# RISC-V Simulator Project Report

Zeyuan He (123090168), Yihan Wang (123090588)

## 1. Introduction

In this project, we modified the simulator from RISC-V64I to RISC-V32I, and add fused instructions into it. We also add an argument to disable data forwarding, which needs to stall corresponding fetch and decode stages to handle data hazard. Finally we optimize a given RISC-V instruction sequence by reordering the instructions and using fused instructions.

## 2. Converting from RISC-V64I to RISC-V32I

The first task involved modifying the existing RISC-V64I simulator to operate as a RISC-V32I simulator. This required careful analysis of the codebase to identify and update all 64-bit specific operations and data structures. We achieved this by changing register width from 64-bit to 32-bit, adjusting ALU operations to handle 32-bit values and modifying memory access patterns to accommodate 32-bit addressing.

## 3. Implementation of Fused Instructions

The implementation of fused integer instructions followed the R4 instruction format specified in the project requirements. The implementation process included:

1. Adding instruction decoding logic in the decode stage to recognize the custom opcode (0x0B).
2. Extracting register operands according to the R4 format in decode stage.
3. Implementing the execution logic for each fused operation in execute stage.
4. Ensuring proper data forwarding for these instructions.

In this part, we gained experience with custom opcode handling and multi-operation instructions. The most valuable insight was understanding how fused operations can reduce both execution cycles and data hazards by combining dependent operations into a single instruction.

## 4. Disable Data Forwarding

To implement the -x option for disabling data forwarding, we modified the pipeline execution logic to handle data hazards. If data forwarding is disabled and data

dependencies are detected, we insert appropriate stalls and bubbles to ensure correct execution.

In the **execution stage**, we detect data hazards where the instruction in the decode stage requires a register that will be modified by the current instruction. When detected, we stall the fetch stage for 2 cycles and the decode stage for 3 cycles. This provides sufficient time for the result to propagate through the pipeline and be written back to the register file.

For the **memory stage**, we implemented similar detection logic but with shorter stall durations (1 cycle for fetch and 2 cycles for decode) since the result is closer to being available. we included checks for whether we have stalled in execution stage and whether jump has occurred to prevent redundant stalls. The jump condition is particularly important because when a jump is occurring, the pipeline is already being flushed and refilled with instructions from the target address, making data hazard stalls unnecessary for those instructions that won't execute.

In the **writeback stage**, we implemented the final hazard detection with a minimal stall of just 1 cycle for the decode stage, as the result will be available in the register file after the current cycle completes. Similar to the memory stage, our implementation includes careful checks for previous stalls and jump conditions to avoid unnecessary pipeline disruptions.

This comprehensive approach to hazard detection without forwarding taught us the importance of considering control flow (jumps and branches) when managing data hazards. We learned that jumps naturally resolve certain hazards by changing the instruction stream, making additional stalls redundant.

**5. Instruction Reordering**

In my analysis of the original code sequence, I identified several data dependencies that would cause pipeline stalls when data forwarding is disabled. My reordering strategy focused on separating dependent instructions while maintaining program correctness. My reordering approach follows several key principles:

1. **Separation of dependent instructions**: I increased the distance between instructions that have data dependencies, such as moving the add a3, a3, a4

further away from its dependent mul a4, a1, a2.

2. **Interleaving independent operations**: I interleaved independent instructions between dependent ones to fill potential stall cycles, such as placing addi x8, x6, 0 early in the sequence.
3. **Balancing register usage**: I carefully tracked register dependencies and ensured that instructions using the same registers were properly separated.
4. **Preserving memory access patterns**: I maintained the order of loads and stores to the same memory location to preserve correctness.

This reordering significantly reduced pipeline stalls. The optimization maintains the program's original functionality while improving pipeline efficiency, demonstrating the effectiveness of instruction scheduling as a compiler optimization technique.

**6. Optimization Using Fused Instructions.**

After successfully reordering the instructions to minimize pipeline stalls, I further optimized the code by introducing fused instructions to replace some sequences of operations.

The fused multiply-add instructions perform both multiplication and addition in a single instruction. This optimization provides four significant benefits:

1. **Reduced instruction count**: Each fused instruction replaces two regular instructions, reducing the total instruction fetch and decode operations.
2. **Elimination of intermediate results**: The multiplication result doesn't need to be stored in a temporary register (a4 in the first case), which eliminates the associated register write and subsequent read operations.
3. **Improved cycle efficiency**: While the fused instructions take more cycles than a single regular instruction (additional 3 cycles), they still execute faster than the two separate instructions they replace (which would take at least 5 cycles combined when accounting for pipeline stages).
4. **Reduced data hazards**: By combining dependent operations, the fused instructions inherently eliminate the data hazard that would exist between the separate operations.

This optimization demonstrates how specialized instructions can be leveraged to

improve both code density and execution efficiency. When combined with intelligent instruction reordering, these fused operations provide a substantial improvement in pipeline utilization and overall performance. The resulting code maintains the same functionality while executing with fewer cycles and reduced stall time, showcasing the effectiveness of both hardware architectural features and software optimization techniques.

## 7. Conclusion

This project provided invaluable hands-on experience with fundamental computer architecture concepts. The most significant lesson was understanding the interplay between hardware mechanisms (pipeline stages, forwarding paths, specialized instructions) and software techniques (instruction scheduling, dependency management). We now have a much clearer understanding of how modern processors achieve high performance despite the inherent challenges of pipelined execution.