COL 216: Computer Architecture

Minor

Preface:

In this assignment we had to build on our existing model of the MIPS simulator adding DRAM timing model. In first part we had to implement the DRAM and its timing and in second part building on that we had to optimize our operation by the concept on non-blocking-Memory access.

Our Approach:

In part 1 we used a **2-D array** of int (4 bytes) of 1024*256 for implementing a DRAM of 1024*1024 bytes. We used an array of int of size 256 for the row buffer and a counter which will keep the count of no of changes to row buffer. We created the **functions sw and lw** which handle all the cases of the DRAM and update the clock and relevant values according to our timing model.

In part2 we had to implement the non-blocking access memory model. For this we used a **queue**(using vector) which would give **DRAM request** and pile up the task and the DRAM request are **sequentially** performed. Whenever we see a **dependency** we keep dequeuing the tasks until the dependency is removed. If dependency is not found we keep executing the **DRAM request in parallel** to our existing request. Whenever we see a **lw** and **sw** we **enqueue** the request of the DRAM.

Strength of Approach:

One of the strength of our approach is that its simple and is quite effective. It reduces the clock cycles considerably by doing process in the background just by implementing a simple queuing strategy. As soon as seeing the first dependency we dequeue all the operation in the queue. This approach doesn't pose risk of mismatch as we have considered very strict definition of dependency hence conflicting in values is impossible.

The flow is always sequential in our approach which is also one of the strength of our approach. This also provide a way out of conflicting results.

Weakness of Approach:

The strict dependency conditions also mean that we lose the opportunity to optimize the timing more. It is possible that the instructions next to the dependent instruction has no dependency which would mean we could in parallel execute it but in our approach we would not be able to do it.

Suppose we load from a register to memory and then perform computation using that register. This should be a dependency but out approach considers it as dependent.

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 the 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.