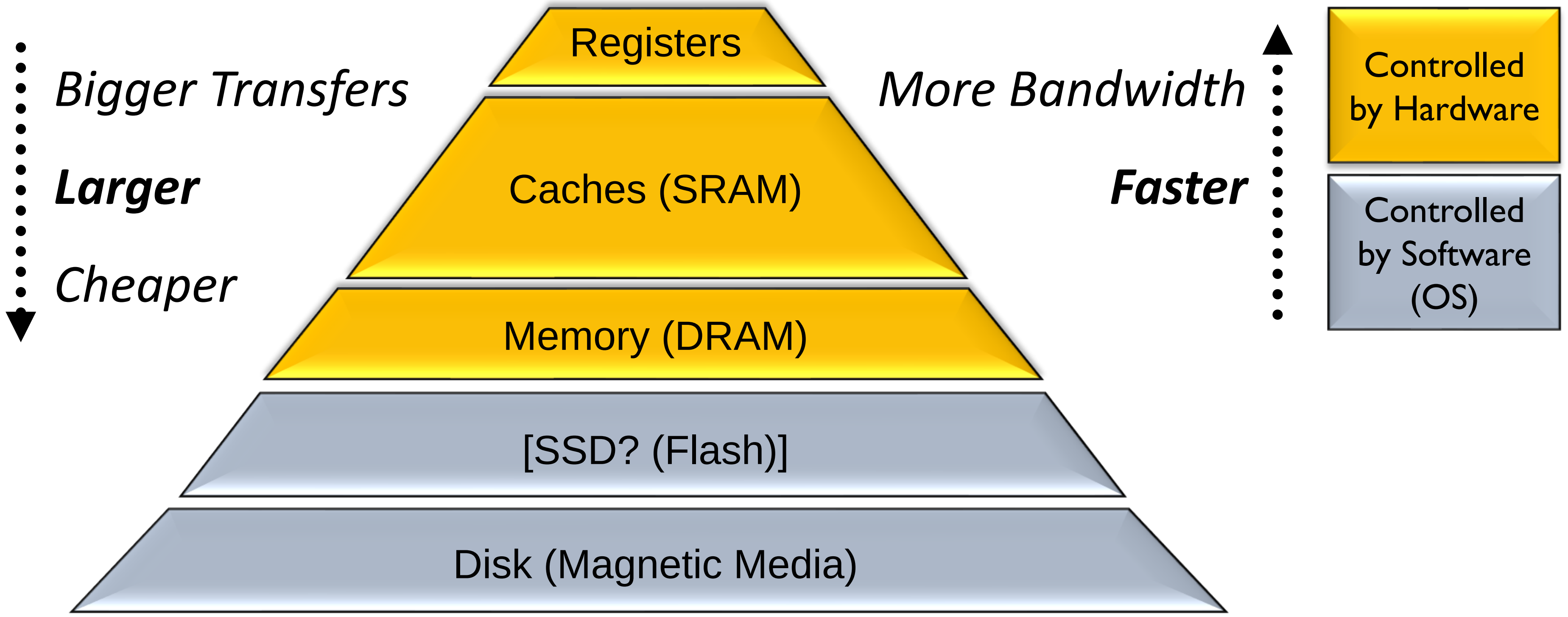
Stony Brook University

Cache Design Basics

Nima Honarmand

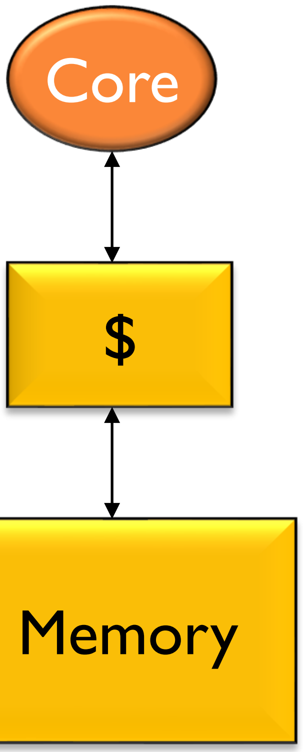
Storage Hierarchy

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



Caches

- An <u>automatically managed</u> hierarchy
- Break memory into **blocks** (several bytes) and transfer data to/from cache in blocks
  - To exploit <u>*spatial locality*</u>
- Keep recently accessed blocks
  - To exploit <u>*temporal locality*</u>



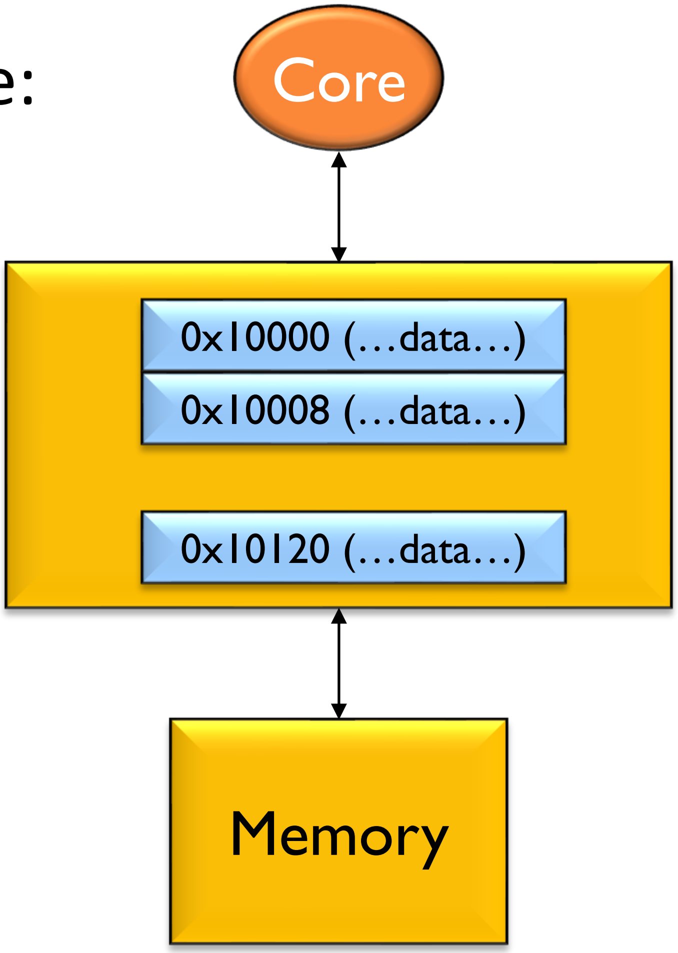
Cache Terminology

- **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 retrieve 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)

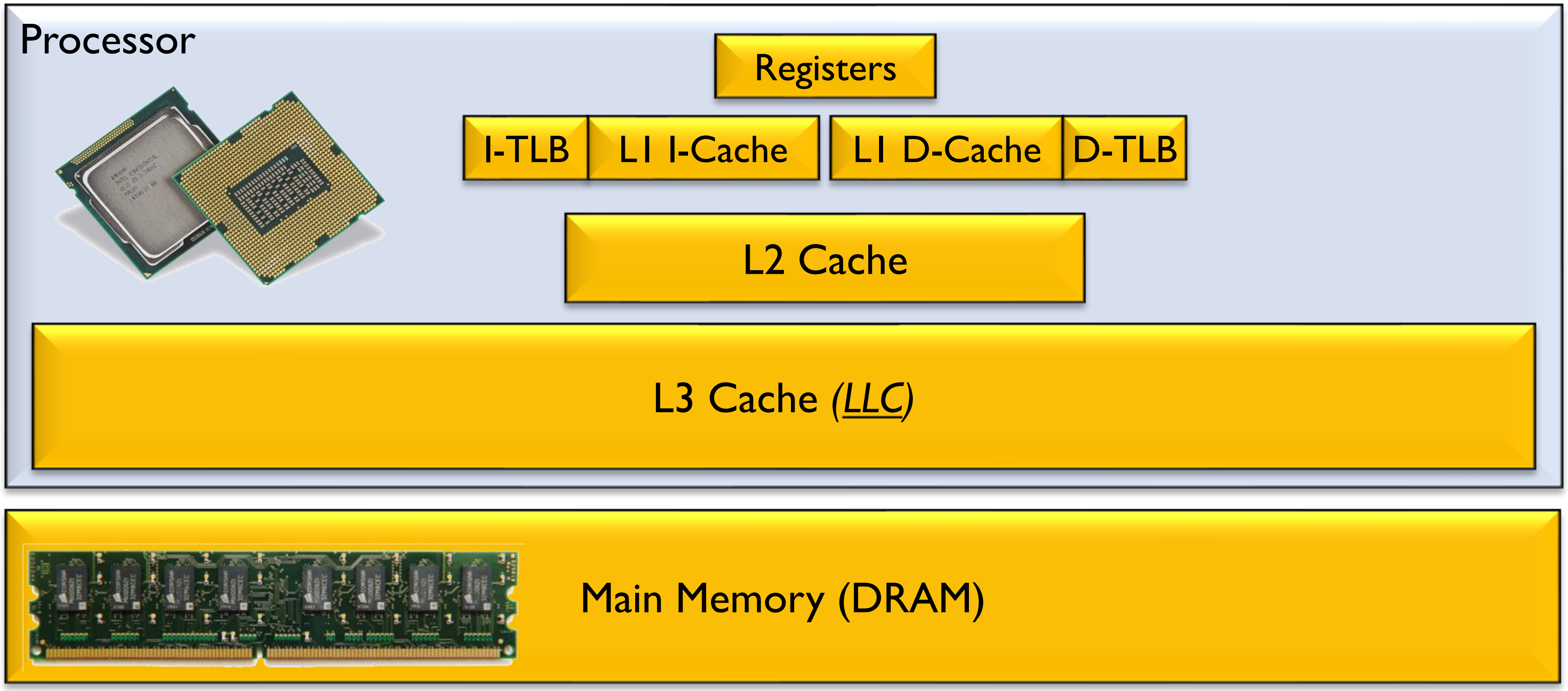
- Or AMAT = *Hit-time* + *Miss-rate* × *Miss-penalty*
- Very powerful tool to estimate performance
- If …
  cache hit is 10 cycles (core to L1 and back)
  miss penalty is 100 cycles (miss penalty)
- Then …
  at 50% miss ratio, avg. access: 10+0.5×100 = 60
  at 10% miss ratio, avg. access: 10+0.1×100 = 20
  at 1% miss ratio, avg. access: 10+0.01×100 = 11

Average Memory Access Time (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 (L2 miss penalty)
- Then …
  at 20% miss ratio in L1 and 40% miss ratio in L2 …
  avg. access: $5+0.2\times(0.6\times20+0.4\times100) = 15.4$

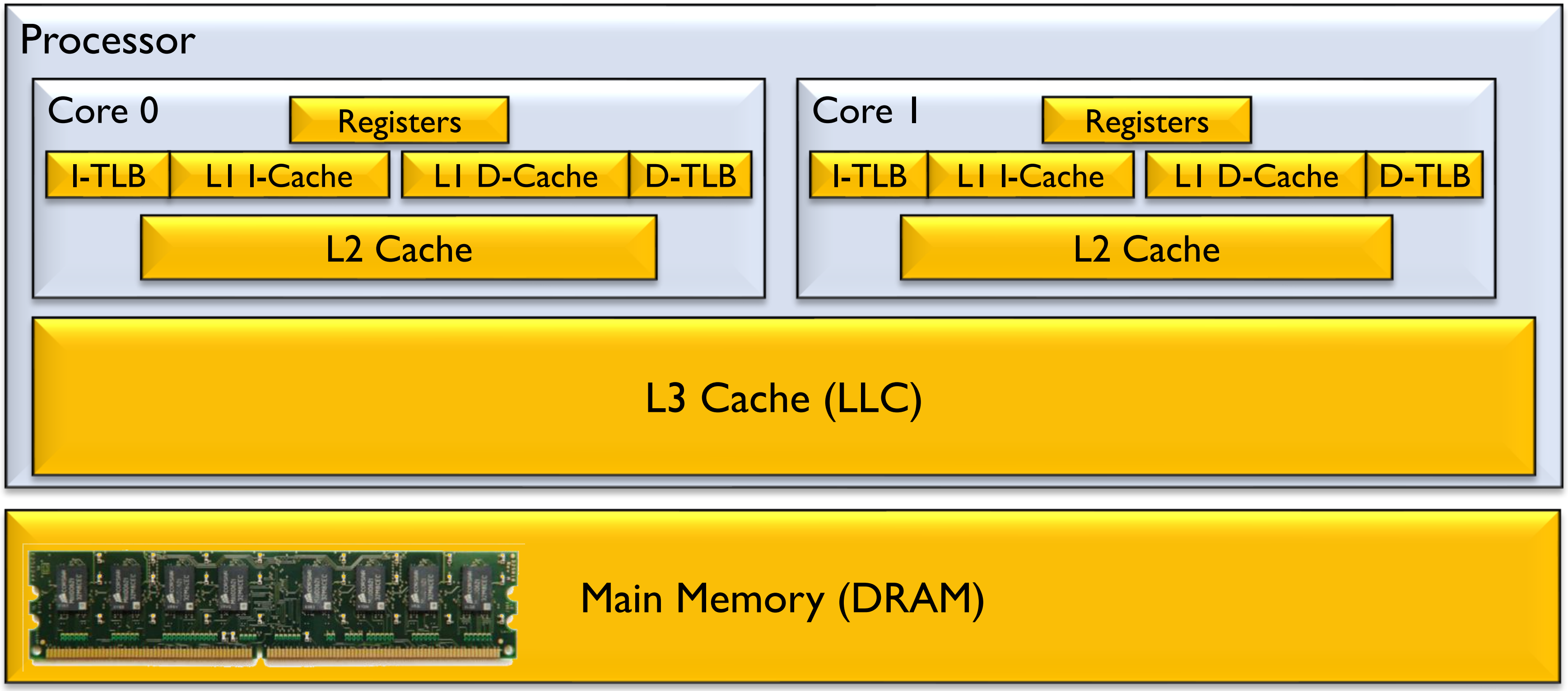
Memory Hierarchy (1)

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



Memory Hierarchy (2)

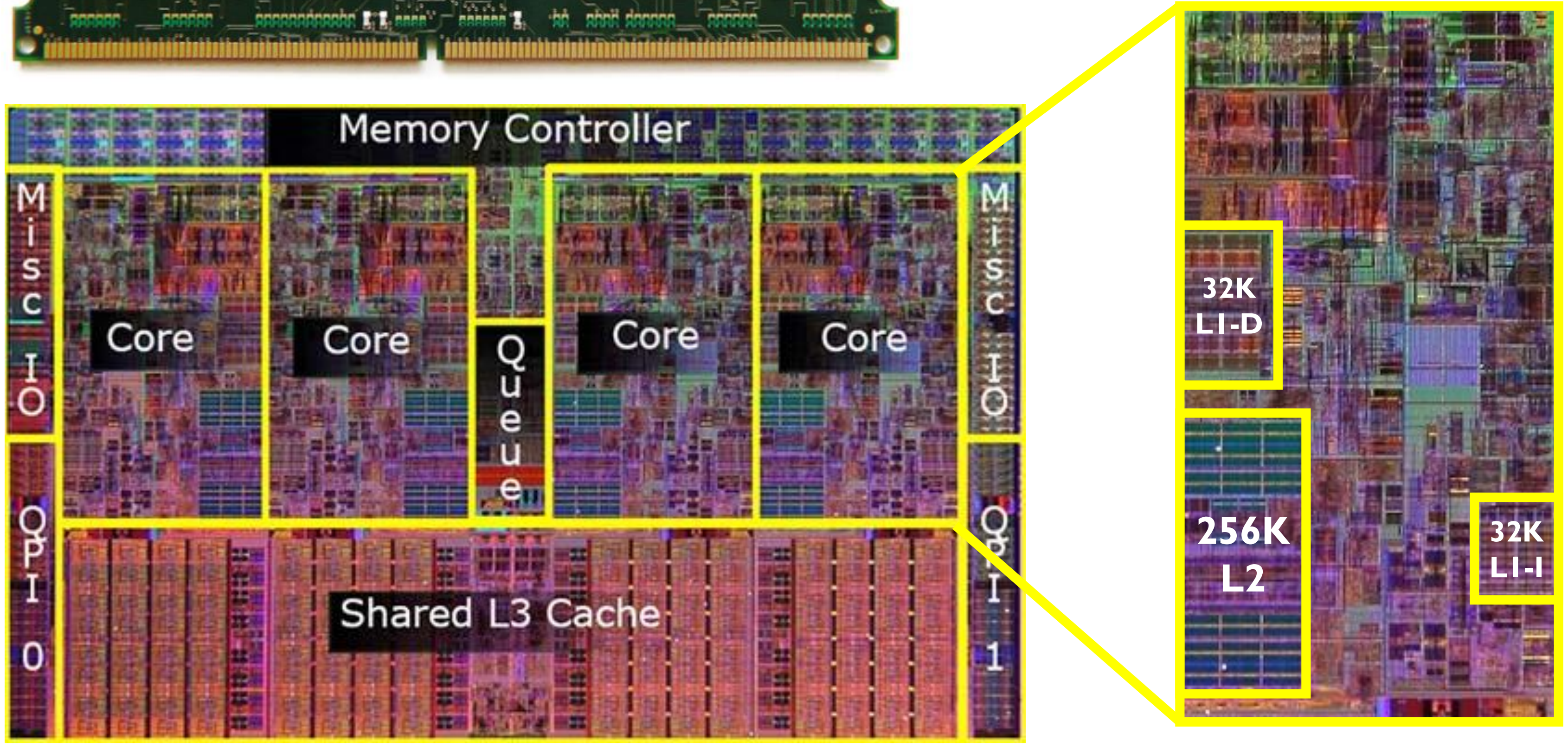
- L1 and L2 are *private*
- L3 is *shared*



Multi-core replicates the top of the hierarchy

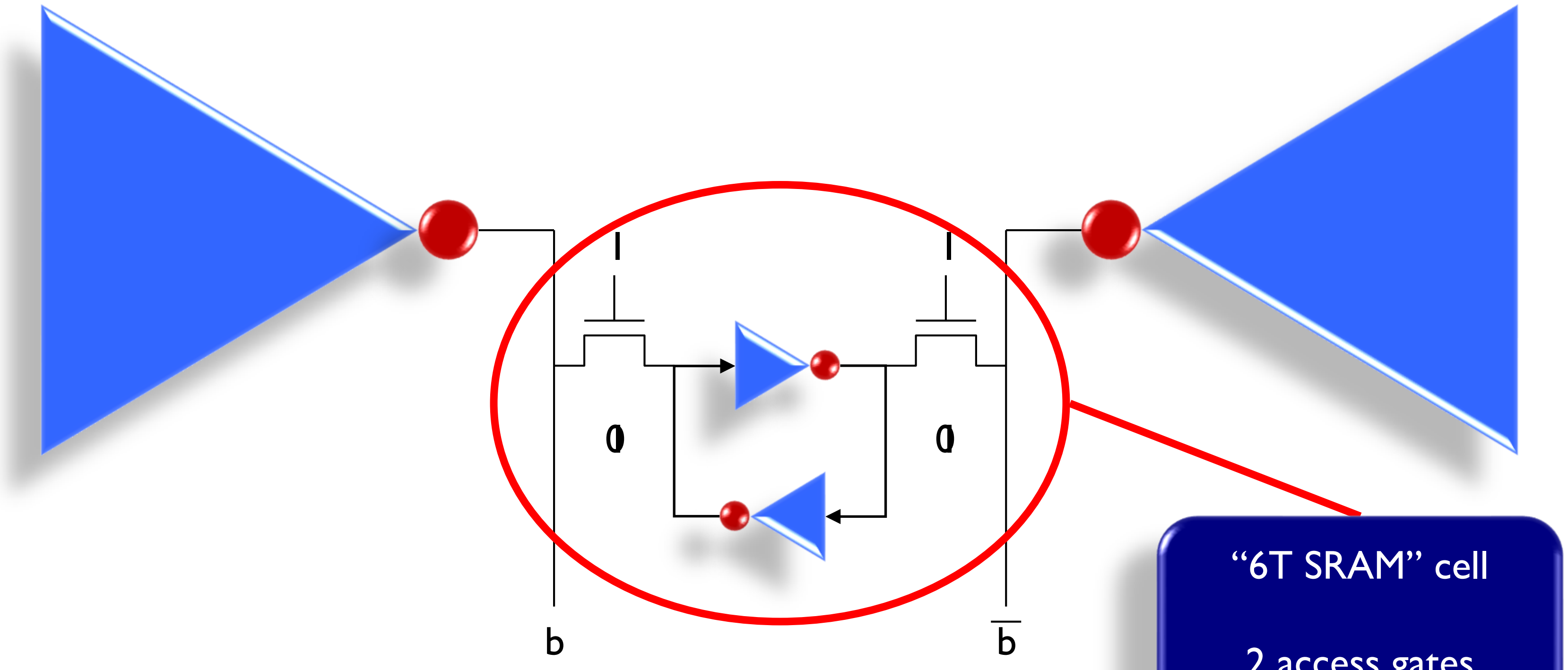
Memory Hierarchy (3)

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



How to Build a Cache

SRAM Overview

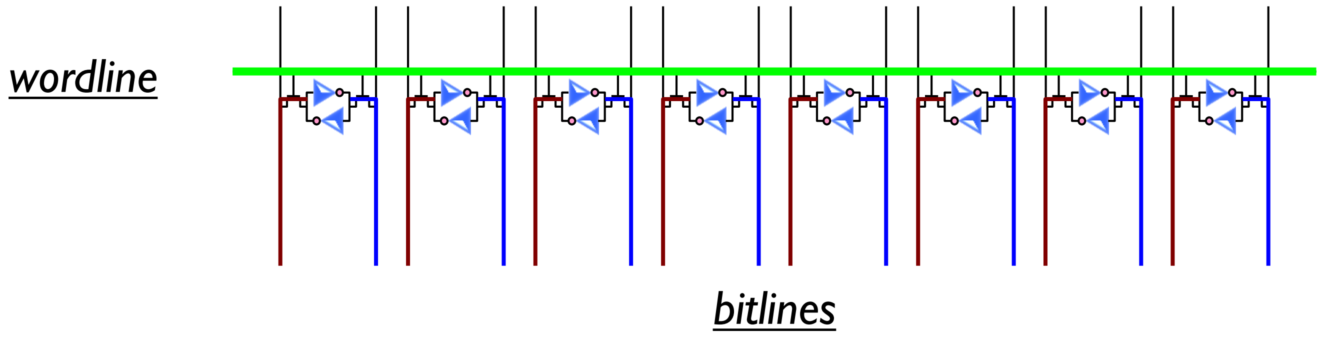


"6T SRAM" cell

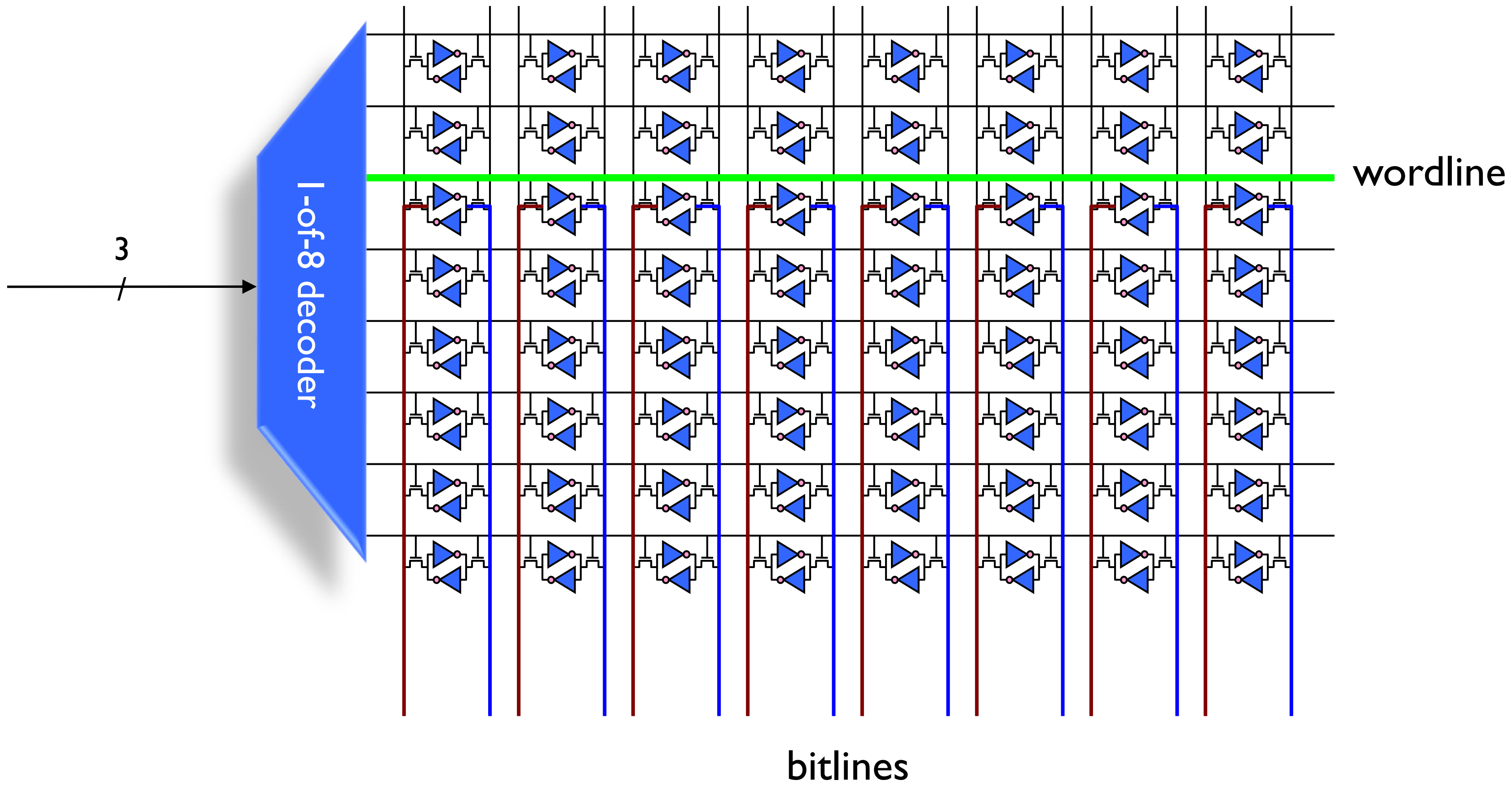
2 access gates
2T per inverter

- 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

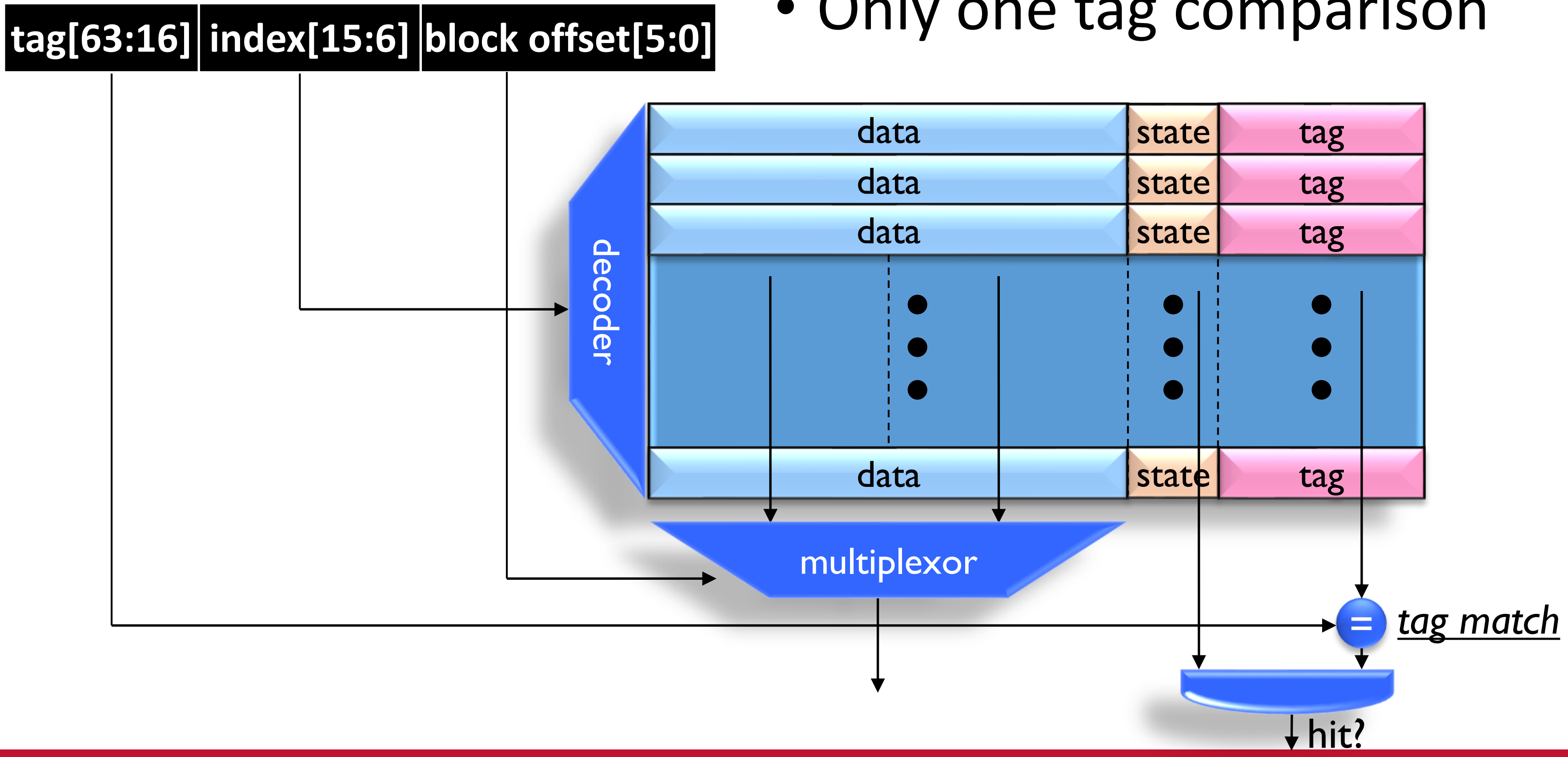


8×8-bit SRAM Array



Direct-Mapped Cache using SRAM

- Use middle bits as index
- Only one tag comparison



Why take index bits out of the middle?

Improving Cache Performance

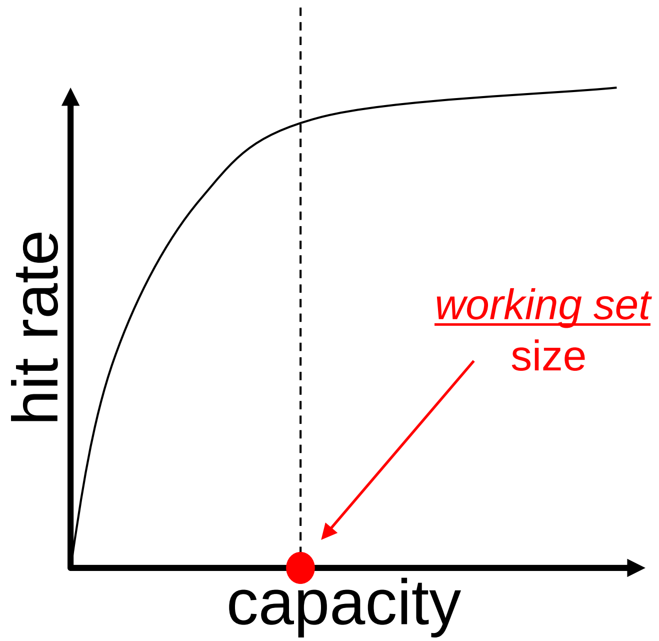
- Recall AMAT formula:
  - *AMAT* = *Hit-time* + *Miss-rate* × *Miss-penalty*
- To improve cache performance, we can improve any of the three components
- Let’s start by reducing **miss rate**

The 4 C's of Cache Misses

- **Compulsory**: Never accessed before
- **Capacity**: Accessed long ago and already replaced because cache too small
- **Conflict**: Neither compulsory nor capacity, because of limited *associativity*
- **Coherence**: (Will discuss in multi-processor lectures)

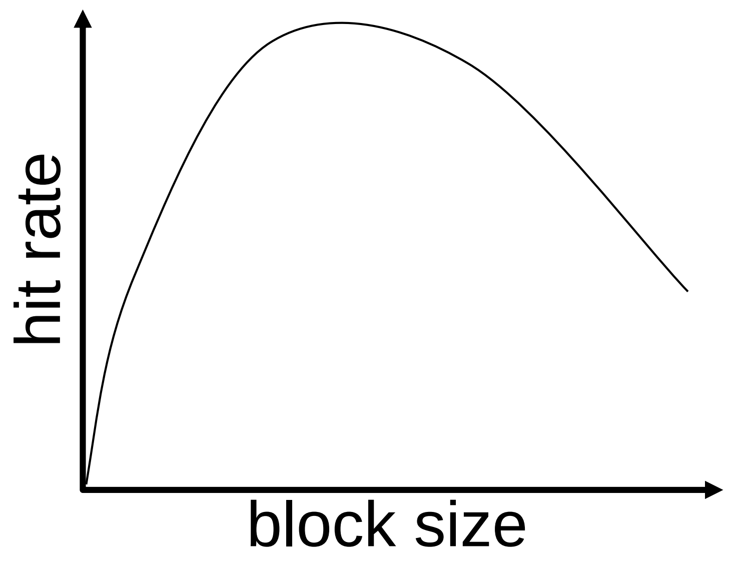
Cache Size

- 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



Common Block Sizes are 32-128 bytes

Cache Conflicts

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

Address sequence:

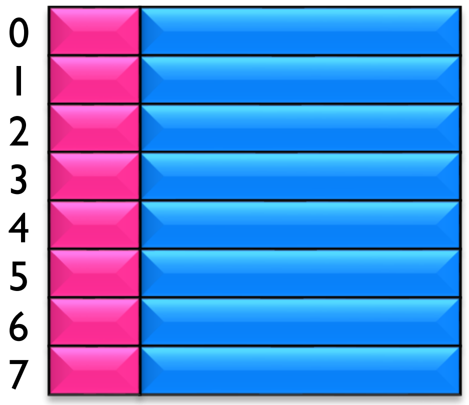
```
0xDEADBEEF    1101111010101101 1011111011 101111
0xFEEDBEEF    1111111011101101 1011111011 101111
0xDEADBEEF    1101111010101101 1011111011 101111
```

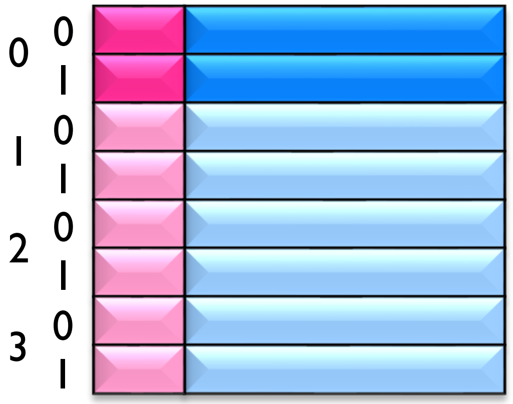
$$\underbrace{1101111010101101}_{\text{tag}}\ \underbrace{\boxed{1011111011}}_{\text{index}}\ \underbrace{101111}_{\text{block offset}}$$

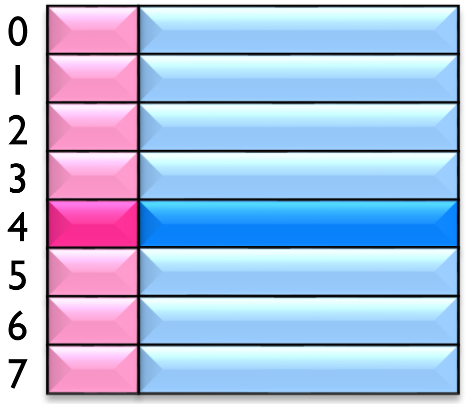
- 0xDEADBEEF experiences a ***Conflict*** miss
  - Not Compulsory (seen it before)
  - Not Capacity (lots of other frames available in cache)

Associativity (1)

- In cache w/ 8 frames, where does block 12 (b'1100) go?







Fully-associative
block goes in any frame

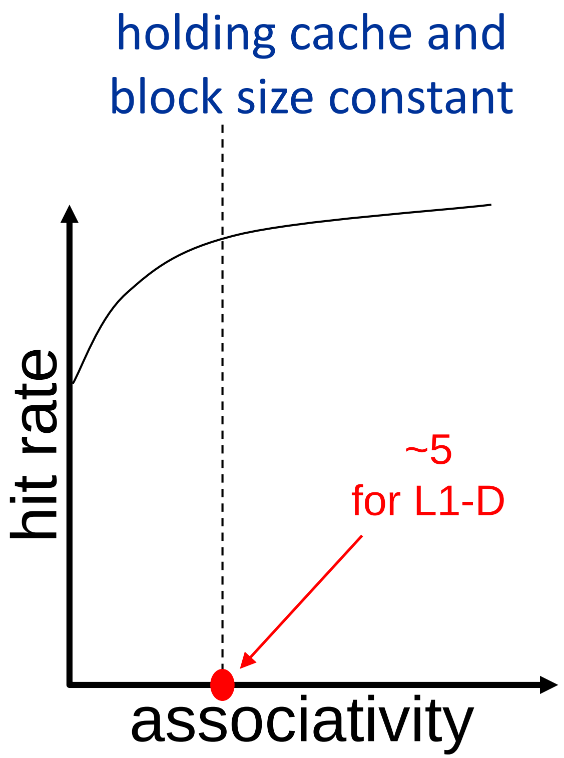
(all frames in 1 set)

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

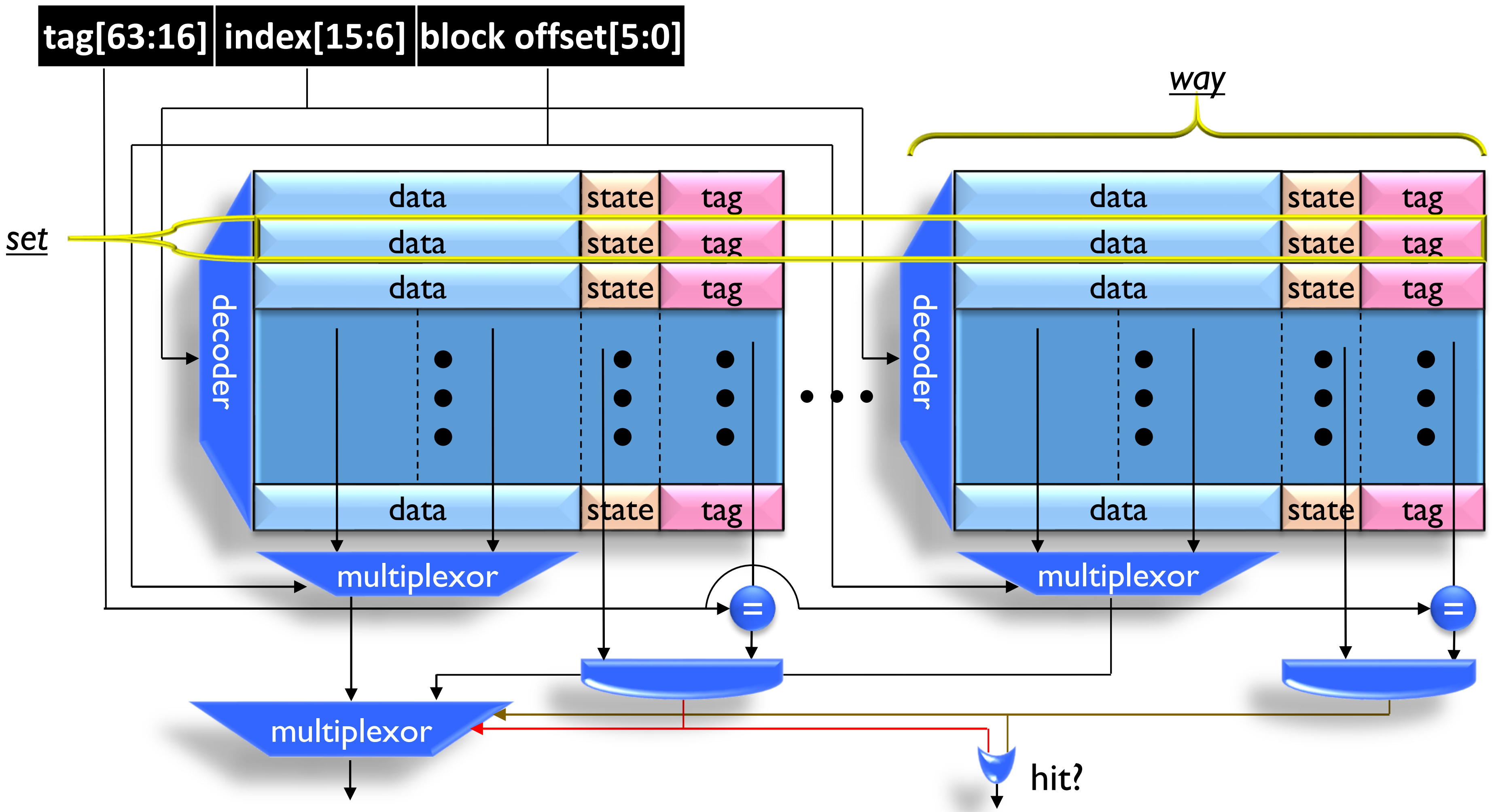
Direct-mapped
block goes in exactly one frame
(1 frame per set)

Associativity (2)

- Larger associativity (for the same size)
  - lower miss rate (fewer conflicts)
  - higher power consumption
- Smaller associativity
  - lower cost
  - faster hit time
- *2:1 rule of thumb*: for small caches (up to 128KB), 2-way assoc. gives same miss rate as direct-mapped twice the size



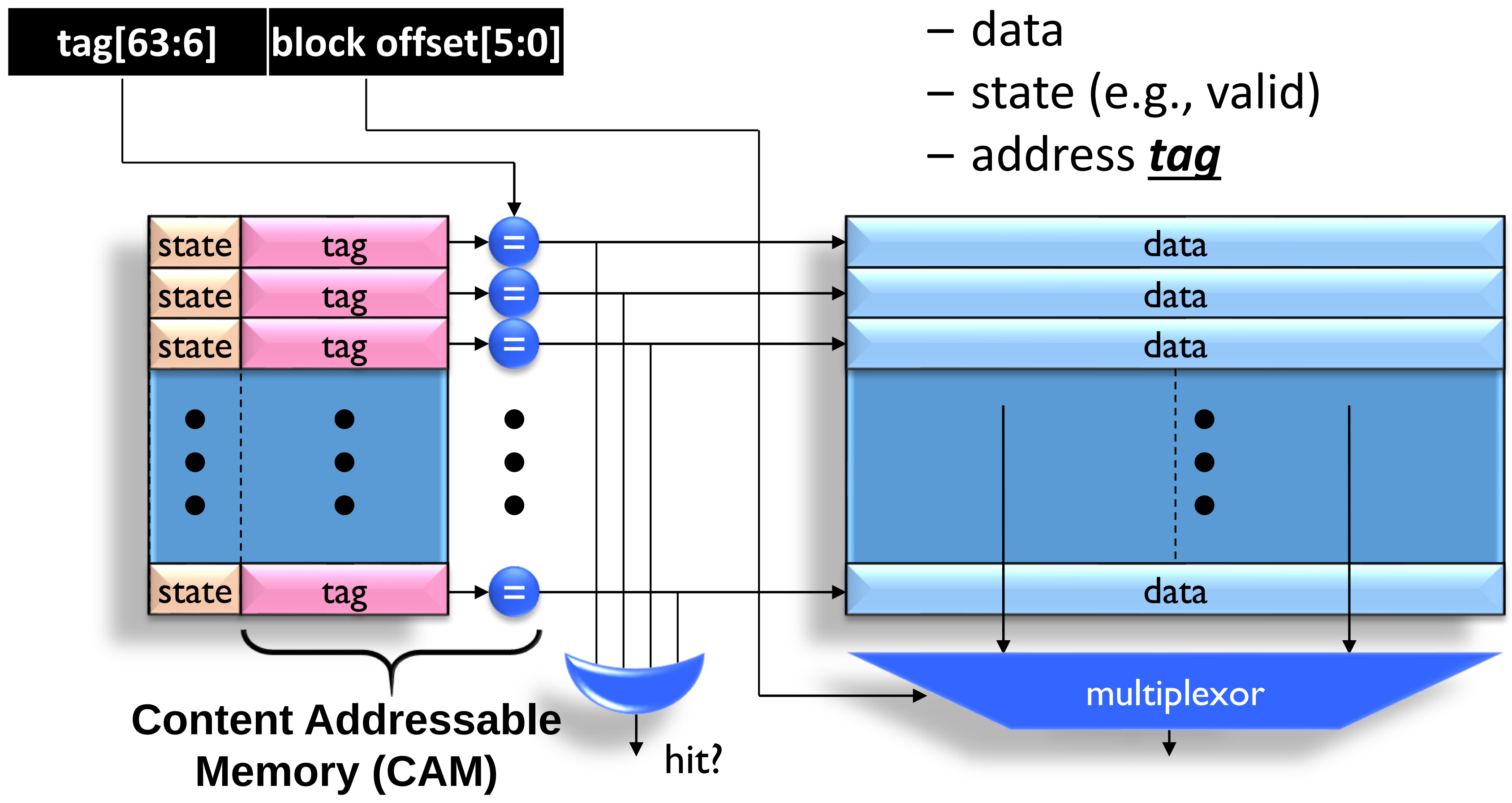
N-Way Set-Associative Cache



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

Fully-Associative Cache

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



Block Replacement Algorithms

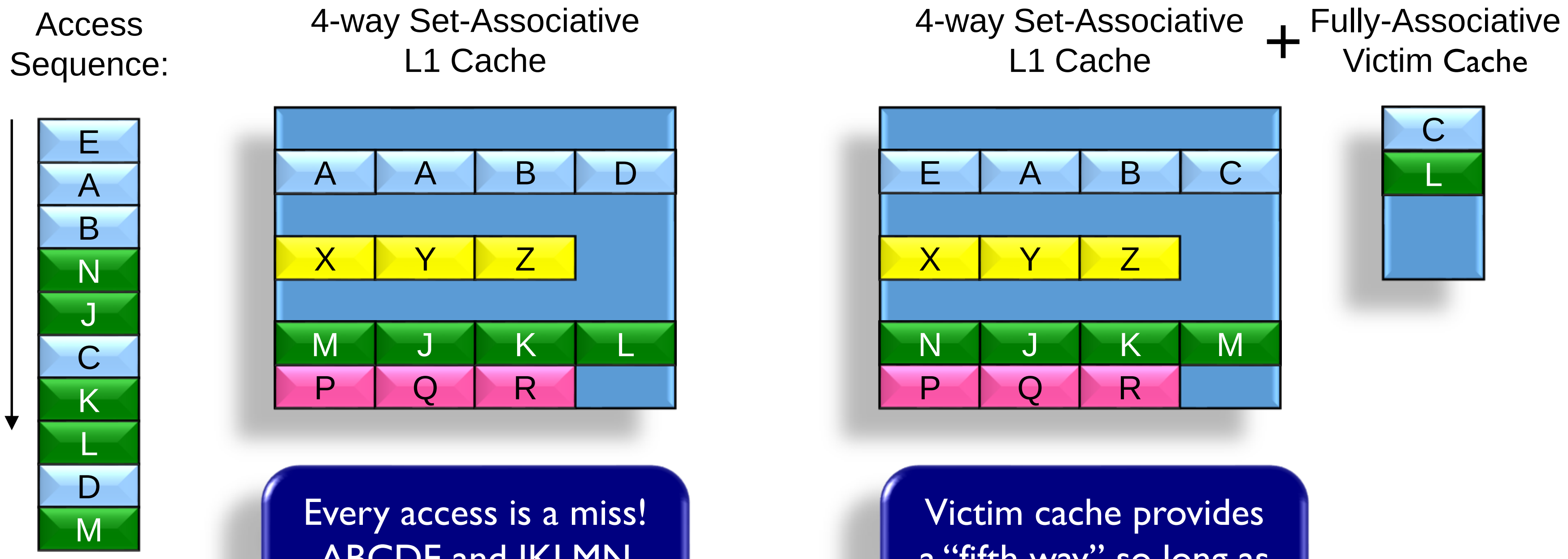
Which block in a set to replace on a miss?

- **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 associativity)
- **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)

- 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
- **Idea**: use a fully-associative "victim" cache to absorbs blocks displaced from the main cache

Victim Cache (2)


Access Sequence:
E
A
B
N
J
C
K
L
D
M
4-way Set-Associative L1 Cache
A A B D
X Y Z
M J K L
P Q R
4-way Set-Associative L1 Cache
+
Fully-Associative Victim Cache
E A B C
X Y Z
N J K M
P Q R
C
L

Every access is a miss! ABCDE and JKLMN do not "fit" in a 4-way set associative cache

Victim cache provides a "fifth way" so long as only four sets overflow into it at the same time

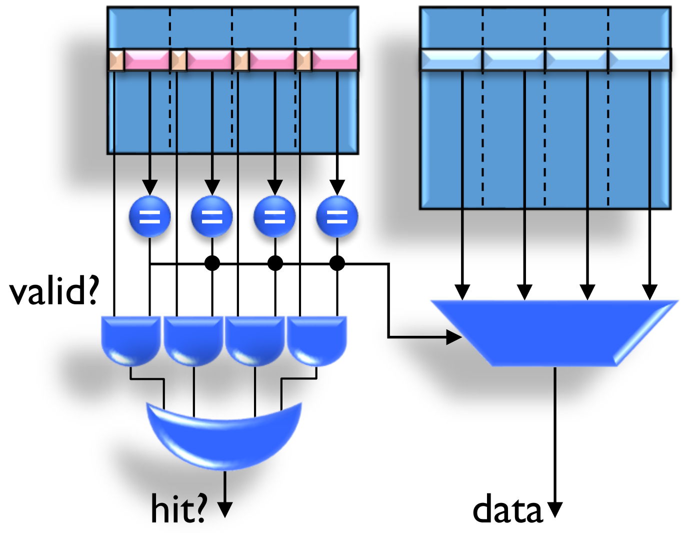
Can even provide 6th or 7th … ways

Provide "extra" associativity, but not for all sets

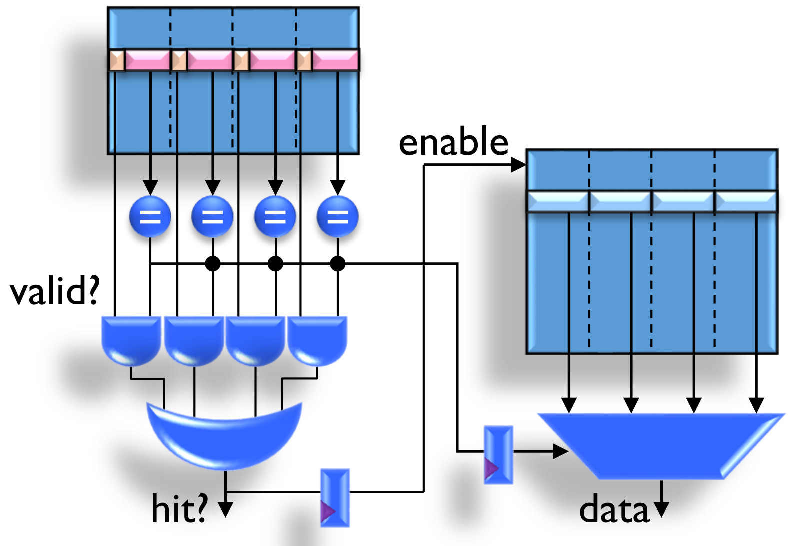
Parallel vs. Serial Caches

- Tag and Data usually separate SRAMs
  - 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+)



Cache, TLB & Address Translation (1)

- Should we use virtual address or physical address to access caches?
  - In theory, we can use either
- Drawback(s) of physical
  - TLB access has to happen before cache access → increasing hit time
- Drawback(s) of virtual
  - Aliasing problem: same physical memory might be mapped using multiple virtual addresses
  - Memory protection bits (part of page table and TLB) should be checked
  - I/O devices usually use physical addresses

So, what should we do?

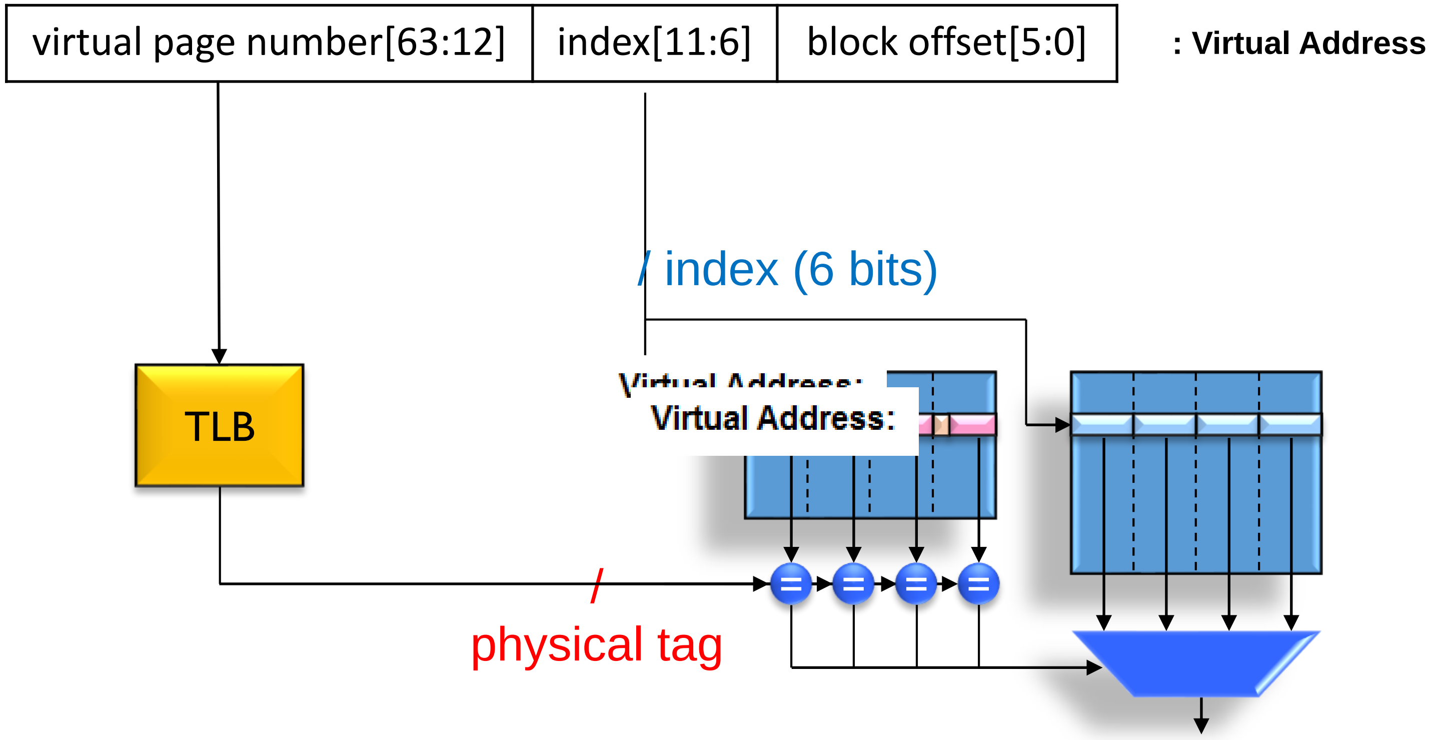
Cache, TLB & Address Translation (2)

- Observation: caches use addresses for two things
  - **Indexing**: to find and access the set that could contain the cache block
    - Only requires a small subset of low-order address bits
  - **Tag matching**: to search the blocks in the set to see if anyone is actually the one we're looking for
    - Requires the complete address
- Solution:
  1) Use part of address common between virtual and physical for indexing
  2) While the set is being accessed, do TLB lookup in parallel
  3) Use physical address from (2) for tag matching

Cache, TLB & Address Translation (3)

- Example: in Intel processors, page size is 4KB and cache block is 64 bytes
  - Page offset is 12 bits
  - Block offset is 6 bits
- What is the max. number of index bits that are common between virtual and physical addrs?
  - 12 – 6 = 6
- What is largest direct-mapped cache that we can build using 6 bits of index?
  - 2^6 blocks × 64 bytes-per-block = 4 KB (same as page size)
- But Intel L1 caches are 32KB. How do they do that?
  - Make the cache 8-way set associative. Each way is 4KB and still only needs 6 bits of index.

Cache, TLB & Address Translation (4)



- By removing TLB from critical path, we reduce the *hit-time* component of AMAT

Caches and Writes

- Writes are more interesting (i.e., complicated) than reads
  - On reads, tag and data can be accessed in parallel
  - On writes, we need two steps
    - First, do indexing and tag matching to find the block
    - Then, write the data to the SRAM

Cache Writes Policies (1)

- On write hits, update lower-level memory?
  - Yes: **write-through** (more memory traffic)
  - No: **write-back** (uses *dirty* state bits to identify blocks to write back)
- What is the drawback of write-back?
  - On a block replacement, should first write the old block back to memory if dirty, increasing the miss penalty
  - With write-through, cache blocks are always "clean", so no need to write back
- In multi-level caches, you can have a mix
  - For example, write-through for L1 and write-back for L2

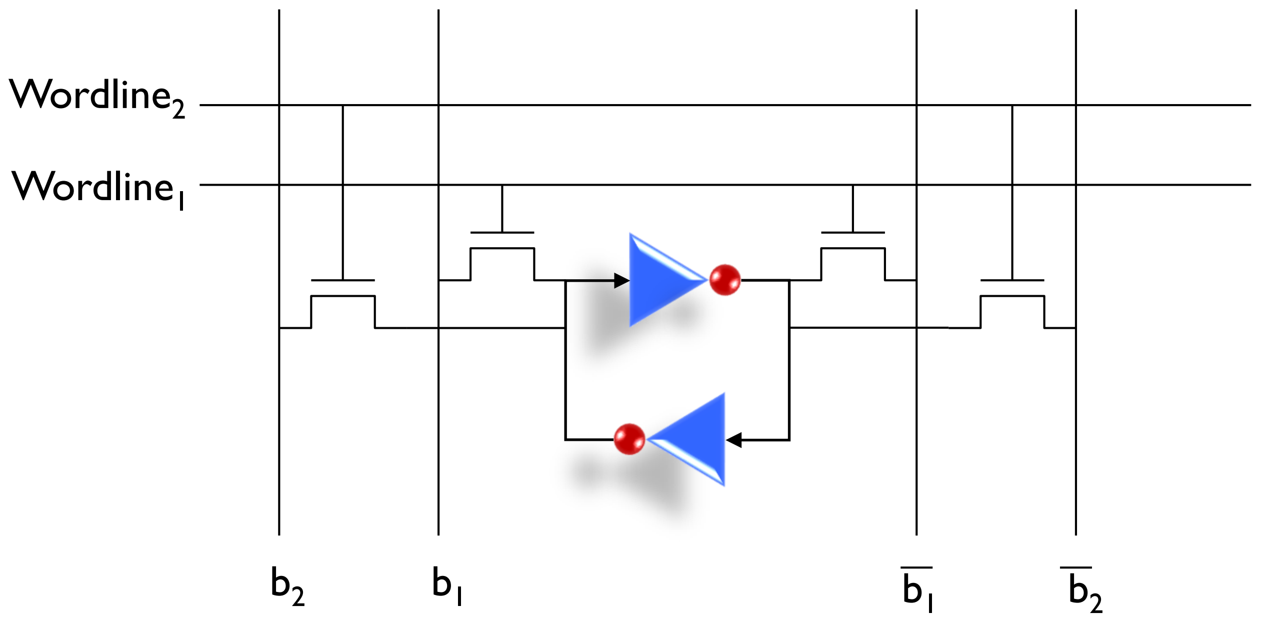
Caches Writes Policies (2)

- On write misses, allocate a cache block frame?
  - Yes: **write-allocate**
    - Bring the data in from the lower level, allocate a cache frame, and then do the write
    - More common in write-back caches
  - No: **no-write-allocate**
    - Do not allocate a cache frame. Just send the write to the lower level
    - More common in write-through caches
- For your HW2, you will implement a write-back, write-allocate cache

Multiple Accesses per Cycle

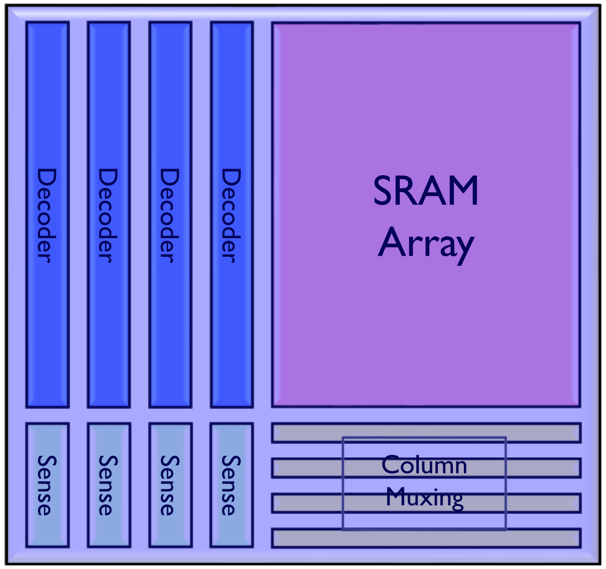
- Super-scalars might make multiple parallel cache accesses
  - Core can make multiple L1$ access requests per cycle
    - E.g., 2 simultaneous L1 D$ accesses in Intel processors
  - Multiple cores can access LLC at the same time
- Must either delay some requests, or…
  - Design SRAM with multiple ports
    - Big and power-hungry
  - Split SRAM into multiple banks
    - Can result in delays, but usually not

Multi-Ported SRAMs

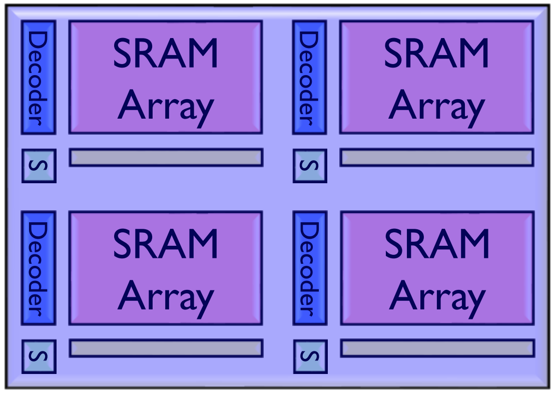


Wordlines = 1 per port
Bitlines = 2 per port $\rightarrow$ Area = $O(ports^2)$

Multi-Porting vs. Banking



4 ports
Big (and slow)
Guarantees concurrent access

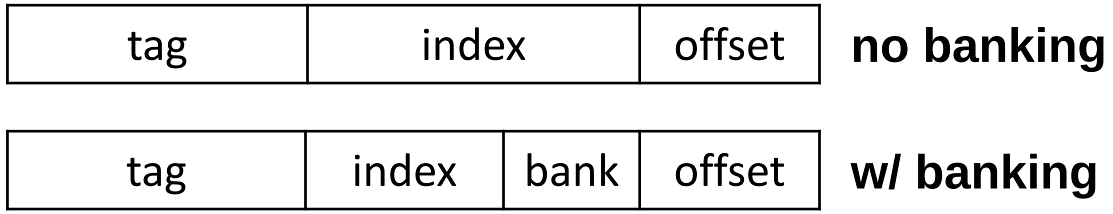


4 banks, 1 port each
Each bank small (and fast)
Conflicts (delays) possible

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%