Name: T Abishek Entry Number: 2019CS10407
Name: Gaurav Jain Entry Number: 2019CS10349

COL216: Assignment 4

Introduction:

This assignment is implemented to improve further the design of the MIPS simulator developed in the previous assignments. In this assignment, we reorder the servicing of the load and write instructions in the 2D DRAM structure implemented in the minor exam.
We also apply optimization techniques like the non-blocking memory features and keep load and write instructions in a queuing system.
After reordering the servicing order of load and write instructions, we can see significant improvements in the number of clock cycles and the number of times the row buffer is updated.
To optimize further, redundant load instructions are removed from the set of instructions to be executed. This may be useful when we have a pending load instruction in the system. We find an add or any other non-DRAM instruction whose destination register is the same as the register in which the awaiting load instruction loads data.
We also check for whether the row buffer is updated or not. Row buffer may not be updated if there are only load instructions and no write instructions. We will not write back the row buffer back into the memory as there are no changes.

Approach to the code:

We reuse a lot of the previous code in the assignments to help with parsing and printing the output. Still, we also had to rewrite a substantial part of the code to meet the requirements of this assignment as it requires us to handle labels for jumping and also includes all the commands used in assignment 2. a map with string as key and int as value is used for storing the instruction addresses which each label is pointing to. This storing of data is done while parsing the instructions and then further referenced when needed while executing the program, i.e., when a j, bqe, or bne command is executed. The address is then passed on to a variable that controls the next instruction that will be executed.
We then go through the vector of instructions executing the instructions. We also use the non-blocking feature from the previous assignment, adapted to work in this assignment. When we encounter the first lw/sw instruction, we execute it and remember the row. We then continue executing the next non-memory instructions as usual. When we encounter the next lw/sw instruction, we check if the row required is the same as the row buffer. If it is, we execute the instruction. If not, we add it to the queue of lw/sw instructions pertaining to the same row. If it is an lw instruction that is stored in the queue, we also mark that register that the lw instruction is writing to with the queue that the instruction is getting stored to. This mark refers that the register is not updated with its actual values. Thus, if used, later on, the relevant lw/sw instructions in the queue should be executed to update the register with its correct values. When the next instructions are executed, if any register with this mark is encountered, the appropriate queue is executed before executing the instruction to keep the correctness of execution.
We also keep track of whether the row buffer is modified after being copied from memory. If it hasn’t been altered and a new row memory instruction is requested, the row buffer isn’t written back to memory to save ROW_ACCESS_DELAY cycles during execution.

Name: T Abishek Entry Number: 2019CS10407
Name: Gaurav Jain Entry Number: 2019CS10349

Correctness of the Approach:

There is a main invariant involved in our approach. The invariant is there is no dependent load or write instruction in the queuing system at any time of execution. So whenever there is a load or write instruction whose register, and address values depend on the queued instructions, the queued instruction is executed first.

This concept also applies to non-memory instructions. Suppose there are dependent values in this instruction. In that case, we can either execute the queue which contains the load instruction whose value is required, or we can terminate a load instruction whose value is overwritten by this non-memory instruction.

All in all, only independent memory instructions are queued, and they are executed whenever we require their output. This ensures that the order of non-memory doesn't change during execution.

Testing method:

Our testing strategy involved thinking and figuring out what can happen when a user interacts with our program and then implementing it to handle such cases. There are various optimization methods used in the code, and we try to verify each method's implementation by separate test cases.

The testing strategy also involved trying out the code on general programs, including constant use of read and write instructions on DRAM. We also include programs in the test cases which has a specific meaning attached to them. The final output can then easily be compared manually, and running time is compared with the previous assignments' code.