**Cache Simulator**

**Basic Specifications**

To start off, we would like to lay out some basic specifications that define the cache described in the project description:

Cache lines = 128k

Sets = 8k

Address bits = 32

Byte select bits = 6

Index select bits = 13

Tag select bits = 13

However, because our cache simulator asks the user for the number of sets, the cache line size in bytes, and the number of ways in each set, these values can change. The above values are only true when using 64 byte lines, 16 ways per set, and 8k total sets.

**Assumptions**

1. Our cache must keep our pseudo LRU bits and our MESIF state consistent with the higher level caches.
2. A read to the L1 instruction cache means the same thing as a read to the L1 data cache for our cache.
3. A read or a write to the L1 cache updates our cache state (tag bits, MESIF state, pseudo LRU bits).
4. Snooping a write should only occur in the invalid state and when another processors cache is evicting a dirty line.
5. We do not need to clear the byte offset bits in the address when doing a bus operation. The receiving/snooping devices will do this if they require it.

**Design Decisions**

A major design decision revolved around how the input trace file would be handled. In the end, we decided to write a Perl script that would pre-process the trace file and output a formatted trace file. The Perl script looks for lines with these two formats at the beginning of the line:

n address

n

If a line with the format of *n address* is found, the Perl script prints out the line to the formatted trace file with only one space between the trace operation and the address. If only *n* is found, the script writes the trace operation to the output file, appending a dummy address of 0 to match the format of *n address.* This ensures that our cache simulator can always read a trace file – even if it has comments!

Another major design decision revolved around allowing the user to decide the size of the cache. At the beginning of the simulation, the simulator will prompt the user for the number of sets (must be a power of 2), the line size in bytes (must be a power of 2), and the number of ways in a set (must be a power of 2 less than 128). If a number is entered for any of the parameters is not a power of 2, it will be rounded down to the nearest power of two. For example, entering 31 will result in 4 bits for the given parameter. The number of ways in set must be less than 128 because our pseudo LRU algorithm can only handle 64 ways in a set. This is because the largest data type available is a uint64\_t. To go into sets with more ways, we would have had to implement a complicated algorithm that involved using an array of uint64\_t variables. We instead decided to constrain the number of ways.

We also decided to implement the posted write buffer for extra credit. Our posted write buffer can hold 10 items total before it becomes full and must write a line back to DRAM. Every 5 trace operations, the posted write buffer writes the oldest item in the buffer back to DRAM. If a cache miss occurs while executing a trace operation, our simulator will first check the posted write buffer for the data before going out to DRAM. If a cache eviction occurs on a dirty cache line, our simulator will first check for a previous write to the same cache line in the write buffer. If the cache line is found in the write buffer, the old write will be discarded and the new write will be added into the write buffer. Also, on cache clear, the write buffer is also cleared by simulating writing all the items in the buffer back to DRAM.

**Verification**

To verify that our simulator functioned correctly, we used a few different testing techniques. Our first testing technique involved unit testing. For each sub-system of the cache (MESIF, pseudo LRU), we created a dummy main function that would test only it's own sub-module. Within the sub-modules, we wrote a battery of tests that would be called from the main function. The tests were designed to be as exhaustive as possible. For example, in the MESIF module, we went into every MESIF state and made sure that the correct transition happened when a trace operation occurred. To avoid having to check the output by hand, we used some extra debug variables and the assert library to make sure our output was correct for the given situation. If a test failed, the simulator would fail out and tell us the exact line that the failure occurred at.

Once all of the unit tests were passing for each sub-module, we hooked the trace decode logic up to the MESIF and pseudo LRU sub-modules. We then re-wrote all of the unit tests for each sub-module into separate trace files. By running through the units tests again in trace file form, we were able to identify bugs that were occurring in the decode logic, since the sub-modules themselves were proven correct in our previous tests.

Once the unit tests in trace form were passing, we created some new trace tests. First, we wrote a test that would test to see if the cache properly handled reads/writes along with hits/misses. If the number of reads/write and hits/misses didn't match what we expected at the end of the trace, we knew that there was an error.

We wrote two tests for our posted write buffer. The first test evicts a non-dirty line, confirming that the write buffer does not receive the non-dirty line. Then a dirty line is evicted and the write buffer is printed, showing the dirty line in the buffer. We then do a read to the same address as the evicted line to make sure that the cache reads from the write buffer and not DRAM. Finally, 3 more reads to the same address occur and then we do a print to confirm that the write buffer writes the line back to DRAM after 5 trace operations. Our second test checks to see if the write buffer successfully removes a currently present line when a new dirty line is written to the same address. The combination of these two tests should check all of the corner cases of the write buffers behavior.

**Pseudo LRU Implementation**

Pseudo LRU (P-LRU) models a balanced binary tree, where the root of the tree is bit 0 and bits increase from left to right each level at a time from top to bottom. To find the next bit to read or write you calculate bit number times 2 plus 2 (bit# \* 2 + 2) when going right and bit number times 2 plus 1 (bit# \* 2 + 1) when going left. The direction of the traversal is determined by computing the value that splits the tree into to two equal sections ( initial value of root is 0 and n=1). If it is greater than or equal to we clear the current bit or if it less than we set the current bit indicating that LRU must be in opposite direction of my current traversal. The process is repeated using the next bit as the new root of the tree, increasing n by 1, then root is calculated by if we go right or if we go left.

Finding a LRU is simple. We begin by having a variable (victim) equal to 0 then on each bit (or level ) we read we left shift only if the bit value is 0 or left shift and set to 1 if the bit value on the P-LRU bit is one.