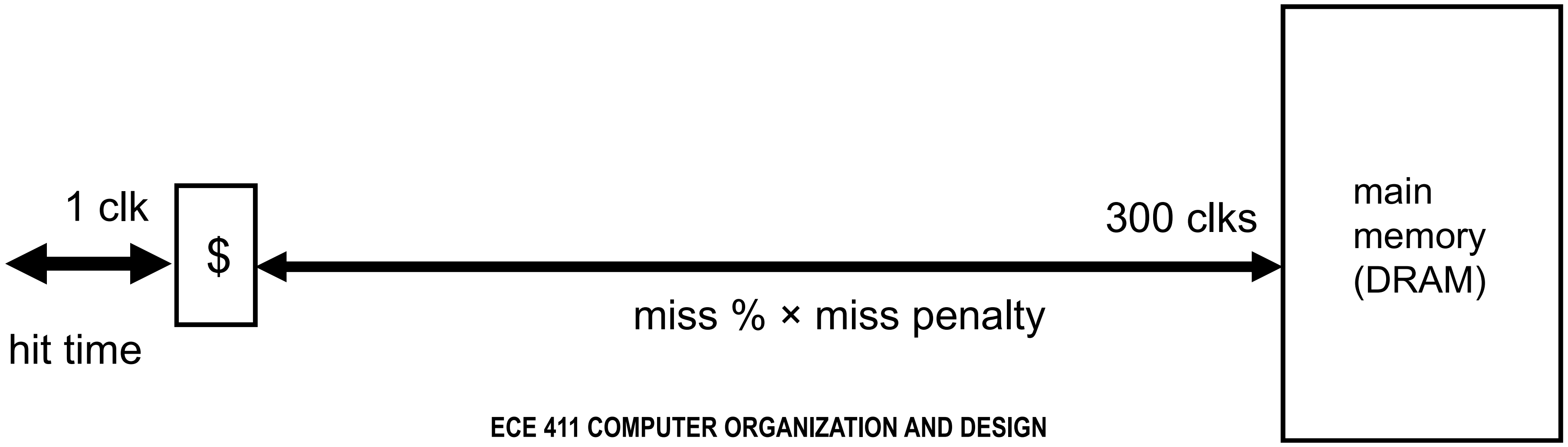
Lecture 6:
Memory Hierarchy (cnt'd)

Review: Memory Hierarchy Performance

- Average Memory Access Time (AMAT)
  - ✓ = hit time + miss rate × miss penalty
  - ✓ = T_{hit}(L1) + miss%(L1) × T(memory)
- example:
  - ✓ cache hit = 1 cycle
  - ✓ miss rate = 10% = 0.1
  - ✓ miss penalty = 300 cycles
  - ✓ AMAT = 1 + 0.1 × 300 = 31 cycles
- can we improve it?



Review: Reducing Penalty: Multi-Level Cache

- Average Memory Access Time (AMAT)

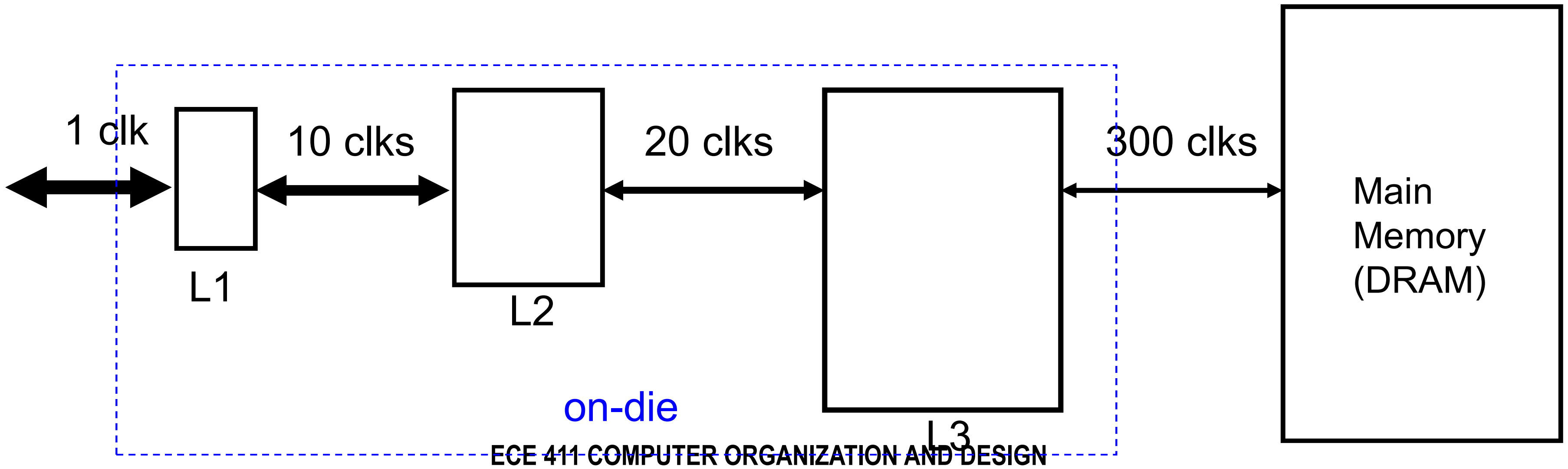
  $= T_{hit}(L1) + miss\%(L1)\times(T_{hit}(L2) + miss\%(L2)\times(T_{hit}(L3) + miss\%(L3)\times T(memory)\ )\ )$

  $= T_{hit}(L1) + miss\%(L1)\times T_{miss}(L1)$

  $= T_{hit}(L1) + miss\%(L1)\times\{T_{hit}(L2) + miss\%(L2)\times(T_{miss}(L2)\ \}$

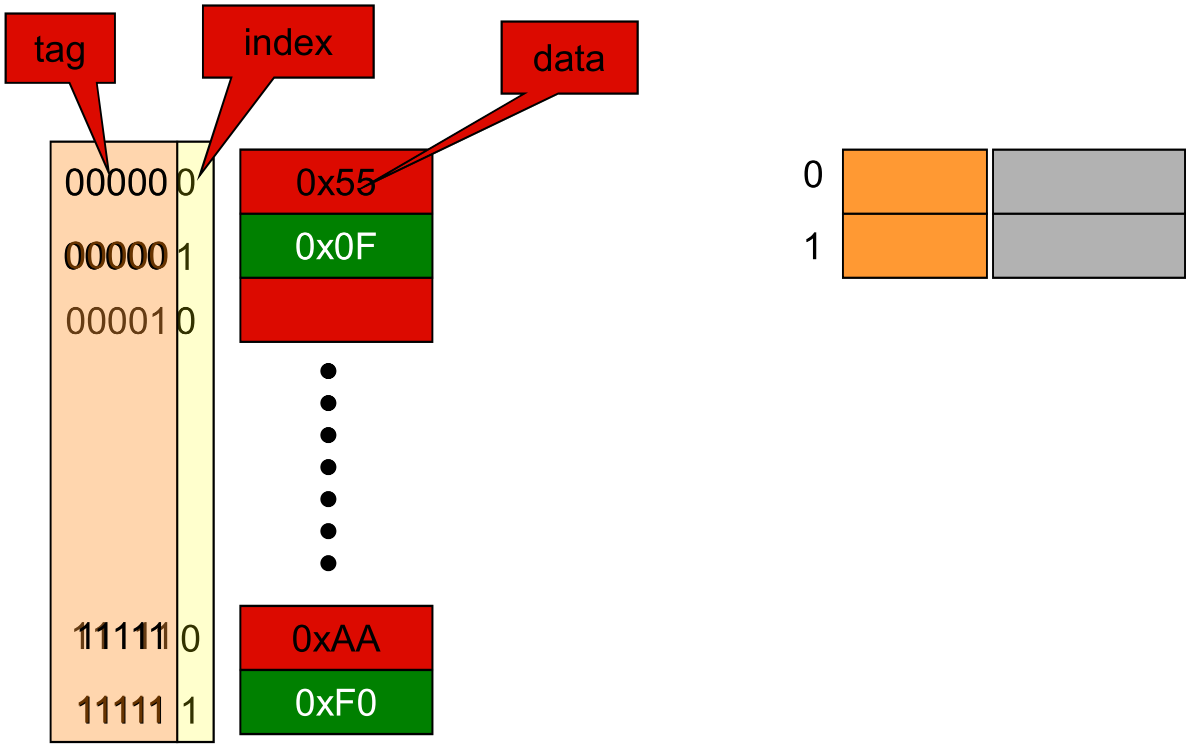
  $= T_{hit}(L1) + miss\%(L1)\times\{T_{hit}(L2) + miss\%(L2)\times(T_{miss}(L2)\ \}$

  $= T_{hit}(L1) + miss\%(L1)\times\{T_{hit}(L2) + miss\%(L2)\times[\ T_{hit}(L3) + miss\%(L3)\times T(memory)\ ]\ \}$



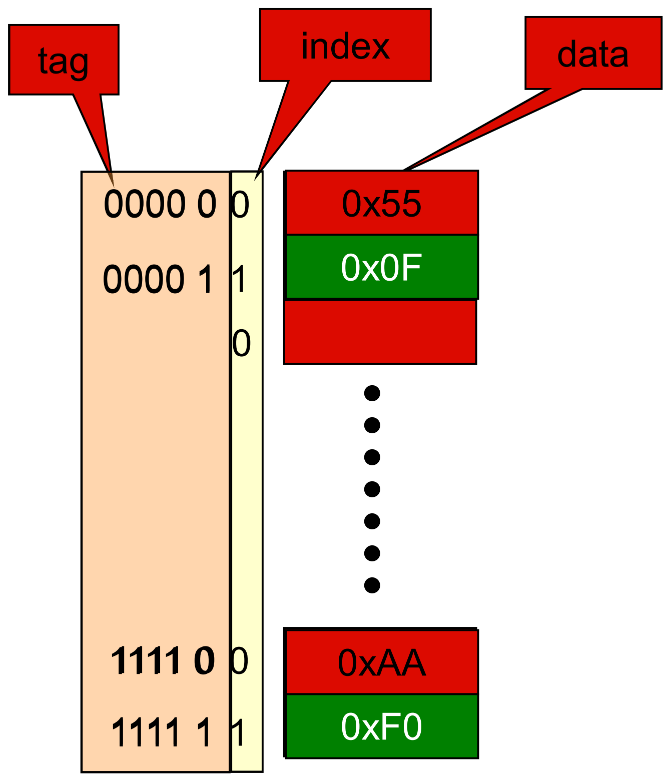
Review: Direct Mapping

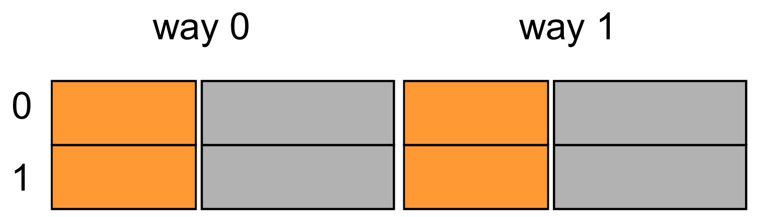
- direct mapping:
  - ✓ a memory value can only be placed at a single corresponding location in the cache



Review: Set Associative Mapping (2-Way)

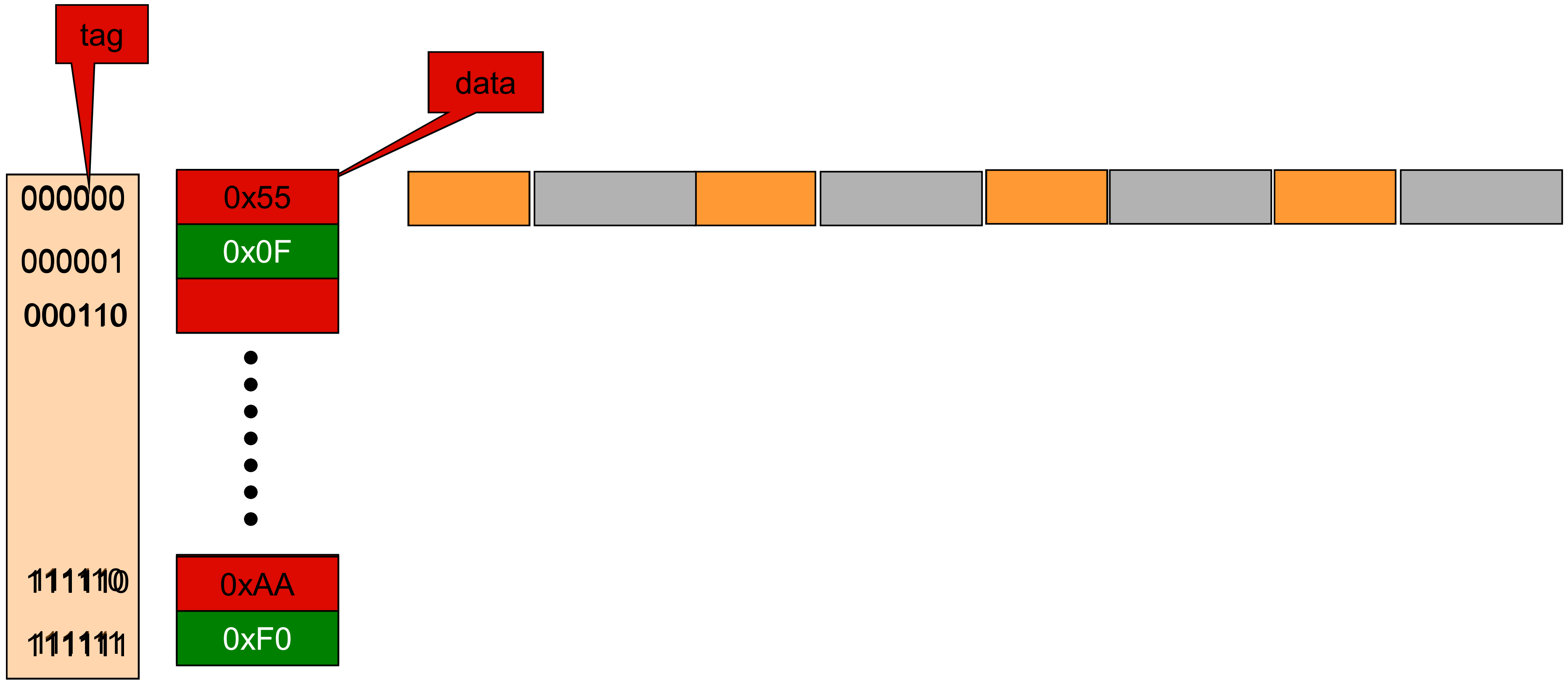
- set-associative mapping:
  - ✓ a memory value can be placed in any location of a set in the cache





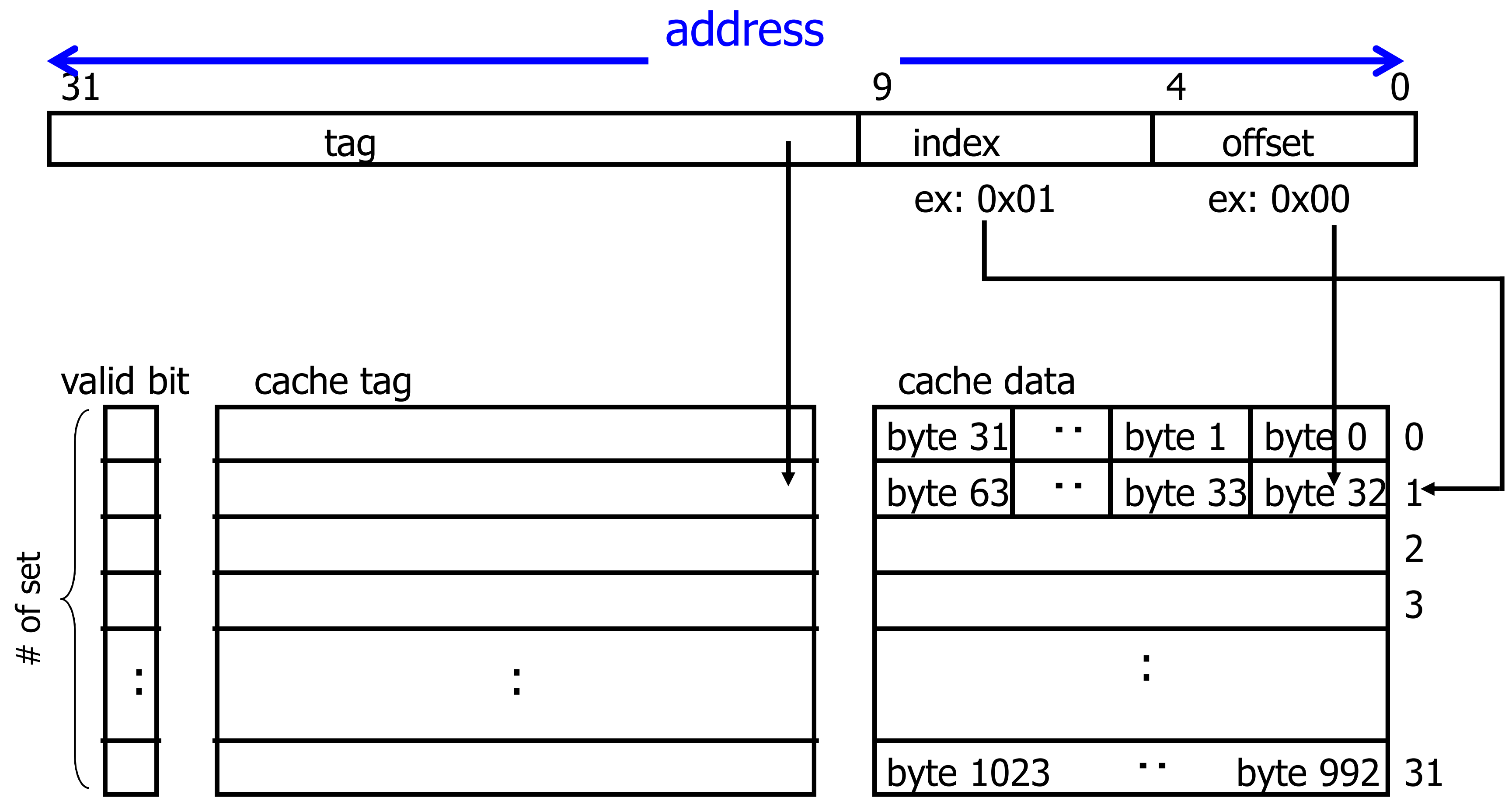
Review: Fully Associative Mapping

- fully-associative mapping:
  - ✓ a memory value can be placed anywhere in the cache



Review Example: 1KB DM Cache, 32-byte Lines

- lowest M bits are offset (Line Size = 2M)
- index = log2 (# of sets)



Review: Example of Caches

- given a 2MB, direct-mapped physical caches, line size=64bytes, and 52-bit physical address
  - ✓ tag size?
  - ✓ now change it to 16-way, tag size?
  - ✓ how about if it's fully associative, tag size?

Cache Architecture Supplementary Material

- I created a playlist linking a series of short youtube videos related to cache
  - ✓ https://www.youtube.com/playlist?list=PLwuMMeQzE5C390O5jmySV8wYP1iqFwBlx
- another lecture slide deck on cache architecture from CMU
  - ✓ www.cs.cmu.edu/afs/cs/academic/class/15213-f08/www/lectures/lecture-10.ppt

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 ← later✓
  - ✓ use similar policies but spend more effort to improve hit rate

LRU: Precise Tracking

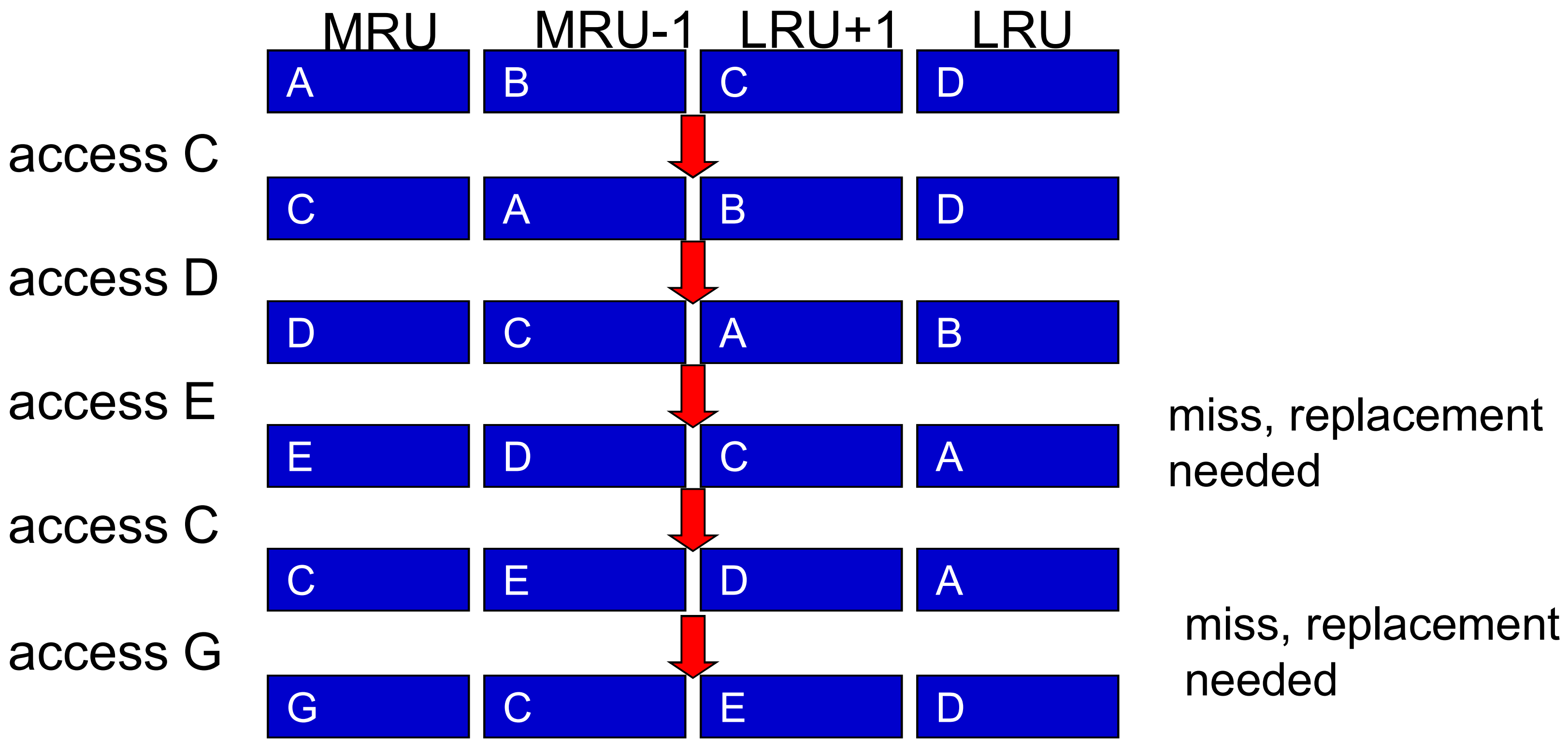
- precise LRU tracking requires a stack for each set
  1. when a block is accessed by the processor, check if its name is in the stack
  2. if so, remove it from the stack
  3. push the name of block at the top of the stack
  4. position (depth) of a block name in the stack gives the relative recency of access to this block compared to others

✓ for a 4-way set associative cache with blocks A, B, C, D.

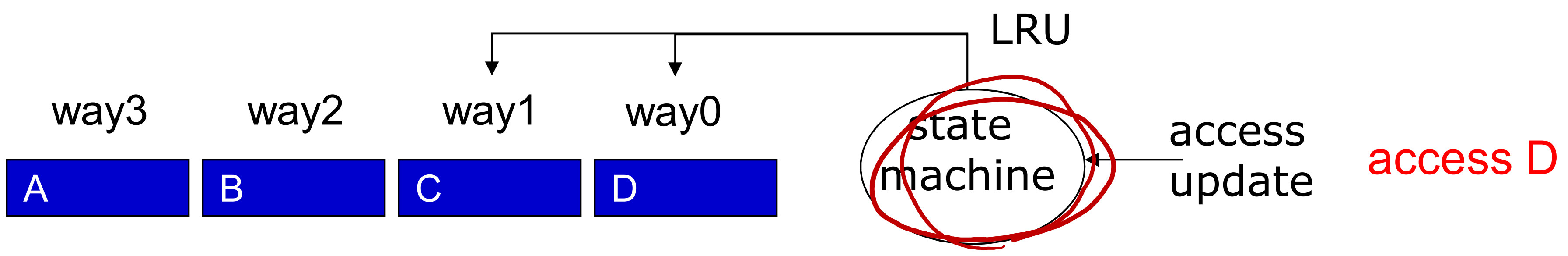
- assume a sequence of accesses C, D, A, B, A, C, B, D
- assume that the stack is initially empty, the configuration of the stack after each access would be [C,-,-,-], [D, C,-,-], [A, D, C,-], [B, A, D, C], [A, B, D, C], [C, A, B, D], [B, C, A, D], [D, B, C, A]
- one on the right most position (bottom 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)

2×4

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



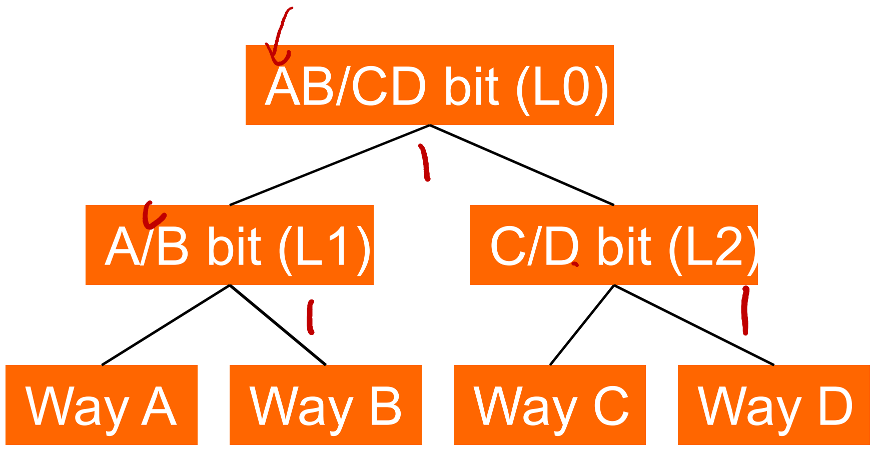
LRU policy increases cache access times as additional hardware bits needed for LRU state machine

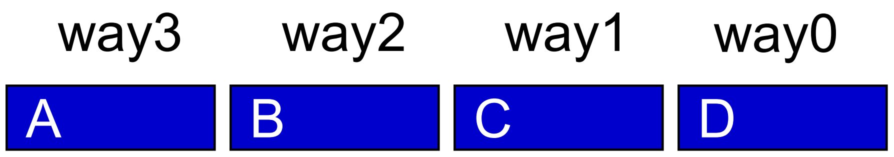
LRU: Approximate Tracking (Pseudo-LRU)

- 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
    - 1st bit tracks the more recently used half
    - 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

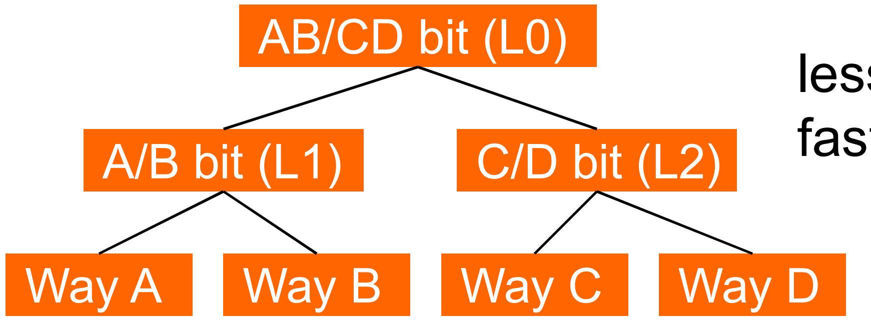
Pseudo LRU Algorithm (4-way SA)

- 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).

$\nearrow$ more examples you can try

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 w/ 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

1. 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.
2. repeat part (a) for a fully-associative cache that uses the LRU-replacement algorithm.
3. 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 256 byte 2-way set associative write-back cache w/ 16 byte blocks, LRU replacement and write-back policies
  - ✓ assume that cache starts empty
  - ✓ 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

You should complete the answer

64 32 16 8 4 2 1

0 0 0 0 0 0 0 0 0

1 0 0 | 0 0 0 0

16 byte blocks, 4 LSB byte offset

0 1 0

064 032 064 000 112 064 128 048 240 000

update

there are ~~4~~ 8 sets with 2 blocks each

16 | 256

16

16

V D tag data	V D tag data

set ~~block~~ addr.	4	2	4	0	7	4	8			
R/W	R	R	R	R	W	W	R	R	R	W
set#	0	0	0	0	0					
tag										
H/M	M	M	H	M	M	H				
WB?										

You should complete the answer

16 byte blocks, 4 LSB byte offset

064 032 064 000 112 064 128 048 240 000

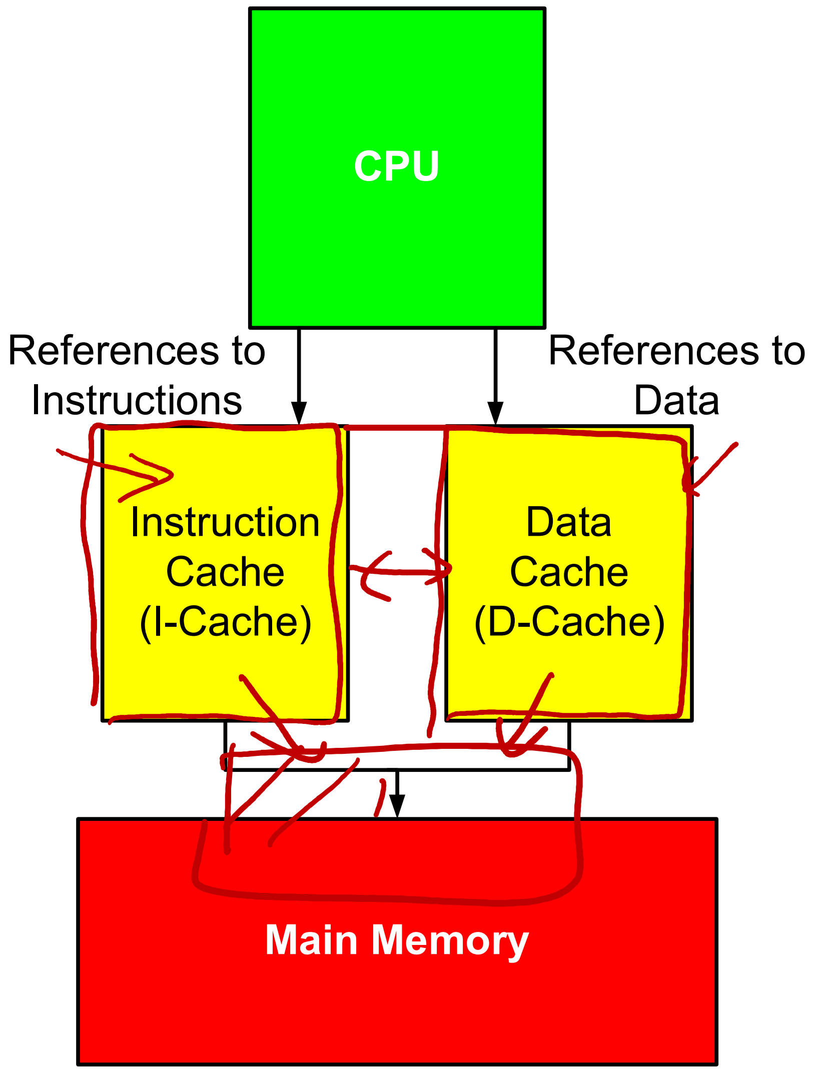
V D tag data	V D tag data

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#										
tag										
H/M										
WB?										

Instruction and Data Caches

L1

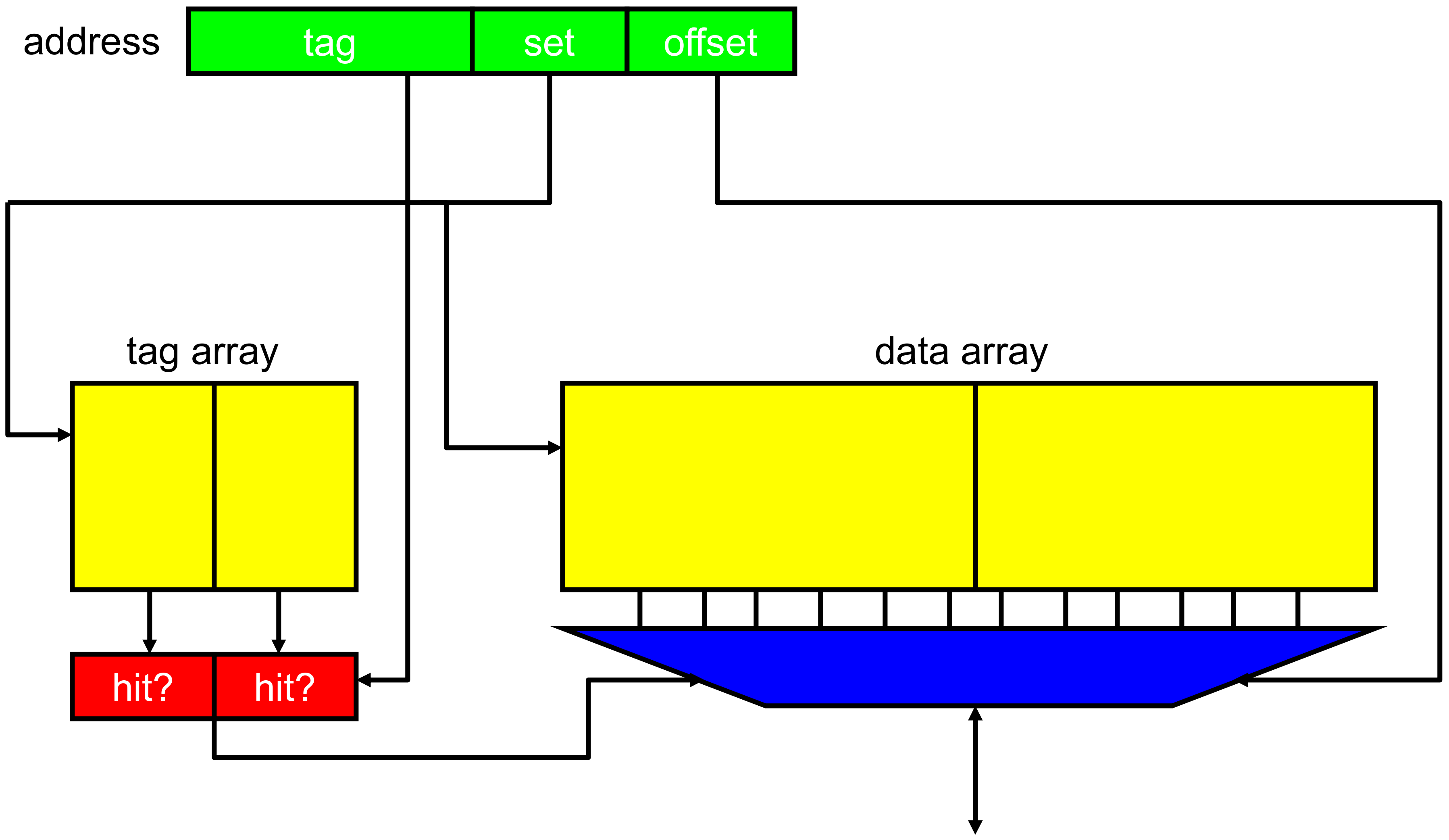


Cache Performance Example

- given
  - ✓ I-cache miss rate = 2%
  - ✓ D-cache miss rate = 4%
  - ✓ Miss penalty = 100 cycles
  - ✓ Base CPI (ideal cache) = 2
  - ✓ Load & stores are 36% of instructions
- what is the actual CPI?

I will go over next week

Microarchitecture of Cache Memories



Why This Organization?

- allows tag array to be faster than data array
  - ✓ tag array is smaller
- don’t really need output of data array until hit/miss detection complete
- overlap some of data array access time with hit/miss detection
- also integrates well with virtual memory, as we’ll see

Announcement

- today's lecture: more cache
  - ✓ Ch. 5.4 – 5.5 (HP1)
- next lecture: other cache topics (e.g., replacement policy)
  - ✓ Ch. 5.6 – 5.7 (HP1)
- MP assignment
  - ✓ MP1 due on 2/4 5pm