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

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 Iles 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, developping 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 enviorment add-ons for easing the application of Git. For software enviorment, various code editors were used, inlcuding 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, specifiaclly the chapter of Pipelineing, aided by the book “*Computer Organization and Design MIPS Edition: The Hardware/Software Interface*” by David A. Petersonand John L. Henneesy.

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. 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. Besides that, the simulation can be further developed to form a base for future projects or to serve as a point of reference on a example of a complex execution of a set of MIPS instructions.

An general 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 achitecture 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.

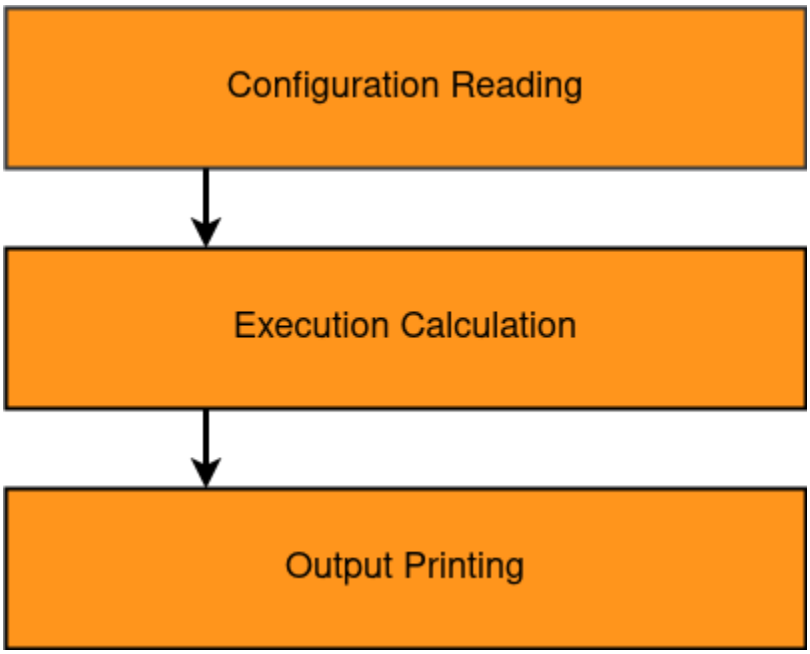

Figure 1 – Layering of the Di-MIPS system

A detailed dive into the system will show a more complex way of communication between the diferent 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, responisble for the main logic of the interpretation of the data
3. The Executor sends the computated 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 communication diagram in Figure 2.

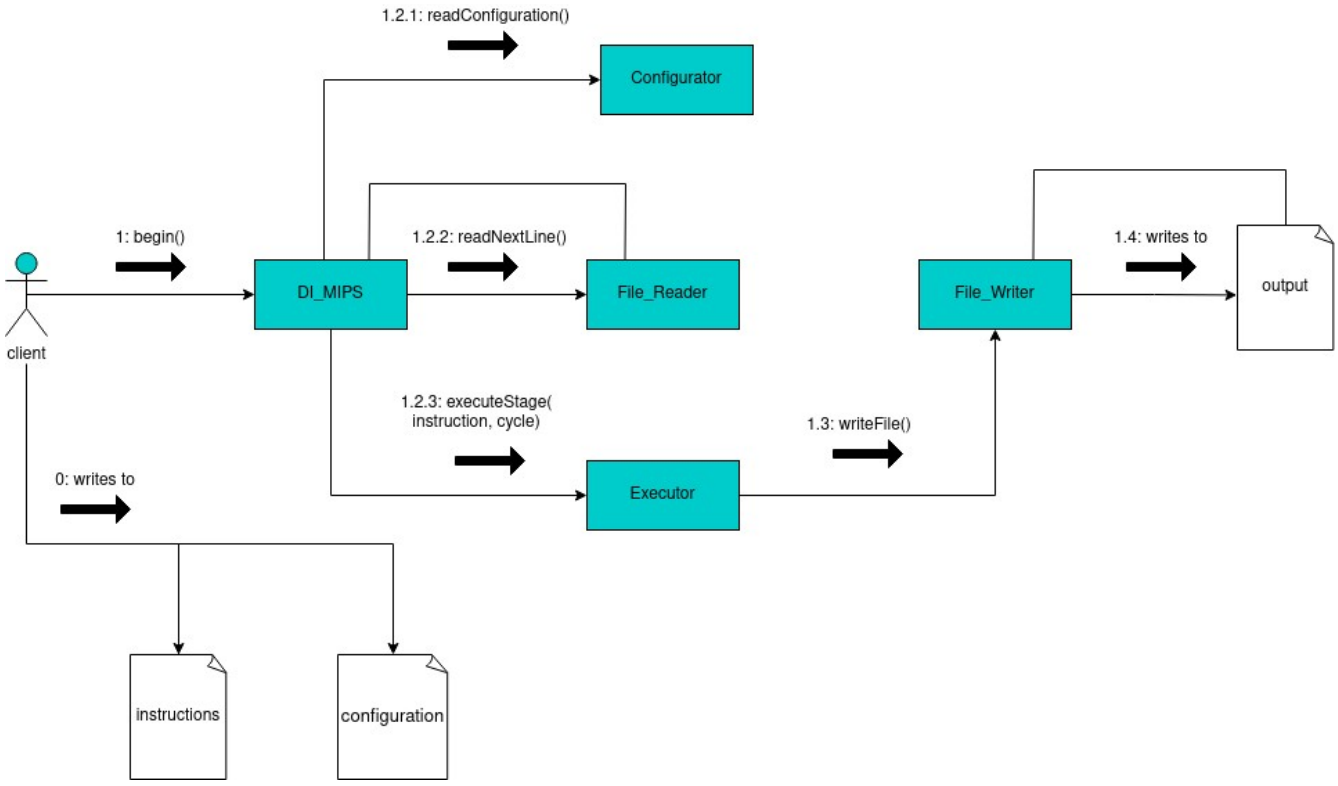

Figure 2 – Communication diagram for the different entities of the application

4. The Code

4.1 Reader

The File_Reader class is the one which manages the program's input, so the instructions written by the user at "instruction.txt" and the registers declared at "register.txt" can be transformed into objects that the program can interpret.

The File_Reader is called by the main class DI_MIPS. When we create an object of type File_Reader, the constructor of this class automatically calls the method *initRegisters()*, which will read "register.txt" and store the readden registers into the program's registers array so they can be used.

```
public File_Reader(String inst_f, Register[] registers) {
    this.inst_file = inst_f;
    try {
        br = new BufferedReader(new FileReader(reg_file));
        initRegisters(registers);
        br = new BufferedReader(new FileReader(inst_file));
    } catch (IOException e){
        System.err.println("Error reading file: " + e.getMessage());
    }

}
```

Figure 4.1.1 – Constructor for the class File_Reader. We can appreciate how Java's FileReader and method *initRegisters()* are called

The method *initRegisters()* reads the registers declarations from the file “register.txt” to initialize the array of registers. These registers can be changed by the user.

R0=0
R1=1
R2=2
R3=3
R4=4
R5=5
R6=6
R7=7
R8=8
R9=9
R10=10
R11=11
R12=12
R13=13
R14=14

R15=15

Example declarations. This is a possible content for the file “register.txt”.

Later, each time another class calls the public method *getNextInstruction()*, we will be reading and passing the instruction that is just at the next line. Using Java's FileReader method *readline()* on a bufferedreader object, we can read and analyze the lines one by one. The method *getNextInstruction* also analyzes if the line is an instruction or something else (label, comment, space between lines, etc.) If the line begins with an instruction operator, this method will return an object of type instruction. If it’s something else, it will keep skipping lines until it finds an instruction or the end of file.

Look if the line is an instruction:

```
Instruction inst = null;
while (inst == null) {
    if ((line = br.readLine()) == null) { // Read the String line
    System.out.println("End of file reached.");
    return null;
```

Figure 4.1.2 – Beginning of *getNextInstruction()*. We can see how *inst* is initialized as *null* and how we look if we reached the end of the file

Pass an instruction when detected:

```
if (isInstruction) {
    inst = new Instruction(numLin_inst, op, dst, src1, src2, Stages.F, label);
    numLin_inst++;
}
```

Figure 4.1.3 – Ending of *getNextInstruction()*. We can see how, after proving the line is actually an instruction, *inst* takes its values before being returned

To know when a line is or not an instruction, *getNextInstruction()* looks for the opcode. This program is capable to detect the next kinds of instructions:

TYPE	OPCODE	PARAMETERS	EXAMPLE
R-Type	ADD	dst, src1, src2	ADD R0, R1, R13
	SUB	dst, src1, src2	SUB R2, R0, R14
Load/Store	LD	dst, src2	LD R2, (R15)
	SW	src2, dst	SW R2, (R12)
Branches	BEQ	src1, src2, label	BEQ R0, R2, my_label

Figure 4.1.4 – Table of explanation of the different kinds of instruction and its use

s

```
ADD R0, R1, R13
SUB R2, R0, R14
LD R2, (R15)
label
SUB R0, R0, R14
BEQ R0, R2, label
SW R2, (R12)
```

Figure 4.1.5 – Example code. This is a possible content for the file “instruction.txt”

Pay attention to the label seen in the BEQ instruction. There is a method called findBranchLine(String label) that is used, as its name suggests, to find at "instruction.txt" the label passed by parameter, placing the pointer to read the next instruction in the line just after the label so the program can continue normally. It is **important** that the labels and instructions are in different lines so this method can detect the labels properly.

```
public boolean findBranchLine(String label){
    try {
        br = new BufferedReader(new FileReader(inst_file)); // We read the file from the beginning
        while ((line = br.readLine()) != null) {
            if (line.equals(label)) {
                System.out.println("Label found.");
                break;
            }
        }
        if (line == null) { // End of file reached (not found)
            System.out.println("End of file reached and label not found.");
            return false;
        }
        // if we arrive here, line = label
    } catch (IOException e) {
        System.err.println("Error reading file: " + e.getMessage());
    }
    return true; // line = label
}
```

Figure 4.1.6 – Method *findBranchLine()*. We can see how it restarts the reading of the document until it finds a line which is exactly the searched label

4.2 Executor

4.3 Writer

The File_Writer is the class that forms the core of the third logical part, the Output Printing from figure 1. 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();
    }
}
```

Figure 4.3.1 – File_Writer public constructor

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;
```

Figure 4.3.2 – File_Writer data collections

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);
        }
    }
```

Figure 4.3.3 – File_Writer's point of connection with the Execution Calculation Layer. This is where the data manipulation for the output starts

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();
    }
}
```

Figure 4.3.4 – File_Writer's function that executes the writing of the output file

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();
}
```

Figure 4.3.5 – A deeper look at the responibility distribution of the private functions to format the output file

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();
}
```

Figure 4.3.6 – Function to construct the header for the output file

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();
        }
    }
}
```

Figure 4.3.7 – File_Writer’s function to directly write to file

5. Example Of Use

6. Further ideas

As final words, the potential of this project is not limited to its current version. A lot more can be implemented and changed to enchance and develop more complex functionality. For example, to convert this into a real-world project, an idea would be to make this project fool-proof by adding validations, error messaging that directly informs the user of the current state of the system, prettyfy the code, unify the project build process and automize it. For now, its state conforms with the academic use for an end-of-semester group work for the subject of Computer Architectures in UIB. There is enough space for improvements and new ideas. It’s latest form is a public project on GitHub, whose address can be accessed at github.com//di-mips.