Cache Simulator

BY

Bhavya Gutta (A04249172)

Sudeesh Naidu Yenugula (A0425357)

# Introduction

In this project, we implement Intermediate Cache Simulator explore the effectiveness of caches and the impact of different cache configurations. The rest of the report is organized as follow. Section [3](#_bookmark12) introduces designed test benchmarks to validate our design. All designed source code and demonstration are shown in the Appendices.

# Instructions

In this section, we give detailed instructions on how to execute our designed Cache Simulation implementation including 1) system requirement; 2) how to compile and generate the executable file 3) How to provide inputs and get output.

## **System Requirement**

In this section, the software platform for designing this project and execution steps is introduced.

### **2.1.1 Software Platform**

The supporting software for compiling and executing this project is listed as follows.

* + - * **OS:** Windows 7/10, Linux.
      * **Language:** Python 3.70.
      * **Execution environment:** PyCharm or others similar with python support.

## **System Design Overview**

Each memory address (64 bit) can be divided into 3 parts: tag, index and offset. The number of bits of offset is determined by cache block size and the number of bits of index is determined by the formulation: cache size/cache block size/associativity. While mapping from memory address to cache, it is required to configure the associativity. Associativity of 1 is direct-mapped cache, while associativity of block number is fully associative cache. Replacement policy is configurable within LRU, FIFO and Random.

## **Source Files and Execution**

In this section, we first briefly introduce the purposes of all source files and the relationship among them. Then, we provide detailed steps on how to generate and execute the executable file.

### **Source Files**

In this project, five source files are designed including “main.py”,”bin\_addr”, “cache.py”, “simulator.py”, “table.py”, and “word\_addr.py”. The purpose of each file is listed as follows:

* **Ma in.py:** This file implements the Consolidate simulator functions into class.
* **bin\_addr.py:** Replace redundant str instances with super () function. Retrieves the binary address of a certain length for a base-10-word address; we must define new instead of init because the class we are inheriting from (str) is an immutable data type.
* **CACHE**.**py**: [Move block replacement logic into separate function](https://github.com/caleb531/cache-simulator/commit/d34b82ab129b858b8c350f4b0de6a8ce5cb461e6). A list of recently ordered addresses, ordered from least-recently.
* **Table.py**: Displaying ASCII tables.
* **Word\_addr.py:** Retrieves all consecutive words for the given word address (including itself).
* **Simulator.py:** Final file that implements all the above python files and prints the final outputs.Ex: 'WordAddr', 'BinAddr', 'Tag', 'Index', 'Offset', 'Hit/Miss'.

### **Packaging and Execution Steps**

In this subsection, the steps to execute the designed source files are described. Since the source files are written in python, no compilation is necessary. Instead, to make the program executable, the source files and the supporting libraries are packaged into an executable file. The detailed packaging process and executing steps will be described in the final report.

## **Input and Output Files**

In this section, we first specify the format of input files and detailed procedure to read them. Then, we specify the format and location of the generated output file.

### **Input Files**

In this project, 12 input files are designed including two hardware configuration files: “processor config.txt,” “config Space.txt,” and 10 test benchmarks. First, two hardware configuration files and one exemplary test benchmark “test case 1.txt” are detailed as follows, then in Section [3](#_bookmark12) the ten benchmarks are explained in detail.

### **Output Files**

The output file generates the instruction timing table and registers values after execution. It is generated in the folder as the source code and is temporarily named “instruction final table.txt”.

# 3. Benchmarks

In this section, we are planning to set a benchmark for our project varying from easy to hard, all are designed by all group members. Based on the level of difficulty, all benchmarks would be categorized into 5 groups.

# 4.0 Design Progress

In this section, the design progress of two phases is specified

## **4.1 Phase 1 Progress**

In Phase I, the project is able to be compiled in the IDE (PyCharm). We will find out a proper way to generate the executable file of the project for demonstration once all implementation details are finalized. In addition, the input configuration in TXT files can be read by the simulated processor and it can correctly setup following the input configuration. Once the configuration process is done, the simulated processor can also read the instruction set in TXT file and generate the output TXT file containing the final instruction status table as well as the contents in registers and memory. Moreover, the simulated processor can successfully figure out the true dependency for all instructions dealing with integers and reflects the dependency on the final instruction status table.

To demonstrate the project more clearly, we made a video showing the main procedures on how to generate and input the configuration file and instruction set file, compile the whole project, and output the file “instruction final table.txt” containing the final instruction status table as well as non-zero contents in register and memory.

* 1. **Phase 2 Progress**

In Phase II, we incorporate Write back function design, CDB arbiter design, Function

Unit design, LSQ design, Write back optimization, Branch Prediction, and System Rollback. All of the test cases can work except the branch which now can partially work and requires further modification.