COL216 MINOR ASSIGNMENT

AVANTIKA AGARWAL (2019CS10338)

PROBLEM STATEMENT

1. To develop a model for the main memory and integrate it into the basic interpreter
2. To make non-blocking memory access

APPROACH FOR PART 1

A DRAM is implemented by creating a memory array of size 1024 bytes x 1024 bytes. The following operations are implemented: add, addi, lw, sw. The operations lw and sw issue a DRAM request. For each request, if the row buffer contains the row required, then the corresponding operation is done by accessing the column in time COLUMN_ACCESS_DELAY. If the row buffer does not contain any row, then required row is copied in the row buffer in time ROW_ACCESS_DELAY. Then the corresponding operation is performed by accessing the column in time COLUMN_ACCESS_DELAY. If the row buffer contains a different row, then the row buffer is written back to the memory array in time ROW_ACCESS_DELAY, new row is copied into row buffer in time ROW_ACCESS_DELAY, and then corresponding operation is performed by accessing the column in time COLUMN_ACCESS_DELAY. For the operations add and addi, we simply access the registers and perform the corresponding operation, and modify the required registers.

APPROACH FOR PART 2

To implement non-blocking memory, first we need to decide when is it safe and not safe to execute an operation and when it is not, while a DRAM operation is already in process.

1. When it is safe to execute an operation:
    a. For a lw instruction, we can execute those succeeding operations which do not read from or write to the first register involved in the lw operation, and which do not write to the second register (for address) involved in the lw operation. This is because, if we read from the first register, the value obtained might be different. If we write to that register, we will write before the write operation from lw has been done, and hence, the lw write will overwrite the value which is supposed to be written later, which will cause incorrect results. If we write to the second register, the lw operation might read an incorrect memory address.
    b. For a sw instruction, we can execute those succeeding operations which do not write to the register involved in the sw operation, and which do not write to the second register (for address) involved in the sw operation. We can safely read from the first register, because the sw operation does not modify the register value. We can't write to the first register, because the sw operation may read the register value after it has been overwritten by the instruction, which will cause incorrect results. We can't write to the second register because then the sw operation might read an incorrect address.
2. When it is not safe to execute an operation:
    a. While a lw or sw instruction is already in process in the DRAM, no other lw or sw instruction can be executed.

b. If an operation reads from or writes to first registers involved in preceding lw operations which have not yet been executed or are in process or if an operation writes to second registers (for address) involved in preceding lw operations which have not yet been executed or are in process.
c. If an operation writes to registers (either first or second) involved in preceding sw operations which have not yet been executed or are in process.

Once this has been decided, I take the following approach:

1. All add and addi instructions are executed sequentially and take 1 clock cycle to execute.
2. When a lw or sw instruction is encountered, the instruction issues a DRAM request, and is executed as explained in the DRAM implementation of part 1.
3. While we are waiting for the DRAM processing to finish, further instructions can be executed if their execution is safe.
4. While the DRAM has not finished its processing, I look ahead for further instructions, and execute them till they are safe. If an unsafe add/addi operation is encountered, the execution stops and I wait for the DRAM result.
5. If I encounter an lw or sw operation while DRAM is still processing, I store the program counter for the instruction, if it is the first such encountered instruction, raise a DRAM access request for the instruction, and move to next instruction (execution of request by DRAM occurs when it becomes free). If it is the second lw/sw instruction encountered in look ahead, I stop executing lookahead instructions and wait for the DRAM processing to finish.
6. If I have one instruction being processed in DRAM, and no other lw/sw instruction has been encountered and stored, I check safety of execution of instructions based only on the instruction in DRAM. If I have encountered one other lw/sw instruction whose program counter I have stored and whose execution has been skipped, I check safety of execution of add and addi instructions based on both these lw/sw operations (one in DRAM and one encountered later).
7. When the look ahead execution is stopped (either because the DRAM processing has finished, or because we encounter an unsafe instruction), I also store the program counter of the last instruction executed, so that execution can start again from there as needed.
8. Once current DRAM processing is finished, I check if there is some skipped lw/sw instruction stored. If so, it is executed and lookahead is performed starting from the instruction after the last instruction. Else, execution starts sequentially from the instruction after the last instruction which was executed.
9. In summary, if an unsafe add/addi instruction is encountered, we wait for DRAM processing to finish. If we encounter an lw/sw instruction while DRAM is processing, it is stored, and DRAM request issued. It is executed after DRAM becomes free, and further safe instructions executed meanwhile. If we encounter an lw/sw instruction, and one such instruction is already stored, then we stop and wait for DRAM processing to finish, then execute the already stored lw/sw instruction, and then issue request and execute this instruction. Whenever we stop and wait for DRAM processing to finish, further execution begins from either the stored lw/sw instruction and then goes to the instruction after the last safe instruction executed, or goes directly to the instruction after the last safe instruction executed if no lw/sw stored.

Strengths of my approach:

1. The extra storage required is constant, since I only store up to one lw/sw instruction, while DRAM is processing another request.
2. Checking for the safety of lookahead instructions takes constant time. Had I stored multiple lw/sw instructions in a queue, the complexity for checking the safety increases linearly with the size of the queue, which can be more than the ROW_ACCESS_DELAY if
3. The execution does not stop and waits for DRAM processing to finish as soon as another lw/sw instruction is encountered.
4. The DRAM access request is issued as soon as another lw/sw instruction is encountered, while one is already in process, which saves one clock cycle.

Weaknesses of my approach:

1. If three lw/sw instructions are to be executed one after the other, my implementations waits for the output of DRAM before executing the next lw/sw instruction, and no further instructions (after the third lw/sw operation) are executed before the DRAM processing finishes for the second lw/sw instruction. This is a time tradeoff that occurs in order to keep constant storage for instructions, and to keep constant time for checking safety of instructions.
2. As soon as an unsafe instruction is encountered (add/addi), the program waits for the DRAM process to finish before executing this unsafe instruction, and any further instructions. Hence, if the instruction very next to lw/sw is unsafe, no instructions will be executed before DRAM processing finishes.

ERROR HANDLING

The code throws an error in following situations:

1. Incorrect command line arguments
2. Syntax errors in code
3. If some operation other than add, addi, lw, sw is used in code
4. Trying to access reserved/non-existing registers
5. Trying to access invalid address (address in instruction memory, address not corresponding to a word address, address outside memory range)

TESTING STRATEGY

The following kinds of testcases have been considered:

1. General testcases: test1.txt, test2.txt (as given along with assignment), test6.txt
2. Basic test for part 2: test3.txt
3. Stores 1 lw/sw instruction, executes it after DRAM processing stops, and then looks ahead starting from the instruction after last instruction executed: test4.txt
4. No lw/sw instruction is stored, unsafe instruction encountered before that: test5.txt, test9.txt
5. Memory address offset different from 0: test3.txt (line 1), test7.txt (line 6), test8.txt (line 18), test9.txt (line 2)
6. Encounters second lw/sw instruction when one is already stored: test8.txt, test1.txt, test2.txt
7. More than two consecutive lw/sw instructions: test1.txt, test2.txt
8. First instruction is lw/sw: test9.txt, test3.txt

9. All instructions encountered while DRAM is processing are safe (assuming ROW_ACCESS_DELAY = 10 and COLUMN_ACCESS_DELAY = 2 cycles): test10.txt, test11.txt (one sw instruction is also stored during DRAM processing in test11.txt)
10. Instruction right after lw/sw is unsafe: test7.txt (line 10, 12), test9.txt (line 2, 3)
11. Look ahead instruction tries to write to second register (for address) in preceding lw/sw operation: test9.txt (line 2,3 for ongoing lw operation), test12.txt (line 10, 11 for stored lw operation)

All these testcases give correct output when run both for part 1 and part 2.