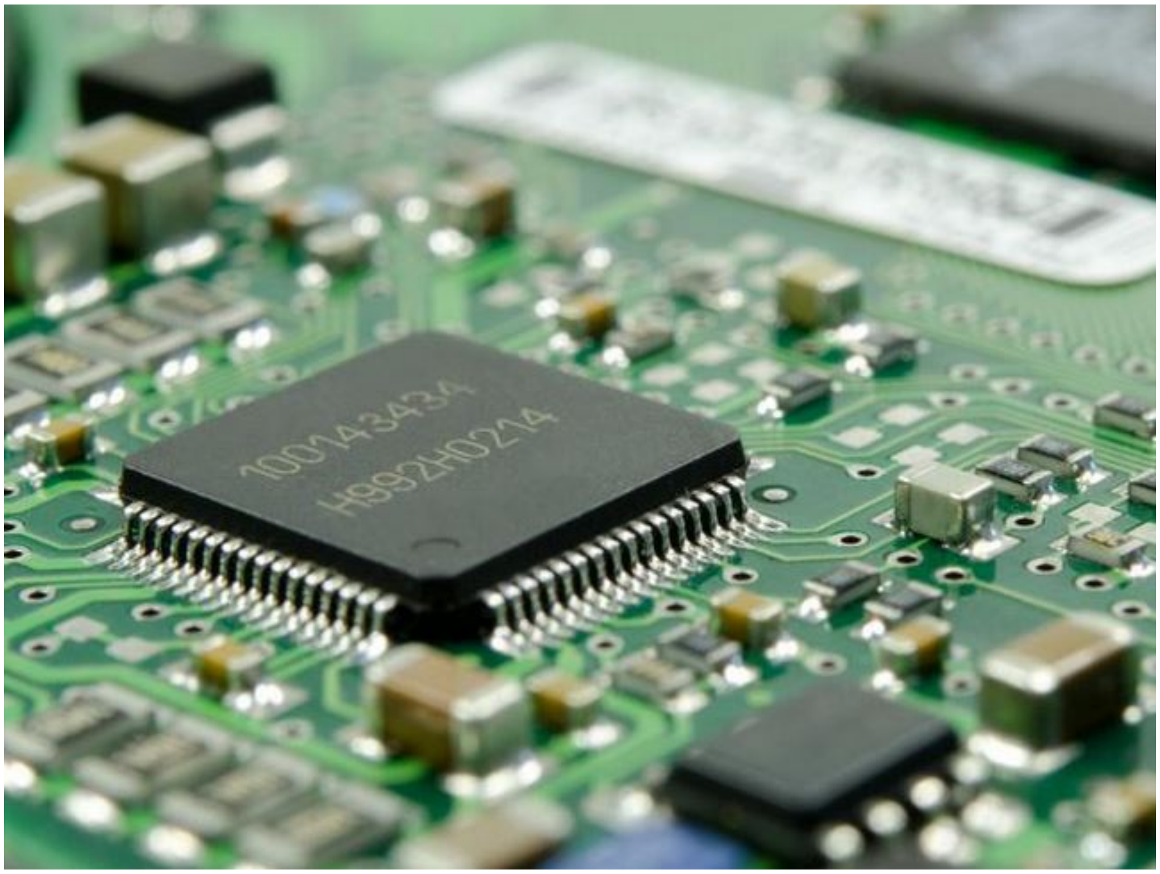
CACHE SIMULATOR

Assembly Project 2

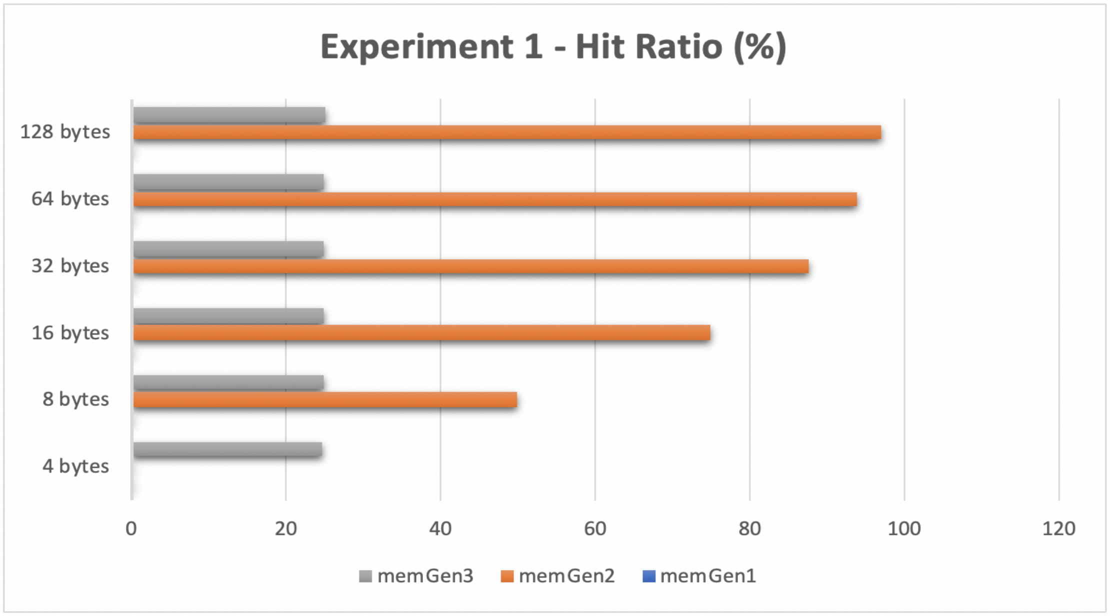


Sara Ahmed
Ramy ElGendi

EXPERIMENT 1

DIRECT CACHE

CACHE SIZE	BLOCK SIZE		HITS	MISSES	HIT RATIO (%)
64 KB	4 bytes	memGen1	0	1000000	0
		memGen2	0	1000000	0
		memGen3	246497	753503	24.6497
	8 bytes	memGen1	0	1000000	0
		memGen2	499999	500001	49.9999
		memGen3	248262	751738	24.8262
	16 bytes	memGen1	0	1000000	0
		memGen2	749999	250001	74.9999
		memGen3	248539	751461	24.8539
	32 bytes	memGen1	0	1000000	0
		memGen2	874999	125001	87.4999
		memGen3	248704	751296	24.8704
	64 bytes	memGen1	0	1000000	0
		memGen2	937499	62501	93.7499
		memGen3	249038	750962	24.9038
	128 bytes	memGen1	0	1000000	0
		memGen2	968749	31251	96.8749
		memGen3	250271	749729	25.0271



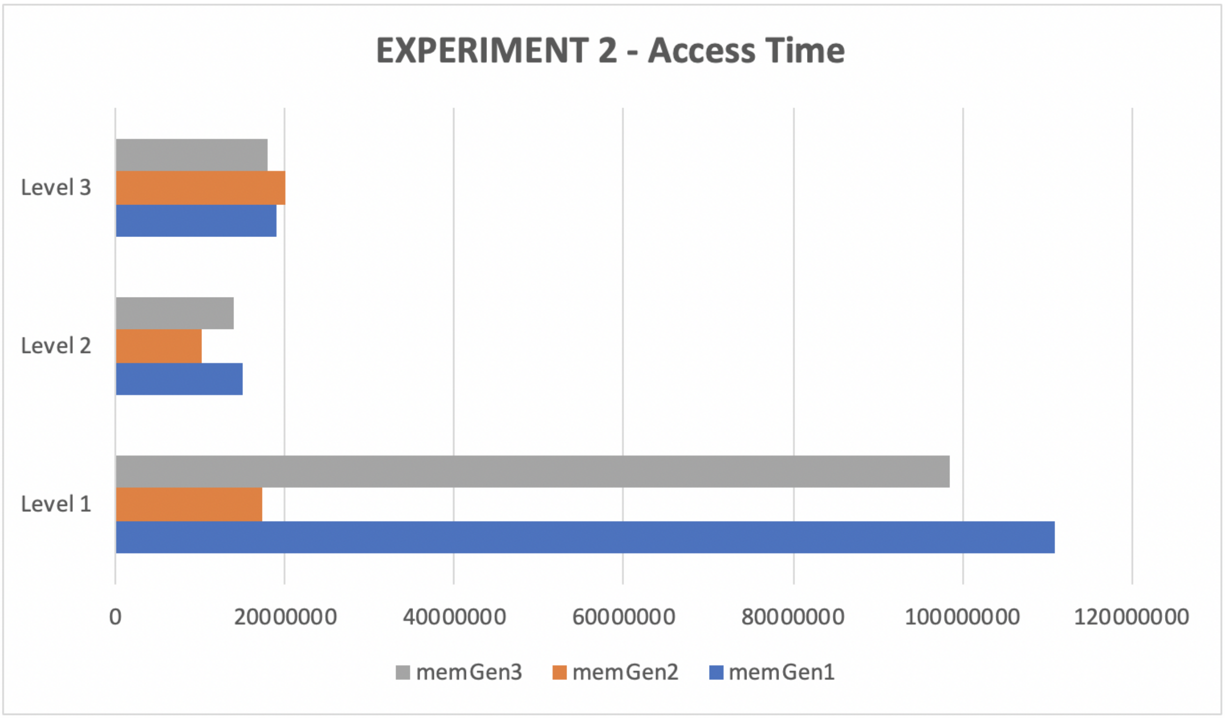
COMMENTS:

- The direct caching had when the block size was 4, a 0% hit ratio in the first 2 memGen() functions. This is due mainly to the fact that the there were conflict misses, due to the index resembling each other (0x00, ..), which resulted in a 100% miss rate.
- The random produced random addresses, which lessened the hit ratio a bit, as the addresses weren't so ordered like in memGen1() 1, and memGen2(). Although the cache size in blocks was quite big, due to the nature of direct cache, the addresses were fighting over mainly one block, the one with the tag (0x00).
- However, when the block size increased to 8 bytes, we found that memGen2(), actually produced quite a lot of hits. It alternated between hit or miss, with the first 2 addresses producing a miss. This, I found, was mainly due to the fact that the block size is capable of taking, 4 bytes or 2 words. Since the memory addresses in menGen2() operate in fours, the first 2 ended up being compulsory misses, as the cache was empty, but it went on to alternating after that.
- memGen1(), on the other hand produced a hit ratio of 0%, which is largely due to the fact that the addresses weren't generated in fours, so although the block size increased, the addresses generated didn't mirror that. This trend can actually be seen throughout due to the nature of the addresses. MenGen3() is randomized so it's rather hard to predict, but it can be seen that the hit ratio does increase with the size of the blocks, however small the changes in the hit ratio are.

EXPERIMENT 2

DIRECT CACHE

Block size	Cache Size	Levels	Access Time	
64 bytes	L1: 32 KB	1	memGen1	111,000,000
			memGen2	17,250,100
			memGen3	98,532,300
	L1: 32KB L2: 256 KB	2	memGen1	15,001,600
			memGen2	10,936,211
			memGen3	14,038,153
	L1: 32KB L2: 256 KB L3: 8192 KB	3	memGen1	15,001,980
			memGen2	12,810,621
			memGen3	13,751,433



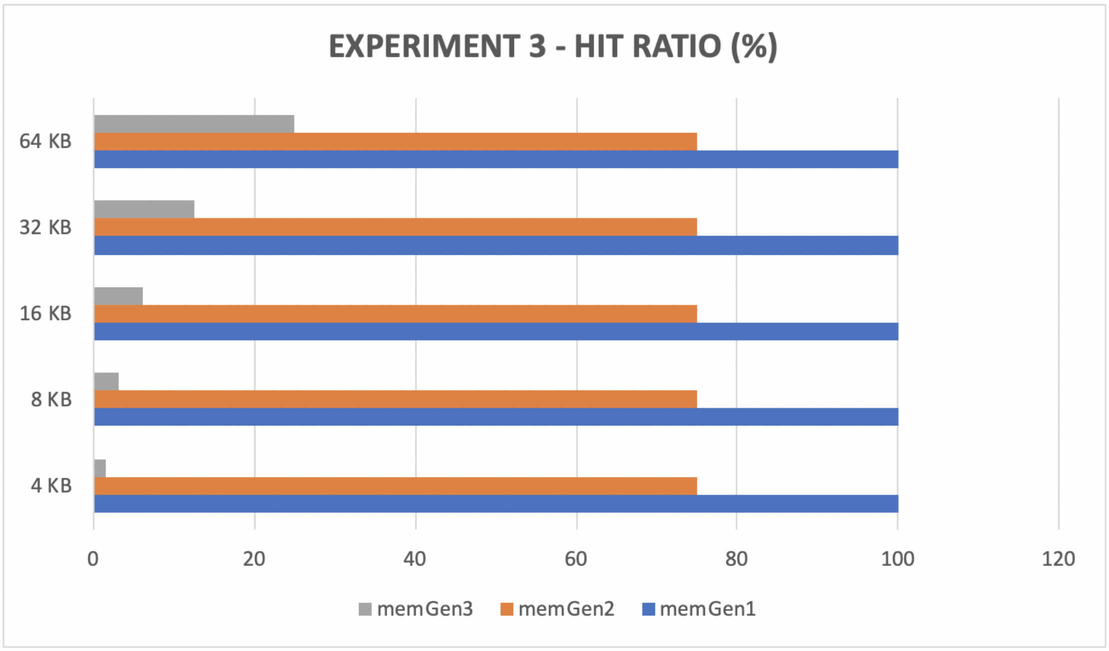
COMMENTS:

- The Since this is a multilevel cache, we had to make quite a lot of assumptions to make it work.
- We assumed that if, a block is not found, it stores it in the cache it is currently searching in. For example, if the cache is 2 levels, and it was not found in the first level, it stores the tag in the first before going to retrieve the rest of the information from the second. We did that to fill up the first cache to lessen the number of cycles needed to go and retrieve the next cache from the cycle. When a miss is had the cycles increases, if not, the cycles remain as the one with the initial cache.
- We learned that the access time can give us quite a lot of information about what is the most efficient cache to implement, as we can see what level the cache operates in its fastest. The capacity of the same cache in different levels are the same since they have the same block size.
- The first level of cache is the slowest, as expected, as its only 1 level of cache that is relatively small, due to it being the closest to the CPU. While it is the slowest the worst memory generator is MemGen1(), as it has a hit rate of 0%, meaning the miss penalty is added 100% of the time. Although the block size here is quite large, the addresses are not generated in four’s, forcing a 100% miss rate, as there is constant conflict with the tags.
- The access time of the second MemGen() function is much smaller, which is due to the large block size, and the addresses being in four’s. Now as the cache levels increase, the access time for the memory functions decreases. This is quite obvious as the increase in levels allows for more places to place the addresses, especially in direct caching. It also shields the cache from having to go into the main memory which takes the most cycles 100. As per our findings, we found that a 2 level cache was ideal, especially seeing as though it helped with the problem discussed earlier with the MemGen1(), as it protected it from the fatal 100% miss ratio, and it didn't have the awful miss penalty for the level 3 cache, 30 cycles.
- All in all, we discovered that the level 2 cache is ideal for a problem such as this.

EXPERIMENT 3

FULLY ASSOCIATIVE CACHE

BLOCK SIZE	CACHE SIZE		HITS	MISSES	HIT RATIO (%)
16 bytes	4 KB	memGen1	999984	16	99.9984%
		memGen2	749999	250001	74.9999%
		memGen3	15545	984455	1.5545%
	8 KB	memGen1	999984	16	99.9984%
		memGen2	749999	250001	74.9999%
		memGen3	31030	968970	3.103%
	16 KB	memGen1	999984	16	99.9984%
		memGen2	749999	250001	74.9999%
		memGen3	62108	937892	6.2108%
	32 KB	memGen1	999984	16	99.9984%
		memGen2	749999	250001	74.9999%
		memGen3	124787	875213	12.4787%
	64 KB	memGen1	999984	16	99.9984%
		memGen2	749999	250001	74.9999%
		memGen3	249038	750962	24.9038%



COMMENTS:

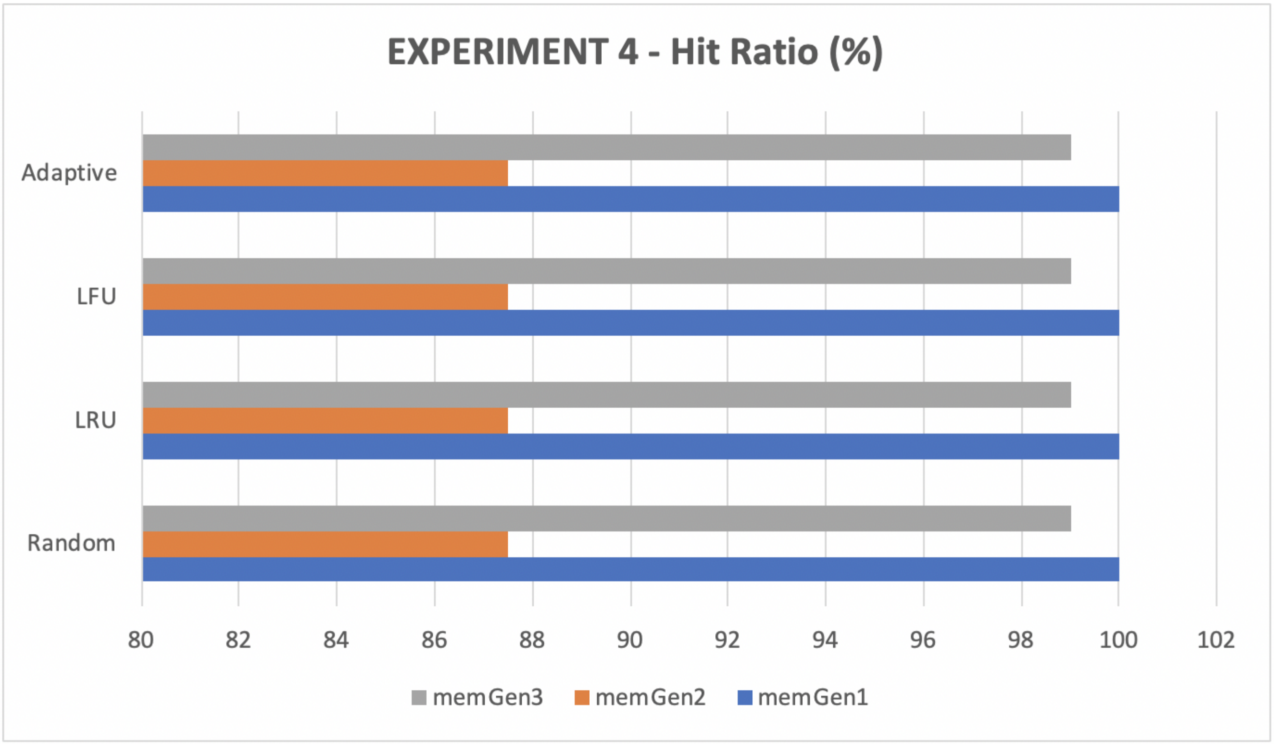
- The fully associative cache is one that stores the memory in whichever slot is available. In that case, it requires a lot of replacement policies, which we have implemented quite a few in experiments 3 and 4.

- The behavior of the fully associative is quite interesting, but very expected. We have concluded by looking at the table, that for MemGen1(), and MemGen2(), the hit ratio don't change.
- For MemGen1(), we find that the first 16 are misses while the rest are hits. This is due to the fact that the addresses are not in fours, so the first 16 will be compulsory misses, however we find that the rest are hits, due to the fact that those same 16 addresses, are repeated, and since they have been stored in the cache they will become hits.
- In MemGen2(), we can see that the hits are as follows (Miss, 3 hits, Miss), however the first iteration of that we see (Miss, 2 hits, Miss). That is due to the fact that the addresses dont start at 0 hex, but rather at 4 hex, and the block started taking from 0 hex, making it 2 hits instead of 3.
- The hit ratio of the MemGen3() increases as the cache size increases, due to the fact that MemGen3() is randomized, so as the size increases there are more blocks allowing more addresses to be added into the cache, so basically it allows for a bigger pool to be chosen from.

EXPERIMENT 4

FULLY ASSOCIATIVE CACHE

CACHE SIZE	BLOCK SIZE	REPLACEMENT POLICIES		HITS	MISSES	HIT RATIO (%)
256 KB	32 bytes	Random	memGen1	999984	16	99.9984%
			memGen2	874999	125001	87.4999%
			memGen3	991808	8192	99.1808%
		LRU	memGen1	999984	16	99.9984%
			memGen2	874999	125001	87.4999%
			memGen3	991808	8192	99.1808%
		LFU	memGen1	999984	16	99.9984%
			memGen2	874999	125001	87.4999%
			memGen3	991808	8192	99.1808%
		Adaptive	memGen1	999984	16	99.9984%
			memGen2	874999	125001	87.4999%
			memGen3	991808	8192	99.1808%



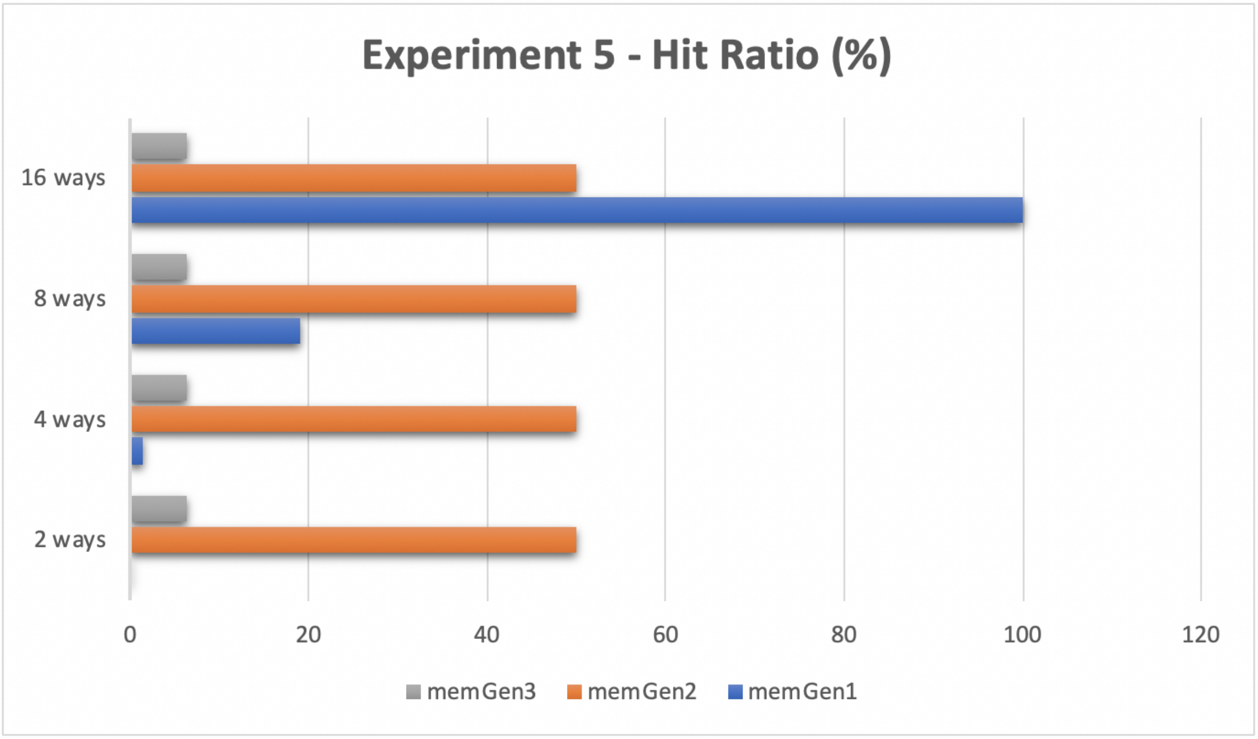
COMMENTS:

- The behavior of the fully associative is quite interesting, but very expected. We have concluded by looking at the table, that for MemGen1(), and MemGen2(), the hit ratio don’t change.
- For MemGen1(), we find that the first 16 are misses while the rest are hits. This is due to the fact that the addresses are not in fours, so the first 16 will be compulsory misses, however we find that the rest are hits, due to the fact that those same 16 addresses, are repeated, and since they have been stored in the cache they will become hits.
- In MemGen2(), we can see that the hits are as follows (Miss, 6 hits, Miss), however the first iteration of that we see (Miss, 7 hits, Miss). That is due to the fact that the addresses dont start at 0 hex, but rather at 4 hex, and the block started taking from 0 hex, making it 2 hits instead of 3.
- The hit ratio of the MemGen3() remains constant as the cache size increases, due to the fact that MemGen3() is randomized, because the block size, and the cache size remains the same so the replacement policy in that sense has little effect on the hit ratio.
- The hit ratio of the Adaptive replacement policy also remains constant, probably due to the assumptions we made, (changes from LRU to LFU when the hit ratio exceeds ½) , which it never does, due to the fact that the hit ratio in LRU, never goes below ½ , so in this specific example, it acts exactly like an LRU replacement policy would.

EXPERIMENT 5

SET ASSOCIATIVE CACHE

CACHE SIZE	BLOCK SIZE	# OF WAYS		MISSES	HIT RATIO (%)
16 KB	**8 bytes**	**2**	memGen1	99961	0.0037
			memGen2	497953	49.9999
			memGen3	935551	6.2401
		4	memGen1	986315	1.3681
			memGen2	497953	49.9999
			memGen3	935650	6.2302
		8	memGen1	808962	19.103
			memGen2	497953	49.9999
			memGen3	935538	6.2414
		16	memGen1	0	99.9984
			memGen2	497953	49.9999
			memGen3	935351	6.2601



COMMENTS:

- In the set associative, we were able to simulate the different ways the sets are arranged.
- We saw that as the set size increased so did the hit ratio, this is due to the fact that the capacity misses in each sets decrease, as the memory address is placed in any empty cell in the set, and as the number of ways increase, so does the number of lines in the cache.
- As shown above, memGen2() has a stable hit ratio, and that is partially due to its addressing being done in fours, or sequentially. That means that they are usually all assigned to the same cache set, when that occurs. While the number of blocks remains the same, what would really matter in the case of memGen2() is that the block size can take just more than 2 words. That is mirrored in the hits and misses, as it starts off with 2 misses, as the addresses don't start from 0x0, and then continues to alternate between 1 hit, and 1 miss. This is due to the fact that the blocks are collected in fours, and the memory addresses, are also in fours, so the set associativity doesn't really affect it.
- What it does significantly affect is memGen1(), due to the fact that the memory addresses are not addressed in fours, but rather in thousands, and the memory addresses are repeated again, so it matters where they are placed, and how big the capacity is to hold them.
- That is shown in the number of ways; as their increase there is significantly more blocks in each set that can store it. The first one has all misses, so a hit ratio of 0%, as even the repeated sets were quickly overwritten even though there were other empty sets in the cache, however when we increased the no, of ways it improved drastically.
- memGen3(), was random, but seemed to have a small hit ratio. Its due to the fact that the addresses all were addressed to the same index as they all had 0x00 hex, so even with the 2 word block, it didn't have much overlap in that sense, and I had to monitor it for so long to even see it produce one random hit. The capacity wasn't there, and since the addresses were in random, and unlike MemGen1(), and MemGen2(), in fours or repeated we saw little improvement with the set associative cache.