Visual RISC V Simulator

Progress report

Noah Hollowell
Department of Computer Science
University of Warwick

1 Introduction

Within the understanding of how a modern processor works we often look into the underlying instructions and physical operations it performs. However, often being able to visualise those are far more beneficial to our understanding than simply emulating the physical code to produce a result.

RISC-V is a new open source instruction set that has numerous emulators available online with a varying degree of visual elements. These all lack the ability to fully visualise the internal operations during each instruction execution. Thus, the aim is to produce a fully fledged RISC-V simulator that not only emulates RISC-V code but also visually demonstrates the flow of data during execution all the way down to data moving across the processor to their respective components updating in real-time.

2 Objectives

So far, the project has progressed along nicely, with a few objectives now being complete, or . Later objectives remain relatively unchanged.

Over the past 8 weeks the following objectives have been completed:

- Emulation of the base instruction set: RV32I [10], as per the RISC-V specification,
- Partial lexical and semantic analysis of code
- The start of the visualisation process for animating instructions including:
  - Speed control,
  - Manipulation of multiple values simultaneously.

All other objectives below remain in a to-do state with the visualisation objectives currently holding the most importance to produce the desired minimval viable product as mentioned in the specification [5]: (New additions/changes are in *italics*)

1. **(M)** A lexical analysis of inputted RISC-V code,
2. **(S)** A semantic analysis of inputted RISC-V code,
3. **(C)** *Loop control via labelled functions*
4. **(S)** Emulation of the "Standard Extension for Integer Multiplication and Division" instruction set,
5. **(A)** Emulation of the "Standard Extension for Single-Precision Floating-Point" instruction set
6. **(M)** A visualisation of the processors register set,
7. **(M)** A visualisation of the memory holding both the instructions and data or just data,
8. **(M)** A comprehensive system allowing for the display of data moving around the processor including:
   (a) **(M)** Numerical data moving from memory to other components and vice-versa,
   (b) **(S)** Addressing requests,
   (c) **(C)** Manipulation of multiple values simultaneously to simulate the effect of processor operations (e.g. addition, subtraction, shifts, etc),
9. **(C)** Control *the* steps of the simulation,

To further help with development objectives have been extended to make use of the MoSCoW prioritisation method [2] (Must, Should, Could, Would) and have prefixed each objective with the respective capital.

Furthermore, the allocation of priorities matches with our timetable, in which the majority of **Must** items are due to be finished by the end of this term, with items such as 4. coming at the start of term 2.

3 Research

Building a simulator for a instruction set from the ground up is no easy task. There are multiple ways to approach building both the emulation and visualisation parts of the simulator. With regards to the visualisation, after researching more into JavaFX [6] we determined it to be a suitable choice after further experimentation into its abilities and how they would fit the the projects needs. A suitable fail-safe was also determined in the form of a web-based interface due to having an extensive history of web development and design.

Research is focused on understanding the RISC-V RV32I [10] instruction set and how each instruction individually operates. RISC provides an open-source reference card [7] which provides information about each instruction including its operands and instruction type, which when coupled with the RISC-V specification [10] provides comprehensive coverage of exactly how to implement each instruction according to the specification.

In order to develop the visual end of the project research was done into existing solutions. A previous solution is Cornell's Online RISC-V Interpreter [9], which provides a comprehensive interface to show of registers and memory as instructions execute either instantly, or in a stepped format. However, it lacks any form of animated view of a processor, which this project aims to build upon. It is also invaluable to have a robust comparison available to cross-reference instruction operations and devise working tests, or than purely using the RISC-V specification. Another useful system is LittleManComputer [4]. It is a simple instructional model of a computer, with a limited instruction set. However, unlike Cornell's it does display the movement of data around the processor, providing a base idea for how the project can animate the flow of data in a clear and comprehensive manner.

4 Progress

The project is currently in a very healthy state, with it being on-track to complete the minimal viable product for, or just after the start of the Christmas break.

The project has been split up into two distinct sections: **Emulation** and **Visualisation**. This decision was made to allow decoupled development, providing the benefit of working exclusively on emulation for the first 5 weeks to ensure the core of the project is stable, reliable and robust. Only then, can work start to build upon the visual aspects which require the core to build upon its functionality into a fully fledged application whilst still being able to be developed independently in terms of GUI design and layout.

4.1 Emulation

Core to emulating RISC-V is its set of 32 unique 32-bit registers. These are core to performing operations on values during execution without having to constantly move data to-and-from memory.

4.1.1 Registers

Registers are implemented as a Map relating each registers name and alternate name to a `Register` instance that holds it value. For easy access each Register provides 3 ways to read out the data: As binary, as hexadecimal and as denary. Instead of writing 3 write methods directly on the register the decision was to have 3 write methods in the `RegisterSet`, a class manager that stores all 32 register instances. This class mainly handles reading and writing binary to the register with reading having 3 unique methods to retrieve data in specific formats (Binary, Hexadecimal and Denary) and 3 methods to write, which internally convert to binary and call the third write method that directly writes binary to the register.

4.1.2 Memory

Memory follows a similar suit to register, but deviates within that the memory consists of far more than 32 cells. As such an implementation very similar to the registers was used, except `MemoryCells` (instances that store the value in memory) are on the fly as they are needed. This helps keep the overhead down compared to pre-generating say 2000 memory cells and having to constantly search them for the required value. Further, with the usage of a Map to store memory cells, access time was also cut down significantly due to the constant O(1) search time compared to storing cells in a list which incurs a linear O(n) search times, which would become noticeable for massive memory operations within the program.

4.1.3 Instructions

A core part of the project is to be able to take in a user written set of instructions and convert them into operable instructions the the program can execute. Within this a very basic lexer has been produced, that allows us to convert written instructions into instances of the `Instruction` class.

The implementation of this relies heavily on a base abstract class (Listing 1) having an abstract `execute(InstructionOperands operands)` method that is then implemented for each instruction within its own implementation (Listing 2).

An instance of `InstructionOperands` is then passed to the execute method, with `InstructionOperands` simply being an object holding up to three strings representing the operands of the current instruction.

To improve the usability and extend-ability of the project Instructions will automatically register themselves. Thus, any class that extends `Instruction` will be automatically found and loaded during run-time and then be available to be used in any user given program. This is performed by a library called Reflections [8] which "scans and indexes your project's classpath metadata, allowing reverse transitive query of the type system on runtime" allowing us to perform the dynamic loading of classes during runtime.

```java
package uk.co.noahdhollowell.emulator.instructions;

import uk.co.noahdhollowell.emulator.exceptions.InvalidValueException;
import uk.co.noahdhollowell.emulator.memory.InvalidMemoryAddressException;
import uk.co.noahdhollowell.emulator.memory.Memory;
import uk.co.noahdhollowell.emulator.memory.UnalignedMemoryAccessException;
import uk.co.noahdhollowell.emulator.registers.ProgramCounter;
import uk.co.noahdhollowell.emulator.registers.RegisterNotFoundException;
import uk.co.noahdhollowell.emulator.registers.RegisterSet;

public abstract class Instruction {

    private final String identifier;
    public final RegisterSet registers;
    protected final Memory memory;
    protected final ProgramCounter programCounter;
    private final int expectedOps;

    public Instruction(String identifier, int expectedOps) {
        this.identifier = identifier;
        this.registers = RegisterSet.getRegisters();
        this.expectedOps = expectedOps;
        this.memory = Memory.getMemory();
        this.programCounter = ProgramCounter.getProgramCounter();
    }

    public Instruction(String identifier) {
        this.identifier = identifier;
        this.registers = RegisterSet.getRegisters();
        this.expectedOps = 3;
        this.memory = Memory.getMemory();
        this.programCounter = ProgramCounter.getProgramCounter();
    }

    public int getExpectedOps() {
        return expectedOps;
    }

    public String getIdentifier() {
        return identifier;
    }

    public abstract void execute(InstructionOperands operands) throws RegisterNotFoundException,
          InvalidValueException, InvalidMemoryAddressException, UnalignedMemoryAccessException;

    @Override
    public String toString() {
        return "Instruction{" +
                "identifier='" + identifier + '\'' +
                '}';
    }
}
```

Listing 1: An abstract class representing the absolute minimum of an instruction that must be extended

```java
package uk.co.noahdhollowell.emulator.instructions.rv32i;

import uk.co.noahdhollowell.emulator.exceptions.InvalidValueException;
import uk.co.noahdhollowell.emulator.instructions.Instruction;
import uk.co.noahdhollowell.emulator.instructions.InstructionOperands;
import uk.co.noahdhollowell.emulator.registers.RegisterNotFoundException;

public class ADD extends Instruction {

    public ADD() {
        super("ADD");
    }
```

```
    @Override
    public void execute(InstructionOperands operands) throws RegisterNotFoundException, InvalidValueException {

        // Load rs1 and rs2
        int rs1 = this.registers.readInt(operands.getR1());
        int rs2 = this.registers.readInt(operands.getR2());


        // Write ADDED value back to destination (rd)
        this.registers.writeInt(operands.getDest(), rs1 + rs2);
        programCounter.increment();

    }
}
```

Listing 2: An implementation of the abstract Instruction class for the ADD instruction.

4.1.4 Parsing & Running

Now that all the main pieces are ready progress can begin on building them up from a text input to being emulated.

Currently, a very basic parser operates by splitting an input line by line and then by splitting the lines. Instructions are expected to appear in multiple forms, mainly being of the form `OPCODE rd, rs1, rs2` (R type), in which `rd` & `rs#` are destination and target registers respectively. Deviation from this format occurs as RISC-V employs multiple instruction formats (`R, I, S, B, U`...)[10] to allow for different size values to be entered. For example the `ADD` instruction is type R in which each operand is expected to be a register. Whereas for `ADDI` the last operand is a 12-bit immediate, taking a signed values between -2048 and 2047. To facilitate the occurrence further of instructions that take less operands the number of operands in each `Instruction`'s constructor must be specified with a default of 3. This way, the program runner can quickly check the amount of operands during parsing and reject any that have to many to to little for the respective instruction.

A slight difficulty in parsing instructions is related to load and store instructions as these require a specific format to select memory registers to allow for more complex addressing. For all types, they follow the format `offset(register)`, which allows continuous data storage in memory (for example: an array) but

also allows writing to any memory location (by using the zero register). However, within this the offset can be represented as both hexadecimal and denary so an method to determine which type is present in order to parse both and produce the same output at the end while being flexible and extendable in the future.

The solution was to encapsulate them into a `MemoryAddress` abstract class with two implementations (one for denary and one for hex) which both use a regex to parse out the offset and register and then combine them to return an integer memory location that the program can use. By doing this the memory controller can alter memory addressing without having to modify any of the memory read/writing code, and in turn the reading and writing code for memory only cares about receiving an instance of MemoryAddress, avoiding having duplicate methods just to allow denary or hex addressing.

Finally, once all the instructions are parsed they are bundled up into a program instance which can then be executed. This is done in a iterative manner in which each instruction is executed, then the program counter is incremented (or jumped for branching), then the next instruction is fetched based on the program counters value and executed. This occurs until the program counter doesn't point to an instruction at which point execution has finished. At this point the memory and registers are printed to the console allowing us to verify operation and also see the values.

4.2 Visualisation

After Week 6 development switched from emulation to visualisation. This began with a rough mock-up (Figure 1) of the interface, and then an actual implementation of the interface using JavaFX (Figure 2)

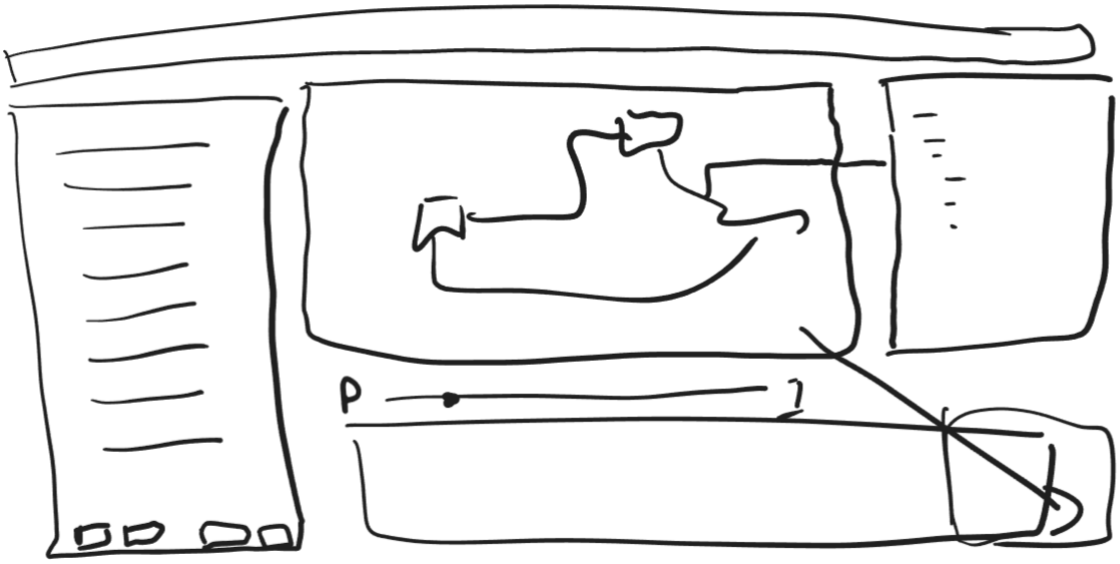

Figure 1: Draft Interface Layout

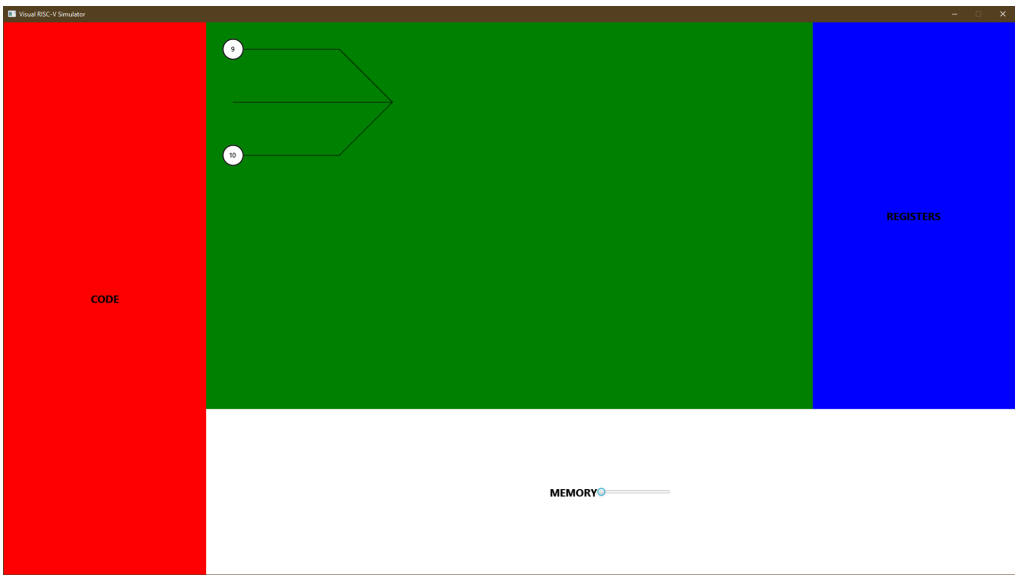

Figure 2: Draft layout implemented in the application

The design of the interface as seen in Figures 1 & 2 was determined based on usage of previous online interpreters such as LittleManComputer [4] and the Cornell RISC-V Interpreter [9] which adopt a common layout of code input being on the left, and visual elements being on the right, centre and below. The layout prominently orients around the main central section in which the animated content will occur showing movement around the chip, with a larger section below to display memory as it read and written, with a smaller registers section on the right which will contain a fixed listing of the 32 registers, with active registers being highlighted.

After, development of the animation component began utilising JavaFX [6]. JavaFX provides a path element to draw paths for us. However, implementing a custom version integrates better with the projects needs and removes unnecessary options that the path element provides. This, allows us to have a very simple pathing structure, whilst also being to directly animate on-top of the path without having to pull data out of the JavaFX path class and the process, instead having direct access to the nodes that makeup the path.

Within the custom paths, the `Line` object is used to draw paths onto the Scene (JavaFX denotes an open application window a Stage, that a Scene is then placed upon that contains our entire interface and its moving parts) as seen in Figure 2. Now, by utilising the `TranslateTransition` and `SequentialTransition` classes provided by JavaFX we can effectively animate data movement around the scene by utilising the path nodes to build up a transition per set of nodes to simulate following that path, using an overarching Sequential Transition per path to wrap all the transitions and play them in the correct order linearly to produce the visual effect of an element perfectly following the path.

To ensure consistency within animations an `Animator` class was developed, which encapsulates a set of stages that represent elements of an entire animation sequence. This starts with an abstract AnimatorStage class which provides two important methods:

- **play**: Runs the given animation implementation
- **after**: Calls the parent Animator class to run the next stage if there is one

With this abstraction other stages can be implemented such as changing text and interpolating to a fixed point.

As for our path animations, each calls `after()` once they have finished execution allowing the animator to call the next animation or call the animators finished callback, allowing us to run something after all the animation stages have finished.

Thanks to this level of abstraction, complex animations can be easily built up, traversing paths, moving to points or changing text at whim, with the option to easily add new animation options in the future with little hassle, with JavaFX taking care of transforming objects in the scene leaving us to just handle ordering and callbacks to ensure continuous operation.

4.2.1 The problem

Animating was not without is problems. JavaFX provides multiple ways to transition in its `TranslateTransition` class, mainly `setToX` and `setByX` (and respective Y variants) which proved problematic due to a lack of documentation as to how these actually transform the given element on execution.

After researching JavaFX's documentation and online resources, it was discovered that JavaFX tracks elements via coordinate positions, however when you apply a transition its base coordinates remain unchanged, but instead they are transformed by the transition provided. These transition coordinates then remain and are stacked upon by further transitions, meaning each transition is relative to the previous.

Originally using `setToX` caused major issues as this would result in the transition playing, but not relative to the previous transition causing the element to fly off-screen and never follow the line. The solution was to use `setByX` which moves the element relative to the previous transition producing the expected outcome of the element correctly following the path.

5 Project Management

Currently, the project has been on track following the set out timetable. However, I foresee that the project may run up to a week late with content due to the end of term being busy with Christmas events, competitions and moving back home for the break. Yet, this was factored in from the start with the Christmas break being intended as a buffer for any delays as well as small bits of extra work to be completed.

Supervisor meetings started as weekly meetings, but have drifted to being more fortnightly. This is mainly down to personal schedule changes and occasionally a lack of material to discuss due to parts of development yielding successful but rather dull material to cover.

Git and Github have remained for version control, making use of the builtin workflows to automate testing in a separate environment, which has helped lead to the discovery of unintended bugs. Branching has also been an indispensable to the project allowing separation of new features and content during development, allowing us to isolate development and merge complete features back together once complete. Also, as discussed in the specification DCS is being used as an alternate remote as an extra backup in-case anything were to happen to Github or my local machine. Thus, reducing the risk of total code loss which would be fatal to the project.

The Agile [1] methodology is still the main approach, with some plan driven aspects. However, the adopted methodology is mixture of agile and personal choices as rather than following a specific methodology. However, we have made good use of a backlog type system, in the form of a to-do list on Notion which has made tracking extra items to accomplish or fix on later dates much easier as-well as noting information about how systems should work.

Testing is still being fronted with test-driven development. However, this is problematic for visual elements due to the necessity for a physical person to need to watch the animations play to ensure they look correct which isn't possible in a test-drive scenario, thus visualisation tests are being performed in parallel with development.

5.1 Risks

No new risks have arisen during the term.

Because of the successful progress of the project the desired minimal viable product as set out in the specification is going to be completed for the Christmas break, giving a basis for feedback and further development.

Of our original risks, they remain unchanged, or resolved as mentioned above.

5.2 Timetable

The projects timetable given in the specification [5] remains unchanged.

6 Legal, Social, Ethical & Professional Issues

Social, Ethical and Professional issues remain unchanged.

A decision to change the JDK used for the project incurs a slight legal change. The choice is due to how JavaFX is loaded into the project, and if it should be incorporated as part of the JDK or separately incorporated into the final application JAR manually. The choice of using a JDK with JavaFX already incorporated would mean using a more lax licensed JDK, which is beneficial to the project, with the majority of JDKs including JavaFX being . Alternatively, JavaFX can be incorporated manually and its own licensing will be followed along with Oracle's OpenJDK [3] licensing.

References

[1] Atlassian. *What is Agile? | Atlassian*. Atlassian, 2022. URL: https://www.atlassian.com/agile (visited on 07/10/2022).

[2] Agile Business. *Chapter 10: MoSCoW Prioritisation*. Agilebusiness.org, 2022. URL: https://www.agilebusiness.org/dsdm-project-framework/moscow-prioririsation.html (visited on 18/11/2022).

[3] Oracle Corporation. *OpenJDK*. Openjdk.org, 2022. URL: https://openjdk.org/ (visited on 07/10/2022).

[4] Peter Higginson. *Little Man Computer - CPU simulator*. Peterhigginson.co.uk, 2014. URL: https://peterhigginson.co.uk/lmc/ (visited on 22/11/2022).

[5] Noah Hollowell. *Visual RISC V Simulator Project specification*. Oct. 2022.

[6] Sun Microsystems. *JavaFX*. Openjfx.io, 2022. URL: https://openjfx.io/ (visited on 07/10/2022).

[7] RISC-V. *RISC-V Reference Card*. URL: https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/files/docs/RISCVGreenCardv8-20151013.pdf (visited on 18/11/2022).

[8] ronmamo. *ronmamo/reflections: Java runtime metadata analysis*. GitHub, Aug. 2022. URL: https://github.com/ronmamo/reflections (visited on 22/11/2022).

[9] Cornell University. *RISC-V Interpreter*. Cornell.edu. URL: https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/ (visited on 18/11/2022).

[10] Andrew Waterman and Krste AsanoviÄ. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA Document Version 20191213*. 2019. URL: https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf (visited on 07/10/2022).