University of South Florida

CDA 4205L Computer Architecture Lab

Lab Report

Semester: Spring 2025		
Experiment	*Number:*	11
	Date:	04/02/2025
Lab	*Section:*	04
	Lab TA:	Rupal Agarwal
Report	*Due Date:*	04/09/2025
Group	*Member #1 name:*	Claude Watson
	Member #2 name:	Anirudh Kasyap Sesham

RUBRIC SUMMARY

[100%] Final submission/report

ABSTRACT:

This lab focuses on understanding pipelined CPU datapaths, hazard conditions, and how CPUs handle them. Using a RISC-V simulator, we analyzed several assembly code examples to identify data hazards and observe how forwarding mechanisms can mitigate performance impacts. We ran programs with and without forwarding enabled, comparing cycle counts and examining execution tables to understand pipeline stalls. Through this investigation, we learned how different types of hazards affect CPU performance and how both forwarding techniques and code rearrangement can significantly improve execution efficiency in pipelined architectures.

INTRODUCTION:

Pipelining is a fundamental technique in modern CPU design that improves performance by allowing multiple instructions to be processed simultaneously at different stages. This lab explores the 5-stage pipeline commonly found in RISC-V implementations, consisting of Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB) stages. While pipelining increases instruction throughput, it also introduces hazards that can stall the pipeline and reduce performance.

The three types of hazards explored in this lab are structural hazards (contention for hardware resources), data hazards (dependencies between instructions), and control hazards (unpredictable changes in control flow). Using an online RISC-V simulator, we examined how these hazards manifest in assembly code and investigated mitigation techniques such as forwarding and code rearrangement.

METHOD:

WebRISCV online simulator (https://webriscv.dii.unisi.it/) - A web-based RISC-V simulator that implements a 5-stage pipelined datapath

Microsoft Excel - Used for creating bar plots comparing cycle counts

Web browser - For accessing the online simulator

Text editor - For examining and modifying assembly code files

Files

- data_hazard.asm - Example RISC-V assembly code with data hazards provided by the simulator
- factorial.asm - RISC-V implementation of a factorial calculation provided by the simulator
- example.asm - Modified assembly code from Lab 10 used for comparative analysis

- example_rearranged.asm - Optimized version of example.asm using Lab 10 algorithm

RESULTS:

Task 1: Analyze the code now, before proceeding with the lab. What hazards are present, and where? For the first hazard, what specific conditions will the Hazard Detection Unit identify?

- There are data hazards between *add t2, t0, t1* and *add t3, t0, t2*
  - t2 is being written in instruction 3 and read in instruction 4.
  - Read After Write: instruction 4 depends on a value not yet written by instruction 3.
- Between add t2, t0, t1 and add t4, t0, t2
  - Similar issue: instruction 5 reads t2 which is written in instruction 3.

The hazard detection unit will detect that the instruction writing to t2 has not yet reached the write-back stage when the next instruction attempts to read it. Therefore, the unit flags a RAW hazard and triggers a stall or forwarding mechanism to resolve.

Task 2: How many total cycles were required? Include a screenshot of the multicycle pipeline diagram (click on Execution Table).

- A total of 9 total cycles were needed.

EXECUTION TABLE

FULL LOOPS	CPU Cycles								
Instruction	1	2	3	4	5	6	7	8	9
addi t0, x0, 1	F	D	X	M	W				
addi t1, x0, 2		F	D	X	M	W			
add t2, t0, t1			F	D	X	M	W		
add t3, t0, t2				F	D	X	M	W	
add t4, t0, t2					F	D	X	M	W

Task 3: What happens to the pipeline when a hazard is encountered? Why?

- There is a stall in the ID stage when a hazard is encountered. This happens because we must allow the earlier instruction time to complete its data write before the dependent instruction proceeds. This ensures the CPU maintains correct data flow and avoids using incorrect or incomplete values.

Task 4: How many cycles are required now that forwarding is deactivated? Take a screenshot of the execution table

- A total of 13 cycles are required

EXECUTION TABLE

FULL LOOPS

	CPU Cycles												
Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13
addi t0, x0, 1	F	D	X	M	W								
addi t1, x0, 2		F	D	X	M	W							
add t2, t0, t1			F	-	-	D	X	M	W				
add t3, t0, t2						F	-	-	D	X	M	W	
add t4, t0, t2									F	D	X	M	W

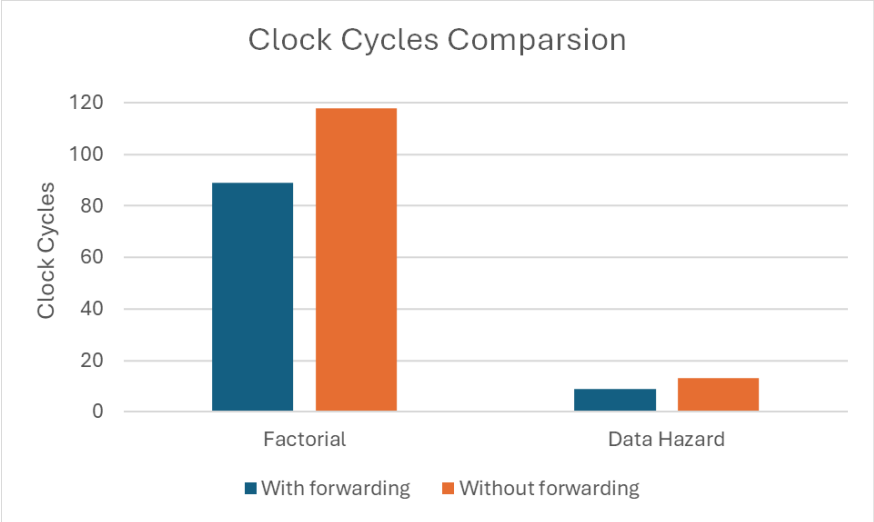
Task 5: Does forwarding address all of the pipeline hazards present in the factorial code? If not, which ones does it not address?

- Forwarding does not address jal, beq and jr instructions. These require pipeline flushing or stalling because the target address or branch outcome isn't known early enough. Forwarding mainly helps with data hazards, but control hazards remain unresolved.

T6: What impact did the optimizations from Lab 10 have on the cycle count in cases where forwarding was and was not active?

- The Lab 10 optimizations had a noticeable impact on the cycle count for both forwarding and unforwarding cases. When forwarding was not active, the optimized version (example_rearranged.asm) ran in 46 cycles compared to 49 cycles for the unoptimized version (example.asm), showing a 3-cycle improvement purely from reordering instructions to avoid hazards. With forwarding enabled, the optimized code completed in 39 cycles versus 42 cycles for the unoptimized version, again showing a 3-cycle gain. These results demonstrate that instruction scheduling effectively reduces stalls from data hazards, especially load-use hazards and back-to-back dependencies. Even with hardware forwarding, software-level reordering can still enhance performance by smoothing out remaining hazards. Overall, the Lab 10 algorithm complements hardware techniques and contributes to a more efficient pipeline.

T7: Generate a bar plot (you can use Excel to create this) that compares the number of cycles, both with and without forwarding enabled, for the data hazard and factorial examples. Label the axes and title appropriately.



DISCUSSION:

The results demonstrate the significant impact that both hardware techniques (forwarding) and software optimizations (code rearrangement) can have on pipelined CPU performance. Forwarding provides a hardware solution that efficiently handles most data hazards without requiring pipeline stalls, resulting in cycle count reductions of approximately 25-30% in our examples. However, certain hazards still require stalls even with forwarding enabled, particularly load-use hazards and control hazards related to branching. Code rearrangement techniques, as implemented in the optimized example from Lab 10, proved to be highly effective, especially when forwarding was disabled. This highlights the importance of compiler optimizations that can arrange instructions to minimize dependencies and maximize pipeline utilization. In practical systems, the combination of hardware forwarding and intelligent code scheduling provides the best results. Future work could explore more advanced techniques such as branch prediction, speculative execution, and out-of-order execution that further mitigate pipeline hazards. Additionally, investigating how different compiler optimization levels affect the generated code's pipeline performance would provide valuable insights into the interplay between software and hardware optimization strategies.

CONCLUSION:

This lab provided hands-on experience with pipelined CPU execution, focusing on identifying and addressing pipeline hazards. Using the WebRISCV simulator, we analyzed how data dependencies in code impact performance and how techniques like forwarding and code rearrangement can mitigate these issues. We observed that forwarding significantly reduces the cycle count by eliminating many pipeline stalls, while intelligent code rearrangement can further optimize performance. The execution tables clearly demonstrated how hazards cause pipeline bubbles, and the comparative analysis quantified

the benefits of each optimization technique. This lab reinforced the importance of understanding both hardware and software approaches to CPU optimization, showing that the highest performance comes from effectively combining both strategies.