EECS 370: Introduction to Computer Organization Notes

Noah Peters

March 29, 2023

Abstract

Lecture notes for EECS 370 at the University of Michigan. LaTeXtemplate by Pingbang Hu.

Contents

Chapter 3

Caches

Lecture 17: Introduction to Caches

3.1 Memory Hierarchy

14 Mar. 12:00

We often need a lot of memory, LC2K alone can handle 2^{18} bytes of memory. We have several choices for memory:

- **SRAM**: Static Random Access Memory
  - fast: 2ns access time or faster
  - decoders are big
  - expensive, high area requirement
- **DRAM**: Dynamic Random Access Memory
  - slower: 50ns access time
  - must stall for dozens of cycles each memory load
  - less expensive
- **Flash**
  - slow: access time varies wildly
  - less expensive than DRAM
  - non-volatile
- **Disks**
  - obnoxiously slow: 3,000,000ns access time
  - dirt cheap
  - non-volatile

As seen above, there are trade-offs among each type of memory. Ideally, we would have cheap *and* fast memory. So, we can use a combination of memory types to optimize the common case via strategic **locality of reference**.

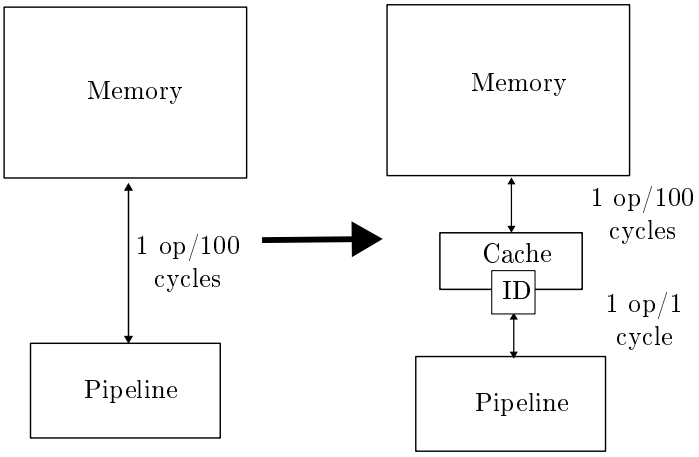

Figure 3.1: Caches Overview

Definition 3.1.1. The **architectural** view of memory is what the machine language (or programmer) sees, i.e. just a big array.

Note. Breaking up the memory system into different pieces (cache, main memory, disk) is **not architectural**. The machine language doesn't know about it.

Can use our variety of memories as follows:

- use small array of SRAM for the **cache**
- use a larger amount of DRAM for **main memory**
- use a lot of flash and/or disk for **virtual memory**

3.2 Cache Basics

Whenever memory returns data, we can store it in a cache. We'll need to store:

- the data
- a tag denoting its memory location
- a "valid" status bit

Then for our next memory access, we can first check if the tag matches the address we are attempting to access.

Definition 3.2.1 (Cache Hit). A **hit** occurs when data for a memory access is found in the cache.

Definition 3.2.2 (Cache Miss). A **miss** occurs when data for a memory access is *not* found in the cache.

Definition 3.2.3 (Hit/Miss Rate). The **hit/miss rate** is the percentage of memory accesses that hit/miss in the cache.

3.2.1 CAMs

Definition 3.2.4. **CAMs: content addressable memories** are akin to a set of data matching a query. Instead of an address we send a key to the memory, asking whether the key exists and, if so, what value it is associated with. Memory answers: yes/no and gives associated value (if there is one).

We apply *operations* on CAMS:

- **Search**: the primary way to access a CAM
  - send data to CAM memory
  - return "found" or "not found"
  - if found, return location of where it was found or its associated value
- **Write**: send data for CAM to remember

3.2.2 Cache Organization

Cache memory can copy data from any part of main memory. Cache memory has two parts:

- the **tag (CAM)**: holds the memory address
- the **block (SRAM)**: holds the memory data

A **hit** in the cache occurs when a tag match is found. The microprocessor sends the address to the CAM containing the tags and searches for the tag. If there's a search result hit, the corresponding block is forwarded to the microprocessor. If not, the address is forwarded to main memory:

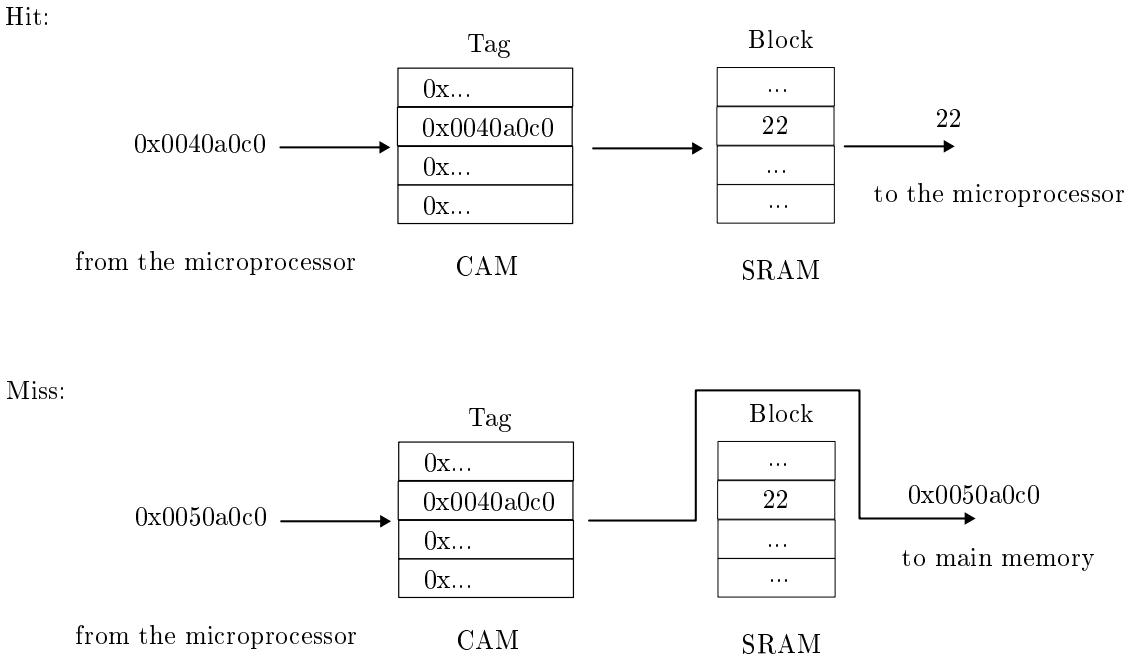

Figure 3.2: Hardware view of caches.

Problem 3.2.1. Given:

- *cache* has 1 cycle access time
- *main memory* has 100 cycle access time
- *disk* has 10,000 cycles access time

What is the average access time for 100 memory references if 90% of the cache accesses are hits

and 80% of the accesses to main memory are hits? Assume main memory access time includes tag array access to determine hit/miss.

Answer. $0.9 \cdot 1 + 0.1 \cdot (100 + 0.2 \cdot 10000) = 210.9$ ⊛

3.2.3 Cache Operation

Every cache *miss* will get the data from memory and *allocate* a cache line to put the data in (just like any CAM write). However... what do we replace in the cache? Does an optimal replacement policy exist?

Definition 3.2.5 (Temporal locality)**.** The principle of **temporal locality** in program references says that if you access a memory location (e.g., 1000) you will be more likely to re-access that location than you will be to reference some other random location.

Remark. Locality is a property of *programs* (not hardware).

Specifically, temporal locality says that the **least recently referenced (LRU)** cache line should be *evicted* to make room for the new line. Because the re-access probability falls over time as a cache line isn't referenced, the LRU line is least likely to be re-referenced.

Definition 3.2.6 (Average Access Latency)**.** Average Access Latency = cache latency · hit rate + memory latency ·

Lecture 18: Cache Organization: Block Size and Writes

16 Mar. 12:00

Definition 3.2.7 (k-Way Set Associative LRU)**.** A set associative LRU with k **ways**, has essentially k cells within each cache line's block. The LRU is set to be the *way* with the highest *count*. 20:43

Each cache hit now essentially brings in two blocks of data, since each block in memory is paired up with an adjacent block. Now, we can have the LSB of our memory address correspond to either block A or block B of the retrieved block, and the remaining bits correspond to a (now-shortened) memory address.

Definition 3.2.8 (Spatial Locality)**.** **Spatial locality** in a program says that we reference a memory location (e.g., 1000), we are more likely to reference a location near it (e.g., 1001) than some random location.

3.3 Cache Organization

We now consider the design of a cache.

3.3.1 Block Size

- choice of block size found by simiulating lots of different block sizes and seeing which ones give the best performance
- most systems use a block size between 32 and 128 bytes
- *longer sizes* reduce overhead by reducing the number of CAM entries and reducing the size of each CAM entry

Problem 3.3.1. Given a cache with the following configuration:

- total size is 8 bytes

- block size is 2 bytes
- fully associative
- LRU replacement
- memory address size is 16bits and is byte addressable

1. How many bits are for each tag? How many blocks in the cache?
2. For the following reference stream, indicate whether each reference is a hit or miss: 0, 1, 3, 5, 12, 1, 2, 9, 4.
3. What is the hit rate?
4. How many bits are needed for storage overhead for each block?

Answer.

1. 16 bits total - 1 bit for block offset = **15 bits** for each tag. 8 bytes total / 2 bytes per block = **4 blocks**.
2.
   - 0: never in cache before → **miss**
   - 1: 0 was just cached and block size is 2 bytes → **hit**
   - 3: 2 nor 3 have been cached yet → **miss**
   - 5: 4 nor 5 have been cached yet → **miss**
   - 12: 12 nor 13 have been cached yet → **miss**
   - 1: 1 was cached and hasn't been booted from cache → **hit**
   - 2: 3 was cached recently → **hit**
   - 9: LRU was 4/5 block, so 4/5 booted from cache, 8/9 in now → **miss**
   - 4: 4/5 block just booted, so not in cache; boots 12/13 → **miss**
3. $3/9 = 0.33$
4.
   - Tag: 15 bits
   - Valid: 1 bit
   - LRU: $\log_2(4) = 2$ bits

   ⇒ total: **18 bits**

⊛

3.3.2 Stores and the Cache

Where should we write the result of a store?

- If that memory location is in the cache?
  - Send it to the cache
  - should we also send it to memory (**write-through policy**)
- If that memory location is *not* in the cache?
  - Allocate the line, i.e. put it in the cache (**allocate-on-write policy**)
  - Write it directly to memory without allocation (**no allocate-on-write policy**)

Add memory diagram from lecture

Definition 3.3.1 (Write-Through Policy)**.** When storing a value from the processor to memory and the memory location is in the cache, we store the value in the cache and simultaneously change the value in memory.

Definition 3.3.2 (Allocate-On-Write Policy)**.** When storing a value from the processor to memory and the memory location is *not* in the cache, we must first *read* the desired block from memory, before storing into the block in the cache and memory.

Definition 3.3.3 (No Allocate-On-Write Policy)**.** When storing a value from the processor to memory and the memory location is *not* in the cache, we can ignore the cache and simply update memory with the given value.

Remark. This method may sometimes be less advantageous if we are going to read the value from memory soon after writing to it. However, write data streams and read data streams are often independent, so this isn't usually the case.

Appendix