Assembly Language (Summer 2020)

Cache Performance Project (Direct Mapped Cache)

Andrew Nady & Mahmoud A Elshinawy

900184042 & 900183926

20th July 2020

Table of Contents

Design of Code:

The code is divided to

- Global declaration & macro
- Memory generators
- Direct cache Function
- Main function

1) Global declaration & macro

So there are some codes in the Marco defining the dram size and the cache size and the line size which then I used to compute the number of blocks (cache size/line size), the in order to get the number of bits of the index, I did log2(block number).
In order to get the word offset, I did " word offset=(log2(CACHE_LINE_SIZE/4)); "

We did an array of structs that contain a Boolean which indicates the valid bit and an integer for the tag.

2) Random Address Generators

```
unsigned int memGen2()
{
    static unsigned int addr=0;
    return (addr++)%(DRAM_SIZE);
}

unsigned int memGen3()
{
    static unsigned int addr=0;
    return  rand_()%(64*1024);
}

unsigned int memGen1()
{
    return rand_()%(DRAM_SIZE);
}
```

```
unsigned int memGen4()
{
    static unsigned int addr=0;
    return (addr++)%(1024*4);
}

unsigned int memGen5()
{
    static unsigned int addr=0;
    return (addr++)%(1024*64);
}

unsigned int memGen6()
{
    static unsigned int addr=0;
    return (addr+=64)%(128*1024);
}
```

The generators are classified into two categories:

a) Random number distant generator:
memGen1 and memGen 3 generate random numbers that most of them do not share the index, that is why we will not be able to make use of the spatial locality when using them.

b) Consequent number generator:

All other functions generate consequent numbers but they differ from each other in the range they generate in. When using them, we can get the maximum benefit of the spatial locality.

3) the target of the function is to indicate if the address is hit or miss, so we compare the valid bit of the address with the valid bit of the cache memory which is in the arrayy of struct, if the valid bit =1, then compares the tag with the tag, if the tags are equal then the this address will be labeled as hit, if one of the two failed, it will be labeled as miss.

So in order to implement that logic, we get the index and the tag by masking, then access the cache by the index, then compare the valid bit of the address with the valid bit inside the block which is inside the cache and do so with tag of both. If the conditions were right, it will return hit, if not return miss.

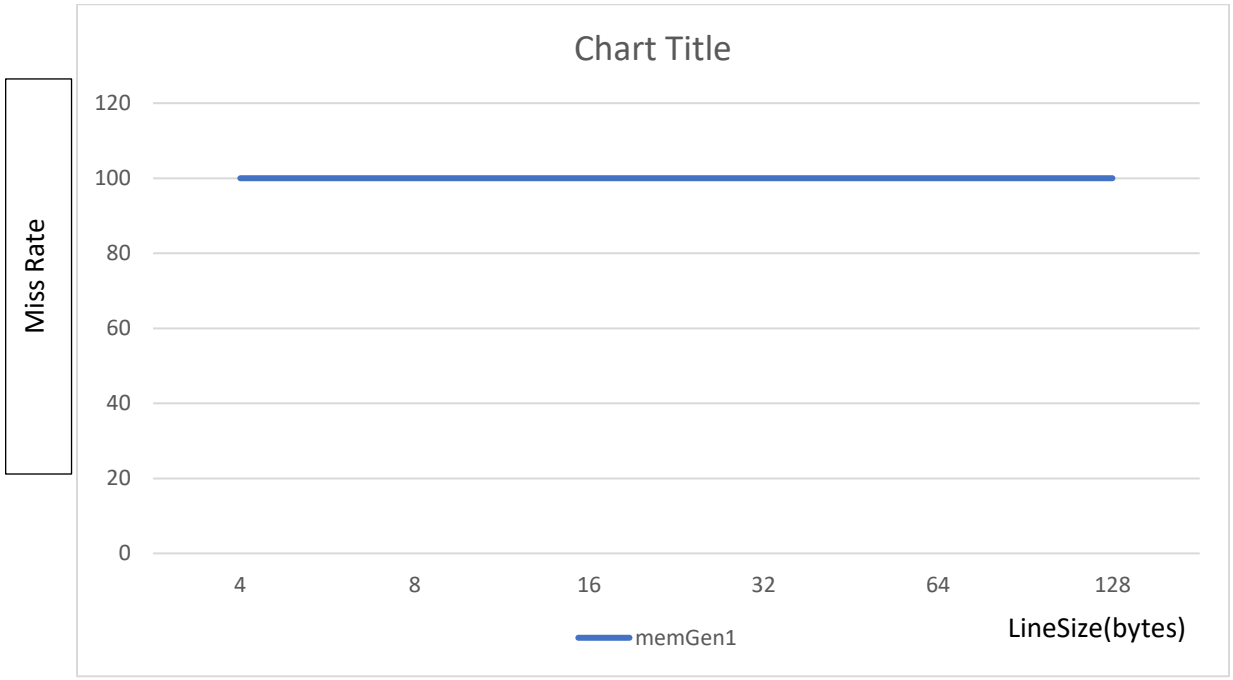
4) inside the main function, we iterate to generate the memory addresses using memgen functions. While we iterate 1000,000 times, the program adds a variable called by 1, if the direct cache simulator function returns hit. After finishing the 1000,000 loops, we printing the hit ration and the miss ratio

Experiment

In this project, the main objective is to simulate the cache and how it performs the direct map cache type. We used six different functions to generate random addresses in different ranges depending on different implementations to ensure that it has nothing to do with the address itself. We tested the performance depending on two setups:
The first to make the cache size fixed at 64 kb, then we change the line size of the cache, that is, how many bytes per block this cache has. We tested it for the values 4,8,16,32,46,128 bytes. The main goal of this part is to study and analyse the relation between the line size and the miss rate.
The second is to make the line size fixed at 16 bytes, that is, every single block has to have 16 bytes, no matter the size of the cache. Then, we vary the cache size starting from 8 kb,16 kb, 32 kb, 64kb. The main goal of this setup is to observe the relation between the cache size and the miss rate.
We use the code submitted as following; for each test case in every setup, we create a loop of 1000000 iteration in each of which we generate a random address using the given functions memGens. We collected the data in the spreadsheet attached to the project's ZIP file, and graphed the data to see the relation between those variables and the miss rate, and thus the cache performance. In the following, we are going to show the graphs and record the observations and the conclusions we can draw from each graph.

Graphs and Data Analysis

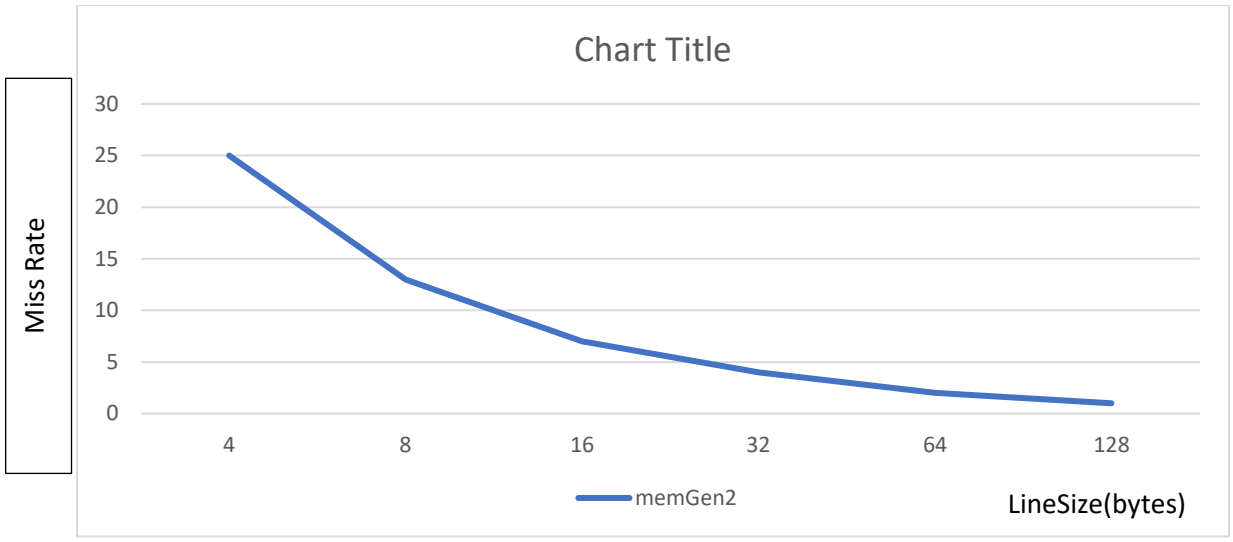
The First Setup (fixed cache size (64KB) and variable line size):



- Observation, the miss ration is 100%. The increasing of the line size doesnot affect the miss ratio. It is almost the same

- Conclusion, We used her generator 1, which generates random numbers from 0 to the size of the Dram. Since the the size of the cache size is only 64KB which is a subset of the Dram so always the memory address generated is always greater than the size of the cache which results in making it miss.

- When I tried to change the cache size to see if this will affect the result, the miss ratio decreases as I expected.



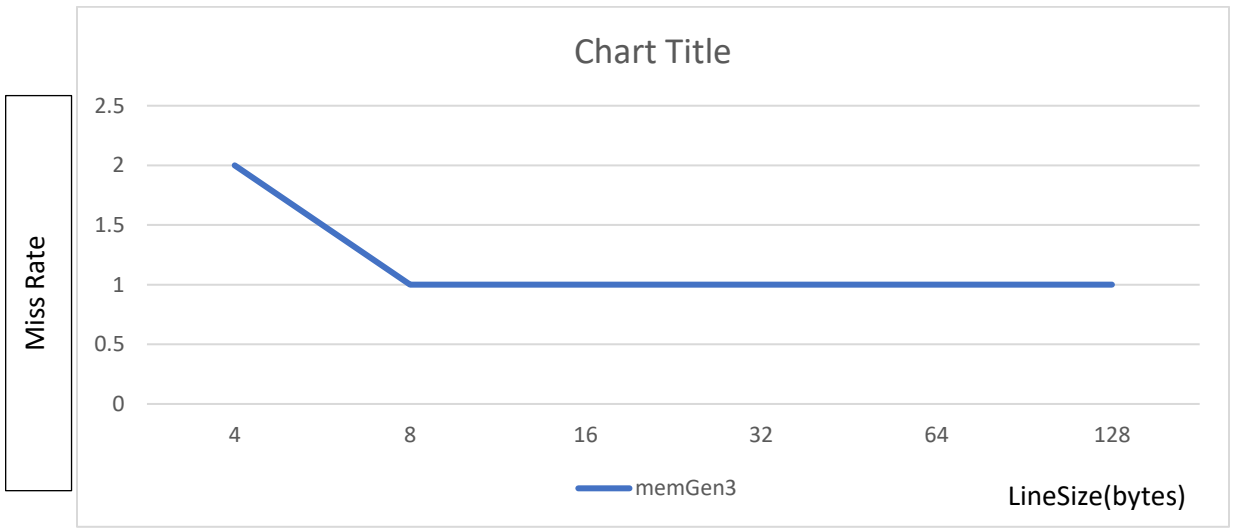
Observation:

it decreases by increasing the size line.

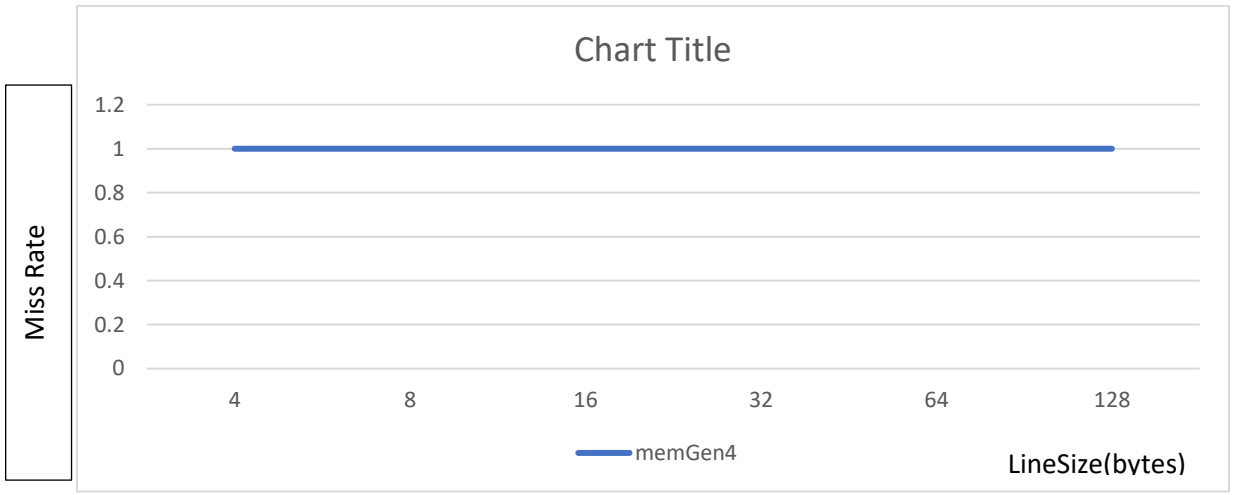
Conclusion:

memgen2 generates sequential addresses (0x1 , 0x2….dram size), so the if the line size is 4 which means that the block contains 1 word, so when 0x1 which is the first bit will be miss but the following 3 bits will be hit (because the line size is 4)

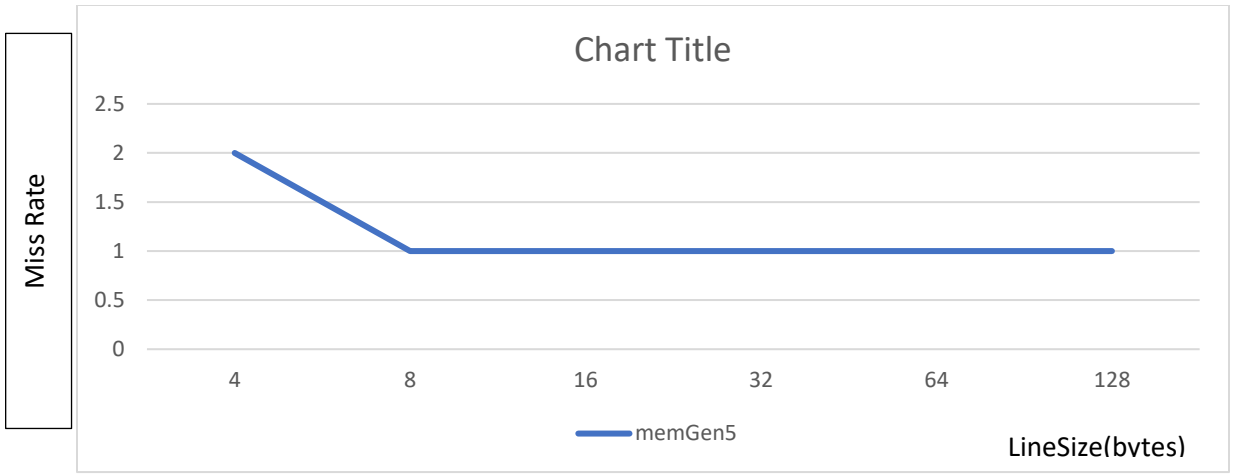
if the line size is 8, then each miss will follows by 7 hits, and so on … by increasing the number of line size the miss ratio decreases



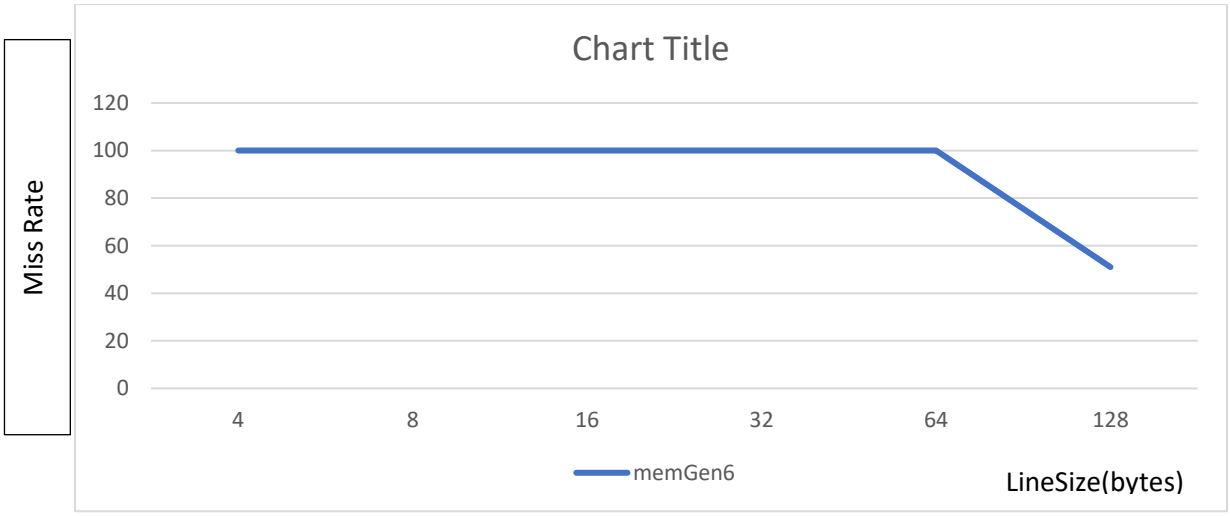
- Observation, When increasing the line size, the miss ratio decreases till it reaches to 1
- Conclusion, We used the generator 3, which generates random numbers from 0 to the 64*1024. When increasing the line size, the miss ratio decreases till it reaches to 1 (which can't be smaller)



- Observation: the miss ratio is fixed at 1%
- Conclusion: the mengen4: generates the address from 0 to 4*1024 and then begins again from 0 to 4*1024 till the number of iletrators ended. This makes all the address after the first" 0 to 4*1024" ended, are hit. So the miss ratio is 25% (if line size = 4)of the first 4096 address which is so small because there are million address generated.
- That is because the miss ratio is so small



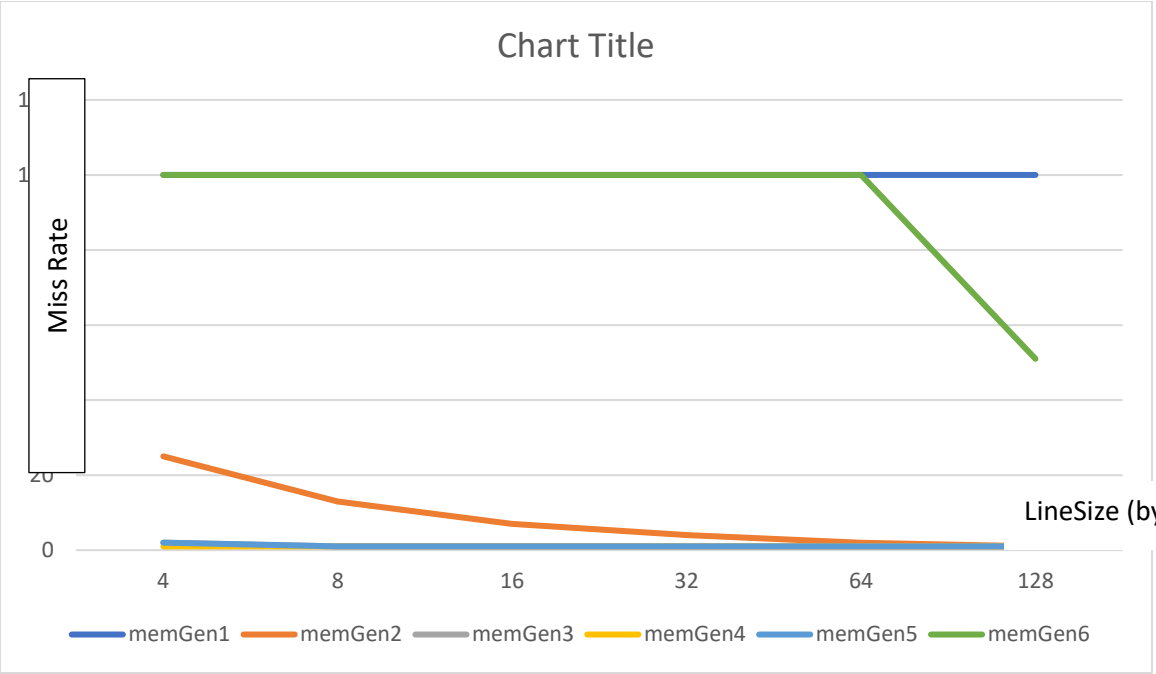
- Observation: the miss ratio decreases by increasing the line size till it reaches to 1%
- Conclusion: the mengen5: generates the address from 0 to 64*1024 and then begins again from 0 to 64*1024 till the number of iletrators ended. This makes all the address after the first" 0 to 64*1024" ended, are hit. So the miss ratio is 25% (if line size=4) of the first 65,536 address which is so small relative to the million address generated
- That is because the miss ratio is so small.
- It is decreases by increasing the line size because the 25% decreases by increases the lines size



- Observation: the miss ratio almost 100% in all line size below 128
- Conclusion: memgen6 generates addresses sequentially from 0 to 128*1024, where each address Is distant 64 bytes from the previous address. So, if there is a miss, there (line size -1) hits after it. So, if the line size 8, the address that are generated will always

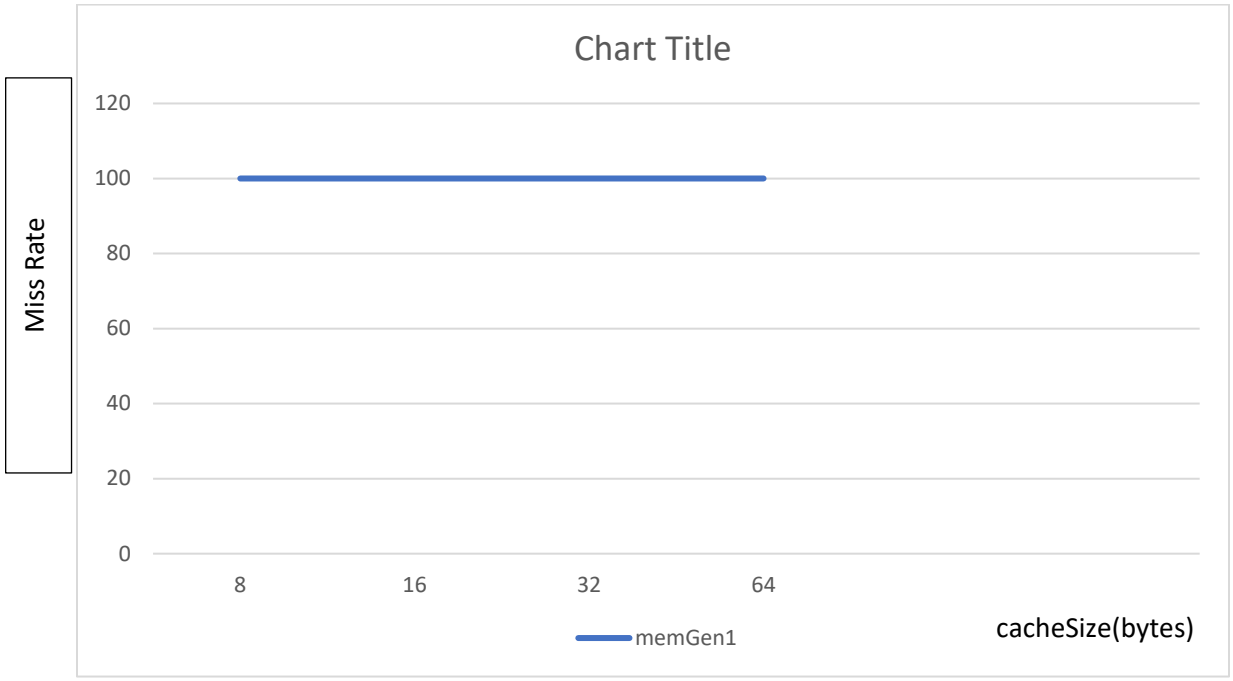
be 100 because the following 7 address won't be generated. Although, if the line size is 128 then if the address is miss, then the whole line (128 bytes) will be will be hit so, if the following address after the miss will be hit

- When the line size is 128, the miss 51% not 50%, because the first address is 0x64 which is included with the previous line. So the following address will be miss also because it is the beginning of new line

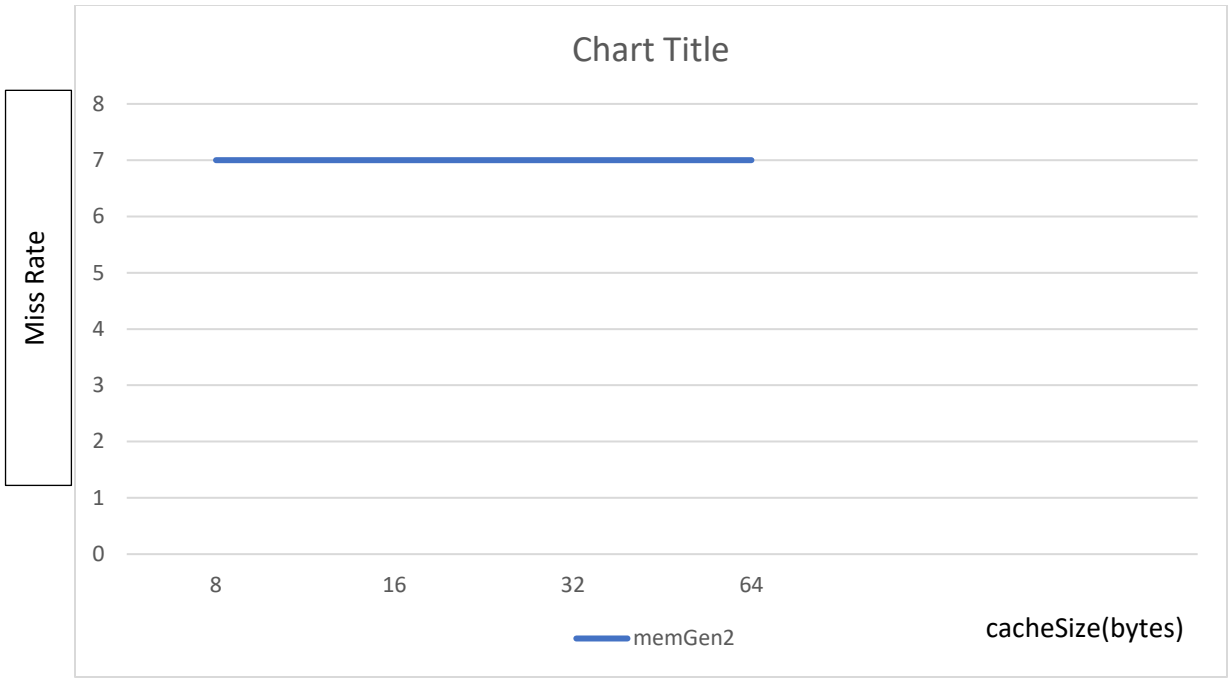


Setup 2 (Fixed line size & Variable cache size):

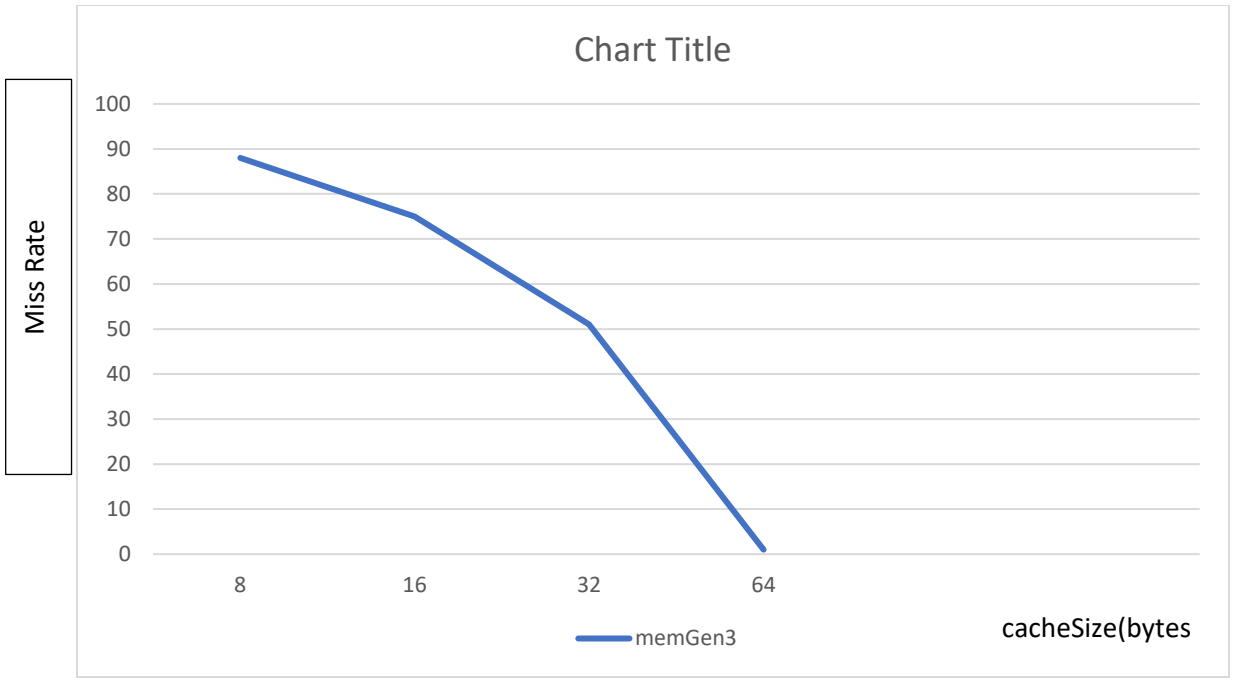
1)



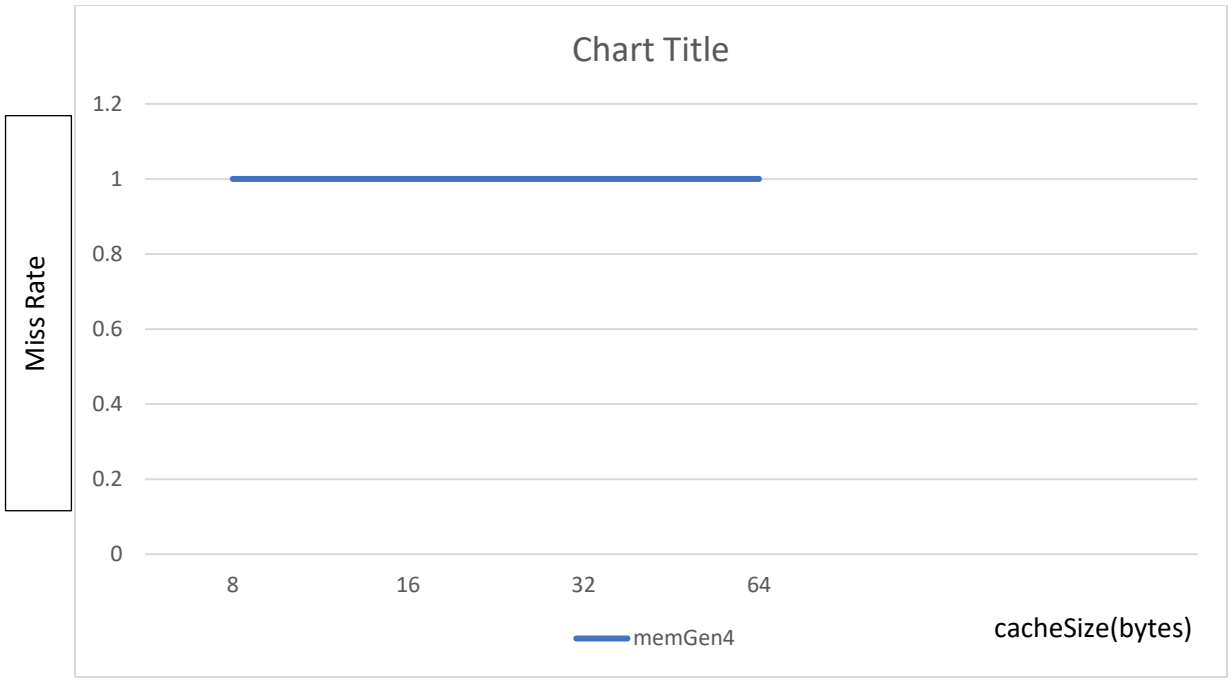
In this test case, we used memGen1 to generate random memory addresses. As you can see from the graph, the miss ratio stays the same at 100 percent by increasing the the cache size. The reason why the cache did not hit any address is that the function generates random numbers that are not repeated or consequent, and also in a very large range that the maximum size of the cache can not hold. The exact opposite is expected to happen with memGen3().



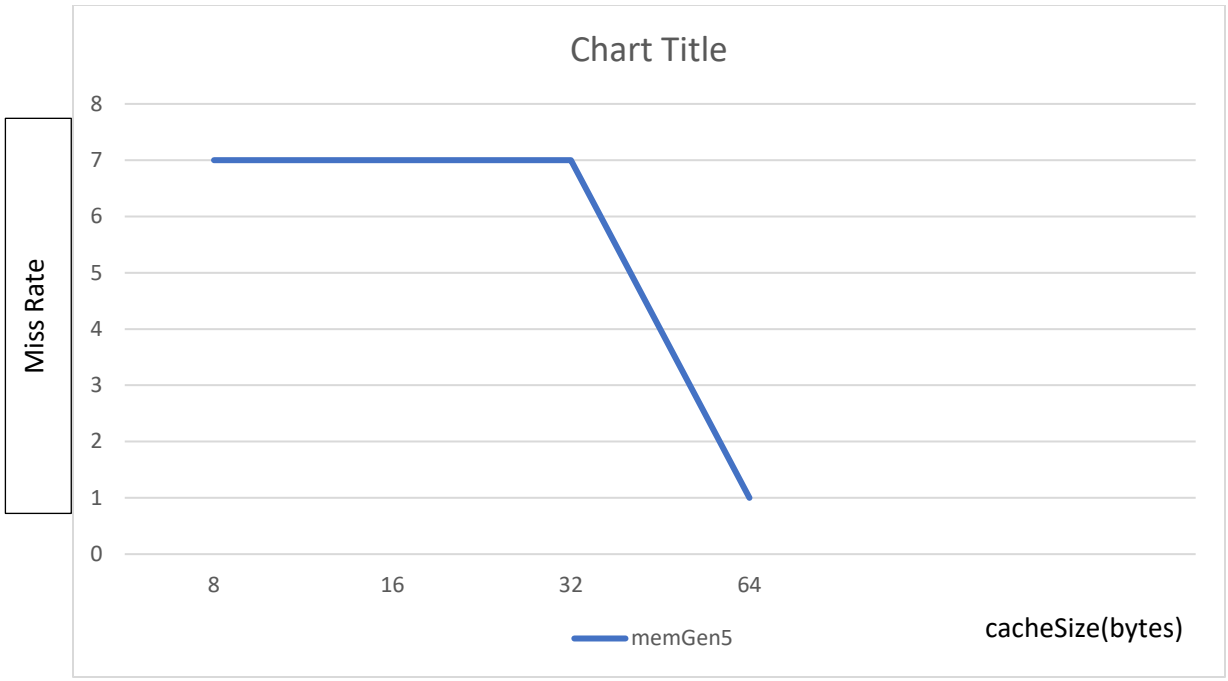
In this test case, we used memGen2 to generate random memory addresses. memGen2 generates consequent addresses, so the cache would take advantage of the spatial locality. As the line Size is fixed at 16 bytes, so for every MISS, the cache would fetch 16 bytes from the memory. That is why the behavior is 1 MISS followed by 15 HITs. The behavior do not change if we change the cache size because whether the cache size is big or small, the addresses are consequent, the maximum bytes the cache can load is 16 byte.



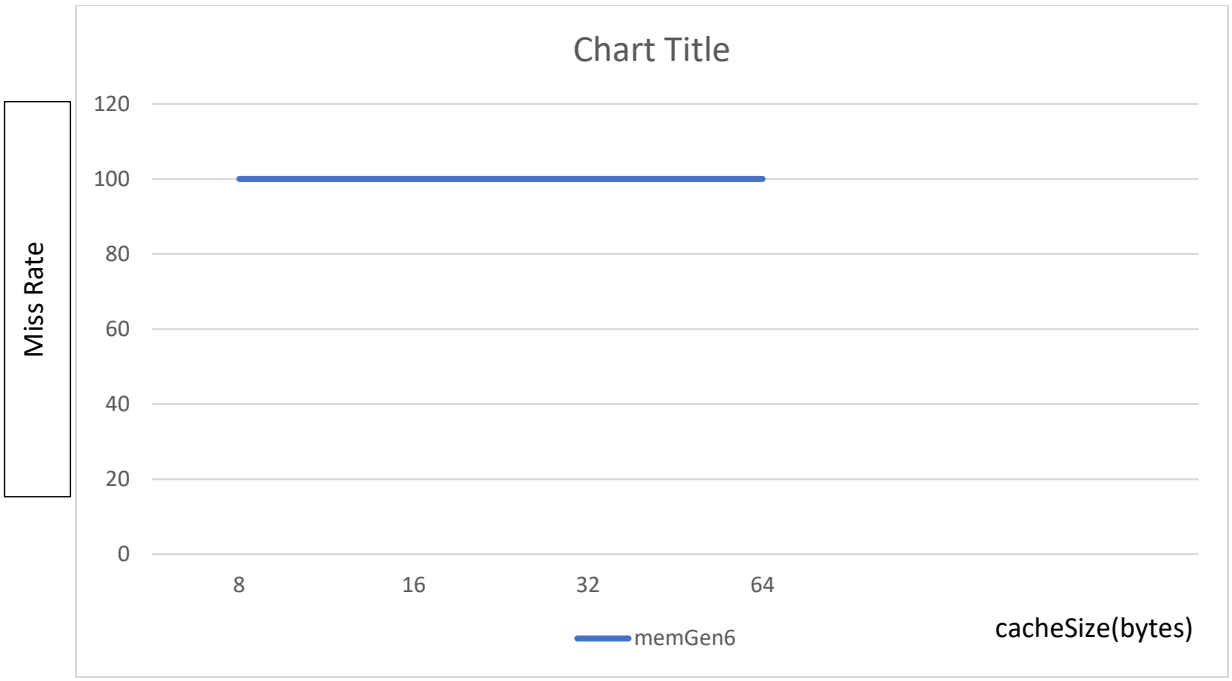
In this test case, we used memGen3 to generate random memory addresses. As you can see from the graph, the miss ratio drops by increasing the the cache size. The reason why the cache hit more addresses is that the function generates random numbers that are not repeated or consequent, but in a smaller range than memGen1. In this particular case, the more cache we have, the less numbers generated that we have to overwrite, the less miss rate will be.



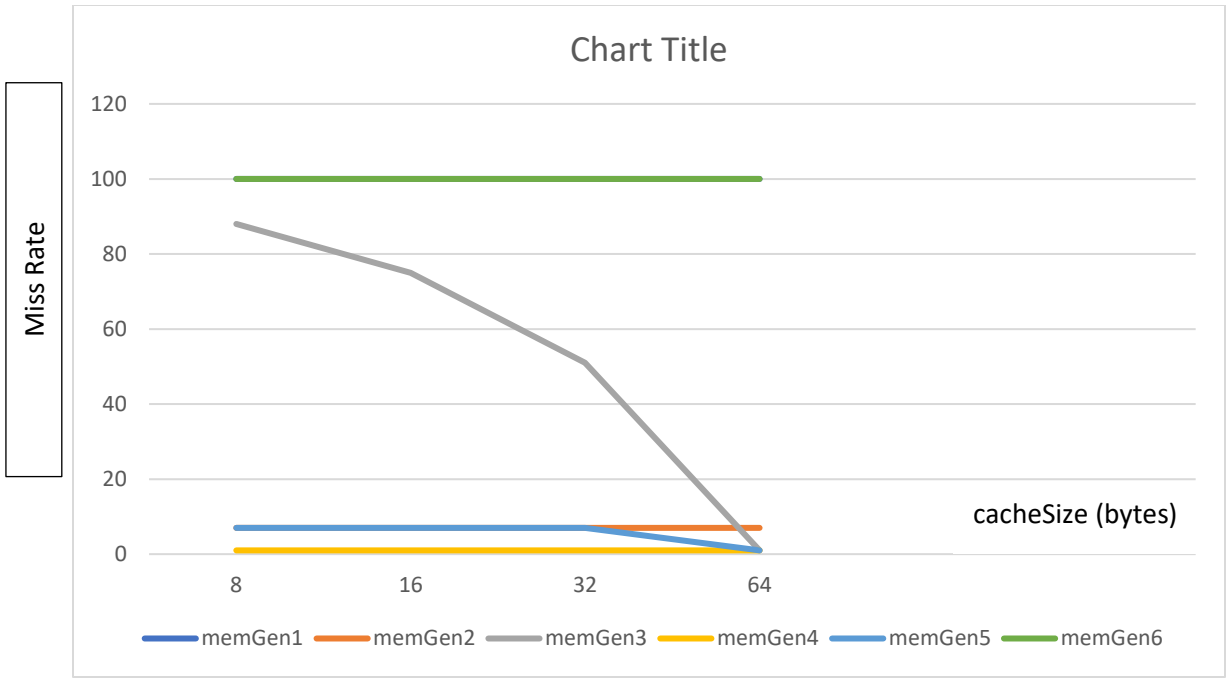
In this test case, we used memGen4 to generate random memory addresses. memGen2 generates consequent addresses, so the cache would take advantage of the spatial locality. As the line Size is fixed at 16 bytes, so for every MISS, the cache would fetch 16 bytes from the memory. That is why the behavior is 1 MISS followed by 15 HITs. The behavior do not change if we change the cache size because whether the cache size is big or small, the addresses are consequent, the maximum bytes the cache can load is 16 byte. It has also a smaller MISS rate than using memGen2 because its range is less. After that range, it starts to regenerate the numbers including those which gave a MISS in the first place, which will now issue a HIT every single time in the million iteration.



In this test case, we used the function memGen5 to generate the random addresses. It generates consequent numbers but in a larger range (64*1024). In the beginning, the cache size couldn't hold this range of numbers, that is why it has to overwrite some of the numbers inside it, and give a MISS. This is not the case with the 64 kb cache size. This cache can hold the whole range, so when it iterates the full million iteration, all addresses will give a HIT when the range is regenerated several times.



It generates consequent numbers but with a unit step 64. As we fix the line size to be 16, the cache will not make use of the spatial locality because between every address and the one that follows it equals 64 (more than 16). That is why we observe a constant relationship at a 100 MISS rate in the graph.



Conclusion, this graph concludes the second setup in which we vary the cache size and we fix the line size at 16 Bytes. Each test case uses different function to generate the random addresses, and you can easily identify each of them with its color indicated. In some test cases, the MISS ratio drops when we increase the cache size, other increases when we increase the cache size. This depends on the addresses we generate in every single test case, the more random they are, the more MISS ratio we get. It also depends on the range the function generates, the more it is, the more cache size we have to provide to proceed without overwriting.

Validation of Results:

This is how to validate our results. We iterate over the loop a fewer times to be able to keep track of the addresses copied to the cache and determine whether the cache is working as it is meant to work. We fixed the line size at 8 bytes and the cache size at 64 kb and tried the different functions that generate the random addresses. We iterated only 20 times. Here are the screenshots labeled for the different memGen functions. This is an example of how we tried to validate our results, we did in all other cases to be sure that we were right.

```
maco@maco-VirtualBox:/media/sf_Assembly/Projects/Project 3$ ./a.out
Direct Mapped Cache Simulator
0x00715819 (Miss)
0x58239316 (Miss)
0x38812584 (Miss)
0x16428375 (Miss)
0x20433936 (Miss)
0x42275409 (Miss)
0x20838482 (Miss)
0x33622392 (Miss)
0x06031520 (Miss)
0x47845968 (Miss)
0x44730907 (Miss)
0x46963704 (Miss)
0x03681719 (Miss)
0x22369172 (Miss)
0x17018153 (Miss)
0x10065639 (Miss)
0x16748486 (Miss)
0x27065303 (Miss)
0x29696153 (Miss)
0x18126844 (Miss)
Hit ratio = 0
Miss ratio = 100
maco@maco-VirtualBox:/media/sf_Assembly/Projects/Project 3$
```

memGen 1()

memGen1() generates completely different numbers. When the cache loads the first address, it loads the address and 7 after it. Because all the numbers generated are distant from each other, even making use of spatial locality will not work.

```
maco@maco-VirtualBox:/media/sf_Assembly/Projects/Project 3$ ./a.out
Direct Mapped Cache Simulator
0x00000000 (Miss)
0x00000001 (Hit)
0x00000002 (Hit)
0x00000003 (Hit)
0x00000004 (Hit)
0x00000005 (Hit)
0x00000006 (Hit)
0x00000007 (Hit)
0x00000008 (Miss)
0x00000009 (Hit)
0x00000010 (Hit)
0x00000011 (Hit)
0x00000012 (Hit)
0x00000013 (Hit)
0x00000014 (Hit)
0x00000015 (Hit)
0x00000016 (Miss)
0x00000017 (Hit)
0x00000018 (Hit)
0x00000019 (Hit)
Hit ratio = 85
Miss ratio = 15
maco@maco-VirtualBox:/media/sf_Assembly/Projects/Project 3$
```

memGen 2()

```
maco@maco-VirtualBox:/media/sf_Assembly/Projects/Project 3$ ./a.out
Direct Mapped Cache Simulator
0x00000000 (Miss)
0x00000001 (Hit)
0x00000002 (Hit)
0x00000003 (Hit)
0x00000004 (Hit)
0x00000005 (Hit)
0x00000006 (Hit)
0x00000007 (Hit)
0x00000008 (Miss)
0x00000009 (Hit)
0x00000010 (Hit)
0x00000011 (Hit)
0x00000012 (Hit)
0x00000013 (Hit)
0x00000014 (Hit)
0x00000015 (Hit)
0x00000016 (Miss)
0x00000017 (Hit)
0x00000018 (Hit)
0x00000019 (Hit)
Hit ratio = 85
Miss ratio = 15
maco@maco-VirtualBox:/media/sf_Assembly/Projects/Project 3$
```

memGen 3()

memGen2() and memGen3() generate consequent addresses, so the cache makes the full use of the spatial locality. When it issues a MISS with the address 0x 0, it loads it, and the seven consequent address, so when the address 1,2---7 is generated, it issued a hit.

```
maco@maco-VirtualBox:/media/sf_Assembly/Projects/Project 3$ ./a.out
Direct Mapped Cache Simulator
0x00000000 (Miss)
0x00000001 (Hit)
0x00000002 (Hit)
0x00000003 (Hit)
0x00000004 (Hit)
0x00000005 (Hit)
0x00000006 (Hit)
0x00000007 (Hit)
0x00000008 (Miss)
0x00000009 (Hit)
0x00000010 (Hit)
0x00000011 (Hit)
0x00000012 (Hit)
0x00000013 (Hit)
0x00000014 (Hit)
0x00000015 (Hit)
0x00000016 (Miss)
0x00000017 (Hit)
0x00000018 (Hit)
0x00000019 (Hit)
Hit ratio = 85
Miss ratio = 15
maco@maco-VirtualBox:/media/sf_Assembly/Projects/Project 3$
```

memGen 4()

```
maco@maco-VirtualBox:/media/sf_Assembly/Projects/Project 3$ ./a.out
Direct Mapped Cache Simulator
0x00000000 (Miss)
0x00000001 (Hit)
0x00000002 (Hit)
0x00000003 (Hit)
0x00000004 (Hit)
0x00000005 (Hit)
0x00000006 (Hit)
0x00000007 (Hit)
0x00000008 (Miss)
0x00000009 (Hit)
0x00000010 (Hit)
0x00000011 (Hit)
0x00000012 (Hit)
0x00000013 (Hit)
0x00000014 (Hit)
0x00000015 (Hit)
0x00000016 (Miss)
0x00000017 (Hit)
0x00000018 (Hit)
0x00000019 (Hit)
Hit ratio = 85
Miss ratio = 15
maco@maco-VirtualBox:/media/sf_Assembly/Projects/Project 3$
```

memGen 5()


maco@maco-VirtualBox:/media/sf_Assembly/Projects/Project 3$ ./a.out
Direct Mapped Cache Simulator
0x00000004 (Miss)
0x00000003 (Miss)
0x00000002 (Miss)
0x00000001 (Miss)
0x00000000 (Miss)
0x00000004 (Hit)
0x00000003 (Hit)
0x00000002 (Hit)
0x00000001 (Hit)
0x00000000 (Hit)
0x00000004 (Hit)
0x00000003 (Hit)
0x00000002 (Hit)
0x00000001 (Hit)
0x00000000 (Hit)
0x00000004 (Hit)
0x00000003 (Hit)
0x00000002 (Hit)
0x00000001 (Hit)
0x00000000 (Hit)
Hit ratio = 75
Miss ratio = 25
maco@maco-VirtualBox:/media/sf_Assembly/Projects/Project 3$

memGen 6()