Computer Architecture - CS 301

Rishit Saiya - 180010027, Assignment - 11

November 21, 2020

1

In the Direct-Mapped Cache, we first try to find all the dependencies of variables and then check the cache.

We are given byte-addressable, direct-mapped cache of size 1 KB. So, that translates as follows:

$$Cache\,Size = 2^{10}\,bytes$$

Next, we are given information about Line Size (64 bytes) as follows:

$$Line\,Size = 2^{6}\,bytes$$

We know that Number of Address bits is 16. Now, to calculate the number of bits assigned to Index bits we do as follows:

$$Index = \frac{Blocks}{Cache\,Line}$$

or,

$$Index\,Bits = \frac{2^{10}}{2^{6}} = 2^4 = 16$$

So, Index will be of 4 bits.

So, finally tag bits become as follows:

$$TagBits = Address\,Bits - Index\,Bits - Offset$$

or,

$$Tag\,Bits = 16 - 4 - 6 = 6$$

We are the following sequence of accesses (16 bit addresses in hexadecimal format): 0x0001, 0x0002, 0x0003, 0x0041, 0x0081, 0x00A1, 0x00C1, 0x0401, 0x0402, 0x0001, 0x0002, 0x0003.

Table 1 shows the iterations on how the Cache is being filled in each value entered in iteration.

The details in each iterations are done with the considerations like the cache was empty before the 1^{st} iteration. We see that there were **6 Cache Hits** and **6 Cache Miss** after 12 iterations.

Address	**Binary**	**Tag Bits**	**Tag Field (Index Bits)**	**Tag Field (Bits)**	**Hit/Miss**
0x0001	0000 0000 0000 0001	000000	0000	000000	Miss
0x0002	0000 0000 0000 0010	000000	0000	000000	Hit
0x0003	0000 0000 0000 0011	000000	0000	000000	Hit
0x0041	0000 0000 0100 0001	000000	0001	000000	Miss
0x0081	0000 0000 1000 0001	000000	0010	000000	Miss
0x00A1	0000 0000 1010 0001	000000	0010	000000	Hit
0x00C1	0000 0000 1100 0001	000000	0011	000000	Miss
0x0401	0000 0100 0000 0001	000001	0000	000001	Miss
0x0402	0000 0000 1000 0001	000001	0000	000001	Hit
0x0001	0000 0000 0000 0001	000000	0000	000000	Miss
0x0002	0000 0000 0000 0010	000000	0000	000000	Hit
0x0003	0000 0000 0000 0011	000000	0000	000000	Hit

Table 1: Direct Mapped Cache

2

In the 2-way Associative Cache, we first try to find all the dependencies of variables and then check the cache.

We are given byte-addressable, direct-mapped cache of size 1 KB. So, that translates as follows:

$$Cache\,Size = 2^{10}\,bytes$$

Next, we are given information about Line Size (64 bytes) as follows:

$$Line\,Size = 2^{6}\,bytes$$

Number of Address bits is 16. Given that number of cache blocks per set is 2, we calculate the number of sets in the cache as follows:

$$Index = \frac{Blocks}{Cache\,Line \times cache\,blocks}$$

or,

$$Index\,Bits = \frac{2^{10}}{2^{6} \times 2} = 2^{3} = 8$$

So, Index will be of 3 bits.

So, finally tag bits become as follows:

$$TagBits = Address\,Bits - Index\,Bits - Offset$$

or,

$$Tag\,Bits = 16 - 3 - 6 = 7$$

Address	Binary	Tag Bits	Tag Field (Index Bits)	Tag Field (Bits)	Hit/Miss
0x0001	0000 0000 0000 0001	0000000	000	0000000	Miss
0x0002	0000 0000 0000 0010	0000000	000	0000000	Hit
0x0003	0000 0000 0000 0011	0000000	000	0000000	Hit
0x0041	0000 0000 0100 0001	0000000	001	0000000	Miss
0x0081	0000 0000 1000 0001	0000000	010	0000000	Miss
0x00A1	0000 0000 1010 0001	0000000	010	0000000	Hit
0x00C1	0000 0000 1100 0001	0000000	011	0000000	Miss
0x0401	0000 0100 0000 0001	0000010	000	0000001	Miss
0x0402	0000 0000 1000 0001	0000010	000	0000001	Hit
0x0001	0000 0000 0000 0001	0000000	000	0000000	Hit
0x0002	0000 0000 0000 0010	0000000	000	0000000	Hit
0x0003	0000 0000 0000 0011	0000000	000	0000000	Hit

Table 2: 2-way Associative Cache

We are the following sequence of accesses (16 bit addresses in hexadecimal format): 0x0001, 0x0002, 0x0003, 0x0041, 0x0081, 0x00A1, 0x00C1, 0x0401, 0x0402, 0x0001, 0x0002, 0x0003.

Table 2 shows the iterations on how the Cache is being filled in each value entered in iteration.

The details in each iterations are done with the considerations like the cache was empty before the 1^{st} iteration. We see that there were **7 Cache Hits** and **5 Cache Miss** after 12 iterations.

3

In the 4-way Associative Cache, we first try to find all the dependencies of variables and then check the cache.

We are given byte-addressable, direct-mapped cache of size 128 B. So, that translates as follows:

$$Cache\,Size = 2^7\,bytes$$

Next, we are given information about Line Size (16 bytes) as follows:

$$Line\,Size = 2^4\,bytes$$

Let the number of address bits be 8. Offset will be of 4 bits then. Given that number of cache blocks per set is 4, we calculate the number of sets in the cache as follows:

$$Index = \frac{Blocks}{Cache\,Line \times cache\,blocks}$$

Address (Binary)	Tag Value	Set Index Bit	Set Entry Bit	Hit/Miss
10010011	4	1	0	Miss
10010010	4	1	0	Hit
10010111	4	1	0	Hit
10010101	4	1	0	Hit
10110001	5	1	1	Miss
01010000	2	1	2	Miss
01011000	2	1	2	Hit
01110101	3	1	3	Miss
10110011	5	1	2	Hit
11010011	6	1	0	Miss
01110011	3	1	3	Hit

Table 3: LRU Replacement Policy

or,

$$Index\,Bits = \frac{2^7}{2^4 \times 4} = 2^3 = 2$$

So, Index will be of 1 bits.

So, finally tag bits become as follows:

$$TagBits = Address\,Bits - Index\,Bits - Offset$$

or,

$$Tag\,Bits = 8 - 1 - 4 = 3$$

3.1 LRU performing better than LFU

Table 3 shows where LRU is used as Replacement Policy.

Table 4 shows where LFU is used as Replacement Policy.

From Table 3 and Table 4 we can see that there are **6 hits** out of 11 cases & **5 hits** out of 11 cases for LRU & LFU respectively. So, LRU performs better than LFU.

3.2 LFU performing better than LRU

Table 5 shows where LRU is used as Replacement Policy.

Table 6 shows where LFU is used as Replacement Policy.

From Table 5 and Table 6 we can see that there are **5 hits** out of 11 cases & **6 hits** out of 11 cases for LRU & LFU respectively. So, LFU performs better than LRU.

Address (Binary)	Tag Value	Set Index Bit	Set Entry Bit	Hit/Miss
10010011	4	1	0	Miss
10010010	4	1	0	Hit
10010111	4	1	0	Hit
10010101	4	1	0	Hit
10110001	5	1	1	Miss
01010000	2	1	2	Miss
01011000	2	1	2	Hit
01110101	3	1	3	Miss
10110011	5	1	2	Hit
11010011	6	1	3	Miss
01110011	3	1	3	Miss

Table 4: LFU Replacement Policy

Address (Binary)	Tag Value	Set Index Bit	Set Entry Bit	Hit/Miss
10010011	4	1	0	Miss
10010010	4	1	0	Hit
10010111	4	1	0	Hit
10010101	4	1	0	Hit
01110001	3	1	1	Miss
10110000	5	1	2	Miss
01011000	2	1	2	Miss
01010101	2	1	3	Hit
10110011	5	1	2	Hit
01110011	6	1	0	Miss
10010101	4	1	1	Miss

Table 5: LRU Replacement Policy

Address (Binary)	Tag Value	Set Index Bit	Set Entry Bit	Hit/Miss
10010011	4	1	0	Miss
10010010	4	1	0	Hit
10010111	4	1	0	Hit
10010101	4	1	0	Hit
01110001	3	1	1	Miss
10110000	5	1	2	Miss
01011000	2	1	2	Miss
01010101	2	1	3	Hit
10110011	5	1	2	Hit
01110011	6	1	0	Miss
10010101	4	1	1	Hit

Table 6: LFU Replacement Policy

4

In the Fully Associative Cache, we first try to find all the dependencies of variables and then check the cache.

We are given byte-addressable, direct-mapped cache of size 128 B. So, that translates as follows:

$$Cache\,Size = 2^7\,bytes$$

Next, we are given information about Line Size (16 bytes) as follows:

$$Line\,Size = 2^4\,bytes$$

Let us say that number of Address bits is 8. Hence Offset will be of 4 bits. We calculate the number of sets in the cache as follows:

$$Index = \frac{Blocks}{Cache\,Line}$$

or,

$$Index\,Bits = \frac{2^7}{2^4} = 2^3 = 8$$

So, Index will be of 3 bits.

So, finally tag bits become as follows:

$$TagBits = Address\,Bits - Index\,Bits$$

or,

$$Tag\,Bits = 8 - 4 = 4$$

Address (Binary)	**Tag Value**	**Set Entry Bit**	**Hit/Miss**
01000010	4	0	Miss
01000100	4	0	Hit
01001000	4	0	Hit
01010000	5	1	Miss
00100001	2	2	Miss
00100011	2	2	Hit
00110011	3	3	Miss
01010100	5	1	Hit
01010001	5	1	Hit
01110011	6	4	Miss
00110101	3	3	Hit
00010001	1	5	Miss
00010000	1	5	Hit
01110000	7	6	Miss
01110001	7	6	Hit
01110101	7	6	Hit
10001000	8	7	Miss
01110101	6	4	Hit
10010101	9	0	Miss
10001011	8	7	Hit

Table 7: LRU Replacement Policy

Address (Binary)	Tag Value	Set Entry Bit	Hit/Miss
01000010	4	0	Miss
01000100	4	0	Hit
01001000	4	0	Hit
01010000	5	1	Miss
00100001	2	2	Miss
00100011	2	2	Hit
00110011	3	3	Miss
01010100	5	1	Hit
01010001	5	1	Hit
01110011	6	4	Miss
00110101	3	3	Hit
00010001	1	5	Miss
00010000	1	5	Hit
01110000	7	6	Miss
01110001	7	6	Hit
01110101	7	6	Hit
10001000	8	7	Miss
01110101	6	4	Hit
10010101	9	7	Miss
10001011	8	7	Miss

Table 8: LFU Replacement Policy

4.1 LRU performing better than LFU

Table 7 shows where LRU is used as Replacement Policy.

Table 8 shows where LFU is used as Replacement Policy.

From Table 7 and Table 8 we can see that there are **11 hits** out of 20 cases & **10 hits** out of 20 cases for LRU & LFU respectively. So, LRU performs better than LFU.

4.2 LFU performing better than LRU

Table 9 shows where LRU is used as Replacement Policy.

Table 10 shows where LFU is used as Replacement Policy.

From Table 9 and Table 10 we can see that there are **11 hits** out of 20 cases & **12 hits** out of 20 cases for LRU & LFU respectively. So, LFU performs better than LRU.

Address (Binary)	**Tag Value**	**Set Entry Bit**	**Hit/Miss**
01000010	4	0	Miss
01000100	4	0	Hit
01001000	4	0	Hit
01010000	5	1	Miss
00100001	2	2	Miss
00100011	2	2	Hit
01100011	3	3	Miss
01010100	5	1	Hit
01010001	5	1	Hit
01100001	6	4	Miss
01101001	6	4	Hit
01100011	3	3	Miss
00100101	1	5	Hit
00010001	1	5	Miss
01110000	7	6	Hit
01110000	7	6	Hit
10000001	8	7	Miss
01100101	6	4	Hit
10011000	9	7	Miss
01000101	4	0	Hit

Table 9: LRU Replacement Policy

Address (Binary)	Tag Value	Set Entry Bit	Hit/Miss
01000010	4	0	Miss
01000100	4	0	Hit
01001000	4	0	Hit
01010000	5	1	Miss
00100001	2	2	Miss
00100011	2	2	Hit
01100011	3	3	Miss
01010100	5	1	Hit
01010001	5	1	Hit
01100001	6	4	Miss
01101001	6	4	Hit
01100011	3	3	Hit
00100101	1	5	Hit
00010001	1	5	Miss
01110000	7	6	Hit
01110000	7	6	Miss
10000001	8	7	Hit
01100101	6	4	Hit
10011000	9	7	Miss
01000101	4	0	Hit

Table 10: LFU Replacement Policy

Address	Tag Value	Offset (Last Bit)	Cache Block	Hit/Miss
20	10	0	1	Miss
21	10	1	1	Hit
22	11	0	2	Miss
23	11	1	2	Hit
24	12	0	3	Miss
25	12	1	3	Hit
26	13	0	4	Miss
27	13	1	4	Hit
28	14	0	1 (LRU)	Miss
29	14	1	1	Hit
22	11	0	2	Hit
30	15	0	3	Miss
21	10	1	3	Miss
23	11	1	2	Hit
31	15	1	3	Hit

Table 11: Fully Associative Cache - LRU

5

Given that Cache size is 8 words/blocks and Line Size is given as 2. So, we calculate number of blocks/lines as follows:

$$Blocks = \frac{Cache\,Size}{Line\,Size}$$

or,

$$Blocks = \frac{8}{2} = 4$$

So, Offset is of 1 but and number of blocks of Cache is 4.

LRU (Least Recently Used):
Least Recently Used (LRU) discards the least recently used items first. This algorithm requires keeping track of what was used when, which is expensive if one wants to make sure the algorithm always discards the least recently used item.

Reference I took help from to solve this question: Reference 1, Reference 2.

Table 11 shows the iterations on how the Cache is being filled in each value entered in iteration.

The addresses are given to us. When we remove the last binary bit from the given addresses, we get the offset. Alternatively, just even address will have **0** a offset and odd address will have **1** as address. Since we are given that we have 2 lines, we put the data accordingly in the 4 boxes of caches. We see that at Address 28, Cache Line 1 (LRU) was assigned. It is to denote that LRU was implemented from there onward. We then decide

	Cache Data	
Cache Block	**[0] Index**	**[1] Index**
1	28	29
2	22	23
3	30	31
4	20	21

Table 12: Cache Contents - Fully Associative LRU

Hit/Miss by previous idea only.

We see the Hit Rate as follows:

$$Hit\ Rate = \frac{8}{15} = 0.53$$

Table 12 shows the Cache data after all the iterations.

6

Given that Cache size is 8 words/blocks and Line Size is given as 2 and each set is with 2 entries. So, we calculate number of blocks/lines as follows:

$$Blocks = \frac{Cache\,Size}{Line\,Size}$$

or,

$$Blocks = \frac{8}{2 \times 2} = 2$$

So, Offset is of 1 but and number of blocks of Cache is 4.

Table 13 shows the iterations on how the Cache is being filled in each value entered in iteration.

The addresses are given to us. When we remove the last binary bit from the given addresses, we get the offset. Alternatively, just even address will have **0** a offset and odd address will have **1** as address. Since we are given that we have 2 sets here, we put the data accordingly in the 2 sets in each line of caches. We see that at Address 28, Set Entry Bit 0 (LRU) was assigned. It is to denote that LRU was implemented from there onward. We then decide Hit/Miss by previous idea only.

We see the Hit Rate as follows:

$$Hit\ Rate = \frac{8}{15} = 0.53$$

Table 14 shows the Cache data after all the iterations.

Address	**Tag Value**	**Offset (Last Bit)**	**Set Entry Bit**	**Hit/Miss**
20	5	0	0	Miss
21	5	1	0	Hit
22	5	0	0	Miss
23	5	1	0	Hit
24	6	0	1	Miss
25	6	1	1	Hit
26	6	0	1	Miss
27	6	1	1	Hit
28	7	0	0 (LRU)	Miss
29	7	1	0	Hit
22	5	0	0	Hit
30	7	0	1	Miss
21	5	1	1	Miss
23	5	1	0	Hit
31	7	1	1	Hit

Table 13: 2-way Associative Cache - LRU

	Set 0		**Set 1**	
Cache Block	[0] **Index**	[1] **Index**	[0] **Index**	[1] **Index**
1	28	29	20	21
2	22	23	30	31

Table 14: Cache Contents - 2 Way Associative LRU