Homework 3

Memory and Cache

1. Reorder the fields in this structure so that the structure will (a) consume the most space and (b) consume the least space on an IA64 machine on Linux.

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

2. Consider a processor that uses 24-bit addresses and can address 2^{24}=16M 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 32768 bytes, with a block size of 256 bytes. Answer each of the following for direct-mapped, 2-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. Fully associative
   b. Direct mapped
   c. 4-way set 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 he 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 _prefetch(x) __builtin_prefetch((x), 0, 3)
```

```
#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`. GCC provides a builtin function named `__builtin_prefetch` that we've wrapped with a macro, because for our purposes we only care about the first argument. `_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.

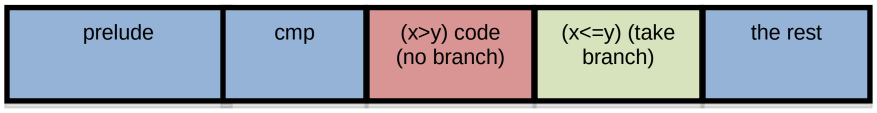
a. How big is the array that `a` points to? Will it fit in the cache?
b. Why did we choose to prefetch at the address `a+(i+1)*COLS` and why did we place it in the outer loop? Explain your reasoning. (Hint: it's no coincidence that `COLS` is 16)
c. Running the above code on our class machine *with* the 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. Your CPU performs best when it executes *straight-line* code, i.e. a sequential stream of instructions, one following immediately after the other in the program text. This *instruction locality* makes it easy for it to prefetch subsequent instructions, just as it does for data. Control flow instructions like `jle` and, to a lesser extent, `jmp`, disrupt these sequential streams and force the program counter to change in a less predictable manner, making the instruction prefetcher less effective. One goal of an optimizing compiler is to make these regions of straight-line code (called *basic blocks*) as large as possible to improve performance. Consider the following function and the assembly it generates.

foo.c
```
int foo (int x, int y) {
    if (x > y)
        return 1;
    else
        return 0;
}
```

foo.S
```
0000000000000000 <foo>:
   0: 55                      push   %rbp
   1: 48 89 e5                mov    %rsp,%rbp
   4: 89 7d fc                mov    %edi,-0x4(%rbp)
   7: 89 75 f8                mov    %esi,-0x8(%rbp)
   a: 8b 45 fc                mov    -0x4(%rbp),%eax
   d: 3b 45 f8                cmp    -0x8(%rbp),%eax
  10: 7e 07                   jle    19 <foo+0x19>
  12: b8 01 00 00 00          mov    $0x1,%eax
  17: eb 05                   jmp    1e <foo+0x1e>
  19: b8 00 00 00 00          mov    $0x0,%eax
  1e: 5d                      pop    %rbp
  1f: c3                      retq
```

Notice how GCC orders the jumps. Consider the case when `x` is *less* than `y`. The CPU has to break sequential execution and jump to address `0x19`. This will negatively affect the performance of the instruction prefetcher. But what if we know that `x` will be less than `y` 99 out of 100 times? That performance hit would be unnecessary. Assume the following logical block layout for this code:



a. CPUs can deal with conditional branches like this by predicting which way they will go using a hardware *branch predictor*. Assume that we have a dumb branch predictor in our CPU that always predicts that a branch is not taken. In other words, it always predicts that it will continue executing at the *next* instruction. How often will our branch predictor be wrong (i.e., what is its misprediction rate)?

b. Given our knowledge of `x` and `y`, how would you reorder the blocks to maintain straight-line code? Now what is the misprediction rate using the same branch predictor?

c. With your blocks reordered, what will the `cmp` and subsequent conditional jump look like in assembly? (will it still be a `jle`?)

d. GCC provides a built-in branch hint function named `__builtin_expect` that will do this reordering for you. It is typically renamed to a pair of functions so that it can be used like this:

```
if (likely(cond))   { //do something }
if (unlikely(cond)) { //do something }
```

Using `likely` will tell the compiler that it should order the block of the `if` statement first, as it is likely to be executed more often. `unlikely` does the reverse. These are used to increase performance in real-world projects like the Linux kernel. Again, given your knowledge of `x` and `y`, how would you rewrite the C code for `foo` using these functions?