COL216: Assignment4

Instructions to run code :

1. After unzipping, two c++ files, interpreter2.0.cpp and ass4.cpp will be formed, interpreter2.0.cpp is the file of minor with changes that cannot be done in different file, and ass4.cpp is with changes that are done in different file.
2. Main function is in ass4.cpp so compile it through g++ compiler using the command "g++ ass4.cpp"
3. An executable a.exe will be formed, run it using instruction of format, "./a.exe testfile rowdel coldel" where testfile is the file that want to be executed, rowdel is ROW_ACCESS_DELAY, coldel is COLUMN_ACCESS_DELAY.
4. For testing purpose I've also allowed one more optional argument which when given 0 performs row ordering and save output in "new.txt" file and if provided anything other than 0 performs without row ordering and save output in "old.txt"
5. A python file will also be formed after unzipping named, checker.py that when given testcase and delays performs both with and without reordering and matches both the outputs and gives clock gain.
6. All the testcases are in folder ./Testcases/

Approach

1. For changing format of j,bne,beq to take label name I had to make following changes:
    a. Adding token type ":" in **lineToken** function, and a map **labelLine** in class **MIPS** that has instruction number for first instruction of each label, so that we can jump to that instruction whenever required.
    b. Made desired changes when bne,beq,j is called and their respective errors.
2. Now for adding the feature of effective execution order of instructions in DRAM, I hade to make changes in class DRAM. I firstly created a **queue** datatype to store all the instructions in queue of DRAM, along with its properties (like address register, change register, row number etc.) To do that I created a class **Waiter**, that contains all this info. Hence element of **queue** is of type (int, Waiter).
3. For the effective ordering I've created another map, named **rowSorts**. Its any element is of type (int, vector) where int is any row and vector contain key value of all such Waiters in queue with this row.
4. **MAIN IDEA:** So, the base of reordering is to execute all the independent instructions of same row consecutively, then new row is activated and all instructions with row are done.
5. To tackle for dependent case, I've used the following strategy:
    a. If current instruction is other than lw/sw then first find all the dependent instructions in DRAM row wise. Then perform complete row of all rows

whose instructions are dependent. And for last row, just initialise the next instruction in row after the dependent one and execute it parallel to the current instruction.

b. If current instruction is lw/sw then also first find all the dependent instructions in DRAM row wise. If there is no instruction of row other than one that is current active, then simply add lw/sw in queue (since it will execute after all the instructions before this in current row are done hence no worries.) else same as previous case perform complete row of all rows whose instructions are dependent. And for last row, just initialise the next instruction in row after the dependent one and execute it parallel to the current instruction.

c. For executing this I've declared a bunch of functions, **bounds** that gives max and min position of instruction in queue, **initWaiter** that add instruction in queue, **start** that starts some instruction in queue, **currIsDep** and **isDep** that check if instruction is dependent with the instruction going on in DRAM and provided Waiter respectively, **depInRow** that gives last dependent instruction in any row, **doRow** that performs all instructions in the row till some bound, **doLS** and **doLSnot** for points b and a respectively.

NOTE: *All the changes made in the same file as minor are enclosed under comments (//-----ass4-addition----[code] //----------) hence can be found by searching string ass4 in the file interpreter2.0.cpp.*

Constraints

1. For the dependence of instructions other than lw/sw the same constraint as of minor is followed.
2. For the dependence check for instruction lw/sw:

   Consider the following nomenclature for instructions:

Instruction in DRAM (s) =	lw/sw	cs	Offset(as)

Current Instruction (i) =	lw/sw	c1	Offset(a1)

   I've considered following cases as dependent:

3. For s = lw:
   a. For i = lw:
      i. **c1 = cs:** This case is independent, in fact the instruction in DRAM becomes insignificant over the current instruction, hence for this case I've removed s from the queue and added i.

ii. **c1 = as:** This case is dependent because if we perform i first then value as will change for s and will give wrong output.
iii. **a1 = cs:** =This is also dependent case because of above reason just role of s and i reverse.
iv. **a1 = as:** This is independent case since it does not matter in which order we read value of some register.

b. For i = sw:
   i. **c1 = cs:** This case is dependent since we need to first load some value in cs then save it in memory.
   ii. **c1 = as:** This case is independent since both c1 and as values are read and it does not matter in which order we read.
   iii. **a1 = cs:** =This is dependent case since first we need to set cs to some value (desired memory) then only we can save.
   iv. **a1 = as:** This can be dependent when offset is also same, since then we have to save value in that memory after reading from it, not before.

4. For s = sw:
   a. For i = lw:
      i. **c1 = cs:** This case is dependent since we need to first save value of cs in memory then load some value in it.
      ii. **c1 = as:** This case is also dependent case since if ordering changes then value of as will change causing in wrong memory address.
      iii. **a1 = cs:** =This is independent case since both cs and a1 are read and order for that does not matter.
      iv. **a1 = as:** This is dependent case with same reason as part iv of s = lw and i = sw.
   b. For i = sw:
      All the cases here are independent since we are just reading some register value and storing it in memory and for that order does not mater, moreover when address of s and i are same then we can just ignore s.

Strengths

1. Clock value decreases than previous case because of effective ordering of commands.
2. Decrease in number of buffer updates.
3. Parallel execution of DRAM decreasing clock wastage in DRAM delays.

Weaknesses

1. To maintain queue, I've to store all the instructions in queue and for parallel execution I have to store a lot of values until it is being executed like relDelay, complete instruction, initialClock etc, increasing space complexity.

2. For better ordering I've to also store row wise instruction number that also space.
3. Also, we need to check every time any function is called inside the DRAM if some instruction is going on and check for independence, that increases time.
4. There might be some other ordering that decreases clock further.

Test Cases

1. I have considered the output of minor as my baseline to check output.
2. Checker.py runs both the baseline and the code and compares them.
3. I first tested for all the independent and dependent cases with some short testcases.
4. Then I tested with all the testcases of minor.
5. Then I made some big random testcases and checked for it.
6. Output matches for all of them, with a significant gain in clock 9more than 50% sometimes).