CS4400 *Alex Clemmer, u0458675*

Homework 8 Problem Set

1 Problem 6.28

1. Addresses that hit for set 1 are: `0x8a4`, `0x8a5`, `0x8a6`, `0x8a7`, `0x704`, `0x705`, `0x706`, and `0x707`.
2. Addresses that hit for set 6 are: `0x1238`, `0x1239`, `0x123a`, and `0x123b`.

2 Problem 6.31

1. $C = E \cdot B \cdot S = 4 \cdot 4 \cdot 8 = 128$ bytes.
2. The address breaks down like so: `CT CT CT CT CT CT CT CT CI CI CI CO CO`.

3 Problem 6.32

1. The address works out to be: `0 0 1 1 1 0 0 0 1 1 0 1 0`.
2. CO = `0x2`, CI = `0x6`, CT = `0x38`. This result in a **hit**, returning `0x93`.

4 Problem 6.36

Each array is 64 bytes in size; since our cache is 128 bytes in size, both the `dst` and `src` can fit, which leaves only the unavoidable cost of the cold misses left:

1.

	`dst`			
	Col 0	Col 1	Col 2	Col 3
Row 0	m	h	h	h
Row 1	m	h	h	h
Row 2	m	h	h	h
Row 3	m	h	h	h

	`src`			
	Col 0	Col 1	Col 2	Col 3
Row 0	m	h	h	h
Row 1	m	h	h	h
Row 2	m	h	h	h
Row 3	m	h	h	h

5 Problem 6.39

Stride-1 access pattern makes this straightforward: we write 4 elements per `point_color`, and we can fit 2 `point_colors` per block. So we will miss $\frac{16 \cdot 16}{2} = 128$ times

1. There are **1024** writes in total.

2. There are **128** writes that miss.
3. The miss rate is therefore **12.8%**.

6 Problem 6.40

The size of our array is exactly twice as big as the size of cache, so any accesses in the second half of the array will evict the corresponding blocks from the first half. In other words, `square[0][0]` will be overwritten by `square[8][0]`. Since we are iterating on `j` for `square[j][i]`, we are guaranteed to complete iteration over `j`, which means that when we increment `i`, we will have replaced the first half of the array in cache with the second half. This means that we start over with all misses again, and of course, that means the second half will be replaced with the first half, which means that when we access the second half again, it too will be all misses.

Luckily, we are writing 4 values for ever element in `square`. So we incur one miss, and 3 writes for every access.

1. There are **1024** writes in total.
2. There are **256** writes that miss.
3. The miss rate is therefore **25%**.

7 Problem 6.41

We are accessing the `y` element by itself. Since we can fit two `point_color`s into one block, only every other access is a miss. There are 256 total accesses, there will be 128 misses here.

The same goes for the second loop. There are 256 total accesses. Fortunately, every miss fetches two `point_color` structs, which means that we will incur 2 hits for the first `point_color` in the block, and 3 times for the second (since it avoids the cold miss that caused us to fetch both blocks). So there will be a total of 128 more misses, but also 640 hits.

1. There are **1024** writes in total.
2. There are **256** writes that miss.
3. The miss rate is therefore **25%**