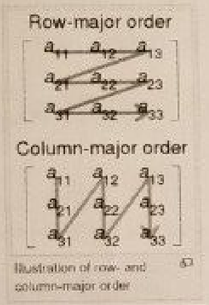
Homework #3-1

Problem 1

Assume memory is byte addressable and words are 64 bits, unless specified otherwise.

The following reference from Wikipedia may be useful in answering this question:


Row-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}
Column-major order
Illustration of row- and column-major order

Row-major order, e.g., C (0-indexed)			Column-major order, e.g., Fortran (1-indexed)		
Address	Access	Value	Address	Access	Value
0	`A[0][0]`	a_{11}	1	`A(1,1)`	a_{11}
1	`A[0][1]`	a_{12}	2	`A(2,1)`	a_{21}
2	`A[0][2]`	a_{13}	3	`A(1,2)`	a_{12}
3	`A[1][0]`	a_{21}	4	`A(2,2)`	a_{22}
4	`A[1][1]`	a_{22}	5	`A(1,3)`	a_{13}
5	`A[1][2]`	a_{23}	6	`A(2,3)`	a_{23}

Matlab is also column-major ordered.

Ref: "Row- and column-major order," *Wikipedia*. 25-Mar-2018.

5.1 In this exercise we look at memory locality properties of matrix computation. The following code is written in C, where elements within the same row are stored contiguously. Assume each word is a 64-bit integer.

```
for (I=0; I<8; I++)
  for (J=0; J<8000; J++)
    A[I][J]=B[I][0]+A[J][I];
```

# of bits in 16 byte cache line = 16 × 8 = 128
of 64 bit integer in 16 byte cache line = 128 / 64 = 2

5.1.1 How many 64-bit integers can be stored in a 16-byte cache block? 2.

5.1.2 Which variable references exhibit temporal locality?

i and j are constantly accessed. ~~Because of~~ These variables are likely to stay in cache and hence exhibit temporal locality. B[i][0] exhibit temporal locality since it is accessed over and over in a row.
∴ i, j, B[i][0]

5.1.3 Which variable references exhibit spatial locality?

A[i][j]. Multiple of them are loaded into cache on a single miss. As i is incremented, nearby values of array are accessed.

Locality is affected by both the reference order and data layout. The same computation can also be written below in Matlab, which differs from C in that it stores matrix elements within the same column contiguously in memory.

```
for I=1:8
  for J=1:8000
   A(I,J)=B(I,0)+A(J,I);
  end
end
```

→ i and j are accessed every iteration of the loop. It will stay in cache because they are exploiting temporal locality.

5.1.4 Which variable references exhibit temporal locality? B[i][0], i, j

5.1.5 Which variable references exhibit spatial locality? A(j, i), B[i][0]

5.1.6 How many 16-byte cache blocks are needed to store all 64-bit matrix elements being referenced using Matlab's matrix storage?

1st. A[i, j] for values of i ranging between 0 and 7999 and

2nd values of j ranging between 0 and 7.

There are 8000 × 8 = 64000

64000 × 2 = 128000

! There is a rectangle of 8 × 8 = 64 element that is common between both groups of elements and should be subtracted.

there are 8 elements being accessed from array B. Total number of elements

128000 − 64 = 127936

~~127976 + 8 = 127944~~ 127936 + 8 = 127944 //

Since each cache line can store 4 elements, we would need 127944 / 4 = 31986 cache lines to store the entire matrix.

Problem 2

5.5 For a direct-mapped cache design with a 64-bit address, the following bits of the address are used to access the cache.

→ how many blocks are present in a directly mapped cache

Tag	Index	Offset
63–10	9–5	4–0

5.5.1 What is the cache block size (in words)? 0–4 offset, $2^5 = 32$, ∴ Each cache block has four 8-byte words

5.5.2 How many blocks does the cache have? 9-5 Index, $2^5 = 32$

5.5.3 What is the ratio between total bits required for such a cache implementation over the data storage bits? 54 bits for each block tag field
With (32) blocks, $54 \times 32/8 = 216$ bytes for tag field.
Total bits for the cache $= 2^{10} + 216 = 1240$ Bytes.

$$\frac{2^{10}+216 = 1240}{2^{10} = 1024} = 1.21$$

Beginning from power on, the following byte-addressed cache references are recorded.

Mandatory Miss first time referenced

Same Index different tag as 0x04

Same Index as 4th frame

same index different tag as 0x1E

Index and tag are same as 0x00

Index and tag are same as 0x04

Address												
Hex	00	04	10	84	E8	A0	400	1E	8C	C1C	B4	884
Dec	0	4	16	132	232	160	1024	30	140	3100	180	2180

M H H M M M M M H M H M → the frame needs replacement

Mandatory Miss

Same index diff tag as 0x400

same index as 0x A0

5.5.4 For each reference, list:

1. Its tag and offset;
2. Whether it is a hit or a miss;
3. Which bytes were replaced (if any).

5.5.5 What is the hit ratio? $\frac{4}{12} = 0.33 \times 100 = 33\%$

00	… 0000 0000 0000 0000
04	… 0000 0000 0000 0100
10	… 0000 0000 0001 0000
84	… 0000 0000 1000 0100
E8	… 0000 0000 1110 1000
A0	… 0000 0000 1010 0000
400	… 0000 0100 0000 0000
1E	… 0000 0000 0001 1110
8C	… 0000 0000 1000 1100
C1C	… 0000 1100 0001 1100
B4	… 0000 0000 1011 0100
884	… 0000 1000 1000 0100

Problem 3

5.6 Recall that we have two write policies and two write allocation policies, and their combinations can be implemented either in L1 or L2 cache. Assume the following choices for L1 and L2 caches:

L1	L2
Write through, non-write allocate	Write back, write allocate

5.6.1 Buffers are employed between different levels of memory hierarchy to reduce access latency. For this given configuration, list the possible buffers needed between L1 and L2 caches, as well as L2 cache and memory. L1 => write

5.6.2 Describe the procedure of handling an L1 write-miss, considering the components involved and the possibility of replacing a dirty block.

5.6.3 For a multilevel exclusive cache configuration (a block can only reside in one of the L1 and L2 caches), describe the procedures of handling an L1 write-miss and an L1 read-miss, considering the components involved and the possibility of replacing a dirty block.

5.6.1. Between L1 and L2 caches, one write buffer is required. When the miss occurs, we directly update the portion of the block into the buffer, which will be waiting to be written into L2 cache. While the process doesn't need to stall if the buffer is not full.

Between L2 cache and the memory, we require write and store buffers. When we have a cache miss, we must first write the block back to memory if the data in the cache is modified. In this situation, a write buffer is required to hold that data, such that the processor can continue the execution while that data is waiting to be written to the memory. In the meanwhile, a store buffer is used, such that the processor places the new data in the store buffer. Then [illegible] when a cache hit occurs, this new data is written from the store buffer into the cache.

5.6.2. First we check whether the block is dirty. If it is, then we write the dirty block to memory. Next, we retrieve the target block from memory (overwriting the block that is in our way). Finally we write to our L2 block.