Computer Architecture 1

Computer Organization and Design

THE HARDWARE/SOFTWARE INTERFACE

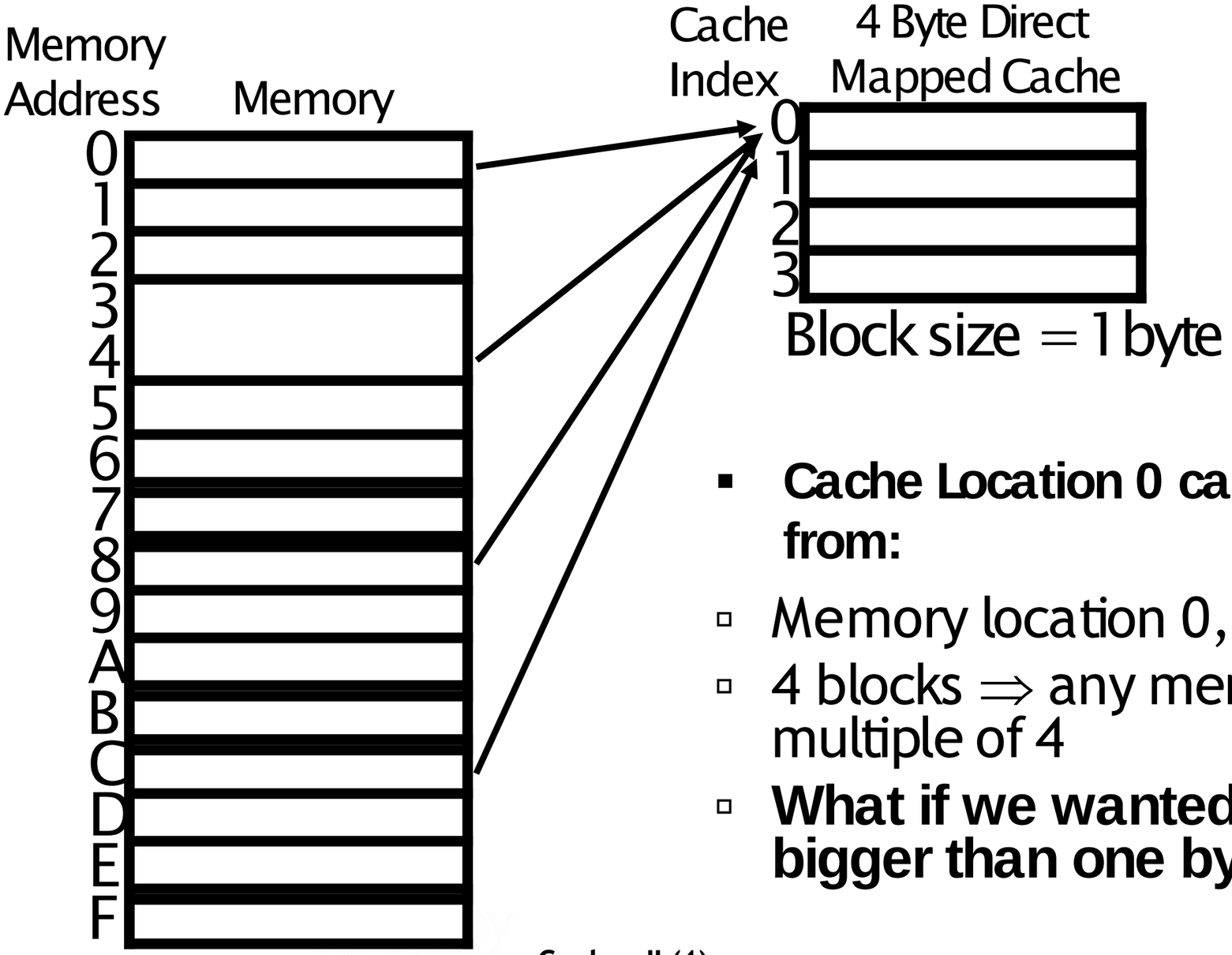
[Adapted from Computer Organization and Design, RISC-V Edition, Patterson & Hennessy, © 2018, MK]
[Adapted from Great ideas in Computer Architecture (CS 61C) lecture slides, Garcia and Nikolíc, © 2020, UC Berkeley]

Direct Mapped Caches

Direct-Mapped Cache (1/4)

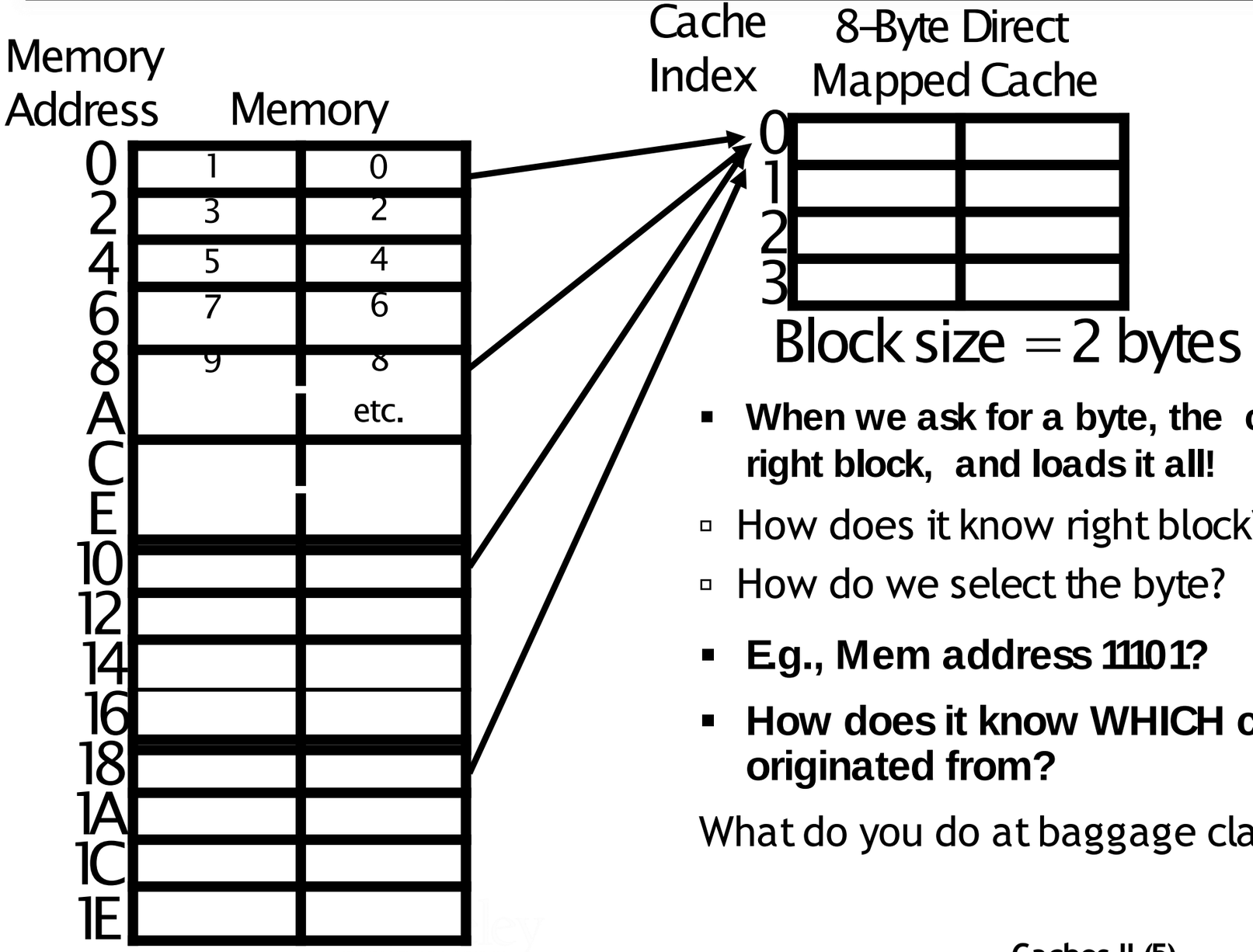
- **In a direct-mapped cache, each memory address is associated with one possible block within the cache**
  - Therefore, we only need to look in a single location in the cache for the data if it exists in the cache
  - Block is the unit of transfer between cache and memory

Direct-Mapped Cache (2/4)



- **Cache Location 0 can be occupied by data from:**
  - Memory location 0, 4, 8, ...
  - 4 blocks ⇒ any memory location that is multiple of 4
  - **What if we wanted a block to be bigger than one byte?**

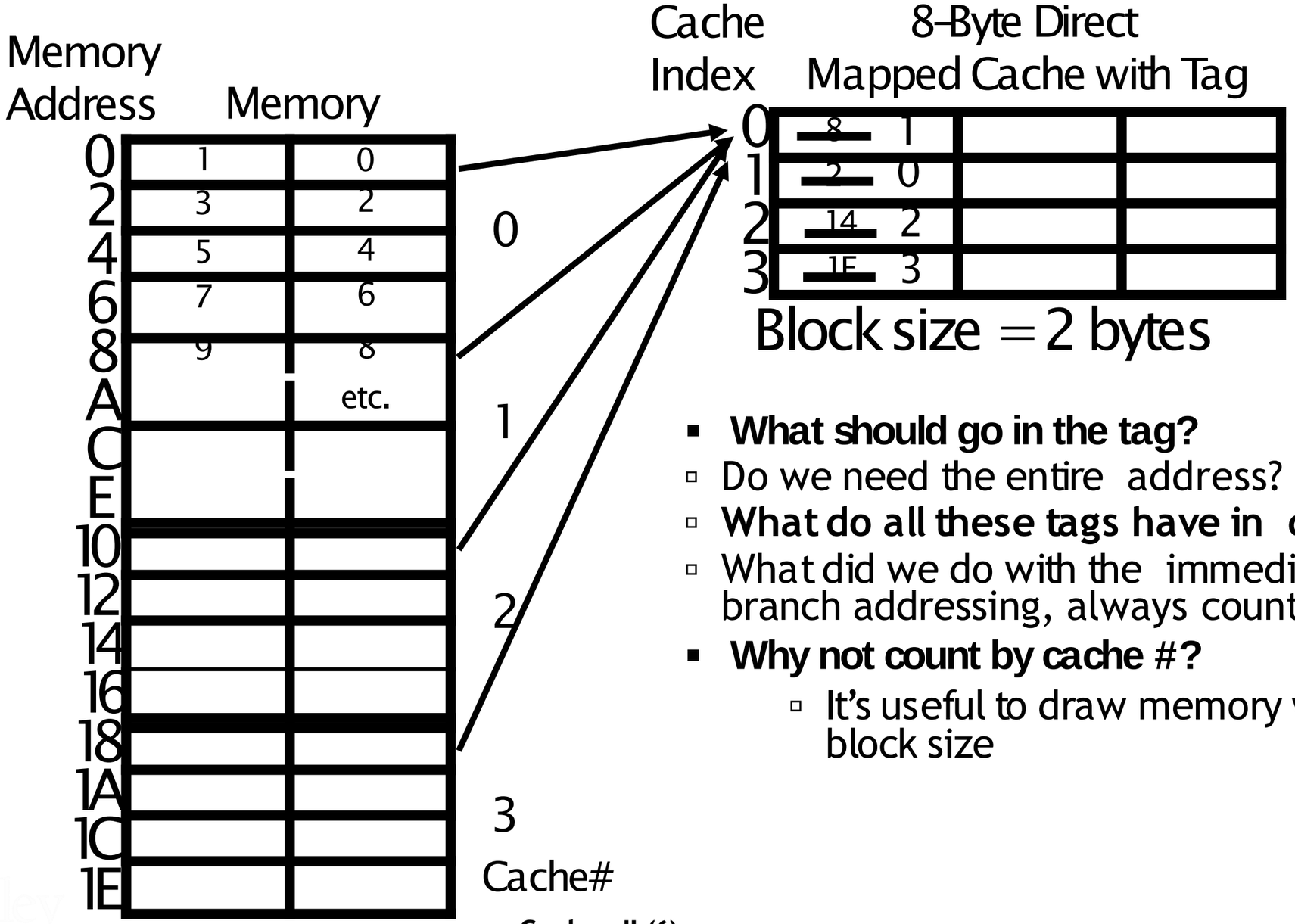
Direct-Mapped Cache (3/4)



- **When we ask for a byte, the controller finds out the right block, and loads it all!**
  - How does it know right block?
  - How do we select the byte?
- **Eg., Mem address 11101?**
- **How does it know WHICH colored block it originated from?**

What do you do at baggage claim?

Direct-Mapped Cache (4/4)



- **What should go in the tag?**
  - Do we need the entire address?
  - **What do all these tags have in common?**
  - What did we do with the immediate when we were branch addressing, always count by bytes?
- **Why not count by cache #?**
  - It's useful to draw memory with the same width as the block size

Issues with Direct-Mapped

- Since multiple memory addresses map to same cache index, how do we tell which one is in there?
- What if we have a block size > 1 byte?
- Answer: divide memory address into three fields

Tag to check if have correct block

Index to select block

Byte offset within block

Garcia, Nikolić

Direct-Mapped Cache Terminology

- **All fields are read as unsigned integers.**
- **Index**
  - specifies the cache index (which "row"/block 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

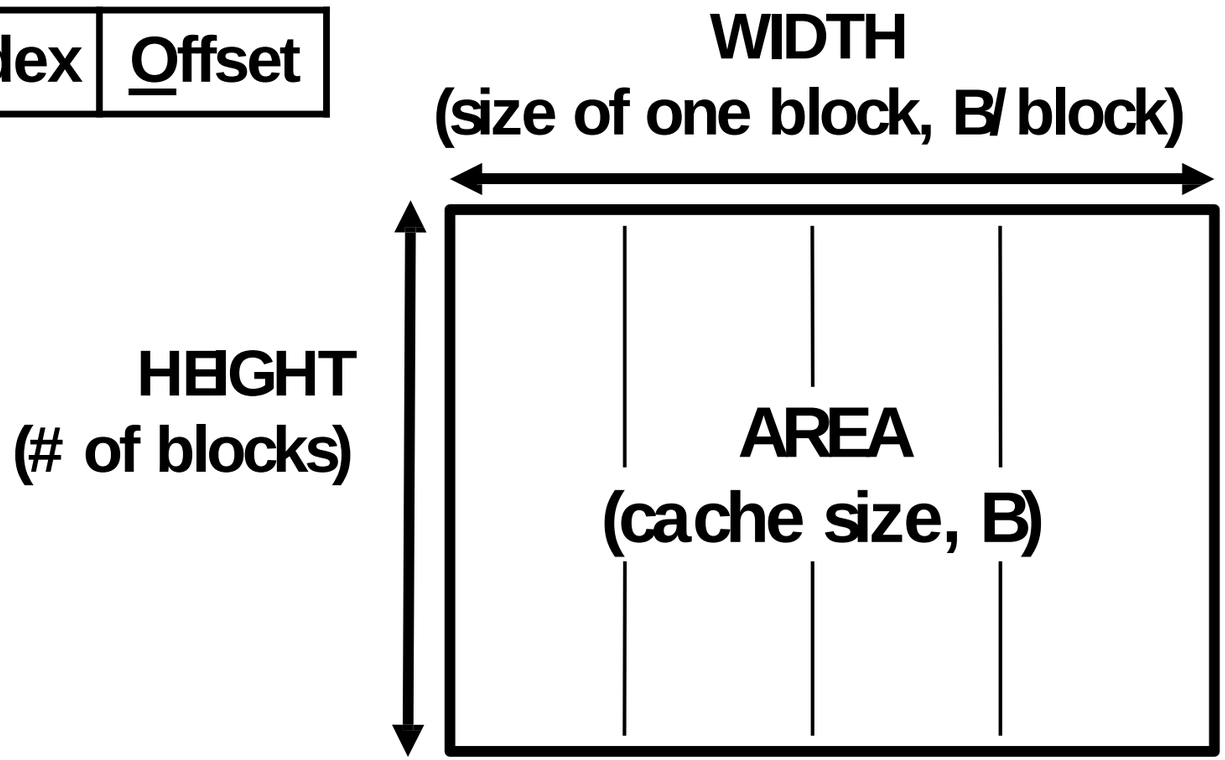
TLO Cache Mnemonic (Thanks Uncle Dan!)

AREA (cache size, B)
= HEIGHT (# of blocks)

$2^{(H+W)} = 2^H * 2^W$

* WIDTH (size of one block, B/block)

Tag	Index	Offset



Direct Mapped Example

Direct-Mapped Cache Example (1/3)

- **Suppose we have a 8B of data in a direct-mapped cache with 2-byte blocks**
  - Sound familiar?
- **Determine the size of the tag, index and offset fields if using a 32-bit arch (RV32)**
- **Offset**
  - need to specify correct byte within a block
  - block contains 2 bytes

    $= 2^1$ bytes
  - need 1 bit to specify correct byte

Direct-Mapped Cache Example (2/3)

- **Index: (~index into an "array of blocks")**
  - need to specify correct block in cache
  - cache contains 8 B = 2^3 bytes
  - block contains 2 B = 2^1 bytes
  - # blocks/cache

$$= \frac{\text{bytes/cache}}{\text{bytes/block}}$$

$$= \frac{2^3 \text{ bytes/cache}}{2^1 \text{ bytes/block}}$$

$$= 2^2 \text{ blocks/cache}$$

  - need 2 bits to specify this many blocks

Direct-Mapped Cache Example (3/3)

- **Tag: use remaining bits as tag**
  - tag length = addr length – offset - index
    = 32 - 1 - 2 bits
    = 29 bits
  - so tag is leftmost 29 bits of memory address
  - Tag can be thought of as "cache number"
- **Why not full 32-bit address as tag?**
  - All bytes within block need same address
  - Index must be same for every address within a block, so it's redundant in tag check, thus can leave off to save memory

Garcia, Nikolić

Memory Access without Cache

- **Load word instruction:** `lw t0, 0(t1)`
- `t1` **contains** 1022_{ten}, `Memory[1022] = 99`

1. Processor issues address 1022_{ten} to `Memory`
2. Memory reads word at address 1022_{ten} (`99`)
3. Memory sends `99` to Processor
4. Processor loads `99` into register `t0`

Memory Access with Cache

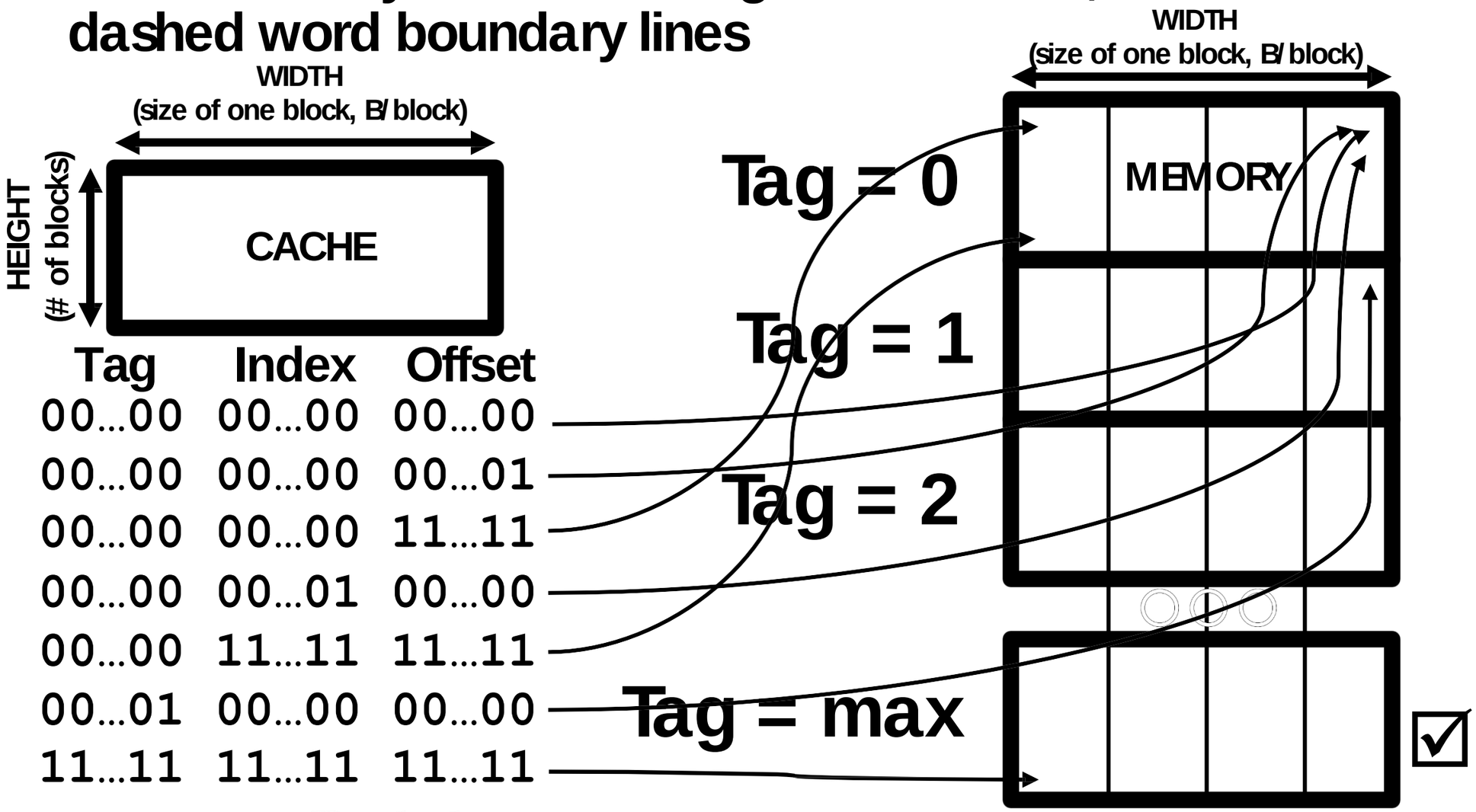
- **Load word instruction:** `lw t0, 0(t1)`
- `t1` **contains** 1022_{ten}, `Memory[1022] = 99`
- **With cache (similar to a hash)**
  1. Processor issues address 1022_{ten} to Cache
  2. Cache checks to see if has copy of data at address 1022_{ten}

     2a. If finds a match (Hit): cache reads `99`, sends to processor 2b. No match (Miss): cache sends address `1022` to Memory
     - I. Memory reads `99` at address 1022_{ten}
     - II. Memory sends `99` to Cache
     - III. Cache replaces word with new `99`
     - IV. Cache sends `99` to processor
  3. Processor loads `99` into register `t0`

Solving Cache problems

Tag	Index	Offset

- Draw memory a block wide given T I O bits, dashed word boundary lines



Cache Terminology

Caching Terminology

- **When reading memory, 3 things can happen:**
  - cache hit:
    cache block is valid and contains proper address, so read desired word
  - cache miss:
    nothing in cache in appropriate block, so fetch from memory
  - cache miss, block replacement:
    wrong data is in cache at appropriate block, so discard it and fetch desired data from memory
    (cache always copy)

Garcia, Nikolić

Cache Temperatures

- **Cold**
  - Cache empty
- **Warming**
  - Cache filling with values you'll hopefully be accessing again soon
- **Warm**
  - Cache is doing its job, fair % of hits
- **Hot**
  - Cache is doing very well, high % of hits

Cache Terms

- Hit rate: fraction of access that hit in the cache
- Miss rate: 1– Hit rate
- Miss penalty: time to replace a block from lower level in memory hierarchy to cache
- Hit time: time to access cache memory (including tag comparison)
- Abbreviation: “$” = cache
  - … a Berkeley innovation!

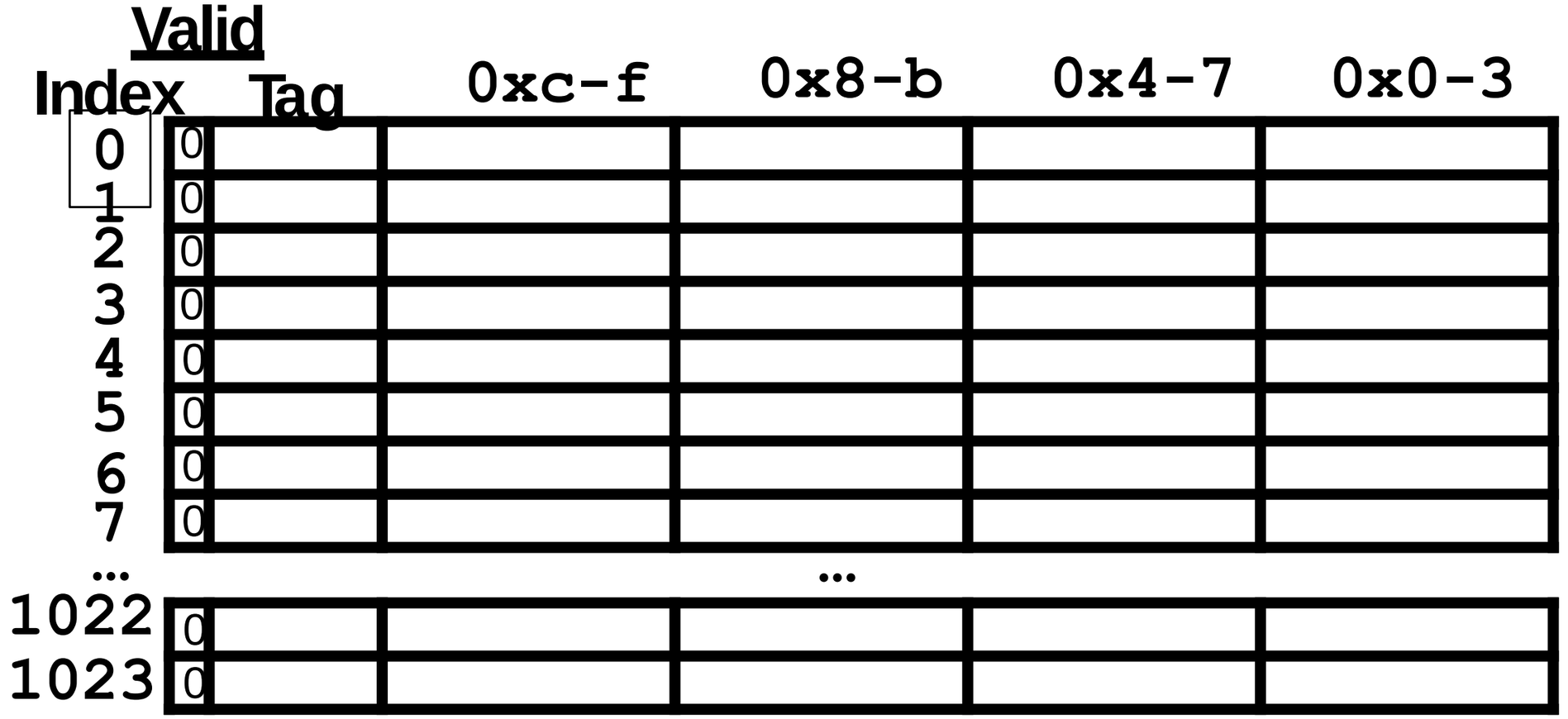
One More Detail: Valid Bit

- **When start a new program, cache does not have valid information for this program**
- **Need an indicator whether this tag entry is valid for this program**
- **Add a "valid bit" to the cache tag entry** 0 → cache miss, even if by chance, address = tag 1 → cache hit, if processor address = tag

Garcia, Nikolić

Example: 16 KB Direct-Mapped Cache, 16B blocks

- Valid bit: determines whether anything is stored in that row (when computer initially powered up, all entries invalid)



Looks like a real cache, wil investigate it some more!

Garcia, Nikolić

"And in Conclusion…"

- **We have learned the operation of a direct-mapped cache**
- **Mechanism for transparent movement of data among levels of a memory hierarchy**
  - set of address/value bindings
  - address → index to set of candidates
  - compare desired address with tag
  - service hit or miss
  - load new block and binding on miss

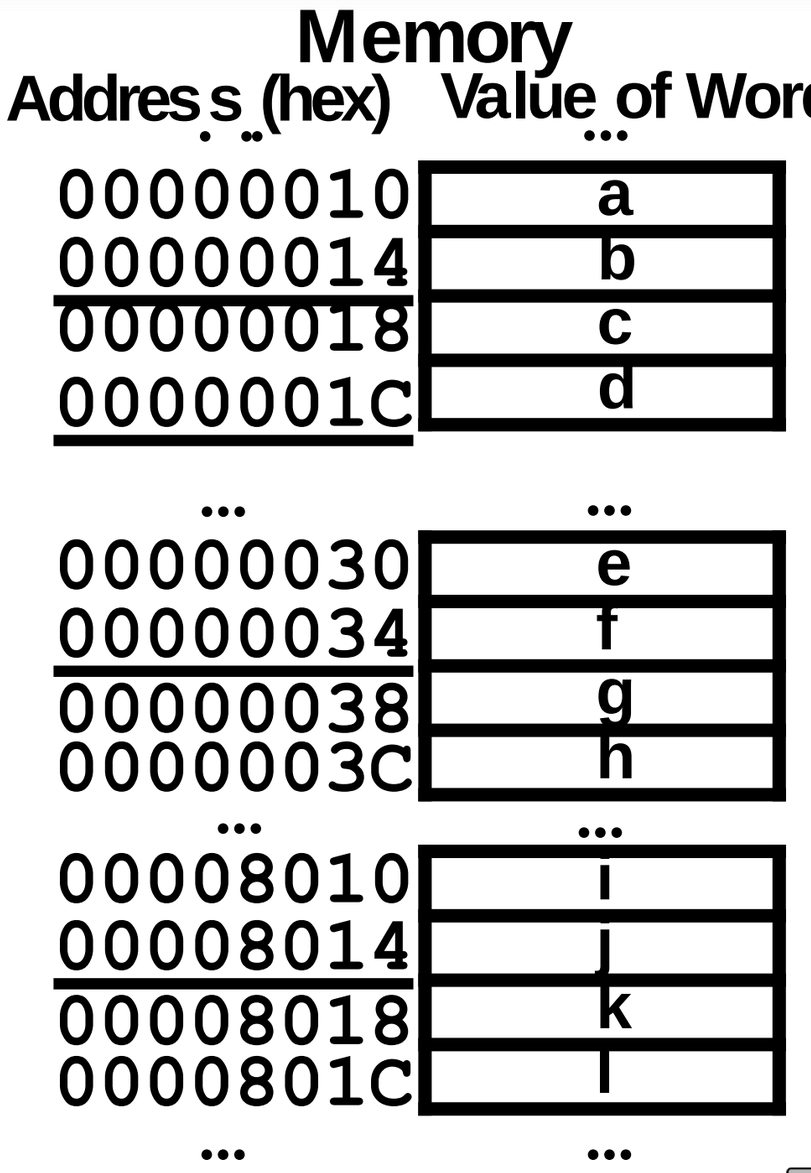
Direct Mapped Example

Accessing data in a direct mapped cache

- **Ex.: 16KB of data, direct-mapped, 4 word blocks**
  - Can you work out height, width, area?
- **Read 4 addresses**

1. `0x00000014`
2. `0x0000001C`
3. `0x00000034`
4. `0x00008014`

- **Memory values here:**



Accessing data in a direct mapped cache

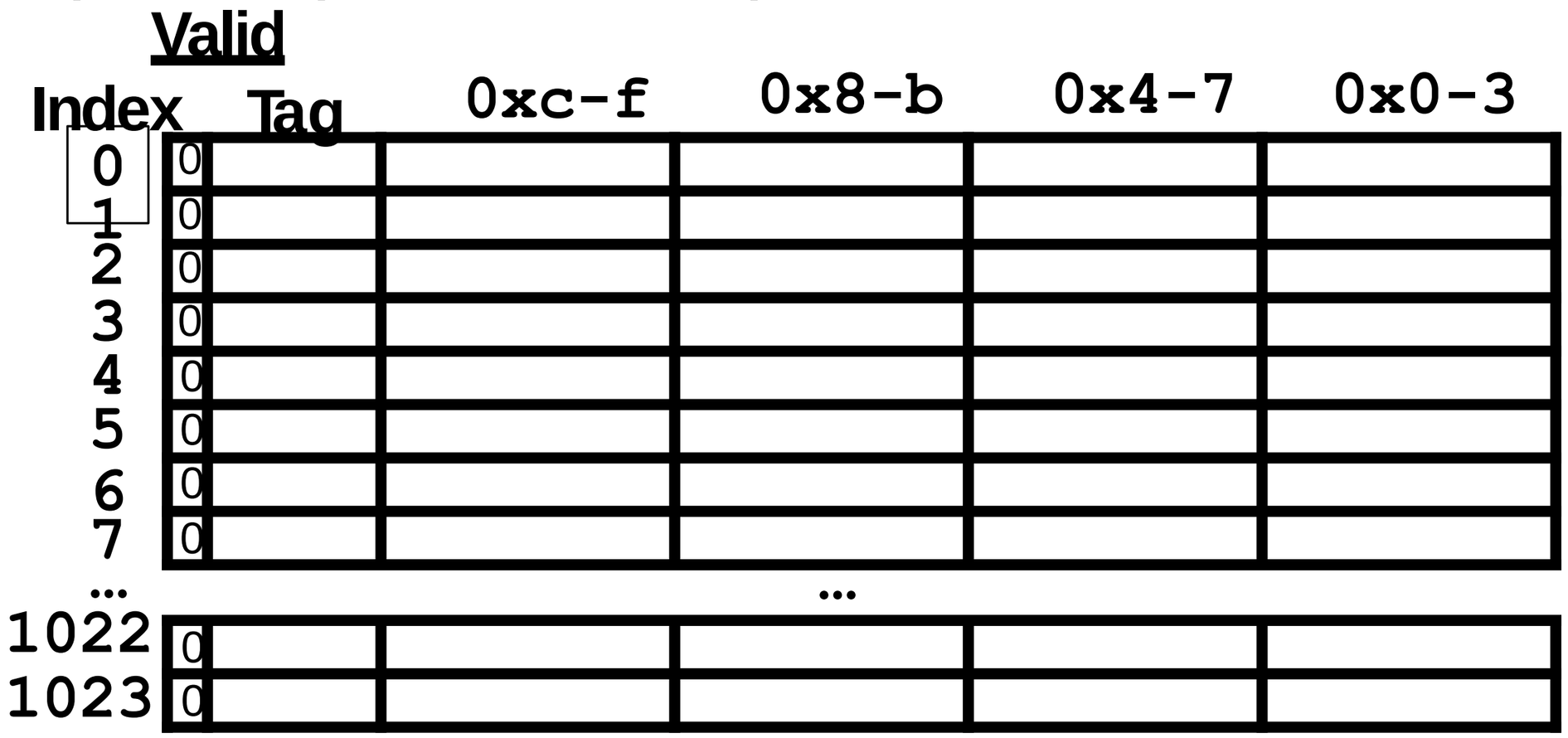
- 4 Addresses:
  - `0x00000014, 0x0000001C,`
    `0x00000034, 0x00008014`
- 4 Addresses divided (for convenience) into Tag, Index, Byte Offset fields

```
0000000000000000 0000000001 0100
0000000000000000 0000000001 1100
0000000000000000 0000000011 0100
0000000000000010 0000000001 0100
```

Tag Index Offset

Example: 16 KB Direct-Mapped Cache, 16B blocks

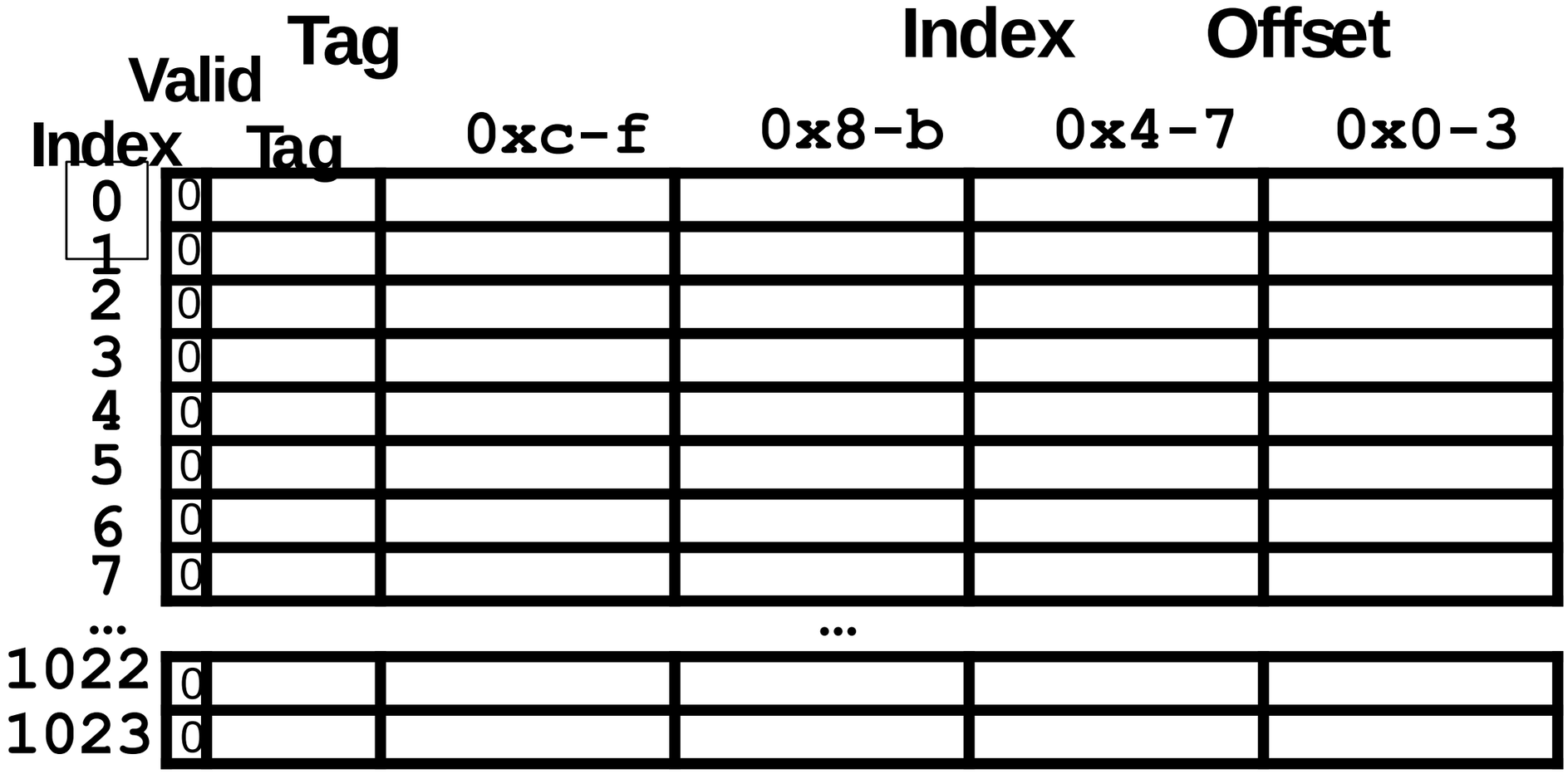
- Valid bit: determines whether anything is stored in that row (when computer initially powered up, all entries invalid)



Garcia, Nikolić

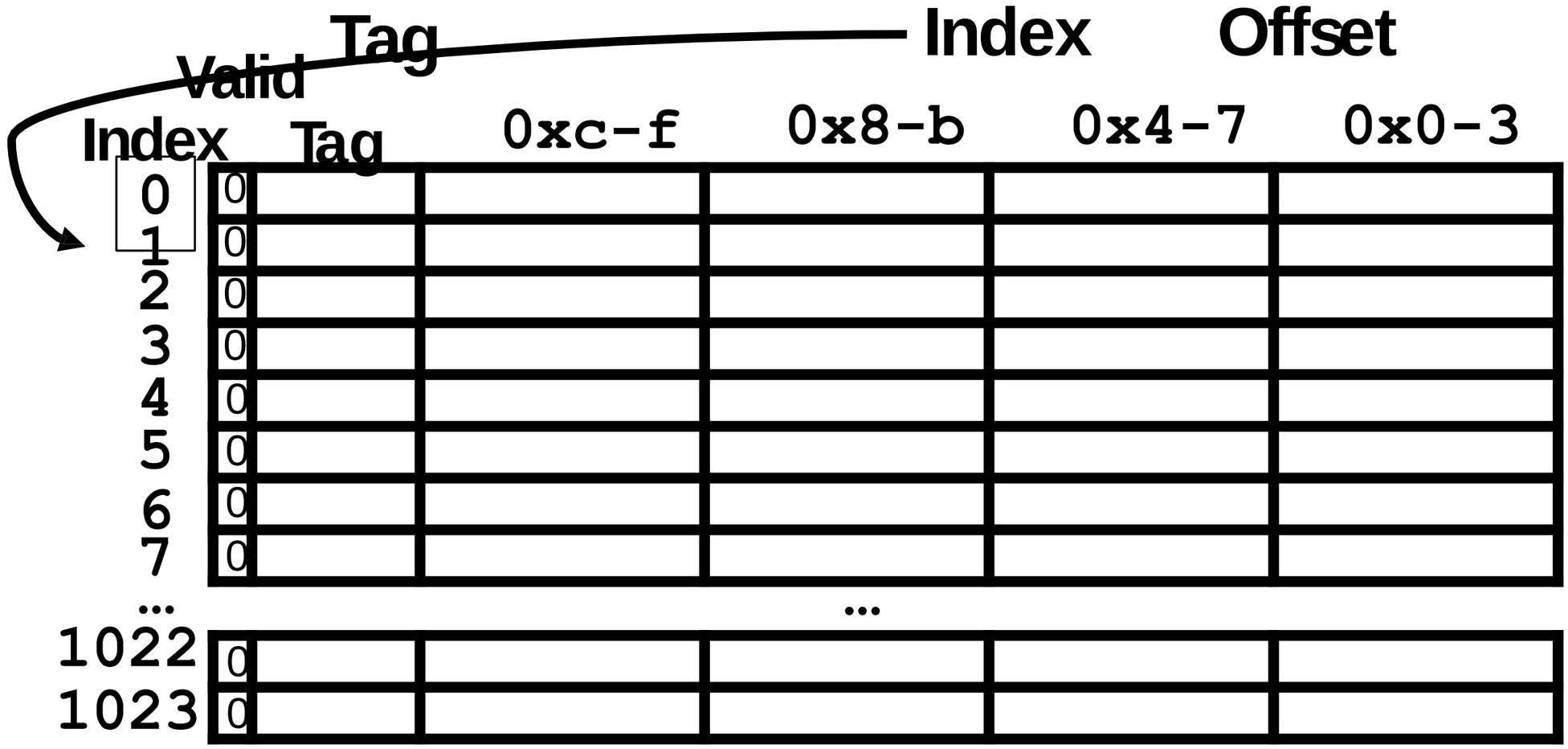
1. Read 0x00000014

- 00000000000000000000 0000000001 0100



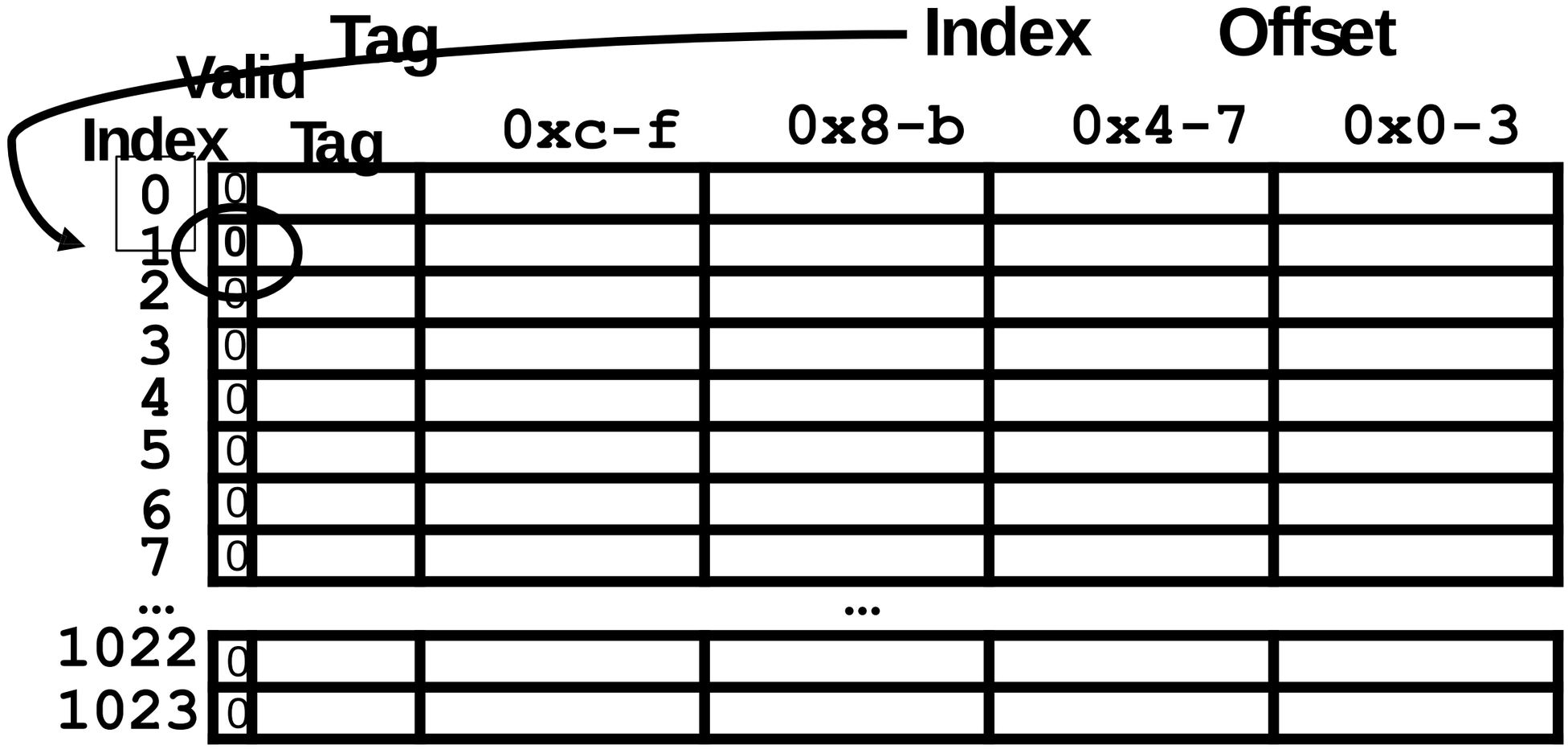
So we read block 1 (0000000001)

- 00000000000000000000 0000000001 0100



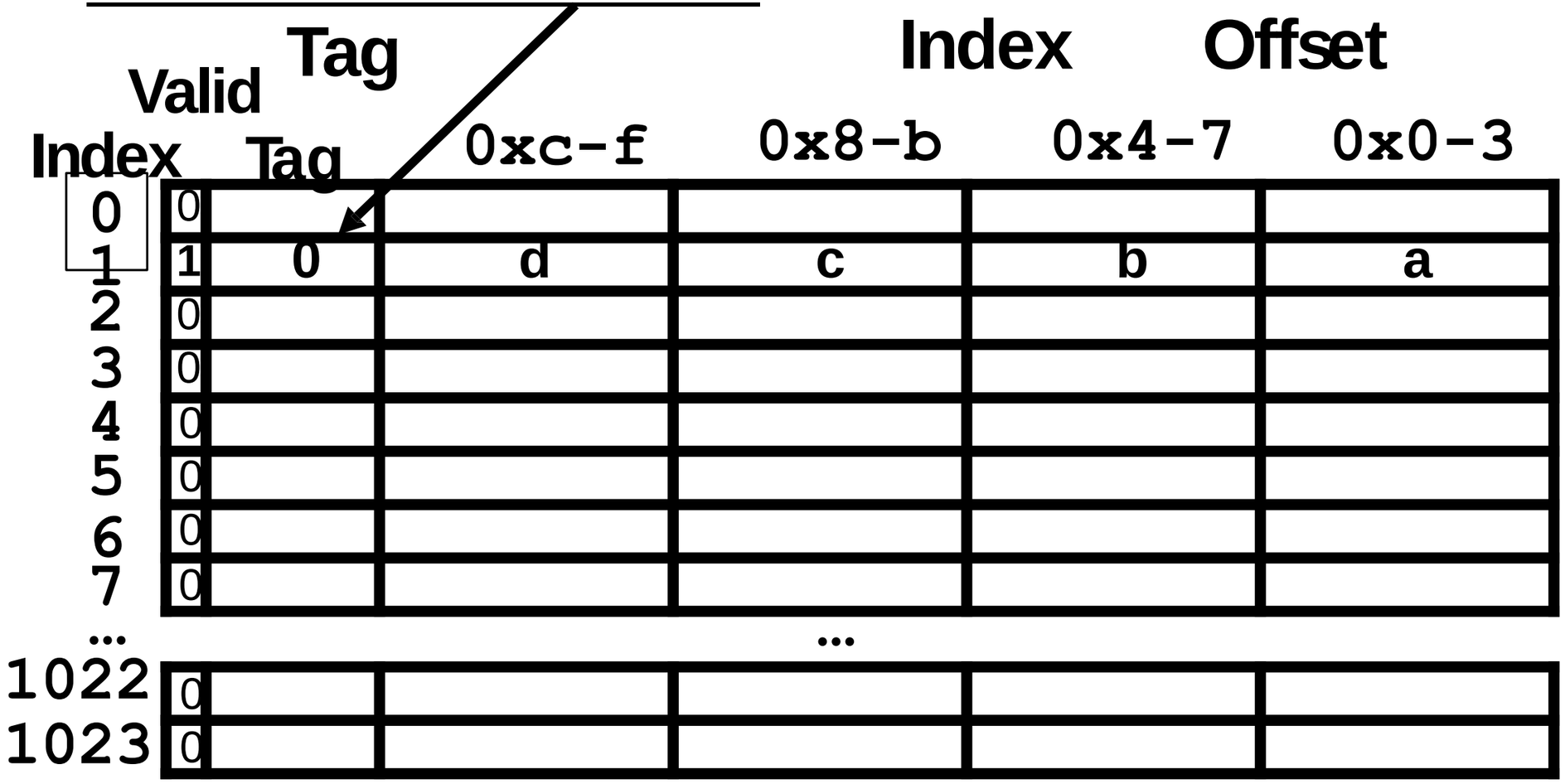
No valid data

- 00000000000000000000 0000000001 0100



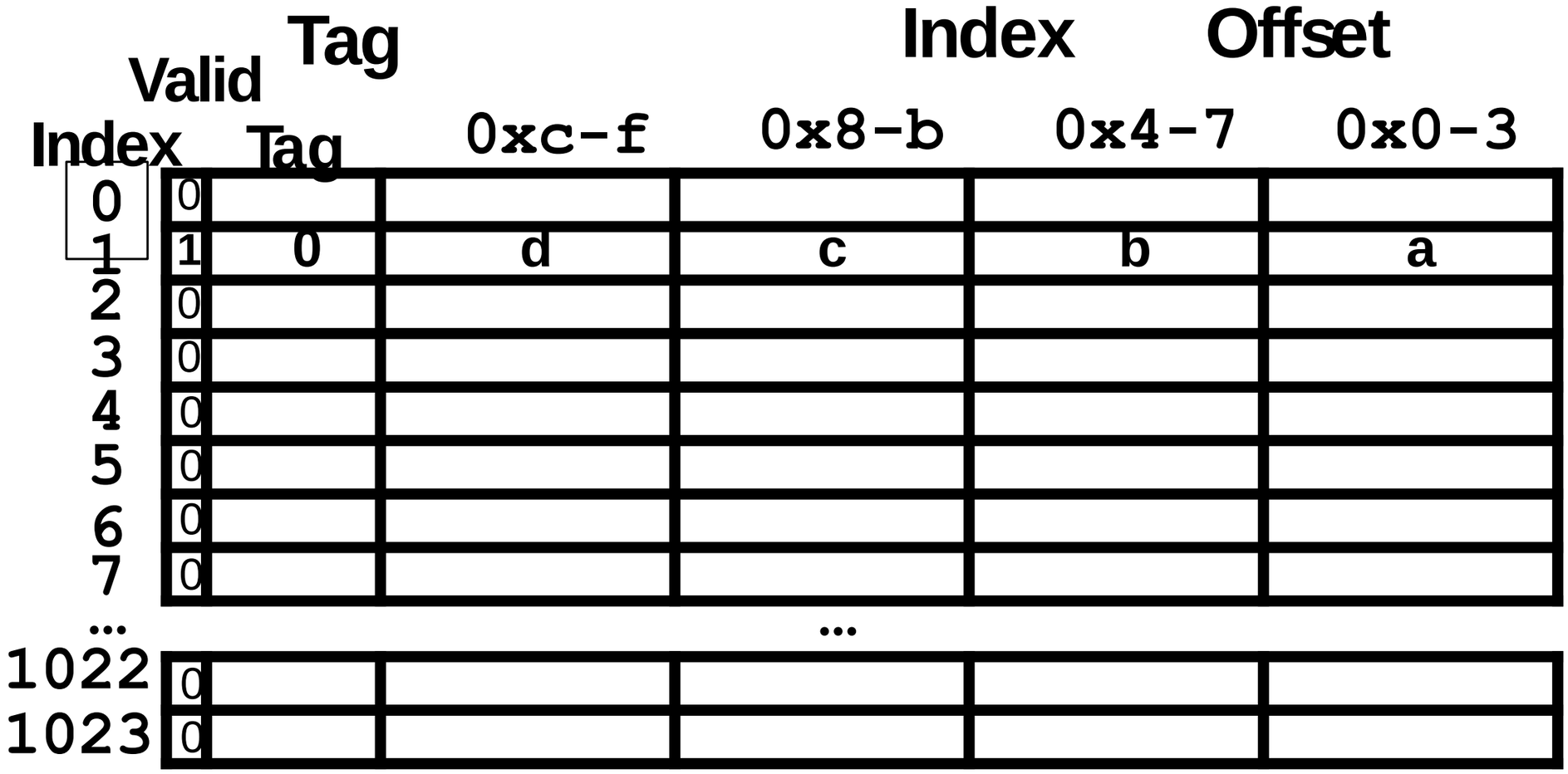
So load that data into cache, setting tag, valid

- 00000000000000000000 0000000001 0100



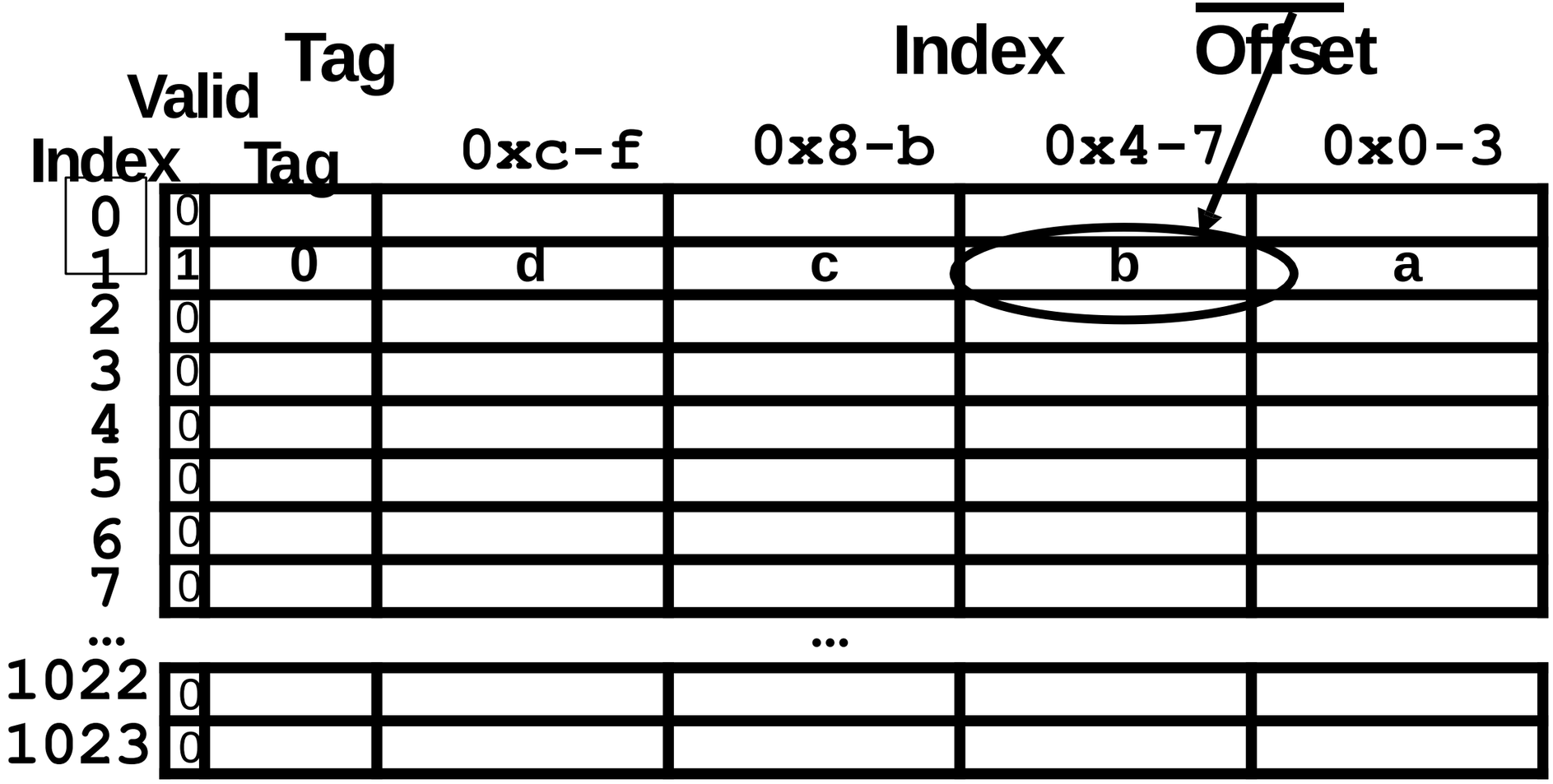
2. Read 0x0000001C = 0…00 0..0011100

- 00000000000000000000 0000000001 1100



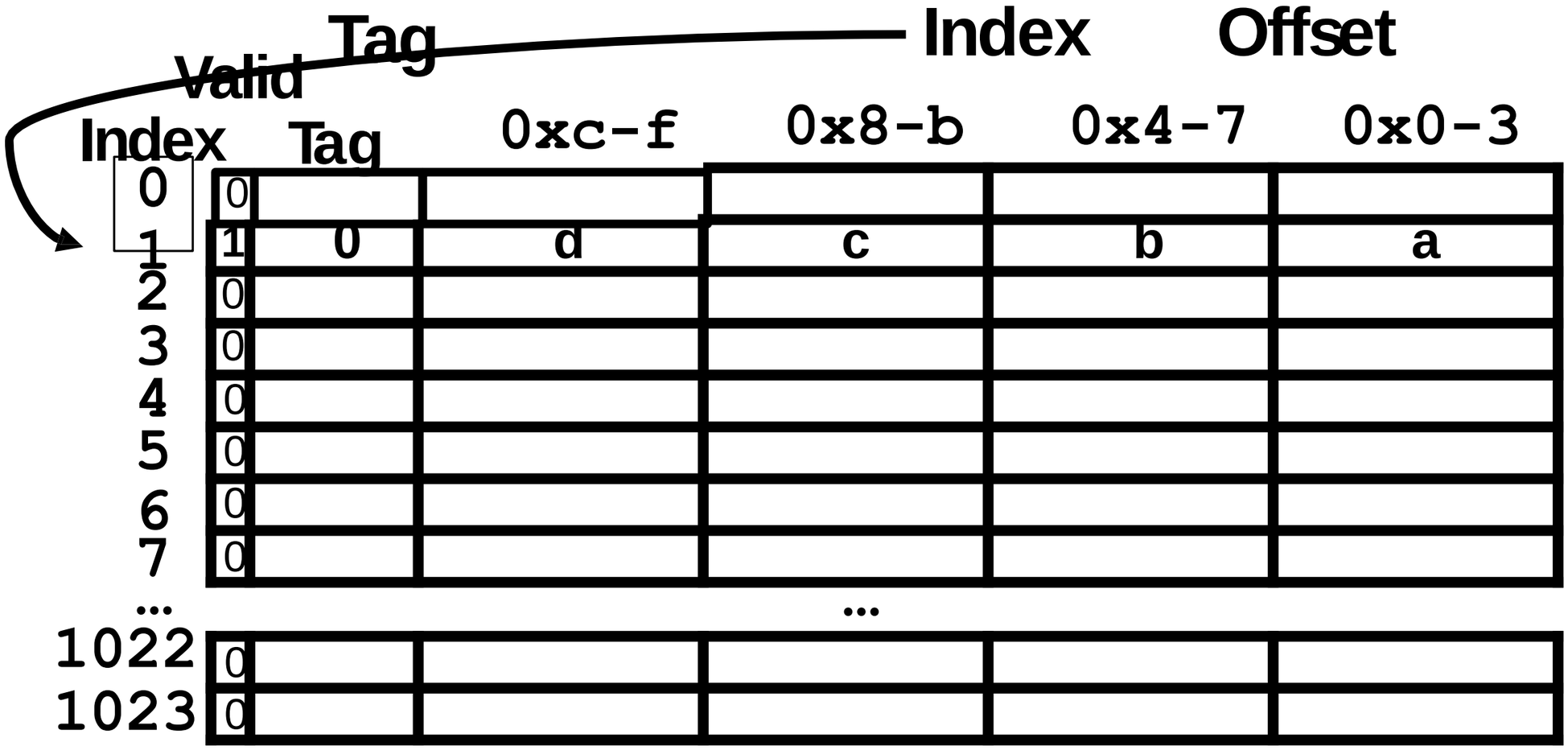
Read from cache at offset, return word b

- 00000000000000000000 0000000001 0100



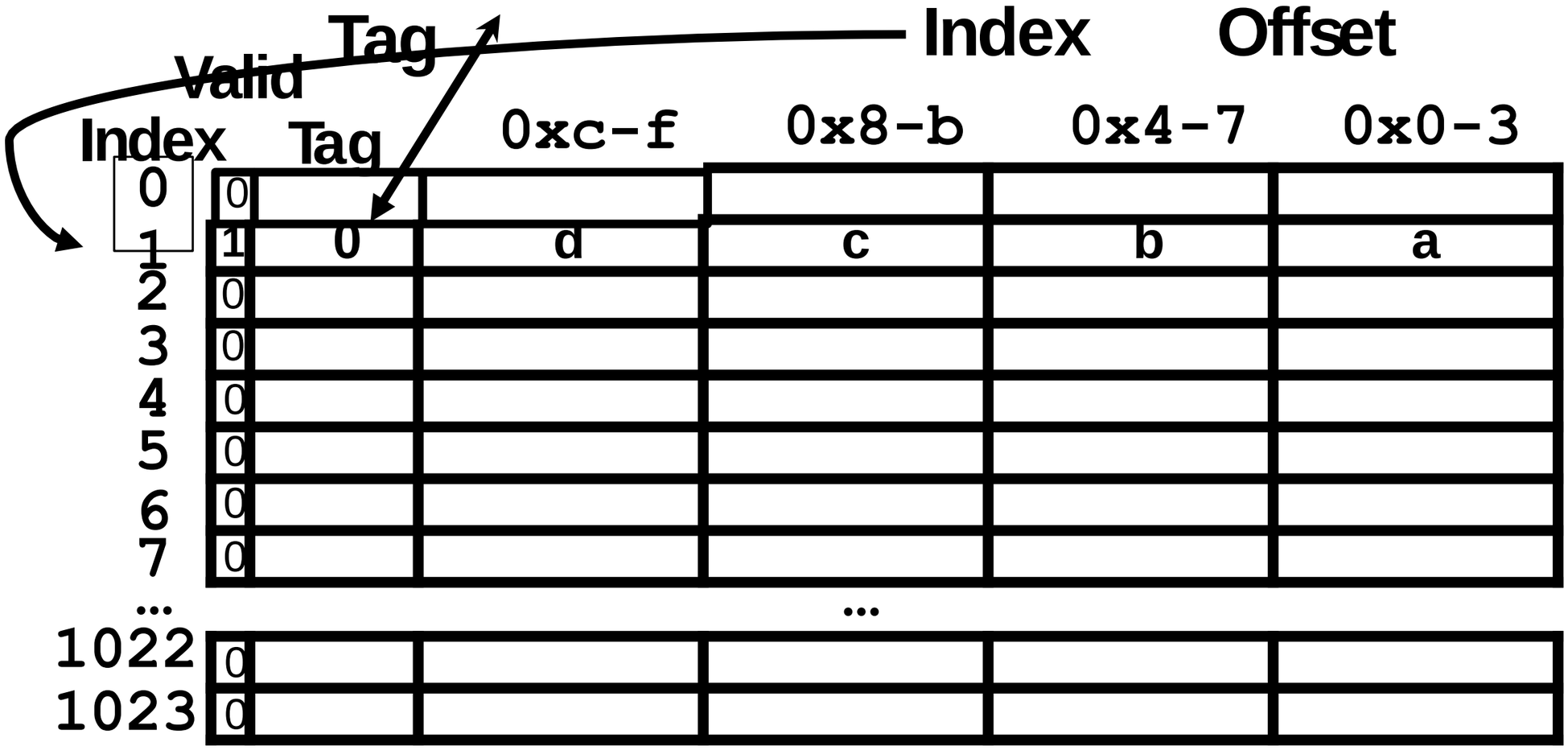
Index is Valid

- 00000000000000000000 0000000001 1100



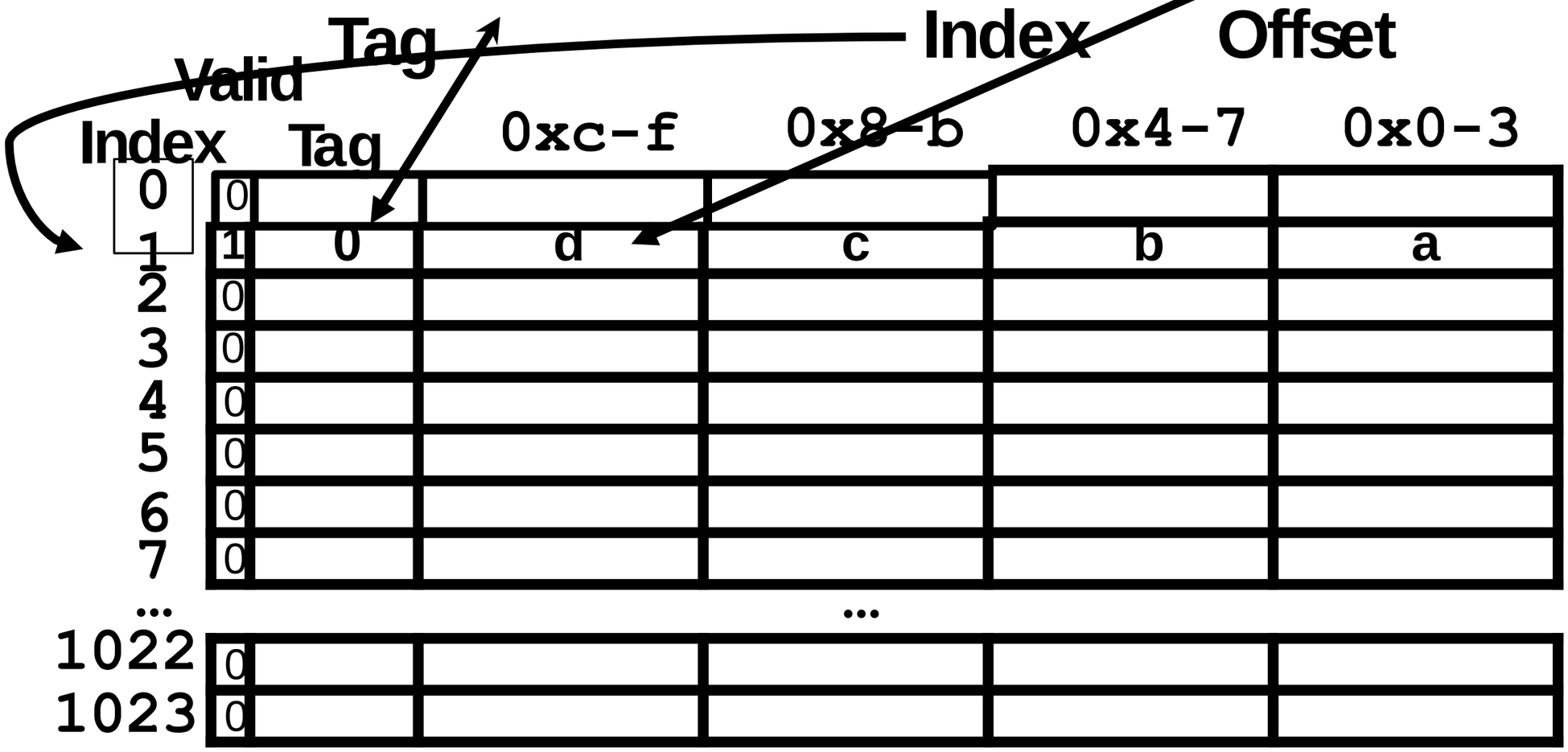
Index is Valid, Tag Matches

- 00000000000000000000 0000000001 1100



Index is Valid, Tag Matches, return d

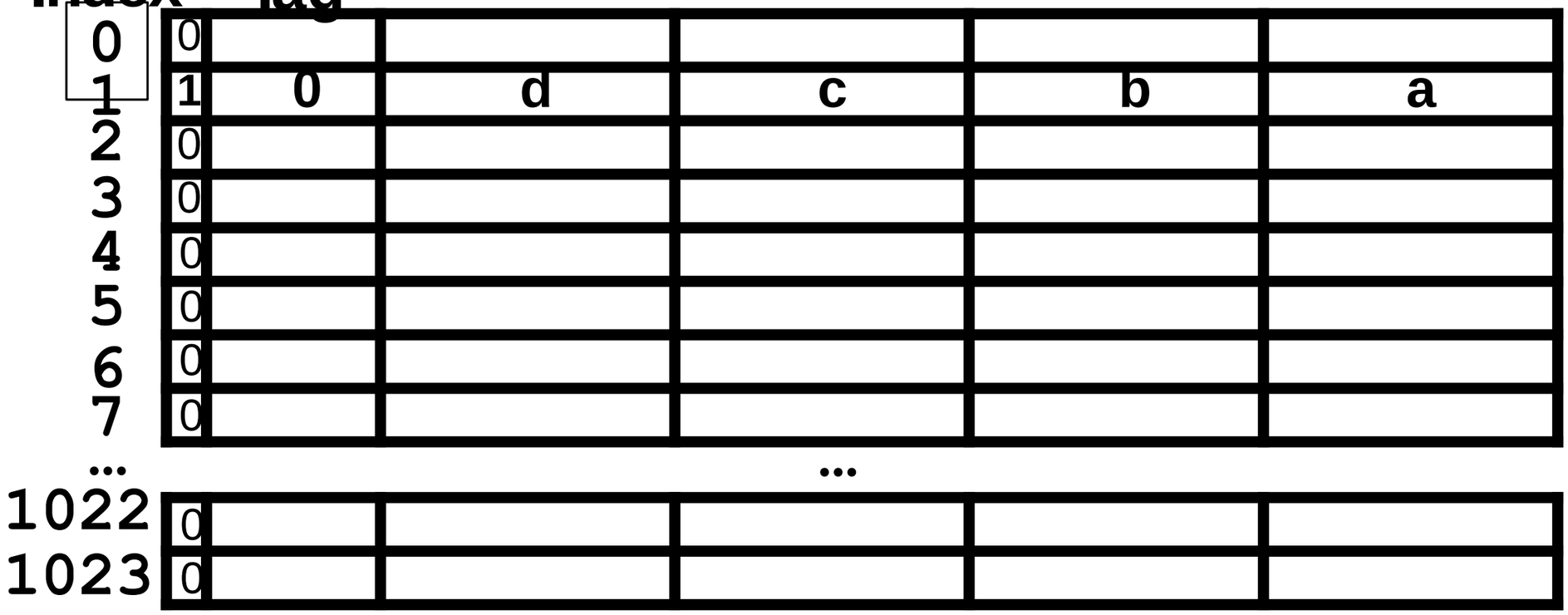
- 00000000000000000000 0000000001 1100



3. Read 0x00000034 = 0…00 0..0110100

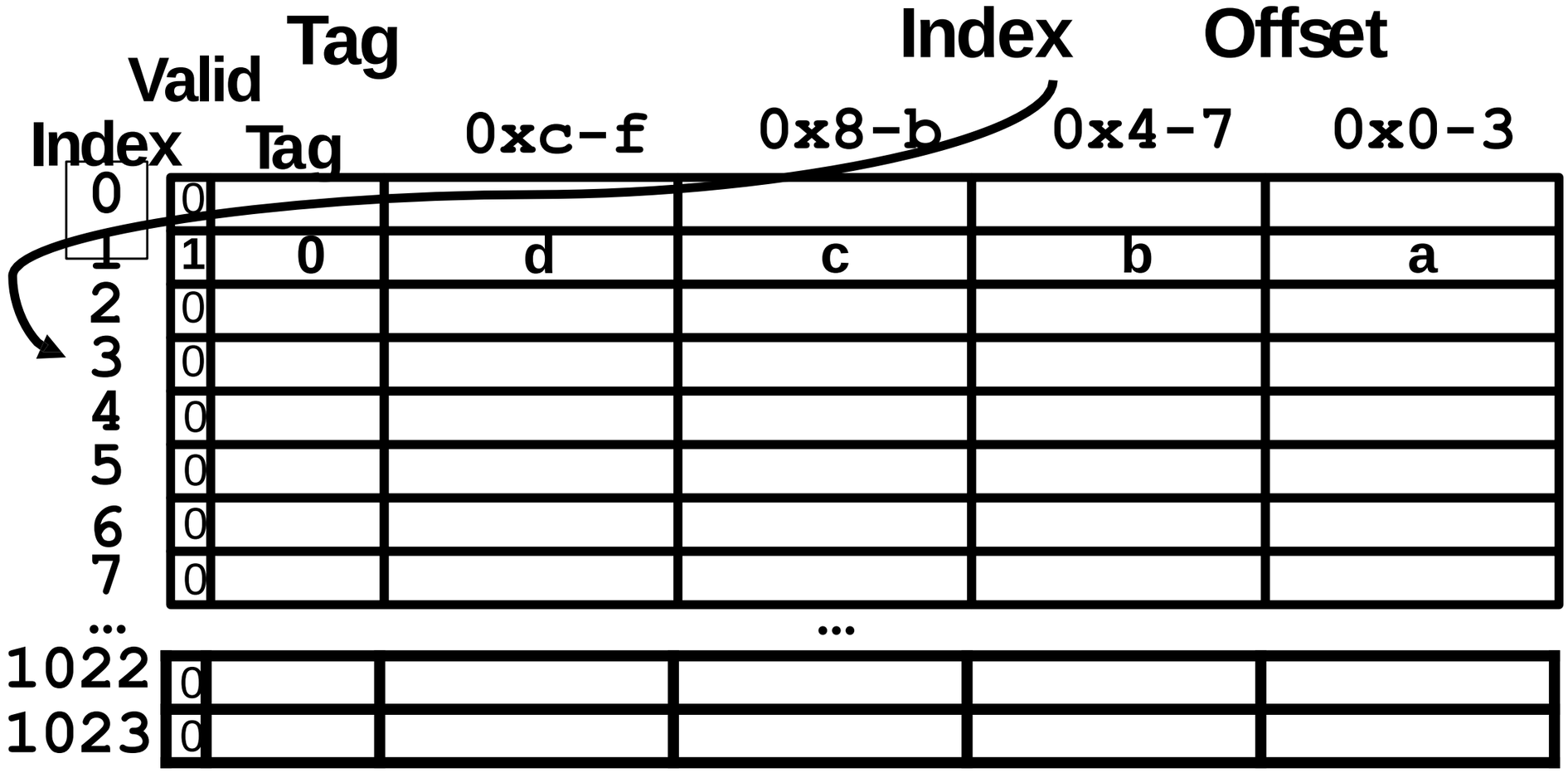
- 00000000000000000000 0000000011 0100

Tag | Index | Offset



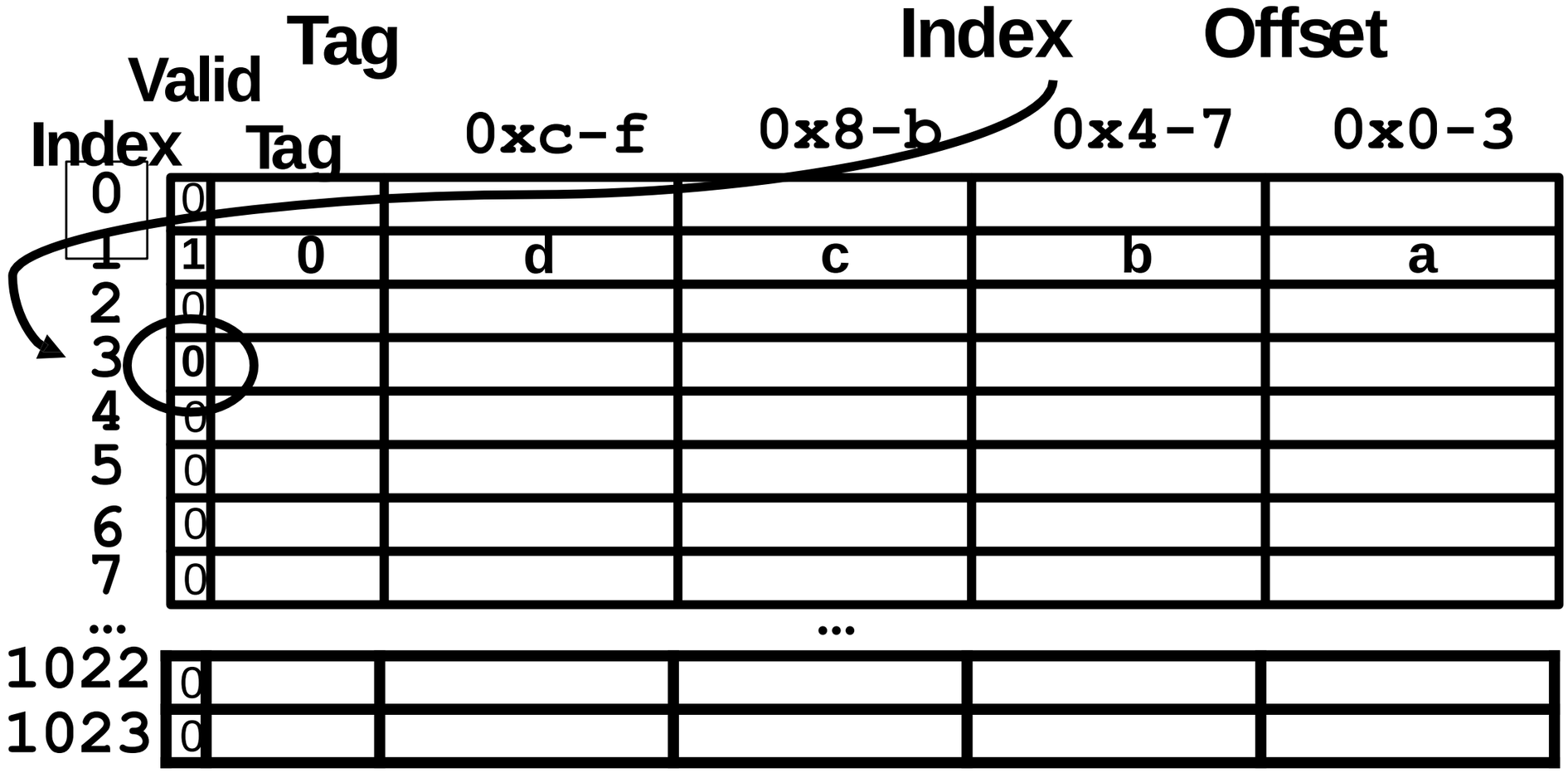
So read block 3

- 00000000000000000000 0000000011 0100



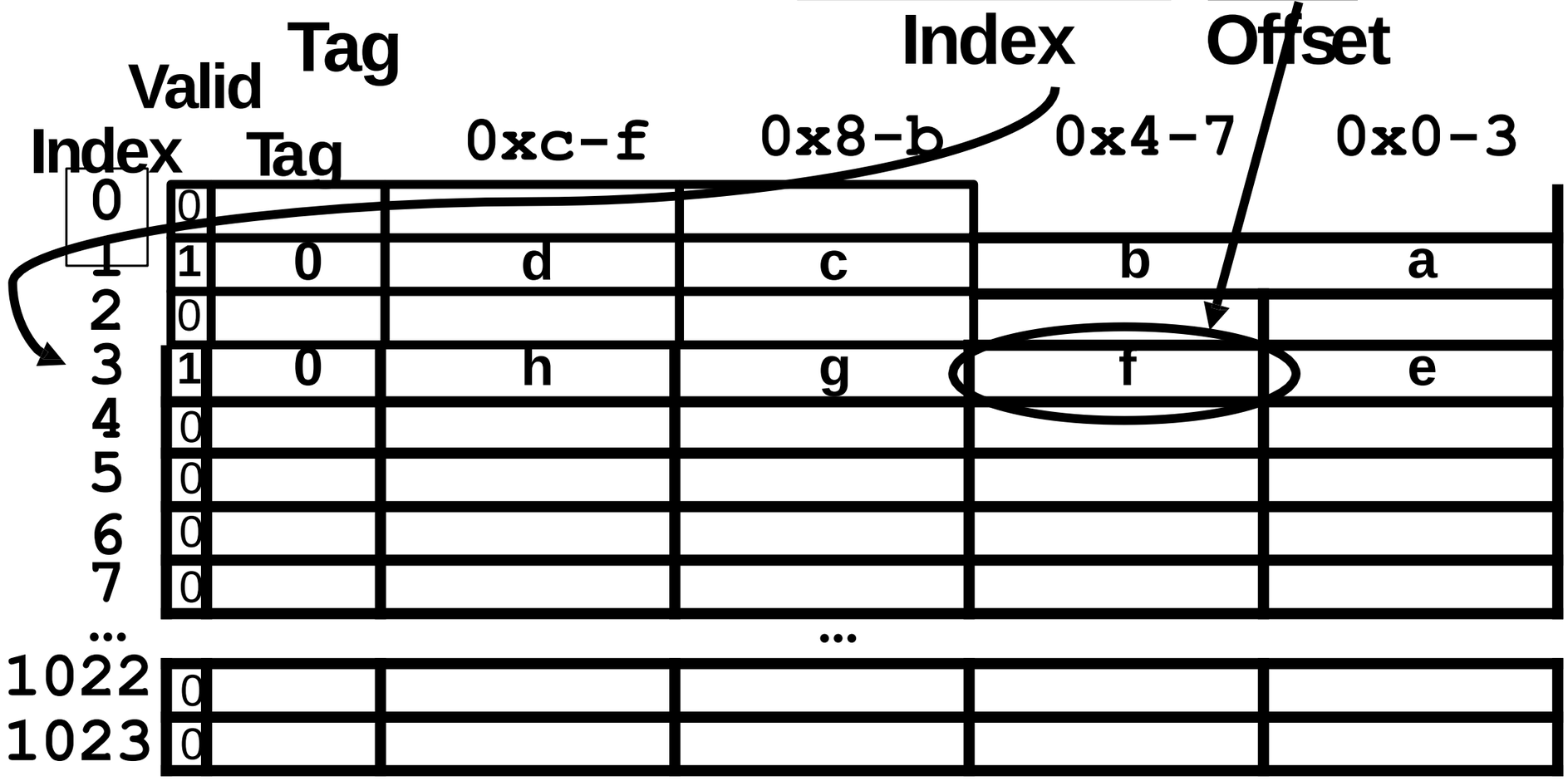
No valid data

- 00000000000000000000 0000000011 0100



Load that cache block, return word f

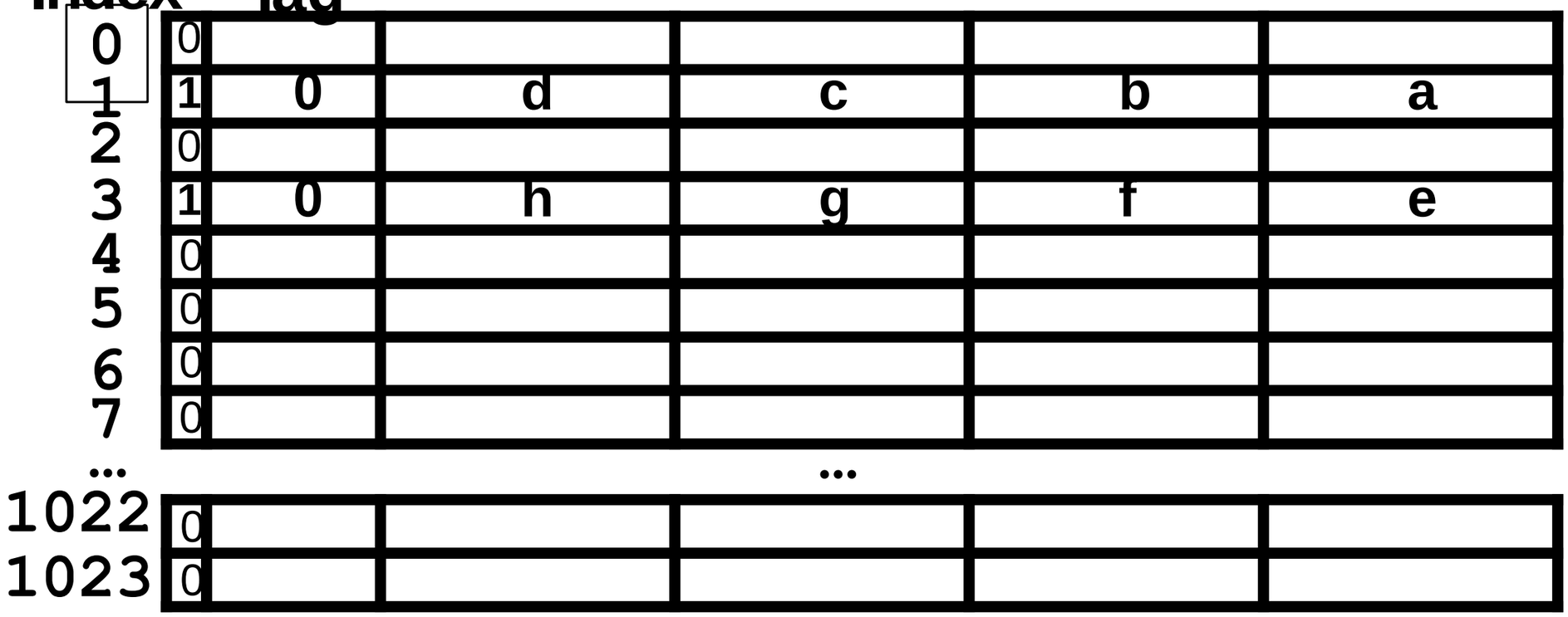
- 00000000000000000000 0000000011 0100



4. Read 0x00008014 = 0…10 0..0010100

- 00000000000000010 0000000001 0100

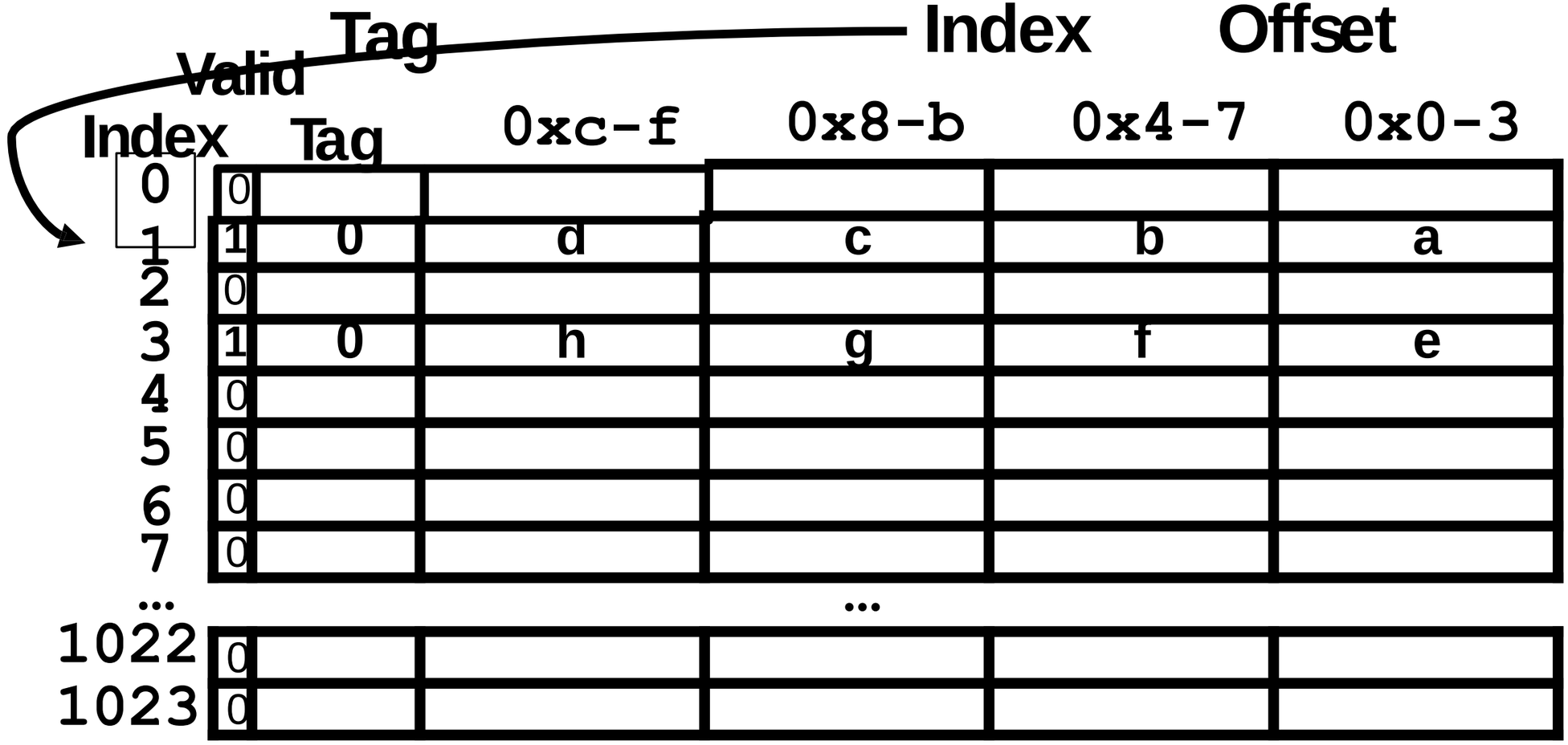
Tag | Index | Offset



So read Cache Block 1, Data is Valid

- 00000000000000010 0000000001 0100

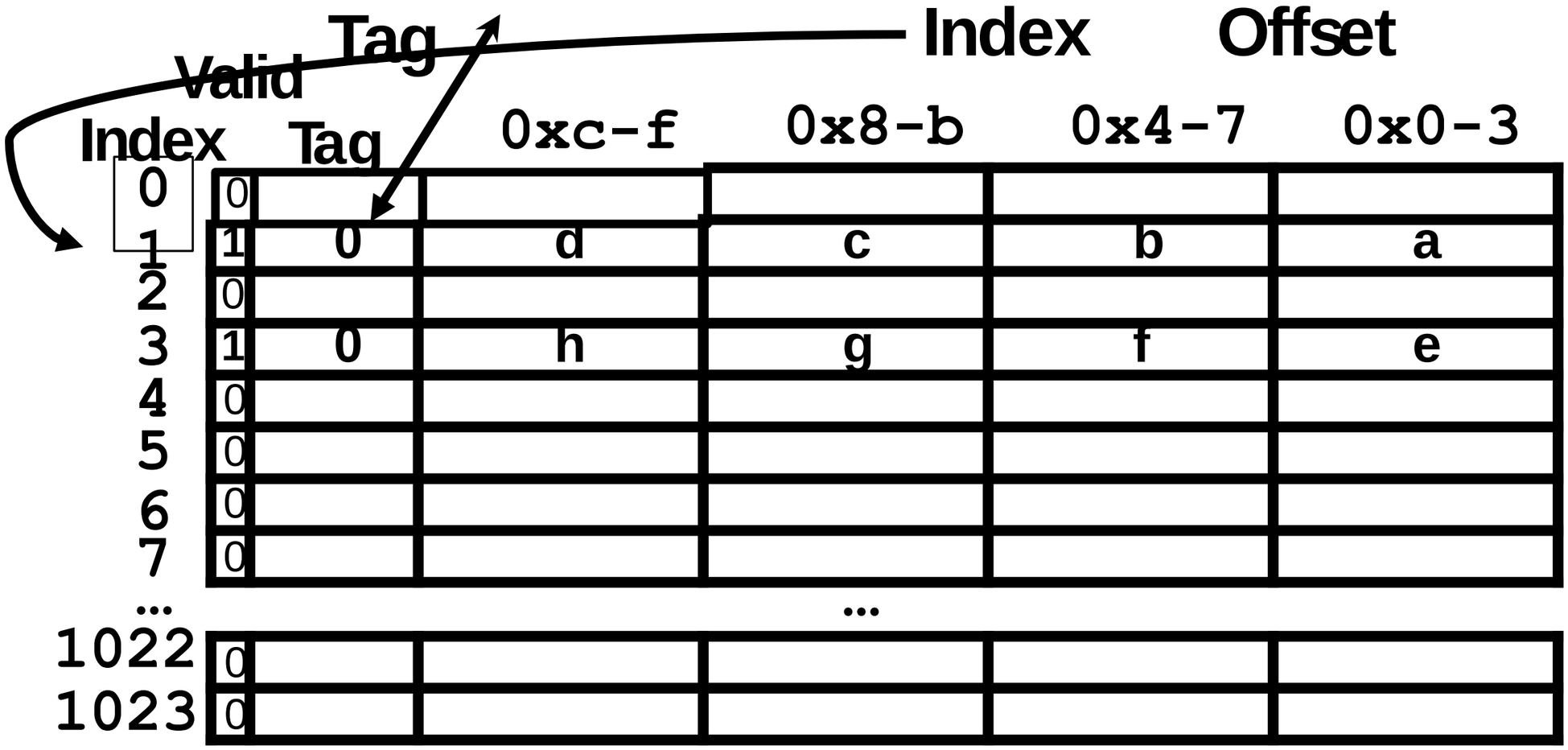
Tag | Index | Offset



Garcia, Nikolić

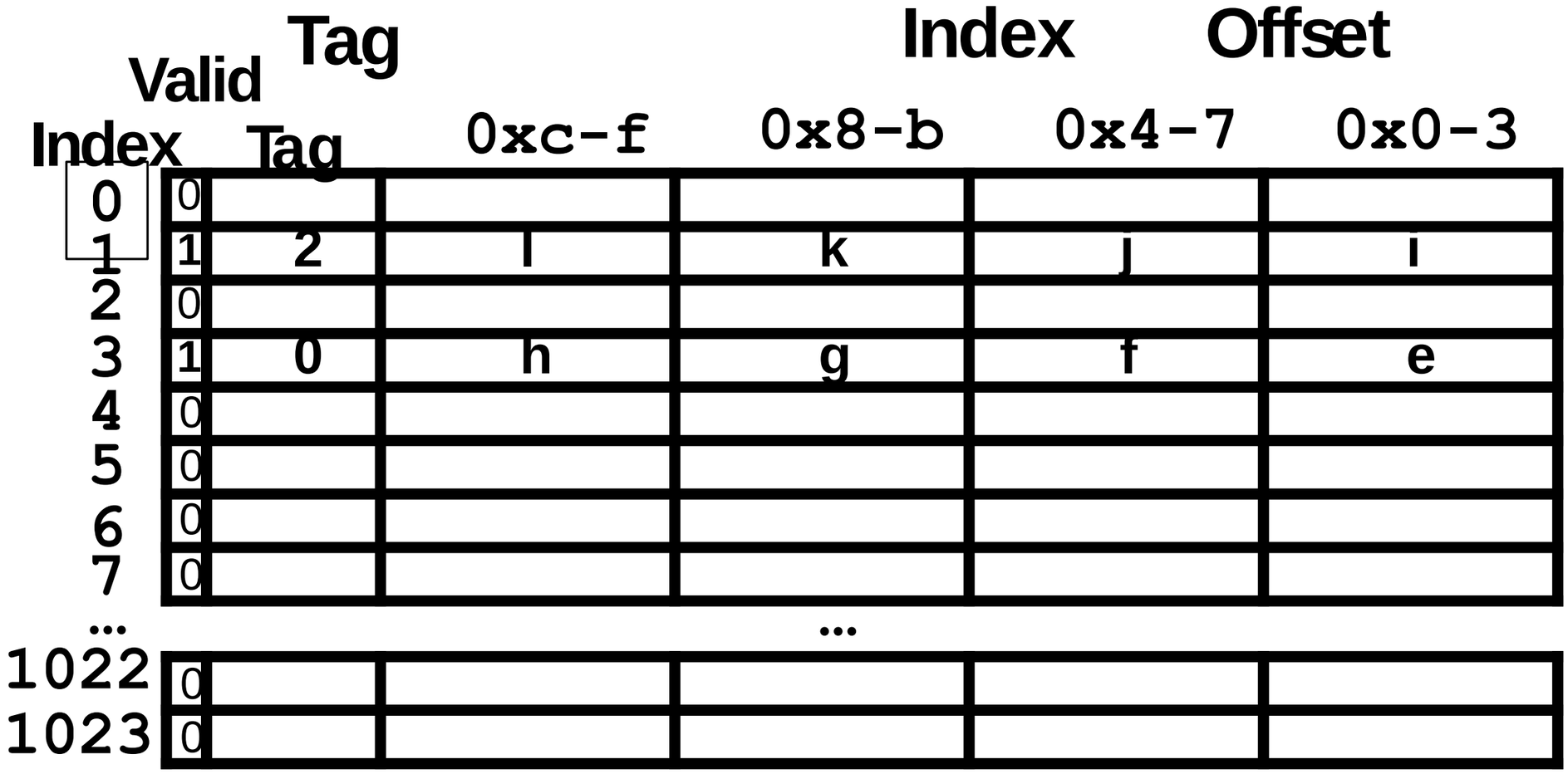
Cache Block 1 Tag does not match (0 ≠ 2)

- 0000000000000010 0000000001 0100



Miss, so replace block 1 with new data & tag

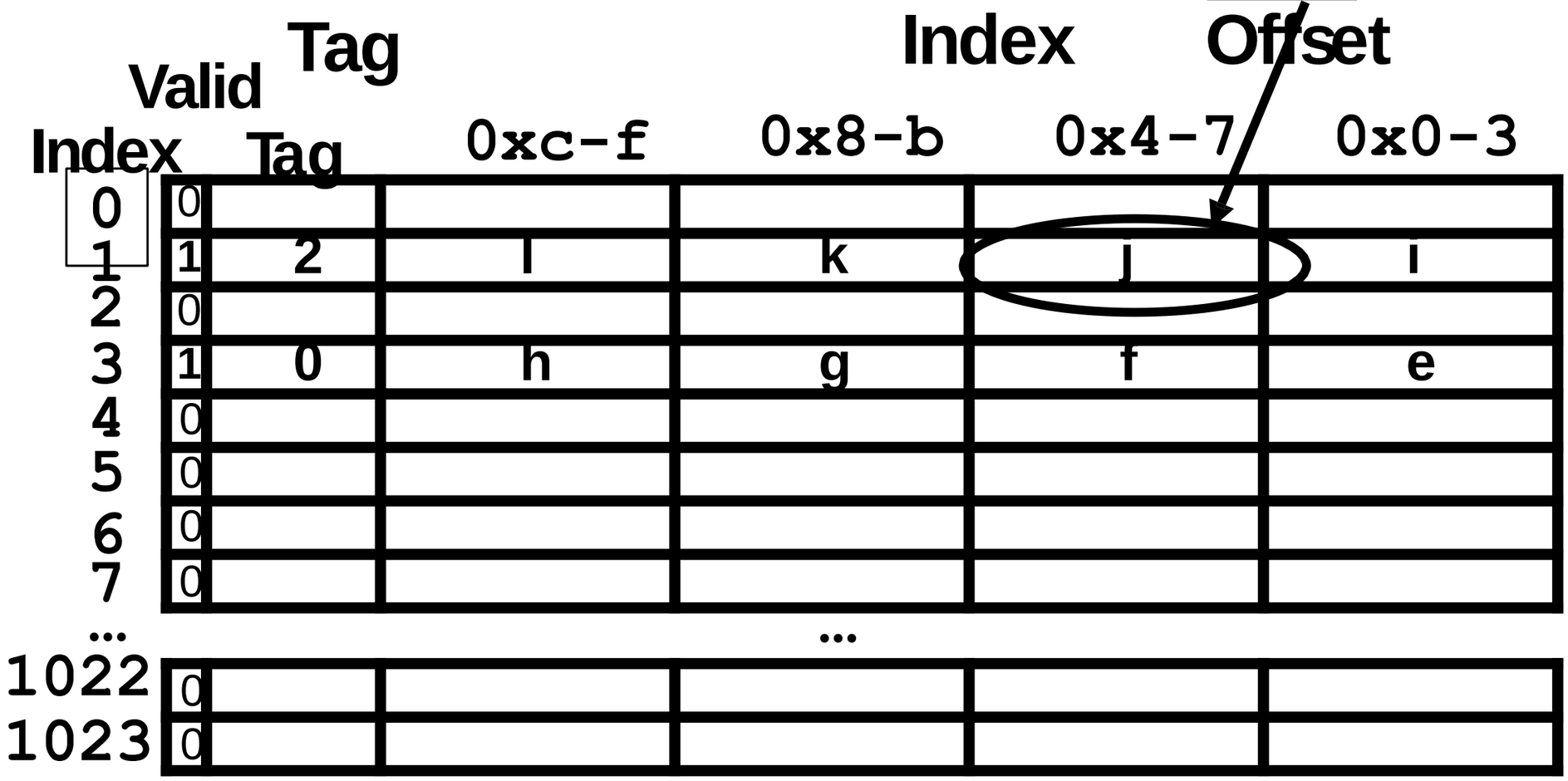
- 00000000000000010 0000000001 0100



Garcia, Nikolić

And return word J

- 0000000000000010 0000000001 0100



Do an example yourself. What happens?

- Chose from: Cache: Hit, Miss, Miss w. replace
  Values returned: a ,b, c, d, e, ..., k, l
- Read address 0x00000030 ?
  0000000000000000 0000000011 0000
- Read address 0x0000001c ?
  0000000000000000 0000000001 1100

Index						
0	0					
1	1	2	l	k	j	i
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					

Answers

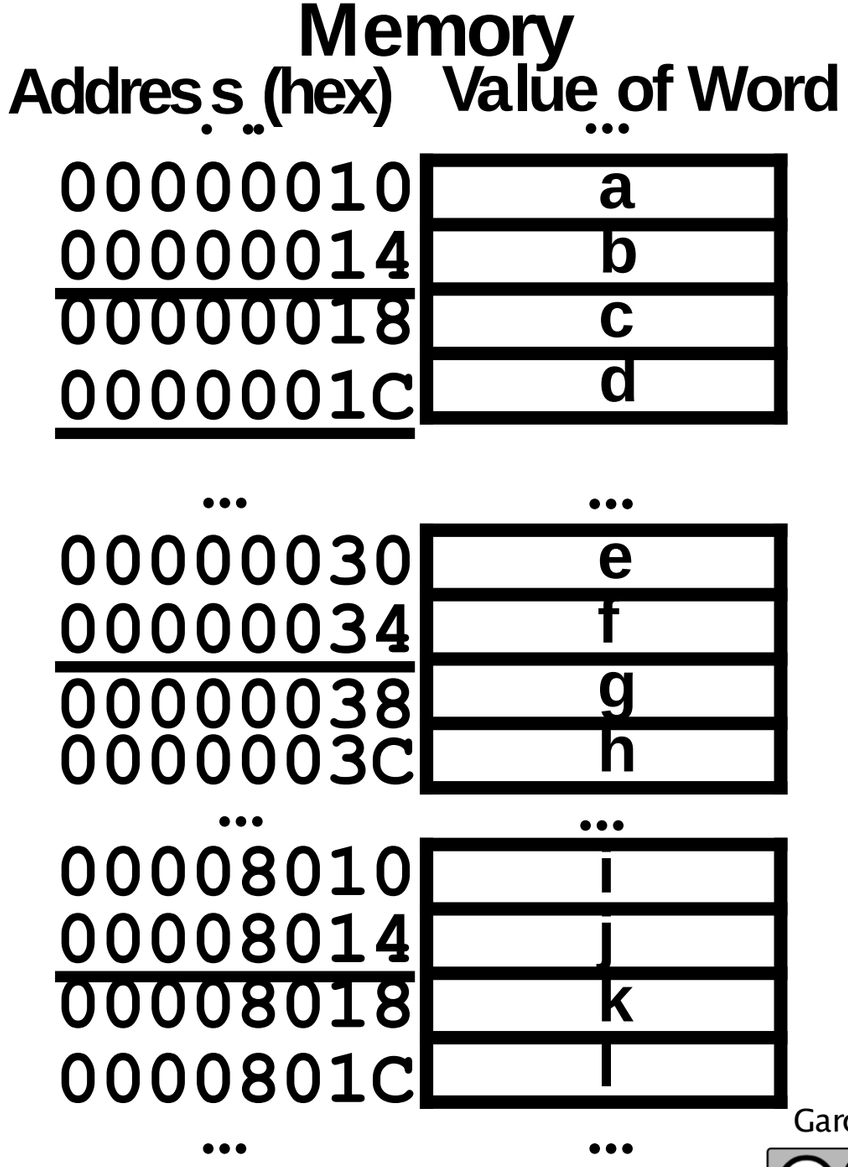
- `0x00000030` a **hit**

  Index = 3, Tag matches,
  Offset = 0, value = e

- `0x0000001c` a **miss**

  Index = 1, Tag mismatch, so
  replace from memory,
  Offset = `0xc`, value = d

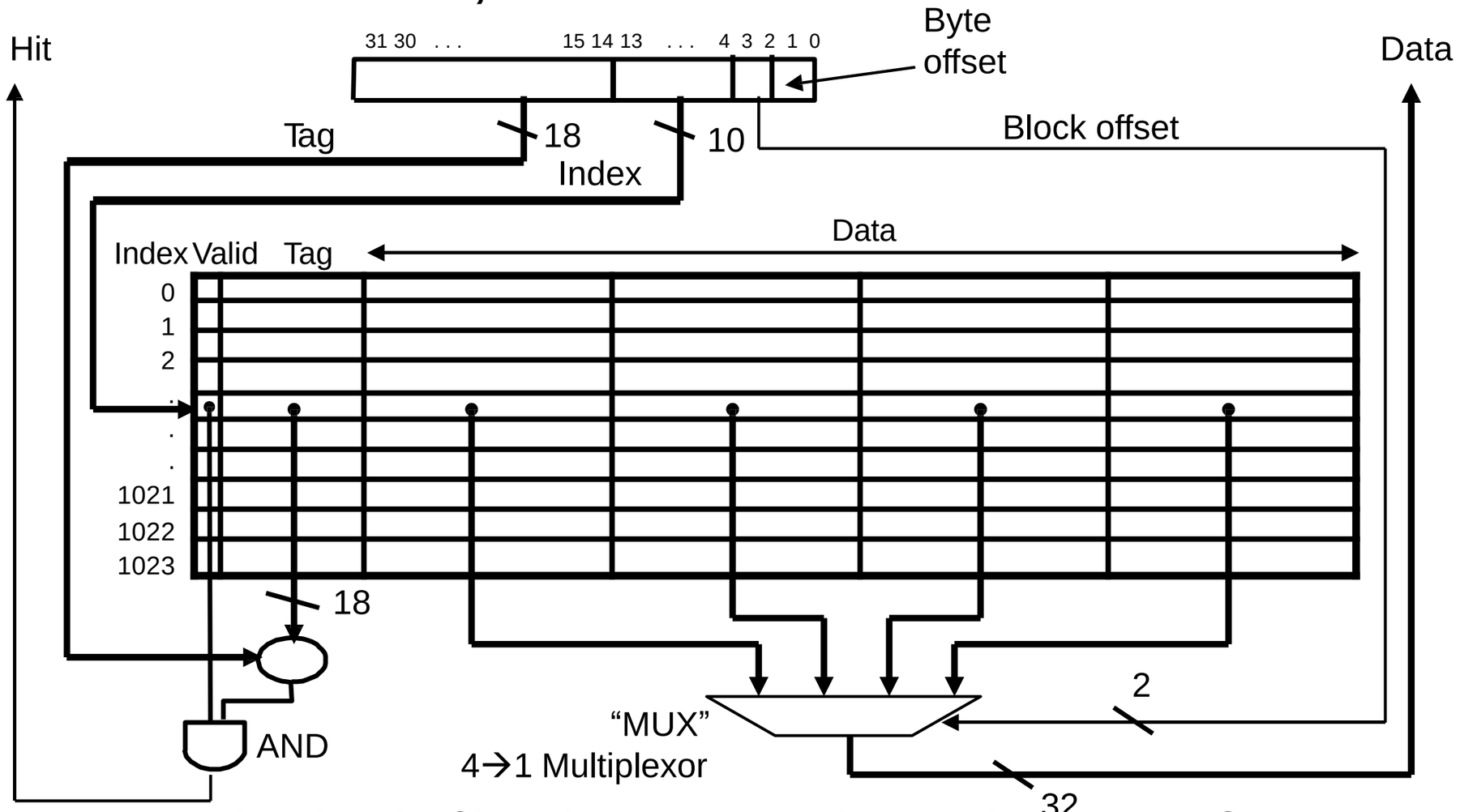
- **Since reads, values must = memory values whether or not cached:**
  - `0x00000030` = e
  - `0x0000001c` = d



Writes, Block Sizes, Misses

Multiword-Block Direct-Mapped Cache

- Four words/block, cache size = 4K words



What kind of locality are we taking advantage of?

What to do on a write hit?

- **Write-through**
  - Update both cache and memory
- **Write-back**
  - update word in cache block
  - allow memory word to be "stale"
  - add 'dirty' bit to block
    - memory & Cache inconsistent
    - needs to be updated when block is replaced
  - ... OS flushes cache before I/O...
- **Performance trade-offs?**

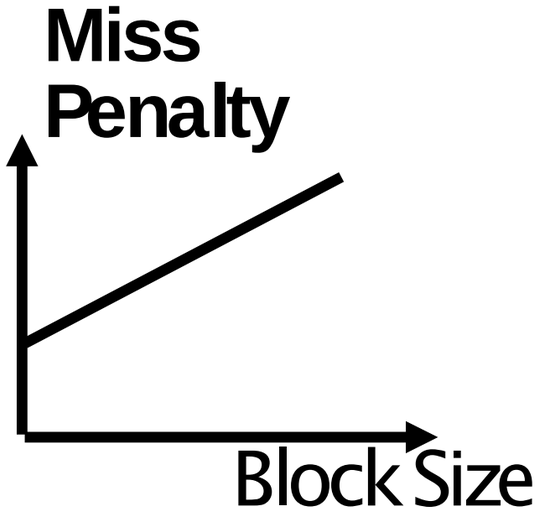
Block Size Tradeoff

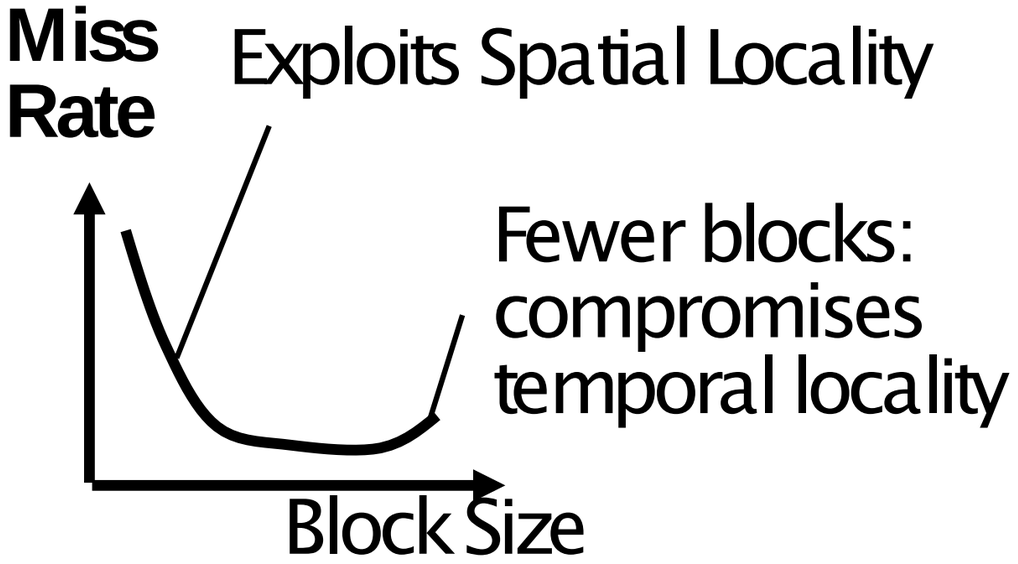
- **Benefits of Larger Block Size**
  - Spatial Locality: if we access a given word, we're likely to access other nearby words soon
  - Very applicable with Stored-Program Concept
  - Works well for sequential array accesses
- **Drawbacks of Larger Block Size**
  - Larger block size means larger miss 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

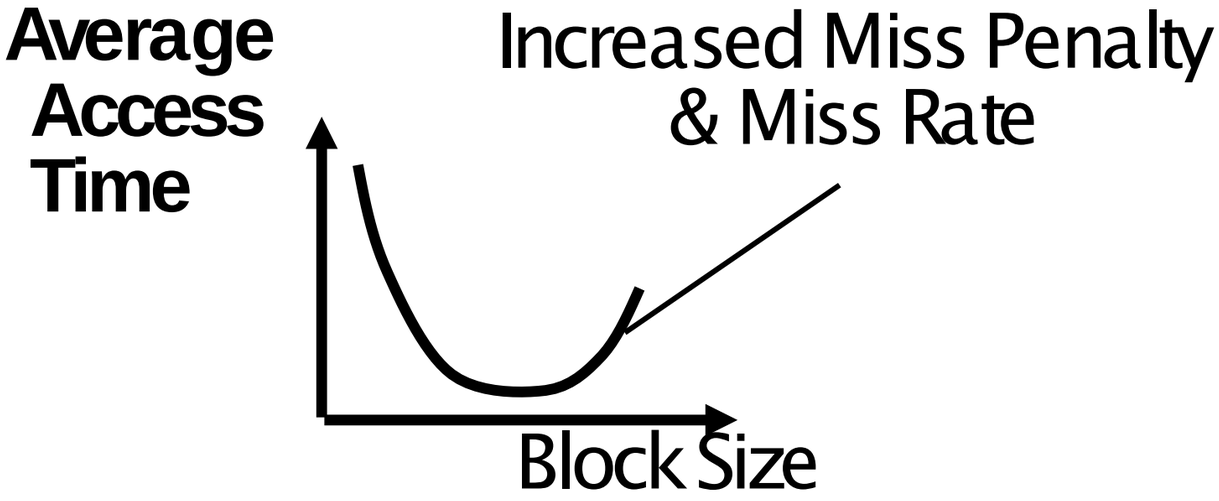
Extreme Example: One Big Block

- **Cache Size = 4 bytes Block Size = 4 bytes**
  - Only ONE entry (row) in the cache!
- **If item accessed, likely accessed again soon**
  - But unlikely will be accessed again immediately!
- **The next access will likely to be a miss again**
  - Continually loading data into the cache but discard data (force out) before use it again
  - Nightmare for cache designer: Ping Pong Effect

Block Size Tradeoff Conclusions







Types of Cache Misses (1/2)

- "Three Cs" Model of Misses
- 1st C: Compulsory Misses
  - occur when a program is first started
  - cache does not contain any of that program's data yet, so misses are bound to occur
  - can't be avoided easily, so won't focus on these in this course
  - Every block of memory will have one compulsory miss (NOT only every block of the cache)

Types of Cache Misses (2/2)

- **2nd C: Conflict Misses**
  - miss that occurs because two distinct memory addresses map to the same cache location
  - two blocks (which happen to map to the same location) can keep overwriting each other
  - big problem in direct-mapped caches
  - how do we lessen the effect of these?
- **Dealing with Conflict Misses**
  - Solution 1: Make the cache size bigger
    - Fails at some point
  - Solution 2: Multiple distinct blocks can fit in the same cache Index?

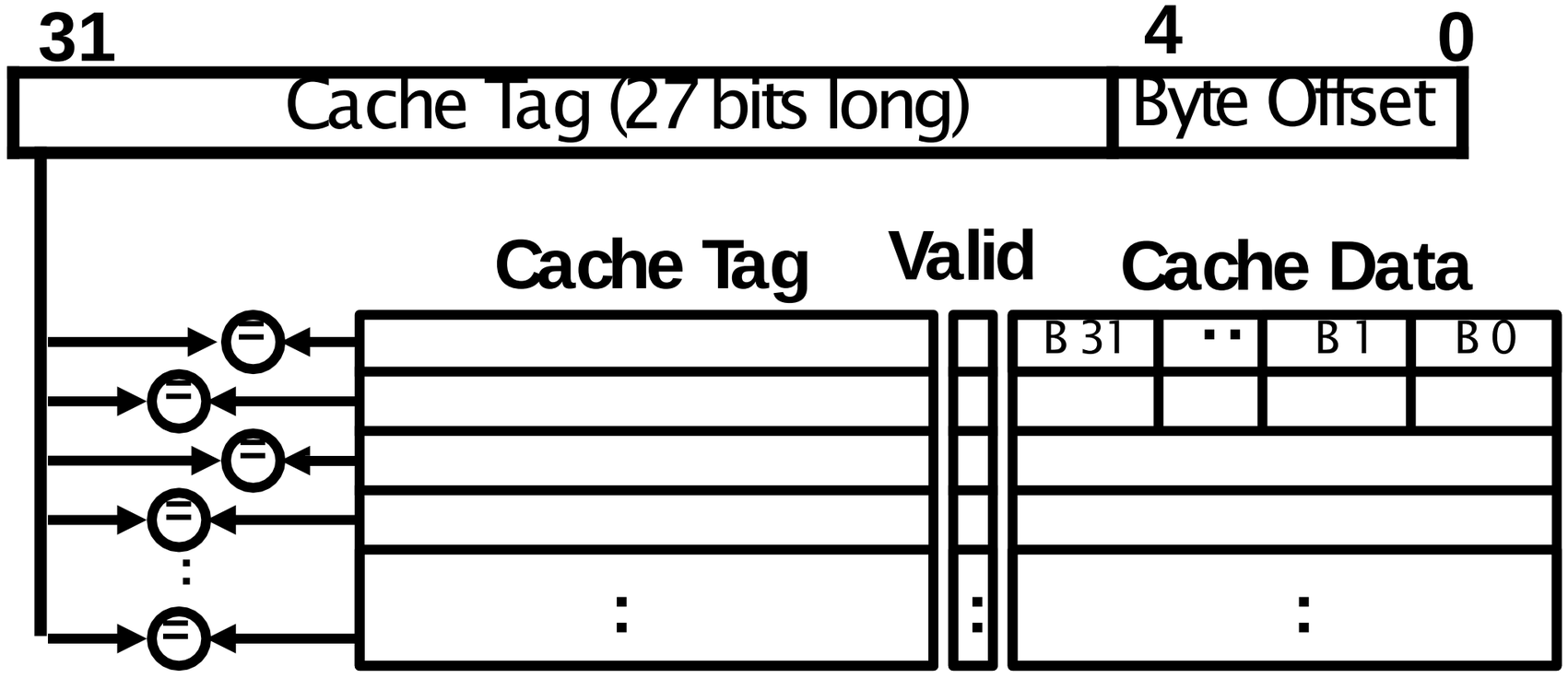
Fully Associative Caches

Fully Associative Cache (1/3)

- **Memory address fields:**
  - Tag: same as before
  - Offset: same as before
  - Index: non-existant
- **What does this mean?**
  - no “rows”: any block can go anywhere in the cache
  - must compare with all tags in entire cache to see if data is there

Fully Associative Cache (2/3)

- **Fully Associative Cache (e.g., 32 B block)**
  - compare tags in parallel



Fully Associative Cache (3/3)

- **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: infeasible

Final Type of Cache Miss

- **3rd C: 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
  - sketchy definition, so just get the general idea
- **This is the primary type of miss for Fully Associative caches.**

How to categorize misses

- **Run an address trace against a set of caches:**
  - First, consider an infinite-size, fully-associative cache. For every miss that occurs now, consider it a compulsory miss.
  - Next, consider a finite-sized cache (of the size you want to examine) with full-associativity. Every miss that is not in #1 is a capacity miss.
  - Finally, consider a finite-size cache with finite-associativity. All of the remaining misses that are not #1 or #2 are conflict misses.
  - (Thanks to Prof. Kubiatowicz for the algorithm)

And in Conclusion…

1. **Divide into T I O bits, Go to Index = I, check valid**
   1. If 0, load block, set valid and tag (COMPULSORY MISS) and use offset to return the right chunk (1,2,4-bytes)
   2. If 1, check tag
      1. If Match (HIT), use offset to return the right chunk
      2. If not (CONFLICT MISS), load block, set valid and tag, use offset to return the right chunk