COL216 Assignment 5

Devanshi Khatsuriya 2019CS10344 | Ishita Chaudhary 2019CS10360

1 Objectives

1. To extend the earlier MIPS simulator (with DRAM timings) to the Multicore CPU scenario, where each core runs a different MIPS program and sends DRAM requests to a Memory Request Manager (MRM) which interfaces with the DRAM.
2. To maximize the instruction throughput (total number of instructions completed by the whole system in a given period, say from Cycle 0 to Cycle M).
3. To estimate the delay (in clock cycles) of the designed Memory Request Manager and incorporate it into the Timing Model.

2 Approach

2.1 Multicore Implementation

- We have assumed the total number of cores available to be 32. Each core has its own register file (size = 32 registers), instruction memory (size = 2^9 words) and program counter.
- Data memory from the DRAM is allocated to each core before execution. This is done by uniformly dividing the number of rows present in the DRAM (1024) among the number of cores that are being used and storing the base address (offset) for each core.
- Each core has 32 sized files like $is\_being\_written, is\_being\_stored, store\_valid$ and $store\_address$ which tell whether each of the 32 main registers are being written to from DRAM or are being stored in DRAM and which address (to facilitate dependency determination and lw-sw forwarding).

2.2 Memory Request Manager

- The Memory Request Manager (MRM) receives DRAM requests from all the cores, from which it chooses one at a time to send DRAM.
- The MRM has two files of size 32: a wait buffer and a second wait queue, which it uses to store DRAM requests. The cores are allowed to send their DRAM requests to the first wait buffer (atmost one core in one cycle, others if present, are stalled for that cycle). The second wait queue contains some requests that have the same row as the current Buffer Row and is maintained by the logic given below.
- The logic used for choosing the request to send is as follows:
  1. The rows of all the requests present in the wait buffer are matched with the index of the row currently in the DRAM Buffer Row. Those requests that have the same row as the Buffer Row are moved to the second wait queue.
  2. Now, if the second wait queue is non empty, its first request is chosen to be sent to DRAM. The first step is also skipped whenever the second wait queue is non empty.
  3. Otherwise, if the second wait queue was already empty and is still empty after Step 1, it means there is no request from the current row in the Buffer Row, and in this case the first request from the wait queue is sent to DRAM.

- The MRM sometimes also needs to delete a lw (DRAM read request) from the wait buffer or the second wait queue if an overwrite to that lw is detected. This is the case when a previous lw is loading into a register, and we encounter another instruction that wants to write into the same register. Here the previous lw can be safely dropped without execution.

2.3 Delay Estimation for Memory Request Manager

- The Memory Request Manager (MRM) utilizes cycles in the following ways:

  1. When the DRAM has finished executing its current request, the MRM takes 1 cycle to choose the next request to send to the DRAM as follows:
     (a) If the second wait queue is non empty, its first request can be sent to DRAM and can be subsequently deleted from the wait queue in 1 clock cycle.
     (b) If the second wait queue is empty, there is a need to check if the wait buffer for requests that have the same row as the present Buffer Row. For this, we propose to have hardware made up of suitable gates and multiplexers connected between each of the 32 blocks of the wait buffer and second wait queue.
     (c) Using the hardware mentioned above, we can select the first request in the wait buffer that has the same row as as the present Buffer Row, and if such a request does not exist, we can also have multiplexers to choose the first request of the wait buffer in that case. All this can be done in 1 cycle.
  2. When the MRM needs to delete a DRAM request (for an overwritten lw as mentioned in the MRM section), it can do so in 1 cycle using hardware which will compare the relevant parts of all the requests in buffers at once and then remove the required one.
  3. In all other cases, i.e. when the wait buffer and the second wait queue are empty and there is no overwritten lw (DRAM read request) to be deleted, the MRM does not take any cycles.

- When the MRM is making a decision (1(a)), the DRAM stays vacant waiting for the next request, so it causes 1 cycle . When the MRM has to delete a DRAM request for an overwritten lw (1(b)), the DRAM can run independently, so this deletion does not contribute to delay caused by MRM.

- Also, when the decision making and deletion is to be done in the same cycle, only 1 cycle is required.

2.4 Throughput Calculation

The Throughput is calculated using the following formula:

$$Throughput = \frac{Number\ of\ instructions\ executed\ completely}{Number\ of\ cycles\ taken}$$

$$Number\ of\ cycles\ taken = Minimum\ (M,\ Number\ of\ cycles\ taken\ for\ execution\ to\ complete)$$

Note: When we delete the DRAM request for an lw to which an overwrite has been detected (as mentioned in MRM section), we still count this lw in the number of instructions executed completely because it has been semantically executed.

2.5 Optimizations Implemented for Throughput Efficiency

To maximize the Throughput when the number of cycles for execution are fixed (M), we need to increase the number of instructions that are completely executed. Since the number of cycles is fixed, to increase number of instructions, we need to decrease the cycles taken by each instruction and also delay caused by MRM.

- If there is any pending request that accesses the same row as the current Buffer Row, then that request is chosen to be sent. This minimizes the number of buffer row write backs required drastically which in turn saves cycles and increases Throughput.

- We have implemented lw-sw forwarding, which means that if a register is currently being stored to some location in the DRAM and a load from the same location is encountered, we do not send a DRAM request for that lw but instead complete the execution of that lw by passing it the data from that register. This saves cycles as DRAM accesses are very costly comparatively as they take multiple cycles.
- We have implemented functionality to remove overwritten lw instructions. So, when a load is pending on a register and another instruction attempts to write to the same register, the previous load is considered overwritten and is dropped. This saves the cycles for DRAM accesses and also saves those cycles for which the processor would be stalled due to encountering a dependent instruction.
- MRM can delete overwritten lw (as mentioned in the MRM section) in parallel with DRAM execution so that delay caused by MRM is minimized which in turn saves cycles and increases Throughput.

2.6 Some Necessary Stalls Implemented

- Only either MRM or one of the cores can access the wait buffer in one cycle as it is a shared resource. So, if MRM is making decision or is deleting an overwritten lw, all cores that encounter a lw or sw instruction are stalled for that cycle. Also, if one of the cores is sending a DRAM request, others that also want to send a DRAM request are stalled for that cycle.
- In the last cycle of Column Access for a DRAM read request, the read value is written to a register, so the write port is busy. So, we that core's instructions that may write in that one cycle need to be stalled (add, addi, sub, mul, slt and lw).
- When the wait buffer is full, the cores that want to send DRAM request need to be stalled.
- A core is also stalled when dependent instruction is encountered. (dependent instruction is one has to read from a register that has a pending load).

3 Strengths and Weaknesses

3.1 Strengths

- The major strengths are all the optimizations mentioned in Section 2.5 which improve Throughput considerably while maintaining the semantics of execution. They decrease the number of cycles instructions take (by lw-sw forwarding) and the number of stalls due to dependent instructions (by deleting overwritten lw requests).
- We have considered the feasibility of implementation of the code and algorithms as Hardware. So, we have not used many attractive features found in high level languages like C++ to keep the implementation simple and feasible.

3.2 Weaknesses

- Stalls occur very frequently while execution, which can be considered as a weakness due to increased amount of cycles required and decreased Throughput. However, the stalls are necessary due to the presence of shared resources as mentioned in Section 2.6.
- When there are multiple cores that want to access the wait buffer, we have just given priority to the core that occurs first and have stalled others. This leads to starvation of resources from the other cores. We could have prioritized a core depending on the DRAM request it wants to send so as to improve Throughput in the larger scale.
- In the duration for which MRM is taking its decision, the DRAM stays vacant, i.e. is not processing any request. This could be prevented by making use of the time when DRAM is running to make the MRM decision, like how MRM makes deletions in parallel with the DRAM.

- We have not implemented functionality to handle sw-sw overwrites to decrease cycles taken. This is the case when an sw request is pending and another sw is encountered which overwrites what the first the first sw writes in memory, in which case it is safe to delete the previous sw.
- The lw-sw forwarding mentioned in Section 2.5 cannot take place when the register which is being stored is changed by another instruction before the lw is encountered. The lw-overwrite handling also mentioned in Section 2.5 can occur only when the request to be deleted has not started execution. This could possibly be improved by stopping its execution if it has started but we have not done so considering the feasibility of such a design.

4 Testing

4.1 Running the Program

Open a terminal in the directory and type in the command

make

to compile. To run the program on N MIPS program files, name the files "t1.txt", "t2.txt", ... "tN.txt". Then use the command

./out N M RAD CAD

where N is the number of cores/files to be used, M is the number of cycles for which execution is to be done, and RAD and CAD are Row Access Delay and Column Access Delay values respectively.

4.2 Test Cases

To run a test case, copy the MIPS text files from that test case's folder into the same directory as the executable and run with the appropriate command as mentioned above.

We have identified some major cases to be handled correctly by our code which are mentioned below. We have tested these for various values of N, M, RAD and CAD.

1) **Test Case 1: Dependency**

   This test case checks the various cases of dependent instructions. This involves read attempts and sometimes write attempts to registers that have pending load requests on them. We expect that when a dependent instruction is encountered, if the present instruction is only overwriting the pending lw and it is possible to delete the pending lw (i.e. if it has not reached execution stage yet), stalling does not occur otherwise proper stalling occurs.

   File t1.txt has dependency that can be resolved by deleting overwritten lw. Files t4.txt and t3.txt has dependent instructions that require stalling. File t2.txt has both of these in a different form.

2) **Test Case 2: Row Reordering, Starvation**

   In this case, we check that proper reordering of DRAM requests occurs and that requests with the same row as the current buffer row get priority. We also observe that starvation occurs, as for as long as the first core wants to send a DRAM request, the other cores are stalled.

3) **Test Case 3: MRM Deciding, Multiple Core Access Stall**

   In this test case, we mainly check that MRM and cores are not accessing the wait buffer at the same time since it is a shared resource. Whenever MRM is accessing the wait buffer to make decision, cores that are sending lw or sw requests are stalled. Also multiple cores can't access the wait buffer at the same time, so when one of the cores is sending a lw or sw request, other cores that do the same need to be stalled. The same has been checked.

4) **Test Case 4: Forwarding**

Files t1.txt to t4.txt contains various cases of forwarding namely - sw directly followed by lw instruction that will forward it, sw directly followed by other instructions followed by lw instruction to be forwarded and the register being stored modified after sw so that forwarding lw is not possible. The output observed for these cases are correct.

5) **Test Case 5: Redundant lw instructions**

In this test case, we show how the redundant lw instructions are deleted from the queue of instructions. The redundancy may have the form of multiple lw loading into the same register from different addresses of memory or a load word instruction is followed by an add/ addi/ sub/ mul /slt instruction writing to the same register. In the last case, if the register's data is read for the execution of add/ addi/ sub/ mul/ slt instruction, then the corresponding lw instruction is not redundant.

It is an important point to note, that the first lw instruction of Core 2 is redundant and is still executed, due to the fact that MRM had already sent the request to execute it and only those redundant lw instructions are deleted which are waiting in the queue to be executed.

6) **Test Case 6: One Write Port**

This test case ensures that the design implemented has only one write port. This can be confirmed in cycle 13 of core 2, where the processor of core 2 is stalled because the DRAM read is loading into register, making the write port busy, and hence is unable to execute the subsequent addi instruction.

7) **Test Case 7: Best Case**

For the best case, IPC should be closest to the number of cores, i.e., in each cycle, one instruction of each core is executed. We achieved an IPC of 5.8 for 6 cores. To make this happen, the cores must finish their execution simultaneously, so that minimum number of cycles are wasted. Along with this, there are fewer lw-sw instructions, and the ones that are present are redundant and/ or uses forwarding to reduce the number of cycles taken. In ideal case, the lw-sw instructions activate minimum number of distinct rows, to save cycles during write backs.

8) **Test Case 8: Worst Case**

The worst case or least IPC will be attained when only one of the many cores is being executed, while all other cores' execution is finished. This wastes the scope of executing other instructions parallel to each other in a multi-core setting.

To make this happen, we used 7 cores (we can increase the number of cores to get worse IPC). Except one, all the cores have only one instruction, that too is executed in a single cycle (no lw-sw instructions). Only one core has exclusively lw instructions, in which each lw instruction activates a new row, resulting in write backs after completion at every step. This produces a very small IPC of 0.05 due to lower number of instructions executed and large number of cycles taken.

9) **Test Case 9: 32 Cores**

This test case contains 32 cores, to show the efficiency of our multi-core design implementation. All the cores execute completely (if M is greater than or equal to the desired number of cycles for completion), without interfering with other cores' functionality.

10) **Test Case 10: Wait Buffer Full**

This test case consists 10 files consisting of a total of 70 lw instructions in total to test what happens when the capacity of the wait buffer is full. In this case, execution of further lw requests is stalled when the wait buffer is at maximum capacity(32). In other cases, it is also possible that at a time

more than 32 DRAM requests are pending as we also have a second queue of size 32 which consists of some requests which have the same row as the present buffer row.

11) Test Case 11: Individual Core Error

In this test case, all files have some error. File t1.txt has an undeclared label name, File t2.txt attempts to write to $zero register and File t3.txt has a syntax error. If error in one core is encountered, the execution of that core is stopped but other cores continue to run. Also, if at some cycle, all cores have encountered some error, the execution for all cores is stopped there and statistics are printed.