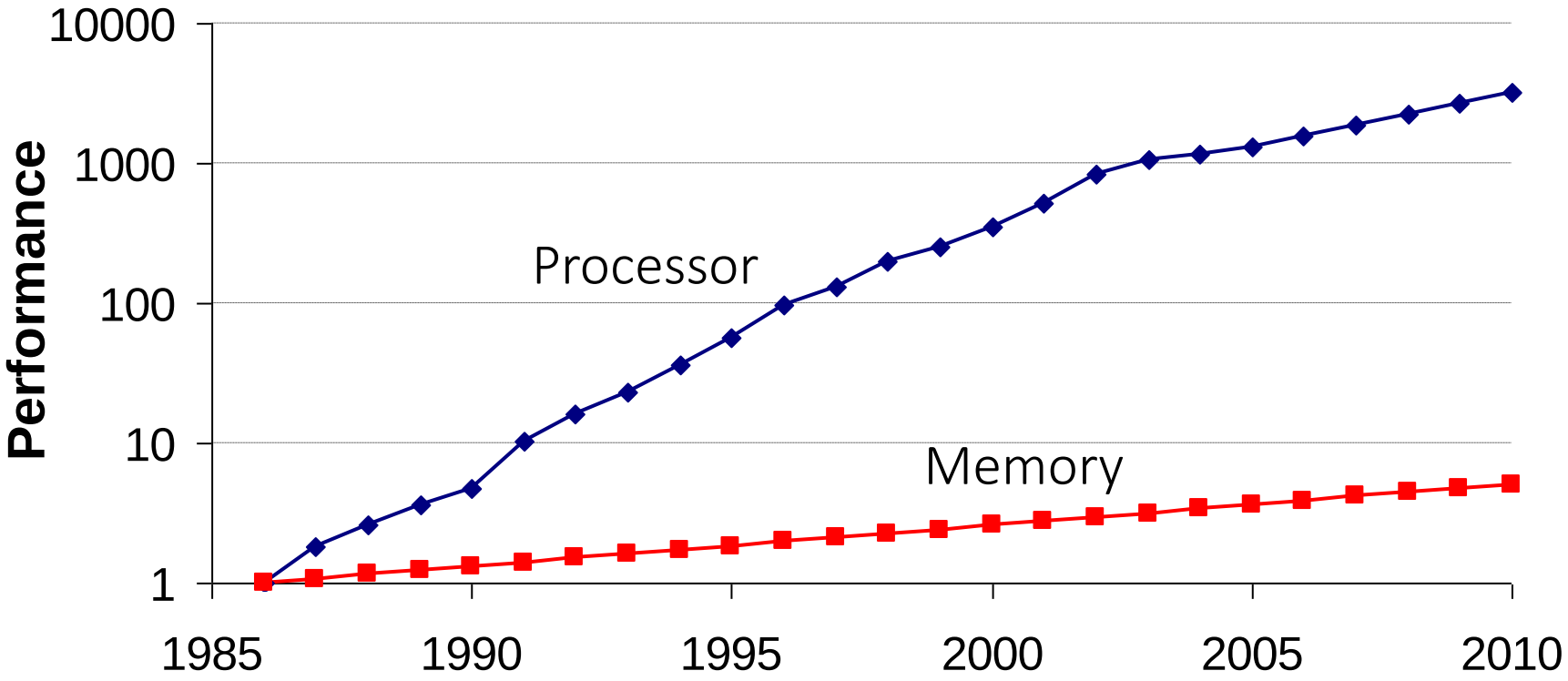
Caches

Nima Honarmand

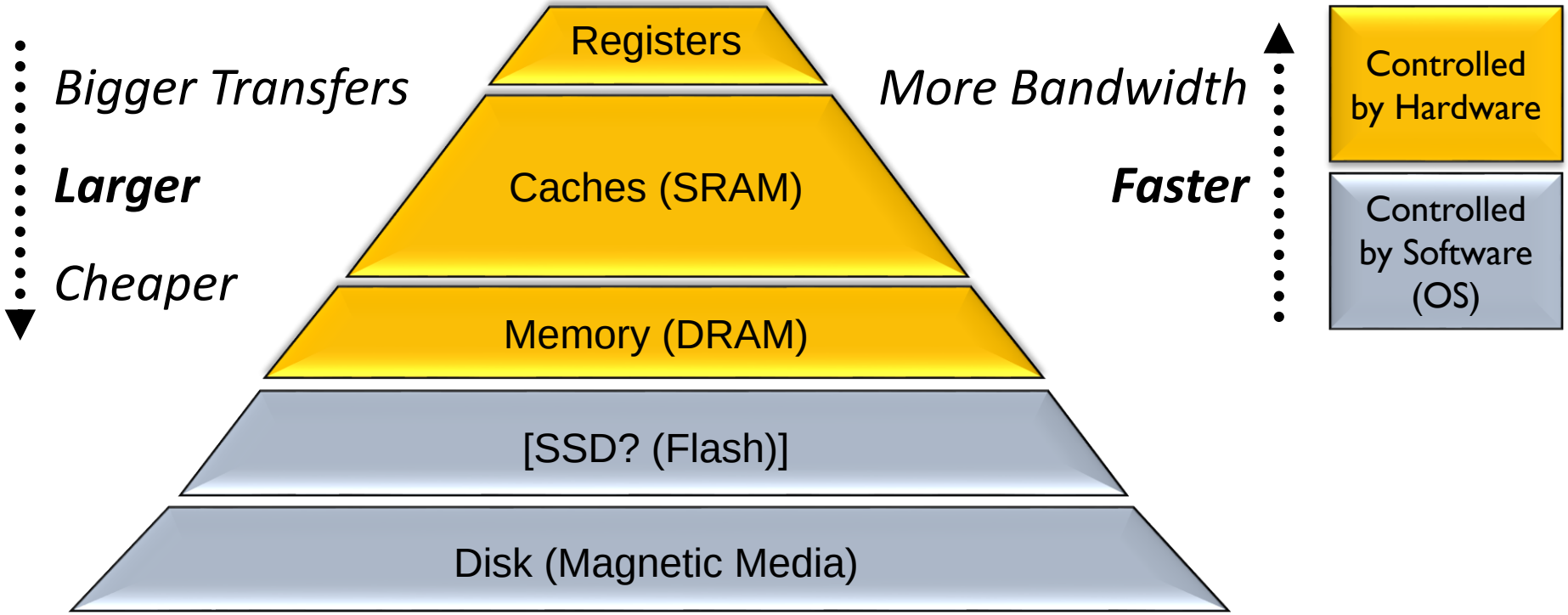
Motivation



- Want memory to appear:
  - As fast as CPU
  - As large as required by all of the running applications

Storage Hierarchy

- Make common case fast:
  - Common: temporal & spatial locality
  - Fast: smaller more expensive memory



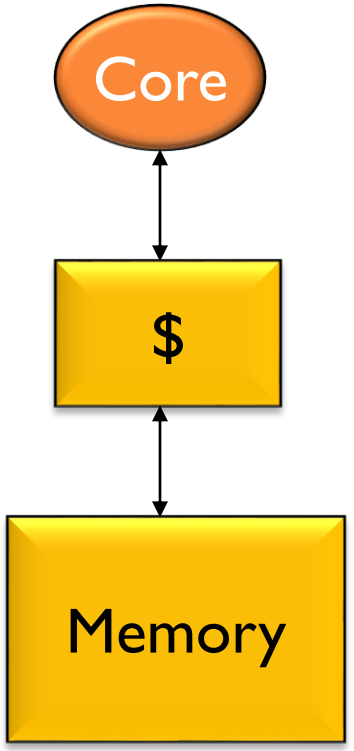
What is **S**(tatic)RAM vs **D**(ynamic)RAM?

Caches

- An ***automatically managed*** hierarchy

- Break memory into *blocks* (several bytes) and transfer data to/from cache in blocks
  - *spatial locality*

- Keep recently accessed blocks
  - *temporal locality*



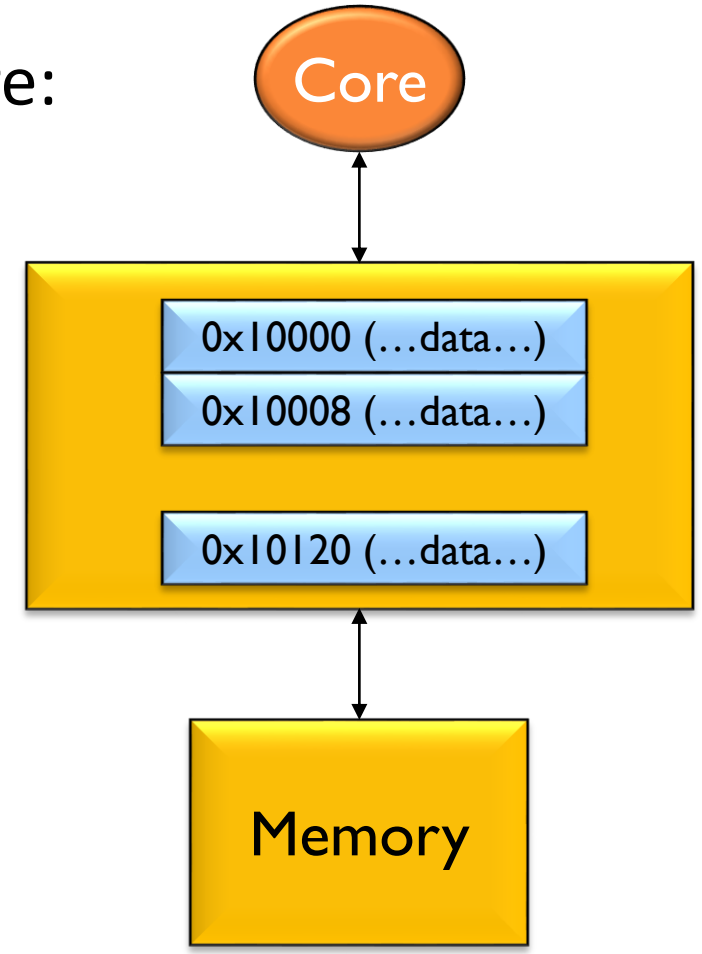
Cache Terminology

- *block* (*cache line*): minimum unit that may be cached
- *frame*: cache storage location to hold one block
- *hit*: block is found in the cache
- *miss*: block is not found in the cache
- *miss ratio*: fraction of references that miss
- *hit time*: time to access the cache
- *miss penalty*: time to replace block on a miss

Cache Example

- Address sequence from core: (assume 8-byte lines)

0x10000	Miss
0x10004	Hit
0x10120	Miss
0x10008	Miss
0x10124	Hit
0x10004	Hit



Final *miss ratio* is 50%

Average Memory Access Time (1/2)

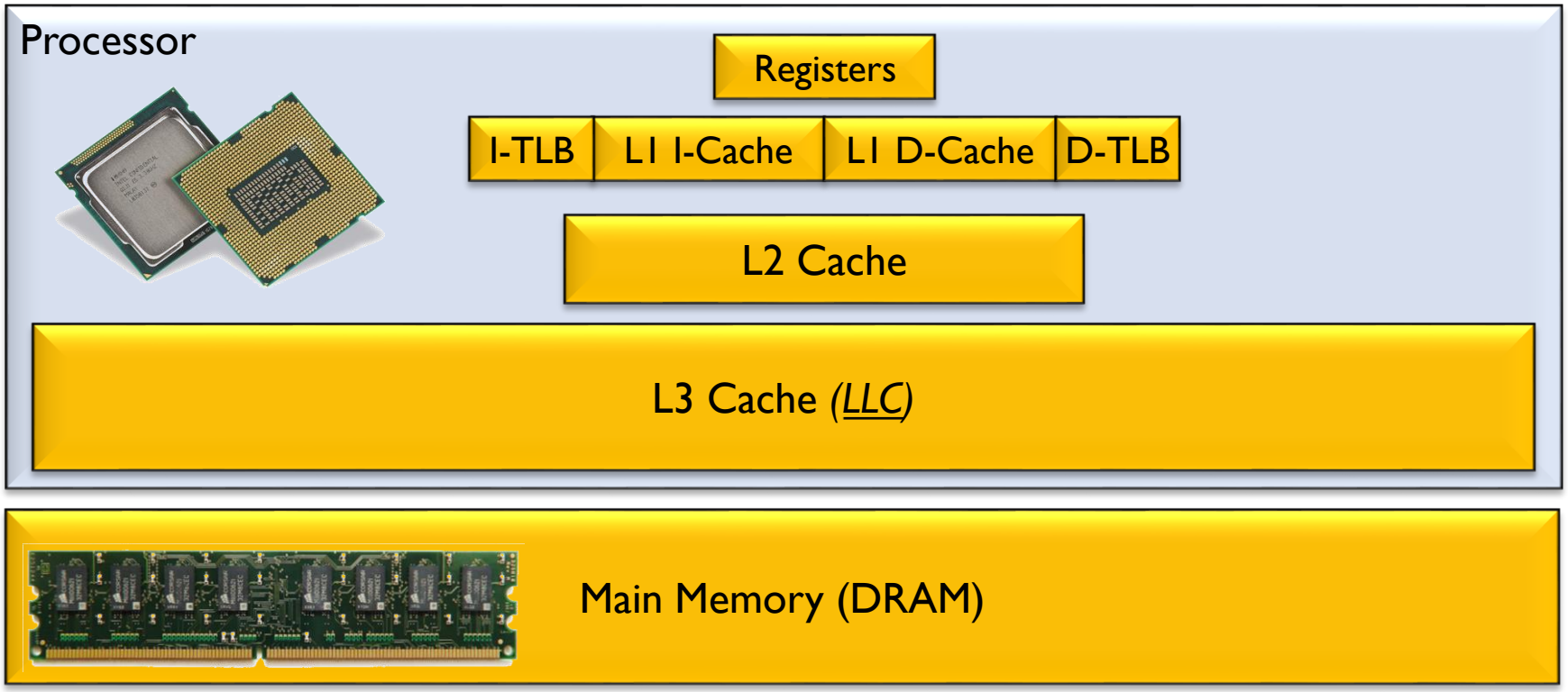
- Or AMAT
- Very powerful tool to estimate performance
- If …
  cache hit is 10 cycles (core to L1 and back)
  memory access is 100 cycles (core to mem and back)
- Then …
  at 50% miss ratio, avg. access: 0.5×10+0.5×100 = 55
  at 10% miss ratio, avg. access: 0.9×10+0.1×100 = 19
  at 1% miss ratio, avg. access: 0.99×10+0.01×100 ≈ 11

Average Memory Access Time (2/2)

- Generalizes nicely to hierarchies of any depth
- If …
  L1 cache hit is 5 cycles (core to L1 and back)
  L2 cache hit is 20 cycles (core to L2 and back)
  memory access is 100 cycles (core to mem and back)
- Then …
  at 20% miss ratio in L1 and 40% miss ratio in L2 …
  avg. access: $0.8\times5+0.2\times(0.6\times20+0.4\times100) \approx 14$

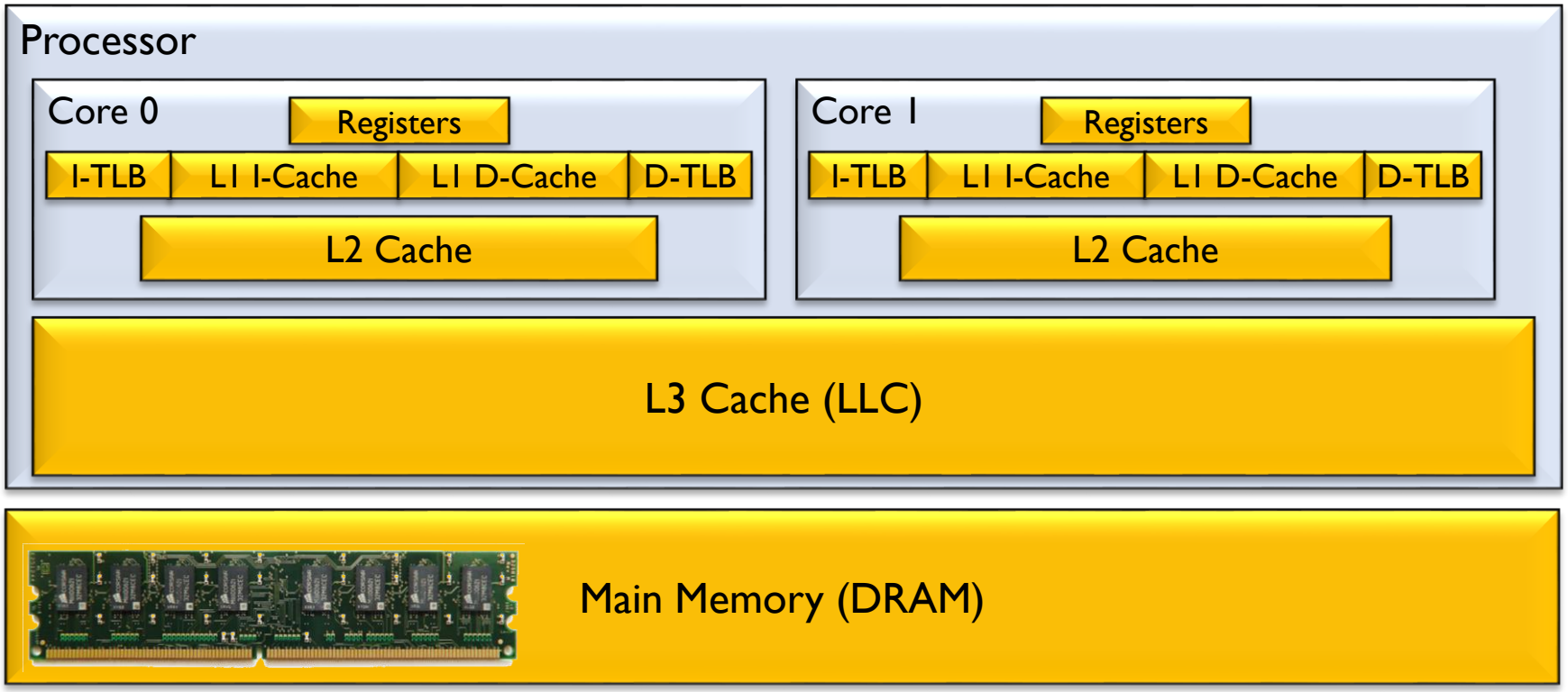
Memory Organization (1/3)

- L1 is *split* — separate I$ (inst. cache) and D$ (data cache)
- L2 and L3 are *unified*



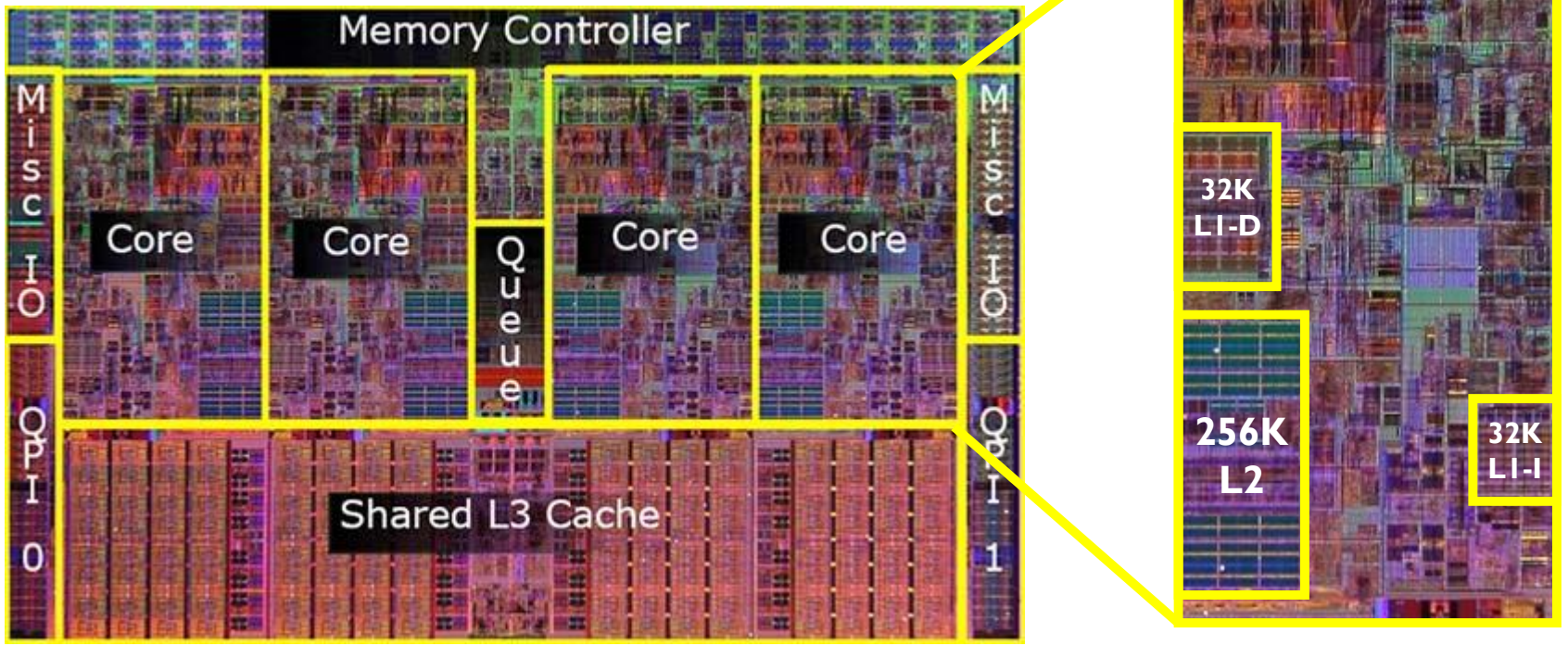
Memory Organization (2/3)

- L1 and L2 are *private*
- L3 is *shared*



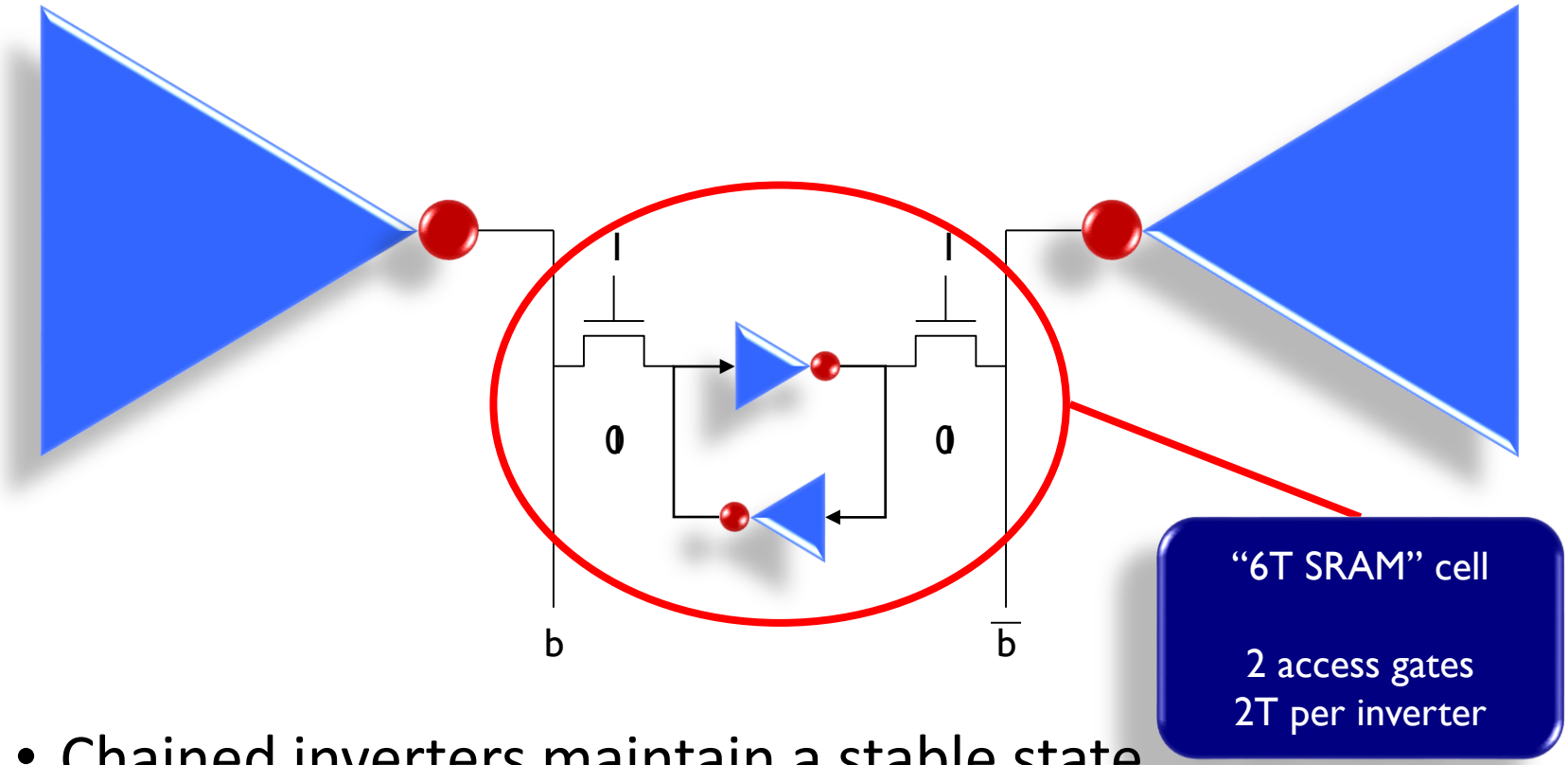
Multi-core replicates the top of the hierarchy

Memory Organization (3/3)



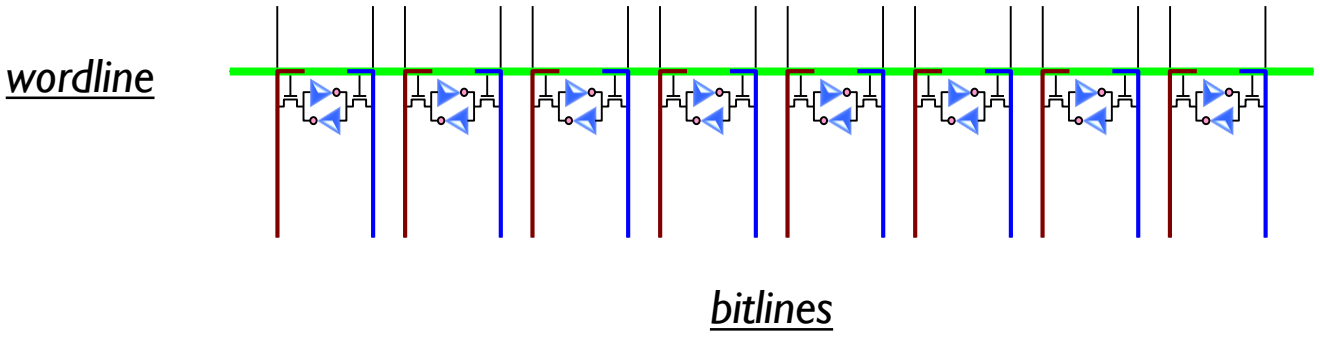
Intel Nehalem
(3.3GHz, 4 cores, 2 threads per core)

SRAM Overview

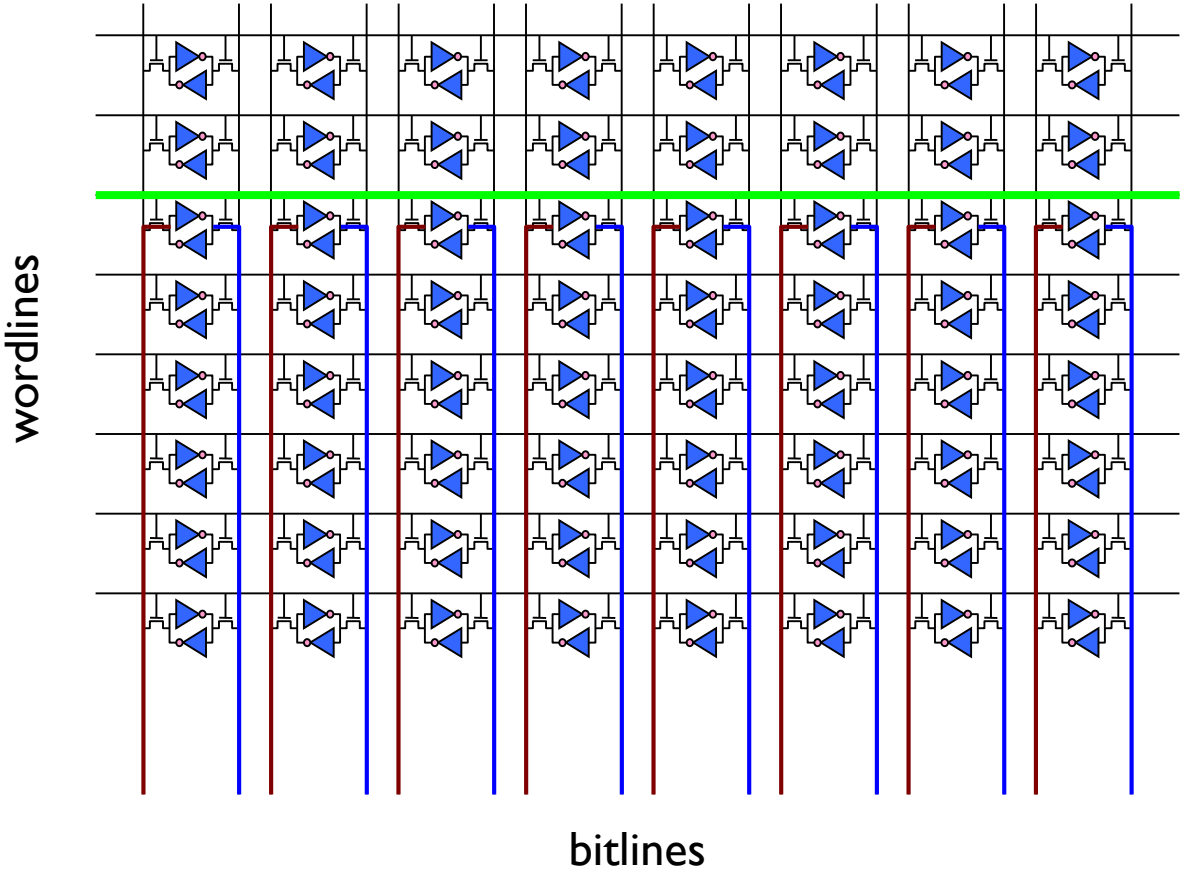


- Chained inverters maintain a stable state
- Access gates provide access to the cell
- Writing to cell involves over-powering storage inverters

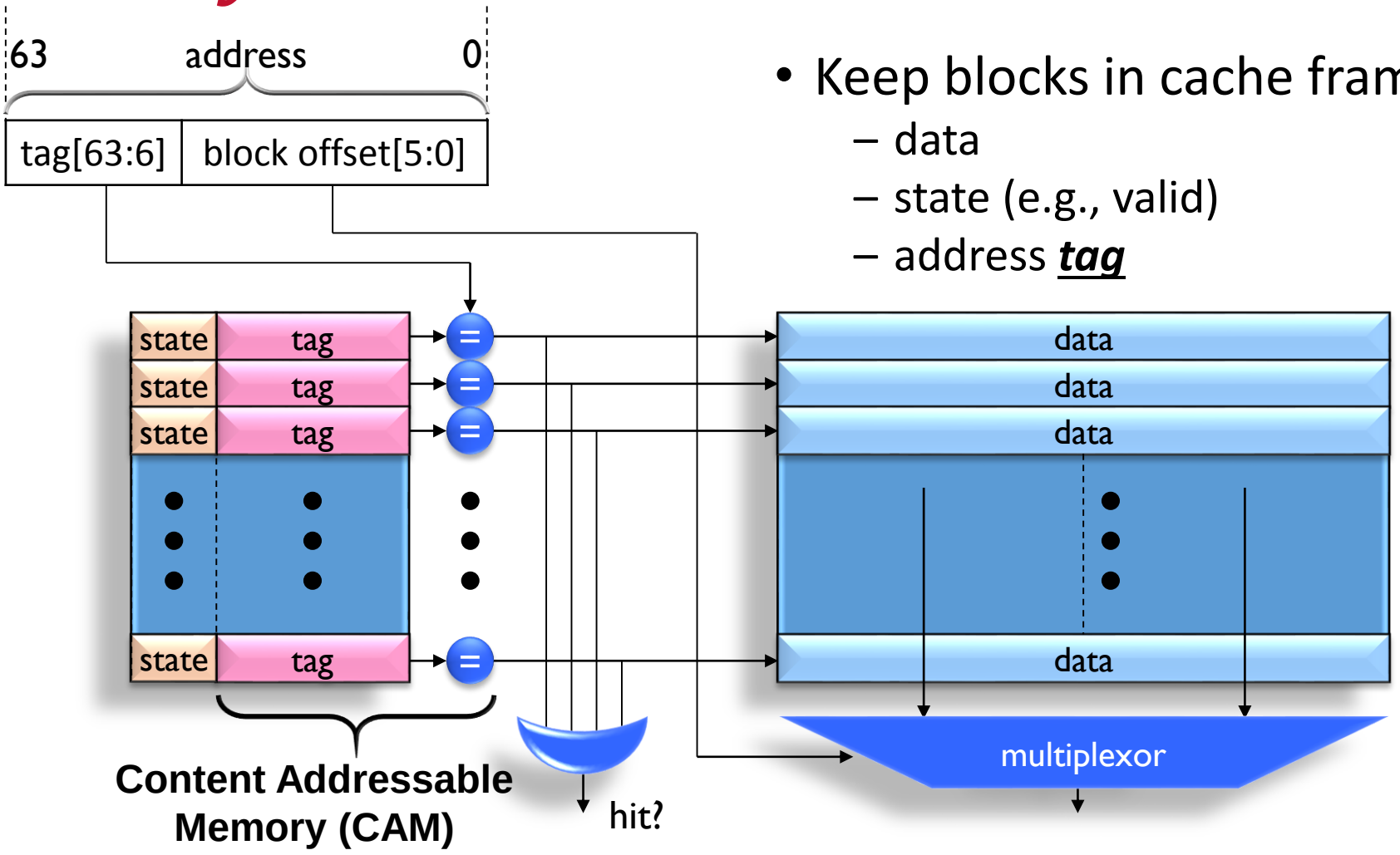
8-bit SRAM Array



8×8-bit SRAM Array



Fully-Associative Cache



- Keep blocks in cache frames
  - data
  - state (e.g., valid)
  - address ***tag***

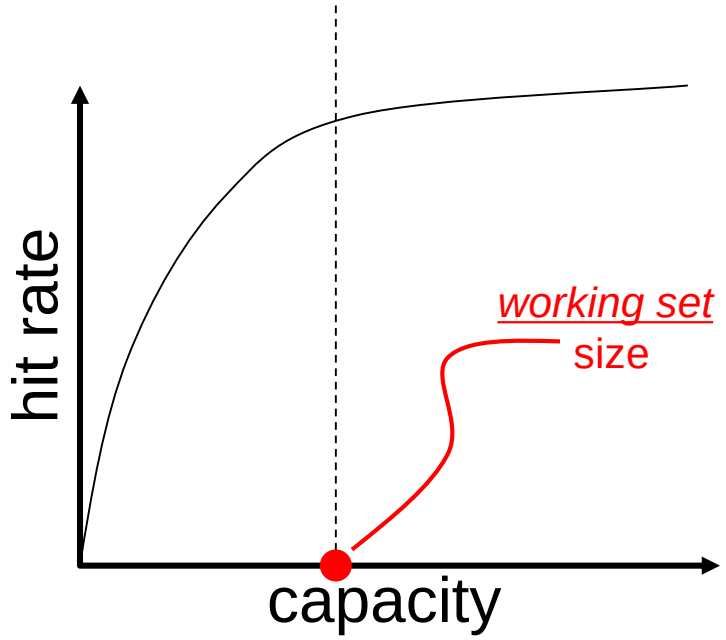
What happens when the cache runs out of space?

The 3 C's of Cache Misses

- *Compulsory*: Never accessed before
- *Capacity*: Accessed long ago and already replaced
- *Conflict*: Neither compulsory nor capacity (later today)
- *Coherence*: (Will discuss in multi-core lecture)

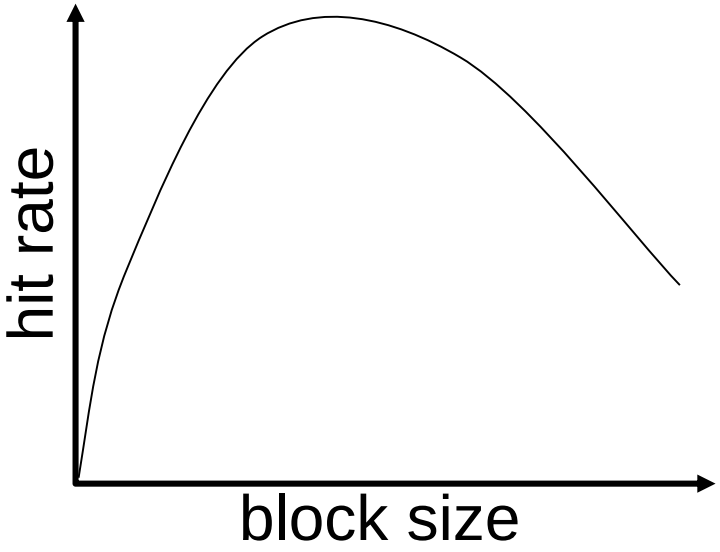
Cache Size

- Cache size is data capacity (don't count tag and state)
  - Bigger can exploit temporal locality better
  - Not always better
- Too large a cache
  - Smaller is faster → bigger is slower
  - Access time may hurt critical path
- Too small a cache
  - Limited temporal locality
  - Useful data constantly replaced



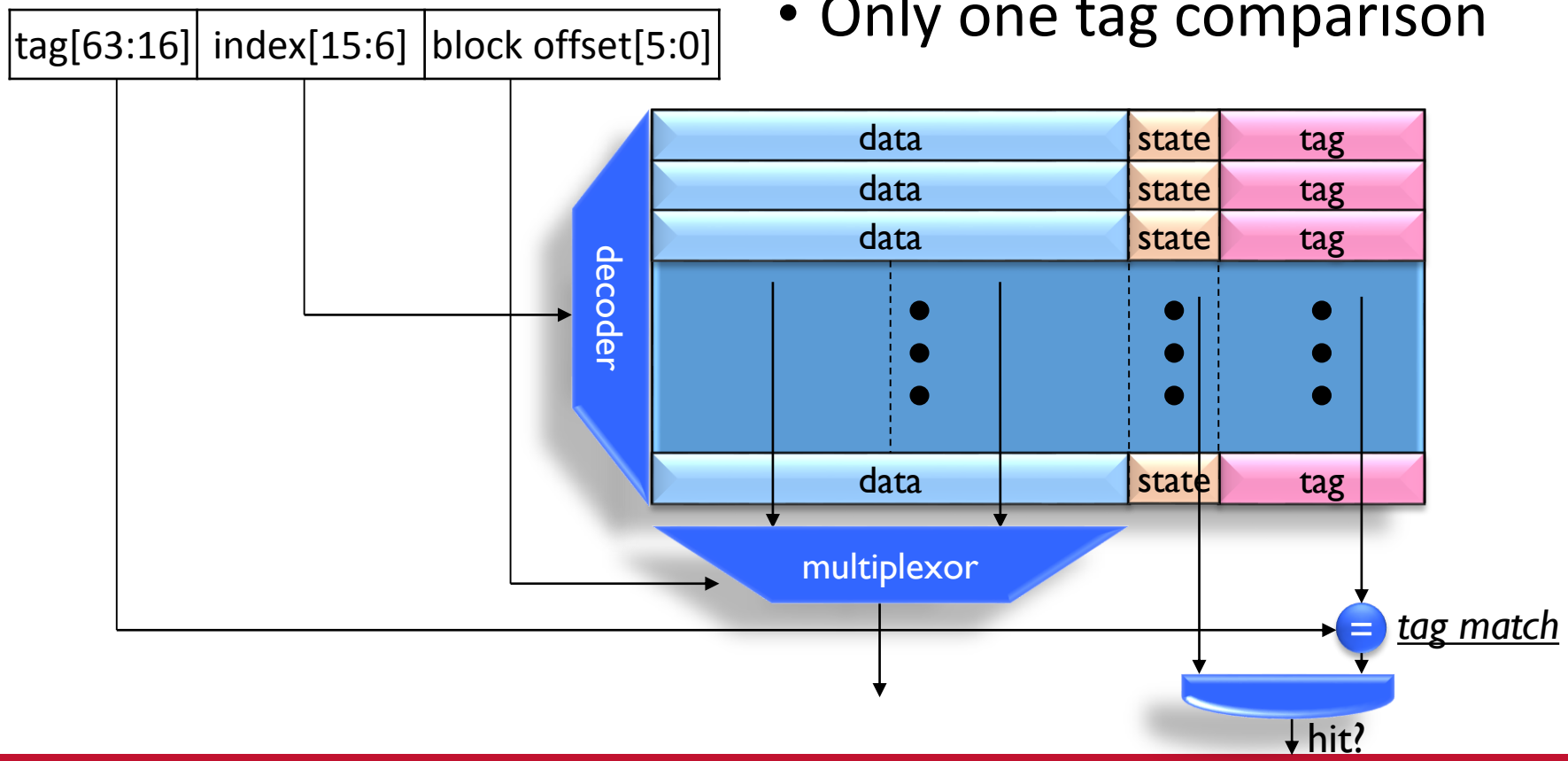
Block Size

- Block size is the data that is
  - Associated with an address tag
  - Not necessarily the unit of transfer between hierarchies
- Too small a block
  - Don't exploit spatial locality well
  - Excessive tag overhead
- Too large a block
  - Useless data transferred
  - Too few total blocks
    - Useful data frequently replaced



Direct-Mapped Cache

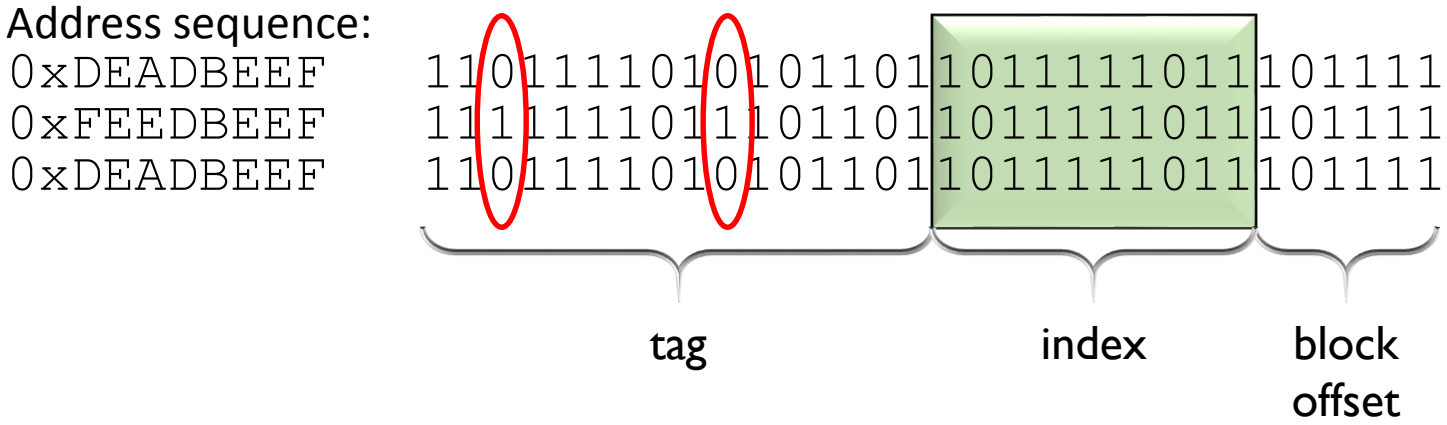
- Use middle bits as index
- Only one tag comparison



Why take index bits out of the middle?

Cache Conflicts

- What if two blocks alias on a frame?
  - Same index, but different tags

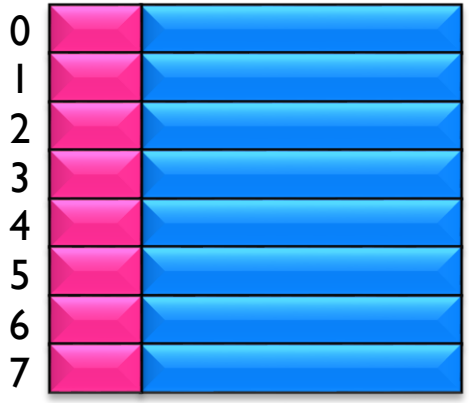


- 0xDEADBEEF experiences a *Conflict* miss
  - Not Compulsory (seen it before)
  - Not Capacity (lots of other indexes available in cache)

Associativity (1/2)

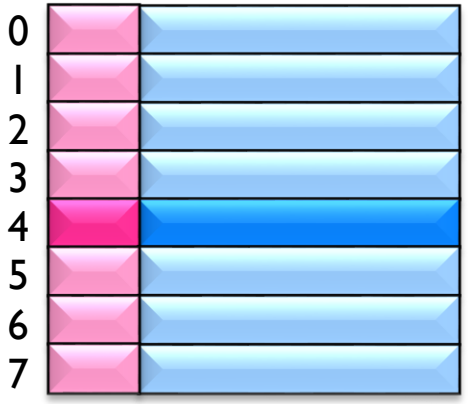
- Where does block index 12 (b'1100) go?

Frame



Set/Frame

Set



Fully-associative
block goes in any frame

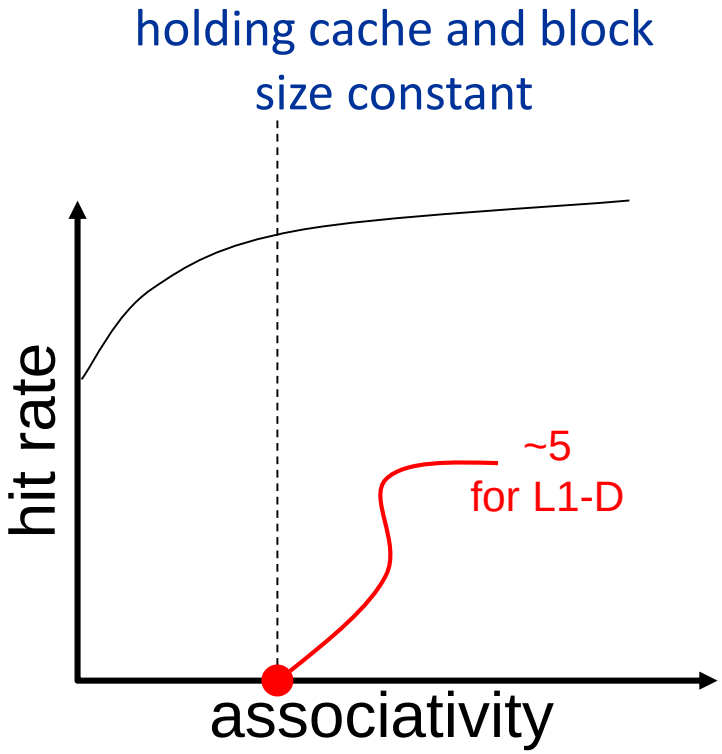
(all frames in 1 set)

Set-associative
block goes in any frame in one set
(frames grouped in sets)

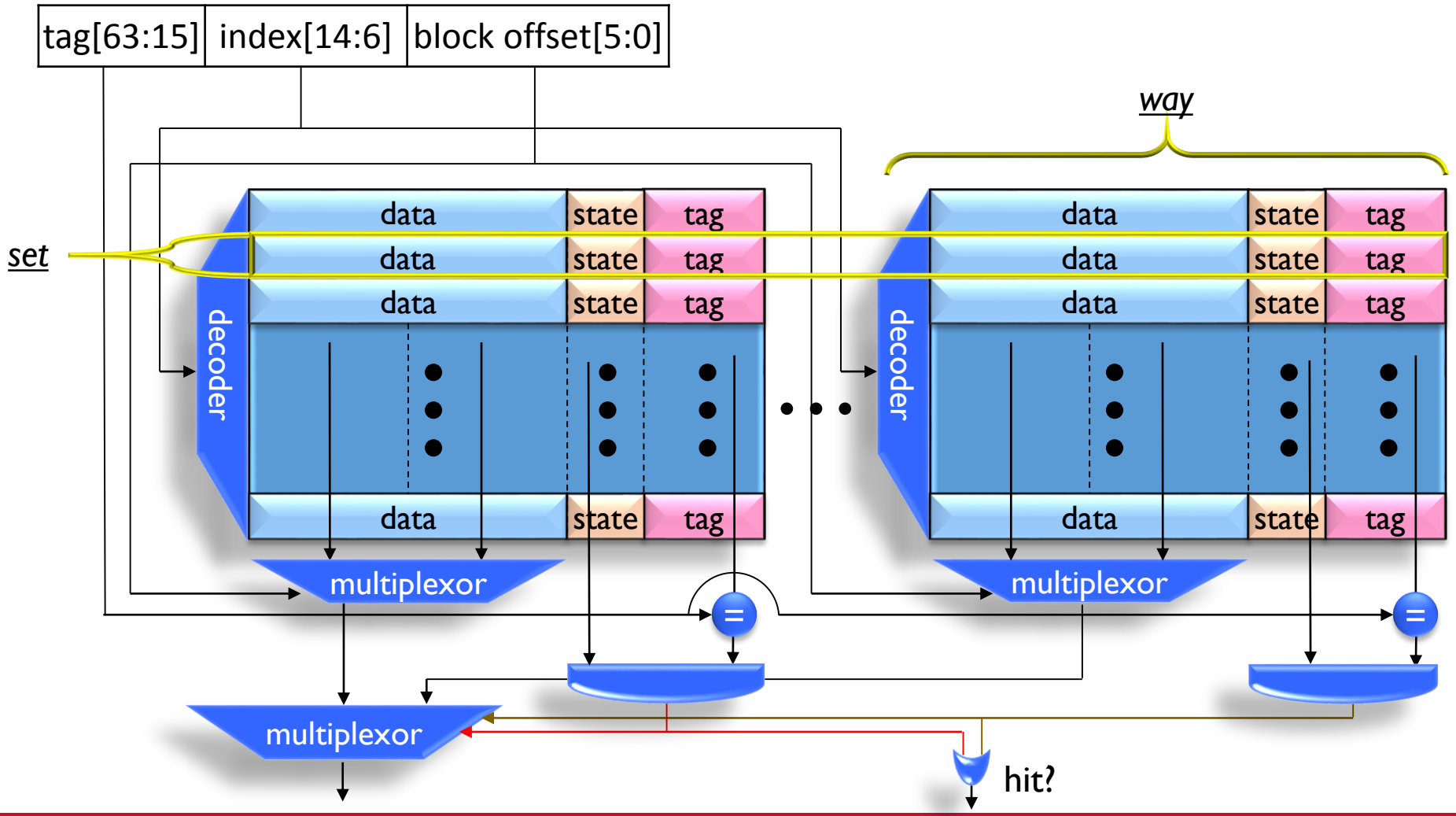
Direct-mapped
block goes in exactly one frame
(1 frame per set)

Associativity (2/2)

- Larger associativity
  - lower miss rate (fewer conflicts)
  - higher power consumption
- Smaller associativity
  - lower cost
  - faster hit time



N-Way Set-Associative Cache



Note the additional bit(s) moved from index to tag

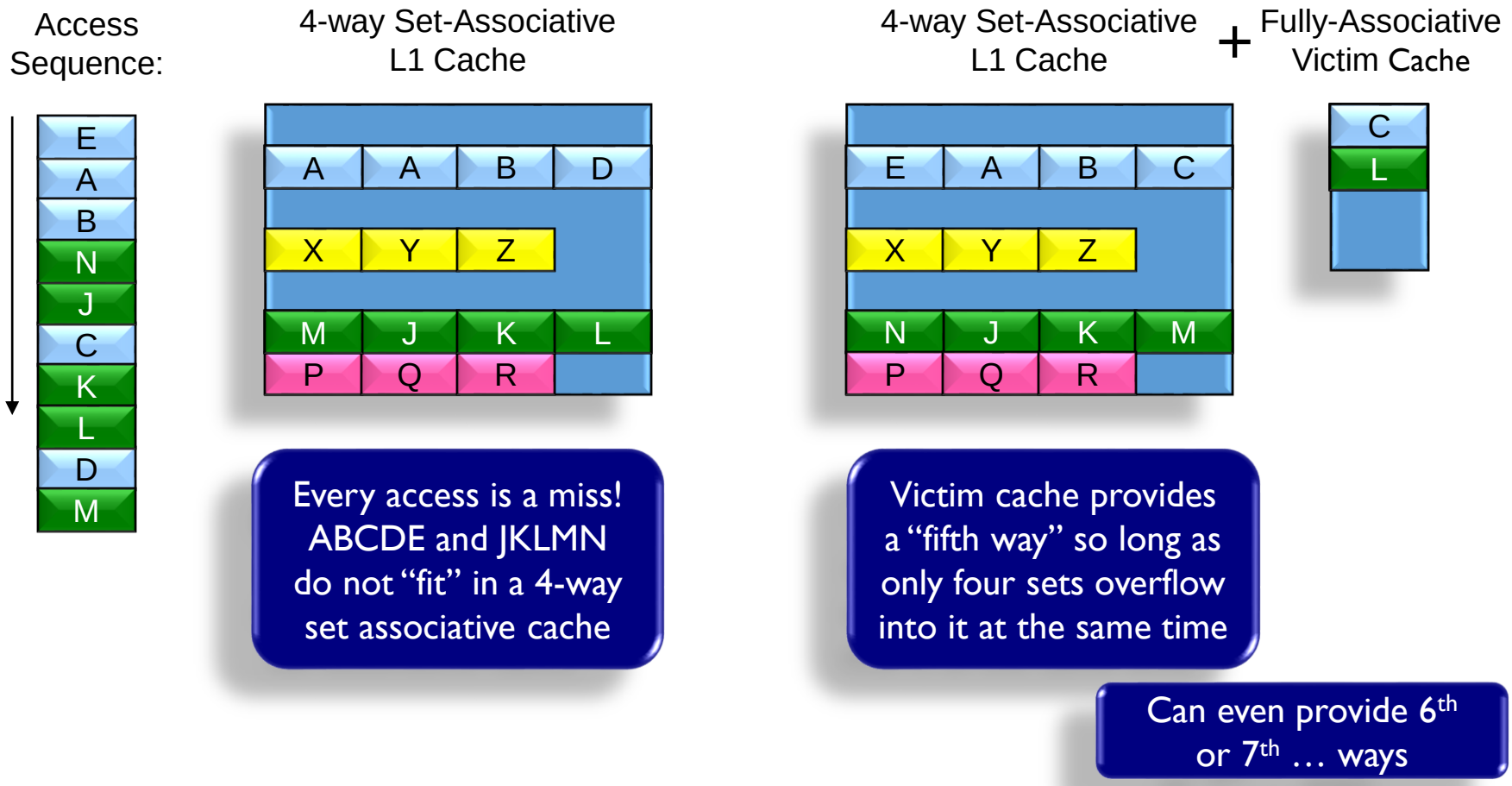
Associative Block Replacement

- Which block in a set to replace on a miss?
- Ideal replacement *(Belady's Algorithm)*
  - Replace block accessed farthest in the future
  - Trick question: How do you implement it?
- *Least Recently Used (LRU)*
  - Optimized for temporal locality (expensive for >2-way)
- *Not Most Recently Used (NMRU)*
  - Track MRU, random select among the rest
  - Same as LRU for 2-sets
- *Random*
  - Nearly as good as LRU, sometimes better (when?)
- *Pseudo-LRU*
  - Used in caches with high associativity
  - Examples: Tree-PLRU, Bit-PLRU

Victim Cache (1/2)

- Associativity is expensive
  - Performance overhead from extra muxes
  - Power overhead from reading and checking more tags and data
- Conflicts are expensive
  - Performance from extra mises
- Observation: Conflicts don't occur in all sets

Victim Cache (2/2)

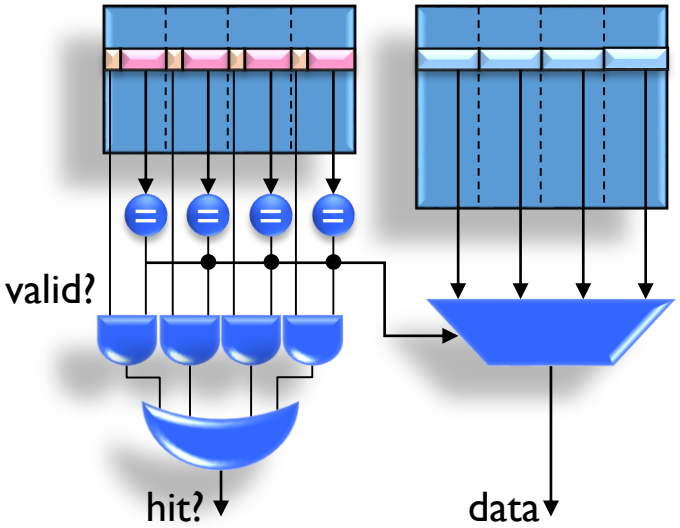


Provide "extra" associativity, but not for all sets

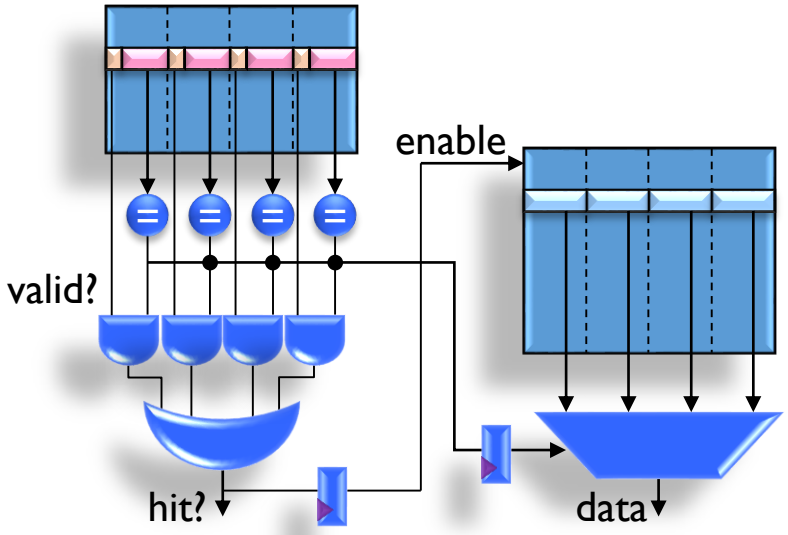
Parallel vs. Serial Caches

- Tag and Data usually separate (tag is smaller & faster)
  - State bits stored along with tags
    - *Valid* bit, "LRU" bit(s), …

Parallel access to Tag and Data reduces latency (good for L1)

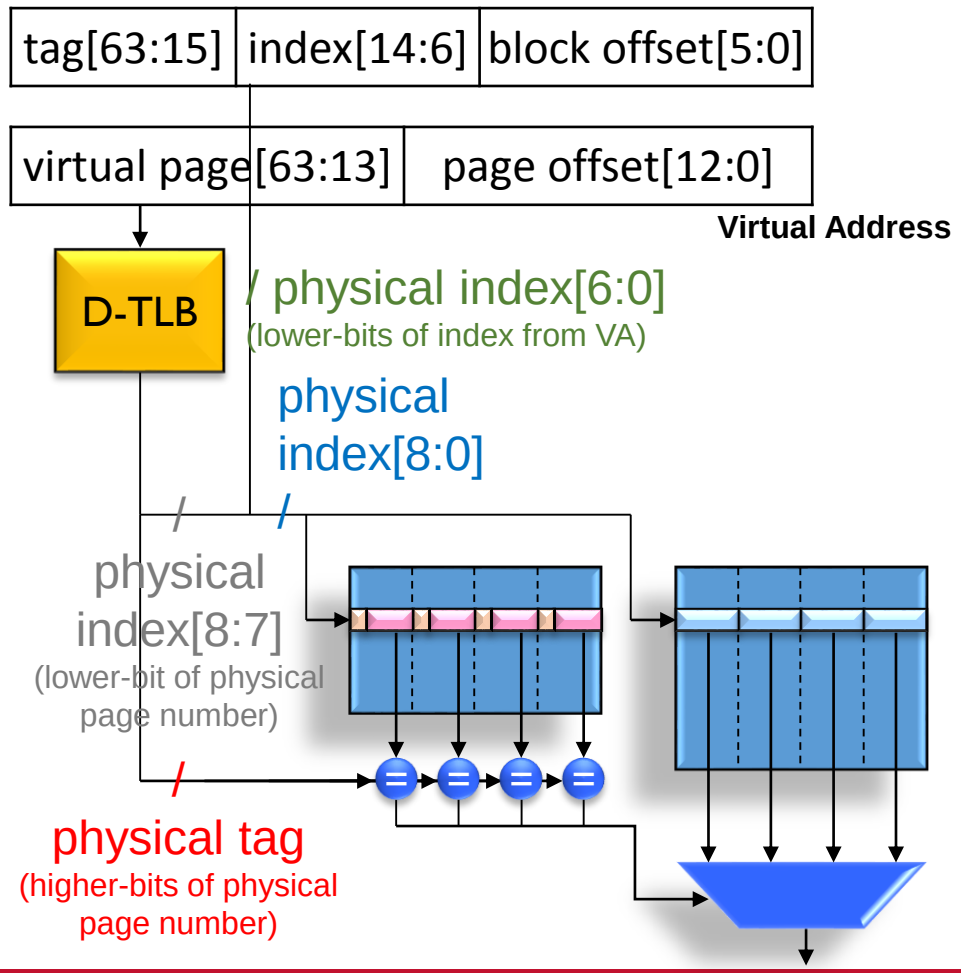


Serial access to Tag and Data reduces power (good for L2+)



Physically-Indexed Caches

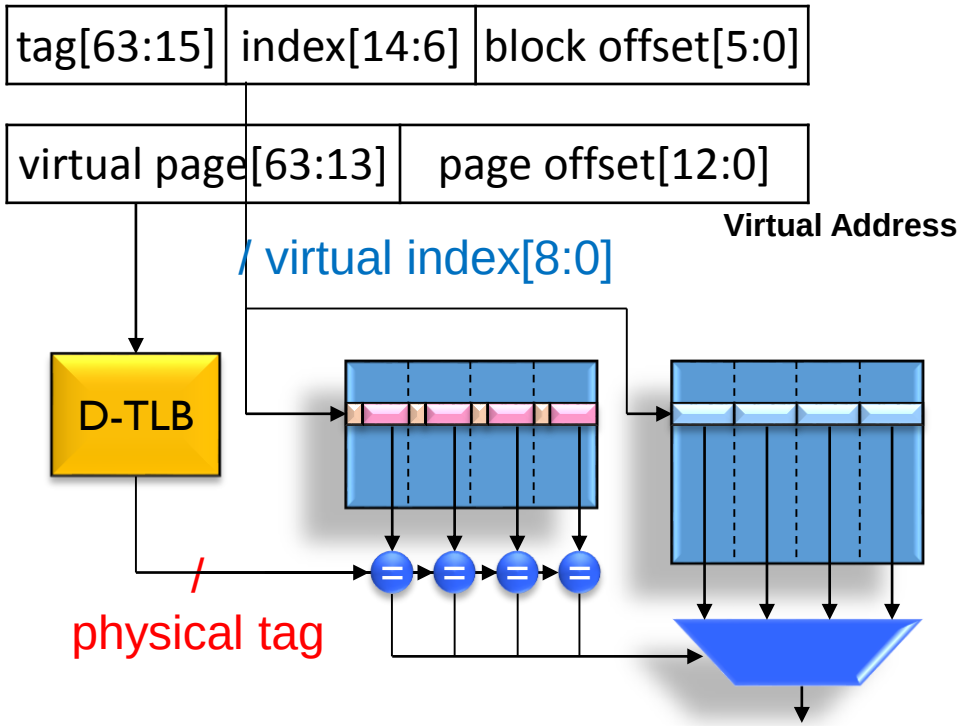
- Assume 8KB pages & 512 cache sets
  - 13-bit page offset
  - 9-bit cache index
- Core requests are VAs
- Cache index is PA[14:6]
  - PA[12:6] == VA[12:6]
  - VA passes through TLB
  - D-TLB on critical path
  - PA[14:13] from TLB
- Cache tag is PA[63:15]
- If index falls completely within page offset,
  - can use just VA for index



Simple, but slow. Can we do better?

Virtually-Indexed Caches

- Core requests are VAs
- Cache index is VA[14:6]
- Cache tag is PA[63:13]
  - Why not PA[63:15]?
- Why not tag with VA?
  - VA does not uniquely determine the memory location
  - Would need cache flush on ctxt switch

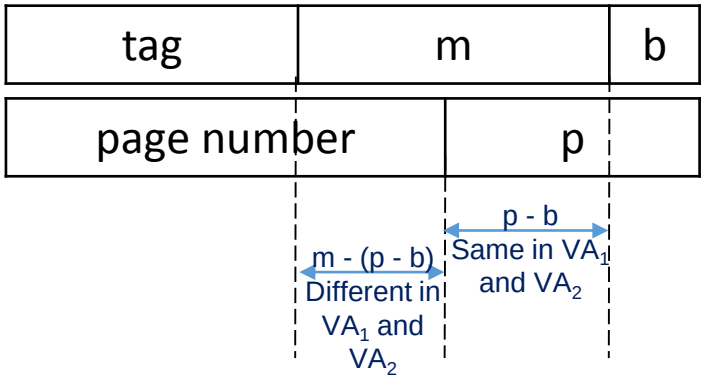


Virtually-Indexed Caches

- Main problem: *Virtual aliases*
  - Different virtual addresses for the same physical location
  - Different virtual addrs → map to different sets in the cache

Tag	Data
VA_1	1st Copy of Data at PA
VA_2	2nd Copy of Data at PA

- Solution: ensure they don't exist by invalidating all aliases when a miss happens
  - If page offset is *p* bits, block offset is *b* bits and index is *m* bits, an alias might exist in any of $2^{m-(p-b)}$ sets.
  - Search all those sets and remove aliases (alias = same physical tag)

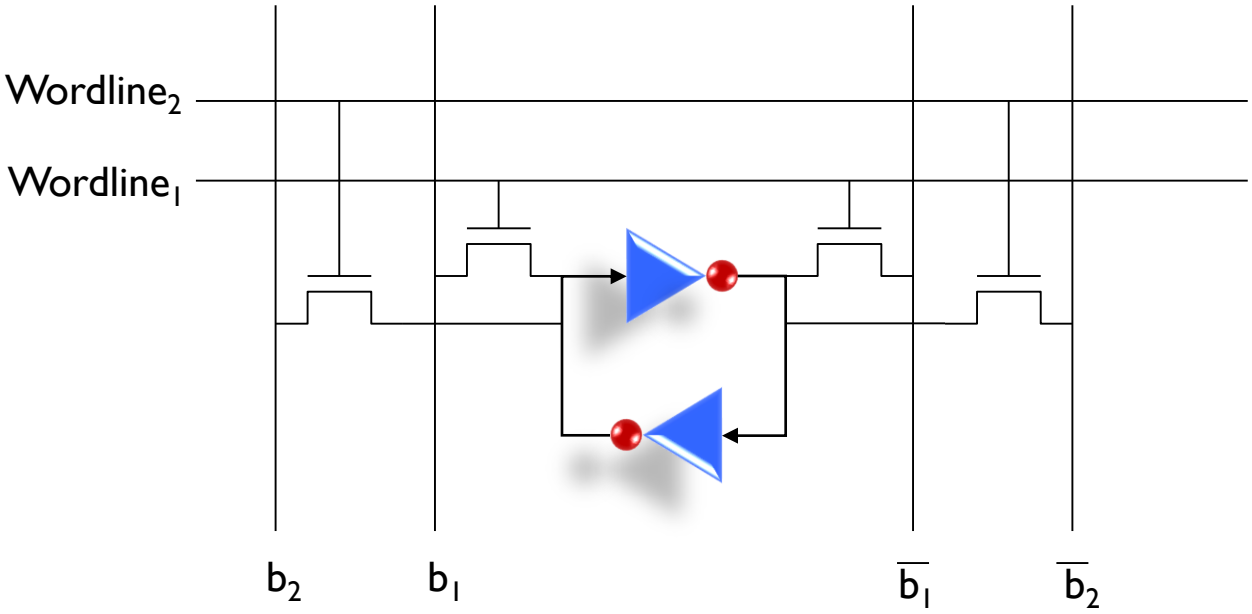


Fast, but complicated

Multiple Accesses per Cycle

- Need high-bandwidth access to caches
  - Core can make multiple access requests per cycle
  - Multiple cores can access LLC at the same time
- Must either delay some requests, or…
  - Design SRAM with multiple ports
    - Big and power-hungry
  - Split SRAM into multiple banks
    - Can result in delays, but usually not

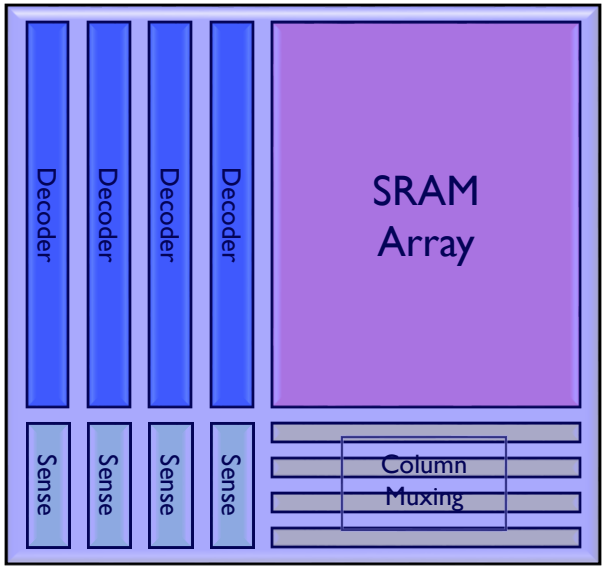
Multi-Ported SRAMs



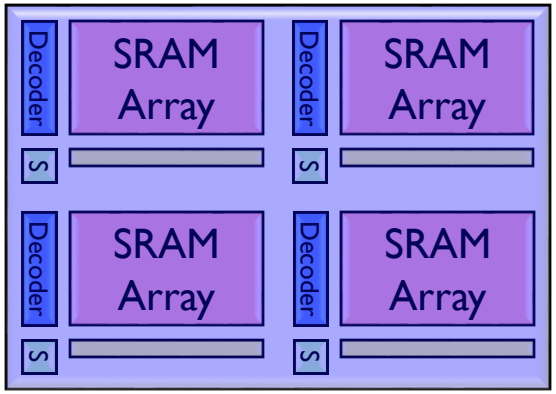
Wordlines = 1 per port
Bitlines = 2 per port

Area = O(ports^2)

Multi-Porting vs. Banking



4 ports
Big (and slow)
Guarantees concurrent access

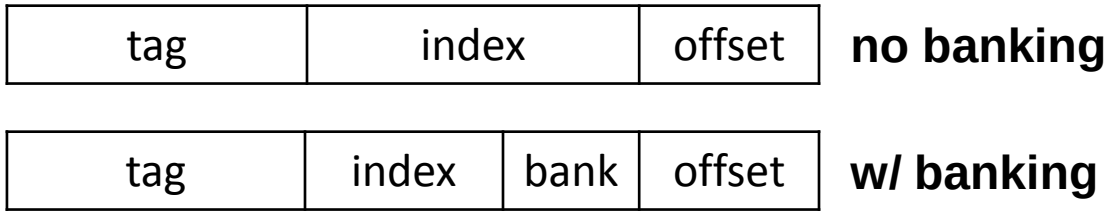


4 banks, 1 port each
Each bank small (and fast)
Conflicts (delays) possible

How to decide which bank to go to?

Bank Conflicts

- Banks are *address interleaved*
  - For block size *b cache with N* banks…
  - Bank = (Address / b) % N
    - Looks more complicated than is: just low-order bits of index



- Banking can provide high bandwidth
- But only if all accesses are to different banks
  - For 4 banks, 2 accesses, chance of conflict is 25%

Write Policies

- Writes are more interesting
  - On reads, tag and data can be accessed in parallel
  - On writes, needs two steps
  - Is access time important for writes?
- Choices of Write Policies
  - On write hits, update memory?
    - Yes: *write-through* (higher bandwidth)
    - No: *write-back* (uses *Dirty* bits to identify blocks to write back)
  - On write misses, allocate a cache block frame?
    - Yes: *write-allocate*
    - No: *no-write-allocate*

Inclusion

- Core often accesses blocks not present in any $
  - Should block be allocated in L3, L2, and L1?
    - Called *Inclusive* caches
    - Waste of space
    - Requires forced evict (e.g., force evict from L1 on evict from L2+)
  - Only allocate blocks in L1
    - Called *Non-inclusive* caches (why not "exclusive"?)
- Some processors combine both
  - L3 is inclusive of L1 and L2
  - L2 is non-inclusive of L1 (like a large victim cache)