**Cache**

[Memory Hierarchy]

* Processor

• executes programs

• runs on order of nanoseconds to picoseconds

• needs to access code and data for programs: where are these?

* Disk

• HUGE capacity (virtually limitless)

• VERY slow: runs on order of milliseconds

• so how do we account for this gap?

* Memory (DRAM)

• smaller than disk (not limitless capacity)

• contains subset of data on disk: basically portions of programs that are currently being run

• much faster than disk: memory accesses don’t slow down processor quite as much

• Problem: memory is still too slow (hundreds of nanoseconds)

• Solution: add more layers (***caches***)

If level is closer to Processor, it must...

• Be smaller

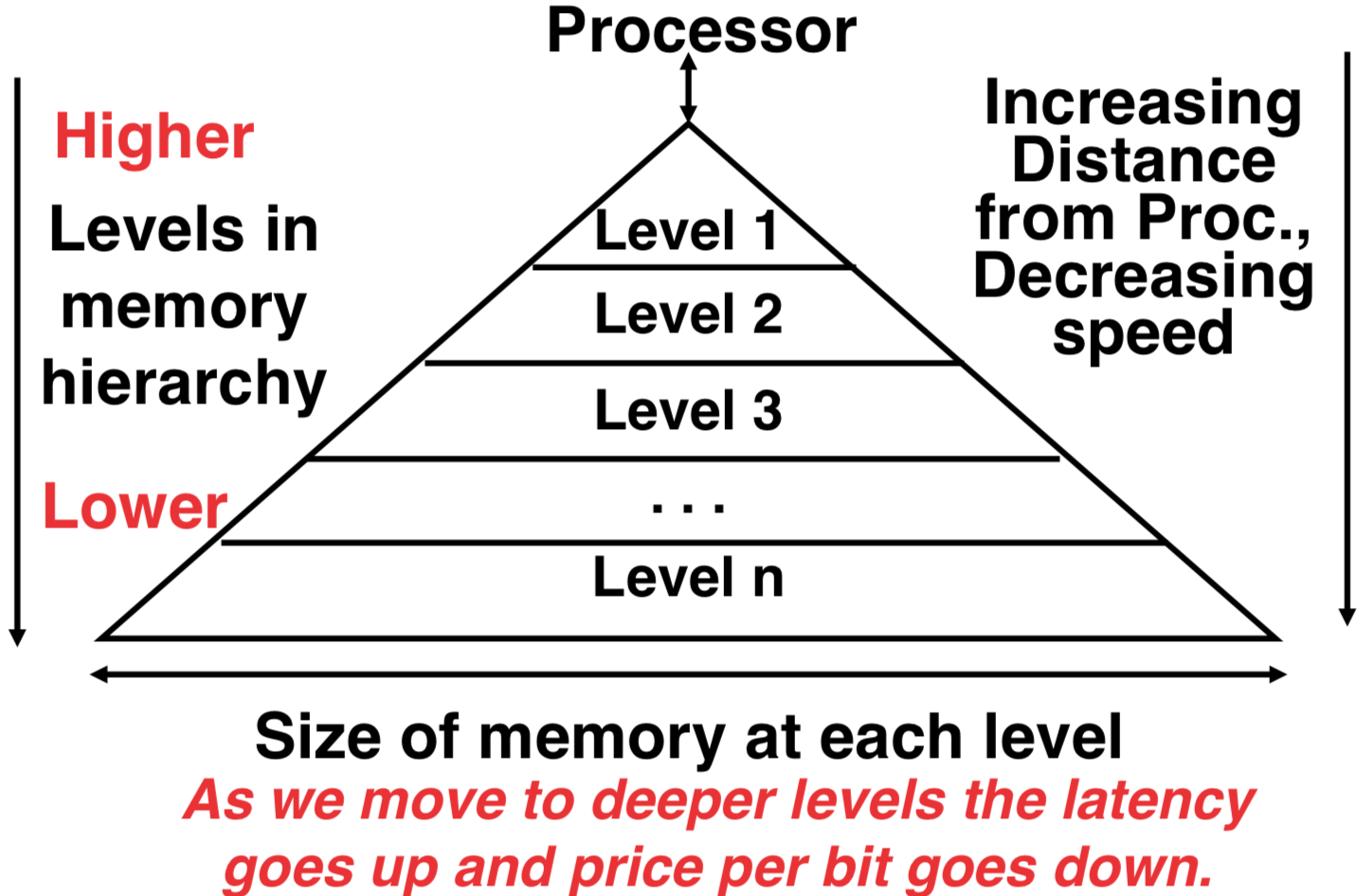
• Be faster

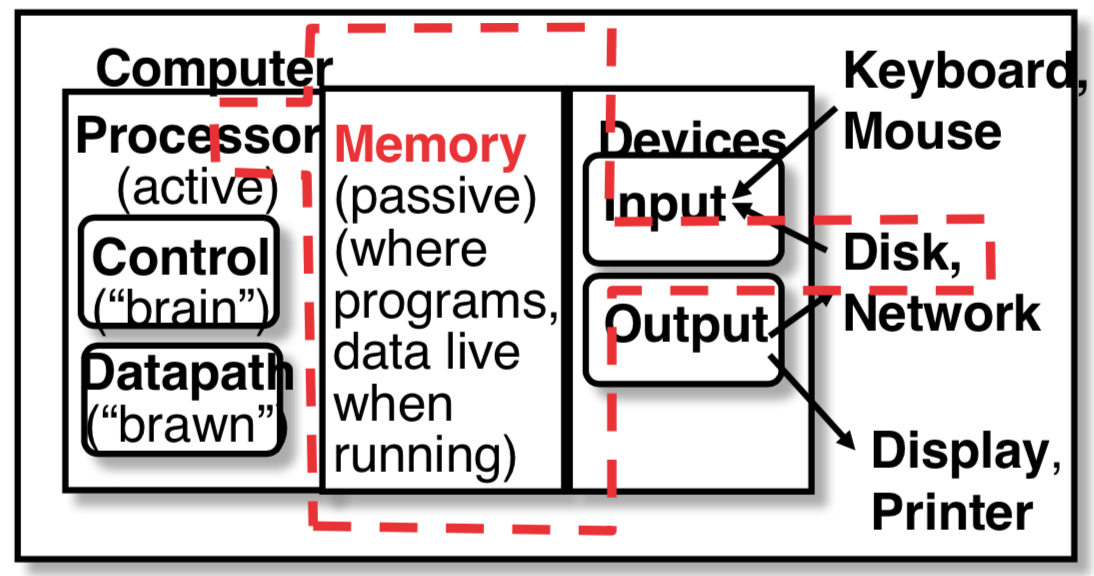
• Contain a subset (most recently used data)

of lower levels beneath it

• Contain all the data in higher levels above it

Lowest Level (usually disk) contains all available data. Is there another level lower than disk? Internet!





Purpose: Faster access to large memory from processor

Analogy: You’re writing a term paper (**processor**) at a table in Schulich

Schulich Library is equivalent to **disk**

• essentially limitless capacity

• very slow to retrieve a book

Table is **memory**

• smaller capacity: means you must return book when table fills up

• easier and faster to find a book there once you’ve already retrieved it

Open books on table are **cache**

• smaller capacity: very few open books can fit on table; when table fills up, must close a book

• much, much faster to retrieve data

Disk contains everything.

When Processor needs something, bring it into all lower levels of memory.

Cache contains copies of data in memory that are being used.

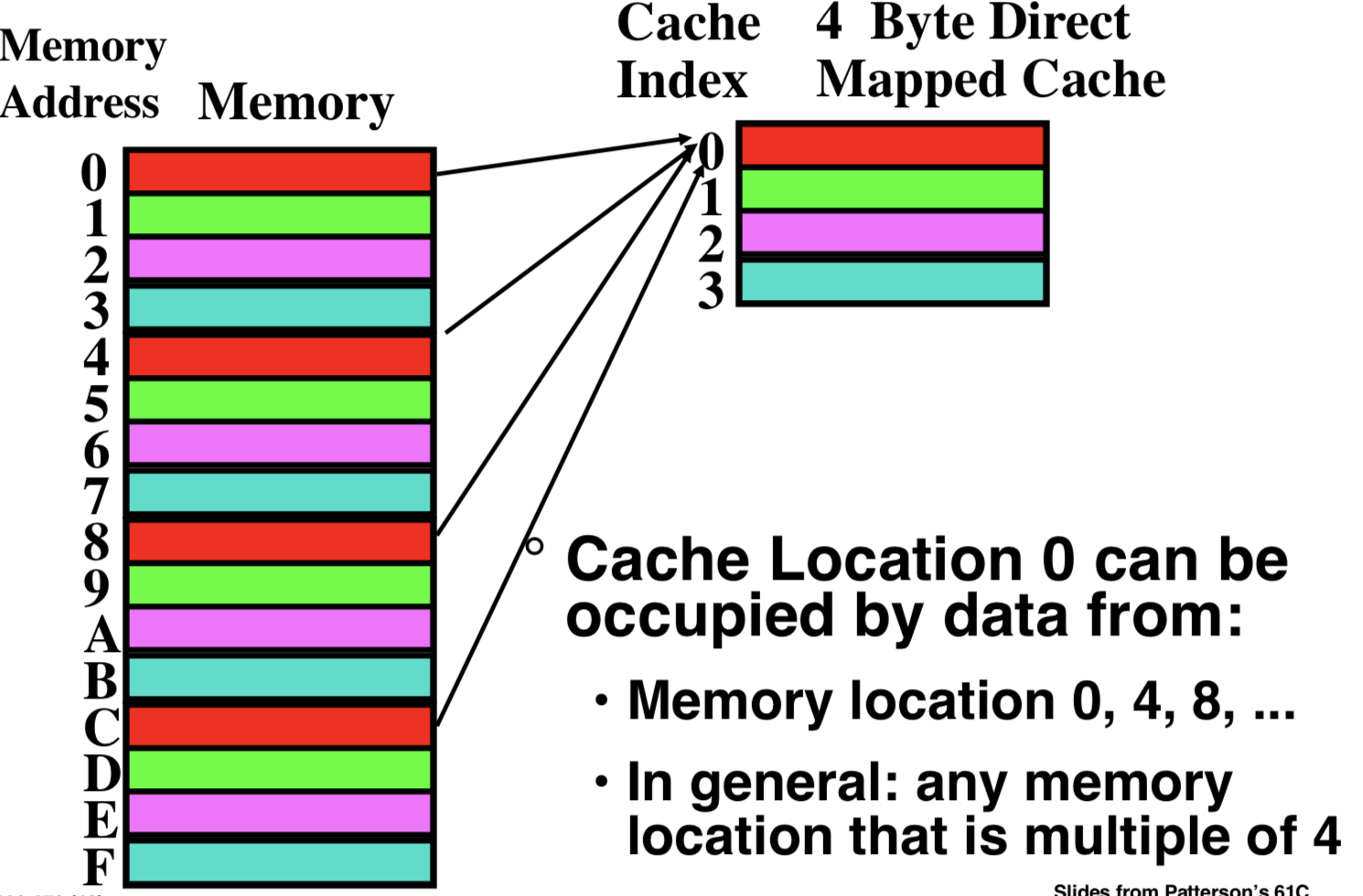
Memory contains copies of data on disk that are being used.

Entire idea is based on *Temporal Locality*: if we use it now, we’ll want to use it again soon

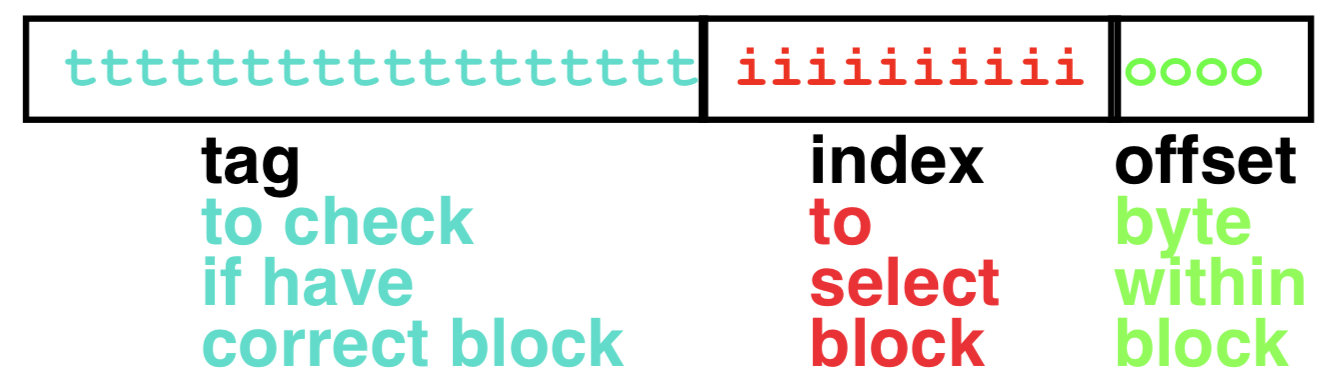
**Direct-Mapped Cache**

In a direct-mapped cache, *each memory address is associated with one possible* ***block*** *within the cache.* Therefore, we only need to look in a single location in the cache for the data if it exists in the cache

[ ***Block*** *is the unit of transfer between cache and memory.* ]

**

Since multiple memory addresses map to same cache index, how do we tell which one is in there? 🡪 Divide memory address into three fields



All fields are read as unsigned integers.

Index: specifies the cache index (which “row” of the cache we should look in)

Offset: once we’ve found correct block, specifies which byte within the block we want

Tag: the remaining bits after offset and index are determined; these are used to distinguish between all the memory addresses that map to the same location

Suppose we have a 16KB direct-mapped cache with 4 word blocks.

Determine the size of the tag, index and offset fields if we’re using a 32-bit architecture.

Offset

• need to specify correct byte within a block

• block contains 4 words = 16 bytes = 24 bytes

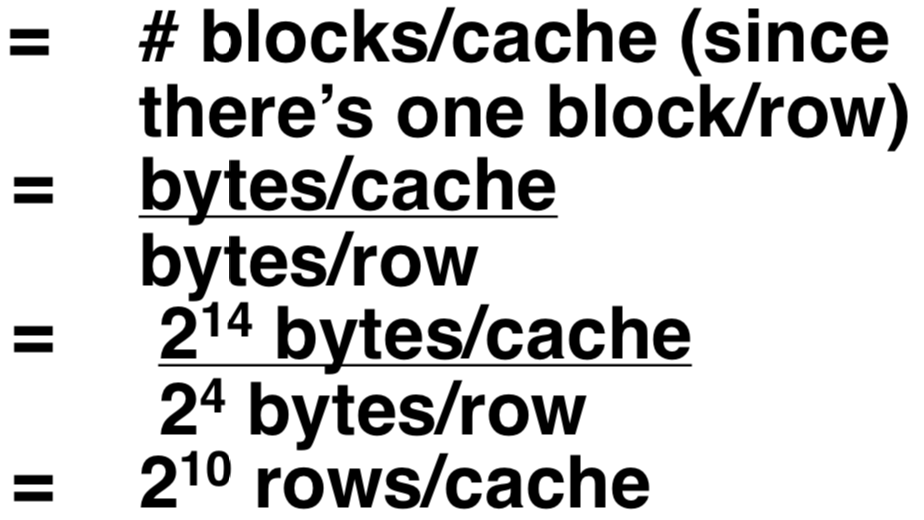
• need 4 bits to specify correct byte

Index

• need to specify correct row in cache

• cache contains 16 KB = 24 210 = 214 bytes

block contains 24 bytes (4 words)

• # rows/cache

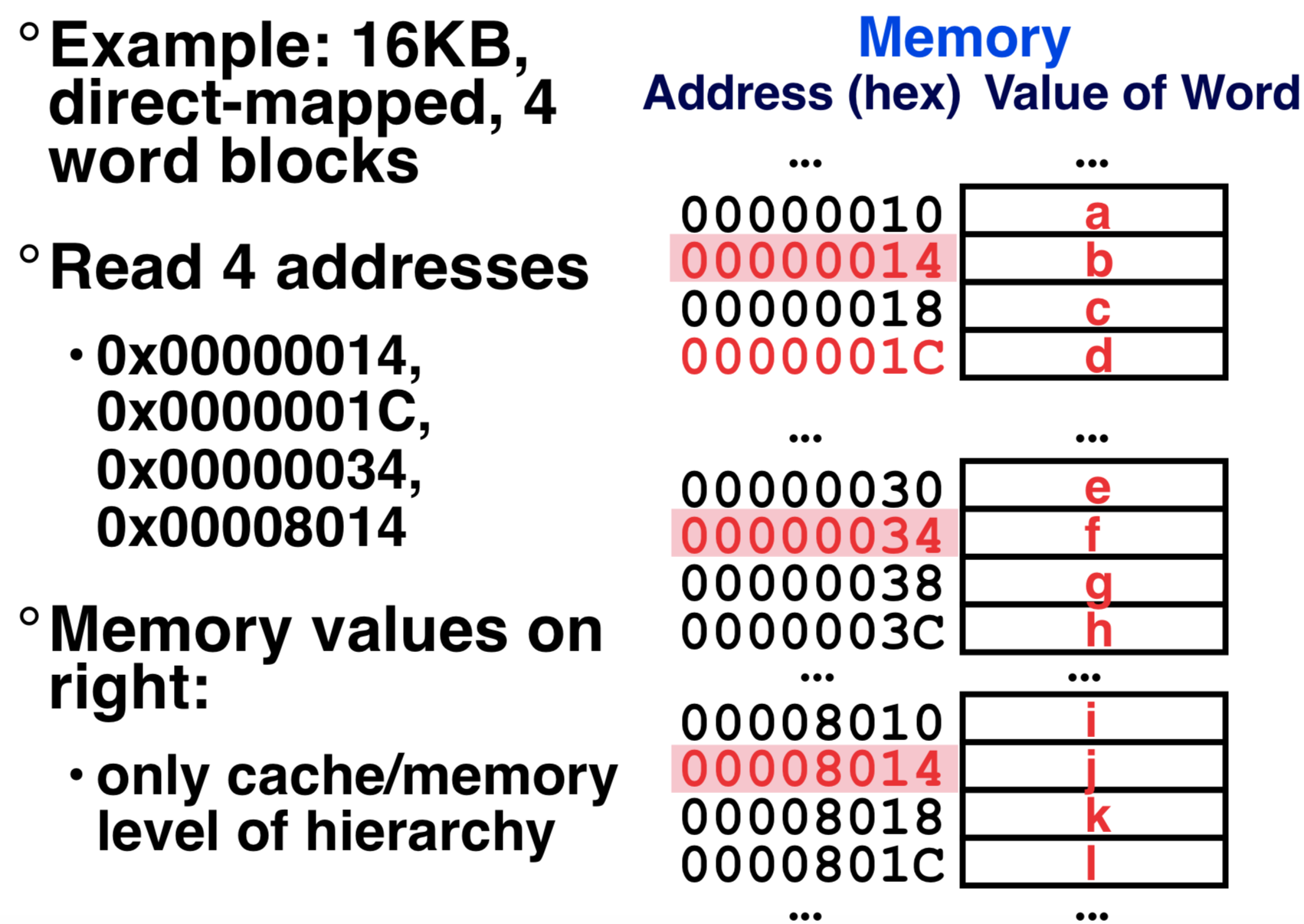
• need 10 bits to specify this many rows

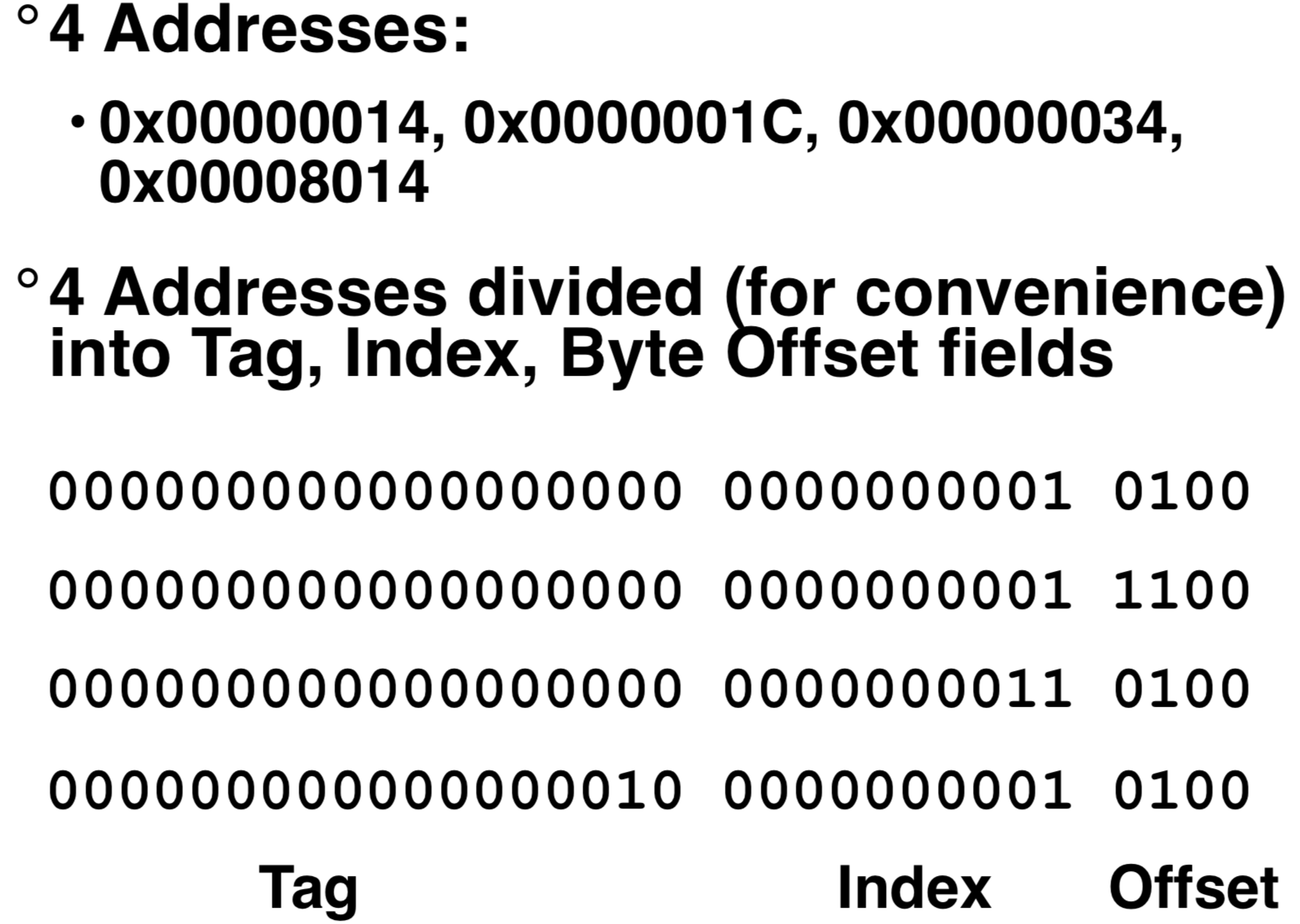
Tag

• used remaining bits as tag

• tag length = mem addr length – offset – index = 32 - 4 - 10 bits = 18 bits

• so tag is leftmost 18 bits of memory address





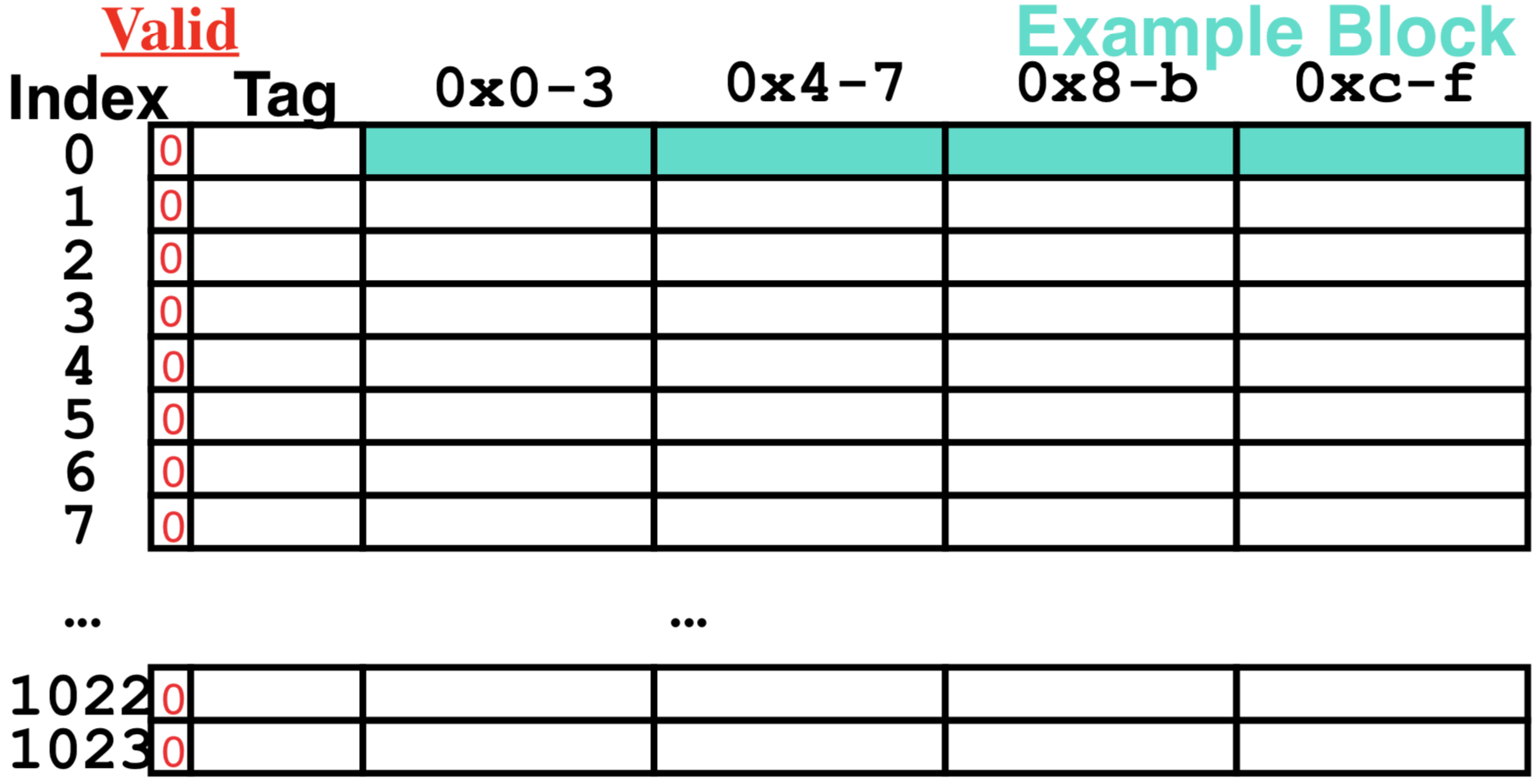
So lets go through accessing some data in this cache

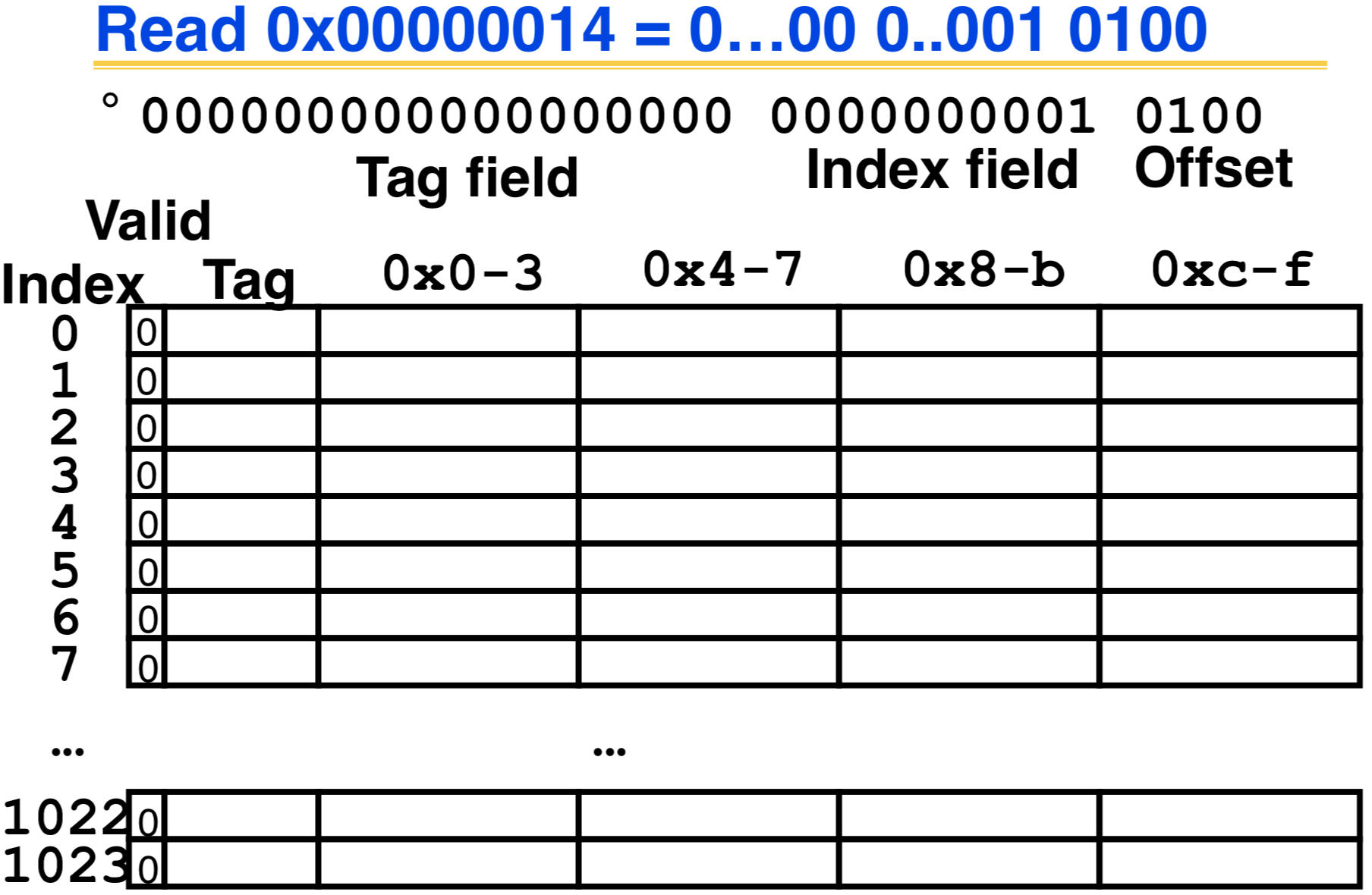
• 16KB, direct-mapped, 4 word blocks

Will see 3 types of events:

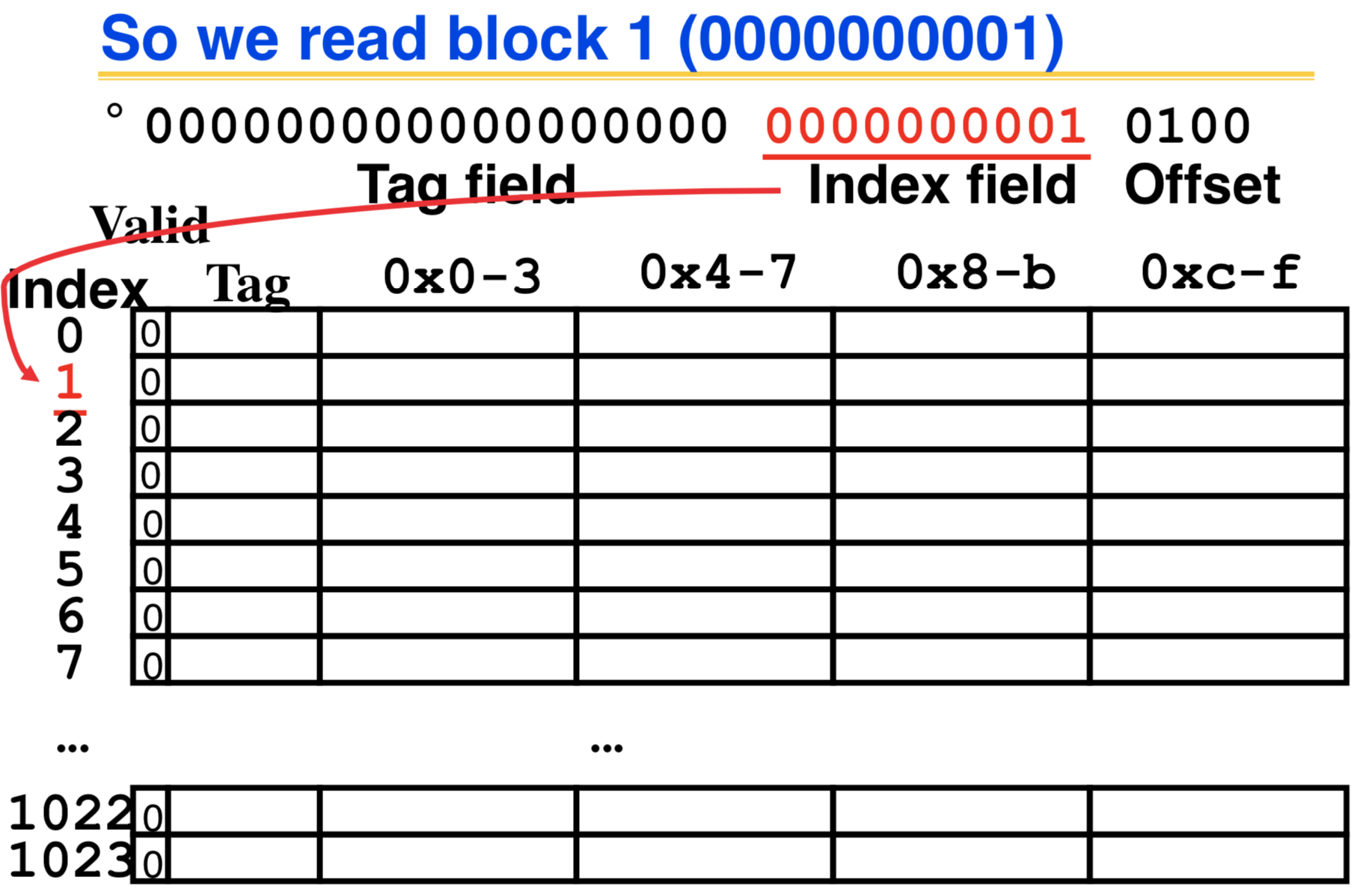
* ***cache miss****: nothing in cache in appropriate block, so fetch from memory*
* ***cache hit****: cache block is valid and contains proper address, so read desired word*
* ***cache miss, block replacement****: wrong data is in cache at appropriate block, so discard it and fetch desired data from memory*

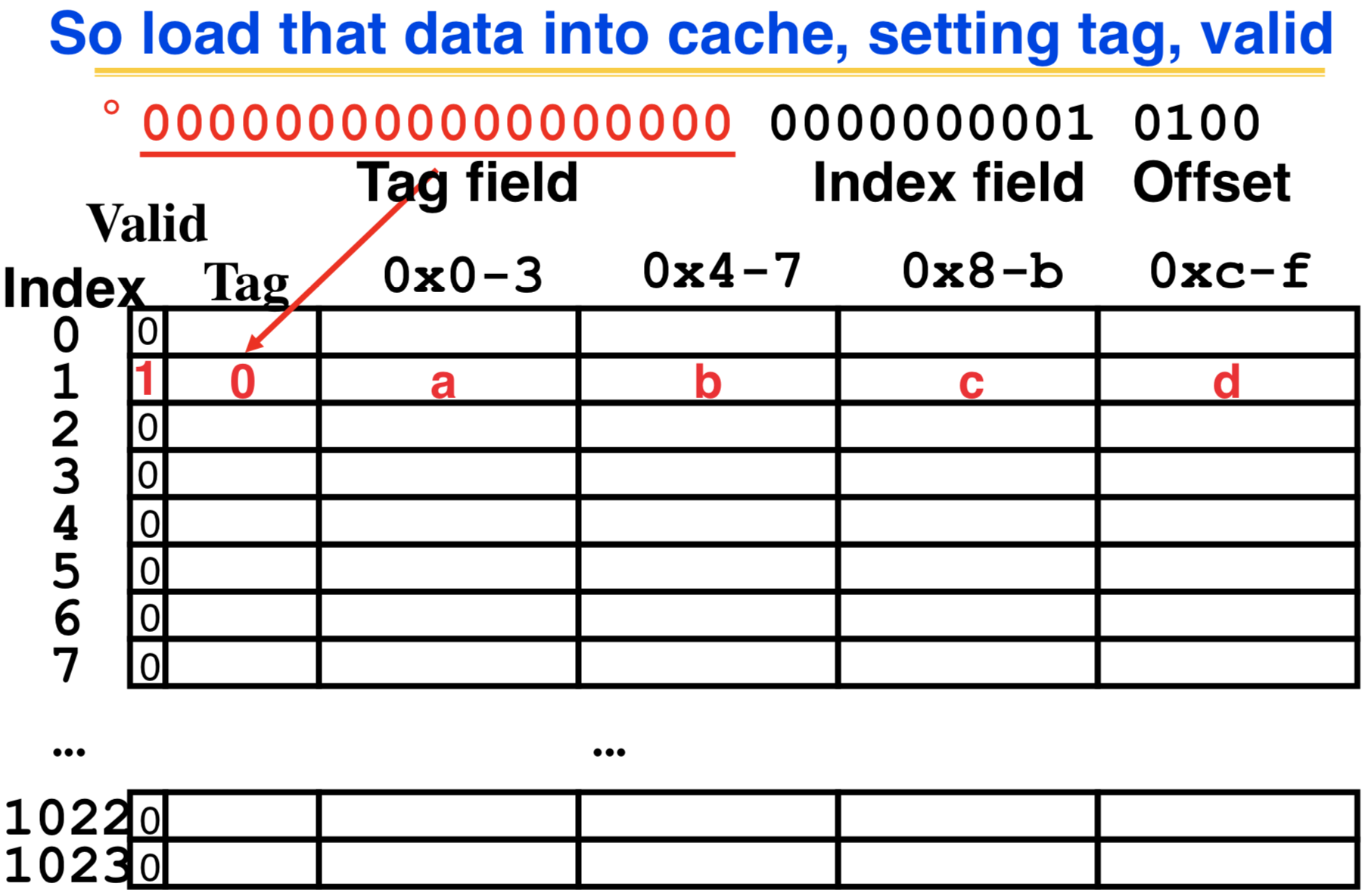
***Valid bit****: determines whether anything is stored in that row (when computer initially turned on, all entries are invalid)*

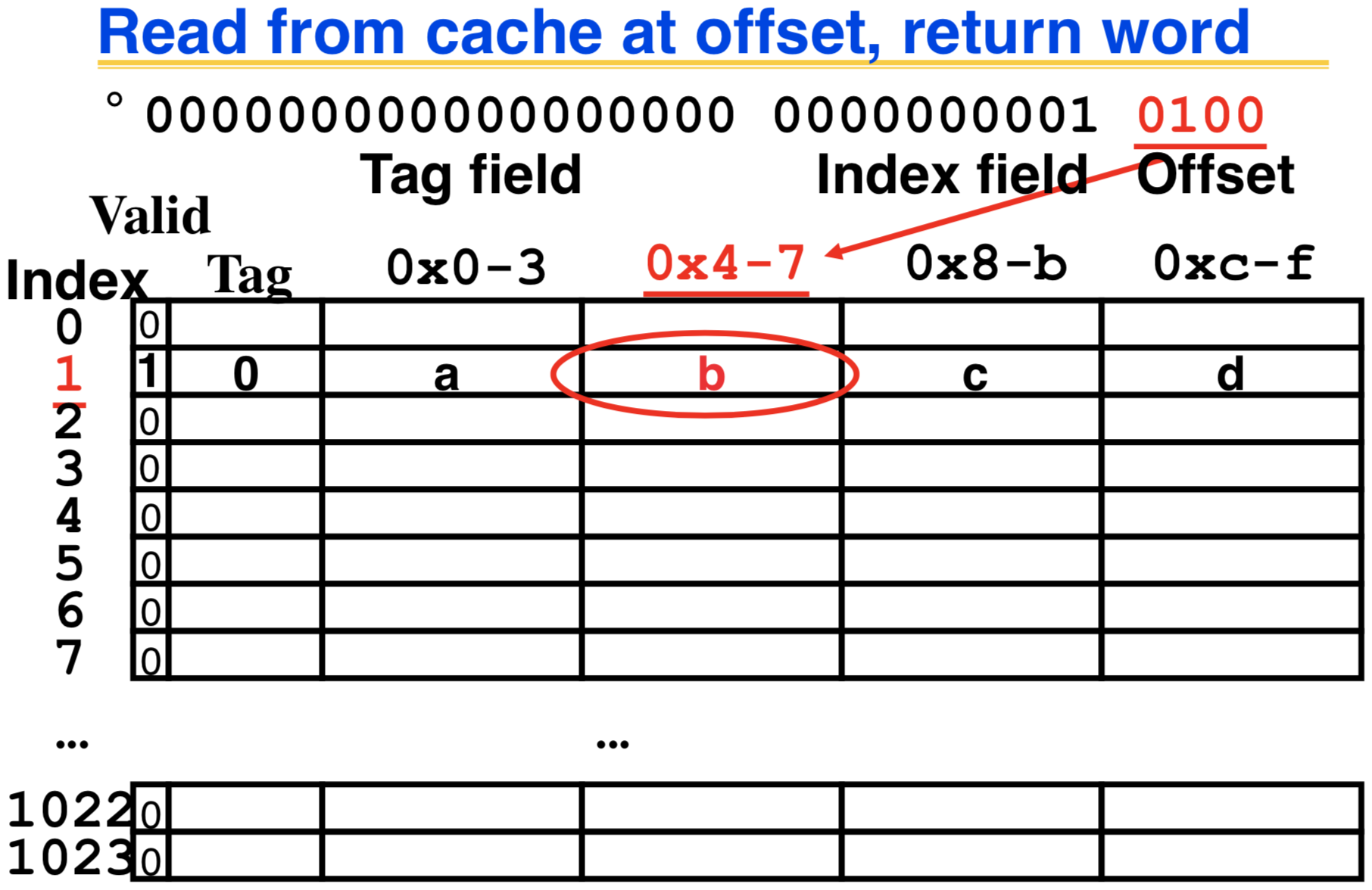




NO valid data 🡪 cache miss





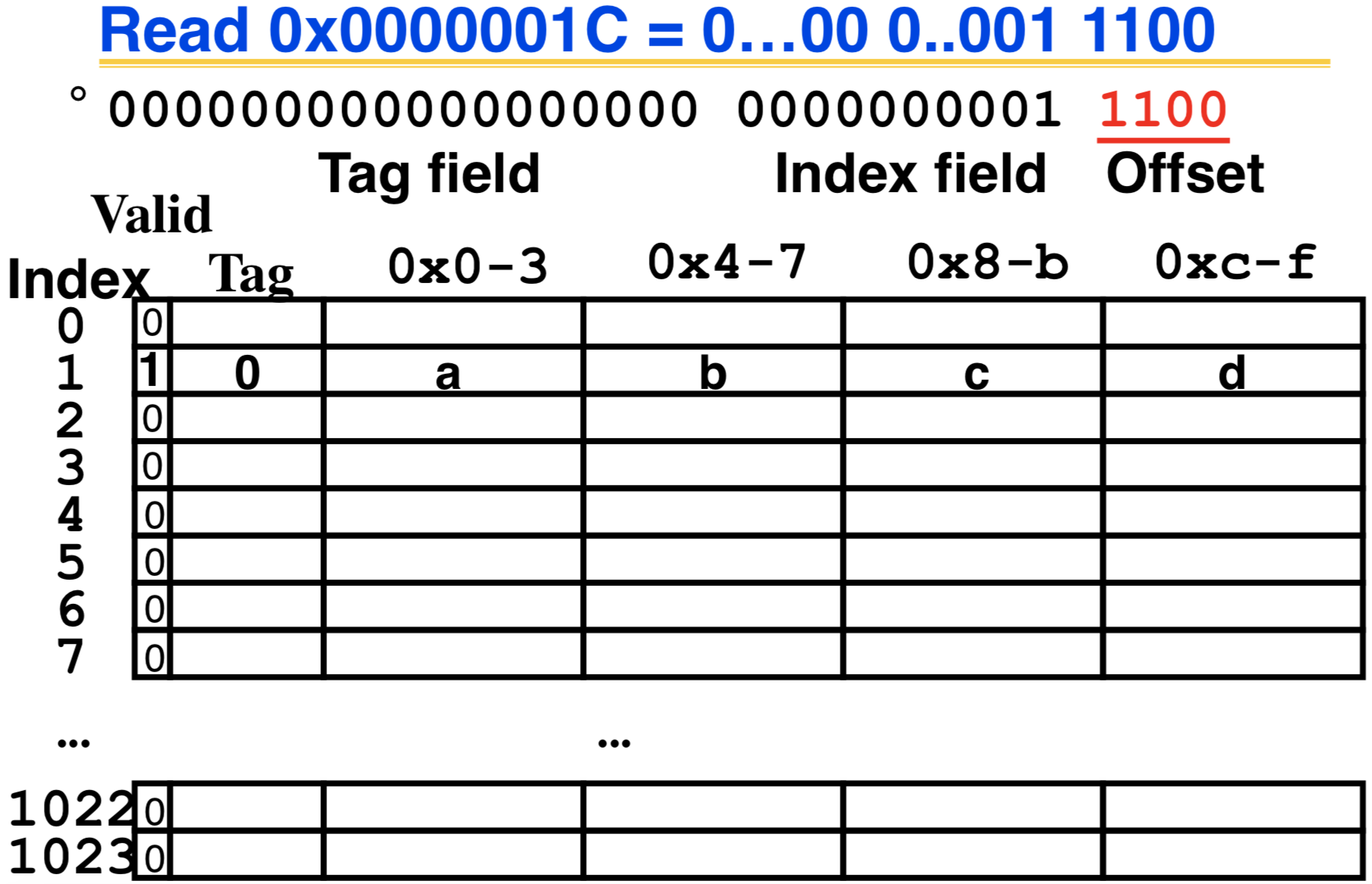


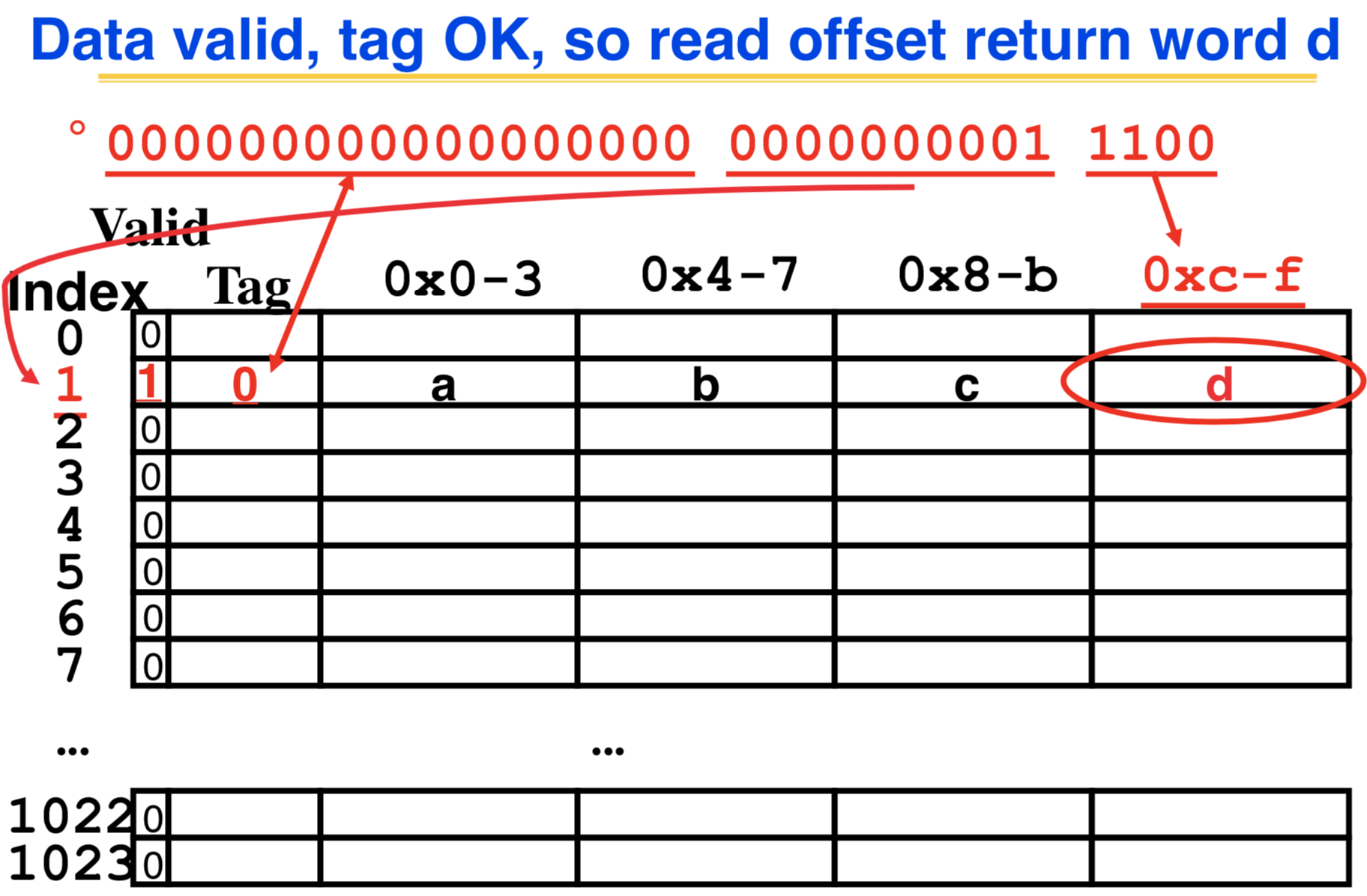
data in the row + tag matched🡪

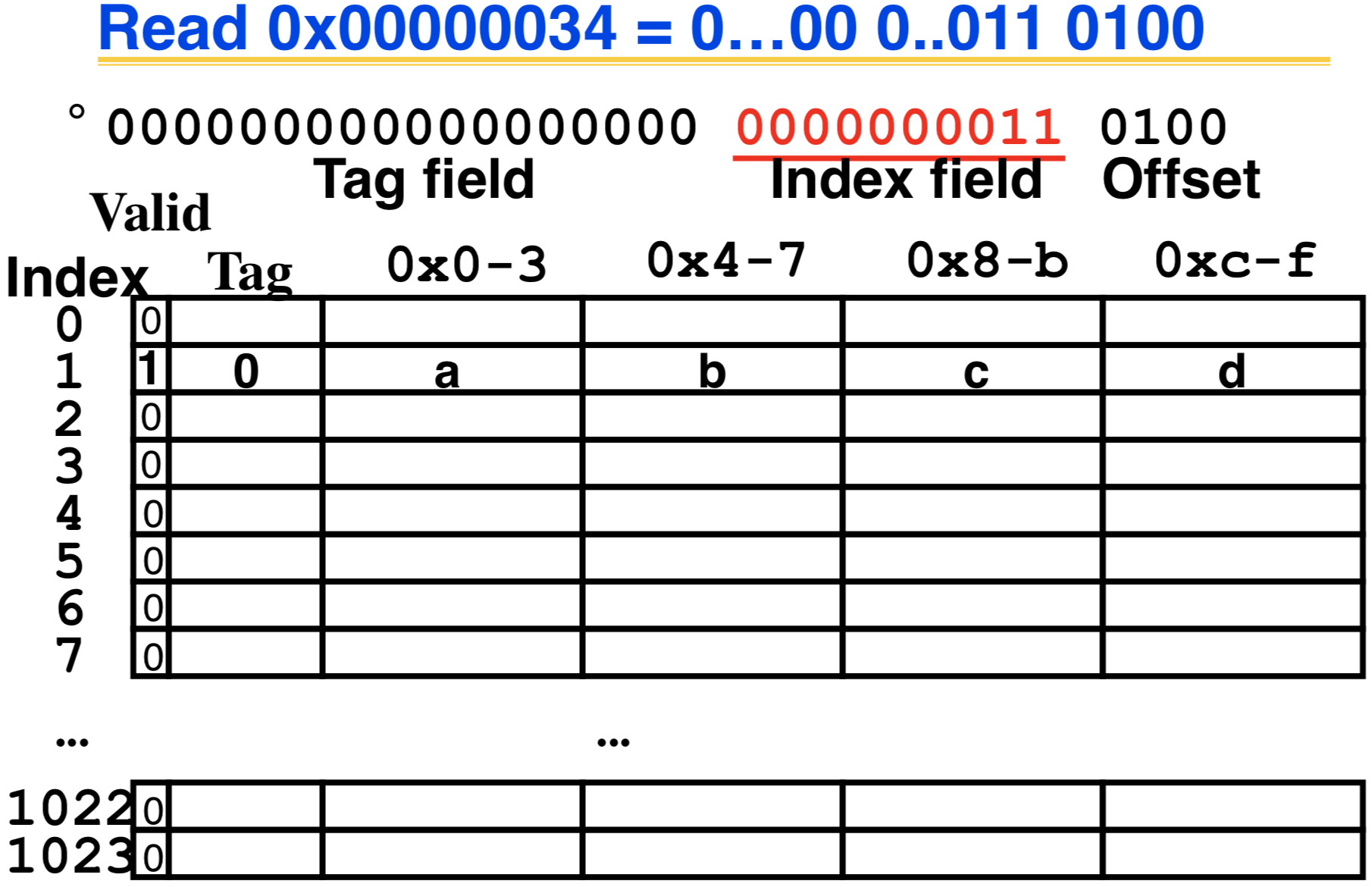
cache hit

data in the row + tag matched🡪

cache hit





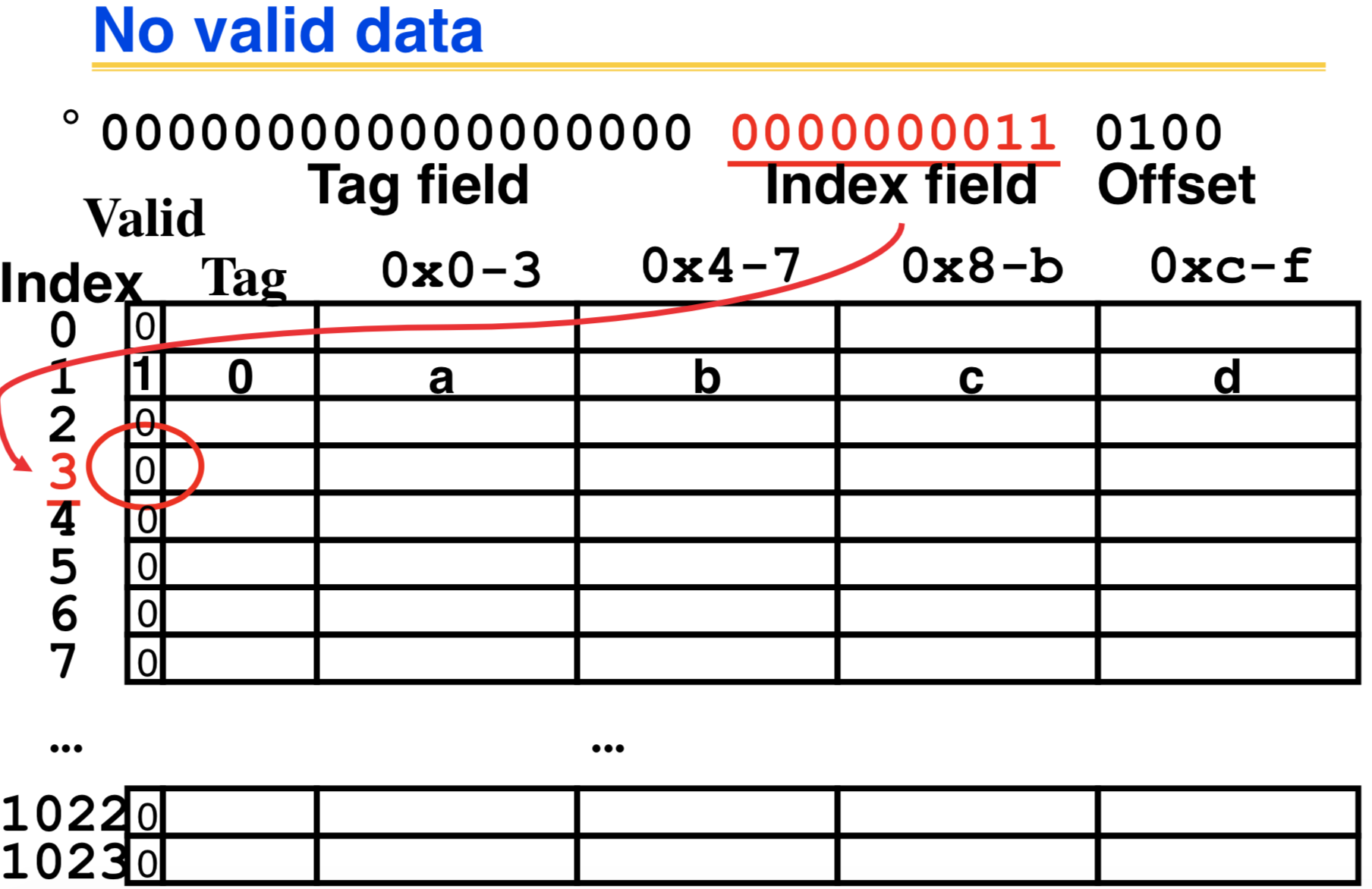


data in the row + tag matched🡪

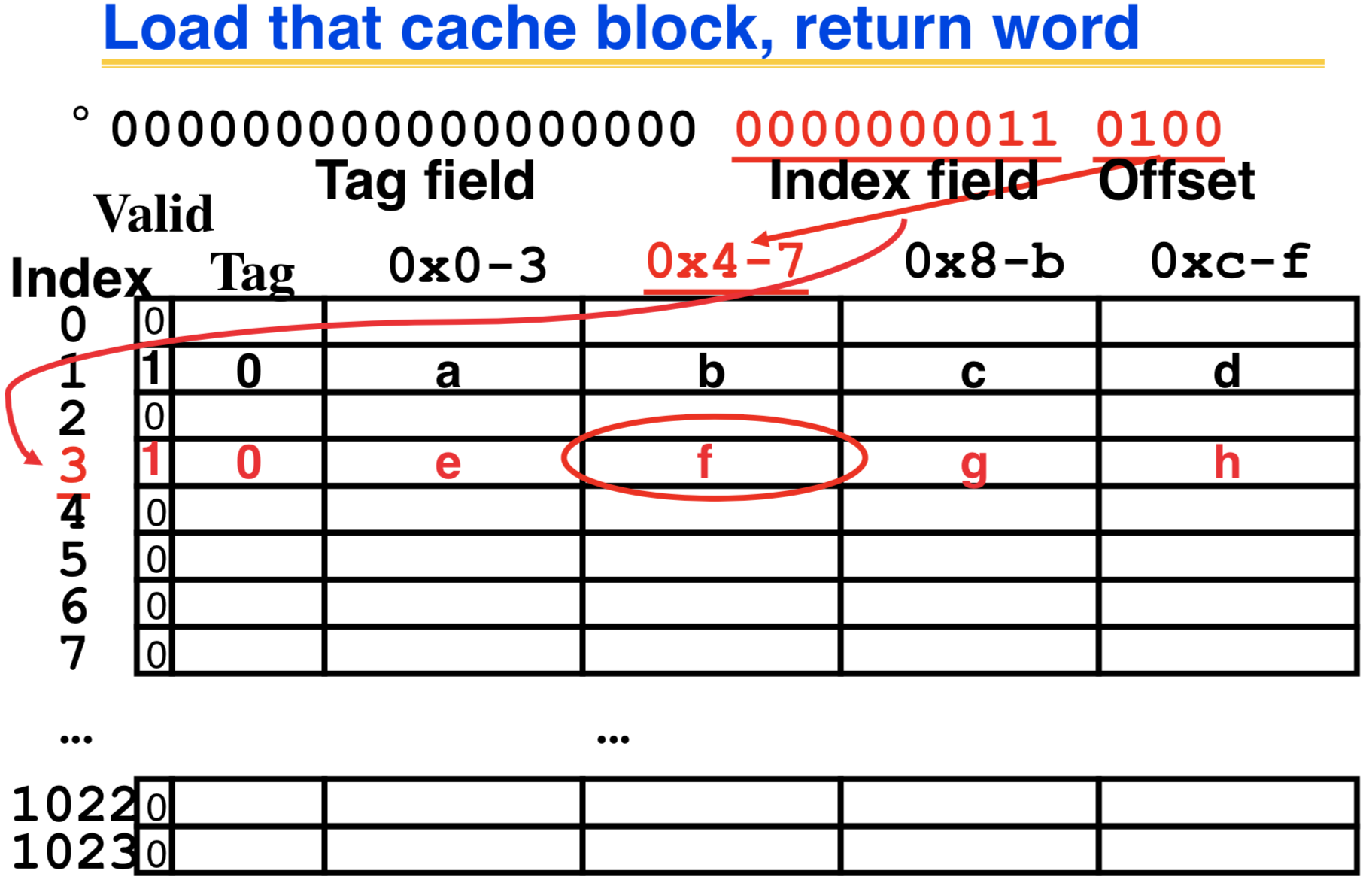
cache hit

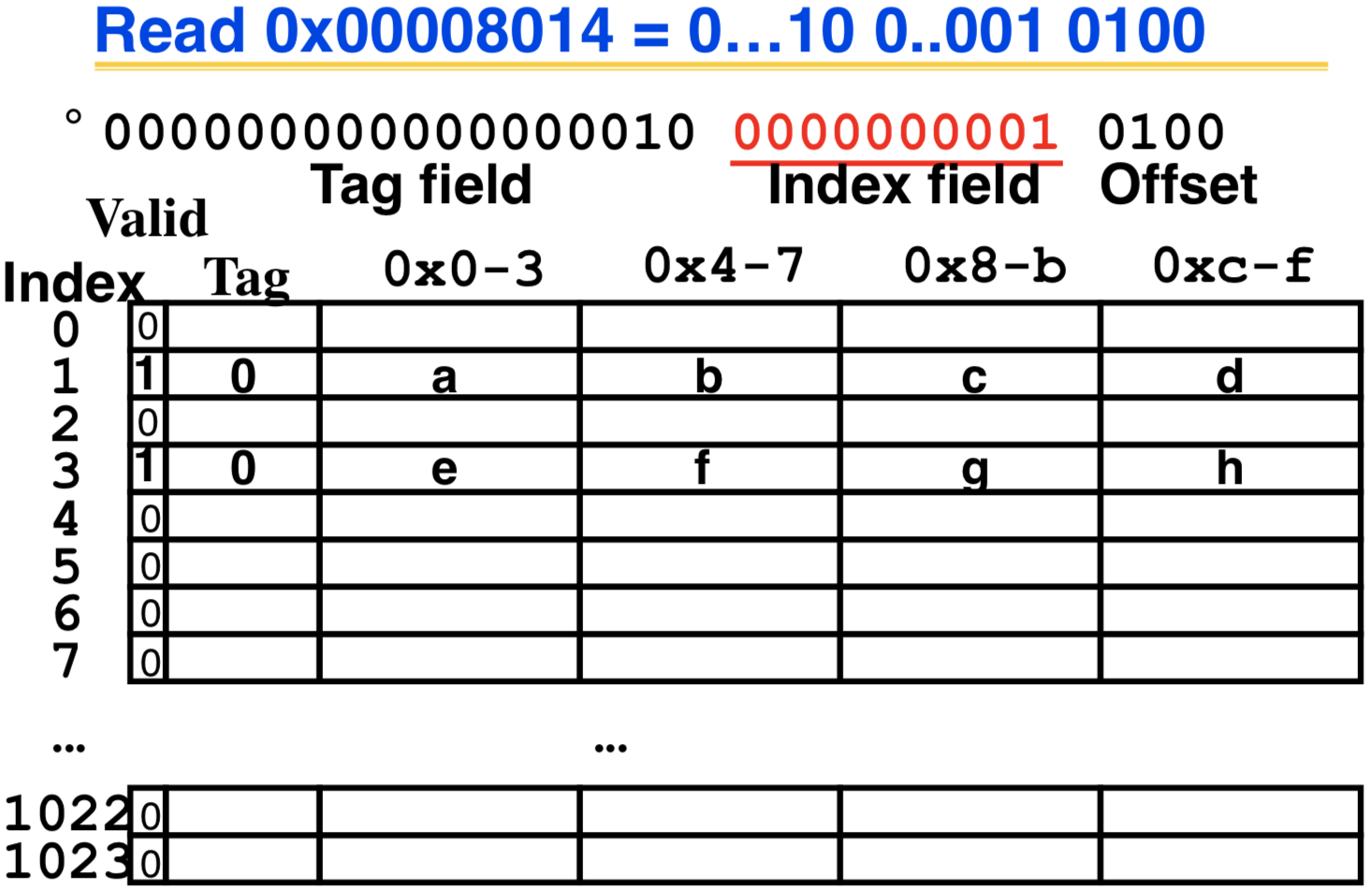
**So read**

**block 3**



cache miss

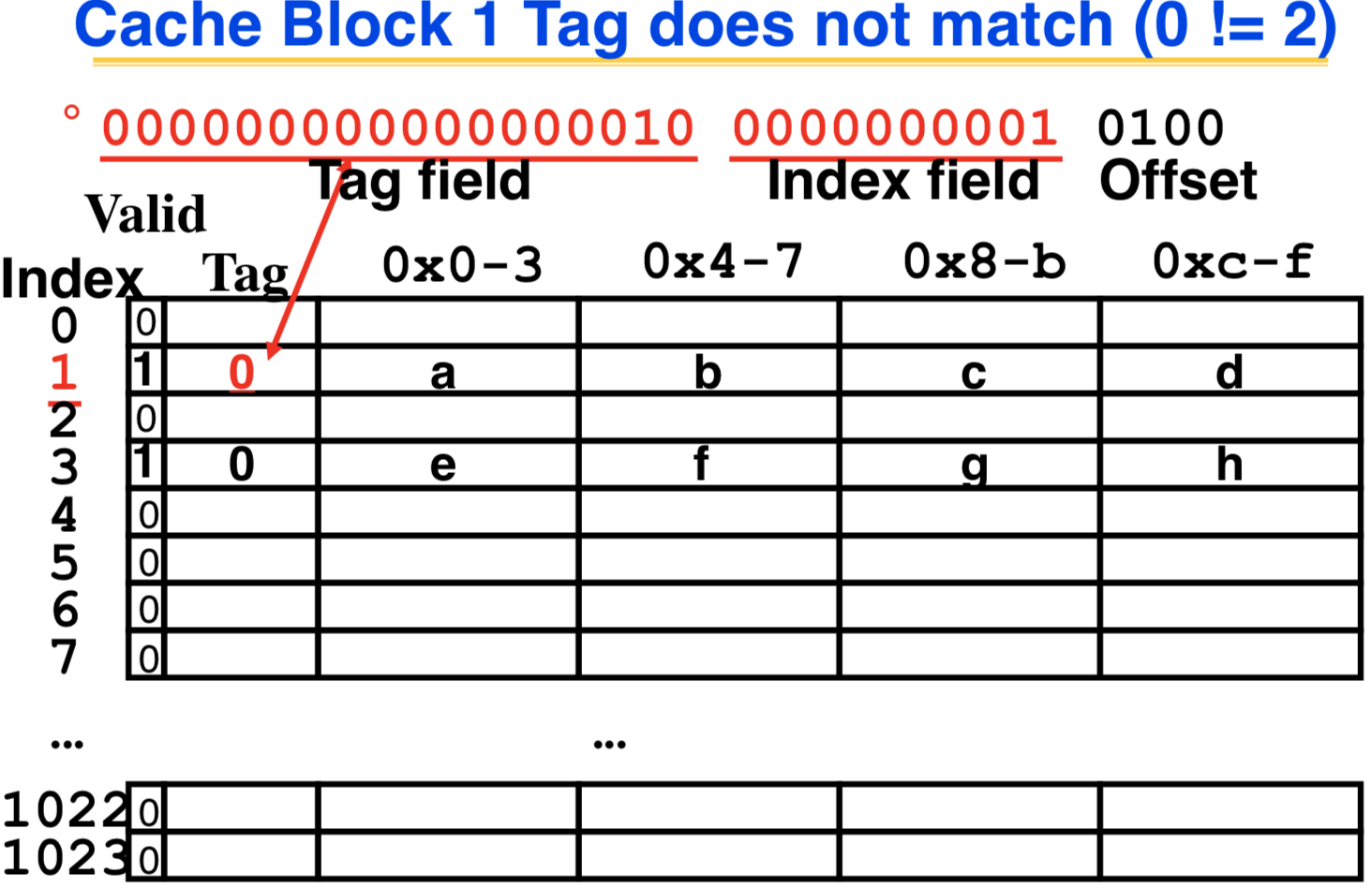


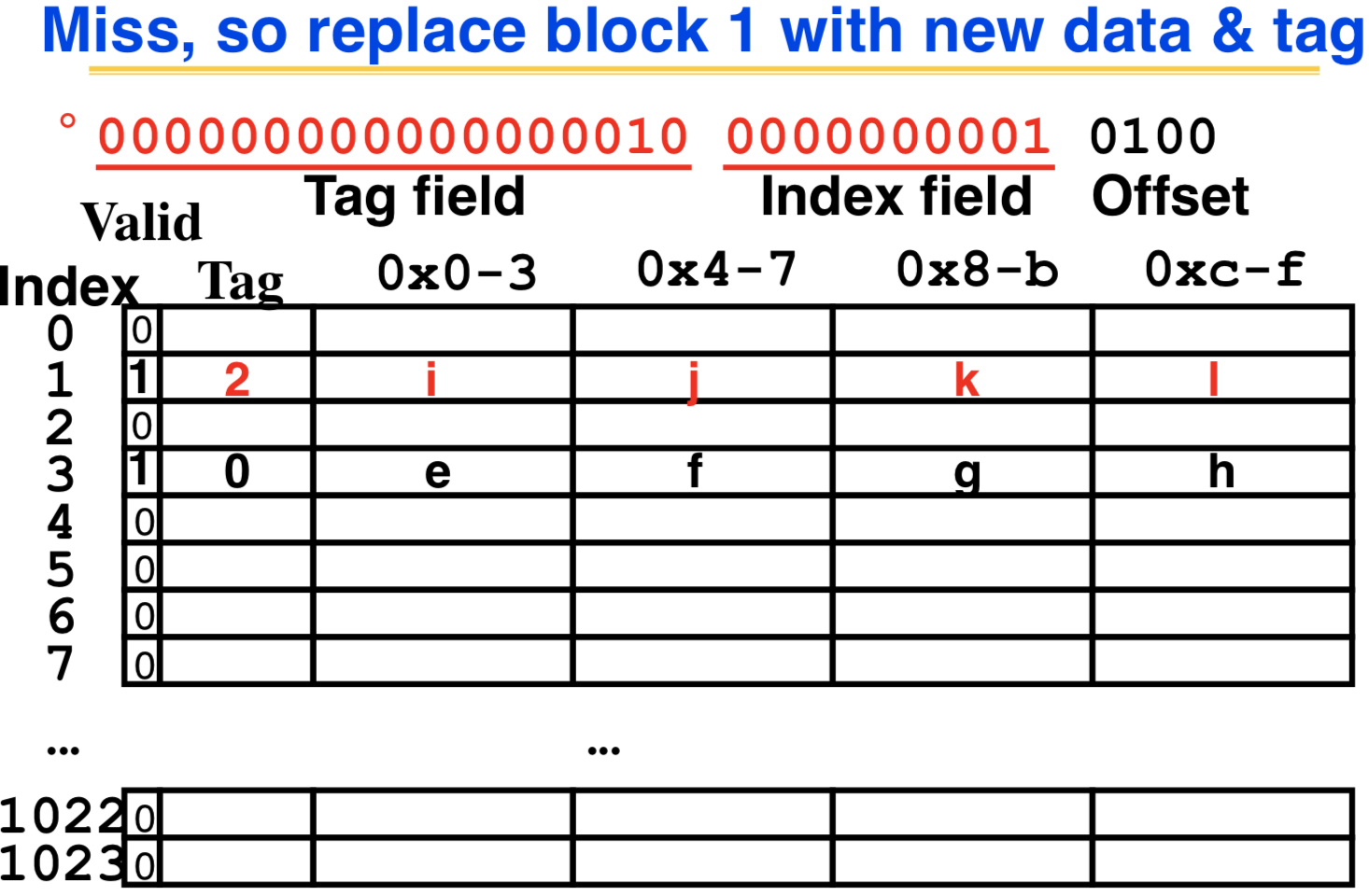


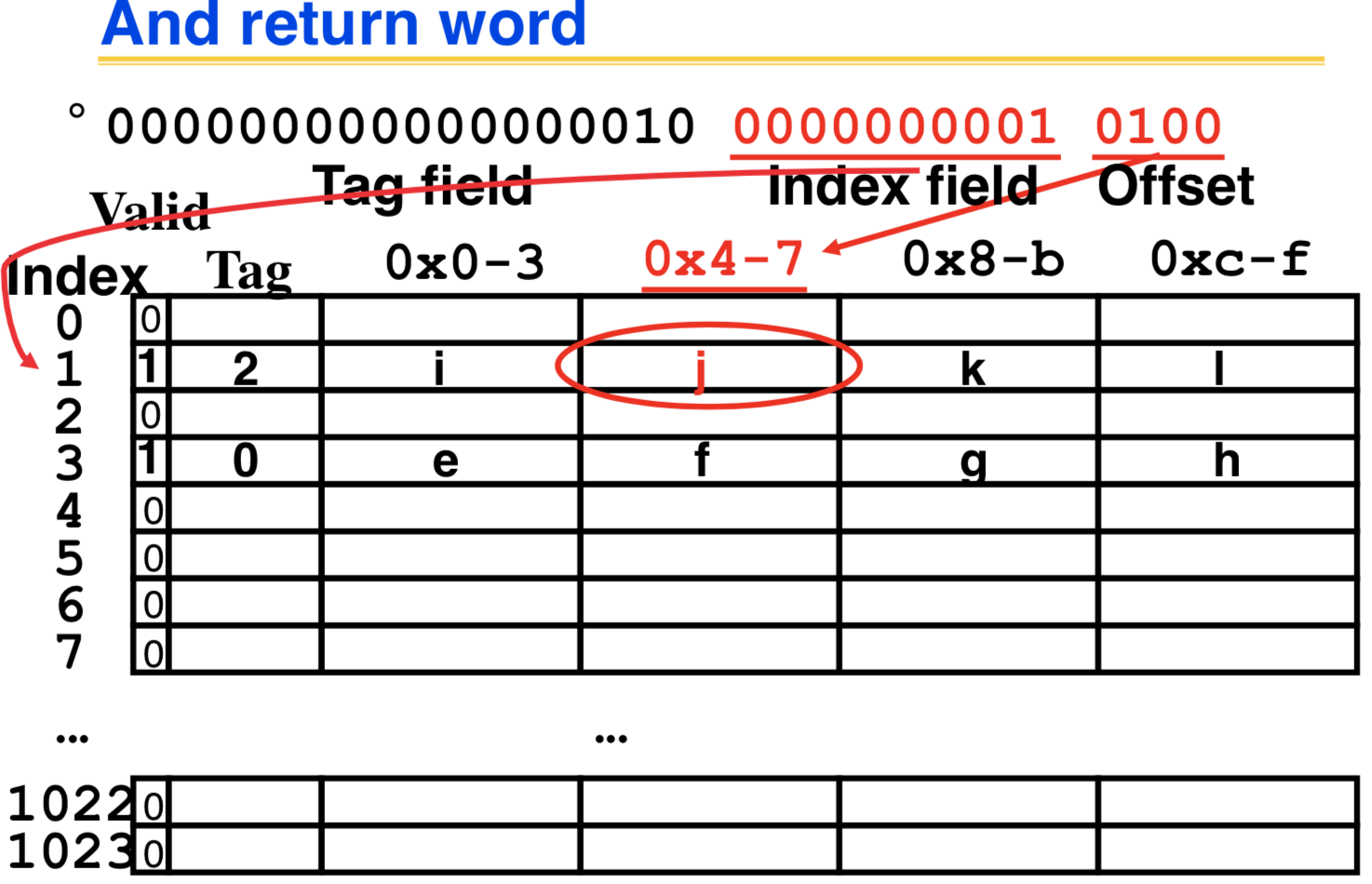
data in the row + tag matched🡪

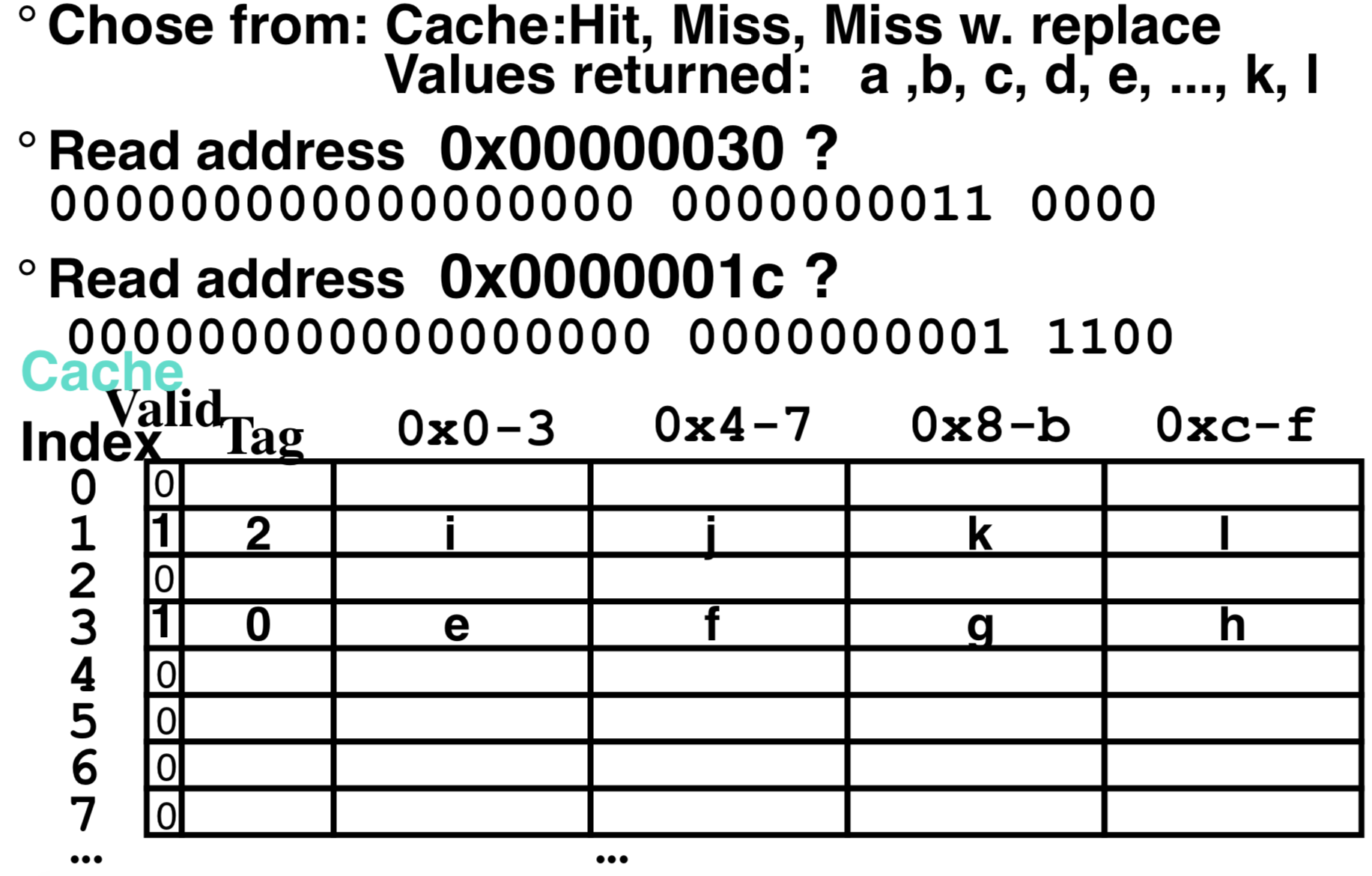
cache hit

So read Cache Block 1, Data is valid











0 a b c d

cache miss with replacement

Limitations:

1. Lots of “conflict” misses or misses with replacement (even though cache is not full)

2. 1 Gb array! Capacity miss

* Direct-mapped cache🡪index tells which row to go[extreme-exactly one row]
* Fully associated cache🡪no index bits, just check all rows[extreme-any row]

**Block Size Tradeoff**

Benefits of Larger Block Size

• *Spatial Locality: if we access a given word, we’re likely to access other nearby words soon*

• Very applicable with Stored-Program Concept: if we execute a given instruction, it’s likely that we’ll execute the next few as well

• Works nicely in sequential array accesses too

Drawbacks of Larger Block Size

• Larger block size means larger miss penalty

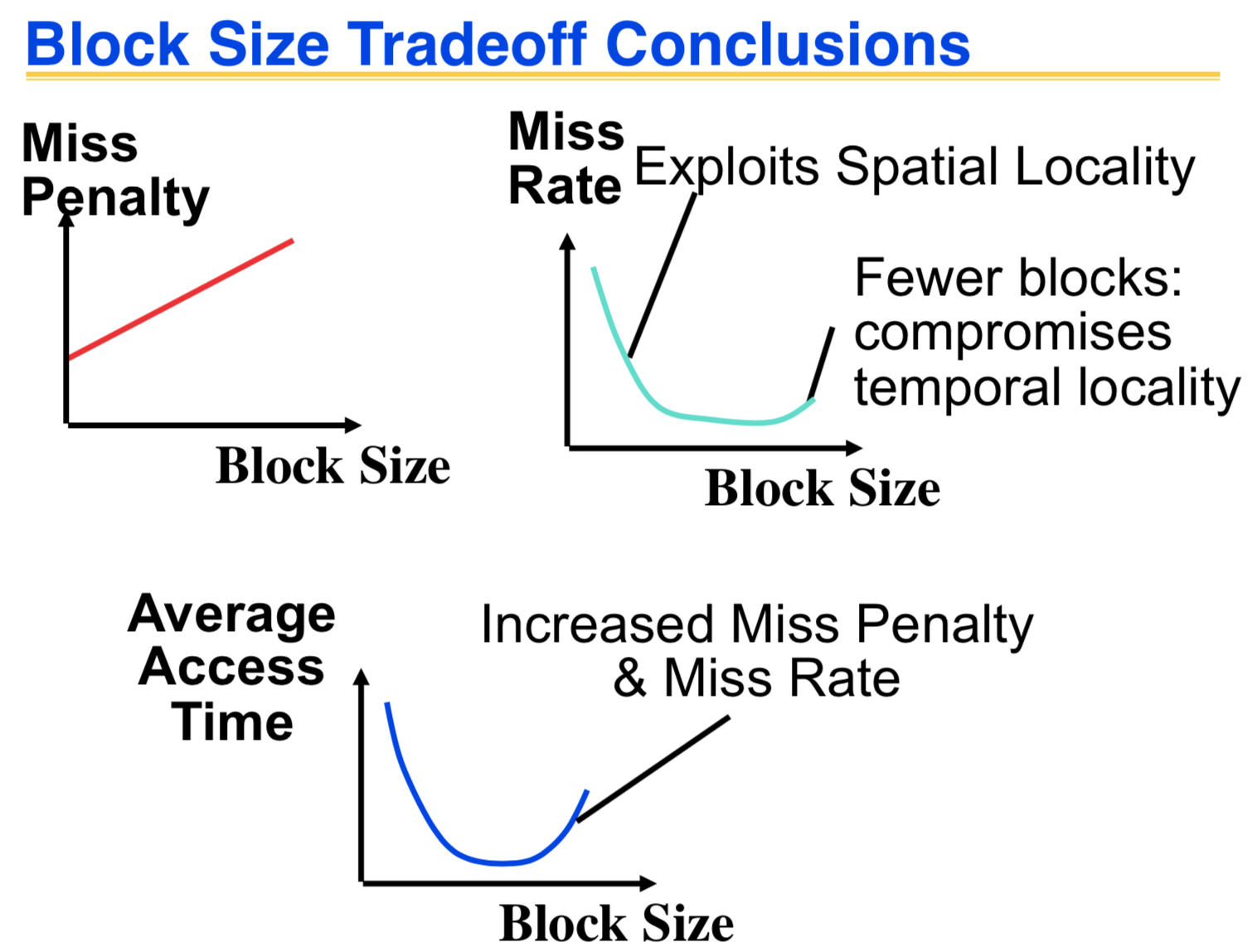
On a miss, takes longer time to load a new block from next level

• If block size is too big relative to cache size, then there are too few blocks

Result: miss rate goes up

Average Access Time = Hit Time + Miss Penalty x Miss Rate

* Hit Time = time to find and retrieve data from current level cache
* Miss Penalty = average time to retrieve data on a current level miss (includes the possibility of misses on successive levels of memory hierarchy)
* Hit Rate = % of requests that are found in current level cache
* Miss Rate = 1 - Hit Rate



**Fully associative Cache**

Memory address fields:

• Tag: same as before

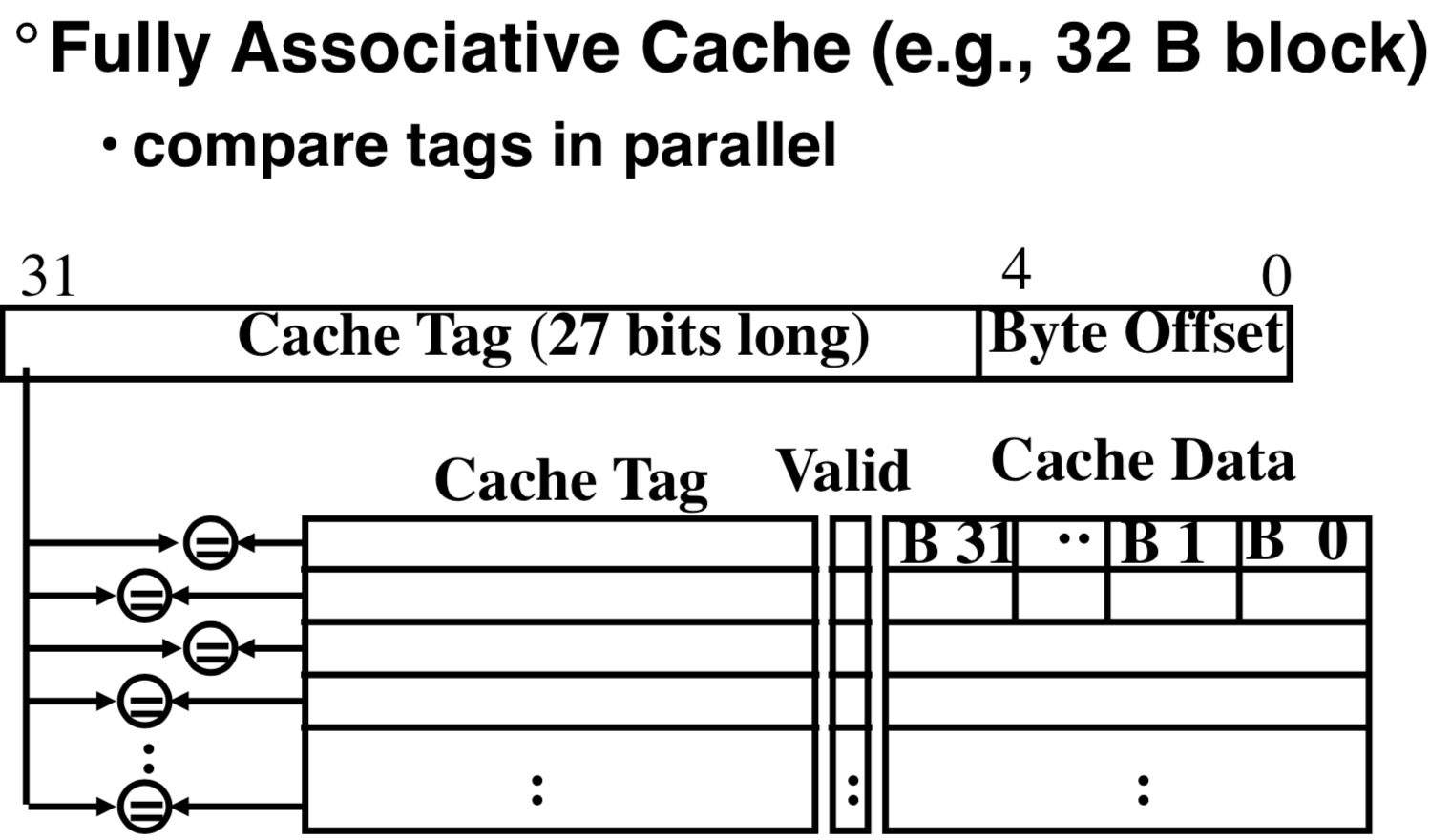
• Offset: same as before

• Index: non-existent

What does this mean?

• any block can go anywhere in the cache

• must compare with all tags in entire cache to see if data is there



Benefit of Fully Assoc Cache

• no Conflict Misses (since data can go anywhere)

Drawbacks of Fully Assoc Cache

• need hardware comparator for every single entry: if we have a 64KB of data in cache with 4B entries, we need 16K comparators: very expensive

Small fully associative cache may be feasible

Third type of Cache Misses-***Capacity Misses***

• miss that occurs because the cache has a limited size

• miss that would not occur if we increase the size of the cache

This is the primary type of miss for Fully Associate caches.

N-Way Set Associative Cache

Direct mapping with respect to sets or groups of blocks. Each set is just fully associated.

Memory address fields:

• Tag: same as before

• Offset: same as before

• Index: points us to the correct “row” (called a ***set*** in this case)

So what’s the difference?

• each set contains multiple blocks

• once we’ve found correct set, must compare with all tags in that set to find our data

Given memory address:

• Find correct set using Index value.

• Compare Tag with all Tag values in the determined set.

• If a match occurs, it’s a hit, otherwise a miss.

• Finally, use the offset field as usual to find the desired data within the desired block.

Benefits

• even a 2-way set assoc cache avoids a lot of conflict misses

• hardware cost isn’t that bad: only need N comparators

In fact, for a cache with M blocks,

• it’s Direct-Mapped if it’s 1-way set assoc (1 block per set)

• it’s Fully Assoc if it’s M-way set assoc (M blocks per set)

• so these two are just special cases of the more general set associative design

If there are any locations with valid bit off (empty), then usually write the new block into the first one. If all possible locations already have a valid block, we must use a **replacement policy** by which we determine which block gets “cached out” on a miss.

**Block Replacement Policy---LRU (Least Recently Used)**

*Cache out block which has been accessed (read or write) least recently*

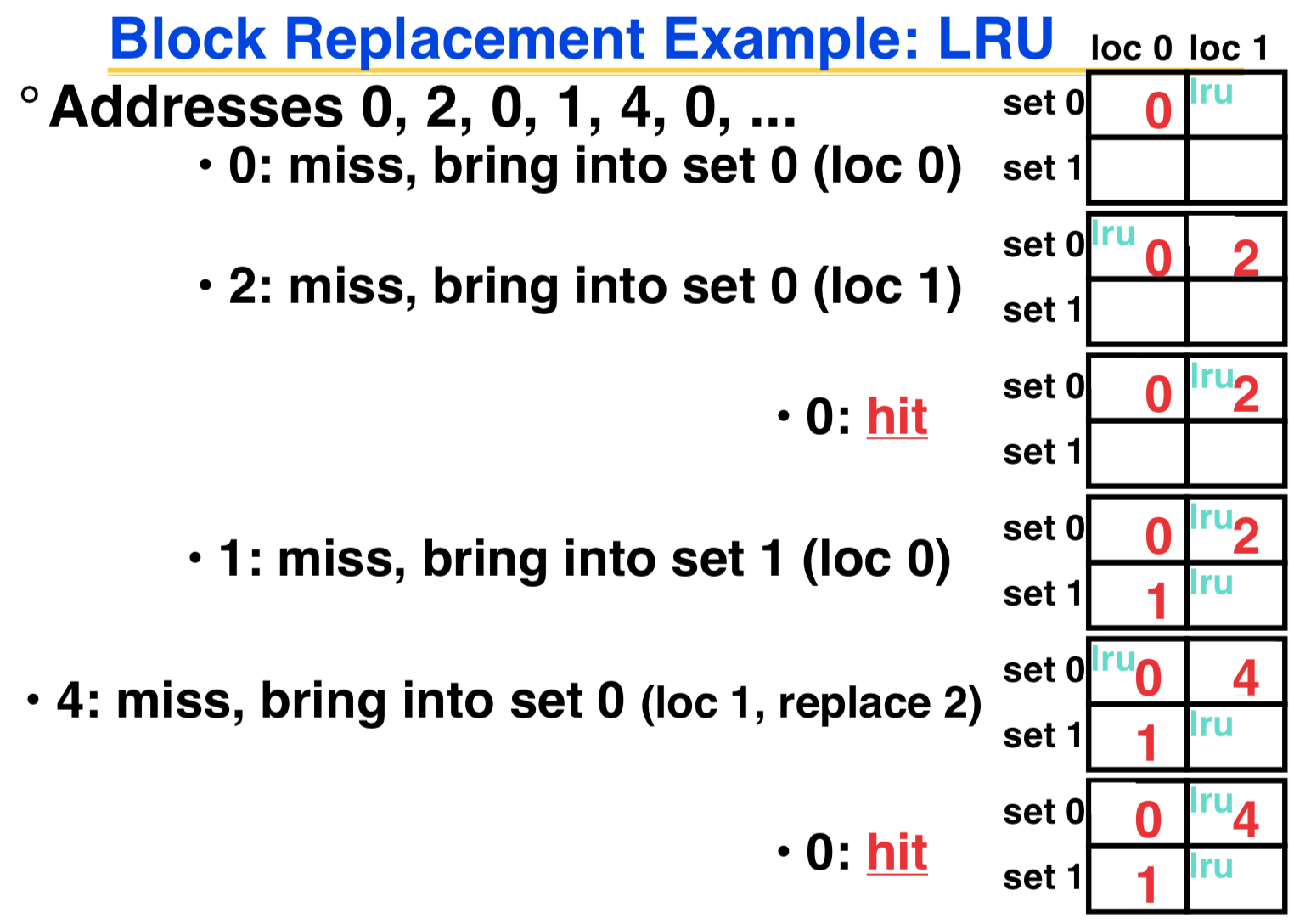
• Pro: temporal locality => recent past use implies likely future use

• Con: with 2-way set assoc, easy to keep track (one LRU bit); with 4-way or greater, requires complicated hardware and much time to keep track of this

Example: We have a 2-way set associative cache with a four word total capacity and one word blocks. We perform the following word accesses (ignore bytes for this problem):

0, 2, 0, 1, 4, 0, 2, 3, 5, 4

How many hits and how many misses will there for the LRU block replacement policy?



How to reduce miss rate?

1. Larger cache

• limited by cost and technology

• hit time of first level cache < cycle time

2. More places in the cache to put each block of memory - associativity

• fully-associative

- any block any line

• k-way set associated

- k places for each block

- direct map: k=1

Average Access Time = Hit Time + Miss Penalty x Miss Rate

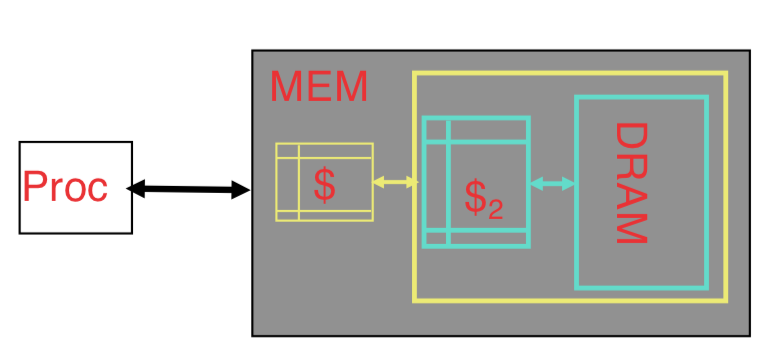
hit rate \* hit penalty

Assume

• Hit Time = 1 cycle

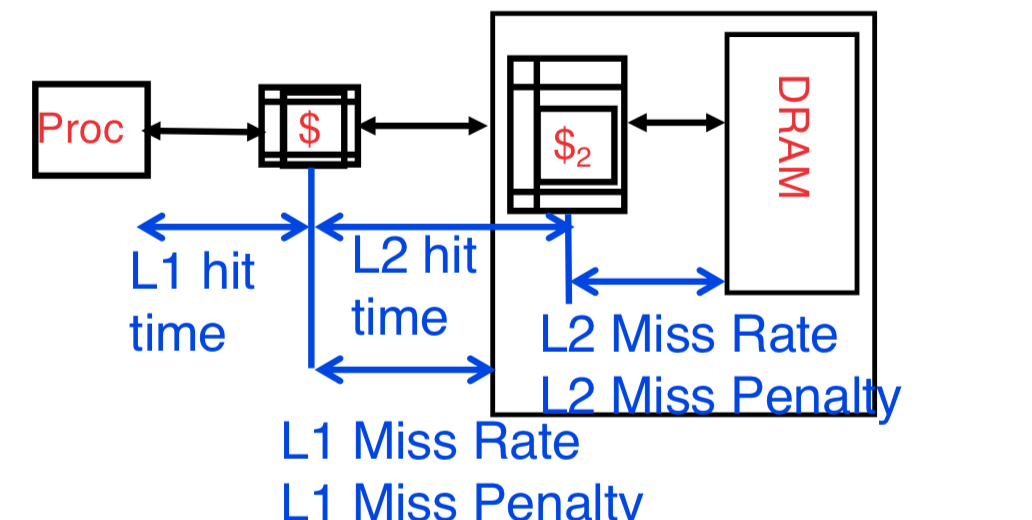
• Miss rate = 5%

• Miss penalty = 20 cycles

Avg mem access time = 1 + 0.05 x 20 = 2 cycle

Improve Miss Penalty:

Add another cache between memory and the processor cache: ***Second Level (L2) Cache***



Avg Mem Access Time = L1 Hit Time + L1 Miss Rate \* L1 Miss Penalty

L1 Miss Penalty = L2 Hit Time + L2 Miss Rate \* L2 Miss Penalty

🡪 Avg Mem Access Time = L1 Hit Time + L1 Miss Rate \* (L2 Hit Time + L2 Miss Rate \* L2 Miss Penalty )

Typical Scale

L1:

• size: tens of KB

• hit time: complete in one clock cycle

• miss rates: 1-5%

L2:

• size: hundreds of KB

• hit time: few clock cycles

• miss rates: 10-20% (L2 miss rate is fraction of L1 misses that also miss in L2)

Assume (WITHOUT L2 cache)

• L1 Hit Time = 1 cycle

• L1 Miss rate = 5%

• L1 Miss Penalty = 100 cycles

🡪nAvg mem access time = 1 + 0.05 x 100 = 6 cycles

Assume (WITH L2 cache)

• L1 Hit Time = 1 cycle

• L1 Miss rate = 5%

• L2 Hit Time = 5 cycles

• L2 Miss rate = 15% (% L1 misses that miss)

• L2 Miss Penalty = 100 cycles

🡪 L1 miss penalty = 5 + 0.15 \* 100 = 20

🡪 Avg mem access time = 1 + 0.05 x 20 = 2 cycle 3x faster with L2 cache

What to do on a write hit?

**Write-through**

• update the word in cache block and corresponding word in memory

**Write-back (today)**

• update word in cache block

• allow memory word to be “stale”

=> add ‘dirty’ bit to each line indicating that memory needs to be updated when block is replaced

=> OS flushes cache before I/O !!!