COL216 Assignment5 I Grade

Rahul Chhabra 2019CS11016 **Harsh Tanwar 2019CS10432**

Objectives:
In Assignment 4 We had implemented a MIPS Simulator which worked on a single core and single bank. In this assignment, We are going to add the support for MultiCore and Multi-banked Dram in our DRAM implementation. The objectives of this assignment are:

- To design and implement a Memory Request manager which manages the order in which various lw/sw instructions are going to be executed in DRAM.
- To implement MultiBank support in DRAM. Our DRAM will now have four independent divisions each having its row buffers so that multiple DRAM instructions can be executed simultaneously.
- To maximize the number of instructions executed per cycle by rearranging the DRAM requests efficiently.
- To estimate the delay of the Memory Request Manager.

Instructions for Code Execution

Firstly type makes all in the command line to generate the executable. This will generate the executable main
Then type ./main N M <filepath for N cores> R C
Where
N corresponds to the Number of Cores
M corresponds to the Maximum number of cycles
After M we need to path of N files corresponding to each core
R corresponds to Row Access Delay
C corresponds to Column Access Delay

This will give the details of everything happening in Cores and MRM at each cycle and the end will print the relevant details like values of non-zero registers for each core, changed memory Location, row-buffer updates for each bank, and also the total row buffer updates and MRM delay.

Approach:
Instruction Parsing :
_____Each instruction is parsed sequentially. If it is a safe register operation then it is executed sequentially, if it is a safe memory operation(lw/sw) then this is sent to MRM. If it is an unsafe operation we set the priority number of the current bank of MRM to the issue-causing process number and halt for time till that operation becomes safe.
Memory Request Manager:
Memory Request Manager(MRM) calls the update function for each of the banks after each iteration. For any bank, if it is idle, it checks whether there is an element in the queue that corresponds to the current Bank. If such an element is present in the queue We sort the queue according to the following algorithm. MRM is also responsible for removing unwanted instructions and Forwarding.

Implementation

We have used object-oriented programming to make the code more modular. We have defined a class named core which can take input from the file and load it to the instruction memory and parse instruction one by one. There is another class named MRM which is nothing but our Memory Request Manager which supports functions for checking the dependency of any instruction, issuing DRAM requests m, Updating the state of the Currently ongoing process, etc. We loop through all cores until they are completely executed. At each iteration, we parse the next instruction of the core if it is safe or wait if it is unsafe.

After each iteration, The update function of MRM is called to update the cycles of any ongoing instruction. If all cores corresponding to any bank get completed We perform a writeback in case the row buffer has changed. After all instructions of cores get parsed We check if there are any pending instructions in the queue or not. If they are We execute them.

Handling multi-core with multi-bank:

Now, in this assignment, our DRAM has four banks which means four requests to DRAM can process simultaneously. So, it means that four different DRAM requests which use addresses in four different banks can run simultaneously independent of each other. So, to achieve that we made queues for storing instructions requested by MRM to each bank. To efficiently parse every core we implemented our MRM under the following two scenarios:

1. If the core has one or more than one bank: This is a case of single, dual, and quad-core. In this case queues of each bank have only the instructions of the same core. So, we can process every core like in the previous assignment but after execution of every instruction, we increment the clock cycle of every bank in that particular core.
2. If banks have more than one core: This is the case of more than 4 cores like octa, Hexa, etc. In this case queues of each bank might have instructions of more than one core. So in this case, if the row buffer of the current core changes or there is a dependency on another core then MRM forcefully changes the core to decrease delay.

MRM Delay Estimation:

For estimation of MRM Delay, we will compare the operations done in the processor in one clock cycle. In one clock cycle of a processor, a lot of operations are happening like fetching of instructions, reading register value, writing to register, arithmetic computations, instructions decoding, etc. All of this is happening in one clock cycle.

In one MRM operation of selecting the process and sending a request to DRAM the processes involved are pushing requests to the queue, priority checking, extraction of the

most important instruction from the queue, comparison of row-buffer row with the new row, and issuing of DRAM Request.

Now, let's compare the operations done in MRM with those done in the processor:

Pushing requests to queue and DRAM are analogous to fetching an instruction from the instruction memory as done by the processor.
Deciding priority based on current core, current instruction, and latest unsafe instruction is equivalent to performing Complex logical arithmetic computations in the processor.
Decoding instruction from the queue is equivalent to Decoding instruction in the processor.
Therefore the delay of one MRM process can be estimated to be equal to delay of one processor request ie. 1 Clock Cycle.

We are trying to maximise the throughput
Throughput is given as ration of total instructions executed to the Max cycles available
Throughput = (Total number of instructions)

Strengths:

- If more than one load word instruction is changing the value of the same register then the operations except the last one are not executed and removed from the queue.
- If more than one store word instruction is changing the value of the same memory location then the operations except the last one are not executed and removed from the queue.
- If a load word instruction is encountered and a store word instruction corresponding to the same memory location is already present in the queue for execution we don’t send a dram request, instead, we take the value from the queue and load it in the required register this saving an appreciable number of cycles (Forwarding).
- Non-Blocking memory is used which allows instructions to execute in the background.
- If after loading a row in row buffer, no change has been made in it and after a new row request the row buffer will not be written back to the dram.
- If an addi, add, sub, etc. instructions attempt to change the value of a register then any lw instruction present in the queue modifying the value of the same register is also considered invalid.
- The size of the queues of each bank is fixed to 16.
- We also implement a priority variable to ensure that no starvation occurs. It also ensures that no core is ignored over the execution.
- Our MRM does not wait for all DRAM banks to become idle in order to send requests for other instructions, it performs forwarding, removes invalid requests, etc. independent of the DRAM. It also sends requests to the DRAM’s bank if it becomes empty even if other banks are occupied. It saves a lot of time.

- In the case of cores more than four, we change the core only when the rows corresponding to the row-buffer in the queue are executed to minimize the delay.

Weaknesses:

- The code cannot identify infinite loops so it has to stop manually in such a case.
- In case of a sequence of instructions with so many dependencies , all instructions will be executed sequentially thus taking a lot of time.
- When a dependent statement occurs, we first complete all the other requests of the same row as row-buffer, then move to the dependent statement. It decreases the total clock cycle but increases time to execute dependent instruction.
- The code cannot tackle addi operations having value of integer greater than $2^{16.}$
- The code cannot tackle offset value of load and store words greater than $2^{16.}$
- The Size of each queue cannot exceed 16 so incase it reaches 16 we will have to wait for the instruction to get executed thus wasting some time.
- Our strategy to maximize throughput by changing core only when current row-buffer changes is only to ensure better throughput and parallel parsing of different cores. It does give better throughput than the one in which we just move to another core as soon as its instruction is pushed into the queue. But it might be worse than parsing all instructions in a single core then change the core but this does not support parallel parsing.

Testing Strategy:

We have broadly created our test cases (in the folder) to test these types of cases:

- Test cases that fall in different categories that are mentioned above
- Test case where one of the subsequent instruction accesses memory allocating to instructions
- All different permutations of lw and sw accessing memory locations which either belong to the same row or different row or same location.
- Tescases where different cores are accessing same registers.
- Test cases where each of the core accesses same memory periodically.
- Test case where the first instruction that has been executed is declared invalid in subsequent instructions
- sw & lw instruction accessing same registers
- Invalid lw instruction trying to change $zero register
- Instruction accessing a busy register of lw but after that particular lw instruction has been executed
- Using multiple instructions accessing the same row.
- Using multiple instructions accessing different rows each time.
- Instructions where no lw/sw kind of instruction is there.
- The cycle of any lw/sw completes just before starting the next lw/sw/instructions.
- The cycle of any lw/sw completes just after starting the next lw/sw/instructions.
- The cycle of any lw/sw completes just after the completion of the next lw/sw/instructions.

- The same memory location is accessed but by using different registers say $s0 is storing 500 and $s1 is storing 1000 so 500($s0) and 0($s1) points to the same location in memory so I have used different sw/lw instruction but all pointing to the same memory locations.
- Checking whether it's ‘safe’ to execute an instruction with a different kind of permutation and combination defining dependencies of all types like all instruction use registers which are independent of each other or executing a linear dependency on the instruction set.
- Checking the compatibility of various dependent register instruction pairs like sw-lw,lw,sw,sw,add,sw,addi,sw-lw-addi,lw-add and so on.
- Intermixing cycles of completion of instructions.
- Checking for dependencies between various DRAM instructions.
- Checking whether instructions are executed sequentially or not.
- Testing for the formation of an infinite loop.