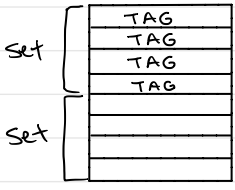
Set Associative Cache:

- Main memory addresses broken into tag, selector, block offset bits
- Each set holds more than one line.
- Any line in a set can hold any block that maps to that set.

4 way set Associative



*Hardware is going to do a parallel search.

*Replacement Policy: select victim block to evict { 1. Random Selection, 2. Get rid of least recently used

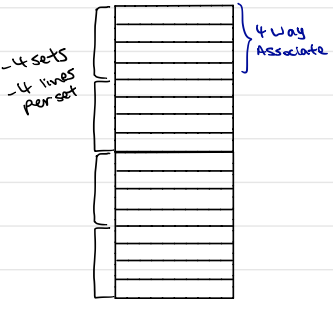
*Better than direct-mapped, b/c less bits needed for selector bits, but way more complicated.

08/02

- Cache tries to find a block that matches an address
- If the requested block is in the cache → "HIT"
- If the requested block is not in the cache → "Miss"
- # of Hits/ #Total requestes = Hit Ratio.

* Goal - to maximize hit ratio.

Set Associative Cache



- Selector bits from address determine set.
- Any line in the set will do.
- Parallel search within a set for matching tag bits.
- Hardware parallel search
- Still some potential underutilization.
- Because selector bits associate to specific sets.

*Direct map cache use underutilization

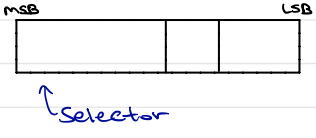
Fully-Associate Cache:

- Only one set
- Different addresses may go to any line in the set
- Find Hit or Miss by linear search thru cache
- Breaks main memory address into tag bits and block offset (No selector bits)
- Finds Hit or Miss with (Expensive) $$ HW Parallel Search

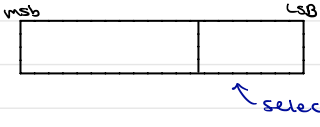
Fully Associative Cache

TAG
TAG
TAG
TAG
TAG
TAG
TAG
TAG
TAG

* Least likely to thrash because any block can go anywhere in the cache.
* Best used for caches with a small number of blocks.



Bad Idea:
Entire address space is not always used, leaving selector bits. The same for all addresses in the process.



Bad Idea:
Adjacent memory addresses would be in diff. cache block

Direct Mapped	VS	Fully Associative
• Same selector bits		• No selector bits
• Always go to same line		• Line could go anywhere in the cache

Cache design is affected by...

1. Cache size
2. Block size (Principle of locality)
3. Replacement Algorithm - determine cache efficiency
4. Write Policy

Block Replacement Policies:

Least Recently Used (LRU)

- Each block has an associated counter, initially set to Zero
- There is a single global (within the cache) counter initially set to Zero

Global Counter is incremented by 1 for every memory access.

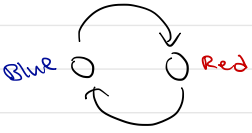
- If the requested block is already in the cache (as determined by the tag bits), then set the counter for the block to the global counter.
- If the requested block is not in the cache, then evict the block with the lowest block counter value.

Beating Overflow:

- Option #1:
  - N blocks numbered 1 to N
  - Access any block K then
    - if counter for block K is already N, do nothing.
    - if counter for block K is not N, then set counter for block K to N and decrement all other block counter by 1.
- Option #2:
  - Red-Blue infinite sequencing
  - All counters now have a "color" - Red or Blue
  - Initial value is Blue 0.
  1) Increment Blue until overflow, then value is Red.
  2) Increment Red until overflow, then value is Blue.

Red-Blue Comparison Rules

- Small Blue < Large Blue
- Large Blue < Small Red
- Small Red < Large Red
- Large Red < Small Blue



*Small & Large as in value of the number.

*Know how to do algebra with this formula

Capacity $C = S * E * B$

Block size in bytes

Number of sets

Number of lines per set

$\rightarrow S = 2^z$ (z = # of selector bits)

What about writing to cache and memory?

Two Policies:

1. **Write-Through**: All data writes sent immediately to main memory
   - Pro: Easy to implement. Data in the cache & Memory always consistent. Good if Reads greatly outnumber Writes.
   - Cons: Slow for lots of writes.
2. **Write-Back**:
   - Write the cache block back to main memory only when the block is evicted.
   - Cache blocks with changed contents are "Dirty". Unchanged blocks are "Clean".

   *Change copy in the cache
   - Pros: Faster
   - Cons: Possible data inconsistency if program crashes before updating memory.

Parallel Processors:

- Multiple CPUs sharing a common memory.
- Each processor has its own L1 cache in the CPU.
- There are other caches (L2, L3) between the CPUs and the common memory.
- Now the caches have to communicate with each other when data is written.

What about writing to an address that is not in the cache?

1. **Write-Allocate**: Load the missing block into the cache and write to the cached block.
   - Pros: Takes advantage of spacial locality.
   - Cons: Every miss caused a read from main memory into the cache.

2. **No-Write-Allocate**: missed writes go directly to main memory without modifying the cache
   - Pros: No read from main memory for uncached writes
   - Cons: Does not take advantage of spatial locality.

* Write-Allocate vs No-Write-Allocate decision is orthogonal (/independent) to write-Through vs Write-Back.

{ Write-Back Cache is typically Write-Allocate.
{ Write-Through Cache is typically No-Write-Allocate.

Memory Hierarchy:

- Lower levels tend to be Write-Back because of larger blocks and longer transfer times.
- Higher levels tend to be Write-Through but write-back is becoming more attractive as chip density increases.

* Maximize spatial locality by accessing adjacent memory in cache. (sequentially)
* Maximize temporal locality by reusing data objects as much as possible.