Tomasulo Simulator

Microprocessor Project Report

Team Number: 22

Team Members:

Name	**ID**	**Tutorial**
Seifeldin Khaled Soliman	55-18254	T-8
Marwan Mohamed Saeed Elsisi	55-1950	T-21
Abdelrahman Ahmed Mostafa	55-1707	T-24
Mohamed Ahmed Gamal	55-6377	T-22
Ahmed Yehia	55-3969	T-18
Ahmed Hesham	55-3590	T-24

1 Approach

To develop the Tomasulo algorithm simulator, we combined Python for the backend computation and React for the frontend interface, ensuring an intuitive and effective solution. Below is a detailed breakdown of the approach:

1.1 Backend: The Brains of the Simulator

The backend, implemented in Python, manages all the computational logic necessary to simulate the Tomasulo algorithm:

- **Core Logic:** It simulates essential components such as reservation stations, registers, caches, and execution units.
- **API Communication:** FastAPI was used to expose backend functionality via API endpoints, enabling seamless interaction with the frontend.
- **Step-by-Step Simulation:** The backend processes instructions (loaded from a file or entered manually) step by step, updating the state of the reservation stations, buffers, registers, and other components at every cycle.
- **Advanced Features:**
  - Handles operation latencies and simulates cache hits and misses for data memory.
  - Resolves data hazards, including RAW (Read After Write), WAR (Write After Read), and WAW (Write After Write).

1.2 Frontend: The Face of the Simulator

The frontend, developed using React, provides an interactive and visually intuitive interface. Key features include:

- **Input and Configuration:** Users can set latencies for operations, and customize cache and buffer sizes.
- **Dynamic Visualization:** The system state is updated cycle by cycle, displaying:
  - The status of reservation stations.
  - The contents of registers.
  - The cache and instruction queue.
- **Seamless Interaction:** The frontend fetches updates from the back-end using API calls, ensuring smooth and dynamic simulation.

1.3 Key Features and Highlights

- **Instruction Support:** The simulator handles a variety of MIPS instructions, including ALU operations, loads, stores, and branches.
- **Hazard Management:** It efficiently resolves RAW, WAR, and WAW hazards as well as structural dependencies.
- **Cache Simulation:** Users can customize cache settings, and the simulator reflects cache misses for data memory.
- **Conflict Resolution:** Scenarios where multiple instructions attempt to write to the Common Data Bus (CDB) are handled gracefully.

1.4 Testing and Validation

The simulator was rigorously tested to ensure accuracy under various scenarios:

- **Loops:** Testing included instruction loops to verify proper flow and execution.

- **Hazards:** Scenarios involving RAW, WAR, and WAW conflicts were tested to ensure robust hazard resolution.
- **Cache Variations:** Different cache configurations were used to validate cache hit and miss handling.

2 Code Structure

The codebase is structured as follows:

- **backend/**
  - **src/**
    - **classes/**
      - **Station/**
        - `__init__.py`: Initialization file for `Station` module.
        - `Station.py`: Base class for reservation stations.
        - `StationEntry.py`: Manages individual entries in reservation stations.
      - `Cache.py`: Handles data cache simulation, including hits and misses.
      - `CDB.py`: Implements the Common Data Bus logic.
      - `Instruction.py`: Defines MIPS instructions and their types.
      - `MemoryManager.py`: Simulates memory operations.
      - `Register.py`: Manages the register file.
      - `Simulator.py`: Core class orchestrating the simulation process.
    - **enums/**
      - `__init__.py`: Initialization file for the `enums` module.
      - `InstructionType.py`: Enumerates the supported instruction types.
      - `Opcode.py`: Defines opcodes for various MIPS instructions.
      - `StationState.py`: Enumerates the states of reservation stations.
    - **exceptions/**
      - `__init__.py`: Initialization file for the `exceptions` module.

      - `CDBException.py`: Handles conflicts related to the Common Data Bus.
      - `MemoryAccessException.py`: Manages memory-related errors.
    - **utils/**
      - `__init__.py`: Initialization file for the `utils` module.
      - `helpers.py`: Contains utility functions for the simulator.
  - `main.py`: Entry point for the backend API using FastAPI.

- **Configuration Files:**
  - `requirements.txt`: Lists Python dependencies.
  - `regs.txt`: Sample register values.
  - `instructions.txt`: Sample instruction file.

- **frontend/**
  - **src/**
    - **components/**
      - **Cache.tsx**: Displays the cache status during the simulation.
      - **ConfigPanel.jsx**: Component for configuring simulation settings (latency, cache, etc.).
      - **InstQueue.jsx**: Displays the instruction queue.
      - **InstructionsTable.tsx**: Shows the instruction statuses during the simulation.
      - **RegisterStatusTable.tsx**: Displays the current status of the register file.
      - **ReservationStationTable.tsx**: Component to visualize the reservation stations.
    - **hooks/**
      - **useConfig.js**: Custom hook to manage configuration state.

        - **useFunctions.js**: Hook to manage simulator-related functions.
        - **useStations.js**: Custom hook for managing reservation station data.
    - **utils/**
        - **helpers.js**: Utility functions for reusable operations.
    - **App.jsx**: Main application component that ties all other components together.
    - **main.jsx**: Entry point of the React application.

- **instructions.txt**: Sample instruction file for simulation.
- **regs.txt**: Sample register file for initialization.

3 Test Cases

The `instructions.txt` file is parsed line by line to identify and classify instructions into the following categories:

- **Load Instructions**: Instructions that load data from memory into registers.
- **Store Instructions**: Instructions that store data from registers into memory.
- **Arithmetic Instructions**: Instructions that perform arithmetic or logical operations.
- **Branch Instructions**: Conditional or unconditional instructions that control the program flow.

3.1 Cycle-by-Cycle Execution

Each instruction progresses through the following stages:

- **Issue**: The instruction is dispatched to the appropriate reservation station if available. If not, it waits in the instruction queue.
- **Execute**: The instruction begins execution once all operands are available. Dependencies are resolved dynamically using reservation stations and the Common Data Bus (CDB).
- **Write Back**: The result of the instruction is written back to the register file or memory, and it is broadcast on the CDB to make it available to other waiting instructions.

At each cycle, the simulator updates the following components:

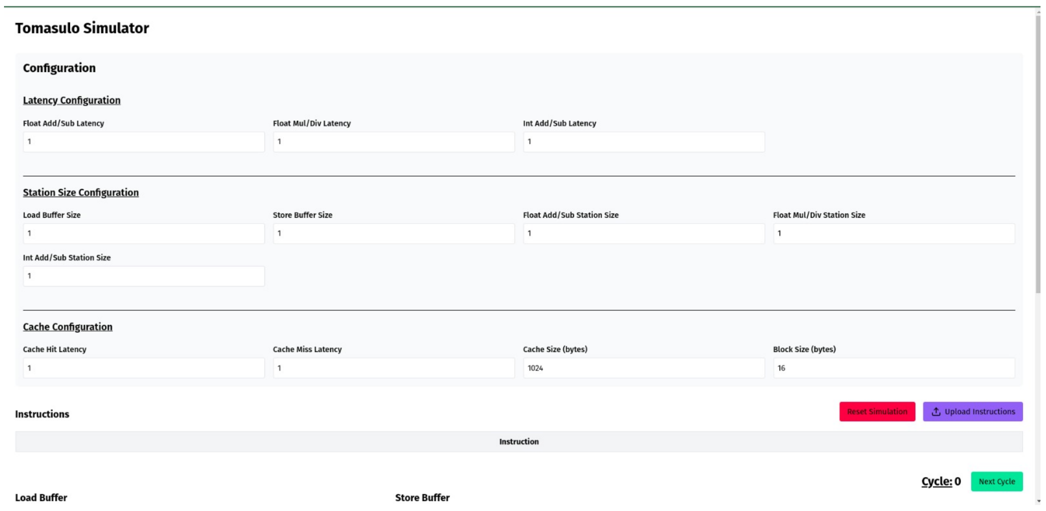
- The state of registers.

- Buffers (e.g., load and store buffers).
- Reservation stations.
- Memory content and status.

3.2 Visualization

The state of the system is visualized on the frontend, providing real-time feedback on the simulation. The visualization includes:

- **Reservation Station Statuses**: Displays the current state of reservation stations, including which instructions are waiting, executing, or completed.
- **Register File Contents**: Shows the values stored in each register.
- **Cache and Memory Statuses**: Indicates cache hits/misses and the content of memory.
- **Instruction Queue and Execution Progress**: Highlights the instructions in the queue and their progress through the pipeline.



Tomasulo Simulator

Configuration

Latency Configuration

Float Add/Sub Latency	1
Float Mul/Div Latency	1
Int Add/Sub Latency	1

Station Size Configuration

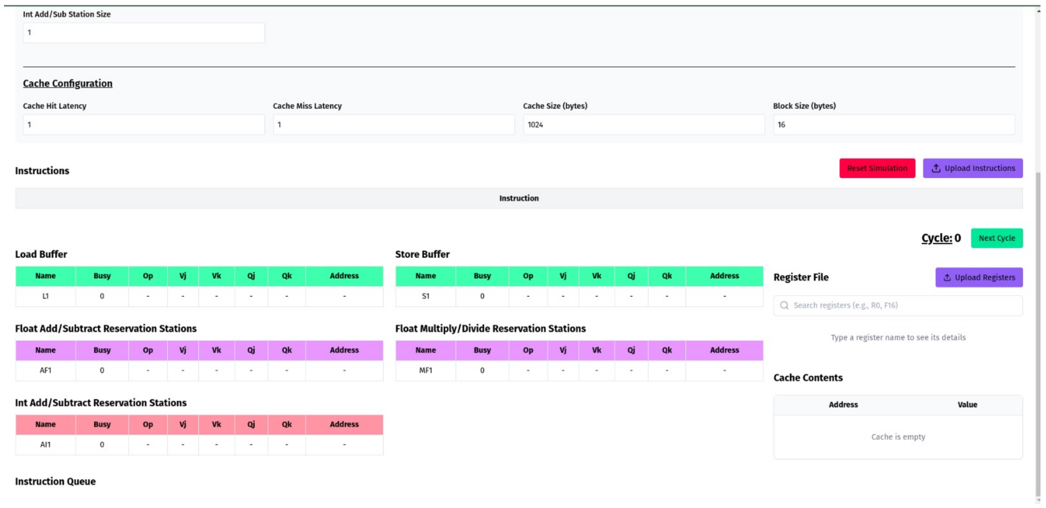
Load Buffer Size	1
Store Buffer Size	1
Float Add/Sub Station Size	1
Float Mul/Div Station Size	1
Int Add/Sub Station Size	1

Cache Configuration

Cache Hit Latency	1
Cache Miss Latency	1
Cache Size (bytes)	1024
Block Size (bytes)	16

Instructions

Figure 1: Input handlers.



Int Add/Sub Station Size	1

Cache Configuration

Cache Hit Latency	1
Cache Miss Latency	1
Cache Size (bytes)	1024
Block Size (bytes)	16

Instructions

Load Buffer

Name	Busy	Op	Vj	Vk	Qj	Qk	Address
L1	0	-	-	-	-	-	-

Store Buffer

Name	Busy	Op	Vj	Vk	Qj	Qk	Address
S1	0	-	-	-	-	-	-

Float Add/Subtract Reservation Stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Address
AF1	0	-	-	-	-	-	-

Float Multiply/Divide Reservation Stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Address
MF1	0	-	-	-	-	-	-

Int Add/Subtract Reservation Stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Address
AI1	0	-	-	-	-	-	-

Register File

Upload Registers
Search registers (e.g., R0, F16)
Type a register name to see its details

Cache Contents

Address	Value
Cache is empty	

Instruction Queue

Figure 2: Buffers , register file and reservation stations