Design Document: RISCV Simulator

The document describes the design aspect of the **RISC-V Simulator**, which simulates the machine-level execution of RISC-V 32-bit instructions using C++ and python as our programming languages.

# INPUT/OUTPUT

## Input

Input to the simulator is **test\_case.mem** file that contains the encoded instruction and the corresponding address at which the instruction is supposed to be stored, separated by space as a separator. For example:

## 0x0 0x00000093

## 0x4 0x00100113

## 0x8 0x00900193

## 0xC 0x00200213

## Functional Behavior and Output

The simulator reads the instruction from instruction memory, decodes the instruction, reads the register, executes the operation, and writes back to the register file. The instruction set supported is the same as given in the lecture notes of CS204 and as per the lab assignments.

The execution of instruction continues till it reaches instruction “**swi 0x11**”. In other words, as soon as the instruction reads “**0xEF000011**”, the simulator stops and writes the updated memory contents onto a memory text file.

There are various output files such as **output\_log.mem, register.mem, D\_Memory.mem.** Example of output of one instruction in **output\_log.mem** is as follows:

*Fetch instruction : 0x102b7 from address 0x0*

*Decode :*

*Formate of instruction : U*

*immediate : 10, RD : 5*

*Operation is lui*

*Execute :*

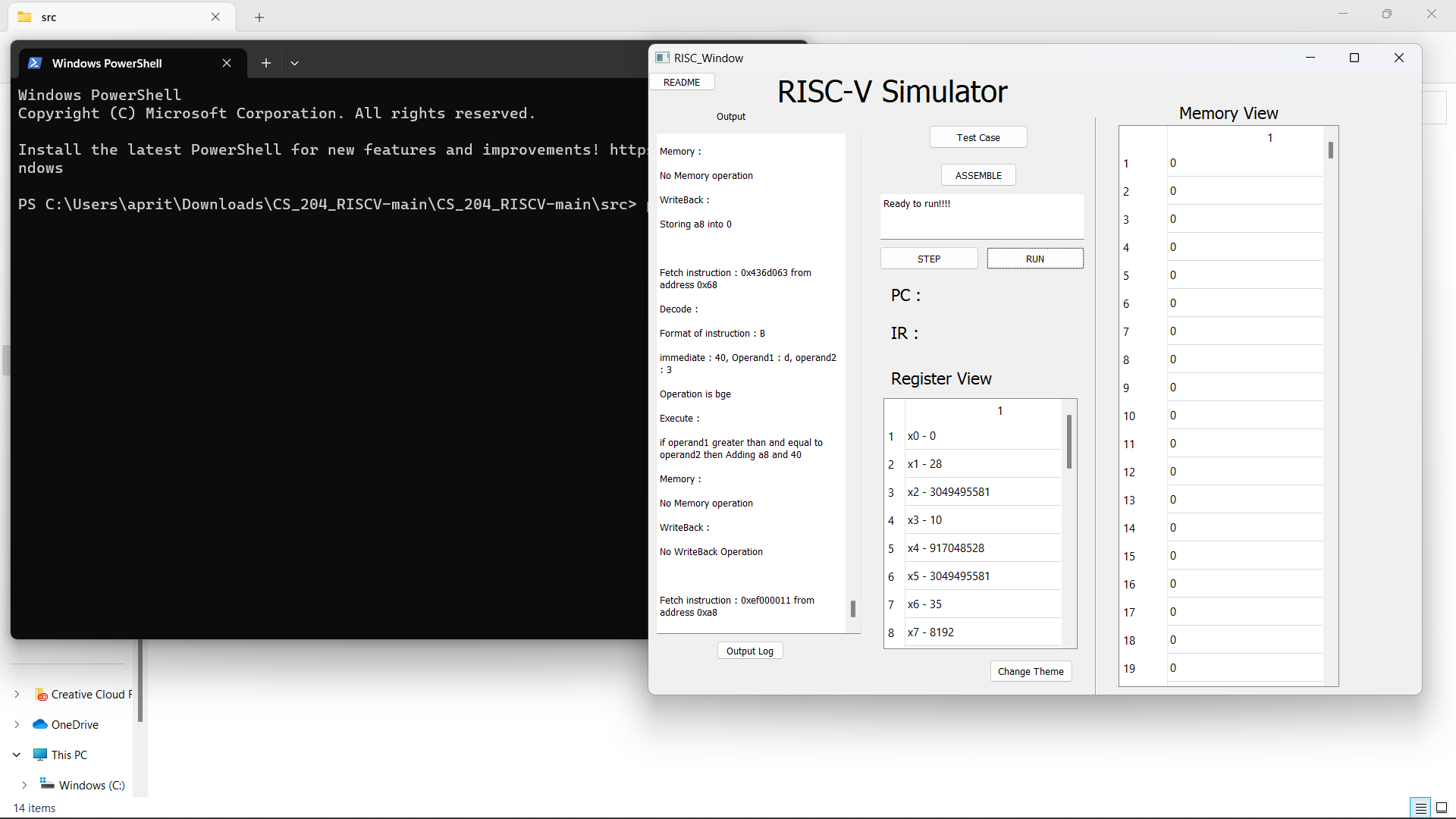
*Memory :*

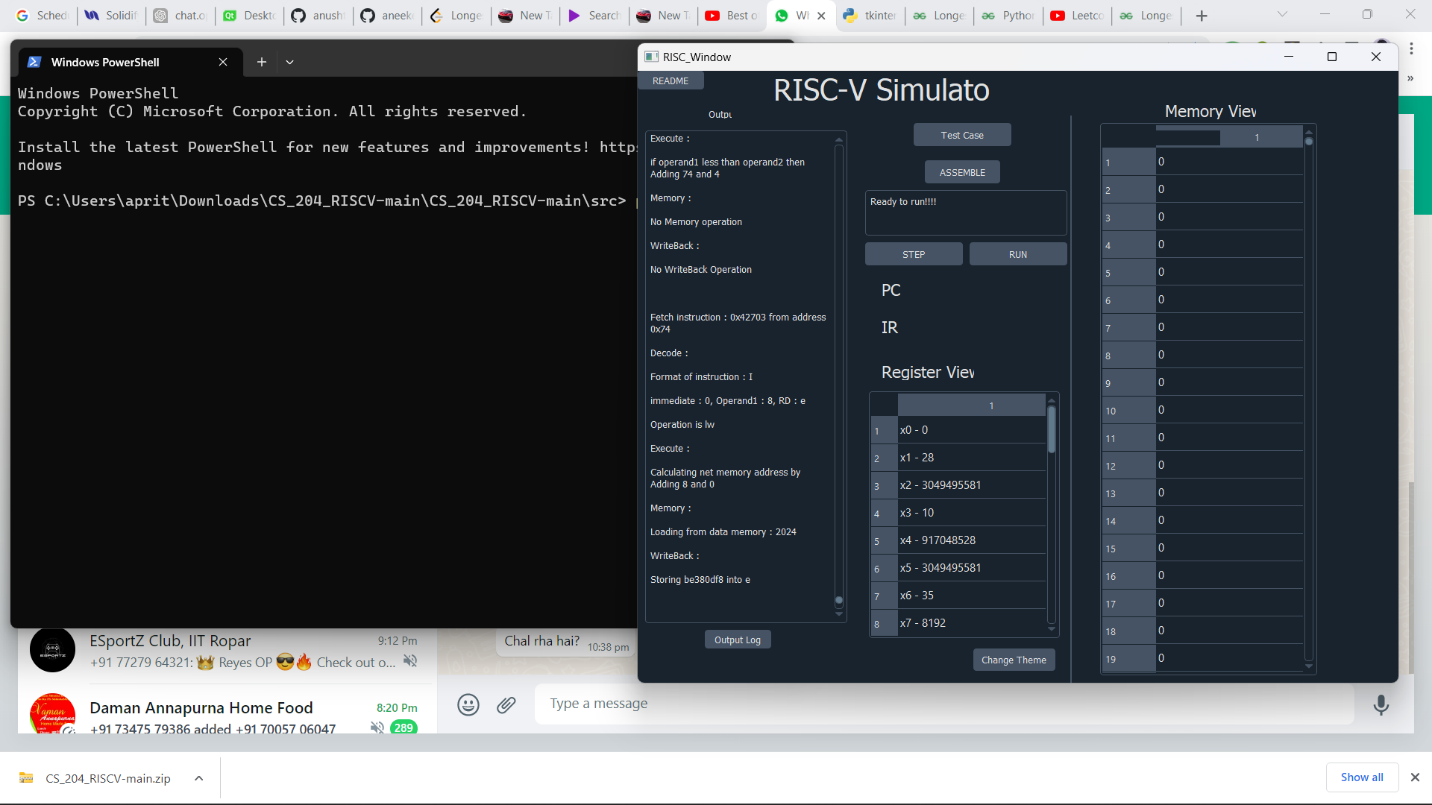
*No Memory operation*

*Write Back:*

*Storing 10000 into 5*

**GUI**





# DESIGN OF SIMULATOR

## Data structure

Registers, memories, intermediate, PC, and, instruction register, used for each stage of instruction execution are declared as global variables.

For the implementation of registers and memories, two separate arrays are used, while for storing the instructions **MEM array** and for storing data in memory **DMEM array** is used. Our memory is Byte Addressable.

## Simulator flow

There are two steps:

1. First memory is loaded with an input memory array and each instruction of the input file is stored in **MEM array** and memory data in **DMEM array.**
2. Simulator executes instructions one by one.

For the second step, there is an infinite loop, which simulates all the instructions till the instruction sequence reads {**EXIT**} “0xffffc”.

Next, we describe the implementation of fetch, decode, execute, memory, and write-back function.

1. **FETCH**: Instruction is fetched from the instruction array at the current PC address.
2. **DECODE**: Each instruction is decoded according to the opcode it has. Further according to each of the R, I, S, B, J, and U type instructions the rs1, rs2, immediate value, the type of operation, and other necessary details are stored. All the required details like func3, func7, rs1, rs2, imm, rd etc. are decoded in this function.
3. **EXECUTE**: The ALU executes the instruction by computing the desired output using values stored in registers in rs1, rs2, rd, imm, etc. The required type of ALU instruction is decided and accordingly the task is performed.
4. **MEMORY ACCESS**: Memory is read and written in this stage. This stage is used only for load and store instructions. For the remaining instructions, this stage is redundant.
5. **REGISTER WRITE-BACK**: Here we are storing the final values in the corresponding registers. The register update is extremely fast compared to memory read and update.

# TEST PLAN

We test the simulator with the following assembly programs:

* Fibonacci Program
* Sum of the array of N elements.
* Bubble Sort Program