

# Project: Report Document

Daniel Alejandro Marin - R11858881

Texas Tech University  
CS3375 - Computer Architecture  
Instructor: Dr. Juan Carlos Rojas

December 6<sup>th</sup>, 2024

## Contents

|          |  |          |
|----------|--|----------|
| <b>1</b> | <b>Introduction</b>                            | <b>2</b> |
| <b>2</b> | <b>Design</b>                                  | <b>2</b> |
| 2.1      | Instruction Scheduler Abstract Class . . . . . | 2        |
| <b>3</b> | <b>Methodology</b>                             | <b>3</b> |
| <b>4</b> | <b>Tests</b>                                   | <b>3</b> |
| <b>5</b> | <b>Results</b>                                 | <b>3</b> |
| <b>6</b> | <b>Discussion</b>                              | <b>3</b> |

# 1 Introduction

Instruction scheduling plays a crucial role in modern computer architecture, especially for achieving high performance in multi-issue processors; they allow for faster instruction throughput. This project aims to simulate the scheduling of ‘assembly’ instructions under various processor configurations. The configurations developed in this project are the following:

- Single-issue Instruction Scheduler (in-order)
- Superscalar Instruction Scheduler (in-order)
- Superscalar Instruction Scheduler (out of order)

For each of these configurations there exists a version with register renaming and one without. In this project, we will simulate the scheduling of a simple assembly instruction set, in each of these configurations.

Throughout this report document, we will be explaining the design, implementation details, test and results of each configuration. The insights gained will highlight the advantages and limitations of these techniques in processor architectures.

# 2 Design

The instruction scheduling simulation system is designed as a hierarchy of classes that simulate different types of processor configurations for instruction fetching and retirement. The design uses abstraction and inheritance to encapsulate common functionality while allowing customization for specific scheduling techniques like: register renaming, in-order retirement, etc. . . Following is a class diagram that encapsulates the core design of this project:

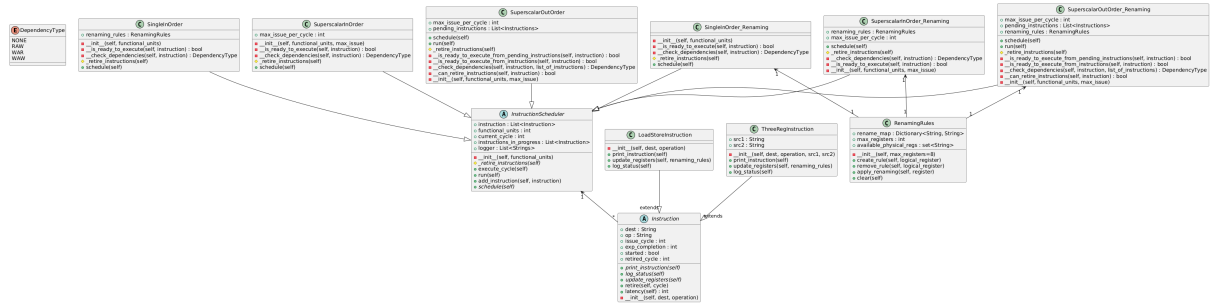


Figure 1: Class Diagram of Design, developed using the Plant UML tools.

The diagram in figure 1 encapsulates the logic that helped develop this simulation. In the following subsections, I will be explaining the logic and reasons behind each class seen in here.

## 2.1 Instruction Scheduler Abstract Class

The InstructionScheduler abstract class defined the layout all configurations of instructions schedulers should follow. It defined the main components and methods they should have, and contained some of the overlapping logic that remained consistent throughout all configurations. It contained the following properties:

- **instructions:** this property looks to contain a list of instructions that need to be scheduled in the simulation.
- **functional\_units:** this property represents the number of parallel functional units the scheduler has. It is a property defined by the user.
- **current\_cycle:** this property contains the current cycle of execution of the processor.
- **instructions\_in\_progress:** this property contains the list of instructions that are currently being executed.

- `logger`: this property contains a ‘log book’ of the logs performed by the scheduler, used for debug statements and keeping track of instructions in the order they retire (outputs).

These are the properties all schedulers contained, additional properties can be added to the subclasses of `InstructionScheduler` when needed. Now, let’s look at the methods defined in this parent class in the following itemize:

- `__init__(self, functional_units=1)`: this constructor method is used to create instances of this class, and it is called by the children classes as to initialize the aforementioned properties.
- `add_instruction(self, instruction)`: this method is in charge of adding an instruction to instructions. It is used to add parsed instructions set that we want to simulate the scheduling.
- `execute_cycle(self)`: this method contains the logic of executing a cycle in the simulation incrementing `current_cycle` by one, attempting to schedule instructions, and retiring instructions. The definition is as follows:

```
def execute_cycle(self):
    self.current_cycle += 1
    self.schedule()
    self._retire_instructions()
```

- `run(self)`: this method is fairly straightforward, it executes cycles until all instructions have been dispatched out of the scheduler. Most configurations used the definition:

```
def run(self):
    while self.instructions or self.instructions_in_progress:
        self.execute_cycle()
```

- `schedule(self)`: this method contains the logic for scheduling instructions. It is abstract, since each subclass configuration schedules instructions differently.
- `_schedule_instruction(self, instr : Instruction)`: this method is in charge of updating the status of an instruction whenever it gets scheduled. It does so in the following manner:

```
def _schedule_instruction(self, instr : Instruction):
    instr.issue_cycle = self.current_cycle
    instr.exp_completion = self.current_cycle + instr.latency()
    instr.started = True
    self.instructions_in_progress.append(instr)
```

- `_retire_instructions(self)`: this method should contains the logic for retiring instructions. It is abstract, since each subclass configuration retires instructions differently.

### 3 Methodology

### 4 Tests

### 5 Results

### 6 Discussion