
**Project 2 Report**

**CSCE 2303: Computer Organization & Assembly Language Programming**

**Summer 2023**

**Dr. Mohamed Shalan**

**Group 3:**

Kirolous Fouty (900212444)

Omar Ibrahim (900192095)

Raafat El Saeed (900191080)

Yehia Ragab (900204888)

Ziad Amin (900201190)

**Project Introduction**

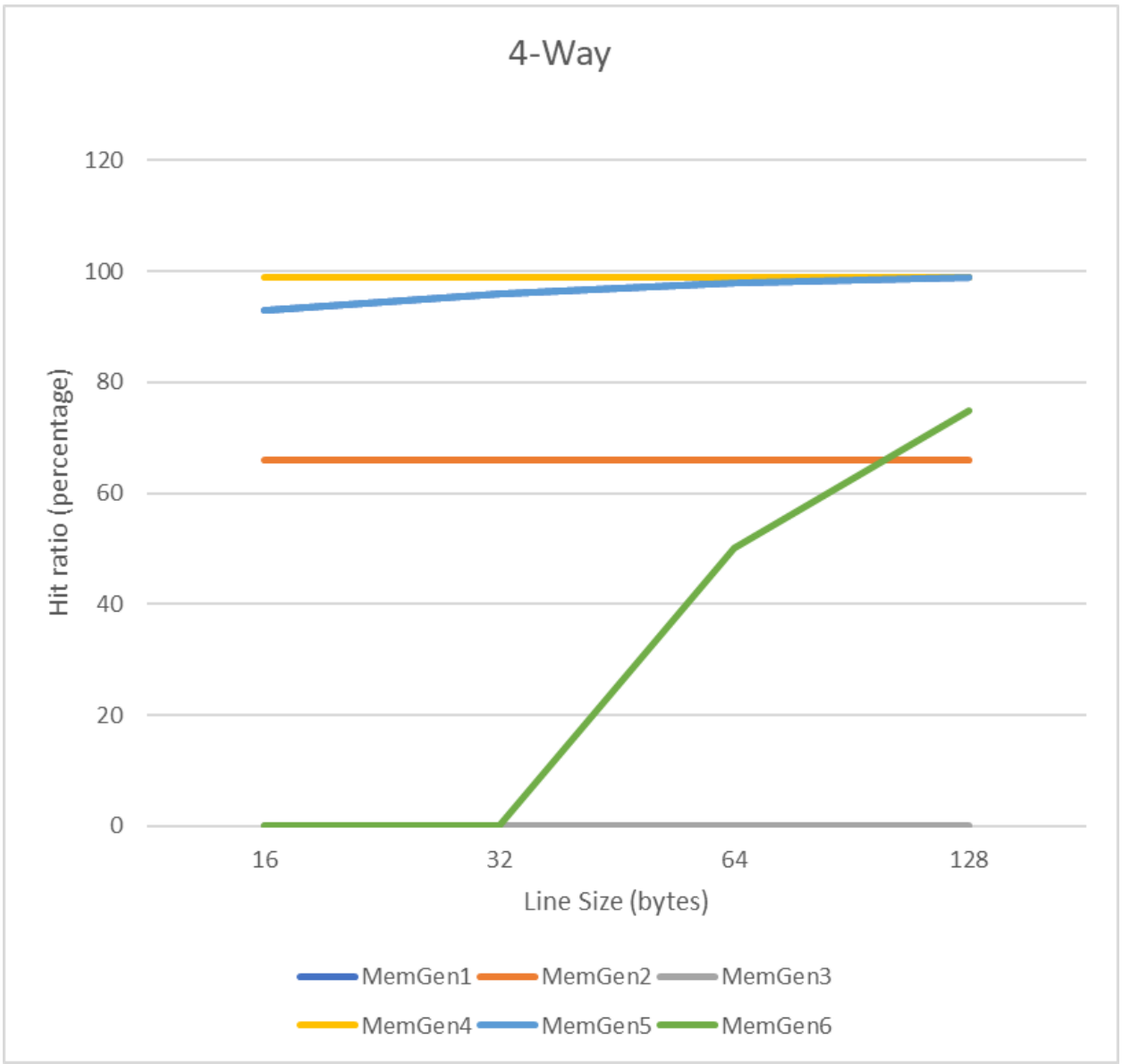
The primary objective of this project is to meticulously simulate an n-way set associative cache which is a crucial component of computer systems. Set-associative caches combine some of the advantages of fully associative and direct-mapped caches in a middle ground. A set-associative cache consists of several sets, each of which can hold a number of cache lines or blocks. Throughout this project, we analyzed the performance of n-way set associative cache and the impact of different factors on the corresponding hit/miss ratio. This project mainly fixes the cache size and assumes variable entries for the cache line size as well as the number of ways of identifying the impact on the hit/miss ratio for each combination of parameters. The main goal of the project is to analyze the final data results by deducing how to keep the number of ways constant and then vary the line size to see how this affects the hit time ratio. Moreover, the same comparison is done when setting the line size as a constant and then varying the number of ways. While the project accommodates different values for the cache line size and the number of ways, we conducted two main experiments. The first is the one in which we fixed the number of ways at 4 and measured the impact of changing the line size on the hit/miss ratio for each of the provided memGen functions. The second experiment was the one in which we fixed the line size at 32 bytes and measured the impact of changing the number of ways on the hit/miss ratio for each of the provided memGen functions.

**Background Related Information on Caches Behavior**

Before proceeding with conducting the above-mentioned experiments, we decided to briefly review the organization of set-associative caches and their benefits over direct mapping and fully associative caches. Firstly, in terms of organization, set-associative caches are organized into sets, each containing a fixed number of cache lines. The number of sets is usually determined in powers of two. Each memory block then can be placed in any of the cache lines within a specific set. Secondly, when considering cache mapping, in a directly-mapped cache, each memory block is mapped to a specific cache line based on the block's address modulo the total number of cache lines while in a fully associative cache, each memory block can be placed in any cache line, allowing for flexible mapping. This presents certain advantages and disadvantages that aim to be tackled with the set-associative cache. The main disadvantage of direct-mapping is that it is not flexible as a lot of memory addresses are mapped to the same cache block and thus cache misses due to conflict could be very frequent. The fully associative cache, however, has zero conflict misses as it compares the cache tags of all cache entries in parallel. However, the main disadvantage of that is that it is very expensive as all cache’s entries must be searched so it is indeed only practical for a very small number of cache blocks. Accordingly, the set-associative cache aimed to tackle that by acting as a middle ground between the direct mapping and the fully associative mapping. In a set-associative cache, a memory block can be mapped to any cache line within a specific set. In other words, there are fixed locations where a block can be placed, allowing for more flexibility than direct mapping and less expensive than fully associative caches.

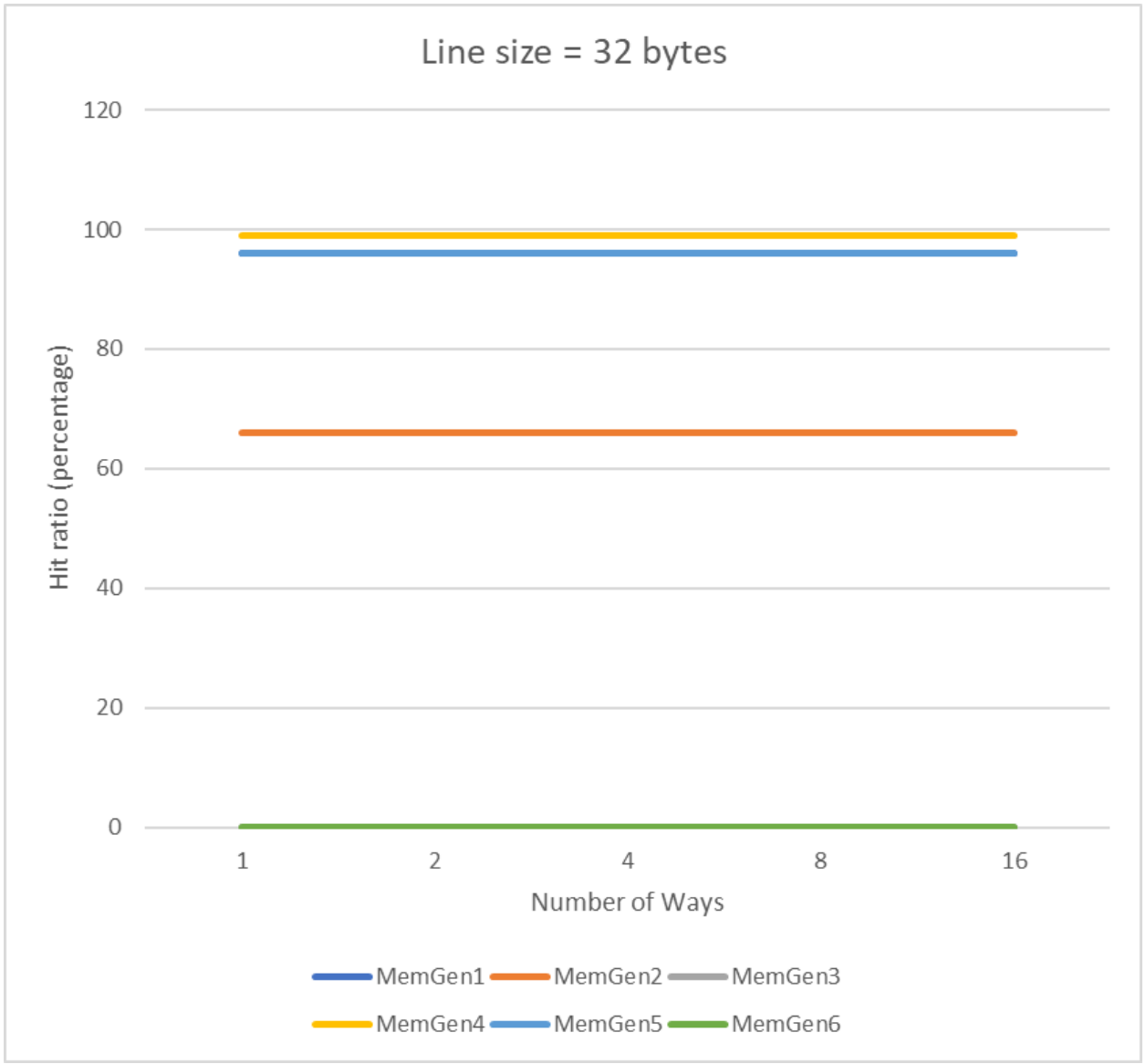
**Experiment 1 Results:**

| | Hit Ratio (%) | | | | | |
|---|---|---|---|---|---|---|
| **Line Size (Bytes)** | **MemGen1** | **MemGen2** | **MemGen3** | **MemGen4** | **MemGen5** | **MemGen6** |
| **16** | 93 | 66 | 0 | 99 | 93 | 0 |
| **32** | 96 | 66 | 0 | 99 | 96 | 0 |
| **64** | 98 | 66 | 0 | 99 | 98 | 50 |
| **128** | 99 | 66 | 0 | 99 | 99 | 75 |



**Experiment 2 Results:**

| | Hit Ratio (%) | | | | | |
|---|---|---|---|---|---|---|
| Number of Ways | MemGen1 | MemGen2 | MemGen3 | MemGen4 | MemGen5 | MemGen6 |
| 1 | 96 | 66 | 0 | 99 | 96 | 0 |
| 2 | 96 | 66 | 0 | 99 | 96 | 0 |
| 4 | 96 | 66 | 0 | 99 | 96 | 0 |
| 8 | 96 | 66 | 0 | 99 | 96 | 0 |
| 16 | 96 | 66 | 0 | 99 | 96 | 0 |



**Analysis & Conclusions:**

Firstly, observing the results of the first experiment, we can generally see that for most of the memGen functions provided, we see an improvement in the hit ratio as we increase the line size. This is mainly reflected in memGen1, memGen5 and memGen6 while the other 3 memGen functions maintain a constant hit ratio for varying line size. Accordingly we can observe that whether or not the line size affects the hit ratio largely depends on the list and sequence of addresses being passed to the cache and given that each memGen function generates numbers in a different manner or order, we see different behaviors in the outcome. Each memGen function will be further analyzed below in order to identify why it did or did not cause an increase in the hit ratio in experiment 1.

Secondly, observing the results of the second experiment, we found significantly surprising results in which the hit ratio remained constant regardless of the number of ways for all 6 of the memGen functions. This seemed like abnormal behavior at first since it is expected that increasing the number of ways would result in lower conflicts and thus higher hit time. Accordingly, we investigated each of the memGen functions to analyze how exactly they generate the memory addresses in order to determine whether any of the functions accounts for any case where it would happen that increasing the number of ways would increase the hit ratio.

**Why Experiment 2 resulted in constant hit ratios?**

As previously explained above, the main benefit of using n-set associative sets over direct-mapping is that the probability of having conflicts is less. In direct-mapping, there are many addresses which have the same index and accordingly are mapped to the same cache block. However, since there are varying tags, once an address with the same index and different tag is read, it overwrites the cache block and results in conflict miss. The sets method avoids this since for each index, there are n possible tags that could be read simultaneously without overwriting. The scenario which would result in a conflict miss here in n-set associative would be one where all lines in the set are full and we read another address that has the same index but a tag different from all the tags of all the current lines in the set. This would result in overwriting one of the lines and result in a miss. Moreover, if I then read another address with the same index but with the overwritten tag, I will get another miss. Now if we increase the number of ways, we would be able to accomodate more tags per set and thus probability of having conflict miss like this would even get lower. Accordingly, it remains true that increasing the number of ways should increase the hit ratio. However, we did not witness this in experiment 2 as none of the memGen functions reach a situation where a conflict miss is obtained.

For example, if we assume a 4 way and 16 byte line size cache and I read 4 addresses of the same index and all have different tags. Accordingly, the set will be full. Now if I read another address of the same index and a new different tag, one of the old tags will be overwritten. Now if I read another address after it that has the same index but the tag that had been overwritten, I am going to get a miss. This is the miss that could be avoided if we increase the number of ways to 8 for instance. However, this scenario never happens in any of the 6 memGen functions and thus we do not see any improvement in the hit ratio. Essentially the above situation resulting in misses, which could be avoided if we increase the number of ways, never occurs in any of the 6 memGen functions. Accordingly, this is why the hit ratio is constant throughout the entirety of experiment 2.

For further support of this argument, we provided the following manual test case to validate our ideas. We passed the following list of addresses to a 4 way, 16 byte line size cache and then to a 1 way, 16 byte line size cache:

Address 1: 0x00000

Address 2: 0x00001

Address 3: 0x04000

Address 4: 0x00001

Now, in this above example if we consider the 1 way and the 4 way cache, at index 0, it will hold tag 0 after the first address. After the second address, both the 1 way and the 4 way will record a hit as there is the same index and the same tag. Now comes the third address. In the 1 way, we will have to overwrite as this address has the same index but a different tag. However, in the 4 way, we will not overwrite as we have the capacity to read 4 different tags. Now comes the 4th address which will result in a hit with the 4 way cache; however, will result in a miss with the 1 way cache as the previous block has been overwritten in the 1 way cache.

This example demonstrates how increasing the number of ways can increase the number of hits and justifies our argument that none of the 6 memGen functions provide such a sequence to result in a such a situation where the hit ratio is improved as we increase the number of ways.

**MemGen Functions Analysis:**

**memGen1():**

Generates all addresses starting by 0x0 till (DRAM size)-1 then repeats from 0x0 again, which in our case is from **0x0** to **0x3FFFFFF inclusively**. That results in **1 miss** then **(Line Size - 1)** repeatedly until the number of addresses specified in **NO_OF_Iterations**, in our case 1,000,000 addresses, are processed. The small difference between each memory address allows the hit rate to be relatively high, especially compared to other memory generators.

**memGen2():**

Assuming **rand_()** generates a random number, **memGen2()** gives us addresses with a lower limit of **0x0000** and an upper limit of **0x600 inclusively**. It works by dividing the random number by (24*1024) to make sure the number falls within the range. For this memory generator, we can see that the hit rate remains constant at 66% despite the changes in the cache line size as well as the number of ways. This could be due to the fact that this generator doesn't take full advantage of the cache's spatial locality to the same degree as memory generators one and four, but has enough spatial locality for the hit rate to remain at 66%. Given that the addresses provided are completely random with the only condition provided is that they must remain within the first 24 KB of the dynamic memory, an experiment with a million instances will always result in the same hit rate.

**memGen3():**

Assuming that **rand_()** generates a random number, **memGen3()** generates a random address that is between 0x0 and (DRAM Size - 1) inclusively, in our case (DRAM Size = 64 MBytes) between **0x0** and **0x3FFFFFF inclusively**. Due to the large range that the memory address is limited to, there is no spatial locality, thus the hit rate becomes extremely low.

**memGen4():**

This function generates all addresses starting by **0x0** till **0xFFF inclusively**, which is an upper limit of (4*1024 - 1) then repeats from 0x0 again until the number of addresses specified in **NO_OF_Iterations**, in our case 1,000,000 addresses, are processed.

**memGen5():**

This function generates all addresses starting by **0x0** till **0xFFFF inclusively**, then repeats from 0x0 again until the number of addresses specified in **NO_OF_Iterations**, in our case 1,000,000 addresses, are processed.

**memGen6():**

This generator generates memory addresses that have a 32 bit interval. In other words, the first address will have an immediate value of 0, the next will be 32, the next 64 and so on and so forth. Moreover, this memory generator ensures that the addresses generated are always located within the initial 64KBs of the dynamic memory. In other words, it generates an address between **0x0 and 0x3FFFF inclusively**. With these specific properties, there should be no hits occuring in the case that the cache lines are 32 bytes or less. And as we can see, from both experiments, no hits occurred in the cases where cache lines are 32 or 16 bytes in size. This is due to the fact that with each new address, a cold open miss will occur. However, once the cache line sizes become 64 bytes, we can see a 50% hit rate, as the cache will follow a simple pattern of miss, hit, miss, hit… We can also see from our first graph, that as the cache line size is set to 128 bytes, we start to see a hit rate of 75% as we see a cold start miss every 3 hits.

**Conclusion:**

Larger line size is beneficial when the spatial locality is high since more data that are in close spatial locality, specifically in sequential order are read together. In that case, hit rate is increased significantly by increasing the line size.

Furthermore, more N-Ways also increases the hit rate but requires additional and more complex hardware in order to compare the TAGs of the lines stored in each set. Increasing the number of lines per set is beneficial but not necessarily because of spatial locality as much as the case of increasing the line size instead of the N-way.