes. For instance, the *ADDI* opcode will be disassembled in:

```
[200018c0] (fe010113)  I-type | ADDI sp,sp(20000fe0),-32
|   PC   | | OPCODE | | TYPE ||  Disassembled opcode   |
```

Exploiting the *uriscv-elf2uriscv* (see section 2.7.1) utility to generate a symbol table (see section 1.2.7), the emulator, based on the current PC, prints the function name where such instruction resides. The latter print will be performed whether the program flow joins a new function/exits one.

2.5.1 Usage

Disassembling is an optional functionality since students would probably use this feature in specific scenarios (e.g. unexpected behavior and GDB not useful). The user to enable it has to pass a specific flag while running the emulator (i.e. *—disass*).

2.6 GNU Debugger (GDB)

GNU Debugger (GDB) [10] is one of the program parts of the GNU toolchain (see section 1.2.8). GDB has plenty of features, but just a subset of them are implemented in this thesis to keep the project simple but powerful enough to permit the debugging of the code. The inclusion of GDB support in the

implementation aims to provide students with the ability to debug without the need to learn assembly language. By utilizing GDB, students can better understand the actual processes occurring and obtain precise insights into the execution of their code. Although μMPS3 includes a debugger embedded in the emulator, certain characteristics limit its effectiveness during actual debugging phases:

- Users can insert breakpoints only at the beginning of a function;
- When a breakpoint is reached, the state of the variables is unknown;
- The execution flow is not fully understandable since the only code visible is the assembly one;

The GDB support resolves each of these issues. To implement the GDB support, the μRISCV emulator has been provided by a *debugging stub* (see section 1.2.6), which works as a bridge between μRISCV and *GDB*.

2.6.1 Design

The GDB feature works as a wrapper for the emulator and manages its steps according to the commands received by the GDB client.

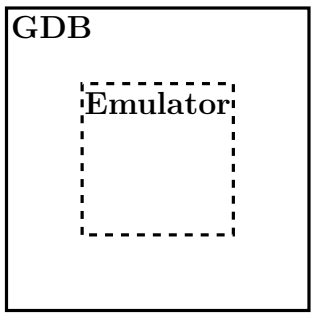

Figure 2.12

This feature implements a simple *TCP Server* that receives commands sent by the *GDB client* on port 8080. The steps to receive, decode and reply to messages are the following:

1. The server starts;
2. The server listens and then accepts one connection;
3. The server starts a thread aimed to run the emulator;
4. The server waits for command;
5. In case of a message from the client:
    a) The message is decoded;
    b) The corresponding action is executed;
    c) The reply is crafted and sent back;
6. Jump to step 4;

To simultaneously listen for commands while the machine is actively running, μRISCV executes the emulator and the TCP server in different threads. In that way, while the emulator is running, threads can listen for a *pause* command and permit the user to add a breakpoint in the mid of the execution (see section 2.6.2). For instance, if the program becomes stuck in an infinite loop, the user can utilize GDB to pause the execution and analyze the path that led to that particular point. This feature allows for a better understanding of the program's behavior and aids in identifying the cause of the infinite loop. To communicate between threads is necessary a messaging system. Two different approaches achieve this: *sockets* and *global variables*. The socket-based approach offers the advantage of having pre-implemented communication, but it has a significant overhead due to the protocol involved. On the other hand, global variables provide a simple solution without any additional overhead. However, using global variables requires proper synchronization between threads and addressing potential issues related to critical sections. Considering that the program behind the GDB server is not complex, the option of using global variables appears to be more suitable for this situation.

2.6.2 Features implemented

GDB offers plenty of features to debug every part of the source code and tempestively find the issues. However, this project has implemented just the essential subset of them. This limitation is justified by the educational purpose of the project, aiming to keep every function simple and accessible to as many students as possible. The commands handled by this implementation can be divided into two groups: *initialization* and *action*. The first group consists of the *supported actions* question and the *question mark* command. The *supported actions* question is used to inquire about the functionalities implemented by the server. In this implementation, the only supported action is *swbreak* (i.e. software breakpoints). The *question mark* command is used to determine the reason why the target halted and is utilized by both the *step* and *continue* actions. The reply to this message is *S05*, representing the signal *SIGTRAP*. In the second group, there are the following commands:

- *g*: returns the list of CPU registers and the PC register;
- *G [registers]*: writes [registers] to the current CPU's registers;
- *c*: continues the execution until a breakpoint is reached;
- *z, Z [addr]*: respectively adds or removes a breakpoint at [addr];
- *m [addr] [length]*: reads [length] addressable memory units at the address [addr];
- *P [reg] [value]*: writes [value] to [reg]-th CPU register;
- *k*: pauses the execution at the current state (e.g. pressing Ctrl-c), that later can be resumed by the *c* command;

Every other command will receive an *empty reply* (i.e. 00).

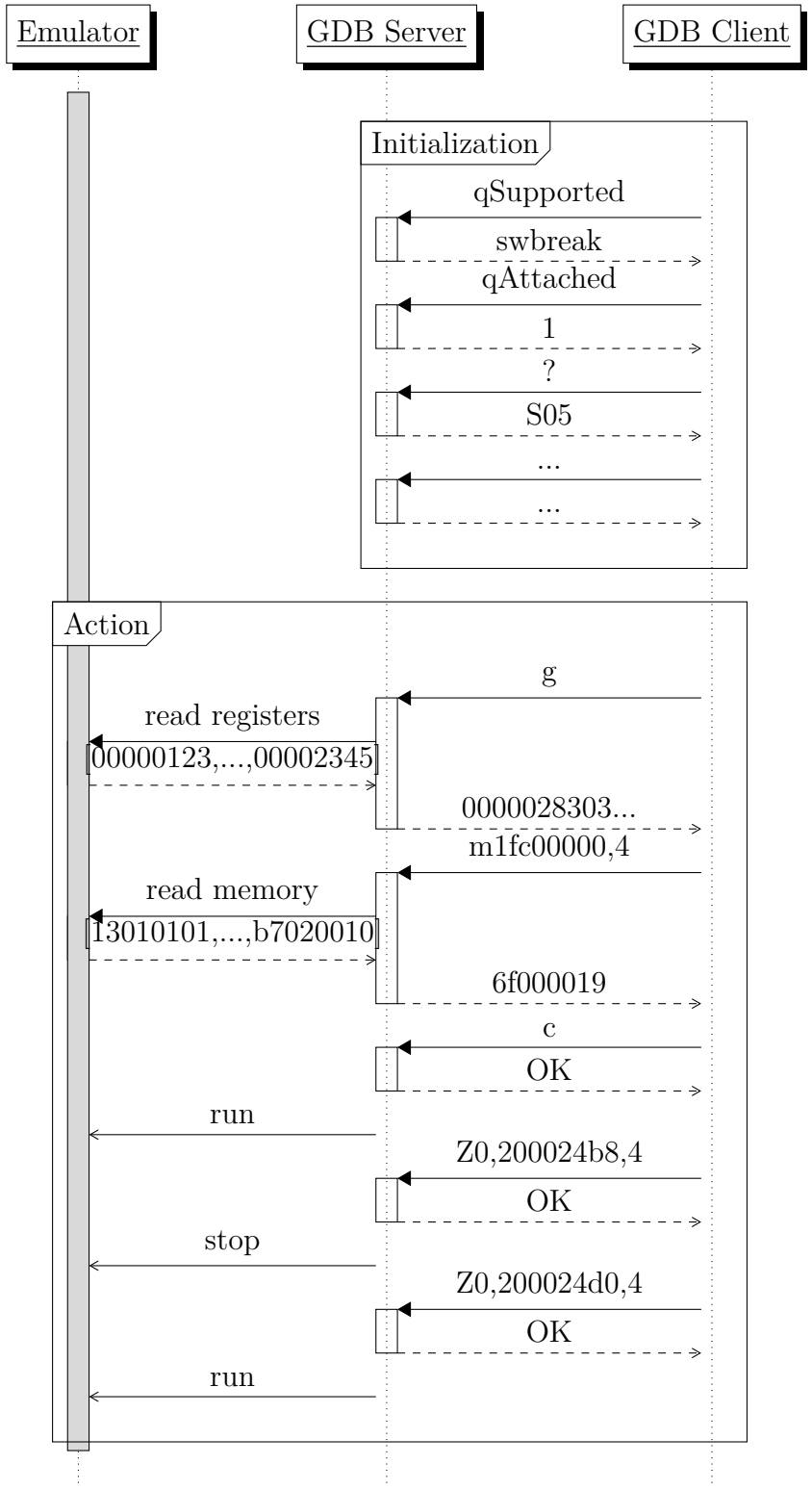

Figure 2.13: Communication between the *emulator*, the *gdb server* and the *gdb client*
Note: the diagram is just an example

2.6.3 Usage

Debugging is an optional functionality and the user to enable it has to pass a specific flag while running the emulator (i.e. *—gdb*). The program starts and creates the TCP Server, while the user just has to open a client to connect. Users can use the command line interface of GDB or the extension for the IDE they prefer the most. For instance, *Visual Studio Code* [24] has an extension called *Native Debug* developed by *Webfreak* [34]. The following configuration is an example of a *launch.json* file for *Native Debug*:

Listing 2.2: VS Code config

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "URISC-V GDB",
            "type": "gdb",
            "request": "attach",
            "executable": "/path/to/the/kernel",
            "target": "localhost:8080",
            "remote": true,
            "cwd": "${workspaceRoot}",
            "valuesFormatting": "parseText",
            "gdbpath":
                ↪ "/path/to/riscv/bin/riscv32-unknown-linux-gnu-gdb",
            "stopAtEntry": true
        }
    ]
}
```

The user have to compile the projects with the *—ggdb* flag [32] to produce debugging information for use by GDB.

2.7 Utilities

The *uriscv-elf2uriscv* and *uriscv-mkdev* utilities are inherited from μMPS3 and modified slightly to ensure compliance with the new architecture. As the current project is independent of the Graphical Interface (GUI), a new utility, called *uriscv-mkconfig*, is necessary to create a configuration file.

2.7.1 uriscv-elf2uriscv

The utility *uriscv-elf2uriscv* converts an ELF object into RISC-V file formats: .core, .rom, or .aout. The utility *uriscv-elf2uriscv* generates a symbol table (see section 1.2.7) and implements a function that, given an address, identifies the symbol containing that address. Although the structure of the ELF remains largely consistent across architectures, a minor modification has been made to the metadata-checking process to ensure compatibility with the current architecture, which should now be identified as `EM_RISCV`.

To convert the ELF formatted executable and object files produced by the gcc cross-platform development tools into the *.core*, *.rom*, and *.aout* formatted files required by μRISCV the command is:

```
uriscv-elf2uriscv [-v] [-m] {-k | -b | -a} <file>
```

Further details can be found in [13].

2.7.2 uriscv-mkdev

The *uriscv-mkdev* utility allows one to create an empty disk only; this way an OS developer may elect any desired disk data organization. The created "disk" file represents the entire disk contents, even when empty. Hence this file may be very large. It is recommended to create small disks which can be used to represent a little portion of an otherwise very large disk unit. Disks are created via:

```
uriscv-mkdev -d <diskfile>.uriscv
```

```
  [cyl [head [sect [rpm [seekt [datas]]]]]]
```

Flash devices in μRISCV are "random access" nonvolatile read/write devices. A μRISCV flash device is essentially equivalent to a seek-free one-dimensional disk drive. The uriscv-mkdev utility allows one to create both slow flash devices (e.g. USB stick) or fast flash devices (e.g. SSDs). Furthermore, the utility allows one to create both empty flash devices as well as ones preloaded with a specific file. The created flash device file represents the entire device contents, even when empty. Hence this file may be very large. It is recommended to create small flash devices which can be used to represent a little portion of an otherwise very large device. Flash devices are created via:

```
uriscv-mkdev -f <flashfile>.uriscv <file> [blocks [wt]]
```

Further details can be found in [13].

2.7.3 URISCV-MKCONFIG

The utility *uriscv-mkconfig* allows the user to create the configuration file (i.e. $config\_machine.json$) that the emulator will use to run the machine. The config is created via:

```
uriscv-mkconfig <file> [--proc <nproc>] [--clock-rate <mhz>]
  [--tlb-size { 4 | 8 | 16 | 32 | 64 }]
  [--tlb-floor  { RAMTOP | 0x4000000 | 0x8000000 | OFF }]
  [--ram-size <size>]
  [--boot-bios <coreboot>.rom.uriscv]
  [--exec-bios <exec>.rom.uriscv]
  [--core <kernel>.core.uriscv]
  [--load-core { true | false }]
  [--stab <kernel>.stab.uriscv]
  [--stab-asid <asid>]
```

where:

- *file*: the path where the config file will be generated;

- —*proc*: the processors number (default 1);
- —*clock-rate*: the clock rate in MHz (default 1);
- —*tlb-size*: the size of the tlb (default 16);
- —*tlb-floor*: the threshold below which address translation is disabled and the logical address is the physical address (default OFF);
- —*ram-size*: the size of the RAM (frames) (default 64);
- —*boot-bios*: the path of the ROM with the bootstrap instructions (default */usr/share/uriscv/coreboot.rom.uriscv*);
- —*exec-bios*: the path of the ROM of the bios (default */usr/share/uriscv/exec.rom.uriscv*);
- —*core*: the path of the kernel (default *kernel.core.uriscv*);
- —*load-core*: load the core file (default *true*);
- —*stab*: the path of the symbol table file (default *kernel.stab.uriscv*);
- —*stab-asid*: the ASID of the symbol table (default $0x40$);

Once the utility is launched the user will be asked whether there are devices. The user can choose from the five available devices in μRISCV: *Disks*, *Flash Devices*, *Network*, *Printers*, and *Terminals*. For each added device, the user will be prompted to insert: the *device file* and whether the device is *enabled*.

2.8 Command-Line Interface (CLI)

A command-line interface (CLI) is a text-based interface that enables users or clients to interact with a device or program using commands. In contrast to the graphical user interface (GUI), the CLI consumes fewer resources and does not rely on a window manager. The CLI is often preferred in scenarios where resource efficiency is crucial or when automation and scripting capabilities are

required. Unlikely μMPS3, μRISCV incorporates a command-line interface (CLI) for the following reasons:

- Automate the testing of the kernel;
- Possibility to execute the kernel on a remote server using, for instance, by utilizing protocols such as SSH;

2.8.1 Usage

To launch the emulator is necessary a console; the command to run is:

```
uriscv-cli [config_machine.json]
  { [--help] | { [--disass] [--gdb] } }
```

Where:

- *config_machine.json*: the path of the config file (default $config\_machine.json$ in the current directory);
- —*help*: prints the help message;
- —*disass*: enables the disassembler messages;
- —*gdb*: enables the gdb server;

3 PandOS plus

This chapter covers the PandoOs plus project, what it is, and how it is composed. Then, an example of a compilation of the project for μRISCV is proposed, and in conclusion an explanation of how to execute the now compiled project in the μRISCV emulator.

3.1 The project

PandOS Plus serves as the laboratory project for the Computer Science course on Operating Systems at the University of Bologna for the academic year 2021/2022. This project is a revised edition of *PandOS* [14], which is one of the projects within the collection developed under the umbrella of *Virtual Square* [12, 15]. PandOS plus is based on the μMPS3 architecture, therefore the only way to run a PandOS plus is to emulate it through μMPS3 [13]. The project is designed to be completed by groups consisting of 3 to 4 individuals. It is divided into three phases, each introduces various concepts to the students. The students at the end of each phase can test their code on a series of tests thought ad hoc to verify their competencies. The phases represent layers of an operating system, in which a layer i is based on the implementation of layer $i-1$, based on *"THE" Multiprogramming system* [6]. The architecture of PandOS plus is composed of 6 abstraction layers (see fig. 3.1).

Phase	Layer	Description
/	6	Interactive shell
/	5	File system
3	4	Support layer
2	3	Kernel of the O.S.
1	2	Queue management
Given	1	Services of the ROM
Given	0	Hardware of μMPS3

Figure 3.1: PandOS plus layers

The successful implementation of the developed emulator can be validated by porting the PandOS Plus project from μMPS3 to μRISCV. Since PandOS Plus covers essential features such as user processes, virtual memory, external devices, and paging, verifying its correct functioning on the μRISCV emulator will serve as a reliable test. By ensuring that PandOS Plus operates seamlessly on the μRISCV architecture, it confirms the emulator's successful implementation and compatibility with a real-world operating system project.

3.1.1 Phase 1

In phase 1 of PandOS plus the students have to implement the various *data structures* of the kernel and the *queue management*. Specifically, the queue management component involves implementing 4 functionalities related to Process Control Block (PBC):

- Allocation and Deallocation of PCBs;
- PCB queues management;
- PCB tree management;
- Active Semaphore List (ASL) management, which manages the queue of blocked processes on a semaphore;

During this phase, the requirements of PandOS Plus are exclusively focused on logic and implementation of the kernel's data structures and queue management. This implies that there is no involvement with the hardware, and thus the original project can remain unchanged during this stage.

Further details can be found in [14].

3.1.2 Phase 2

With all the structures implemented by the previous phase, it is the moment to implement the operating system *kernel*. The functionalities that the kernel has to manage are:

- *Initialization of the system*: initialize the data structures (i.e. PCBs and semaphore lists), the devices, the interval timer, the pass-up vector, and the scheduler;
- *Scheduling of the processes*: scheduling of the processes based on two priority queues (low and high) and the round robin algorithm;
- *Exception handling*: handling of the exceptions that could occur during a normal program flow. Such exceptions can be broken down into two categories:
  - *TLB-Refill events*, triggered every time, during an address translation, there is no matching between the entries in the TLB;
  - *All other exceptions*, that can be grouped like:
    - *Interrupts*: external devices and timers;
    - *System Service Calls*;
    - *TLB exceptions*;
    - *Program Trap exceptions*;

In that phase, the kernel should be able to execute sequential processes and grant synchronization primitives. Since the virtual memory will be implemented in the following phase (see section 3.1.3), all the addresses have to be physical.

During that stage, the kernel needs to possess the capability to execute processes sequentially and provide synchronization primitives. All addresses utilized during this stage are physical since virtual memory implementation is planned for phase 3 (see section 3.1.3).

No effective modifications have been made as the initialization and scheduling functionalities are independent of the hardware or architecture in general. On the other hand, the exception handling has been tweaked to adequate with the new *processor state* (see section 2.1.1), in particular: *cause register* (mcause, see fig. 2.1), *interrupt pending bits* (mip, see fig. 2.7), and *interrupt enable bits* (mie, see fig. 2.6).

Further details can be found in [14].

3.1.3 Phase 3

After the development of the scheduler and ensuring the readiness of processes to run, the subsequent task is to implement virtual memory. This implementation enables the creation of user processes and provides a supportive layer for them. During this phase, the students are given 8 pre-compiled processes, each contained within a flash device [13], and having distinct behaviors. The main function has to:

- initialize the various data structures like in phase 2 (see section 3.1.2);
- load and start the 8 processes, each with its local address space (kuseg) and a unique identifier, ASID [13]. The support layer provides the exception handlers that manage exceptions unhandled by the kernel, such as page faults (pager) or “custom” syscalls;
- wait for all the processes to terminate;

The project did not require any revisitation since the memory layout and paging remained unchanged. However, similar to phase 2, there were slight modifications in the activation and deactivation of interrupts due to differences in the register configuration (see fig. 2.2).

Further details can be found in [14].

3.2 Compiling

μRISCV is based on the RISC-V ISA architecture, so to build an executable for this emulator a RISC-V compiler is mandatory. This section describes which compiler could be used to build PandOS Plus, including a step-by-step guide to compiling both kernel files and U-procs.

3.2.1 Toolchain

A toolchain, as described in section 1.2.8, is a set of programming tools that can be used to build and test a complex software program. Given that students typically do not have access to a physical RISC-V machine to execute a native compiler, the utilization of a cross-compiler becomes necessary. The toolchain developed by GNU [9] can be used to accomplish that, it has all the necessary tools, such as *compiler*, *linker*, *objdump*, and *debugger*. The RISC-V GNU compiler toolchain can be found on GitHub [32], allowing students to build it themselves if desired. Otherwise, it is available at the following locations:

- Archlinux repository (AUR): *riscv–gnu–toolchain–bin*[1];
- Debian bullseye repository: *gcc–riscv*64*–unknown–elf*[2];

Note:
At present, GDB (GNU Debugger) is exclusively accessible in the Arch Linux

[1] `https://aur.archlinux.org/packages/riscv-gnu-toolchain-bin`
[2] `https://packages.debian.org/bullseye/gcc-riscv64-unknown-elf`

repository. Debian users, on the other hand, are required to build GDB from source in order to run it on their system.

3.2.2 Compiling the kernel

This is just an example using:

- *phasex.o*: compiled phases;
- *uriscvcore.ldscripts*: linker scripts for .aout and .core executables [19];
- *crtso.o*: start-up modules for .core executables [19];
- *liburiscv.o*: uriscv library [19];

To build the kernel the commands are:

```
riscv32-unknown-linux-gnu-ld -G 0 -nostdlib -c    \
  -march=rv32imfd -mabi=ilp32da -m elf32lriscv    \
  <...>.o -o phasex.o

riscv32-unknown-linux-gnu-ld -G 0 -nostdlib       \
  -march=rv32imfd -mabi=ilp32da -m elf32lriscv    \
  -T /usr/local/share/uriscv/uriscvcore.ldscript  \
  phase1.o phase2.o phase3.o                      \
  /usr/local/lib/uriscv/crtso.o                   \
  /usr/local/lib/uriscv/liburiscv.o               \
  -o kernel
```

The *kernel* file is properly a RISCV executable but has to be converted in a format that μRISCV is able to read. To do that the user should use the command:

```
uriscv-elf2uriscv -k kernel
```

Further details about *uriscv–elf2uriscv* can be found in the manual [13].

3.2.3 Compiling the User processes

In PandOs plus, the students are provided with 8 *.c* files (u-procs), each representing a different program that has to run on μRISCV.

To build a test program *test.c* the commands are:

```
riscv32-unknown-linux-gnu-gcc -G 0 -nostdlib -c   \
  -pedantic -ffreestanding -Werror -Wall -ansi    \
  -std=gnu99 -c -static -ggdb -nostartfiles       \
  -nostdlib -O0                                   \
  -march=rv32imfd -mabi=ilp32da -m elf32lriscv    \
  test.c -o test.o

riscv32-unknown-linux-gnu-ld -G 0 -nostdlib       \
  -march=rv32imfd -mabi=ilp32da -m elf32lriscv    \
  -T /usr/local/share/uriscv/uriscvcore.ldscript  \
  /usr/local/lib/uriscv/crti.o                    \
  /usr/local/lib/uriscv/liburiscv.o               \
  test.o                                          \
  -o test.t

uriscv-elf2uriscv -a test.t # generates a test.aout.uriscv file
```

The U-proc executable now has to be pre-loaded in a flash device

```
uriscv-mkdev -f test.uriscv test.aout.uriscv
```

3.3 Running PandOS on μRISCV

3.3.1 Making the config

To run the emulator, it is necessary to create a configuration file. This file should include all the essential information: kernel's path, bios' path, ram size, etc[3]. To generate such a file the user has 2 options:

- *uriscv-mkconfig*: command line interface program (see section 2.7.3);
- *μMPS3*: since the config file of μMPS3 is compatible with μRISCV the user can generate it using the μMPS3 emulator;

3.3.2 Running the Emulator

After compiling the kernel, preparing the user processes (u-procs), and creating the configuration file, the final step is to execute the emulator using the following command:

```
uriscv-cli
# or
uriscv-cli /path/to/the/config.json
```

Further details on *uriscv–cli* can be found in section 2.8.

[3]Exstensively described in the μMPS2 dissertation [19]

4 Conclusions

4.1 Improvements

In this section, various improvements are suggested for future works. The project is distributed under the GPL-3.0 license, allowing individuals the freedom to share and modify the software according to their requirements without any restrictions.

4.1.1 Memory Management Unit (MMU)

The *memory management unit* (MMU) is a computer component that manages access to the memory. In the μMPS3 project [13], the MMU's main activity is translating virtual memory addresses to their respective physical address using paging and TLB (see section 1.2.4). The current solution devised for μMPS3 is a simplified version of the MIPS' MMU, which due to its complexity, has been simplified. Such modifications are the fruit of years of change based on students' feedback. The virtual-memory system proposed by the RISC-V manual [33] is thought to be used on real-world computers, which includes features such as multi-level paging. In μRISCV, the MMU is completely inherited from μMPS3. In the future, energies could be spent to design a new MMU that follows the RISC-V directives, while keeping the complexity at the undergraduate level. For instance, the multi-level paging support could be removed and substituted with a classical one-level paging, similar to the one used in μMPS3.

4.1.2 Terminal Device

Within the μRISCV framework, one of the implemented devices is the Terminal, which serves as a means for kernel and user processes to output strings to the console. In the μMPS3 project, terminals not only enable the printing of information but also facilitate user input, thereby enabling the development of more interactive and expressive programs. As μRISCV does not feature a graphical user interface (GUI), the console itself, where the emulator is initiated, becomes the primary and practical method for user output. In general, an input event on the console is a blocking event, meaning that the whole emulator should be paused until the user has not input some data. Such a situation can not happen, since it would temporarily disable the parallelism achieved by μRISCV, harming its realistic characteristic, which is one of the objectives of the project (see section 1.3). A possible solution could be to use a *terminal multiplexer* [1] as *tmux* [21] to multiplex several pseudoterminals, one for the emulator, and other N-pseudoterminals, one per terminal device installed. Alternatively, a file could be targeted as the input source for terminal "i", and every time the user writes on it, the emulator captures it and sends it to the terminal "i" controller.

4.1.3 Graphical Interface and GDB

The objective of this thesis is to port only the emulator itself, without including the GUI. In future work, the GUI can incorporate the GDB features developed by μRISCV. It should be noted that μMPS3 already possessed debugging utilities [13]. When porting the GUI, a decision needs to be made whether to replace the existing utilities to uniquely provide support for GDB or to devise a solution that maintains both sets of features. This would enable students to choose the debugging support that best suits their needs and abilities.

[1] A terminal multiplexer is an application that can be used to have multiple pseudoterminals inside a single terminal display

4.1.4 Software Packaging

Software packaging refers to the process of bundling software and its associated components into a distributable format that can be easily installed and managed on various computer systems. Packaging involves organizing the necessary files, libraries, dependencies, and configuration settings to create a self-contained package that can be distributed to end-users or other systems. Packaging permits tracking installed files and automatically updates the software while providing users with a simple way to install their programs. After the release of μRISCV, as happened for μMPS3, the usual distribution process expects the software to be packaged for Linux distros [2]. The major Linux distributions for which μRISCV should be packaged are *Debian*, *ArchLinux*, and *Ubuntu*. Those three distros cover the majority of the operating systems usually installed by students. Whether a user has a niche Linux distro, they can easily clone [3] the μRISCV repository available on *GitHub* [23]. The ideal solution should be to package μRISCV ad hoc for the various distros, similar to what has been done with the μMPS3 project. Alternatively, other package manager software could be used such as Flatpak [7], Snapcraft [2], and AppImage [30], which enable software distribution compatible with every Linux distro.

4.2 Final thoughts

In conclusion, this thesis has explored the design and implementation of an educational computer architecture called μRISCV and its emulator. Moreover, giving an example of a project, PandOS Plus (see chapter 3), that students can achieve to develop and run on μRISCV.

Implementing such a computer emulator makes how feasible can be to provide a realistic and interactive environment for students to explore computer architecture concepts. The emulator allows students to experiment with different

[2] A Linux distro is a software distribution of an operating system based on the kernel Linux

[3] Clone is a Git [11] command line utility used to target an existing repository and create a copy of it

configurations and observe their effects, while providing valuable insights into the inner workings of processors, memory management, and external devices.

μMPS is an inspiring project that surely has reached its goals, but the μMPS architecture is based on MIPS, a family of instruction sets released in 1985. Although studying historical systems is significant, keeping up with cutting-edge technologies is equally crucial. The technology world is in continuous evolution, new techniques and project versions are released every year. RISC-V is a relatively new instruction set architecture, that lately has gained popularity, thanks to its features, such as modularity and scalability. Undoubtedly, RISC-V is yet to be perfect and at the moment it can not compete with other ISAs, such as ARM or x86. The RISC-V ecosystem is rapidly evolving, with new extensions, tools, and frameworks being developed [5]. While RISC-V offers simplicity and flexibility, there is room for research to further optimize its performance. The μRISCV project tries to introduce such an ISA to students and let them familiarize themselves with it. Students not only broaden their knowledge about computer architecture, but they study a technology that promises to revolutionize the whole computer landscape. μRISCV is an entry point to RISC-V, from where students can begin their journey and later on make beneficial improvements for the whole community.

Furthermore, μRISCV has introduced a notable debugging feature: support for GDB (see section 2.6). GDB is a widely used debugger that offers numerous functionalities. By incorporating support for this well-known debugger, experienced users can have a more seamless experience with the emulators. Simultaneously, students who are unfamiliar with GDB have the opportunity to explore and learn about this powerful tool, which can be utilized not only with μMPS3 but also in their future projects or professional endeavors.

In addition, μRISCV now includes the capability to run the emulator in a command-line interface mode. This feature enables experienced users and tutors to perform automated tests on projects created using this architecture, including projects like PandOS Plus (see chapter 3).

While the study of computer architecture may appear challenging, the students need to remember that even the construction of a skyscraper is done brick by brick. This analogy emphasizes the incremental nature of learning and highlights that complex concepts can be understood by breaking them down into smaller, manageable components. Just as a skyscraper is built by laying one brick at a time, students can approach computer architectures by gradually building their knowledge and understanding, mastering the fundamental concepts and later tackling more advanced topics. By taking it step by step, students can overcome the initial difficulty and gradually develop a solid understanding of computer architectures.

Bibliography

1. M. Biondi. "An Updated Emulated Architecture to Support the Study of Operating Systems". Bachelor's Thesis. URL: http://amslaurea.unibo.it/20751/.
2. Canonical Ltd. *Snapcraft*. URL: https://snapcraft.io/.
3. L. Cassel, M. Holliday, D. Kumar, J. Impagliazzo, K. Bolding, M. Pearson, J. Davies, G. Wolffe, and W. Yurcik. "Distributed expertise for teaching computer organization & architecture". *SIGCSE Bulletin* 33, 2001, pp. 111–126. DOI: 10.1145/571922.571965.
4. A. Cedilnik, B. Hoffman, B. King, K. Martin, and A. Neundorf. *CMake*. 2000. URL: https://cmake.org/.
5. E. Cui, T. Li, and Q. Wei. "RISC-V Instruction Set Architecture Extensions: A Survey". *IEEE Access* 11, 2023, pp. 24696–24711. DOI: 10.1109/ACCESS.2023.3246491.
6. Dijkstra and W. Edsger. "The Structure of the "THE" Multiprogramming System". In: *Classic Operating Systems: From Batch Processing To Distributed Systems*. Ed. by P. B. Hansen. Springer New York, New York, NY, 2001, pp. 223–236. ISBN: 978-1-4757-3510-9. DOI: 10.1007/978-1-4757-3510-9_12. URL: https://doi.org/10.1007/978-1-4757-3510-9_12.
7. Flatpak's Contributors. *Flatpak*. URL: https://www.flatpak.org/.
8. Free Software Foundation, Inc. *GNU Autotools*. URL: https://www.gnu.org/software/automake/.
9. Free Software Foundation, Inc. *GNU Project*. URL: https://www.gnu.org/.

10. Free Software Foundation, Inc. *The GNU Project Debugger*. URL: https://www.sourceware.org/gdb/.
11. Git's Contributors. *Git*. URL: https://www.git-scm.com/.
12. *ITiCSE '05: Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. Association for Computing Machinery, Caparica, Portugal, 2005, pp. 301–305. ISBN: 1595930248.
13. M. Goldweber and R. Davoli. *μMPS3 Principles of Operation*. lulu.com, 2020. ISBN: 978-1716476402. URL: http://wiki.virtualsquare.org/education/doc/uMPS3princOfOperations.pdf.
14. M. Goldweber and R. Davoli. *Student Guide to the PANDOS Project*. lulu.com, 2020. ISBN: 978-1-716-31559-6. URL: https://hdl.handle.net/11585/788693.
15. M. Goldweber, R. Davoli, and M. Morsiani. "The Kaya OS Project and the μMPS Hardware Emulator". *SIGCSE Bull.* 37:3, 2005, pp. 49–53. ISSN: 0097-8418. DOI: 10.1145/1151954.1067462. URL: https://doi.org/10.1145/1151954.1067462.
16. J. Heinrich. *MIPS R4000 Microprocessor User's Manual*. URL: https://groups.csail.mit.edu/cag/raw/documents/R4400_Uman_book_Ed2.pdf.
17. Imperas. *riscvOVPsim*. URL: https://www.imperas.com/riscvovpsim-free-imperas-risc-v-instruction-set-simulator.
18. M. N. Ince, J. Ledet, and M. Gunay. "Building An Open Source Linux Computing System On RISC-V". In: *2019 1st International Informatics and Software Engineering Conference (UBMYK)*. 2019, pp. 1–4. DOI: 10.1109/UBMYK48245.2019.8965559.
19. T. Jonjic. "Design and Implementation of the uMPS2 Educational Emulator". Bachelor's Thesis. URL: http://amslaurea.unibo.it/4472/.
20. T. R. Kuphaldt. *Lessons In Electric Circuits*. Koros Press, 2012. ISBN: 1907653090.
21. N. Marriott. *tmux*. 2007. URL: https://github.com/tmux/tmux.

22. M. Melletti. "Studio e realizzazione dell'emulatore μARM e del progetto JaeOS per la didattica dei Sistemi Operativi". MA thesis. URL: http://amslaurea.unibo.it/11866/.
23. Microsoft Corporation. *GitHub*. URL: https://www.github.com/.
24. Microsoft Corporation. *Visual Studio Code*. 2015. URL: https://code.visualstudio.com/.
25. MITRE Corporation. *Common Vulnerabilities and Exposures*. URL: https://cve.mitre.org/.
26. T. H. Morris. "Experiential learning – a systematic review and revision of Kolb's model". *Interactive Learning Environments* 28:8, 2020, pp. 1064–1077. DOI: 10.1080/10494820.2019.1570279. eprint: https://doi.org/10.1080/10494820.2019.1570279. URL: https://doi.org/10.1080/10494820.2019.1570279.
27. M. Morsiani and R. Davoli. "Learning Operating Systems Structure and Implementation through the MPS Computer System Simulator". *SIGCSE Bull.* 31:1, 1999, pp. 63–67. ISSN: 0097-8418. DOI: 10.1145/384266.299683. URL: https://doi.org/10.1145/384266.299683.
28. N. Nisan and S. Schocken. *The Elements of Computing Systems: Building a Modern Computer from First Principles (History of Computing S.)* The MIT Press, 2008. ISBN: 9780262640688.
29. A. L. Pažitnov. *Tetris*. [CD-ROM]. 1984.
30. S. Peter. *AppImage*. URL: https://appimage.org/.
31. G. Savaton. *EmulsiV*. URL: https://eseo-tech.github.io/emulsiV/doc/.
32. R. Stallman. "Using the GNU Compiler Collection", 2004.
33. A. Waterman, K. Asanovi, and SiFive Inc. *Privileged Specification*. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf.
34. WebFreak. *Native Debug*. URL: https://marketplace.visualstudio.com/items?itemName=webfreak.debug.
35. Woolbright, David, Zanev, Vladimir, Rogers, and Neal. "VisibleZ: A Mainframe Architecture Emulator for Computing Education". eng. *Serdica*

Journal of Computing 8:4, 2014, pp. 389–408. URL: http://eudml.org/doc/281459.

36. C. Yehezkel, W. Yurcik, M. Pearson, and D. Armstrong. "Three simulator tools for teaching computer architecture: Little Man computer, and RTLSim". *ACM Journal of Educational Resources in Computing* 1, 2001, pp. 60–80. DOI: 10.1145/514144.514732.
37. W. Yurcik and L. Brumbaugh. "A Web-Based Little Man Computer Simulator". *SIGCSE Bull.* 33:1, 2001, pp. 204–208. ISSN: 0097-8418. DOI: 10.1145/366413.364585. URL: https://doi.org/10.1145/366413.364585.

Acknowledgements

I would like to express my deepest gratitude to my family for their immense support, both financially and emotionally, throughout my journey to completing this degree. Their unhesitating encouragement has been a constant source of motivation for me, and I am truly grateful for their presence and belief in my abilities.

I am immensely grateful to my supervisor, Renzo Davoli, for providing me with the opportunity to undertake this insightful and fascinating project. This thesis has played a pivotal role in shaping my choice to pursue a master's degree and will undoubtedly influence my future endeavors.

I would like to extend my heartfelt appreciation to the "Ranzani" guys who have made my intense study days more enjoyable with their laughter and ice cream breaks. I cherish the afternoons spent with Luca, attempting to set up a router, and the moments of anxiety shared with Fabio and Gabriele while filling out the application for TUM. The breaks we took at the cafè machine, listening to Daniele's amusing stories and learning about Pokémon cards from Simone, hold a special place in my memories. I want to express my gratitude to Alessandro and Francesco for introducing me to such an amazing group. Gaia, thank you for supporting me through various issues with the student office and guiding me on how to handle them. Andreea and Federica, I apologize for talking too much, and I appreciate all the ice creams you bought for us while studying Programming Languages. I am especially grateful to Erik, who reminded me to persevere even when things got tough, as the effort would ultimately be worth it. Stefano, you have been an inspirational figure to me since the first time we spoke during class. Although Andrea left Bologna during

the second year, I want to express my gratitude for his genuine positivity, which has always had a significant impact on my life. I am truly grateful to Paolo for organizing the enjoyable day of cooking pizzas for the "Ranzani" guys, creating a cherished memory that I will never forget. Last but certainly not least, I want to thank Yonas, my first friend at the university and a member of the Man Cave, along with Lorenzo and Vlad, who are the fathers of MontagnolaPunk.

I am deeply indebted to Leonardo for his resolute support and patience throughout the past two years. He stood by me during my preparation for the IELTS exam, assisted me in crafting this thesis, and helped me write admission letters for university. It was not easy to be around me during those times, considering my neurosis, and for that, I am immensely grateful to have had him by my side during those crucial moments.

When I initially arrived in Bologna, I found myself alone and concerned about making friends or forming connections with my fellow students. I was a complete mess, lacking even the basic knowledge of how to cook pasta. Fortunately, Luca G. came into my life. He provided me with support and guidance, teaching me how to navigate life and face my struggles independently. I am filled with gratitude for the fact that we have crossed paths.

I want to express my sincere gratitude to Giada and Martina, my apartment mates since my first year of university. They not only tolerated my presence but also provided me with unwavering support. Knowing that they were at home after a challenging day was incredibly comforting and reassuring. I would also like to extend my thanks to Ilaria, the newest addition to the Gastone house. After a series of roommates with whom I struggled to feel comfortable, Ilaria brought a sense of ease and harmony to our living environment.