COL216
ASSIGNMENT 4

PRAKHAR AGGARWAL (2019CS50441)
DHAIRYA GUPTA (2019CS50428)

Problem Statement: Memory Request Ordering

A basic MIPS interpreter handling a subset of the ISA was built in Assignment-3 and was improved in Minor Exam. Objective for this Assignment is to implement a strategy for efficient ordering of DRAM requests at runtime.

DESIGN DECISIONS:

Steps such as tokenizing, parsing the input file and storing them in “instruction” struct remain same as in Minor Exam. Some changes have been made to improve the readability of code. Way to execute instructions has changed a bit, output format has been changed, new errors have been taken into account.

Input/Output:

User is prompted to give input on the console. Input format is as follows:

1.) We need to compile the cpp code using command:
g++ -o code code.cpp

2.) To run the executable file, use the command:
./code <TestcaseFile> <ROW_ACCESS_DELAY> <COL_ACCESS_DELAY> > <OutputFile>

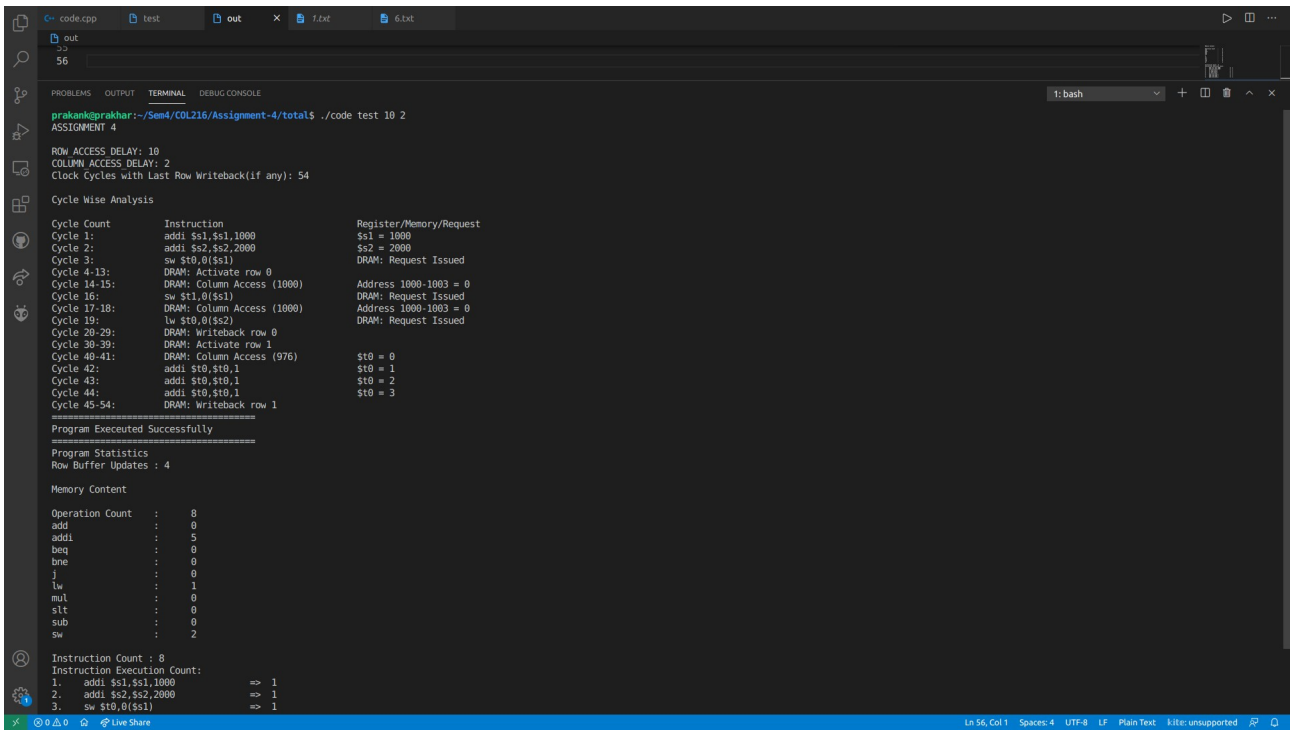
In this, TestcaseFile is a required parameter whereas ROW_ACCESS_DELAY, COL_ACCESS_DELAY and OutputFile are optional parameters. By default, values of RowDelay and ColDelay are taken 10 and 2 respectively unless specified via the console.

Sample I/O is shown below:

```
prakank@prakhar:~/Sem4/COL216/Assignment-4/total$ ./code test 10 2 > out
prakank@prakhar:~/Sem4/COL216/Assignment-4/total$ ./code test > out
prakank@prakhar:~/Sem4/COL216/Assignment-4/total$
```

Both are acceptable ways of providing input.

If output file is not provided on console, then output is printed on console itself.



Implementation:

Instructions are read sequentially. Different decisions are taken on the basis of instruction to be executed or to be stored in the queue. The implementation tries to mimic roughly what MIPS simulator does in reality. The execution output is stored in an array and is printed at the end of execution if there are no errors. All the relevant statistics are printed as well after the execution.

Variable named DesignDecision can be set to True which will writeback to memory only if there is some modification in RowBuffer.

I have implemented all the instructions mentioned in Assignment-3: add, sub, mul, beq, bne, slt, j, lw, sw, addi.

Following points are taken into care while implementing:

1. Instructions execute "SEQUENTIALLY".
2. ONLY ONE instruction can be processed at a time in a processor.
3. ONLY ONE instruction request can be processed at a time in a DRAM.
4. Final value to be stored in the memory does not change with DRAM implementation

For example, I have a processor connected to DRAM:

1. If DRAM is idle, lw/sw can execute and request can be sent to DRAM
2. If DRAM is busy, no new lw/sw is safe to be processed

3. If DRAM is busy, then processor waits for the response from DRAM for the prev lw/sw

Implementing DRAM without Non-Blocking Memory Access feature.

- All the instructions are read sequentially.
- The program reads an instruction and checks whether it is an lw/sw instruction. If yes, then program checks the Row in Buffer. If there is a row in buffer, program checks whether the row of current instruction matches with the one in Buffer.
- If same row is in buffer, then program checks whether the registers are safe to use or not. Program also checks for Memory Address, if both registers and memory address are safe to use, program executes the lw/sw instruction. In all other cases, program pushes the current lw/sw instruction to a queue.
- Each instruction in queue contains a dependency list which gives the index of instruction in queue on which the current instruction depends. If current instruction is independent of any prior lw/sw instruction, (-1,-1) is stored.
- If the instruction is not an lw/sw instruction, program checks whether it is safe to use the registers or not i.e. whether the registers are dependent on some lw/sw instruction in queue. If the registers involved are independent, program executes the current instruction. If not, then it will continue to execute the lw/sw instructions lying in queue to resolve the dependency using recursion.
- Maps such as RegisterUpdation, RegisterUse, MemoryUpdation, MemoryUse are used to do the needful and a queue is implemented to keep a track of instructions to be executed in future.
- Also, once an instruction in a queue is executed, it is not deleted but given a value (-2,-2) indicating it has been executed.
- RegisterUpdation is a map which stores the index of latest instruction in queue which will update the value of the register.
- RegisterUse is a map which stores the index of latest instruction in queue which will use the value of the register. Similarly, MemoryUpdation and MemoryUse stores the memory address.
- This approach helps to reduce the number of clock cycles by optimizing the DRAM usage and reducing the number of DRAM requests

Implementing DRAM with Non-Blocking Memory Access feature.

- Difference in this part is that instructions are not blocked (similar to the one in Minor Exam).

- While an sw/lw instruction is under execution, I can let the sw/lw instruction to run in background and execute the instructions such as add, addi etc.

Strengths and Weaknesses:

Strength of this approach is that it significantly reduces the clock cycles by reordering the DRAM request. In Non-Blocking part, number of clock cycles required to execute the program are less than those required to execute the program without Non-Blocking feature.
In testcase 1, cycles required with Minor Exam implementation -> 180
cycles required with Assignment 4 implementation -> 70

Difference in clock cycles depicts the efficiency of this approach.

Weakness of this approach is the memory consumption. A separate queue is maintained and 2 extra maps are maintained to execute the process which leads to extra memory usage. So, in an example like,

```
sw $t0, 1000($zero)
addi $t0, $zero, 12
sw $t0, 1004($zero)
addi $t0, $zero, 12
```

Though number of cycles used will be same (according to Minor and Assignment 4), but memory consumption in Assignment 4 is way more than that in Minor. Hence, the cost of hardware required to implement this approach is more than that required to implement Minor Exam approach.

Also, in a testcase like testcase 1,

input

```
1   main:
2       addi $s0, $zero, 1000
3       addi $s1, $zero, 2500
4       addi $t0, $zero, 1
5       addi $t1, $zero, 2
6       addi $t2, $zero, 3
7       addi $t3, $zero, 4
8
9       sw $t0, 0($s0)
10      sw $t1, 0($s1)
11
12      sw $t2, 4($s0)
13      sw $t4, 4($s1)
14
15      lw $t5, 0($s0)
16      lw $t6, 0($s1)
17      lw $t7, 4($s0)
18      lw $t8, 4($s1)
19
20  exit:
```

My program will execute the first lw/sw instruction in line 13 rather than executing the one in line 9. This can act as a weakness as well as a strength depending on the type of testcase.

Testcases

The implementation has been thoroughly tested with multiple test cases and 8 of the testcases and their respective outputs for both the parts of the assignment have been attached with this document and the supporting C++ file.