Team 56

Zeyad AlaaEldeen Hassan Habash 52-16824
Shorok Abdulraof Aldosoky 52-20880
Mariam Mohamed Atef 52-7062

Introduction:

The goal of this project was to develop a simulator for the Tomasulo algorithm, a dynamic scheduling algorithm. The simulation accepts MIPS instructions in assembly format, executes them step by step, and visualizes the state of the reservation stations, register file, cache, and instruction queue at each cycle. The project was implemented in Java with minimal GUI.

Assumptions

- If two instructions wish to write in the same cycle , higher priority cycle is the one chosen to write on the bus
- If the two instructions have the same priority then it's FIFO
- Priority depends on how many times this instruction is needed in Register file and other reservation stations
- In BNEZ: we know the result at the end of the execution and fetch the next instruction accordingly in the upcoming cycle
    - In this example: BNEZ -> exec 7..7 write 8, We fetch the next instruction in 8 as well
- In S.D: we store the value in the cache in the write Result stage

Approach:

Tomasulo's algorithm is a dynamic scheduling algorithm used in computer architecture for out-of-order execution of instructions. It aims to improve instruction-level parallelism by allowing instructions to execute as soon as their operands are available. The key components include reservation stations

Code Structure:

The project is divided into several modules, each responsible for a specific part of the simulator. Here's a detailed overview of the modules:

1. **Parser Module (*CodeParser.java*):** This module is responsible for reading and parsing MIPS assembly instructions from a file. It extracts operation codes, source/destination registers, immediate values, etc. The *readFile* method reads the file and parses the instructions into *Instruction* objects.

2. **Data Structures Module**: This module includes classes that represent the various data structures used in the simulator, such as instructions, instruction queue, registers, and reservation stations.
    - *Instruction.java*: This class represents an instruction in the MIPS assembly language. It contains fields for the operation, destination register, source registers, and immediate value. Each type of instruction uses a different constructor which sets different fields. For example: L.D uses a constructor which sets destination register, 1 source register and an immediate value (for address), while FP instructions use a constructor which sets destination, 2 sources and no immediate value.
    - *InstructionQueue.java*: This class represents the instruction queue. It contains an *ArrayList* of *Instruction* objects and methods for adding instructions to the queue, getting an instruction from the queue, and printing the queue.
    - *Register.java*: This class represents a register. It contains fields for the label, Qi (the reservation station that will write to this register), and the value of the register. It also contains methods for getting and setting these fields.
    - *RegisterFile.java*: This class represents the register file. It contains an array of 64 *Register* objects, 32 for Integer registers and 32 for FP registers. and methods for initializing the register file, getting a register, and printing the register file.
    - *ReservationStationRow.java*: This class represents the rows of the reservation stations. It contains fields for the busy flag, tag of the row (A0,A1,M0,etc), operation, Vj, Vk, Qj, Qk, Address, priority, result of execution and a reference to the instruction currently in this row. It also contains methods for getting and setting these fields, printing the row and clearing a row
    - *ReservationStation.java*: This class represents the different reservation stations. It contains fields for the name, size, and an array of *ReservationStationRow* objects. It also contains methods for checking if the reservation station is full or empty, issuing an instruction, and printing the reservation station.

3. **Tomasulo Algorithm Module (*Tomasulo.java*)**: This is the core module of the simulator. It implements the logic of the Tomasulo algorithm, including the issue, execution, and write-back stages for each instruction type. It also handles hazards (RAW, WAR, WAW). It contains fields for the current cycle, program counter, cache (which is an array of doubles of size 1024), register file, reservation stations, instruction queue, and program instructions. It also contains methods for initializing the simulator, running the simulator, issuing instructions, executing instructions, writing results, and printing the state of the simulator.
    - Initialization: The Tomasulo class is a singleton, meaning only one instance of it can exist. It's initialized with parameters for the sizes of the reservation stations and buffers, and the latencies of different operations. The init() method initializes the cache, register file, reservation stations,

buffers, and instruction queue. It also parses the input file to get the program instructions.
- Running the Algorithm: The run() method in the Tomasulo class is the main driver of the algorithm. It runs in a loop until all instructions are executed. In each cycle, it performs the following steps:
- Issue: If there's no branch stall or reservation station stall, it gets the next instruction from the program and issues it. The issue() method checks if the reservation station for the instruction is full. If not, it issues the instruction to the reservation station and updates the Qi field of the destination register.
- Execute: The execute() method checks if any instruction is ready to execute in the reservation stations. If an instruction is ready, it executes the instruction and updates the execution start and end cycles.
- Write Result: The writeResult() method gets the row with the highest priority to write back. If a row is ready to write back, it updates the value of the destination register and clears the reservation station row.

4. **User Interface Module (*TomasuloGUI.java*)**: This module manages user input, simulation configuration, and command-line interaction. It includes a graphical user interface (GUI) that allows the user to input MIPS instructions and configure simulation parameters. It also includes a search functionality in the output area. It contains fields for various buttons, text fields, and the output area. It also contains methods for creating the GUI, adding fields to the GUI, and running the simulator when the "Run" button is clicked.

Testing:

Basic Operations: We tested the simulator with basic arithmetic and load/store instructions to ensure that the basic functionality was working correctly. This included testing with different operand values and checking the final results.
Examples:

- L.D R1 20
- S.D R0 30
- MUL.D F5 F2 F3
- DIV.D F6 F4 F1
- ADD.D F10 F11 F12
- SUB.D F13 F14 F15
- DADD R4 R5 R6
- DSUB R7 R8 R9
- SUBI R3 R3 8
- ADDI R4 R4 8

Instruction Sequences: We created test cases with sequences of instructions to validate the correct ordering and execution of instructions. This included testing with dependencies between instructions to ensure that hazards were handled correctly.
Example:

Same as the above but in one program

```
L.D R1 20
S.D R0 30
MUL.D F5 F2 F3
DIV.D F6 F4 F1
ADD.D F10 F11 F12
SUB.D F13 F14 F15
DADD R4 R5 R6
DSUB R7 R8 R9
SUBI R3 R3 8
ADDI R4 R4 8
```

Hazard Scenarios: We specifically designed test cases to trigger RAW, WAR, and WAW hazards. We verified that the simulator correctly stalled or reordered instructions as necessary to resolve these hazards.

Example:

```
L.D R1 20
DADD R4 R1 R6   → RAW
S.D R4 30 → RAW
DSUB R4 R8 R9 → WAR
ADDI R4 R4 8 → WAW
```

Branch Instructions: We tested the simulator's handling of branch instructions, including the correct updating of the program counter and stalling till branch result is available.

Example:

```
L.D F0 20
LOOP DADD R5 R2 R2
S.D F5 30
SUBI R3 R3 8
BNEZ R3 LOOP
L.D F1 30
LOOP2 MUL.D F2 F1 F0
S.D F2 40
L.D F3 60
SUBI R4 R4 8
BNEZ R4 LOOP2
```

Example from lecture 13 (with result)**:**

```
LOOP L.D F0 20
MUL.D F4 F0 F2
S.D F4 30
SUBI R1 R1 8
BNEZ R1 LOOP
```

RS Sizes and instruction latencies:

Sizes

Add Station Size:	3
Mul Station Size:	2
Load Buffer Size:	3
Store Buffer Size:	3

Latencies

Add Latency:	2
Sub Latency:	4
Mul Latency:	4
Div Latency:	20
Load Latency:	2
Store Latency:	5
SUBI Latency:	1
DADD Latency:	1
DSUB Latency:	1

Final Cycle:

Cycle:15

Stalling: false

Reservation Stations:

Add Reservation Station:

Tag	Busy	Op	Vj	Vk	Qj	Qk	A	Result	Use Count
A0	false	-	-	-	-	-	0	-	0
A1	false	-	-	-	-	-	0	-	0
A2	false	-	-	-	-	-	0	-	0

Mul Reservation Station:

Tag	Busy	Op	Vj	Vk	Qj	Qk	A	Result	Use Count
M0	false	-	-	-	-	-	0	-	0
M1	false	-	-	-	-	-	0	-	0

Load Buffer:

Tag	Busy	Address	Result	Use Count
L0	false	0	-	0
L1	false	0	-	0
L2	false	0	-	0

Store Buffer:

Tag	Busy	V	Q	A	Result	Use Count
S0	false	-	-	0	-	0
S1	false	-	-	0	-	0
S2	false	-	-	0	-	0

Instruction Queue:

Op	Dest	j	k	Issue	Exec	WriteRes
L.D	F0	0	20.0	1	2...3	4
MUL.D	F4	F0	F2	2	5...8	9
S.D	F4	0	30.0	3	10...14	15
SUBI	R1	R1	8.0	4	5...5	6
BNEZ	0.0	0	R1	5	7...7	8

Register File:

Register	Qi	Value
R0	0	0.0
R1	0	0.0
R2	0	16.0
F0	0	14.58
F1	0	82.5
F2	0	85.0
F3	0	87.5
F4	0	1239.3

Cache:

| | [30]: 1239.3 | |