COL216 Minor

Sayam Sethi

21 March 2021

1 Assumptions

The following hardware assumptions were made in the final design of the **MIPS Interpreter**:

1. There are **three ports** for communication with the processor:
   (a) A **bidirectional port** with the **registers** (the call is issued at the beginning of cycle and result obtained within the same cycle).
   (b) An **output port** to send the DRAM request to the **DRAM queue** (explained in detail in the next point).
   (c) An **input port** to receive the result of the DRAM request (with corresponding register updates, if any) after delay.
2. The **DRAM architecture** is as follows:
   (a) It contains a **2D square array** of 1024 rows with each row having 1024 bytes (256 words).
   (b) It contains another storage location called the **Row buffer** which is used to store the current memory retrieval row.
   (c) To enhance the non-blocking nature of the DRAM, a **queue** is also present which contains all the required information about the pending load/store instructions.
3. There is a **common access counter** which keeps a track of number of pending updates for each register.

(Points 2(c) and 3 were assumed only for the second design choice of Part 2)

2 Implementation

2.1 Part 1

The implementation in Part 1 was a simple extension of the code for Assignment 3. The only instructions that had to be modified were those of *lw* and *sw*. A *delay* variable was introduced to keep a track of number of cycles taken for each instruction.

2.2 Part 2

2.2.1 Design Choice 1

In this design, parallel execution of DRAM retrieval and register calculations is carried out. In accordance with sequential execution of the instructions, the following design decisions were made:

1. The initialisation of DRAM request was done in the sequential order of the instructions.

2. After a request has been sent, following instructions may be executed if they are "safe".

"Safe" instructions here refer to those register-only instructions which don't use the register to which data is being loaded by the DRAM. Consequently, if a store instruction is being executed, all register-only instructions are "safe".
Note: This design was implemented in an earlier commit which was then modified to implement Design 2.

2.2.2 Design Choice 2

This design is architecturally more complex as it requires a queue implementation and a few checks before deciding between execution of the instruction or waiting. The following modifications were made with respect to the previous implementation:

1. Instead of a single instruction being executed and stored at a time by the DRAM, a queue was introduced to store the list of pending instructions.

2. A counter was used to keep a track of the number of times a particular register had pending updates.

3. For each command to be executed, the counter of the required registers was first checked. If none of them had pending updates, the command was executed, else execution was paused until the next DRAM deque, when this check was repeated.

4. The queue contains information about the register (for load), the value (for store), the address. For additional debugging and software implementation, the queue also contains the delay cost of that instruction, the cycle number when the instruction actually began to execute and the PC address of the command (this would not be required for a physical implementation since the abstract concept of clock cycles in the interpreter is converted to actual hardware).

Note: For simplification of the interpreter code, an actual row buffer isn't maintained but the row number for the instruction is stored in the queue (and delay is calculated beforehand). To implement the same, another variable of the actual buffer row along with the buffer data wound be required.

3 Strengths and Weaknesses

3.1 Strengths

The design used in second design choice is relatively complex, however, it helps in saving the waiting cycles in most cases. The completion of store instructions require no waiting whatsoever, although they might have to be completed so that another pending load instruction is executed which is required by some register.

This also ensures the execution of every single safe instruction possible before a dependent load instruction execution (if any) is completed for an unsafe instruction. Another possible tweak to this is to simply ignore all loads called on the same register except for the last one (however, this redundancy is induced by the programmer and hence it might not be a good idea to perform this optimisation since such code might be intentional. Hence, this implementation in the design was omitted).

3.2 Weaknesses

The design is pretty intensive and requires multiple checks. This will translate to a comparatively complicated hardware and might possible require a higher value of instructions per cycle to actually perform the required instructions. This might not be possible for simpler or smaller hardware and the best possible optimisation in such cases would be choice 1 (which has the disadvantage of not making full use of the waiting cycles).

4 Testing Strategy

Testing was pretty simple to be carried out for Part 1, since the order of instruction is irrelevant to the implementation. Testing with the same cases used for Part 2 was carried out.

For Part 2, the testing was carried keeping the following cases in mind:

1. Queuing up of DRAM instructions while safe instructions are parallelly executed.
2. Queuing up of instructions along with an unsafe instruction requiring a pending instruction to be completed.
3. Multiple load/store instructions involving the same register so that the queue handling is tested effectively.
4. Tested on large cases which were give for Assignment 3 and Minor which had some other combinations of cases that were not handled above (like a combination of safe and unsafe load instructions).