Module 12 – Cache

(ECE M116C- CS M151B) Computer Architecture Systems

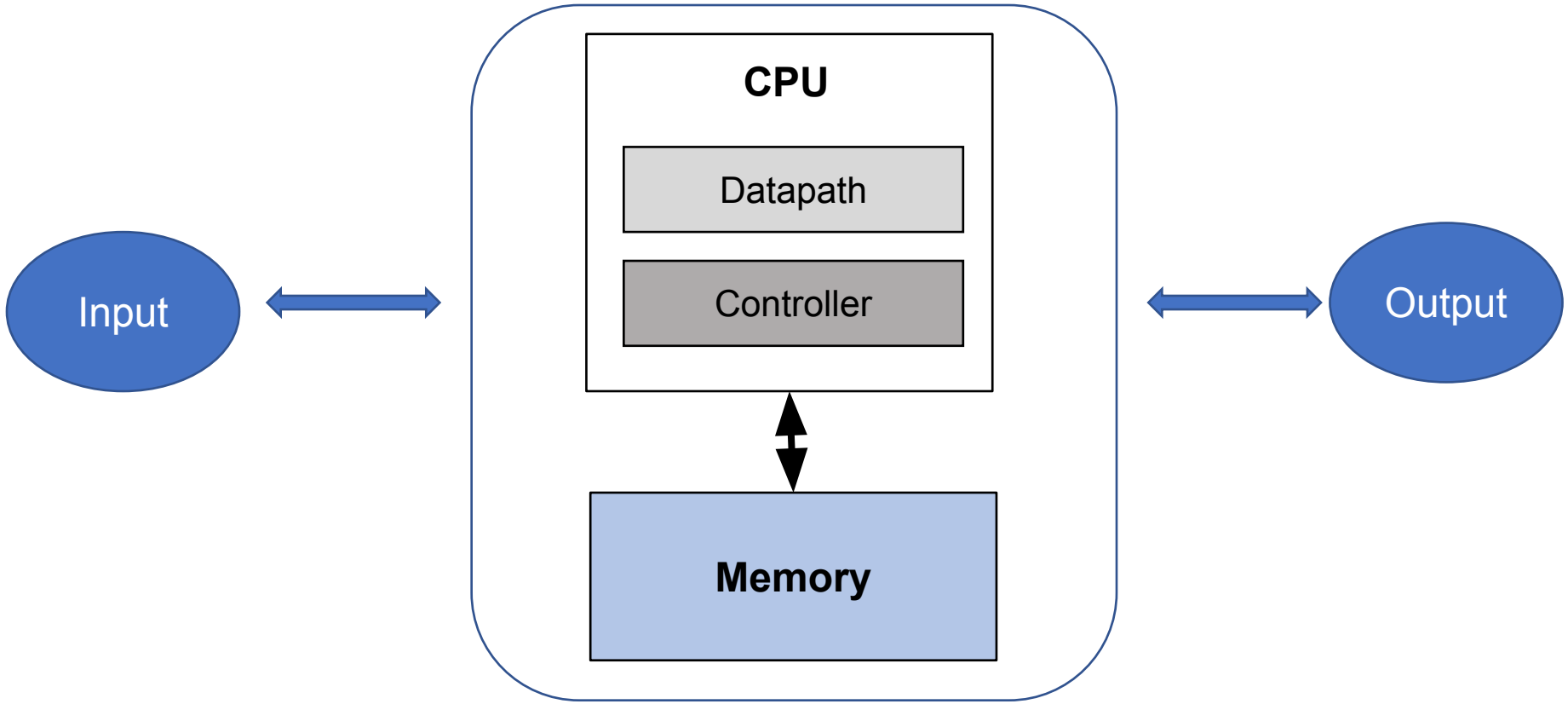
Nader Sehatbakhsh
Department of Electrical and Computer Engineering
University of California, Los Angeles

What's next?

- It's time to move on from the processor …

Memory Hierarchy

Von-Neumann Model



Memory "Wall"

- CPU is much faster than memory.
  - Accessing DRAM takes 1000s of cycles in modern processors.
- CPU speed is ***growing*** much faster
  - The performance gap between memory and CPU is growing by around 50% per year.

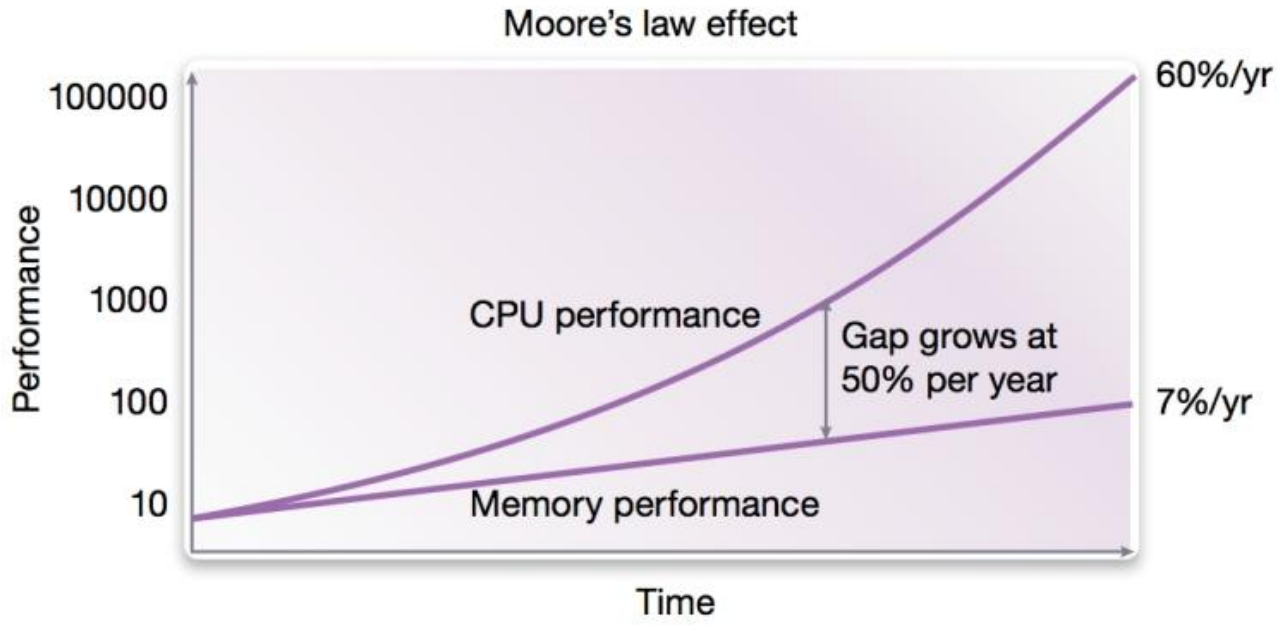


Image taken from:
https://www.extremetech.com/computing/233691-phase-change-memory-can-operate-thousands-of-times-faster-than-current-ram

Fundamental Challenge

- Memories can be either large OR fast

Different Storage (Memory)

- Latches and Flip-Flops (aka Registers)
  - Very fast, parallel access.
  - Very expensive (one bit costs ***tens*** of transistors).
- Static RAM (SRAM)
  - Relatively fast, only one data word at a time.
  - Expensive (one bit costs **6+** transistors).
- Dynamic RAM (DRAM)
  - Slower, one data word at a time, reading *destroys* content (refresh), needs special process for manufacturing.
  - Cheap (one bit costs only ***one*** transistor plus one capacitor).
- DISK (*flash memory*, *hard disk*)
  - Much slower, access takes a long time, ***non-volatile.***
  - Very cheap.

Size and Speed?

- Register File
  - ***SIZE:*** a few KB
  - ***ACCESS TIME:*** within a cycle (<1ns)
- SRAM
  - ***SIZE:*** 10s-100s kilobytes
  - ***ACCESS TIME:*** 2-5 to 10s cycles (1-3/10s ns)
- DRAM
  - ***SIZE:*** 1-10s gigabytes
  - ***ACCESS TIME:*** 500-1000s cycles (>100ns)
- DISK
  - ***SIZE:*** 1-10s terabytes
  - ***ACCESS TIME:*** 10k-100k cycles (>10ms)

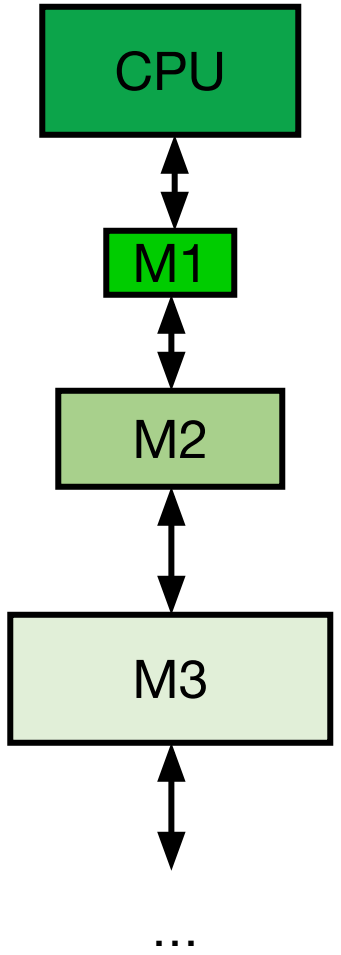
Fundamental Challenge

- Memories can be either large OR fast
- *But we need both fast and large memories.*

Typical programs need 10s KB – 10s MB of memory.

Memory Hierarchy

★ ***Idea:*** *have multiple layers of memory to balance between size and speed!*



Why hierarchy would work?

Memory Hierarchy

Library Analogy

- Consider books as data.

Memory Hierarchy

Library Analogy

- Consider books as data.
  - The *currently* used book is at your hand.
  - Frequently-used books (e.g., CompArch book, book for your other classes) are on your desk.

Memory Hierarchy

Library Analogy

- Consider books as data.
  - The *currently* used book is at your hand.
  - Frequently-used books (e.g., CompArch book, book for your other classes) are on your desk.

★ *But you can't keep all the books on the desk.*

Memory Hierarchy
Library Analogy

- Consider books as data.
  - The *currently* used book is at your hand.
  - Frequently-used books (e.g., CompArch book, book for your other classes) are on your desk.
  - Less recently-used books (but still important) are stored in your room in the shelf.

★ *Shelf space is also limited*

Memory Hierarchy
Library Analogy

- Consider books as data.
  - The *currently* used book is at your hand.
  - Frequently-used books (e.g., CompArch book, book for your other classes) are on your desk.
  - Less recently-used books (but still important) are stored in your room in the shelf.
  - Rarely-used books are in the garage/attic/box.

Memory Hierarchy

Library Analogy

- Consider books as data.
  - The *currently* used book is at your hand.
  - Frequently-used books (e.g., CompArch book, book for your other classes) are on your desk.
  - Less recently-used books (but still important) are stored in your room in the shelf.
  - Rarely-used books are in the garage/attic/box.
  - Very rarely-used books are in the storage.
  - …

Why hierarchy would work?

- ***Locality!***
  - Programs access a ***small*** proportion of their address space at any time

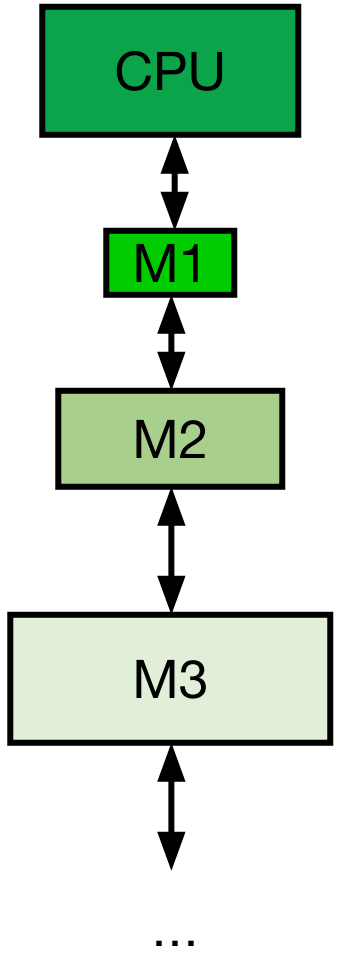
Why hierarchy would work?

- ***Temporal locality***
  - Items accessed recently are likely to be accessed again soon.
  - *Examples: instructions in a loop, induction variables.*

- ***Spatial locality***
  - Items *near* those accessed recently are likely to be accessed soon.
  - *Examples: sequential instruction access, array data.*

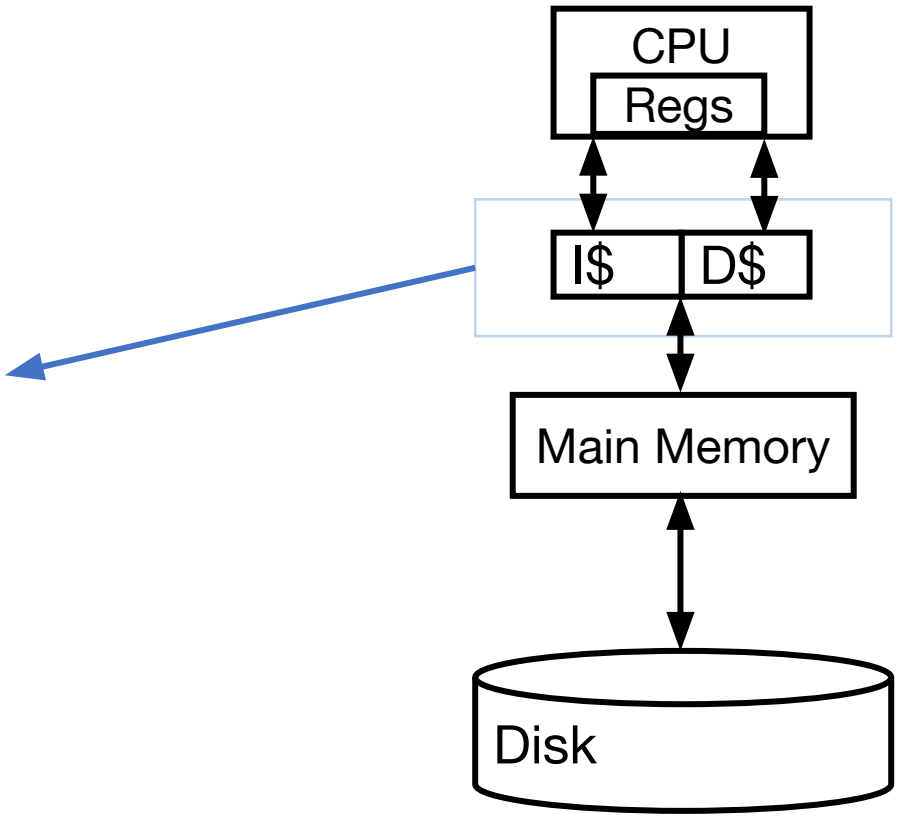
Memory Hierarchy

★ *Bring frequently used data closer.*



Cache

Storing temporary data close to the CPU.



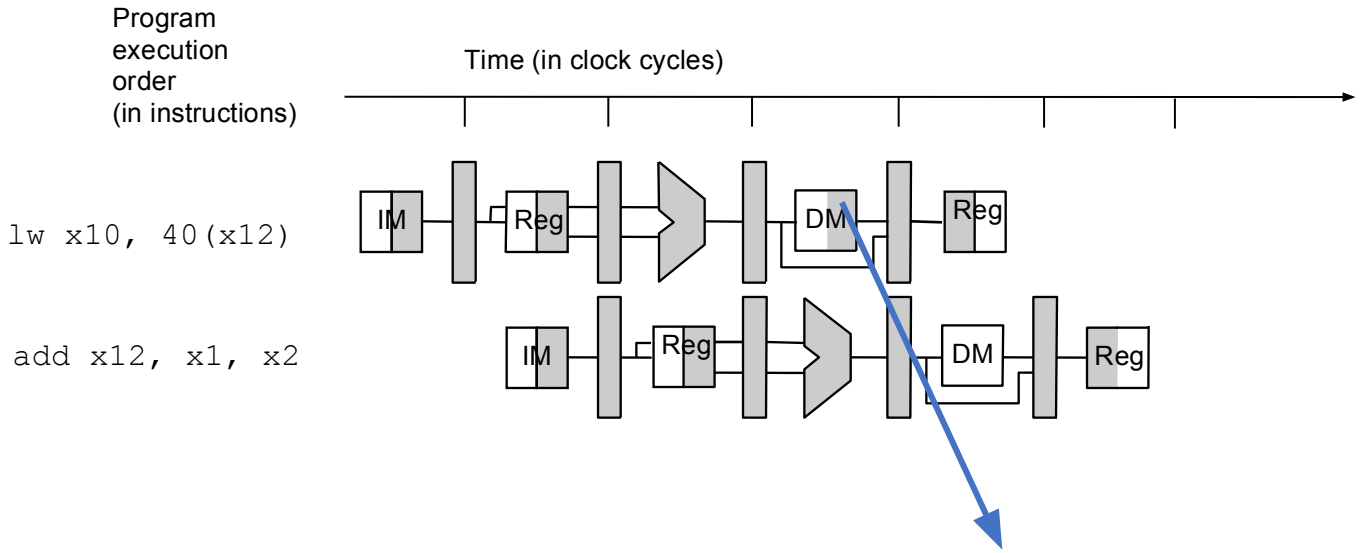
Cache Design

- If the requested data exists in the cache, we call it a ***cache hit***.

- If the requested data does not exist in the cache, it is a ***cache miss***.
  - We need to bring it from the main memory and put it in the cache.

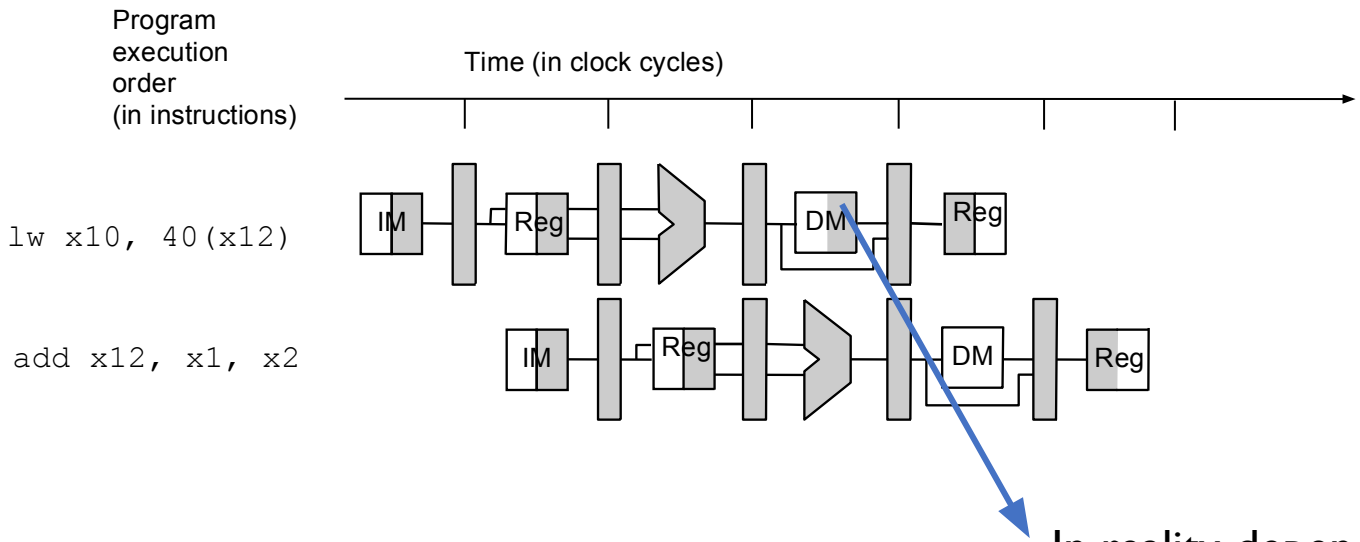
Why cache is so important for performance?

Memory Delay



Ideal: Accessing DM takes ***one*** cycle.

Memory Delay



In reality, depending on where the data is, it may take ***1-1000*** cycles!! ***– stalling***

If in L1 cache? 1 cycle
Else if in L2? 20 cycles – stall the processor!
Else if in L3? 100 cycles – stall the processor!
Else …

Cache Design

- ***Goal:*** *minimizing the access to the main memory!*

  (i.e., minimizing stalls!)

Cache Performance

- Average time to access the memory:

  AAT = HitTime + MissRate × MissPenalty

  - HitTime: Time it takes to access the (L1) cache.
  - MissRate: The average frequency of misses (in L1).
  - MissPenalty: The time required to access the main memory.

Cache Design

- ***Goal:*** *minimizing the access to the main memory!*

- ***The Problem:*** *limited space!*

Cache Design

- ***Goal:*** *minimizing the access to the main memory!*
- ***The Problem:*** *limited space!*
- ***Solution:*** *Cache management!*

Where to store/find data?

Where to store/find data?

- *What do we have?*
  - *Address* (32 bit)

- *What do we need?*
  1. Whether this address exists in the cache.
  2. The value/content.

Ideal Scenario

- How many lines does the memory have?

→ Assuming a 32-bit PC, each unique value of PC is one line in memory → 4 billion lines of memory!

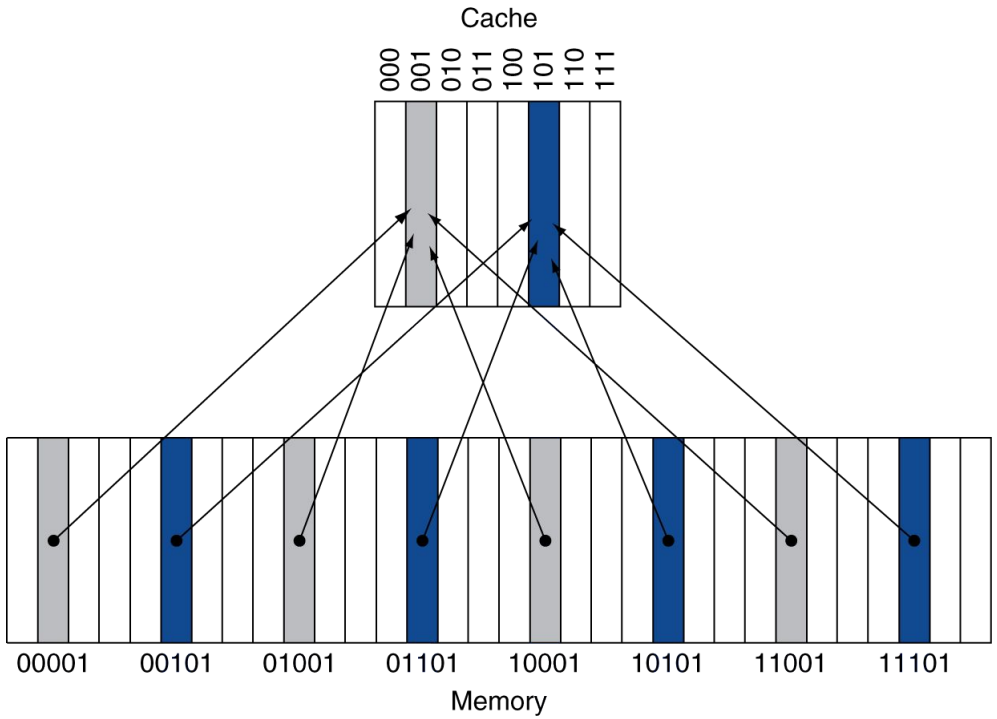
Where to store/find data?

- *What do we have?*
  - *Address* (32 bit)

- *What do we need?*
  1. Whether this address exists in the cache. → if we had all the lines, there was no need for this!
  2. The value/content.

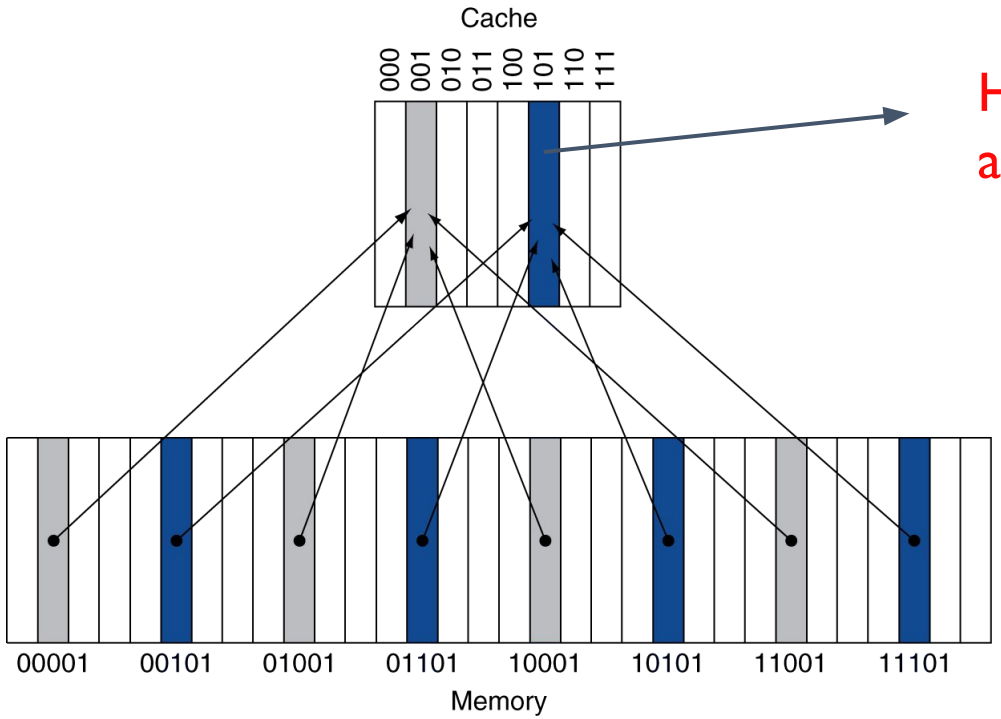
Reality: Cache << Memory

- ***One*** cache row is assigned to ***many*** memory lines.



Reality: Cache << Memory

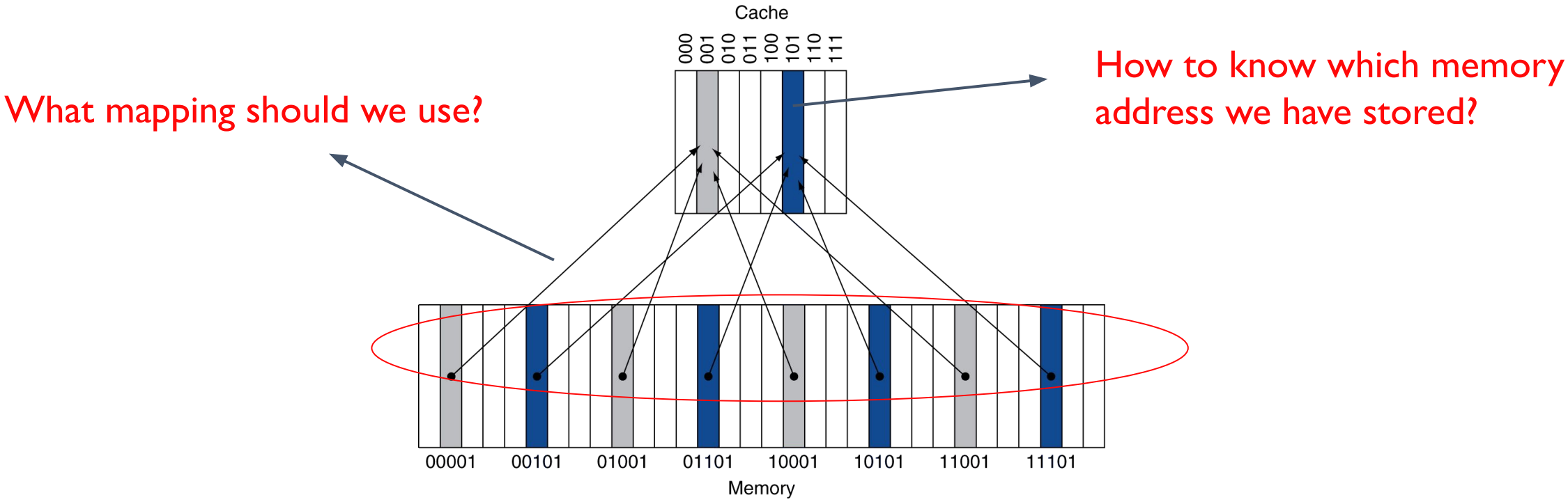
- ***One*** cache row is assigned to ***many*** memory lines.



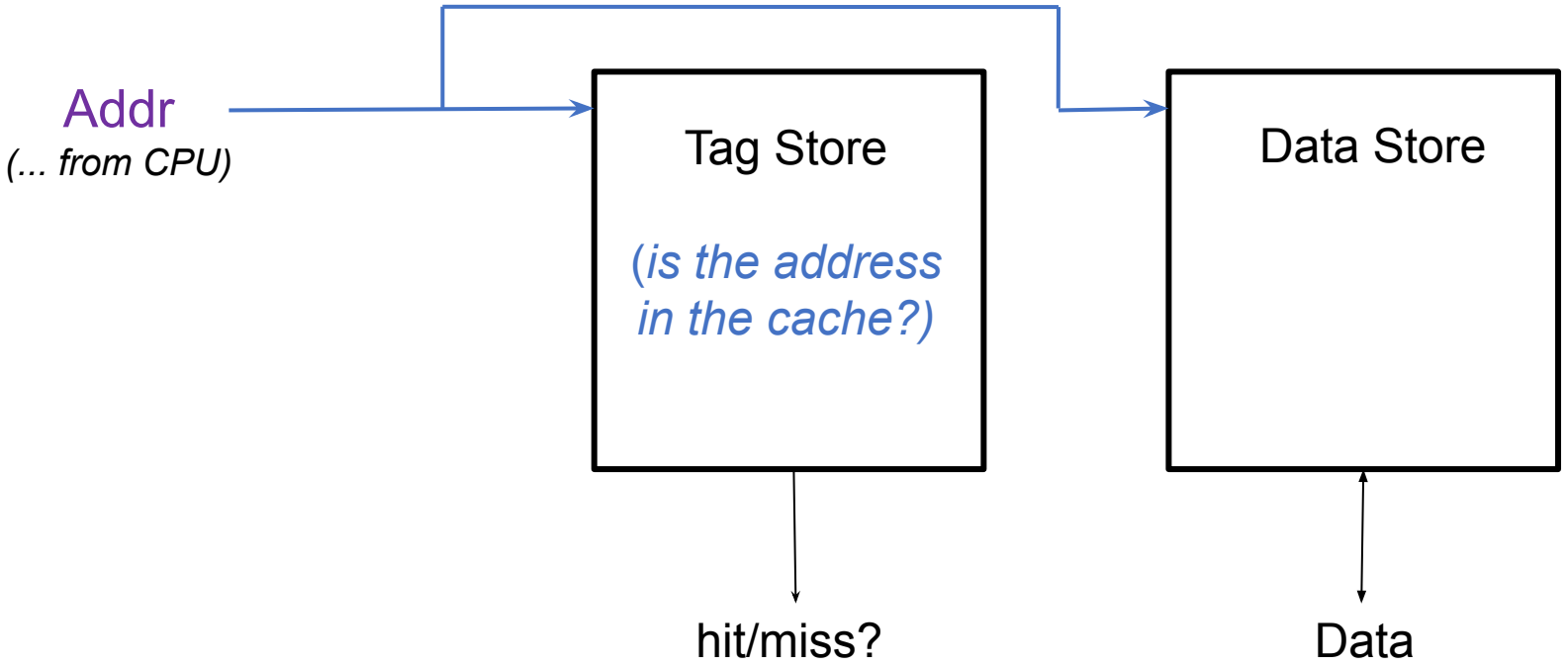
How to know which memory address we have stored?

Reality: Cache << Memory

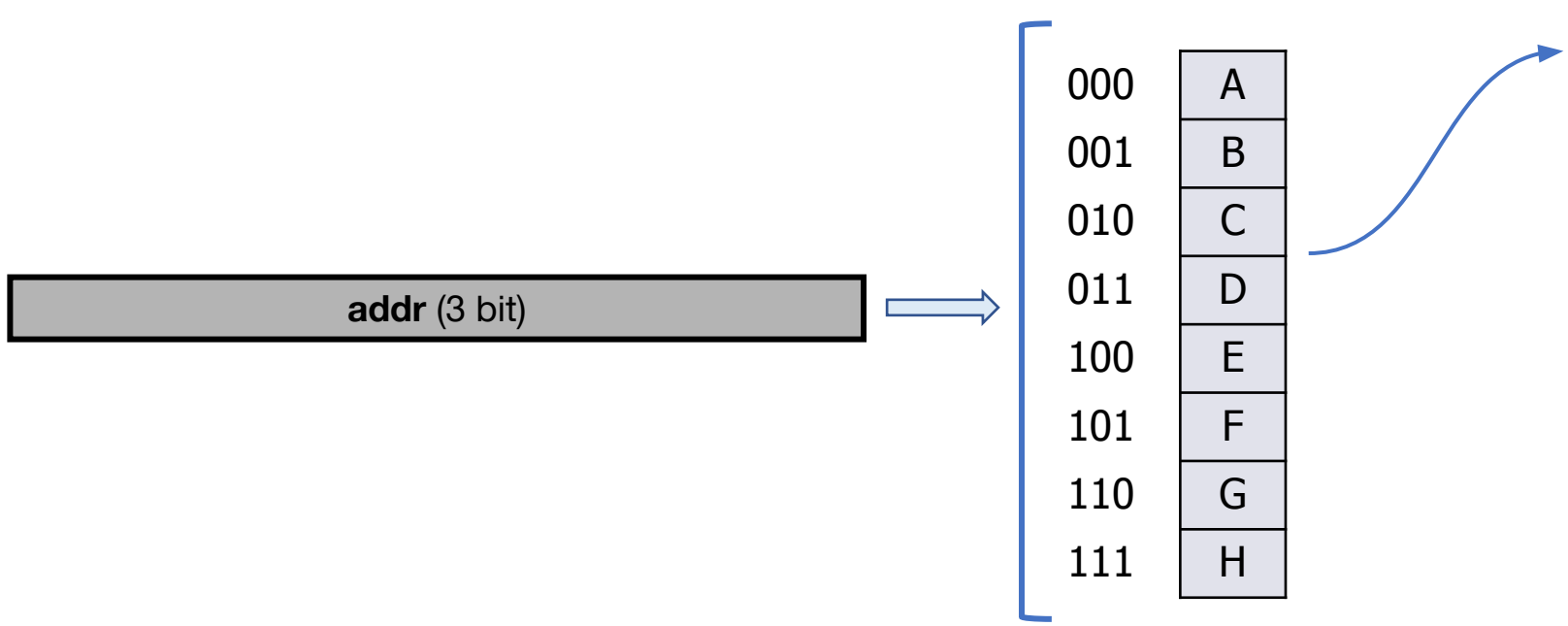
- ***One*** cache row is assigned to ***many*** memory lines.



Where to store/find data?

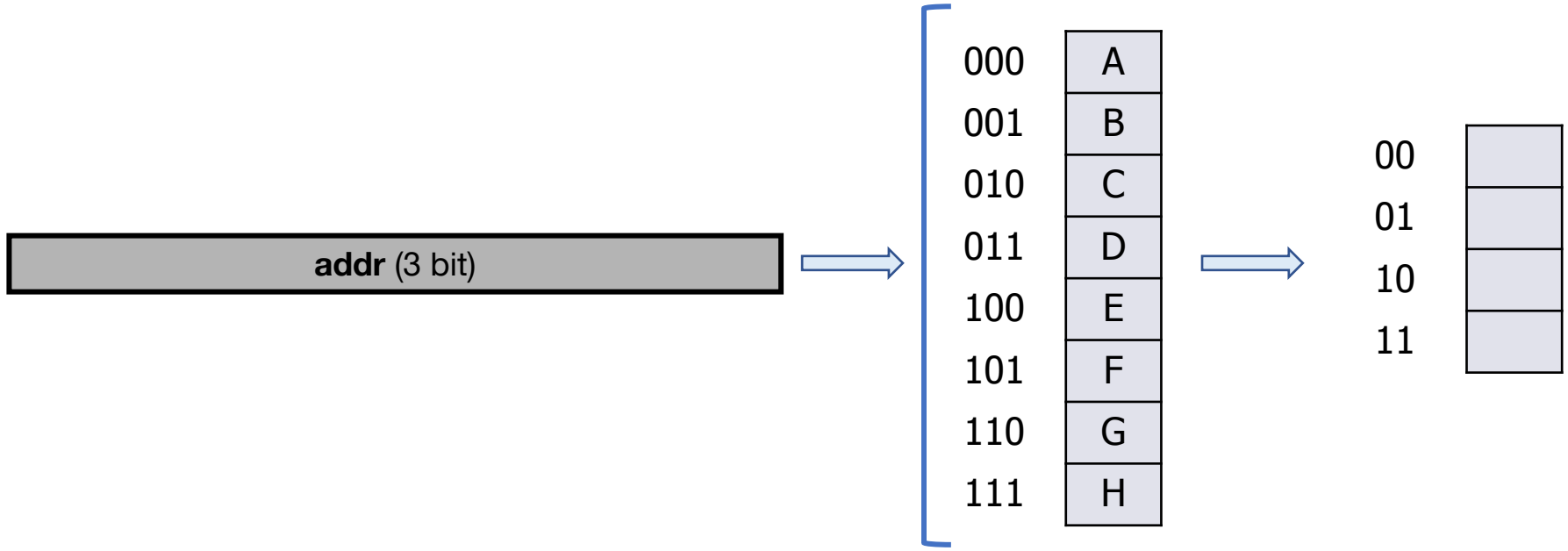


What should we store as the tag?

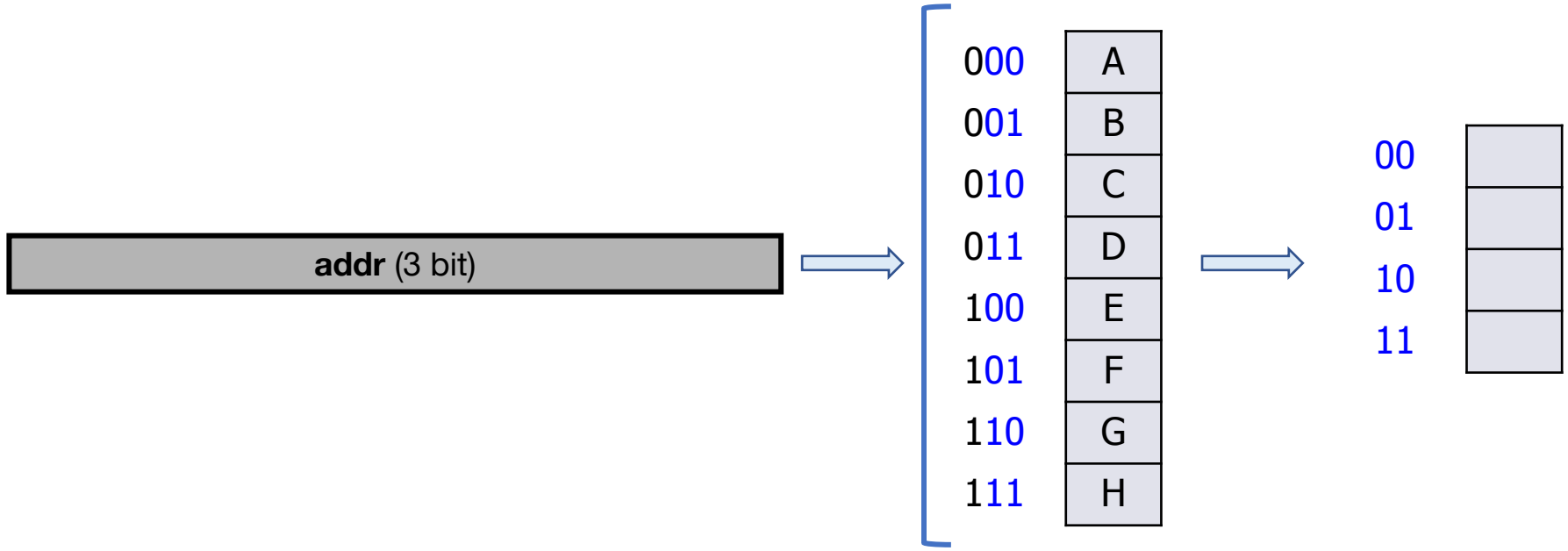


In an ideal scenario, we have 1-to-1 mapping.

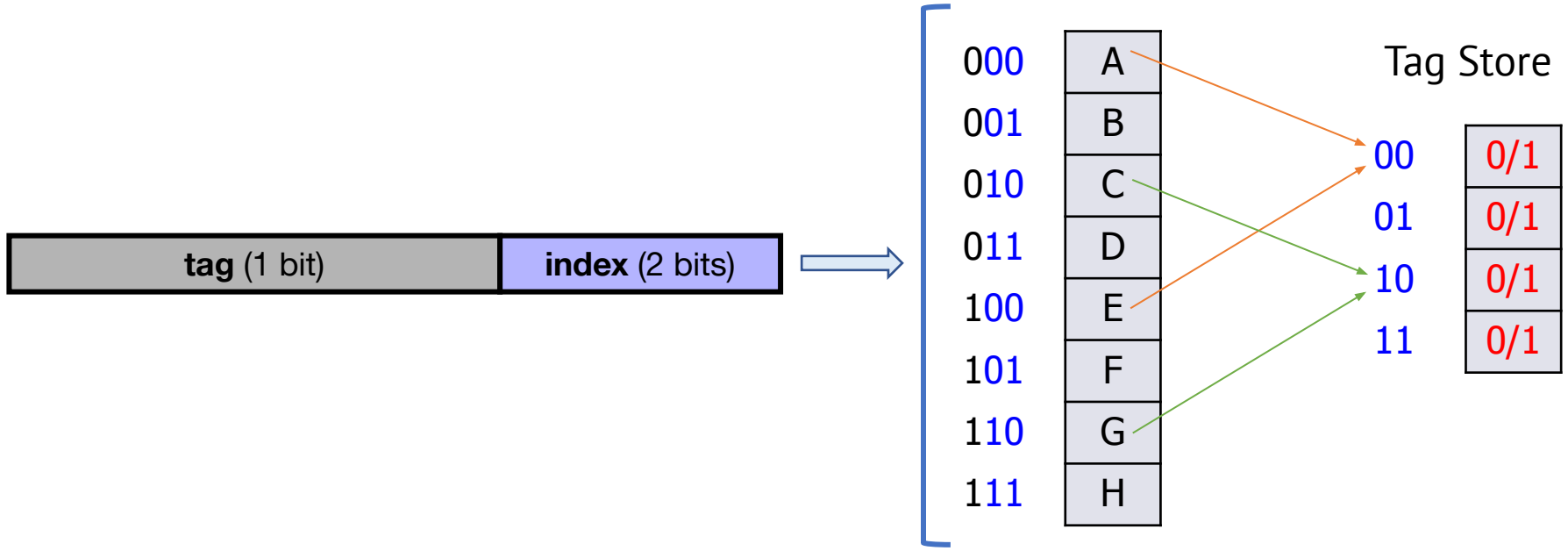
What should we store as the tag?



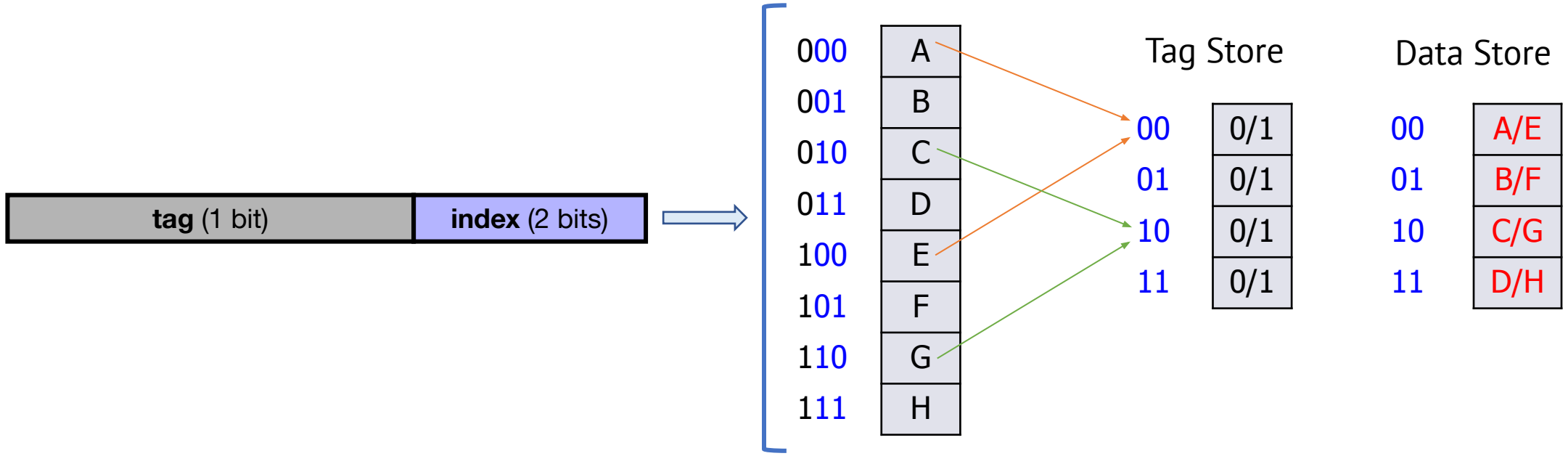
What should we store as the tag?



What should we store as the tag?

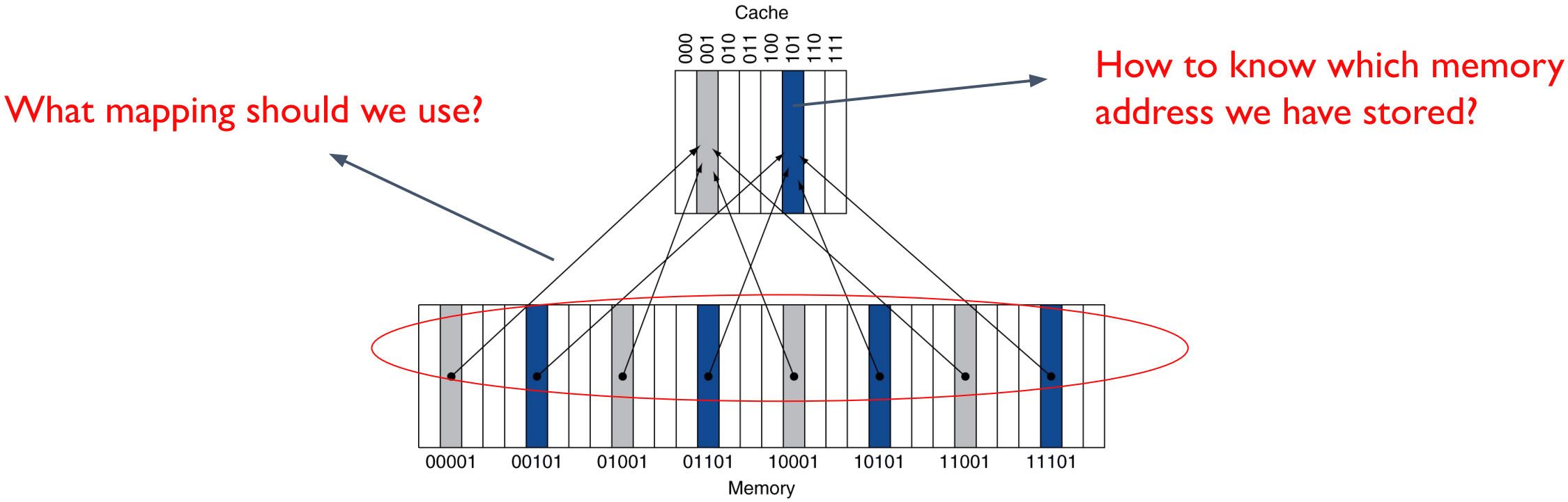


What should we store as the tag?



Reality: Cache << Memory

- ***One*** cache row is assigned to ***many*** memory lines.



What mapping should we use?

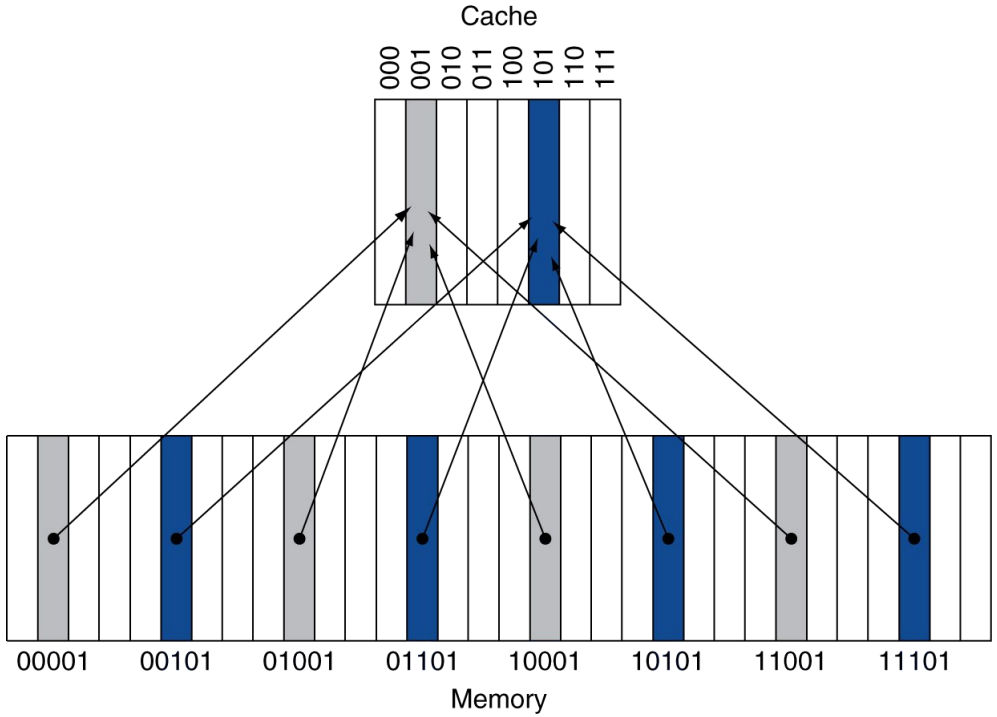
How to organize data in cache?

- There are different addressing modes
  a. Find the first available spot! (problem?)
  b. Use index to *only* go to one line! (problem?)

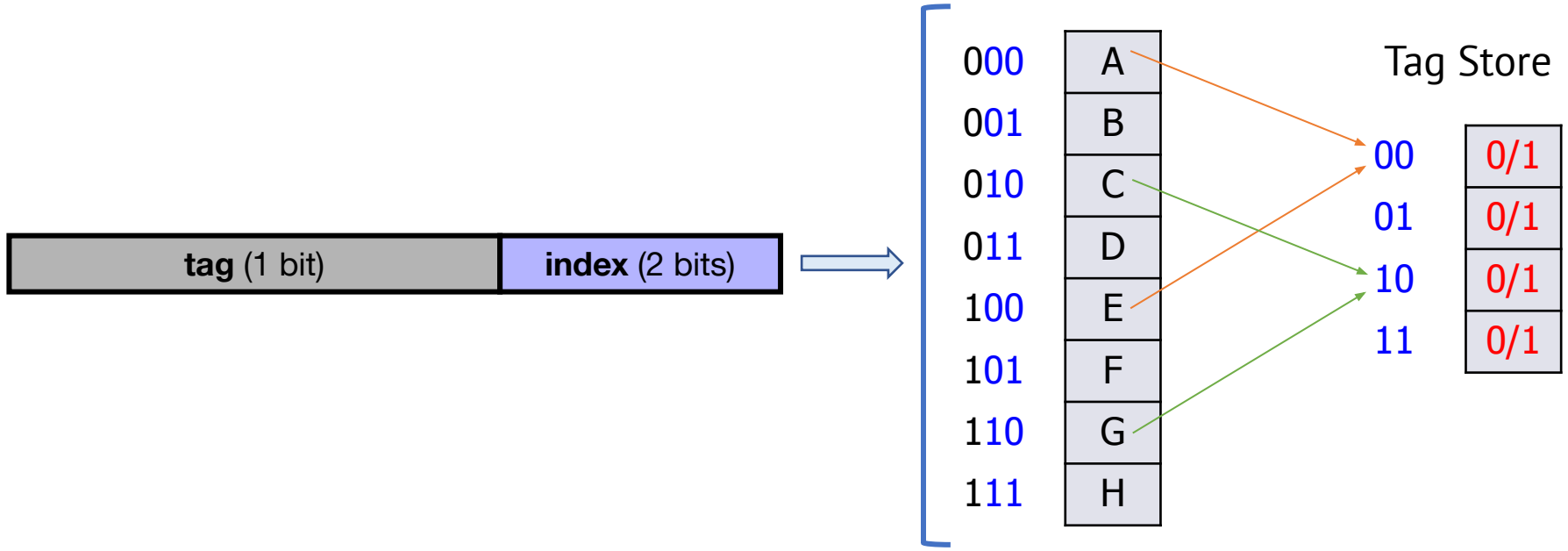
    - *Solution?*

Direct-Mapped Cache

- Same index should go to the same row!

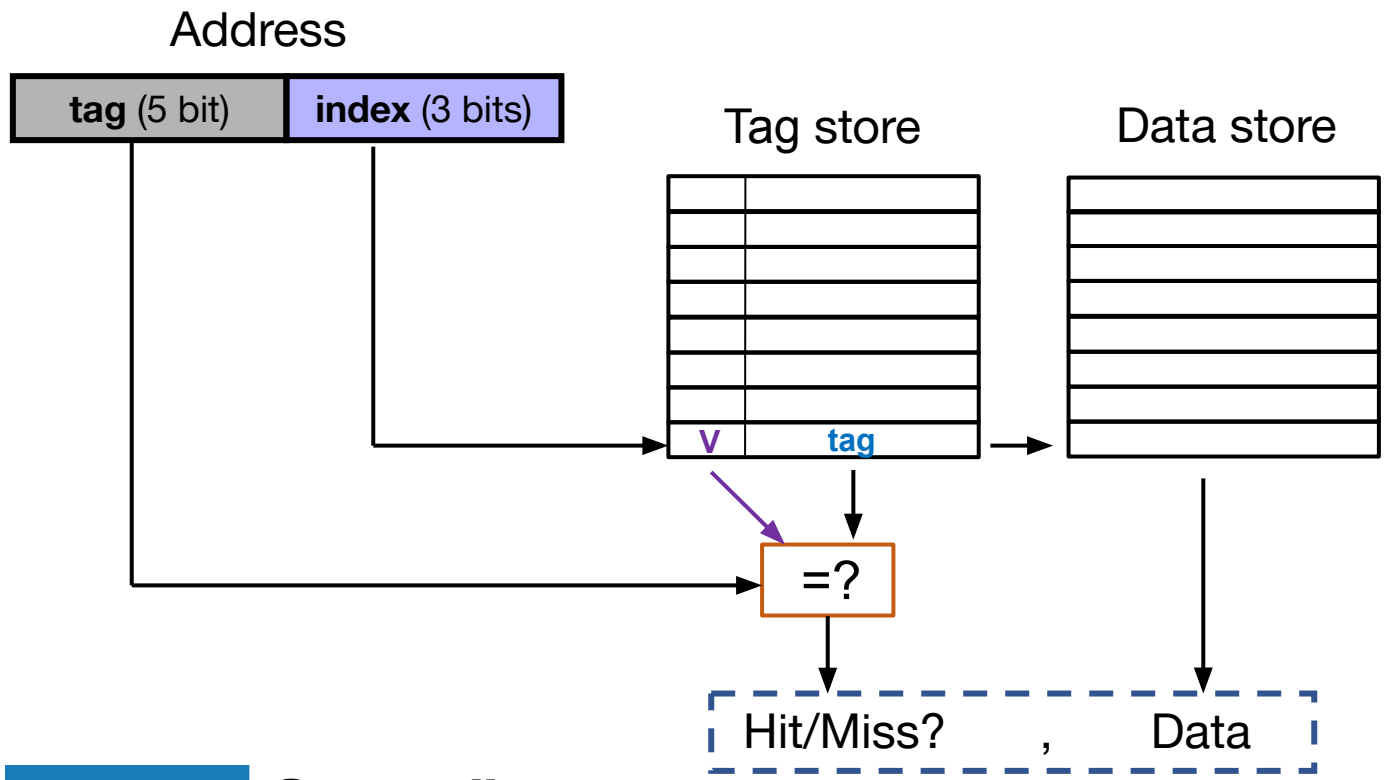


What should we store as the tag?



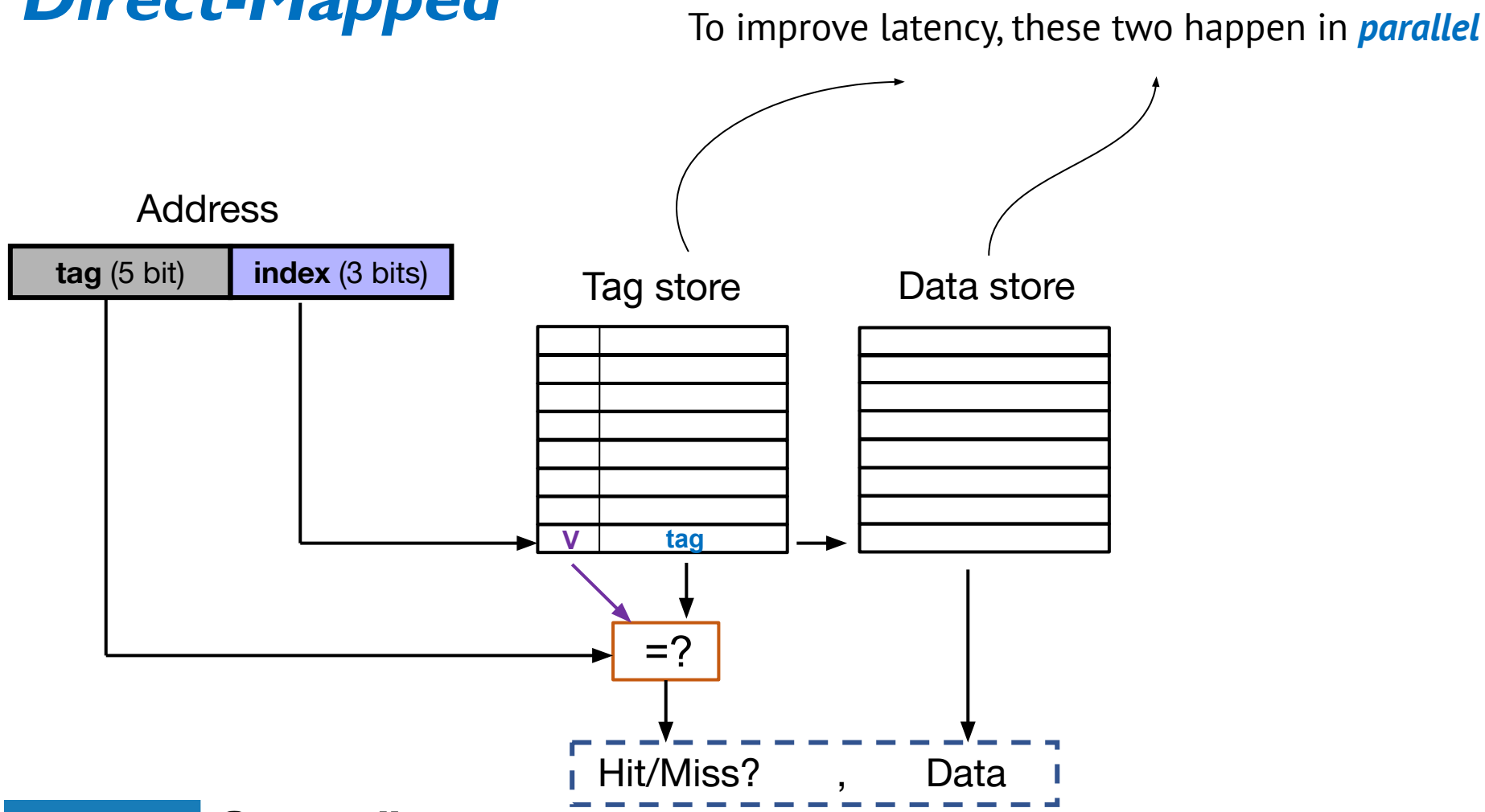
Cache Lookup

Direct-Mapped



Cache Lookup

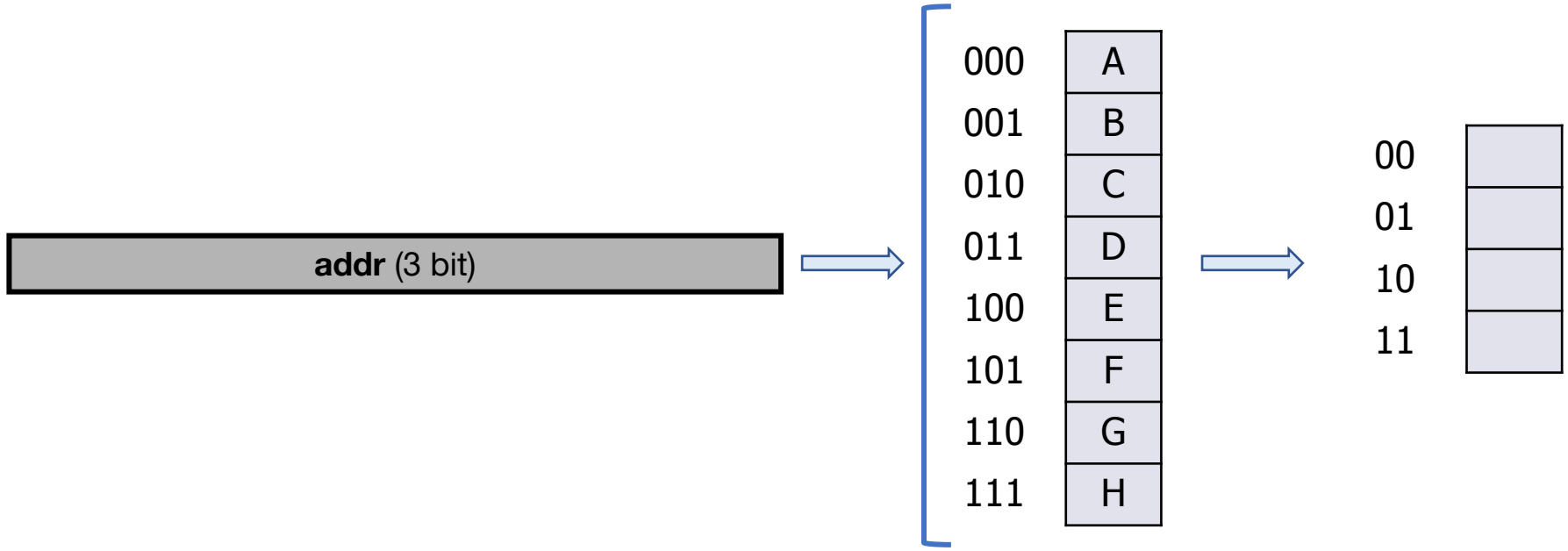
Direct-Mapped



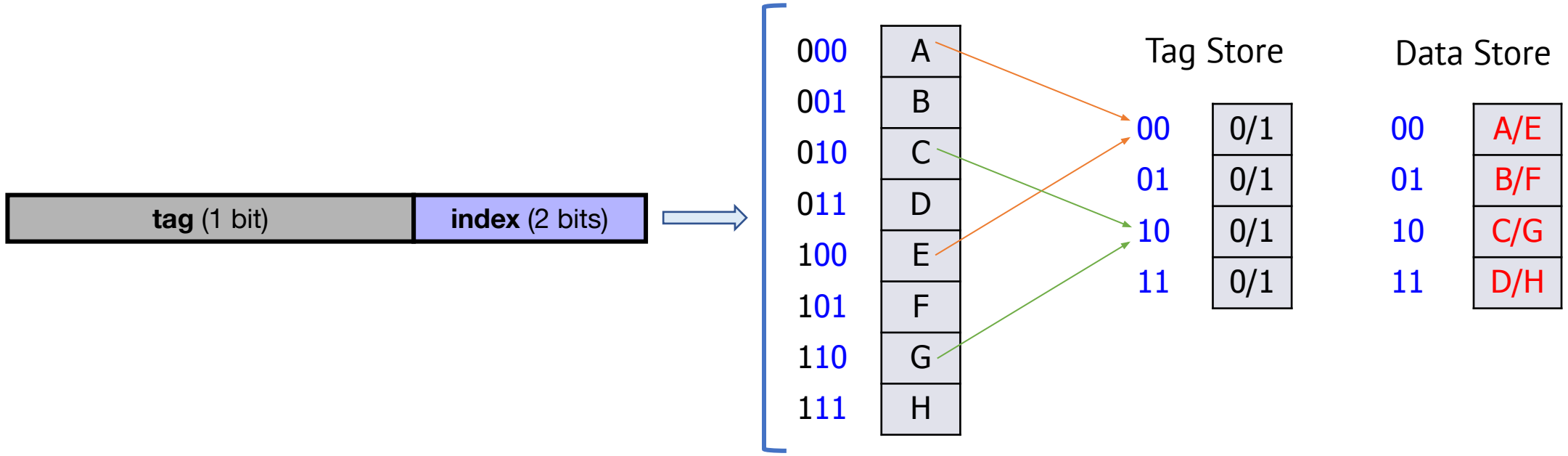
Alternative to Direct-Mapped?

- *Instead of assigning each memory address into a predetermined set, why don't we put the new data into any free set?*

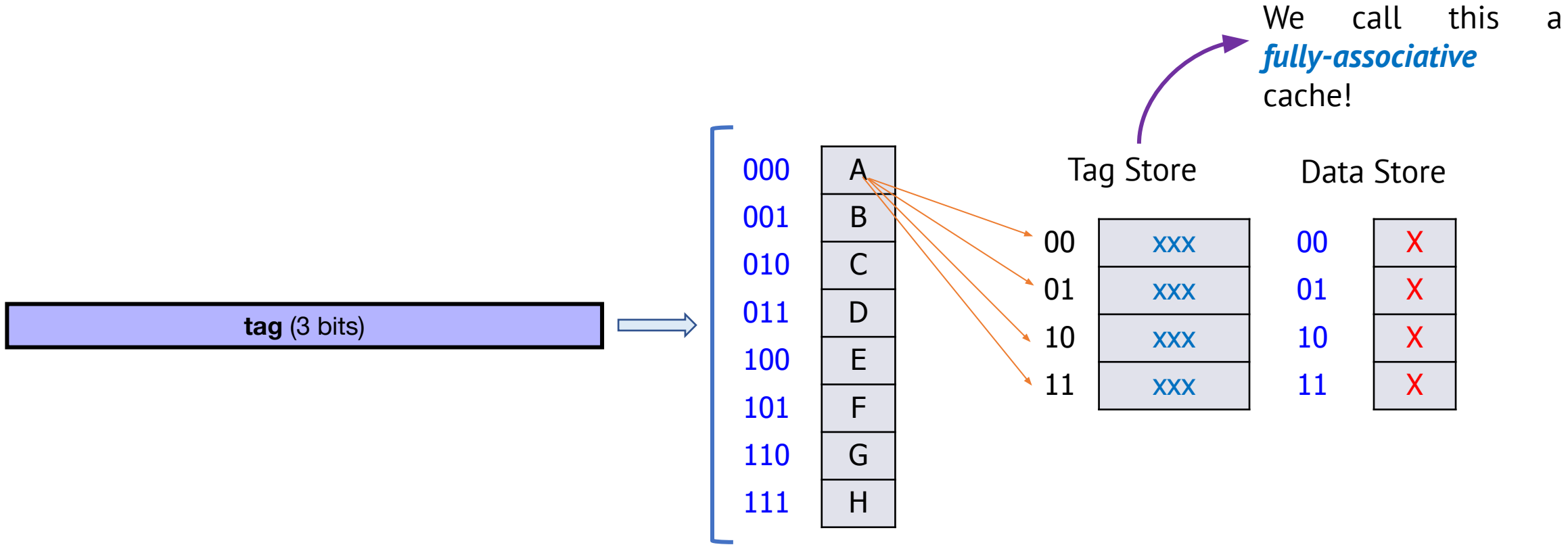
Alternative to Direct-Mapped?



Alternative to Direct-Mapped?



Alternative to Direct-Mapped?



Nader Sehatbakhsh <nsehat@ee.ucla.edu>

Direct-Mapped vs. Fully Associative

Example:

- 8-bit address
- 8 rows

Direct-Mapped

V	Index	Tag	Data
0	0		
0	1		
0	2		
0	3		
0	4		
0	5		
0	6		
0	7		

Fully Assoc.

V	Tag	Data
0		
0		
0		
0		
0		
0		
0		
0		

Direct-Mapped vs. Fully Associative

Example:

- 8-bit address
- 8 rows

★ Tag size?

Direct-Mapped

V	Index	Tag	Data
0	0		
0	1		
0	2		
0	3		
0	4		
0	5		
0	6		
0	7		

Fully Assoc.

V	Tag	Data
0		
0		
0		
0		
0		
0		
0		
0		

Direct-Mapped vs. Fully Associative

Example:

access pattern: 0, 1, 2, 8

Direct-Mapped

V	Index	Tag	Data
	0		
	1		
	2		
	3		
	4		
	5		
	6		
	7		

Fully Assoc.

V	Tag	Data

Direct-Mapped vs. Fully Associative

Example:

access pattern: 0,8,0,8,0,8

Direct-Mapped

V	Index	Tag	Data
	0		
	1		
	2		
	3		
	4		
	5		
	6		
	7		

Fully Assoc.

V	Tag	Data

Direct-Mapped vs. Fully Associative

- Average Access Time = HitTime + MissRate × MissPenalty
- HitTime
- MissRate
- Miss Penalty

Direct-Mapped vs. Fully Associative

- Average Access Time = HitTime + MissRate × MissPenalty
- HitTime
  DM << FA
- MissRate
  DM >> FA
- Miss Penalty
  DM == FA

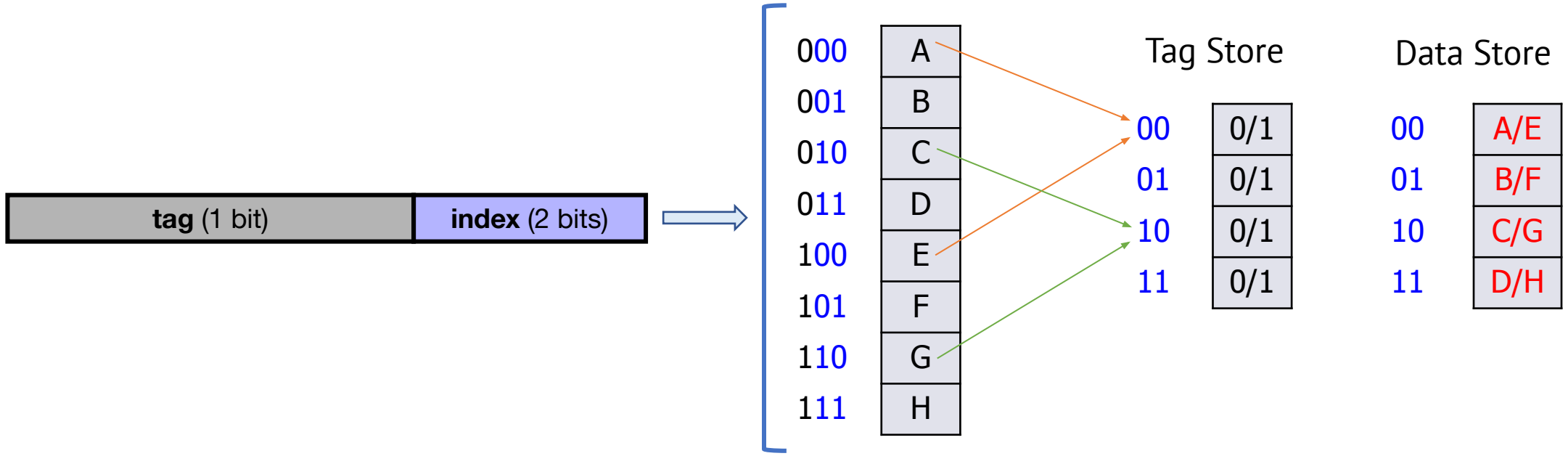
Can we have ***both***?

Can we have ***both***?

★ *Set-associative cache*

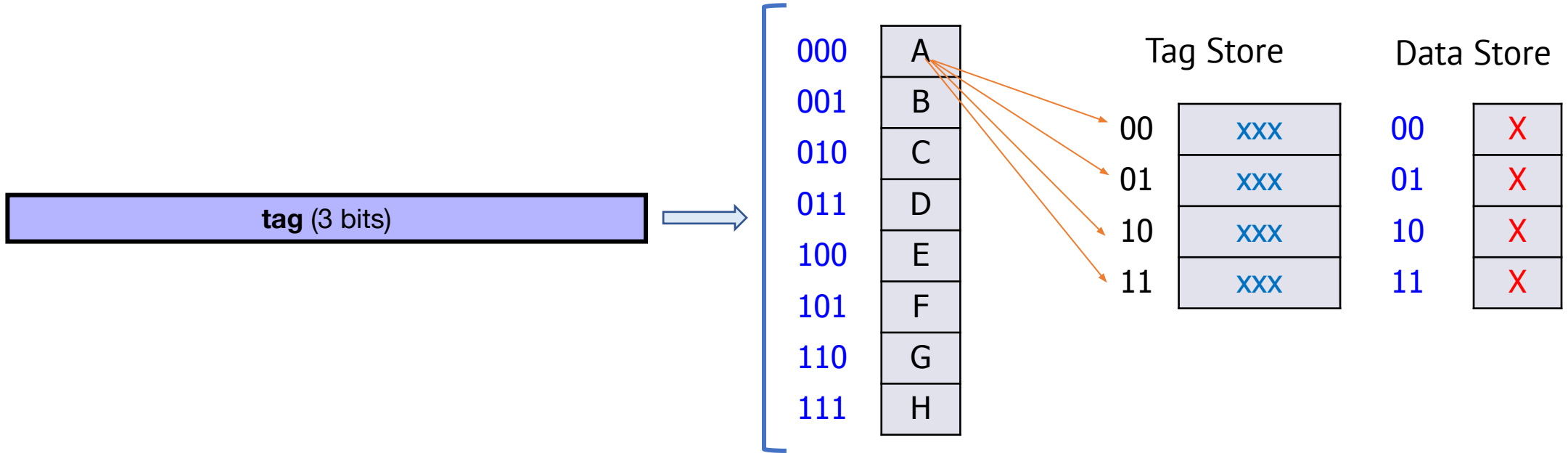
- *Add associativity within each set!*

Recall: DM



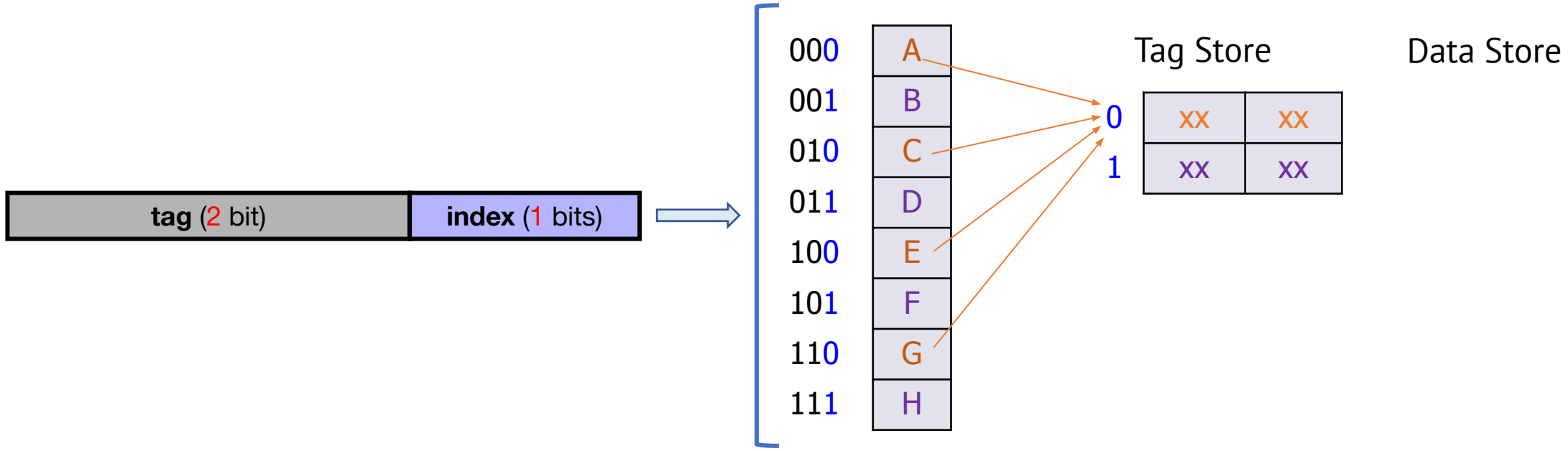
Nader Sehatbakhsh <nsehat@ee.ucla.edu>

Recall: FA



Nader Sehatbakhsh <nsehat@ee.ucla.edu>

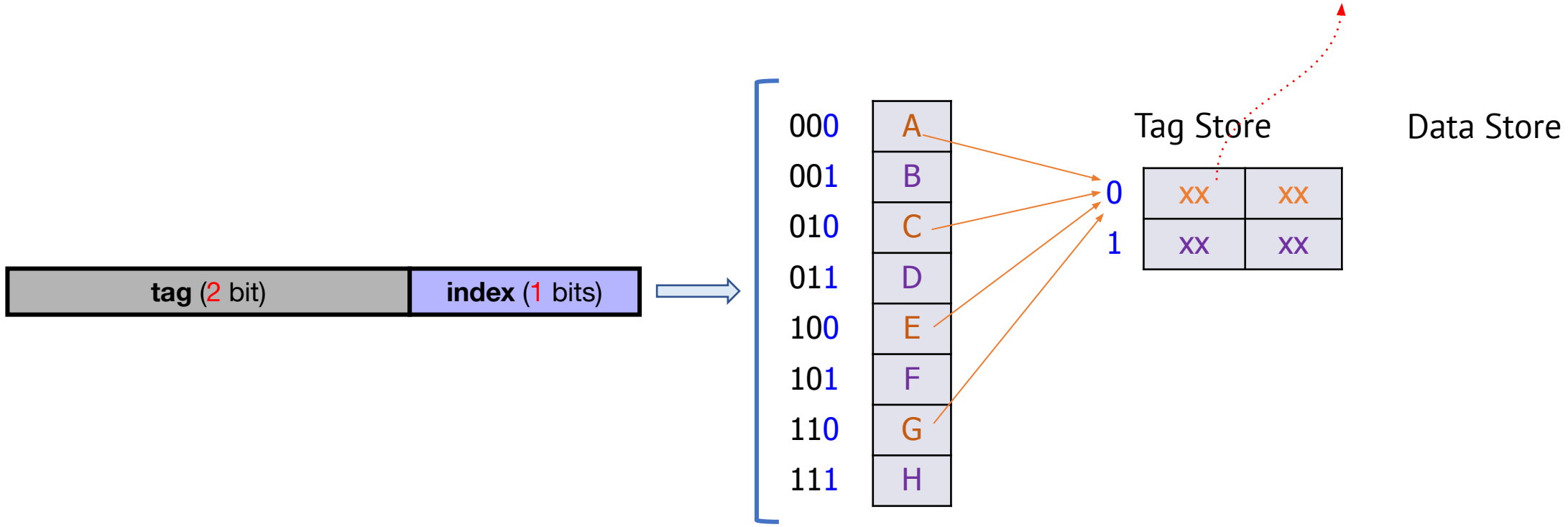
Set-Associative Cache



Nader Sehatbakhsh <nsehat@ee.ucla.edu>

Set-Associative Cache

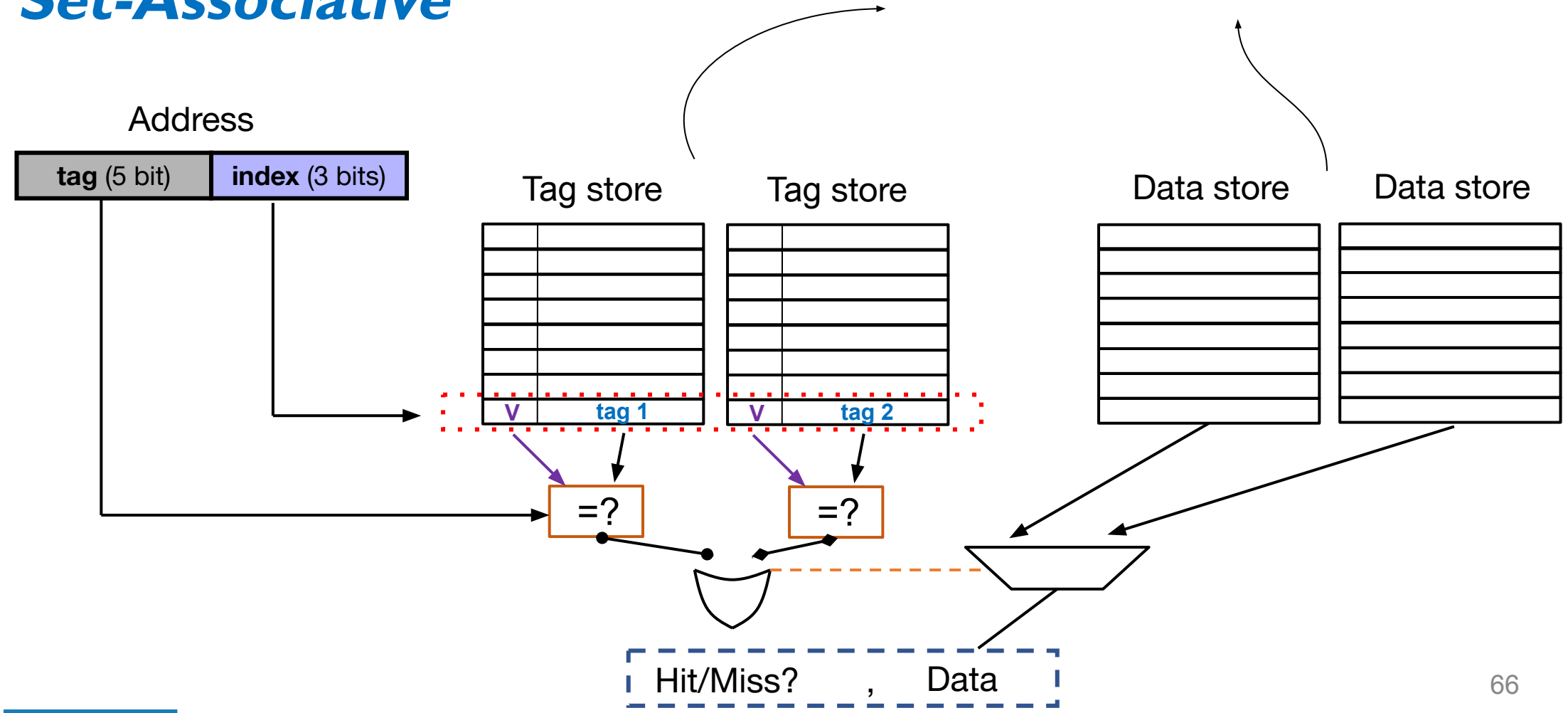
We call each column ***way***. This cache is 2-way set-assoc.



Cache Lookup

Set-Associative

To improve latency, ***all*** tag searches and ***all*** data accesses happen ***parallel***



Nader Sehatbakhsh <nsehat@ee.ucla.edu>

- Example:
  - 8-bit address

★ tag size?

Direct-Mapped

V	Index	Tag	Data
0	0		
0	1		
0	2		
0	3		
0	4		
0	5		
0	6		
0	7		

Set Assoc.

V	Index	Tag	Data	V	Index	Tag	Data
0	0			0	0		
0	1			0	1		
0	2			0	2		
0	3			0	3		

- ***pattern***: 0,8,0,8,0,8

DM						
SA						
FA						

- ***pattern***: 0,8,0,8,0,8

DM	M	M	M	M	M	M
SA	M	M	H	H	H	H
FA	M	M	H	H	H	H

Problem

P) Compute the outcome of DM, FA, and SA caches for this pattern:
0, 1, 2, 8, 1, 2, 0, 1

DM								
SA								
FA								

Direct-Mapped vs. Fully Associative vs. Set Associative

- Average Access Time = HitTime + MissRate × MissPenalty
- HitTime
- MissRate
- Miss Penalty

Direct-Mapped vs. Fully Associative vs. Set Associative

- Average Access Time = HitTime + MissRate × MissPenalty
- HitTime
  DM < SA << FA
- MissRate
  DM >> SA > FA
- Miss Penalty
  DM == FA == SA

What to do when cache is full?

What happens on a miss?

- *Pattern:*

  0,1,2,3,4,5,6,7,***8***

Direct-Mapped

V	Index	Tag	Data
0	0		
0	1		
0	2		
0	3		
0	4		
0	5		
0	6		
0	7		

Fully Assoc.

V	Tag	Data
0		
0		
0		
0		
0		
0		
0		
0		

What happens on a miss?

- *Pattern:*

  0,1,2,3,4,5,6,7,***8***

- *What to evict?*

Direct-Mapped

V	Index	Tag	Data
0	0		
0	1		
0	2		
0	3		
0	4		
0	5		
0	6		
0	7		

Fully Assoc.

V	Tag	Data
0		
0		
0		
0		
0		
0		
0		
0		

What happens on a miss?

- *Pattern:*

  0,1,2,3,4,5,6,7,***8***

  …, 7, …

- *What to evict?*

  -- it's important!

Direct-Mapped

V	Index	Tag	Data
0	0		
0	1		
0	2		
0	3		
0	4		
0	5		
0	6		
0	7		

Fully Assoc.

V	Tag	Data
0		
0		
0		
0		
0		
0		
0		
0		

What about Set Associative?

- *Pattern:*

  0,1,2,3,4,5,6,7,***8***

  …, 7, …

Set Assoc.

V	Index	Tag	Data	V	Index	Tag	Data
0	0			0	0		
0	1			0	1		
0	2			0	2		
0	3			0	3		

What to ***evict*** on a miss in a fully/set associative cache?

What to ***evict*** on a miss in a fully/set associative cache?

- Option 1: pick at ***random!***

What to ***evict*** on a miss in a fully/set associative cache?

- Option 1: pick at ***random!***
  - Very easy to implement
  - Not the best usage for *temporal* locality

V	Tag	Data
1	0	
1	1	
1	2	
1	3	
1	4	
1	5	
1	6	
1	7	

What to ***evict*** on a miss in a fully/set associative cache?

- Option 2: pick ***least-recently-used (LRU)!***
  - Find the oldest line and evict that!

V	Tag	Data
1	0	
1	1	
1	2	
1	3	
1	4	
1	5	
1	6	
1	7	

What to ***evict*** on a miss in a fully/set associative cache?

- Option 2: pick ***least-recently-used (LRU)!***
  - Find the oldest line and evict that!
    - Locality
    - Need to store age for each line *(overhead!)*

V	Tag	Data
1	0	
1	1	
1	2	
1	3	
1	4	
1	5	
1	6	
1	7	

LRU Replacement Policy

Pattern: 1,2,8,9,4,1,3,4,5,6,7,8,9, …

V	Tag	Pos.	Data
1	0		
1	1		
1	2		
1	3		
1	4		
1	5		
1	6		
1	7		

LRU Replacement Policy

Pattern: 1,2,8,9,4,1,3,4,5,6,7,8,9, …

-- Too costly for large caches!

V	Tag	Pos.	Data
1	0		
1	1		
1	2		
1	3		
1	4		
1	5		
1	6		
1	7		

What can we do?

Pseudo-LRU Replacement Policy

-- Store only one bit per row

- Start from all zero.

V	Tag	flag	Data
1	0	0	
1	1	0	
1	2	0	
1	3	0	
1	4	0	
1	5	0	
1	6	0	
1	7	0	

Pseudo-LRU Replacement Policy

-- Store only one bit per row

- Start from all zero.
- On a hit, set bit to 1

V	Tag	flag	Data
1	0	0	
1	1	0	
1	2	0	
1	3	0	
1	4	0	
1	5	0	
1	6	0	
1	7	0	

*Pseudo-*LRU Replacement Policy

-- Store only one bit per row

- Start from all zero.
- On a hit, set bit to 1
- On a miss, find the first row with flag== 0.

V	Tag	flag	Data
1	0	0	
1	1	0	
1	2	0	
1	3	0	
1	4	0	
1	5	0	
1	6	0	
1	7	0	

Pseudo-LRU Replacement Policy

-- Store only one bit per row

- Start from all zero.
- On a hit, set bit to 1
- On a miss, find the first row with flag== 0.
- If all bits are 1, resets. Pick the first row.

V	Tag	flag	Data
1	0	1	
1	1	1	
1	2	1	
1	3	1	
1	4	1	
1	5	1	
1	6	1	
1	7	1	

Replacement Policy

- *Install:* Where to put a new line?
- *Update:* What to do for a hit?
- *Replace:* If cache is full, what to replace?

Data Management

LRU vs. Random vs. PLRU

- *Install:* Where to put a new line?
- *Update:* What to do for a hit?
- *Replace:* If cache is full, what to replace?

Can we do better?

Replacement Policy Tradeoffs

- Storage vs. Accuracy
  - *Possible solutions:*
    - Approximate (pseudo-LRU)
    - Adaptive
    - Hybrid
    - …

Important Note: replacement policy performance is quite dependent on the access pattern (i.e., what type of locality are we seeing).

Impact of Replacement Policy on AAT

- Average Access Time = HitTime + MissRate × MissPenalty
  - HitTime
  - MissRate
  - Miss Penalty

Three Types of Miss in a Cache

- ***Compulsory Miss***
  - To have something in the cache, first it must be *fetched*.
  - The initial fetch of anything is a *miss*.
  - Also called unique references or first-time references.

Three Types of Miss in a Cache

- ***Compulsory Miss***
  - To have something in the cache, first it must be *fetched*.
  - The initial fetch of anything is a *miss*.
  - Also called unique references or first-time references.
- ***Capacity Miss***
  - A miss that occurs due to the ***limited capacity*** in a *fully-associative cache*.
  - The block/data was *replaced* before it was re-referenced.

Three Types of Miss in a Cache

- ***Compulsory Miss***
  - To have something in the cache, first it must be *fetched*.
  - The initial fetch of anything is a *miss*.
  - Also called unique references or first-time references.
- ***Capacity Miss***
  - A miss that occurs due to the ***limited capacity*** in a *fully-associative cache*.
  - The block/data was *replaced* before it was re-referenced.
- ***Conflict Miss***
  - For *set associative* or *direct-mapped* only.
  - Misses due to the index bits matching (conflicts).
  - Also called mapping misses.

Optimizing Cache

Average Access Time = HitTime + MissRate × MissPenalty

- *How to improve each term?*

Reducing Miss Rate

Reducing Miss Rate

increase block size

- Increase Block Size
  - Bring more than one byte per access → Exploit more *spatial locality*
  - Think about the library example!
  - Arrays, integers, etc.

Cache Block

- Instead of storing 1 byte per row, we can store a block with multiple bytes.
  - Every time we need to load something to the cache, we load it at block level.

Cache Block

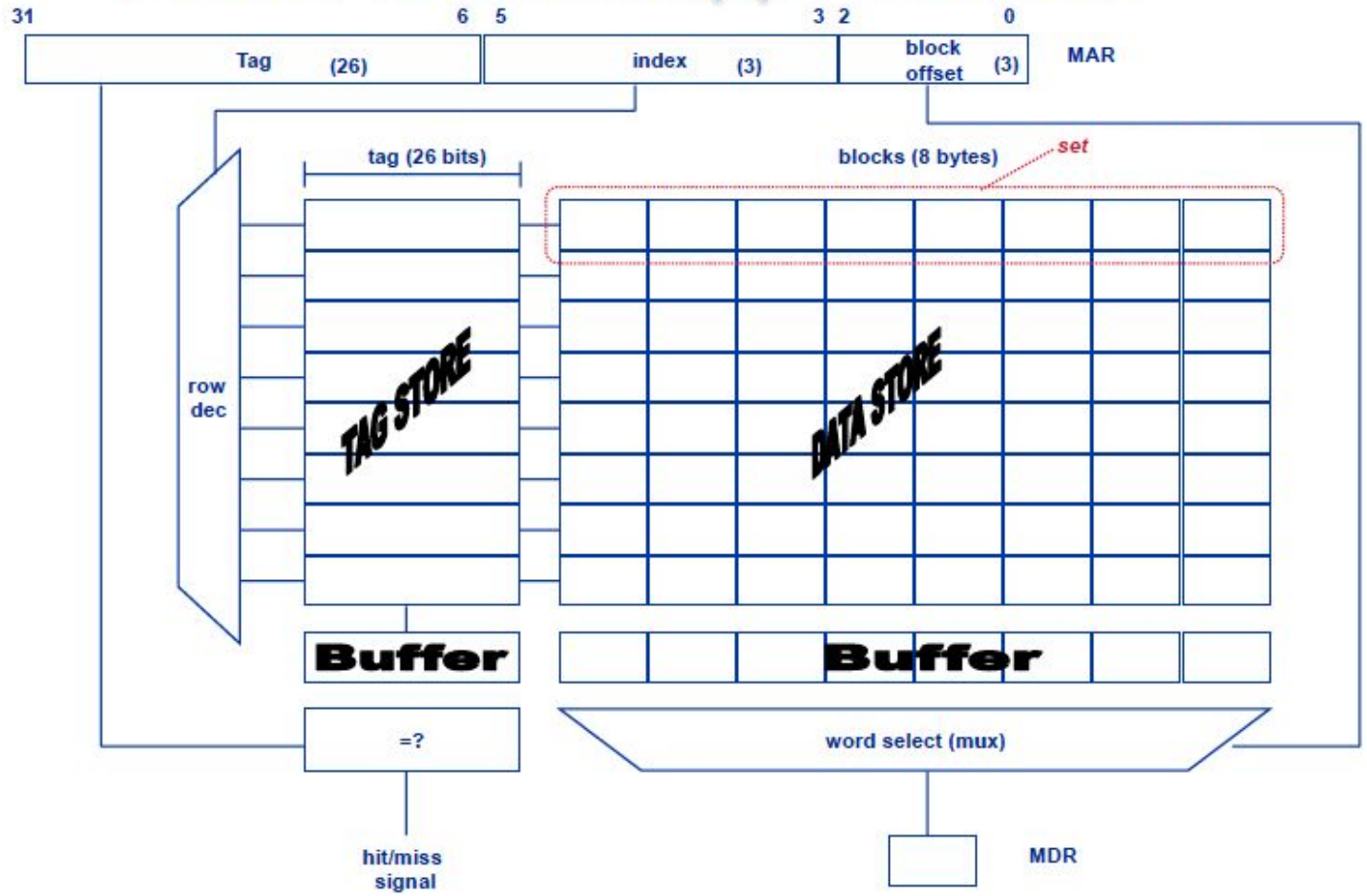
- Instead of storing 1 byte per row, we can store a block with multiple bytes.
  - Every time we need to load something to the cache, we load it at block level.
  - We still send things to the CPU at byte level.
    - How to do that? → We need *block offset* to decide which byte within the block should be selected!
  - How big a block should be?
    - It depends! Typically, somewhere between 8-64B.

Cache Block

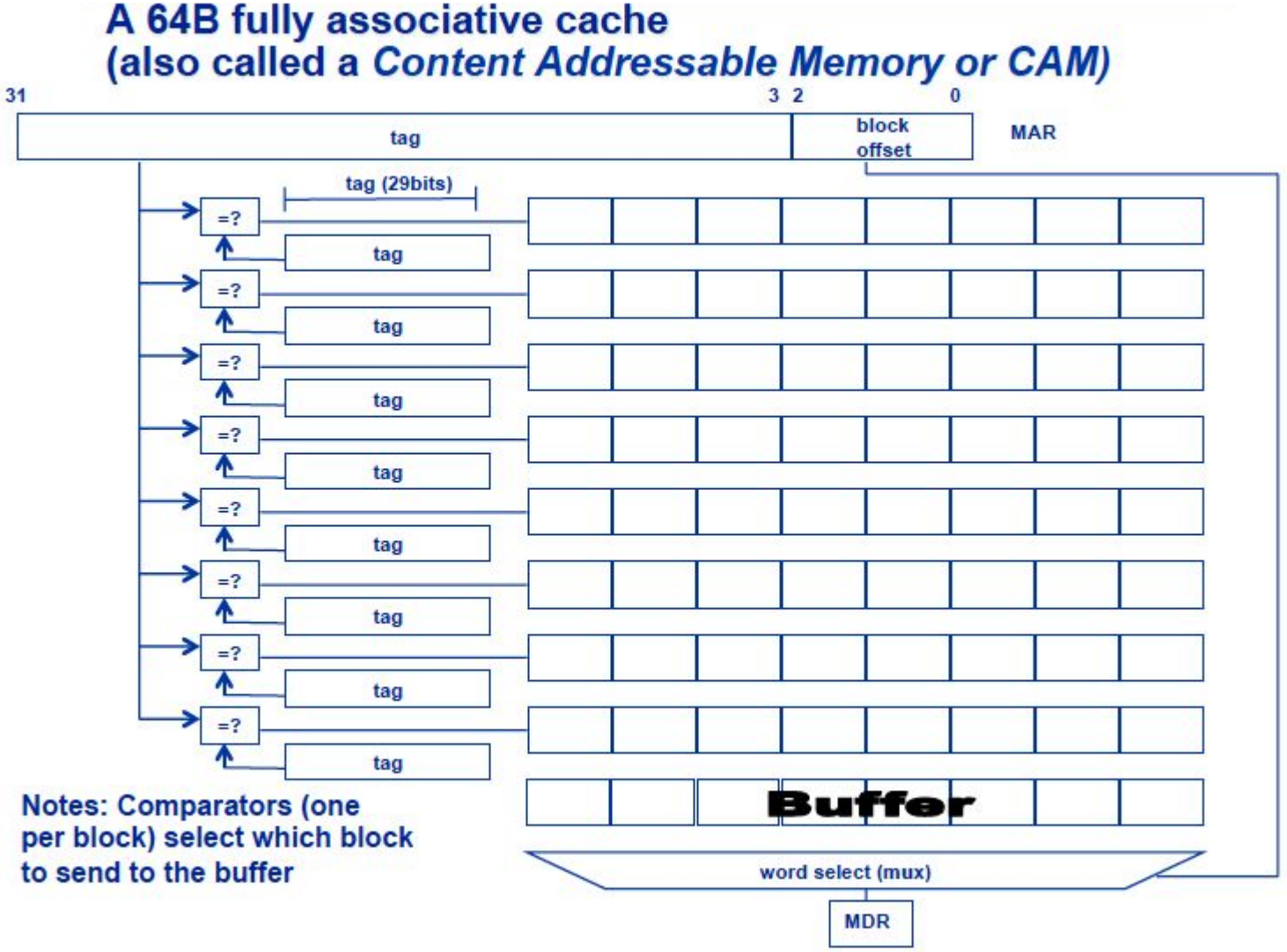
- Instead of storing 1 byte per row, we can store a block with multiple bytes.
  - Every time we need to load something to the cache, we load it at block level.
  - We still send things to the CPU at byte level.
    - How to do that? → We need *block offset* to decide which byte within the block should be selected!
  - How big a block should be?
    - It depends! Typically, somewhere between 8-64B.

- Cache Address = {tag, index, block offset}

A 64B direct mapped cache

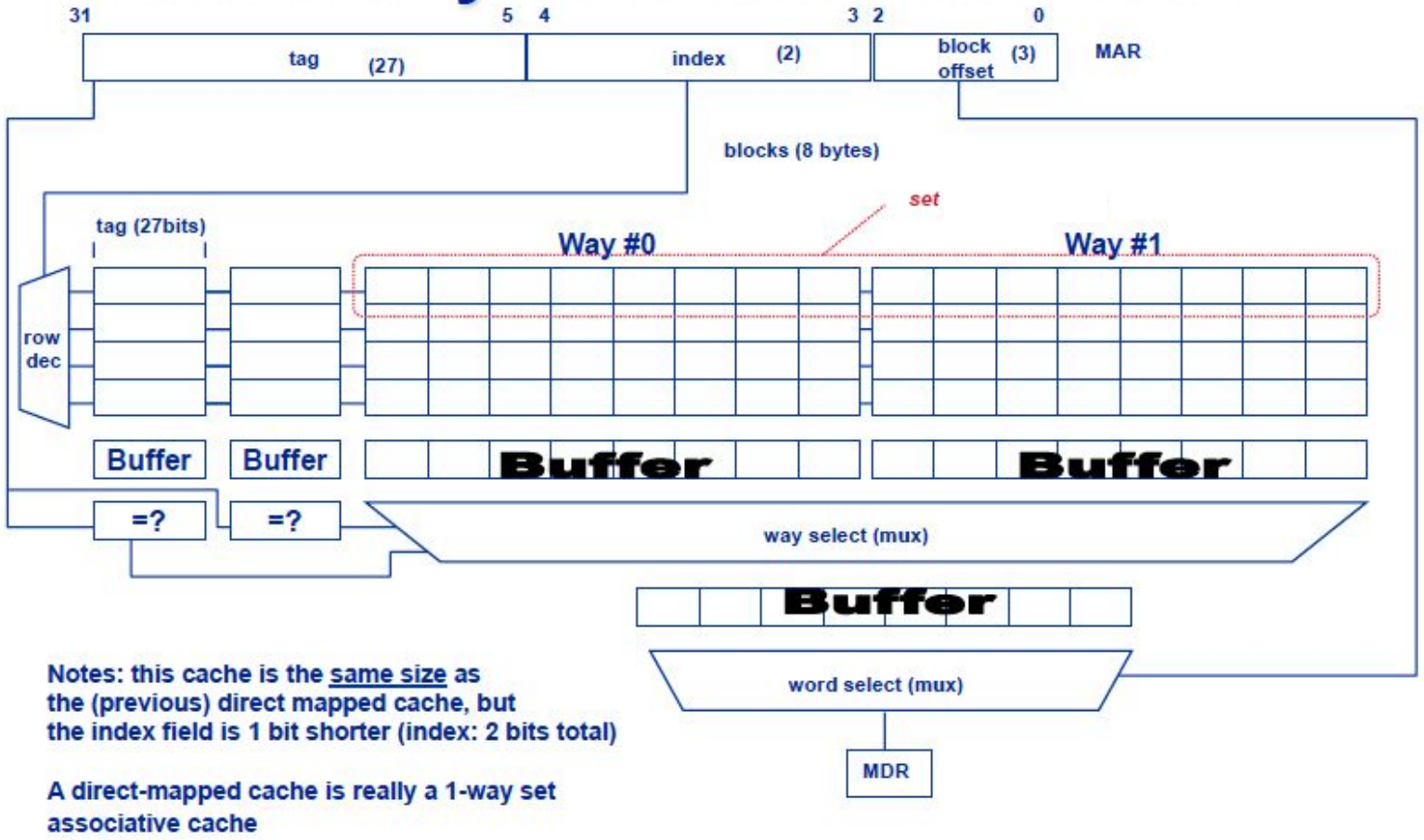


Picture is borrowed from CS4290, Tom Conte (Georgia Tech).



Picture is borrowed from CS4290, Tom Conte (Georgia Tech).

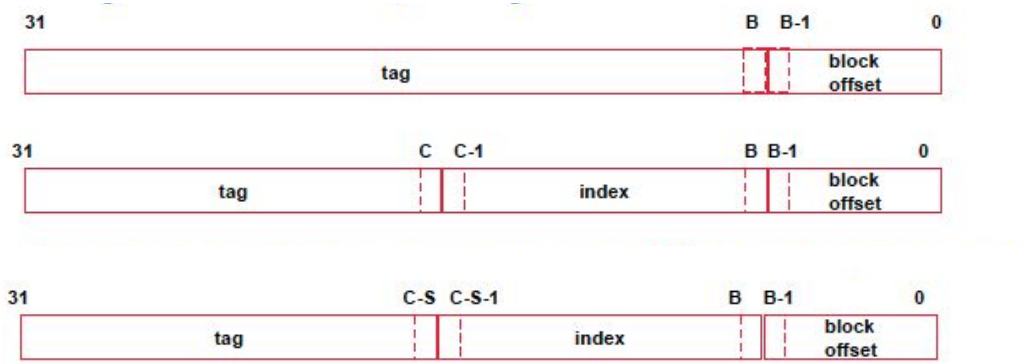
A 64B 2-way set associative cache



Picture is borrowed from CS4290, Tom Conte (Georgia Tech).

(CBS)

- C = log(bytes per cache)
- B = log(bytes per block)
- S = log(blocks per set)

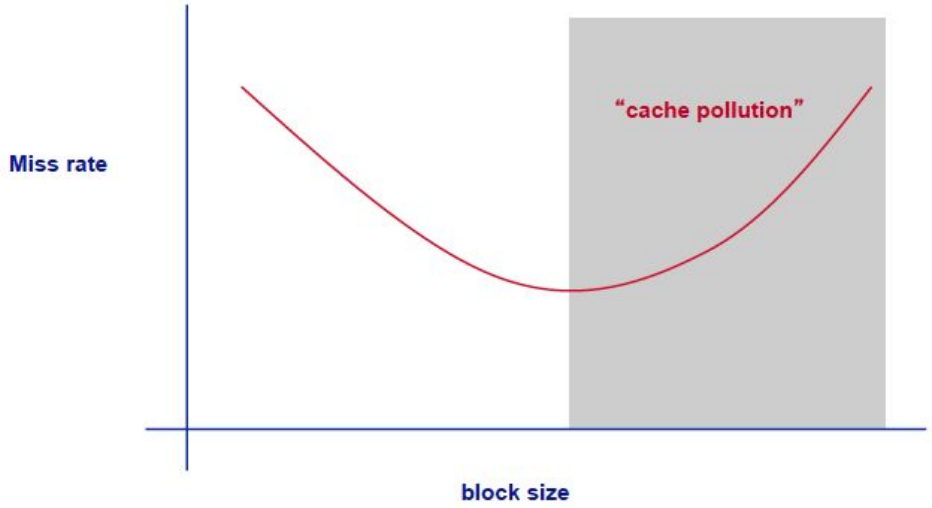


How big a block should be?

- If bringing more is helpful, why don't we have very large blocks?

Reducing Miss Rate
increase block size

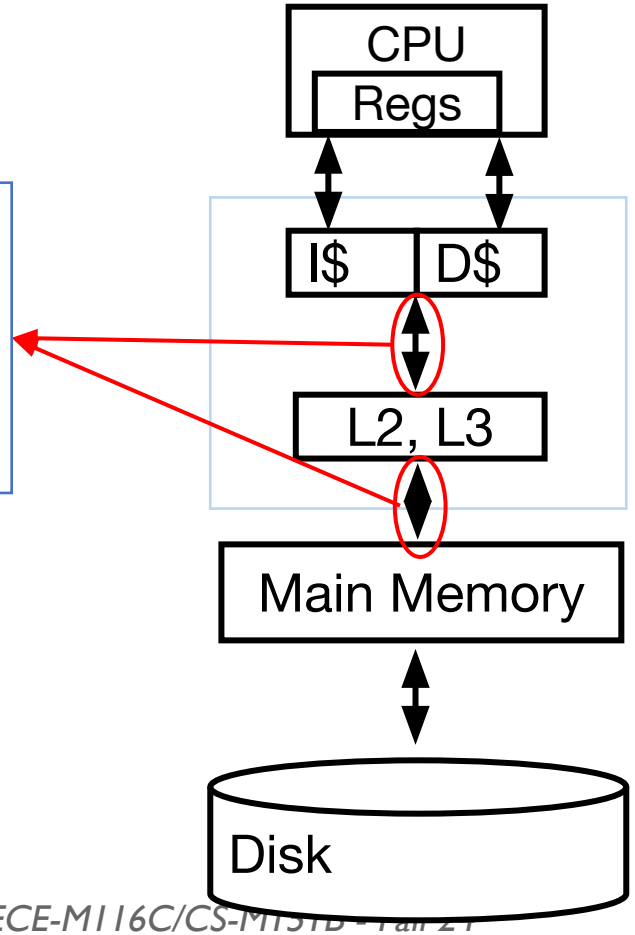
- Increase Block Size
  - Exploit more *spatial locality*
  - Large block size brings too many useless data *(called cache pollution).*
  - Increase miss penalty
    (have to bring more in on a miss)



Bandwidth vs. Latency

Latency: how long it takes to bring 1 block?

Bandwidth: how many blocks can we bring in unit time?



Reducing Miss Rate

increase associativity

- FA > SA > DM

Reducing Miss Rate

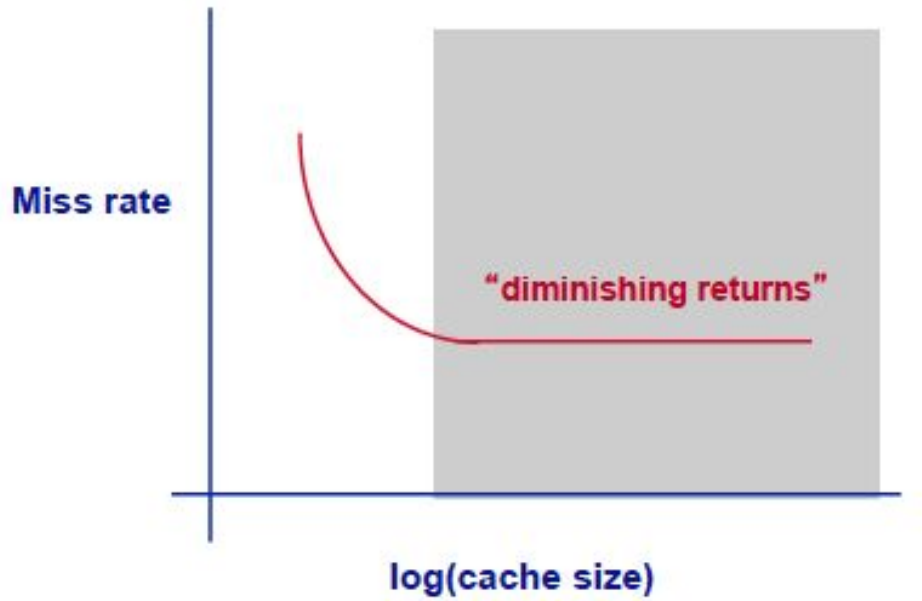
increase associativity

- FA > SA > DM
- More ways → higher hit time (diminishing returns)

-- How many ways is good enough?

- 8-way is as good as FA

(after that, the limit is capacity miss).



Reducing Miss Rate

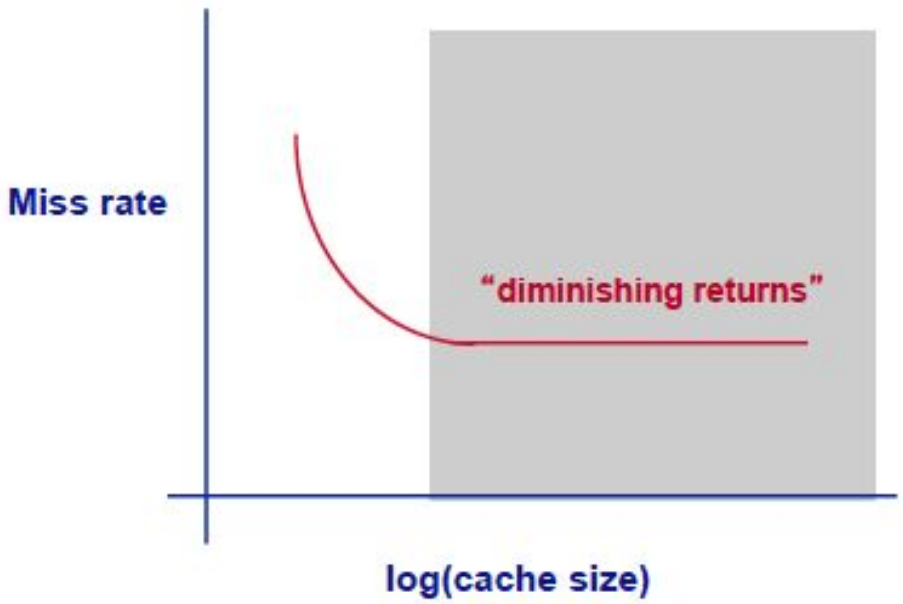
increase cache size

- Larger cache → more data!

Reducing Miss Rate

increase cache size

- Larger cache → more data!
- Larger cache → slower hit time
- Diminishing returns:
  - For a large cache
    double size != double performance
    (the limit becomes compulsory misses!)



Reducing Miss Rate
prefetching

★ Idea: *if we can guess the access pattern, we can bring data before it's needed!*

i, i+1, i+2, i+3, …

Reducing Miss Rate
prefetching

★ Idea: *if we can guess the access pattern, we can bring data before it's needed!*

- Reduce compulsory misses
- Cache pollution
  - Need to monitor prefetching accuracy to change its *aggressiveness.*
  - Other than this, no other negative impacts! No correctness issues!

i, i+1, i+2, i+3, …

Prefetching - Four Questions!

- What
  - What addresses to prefetch (i.e., address prediction algorithm)
- When
  - When to initiate a prefetch request (early, late, on time)
- Where
  - Where to place the prefetched data (caches, separate buffer)
- How
  - How does the prefetcher operate and who operates it (software, hardware, hybrid)

Software vs. Hardware Prefetch

- Software prefetching
  - ISA provides prefetch instructions
  - Programmer or compiler inserts prefetch instructions (effort)
  - Usually works well only for "regular access patterns"

- Hardware prefetching
  - Hardware monitors processor accesses
  - Memorizes or finds patterns/strides
  - Generates prefetch addresses automatically

Examples: Software

```
for (i=0; i<N; i++) {
__prefetch(a[i+8]);
__prefetch(b[i+8]);
sum += a[i]*b[i];
}

while (p) {
__prefetch(pnext);
work(pdata);
p = pnext;
}
```

Example: Hardware

- Next line prefetcher:
  - Always prefetch next N cache lines after a demand access
  - Tradeoffs:
    - Simple to implement.
    - No need for sophisticated pattern detection
    - Works well for sequential/streaming access patterns (instructions?)
    - Can waste bandwidth with irregular patterns
    - Low prefetch accuracy if access stride = 2 or when the program is traversing memory from higher to lower addresses

Example: Hardware

- Next line prefetcher:
  - Always prefetch next N cache lines after a demand access
  - Tradeoffs:
    - Simple to implement.
    - No need for sophisticated pattern detection
    - Works well for sequential/streaming access patterns (instructions?)
    - Can waste bandwidth with irregular patterns
    - Low prefetch accuracy if access stride = 2 or when the program is traversing memory from higher to lower addresses

★ *Better options?* stride prefetcher, stream buffers, etc.

Reducing Miss Rate
prefetching

★ Idea: *if we can guess the access pattern, we can bring data before it's needed!*

- Reduce compulsory misses
- Cache pollution
  - Need to monitor prefetching accuracy to change its *aggressiveness.*
  - Other than this, no other negative impacts! No correctness issues!

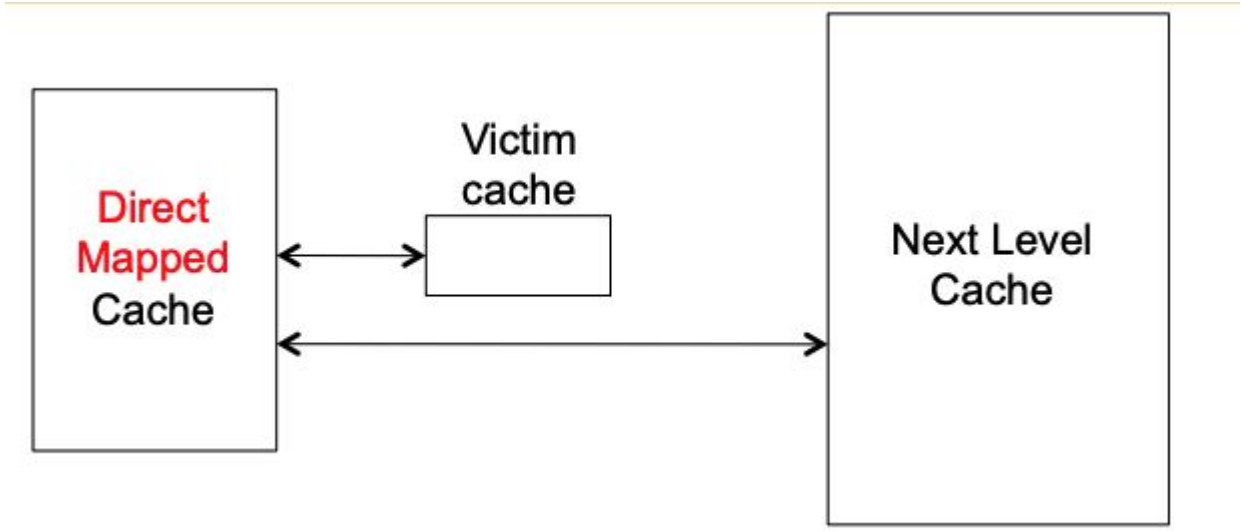
i, i+1, i+2, i+3, …

Reducing Miss Rate

victim cache

★ Idea: *for heavily conflicting addresses, a few "extra" temporary sets could remove conflicts!*

- *Use a very small buffer (called victim cache) to save the recently discarded blocks. Search through them as well.*



How does Victim cache help?

- Example:
  - 8-bit address

Direct-Mapped

V	Index	Tag	Data
0	0		
0	1		
0	2		
0	3		
0	4		
0	5		
0	6		
0	7		

Set Assoc.

V	Index	Tag	Data	V	Index	Tag	Data
0	0			0	0		
0	1			0	1		
0	2			0	2		
0	3			0	3		

- ***pattern***: 0,8,0,8,0,8

DM						
SA						
FA						

Reducing Miss Rate

victim cache

★ Idea: *for heavily conflicting addresses, a few "extra" temporary sets could remove conflicts!*

- *Use a very small buffer (called victim cache) to save the recently discarded blocks. Search through them as well.*

- Reduce conflict misses
  - Research shows a 4-entry victim cache can remove up to 90% of conflicts.
- Extra overhead.
- More complex design.

Reducing Miss Rate

compiler and software

- Re-order accesses/arrays to increase locality.
- Combine loops with similar behavior.
- Use "tiling" to access arrays region by region instead of whole.
- Use compiler profiling to improve prefetching.
- …

Tiling

- If column-major
  - `x[i+1,j]` follows `x[i,j]` in memory
  - `x[i,j+1]` is far away from `x[i,j]`

Poor code

```
for     i     =     1,     rows
  for    j     =    1,    columns
    sum = sum + x[i,j]
```

Better code

```
for     j     =     1,     columns
  for    i     =    1,    rows
    sum = sum + x[i,j]
```

Reducing Miss Rate

replacement policy

- LRU vs. PLRU vs. Random

- Storage vs. accuracy tradeoff!

How to reduce miss penalty?

Reducing Miss Penalty
write buffer

- *Use load-store queue.*

- No wait for stores needed.
- Lower miss penalty for loads.
- More overhead.

What to do on a store?

What to do on a store?

1- Data exists in the cache?

What to do on a store?

1- Data exists in the cache?

- Should we update memory AND cache on every write?
- Update the memory only when the line is evicted.
  - Need to keep track of changes (*dirty* bit).

What to do on a store?

1- Data exists in the cache?

- Should we update memory AND cache on every write?
  -- *write through strategy*
- Update the memory only when the line is evicted.
  -- *write back strategy*

***Tradeoff:** Less writes vs. Storage overhead vs. Memory status*

What to do on a store?

1- Data exists in the cache?

2- Data does not exist.

- Should we bring it to the cache? *– write allocate*

What to do on a store?

1- Data exists in the cache?

2- Data does not exist.

- Should we bring it to the cache? *– write allocate*
- We probably don't need it anymore, so don't bring it. *– write no allocate*

What to do on a store?

- *Write back* often combined with *write-allocate.*
- *Write-through* often combined with *write-no allocate.*

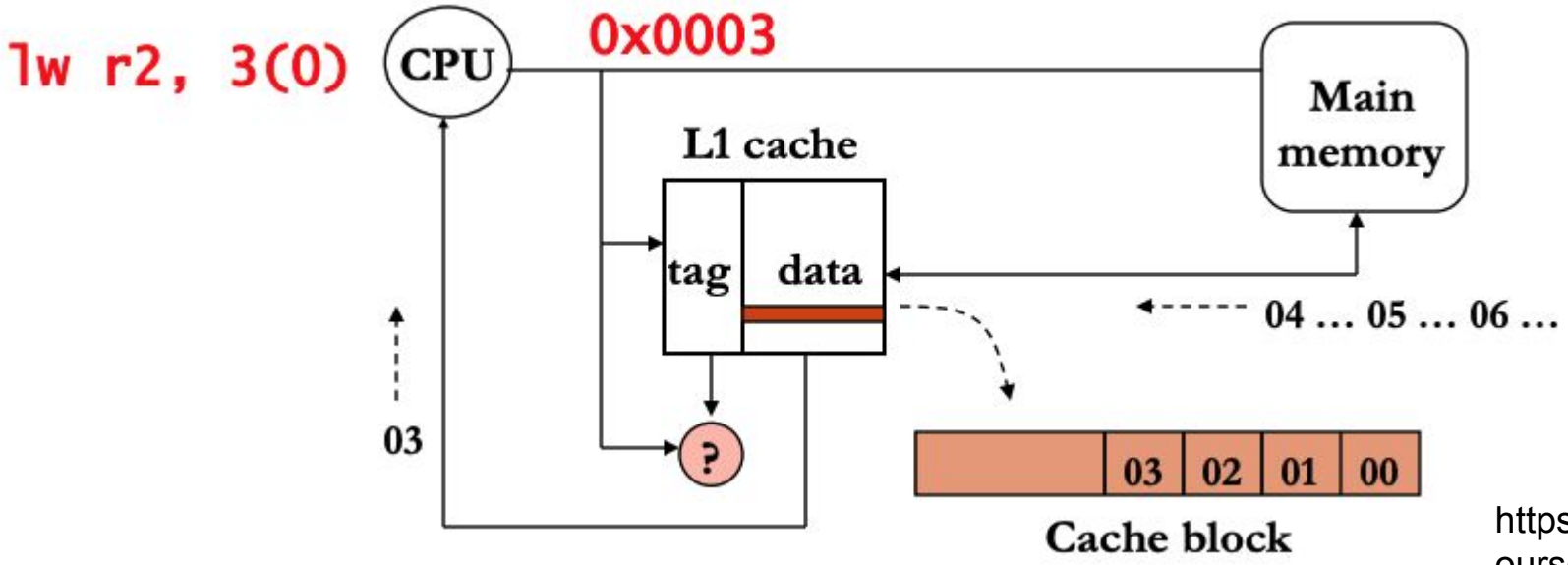
How to pick?

- It *depends*!
  - Could be different for each level!
  - Can be optimized using simulation and architectural search!

Reducing Miss Penalty

early restart

- *Instead of waiting for all bytes (in a block) to arrive, forward data to CPU as soon as the requested byte(s) arrives.*



https://www.inf.ed.ac.uk/teaching/courses/car/Notes/2013-14/lecture07a-cache_performance.pdf

Reducing Miss Penalty

early restart with critical word first

- *Instead of waiting for all bytes (in a block) to arrive, forward data to CPU as soon as the requested byte(s) arrives.*

- *To further optimize this, first read the requested byte!*

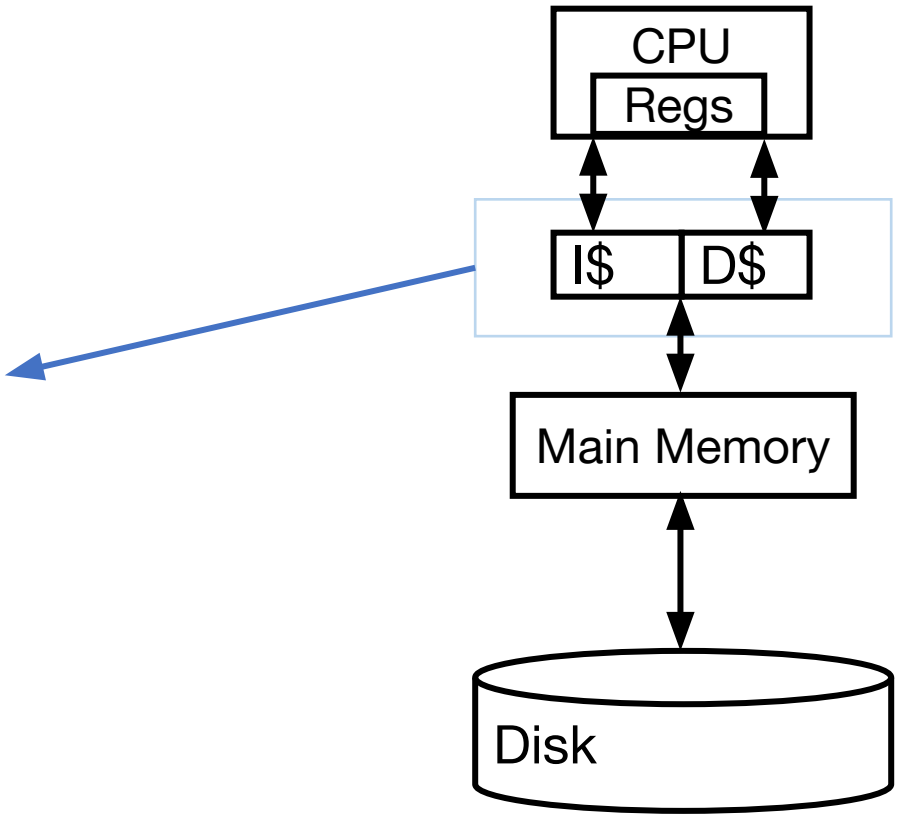
Reducing Miss Penalty

add more levels

- *Adding L2, L3, etc. to further reduce miss penalty!*

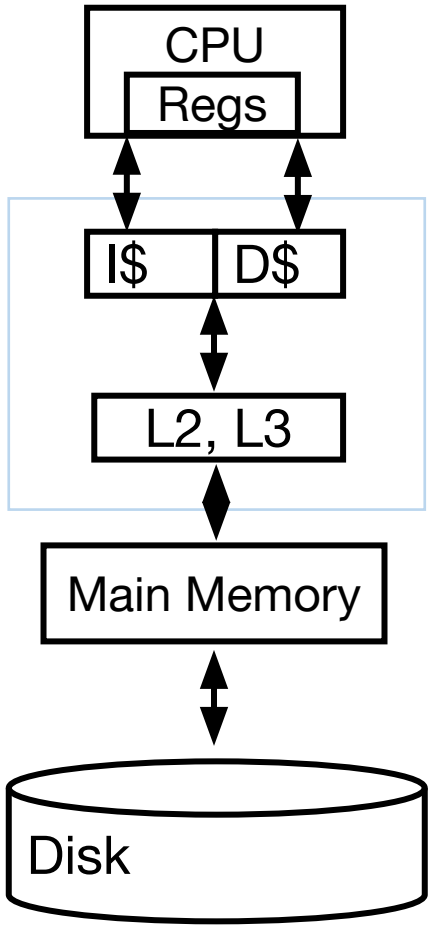
Cache

Storing temporary data close to the CPU.



Multi-Level Cache

Higher level → bigger but slower
(still both smaller and faster than the main memory.)



Cache Performance

- Average time to access the cache:

  AAT = HitTime + MissRate × MissPenalty

  - HitTime: Time it takes to access the (L1) cache.
  - MissRate: The average frequency of misses (in L1).
  - MissPenalty: The time required to access the main memory.

Cache Performance

- Average time to access the cache:

  AAT = HitTime + MissRate × MissPenalty

  - HitTime: Time it takes to access the (L1) cache.
  - MissRate: The average frequency of misses (in L1).
  - MissPenalty: The time required to access the main memory.

- *What if there are multiple levels?*

Cache Performance

multi-level

$$AAT = HitTime + MissRate \times MissPenalty$$

Example

Hit time (L1) = 2 cycles
Miss rate (L1) = 0.05
Hit time (L2) = 4 cycles
Miss rate (L2) = 0.01
Miss penalty (L2) = 10 cycles
Miss penalty (L1) = ?
→ AAT (with L2) = ?
→ AAT (without L2) = ?

Inclusive vs. Exclusive Cache

-- Inclusive: L_i is a subset of L_{i+1}
- Easier to find data
- Wasted capacity

-- Exclusive: Data is ***only*** in one of the levels.
- Difficult to find data
- Efficient capacity

Modern Designs

- Split vs Unified "caches"
  - L1 I/D caches commonly split and asymmetrical
    - Double bandwidth and no-cross pollution on disjoint I and D footprints
    - i-cache is smaller, simpler with more spatial locality. Usually a prefetcher and/or trace cache is connected to i-cache.
  - L2 and L3 are unified for simplicity.

Harvard vs. Princeton

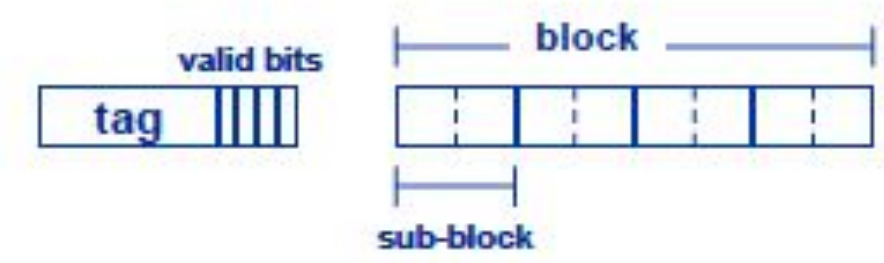
- "Harvard" referred to a microarchitecture with ***separate*** instruction and data memory.

- "Princeton" referred to von Neumann's ***unified*** instruction and data memory. This is the most common design.

Reducing Miss Penalty
sub-blocking

- Higher block size improves miss rate but increase miss penalty!

★ ***Idea:*** keep a large block size, but divide it into smaller *"subblocks"*. Bring only a subset of subblocks on a miss.



Reducing Miss Penalty
sub-blocking

- Subblocks:
  - Lower miss rate.
  - Lower miss penalty.
  - Need separate storage for valid bits for each subblock.
  - More complex circuitry.

How to reduce hit time?

Reducing Hit Time

reduce associativity and size

- DM < FA
  - ➔ Use SA to balance between the two

- Use smaller cache in lower levels (L1, L2, …)

Reducing Hit Time

parallel lookup

- Access tag and data in parallel.

- Access each way in parallel.

Reducing Hit Time

speculative load

- *Instead of waiting for a store (potentially conflicting), issue the load speculatively.*
  - Once store is resolved, check whether there was a conflict or not. Recover if there was.

Summary

- Miss Rate
  - Increase block size
  - Increase associativity
  - Increase cache size
  - Prefetching
  - Victim cache
  - Compiler
  - Replacement Policy
- Miss Penalty
  - Write buffer
  - Early restart with critical block first
  - Adding more levels
  - Sub-blocking
- Hit Time
  - Set associative cache
  - Add more levels
  - Parallel lookup
  - Speculative loads

End of Presentation

Acknowledgement

- This course is partly inspired by the following courses created by my colleagues:
  - CS152, Krste Asanovic (UCB)
  - 18-447, James C. Hoe (CMU)
  - CSE141, Steven Swanson (UCSD)
  - CIS 501, Joe Devietti (Upenn)
  - CS4290, Tom Conte (Georgia Tech)
  - 252-0028-00L, Onur Mutlu (ETH)