Cache Memory

Assembly and Computer Organization

Project 2

By:

Ahmed Ibrahim

Andrew Fahmy

Dr. Shalan

Program Description:

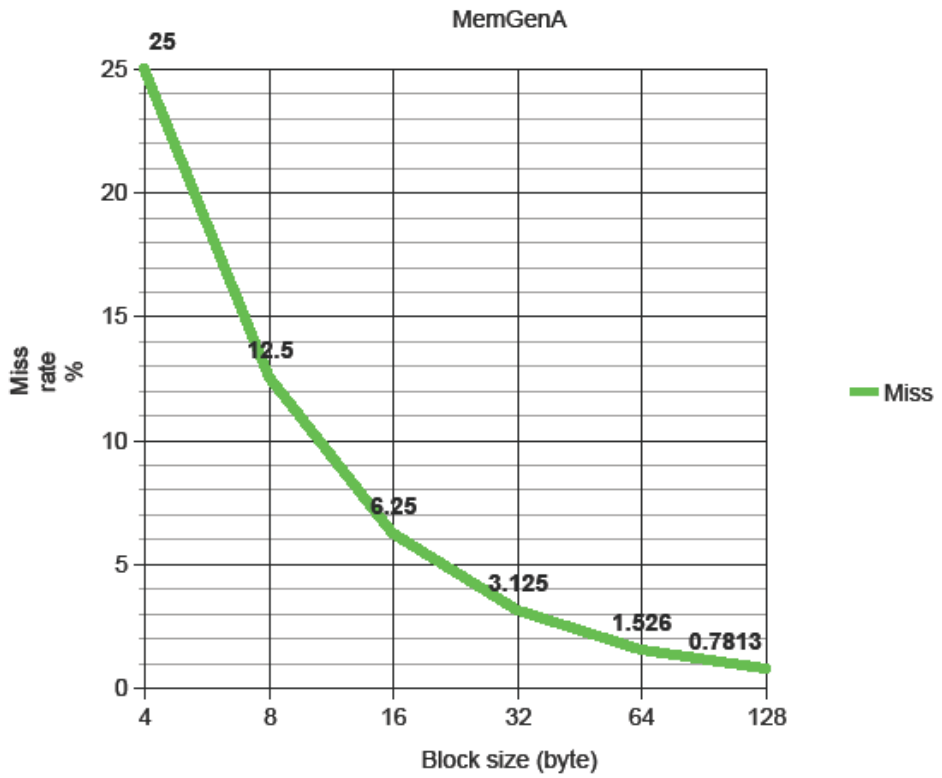
This project is a simulation and an implementation for the Cache Memory using direct mapped cache that has a size of 64KB (64 *1024B). The line/ block size was interchanged, from 4, 8, 16, 32, 64, and 128 for each test case. Before simulating the memory generators given in the handout, we implemented different test cases, from past assignments and lecture notes, to validate our program. The structure of our cache is recognized as a simple struct, in which inside the cache there's a valid bit initialized as zero, and an empty index. The direct cache implementation checks the tag and the valid bit according to the specific index, where the valid bit shows whether the data in the cache is valid or not, and the tag represents the key to the data inside the cache. If the tag is not the same or the valid bit is 0, it's considered as a miss and respectively the index or valid bit are modified.

The program has 6 memory generation functions along with 6-line sizes so that accounts for a total of 36 runs (simulations) iterated million times. We start by memGenA and then change the line spaces from 4 to 128 accordingly and we then move onto the next memGen function (memGenB, memGenC, etc.) The cacheSimDM function is the function that checks the miss/ hit ratio and implements the program described above. If the return value is a hit, then the hit counter is incremented. After the set number of iterations is over, the program prints out the hit and misses ratios. Such ratios are graphed in a line graph separately for analysis every memory generator.

Limitation:

The results presented by this program are dependent on some factors, such as :

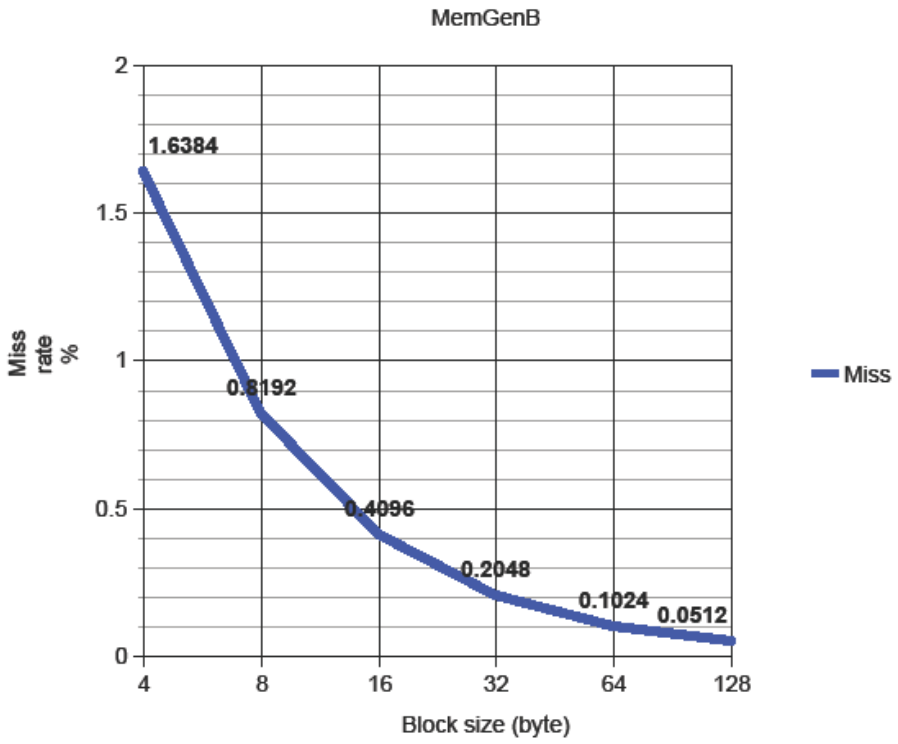
- 1000000 iterations in the loop.
- The cache size is constant as well as the DRAM size.
- Results are dependent on specific sizes and have to modified after every iteration



Analysis:

```
unsigned int memGenA()
{
    static unsigned int addr=0;
    return (addr++)%(DRAM_SIZE/4);
}
```

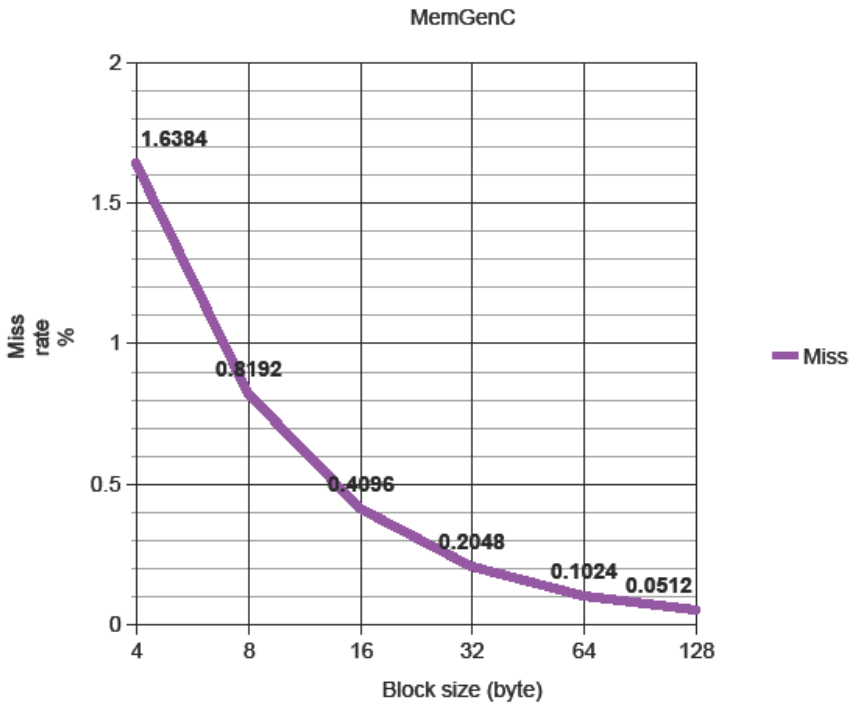
Firstly the MemGenA function is implemented by having a static array, in which the address is incremented by one every iteration, which is capped at (Dram size /4). Looking closely to the functionality of such function and its line graph, we can observe that as we increase the size, the hit ratio is increased and miss ratio is decreased. Such conclusion is obvious due to spatial locality, in which every miss would result in filling the block size, thus the hit ratio would be, (linesize-1/linesize). This shows why as we increase the block size, the hit ratio increases as well.



Analysis:

```
unsigned int memGenB()
{
    static unsigned int addr=0;
    return  rand_()%(64*1024);
}
```

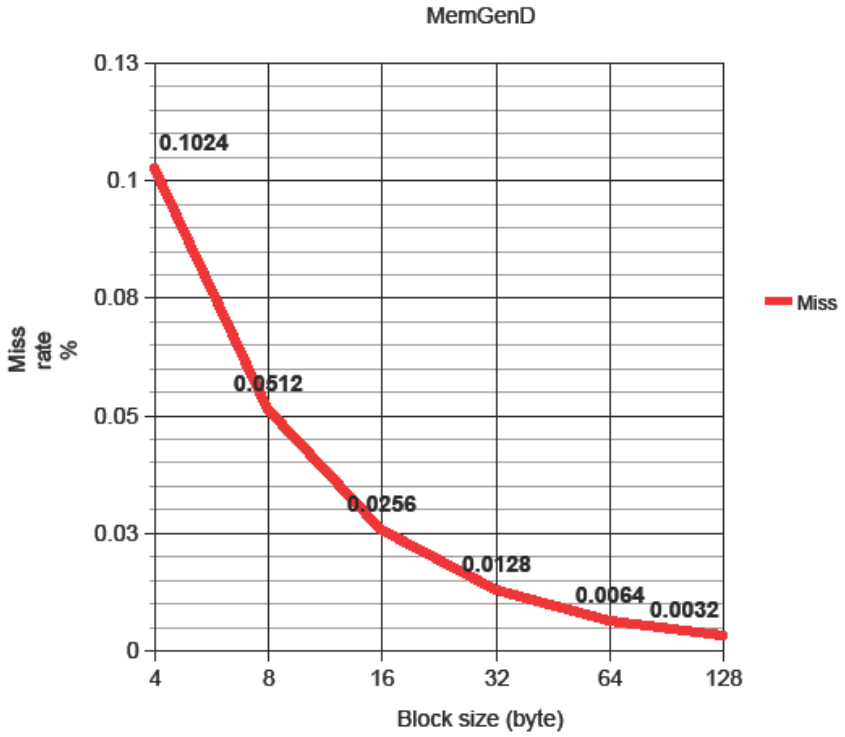
As shown in this function, a rand function is implemented, which is capped at the (64*1024). We can observe that as the iteration number is greater than the range of the addresses, we would expect at one point of time, assuming in the worst case that the first (64*1024) would be misses, the cache would have all the possible addresses. Thus, in the next cycle it's guaranteed to have a hit. Moreover, as this function is random, it's hard to predict its ratio behavior, however, as we are guaranteed that the number doesn't exceed the cache size, we can determine the total hit/miss ratio.



Analysis:

```
unsigned int memGenC()
{
    static unsigned int a1=0, a0=0;
    a0++;
    if(a0 == 512) { a1++; a0=0; }
    if(a1 == 128) a1=0;
    return(a1 + a0*128);
}
```

The function memGenC iterates 512 to increment in a1 and resets at 128. This function is similar to memGenA, in which address is incremented by one; however, it's incremented every 512 cycle. So in short this function acts a delayed version of memGenA. Determining the hit ratio would be linesize-1/linesize every 512 iteration.

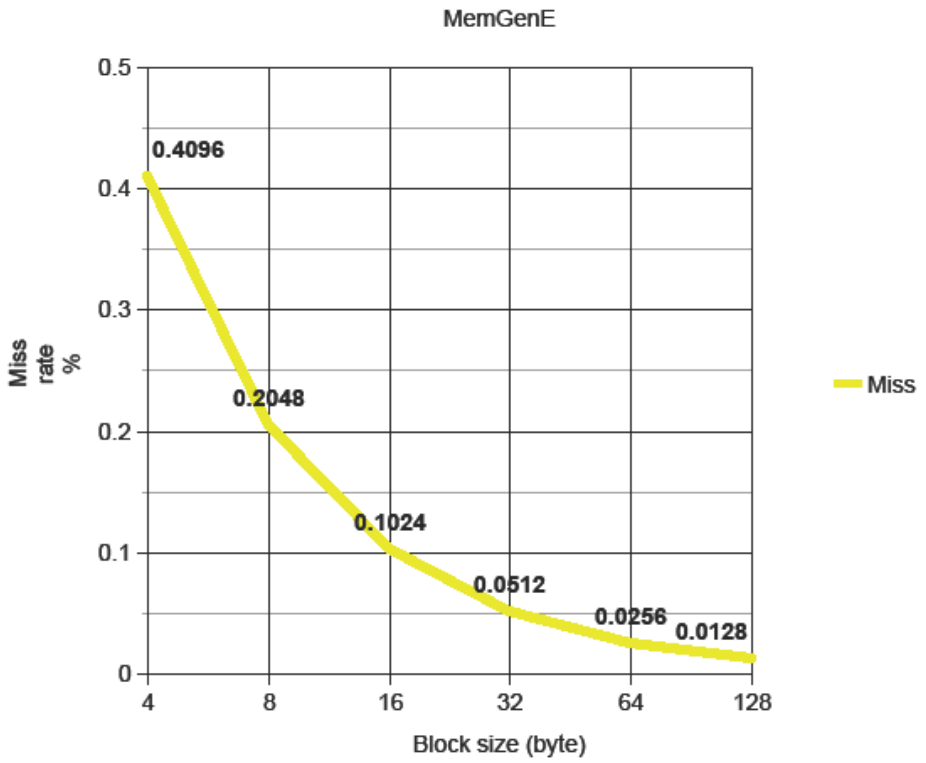


Analysis:

```
unsigned int memGenD()
{
    static unsigned int addr=0;
    return (addr++)%(1024*4);
}
```

MemGenD function is a similar to memGenA, in which the address is incremented by one and due to spatial locality, which a miss would result to n-1 hits, where n is the block size. However, this function is capped to (1024*4), which means that in the first cycle the hit ratio would be n-1/n in the first 4096, due to the spatial locality, then cycle restarts with all hits.

An equation to represent hit ratio of this function is 1 miss to n-1 hits, where n is the size of block, so (1024*4) would equal to (((1024*4)* n-1)- total iterations)/ total iterations).



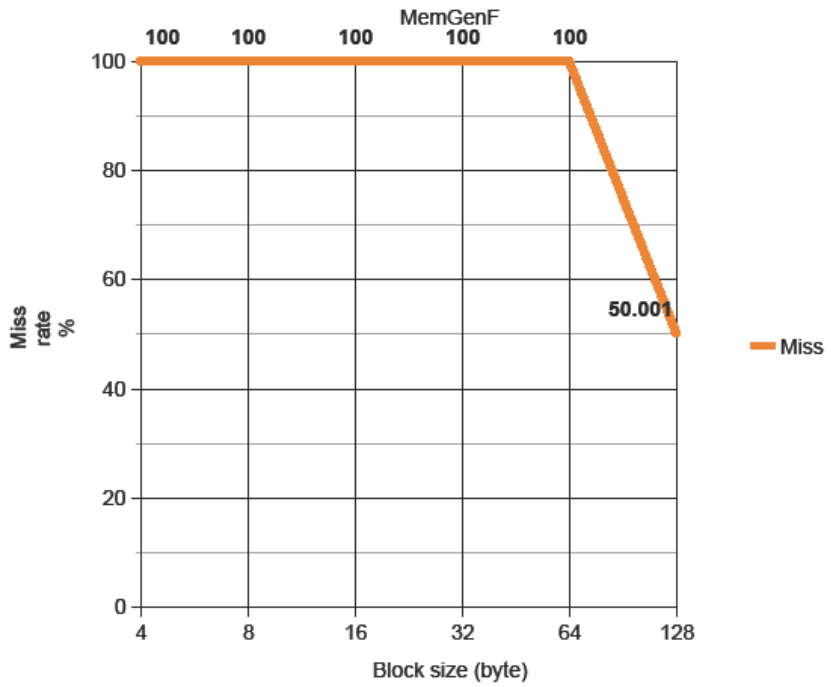
Analysis:

```
unsigned int memGenE()
{
    static unsigned int addr=0;
    return (addr++)%(1024*16);
}
```

MemGenE function is a similar to memGenD, in which the address is incremented by one and due to spatial locality, which a miss would result to n-1 hits, where n is the block size. However, this function here is capped to (1024*16) rather than (1024*16), which means that in the first cycle the hit ratio would be n-1/n in the first (1024*16), due to the spatial locality, then cycle restarts with all hits.

An equation to represent hit ratio of this function is 1 miss to n-1 hits, where n is the size of block, so (1024*16) would equal to

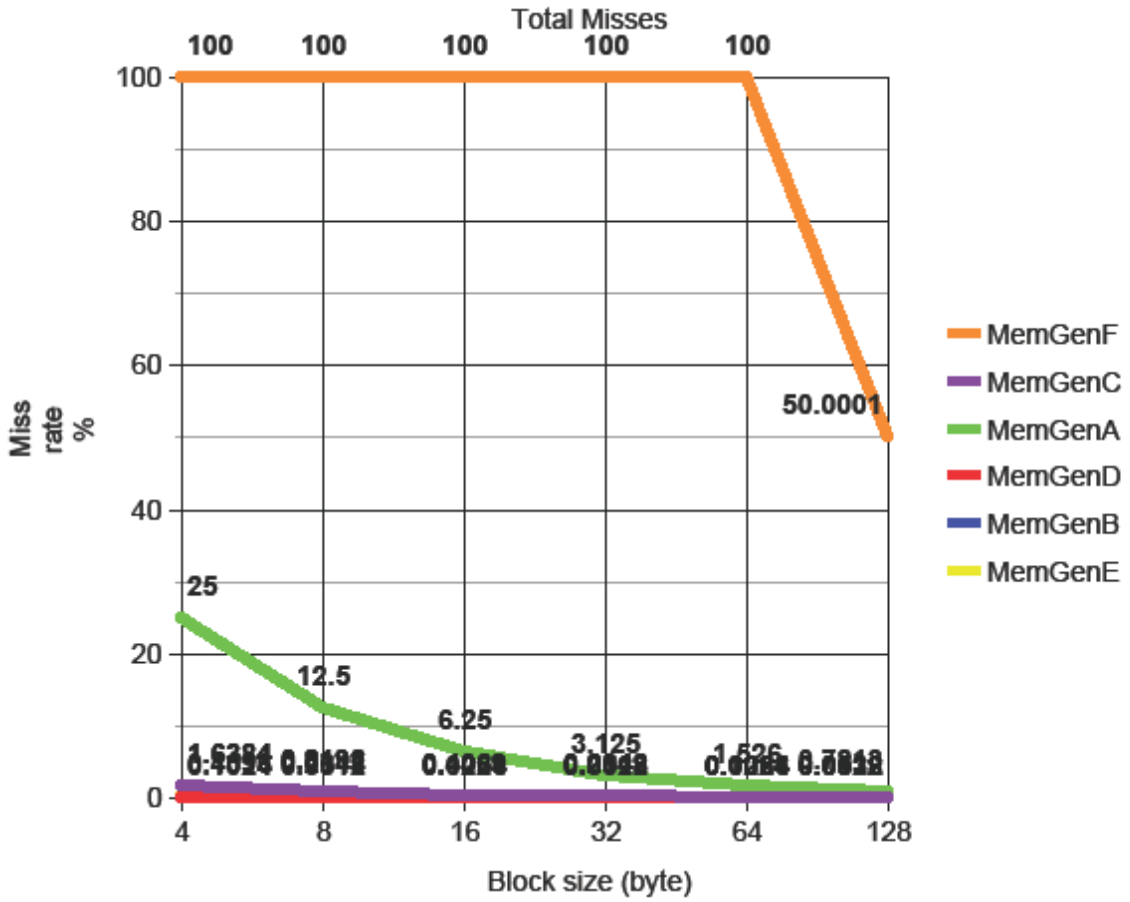
(((1024*16)* n-1)- total iterations)/ total iterations



Analysis:

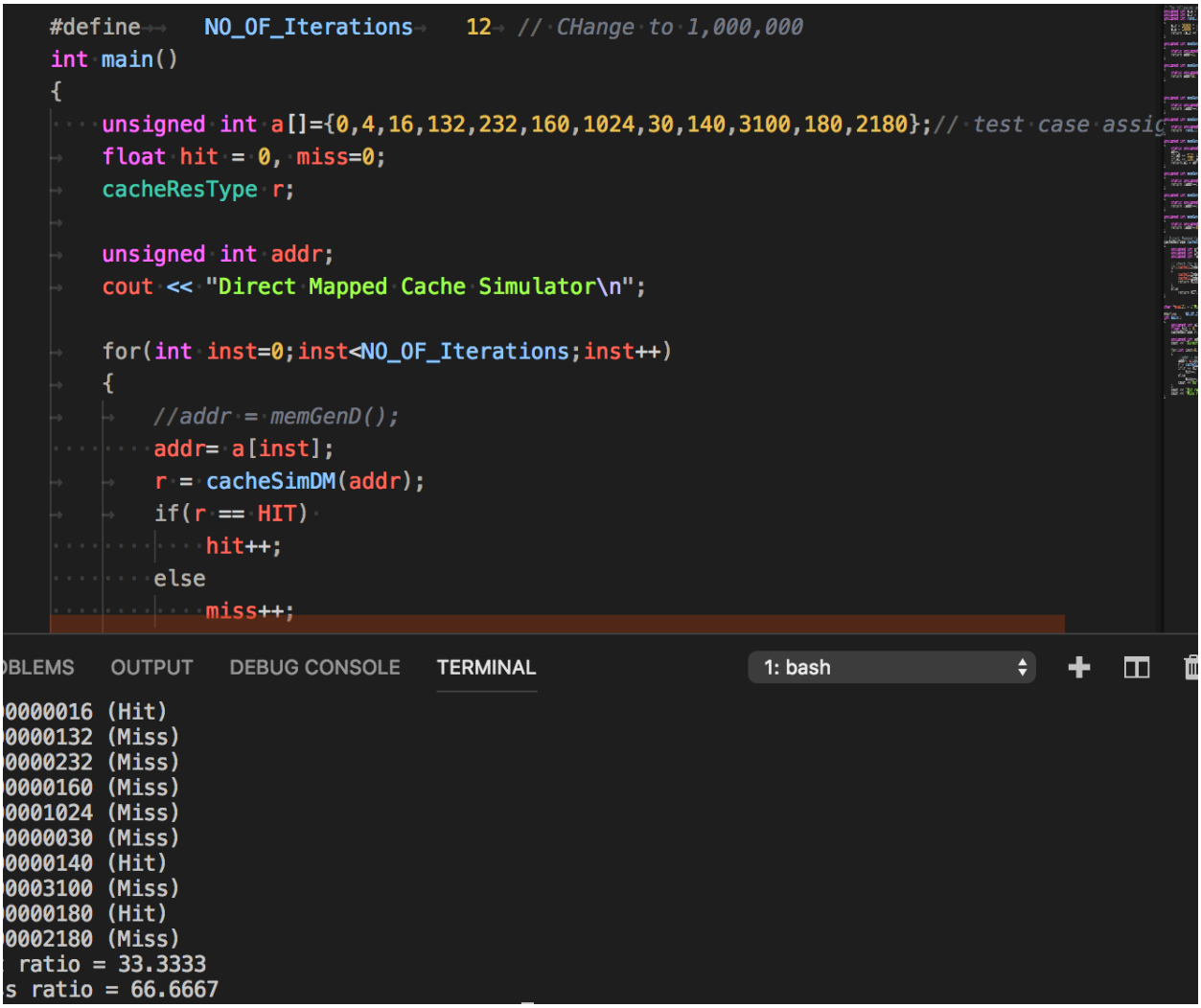
```
unsigned int memGenF()
{
    static unsigned int addr=0;
    return (addr+=64)%(128*1024);
}
```

In memGenF, we can observe a non-interacting behavior shown in the graph above, however, as we reach a specific size, we can observe interaction regarding the hit/miss ratio. This is due to incrementing the addresses by 64, in which spatial locality is not utilized due to the ratio of the address to the block size. To clarify, to be able to utilize spatial locality, we would need a block size greater than 64, for example, if the block size is 128, then due to locality one block would have 2 addresses; thus, the hit ration would be 1 miss/ 1 hit 50:50 ratio. Another example would be if we have the block size as 256, which can be divided into 4 blocks of 64, thus resulting into 1 miss to 3 hits. Thus resulting in a hit ratio of 4/3.



Conclusion:

In conclusion, we can observe that in all the previous functions, spatial locality played a crucial role in increasing the hit ratio of direct mapping cache. It's also evident that as we increase the size of the block size the hit ratio increases while the miss ratio decreases, which is also due to utilizing spatial locality more and more.



Validating from the assignment