

Di-MIPS

Practicum

Computer Architecture Project For Simulation Of Dual-Issue MIPS Architecture

Academic Year/course: 2022/23

Code: 21735

Authors:
Tomas Dengra & Vasil Andreev & Vicenç Servera

Index

| | |
|--------------------------|----|
| Introduction..... | 10 |
| Goals of the system..... | 20 |
| The system..... | 30 |
| Code..... | 50 |
| Examples of use..... | 60 |
| Further ideas..... | 70 |

1. Introduction

Di-MIPS is an academic project, conducted by Tomas Dengra, Vasil Andreev and Vicenç Servera for the elective course of Computer Architectures in the University of Illes Balears during the first semester of year 2023-2024. Its main purpose is to apply the knowledge of the material studied in the course, using tools and processes that are learned during the academic and professional realization of the students. This includes the work of communication, software engineering, software architecture, writing code and documenting the process.

The time spent on the project is 30+ hours total, including computing laboratory sessions during the course and the out-of-university sessions organized by the participants in the project. The communication during the out-of-university sessions was piloted by the platform for voice chat Discord [discord.com]. The methodology for the process was sorted with an informal implementation of a mixture by the iterative methodologies Agile and the Kanban, dividing the process into steps of planning, designing, developing and reviewing.

During the planning the work was split into small tasks that were then tackled by the contributors, while a review session was regulating the progress of the system weekly or biweekly. For version controlling, the technology of Git with the platform GitHub [github.com] was used, alongside with various tools and environment add-ons for easing the application of Git. For software environment, various code editors were used, including Visual Studio Code, Eclipse and NetBeans. The programming language of choice was Java 11 [java.com]. Documentation of the project is realized in LibreOffice Writer [libreoffice.org], while the diagrams are created with the help of the Visual Paradigm [visual-paradigm.com] and DrawIO [drawio-app.com].

Source of inspiration for the project were the lectures led by Dra. Catalina Lladó, specifically the chapter of Pipelineing, aided by the book “*Computer Organization and Design MIPS Edition: The Hardware/Software Interface*” by David A. Peterson and John L. Hennessy.

2. Goals Of The System

The goal of the system is to reproduce the execution of instructions by the Dual-Issue MIPS computer architecture. The implementation of which could be used for future teaching practices or further investigation and development of the simulation. It could be used to solve problems in entry-level academic investigation or a tool that can help the observation of more complex examples of MIPS executions.

An exact use-case is the execution of a specific set of instructions for the architecture and visualizing their timeline in cycles.

3. The System

The simulation of the Dual-Issue architecture is broken down into three phases: configuration, execution and outputting. Using the configuration, we set the parameters as the number of instructions, their order, their types and the registers. Then after we start the simulation, the Di-MIPS executes code

based on the data of the configuration file, creating an output file, where the results of the simulation are formatted to be read. Those 3 layers are shown in figure 1.

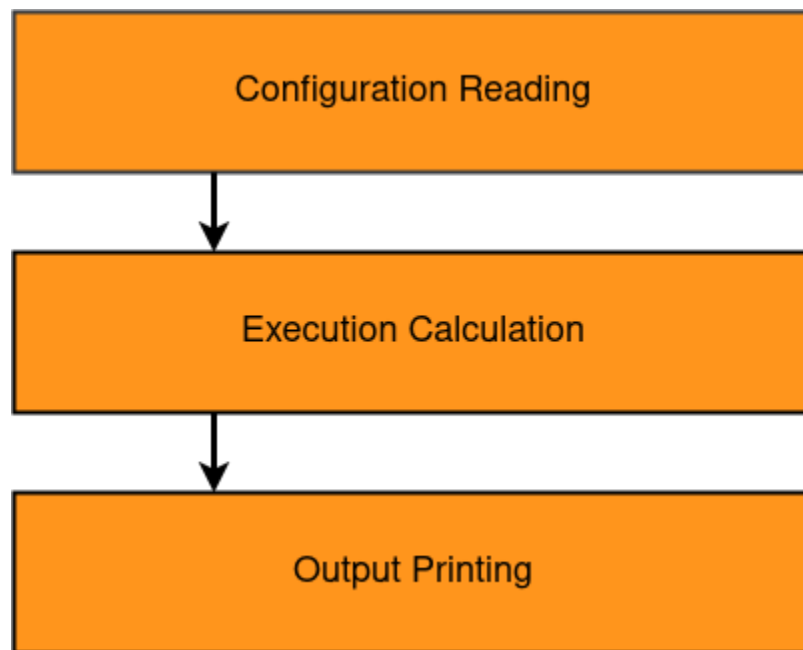


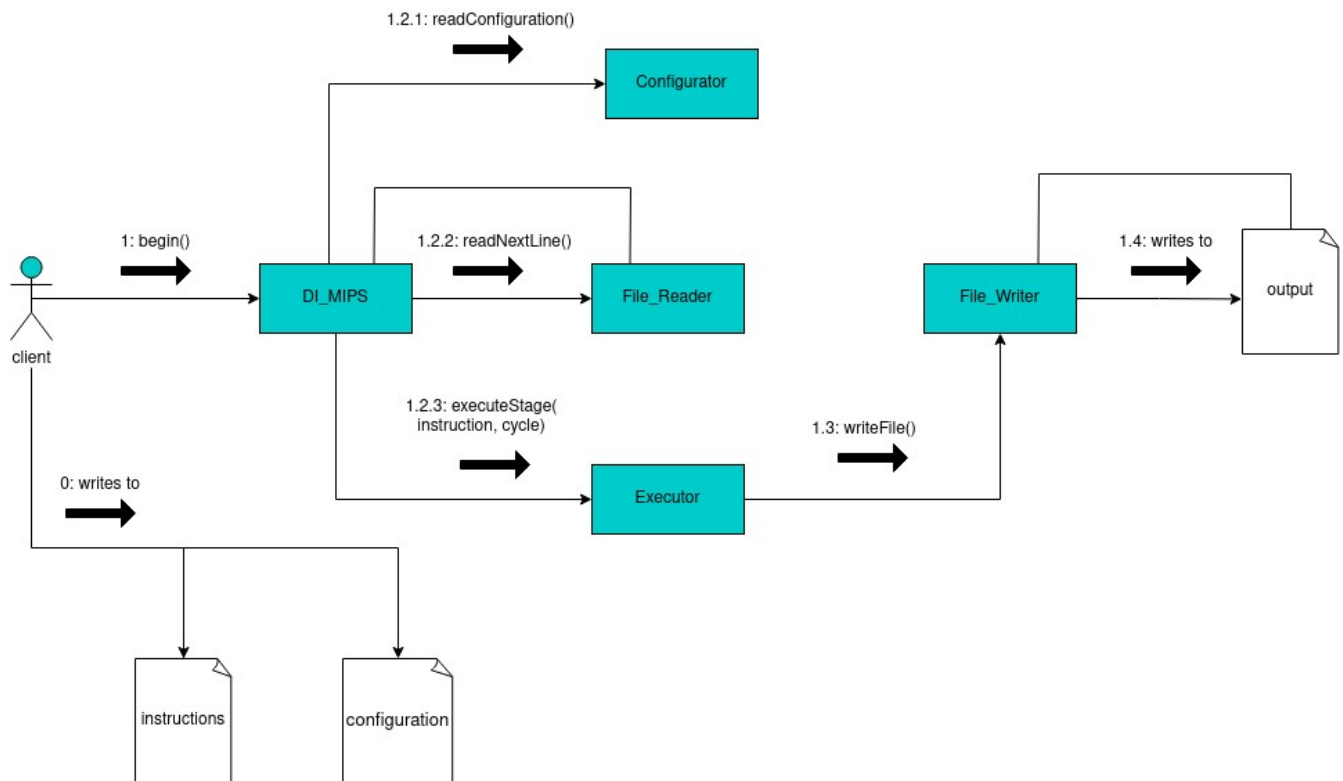
Figure 1 – Layering of the Di-MIPS system

A detailed dive into the system will show a more complex way of communication between the different layers of the system. For example, the Configuration Reading layer has 3 components: the configuration file, the instructions file and the File_Reader class that reads from them. The user should edit the configuration file to change the parameters of the type of execution and the instruction file to set the arguments to the order and type of the instructions he wishes to simulate executions for.

Starting the Di-MIPS system will trigger the execution of the simulation in the following manner:

1. Reading from the two user-edited files.
2. Passing the data from the instruction and configuration file to the Executor class, responsible for the main logic of the interpretation of the data
3. The Executor sends the computed stages and cycles to the writer
4. The Writer handles the formatting and the writing the data into an output file

The communication of the different components is described in the following communication diagram:



4. The Code

4.3 Writer

The `File_Writer` is the class that forms the core of the third logical part, the Output Printing. It has the responsibility to take the data of the Execution Calculation, prepare it in a suitable format for reading and store it in a file. It is initialized without parameters, for simplicity, because by default it always creates the output file in the source folder, not leaving it customizable. There are two public functions: `writeStage` for writing data to the class and `makeFile` to write the class's data to an output file. We use Java's `FileWriter` class.

```

public File_Writer() {
    this.stages = new ArrayList<>();

    this.instructionsOperators = new LinkedHashMap<Integer, Operator>();
    this.instructionsStart = new HashMap<Integer, Integer>();

    this.cyclesCount = 0;
    String fileName = "./output.txt";

    try {
        this.writer = new FileWriter(fileName);
    } catch (IOException e) {
        System.out.println(e.getMessage());
        e.printStackTrace();
    }
}

```

The way the class is working is that, before writing the output, it stores data in three collections:

- **stages** - the matrix of instructions by cycles. It is in a list format, where every element is representing one instruction. The instruction is represented as a map (Hash Map), which maps every stage of the instruction, to a specific cycle number, according to when it is executed.
- **instructionsOperators** - a map which relates the identification number of the instruction with the type of instruction (i.e. ADD, SW, etc.). Its implementation is Linked Hash Map so that we can keep the order of the instruction that we put into it, with the idea to iterate through it.

instructionsStart - another map that relates the instruction identification number with the cycle where the instruction starts execution. As we know, because many factors influence the cycle where an instruction starts, we allow a specification through this second Hash Map

```

public class File_Writer {
    private List<Map<Integer, Stages>> stages;
    private Map<Integer, Operator> instructionsOperators;
    private Map<Integer, Integer> instructionsStart;

    private FileWriter writer;
    private int cyclesCount;
}

```

The first public function there is, is the writeStage, which contains the main logic for the data manipulation of the aforementioned collections. First it updates the cycle count if it gets a stage in the next cycle, progressing the execution. Second it checks if the stage that is written is related to an existing instruction or is the start of a new instruction. We will omit the in-depth explanation of the data manipulation, as it is standard for the types of collections and nature

of the data. It is essentially matching the ids of the instruction with the stages, until the whole matrix of instructions/cycles is written in.

```
public void writeStage(int cycle, int instructionId, Stages stage, Operator operator) {
    if (cycle > this.cyclesCount) {
        this.cyclesCount = cycle;
    }

    if (!this.isInstructionEncountered(instructionId)) {
        this.addNewInstruction(cycle, instructionId, stage, operator);
    } else {
        this.addNewStageForInstruction(cycle, instructionId, stage);
    }
}
```

The other part of the File_Writer is the writing of the file. We also close the writer that we opened in the constructor and handle errors.

```
public void makeFile() {
    try {
        this.writeFile();
        this.writer.close();
    } catch (IOException e) {
        System.out.println(e.getMessage());
        e.printStackTrace();
    }
}
```

We split the writing into three parts: printing the cycle numbers headline, the header with the timeline for the cycles; formatting the newlines and spacings of the file and lastly - printing the stages for the corresponding instructions.

```
private void writeFile() {
    this.writeCycleTimelineHeader();
    this.writeNewLineToFile();
    this.writeInstructionRows();
}
```

For the purpose of simplicity, we will have a look at just one function from the writing to the file, as they are similar in their responsibility and logic. Here we have the writeCycleTimelineHeader:

```

private void writeCycleTimelineHeader() {
    final String TAB = "\t";
    final String SPACE_BETWEEN_NUMBERS = "  ";

    StringBuilder cyclesHeader = new StringBuilder(TAB);
    for (int i = 0; i < this.cyclesCount; i++) {
        cyclesHeader.append(i + 1);
        cyclesHeader.append(SPACE_BETWEEN_NUMBERS);
    }

    this.writeLineToFile(cyclesHeader.toString());
    this.writeNewLineToFile();
}

```

Knowing the cycles count, we iterate in a loop, concatenating the strings for the spacing between the cycles that we also add in a String Builder. After adding the tabulations and spacings, we finish the builder and pass the final string to the writeLineToFile function. It's the most simple way to write a string to a file, using the write method of Java's FileWriter class.

```

private void writeLineToFile(String line) {
    for (int i = 0; i < line.length(); ++i) {
        try {
            this.writer.write(line.charAt(i));
        } catch (IOException e) {
            System.out.println(e.getMessage());
            e.printStackTrace();
        }
    }
}

```