MIPSter32 - A 32 bit MIPS Simulator

João Carlos de Oliveira Quintas

Thesis to obtain the Master of Science Degree in

Engenharia Informática e de Computadores

Supervisors: Prof. Dr. Nuno Miguel Carvalho Santos
            Prof. Dr. José Carlos Alves Pereira Monteiro

Examination Commitee

Chairperson: Prof. Dr. Ernesto José Marques Morgado
Supervisor: Prof. Dr. Nuno Miguel Carvalho dos Santos
Members of the Commitee: Prof. Dr. Nuno Filipe Valentim Roma

June 2016
Abstract

MIPS is a RISC Instruction Set Architecture (ISA) that is widely used in the industry and taught in many computer science courses. In particular, the 32-bit version of the MIPS architecture is the basis of the popular textbook Computer Organization and Design by Hennessy and Patterson, which is also used in the Computer Organization course at IST. To effectively teach computer science students the internals of a RISC processor, an adequate MIPS32 simulator is a valuable tool. However, existing simulators have important drawbacks and fall into one of two categories. In one category, we find simulators like SPIM which implement the MIPS32 instruction set, but lack the visual controls that are necessary to acquire a good insight of the internals of the processor. In a second category, we find simulators like WinMIPS64 which provide a good graphical user interface (GUI), but are incompatible with MIPS32. MIPSter32 is a simulator that is both compatible with MIPS32 and provides visual controls of the simulation process. Takes compiled MIPS32 program, and simulates the execution of the program instructions in the pipelined architecture of the processor. The GUI allows the user to observe the execution state of instructions in the multiple stages of the pipeline, and inspect the content of the processor registers and main memory. The GUI also offers controls for a stepwise execution of the program. MIPSter32 is internally designed to support future extensions, namely the incorporation of a cache simulator. The simulator is expected to be used in the future by the students of the Computer Organization course.

**Keywords:** MIPS, 32-Bit architecture, Simulation, Educational Simulation Tools, Modular, Extensible, Open-Source
Sumário

MIPS é um conjunto de instruções (ISA) RISC, que é amplamente utilizado na indústria e ensinado em vários cursos de ciência da computação. Em particular, a versão da arquitetura MIPS de 32 bits é a base do livro popular Computer Organization and Design por Hennessy and Patterson, que também é utilizada na cadeira Organização de Computadores no IST. Para ensinar eficazmente alunos de ciência da computação os detalhes de um processador RISC, um simulador de MIPS32 adequado é uma ferramenta valiosa. No entanto, os simuladores existentes têm desvantagens importantes. Numa categoria, encontramos simuladores como SPIM que implementa o conjunto de instruções MIPS32, mas não possuem os controles visuais que são necessárias para adquirir uma boa perspectiva do processador. Na segunda categoria, encontramos simuladores como WinMIPS64 que proporcionam uma boa interface gráfica (GUI), mas são incompatíveis com MIPS32. MIPSter32 é um simulador que é compatível com MIPS32 e proporciona controles visuais do processo de simulação. MIPSter32 pega no programa compilado, e simula a execução das instruções de programa na arquitetura do processador. A GUI permite ao utilizador observar o estado de execução de instruções em múltiplos estágios do pipeline, e inspecionar o conteúdo dos registos do processador e memória principal. A GUI também oferece um controlo para execução por passos do programa. MIPSter32 está concebido para suportar internamente extensões futuras, ou seja a incorporação de um simulador de cache. O simulador está previsto ser usado no futuro pelos alunos do curso de Organização de Computadores.

**Keywords:** MIPS, Arquitetura 32-Bit, Simulação Educacional, Modular, Extensível, Open-Source
Acknowledgments

I want to thank all the people that directly and indirectly helped me complete this thesis. I start with a special thanks for my Professors Nuno Santos, José Monteiro, and Ricardo Chaves for all the support and availability that they demonstrated along this project. I also want to thank them for believing in my capability to complete this project with success.

I also want to thank all the students that participated and gave excellent feedback for this thesis. Finally I want to thank my family for all the moral support that they gave during my college years.
# Contents

Abstract iv

Sumário vi

Acknowledgments viii

1 Introduction 3

2 Concepts and Related Work 7

2.1 Basic MIPS Concepts .................................................. 7

2.1.1 MIPS ISA .............................................................. 7

2.1.2 MIPS Instruction Types ............................................. 8

2.1.3 MIPS Pipeline .......................................................... 9

2.1.4 Microprocessor without Interlocked Pipeline Stages (MIPS) Pipeline .......... 10

2.1.5 Data Hazards .......................................................... 11

2.1.6 Control Hazards ...................................................... 12

2.1.7 Structural Hazards .................................................... 13

2.2 Related Work ............................................................ 13

2.2.1 Target System ......................................................... 13

2.2.2 Simulation Scope .................................................... 17

2.2.3 Abstraction Level ..................................................... 24

2.2.4 Graphical Interfaces ............................................... 26

2.3 Summary ................................................................. 31
3 Proposed Solution

3.1 MIPSter32 Architecture ............................................. 33
   3.1.1 Module ..................................................... 34
   3.1.2 Wire ...................................................... 36
3.2 Graphical User Interface ........................................... 37
   3.2.1 User Survey ............................................... 37
   3.2.2 GUI Design ............................................... 38
3.3 Programming Language ............................................ 41
3.4 Unit Tests ....................................................... 42
3.5 Used MIPS Assembler ............................................. 42
3.6 Documentation and User Manual ................................. 43
3.7 Summary ......................................................... 43

4 Implementation ..................................................... 45

4.1 Implementation Overview ........................................ 45
4.2 Hardware Building Blocks ...................................... 46
   4.2.1 Hardware Simulation Components ......................... 47
   4.2.2 Module ................................................... 47
   4.2.3 Combinatorial Modules .................................. 47
   4.2.4 Register Modules ........................................ 51
   4.2.5 Composite Modules ..................................... 52
   4.2.6 CPU ....................................................... 53
   4.2.7 Wires ..................................................... 54
   4.2.8 Memory .................................................. 55
4.3 Pipelined MIPS Architectures ................................... 56
   4.3.1 Simple Pipelined MIPS Architecture .................... 57
   4.3.2 Jump Implementation .................................... 61
   4.3.3 Hazard Detection Unit Implementation .................. 63
   4.3.4 Branch Not Equal Implementation ........................ 64
B.1 Graphical User Interface .................................................. 95
B.2 Import QtSpim code to MIPSter32 ...................................... 96
B.3 Execute a Program ........................................................... 96

C Conducted Survey ............................................................. 99
List of Tables

2.1 Comparison of MIPS processor simulators. Referenced in Section 2.2.2.2 . . . . . . 24

4.2 Arithmetic logic unit (ALU) Control Lines: Defines the six combinations of four
control inputs of the ALU. This table was based on the Patterson’s book [33]. . . . 68

4.3 Conversion Table for the ALU operation code. This table shows how to set
the control inputs for the ALU using the 2-bit ALUOp control in combination with
the instruction’s operation code. The XX denotes that for a specific operation the
function field is ignored. This table was based from the Patterson’s book [33] . . 68

5.4 Performance test results. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

5.5 Cycles that each architecture takes to execute the same assembly program. Legend: NO: No Optimization; DS: Delay Slot; BPNT: Branch Prediction Not Taken;
BO: Branch Optimization; DS/BPNT: Delay Slot and Branch Predict Not Taken;
DS/BO: Delay Slot and Branch Optimization. . . . . . . . . . . . . . . . . . . . . . . . . 78
List of Figures

2.1 R-Type instruction format. .................................................. 9
2.2 I-Type instruction format. .................................................. 9
2.3 J-Type instruction format. .................................................. 9
2.4 Simple five stage MIPS pipeline diagram. .............................. 11
2.5 (a): MiniMIPS single cycle simulator program modules. (b): Stages in the multicycle simulator. [12] ........................................... 23
2.6 COSMOS Program Structure. ............................................. 27
2.7 (a): WebMIPS pipeline diagram. (b): WinDLX, WinMIPS64 and pipeline diagrams. 28
2.8 The laundry analogy for pipelining. Image taken from the Pattersons book [33] . . 30
2.9 WinMIPS64 cycle panel. ..................................................... 30
3.10 Modules UML diagram. .................................................... 35
3.11 Graphical representation of a clock cycle over time. ............... 36
3.12 MIPSter32 implemented GUI. ........................................... 38
4.13 Combinatorial module UML diagram. ................................. 51
4.14 Register module UML diagram. ......................................... 52
4.15 Composed module UML diagram. ..................................... 53
4.16 CPU UML diagram. ....................................................... 54
4.17 Wire UML diagram. ....................................................... 55
4.18 Simple Pipelined MIPS architecture. .................................. 57
4.19 Simple Pipelined MIPS architecture implementing Jump instruction. .............. 62
4.20 Simple Pipelined MIPS architecture with Hazard Detection Unit. .................. 63
4.21 Simple Pipelined MIPS architecture with Hazard Detection Unit. 64
4.22 GUI UML diagram. 67
5.23 MIPSter32 initial state. 70
5.24 MIPSter32 state after some cycles. 70
5.25 ALU diagram implemented by João Vieira. 74
5.26 Branch prediction not taken implemented by João Vieira. 75
5.27 Branch optimization implemented by João Vieira. 76
5.28 Cycle analysis between different architectures performed by João Vieira. 76
B.29 MIPSter32 GUI. 97
Chapter 1

Introduction

Computer architecture is a set of disciplines that teach and describe a computer system by specifying its parts and explaining their relations. Any computer science course includes at least one computer architecture or organization discipline. A computer system can be seen as layers of abstractions on top of each other where one end we have the digital electronics such as caches, pipelines, memory, registers, among others, and the other end we have the complex applications using techniques such as polymorphic inheritance [15]. In order to learn computer architectures and systems students need to understand and master abstractions [15]. Professors over the years have studied, tested, and implemented various techniques to optimize the learning of their students. Around the 70’s and 80’s students used paper and pencil to design CPU components using Boolean algebra and Karnaugh maps, but as technology evolved teaching methods have also grown by relying on simulators to study computer operation and visualize the interrelated, simultaneous events that occur during program execution [47].

MIPS is a reduced instruction set computer (RISC) instruction set architecture (ISA) developed by MIPS Technologies, that greatly influenced later RISC architectures. Due to its simplicity, it is vastly used in university courses and technical schools to teach computer architectures. One of the most used and famous computer architecture book, the Hennessy and Patterson book [33], teaches architectural concepts using MIPS. Even though there are many simulators, they are not adequate to the Patterson book mainly because they do not simulate a 32-bit architecture.

The goal of this project is to build a MIPS (Microprocessor without Interlocked Pipeline Stages) 32-bit simulator to follow as close as possible the Patterson book [33]. This is the chosen book because it is widely used in the most prestigious universities around the world to teach computer architecture classes. The vastly usage of this book [33] will increase the impact of this work significantly, offering a great tool for professors around the world. The simulator will mainly target
the processor and MIPS pipeline focusing on the first chapters of the Patterson book [33], leaving out the memory simulation for future work. The final work will be used in computer architecture classes by the professors and students at Instituto Superior Técnico.

The main requirements for this work are grouped in three main topics. Firstly, the final prototype must be user friendly, aiding students learning CPU architectures and fundamental aspects such as pipelining. To accomplish this a major requirement is the implementation of a graphical interface to display what is happening at each clock cycle, helping students demystify the black box that a CPU may be. The interface should aid students visualize how a pipeline works, give useful information on important statistics, show stalls, among others. Secondly, the simulator must be independent of the operating system making it accessible to a greater number of students. The final requirement is to have a modular and extensible design, offering a simple way for users to implement a cache or a multiprocessor, and even change some components such as the ALU.

Although there has been extensive work on simulation, specifically for MIPS, none of the existing systems satisfies entirely our requirements. There is a large and excellent offer on CPU simulators, a range of different architectures simulation, multiple Instruction Set Architecture (ISA), and with distinct purposes. Still there is a need to create a 32-bit simulator that follows the Patterson [33] book closer and with more detail. The WinMIPS64 [37] and the WebMIPS [14] have a rich interface that simulates and teaches crucial aspects of the pipeline but simulate a different ISA than the Patterson book [33]. Being 64-bit makes it incompatible with the 32-bit because they have different instructions, pipelines, and ISAs. MARS [36] has a very complete IDE, implements breakpoints, along with other great features but misses out on the pipeline visualization.

Contributions The main contribution of this work is the design and implementation of MIPSter32, a 32-bit MIPS ISA simulator. As described earlier, there are several offers of processors simulators, but none that follow the Patterson [33] book closely together with having a Graphical User Interface (GUI) designed to help students learn the way a processor works. MIPSter32 simulates the MIPS 32-bit ISA described in Patterson [33] book. We highlight some of the most important features of MIPSter32:

- **Graphical User Interface (GUI)** The developed GUI is based on WinMIPS64 [37], mainly because it has a simple yet complete GUI. To define the most adequate interface that meets user needs nothing better then conducting a survey with Instituto Superior Técnico (IST) students to refine several aspects of the WinMIPS64 GUI, later detailed on Section 3.2.

- **Modular and Extensible** The greatest differentiating feature of MIPSter32 is the modular and extensible architectural design. Having a modular and extensible design allows
a community based development, allowing MIPster32 adaptation to specific user needs. As proven in Section 5.4, the implemented architecture allows complex modification to the MIPster32 core behavior with little development effort.

- **Documentation** The developed documentation is divided into user’s and developer’s manuals. The user’s manual specifies all graphical components present on the GUI along with all possible actions that may be performed. Both manuals include a detailed tutorial on how to import compiled code from QtSpim to be executed on the MIPster32. The developer’s manual include a detailed description of the project’s structure along with a complete Javadoc.

- **Compatibility and Portability** MIPster32 was all developed in Java for compatibility and portability purposes. Since it is a learning tool it is essential that the simulator runs on all main operating systems available on the market. Besides the compatibility and portability advantages, most developers have programmed using Java, decreasing the language barrier for future developers who decide to modify and extend MIPster32.

- **Test Set** To validate that MIPster32 has the desired behavior, a test set was implemented, later described in Section 5.3. The test set consists of test cases that will test all possible aspects of the simulation as well as GUI helper functions, guaranteeing the correct execution.

**Document Organization** The contents of the dissertation are organized as follows. Chapter 2 presents the related work. Chapter 3 describes the proposed solution’s architecture, software architecture, GUI, programeing language, unit testing, and documentation. In Chapter 4, the implementation of the proposed solution is described. Chapter 5 details the evaluation methods that were implemented. The last chapter, Chapter 6, presents the conclusions of the dissertation along with future work.
Chapter 2

Concepts and Related Work

This chapter starts by explaining the basic MIPS concepts that are present along this work. Following this, we present the related work on computer architecture simulators.

2.1 Basic MIPS Concepts

MIPS is a RISC ISA that attempts to achieve high performance with the use of a simple instruction set [23]. In 1982, the MIPS architecture was experimental since it was a radical break with the trend on modern computer architectures. The processor has a five stage architecture without pipeline interlocks, and the basic philosophy of MIPS was to present an instruction set that is a compiler-driven encoding of the micro-engine. Thus, little or no decoding is needed and the instructions correspond closely to microcode instructions. In the following sections, we introduce the main concepts of MIPS.

2.1.1 MIPS ISA

MIPS is a RISC Instruction Set Architecture (ISA). There is a big range of Instruction Set Architectures all developed and designed to target specific needs. ISAs are generally classified as:

- Reduced Instruction Set Computer (RISC)
- Complex Instruction Set Computer (CISC)

Complex instruction set computing (CISC) is a CPU design where a single instruction translates
into several low-level operations or that are capable of multi-set operations. In pedagogical terms translates to a greater complexity to understand how an instruction is decoded and processed on the CPU pipeline. On the other hand, reduced instruction set computers (RISC) aim for simplicity in both hardware design and synergy between architectures and compilers [32]. Since RISC is often less complex CPU design this reflects directly on the effort needed for students to understand the underlying concepts of instruction processing on the CPUs pipeline, allowing students to focus on the study of the pipeline stages.

The MIPS ISA, being the choice of Brorsson [15] for his simulator due to its simplicity, is one of the main reasons to use it for teaching simulators. Also, due to its simplicity students easily understand the instruction set and rapidly switch their attention to the pipeline, types of instructions, hazard resolutions, and optimizations such as delay slots.

2.1.2 MIPS Instruction Types

MIPS has three instruction formats:

R-Type: Include instructions that do not require a target address, immediate value, or branch displacement. Figure 2.1 describes how the instruction bits are arranged.

- **op**: Operation code (also known as opcode) that selects a specific operation.
- **rs**: The first source register.
- **rt**: The second source register.
- **rd**: The destination register.
- **sa**: The shift amount that is only used on shift instructions.
- **func**: Is used together with the opcode to select an arithmetic instruction.

I-Type: Include instructions such as load, store, branch and arithmetic instructions that use immediate values. Figure 2.2 describes how the instruction bits are arranged.

- **op**: Operation code (also known as opcode) that selects a specific operation.
- **rs**: An address for loads and stores, or an operand for branch and immediate arithmetic instructions.
- **rt**: A source register for branches, but a destination register for the other I-type instructions.
- **address**: The immediate value of the instruction.
2.1. BASIC MIPS CONCEPTS

2.1.3 MIPS Pipeline

One of the most effective architectural changes for performance reasons was the introduction of a pipeline. A non-pipelined Central Processing Unit (CPU) executes instructions sequentially, whereas a pipelined architecture splits up into a sequence of steps so different steps can be executed in parallel. This difference will drastically affect the instruction throughput of a processor.

Non-Pipelined Architectures: When using a non-pipelined architecture the CPU frequency will be smaller, since it is necessary to leave enough room for all the components of the CPU to complete their task. Each component will have to wait for the previous one for the dependable output result to be computed. The overall system will be slow, but will maintain a constant Cycles Per Instruction (CPI) of one. The main advantage of non-pipelined architectures is that no
hazards will occur due to the fact that each instruction will be executed in one cycle.

**Pipelined Architectures:** When a pipeline is introduced, the frequency is drastically increased, because at each clock cycle the amount of components that depend on each other decreases, because the maximum frequency will depend on the slowest stage of the pipeline. The main advantage that the pipeline will offer is the increase of instruction executing simultaneously. In a pipelined architecture, at a given cycle, the pipeline may be processing N (where N is the number of stages) instructions at the same time. The problem when implementing a pipeline will come from hazards that may occur during the execution. The complexity of a pipelined architecture will be much higher when compared to a non-pipelined, because of all the extra hardware components necessary to make sure that no hazard occurs during the execution.

**Implementation Complexity** The architectural differences can also reflect on instruction implementations. A simple architecture may only implement a subset of the ISA, whereas a more complex one may implement the entire ISA. There are MIPS architectures that do not support floating point operations, simply because the necessary hardware components are not implemented. Others may not support some operations such as shifts, or branches.

**32 bit vs. 64 bit** Another very notable difference is the bits that the CPU works with. There are MIPS architectures that use 64 bits and others that use 32 bits. This is a more profound change on the architecture simply because these two are very different from each other. It affects the maximum amount of Random Memory Access (RAM) that the processor can support and the computation speed is also affected. These two differences have a great impact on the architecture because all components are affected.

### 2.1.4 MIPS Pipeline

Pipelining improves throughput rather than individual instruction execution time [33]. In Figure 2.4 a simple block diagram of MIPS five stage pipeline is shown. The MIPS pipeline is composed of five different stages that are as follows:

- **Instruction Fetch**: This stage is where the next instruction is read from memory and placed on the pipeline to be executed.

- **Instruction Decode**: In this stage the instruction is decoded and the necessary values are read from the register bank.
2.1. BASIC MIPS CONCEPTS

Figure 2.4: Simple five stage MIPS pipeline diagram.

- **Execute**: This is where all the arithmetic and logic operations take place.

- **Memory Access**: As the name suggests, this stage of the pipeline reads and writes the necessary values to memory.

- **Write Back**: This stage writes to the register bank the necessary values from memory reads, or Arithmetic Logic Unit (ALU) operations.

2.1.5 Data Hazards

Data hazards occur when a stall has to be introduced in the pipeline due to a data dependency of an instruction that has not yet terminated [33].

```plaintext
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

In the code sample above, the subtraction instruction requires the result of the previous addition. The register $s0 is the addition of registers $t0 and $t1. But since the MIPS architecture is pipelined, the subtraction instruction will be placed in the pipeline and executed before the add instruction is completed. Stalls have to be placed on the pipeline to allow the add instruction to complete before executing the subtraction instruction. This instruction sequence is an example of a data hazard.

**Pipeline Stall**  Pipeline stall or bubble, is a delay in the execution of an instruction to resolve a hazard. At the decode stage, the control unit determines if the decoded instruction reads from a register that the instruction currently in the execution stage writes to. If this is verified, then a stall is introduced in the pipeline to avoid potential incorrect computation results. Stalls are also introduced when a register is going to be read and a load instruction will write a value to one of the operand registers.
Forwarding  Forwarding occurs when the result is passed forward from an earlier instruction to a later instruction, "bypassing" the register file to the desired unit. [33] In the example shown on Section 2.1.5 instead of waiting for the write back of the add instruction, the ALU could be directly connected to the register stage, forwarding the result to be used by the sub instruction. With this solution there would be no stalls inserted in the pipeline, resulting in a faster execution of the program excerpt.

2.1.6 Control Hazards

Control hazards, also known as branch hazards, occur when a branch instruction is executed and the branch is taken. The following techniques are used to minimize the impact of control hazards during the execution.

Branch Prediction  Branch prediction is a technique aimed at reducing pipeline contention caused by control hazards. When a branch operation is executed two possible scenarios are possible. The branch condition is met and the program counter is set to the correct instruction of the conditional statement, or the condition is not met and the instructions that are encapsulated on the branch condition will not be executed. While the correct program counter value is not calculated a bubble has to be placed on the pipeline, producing a stall on the execution. The branch prediction will always execute one of two possible instructions, discarding it if the prediction was wrong. Branch prediction can be implemented on the architecture to prevent some stalls, and thus improving its performance. This solution will not eliminate all the stalls caused by branches, because the possible outcome varies. There are two ways that branch prediction can be implemented:

- **Branch Taken**: In this case, the architecture assumes that the branch will always be taken, placing the next instruction executing on the pipeline. If the branch is then taken, the correct instruction will already be executing on the pipeline, consequently avoiding a stall. If the branch is not taken, the instruction will be discarded.

- **Branch not Taken**: It is the same as the branch taken but chooses the instruction that would be executed if the branch was not taken.

Branch Delay Slots  Another common control hazard mitigation technique is to use branch delay slots. Branch delay slots are used to minimize the number of stalls that will occur during the execution. Instead of inserting a bubble on the pipeline when there is a branch instruction,
the next instruction on the program memory will be placed to execute, even though it could be the wrong instruction. After the branch target is calculated, the control unit will see if the instruction that has been executing is the correct one. This technique will avoid stalls in some cases, improving the throughput of the CPU.

2.1.7 Structural Hazards

A structural hazard happens when a part of the processor's hardware is needed by two or more instructions at the same time. An example is a single memory unit that is accessed both in the fetch stage and the memory stage where data is written and/or read from memory.

2.2 Related Work

Architectural simulation is a widely used technique for various purposes, consequently having different abstraction levels and categories. A simulator can target hardware components like WinMIPS64 [37], instruction set architectures such as MARS [45], and computer systems including processors, memories and I/O such as GEMS [29]. The studied simulators will be categorized in several according to the goals and requirements of this thesis. In Section 2.2.1, the various target systems will be introduced with several examples, followed by simulation scope on Section 2.2.2 separated into full-system (Section 2.2.2.1) and micro-architecture (Section 2.2.2.2). The various abstraction levels of simulation will be in Section 2.2.3, ending with the different graphical interfaces of the presented simulators on Section 2.2.4.

2.2.1 Target System

Categorization helps conceptually visualize patterns and similarities by grouping them together. Relevant simulators can be categorized into: historical machine, followed by digital logic simulators, simple hypothetical machine simulators, intermediate instruction set, advanced micro-architecture, multi-processor, and finally the memory subsystem simulators. These seven main categories were defined by Wolfe [47].

**Historical Machine Simulators** Professors often enhance learning activities using examples of computers that no longer exist, are too expensive to buy solely for educational purposes, or are only available in museums [47]. With a careful analysis and simulation expertise, a computer can be virtually recreated and indefinitely maintained [47]. Simulation overcomes these barriers in
simple and effective way. Therefore professors can rely on simulators to explain and demonstrate specific architectural concepts.

Historical computers like the Analytical Engine by Fourmilab [2], the Apple IIE [43], or the Nazi’s encryption machine Enigma [1] are some examples of historical simulators. The analytical engine was a mechanical general-purpose computer, and it’s web based Java applet simulator allows the user to experience with cards and an old machine in the comfort on his/hers favorite browser. The Enigma simulator takes the user into an historical journey through the World War II encryption technology. This experiences would be impossible without the use of simulation because of the inaccessibility and rarity of such machines.

Digital Logic Simulators When studying computer architecture, a bottom-up foundation is commonly used [47]. At Instituto Superior Técnico, the Computer Science and Engineering Department adopts this approach, where students start learning basic logic components such as AND/OR/XOR/NAND gates, flip-flops, latches, multiplexers, comparators, among others. Digital logic simulators, as the name points out, simulate circuit designs at the digital level. Digital logic simulators are a great tool to teach and demonstrate how the computer architecture foundations work [47].

The 6.111 C simulator is a time domain digital simulator that is used to build a program that simulates the operation of a circuit [21]. One other example is the SmartSim [6]. This free and open source digital logic simulator allows the user to create from the most simple to the most complex circuits and allows the creation of custom components as if they are a built-in component. The last feature opens the possibility of testing new logic components in an easy and controlled environment. The Iowa Logic Simulator [25] is an interactive program which, through the Iowa Logic Specification Language, allows digital circuit design to be tested without the need to actually build them.

Simple Hypothetical Machine Simulators As the complexity of modern machines increases, it becomes less suitable to teach the basic concepts usually found in introductory computer organization courses [47]. Simple Hypothetical Machine simulators serve an important role in teaching by giving students a simplified version of the internal operation of a system [47]. They excel at illustrating core concepts such as the Von Neumann architecture [46], the stored program concept [9], the use of registers, among others. In sum, the hypothetical machine simulators allow educators to selectively focus on the desired concepts without losing focus of complex machine details [47].

A great example of a Hypothetical Machine simulator is the CPU Sim [40]. It is a Java application
2.2. RELATED WORK

that allows the design of simple computer CPU. At the microcode level that can be used to simulate a variety of architectures including accumulator-based, RISC-like, or stack-based (JVM) architectures [40]. The author claims that instructors may use CPU Sim to help their students get hands-on exposure to a variety of architectures and write programs in machine language for their designed and implemented architectures. One other example is the RTLSim [8]. This simulator is a UNIX program that simulates the data-path of a simple non-pipelined MIPS processor. It allows the user to select the control signals that will be active at each control step, making a great tool for students to understand how the control unit works on the MIPS CPU.

Intermediate Instruction Set Simulators In contrast with the Simple Hypothetical Machine simulators [47], the Intermediate Instruction Set simulators include a more realistic set of addressing modes, more complete instruction sets, more realistic memory hierarchy simulation, and sometimes even include an interrupt mechanism. Being more complete in terms of simulation, these simulators provide a more realistic programming application and give the opportunity to explore computer science concepts that are important throughout the broader curriculum.

SimNow [5] is one example of Intermediate Instruction Set simulators. It is a fast and configurable x86 and AMD64 dynamically- translating instruction level platform simulator [5]. SimNow is widely used by AMD and their partners in a commercial role for BIOS and device driver development, prototyping software visible architectural changes, non-intrusive and deterministic measurement and testing of software at the instruction-level, and modeling of future platform trade offs for correctness and performance analysis [5].

Spim [30] is a free and open-source simulator developed in C++ that runs MIPS 32 bit programs. A new version entitled QtSpim expands the old Spim to multiple platform support maintaining the same user interface. Spim and QtSpim do not include a code editor, forcing the user to edit the assembly code with an external tool. It offers debugging tools such as step-by-step and breakpoints, the content of the memory and registers is always updated and available to the user making it a great MIPS 32 bit assembly language debugger.

Advanced Micro-Architecture Simulators Advanced Micro-Architecture simulators [47] are designed to show how machine language is executed at the microcode level. This includes the study of data paths, control units, and sometimes memory access times. These simulators are often used to investigate the advantages and disadvantages of performance-enhancing techniques such as pipelining, branch prediction and instruction level parallelism. More detailed examples of MIPS Intermediate Instruction Set simulators can be found in Section 2.2.2.2. The simulator proposed in this thesis should be extensible enough to accommodate such a feature in future
work.

**Multi-Processor Simulators** As the name suggests, Multi-Processor simulators [47] are targeted for CPUs with more than one core. This requires the emulation of features such as shared interconnection networks and shared memory that do not exist in uniprocessors. It is also required correctness on the simulation of simultaneous execution that reflects the fact that instructions on different processors are occurring simultaneously. One great challenge with multi-core simulation is keeping the simulation time realistic, because execution time increases along with the number of cores being simulated.

One example of a MIPS R3000 based multiprocessor simulator is Mint [44]. Mint is a software package designed to ease the process of constructing event-driven memory hierarchy and provides a set of simulated processors that run standard Unix executable files. A pioneer hybrid technique that exploits the best aspects of native execution and software interpretation minimizes the overhead of the simulation making Mint very efficient on uniprocessor hosts [44]. An example of a full-system(Section 2.2.2.2) multi-core simulator is the SimOS [34]. SimOS simulates the hardware in such a detailed manner that it is capable of booting a commercial operating system and run realistic workloads. The implementation of interchangeable simulation models for each hardware component allows the user to control explicitly the trade off between simulation speed and simulation detail.

**Memory Subsystem Simulators** CPU simulators are great tools for various purposes, but to fully comprehend code optimization, memory hierarchies and cache configurations are an essential part of computer architecture courses. The memory subsystem simulators are designed to allow the user to configure cache memory levels, block sizes, and associativity and reveal execution detailed statistics on performance metrics such as cache hit ratios [47].

An exact cache simulator for single processor cache is the Dinero IV [24] developed by Jan Edler and Mark Hill. Its purpose is to estimate several memory statistics such as number of cache misses given a specific configuration. Some features include separate or combined instruction and data caches and simulation of various cache levels. Dinero IV is designed to show performance related statistics to the user rather then teaching concepts. Another memory subsystem simulator is the Cacheprof [39]. It is a tool designed to help find where and why a program causes cache misses. This simulator is different from most memory simulators because it targets programmers and algorithm designers that fully understand what a cache is [39]. Cacheprof will not profile instruction cache effects, but will count data reads and writes [39].
2.2. RELATED WORK

2.2.2 Simulation Scope

2.2.2.1 Full-System Simulators

Full-System simulation is an architecture simulator that simulates an entire electronic system at a high detail level. It allows complete software stacks from real systems to be executed on the simulator without any extra modifications. A full-system simulator must effectively provide virtual independent hardware from the host machine. These types of simulators usually offer processor cores, peripheral devices, memories, interconnection, among others [7]. Popular full-system simulators such as the COTSOn [7], PTLsim [31], the MARSS [31], the M5 [13], and the GEMS [29] are great examples that illustrate what a full-system simulator is. This kind of simulators do not correspond to the goal of our simulator because of their wide simulation range of the system. To demonstrate and teach specific concepts such as pipelining, a micro-architecture simulator is more adequate to fulfill this purpose. Even so, it is important to demonstrate the basic concept for a better understanding of why micro-architecture simulators are the appropriate type for the objectives and requirements of this thesis.

COTSOn HP Labs and AMD joined forces to develop the COTSOn [7] simulator framework. The goal was to provide fast and accurate evaluation of current and future computing systems, covering the full software stack and complete hardware models. COTSOn adopts a functional-directed philosophy and targets cluster-level systems where fast functional emulators and timing models cooperate to improve the simulation accuracy at a speed that is able to simulate the full stack applications, middleware and operating systems [7].

COTSOn is unique by the capabilities to dynamically adjust accuracy and speed. Due to this characteristic, COTSOn can be used as a micro-architectural level or as a full-system tool to optimize the combination of computing resources to match customer requirements in terms of performance, power and cost. The micro-architectural allows researchers in computer architecture to measure performance of their new branch predictor or cache prefetching [7], while the full-system simulation allows the researchers to test out the impact of certain architectural changes on the entire system.

PTLsim PTLsim is the an open source cycle-accurate full-system x86 and x64 microprocessor simulator and can also be used as a virtual machine, if desired [48]. It models a complete modern superscalar out-of-order processor with a configurable level of detail ranging from RT level (described in more detail on Section 2.2.3) of all key components such as pipeline structures, caches, memories among others. This simulator targets commercially available ISAs [48].
PTLsim provides full system simulation capabilities by using the Xen VM framework along with a modified Xen hypervisor [31].

Co-simulation is one of the strong selling points of PTLsim. This technique allows any virtual machine to be seamlessly switched between the host machine's physical CPUs, the native mode, and PTLsim's cycle accurate processor core models. All this is performed maintaining strict timing accuracy and continuity [48]. This co-simulation technology allows rapid profiling of relevant code segments, and makes the simulator self-debugging because it can be continuously validated against any commercial x86/x64 processors [48] that are found on almost all personal computers.

**MARSS** MARSS [31], not to be confused with MARS [45], is an open source full-system simulation tool built on QEMU [11] to support cycle-accurate simulation of superscalar homogeneous and heterogeneous multicore x86 processor. QEMU is used as a base framework that provides a full system environment capable of booting and running an unmodified OS [31]. MARSS includes detailed models of system components such as coherent caches, interconnections between various components, chipsets, memories and I/O devices. MARSS is also capable of simulating the execution of all software components present in the system, ranging from unmodified binaries of applications to operating system's libraries [31].

MARSS uses PTLsim [48] presented earlier, as the base of its CPU simulation environment, running everything on top of QEMU as explained before. MARSS is not just a simple port of PTLsim [48], but instead offers several additions such as MMX support, light assist functions, new statistics collection framework, switch from Xen to QEMU, and communication between simulator and virtual machine [31].

**M5 and GEMS** Simulators do not necessarily need to simulate a computer or a processor, just like the M5 [13] and the GEMS [29] that both simulate a complete network. In the case of the M5 we have a full-system simulation of network hosts. With a growth on net-centric software growing every day, the importance of simulating TCP/IP networking is gaining emphasis on the simulation world. The M5 is open source which have led to its adoption in both academic and commercial groups [13]. It offers a wide range of features including full-system capability, detailed I/O subsystems, and the ability to simulate multiple networked systems deterministically [13].

GEMS is a simulation tool set designed to evaluate multiprocessor architectures built on top of Simics [28]. Its purpose is to evaluate the performance of multiprocessor hardware systems commonly used on web servers and databases [29].

When modeling networked systems the key requirements to build a network simulator, according to the article [29], are as follows:
2.2. RELATED WORK

- the ability to execute operating-systems
- the ability to execute application code
- detailed performance models of both memory, I/O devices, and network interfaces
- the ability to model multiple network systems, allowing the configuration of the number of server/client ratio.

According to the implementation presented in [13], the M5 simulator meets all these requirements. Both simulators have detailed multiprocessor memory systems, but the GEMS lacks on offering a detailed I/O models and multiple-system capability [13]. These two examples are great to demonstrate how vast the simulation world is.

Both simulators were then merged as one called GEM5.

2.2.2.2 Micro-Architecture Simulators

A micro-architecture simulator is a tool developed to model the design and behavior of a microprocessor and its components; such as the control unit, ALU, cache memory, data paths, among others. This type of simulators is ideal for testing concepts such as branch prediction, cache trace, and reorder buffers. In addition, the simulation also allows educators to teach computer organization and architecture courses with hand-on experiences, but lack on some key points that do not absolutely complement the Patterson’s book [33]. This thesis fits on the micro-architectural simulation. Simulators can be compared against four criteria mentioned by Sarjoughian [36] that helps students to understand processor implementations. The criteria are as follows:

- **A**: It models the MIPS32 single-cycle, multi-cycle, and pipeline processor implementations described in [32] at the RT level
- **B**: It provides statistical data (including execution time, cycles per instruction (CPI), and cycle count) for performance comparison
- **C**: It provides run-time animation that demonstrates how the signals are sent between components at RT level during instruction execution
- **D**: It is platform independent so students can use the simulator on alternative hardware platforms.

There are many micro-architecture simulators such as Shade [19], Spim [30], MARS [45], Win-MIPS64 [38], EduMIPS64 [17], SimpleScalar [10], WinDLX, MiniMIPS [12], WebMIPS [14], Pro-cessorSim [22], among others. In this section the focus will be more one functional features.
SimpleScalar  SimpleScalar [10] toolset provides an open source infrastructure for simulation and architectural modeling for both researchers and instructors. The flexibility of the toolset allows a range of simulation that starts on a simple unpipelined processor up to a detailed dynamically scheduled microarchitectures with multiple-level memory hierarchies. If this is not enough, it offers a well-structured and documented design, allowing users to extend the toolset for their specific individual needs. SimpleScalar reproduces computing device operations by executing all program instructions using an interpreter that supports several instruction sets, such as x86, Power PC, ARM and Alpha [10].

Since SimpleScalar is an execution-driven simulator, it provides access to all data produced and consumed during the program execution. These values are essential to analyze and optimize compressed memory systems, dynamic power analysis and value prediction. SimpleScalar, due to its execution-driven simulation, can reproduce in great detail the speculative computation making it possible to correctly model its impact on program performance. Due to all its functionalities, the model complexity increases significantly, consequently making the program heavier to execute. Even though this is a widely used simulator, about 500 researchers up to this article’s date [10], it does not simulate MIPS ISA. Its complex design and features therefore target the simulator for more expert users making it an inappropriate tool to teach initial concepts of computer architecture classes [10].

MARS  MARS is a Java-based assembler and runtime simulator for the MIPS 32-bit ISA. The main goal of this simulator was to develop an alternative to SPIM [30] (see Section 2.2.1 under Intermediate Instruction Set Simulators) targeted for the typical undergraduate students and their professors. Regarding the implementation of the MARS simulator, due to the simplicity of the MIPS instructions, a separation between the specification of the MIPS instruction from the MARS source code was made. Macro instructions (or pseudo-instructions) are also possible by using a separate text file where the expanded instructions composing the macro are specified. This feature can be used in compiler writing courses where students define their language as macros and translate them to the corresponding MIPS instructions [45].

MARS limits the memory segments for text, data, kernel text and kernel data to 4 MB. Among these limitations, the error message highlighter does not select the first assembly error, and a memory leak problem has been pointed out to exist in the editor [45]. The MARS, even though it is a MIPS 32-bit simulator, does not demonstrate the main aspects that this thesis aims, such as teaching important concepts on the Patterson’s book [33].
2.2. RELATED WORK

WebMIPS  WebMIPS is an Active Server Page (ASP) and HTML MIPS 32-bit assembler and simulator. Users are able to upload and assemble their code through a web browser. It is also possible to visually see the five stage pipeline, step-by-step or completely, and the values of all registers. The input and output data of all pipeline elements are always updated and displayed to the user. The motivation for this work relies on the need of students that already used Intermediate Instruction Set simulators (described in Section 2.2.1) and want to have a more detailed coverage of computer architectures but are not ready for the simulator that captures all the features of the current state-of-the-art computer research [14]. It includes hazard detection and has forwarding always enabled. By being a web based simulator it allows students to use the simulator without any prior installation and the possibility of monitoring their activity over the Web by their professors. Students are not limited by their machine’s operating system [14].

WebMIPS is a good tool with its flaws. It is capable of recognizing errors in the provided code, but it is not a real assembler. Since it runs on a server and not on the host’s machine some measures that limit the execution had to be taken. WebMIPS is limited to 1000 clock cycles to each uploaded program to avoid blocking the server in case of infinite loops, erroneous memory references, and other common programming errors [14]. Even though WebMIPS is a 32-bit simulator that focuses on teaching there are some concepts covered on the Patterson’s book [33], such as data forwarding, that students may not experiment with because it is always enabled.

WinDLX, WinMIPS64, and EduMIPS64  All these three simulators belong to the same family, being WinDLX the predecessor of WinMIPS64 and EduMIPS64 a port to Java of its older version [38]. WinDLX simulates the DLX architecture while the other two focus on the MIPS 64 bit instruction set. Both WinDLX and WinMIPS64 only run on windows, while the most recent EduMIPS64 is capable of running on any operating system as long as a Java virtual machine is up and running. In all three simulators no editor is available, making it necessary to use an external tool to fulfill this need.

Besides the fact that WinDLX and WinMIPS64 simulate different architectures, there are some more noticeable changes. Forwarding is now indicated by coloring the forwarded register. This color indicates the stage in which it will be forwarded. Delay slots and branch predictions are features that can be switched to study the effects on the execution. WinMIPS64 allows users to change the values on the registers and on the memory manually. The memory values can also be displayed in double-precision decimal floating point format instead of their hexadecimal value. The break points can be set simply by left-double-clicking on the code window [38].
EduMIPS64, like the WinMIPS64, implements the five stages of the MIPS CPU pipeline attributing a similar color scheme to each of the different stages. The execution can be ran with or without forwarding, something that the WebMIPS [14] does not support. Having the possibility to enable or disable data forwarding allows students to compare CPI and execution times of the same executed with and without data forwarding allowing a detailed comparison that will demonstrate statistically the advantages of data forwarding. EduMIPS64 also supports synchronous exceptions like division by zero, as well as six system calls that include `exit()`, `open()`, `close()`, `read()`, `write()`, and `printf()`. One great feature that rarely appears on educational MIPS simulators like these three is a cache simulator. EduMIPS64 does not offer one, but eases the effort of integrating the execution of a program with a memory simulator by having built in functions that export specific files that can be fed to external cache simulators like the Dineroy IV [17]. Even though WinMIPS64 and EduMIPS64 have desirable features for teaching they both simulate a different architecture and ISA from the Patterson’s book [33].

**MiniMIPS** As it has been introduced earlier, MIPS is a 32-bit architecture, with posterior versions being 64 bits. The MiniMIPS is an adaptation of the real architecture to a subset of MIPS of 8-bit using only 8-bit datapaths, only eight registers implemented and an 8-bit program counter [26]. The motivation behind this project originated from a series of laboratory exercises designed by Dr K.A. Robins for teaching Computer Organizations classes [12]. Students were asked to write a parser that extracts fields from MIPS instructions. After completing this step, students were then asked to write a simulator for each of the functional units used, such as the ALU, instruction and data caches, and write a test program for each unit. The exercise then takes students to write a simulator for the control unit that produces the correct control signals from a MIPS instruction. The final step requires the integration of the datapath and control unit developed on previous steps [12].

The main motivation of this project was to build a simulator that teaches students the foundations of computer architecture and organization. It is mainly a tool designed to follow a certain type of exercises that help students understand the foundations before moving to more complex architectures. Also for simplicity reasons the text and data segments start at zero addresses in their respective cache, making no relocation of code needed [12]. All these simplifications take a toll on the advantages of using this simulator. Its design is too focused on one goal making it hard to expand or adapt to different situations.

Due to time constraints of the classes, instead of asking students to write an assembler one was provided. For simplicity purposes, only a limited amount of instructions were accepted. ADD and SUB were the only arithmetic instructions, AND and OR the only logical bitwise operations, SLT
the only register comparison, LW and SW the load and store instructions, BEQ the only branch instruction, and the unconditional jump J. Even with a small subset of instructions like this one, it is still possible to write programs that calculate Fibonacci series, sorting algorithms, or even calculating perfect squares [12].

**ProcessorSim**  ProcessorSim [22], also known as ProcSim, is a MIPS 32-bit R2000 Single-Cycle processor simulator developed by James Garton for his master's degree in Software Engineering. It was designed so that the simulation of a processor's internal circuits, components and buses, while executing assembly code is viewed as an animation. It is mainly used to visually understand how a CPU works so it is a helpful educational tool for undergraduates in Computer Science or even anyone who is curious on understanding computer architectures. Any assembly instruction from the supported instructions can be executed and the simulator will animate the instruction while it is being fetched, decoded, executed, and so on by the various components [22].
A differentiator of the ProcSim is a configurable datapath. ProcessorSim includes a number of pre-made MIPS datapaths that range from a simple incrementing PC and fetching an instruction up to a more complex five stage pipeline datapath. Besides the default datapath options included the user has the possibility of creating their own personalized datapaths using a XML structured file or using the built in editor. This feature allows users to experiment with different datapaths, something that other simulators mentioned above do not allow [22].

Since ProcSim is developed in Java, it is compatible with most known operating systems and it does not require installation besides the Java virtual machine [22]. Since it is focused on teaching computer architectures a main feature showing simple statistics such as number of stalls, CPI, or a count of clock cycles is missing. These simple statistics help understand some code optimization techniques or even the advantages of having a pipeline. Like WebMIPS, ProcSim does not allow to demonstrate essential concepts taught on the Patterson’s book [33] like data forwarding.

**Summary** All the presented simulators in this section have differences between each other. All except SimpleScalar, that simulates ARM architecture, are MIPS simulators. They have their advantages but all of them lack some crucial features that are explained on the Patterson’s book [33]. Table 2.1 compares existing simulators against four criteria mentioned by Sarjoughian [36] earlier.

<table>
<thead>
<tr>
<th>Simulator</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spim</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>C++</td>
</tr>
<tr>
<td>MARS</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Java</td>
</tr>
<tr>
<td>WinMIPS64</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>C++</td>
</tr>
<tr>
<td>EduMIPS</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Java</td>
</tr>
<tr>
<td>MiniMIPS</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>C</td>
</tr>
<tr>
<td>WebMIPS</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>ASP, HTML</td>
</tr>
<tr>
<td>ProcessorSim</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Java, XML</td>
</tr>
</tbody>
</table>

*Table 2.1: Comparison of MIPS processor simulators. Referenced in Section 2.2.2.2*

### 2.2.3 Abstraction Level

When analyzing or implementing a simulator it is crucial to know and understand the different levels of abstraction of the simulator. One simulator may simulate the simple external behavior of the system, like full-system simulators, while the other may go deeper by simulating the interactions of the internal components, simulating all wires and logical gates present on the system. At the same architectural level, simulators can be implemented on different levels of abstractions depending on the purpose of the simulator [36]. There are three distinct abstraction levels. Starting
with the most abstract, the black box approach, to the Register-Transfer (RT) and finally moving to the gate level component simulation [36]. A detailed description of the three abstractions levels will be presented next.

**Black Box Simulators**  This is the most common abstraction level present for the presented examples of Section 2.2.2.2. On black box simulators the components are not seen in detail, showing only an abstract representation of each component. It is detailed enough for the necessity presented by the common MIPS simulator users so there is no need to simulate in lower levels. When simulating at lower abstraction levels the complexity of the program may be greater due to an increase on detail that needs to be simulated. For example, the ALU of a black box simulator will be just a box called ALU that performs arithmetic operations, while on lower abstraction levels the ALU is composed of several hardware components that are also simulated.

**RT Level Simulators**  Register transfer level simulators are an intermediate abstraction level. They are not as detailed as gate-level simulators and not as abstract as the black box simulators. RT level is a design which models a digital circuit in terms of digital signal flow, also referred as data, between hardware registers and the logical operations performed on those signals. This kind of abstraction is typically used on hardware description languages such as Verilog and VHDL. The only presented example of this type of abstraction level is the MiniMIPS [12] discussed in Section 2.2.2.2. Black box simulation is less CPU intensive when compared with RT level because the detail of simulation is greater. The RT level simulation can simulate and demonstrate in greater detail certain components of the system that the Black Box simulation is not capable.

**Gate-Level Simulators**  Gate-level simulation operates at the lowest, of the presented three, abstraction level of simulation. As the name points out, it simulates the hardware at the level of gates present in the CPU. Simulating at such a low level of abstraction gives the opportunity to check if the circuit still works properly after some changes, gives an accurate switching activities for each gate on the design, and may even estimate power and energy consumption of the designed hardware [27]. In this section several low level simulators will be covered.

COSMOS [16] (COmpiled Simulator for MOS Circuits) provides a fast and accurate switch-level modeling of MOS digital circuit. It accomplishes high performance by preprocessing the transistor network into functionally equivalent boolean representation. Many MOS low-level simulators operate directly on transistor level, but COSMOS doesn’t. It preprocesses the transistor network into a boolean description that captures all aspects of switch-level networks including bidirectional
transistors, stored charge, different signal strengths, and indeterminate logic values [16]. Figure 2.6 shows the structure of the COSMOS simulator. It consists of a set of C language programs, ANAMOS reads the switch-level representation of MOS, and LGGC translates the Boolean representation into a set of C language evaluation procedures. In the end, the code generated by the LGGC and the simulation kernel user interface code are compiled to generate the simulation program.

GCS is a gate-level simulator developed on top of commercial NVIDIA CUDA GP-GPU graphical devices, which allow an easy deployment on modern simulation farms [18]. The goal of this work was to develop a novel cycle-based, oblivious, compiled logic simulator. This approach enables a specialized design compilation process, which partitions a netlist, optimizes it, and maps the gates to CUDA architecture [18]. The solution presented here has clustering and gate balancing algorithms and a clever organization of the data structures to exploit the memory hierarchy of the NVIDIA devices [18]. The GCS is capable of simulating complex industrial designs while delivering an order of magnitude performance speed up when compared with the state-of-the-art [18].

The next presented simulator targets the MIPS ISA used on the widely used Patterson book [33] using a FPGA. Field Programmable Gate Array is an integrated circuit that can be reconfigured after manufacture [35]. This simulator implements a single-cycle and pipeline representation of the MIPS processor onto an FPGA. All the initial design was written on VHDL using the Altera UP2 Development Board [35]. All the components such as the Instruction Fetch Unit, Instruction Decode Unit, Control Unit, Execution Unit, and Data Memory Unit are all implemented in VHDL describing all the components that make up each unit. Even though this simulator follows the Patterson book [33] the main problem with this solution is a lack of a user interface, which can bring difficulties to students on understanding the main core.

From the presented examples of gate-level simulators, all of them are mainly used for industrial development and testing of new CPUs or specific hardware components. It is very complex and strict and rarely used to teach pipeline concepts (except [35]). Since the goal of this work is to develop an aiding tool that follows as closely as possible the Patterson book [33], the gate-level abstraction was disposed as a viable solution to implement the proposed solution mainly because none present a user friendly interface, and in some cases special hardware components have to be bought by the universities.
2.2. RELATED WORK

2.2.4 Graphical Interfaces

This section analyses graphical interfaces from different MIPS advanced micro-architecture simulators. Upon analyzing the simulators, these were the chosen ones because of the similarities to the proposed prototype.

Importance of Graphical Interface Pipelining instructions on the CPU has been a standard practice since the 60’s when the IBM 360/91 was announced [20]. Forty seven years later, pipelining instructions on the pipeline is used extensively. Therefore, for students taking computer architecture courses understanding and mastering this design technique is crucial.

Although the idea of pipelining is conceptually simple, students at Chalmers University of Technology found it difficult to visualize [41]. Professors found out that graphical tools that can show that parts of several instructions executed in parallel aided their students on learning, understanding and visualizing the pipelining concept.

Introducing an interface can be a good helping aid, yet there are some risks associated with it. According to this paper [42], research on static graphics shows that only a carefully studied and designed graphical interface proves to be beneficial for students. Effective graphics conform to the Congruence Principle, which states that the content and format of the graphical aid need to match those of the concepts that are being conveyed [42].

Pipeline The pipeline diagram is the central piece to understanding the fundamentals of processor pipelining. However, only some of the referenced micro-architecture simulators in Section 2.2.2.2 contain a graphical representation of the MIPS pipeline. MARS [45], MiniMIPS [12], ProcessorSim [22], and Spim [30] do not contain a graphical representation of the pipeline. On the other hand we have WinDLX, WinMIPS64 [38], EduMIPS64 [17], and WebMIPS [14] that display a pipeline diagram. On those where a diagram is shown, there is a difference in the diagrams.
WinDLX, WinMIPS64 [38] and EduMIPS64 [17] display a simplified version of the pipeline. WebMIPS [14] is the only simulator presented earlier to have a detailed graphical representation of almost every component that compose a pipeline. Figure 2.7 (b) shows the simplified version of the CPU’s pipeline presented on WinMIPS64 and EduMIPS64 while Figure 2.7 (a) shows the detailed version seen on WebMIPS.

**Code** All the simulators mentioned in Section 2.2.2.2 contain a visual component that displays the complete program’s code to the user. The instructions being executed in a program are displayed to the user in the code window/section. The key features to point out on this section are the way that it is done and the features available to the user.

For simulators like MARS [45] and Spim [30] that do not display the pipeline diagram discussed above, the code section works much like a normal IDE does. The code window is mainly used for debugging purposes such as adding or removing breakpoints. On WinDLX, WinMIPS64 [38], EduMIPS64 [17], and WebMIPS [14] the user is allowed to execute the program step-by-step but has no possibility of adding breakpoints. This lack implies that to get to a certain code section the user is obliged to single step manually to the desired section.

Regarding WinDLX, WinMIPS64 [38], EduMIPS64 [17], and WebMIPS [14] all have a color scheme for the pipeline. Each stage of the pipeline is identified with a color, and when the step-by-step program execution is used the code panel highlights the instructions with the same color scheme to visually demonstrate where on the pipeline a specific instruction is.

**Cycles** Teaching students how the pipeline optimizes the execution of a program on the CPU is a hard task. In the Patterson’s book [33] the laundry analogy is used to explain the pipeline concept. In this analogy four people have to do their laundry. The top part of Figure 2.8 exemplifies the time taken for the four individuals to do their laundry one at a time. But, if a deeper analysis is made it can be easily optimized so that when the first person finishes using the washer, the second person can already start using the washer while the first person moves on to the dryer. Using this technique it is visually simple to understand that the bottom option of Figure 2.8 is much more time and resource usage efficient. This analogy can then be applied to the pipeline concept to aid students understand how it optimizes the execution time.

Except for the EduMIPS64 [17] family, all the simulators presented in Section 2.2.2.2 present a graphical representation similar to the laundry analogy shown in Figure 2.8. As pointed out earlier the WinMIPS64 and the EduMIPS64 simulate different architectures then the one on the Patterson’s book [33].
2.2. RELATED WORK

Figure 2.7: (a): WebMIPS pipeline diagram. (b): WinDLX, WinMIPS64 and pipeline diagrams.
Figure 2.8: The laundry analogy for pipelining. Image taken from the Pattersons book [33]

Figure 2.9: WinMIPS64 cycle panel.
2.3. **SUMMARY**

**Statistics**  The statistics panel is used by the simulators to show several types of program and execution statistics. All the presented simulators of Section 2.2.2.2 that focus on the MIPS architecture display to the user some kind of statistics regarding the program code and its execution. These statistics include: program size, number of instruction, cycles per instruction, total cycle count, and number of different stalls. During step-by-step execution, it is possible to notice how long a stall took.

**Registers and Memory**  All simulators described in Section 2.2.2.2 present an editable register and memory panel. This simple panel displays the current content of the processors registers and the system’s memory and allows users to manually change their values. Being able to manually change the values of the memory and registers is a good debugging tool. On those simulators it is also possible to change the number representation from hexadecimal to decimal, some in the property window, others by double right or left clicking the value. Even though it is a simple and not interactive windows/sections, it is an essential feature to have for a quick debug of the code. It even resembles the more advanced IDEs like Eclipse or Visual Studio that on debug mode have a small window to show the current values present in variables.

### 2.3 Summary

This chapter starts by presenting and explaining several crucial concepts on the thesis subject. It starts by explaining the MIPS ISA, moves to MIPS instruction types, MIPS pipeline, and hazards. The related work is then presented in Section 2.2. The next chapter will introduce the proposed solution for this thesis.
Chapter 3

Proposed Solution

This chapter presents the proposed solution for this thesis. Decisions and specific aspects will be detailed and explained. It starts with the specification of the MIPSter32 architecture in Section 3.1. Other important aspects of the proposed solution, such as the chosen programming language (Section 3.3) and additional documentation (Section 3.6), will be discussed.

3.1 MIPSter32 Architecture

As mentioned in Section 1, the proposed work includes a simulator that is both modular and extensible. This section presents a thorough explanation of all the essential components that make this architecture practicable.

To accomplish a modular and extensible design, patterns have to be found, making it possible to modulate the system. The main motivation of a modular and extensible design relies on the fact that future implementations of different pipeline architectures or implementation of new modules such as memory or I/O devices, may be implemented without changing the original source code. The most simple and forward approach would be to look at each individual pipeline component and implement each one separately. The system would work but no flexibility would be provided. Any change would require extensive amounts of reconfigurations and re-implementations.

Looking closer to a simple pipelined MIPS datapath patterns are found. To start off, some components of the pipeline can be reused. Examples include multiplexers, adders, and pipelines. The first level of abstraction is very straightforward, and to keep up climbing to superior layers of abstraction a closer examination has to take place. All modules, at its essence, have two main roles:
• **Combinatorial Role**: The main goal of these components is to read the values at its input and place on the output its calculated value. An adder will grab the values placed at the input, add the values together, and place at the output the calculated added value. For the MUXs, the output value will be chosen depending on the selector value.

• **Register Role**: The main purpose of these components is to place at the output the memorized value from the last clock cycle. Examples include the program counter as well as the pipeline. At the end of each clock cycle, the value present at the input is saved and placed at the output.

Looking at the entire scope, the pipelined datapath has five stages as explained in Section 2.1.4. All stages of the pipeline have a similar behavior. In each clock cycle, all the components have to be evaluated and the calculated output saved on the corresponding pipeline registers. Hence a new abstraction is placed on the system, that can be reused for the entire CPU. Taking this abstraction approach, the CPU is a component that has several stages that at each clock cycle will evaluate each component in order.

When figuring out the most efficient and viable software architecture a top-down approach was used. After analyzing each individual component, abstractions were added to the software architecture until the entire system could be modeled using the specified abstractions. In the following sections all the components will be carefully explained.

### 3.1.1 Module

Module is at the top of the abstraction level. As shown in Figure 3.10 its just an abstract class that defines the mandatory behavior of a module. All modules have an instance of a Wire (this will be explained latter on this chapter) where the output result is placed. The input wires are not defined, because, contrary to the output, different modules may have varying inputs, whereas all simulated components may only have one output\(^1\). The Module class also defines the one mandatory method that is called to return the calculated output of the module.

The other variables and methods are not directly used for the simulation. For debugging purposes, modules have a string and a method that dumps to the console the inputs and outputs at the current cycle that the method is called. The string is used as a tag to distinguish the different dumped information about the modules. The last method is used to connect the simulation to a GUI. Each module implements a way to display its information to be used on the GUI.

As explained earlier in Section 3.1, there are two main roles that a hardware component can

\(^1\)This is a Java limitation. The ALU has two outputs, but the output wire will return an array containing both outputs
assume, the combinatorial and the register roles. The structural differences translate to three classes that extend from Module and implement specific behaviors that are not specified on the Module class. These differences will be explained in greater detail next.

**ModuleC:** ModuleC is an abstract class that represents the combinatorial module described earlier. Examples such as adders, multiplexers, ALUs, shifters, among others fall into this category. The behavior described by this class is the same as the one mentioned on the module, but for coherence purposes a separate class was created to better distinguish module’s parent classes. The combinatorial modules read the input and place at the output the calculated value. Unlike other modules no internal memory is used, because there is no need to store any values. The output is solely dictated by the values presented at the input. Even though the abstract behavior is very simplistic, the resulting components may perform very complex tasks such as detecting hazards along the pipeline.

**ModuleR:** ModuleR is the abstraction class for register modules. Unlike ModuleC, these have an internal memory, which consequently change the expected behavior. At each clock cycle the value at the input is stored, so regardless of the input signal changing, the output will always remain constant for one clock cycle. At the end of the clock cycle, the stored value is updated to what ever value is at the input of the module.

Before describing the changes of ModuleR, it is important to discuss the clock edge in detail. As shown on Figure 3.11, the cycle is divided into ascending and descending. The first architectural design did not take this fact into account, making it impossible to read and write values in the same clock cycle. The majority of the register modules execute on the ascending edge, while only one module executes on the descending edge. The Registers component, described after in Section 4.3.4 Figure 4.21, is the only one to execute on the descending edge. It caused a faulty execution because, to write the value to the correct register, the MEM/WB pipeline values have
to be updated before the Registers component reads the values to store. To make this possible, the introduction of the ascending and descending edge had to be introduced to the ModuleR abstraction.

To modulate this behavior a couple of methods have to be implemented. The first method computes the input and stores it as the out value. The other method places the computed value at the output of the module. Both methods have the information of which edge is being executed. New variables have also been introduced, making it possible to store the necessary values in between cycles. The edge information comes from the ModuleS where the ModuleR is located. This will be explained next.

**ModuleS:** Unlike the previous modules, ModuleS is not a direct hardware abstraction. The 'S' stands for super, in the sense that it encapsulates the other two modules in one, translating in a component that is capable of performing combinatorial operations and at the same time behave like a register. The goal is to modulate more complex hardware behavior.

A ModuleS orchestrates a combination of modules that are both combinatorial and registers. To achieve this goal some new features had to be introduced. The most notorious change when compared to the other two types, is an array list of combinatorial and register modules. In the class constructor, all the necessary modules are initialized, and stored in the respective arrays. The execution behavior follows the same concept as a ModuleR, explained above.

**3.1.2 Wire**

When developing hardware, all the components are connected using wires to transmit information from one end to another. In a processor, a wire is used mainly to connect all the components to each other, making it possible to pass information from one module to another. To mimic this behavior, an abstract class called Wire was created. Even though it is a simple concept, without it the architecture will not function.

The Wire class has one associated module and two methods. One method sets the corresponding input module, where the other returns the output value for the associated module in. The Wire
class only knows where to fetch the desired output to place at the other end. In the next chapter will be a detailed description of how this software base architecture was used to implement the MIPS processor.

3.2 Graphical User Interface

As mentioned above in Section 2.2.4, when properly designed, a graphical interface can be very helpful. To help figure out what users expect from a graphical interface we conducted a survey among the students of computer architecture class at Instituto Superior Técnico to figure out what graphical components could be enhanced and improved. The main goal of this section is to aid students understand the pipeline using a graphical representation. The surveys’ methodology will be presented next.

3.2.1 User Survey

All the questions were targeting the WinMIPS64 because it was the simulator used in class. The main goal of the survey was to determine what graphical aids present on the WinMIPS64 that students found out to be helpful and where changes could be applied to improve the user experience of the simulator. The questionnaire was answered by a total of sixteen people/groups over a period of ten days. The complete survey can be seen in the Appendix C. Below are the main conclusions from the survey results:

- Cycle panel got a 5.7 out of 6 possible points, being considered as very helpful by the questioned students. It is a very important panel because it displays the program execution evolution.
- Register panel got 4.2 out of 6, being considered as helpful by the questioned students. Even though it has a lower score when compared to the cycle panel, the register panel is crucial to the MIPSter32 and WinMIPS64 because it displays the state of the registers in any given clock cycle. It also allows users to edit the registers values before, during, and after the programs execution.
- Statistics panel got a 5.5 out of 6, being considered as very helpful. It is a crucial element to the simulator because it displays the programs execution statistics.
- Pipeline panel got 4.3 out of 6, being considered as helpful by the questioned students. On this question we got mixed responses, and when asked if the students understood the
CHAPTER 3. PROPOSED SOLUTION

• Code panel got 4.7 out of 6, being considered as helpful by the questioned students. By highlighting the code lines with the same color scheme as the pipeline and cycle panels, the code panel can quickly display to the user where each instruction is being executed on the pipeline.

3.2.2 GUI Design

Figure 3.12 is the implemented MIPSter32 GUI. After analyzing the questionnaire results, the MIPSter32 GUI was decided to contain the following:

• 1) The main simulator menu, where users may perform several operations such as loading the programs code, hide or show panels, among others.

• 2) The shortcut button bar grants users quick access to important operations that are usually performed multiple times during the programs execution.

• 3) The pipeline panel demonstrates the five stages of the pipeline and has a color fill when an instruction is being executed in that stage.

• 4) The cycle panel which demonstrates the program execution along the clock cycles.
5) The register panel shows the content present in the simulator’s registers. These values may be changed before, during, and after the execution by double clicking on the pretended register.

6) The memory panel shows the content present in the simulator’s data memory. By double clicking on a specific memory index the user may change its content, and by double clicking on an empty space the user may add new values to the data memory.

7) The statistics panel shows to the user some performance statistics in real time.

8) The program panel shows the entire program that is being executed, and highlights the specific instructions that are being with the corresponding color of the pipeline panel.

Pipeline As referred earlier, the concept of pipelining is a crucial part of computer architecture classes. This is also a theme where students usually have greater difficulties understanding without a visual aid or an analogy. The Computer Organization and design book [33] gives a great laundry analogy explained earlier in Figure 2.8. In the same chapter, the MIPS 32-bit pipelined processor architecture figure is also explained in detail. The combination of the two gives students a clear idea of how pipelining in the 32-bit MIPS works.

It will be designed to have the greatest positive impact on the students learning of this concept. When asked on the questionnaire if they (the students) found the Pipeline panel in Figure 2.7A helpful to the understanding of the concept, on the scale from one to six 75% answered value three or above. When asked if they understood the diagram, they all answered yes, but a couple of students detailed their answer by saying yes but only because they had previously came in contact with a similar figure. One student even detailed “I also understand that those "parallel" stages in the middle are exclusive, meaning that the execution stage can be a multiplication (sub-divided in 7 sub-stages), an addition (4 sub-stages), a division or some other function, depending on the instruction, and that each one of them requires different hardware”.

The pipeline diagram, for these students, seems to be a positive aspect that the WinMIPS64 provides. However, as shown in Figure 2.7, we can consider two ways to show the pipeline. A “black-box” styled diagram or the complete hardware diagram. When asked which one they preferred, 81% of the students chose option A of the Figure 2.7, positive comments also came in favor of option B of the Figure 2.7. Students pointed out that even though option A of Figure 2.7 is easier to comprehend at the beginning, option B of Figure 2.7 is better for a deeper understanding of concepts such as the control unit. One student even pointed out that having both options would be the ideal situation. No simulator presented has the possibility of choosing between options, offering only one or the other.
After analyzing several state-of-the-art MIPS simulators and having the goal to complement the Patterson's book [33], including a pipeline diagram is essential for the proposed solution. Along with the pipeline diagram a cycle panel like the one of Figure 2.9 has proven to help students understand the pipelining concept. This panel, when asked to the students on the survey, was the only one to have 100% of the answers to be more or equal to five on a scale that goes up to six. This fact could be explained with the similarities that can be found over Figure 2.8 and Figure 2.9. This simple panel is so similar to the book [33] that students immediately understand it's meaning. This is probably one of the most helpful and simple features that most of simulators ignored. MIPSter32 will include a cycle panel and a "black-box" styled pipelined diagram. On the pipeline diagram will be very similar to the one present on WinMIPS64 and the EduMIPS64 with an extra added feature. Under each pipeline stage a text box will be included to display which instruction is being executed. This feature provides a fast access to students of the instruction’s path on the pipeline. A color scheme will also be implemented for each pipeline stage that will correlate both diagrams and the program's code explained on Section 3.2.2.

**Code**  
Even though the code section could just be a simple window where all the instructions are displayed to the user, the MIPSter32 takes the most advantage of this panel. To make it as interactive as possible, a color highlight corresponding to each different pipeline stage shows as a highlight on the instruction. The color scheme is the same as the one pointed out in Section 3.2.2 giving the user an easy localization of the instruction inside the pipeline just by looking at the code panel. The code window also shows the instruction hexadecimal value. When asked, 94% of the students think the code window on the WinMIPS64 [38] to be very helpful. When adding the color scheme it is an excellent method to show where an instruction is being executed on the pipeline quickly and intuitively. This way this section offers the user different valuable information in a quick and organized way.

**Statistics**  
The statistics section is where all the program's information and execution details will be presented to the user. It will group information about the program's size and total number of instructions, as well as the cycle count, cycles per instruction, structural, branch taken, and branch miss prediction stalls count. These statistics are the ones that every MIPS simulator presented on Section 2.2.2.2 offer to the user. This window is design in an easy way for the user to see specific statistics, facilitating the user's analysis of the execution.

**Registers and Memory**  
On the MIPSter32 it displays thirty two registers to the user. Since the base architecture does not support floating points, no panel window will be necessary to display
their value. The memory window shows the user the complete memory divided into blocks and its value. Both display windows show the content in hexadecimal, but an option to display in decimal is also available to the user. Both windows allow the user to manually change the values, since it seems to be a norm in all the studied MIPS simulators.

**Panels** The goal of having panels to display information to the user adds interface flexibility. By having the possibility of moving and resizing panels allows the user to adapt the interface to his or her needs. This is a helpful feature that is only present in WinDLX, WinMIPS64, and EduMIPS64.

**Specific Features** To guide students’ learning of computer architecture, we include specific features that some simulators like MARS [45] do not offer. MIPSter32 includes options for toggling data forwarding and delay slot. It will also enable the user to choose between branch taken, not taken, and none of the branching techniques. With MIPSter32, users will be able to change some architectural components, such as branch prediction methods, memory access latencies, among others, of the simulation to easily see the difference between certain concepts explained in the referenced book [33].

### 3.3 Programing Language

Deciding which programming language to adopt is a crucial decision for the implementation. Because each programing language has its own features, advantages and disadvantages, a close examination is very important. In particular, factors such as portability and efficiency need to be taken into account for an appropriate decision.

MIPSter32 is an extensible and modular simulator, making an object oriented language as JAVA very appealing. In addition, Java is multi-platform compatible. As long as the host machine executes a Java VM it is possible to run a Java program. Having the educational goal in mind, and having bachelor students as the main users of the proposed simulator the first language that comes to mind is Java. It is a very popular programming language and familiar to almost every student of computer science. In the presented MIPS simulators the only language that is as portable as Java is the HTML with ASP. However, this alternative has some crucial drawbacks, since the execution takes place on a remote server some precautions have to be made. Giving the example of WebMIPS [14] it is limited to 1000 cycles to avoid erroneous code submitted by users. Every presented MIPS simulator implemented with Java does not have this limitation because it executes on the local machine.
Plausible alternatives to Java include C and C++, which are faster and more efficient than Java. However, they introduce a portability problem. The user interface of C and C++ would have to be done using external libraries, while in Java offers Swing. Both languages have to be compiled for different computer architectures. The educational focus of the project makes portability more desirable than performance.

3.4 Unit Tests

The implementation includes tests to guarantee correctness of the execution. These tests were performed and analyzed to ensure a great end product, using unit tests in Java, a common tool known as JUnit [4]. Each independent test produces an output that are compared to an expected result from a trustworthy simulator. If the assert passes then the simulator is producing a correct result. Otherwise there is an error that needs to be fixed.

The unitary tests are divided into two main categories. Architectural tests and graphical consistency tests. The architectural tests were developed and designed to assure the correct output from the simulator. Graphical unit tests guarantee that all the conversions displayed on the graphical interface are being well done. Aspects such as hexadecimal to decimal values, converting binary instructions to user friendly strings such as addi r2, r3, 15 among others are assured with the graphical unitary tests.

The implemented unit tests consist on a Java class that initializes the CPU to the desired state. After, specific instructions are executed and the data memory as well as the registers values are compared with the expected result. The unitary test passes if the expected result matches the output produced by MIPSter32.

3.5 Used MIPS Assembler

The presented thesis scope does not include the implementation of an assembler. QtSpim was the chosen MIPS simulator because it incorporates a 32-bit assembler and runs in Linux, Mac, and Windows. To run sample code on MIPSter32, assembly code was written and then compiled on QtSpim.

When implementing branch instructions, an erroneous execution took place. After a great amount of debugging and analysis, the program was executing incorrectly because the Program Counter (PC) was being propagated already incremented by four (the same behavior as described in the
Patterson book), while QtSpim does not propagate the PC value incremented. To deal with this problem, when a branch instruction is compiled, the branch address has to be manually changed so that the program executes correctly.

3.6 Documentation and User Manual

As an open source, modular, and scalable project a good documentation is essential to take full advantage of these features. If the user is unable to understand the code it will be nearly impossible to change or add new features. Along with the source code a developer’s guide and Javadoc [3] will be given to aid in the development of extra features.

For regular users a user manual will be provided to help and guide users on a correct simulator usage. It will also contain detailed information of all the different panels and possible configurations. Having good documentation and user manual will contribute for an overall friendly experience.

3.7 Summary

Chapter 3 covered several key aspects of the proposed solution. Section 3.1 talks about the implemented MIPSter32 architecture. Main GUI decisions are explained and detailed in Section 3.2. Section 3.3 details and explains crucial decisions regarding the chosen programming language. Section 3.4 explains the importance of the implemented unitary tests. Section 3.5 details the used MIPS assembler. Finally, in Section 3.6 introduced the importance that documentation is for an open source project. In the next chapter the implementation of MIPSter32 will be detailed and explained.
Chapter 4

Implementation

Chapter 4 presents the implementation of the proposed solution described in Chapter 3. Specific implemented hardware components such as modules (Section 4.2), memory (Section 4.2.8), wires (Section 4.2.7), and CPU (Section 4.2.6) will be explained and detailed. In this chapter will also be contemplated how the GUI was implemented and incorporated with the hardware simulation (Section 4.4).

4.1 Implementation Overview

MIPS architecture has several variants. These variants are usually used in simulation to teach or demonstrate how several techniques improve the overall performance of the processor. Some reduce the clock cycle time, others the CPI. By adding or removing some components that implement these performance enhancers it is possible to demonstrate several aspects of how and why it is important to have them. For example, when comparing the execution of a program with and with no forwarding, it is clear that the one with no forwarding will take more cycles to execute than the one with forwarding, because less NOPs\(^1\) or stalls have to be added to the execution that prevent data hazards.

Four different MIPS architectures were implemented using the hardware building blocks described in greater detail in Section 4.2. The first implementation consists of a simple pipelined MIPS architecture that had several limitations, such as an incomplete ISA and had no control hazard. Due to these flaws, the execution of some programs was incorrect or impossible to run on the simulator. The main purpose of this implemented architecture was to have a simple pipelined

\(^1\)short for No Operation
MIPS CPU to understand if the designed solution works as a whole and have a solid base to implement the remaining features.

The second implemented architecture adds the jump instruction to the supported ISA. The third implementation adds a hazard detection unit that inserts NOPs in the pipeline when a data hazard is detected. The fourth and final implemented architecture, like the second implementation, only adds the BNE (branch not equal) instruction. All these different implementations will be further explained and detailed in Section 4.3.

To aid current and future users to use and develop MIPSter32, a documentation was developed. When proposing a modular and extensible design the correct tools also have to be created ensuring future development of the project. In this section the two types of documentation will be presented and explained.

**Developer Manual**  The developer manual focuses on aiding developers implement new features and modifying the current implementation to better fit their specific needs. It includes a detailed explanation of the project's structure. It also specifies the packages where developers may add new Modules depending on its characteristic. Classes and interfaces that should not be modified are also pointed out. Appendix A along with the project's source code and Javadoc are the necessary tools for a developer to start modifying and extending MIPSter32 simulator.

**User Manual**  Is a simple document that contains the necessary instructions on how to use MIPSter32 simulator. As it can be seen on Appendix B the user manual includes an overview of the menus and buttons that describe and explain each button and menu action. Also includes an overview of the displayed panels and what their goal is.

### 4.2 Hardware Building Blocks

Up to this point, a couple of key aspects have been discussed. A base architecture for the proposed solution has been defined and a simple but powerful abstraction level has been achieved. With a limited amount of methods and base classes complex architectures can be achieved. In this section all the steps taken to reproduce the proposed solution will be discussed and explained.
4.2. HARDWARE BUILDING BLOCKS

4.2.1 Hardware Simulation Components

As described earlier the proposed MIPS architecture is a five stage pipelined processor. In this section the implementation steps taken to reproduce the proposed MIPS architecture shown on Figure 4.21 with the specified software architecture mentioned on Section 3.1 will be detailed and explained.

4.2.2 Module

Module is the parent abstract class. It defines the base desired behavior for all modules. To accomplish this, some variables and methods are defined in order to impose a specific architecture to classes that implement Module.

- **wire\_output**: Represents the output wire where the calculated output from the module will be fetched. It is declared as a Wire, but depending on the module may implement any Wire subclasses.

- **dumpTag**: A string that is used only for debug purposes. This variable is defined by the specific module's constructor and only used as a tag to identify the specific module for debug print.

- **getOutput()**: Abstract method used by the Wire class to retrieve the output for the Module class. The returned value is from the class Object, to make the method's output as generic and flexible as possible.

- **dumpToSys0()**: Abstract method to dump the module's state to the system out. As the dumpTag, the dumpToSys0 method is used only for debug purposes.

- **dumpToGUI()**: Abstract method used to convert the Module information for the GUI. This method is only implemented if the specific module has information to display on the MIP-Ster32 GUI.

4.2.3 Combinatorial Modules

Combinatorial modules, represented by the abstract class ModuleC, have a simple behavior of placing at the output the calculated input values. Figure 4.13 demonstrates all ModuleC components present in the proposed MIPS architecture.
ModuleC implements all methods and inherits all variables defined on the parent class Module(Section 4.2.2). Each class presented on Figure 4.13 may have more than one instance present on the simulator.

- **Add**: Simulates the hardware Adder. This class constructor receives a dumpTag string, and three wires. In the proposed MIPS architecture adders only have two inputs, so two of the three wires are the input values to add, and the other is for the output. The getOutput method verifies that the input is valid and returns its sum. The dumpToSyso method prints to the System output the return value identified by the dumpTag.

- **ALU**: Simulates the arithmetic logic unit of the processor. The constructor for this class has the dumpTag string, and four wires. Two wires for the input values to process, one for the ALU control line (mentioned on table4.2), and one to place the output value. The getOutput method checks the ALU function wire to decide what operation should be applied to the input values. Like the Add module, the ALU does not implement the dumpToGUI method, because there is no graphical component that requires to get specific information from the ALU. Even though the ALU is presented as a ModuleC, it could be implemented as a ModuleS composed with the necessary hardware components.

- **AluControl**: Simulates the necessary hardware that controls the ALU. As the ALU, mentioned above, the AluControl is implemented as a ModuleC, it could be represented as a ModuleS. The constructor receives as input variables the dumpTag and five wires. The dumpTag is for the same purpose as mentioned on the above ModuleC classes and the five wires are as follows:
  - **aluFunctionWire.in**: This wire contains the ALU function retrieved from the instruction’s byte code. It represents the function field column on table 4.3.
  - **aluOp1Wire.in**: This wire represents the left bit of the ALUOp’s two bits that are represented as ALUOp on table 4.3.
  - **aluOp0Wire.in**: This wire represents the right bit of the ALUOp’s two bits that are represented as ALUOp on table 4.3.
  - **opCodeWire.in**: This wire represents the the op code from the instruction.
  - **wire.out**: The wire where the AluControl places the returned value of calling the getOutput() method.

The aluOP(ALU Operation) wires are the first juncture that will decide what output will be produced. The second decision filter will be decided by the opCodeWire. Based on this
information the correct ALU control input (shown on table 4.3) will be calculated and placed at the output of the module's wire_out.

- **And**: Simulates the hardware AND gate. AND ModuleC has a simple constructor that receives the dumpTag, two wires containing the input values, and an instance of a Wire object to place the output.

- **Control**: The ModuleC that represents the pipeline control unit shown on Figure 4.20. The constructor has as input the dumpTag, a wire_in containing the instruction OP code, and a wire_out to place the output result. Depending on the operation code, the nine control flags will be correctly defined. They are as follows:
  
  - **regDst**: The flag will be the first MUX selector on Execute pipeline stage that will determine if the write register is from the 20-16 or 15-11 bits from the instruction's byte code.
  
  - **aluOp1 and aluOp0**: Represent the two bits that form the ALUOp column on table 4.3.
  
  - **aluSrc**: The flag that is the second MUX selector that determines if the ALU's second input value is from the Register Module or if it is the instruction's first fifteen bits.
  
  - **branch and bne**: These flags are set to true if the instruction is a BEQ or BNE respectively.
  
  - **memRead and memWrite**: If the instruction is a read operation the memRead flag is set to true. If the instruction is a write operation then the memWrite flag is set to true.
  
  - **memToReg**: Is the Write Back stage's MUX selector. It determines what data is written back on the Register Module.

- **DataMemory**: Since a memory simulation is out of scope for this project, the DataMemory is represented as a ModuleC. It’s constructor has as input the dumpTag and five wires that are as follows:

  - **addressWire**: The wire where address to read from or write to is placed.
  
  - **writeDataWire**: Has the data to be written in memory.
  
  - **memReadWire**: The control flag for read from memory instructions.
  
  - **memWriteWire**: The control flag for write to memory instructions.
  
  - **wire_out**: The wire where the DataMemory places the returned value of calling the getOutput() method.
• **HazardDetectionUnit**: Represents the hardware that implements the pipeline’s hazard detection unit. It is implemented as a ModuleC but it could also be modulated using a ModuleS. The goal of this module resides on detecting hazards that may occur in the pipeline and insert the necessary bubbles to prevent them from happening. To achieve this the module receives as input variables on its constructor, the dumpTag eight wires that are as follows:

  - **instructionWire**: The wire containing the instruction that is being executed on the Instruction Decode stage. It will be used to compare the instruction’s registers with the registers that are being used along the rest of the pipeline. By comparing this information data hazards are detected. If this is the case a NOP will be introduced in the pipeline to prevent the hazard to occur.
  
  - **EX.Wire**: Contains the write register information of the instruction that is being executed on the Execute Stage.
  
  - **MEM.Wire**: Contains the write register information of the instruction that is being executed on the Memory Stage.
  
  - **WB.Wire**: Contains the write register information of the instruction that is being executed on the Write Back Stage.
  
  - **memJumpWire**: The control wire from the Memory stage for J instructions.
  
  - **memBranchWire**: The control wire from the Memory stage for branch instructions.
  
  - **wire.out**: The wire where the DataMemory places the returned value of calling the `getOutput()` method.

By comparing the current instruction with the extra information retrieved from other stages, the hazard detection unit is capable of handling the possible pipeline hazards.

• **InstructionMemory**: Since a memory simulation is out of scope for this project, the InstructionMemory is represented as a ModuleC. It’s constructor has as input the dumpTag, the `wire_in` containing the program counter(PC) value, and the `wire_out`. Depending on the input, the InstructionMemory ModuleC will place at its output the next instruction to be executed.

• **Or**: Simulates the hardware OR gate. OR ModuleC has a simple constructor that receives the dumpTag, two wires containing the input values, and an instance of a Wire object to place the output.

• **ShiftLeft**: As the name suggests, this ModuleC shifts left x bits to the left. ShiftLeft’s constructor has as input the dumpTag, a shift amount, the input value to be shifted left, and
4.2. HARDWARE BUILDING BLOCKS

Figure 4.13: Combinatorial module UML diagram.

a wire_out.

- **SignExtend**: Simple hardware component that extends the sign of a given input bits. This ModuleC has a simple constructor that receives the dumpTag, wire_in containing the value to be sign extended, and an instance of a Wire object to place the output.

4.2.4 Register Modules

Register modules, represented by the abstract class ModuleR, have a simple behavior of placing at the output the stored values. Figure 4.14 demonstrates all ModuleR components present in the proposed MIPS architecture.

ModuleR, just as ModuleC, implements all methods and inherits all variables defined on the parent class Module(Section 4.2.2). Each class presented on Figure 4.14 may have more than one instance present on the simulator, but in the proposed MIPS architecture this does not occur.

- **IFIDPipeline, IDEXPipeline, EXMEMPipeline, and MEMWBPipeline**: Represent the four pipelines between the five pipeline stages. Even though the behavior of all four pipelines is equal, different classes have to be declared due to the differences in output wires. The constructors receive the dumpTag, a wire_in, and all the different output wires. The evaluateEdge() function switches the out value with the new value, the computeInput() function saves the in value to memory, and the getOutput() function returns the out value.

- **ProgramCounter**: Is the ModuleR responsible for the program counter(PC). Each cycle the PC value will determine which instruction will be executed.

- **Registers**: Unlike the previous ModuleR classes, the behavior for Registers is more complex. The methods implemented on this Module are as follows:
  - `evaluateEdge()`: Writes the output value for the register.
  - `computeInput()`: Depending on the input control wires, the Registers module will perform different actions. If the regWrite control flag is true, the write data value will be
stored on the register present on the write register wire. Each cycle the Registers module also places at its output, the stored values for the registers read register one and two.

- `getOutput()`: Returns the stored output value.

- `dumpRegisters()`: Dumps the saved registers values into the system output. This function is used only for debug.

- `dumpForUnitTests()`: Dumps the saved registers value into the system output with a specific format. This function is called to validate the unit test.

- `setRegisters()`: Sets all the registers value equal to the given input. This function is mainly used by the GUI. It is also used by the unit tests to set all registers with specific values before performing the test.

- `getRegisterValues()`: Returns all the register’s values. It is mainly used by the GUI to display the current stored values to the user.

### 4.2.5 Composite Modules

Composed modules, represented by the abstract class `ModuleS`, have a complex behavior. Figure 4.15 demonstrates how different stage implementations may be incorporated to the architecture. `ModuleS` implements all methods and inherits all variables defined on the parent class `Module` (Section 4.2.2).

Unlike `ModuleR` and `ModuleC`, classes that implement `ModuleS` have more variables and methods. The changes reflect an increase of complexity of the module’s expected behavior. Since these modules may contain combinatorial and register components, the `evaluateEdge()` and `computeInput()` functions have to be implemented. Two new variables, `registerModules` and `combinatorialModules`, are array lists containing the `ModuleR` and `ModuleC` that compose the `ModuleS`. 
When executing a cycle, the ModuleS will iterate through all its modules to compute the correct output. In the proposed solution only pipeline stages are composed modules. As explained earlier other complex modules such as the ALU, the ALU control unit, or even the hazard detection unit may be implemented as a ModuleS.

To facilitate the support of different architectures, the abstract classes for each stage has been implemented. The IFStage, IDStage, EXStage, MEMStage, and WBStage classes all extend from ModuleS, but implement other methods to deal with compatibility issues regarding the GUI and unitary tests. Each architecture will have its own stage implementation.

4.2.6 CPU

The abstract class CPU is where all the hardware simulation components will be initiated and organized to form a functional architecture. Depending on the architecture that is to be simulated, a CPU class is implemented with the specific components. By implementing the CPU class allows MIPSter32 to handle different architectures with the same GUI. CPU interface also defines all the necessary methods to interact with the GUI, being the bridge that connects simulation to interface. The methods are as follows:

- **Constructor**: Receives no input and when called initializes all the necessary Modules, Wires, and memory components, connecting them accordingly.

- **setProgramFile()**: A GUI method that is called when the user selects a new file to be loaded into the simulation.

- **executeProgram()**: Executes an inputed amount of cycles on the simulator.
• `setRegisterValues()`: Alters the content of the registers. Used by the GUI when the user manually changes a value.

• `stepNext()`: Executes one cycle on the simulator.

• `getRegisterValues()`: Returns the values stored on registers to display on the user interface.

• `getTotalInstructionCount()`: Returns the instruction count of the simulation. Used in the GUI’s statistics panel.

• `getTotalCycleCount()`: Returns the total cycles performed during the execution of the program. Used in the GUI’s statistics panel.

• `drawSelf()`: Draws to the Pipeline panel a block diagram of all the stages.

The CpuBO and CpuNO are two different architecture implementations that may be switched by the user in run time. CpuNO has no implemented optimization, whereas the CpuBO has the branch decision moved from the MEM stage to the ID stage. The implemented work only has these two different architectures, therefore only two are present on the Unified Modeling Language (UML) diagram.

### 4.2.7 Wires

The Wire class represented on Figure 4.17 is an implementation of the basic wire behavior. It includes an instance of the module from where the value is retrieved. The function `setModuleIn()` connects the module with the wire, and `getValue()` returns the executed value of performing a `getOutput()` call to the instance of the connected module to the wire.
To implement the proposed MIPS architecture three more wire classes that extend from Wire class were implemented. These are as follows:

- **ArrayWire**: A wire that returns an array list.
- **WireConstant**: A wire that has a fixed value as output that is passed as input to the constructor.
- **ArrayDisjoinWire**: Returns as output a value disjointed by a defined amount of bits. The disjoint value, just like in the WireConstant, is passed as input to the constructor.

### 4.2.8 Memory

As mentioned earlier, the proposed solution’s scope does not include an accurate memory simulation and implementation. But for MIPSter32 to have a complete and correct execution memory classes had to be implemented. For simplification purposes the program and processor memory were implemented separately and the memory access is done in one cycle.

As explained before the InstructionMemory and DataMemory interfaces use the ProgramMemory and Memory classes respectively to perform the necessary read and write operations. A detail explanation of these two classes will be done next.

- **ProgramMemory**: The class follows a singleton design pattern to ensure that the memory is persistent and coherent along the execution. This design pattern guarantees that only one instance of this object is created. Following are the methods implemented by the class:
  - **Constructor**: Is private, due to the singleton design pattern. Has no input variables and when called initializes the object in the correct state.
  - **getInstance()**: Case there is no instance created already, creates and stores the new object instance. Otherwise the previously created instance is returned.
- **resetProgramMemory()**: Forces the program memory instance to its original state. It is used before loading a new program to be executed.

- **read()**: Receives as input program counter and returns that position's value.

- **write()**: Writes to the next memory position. Does not receive a position value because this is done automatically.

- **jumpToMemPos()**: Receives as input a specific address and updates the current read pointer index. Used when a change in program counter occurs by a jump or branch instruction.

- **setProgramFromFile()**: Has as input a file name, and writes to the program memory all the pre-compiled instructions on the file. It's an utility function that allows the user to easily load programs from files for execution on the simulator.

- **getTotalInstructionCount()**: Utility function that returns the total program instruction count. It is used to calculate statistics that are displayed by the GUI to the user.

- **dumpToFrame()**: Returns a formatted string to create a frame layout that displays on the GUI the program memory.

**Memory**: As the ProgramMemory, Memory follows a singleton design pattern. Besides the constructor and the getInstance() methods that are the same as ProgramMemory, here are the methods implemented by the class:

- **resetMemory()**: Forces the memory instance to its original state. It is used before loading a new program to be executed or when the execution has come to an end.

- **read()**: Receives as input program counter and returns that position's value.

- **write()**: Writes the desired value to a specific memory position.

- **dumpToFrame()**: Returns a formatted string to create a frame layout that displays on the GUI the memory content.

### 4.3 Pipelined MIPS Architectures

This section presents and explains the aspects of the proposed solution's architecture. It covers MIPS pipeline and software architectures. It first presents the proposed MIPS architecture that the software will implement, followed by the proposed software's solution that complies the modular and scalable concepts to simulate the chosen MIPS architecture.
4.3. PIPELINED MIPS ARCHITECTURES

Next will be presented chronologically all the architectures implemented during this thesis project. A detailed explanation of each individual component, along with the limitations of each presented solution will be done in order to understand the evolutionary process that lead to a final solution.

4.3.1 Simple Pipelined MIPS Architecture

To tackle the complex task of designing a pipelined MIPS CPU architecture, a simplified version with several limitations was designed. The approach taken to design the proposed solution's architecture was incremental, meaning that it starts with simple functional version, moving up the complexity along the way.

The first developed MIPS architecture is presented in Figure 4.18. This version is functional and works with its limitations. It implements almost the entire MIPS ISA \(^1\) and it is already a pipelined architecture. Following will be a detailed description of each pipeline stage of the proposed solution.

**Instruction Fetch (IF) Stage:** In this stage of the pipeline, the correct instruction is fetched to be executed. To perform this task several hardware components have to be implemented. Following is a list of the necessary components and their task on the stage.

- **Instruction Memory:** The instruction memory is a hardware component that stores in registers the instructions that compose the program. It receives as input the desired instruction’s address and returns the corresponding instruction to that address. This component

\(^1\)The pipelined MIPS architecture presented on Figure 4.18 does not implement branch not taken instructions as well as some optimizations including delay slots, branch prediction, among others.
does not belong to the CPU, and on the implemented solution is a simple interface that can later be part of a more complex memory simulator.

- **Add**: The add module is a simple arithmetic unit that adds two values placed on the input. In this case the adder receives the current PC value, increments by four and returns the value to the MUX. The output value is also stored in the IF stage’s pipeline registers to be propagated onto the next stages. In the Patterson’s book [33] the PC value is propagated with the incremented value already. In some other examples it may be possible to find that PC value is propagated without being incremented. This will change the branch calculation, because, depending on whether or not the PC value is incremented, the calculated jump address of the branch has to take this information into account, to assure that the correct jump address is calculated. The PC value is incremented by four, because a 32-bit architecture is being simulated, and therefore each memory position is eight bits.

- **PC**: The program counter is a very simple hardware component. It has one 32-bit register and at the end of each clock cycle it replaces the old PC value with the new one. The output for this module is the stored program counter value.

- **MUX**: The multiplexer on this stage has a very important role. Depending on whether or not a branch occurred it will choose the correct PC value to store for the next instruction fetch. The has as input the PC value calculated from the Add of this stage, as well as the PC value calculated from the branch, and depending on the selector wire (also part of the input values) will choose the correct value to pass onto the program counter component.

**Instruction Decode (ID) Stage**: The ID stage is responsible for decoding the fetched instruction, retrieving the necessary register values as well as set all the correct control signals. Following is an explanatory list of the hardware components that compose the ID stage:

- **Control**: The control unit has as input the instruction and places at its output all the control signals. These control signals will be saved on the ID/EX pipeline registers, and will dictate the operations that will execute on the rest of the pipeline stages.

- **Registers**: This component is the where all the register’s values are stored. As input it will receive the correct register index to read and place on its output the stored values. It will also have information of the write register index as well as the write value from the Write Back (WB) stage. Finally, as input, the control signal that will inform if the write value is in fact to store in the register bank. As output this component will have the two values stored for the register’s indexes at the input.
• **Sign-Extend:** As the name suggests, it will extend the input signal and return the correct extended result. This is used for the immediate values. In the I-Type instructions, a value is passed directly with the instruction. The 16 bits has to be extended because of the 32-bit architecture.

Besides storing all the components outputs, this stage also places the write register of both R-Type and I-Type instructions on the pipeline registers.

**Execute (EX) Stage:** Executes all the operations that involve the ALU. In the following is an explanation of the hardware components that are present in the EX stage:

• **Add:** The adder on this stage adds the incremented PC value with the jump increment of the branch instruction, resulting in the final address where to jump if the condition is met. The output result will then be saved on the EX/MEM pipeline registers.

• **Shift Left 2:** This is a simple hardware component that shifts the input value by two. The input value comes from the previous stage, referring to the low sixteen bits of the instruction. The shift is equivalent to a multiplication by four, and is necessary to retrieve the correct value for the branch jump address calculation. This value is multiplied by four because on the instruction the value is placed as the number of instructions, instead of the memory position to jump over to.

• **ALU:** The ALU has as inputs the two values to perform the arithmetic logic operation and a control signal that will dictate which operation needs to be executed. The output of this component will be the calculated result as well as a signal saying whether or not the operation's result was equal to zero. This last signal will then be used to figure out if the branch is taken or not in the next stage. Table 4.2 defines the six combinations of four control signals that are passed to the ALU by its control unit.

• **MUX 1:** The first MUX of this stage is to select the correct second operand of the ALU. It has as input the second value read from the Register's component as well as the immediate value read from the instruction on the previous stage. This MUX will choose, depending on the control flag, if the operation to be performed is from an I-Type or R-Type instruction.

• **ALU Control Unit:** The ALU's control unit will be responsible to convert the instruction operation code into the function that the ALU has to perform. Based on the two ALUOp control signals, the instruction's operation code is determined based on the combinations described in table 4.2. Table 4.3 shows in detail how this conversion is made.

---

1 The ALUOp are two signals that have been defined previously by the control unit mentioned on the ID stage.
• **MUX 2**: The second multiplexer will choose the correct write register index. It has as input the two possible write register’s indexes from either I-Type and R-Type instructions and depending on the control signal will choose the appropriate one to propagate to the next stage.

In this stage, besides the component’s outputs, the register where to write the value, and the value of the register two from the instruction are also propagated for the next stages.

**Memory Access (MEM) Stage**: The MEM stage is where the access to the data memory takes place. This stage will either write a register’s value into memory or will fetch a specific memory value to be stored on a register. It is also in this stage that the branch operation is resolved. In the following is an explanation of the hardware components that are present in the MEM stage:

• **AND**: The AND gate on the MEM stage controls the branch equals instruction. In Table 4.3 it is shown that when there is a branch equal instruction the ALU will subtract the two values, and in case the result is zero, it will set the control signal to true (indicating the values were equal). For a branch equal operation to be successful two conditions have to be true at the same time. The values have to be true, and the instruction has to be a branch equals. Therefore the AND gate with the ALU zero flag as well as the branch control signal.

• **Data Memory**: Similar to the Instruction Memory in the IF stage and the Registers component in the ID stage, the Data Memory is a register bank that saves data related information into memory. Usually the Data and the Instruction memory are the same hardware component, but for visual purposes they are displayed separately. The Data Memory receives an address and the data to write. It also has two control signals that will inform the Data Memory if it is a write or read operation. As output the Data Memory will return the stored value for a given input address.

The ALU result as well as the register where to write the value, are propagated for the next stage.

**Write-Back Stage**: The WB stage is responsible to place the result into the registers present in the ID stage. Following is an explanation of the hardware:

• **MUX**: The only hardware component present in the WB stage is this multiplexer. Its purpose is to select the appropriate value to write back to the Registers component back in the ID stage. As input, it will have the read data from the Data Memory component present in the MEM stage, and the ALU result that has been propagated since the EX stage. The control
signal will choose if the value to write back to the Registers will be from memory or from an arithmetic operation.

Besides all the above mentioned components, this architecture also has five pipeline stages. The names use to identify which pipeline is being refer to include the name of the stage to the left using a / to split from the name of the stage on the right. So for the pipeline component between the IF and the ID stage will be named as IF/ID pipeline. The pipeline hardware component is nothing more then registers that store the output of one stage and place its saved values available to the next stage.

**Architectural Overview:** The architecture presented in Figure 4.18 works but has several limitations. The main purpose behind this architecture was to have the simplest pipelined MIPS CPU to understand how everything works as a whole, figure out specific details of the individual hardware components, and have a solid base to implement the remaining functionalities.

This architecture implements almost all of the instruction set, excluding the branch not equal and jump instructions. Besides the missing instruction implementation, no hazard detection is implemented. Without being able to detect hazards, the execution of a program can produce incorrect outputs. As explained in Section 2.1.5 when a hazard occurs the pipeline has to be stalled in order to avoid an incorrect output. The example used is a data hazard, which occurs when one of the operands is dependent on the execution of one instruction’s output that has yet to finish.

The presented architecture also has an incorrect behavior when executing the branch instruction. Without the hazard detection unit, the CPU has no way of knowing when to insert stalls to let the branch instruction update the PC value that fetches the next instruction. The IF stage will be fetching the incorrect instructions until the branch is decided on the MEM stage. By the time the branch is decided, the pipeline may have two incorrect instructions running on its pipeline, because no stalls have been introduced.

Even though the pipeline supports almost all of the instruction set, most programs will have an erroneous output, due to the lack of a hazard detection mechanism. For the next architectures the several missing components will be introduced and explained.

### 4.3.2 Jump Implementation

One missing feature from the architecture presented in Section 4.3.1 is the jump instruction. To implement this instruction new hardware had to be added to some of the pipeline stages. Only
the affected stages will be mentioned in this section.

**IF Stage:** The first change required, as shown in Figure 4.19 is in the IF stage. A new multiplexer is introduced before the PC to deal with the added instruction implementation. As input it will receive the jump address and the multiplexer’s value that was previously explained. It will have a control signal that comes from the MEM stage (it comes from this stage, because, as it was explained earlier, it is where the branch decision is made). Now the new PC value is decided by the result of two connect multiplexers. This is the only required change in the IF stage.

**ID Stage:** In the ID stage only a couple of changes are required. The first includes adding a new shifter. The shifter will left shift by two the instruction’s 25 first bits. The reason for this shift by two is the same as the reason for shifting the branch value explained earlier in Section 4.3.1 MEM stage. The output value will then be propagated to the MEM stage, which will then be used by the new IF stage’s multiplexer.

**Architectural Overview:** The main hardware changes were mentioned above. Since the branch decision only takes place on the MEM stage, these new signals also have to be propagated up until there. This architecture only implements a new instruction from the MIPS ISA, so all the hazard issues in the previous architecture still apply.
4.3. PIPELINED MIPS ARCHITECTURES

4.3.3 Hazard Detection Unit Implementation

To solve the hazard problems that occur in the pipeline, a hazard detection unit needs to be implemented. Its main roles include detecting a hazard occurrence and introducing the necessary stalls on the pipeline to prevent them from happening. To do this, a new hardware component has to be added that will monitor the ID, EX, and MEM stages, and depending on the situation will act upon the IF stage and the IF/ID pipeline registers.

Hazard Detection Unit: It is a new hardware component implemented in the pipelined MIPS architecture that, as mentioned earlier, will detect the hazards, and stall the pipeline. Even though it is not the only change required to fix the hazard issue, it is the main unit added to the pipeline. As inputs the hazard detection unit has the read registers for the ID stage and write register for the EX, MEM, and WB stages that will detect any data hazard that may occur on the pipeline. Also as input, it will have the jump and branch flags to stall the pipeline until these instructions are completed. For output, the hazard detection unit will have control signals for the PC, IF/ID pipeline registers, and a multiplexer that will be explained later on this section. When a hazard is detected, the correct signals are activated, causing the pipeline to stall until the hazard is no longer in effect.

• Data Hazards handling: A data hazard (explained in Section 2.1.5) will be detected when one of the read registers on the ID stage matches one of the write register for the EX, MEM, and WB stages. When this happens the next instruction is fetched, but the pipeline will stall until the write back execution completes.

Figure 4.20: Simple Pipelined MIPS architecture with Hazard Detection Unit.
• **Jump and Branch handling:** When a branch or jump instruction is detected on the ID stage, the instruction fetched from the previous stage will be flushed, because it may be the wrong instruction. The pipeline will be stalled until the branch or jump is completed on the MEM stage.

**Multiplexer:** The other new hardware component added to the architecture is a MUX. This multiplexer has as input a No Operation (NOP) and the control output, and is controlled by the hazard detection unit. The NOP instruction control signal will only happen when a branch or jump operation is detected, because the instruction fetched from the previous stage will have to be canceled, even though it is already placed on the pipeline. In case of a data hazard, then the control signal will be chosen by the hazard detection unit because the instruction fetched on the IF stage, will have to be delayed. The instruction will only be stalled until the write back completes, and not disposed of.

### 4.3.4 Branch Not Equal Implementation

The last change to the proposed MIPS architecture is the implementation of the branch not equal instruction. The required change was implemented mainly on the MEM stage, where all the other jump and branch instructions are being calculated. The implemented architecture is demonstrated in Figure 4.21.
MEM Stage: To implement the new instruction from the MIPS ISA two new hardware components have to be added to the MEM stage.

- **BNE AND Gate:** The AND gate for the branch not equal instruction has the same behavior as the AND gate for the branch equal instruction mentioned earlier. It has two inputs, one signal that comes from the Control unit determining if it is a BNE instruction, and the ALU zero signal negated. Now the ALU signal is negated simply because the jump should only occur when the instruction is branch not equal and not ALU zero result.

- **Branch OR:** The pipeline will jump to the branch value if the branch equal or branch not equal instructions are true. So, following this logic, an OR gate has to be implemented to pass back to the ID stage the information whether or not a branch is taken.

### 4.3.5 Concluding Remarks

The presented MIPS architecture is based on the Patterson’s book [33]. Some implementations do not follow the presented architecture on the book, due to a lack of detail. The presented architecture also has no implemented optimizations, such as delay slots and forwarding. Overall the presented architecture implements a complete MIPS ISA, which is mainly used to teach students the main concepts on computer architecture. Even though no optimizations are implemented, the software architecture will be modular and extensible to allow these changes to be easily implemented in future works.

### 4.4 Graphical User Interface

The graphical user interface allows the user to interact with the simulation in a graphical environment. In this section the GUI architecture will be explained and detailed. Just like the simulation aspect, the GUI was also designed to support future extensions and/or modulation.

#### 4.4.0.1 Graphical Simulator

The graphical simulator is the main class that roosts and manages all the graphical components. The main graphical components include the menu options, a button tool bar, and the different panels that display specific information to the user. All panels, menus, and buttons are initialized and placed on the starting positions. Along with the graphical components, the simulation aspect
is also created and managed on this class. The initial architecture is instantiated and at runtime may be switched.

Unlike the other graphical components the menus are implemented directly in the graphical simulator class. The performed actions by the menu are closely related to the graphical simulator, so to simplify the implementation this decision was made. They include opening files, toggling the panels visibility, and changing architectures. All this actions are directly related to this component's tasks.

4.4.0.2 Button Bar

Is the graphical component where all buttons are placed. This is a shortcut to the most used actions by the user. It includes buttons to perform a step on the simulation or a reset action. For future implementations the developer can easily add new buttons to this class along with the necessary actions to be performed.

4.4.0.3 Panels

These graphical components are the core where information is displayed to the user. On of the main aspects on the proposed solution were panels due to the personalization capabilities that they bring to an interface. The user may choose to move, scale, hide, or show panels to customize the simulator to their specific needs. In figure 4.22 the GraphicalPanel is an abstract class that has the mandatory variables and methods that a panel must implement, and the subclasses are the implemented panels.

- **CycleFrame**: This frame shows the simulation's execution as the laundry example shown on figure 2.8. This panel is very useful because it shows in a block diagram the instructions that were executed and in what stages they were throughout the execution.

- **MemoryFrame**: Displays the content present in memory in a friendly manner. By double clicking any position the user may change or add new memory positions at runtime.

- **PipelineFrame**: A graphical panel that displays the five stages of the pipeline. When a stage is executing a specific instruction the corresponding pipeline stage gets filled with a solid color, demonstrating in a graphical way that an instruction is being executed. By comparing with the information displayed on the CycleFrame, the user rapidly identifies which instruction is being executed.
4.5 Summary

Chapter 4 started by introducing a general overview of the implementation in Section 4.1. Section 4.2 details and explains the hardware building blocks used to create all the implemented pipelined MIPS architectures explained in Section 4.3. This chapter ends by describing the implemented GUI. Chapter 5 will explain the evaluation methods.
Table 4.2: ALU Control Lines: Defines the six combinations of four control inputs of the ALU. This table was based on the Patterson’s book [33].

<table>
<thead>
<tr>
<th>ALU control lines</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>AND</td>
</tr>
<tr>
<td>0001</td>
<td>OR</td>
</tr>
<tr>
<td>0010</td>
<td>add</td>
</tr>
<tr>
<td>0110</td>
<td>subtract</td>
</tr>
<tr>
<td>0111</td>
<td>set on less than</td>
</tr>
<tr>
<td>1100</td>
<td>NOR</td>
</tr>
</tbody>
</table>

Table 4.3: Conversion Table for the ALU operation code. This table shows how to set the control inputs for the ALU using the 2-bit ALUOp control in combination with the instruction’s operation code. The XX denotes that for a specific operation the function field is ignored. This table was based from the Patterson’s book [33].

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>Instruction Operation</th>
<th>Function field</th>
<th>Desired ALU action</th>
<th>ALU control input</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Load Word</td>
<td>XX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>00</td>
<td>Store Word</td>
<td>XX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>01</td>
<td>Branch Equal</td>
<td>XX</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td>10</td>
<td>Add</td>
<td>100000</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>10</td>
<td>Subtract</td>
<td>10010</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td>10</td>
<td>AND</td>
<td>100100</td>
<td>AND</td>
<td>0000</td>
</tr>
<tr>
<td>10</td>
<td>OR</td>
<td>100101</td>
<td>OR</td>
<td>0001</td>
</tr>
<tr>
<td>10</td>
<td>Set on Less Than</td>
<td>101010</td>
<td>set on less than</td>
<td>0111</td>
</tr>
</tbody>
</table>
Chapter 5

Evaluation

In this chapter evaluation process of the presented work is described. This chapter starts by describing MIPSter32 usability in Section 5.1. Section 5.2 presents the performance tests executed to evaluate MIPSter32. In Section 5.3 the importance of implementing unit tests is discussed along with its impact in the present and future work. It also refers on how the implemented test set assures the correct simulation of the proposed MIPS ISA. Section 5.4 details an internship program done to validate the key aspects of modularity and extensibility of the MIPSter32 architectural design. It describes the design tasks assigned to two students and explained how it proves that MIPSter32 is modular and extensible.

5.1 Usability

As mentioned in Section 3.2, the GUI has multiple panels that display relevant information of the simulation. In this section is presented a small description of how the interface changes along a programs execution. Figure 5.23 shows the initial state when a user uploads compiled code to MIPSter32.

Figure 5.24 demonstrates the interface change after executing three cycles. The pipeline panel shows that the IF, ID, EX, and MEM stages are executing instructions. The code panel has the entire program code, highlighting the instructions with a corresponding color, making it simple to identify which and where the instruction is being executed. The other panels are also updated every clock cycle.
Figure 5.23: MIPSter32 initial state.

Figure 5.24: MIPSter32 state after some cycles.
5.2 Performance

Simple performance tests were made to see how the simulator behaves when pushed to the limits. The tests consisted on executing looping programs for a high amount of cycles. All tests were executed with the following computer characteristics:

- **Operating System**: Windows 7 Professional
- **Processor**: Intel Core i-5-2520M @ 2.5GHz
- **RAM**: 8GB

Table 5.4 contains the results of executing three different looping programs 1000, 5000, 10000, and 100000 clock cycles. The measurements were made automatically, therefore no human error was introduced in the measurements. These tests were made to guarantee that the simulator is capable of performing a great amount of clock cycles in a reasonable time. After analyzing the results, taking around twenty seconds to perform one hundred thousand clock cycles seems to be very reasonable, taking into account MIPSter32 is not a performance oriented simulator.

5.3 Correctness

Unit tests are designed and used to assure a correct behavior from the developed software. The presented work includes a test set that are composed of a test case for each instruction, as well as a test case for each GUI helper functions. As [49] writes, there are four testing adequacy notions.

- **Statement Coverage**: A test set that covers all the program’s statements.
- **Branch Coverage**: A test set that covers all the program’s control transfers.
- **Path Coverage**: A test set that covers all the program’s execution paths from the program’s entry to its exit.
- **Mutation Coverage**: A mutant software has artificial faults. By running the test set on the original and mutant software where unit tests fail on the mutant and pass on the original software, the test passes.

The implemented test set include forty three test cases that assure the correct execution of all the implemented MIPS instruction set as well as specific hardware components like the hazard
detection unit. It also includes thirty three test cases that assure the correct information displayed to the user. The GUI uses a helper class that translate the instruction bit code to ‘readable’ values and memory sorting function that are tested with the GUI helper test cases.

From the four adequacy notions mentioned above the architecture test cases fit the path coverage testing because an input is given to the simulator and a pass considers the final output of the execution. These tests receive a compiled MIPS assembly program, executes a given number of cycles, and the output is then compared with the expected result. Test cases include both equal and not equal asserts to guarantee that the simulator produces a correct output with the expected amount of clock cycles. For each MIPS instruction, there is a test case with the exact amount of cycles that the assert equals has to pass. There are cases where the assert equals test executes more cycles then the exact amount to assure that the simulator’s program memory is working properly. The assert not equals test cases execute the simulator with less clock cycles then the exact expected amount to assure that the pipeline is working correctly.

GUI helper test cases fall into the statement coverage adequacy notion. All function possibilities have a unit test that assure the correct output for the possible outputs. As the architectural test cases, the GUI helper tests use the assert equals and the assert not equals to confirm that the functions have the expected output.

The test set was developed along side the simulator implementation assuring the correction of the architecture. Developing the unit test along side the simulator implementation brought several advantages. When a change was implemented the test cases would be executed. When a failure occurred it would be simple to pin point the fault’s origin based on the failed tests. This saved a great deal of time when adding or changing a feature. To ensure that the change was well succeeded, the test set would be executed. If it passes, then the previous behavior was not changed. Without the test sets, when changing or adding new features, the developer had to manually assure that every thing was working properly, meaning a great amount of effort and time to assure something that takes seconds to do with unit testing.

5.4 Extensibility

Two of the most differentiating aspects of MIPSter32 compared to other MIPS simulators described in Section 2.2 is the modularity and extensibility of the implemented software architecture. Proving this with concrete results is not an easy task to accomplish. After thinking and discussing several options, the most solid and credible solution was to have a developer implementing new features and altering existing ones.
In the Summer of 2015 an internship was offered at Instituto de Engenharia de Sistemas e Computadores (INESC) for candidates that would be interested in developing new features for MIPSter32. Two interns, João Miguel Vieira and João Mota, accepted the challenge. Both students are enrolled at IST in the bachelor’s degree in electrical engineering.

5.4.1 Methodology

The given material to both students included the MIPSter32 source code, the developer’s guide presented in Appendix A, and the project’s Javadoc. The students were then asked to study and analyze the code. A week later, a meeting with the students took place in order to clarify any questions that they may have. After making sure both interns were abreast with the project, the specific tasks to be implemented were given to them. The tasks were carefully designed to prove the extendability and modularity of the MIPSter32. They were as follows:

- **Modular Design:** The simple task consisted of changing the MIPSter32 ALU implementation from a ModuleC to a ModuleS. This was the first task executed by the interns and its main goal was to demonstrate that the software architecture described in Section 3.1 is modular. The second goal of this task was to ensure that the students were understanding the software architecture as well as pinpointing the main flaws of the documentation.

- **Extensible Design:** After completing the task mentioned above, the students were more familiarized with the MIPSter32 software architecture and were now capable of performing more complex tasks. One student was asked to implement delay slots (described in Section 2.1.6) and the other student was asked to implement forwarding (described in Section 2.1.5). By implementing new features using the designed software architecture, MIPSter32 proves to be extensible.

During the internship period, regular meetings were scheduled with the students to help with any issues or questions that they might have. These meetings were around one hour and theoretical and practical questions were discussed. The students presented more difficulties with the theoretical component of the implementation than with the development process. This reinforces the fact that the given documentation was well done.

5.4.2 Results

The overall result for the internship was positive. Both students managed to understand the project's architecture and perform the desired tasks. It is important to note that both students are
not taking computer science degrees, making a greater challenge when implementing the desired features in Java. It would be expected that the greatest challenges would be the implementation, but only João Mota demonstrated difficulties in the development phase. One student showed the greatest commitment to the project, and the final result stands out with an excellent internship result.

The first task was implemented with success by both students. They developed a ModuleS ALU. As shown on Figure 5.25 the ALU is now implemented using other hardware components. The ALU diagram was developed and implemented on MIPSter32 by João Vieira. It took about two weeks for the students to complete the task. Due to their inexperience with Java, the main difficulty faced by the students were on how to translate the diagram to code. After understanding better the implementation methodology, the students demonstrated a great capacity of translating developed diagrams to Java code.

After completing the first task, one student rapidly developed the other task, where as the other student faced more difficulties that lead to a demotivation on his part. In Figure 5.26 the proposed architecture to implement branch prediction not taken by João Vieira is shown. When the architecture design was approved, he rapidly implemented the necessary changes to the MIPSter32. He also developed more unit tests to assure that the implemented architecture was performing as expected.

The implemented architecture followed the designed mentioned on the Patterson’s book ((33)). A known optimization to the pipeline is passing the branch resolution from the MEM stage to
the Instruction Decode stage. After this implementation he went on to develop MIPSter32 with branch optimization. Figure 5.27 is the proposed and implemented architecture of MIPSter32 with branch optimization by João Vieira.

After completing the two new implementations he finished by comparing the implemented architectures with each other to find out how these optimizations impacted the overall performance of the CPU. This was an important analysis that demonstrated the impact of these optimizations when executing the same program. The test consisted on running two programs, one that calculates the factorial value of a given number, and another that does iterative multiplication. The final result can be seen on Figure 5.28 and Table 5.5.

João Mota implemented with ease the ALU task, but when faced with a more complex task some difficulties arose. He faced troubles while implementing the architecture diagram and understanding the forwarding concept. After accomplishing a theoretical architecture that implemented forwarding, he got onto the development where more difficulties arose. He did implement an almost correct implementation of the data forwarding optimization, that only failed three tests of the forty three given test set.

This internship had an overall positive reaction from the students, and came to prove that with the developed base architecture and documentation, developers are capable of changing MIPSter32 base to their needs.
CHAPTER 5. EVALUATION

Figure 5.27: Branch optimization implemented by João Vieira.

Figure 5.28: Cycle analysis between different architectures performed by João Vieira.
Chapter 5 focuses on the evaluation of this thesis. In Section 5.1 introduced the usability of the implemented GUI. Section 5.2 describes the performed performance tests and the respective results. Section 5.3 explains in detail the performed unit tests that guarantee the correctness of MIPSter32. This chapter ends in Section 5.4 with the performed internship that proves the extensibility of the presented work.
### Table 5.4: Performance test results.

<table>
<thead>
<tr>
<th></th>
<th>Program A (seconds)</th>
<th>Program B (seconds)</th>
<th>Program C (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000 clock cycles</td>
<td>0.23</td>
<td>0.312</td>
<td>0.254</td>
</tr>
<tr>
<td>5,000 clock cycles</td>
<td>1.054</td>
<td>1.143</td>
<td>1.1</td>
</tr>
<tr>
<td>10,000 clock cycles</td>
<td>2.193</td>
<td>2.319</td>
<td>2.274</td>
</tr>
<tr>
<td>100,000 clock cycles</td>
<td>19.627</td>
<td>21.37</td>
<td>20.582</td>
</tr>
</tbody>
</table>

### Table 5.5: Cycles that each architecture takes to execute the same assembly program. Legend: **NO**: No Optimization; **DS**: Delay Slot; **BPNT**: Branch Prediction Not Taken; **BO**: Branch Optimization; **DS/BPNT**: Delay Slot and Branch Predict Not Taken; **DS/BO**: Delay Slot and Branch Optimization.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Iterative multiplication (1000 × 100)</th>
<th>Factorial (12!)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>1113</td>
<td>1103</td>
</tr>
<tr>
<td>DS</td>
<td>1113</td>
<td>1037</td>
</tr>
<tr>
<td>BPNT</td>
<td>813</td>
<td>846</td>
</tr>
<tr>
<td>BO</td>
<td>711</td>
<td>717</td>
</tr>
<tr>
<td>DS/BPNT</td>
<td>813</td>
<td>845</td>
</tr>
<tr>
<td>DS/BO</td>
<td>611</td>
<td>639</td>
</tr>
</tbody>
</table>
Chapter 6

Conclusions

One of the most used book to teach computer architecture classes is the Patterson [33] book. To teach the main concepts, the book uses MIPS 32-bit ISA due to its simplicity. There are several MIPS simulators, but none that offer such a complete toolkit as the MIPSter32. MIPSter32 follows the same architecture as the Patterson book, has a GUI designed that meets the user’s needs, and has an extensible and modular design.

MIPSter32 can be used in different aspects. The two main practical uses consist on the following:

- **Educational Uses:** The presented thesis project can be used as a complementary tool for the Patterson [33] book. It was developed in the lights of the presented architecture taught by Patterson, making it the perfect laboratory tool for students learning computer architecture.

- **Scientific Uses:** Being extensible and modular researchers and curious students can alter the simulator to implement specific architectural optimizations such as branch delay slots and study its impact on the execution of a program.

6.1 Main Contributions

The presented work is an Advanced Micro-Architecture (explained in Section 2.2.1) open source simulator. MIPSter32 takes compiled program binaries, and simulates its execution on a simple computer system comprised of a MIPS 32-bit pipeline CPU and a RAM memory bank. MIPSter32 provides a rich and user-friendly interface, and a set of controls that allow for fine grain control of a programs execution flow. It will not be an assembler like some examples presented in the
CHAPTER 6. CONCLUSIONS

Related Work (Section 2.2). The key contributions for the thesis are presented next.

6.1.1 Graphical User Interface (GUI)

The developed GUI is based on WinMIPS64 [37], mainly because it is widely used by the community. To achieve a personalized GUI the WinMIPS64 interface was carefully studied and analyzed with refinements done by conducting a survey with IST students, adapting the WinMIPS64 GUI to the specific future user needs. The developed GUI is designed to show at any given clock cycle the processor's state along with other statistical information. It also allows the user to manually alter the memory and register information at runtime. The implemented panel based interface allows the user to define how the information is displayed, making it adaptable to specific user needs.

The importance of having a well designed user interface will help future users to a better understanding of computer architecture concepts. With the personalization aspect of the implemented GUI, the same interface can be used to fit several user's specific needs. Depending on what the focus is, users may hide or change a panel's dimension to get a greater focus on what they are studying.

6.1.2 Modular and Extensible

Having a modular designed means that the system can be subdivided into smaller parts that are usually called modules. These modules can be created separately and integrated into the existing system offering the possibility of scaling the project to specific needs. Having a modular system will allow the change in modules making the simulator more adaptable to different needs and goals. Users may want a more detailed and accurate module such as the ALU. Having a modular system the user may recreate the ALU with a different architecture with out having the need to rebuild the entire simulator. The possibility of being scalable makes it possible to easily add new functionalities to the simulator for future works. A great example is a memory simulation tool. Instead of adapting existing tools to complement missing features the possibility of adding them with reduced effort is a great feature that MIPSter32 offers to the scientific community. Since all the source code will be available any one can change and add modules, making this simulator very adaptable to different user needs.

MIPSter32 is modular and scalable within limits. The design does not have in focus the possibility of adding new stages to the pipeline. Additional extensions will have to implement certain interfaces to be compatible with MIPSter32 function calls. If a user desires to implement a new
memory, that class has to implement the required functions called by the MIPSter32 when accessing the memory for values. Only some pipeline stages such as the ALU and memory access stage can be modulated. To modulate a certain component, the provided interface of that component has to be implemented.

### 6.1.3 Documentation

As an open source, modular, and scalable project a good documentation is essential to take full advantage of these features. If the user is unable to understand the code it will be nearly impossible to change or add new features. Along with the source code, a developer’s guide and Javadoc [3] are available to aid on the development of extra features.

For regular users a user manual is provided to help and guide users on a correct simulator usage. It also contains detailed information of all the different panels and possible configurations. Having good documentation and user manual contributes to an overall greater user friendly experience.

### 6.1.4 Test Set

To validate that MIPSter32 has the desired behavior, a test set was implemented, previously described in Section 5.3. The test set consists of test cases that will test all possible aspects of the simulation as well as GUI helper functions, guaranteeing correct execution of the presented work.

The importance of having a well designed test set are essential to assure the expected behavior of the simulator. Also, by providing future developers with a complete test set, eases their work, because any implemented change can quickly be proven to not affect the expected simulator’s execution. If any implementation causes a fail on the test set, then it can rapidly be identified, saving time to the developer on pin pointing the origin of the failure.

### 6.1.5 Compatibility and Portability

MIPSter32 was all developed in Java for compatibility and portability purposes. Deciding which programming language to adopt was a crucial decision for the implementation. Because each programming language has its own features, advantages and disadvantages, a close examination is very important. In particular, factors such as portability and efficiency need to be taken into account for an appropriate decision.

Having the educational goal in mind, and having bachelor students as the main users of the
CHAPTER 6. CONCLUSIONS

proposed simulator the first language that comes to mind is Java. It's a very popular programming language and familiar to almost every student of computer science. In the presented MIPS simulators the only language that is as portable as Java is the HTML with ASP. However, this alternative has some crucial drawbacks. Since the execution takes place on a remote server some precautions have to be made. Giving the example of WebMIPS [14] it is limited to 1000 cycles to avoid erroneous code submitted by users. Every presented MIPS simulator implemented with Java does not have this limitation because it executes on the local machine.

Plausible alternatives to Java include C and C++, which are faster and more efficient then Java. However, they introduce a portability problem. The user interface of C and C++ would have to be done using external libraries, while in Java offers Swing. Both languages have to be compiled for different computer architectures. The educational focus of the project makes portability more desirable than performance.

6.2 Future Work

For future works there are some aspects that can be worked to better the practicality and usability of MIPSter32. The first step would imply some modification and implementation to the developed software architecture. The first thing would be to implement an exception package that the simulator may throw. This could then be used to simulate some functionalities such as system function calls, detect ALU errand operations such as dividing by zero, and assure the correct values are being passed to a specific module. To clarify on the last point, a multiplexer expects two values on the input, but the flag must be a boolean. The MUX class may throw an exception when the selector is not a boolean. The last architectural change suggested for future work is the reorganization of the GUI menu classes. The presented work has all menu options implemented on the main GUI interface due to the low amount of options offered by the menus. As MIPSter32 grows, more options will arise, making it one important future work implementation.

Other future works may consist on implementing different MIPS architectures. MIPSter32 implements a simple and non optimized architecture. Having other optimizations such as branch delay slots or forwarding would be good to graphically demonstrate students and users the impact of these changes on the overall execution of a more complex problem. Having different architectures also enriches MIPSter32 by offering users more material to study.

The final suggestion for future work would be to implement a complete memory simulator to incorporate with MIPSter32. Implementing this would greatly enrich the simulator by turning the simulation more accurate due to memory conflicts and delays that are not implemented on the
6.2. FUTURE WORK

presented work.

MIPSter32 was designed and implemented with all these features in mind. The modular and extensible design eases future implementations, making it the perfect educational tool on a computer architecture lab.
Bibliography


[38] Mike Scott. Winmips64. http://indigo.ie/ mscott/.


Appendix A

Developer Manual

This manual aids developers that want to extend or modify the present project. Along with the project’s JavaDoc and source code developers have the necessary documentation and material to start developing the MIPSter32.

A.1 Project Structure

Mipster32 project setup consists of various packages to better organize the project. The main package is pt.ulisboa.tecnico.mipster32. This package contains the upper level classes of the project. It includes:

- Module.java
- ModuleC.java
- ModuleS.java
- ModuleR.java
- Cpu.java
- CpuNO.java
- CpuBO.java
- GraphicalPanel.java
- GraphicalSimulator.java
The module classes abstract the main attributes and methods that the individual and more specific hardware simulation classes have to implement. The ModuleC.java is an interface that represents combinatorial modules, the ModuleS.java is an interface that represents a composed module (S for Super), and the ModuleR represent a register. It is essential to differentiate between these three different module types due to their distinct behaviour. Cpu.java is just an abstraction class that provides an interface for all necessary methods and variables that a CPU has to implement for the MIPSter32's interface to work correctly. This way it is simple to switch architectures at run time. CpuNO.java implements an architecture with no optimizations, where as CpuBO.java implements a MIPS architecture with a branch optimization. GraphicalSimulator.java is the main class that executes the graphical interface where as the GraphicalPanel is a class that all graphical panel's extend from. These two last classes are regarding the GUI and do not impact in any way the architecture of the simulation. All these classes are MIPSter32's skeleton, therefore should never be modified.

The next package is where all the combinatorial modules are placed. This package is called pt. ulisboa. tecnico. mipster32. combinatorial_module. It includes several simulated modules such as an add module, an or module, a shift left module, among others. All new combinatorial modules should be placed in this package and extend the ModuleC class.

The second type of module is the composed module (or super module), that are all present in the package pt. ulisboa. tecnico. mipster32 .composed_module. These modules are just a container class that group the combinatorial and register modules. MIPSter32 uses these modules to create the pipeline stages. Each stage is composed of several different kinds of modules that perform a more complex task. In this package should be placed any kind of module that aggregates multiple modules including new or different stage implementations.

The next package refers to the register modules. All modules that include registers are placed at pt. ulisboa. tecnico. mipster32.register_module. These modules have an internal memory, therefore the differentiation from the composed modules. A simple example is the program counter that is represented by the class ProgramCounter.java, that consists of one register that stores the PC value.

All GUI components are placed at pt. ulisboa. tecnico. mipster32. gui and pt. ulisboa. tecnico. mipster32. gui. classes. Classes that are directly involved with graphical components such as button bars, menus, and pannels are placed in the gui package. The gui.classes are specific objects that module specific information that is then used at the interface. Developers should only work on these two packages if they desire any graphical change.

MIPSter32 is a CPU simulator, but to make it realistic and functional memory components have
to be implemented. At \texttt{pt. ulisboa. tecnico. mipster32. memory} simple simulated memory components are placed. For future work an entire memory system simulator can be placed here as long it extends the Memory.java class.

For abstraction purposes abstract classes for each individual pipeline stage have been created and placed at the \texttt{pt. ulisboa. tecnico. mipster32. stages}. If a developer wants to change a specific stage to implement a specific optimization it will have to extend one of the stages present in this package. These classes cannot be modified because they are interfaces that the GUI uses to work properly. If a new stage is to be implemented, then the corresponding abstract class needs to be created and placed in this specific package.

For system configurations and constants developers should used the \texttt{pt. ulisboa. tecnico. mipster32. system.configurations} package to place their files.

To ensure a correct simulation unit tests have been developed and placed at \texttt{pt. ulisboa. tecnico. mipster32. unit.tests}. The current tests ensure the correct simulation of all the different CPU architectures as well as several GUI components.

The package \texttt{pt. ulisboa. tecnico. mipster32. wires} includes the different kinds of wires that connect the modules to each other. These classes represent the connection between different modules that build the CPU.

\textbf{A.2 Import QtSpim code to MIPSter32}

- Compile code on QtSpim

- Copy the instruction bytes and paste on a file (note: ignore the initialization instructions of QtSpim. Just copy the main program instructions)

- For the Branch and Jump instructions decrement the value by the PC Value

- Comments may be used by placing a (Space) followed by a # after the instruction
Appendix B

User Manual

This manual is directed to current and future users of the MIPSt32 simulator. It includes a detailed overview of the interface and the possible actions from the menus buttons and panels.

B.1 Graphical User Interface

Figure B.29 is the MIPSt32 user interface. Its main sections are the top menu(item 1), the button bar (item 2), and the panel section (item 3-8). Following is the item list present on figure B.29.

• 1: The menu is used for simulation options.
  – **File**: Only has the option Choose File. It opens a specific compiled assembly code to the simulator. The shortcut SHIFT+O can be used to perform the same operation.
  – **Simulation**: Choose the desired architecture to simulate.
  – **View**: The Reset Window Size is used to set the window size to the pre-defined dimension. The Hide/Show options toggle the panels visibility.

• 2: The button tool bar. It contains several quick access functions.
  – **Step**: Performs one clock cycle.
  – **Play**: Executes the program to the end.
  – **Reset**: Resets the simulator’s state back to default values.
  – **Reg Number**: Shows the number of the register on the Register panel.
  – **Reg Alias**: Shows the alias of the register on the Register panel.
Hexadecimal: When active all the values are displayed in hexadecimal. Otherwise the values are displayed as decimal values.

3: The Pipeline panel displays the five pipeline stages of the MIPSter32 architecture represented as squares. The stages follow a color scheme that is used on panels 7 and 8. Each color represents a stage, and when it is used demonstrates that a specific instruction is being executed at that stage.

4: Registers panel displays the registers content. Double clicking allows to change its value.

5: Memory panel displays the memory content. Double clicking allows to change or add a new memory value.

6: Statistics panel displays several program execution statistics.

7: Cycles panel displays a grid view of the instructions life cycle. Each element represents where the instruction was at each clock cycle. When that instruction is at a specific stage, the cell is filled with the stage’s color. When the instruction has finished or is not being executed then a "-" is placed on the cell.

8: The Code panel displays the loaded program’s instructions. When an instruction is being executed it is highlighted with the stage’s corresponding color.

B.2 Import QtSpim code to MIPSter32

- Compile code on QtSpim
- Copy the instruction bytes and paste on a file (note: ignore the initialization instructions of QtSpim. Just copy the main program instructions)
- For the Branch and Jump instructions decrement the value by the PC Value
- Comments may be used by placing a (Space) followed by a # after the instruction

B.3 Execute a Program

To execute a program on the Simulator, the compiled assembly has to be loaded using the menu option Choose File. After the code has been loaded, the code panel displays the instructions. After it has been loaded, just hit step or play to run the program.
Appendix C

Conducted Survey

This survey was conducted on IST students to determine what graphical components could be enhanced and improved.
WinMIPS64

All scale question are rated 1-6. The goal of this questionnaire is to understand how you, as a student, think the simulator helps on your understanding of a pipeline.

* Required

1. Your e-mail for later contact regarding this project.
   Your w-mail will not be published or turned public

2. How helpful do you consider the Cycles panel? *
   Mark only one oval.

   Not at all   1   2   3   4   5   6

   Very helpful

Cycles Panel
3. How helpful do you consider the Registers panel? *
   Mark only one oval.

   Not at all   Very helpful

Registers Panel

4. How helpful do you consider the Statistics panel? *
   Mark only one oval.

Statistics Panel
6. How helpful do you consider the Pipeline panel? *
   *Mark only one oval.*

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not at all</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Very helpful</td>
</tr>
</tbody>
</table>

7. Do you understand this figure? Why not?
   *
8. How helpful do you consider the Data panel? *
*Mark only one oval.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Not at all</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Very helpful</td>
</tr>
</tbody>
</table>

Data Panel
9. How helpful do you consider the Code panel? *

Mark only one oval.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not at all</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Very helpful</td>
</tr>
</tbody>
</table>

Code Panel
10. Which pipeline diagram do you like the most? *

See image below

Mark only one oval.

☐ A
☐ B

11. Why?

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

Different Pipeline Diagrams
12. Would you like to have an option to undo the previous instruction? *
   Mark only one oval.
   
   [ ] Yes
   [ ] No

13. What would be some features that the WinMIPS64 does not support that you think would be helpful for a better understanding of a pipeline?
   You can suggest more then one

   ________________________________________________________________
   ________________________________________________________________
   ________________________________________________________________
   ________________________________________________________________
   ________________________________________________________________