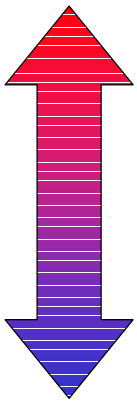
10/18: Lecture topics

- Memory Hierarchy
  - Why it works: Locality
  - Levels in the hierarchy
- Cache access
  - Mapping strategies
- Cache performance
- Replacement policies

Types of Storage

- Registers
- On-chip cache(s)
- Second level cache
- Main Memory
- Disk
- Tape, etc.



fast, small, expensive

slow, large, cheap

The Big Idea

- Keep all the data in the big, slow, cheap storage
- Keep copies of the "important" data in the small, fast, expensive storage

- The *Cache Inclusion Principle:* If cache level B is lower than level A, B will contain all the data in A.

Some Cache Terminology

- Cache *hit rate:* The fraction of memory accesses found in a cache. When you look for a piece of data, how likely are you to find it in the cache?
- *Miss rate:* The opposite. How likely are you not to find it?
- *Access time:* How long does it take to fetch data from a level of the hierarchy?

Effective Access Time

$$t = ht_c + (1-h)t_m$$

effective access time

memory access time

cache hit rate

cache access time

cache miss rate

Goal of the memory hierarchy: storage as big as the lowest level, effective access time as small as the highest level

Access Time Example

- Suppose t_m for disk is 10 ms = 10^{-2} s.
- Suppose t_c for main memory is 50 ns = 5×10^{-8} s.
- We want to get an effective access time t right in between, at 10^{-5} s.
- What hit rate h do we need?

The Moral of the Story

- The "important" data better be *really* important!
- How can we choose such valuable data?
- But it gets worse: the valuable data will change over time
  - Answer: move new important data into the cache, *evict* data that is no longer important
  - By the cache inclusion principle, it's OK just to throw data away

Temporal Locality

- *Temporal* = having to do with time
- *temporal locality:* the principle that data being accessed now will probably be accessed again soon
- Useful data tends to continue to be useful

Spatial Locality

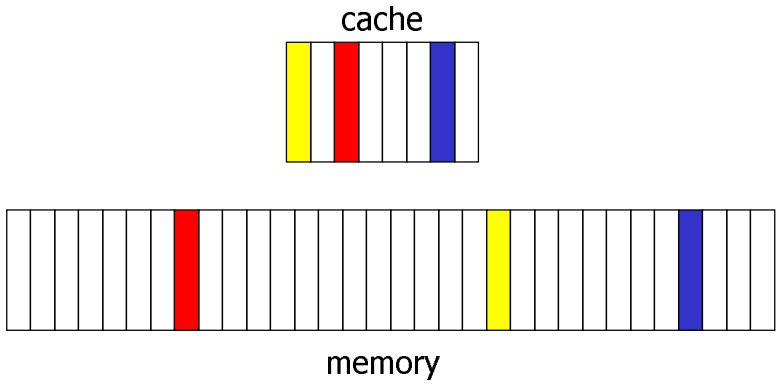
- *Spatial:* having to do with space -- or in this case, proximity of data
- *Spatial locality:* the principle that data near the data being accessed now will probably be needed soon
- If data item *n* is useful now, then it's likely that data item *n+1* will be useful soon

Applying Locality to Cache Design

- On access to data item *n*:
  - Temporal locality says, "Item *n* was just used. We'll probably use it again soon. Cache *n*."
  - Spatial locality says, "Item *n* was just used. We'll probably use its neighbors soon. Cache *n+1*."
- The principles of locality give us an idea of which data is important, so we know which data to cache.

Concepts in Caching

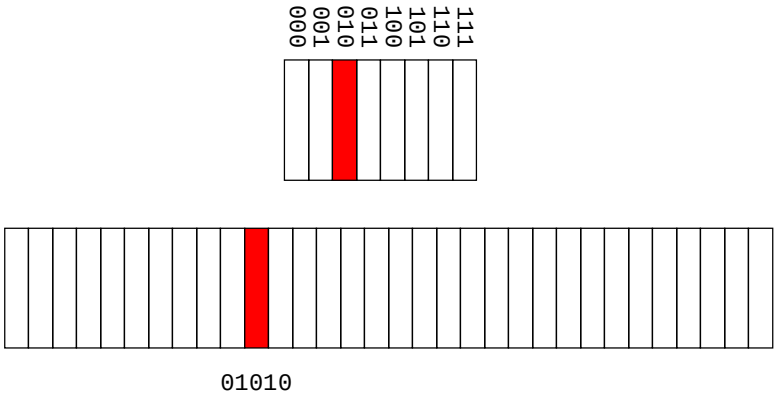
- Assume a two level hierarchy:
  - Level 1: a cache that can hold 8 words
  - Level 2: a memory that can hold 32 words



Direct Mapping

- Suppose we reference an item.
  - How do we know if it's in the cache?
  - If not, we should add it. Where should we put it?
- One simple answer: *direct mapping*
  - The address of the item determines where in the cache to store it
  - In this case, the lower three bits of the address dictate the cache entry

Direct Mapping Example



Issues with Direct Mapping

- How do you tell if the item cached in slot 101 came from 00101, 01101, etc?
  - Answer: Tags
- How can you tell if there's any item there at all?
  - Answer: the Valid bit
- What do you do if there's already an item in your slot when you try to cache a new item?

Tags and the Valid Bit

- A *tag* is a label for a cache entry indicating where it came from
  - The upper bits of the data item's address
- The *valid bit* is a bit indicating whether a cache slot contains useful information
- A picture of the cache entries in our example:

vb	tag	data

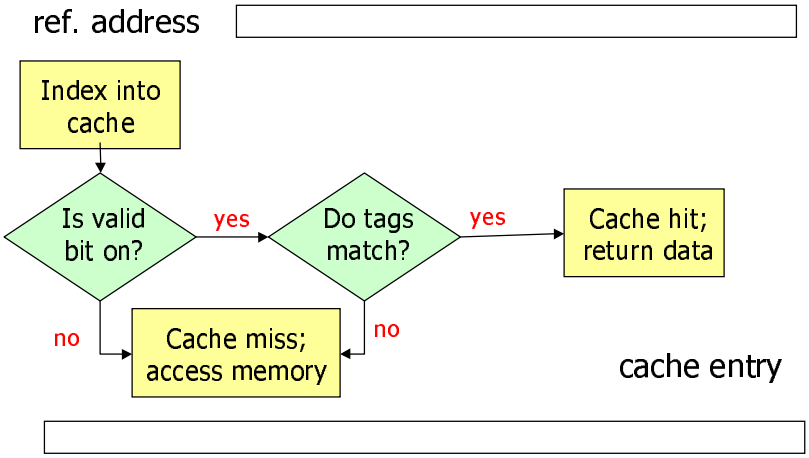
Reference Stream Example

index	vb	tag	data
000			
001			
010			
011			
100			
101			
110			
111			

11010, 10111, 00001, 11010,
11011, 11111, 01101, 11010

Cache Lookup

- Return to 32 bit addresses, 4K cache



i-cache and d-cache

- There are two separate caches for instructions and data. Why?
  - Avoids structural hazards in pipelining
  - Reduces contention between instruction data items and data data items
  - Allows both caches to operate in parallel, for twice the bandwidth

Handling i-Cache Misses

1. Send the address of the missed instruction to the memory
2. Instruct memory to perform a read; wait for the access to complete
3. Update the cache
4. Restart the instruction, this time fetching it successfully from the cache

d-Cache misses are even easier

Exploiting Spatial Locality

- So far, only exploiting temporal locality
- To take advantage of spatial locality, group data together into *blocks*
- When one item is referenced, bring it and its neighbors into the cache together
- New picture of cache entry:

vb	tag	$data_0$	$data_1$	$data_2$	$data_3$

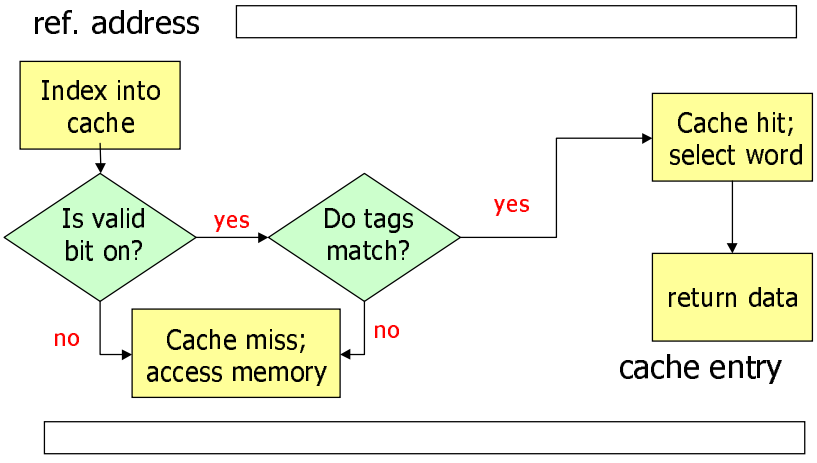
Another Reference Stream Example

index	vb	tag	data			
00						
01						
10						
11						

11010, 10111, 00001, 11010,
11011, 11111, 01101, 11010

Revisiting Cache Lookup

- 32 bit addr., 64K cache, 4 words/block



The Effects of Block Size

- Big blocks are good
  - Reduce the overhead of bringing data into the cache
  - Exploit spatial locality
- Small blocks are good
  - Don't evict so much other data when bringing in a new entry
  - More likely that all items in the block will turn out to be useful
- How do you choose a block size?

Associativity

- Direct mapped caches are easy to understand and implement
- On the other hand, they are restrictive
- Other choices:
  - Set-associative: each block may be placed in a set of locations, perhaps 2 or 4 choices
  - Fully-associative: each block may be placed anywhere

Full Associativity

- The cache placement problem is greatly simplified: place the block anywhere!
- The cache lookup problem is much harder
  - The entire cache must be searched
  - The tag for the cache entry is now much longer
- Another option: keep a lookup table

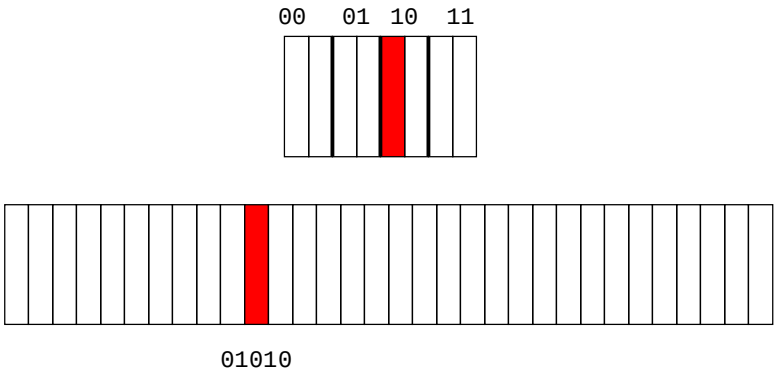
Lookup Tables

- For each block,
  - Is it currently located in the cache?
  - If so, where
- Size of table: one entry for each block in memory
- Not really appropriate for hardware caches (the table is too big)
  - Fully associative hardware caches use linear search (slow when cache is big)

Set Associativity

- More flexible placement than direct mapping
- Faster lookup than full associativity
- Divide the cache into *sets*
  - In a 2-way set-associative cache, each set contains 2 blocks
  - In 4-way, each set contains 4 blocks, etc.
- Address of block governs which set block is placed in
- Within set, placement is flexible

Set Associativity Example



Reads vs. Writes

- Caching is essentially making a copy of the data
- When you read, the copies still match when you're done
- When you write, the results must eventually propagate to both copies
  - Especially at the lowest level, which is in some sense the permanent copy

Write-Back Caches

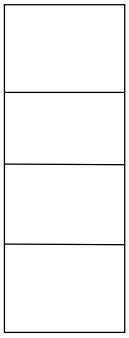
- Write the update to the cache only. Write to the memory only when the cache block is evicted.
- Advantages:
  - Writes go at cache speed rather than memory speed.
  - Some writes never need to be written to the memory.
  - When a whole block is written back, can use high bandwidth transfer.

Cache Replacement

- How do you decide which cache block to replace?
- If the cache is direct-mapped, easy.
- Otherwise, common strategies:
  - Random
  - Least Recently Used (LRU)
  - Other strategies are used at lower levels of the hierarchy. More on those later.

LRU Replacement

- Replace the block that hasn't been used for the longest time.



Reference stream:

A B C D B D E B A C B C E D C B

LRU Implementations

- LRU is very difficult to implement for high degrees of associativity
- 4-way approximation:
  - 1 bit to indicate least recently used pair
  - 1 bit per pair to indicate least recently used item in this pair
- Much more complex approximations at lower levels of the hierarchy

Write-Through Caches

- Write the update to the cache and the memory immediately
- Advantages:
  - The cache and the memory are always *consistent*
  - Misses are simple and cheap because no data needs to be written back
  - Easier to implement

The Three C's of Caches

- Three reasons for cache misses:
  - *Compulsory* miss: item has never been in the cache
  - *Capacity* miss: item has been in the cache, but space was tight and it was forced out
  - *Conflict* miss: item was in the cache, but the cache was not associative enough, so it was forced out

Multi-Level Caches

- Use each level of the memory hierarchy as a cache over the next lowest level
- Inserting level 2 between levels 1 and 3 allows:
  - level 1 to have a higher miss rate (so can be smaller and cheaper)
  - level 3 to have a larger access time (so can be slower and cheaper)
- The new effective access time equation:

Summary: Classifying Caches

- Where can a block be placed?
  - Direct mapped: one place
  - Set associative: perhaps 2 or 4 places
  - Fully associative: anywhere
- How is a block found?
  - Direct mapped: by index
  - Set associative: by index and search
  - Fully associative:
    - search
    - lookup table

Summary, cont.

- Which block should be replaced?
  - Random
  - LRU (Least Recently Used)
- What happens on a write access?
  - Write-back: update cache only; leave memory update until block eviction
  - Write-through: update cache and memory