Department of Computer Science and Engineering, IIT Delhi

COL216 Computer Architecture

Assignment 5 Report

MIPS Simulator - Multi Core Functionality

Submitted by-
Pratyush Saini (2019CS10444)
Anirudha Kulkarni (2019CS50421)

Supervised by-
Prof. Preeti Ranjan Panda
Associate Professor
CSE, IITD

Input:

1. **Parameter N** : The number of CPU Cores. (Can be upto 64)

2. **Parameter M** : Simulation time. The number of clock cycles upto which the simulation needs to be run (though all the cores execution might not have been completed by then).

3. MIPS assembly language files

t1.txt
t2.txt
...
tN.txt

4. DRAM timing values, **ROW_ACCESS_DELAY** and **COL_ACCESS_DELAY**.

Output:

1. **Activity in the system for each clock cycle for all the cores.**
   - This may include DRAM issue operations, DRAM execute operations, DRAM loading/write back operation or execution of R/I/J-type instructions for different cores.

2. **The relevant statistics for each CPU.** This includes
   - The contents of register File for each core present. (As in previous assignments).
   - Memory content at the end of execution. This corresponds to the values stored at relative addresses for each core.
   - The number of times each instruction was executed for different CPUs.

3. **Throughput Efficiency** corresponding to total number of instructions executed upto cycle M.

4. **Delay caused by Memory Request Manager.**

Build Instructions:

Program can be build very easily with makefile
Instructions for makefile:

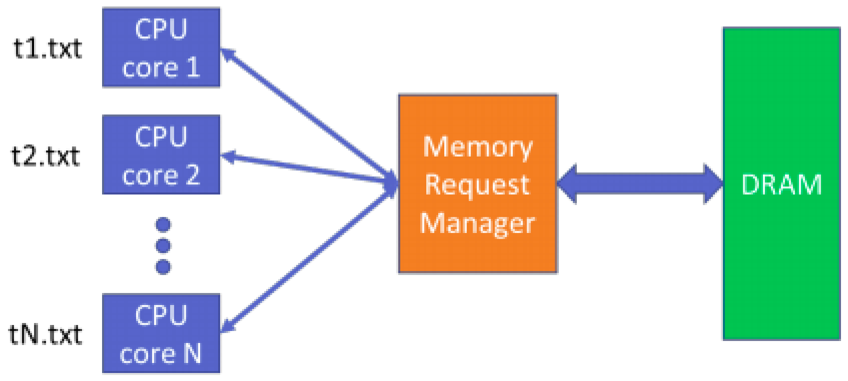
1. To build : make build
2. To run the executable from build : make run
3. To remove the old build files: make clean

All these can be avoided with make all initially.

Parameters:

1. **N = number of cores**
2. **M = To print clock cycle till that cycle.**
   Use M=-1 for all clock cycles. Else specify clock cycle number till which it will be printed to the console.

Memory Request Manager:



In multi-core systems, main memory is a major shared resource among all the processor cores. Tasks running concurrently on different cores contend with each other to access main memory, thereby increasing their execution times.

We incorporate this functionality in our MIPS Simulator by a separate unit **Memory Request Manager.**

The memory request manager manages multiple requests coming from all the CPU cores and issues them to the DRAM buffer on some priority basis.

- Row-hit memory requests have higher priorities than row-conflict requests.
- In case of a tie, the core which has been stalled first (i.e. the oldest core whose dependency has to be resolved to continue it’s execution) is given higher priority.

Since there are N processors and only 1 DRAM, some "arbitration" is needed to ensure that only one processor's request goes to the DRAM at any time. This task is performed by the Memory Request Manager, which sees N CPUs on one side, and 1 DRAM on the other.

There is only 1 DRAM. All cores will access the same DRAM, but each core has it's share of DRAM addresses it can access. There can be only 1 DRAM access at a time and while
DRAM is being accessed, other DRAM requests can not be issued to DRAM. The pending requests wait in the buffer.

Implementation:-

We have a separate class of MRM, which takes in lw/sw requests from different cores.
On each cycle, the MRM sees the state of the DRAM queue and issues requests to the DRAM buffer after suitable recording.
This is bound to produce some delay in the system which is discussed in the upcoming sections.

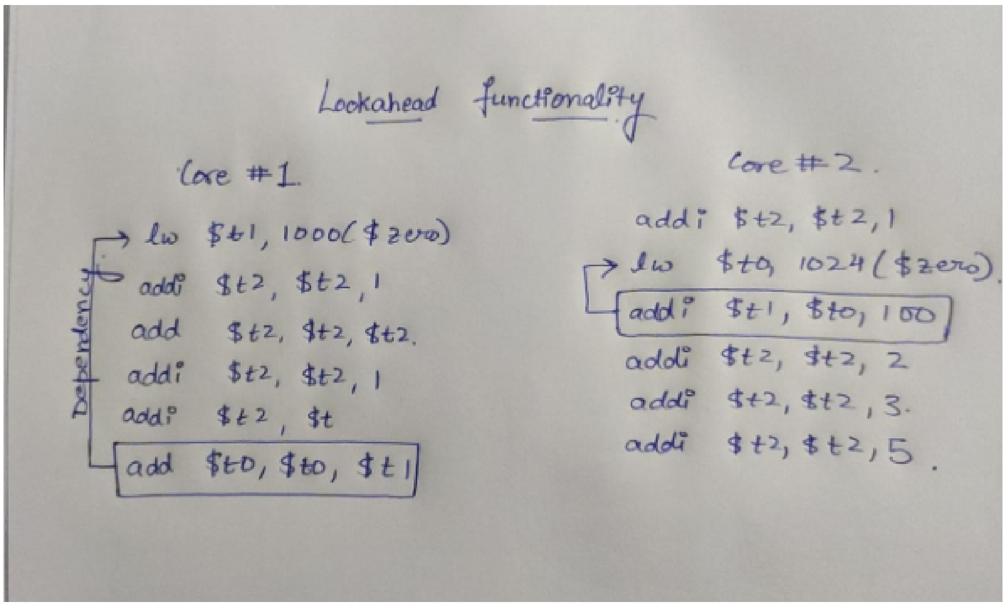
Single Core Starvation:

It might be possible that sometimes, multiple lw/sw instructions (thousands or millions) are present simultaneously in a single core which corresponds to the same row in the DRAM buffer.

In order to prevent starvation of our program on a single core, we incorporated a starve limit , due to which, if the number of instructions waiting in DRAM Waiting queue exceeds the starve limit, higher priority is given to (if present), other cores present in the waiting queue and once their execution is complete, the simulator shifts execution to original core.

<u>MRM Logic :-</u>
We have incorporated various MRM logics in our simulator, each one having its own strengths and weaknesses.

a) <u>Lookahead Functionality</u>



```
lw   $t1, 1000($zero)
addi $t2, $t2, 1
add  $t2, $t2, $t2.
addi $t2, $t2, 1
addi $t2, $t
add  $t0, $t0, $t1
```

```
addi $t2, $t2, 1
lw   $t0, 1024($zero).
addi $t1, $t0, 100
addi $t2, $t2, 2
addi $t2, $t2, 3.
addi $t2, $t2, 5 .
```

This approach relies on the decision of MRM on which request to issue in the DRAM first.
Memory manager needs some cycles to decide which request should be sent to DRAM next. This might cause delays to overall execution.

Strengths :

Increases throughput efficiency in some cases where core is stalled in almost the beginning of the program.

Decides which request to send to DRAM by giving priority to some Cores, based on the waiting time for each core.

Weakness :

Might waste some additional clock cycles in the beginning in deciding which request to issue first in the DRAM buffer.

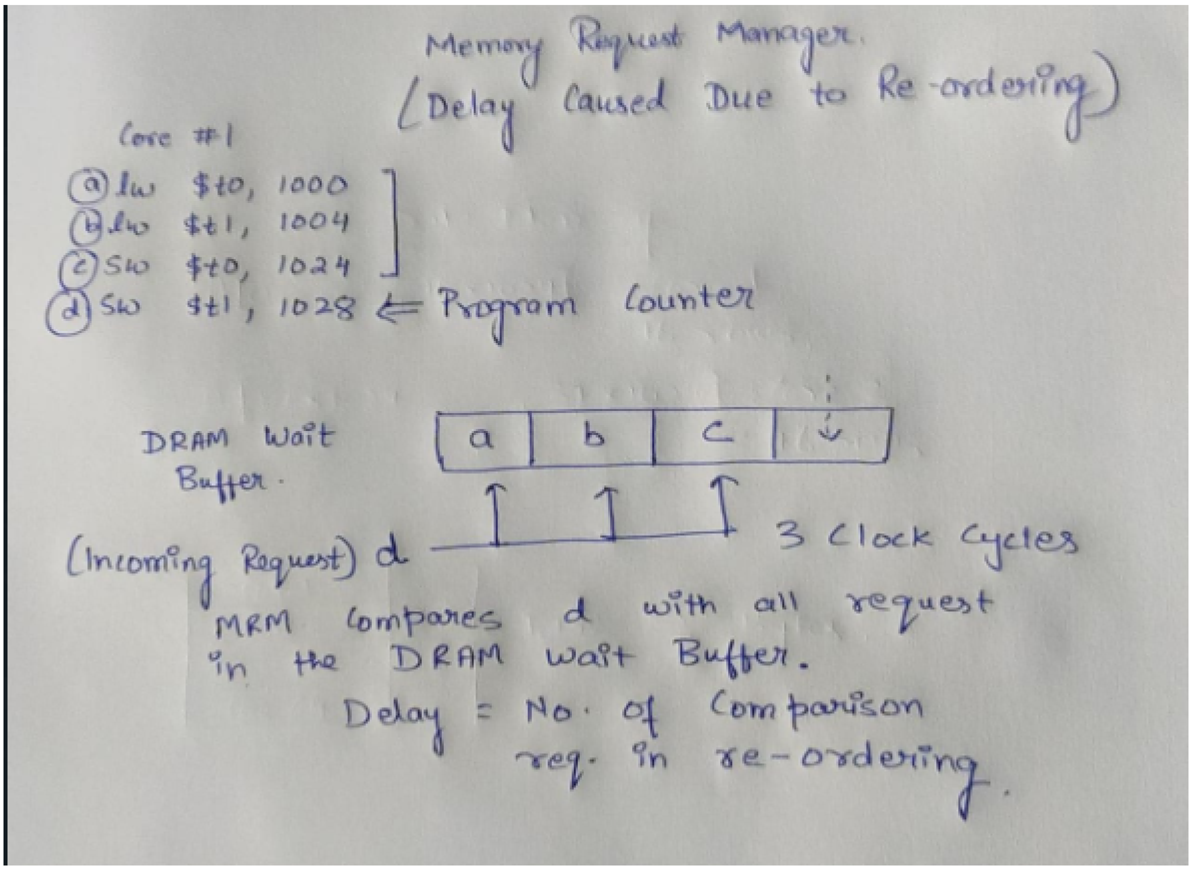
b) First come first serve

In this MRM logic, the DRAM starts performing it’s operation as soon as the MRM encounters any lw/sw request and sees the state of DRAM buffer to be empty.
The subsequent request corresponding to row hits get appended in the front of DRAM queue and rest of them gets lined at their appropriate positions in DRAM buffer after consuming some clock cycles.

Strengths :

The DRAM never remains in idle state until and unless the MRM buffer is empty.
As in look ahead functionality, it gives some priority to the cores on the basis of their waiting time.

MRM Delay Estimation :



Memory Request Manager.
(Delay Caused Due to Re-ordering)
Core #1
(a) lw $t0, 1000
(b) lw $t1, 1004
(c) sw $t0, 1024
(d) sw $t1, 1028 ⇐ Program Counter
DRAM Wait Buffer.
a
b
c
(Incoming Request) d
3 Clock Cycles
MRM Compares d with all request in the DRAM wait Buffer.
Delay = No. of Comparison req. in re-ordering.

In actual hardware implementation, re-ordering of requests is not such a simple task.

For re-ordering requests in the MRM unit, we have to re-arrange the existing requests such that the number of row-conflicts is minimum. These operations are based on certain decision trees where we need to compare incoming requests with the already existing requests and allocate appropriate positions such that the number of row-conflicts caused are minimum.

The causes of delay in our Memory Request manager is mainly due to:

a) In a single clock cycle, only one request can be issued in the DRAM Waiting buffer.
b) While issuing a request in the DRAM buffer, we need to compare the incoming request to the existing requests for proper re-ordering. We assume that a single comparison will consume one clock cycle.
c) For the look-ahead MRM logic, we need to wait for some cycles in the starting to make a decision on which request to issue first, which might also cause delay in certain cases.

Main Memory Specifications :

Avoid Memory Contention: Memory contention is the situation in which two different programs try to make use of the same memory resources such as disk space, RAM, cache, or processing threads at the same time.

This could result in deadlock or thrashing (when the memory is forced to constantly receive and store data in secondary storage blocks called pages).

Avoid False Sharing: False sharing occurs when two or more processors in a multi-core system are making use of the same cache line that is not related to their operations concurrently.

The cache system could become confused and this might result in invalidating or rewriting the cached copy of other processors.

To avoid much mishaps in the Simulator, we have allocated disjoint address spaces to each core.
We divided the memory blocks across all the cores available.

Each core can access it's memory blocks only and can not access memory blocks of other cores.

Eg. Row 0 to Row 15 => Core 1
Row 16 to Row 31 => Core 2
....
i.e. 16 x 1024 bits of data memory to each core.

If the cores are running different programs, then separate address spaces is a simple way of ensuring security (one program shouldn't corrupt another program's data).

However, some ways to communicate across cores though safe and controlled mechanisms are indeed needed.

Multicore Functionality :

- The MIPS simulator is capable of incorporating upto 64 cores (can be changed) in parallel.

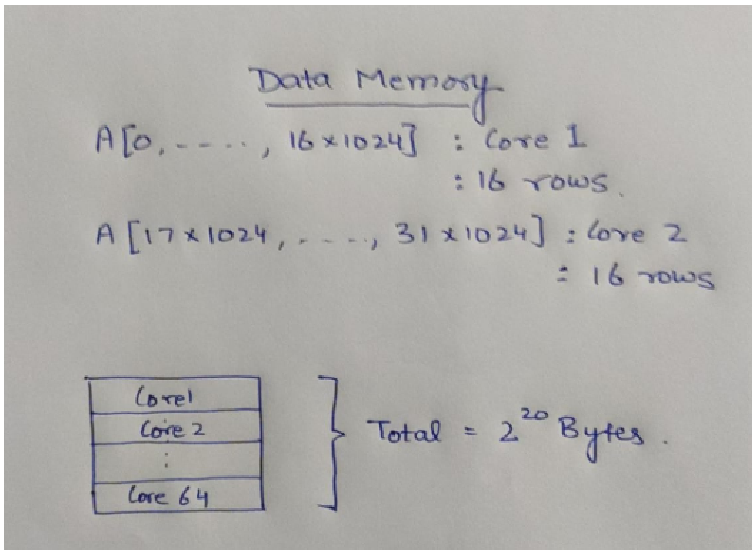
- Programs running in the different CPU cores are independent of each other.

- The programs corresponding to multiple CPU cores run in parallel with each. One program’s resources cannot interfere with other other program resources.

Summary-

Assumptions:

1. Each core has Separate Register File which do not interfere with each others
2. Separate Instruction Memory (as given in the specifications that we have to maintain separate memory space for Instructions and Data).



3. Different set of rows in the main Data Memory corresponding to each core.

a) Strengths-

i) Multiple programs, each corresponding to different CPU cores run in parallel.

ii) No two cores can share the same resource with each other (Register File / Instruction Memory / Main Data Memory).

b) Weakness-

i) Once a request is sent to DRAM for row activation/access/writeback, we cannot stop it. Any optimization/processing can only happen in the queue.

ii) At each call of function complete_dram_activity, we are fulfilling all the pending DRAM Requests. Rather, a better optimization would be to fulfill only those requests which could resolve dependency of the current instruction. This was not feasible in our design decision due to the limit of lookahead possibility.

NOTE: Please find the testing doc for further elaborations of the strengths and weakness