**Cache Simulator**

**Basic Specifications**

To start off, we would like to lay out some basic specifications that define what our cache simulation looks like:

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, and the number of ways in each set, these values can change. The above values are only true when using 64 byte line sizes, 16 ways per set, and 8k total sets.

**Assumptions**

1. Our cache must keep our psuedo 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, psuedo LRU bits).
4. Snooping a write should only occur in the invalid state or when another processors cache is evicting a dirty line.

**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:

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, 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). The line size and the number of ways must be a power of two so that the address can be cleanly divided up in the byte select, index and tag bits. The number of ways in set must be less than 128 because our psuedo 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.

**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, psuedo LRU), we created a dummy main function that would function as a stand alone test for the module. Within the modules, we wrote a battery of unit tests that would be called from the main function. We wrote unit tests that would test the module in every possible situation that we could imagine. 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 unit test failed, the test would fail out and tell us the exact line that the failure occurred at.

Once all of the unit tests were passing for each module, we hooked the trace decode logic up to the MESIF and pseudo LRU modules. We then re-wrote all of the unit tests for each 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 the decode logic, since the sub-modules themselves were proven correct in our initial unit 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.