COL216
ASSIGNMENT – 5

ABHINAV SINGHAL (2019CS50768)   ABHISHEK KUMAR (2019CS10458)

Problem Statement: DRAM Request Manager for Multicore Processors

This MIPS Simulator has been built upon the MIPS Interpreter which was implemented in the previous assignments.

Major Design Considerations:

1 Input/Output:

- The various parameters such as the input folder where MIPS files for all the cores reside, the output folder where the outputs of DRAM and all the cores will be written in separate files, maximum number of clock cycles, number of cores, individual core's instruction buffer length, the row access delay, column access delay, all are taken from the command line.

- All the steps up to lexing and parsing the input file, storing the instructions in a struct format in the instruction memory remain same as in previous assignments, except for the fact that now we do so for each of the individual cores.

- We have handled all syntax and runtime errors in our implementation. If there are any errors, the simulator will print all the errors on the terminal and mention for which core(s) were the errors coming from. From then on, that core(s) stop functioning and rest of the cores keep on executing their own instructions.

2 Execution:

- At the beginning of each clock cycle, all the cores are instructed to execute their next instruction and update the pc accordingly.

- For each individual core we maintain a register file and an instruction queue buffer of fixed length as taken from the command line - the recommended size is 20, neither too short nor too long (because if it is too long then it will increase the duration of the clock cycle and if it is too short then the core will stall more frequently which would in turn decrease the performance of the core).

- Any kind of instruction can be sent into the buffer (except j instruction which can be always executed). All the instructions that will go into the buffer will also have some cost attached to them so that the memory request manager can prioritize which instruction has the least cost among all those waiting in the buffer and hence forward that request into the DRAM. We have kept this cost to be proportional to the amount of delay caused in terms of number of clock cycles wasted to execute that instruction.

First let us look at instructions other than lw or sw -

- For every instruction other than lw and sw, we check if they are independent or not by iterating over the buffer. If the instruction is independent, then it will be executed immediately (if the instruction modifies any register, then we first check if the write port of the register file is free or not (lw instruction from the DRAM could be modifying a register at that clock cycle). If so, we postpone the current instruction and take it up in the next cycle).

- If the instruction is dependent on some instructions in the buffer, then we enqueue it into the buffer with the cost of the instruction being the sum of the costs of all the instructions it is dependent on. But we also decrease the cost of all those instructions on which the current instruction was dependent on by a very small amount (say 0.01) so that these instructions get higher priority because by executing those instructions, we make this instruction independent and now this instruction can be taken up by the core.

- But here is a special case: beq and bne which are branch type instructions – if they are dependent, then once they are pushed into the buffer, then the core needs to **stall**. The core keeps waiting until that branch instruction is executed because it won't know which should be the next instruction.

Now, for lw and sw instructions –

- Regardless of whether they are independent or dependent, they are sent into the buffer and if they are independent, their cost is set to (2* row_access_delay + col_access_delay), whereas if they are dependent on some instructions, then the cost is summation of the cost of those instructions (and if that instruction is a lw or sw instruction, then we also add additional cost according to whether the row accessed is same or not, the column accessed is same or not etc. This helps the memory request manager to know that if a lw/sw instruction is executed, then it should prioritize that lw/ sw instruction next which has the same row and/or same column because its cost will be less*).

Like this the core will keep on taking new instructions and either execute them immediately or push them into the buffer. In case the instruction buffer is full, and the core wishes to push more instructions into the buffer, it will not be able to do so, and it will have to stall. Till the time some instruction from that buffer is not executed thereby reducing the buffers' length, the core will remain to be in a stalled state (meaning it will not take any further instructions for execution).

For each core, we also maintain another buffer (called the **free buffer**), which will have instructions which have been freed from the instruction buffer by the memory request manager because those instructions have become independent now and can be executed freely by the core.

3 Role of the memory request manager (MRM) –

The MRM will first pick the instruction with the lowest cost amongst the buffers of all the cores. The way in which we have implemented the priorities, executing this instruction would cause minimum wastage of clock cycles. This would automatically imply that that instruction is independent of all the instructions and hence can be processed immediately.

Due to our implementation design, there are only two possibilities for this minimum cost instruction –

- Either it is an independent lw/sw instruction, in which case it is issued to the DRAM,

- Or it is an independent instruction other than lw/sw, in which case it is pushed into the free buffer.

MRM Delay: Once that minimum cost instruction has been executed, we now need to delete it from the instruction buffer. Here is where the MRM wastes some clock cycles. MRM will need to iterate over the entire buffer and check which all instructions are dependent on the current executed instruction. For all those instructions which were dependent on it, we subtract the cost of those instructions by the cost of the executed instruction so that the priorities for the MRM are in correct order.

The number of instructions which were dependent on the executed instruction becomes our MRM delay because those many subtractions need to be performed by the MRM which will be time consuming.

As and when the current instruction is executed completely, the MRM begins to look for the next instruction to send to DRAM, and if the number of clock cycles taken by MRM is greater than the number of clock cycles the DRAM will take to finish its current instruction, then even if some lw/sw instructions are pending in the instruction buffers, the DRAM will not get those requests and it will sit idle because MRM is still deciding.

Strengths:

- We have used fixed sized array for DRAM requests for each core. Fixed size array is easily implementable in hardware.

- We are using non-blocking so that independent instruction donot have to wait for lw and sw to finish.

Weaknesses:

- We have focused primarily on increasing IPC. Thus it will take not care of any starvation. So a core having less contribution to IPC will remain stalled.

- We have only used estimation for MRM Delay. The actual delay depends upon hardware implementation.

Test – Cases: The implementation has been thoroughly tested with multiple test cases and five of the test cases and their respective outputs for all the cores and the DRAM have been attached with this document and the supporting C++ files.

TC – 1,2: Example of multi core functioning and single write port of register file, include loops, MRM Delays.

TC – 3: Includes all 10 type of instructions that we are considering.

TC – 4: More number of cores

TC – 5: Shows MRM Delay.

TC – 6: Shows handling of runtime errors.

~THANK YOU~