CS2053 - Computer Architecture

Department of Computer Science and Engineering - University of Moratuwa

Lab 7: Instruction Pipelines

Learning Outcomes:

After completing this lab, you will be able to:

- Clearly define the concept of instruction pipelining and its importance in computer systems. Emphasise its significance in enhancing instruction execution performance and system efficiency.
- Demonstrate the ability to evaluate instruction execution performance metrics, such as throughput and speedup. Understand their implications on the overall efficiency of computer systems.
- Explain the need for and the methods of instructions pipelining in RISC-V & recognize the patterns in pipeline diagrams that benefit from pipelining
- Recognize and describe various pipeline issues, including pipeline hazards and stalls. Understand the complexities involved in managing these issues and the strategies employed to mitigate them effectively.

Introduction

Pipelining

Instruction pipelining in CPU is another important technique used in computer architecture which enhances the speed and throughput of program execution. It involves organizing a series of sequential tasks into a pipeline, allowing certain stages to be executed in parallel if necessary.

In RISC-V architecture, one example of an instruction pipeline is a 5-stage pipeline which contains the following stages:

IF	- Instruction Fetch
ID	- Instruction Decode
EX	- Execution
MEM	- Memory Access
WB	- Write back

Even though instruction pipelining lets you increase the throughput of the CPU, there can be several issues associated with pipelining which may introduce additional overhead to the program. In this lab, you will get familiar with such issues and how to overcome them as well.

Example :

Here we look at a simple sequential assembly program and see how the instructions are executed in a processor architecture without instructions pipelining and with instructions pipelining. Paste the following assembly code into the code editor in the Ripes simulator.

```
.data
        array: .word 10, 20, 30, 40, 50 # Initialize an array of integers
.text
        la a1, array # Load data from memory into register a1

        lw a0, 0(a1) # Load first element of the array to register a0
        lw a0, 4(a1) # Load second element of the array to register a0
        lw a0, 8(a1) # Load third element of the array to register a0
        lw a0, 12(a1) # Load fourth element of the array to register a0
        lw a0, 16(a1) # Load fifth element of the array to register a0
```

Now go to the processor view of the simulator by clicking on the Processor icon on the left sidebar.

To change the processor architecture, click on the icon on the toolbar on top, and select the RISC-V 32-bit Single-cycle processor first. Click OK to apply the selected processor.

Now execute the program instruction by instruction using the > icon on the toolbar, and observe the currently executing instruction appearing at the top of the processor diagram. Also, observe the pipeline diagram for the program execution (explained below).

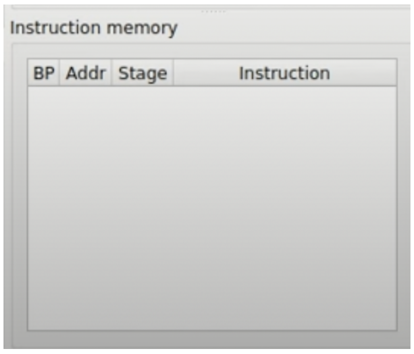
Then change the processor architecture to RISC-V 32-bit 5-stage processor, and again run the program. Note that the processor diagram is now changed with the additional registers to control the pipeline. Also note that as you pass through the instructions, the instructions that are currently on a pipeline stage appear on top of the processor diagram. Again observe the pipeline diagram for the program execution, and see how the instruction level parallelism is achieved.

You can also try the same program for other processor architectures available in the Ripes simulator as well. They are designed for different use cases which will be explored in the latter exercises of this lab.

Observe pipeline diagrams in Ripes :

In this section, we'll observe instruction pipelining in RISC-V.

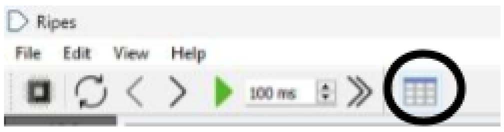
Instruction memory is the place where we can see what are the instructions which are in the pipeline in a given time.



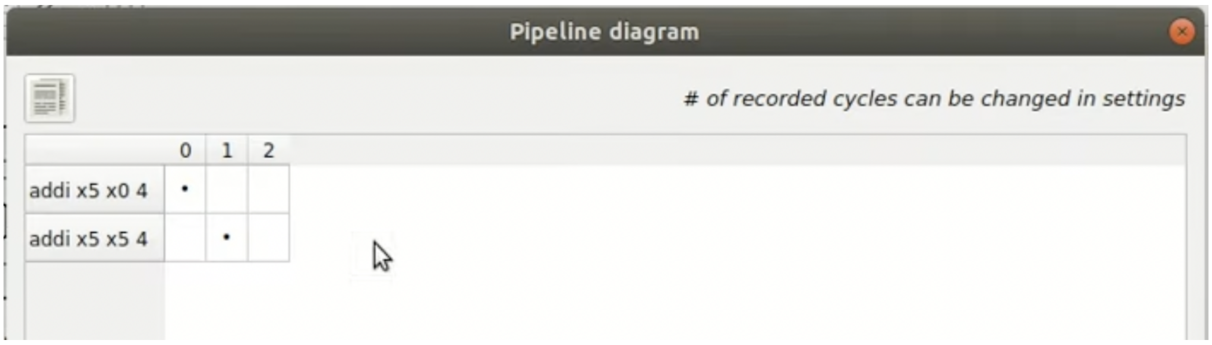
You can use various processors by selecting one from the processor tab.

Pipeline Diagram

In order to view the Pipeline diagram, you need to click on the button which is highlighted in the following diagram.



You can view the Pipeline diagram as follows.



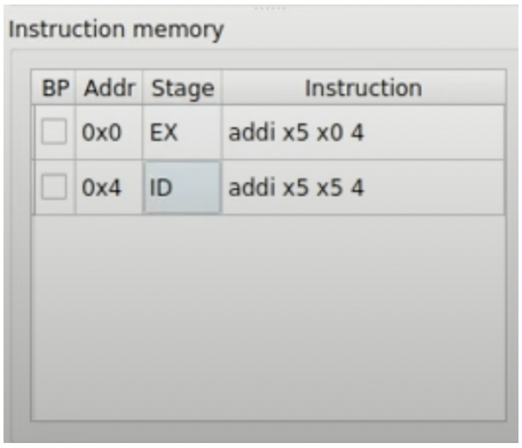
Pipeline diagram

# of recorded cycles can be changed in settings

	0	1	2
addi x5 x0 4	•		
addi x5 x5 4		•	

This shows in which cycle that particular instruction gets completed.

Instruction memory


Instruction memory

BP	Addr	Stage	Instruction
☐	0x0	EX	addi x5 x0 4
☐	0x4	ID	addi x5 x5 4

As the above table suggests, the top instruction is in the "Execution" stage and the second instruction is in the "Instruction Decode" stage.

As above you can observe and get a deeper understanding of how pipeline works in RISC-V.

To further understand instruction pipelining in the Ripes simulator, you can refer to this video as well. https://www.youtube.com/watch?v=Zv515YgYzU4

Exercise 1 - Control hazard detection pipeline

In a pipelined RISC-V processor, control hazards are a type of hazard that can occur when instructions that modify the control flow of the program (e.g., branches or jumps) are encountered.

Using the following code we can observe how Ripes Simulator executes control hazard detection.

```
main:
        addi x1, x0, 2
        addi x2, x0, 3
        addi x3, x0, 5

        addi x4, x0, 0
        addi x5, x0, 1
        addi x6, x0, 2

        add x1, x1, x2
        beq x1, x3, branch1

        add x4, x4, x5

branch1:
        add x4, x4, x6
```

Read and understand the code. Three values are loaded into registers x1, x2 and x3. The values of x1 and x2 are added and the result is stored in x1. Then, if the values of x1 and x3 are equal, the execution is switched to branch1 and the value of x6 is loaded into register x4. Otherwise, the same flow is carried out and the value of x5 is loaded into x4.

According to the current initial values, x4 should contain 2 at the end of execution.

First, run the program with the processor '5-stage processor' and observe the instruction memory. Notice how once the 'beq' instruction is executed, the fetched and decoded instructions are erased and the instruction in branch 1 is fetched again. This is known as flushing the pipeline and it is the mechanism used in Ripes to avoid control hazards. The end result is as expected.

Then, run the program with the '5-stage processor w/o hazard detection' and observe the same. Notice how even when the branching instruction is executed, branching doesn't occur and instructions are fetched, decoded and executed sequentially. Therefore, the end results are not as expected and the program is executed incorrectly. In the RISC-V implementation in Ripes, control hazards can result in incorrect program execution.

Read the description given when selecting the processor w/o hazard detection. According to it nop's can be used to stall the pipeline to avoid hazards. Try to implement it.

Exercise 2 - Forwarding pipeline

Data hazards occur when subsequent instructions depend on the results of prior instructions before those results are written back to the registers. **Forwarding (or data hazard detection)** is a mechanism to address these hazards. Forwarding, also known as data hazard detection or data forwarding, is a technique used in pipelined microarchitectures to handle data hazards, particularly the read-after-write (RAW) hazard.

Using the following assembly code we can observe how to observe data hazards in Ripes.

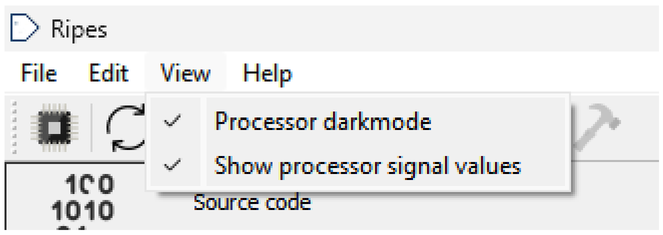
```
.data

.text
        # Initialize registers with values
        li x5, 5 # Load immediate value 5 into x5
        li x6, 3 # Load immediate value 3 into x6
        li x7, 4 # Load immediate value 4 into x7
        li x8, 2 # Load immediate value 2 into x8
```

add x10, x5, x6 # Instruction 1: x10 = x5 + x6
sub x11, x10, x7 # Instruction 2: x11 = x10 - x7
and x12, x10, x8 # Instruction 3: x12 = x10 & x8

Read and understand the code. In this code, it is clear that the *sub* & *and* instructions depend on the result of the *add* instruction. So there's a data hazard when this is executed as pipelined instructions.

To observe data hazard stalls, execute the above code line by line from *add* instruction onwards.

Next, let's understand how data hazard stall is mitigated using forwarding. First, understand the architectural differences between '5-Stage processor' and '5-Stage processor w/o forwarding unit'. Change the layout to *extended* from *standard* when selecting each processor. You can notice some extra multiplexers and a forwarding unit are added in the '5-Stage processor'. These are used to choose the values in registers when no data hazard is detected and to choose the results from the instructions in MEM or WB stages when a data hazard is detected.
To view the values stored in these registers enable the 'Show processor signal values' option in the view menu.



View register values

Select the '5-Stage processor' and run the code line-by-line from the *add* instruction onwards. Observe how the values get stored in intermediate registers.
Can you explain how the architectural changes in the '5-Stage processor' mitigate data hazards?