# Design for Assignment 5

Shrey J. Patel, 2019CS10400
Aayush Goyal 2019CS10452

May 2021

## 1 Aim

Till now, we have implemented, improved and optimized our simulator for handling instructions from a single input file i.e. single core. So, till now only one file was issuing requests to a single DRAM. Now our objective is to extend the functionality of the simulator to handle **multiple cores**(N) at once. In particular, the objectives are:

1. To implement a **Memory Request Manager** which decides the sequence/order in which the DRAM requests of different cores are issued.
2. Given an upper bound on the total number of clock cycles(M), we need to implement an algorithm to try to maximise the throughput of the simulator.
3. To estimate the delays in the memory request manager and incorporate it in the code.

## 2 I/O Specifications

- **Input:**
  1. N : Total number of cores (i.e. number of input files to execute simultaneously)
  2. M : Simulation Time(upper bound on the number of clock cycles, all the instructions after this clock cycle are not counted in throughput, if the execution hasn't already ended)
  3. All the individual input files.
  4. DRAM specifications: ROW_ACCESS_DELAY and COL_ACCESS_DELAY.
- **Output:**
  1. After every instruction:
     - Current instruction, its number and the core it belongs to, instruction address is: (Instruction no. printed - 1)*4
     - Clock cycle(s) required for that instruction.
     - Modified registers, if any.
     - Modified memory locations, if any.
     - Activity on DRAM.
     - Activity of Memory Request Manager(changing core, skipping instructions, etc.) as well its delay information.

2. After completion of execution:
   - Total number of row buffer updates.
   - Final list of non-zero memory locations.
   - Final values of non-zero registers in eahc core
   - Throughput information (i.e. total number of instructions executed during the simulation).

- **Execution Instructions:**
  - The code is divided into a header file core.hpp and the main file assignment5.cpp. It can be compiled with the command '**make all**' in the command line as a result of which an executable assignment5.exe is generated.
  - A particular testcase can be executed by using the following command:
    **./assignment5.exe N M input row_delay col_delay**

    *where the input can be provided as a folder with N input files or as N space separated files.*

  - The executable file can be removed using the command '**make clean**'.

# 3 Approach

Our two main objectives in converting a single core simulator to a multi core simulator are to implement parallel execution of the cores and the proper sequencing of DRAM requests by the different core by the memory request manager so as to maximise the throughput in a fixed simulation time. For parallel execution, we have simultaneously called the parser on multiple cores at once, so the cores whose next instruction is a normal instruction can continue their execution without submitting any request to the manager. All such normal instructions can be simultaneously completed in a single clock cycle. While some cores encounter an lw/sw instruction, in which case they need to submit a request to the memory access manager, and the non-blocking memory starts to act on those cores. Some cores may even come at a halt due to unsafe/engaged registers because at a time, a memory operation of only one core can be active, while other cores which have potentially unsafe instructions due to queued memory accesses have to wait for their turn. We also need to take care of reordering of lw/sw instructions of a single core if it submits multiple requests to the DRAM.

Because a halt in execution costs us extra clock cycles, it is always better to execute the DRAM requests of different cores in some particular order, so as to achieve minimum delay and therefore maximise the number of instructions executed in a given simulation time. And for that the memory request manager needs to follow some algorithm or a pattern so that there is maximum throughput and no core is starved i.e. lw/sw instructions of no core are ignored intentionally. For this, we have implemented an idea of continuing with the memory requests of the same core until either the row changes or a certain maximum number of requests are catered. And after this, the manager forcefully switches to some another core which has submitted requests to it. In this way, we can ensure that none of the cores are starved.

# 4 Code Design and Implementation in C++

The code for this assignment is divided into two files: a header file, core.hpp and the main file, assignment5.cpp.

## 4.1 Data Structures

1. **Core class:** Earlier, we only required to handle a single core, and all the resources (like register values, label values, etc.) were stored in the main program as local variables. But multiple cores have their own independent resources, which can be stored in separate core objects which are instances of the core class.

   Each core object contains:

   (a) Registers
   (b) Operation counts
   (c) Instructions
   (d) Labels
   (e) Separate counters :
      i. itr : (for iterating through instructions)
      ii. counter : (keeping track of the total instructions executed till now)
      iii. insCounter : (for mapping every instruction with its number for printing)
   (f) Error variable for switching the core off due to potential errors.
   (g) Data structures to implement non-blocking memory in a single core, like for keeping track of engaged/unsafe registers, etc. (e.g. registerUpdate)
   (h) Data structures to implement memory request reordering, like keeping track of last updated address for forwarding, or last lw/sw instruction, etc. (e.g. forRefusing)

2. **cores:** A vector of core objects for indexing the N cores.

3. **waitingList:** This map is the main data structure which stores pending lw/sw instructions, in a flexible order(unlike queue). For a particular memory operation, it stores the core which has issued the request, the row and column indices of the requested memory address, the register which it uses, and the counter which indicates how many operations have been executed before this.

   This waiting list is indexed by the core number, and for each core, it stores another map, which is indexed by the row of the requested memory address. The inner map maps the row number with a tuple of the type (int,string,int) storing the current value of counter , lw/sw and column number respectively.

   Note that both the inner and outer maps are ordered, so as to maintain a sequence cores as well as of lw/sw operations.

4. **print:** A map to store the contents to be printed. The map is indexed by the ending and starting clock cycles of the instruction and it stores the particular the string to be printed to explain the execution that takes place in that duration.

5. **command**: A tuple which stores information of the next lw/sw instruction to be executed as decided by the memory request manager.

6. **address_core:** Our algorithm assigns certain rows of DRAM to a particular, so that no row can be accessed by more than one cores. This map stores which row belongs to which core.

7. **store**: A tuple which stores the information to be printed once the next DRAM instruction/operation has finished execution. In particular, the tuple is of the format (string,string,string,string,int) which stores two different kinds of tuples depending on whether the next operation is a load or store operation:
   (a) (sw, clockCycles, address range, value, counter, core)
   (b) (lw, clockCycles, register name, value, counter, core)
8. **priority:** This array stores the priority value (of string type) of each core. Whenever a core needs it's register value to get completed, it tells the MRM whether we need to first go on that core ignoring the other cores in a cyclic order.

## 4.2 Algorithm

The main objective is to extend the features of our simulator like non-blocking memory and memory request ordering to multiple cores such that the normal instructions of the cores are executed in parallel simultaneously while the lw/sw instructions are executed sequentially due to a single DRAM memory. Now, the incoming requests from different cores can also be ordered in different ways by the memory request manager to obtain different throughputs.

So, our algorithm is mainly aimed at achieving this parallel execution as well as a suitable ordering mechanism for the memory request manager which tries to maximise the throughput.

### 4.2.1 Achieving Parallelism

- We have divided all the cores into separate core objects so that all their resources stay independent of each other.
- And because we have implemented a cycle by cycle execution, we can treat our execution as an iteration over the value of DRAMclock (global/master clock). At every increment of this clock, we run the parser on the input file of every core, and depending on the type of the next instruction, there are three types of cores:
  1. Those cores which have a normal instruction go ahead and execute without submitting any request to the request manager, provided no register is already engaged in a queued lw/sw instruction. This type of instructions are generally dominant in any MIPS program, and this is the reason while using multiple cores significantly speeds up our execution as multiple instructions of these type are running simultaneously. Note that this is true even for the core whose lw/sw instruction is currently executed by the DRAM instruction, due to non-blocking memory.
  2. Those cores which encounter an lw or an sw instruction treat it as a normal instruction at the first encounter and execute it in a single clock cycle, but the difference here is that because this are memory access requests, they are submitted to the request manager for issuing the request to the DRAM. Thus issuing the request requires a single clock cycle while actually executing that request may take more.
  3. Some cores may also encounter unsafe instructions, in which case the core halts until the engaged lw/sw instruction completes its execution. The throughput in this case decreases quite a bit because the number of effective cores decreases for that duration.
- However, delays due to such halts can be reduced by prioritising some lw/sw instructions and reordering the pending instructions on the basis of this priority. All of this is managed by the Memory Request Manager, which also makes sure that none of the cores is starved

#### 4.2.2 Memory Request Manager

- The memory request manager continues to execute the memory requests of the same core until either of the two conditions are satisfied:

  1. When the row buffer updates i.e. the lw/sw instruction demands to access a different row.
  2. When the number of consecutive lw/sw instructions executed for a single core exceeds some upper bound(in our case 5).

- In both of the cases, the request manager immediately switches to some other core depending on the priority of the other cores. The priority value of a core is pre-computed and updated, precisely at the time when the said core submits a request to the MRM.

- **Cyclic Priority:** Every core has three types of priority values: an empty string if that core hasn't submitted any request to the MRM, "do" if the core has submitted the maximum permissible number of requests (i.e. 32) and a string equal to a register if this particular register is engaged in some lw/sw instruction requsted by this core. The highest priority is assigned to the third type as it indicates an unsafe instruction, followed by the second type and then the first. The highest priority core in the cyclic order(i.e. numerically the next, except for the last core, in which case cycle back to first core) is the next one to be executed by the MRM.

- This decision not only prevents significant delay, but also prevents starvation in any core i.e. this rotating priority ensures that lw/sw instructions of none of the cores are intentionally ignored.

- **runMRM():** This function retrieves the next lw/sw instruction to be executed by the DRAM, depending on the current core and the cyclic priority values. This is called at every iteration of the clock cycle.

- **processCommand(tuple)**: This procedure handles the execution of the next scheduled command which is obtained as a tuple from getMRM().

  This function also updates the tuple 'store', which stores the details of the recently executed operation, for printing later.

- **processCompletion()**: This causes the information stored in 'store' updated in the above function to be stored in the print buffer at the appropriate clock cycle along with the corresponding instruction and the counter. The print buffer stores the entire output until the end of the execution. The whole buffer is printed onto the console at the end.

  This procedure is responsible for refreshing the values of the store data structure and the alternate cycle. This alternate clock tracks the current state of the processor. So, when its value is (-1), the processor is idle and ready to process. But note that the processor and the memory being separate, only the issuing of requests and the execution of normal instructions are handled by the processor, while the actual execution of DRAM requests are handled by the DRAM memory in a real simulator.

- **checkComplete(reg,i)**: This function checks if the input register of i th core is busy in some lw/sw instructions, in which case we allot the third type priority to this particular core so that the next time the manager decides to switch cores, this particular core has higher probability to be selected because of unsafe instruction.

## 4.3 Testing

We have manually created our own test cases and provided them as input to the assignment5.cpp . In case of correct input, we have printed the detailed output. While for incorrect expressions, we terminated the program with a appropriate error message which would be helpful to the user for debugging. The example test cases are present in the submission folder. The various cases that we considered are:

1. **1 write port**: Since we have only one write port in the processor, so in the same cycle we cannot write to 2 registers. One write comes "lw" and the other write comes from processor. The write from the processor is delayed in that core.
2. **Maximum M cycles**: We have made sure that the program only runs for maximum of M cycles as provided by the user. A final writeback from the buffer to DRAM is not included in this M cycle cap.
3. **Cyclic Priority**: The next core that has to be executed is estimated by running in a circular way. Among these if aa core has unsafe instruction due to which it's execution has halted then that core is given higher priority by the priority encoder.
4. **Same Row access among diff cores**: Same row cannot be accessed across different cores and we have thrown an error in that case.
5. **Finite Size**: The size of Waiting List for Core is finite (32). So we have made sure that it never has more than 32 pending requests in it. A request is said to be removed from the Waiting List if it's execution has completed in the DRAM.
6. **Forwarding**: We have made sure that forwarding is done only when the request is present in the Waiting list.
7. **Only lw**: we have implemented the dirty clean method. hence made sure that if the buffer is clean then we are not doing any redundant writeback. This saves one Row access delay.
8. **Scrapping**: scrapping is done such that there important instruction is missed.
9. **Amortization**: Making that Cyclic priority encoder has helped us bring down the worst case time from 450 cycles to 250 cycles. This is a very good improvement achieved because we have amortized the IPC for DRAM instructions.

Hence all the test cases have helped us to ensure that our assignment5.cpp is correct. Getting correct outputs on some test cases can never be the right criteria to judge if an algorithm is correct or not but extensive checking can still ensure us that we have done the things correctly.

## 4.4 Throughput Estimation

The throughput of any program is the the total number of instructions executed in a unit time. So, in our case, the throughput of our simulator can be calculated as the total number of instructions executed divided by the total simulation time(which is given to be M). We have provided the total number of instructions in the output. So, throughput of the simulator for a given test-case is:

$$Throughput = (Total\ number\ of\ instructions)/M$$

As discussed earlier, the maximum throughput in the simulator can be achieved when all the instructions are executed in parallel without any halts. Further, a single lw/sw instruction takes greater than one clock cycle to complete, while any instruction takes a minimum of one clock cycle to complete execution. So, for maximum throughput, we need to maximise the number of instructions for the same amount of simulation time(M), which is possible if all the instructions in all the cores are normal (non lw/sw) instructions, in which case all the cores are simultaneously active in all the clock cycles. So, if there are N cores, N instructions will be executed per clock cycle, one in each core. So, regardless of the total simulation time:

$$Maximum\ throughput\ of\ our\ simulation\ design = N \text{ (where N = Total number of cores)}$$

Also when there are a lot of lw and sw instructions in the code (which is generally the case in real life) we have made the cyclic type of priority encoder. This gives us a amortized cost (like 30 per 5 DRAM instructions if row delay = 10, and col delay=2). Now this amortization saves us from some worse cases. For this we have given an example in which, if we change the row everytime a core encounters an unsafe instruction, then the total time taken would be 450 cycles. But this amortization has helped us bring down that time to 250 cycles.

## 4.5 MRM Delay Estimation

1. A clock cycle in processor consists of various processes like: instruction fetch, instruction decode, register read, ALU operations, the heavy combinational logic of the control signal also happens in parallel with all these, sending DRAM requests and writebck to register port. So whatever we do in the MRM's cycle should be decided according to this. Else it would be a wastage of time.

2. In the MRM we have assumed that there is a waiting list for every core of size 32. The maximum allowable cores are 16. Now whenever a processor sends any instruction to the MRM, all those cores push a DRAM request in their respective queues (with some other instruction being encoded as bits). So the DRAM requests from all the cores is written in parallel in their respective wait buffers.

3. The MRM maintains 2 variables Current Core and Current Row, they denote which core and row is being executed in the DRAM currently. The MRM also maintains how many continuous instructions from a core it has executed, this is stored as Current count. This information is directly available from the current instruction being executed in the DRAM (that is why we have not yet removed it from the Waiting list).

4. Meanwhile the cores are running and their maybe some unsafe instruction in them. Now the cores which have any unsafe instruction will keep sending a message to the MRM that they are waiting for their "lw" for a register to get complete (or any instruction in case the size of Waiting buffer is complete). The MRM will receive this information and use it in the priority encoder.

5. Now we will use a Priority encoder to determine the next instruction that we will send. Note that this dynamically keeps updating and the final instruction which was decided just before the completion of a instruction in DRAM is sent next to the DRAM.

6. The working of priority encoder is such that it gives priority by considering the following order:

1) Whether the current row has some pending instructions and we have done less than 5 instructions in that core
2) If it has a unsafe instruction (this information is continuously sent by the respective core)
3) Core which is cyclically leftmost to the current core. This can be encoded in the priority encoder using some combinational logic.

7. Now once we have the core to be executed (and the specific row in case of unsafe instruction in some core) we will have to fetch the command from waiting buffer of that core. Also among the various instructions from the same row we choose the one which is lowest in the queue. This choosing of lowest can again be done using a priority encoder. For other selections we can use multiplexers, the number of units that wee will require will be high for this but all this can be done in parallel.

8. The last thing it does is whether the instruction actually needs to be executed or not. For this again some combination logic and some input from the processor can be used.

   So what we are doing is: pushing into wait buffers, priority encoder to determine core and row, fetching the instruction from that core's buffer, choosing the appropriate row and lowest instruction (by time of arrival), checking whether it is redundant or not.
   So to determine all this and preparing an instruction for the DRAM, we will require one clock cycle (because the similar amount of work is done in the processor as well). When the clock edge will come then the DRAM will accept the request if it is ideal. Thus the Delay of Memory request manager is 1 cycle to determine each instruction.

## 4.6 Features of the simulator

### 4.6.1 Strengths

1. We have designed our algorithm to ignore redundant instructions. Some lw/sw instructions have no effect on the final state of the memory addresses. For e.g. in case of multiple consecutive lw instructions in which values from different addresses are fetched and stored in the same register, then the final value stored in the register depends only on the last lw instruction and so all those intermediate lw instructions, whose execution hasn't started at the moment the last lw instruction is issued, can be considered redundant and so they can be ignored, thus saving clock cycles.

2. We have also also implemented forwarding in which if an lw instruction request a recently updated memory address(using sw), then instead of fetching it from DRAM, it is directly accessed from the buffer, thus avoiding row access delay. Note that this has to be done only when the sw instruction to that Address is present in the Waiting buffer of that core.

3. In all the earlier versions of this simulator, whenever we changed the row buffer, we compulsively wrote back the whole buffer to the corresponding row, but this writeback may sometimes be unnecessary if the current row buffer has not been updated. So, in the current version, we write back the buffer only if it has been updated(i.e. is dirty).

4. We have also implemented a rotating priority function so that there is no starvation of any core. This ensures that none of the core is ignored over the course of execution. Also this gives us an idea of the amortized cost of the operations. This is better than just changing the instruction whenever a core encounters an unsafe instruction because in that we may be

benefited a lot, but at the same time we may be penalized a lot. To reduce the risk of this penalty it is better to go with a version that works good enough in the cases.

5. We are executing the MRM in parallel with the other DRAM and processor. The MRM doesn't wait for the DRAM to become idle. Also it removes many redundant instructions in the meanwhile. This also saves us a lot of time and mostly we will not have to wait for the MRM to prepare a instruction for us.

6. We have tried to keep our implementation close to the actual hardware implementation by simulating clock-cycle by clock-cycle execution. Earlier, we used two clock cycle values to hop from one point in the program to another. For e.g. as soon as an lw instruction starts execution we advance the time by the total number of clock cycles required at once, while this time, we have incremented the number of clock cycles by 1 in every iteration so that we can get information about each core in every clock cycle. This is essential in ensuring parallelism between the cores. Another consequence of this approach is that only one master clock cycle is required for keeping track of elapsed time.

#### 4.6.2 Weaknesses

1. In the MRM we have assumed considered a waiting buffer of size 32 for every core. The total number of cores that we have assumed is at max 16. Earlier we had a wait buffer of size 32/64. But this time, we require a significantly larger buffer size for storing the DRAM requests, which adds to the space complexity.

2. The decisions taken to maximise throughput are non-deterministic, i.e. there is no concrete proof that choosing a maximum of 5 lw/sw instructions for any core before switching to another would give us the best possible throughput. This is just an **estimate** for achieving a throughput that is better than the one we would have achieved if we would have executed another cores only after finishing the requests from this core. This strategy was chosen as reduces delays due to halts and prevents starvation. The motivation for using this method comes from the fact that allowing multiple cores to run simultaneously is always better than allowing a single core. So, it is always logical to keep changing cores instead of staying on the same core, so any possible halt can be prevented, and thus maximising the number of cores that are simultaneously running, which is trivially better if lesser number of cores were running. Choosing a suitable interval for changing cores is still non-deterministic.

3. In continuation to the above point, there is no method to determine the best throughput in this simulator without any sort of look ahead.