COL216 A-4 Write-up

Sayam Sethi 2019CS10399
Mallika Prabhakar 2019CS50440

April 2021

1 Assumptions

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

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, and accept or reject 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 implement non-blocking and reordering of the DRAM, a **queue** is also present which contains all the required information about the pending load/store instructions. It is used to execute the DRAM instructions and also determine which instruction to execute next (or ignore). It can store atmost 32 instructions at a time.
3. There is a **common access {cycle, address} pair** which keeps a track of the last instruction which loads a value into each register. It can be accessed by both the processor and the DRAM.
4. A **pair of {address, value}** which stores the value of the DRAM location last interacted with (stored to/ loaded from).

2 Changes and Additions to Minor Code

1. A new data type $QElem$ was created to contain all the necessary details for instructions
2. Variable changes: $registersBuffer$ changed to $registersAddrDRAM$, $lastAddr$ added
3. Instead of having the $updateRegisterBuffer$ function, we created functions $finishCurrDRAM$ and $setNextDRAM$ along with the helper function $popAndUpdate$ for the same purpose because of the change of queue into an unordered_map of unordered_map of queue for our modified implementation.
4. Following functions were modified: $addi$, slt, op, bOP, lw, sw, $locateAddress$, $executeCommands$, $bufferUpdate$, $printDRAMCompletion$, $printCycleExecution$, $initVars$ because of the same reason as stated above.
5. Remaining functions, namely- $MIPS\_Architecture$ (constructor), add, sub, mul, beq, bne, j, $checkLabel$, $checkRegister$, $checkRegisters$, $parseCommand$, $constructCommands$, $printRegisters$, $handleExit$, $main$ remained the same.

3 Implementation

The idea to use a queue for every row and column arose from the fact that every lw instruction requires the previous stores to that location be completed. The queue helped in ensuring this dependency. Then, to skip the redundant load instructions, the last address and the cycle in which the request has been issued has been stored which uniquely determines the instruction.

The information pushed in the queue is the instruction type, value (register number for load, register value for store), and the cycle in which the request was issued. Other information stored is not required by the physical hardware, but by the simulator to print relevant information. This requires $1 + 32 + 32$ bits of memory. Thus, the complete queue requires $9 \cdot 32$ bytes of memory in the case a single bit cannot be stored separately.

The idea for execution of the instruction is the same as that used in the minor code. However, popping of the queue and setting the next instruction is changed to incorporate reordering and priority selection.

In the case of no priority, i.e., when there is no dependent instruction which requires immediate execution, an instruction of the same column is attempted to be executed. If the queue of that column is empty, then an instruction of the same row is attempted to be executed. If the row is empty too, then the first element of the unordered_map is chosen. In the case of priority, the required {row, column} is chosen.

Before choosing the next instruction, in the case of a load instruction, the {last address, cycle number} is matched with the corresponding register. If it doesn't match, then that element is simply popped from the queue without executing it.

Additionally, to save clock cycles, forwarding is implemented by storing the {address, value} of the last interaction with DRAM. Also, rejection of DRAM requests is introduced which helps in execution of "unsafe" instructions updating a register having an ongoing DRAM load and marking that instruction as safe.

4 Strengths and Weaknesses

4.1 Strengths

The reordering of instructions is decided in such a way that it reduces the number of times the row buffer is written back and updated with a new buffer as much as possible.

The design choice implements reordering (and skipping) of DRAM instructions with the least amount of additional data storage required. The heaviest part of the algorithm is the selection of the next instruction, which can happen in the same clock cycle in which a previous instruction is completed.

The skipping of instructions helps in avoiding a lot of redundant loads which reduce the number of cycles taken to execute the code by a lot. Forwarding of instructions ensures that if the user is working with a particular location in the memory, the load time is greatly reduced.

The maximum size of the **queue** is set as 32 since this allows for maximum data capacity without having the need to make the DRAM access blocking (on reaching maximum size). Any larger would lead to unused memory for most of the times and any smaller would lead to frequent blocking.

4.2 Weaknesses

The design is kept as simple as possible to reduce the hardware complexity of the processor. This led to the pushing of every store instruction to the **queue** and the same leads to some redundant store operations. Removing this redundancy would lead to using a data structure whose size would be comparable to the data memory, i.e. 2^{20} bytes, hence the inefficiency of *sw*.

The retrieval time for the next most optimal instruction has been kept as small as possible which has lead to the chance of selection of a slightly sub-optimal instruction at times (this is governed by the hash values of the row/column). To enhance the quality of selection, order of the arrival of instructions would need to be maintained which would lead to a slower selection (spanning over multiple clock cycles) and would require memory comparable to the size of the instructions (which, in the worst case can be equal to the size of the data memory, i.e. 2^{20} bytes).

5 Testing Strategy

Testing was carried keeping the following cases in mind:

1. Queuing up of DRAM instructions while safe instructions are executed in parallel.
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.

Test Cases:

1. **lw.asm -** single load word operation
2. **sw.asm -** single store word operation
3. **lwLocate.asm -** multiple load words at same location
4. **swLocate.asm -** multiple store words at the same location
5. **size.asm -** a large number of commands to show max size of data structure
6. **unsafeAddress -** testcase with address register unsafe
7. **random.asm -** initial testing of overwrite
8. **random2.asm -** rejected DRAM testing
9. **random3.asm -** a random testcase
10. **rand.asm -** to test reordering