Cache Memories

Sample Problem

A cache has a capacity of 32 KB and 256-byte lines. On a machine with a 32-bit virtual address space, how many bits long are the tag, set, and offset fields for

- A direct-mapped implementation?
- A four-way set-associative implementation?
- A fully-associative implementation?

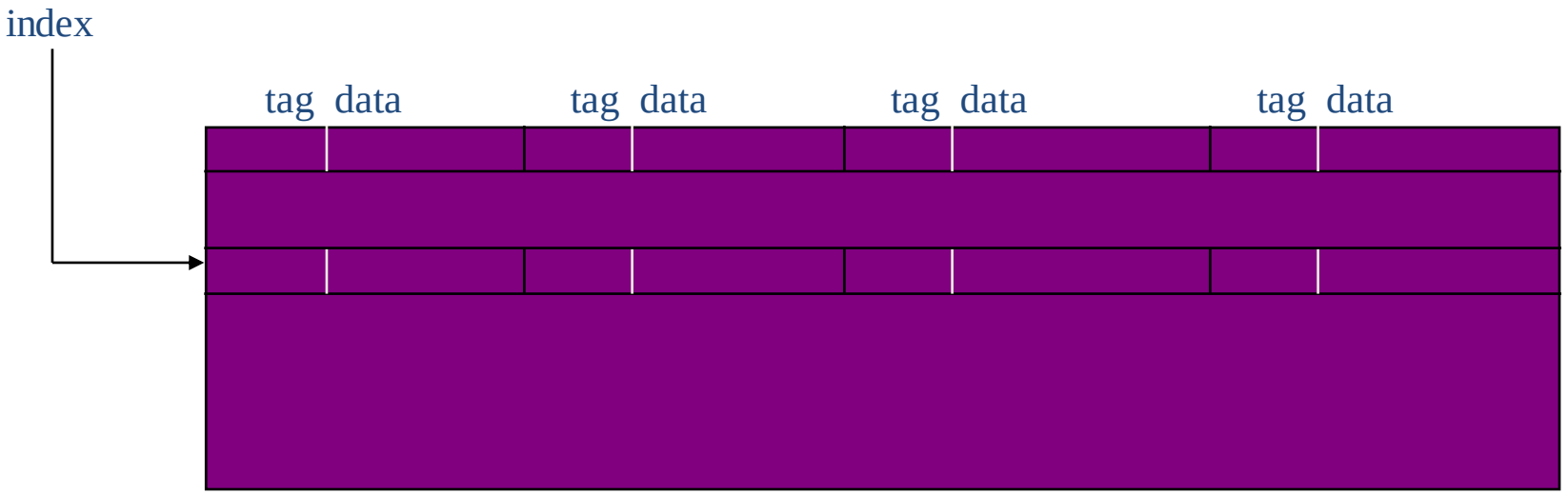
Sample Problem Version II

A cache has a capacity of 32 KB and 256-byte in a set. On a machine with a 32-bit address space, how many bits long are the tag, set, and offset fields for

- A direct-mapped implementation?
- A four-way set-associative implementation?
- A eight-way set-associative implementation?

Another example

- 16 KB, 4-way set-associative cache, 32-bit address, byte-addressable memory, 32-byte cache blocks/lines
- how many tag bits?
- Where would you find the word at address 0x200356A4?



PIRW

- Placement
- Identification
- Replacement
- Writes

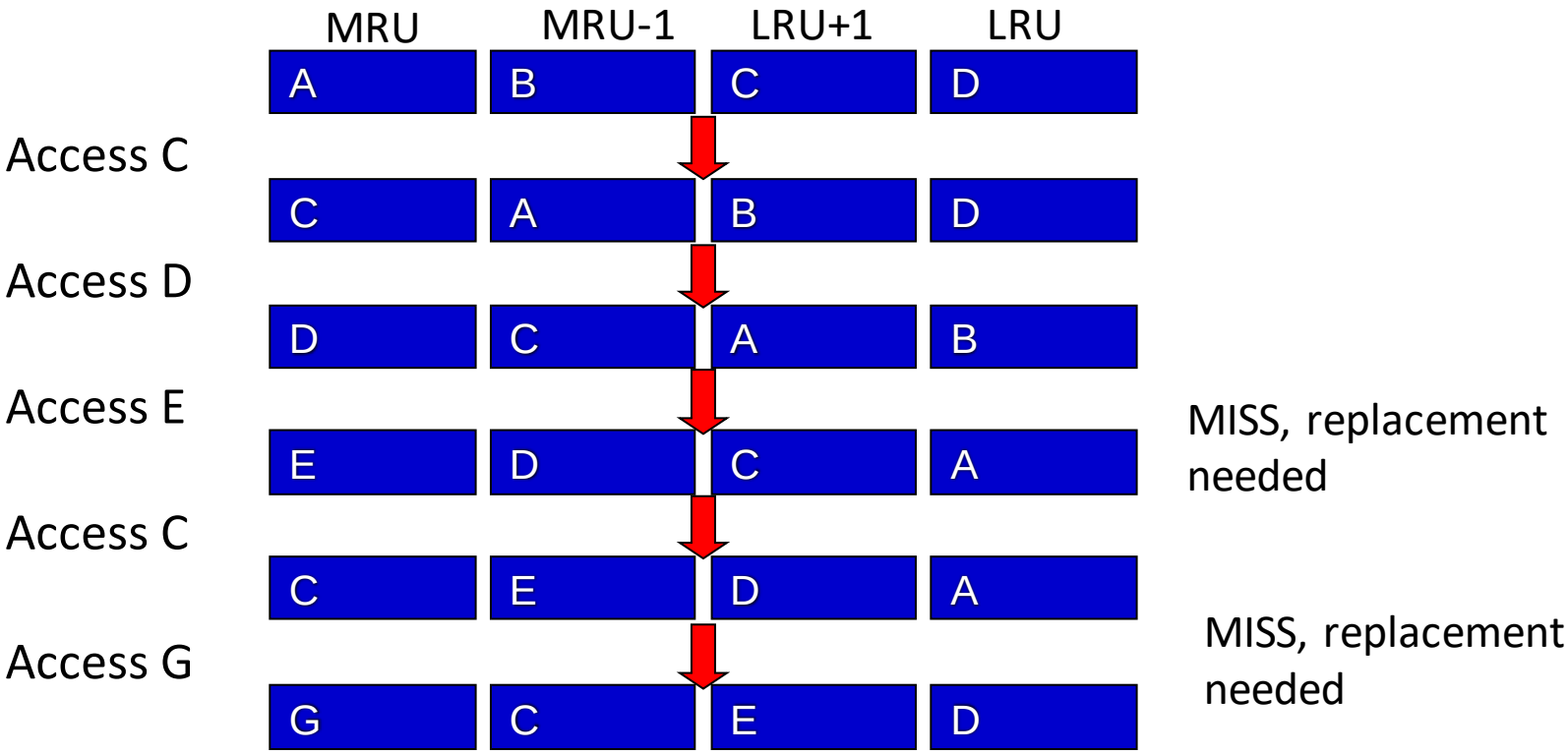
Replacement Policies for Associative Caches

- Least-Recently-Used (LRU): Evict the line that has been least recently referenced
  - Need to keep track of order that lines in a set have been referenced
  - Overhead to do this gets worse as associativity increases
- Random: Just pick one at random
  - Easy to implement
  - Slightly lower hit rates than LRU on average
- Not-Most-Recently-Used: Track which line in a set was referenced most recently, pick randomly from the others
  - Compromise in both hit rate and implementation difficulty
- Virtual memories use similar policies, but are willing to spend more effort to improve hit rate

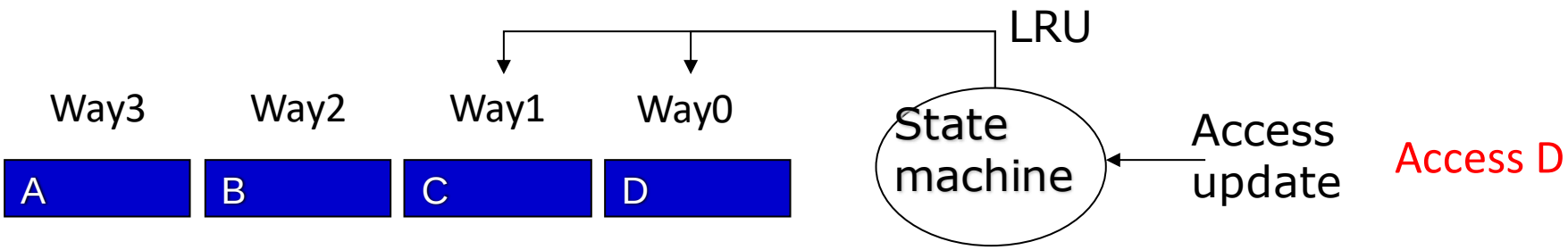
LRU: Precise Tracking

- Precise LRU tracking requires a stack for each set
  - When a block is accessed by the processor, check if its name is in the stack. If so, remove it from the stack. Push the name of block at the top of the stack
  - The position (depth) of a block name in the stack gives the relative recency of access to this block compared to others
  - For a 4-way set associative cache with blocks A, B, C, D.
    - Assume a sequence of accesses C, D, A, B, A, C, B, D
    - Assume that the stack is initially empty, the configuration of the stack after each access would be [C,-,-,-], [D, C,-,-], [A, D, C,-], [B, A, D, C], [A, B, D, C], [C, A, B, D], [B, C, A, D], [D, B, C, A]
    - The one on the right most position (bottom of the stack) is the least recently used
    - Each name takes two bits, so the stack is an 8-bit register with associated logic ($S\log_2 S$ where S is associativity)

LRU Example


MRU
MRU-1
LRU+1
LRU
A
B
C
D
Access C
C
A
B
D
Access D
D
C
A
B
Access E
E
D
C
A
MISS, replacement needed
Access C
C
E
D
A
Access G
G
C
E
D
MISS, replacement needed

LRU From Hardware Perspective



LRU policy increases cache access times

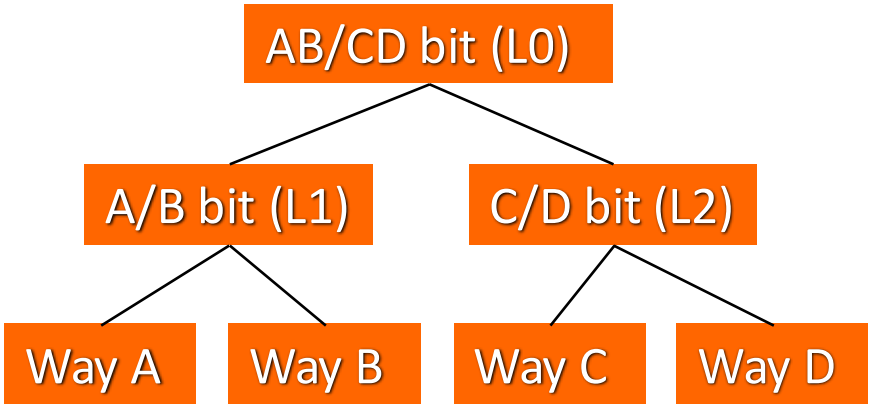
Additional hardware bits needed for LRU state machine

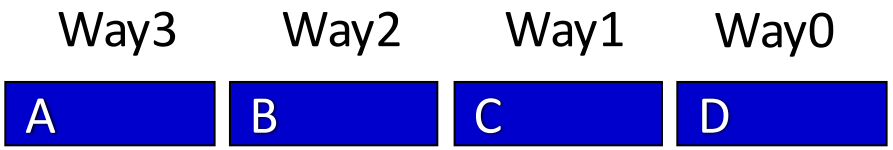
LRU: Approximate Tracking (Pseudo-LRU)

- LRU stacks expensive to implement at high ‘S’
- Most caches use approximate LRU
  - A popular approach uses S-1 bits for an S-way cache
    - The blocks are hierarchically divided into a binary tree
    - At each level of the tree, one bit is used to track the least recently used
  - For a 4-way set associate cache, blocks are first divided into two halves, each half has two blocks
    - The 1st bit tracks the more recently used half
    - The 2nd bit (3rd) tracks the more recently block in the first (second) half
    - The one to replace is the less recently used block in the less recently used half
- used in the CPU cache of many commercial processors

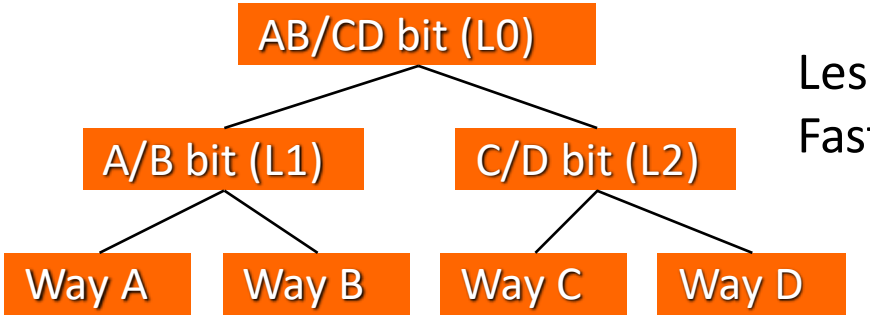
Pseudo LRU Algorithm (4-way SA)

- Tree-based
  - O(N): 3 bits for 4-way
  - Cache ways are the leaves of the tree
  - Combine ways as we proceed towards the root of the tree





Pseudo LRU Algorithm



Less hardware than LRU &
Faster than LRU

L2L1L0 = 000,
there is a hit in Way B, what is the new updated L2L1L0?

L2L1L0 = 010,
a way needs to be replaced, which way would be chosen?

LRU update algorithm

	CD	AB	AB/CD
Way hit	**L2**	**L1**	**L0**
Way A	---	0	0
Way B	---	1	0
Way C	0	---	1
Way D	1	---	1

Replacement Decision

CD	AB	AB/CD	
L2	**L1**	**L0**	**Way to replace**
X	1	1	Way A
X	0	1	Way B
1	X	0	Way C
0	X	0	Way D

Pseudo LRU example

- For a 4-way cache with blocks A, B, C, & D.
  - Assume that A and B form the first half (half 0) and C and D form the second half (half 1)
  - Within the first half, A is block 0 and B is block 1.
  - Within the second half, C is block 0 and D is block 1.
  - Assume sequence of accesses C, D, A, B, A, C, B, D
  - The configuration of the tree after each access would be [1, -, 0], [1, -, 1], [0, 0, 1], [0, 1, 1], [0, 0, 1], [1, 0, 0], [0, 1, 0], [1, 1, 1]
  - Note that a replacement after C, D, A, B, A, C would have chosen B rather than D (D would be picked by LRU).

LRU Algorithms

- True LRU
  - Expensive in terms of speed and hardware
  - Need to remember the order in which all N lines were last accessed
  - N! scenarios – O(log N!) ≈ O(N log N) LRU bits
    - 2-ways → AB BA = 2 = 2!
    - 3-ways → ABC ACB BAC BCA CAB CBA = 6 = 3!
- Pseudo LRU: O(N)
  - Approximates LRU policy with a binary tree

How about an approximate LRU for 8-way?

A Sample Replacement Policy Question

- A byte-addressable computer has a small 32-byte cache. Assume 4-byte cache lines. When a given program is executed, the processor reads data from the following sequence of hex addresses: 200, 204, 208, 20C, 2F4, 2F0, 200, 204, 218, 21C, 24C, 2F4. This pattern is repeated four times.
  - (a) Show the contents of the cache at the end of each pass throughout this loop if a direct-mapped cache is used. Compute the hit rate for this example. Assume that the cache is initially empty.
  - (b) Repeat part (a) for a fully-associative cache that uses the LRU-replacement algorithm.
  - (c) Repeat part (a) for a fully-associative cache that uses the approximate LRU replacement algorithm.

You should be able to complete the answer

- 200, 204, 208, 20C, 2F4, 2F0, 200, 204, 218, 21C, 24C, 2F4 (Hex Address)
- 4-byte blocks so block address sequence in Hex is
  - 80, 81, 82, 83, BD, BC, 80, 81, 86, 87, …
- Direct mapped, 8 blocks
  - 1000 0000, 1000 0001, 1000 0010, 1000 0011, 1010 1101, 1010 1100, 1000 0000, 1000 0001, 1000 0110, 1000 0111,
  - For direct map, there is no room for policy
  - Complete your answer…
- For fully associative cache, it is really an 8-way, 1-set cache, all block address bits are used
  - Complete your answer…

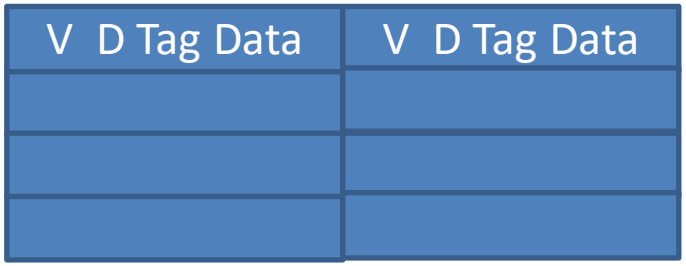
Another Question

- Consider a 128 byte 2-way set associative write-back cache with 16 byte blocks. Assume LRU replacement and that dirty bits are used to avoid writing back clean blocks. Complete the table below for a sequence of memory references (occurring from left to right).
- Address (in decimal): 064 032 064 000 112 064 128 048 240 000 read/write: r r r r w w r r r w
- Assume that cache starts empty

You should complete the answer

16 byte blocks, 4 LSB byte offset

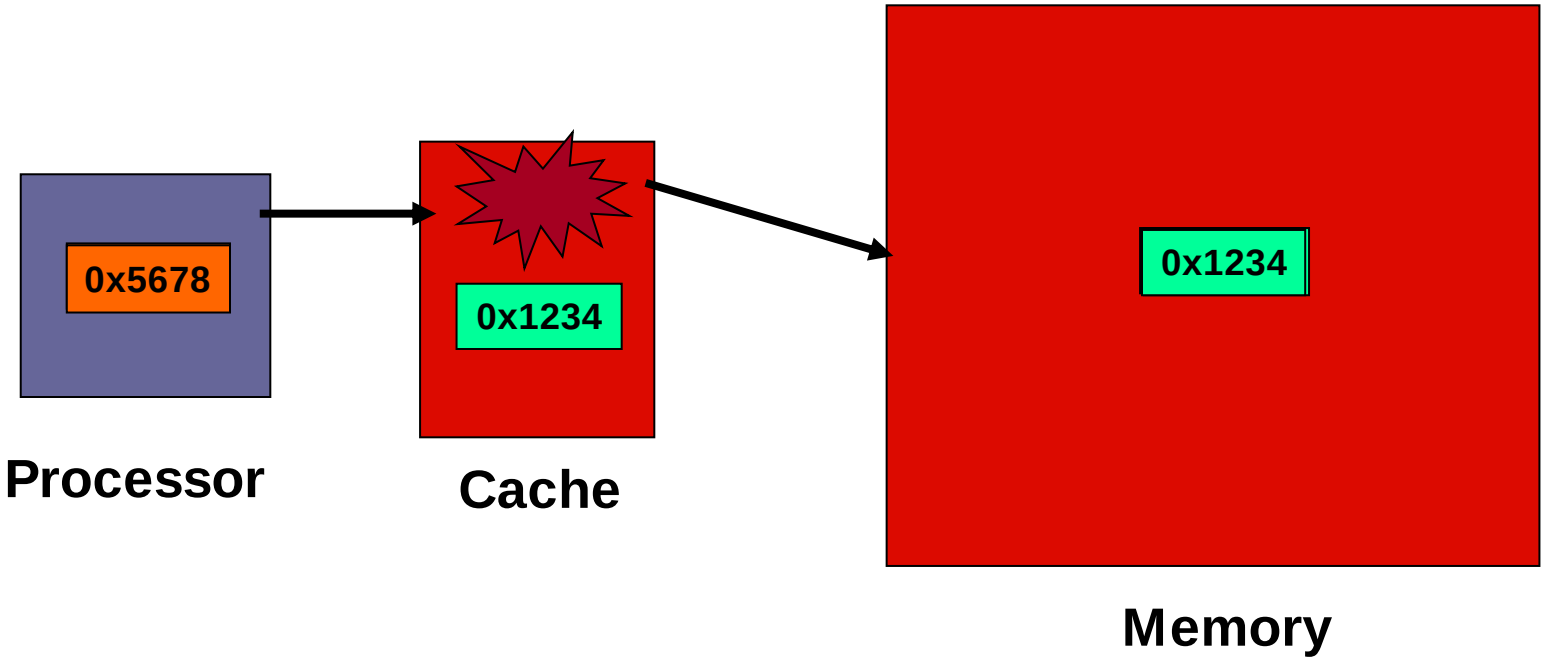
064 032 064 000 112 064 128 048 240 000



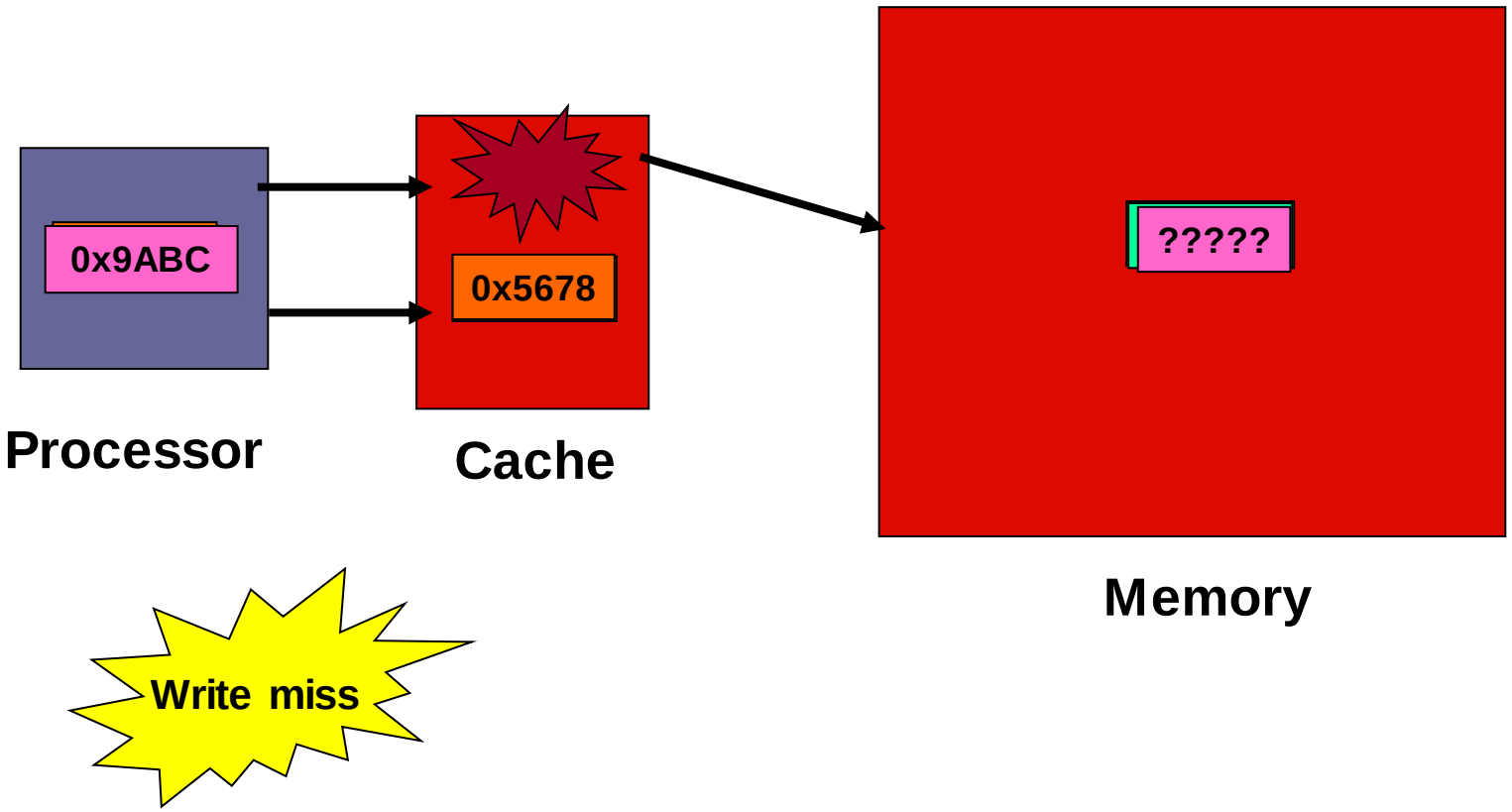
There are 4 sets with 2 blocks each

Block Addr.	4	2	4	0	7	4	8			
R/W	R	R	R	R	W	W	R	R	R	W
Set#	0	2	0	0	3	0				
Tag	1	0	1	0	1	1				
H/M	M	M	H	M						
Write back?	N	N								

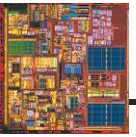
Write-through Policy



Write-back Policy

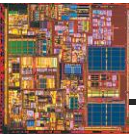


On Write Miss



- Write allocate
  - The line is allocated on a write miss, followed by the write hit actions above.
  - Write misses first act like read misses
  - Better performance if data referenced again
- No write allocate
  - Write misses do not interfere cache
  - Line is only modified in the lower level memory
  - Mostly use with write-through cache
  - Simpler write hardware
  - May be better for small caches if written data won't be read again soon

Another Question



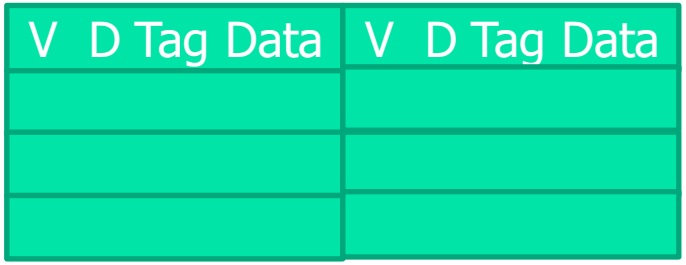
- Consider a 128 byte 2-way set associative write-back cache with 16 byte blocks. Assume LRU replacement and that dirty bits are used to avoid writing back clean blocks. Complete the table below for a sequence of memory references (occurring from left to right).
- Address (in decimal): 064 032 064 000 112 064 128 048 240 000 read/write: r r r r w w r r r w
- Assume that cache starts empty

You should complete the answer

16 byte blocks, 4 LSB byte offset

064 032 064 000 112 064 128 048 240 000

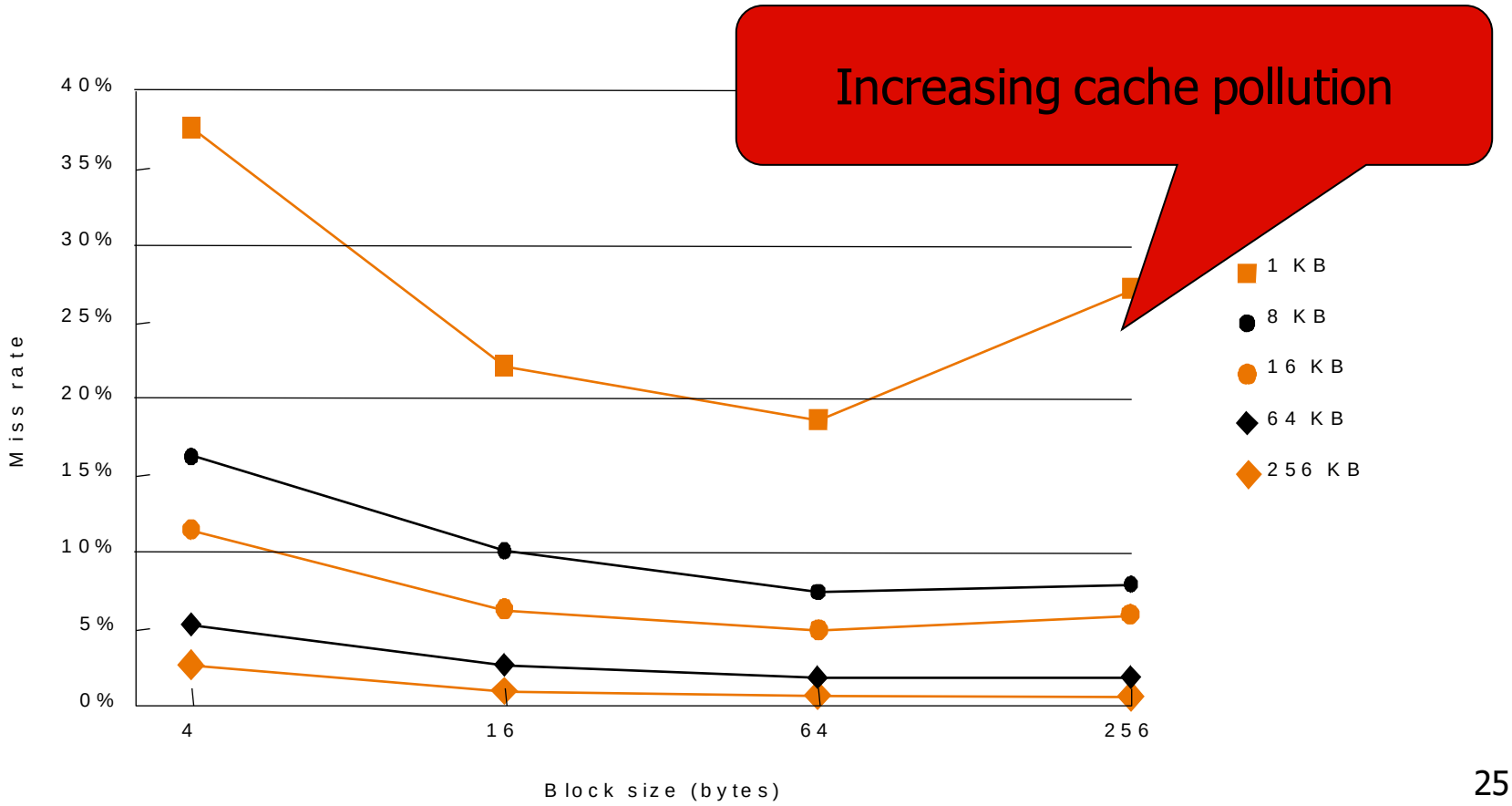
There are 4 sets with 2 blocks each



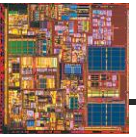
Block Addr.	4	2	4	0	7	4	8			
R/W	R	R	R	R	W	W	R	R	R	W
Set#	0	2	0	0	3	0				
Tag	1	0	1	0	1	1				
H/M	M	M	H	M						
Write back?	N	N								

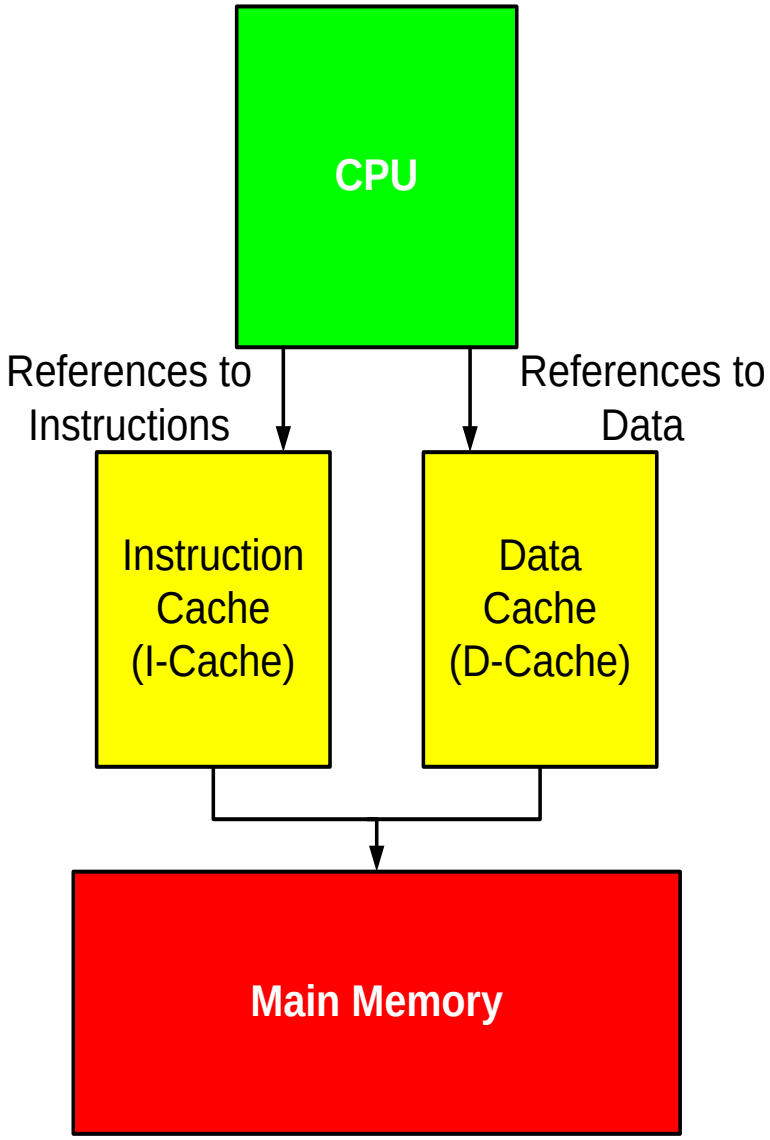
Reducing Miss Rate

- Enlarge cache
- If cache size is fixed
  - Increase associativity and/or line size (does this always work?)

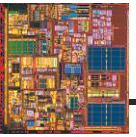


Instruction and Data Caches





Why Do We Do This?



- Bandwidth: lets us access instructions and data in parallel
- Most programs don't modify their instructions
- I-Cache can be simpler than D-Cache, since instruction references are never writes
- Instruction stream has high locality of reference, can get higher hit rates with small cache
  - Data references never interfere with instruction references