# HW01 Report:-

# Reverse Engineering AMD Ryzen 7 5800H

OS :- Windows 11, Ubuntu 22.04

Microarchitecture :- Zen 3

Q1) Basic Processor Features :-

a) Confirm if the Processor has a cache :-

Methodology:-

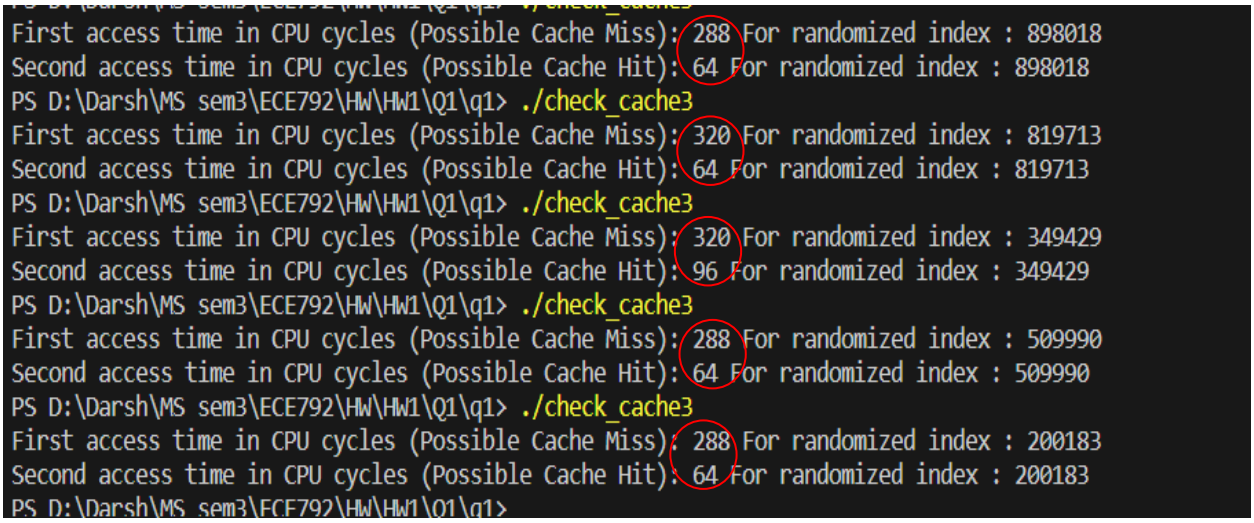
The method we use to confirm whether the processor has a cache or not is to measure the memory access time for accessing a randomized index from an array twice.

Basic idea is that when we first try to access a particular data in an array, CPU will 1st try to search the cache of that core. As it doesn't detect that data in the cache, it requests data from higher level cache or main memory. Due to this, the execution time of the given function will be very high. Execution time also includes the memory access time. When we try to access the same data for the second time, we get the results much faster as data is already in some form of memory which is not the main memory, because of difference in the execution time. This confirms that cache is present in the Processor.

Besides is the function which I am using to access the address.

```
uint64_t time_access(volatile uint8_t *addr) {
    uint64_t time1, time2;
    volatile uint32_t tmp;

    // Serializing instruction before timing
    __asm__ volatile (
        "CPUID\n\t"
        "RDTSC\n\t"
        : "=a"(time1)
        : : "%rbx", "%rcx", "%rdx"
    );

    // Access the value
    tmp = *addr;

    // Serializing instruction after timing
    __asm__ volatile (
        "RDTSCP\n\t"
        : "=a"(time2)
        : : "%rbx", "%rcx", "%rdx"
    );

    return time2 - time1;
}
```

Results :-




Analysis:-

From the above results, we can see that a cache miss takes around 288 cycles and cache hit takes around 64 cycles. In addition to this I am also making sure to access randomized index so that prefetcher cannot be effective here.

b) Determine the number of Cache Levels :-

The method I used to detect the number of Cache Levels, is based on the memory access time it takes for each cache to access its data. As L1 is closer to Processor core than L2, it's memory access time will be lesser. I am using this same logic for L2 and L3 as well.

At the same time, I am making sure to access the random data values of upto 1Kb data size. So I will get the cache access time of 1Kb of data for each level of Cache. Then I make sure to divide the time I got by 1024, to see the access time for 1 data value, which assuming to be of 1 Byte.

I define a multiple size right from 4Kb to 64Mb. Then I make an array of those sizes. After I have allocated memory for the array, I make sure to access the whole array once, so that it comes into the Cache. After they come into the Cache, I measure the time it takes to access 1Kb of data which is uniformly random. So, here we can expect some data will be stored in L1 cache, L2 cache and L3 cache. So the time we get in total, is sum of access time of L1 caches for some data, access time of L2 cache for some data ,L3 cache for some data and Main Memory for some data.

The downside of this is, that sometimes, we don't see a lot of difference in the access time through which we can infer the number of Cache levels. I ran this experiment multiple times and got results through which I could not infer anything.

After conducting the experiment, I can conclude that there are 3 levels of Cache in the Processor of my laptop. I can also conclude that size of L1 cache is 32 Kb, L2 size is of 512 Kb and L3 cache which is shared between all cores is 16 Mb. After 16 Mb, the time increases quite significantly. So I can conclude that data is being accessed from the main memory.

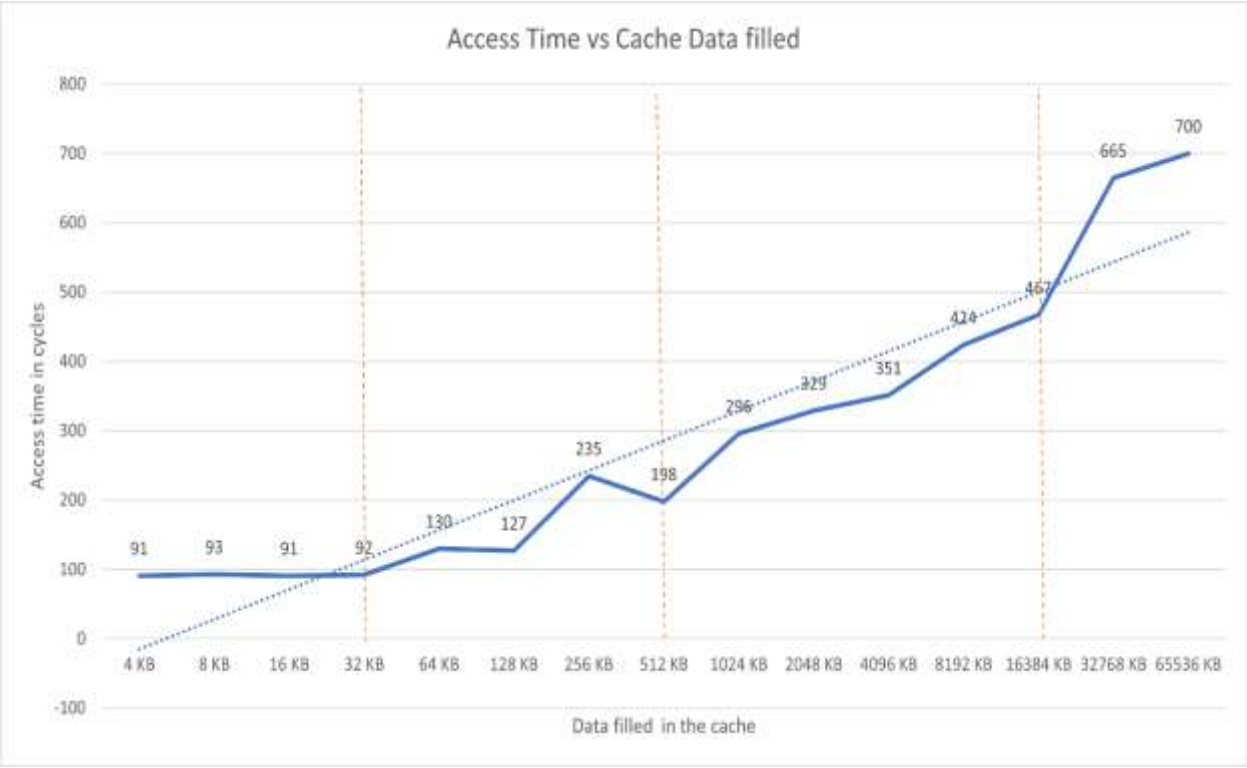
```cpp
// Pseudo-random Memory Access within the given size range
void accessData(int* array, int size) {
    for(int i =0;i<size;i++){
        volatile int temp = array[i];
    }
}

void accessData_1kb(int* array, int size) {
    // Create a random device and generator
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distr(size/2, size-1);
    for(int i = 0;i<1*1024;i++){
        volatile int temp = array[distr(gen)];
    }
}
```

Snippet of code through which I am accessing memory

Results :-

```
Size: 4KB, Time: 90
Size: 8KB, Time: 92
Size: 16KB, Time: 90
Size: 32KB, Time: 92
Size: 64KB, Time: 92
Size: 128KB, Time: 92
Size: 256KB, Time: 197
Size: 512KB, Time: 223
Size: 1024KB, Time: 226
Size: 2048KB, Time: 176
Size: 4096KB, Time: 135
Size: 8192KB, Time: 254
Size: 16384KB, Time: 416
Size: 32768KB, Time: 575
Size: 65536KB, Time: 563
```

Sample snip of the results




Here in this graph, I can observe, that time stays constant till 32 Kb, then it begins to rise, that where I put my mark. Then time increases for 256Kb, but it also decreases at 512Kb. So we can say that there can be a small issue at 256 Kb, which I am not sure about. My hypothesis is that some other processor could have messed up the timings at that specific memory. After 512Kb, we see constant rise, so I concluded that my L2 ends at 512Kb. After 16 Mb, we can see a very steep rise. So, I can conclude that data is accessed from main memory.

c) Confirm if Processor has Branch Prediction :-

Methodology:-

The method which I am using here, is that I am comparing the execution time of a loop with conditional branch and a loop without conditional branch. I also made sure to introduce strides here, so that there is more variation in the pattern we witness.

Idea here is that if my Processor has a Branch prediction, it should be able to detect branches and predict what branch it will go to. This will result in execution time of Loop without branches and the one with branches to be almost similar.

Results :-

```
Execution Time Without Branches with stride 1 : 704 microseconds
Execution Time With Branches with stride 1 : 808 microseconds

Execution Time Without Branches with stride 2 : 343 microseconds
Execution Time With Branches with stride 2 : 449 microseconds

Execution Time Without Branches with stride 4 : 172 microseconds
Execution Time With Branches with stride 4 : 223 microseconds

Execution Time Without Branches with stride 8 : 86 microseconds
Execution Time With Branches with stride 8 : 98 microseconds

Execution Time Without Branches with stride 16 : 43 microseconds
Execution Time With Branches with stride 16 : 55 microseconds

Execution Time Without Branches with stride 32 : 22 microseconds
Execution Time With Branches with stride 32 : 24 microseconds

Execution Time Without Branches with stride 64 : 11 microseconds
Execution Time With Branches with stride 64 : 12 microseconds

Execution Time Without Branches with stride 128 : 5 microseconds
Execution Time With Branches with stride 128 : 6 microseconds

Execution Time Without Branches with stride 256 : 3 microseconds
Execution Time With Branches with stride 256 : 3 microseconds

Execution Time Without Branches with stride 512 : 2 microseconds
Execution Time With Branches with stride 512 : 1 microseconds
```

Analysis:-

From the Results we can conclude that, execution time of executing loop with branches and without branches is not very high. As we increases the stride, the time for execution decreases. The reason for that is, that it is using less number of points for such execution.

For this I have varied strides from 2 to 512.  As strides increases, the difference between execution times also decreases, which indicates that branch predictor works efficiently.

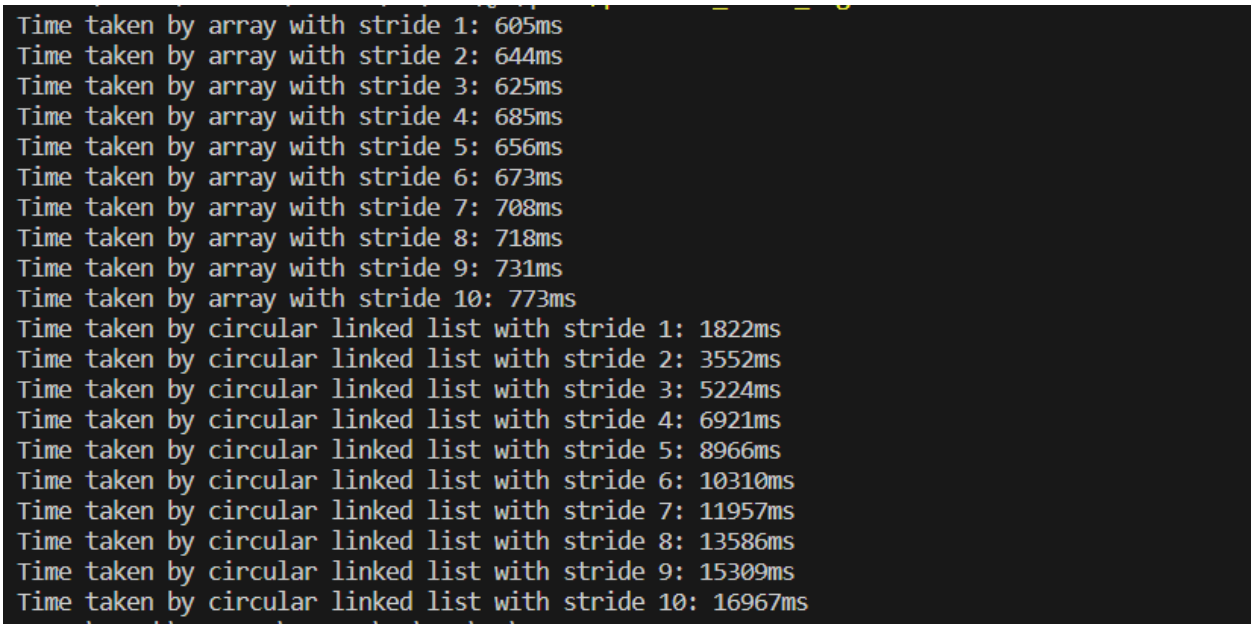
d) Confirm if Processor has Basic Prefetching Technique: -

Methodology:-

My idea here is to compare the execution time of accessing the array with strides and accessing the Linked List with different strides. Prefetcher can compute the memory addresses of array very efficiently, as it works on the principle of spatial locality, which says that if a particular memory is accessed, it neighboring elements will also be accessed in the near future.

Prefetcher fetches the cache line. Cache line is made up of contiguous memory elements. Array is made up of contiguous memory elements. Linked List is not made up of contiguous memory elements. So each time I fetch a element from Linked List, I might have to access a lot of cache lines. So the Prefetcher won't be able to predict which memory will be accessed in case of Linked List.

Results :-




Here I am using Circular Linked List to maintain constant memory accesses even if I am taking strides.

Analysis:-

Based on the Results above, we can say that Prefetcher exists in my Processor. I make this conclusion, on the basis that execution time for Array accesses is way less than that of Linked List accesses, even though the number of accesses are constant for both.

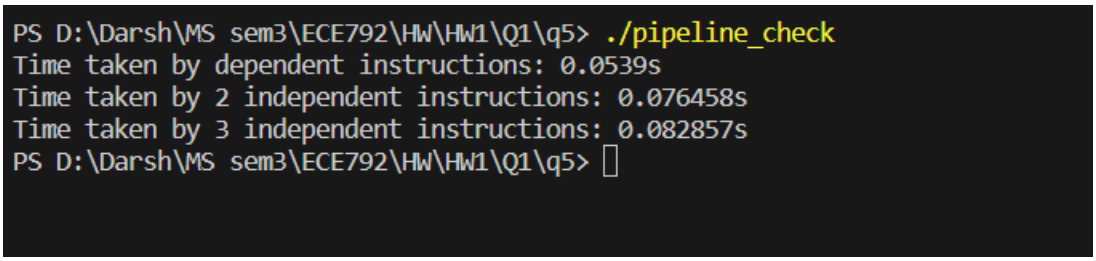
e) Confirm if the Processor is Pipelined:-

Methodology:-

My idea here is to run the 3 loops of same size but a large number like 100000000. Each loop will have different number of independent instructions. 1st loop will have 1 independent instruction. 2nd loop will have 2 independent instructions and 3rd loop will have 3 independent instructions. Each instruction does the same function of multiplying with 2, just with different variables.

So if processor is not pipelines, the time to run 3 instructions should be roughly be equal to 3 times the execution time of running a single instruction.

Results :-




Here we can see the execution time it takes to for all 3 instructions to run and it is not the multiple of the execution time of the 1st instructions.

Analysis:-

Based on this we can conclude, that some processes are taking place in parallel with each other. It is not due to multicore architecture of the Processor as I have explicitly assigned only 1 core for the execution of the program. So this proves the existence of Pipeline in Zen 3 Architecture.

f) Confirm if Processor is executing Out of Order :-

Methodology :-

My idea to check if the Processor is running Out of Order is that, we make 2 functions, one which executes bunch of independent instructions and other one executes the bunch of dependent instructions. Sample size of the number of instructions is same for both of them.

Now the main idea is that if Processor executes, Out of Order, there will be difference in the execution timing of Independent instructions and Dependent Instructions. Dependent instructions will stall the Processor, while the Processor executes other Independent instructions. When the dependencies will be solved, those dependent instructions will be executed. Other wise, the timing for both the dependent and Independent instructions would be same in an In Order Processor.

Results :-

```
PS D:\Darsh\MS sem3\ECE792\HW\HW1\Q1\q6> g++ -O0 ooo_check.cpp -o ooo_check
PS D:\Darsh\MS sem3\ECE792\HW\HW1\Q1\q6> ./ooo_check
Dependent instructions execution time: 2.26317 s
Independent instructions execution time: 1.66504 s
The processor seems to be able to execute instructions out of order!
PS D:\Darsh\MS sem3\ECE792\HW\HW1\Q1\q6> ./ooo_check
Dependent instructions execution time: 2.26013 s
Independent instructions execution time: 1.67283 s
The processor seems to be able to execute instructions out of order!
PS D:\Darsh\MS sem3\ECE792\HW\HW1\Q1\q6> ./ooo_check
Dependent instructions execution time: 2.2549 s
Independent instructions execution time: 1.67252 s
The processor seems to be able to execute instructions out of order!
PS D:\Darsh\MS sem3\ECE792\HW\HW1\Q1\q6> ./ooo_check
Dependent instructions execution time: 2.35403 s
Independent instructions execution time: 1.76609 s
The processor seems to be able to execute instructions out of order!
```

Analysis :-

Here we can notice the timing difference between the Dependent and Independent Instructions. Independent Instructions can execute Out of Order and be very Quick. But Dependent Instructions are stalled. So, their Execution takes more time than the Independent Instructions.

Q2)

a) Measure the Cache Line of the Processor :-

The general concept would be to write a program that tests memory access times for different strides through an array of data. When the stride size equals the cache line size, you should see a notable increase in access time, because each memory access will have to fetch a new cache line.

Now this concept works because, cache controller generally brings contiguous memory block know as cache lines into the cache. Suppose we have a continuous memory address from A0-A1000 in main memory, let our cache line size be 16. So, when it brings cache line with memory address A0-A15 into the cache. Now depending on the striding, we can see if it is equal stride is equal to cache line size or not. Because stride of 1 will cause a lot of cache hits, like after 1st miss all 15 will be cache hit, in stride of 2 that will be 1 miss and 7 cache hits, but in case stride of 16, there will be 1 cache miss, and then it will go to A16, which is not in cache line, so again a cache miss, and so on.

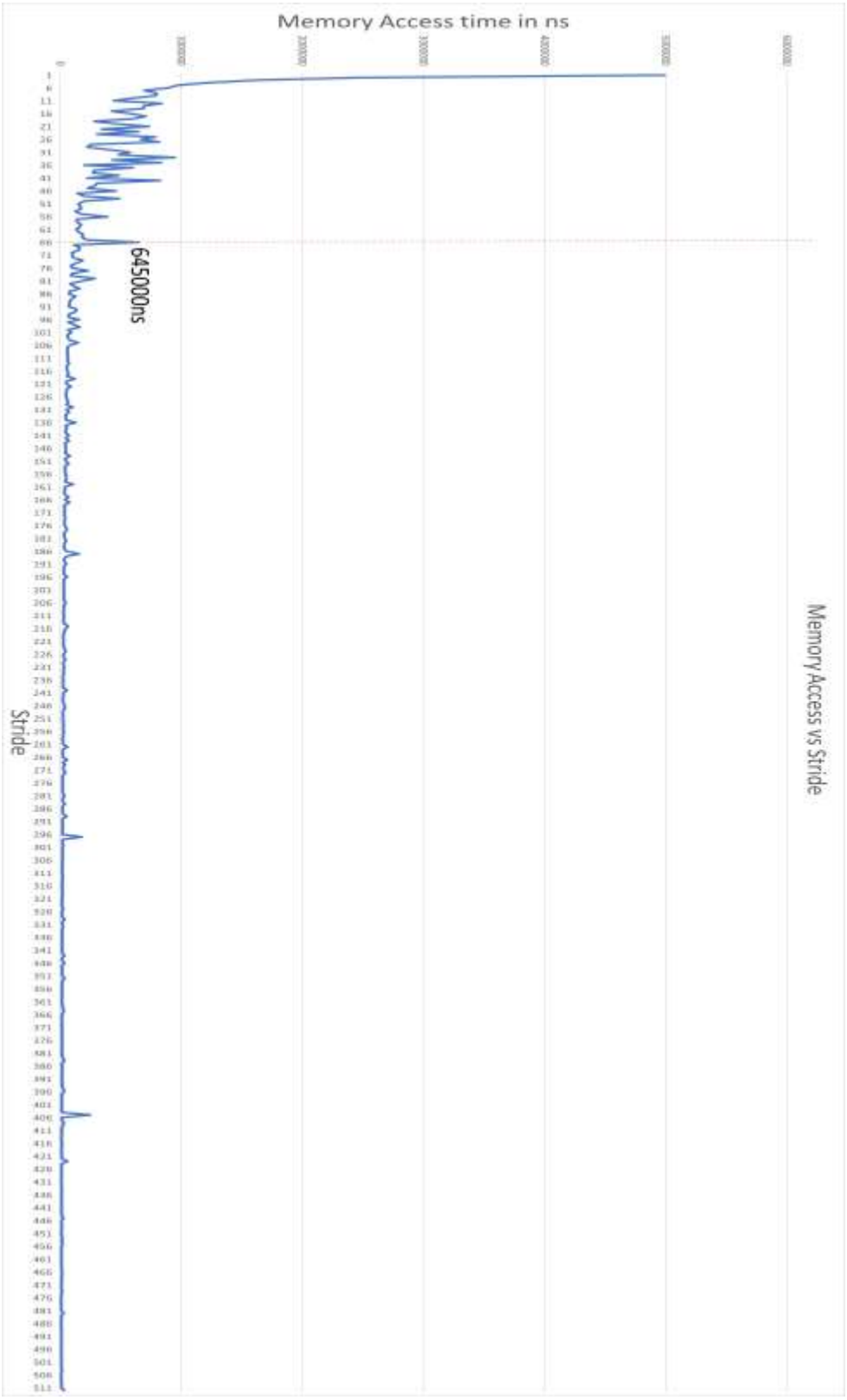
This is the basic idea behind finding the Cache Line of the Processor. I keep the array size constant for 2 Mb of data. I make variation in the Stride from 1 to 512 and see the execution time for each of them.

Before beginning these processes, I make sure that the array is already in the Cache, for that I access the data before I begin my timing operations.

```
auto startTime = high_resolution_clock::now();

for (int i = 0; i < arraySize; i += stride){
    array[i] *= 3;
    volatile int temp = array[i];
}
```

I use this loop to access the elements in stride. Multiplication operation is used to get better resolution. Merely accessing the strides of data gives a lot less timing.

Results :-


Memory Access time in ns
645000ns
Stride
Memory Access vs Stride

Analysis :-

From the Graph above, we can see that there is a very sharp increase in the peak when the Stride is 66. This means that Cache Line Size is equal to 66. Execution timings for the smaller strides is large because, because they are accessing a lot more data than other strides. So even if their execution time is high, they are not cache lines as their they are accessing more number of elements. I was experiencing a lot of issues with RDTSC timer in this microbenchmark, so I preferred to switch to the chrono library. Therefore there are lot of spikes in the Graph, because, Chrono library instruction have their own Latencies.

b) Determine the Cache Miss Latency of L1, L2 and L3 Cache :

Methodology :-

My idea here to measure Cache Latency is to measure the time taken of Cache miss for certain number of data values and then divide the timing by that number. I divide the total timing so that I can get the Cache miss latency for 1 cache miss.

I define 3 array sizes of 4 Kb, 256Kb and 8Mb. I choose these values as they are lesser than the size of L1 Cache, L2 Cache and L3 Cache respectively. I make sure to warm up my cache before I begin this experiment, because a lot of Latency in case I did not access my cache before this experiment.

I also make sure to assign the execution of whole code to 1 core of the Processor, and introduce Instruction barriers so no other process interferes the execution of the code.

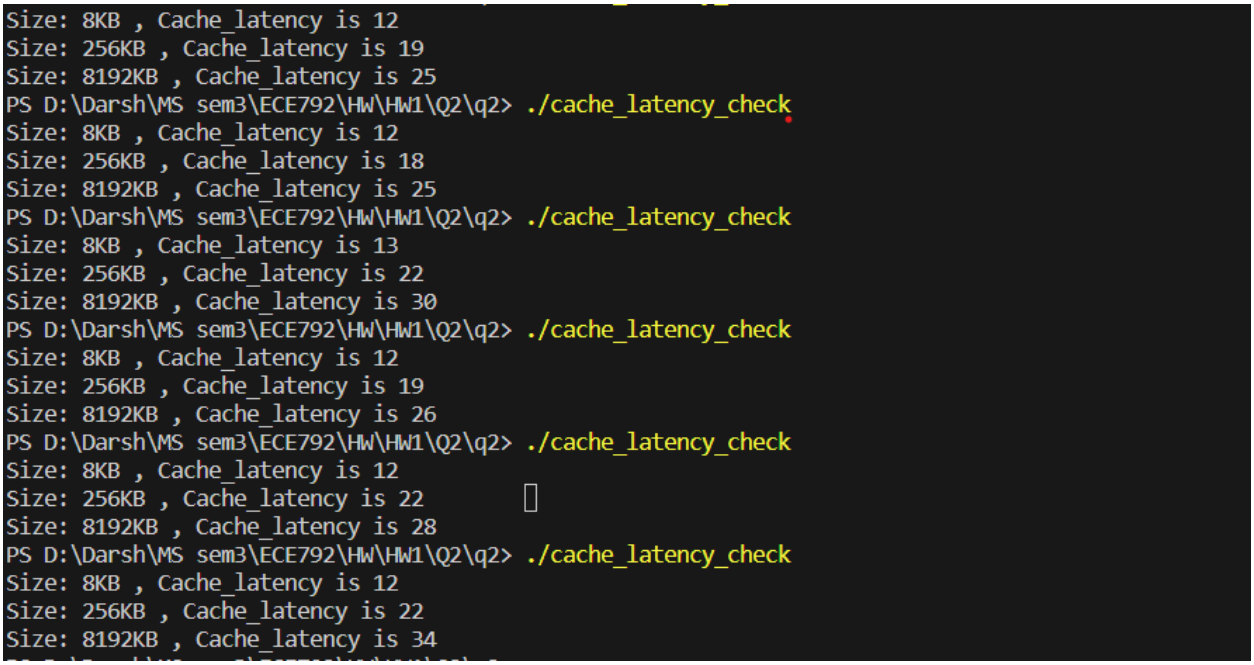
First, I access the 2 Kb of data from Array in order to introduce a Cache miss. I measure the amount of time it takes to execute the whole fragment of code and then I access all the elements of data. I do so to completely fill the L1 Cache. After doing this I begin my process for the L2 cache. I access 2 Kb of data from the array of size 256Kb. As L1 Cache is completely full, Cache has no other option but to store data in L2 Cache. In doing so, I get the cache miss latency in L2 Cache, After this completely load L2 Cache and repeat the same process for L3 Cache.

```
void accessData_2kb(int* array) {
    for(int i =0;i<2048;i++){
        volatile int temp = array[i];
    }
}
```

I am using the above piece of code to access the limited amount of data from the array.

Results :-

After Running the Experiments Multiple number of times, these are the results I get :-




Analysis :-

From the above results we can see that L1 cache miss latency is around 12-13 cycles. L2 Cache Latency seems to be around 18-22 cycles. L3 Cache Latency is bit tricky as it shows a lot of variations from 25-34 cycles. One major reason for this is that L3 cache is shared between multiple cores. So there are multiple processes going outside on in the other parts of the Laptop. Therefore L3 cache is not completely isolated. I estimate the cache latency to be around more than 30 cycles.

Q4)

a) Estimate the Size of Rob in the Processor :-

Methodology:-

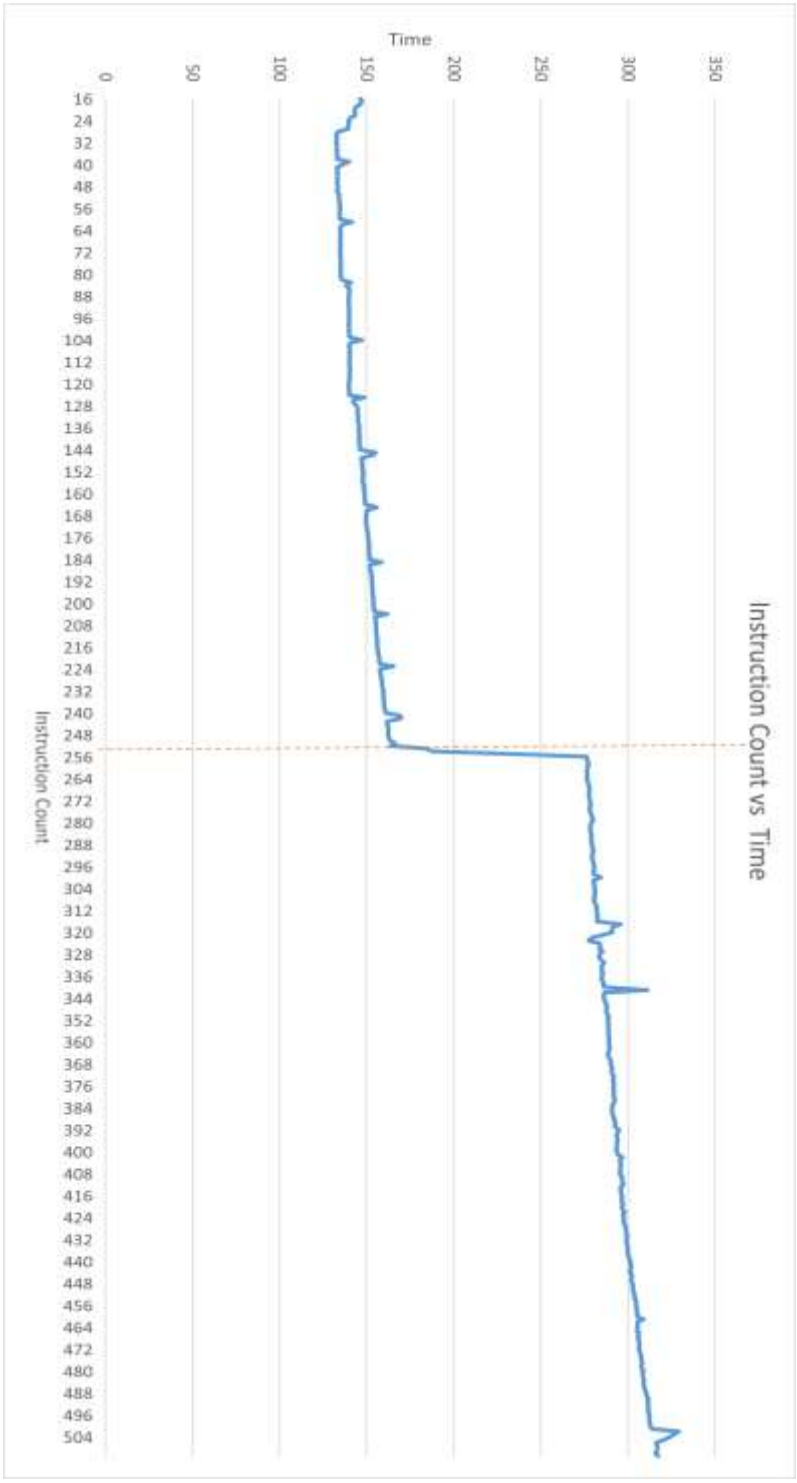
To estimate the ROB size, I have used the microbenchmark created by Henry Wong. He has explained quite wonderfully his thinking behind the measuring the ROB Size.

The instruction window defines the maximum number of instructions that a processor can scan beyond the current unfinished instruction for parallel execution opportunities. This microbenchmark typically uses long latency instruction that can block the instruction commit of the processor. Then we measure the number instructions that can be in this instruction commit.

For this microbenchmark, NOP is used as a filler instruction here. This is used to fill the Reorder Buffer. Here we make sure that after 1$^{st}$ Cache miss, we don't have another Cache miss in the same instruction window. If such things happen then the latency will be handled in parallel and we won't be able to get the size of Reorder Buffer. NOP is also an ideal choice for being used as a filler instruction as it executes swiftly and it has no destination register.

For introducing Long latency instruction, we use MOV. MOV instruction takes around 5 to 11 cycles to execute. This can be a very good source for Long latency instructions, as it will take 5 to 11 cycles more to execute in addition to to the latency introduced by the Pipeline stages.

Results :-


Instruction Count vs Time
Time
0
50
100
150
200
250
300
350
Instruction Count
16
24
32
40
48
56
64
72
80
88
96
104
112
120
128
136
144
152
160
168
176
184
192
200
208
216
224
232
240
248
256
264
272
280
288
296
304
312
320
328
336
344
352
360
368
376
384
392
400
408
416
424
432
440
448
456
464
472
480
488
496
504

Analysis :-

From the above graph we can see that timing increases very significantly as the Instruction count exceed 256. Therefore we can say that size of ROB is 256.

Q7) Check the Existence of Hyperthreading ROB

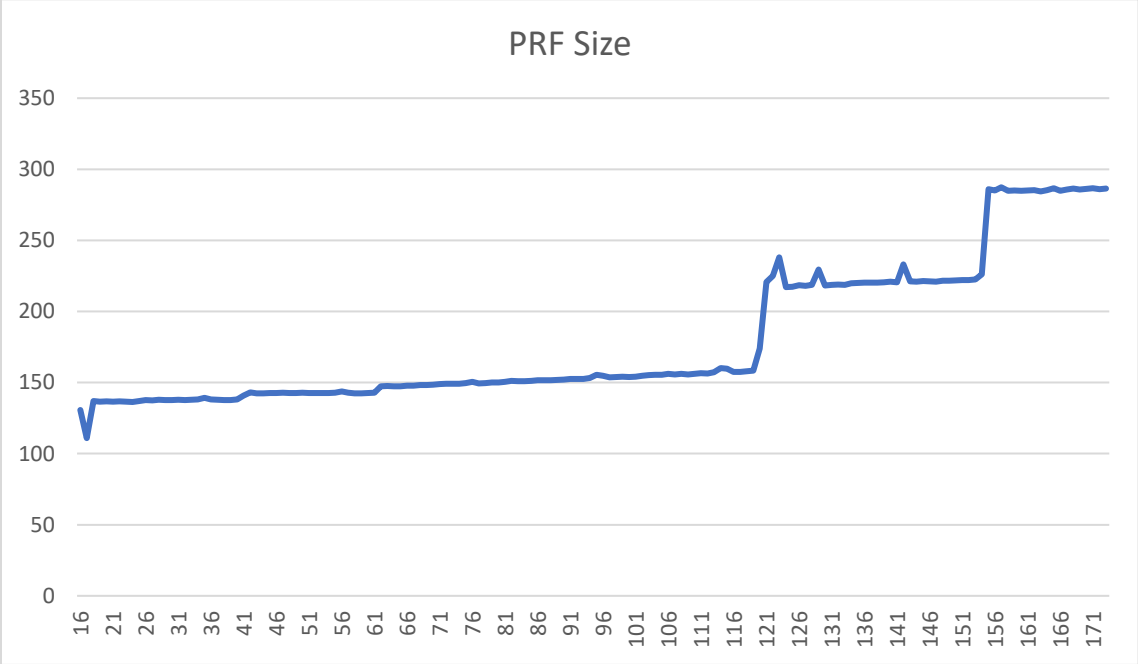
Q4b) Determine the PRF size :-

Methodology :-

This method is very similar to the previous method, but it has one difference. Here ADD is used as a filler instruction as it has a destination register. As a result of this it can be executed Speculatively and also have committed register.

The issue we face with this is that, we can only find the size of speculative registers. I didn't find any method that could help me infer the size of committed registers. One way through which we can find is the number of committed registers, if we know the size of PRF in advance.

Results :-




Analysis :-

Here we see that 2 places where we see there is a significant rise in timing. I would estimate PRF size to be around 121. But this number is only for the speculative registers. So the real number might be a lot higher than the estimated number.

Here the 2 places with significant rise in timing suggest the existence of Hyper Threading in the ROB and PRF. I think we can generate the similar graph for the ROB if increase the number of instruction count. PRF can produce similar results for less number of instruction counts.

Q5)

a) Determine the size of TLB :-

Methodology :-

The Translation Lookaside Buffer (TLB) is a CPU cache used to speed up virtual-physical address translation. The page size used in a system plays a significant role in the efficiency of the TLB, which in turn impacts the overall system performance.

Larger page sizes can be beneficial as fewer entries would be required in the TLB to reference the same amount of memory due to reduced fragmentation. This could potentially slow down the address translation process but might result in fewer TLB misses, because more memory can be addressed by each entry in the TLB.

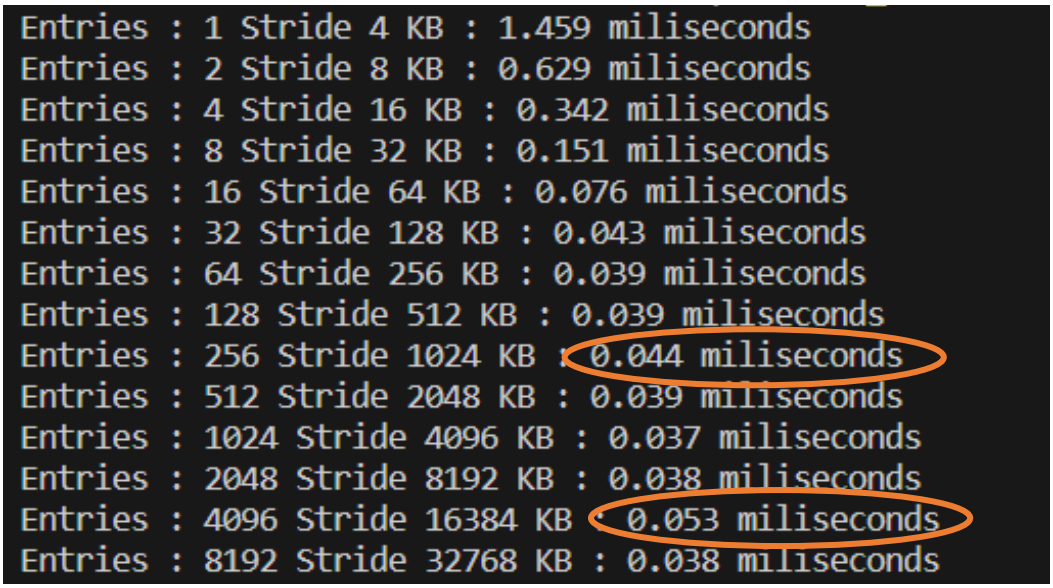
Conversely, smaller page sizes would lead to more entries in the TLB for the same amount of memory space, which might speed up translation time but can increase the likelihood of TLB misses. This is because a TLB of a given size can hold references to fewer total bytes in memory when pages are small. Also, small pages lead to more overhead because of the increased number of page table entries.

The idea to calculate the size of TLB is very similar to that of finding the Cache Line size. To make this happen, we make sure to take strides in terms of the page size instead of string for array elements.

From the specification available online, we know that the page size in the processor is around 4Kb. Therefore, we multiply the page size with strides which we keep on taking. When the stride is equal to size of TLB, our access time increases.

Here we take a very large array size of 1GB and keep the minimum iterations to be run around the 8192. This number is chosen because when we traverse through the TLB and access the elements of the arrays, we are making sure that those number of accesses are larger than the TLB entries.

Results :-




Analysis :-

From the time results which I get above, I can conclude that there are 2 TLBs in my Processor. According to my analysis, size of 1$^{st}$ level TLB is 128 entries as we see the rise in timing after it and 2$^{nd}$ level of TLB is 2048 entries.

b) Determine the size of the TLB Miss Latency

Methodology :-

My idea to measure the TLB Miss penalty, is to measure the TLB hit time and TLB miss time and take the difference between them. TLB penalties are among the largest penalties in the CPU latencies.

To measure the TLB hit time, I access an array which is smaller than the TLB size. Then we measure the time it takes us to access the few iterations of TLB hits, which are less than the entries of the TLB size.

To measure the TLB miss time, I access an array which is very large. The number access in this TLB are larger than the TLB entries. Due to this there is a large latency in created by the TLB miss.

Here we also make sure to divide the time by the number of accesses which we are making, to get the value of single TLB miss.

Results :-

```
 TLB Hit time is 89
 TLB miss time is 357
 TLB miss latency is 268
PS D:\Darsh\MS sem3\ECE792\HW\HW1\Q5\qb> ./TLB_miss_latency
 TLB Hit time is 43
 TLB miss time is 409
 TLB miss latency is 366
PS D:\Darsh\MS sem3\ECE792\HW\HW1\Q5\qb> ./TLB_miss_latency
 TLB Hit time is 70
 TLB miss time is 360
 TLB miss latency is 290
PS D:\Darsh\MS sem3\ECE792\HW\HW1\Q5\qb> ./TLB_miss_latency
 TLB Hit time is 85
 TLB miss time is 358
 TLB miss latency is 273
PS D:\Darsh\MS sem3\ECE792\HW\HW1\Q5\qb> ./TLB_miss_latency
 TLB Hit time is 63
 TLB miss time is 365
 TLB miss latency is 302
PS D:\Darsh\MS sem3\ECE792\HW\HW1\Q5\qb> ./TLB_miss_latency
 TLB Hit time is 86
 TLB miss time is 371
 TLB miss latency is 285
PS D:\Darsh\MS sem3\ECE792\HW\HW1\Q5\qb> ./TLB_miss_latency
 TLB Hit time is 88
 TLB miss time is 362
 TLB miss latency is 274
PS D:\Darsh\MS sem3\ECE792\HW\HW1\Q5\qb> ./TLB_miss_latency
 TLB Hit time is 76
 TLB miss time is 357
 TLB miss latency is 281
PS D:\Darsh\MS sem3\ECE792\HW\HW1\Q5\qb>
```

Analysis :-

I ran this code multiple times, and found the cost of TLB miss is the range of 275-366 cycles. I estimate the TLB miss penalty to be 285, as it seems to have been repeated multiple times in the course of the experiment. Excess timing can be explained by the latencies involved in searching for pages.