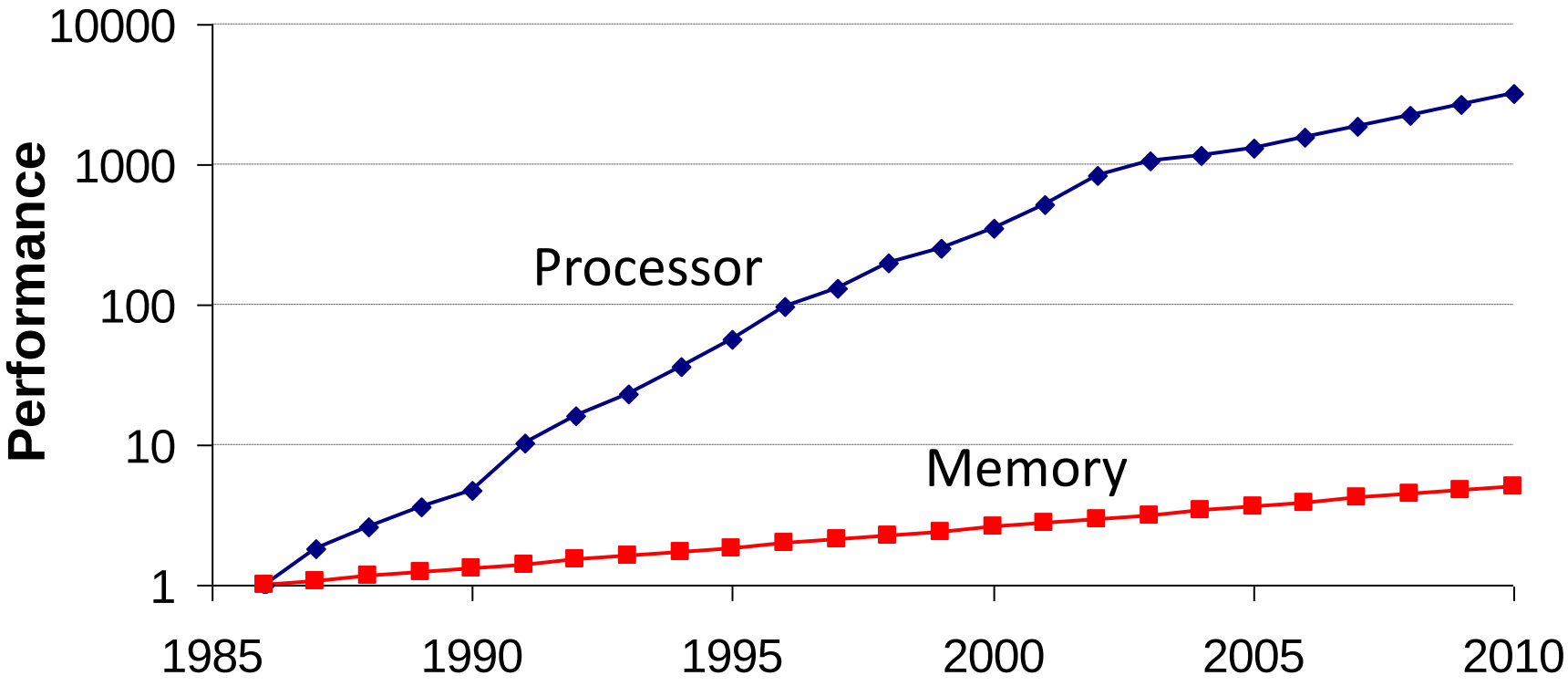
CSE 502: Computer Architecture

Memory Hierarchy & Caches

This Lecture

- Memory Hierarchy
- Caches / SRAM
- Cache Organization and Optimizations

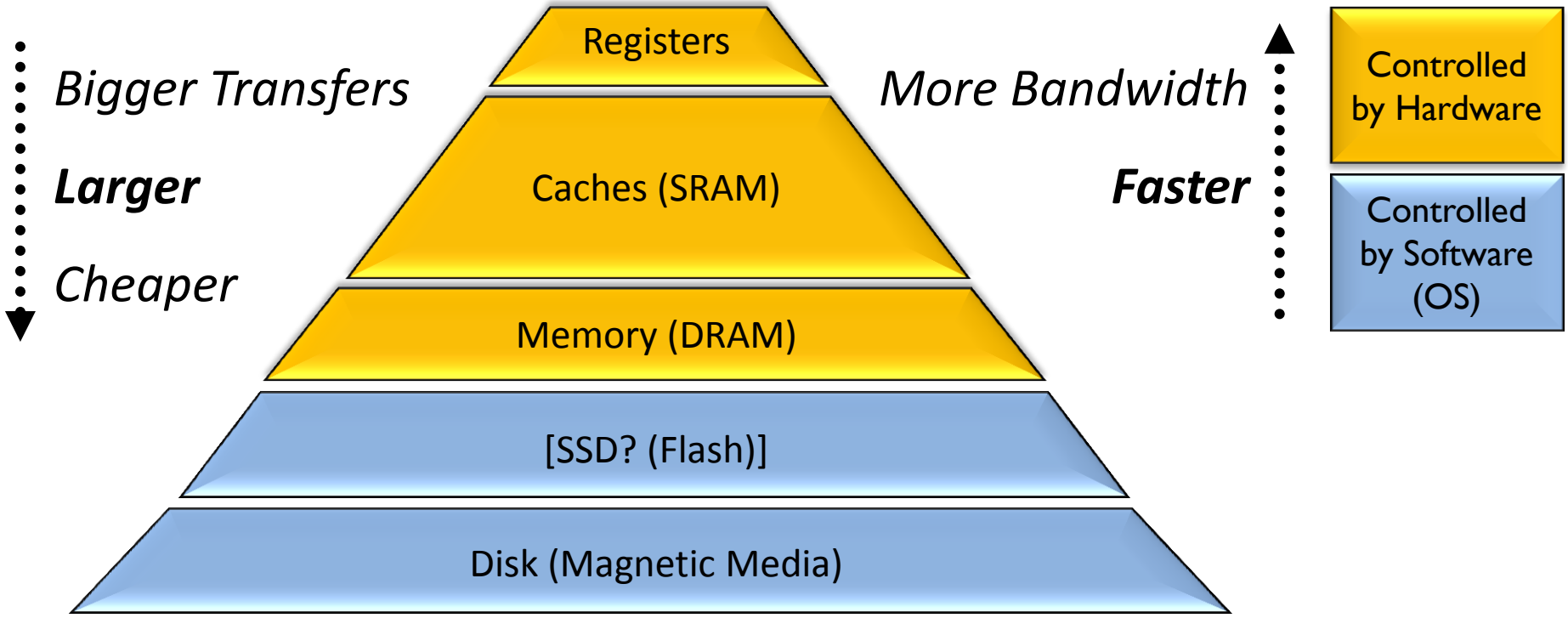
Motivation



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

Storage Hierarchy

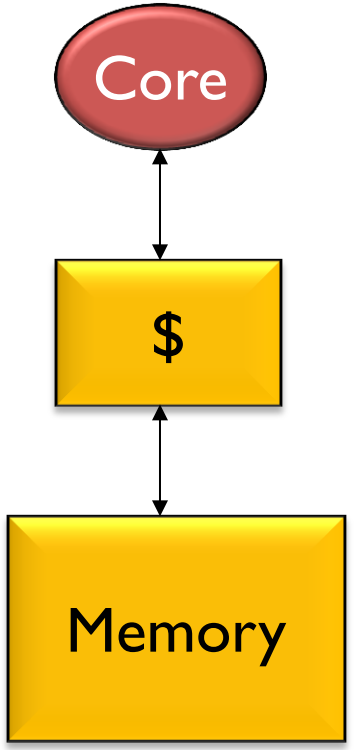
- Make common case fast:
  - Common: temporal & spatial locality
  - Fast: smaller more expensive memory



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*



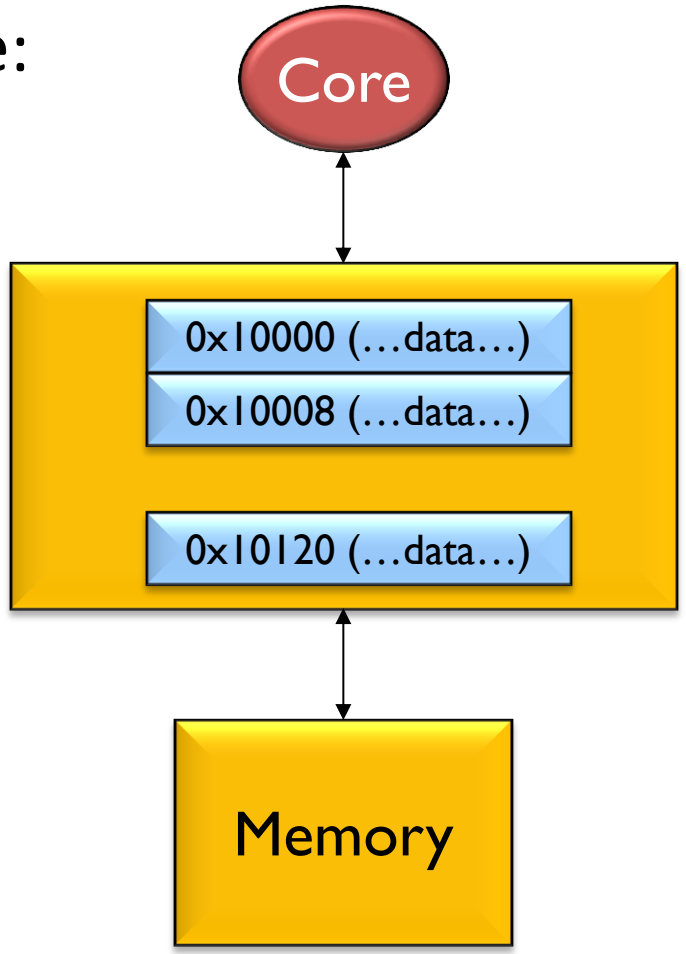
Cache Terminology

- *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%

AMAT (1/2)

- 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

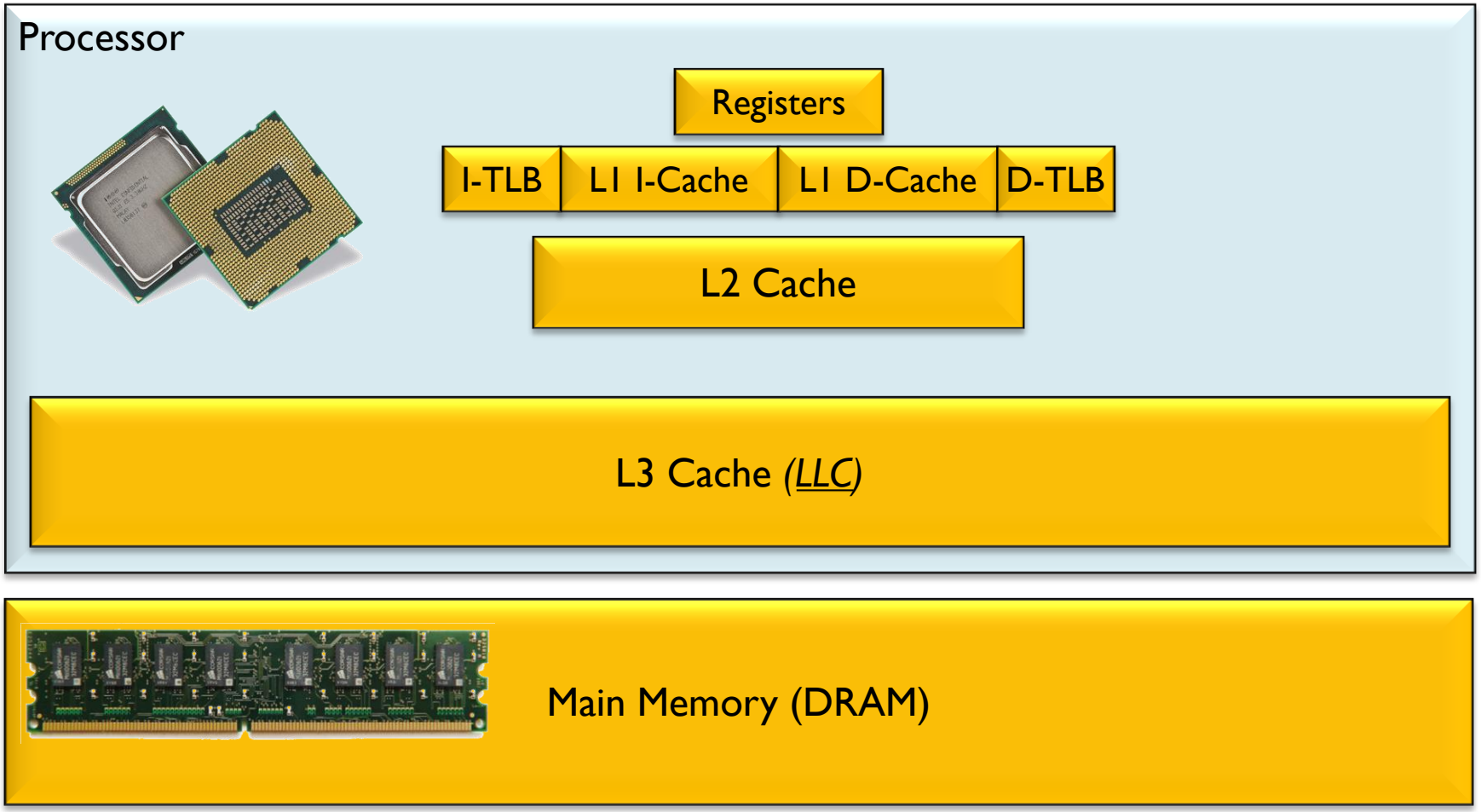
AMAT (2/2)

- Generalizes nicely to any-depth hierarchy

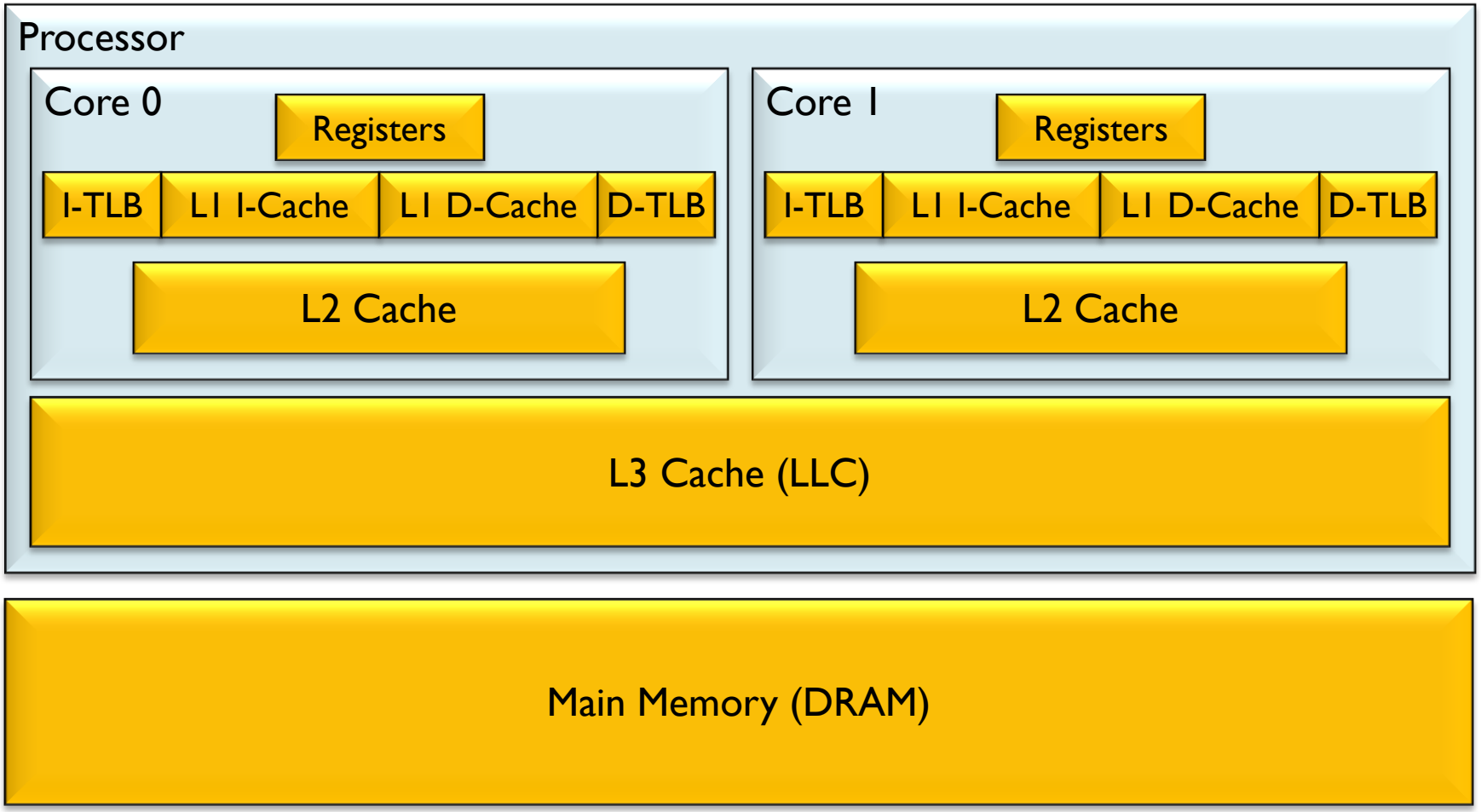
- If …
  L1 cache hit is 5 cycles (core to L1 and back)
  L2 cache hit is 20 cycles (core to L2 and back)
  memory access is 100 cycles (core to mem and back)

- 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)



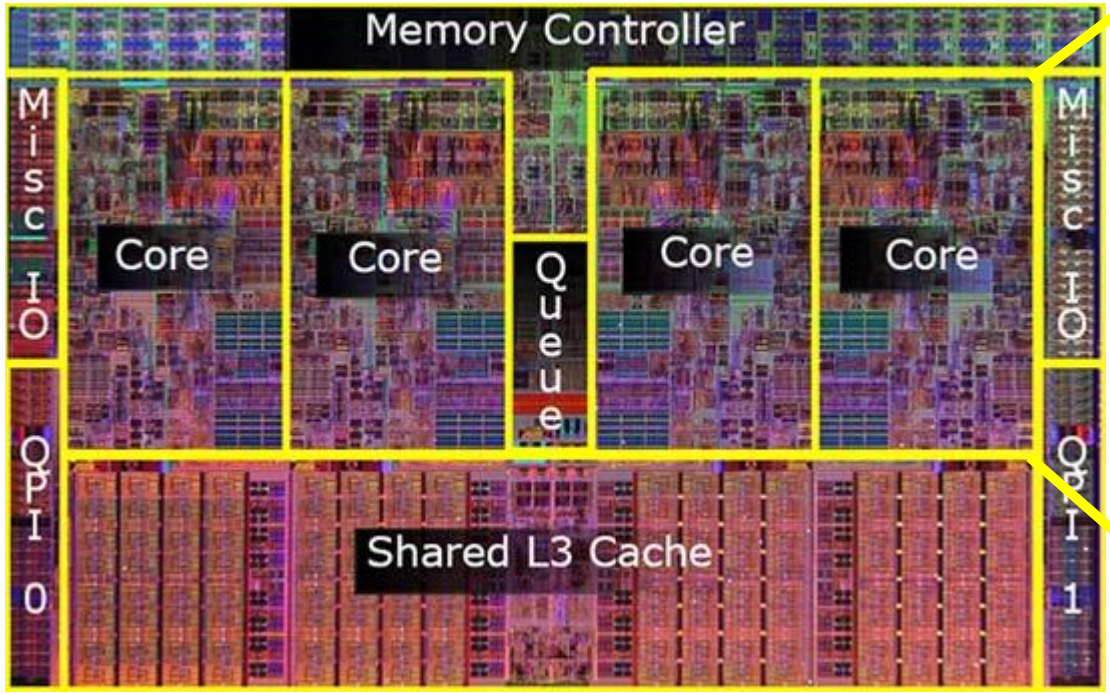
Memory Organization (2/3)

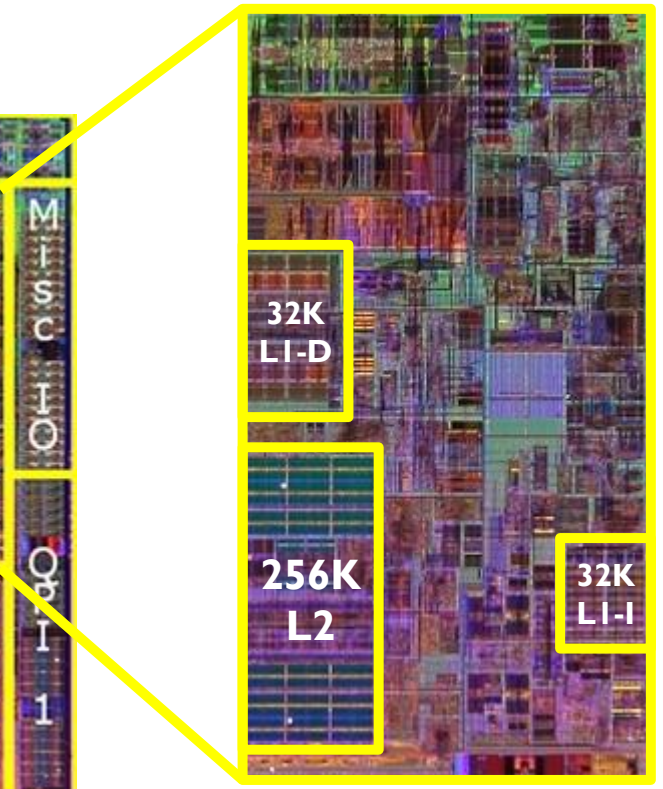


Multi-core replicates the top of the hierarchy

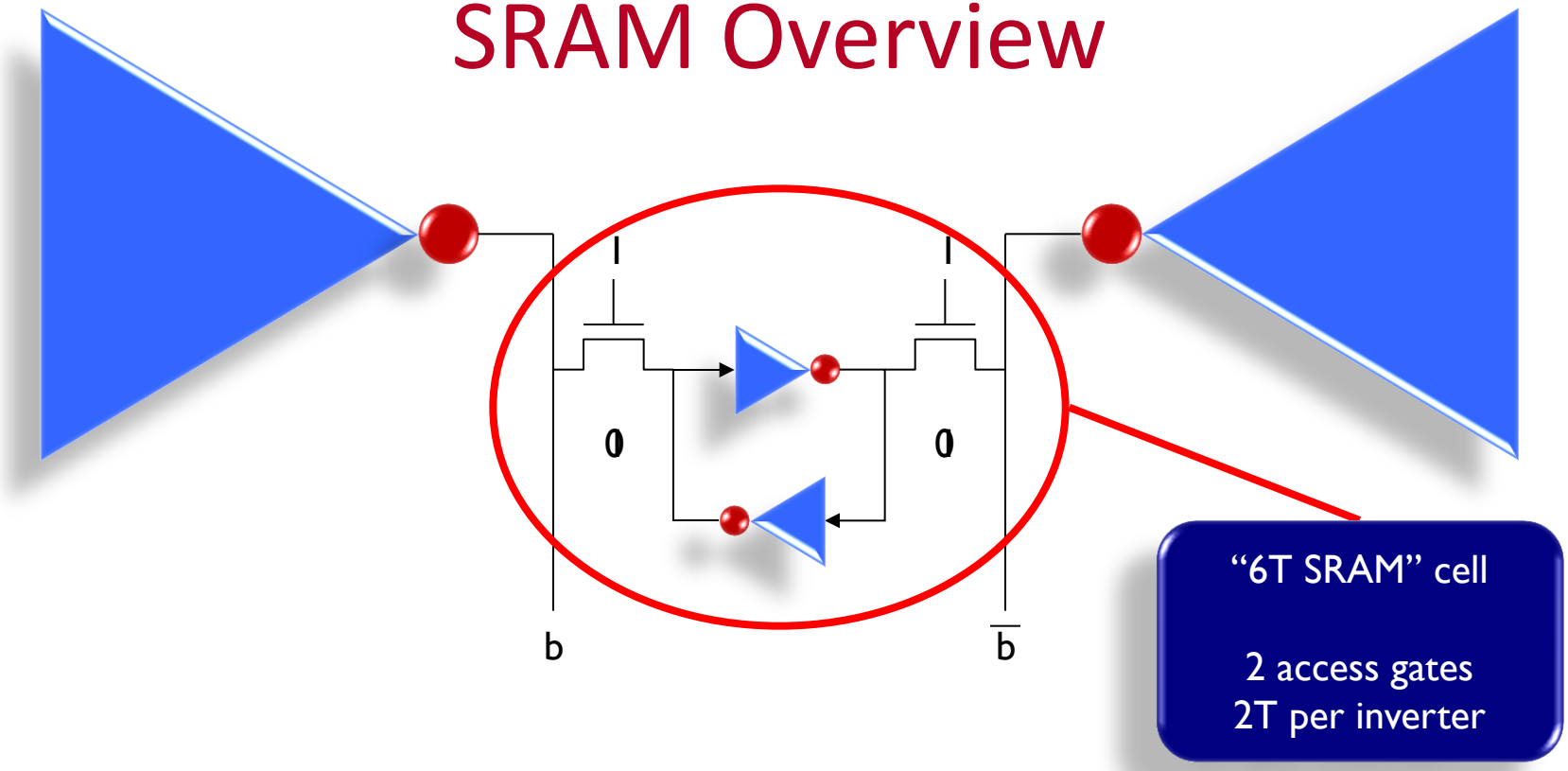
Memory Organization (3/3)

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



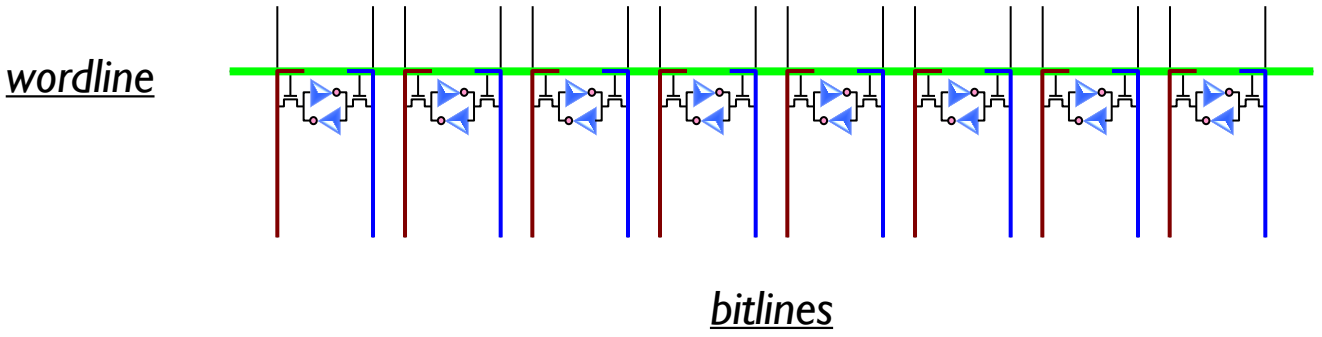


SRAM Overview



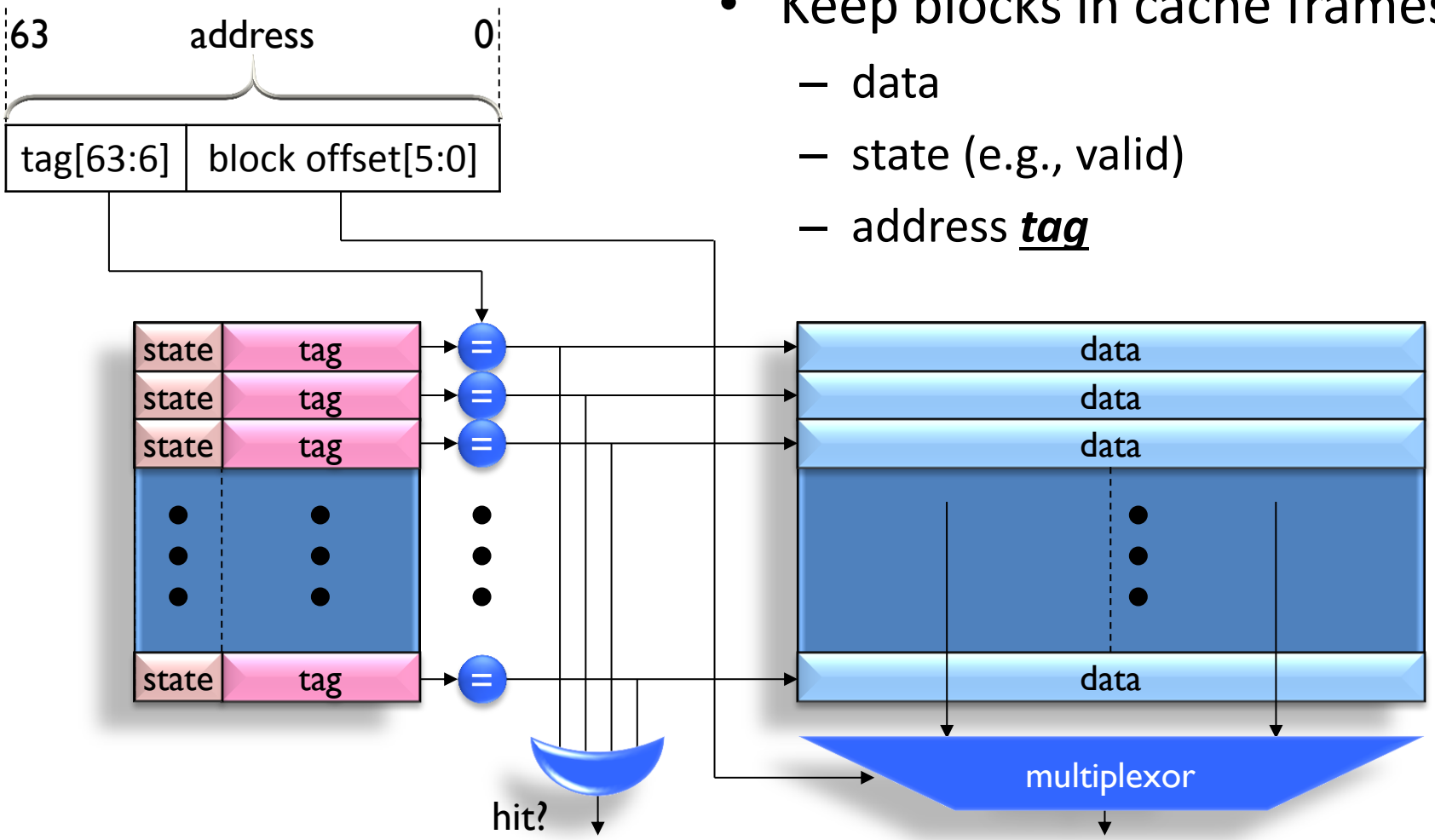
- Chained inverters maintain a stable state
- Access gates provide access to the cell
- Writing to cell involves over-powering storage inverters

8-bit SRAM Array



Fully-Associative Cache

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



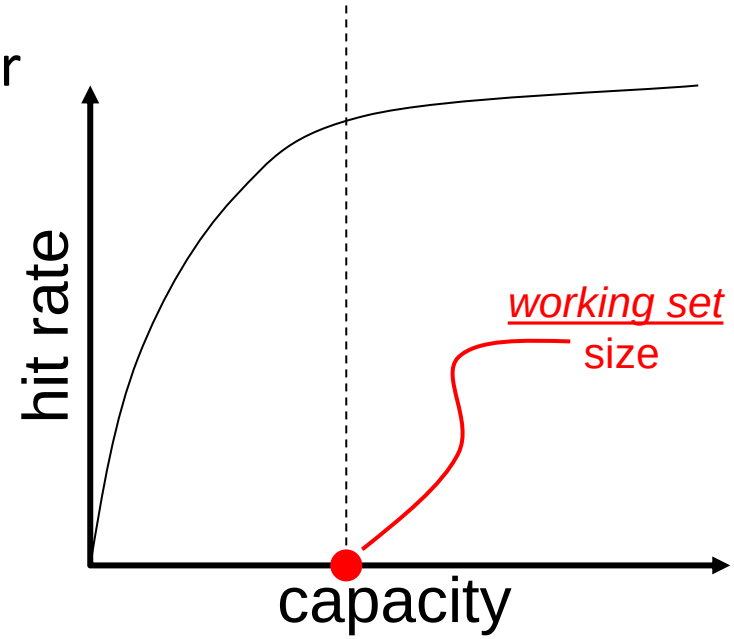
What happens when the cache runs out of space?

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*: (To appear in multi-core lecture)

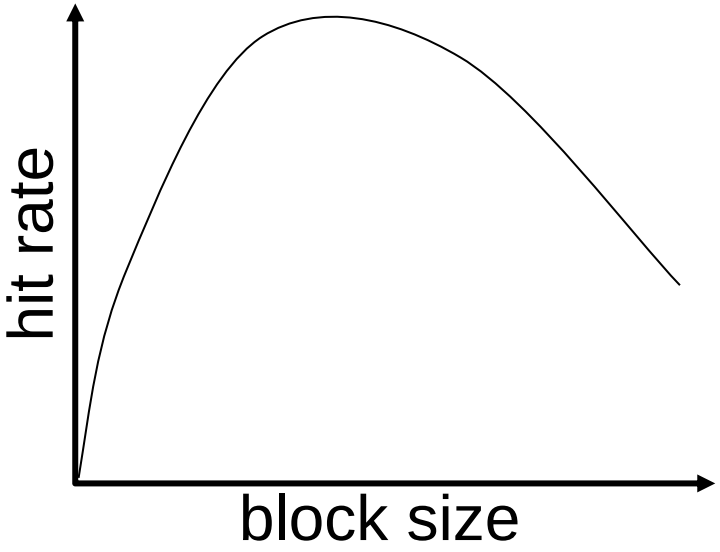
Cache Size

- 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

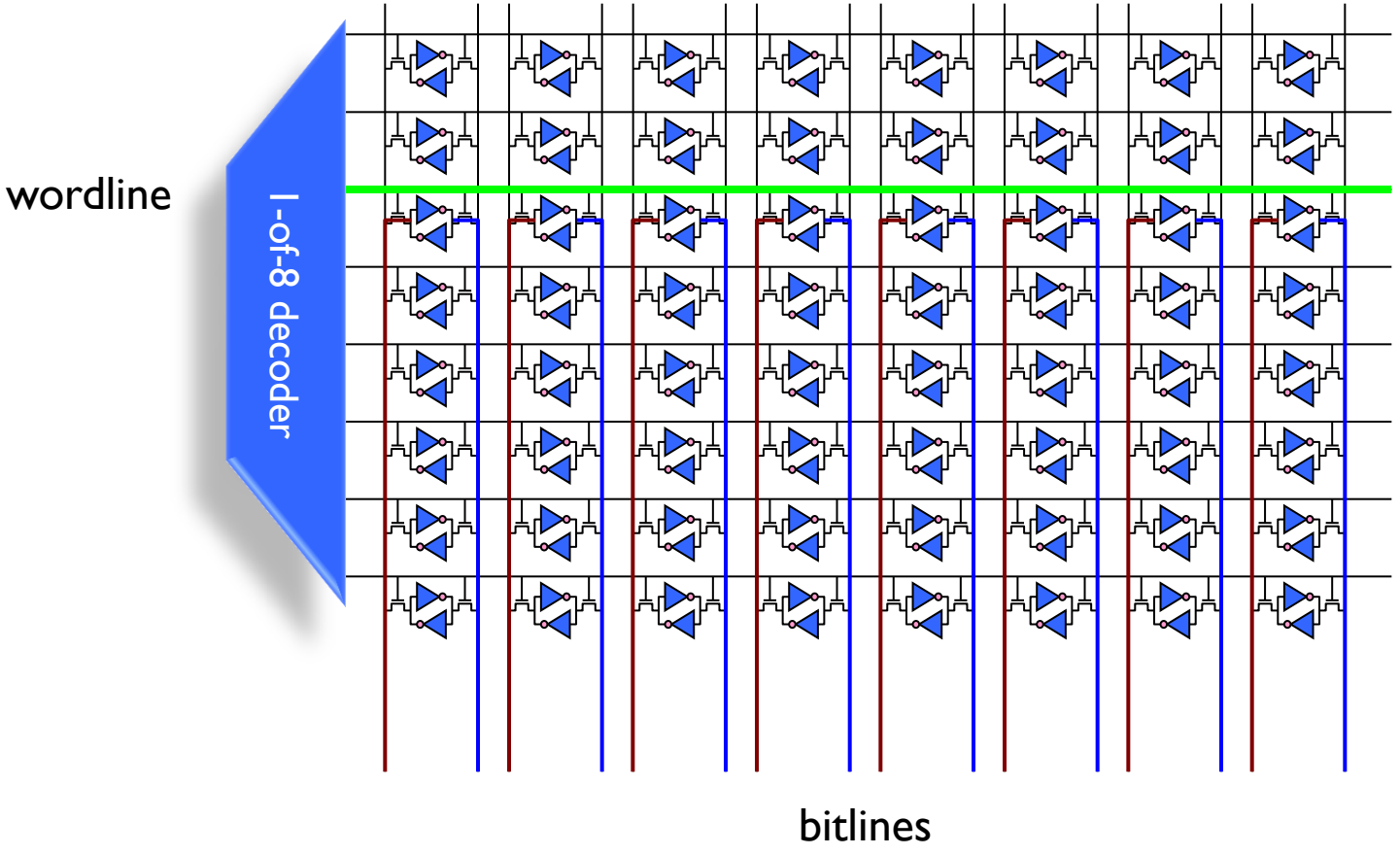


Block Size

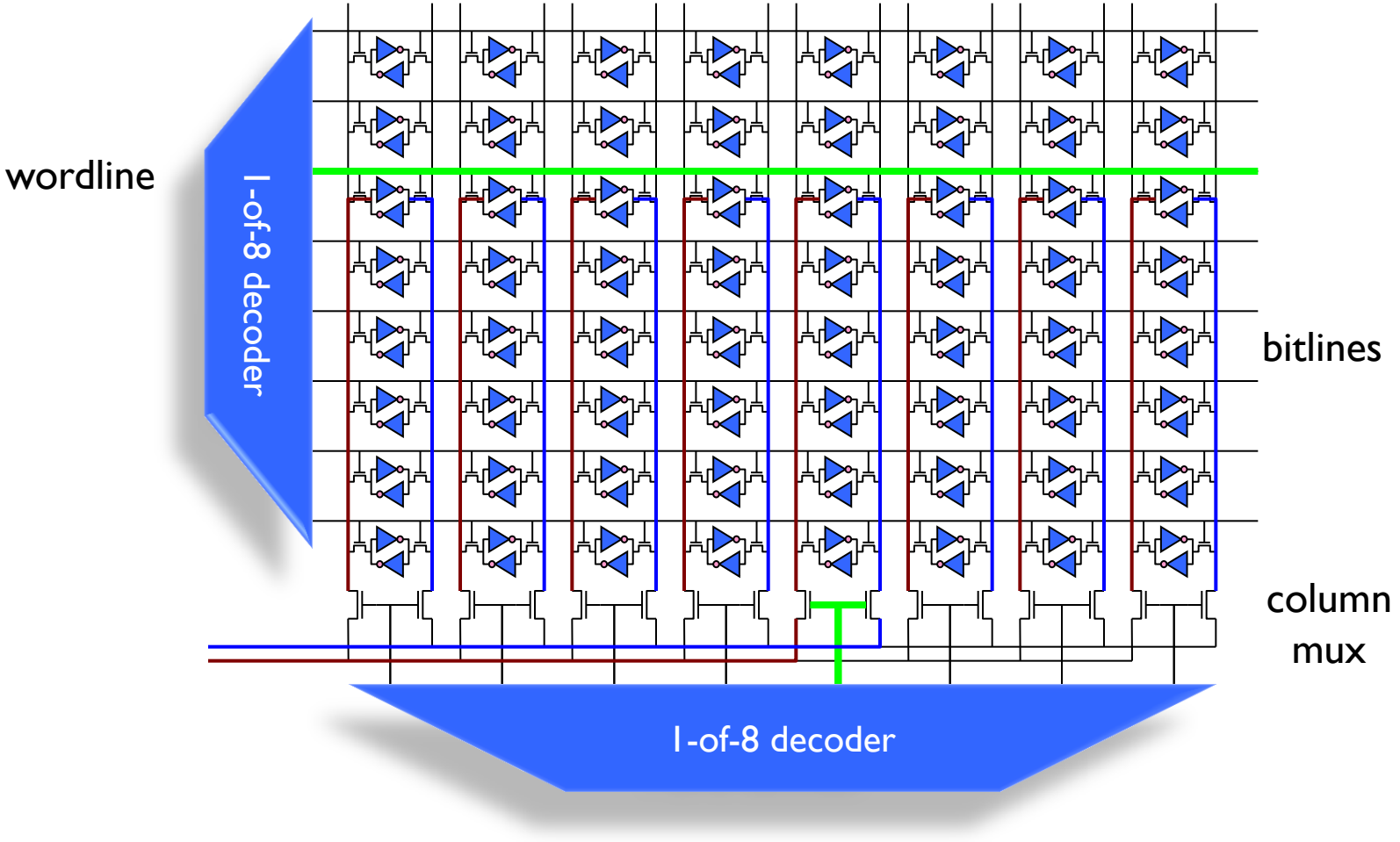
- Block size is the data that is
  - Associated with an address tag
  - Not necessarily the unit of transfer between hierarchies
- Too small a block
  - Don't exploit spatial locality well
  - Excessive tag overhead
- Too large a block
  - Useless data transferred
  - Too few total blocks
    - Useful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



8×8-bit SRAM Array

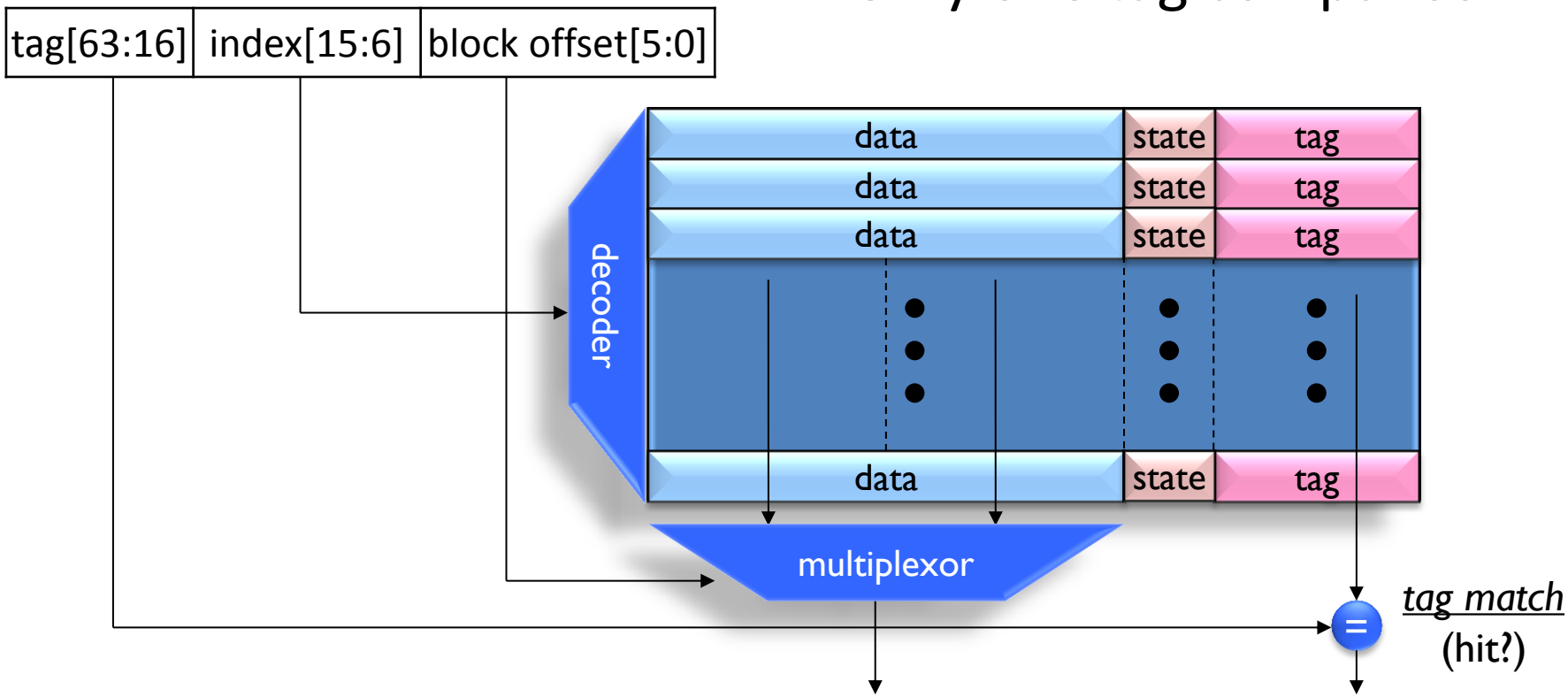


64×1-bit SRAM Array



Direct-Mapped Cache

- Use middle bits as index
- Only one tag comparison



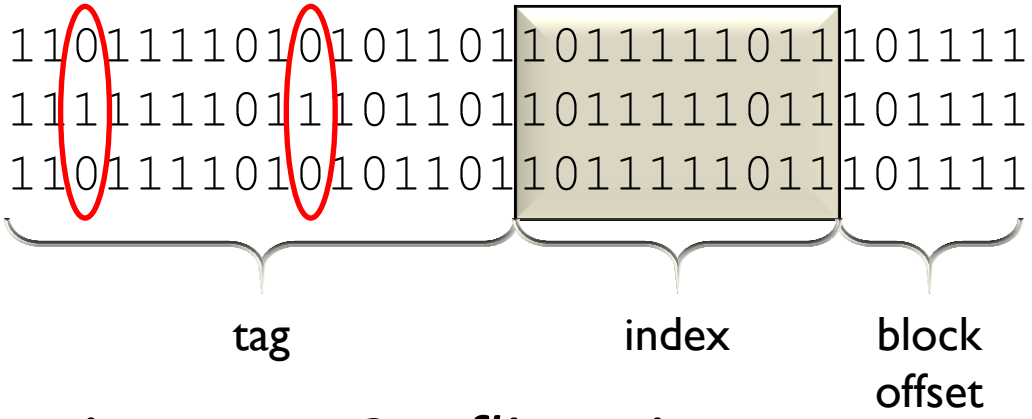
Why take index bits out of the middle?

Cache Conflicts

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

Address sequence:

```
0xDEADBEEF   11011111010101101011111011101111
0xFEEDBEEF   11111111011101101011111011101111
0xDEADBEEF   11011111010101101011111011101111
```

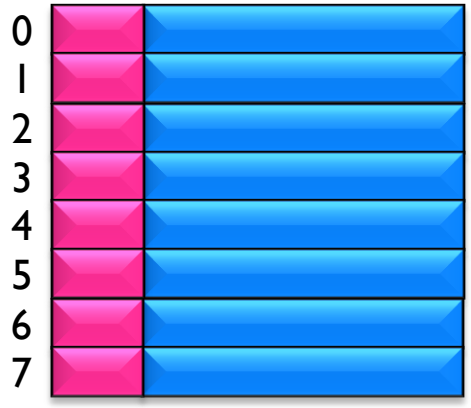


- 0xDEADBEEF experiences a *Conflict* miss
  - Not Compulsory (seen it before)
  - Not Capacity (lots of other indexes available in cache)

Associativity (1/2)

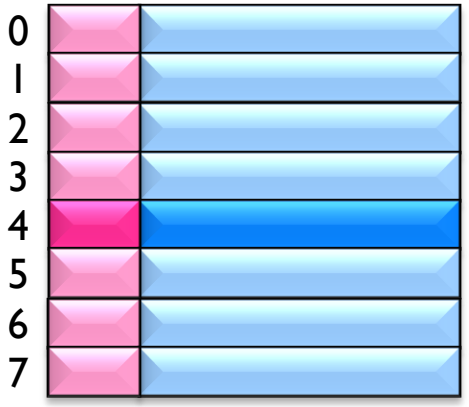
- Where does block index 12 (b'1100) go?

Block



Set/Block

Set



Fully-associative
block goes in any frame

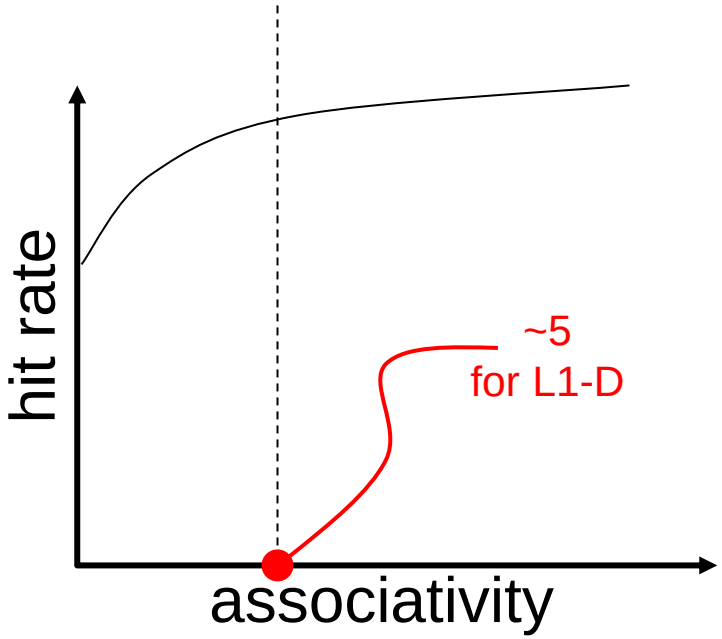
(all frames in 1 set)

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

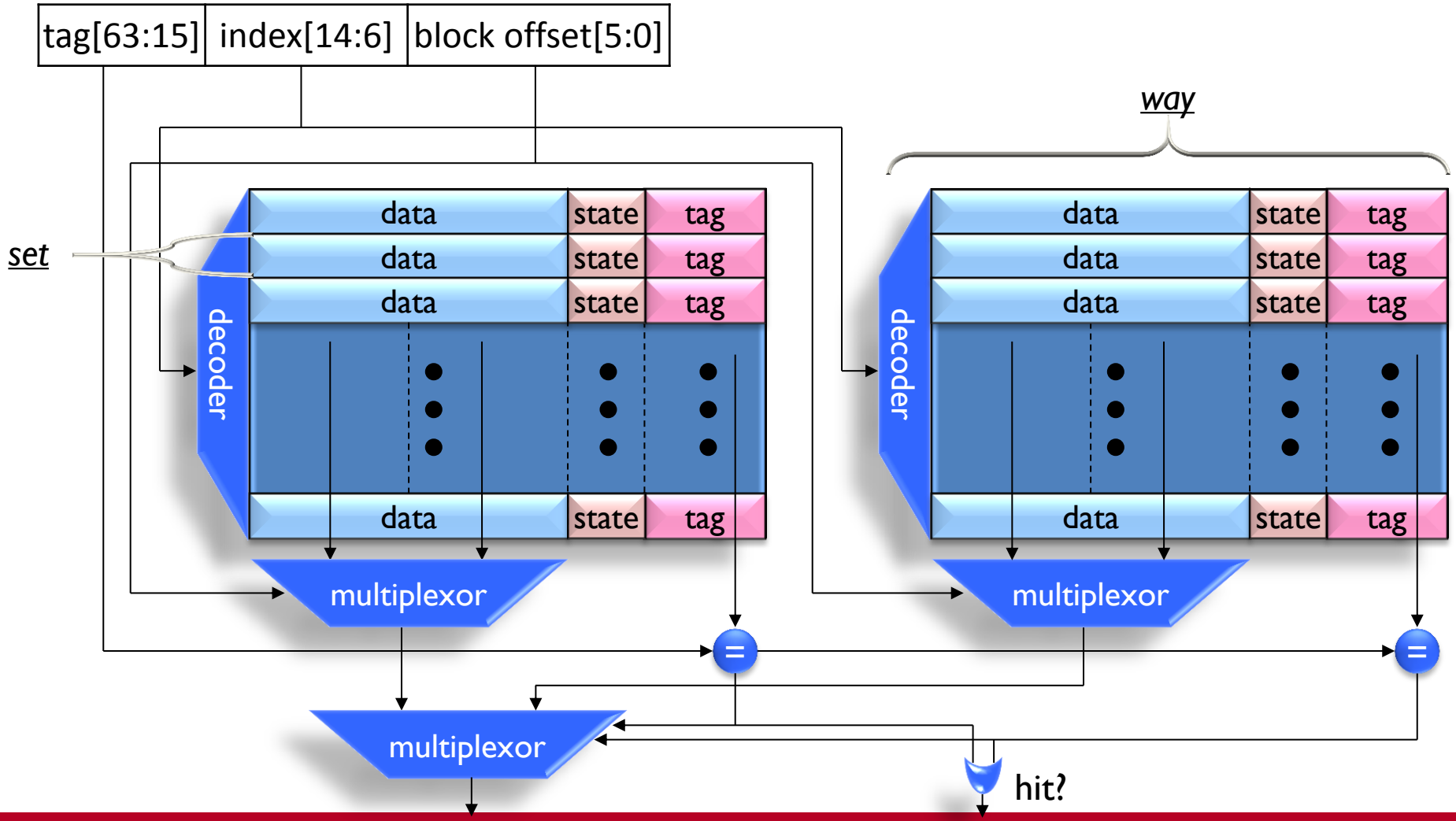
Direct-mapped
block goes in exactly one frame
(1 frame per set)

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?

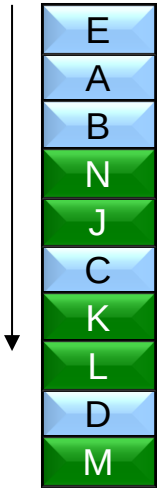
- *Least Recently Used (LRU)*
  - Optimized for temporal locality (expensive for >2-way)
- *Not Most Recently Used (NMRU)*
  - Track MRU, random select among the rest
- *Random*
  - Nearly as good as LRU, sometimes better (when?)

Victim Cache (1/2)

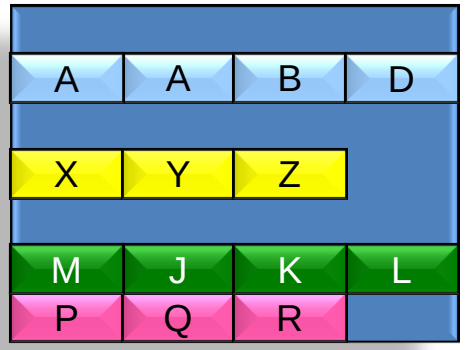
- Associativity is expensive
  - Performance from extra muxes
  - Power from reading and checking more tags and data
- Conflicts are expensive
  - Performance from extra mises
- Observation: Conflicts don't occur in all sets

Victim Cache (2/2)

Access Sequence:

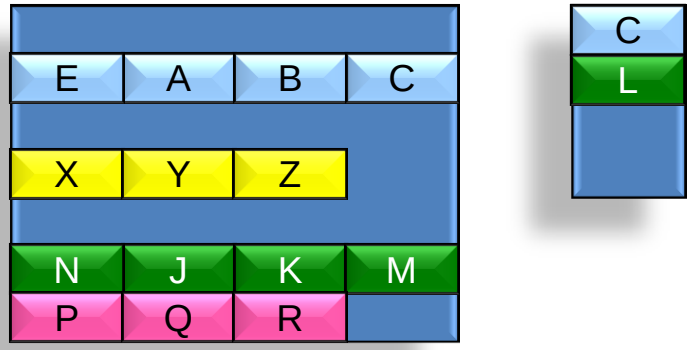


4-way Set-Associative L1 Cache



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

4-way Set-Associative L1 Cache + Fully-Associative Victim Cache



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

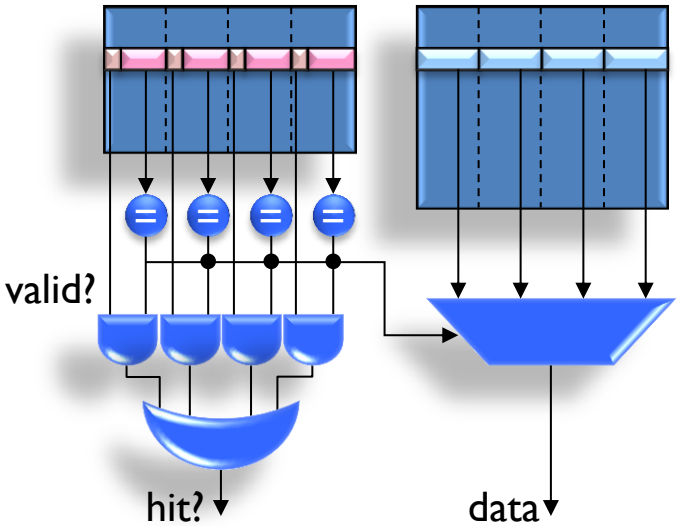
Can even provide 6th or 7th … ways

Provide "extra" associativity, but not for all sets

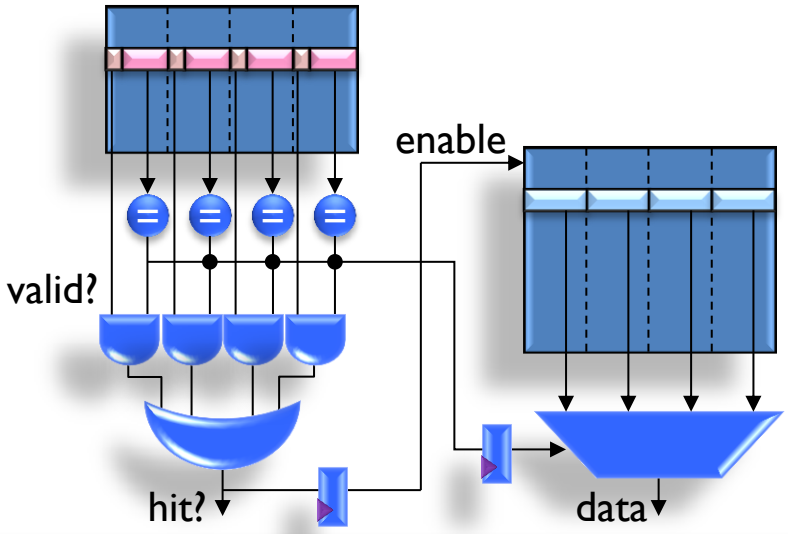
Parallel vs Serial Caches

- Tag and Data usually separate (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+)

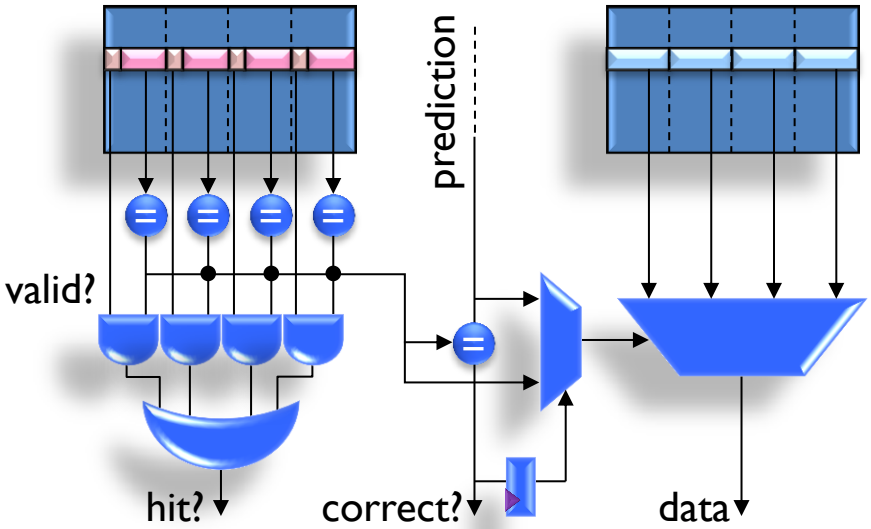


Way-Prediction and Partial Tagging

- Sometimes even parallel tag and data is too slow
  - Usually happens in high-associativity L1 caches
- If you can't wait… guess

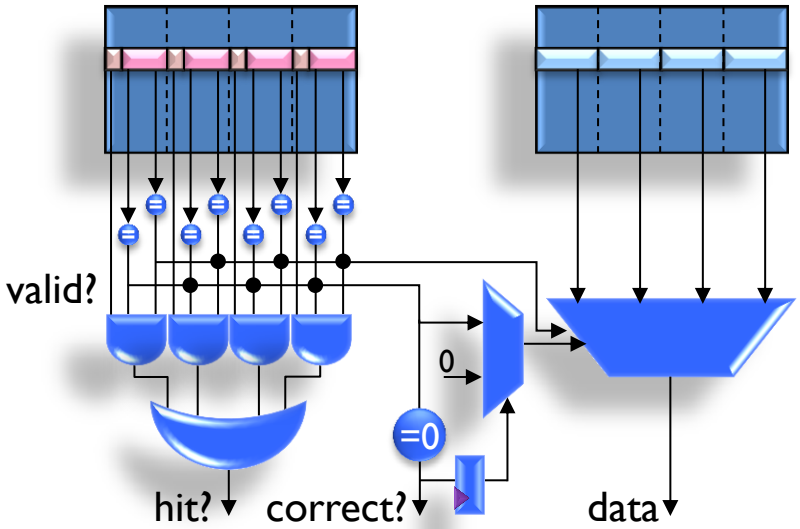
Way Prediction

Guess based on history, PC, etc…



Partial Tagging

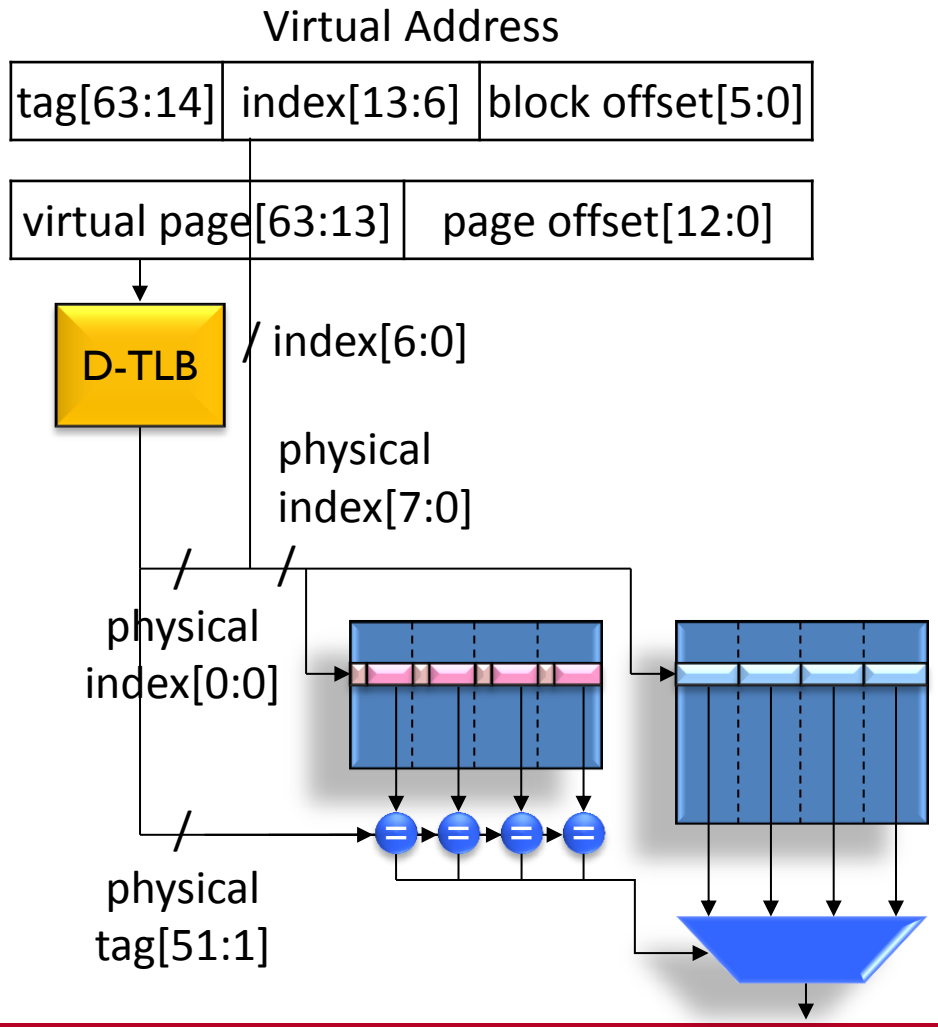
Use part of tag to pre-select mux



Guess ***and*** verify, extra cycle(s) if not correct

Physically-Indexed Caches

- Core requests are VAs
- Cache index is PA[15:6]
  - VA passes through TLB
  - D-TLB on critical path
- Cache tag is PA[63:16]
- If index size < page size
  - Use VA for index

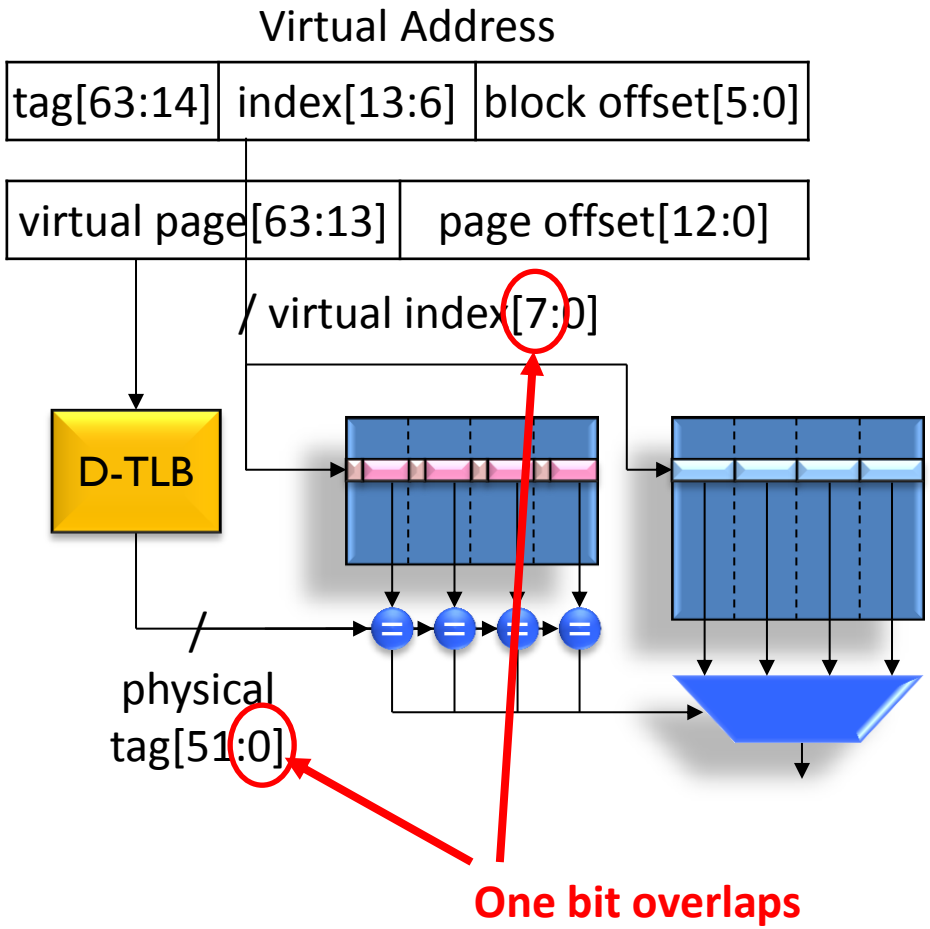


Simple, but slow. Can we do better?

Virtually-Indexed Caches

- Core requests are VAs
- Cache index is VA[15:6]
- Cache tag is PA[63:16]

- Why not tag with VA?
  - Cache flush on ctx switch
- Virtual aliases
  - Ensure they don't exist
  - … or check all on miss



Fast, but complicated

Compute Infrastructure

- "rocks" cluster and several VMs
  - Use your "Unix" login and password
    - allv21.all.cs.sunysb.edu
    - sbrocks.cewit.stonybrook.edu
  - Once on rocks head node, run "condor_status"
    - Pick a machine and use ssh to log into it (e.g., compute-4-4)
  - Don't use VM for Homework
- I already sent out details regarding software
  - Including MARSS sources and path to boot image
  - Go through
    - http://marss86.org/~marss86/index.php/Getting_Started
    - http://cloud.github.com/downloads/avadhpatel/marss/Marss_ISCA_2012_tutorial.pdf

Send email to list in case you encounter problems

Homework #1 (of 1) part 2

- Already have accurate & precise time measurement
- Extend your program to…
  - Measure how many cache levels there are in the CPU
  - Measure the hit time of each cache level
- Output should resemble this:

```
123..890 Loop: 2500000000ns
Cache Latencies: 2ns 6ns 20ns 150ns
```

Note: You must measure it, not read it from MSRs

Homework #1 Hints

- Instructions take time – use them wisely
  - Use high optimization level in compiler (-O3)
  - Check assembly to see instructions (use -S flag in compiler)
  - Independent instructions run in parallel (don't count time)
  - Dependent instructions run serially (precisely count time)
  - Use random walks in memory (sequential will trick you)
- Smallest linked-list walk:

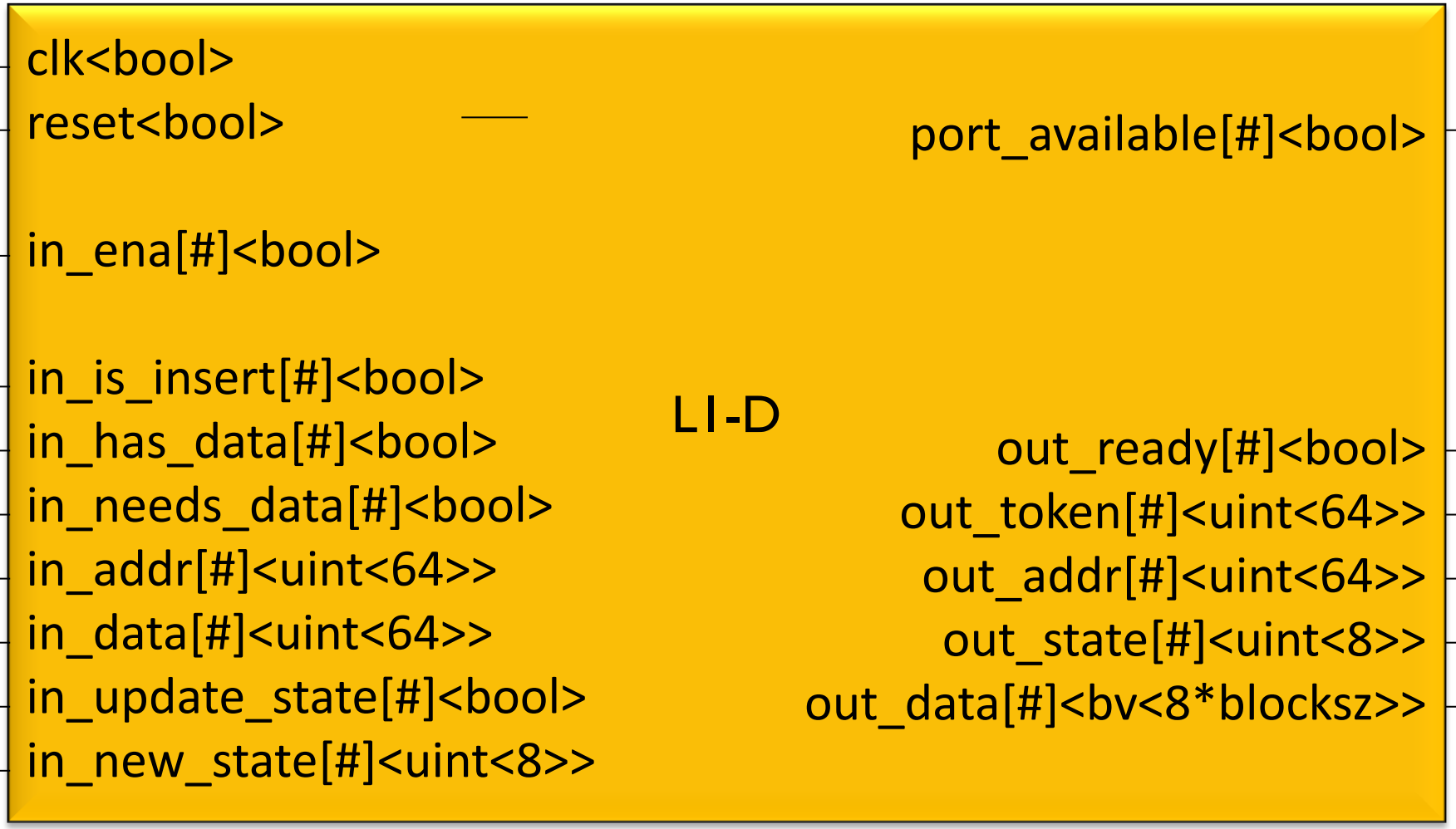
```
void* list[100];
for(y=0;y<100;++y) list[y]=&list[(y+1)%100];
void* x = &list[0];
x = *x; // mov %rax,(%rax)
```

Warm-Up Project (1/3)

- Build an L1-D cache array
- Determine groups
  - Email member list to Connor and CC me (by Feb 20)
- Must use SRAM<> module for storage
- Grading reminder:

Warm-up Project	Points
1 Port,Direct-Mapped,16K	10
2 Port,Direct-Mapped,16K	12
1 Port,Set-associative,16K	14
2 Port,DM,64K,Pipelined	16
2 Port,SA,64K,Pipelined	18
2 Port,SA,64K,Pipel.,Way-pred.	20

Warm-Up Project (2/3)

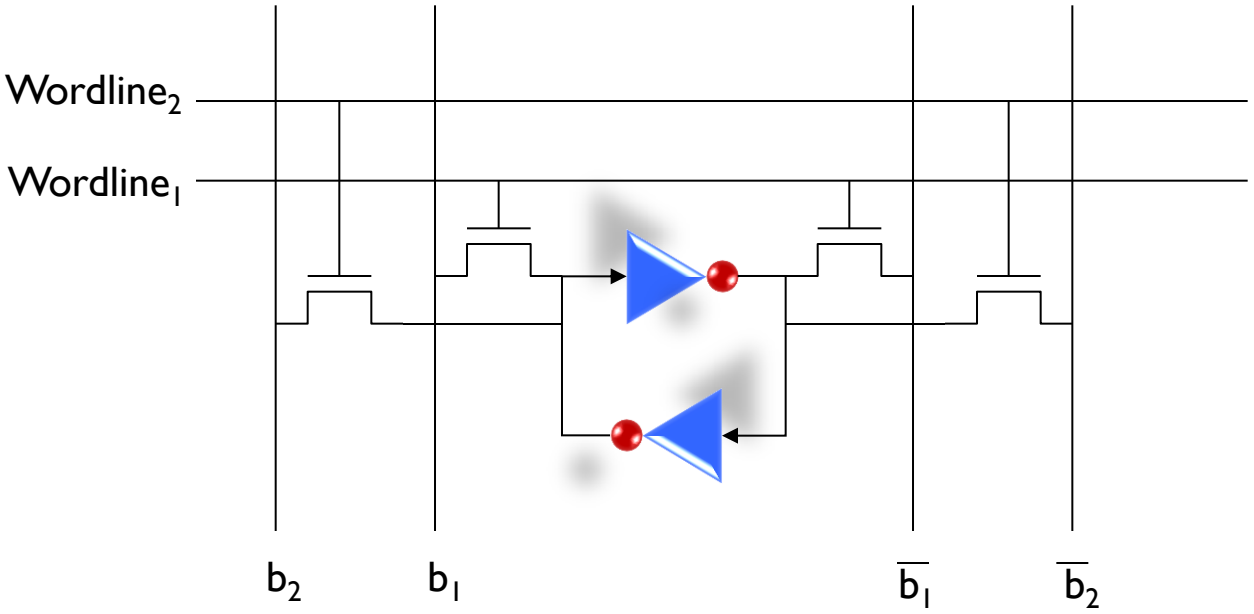


Warm-Up Project (3/3)

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

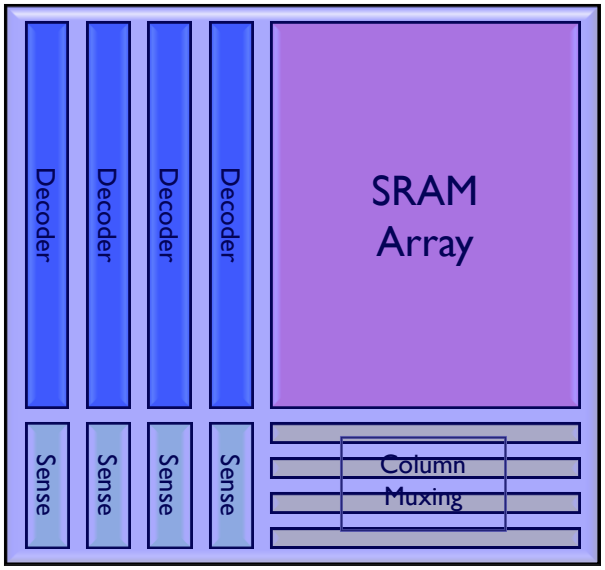
Multi-Ported SRAMs



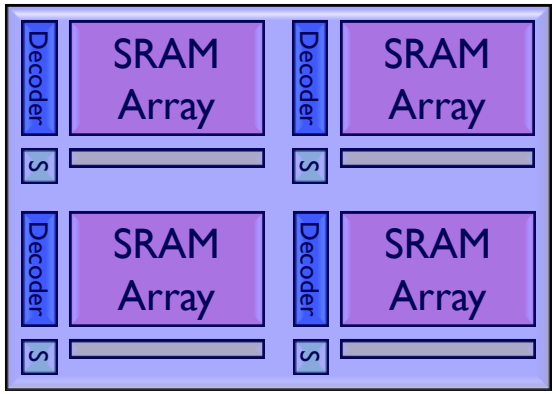
Wordlines = 2 × ports
Bitlines = 4 × ports

→ 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?

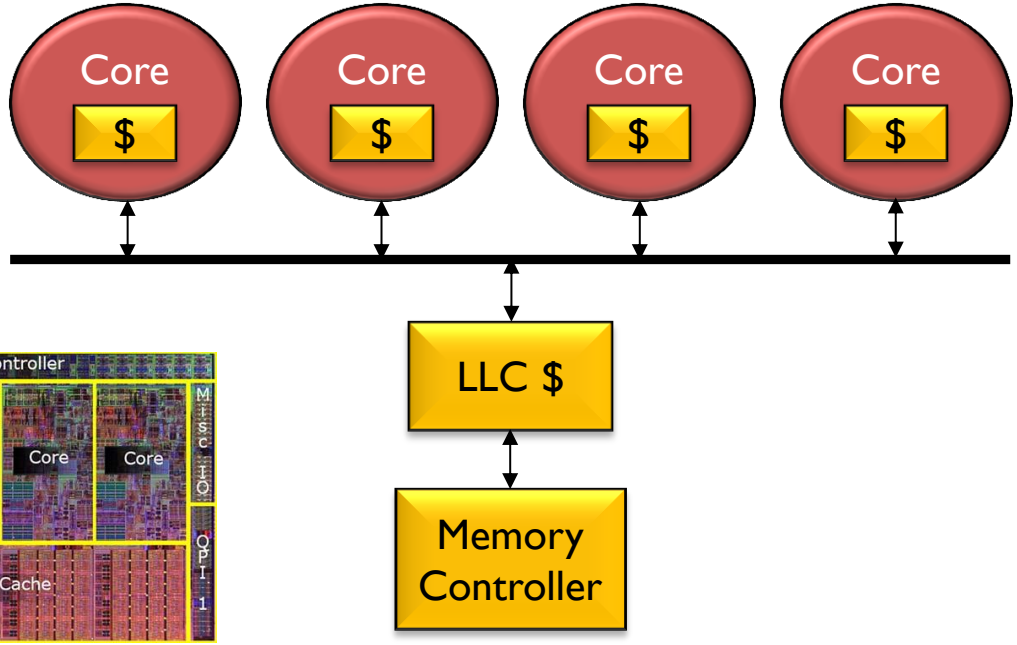
Bank Conflicts

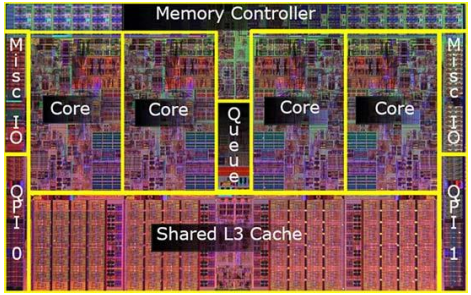
- Banks are *address interleaved*
  - For block size *b cache with N* banks…
  - Bank = (Address / b) % N
    - More complicated than it looks: just low-order bits of index
  - Modern processors perform *hashed cache indexing*
    - May randomize bank and index
    - XOR some low-order tag bits with bank/index bits (why XOR?)
- Banking can provide high bandwidth
- But only if all accesses are to different banks
  - For 4 banks, 2 accesses, chance of conflict is 25%

On-chip Interconnects (1/5)

- Today, (Core+L1+L2) = "*core*"
  - (L3+I/O+Memory) = "*uncore*"
- How to interconnect multiple "core"s to "uncore"?

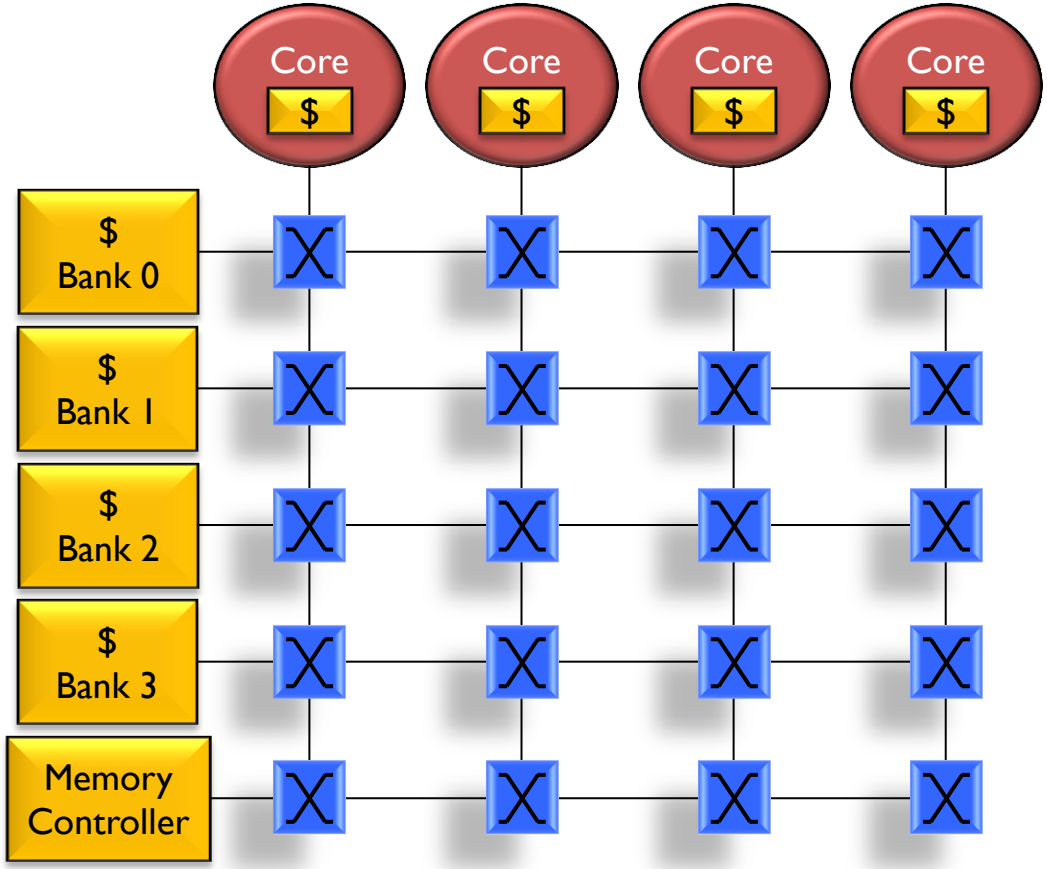
- Possible topologies
  - *Bus*
  - Crossbar
  - Ring
  - Mesh
  - Torus

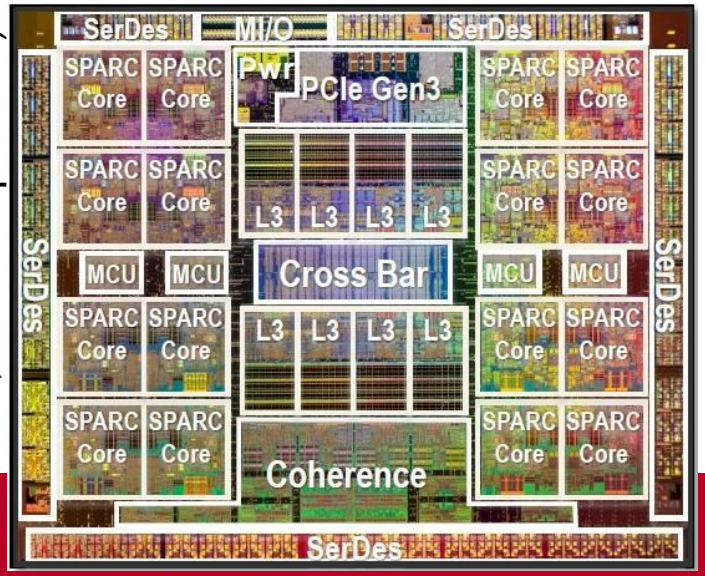




On-chip Interconnects (2/5)

- Possible topologies
  - Bus
  - *Crossbar*
  - Ring
  - Mesh
  - Torus

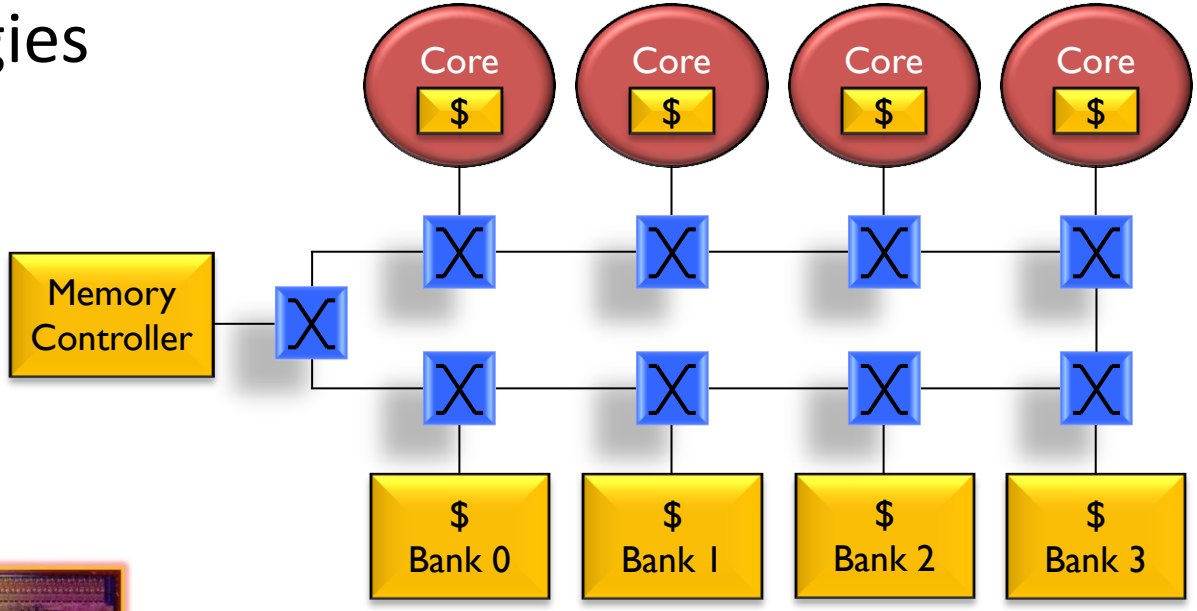


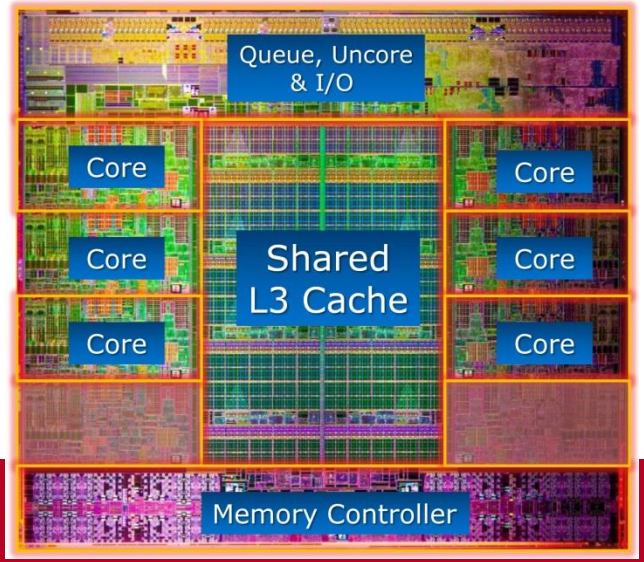


Oracle UltraSPARC T5 (3.6GHz, 16 cores, 8 threads per core)

On-chip Interconnects (3/5)

- Possible topologies
  - Bus
  - Crossbar
  - *Ring*
  - Mesh
  - Torus





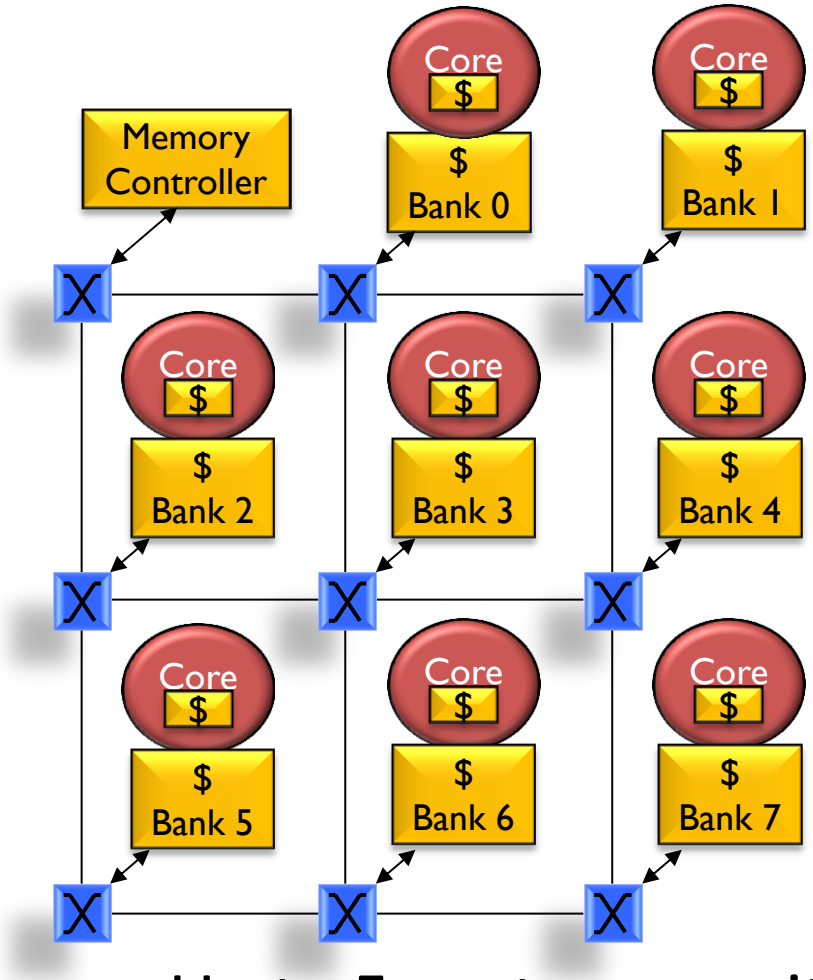
Intel Sandy Bridge (3.5GHz, 6 cores, 2 threads per core)

- 3 ports per switch
- Simple and cheap
- Can be bi-directional to reduce latency

On-chip Interconnects (4/5)

- Possible topologies
  - Bus
  - Crossbar
  - Ring
  - *Mesh*
  - Torus

Tilera Tile64 (866MHz, 64 cores)

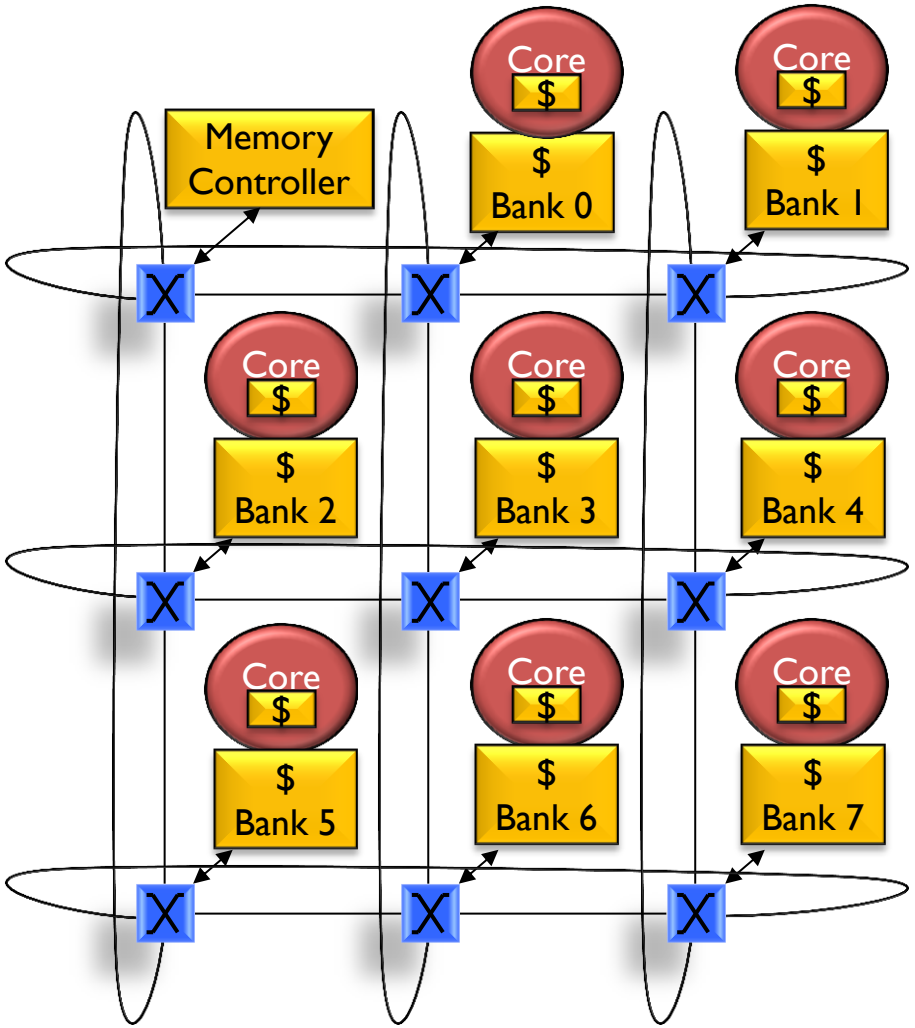


- Up to 5 ports per switch

Tiled organization combines core and cache

On-chip Interconnects (5/5)

- Possible topologies
  - Bus
  - Crossbar
  - Ring
  - Mesh
  - *Torus*
- 5 ports per switch
- Can be "folded" to avoid long links



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?

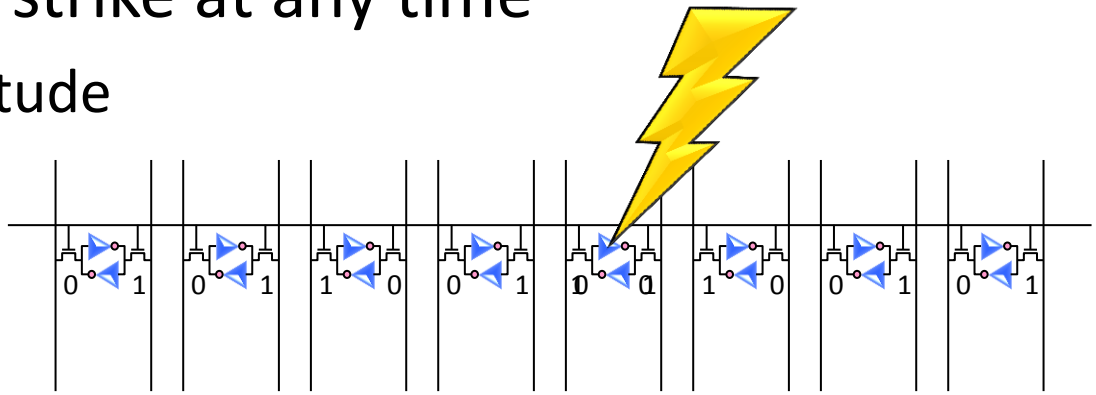
- Choices of Write Policies
  - On write hits, update memory?
    - Yes: *write-through* (higher bandwidth)
    - No: *write-back* (uses *Dirty* bits to identify blocks to write back)
  - On write misses, allocate a cache block frame?
    - Yes: *write-allocate*
    - No: *no-write-allocate*

Inclusion

- Core often accesses blocks not present on chip
  - 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 (who not "exclusive"?)
    - Must write back clean lines
- Some processors combine both
  - L3 is inclusive of L1 and L2
  - L2 is non-inclusive of L1 (like a large victim cache)

Parity & ECC

- Cosmic radiation can strike at any time
  - Especially at high altitude
  - Or during solar flares



- What can be done?
  - *Parity*
    - 1 bit to indicate if sum is odd/even (detects single-bit errors)
  - *Error Correcting Codes (ECC)*
    - 8 bit code per 64-bit word
    - Generally *SECDED (Single-Error-Correct, Double-Error-Detect)*
- Detecting errors on clean cache lines is harmless
  - Pretend it's a cache miss

Homework #1 (of 1) part 3

- Extend your program to…
  - Measure and report the block size of each cache
  - Measure the number of sets in each cache
  - Measure the associativity of each cache
  - Compute the total capacity of each cache

- Output should resemble this:

```
123..890 Loop: 2500000000ns
Cache Latencies: 2ns 6ns 20ns 150ns
Caches: 64/256x4(64KB) 64/512x8(256KB) 128/8192x24(12MB)
```