The American University in Cairo

Computer Organisation and Assembly Language Programming

Fall 2024
Department of Computer Science and Engineering

Project 1 Report: RISC-V RV32I Simulator

Authors:

Name: Ahmed Ayman Elkhodary
ID: 900213472
Email: aae121@aucegypt.edu

Name: Ahmed Fawzy Abdelkader
ID: 900222633
Email: ahmed-fawzy@aucegypt.edu

Name: Omar Leithy
ID: 900221663
Email: omarleithym@aucegypt.eduu

Supervised by:
Dr. Cherif Salama

GitHub Repository:
https://github.com/Ahmed-Fawzy14/RISC-V-RV32I-Simulator/tree/main

Contents

1 Introduction **2**
1.1 Project Overview . . . 2
1.2 RISC-V ISA Background . . . 2

2 Implementation **2**
2.1 Simulator Design . . . 2
2.2 Instruction Handling . . . 2
2.3 User Input and Output . . . 2

3 Bonus Features **3**
3.1 GUI Implementation . . . 3
3.2 Output in Hexadecimal and Binary . . . 3

4 Testing and Results **3**
4.1 Test Programs . . . 3
4.2 Results . . . 3

5 Known Bugs and Limitations **4**
5.1 Bugs . . . 4
5.2 Limitations . . . 4

6 Design Decisions and Assumptions **4**
6.1 Program Counter Initialization . . . 4
6.2 Sparse Memory Model . . . 4

7 User Guide **5**
7.1 Development Environment . . . 5
7.2 Compiling and Running the Simulator . . . 5
7.3 Loading Test Cases . . . 5
7.4 Test Cases Used . . . 5
7.5 Example Simulation Process . . . 6

8 Experience and Reflection **6**

A Screenshots of GUI and Simulation Outputs **6**

1 Introduction

1.1 Project Overview

The RISC-V RV32I Simulator is designed to simulate the RV32I instruction set. This allows us to execute assembly code and observe the behavior of a RISC-V-based CPU through various register and memory operations.

1.2 RISC-V ISA Background

RISC-V is an open, modular ISA commonly used for educational purposes. This project implements the RV32I 32-bit integer instruction set, excluding specific instructions (e.g., ECALL, EBREAK), to better understand low-level CPU processes.

2 Implementation

2.1 Simulator Design

Our simulator is divided into key components:

- **Instruction Decoder:** Decodes each instruction, identifying operation codes and registers.
- **Program Counter:** Tracks the current instruction's address.
- **Register File:** Holds 32 general-purpose registers, with Register 0 hardcoded to zero.
- **Memory Model:** Uses a sparse data structure to track only relevant memory addresses.

2.2 Instruction Handling

We implemented all 42 RV32I instructions, handling each as specified, except for ECALL, EBREAK, PAUSE, FENCE, and FENCE.TSO. These halt the simulation by freezing the program counter.

2.3 User Input and Output

Users can input assembly programs via text files or interactive entry. Outputs display the program counter, register values, and memory states after each instruction, allowing real-time tracking.

3 Bonus Features

Our project includes two bonus features:

3.1 GUI Implementation

A graphical user interface (GUI) was created using Python's 'Tkinter' library to simplify user interaction. Key features of the GUI include:

- **Program Loading**: The GUI includes a file dialog for easy program loading. This required changes in the backend, so loaded programs initialize memory correctly within the GUI.
- **Interactive Controls**: Step-by-step execution buttons allow users to view changes in registers and memory for each instruction.
- **Visual Output**: Displayed outputs are arranged in a grid layout, presenting program counter updates, register values, and memory state changes in an organized manner.

3.2 Output in Hexadecimal and Binary

To provide clear insights, outputs are displayed in decimal, hexadecimal, and binary. This formatting applies to:

- **Program Counter**: Shown in all three formats for each instruction.
- **Registers**: Each register's content is updated and displayed in decimal, hexadecimal, and binary after every instruction.
- **Memory Addresses**: Memory locations updated by instructions are similarly presented in all three formats.

4 Testing and Results

4.1 Test Programs

Three programs were used for testing:

- Basic arithmetic operations.
- A program with branching and looping.
- Memory load/store operations.

These programs validate the simulator's handling of program counter updates, register modifications, and memory changes.

4.2 Results

The output after each test case verifies that the simulator executes instructions accurately. Screenshots (see Appendix) show outputs and the GUI layout for each test case.

5 Known Bugs and Limitations

5.1 Bugs

Key bugs encountered included:

- **Backend Changes for GUI**: Shifting from command-line to GUI required significant changes, which introduced challenges for capturing inputs and outputs within the GUI structure.
- **Memory Initialization**: Ensuring each bit aligns correctly was crucial for accurate memory initialization.
- **GUI Layout Issues**: Displaying text boxes and labels in the correct grid positions was challenging.
- **Instruction Splitting**: Errors in parsing instructions led to issues with decoding.
- **Program Counter Handling**: Ensuring the counter advanced or halted as expected was a repeated debugging point.
- **Branching and Jump Instructions**: Instructions like JAL, JALR, and other branch instructions occasionally failed due to unhandled labels or improper branch execution.

5.2 Limitations

This simulator does not support compressed instructions from the RV32IC instruction set or multiplication/division from the RV32IM set, limiting some test cases.

6 Design Decisions and Assumptions

6.1 Program Counter Initialization

The program counter initializes at the starting address specified by the user, allowing flexible program loading. This decision enables easy compatibility with different instruction sets.

6.2 Sparse Memory Model

To handle RISC-V's large 4GB address space, we used a sparse model to store only relevant memory locations.

7 User Guide

7.1 Development Environment

We developed and ran the simulator using PyCharm. This IDE facilitated code organization, debugging, and testing through its integrated file management and terminal.

7.2 Compiling and Running the Simulator

To compile and run the simulator, follow these steps:

1. Open the project in PyCharm.
2. Ensure all necessary libraries (e.g., `Tkinter` for GUI) are installed.
3. Run the main simulation file (e.g., `main.py`) in PyCharm to launch the GUI.

7.3 Loading Test Cases

For testing the simulator's functionality, we used a series of assembly instruction test cases in `.txt` files. Users can input the path to a test case file directly within the GUI, allowing the simulator to execute and display results for each instruction in sequence.

7.4 Test Cases Used

To thoroughly test the simulator, we designed the following test cases, each containing a set of assembly instructions targeting different instruction categories:

- `arithmetic_immed_test_case.txt`
- `arithmetic_test_case.txt`
- `branch_inst_test_case.txt`
- `branch_jump_test_case.txt`
- `jal_jalr_test_case.txt`
- `logical_test_case.txt`
- `memory_inst_test_case.txt`
- `set_less_than_test_case.txt`
- `shift_test_case.txt`
- `upper_immed_test_case.txt`

Additional complex test cases focused on branching and looping:

- Branch Instructions with Labels 1
- Branch Instructions with Labels 2
- Control Instructions
- Jump Instructions
- Mixed Instructions with Labels and Branching
- Test Loop

7.5 Example Simulation Process

To run a simulation with a specific test case:

1. Choose a test case from the list above and obtain its file path.
2. Enter the file path into the GUI's file input field.
3. Click `Run` to execute the loaded instructions.
4. Step through each instruction to observe real-time updates to the program counter, registers, and memory.

8 Experience and Reflection

The project taught us valuable lessons in low-level programming, exposure to new programming languages like Python, CPU architecture, and user interface design, as part of the Computer Organization and Assembly Language Programming course. Challenges included handling instruction decoding, memory management, and GUI integration, which required teamwork and iterative debugging to solve.

A Screenshots of GUI and Simulation Outputs

Note: please check our github repo video user guide for a more interactive and visual experience.

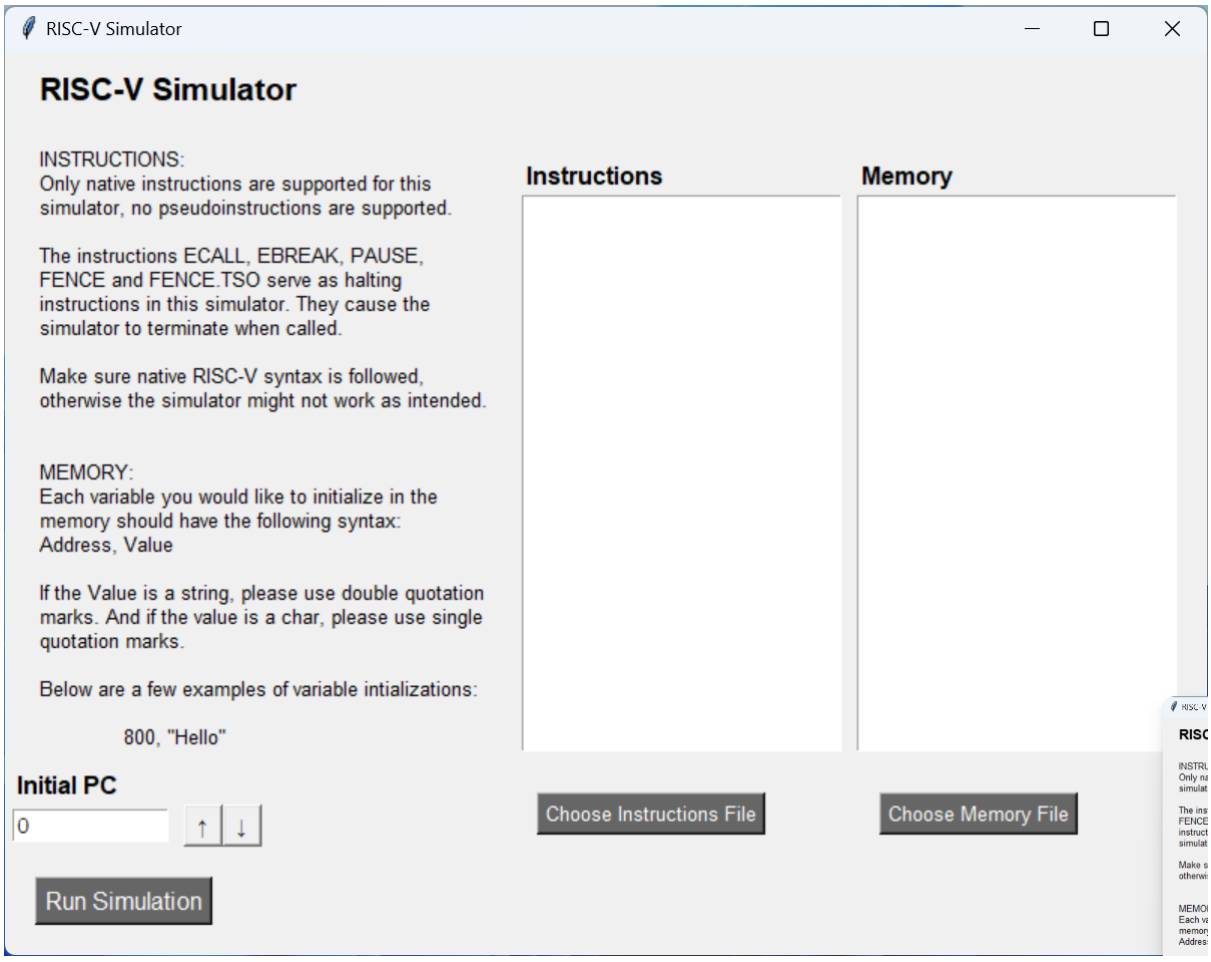
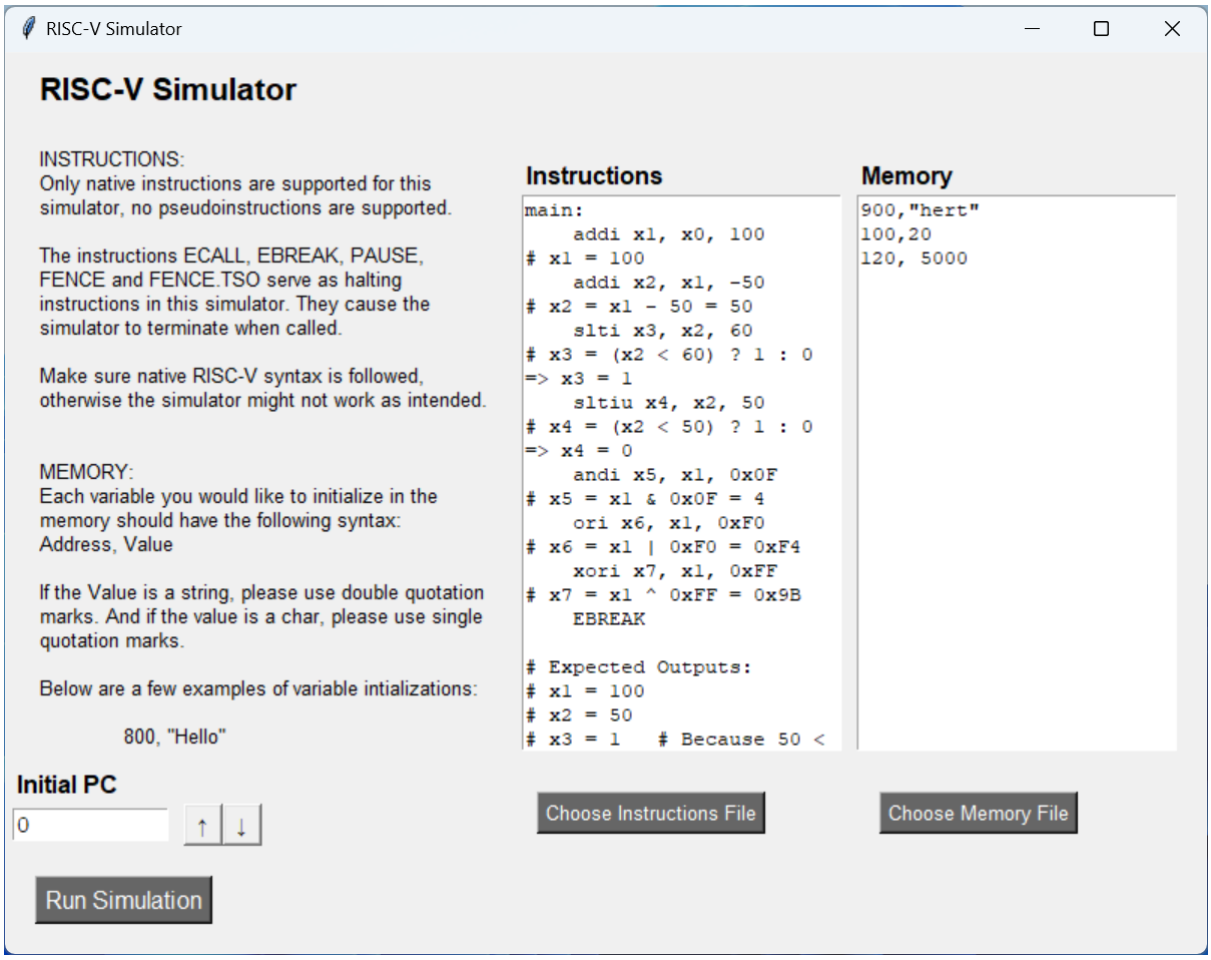


Figure 1: Simulation Welcome Window



Input your `Instructions` file and `Memory` file from your file directory. Specify your desired program counter (PC), then click `Run Simulation`.
For example, the instruction file used here is `arithmetic_immed_test_case.txt` with the starting PC set to 0.

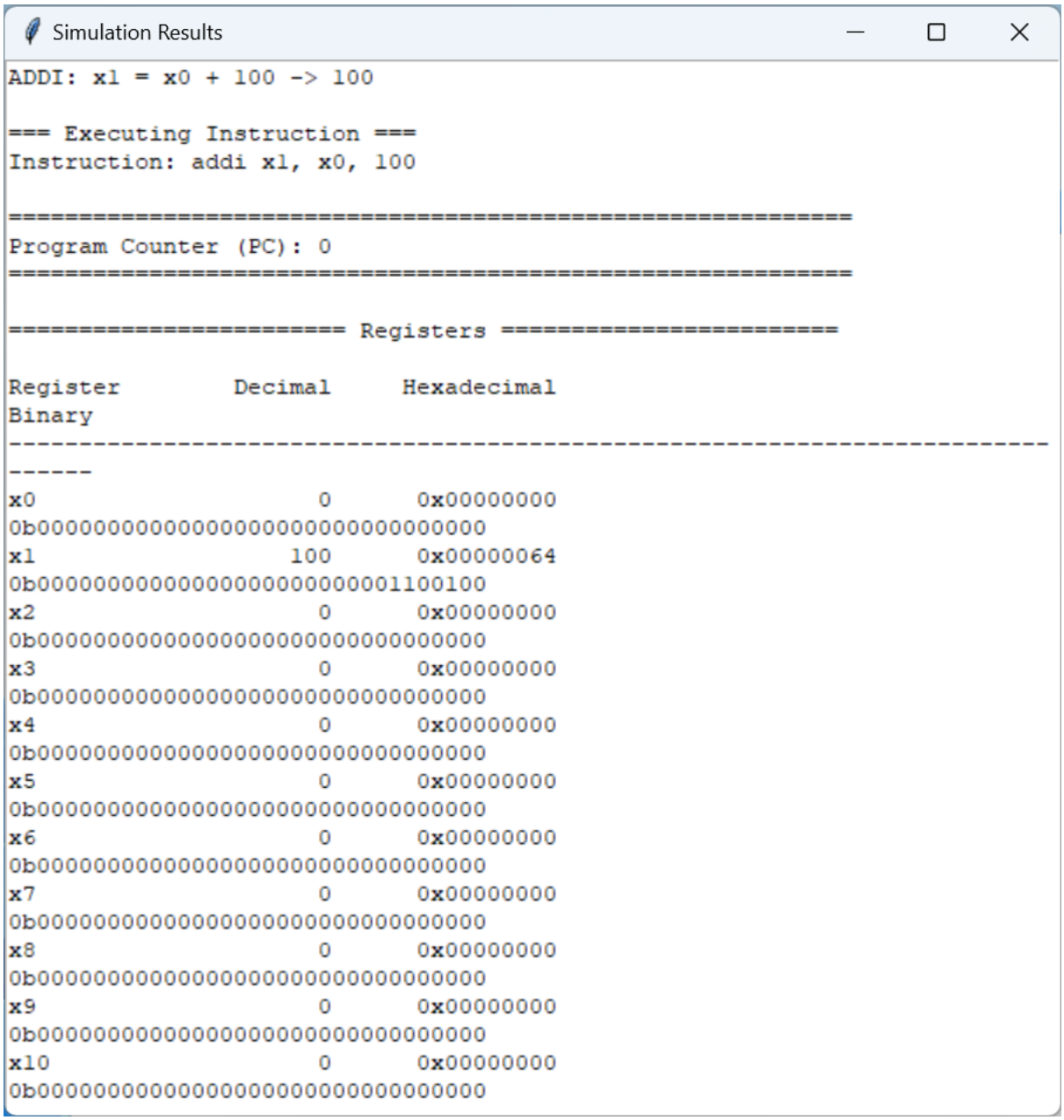


Figure 2: Simulation Result of `arithmetic_immed_test_case.txt` with the starting PC set to 0.

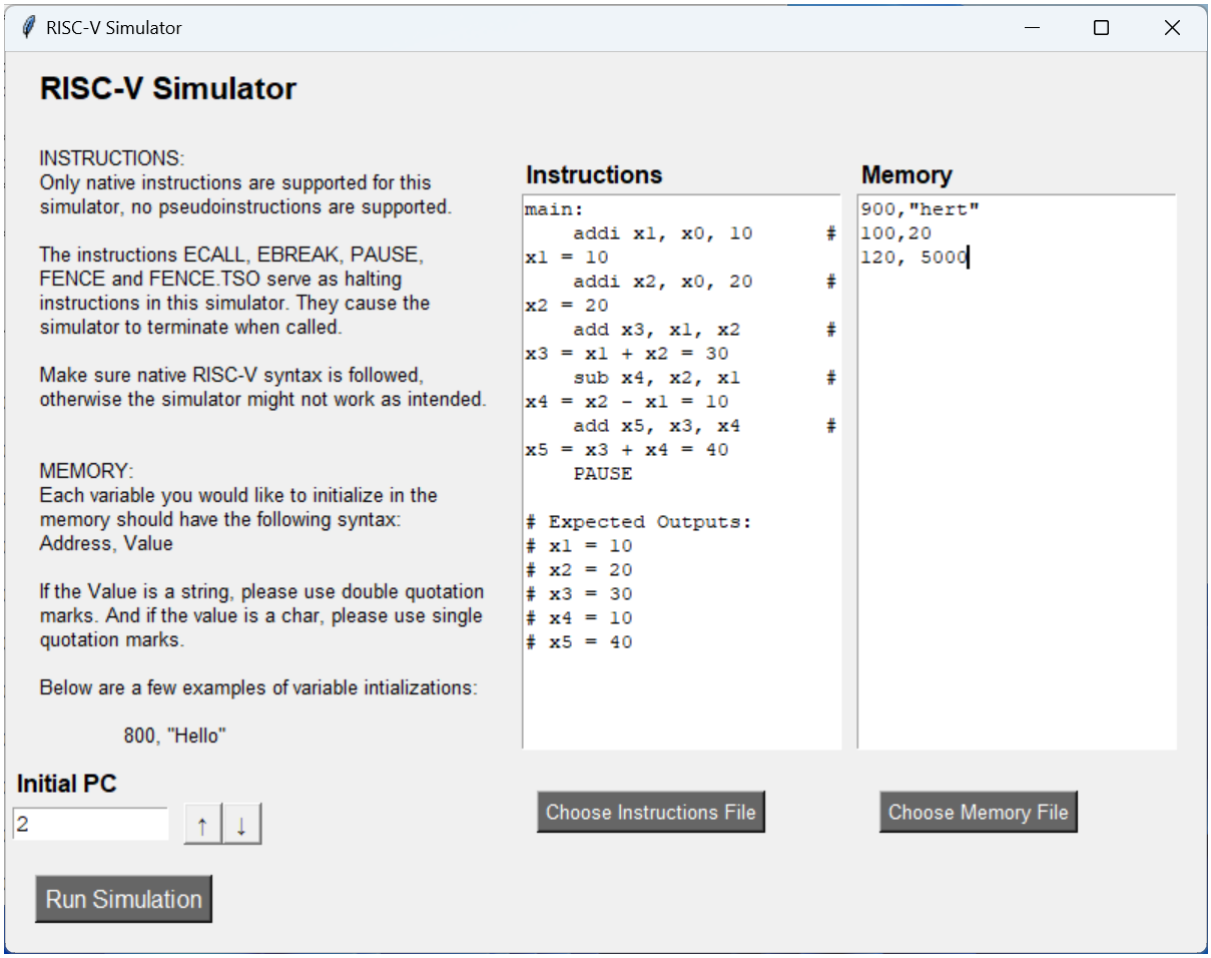


Figure 3: The instruction file used here is `arithmetic_test_case.txt` with the starting PC set to 2.

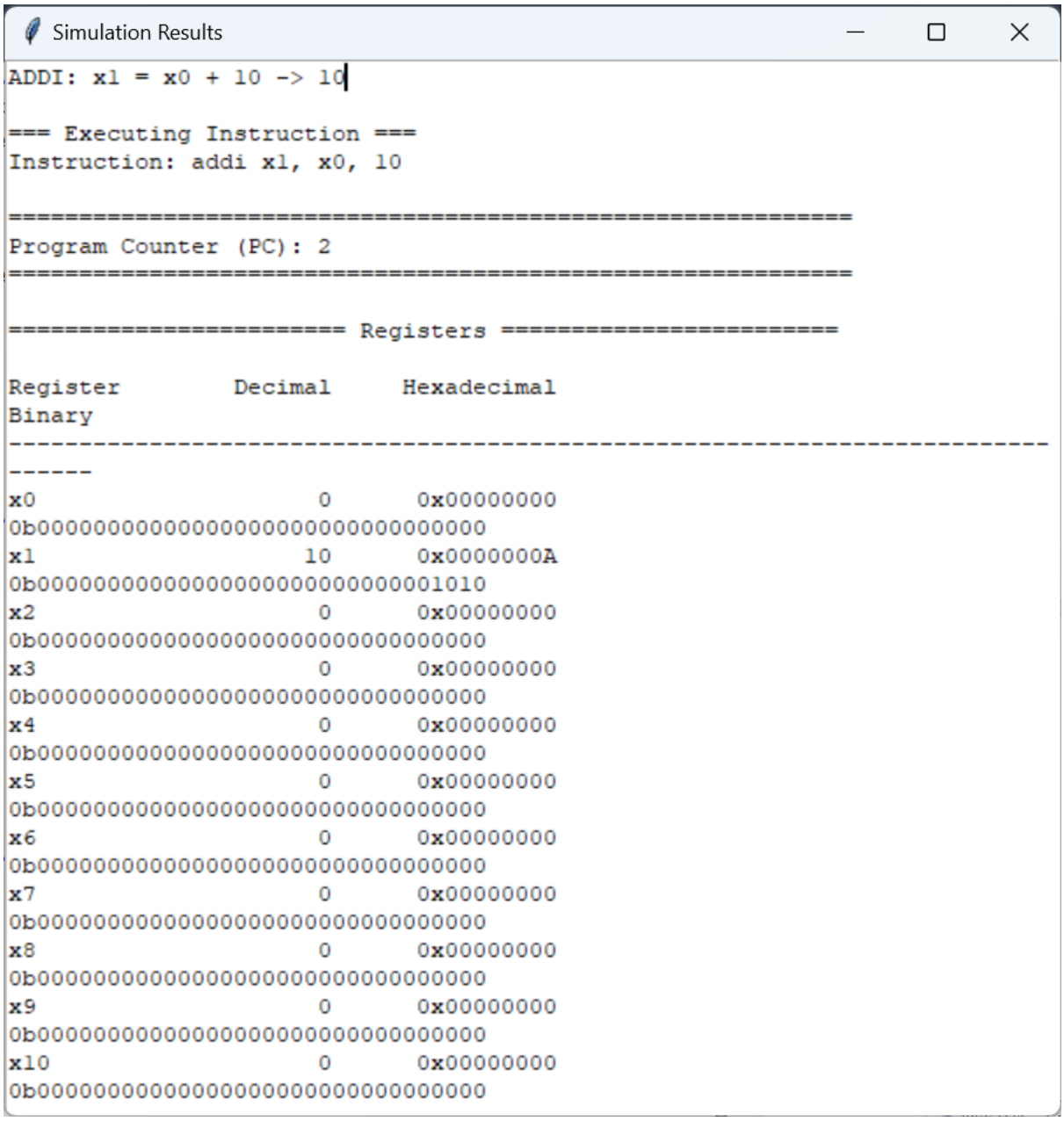


Figure 4: Simulation Result of `arithmetic_test_case.txt` with the starting PC set to 2.

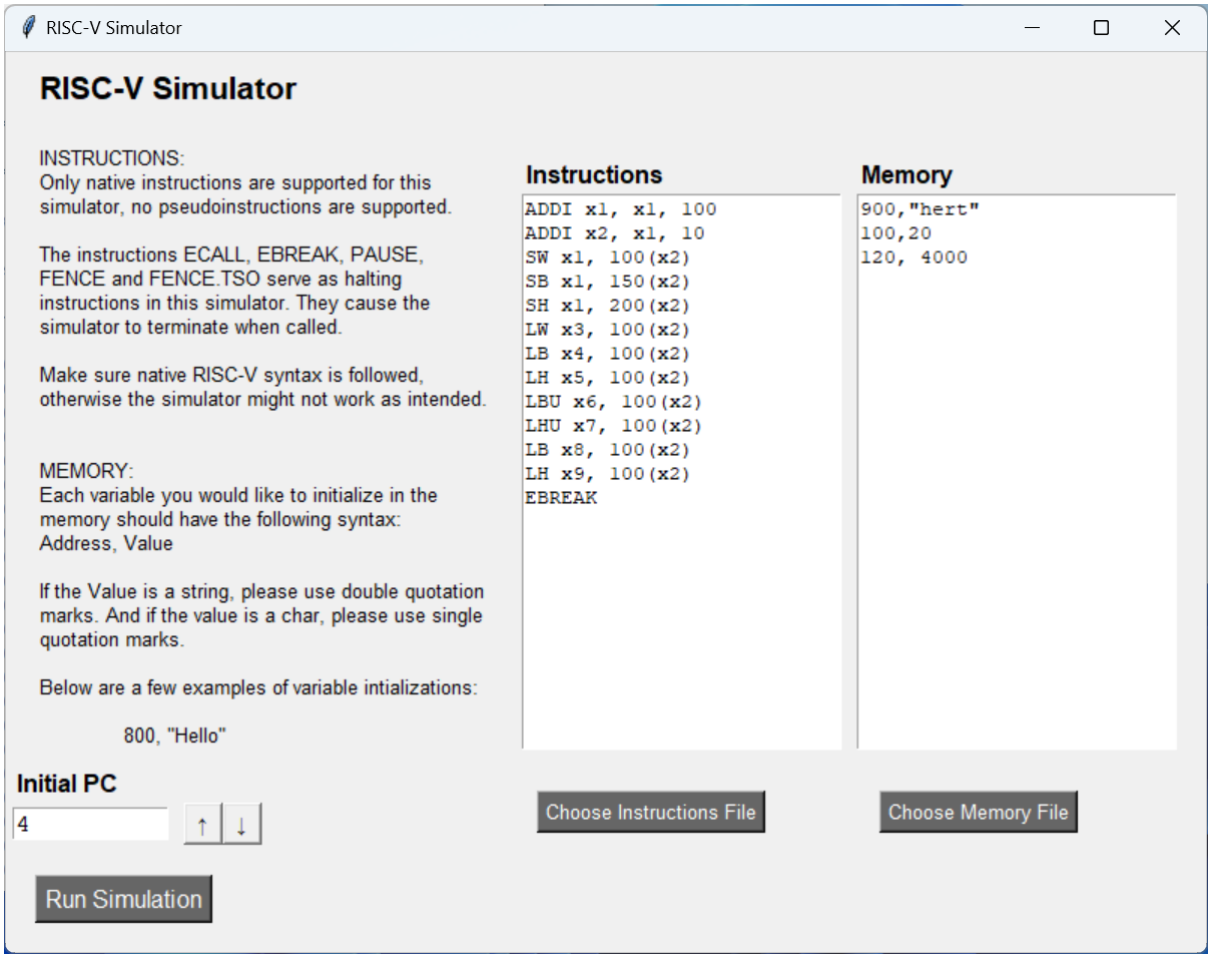


Figure 5: The instruction file used here is `memory_test_case.txt` with the starting PC set to 4 and memory = 4000 MB.

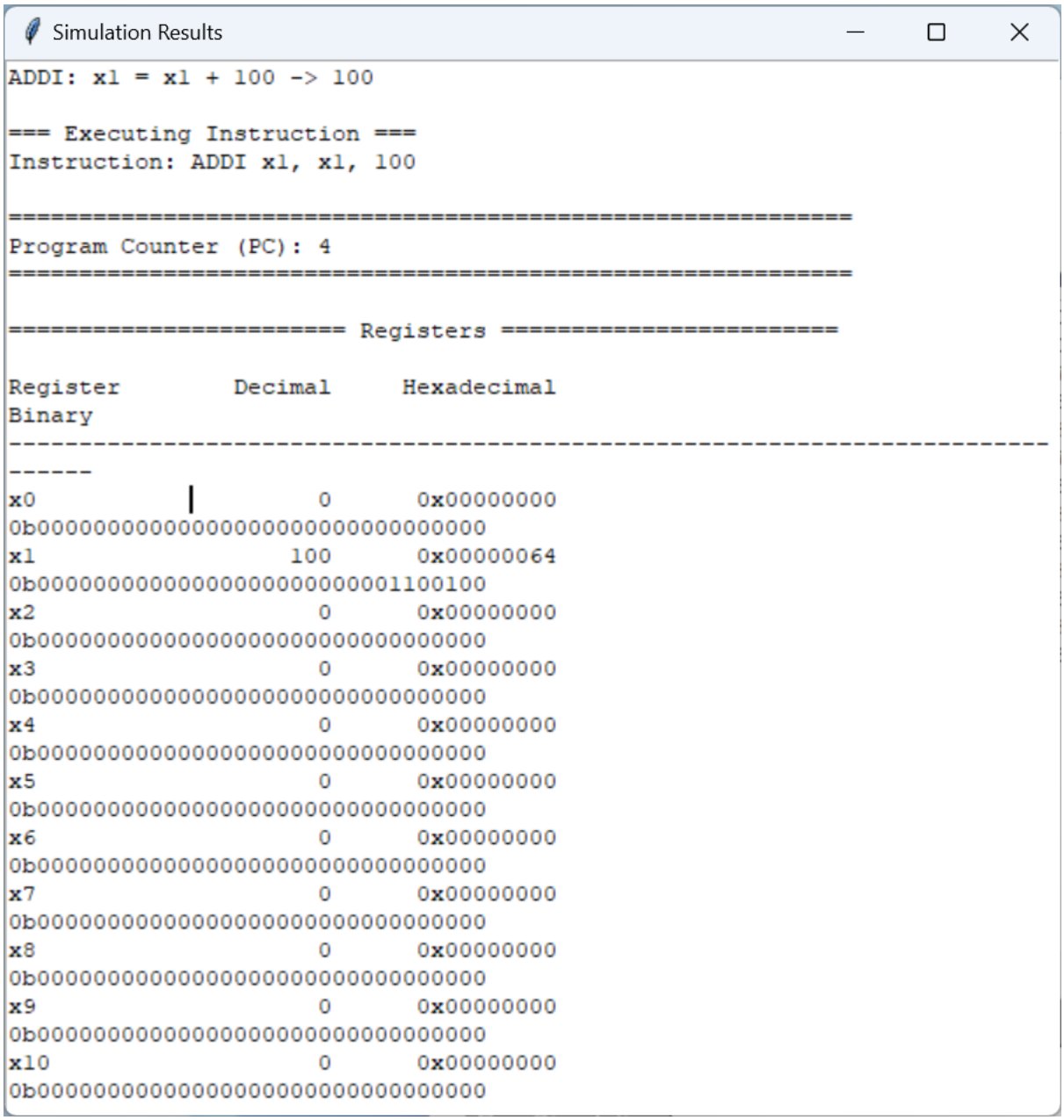


Figure 6: Simulation Result of `memory__test_case.txt` with the starting PC set to 4 and memory = 4000 MB.