COL216 Assignment 5

-Shreya Arora (2019CS10401)
-Rakshita Choudhary (2019CS10389)

Approach and Design Considerations

Inputs and Variables Considered

We take the number of CPU cores, N, as an input from the user along with the simulation time, M, in the form of the number of cycles till where we would be observing the activity in the different cores. We then take as input the names of the instruction files for each core in sequential order along with the value of the row_access_delay and the col_access_delay. We add the names of these instruction files to a 'files' vector which is used in the sim function.

In the sim function, we have defined a memoryBlock of size 2^{20} bytes and an instBlock for storing the instructions of size 2^{16}. PC is the program counter initialized to the start address of the memory block. We define a string s to be used to read lines from the input files. We define num[N] as the counter for the number of instructions in each input file and invalid_instruction[N] as the count to indicate the number of invalid instructions found in each core. We initialize the ROW_BUFFER_UPDATE variable to 0. The data is stored in a separate 'dram' memory block (which is different from the one where the instructions are stored). This dram memory block is divided equally among the N cores that we have and each core gets its individual space in the dram memory block of size dram_mem = dram/N. We define the vector counter as the iterator across the start instruction of all cores and we initialize it to the instBlock that we have defined. The vector labels[N] stores the label strings for all cores and the vector labelnum[N] stores the instruction numbers corresponding to the labels for all cores.

The boolean vector error[N] stores the number of error instructions in each core. The vector inst_num[N] stores the iterators across the instructions for each core. The boolean vector stopped maintains the status of each core indicating whether the further execution of the instructions of that core is ongoing or stopped for the time being with true indicating stopped and false indicating ongoing. For each input file corresponding to each core, we read its instructions line by line and store them in the memory and check for invalid instructions if any in each of the cores. The instruction memory stores the instructions of each core in a sequential manner.

The vector lw_dependencies[N] stores those registers in which the values are being loaded from some other address registers in the lw instructions appearing in our input set of instructions in each core and lw_instr_num[N] stores the instruction numbers corresponding to register names in lw_dependencies[N] in each core. The vector sw_dependencies[N] stores the address registers from where the value is to be loaded in the lw instructions or both the registers of the sw instructions are stored for all cores. The vector sw_instr_num[N] stores the instruction numbers corresponding to register names in sw_dependencies[N] in each core. The vector addresses[N] stores the addresses being accessed by the lw and sw instructions in each core and addr_inst[N] stores the instruction numbers of the corresponding addresses for each core. ROW_BUFFER is the array of size 1024.

Row_buffer_num stores the row number in row buffer. The vector reg_names[N] stores the names of the registers for all cores. The vectors count_add[N], count_sub[N], count_mul[N], count_beq[N], count_bne[N], count_slt[N], count_j[N], count_lw[N], count_sw[N], count_addi[N]indicate the number of instructions of each type that will be executed in the entire program as per the instructions given in the input text files for all cores and all are initialized to 0. The variable total_exe_inst maintains the total number of instructions executed in the entire program for all cores. The pair vector queue maintains the waiting instructions in pair formats where the first element of that pair is the core number of the waiting instruction and the second element of the pair maintains the instruction number of that instruction belonging to the core present as the first element of the pair.

The boolean variable DRAM_busy when false indicates that the DRAM is available to execute instructions and when it is true, it indicates that the DRAM is busy with the execution of another instruction of some core. The integral variable time_left maintains the number of cycles remaining for an ongoing DRAM instruction to complete its execution. The integral variable actual_time maintains the number of cycles passed from the point of beginning the execution of an ongoing DRAM instruction. The pair ‘current’ maintains the instruction whose execution is ongoing in the DRAM currently. The pair execute_now maintains the waiting instruction selected to be executed next based on the decision made by the memory request manager. The string curr maintains whether the instruction is of lw type or of sw type. The variable rbu maintains the number of row buffer updates for a particular instruction of a particular core.

The execution of the instructions of all cores goes on till we reach the point when cycle_count becomes equal to the simulation time, M. We begin by checking if the time_left is non zero and further go on to check if it is equal to 1 or not. If it is not 1, then we can deduce that DRAM execution of an instruction is still going on and then in this cycle, we reduce the time_left by 1. If the time_left is equal to 0 in the current cycle then the DRAM execution of the ongoing instruction is completed. If the DRAM instruction that just completed was an lw type instruction then we remove the dependencies of this instruction from lw_dependencies, lw_instr_num, addresses, addr_inst, and the queue and set it to be executed after loading the required value in the registers. If the DRAM instruction that just completed was an sw type instruction then we remove the dependencies of this instruction from addresses, addr_inst, and the queue and set it to be executed after storing the required value in the registers.

Since we have a word addressable memory, all addresses need to be divisible by 4, and hence in order to avoid any problem relating to addresses not divisible by 4, while dividing the DRAM among the different cores, we reduce the size of each of the DRAM blocks for the different core by 0 or 1 or 2 or 3 to arrive at a number that is divisible by 4 so that all addresses are divisible by 4.

Memory Request Manager (MRM) Implementation

Our **memory request manager (MRM)** begins by checking if our queue is empty or not. If it is not empty then, in case there is only one waiting request in the queue then that is set to be executed by assigning execute_now to the waiting request core and corresponding instruction number pair, and consequently, the DRAM is set to be busy. If there are more than one waiting requests in the queue, then we set mrmWorking to be true to indicate that the MRM has begun its process is in the decision-making process currently to arrive at the next instruction pair to be executed.

We iterate through the queue and calculate the row number being accessed by the lw or sw waiting instructions of the different cores and the **first preference** is given to the first lw or sw instruction encountered in the queue while iterating from the beginning of the queue that is accessing the same row in the DRAM. If such an instruction is encountered then we first check if there are any dependencies of such an lw instruction in the lw_dependencies, sw_dependencies, and addresses of that core, or in case of an sw instruction, we check for dependencies in the lw_dependencies, and addresses of that core. In case no dependencies are found, this instruction is set to be executed by assigning execute_now to its core and corresponding instruction number pair, the variable found, which indicates whether a decision has been made by the MRM on the next instruction to be executed in the DRAM based on row similarity, is set to be true, and the DRAM is set to be busy.

If a decision on the next instruction to be executed is not made till now, which means that there are no instructions in the waiting queue that are accessing the same row as that activated in the DRAM buffer then the **next preference** is given to an instruction in the same core as that of the current instruction that just completed its execution in the DRAM. While iterating through the instructions after the current instruction number of the current core, we check if these instructions have been executed already. If an instruction other than the lw or sw type instruction is encountered then, we set stopped for that core to be false in order to continue the execution of the instructions other than lw or sw type instructions that were not able to execute earlier on because of being dependent on a waiting lw or sw type instruction. Then, if found was not true and an lw or sw type instruction is encountered in this core then the **second preference** for the DRAM is given to such an instruction if any dependencies are not found and the DRAM is set to be busy. If the next instruction to be executed in the DRAM is still not found then we iterate through those instructions in the waiting queue that belong to this core and the **third preference** is given to the first lw or sw type instruction if encountered in the waiting queue belonging to this core. If the next instruction to be executed in the DRAM is still not decided then we iterate through all the different cores from the beginning and check if that core's execution has been stopped due to a dependency on a waiting instruction in the queue previously. If an lw or sw type instruction is encountered here the **fourth preference** is given to such an instruction after checking for no dependencies and the DRAM is set to be busy. If the decision for the next instruction to be executed in the DRAM is still not made, based on the above conditions then we iterate through the cores starting from the next core of the current one being considered till now. Here, again, if an lw or sw type instruction is encountered then the **fifth preference** is given to such an instruction after checking for no dependencies and the DRAM is set to be busy. If the decision for the next instruction to be executed in the DRAM is still not made, based on the above conditions, then we iterate through the queue and check for any waiting

instruction of the core being accessed presently during the iteration through the cores starting from the i+1th core. If a waiting instruction is encountered for this core in the queue, then that is set to exeucte_now and the DRAM is set to be busy. Finally, if the decision for the next instruction to be executed in the DRAM is still not made, then we check if there is only one instruction in the queue and if that is the case then we set this instruction to be executed. If that is not the case, then we execute the waiting instruction of the smallest instruction number of the core of the first waiting instruction in the queue as our **final sixth preference**. During the execution of the MRM, no new instructions are added to the waiting queue in order to ensure the efficient working of the MRM.

Memory Request Manager Delay Estimation

We have considered our MRM to take 3 cycles in order to make a decision on which instruction is going to be executed next in the DRAM. Our rationale beyond this is that the working of our Memory Request Manager which utilizes loops, arithmetic operations, and comparisons to check for conditions can be sufficiently implemented via hardware consisting of gates and multiplexers that would be capable of performing parallel computations involved in the MRM. Our assumption is that there would be sufficient hardware that would be able to implement the loops across all cores or all instructions in the queue can occur parallelly. The delay of 3 cycles is an average figure across all cases and this delay can be lesser in cases the decision for the next instruction to be executed is made as early as the first preference case itself. We have considered that identification of the first two preferences takes up one cycle while looping through the queue in order to identify the instruction accessing the same row number and while iterating through the current core instructions to identify the second preference. The second cycle is taken up in finding the third and the fourth preferences while looking for the first waiting instruction of the current core present in the waiting queue and while iterating through the cores to check for the core whose execution has been stopped due to a dependency. Finally, the last cycle will be taken up by the last two preferences if the decision for the next instruction has not been made still.

Execution of Instructions

We iterate through all the cores and if the iterator for the core is less than the total number of instructions of this core, and the execution for this core has not been stopped because of a dependency on a waiting instruction in the queue and if no invalid or error instructions are found then we proceed with the execution of the instructions for this core.

If the current instruction is not of lw or sw type, has not been executed yet and in case it is being executed alongside an ongoing DRAM instruction for this core then, the time_left is less than col_access_delay and the instruction in the DRAM is of lw type and mrmWorking is not set to be true and the present instruction type is not j, then we proceed further. This takes care of the constraint that no two registers from the same core can be simultaneously written to, during the last col_access_delay cycles of lw execution, we do not perform any other writing into the register file (i.e. no other R-type instruction).

For the instructions add, sub, mul, and slt, we first check if lw_dependencies and sw_dependencies are empty or if any of the 3 registers involved in these instructions are not present in lw_dependencies and if the result register is not present in sw_dependencies then, these instructions are executed straight away. In case there are dependencies in these instructions i.e. if any of the 3 registers involved in these instructions are present in lw_dependencies or if the result register is present in sw_dependencies, then, the index for instruction on which current instruction is dependent is stored in dependent_on for all the cores and then we go back to that instruction and set stopped to be true for this core while giving permission to execute that lw or sw type instruction causing the dependency.

For the addi instruction, we first check if lw_dependencies and sw_dependencies are empty or if any of the 2 registers involved in these instructions are not present in lw_dependencies and if the result register is not present in sw_dependencies then, these instructions are

executed straight away. In case there are dependencies in these instructions i.e. if any of the 2 registers involved in these instructions are present in lw_dependencies or if the result register is present in sw_dependencies, then, we go back to that instruction and set stopped to be true for this core while giving permission to execute that lw or sw type instruction causing the dependency.

For the instructions beq and bne, we first identify the line number corresponding to the label that is specified in the instruction. At this point, we also check if the instruction mentions a valid label present in the input instructions set, and if not, then we increase invalid_instructions for that particular core by one. After this, we execute all the instructions in the waiting queue for this core and keep a pause on current beq or bne instruction. After executing all waiting instructions for this core, we check if the check variable is false then we execute the current beq or bne instruction. Apart from this, if the destination line number that the instruction requires us to go to in the input instructions set is after the current instructions then we set all the instructions between this beq or bne instruction and the destination instruction to be executed. Similarly, if the destination line number that the instruction requires us to go to in the input instructions set is before the current instructions then we set all the instructions between this beq or bne instruction and the destination instruction to be not executed. In case the variable check is not false, then we stop the execution for this core for the time being waiting for the pending instructions to execute.

For the j instructions, we first identify the line number corresponding to the label that is specified in the instruction. At this point, we also check if the instruction mentions a valid label present in the input instructions set, and if not, then we increase invalid_instructions by one. After this, we execute all the instructions in the waiting queue for this core and keep a pause on current j instruction. After executing all waiting instructions for this core, we check if the check variable is false then we execute the current j instruction. Apart from this, if the destination line number that the instruction requires us to go to in the input instructions set is after the current instructions then we set all the instructions between this j instruction and the destination instruction to be executed. Similarly, if the destination line number that the instruction requires us to go to in the input instructions set is before the current instructions then we set all the instructions between this j instruction and the destination instruction to be not executed. In case the variable check is not false, then we stop the execution for this core for the time being waiting for the pending instructions to execute.

We have taken our waiting queue to be of finite size having a maximum of 32 requests at a time. Our rationale behind this is based on our extensive testing of the program for different sizes of the queue with different test cases and we got the most optimum performance of our program with simpler hardware in the case of the queue size equal to 32. We have assumed that our lw and sw instructions of the different cores can be added simultaneously to the waiting queue and that our hardware is capable enough to handle this since we have considered this to be a multiport scenario.

For any lw instruction, after reading the instruction, we check if this register already occurred in instructions and has a register mapped to it in the register array. Then we calculate the row number in the DRAM being accessed by this lw instruction. Then, we check if the queue is empty and if the execute_now pair is equal to this core and the corresponding instruction number pair. If the queue is empty, then we set the DRAM to be busy and push this

instruction and its corresponding dependencies into the queue. Post this, we execute the lw instruction depending on the row number it is trying to access.

Similarly, for any sw instruction, after reading the instruction, we check if this register already occurred in instructions and has a register mapped to it in the register array. Then we calculate the row number in the DRAM being accessed by this sw instruction. Then, we check if the queue is empty and if the execute_now pair is equal to this core and the corresponding instruction number pair. If the queue is empty, then we set the DRAM to be busy and push this instruction and its corresponding dependencies into the queue. Post this, we execute the sw instruction depending on the row number it is trying to access.

Finally, we iterate through all instructions of all cores to check if they haven't been executed yet and execute any and all pending instructions at this point.

Strengths and Weaknesses

Strengths of Our Approach

This approach reduces the number of clock cycles and the row buffer updates from what would have been possible if all instructions were executed sequentially for all cores and for all instructions in each core one at a time in the DRAM and processor combined for all cases wherever is feasible. Our program allows for an efficient execution of the instructions in the DRAM where the only time the DRAM stays unutilized is during the MRM delay period when the MRM is working on deciding which instruction is to be sent to the DRAM next for execution. While adding the lw or sw type instructions to the queue, we also take note of the values of the registers in those instructions at the time of reading them. Our program ensures that the semantics of our input instruction sets are maintained accurately for all cores. The implementation of our Memory Request Manager is efficient as well as practically implementable in hardware with a delay period of 3 cycles. We have not used any complex data structures like trees, maps, etc. which are difficult to implement when it comes to hardware. So we have confined ourselves to simple data structures like stacks, queues.

Weaknesses of Our Approach

One weakness of our approach is related to the execution of the branch and jump instructions as in these cases, we first execute all waiting instructions which leads to a greater number of clock cycles and row buffer updates as there could have been a more efficient approach of checking the destination line number and executing only those instructions which are absolutely necessary, however, to be on the safer side and ensure the correctness of the output, we have executed all instructions in the waiting queue whenever a beq or bne or j instruction is encountered. Our MRM does not simultaneously work with DRAM execution, thereby adding a delay of 2 cycles each time during its processing. This is because of the design of our MRM, it takes into account the queue from the time its processing starts and confines itself to the same. But it makes up for this weakness by providing a more efficient decision, hence overall saving up a lot of cycles.

Testing Strategy and Test Cases

It is unfeasible to test the code against all possible test cases because they can be infinite and there are no bounds to the number of test cases possible. So in order to test our code, we have identified the following categories of test cases and taken a couple of samples for them in order to verify if our code is working against all kinds of test cases. This would ensure that the code would work against all possible test cases as if it works for some samples of test cases of one category, we can assume that it would work for all cases of that category. Most categories of invalid inputs remain the same as the last assignment, except a few (following) mentioned changes.

The following categories of valid test cases have been considered:

1. Number of cores (N)= 0
2. Simulation Time (M) = 0
3. Single-core input instructions without lw or sw type instructions observed till complete or partial execution
4. Single-core input instructions including single sw or lw instructions consecutive block with combinations of add/addi/sub/mul etc till complete or partial execution
5. Single-core input instructions including multiple consecutive sw instructions, which are independent of one another in terms of update registers, address registers, memory addresses till complete or partial execution
6. Single-core input instructions including multiple consecutive lw instructions, which are independent of one another in terms of update registers, address registers, memory addresses till complete or partial execution
7. Single-core input instructions with sw or lw instructions consecutive to one another and dependent in terms of memory addresses or registers till complete or partial execution
8. Single-core input instructions with sw or lw instructions consecutive to one another and dependent in terms of update registers till complete or partial execution
9. Single-core input instructions with separate blocks of multiple lw and sw instructions, both starting with lw block as well as sw block till complete or partial execution
10. Single-core input instructions with combinations of lw and sw instructions independent of one another and consecutive to each other till complete or partial execution
11. Single-core input instructions with combinations of lw and sw instructions consecutive to each other and dependent in terms of update registers (single/multiple dependencies) till complete or partial execution
12. Single-core input instructions with combinations of lw and sw instructions consecutive to each other and dependent in terms of memory addresses/registers (single/multiple dependencies) till complete or partial execution
13. Single-core input instructions with combinations of lw and sw instructions consecutive to each other and dependent in terms of memory addresses as well as update registers till complete or partial execution
14. Single-core input instructions with lw and sw instructions with single add/addi/sub/mul/slt instruction in between, independent/dependent on update registers and/or memory registers till complete or partial execution
15. Single-core input instructions with lw and sw instructions with multiple add/addi/sub/mul/slt instruction in between, independent/dependent on update registers and/or memory registers till complete or partial execution

16. Single-core input instructions with lw and sw instructions with bne/beq/jump instructions (both jump/branch ahead and jump/branch behind) along with add/addi/sub/mul/slt instructions till complete or partial execution
17. Single-core input instructions with lw and sw instructions with some instructions having different address registers but same address values, and similarly same address registers but different addresses till complete or partial execution
18. Single-core input instructions with instructions involving addresses from multiple (different) rows, one after the other, forcing the program to make complex decisions and multiple iterations of the waiting queue till complete or partial execution
19. More than one cores with an error in instructions in at least one core
20. More than one cores with more than 32 DRAM requests at the same time from all cores
21. More than one cores without any lw or sw instructions in all cores till complete or partial execution

The following cases can have any combinations of the other instructions like add, sub, mul, slt, beq, bne, j, addi

22. More than one cores with only core have lw or sw instructions for all cases valid for a single-core case as specified above till complete or partial execution
23. More than one cores with all cores having lw or sw instructions, all accessing the same row in the DRAM occurring at different instruction numbers in different cores till complete or partial execution
24. More than one cores with all cores having lw or sw instructions, all accessing different rows in the DRAM occurring simultaneously in different cores till complete or partial execution
25. More than one cores with all cores having lw or sw instructions, all accessing different rows in the DRAM occurring at different instruction numbers in different cores till complete or partial execution
26. More than one cores with all cores having lw or sw instructions, all accessing different rows in the DRAM occurring simultaneously in different cores till complete or partial execution
27. More than one cores with all cores having lw or sw instructions such that at a time, there is only one instruction in the waiting queue for all cores combined till complete or partial execution
28. More than one cores with all cores having lw or sw instructions such that the waiting queue has more than one instruction accessing the same row till complete or partial execution. This is the case when the first preference would be selected by our MRM.
29. More than one cores with all cores having lw or sw instructions such that the waiting queue has instructions accessing different rows and the execution of one core is stopped due to the presence of a dependency on an lw or sw instruction till complete or partial execution. This is the case when the second preference would be selected by our MRM.
30. More than one cores with all cores having lw or sw instructions such that the waiting queue instructions are accessing different rows, the execution of the current cores has not been stopped and the core that just completed its execution has more than one pending instructions in the waiting queue till complete or partial execution. This is the case when the third preference would be selected by our MRM.

31. More than one cores with all cores having lw or sw instructions such that the waiting queue instructions are accessing different rows, the execution of the core has not been stopped, the current core doesn't have more than one instruction in the waiting queue and some other core's execution has been stopped due to a dependency till complete or partial execution. This is the case when the fourth preference would be selected by our MRM.
32. More than one cores with all cores having lw or sw instructions such that the waiting queue instructions are accessing different rows, the execution of the core has not been stopped, the current core doesn't have more than one instruction in the waiting queue, and some other core occurring after the current core sequentially has its execution stopped due to a dependency. This is the case when the fifth preference would be selected by our MRM.
33. More than one cores with all cores having lw or sw instructions such that the waiting queue instructions are accessing different rows, the execution of the core has not been stopped, the current core doesn't have more than one instruction in the waiting queue and no other core's execution has been stopped due to a dependency. This is the case when the sixth preference would be selected by our MRM.