UNIVERSITY OF COPENHAGEN

See KUDOS Run

A MIPS machine simulator

Jan Mezník
PZJ895

June 12, 2016

Abstract

This project covers the writing of a pipelined MIPS simulator, used to run the KUDOS operating system, developed at University of Copenhagen.
For the implementation, the required components, such as the memory-management unit and IO-mapped devices have been studied and implemented in the simulator, in order to fully utilize KUDOS.
Although the simulator at the current state is not able to run KUDOS, it is shown that it is very well possible, with only minor performance overhead to the host machine, compared to the older YAMS simulator.

Resume

Dette projekt omfatter udvikling af en "pipelined" MIPS simulator, brugt til at afvikle KUDOS operativsystemet, udviklet på Københavns Universitet.
De nødvendige systemer, krævet for at køre KUDOS, såsom memory-management unit og IO-mapped enheder, er blevet studeret og implementeret i simulatoren, så KUDOS kan blive kørt i dens fulde potentiale.
Selvom simulatoren i dens nuværende tilstand ikke er klar til at køre KUDOS, bliver det vist at det er muligt med meget små krav til ekstra omkostninger på ydeevnen, sammenlignet til den forrige YAMS simulator.

Overview

This report serves as documentation to the MIPS simulator, codename `See KUDOS Run`, written as a part of a bachelor project at University of Copenhagen. See KUDOS Run is a machine simulator, containing pipelined CPUs, memory-management unit, IO-mapped devices as well as external devices.
The simulator and the version-controlled source-code is located at `https://github.com/JanmanX/MIPS-Simulator`.

Building the simulator

After downloading the source, the simulator can be built using:

```
$ make
```

The simulator should be built in the `./bin/` directory as `mips-sim`.
A few test-programs are supplied along the simulator. These must be compiled using a cross-compiler. A shell-script installing the cross-compiler is supplied along side the simulator:

```
$ ./tools/cross_compiler.sh
...
$ ./run_tests.sh
```

Running the simulator

The simulator has a variety of command-line options, which can all be listed using the `-h` or `--help` flag.
Running the simulator can be as simple as only giving the program as the first command-line argument:

```
$ ./bin/mips-sim tests/test_add_addiu.elf
$ echo $?
60
```

To run a program with debugging enabled, the `-d` flag is supplied, which will let the user step the instructions:

```
$ ./bin/mips-sim -d tests/test_add_addiu.elf
PC: 0x80010000
0x2408000A ADDIU  rs = zero = 0x00000000,  rt = t0 = 0x00000000,  imm = 0x0000000A
> q
```

Preface

Updated Problem Statement

Is it possible to implement a pipelined MIPS simulator, to fully support the current version of KUDOS?

Target Audience and Prerequisites

This report is intended for anyone interested in computer architecuture, with the desire to expand the knowledge on the inner workings of MIPS32 processors.
The reader is expected to have either recently have taken a computer architecture course or read *Computer Organization And Design: The Hardware/Software Interface, 5th edition, by David A. Petterson and John L. Hennessy*, particularly focusing on the instruction pipelining within the processing unit.
Additionally, some fundamental understanding of computer hardware and related lingo is required, as well as a basic understanding of programming languages and data-structures.

Contents

1 Introduction

This report describes the development of a MIPS simulator, intended to support the operating system KUDOS. The simulator will be written in C, and will support the most important processor features and I/O devices, required to run KUDOS, such as the translation lookaside buffer (TLB), memory management unit (MMU), user and kernel CPU modes, multiple cores (SMP), and I/O device emulation.

1.1 Motivation

KUDOS is a small operating system skeleton intended to be used by students attending operating system project courses at university of Copenhagen. It is used to explore operating system concepts by extending and improving on existing system. Initially, KUDOS targets the MIPS32 architecture, which leverages on the advantages of a reduced instruction set computing — RISC.
To ease the development and debugging of KUDOS, it is desireable to run the operating system in a simulated machine. This enables the students and other developers to better inspect the state of the machine, as well as making up for the difference in the hardware of the host machine.

1.2 A simulator

A simulator is a program or a machine, that models some key characteristics and functions of a given target. The purpose of a simulator is to be able to look inside the simulation and inspect the properties and behaviours, that would otherwise only be seen in the real target. A possible by-product of a simulator is that the simulation model will emulate the target and its behaviour, practically immitating the target.
An emulator, on the other side, is intended to only mimick the behaviour of a target on the outside, not correctly reflecting the internal state of the target. Emulators are often used as a substitute for the real targets. The difference between emulation and simulation is therefore, that unlike simulating a system, an emulator only imitates the outward behaviour of its target, and it is hard to predict how the target would act internally.
For example, mobile developers often use an emulator to test their applications. Instead of having thousands of real smartphones, they can simply emulate[1] the devices on their computers, saving both time and money.
A common use of a simulator is in the aircraft business, where flight-simulators are being used for both pilot training, engineering, design, and many other purposes. While it gives a very good insight on how a simulated airplane might react in given situations, it does not actually move the users from one point to another.
The purpose of our project is to write a program used at the Computer Systems course at Copenhagen University, that can emulate KUDOS, while at the same time, exposes the internal workings of the machine, for the students and other participants to inspect and study.
Our target is therefore to simulate the academically important and relevant parts, while others will be emulated.

2 Background

KUDOS is heavily based on the BUENOS operating system, originally developed at Aalto University, Finland [5].
Although KUDOS has been extended for the Intel IA-32 architecture, it mainly supports MIPS32 architecture, just as its predecessor. The OS has been developed along side a machine simulator YAMS (*Yet Another Machine Simulator*), which is used to run the operating system. [5]

2.1 KUDOS and YAMS

BUENOS, and thus KUDOS, has been developed very closely with the YAMS sim-

[1] To be precise, they are emulating the hardware devices, but the applications they are testing, are being simulated.

ulator. Since the intetion of the OS is to be used at operating system courses, it is heavily dependant on the simulator, its interfaces, and its devices.
Despite being a very simple and basic simulation of a machine, it is still a very realistic hardware interface.

2.2 Analysis

The KUDOS operating system is written to fully utilize the many of MIPS32 features, which the students can research and extend.
To be able to run KUDOS correctly, the simulator must at least support the features listed below:

- `Translation Lookaside Buffer`
  Most noteably, KUDOS lets the students research and extend the Translation Lookaside Buffer (TLB), which is a cache used to translate virtual addresses [22]. It is an essential element in modern operating systems, and thus, very interesting to support in the simulator.

- `TTY device`
  A typewriter, more commonly known as a TTY is required, to be able to communicate with the operating system. These devices are usually very basic, only being able to transmit textual messages.

- `Shutdown device`
  The operating system needs a way to cleanly shutdown. The simulator will setup this device, and when required, will efficiently bring the system down.

- `Mapping of all memory segments`
  KUDOS may be extended to fully utilize the entire main memory. As memory is split into segments in MIPS32, it is essential for the simulator to be able to handle this.

- `Bootstrapping mechanism`
  KUDOS is shipped without a bootloader, relying on the simulator to correctly load the OS and start it from an appropriate point.

2.3 Design Goals

For the simulator to work flawlessly with KUDOS, it will be heavily inspired by the YAMS simulator. Since the OS is relying on the simulator, and vice versa, we do not want to make any drastic changes to the simulator, as to avoid rewriting parts of KUDOS. The main design goals of the simulator are:

- *Avoid breaking compatibility*
  For better integration of the simulator into the operating systems course, it is undesireable to make such big design changes, that would need to be patched in the OS as well. Even minor changes in KUDOS could break previously working code, such as assignments and exercises.

- *Simplicity over performance*
  The goal of the simulator is to seamlessly interface with the OS, while still being simple to understand. KUDOS is a very simple OS, and YAMS is by no means computationally intensive – the host machine should have no problems running both. Therefore, we want to focus on the readability rather than on the performance.

- *Usability*
  The simulator should be easy to use, stable, and flexible in regards of its functionality. Apart from running the OS and other simulated programs, the users should be able to stop the running program, to inspect the state of the simulated machine.

3 CPU Architectures

At the heart of every computer lies the Central Processing Unit (CPU), which is an electronic circuit that carries out the basic arithmethic- and logic calculation as well as process and redirect input and output to other devices in the computer, using the shared busses.

Most modern CPUs are contained on a very small, yet packed intergrated circuit

chip, which can also house memory caches, multiple cores, and other processing units.

The functionality of all processors is fundamentally the same. The processor executes some primitive operation by fetching an instruction in the form of binary signals, act upon the instruction and store the result in either one of the its registers, or in the main memory.

A single instruction does very little, but a collection of instructions make up a program. In the very early computing days, computers were programmed in an assembly language, which is simply human-readable instruction code. As the computers grew more powerful, more complex and much faster, larger programs could be executed. Because it is hard and time-consuming to write programs using only the assembly language, compilers are used to remove this complexity. A compiler takes a high-level language, such as C, C++ or Rust, and creates the corresponding assembly language program for the specific architecture, containing the instructions. This assembly language file is in turn assembled or translated to binary, that the particular CPU can understand.

Besides hiding the complexity of the underlying architecture away from the programmer, it can usually also compile programs to multiple architectures as well as optimising the code to run faster.

3.1 Instruction Set Architectures

The instructions supported by a particular processor is determined by the Instruction Set Architecture (ISA), which is the specification of how the CPU works. An ISA determines the instructions supported, the registers available, memory architecture, addressing modes as well as handling of interrupts.

There exist many different types of ISAs, with both their advantages and disadvantages. For example, some architectures have a very few instructions and registers, which is very practical for small embedded devices, whereas large servers might make use of a large array of registers for complex computations.

Besides the current use of an architecture, designers must also take into account its future uses and applications. As the world of computation is ever growing and evolving at exponential rates, the architectures must be up to the challenge of future computing. Introducing a completely new architecture to the market is very troublesome, and causes a list of problems. One of the main issues is that old software written for older architectures will no longer work, and it requires to be either recompiled, rewritten, or even emulated. One such example is the Intel Itanium (IA-64) architecture, which had a very bad marked reception due to its lack of backwards compatibility with the x86 architecture. The emulation of the architecture on IA-64 yielded suboptimal performance and ultimately lost to the AMD x86_64, which in turn was compatible. [9]

Indeed, there are a lot of factors to take into account when designing a new architecture, and every decision has big implications on the future of the whole ISA.

3.2 MIPS Architecture

The MIPS architecture (acronym for Microprocessor without Interlocked Pipeline Stages) was first created in the early 1980s [25]. MIPS is a reduced instruction set architecture (RISC), developed by MIPS technologies, to bring new levels of performance and efficiency into the world of processing units. As an RISC architecture, MIPS aims to implement only the most essential instructions, so that they in return can get highly optimised. This is the based on the RISC philosophy, that by implementing only the most common instructions, the architects can simplify the design and speed up the crucial parts of the instructions. This enables the processor to execute programs faster, but also removes a lot of complexity of implementing large programs.

In contrast to RISC, complex instruction set architecture (CISC) aims to reduce the number of instructions needed to execute a program by implementing instructions packed with functionality. This means that

a single instruction in CISC can execute several operations at once, such as loading from memory, arithmetic and storing. While complex programs indeed execute faster on a CISC architecture, the burden of implementing efficient and maintainable code and compilers can outweigh its advantages. [23]

Besides the inspiration from RISC, MIPS has added its own design principles, which are honored and used to question every change, implementation, or design. These are [22]:

- *Design Principle 1:* Simplicity favors regularity.
- *Design Principle 2:* Smaller is faster.
- *Design Principle 3:* Good design demands good compromises.
- *Design Principle 4:* Make the common case fast.

These decisions withstood the trial by fire and proved, that honoring these principles yields good design, easing implementation as well as simplifying hardware.

4 MIPS Core Processing Unit

MIPS CPUs are pipelined, meaning that they implement an instruction pipeline which enables them to execute different stages of multiple instructions at once. This gives the processor a higher throughput than would otherwise be possible at a given clock-rate. The processor has 31 general purpose (GP) registers, with additional registers per co-processing unit. A MIPS processor can have up to 4 co-processing units, the first one being required. Co-processing units add functionality to the processor, that might not otherwise be required. For example, the co-processor 1 contains floating-point logic specially designed to carry out operations on floating point numbers.
Even the first models of the MIPS CPUs had many features, such as the MIPS R2000, had memory caches and a translation lookaside-buffer, which improved the speed of the processor by reducing the number of main memory lookups.

4.1 Registers

MIPS CPUs have multiple types of registers. The most common and most used registers are the general-purpose registers (GP), which can be used for practically anything by the programmer. Special registers are registers implemented for cases where GP registers were either too small or otherwise unsuitable for the purpose.
For additional functionality, the MIPS co-processor 0 also has its own set of registers that, along with an operating system, bring many features to the system.

4.1.1 General Purpose Registers

In MIPS32, there are 32 general-purpose registers, all 32 bit wide. Although they can all theoretically be used however the programmer or assembler wants[2], there are some conventions for the use of the registers.

Mnemonic[3]	#	Use
`$zero`	0	Constant Value 0
`$at`	1	Reserved Temporary
`$v0-$v1`	2-3	Function Results
`$a0-$a3`	4-7	Function Arguments
`$t0-$t7`	8-15	Temporaries
`$s0-$s7`	16-23	Saved Temporaries
`$t8-$t9`	24-25	Temporaries
`$k0-$k1`	26-27	Reserved for OS
`$gp`	28	Global Pointer
`$sp`	29	Stack Pointer
`$fp`	30	Frame Pointer
`$ra`	31	Return Address

4.1.2 Special Registers

The special registers in MIPS cannot directly be accessed from the program.

[2] Except the 0'th ($0) register, which can only hold the value 0.

[3] Textual representation used in the assembly language

Rather, they are modified by dedicated instructions.

Name	#	Use
`$HI`	-	Hi-word of 64bit value
`$LO`	-	Lo-word of 64bit value
`$PC`	-	Program Counter

`HI` and `LO` registers are used to contain the result of a multiplication or division, which, using 2 32bit registers, can end with a 64bit result.
The PC register is pretty self-explanatory, as it simply points the current location in the program (or "counts" the instructions). On other architectures, this register is better known as the Instruction Pointer (IP).

4.1.3 Co-processor 0 registers

Registers in co-processor 0 are mainly used by the system, to provide features required to run an operating-system. The co-processor can have 32 registers, but only few of them are used consistently across the different CPU implementations. That is due to the wide applications of the MIPS architecture, where manufacturers only define and implement the truly necessary co-processor registers.

Name	#	Use
`index`	-	TLB entry index
`random`	-	TLB random access register
`entrylo`	-	Current low TLB entry
`context`	-	Page-Table lookup addr.
`vaddr`	-	Virtual address of exceptions
`entryhi`	-	Current high TLB entry
`status`	-	Processor status
`cause`	-	Exception cause
`epc`	-	PC when exception occured

The `status` register is a bit-field of flags used to signal the current state of the processor. It is similar to the `EFLAGS` register on x86 architectures.

The rest of the registers will be discussed in depth in the SMP chapter.

4.2 Instructions

Each instruction in MIPS is 32-bit long, aligned to word. This simplifies the instruction fetching, decoding, as well as disassembly of the program, for both the processor as well as the programmer.

In MIPS, the instructions have 3 basic formats:

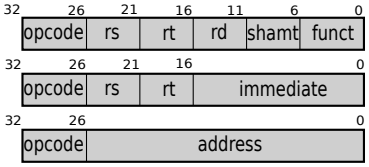

Figure 1: The 3 MIPS instruction formats.

It is clear that, as they have common fields, mainly the opcode field, they are easily distinguishable.
The R-format instructions are mainly used when all the data being processed is located in the registers. That includes adding between registers, binary operations on values in registers as well as jumping to an address located in a register.

The I-format instructions can operate on both data from registers and immediate values encoded directly in the instruction (thus the 16-bit immediate field). I-format instruction share a lot of common operations with the R-format, where one of the operands is the immediate.

J-format instructions are used solely for jumping instructions, thus the large address field. As it only has 26 bits to address an 32 byte memory location, it shifts the whole value twice to the left, as to align the value in words. The upper 4 bits are retrieved from PC. In practise, this is enough to jump to any address in the program.

4.3 Arithmetic Logic Unit

Without any extension co-processing unit, the Arithmentic Logic Unit (ALU) in MIPS32 only supports operations on integers.
The ALU supports basic mathematical operations such as adding (`add`), subtracting (`sub`), as well as logical shifting to both left and right (`sll, srl`), which also can be used to division or multiplication by even numbers.

All bitwise logical operators `and, or, and nor` are implemented Using these, which additional missing logical operations can be created, such as `nand` and `not`.
Since both the operand and destination registers are 32 bit wide, an overflow in the result might occur - that is, the result is larger than what a 32 bit register can hold. In that situation, an exception is raised in the processor, and code to recover from this error is run [22]. This will be explained in later sections.

4.4 Load and Store

MIPS is a "load/store" architecture, where memory is only accessed by specific load and store instructions [7]. This design is a very common for RISC architectures, as it greatly simplifies the pipeline stages and clock timings. In contrast, CISC architectures have many instructions that can do operations on both memory and registers at the same time. For example, on the x86_64 architecture, the `MOVSW` instruction reads from a memory location pointed to by register SI, stores it in memory location DI, and at last, increments (or decrements[4]) both registers [11]. This adds additional stall and hazard logic to the processor, and makes it is hard for the CPU to determine how many clock-ticks the instruction will take.

MIPS32 uses `lw` for loading a word from the main memory into the register, and a `sw`, which stores the value from register into the specified memory location. In reality, MIPS32 also has `LH`, `LB` and their store counterparts `SH SB`, which operate on half-word and byte sized loads and stores. However, for performance reasons, the main memory always reads a word (4 bytes), and so, the desired size is computed in the CPU.

[4]This is determined by the direction flag, which determines whether the CPU reads memory from top to bottom or in reverse.

4.5 Jumping and Branching

To be turing complete, the processor needs to be able to do conditional jumps to other memory locations. This is done with the jumping instructions: jump (`j`), jump-register (`jr`) and jump-and-link (`jal`). The conditional jumps are: branch-equal (`beq`) and branch-not-equal (`bne`).
On the bare-metal level of the processor, these instructions simply modify the value of the Program Counter register, which is otherwise inaccessible from assembly.

4.6 Interrupts

Interrupts are a special way to control what the CPU. It actively "interrupts" the CPU from its current job, and makes it execute a special procedure, specified by an interrupt number. There are usually 3 types of interrupts [20]:

- Exceptions
  Exceptions occur in software, usually when an error has occurred that needs attention from the kernel. This is usually caused by reading from illegal memory addresses or when arithmetic overflow occurs.

- Hardware Interrupt
  Hardware interrupts are initiated from hardware devices, such as a mouse or a keyboard. When a user presses a key or moves the mouse, the hardware devices send a signal to the CPU that something has happened that needs attention from the kernel.

- System Call (syscall)
  Syscalls are usually used by programs, when they need attention from the kernel. An operating system and the underlying kernel will usually expose an interface with a whole set of functions, that the program can access by syscalls. This can be everything from reporting termination of a program to writing data to the disk.

The action that the CPU has to perform is determined by an interrupt vector table. For each interrupt vector, there is specific

code to be executed. Because the interrupt vector table is limited in size, operating systems, such as Linux, use a single interrupt vector number 0x80. Additional arguments for further determination of the service are passed in service number, which is stored in the general purpose registers, and if needed, on the stack.
System call handling is made somewhat easier in MIPS. Whereas in x86_64, you have to set the appropriate system-calls arguments and then do an interrupt on the correct vector, MIPS has a dedicated system-call instruction `syscall`. The operating system can choose however the arguments are passed, but usually, the service number is stored in `$v0`, and the arguments in `$a0-$a3` [22].

4.7 Memory

Memory is an essential part of a working computer. The memory stores both the program code (the instructions) as well as the program variables, which cannot fit in all the CPU registers.
The memory is a large chuck of consecutive bytes. While these can store practically anything, operating systems designate memory areas into multiple segments (also called sections), which eases memory management. While many operating systems create many different segments in the memory map of a program, these segments usually fall into 4 categories [6] [22] [27]:

- `.text`
  This section consists of the executable contents of a program, which is the instructions. In modern processors and OS, this section cannot be written to.

- `.data`
  This segment contains the initialized variables used in the program. This section can be both written and read from, but is not executable.

- `.bss`
  Uninitialized memory segment. This allocates memory for variables, which are not necessarily initialized by the OS, and can contain random values.

- `stack`
  The stack usually starts at the top of the user memory area. It is a Last-In First-Out (LIFO) structure, meaning that elements added last are the first ones being removed, hence the name. The top of the stack is usually pointed to by a designated register in the CPU, in MIPS32, that is the `$SP` register. The stack is used for local variables within functions, which usually have a very short lifespan.

- `heap`
  The heap starts immediately after the initialized segments (`.text` and `.data`), and is used for variables with a longer lifespan. This memory area grows from bottom up, that is, starting at a lower address, moving to higher addresses. This memory area is maintained by the program, and can store practically any variables and other data-structures. This area can also be extended by the operating system, which makes it ideal area for storing large amounts of data.

While the memory layout of a process is completely implementation specific on the underlying operating-system, on most 32-bit UNIX systems, the memory area is very similar to what can be seen on figure 2.

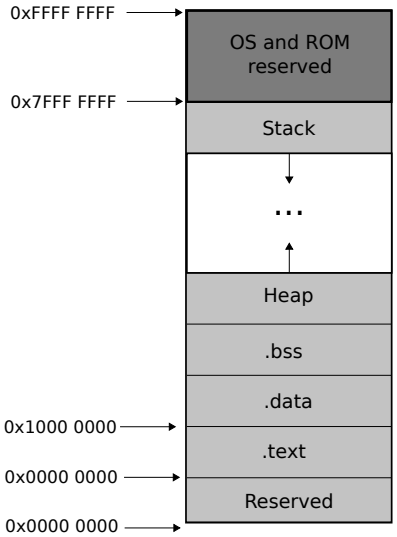

Figure 2: Process memory map.

Although the address space is 32 bit, the top half is not available for the user, and it

is usually used by the operating system [29].

5 Privilege levels

To tighten the security, prevent software faults, and to protect the user from malware, most modern hardware support mechanisms to detect the malicious behaviour and act accordingly.
Most computer operating systems have access to different resources, ranging from widely available devices to critical subsystems. To prevent malicious program or defective code to take over the machine, it is desireable to keep track of the execution, limiting the access to certain systems.
While limited checks can be done by the operating system, it is much more secure and efficient to let the hardware, in our case, the CPU, to keep track of each action, faulting on any suspicious activity.
This protection mechanism is often called Protection Rings, where the privilege of a program is determined by the position on a privilege ring — the inner ring 0 often called "kernel" for being the most privileged, to the outtermost ring, often used by user applications [27]. Depending on the architecture, the protection ring can have multiple layers.

5.1 Kernel and User mode

MIPS supports 2 privilege levels[5], called the kernel and user mode, distinguished by a bit in the status register. While some competing processors support more levels, such as x86 supporting 4 [11] levels, they are very rarely used in practice.
The usage of the two privilege modes is very intuitive from the perspective of the OS — kernel mode is used by the operating system and its kernel, while user mode is used for user applications only.
There are multiple functionalities these modes control, ranging from memory access to instruction access.

[5]All MIPS processors after R4000 model have a third *supervisor* mode. This mode is mostly ignored by operating systems, and will not be discussed or used here. [27]

5.1.1 Memory and IO access

In user mode, a MIPS32 processor can only access the bottom half of the memory space, that is, the first 2GB. For addresses over, the processor will check the top bit of the address and the privilege level, restricting non-kernel mode. However, if the processor has a memory-management unit (discussed in section 8), the operating itself can control the process access to the memory, preventing any reads and writes to restricted areas.

5.1.2 Kernel Instructions

Some instructions are restricted to the kernel mode only, because they can have system-wide implications. These instructions are called privileged instruction, and consist mainly of instructions that move data to and from co-processor 0, such as `mfc0` and `mtc0`.

5.2 Implementation

In the simulator, there is no separate need for a module checking the privileges of the executing process. The privilege checks are done every time the processor executes an operation with special need, by inspecting the 3rd and 4th bit of the status register, called the KSU[6].
If the process does not satisfy the privilege check, an appropriate exception is returned from the originating function. This exception is then handled in the WB stage, described in section 7.
It is to be noted that when handling exceptions, if the EXL flag is on, the processor is automatically assumed to be in kernel-mode [27].

6 Pipeline

CPU speeds are usually measured by timing the execution time of programs. Since a computer program is just a collection instructions, the speed of the CPU is determined by how fast it can process each in-

[6]In COD5, the lower bit of this field is called `UM` for User-Mode [22]

struction. Every CPU has a clock, which ticks at a given rate. For every tick, a new instruction is executed. This clock ensures that all instructions "flow" through the processor without problems, and that the electrical components, such as the ALU or the control-unit, can manage to carry out their tasks in that time. Naturally, electrical engineers have pushed the limits of the circuits to manage the highest clock rate. The clock rate of the very first processors was measured in hertz and kiloHertz (kHz), but most modern desktop CPUs reach in multiple GigaHertz (GHz) [32]. However, even with those speeds, the demand for faster processing units is ever-growing, and other techniques to speed-up the execution are used.
One of those techniques is pipelining, which separates the circuit into multiple stages, much like the assembly lines in factories. In such factories, workers have their own station at the assembly line, do a specific task repeatedly, and forward it down the line. This greatly increases the throughput of a factory and decreases the labor need. In MIPS, this idea is implemented by separating the processor into 5 stages [22]:

- Instruction Fetch (IF)
  Fetches the next instruction.
- Instruction Decode (ID)
  Reads the instruction, sets the appropriate control flags, reads the relevant registers and sends the data to the next stage.
- Execute (EX)
  Executes the instruction. This is typically done by the ALU with the appropriate operation supplied.
- Memory Access (M/MEM)
  All operations on memory happen here. This stage either loads a memory address or stores a value at an appropriate address.
- Write Back (WB)
  Writes the results to the CPU registers.

Each of these stages will naturally use less time than all of them combined, and since the clock is shared in all stages, it is set to the slowest stage in the pipeline.
Not only do we have faster tick rate on our clock, but we are also able to perform multiple operations concurrently. Figures 3 and 4 show the timing of each instruction, and how pipelining might improve the whole process.

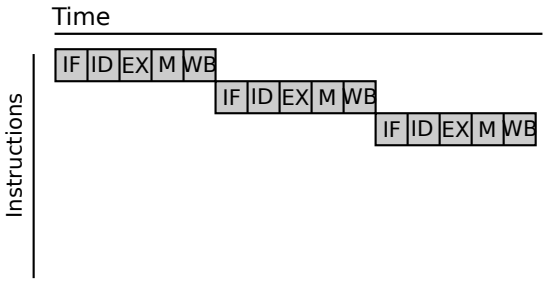

Figure 3: Single cycle implementation.

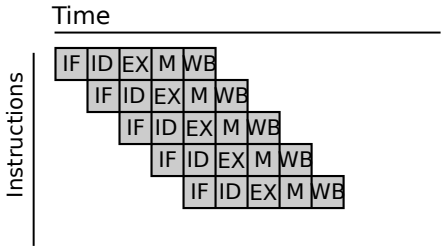

Figure 4: Pipelined approach.

6.1 Design of the MIPS32 Pipeline

The advantages of a pipelined design does not come without a price. Although the single-cycle implementation of the processor is very similar to the pipelined approach, it has its fair sets of challenges. The main problem with executing instructions concurrently is that the instructions will often rely on the result of the previous instructions. These situations are referred to as hazards.

6.1.1 Data Hazard

Data hazards mainly occur when an instruction cannot continue, because it must wait for the result from an earlier instruction. Suppose a program wants to calculate the sum of 4 integers:

$$A = A + B + C + D$$

In MIPS32 assembly, this would be written as:

```
# t0 = A, t1 = B, t2 = C, t3 = D
add $s0, $t0, $t1   # s0 = A + B
add $s1, $t2, $t3   # s1 = C + D
add $v0, $s0, $s1   # $v0 = $s1 +
    $s0
```

Figure 5: Code exposing data hazard situation.

Here, the first two instruction will have no trouble executing, as they do not share any source or destination registers. The third instruction however, will not be able to fetch the updated values. When it is in the ID stage, where it decodes the register `s0` and `s1` values, the previous instructions are still in the pipeline, in the EX and MEM stage! These instructions have not written back their results in the appropriate registers, and so, instruction 3 cannot fetch the correct value of `s0` and `s1`, unless it waits 3 clock cycles. This situation can be visualized in a sequential graph show in figure 5, where it can be seen that the result of the first two instructions is needed in the third.

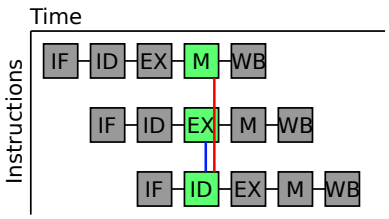

Figure 6: Time-sheet of the program listed in figure 5.

The problem is very clear - the data is needed before it is saved at the appropriate place. However, the data is usually already available in the EX stage in the ALU, and can therefore be used in the same clock-cycle in the ID stage. This is called forwarding or bypassing, and is handled by a dedicated Forwarding Unit. This unit is wired to both EX, MEM and WB stages, and has a logic unit that checks for matching registers in the instruction, in which case it forwards the correct result back in the pipeline.

6.1.2 Control Hazard

Another type of hazard is the Control Hazard, which occurs when the processor must make a decision based on an instruction, while others are executing. Whether a branch is taken or not can be determined by the ALU in the EX stage, which does a equality check on the two values. This does not add much additional logic to the circuit, as the forwarding unit already bypasses all the most recent results as the operands to the ALU. If a branch is taken, a challenge emerges that we already have the next instructions in both ID and IF stage, even though our program is inteded to branch away. There are many possible ways to solve this issue, the most simple, but inefficient one, is to stall the pipeline by inserting two No-Operation instructions after the branch. This way, the instruction after the branch will do nothing, but in both situations, we loose a 2 clock cycles. Although this does not sound like much, considering that branching instruction make up about 25% of any assembly code [14], this can be a lot.
A way to ease the problem with this many wasted clock cycles is to move the branching calculation in the ID stage. A simple comparison unit for the register values is required, but the forwarding unit also needs to be be modified to forward results to the ID stage.

A better, but much more complex technique to solve this issue, is to have a dedicated "Prediction" unit in the processor, which will predict whether a branch will be taken or not, and fetch instructions to the IF stage accordingly.

6.2 Implementation

The implementation of the pipelined simulator will group the devices in the machine together, so that each device or system has its own C structure.

6.2.1 Simulator structures

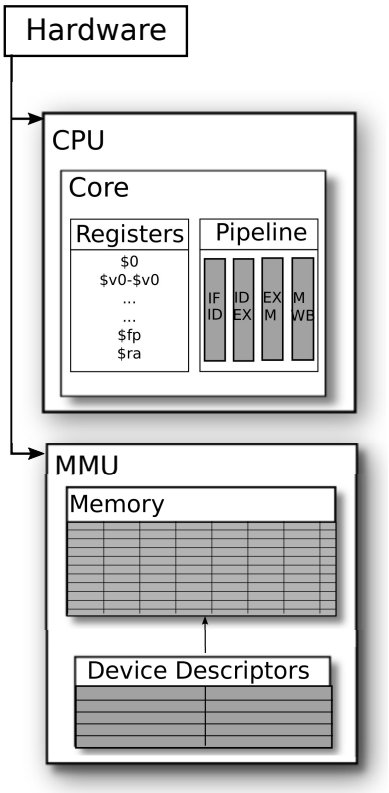

Figure 7: Relations between the simulator structures.

In the simulator, the processor cores, pipeline registers, and memory unit all belong to a "hardware" structure, that binds them together, making up the state of the simulated machine. The relations between the components can be seen in figure 7.
In the simulator, the hardware structure will be used to pass along the simulated hardware to functions, which then can access the underlying devices, modifying them properly.

6.2.2 Pipeline functions

The pipeline is implemented by running each pipeline stage separately. For each clock-tick, the pipeline stage functions are being called in reverse. This can seem very unnatural, but this prevents pipeline registers to be overwritten before they are used. This also prevents an instruction to get throught the pipeline in one tick of the clock, and simplifies the assignment of the pipeline registers.

```
void tick() {
        interpret_wb(...);
        interpret_mem(...);
        interpret_ex(...);
        interpret_id(...);
        interpret_if(...);
}
```

Each of the functions receive the relevant pointer to the CPU core structure, which holds all the relevant information about the state of the core.

6.2.3 Forwarding Unit

The pipeline stages work without regard for data hazards, which is where data is required for an instruction before it is stored by the previous instruction. Therefore, we add a forwarding unit function `forwarding_unit()`, which is run by the end of each clock cycle. Being the last function, it has all the recent information in all the pipeline stages, which enables it to decide which values need to be forwarded.
The forwarding unit only forwards values to the ALU input mutual exclusion units MUX A and MUX B. It does that by first checking if the previous instructions write to a register, that is, modify a register value. If it indeed is the case, the destination registers are compared, and if the match, the value from the previous instruction is bypassed. This is done for both the A and B input to the ALU in both Memory- and Writeback-stages. The code for forwarding values from stage MEM to EX can be seen in figure 8. Forwarding from the WB stage is almost identical.

```
/* Forward to A MUX */
if(EX_MEM.c_reg_write == 1
   && EX_MEM.reg_dst != 0
   && EX_MEM.reg_dst == ID_EX.rs)
       ID_EX.rs_value = EX_MEM.
           alu_res;

/* Forward to B MUX */
if(EX_MEM.c_reg_write == 1
   && EX_MEM.reg_dst != 0
   && EX_MEM.reg_dst == ID_EX.rt)
       ID_EX.rt_value = EX_MEM.
           alu_res;
```

Figure 8: ALU result being forwarded from the MEM stage to EX.

6.2.4 Hazard detection unit

Control hazards arise when the processor is branching to another execution path based on the result of the previous instructions. As there is no "correct" solution to the branching hazard, the MIPS CPU manufacturers often implement their own branch predictions based on the intended use of the processor. For simplicity in our simulator, we will always predict that the branch is *not* taken. We will also simulate that the branching is decided in the ID stage[7], so that only we only have 1 branch delay slot. In that case, we always run the next instruction after the branch. This works well, because this is a widely accepted approach in many RISC architectures [31]. Knowing that the instruction following a branch is always executed, many standard compilers and assemblers restructure the code so that branch-delay slot is not wasted.
For example, where assignment of a value to a register and a branch would consume 3 instructions, utilization of the branch-delay slot can bring it back to 2. The difference can be seen on figure 9. Note that the order of the instruction does not matter in that case.

[7]Technically, we are deciding this in the `interp_ex` function, but this will have the same effect due to the reverse execution of our pipeline stages.

```
addi $v0, $0, 10
beq $0, $0, exit
nop
```

```
beq $0, $0, exit
addi $v0, $0, 10
# ...
```

Figure 9: Wasted branch delay slot (left) and utilized branch delay slot (right).

7 Exceptions and Interrupts

As mentioned in section 4.6, exceptions and interrupts are a way to signal the CPU of an event, that needs immediate attention. In MIPS, everything that defers the usual execution path of a program to attend to another event, such as interrupts, traps, syscall, are called exceptions [27]. Exceptions are all handled the same way in MIPS.

7.1 Precise Exceptions

Due to the pipelined nature of MIPS (and other RISC architectures), exceptions can occur anywhere in the pipeline. That is, when an exception is raised, there are probably uncommitted instructions[8] in the pipeline. For simplicity of both hardware and software, it is desireable to handle an exception exactly when it occurs. In MIPS, precise exceptions means that all instructions preceding the exception victim[9], are committed, while all instructions after are not.
Precise exceptions give the programmer the usual, sequential view of the program. However, precise exceptions in MIPS can be very expensive, potentially clearing the whole pipeline of otherwise faultless instructions.

7.2 Handling Exceptions

Depending on the type of the exception, a different exception handler must be used. Many other architectures, such as the x84 and later x86_64 use vectored interrupts. In vectored interrupts, a table of

[8]Instructions, whose result is not yet written back to the registers.
[9]The instruction that caused the exception.

addresses are kept in the Interrupt Vector Table (IVT), specifying start-addresses of different interrupt handlers. When an exception is raised, the type of exception is determined by some microcode or other hardware, and an appropriate handler is chosen from the IVT[10] [19] [27].
Although this approach is arguably faster, it is not used efficiently in practice. In many modern operating systems, all interrupt handlers have a lot of code in common. To avoid redundant code and large interrupt handlers, the handlers often jump to a shared location in which the redundant code is located, for example, to save the current registers on the stack. First, by jumping to a common location, the purpose of the IVT is (almost) defeated, as the time saved is being spent by jumping to another location anyhow. Secondly, the information about the interrupt retreived from pure hardware or microcode is very limited without building very complex hardware. A large OS would have to analyse the exception further.

Although MIPS32 has a type of an interrupt vector table [27], it is very rarely used due to its flaws mentioned above, and the fact that each interrupt handler is only 32 instructions long - something no real exception handler can fit into [27].
Due to the speeds of modern processors, MIPS32 exception handling routines always start at the same address, when an exception arises.
This exception handler is by default located in the unmapped and uncached memory segment KSEG1, address `0x80000080`, although this can be changed by modifying the CP0 `EBase` register, while also setting the status register bit `BEV` [27].

7.3 Interrupts

Hardware caused exceptions are often called "interrupts" in the MIPS world. While these interrupts are handled just as any other type of exception, they are not triggered in the code by the means of a syscall instruction. Since interrupts happen from outside the CPU, a system has to be implemented to notify the processor of the external event.
Like in many other processor architectures, two pins are added to the MIPS CPU, which are used to signal of a hardware interrupt occurance [18].
The first pin is called the Interrupt Request (IRQ), which indicates a pending interrupt. The second pin is the Interrupt Acknowledge (IACK), and is controlled by the processor to indicate that an interrupt request is acknowledged and is being serviced. The IACK pin is needed, since the CPU might be busy servicing another exception, not being able to attend the hardware interrupt immediatelly. Thus, the IRQ pin can be enabled over multiple clock-cycles before it is acknowledged [18]. While it may sound that having a hardware device wait several clock-cycles results in delayed output, it should be noted how fast MIPS CPUs (and many others) are, and how many instructions they are able to carry out before the external device even notices.
It should be noted that this is another point where MIPS32 deviates from its more mainstream x86 counterpart. While the MIPS processor knows that an hardware interrupt has occured, it does not know which device raised it. To detect the device that raised the interrupt, it must scan all the device IO registers, discussed in section 9. This approach is called "Polled Interrupts".

7.3.1 Polled Interrupts and Vectored Interrupts

In polled interrupts, the processor does not know the hardware source of the interrupt, and must therefore check the status IO register of each device in order to determine if the device needs to be serviced [18]. This method is used in MIPS, for simplicity of the hardware, and because exceptions are handled by generic exception handler code as well.
Another method of handling interrupts is using vectored interrupts, used by the x86 architecture. In vectored interrupts,

[10] A x86 assembly programmer might note that `int 0x80` in Linux calls the interrupt handler, located at index 128 in the IVT.

much like vectored exceptions, the processor jumps to a certain interrupt handling code, based on the interrupt type. However, since the event is external to the processor (caused by hardware), additional wires must be attached to the CPU, informing it of the IRQ number, which is unique for each type of device [20]. This requires a lot of added hardware complexity in the form of an interrupt controller, such as the well known Intel 8259 IRQ controller, which can queue up multiple interrupts, even prioritizing them according to the need of the CPU [20].

7.3.2 Exception procedure

Before exception handler code is executed, the processor must perform certain actions, required to correctly recover from the exception. These steps are [27]:

1. Setup the `EPC` register, which hold the address of the victim instruction. This is used when returning from the exception, so that the execution can continue from the correct position. `EPC` is very similar to the general-purpose `$RA`, which holds the return-address for user programs.

2. Flip on the `EXL` bit in the status register. This bit forces the CPU into kernel-mode, giving it higher privileges.

3. Set the `Cause` register, which holds all the information about the exception.

4. Jump to `0x80000080`, where the exception handler code starts.

When the exception handler code is finished, the `eret` instruction should be used. Similar to the `ret` instruction, `eret` fetches the return address from `EPC` register, clears the `EXL` bit in the status register, and jumps to the return address.

7.4 Implementation

In the simulator, an enum structure `exception_t` has been implemented, which holds all the necessary excetion types (none-exception included). All functions in the simulator, that can cause an exception, such as reading from the memory, now return an exception to signal the simulator of the event. In the simulator, this exception is stored in the corresponding pipeline-register, forwarded to the subsequent stages. When an instruction with an exception code other than "exception none" reaches the Write-Back stage, the exception handling mechanism is called.
Note that the exceptions are not necessarily handled in the order they are raised, but rather, the order of the instructions. For example, in the following assembly code, a memory exception might occur in MEM at the same time as the arithmetic overflow exception in EX:

```
...
lw $t0, 10($t1)
add $t3, $t2, 100
...
```

However, due to the way exceptions are forwarded in the pipeline, the memory exception reaches WB stage first, and gets handled first.

7.4.1 Interrupts

As mentioned in the interrupt section 7.3, interrupts need another way to signal the processor of an exception that needs attention. In the simulator, for every clock-cycle, we iterate over every external device that can trigger an interrupt. In the loop, we check for its interrupt status, and if needed, handle the exception.
In our simulator, the interrupt request is checked every clock-cycle. However, in some simulators, such as SPIM, a short artificial delay is added to enhance the realism of the simulator [13].

7.4.2 Preparing for exception handling

When an instruction raises an exception, `handle_exception(...)` is called. This function follows the exception entrance procedure very closely, described in section

7.3.2. First, it clears all previous pipeline stages, as they now are discarded. The WB stage is kept, as we need to extract some information about the state. Then, the cause is determined by analyzing the exception, and stored in the `CAUSE` register. Privileged mode (kernel mode) is entered and PC is set to the exception- handler address to force the jump. In the end, stage WB is also cleared to ensure no faulty results are written back. It is to be noted that this function is *not* pipelined in the simulator, giving the impression that all this happens in only one clock cycle.
Given that this procedure is executed regardless the type of the exception, one might argue that this might be implemented in microcode or hardware, really only using 1 clock-cycle.

8 MMU

Most processors strive to help the OS and its developers to ensure optimal utilization of the main memory, as well as the overall security of the userland memory usage.
In most systems, it is a common practice to split the memory into multiple segments, where each part can only be accessed by the intended user. For example, a user program should never be able to modify or even read memory of the kernel. If that was possible, a single program would be able to break the running OS by overwriting some essential values, or even retrieve values from variables in other programs (e.g. passwords).
These mappings abstract the program addresses from the physical addresses, and translation has to be implemented, to map these two address spaces. This is usually done by the Memory Management Unit (MMU).
One of the big differences of the MIPS architecture is that many components in the processor are completelly optional, and up for the manufacturer to decide whether they suit the needs of the desired application, be it embedded device or a supercomputer [26].
The memory management in MIPS32 is no exception, as MMU is completelly optional. However, on MIPS32, to still ensure some basic form of user-restricted memory areas, the memory is split into 4 segments, with each a designated usage: [24] [27]

- *kseg0*, `0x80000000-0x9fffffff`, `512MiB`
  The first kernel segment is which is translated by stripping of the first bit in the address field. This segment is using caching, and can therefore first be used when caches have been initiated [27].
- *kseg1*, `0xa0000000-0xbfffffff`, `512MiB`
  Kernel segment which is *not* cached. It is the only segment that is guaranteed to be available immediatelly after a system reset (or boot), where no other CPU devices are initialized. This is the segment where bootup-code is stored.
- *kuseg*, `0x00000000-0x7fffffff`, `2GiB`
  This is the only segment that can be used by user programs. It is mapped and cached, so the OS needs to initialize caches and possible MMU for this segment to be useable.
- *kseg2*, `0xc0000000-0xffffffff`, `1GiB`
  This segment is for additional access modes, such as the *kernel* mode. This segment is mapped and cached, and thus, cannot be used immediatelly after bootup.

8.1 Memory access privileges

When the CPU starts up, it is by default in kernel mode. In this mode, it has the privilege to access all addresses (along with many other things). The kernel will retain this privilege until it starts user programs, in which case, the OS flips the `KSU` bit in the co-processor 0 status register [30]. Privilege levels are described in further detail in section 5, page 8.

8.2 Implementation

The simulator must support multiple memory sizes, for better simulation of

MIPS programs, as well as being more portable on host machines with limited memory.
Since the physical addresses do not match the address of the data on the host machine, an additional layer of memory mapping is created, which will ensure the segment sizes still match. We will call these addresses "host" addresses, representing the actual address on the host machine.
A command-line flag "`-m`" followed by an integer can be supplied to the simulator, letting the user control how much actual memory is allocated for the simulator memory. By default, this is 512MiB.
When the MMU module is initialized, the 4 segments are scaled accordingly – `kuseg` is half of the allocated memory, `kseg2` makes up a quarter of the memory and `kseg0` and `kseg1` share the rest.
Now that the addresses are allocated, the simulator needs a way to translate between the layers of mappings.
All addresses an ordinary MIPS program uses, are virtual, that is, translated by the processor. For translation of a virtual address, the simulator uses `translate_vaddr(uint32_t vaddr)`,
which translates an address to the physical location on the simulated memory, or an IO device, discussed in section 9. However, since we are using a simulator, we must translate the address further, into an "actual" address. For this, `translate_paddr(uint32_t paddr, ...)` is used, which translates a physical address into an actual address — the address of the data on the host machine. A figure of an example mapping can be seen on figure 10 on page 17.

9 Memory Mapped IO

There are multiple ways a computer can communicate with its peripheral devices. The most used techniques are Memory-Mapped I/O (MMIO) and Port-Mapped I/O (PMIO). Port-Mapped I/O is an older method of accessing peripherals, and is much more common in consumer computers with Intel or AMD CPUs [11].
PMIO requires the processor to have additional pins and special instructions, just for communication with I/O devices. I/O devices have a specified port number, which the processor uses to identify and communicate with the device, much like an address. For example, Intel x86 processors might be polling a keyboard using the special `in` instruction, specifying the predefined keyboard status ID `0x60`, then reading from keyboard data, with ID `0x64` [12] [21]:

```
_wait:
  in   al, 64H ; Read keyboard status
  and  al, 1   ; Check if ready
  jz   _wait
  in   al, 60H ; Read keyboard data
```

Besides the added circuitry in the processor, port-mapped I/O are fairly simple to implement for chip manufacturers, and easy to use for programmers. They have their own special instructions, and they do not share memory space, which prevents confusion about memory segmenting. However, PMIO approach is very limited to only `in` and `out` instructions, reading only a few bytes at a time into the EAX register, as well as a limited number of ports for devices. [11]

Memory-Mapped I/O is commonly used by RISC architectures. In MMIO, communication with the peripherals happens over the same address bus as the memory, so that memory and I/O communication share address space. This means that when a processor, that uses MMIO, wants to access a peripheral, it uses a memory address for communication.
The advantage of memory-mapped I/O is that the method uses already existing memory bus lanes for communication, unlike the port implementation. However, since the memory is shared with peripherals, and these addresses are not maintained by the processor, additional logic has to be created in the memory-management unit (discussed in the previous section).
Due to these reasons, memory-mapped I/O is becoming increasingly popular, even

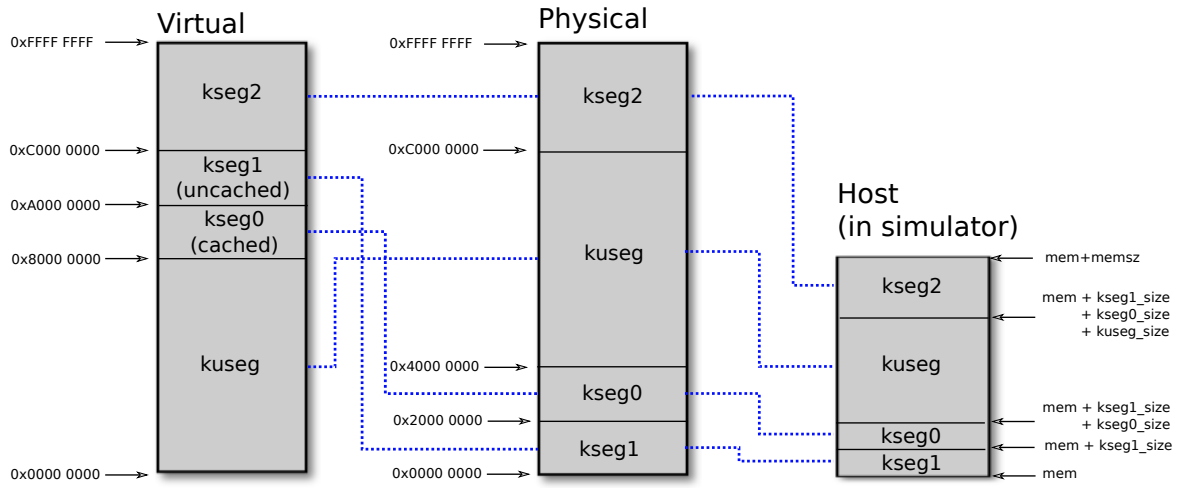

Figure 10: Main memory mapping in the simulator.

on modern x86 hardware [21].
MIPS initially sees Input/Output (IO) devices as a set of special-purpose registers. These special registers are the processors only way of communicating with a given device.
For every device, there 3 types of registers [1] [26]:

- `Status registers`
  Provide information about the underlying device. These registers are read-only for the CPU.

- `Control registers`
  Used to communicate and control the device. These registers are writeable by the CPU, but may not always be readable.

- `Data registers`
  Used for the actual data-transport. For example, the latest key pressed on the keyboard might be stored in a data register.

On MIPS32, these registers have allocated 4 memory bytes each, and are always located sequentially in the memory.

For example, figure 11 shows an example of IO register layout of a very simple keyboard device.

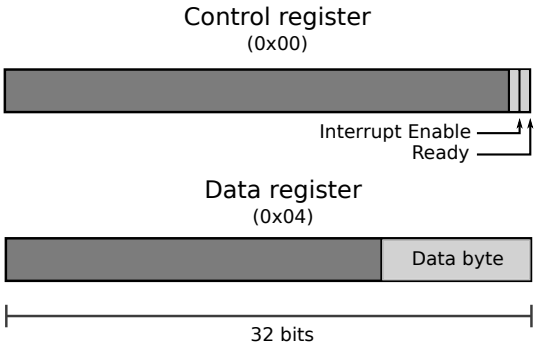

Figure 11: Set of memory mapped I/O registers [26]

When a device is detected, the memory-management unit will allocate an address for the device, notifying both the CPU and the device of these newly allocated registers. From now on, it is up to the driver to interface with the devices.
Depending on the type of the IO device, the device can be represented from a few registers to dozens. For example, a mouse or a keyboard only transmits few bytes of information at a time, needing only a few registers, while graphic adaptors or disk drives might need more [1].
These special-purpose registers are located in memory, mapped to a certain segment. A device controller maintains a list of these registers, and maps new devices to memory. [2]
On MIPS32, the highest 64 kilobytes (`0xffff0000 - 0xffffffff`) in the available memory are used for mapping these special registers [1]. The memory mapping

of the IO area must not be cached, as a cache assumes the memory is only modified from the CPU, and not by any external device, such as an IO device.
However, it is obvious that IO registers can be very inconvenient for larger and fast data-transfer, due to the very limited register size. To dodge this bottleneck, memory controllers are implemented to give IO devices direct access to larger parts of the main memory, without the need of going through the CPU first.

9.1 Direct Memory Access

Direct Memory Access controllers, also called DMA controllers, allow certain hardware to access the main memory independently of the CPU. In MIPS, this feature is built-in in the MMU.
Larger transfers between hardware and memory are usually initiated through the device registers, where the CPU sets up the MMU and communicates the target address. Then, the IO device is free to read and write in the target address. When the peripheral is finished with its memory actions, it will usually notify the CPU using an interrupt, having the processor taking care of the rest.
Direct Memory Access is often utilized by peripherals with heavy IO use, such as the storage drives, as well as network-, sound-, and graphics-cards.
Because KUDOS does not read large portions of data from its IO devices, nor is there any DMA support in the current YAMS simulator, DMA will not be implemented.

9.2 Detecting and mapping peripherals

When a computer is turned on, or a new peripheral device is plugged in, the MIPS processor and its MMU automatically map the device to a given address, allocating the necessary IO registers. All this happens seamlessly and without the interaction of the operating system. This is problematic, because the OS needs to know the type of the device and the address of its memory mapped registers, amongst other things. There is no real way to communicate this between the processor and the operating system. To make matters worse, each processor has its own addresses for the registers, and a slightly different configuration. On top of that, each board has its own set of external components, busses and metadevices [4] [33]. Instead of compiling a very specialized kernel for seemingly every configuration of a System on a Chip (SoC), operating system and bootloader developers have designed a specification *devicetree*, which is used to describe system hardware [4].
The devicetree specification can describe a lot of aspects of a hardware devices, such as the interrupt number, size of the IO registers, even the device clock speeds.
The Linux kernel also enjoys this devicetree specification, both for ARM and MIPS architecuters [28, `linux-4.6/arch/mips/boot/dts`], reducing the time and complexity of implementing a new board or a SoC.
In KUDOS, this data structure is unfortunatelly not used, because the hardware it runs on is very limited, consisting almost only of simulated devices. As a result of that, KUDOS looks at a predefined address for data structures describing the devices, called the device descriptors.

9.3 Device descriptors

To detect and communicate with external devices, device descriptors are used. A device descriptor is a data structure which, just like a devicetree, specifies all the relevant information about the device, such as its type, name, register addresses and so on. The format of the device descriptor is defined in Yet Another MIPS Simulator, the current simulator used to run KUDOS. The same device descriptor structure is defined in KUDOS, so that KUDOS can interface with the underlying simulator, retrieving the relevant information. The structure is defined in [17, kudos/drivers/mips32/arch.h] as:

```
typedef struct {
    uint32_t type;
```

```
    uint32_t io_area_base;
    uint32_t io_area_len;
    uint32_t irq;
    char vendor_string[8];

   /* Reserved area (unused). */
    uint32_t resv[2];
} io_descriptor_t;
```

Learning from the devicetree specification and the seasoned Linux developers, it would probably be better with a more general method to describe system hardware. However, to be compatible with KUDOS and not break it, we will use this exact same interface with devices, although modifying its location in memory slightly.

9.4 Implementation

The OS, or rather, KUDOS in our case, needs to know the exact memory address of the device descriptors, so that it can interact with the underlying IO devices. KUDOS and the simulator agree on a memory address, which is the start of the device descriptor area. In KUDOS and the simulator, this area is defined by `IO_DESCRIPTOR_AREA`, with the value of `0xFFFE0000`. From there, KUDOS searches for the underlying devices. The structure is represented on figure 12

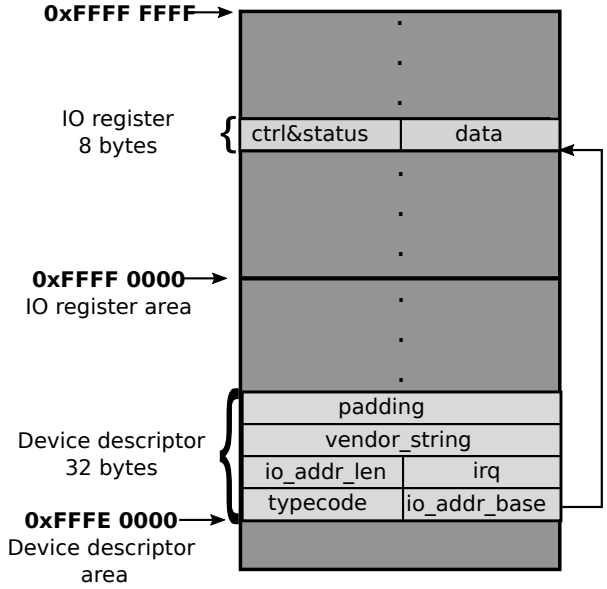

Figure 12: Memory layout of IO device descriptors and registers.

9.5 Hardware Devices

Naturally, we are going to need to implement some essential hardware devices for KUDOS to use.
There are no definite specifications for how a hardware device must work. Effectivelly, each manufacturer decides the specification of their devices themselves. It is impossible for an operating system to keep track of all these devices, and so, a special kind of interface, called a `driver`, is implemented and used for each device.
Most drivers are specific to certain devices, removing the subsequent complexity for the communication with the device. This is also why certain devices refuse to work in Linux, Windows or other OS, before the correct driver is installed [16].
Lucky for us, we can define our own devices and its drivers – that is, we do not emulate an existing device, rather we define our own, with its own set of magic numbers, IO registers etc.

9.5.1 Shutdown device

The first most important device is the shutdown-device, which powers off the machine (in our case, the simulator). In KUDOS and YAMS, this device is implemented with only 1 data register, which will trigger a shutdown, if a magic number `POWEROFF_SHUTDOWN_MAGIC` is written to it [17, drivers/metadev.h].
In the simulator, if the write-address is that of the shutdown device, and the magic number is known, the global boolean variable `g_finished` is toggled, stopping the simulator.
Once again, to avoid unneccesarry compatibility breaks with YAMS, we will use the same magic number.

9.5.2 TTY device

To avoid a separated keyboard and display, a TTY device is implement, which takes care of both. Once again, it will use the same number of IO registers and the same typecode, so that the KUDOS TTY drivers do not have to be re-written.
For usability, the TTY device needs to

be controlled from a separate terminal emulator, as the simulator itself prints a lot of logging, debugging, and error messages, which would choke the actual OS input and output. For simplicity of the simulator, this TTY window will be implemented in a separate program, which will wait for the simulator to start, and then connect to it.

9.5.3 External program communication

Each external program, that acts as an IO device, needs a way to communicate with our simulator. In Linux systems, there are multiple ways for separate programs to communicate, such as using files, sockets, signals, message passing as well as named pipes and shared memory [15].
For simplicity, the simulator and the programs will use shared memory model, in which the OS will create and share a section of the process' memory, identified with a key.
In the simulator, the IO register memory area will be split up into chunks of shared memory, each identified with a unique key. Every IO device, that the simulator supports, will have this key, and use it to access the registers at the location.
Since MIPS32 does not implement interrupt numbers, we need not need any elaborate way of signalling the counterpart program of an event – we simply need a binary signal, just as the one implemented in the bare-metal of the processor. Thus, for signalling an interrupt, the simulator uses a POSIX user-defined signal `SIGUSR1`, which will be our indicator of an event.
This way, the simulator and the device can work independently of each other.

10 Tests

All test programs lie in the `tests/` subdirectory, with a corresponding file ending with `".text"`, containing the expected result. To run all tests, a bash script `run_tests.sh` is stored on the project root, which runs each test program with the simulator, comparing the return value with the expected return value.
It is essential to repeatedly perform tests on the simulator, to ensure its correct behaviour, and in cases, detect and fix errors and other bugs.
The very first tests ensure the basic functionality of the simulator, usually testing each instruction for the correct behaviour. When the simulator gets more advanced, additional features are added, that also need to be tested.

10.1 Basic tests

Tests for the correct behaviour of an instruction is very simple. Each test uses the related instruction in as many ways as possible, using as few other instructions as possible. For example, the `add` instruction is tested for both being able to add, but also subtract in one test.
For simplicity, the tests return their value in `$v1`, which the simulator uses as its exit value.

10.2 Advanced tests

Executing instructions is only a part of the simulation. The simulator also needs to be able to handle certain situations and problems correctly, and according to the specification.
One of these advanced tests is the pipeline testing, which are programs with instructions structured in a way so that the pipeline forwarding unit and hazard unit are used repeatedly. If one of these units fail, the result will be seen immediately on the output value.
Other more advanced tests ensure many side-effects of instructions, such as the branch-delay slot, or correct flag-toggling of the status register on exception occurrence.

10.3 Travis-CI

To make development easier, Travis-CI is used. Travis-CI is a service used to automatically build and test software projects

[3]. For each new change in the simulator project, Travis-CI builds and runs the test suite, reporting back on any errors. This way, it is easy to find working builds, as well as finding sources of bugs in the code.

10.4 Partial conclusion

Having a suite of testing programs for each instruction was a tremendous help to find and fix bugs.
However, a big issue with the test suite is that some of the tests depend on other instructions to work, as to setup the test. This is not necessarily always the case, and can cause false-negatives, or worse, false-positives. Optimally, the simulator should be able to start in a certain state, defined by some configuration file, describing the values and flags of all the processor registers, flags, even pipeline registers. This would help to test certain cases in the simulator, avoiding possible side-effects of other possibly faulty instructions.
Another small, but useful improvement would be having dynamic tests, that would generate test values on every run, so that each time a test is run, it uses new values, covering a bigger area of the instruction, and further increasing the chance of finding a bug in the simulator.

11 Performance

Although the primary objective of the simulator is not the performance, it is an interesting aspect to study.
The pipeline approach of the simulator takes a lot of processing to perform, and is implemented only for academic purposes – to study the state and behaviour of the processor. It goes without saying that the pipeline is not needed to emulate the target CPU, as the emulated instructions are not held back by a time restricting clock, nor do they have to wait for result from the previous instruction to continue.

```
# Hardcoded addresses.
# tick() = 0x0000000000403ba1
set pagination off

set variable $i=0
set variable $count=0
while ($i < 1000)
while ($pc != 0x0000000000403ba1)
stepi
set variable $count=1+$count
end
stepi
set variable $i=1+$i
end

p $count
p $i
```

Figure 13: GDB script to test performance.

11.1 Comparison

The simulator is compared to its predecessor YAMS using GNU debugger (GDB). GNU debugger is a program used to debug, control and inspect other programs. It is a widely used and very powerful tool used for development on Linux [8].
In GDB, a breakpoint is set on every simulator clock tick, that is, every time the simulator reaches to the start cycle of simulating a new instruction. Then, GDB steps to next instructions, until the breakpoint is reached again. This process is repeated a number of times, the number of "steps" saved and printed at the end. A GDB script `perf.gdb` is used for this task, which consists of two `while` loops - one iterating over the number of simulated instructions, and the other one counting the host instructions used. At the end of the script, GDB prints `$i`, which is the number of simulated instructions executed, and `$count`, which is the number of host instructions used. The script can be seen on figure 13.

Due to compatibility, the simulator will executing KUDOS, while YAMS will be executing BUENOS. Although this might cause the simulators to execute different instructions, the test runs enough instructions to approximate their performance.

	i = 10	i = 1000	ratio
`YAMS`	3545	430328	429
`Simulator`	9218	788851	790

By this very primitive test, YAMS is approximatelly 2 times faster, as it uses the half the instructions to emulate an instruction.
On the testing computer, an Intel® Core™ i5-6200U Processor is used, with turbo frequency up to 2.80GHz [10]. If fully utilized, the simulator can on this machine simulate a very fast MIPS processor with over 3.5MHz clock rate, which is more than enough for the KUDOS operating system:

$$\frac{2.80GHz}{790} \approx 3.5443MHz$$

12 Running KUDOS

The simulators main target is to run KUDOS, and so, operating systems such as KUDOS can be run just as any other userspace program.
For initial testing purposes, KUDOS is modified to startup, initialize only the shutdown device, and trigger the shutdown device.
When running the simulator (git commit version `0ff2a1c8`) on the modified KUDOS, a crash occurs:

```
$ bin/mips-sim -l out.txt kudos-mips32
...
[ERROR] src/mem.c, translate_paddr():216:
ADDRESS OVERFLOW, PADDR: 0x40000004.
Out of bounds: 0x38000000
Segmentation fault (core dumped)
```

Inspecting the code, the crash occurs at `0x80011034`[11]. Since KUDOS correctly only uses the user segment (`kuseg`) for user-space programs, the modified version of KUDOS, which does not start any userspace programs, should *never* write to the middle of the user segment.
The faulting instruction address is located just at the start of `semaphore_P`, which are not used in the modified KUDOS. Indeed, the simulator has gone wild and continued execution, where it should clearly have been shutdown, until a crash occurs.
Upon further inspection of the code, a very peculiar behaviour is uncovered — the simulator enters a kind of "NOP" sled, that is, a long section of a program that only consists of instructions that do nothing. This occurs after instruction located at `0x80000084`:

```
0x8001891C: 0x1466FFFA  BNE  rs = ...
0x80000088: 0x00000000  SLL  rs = ...
0x8000008C: 0x00000000  SLL  rs = ...
0x80000090: 0x00000000  SLL  rs = ...
0x80000094: 0x00000000  SLL  rs = ...
0x80000098: 0x00000000  SLL  rs = ...
0x8000009C: 0x00000000  SLL  rs = ...
```

Another peculiar behaviour is exposed — the program executes in kernel segment (`> 0x80000000`), but it is not after the KUDOS start-point `0x80001000`. However, the instruction prior can be tracked down to be the comparison of the device typecode of the shutdown function:

```
8001891c: bne v1,a2,80018908 <shutdown+0x34>
```

It is clear that the execution can not continue at the "NOP-sled" addresses, and therefore, a bug is present in the address translation mechanism in the simulator, more specifically, the `translate_paddr(...)` function of the memory module.

13 Conclusion

The simulator has been written using modules resembling the actual design of MIPS32 processors. This yields a natural flow of instructions, as well as a very natural behaviour of the actual code.
While the simulator can successfully run all the test-cases supplied along side the code, we have shown that the simulator still contains bugs that, when running KUDOS, will cause the simulator to continue execution at unknown memory addresses, eventually reaching some actual code that will make the simulator enter an invalid state, and crashing. However, despite this bug, enough material has been presented to show that a pipelined MIPS32 simulator with a working MMU, memory-mapped

[11] The instructions are logged when they enter the pipeline, but the instruction causing the fault is in memory stage.

IO, exception mechanism as well as debugging functions, can be written, with excellent performance. After comparison to the more field-tested YAMS, the simulator was only about half as fast, with all the same features as its counterpart. However, due to the low performance requirements of the KUDOS system, the simulator should be more than fast enough.

References

[1] Jason W. Bacon. Mips assembly language programming. http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch14s03.html, 2010-2011. [Online; accessed 09-May-2016, (Archived by WebCite at http://www.webcitation.org/6hPM4fNga)].

[2] Robert Britton. Mips assembly language programming, 2002. Computer Science Department California State University, Chico Chico, California.

[3] Travis CI. https://github.com/travis-ci/travis-ci. [Online; accessed 6-June-2016, (Archived by WebCite at http://www.webcitation.org/6i6fuiYI3].

[4] devicetree.org. Devicetree specification, release 0.1-pre1-20160430, 30 April 2016. Available at: http://webdev.linaro.org/devicetree.org/specifications/ and https://github.com/devicetree-org/devicetree-specification-released.

[5] KUDOS Documentation. http://kudos.readthedocs.io/en/latest/. [Online; accessed 6-June-2016, (Archived by Wayback Machine at https://web.archive.org/web/20160607150655/http://kudos.readthedocs.io/en/latest/].

[6] Gustavo Duarte. Anatomy of a program in memory. http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/, 2009. [Online; accessed 9-April-2016, Archived by WebCite ®(at http://www.webcitation.org/6gf8eojPM].

[7] Michael Flynn. *Computer architecture : pipelined and parallel processor design*. Jones and Bartlett, Boston, MA, 1995.

[8] Free Software Foundation (FSF). Gdb: The gnu project debugger. https://www.gnu.org/software/gdb/. [Online; accessed 9-June-2016, (Archived by WebCite at http://www.webcitation.org/6iBPQJO5b].

[9] Johan De Gelas. Itanium - is there light at the end of the tunnel?, 2005. [Online; accessed 28-March-2016].

[10] Intel. Intel® core™ i5-6200u processor. http://ark.intel.com/products/88193/Intel-Core-i5-6200U-Processor-3M-Cache-up-to-2_80-GHz. [Online; accessed 9-June-2016, (Archived by WebCite at http://www.webcitation.org/6iBTSzcQD].

[11] intel. *Intel® 64 and IA-32 ArchitecturesSoftware Developer's Manual*. December 2015. Combined Volumes:1, 2A, 2B, 2C, 3A, 3B, 3C and 3D.

[12] intel. *ntel® 7 Series / C216 Chipset Family Platform Controller Hub (PCH)*. June 2012. Order Number: 326776-003.

[13] James R. Larus. Appendix: Assemblers, linkers,and the spim simulator, 2014. Appendix to Computer Organization and Design: The Hardware/-Software Interface.

[14] Dr. Lawlor. Instruction encoding & frequency. https://www.cs.uaf.edu/2012/spring/cs641/lecture/01_24_encoding.html, 2012. [Online; accessed 8-April-2016, Archived by WebCite ®http://www.webcitation.org/6gdfU63WO].

[15] Robert Love. *Linux system programming*. O'Reilly Media, Inc, Sebastopol, CA, 2013.

[16] Microsoft. What is a driver? http://windows.microsoft.com/en-us/windows/what-is-driver#1TC=windows-7. [Online; accessed 26-May-2016, (Archived by WebCite at http://www.webcitation.org/6hoOf8I6f].

[17] University of Copenhagen. Kudos—københavns universitets didactic operating system. https://github.com/DIKU-EDU/kudos.

[18] University of Pittsburgh. Exception and interrupt handling in the mips architecture. https://people.cs.pitt.edu/~don/coe1502/current/Unit4a/Unit4a.html. [Online; accessed 09-May-2016, (Archived by WebCite at http://www.webcitation.org/6hPGPDvWy)].

[19] OSDev.org. Interrupt vector table. http://wiki.osdev.org/Interrupt_Vector_Table, 2016. [Online; accessed 07-April-2016, (Archived by WayBack machine at https://web.archive.org/web/20160427193853/http://wiki.osdev.org/Interrupt_Vector_Table)].

[20] OSDev.org. Interrupts. http://wiki.osdev.org/Interrupts, 2016. [Online; accessed 19-May-2016, Archived by WayBack Machine at https://web.archive.org/web/20160519163614/http://wiki.osdev.org/Interrupts].

[21] OSDev.org. I/o ports. http://wiki.osdev.org/I/O_Ports, 2016. [Online; accessed 09-May-2016, (Archived by WayBack machine at https://web.archive.org/web/20160512212121/http://wiki.osdev.org/I/O_Ports)].

[22] David Patterson. *Computer organization and design : the hardware/software interface*. Morgan Kaufmann, Oxford Waltham, MA, USA, 2014.

[23] David A. Patterson and David R. Ditzel. The case for the reduced instruction set computer. *SIGARCH Comput. Archit. News*, 8(6):25–33, October 1980.

[24] Imagination Technologies Group Plc. Mips microprocessors overview. http://cdn.imgtec.com/mips-training/mips-basic-training-course/slides/Memory_Map.pdf. [Online; accessed 18-April-2016, Archived by WebCite ®at http://www.webcitation.org/6guotdraU].

[25] Imagination Technologies Group Plc. Mips microprocessors overview. https://imgtec.com/?do-download=4408, 2014. [Online; accessed 29-March-2016].

[26] Imagination Technologies Group Plc. Mips® architecture for programmers vol. iii: Mips32® / micromips32™ privileged resource architecture. https://imgtec.com/?do-download=5145, July 10, 2015. [Document Number: MD00090, Revision 6.02].

[27] Dominic Sweetman. *See MIPS run*. Morgan Kaufmann Publishers/Elsevier, San Francisco, Calif, 2007.

[28] Linus Torvalds. Linux stable kernel, mainline 4.6, 2016-05-15. https://www.kernel.org/.

[29] Central Connecticut State University. Memory layout. http://chortle.ccsu.edu/assemblytutorial/Chapter-10/ass10_3.html, 2015. [Online; accessed 10-April-2016, Archived by WebCite ®at http://www.webcitation.org/6gfKPa3Iu].

[30] Matt Welsh. Mips r2000/r3000 architecture. http://www.eecs.harvard.edu/~mdw/course/cs161/handouts/mips.html, 2005. [Online; accessed 20-April-2016, Archived by WebCite ®at http://www.webcitation.org/6guuHNcwJ].

[31] Wikipedia. Branch delay slots. https://en.wikipedia.org/wiki/Delay_slot#Branch_delay_slots, 2016. [Online; accessed 9-April-2016, Archived by WebCite ®at http://www.webcitation.org/6geBm0bpx].

[32] Wikipedia. Clock rate. https://en.wikipedia.org/wiki/Clock_rate, 2016. [Online; accessed 7-April-2016].

[33] Xillybus.com. A tutorial on the device tree (zynq) – part i. http://xillybus.com/tutorials/device-tree-zynq-1. [Online; accessed 20-May-2016, (Archived by WebCite at http://www.webcitation.org/6hdeUTNk1)].