Homework 3

Memory and Cache

1. (a) Describe the layout of the following struct on an x64 machine running Linux. Your description should include its padding, and indicate how many bytes are used and how many bytes are wasted for padding.
(b) Reorder the fields in the struct so that it will consume the most space. Describe its layout. How many bytes is the reordered struct?
(c) Reorder the fields of the struct so that it will consume the least space. Describe its layout. How many bytes is the reordered struct?

```
struct foo {
  short  a;
  long   *b;
  short  c;
  double d;
  int    *e;
  char   f;
};
```

2. Consider a processor that uses m=32-bit addresses and can address 2^{32}= 4 Gigabytes of memory. Suppose that it has one level of cache. As in Figure 6.25 of your textbook, the address is split into a *t* bit tag, an *s* bit set index, and a *b* bit block offset. The cache consists of 16,384 bytes, with a block size of 128 bytes. Answer each of the following for direct-mapped, 8-way set associative, and fully associative versions of the cache.

    a. How many cache lines are there?
    b. What is *b*?
    c. What is *s*?
    d. What is *t*?

3. For the cache in problem 2, draw the cache given that it is structured as follows. Please elide replicated components, but annotate your drawing with how many components there are.

    a. Direct-mapped
    b. 8-way set associative
    c. Fully associative

4. Our company wants to optimize the performance of the following code

```
void vector_add(int n, int *a, int *b, int *c) {
    int i;
    for (i=0;i<n;i++) {
      c[i]=a[i]+b[i];
    }
}
```

run on the same processor and cache as described in problem 2. The cache is write-back, write-allocate, and has an LRU replacement policy. Integers are 32 bits.

a. **Suppose the cache is direct mapped**. Let `n`=4096, `a`=0xa0000, `b`=0xc0000, `c`=0xe0000. On average, how many times per loop iteration will you load a cache block from main memory? How many times per loop iteration will you flush a cache block back to main memory?
b. While we're all fired up to buy ultra-cool mega-associative cache hardware (which comes only in machined aluminum), a smart alec programmer claims that we can get the same effect by having `a`=0xa0100, `b`=0xc0200, and `c`=0xe0300. Is she right? Why or why not?

5. You should now be coming to realize just how important leveraging locality of access is for performance. Modern processors take advantage of this locality in hardware using many mechanisms, but one important one is called *prefetching*. The prefetch unit of a CPU will monitor memory accesses and try to predict accesses that might follow soon after. That way, when the CPU issues a load or store to the *next* address, the cache will have already been populated with it, and it will not need to wait for an expensive access to main memory to complete. However, hardware prefetch units have limited foresight, and sometimes it may be better for the programmer to provide some hints as to what memory will be accessed in the near future. Consider the following piece of code that simply sums up a *large* 2D array.

```
#define ROWS (1024*1024*16)
#define COLS 16

int 2darray_sum() {
      int sum = 0;
       int i, j;
      int * a = malloc(sizeof(int)*ROWS*COLS);
      /* skip array initialization code */
      for (i = 0; i < ROWS; i++) {
            for (j = 0; j < COLS; j++) {
                  sum += *(a+i*COLS+j);
            }
            _prefetch(a+(i+1)*COLS);
      }
      return sum;
}
```

This code loops through a 2D-array using pointer arithmetic and sums the elements. The interesting thing we've added here is the call to `_prefetch`. `_prefetch` takes a memory address and instructs the hardware to preload the cache with *one* line starting at that address. For the following question, assume that our data cache size is 32KB and the block size is 64 bytes.

a. How big is the array that `a` points to? Will it fit in the cache?
b. Why did we choose to prefetch at the address `a+(i+1)*COLS` and why did we place it in the outer loop? Explain your reasoning. (Hint: it's no coincidence that `COLS` is 16)
c. Running the above code on our class machine *with* the prefetch produces a speedup over the case *without* the prefetch by about 56%. Suppose we make this same comparison, but we make it for various array sizes. When would you expect the prefetch to help us more, with much smaller arrays or with larger arrays? Why?

6. Caches form a hierarchy, with slower, bigger caches operating below faster, smaller caches. At the top of the hierarchy, the L1 cache is often split into two caches, one for data, and one for instructions. When the CPU fetches an instruction, it first looks in the L1 instruction cache. If it misses there, the request proceeds to the L2 cache, which is shared among instructions and data, and then on to the L3 shared cache, and so on.

a. The L1 data cache works equally well for reads and writes, but the L1 instruction cache is *highly centered around reads.* Why? What characteristic of fetching and executing instructions might lead to this kind of design?

b. It is possible to build *self-modifying code*, where the program actually overwrites itself as it runs. Some rare parts of the operating system and application support libraries use this capability. A common reaction of an L1 instruction cache design when it sees a write to memory it is caching is to flush and clear the whole cache, not just the line that contains the write, and let the write proceed to the L2 cache. Why might this be considered a reasonable design?

c. Why are only some caches unified? Why don't we split some of the unified L2 into an instruction-L2 (L2i) and data-L2 (L2d)? Or why not add more cache so we can have the non-unified cache without needing to split the unified one? This question also holds for L3.