Abstract

MIPS is a RISC instruction set architecture 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 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.

Categories and Subject Descriptors CR-number [subcategory]: third-level

Keywords MIPS, 32-Bit architecture, Simulation, Educational Simulation Tools, Modular, Extensible, OpenSource

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 (Brors- sill, Sm), instruction set architectures such as MARS (Vollmar and Sanderson 2006), and computer systems including processors, memories and I/O such as GEMS (Martin et al. 2005). Simulators can fall into seven main categories defined by Wolfe (Wolfe et al. 2002). Historical machine simulators, digital logic simulators, intermediate instruction set simulators, simple hypothetical machine simulators, intermediate instruction simulators, advanced micro-architecture simulators, multiprocessor simulators, and memory subsystem simulators. The presented work falls into the micro-architectural simulators.

Micro-Architecture Simulators is a tool developed to model the design and behavior of a microprocessor and its components. This type of simulators is ideal for testing concepts such as branch prediction, cache trace, and re-order 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 (Patterson and Hennessy 2013). MIPSter32 fits on the micro-architectural simulation.

There are many micro-architecture simulators such as Shade (Cme- lik and Keppel 1995), Spim (Page 2009), MARS (Vollmar and Sanderson 2006), WinMIPS64 (Mike Scott), EduMIPS64 (Cata- nia et al. 2014), SimpleScalar (Austin et al. 2002), WinDLX, MiniMIPS (Bem and Petelczyc 2003), WebMIPS (Branovic et al.
2004), ProcessorSim (James Garton), among others. In this section the focus will be more one functional features of similar simulators to MIPSter32.

**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 (Page 2009) 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 (Vollmar and Sanderson 2006).

**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 staged 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 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 (Branovic et al. 2004).

**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 (Mike Scott). 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. Break points can be set simply by left-clicking on the code window (Mike Scott).

The execution can be run with or without forwarding, something that the WebMIPS (Branovic et al. 2004) 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. Even though WinMIPS64 and EduMIPS64 have desirable features for teaching they both simulate a totally different architecture and ISA from the Patterson’s book (Patterson and Hennessy 2013).

**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 (Patterson and Hennessy 2013). Table 1 compares existing simulators against four criteria mentioned by Sarjoughian (Sarjoughian et al. 2008).

<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>Java</td>
</tr>
<tr>
<td>WebMIPS</td>
<td>No</td>
<td>No</td>
<td>Yes</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 1. Comparison of MIPS processor simulators. Referenced in Section 2

### 3. MIPSter32 Architecture

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 modularize 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 Microprocessor without Interlocked Pipeline Stages (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 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. 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 central processing unit (CPU). Taking this abstraction approach, the CPU is a component that has several stages that at each cycle clock 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 this section all the components will be carefully explained.

#### 3.1 Module

Module is at the top of the abstraction level. As shown in Figure 1 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 later on this section) 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. 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 Graphical User Interface (GUI). Each module implements a way to display its information to be used on the GUI.

There are two main roles that a hardware component can 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\textsuperscript{1} 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 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 2, 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 earlier, 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 output 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, 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.

### 3.2 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.

### 3.3 Graphical User Interface

When designed properly, 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.

**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. Bellow 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 than 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 figure all said yes, but some commented that only because they had previously studied it in class.
- 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.

3.4 GUI Design

Figure 3 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.

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 4A 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 (subdivided 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 4, 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 4, positive comments also came in favor of option B of the Figure 4. Students pointed out that even though option A of Figure 4 is easier to comprehend at the beginning, option B of Figure 4 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 (Patterson and Hennessy 2013), including a pipeline diagram is essential for the proposed solution. Along with the pipeline diagram a cycle panel 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 simple panel is so similar to the book (Patterson and Hennessy 2013) 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 3.4.

**Code**

Even though the code section could just be a simple window where all the instructions are displayed to the user, the MIPSter32 - A 32 bit MIPS Simulator 4 2016/5/8
Ster32 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 3.4 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 (Mike Scott) 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 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 (Vollmar and Sanderson 2006) 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 (Patterson and Hennessy 2013).

3.5 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 programing 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 (Branovic et al. 2004) 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 then performance.

3.6 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 know as JUnit (jun). 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.
3.7 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.

3.8 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 (jav) 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.

4. Evaluation

In this section the evaluation process of the presented work is described.

4.1 Usability

As mentioned in Section 3.3, 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 shows the initial state when a user uploads compiled code to MIPSter32.

Figure 6 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.

4.2 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 2 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.

4.3 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 (Zhu et al. 1997) 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 programs 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.

4.4 Extensibility

Two of the most differentiating aspects of MIPSter32 compared to other MIPS simulators described in Section 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 Instituto Superior Técnico (IST) in the bachelor’s degree in electrical engineering.

Methodology

The given material to both students included the MIPSter32 source code, a developer’s guide, 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 is modular. The second goal of this task was to ensure that the students were understanding the software architecture as well as pin pointing 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 and the other student was asked to implement forwarding. 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.

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 7 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 8 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 (Patterson and Hennessy (2013)). 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 9 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 10 and Table 3.

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.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Iterative mult ((1000 \times 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>

Table 3. 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.

5. Conclusions

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 (Patterson and Hennessy 2013) 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.

5.1 Main Contributions

The main contribution of this work is the design and implementation of MIPSter32, a 32-bit MIPS Instruction Set Architecture (ISA) simulator. As described earlier, there are several offers of processors simulators, but none that follow the Patterson (Patterson and Hennessy 2013) book closely together with having a GUI designed to help students learn the way a processor works. MIPSter32 simulates the MIPS 32-bit ISA described in Patterson (Patterson and Hennessy 2013) book. We highlight some of the most important features of MIPSter32:
• **Graphical User Interface (GUI)** The developed GUI is based on WinMIPS64 (Scott), mainly because it has a simple yet complete GUI. To define the most adequate interface that meets user needs a survey with IST students was conducted to refine several aspects of the WinMIPS64 GUI.

• **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 4.4, 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. The test set consists of test cases that test all possible aspects of the simulation as well as GUI helper functions, guaranteeing the correct execution.

5.2 Future Work

The first future work could 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 that the simulator may throw. This could then be used to simulate the execute all possible aspects of the simulation as well as GUI helper functions, guaranteeing the correct execution.

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 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.

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, Jos 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.

References

References


