Solution to Self-evaluation Exercise on Caches/memory system/TLB

Q1. Consider a fully associative cache following the LRU replacement scheme and consisting of only 8 words. Consider the following sequence of memory accesses (the numbers denote the word address):
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 22, 30, 21, 23, 31

Assume that we begin when the cache is empty. What are the contents of the cache after the end of the sequence of memory accesses?

Ans. 28, 29, 22, 30, 21, 23, 31, 27 at locations 0 to 7 respectively (please verify yourself).

Q2. Answer previous question assuming a FIFO replacement scheme.

Ans. 28, 29, 30, 21, 23, 31, 26, 27 at locations 0 to 7 respectively

Q3. What is the total size (in bytes) of a direct mapped cache with the following configuration in a 32 bit system? It has a 10 bit index, and a data-block size of 64 bytes. Each block has 1 valid bit and 1 dirty bit.

(You have to show the size of **data + tag + valid bit + dirty bit**).

Ans. 6 bits for offset, 10 bits for index) 16 bits for tag. It is a direct mapped cache, which implies there are 2^10 entries in all.
Size of each entry (in bits) = 16(tag of virtual address) + [64(data)]* 8 + 2

From this, the total size of cache can be found out.

Q4. Consider a processor with 16-bit virtual and physical addresses. This processor uses a virtual to physical address translation scheme using a page table (even though both virtual and physical address spaces are of same size). The cache in the system has the following characteristics: 8 sets with 256B block size, 4-way associativity. Also, the cache is virtually-indexed, physically-tagged. It is given that binary representation of address 0xCDAB is 1100 1101 1010 1011.

Cache is virtually-indexed, which means the set-index is determined from the virtual address and not physical address. Since cache block size is 256B, 8 bits are required for block offset. Cache has 8 sets, so three bits are required for finding set-index. Thus, starting the counting from rightmost bit, VirtualAddress[10,9,8] will be used to find out the set-index (assuming counting starts from zero).
a) Given that the page size is **512-byte**. In the diagram below, please mark with crosses (X), all the locations where **a virtual address** 0xCDAB can be stored.

Virtual address = 1100 1101 1010 1011
⇨ Set-index = $(101)_2$ =$(5)_{10}$.
Since associativity is 4, the address can reside in any of the 4 ways, hence, its location is shown below:

Set Index	Way 0	Way 1	Way 2	Way 3
0				
1				
2				
3				
4				
5	X	X	X	X
6				
7				

(b) Given that the page size is **512-byte**. In the diagram below, please mark with crosses (X), all the locations where **a physical address** 0xCDAB can be stored.

Given Physical address: 1100 1101 1010 1011

Corresponding Virtual address: xxxx xxx1 1010 1011

⇨ Set-index = $(xx1)_2$

Thus, the address can be stored at any of the four sets and in any of the four ways in those sets.

Set Index	Way 0	Way 1	Way 2	Way 3
0				
1	X	X	X	X
2				
3	X	X	X	X
4				
5	X	X	X	X
6				
7	X	X	X	X

c) Now, assume that the page size is **2048-byte**. In the diagram below, please mark with crosses (X), all the locations where **a physical address** 0xCDAB can be stored.

Here the page size is 2048 bytes. So, lower 11 (=log2 2048) bits become part of the page offset. Hence, they do not go through virtual-to-physical address translation. In part (b) of the question, page size was 512 bytes, so only lower 9 bits become part of the page offset.
For part (c),
Given Physical address: 1100 1101 1010 1011
Corresponding Virtual address: xxxx x101 1010 1011
Set-index = $(101)_2$

Set Index	Way 0	Way 1	Way 2	Way 3
0				
1				
2				
3				
4				
5	X	X	X	X
6				
7				

Q5. Consider the data structures in two cases:

Case 1:

```
int RollNumber[SIZE];
int ExamMarks[SIZE];
```

Case 2:

```
struct StudentRecord {
int RollNumber_;
int ExamMarks_;
};
struct StudentRecord merged_array[SIZE];
```

Assume that a program reads the roll numbers of students from a file in the following format:

RollNumber Marks

101 45

102 56

....

Which of the data structures (case 1 or 2) will be more cache-friendly and why?

Ans. Case 2 will lead to more friendly data structure since roll number and marks are accessed together, so it is better to put them nearby in physical memory so they can be fetched in the same cache block.

Q6. For each of these statements, write whether the statements are true or false. Also write one line explanation/reasoning.

Doubling the line size (while keeping total cache capacity same) halves the number of tags in the cache

TRUE. Since cache size is unchanged, the line size doubles, the number of tag entries is halved.

Doubling the associativity (while keeping total cache capacity same) doubles the number of tags in the cache.

FALSE. The total number of lines across all sets is still the same, therefore the number of tags in the cache remain the same.

Doubling cache capacity of a direct-mapped cache usually reduces conflict misses.

TRUE. Doubling the capacity increases the number of lines from N to 2N. Address i and address i+N now map to different entries in the cache and hence, conflicts are reduced.

Doubling cache capacity of a direct-mapped cache usually reduces compulsory misses

FALSE. The number of lines doubles but the line size remains the same. So the compulsory “cold-start” misses stays the same.

Doubling the line size usually reduces compulsory misses.

TRUE. Doubling the line size causes more data to be pulled into the cache on a miss. This exploits spatial locality as subsequent loads to different words in the same cache line will hit in the cache reducing compulsory misses.