Structs & Data Structures

The primary struct variable in my project was a LINE. Each LINE object consisted of a tag bits line, a valid line, a search index to handle LRU/FIFO policy computations on set-associative caches, and a LINE pointer to the next LINE node in a potential linked list.

The primary data structures in my project consisted of arrays and linked lists. I also kept track of 4 global LINE pointer head variables in order to implement the policies.The 4 pointers represented the three caches and the FA simulator cache. The functions were identified as such:

direct-mapped cache- This implementation was designed with a one dimensional array, that comprised of the # of sets in the cache. Tag Mask ID's and Set Mask ID's were also computed by right shift and bitwise-and(&) operations to identify proper set and tag indices. Once the set index was computed, the array checked the set index line to see if it is a valid bit or not,. If it wasn't, then a cold miss was returned. If it was valid, but not a correct tag bit, then nothing was returned as a simulated FA cache was implemented to handle this case. Lastly, a hit was reported when the valid bits and tag bits were matching on the proper set index.

full-associative cache- This implementation also used a one dimensional array, but the size of it was based on the number of lines/blocks computed by cache size/block_size. This is because a FA cache has only one set.Tag Mask ID's were computed by right shift and bitwise-and(&) operations to identify proper tag indices. I also kept track of the remaining space in order to identify when a capacity miss could potentially happen, and to to keep track of the dequeue operations of the selected policy. If the policy was LRU, I also kept track of the case when a hit occurs in the cache. Cold misses were calculated just like in DM cache, which is checking if the specific line had a valid bit of 0. No additional simulations were performed, and the capacity miss calculation was from total misses-cold misses.

set-associative cache- This implementation used a two-dimensional array, where the the first dimension consisted of the number of sets in the cache, and the 2nd dimension consisted of the number of lines that each set consisted of. Tag Mask ID's and Set Mask ID's were also computed by right shift and bitwise-and(&) operations to identify proper set and tag indices. This was similarly tracked like the FA cache in order to identify cold misses, as well as identify capacity/conflict misses. The only change was that the 2d array had to be additionally managed by line number and set index. However, the biggest difference was the implementation of the LRU/FIFO policies because it had to keep track of the set indices that were being enqueued and dequeued from the functions.

setenq and enqueue- This was a singular linked list implementation that handled the enqueueing of LINE nodes that were a cold miss in the FA or SA caches, or also when an item was dequeued and the new item had to be inserted into the queue. The first/least recently used item was at the head of the list, while the last/most recently used item was at the end of the list.

In setenq, I also had to maintain the incoming set indices that were being inserted on to the list. I arranged all the sets in their own most/recently used order in order to maintain the dequeuing of the first set index.

setdequeue and dequeue- This used the same linked list from enqueue, but the function was responsible for returning the tag bits of the head of the linked list. In setdequeue's case, the linked list had to find the first identical set index in order to return the tag bits, and then it had to move the previous node's next into that set index's next.

setLRUhit and LRUhit- This operation also used the global linked list, and handled the operation of moving the tag bits that were a hit to the end of the list for that specific set index. This was done in order to identify the incoming tag as the most-recently used index in that specific set. In the set associative cache case, the tag bit was identified, and then it had to be moved to the end of the last similar set index.

Calculation of MIsses and HIts

Cold Mises were identified when the valid bit of a LINE node was 0. Capacity and Conflict misses were indirectly calculated at the end of the simulation.

A total miss was incremented from a cold miss, or when a potential conflict/capacity miss occurred.

A hit was incremented when matching tag and valid bits were identified by the cache data structure.

In SA and DM caches, a FA cache simulation had to be performed in order to compute the capacity misses. Once that was identified, the total misses off the SA or DM Caches were subtracted from the cold misses and capacity misses in order to find the total conflict misses.

In a FA cache, no conflict misses were computed and the capacity misses were calculated from the subtraction of the total misses from the cold misses.