

COL216

MINOR 2021 ASSIGNMENT

PRAKHAR AGGARWAL (2019CS50441)

Problem Statement: MIPS simulator with DRAM timing model

A basic MIPS interpreter handling a subset of the ISA was built in Assignment-3. Objective for this Assignment is to enhance the MIPS interpreter with two new features making it a MIPS simulator.

- Develop a model for the Main Memory and integrate into the basic interpreter.
- The memory access should be non-blocking (subsequent instructions don't always wait for the previous instructions to complete).

DESIGN DECISIONS:

Steps such as tokenizing, parsing the input file and storing them in “instruction” struct remain same as in Assignment-3. 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 2019CS50441 2019CS50441.cpp
- 2.) To run the executable file, use the command:
**./2019CS50441 <TestcaseFile> <PartNumber> <ROW_ACCESS_DELAY>
<COL_ACCESS_DELAY> > <OutputFile>**

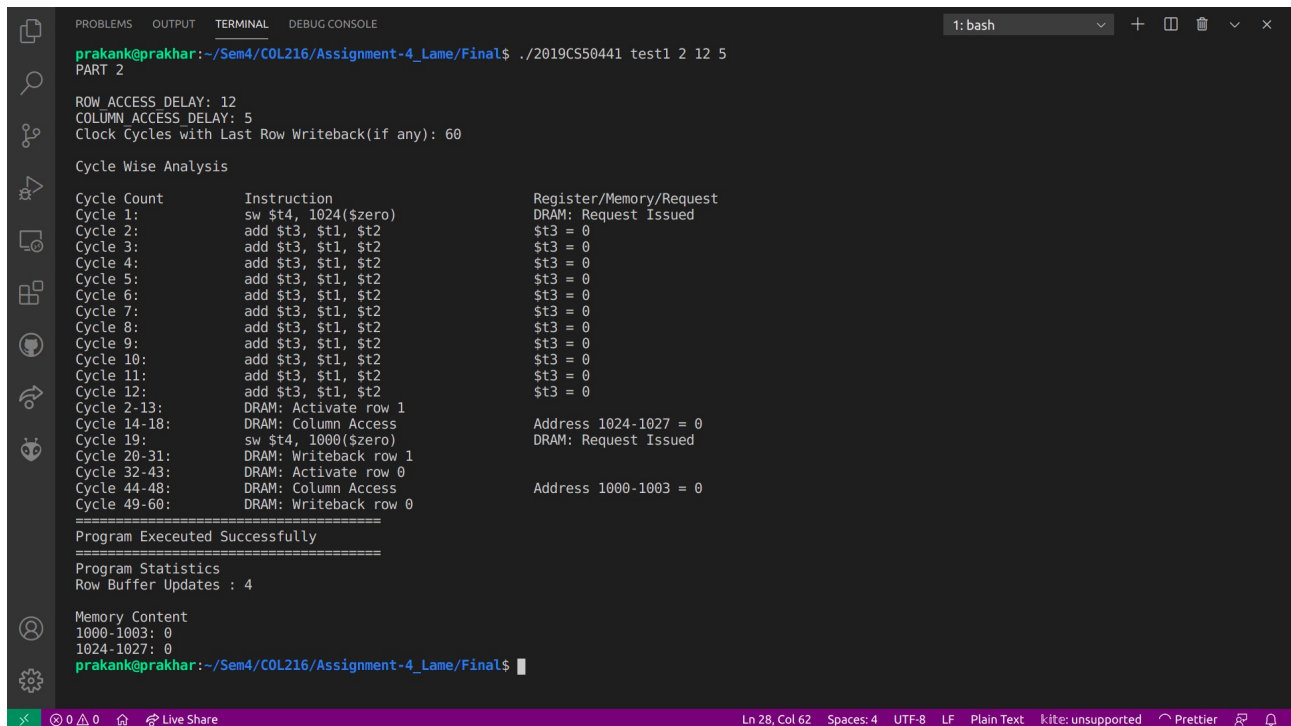
In this, TestcaseFile, PartNumber are the required parameters whereas ROW_ACCESS_DELAY, COL_ACCESS_DELAY and OutputFile are optional parameters. By default, values of RowDelay and ColDelay are taken 10 and 2 respectively unless specified via the console.

Sample I/O is shown below:

```
prakank@prakhar:~/Sem4/COL216/Assignment-4_Lame/Final$ ./2019CS50441 test1 2 12 5 > out1
prakank@prakhar:~/Sem4/COL216/Assignment-4_Lame/Final$ ./2019CS50441 test1 2 > out1
prakank@prakhar:~/Sem4/COL216/Assignment-4_Lame/Final$
```

Both are acceptable ways of providing input.

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



```
prakank@prakhar:~/Sem4/COL216/Assignment-4_Lame/Final$ ./2019CS50441 test1 2 12 5
PART 2
ROW ACCESS DELAY: 12
COLUMN ACCESS DELAY: 5
Clock Cycles with Last Row Writeback(if any): 60

Cycle Wise Analysis

Cycle Count      Instruction      Register/Memory/Request
Cycle 1:         sw $t4, 1024($zero)    DRAM: Request Issued
Cycle 2:         add $t3, $t1, $t2      $t3 = 0
Cycle 3:         add $t3, $t1, $t2      $t3 = 0
Cycle 4:         add $t3, $t1, $t2      $t3 = 0
Cycle 5:         add $t3, $t1, $t2      $t3 = 0
Cycle 6:         add $t3, $t1, $t2      $t3 = 0
Cycle 7:         add $t3, $t1, $t2      $t3 = 0
Cycle 8:         add $t3, $t1, $t2      $t3 = 0
Cycle 9:         add $t3, $t1, $t2      $t3 = 0
Cycle 10:        add $t3, $t1, $t2      $t3 = 0
Cycle 11:        add $t3, $t1, $t2      $t3 = 0
Cycle 12:        add $t3, $t1, $t2      $t3 = 0
Cycle 13:        DRAM: Activate row 1
Cycle 14-18:     DRAM: Column Access    Address 1024-1027 = 0
Cycle 19:        sw $t4, 1000($zero)    DRAM: Request Issued
Cycle 20-31:     DRAM: Writeback row 1
Cycle 32-43:     DRAM: Activate row 0
Cycle 44-48:     DRAM: Column Access    Address 1000-1003 = 0
Cycle 49-60:     DRAM: Writeback row 0

=====
Program Executed Successfully
=====
Program Statistics
Row Buffer Updates : 4

Memory Content
1000-1003: 0
1024-1027: 0
prakank@prakhar:~/Sem4/COL216/Assignment-4_Lame/Final$
```

Implementation:

Instructions are read sequentially. Different decisions are taken on the basis of instruction to be executed. 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.

I have implemented add, addi, lw, sw instructions as mentioned in the Assignment.

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

I have followed Design Choice 1 as mentioned on Piazza :

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

PART A: Implementing DRAM without Non-Blocking Memory Access feature.

- All the instructions are executed sequentially, i.e. the program executes the instructions in the order of their occurrence in the input File.
- In this case, the program doesn't jump over to the next instruction even if the next instruction is independent of current one while the lw/sw instruction is under process i.e. the program waits for the current instruction to execute completely without looking at the next instruction.
- This has been implemented by keeping a check on DRAM and checking if DRAM request is under process or not. If yes, then we wait for the clock cycles required to complete the process.

PART B: Implementing DRAM with Non-Blocking Memory Access feature.

- In this case too, instructions are executed sequentially, i.e. the program executes the instructions in the order of their occurrence in the input File.
- This case allows jumping to next instructions (if possible) while running previous lw/sw instruction.
- While the lw/sw instruction is running, program looks over the next instruction and if that instruction is add or addi and is independent of the registers involved in lw/sw running operation, the program executes that instruction.
- Program continues to do the procedure mentioned in above point until it sees an instruction dependent on result of lw/sw operation under execution or a new lw/sw operation itself.
- When the first instruction dependent on the running lw/sw operation is encountered, program waits for the lw/sw operation to complete and stops at that point.
- This has been implemented by keeping track of whether a lw/sw operation is under process or not. A map<string,int> named "Active" has been used which

keeps a track of upto what time are the registers involved busy. An integer named NumberOfCycles is used to keep a track of current cycle number and compare it with the busyTime of Registers.

- The program jumps to next instruction and checks whether the busy time of registers of this operation is less or equal to the NumberOfCycles. If yes, then this implies these registers can be safely used to execute the current instruction (without relying on the result of ongoing lw/sw operation) and does so. If not, then the program waits for the ongoing lw/sw operation to complete.
- This approach helps to reduce the number of clock cycles which are independent of the ongoing operation but still waiting for it to complete.

Strengths and Weaknesses:

Strength of Program is majorly in Part B which helps to reduce the number of clock cycles by jumping to next instructions and executing them.

One major weakness of this approach is that the program waits if next instruction is lw/sw even if it is independent of the ongoing one. Also, the program doesn't jump over an lw, sw operation hence failing to execute lw/sw operations (if any) with same row (if possible).

Eg.

```
addi $s0, $zero, 1000
addi $s1, $zero, 1024
lw $t0, 0($s0)
lw $t1, 0($s1)
lw $t0, 0($s0)
```

In this case, after reading instruction lw \$t0, 0(\$s0), my program waits whereas if we could have stored more rows in Row buffer, then this would have helped to reduce clock cycles.

Also, in case with only 1 row in row buffer, after completing instruction lw \$t0, 0(\$s0), my program will writeback row 0 to memory, Activate row 1, then will implement lw \$t1, 0(\$s1), then writeback row 1 to memory, Activate row 0. Whereas, better way would have been to execute instructions in the order:

```
lw $t0, 0($s0)
lw $t1, 0($s1)
lw $t1, 0($s1)
```

which would have helped to avoid unnecessary Activation and writeback to memory.

This approach could have been implemented using a queue and providing each instruction an end time and each register an end time and do the required updates on reading new lw/sw instruction.

Testcases

The implementation has been thoroughly tested with multiple test cases and 5 of the test cases and their respective outputs for both the parts of the assignment have been attached with this document and the supporting C++ file.