COL216 ASSIGNMENT 4

Avantika Agarwal (2019CS10338)

Aryan Jain (2019CS10334)

This assignment 4 builds over the interpreter built in assignment 3 and minor examination by Aryan Jain (2019CS10334) and Avantika Agarwal (2019CS10338). Hence, all design choices taken in assignment 3 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 minor assignment we were required to maintain a DRAM model for the memory, while using a row buffer to access any row from the memory. We were only required to implement the **add, addi, lw, sw** instructions, and make a non-blocking memory for the same.

Now we have 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 have implemented non-blocking memory for all the instructions mentioned above.

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.
7. A set of unsafe registers is maintained, and registers are removed from it, when they become safe.
8. lw instructions are stored in the DRAM_queue in the format <"lw", register, row_number, column_number, starting clock_cycle>
9. sw instructions are stored in the DRAM_queue in the format <"sw", value in the register, row_number, column_number, starting clock_cycle>
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.

APPROACH

1. Execution starts from the first instruction and stops when we reach the end of instructions and also the DRAM_queue is emptied.
2. When the DRAM is working, other safe instructions get executed, 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.
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. 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 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 the instruction specified above is not found, then the new instruction can be inserted anywhere in the queue, and we simply look for a location with the same row.
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.

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 the DRAM_queue is of unbounded size, any number of DRAM requests can be queued in it, so the execution does not stop at any lw/sw instruction, unless it is unsafe (whether it is safe or unsafe, can be checked using point 10 of design choices).
4. 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.
5. 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.

WEAKNESSES OF THE APPROACH

1. The memory used increases linearly with the size of the queue. In the worst case, the DRAM_queue might occupy as much space as the code does, in the memory.
2. As the queue grows, so does the set of unsafe registers, which makes checking the safety of a register in any further instruction much more time consuming.
3. If the set of unsafe registers becomes too large, then checking for the safety of a register can end up taking more than a clock_cycle, which would make our simulator unrealistic.
4. The time for determining the appropriate location for insertion in the queue for a lw/sw instruction also grows linearly with the size of the queue, and if this time becomes too large, it can end up taking more than one clock cycle, which would make our simulator unrealistic.
5. In the cases where we are removing the request currently in execution from DRAM_queue (to reduce redundancy), the number of clock cycles might increase in some cases (test13.asm), though not in all cases (test18.asm).

TESTING STRATEGY

1. General test cases: test1.asm, test2.asm (shows reduction of 40 clock cycles) (Testcases provided with minor examination)
2. Safe/unsafe registers: test3.asm, test4.asm, test5.asm, test6.asm
3. lw/sw only (to explicitly show how the requests are reordered): test7.asm, test8.asm (no optimization possible), test9.asm, test10.asm, test11.asm, test12.asm
4. Loops: test13.asm (shows weakness 5), test14.asm, test15.asm
5. Random testcases: test16.asm, test17.asm
6. Unsafe lw/sw instructions: test13.asm, test14.asm, test16.asm (reduction from 58 to 16 clock cycles), test17.asm (reduction from 124 to 82 clock cycles)