Caches

CS141 Spring 2019

Introduction

A cache is a piece of fast memory that is used to speed up memory accesses by storing certain blocks of memory that are likely to be used soon. Main memory is usually made from DRAM which is dense but slow compared to SRAM. Accesses a block of memory in DRAM may take 20 to 50 times longer than an access from SRAM. However, the amount of fast memory is limited so the computer can only keep a subset of main memory in the cache at any given time.

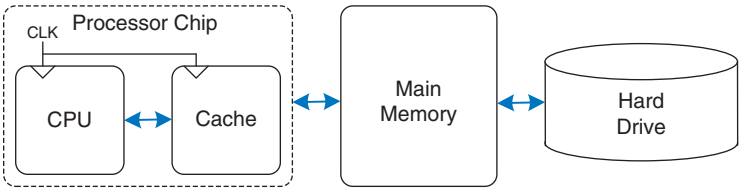

Figure 1: Memory hierarchy

When the processor wants to access data from main memory, it first checks if the data is in the cache. If so, this is called a *cache hit* and data is accessed quickly. If the data is not found in the cache, it must be loaded into the cache from main memory, and then sent to the processor. This is called a *cache miss*, and we want to minimize the number of misses. If an address is requested by the processor that has never been in the cache before, then a miss is unavoidable, and this is called a *cold miss*, because we haven't given the cache time to "warm up."

The best cache policy would be to know ahead of time what addresses the processor would like to access and load them beforehand. Unfortunately, we can't predict the future so we instead use heuristics to try to optimize cache performance. In particular, the cache takes advantage of temporal and spatial locality. Temporal locality means that data that is accessed once is likely to be accessed again soon, and spatial locality means that the processor is likely to access memory locations that are near to those that it has already accessed.

In designing caches we must ask the following questions:

- Where should we put a block of memory when we load it into the cache?
- How can we tell if a block of memory is already in the cache?
- What policy should be used for replacing memory blocks once the cache fills up?

A cache is made up of S *sets*, each responsible for one or more blocks of memory. Along with the memory data that must be stored in the cache, we also have to keep track of any information needed to remember where the data is from in main memory. For different kinds of caches, this *mapping* between sets and memory locations is defined differently.

For now we will assume that each set only contains one memory block. We will revisit what happens with multiple blocks per set at the end of the document.

Direct mapped cache

A *direct mapped* cache is the simplest policy: we take each main memory address and map it to a certain set in the cache.

Consider an example where each memory block is one word (4 bytes), and our cache can store 8 sets (each containing one memory block). If we use byte-addressable memory, the lower order two bits will always be 00 because the addresses must be word-aligned. Since our cache stores 8 sets, we have 3 bits for the cache index. We just take the next 3 bits in the address to determine which cache block the main memory block should be mapped to.

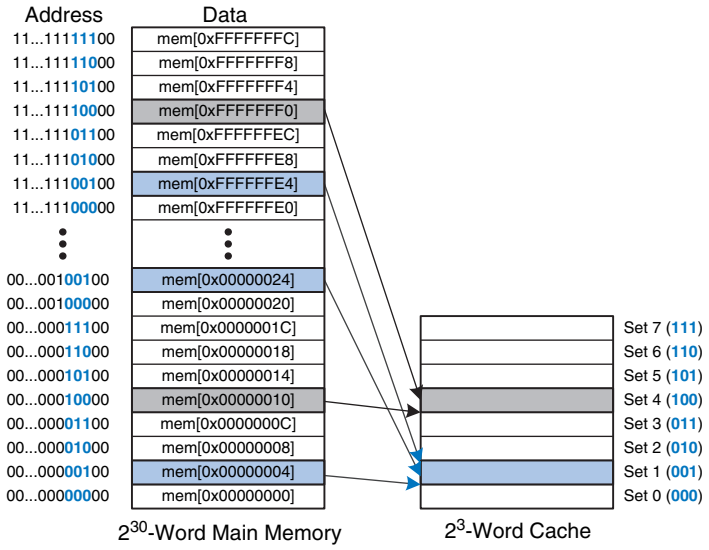

Figure 2: Mapping of main memory to a direct mapped cache

This answers the first question of how to put the memory in the cache. Now we must consider what information is necessary for us to know to be able to tell if a certain address is in the cache.

We can divide a memory address into various fields: The byte offset, the set index, and the tag. In cases where the memory address isn't word aligned, we still load the word-sized memory block containing the target location into the cache, but we mark what the lower order bits were as the *byte offset*. The next three bits are the *set bits* which indicate which cache index the memory block is stored in. Finally the remaining bits of the address are the *tag bits* and indicate the full address.

Figure 3: Cache address fields for address `0xFFFFFFE4`

We also have to actually store the data in the cache. The cache also uses a *valid bit* to indicate for each block whether it holds meaningful data. If the valid bit is 0 for a certain block, then that location in the cache is free and the data there is meaningless. The figure below shows the hardware implementation of the cache we discussed above.

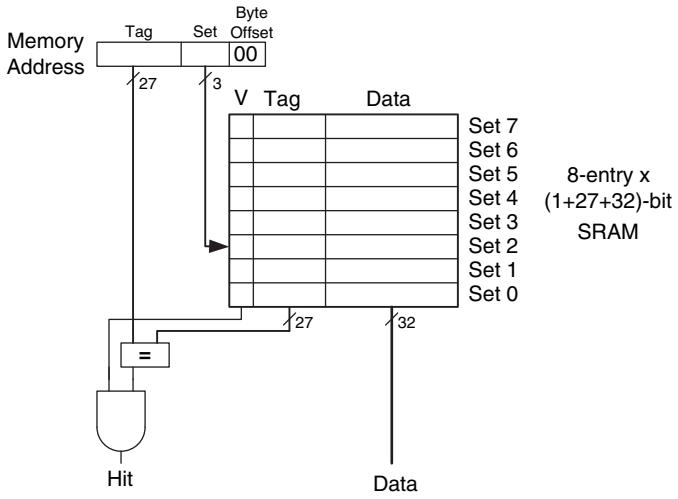

Figure 4: Direct mapped cache hardware

Now we know how to tell if an address is in the cache and how to read it.

When new data is loaded into the cache into a slot that already contains data, there is a conflict. The conflicting cache block must be *evicted* from the cache and replaced with the new data. Due to the mapping of direct mapped caches, certain addresses will always conflict with each other. This is not always the case for other cache policies.

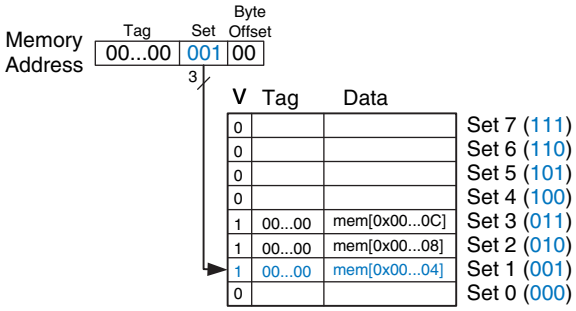

Figure 5: Direct mapped cache conflict

Direct mapped caches aren't so good for temporal locality. Consider the following access pattern: 0x04, 0x24, 0x04, 0x24, ... Even though our cache has 8 slots in it, we only use one of them and each time we evict the previous value we put in the cache. This means that every single access is a miss even though we are only accessing 2 distinct locations in memory!

Fully associative cache

A fully associative cache fixes this problem by allowing memory data to be stored in any set. The trade-off is that the logic for managing the cache is much more complex. Finding if a block is in the cache requires a lot of hardware, and eviction becomes more difficult as well.

With a fully associative cache, we simply fill up the cache however we like until it is full. Since the tag bits now must consist of the entire memory address, looking up an address to see if it is in the cache involves S comparisons, where S is the number of sets the cache can store. For a fully associative cache, every set is also a *way*, which will become clear in the next section when we generalize to the set associative cache. An eight-set fully associative cache is shown below.

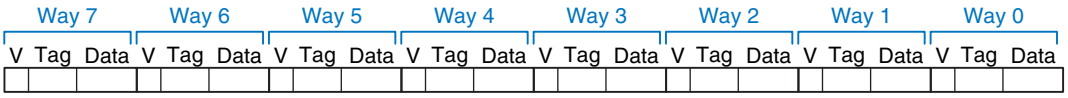

Figure 6: Eight-way fully associative cache

We must also have a policy for which block is evicted when new data is read and the cache is full. One good policy is *least recently used* (LRU). With this policy, we evict the memory block that was used the longest in the past. Unfortunately keeping track of this is expensive in hardware, so some approximations are used which we won't discuss. Other eviction policies may be used such as FIFO (First In First Out) or LFU (Least Frequently Used).

N-way set associative cache

A set associative cache is a mix between direct mapped and fully associative. Here the cache is divided into *sets* (or blocks in this case) each with N *ways*. Within each set, the data may be placed in any way, but addresses are directly mapped to certain sets like in the direct mapped cache.

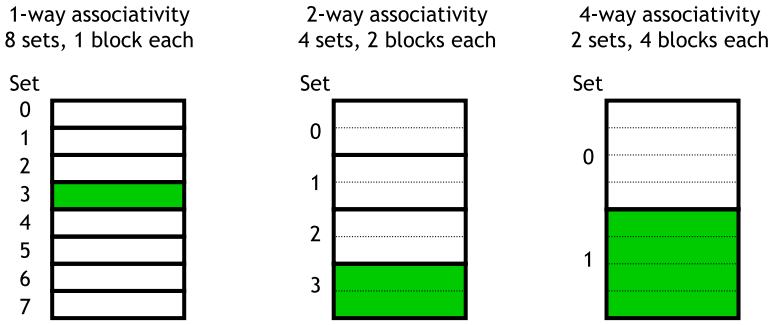

Figure 7: Various set associative caches

To check if a block is stored in the cache, we first use the set bits to determine which set the address is mapped to. Then within that set there are N ways, and the tags of each way must be compared against the tag of the address we are checking against. The hardware is shown below.

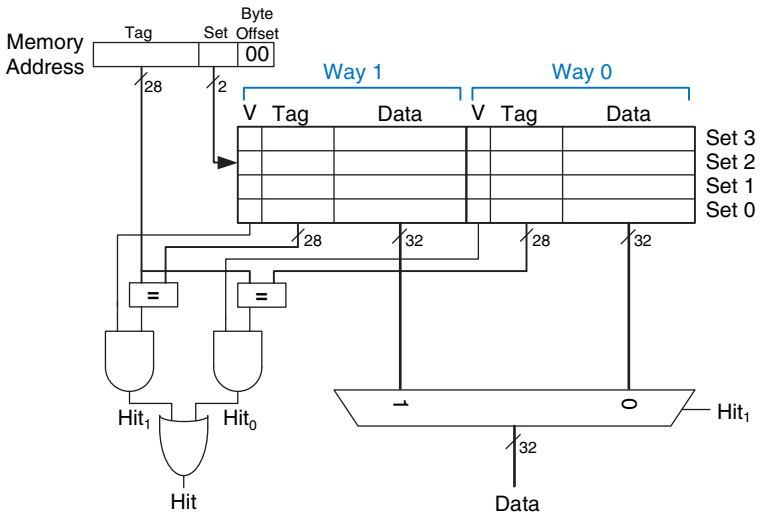

Figure 8: Set associative cache hardware

Set associative caches generally perform better than direct mapped caches of the same size because they reduce the number of conflicts (though not as much as a fully associative cache). They also don't have as much overhead as a fully associative cache because only N comparators are needed for a cache of size S rather than S comparators (this is still more overhead than a direct mapped cache). Most commercial systems use set associative caches.

You may have also noticed that direct mapped caches and fully associative caches are just more specific versions of a set associative cache. Use 1 way and you get a direct mapped cache and use S ways and it becomes fully associative.

Multi-block sets

So far we have only examined caches where one set stores one block. This doesn't exploit spatial locality because we don't try to keep memory blocks in the cache that are near locations we have read. If we extend the cache so that sets store more than one block, we can use pre-fetching to try to take advantage of spatial locality. When we use a block size that is greater than one, the data adjacent to the data we want to read is also loaded into the cache. This is great for spatial locality because if the next read is nearby, we will have already loaded it into the cache. However, increasing the block size will mean the cache will have fewer blocks which may lead to more conflicts. The cache will also be slower on a miss because the extra blocks will have to be loaded.

The hardware for a direct mapped cache with a 4-word block size is shown below.

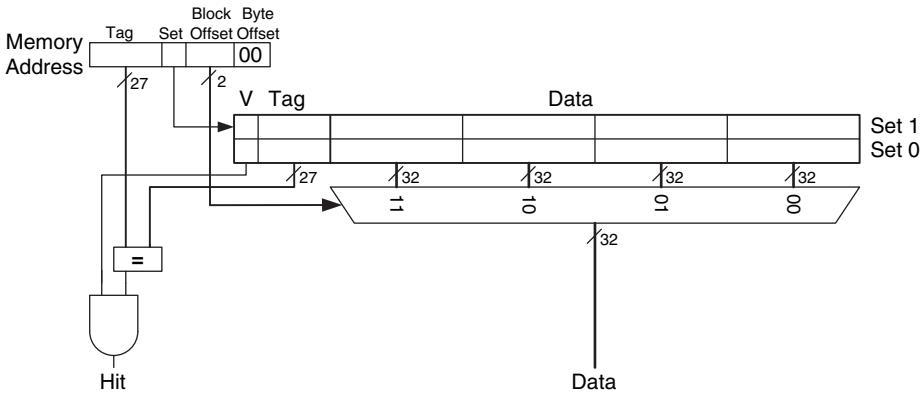

Figure 9: Multi-block direct mapped cache hardware

We now also need to keep track of the *block offset* bits to specify where the data is within each block. The cache fields for the direct mapped cache are shown below.

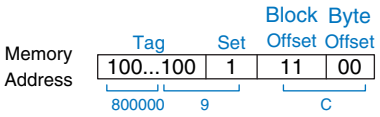

Figure 10: Cache fields for address `0x8000009C`

Writing to the cache

So far we have only discussed caches for reading data from memory. Caches can also be used for writing data. During a store, the processor checks the cache for the address. If there is a miss the block is fetched from memory and placed in the cache, and the data is written into the cache. If the cache hits, the data is written to the cache.

Caches can be classified as *write-through*, which indicates that when data is written to the cache, it is immediately also written to main memory. Alternatively a cache may be *write-back*, where a *dirty bit* is kept for every cache block. The bit is 1 if the cache block has been written and 0 otherwise. When a dirty cache block is evicted from the cache, its data is written to main memory. Modern caches tend to be write-back, because write-back caches don't have as many writes to main memory.

Bibliographic notes

Notes by Zach Yedidia.

Figures and examples from *Digital Design and Computer Architecture* by Harris and Harris.

Figure 7 is from University of Washington CSE378 lecture slides.