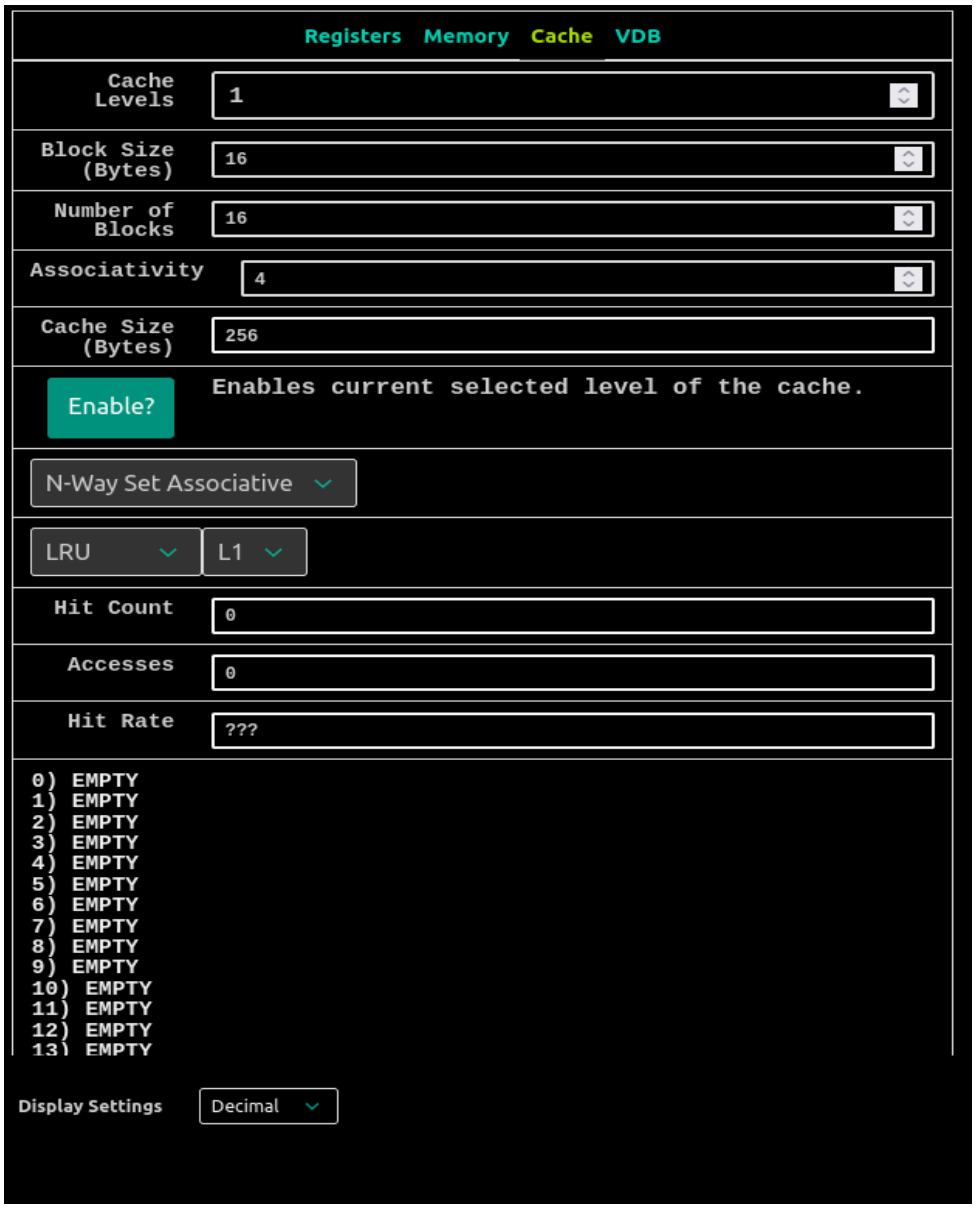
Task 2: Memory Accesses

1. Venus setup for Scenario 2 showing Program parameters, Cache parameters etc

The programming and cache parameters are shown below

```
22 ##################################################################################################
23 # You MAY change the code below this section
24 main:   li  a0, 256     # array size in BYTES (power of 2 < array size)
25     li  a1, 2       # step size  (power of 2 > 0)
26     li  a2, 1       # rep count  (int > 0)
27     li  a3, 1       # 0 - option 0, 1 - option 1
28 # You MAY change the code above this section
29 ##################################################################################################
```



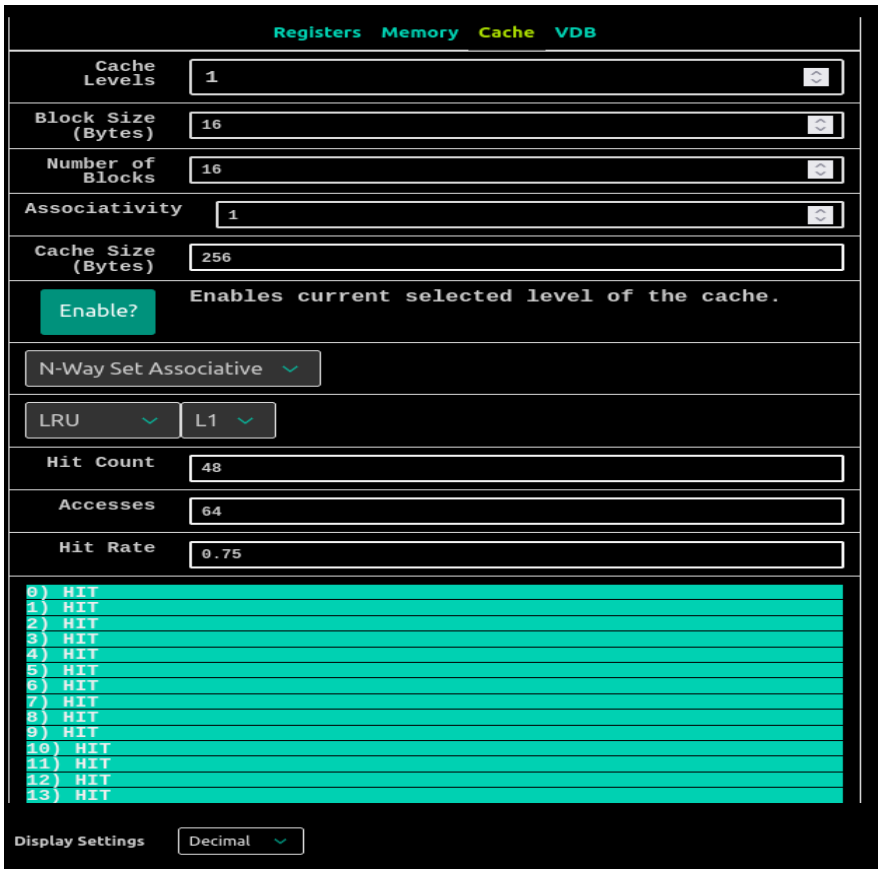
Tasks

1. How many memory accesses are there per iteration of the inner loop (not the one involving Rep Count)?

There would be total of **2 memory accesses** per iteration of the inner loop. Since the **option 1** is selected and in the first iteration of the inner loop (not the one involving rep count) the first **read access** will have a **cache miss** and so that **one memory access** would be there and after that there would be **cache hit** but since the cache we are using is write through, write allocate cache so the data written to cache is written simultaneously written to memory as well so there would be **one memory** access here also so total of two memory access will be there

2. What is the repeating hit/miss pattern? Write your answer in the form "mmhhmh" and so on, where your response is the shortest pattern that gets repeated.

The repeating hit/miss pattern is "**mhhh"** and with this the hit rate is 75% as sshwon in the figure below, which will be repeated **16 times** since the array size is of 256B/4B = **64 words,** But the step size is of 2 so there would be **32 words** accesses, and each block is of **4 words** for and for each way (say N = 0) has 16 words for each **4words** the pattern is "mhhh" which will be repeated **16 times.**



3. Keeping everything else the same, what does our hit rate approach as Rep Count goes to infinity? Try it out by changing the appropriate program parameter and letting the code run! Write your answer as a decimal.

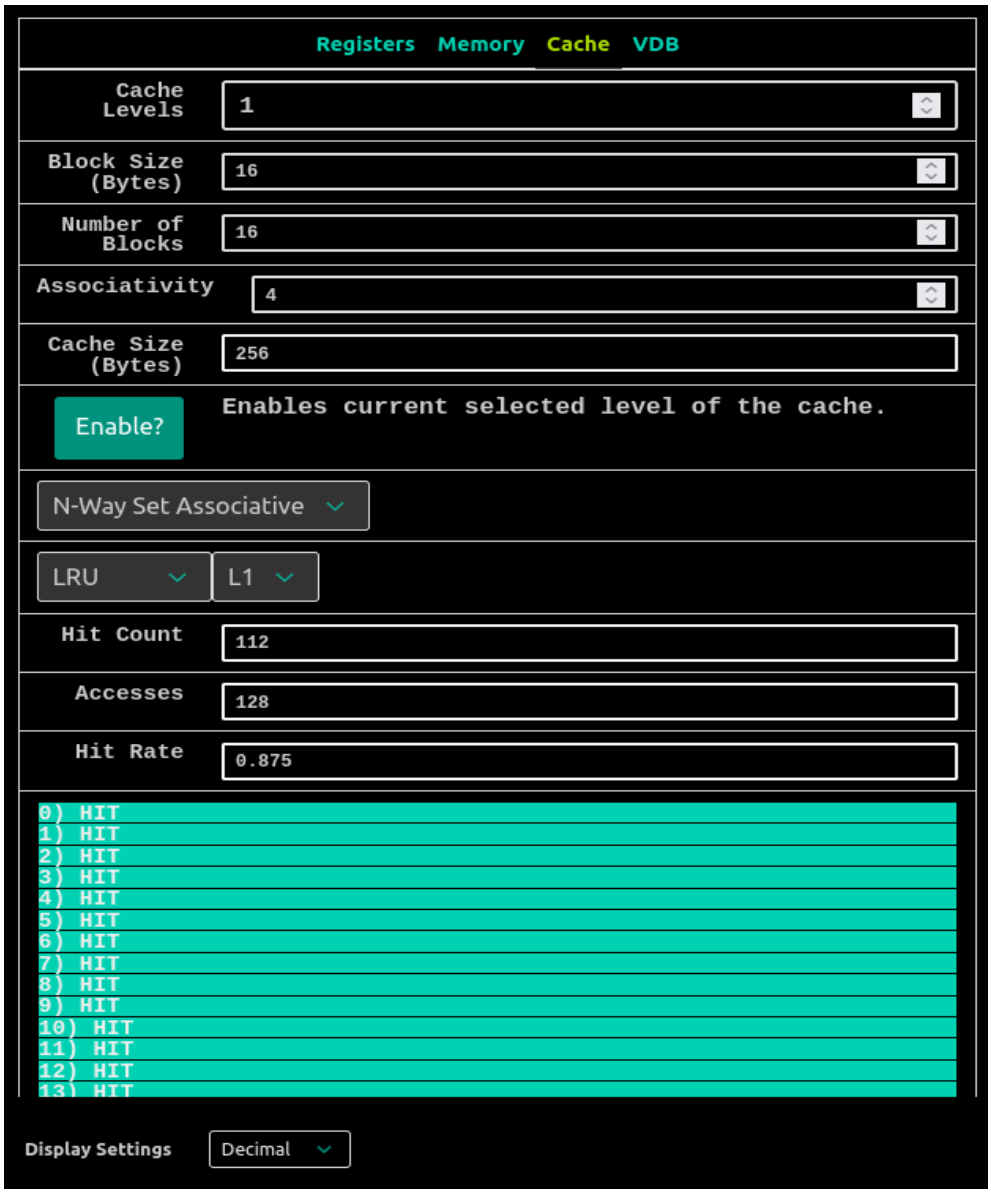
As rep count approaches to infinity the hit rate approaches to 1 (100 %) , for example if we increase the rep count to 2 instead of 1 the hit rate would be greater than 75% which is calculated as for the **first rep count** there would be **16 misses** and in the next **rep count** those 16 misses would not occur and the overall hit rate would be

Hit rate = No of hits / No of accesses

for **rep count** = 2 we have

Hit rate = (128-16) / 128 = 0.875 = **87.5**%

The same hit rate was found in the VENUS simulator when the rep count was set to 2, as shown in snapshot below



4. Suppose we have a program that iterates through a very large array (i.e. way bigger than the size of the cache) Rep Count times. During each Rep, we map a different function to the elements of our array (e.g. if Rep Count = 1024, we map 1024 different functions onto each of the array elements, one per Rep). For reference, in this scenario, we just had one function (incrementation) and one Rep.

> **HINT: You do not want to iterate through the entire array at once because it's much bigger than your cache. Doing so would reduce the amount of temporal locality your program exhibits, which makes the cache hit rate suffer. We want to exhibit more locality so that our caches can take advantage of our predictable behavior. So, instead, we should try to access ____ of the array at a time and apply all of the ____ to that ____ so we can be completely done with it before moving on, thereby keeping that ____ hot in the cache and not having to circle back to it later on! (The 1st, 3rd, and 4th blanks should be the same. It's not some vocabulary term you should use to fill them in. It's more of an idea that you should have.)**

The answer of the blank is **cache-sized portion** which is explained as In this situation, it is advisable to **cache-sized portion o**f the array that is equivalent in size to the cache and perform all the required operations on that portion. By breaking down the array into smaller segments, such as cache lines or cache blocks, we can guarantee that the necessary data for each iteration is present in the cache. This approach minimizes cache misses and enhances the cache hit rate, leading to improved overall performance.