Homework 2

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;
  int    b;
  long   *c;
  char   d;
  double e;
  int    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 32,768 bytes, with a block size of 64 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. Running accesses through a tiny cache. A tiny machine uses m=8 bit addresses. It has a tiny 16 byte data cache consisting of 4 lines and a block size of 4 bytes. The cache starts empty and uses LRU replacement.

(a) Consider three different cache designs below. For each design below, fill in the number of sets (S), lines per set (E), and the bit counts b, s, and t. (Hint: write an 8-bit address, mark the t, s, and b bits, and check t+s+b= 8.)

Design	S	E	b	s	t
Direct-mapped					
2-way set associative					
Fully associative					

(b) A program accesses the following memory addresses (in order).

Accesses: 0x40, 0x50, 0x42, 0x48, 0x53, 0x44, 0x4C, 0x41

Step through the memory accesses for each cache design and fill in the tables below (first row is done). For each access, write:

- the tag (in hex)
- the set number (not needed for fully associative)
- if the access is a cache hit or miss
- the cache contents after the access (write "—" for empty lines). For the 2-way and fully associative caches, list tags from least recently used to most recently used.
- for misses, the miss type: compulsory (first access to that block), conflict (the fully associative cache would have hit), or capacity (any other miss)

Direct-mapped

#	Address	Tag	Set #	Hit/Miss	Set 0	Set 1	Set 2	Set 3	Miss Type
1	0x40	0x04	0	Miss	0x04	—	—	—	Compulsory (cold)
2	0x50								
3	0x42								

4	0x48								
5	0x53								
6	0x44								
7	0x4C								
8	0x41								

2-way set associative

#	Address	Tag	Set	Hit/Miss	Set 0 Contents	Set 1 Contents	Miss Type
1	0x40	0x08	0	Miss	0x08	—	Compulsory (cold)
2	0x50						
3	0x42						
4	0x48						
5	0x53						
6	0x44						
7	0x4C						
8	0x41						

Fully associative

#	Address	Tag	Hit/Miss	Contents (LRU to MRU)	Miss Type
1	0x40	0x10	Miss	0x10	Compulsory (cold)
2	0x50				
3	0x42				
4	0x48				
5	0x53				
6	0x44				
7	0x4C				
8	0x41				

(c) How many misses does each version have? Calculate the miss rate for each design (miss rate = misses ÷ total accesses). What happens as associativity increases?

(d) Why can a fully associative cache never have conflict misses? Why does it still miss on the last access?

(e) For each miss type, what change to the cache would help reduce it? Why don't we just make every cache fully associative?

Bonus: Now the trace is **0x40, 0x44, 0x48, 0x4C, 0x50,** repeated forever. What is each version's miss rate once it settles? Is more associativity always better?

5. 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 titanium), 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?

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 *focuses on 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.