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

COL216: Assignment 5

Introduction:

This assignment is implemented to extend the MIPS simulator of a single-core processor to a simulator of a multicore processor. The main objective of this assignment is to maximize the throughput, that is, the instructions per cycle for N cores CPU, also called the IPC of the CPU.
The main feature added in this assignment is the Memory Request Manager (MRM). It acts as an interface for the DRAM. All DRAM instructions have to pass through this system. The main task of MRM is to choose the next memory instruction to be executed from the available choices. It should also react to memory instructions that are urgently required to be executed.
The MRM has to be implemented efficiently for higher IPC. The MRM should take the minimum number of cycles for selecting the next memory instruction.

Approach to the code:

- A separate class for MRM is added to the assignment four code to include all the operations that MRM does. In previous assignments, a variable was maintained to denote the current row number. In contrast, in this assignment, since all memory instructions pass through MRM, all memory instructions are queued in their respective core queues.
- The main job of MRM is to select the best memory instruction to execute next. The first check that MRM performs is to find a queued memory instruction of the same row. If the search is successful, then this instruction is selected to execute next. The number of cycles taken by MRM to perform this search is discussed in the timing model.
- If the search is unsuccessful, then the MRM selects the core whose queue has maximum memory instructions. The first instruction of such a queue is set to execute next. Again, the number of cycles taken by MRM is discussed in the timing model.
- The MRM also needs to take care of priority requests by the cores. A core will send a priority request if a dependent instruction is encountered while executing non-memory instructions.
- In such a case, the MRM checks for the same row normal memory instructions and continues doing them. If no such instruction is found, it selects a priority memory instruction. If there are multiple priority requests, then the MRM puts them in a separate queue which is executed unless it is empty. MRM selects a priority request of the same row as the current instruction being executed. Otherwise, it selects the first element of the priority queue. Priority requests can be present at any stage of the execution, and the DRAM can be active or inactive. So, the MRM keeps on checking the priority queue for a priority instruction request.
- Synchronization is necessary for the proper execution of all cores. We have three layers of execution. In the first layer, non-memory instructions are executed in one clock cycle. The second layer is the MRM which selects the best instruction to

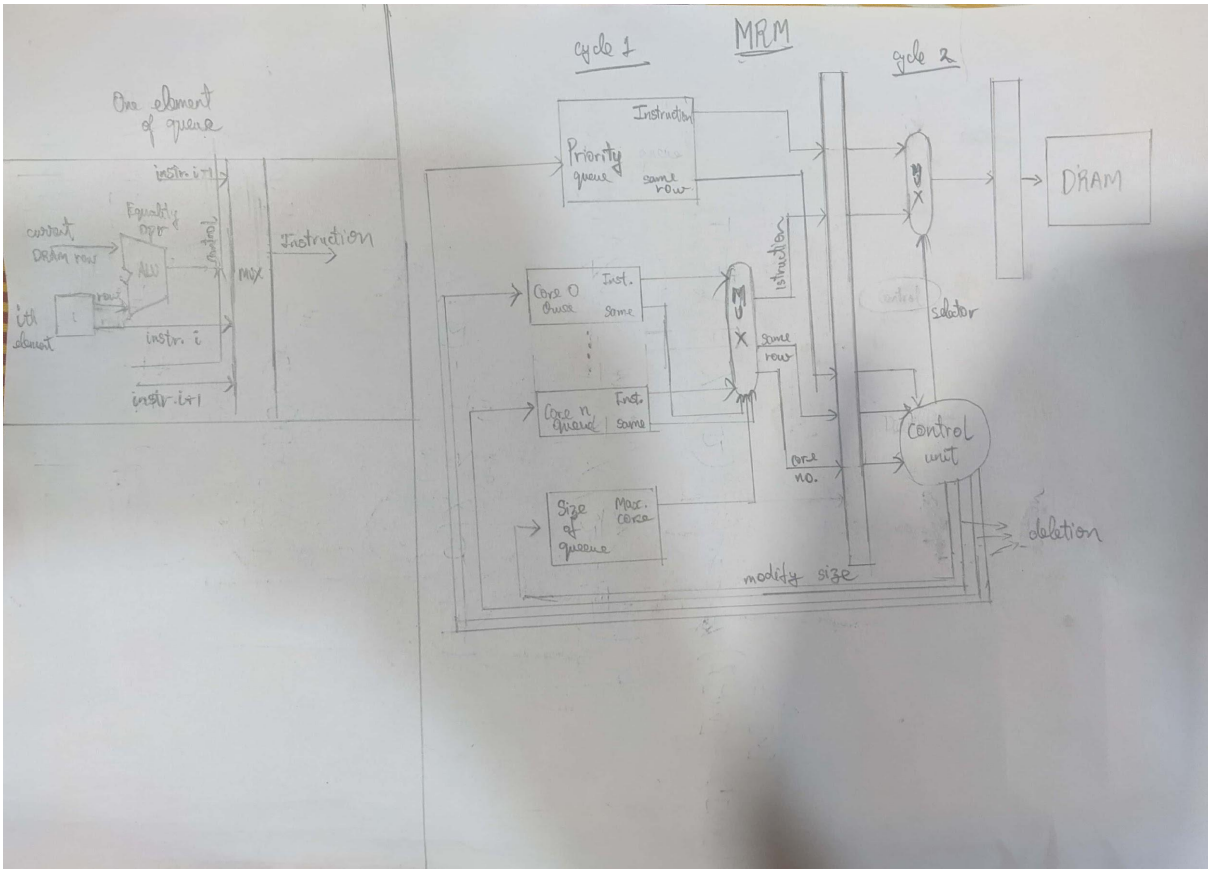
process next. The third layer is the DRAM which performs the load and store instructions.

- All these three layers work parallel to each other and are interlinked through priority requests and DRAM calls. There are separate functions for each layer, and these functions are called in every clock cycle.

Correctness of the Approach:

The one invariant that can be used to explain the correctness of the code is that the queues of each core contain only independent memory instructions. If we encounter a dependent instruction, we stop the execution of that core until the memory instruction which makes the encountered instruction is executed.
Forwarding and removing redundant instructions are also present to show that only instructions with different registers or memory locations are present in the queues.
The correctness of the approach is also ensured by the fact that there is no change in the order of non-memory instructions. This indicates that the state of registers will remain the same compared to normal execution.

MRM Timing Model:



We have shown a rough representation of our algorithm in hardware in the above diagram. Each queue element has its separate ALU unit, which compares the row with the current row buffer and simultaneously sends this information to the queue's multiplexer. This multiplexer then sends the oldest instruction of the same row buffer forward along with a *bit* stating that

that instruction is of the same row buffer. If a request of the same row buffer is not available, it sends the oldest instruction. Using the information forwarded by each queue's multiplexer, the overall multiplexer then decides which instruction to send forward, preferring the same row buffer over the oldest instruction of the biggest queue. Relevant information is also sent to the pipeline register for the control unit in the next cycle. A similar event occurs in the priority queue too, where the oldest instruction of the same row buffer is preferred over the oldest instruction in the queue. All of this is stored in the pipeline registers, which will then be used in the next cycle.
In the next cycle, the information from the pipeline registers is read. The control unit uses the multiplexer to choose between the priority instruction and the normal instruction where the same row buffer instruction is preferred. If both are the same, priority instruction is preferred. Relevant controls are also activated to delete the instruction from their respective queues and modify the registers containing the size of all queues.
The final instruction is stored in the pipeline register from which the DRAM reads in the next cycle. Overall, MRM takes two cycles for this processing.

Strength and Weaknesses:

Our MRM algorithm code gives preference to instructions in the descending order given below:

1) Priority instructions of the same row buffer
2) Regular instructions of the same row buffer
3) Priority instructions head(oldest instruction)
4) Head of the largest queue. (oldest instruction of the core that gives the most memory requests)

Doing this gives us certain advantages and disadvantages in our code, which helps some instances and is detrimental in other cases.
By giving the first preference to **priority instructions** over regular instructions, we minimize the time a core is idle, thereby increasing overall throughput. By preferring the same row buffer instructions over the oldest instruction in the **priority queue,** we minimize the wasted cycles while changing the row buffer and saving 20 cycles. This also has a **weakness**. I.e., in cases where a core keeps giving instructions of the same row buffer, it would make the other cores who have priority instructions waste cycles by idling. This can even be extreme in some instances, where only one core keeps executing instructions while all the cores go idle.
Picking the queue with the most instructions and then picking its head is advantageous because the queue with the most requests is the core that gives more memory requests than the other cores. Thus, it is more likely to have dependent instructions further on and therefore needs to be executed first. Picking its head gives us the oldest instruction and is picked because there are more chances of further instruction in the core dependent on it.
There can be cases where the MRM algorithm fails. The cycles taken using MRM can be more compared to sequentially executing the cores one by one. This type of case is needed to be handled separately. But, this case is scarce, and thus, we have not designed the approach for such cases. One such case is the same file is passed to all cores, and due to the same nature of the instructions, the MRM keeps on switching the row buffer. This will cost us a lot of clock cycles.

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

Testing methodology:

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. We also include programs in the test cases which has a specific meaning attached to them. These cases are used to check the MRM algorithm in different scenarios to verify whether all functionalities that we have implemented are working correctly. This can be seen by the test cases that we have provided. These cases are comparatively small, and their description of each cycle can easily be checked for verification.