COL216 Assignment-4

Dhruv Kumar Gupta 2019CS10346
Swaraj Gawande 2019CS10406

April 2021

1 Design Overview

1.1 Input and Output

The Program accepts a command line input for the file name and path which contains the MIPS program and the access delays in the order. :

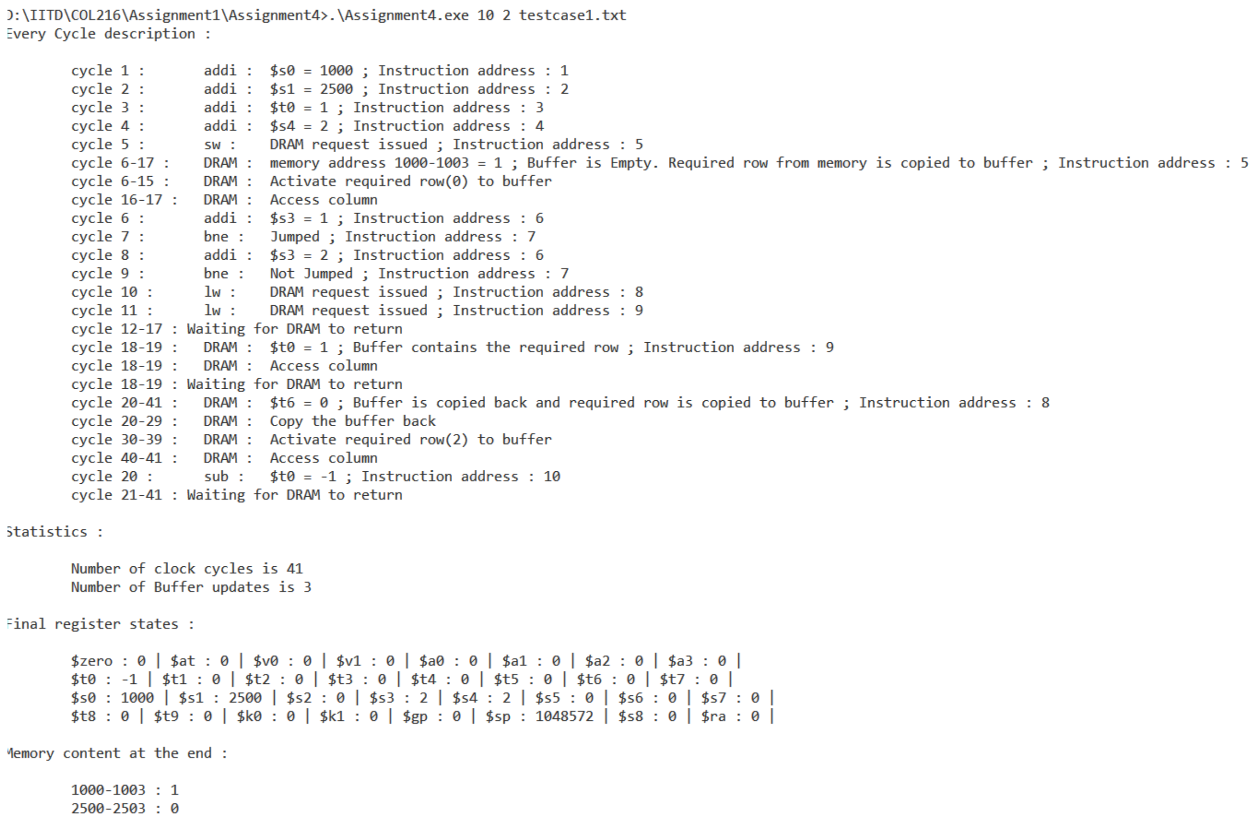
Row_access_delay , Column_Access_delay , filename

with the default values being 2, 10 and "code.txt".

Program then loads the instructions in the file, raises syntax errors if any, and if no syntax error is present, executes the instructions.

After each execution, Summary of the clock cycle(s) involved is printed. At the end of execution, state of registers and statistics like total cycle count, number of updates of row buffer and the final contents of memory which are used and the register file is printed.

The same clock number can appear twice, with the first being for the work done by DRAM and the other instance being the work done by the CPU. This shows that DRAM and CPU are working in parallel.
Example screen :



```
):\IITD\COL216\Assignment1\Assignment4>.\Assignment4.exe 10 2 testcase1.txt
Every Cycle description :

        cycle 1 :       addi :  $s0 = 1000 ; Instruction address : 1
        cycle 2 :       addi :  $s1 = 2500 ; Instruction address : 2
        cycle 3 :       addi :  $t0 = 1 ; Instruction address : 3
        cycle 4 :       addi :  $s4 = 2 ; Instruction address : 4
        cycle 5 :       sw :    DRAM request issued ; Instruction address : 5
        cycle 6-17 :    DRAM :  memory address 1000-1003 = 1 ; Buffer is Empty. Required row from memory is copied to buffer ; Instruction address : 5
        cycle 6-15 :    DRAM :  Activate required row(0) to buffer
        cycle 16-17 :   DRAM :  Access column
        cycle 6 :       addi :  $s3 = 1 ; Instruction address : 6
        cycle 7 :       bne :   Jumped ; Instruction address : 7
        cycle 8 :       addi :  $s3 = 2 ; Instruction address : 6
        cycle 9 :       bne :   Not Jumped ; Instruction address : 7
        cycle 10 :      lw :    DRAM request issued ; Instruction address : 8
        cycle 11 :      lw :    DRAM request issued ; Instruction address : 9
        cycle 12-17 : Waiting for DRAM to return
        cycle 18-19 :   DRAM :  $t0 = 1 ; Buffer contains the required row ; Instruction address : 9
        cycle 18-19 :   DRAM :  Access column
        cycle 18-19 : Waiting for DRAM to return
        cycle 20-41 :   DRAM :  $t6 = 0 ; Buffer is copied back and required row is copied to buffer ; Instruction address : 8
        cycle 20-29 :   DRAM :  Copy the buffer back
        cycle 30-39 :   DRAM :  Activate required row(2) to buffer
        cycle 40-41 :   DRAM :  Access column
        cycle 20 :      sub :   $t0 = -1 ; Instruction address : 10
        cycle 21-41 : Waiting for DRAM to return

Statistics :

        Number of clock cycles is 41
        Number of Buffer updates is 3

Final register states :

        $zero : 0 | $at : 0 | $v0 : 0 | $v1 : 0 | $a0 : 0 | $a1 : 0 | $a2 : 0 | $a3 : 0 |
        $t0 : -1 | $t1 : 0 | $t2 : 0 | $t3 : 0 | $t4 : 0 | $t5 : 0 | $t6 : 0 | $t7 : 0 |
        $s0 : 1000 | $s1 : 2500 | $s2 : 0 | $s3 : 2 | $s4 : 2 | $s5 : 0 | $s6 : 0 | $s7 : 0 |
        $t8 : 0 | $t9 : 0 | $k0 : 0 | $k1 : 0 | $gp : 0 | $sp : 1048572 | $s8 : 0 | $ra : 0 |

Memory content at the end :

        1000-1003 : 1
        2500-2503 : 0
```

1.2 Error Handling

Since, it is an extension of Assignment-3 and Minor, error handling has been inherited from the assignments and no additional error handling is required as any valid input as in Assignment-3 is also a valid input for this assignment.

2 Explanation of approach and code

The modular structure of Assignment-3 and the Minor, helped in extending the features of the simulator.

Memory class is the same as Minor where the Memory array is a 2D array of words[1] size 256×1024. This is so, because the Memory is word addressable, and each word uses 4 bytes.

The Memory now returns the number of cycles it takes for the access and writing of data, which is used by the CPU component to decide to run further instructions during that time or not.

The Memory now, also supports returning the number of buffer updates. We have considered the buffer updates, as any change in rows buffer. This number, thus indicates how many times, the DRAM had to copy a whole row to the buffer and how many times any single value in the buffer was changed i.e number of **sw** instructions executed.

The **run()** function, executes each instruction, and does the computation with the help of Auxiliary functions which use the returned number of cycles by memory to print details of the cycle(s) involved, specifying the instructions being executed and their result along with the buffer handling of memory. The details indicate whether the description is of process inside DRAM or CPU, where the CPU instructions are preceded by the corresponding instruction. Each DRAM clock description is followed by subparts indicating how many clocks each step inside the DRAM takes.

The simulator maintains a list[2] structure of memory instructions, which is technically existing in DRAM, but has been implemented in Simulator for easy access of registers and summary printing functions. Every safe memory instruction when encountered by CPU is added to the end of the list, and when DRAM is idle, a best choice instruction is chosen from the list.
To implement the feature of non blocking access, the simulator runs further instructions, as long as they are safe to run. If an unsafe instruction is encountered, the simulator waits for the DRAM to return and then continues as usual.

At the end, the statistics are printed, like number of cycles and number of buffer updates and the data inside used memory words and the registers.

2.1 Safe non memory instructions

Since there can be more than one Memory Instruction in List, the encountered non-memory instructions, are compared with each of the element of the list.

2.1.1 sw instruction in list

To save a word, DRAM gets the new value and the address of the word in the form of register and an immediate address or address stored in register with offset.

Thus, when invoking the DRAM, both the inputs are passed to the DRAM, in the form of a word and an address, and thus the involved registers can be used for read and write instructions by the cpu as the original values have been passed to the DRAM.

This implies that **every** non memory instruction is safe while list contains sw instruction.

2.1.2 lw instruction in list

To load a word, the destination register is filled with the word at the given address at the end of the execution by DRAM.

Since the register is updated, it should be used neither for reading nor writing, as any instruction after the load word instruction would either be intended to use the loaded word or replace the loaded word. But, since the address is passed to the DRAM in the starting, the register storing the address can be modified or read without deviating from intended use, as the address has been already passed.

Thus any instruction involving the destination register of **lw** instruction either as destination or source would be dependent and hence is unsafe and should not be executed.

[1]each memory location is of type int in C++
[2]The list is used instead of Queue, as Queue doesn't support reordering or deletion from places other than the front.

2.2 Algorithm and analysis

The algorithm besides the general approaches above, is determined by 2 processes :

- Safety of memory instructions
- choosing instruction to execute from the list

There are two optimisations done by us stepwise, both of them being based on the general principle of FR-FC-FS, which means

- **Ready First** : Executing the instructions which are ready, which in our case are the ones in the list.
- **Column First** : Prioritising the ones which require only column access, that is, belong to the same row as buffer.
- **Scheduled First** : Prioritising the ones which are older and earlier in the list.

2.2.1 First Approach

We created a preliminary design with same safety restriction as non-memory instructions. This choice led to all the instructions in list being completely independent at any given point, which allowed any permutation order of the list to be executed. Then, the first row-hit instruction from head to end in the list is executed whenever Memory is idle. If such an instruction is not present, the first instruction is executed.
This led to large decrease in number of clock cycles, and worked properly. There are some drawbacks present, as memory instructions using same registers are not pushed into list and thus, further instructions can not be executed.

This leads to the improved approach.

2.2.2 Improved Approach

Instead of waiting when same register memory instructions are encountered, we pushed it into the list. Hence, no dependent criteria for memory instructions. This allowed the program to explore further R-instructions.

This however, led to complications when choosing the instruction to execute from the list. To ensure, that in worst case, it performs at-least as good as first approach, the first row-hit is found, and its register is compared with previous elements to ensure independence. If no such element exists, this leads to better throughput, but if such element exists, choosing the first element of list gives equal results as compared with first approach, as the first approach would have done the same.
This leads to better results than first approach in almost all of the cases and equal clock cycles as first approach in other cases.

2.2.3 Drawbacks

Some possible drawbacks of our approach is actually the queue would be a part of memory and may have its delays while implementing at architecture level. We also perform many checks to decide which DRAM instruction to apply next. The final approach however does well in reducing number of clock cycles and increasing memory throughput.

3 Testing

Since testing of parser and execution of instructions had been performed during assignment 3, the main focus of the testing was on the features of DRAM instruction and non-blocking memory.

We first tested that the changed implementation of memory instructions has not introduced any bugs in the execution of other operations.

The printed statements were changed according to the specifications of the given test cases, and the number of cycles were compared with execution in case of Assignment-3 and the minor there was a clear reduction in number of clock cycles required.

As the final values of altered memory locations and the final state of registers is also printed and then tested so that the change in order of execution of memory instruction does not change the actual execution which was intended. Cases like negative delays and zero delays are handled before the execution as they are superfluous inputs.

After ensuring that correct delays and buffer updates are being done as expected. This was tested more extensively using cases like:

- Consecutive memory access instructions
- No independent instruction
- number of independent instruction greater than or equal to clock cycles required by DRAM.
- dependency of instruction in both destination register and source registers individually.
- memory instructions having same memory address or same destination registers.
- unsafe instructions.

3.1 Some testcases

Some over all testcases are given below while others are submitted with this assignment.

1 checking order of execution.

```
main:
addi $s0, $zero, 1000
addi $s1, $zero, 2500
addi $t0, $zero, 1
addi $t1, $zero, 2
addi $t2, $zero, 3
addi $t3, $zero, 4
addi $s3, $zero, 0
addi $s4, $zero, 4

sw $t0, 0($s0) #store 1 at location 1000
sw $t1, 0($s1) #store 2 at location 2500
sw $t2, 4($s0) #store 3 at location 1004
sw $t4, 4($s1) #store 4 at location 2504

lw $t5, 0($s0) #1000
lw $t6, 0($s1) #2500
j: addi $s3,$s3,1
bne $s3,$s4,j

lw $t7, 4($s0) #1004
lw $t8, 4($s1) #2504

exit:
```

2 Instructions with same destination registers.

```
main:
addi $s0, $zero, 1000
addi $s1, $zero, 2500
addi $t0, $zero, 1
addi $t1, $zero, 2
addi $t2, $zero, 3
addi $t3, $zero, 4
addi $s3, $zero, 0
addi $s4, $zero, 4

sw $t0, 0($s0) #store 1 at location 1000
lw $t0, 0($s0) #1000

sw $t1, 0($s1) #store 2 at location 2500
sw $t4, 4($s1) #store 4 at location 2504
lw $t6, 0($s1) #2500
lw $t4, 4($s0) #1004
sw $t0, 1024($zero) #lw on $t0 is to be finished first
lw $t8, 4($s1) #2504

exit:
```

3 waiting for unsafe instruction.

```
main:
addi $s0, $zero, 1000
addi $s1, $zero, 2500
addi $t0, $zero, 1
addi $t1, $zero, 2
addi $t2, $zero, 3
addi $t3, $zero, 4
addi $s3, $zero, 0
addi $s4, $zero, 20

sw $t0, 0($s0) #store 1 at location 1000
lw $t0, 0($s1) #2500
lw $t0, 0($s0) #1000
sw $t0, 0($s1) #store 2 at location 2500
j: addi $s3,$s3,1
bne $s3,$s4,j
sw $t4, 4($s1)
#store 4 at location 2504
add $t0,$t0, $t1
sw $t2, 4($s0) #store 3 at location 1004

lw $t5, 0($s0) #1000
addi $s3, $zero, 0
ja: addi $t2,$t2,1
bne $t2,$s4,ja
lw $t7, 4($s0) #1004

exit:
```

4 small for easier demonstration.

```
main:
addi $s0, $zero, 1000
addi $s1, $zero, 2500
addi $t0, $zero, 1
addi $s4, $zero, 2

sw $t0, 0($s0) #store 1 at location 1000
j: addi $s3,$s3,1
bne $s3,$s4,j
lw $t6, 0($s1) #2500
lw $t0, 0($s0) #1000
sub $t0,$zero,$t0

exit:
```