Samuel Miles
12/7/2019
ECE 36500

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:

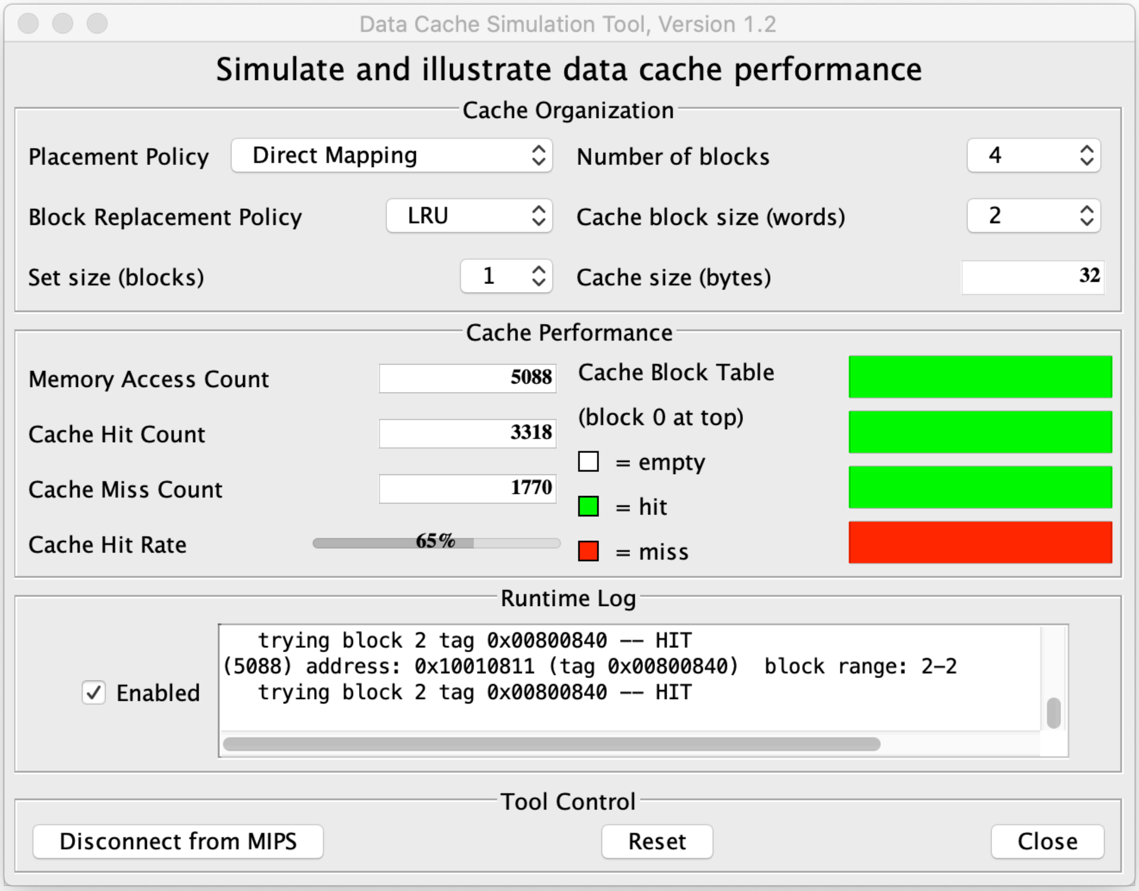
```
# Samuel Miles
# Project Phase 3 - Application 1 (Generate an array of Random Numbers and sort it as you go using Bubble-Sort)
# ECE 36500 - F19
# 12-05-19
         .data
arr:     .space  2048          #Max Array Size in Bytes (64 Values)
max:     .word   300           #Max Value for Generating Random 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
```

Samuel Miles
12/7/2019
ECE 36500

```
# 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
```

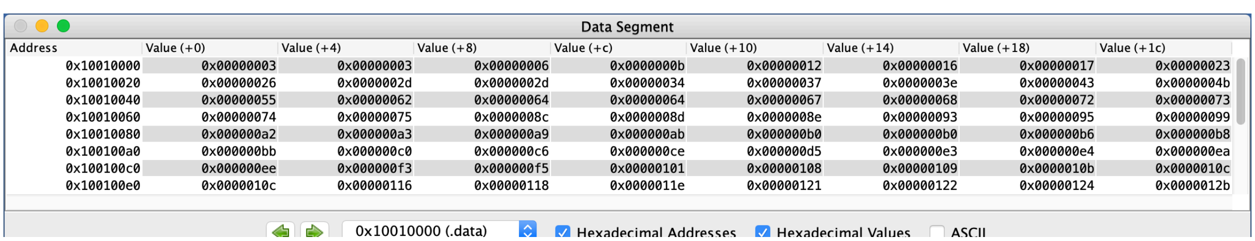
Samuel Miles
12/7/2019
ECE 36500

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:



Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000003	0x00000003	0x00000006	0x0000000b	0x00000012	0x00000016	0x00000017	0x00000023
0x10010020	0x00000026	0x0000002d	0x0000002d	0x00000034	0x00000037	0x0000003e	0x00000043	0x0000004b
0x10010040	0x00000055	0x00000062	0x00000064	0x00000064	0x00000067	0x00000068	0x00000072	0x00000073
0x10010060	0x00000074	0x00000075	0x0000008c	0x0000008d	0x0000008e	0x00000093	0x00000095	0x00000099
0x10010080	0x000000a2	0x000000a3	0x000000a9	0x000000ab	0x000000b0	0x000000b0	0x000000b6	0x000000b8
0x100100a0	0x000000bb	0x000000c0	0x000000c6	0x000000ce	0x000000d5	0x000000e3	0x000000e4	0x000000ea
0x100100c0	0x000000ee	0x000000f3	0x000000f5	0x00000101	0x00000108	0x00000109	0x0000010b	0x0000010c
0x100100e0	0x0000010c	0x00000116	0x00000118	0x0000011e	0x00000121	0x00000122	0x00000124	0x0000012b

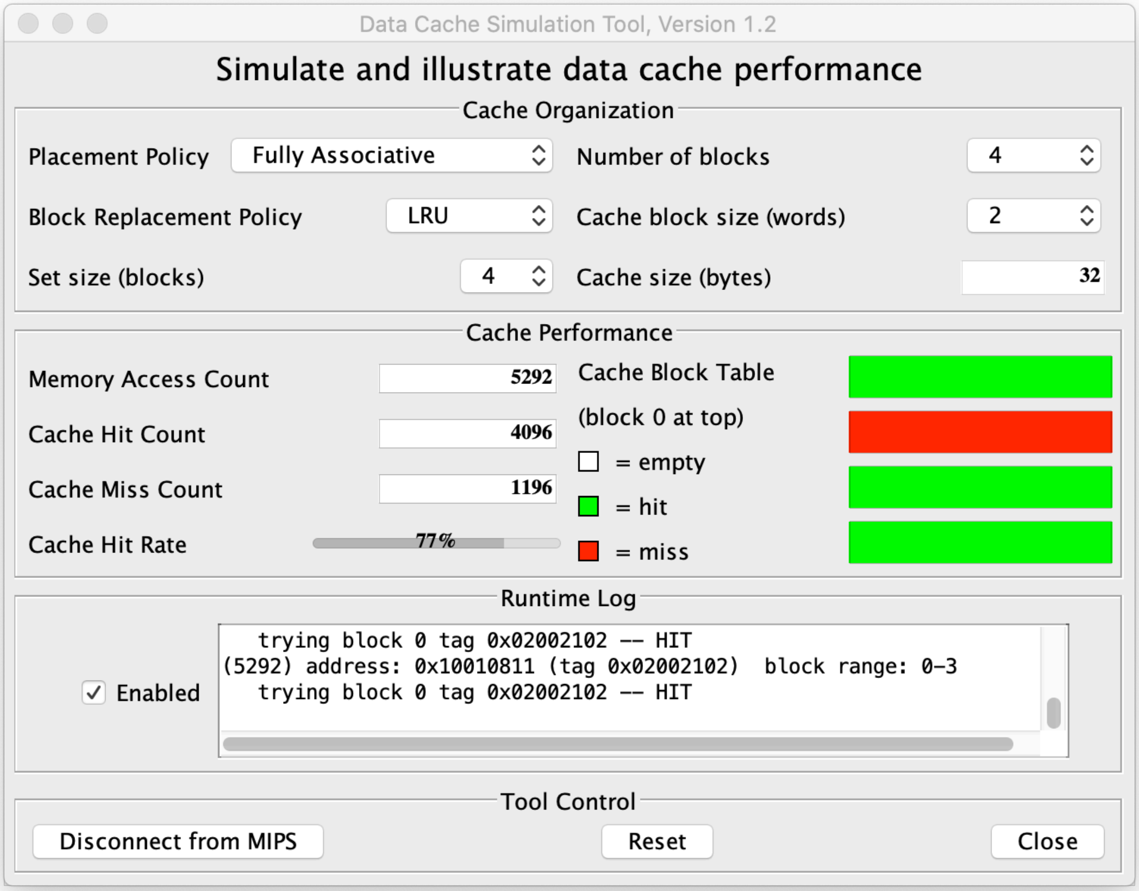
Samuel Miles
12/7/2019
ECE 36500

Output:

```
Reset: reset completed.

3
3
6
11
18
22
23
35
38
45
45
52
55
62
67
75
85
98
100
100
103
104
114
115
116
117
140
141
142
147
149
153
162
163
169
171
176
176
182
184
187
192
198
206
213
227
228
234
238
243
245
257
264
265
267
268
268
278
280
286
289
290
292

-- program is finished running --
```

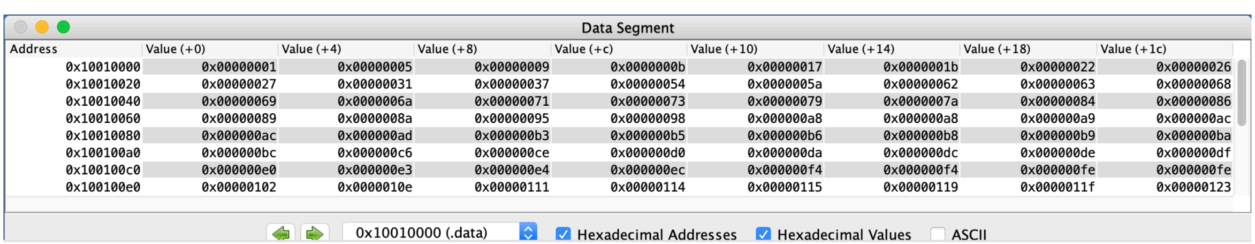
Samuel Miles
12/7/2019
ECE 36500

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:



Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000001	0x00000005	0x00000009	0x0000000b	0x00000017	0x0000001b	0x00000022	0x00000026
0x10010020	0x00000027	0x00000031	0x00000037	0x00000054	0x0000005a	0x00000062	0x00000063	0x00000068
0x10010040	0x00000069	0x0000006a	0x00000071	0x00000073	0x00000079	0x0000007a	0x00000084	0x00000086
0x10010060	0x00000089	0x0000008a	0x00000095	0x00000098	0x000000a8	0x000000a8	0x000000a9	0x000000ac
0x10010080	0x000000ac	0x000000ad	0x000000b3	0x000000b5	0x000000b6	0x000000b8	0x000000b9	0x000000ba
0x100100a0	0x000000bc	0x000000c6	0x000000ce	0x000000d0	0x000000da	0x000000dc	0x000000de	0x000000df
0x100100c0	0x000000e0	0x000000e3	0x000000e4	0x000000ec	0x000000f4	0x000000f4	0x000000fe	0x000000fe
0x100100e0	0x00000102	0x0000010e	0x00000111	0x00000114	0x00000115	0x00000119	0x0000011f	0x00000123

0x10010000 (.data) Hexadecimal Addresses Hexadecimal Values ASCII

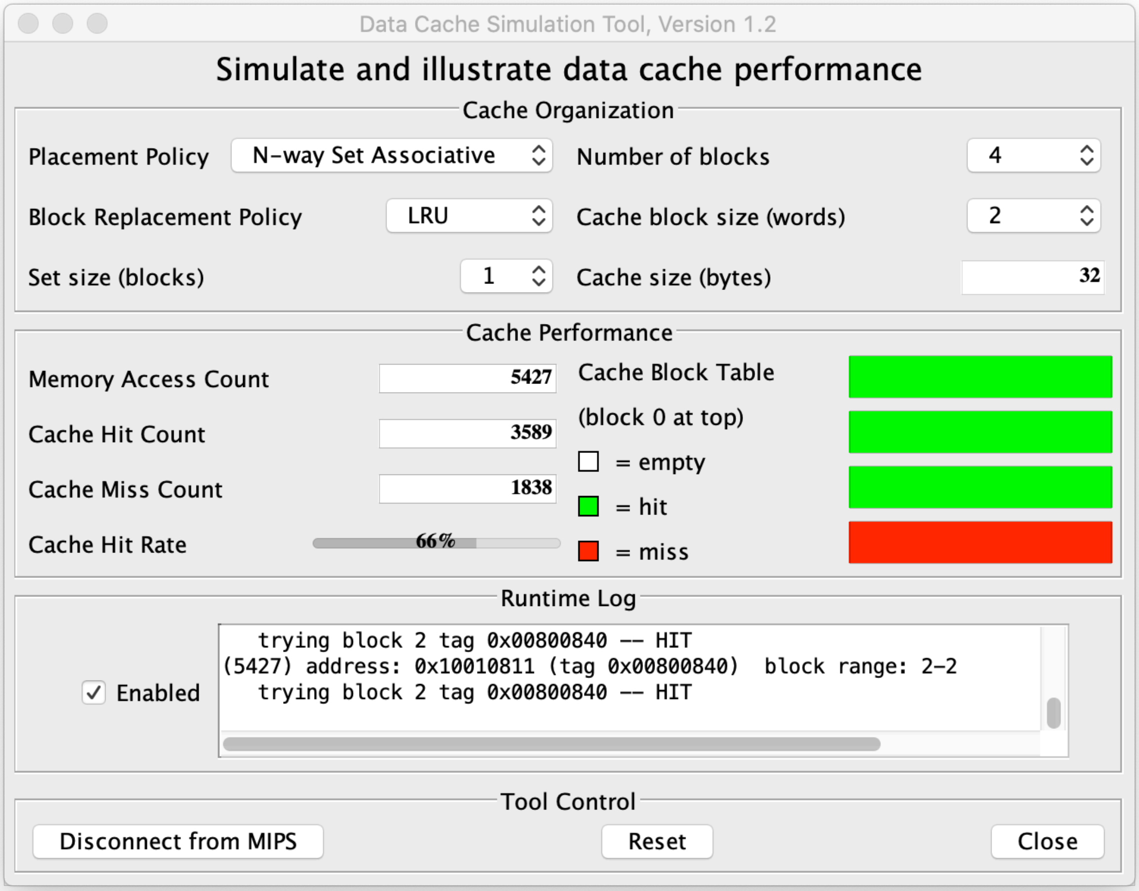
Output:

```
Reset: reset completed.

1
5
9
11
23
27
34
38
39
49
55
84
90
98
99
104
105
106
113
115
121
122
132
134
137
138
149
152
168
168
169
172
172
173
179
181
182
184
```

```
184
185
186
188
198
206
208
218
220
222
223
224
227
228
236
244
244
254
254
258
270
273
276
277
281
287
```

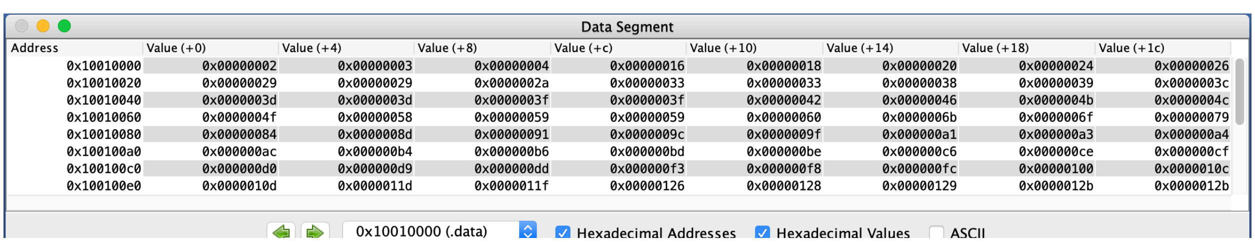
Samuel Miles
12/7/2019
ECE 36500

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:



Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000002	0x00000003	0x00000004	0x00000016	0x00000018	0x00000020	0x00000024	0x00000026
0x10010020	0x00000029	0x00000029	0x0000002a	0x00000033	0x00000033	0x00000038	0x00000039	0x0000003c
0x10010040	0x0000003d	0x0000003d	0x0000003f	0x0000003f	0x00000042	0x00000046	0x0000004b	0x0000004c
0x10010060	0x0000004f	0x00000058	0x00000059	0x00000059	0x00000060	0x0000006b	0x0000006f	0x00000079
0x10010080	0x00000084	0x0000008d	0x00000091	0x0000009c	0x0000009f	0x000000a1	0x000000a3	0x000000a4
0x100100a0	0x000000ac	0x000000b4	0x000000b6	0x000000bd	0x000000be	0x000000c6	0x000000ce	0x000000cf
0x100100c0	0x000000d0	0x000000d9	0x000000dd	0x000000f3	0x000000f8	0x000000fc	0x00000100	0x0000010c
0x100100e0	0x0000010d	0x0000011d	0x0000011f	0x00000126	0x00000128	0x00000129	0x0000012b	0x0000012b

0x10010000 (.data) Hexadecimal Addresses Hexadecimal Values ASCII

Output:

```
Reset: reset completed.

2
3
4
22
24
32
36
38
41
41
42
51
51
56
57
60
61
61
63
63
66
70
75
76
79
88
89
89
96
107
111
121
132
141
145
156
159
161
163
```

```
163
164
172
180
182
189
190
198
206
207
208
217
221
243
248
252
256
268
269
285
287
294
296
297
299

-- program is finished runnir
```

Samuel Miles
12/7/2019
ECE 36500

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.

Samuel Miles
12/7/2019
ECE 36500

<u>Application 2:</u> - 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:

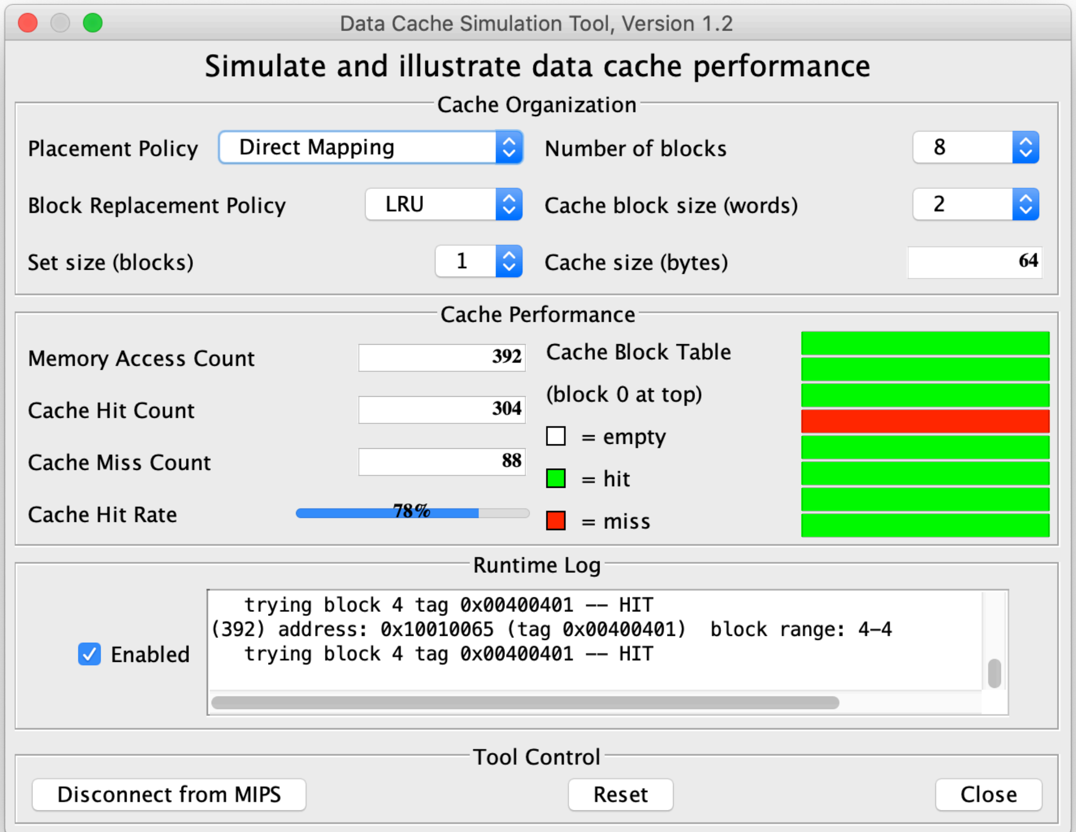
```
# Samuel Miles
# Project Phase 3 – Application 2 (The Next Value in the Sequence is a Summation of Pseudo-Random Values based on the Previous Values in the Sequence)
# ECE 36500 – F19
# 12-07-19
        .data
arr:    .space  96              #Max Array Size in Bytes (24 Values) **24 was chosen because 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
```

Samuel Miles
12/7/2019
ECE 36500

```
# 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
```

Samuel Miles
12/7/2019
ECE 36500

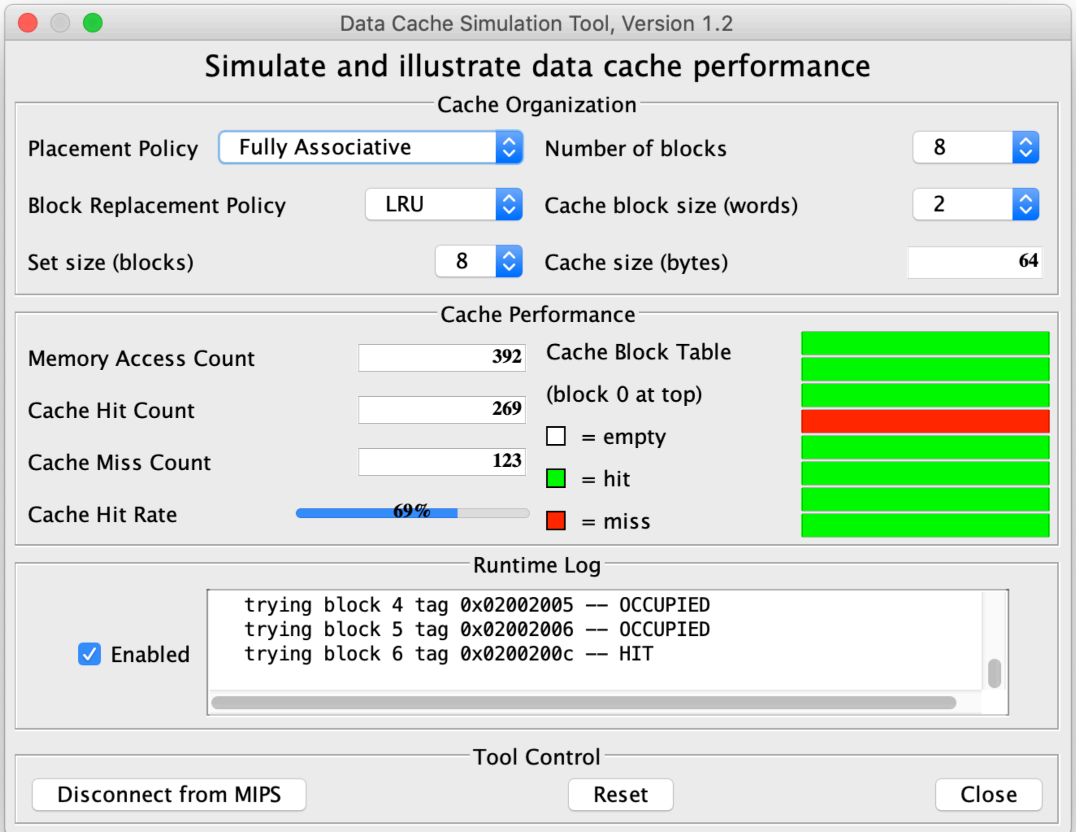
Direct Mapping Cache:



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

Samuel Miles
12/7/2019
ECE 36500

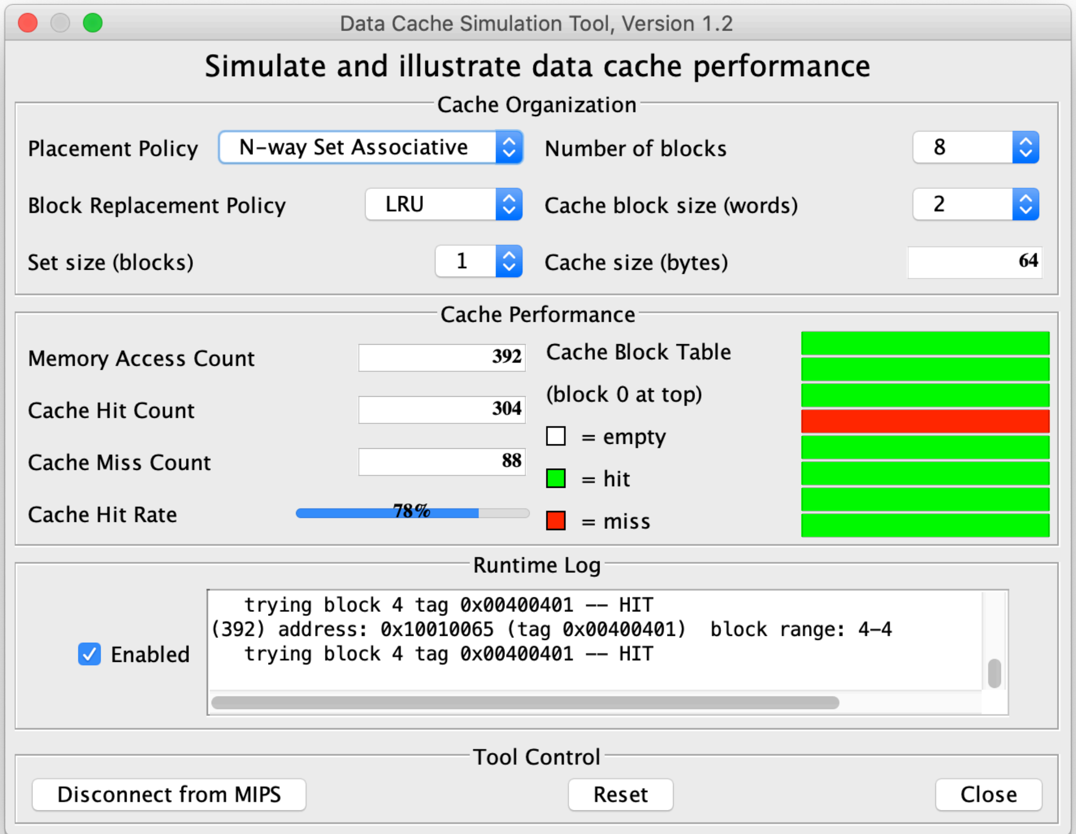
Fully Associative Cache:



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

Samuel Miles
12/7/2019
ECE 36500

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%

Samuel Miles
12/7/2019
ECE 36500

Data Segment:

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000001	0x00000001	0x00000003	0x00000008	0x00000015	0x00000037	0x00000090	0x00000179
0x10010020	0x000003db	0x00000a18	0x00001a6d	0x0000452f	0x0000b520	0x0001da31	0x0004d973	0x000cb228
0x10010040	0x00213d05	0x005704e7	0x00e3d1b0	0x02547029	0x06197ecb	0x0ff80c38	0x29cea5dd	0x6d73e55f
0x10010060	0x00000018	0x0009000a	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

Output:

```
1
1
3
8
21
55
144
377
987
2584
6765
17711
46368
121393
317811
832040
2178309
5702887
14930352
39088169
102334155
267914296
701408733

-- program is finished running --
```

Samuel Miles
12/7/2019
ECE 36500

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.