CSCI 2021 Machine Architecture and Organization, Fall 2018, Written Assignment #4

Problem 1 (35 points)

Consider the following matrix transpose routine:

```
typedef int array[4][4];
void transpose (array dst, array src) {
    int i, j;
    for(i = 0; i < 4; i++) {
        for(j = 0; j < 4; j++) {
            dst[i][j] = src[j][i];
        }
    }
}
```

Assume that this code runs on a machine with the following properties:

- The `int` type is 4 bytes long
- The `src` array starts at address 0x00 and the `dst` array starts at 0x50
- There is a single L1 data cache that is direct-mapped, write-through, write-allocate, with a block size of 8 bytes.
- The cache has 8 data lines, for a total of 64 data bytes, and the cache is initially empty
- Accesses to the `src` and `dst` arrays are the only sources of read and write misses For each row and column, indicate whether the access to `src[row][col]` and `dst[row][col]` is a hit (h) or a miss (m). For example, reading `src[0][0]` is a miss and writing `dst[0][0]` is also a miss.

dst	Col 0	Col 1	Col 2	Col 3
Row 0	M	H	M	H
Row 1	M	H	M	H
Row 2	M	H	M	H
Row 3	M	H	M	H

src	Col 0	Col 1	Col 2	Col 3
Row 0	M	M	M	M
Row 1	M	M	M	M
Row 2	M	M	M	M
Row 3	M	M	M	M

What is the cache miss rate for this function?

Miss rate = Total # of misses / Total # of accesses

Total # of misses between dst and src = 24
Total # of accesses between dst and src = 32
Miss rate = 75%

Problem 2 (25 points)

For this question, we consider the memory system of a small embedded processor. The size of the physical address space is 4K bytes, and the memory is byte-addressable. The single-level cache is 3-way associative, with a 2-byte block size and 24 total lines.

In the following tables, all numbers are given in hexadecimal. The content of the cache is as follows (V = Valid, B0 = Byte 0, B1 = Byte 1):

3-way set associative cache

Index	Tag	V	B0	B1	Tag	V	B0	B1	Tag	V	B0	B1
0	27	0	C7	B1	C8	1	86	DE	21	0	E3	E1
1	A6	1	FA	DD	53	0	AD	0F	7B	1	A1	47
2	FC	1	D9	8D	0B	0	B2	39	FD	1	6A	AC
3	·1E	0	1E	62	36	1	8F	B5	1A	1	D7	92
4	F1	1	BE	BF	CE	0	D5	21	A9	1	19	95
5	49	0	55	11	7A	1	C4	16	66	0	18	2F
6	8B	1	F2	BD	1C	0	14	1D	FD	1	01	97
7	D1	1	05	39	8F	1	BA	E7	38	1	D1	B8

A. Please indicate (by labeling the following diagram) the bits in the physical memory address that would be used to determine the following (ignore extra unused fields): CO the cache offset, CI the cache set, CT the cache tag.

CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO

B. For physical address **0xA64**, indicate the cache entry accessed and the cache byte value returned in hexadecimal. Indicate whether a cache miss occurs. If there is a cache miss,

enter “unknown” for “Cache Byte returned.” First, write the physical address in the same format as above, putting one bit per box and ignoring unused boxes.

1	0	1	0	0	1	1	0	0	1	0	0

Then, compute the following parameters of the cache access:

Cache Offset (CO)	0
Cache Index (CI)	010
Cache Tag (CT)	10100110 (A6)
Cache Hit? (Y/N)	Y
Cache Byte Returned	FA

C. Repeat part B, with the physical address **0x367**.

0	0	1	1	0	1	1	0	0	1	1	1

Cache Offset (CO)	1
Cache Index (CI)	011
Cache Tag (CT)	00110110 (36)
Cache Hit? (Y/N)	Y
Cache Byte Returned	B5

D. What is the total size of this cache in bits, including the data bits, and the space used to store tags and valid bits?

600 bits

E. If we can improve the cache hit rate of a program from 95% to 97% on a cache with a hit time of 15 cycles and a miss penalty of 300 cycles, what is the percentage of improvement in its average memory access time?

20% improvement

Problem 3 (20 points)

The following table gives the parameters for a number of different caches, where *m* is the number of physical address bits, *C* is the cache size (number of data bytes that the cache can store), *B* is the block size in bytes, *E* is the number of lines per set, *S* is the number of cache sets, *t* is the number of tag bits, *s* is the number of set index bits, and *b* is the number of block offset bits. For each cache, use what is given to fill out the rest of the row.

Cache	*m*	*C*	*B*	*E*	*S*	*t*	*s*	*b*
1	16	512	4	8	16	10	4	2
2	16	512	2	4	64	9	6	1
3	32	1024	16	4	16	24	4	4
4	32	1024	16	16	4	26	2	4
5	32	2048	16	8	16	24	4	4

Problem 4 (20 points)

Given the code below, describe two optimizations that could be used to improve the performance of the function `func`. These optimizations may relate to cache performance discussed in Chapter 6 or any of the optimizations discussed in Chapter 5 of the textbook. For each optimization, either provide an example of the code that would implement the optimization, or describe how to do it in sufficient detail that it could be easily implemented. In addition, briefly explain why the change improves the program's performance.

```
int f(int x, int y) {
return x * x + y * y;
}
void func(int mtx[N][N], int* res) {
   *res = 0;
```

```
    for(int i = 0; i < N; i++) {
      for(int j = 0; j < N; j++) {
         for(int k = 0; k < f(i, j); k++) {
             *res = *res + f(mtx[j][i], i + j) + f(mtx[j][i], k);
          }
       }
     }
}
```

1. Change matrix operation from column major to row major in line 9

   Change:
   (*res = *res + f(mtx[i][j], i + j) + f(mtx[i][j], k);

   By looping over the columns then the rows, the matrix mtx[i][j] has better cache performance since there will be more hits than there will be misses. This is because C arrays are stored in row major order.

2. Utilize loop unrolling on the inner most loop in order to reduce iterations. Conditions must be met to use this method prior to looping. Specifically, if j is odd and i is even, or j is even and i is odd, then, f(i, j) would return an odd number, thus k cannot be incremented by 2. The only cases that can be used are when i and j are either both odd or both even, since (even*even) + (even*even) = even and (odd*odd) + (odd*odd) = even

   Change:

```
if ((j % 2 != 1and i % 2 != 0) or (i % 2 != 1and j % 2 != 0)){
   for(int k = 0; k < f(i, j); k += 2) {
      *res = *res + f(mtx[j][i], i + j) + f(mtx[j][i], k);
      *res = *res + f(mtx[j][i], i + j) + f(mtx[j][i], k + 1);
   }
} else{
   for(int k = 0; k < f(i, j); k ++) {
      *res = *res + f(mtx[j][i], i + j) + f(mtx[j][i], k);
   }
}
```

   By using the method of loop unrolling, statements in the loop are executed in parallel, thus limiting the number of iterations which use overhead such as loop control instructions and loop test instructions. By doing this, we increase the program speed while sacrificing some memory space.