Objective
In this lab, you will implement a more realistic memory hierarchy than in past labs. In particular, you will implement an L2 cache and a simple DRAM model. With these additions, your simulator will be able to more realistically model the performance of memory-intensive programs. As in Lab 5, we will fully specify the cycle-by-cycle behavior of the L2 cache and DRAM, and we will give reference cycle counts for several benchmarks.

Timing Specifications: L2 Cache and DRAM
Your job is to implement the following behavior in a cycle-accurate manner, starting with your Lab 5 simulator. All instruction and data cache and branch predictor behavior and specifications from Lab 5 (normal version, non-extra-credit) still apply to this lab.

You will implement a unified L2 cache. A unified L2 cache services cache misses from both the instruction and data caches. The cache allocates on stores and loads. However, for the purposes of this lab, we will not model writebacks, nor track “dirty” status on cache lines.

As we have discussed in class, while the instruction and data L1 caches are typically tightly integrated with the pipeline, the L2 cache is more decoupled. In this lab, we will maintain the semantics for the L1 caches that were described in Lab 5, but replace the constant 50-cycle memory access time with a variable-latency access to the memory hierarchy, which involves at least the L2 cache and may involve DRAM. Whenever the processor would have begun a 50-cycle memory access stall in Lab 5, it now issues a request to the memory hierarchy. Once a request is issued to L2, it cannot be canceled (even if the L1 request is canceled, e.g., as is specified in Lab 5 for the instruction cache on branch recoveries). When the data is available from the L2 or DRAM (as specified below), the memory hierarchy issues a “cache fill” notification back to the L1 caches. If the L1 cache is still stalling on the missing data (i.e., no cancellation occurred), the data is inserted into the cache and the stall ends on the next cycle. We now specify exactly how the memory hierarchy is structured and how requests are handled in more detail.

The L2 cache will be a 256KB 16-way-associative cache with 32 byte blocks. As with both level 1 caches, the L2 cache implements a true least-recently-used replacement policy within each set. Initially, all blocks in both instruction and data L1 caches, and in the L2 cache, are invalid (not present).

When a request is issued to L2, the L2 cache tags are probed immediately. If the requested data is present (an L2 hit), the request is completed (a cache fill notification is sent) after 15 cycles. (Assume that the L2 cache has enough ports so that accesses never contend with each other.) Otherwise, the request is an L2 miss.

The L2 cache maintains a set of 16 miss-status handling registers (MSHRs) to track in-progress L2 misses. Each MSHR has a bit indicating whether it is valid (in-use), a field indicating the associated cache block, and information about any associated memory request. When a request is first discovered to be an L2 miss, the L2 cache searches all valid MSHRs. If a request is already outstanding for the cache block that is being requested, the present request is a duplicate, and is ignored (the subsequent cache fill notification will satisfy the original request and this duplicate request). Otherwise, a new MSHR is allocated for the request.
In order to ensure that an MSHR is always available at this stage, if a level 1 cache experiences a miss and would have sent a request but all MSHRs are occupied, the cache miss must stall until an MSHR becomes available.

Once an L2 miss is allocated an MSHR, the L2 controller waits 5 cycles, and then sends a memory request to the memory request queue (whose behavior we describe below). This 5-cycle latency models an internal transfer of the request to the memory controller. Once the request has been sent to the memory controller, the L2 controller waits for the request to complete. When the request is complete, after 5 cycles of additional latency (for internal data transfer back to the caches), the new cache block is inserted into L2 (using the LRU policy), and a cache fill notification is sent to the L1 caches. If either the instruction or data L1 cache is stalling on the cache block that just returned from memory, the cache block is inserted into the respective L1 cache and the corresponding pipeline stage will no longer stall in the next cycle. The MSHR is freed at that point.

We will model main memory (DRAM) at the bank level. The main memory system has one channel (with a separate command/address bus to memory, and a data bus returning from memory) and one rank on that channel. The single rank has eight banks. The data is cache-block interleaved across banks: the bank number for a given address is determined by the 3 bits immediately above the cache block offset, i.e., bits \([7:5]\). A row is 8 KB in size. Thus, there are 256 cache blocks per row, and the column number for a cache block within a row is determined by bits \([15:8]\). The row number for a given address within a bank is given by bits \([31:16]\) of the address (we are assuming that there are enough rows to cover the full physical address space).

Each bank of the memory system tracks the currently-open row (if any). A different row can be open at each memory bank. Initially, no rows are open. When a memory access occurs, the access is classified either as a row hit (if the access is to the currently-open row at the corresponding bank), a row conflict (if another row is open at the bank), or a closed-row access (if no row is open at that bank). In each case, a different set of commands is necessary to access the requested data. These commands are:

<table>
<thead>
<tr>
<th>Request Case</th>
<th>Commands (Latency)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row Buffer Conflict</td>
<td>Precharge (100 cyc), Activate (100 cyc), Read (100 cyc), Data Transfer (50 cyc)</td>
</tr>
<tr>
<td>Row Buffer Hit</td>
<td>Read (100 cyc), Data Transfer (50 cyc)</td>
</tr>
<tr>
<td>Closed Row Buffer</td>
<td>Activate (100 cyc), Read (100 cyc), Data Transfer (50 cyc)</td>
</tr>
</tbody>
</table>

In all cases, the command bus is occupied for five cycles (because the memory bus is slower than the processor clock) at the beginning of Precharge, Activate, and Read commands (but not at any point during the data transfer for this particular access). The data bus is occupied for the full 50-cycle data transfer. The commands always happen back-to-back for a given request. Thus, for example, on a row hit, the command bus will be occupied for five cycles by the Read command; then for 95 cycles, the command will execute internally in the bank (during which the command bus may be used by other commands); then for 50 cycles, the data bus will be occupied as the data is transferred. The bank is considered “busy” for the entire period during which commands and the data transfer occur. Multiple requests can be handled concurrently as long as the requests do not need to use the command or data bus at the same time, and as long as they do not go to the same memory bank.

In each cycle, the memory controller scans the memory request queue (which we assume to have infinite size) for requests that are eligible to be sent to memory. A request is eligible to be sent to memory if the bank to which the request maps is not currently busy (serving another request). A request can be sent to a bank only if the command bus and data bus will be free at the appropriate cycles when they will be required for that request (your simulator should track in which cycles the command bus and data bus will
be free for a window into the future). When multiple requests can be sent to a given bank in a single cycle, the memory controller uses the FR-FCFS (First-Ready First-Come First-Served) scheduling policy to resolve the contention. Specifically, any request to the currently-open row at that bank (if a row is open) is chosen first; if not, any other request is chosen. Ties between requests are broken by their arrival order, i.e., a request created in an earlier cycle is chosen first. If two requests arrive in the same cycle and are not otherwise arbitrated by this policy, the request from the data cache takes priority over the request from the instruction cache. Once the next request to a bank is chosen, the bank’s status (currently open row) is updated, and the bank is kept busy for the number of cycles specified above. After this latency elapses, the request is complete, and L2 and L1 caches are filled as specified above (after the additional 5-cycle latency).

Getting Started
There is no lab handout directory for this lab. Instead, you should begin with your Lab 5 simulator (ensuring that it is correct), and extend it to model the L2 cache and DRAM as described above.

Handin and Other Requirements
As in previous labs, please hand in your lab electronically in your handin directory: /afs/ece/classes/ece447/handin/ANDREW-ID/lab6/.

You may modify any file in the simulator source (and/or add new files) except that you may not modify shell.c. We reserve the right to use a modified shell.c file in order to facilitate autograding.

Grading
As in previous labs, we will grade this lab objectively based on a set of tests which are not released beforehand. Your simulator must be functionally correct with respect to the Lab 1 golden simulator. Furthermore, your simulator should model timing accurately with respect to the cycle-by-cycle specifications in this lab document. We will, at our discretion, give some partial credit for results that are “close,” with a scale to be determined. However, you should aim to match cycle counts exactly.

Reference Timing Results
As in Lab 5, we are specifying several reference timing results here. Note that we may grade your submission using many other tests, for which your timing should also match. These timing results are given only in order to allow you to debug your implementation and verify that you are correctly modeling timing.

All of the benchmarks named here are available as part of the lab 5 distribution.

This set of tests is not meant to be exhaustive. You should write other tests that exercise the various aspects of the L1 and L2 caches and the memory (for example: write tests that have working sets entirely in L1, or entirely in L2, or stream through large memory arrays; and write tests that experience memory bank conflicts, row buffer conflicts and hits, bus contention, branch mispredictions at the same time as wrong-path misses, etc.), and examine cycle-by-cycle behavior, to make sure that you are modeling everything as specified in this lab document.

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Cycles</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand_tests/random3</td>
<td>45874</td>
<td>2104</td>
</tr>
<tr>
<td>brtests/test1</td>
<td>11728</td>
<td>1762</td>
</tr>
<tr>
<td>cache_tests/test1</td>
<td>1710789</td>
<td>393220</td>
</tr>
<tr>
<td>long_tests/fibonacci</td>
<td>5243186</td>
<td>5242885</td>
</tr>
<tr>
<td>long_tests/primes</td>
<td>2438146</td>
<td>2096285</td>
</tr>
</tbody>
</table>

Extra Credit: Prefetching
For extra credit, you may implement prefetching from main memory into the L2 cache. A prefetcher modeled in your simulator should create new requests at the L2 cache that are inserted into the MSHRs (in
the same way that L1 misses create requests) for cache blocks that it predicts will be needed in the future. The prefetcher is allowed only to observe the request stream that comes to the L2 from the L1 caches, and whether each request results in an L2 hit or an L2 miss. To try for this extra credit, you are free to implement any previously-proposed prefetcher or a prefetcher of your own design. We recommend you start with a simple stream prefetcher, e.g. [1][2]. You can also implement other prefetchers and make the designs more sophisticated, to optimize for execution time.

As before, extra credit will be based on a performance ranking among all submitted implementations of prefetching. You can earn up to 20% extra credit for the best three implementations (judged by IPC on several benchmarks). You will be eligible for extra credit only if you have turned in a fully-correct version of the required portion of the lab described above. Turn in your separate extra-credit simulator in the lab6-ec handin directory.

Because we are now using a C-based timing simulator, it is possible to “cheat” in your results if you are not careful (or if you are not honest!). At one extreme, the simulator could actually execute ahead in the program and find out which cache misses will occur, then prefetch exactly those cache blocks, all in in one cycle. Real hardware would not be able to do this. In general, it is possible to implement very complex algorithms in software that might not be possible to efficiently build in hardware (with a reasonable cycle time). For this extra credit, you should implement an algorithm that can be reasonably implemented with a small amount of storage and simple calculations that can be done in one or a few cycles. For example, a stride prefetcher can be built by tracking the last several cache miss addresses and using adders and comparators to determine whether a stride (regular distance between miss addresses) exists. The TAs and the instructor will be the final judges of whether a prefetcher design is acceptable. If there is any doubt, you should talk to the TAs about your extra-credit prefetcher design before the lab deadline (this is encouraged!).

References
