# University of South Florida

## CDA 4205L Computer Architecture Lab

# Lab Report

| Semester: Spring 2025 | | |
|---|---|---|
| ***Experiment*** | *Number:* | 10 |
| | *Date:* | 03/26/2025 |
| | | |
| ***Lab*** | *Section:* | 04 |
| | *Lab TA:* | Rupal Agarwal |
| | | |
| ***Report*** | *Due Date:* | 04/02/2025 |
| | | |
| ***Group*** | *Member #1 name:* | Claude Watson |
| | *Member #2 name:* | Anirudh Kasyap Sesham |

**RUBRIC SUMMARY**

*[100%] Final submission/report*

## ABSTRACT:

*In this lab, we tackled the challenge of identifying and eliminating data hazards in assembly code. Our goal was to shuffle around instructions while keeping the program's functionality intact, but getting rid of those pesky pipeline stalls. We built a system that analyzes instruction dependencies and cleverly reorders them to make the code run more efficiently.*

## INTRODUCTION:

Pipeline hazards (especially data hazards) can really slow down CPU performance. This lab dives into ways to spot these hazards and optimize code by moving instructions around. We implemented an algorithm that breaks assembly code into chunks, figures out which instructions depend on each other, and rearranges them to minimize stalls.

## METHOD:

**Software Tools Used**

- **Python** – Used to implement our instruction reordering algorithm
- **Google Colab** – The online environment where we developed and tested our code
- **RISC-V Instruction Set** – The assembly language we analyzed for hazards

**Files**

- **example.asm – Sample assembly code provided for testing our implementation**
- **Lab10.ipynb – The Python notebook containing our implementation**
- **splitAssemblyIntoSubsets() – Function that divides code into logical blocks**
- **are_data_dependent() – Function that detects dependencies between instructions**

# RESULTS:

Task 1:

```
Testing splitAssemblyIntoSubsets() with:
        ['main:']
        ['lw', 'x4', 'x0', '16']
        ['add', 'x5', 'x2', 'x4']
        ['sub', 'x7', 'x3', 'x6']
        ['loop:']
        ['sw', 'x5', 'x0', '32']
        ['lw', 'x4', 'x0', '16']
        ['addi', 'x9', 'x1', 'x4']
        ['bne', 'x9', 'x1', 'x4']
        ['lw', 'x4', 'x0', '16']
        ['add', 'x5', 'x1', 'x4']
        ['j', 'x5']
        ['sw', 'x5', 'x0', '32']

Correct answer:
1.      ['main:']

2.      ['lw', 'x4', 'x0', '16']
        ['add', 'x5', 'x2', 'x4']
        ['sub', 'x7', 'x3', 'x6']

3.      ['loop:']

4.      ['sw', 'x5', 'x0', '32']
        ['lw', 'x4', 'x0', '16']
        ['addi', 'x9', 'x1', 'x4']
        ['bne', 'x9', 'x1', 'x4']

5.      ['lw', 'x4', 'x0', '16']
        ['add', 'x5', 'x1', 'x4']
        ['j', 'x5']

6.      ['sw', 'x5', 'x0', '32']


Returned answer:
1.      ['main:']

2.      ['lw', 'x4', 'x0', '16']
        ['add', 'x5', 'x2', 'x4']
        ['sub', 'x7', 'x3', 'x6']

3.      ['loop:']

4.      ['sw', 'x5', 'x0', '32']
        ['lw', 'x4', 'x0', '16']
        ['addi', 'x9', 'x1', 'x4']
        ['bne', 'x9', 'x1', 'x4']

5.      ['lw', 'x4', 'x0', '16']
        ['add', 'x5', 'x1', 'x4']
        ['j', 'x5']

6.      ['sw', 'x5', 'x0', '32']
```

Task 2:

```
t2_test()
```

```
Testing get_operands() with:
['lw', 'x4', 'x0', '16']
Correct answer:
('x4', 'x0')
Returned answer:
('x4', 'x0')

Testing get_rd() with:
['lw', 'x4', 'x0', '16']
Correct answer:
x4
Returned answer:
x4

Testing get_rs() with:
['lw', 'x4', 'x0', '16']
Correct answer:
x0
Returned answer:
x0

Testing are_data_dependent() with:
['lw', 'x4', 'x0', '16']
['add', 'x5', 'x1', 'x4']
Correct answer:
True
Returned answer:
True
```

Task 3:

```
t3_test()
```

```
Testing find_above_instruction_without_dependencies() with:
['lw', 'x2', 'x0', '16']
['addi', 'x4', 'x6', '37']
['sub', 'x5', 'x1', 'x4']
Correct answer: 0 (['lw', 'x2', 'x0', '16'])
Returned answer: 0 (['lw', 'x2', 'x0', '16'])

Testing find_above_instruction_without_dependencies() with:
['lw', 'x2', 'x0', '16']
['addi', 'x4', 'x6', '37']
['sub', 'x5', 'x1', 'x4']
['add', 'x5', 'x1', 'x4']
['sw', 'x5', 'x0', '32']
Correct answer: 0 (['lw', 'x2', 'x0', '16'])
Returned answer: 0 (['lw', 'x2', 'x0', '16'])

Testing find_above_instruction_without_dependencies() with:
['lw', 'x2', 'x0', '16']
['addi', 'x4', 'x6', '37']
Correct answer: False
Returned answer: False

Testing find_above_instruction_without_dependencies() with:
['lw', 'x8', 'x9', '42']
['add', 'x7', 'x7', 'x9']
['sub', 'x8', 'x6', 'x10']
['mul', 'x6', 'x0', 'x3']
['add', 'x3', 'x1', 'x4']
['beq', 'x2', 'x3', 'loop']
Correct answer: 1 (['add', 'x7', 'x7', 'x9'])
Returned answer: 1 (['add', 'x7', 'x7', 'x9'])
```

Task 4:

```
t4_test()
```

Code cell output actions

```
perands() with:
['lw', 'x4', 'x0', '16']
['add', 'x5', 'x1', 'x4']
['add', 'x5', 'x1', 'x4']
['sw', 'x5', 'x0', '32']

Correct answer:
['lw', 'x4', 'x0', '16']
['sw', 'x5', 'x0', '32']
['add', 'x5', 'x1', 'x4']
['add', 'x5', 'x1', 'x4']

Returned answer:
['lw', 'x4', 'x0', '16']
['sw', 'x5', 'x0', '32']
['add', 'x5', 'x1', 'x4']
['add', 'x5', 'x1', 'x4']
```

Task 5:

```
t5_test()
```

```
Testing reorder_instructions() with:
['lw', 'x1', 'x0', '0']
['lw', 'x2', 'x0', '8']
['add', 'x3', 'x1', 'x2']
['sw', 'x3', 'x0', '24']
['lw', 'x4', 'x0', '16']
['add', 'x5', 'x1', 'x4']
['sw', 'x5', 'x0', '32']

Correct answer:
['lw', 'x1', 'x0', '0']
['lw', 'x2', 'x0', '8']
['lw', 'x4', 'x0', '16']
['add', 'x3', 'x1', 'x2']
['add', 'x5', 'x1', 'x4']
['sw', 'x3', 'x0', '24']
['sw', 'x5', 'x0', '32']

Returned answer:
['lw', 'x1', 'x0', '0']
['lw', 'x2', 'x0', '8']
['lw', 'x4', 'x0', '16']
['add', 'x3', 'x1', 'x2']
['add', 'x5', 'x1', 'x4']
['sw', 'x3', 'x0', '24']
['sw', 'x5', 'x0', '32']
```

Task 6:

```
t6_test(filename)

Original                        Reordered
['load_use1:']                  ['load_use1:']
['lw', 't1', 't1', '12']        ['lw', 't1', 't1', '12']
['add', 't5', 't1', 't7']       ['add', 't5', 't1', 't7']
['sub', 't8', 't6', 't7']       ['sub', 't8', 't6', 't7']
['or', 't9', 't6', 't7']        ['or', 't9', 't6', 't7']
['load_use2:']                  ['load_use2:']
['lw', 't1', 't1', '12']        ['lw', 't1', 't1', '12']
['add', 't5', 't1', 't7']       ['add', 't5', 't1', 't7']
['sub', 't1', 't6', 't7']       ['sub', 't1', 't6', 't7']
['or', 't9', 't6', 't7']        ['or', 't9', 't6', 't7']
['no_dep:']                     ['no_dep:']
['lw', 't0', 't1', '12']        ['lw', 't0', 't1', '12']
['add', 't5', 't6', 't7']       ['add', 't5', 't6', 't7']
['sub', 't2', 't0', 's1']       ['sub', 't2', 't0', 's1']
['or', 's5', 's6', 't6']        ['or', 's5', 's6', 't6']
['alu_then_branch:']            ['alu_then_branch:']
['sub', 't3', 't4', 't5']       ['sub', 's3', 's3', 't0']
['sub', 's3', 's3', 't0']       ['add', 't0', 't1', 't2']
['add', 't0', 't1', 't2']       ['sub', 't3', 't4', 't5']
['beq', 't0', 't5', 'loop']     ['beq', 't0', 't5', 'loop']
['load_then_branch:']           ['load_then_branch:']
['add', 't4', 't6', 't7']       ['add', 't4', 't6', 't7']
['sub', 't2', 't3', 's1']       ['lw', 't0', 't1', '0']
['lw', 't0', 't1', '0']         ['sub', 't2', 't3', 's1']
['beq', 't0', 't5', 'loop']     ['beq', 't0', 't5', 'loop']
['fix_no_steal:']               ['fix_no_steal:']
['lw', 't0', 't1', '12']        ['lw', 't0', 't1', '12']
['add', 't5', 't0', 't7']       ['add', 't5', 't0', 't7']
['sub', 't2', 't4', 't6']       ['or', 's5', 't0', 't6']
['or', 's5', 't0', 't6']        ['sub', 't2', 't4', 't6']
['add', 's3', 's5', 's6']       ['add', 's3', 's5', 's6']
['handshake:']                  ['handshake:']
['lw', 't0', 't1', '12']        ['lw', 't0', 't1', '12']
['add', 't5', 't0', 't7']       ['sub', 's5', 't4', 't6']
['sub', 's5', 't4', 't6']       ['or', 's5', 't7', 't6']
['or', 's5', 't7', 't6']        ['add', 't5', 't0', 't7']
['add', 's3', 's5', 's6']       ['add', 's3', 's5', 's6']
```

Task 7: What are pipeline hazards? Give one example and provide a sample of assembly code that would cause such a hazard.

Pipeline hazards are situations in pipelined processors where the next instruction cannot execute in the following clock cycle, which prevents the pipeline from executing smoothly. Hazards disrupt the normal flow of instructions and reduce performance if not handled properly. An example of a pipeline hazard is a data hazard, in which an instruction depends on the result of a previous instruction that hasn't completed yet.
Assembly:
sub x5, x6, x7 # Compute a result and store in x5
beq x5, x0, label # Immediately use x5 in a branch condition — hazard!
This would cause a hazard since the branch decision is made in the decode stage, and sub writes back in the write-back stage, the value of x5 might not be ready yet.

Task 8: How can CPU hardware deal with hazards? How can a compiler help?

· The CPU handles pipeline hazards using techniques like forwarding, which passes values between pipeline stages without waiting for them to be written back. When forwarding isn't possible, the CPU inserts stall cycles (bubbles) to delay dependent instructions. For control hazards, the CPU uses branch prediction to guess the outcome of branches and keep the pipeline moving. If the guess is wrong, the pipeline is flushed and corrected. Compilers help by reordering instructions to separate dependent operations and reduce stalls. They can also fill branch delay

slots with useful instructions and apply register renaming to avoid unnecessary dependencies.

## DISCUSSION:

This lab gave us valuable insights into the real-world challenges of optimizing code for pipelined architectures. We discovered that finding data dependencies is relatively straightforward, but determining which instructions can be safely moved without changing program behavior is much more complex. The most challenging aspect was handling instructions with multiple source and destination registers, especially when dealing with memory operations. Our algorithm performed well on simple code segments but might struggle with more complex scenarios involving multiple interdependent instructions. We noticed that sometimes reordering one hazard would inadvertently create another, highlighting the importance of a holistic approach to optimization.

## CONCLUSION:

The experience of implementing a hazard detection and resolution system has deepened our understanding of both compiler design and processor architecture. We've learned that effective pipeline optimization requires a delicate balance between hardware solutions (like forwarding) and software techniques (like instruction reordering).

Looking forward, this knowledge will be valuable in any system design scenario where performance optimization is critical. The principles we've learned about dependency analysis and code transformation extend beyond RISC-V to any pipelined system. While our implementation was relatively simple, it demonstrates the fundamental concepts that underpin sophisticated commercial compilers and optimization tools.