EECS 370: Introduction to Computer Organization Notes

Noah Peters

March 28, 2023

Abstract

Lecture notes for EECS 370 at the University of Michigan. $\LaTeX$template 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**.

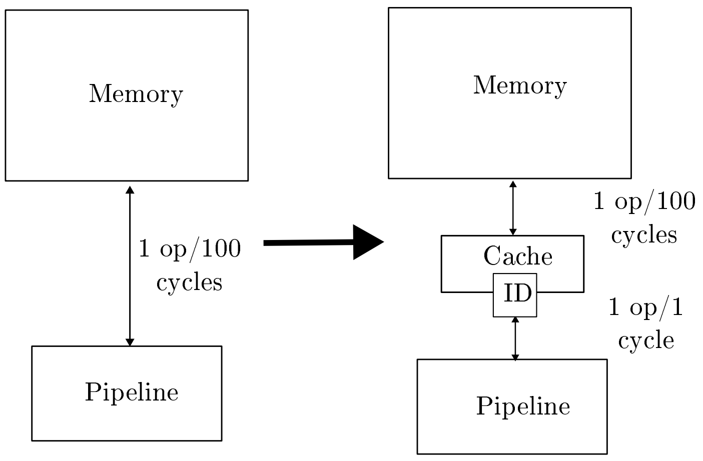


Figure 3.1: Caches Overview

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:

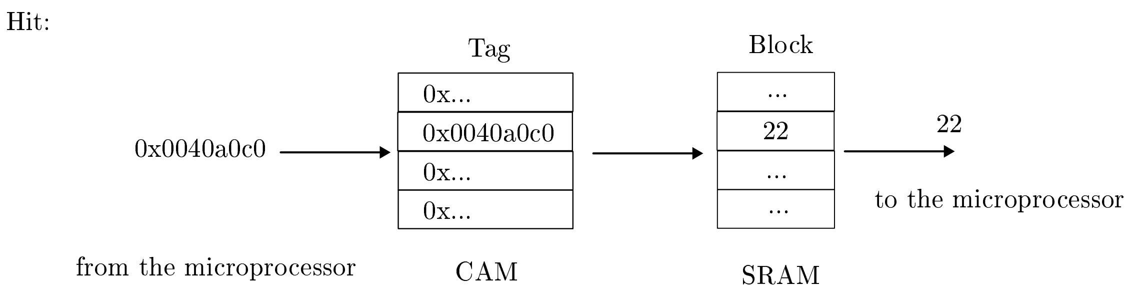
- **Search**: the primary way to access a CAM
  - send data to CAM memory
  - return "found" or "not found"
  - if found, return location of where it was found or its associated value
- **Write**: send data for CAM to remember

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:



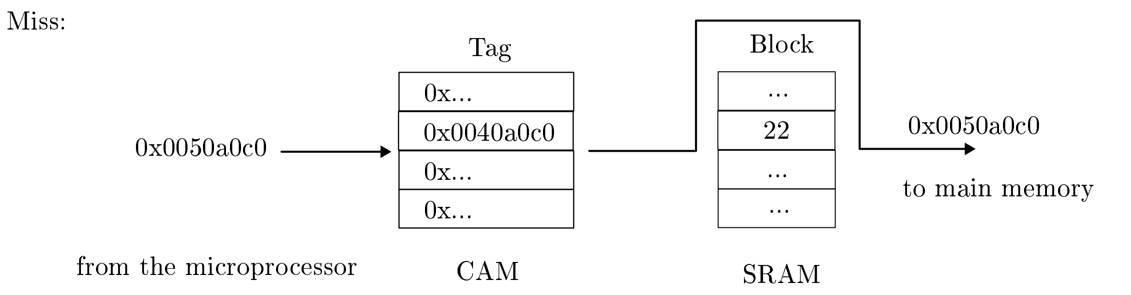


Figure 3.2: Hardware view of caches.

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

3.3 Cache Organization

Definition 3.3.1 (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

Appendix

Appendix A

Additional Proofs

A.1 Proof of ??

We can now prove **??**.

Proof of ??. See here. ∎