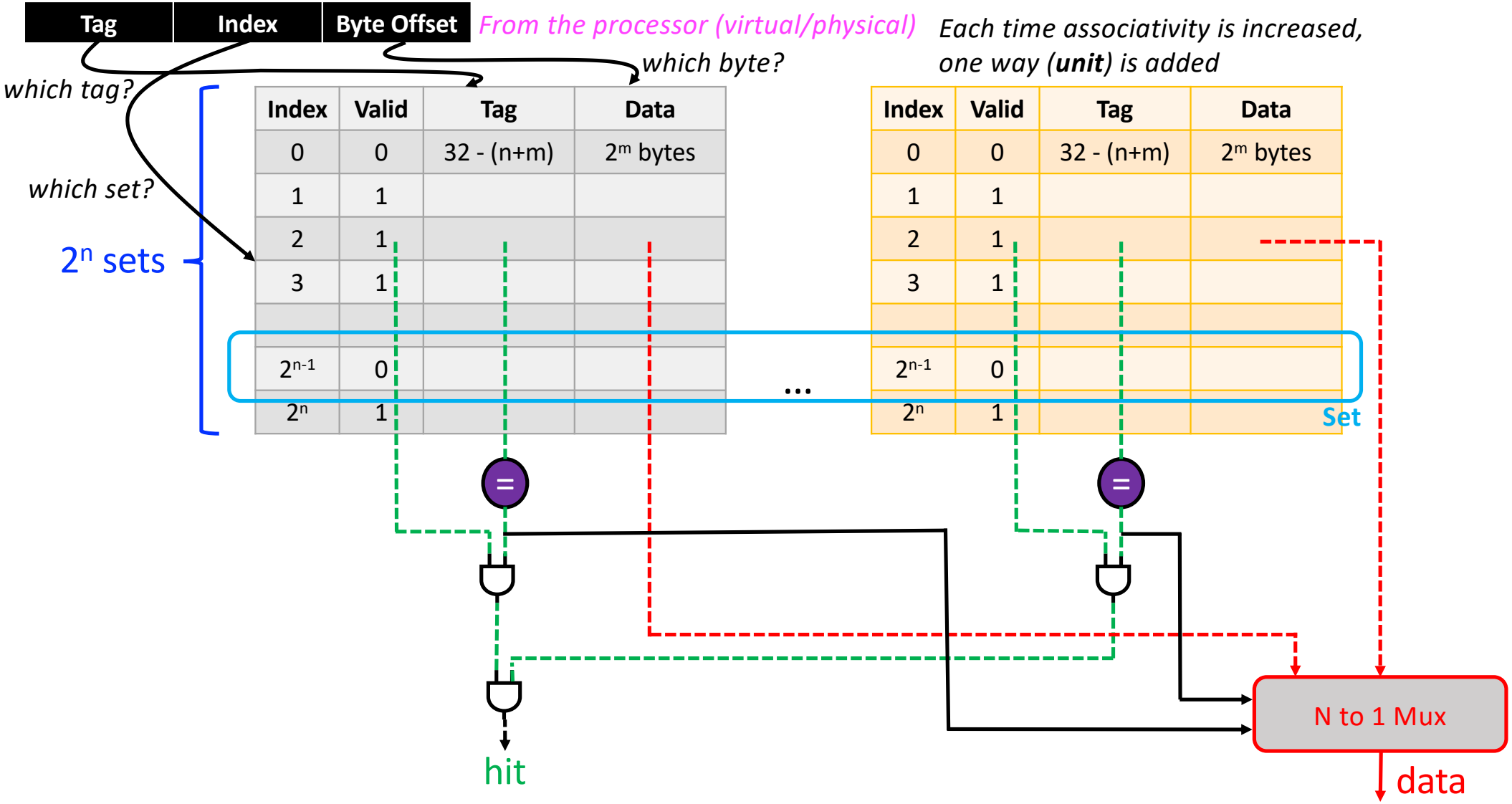
COMP3710 (Class # 5176)
Special Topics in Computer Science
Computer Microarchitecture

Convener: Shoaib Akram
shoaib.akram@anu.edu.au

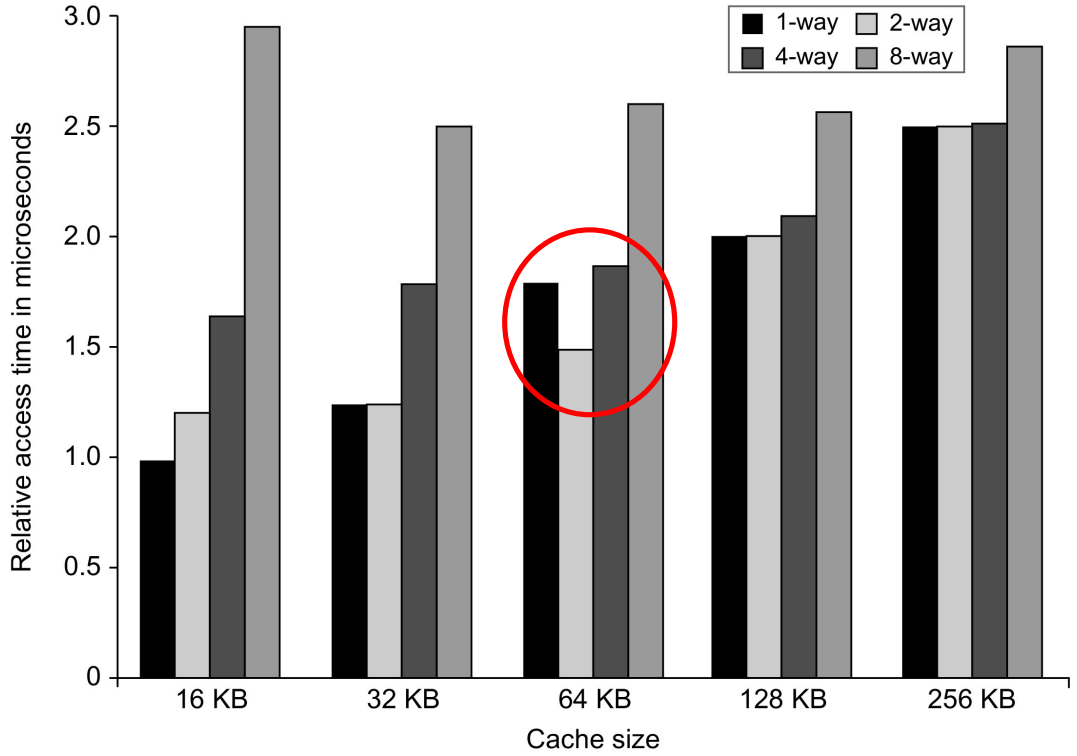
Australian
National
University

Revision: Set-Associative Cache



Access Time vs Cache Size

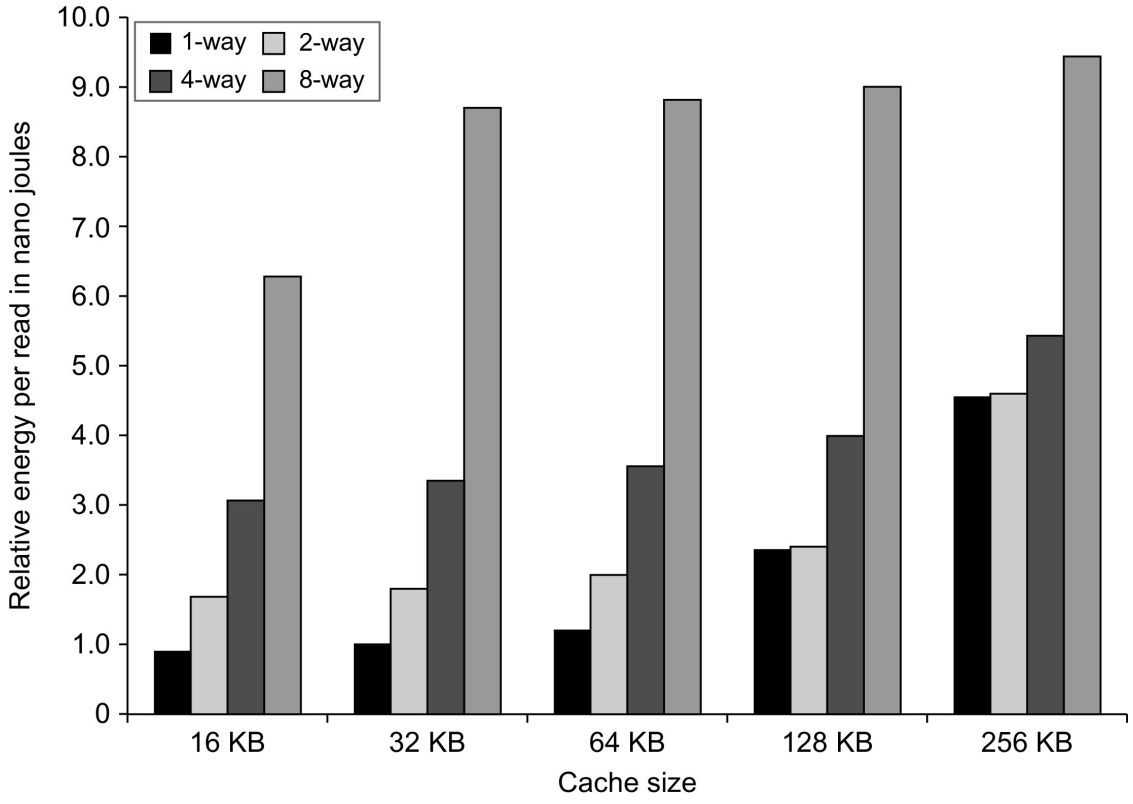
Single bank, 64 bytes, embedded SRAM
Associativity increases access time



CACTI is a tool for modeling caches
https://github.com/HewlettPackard/cacti

Read Energy

Single bank, 64 bytes, embedded SRAM
Associativity increases read energy



Revision: Replacement Policy

- **Direct-mapped cache**
  - New block (A) arrives at location X in the cache
  - Previously residing block at location X is B
  - *Replacement rule is simple: replace B with A*
  - *A and B both cannot reside at location X (one has to make space for the other)*

Revision: Replacement Policy

- **2-way set associative**
  - New block (A) arrives at Set X
  - B and C reside at Set X (2 way set-associative, so two blocks in one set)
  - Replace either B or C with A (which one?)
  - One possiblity is choose randomly
    - We need a more sophisticated rule than *random*
  - ***Least Recently Used (LRU):*** *Replace the least recently used block with A*
  - **References: B, C, C, C, B, B, A** → Replace C with A
    - C is the least recently used, i.e., oldest access time (B has better temporal locality)

Revision: Replacement Policy

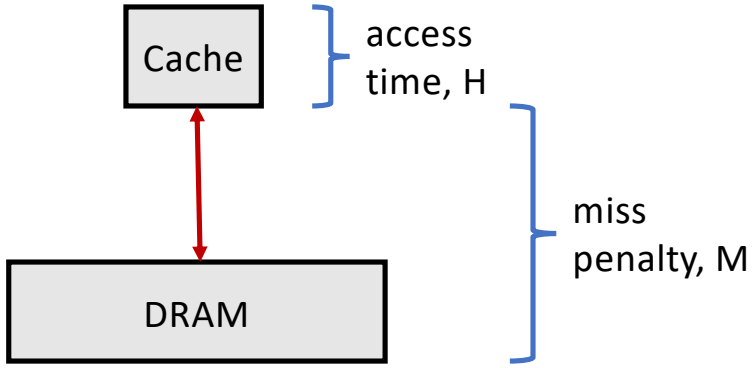
- **Implementing LRU (2-way)**
  - Hit in way # 1 (set a per-set bit to 0)
  - Hit in way # 2 (set a per-set bit to 1)
  - Hit in way # 2 (set a per-set bit to 1)
  - Miss in the set (The bit tells us block in way # 1 is the least recently used)

- **Try to implement LRU with 4-way**
  - Difficult to track age
  - Pseudo-LRU and Bit-LRU are approximations used in practice
    - https://en.wikipedia.org/wiki/Pseudo-LRU

Comparing Cache Designs

*Average Memory Access Time (AMAT) = hit-time + (miss-rate * miss-penalty)*
where,

hit-time = cache access time
miss-rate = 1 – hit-rate
miss-penalty = cache access time + next-level access time



AMAT = H * hit_rate + (H + M) * (1– hit_rate)
AMAT = H * hit_rate + H – H* hit_rate + M – M * hit_rate
AMAT = ~~H * hit_rate~~ + H ~~– H* hit_rate~~ + M – M * hit_rate
AMAT = H + M * (1 – hit_rate)
AMAT = H + M * (miss_rate)

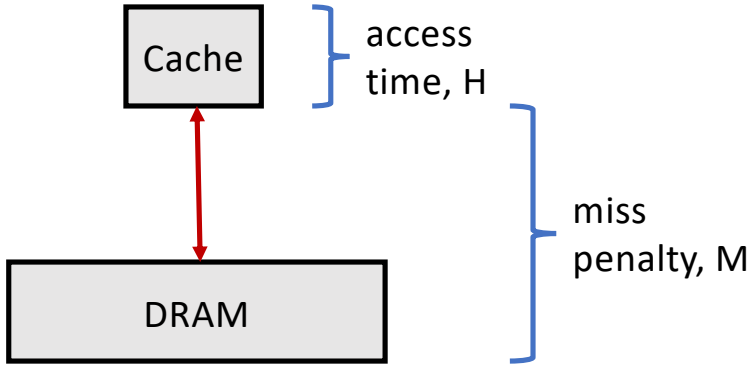
Comparing Cache Designs

*Average Memory Access Time (AMAT) = hit-time + (miss-rate * miss-penalty)*
where,

hit-time = cache access time
miss-rate = 1 – hit-rate
miss-penalty = cache access time + next-level access time

The ultimate goal of cache design is to minimize AMAT

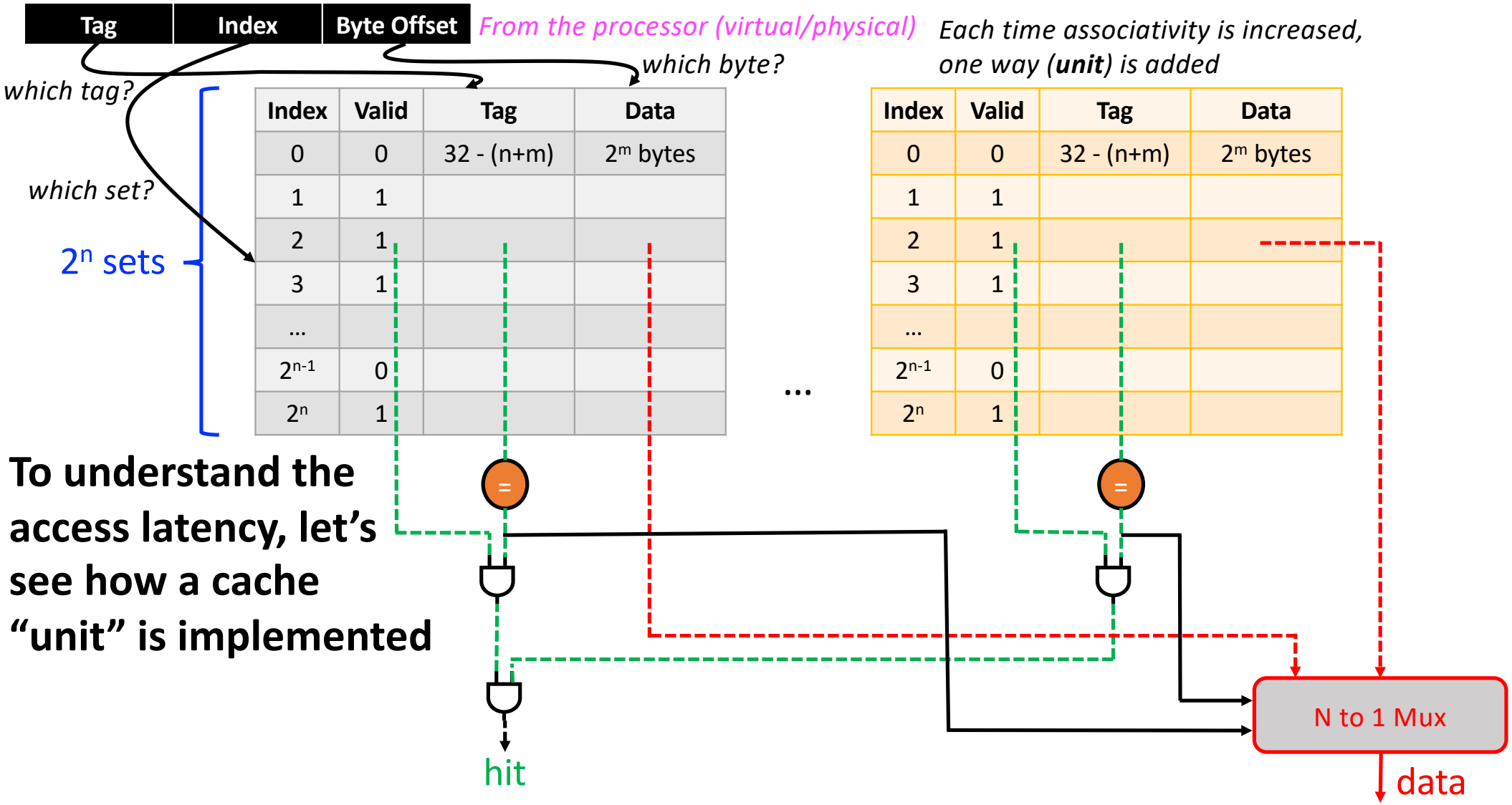
- Maximize hit rate (minimize miss-rate)
  - E.g., via associativity (reduce conflict misses)
- Minimize hit-time
  - Associativity increases hit-time
  - Complex trade-off
- Minimize miss penalty
  - Multi-level caches



Cache Optimization

- Let's look at an optimization that reduces the hit-time while aiming for a low miss rate
  - *Warning: Build up is long and is intentional*
  - *Need to understand how caches are implemented at a low level*

Set-Associative Cache

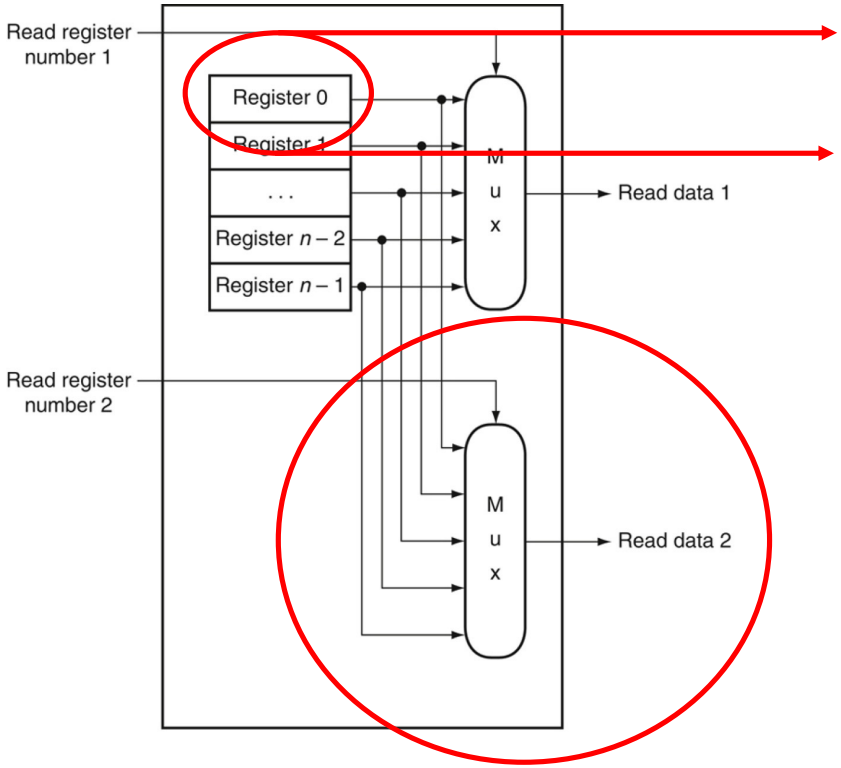


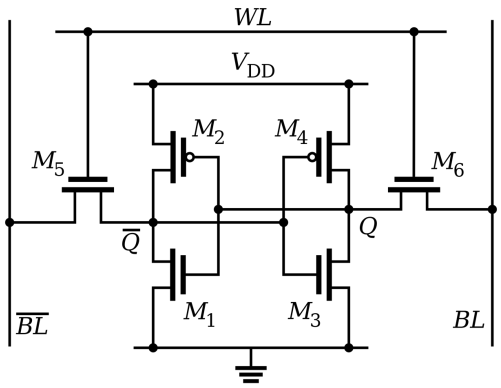
*Each time associativity is increased, one way (**unit**) is added*

To understand the access latency, let's see how a cache "unit" is implemented

SRAM Structure & Organization

Let's revisit how the register file is implemented





SRAM cell at the CMOS level

- 4T/6T/8T (transistors)
- Latch of flip-flop (clocked)
- Details not in scope

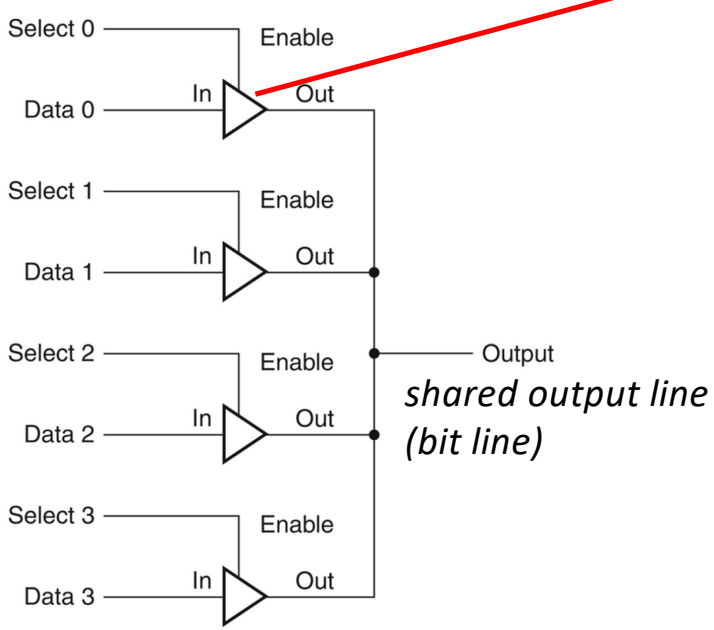
Important

- Word Line: Select/Enable
- Bit Line: Read/Write stored bit

- Mux-based read ports work for register files
- They do not work for caches
  - If we have 4 M sets, we need a 4M to 1 mux for each bit of the cache line
  - Simply too big!

SRAM Structure & Organization

Alternative to multiplexers



Tristate buffer

- Two inputs: Data and Enable
- Output: 0, 1, or Z (high impedence)
- If Enable is 1: Out = Data
- If Enable is 0: Out = Z
- When Out is Z, a different tristate buffer can write to the shared bit line

Four tristate buffers form a multiplexer

- Only one of the four Select lines can be asserted

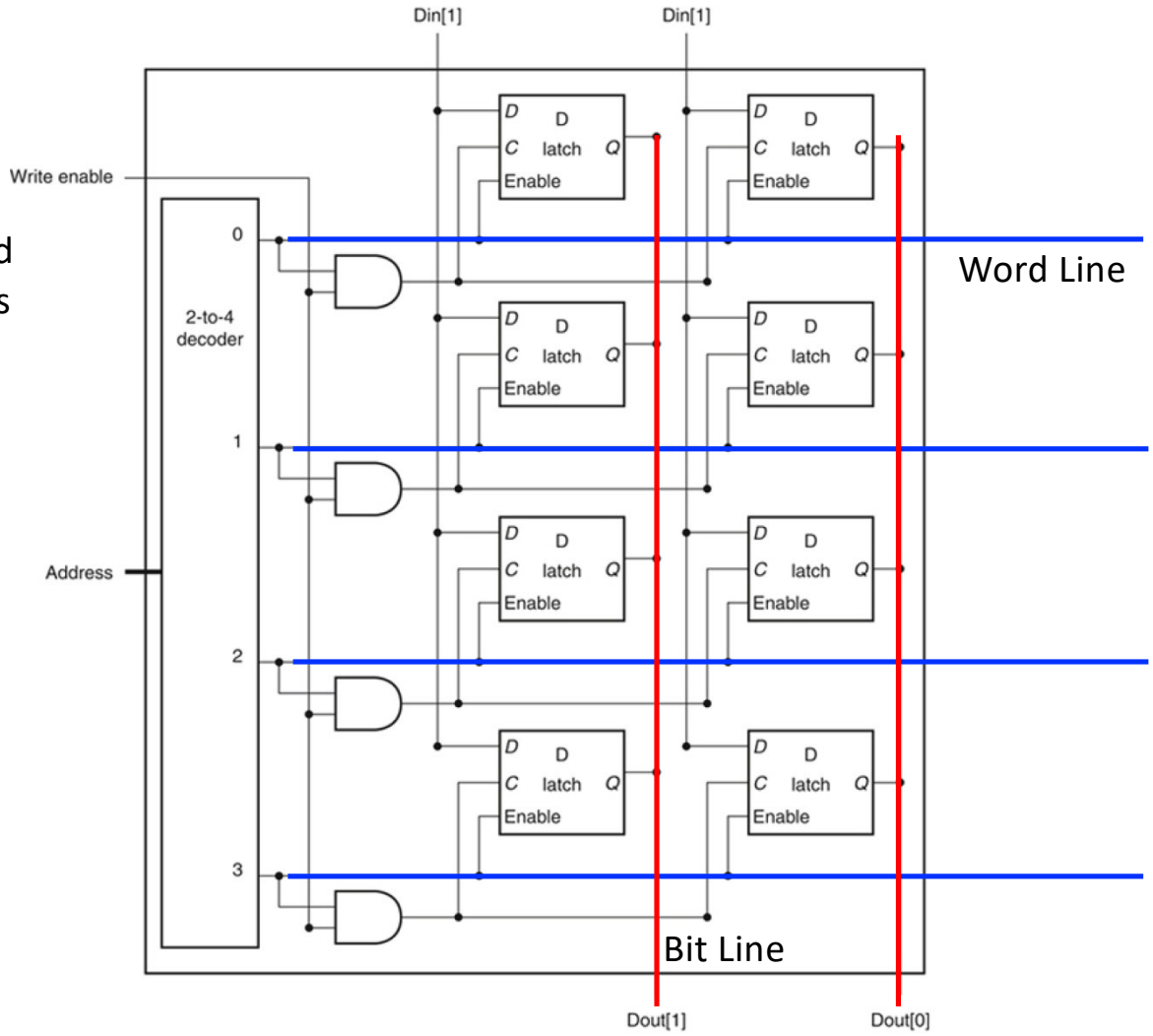
SRAM Structure & Organization

Operation of an SRAM array

- Decoder asserts the correct word line
- Each SRAM cell attached to the word line outputs the stored value
- Tristate buffers (**not shown**) ensure correct operation

Latency of operation

- $T_{decode} + T_{array\text{-}read}$

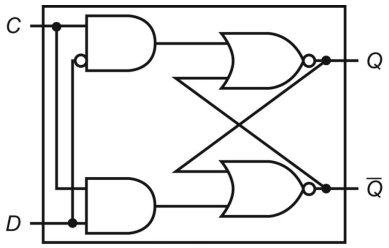


SRAM Structure & Organization

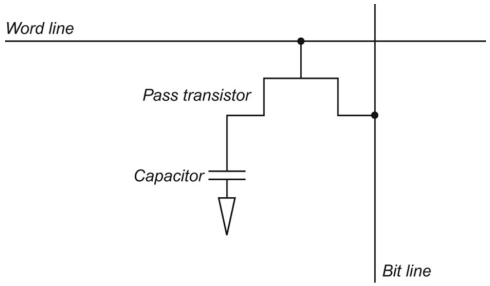
SRAM features

- S stands for static, +ve feedback via a cross-coupled inverter arrangement
- Power consumption is high (need constant voltage supply for the inverters)
- Density is low
- Very fast
- Dynamic RAM: In contrast, stores charge in a capacitor that requires refreshing (high density, slow, low power)

- What about the size of the decoder?
  - Still very big (impractical for large arrays)
  - Two-level decoding provides some relief
    - Will return to this in week 11 (DRAM)
  - Another alternative is multi-banking (in a few slides)

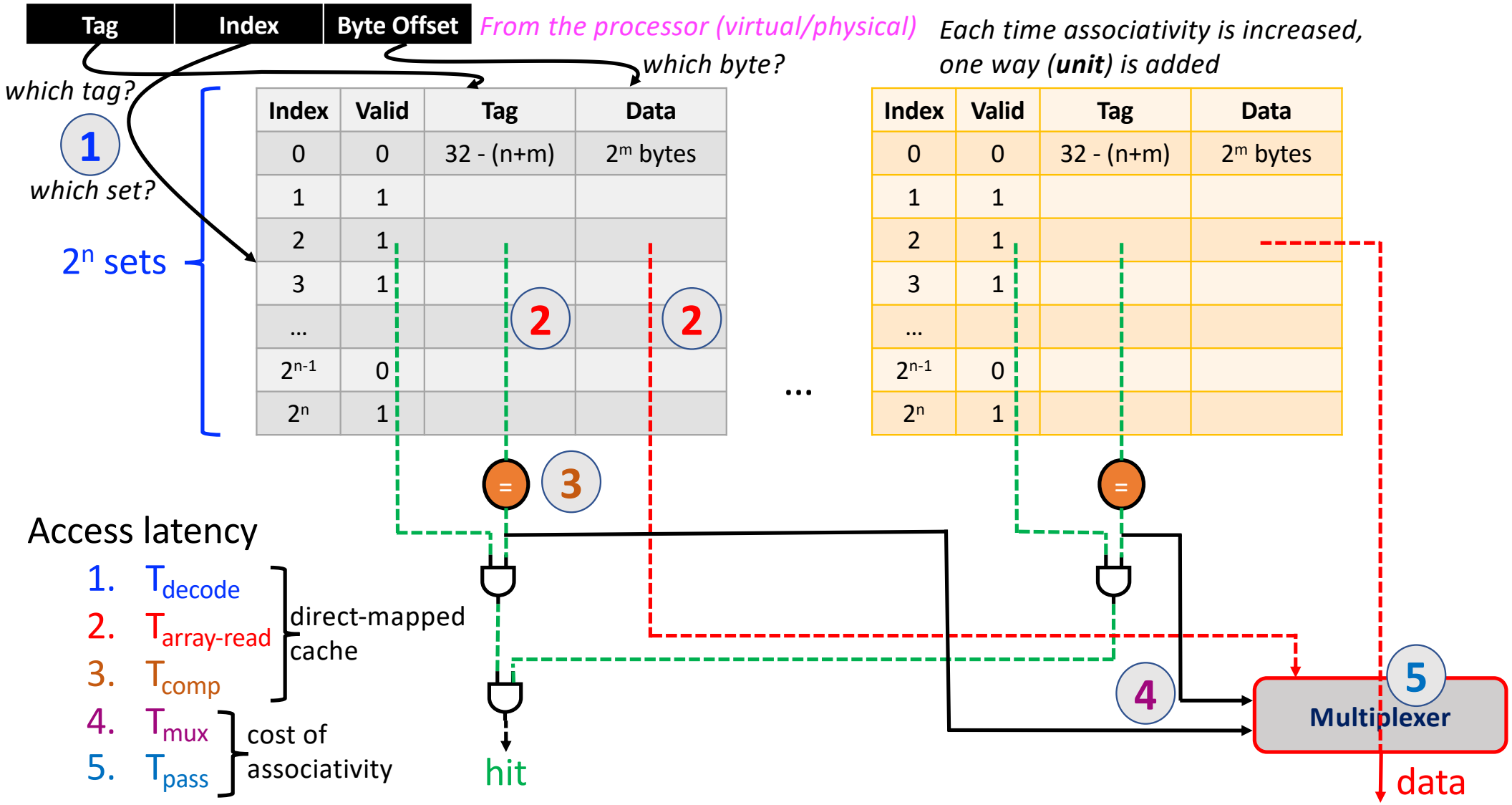


SRAM Cell



DRAM Cell

Set-Associative Cache



Access latency

1. T_{decode}
2. $T_{array\text{-}read}$
3. T_{comp}

direct-mapped cache

4. T_{mux}
5. T_{pass}

cost of associativity

Problem

We want the latency of a direct-mapped cache

1. T_{decode}
2. $T_{array\text{-}read}$
3. T_{comp}
4. T_{mux} ✗
5. T_{pass} ✗

We also want the flexibility of a set-associative cache

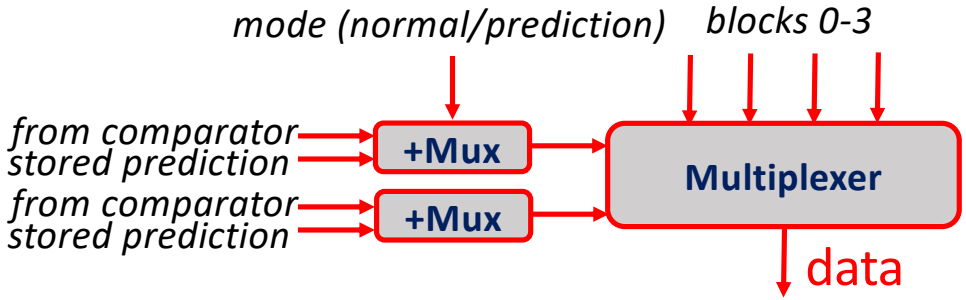
- Having many ways reduces the miss rate (less collisions)

What can we do?

- What is the problem?

Way Prediction

- **Predict the way that is likely to result in a tag match**
  - Keep "prediction" bits in each set
  - When the indexing operation is complete, read the prediction bits in addition to tag/data
  - Send the prediction bits to the multiplexer
  - Overlap the latency of "tag comparison" with
    - Warming up the multiplexer
    - Pass through of the chosen block of data
- **If predictiction is wrong**
  - There is a tag mismatch
  - Fall back to the normal mode
  - Check everything
- **If misprediction rate is low**
  - Hit time of a direct-mapped cache
  - Flexibility of a set-associative cache



Way Selection

- **We can reduce the power/energy of a set-associative cache that uses way prediction**
  - Only read the data array of the predicted way
  - Use gating to disable all other arrays reads
    - Enable signal of unselected arrays is AND-gated
  - Saves power when misprediction rate is low

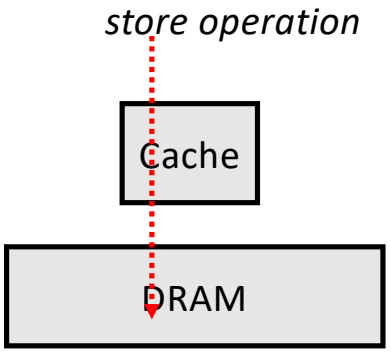
Some Quantitative Data

- First used in MIPS R10K (1990s)
- Used in several ARM processors
- 80% - 90% prediction accuracy reported in literature
- Way selection is a latency-energy tradeoff
  - I-cache access time increases by 4% (4-way)
  - D-cache by 13%
  - 72% reduction in I-cache power
  - 75% reduction in D-cache power

Handling Writes: Write-Through

Write-through Cache

- On a write, update the cache and the next level of the memory hierarchy
- Data is always consistent across all levels
- Advantage: Memory hierarchy is always in a consistent state
  - Simplifies coherence in multicore processors
  - A different processor gets the shared data from memory
- Disadvantage: Generates a lot of memory traffic
- Disadvantage: Write latency is very high



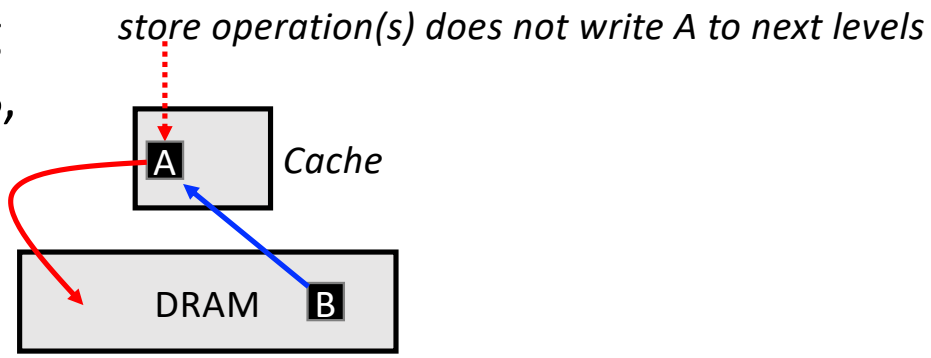
Exercise: Spending 100 extra cycles on 10% of the instructions that are stores would increase the base CPI from 1 to 11.

- *1 + 0.1*100 = 1*

Handling Writes: Writeback

- **Writeback Cache**
  - On a write, update the block in the cache only (if present)
  - When the modified block is replaced, write it to the next level
  - Data in cache and memory maybe inconsistent
  - Advantage: Memory traffic is reduced
  - Disadvantage: Coherence is tedious to implement

Block **A** is being replaced with **B**, only now move modified block **A** to the next level



Write-through vs Writeback

- **Write-through**
  - Uniform latency on misses
  - Requires extra bus bandwidth
- **Writeback**
  - Requires extra “dirty” bits in cache
  - Stresses the bus bandwidth less
  - Non-uniform miss latency
  - Clean miss: one transaction with lower level: fill
  - Dirty miss: two transactions with lower level: fill, writeback

Handling Write Misses

- **Handling read miss of block A is simple**
  - Index the cache
  - Find the replacement candidate (say block B)
  - Bring the block A from the next level (or the level after that)
  - Replace the exisiting block B with the new block A
  - If B is modified (dirty) then write B to the next level

- **A question emerges for handling write misses**
  - To allocate or not to allocate a block in the cache on a write miss?
  - **Write-allocate:** Fetch the block from memory, allocate an entry for the block in the cache and then write the modified portion of the block
    - For many scenarios, this approach is inefficient
    - Operating systems zeroes large regions of memory to offer memory safety
    - Garbage collectors copy entire blocks of memory (objects) from one location of memory to another (defragmentation)

Handling Write Misses

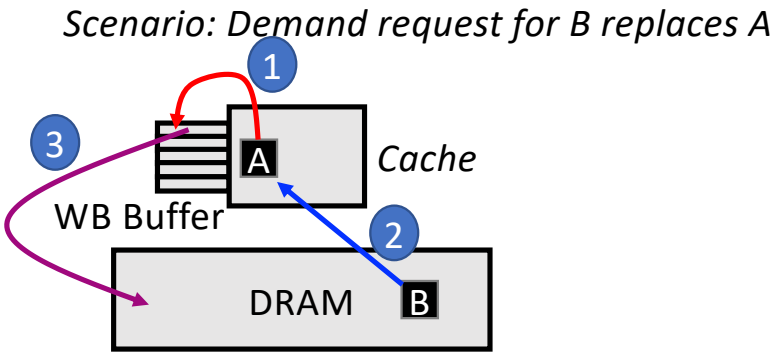
- **Write-allocate:** *Fetch the block from memory, allocate an entry for the block in the cache and then write the modified portion of the block*
  - For many scenarios, this approach is inefficient
    - Operating systems zeroes large regions of memory to offer memory safety
    - Garbage collectors copy entire blocks of memory (objects) from one location of memory to another (defragmentation)
    - These are streaming accesses likely to thrash the cache (no locality advantage)

- **No-write-allocate:** *Update the portion of the block in memory but not in the cache*
  - The fetch associate with the initial write miss is unnecessary
  - Some processors allow changing the allocation policy dynamically at run-time
  - Also called ***non-temporal stores***, Intel x86 ISA has special instructions to do such stores

Writeback Buffer

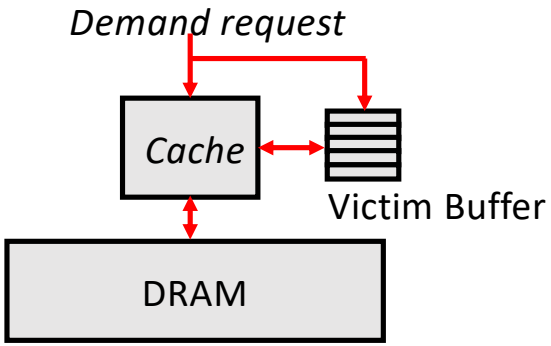
- **Consider a demand miss in a writeback cache**
  - Demand miss: request from the processor for a word in block A
  - On a demand miss the writeback cache needs to
    - Write the replacement candidate to the next level of memory
    - Bring the demanded block from memory and install it in the cache
    - Which one should happen first? Only one answer: Cannot overwrite modified block
  - Writeback buffer: Move the evicted block to the writeback buffer and handle the demand miss first (an optimization to reduce the miss penalty)

Block **A** is written back to memory asynchronously and demand miss is handled first



Victim Cache

- A small (16-entry) fully-associative buffer
  - Whenever a block is replaced (modified or otherwise) store it in the victim cache
  - On a cache miss, check the victim cache and refill from there instead of brining from the next level
  - On a cache hit, victim cache is untouched
  - Poor man's associativity
  - Useful if the cache is direct-mapped
  - These days, multiple levels of the cache hierarchy provide the same functionality
    - Direct-mapped caches not in use anymore
  - Writeback buffer is a special case of victim buffer



Store Buffer: Motivation

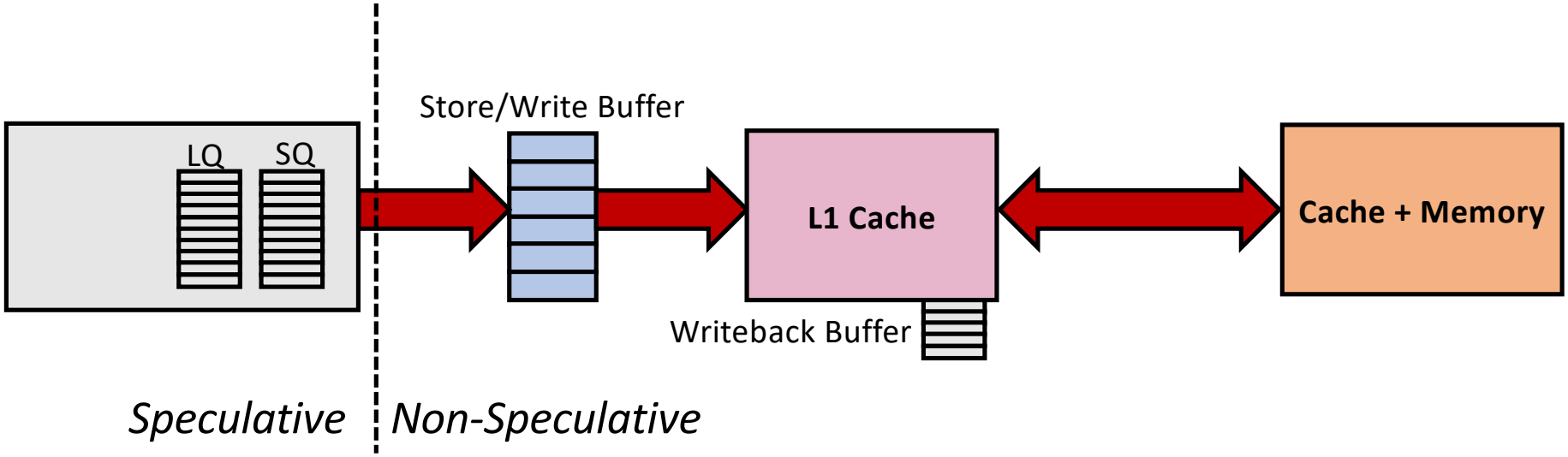
In a write-through Cache

- Each store instruction takes 100+ cycles to complete
- Blocks the OOO processor (ROB head and SQ head both blocked)
- Retire the store from the ROB and SQ head and place it in a store (write) buffer

In a writeback Cache

- Store instruction still takes more than one cycle
- Tag check + Copy victim into the writeback buffer
- What if writeback buffer is full?
- What if the cache is busy serving another request?
- Same idea: Retire the store from the ROB and SQ head and place it in a store (write) buffer

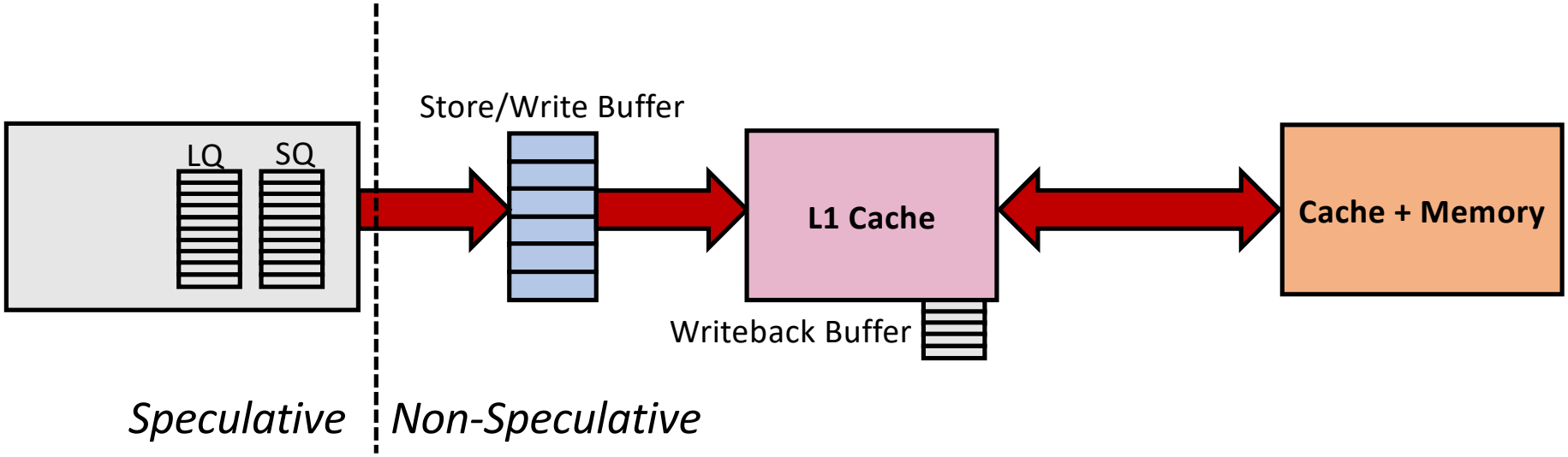
Store Buffer



Store instructions are retired from the head of ROB to the store buffer

- Stores are architecturally complete once in the store buffer (cannot be undone)
- Visible to another processor core (other cores snoop the store/writeback buffer)
- SQ is not visible to the outside world (possibly wrong-path store instruction)
- Store/Writeback buffers can do write merging
  - Intel Core i7 does write merging

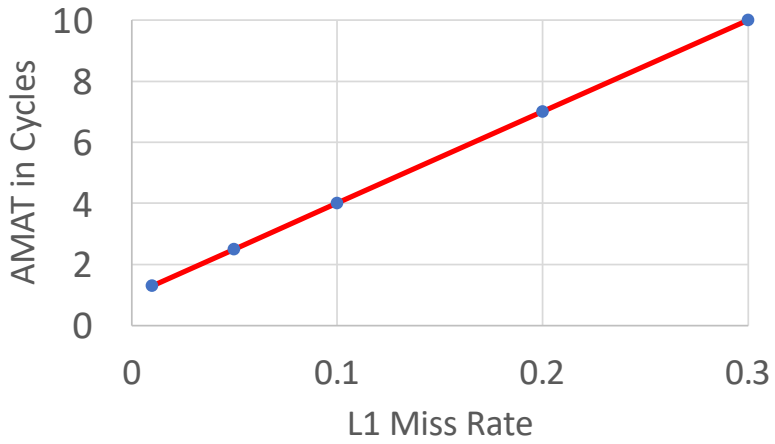
Store Buffer-Related Stalls



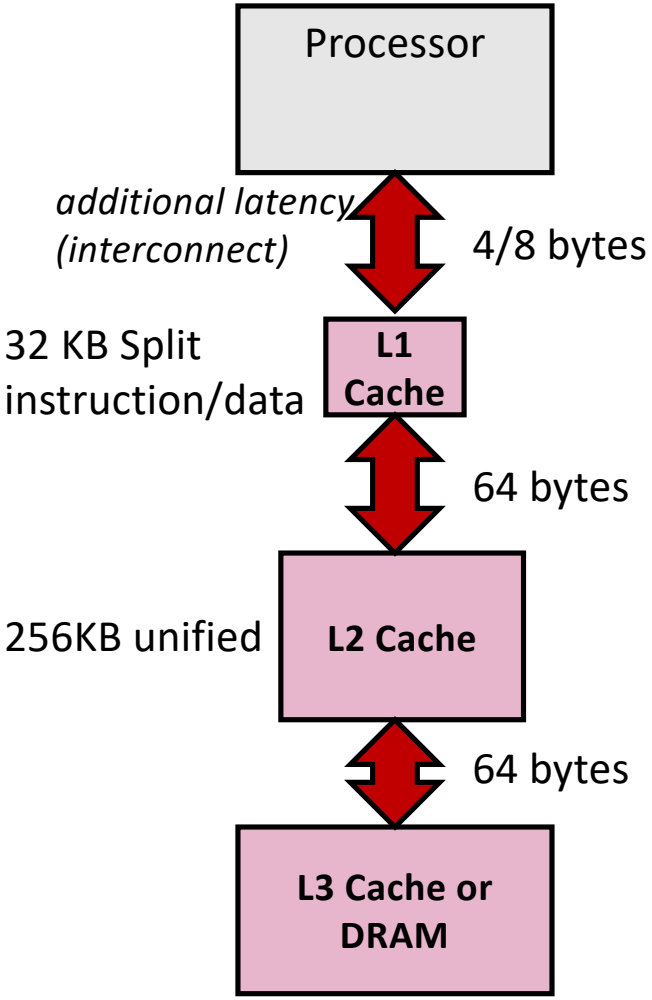
- If the store buffer is full, new stores cannot retire
- Store bursts observed more frequently in Java & other managed applications
  - Special kind of memory management (garbage collection)
  - Frequent copying and guaranteed zeroing of program heap

AMAT: Multi-Level Caches

$AMAT = H_{L1} + (miss\_rate_{L1} * M_{L1})$
$AMAT = H_{L1} + (miss\_rate_{L1} * (H_{L2} + (miss\_rate_{L2} * M_{L2}))$

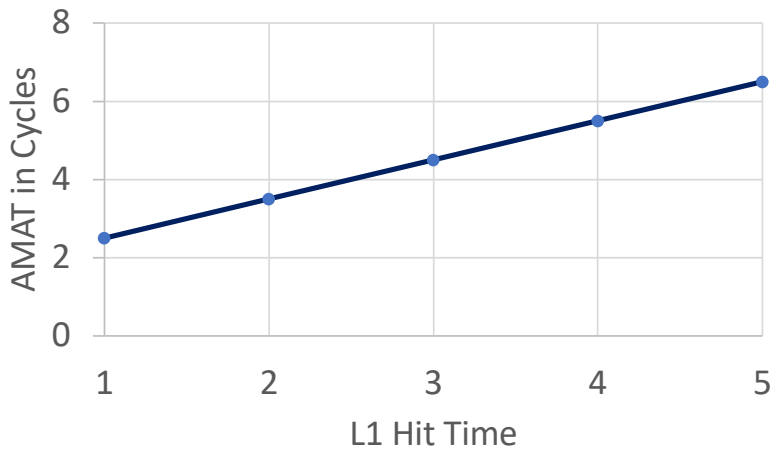


H_{L1} = 1 cycle
H_{L2} = 10 cycles
M_{L2} = 100 cycles
$miss\_rate_{L2}$ = 20%

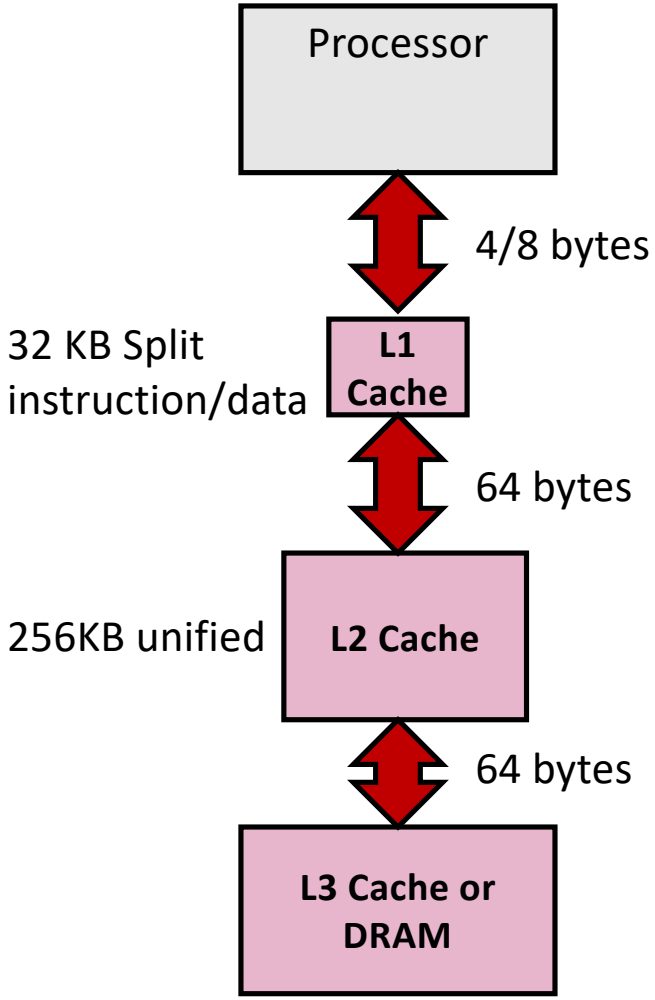


AMAT: Multi-Level Caches

$AMAT = H_{L1} + (miss\_rate_{L1} * M_{L1})$
$AMAT = H_{L1} + (miss\_rate_{L1} * (H_{L2} + (miss\_rate_{L2} * M_{L2}))$

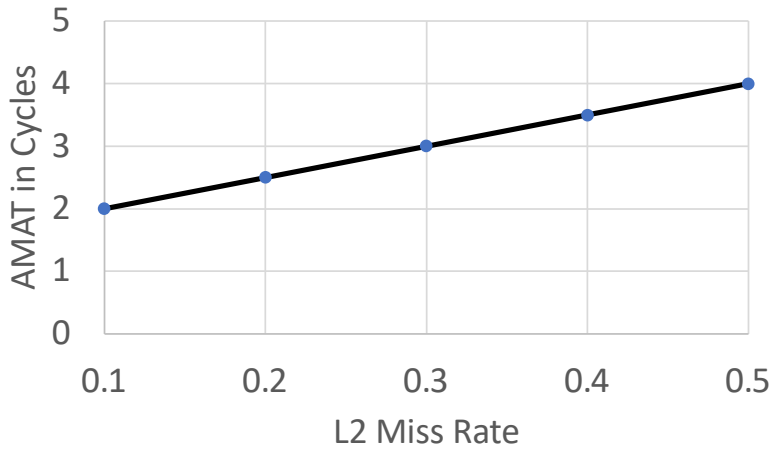


$miss\_rate_{L1} = 0.05$
H_{L2} = 10 cycles
M_{L2} = 100 cycles
$miss\_rate_{L2}$ = 20%

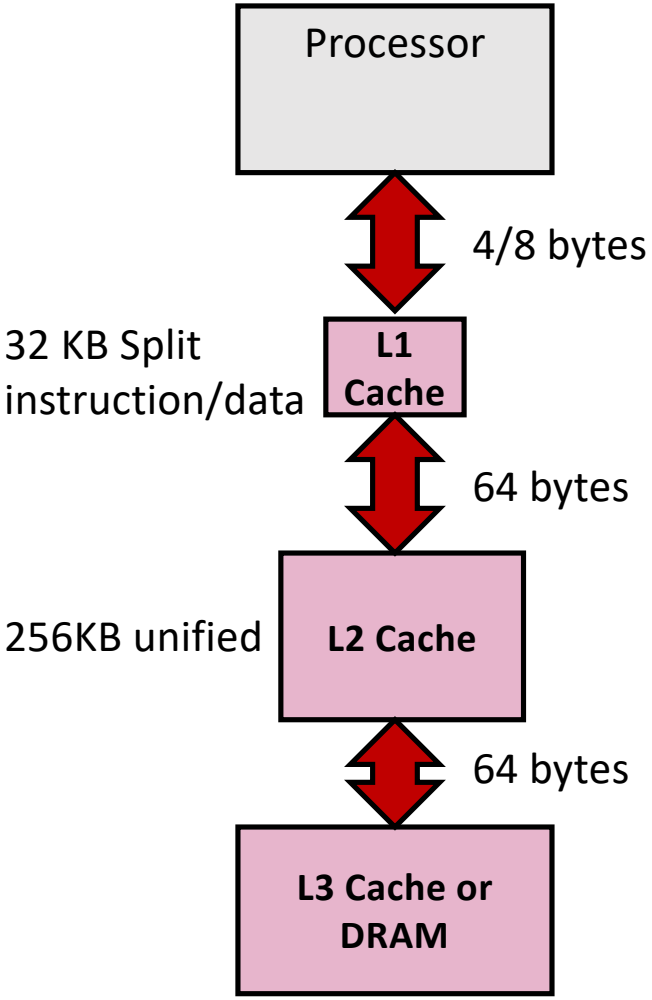


AMAT: Multi-Level Caches

$AMAT = H_{L1} + (miss\_rate_{L1} * M_{L1})$
$AMAT = H_{L1} + (miss\_rate_{L1} * (H_{L2} + (miss\_rate_{L2} * M_{L2}))$



H_{L1} = 1 cycle
$miss\_rate_{L1}$ = 0.05
H_{L2} = 10 cycles
M_{L2} = 100 cycles



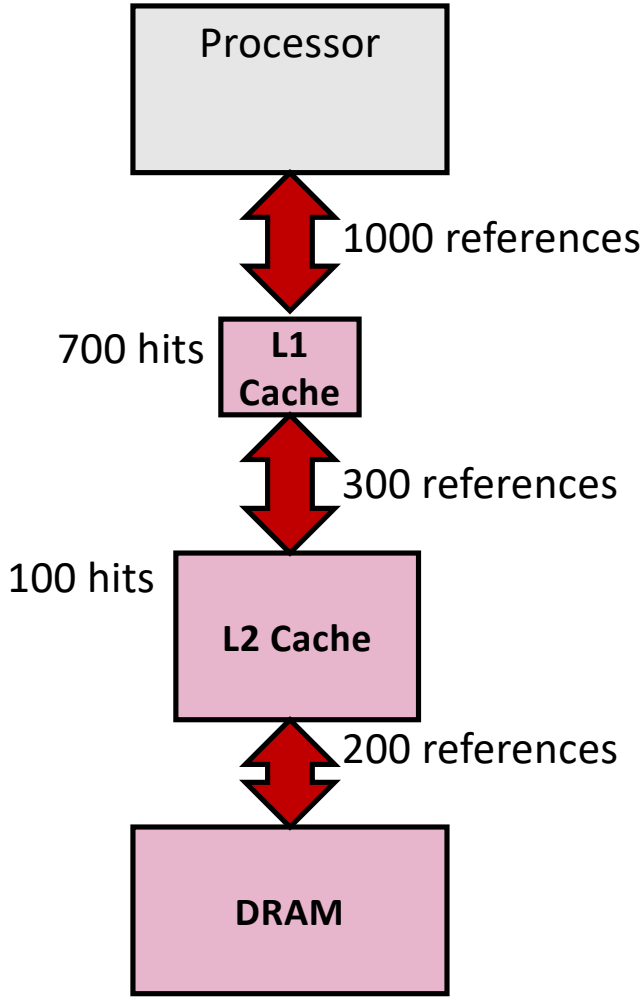
Local vs Global Miss Rate

Local miss rate, L1 = 300/1000 = 0.3

Local miss rate, L2 = 200/300 = 0.66

Global miss rate, L2 = 200/1000 = 0.2

Global miss rate, L2 = L1 miss rate * L2 Local miss rate



L1 Cache Design Tradeoffs

Holistic view is important

- Cannot design L1 (or any level) in isolation of the rest of the hierarchy

Scenario # 1: L2 miss rate is very low

L2 hit cycles = 10
L2 miss cycles = 100

L2 Miss Rate = 2%				
L1 Miss Rate	5%	30%	5%	30%
L1 Hit Cycles	1	1	5	5
AMAT	1.6	4.6	5.6	8.6

If the choice is to optimize for hit time or hit rate, the designer should focus in this scenario on keeping the hit latency as small as possible

L1 Cache Design Tradeoffs

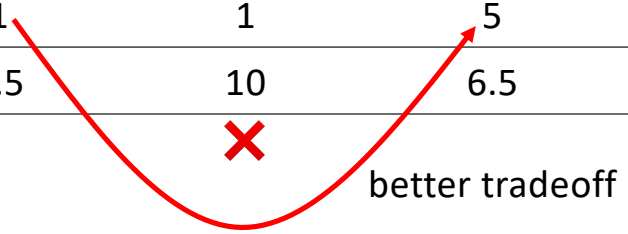
Holistic view is important

- Cannot design L1 (or any level) in isolation of the rest of the hierarchy

Scenario # 1: L2 miss rate is very high

L2 hit cycles = 10
L2 miss cycles = 100

L2 Miss Rate = 20%				
L1 Miss Rate	5%	30%	5%	30%
L1 Hit Cycles	1	1	5	5
AMAT	2.5	10	6.5	14



If the choice is to optimize for hit time or hit rate, the designer should focus in this scenario on keeping the hit rate as high as possible

Multi-level Cache Design Tradeoffs

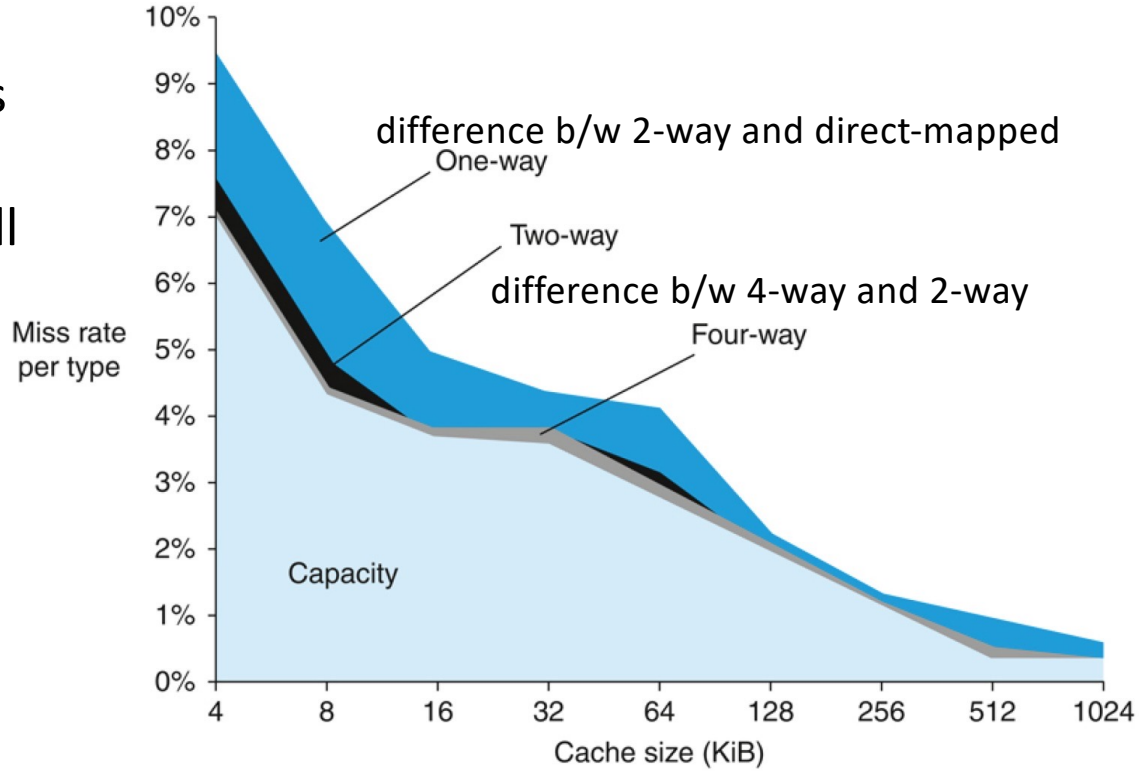
- **L1 Cache**
  - Keep the hit time as low as possible (smaller decoders/rows)
  - Hit rate should still be reasonable otherwise IQ fills up very quickly
  - If we miss, there is L2/L3 (OOO machine can tolerate L2/L3 latency)
- **L2 and L3 cache**
  - Optimize for hit rate
  - Avoid going to memory as much as possible
  - OOO machine cannot tolerate 100+ cycle memory latencies
  - Very large capacity, big decoders, lots of rows
- **Assumptions**
  - When simulating/analyzing caches, remember the tradeoffs
    - Low hit time → Worst hit rate (e.g., direct mapped)
    - Bigger cache = Larger hit/miss penalty

The Three Cs (3C) Model

- **Compulsory misses**
  - First access to a block (cold-start misses)
  - How can we avoid compulsory misses?
- **Capacity misses**
  - Cannot contain all blocks during the execution of the program
  - When blocks are replaced and then retireved later
- **Conflict misses**
  - When multiple blocks compete fro the same set
  - Also called collision misses
  - Eliminated with a fully associative cache of the same size
- **Example**
  - Direct-mapped cache with 2 blocks, main memory with 32 blocks
  - References: 0, 2, 4, 6, 8, 2, 4
    - Homework (do before the next lecture): Is 2 and 4 conflict or capacity?

3C: Quantitative Data

SPEC integer and floating point benchmarks
Compulsory component: 0.006%
Difference b/w 8-way and 4-way is too small



Design Challenges

- **No free lunch**
  - Tradeoff between performance (hit latency) and miss-rate
  - A good design must strike a careful balance
- **Question:** Why does increasing the block size increases the miss penalty?

Design change	Effect on miss rate	Possible negative performance effect
Increases cache size	Decreases capacity misses	May increase access time
Increases associativity	Decreases miss rate due to conflict misses	May increase access time
Increases block size	Decreases miss rate for a wide range of block sizes due to spatial locality	Increases miss penalty. Very large blocks could increase miss rate

Practice

Processor specifications

- L1$ and L2$
- 5 cycle L1$ hit time and 4% L1$ miss rate
- 100 cycle penalty to go to main memory
- 0.5% L2$ global miss rate
- 25 cycles penalty to go to L2$

Question: What is AMAT with and without L2$?

Practice

- **Without L2$**

  AMAT = (5 + 0.04 * 100) = 9 cycles

- **With L2$**

  Global L2$ miss rate = L1 $ miss rate * Local L2$ miss rate
  Local L2$ miss rate = L2$ global miss rate / L1$ miss rate
  = 0.5/4 = 0.125

  $AMAT = H_{L1} + (miss\_rate_{L1} * (H_{L2} + (miss\_rate_{L2} * M_{L2}))$
  = 5 + (0.04 * (25 + .125 * 100) = 6.5 cycles

Practice

A cache has 4096 blocks and a 4-word block size. Assuming 32-bit addresses, find the total number of sets and the total number of tag bits for caches that are (1) direct-mapped (2) 2-way set associative (3) 4-way set-associative (4) fully associative.

bits for index + tag = 32 – 4 = 28 (4-word block means 16 addressable bytes in each block)
sets (blocks) in direct-mapped cache = 4096 = 12-bit index and 16-bit tags
tag bits = 16 * 4096 = 66 K tag bits

Each degree of associativity decreases the # sets by 2 and decreases the # bits to index the cache by 1 and increases the tag bits by 1

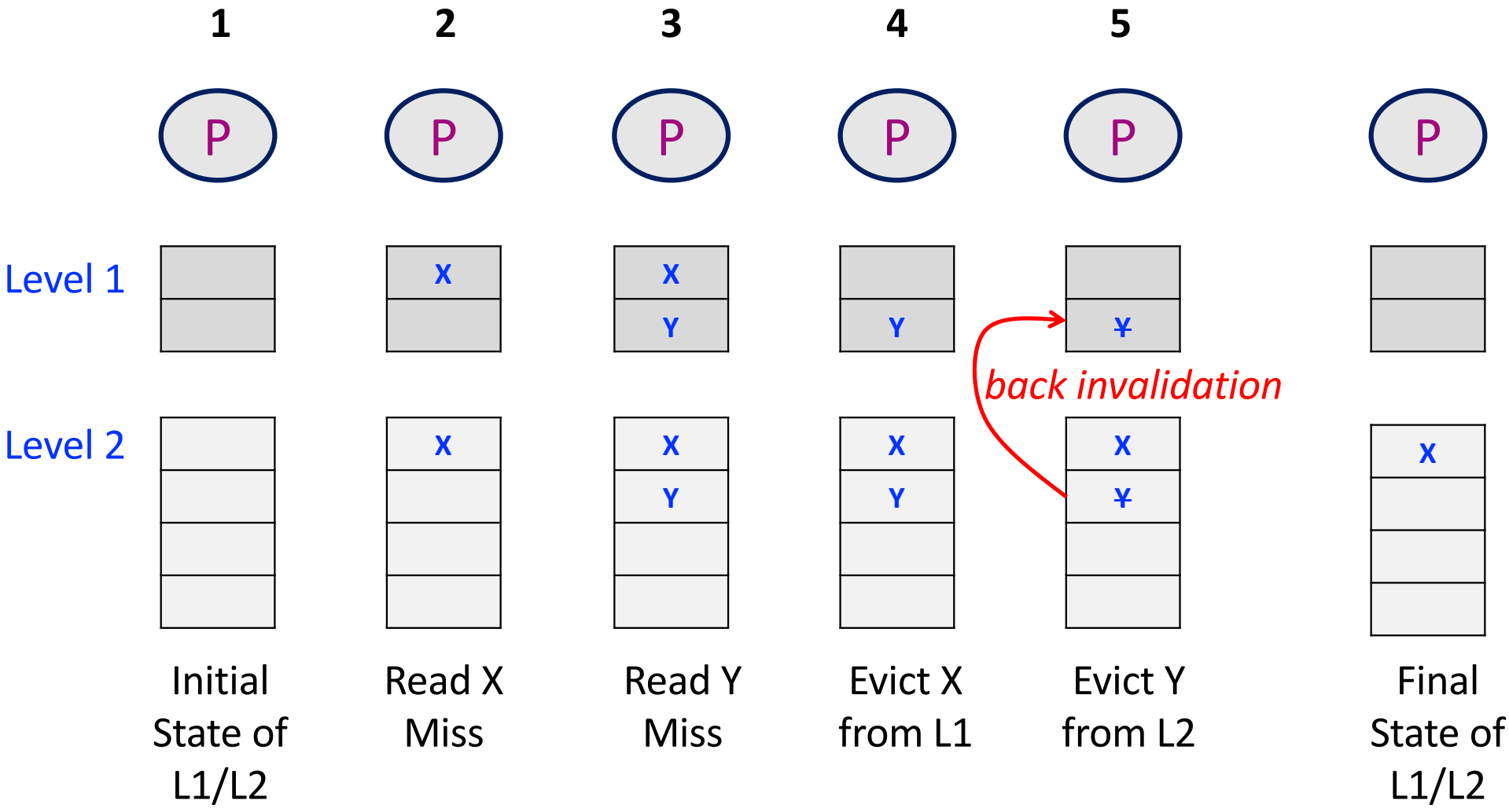
- 2-way cache: 2048 sets, tag bits = (28 – 11) * 2 * 2048 = 70 Kbits
- 4-way cache: 1024 sets, tag bits = (28 – 10) * 4 * 1024 = 74 Kbits
- Fully associative: 28 * 4096 * 1 = 115 Kbits

Multi-Level Inclusion

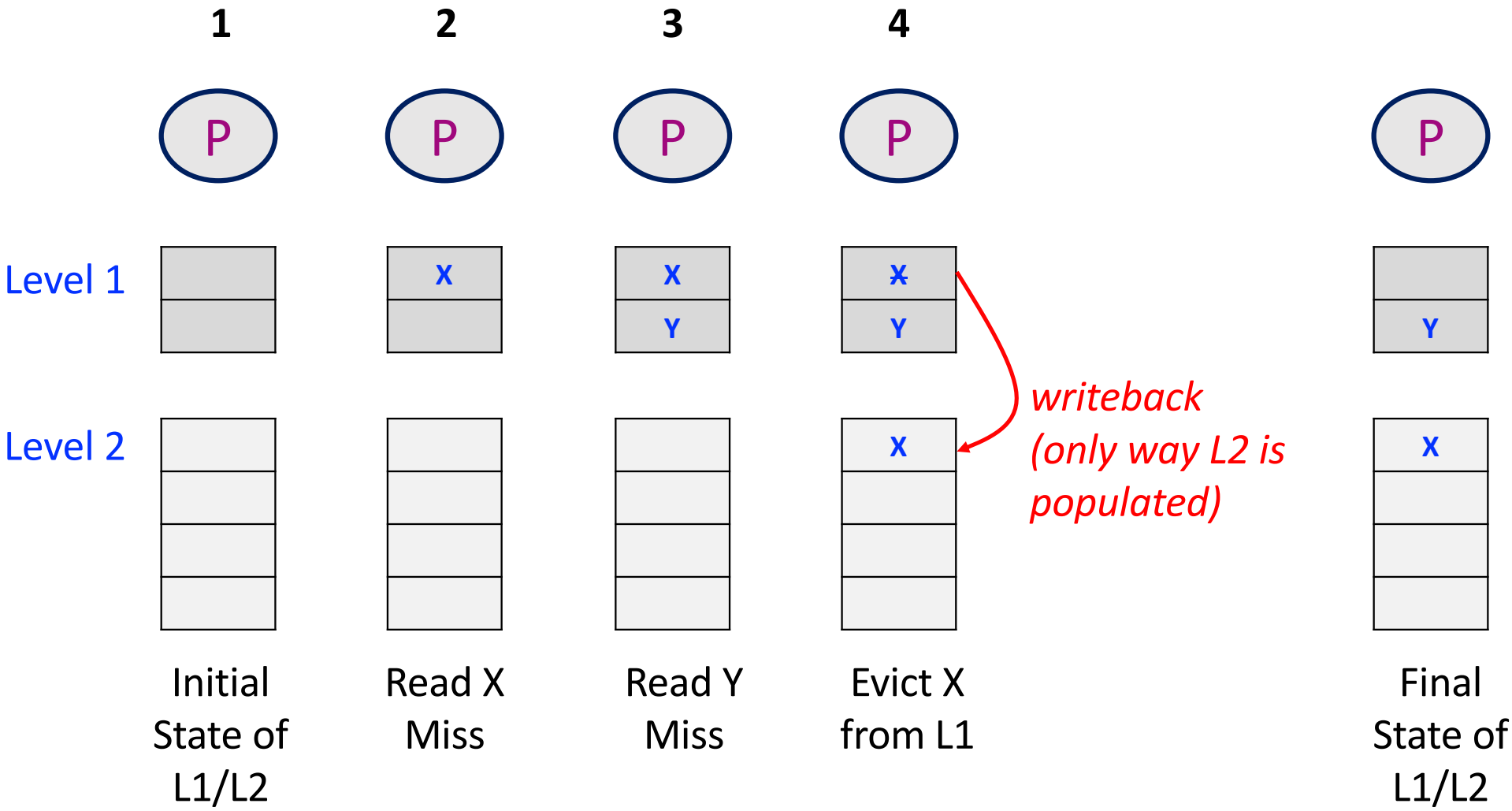
- Multi-level cache hierarchies can be inclusive, exclusive, or neither inclusive nor exclusive
  - **Inclusive:** All blocks in the upper level are also present in the lower level
  - **Exclusive:** A block is present either in the upper level or in the lower level

- **Inclusive pros and cons**
  - Cache coherence is simplified, need not check all levels of the hierarchy if another processor (or core) needs a shared cache line (+)
  - More work each time a block is replaced from the lower levels (-)
  - Redundant cache lines in all levels of the hierarchy wastes space (-)
- **Exclusive pros and cons**
  - Exploit the full capacity of the cache hierarchy (+)
  - Need to check all levels when another processsor (or core) wants a shared line (-)

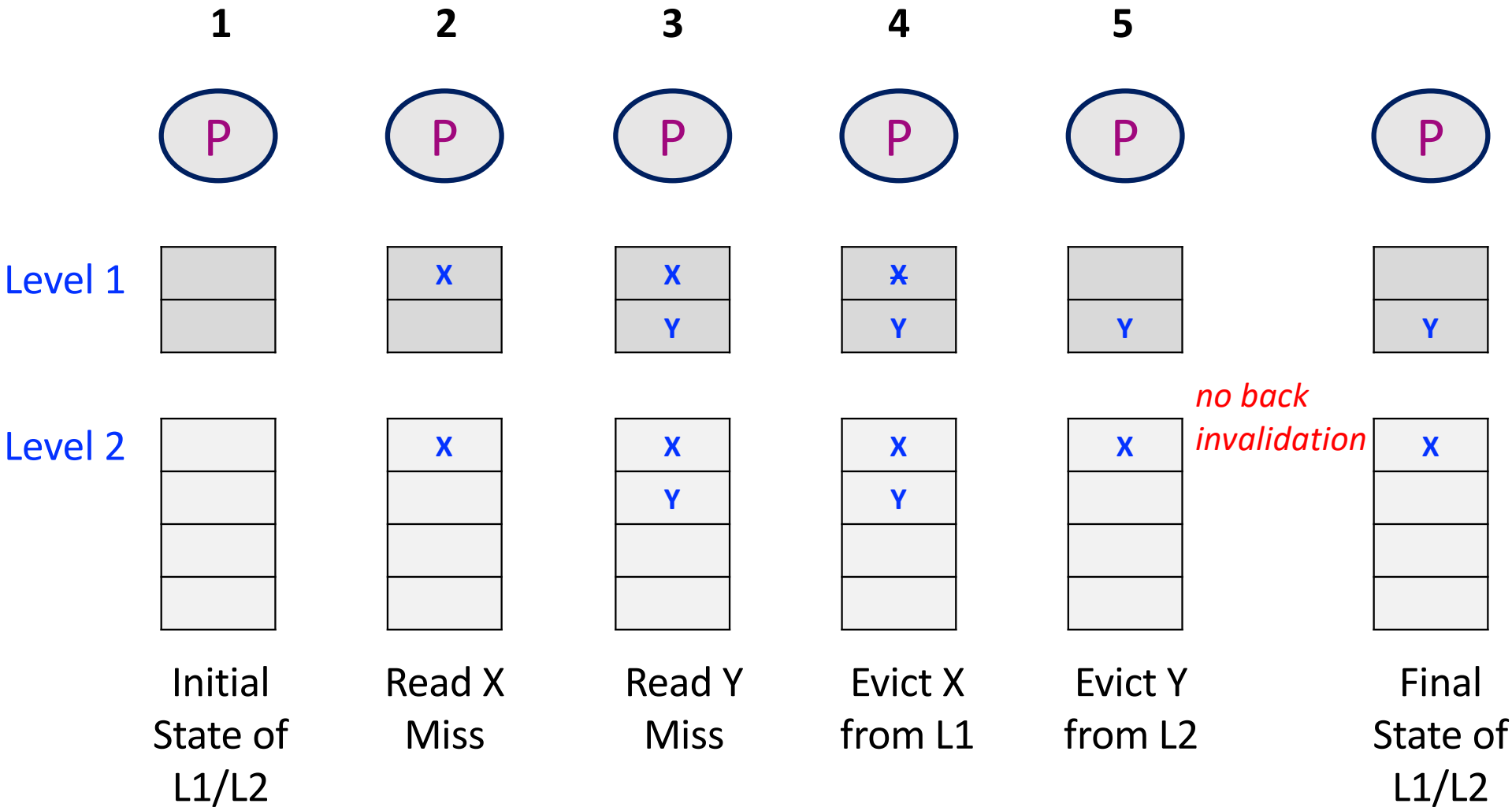
Inclusive Policy



Exclusive Policy



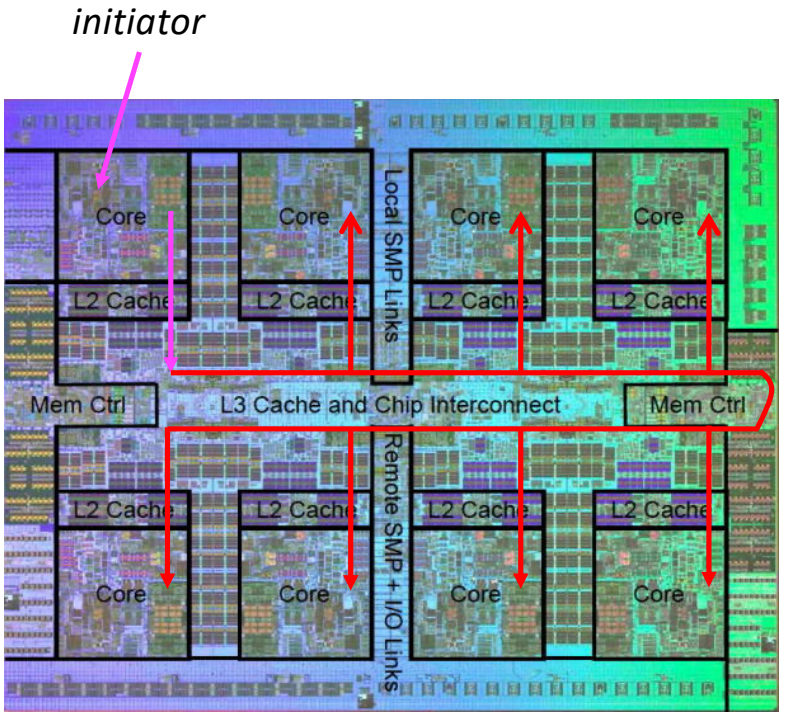
NINE Policy Neither Inclusive Nor Exclusive



Quick Notes

Implementation complexity of snoopy-based cache coherence protocol and multicore interconnects interact with inclusion policy

- *https://en.wikipedia.org/wiki/Cache_inclusion_policy*

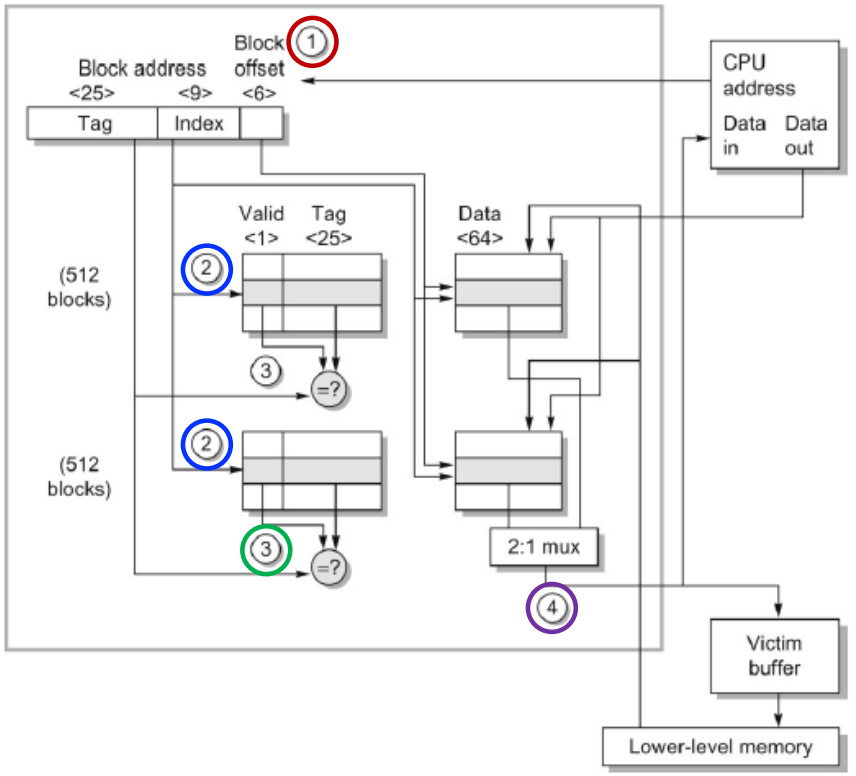


Recent Intel processors (after Nehalem)

- *Tag inclusion*
- *(Historically) Inclusion means value inclusion*
- *https://stackoverflow.com/questions/59450056/can-an-inner-level-of-cache-be-write-back-inside-an-inclusive-outer-level-cache*

AMD Opteron Data Cache

Total data: *64 KB*
Block size: *64 Bytes*
Associativity: *2-way*
Replacement Policy: *LRU*
Write Policy: *Writeback*
Write Allocation Policy: *Allocate on miss*
Hit Time: 2 cycles (four steps)
Miss Time: *7 cycles (first 8 bytes, 2 clock cycles per 8 bytes afterwards)*
Victims: 8



Latency vs Bandwidth

- **Latency:** How long does it take for the cache to serve a single request? In other words, how long does it take to access a single cache block? (critical for in-order, important for OOO)
- **Bandwidth:** How many requests can the cache serve in one unit of time? E.g., Requests per second (very important for OOO processors)

- **Optimizations to reduce hit time**
  - Small cache sizes
  - Way prediction
  - Low associativity
- **Optimizations to reduce miss penalty**
  - Writeback buffer
  - Victim buffer
  - Early restart and critical word first
  - Prefetching
- **Optimizations to improve cache bandwidth**
  - Split instruction/data cache
  - Lock-up free (a.k.a. non-blocking caches)
  - Multibanking
  - Pipelining

Critical Word First & Early Restart

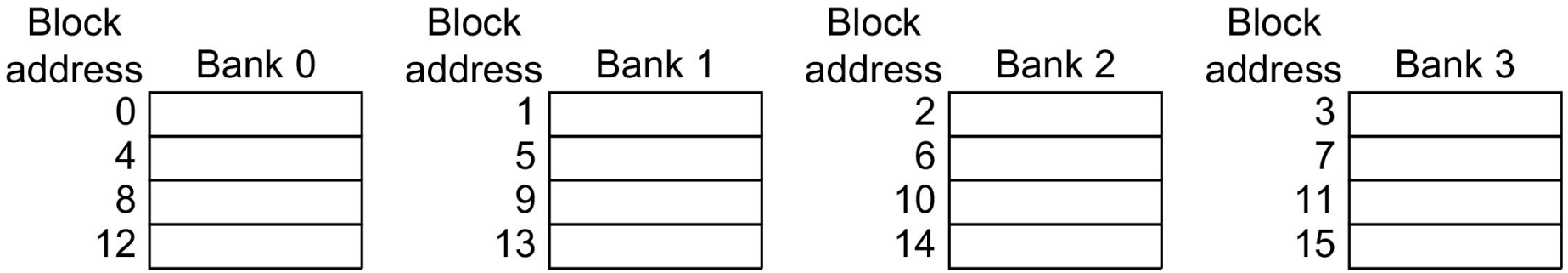
- **Critical Word First**
  - Request the missed word first from memory
  - Send it to the processor
  - Continue with the rest of the request
  - The processors begins execution as soon as it receives the missed word
- **Early Restart**
  - Fetch the words in normal order
  - As soon as the requested word arrives, send it to the processor

- ***Note:*** *Although memory is read in blocks, these optimizations inform the memory system the specific word in a block to send first*

- Question: How do these optimizations interact with spatial locality?

Multibanking

- **Divide the cache into independent banks each supporting an independent acceess**
  - One approach is sequential interleaving
  - Four banks: 0→Bank 0, 1→Bank 1, 2→Bank 2, 3→Bank 3
  - Increase request bandwidth
- **Some questions:**
  - When does banking work well? Does banking reduce power/complexity?



Non-Blocking Caches

- **Blocking cache**
  - On a miss, the cache is locked up
  - Cannot serve another request
  - No internal structures (buffers, memory) to remember miss information
    - Therefore, cannot take another request until the miss is serviced
- **Non-blocking (lockup-free cache)**
  - Continue serving request underneath a miss
  - Hit-under-Miss
  - Hit-under-Multiple-Miss
    - The memory system should be able to serve multiple requests as well
- **OOO processors exploit non-blocking caches really well**
  - Does not help to have an in-order core that stalls on a miss and be able to serve many memory requests in parallel

Miss Status Handling Register (MSHR)

- **Miss Status Handling Register**
  - It is like a miss buffer
  - Tracks outstanding cache misses
  - Fields of an MSHR entry
    - Valid bit
    - Cache block address (to match incoming addresses)
    - Block data
    - Other control information
      - Load/Store, Destination register