COL216 Assignment 5

Abhay Pratap Singh Rathore 2019CS50414
Himanshi Ghai 2019CS50433

May 2021

1 Introduction

The report contains instruction to run the code, design and approach of the memory request manager, and the testing strategy.

1.1 Processors

There are total n number of processors, where n is given as an input. Each processor executes program independent of other cores. Equal and disjoint memory blocks allocated to each processor. The instructions have separate memory storage than the main memory. Processors generates memory read and write request on instructions *lw* and *sw* respectively. These requests are transferred to the *Memory Request Manger*.

1.2 Memory Request Manager (MRM)

The memory request manager stores the incoming request from all the n cores and selects which request to process first by the DRAM.

1.3 DRAM

DRAM accepts request from the **Memory request manager** and process the request in the DRAM. The DRAM has a 2D structure with 1024 rows and 1024 columns of one byte each, with a row buffer of size 1024 bytes. The Row access and column access delays are given as input.

2 How to Run

To run the code,

1. Build the project using the command **make**
2. Run the code with command **./main "input file path"**

Input file format

1. The first line contains two integers n and m, the total number of cores and the number of cycles to be executed.
2. Next line contains two integers, Row access delay and column access delay.
3. The ith line of the next n lines contains path(relative to the executable **main**) of the code files for the ith processor.

The code files for each processor follows MIPS format and has all the available instructions as previous assignments.

The output of the program will be in a new file names **out.txt**. The output format is,

1. At the beginning of the output, the output is of the following format
   (a) Cycle i, where i is the cycle number
   (b) Next n lines contains the description of what each processor has performed in this cycle
   (c) Next line has the description of work done by the DRAM in that cycle
2. At the end when the description of all the cycles has been completed, then final state of each processor is printed, which contains the register file and the total number of times each instruction is executed.
3. The last line contains the throughput of the complete system, which is the total number of instructions executed by all the processors per cycle.

3 Design and Approach

MRM contains one load queue and one save queue corresponding to every core, i.e., n number of total load queues and n number of total save queues. And one row miss priority queue. The basic mechanism works in following steps.

Step 1: Whenever a processor executes any *lw* or *sw* instruction, a new request is inserted in one of queue corresponding to the core in the MRM depending upon the details of request and the other MRM signals.

Step 2: MRM selects the best possible request form both load and save queues for all the n cores at the same time as they are different copies of same hardware. Lets call this hardware of selection as **Internal Selector** which is first stage of MRM. This hardware has n copies corresponding to each core.

Step 3: Now, MRM selects one out of n best requests corresponding to each core. This request is then set as the best of all possible requests. Lets call this selection hardware as **External Selector** which is the second stage of MRM.

Step 4: If DRAM is not processing any request, then DRAM receives the best of all requests generated by the External selector in the previous cycle and starts executing it.

Timing: Internal selector is a combinational circuit and selects best request in the same cycle in which the core writes a new request to the queues. Which means, steps 1 and 2 executes in one cycle one after another. Step 3 is executed completely in one cycle.

So, lets say if at cycle 8, an *lw* instruction is executed by a processor, then at cycle 8, **Step 1** will occur. Then at cycle 8 itself, this request is seen by MRM and it may select this request as the best possible request by the Internal Selector. Then in cycle 10, this request is seen by the external selector and may be selected as the best. Then in cycle 11 this request is seen by DRAM. (In this example we ignored interference of other requests). The best request selected by Internal Selector is stored in a D Flip-Flop based register, in which the Internal Selector sets the D signal of this register and External selector read from the Q signal of these registers.

3.1 Data structures

Both these queues are of size 32, i.e., they can hold maximum of 32 requests each at a time. And the row miss priority queue is of length n.

3.1.1 Load queue

The load queue is an array of blocks of memory indexed from 0. The index where There are total n load queues corresponding to each of the n processors in the MRM. Load queue has 32 requests space. Each request block stores following signals,

1. Valid signal (1 bit)
2. Busy signal (1 bit)
3. Wait signal (1 bit)
4. Address from where we need to read (32 bits)
5. Other control signals (Mostly 1 bit size)

The rest of the information about to which register the memory value needs to be loaded is embedded in the index of the request in the queue, i.e., a load word instruction to the register 5 will be stored at index 5 of the load queue. This method of indexing makes it easier to find which register needs to be loaded before using that register as the input to some future instruction. And this is efficient as it is a direct map from the instruction to the index of the request in the load queue. The busy signals denotes whether or not there is a pending load request currently being executed by the DRAM to some register.

This queue is initialized by requests with valid and busy signals not asserted.

3.1.2 Save queue

The save is very similar to load queue. It is an array of 32 memory blocks. Save word requests are stored in this queue. There are total n save queues corresponding to each of the n processors in the MRM. This queue is also a direct map and it is addressed using the **mod** function.

$$Index = Address \ mod \ 32$$

Save queue stores following signals,

1. Valid signal (1 bit)
2. Value to be stored at the given address (32 bits)
3. Address to where we need to store (32 bits)
4. Other control signals (Mostly 1 bit size)

This queue is initialized by requests with valid signal not asserted.

3.1.3 Row Miss Priority Queue

This queue is of length n. The data in this array is stored in increasing order. It queue stores number of row misses for every core. Whenever there is a row miss, row miss value of corresponding core in the queue is increased by one and the value is shifted until the queue is in increasing order again. We gives the priority to the core with lowest number of core misses.

3.1.4 Other registers and signals

There are also other small registers and signals that contains necessary data to set control signals of the MRM. Some of them are listed below,

1. Busy memory (32 bit), this register is given the memory address at which current save request is being executed in the DRAM. In case when no save request is being executed in the DRAM, this register stores -1.
2. Active row (10 bits). This register stores the currently active row of the DRAM, that is latest accessed row from the DRAM.
3. Write busy (1 bit). There will be n such signals corresponding to each core. If this signal is asserted than it means that the register file is being written in this cycle by the DRAM, than no other instruction is allowed to write in that same cycle. For doing this, the processor is stalled for one cycle if necessary (except for branch and jump instructions).
4. Signals to store whether the DRAM is busy and other control signals needed for execution.

3.2 Execution and Estimation

Now, let us see in greater depth about how all the steps mentioned above are executed and what are their consequences.

3.2.1 Non-blocking memory

Before we move forward in the steps of execution, lets see how non-blocking memory works. Whenever there is a pending request of load type to a register, we first need to load the memory to that register and then execute and future instruction that uses the value of this register as input. For determining this, we use busy signals of the load queue corresponding to the core in the MRM. If the busy signal of any of the input register is asserted we stall the pipeline until that signal turn 0. So, that the incorrect value of any register is not used in any instruction.

Like previous assignments, if DRAM and MRM are busy and independent instruction comes to processor then the instruction is executed independently of DRAM and MRM, else the processor is stalled.

3.2.2 Step 1

In this step as mentioned before, the request is generated from the instruction of a core and stored to either load or save queue corresponding to that processor in the MRM. On the basis of instruction type, there are two cases.

Load: If the instruction is load, then we first check if the busy signal for the input register that is used to calculate the address of the memory is not asserted. Then we follow the following steps,

1. If there is a save request in the save queue corresponding to the same core, which is going to store some value at the same address as the load, we forward this value to the register at which the load is needed to be done. This saves many cycles as the load instruction will be completes in just one cycle. This scheme is called **sw-lw Forwarding**.
2. If forwarding is not possible then a request is written in the corresponding load queue at the index overriding what so ever the value was previously. This helps to prevent redundant load word instructions on same register
3. Set busy and valid signals to 1
4. If there is already a load request is present to the input register of the request then we stall the pipeline and waits for the request to complete.

Save: If the instruction is save, then if first check if the busy signal for the input registers is not asserted. Then we follow the following steps,

1. Calculate the index in the corresponding save queue where the request of this instruction needs to be stored by using modulus function.
2. If there is already a request of address other than the address of the new instruction is present at the same index, than we must wait, so the processor is stalled.
3. If there is any load request present in the load queue at the same address then we need to stall the core else the program will get corrupted.
4. Else if either the already present request is not valid or the already present request is at the same address as new instruction, we write the request in the save queue. And wait for the selectors and DRAM to do the job.

For any other instruction, if the input registers need not to be loaded, then we proceed with the execution without waiting for any pending request in the queue to complete. If there is some load request pending to the same register which is going to be overridden due to the new instruction in the core, then is possible we delete this pending requests as at the end the value will be overridden.

3.2.3 Step 2

In this stage, Internal Selector of each core in MRM works on their respective load and save queue to select the best request corresponding to it. It is combinational logic. And sets the D input of the Flip-Flop storing the best request.d

Load Word Queue: Request are given priority in the following order

Waiting Register < *Same Row Access* < *Waiting Register and Same Row Access*

1. Waiting register benefit is that, if the core is stalled due to the fact that any of the input register to the next instruction(that needs to executed) is busy due to a pending load request that needs to load the register first before executing, then we say the input register is waiting for some pending load request to be completed. If any load request loads to a waiting register than that request offers waiting register benefit.
2. Same Row Access tell is the load request access same row which is currently in the row buffer of the DRAM. This benefit will protect us from spending so many cycles in row write back and access. So we give this benefit some better priority.
3. If no such request exist then the request on the register with lowest number is selected. This decision is consequence of using a priority queue in the Internal selector.

Store Word Queue: Request are given priority in the following order

Waiting Memory Block < *Same Row Access* < *Waiting Memory Block and Same Row Access*

1. A request R, has Waiting Memory benefit if there is another save request waiting in the stalled core which is stalled because the request needs to be stored at the same index where the request R is already present. So, we first needs to complete this request so that the processor starts working as soon as possible.
2. Same Row Access tell that the request same row which is present currently in DRAM row buffer.
3. If no such request exist then the request with the smallest index in the store word queue is passed set best request.

3.2.4 Step 3

In this stage external selector selects the best request in the requests coming from multiple cores which were selected by internal selector in previous cycle as best request of each core. It has same logic as Internal Selector.

Priority is given as: *Waiting Memory Block* < *Waiting Register* < *Same Row Access* < *Waiting Memory Block and Same Row Access* < *Waiting Register and Same Row Access*

1. If there are more than one request with highest priority then request from the last most core is selected.
2. If there is no best request with any of the above benefit then the request from the core with least **Row Miss** is selected.

3.2.5 Step 4

In this stage, DRAM, if idle, gets the best request from External Selector which was found out in previous cycle.

If there is a row miss, then,

1. Increase the row miss count of the processor from which the request is selected
2. Start updating the row miss queue parallel to the execution of the DRAM.
3. DRAM will take minimum of *Row Access Delay*+*Column Access Delay* number of cycles to complete this request. So, we will have a large number of cycles to update the row miss queue in increasing order.
4. During the update process of row miss queue, the minimum value won't matter because the best request selected by the External selector is of no use as the DRAM is busy and will reject any new request.

5. So, we can update the row miss queue parallel to the execution of the DRAM.

6. In case the DRAM execution stops before the completion of the row miss update, then we stall the DRAM to take the new request until the update is done.

7. But since DRAM generally takes large number of cycles, there won't be any situation when this will happen.

The steps 1 and 2 are executed in same cycle. So, the best request is selected by the Internal Selector in same cycle in which the request is generated from the core. The step 2 is executed separately from steps 1 and 2, and takes one cycle. So, the Internal selector selects the best request in same cycle as step 1, so no cycle addition due to internal selector, external selector although takes one extra cycle to select the requests from the multiple cores. So, after the delay of one cycle, a request from core can reach to DRAM in best possible case.

Row miss queue update depends upon how the swapping occurs in an array. But in general case, the number of cycles taken by the DRAM are large enough to update the row miss queue efficiently before the DRAM is available for next request.

4 Strengths and Weaknesses

4.1 Strengths

1. Best Request selection based on priority computed using row benefit and dependencies of other instructions on read and write request.

2. Redundancies removed. By removing unnecessary pending requests.

3. When there is no best request selected, next request is selected using number of row miss of individual cores. Here, request is selected from the core with spatial locality. This improves performance as those request would be executed which access same row.

4. In every cycle best request is updated by MRM.

5. We do not write back the row buffer on row miss if no values are changed on it.

6. Enabled forwarding for save request to load request.

7. Direct mapping in load queue enables fast access to check states of requests for particular register.

8. Internal selector for each cores works in parallel for each core benefited by the copying of same hardware.

4.2 Weaknesses

1. When request is selected from a core according to least number of row miss, then we just select the first request from the queue. It might not be the request which access the row which most of the pending request accesses.

2. We do not account priority to the number of requests which access same row. One could also select a request which has more number of requests in queue that access the same row as this request. This could be beneficial in case of compulsory misses.

3. No prefetching is done

4. Limitation of direct map in save queue. One could make the save queue n-way associative. This could enable multiple requests mapping to same index to be stored at same time.

5 Testing Strategy

We need to check all the schemes and logic we implemented in the MRM. First we check individual schemes like forwarding, non blocking execution of instructions, best request selection based on different priorities, and other schemes we implemented. After testing single core execution for all the test cases we go to multiple cores. Starts with basic multiple core execution then tests all the decisions in the design choice, working of external selector, etc. The detailed testing strategy in given below. Further, while testing, our aim was also to improve our design to improve the throughput.

1. Testing single core scenario.
   (a) Check non-blocking scheme.
   (b) Check for best request in different cases of priority as explained in best request selection logic.
   (c) Check for sw-lw forwarding.
   (d) Check for overriding of redundant request in case of sw, lw and any other instruction which overrides same register as any valid request in load word queue at same register.

2. Testing multiple core scenario.
   (a) Checking for the non-blocking instruction execution.
   (b) Checking for best request selection with same row benefit.
   (c) Checking for case when there is no best request from any core, so selection on basis of least row misses.

(d) Checking for redundant requests when in queue, internal selector best request buffer and external selector best request buffer.

(e) Check for overriding of lw and sw request when load to same register and store to same memory respectively.

(f) Checking sw-lw forwarding.

3. Throughput Analysis

(a) Checking with more number of cores. Input MIPS code of each core targeting various efficiency schemes described above.

(b) In general, basic MIPS codes for different cores.

(c) By cycle analysis, adding more improvements to the design.

All test cases are present in "Test Cases" folder. It contains two sub-folders, "Single Core" and "Multi Core", which contains the corresponding test cases. Every test case in Multi core contains one input file that contains the input to the executable, a description file, and one input file for each core. Single test cases folder contains one input file, one description file that contains description of all the test cases, and 7 input files for 7 test cases of single core execution.