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 :*

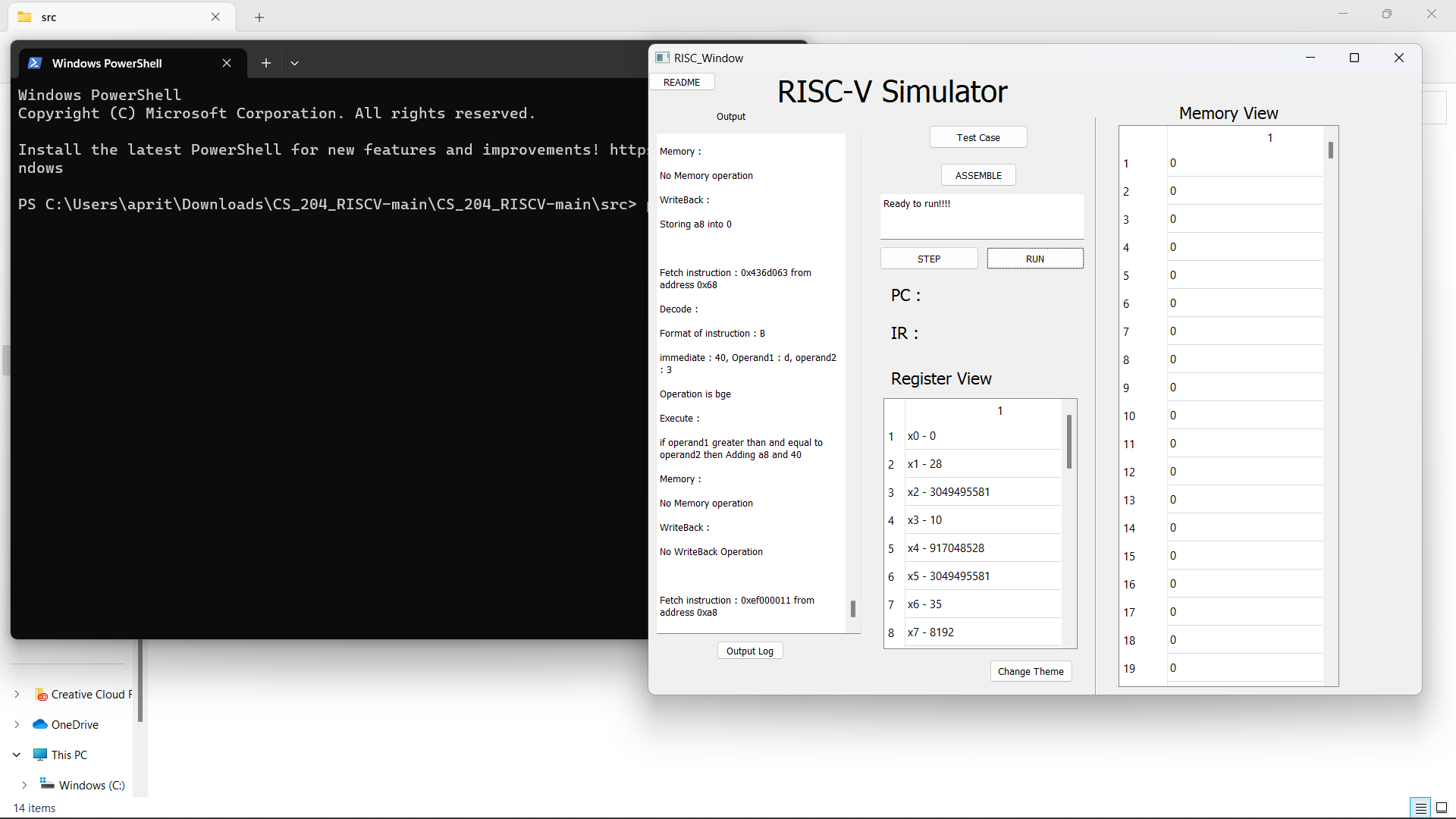
*Memory :*

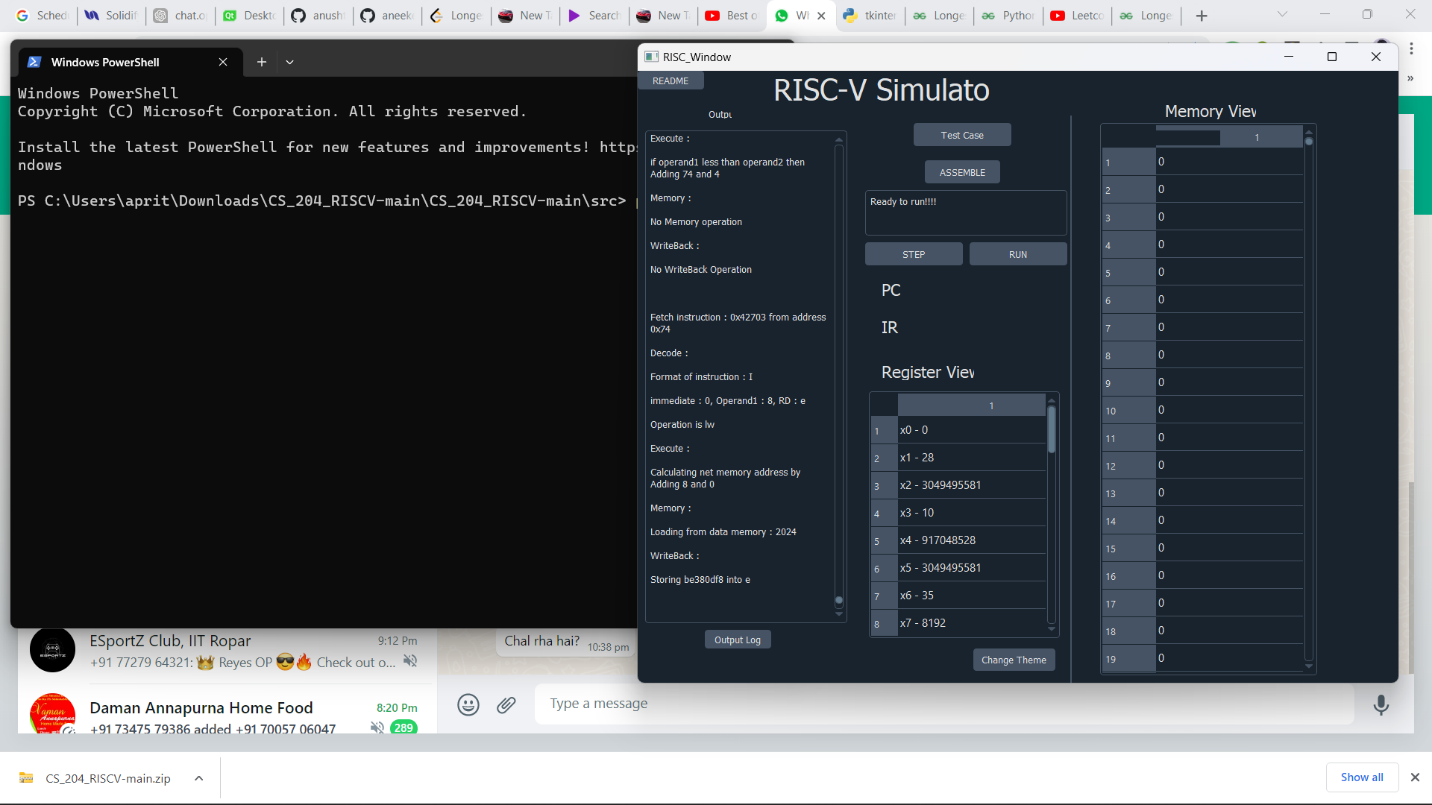
*No Memory operation*

*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, immed, 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, immed, 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.

Simple Single Cycle implementation is done in the file **SingleCycle.cpp**.

**PIPELINING**  
In this, a cycle can have all five functions working together. We **introduced pipeline registers**, namely IF\_DE, DE\_EX, EX\_MA, MA\_WB and a buffer register WBB (not used in the actual structure but we needed it for the Simulator) using struct. We are storing various values after a particular function is executed so that the resultant data is not lost in the same cycle due to the use of the same global variables in other functions.

The functions are executed in main function in the order Write Back, Data Memory, Execute, Decode and Instruction Fetch. This is because if we use Fetch of an instruction before Decode of its previous instruction, the register values in IF\_DE of the former will be overwritten over the latter and we will not be able to use Decode for the lost instruction. We have placed proper if conditions for the cases in which a cycle will not have particular functions (for example, the second cycle will not run Execute, Data Memory and Write Back functions).

But beware, this file does not have way of dealing with hazards! So, other files have been made for tackling this problem.

## Stalls

The best way to deal with hazards is to add stalls in the code. As explained in theory, stalls prevent data hazards like RAW and control hazards. There are other types of hazards (WAW and WAR in data hazards and structural hazards), but they occur on advanced and latest computers, so we would not be dealing with them. Pipelining (stalls included) is implemented in the file named **StalledPipeline.cpp**.

Stalls for data hazards are found out in the decode part of the function, and for control hazards, stalls are found in the execute part. So, using data in the pipeline registers (structs) and if else statements, we were able to specify the type and number of stalls to be implemented.

## Forwarding

Stalls are a good way to keep the correctness of the machine code. But it slows down the process of compiling due to stalls increasing the number of cycles and hence the CPI. So, forwarding is implemented to tackle the problem of increases in cycles due to data hazards. Again, the pipeline registers values have been used, as they should be, to direct values in the previous functions. If conditions in the stalls have been removed and the conditions have only been marked to keep note of the type of stall and forwarding done in the program.

## Branch Prediction

We take care of the data stalls using forwarding. But what about the control hazards? We use branch prediction to speed up the same. A global Boolean array Branch\_prediction has been taken to predict the loop values, and thus minimize stalls. 1-bit branch prediction has been implemented, so it has an accuracy of (n-2)/n \* 100%.

## Overall

We have combined the best-pipelined RISCV-Simulator in the file **myRISCVSim.cpp**. GUI has been updated as per the needs and requirements of the problem statement and all of the above can be accessed using the same GUI.

# TEST PLAN

We test the simulator with the following assembly programs:

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

**IMPLEMENTATION OF CACHE**

In this phase, we are implementing the cache module in our pipeline. Instead of fetching instructions directly from the memory, to make our pipeline more efficient we are accessing the cache first instead of memory. Using the principle of memory hierarchy, we are implementing two cache modules – **Instruction Cache** and **Data Cache.**

## Instruction Cache

An instruction cache is a type of cache memory used in computer processors to improve performance by storing frequently accessed instructions from the main memory. When a program is executed, the processor fetches instructions from the memory and stores them in the instruction cache. If the processor needs the same instruction again, it can quickly access it from the cache instead of going back to the main memory, which can be much slower. The cache uses a tag to identify which set a particular instruction belongs to, and it uses an index to select the appropriate cache line within that set. The size of the instruction cache can vary depending on the specific processor architecture and design.

## Data Cache

A data cache is a type of cache memory used in computer processors to improve performance by storing frequently accessed data from the main memory. When a program reads or writes data, the processor fetches the data from the main memory and stores it in the data cache. If the processor needs the same data again, it can quickly access it from the cache instead of going back to the main memory, which can be much slower. The cache uses a tag to identify which set a particular data belongs to, and it uses an index to select the appropriate cache line within that set.

The overall implementation of our code includes the user to input the cache size, block size and the mapping which is required to be run. Further, for set-associative, no. of sets desired are also taken as input. The number of accesses, number of hits, number of misses, number of cold, conflict and capacity misses and the total number of memory stalls etc. are given as output.

Furthermore, instruction cache and data cache are also printed.