Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

An Updated Emulated Architecture to Support the Study of Operating Systems

Relatore:
Prof. Renzo Davoli

Correlatore:
Prof. Michael Goldweber

Presentata da:
Mattia Biondi

Sessione Straordinaria
Anno Accademico 2018/2019

dedica

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 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 emulator's complexity 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 feed-backs 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 un ricco 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 realizare 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 every Computer Science's curriculum. It is, actually, one of the crucial components of a computer and it is responsible of ensuring it's basic operations, by coordinating and managing the system's 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 must experience, thanks to complex intercommunication system that has to exist between the different machine's components, 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 [1], designed and realized by Professor Renzo Davoli and one of his graduate students Mauro Morsiani, in the late 1990s at the University of Bologna.

The initial purpose 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 processor's architecture studied were the same available on real machines, but it has gone lost through the years because of the high-speed development of new 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.

The concerned CPU was genuinely emulated together with its complex virtual memory management system, which was the main subject of the feed-backs 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 *Kaya*'s [2] implementation, 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 [3], 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 in the two already acknowledged teaching courses.

As already mentioned, the students' observations during this period were essential to in-depth testing the μMPS emulator as a pedagogical tool, showing as a result that further changes were needed. Some examples of these modifications would be the memory's structure and management system of both physical and virtual ones, 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. On the other hand, other changes are related to the passage of the years, such as the replacement of memory tapes with flash drive devices, also know as "USB sticks" or "SD cards".

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 can't find a match in current technological implementations.

Another consequence of μMPS's modernization has been the shift from the historical and well established Autotools to a more modern building tool, CMake, that speeds up the compilation process in addition to simplify the project's structure. It also fits best with the current emulator's graphical user interface, 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 his predecessor μMPS2—this new version is not backward compatible with older ones.

`TODO: Debian Packaging`

1.4 Document's Structure

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 emulator's memory internal view, 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 *kernel 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 machine's configuration dialog 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`.

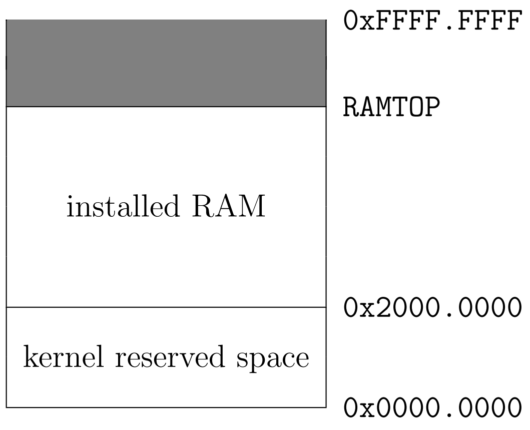


Figure 2.1: Physical Memory

2.1.1 Kernel Reserved Space

The first memory's big area [Figure 2.2] is reserved for:

- the Execution ROM code, which lays in a read-only segment from address `0x0000.0000` to `ROMTOP`. This latter value is calculated upon the size of the Execution ROM;
- the BIOS Data Page, from address `0x0FFF.F000` (`BIOSDATABASE`) to `0x1000.0000`. This is a new area that has been implemented in a μMPS2's unused memory space. See subsection 3.3.1 for more information regarding this read/writable area segment;
- the Bus Register Area, from address `0x1000.0000` to `0x1000.002C`: a 11 words area that will be described later in subsection 2.2.2;
- the Installed Devices Bit Map, from address `0x1000.002C` to `0x1000.0040`: a read-only five words area that indicates which devices are actually installed and where [`TODO` of [?]];
- the Interrupting Devices Bit Map, from address `0x1000.0040` to `0x1000.0054`: a read-only five words area that indicates which devices have an interrupt pending [`TODO` of [?]];

- the Devices' Registers Area, from address `0x1000.0054` to `0x1000.02D4` (`DEVTOP`): a 160 words area (number of interrupt lines (8) × devices per interrupt line (5) × words per register (4)). See `TODO` of [?] for more information, or Table 5.1 for an example of one device's register;
- the Bootstrap ROM code, which lays in a read-only segment from address `0x1FC0.0000` to `BOOTTOP`. This latter value is calculated upon the size of the Bootstrap ROM.

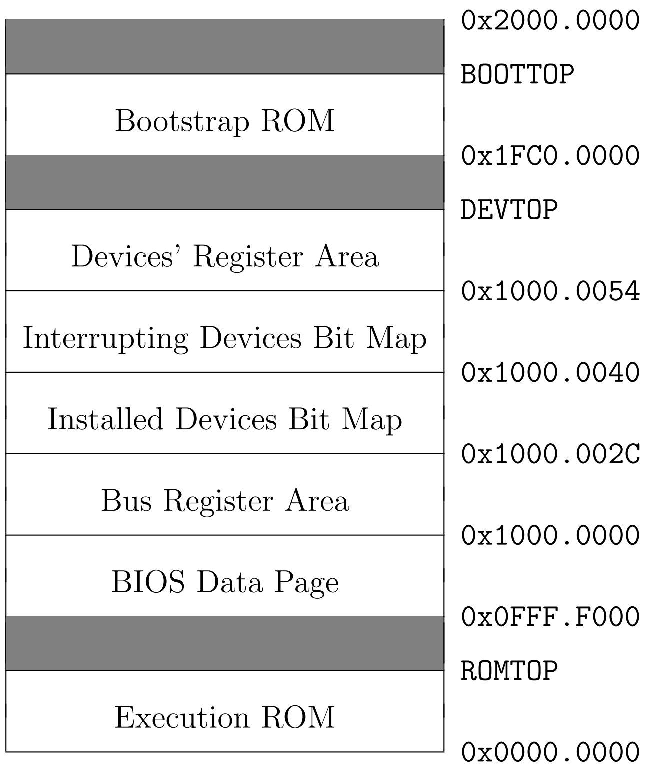


Figure 2.2: Kernel Reserved Space's Structure

Any attempt to access an undefined memory area (`ROMTOP...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 [4]].

2.1.2 RAM Space

The maximum `RAMTOP` value that could be actually set 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 operating system's code (*text*), global variables/structures (*data*), and stack(s);
- The user processes' *text*, *data* and stacks.

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

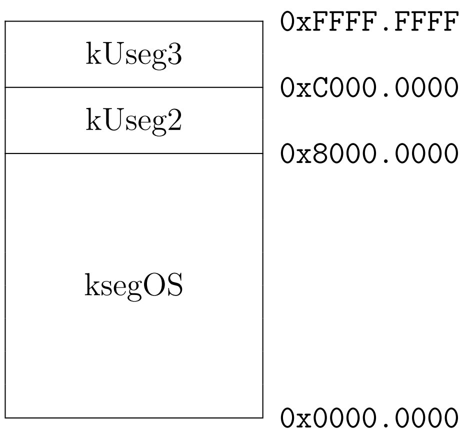
2.2 Virtual Memory

The virtual memory subsystem is implemented through a segmented-paged scheme. Hence, a new abstraction layer is laid on the whole physical memory structure, subdividing it into four big segments [Figure 2.3b]:

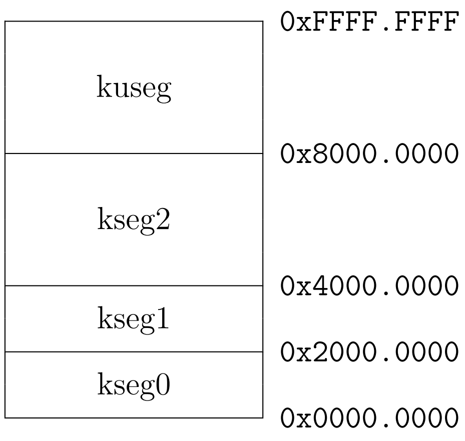
- *kseg0*, from address `0x0000.0000` to `0x2000.0000`: a 0.5GB segment reserved for the Kernel Reserved Space [subsection 2.1.1];
- *kseg1*, from address `0x2000.0000` to `0x4000.0000`: a 0.5GB segment reserved for OS *text*, *data* and stacks;
- *kseg2*, from address `0x4000.0000` to `0x8000.0000`: a 1GB segment reserved for OS *text*, *data* and stacks;
- *kuseg*, from address `0x8000.0000` to `0xFFFF.FFFF`: a 2GB segment reserved for user processes' *text*, *data* and stacks;

This subdivision differs from μMPS2's one, as it was divided in three different segments [Figure 2.3a]:

- *ksegOS*, from address `0x0000.0000` to `0x8000.0000`: a 2GB segment reserved for OS *text*, *data* and stacks, along with all the structures that sit in the kernel reserved space up to address `0x2000.0000`;
- *kUseg2*, from address `0x8000.0000` to `0xC000.0000`: a 1GB virtual address space reserved for user processes' *text*, *data* and stacks;
- *kUseg3*, from address `0xC000.0000` to `0xFFFF.FFFF`: another 1GB virtual address space reserved for the use of user-mode processes.



(a) Old Segments



(b) New Segments

Figure 2.3: Virtual Address Space

The first 2GB of physical memory, corresponding to all the addresses below `0x8000.0000`, were and 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 memory is, now, dedicated to user processes, and each one "sees" his own virtual version. One process' kuseg is separated from another process' one by a unique number called *ASID* (Address Space Identifier) [`TODO` of [?]].

In previous versions, wether this subdivision was active or not, it 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 virtual, and therefore subject to *virtual address translation*—the process to associate a non-accessible address to a physical one. By doing this, instead of the actual physical memory available, the processes perceive an abstract view of the address space.

The virtual address translation process went through a significant change in this major revision: the formal segment table and page table formats [subsections 4.3.1 and 4.3.2 of [4]], introduced in the first implementation of μMPS, have been removed, resulting in a simpler procedure for the users [`TODO` of [**?**]].

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

2.2.1 VM Bit Removal

The VM bit was originally introduced in the first version of μMPS, with 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 at `TODO` of [**?**]. Along with the current register's options set, it also provided the possibility, by using bitwise operations, to control the VM bit's status: 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 [4]].

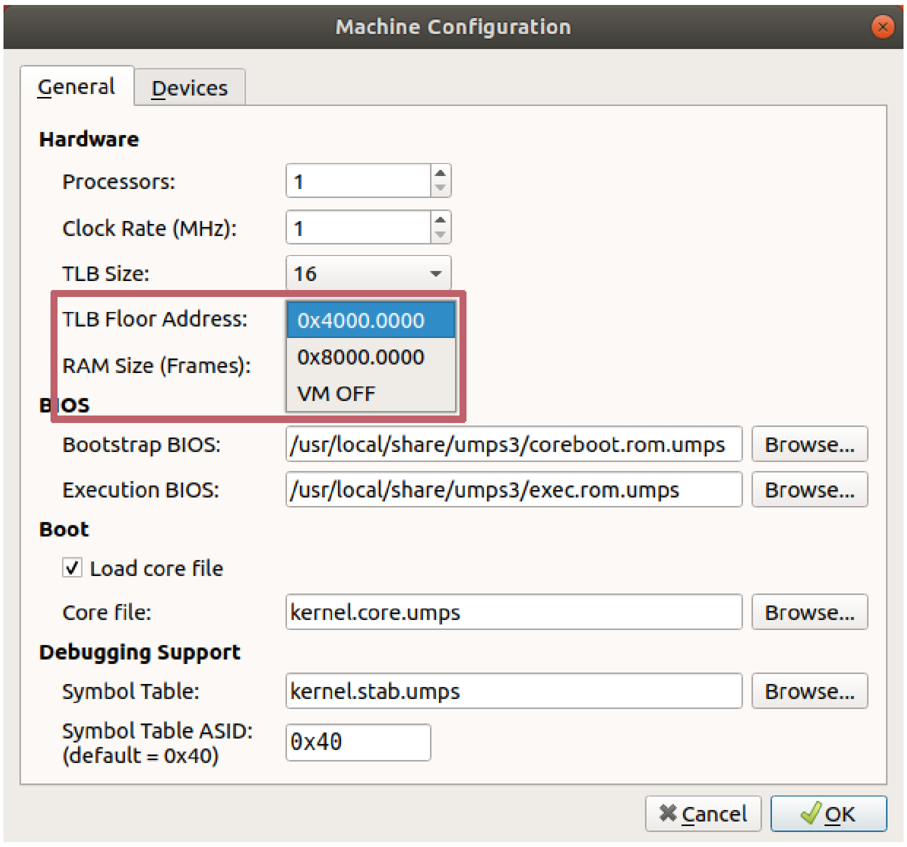
The VM bit removal is one of μMPS3's main features: the push/pop system has been removed, and the three bits that were reserved for it are now completely ignored. Hence, from now on, the virtual memory is conceptually

always on. This means that all the addresses above a certain value are considered virtual and therefore, subject to virtual address translation. This specific value, while in previous version was fixed at `0x2000.0000`, is now contained in a variable called *TLB Floor Address*

2.2.2 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 virtual.

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


Machine Configuration

General | Devices

Hardware

Processors: 1

Clock Rate (MHz): 1

TLB Size: 16

TLB Floor Address: 0x4000.0000 / 0x8000.0000 / VM OFF

RAM Size (Frames):

BIOS

Bootstrap BIOS: /usr/local/share/umps3/coreboot.rom.umps

Execution BIOS: /usr/local/share/umps3/exec.rom.umps

Boot

☑ Load core file

Core file: kernel.core.umps

Debugging Support

Symbol Table: kernel.stab.umps

Symbol Table ASID: (default = 0x40) 0x40

Figure 2.4: TLB Floor Address

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.

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

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

Table 2.1: Bus Register Area

This area contains various fields used in μMPS3's lifetime cycle to properly work. Some of them are read-only and calculated at boot/reset time, while others are read/writable. See `TODO` of [?] for more information regarding this area.

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

After saving the machine's configuration from the front-end dialog, the TLB Floor Address's value will be placed inside the JSON-based configuration's file, described in detail in subsection 4.2.1 of Jonjic's Thesis [5].

An example of the final portion of a simple machine's 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, used across all μMPS3's environment, consisting of addresses written in hexadecimal base, preceded by the "`0x`" prefix.

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.

Chapter 3

Exception Handling

Exceptions are particular events that interrupt a machine's normal flow of execution. 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 with the ordinary system's workflow.

μMPS3 supports four different types of exceptions:

- *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 request 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 `TODO` of [?]:

- Address Error is raised when:
  - a load/store/instruction fetch of a word is not aligned on a word boundary;
  - a load/store of a half-word is not aligned on 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 ROM storage;
- 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 2's complement overflow;

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

> The TLB is an *associative cache*, that can hold from 4 to 64 TLB entries. Each TLB entry describes the mapping between one ASID–virtual page number pairing and a physical frame number/location in RAM. By utilizing a cache of recently used TLB entries, μMPS3's virtual address translation mechanism can avoid making multiple memory accesses for each translation by linearly searching the TLB for a matching entry.

Hence, regarding its exceptions, from `TODO` of [?]:

- 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's 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 actions (non-interruptible, without any visible intermediary state). Some fundamental steps are, then:

- 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's state is denied to user-mode processes.

In addition to these, another step to disable the virtual memory was taken in μMPS' previous versions, since the processor's state reserved area was above the address `0x2000.0000`. With the virtual memory bit enabled, those addresses were considered virtual, leading to possible raises of TLB-management exceptions, which would have meant entering in an *infinite loop*. However, as explained in subsection 2.2.1, this has been completely removed in μMPS3.

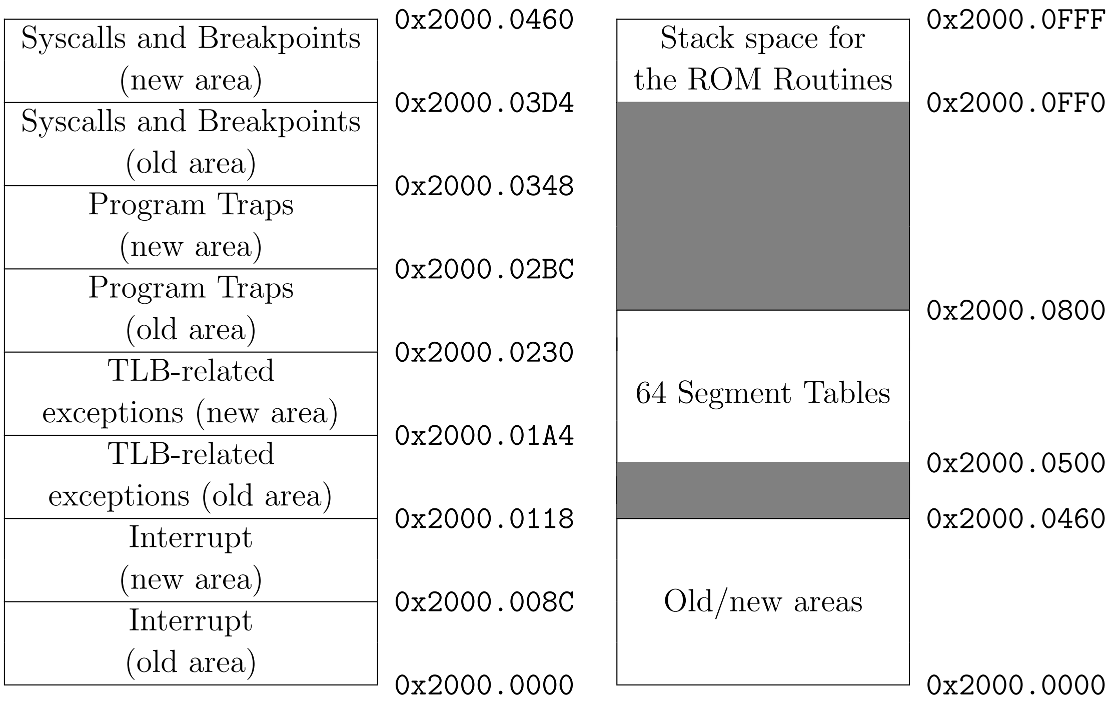
Other essential processor actions are taken, some of them based on the type of the exception. They remain unchanged from μMPS2, and explained in detail in `TODO` of [?]. This chapter will focus on the differences between the two versions and the modifications made.

3.2 Previous Implementation

The exception handling of μMPS' previous versions 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's state (called *old areas*), and other four to retrieve the state to be loaded (called *new areas*), which contained the address of the exception's handler. These four pair of areas, each one linked to its specific exception's type, were stored in a 35 (state's size) $\times$ 8 words area [Figure 3.1a] at the beginning of the so-called *ROM Reserved Frame* [Figure 3.1].

3.2.1 ROM Reserved Frame



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

Figure 3.1: ROM Reserved Frame

The ROM Reserved Frame was a 4KB (the size of a frame) area at the beginning of the installed RAM. It was used, apart from storing the old/new processor's state, also to contain other structures, as Figure 3.1b shows:

- a *Segment Table*, from address `0x2000.0500` to `0x2000.0800`: a 64 (number of maximum concurrent processes) $\times$ 3 (number of μMPS2's segments) words area that contained page table pointers [subsection 4.3.1 of [4]];
- a *Stack Space*, from address `0x2000.0FF0` to `0x2000.0FFF`, dedicated to the execution of the *ROM exception handlers.*

3.2.2 ROM Exception Handlers

The ROM Exception Handlers are routines whose job is to prepare the environment on which the kernel's 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 ROM (also called *BIOS*), depending on the exception's type.

Apart from some error checking, their essential task is to store off the processor's state in the exact moment the execution's flow has been interrupted in a "safe place" in memory (the old areas). Furthermore, a new state is loaded onto the processor from a previously set area (the new areas), the one containing the kernel's handler, thus ending the ROM's handlers job.

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

3.3 Current Implementation

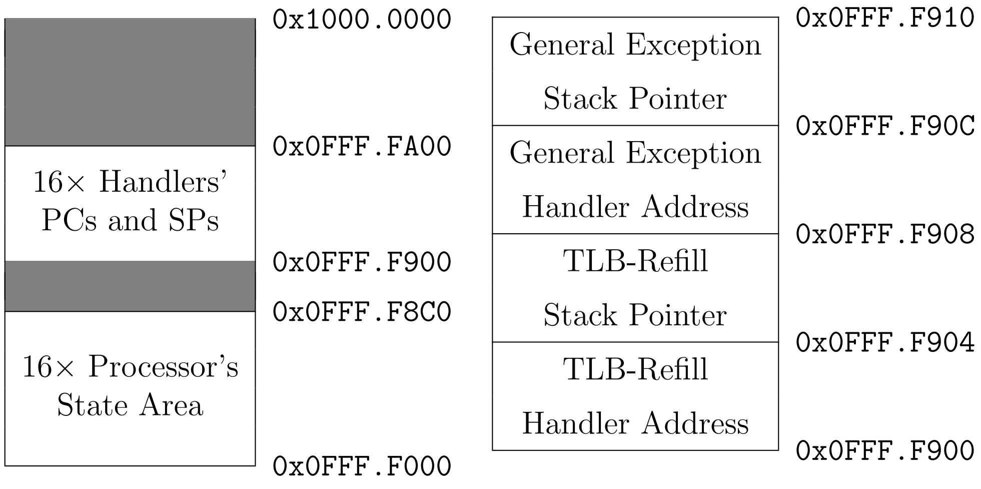
Since all the internal exception handling was dealt upon the "TLB-Refill/all the others" exceptions separation, the current μMPS' implementation brings it even to the front, to the user-perceivable part. Hence, the old processor's state areas have been reduced from four to two, one for each category.

In addition, the kernel's handlers to which pass the control to 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's 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 handlers' stack pointers, so the stack space that was previously reserved at the end of the first frame of RAM has been separated for this purpose.

Finally, 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), renamed into *BIOS Data Page* [Figure 3.2].

3.3.1 BIOS Data Page


0x1000.0000
0x0FFF.FA00
16× Handlers' PCs and SPs
0x0FFF.F900
0x0FFF.F8C0
16× Processor's State Area
0x0FFF.F000
General Exception Stack Pointer
General Exception Handler Address
TLB-Refill Stack Pointer
TLB-Refill Handler Address
0x0FFF.F910
0x0FFF.F90C
0x0FFF.F908
0x0FFF.F904
0x0FFF.F900

(a) Overview (b) Processor 0's Handlers Area

Figure 3.2: BIOS Data Page

The BIOS Data Page is the new μMPS3's 4KB area reserved for the exception handling. As Figure 3.2a shows, it contains:

- a *Processors' States Area*, from address `0x0FFF.F000` to `0x0FFF.F8C0`: a 35 (state's size) $\times$ 16 (maximum number of active CPUs) words 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. Processor 0's area is at address `0x0FFF.F000`, processor 1's one at `0x0FFF.F08C` (+35 words) and so on;

- a *Handlers' Area*, from address `0x0FFF.F900` to `0x0FFF.FA00`: a 4 $\times$ 16 words area containing, for each active CPU:

  - the address of its TLB-Refill handler, to which the workflow will jump after saving the processor's state;
  - the TLB-Refill handler's stack pointer;
  - the address of its general exceptions handler;
  - the general exceptions handler's stack pointer;

  Figure 3.2b shows this four-word area for the first processor. As for the previous reserved space, these areas are ordered one after the other, from low to high processors' numbers. Processor 1's one will be at address `0x0FFF.F910`, processor 2's one at `0x0FFF.F920` (+4 words) and so on.

Note: the 192 words Segment Table area has been completely removed, since it is no longer used in the new address translation process.

The ROM exception handlers have been updated as well, simplifying the code which was sometimes redundant and in order to comply with the exception handling's changes.

How μMPS3 knows where to find the right area to save each processors' state and 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 emulator's version, described in detail at chapter 7 of [4]. However, it is important to understand how their initialization has been modified in order to comply with the new exception handling [section 3.3].

4.1 Processors Initialization

As subsection 7.2.2 of [4] 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 thing that is loaded right after the machine has been turned on (or restarted), is the *Bootstrap ROM* [`TODO` of [?]]. It is responsible for setting up the correct handling areas [subsection 3.3.1] for the default processor, the first one.

BIOS Reserved 1 and *BIOS Reserved 2* are used for this purpose. For the first processor, the first one was previously set (in the Boostrap code) to the beginning of the Old/New State Area [Figure 3.1a], at address `0x2000.0000`, while the latter to the stack space at the end of the first frame of RAM, at address `0x2000.0FFF`.

At boot/reset time, BIOS Reserved 1 is now set up to point to the bottom of the BIOS Data Page [Figure 3.2], where its exceptions state's reserved space starts. On the other hand, BIOS Reserved 2 has now to point to the four-word area reserved for kernel's handlers and their stack pointers, at address `0x0FFF.F900`. After that, the Bootstrap ROM loads 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 [4].

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

This particular function is called `initCPU` [subsection 7.5.2 of [4]] and the original syntax was:

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

The ROM service's job was to initiate the processor specified by `cpuid`, loading onto it the processor's state from the supplied `start_state` parameter. The address of BIOS Reserved 2 was calculated by multiplying the processor's number × the size of the stack space BIOS needed for each CPU's handlers (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.

The current `initCPU` implementation keeps only the first two parameters, calculating internally the correct areas without needing the user. The `cpuid` parameter is now multiplied × the size of a processor's state (35 words) to get an offset to be summed to the bottom of the BIOS Data Page [subsection 3.3.1]: this value indicates the address of the processor's exception state area, the same one BIOS Reserved 1 has to be set to. The other field is determined similarly, multiplying the processor's number × the size of the handlers' area (4 words): this value is the offset from address `0x0FFF.F900` to which set BIOS Reserved 2.

The ROM handlers have now only to multiply the processor's number which raised the exception × the size of the area they need, and summing it to the respective base address (`0x0FFF.F000` or `0x0FFF.F900`). As concerns the handlers, taking advantage of how the four words are organized for each processor, every field is simply obtained by summing 4 (word size) the number of times needed from the base address.

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 `TODO` [**?**].

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 talking about 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's 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' final days have come.

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's 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's 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.

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

Flash devices' read time cannot be directly defined, but it will be calculated *ad-hoc* when needed by multiplying the device's 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 tape devices' image files, `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 a flash device image's creation instead of a disk's one (`-d`);
- `<flashfile>` : the flash device image file's name 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 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's argument, if existing.

Then, it will attempt to open, via a common `fopen`, the flash device's image file in write-mode. If the operation is successfull, a `FLASHFILEID` (`0x0153504D`) will be written into the first word of memory of the file by using an `fwrite` function. 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 by using an `fread` function, the existing file will be written inside the image file block-by-block.

Once the *end-of-file* indicator is reached, in case no errors have 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 on the manual *μMPS2 Principles of Operation* [?]:

> Each 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

A flash device's register `STATUS` field 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 [?]:

> 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.

A flash device's register `COMMAND` field 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:

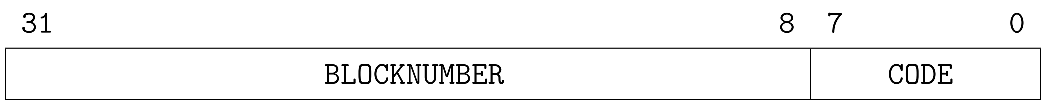


Figure 5.1: Flash Device `COMMAND` Field

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's 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.

A flash device's register `DATA0` field 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 address of the 4KB block about to be transferred.

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

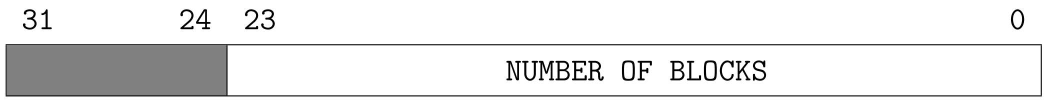


Figure 5.2: Flash Device `DATA1` Field

Chapter 6

Project Modernization

6.1 CMake Migration

6.2 Qt5 Transition

6.3 Logo and Icon Theme

Chapter 7

Linux Packaging

7.1 Debian Package

7.2 Arch Linux Package

Chapter 8

Conclusions

Knowing how an operating system works should be common knowledge and not something restricted only to the ones who studied in the IT field. If you are reading this document there are high chances you are doing it on a device of your property, which is running an operating system, and you should know how all of this really works in other to really feel like you own this particular system.

Bibliography

[1] 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.

[2] 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.

[3] 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.

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

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