

# 18-447 Lecture 15: Principles of Caching

James C. Hoe

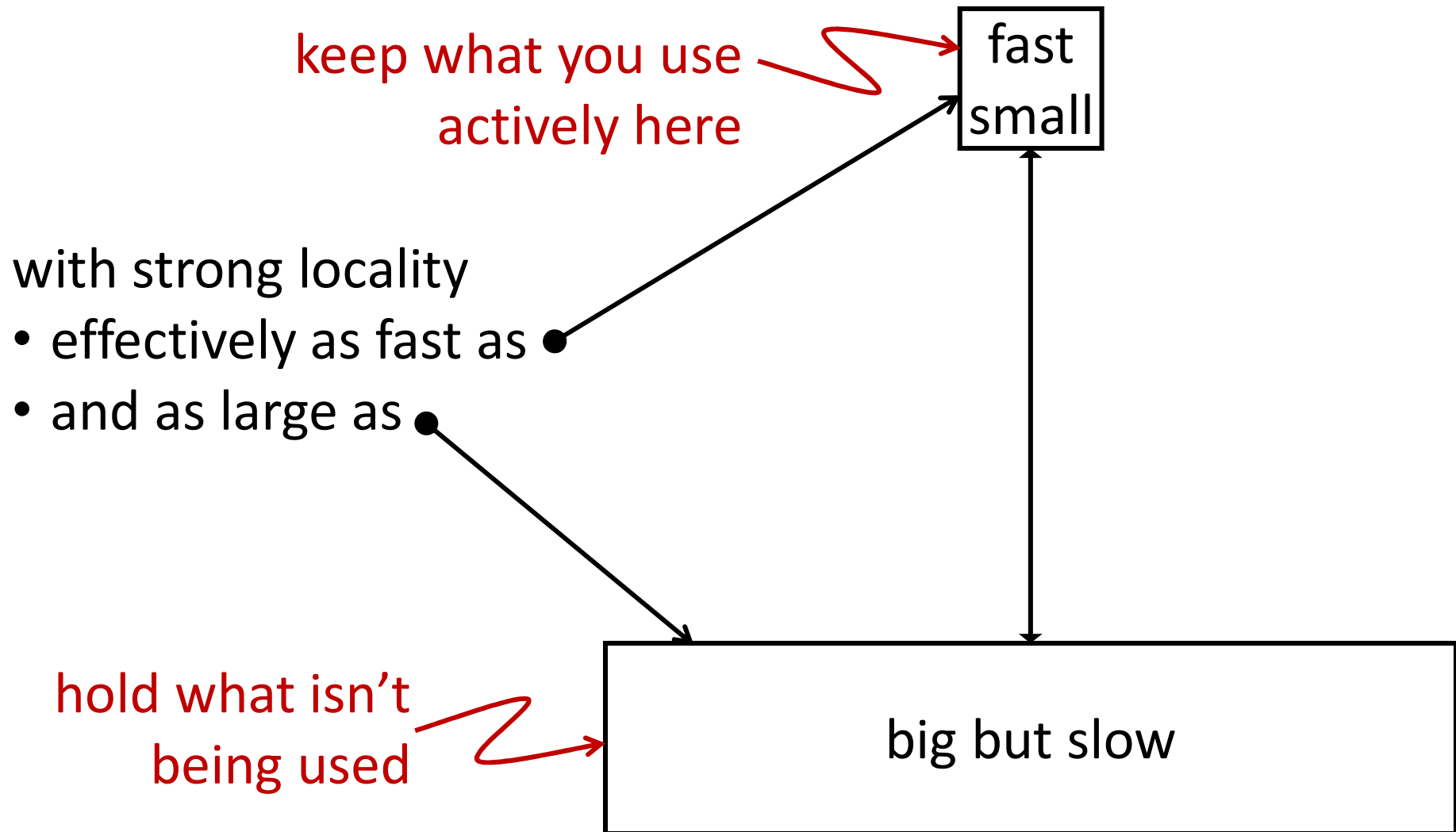
Department of ECE

Carnegie Mellon University

# Housekeeping

- Your goal today
  - recover from Spring Break
  - understand “aBC” of caches
  - understand “3 C’s” of caches
- Notices
  - Lab 3, **due week of 3/30 (extended)**
  - Handout #12 HW 4, **due Friday 4/3**
  - Midterm 1 regrade due Friday on Gradescope
- Readings
  - P&H Ch 5

# Memory Hierarchy



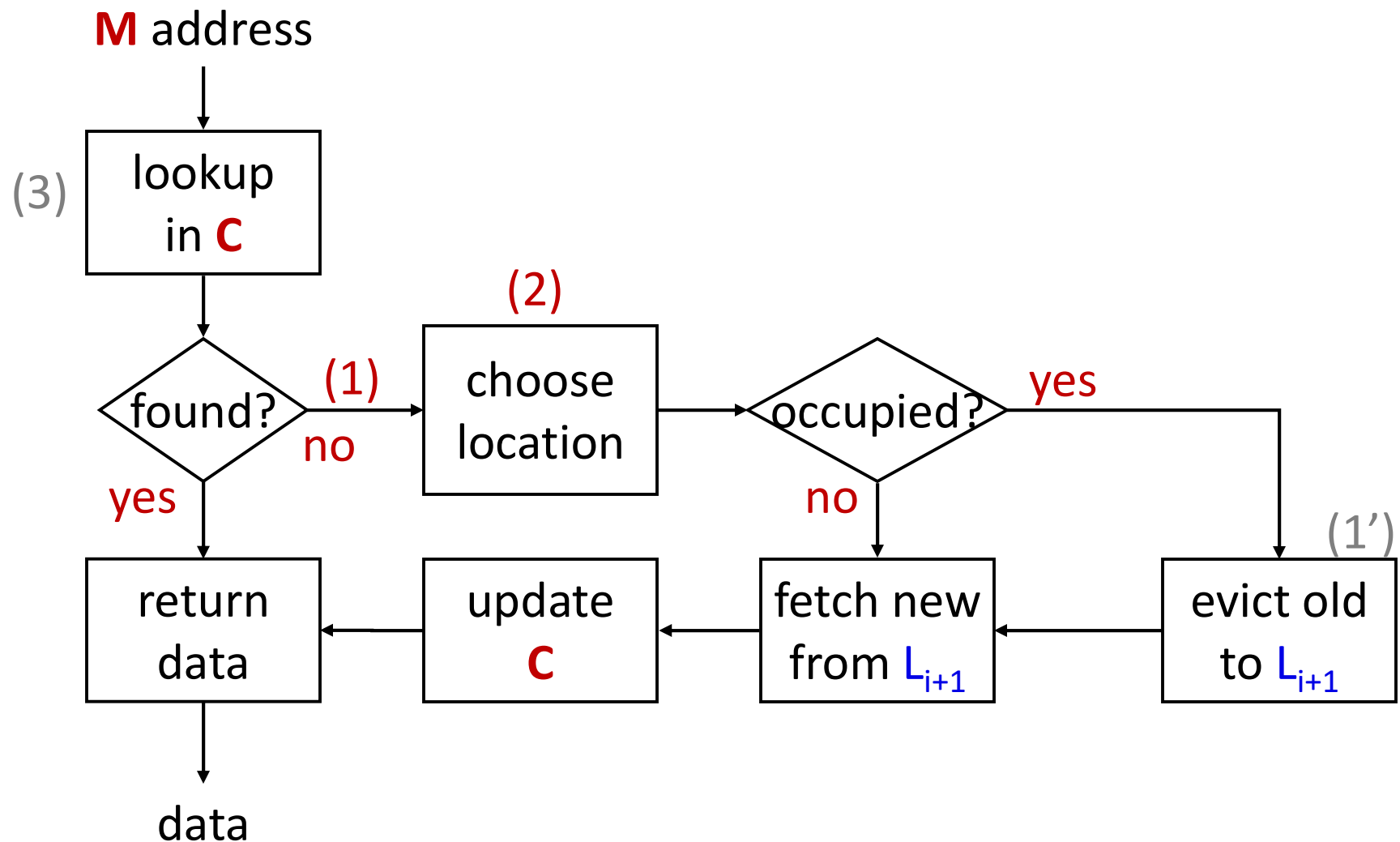
# Problem in the Abstract

- Potentially  $M=2^m$  bytes of memory, how to keep “copies” of most frequently used locations in  $C$  bytes of fast storage where  $C \ll M$
- Basic issues (intertwined)
  - (1) when to cache a “copy” of a memory location
  - (2) where in fast storage to keep the “copy”
  - (3) how to find the “copy” later on (*LW and SW only give indices into  $M$* )

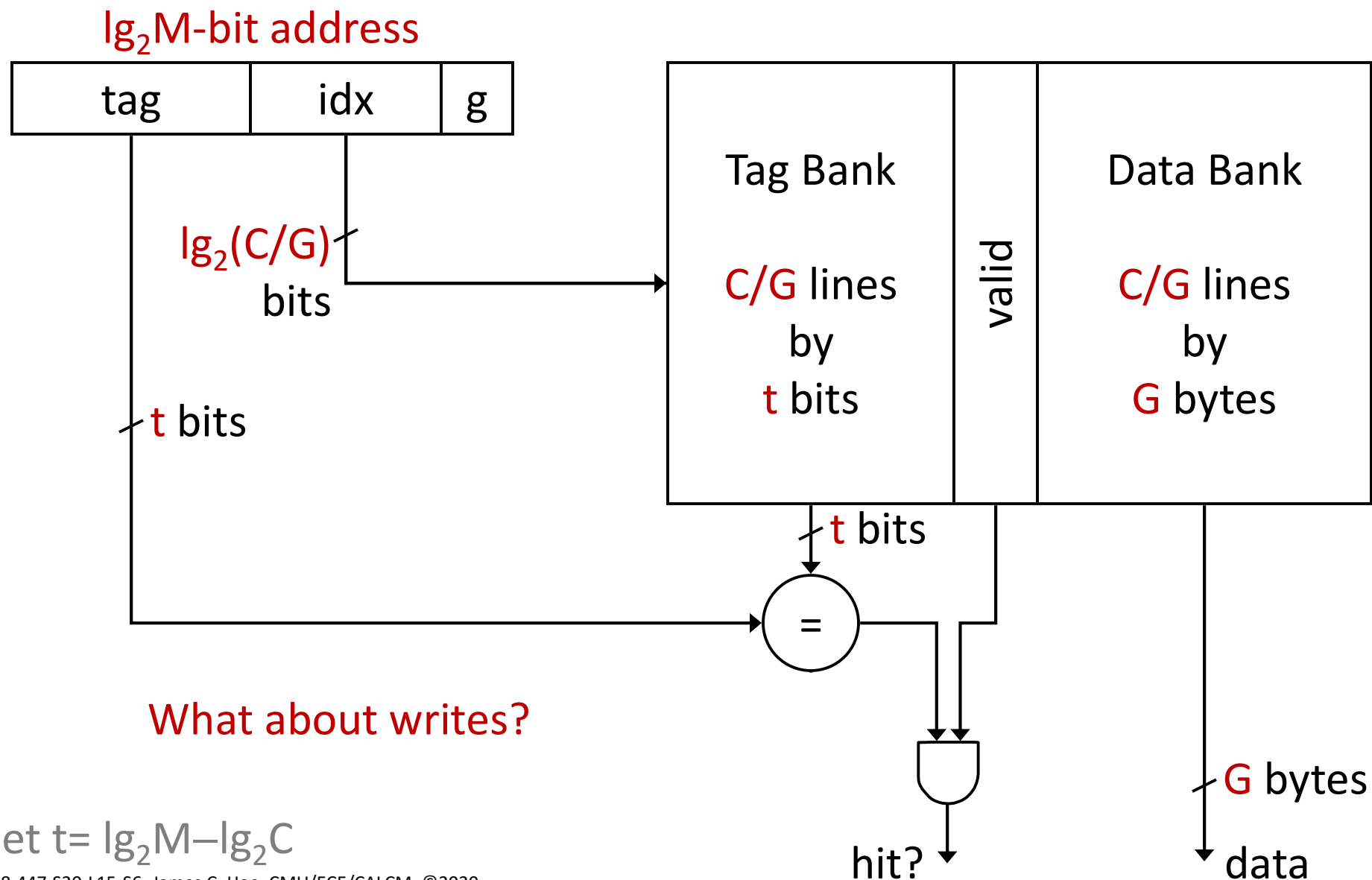
C  
Capacity

What to do if  $C$  is 32 GPR registers?

# Demand-Driven Strategy



# Direct-Mapped Placement (v1)



$$\text{let } t = \lg_2 M - \lg_2 C$$

# Storage Overhead and **B**lock Size

- For each cache block of **G** bytes, also storing “**t+1**” bits of tag (where  $t = \lg_2 M - \lg_2 C$ )
  - if  $M = 2^{32}$ ,  $G = 4$ ,  $C = 16K = 2^{14}$
  - $\Rightarrow t = 18$  bits for each 4-byte block

60% overhead; 16KB cache actually 25.5KB SRAM

- Solution: “amortize” tag over larger **B**-byte block
  - manage **B/G** consecutive words as indivisible unit
  - if  $M = 2^{32}$ ,  $B = 16$ ,  $G = 4$ ,  $C = 16K$
  - $\Rightarrow t = 18$  bits for each 16-byte block

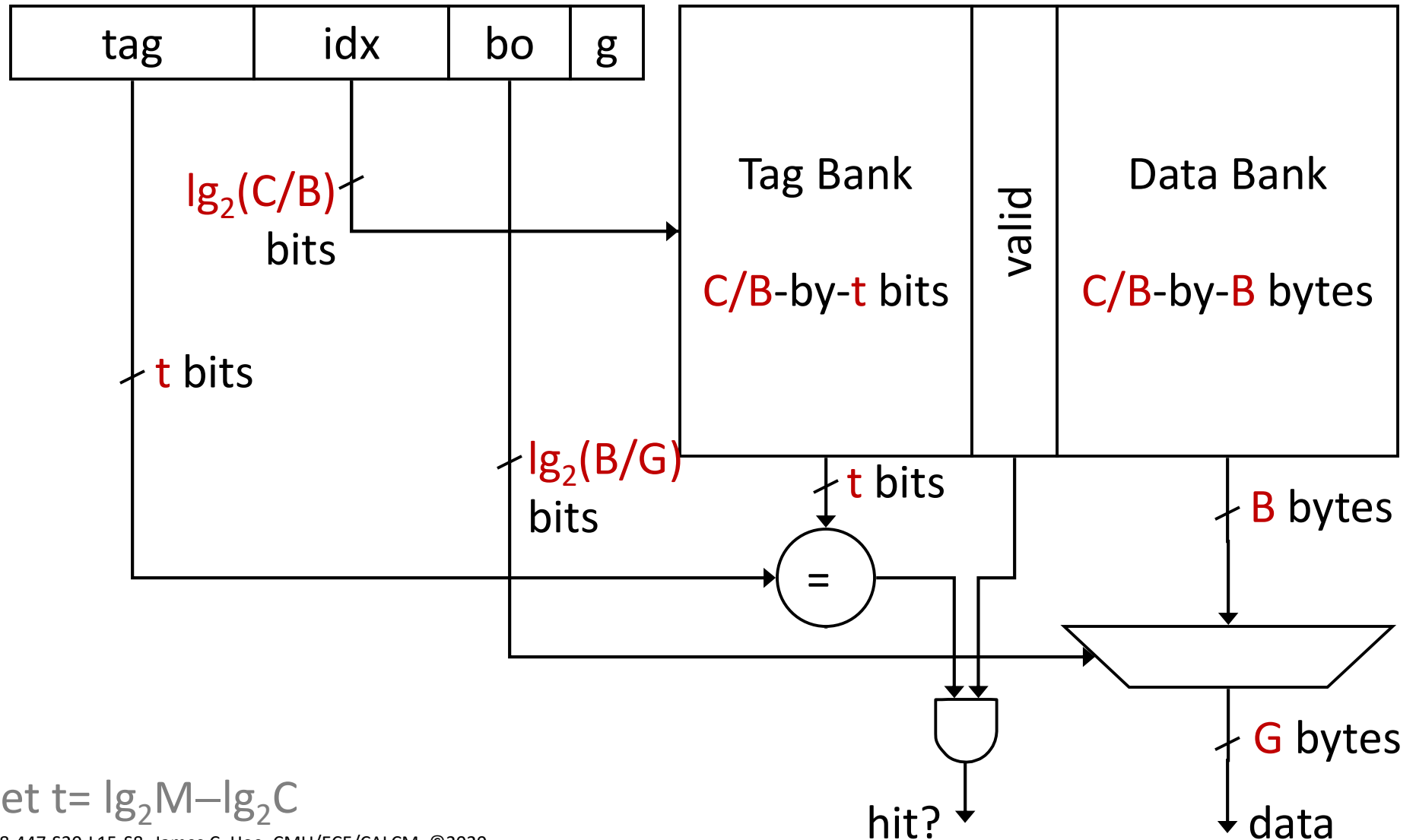
15% overhead; 16KB cache actually 18.4KB SRAM

– spatial locality also says this is good (*Q1: when*)

- **B**. Larger caches wants even bigger blocks

# Direct-Mapped Placement (final)

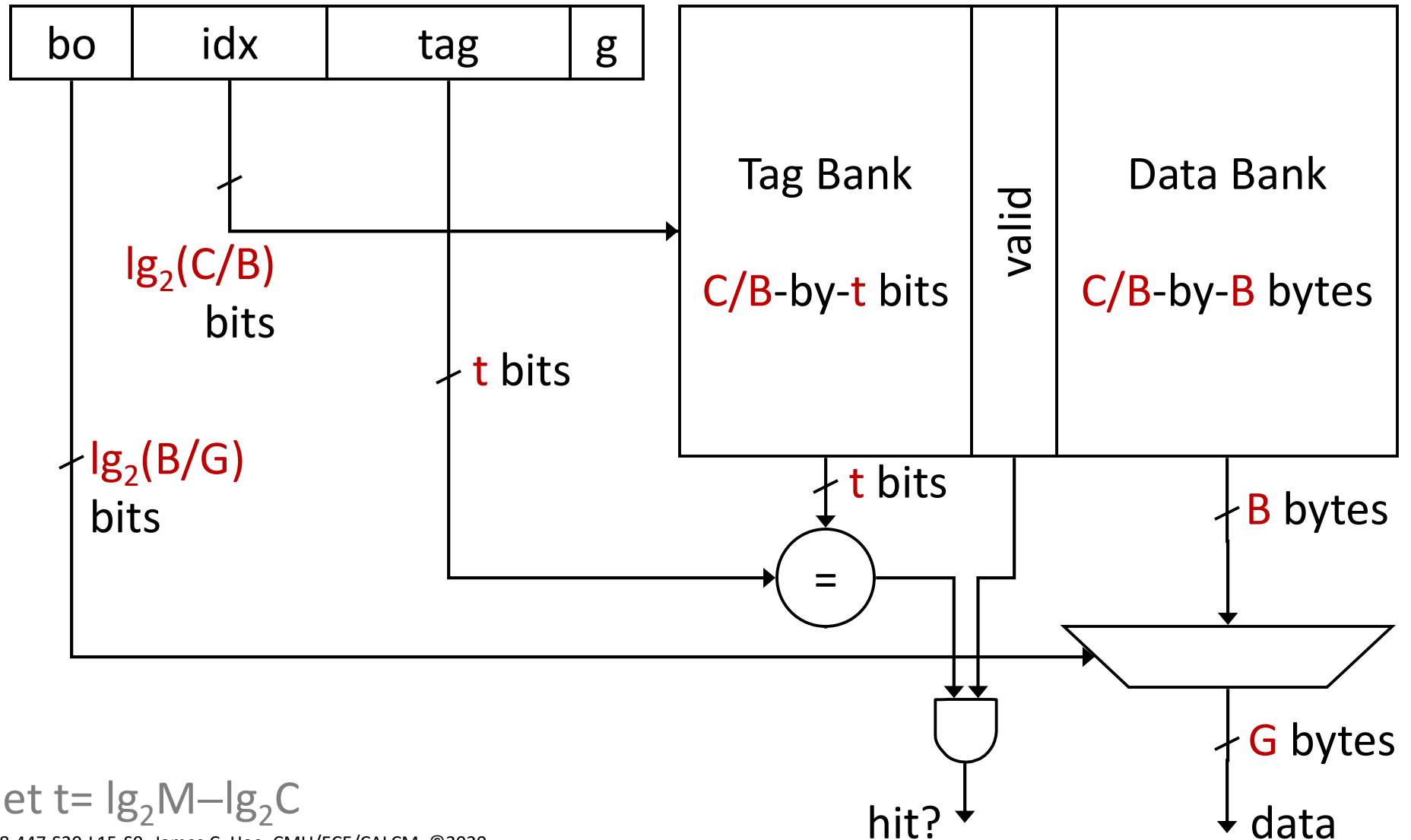
$\lg_2 M$ -bit address



let  $t = \lg_2 M - \lg_2 C$

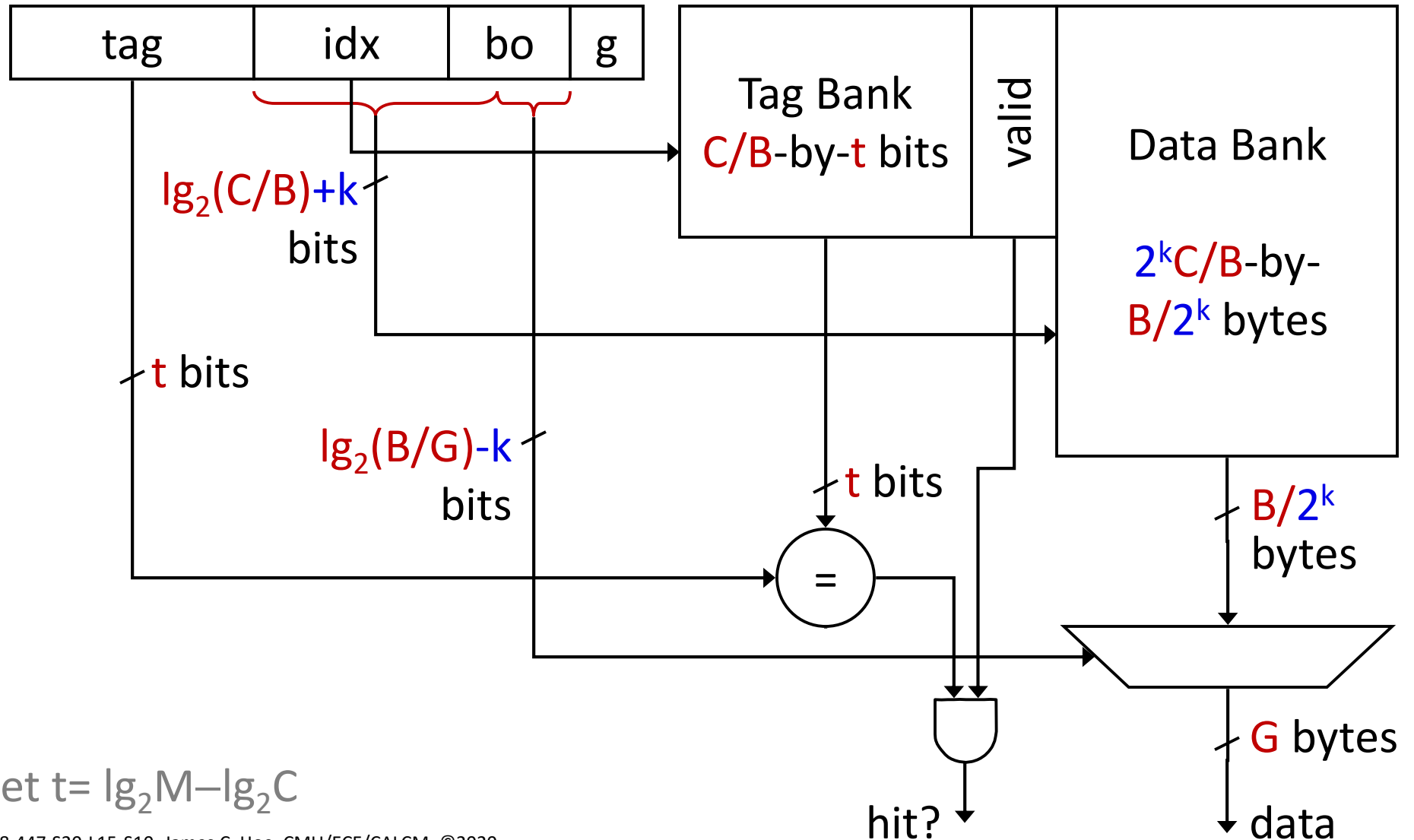


# Is this okay?



let  $t = \lg_2 M - \lg_2 C$

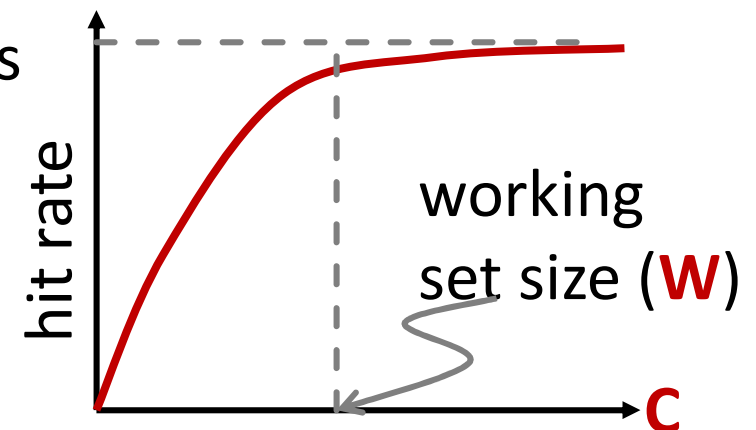
# Is this okay?



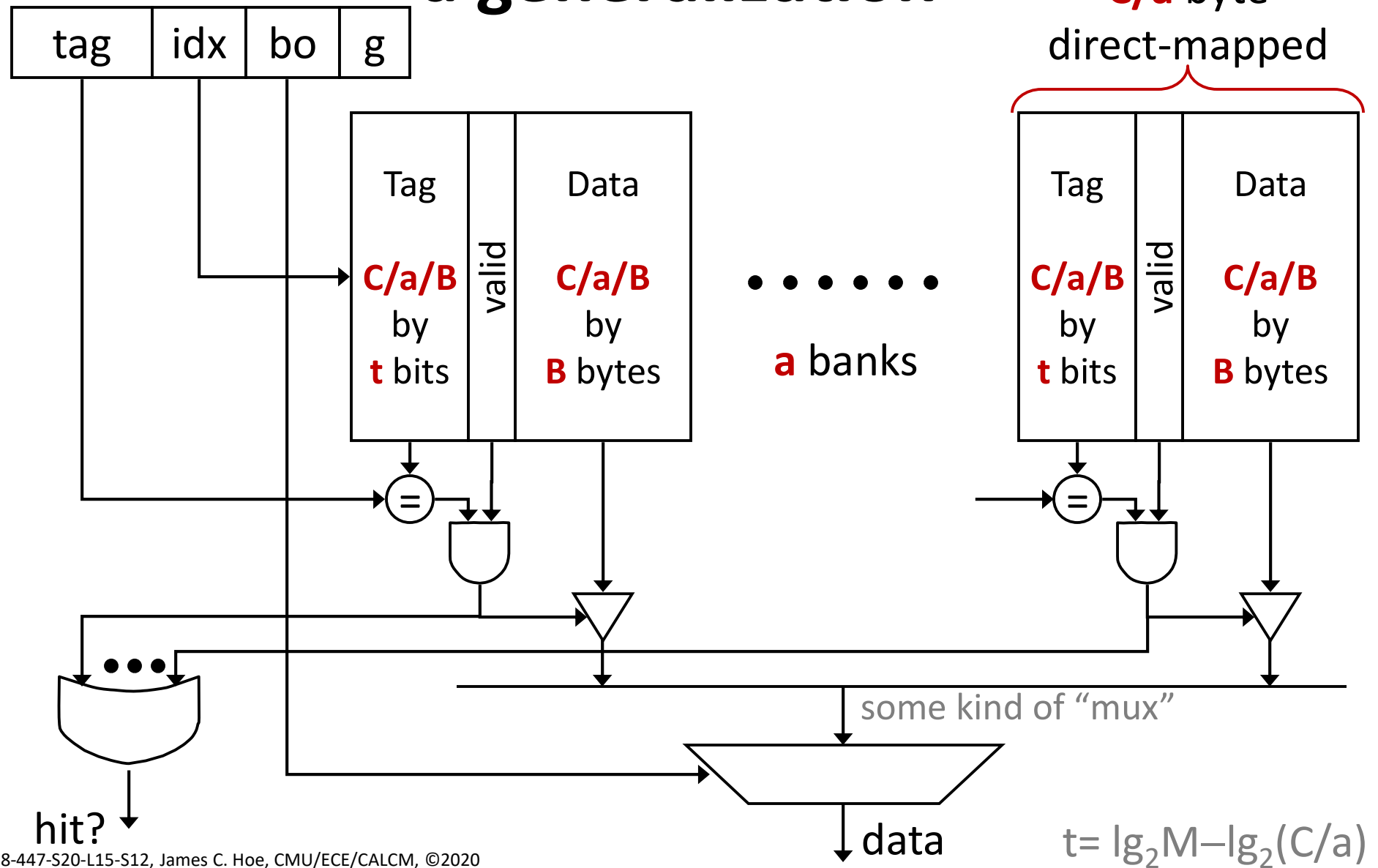
let  $t = \lg_2 M - \lg_2 C$

# Direct-Mapped Policy in Essence

- $C$  bytes of storage managed as  $C/B$  cache blocks
- A given block address directly maps to exactly one choice of cache block (by block index field)
- Block addresses with same block index field map to same cache block
  - of  $2^t$  such addresses, hold only one at a time
  - even if  $C >$  working set size, conflict is possible  
 (“working set” is not one continuous region)
  - probability 2 random addresses conflict is  $1/(C/B)$ ; likelihood for conflict decreases with increasing number of blocks



# Set Associative Placement Policy: a generalization



# **a**-way Set-Associative Placement

- **C** bytes of storage divided into **a** direct-mapped banks (aka “ways”)
  - each “way” has  $(C/a)/B$  cache blocks
  - a given block address maps to exactly one choice per “way”; **a** choices constitute the “set”

direct-mapped is special case **a**=1

- overhead: **a** comparators and **a**-to-1 multiplexer
- Block addresses with same index map to same set
  - $2^t$  such addresses; hold **a** different ones at a time
  - if **C** > working set size

higher-degree of associativity  $\Rightarrow$  fewer conflicts

What if **C** < working set size?

**a**ssociativity

# Replacement Policy to Choose from **a**

- New block displaces an existing block from “set”
  - pick the one that is least recently used (LRU)
    - exactly LRU expensive for **a**>2
  - pick any one except the most recently used
  - ~~pick the most recently used one~~
  - ~~pick one based on some part of the address bits~~
  - pick the one used again furthest in the future Belady
  - pick a (pseudo) random one
- No real best choice; second-order impact only
  - if actively using less than **a** blocks in a set, any sensible replacement policy will quickly converge
  - if actively using more than **a** blocks in a set, no replacement policy can help you

# Policy vs Realization

- Associativity is a placement policy
  - it says a block address could be placed in one of **a** different blocks
  - it doesn't say "ways" are parallel look-up banks
- "Pseudo" **a**-way associative cache
  - given a direct-mapped array with **C/B** blocks
  - logically partition into **C/B/a** sets
  - given an address **A**, index into set and sequentially search its ways:
- Optimization: record the most recently used way (MRU) to check first

set0 way0  
set0 way1  
set0 way2

.....

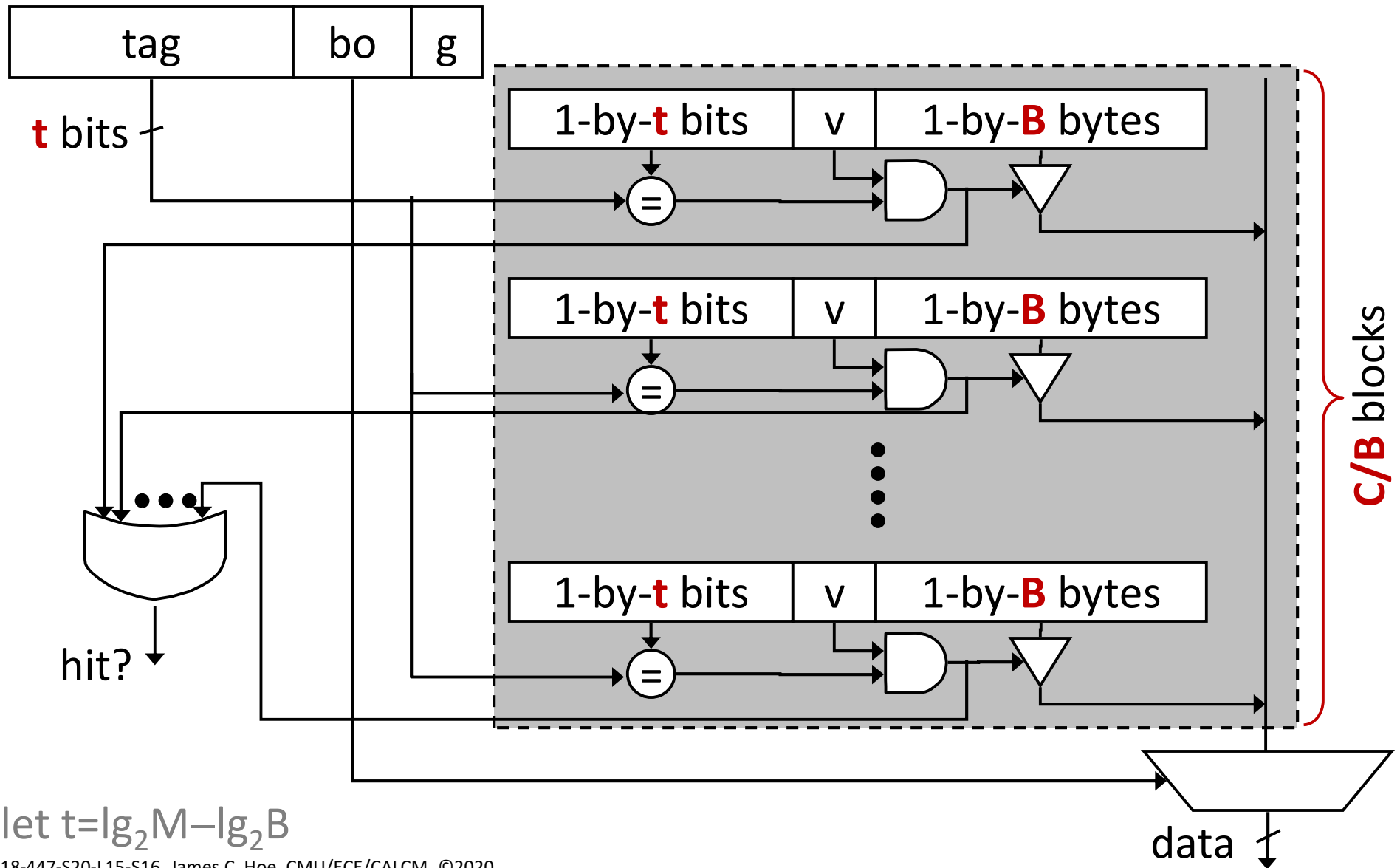
set1 way0  
set1 way1  
set1 way2

.....



e.g., used by MIPS R10K L2

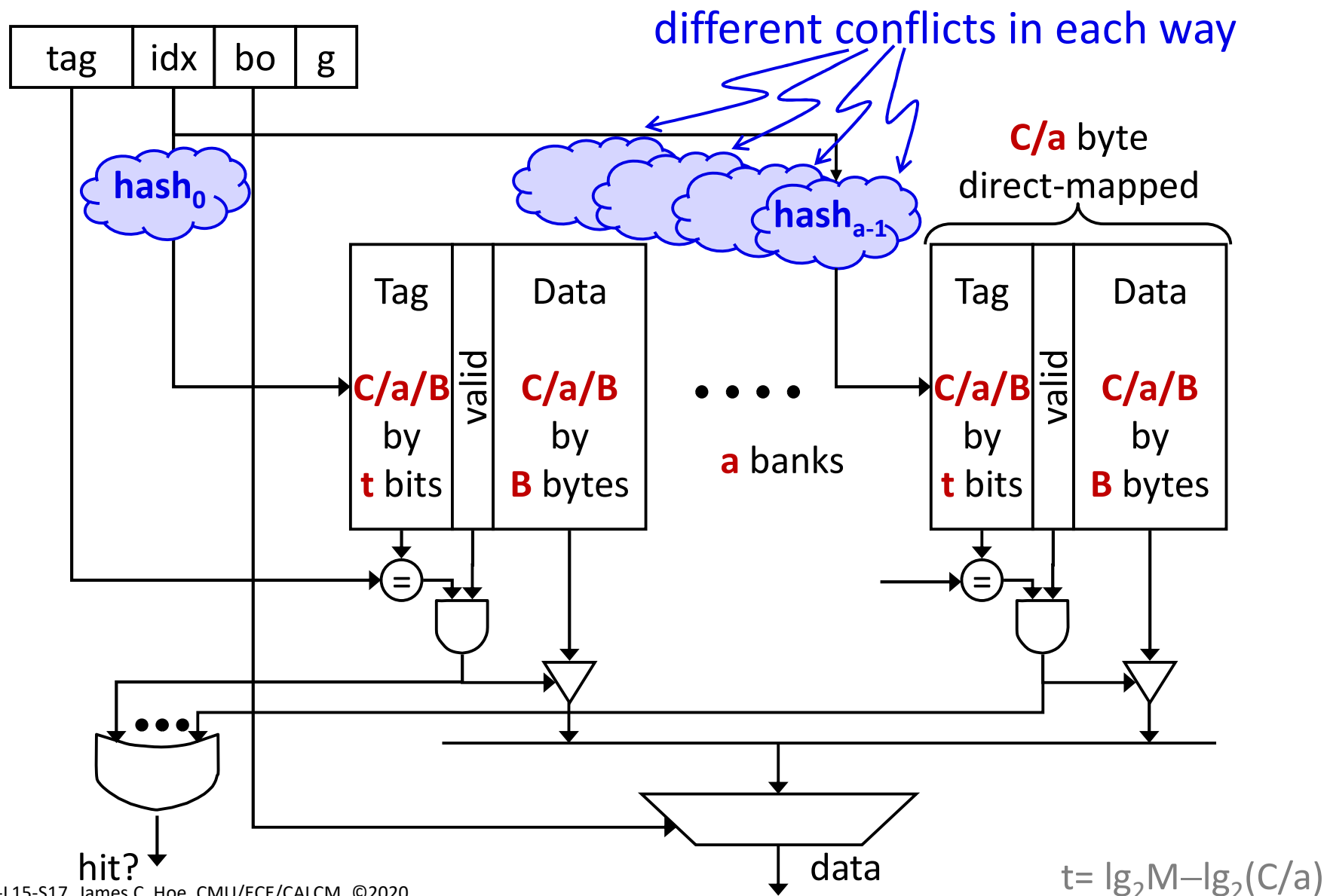
# Fully Associative Cache: $a \equiv C/B$



$$\text{let } t = \lg_2 M - \lg_2 B$$



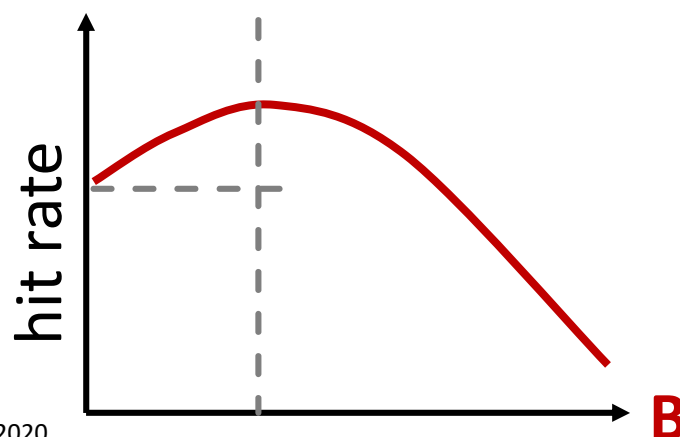
# Skewed Associative Cache



# 3C's of Cache Misses

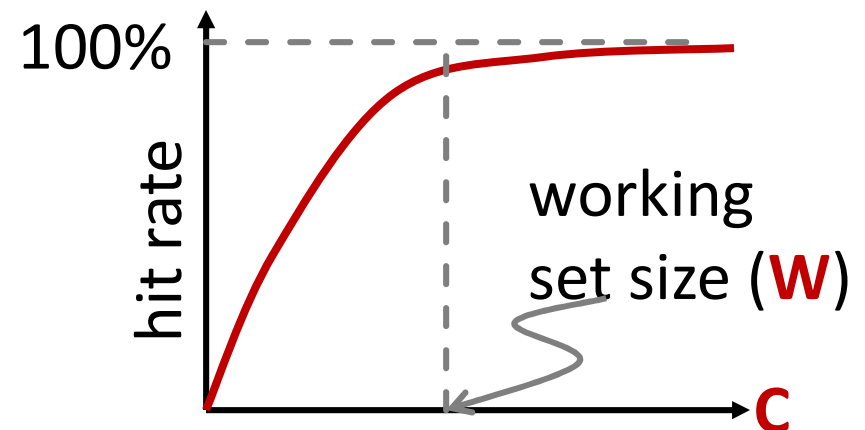
# Compulsory Miss

- First reference to a block address always misses (if no prefetching)
- Dominates when locality is poor
  - for example, in a “streaming” data access pattern where many addresses are visited, but each is used only once
- Main design factor: **B** and “prefetching”



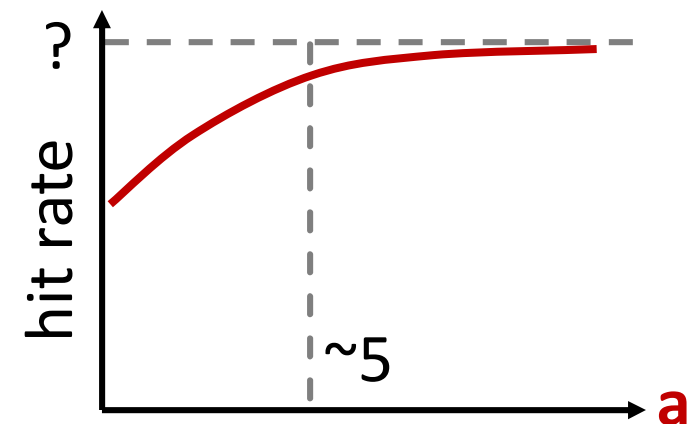
# Capacity Miss

- Cache is too small to hold everything needed
- Defined as the misses that would occur in a fully-associative cache of the same capacity using optimum (Belady) replacement
- Dominates when  $C < W$ 
  - for example, the L1 cache usually not big enough due to cycle-time tradeoff
- Main design factor:  $C$



# Conflict Miss

- Miss to a previously visited block address displaced due to conflict under direct-mapped or set-associative allocation
- Defined as “a miss that is neither compulsory nor capacity”
- Dominates when  $C \approx W$  or when  $C/B$  is small
- Main design factor:  $a$



# 3'C worksheet: **a=1**, **B=1**, **C=2**

addr	set#	which C?	set[2]	F.A. + Belady
0x0	0	compulsory	$[-,-] \rightarrow [0,-]$	$\{\} \rightarrow \{0\}$
0x2	0			
0x0	0			
0x2	0			
0x1	1			
0x0	0			
0x2	0			
0x0	0			

# 3'C worksheet: **a=1, B=1, C=2**

addr	set#	which C?	set[2]	F.A. + Belady
0x0	0	compulsory	$[-,-] \rightarrow [0,-]$	$\{\} \rightarrow \{0\}$
0x2	0	compulsory	$[0,-] \rightarrow [2,-]$	$\{0\} \rightarrow \{0,2\}$
0x0	0	conflict	$[2,-] \rightarrow [0,-]$	$\{0,2\}_{hit}$
0x2	0	conflict	$[0,-] \rightarrow [2,-]$	$\{0,2\}_{hit}$
0x1	1	compulsory	$[2,-] \rightarrow [2,1]$	$\{0,2\} \rightarrow \{0,1\}$
0x0	0	conflict	$[2,1] \rightarrow [0,1]$	$\{0,1\}_{hit}$
0x2	0	capacity	$[0,1] \rightarrow [2,1]$	$\{0,1\} \rightarrow \{0,2\}$
0x0	0	conflict	$[2,1] \rightarrow [0,1]$	$\{0,2\}_{hit}$

# Recap: Basic Cache Parameters

ISA

- **$M = 2^m$**  : size of address space in bytes  
sample values:  $2^{32}$ ,  $2^{64}$
- **$G=2^g$**  : cache access granularity in bytes  
sample values: 4, 8

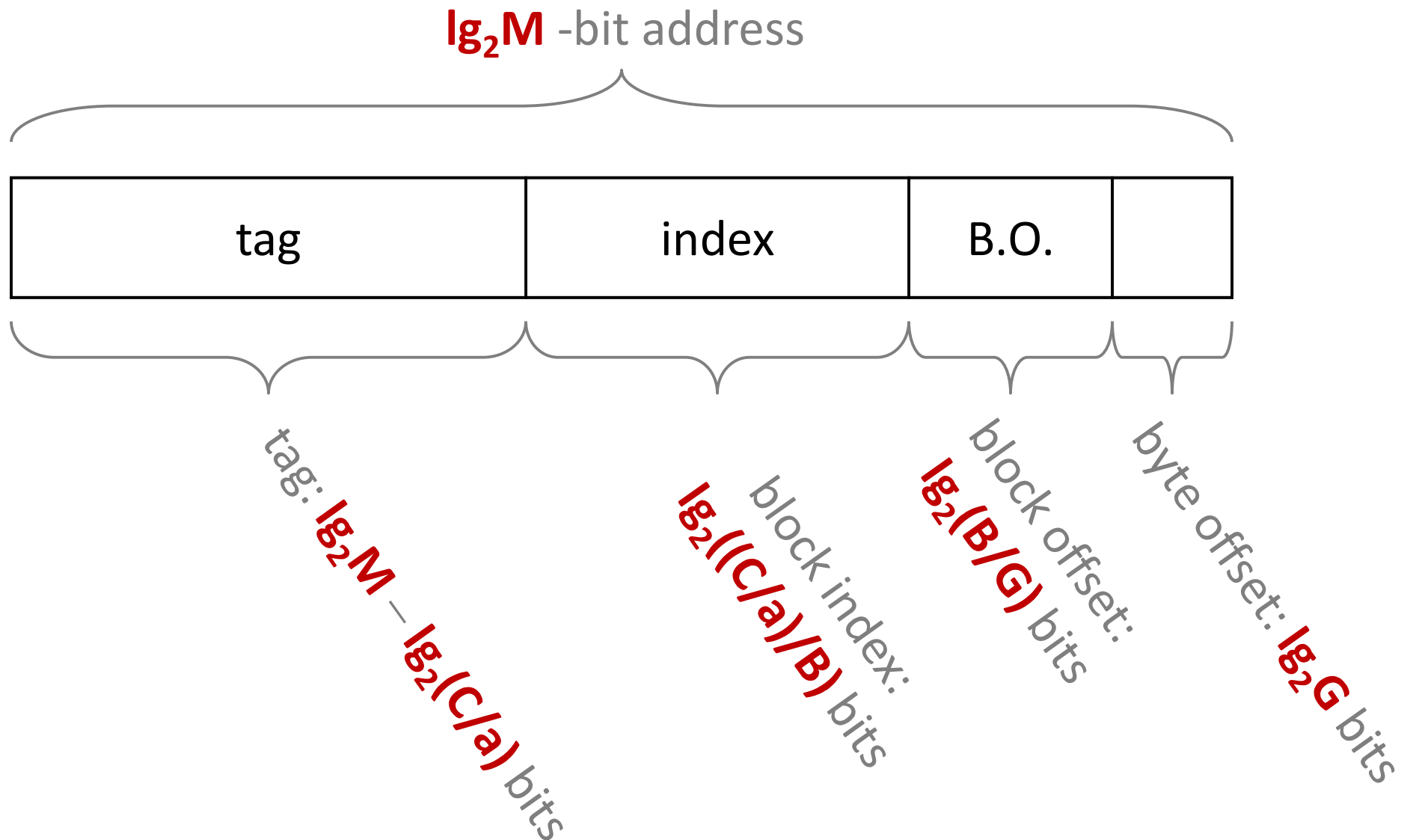
Implementation

- 
- **$C$**  : “capacity” of cache in bytes  
sample values: 16 KByte (L1), 1 MByte (L2)
  - **$B = 2^b$** : “block size” in bytes  
sample values: 16 (L1), >64 (L2)
  - **$a$** : “associativity” of the cache  
sample values: 1, 2, 4, 5(?),... “C/B”

C/a should be a 2-power

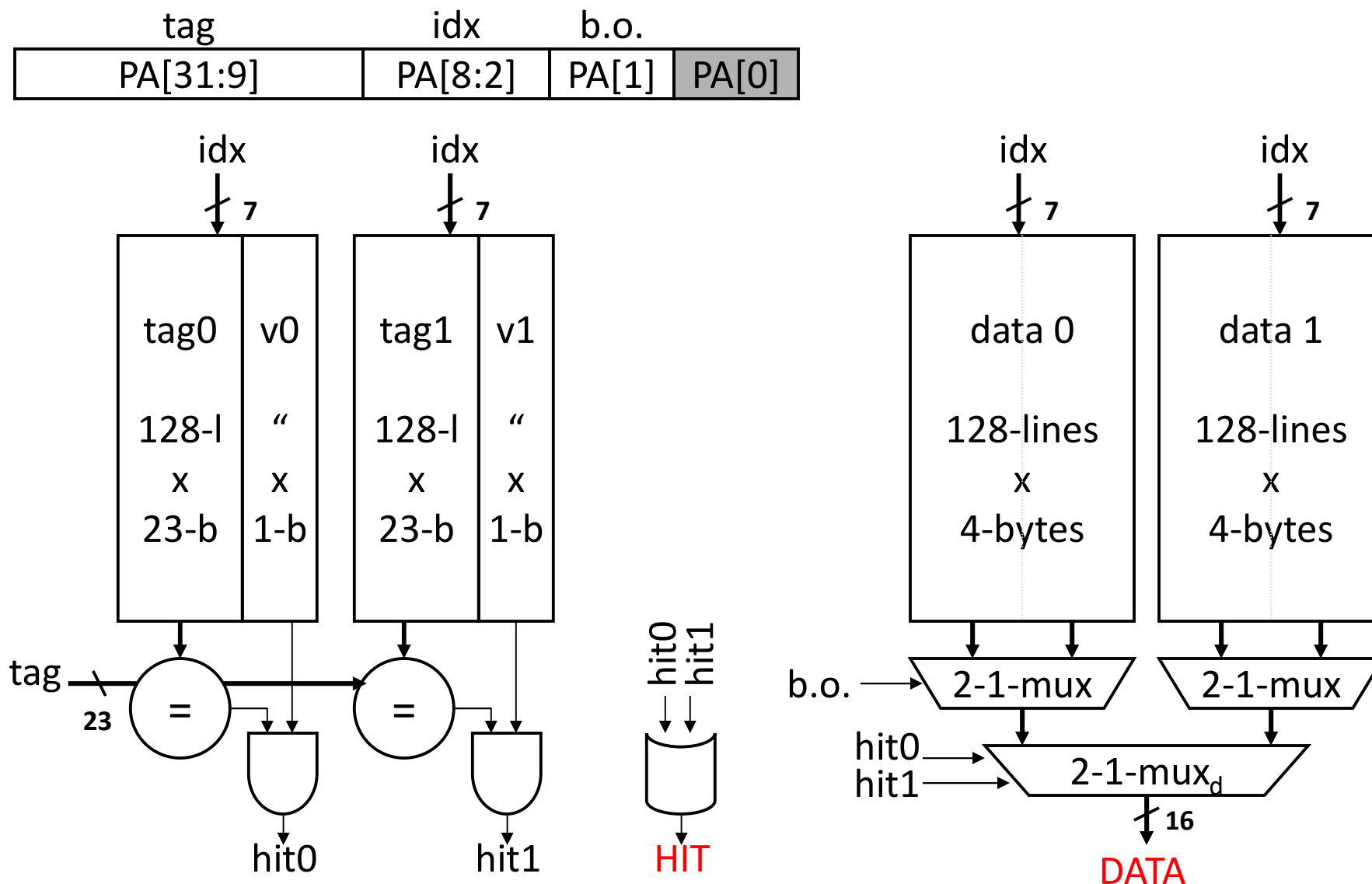


# Recap: Address Fields

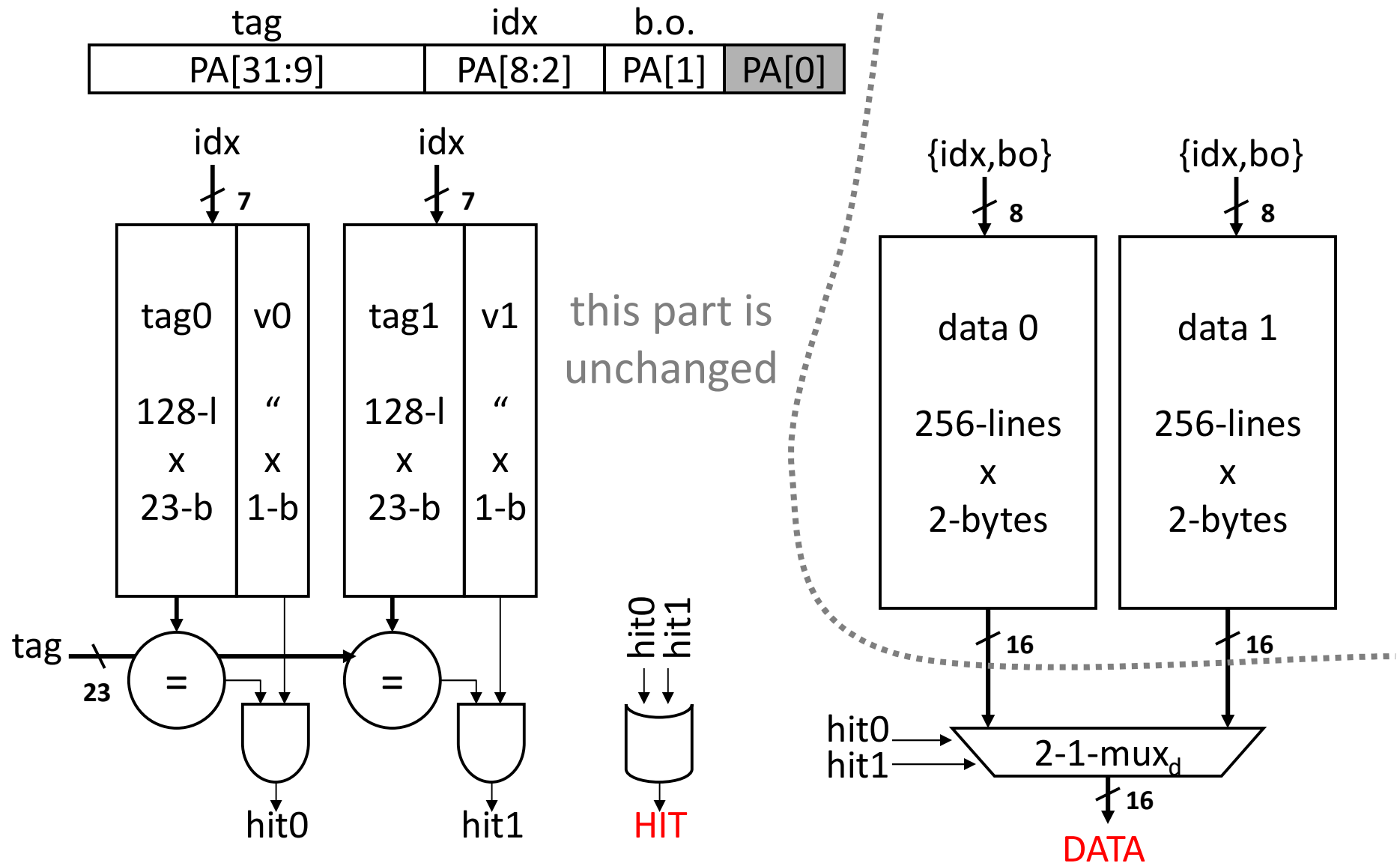


**M=2<sup>32</sup>, a=2, C=1K, B=4, G=2**

# $M=2^{32}$ , $a=2$ , $C=1K$ , $B=4$ , $G=2$ : basic solution

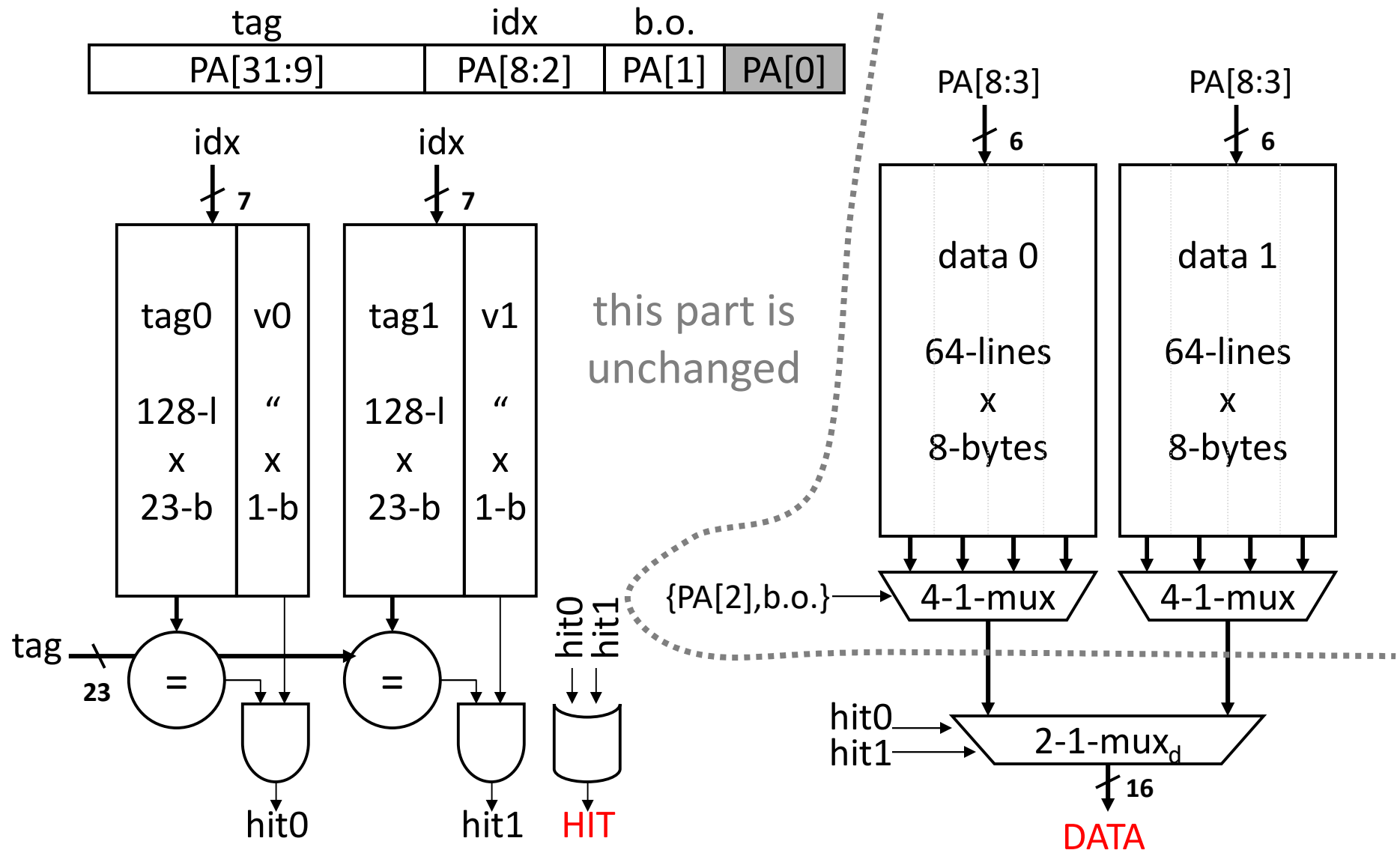


## Same cache parameters but tune for “narrower” data SRAMs



Can you play the same trick on the tag SRAMs?

## Same cache parameters but tune for “fatter” data SRAMs



Can you play the same trick on the tag SRAMs?

# Same cache parameters but each block frame is interleaved over 2 SRAM banks

