Course Name: *Computer Architecture Lab*

Course Number and Section: **14:332:333:*04***

Experiment: *5 – Simulating a Cache*

Lab Instructor: *Christos Mitropoulos*

Date Performed: *November 17th, 2016*

Date Submitted: *December 8th, 2016*

Submitted by: *FAHD HUMAYUN – 168000889 – fh186*

-------------------------For Lab Instructor Use ONLY-------------------------

GRADE: _______________

COMMENTS:

Electrical and Computer Engineering Department
School of Engineering
Rutgers University, Piscataway, NJ 08854

Introduction:

The assignments are based on simulating the cache, analyzing, and understanding the behavior of the cache and how the cache works/performs/process the instructions and the data. Implementing programs with arrays to analyze and observe the behavior of data cache when it accesses data from main memory and maps it onto data cache with different cache configurations.
A new simulator spim-cache is used, which is similar to the simulator qtspim, with additional windows for instruction cache and data cache, and additional settings/configurations for the cache.

Assignment – 1 (a):

Final values in registers		***Decimal***
R2	*0x0000000A*	*10*
R3	*0x100004B8*	*-*
R4	*0x00000000*	*0*
R5	*0xFFFFFFFD*	*-3*
R6	*0xFFFFFFEB*	*-21*
R7	*0x0000000A*	*10*

R2 was used for the address of the words of $Array_A$, but at the end of the program it is 10 because of the *li $2, 10* to exit the program

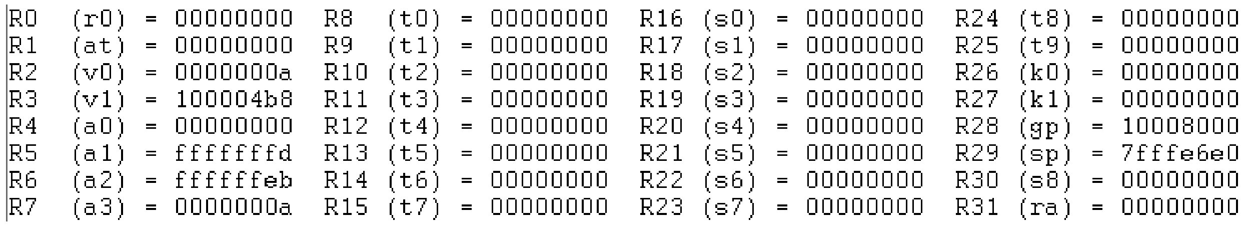
R3 was used for the address of the words of $Array_B$, and at the end has the last word's address of $Array_B$ incremented, i.e. last word of $Array_B$ is at the address 0x100004B4 and in the last iteration of the loop this address is incremented by 4 to 0x100004B8 before exiting the loop as the condition is checked after the execution of the increment instruction.

R4 was used as a counter for the loop (i.e. total number of elements in the array) and was decremented each iteration till it reached 0 and the loop ended, that is why it has a value of 0 when the program ended

R5 was used when a word was loaded from $Array_A$ and then it stored the result obtained from$Array_A[i] - Array_B[i]$ which was 0xFFFFFFFD, as this is the two's compliment representation of a number so in decimal is equivalent to -3.

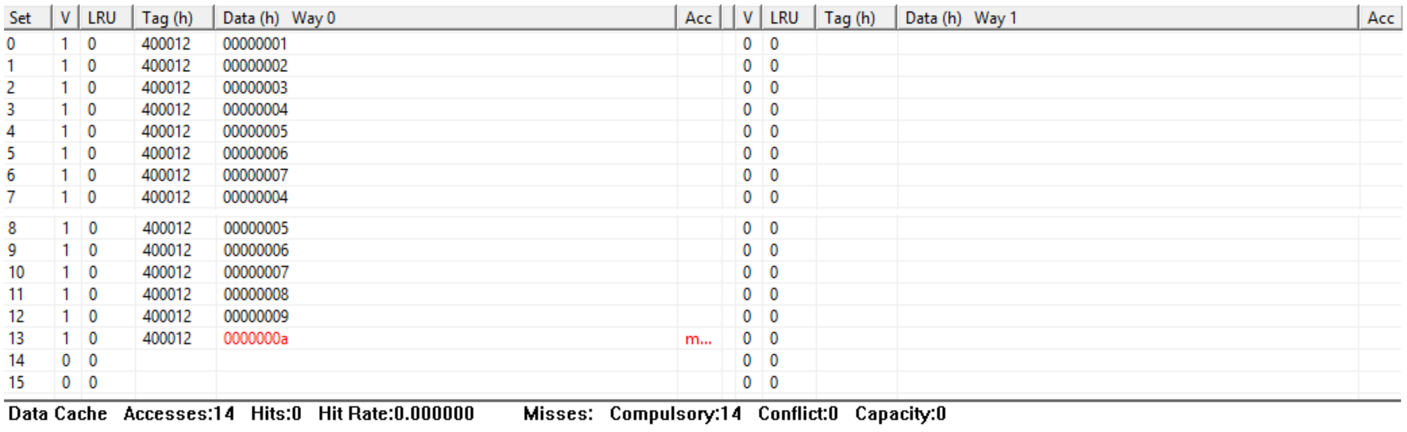
R6 was used to store the final result i.e. $result = \sum(Array_A[i] - Array_B[i])$, and the final result obtained was 0xFFFFFFEB which in decimal is -21.

R7 was used for holding the word loaded from $Array_B$ and the last word of $Array_B$ loaded was 10 in dec or 0x0000000A in hex



```
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 00000000  R9  (t1) = 00000000  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 0000000a  R10 (t2) = 00000000  R18 (s2) = 00000000  R26 (k0) = 00000000
R3  (v1) = 100004b8  R11 (t3) = 00000000  R19 (s3) = 00000000  R27 (k1) = 00000000
R4  (a0) = 00000000  R12 (t4) = 00000000  R20 (s4) = 00000000  R28 (gp) = 10008000
R5  (a1) = fffffffd  R13 (t5) = 00000000  R21 (s5) = 00000000  R29 (sp) = 7fffe6e0
R6  (a2) = ffffffeb  R14 (t6) = 00000000  R22 (s6) = 00000000  R30 (s8) = 00000000
R7  (a3) = 0000000a  R15 (t7) = 00000000  R23 (s7) = 00000000  R31 (ra) = 00000000
```

Assignment - 1(b):



Set	V	LRU	Tag (h)	Data (h) Way 0	Acc	V	LRU	Tag (h)	Data (h) Way 1	Acc
0	1	0	400012	00000001		0	0			
1	1	0	400012	00000002		0	0			
2	1	0	400012	00000003		0	0			
3	1	0	400012	00000004		0	0			
4	1	0	400012	00000005		0	0			
5	1	0	400012	00000006		0	0			
6	1	0	400012	00000007		0	0			
7	1	0	400012	00000004		0	0			
8	1	0	400012	00000005		0	0			
9	1	0	400012	00000006		0	0			
10	1	0	400012	00000007		0	0			
11	1	0	400012	00000008		0	0			
12	1	0	400012	00000009		0	0			
13	1	0	400012	0000000a	m...	0	0			
14	0	0				0	0			
15	0	0				0	0			

Data Cache Accesses:14 Hits:0 Hit Rate:0.000000 Misses: Compulsory:14 Conflict:0 Capacity:0

The cache settings were configured to

$Cache\ size = \ 128B, Block\ or\ line\ size = 4B$
And two-way set associative for mapping, i.e. 2 blocks per set

So, total number of blocks, and the number of sets are obtained as follow

$$total\ number\ of\ blocks = \frac{128B}{4B} = 32$$
$$number\ of\ sets = \frac{number\ of\ blocks}{number\ of\ blocks\ per\ set} = \frac{32}{2} = 16$$

The screenshot above shows the *sets* in first column numbered from 0 to 15 i.e. 16 sets in total.
The contents of the data cache after the program is completed are in the screenshot above.

The data cache contents are obtained after the instructions $lw\ \$5, 0(\$2)\ and\ lw\ \$7, 0(\$3)$.
Let us have a look at the first load instruction in the beginning of the loop the register 2 has an address of the first word of the $Array_A$ i.e. 0x10000480 which can be seen in the DATA MEMORY window of the PCSpim-Cache simulator. When the instruction is executed from the instruction cache (information about instruction cache in part c of this assignment) there is a *miss* in the data cache (initially data cache is empty, all the sets and blocks/lines empty) it means that the word/element of the $Array_A$ needs to be mapped to the data cache. This is achieved as follow:
The address is 0x10000480 which in binary is 0001 0000 0000 0000 0000 0100 1000 0000, now these are 32 bits of the address which represents the *tag, index,* and *offset*, which are given as follow:

$Index = \log_2(total\ number\ of\ sets) = \log_2(16) = 4\ bits$
$Offset = \log_2(block\ size) = \log_2(4) = 2\ bits$
$Tag = (total\ number\ address\ bits) - (index\ bits) - (offset\ bits) = 32 - 4 - 2$
$= 26\ bits$

Address	***Tag***	***Index***	***Offset***
0x10000480	*26 bits*	*4 bits*	*2 bits*
Binary	*00 0100 0000 0000 0000 0001 0010*	*0000*	*00*
Hex	*400012*	*0*	*0*

The *index* shows in which set it should be placed and the offset is block offset. If the data at that address is in the cache, then we use the block offset from that address to find the data within the cache block where the data was found. So, the first word of $Array_A$ (i.e. 1) is mapped to the *way 0* or *block 0 of set 0.*

Similarly, for the second load instruction for the $Array_B$, its first word is at 0x1000049C, so we get

Address	***Tag***	***Index***	***Offset***
0x1000049C	*26 bits*	*4 bits*	*2 bits*
Binary	*00 0100 0000 0000 0000 0001 0010*	*0111*	*00*
Hex	*400012*	*7*	*0*

As the index can be seen as 7 that is why when the load instruction for $Array_B$ is executed to load the first word from the array there is a *miss* at set 7 in the data cache and then the word/element 4 is mapped there. So, the first word of $Array_B$ (i.e. 7) is mapped to the *way 0* or *block 0 of set 7.*

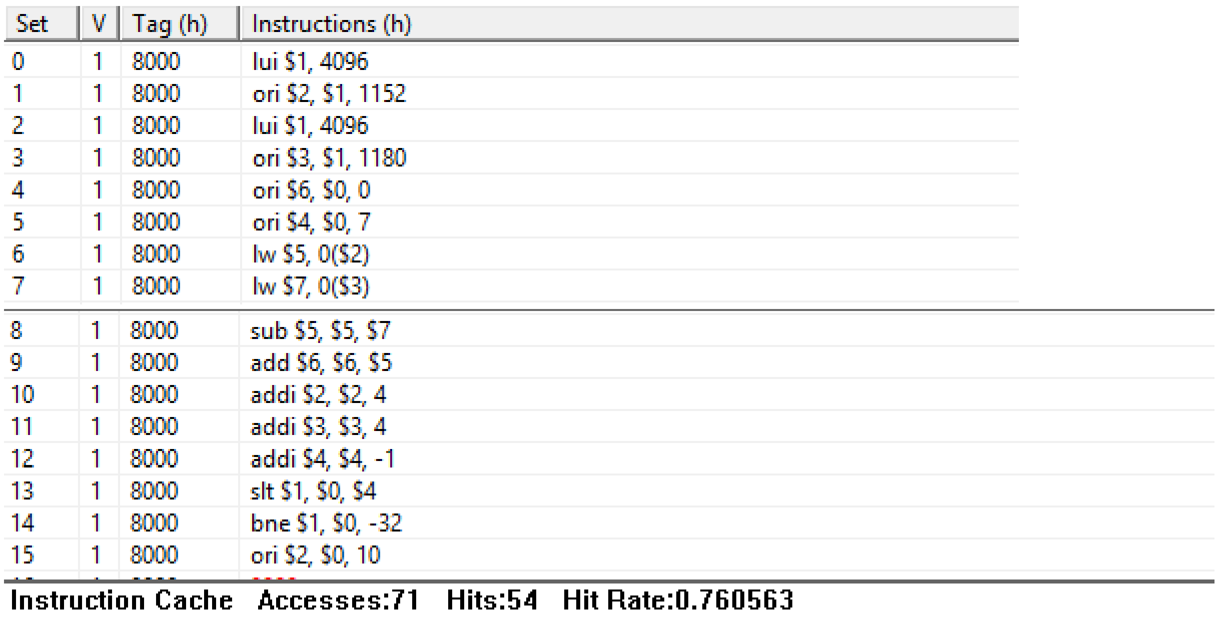
This procedure is followed for mapping all of the contents to data cache.

In the *data statics* it can be seen that the *hits* and *hit rate* is 0 because the size of block is 4 bytes which means only one word is stored in a block and none of the words accessed were accessed again each word from the data memory was accessed once that is why each of them as was a miss and hence resulting in a *hit rate* as 0 (i.e. 0/14 = 0 or 0%). The misses are 14 as all of the 14 words ended up with a miss so miss rate is 1 (i.e. 14/14 = 1 or 100%)

V is the valid bit which is set to 1 when a word is loaded into the data cache.

LRU is the least recently used bit which is used when replacing of block is needed. When a word/block is to be replaced and there is already a word placed so then LRU bit determines which block of the set to replace, LRU is only needed when there is set associative mapping.

Assignment - 1(c):



Set	V	Tag (h)	Instructions (h)
0	1	8000	lui $1, 4096
1	1	8000	ori $2, $1, 1152
2	1	8000	lui $1, 4096
3	1	8000	ori $3, $1, 1180
4	1	8000	ori $6, $0, 0
5	1	8000	ori $4, $0, 7
6	1	8000	lw $5, 0($2)
7	1	8000	lw $7, 0($3)
8	1	8000	sub $5, $5, $7
9	1	8000	add $6, $6, $5
10	1	8000	addi $2, $2, 4
11	1	8000	addi $3, $3, 4
12	1	8000	addi $4, $4, -1
13	1	8000	slt $1, $0, $4
14	1	8000	bne $1, $0, -32
15	1	8000	ori $2, $0, 10

```
Instruction Cache   Accesses:71   Hits:54   Hit Rate:0.760563
```

First instruction is at address 0x00400000. In the first step PC register can be seen as having the value 0x00400000. In the next step there is a *miss* while accessing the instruction from the instruction cache because in the beginning instruction cache is empty. In the next step because of the miss, instruction is mapped/loaded into the instruction cache, and then the next step executes the instruction.

The procedure for mapping is as follow.
Settings for instruction cache:
$Cache\ size = 128B,\quad Block\ or\ line\ size = 4B$
Direct-mapping (or 1 block per set)

$$Total\ number\ of\ blocks = \frac{128B}{4B} = 32$$
$$Total\ number\ of\ sets = Total\ number\ of\ blocks = 32\ (for\ direct\ mapping)$$
$$Index = \log_2(\ total\ number\ of\ sets\ or\ blocks) = \log_2(32) = 5\ bits$$
$$Offset = \log_2(block\ size) = \log_2(4) = 2\ bits$$
$$Tag = (address\ bits) - (index\ bits) - (offset\ bits) = 32 - 5 - 2 = 25\ bits$$

Address	***Tag***	***Index***	***Offset***
0x00400000	*25 bits*	*5 bits*	*2 bits*
Binary	*0 0000 0000 1000 0000 0000 0000*	*0000*	*00*
Hex	*8000*	*0*	*0*

As discussed for the data cache, same is for instruction cache index would determine the set/block (for direct mapping) to map the instruction in instruction cache. As in the table it can be seen that tag after splitting the bits of address was obtained as 8000 and same can be seen in the screenshot that the tag column has 8000.

The instructions are mapped onto the instruction cache and they are a miss, the first hit is obtained when the next iteration of loop starts that is from the instruction *lw $5, 0($2)*, now this time the instruction is already in the instruction cache at set 6, so the instruction is executed in a single step because of the hit (saving the step of copying the instruction from memory and loading it to the instruction cache).

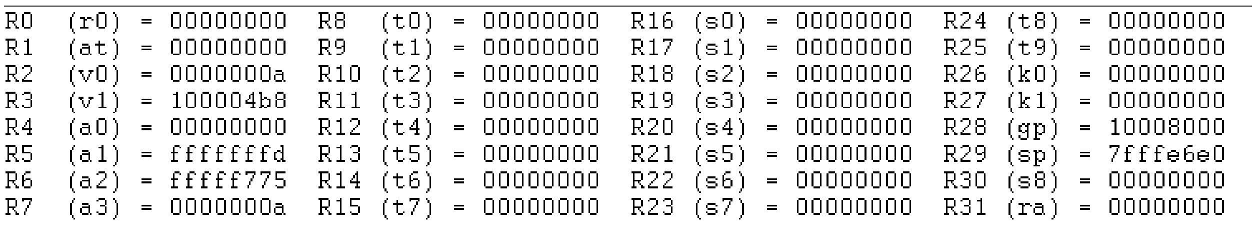
In the instruction cache it can be seen as there are total of 71 accesses and 54 of those accesses are hit with a hit rate of 0.76. The 54 hits are because there are 9 instructions executed in the loop (i.e. from set number 6 to set number 14 in the instruction cache), the loop is run 7 times, the first time all of the instructions are a miss and loaded into the instruction cache, the next 6 iterations of the loop given 6*9 = 54 hits because the instructions are already in the instruction cache that needs to be accessed during the loop.

V is the valid bit which is set to 1 when an instruction is loaded into the instruction cache.
As this is direct mapping so LRU not needed.

Assembly code attached.

Assignment - 2:

Final values in registers		*Decimal*
R2	*0x0000000A*	*10*
R3	*0x100004B8*	*-*
R4	*0x00000000*	*0*
R5	*0xFFFFFFFD*	*-3*
R6	*0xFFFFF775*	*-2187*
R7	*0x0000000A*	*10*

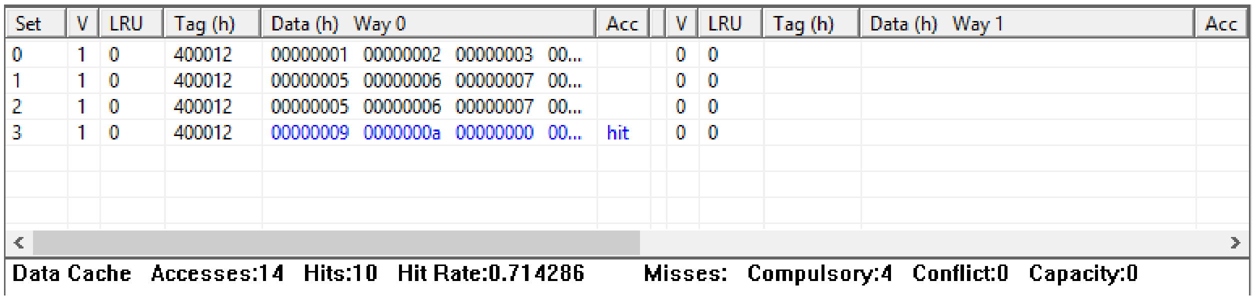


```
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 00000000  R9  (t1) = 00000000  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 0000000a  R10 (t2) = 00000000  R18 (s2) = 00000000  R26 (k0) = 00000000
R3  (v1) = 100004b8  R11 (t3) = 00000000  R19 (s3) = 00000000  R27 (k1) = 00000000
R4  (a0) = 00000000  R12 (t4) = 00000000  R20 (s4) = 00000000  R28 (gp) = 10008000
R5  (a1) = fffffffd  R13 (t5) = 00000000  R21 (s5) = 00000000  R29 (sp) = 7fffe6e0
R6  (a2) = fffff775  R14 (t6) = 00000000  R22 (s6) = 00000000  R30 (s8) = 00000000
R7  (a3) = 0000000a  R15 (t7) = 00000000  R23 (s7) = 00000000  R31 (ra) = 00000000
```

The result is obtained as:

$$\prod(Array_A[i] - Array_B[i]) = (-3) * (-3) * (-3) * (-3) * (-3) * (-3) * (-3) = (-2187)_{10}$$
$$= (FFFFF775)_{16}$$

Contents of the data cache:



Set	V	LRU	Tag (h)	Data (h) Way 0	Acc	V	LRU	Tag (h)	Data (h) Way 1	Acc
0	1	0	400012	00000001 00000002 00000003 00...		0	0			
1	1	0	400012	00000005 00000006 00000007 00...		0	0			
2	1	0	400012	00000005 00000006 00000007 00...		0	0			
3	1	0	400012	00000009 0000000a 00000000 00...	hit	0	0			

Data Cache Accesses:14 Hits:10 Hit Rate:0.714286 Misses: Compulsory:4 Conflict:0 Capacity:0

The cache settings were configured to

$$Cache\ size = 128B$$
$$Block\ or\ line\ size = 16B$$

And four-way set associative for mapping, i.e. 4 blocks per set

So, total number of blocks, and the number of sets are obtained as follow

$$total\ number\ of\ blocks = \frac{256B}{16B} = 16$$
$$number\ of\ sets = \frac{total\ number\ of\ blocks}{number\ of\ blocks\ per\ set} = \frac{16}{4} = 4$$

The screenshot above shows the *sets* in first column numbered from 0 to 3 i.e. 4 sets in total. The contents of the data cache after the program is completed are in the screenshot above.

The data cache contents are obtained after the instructions $lw\ \$5, 0(\$2) and\ lw\ \$7, 0(\$3)$. Let us have a look at the first load instruction in the beginning of the loop the register 2 has an address of the first word of the $Array_A$ i.e. 0x10000480 which can be seen in the DATA MEMORY window of the PCSpim-Cache simulator. When the instruction is executed from the instruction cache there is a *miss* in the data cache (initially data cache is empty, all the sets and blocks/lines empty) it means that the word/element of the $Array_A$ needs to be mapped to the data cache. This is achieved as follow:

The address is 0x10000480 which in binary is 0001 0000 0000 0000 0000 0100 1000 0000, now these are 32 bits of the address which represents the *tag, index,* and *offset*, which are given as follow:

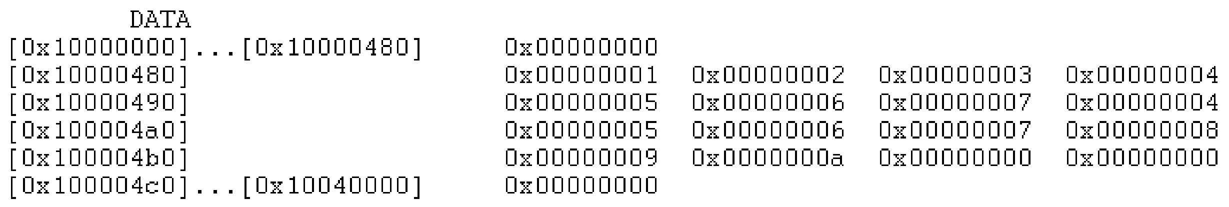
$$Index = \log_2(total\ number\ of\ sets) = \log_2(4) = 2\ bits$$
$$Offset = \log_2(block\ size) = \log_2(16) = 4\ bits$$
$$Tag = (address\ bits) - (index\ bits) - (offset\ bits) = 32 - 4 - 2 = 26\ bits$$

Address	*Tag*	*Index*	*Offset*
0x10000480	*26 bits*	*2 bits*	*4 bits*
Binary	*00 0100 0000 0000 0000 0001 0010*	*00*	*0000*
Hex	*400012*	*0*	*0*

The *index* shows in which set it should be placed and the offset is block offset. If the data at that address is in the cache, then we use the block offset from that address to find the data within the cache block where the data was found.

Spatial locality: if an item is referenced, items with addresses that are close by will tend to be referenced soon.

As the size of block is 16B or 4 words per block, so because of spatial locality when the word is accessed from data memory, the neighboring words (addresses) are also placed in the block which is determined by the size of the block. In this case the block can have 4 words and the address is 0x10000480 so all of the addresses in that block are placed. The screenshot below shows the data memory:



```
        DATA
[0x10000000]...[0x10000480]     0x00000000
[0x10000480]                    0x00000001  0x00000002  0x00000003  0x00000004
[0x10000490]                    0x00000005  0x00000006  0x00000007  0x00000004
[0x100004a0]                    0x00000005  0x00000006  0x00000007  0x00000008
[0x100004b0]                    0x00000009  0x0000000a  0x00000000  0x00000000
[0x100004c0]...[0x10040000]     0x00000000
```

As it can be seen from the image above each of the row can be considered as a block because in our block it can have 4 words and each row in the above image has 4 words, so, the address 0x10000480 is in the first row and that entire row is placed in set 0 with the words 1 2 3 4 of the $array_A$, which can be seen in the screenshot of data contents given earlier. Now, in the next iteration when the next word of first array (i.e. 2) is accessed it will result in a hit as it is already in data cache and it can be accessed as follow:

Address	*Tag*	*Index*	*Offset*
0x10000484	*26 bits*	*2 bits*	*4 bits*
Binary	*00 0100 0000 0000 0000 0001 0010*	*00*	*0100*
Hex	*400012*	*0*	*4*

The offset in this case is 4 which means that the second word in the data cache set 0 (because index 0) is going to be accessed (valid bit is 1 which means that what is in the set in particular block is valid data).

Similarly, for the second load instruction for the $Array_B$ its first word is at 0x1000049C, so we get

Address	***Tag***	***Index***	***Offset***
0x1000049C	*26 bits*	*2 bits*	*4 bits*
Binary	*00 0100 0000 0000 0000 0001 0010*	*01*	*1100*
Hex	*400012*	*1*	*12 (in decimal)*

As the index can be seen as 1 that is why when the load instruction for $Array_B$ is executed to load the first word from the array there is a *miss* at set 1 in the data cache and then the word/element 4 is mapped there.

So, the first word of $Array_B$(i.e. 4) is at the address 0x1000049C which in the screenshot of data memory can be seen as in row 2 (the word is at the end of row 2) but the entire row is placed in set 1 because index is 1 and the offset is 12 which means the fourth word to be accessed as the first word in set 1 is offset 0, the second word is offset 4, the third word is offset 8, and the fourth word is offset 12, because size of the word is 4 bytes.

One thing to notice, in the second iteration while trying to access the second word of first array ended up with a hit, that would not be the case while trying to access the second word of second array because it cannot be found in the data cache. That is why it would result in a miss and then similar procedure as discussed is followed and the third row from the data memory is placed in the set 3.

This procedure is followed for mapping all of the contents to data cache.

In the *data statics* it can be seen that the *hits is 10* and *hit rate* is 0.71. This is because of placing 4 words from the memory into the set of data cache. So, when the first word of first array (i.e. 1) was a miss the other 6 words were a hit (i.e. 2, 3, 4, 5, 6, 7), similarly, the first, second, and second last words of the second array were a miss (i.e. 4, 5, 9), and the other words were a hit (i.e. 6,7,8,10). So, total of 4 misses and 10 hits. Hit rate is 10/14 = 0.71 (i.e. 71%) and the Miss rate is 4/14 = 0.29 (i.e. 29%).

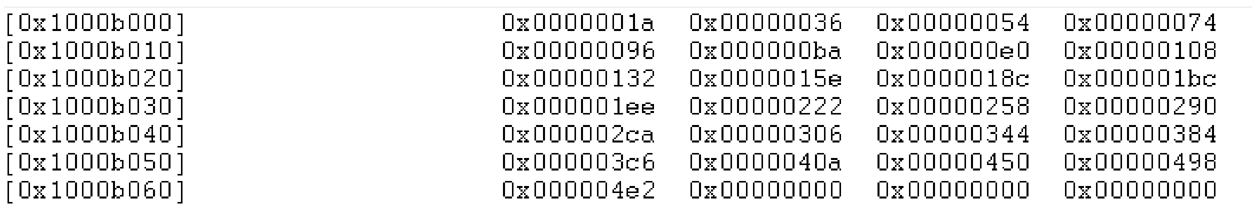
V is the valid bit which is set to 1 when a word is loaded into the data cache.

LRU is the least recently used bit which is used when replacing of block is needed. When a word/block is to be replaced and there is already a word placed so then LRU bit determines which block of the set to replace, LRU is only needed when there is set associative mapping.

Assembly code attached:

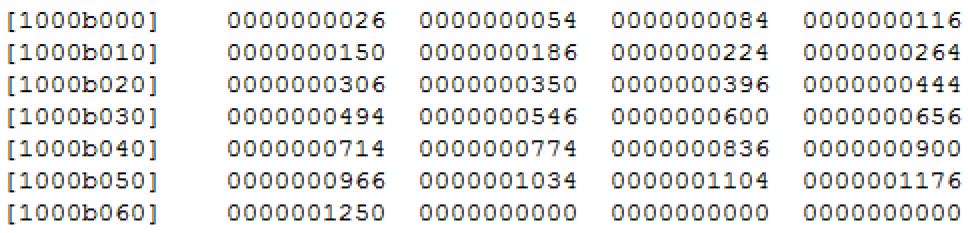
Assignment - 3:

The screenshot of matrix_C starting at address 0x1000B000, the values are below are in HEX:



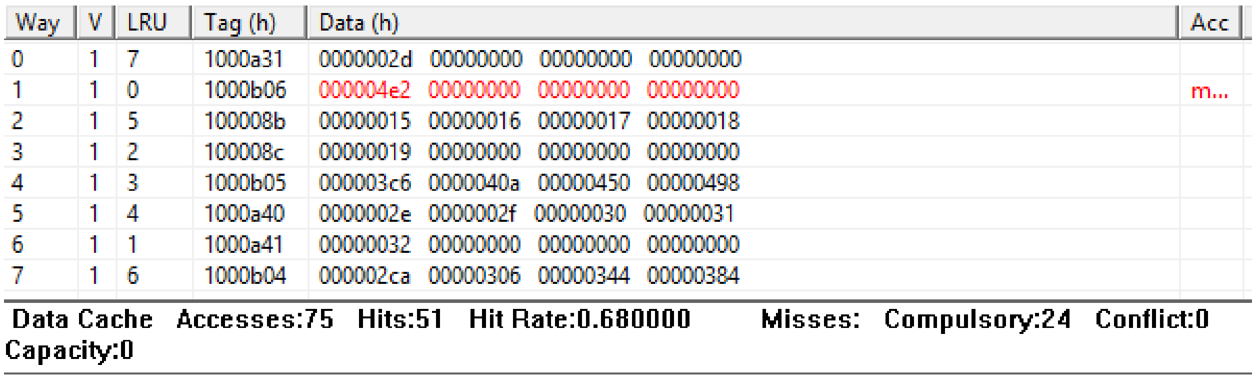
```
[0x1000b000]                        0x0000001a   0x00000036   0x00000054   0x00000074
[0x1000b010]                        0x00000096   0x000000ba   0x000000e0   0x00000108
[0x1000b020]                        0x00000132   0x0000015e   0x0000018c   0x000001bc
[0x1000b030]                        0x000001ee   0x00000222   0x00000258   0x00000290
[0x1000b040]                        0x000002ca   0x00000306   0x00000344   0x00000384
[0x1000b050]                        0x000003c6   0x0000040a   0x00000450   0x00000498
[0x1000b060]                        0x000004e2   0x00000000   0x00000000   0x00000000
```

The screenshot of matrix_C starting at address 0x1000B000, the values in DEC:



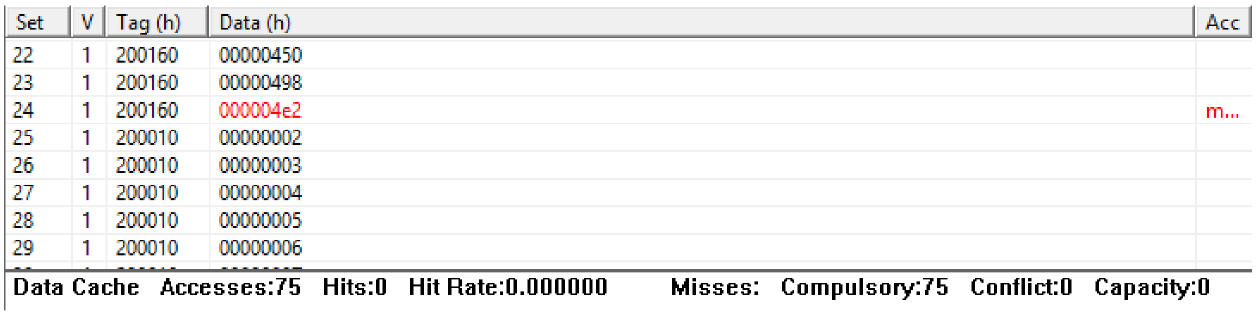
```
[1000b000]    0000000026  0000000054  0000000084  0000000116
[1000b010]    0000000150  0000000186  0000000224  0000000264
[1000b020]    0000000306  0000000350  0000000396  0000000444
[1000b030]    0000000494  0000000546  0000000600  0000000656
[1000b040]    0000000714  0000000774  0000000836  0000000900
[1000b050]    0000000966  0000001034  0000001104  0000001176
[1000b060]    0000001250  0000000000  0000000000  0000000000
```

Couple of screenshots of data cache with different configurations:



Way	V	LRU	Tag (h)	Data (h)	Acc
0	1	7	1000a31	0000002d 00000000 00000000 00000000	
1	1	0	1000b06	000004e2 00000000 00000000 00000000	m...
2	1	5	100008b	00000015 00000016 00000017 00000018	
3	1	2	100008c	00000019 00000000 00000000 00000000	
4	1	3	1000b05	000003c6 0000040a 00000450 00000498	
5	1	4	1000a40	0000002e 0000002f 00000030 00000031	
6	1	1	1000a41	00000032 00000000 00000000 00000000	
7	1	6	1000b04	000002ca 00000306 00000344 00000384	

Data Cache   Accesses:75   Hits:51   Hit Rate:0.680000   Misses: Compulsory:24   Conflict:0 Capacity:0



Set	V	Tag (h)	Data (h)	Acc
22	1	200160	00000450	
23	1	200160	00000498	
24	1	200160	000004e2	m...
25	1	200010	00000002	
26	1	200010	00000003	
27	1	200010	00000004	
28	1	200010	00000005	
29	1	200010	00000006	

Data Cache   Accesses:75   Hits:0   Hit Rate:0.000000   Misses: Compulsory:75   Conflict:0   Capacity:0

The results in data cache after running the program with different configuration is summarized below:

Direct Mapping						
cache size	*block size*	*Hits*	*Hit rate*	*compulsory*	*conflict*	*misses*
1024	*16*	*45*	*0.6*	*24*	*6*	*30*
	8	*30*	*0.4*	*41*	*4*	*45*
	4	*0*	*0*	*75*	*0*	*75*
512	*16*	*45*	*0.6*	*24*	*6*	*30*
	8	*30*	*0.4*	*41*	*4*	*45*
	4	*0*	*0*	*75*	*0*	*75*
256	*16*	*45*	*0.6*	*24*	*6*	*30*
	8	*30*	*0.4*	*41*	*4*	*45*
	4	*0*	*0*	*75*	*0*	*75*
128	*16*	*40*	*0.53*	*24*	*11*	*35*
	8	*30*	*0.4*	*41*	*4*	*45*
	4	*0*	*0*	*75*	*0*	*75*

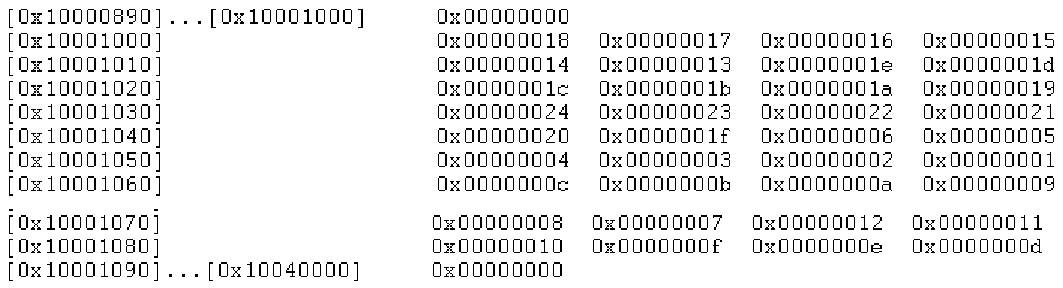
2 Ways Set-Associative						
cache size	***block size***	***Hits***	***Hit rate***	***compulsory***	***conflict***	***misses***
1024	*16*	*51*	*0.68*	*24*	*0*	*24*
	8	*34*	*0.45*	*41*	*0*	*41*
	4	*0*	*0*	*75*	*0*	*75*
512	*16*	*51*	*0.68*	*24*	*0*	*24*
	8	*34*	*0.45*	*41*	*0*	*41*
	4	*0*	*0*	*75*	*0*	*75*
256	*16*	*51*	*0.68*	*24*	*0*	*24*
	8	*34*	*0.45*	*41*	*0*	*41*
	4	*0*	*0*	*75*	*0*	*75*
128	*16*	*51*	*0.68*	*24*	*0*	*24*
	8	*34*	*0.45*	*41*	*0*	*41*
	4	*0*	*0*	*75*	*0*	*75*

4 Ways Set-Associative						
cache size	***block size***	***Hits***	***Hit rate***	***compulsory***	***conflict***	***misses***
1024	*16*	*51*	*0.68*	*24*	*0*	*24*
	8	*34*	*0.45*	*41*	*0*	*41*
	4	*0*	*0*	*75*	*0*	*75*
512	*16*	*51*	*0.68*	*24*	*0*	*24*
	8	*34*	*0.45*	*41*	*0*	*41*
	4	*0*	*0*	*75*	*0*	*75*
256	*16*	*51*	*0.68*	*24*	*0*	*24*
	8	*34*	*0.45*	*41*	*0*	*41*
	4	*0*	*0*	*75*	*0*	*75*
128	*16*	*51*	*0.68*	*24*	*0*	*24*
	8	*34*	*0.45*	*41*	*0*	*41*
	4	*0*	*0*	*75*	*0*	*75*

Fully Associative						
cache size	***block size***	***Hits***	***Hit rate***	***compulsory***	***conflict***	***misses***
1024	*16*	*51*	*0.68*	*24*	*0*	*24*
	8	*34*	*0.45*	*41*	*0*	*41*
	4	*0*	*0*	*75*	*0*	*75*
512	*16*	*51*	*0.68*	*24*	*0*	*24*
	8	*34*	*0.45*	*41*	*0*	*41*
	4	*0*	*0*	*75*	*0*	*75*
256	*16*	*51*	*0.68*	*24*	*0*	*24*
	8	*34*	*0.45*	*41*	*0*	*41*
	4	*0*	*0*	*75*	*0*	*75*
128	*16*	*51*	*0.68*	*24*	*0*	*24*
	8	*34*	*0.45*	*41*	*0*	*41*
	4	*0*	*0*	*75*	*0*	*75*

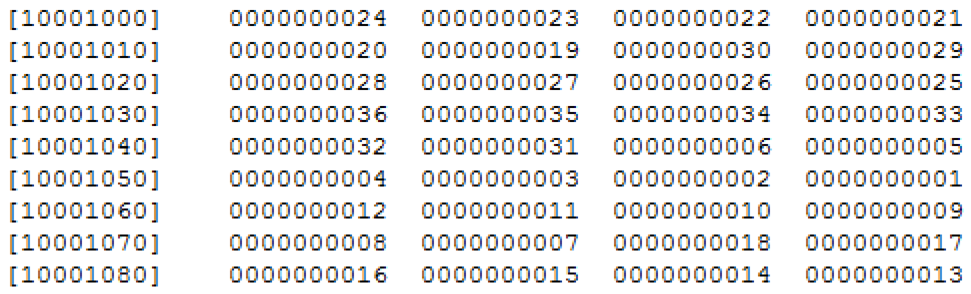
Assignment - 4:

Screenshot of transform matrix in hex values



```
[0x10000890]...[0x10001000]      0x00000000
[0x10001000]                     0x00000018  0x00000017  0x00000016  0x00000015
[0x10001010]                     0x00000014  0x00000013  0x0000001e  0x0000001d
[0x10001020]                     0x0000001c  0x0000001b  0x0000001a  0x00000019
[0x10001030]                     0x00000024  0x00000023  0x00000022  0x00000021
[0x10001040]                     0x00000020  0x0000001f  0x00000006  0x00000005
[0x10001050]                     0x00000004  0x00000003  0x00000002  0x00000001
[0x10001060]                     0x0000000c  0x0000000b  0x0000000a  0x00000009
[0x10001070]                     0x00000008  0x00000007  0x00000012  0x00000011
[0x10001080]                     0x00000010  0x0000000f  0x0000000e  0x0000000d
[0x10001090]...[0x10040000]      0x00000000
```

Screenshot of transform matrix in decimal values



```
[10001000]     0000000024  0000000023  0000000022  0000000021
[10001010]     0000000020  0000000019  0000000030  0000000029
[10001020]     0000000028  0000000027  0000000026  0000000025
[10001030]     0000000036  0000000035  0000000034  0000000033
[10001040]     0000000032  0000000031  0000000006  0000000005
[10001050]     0000000004  0000000003  0000000002  0000000001
[10001060]     0000000012  0000000011  0000000010  0000000009
[10001070]     0000000008  0000000007  0000000018  0000000017
[10001080]     0000000016  0000000015  0000000014  0000000013
```

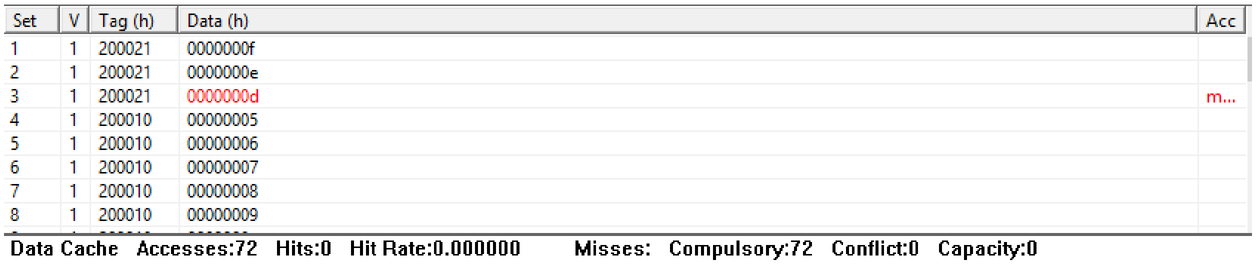
Each row has 6 elements so according to the data memory the rows are:

Row 1 of transform matrix:	24	23	22	21	20	19
Row 2 of transform matrix:	30	29	28	27	26	25
Row 3 of transform matrix:	36	35	34	33	32	31
Row 4 of transform matrix:	06	05	04	03	02	01
Row 5 of transform matrix:	12	11	10	09	08	07
Row 6 of transform matrix:	18	17	16	15	14	13

Assembly code attached:

Assignment - 5:

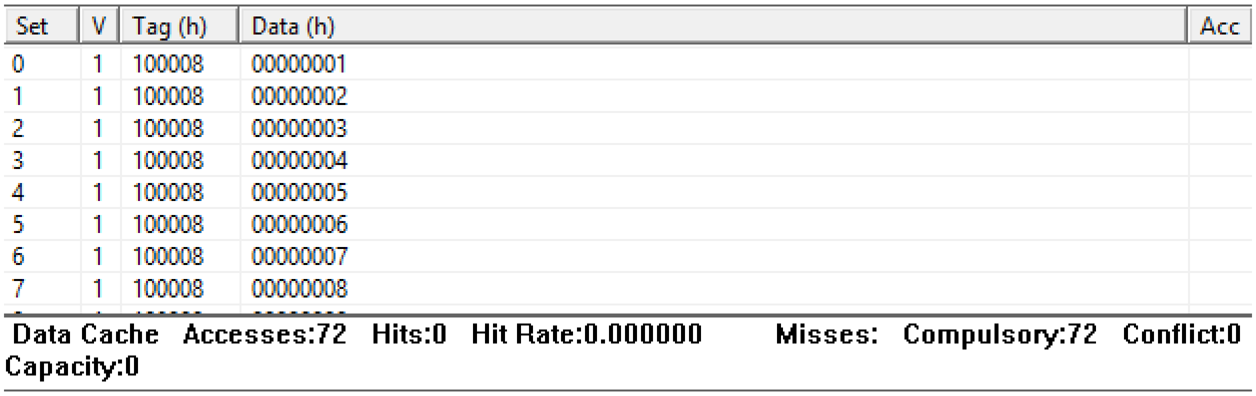
Screenshot of configuration Cache size = 128 bytes, Block size = 4 bytes, direct-mapping



Set	V	Tag (h)	Data (h)	Acc
1	1	200021	0000000f	
2	1	200021	0000000e	
3	1	200021	0000000d	m...
4	1	200010	00000005	
5	1	200010	00000006	
6	1	200010	00000007	
7	1	200010	00000008	
8	1	200010	00000009	

```
Data Cache   Accesses:72   Hits:0   Hit Rate:0.000000      Misses:   Compulsory:72   Conflict:0   Capacity:0
```

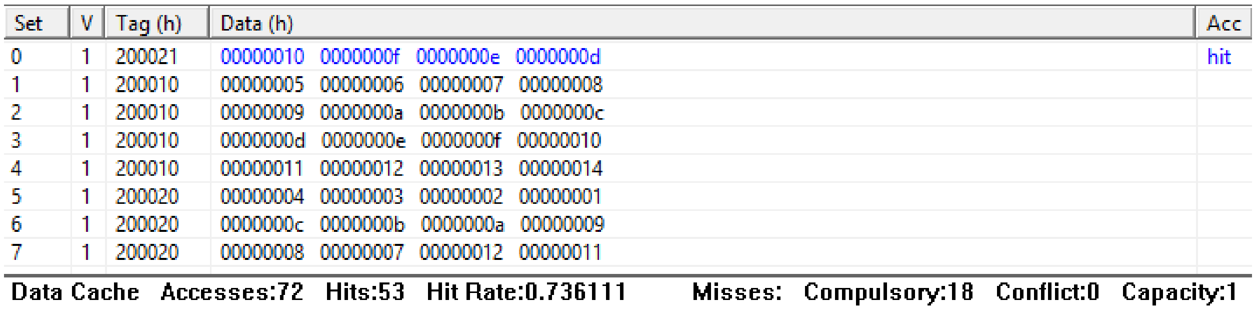
Screenshot of configuration Cache size = 256 bytes, Block size = 4 bytes, direct-mapping



Set	V	Tag (h)	Data (h)	Acc
0	1	100008	00000001	
1	1	100008	00000002	
2	1	100008	00000003	
3	1	100008	00000004	
4	1	100008	00000005	
5	1	100008	00000006	
6	1	100008	00000007	
7	1	100008	00000008	

```
Data Cache   Accesses:72   Hits:0   Hit Rate:0.000000      Misses:   Compulsory:72   Conflict:0
Capacity:0
```

Screenshot of configuration Cache size = 128 bytes, Block size = 16 bytes, direct-mapping



Set	V	Tag (h)	Data (h)	Acc
0	1	200021	00000010 0000000f 0000000e 0000000d	hit
1	1	200010	00000005 00000006 00000007 00000008	
2	1	200010	00000009 0000000a 0000000b 0000000c	
3	1	200010	0000000d 0000000e 0000000f 00000010	
4	1	200010	00000011 00000012 00000013 00000014	
5	1	200020	00000004 00000003 00000002 00000001	
6	1	200020	0000000c 0000000b 0000000a 00000009	
7	1	200020	00000008 00000007 00000012 00000011	

```
Data Cache   Accesses:72   Hits:53   Hit Rate:0.736111      Misses:   Compulsory:18   Conflict:0   Capacity:1
```

Summarized in the table below:

Direct Mapping						
Run	***Cache Size***	***block size***	***Accesses***	***Hits***	***Misses***	***Miss Rate***
1	*128*	*4*	*72*	*0*	*72*	*1.00 (100%)*
2	*256*	*4*	*72*	*0*	*72*	*1.00 (100%)*
3	*128*	*16*	*72*	*53*	*19*	*0.26 (26%)*

The miss rate is 1.00 (100%) in the first two configurations because the block size is as such that only one word can fit in the block, as each of the word is only accessed once (i.e. each element in the matrix) and there is only one word to be stored in the block, therefore, each of the words ends up as a miss. Even though the cache size has been increased in the second run but the block size was kept same in the first two run, and if the block size is large multiple words can be placed in a block (spatial locality) which is helpful special accessing words in an array as multiple words would be placed from an array in the data while one of them is accessed, so larger the block size lower the miss rate (but not good for miss penalty, would increase the miss penalty time as larger block size would require more time to load the block in data cache).

For the third run the cache size is reduced back to 128 bytes while the block size is increased to 16 bytes, in this case 4 words are placed in a block, and when a word is accessed the addresses nearby are placed in the block, the words not in the data cache end up in a miss while the adjacent ones when accessed would end up in a hit (as discussed in the assignment earlier). So, at the end it turns out that the miss rate is only 0.26 (i.e. 26%) only.

Conclusion:

The assignments were based on understanding, and analyzing the cache memory, by changing the configuration of cache and testing it on the programs. Once an instruction or data was mapped or loaded into the cache then it would immediately execute the instruction or load the data rather than doing it in several steps i.e. first fetching them from main memory and then executing or loading them from the cache memory. In the assignments it was observed that direct mapping would have high misses (miss rates) compared to the miss rates of the 2-way, 4-way, and fully associative mapping, however, for the block size of 4 bytes (i.e. block size of only word) all of the mappings ended up with hit rate of 0, because the programs/assignments were as such that the data loaded from the main memory was only accessed once. The hit rates would go up when the block size would increase, because, arrays were used and when a word from array was accessed, the addresses nearby to that word were mapped onto the data cache also, because of which the nearby words when accessed would result in a hit.
The instructions would also result in a miss or hit in instruction cache, the hits would only occur in the loop because of jumping back to the previous instruction which was already mapped onto instruction cache.