Adding "1 Ahead" Forwarding
The "1 ahead" forwarding mechanism was made to reduce pipeline stalls by allowing the immediate forwarding of data between adjacent instructions. To implement this, we modified the forwardingunit.v and CPU.v. Specifically, the forwardingunit.v was updated to detect scenarios where an instruction depends on the output of the preceding instruction. When this condition is met, the forwarding unit reroutes the needed data directly from the pipeline register or output stage of the preceding instruction, bypassing the normal register write-back phase. This allows the dependent instruction to execute without waiting for the register write. This leads to reducing stalls. We confirmed functionality by testing with forwardingunit_tb.v, which showed that data dependencies between back-to-back instructions were resolved without introducing delays.

Adding "2 Ahead" Forwarding
The "2 ahead" forwarding extension allows the CPU to handle data dependencies that arise two instructions apart. This enhancement required further logic within forwardingunit.v and CPU.v to find cases where data needed by the current instruction is available from an instruction that was executed two cycles earlier. By forwarding data from the pipeline stage of this second-previous instruction, we prevent unnecessary stalls in situations where this data would otherwise be delayed. forwardingunit_tb.v testing shows that with "2 ahead" forwarding, the CPU successfully manages these data dependencies, which leads to correct instruction flow without interruption.

Arbitration Logic for 1 & 2 Ahead
To streamline decision-making between "1 ahead" and "2 ahead" forwarding, arbitration logic was introduced within forwardingunit.v. This logic prioritizes the closest available forward, ensuring that data is fetched from the most recent stage with valid information. The arbitration logic checks both the "1 ahead" and "2 ahead" conditions and chooses the nearest source. This minimizes latency by reducing the data retrieval path. The testbench we made confirmed that the CPU correctly selected the closest forward path in all scenarios so that the pipeline's efficiency is maintained.

$0 Write Prevention
MIPS architecture calls for the $0 register to always hold the value zero. To enforce this, modifications were made in RegFile.v to prevent any write operations to $0. This was accomplished by adding a condition that blocks write operations targeting the $0 register, regardless of the instruction. The logic ensures that any attempt to write to $0 is effectively ignored, keeping it constant at zero throughout all operations. Testing verified that, even when instructions inadvertently targeted $0, its value remained unchanged.

No Write Logic
As we know, some instructions do not require a write operation to any register. In order to incorporate this, we added No Write logic within CPU.v and RegFile.v to detect such instructions and bypass the write stage entirely. By skipping the write operation, the CPU conserves resources and lowers register activity, which enhances overall processing efficiency. This functionality was validated through testing, which confirmed that non-write instructions passed through the pipeline without triggering a write, demonstrating correct implementation.

Register Bypass Verification

For the register bypass verification, we conducted tests using testbenches on different register interactions and data dependencies. The tests showed that register bypass mechanisms worked correctly across all scenarios. This resolves data dependencies accurately and prevents pipeline hazards without stalling. This ensured that the implemented forwarding and arbitration logic operated as expected even with the complex dependency chains.