CS 24: Introduction to Computing Systems

Cache Memories

Last time, we talked about caches as if they were magical devices that knew where memory was, but how do they actually work? Today, we'll discuss how addresses are mapped from main memory to cache memories.

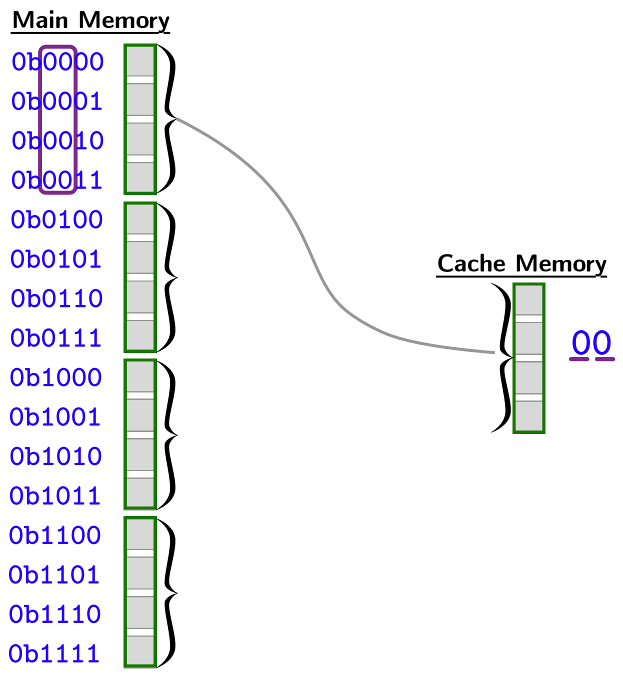
Blocks

We split up a cache into *blocks* which are consecutive chunks of memory of a particular size. The block size is supposed to be the amount of data the cache can transfer in one read/write. In our examples, we will use a block size of **4B**, but a more realistic size is **64B**.

Single Block Cache

The simplest cache imaginable is a single block cache. There is no choice about where to put the block to cache: they all go in the same place. In effect, this cache is just like a "hardware managed" *register*. Notably, to figure out which thing is loaded, we need to keep track of two extra bits (i.e., the high bits of the memory addresses) which we call "tag" bits. Then, to serve a request, the cache checks the two "tag" bits, and one of two outcomes occurs:

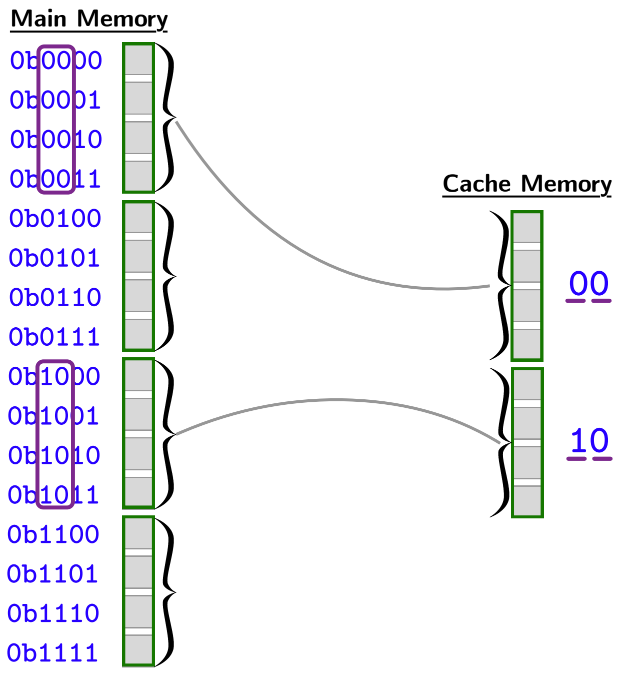
- If they match, then we just serve the byte requested from the cache using the rest of the address to figure out which one it is.
- If they don't match, we forward the request down the memory hierarchy. After the next memory sends back the result, we update the "tag" two bits and the data, and return it.



Two Block Cache

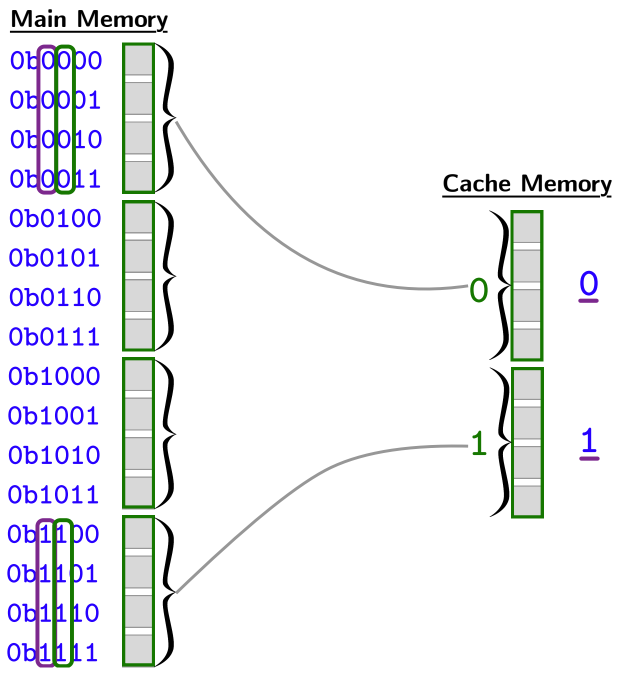
Now, let's imagine that our cache has two blocks. Unfortunately, now we have some decisions to make. We could do either of the following:

- Assign any memory address to any block, like we did previously. In this case, we might have to traverse through all the blocks only to find that the requested address isn't there. This should feel very reminicent of the *implicit free list* in `malloc`.



In this strategy, (called a "fully associative cache") the bits of the address are split into two parts (like before): the "tag" and the "offset" into the block.

- Assign half the blocks in main memory to each of the two blocks in cache memory. In other words, we could split memory up and cache separate pieces of it in separate blocks of the cache.



In this strategy, (called a "direct mapped cache") the bits of the address are split into three parts: the "tag", the "block", and the "offset" into the block.

Four Block Cache

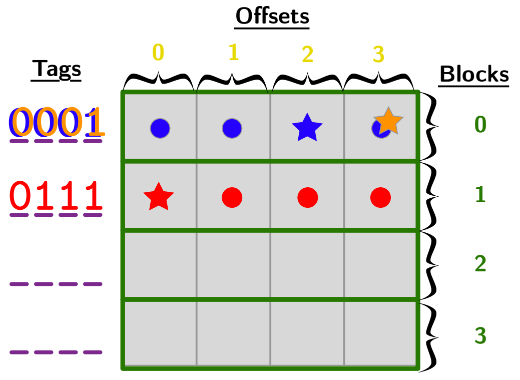
Now, let's consider addresses with 8 bit addresses (256B main memory), blocks of 4B, and caches of 16B. Naturally, we now have four blocks in our cache. Let's assume it's a direct-mapped cache to begin with.

Breaking down the address, we need two bits for the offset (the rightmost two bits), two bits to indicate which block, and the remaining bits are the tag. Here is an example of three addresses loaded into this kind of cache:

Address: $0b\underline{0001}\underbrace{00}_{0}\underbrace{10}_{2}$

Address: $0b\underline{0001}\underbrace{00}_{0}\underbrace{11}_{3}$

Address: $0b\underline{0111}\underbrace{01}_{1}\underbrace{00}_{0}$



In this example, the orange and blue addresses were in the same block in main memory; so, the cache only had to load the data once. But what happens if we try to load two *different* blocks from main memory that map to the same cache block?

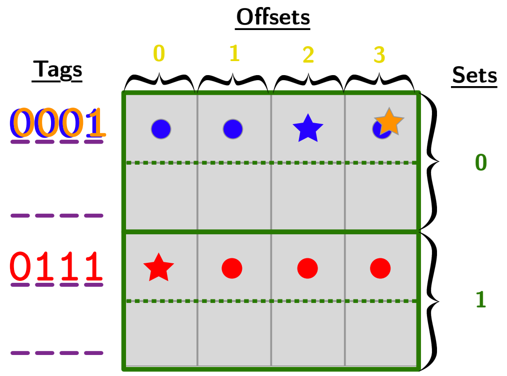
Noting that direct-mapped caches are more-or-less *hash tables* (where the block number is the hash code and the tag is the key), we can use similar collision resolution strategies here. In particular, we can split up the cache memory into “sets” of associative caches (i.e., fixed-size lists of buckets in the hash table).

At a high level, this strategy is a combination of the other two: we have a hash table with buckets that are implicit lists. Here is a similar example to the above:

Address: $0b\underline{0001}\underbrace{00}_{0}\underbrace{10}_{2}$

Address: $0b\underline{0001}\underbrace{00}_{0}\underbrace{11}_{3}$

Address: $0b\underline{0111}\underbrace{01}_{1}\underbrace{00}_{0}$



Replacement and Eviction

In all three strategies, there is still a chance that all the valid spots to place a block from main memory will be taken. In such a case, we have no choice but to "evict" one of the blocks, but which one? Most hardware caches implement some version of a `Least Recently Used` (or LRU) policy. In such a policy, the block that was least recently read from/written to is chosen to be evicted. (Of course, this requires even more information to be stored; so, the hardware implements an approximation of this policy.) Additionally, we need to be able to initialize the cache when the machine is turned on; so, we store a "valid" bit which keeps track of if the data in that block is usable or not.