CprE 381 Homework 9

[Note: This homework is meant to help you form insights regarding the principles behind caches.]

1. Principle of Locality
    a. Write a valid MIPS assembly program that executes at least 20 instructions and demonstrates spatial locality in instruction fetching, but not data accesses. Explain this locality in the assembly comments.
    Answers vary, but see prob2a.s for a sample.

    b. Write a valid MIPS assembly program that executes at least 20 instructions and demonstrates temporal locality in data accesses, but not instruction fetching. Explain this locality in the assembly comments.
    Answers vary, but see prob2b.s for a sample.

    c. Spend some time looking at open-source programs on Github.com. Find a piece of a C or C++ program on github that appears to display a significant amount of data locality. Provide the html browsable file URL and line numbers of the example. Justify why these lines demonstrate data locality. *[Note that since this is real code, you may need to reference multiple files to demonstrate locality even in a single example.]*
    Answers vary widely.

2. Breaking Locality
    Complete the following C function that scales each element of a 2D array by a scalar. First, complete it in row-major ordering (https://en.wikipedia.org/wiki/Row-_and_column-major_order). Second, complete it in column-major ordering. Which is faster on your computer (report the time each takes to execute 1000 calls to scale and what processor model you have)? Why is it faster? Please use the C template provided with this homework.

```
void scale(int n, int m, int array[n][m], int scale);
```

    Solution code provided on assignment page. The column-major implementation about 70% longer to run on my system (4.5s vs 2.65s). Since both implementations have the same set of accesses and should have the same number of instructions executed, the order of memory access is a likely source of the difference.

    C stores 2D arrays in row-major order where elements within a row are stored in adjacent memory and then rows themselves are stored in adjacent memory. So, array[i][j] and array[i][j+1] are directly adjacent to each other in memory while array[i][j] and array[i+1][j] are m (row length) elements apart from each other in memory. When iterating through the array in row-major ordering, only the first access to a cache block misses (it will always miss since the array sizes are much larger than the L1 cache – this cache actually can't leverage the temporal locality from performing many iterations of scale) and then the other memory locations within the cache block are accessed next.

When iterating through the array in column-major ordering, each access is to a different row which will guarantee that each access will be mapped into a different set in a cache since the row size is much larger than block size. Since the number of rows within the array is likely more than the number of physical cache frames/blocks within the cache (on my computer there are 512 such frames verses 1000 rows), there is never another access to a cache block before it is evicted. The result is that all array access miss in the cache.

In summary, the row-major ordering misses many fewer times in the cache and, therefore, incurs fewer CPU stall cycles than the column-major ordering. Note that this does show the possibly very large impact nuances of hardware can have on software performance. This is despite significantly powerful optimizing compilers and expensive hardware techniques (e.g., out-of-order processors and memory prefetching).

3. Direct-Mapped Cache Configuration and Simulation
    a. ZyBooks 5.19.2.a

There are 12 32 bit word addresses provided.
3 in binary is 0000 0000 0000 0000 0000 0000 0000 0011
180 in binary is 0000 0000 0000 0000 0000 0000 1011 0100
And so on

Notice that only last 8 bits matter as all the addresses are < 256, so let's represent the last 8 bits for binary representation.

Direct mapped cache with 16 one-word blocks implies:

- No offset (since one word block, 2^0 = 1)
- Need last 4 bits for index (2^4 = 16)
- Remaining are tag bits

So for address 3, tag = 0000; index = 0011
For address 180, tag = 1011; index = 0100
And so on

So, 3 gets stored at 3rd index in cache and 180 at 4th index in cache.

Cache (one word)

0	

1	
2	
3	3rd address value
4	4th address value
…	…
15	

Since cache was initially empty for both 3 and 180, it was a miss, so the value (one word at 3rd and 180th address in memory) was fetched and loaded in cache.

Now, we can proceed with answering the question.

Word Address	Binary Address	Tag	Index	Hit/Miss
3	0000 0011	0	3	M
180	1011 0100	11	4	M
43	0010 1011	2	11	M
2	0000 0010	0	2	M
191	1011 1111	11	15	M
88	0101 1000	5	8	M
190	1011 1110	11	14	M
14	0000 1110	0	14	M
181	1011 0101	11	5	M
44	0010 1100	2	12	M
186	1011 1010	11	10	M
253	1111 1101	15	13	M

b. ZyBooks 5.19.2.b

Direct mapped cache, 2 word blocks, 8 blocks, implies:

- 1 bit for offset (2^1 = 2 word block)
- 3 bits for index (2^3 = 8 blocks)
- Rest are tag bits

So for 3, Tag = 0000, Index = 001, Offset = 1

For 180, Tag = 1011, Index = 010, Offset = 0
Offset shows where in block that value will lie. Also, if a cache miss occurs, cache fetches 2 (consecutive) words form memory. For instance, when 3 was a miss, both 2 and 3 were fetched (since 2 word blocks) and loaded in cache. This is good for spatial locality.

CACHE

	1 word (0)	1 word (1)
0		
1	2nd address value	3rd address value

2	180th address value	181 address value
…	…	…
7		

So notice if we try to access 2nd address, it'll be a hit since it was loaded in cache when 3 was loaded when it was a miss.

Word Address	Tag	Index	Offset	Hit/Miss
3	0000	001	1	M
180	1011	010	0	M
43	0010	101	1	M
2	0000	001	0	H
191	1011	111	1	M
88	0101	100	0	M
190	1011	111	0	H
14	0000	111	0	M
181	1011	010	1	H
44	0010	110	0	M
186	1011	101	0	M
253	1111	110	1	M

Hit occurs if both Tag and index match, offset is not checked for hit/miss

c. ZyBooks 5.19.2.c

All caches are direct mapped with total 8 words
Number of blocks in the 3 caches will be different though
Here's the analysis:

C1 (1 word blocks):
- 8 blocks (8/1 = 8)
- Index = 3 bits (2^3 = 8 blocks)
- Offset = 0 bits (2^0 = 1 word)

C2 (2 word blocks):
- 4 blocks (8/2 = 4)
- Index = 2 bits (2^2 = 4 blocks)
- Offset = 1 bits (2^1 = 2 word)

C3 (4 word blocks):
- 2 blocks (8/4 = 2)
- Index = 1 bit (2^1 = 2 blocks)
- Offset = 2 bits (2^2 = 4 word)

addr	binary	Tag	C1 Index	C1 Offset	C1 H/M	C2 Index	C2 Offset	C2 H/M	C3 Index	C3 Offset	C3 H/M
3	0000 0011	00000	011	-	M	01	1	M	0	11	M
180	1011 0100	10110	100	-	M	10	0	M	1	00	M
43	0010 1011	00101	011	-	M	01	1	M	0	11	M
2	0000 0010	00000	010	-	M	01	0	M	0	10	M
191	1011 1111	10111	111	-	M	11	1	M	1	11	M
88	0101 1000	01011	000	-	M	00	0	M	0	00	M
190	1011 1110	10111	110	-	M	11	0	H	1	10	H
14	0000 1110	00001	110	-	M	11	0	M	1	10	M
181	1011 0101	10110	101	-	M	10	1	H	1	01	M
44	0010 1100	00101	100	-	M	10	0	M	1	00	M
186	1011 1010	10111	010	-	M	01	0	M	0	10	M
253	1111 1101	11111	101	-	M	10	1	M	1	01	M

Now miss rates = No. of misses/ No. of accesses * 100

C1 = 12/12 *100 = 100%

C2 = 10/12 *100 = 83.33%

C3 = 11/12 *100 = 91.66%

Cache total cycles = miss stall time * no. of misses + no. of access * access time

C1 = 25 * 12 + 12 * 2 = 324

C2 = 25 * 10 + 12 * 3 = 286

C1 = 25 * 11 + 12 * 5 = 335

C2 has best performance.

d. (From P&H 5th Edition) There are many different design parameters that are important to a cache's overall performance. Below are listed parameters for different direct-mapped cache designs.
Cache Data Size: 32 KiB
Cache Block Size: 2 words
Cache Access Time: 1 cycle

The formula shown in Section 5.3 shows the typical method to index a direct-mapped cache, specifically (Block address) modulo (Number of blocks in the cache). Assuming a 32-bit address and 1024 blocks in the cache, consider a different indexing function, specifically (Block address[31 :27] XOR Block address[26:22]). Is it possible to use this to index a direct-mapped cache? If so, explain why and discuss any changes that might need to be made to the cache. If it is not possible, explain why.

It's possible but we'll need more tag bits. You would need to have some of the index bits as the tag. Consider the following:

1000 0100 0000 … 0011 index = 10000 XOR 10000 = 00000
0000 0000 0000 … 0011 index = 00000 XOR 00000 = 00000

Since [31:27] and [26:22] are used in XOR their info is lost, so using [21:0] will cause a problem. Hence we should at least use [26:0] of the address. (Using the full address as the tag would work, but would be unnecessary since for a given [26:22] assignment, a difference in [31:27] would mean the word has a different index)