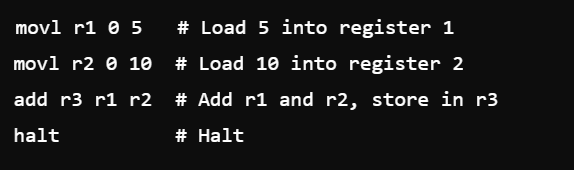
**Verifying Correct Execution of Test Programs in the LC Pipelined Simulator**

To ensure the correctness of the LC pipelined simulator, a variety of test programs were devised. These programs are designed to stress-test different aspects of the pipeline stages, including instruction fetch (IF), instruction decode (ID), execute (EX), memory (MEM), write-back (WB), data hazards, and control hazards (e.g., branches and jumps). By observing how the simulator handles these scenarios, we can verify its ability to correctly execute any legal program. Below is a breakdown of how specific test programs help verify the correct execution of the simulator.

**Basic Arithmetic Operations Test Program**

This test program focuses on basic arithmetic operations (addition, subtraction, NAND) and ensures that the simulator correctly executes ALU instructions through all pipeline stages. Here’s an example of a simple addition test program:

Program 1: Simple Addition and HALT



In this program:

- Two `movl` instructions load immediate values into registers (`r1` and `r2`).

- The `add` instruction performs an arithmetic operation and stores the result in `r3`.

- Finally, the `halt` instruction stops execution.

**Verification:**

- Register File Verification: After running this program, the final value of register `r3` should be the sum of `r1` and `r2`, i.e., `5 + 10 = 15`. This tests whether the pipeline handles the instruction correctly and ensures the register file is updated with the correct result.

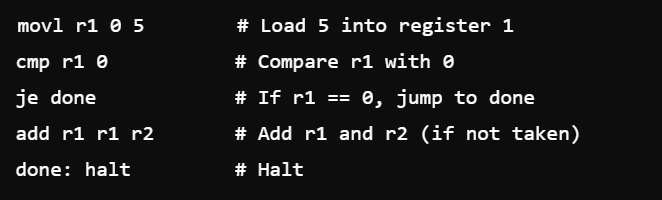
- Pipeline Stage Observation: During execution, each instruction will pass through the pipeline stages (IF, ID, EX, MEM, WB). By examining the values of registers at each stage, we can ensure the pipeline works as intended.

- No Data Hazards: This program has no dependencies between instructions, so we can also verify that no unnecessary stalls or hazards are introduced during execution.

**2. Branch and Comparison Test Program**

Branch instructions are critical in verifying the simulator’s handling of control hazards. This test focuses on comparison (`cmp`) and conditional jump (`je`) instructions. Correct branch prediction and updating of the program counter (PC) are necessary for successful execution of this test.

Program 2: Branching Example



In this program:

- `movl` loads the value `5` into `r1`.

- `cmp` compares the value in `r1` with `0`.

- `je` checks if the comparison is true (if `r1 == 0`), and if so, jumps to the `done` label. Otherwise, it continues executing the next instruction.

- If the branch is not taken, the program adds `r1` and `r2` and continues execution.

Verification:

- Branch Handling: The value in `r1` is `5`, so the `cmp` instruction will not set the comparison flag, and the branch will not be taken. We expect the `add` instruction to execute before the program halts.

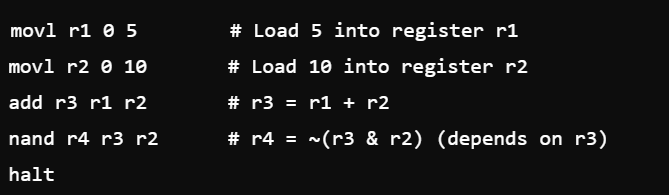
- PC Update: The `je` instruction should not change the PC value since the condition is false. This verifies that the simulator correctly handles branch instructions when the condition is not met.

- Control Hazard Handling: The simulator uses branch prediction logic. If a branch is mispredicted, it should recover by flushing incorrect instructions in the pipeline and ensuring correct execution after the branch is resolved.

**3. Data Hazard Test Program**

Data hazards arise when instructions that are close together in a program depend on one another for data. The simulator needs to properly handle such hazards using data forwarding or stalling mechanisms. The following test program induces a data hazard.

Program 3: Data Hazard with Forwarding



In this program:

- The first two `movl` instructions load immediate values into `r1` and `r2`.

- The `add` instruction computes the sum of `r1` and `r2` and stores it in `r3`.

- The `nand` instruction operates on `r3` (the result of the previous `add`), creating a data hazard.

Verification:

- Forwarding Mechanism: The value of `r3` (from the `add` instruction) must be forwarded to the `nand` instruction. The simulator should detect the hazard and forward the result of the `add` instruction from the EX stage to the next instruction in the ID or EX stage.

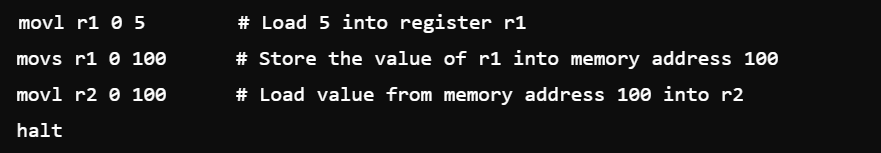
- Correct Result: After execution, the correct value should be present in `r4`, which confirms that data forwarding occurred successfully.

- No Stalls: Since data forwarding is implemented, the program should execute without stalls. If stalls occur, this may indicate an issue with hazard detection or forwarding.

**4. Memory Access Test Program**

Memory operations (`movl` and `movs`) access data memory and verify the simulator’s handling of load and store instructions. Correct memory read/write operations are critical for ensuring that the simulator interfaces correctly with the data memory.

Program 4: Memory Access Test



In this program:

- `movl` loads a value into `r1`.

- `movs` stores the value of `r1` into memory at address 100.

- Another `movl` instruction loads the value from memory address 100 back into `r2`.

Verification:

- Memory Write and Read: After the `movs` instruction, memory address 100 should contain the value `5`. The subsequent `movl` should load this value into `r2`. Verifying that `r2` holds the correct value after execution ensures that memory operations are functioning correctly.

- Pipeline Timing: Load and store instructions introduce unique pipeline timing issues. By ensuring that the value is correctly read from memory in subsequent cycles, we verify that the simulator handles memory access properly.

**Summary**

The test programs described above collectively verify that the LC pipelined simulator handles a wide range of instructions and scenarios, including:

1. Basic arithmetic and logical operations.

2. Control hazards caused by branching instructions.

3. Data hazards requiring forwarding or stalling.

4. Memory access operations.

By carefully observing the contents of the register file, memory, and pipeline stages during and after the execution of these programs, we can validate that the simulator behaves correctly under all legal conditions. Furthermore, since these test cases represent common patterns found in typical LC programs, they provide confidence that the simulator will handle more complex programs and edge cases without error.