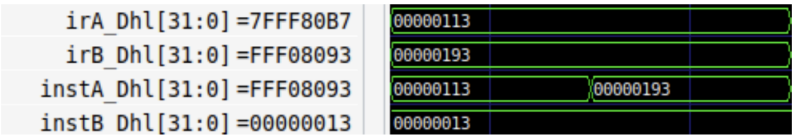
Lab 3 Report

1. Introduction

This assignment focuses on implementing a 2-wide pipeline that can fetch and process two instructions at the same time. The task is divided into two phases: in the first phase, the pipeline processes instructions sequentially, feeding them one at a time. In the second phase, the goal is to load and execute two instructions simultaneously whenever possible. However, if a stall is necessary due to hazards such as WAW or RAW dependencies or other waiting conditions, only one instruction will continue through the pipeline. Additionally, instead of using the original bypassing mechanism, a scoreboard will be implemented to manage dependencies and ensure proper instruction execution.

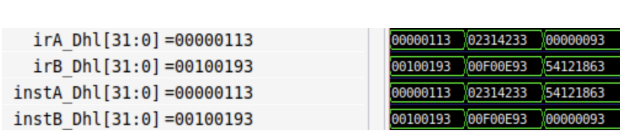
2. Design



a. Implementation Methodology

(1) In the Part 1 of the design, the pipeline operates sequentially, meaning that dependencies between the two fetched instructions are not considered. This simplified approach ensures that instructions enter the pipeline in order, with only `instA_Dhl` actively processed while `instB_Dhl` is treated as a no-operation (NOP). This approach reflects the single-instruction limitation of the pipeline, where only one instruction progresses per cycle. As a result, there is no need to handle dependencies or hazards in this phase, and the secondary pipeline for `instB_Dhl` can be temporarily ignored.

(2) In Part 2 of the design, we need to determine which of the two instructions fetched together should enter pipeline A and which should enter pipeline B. The result and my approach are respectively illustrated in Figure 2 and Figure 3.




```
// Steering Logic
assign instA_Dhl = stall_ls_A_Dhl ? instA_Dhl :
                   stall_double_nALU_reg ? irB_Dhl :
                   steering_mux_sel_Dhl ? irB_Dhl : irA_Dhl;
assign instB_Dhl = stall_ls_A_Dhl ? instB_Dhl :
                   stall_double_nALU_reg ? 32'bx :
                   steering_mux_sel_Dhl ? irA_Dhl : irB_Dhl;
```

We use the signal steering_mux_sel_Dhl to identify cases where the first instruction is an ALU operation and the second is a non-ALU operation. Only in this situation do we need to swap the instructions; otherwise, the default setup sends the first instruction into pipeline A and the second into pipeline B. Next, we consider cases where both instructions are non-ALU operations. Since both instructions need to enter through pipeline A in this scenario, we need to stall the second instruction, allowing the first one to proceed. Only after the first instruction enters can the second follow. This requires handling across two consecutive cycles, so I maintain a

stall_double_nALU_reg register to record the previous cycle's status, allowing the second instruction to enter the pipeline on the second cycle.

Finally, if a store instruction precedes a load, data hazards may arise that aren't visible by register number alone, as offsets also play a role. To prevent address conflicts, we stall the load instruction whenever it follows a store instruction. This ensures all hazards are avoided by pausing the load in load-after-store scenarios.

b. Design Justification

The 2-wide pipeline is designed to maximize throughput while managing hazards between fetched instructions. Separate stall signals (`stall_A` and `stall_B`) enable independent stalling, allowing one instruction to proceed while the other waits, optimizing efficiency.

Key signals like `stall_SB_Dhl`, `stall_RAW_Dhl`, and `stall_WAR_Dhl` detect and handle hazards such as RAW, WAR, and Store-Load conflicts by tracking pending registers and ensuring data correctness. The `stall_double_nALU_reg` manages cases where both instructions require the same pipeline, staggering their entry over two cycles to preserve order. Load-after-store hazards are also resolved by stalling load instructions with potential address conflicts, ensuring safe memory access.

3. Testing Methodology

To verify data hazards and ensure correct handling of edge cases, I designed assembly tests that specifically target Write-After-Write (W AW) and Read-After-Write (RAW) scenarios. Here's a breakdown of each hazard and how the test checks for correctness:

a. WAW Hazard
The instruction la x5, tdata_0 loads the address of tdata_0 into x5.
lw x1, 4(x5) loads the value at the specified memory address into x1.
li x1, 2 immediately overwrites x1 with the value 2.
TEST_CHECK_EQ(x1, 2) confirms that the latest write operation to x1 has the correct value (2), validating that the processor correctly handles the overwrite without unintended behavior from the earlier lw operation. This test ensures that any W AW hazard handling logic allows the second write to execute as expected.
b. RAW Hazard
la x8, tdata_0 loads the address of tdata_0 into x8.
lw x6, 4(x8) loads the value from memory into x6.
addi x7, x6, 1 performs an addition using x6 as an operand, with the expectation that the loaded value is correctly read and used.
TEST_CHECK_EQ(x7, 0x00007f01) checks that x7 contains the expected result.

This ensures that the RAW dependency between x6 and x7 is managed correctly by the pipeline, with x7 obtaining the correct value from x6 without any bypassing or stalling issues.

These tests are effective in validating the processor's ability to handle WAW and RAW dependencies by checking that the final register values match expectations even in the presence of data hazards.

4. Evaluation

Benchmark	Processor	Num cycles	Num inst	IPC
ubmark-vvadd.c	dualfetch	489	453	0.926380
ubmark-vvadd.c	ssc	436	453	1.038990
ubmark-vvadd.c	long	471	453	0.961783
ubmark-cmplx-mult.c	dualfetch	2979	1873	0.628734
ubmark-cmplx-mult.c	ssc		1873	
ubmark-cmplx-mult.c	long	2750	1873	0.681091
ubmark-masked-filt.c	dualfetch	7345	4764	0.648604
ubmark-masked-filt.c	ssc	5086	4764	0.936689
ubmark-masked-filt.c	long	6553	4764	0.726955
ubmark-bin-search.c	dualfetch	1687	1030	0.610551
ubmark-bin-search.c	ssc		1030	
ubmark-bin-search.c	long	1415	1030	0.727915

5. Discussion

IPC Performance and Cycle Counts Summary: For ubmark-vvadd.c, SSC achieves the highest IPC (1.038990), surpassing dualfetch (0.926380) and long (0.961783), showcasing its efficiency in vector addition. In ubmark-cmplx-mult.c, the long architecture performs better (0.681091) than dualfetch (0.628734), but SSC data is incomplete. For ubmark-masked-filt.c, SSC leads (0.936689), followed by long (0.726955) and dualfetch (0.648604), benefiting from better branch handling. In ubmark-bin-search.c, the long architecture excels with an IPC of 0.727915, highlighting its optimization for binary search.Dualfetch often has higher cycle counts, reflecting inefficiency in handling computationally or branch-heavy tasks, while SSC consistently shows lower cycle counts, emphasizing its effective instruction scheduling.

Pipeline and Architectural Insights: SSC excels in managing dependencies and

memory operations, as seen in ubmark-vvadd.c and ubmark-masked-filt.c. The long architecture performs well with fewer dependencies, as in ubmark-cmplx-mult.c and ubmark-bin-search.c. Dualfetch shows lower IPC and higher cycles, indicating inefficiency, while SSC proves versatile across workloads, albeit with potentially higher implementation complexity.

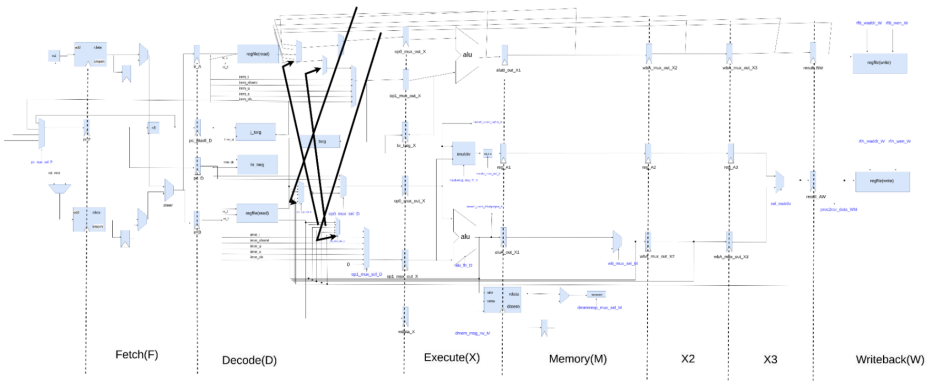
Performance Analysis Based on .S Test Results

Given the limitations preventing completion of the ubmark benchmarks, I continued with analysis based on the .S test results.

- **Dual-fetch, dual-issue (riscvssc):** Fetches and issues two instructions per cycle, maximizing parallelism when dependencies are low. Its high IPC of 1.46 reflects efficient handling of independent instructions, reducing stalls and cycle counts.
- **Dual-fetch, single-issue (riscvdualfetch):** Fetches two instructions per cycle but issues only one, limiting its effectiveness under dependencies. Increased stalls in the decode stage reduce efficiency, resulting in a lower IPC of 0.97, slightly underperforming compared to riscvlong.
- **Single-fetch, single-issue (riscvlong):** Fetches and issues one instruction per cycle, maintaining simplicity and minimizing dependency stalls. Its IPC of 0.98 slightly exceeds riscvdualfetch, benefiting from a streamlined single-instruction path without the overhead of managing dual instructions.

6. Figures

a. Dual-Issue I4 Processor



b. Scoreboard