An Updated Emulated Architecture to Support the Study of Operating Systems

Relatore:
Prof. Renzo Davoli

Presentata da:
Mattia Biondi

Correlatore:
Prof. Michael Goldweber

Agli amici di sempre,
alla mia famiglia,
che non mi ha mai impedito nulla,
e a Jas,
senza la quale non sarei qui oggi.

Abstract

One of the most effective ways to learn something new is by actively practising it, and there is—maybe—no better way to study an Operating Systems course than by building your own OS.

However, it is important to emphasize how the realization of an operating system capable of running on a real hardware machine could be an overly complex and unsuitable task for an undergraduate student. Nonetheless, it is possible to use a simplified computer system simulator to achieve the goal of teaching Computer Science foundations in the University environment, thus allowing students to experience a quite realistic representation of an operating system.

μMPS has been created for this purpose, a pedagogically appropriate machine emulator, based around the MIPS R2/3000 microprocessor, which features an accessible architecture that includes a rich set of easily programmable devices. μMPS has an almost two decades old historical development and the outcome of this following thesis is the third version of the software, dubbed μMPS3. This second major revision aims to simplify, even more, the complexity of the emulator in order to lighten the load of work required by the students during the OS design and implementation. Two of these simplifications are the removal of the virtual memory bit, which allowed address translation to be turned on and off, and the replacement of the tape device, used as storage devices, with a new flash drive device—certainly something more familiar to the new generation of students.

Thanks to the employment of this software and the feedback received over the last decade, it has been possible to realize not just this following thesis, but also to develop some major improvements, which concern everything from the project building tools to the front-end, making μMPS a modern and reliable educational software.

Sommario

Uno dei metodi più efficaci per imparare qualcosa di nuovo è facendo pratica, e probabilmente, non esiste modo migliore di studiare un corso di Sistemi Operativi, se non scrivendo il proprio SO. È tuttavia necessario prendere atto di quanto la realizzazione di un sistema operativo, in grado di girare su una vera macchina hardware, sia un compito eccessivamente complesso e quasi inadeguato per uno studente universitario. Ciò non toglie che sia però possibile far uso di simulatori di sistemi semplificati rispetto alla realtà, al fine di riuscire nell'insegnamento dei fondamenti dell'Informatica in contesti universitari, permettendo così agli studenti di sperimentare con una rappresentazione di un sistema operativo abbastanza realistica.

μMPS è stato sviluppato proprio a questo scopo, un emulatore di sistemi pedagogicamente appropriato, basato sul microprocessore MIPS R2/3000, e dotato di un'architettura accessibile e di una ricca lista di dispositivi facilmente programmabili. μMPS conta quasi vent'anni di sviluppo, ora aggiornato tramite la seguente tesi alla sua terza versione, chiamata μMPS3. Quest'ultimo aggiornamento mira a semplificare, ancor di più, la complessità dell'emulatore, al fine di alleggerire il carico di lavoro richiesto agli studenti durante lo sviluppo e l'implementazione del sistema operativo. Due di queste semplificazioni sono la rimozione del bit di memoria virtuale, che permetteva l'attivazione e la disattivazione della traduzione di indirizzi, e la sostituzione dei nastri, utilizzati come dispositivi di archiviazione, con delle nuove unità di memoria flash—certamente più familiari alla nuova generazione di studenti.

Grazie all'utilizzo di questo software e dei feedback ricevuti, nel corso degli ultimi dieci anni, è stato possibile non solo realizzare la seguente tesi, ma anche apportare alcune importanti migliorie che riguardano tutti i campi, dagli strumenti di compilazione del progetto alla parte front-end, rendendo così μMPS un software pedagogico moderno e affidabile.

Contents

List of Figures

List of Tables

Listings

Notational Conventions

- Words being defined are *italicized*;

- A `typewriter-like typeface` is used for memory addresses, machine registers, instructions, file names, identifiers, and code fragments;

- Memory addresses and operation codes are given in hexadecimal and displayed in big-endian format;

- Bits of storage are numbered right-to-left, starting with 0;

- All diagrams illustrating memory are going from low addresses to high addresses, using a left to right, bottom to top orientation.

Chapter 1

Introduction

1.1 Background

The study and the consequent implementation of how an operating system works is, by now, a long-established and consolidated practice in many Computer Science curriculum. It is, actually, one of the crucial components of a computer and it is responsible for ensuring its basic operations, by coordinating and managing the system resources like processor, memory, devices and processes, thus allowing hardware and software to interface each other.

This is probably the first real example of "big project" which students should experience, thanks to the complex intercommunication system that has to exist between the different components of the machine, and the study of it allows to comprehend the most common software engineering practices.

The approach on practical contexts is essential to fully understand how a machine works behind the theoretical notions studied in the early stages of the course of study, and it is usually followed by the debate of which is the best teaching choice concerning processor architectures.

Obviously, there is not only one way of how a central processor unit can be implemented, and while older realizations—although applicable for educational purposes—are now obsolete and incompatible with current platforms, modern ones are designed to achieve high speed and quality, which makes

them overly complex and unsuitable for the pedagogic experience.

Over the years, the MIPS architecture has become one of the landmarks in this teaching choice due to its clean and elegant instruction set, despite being excessively convoluted to student's perception, because of the high level of details obscuring the basic underlying features of it. A potential solution to this problem is the adoption of a simplified computer system simulator, like μMPS, to bring together an adequate level of understanding and a realistic representation of a real operating system.

1.2   History of μMPS

μMPS is based on the machine emulator *MPS* [4], designed and realized by Professor Renzo Davoli and one of his graduate students Mauro Morsiani, in the late 1990s at the University of Bologna.

MPS, in turn, was created with the intention of recreating the same asset that a previous, fell out of use, machine emulator provided. *CHIP* [1, 2] (Cornell Hypothetical Instruction Processor) was its name, developed at the Cornell University and based on a made-up architecture that was a cross between a *PDP-11* and an *IBM S/370*. The operating system designated to run on it was named *HOCA* [3], a three phase/layer project, named after the Turkish word *hoca* ("schoolmaster" or "giver of good advice") which was proposed by one of its author, Özalp Babaoğlu, who is currently teaching at the University of Bologna.

The initial purpose of MPS was to bring back to life the layout and implementation experience of an operating system through an educational emulator, which could be run on real hardware. This practice was already possible in the past years when the architecture of the processor studied was the same available on real machines, but it has gone lost through the years because of the high-speed development of new and more complex technologies used on physical implementations.

MPS was able to emulate the MIPS R3000 processor along with five

other different device categories: disks, tapes, network adapters, printers and terminals. In addition, the Hoca project was updated and adapted to the new MPS architecture.

The concerned CPU was genuinely emulated together with its complex virtual memory management system, which was the main subject of the feedback received during class testing of MPS as a pedagogical tool at the University of Bologna and Xavier University, in an undergraduate operating systems courses, taught respectively by Renzo Davoli and Michael Goldweber.

It was tested through the implementation of a direct descendant of HOCA, *Kaya* [5], one of the variety of graduate-level projects that the emulator can support.

The urge of simplification led to the creation of μMPS, virtually identical to MPS but with the addition of a virtual memory management subsystem which had to resemble as much as possible to the conceptual one found in popular introductory OS texts. The only other difference was the new novice-friendly graphical user interface, significantly improved again in 2011 by Tomislav Jonjic during the development of the first major revision of the emulator, μMPS2 [6], which also implemented the support to up to sixteen MIPS R3000-style processors.

1.3   μMPS3

More than ten years have passed since the first release of μMPS and as many have passed from the moment it has been developed into a consolidated educational tool. At its peak, it was adopted by more than 50 institutions, proving that its initial purpose had been achieved.

Over time, this number has decreased, showing that a refresh of the tool was needed to regain the attention of new generations of OS instructors. The students' observations during this period of use were essential to understand which of these further changes were required to give μMPS a new life as an

useful pedagogical tool.

Some examples of these modifications would be the structure and management system of both physical and virtual memory, a nuance probably still too complex and confusing, which could feel like a regression, since the attempt to simplify it has led to the removal of the VM bit. This was originally introduced in the first version of μMPS, as a distinctive feature from its predecessor MPS. Virtual memory segmentation has been eliminated too, since it is considered as legacy in current architectures, making its implementation an unnecessary complication for the students.

There are also other changes related to the passage of the years, such as the replacement of memory tapes with flash drive devices, also known as "USB sticks", "SD Cards" or even "SSDs". Tapes are probably still studied in their operation nowadays, but they cannot be found in today's practical contexts, and therefore it could be difficult to understand for the new generation of students, who cannot find a match in current technological implementations.

Another consequence of the update of μMPS has been the shift from the historical and well established Autotools to a more current building tool, CMake, that speeds up the compilation process in addition to simplifying the structure of the project. It also fits best with the current graphical user interface of the emulator, originally built upon Qt4, which also undergoes to the migration to the new version, Qt5.

These and more changes are collected together in a major release of the project, μMPS3, consequently bringing even more reliability in terms of pedagogical tool.

Due to implementation aspects—unlike its predecessor μMPS2—this new version is not backward compatible with older versions.

1.4   Structure of the Document

This thesis will primarily focus on the modifications made on the previous version of the emulator, their implementation and the reasons to do it. This document is not intended to be used as a substitute of the manual *µMPS3 Principles of Operation* [7], but only as an accompanying guide to understand how the two versions differ from each other.

This first chapter aims to give a brief overview of why tools like µMPS are needed, its history to date and the considerations on why an update was necessary.

Chapter 2 and 3 will respectively describe the new memory management, as both physical and virtual ones have been updated, and the new exception handling mechanism. Both current and previous implementations will be compared in every aspect.

Chapter 4 will explain the advanced feature of the multiprocessor support. The CPUs initialization has been updated to comply with the new exception handling.

Chapter 5 is an introduction of the new flash device, which takes over tapes. The reasons why this replacement was necessary are described in the first part of the chapter, while the rest of the chapter illustrates its creation and its usage.

Chapter 6 is the less technical one, since it describes the migration from the old project building tool to the new one, and the renewal of the graphical user interface.

Finally, chapter 7 will provide a brief summary of the topics covered by this thesis and the reasons for the changes made. A shortlist of possible future implementation is also given.

Chapter 2

Memory Management

Like for the two prior versions, the memory subsystem of μMPS3 is divided into two views: physical and virtual. Both parts have undergone significant modifications in this major revision, and they will be described in detail in this chapter, alongside their old versions and respective changes.

The main reasons for the modifications made are the further simplification of the work required by the user for the kernel implementation and the better clarification of the internal view of the emulator memory, which, in the past, have caused some difficulties to the students.

2.1 Physical Memory

The physical address space is divided into two big areas [Figure 2.1]: a *BIOS reserved space*, from address `0x0000.0000` to `0x2000.0000`, and the *installed RAM*, from address `0x2000.0000` (`RAMBASE`) to `RAMTOP`.

This last value is calculated upon the one retrieved from the configuration file, settable from the configuration dialog of the machine in the front-end emulator, which goes from a minimum of 8 to a maximum of 512 memory frames. Being the size of each frame 4 kilobyte, μMPS3 can have from 32KB up to 2MB of installed RAM. Hence, the value of `RAMTOP` range from `0x2000.8000` to `0x2020.0000`.
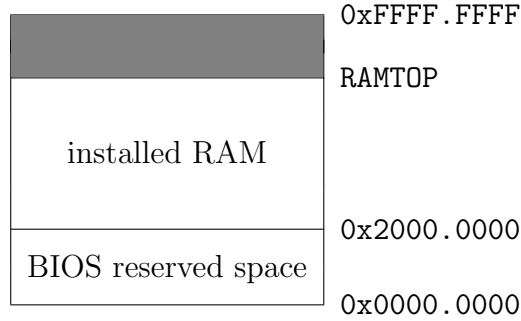
Figure 2.1: Physical Memory

2.1.1   BIOS Reserved Space

The first big area of the memory [Figure 2.2] is reserved for:

- the *Execution BIOS Services* [7], which lay in a read-only segment from address `0x0000.0000` to `EXECTOP`. This latter value is calculated upon the size of the Execution BIOS Services code;

- the *BIOS Data Page*, from address `0x0FFF.F000` (`BIOSDATABASE`) to `0x1000.0000`. This is the same 4KB area that was used to be called "ROM Reserved Frame" [subsection 3.3.1] in μMPS2, which has been renamed and moved. However, it still serves as an interface between the BIOS handlers [subsection 3.2.1] and the user-implemented kernel. See section 3.2 for more information regarding how this read/writable area segment has been updated;

- the *Bus Register Area*, from address `0x1000.0000` to `0x1000.002C`: a 11 word area containing various fields used in the lifetime cycle of μMPS3 to properly work. Some of them are read-only and calculated at boot/reset time, while others are read/writable. See [7] for more information regarding this area;

- the *Installed Devices Bit Map*, from address `0x1000.002C` to `0x1000.0040`: a read-only five word area indicating which devices are

actually installed and where [7];

- the *Interrupting Devices Bit Map*, from address `0x1000.0040` to `0x1000.0054`: a read-only five word area indicating which devices have an interrupt pending [7];

- the *Devices Registers Area*, from address `0x1000.0054` to `0x1000.02D4` (`DEVTOP`): a 160 word area (number of interrupt lines (8) × devices per interrupt line (5) × words per register (4)). See [7] for more information, or Table 5.1 for an example of one device register;

- the *Bootstrap BIOS Services* [7], which lay in a read-only segment from address `0x1FC0.0000` to `BOOTTOP`. This latter value is calculated upon the size of the Bootstrap BIOS Services code.

Any attempt to access an undefined memory area (`EXECTOP...0x0FFF.F000`, `DEVTOP...0x1FC0.0000`, `BOOTTOP...0x2000.0000`) will generate a *Bus Error exception* [chapter 3].

Note: the memory area between address `0x1000.02D4` (`DEVTOP`) and `0x1FC0.0000` is not completely undefined [section 7.3 of the old manual [8]].

2.1.2 RAM Space

The maximum `RAMTOP` value actually settable is `0x2020.0000`. However, since μMPS3 uses 32-bit addresses, it could support up to [2^{32} - `RAMBASE`] $\simeq$ 3.5GB of RAM. The 2MB graphical user interface limitation is imposed because, while it seems a ridiculously small amount of memory in today's ordinary systems, it is, actually, a lot more than has ever been required in all these years of μMPS use.

This area will hold:

- the code (*text*), global variables/structures (*data*), and stack(s) of the operating system;

(gray)	0x2000.0000
	BOOTTOP
Bootstrap BIOS Services	
	0x1FC0.0000
(gray)	DEVTOP
Devices Registers Area	
	0x1000.0054
Interrupting Devices Bit Map	
	0x1000.0040
Installed Devices Bit Map	
	0x1000.002C
Bus Register Area	
	0x1000.0000
BIOS Data Page	
	0x0FFF.F000
(gray)	EXECTOP
Execution BIOS Services	
	0x0000.0000
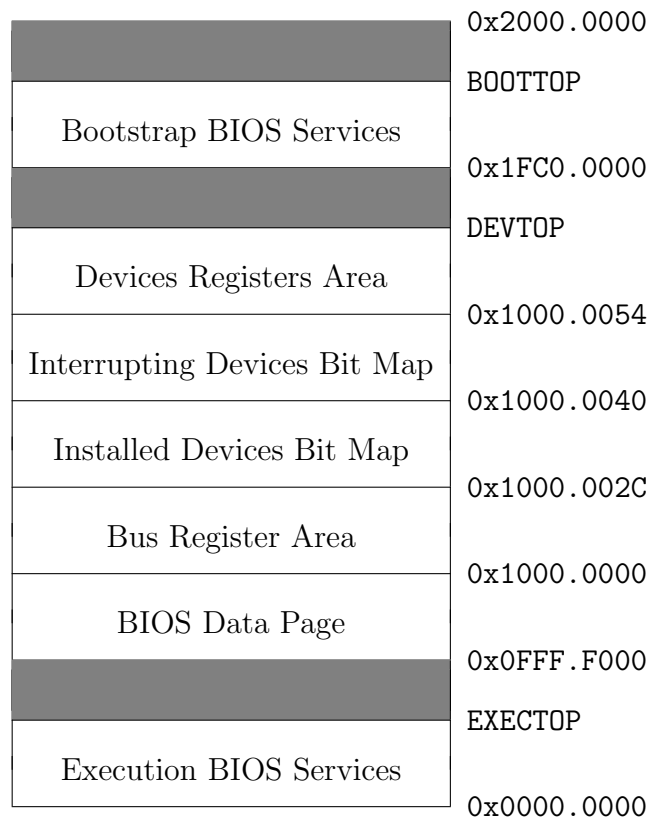
Figure 2.2: Structure of the BIOS Reserved Space

- enough space for the "frame pool" to be used for the text, data and stacks of the user processes during virtual memory management.

As for the previous big area, any attempt to access the undefined memory area laying from RAMTOP up to 0xFFFF.FFFF will generate a Bus Error exception.

2.2 Virtual Memory

While the physical one is divided into two big areas, the virtual memory subsystem is logically split into four units [Figure 2.3]:

- *kseg0*, from address `0x0000.0000` to `0x2000.0000`: a 0.5GB address space dedicated to the BIOS Reserved Space [subsection 2.1.1];

- *kseg1*, from address `0x2000.0000` to `0x4000.0000`: a 0.5GB address space dedicated to the text, data and stacks of the OS;

- *kseg2*, from address `0x4000.0000` to `0x8000.0000`: a 1GB address space dedicated to the text, data and stacks of the OS;

- *kuseg*, from address `0x8000.0000` to `0xFFFF.FFFF`: a 2GB address space dedicated to the text, data and stacks of the user processes;

Since the *TLB Floor Address* [subsection 2.2.1] minimum value is `0x4000.0000`, kseg0 and kseg1 are always referring to physical memory space.

```
                          ┌─────────────┐ 0xFFFF.FFFF
                          │             │
                          │    kuseg    │
                          │             │
                          ├─────────────┤ 0x8000.0000
                          │             │
                          │    kseg2    │
                          │             │
                          ├─────────────┤ 0x4000.0000
                          │    kseg1    │
                          ├─────────────┤ 0x2000.0000
                          │    kseg0    │
                          └─────────────┘ 0x0000.0000
```

Figure 2.3: Virtual Memory Structure

Despite their name, these four sections are not real segments, since virtual memory segmentation has been removed in μMPS3. However, they are called like this for traditional reasons, as their size and their label are the same as the areas from the original MIPS address space.

The first 2GB of the address space, corresponding to all the addresses below `0x8000.0000`, are reserved only to kernel-mode access, therefore, any attempt to access them while the processor is in user-mode will cause the raising of an *Address Error* [chapter 3].

The other half of the address space is dedicated to user processes, and each one "sees" its own logical version. The kuseg of one process is separated from the one of another process by a unique number called *ASID* (Address Space Identifier) [7].

2.2.1   TLB Floor Address

The TLB Floor Address is a new parameter implemented in μMPS3, responsible for indicating which addresses from a certain value on are considered logical, and therefore subject to *address translation*—the process of mapping logical addresses to physical ones. By doing this, the processes perceive more physical memory than the one actually available, recreating an abstract view of the address space.

This parameter can be chosen directly from the configuration dialog in the emulator front-end, as Figure 2.4 shows:

The last option, `VM OFF`, is internally declared as `0xFFFF.FFFF`: since it is not possible to access to an address above this value, the entire physical memory is necessarily below it, and therefore, an address translation could never happen.

2.3   Previous Implementation

The physical address space was substantially equal to the new one, except for the moving and renaming of the new "BIOS Data Page".

On the other hand, the virtual memory was implemented through a segmented-paged scheme. Hence, a new abstraction layer was laid on the whole physical memory structure, subdividing it into three big segments [Figure 2.5]:

- *ksegOS*, from address `0x0000.0000` to `0x8000.0000`: a 2GB segment reserved for the text, data and stacks of the OS, along with all the structures sitting in the BIOS reserved space up to address `0x2000.0000`;
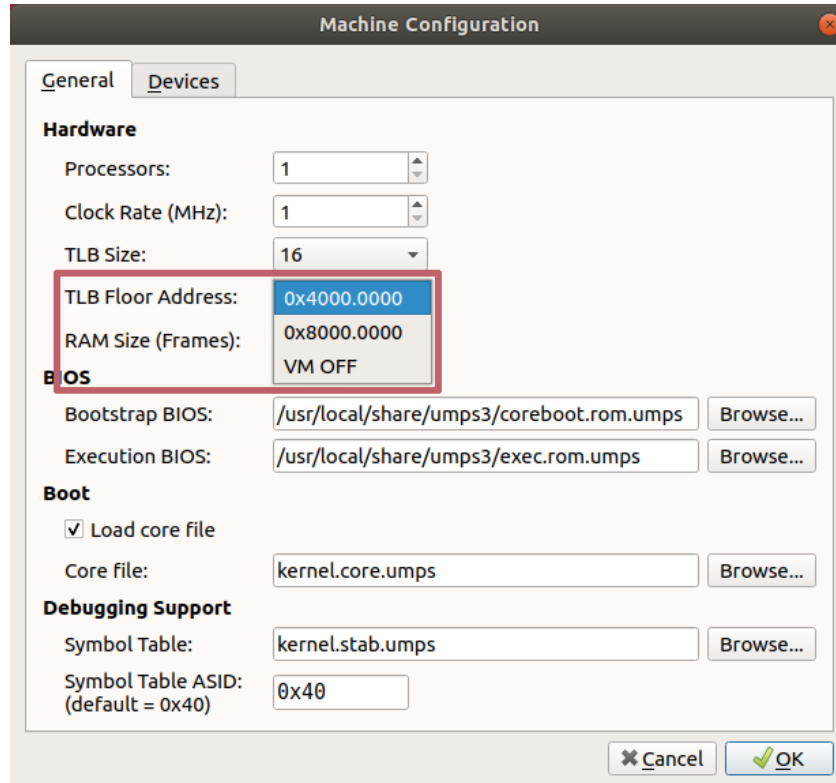
Figure 2.4: TLB Floor Address

- *kUseg2*, from address `0x8000.0000` to `0xC000.0000`: a 1GB logical address space reserved for the text, data and stacks of the user processes;

- *kUseg3*, from address `0xC000.0000` to `0xFFFF.FFFF`: another 1GB logical address space reserved for the use of user-mode processes.

As for the new version, all the addresses below `0x8000.0000` were reserved to kernel-mode access only.

Whether this subdivision was active or not was controlled by the *VM bit*. With the VM bit off, no particular actions were required when accessing the memory space. Instead, when the VM bit was on, all the address above `0x2000.0000` were considered logical and sent to the *MMU* for address translation.

The VM bit was originally introduced in the first version of μMPS, with

```
                                    0xFFFF.FFFF
            kUseg3
                                    0xC000.0000
            kUseg2
                                    0x8000.0000

            ksegOS

                                    0x0000.0000
```
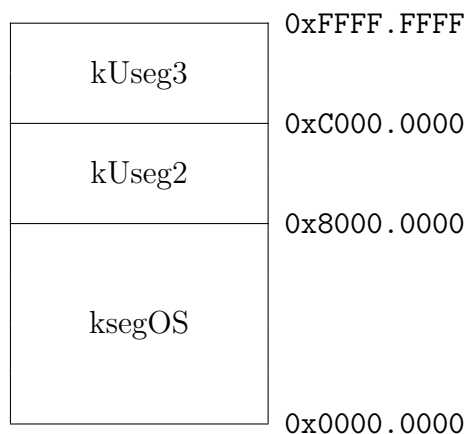
Figure 2.5: Old Virtual Memory Structure

the intent of simplifying the complex virtual memory management system of the MIPS R3000 microprocessor. The activation and deactivation of it was accomplished through the system control coprocessor *CP0*.

It provides a 32-bit *Status* register that controls various aspects of the emulator, described in details in [7]. Along with the current options set of the register, it also provided the possibility, by using bitwise operations, to control the status of the VM bit: it was implemented as a 3-slot deep bit stack (current, previous, previous of previous) which was pushed or popped depending on whether an exception was raised or an interrupted execution stream was restarted [section 3.2 and subsection 6.2.1 of [8]].

The virtual address translation process went through a significant change in this major revision: the formal segment table [subsections 4.3.1 of [8]], introduced in the first implementation of μMPS, have been removed. Consequently, only TLB table formats remain implemented, resulting in a simpler procedure for the users [7].

During the implementation of a new feature of the emulator, the VM bit has been completely removed too.

2.4   VM Bit Removal

The VM bit removal is one of the main features of μMPS3: the push/pop system has been removed, and the three bits previously reserved for it are now completely ignored. Hence, from now on, the virtual memory is conceptually always on. This means that all the addresses above a certain value are considered logical and therefore, subject to address translation. This specific value, while in previous version was fixed at `0x2000.0000`, is now contained in a variable called TLB Floor Address [subsection 2.2.1].

The TLB Floor Address value is stored in the Bus Register Area, thereby adding one word to the ten word area of the previous version [Table 2.1]:

Physical Address	Field Name
0x1000.0028	TLB Floor Address
0x1000.0024	Time Scale
0x1000.0020	Interval Timer
0x1000.001C	Time of Day Clock - Low
0x1000.0018	Time of Day Clock - High
0x1000.0014	Installed Bootstrap BIOS Services Size
0x1000.0010	Bootstrap BIOS Services Base Physical Address
0x1000.000C	Installed Exec. BIOS Services Size
0x1000.0008	Exec. BIOS Services Base Physical Address
0x1000.0004	Installed RAM Size
0x1000.0000	RAM Base Physical Address

Table 2.1: Bus Register Area

One of the consequences of the implementation of this value at the end of the Bus Register Area has been the shifting up of one word of the two successive areas, Installed Devices Bit Map and Interrupting Devices Bit Map.

The TLB Floor Address field cannot be changed while the emulator is running, but only from the front-end dialog.

After saving the machine configuration from the GUI, the TLB Floor Address value will be placed inside the JSON-based configuration file, described in detail in subsection 4.2.1 of Jonjic's Thesis [11].

An example of the final portion of a simple machine configuration is represented in Listing 2.1:

```
19      ...
20      },
21      "tlb-floor-address": "0x40000000",
22      "tlb-size": 16
23    }
```

Listing 2.1: TLB Floor Address

On line 21 is shown the value saved as *string*: this decision was taken to preserve the human-readable notational convention.

Chapter 3

Exception Handling

Exceptions are particular events interrupting the normal execution flow of a machine. Different categories of them exist, each one raised in correspondence of a certain occurrence. Depending on their type, the exceptions require to be handled in a distinct special way rather than the ordinary workflow of the system.

From a hardware perspective, μMPS3 supports two major types of exceptions: *TLB-Refill* exceptions, and all the other ones. However, from the software viewpoint, thanks to a high-level abstraction, they could be subdivided in four different categories:

- *Program Traps*: these exceptions are usually raised when the user-defined workflow commits an error, like: *Address Error*, *Bus Error*, *Reserved Instruction*, *Coprocessor Unusable* or *Arithmetic Overflow*;

- *System Calls and Breakpoints*: these two occur whenever a process requests an OS service through the `SYSCALL` or the `BREAK` instruction;

- *Interrupts*: I/O devices that complete their previously started operation must notify it to the processor, and they do it by raising this type of exception. There are a total of 8 interrupt lines: 3 of them are dedicated to internally generated interrupts, while the other 5 to external devices. Each of these last five ones support up to 8 devices attached

to them;

- *TLB-related exceptions*: exceptions linked to the *Translation Lookaside Buffer*: *TLB-Modification*, *TLB-Invalid* and TLB-Refill.

Going more in details on Program Traps, from [7]:

- Address Error is raised when:
    - a load/store/instruction fetch of a word is not aligned with a word boundary;
    - a load/store of a half-word is not aligned with a halfword boundary;
    - a user-mode access is made to an address below `0x8000.0000`;
- Bus Error is raised whenever an access is attempted on a non-existent physical memory location or when an attempt is made to write onto the BIOS device;
- Reserved Instruction is raised whenever an instruction is ill-formed, not recognizable, or it is privileged and executed while in user-mode;
- Coprocessor Unusable is raised whenever an instruction requiring the use of (or access to) an uninstalled or currently unavailable coprocessor is executed;
- Arithmetic Overflow is raised whenever an `ADD` or `SUB` instruction execution results in a two's complement overflow;

The TLB (Translation Lookaside Buffer) is a component of the Virtual Memory management system whose functioning has not changed. However, its exception handling did, and, therefore, a little background is needed to explain the modifications made. From [7]:

The TLB is a theoretical *associative cache*, that can hold from 4 to 64 TLB entries. Each TLB entry describes the mapping between one ASID-logical page number pairing and a physical frame

number/location in RAM. By utilizing a cache of recently used TLB entries, the virtual address translation mechanism of μMPS3 can avoid making multiple memory accesses for each translation. Since it is impossible to realistically emulate this kind of implementation, the TLB is therefore linearly searched.

Hence, regarding TLB-related exceptions, from [7]:

- TLB-Modification is raised when on a write request a "matching" entry is found, the entry is marked valid, but not dirty/writable;

- TLB-Invalid is raised whenever a "matching" entry is found but the entry is marked invalid;

- TLB-Refill is raised when no "matching" entry is found.

3.1 Processor Actions on Exception

When an exception is raised, the processor state (what it was "doing") must be saved in order to restart from that point after the exception has been handled. Before doing this, the processor has to take a number of atomic steps (non-interruptible, without any visible intermediary state), which are possible by:

- disabling all interrupts, to make sure nothing could disturb the entire handling process;

- activating the kernel-mode, since the access to the memory area intended for saving (and retrieving) the processor state is denied to user-mode processes.

Other essential processor actions are taken, some of them based on the type of the exception. They remain unchanged from μMPS2, and explained in detail in [7].

3.2   BIOS Data Page

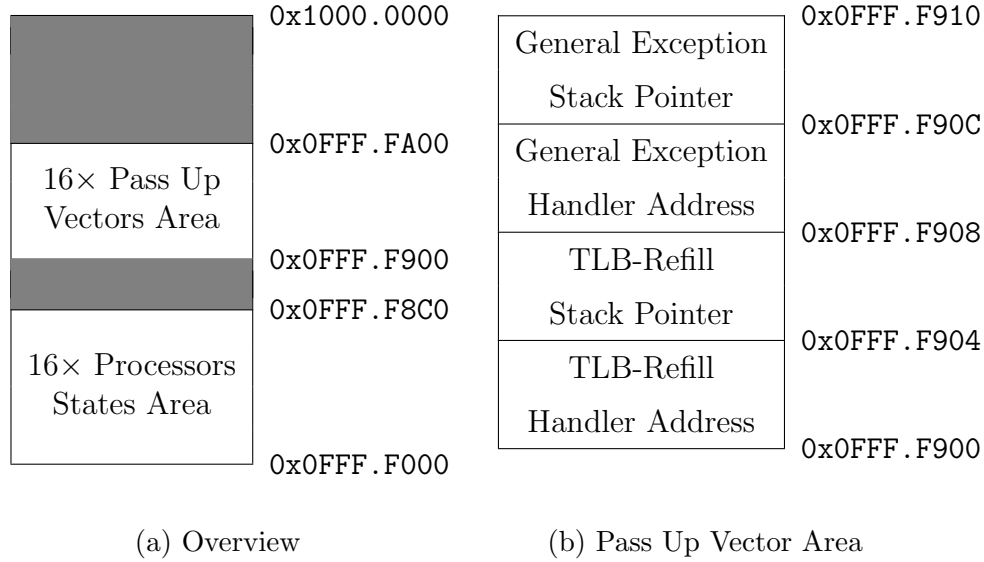(a) Overview   (b) Pass Up Vector Area

Figure 3.1: BIOS Data Page

The *BIOS Data Page* is the new 4KB area of μMPS3 reserved for the exception handling. As Figure 3.1a shows, it contains:

- a *Processors States Area*, from address `0x0FFF.F000` to `0x0FFF.F8C0`: a 35 (state size) × 16 (maximum number of active CPUs) word area dedicated to store the state loaded onto one processor at the moment an exception occurred. Since μMPS2 implemented the multiprocessor support, one area for each active CPU is needed. The area of processor 0 is at address `0x0FFF.F000`, the one of processor 1 at `0x0FFF.F08C` (+35 words) and so on;

- a *Pass Up Vectors Area*, from address `0x0FFF.F900` to `0x0FFF.FA00`: a 4 × 16 word area containing, for each active CPU:

  - the address of its *TLB-Refill handler*, to which the workflow will jump after saving the processor state;

- the stack pointer of the TLB-Refill handler;

- the address of its *general exceptions handler*;

- the stack pointer of the general exceptions handler;

Figure 3.1b shows this four-word area for the first processor. As for the previously reserved space, these areas are ordered one after the other, from low to high processors numbers. The processor 1 Pass Up Vector will be at address `0x0FFF.F910`, the processor 2 Pass Up Vector at `0x0FFF.F920` (+4 words) and so on.

This area is used as an interface between the two kernel handlers and the *BIOS Exception Handlers.*

3.2.1 BIOS Exception Handlers

The BIOS Exception Handlers are routines whose job is to prepare the environment on which the kernel exception handlers will work. This means that, after an exception has been raised, before the user-defined functions take place, a series of different actions are made by the BIOS, depending on the type of the exception.

Apart from some error checking, their essential task is to store off the processor state in the exact moment the execution flow has been interrupted in a "safe place" located in memory (the Processors States Area). Furthermore, the *Program Counter* will jump to the address of the appropriate kernel handler, thus ending the BIOS handlers job.

This "passing" set of instructions was internally distinguished into two main categories of events: TLB-Refill exceptions and all the other ones. The entire new exception handling system of μMPS3 is based on this distinction.

3.3 Previous Implementation

The exception handling of μMPS2 was managed by distinguishing them by their type: Program Traps, System Calls and Breakpoints, Interrupts and

TLB-related exceptions.

Since this subdivision existed, four different areas were needed to save the processor state (called *old areas*). The functions to subsequently call were treated as new states to load onto the processor, hence, four other areas were needed (called *new areas*), which contained the addresses of the exception handlers.

These four pair of areas, each one linked to its specific type of exception, were stored in a 35 (state size) × 8 word area [Figure 3.2a] at the beginning of the so-called *ROM Reserved Frame* [Figure 3.2].

3.3.1 ROM Reserved Frame

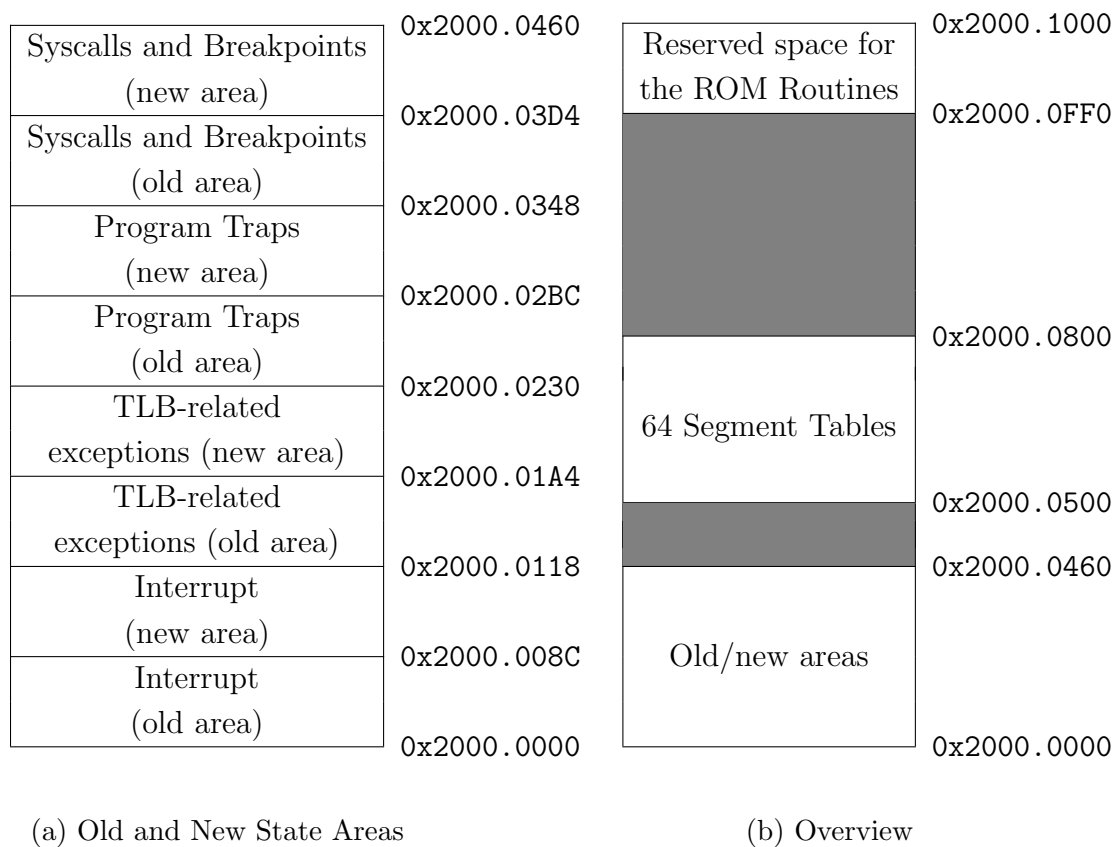(a) Old and New State Areas   (b) Overview

Figure 3.2: ROM Reserved Frame

The ROM Reserved Frame was a 4KB (frame size) area at the beginning of the installed RAM. It was used, apart from storing the old/new states of the processor, also to contain other structures, as Figure 3.2b shows:

- a *Segment Table*, from address `0x2000.0500` to `0x2000.0800`: a 64 (number of maximum concurrent processes) × 3 (number of µMPS2 segments) word area containing page table pointers [subsection 4.3.1 of [8]];

- a *Reserved Space*, from address `0x2000.0FF0` to `0x2000.1000`, dedicated to storing pointers to each old/new areas of the processors needed for the execution of the ROM (now called BIOS) exception handlers.

3.4 Differences from µMPS2

In addition to the two atomic actions described in section 3.1, another step to disable the virtual memory was taken in µMPS2, since the area reserved for the processor states was above the address `0x2000.0000` [Figure 3.2]. With the virtual memory bit enabled, those addresses were considered logical, leading to possible raises of TLB-management exceptions, which would have meant entering in an *infinite loop*. However, as explained in section 2.4, this complexity has been completely removed along with the virtual memory disabling process performed by the processor.

The 192 word Segment Table area [Figure 3.2b] has been eliminated too in µMPS3, since it is no longer used in the new address translation process.

Since all the internal exception handling was already dealt upon the separation of the "TLB-Refill/all the others" exceptions by the Execution BIOS Services, the current implementation elucidates this split to the users, who previously had to distinguish between four categories of events. While TLB-Refill events were previously handled by the BIOS, it is now a kernel responsibility; for this reason, two exception types, raised only during TLB-Refill events, have been removed.

In addition, the kernel handlers to which the control is passed are no more treated as new states to load onto the processor, but just as functions to which let the Program Counter jump to. This means that the four "new" processor state areas are no longer needed too, and has been reduced to only two one-word areas.

Two more one-word areas are needed for the two stack pointers of the handlers, something analogue to what was previously stored at the end of the first frame of RAM.

The ROM Reserved Frame, previously sat in the first frame of RAM, has been moved to the last frame of memory below the address `0x1000.0000` (where the Bus Register Area starts) and renamed into BIOS Data Page [Figure 3.1].

Finally, the BIOS exception handlers have been updated as well, simplifying the code which sometimes was redundant and in order to comply with the changes of the exception handling system. They have also been renamed from "ROM exception handlers" for a better overall analogy, since they belong to a reprogrammable BIOS and do not reside in a *Read-Only Memory* space.

How μMPS3 knows where to find the right area to save the state of each processor and where to retrieve the handlers is described along with the updated multiprocessor support in chapter 4.

Chapter 4

Multiprocessor Support

The first major revision, μMPS2, has implemented the multiprocessor support: the emulator is currently capable of supporting up to 16 simultaneously active CPUs. Their functioning has not been changed in this new version of the emulator, described in detail at chapter 7 of [8]. However, it is important to understand how their initialization has been modified in order to comply with the new exception handling [chapter 3].

4.1 Processors Initialization

As subsection 7.2.2 of [8] explains, each processor has a private set of five *Processor Interface Registers* [Table 4.1], and each one sees only its own private instance. While the first three fields have not changed in their working, the other two have.

Address	Register	Type
0x1000.0400	Inbox	Read/Write
0x1000.0404	Outbox	Write Only
0x1000.0408	TPR	Read/Write
0x1000.040c	BIOS Reserved 1	Read/Write
0x1000.0410	BIOS Reserved 2	Read/Write

Table 4.1: Processor Interface Registers

4.1.1   Processor 0

The first things loaded right after the machine has been turned on (or restarted), are the Bootstrap BIOS Services [7]. They are responsible for setting up the correct handling areas [section 3.2] for the default processor, the first one. *BIOS Reserved 1* and *BIOS Reserved 2* are used for this purpose.

In μMPS2, for the first processor, the BIOS Reserved 1 register was set, by the Bootstrap BIOS Services, to point the beginning of the Old/New State Area [Figure 3.2a], at address `0x2000.0000`. The BIOS Reserved 2 register was set to point the reserved space at the end of the first frame of RAM, at address `0x2000.1000`.

In μMPS3, at boot/reset time, the Bootstrap BIOS Services set up the BIOS Reserved 1 register to point the bottom of the BIOS Data Page [Figure 3.1], at address `0x0FFF.F000`, where the reserved exception state space for processor 0 starts. On the other hand, BIOS Reserved 2 now points to the Pass Up Vector of processor 0, at address `0x0FFF.F900`.

After that, the Bootstrap BIOS Services load the OS code.

4.1.2   Processors 1-15

How all the other processors were previously initialized is described in subsection 7.1.2 of [8].

To sum up, the *libumps* library provides a BIOS service designed to simplify the initialization of the processors. Libumps is a C library supplied with μMPS installation to provide access to CP0 instructions, the CP0 registers, and the extended BIOS-based services/instructions avoiding to program in MIPS assembler.

This specific BIOS service is called `initCPU`, and the original syntax was:

```
void INITCPU (uint32_t cpuid,
              state_t *start_state,
              state_t *state_areas);
```

In μMPS2, `initCPU` initiated the processor specified by `cpuid`, loading the processor state from the supplied `start_state` parameter onto it. The address of BIOS Reserved 2 was calculated by multiplying the processor number $\times$ the size of the reserved space the BIOS needed for the handlers of each CPU (32 bytes). Since a separate space was needed for Old/New Processor State Areas too, the third parameter, `state_areas`, was used to indicate the address for BIOS Reserved 1 [subsection 7.5.2 of [8]].

In μMPS3, the implementation of `initCPU` keeps only the first two parameters, calculating internally the correct areas where to save the processor state without needing the user:

```
void INITCPU (uint32_t cpuid,
              state_t *start_state);
```

The `start_state` parameter is used to initialize the `cpuid` processor as in μMPS2. The `cpuid` parameter is now multiplied $\times$ the size of a processor state (35 words) to get an offset to be summed to the bottom of the BIOS Data Page [section 3.2]: this value indicates the address of the processor exception state area, the same one BIOS Reserved 1 has to point to (e.g. `0x0FFF.F08C` for processor 1, `0x0FFF.F118` for processor 2 and so on). The other Processor Interface Register is set up similarly, multiplying the processor ID $\times$ the size of the Pass Up Vector area (4 words): this value is the offset from address `0x0FFF.F900` to which BIOS Reserved 2 has to point to (e.g. `0x0FFF.F910` for processor 1, `0x0FFF.F920` for processor 2 and so on).

Now, the BIOS routines only have to use BIOS Reserved 1 and BIOS Reserved 2 of a processor to retrieve the correct exception state area and the required Pass Up Vector.

Chapter 5

Device Interfaces

Since its first implementation, besides the MIPS R3000 microprocessor, μMPS has always been able to emulate five different device categories: disks, tapes, network adapters, printers and terminals. Furthermore, it can support up to eight instances of each device type.

This chapter will provide the reasons why, along with this major revision, one of the devices has been replaced with a new one.

As for the functioning and implementation of the remaining devices, the reader may refer to the manual [7].

5.1 Tape drives

MPS has been conceived and developed in the late '90s, and many updates and improvements have followed to make it as it is nowadays.

It is simple to imagine how much can technology achieve in over 20 years, and μMPS devices are not exempt from this process. Therefore, it may not be a surprise how hard disk drives are, nowadays, less and less used, outclassed by new solid-state drives, despite their still being common knowledge.

Network adapters are still present and properly taught in every IT class, although changed in their appearance, from physical to wireless, and certainly much faster, but the speed of the devices is not what concerns μMPS, on the

contrary, the goal is to teach their functioning.

Whereas there is not a real reason to think about the replacement of printers and terminals, there is one last device category about which the same cannot be said: tape drives.

In modern ages, tape drives are only studied in History of computer science classes, and are no longer found in real implementations if not in rare cases. As a result, only older users still know what tapes are, their functioning and their use, therefore new generations are slowly losing consciousness of what they are or, admittedly, never heard of them.

Given all of this, it would have been a misstep not to use them, as when μMPS2 was released in 2011, technology was going through a transaction, and those who were still students knew of them. However, after almost ten years, the situation has changed, and tapes are no longer used in common systems.

Tapes, in μMPS2, were implemented as read-only DMA (Direct Memory Access) devices, capable of transferring 4KB blocks of data per time, concurrently for each installed device of this category. Internally, they were divided into equal sized frames of 4KB each, through the use of four marked codes: `EOT` (end-of-tape), `EOF` (end-of-file), `EOB` (end-of-block), `TS` (tape-start), to scan the reading process and simulating the movement of a head on a tape.

In order to let the students work with a storage device slower than disks, but still familiar to them, μMPS3 had to completely remove tapes to implement a new category of devices: flash drive devices.

5.2   Flash Drives

A flash drive device, also know as "USB sticks" or "SD cards", is a storage device using flash memory, which is a solid-state memory, therefore not implemented through physical disks or tapes, that can be electrically erased and reprogrammed. Unlike disks or tapes, a "seek" operation—that moves a head assembly over a physical surface—is not needed before reading from

or writing to a specific block or sector.

In order to reproduce the same experience in μMPS3, the easiest way is to take a similar implemented device, the disk, remove the "seek" operation from it and simplify the coordinates system (cylinder, head, sector) to a single contiguous block addressable space [`0...MAXBLOCKS-1`].

Hence, a flash drive device is a read/writable DMA device, divided into equal-sized frames of the same dimension of μMPS3 framesize, 4KB, each one accessible via specific block number.

μMPS3 uses a 3 byte (24 bit) address space for flash device, consequently, each device in this category can have up to 2^{24} (`0xFFFFFF`) blocks of memory, which is equivalent to a maximum size of 64GB.

As already said, a flash drive device is usually slower than a disk device. However, the `umps3-mkdev` utility, which comes with μMPS3 installation and allows the user to create a disk or a flash device image, accepts a speed argument for both of them: "seek time" for disks and "write time" for flash devices. Therefore, this does not prevent anyone to voluntarily create a flash device significantly faster than a disk device. Theoretically, thanks to the implementation of this device, it would represent an SSD in every aspects.

The speed-related argument is optional, and, if not given, `umps3-mkdev` will use default values:

```
#define DFLSEEKTIME 100
#define DFLWTIME    DFLSEEKTIME * 10
```

Like intuitively understandable from the constants name, the first one is for the default disks seek time, while the latter is for the default flash devices write time. This means that, if not voluntarily indicated by the user who creates a device, the write time will always be ten times slower than the seek time for a default disk device.

The read time for a flash device cannot be directly defined, but it will be calculated *ad-hoc* when needed by multiplying the device set write time by:

```
#define READRATIO   3/4
```

This way, the read speed will always be 25% faster than write speed, since it will take 3/4 of time, almost as in real-world implementation.

5.2.1   Creation

As `umps2-mkdev` allowed to create image files of tape devices, `umps3-mkdev` permits to create flash device ones via the following syntax (also visible by running the utility with no arguments):

```
$ umps3-mkdev -f <flashfile>.umps <file> [blocks [wt]]
```

where:

- `-f` : specify the creation of a flash device image instead of a disk one (`-d`);

- `<flashfile>` : the name of the flash device image file that will be created;

- `<file>` : file to be written into the new flash device image;

- `blocks` : number of blocks [`1...0xFFFFFF`] (default = 512);

- `wt` : average write time (in microseconds) [1...100000] (default = 1000);

It is also possible to create an empty flash device image file by passing "`/dev/null`" as `<file>` argument.

Going more in detail, after correctly running the previous command, the utility will decode command-line arguments, if existing.

Then, it will attempt to open the flash device image file in write-mode. If the operation is successful, a `FLASHFILEID` (`0x0153504D`) will be written into the first word of memory of the file. This file recognition tag is used to distinguish this type of file from all the other types of files generated or recognized by μMPS3.

Number of blocks and write speed are also subsequently written, which, along with the `FLASHFILEID`, compose the *header* of the flash device. These

three parameters will be needed by μMPS3, when interacting with the device, to correctly simulate it.

Lastly, after opening it in read-mode, the existing file will be written inside the image file block-by-block.

Once the *end-of-file* indicator is reached, the remaining space will be filled with empty blocks. In case no errors occurred, the flash device image file will be successfully created.

5.2.2   Usage

The use of a flash device, in μMPS3, is not significantly different from the use of any other device. As reported in the manual *μMPS3 Principles of Operation* [7]:

> Every single device is operated by a controller. Controllers exchange information with the processor via device registers; special memory locations. A device register is a consecutive 4-word block of memory. By writing and reading specific fields in a given device register, the processor may both issue commands and test device status and responses. μMPS3 implements the full-handshake interrupt-driven protocol. Specifically:
>
> 1. Communication with device i is initiated by the writing of a command code into device i's device register.
>
> 2. Device i's controller responds by both starting the indicated operation and setting a status field in i's device register.
>
> 3. When the indicated operation completes, device i's controller will again set some fields in i's device register; including the status field. Furthermore, device i's controller will generate an interrupt exception by asserting the appropriate interrupt line. The generated interrupt exception informs the processor that the requested operation has concluded and that the device requires its attention.

4. The interrupt is acknowledged by writing the acknowledge command code in device i's device register.

5. Device i's controller will de-assert the interrupt line and the protocol can restart. For performance purposes, writing a new command after the interrupt is generated will both acknowledge the interrupt and start a new operation immediately.

The flash device registers are located in low-memory, starting at `0x100000D4` for `flash0` up to `0x10000144` for `flash7`. Each register consists of 4 fields:

Field #	Address	Field Name
0	(base) + 0x0	STATUS
1	(base) + 0x4	COMMAND
2	(base) + 0x8	DATA0
3	(base) + 0xc	DATA1

Table 5.1: Device Register Layout

The `STATUS` field of a flash device register is read-only and will contain one of the following status codes:

Code	Status	Possible Reason for Code
0	Device Not Installed	Device not installed
1	Device Ready	Device waiting for a command
2	Illegal Operation	Device presented unknown command
3	Device Busy	Device executing a command
4	Read Error	Illegal parameter/hardware failure
5	Write Error	Illegal parameter/hardware failure
6	DMA Transfer Error	Illegal physical address/hardware failure

Table 5.2: Flash Device Status Codes

From [7]:

Status codes 1, 2, and 4-6 are completion codes. An illegal parameter may be an out of bounds value (e.g. a block number outside of [`0..MAXBLOCK-1`]), or a non-existent physical address for DMA transfers.

The `COMMAND` field of a flash device register is read/writable and is used to issue commands to the device:

Code	Command	Operation
0	RESET	Reset the device interface
1	ACK	Acknowledge a pending interrupt
2	READBLK	Read the block located at BLOCKNUMBER and copy it into RAM starting at the address in DATA0
3	WRITEBLK	Copy the 4KB of RAM starting at the address in DATA0 into the block located at BLOCKNUMBER

Table 5.3: Flash Device Command Codes

The format of the `COMMAND` field is illustrated in Figure 5.1:

```
31                                              8 7           0
+-----------------------------------------------+-------------+
|                  BLOCKNUMBER                   |    CODE     |
+-----------------------------------------------+-------------+
```
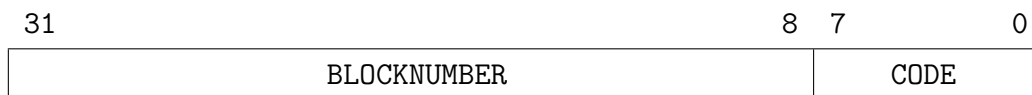
Figure 5.1: Flash Device `COMMAND` Field

An operation on a flash device is started by loading the appropriate value into the `COMMAND` field. For the duration of the operation, the device status is "`Device Busy`". Upon completion of the operation, an interrupt is raised and an appropriate status code is set: "`Device Ready`" for successful completion or one of the error codes. The interrupt is then acknowledged by issuing an `ACK` or `RESET` command.

The `DATA0` field of a flash device register is read/writable and is
used to specify the starting physical address for a read or write
DMA operation. Since memory is addressed from low addresses
to high ones, this address is the lowest word-aligned physical ad-
dress of the 4KB block about to be transferred.

The `DATA1` field of each device register is read-only and describes
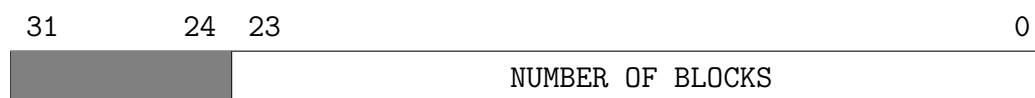the physical characteristics of the device geometry [Figure 5.2].

31   24 23   0

	NUMBER OF BLOCKS

Figure 5.2: Flash Device `DATA1` Field

Chapter 6

Project Structure Update

Many years have passed since the initial release of μMPS, and some aspects of it have become inevitably outdated. Along with all the changes described in the previous chapters, the software went through a refresh attempt which tried to update it to current standards and common procedures. Each modification made will be described in this chapter, together with the reasons to implement them, crucial to the aim of using the software for many years to come.

6.1 Build System

Developing software to use as a pedagogical tool requires being aware of some aspects. It will need to be spread among the students, hopefully, one copy per person. Schools and Universities usually dispose of computer labs, but the possibility of working on it from home is always appreciated. This will be particularly convenient when the software concerned will be used as a study aid in a computer class—and it is well-known that programmers work at any time of day (or night).

Given this premise, the tool should be installable on the students' personal computers, and one problem that does not usually show up on laboratories machines is the system diversity. The software developers could surely give

requirements and instructions on how to install it on different computers depending, for example, on the hardware, but when it comes to larger projects with lots of dependencies and steps to be able to run it, it is always preferable to use an automatic process. μMPS represents a classical example of this kind of big project, with a not so simple building and installing procedure.

In the late 1990s, when its development started, there was principally one solution: the *GNU Autotools* [9]. GNU Autotools were, *de facto*, the reference toolchain of programs helping developers to distribute their software fulfilling the users of complex procedures.

These tools are known as the *Build System* of a project, and they allow to define a series of instructions spread over files within the source code directory. The initial *configuration* phase will follow these instructions, generating a specific file depending on the operating environment on which it will run. From the programming view, they avoid distributing different procedures or source code for every system implementation, because, after setting everything up as needed, they will take care of it. From the user perspective, after obtaining the source code, the *building* phase will be possible in just a couple of predefined steps. It is also important to note that the building tools of a project are the first thing a user needs to interface to. In addition, for those not skilled students, the installation of the software could be their first-ever experience with them.

The Autools are still a very popular choice during the design stage of a software, but, with time, many other valid alternatives arose. For this reason, μMPS3 tries to simplify its installation phase by moving to a new building suite, which has been one of the most popular choices over the last few years: *CMake.*

6.1.1   CMake

Migrating μMPS3 to CMake has brought some benefits. First of all, whoever experienced the installation of the previous version can now notice a significant increase speed of this phase. CMake has a faster building time

due to the avoidance of numerous system checks that the other building tools do. More verifications mean more reliability, but most of Autotools checks were on system features which have been integrated by default in every distribution or in the Linux kernel [10] itself, making them useless.

CMake is also far less wordy, requiring a minor number of files in the source code directory and far fewer lines of code. Hence, from a programming perspective, the whole codebase is now "lighter".

Since a lot of relatively recent projects are based on this building tool, it should be easier to learn too, and this could encourage development contributions by novice programmers.

Lastly, μMPS2 have gone through a graphical user interface upgrade, discarding the original one and re-designing everything in *Qt*. This important process was necessary to keep up with current standards, but, unfortunately, the Autotools don't combine well with the new GUI of choice. It is surely possible to make them work together, as it has been until now, but this requires a heavy configuration, not recommended most of the time given the alternatives. μMPS3 could have just used the already implemented Autotools structure, but another update, almost completely incompatible with the old building tools, had to be done: the transition from Qt4 to Qt5.

6.2   Graphical User Interface

The μMPS front-end is one of its most important aspects, since it is the first thing a student sees after launching it. Being a pedagogical tool, one would expect it to be used by any level of experience in the area, hence, it should be the most intuitive possible. Nevertheless, this must not prevent the possibility to access every advanced feature directly from the GUI. Jonjic has perfectly succeeded in this goal implementing the current one, described in detail in the 3rd chapter of his thesis [11], and this third version of the software brings only an update of the one built by him.

6.2.1   Qt5

Qt is a toolkit for creating graphical user interfaces, and the currently available version is the fifth one. It has been originally released just a couple of years later than μMPS2, which is built upon the fourth one. The time to migrate to this new version has certainly come, and thanks to their high compatibility all the process went significantly smooth. The rule applied during the transition was the classical "if it is not broke, do not fix it", maintaining the same interface already implemented and upgrading just the essential.

From the user perspective view, μMPS3 tries to make a couple of aspects even more intuitive than before, thanks to the students feedback acquired during the past 10 years.

6.2.2   Logo and Icon Theme

The very first visible characteristic is the new icon theme. The decision to update has been made because the old ones were not much intuitive for some common actions, like the two dedicated for switching on and off the machine, compared along with the new ones in Table 6.1.

	Old	New
Power ON		
Power OFF		

Table 6.1: Start and Halt icons

These two icons were available as shortcuts in the top bar of the GUI. Since they were alternating each other (the one for turning on the machine was disabled once pressed, the other one was available only with the machine powered on), they have been merged in a single shortcut displaying only the action currently available.

Similarly, the same procedure has been done for the two "start" and "stop" icons.

Another reason for changing the icon theme was the replacement of tapes with flash devices, whose icon was not available in the previous one.

All the new images are in the SVG format, which is an advantage in years like these, where the resolution and DPI of computers screens are increasing quickly. With the drop of all the other PNG icons, these new ones could be easily scaled up in the future if needed, without losing quality.

The chosen theme is *Papirus* by Papirus Development Team [12], licensed under *GPL-3.0* [13]. Following the license terms, the one of μMPS3 has been upgraded from the original GPL-2.0 to the new GPL-3.0 too.

All the other non-available icons were obtained from similar existing ones with the *Inkscape* [14] software, following the same graphic style for the best eye pleasure possible. With the same procedure, the new μMPS logo has been created [Figure 6.1]. The font used for the central letter "μ" is *Ubuntu Monospace* from Canonical [15].
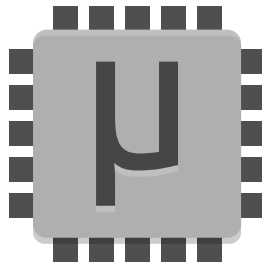
Figure 6.1: μMPS3 Logo

6.2.3   Other Improvements

There have been other small changes in the emulator front-end, intending to improve the user experience as much as possible.

The "*Edit configuration*" menu entry has been moved from the "Machine" menu voice to the "Simulator" one, in a more intuitive position [Figure 6.2].
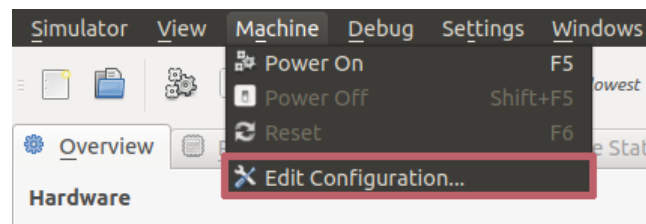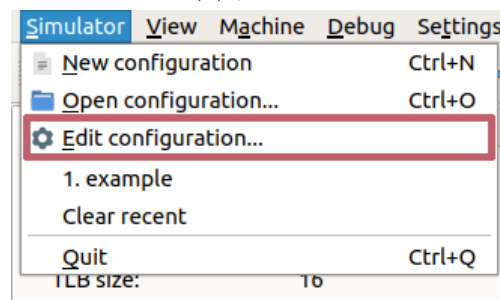
(a) μMPS2

(b) μMPS3

Figure 6.2: Edit Configuration Menu Entry

A "*Clear recent*" option has been added, an underestimated, yet still useful, function offering the possibility to remove all the used machine configurations from the "Recent menu" [Figure 6.3]. Previously, the only way to do it was by manually editing $\sim$/.config/umps3/umps.conf, not a very practical method to clear the emulator overview.
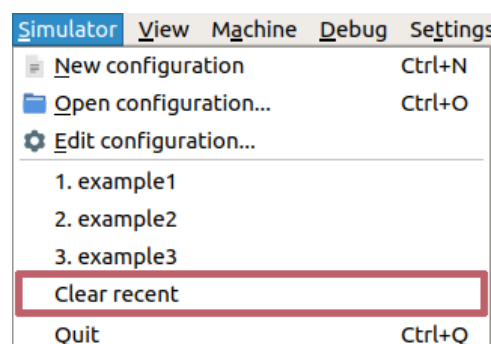

Figure 6.3: Clear Recent Menu Entry

The "*Reset*" shortcut has been made available in the processor view as well. This is a convenient option when debugging using only the keyboard, thus allowing the restart of the machine without the need to move between front-end windows [Figure 6.4].
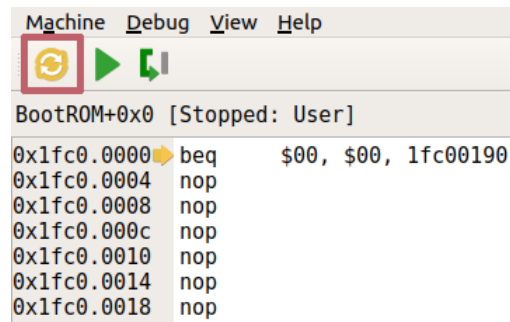

Figure 6.4: Reset Shortcut

The `RAMTOP` value of the machine is now available from the configuration overview [Figure 6.5], increasing the transparency for the user on how the physical memory of μMPS works [section 2.1].
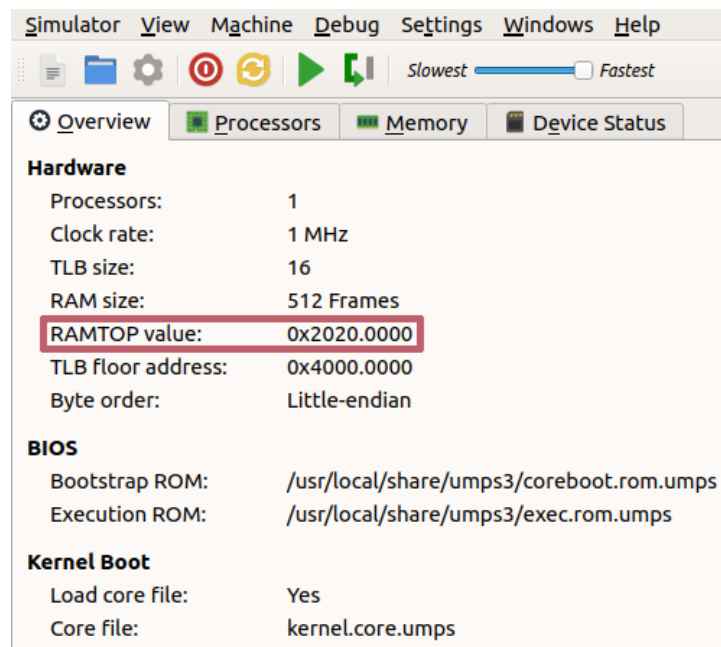

Figure 6.5: `RAMTOP` value

Chapter 7

Conclusions

The μMPS project was created almost twenty years ago as a didactic support for operating system courses. Over time, many instructors have chosen it as an aid in their classes, allowing it to reach its initial goal. However, from the day the last version has been released, educational and technological standards have changed in many ways, and the consideration of it has been dropping to these days. A refresh of the emulator was needed, in order to renovate the outdated features and reintroduce it to new OS instructors. Thus, μMPS3 has been designed and implemented, and describing how this update differs from the previous version is the work of this thesis.

The most changed aspect is, probably, the whole memory system, which has been simplified compared to the original design, aligned to modern management techniques, and, ironically, unintentionally brought back to the original MIPS architecture in various facets.

The internal exception handling system (BIOS routines) has been completely redesigned, facilitating its operation and requiring less work from the students. By moving some responsibilities from the kernel to the BIOS, the users kernel designing job is simpler and less sophisticated. This new implementation led also to the modification of the processors initialization process.

From the set of five previously supported devices, the tape is the one

which suffered the most the passage of time. Very few μMPS users from the current generation of students are aware of how it works or what it even is. Current hardware like USB sticks or SD cards is surely more appropriate for educational purpose, since a familiar match of what one is working on helps to understand better. For these reasons, the new class of flash devices has been introduced, filling the need for substituting another outdated view of the project.

Last but not least, the project building process had to be simplified. It was probably the very first (underestimated) obstacle that students encountered while trying to install μMPS, and it was based on the GNU Autotools. The migration to a new, current, building tool like CMake made the whole procedure more linear and pretty straightforward. The graphical user interface has not been revolutionized, but it has been updated anyway from Qt4 to Qt5 with the addition of a few adjustments.

With this second major revision, the μMPS project is once again in line with how a proper current operating systems study aid should be structured.

7.1 Future Work

μMPS is one of those projects where the work to be done is potentially endless. Uncountable changes are possible, both big or small, in order to enhance the quality of the software. The most important, obvious or already planned proposal, will be explained in this final part of the thesis, but it is important to remember that no matter how many examples will be given, it will always be a non-exhaustive list.

7.1.1 Software Packaging

Once a software has been released, it has to be distributed to future users. μMPS is an open-source project, and its code will be always available in public *Git* [16] repositories like *GitHub* [17]. This allows it to be always built from source and installed in almost every system. However, it is always

preferable to install software from official repositories when available, since it permits an integrated tracking of installed files and an automatic updates system. Therefore, the first task after μMPS3 release is packaging it for Linux distros, to simplify even more its installation process, which will also permit a solid future updates distribution.

Debian [18] is the main distribution for which the software must be packaged, since along with all its derivates it covers the majority of the most popular distros used. The guidelines of its packaging process are the most stringent, but once the package is complete the porting to other derivates repositories should be straightforward.

The second major distribution that is taking hold in these days is *Arch Linux* [19], along with a long list of derivates as well. The packaging process for this distribution is a lot more simpler than for Debian, and users-produced packages can be uploaded in the dedicated repository at any time in a few steps. The second version of the software, μMPS2, has been available in the *AUR* (Arch User Repository) for over a year.

There are obviously a large number of other Linux distributions not based on Debian or Arch, for which the two packaging processes listed before are, then, useless. Satisfying all the possible Linux "flavours" is an almost infinite work, and it is hoped that others will contribute to this cause by creating the package for the distribution they use.

Other possible solutions are distro agnostic packages, like *Flatpak* [20], *AppImage* [21], and *Snapcraft* [22]. They provide a way to distribute software that could run on every system regardless of the distribution used.

7.1.2 Persistent Debugging Settings

Thanks to the advanced debugging support, μMPS is unique in its kind. All the related features allow a high-level dismantling of the machine workflow, that enables the user to exactly understand what the emulator is doing or where the code is failing. Sadly, the settings environment related to a machine that can be created at runtime cannot be saved on the disk, and it

is lost when closing the program.

A nice new feature to develop could be linked to this aspect, since storing off all the breakpoints, suspects, targets or all the traced regions set during an hour of debugging could be convenient. Later in time, one could load the setup previously saved and start debugging from where left off. Undoubtedly, a cycle of compatibility checks is necessary to avoid loading incompatible debugging settings on a machine.

7.1.3 Better Block Devices Creation

A block device is a type of file which represents a device of some kind internally block-structured. This means that its data is read or written in it in blocks. μMPS supports two types of block devices: disks and flash devices. During their creation, a certain amount of space is always left blank, not loaded with any information given. This space is currently filled with empty blocks, all set to the `0` value. Consequently, a certain amount of memory is occupied on the real disk, without any useful information to maintain. By re-designing the creation process, for example by using the `truncate` function, it would be possible to achieve the same result without filling the actual file on the disk with unnecessary information. A new implementation has already been proposed, and it should be integrated in a future minor release of μMPS3.

Furthermore, at the moment, it is possible to write only one file into a newly created flash device. An update to the creation process of the device should implement also the ability to load multiple files, as it was possible with tapes.

Similarly, disks creation does not allow to load any files into them, but only to generate blank devices. Along with the suggestions above, the `umps3-mkdev` utility should be altered to also permit the creation of non-empty disks images. Due to the physical characteristics of their geometry, this will be a nontrivial task compared to the filling process of flash devices contiguous space.

Bibliography

[1] O. Babaoglu, M. Bussan, R. Drummond, F. Schneider, *Documentation for the CHIP computer system*, Department of Computer Science, Cornell University, (Ithaca, NY, USA), Technical Report TR 83-584, December 1983.

[2] L. Alvisi, F. Schneider, *A graphical interface for CHIP* Department of Computer Science, Cornell University, (Ithaca, NY, USA), Technical Report TR 96-1591, June 1996.

[3] O. Babaoglu, F. Schneider, *The HOCA Operating System Specifications*, Department of Computer Science, Cornell University, (Ithaca, NY, USA), Technical Report TR 83-585, December 1983.

[4] M. Morsiani, R. Davoli, *Learning Operating Systems Structure and Implementation through the MPS Computer System Simulator*, in *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '99, (New York, NY, USA), pp. 63-67, ACM, March 1999.

[5] M. Goldweber, R. Davoli, and M. Morsiani, *The Kaya OS Project and the μMPS Hardware Emulator*, in *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '05, (New York, NY, USA), pp. 49–53, ACM, June 2005.

[6] M. Goldweber, R. Davoli, and T. Jonjic, *Supporting Operating Systems Projects Using the µMPS2 Hardware Simulator*, in *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '12, (New York, NY, USA), pp. 63–68, ACM, July 2012.

[7] M. Goldweber, R. Davoli, *µMPS3 Principles of Operation*, Lulu Books, 2020.

[8] M. Goldweber, R. Davoli, *µMPS2 Principles of Operation*, Lulu Books, August 2011.

[9] GNU Project, *Autotools*, `https://www.gnu.org/`.

[10] Linux's Contributors, *Linux Kernel*, `https://www.kernel.org/`.

[11] T. Jonjic, *Design and Implementation of the µMPS2 Educational Emulator*, University of Bologna, 2012.

[12] Papirus Development Team, *Papirus Icon Theme for Linux*, `https://github.com/PapirusDevelopmentTeam/papirus-icon-theme`.

[13] Free Software Foundation, Inc., *GNU General Public License Version 3*, `https://www.gnu.org/licenses/gpl-3.0.en.html`.

[14] Inkscape's Contributors, *The Inkscape Project*, `https://inkscape.org/`.

[15] Canonical Ltd., *Ubuntu font*, `https://design.ubuntu.com/font/`.

[16] Git's Contributors, *Git*, `https://git-scm.com/`.

[17] Microsoft Corporation, *GitHub*, `https://github.com/`.

[18] Debian Project, *Debian*, `https://www.debian.org/`.

[19] Arch Linux's Contributors, *Arch Linux*, `https://www.archlinux.org/`.

[20] Flatpak's Contributors, *Flatpak*, `https://www.flatpak.org/`.

[21] Simon Peter, *AppImage*, `https://appimage.org/`.

[22] Canonical Ltd., *Snapcraft*, `https://snapcraft.io/`.

Acknowledgements

Thanks to my mentors, Prof. Renzo Davoli and Prof. Michael Goldweber, who patiently guided me through the development of the project and the drafting of this document with invaluable advice. Their contagious passion in Operating Systems and everything related to it made the final part of my course of study the most enjoyable one.

Special thanks to Jasmine, who is, *de facto*, the third supervisor of the thesis.