COL 216: Computer Architecture

Assignment 4

Preface:

In this assignment we had to optimize our DRAM access methods. Initially what we did was we performed DRAM requests sequentially which was time-consuming due to repeated row buffer writeback. I have build on my existing work in the minor exam and further added the features to achieve this optimization.

Our Approach:

In the Minor implementation I had maintained a single **queue** which would keep track of the DRAM requests. This method did provide some optimization over the traditional approach but not enough.

For this part I maintained an **array of 1024 queues** denoting the access to each row. Now what I do is as soon as I get a request I enqueue it into the appropriate queue according to the row which they are accessing. This gives us the added advantage as all the requests accessing the same row would have just column access delay as the time.

Now whenever a dependency occur I first empty the **queue of the row** which is presently **active**. If the dependency is removed well and good if not we check which row gives us the dependency and the empty that queue.

For implementing such a structure I have used and array of vector of size 1024 and a map giving me the queue no where a particular task is.

Strength of Approach:

One of the strengths of our approach is that it is simple and is quite effective. It reduces the clock cycles considerably by maintaining simple structures. There is no chance of mismatch between the expected value and the value we get. It doesn't empty the whole DRAM just by seeing a dependency it only empties the dependent row hence this gives us more advantage as some requests can be further done in parallel.

This approach is more structured and there is no lookahead or other such things which would require complex dependency check. Here the check for the dependency is pretty straightforward and easy.

Weakness of Approach:

There are few weakness to our approach which hinders further optimization of the timing. One of the design choice is that after completing the active row queue the next row queue on which the operations to be done are chosen via their position in the array. Meaning lower row no. queue gets the nod before the higher. So now suppose in some case the

dependency occurs at a later row then first the active row would be completed then the the dependency row. This could lead to a loss of time which could be optimized.

Another weakness due to our design choice is that we are emptying the whole row at a time rather than the dependent registers only. This takes extra time on the one hand but although it does save us a lot of hassle of tracking the switch between rows.

Constraints:

1. Initially all the registered are set to 0.
2. Memory values used will be multiple of 4 only, no intermediate memory change available.
3. No direct integer addition using “add”, you have to provide the register values whose corresponding values are to be added, well, “addi” is specifically for that purpose only use it!!
4. Since all the instructions provided were integer operations, float registered are not handled.
5. Maximum memory size is 2^20, any value more than that will cause “Overflow Error”.

Error Handling:

For every command we have type matched them the line tokens with the format that it is supposed to be and if there is some type mismatch then we ended the program with appropriate error line. Also if type is correct but the execution is not possible or meaningful like jumping to a line out of scope of program or reading or writing the value in memory that is more than the range, then also we have thrown appropriate error.

We have tried our code to be as complete as possible, that can handle every possible combination of syntactically correct or incorrect commands.

Moreover, we have also taken into account the ignoring of comments.

Output:

First line by line we have printed what is done per cycle. This includes the operation performed, register value change, memory address value update etc.

After this in the next line we have printed the total clock cycles taken and after that no of changes in the buffer.

Testing Strategy:

We have taken a lot of test cases to check our code and then manually calculated the cycles it must take to evaluate and then compared it with the output of our code.

Some of the testcases evaluated are in Testcase folder of our submission.