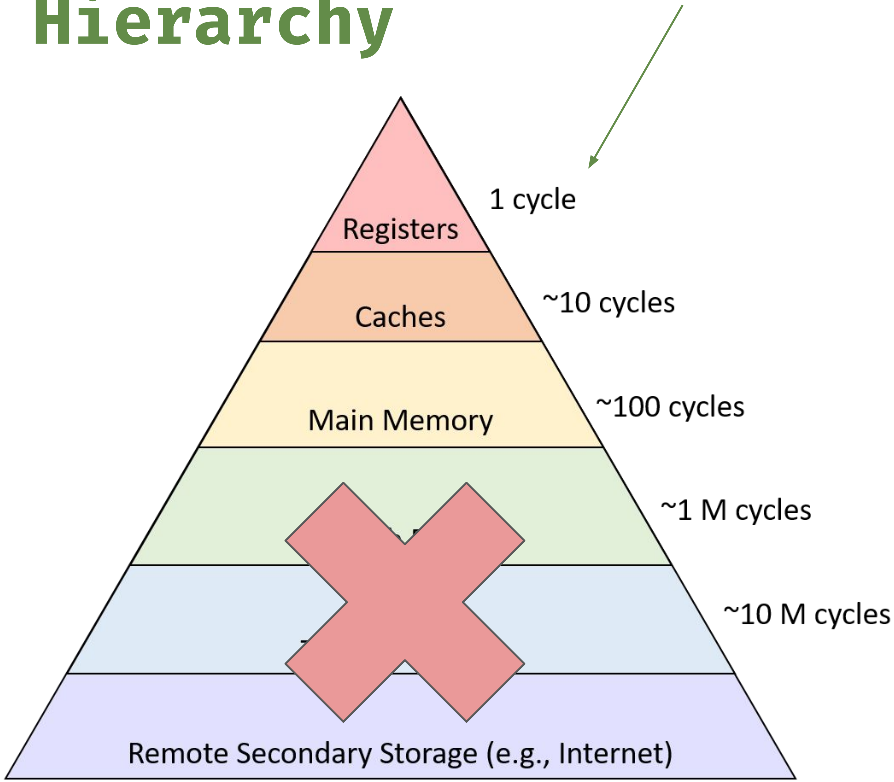
0x BARE C8 METAL

CMPSC 0xC8 ALLEGHENY COLLEGE 0x7E7

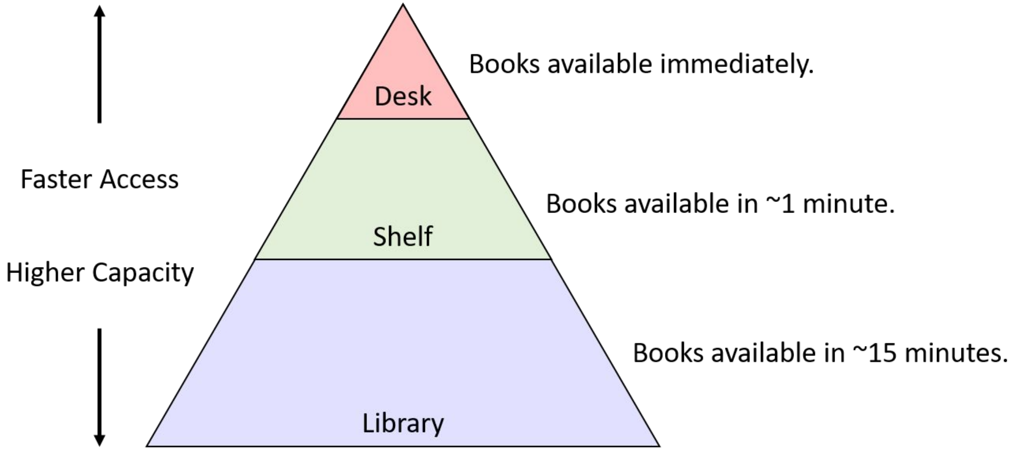
The Memory Hierarchy

More on these later.



The Memory Hierarchy

Temporal locality



Example Book Storage Hierarchy

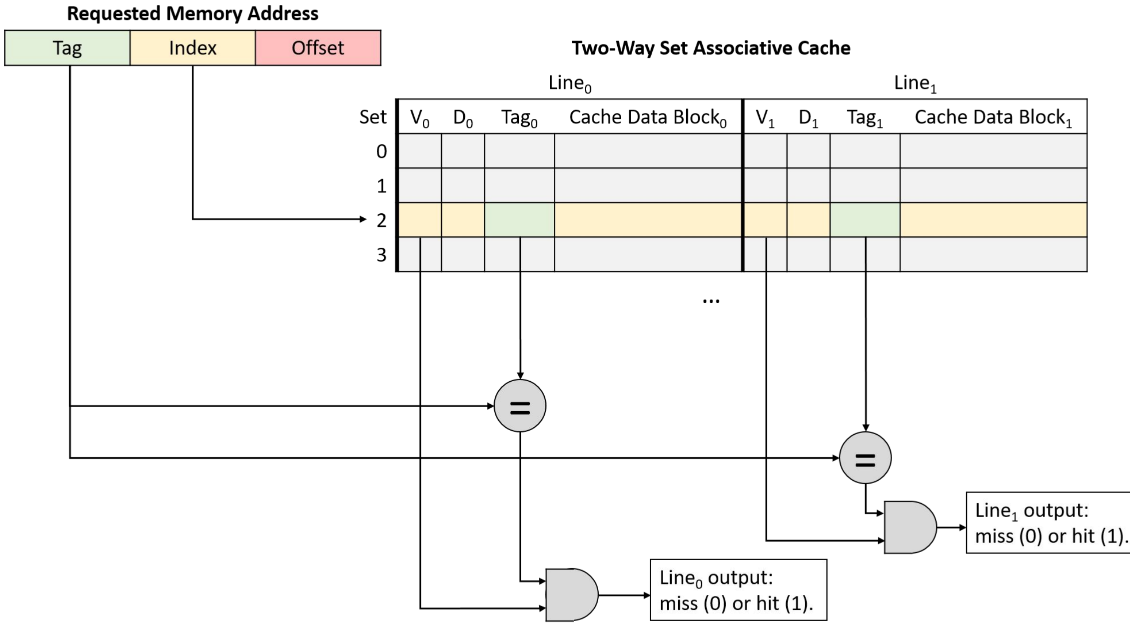
“Temporal” locality

Spatial locality

```
for (int i = 0; i < size; i++) {
    sum += array[i];
}
```

```
0000000020041fc0 93 11 00 00 00 00 00 00 01 00 00 00 02 00 00 00
0000000020041fd0 03 00 00 00 04 00 00 00 05 00 00 00 06 00 00 00
0000000020041fe0 07 00 00 00 08 00 00 00 09 00 00 00 0a 00 00 00
```

"Associative" caching



Instructions, data have a **tag number** which allows cache to find quickly

Spatial locality

```
for (int i = 0; i < size; i++) {
    sum += array[i];
}
```

```
0000000020041fc0 93 11 00 00 00 00 00 00 01 00 00 00 02 00 00 00
0000000020041fd0 03 00 00 00 04 00 00 00 05 00 00 00 06 00 00 00
0000000020041fe0 07 00 00 00 08 00 00 00 09 00 00 00 0a 00 00 00
```

“Associative” caching

LRU: a one-bit flag that indicates whether the leftmost $line_0$ of the set was least recently used (LRU = 0) or the rightmost $line_1$ of the set was least recently used (LRU = 1).

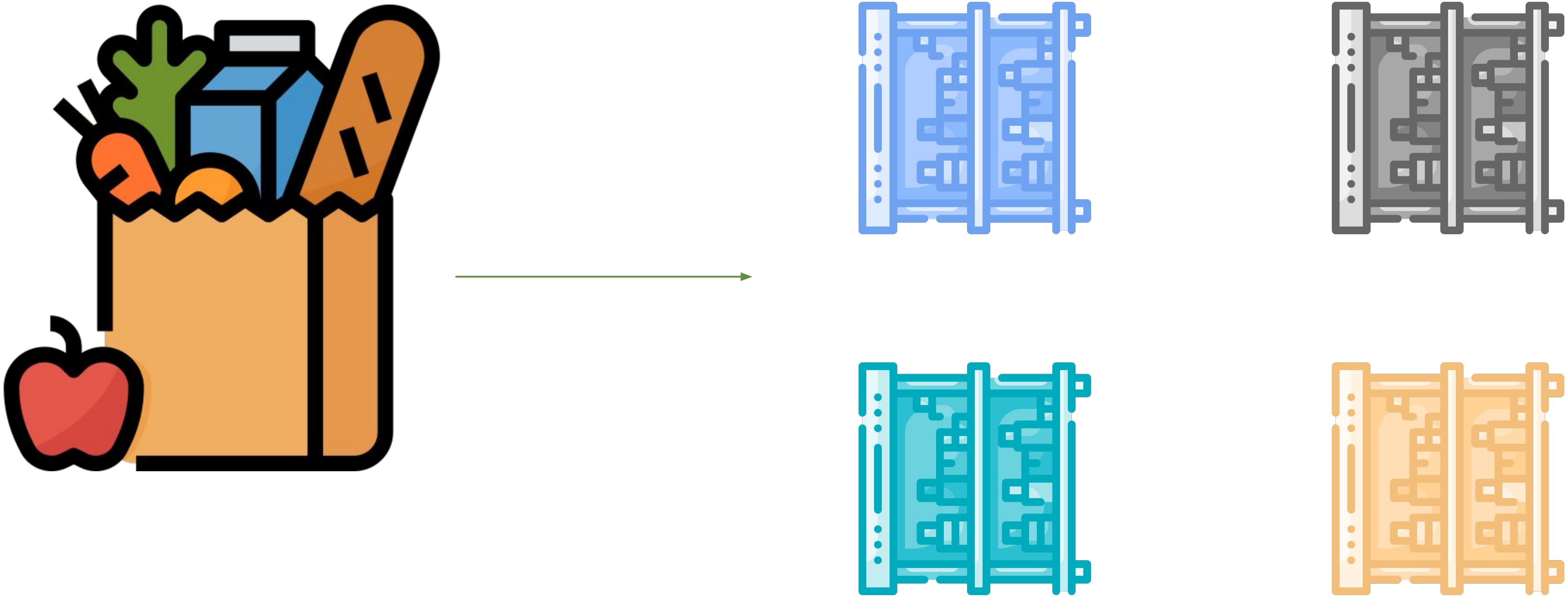
Leads to “cache eviction” if one match used more recently than another, freeing up space.

Two-Way Set Associative Cache

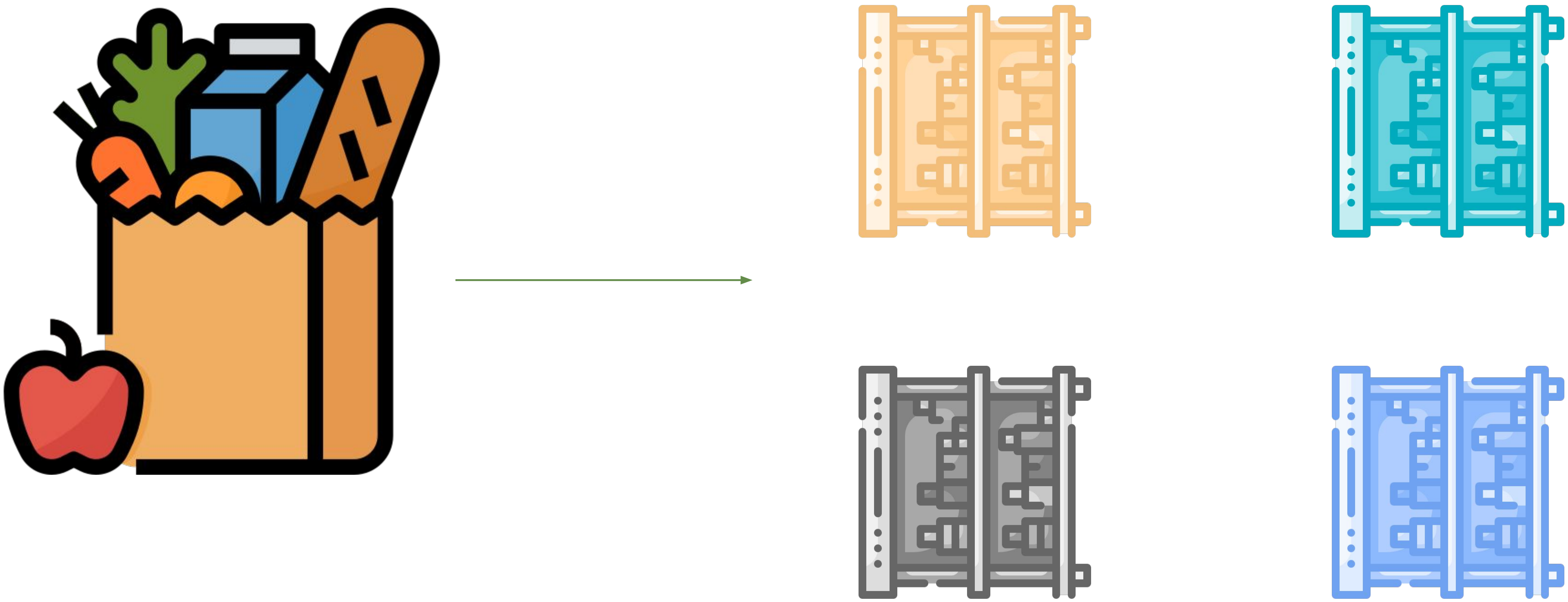
		$Line_0$				$Line_1$			
Set	LRU	V_0	D_0	Tag_0	Cache Data $Block_0$	V_1	D_1	Tag_1	Cache Data $Block_1$
0									
1									
2									
3									

...

Cache rules everything around me



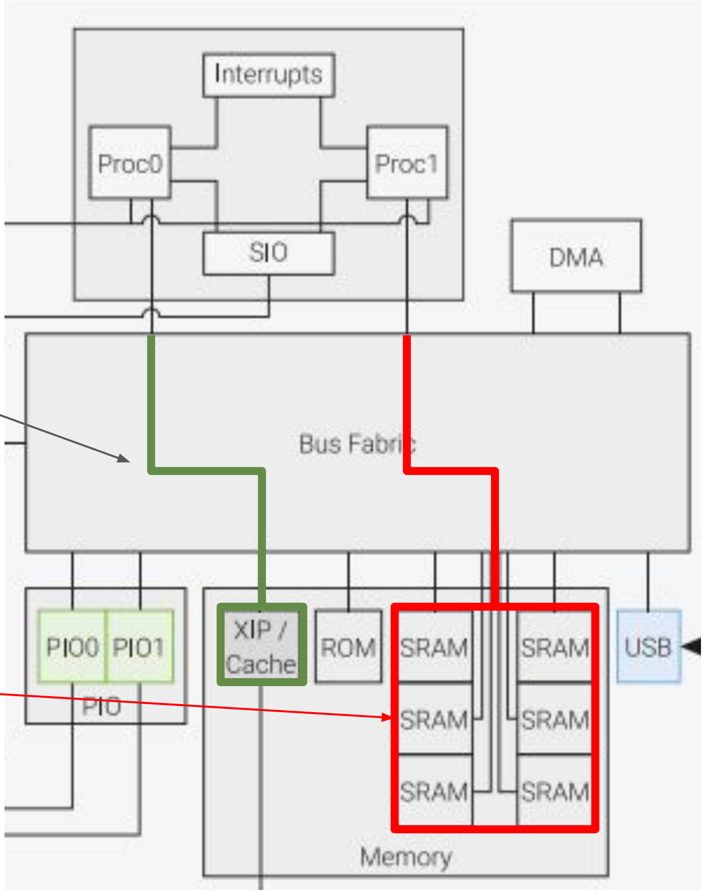
Cache rules everything around me



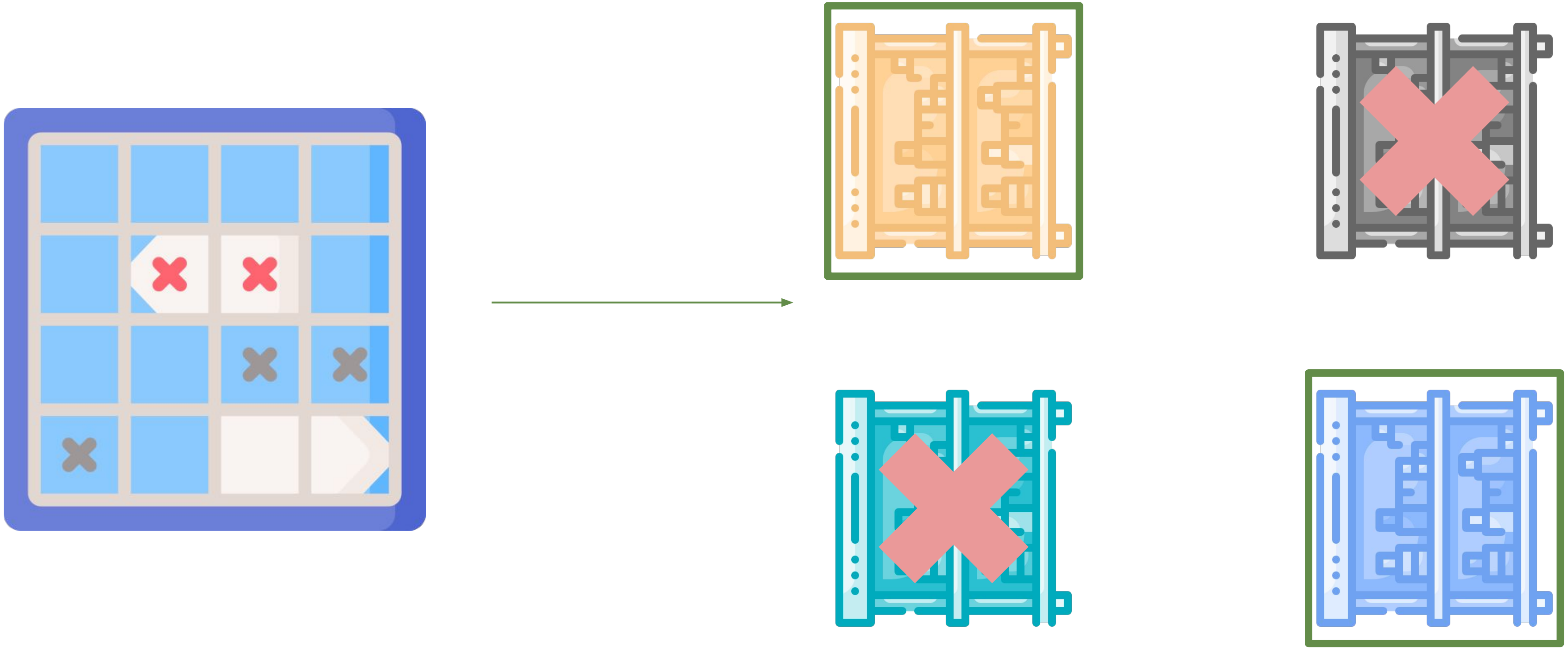
Why cache at all?

- Single source
- Direct line to processing unit

- Many sources
- First have to figure out *where* data is



Cache rules everything around me



Hit or miss

Theoretically…

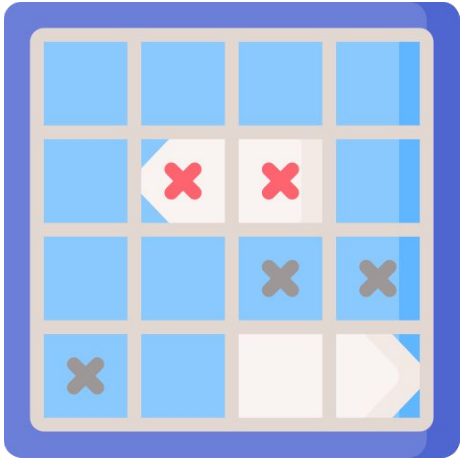
$$misses = \frac{num_{elements}}{\frac{block\ size}{size_{elements}}}$$

10

32

4

Hit or miss



$$hit\ rate = hits * \frac{100.0}{accesses}$$

$$miss\ rate = 1 - hit\ rate$$

Hit or miss

$$hit\ rate = hits * \frac{100.0}{accesses}$$

```
float get_cache_hit_rate (void) {
    return xip_ctrl_hw->ctr_hit * 100.f / xip_ctrl_hw->ctr_acc;
}
```

All in the timing

We can naively prove the effectiveness of the cache by looking at program execution times with data *in* and *out* of the cache.

$$net\ time = time\ at\ end - time\ at\ start$$

Movin' on up

To perform this experiment, we actually have to *physically move the code* to a region of the dedicated cache chip which *voids caching* using a special kind of pointer.

The range where the cache dares not go

```
func_ptr_t_array sum_array_nocache = (func_ptr_t_array)
    ((int)sum_array - CACHE_BASE) + CACHE_BYPASS;
```

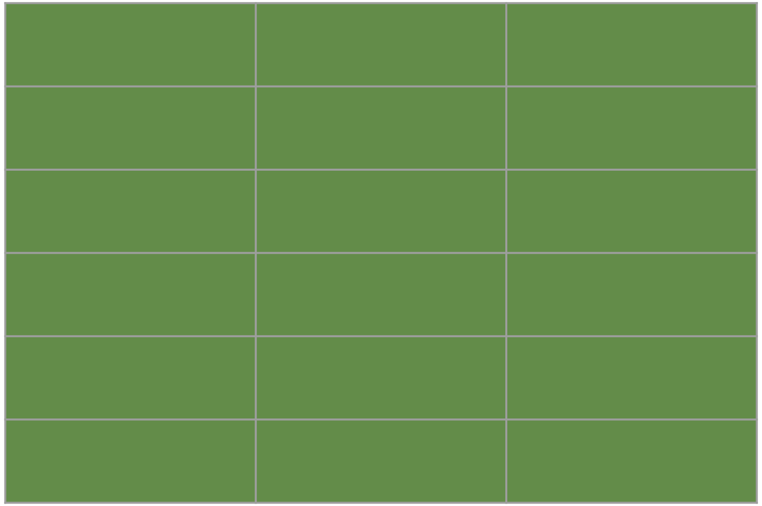
Where function “lives” right now

The base address of that range

So, I have a problem.

Which one reads *faster* with the lowest *miss rate*?

[i][j] matrix



[j][i] matrix