THE AMERICAN UNIVERSITY IN CAIRO

الجامعة الأمريكية بالقاهرة

CSCE 2303

Computer org and Assembly

Summer 2024

Dr Mohamed Shalan

Cache Simulator Project Report

Ahmed Abdeen

Ismaiel Sabet 900221277

Mostafa Gaafar 900214463

Youssef Badawy

Contents:

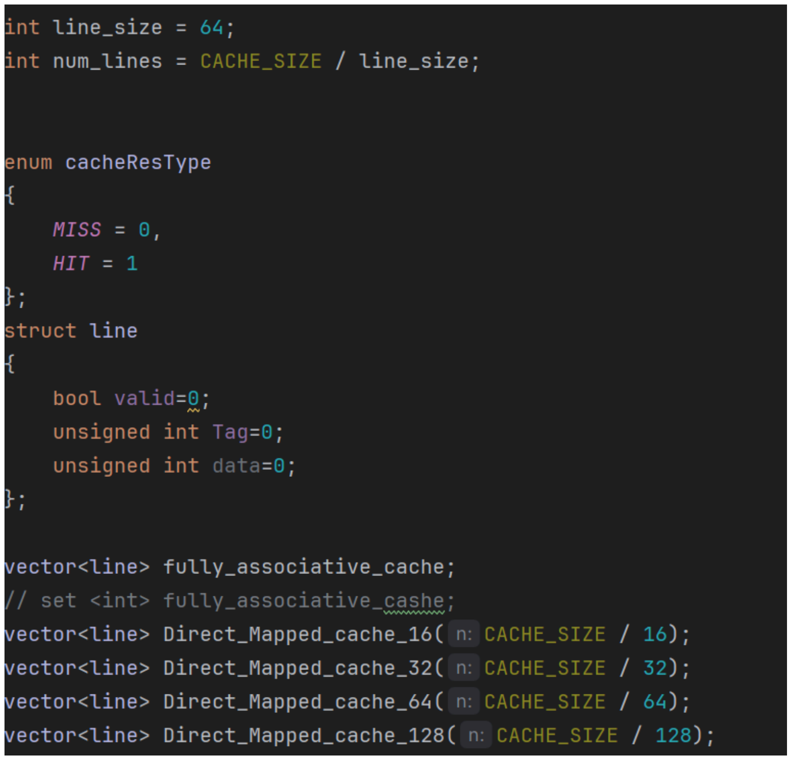
Abstract:

As mentioned in the project description file, the purpose of this project is to implement a cache simulator that directly mimics the behavior of both the Direct Mapped and the Fully Associative caches respectively for different line sizes and using different functions as the replacement protocol.

We first complete the provided skelton code, providing snippets of the code with explanation. Following that, we conduct the experiments using our code and generate the hit rate for the different cache sizes for both Direct Mapped and Fully Associative caches, utilizing each of the six random number generator functions. Then, we conduct similar experiments but manually. What follows is a comparison of the two approaches and an analysis of the data, followed by the conclusion of our results and a wrap-up of all the explored ideas in the project.

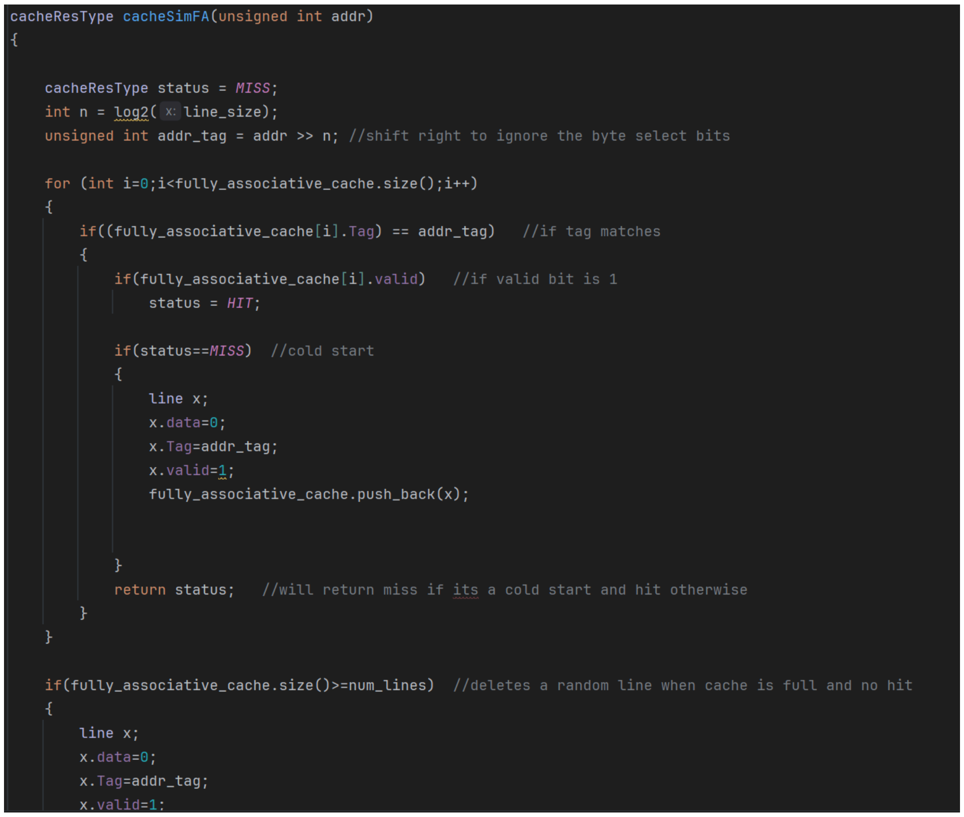
Code Snippets:

As most of the code provided in the skeleton is unchanged, only images of the additions we made will be provided below and discussed.

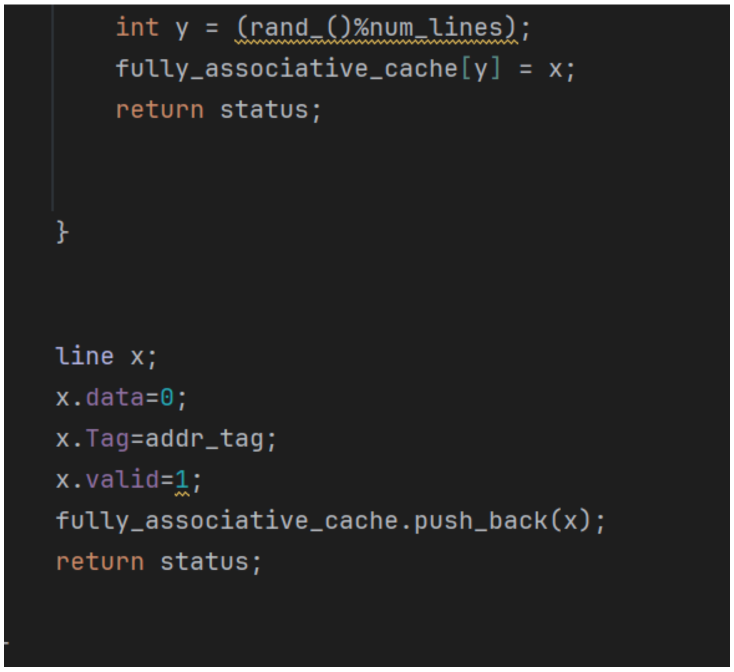


```
int line_size = 64;
int num_lines = CACHE_SIZE / line_size;


enum cacheResType
{
    MISS = 0,
    HIT = 1
};
struct line
{
    bool valid=0;
    unsigned int Tag=0;
    unsigned int data=0;
};

vector<line> fully_associative_cache;
// set <int> fully_associative_cashe;
vector<line> Direct_Mapped_cache_16( n: CACHE_SIZE / 16);
vector<line> Direct_Mapped_cache_32( n: CACHE_SIZE / 32);
vector<line> Direct_Mapped_cache_64( n: CACHE_SIZE / 64);
vector<line> Direct_Mapped_cache_128( n: CACHE_SIZE / 128);
```

The above image reveals important data declarations that were made which ultimately assisted the development of the simulator. We defined the line size (and consequently change it whenever we desire to test the other three cases). The num of lines (in other words the maximum number of lines) was computed to be simply the CACHE_SIZE/ line_size where the CACHE_SIZE is given and the line_size is one of 16, 32, 64, and 128. We then declared a struct called line. In addition, all of the five provided vectors are simply the different Direct Mapped caches for the different line sizes (notice we are dividing by the line size which is one of the four previously mentioned values) and one for the Fully Associative cache. The vectors are all of type line, which is the struct we declared.

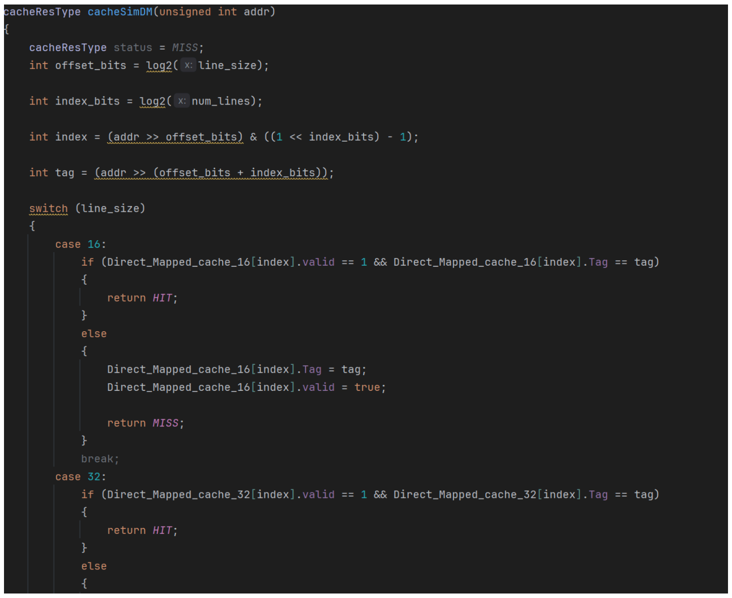


```
cacheResType cacheSimFA(unsigned int addr)
{

    cacheResType status = MISS;
    int n = log2( x: line_size);
    unsigned int addr_tag = addr >> n; //shift right to ignore the byte select bits

    for (int i=0;i<fully_associative_cache.size();i++)
    {
        if((fully_associative_cache[i].Tag) == addr_tag)   //if tag matches
        {
            if(fully_associative_cache[i].valid)   //if valid bit is 1
                status = HIT;

            if(status==MISS)  //cold start
            {
                line x;
                x.data=0;
                x.Tag=addr_tag;
                x.valid=1;
                fully_associative_cache.push_back(x);


            }
            return status;   //will return miss if its a cold start and hit otherwise
        }
    }

    if(fully_associative_cache.size()>=num_lines)  //deletes a random line when cache is full and no hit
    {
        line x;
        x.data=0;
        x.Tag=addr_tag;
        x.valid=1;
```

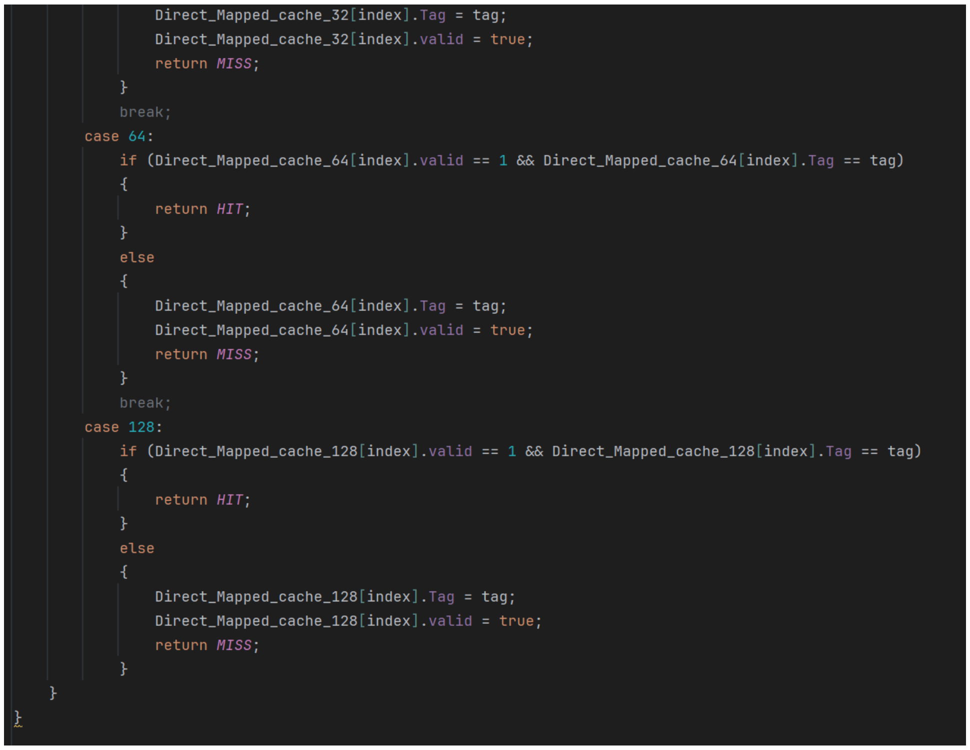


```
        int y = (rand_()%num_lines);
        fully_associative_cache[y] = x;
        return status;


    }



    line x;
    x.data=0;
    x.Tag=addr_tag;
    x.valid=1;
    fully_associative_cache.push_back(x);
    return status;
```

The above images highlight the simulation function that handles the Fully Associative cache. We begin by initializing a cacheResType to be “MISS” and then get the n by getting the log of the number of bytes in a single tag (so 16, 32, 64 etc). This is important as it will influence how we get the tag. We loop across all the elements in the cache, checking if the tags match. If they do, we then check if it’s valid. If both of these conditions hold, then we have a hit. Otherwise, we explore the possibility of a cold start miss. Further, outside the loop we explore the possibility that the size of the vector has exceeded the max number of lines allowed. In this case, the cache is full, and we select a random index which as such corresponds to a random location in the memory and delete whatever is in said address.



```
cacheResType cacheSimDM(unsigned int addr)
{
    cacheResType status = MISS;
    int offset_bits = log2(line_size);

    int index_bits = log2(num_lines);

    int index = (addr >> offset_bits) & ((1 << index_bits) - 1);

    int tag = (addr >> (offset_bits + index_bits));

    switch (line_size)
    {
        case 16:
            if (Direct_Mapped_cache_16[index].valid == 1 && Direct_Mapped_cache_16[index].Tag == tag)
            {
                return HIT;
            }
            else
            {
                Direct_Mapped_cache_16[index].Tag = tag;
                Direct_Mapped_cache_16[index].valid = true;

                return MISS;
            }
            break;
        case 32:
            if (Direct_Mapped_cache_32[index].valid == 1 && Direct_Mapped_cache_32[index].Tag == tag)
            {
                return HIT;
            }
            else
            {
```



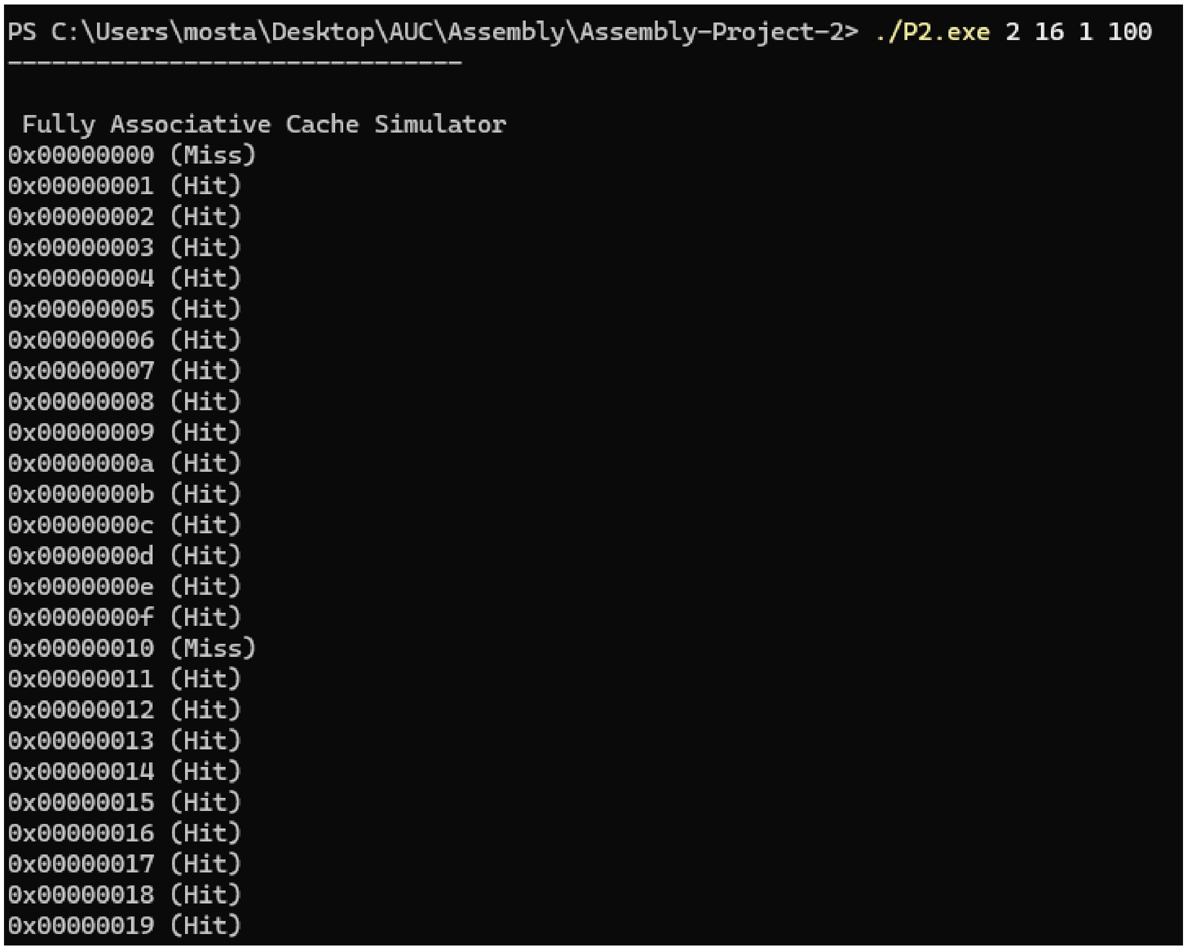
```
                Direct_Mapped_cache_32[index].Tag = tag;
                Direct_Mapped_cache_32[index].valid = true;
                return MISS;
            }
            break;
        case 64:
            if (Direct_Mapped_cache_64[index].valid == 1 && Direct_Mapped_cache_64[index].Tag == tag)
            {
                return HIT;
            }
            else
            {
                Direct_Mapped_cache_64[index].Tag = tag;
                Direct_Mapped_cache_64[index].valid = true;
                return MISS;
            }
            break;
        case 128:
            if (Direct_Mapped_cache_128[index].valid == 1 && Direct_Mapped_cache_128[index].Tag == tag)
            {
                return HIT;
            }
            else
            {
                Direct_Mapped_cache_128[index].Tag = tag;
                Direct_Mapped_cache_128[index].valid = true;
                return MISS;
            }
    }
}
```

The above three images describe the simulation function that handles the Direct Mapped cache. The first thing done is extracting all the necessary bits i.e. index and tag. Then based on the line_size , which could be 16, 32, 64, or 128, we pick the appropriate Direct Mapped vector as declared at the start of the code. Ultimately, each of these cases do the exact same thing, but they differ only in which vector is being accessed. So for demonstration sake, let’s take case 16. Firstly, we check for validity and if the tags match, and if they do, then we have got ourselves a hit. Otherwise, we replace the available tag with the tag we have, make the valid bit true, and return a miss. As previously mentioned, case 32, 64, and 128 will all behave in the exact same way.

Testing :

FA cache testing :

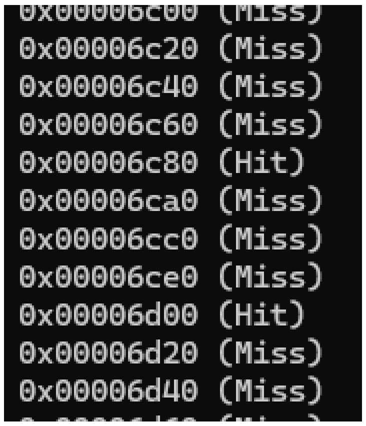
To test the fully associative cache we had to test 3 things exactly, Firstly we had to test that cold starts are counted for and are intercepted correctly, next we needed to check that our code can successfully interpret a memory hit. And lastly we had to check that our code correctly replaces a cache line in the event of a capacity miss. To test for the first case we simply ran memGen1 on a small number of iterations which ended up proving that the function works for both case 1 and case 2, namely a cold start and a hit.



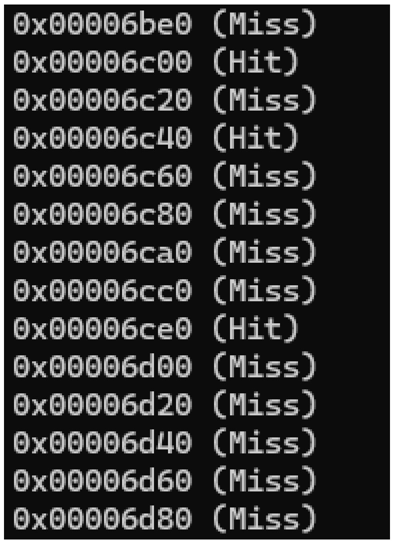
```
PS C:\Users\mosta\Desktop\AUC\Assembly\Assembly-Project-2> ./P2.exe 2 16 1 100
-------------------------------

 Fully Associative Cache Simulator
0x00000000 (Miss)
0x00000001 (Hit)
0x00000002 (Hit)
0x00000003 (Hit)
0x00000004 (Hit)
0x00000005 (Hit)
0x00000006 (Hit)
0x00000007 (Hit)
0x00000008 (Hit)
0x00000009 (Hit)
0x0000000a (Hit)
0x0000000b (Hit)
0x0000000c (Hit)
0x0000000d (Hit)
0x0000000e (Hit)
0x0000000f (Hit)
0x00000010 (Miss)
0x00000011 (Hit)
0x00000012 (Hit)
0x00000013 (Hit)
0x00000014 (Hit)
0x00000015 (Hit)
0x00000016 (Hit)
0x00000017 (Hit)
0x00000018 (Hit)
0x00000019 (Hit)
```

As seen in this screenshot after running the FA cache on a 16 byte line, the cold start misses works as well as the hits since it misses once and hits 15 times and so on and so forth, it was also checked for larger lines and it works in the same way, as in for 32 bytes it mises once and hits 31 times.

For the capacity misses it was a bit trickier as we had to fill the cache first and then delete from it, several memGen functions were used for this, namely memGen6 and memGen3.
We used memGen6 with a 32 byte line and a replacement policy of random replacement, we see how after missing enough times to fill the cache, requesting an address that was previously stored in the cache returns a miss because it was replaced



```
0x00006c00 (Miss)
0x00006c20 (Miss)
0x00006c40 (Miss)
0x00006c60 (Miss)
0x00006c80 (Hit)
0x00006ca0 (Miss)
0x00006cc0 (Miss)
0x00006ce0 (Miss)
0x00006d00 (Hit)
0x00006d20 (Miss)
0x00006d40 (Miss)
```



```
0x00006be0 (Miss)
0x00006c00 (Hit)
0x00006c20 (Miss)
0x00006c40 (Hit)
0x00006c60 (Miss)
0x00006c80 (Miss)
0x00006ca0 (Miss)
0x00006cc0 (Miss)
0x00006ce0 (Hit)
0x00006d00 (Miss)
0x00006d20 (Miss)
0x00006d40 (Miss)
0x00006d60 (Miss)
0x00006d80 (Miss)
```

focus on 6c80 how it was a hit which means it was in the cache but shortly after it was replaced and gave us a miss.

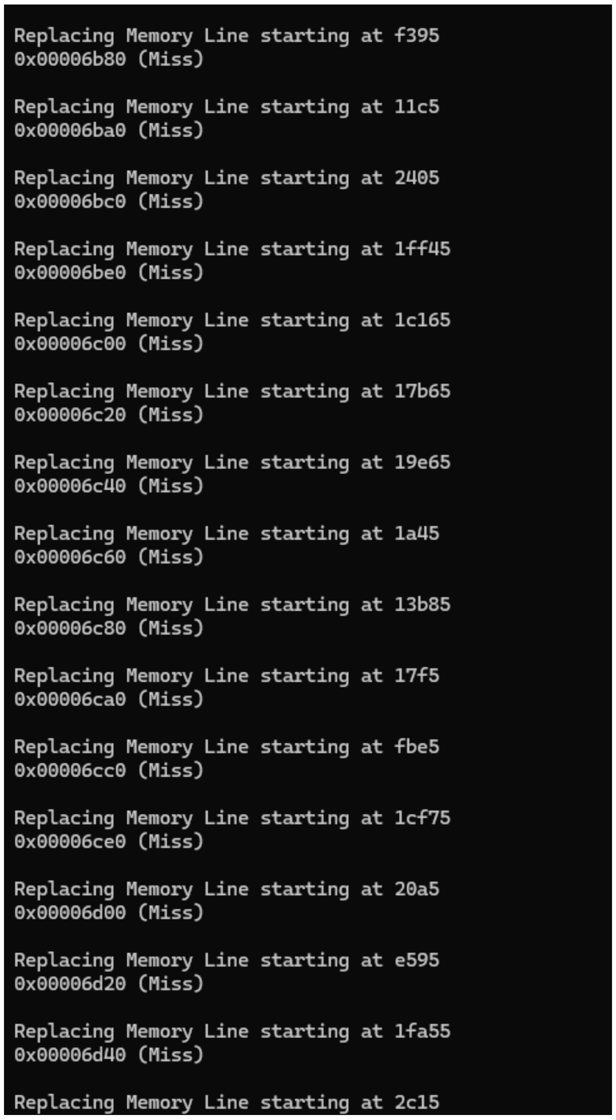
The images below prints out more details to what was happening, since the cache was full every new memory access was a capacity miss which deleted one of the lines what already existed. We can also see the moment ourcache becomes full, which was at address call 0x10000 which corresponds to the maximum numbers of lines since it was 64Kbytes / 32 bytes = 2048. And since we are accessing memory in intervals of 32 bytes this means it should be full at memory address 65536 decimal which is 0x1000 in hex. This concludes the code for the fully associative cache because after making these tests we were sure that it hits correctly and does both cold start and capacity misses correctly using the rand_() function provided.

Command line prompts used in testing

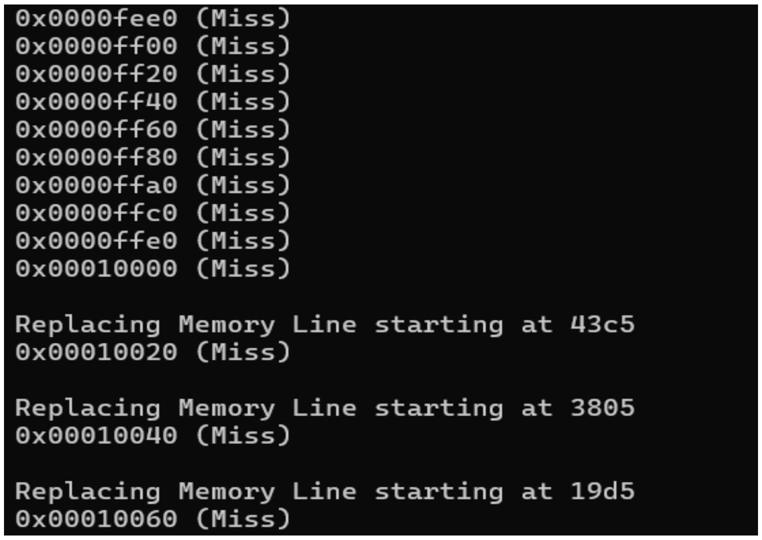
```
./P2.exe 2 32 6 1000
```

./P2.exe 2 16 1 1000
./P2.exe 2 321 1000

For the first prompt, 2 is representing that we want to run FA cache, 32 is the line size, 6 is the memgen function and 1000 is the iteration, similarly the rest of the prompts follow this division (Note that the code for testing is commented out in the source code but it can be found there).

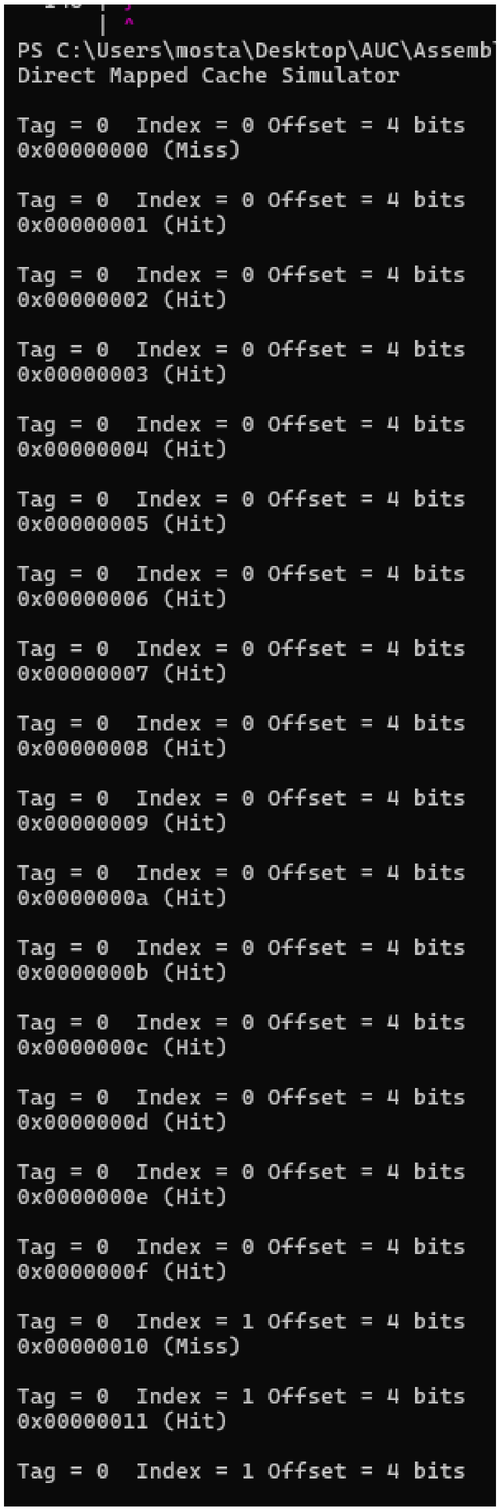


```
Replacing Memory Line starting at f395
0x00006b80 (Miss)

Replacing Memory Line starting at 11c5
0x00006ba0 (Miss)

Replacing Memory Line starting at 2405
0x00006bc0 (Miss)

Replacing Memory Line starting at 1ff45
0x00006be0 (Miss)

Replacing Memory Line starting at 1c165
0x00006c00 (Miss)

Replacing Memory Line starting at 17b65
0x00006c20 (Miss)

Replacing Memory Line starting at 19e65
0x00006c40 (Miss)

Replacing Memory Line starting at 1a45
0x00006c60 (Miss)

Replacing Memory Line starting at 13b85
0x00006c80 (Miss)

Replacing Memory Line starting at 17f5
0x00006ca0 (Miss)

Replacing Memory Line starting at fbe5
0x00006cc0 (Miss)

Replacing Memory Line starting at 1cf75
0x00006ce0 (Miss)

Replacing Memory Line starting at 20a5
0x00006d00 (Miss)

Replacing Memory Line starting at e595
0x00006d20 (Miss)

Replacing Memory Line starting at 1fa55
0x00006d40 (Miss)

Replacing Memory Line starting at 2c15
```

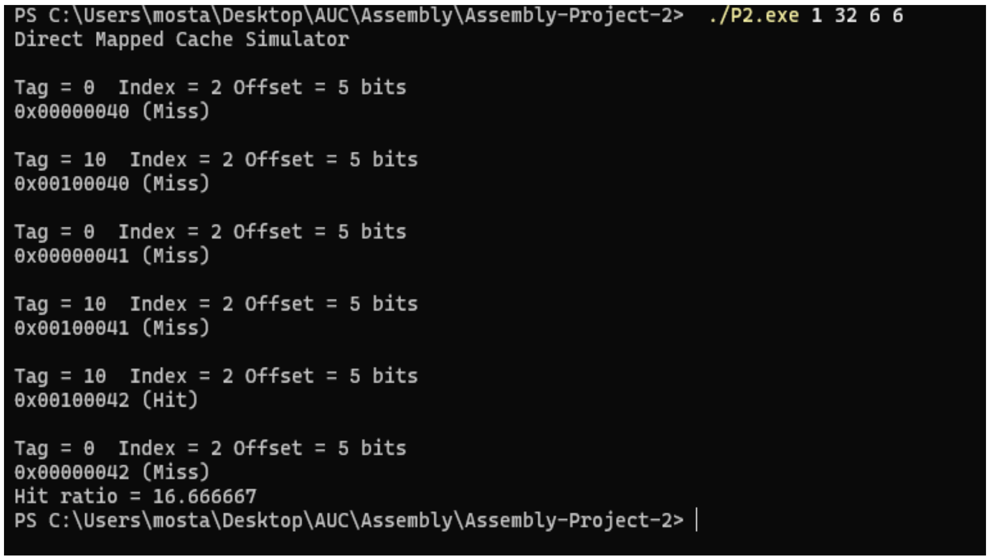


```
0x0000fee0 (Miss)
0x0000ff00 (Miss)
0x0000ff20 (Miss)
0x0000ff40 (Miss)
0x0000ff60 (Miss)
0x0000ff80 (Miss)
0x0000ffa0 (Miss)
0x0000ffc0 (Miss)
0x0000ffe0 (Miss)
0x00010000 (Miss)

Replacing Memory Line starting at 43c5
0x00010020 (Miss)

Replacing Memory Line starting at 3805
0x00010040 (Miss)

Replacing Memory Line starting at 19d5
0x00010060 (Miss)
```

For the direct mapping cache, we also need to test for 3 things, namely the cold start misses, the hits, and the conflict misses. For the first 2 we can use the same test case as the fully associative with the addition of printing the tag, index, and offset bits to check if our algorithm computes it correctly or not. And as we can see in the output below, it it evident that the cold start misses as well as the normal hits work. Now what remains is testing for the conflict misses. For conflict misses we created a custom run that goes through specific addresses and as seen, it successfully replaces the conflicted line with the correct data as per the screenshot (Note that the code for this run is commented out but its included in the source file).



```
      | ^
PS C:\Users\mosta\Desktop\AUC\Assemb
Direct Mapped Cache Simulator

Tag = 0  Index = 0 Offset = 4 bits
0x00000000 (Miss)

Tag = 0  Index = 0 Offset = 4 bits
0x00000001 (Hit)

Tag = 0  Index = 0 Offset = 4 bits
0x00000002 (Hit)

Tag = 0  Index = 0 Offset = 4 bits
0x00000003 (Hit)

Tag = 0  Index = 0 Offset = 4 bits
0x00000004 (Hit)

Tag = 0  Index = 0 Offset = 4 bits
0x00000005 (Hit)

Tag = 0  Index = 0 Offset = 4 bits
0x00000006 (Hit)

Tag = 0  Index = 0 Offset = 4 bits
0x00000007 (Hit)

Tag = 0  Index = 0 Offset = 4 bits
0x00000008 (Hit)

Tag = 0  Index = 0 Offset = 4 bits
0x00000009 (Hit)

Tag = 0  Index = 0 Offset = 4 bits
0x0000000a (Hit)

Tag = 0  Index = 0 Offset = 4 bits
0x0000000b (Hit)

Tag = 0  Index = 0 Offset = 4 bits
0x0000000c (Hit)

Tag = 0  Index = 0 Offset = 4 bits
0x0000000d (Hit)

Tag = 0  Index = 0 Offset = 4 bits
0x0000000e (Hit)

Tag = 0  Index = 0 Offset = 4 bits
0x0000000f (Hit)

Tag = 0  Index = 1 Offset = 4 bits
0x00000010 (Miss)

Tag = 0  Index = 1 Offset = 4 bits
0x00000011 (Hit)

Tag = 0  Index = 1 Offset = 4 bits
```

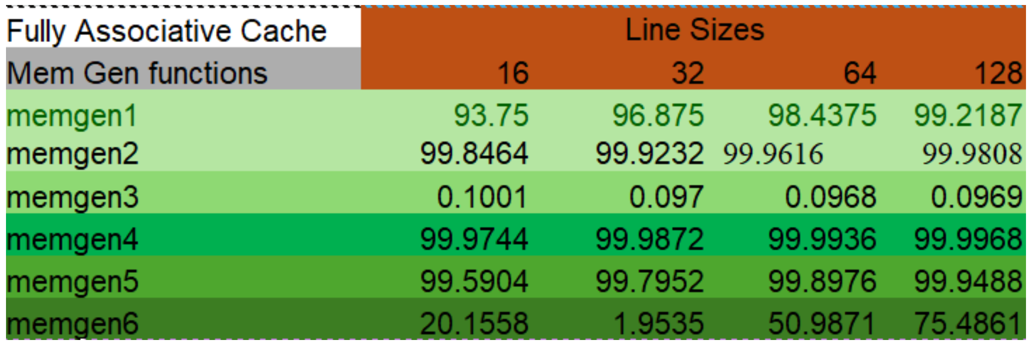

```
PS C:\Users\mosta\Desktop\AUC\Assembly\Assembly-Project-2>  ./P2.exe 1 32 6 6
Direct Mapped Cache Simulator

Tag = 0  Index = 2 Offset = 5 bits
0x00000040 (Miss)

Tag = 10  Index = 2 Offset = 5 bits
0x00100040 (Miss)

Tag = 0  Index = 2 Offset = 5 bits
0x00000041 (Miss)

Tag = 10  Index = 2 Offset = 5 bits
0x00100041 (Miss)

Tag = 10  Index = 2 Offset = 5 bits
0x00100042 (Hit)

Tag = 0  Index = 2 Offset = 5 bits
0x00000042 (Miss)
Hit ratio = 16.666667
PS C:\Users\mosta\Desktop\AUC\Assembly\Assembly-Project-2> |
```

Human Output:

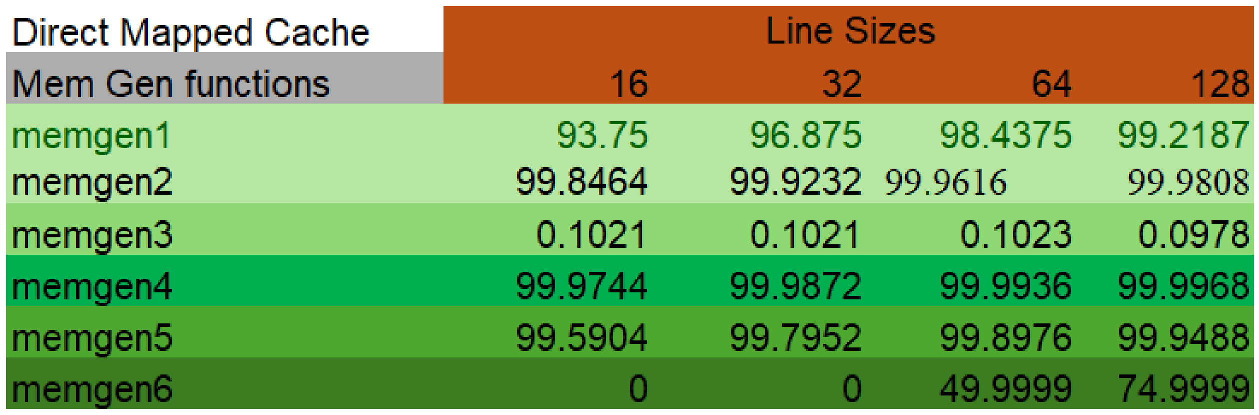
For the Fully associative cache simulator, the hit rate for memgen 1 and memgen2 is expected since the memory access is sequential. For memgen1 memory access the trend is basically to miss once (whether it's a coldstart miss or a capacity miss) and hit for whatever the length of the line is so the hit rate could be expected to be in the range of 93.75-99.3% for a large number of iterations. For memgen2 the addresses are fluctuating between 0-24*1024 which is less than the memory size so even though we will have a lot of initial misses, if we kept accessing the memory eventually we won't have any misses since we would have all of the memory we need to access in the cache. The more we iterate the larger of a hit rate we get. For memGen3 it was expected to have little to no hit rate at all given that the memory accesses are totally random which disregards the concept of temper locality and spatial locality which is why caches were implemented in the first place. Regarding memGen 4 and 5 they are somewhat similar since the memory reads are sequential up to a point, this point is reached quicker in memGen4 which explains how it always had better hit rate when compared to memGen5. Finally with memGen6 given how the memory accesses jumps frequently by 32 we initially get a low hit for small block sizes however, as we increase the size of the blocks we start seeing more hits as more memory data are stored in the line that will be used later therefore they won't be passed and later replaced

Program Output:

Below are the outputs of the hit ratio for both the Fully Associative and Direct Mapped caches respectively in a tabular format:



Fully Associative Cache	Line Sizes			
Mem Gen functions	16	32	64	128
memgen1	93.75	96.875	98.4375	99.2187
memgen2	99.8464	99.9232	99.9616	99.9808
memgen3	0.1001	0.097	0.0968	0.0969
memgen4	99.9744	99.9872	99.9936	99.9968
memgen5	99.5904	99.7952	99.8976	99.9488
memgen6	20.1558	1.9535	50.9871	75.4861

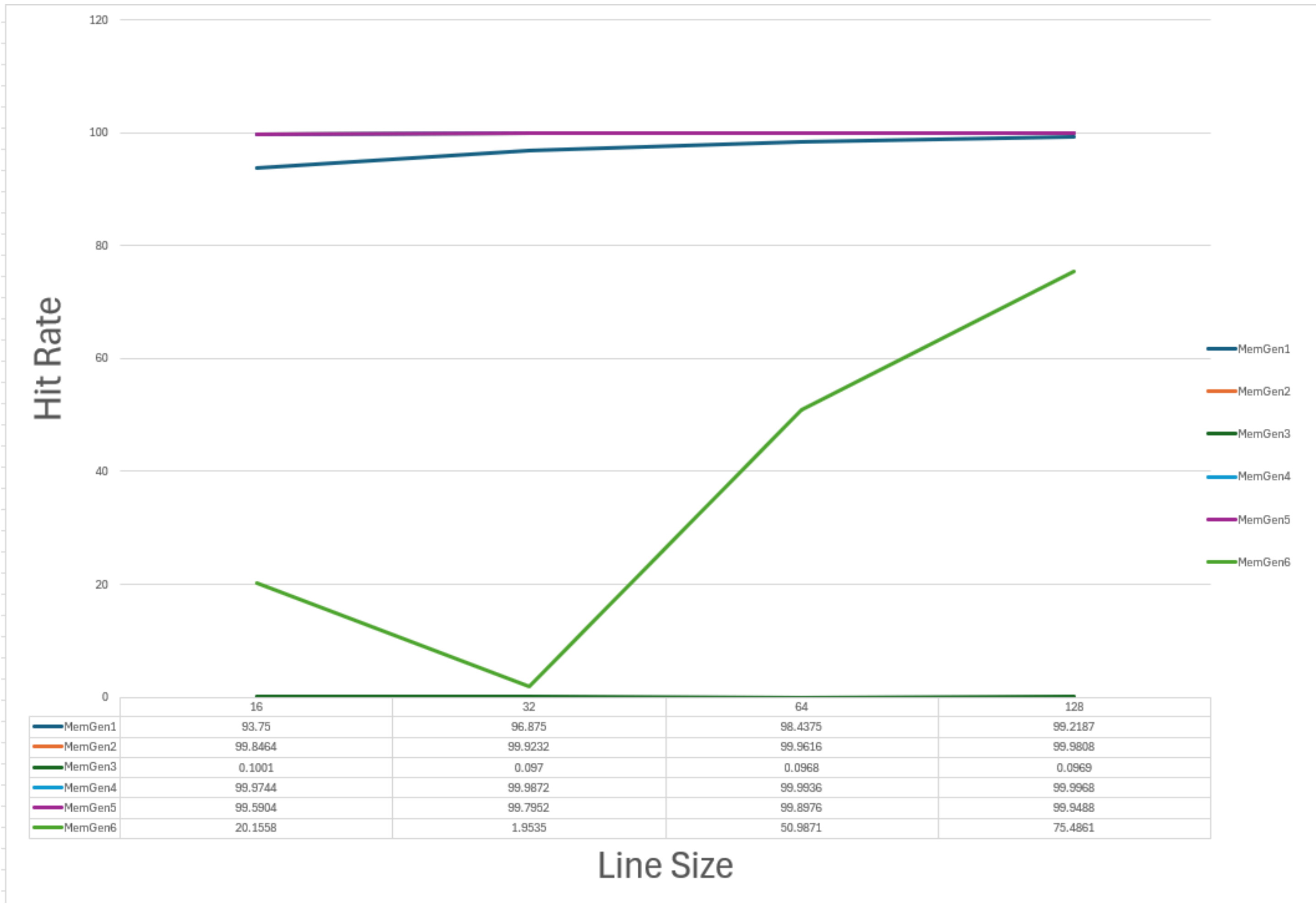


Direct Mapped Cache	Line Sizes			
Mem Gen functions	16	32	64	128
memgen1	93.75	96.875	98.4375	99.2187
memgen2	99.8464	99.9232	99.9616	99.9808
memgen3	0.1021	0.1021	0.1023	0.0978
memgen4	99.9744	99.9872	99.9936	99.9968
memgen5	99.5904	99.7952	99.8976	99.9488
memgen6	0	0	49.9999	74.9999

Graphical Output:

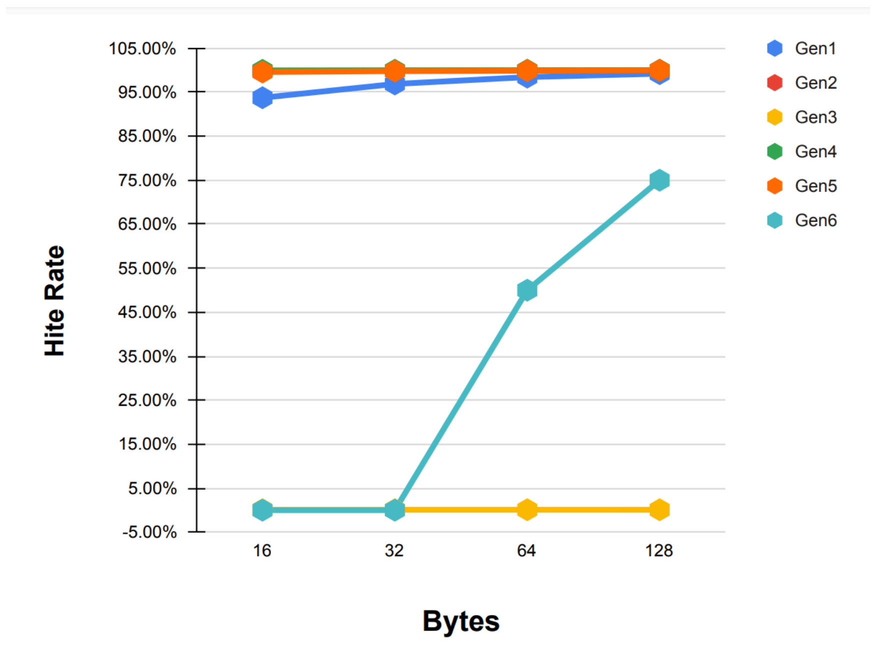
Below is the graphical representation of the tabulated data in the previous section. On the y-axis, we plot the Hit Ratio, and on the x-axis, we plot the line size.

Fully Associative cache curve:



	16	32	64	128
MemGen1	93.75	96.875	98.4375	99.2187
MemGen2	99.8464	99.9232	99.9616	99.9808
MemGen3	0.1001	0.097	0.0968	0.0969
MemGen4	99.9744	99.9872	99.9936	99.9968
MemGen5	99.5904	99.7952	99.8976	99.9488
MemGen6	20.1558	1.9535	50.9871	75.4861

Direct Mapped cache curve:



Comparison & Analysis:

For the Fully Associative cache, we see that for almost all the address generating functions, as the number of bytes (or line size) increases, the Hit Ratio tends to increase as well, this is because of the concept of spatial locality. The one functions that doesn't follow this pattern is MemGen6, which is decreasing for line sizes less than 32 and then is increasing after that, thus making the point corresponding to (32,1.9535) as the absolute minimum of that curve. As stated, the other 5 functions show trends of increase, though it differs by how much. For instance, MemGen1 increases at the start by 3.125 (from 16 to 32), then by 1.5625 (from 32 to 64); notice the reduction by almost half (from 3.125 to 1.5625)! The next increase is by 0.7812, again almost half of the previous increase. For the case of MemGen2, there is a very slight increase, but as we can see the hit ratio is already saturated at 100, so the curve is almost a horizontal line passing through 100 from 16 all the way to 128 bytes. The same argument can also be applied to MemGen4.

For the Direct Mapping cache, it can be observed that there are some variations in how the hit ratio changes with respect to line size. The hit ratio in memgen2, memgen4 and memgen5 is quite similar, where the hit ratio increases at a significantly low rate. The hit ratio increased by 0.01 or 0.14 at the most in these functions. Memgen1, which shows similar results, had a higher increase (from 93.75 to 99.2187). This higher increase could be due to the fact that memgen2, 4, and 5 had a very high hit ratio at the beginning of the graph using a line size of 16 bytes (99.5 and above), making those curves almost saturated. Therefore, it cannot show a higher increase in hit ratio than 0.5 unlike the function memgen1 which can theoretically have a 6.25 increase. The two functions that do not follow the same pattern are the functions memgen3 and memgen6. The memgen3 function shows an almost constant low hit ratio in all line sizes (ranging from 0.1023 down to 0.0978). Memgen6, on the other hand, has a zero hit ratio in line sizes 16 and 32 bytes; however, it shows a significant leap in hit ratio in both 64 and 128 line sizes.

Conclusion:

We can conclude from our data that the concept of spacial locality is important and it usually works for most applications, we also concluded that increasing the line size usually improves the hit rate however in some rare cases where the addresses are read in a certain order, increased line size might actually decrease the hit rate. We also concluded that if the memory access is completely random over a huge range thats not within the bounds of our cache the miss rate becomes minimal and almost zero. Moreover, if the memory access is random however it happens within a range that can fit or partially fit in the cache then over a long time and over many memory accesses we start seeing mostly hits as all the required memory is stored in the cache. By analyzing the difference between fully associative and directly mapped we can also say that the main difference is with placement and replacement where directly mapped doesn't work if a lot of the memory accesses will be mapped to the same index however apart from that their performance is highly comparable and in many cases is the same. Over the course of this project, we managed to complete a functional simulator that processes the hit ratio for both Direct Mapped and Fully Associative caches for four different line sizes: 16, 32, 64, and 128. We simulated over the above line sizes, collected the hit ratios, tabulated them, and finally plotted. Following that, an analysis of the two curves was conducted, which ultimately led to a growth in our understanding of how caches truly operate.