CSCI 564 Advanced Computer Architecture

Lecture 05: Cache Introduction

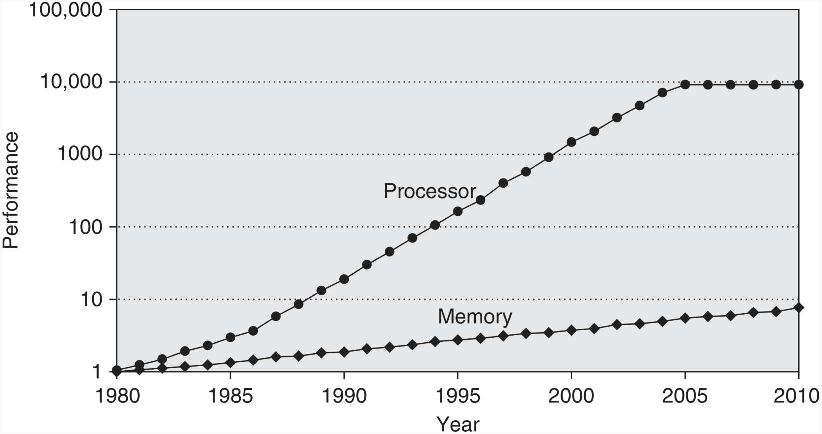
Dr. Bo Wu (with modifications by Sumner Evans)

March 3, 2021

Colorado School of Mines

Why do we need the memory hierarchy?

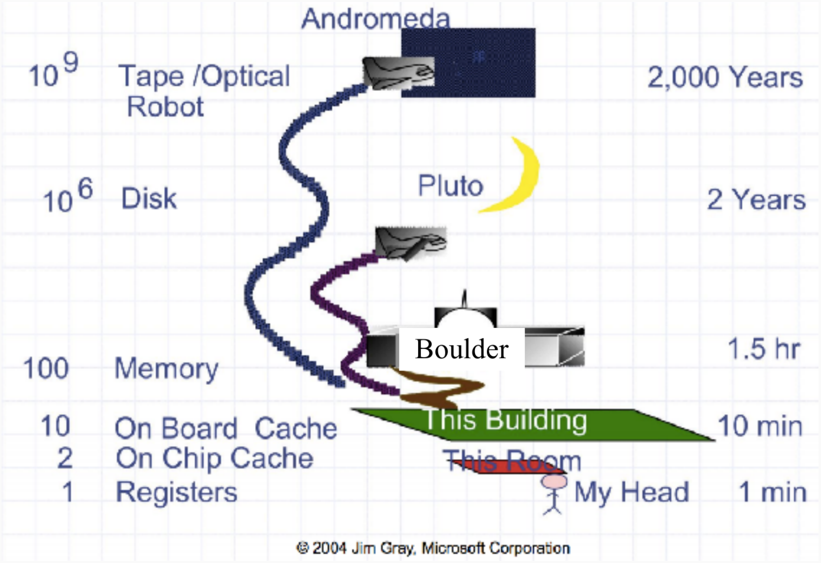
Processor vs. Memory Performance



Page 73

- 1980: no cache in microprocessor
- 1995: 2-level cache

Really, How Bad Can it Be?



Memory's Impact

$$\begin{aligned} M &= \text{\% of the instructions which are memory operations} \\ M_{lat}(\text{cycles}) &= \text{average memory latency} \\ CPI_{base} &= \text{base CPI with single-cycle data memory} \end{aligned}$$

Then, $CPI_{tot} = CPI_{base} + M \times M_{lat}$.

Example: $CPI_{base} = 1$; $M = 0.2$; $M_{lat} = 240$ cycles

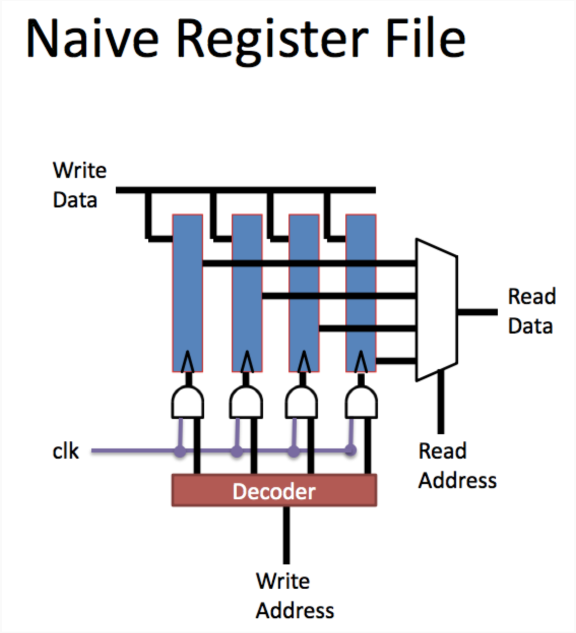
$$CPI_{tot} = CPI_{base} + M \times M_{lat} = 1 + 0.2 \times 240 \text{ cycles} = 49$$

Which means that Speedup $= \frac{1}{49} = 0.02$ which results in a 98% drop in performance.

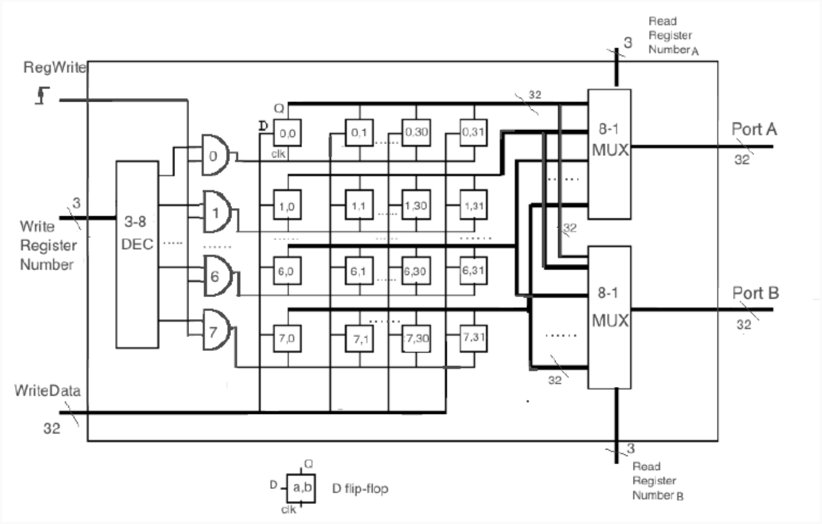
Remember: Amdahl's Law does not bound slowdown! Poor memory performance can make your program arbitrarily slow!

Why Are Registers and Cache Fast?

Implementing a Register File



Smarter Register File



https://stackoverflow.com/questions/22418555/test-bench-of-a-32x8-register-file-vhdl

Why is Cache Fast?

Registers and cache are built on SRAM (Static Random Access Memory) technology.

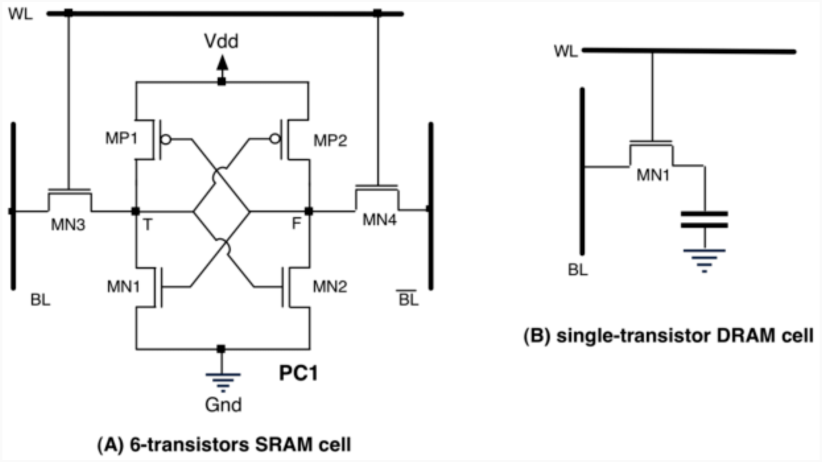
- Not *dense*
- Bandwidth [1]
    - L1 Cache — 210 GB/s
    - L2 Cache — 80 GB/s

Main memory is built on DRAM (Dynamic Random Access Memory) technology

- Needs to be *refreshed* periodically to persist data.
- Very dense
- Bandwidth of DDR4 — 25.6 GB/s (Dual In-line Memory Module)

[1] `https://www.forrestthewoods.com/blog/memory-bandwidth-napkin-math/`

What does “Dense” Mean?

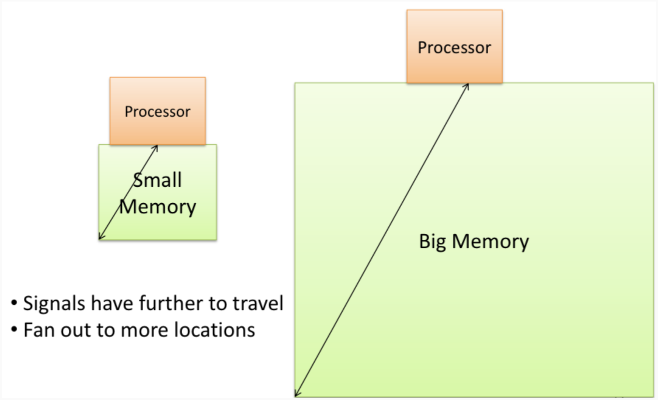


Di Carlo, Stefano & Prinetto, Paolo. (2010). Models in Memory Testing, From functional testing to defect-based testing.

SRAM Everywhere?

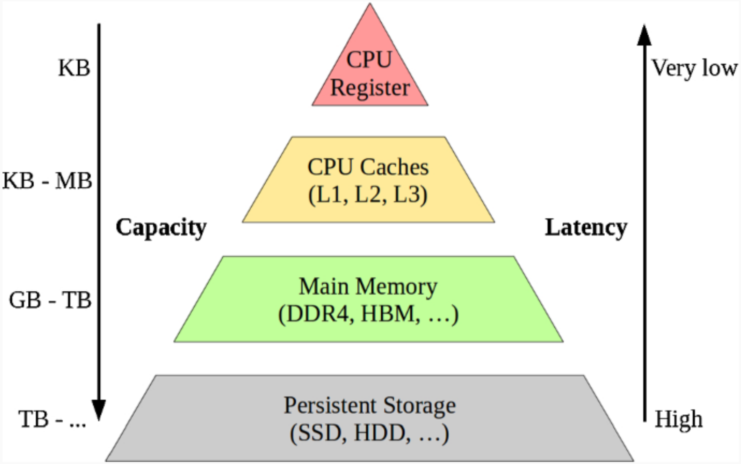
OK, so cache is fast, but why don't we build main memory with SRAM?

SRAM is much more expensive than DRAM!



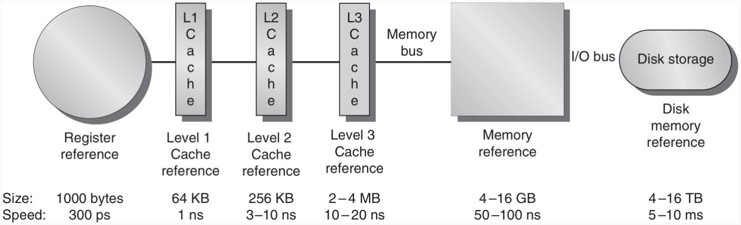
The Memory Hierarchy

The Memory Hierarchy



- Bandwidth: on-chip $\gg$ off-chip

The Memory Hierarchy



Page 72

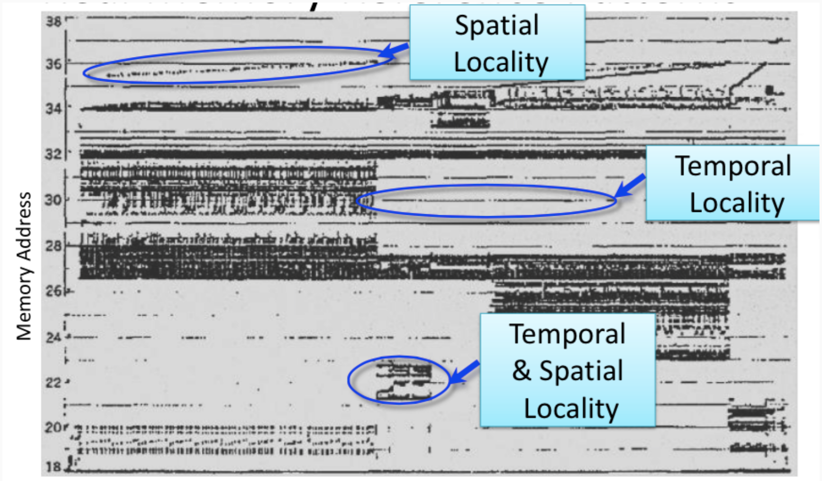
- On a data access instruction:
  - if the data is in fast memory $\rightarrow$ low-latency access to SRAM.
  - if the data is *not* in fast memory $\rightarrow$ high-latency access to DRAM.
- **Memory hierarchies only work if the small, fast memory actually stores data that is reused by the processor.**

The Principle of Locality

Locality is the tendency of data accesses to be predictable. There are two kinds of locality:

1. **Spacial Locality**: The program is likely to access data that is close to data it has accessed already.
2. **Temporal Locality**: The program is likely to access the same data repeatedly.

Evidence of Locality



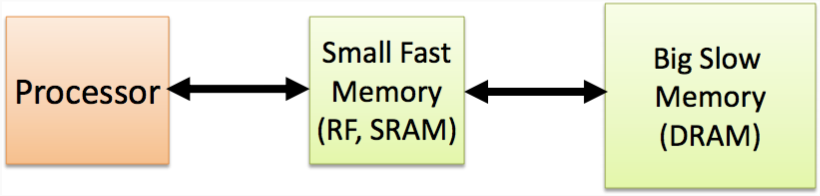
Time (one dot per access to that address at that time)

Locality in Action

Label each of the following accesses with whether it exhibits *temporal locality*, *spacial locality*, or neither.

- 1 — n
- 2 — s
- 3 — s
- 10 — n
- 4 — s
- 1800 — n
- 11 — s
- 30 — n
- 1 — t
- 2 — s,t
- 3 — s,t
- 4 — s,t
- 10 — s,t
- 190 — n
- 11 — s,t
- 30 — t
- 12 — s
- 13 — s
- 182 — n
- 1004 — n

Caches Exploit Both Types of Locality



http://www.cs.umd.edu/~meesh/411/CA-online/chapter/memory-hierarchy-design-basics/index.html

- Caches exploit **temporal locality** by remembering the contents of recently accessed locations.
- Caches exploit **spacial locality** by fetching blocks of data around recently accessed locations.

Four Fundamental Caching Questions

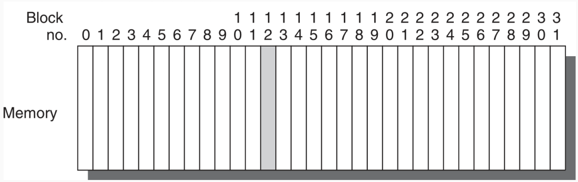
Four Fundamental Caching Questions

1. Where can a block be placed in the cache? (*block placement*)
2. How is a block found if it is in the cache? (*block identification*)
3. Which block should be replaced on a miss? (*block replacement*)
4. What happens on write? (*write strategy*)

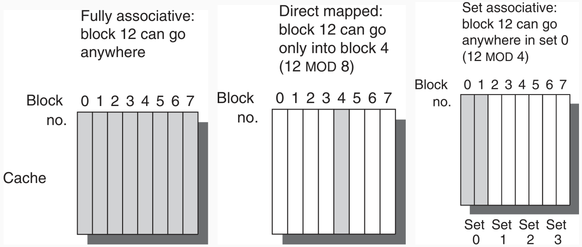
These questions arise because the cache is smaller than the main memory, so we can't store everything we might need at all times.

1. Where Can a Block be Placed in the Cache?

Given the following memory, where do we store block 12?



Three strategies for determining where to place that block:


Fully associative: block 12 can go anywhere
Block no.
0 1 2 3 4 5 6 7
Cache
Direct mapped: block 12 can go only into block 4 (12 MOD 8)
Block no.
0 1 2 3 4 5 6 7
Set associative: block 12 can go anywhere in set 0 (12 MOD 4)
Block no.
0 1 2 3 4 5 6 7
Set 0
Set 1
Set 2
Set 3

1. Where Can a Block be Placed in the Cache?

Note that fully associative and direct mapped are just special cases of the set-associative cache.

- A fully associative cache is just a cache where the associativity is the same as the size of the cache.
- A direct mapped cache is just a cache where the associativity is 1.

You may hear this referred to as an *N-way set-associative cache* where *N* is the associativity.

Practice: Block Placement: Worksheet Problem 1

In the following diagram of a 4-way set-associative cache with 32 blocks, highlight the areas of the cache where block 17 can be placed.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

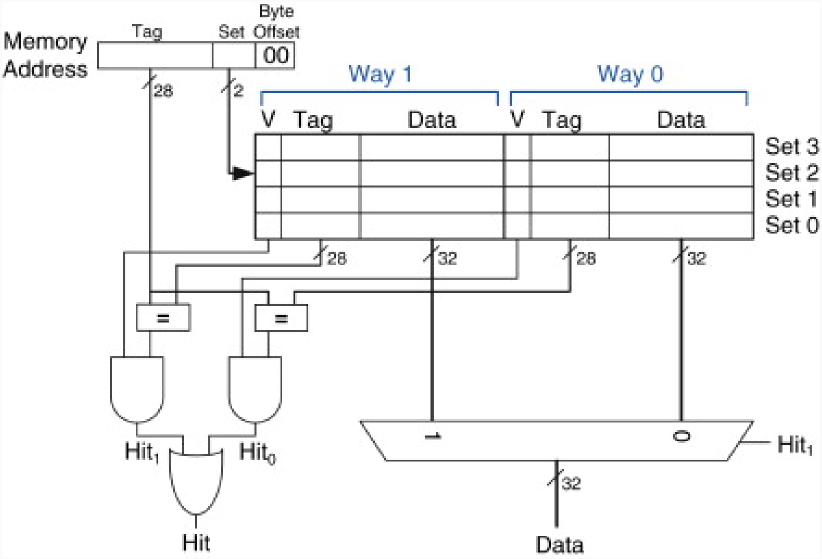
2. How is a Block Found if it is in the Cache?

Whenever a memory access happens, the address is split up into three parts: the *tag*, *index*, and *offset*.

Block address		Block **offset**
Tag	**Index**	

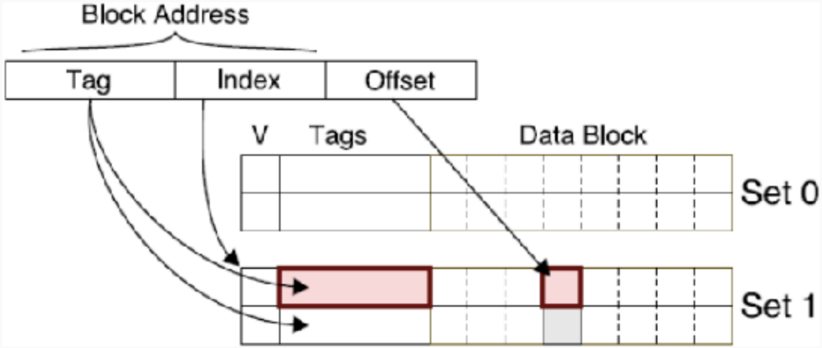
- The *index* is used to find the set where a *potential match* may reside.
- Then, we check if the *tag* is in the set in parallel.
- If it matches, check if the cache line is valid by looking at the *valid bit*.
- If it matches and is valid, then the *offset* is used to index into the block.

2. How is a Block Found if it is in the Cache?



https://www.sciencedirect.com/topics/computer-science/set-associative-cache

2. Dealing with the Offset



- The cache line contains more than one byte, making the offset necessary.

2. How Many Bits for Tag, Index, and Offset?

The number of bits required for the tag, index, and offset depends on the **cache geometry**.

Cache Geometry Formulas

Let L be the # cache lines, B be the cache line size, A be the address length (32 bits in our case), and W be the associativity. Then,

$$\begin{aligned}\text{Index bits} &= \log_2(L/W)\\ \text{Offset bits} &= \log_2(B)\\ \text{Tag bits} &= A - (\text{Index bits} + \text{Offset bits})\end{aligned}$$

For this class, always assume that $A = 32$ bits unless otherwise specified.

Practice: Cache Geometry: Worksheet Problem 2

Calculate the number of bits required for the *index*, *offset*, *tag* for a *direct-mapped cache* with **1024 cache lines** and **32 bytes per line**.

Practice: Cache Geometry: Worksheet Problem 3

Calculate the number of bits required for the *index*, *offset*, and *tag* for a **32 KiB direct-mapped cache** with **64-byte cache lines**.

Practice: Cache Geometry: Worksheet Problem 4

Calculate the number of bits required for the *index*, *offset*, and *tag* for a **32 KiB** cache with **2048 lines** that is **4-way associative**.

3. Which Block Should be Replaced on a Miss?

If the data we want isn't in the cache, we may have to *evict* a line from the cache to make room for the new data. How we make this choice is called the *cache eviction policy*.

- **Random** — uniformly evict lines. Easy to implement!
- **Least Recently Used (LRU)** — evict the line that was accessed the longest time ago.
- **Prefer Clean** — try to evict clean lines to avoid write-back. (More on this in a minute.)
- **Farthest further use** — evict the line whose next access is farthest in the future. This is provably optimal. It's also impossible to implement.

4. What Happens on Write?

When a write occurs, we have two main decisions:

1. Do we write to just the cache or to the entire memory hierarchy?
2. Should we pull the cache line in to the cache if there's a cache miss?

4.1. Just Write to Cache or to the Entire Memory Hierarchy?

When we perform a write, we have two options:

1. Just update the cache.
2. Update the cache and also forward the write to lower cache levels.

If we *do not* forward the write, the cache is **write back** since the data must be written back on eviction (the cache line can be *dirty*).

Advantages: Fewer writes farther down the memory hierarchy. Less bandwidth. Faster writes.

If we *do* forward the write, the cache is **write through**. In this case, a cache line is never dirty.

Advantages: No write back overhead required on eviction.

4.2. Pull the Cache Line in on Write Miss?

When we perform a write and the cache line that we are writing to is not in cache (that is, we have a *write miss*), we have two options:

1. Pull in the cache line to the cache. This is called **write allocate**.
2. Do not pull the cache line into the cache. This is called **no-write allocate.**

Write Allocate Advantages: Exploits temporal locality. Data written will likely be read soon, so that read will be faster.

No-Write Allocate Advantages: Fewer unnecessary evictions. If the data is not read in the near future, the eviction is a waste.

Some Tradeoffs to Consider

The Cost of Associativity

Increasing associativity requires more tag checks.

- N-way associativity requires N parallel comparators
- This is expensive in hardware and potentially slow

This limits associativity of L1 caches to 2-8.

Larger, slower caches can be more associative.

Examples:

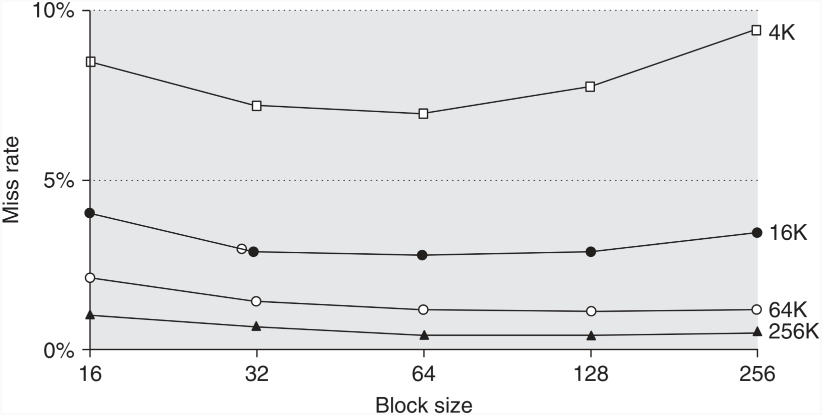
- Intel Nehalem processors have an **8-way L1 cache** and **18-way L2 and L3 caches**.
- Core 2's L2 was 24-way.

Cache Line Size

How big should a cache line be?

- Why is bigger better?
  - Exploits more spacial locality
  - Large cache lines effectively *prefetch* data that we have not explicitly asked for.
- Why is smaller better?
  - Focuses on temporal locality
  - If there is a little spacial locality, large cache lines waste space and bandwidth.
- In practice, 32-64 bytes is pretty good for L1 caches where space is scarce and latency is important.
- Lower level caches use 128-256 bytes.

Cache Line Size Affects Miss Rate



Page B-27

Data and Instruction Cache

Most processor shave two different caches, one for instructions (I) and one for data (D). **Why?**

- Different areas of memory.
- Different access patterns.
  - I-cache accesses have lots of *spacial* locality because most of the time instructions are accessed sequentially.
  - I-cache accesses are also predictable to the extent that branches are predictable.
  - D-cache accesses are typically less predictable.
- Not just different, but could interfere with one another!
  - Sequential instruction accesses could interfere with the data accesses.
  - Separating them helps eliminate this issue.