Datapath Q&A

What is the role of the ALU in lw, sw, lb, and sb?

Instruction opcode tells us we have an lw instruction.
Your microprogram configures the ALU to perform addition.

How should we test lb and sb?

Probably just a very simple test for now.

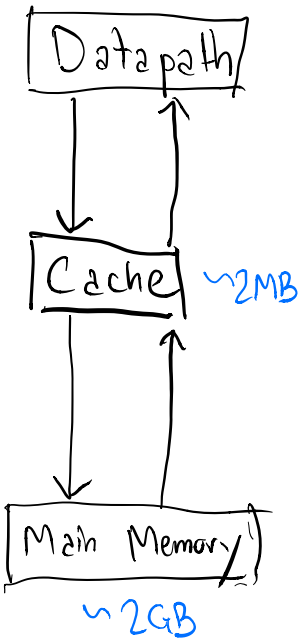
Dilemma: we want memory to be both fast and large. Large memories are inherently slower than small ones.

Why do we have this trade-off?

- small memories can be closer to the datapath, so there's less travel time. The physical size of the memory itself can slow it down too.
- decoding / looking up entries in large memory takes longer
- large memories usually use less fast technology like DRAM, while small memory can use SRAM.

The fix: caching

Caching



Any time the datapath accesses memory, look in the cache first.

Cache hit: the cache contained the memory location we were looking for.

Cache miss: the cache did <u>not</u> contain the location we needed.

What makes caching effective?

Temporal locality - when a program accesses a location in memory, it is likely to access that location again soon.

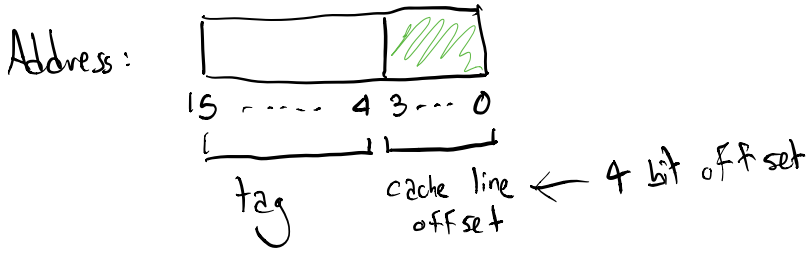
Spatial locality - when a program accesses a location in memory, it is likely to access <u>nearby</u> locations soon.

Caching Policy:

When the datapath accesses a memory location, keep it in the cache - this exploits temporal locality.

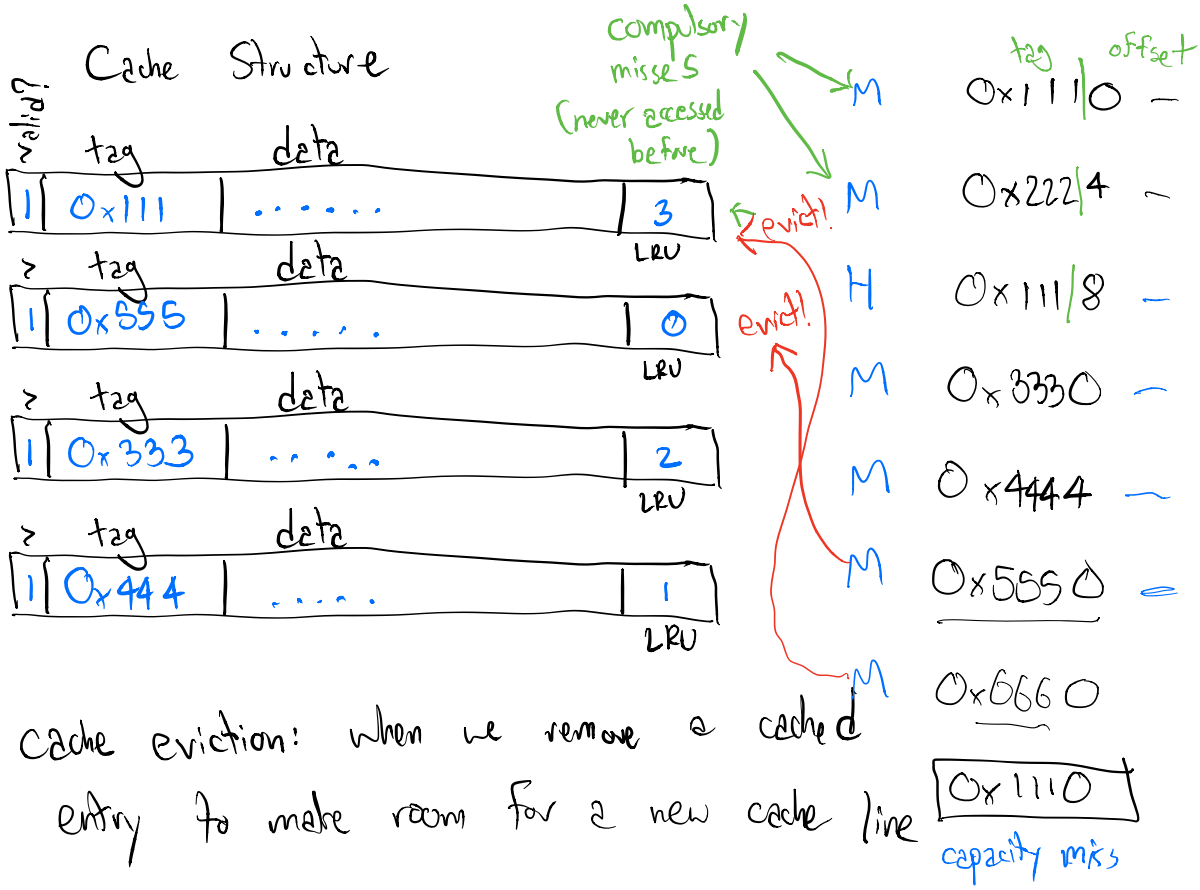
When we cache one location, also cache nearby locations - this exploits spatial locality.

Cache Design: Fully Associative Cache



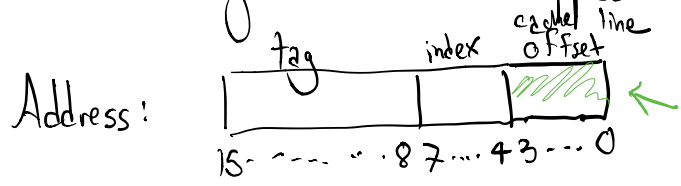
When we store data in the cache, we also store nearby data. A cache line is a collection of adjacent bytes whose addresses have the same tag.

A cache line is a sequence of bytes we cache together, in this case it is 16 bytes (2^4).



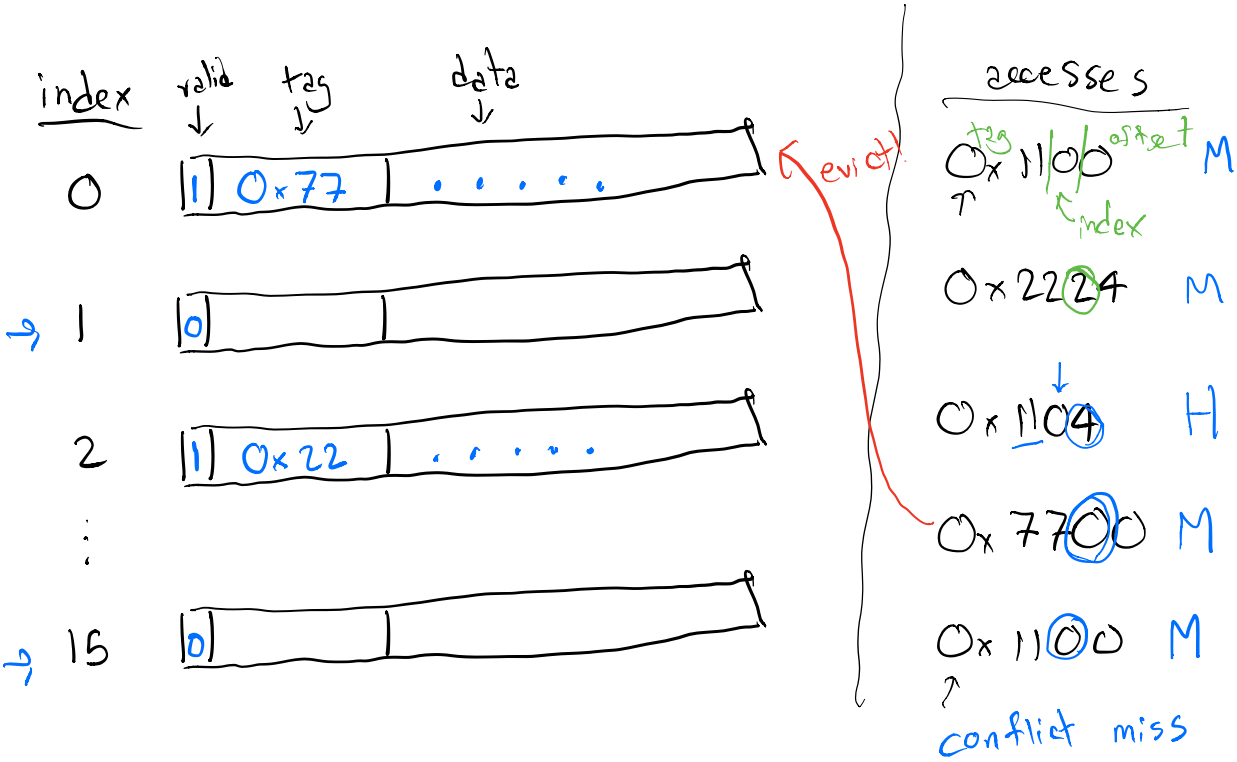
Cache eviction: when we remove a cached entry to make room for a new cache line

Cache Design: Direct Mapped Cache



For a given address, a direct-mapped cache has exactly one place to cache the data loaded from that address.

4 index bits → 16 entries in the cache (2^4)



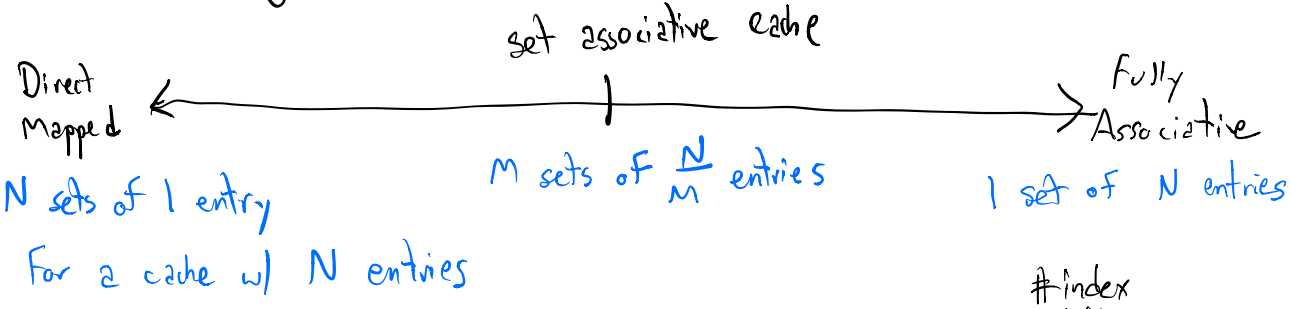
Advantages over fully-associative:

+ no LRU logic required
+ large cache still has simple lookup logic (scales well)

Disadvantages relative to fully-associative:

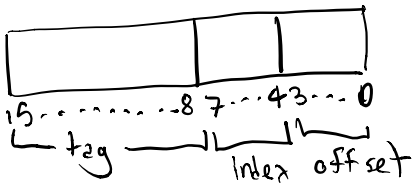
- accessing two things w/ same index → repeated evictions (thrashing)
- direct mapped caches don't do as good a job at exploiting temporal locality

Cache Design: Set-Associative Cache



$$\#sets = 2^{\#index\ bits}$$

Address:



2-way set associative cache