ECE 485/585

Final Project Report

Ryan Bentz

Spring 2018

LIST OF DELIVERABLES

A. Introduction

The goal of this project was to design and simulate a memory controller serving a single-ranked 8GB DIMM. The team was given the option of implementing the controller using one of three scheduling policies: in-order no access scheduling with a closed-page policy, in-order no access scheduling with an open-page policy, or access scheduling, possibly out of order. Additional constraints will be addressed in Section II, and the team's choices and design decisions will be discussed in Section III. Please note that this report was written for the specific intended audience of the course instructor. That is, someone who is well versed in how DRAM and memory controllers work, and is primarily interested in our approach to the project and our design decisions.

B. Constraints and Design Specifications

The memory controller is to serve the shared last-level cache of a four core 3.2 GHz processor which employs a single memory channel and uses a relaxed consistency model. The team was free to choose any programming language or HDL with which to implement the controller. The DRAM served by the controller is a single-ranked 8GB PC4-25600 DDR-4 DIMM organized as x8 devices with a 1 KB page size and 24-24-24 timing. Additionally, we were given a list of timing constraints which will be addressed in section E.

C. Memory Controller Design

- In order to maximize our potential grade on the project, the team chose to attempt the most challenging scheduling policy: access scheduling, possibly out of order.
- We implemented the queue as a statically-allocated array of structs in order to increase efficiency versus dynamic allocation.
- We implemented the queue as a circular linked list to allow for easy traversal of the queue. This also makes add/remove functions trivial.
- We fully encapsulated the controller functions within the controller in order to make the program as portable as possible. This will make testing easier.

D. Design Decisions

Scheduling Policy

The memory controller implements a first-ready-first-access scheduling policy. On every clock tick, it traverses the queue and looks at the status of each memory request to see if a command can be issued on behalf of the request. The scheduling of commands is done on a first-come- first-serve basis with requests at the head of the queue given highest priority. The first valid command scheduled ends the scheduler and it returns to send the command. The decision to schedule commands in this manner was a result of not having enough time to fully implement and test a more aggressive scheduler.

Page Policy

The memory controller page policy is a "modified" closed page policy. When a request has completed its READ/WRITE command, the controller looks to see if any other requests in the queue want to access the page. If there is another request in the queue that wants to use the page, it will leave the page open, otherwise it will close the page. We recognize the drawbacks to this approach given that there are 16 banks in the SDRAM and 16 slots in the queue. Optimized memory requests will want to interleave across bank groups. Meaning pages are probably getting closed too early. However, completeness of a solution was the top priority. The decision to implement this modified close page policy was the result of not having enough time to fully implement a more robust open-page policy.

Controlling When Requests are Read from the File

To account for requests being stacked up outside the queue due to the CPU sending them faster than the controller can process them, enqueuing requests will only happen if the pending request time is less than or equal to the master clock time. If the queue is full when a simulated CPU request comes in, the request is held in the "next req" data member and no further requests are read from the file. Having a single entry point allows us to control/debug requests as they come in. Having the structure statically allows us to avoid using malloc and simply just copy data when the member is added to the queue. One pitfall of this design is that scanf reads the last as garbage before returning end-of-file and that garbage gets added to the queue in an endless loop. We solved this problem by adding an extra end of file check before adding something to the queue. If end of file was reached, technically next_req has data in it, but it is garbage so we can just ignore it forever while the queue finishes.

Queue Design

The data members for the queue are statically allocated as an array of memory requests. The elements are then linked as a circular linked list with tail and head pointers. New requests are added to the tail and dequeued requests can be removed at the head or any item in the queue. The structure is statically allocated as an array to maximize runtime performance, as dynamic allocation is time consuming. The structure is implemented as a circular linked list for a few reasons:

1. Implementing the list in a circular fashion eliminates the needs for extra pointers to maintain the head and empty spots.
2. Implementing the list in a circular fashion eliminates the need to traverse the list and reorder the nodes when a spot becomes available in a previously full queue and then is filled again. This will probably be happening frequently since requests will be coming faster than the DRAM can process them and then could empty while the CPU gets its information from the cache.

Bridging CPU and DIMM Time Domains

In order to bridge the time domains, we think of the memory controller as operating in the faster CPU domain. On startup, the controller determines the ratio between the CPU and DIMM clock domains as set in the customizable defines. The multiplier is then applied to all DRAM timing specs when performing command analysis. The variable is stored as an integer type because the multiplier for this example is an integer. Floating point computations are time consuming and should not be used unless absolutely necessary if we are concerned about the operational performance of the simulator. Should the multiplier be a fractional number, the computer rounds down to the nearest integer and while the controller will not perform optimally it will still meet timing specs.

Controlling Command Timing

Each memory request in the queue keeps track of the last command it generated and the time since that command was generated. Each bank and bank group keep relative times for each type of command issued to that bank/group. The bank relative times and request relative times are used to check against the timing specifications to determine if a valid command can be sent. Bank relative times for each type of command are necessary because simply tracking the last command issued to a bank would not allow us to check for timing parameters that may span multiple commands.

E. Testing

Testing is divided into two parts: Functional Testing and Timing Testing.

1. Functional Testing

Test: **File I/O**

Description: Test that the files open properly and are written to properly with basic input strings. Test that the input file is opened and read properly by parsing the data and verifying it is correct.

Test: **Input File Format**

Description: Created test inputs with varied delimiters or spaces and tabs before, in between, after the request data to make sure information was extracted properly.

Test: **Output File Format**

Description: During functional testing a random input file was used to check the output format adhered to specifications. Each type of CPU request was used and the output was checked to make sure it was properly formatted with spaces.

Test: **Command Line Test**

Description: Test that the controller could accept command line arguments for file names and then open the files.

Test: **Queue Functionality**

Description: In order to test the queue functionality, a separate test program was created to allow the user to dynamically interact with the queue and test adding to empty queue, adding to full queue, removing from an empty queue, remove at head, removing at tail, removing at middle of queue, and queue traversal operations. Once satisfied with the operation, the queue was again tested as part of the memory controller with input files designed to simulate filling the queue, adding to full queue, emptying the queue, re-filling the queue, removing at head, removing last item of full queue, removing last item of non-empty queue, and removing intermediate item from full queue and non-empty queue.

Test: **Address Decoding**

Description: Created an excel spreadsheet to both encode and decode addresses. Input files were created with bit streams to test group/bank/row/columns. Bit patterns were chosen to highlight specific areas and to test for edge cases such as 101010101 and 1111000011110000.

Test: **Clock Management**

Description: **T**ested that the clock simulator jumps to the time of the next memory request if the queue is empty by designing an input file with the first request starting at time > 100.

Test: **Scheduler Test**

Description: Simple single request input file to make sure that the commands are scheduled properly and the program terminates properly. Each CPU instruction was tested.

Tests were expanded with input files designed to test the out-of-order execution and make sure that multiple requests were services one after the other.

Test: **Page Policy Test**

Description: Input files were created with consecutive addresses to the same bank/group/row to make sure the page stayed open and the subsequent addresses go the page hit. Input files were also created to insert requests to different bank/groups in between the requests to the same bank/group/row and verify the page would still stay open.

2. <u>Testing for Timing Parameters</u>

Each of the following scenarios was run through the controller to ensure that all timing parameters were being met:

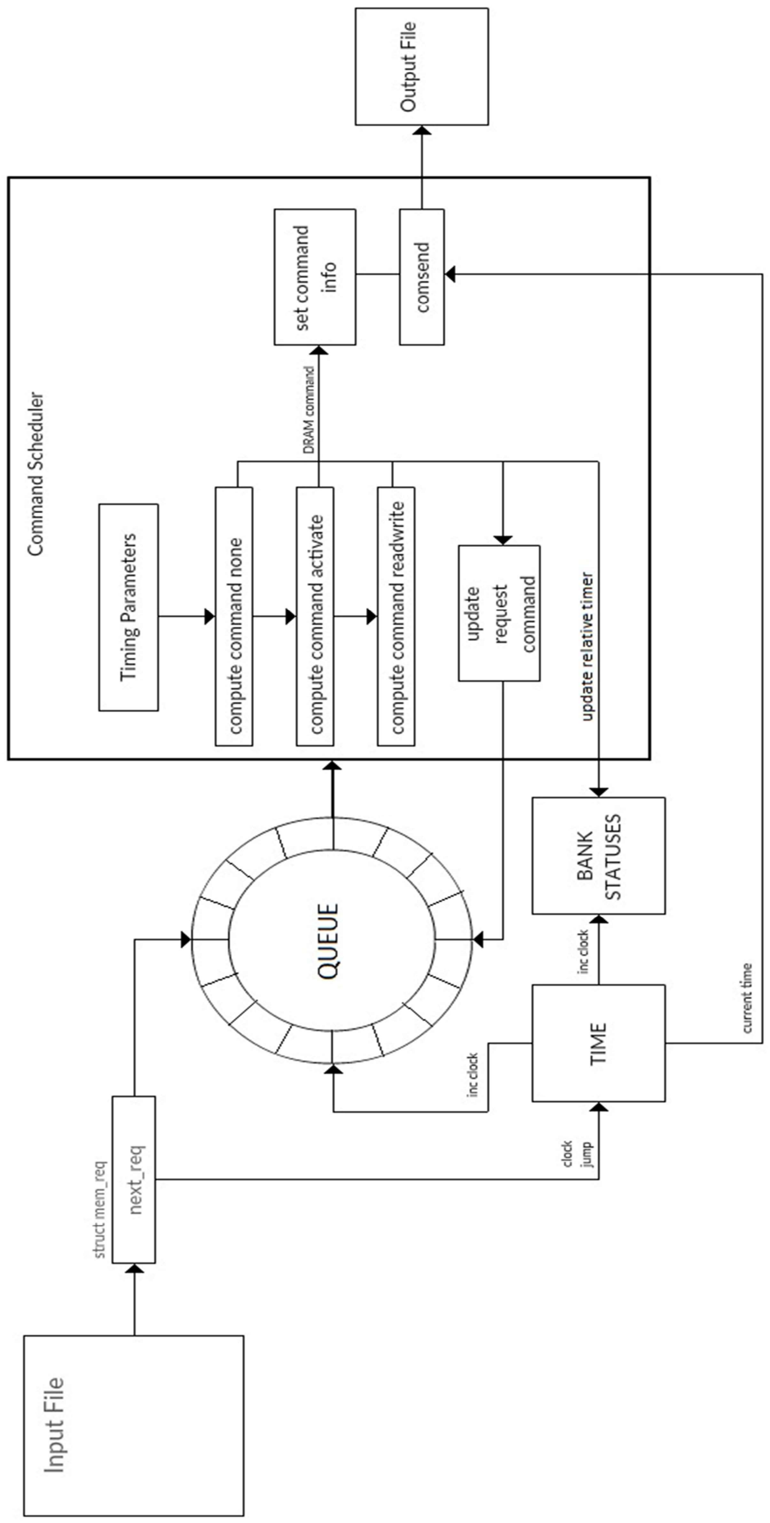
- Reads from contiguous locations
- Writes to contiguous locations
- Random reads and writes
- Multiple reads from an open page interrupted by reads from other locations
- Consecutive reads to the same bank
- Consecutive reads from different banks in the same bank group
- Consecutive reads from different banks in different bank groups
- Consecutive writes to the same bank
- Consecutive writes to different banks in the same bank group
- A read followed by a write in the same bank group
- A read followed by a write in a different bank group
- A write followed by a read in the same bank group
- A write followed by a read in a different bank group

F. Conclusion

Thus, a memory controller capable of serving the shared last level cache of a four core 3.2 GHz processor, employing a single memory channel, employing out of order access scheduling policy was successfully designed, simulated and verified.

G. Appendix

Exhibit 1: Functional Flow Chart of Controller


Input File
struct mem_req
next_req
clock jump
QUEUE
inc clock
TIME
inc clock
BANK STATUSES
current time
Command Scheduler
Timing Parameters
compute command none
compute command activate
compute command readwrite
update request command
update relative timer
DRAM command
set command info
comsend
Output File