Lab 2 Report

313553054 陳冠霖

1. Introduction

This lab consists of three tasks: First, extend the instruction set and generate the corresponding VMH and VCD files.Second, design a pipeline bypass to solve data hazards by removing the original stall and adding `is_load` to handle load-use hazards.Third, add two additional stages (X2, X3) and remove the original imuldiv unit, replacing it with a pipeline_muldiv unit that operates independently. This allows the processor to continue running in parallel and only stalls when there is a data dependency.Finally, benchmark the performance of all three designs to understand the improvements achieved with each design iteration.

2. Design

2-1 Implementation

BYP: To ensure that the data read from the registers is always up-to-date, I concentrated on making sure that when the “op0_mux” or “op1_mux”fetches values from the register file, they have access to the most current data, even if the register file hasn't been updated yet. I accomplished this by introducing additional multiplexers, known as “rs_mux”. These multiplexers check whether the required register values were recently updated in the X, M, or W stages. If so, the muxes select the forwarded values instead of causing a pipeline stall.

For instance, when reading from `rs1`, we check the following five conditions:(1)The `rs1` read is enabled.(2)The instruction in the X stage is valid (i.e., a bubble).(3) The instruction in the X stage requires a writeback.(4)The writeback register address in the X stage is non-zero.(5)The `rs1` address matches the writeback register address in the X stage.If all these conditions are true, the “rs1_mux’ is set to choose the bypassed value from the X or M or W stage. This ensures that data dependencies are handled without stalling the pipeline.

After implementing this bypass mechanism, one issue remained with load-use data hazards. Since the load instruction only gets its result in the M stage, bypassing its value during the X stage isn't possible. This necessitates a one-cycle stall to prevent the use of incorrect data. Specifically, I added logic in the D stage to detect if the instruction in the X stage is a load. If it is, and a subsequent instruction depends on the load’s result, a one-cycle stall is inserted. Once the load instruction reaches the M stage, its value is forwarded to the decode stage, allowing the dependent instruction to proceed with the correct data. This solution addresses the load-use

hazard while minimizing stalls.

LONG: Following the design of the bypass, I added the X2 and X3 stages, and introduced two new forwarding paths to both “rs1” and “rs2”. I removed the original “muldiv” unit and replaced it with a “Pipeline_muldiv” unit. This change simplified the “execute_mux_out” to only take “alu_out” as an input, while the “muldiv_sel_Xhl” signal is passed all the way to the X3 stage, achieving parallelism between the “muldiv” and the original pipeline. Finally, I added a multiplexer in the X3 stage to select the final write-back data source.

2-2 Justifications

The key change I made with bypassing was to significantly reduce stalls. In the original design, if an instruction in the decode stage had a Read-After-Write (RAW) data dependency on any of the previous three instructions, the pipeline would stall for up to three cycles, waiting for the data to be written back before allowing the instruction to proceed with the correct values. However, stalling increases cycle count and negatively impacts performance. To address this, I implemented a system that forwards data from the X (execute), M (memory), and W (writeback) stages directly to the decode stage, eliminating most stalls.The decision to introduce a stall only in the case of load-use hazards was based on the fact that load instructions can't forward their results in earlier stages. In this case, a single-cycle stall is required to ensure the program executes correctly, but this keeps the number of stalled cycles to a minimum.

3. Testing Methodology

For example,I run ./riscvstall-sim +disasm=2 +exe=../tests/build/vmh/riscv-addi.vmh to check what was the exact operation in the test case. Additionally, I also use vcd wave images. With both testing methodology ,I successfully debug all 3 objectives.

Finally, full system testing with benchmark programs was conducted to evaluate overall performance improvements.

4. Evaluation

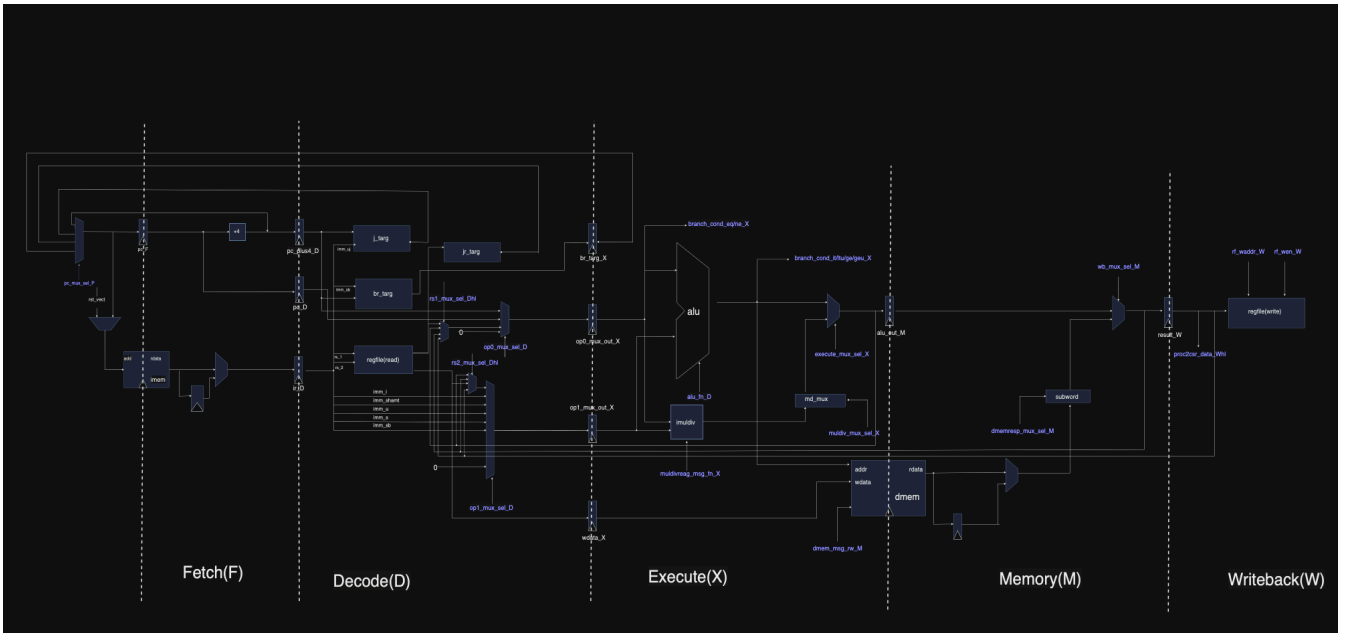
Benchmark	Processor	Num cycles	Num inst	IPC
ubmark-vvadd.c	stall	570	453	0.794737
ubmark-vvadd.c	byp	471	453	0.961783
ubmark-vvadd.c	long	471	453	0.961783
ubmark-cmplx-mult.c	stall	16846	1873	0.111184
ubmark-cmplx-mult.c	byp	15325	1873	0.122219
ubmark-cmplx-mult.c	long	2750	1873	0.681091
ubmark-masked-filt.c	stall	16271	4764	0.292791
ubmark-masked-filt.c	byp	14015	4764	0.339922
ubmark-masked-filt.c	long	6553	4764	0.726955
ubmark-bin-search.c	stall	3096	1030	0.332687
ubmark-bin-search.c	byp	1415	1030	0.727915
ubmark-bin-search.c	long	1415	1030	0.727915

5. Discussion

Through this table, it was observed that across the four test cases, using different processors (stall, bypass, long) can achieve better performance. With the same number of instructions, the number of cycles shows a decreasing trend, while the IPC (Instructions Per Cycle) shows an upward trend. This also demonstrates that utilizing bypassing techniques effectively reduces the number of stall cycles. Additionally, the performance difference between the long and bypassing implementations depends on the number of muldiv operations; the more such operations there are, the more cycles can be saved with the long implementation.

6. Figures

1.Bypassing (5 stages)



2.Pipeline Muldiv (7 stages)