FUNCTIONAL SIMULATOR FOR SUBSET OF RISC-V INSTRUCTION SET

Group Members

| Atharva Suhas Mulay | 2021CSB1076 |
| --- | --- |
| Karanraj Mehta | 2021CSB1100 |
| Nishad Dhuri | 2021CSB1116 |
| Sumit Patil | 2021CSB1135 |

Summary

The project has two components: a command-line interface (CLI) and a web-based graphical user interface (GUI) built with Flask and Tailwind. The CLI reads a machine code file and simulates the execution of the instructions using a single-cycle processor architecture. The GUI provides a visual representation of the processor's state at each step of execution and allows users to input machine code directly. The project includes a set of sample assembly files for testing. The code is organised into separate modules for instruction fetch, decoding, execution, memory access, and write-back. The documentation includes a description of the project structure and modules and instructions for using the CLI and GUI.

The simulator is designed for a subset of the RISC-V instruction set. The simulator reads encoded instructions from a memory file (machine code), decodes, executes, and writes back the results to a register file. The supported instruction set is RISCV-32I, and the simulation stops when the instruction is '\_'. The simulator also prints messages for each stage of instruction execution. The data structure includes registers, memories, and intermediate output for each stage declared as objects in all the python modules.

The simulator flow involves two steps:

Loading the memory with the input memory file and executing the instructions one by one in an infinite loop until the '\_' instruction sequence is reached.

Finally, the simulator is tested with three assembly programs:

Fibonacci, Sum of an array of n elements and Bubble sort

Input / Output

Input

The CLI and GUI versions of the single-cycle RISC-V processor require machine code as input. The machine code instructions must be written in the same format as the project description to provide input.

Each line should have the following:

<address of instruction><delimiter - space><machine code of the instruction>

For the CLI, the machine code instructions must be written in the input.mc file provided in the single\_cycle folder. Run the CLI using the command mentioned in the README file in the same folder.

For the GUI, the machine code instructions must be entered into the text area provided on the web page and saved using the save button.

It is important to note that the instruction '\_' is used to exit the simulator and is required as the last instruction to terminate the simulator.

For example,

0x0 0x00200093

0x4 0x00300113

0x8 0x002081B3

0xC \_

Functional Behavior and Output

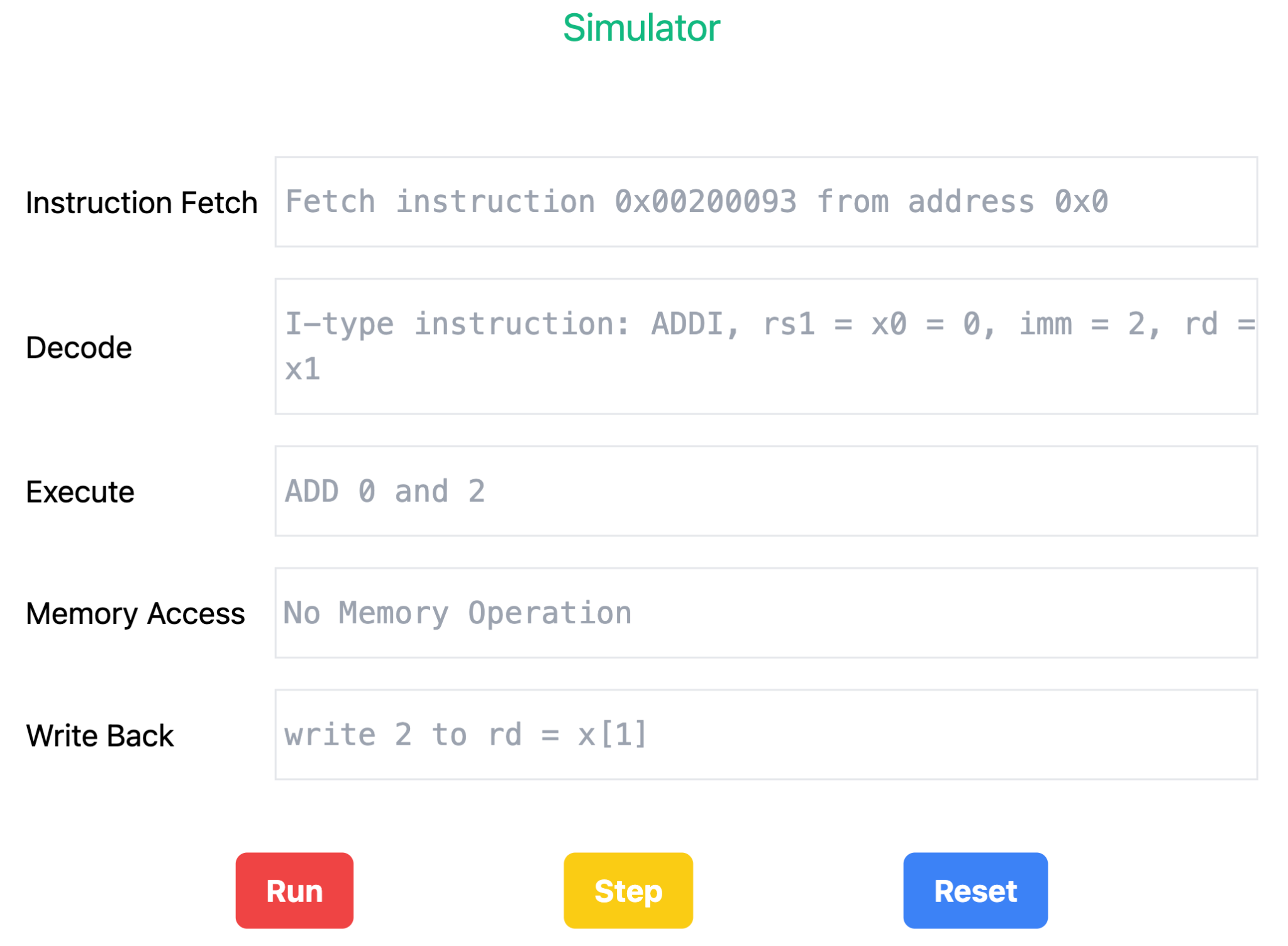
The functional behaviour of the simulator is the same for both the CLI and GUI versions. The simulator can read machine code from the instruction memory, decode the instruction, read from the register, execute the operation, and write back to the register file. The simulator supports all instructions in RISCV-32I.

In the CLI version, three options are available: run, step, and reset. The run option allows the simulator to execute all instructions until the program is complete or the exit instruction is reached. The step option allows the user to execute one instruction at a time, and the reset option resets the simulator to its initial state.

The simulator generates output messages for each stage of execution in the output.txt file.

Once the execution of the program is complete, the simulator prints the state of the register file and memory.

The same behaviour applies to the GUI version of the simulator, with the output displayed on the web page.



Design of the Simulator

Data Structures

The simulator uses several data structures to maintain the system's state. The registers are stored in the file registers.py as a list of 32 integers. Each integer in the list corresponds to a register and can hold a 32-bit value.

The instruction memory is stored in the file instruction\_fetch.py as a dictionary where the key is the address of the instruction in hexadecimal format, and the value is the instruction itself in hexadecimal format. Similarly, the data memory is stored in the file memory\_access.py as a dictionary where the key is the address of the data in hexadecimal format, and the value is the data itself in binary format.

The control signals are declared as regular Python variables in the file decode.py. These signals control the flow of instructions and data in the simulator.

Using these data structures, the simulator can maintain the system's state and execute instructions accurately.

Simulator Flow

The simulator flow follows a single-cycle processor design. During each clock cycle, the following steps are executed:

Fetch

The instruction pointed by the Program Counter (PC) is fetched from the Instruction Memory.

Decode

The fetched instruction is decoded to determine the operation to be performed, the operands involved, and the destination register.

Execute

The ALU (Arithmetic Logic Unit) operates on the operands.

Memory Access

If the instruction involves a memory operation (e.g. load/store), the memory is accessed to read/write data.

Write Back

The result of the execution is written back to the destination register.

These five stages are executed for each instruction, one after the other. In other words, each instruction takes one clock cycle to complete.

At the end of the execution, the simulator prints the final state of the Register File and the Data Memory. This provides the user with the output of the program executed.

Implementation Details of RISCV-32I Simulator

Instruction Fetch

The fetch stage is the first stage in the instruction execution process. Its main task is to fetch the instruction from the instruction memory using the program counter (PC). In the implementation, the instruction memory is stored as a dictionary where the key is the address of the instruction, and the value is the instruction itself in hexadecimal format. The program counter is initialised to 0x0 and is updated in increments of 4 bytes after each instruction fetch.

The fetch stage is implemented as a function named fetch(). It returns True if the program has not reached the end of the instruction memory and False if it has. This function is called repeatedly during the instruction execution process until the end of the instruction memory is reached.

In addition to the fetch function, an init() function initialises the program counter to its initial value, 0x0. This function is called at the beginning of the instruction execution process to ensure that the program counter is reset to its initial value.

Decode

In the decode stage, the instruction fetched from the instruction memory is decoded to determine the control signals, the registers' values, and immediate values used by the instruction.

The decode function extracts the opcode, registers, and immediate values from the instruction using bitwise operations and string slicing. Based on the opcode, the control signals such as OP2Select, ALUOperation, MemOp, ResultSelect, and RFWrite are set to control the execution of the instruction.

The opcode determines the instruction type and the registers used in the instruction. The values of the registers are obtained from the register file maintained as a list of integers in registers.py. The immediate values are extracted from the instruction and sign extended.

The decoded values are then passed to the execute stage to perform the actual execution of the instruction.

The init function is used to reinitialise all the global variables to their initial values. This is useful for resetting the simulator.

Execute

The execute stage performs the execution of the instruction based on the control signals generated by the decode stage. The execute stage is implemented as a function named execute(), which uses the control signals and decoded values to calculate the ALUResult.

In the case of branch instructions, pc is updated if the branch is taken.

For instructions that require memory access, execute stage is ignored.

The init() function is used to reinitialise all the global variables to their initial values at the beginning of the instruction execution process to ensure that the simulator is reset to its initial state.

Memory Access

The memory access stage is responsible for accessing the memory and performing any memory-related operations required by the instruction being executed.

The module starts by checking the MemOp signal from the previous stage. If it is zero, no memory operation is performed, and the function returns. Otherwise, the function checks the func3 field to determine the type of memory operation required.

Based on the type of memory operation, 32 bits, 16 bits or 8 bits from immediate are loaded or stored.

Write Back

The simulator checks the RFWrite control signal's value to determine whether to write the result into the register file. If RFWrite is set to 0, the result is not written into the register file. However, if RFWrite is set to 1, the simulator writes the result into the register file.

During this stage, the simulator selects one of the following values based on the value of the control signal ResultSelect: aluResult (the result of the ALU operation), loadData (data that is read from memory), immU (in case of U-type instructions), pc + 4 (in case of jump and branch instructions), or immU + pc (in case of jump instructions like AUIPC). The selected value is then written into the destination register specified in the instruction.

No write operation is performed if the destination register is x0, as x0 is a read-only register. Once the write-back stage is completed, the clock cycle is incremented by 1 to prepare for the next instruction execution.