# CSE 140 Project 2 Answers



A) If we assume that total is being stored into a register, then line 3 would fill the entire cache with values from the array. Since the array is 64B long and the cache is also the same size, the cache would be 100% filled with the array. This means that every time we want to fetch the information from the array, it is in the cache and will result in a 100% hit rate.

B) The same thing as A would occur, the entire array would fit into the cache and would result in a 100% hit rate. No blocks would be missing from the cache, as they would be placed there on line 3.

C) Since the offset starts at a value of 1, we are unsure of what the typical behavior for the cache is. This appears to be considered misaligned memory access and different systems will handle it differently. I can only assume that the cpu can handle the misaligned access and will treat the cache like it treats the cache in A or B with 100% hit rate with some time penalty.

D) Same as with C, we can only assume that the CPU will handle the misaligned data and place it into the cache correctly, similar to A and B. This assumption would lead to a 100% hit rate.

Aside: If our assumption about total not being stored in cache is incorrect, and it indeed is stored, then the hit rate would not be 100% and would depend on the block replacement policy and associativity. Since we don’t know where the total would be stored on the cache if it was, we don't know exactly what the hit/miss rate would be. In DM, I would assume there would be 1 miss where the total took storage in the cache, and the rest are hits, so 63/64. For fully associative, this could result in a hit of 3/4 ratio if the LRU replacement is used. Same answers for C and D.

E) If our assumptions are correct, then it isn’t very surprising that this would occur given the array size and the cache size. Since we think total would be stored in something like a register to do all the operations, then everything should be in the cache. If total is stored in the cache, then the hit/miss would change with replacement policy, with 2 way assoc making a ration of 62/64 and 4 way with a ratio of 60/64 using LRU.

1. In a write-through cache, when a new entry is updated within a cache block, it is also updated in memory. This allows for easily implemented architecture and ensures both the cache and memory are coordinated and up-to-date; however, this can cause the processor to become held up on writes.

In a write-back cache, when a word is updated in a cache block, the corresponding block in memory can become “stale” or outdated. To account for this, each block is given a “dirty bit” to indicate if memory needs to be updated when the block is replaced. This alleviates the strain on the processor for repeated writes, however, it creates more complex reads and misses may require the write-back of dirty data.

A write-back cache may be preferred over a write-through cache in the interest of speed and efficiency; but if data hazards are a major concern, a write-through cache may be preferable since both the cache and memory will always be in sync.

1. A 2-way set associative cache could outperform in the event the caches are full. In this case, when accessing the 2-way set associative cache, it only needs to search two blocks to discover if an entry is a hit or a miss; this makes the cache perform faster than if it had to check each individual block tag as it would in a fully associative cache.
2. To implement an L2 cache, once we discovered there was a miss in the L1 cache, we would simply have to search the L2 cache for the desired tag/index block. This could be implemented identically to the way we have implemented our L1 cache; the only other changes we would need to make is to move the memcpy calls from the L1 cache miss to the L2 cache so we don’t load the data from memory into the L1 cache before searching the L2 cache.
3. If virtual memory was included, we would need a pointer to virtual memory’s address in order to check the TLB for the address translation to the physical address. We would also require the pointer to data, as well as the WriteEnable variable. Additionally we would need pointers to the the page table for both the L1 cache and virtual memory so we can update/check them properly as the reads and writes are performed.