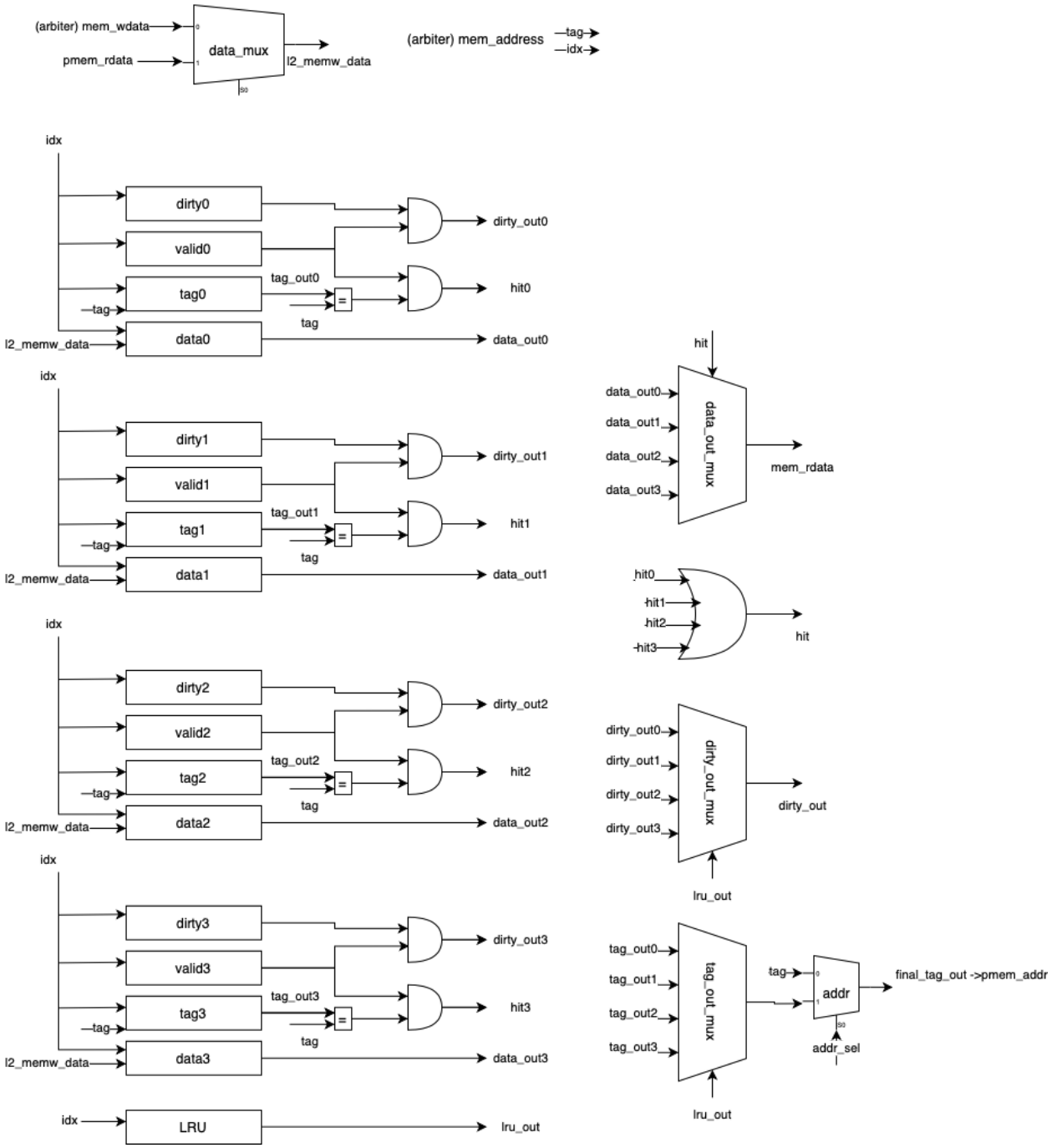
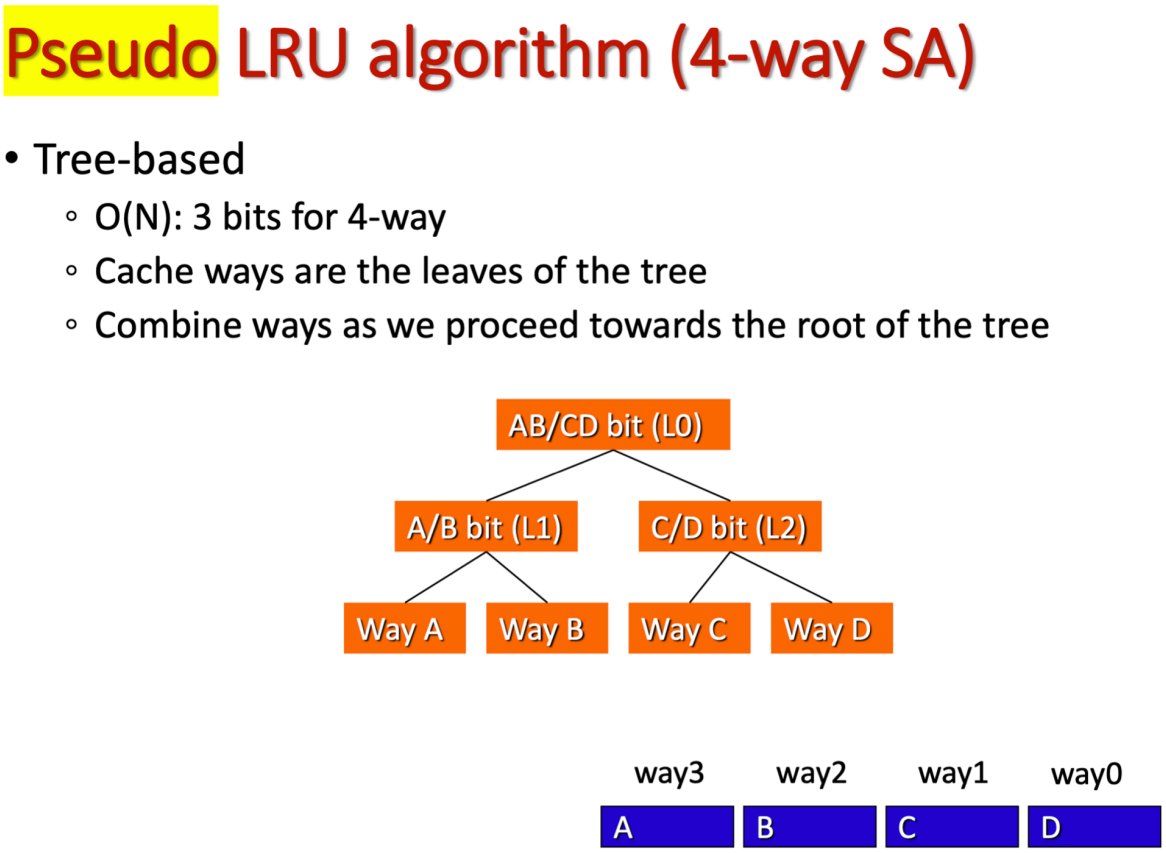
Advanced Features Performance: Parameterized L2 Cache



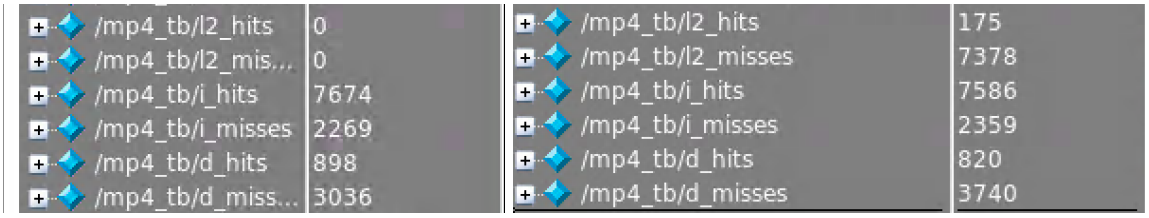
Implementation:

We implemented the parameterized way and set cache with our L2 Cache. Above we have the 4 way cache datapath, but we have since expanded this design to include up to 16 ways and 4 sets for the L2 Cache, depending on the parameters. The process to increase the ways and

sets involved parameterizing all of the muxes involved in the L2 Cache, as well as parameterizing the amount of dirty, valid, tag and data arrays used. We switched to a pseudo LRU cache, shown below as taken from lecture, and was able to parameterize the algorithm giving us log2(N)+1 bits for an N-way set associative cache.

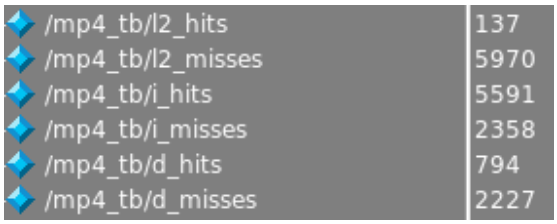


Performance:

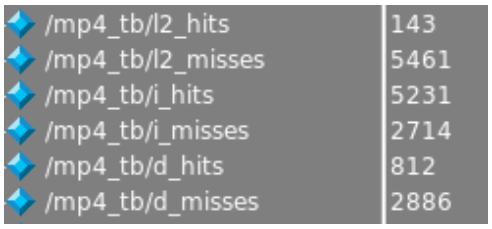


Above we have kept track of the hits and misses for the instruction cache, data cache, and l2 cache (when applicable) for 100000ns each run. On the left is the number of hits and misses for the i cache and d cache without the l2 cache implemented, and on the right is the number of hits and misses for the i cache, d cache and the l2 cache when we connected the L2 Cache between the arbiter and the physical memory. There seems to be an improvement in performance given the hit/miss count, as the amounts of reads and writes from the caches clearly increased with the implementation of the L2 cache as total hits and misses increased within the same time frame.

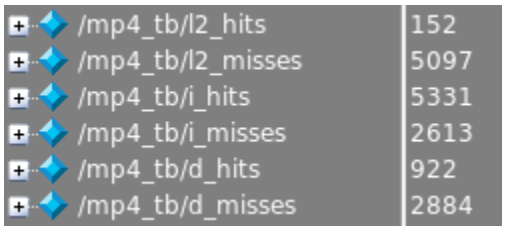
4 sets, 2 ways



8 sets, 4 ways



16 sets, 4 ways



We have kept track of the hits and misses for the parameterized L2 cache as shown above, with the respective parameters listed above the performance counts for each of the L2 caches. As we know, with an increased amount of sets within a cache, we may have greater hit time, as associativity increases. An 8 set cache or higher is categorized as fully associative, which in turn, results in the placement policy being slow as it takes time to iterate through all the lines, as well as power hungry as it has to iterate over the entire cache set to locate a block. It offers the most flexibility, but given the statistics as well as the theoretical ideas behind multiple set caches, a 4 set cache would offer the best performance.

Testing:

Testing for the cache mostly involved parameterizing, as the given directly mapped cache given to us for use in checkpoint 2 gave us a guideline in how to determine the effectiveness of the L2 cache. Testing the cache using the MP3 environment also helped determine the validity of our design, as the conversion from the main cache to the L2 cache was primitive, so testing within the MP3 environment helped with checking waveforms much easier than fully placing the L2 cache within the entire MP3, as it helped isolate the L2 cache and made it easier to pinpoint which signals caused errors within the cache.

In regards to edge cases, we edited some of the script files in order to see whether the implemented cache would be able to pick up on the certain stores and loads. We used continuous load hits, store hits, load then stores, and vice versa. Edge casing did not help us to

see the correct responses in the registers, however. Looking at the waveforms, and realizing that certain bits, like valid_out was set high when it should have been set low, helped us debug and correct our cache implementation, especially during parameterization where signals were easily mishandled as the set association grew bigger.