COL216

MINOR EXAM

Hardeep Kaur

2019CS10354

PART 1 : MODEL FOR THE MAIN MEMORY

1. Memory Address Format: Accepts address as a constant

2. Memory is word addressable.

3. DRAM is implemented as a 256 x 256 matrix.

4. Instructions are stored in Instruction memory after reading from a file “input.txt”.

5. This instruction memory is not a part of DRAM, since I had implemented main memory as having two units : data memory and instruction memory in assignment 3.

6. Instruction fetch from instruction memory has zero delay.

7. Execution of all the instructions except lw, sw takes one cycle.

8. For lw, sw instructions, the DRAM request is sent in first cycle, then activation/ writeback of a row/ column request is made in further cycles depending upon the instruction and active row in the row buffer.

9. Row buffer is implemented as a vector into which the entire row from DRAM is copied while activation.

10. Row buffer updates include activation of a new row into the buffer and updation of value in the buffer row.

11. Values are written to DRAM only during writeback operation, hence only at that time, the updated values in DRAM, if any, are printed to “dataMem_final.txt”, with the corersponding cycle number.

12. The final writeback is performed only if there is some updated value in the row buffer. This writeback needs additional number of cycles (= row_access_delay).

13. Whenever a register’s value is changed during execution, it is printed to “RF_final.txt”, alongwith the corresponding cycle number.

14. Only non zero values and their memory addresses are printed in “dataMemFinal.txt”.

PART 2 : NON BLOCKING MEMORY ACCESS

1. Instructions execute sequentially.

2. Only one instruction is processed at a time in the processor.

3. Only one instruction request is processed at a time in DRAM.

DESIGN :

1. An lw or sw instruction needs access to DRAM. So it requests for the access in the first cycle and the value is obtained or stored in further cycles depending on row access and column access delays.

2. Meanwhile, if next instruction doesn't need a DRAM access (i.e. add, addi, sub, mul, slt) and is independent of the instruction already being executed in DRAM, then it can be executed in the next cycle itself, without waiting for the lw/sw instruction to complete.

3. If the next instruction is jump instruction, (i.e. beq, bne, j) or DRAM instruction (i.e. lw, sw) , it would wait for the DRAM instruction to be completed.

4. If DRAM is idle, any one of the instructions (add, sub, mul, addi, beq, bne, slt, j) can execute.

APPROACH :

Q. WHEN IS THE NEXT INSTRUCTION SAFE TO EXECUTE?

Let to_Reg and from_Reg denote the registers to which an lw instuction would store data and an sw instruction would read from, respectively.

In the case of next instruction being an add/addi/sub/mul/slt instruction, let Read_Reg and Write_Reg be the registers from which values are read from or written to by the next instruction.

There are 3 cases when execution of this next instruction would not be safe :

1. When to_Reg and Read_Reg are same, i.e. the register to which lw is writing, is being read in execution of next instruction.

Ex. lw $t1, 1000

add $t2, $t1, $t3

2. When to_Reg and Write_Reg are same, i.e., both the instructions are writing to the same register.

Ex. lw $t1, 1000

add $t1, $t2, $t3

A sequential execution of above instructions should lead to the result of add instruction to be stored finally in $t1. But if we allowed this add instruction to execute before completion of lw, the lw operation would overwrite the value. Hence, unsafe.

3. When from_Reg and Write_Reg are same, i.e., the register which is being read in sw instruction is being written in the next instruction.

Ex. sw $t3, 1004

add $t3, $t2, $t1

In other cases, it is considered safe to execute the next instruction.

STRENGTHS

This implementation leads to faster execution when the input contains of a large number of independent instructions following the lw/sw instructions.

In such cases, large values of row and column access delays lead to a significant decrease in total number of clock cycles .

WEAKNESSES

In case of consecutive lw/sw instructions or a large number of dependent instructions following these lw/sw instructions, this approach won't help to reduce the number of cycles significantly.

If row and column access delays have small values, then there would not be much improvement in performance.

SCOPE OF IMPROVEMENT

Whenever there is an lw/sw instruction, it needs to wait till the previous instruction accessing DRAM is completed. The futher instructions, even if independent of these two, would not be executed till the second lw/sw instruction issues DRAM request. Such cases can be better handled by storing some reference to the next lw/sw instruction and executing the further instructions, if they are independent.

Such an implementation would be faster in some sense, but it would increase the space complexity, since we need to store the instructions. Moreover, while checking whether the further instruction is independent or not, we would need to make comparisons with every instruction in queue.

INPUT - OUTPUT FILES

- Row_access delay and col_access_delay are passed as command line arguments to the executable file.
- INPUT FILES :
  "input.txt" contains the instructions following MIPS format conventions.
  "RF_init.txt" contains the values to initialise the registers.
  "dataMem.txt" contain the initial values stored in memory (address – value pair).
- OUTPUT FILES:
  The updated values in data memory are written to "dataMem_final.txt".

The states of registers after each instruction updating them are written to “RF_final.txt”.

- Statements related to processing are printed on console.
- Statistics involving the number of clock cycles, number of buffer updates and the number of times different instructions are executed, are also printed to console.

TESTING STRATEGY

- SAMPLE TESTCASE

```
lw $t1, 1000
addi $t2, $t2, -5
slt $t3, $t2, $zero
mul $t3, $t3, $t2
sub $t2, $t2, 5
slt $t3, $t3, 5
addi $t2, $t2, 5
mul $t3, $t3, 5
add $t2, $t2, 5
mul $t3, $t3, 5
add $t2, $t2, 5
addi $t2, $t2, -5
slt $t3, $t2, $zero
mul $t3, $t3, $t2
```

- OUTPUT

```
Number of cycles using part1 of implementation : 26
Number of cycles using part2 (i.e. non blocking)  : 14
```

NOTE :
Since each test case requires different initialisation of data memory, dataMem.txt needs to be updated accordingly before running each testcase. The submitted dataMem.txt has data initialised for “input.txt”, “testcase1.txt”. Also, given implementation does not take file name as input. Hence, kindly name the required testcase as “input.txt” while checking.