COL216 ASSIGNMENT 5

Avantika Agarwal (2019CS10338)

Aryan Jain (2019CS10334)

This assignment 5 builds over the interpreter built in assignment 3 and 4 and minor examination by Aryan Jain (2019CS10334) and Avantika Agarwal (2019CS10338). Hence, all design choices taken in assignment 3 and 4 follow through in this assignment.

The simulator is expected to load and execute **add, sub, mul, beq, bne, slt, j, lw, sw, addi** instructions, while also handling and properly processing any labels and/or comments in the file. The program was further required to maintain a structure representing a memory of size 2^20 as well as 32 integer registers (including the stack pointer).

In the previous assignment, we had to implement add, sub, mul, beq, bne, slt, j, lw, sw, addi instructions, and reorder the DRAM requests for lw, sw instructions so that the number of times the row is changed by DRAM is minimized. Further we had implemented non-blocking memory for all the instructions mentioned above.

Now we have to implement add, sub, mul, beq, bne, slt, j, lw, sw, addi instructions, for N processor cores, with a single DRAM queue, controlled by a memory request manager, which manages memory requests for all the processor cores. The goal is to maximize the throughput (number of instructions executed), within M clock cycles.

DESIGN CHOICES

1. The DRAM is an integer array of 1024 by 1024 addresses, all initialized to 0.
2. The row buffer is virtually an integer array of size 1025. The first integer element is used to store the row number that is currently present in the buffer. The remaining 1024 elements store the row currently present in the buffer.
3. An integer global variable row_in_buffer represents the first element of the row buffer.
4. The ROW_ACCES_DELAY and COLUMN_ACCESS_DELAY is taken as input from the user as arguments in the command line.
5. A row buffer update is only mentioned in the clock cycle where it is completed.
6. A vector (named DRAM_queue) is used to stores the DRAM requests as encountered during the execution process, at a suitable location, such that the number of times DRAM has to update its row, can be minimized. This vector is used by DRAM to determine which request has to be processed next. The order of requests in the DRAM queue is managed by a memory request manager. The size of this queue is fixed to 32, in accordance with the most common hardware architectures available.
7. A vector of sets of unsafe registers is maintained, and registers are removed from it, when they become safe. Each core has a different set, hence the registers do not clash.
8. lw instructions are stored in the DRAM_queue in the format
   <”lw”, register, row_number, column_number, starting clock_cycle, processor core number>

9. sw instructions are stored in the DRAM_queue in the format
   <"sw", value in the register, row_number, column_number, starting clock_cycle, processor core number>
10. Due to design choices 8 and 9, only the register in the lw instructions which will finally store the data from the memory, is unsafe. All other registers in an lw or sw instruction are safe as the data from them is stored in the DRAM_queue and hence an alteration in their values does not affect the resultant memory or register values.
11. We have implemented sw-lw forwarding.

APPROACH

1. Execution starts from the first instruction of each core and stops when we reach the end of instructions and the DRAM_queue is emptied, or when we reach the end of M clock cycles.
2. When the DRAM is working, other safe instructions in each of the cores get executed independently, except in the clock cycle where DRAM execution prints something.
3. However due to sequential execution, if any instruction uses an unsafe register, then we wait at that instruction until that register becomes safe to use. In the meantime, execution of instructions in other cores does not stop if they are safe.
4. When the program counter encounters a safe lw instruction, we start iterating from the end of DRAM_queue, and look for an lw instruction with the same register and belonging to the same processor core. If such an instruction is found, we remove that instruction from the queue, since its execution won't change the output. Next, we again start iterating from the end of queue and look for an sw instruction with the same address as the lw instruction. As soon as such an instruction is found, we implement sw-lw forwarding, and directly write the value in the register corresponding to sw, to the register corresponding to lw instruction in 1 cycle. If no such sw instruction is found, we find an appropriate location in the queue, where the row buffer already has the same row as the new DRAM request (we can now insert lw anywhere in the queue, since its execution does not affect any other instruction). If no such location is found, the instruction is inserted at the end of the queue.
5. When the program counter encounters a safe sw instruction, we start iterating from the end of the DRAM_queue, and look for an lw/sw instruction with the same address. If an lw instruction is found, we find an appropriate location in the queue after this instruction, where the row buffer already has the same row as the new DRAM request. If no such location is found, the instruction is inserted at the end of the queue. If an sw instruction writing to the same address is found, it is removed from the queue without affecting the semantics of the program, and we continue to iterate the queue, looking for lw/sw registers with the same address. If none of the instructions specified above is found, then the new instruction can be inserted anywhere in the queue, and we simply look for a location with the same row.
6. Note that in steps 5 and 6, we do not need to check the processor core number, except in the case of removing redundant lw instructions, because the memory blocks accessed by each of the processor cores are disjoint.
7. The number of instructions executed is increased by 1, after each instruction finishes execution. We maintain the number of times every instruction is processed by the processor.

8. If MRM tries to write to a register (due to sw-lw forwarding) in the same clock cycle as DRAM queue wants to write to a register of same processor, then preference is given to DRAM queue, and MRM write is executed in next clock cycle.

STRENGTHS OF THE APPROACH

1. Since we implement non-blocking memory as well, we save a lot of clock cycles by executing safe instructions while the DRAM is processing.
2. We rearrange the DRAM requests at the same time when we insert in the DRAM_queue, instead of modifying the queue after insertion. This saves time, and reduces the complexity of the implementation.
3. As we remove lw instructions with the same register from the queue, when inserting a new one, it saves a lot of clock cycles without changing the semantics of the program, and there is at most one lw instruction writing to a particular register in the DRAM_queue at a time.
4. As we remove some of the sw instructions with the same address from the queue, when inserting a new one, it saves a lot of clock cycles without changing the semantics of the program.
5. Sw-lw forwarding has also been implemented, so if an sw instruction writes to a memory address, and then an lw instruction reads from the same memory address, then we can directly write the sw register value to lw register. This saves clock cycles spent in accessing the memory address again for lw instruction.
6. Since we insert an instruction at the end of queue when no optimum location is found for it, the requests from one processor core do not get collected together in the queue, thus ensuring that no processor core is stuck at an unsafe instruction for too long.

WEAKNESSES OF THE APPROACH

1. Since the DRAM queue is of finite size (32), the execution of some processor cores might get stopped at lw/sw instructions, waiting for the queue to have free space, thus stopping execution of other safe instructions.
2. In our current implementation, the memory request manager will choose the lw/sw request from the processor core having smallest number, in case lw/sw instruction is encountered in more than one processor in a single clock cycle. So, if core 1 has continuous lw/sw requests, then the requests of other cores might never get served, thus leading to less throughput.
3. Since the MRM delay is a bit high, it reduces the impact of the optimizations we do in the DRAM queue ordering.

MEMORY REQUEST MANAGER TIME DELAY CALCULATION

We make use of 4 ALU units, all hardwired for equality check to reduce time delay. The first ALU checks for the instruction type in queue, second ALU checks for register equality, third ALU checks for address/row equality, and fourth ALU checks for processor core equality. We also make use of a multiplexer, which determines based on the output of 4 ALUs, what action to perform. We make 2 passes over the DRAM queue, for inserting each DRAM request in the queue. In the first pass, we make the checks for determining whether sw-lw forwarding can be achieved, redundant instructions can be removed, or if there is an instruction already in queue before which the new instruction can not be executed. As soon as we find that forwarding can be done or an instruction can be removed, we do that

simultaneously. In the second pass, we simply find a place appropriate for inserting the new instruction, such that row buffer does not need to be updated. Since the DRAM queue has a finite size 32, we need to perform 64 sequential comparisons in the worst case.

Time Delay of general ALU unit = 200ps

Time Delay of ALU unit hardwired to check equality (an approximation made by us) = 50 ps

Time delay of Multiplexer (approximation) = 25 ps

Total time delay for inserting one request by MRM = 64*(50ps + 25ps) = 4800 ps

Time required for R-format instructions = 600 ps (200 + 100 + 200 + 100)

Time required for branch instructions = 500 ps (200 + 100 + 200)

Since we have a single cycle implementation for R-type and branch instructions, we assume our clock cycle to be 600 ps long.

Therefore, MRM delay (in clock cycles) = 4800ps / 600ps = 8 clock cycles

TESTING STRATEGY

1. General test cases: Set 1
2. No lw/sw instructions (so MRM not used): Set 2
3. sw-lw forwarding: Set 3
4. Single processor stuck at unsafe instruction: Set 4
5. Processor stuck at unsafe in the beginning (affecting throughput for small M): Set 5
6. Multiple processors stuck at unsafe instructions: Set 5
7. MRM time delay greater than M: Any test set with M <= MRM delay
8. DRAM queue gets filled, so processor stuck at lw/sw instruction: Set 7
9. No unsafe instructions: Set 9
10. Lw/sw instructions only in one of the cores (to show queue reordering): Set 10
11. Continuous lw/sw requests from one core (to show that lw/sw requests from other cores are not served): Set 11, Set 7
12. Lw/sw instruction encountered while MRM is still processing another one: Set 11
13. Infinite loops: Set 12, Set 5
14. Instruction trying to access memory block outside its allocated domain: Set 13
15. Random testcases: Set 14
16. Small number of cores: Set 13
17. Large number of cores: Set 15