COL216: MINOR

Instructions to run code :

1. After unzipping, a c++ file, interpreter2.0.cpp will be formed, this is the main file containing both the parts.

2. Compile the cpp file using g++ compiler using the command “g++ interpreter2.0.cpp”

3. An executable a.exe will be formed, run it using instruction of format, “./a.exe testfile rowdel coldel mode” where testfile is the file that want to be executed, rowdel is ROW_ACCESS_DELAY, coldel is COLUMN_ACCESS_DELAY, and for part a, mode = “a” and for part b mode = “b”.

4. A folder named testcases will also be formed, containing testcases files and additional 3 text files containing output, one for testcases of part a another for part b and third containing output of some big test cases of both part a and part b.

PART a:

Approach

1. **Memory:** I have implemented DRAM memory using a 2-d array of size 1024*256. **NOTE**: Like, assignment 3, 4 bytes of MIPS memory is considered and one word is stored as single element of array. Causing the 1024*1024 bytes memory structure shrinking to only 1024*256.

2. **Row Buffer:** A 256 size array named rowBuffer will denote row buffer and is used to update and read values from a corresponding row. During activation, appropriate row of memory is loaded to this array, and during write back this array is copied to corresponding row of memory.

3. I have created some helper values, like **currRow**, pointing to the row value that **rowBuffer** corresponds to in memory, **rowDelay** and **colDelay,** denoting ROW_ACCESS_DELAY and COLUMN_ACCESS_DELAY respectively, Boolean **updated**, that denotes whether memory is updated or not, a 2 element array **updates[]** whose first element denotes the number of times row activation has occurred and other denotes number of times row values are changed.

4. So, my basic approach is that when any instruction other than sw or lw is called, do corresponding functionality and increase clock by 1, but if lw or sw is called first find the address that is going to be needed for write or read, then find its position in memory, i.e. row and column values, then check if currRow is same as

row, if that is not the case then writeback the rowBuffer in memory and activate needed row (increasing clock by 2*rowdel) and then increase clock by colDelay and update rowBuffer or register otherwise just increase clock by colDelay and update rowBuffer or register.

5. **Corner cases:** When rowBuffer is activated for the first time, then no need to writeback. For this functionality I have used a Boolean **isFirst**. Also, when the code is executed completely there might be the case that memory is not updated with rowBuffer, to account for that I have used a Boolean, **updated**, if that remains false at the end of execution, update the memory.

Constraints

1. Address provided for lw and sw can only be multiple of 4(i.e., no address between an "word" is accessed).
2. If address provided more than 2^20, "Overflow error" will be raised.
3. If input is syntactically incorrect, same as assignment 3, Syntax error with appropriate help will be thrown.
4. All registers and memory are initialised with 0.
5. If lw or sw are encountered, they are first completely executed then further code is executed.

Test Cases

1. Firstly, I have checked for simple code without any lw and sw, to check if their functionality and clock is unchanged from assignment 3.
2. Then I checked for cases containing only lw where rowBuffer is updated at some point, to check if correct row is activated and clock value is incremented correctly and correct value is loaded in register.
3. Then I checked for cases containing only lw and where rowBuffer is not needed to be activated (same rowBuffer will suffice) at some point, to check if there is no unnecessary activation.
4. I then executed points 2 and 3 for cases containing only sw. Also checking right value is updated in memory.
5. Finally, I checked for cases when both sw and lw are present, for their co-functionality.
6. Further rigorous testing is done with part b explained further.

PART b:

Approach

1. For this case lw and sw can run simultaneously with other instructions, (well not specifically lw and sw, its DRAM memory access delay that can occur simultaneously). But this can give false results when registers values used in lw

and sw are used in the instruction running parallel to it, since then user who wrote the code expects that at this stage lw/sw is completed and register values are updated.

2. So, we need to find cases when lw/sw can run in parallel with other instructions. I have come up with the cases when to run parallel and when to wait for them to completely execute described in Constraints section.
3. For this case I've made a new class named DRAM, containing memory and clock and many helper values like Boolean **isOn** that denotes if there is some DRAM execution going on or not, **relClock** denoting relative number of clocks occurred while executing present lw/sw, **initialClock** denoting the absolute clock at which current execution is initialised, **address** denoting the address that is to be accesses in the execution, and other values same as part a for clock counting and corner cases.
4. So, my way of doing this is that whenever some instruction other than lw/sw are called, if DRAM is off (isOn = false) then do the operation increase the clock and if it's on and instruction to be done is dependent on DRAM execution then first execute the DRAM, increase clock value accordingly and then execute the instruction, otherwise execute the instruction straight away and increase both clock and relClock. After increasing relClock also check the status of DRAM.
5. Corner cases are same as part a, additionally one more case is added when execution of code is completed but still some execution is going on in DRAM, then do it and increase clock accordingly.

Constraints

Consider the following nomenclature for instructions:

Instruction a =

lw/sw	changeReg	Int(addReg)

Instruction b =

function_b	dest	source1	source2

struction c =

function_c	source1	source2	instruction

Where function_b is functions like "add", "addi", "sub" "mul" or "slt".

function_c is functions like "bne", "beq".

Suppose "a" is still pending in DRAM, and b is the current instruction that is to be checked if independent or dependent. I've considered following cases as dependent:

1. For lw:
    a. changeReg = dest: In this case user wants us to load some value in a register from memory first then through the function_b, hence we need to execute a first, then b can be executed so this is dependent.

    b. changeReg = source1/source2: This case is directly dependent because, first the value is loaded in the register then only we can execute the function otherwise the result of function won't be correct as we will be using old value of source register, not the updated one.
    c. addReg = dest: This case is also dependent because is we execute function first then addReg will be changed and lw will result in false output.
    d. Other cases of lw will be dependent since we are reading source1/source2 and addReg and if they are equal it does not matter if we read that register first for lw or that function, since we are bot changing value. Also, we none of the registers are same, then clearly, they are independent since no contradiction of values.
2. For sw:
    a. addReg = dest: This case is dependent since if we execute function_b first and then execute the sw, we will be using false value of address.
    b. changeReg = dest: This case is also dependent because of reason same as above that function will change the value of changeReg and we will be storing false value in memory using sw.
    c. All other cases are independent since we are only reading source1, source2, changeReg and addReg in their corresponding instructions and it does not matter which to read first.
3. If lw/sw is needed to be executed while some DRAM execution is going, first complete previous call then new DRAM call is initialised, (Assuming single port design and that DRAM can't initialise some instruction while it is performing some instruction)

Strengths

1. Clock value decreases than previous case because of parallel execution of some commands.
2. No need to waste clock time in DRAM delays, rather perform independent functions.
3. In my code there is no queue functionality (initiate DRAM for consecutive lw/sw in advance) that removes the space requirement for queue storage although some clock cycles have to be sacrificed for that.

Weaknesses

1. In my design I've assumed that DRAM cannot initialise some instruction (i.e., can't make queue of instruction) while some instruction is being executed, it first executed previous one then initialise and execute the next. It increases clock count by some numbers when there are simultaneous lw/sw commands.
2. I have also assumed single port design, that too constraints me from executing some instructions parallelly, increasing the clock count.
3. To perform executions parallelly I have to store a lot of values until it is being executed like relDelay, complete instruction, initialClock etc.

4. Also, we need to check every time any function is called inside the DRAM if some instruction is going on and also increase relDelay if yes that too increases complexity.

Test Cases

1. I first tested part b with all the test cases of part a to check if some basic functionality is not corrupted.
2. Then I tested with every possible combination of changeReg, addReg, source1, source2, dest (refer Constraints for nomenclature).
3. For rigorous testing I then ran both part a and part b on some big inputs to check if their output is same (only clock will differ).
4. This gave a conclusion that part b indeed take less cycles than part a for same program.