COL 216 Assignment 5: MIPS instructions interpreter

Ramneet Singh | Mustafa Chasmai
2019CS50445 | 2019CS10341

July 7, 2021

Design Characteristics:

1. Instruction buffer used by the MRM is of a fixed size. Also, most scheduling operations can be parallelized. These two simple observations allow a hardware implementation such that the MRM can perform scheduling in one clock cycle itself.

2. Memory is divided between the cores and each CPU core can only access/modify a part of the total memory. The instruction buffer in the MRM has also been divided into separate sections, so that all the cores can write instructions into the buffer in parallel. This allows the MRM to service all CPU cores in the same clock cycle.

3. We have used aggressive reordering, where row hits are given the highest priority. For ordering between multiple row hits and row misses, we have used the FR-FCFS scheduling policy, first introduced in [Rixner00], where the oldest instruction in the buffer is chosen. Since sw-lw forwarding leads to a reduction of memory access, instructions using forwarded values are given a higher priority over row-hits.

Tradeoffs:

1. **Instruction Buffer Size**
   A larger instruction buffer will lead to better reordering decisions. However, it will lead to a higher hardware cost and higher latency.
   We have currently taken a buffer size (per core) of 32 instructions in the buffer.

Approach of Solution:

In the current assignment, the main scheduling (reordering) is done by the MRM, while the DRAM only services requests sent by the MRM. The MRM maintains a buffer of pending memory-access instructions and decides which instruction should be serviced next by the DRAM.

Memory Request Manager

For every instruction that requires memory access, we create an instruction object having its target, address, id and an array of dependencies. In this dependencies array, the locations corresponding to other instructions are set to 1 or 0 depending on whether they need to be executed

before the current instruction can be considered safe to run. Every time we add a new instruction to our MRM buffer, we first check for conflicts and update the dependent instructions in the new instruction's dependencies. Again, when an instruction gets executed, we iterate over all pending instructions and update their dependencies. An instruction is safe if and only if its dependencies array has all zeroes.

While scheduling the next instruction for DRAM to service, the MRM uses an aggressive re-ordering algorithm, where all instructions with row-hits are executed even if the buffer has older instructions. For ordering between multiple row hits and row misses, we have used the FR-FCFS scheduling policy, first introduced in [Rixner00], where the oldest instruction in the buffer is chosen.

We have implemented sw-lw forwarding. As this forwarding leads to an elimination of memory access in the lw instruction, forwarding is given the highest priority. The MRM first checks for instructions with forwarded values, and checks for row hits only if no such instruction is found.

Multi-core Functionality

For processing multiple cores in parallel, we initialise multiple interpreter objects, one for each program file given as input. We initialise a single shared MRM that is used by all the interpreter objects. The interpreter objects themselves do not have complete access to the DRAM, and have to interface with the MRM for any memory access. Note that the instruction memory is a separate memory space, that is not shared across multiple cpu cores (interpreter objects).

Although the DRAM memory is shared across the cpu cores, each core is statically assigned its own memory space and any core trying to access memory outside of this allocated space is flagged with an error. The registers are also maintained separately.

The instruction buffer in the MRM is also partitioned, with different cores adding instructions to different locations in the buffer. This partitioning policy allows us to parallelize the process of adding an instruction into the buffer. While executing the instruction, all the cores can add their corresponding instructions to the row buffer in the same clock cycle. This further optimises the MRM.

Hardware Implementation

We have implemented the MRM to most efficiently utilise the parallelism offered by the use of multiple cores. The algorithm used for scheduling is intelligent enough to give optimsed performance, but is also simple enough to be easily implementable in hardware and to be completed in one clock cycle.

For each instructions in the buffer, we first have its index. As the buffer size is fixed (currently 32), the index is stored in a fixed number of bits (currently 5). Then each of the dependencies is

stored as a bit-mapped number (again from 0-31, so 5-bit). While adding an instruction, a combinational circuit checks for conflicts with all the instructions in parallel. Since the instructions in the buffer are independent of each other, operating on all of them parallely will not lead to any errors. Again, when an instruction is removed from the buffer, the bit corresponding to its index is simple set to 0 for all the other instructions. In addition to these bits, we have a valid/invalid bit such that if it is set to 0, then the instruction is actually empty (the MRM does not use such instructions in its processing).

Other than these, each of the instruction has with it some priority bits stored. The priorities of different scheduling parameters (eg. row hit, time) are encoded in accordance with their relative order of preference. Since forwarding has the highest preference, the most significant bit stores whether the instruction is being forwarded. After this, the next bit stores whether the instruction is a row hit or a row miss, and finally the last few significant bits store the temporal order in which the instructions were added to the buffer. The priority bits are then sent to a priority encoder, and this priority encoder selects which instruction is to be sent to DRAM next.

The instruction buffer is partitioned so that each core has a separate space where they can add instructions. In hardware, this can be implemented with n (maximum allowed cpu cores) separate write lines (busses), and each line is granted access only some portion of the buffer. As multiple cores cannot write to the same buffer index, there will be no write conflicts. This allows us to parallelize the process of all the cores adding their instructions into the MRM buffer.

MRM Delay estimation

As discussed in the previous section, most of the processing required for scheduling in the MRM occur in parallel. As the size of the buffer is fixed, a fixed circuit can be implemented that processes all the instructions at the same time. We can implement a priority encoder which has a depth bounded by $log_2 n$, where n is the number of entries it compares. Since our buffer size is capped at 32, this means a critical path of maximum 5 MUX gates. Assuming 5 MUX gates can run in one cycle, this entire operation can be completed in one clock cycle.Thus, the entire scheduling takes a single clock cycle.

Again, the buffer is partitioned across the cores. So all cores can write to the instruction buffer simultaneously. Thus, the writing to buffer, i.e. interface between cores and the MRM, takes another clock cycle.

During the clock cycle in which MRM performs scheduling, it also sends to the DRAM, the relevant data about which instruction is to be executed next. The DRAM receives this instruction in the edge after the same clock cycle in which the MRM was performing scheduling, and the DRAM can start performing its activities (writeback, copy to buffer etc) from the subsequent clock cycle.

Thus, the MRM effectively requires only two clock cycles. After the end of the second clock cycle, (i.e. in the third clock cycle), the DRAM starts performing the required activities corresponding to the instruction it received from the MRM.

Testing strategy:

The test cases are written as separate files inside relevant folders inside the main directory 'test_cases' included in the submission. The testing has been divided into the following sub-categories and test cases of each have been checked to ensure correct working of our interpreter.

1. **Reordering**

   Here, we consider test cases where aggressively reordering significantly improves performance over the first-come first-serve scheduling policy.

   - We consider one testcase which is a long stream of load word instructions. We believe this emulates a scenario where a program copies 2 different arrays but only uses one loop for that. Both those arrays occupy different rows of the DRAM. Without an aggressive reordering policy, this would lead to 3 writebacks every iteration, since we have a stream of load word instructions with rows alternating.
   - The second testcase has a loop, in which we have 2 load word instructions to different rows, and a store word instruction as well to a different row. Since these are large numbers of memory requests. but they go to different rows, not reordering them would cost a lot of cycles. This case specifically also shows that the handling of writes is integrated with the handling of reads, and works fine. It also contains a mechanism to check that the value being stored in every iteration is the correct one, to check that reordering doesn't change the semantics of the program.

2. **SW - LW Forwarding**

   This testcase shows the significant improvement on performance that sw-lw forwarding has. The testcase consists of a loop, and in every iteration we have a lw instruction, on whose value the next addi instruction is dependent. We also have a sw instruction at the end of each iteration, to the same address as the lw at the beginning of the next iteration.

   This can be a very typical case for programs, because in we are just fetching a value from memory, performing some computations with it, and then storing the result to memory. Since we have implemented sw-lw forwarding, the lw instruction at the beginning doesn't have to wait for the sw instruction before it to finish. This not only reduces the number of cycles the DRAM is working for (also decreasing power consumption), but also, this crucially prevents the next addi from being stalled for a long time. This enables the processor also to keep moving and thus exploits the ability to run DRAM and processor in parallel.

3. **Syntax errors**

   We had made the interpreter robust towards syntax errors in the input MIPS program. The syntax errors are flagged and appropriate error messages are displayed on the terminal. We have handled the following syntax errors:

   (a) Unrecognized command.

   (b) Unrecognized registers.

   (c) Less or more arguments to an instruction (flagged when comma missing also).

(d) Editing the $zero register (flagged only if value is being stored at $zero).

(e) Data load or store instructions accessing instruction memory and branch instructions accessing data memory. Un-aligned words (invalid for byte-addressable) being accessed are flagged too.

(f) Finally, integer overflows during addition and subtraction are flagged.

The interpreter has been extensively tested with the specifications assumed with our design choices. We have included the outputs displayed on the terminal while running each program as files with name 'output.txt' in the same folder as the program. We have written a shell script to make it easier to run our program on an entire folder of testcases and store each output in an appropriate file.