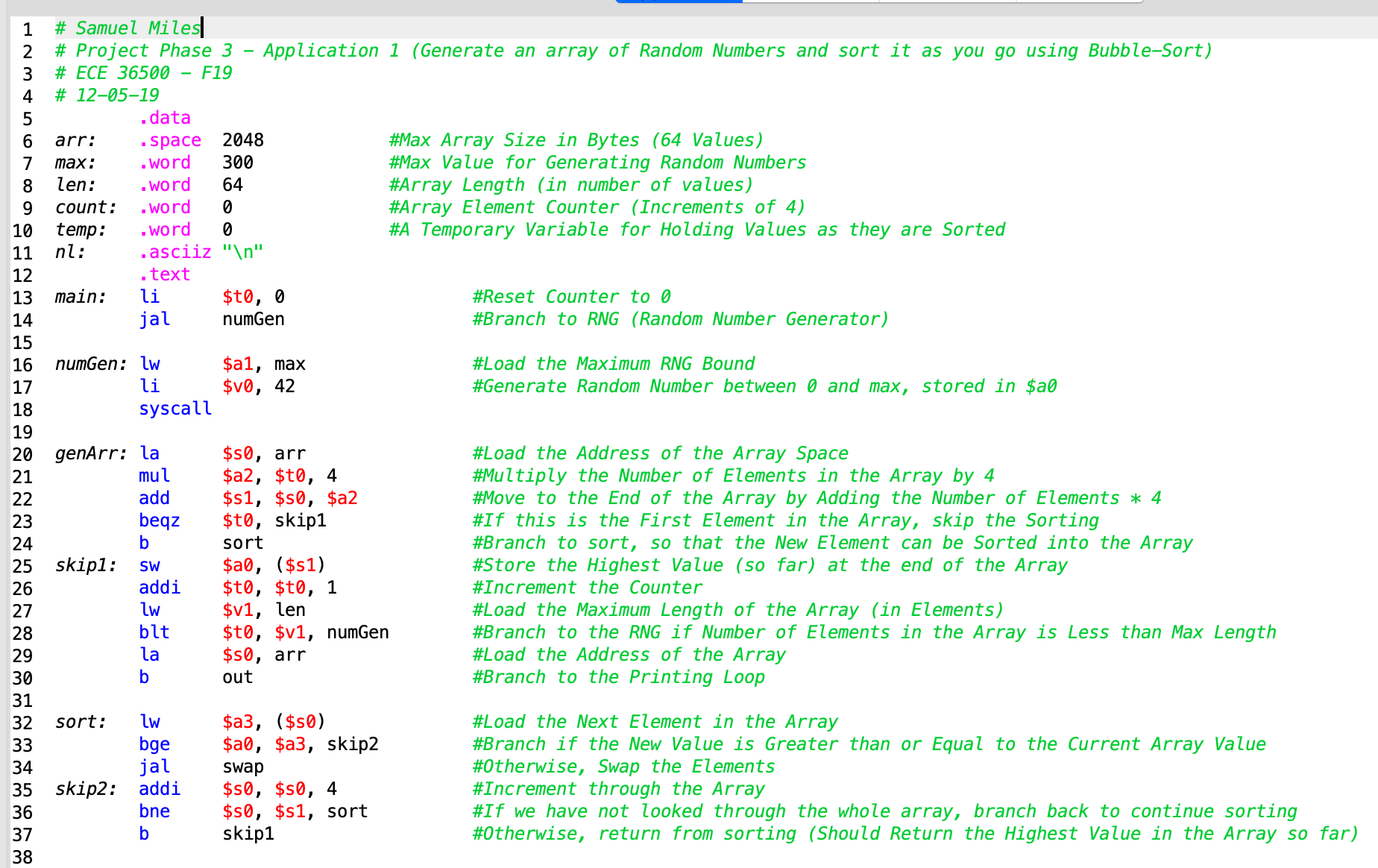
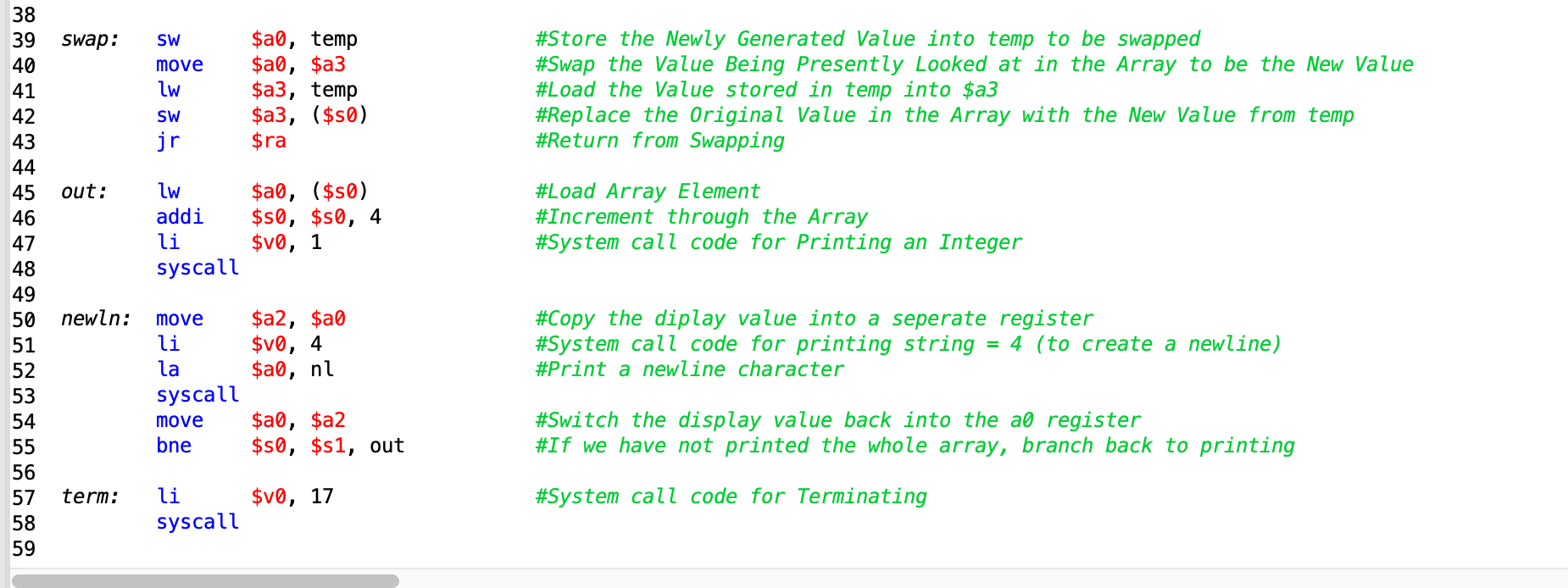
**Project Phase 3:**

**Application 1:** - Random Integer Array Generator and Sorting

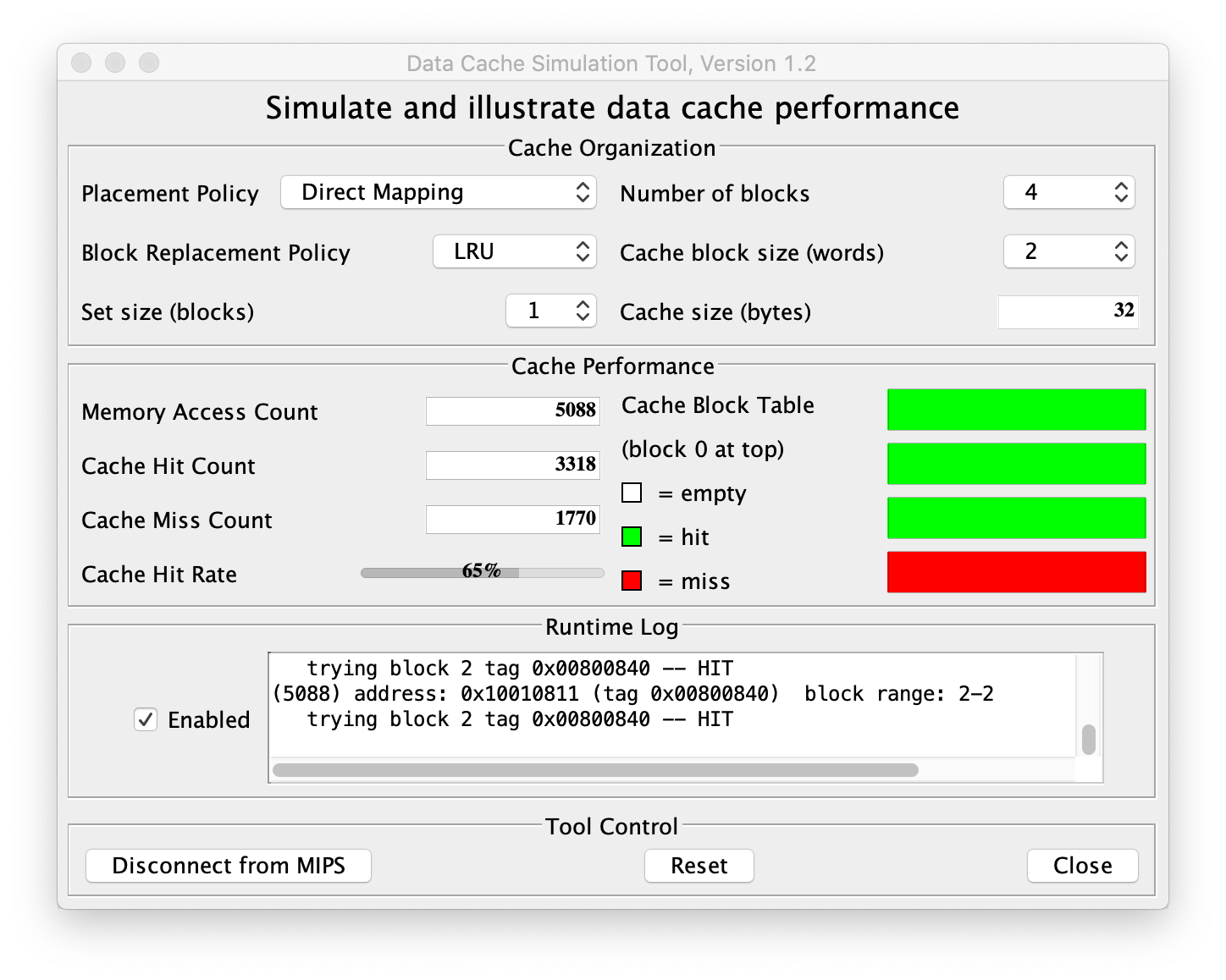
The first application generates 64 random integer values between 0 and 300 to be stored in an array. As each value is sequentially generated, the new entry is sorted using the Bubble Sort method by comparing the new entry to each value that is already in the array. Finally, after the array has been fully generated and sorted, each element of the array is printed sequentially, each value on a new line of the console. The maximum size of the array is defined in the data section as a space of 2048 Bytes. When choosing a maximum bound for the random values in the array, 300 was chosen over something smaller to attempt to minimize the number of repeated values that appear in the array. The first application utilizes a smaller cache size with a 4 blocks and cache block size of 2 words per block. Included is both a screenshot of the code for ease of reading, as well as a text version of the code for easy evaluation and testing. Additionally, the data segment and the program output are included for each caching method.

**Code:**



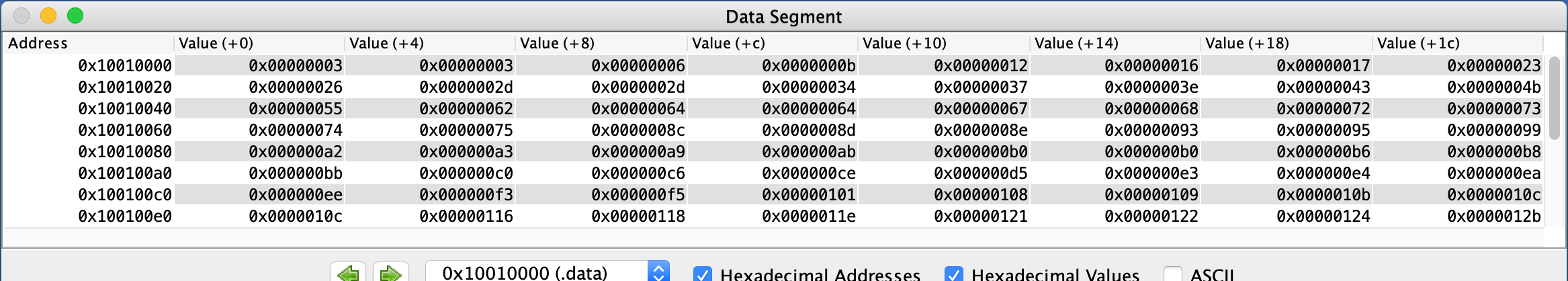
|  |
| --- |
| # Samuel Miles  # Project Phase 3 - Application 1 (Generate an array of Random Numbers and sort it as you go using Bubble-Sort)  # ECE 36500 - F19  # 12-05-19  .data  arr: .space 2048 #Max Array Size in Bytes (64 Values)  max: .word 300 #Max Value for Generating Random Numbers  len: .word 64 #Array Length (in number of values)  count: .word 0 #Array Element Counter (Increments of 4)  temp: .word 0 #A Temporary Variable for Holding Values as they are Sorted  nl: .asciiz "\n"  .text  main: li $t0, 0 #Reset Counter to 0  jal numGen #Branch to RNG (Random Number Generator)    numGen: lw $a1, max #Load the Maximum RNG Bound  li $v0, 42 #Generate Random Number between 0 and max, stored in $a0  syscall  genArr: la $s0, arr #Load the Address of the Array Space  mul $a2, $t0, 4 #Multiply the Number of Elements in the Array by 4  add $s1, $s0, $a2 #Move to the End of the Array by Adding the Number of Elements \* 4  beqz $t0, skip1 #If this is the First Element in the Array, skip the Sorting  b sort #Branch to sort, so that the New Element can be Sorted into the Array  skip1: sw $a0, ($s1) #Store the Highest Value (so far) at the end of the Array  addi $t0, $t0, 1 #Increment the Counter  lw $v1, len #Load the Maximum Length of the Array (in Elements)  blt $t0, $v1, numGen #Branch to the RNG if Number of Elements in the Array is Less than Max Length  la $s0, arr #Load the Address of the Array  b out #Branch to the Printing Loop    sort: lw $a3, ($s0) #Load the Next Element in the Array  bge $a0, $a3, skip2 #Branch if the New Value is Greater than or Equal to the Current Array Value  jal swap #Otherwise, Swap the Elements  skip2: addi $s0, $s0, 4 #Increment through the Array  bne $s0, $s1, sort #If we have not looked through the whole array, branch back to continue sorting  b skip1 #Otherwise, return from sorting (Should Return the Highest Value in the Array so far)    swap: sw $a0, temp #Store the Newly Generated Value into temp to be swapped  move $a0, $a3 #Swap the Value Being Presently Looked at in the Array to be the New Value  lw $a3, temp #Load the Value stored in temp into $a3  sw $a3, ($s0) #Replace the Original Value in the Array with the New Value from temp  jr $ra #Return from Swapping    out: lw $a0, ($s0) #Load Array Element  addi $s0, $s0, 4 #Increment through the Array  li $v0, 1 #System call code for Printing an Integer  syscall    newln: move $a2, $a0 #Copy the diplay value into a seperate register  li $v0, 4 #System call code for printing string = 4 (to create a newline)  la $a0, nl #Print a newline character  syscall  move $a0, $a2 #Switch the display value back into the a0 register  bne $s0, $s1, out #If we have not printed the whole array, branch back to printing    term: li $v0, 17 #System call code for Terminating  syscall |

**Direct Mapping Cache:**

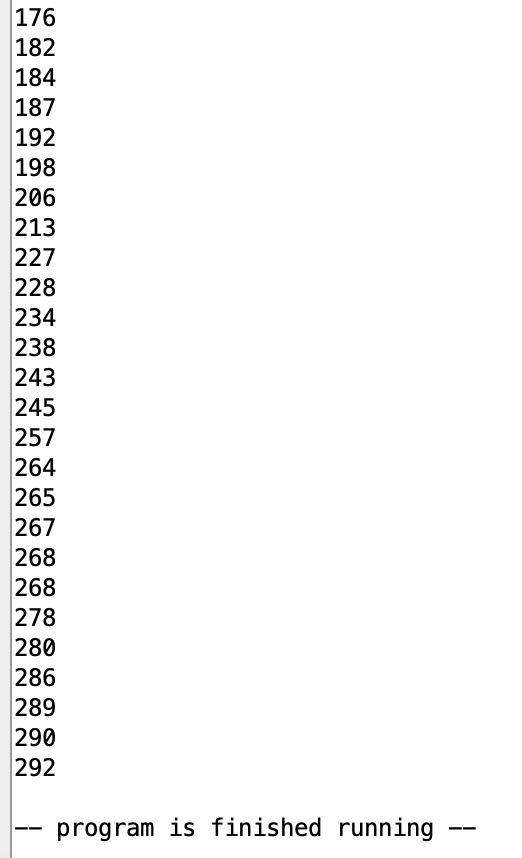
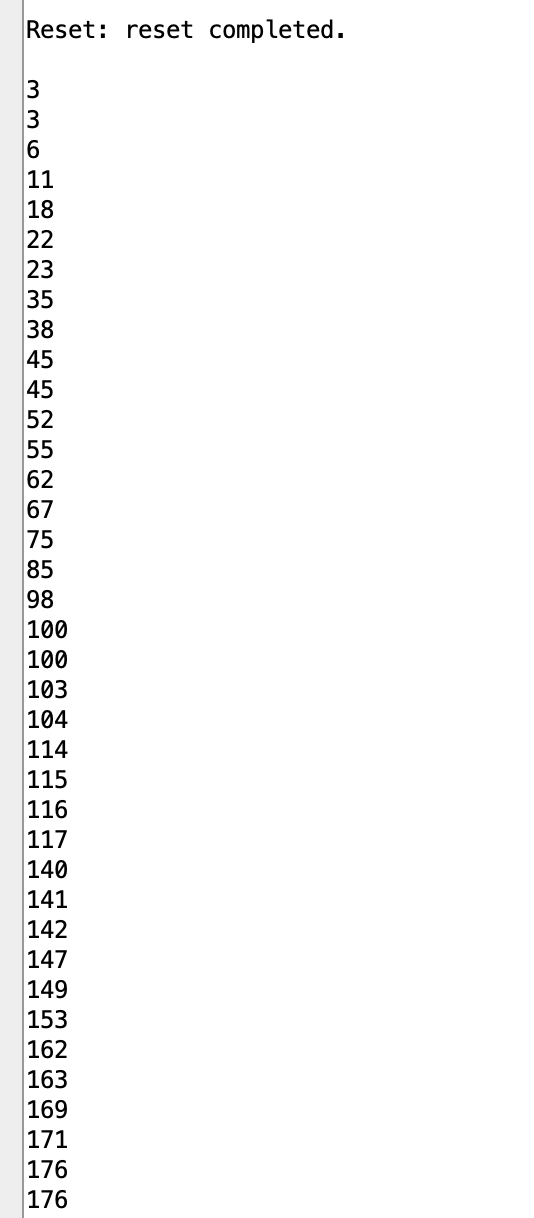


|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Number of Blocks: | Cache Block Size: | Memory Access Count: | | Cache Hit Count: | Cache Miss Count: | Cache Hit Ratio: |
| 4 | 2 | 5088 | 3318 | | 1770 | 65% |

**Data Segment:**

****

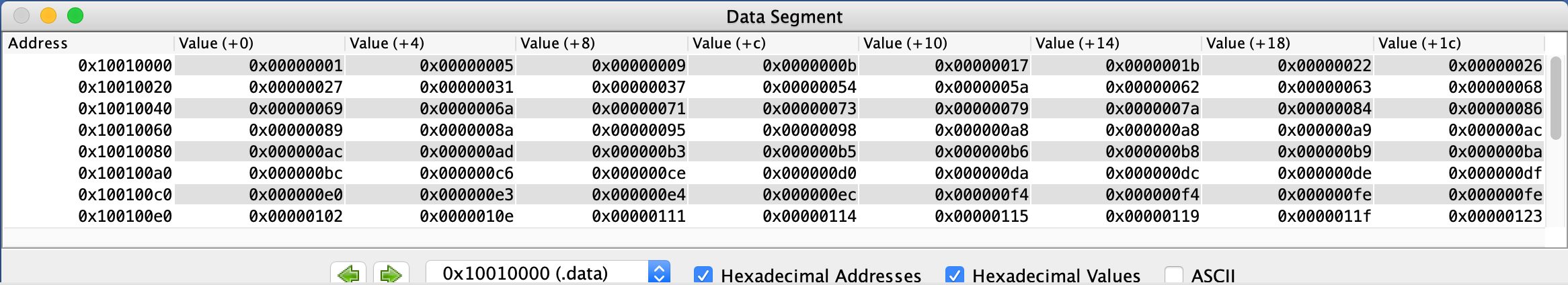
**Output:**



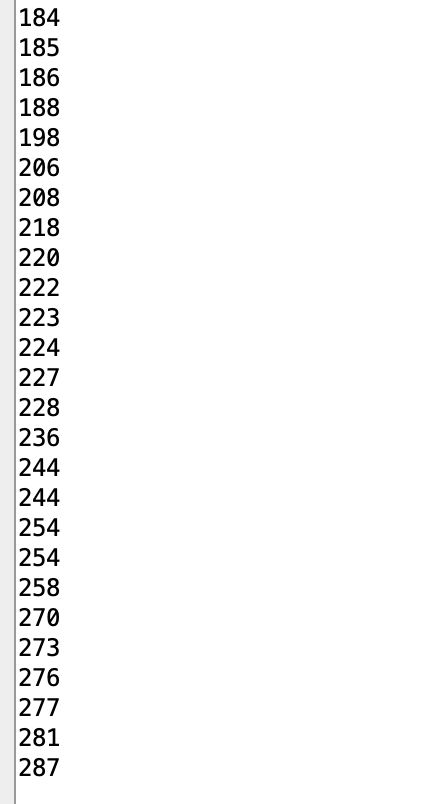
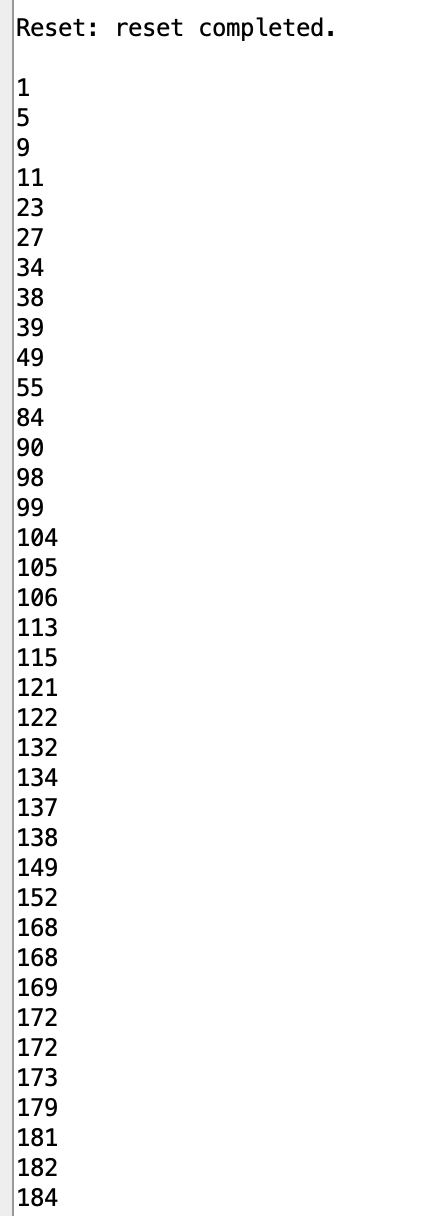
**Fully Associative Cache:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Number of Blocks: | Cache Block Size: | Memory Access Count: | | Cache Hit Count: | Cache Miss Count: | Cache Hit Ratio: |
| 4 | 2 | 5296 | 4096 | | 1196 | 77% |

**Data Segment:**



**Output:**

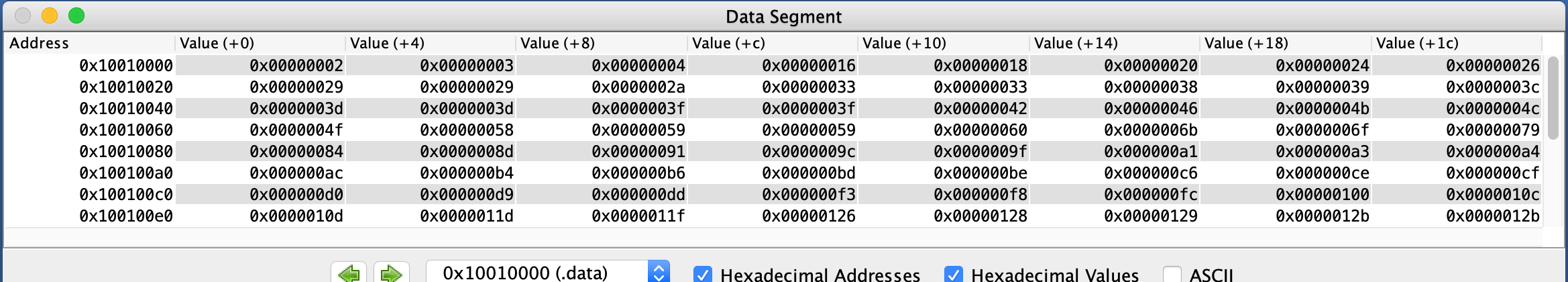


**N-way Set Associative Cache:**

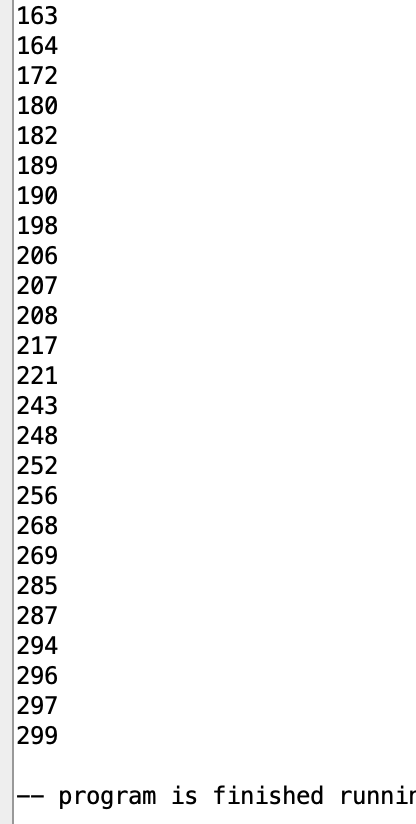
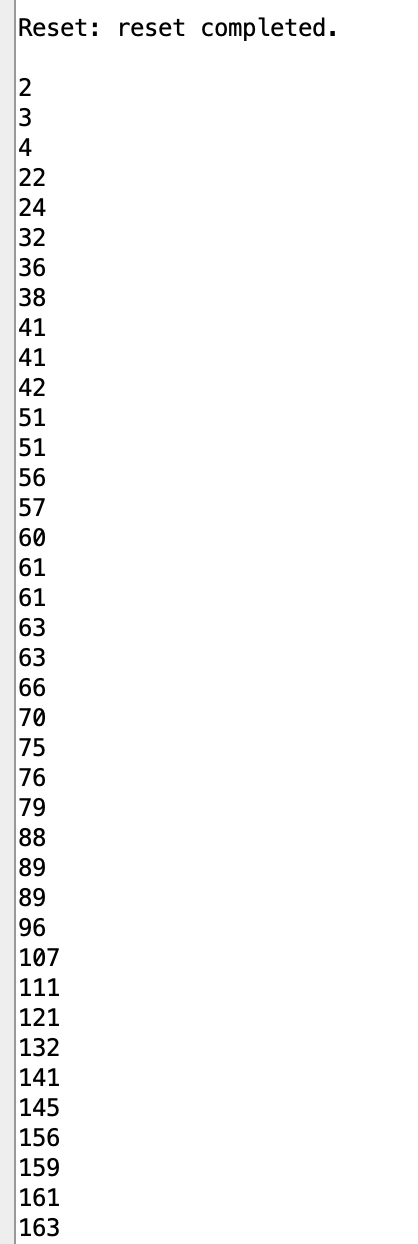


|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Number of Blocks: | Cache Block Size: | Memory Access Count: | | Cache Hit Count: | Cache Miss Count: | Cache Hit Ratio: |
| 4 | 2 | 5427 | 3589 | | 1838 | 66% |

**Data Segment:**



**Output:**



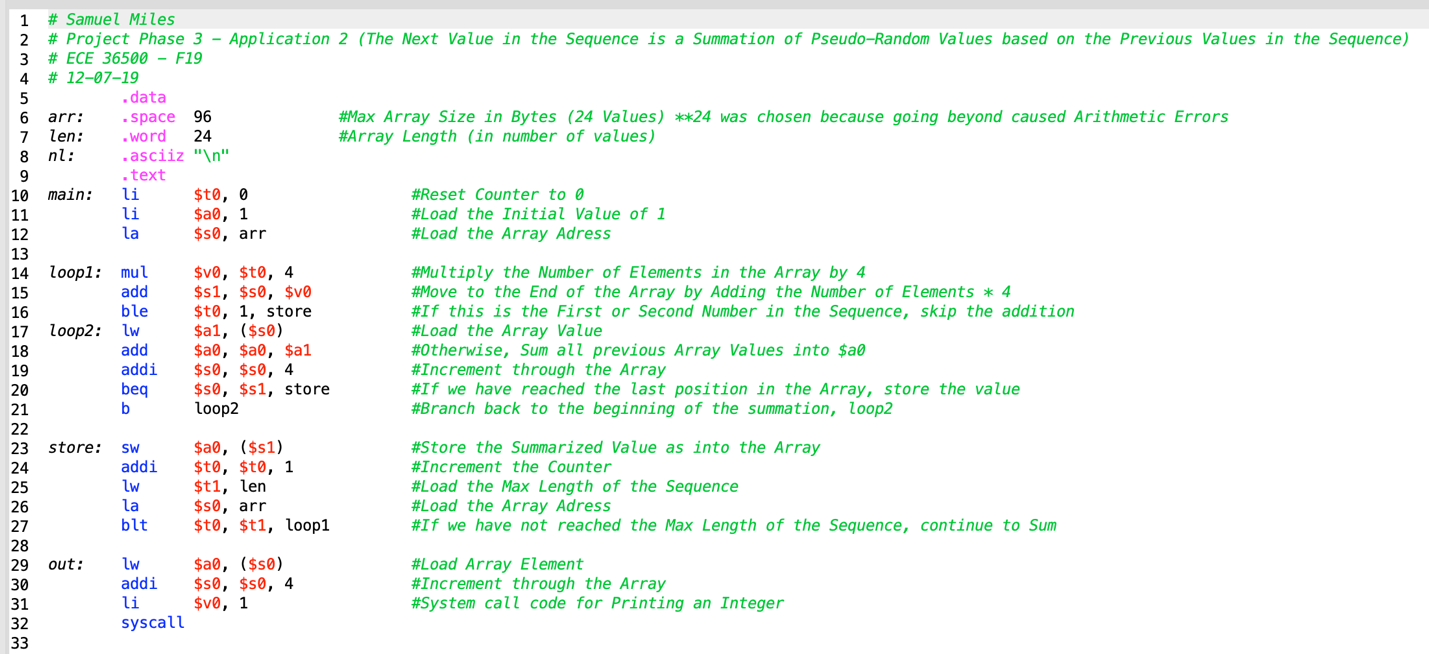
**Conclusion:**

This first application is meant to make it easier to visualize the way that the different types of cache mapping affect the performance of the program. Due to the nature of the random number generator, it is possible for the array to have duplicate values. While this is mostly mitigated by utilizing an upper bound for the numbers generated of 300, there are still some duplicates. As such, a smaller cache size was chosen to make the performance benefits of different cache maps more obvious. It can be observed that in application 1 the Direct Mapping Cache and the N-way Set Associative Cache yield nearly the same hit rate at 65% and 66% respectively, where-as the Fully Associative Cache yielded the best performance with a hit rate of 77%. This is likely due to the much more flexible nature of a Fully Associative Cache that allows for a memory block to be placed in any available cache block. Compared to the Direct Mapping Cache and the N-way Set Associative Cache that are not able to take advantage of the same level of flexibility.

**Application 2:** - Array Summation of Pseudo-Random Values

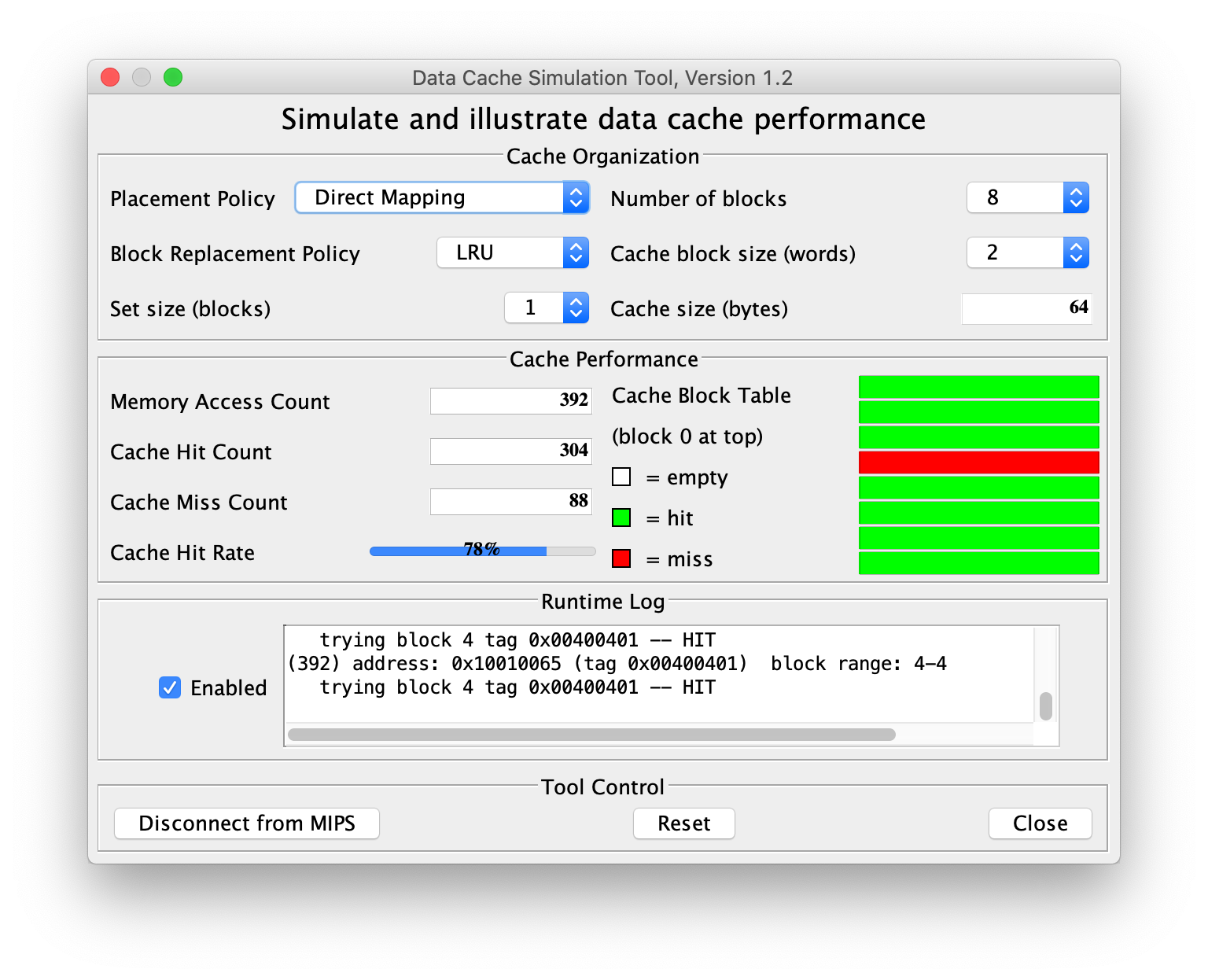
The second application works similarly to a Fibonacci sequence, but with a few small differences. An array is generated where the value of each entry of the array is equal to the sum of each of the previous values of the array with a small modification that adds a small amount of randomness to each of the values based on the number of times that the program loops to add the values together. The initial value of the array is 1, and the maximum size of the array is limited to 96 Bytes, or 24 values. The reason for this limitation is because an arithmetic error is encountered when the program goes beyond 24 values in the array. This is likely due to the exponential growth of the values in the array as larger and larger values are calculated, until the mathematical limit of the simulator is reached. The size of 8 cache blocks with a block size of 2 words per block were chosen because it was found that these values best demonstrate the performance differences between each of the cache types. Included is both a screenshot of the code for ease of reading, as well as a text version of the code for easy evaluation and testing. Additionally, the data segment and the program output are included for the program, these values are the same for each type of cache and thus are only listed once.

**Code:**



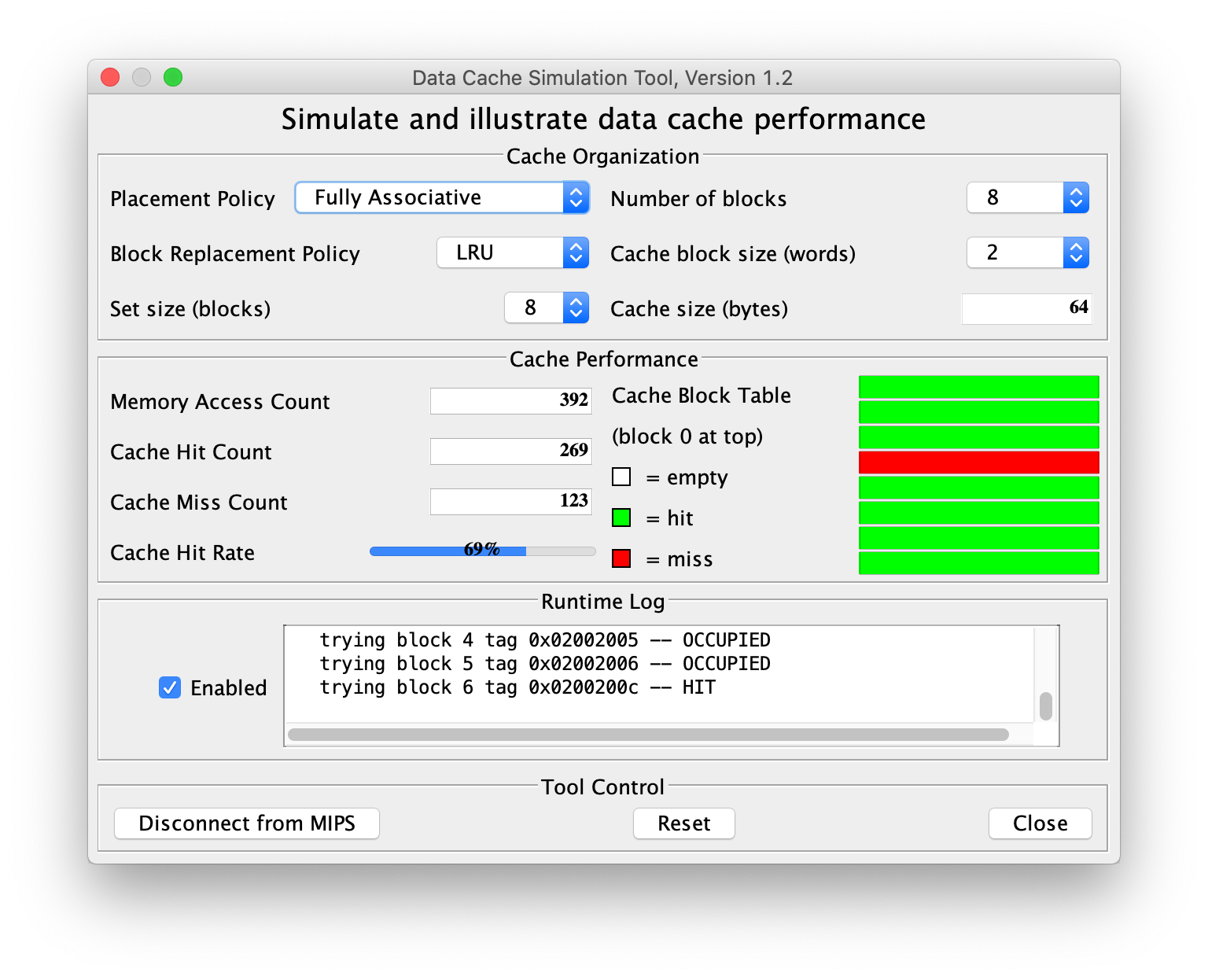
|  |
| --- |
| # Samuel Miles  # Project Phase 3 - Application 2 (The Next Value in the Sequence is a Summation of Pseudo-Random Values based on the Previous Values in the Sequence)  # ECE 36500 - F19  # 12-07-19  .data  arr: .space 96 #Max Array Size in Bytes (24 Values) \*\*24 was chosen because going beyond caused Arithmetic Errors  len: .word 24 #Array Length (in number of values)  nl: .asciiz "\n"  .text  main: li $t0, 0 #Reset Counter to 0  li $a0, 1 #Load the Initial Value of 1  la $s0, arr #Load the Array Adress  loop1: mul $v0, $t0, 4 #Multiply the Number of Elements in the Array by 4  add $s1, $s0, $v0 #Move to the End of the Array by Adding the Number of Elements \* 4  ble $t0, 1, store #If this is the First or Second Number in the Sequence, skip the addition  loop2: lw $a1, ($s0) #Load the Array Value  add $a0, $a0, $a1 #Otherwise, Sum all previous Array Values into $a0  addi $s0, $s0, 4 #Increment through the Array  beq $s0, $s1, store #If we have reached the last position in the Array, store the value  b loop2 #Branch back to the beginning of the summation, loop2    store: sw $a0, ($s1) #Store the Summarized Value as into the Array  addi $t0, $t0, 1 #Increment the Counter  lw $t1, len #Load the Max Length of the Sequence  la $s0, arr #Load the Array Adress  blt $t0, $t1, loop1 #If we have not reached the Max Length of the Sequence, continue to Sum    out: lw $a0, ($s0) #Load Array Element  addi $s0, $s0, 4 #Increment through the Array  li $v0, 1 #System call code for Printing an Integer  syscall    newln: move $a2, $a0 #Copy the diplay value into a seperate register  li $v0, 4 #System call code for printing string = 4 (to create a newline)  la $a0, nl #Print a newline character  syscall  move $a0, $a2 #Switch the display value back into the a0 register  bne $s0, $s1, out #If we have not printed the whole array, branch back to printing    term: li $v0, 17 #System call code for Terminating  syscall |

**Direct Mapping Cache:**



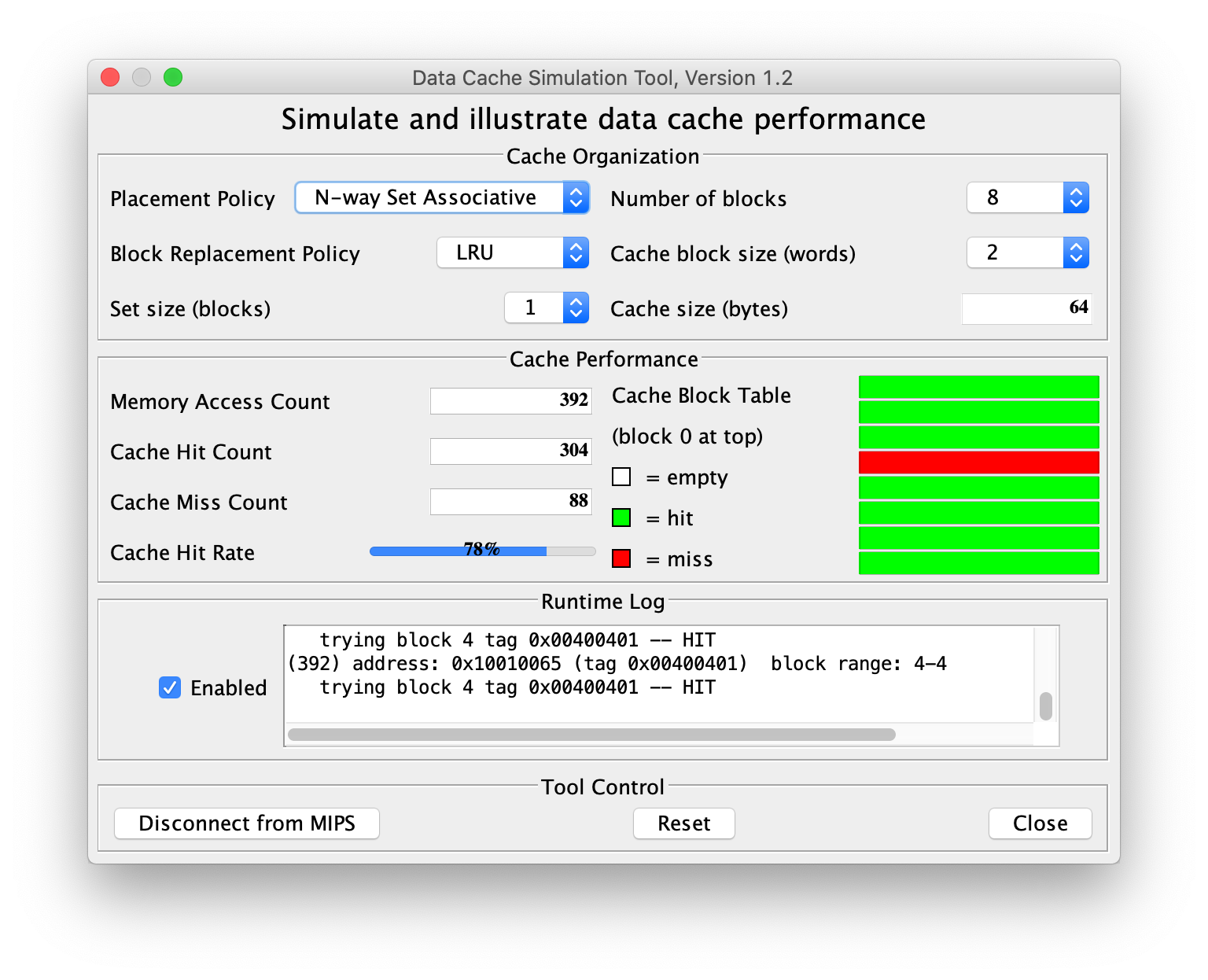
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Number of Blocks: | Cache Block Size: | Memory Access Count: | | Cache Hit Count: | Cache Miss Count: | Cache Hit Ratio: |
| 8 | 2 | 392 | 304 | | 88 | 78% |

**Fully Associative Cache:**



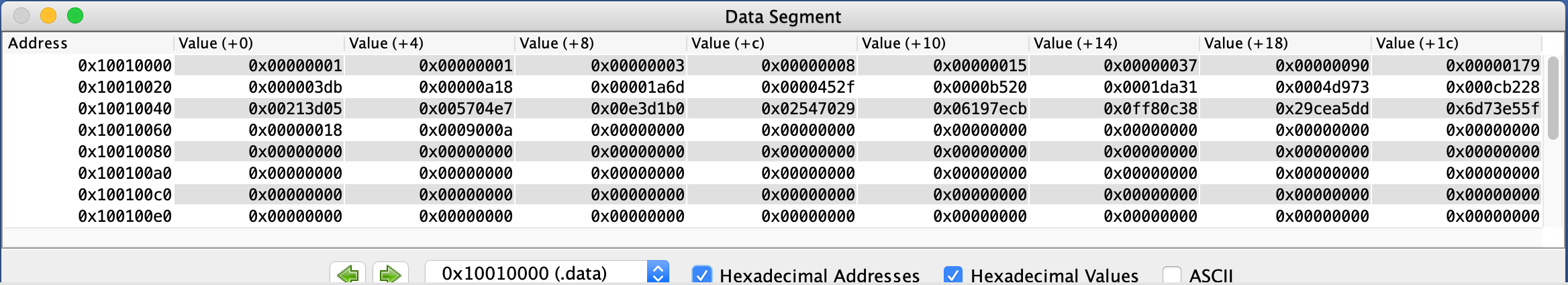
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Number of Blocks: | Cache Block Size: | Memory Access Count: | | Cache Hit Count: | Cache Miss Count: | Cache Hit Ratio: |
| 8 | 2 | 392 | 269 | | 123 | 69% |

**N-way Set Associative Cache:**

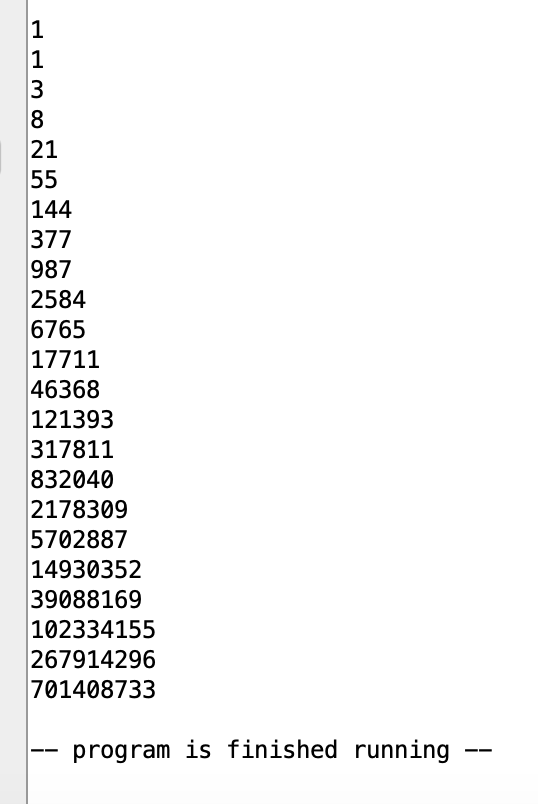


|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Number of Blocks: | Cache Block Size: | Memory Access Count: | | Cache Hit Count: | Cache Miss Count: | Cache Hit Ratio: |
| 8 | 2 | 392 | 304 | | 88 | 78% |

**Data Segment:**



**Output:**



**Conclusion:**

The second application shows the opposite of the conclusions made in the first application. While the Direct Mapping Cache and the N-way Set Associative Cache still yield the same or very similar values for the hit rate, with this particular instance having both at 78%, the Fully Associative Cache does not show any improvement whatsoever when compared to the other caching techniques. In fact, the Fully Associative Cache shows a reduced performance with a hit rate of just 69%. This trend was consistent even when the cache size was varied, however the choice of 8 blocks with 2 words per block was chosen because the performance deficit of the Fully Associative Cache is much more noticeable at this scale. It appears that in this case, the additional flexibility to place a memory block into any cache block is actually acting as more of a hinderance to the performance of the program than a benefit. This observation is the exact opposite of what was observed in the conclusions drawn from the first application. This may be due in part to the LRU replacement method because with full access to write a memory block to any cache block, the program may have overwritten data that it did not know it was going to need in down the line, and as a result was not able to produce the same level of performance gain when compared to the Direct Mapping Cache or the N-way Set Associative Cache. This program demonstrates how additional flexibility does not always provide the greatest performance gain, but rather having a more structured or restricted method for replacing values in a cache can lead to better results.