This project involves the creation of an assembler and simulator consisting of 6 classes of instructions. The interconnectivity of these classes forms a simulator, allowing the execution of a variety of operations. The supported instructions include:

AND: Performs a bitwise AND operation on two registers and stores the result in the destination register (rd).
OR: Performs a bitwise OR operation on two registers and stores the result in the destination register (rd).
ADD: Adds the values of two registers and stores the result in the destination register (rd).
SUB: Subtracts the value in register rs2 from the value in register rs1 and stores the result in the destination register (rd).
ADDI: Adds an immediate value to the content of register rs1 and stores the result in the destination register (rd).
BEQ: Branches to a specified location if the values in registers rs1 and rs2 are equal.
LW (Load Word): Loads a value from memory and stores it in the destination register (rd).
SW (Store Word): Stores the value of register rs2 into memory.
SLL (Logical Left Shift): Performs a logical left shift operation on the value in register rs1 by 5 lower bits of rs2.
SRA (Arithmetic Right Shift): Performs an arithmetic right shift operation on the value in register rs1 by 5 lower bits of rs2

.

To use the simulator, simply run the CPU class in any Python compiler. The simulator we implemented seems to follow a basic five-stage pipeline commonly found in modern processors. The pipeline stages are Fetch (F), Decode (D), Execute (X), Memory (M), and Writeback (W). Let's go through each stage:

Fetch (F):

- In this stage, the simulator fetches the instruction from the instruction memory.
- The **fetch** function takes the current instruction from the instruction memory, removes any spaces, and returns the instruction to be processed.

Decode (D):

- The **decode** function extracts the opcode from the instruction and returns it for further processing.
- It checks the opcode to determine the instruction type (R, I, S, SB, U, UJ, or NOC).
- Depending on the instruction type, it proceeds to the corresponding execution stage with the decoded instruction.

Execute (X):

- The **execute** function handles the execution of instructions based on their types (R, I, S, SB, U, UJ, or NOC).
- For R-type and I-type instructions, it performs the necessary computation and returns the result.
- For S-type instructions, it calculates the memory address where the data needs to be stored.
- For SB-type instructions (branches), it determines whether the branch should be taken and adjusts the program counter accordingly.
- For U-type and UJ-type instructions, it calculates the immediate value.

Memory (M):

- The **memory_stage** function is responsible for handling memory-related operations.
- For S-type instructions, it writes data to the memory.
- For NOC-type instructions, it writes data to the memory-mapped register (MMR).

Writeback (W):

- The **writeback** function updates the register file or memory-mapped register based on the result of the execution.
- It also sets the corresponding register as not in use, allowing other instructions to use it.

In addition to the main pipeline stages, the simulator includes latches (**IF/ID, ID/EX, EX/MEM, MEM/WB**) to store intermediate results between stages. This helps handle data dependencies and control hazards.

The pipeline is managed through a series of conditional statements, checking for data hazards and handling stalls appropriately. Stalls are tracked in the **graph_data** dictionary, which is later used to generate graphs showing stalls versus cycles.

The simulator provides functions to print the current state of the registers, pipeline, and memory.

Graph plotter

It includes functions to generate graphs for instruction types, stalls versus cycles, and memory accesses.

Instruction Types Graph:

- Bar graph showing the frequency of different instruction types executed.
- X-axis: Instruction types, Y-axis: Frequency.

Stalls versus Cycles Graph:

- Line graph indicating the occurrence of stalls during different cycles.
- X-axis: Cycles, Y-axis: Number of stalls.

Memory Accesses Graph:

- Bar graph displaying the frequency of memory reads and writes.
- X-axis: Memory access types, Y-axis: Frequency.

Each graph provides a visual overview of the simulator's performance, helping to analyze and optimize its behavior.