Homework 3

Memory and Cache

1. *(This is a repeat of the additional question given for HW2 in preparation for the midterm).* Reorder the fields in this structure so that the structure will (a) consume the most space and (b) consume the least space on an IA32 machine on Linux.

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

2. Consider a processor that uses 20-bit addresses and can address 2^{20}=1M bytes 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 8192 bytes, with a block size of 128 bytes. Answer each of the following for direct-mapped, 4-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 3, draw the cache given that it is structured as follows. You can elide replicated components, but annotate your drawing with how many components there are.

   a. Direct-mapped
   b. 2-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 3. 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`=0x0a000, `b`=0x0c000, `c`=0x0e000. 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`=0x0a000, `b`=0x0c080, and `c`=0x0e100. 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?