Low on cache?

Outline

- Memory Hierarchy
- Direct-Mapped Cache
- Types of Cache Miss
- A (long) detailed exa

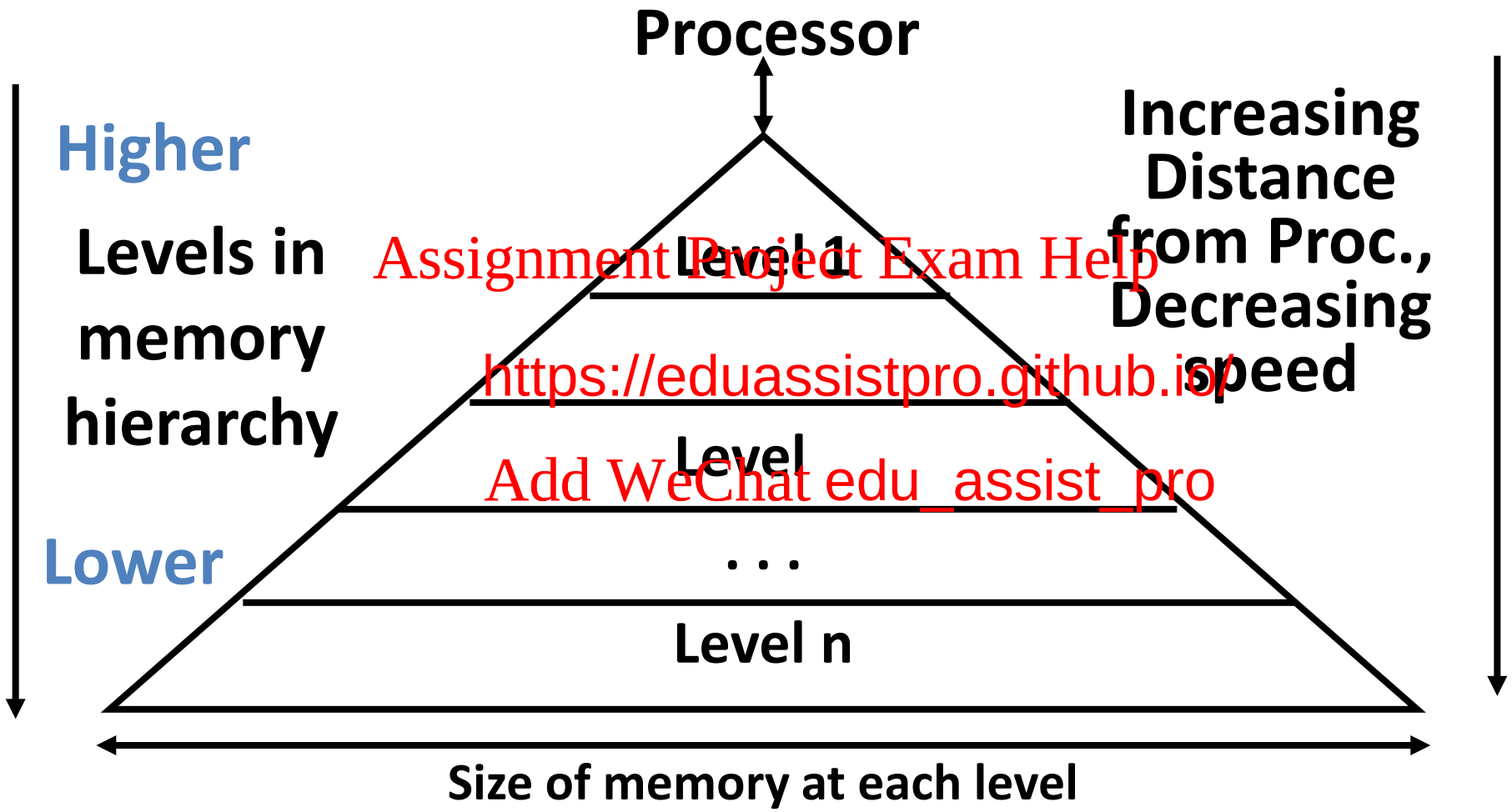
Memory Hierarchy (1/4)

- Processor
  - executes programs
  - runs on order of nan ds
  - needs to access cod s: where are these?
- Disk
  - HUGE capacity (virtually limitless)
  - VERY slow: runs on order of milliseconds
  - so how do we account for this gap?

Memory Hierarchy (2/4)

- Memory (DRAM)
  - smaller than disk (not limitless capacity)
  - contains subset of d ortions of programs that are currently being r
  - much faster than disk; memory ac ’t slow down processor quite as much
  - Problem: memory is still too slow (hundreds of nanoseconds)
  - Solution: add more layers (caches)

Memory Hierarchy (3/4)

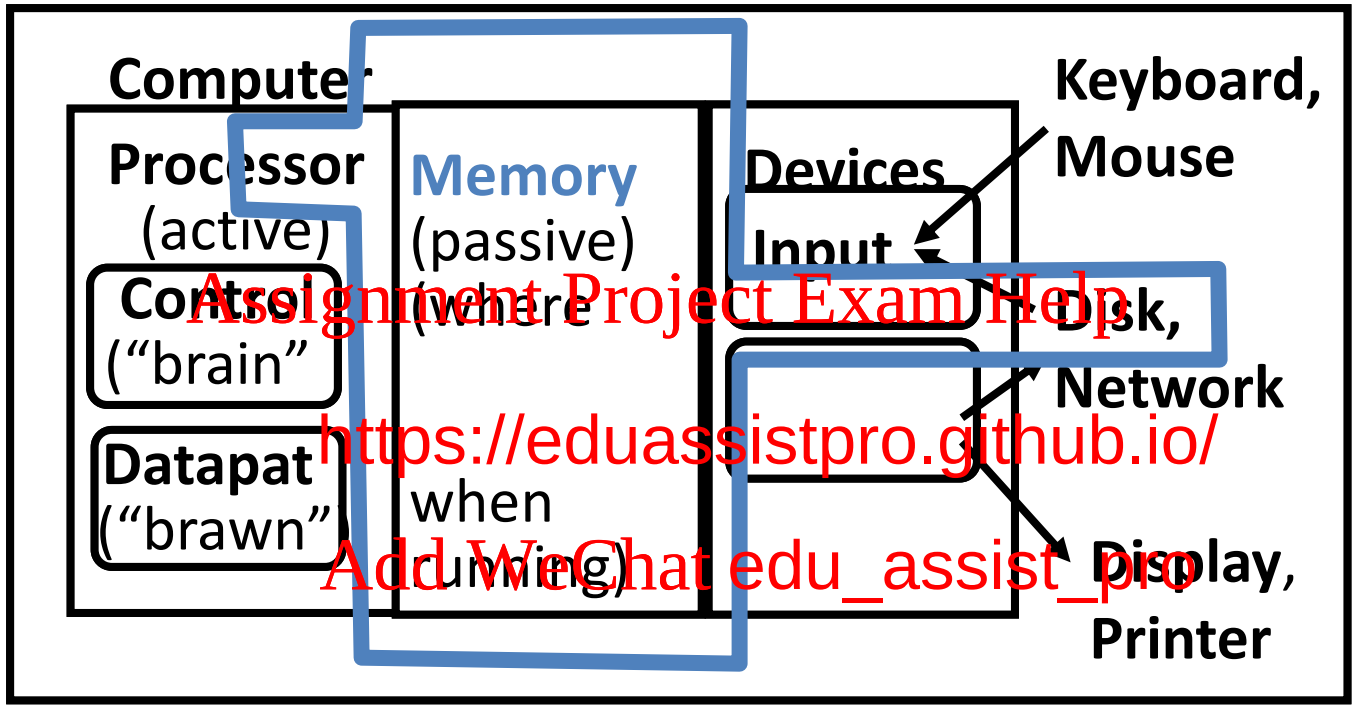


As we move to deeper levels the latency goes up and price per bit goes down.

Memory Hierarchy (4/4)

- If level is closer to Processor, it must…
  - Be smaller
  - Be faster
  - Contain a subset (m) of lower levels beneath it (i.e., levels farther from processor
  - Contain all the data in higher level (i.e., levels closer to processor)
- Lowest Level (usually disk) contains all available data
- Is there another level lower than disk?

Memory Hierarchy



- Purpose:
  - Faster access to large memory from processor

Memory Hierarchy Analogy: Library (1/2)

- You (the processor) are writing a term paper at a table in Schulich
- Schulich Library is e
  - essentially limitless
  - 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

Memory Hierarchy Analogy: Library (2/2)

- Open books on table are cache
  - smaller capacity: can have very few open books fit on table; again, when table fills up, you must close a book
  - much, much faster t
- Illusion created: whole library on tabletop
  - Keep as many recently used books open on table as possible since likely to use again
  - Also keep as many books on table as possible, since faster than going to library

Memory Hierarchy Basis

- Disk contains everything
- When Processor needs something, bring it into to all lower levels of memory
- Cache contains copi ry that are being used
- Memory contains copies of data hat are being used
- Entire idea is based on ***Temporal Locality***: if we use it now, we'll want to use it again soon (a Big Idea)

Athlon XP-64 Core

- The greatest share of the surface (over 50 percent) is taken up by the 1 MB L2 cache.

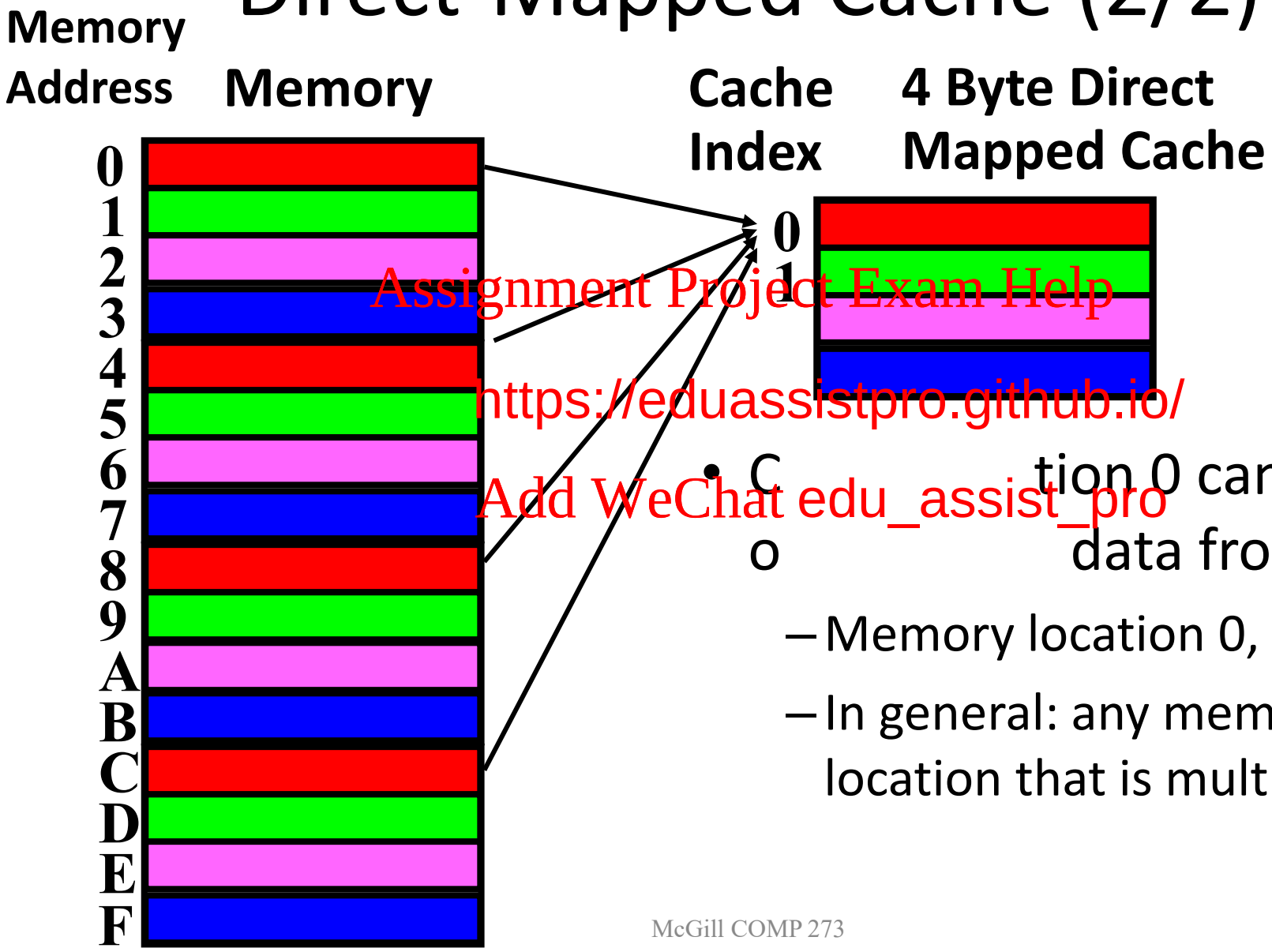
Cache Design

- How do we organize cache?
- Where does each memory address map to?
  - Remember that cac , so multiple memory addresses map to th .
- How do we know which elemen cache?
- How do we quickly locate them?

Direct-Mapped Cache (1/2)

- In a ***direct-mapped cache***, each memory address is associated with one possible ***block*** within the cache
  - Therefore, we only location in the cache to see if the data exists
  - A *block* is the unit of transfer bet and memory

Direct-Mapped Cache (2/2)



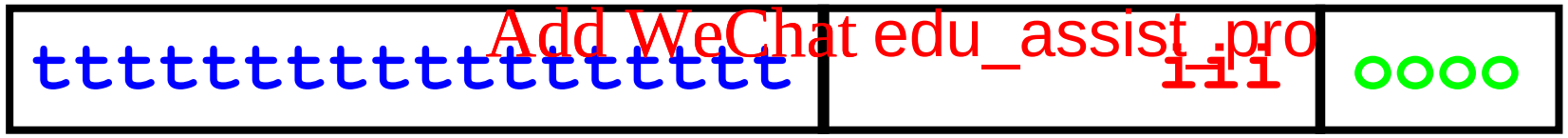
- C tion 0 can be
  - o data from:
    - Memory location 0, 4, 8, ...
    - In general: any memory location that is multiple of 4

Issues with Direct-Mapped

1 Since multiple memory addresses map to same cache index, how do we tell which one is in there?

2 What if we have a block size > 1 byte?

- **Solution**: divide memory into fields



tag to check if have correct block

index to select block

offset byte within block

Direct-Mapped Cache Terminology

- All fields are read as unsigned integers.
- The ***Index***: specifies the cache index (which "row" of the cache we should look in)
- The ***Offset***: once we [illegible] lock, specifies which byte within the block we want
- The ***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

Direct-Mapped Cache Example

- Suppose we have a direct-mapped **16KB cache** with **4 word blocks**.
- Determine the size offset fields if we're using a 32-bit archit

Direct-Mapped Cache Example

- Offset
  - need to specify correct byte within a block
  - block contains

    4 words = 16
  - need **4 bits** to specify correct byte

Direct-Mapped Cache Example

- Index
  - need to specify correct row in cache
  - cache contains 16 KB = $2^4 \cdot 2^{10} = 2^{14}$ bytes
    block contains 2^4 byt

rows/cache = # blocks/cache (th lock/row)

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

$$= \frac{2^{14}\ \text{bytes/cache}}{2^4\ \text{bytes/row}}$$

$$= 2^{10}\ \text{rows/cache}$$

  - need ***10 bits*** to specify this many rows

Direct-Mapped Cache Example

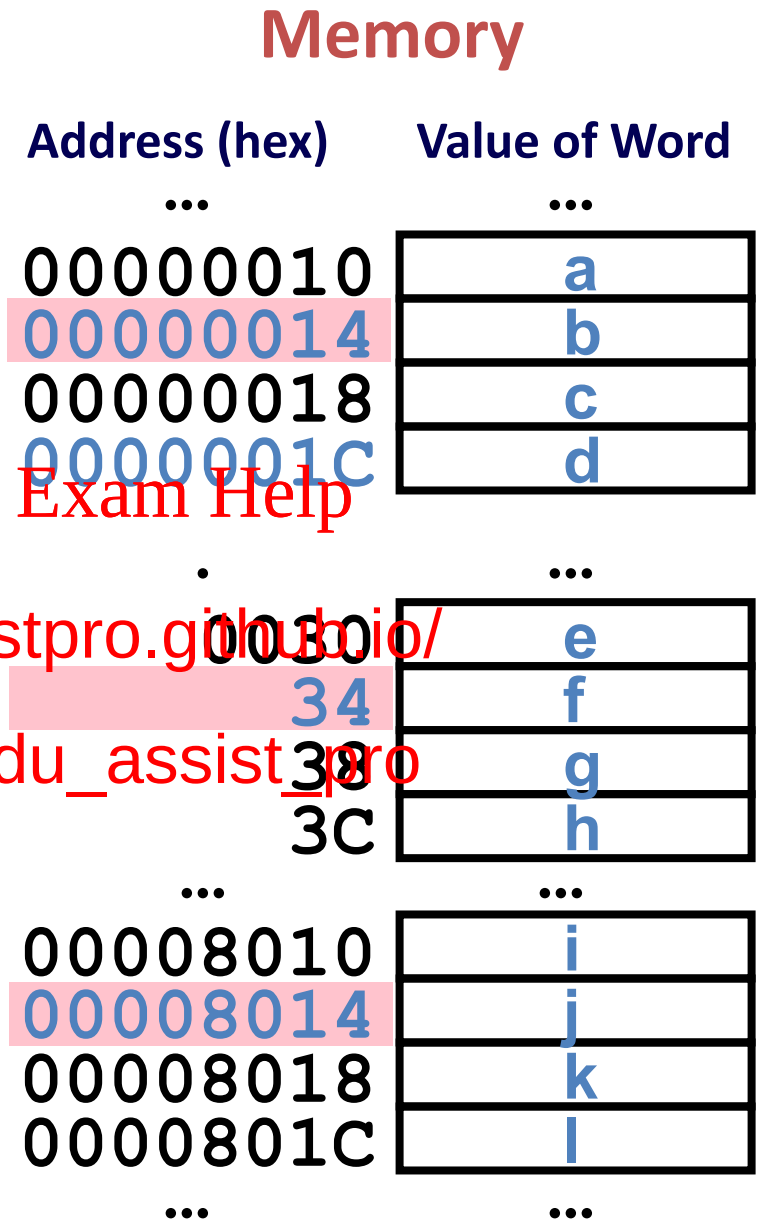
- Tag
  - used remaining bits as tag
  - tag length = memory fset bits minus index bits
    = 32 - 4 -
    = 18 bits
  - so the tag is leftmost ***18 bits*** of memory address

Accessing data in a direct mapped cache

- Example: 16KB, direct-mapped, 4 word blocks
- Read 4 addresses

  0x00000014
  0x0000001C
  0x00000034
  0x00008014

- Memory values on right:
  - Let us only consider cache and memory levels of hierarchy



Accessing data in a direct mapped cache

- 4 Addresses:

  0x00000014, 0x0000001C, 0x00000034, 0x00008014

- 4 Addresses divided (for convenience) into Tag, Index,

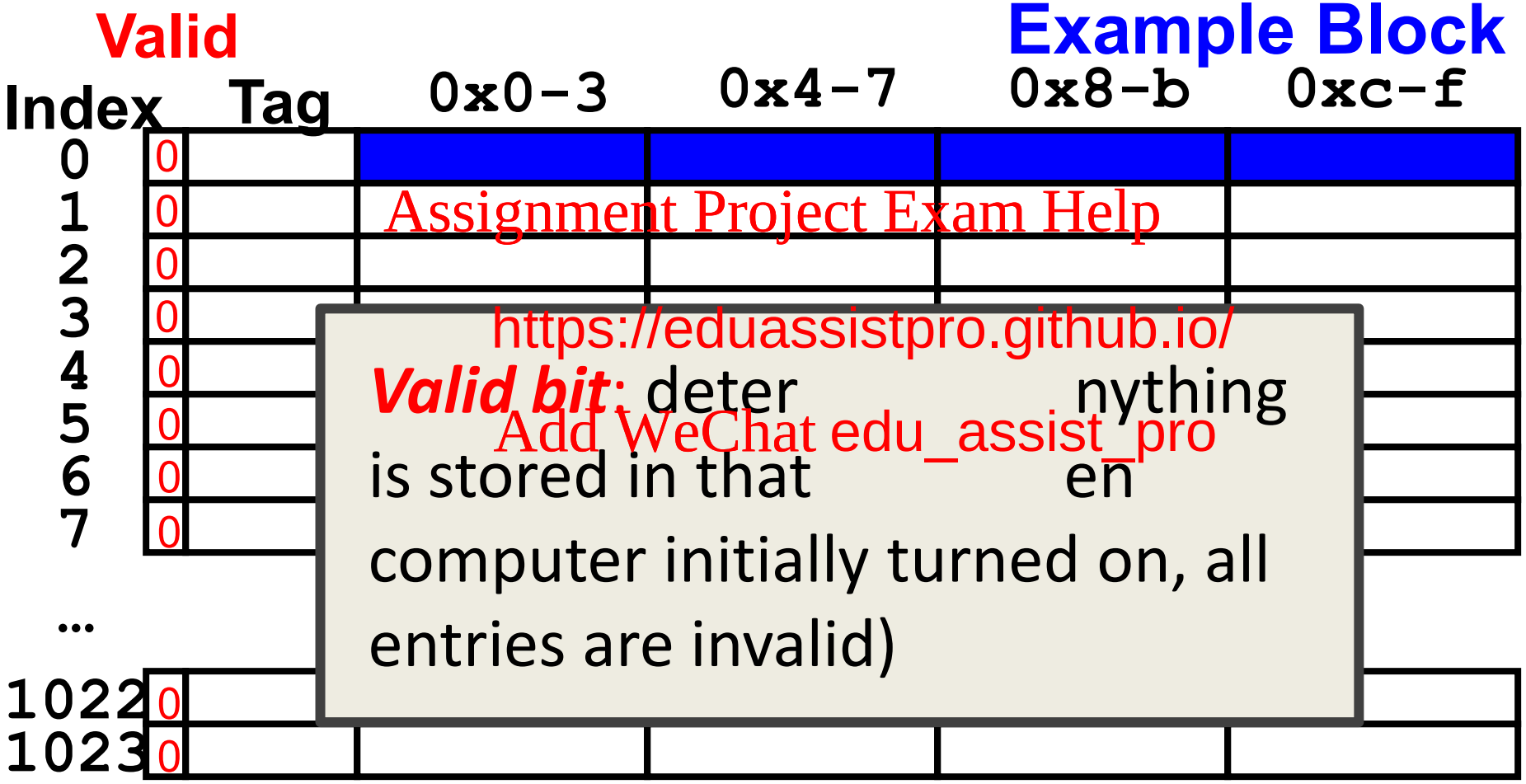

```
000000000                          00001 0100
00000000000000000                   0001 1100
00000000000000000 0000000011 0100
00000000000000010 0000000001 0100
```

Tag **Index** **Offset**

Accessing data in a direct mapped cache

- 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 c ck, so fetch from memory
- cache hit: cache block is valid and con r 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

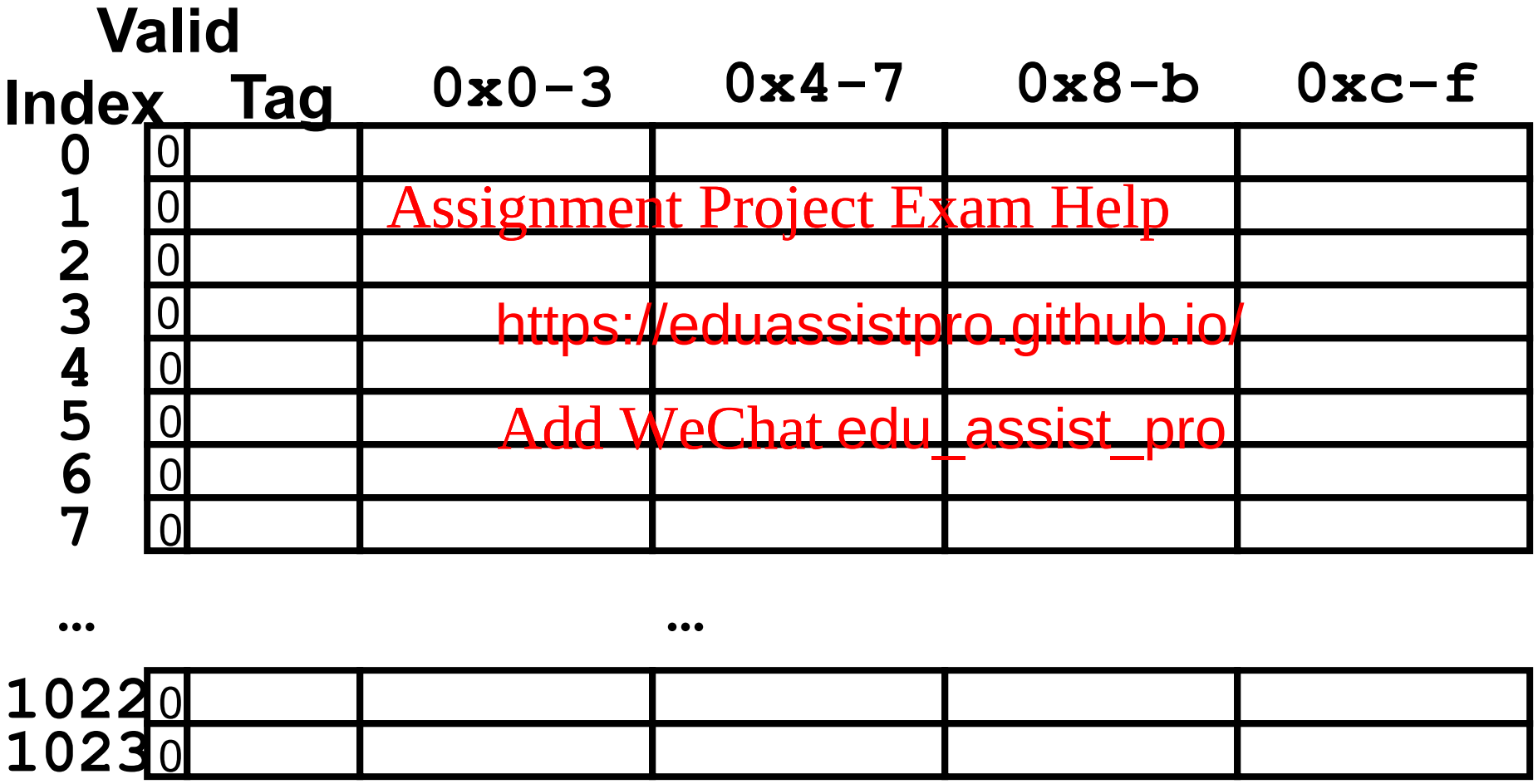
16 KB Direct Mapped Cache, 16B blocks



Read 0x00000014

0000000000000000000 0000000001 0100

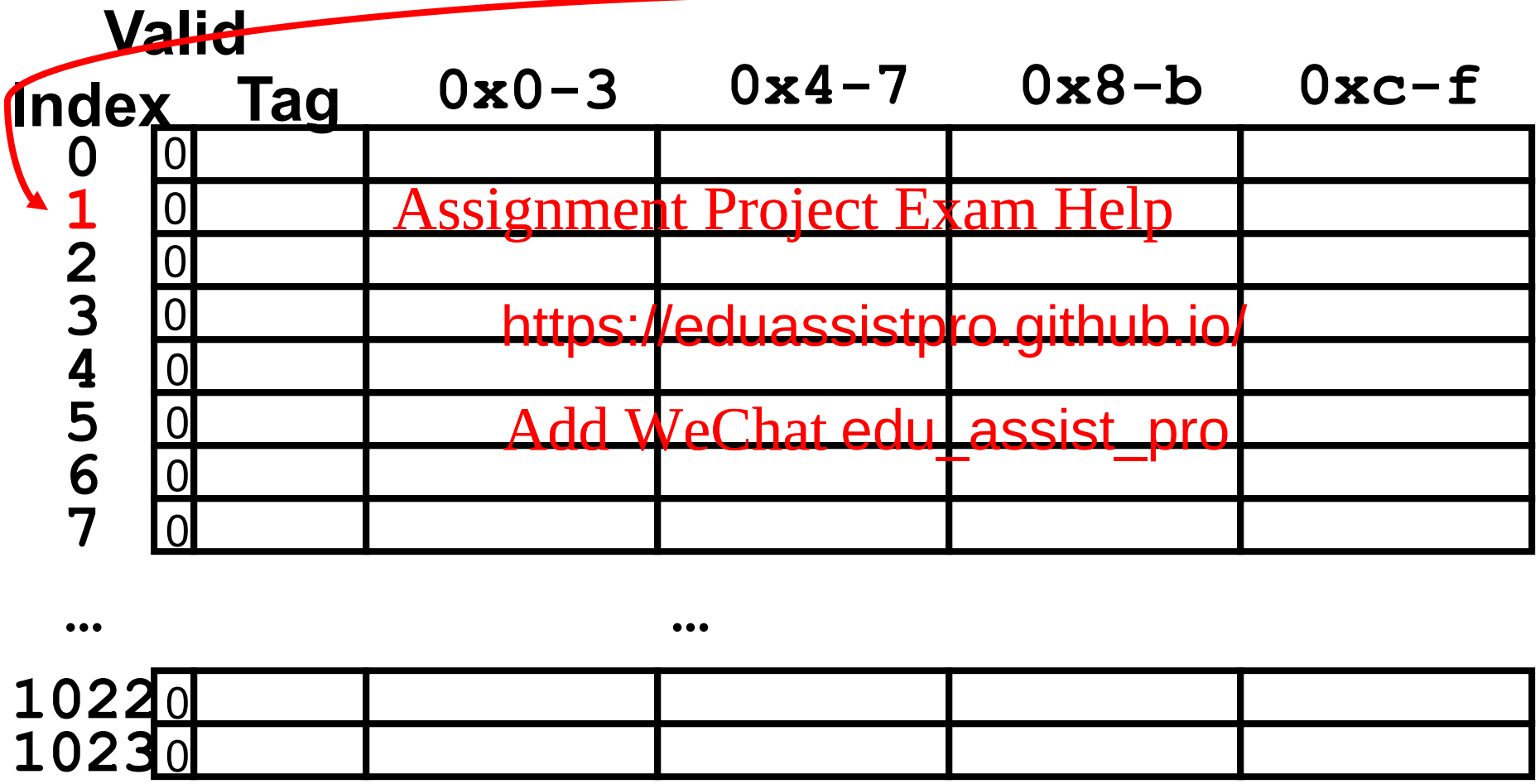
Tag field **Index field** **Offset**



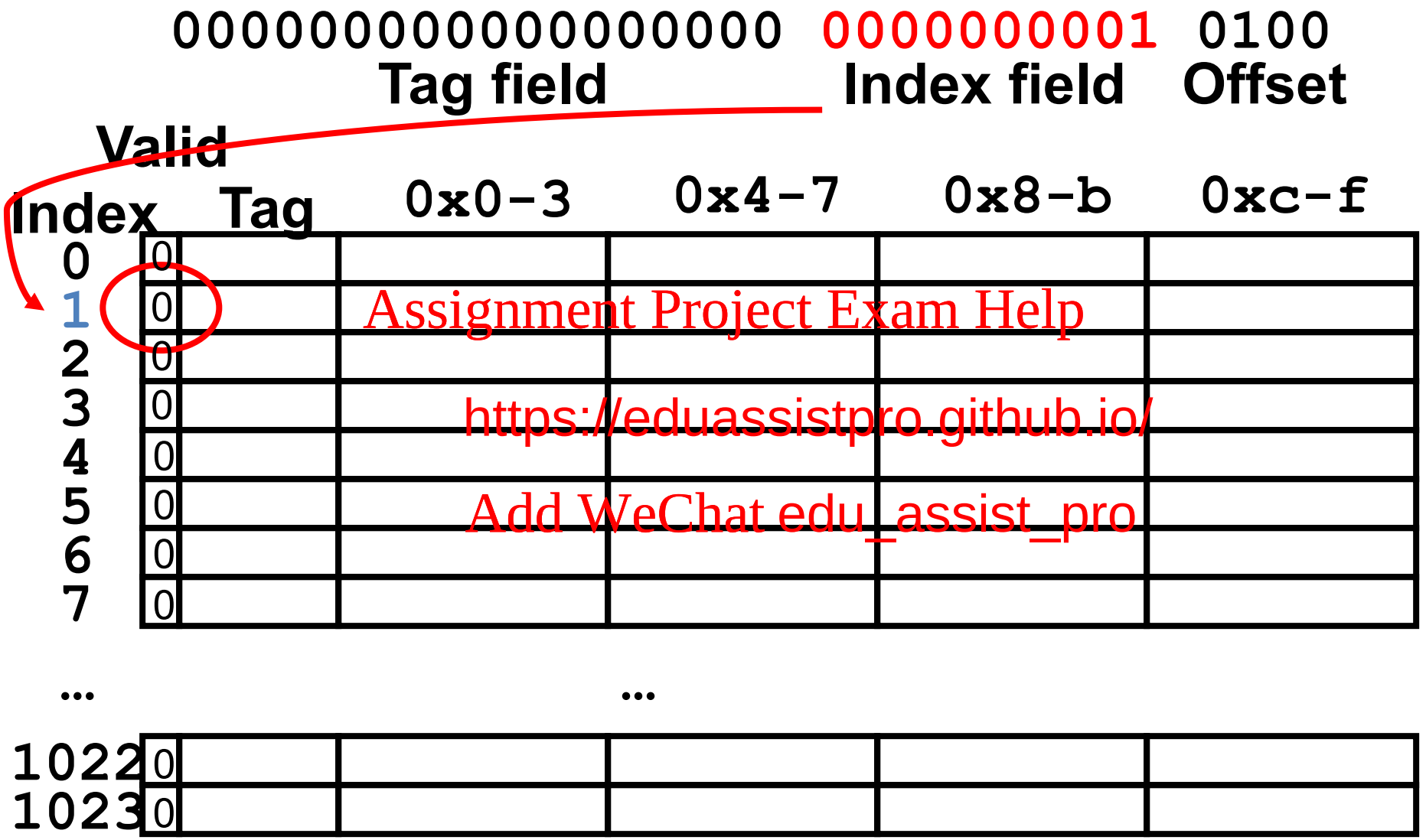
So we read block 1 (0000000001)

0000000000000000 0000000001 0100

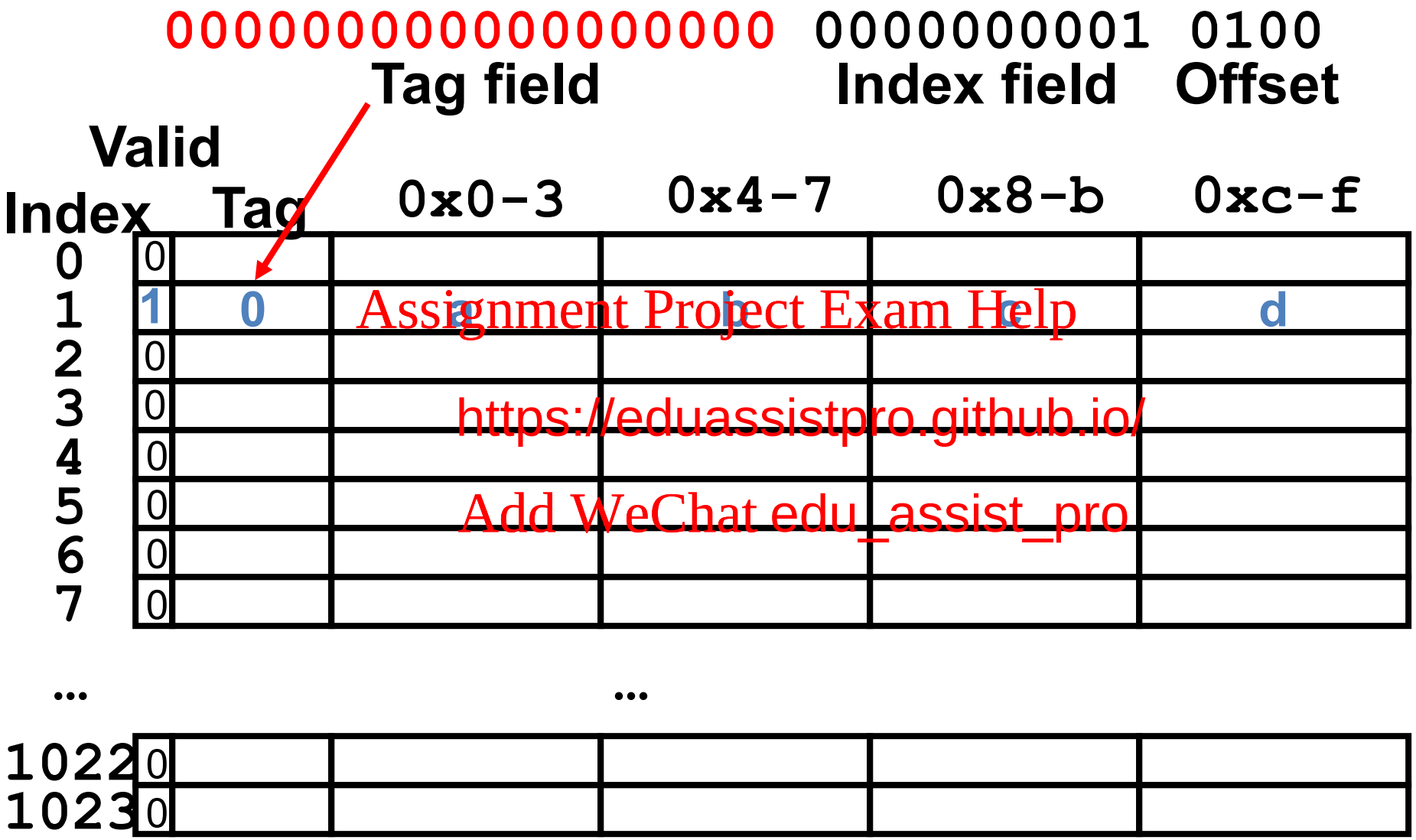
Tag field | Index field | Offset



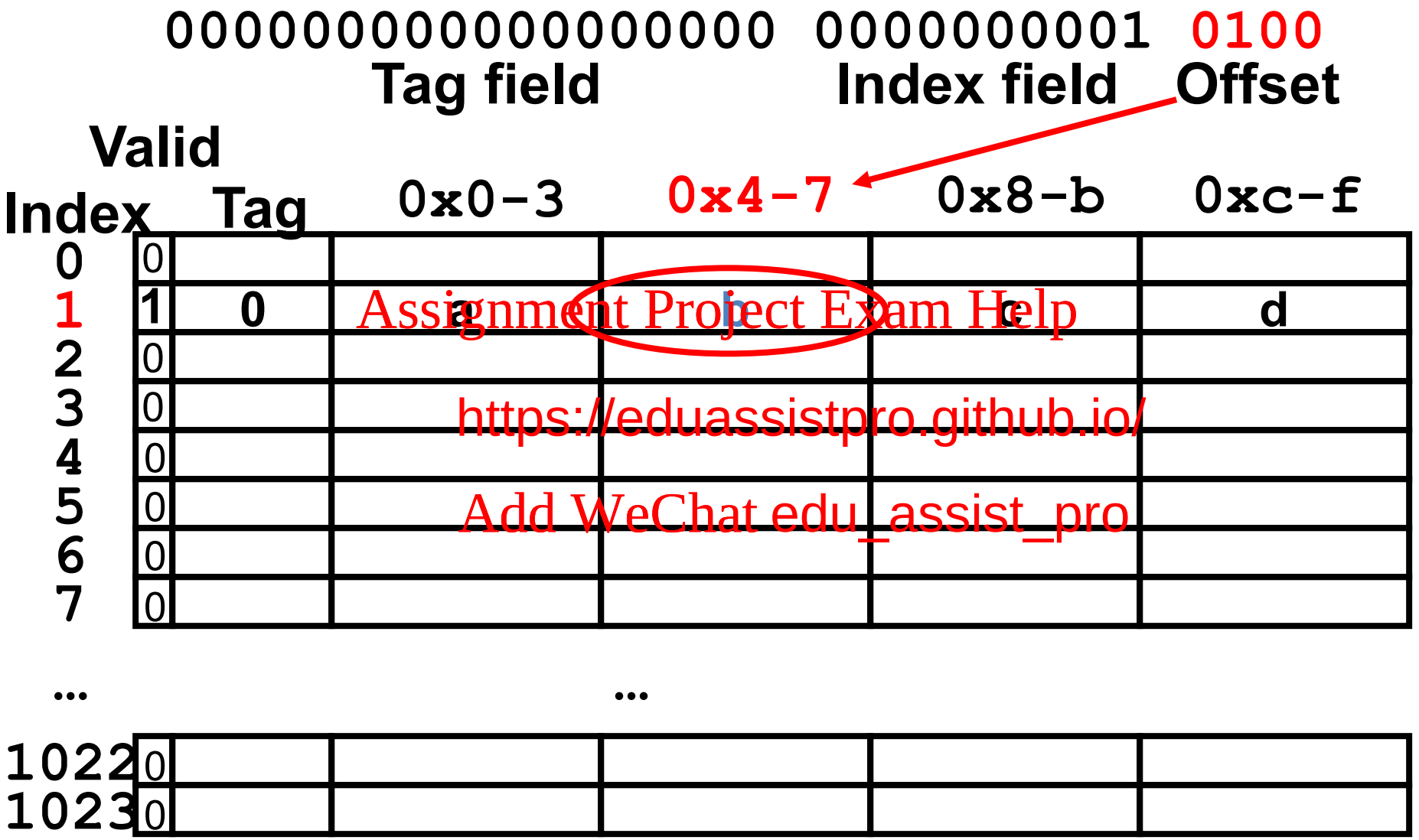
No valid data



So load that data into cache, setting tag, valid



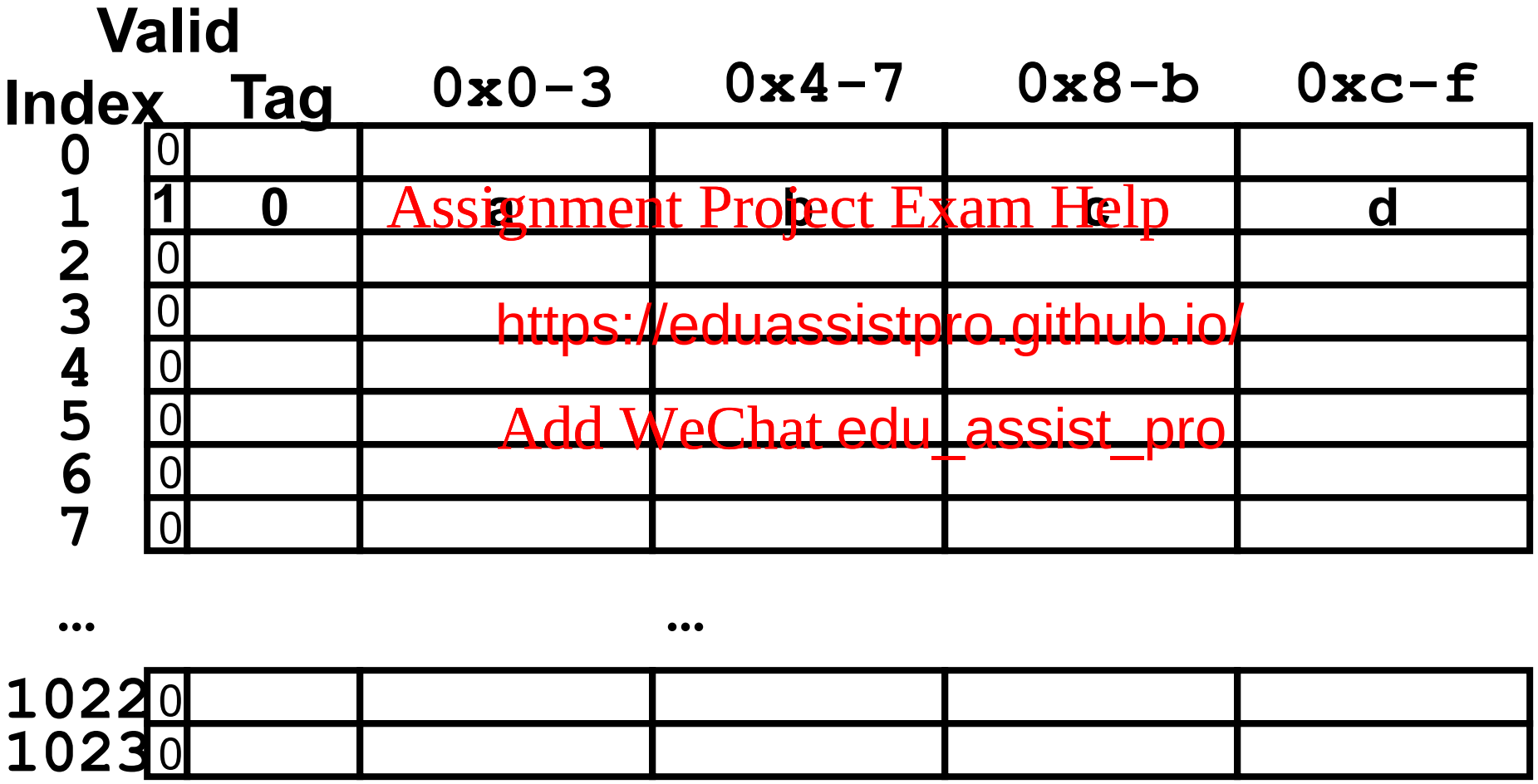
Read from cache at offset, return word **b**



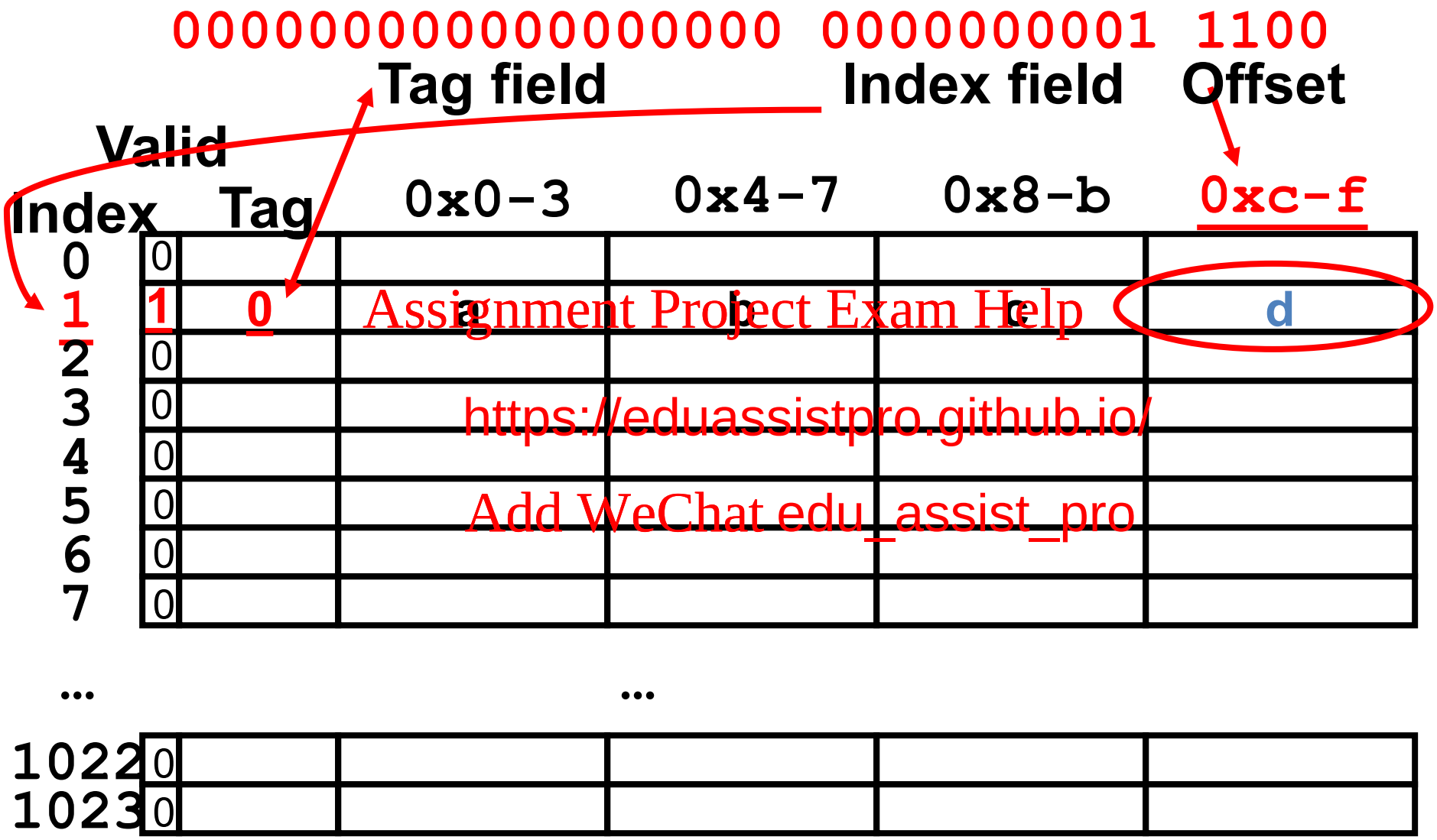
Read 0x0000001C

00000000000000000000 0000000001 1100

Tag field | Index field | Offset



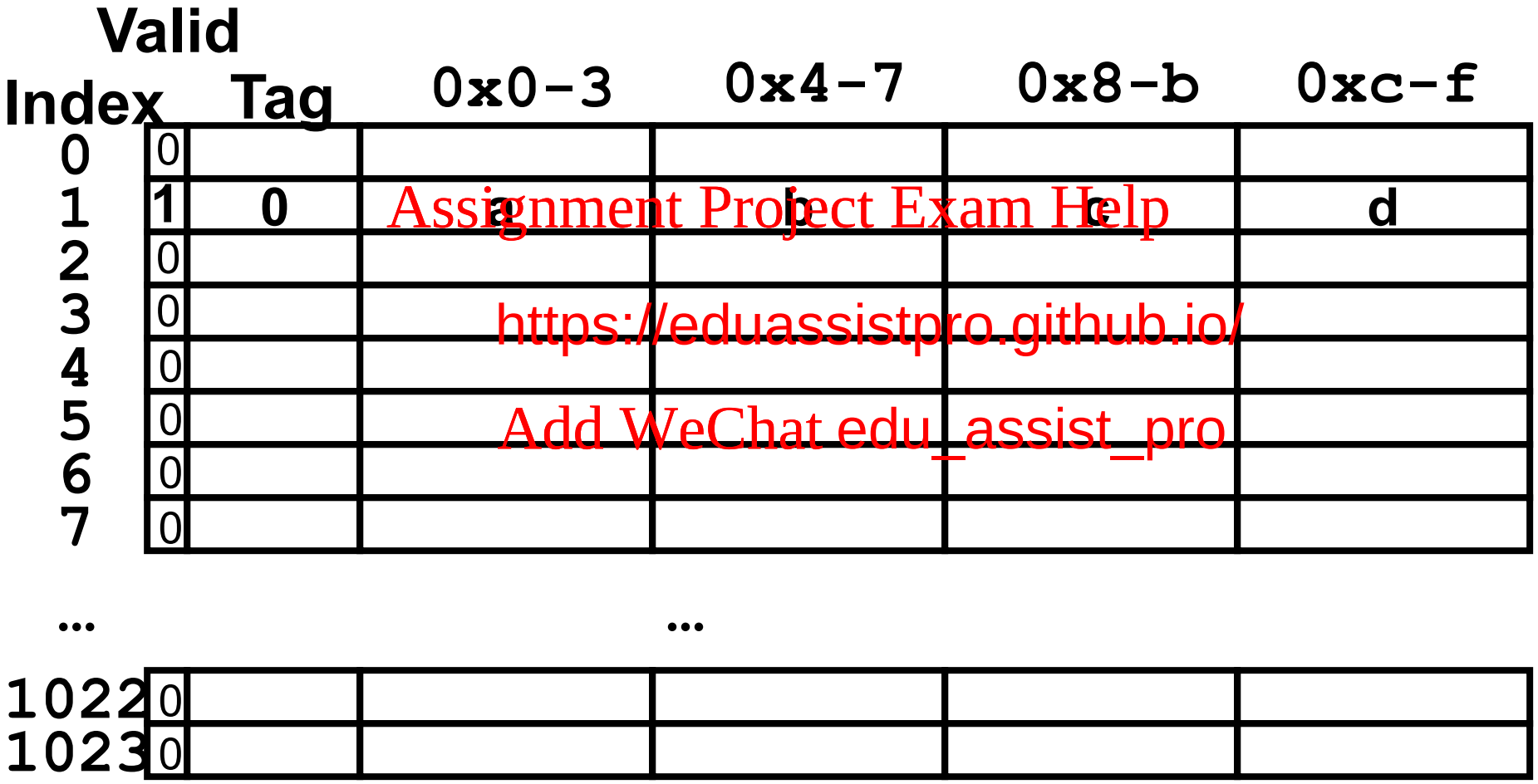
Data valid, tag OK, so read offset return word **d**



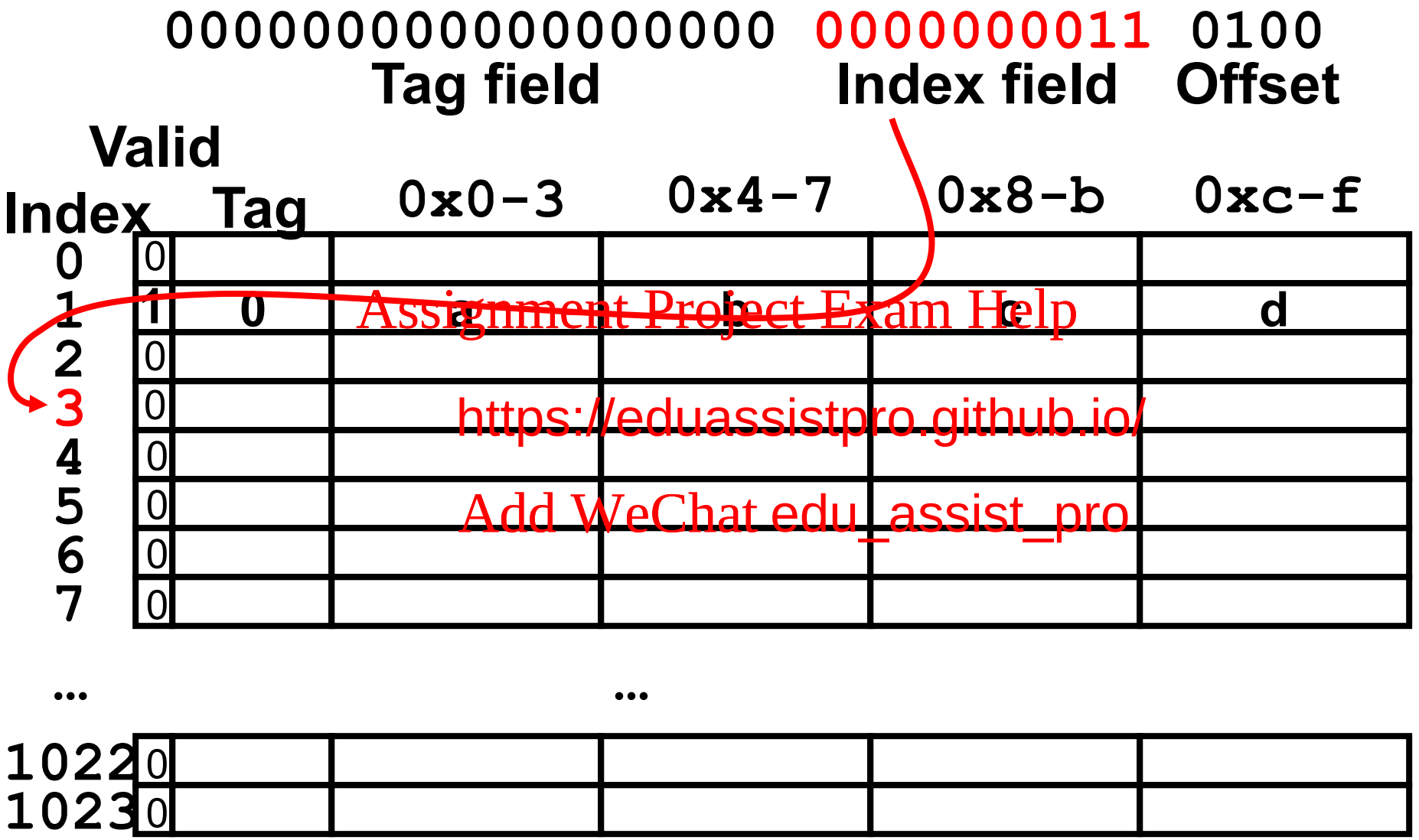
Read 0x00000034

00000000000000000000 0000000011 0100

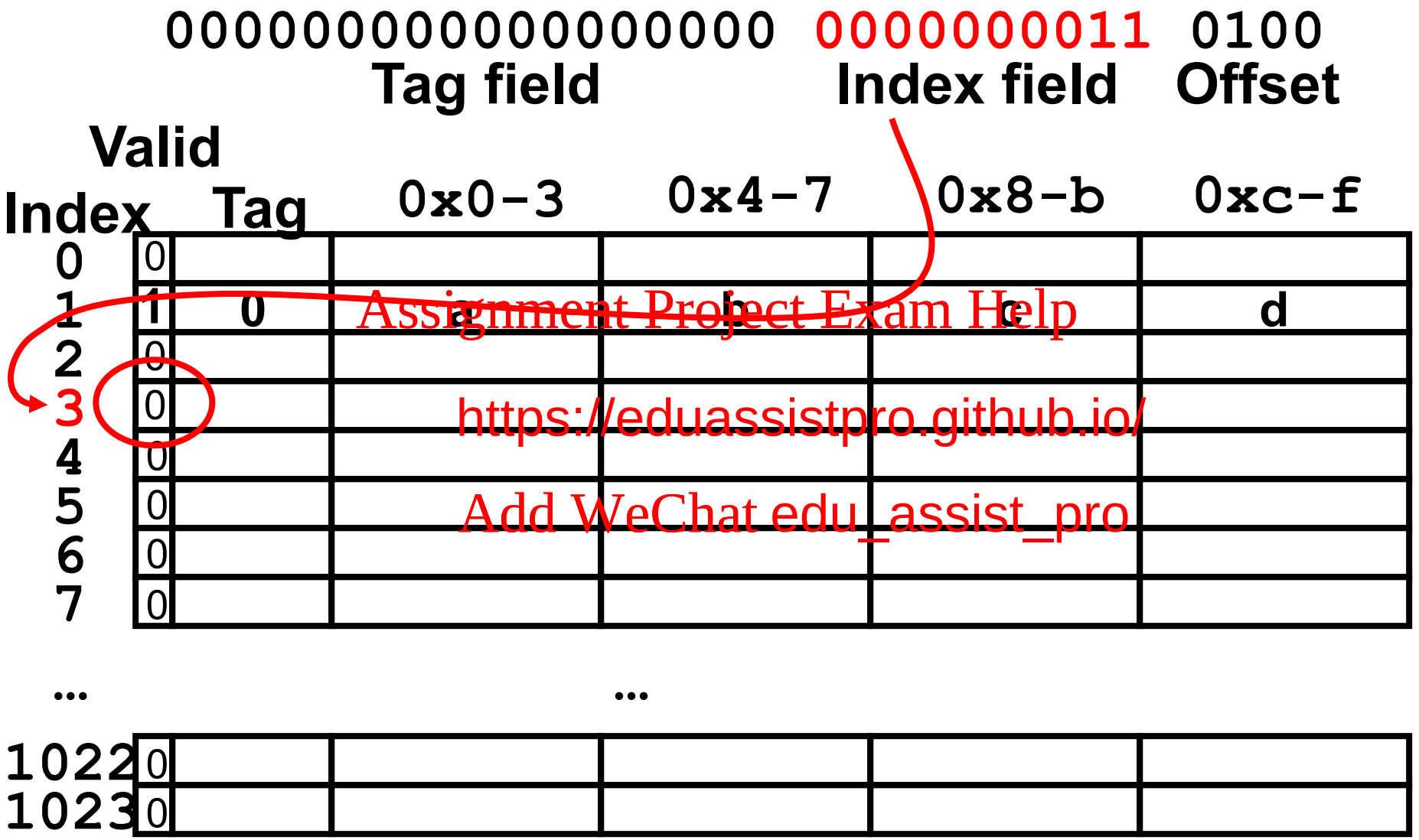
Tag field | Index field | Offset



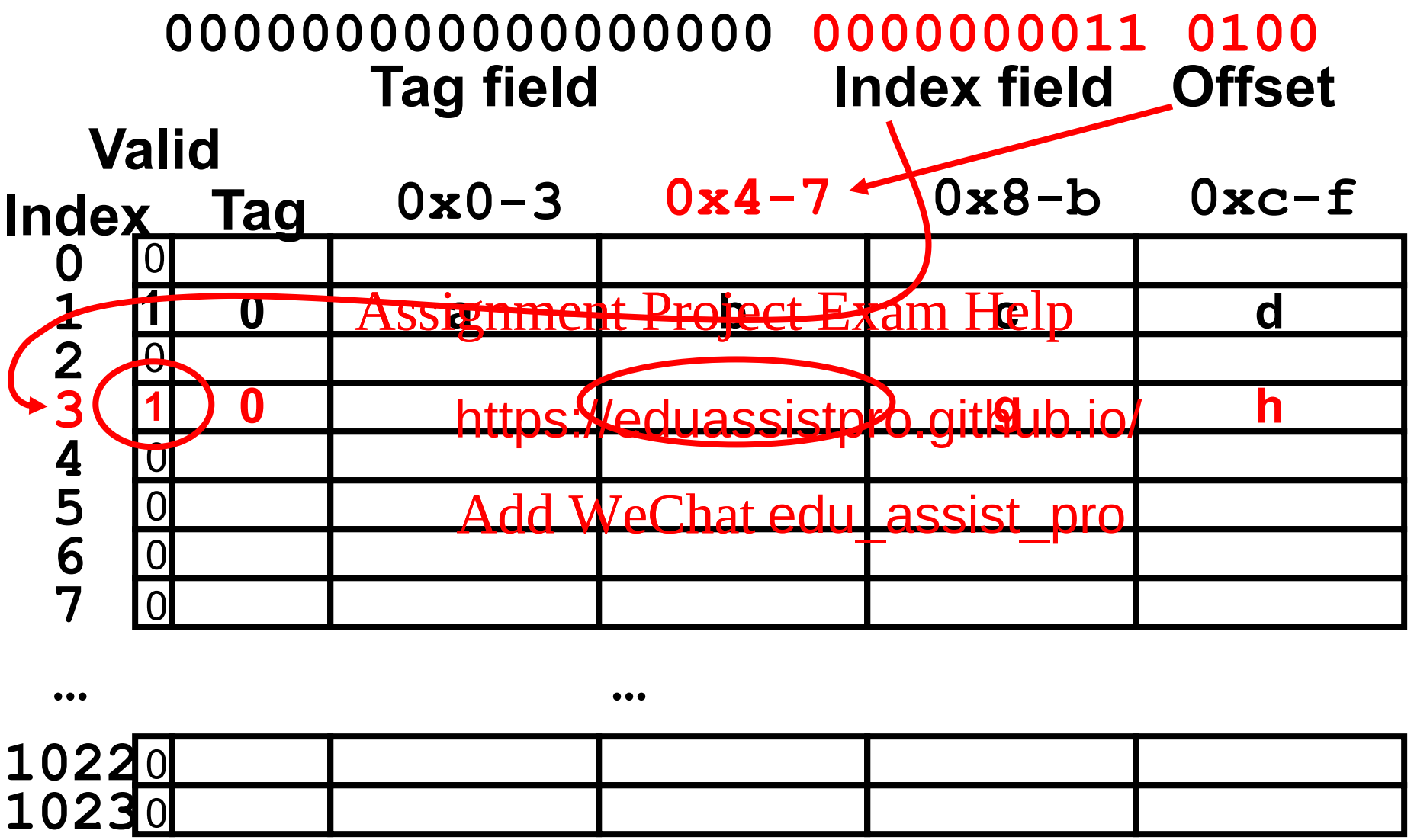
So read block 3



No valid data



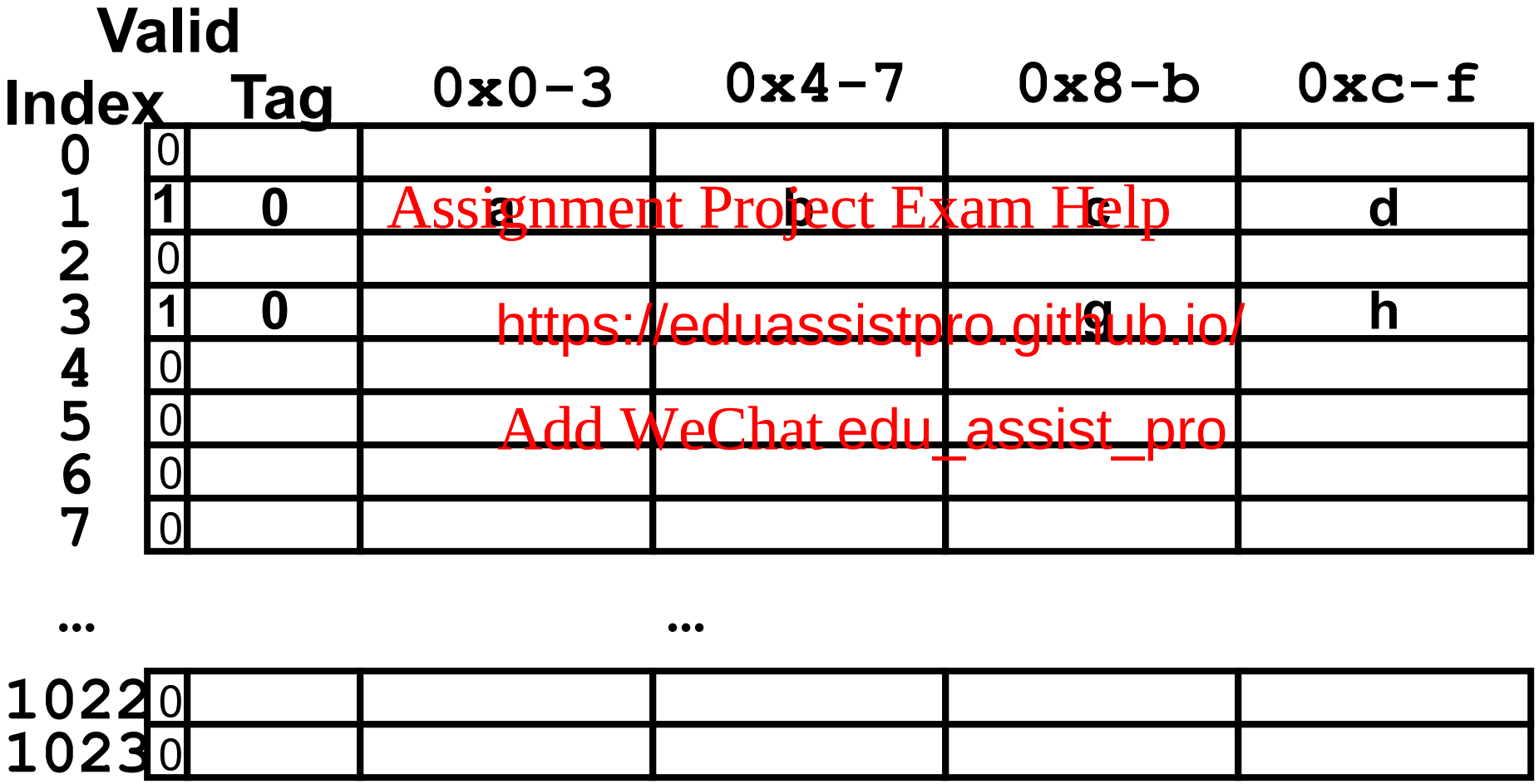
Load that cache block, return word f



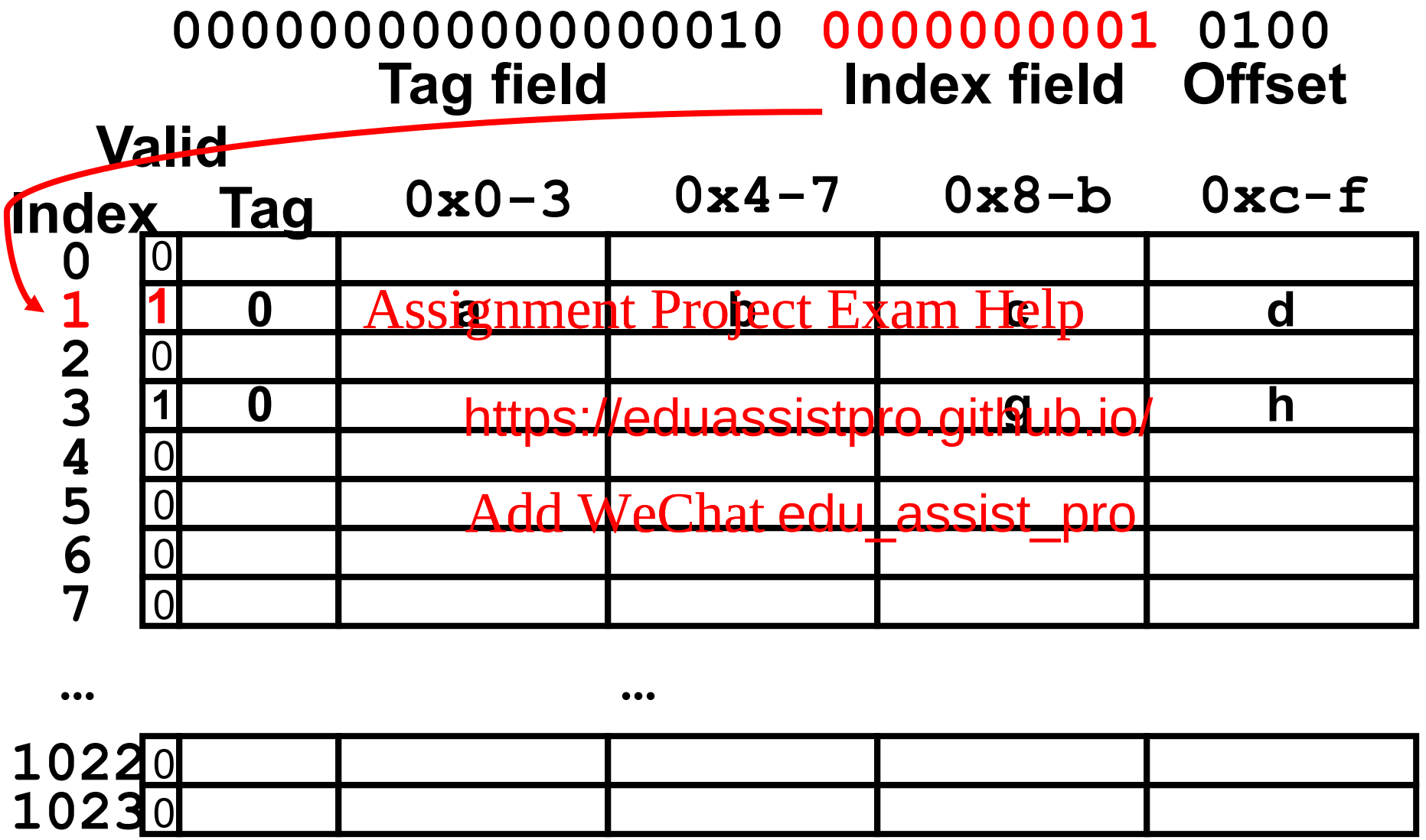
Read 0x00008014

0000000000000010 0000000001 0100

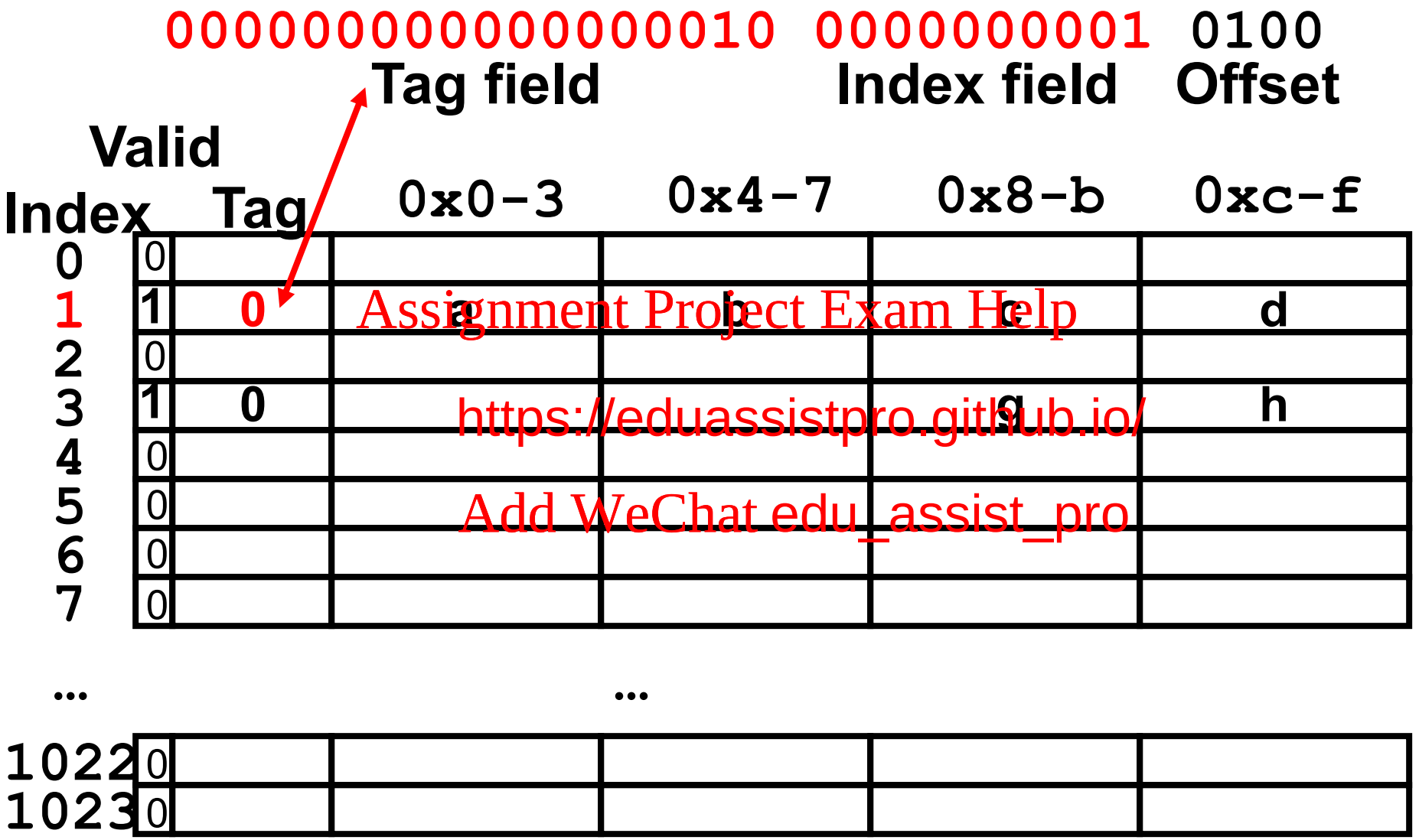
Tag field | Index field | Offset



So we read block 1, Data is Valid



Cache Block 1 Tag does not match (0 != 2)



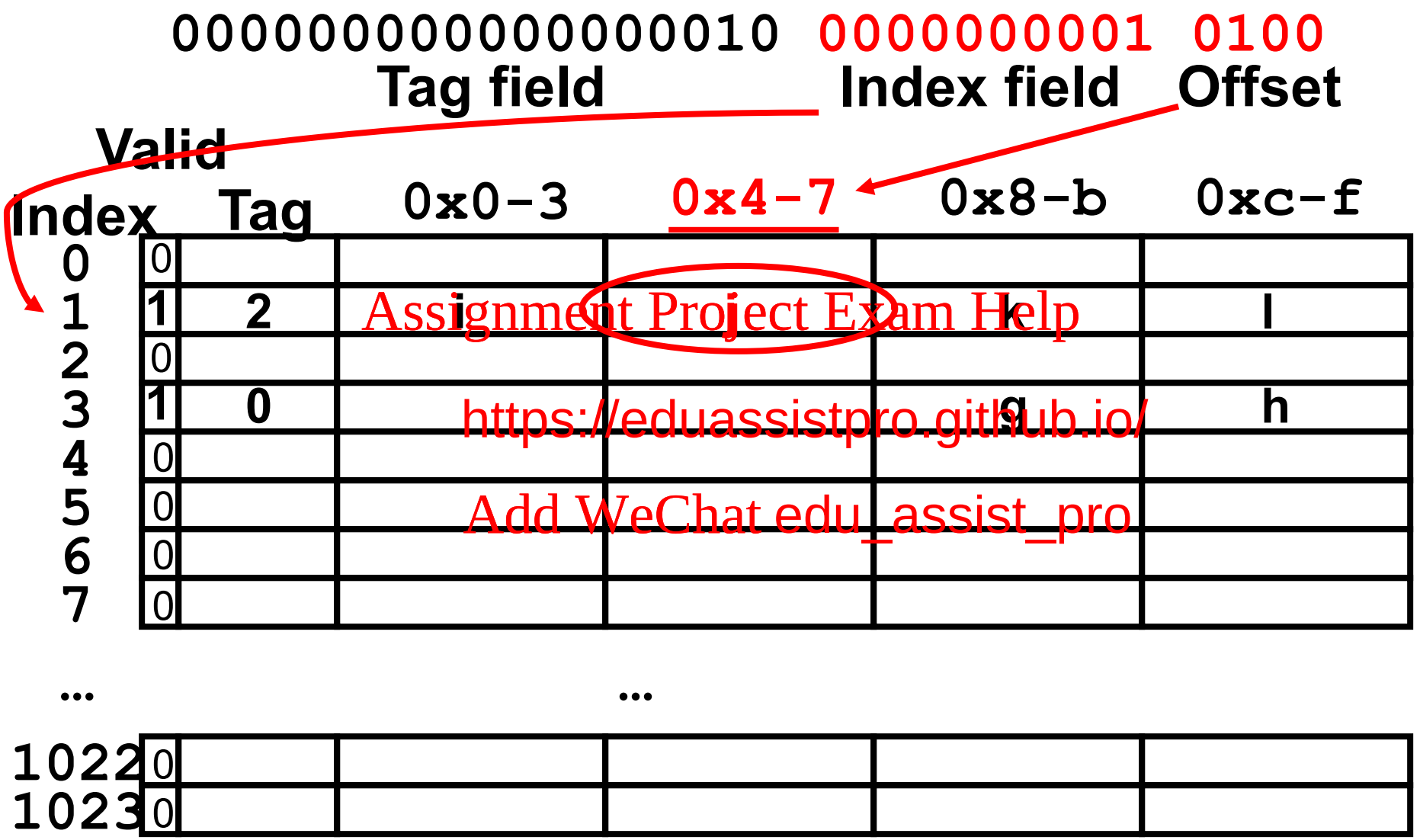
Miss, replace block 1 with new data & tag

0000000000000010 0000000001 0100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	2				
2	0					
3	1	0				
4	0					
5	0					
6	0					
7	0					
…			…			
1022	0					
1023	0					

And return word j



Do an example yourself. What happens?

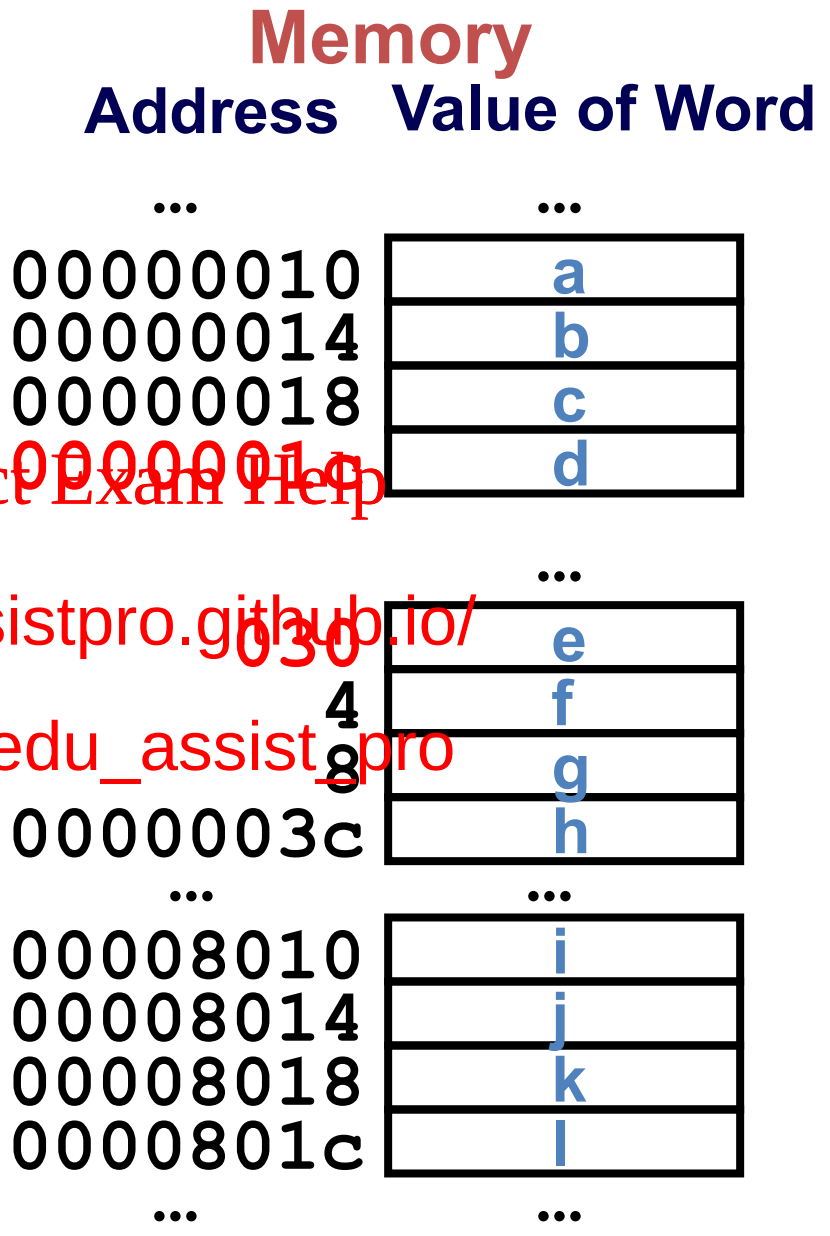
- Cache: Hit, Miss, Miss with replace ?
  Values returned: a ,b, c, d, e, ..., k, l ?
- Read address 0x00000030 ?
  `0000000000000000 0000000011 0000`
- Read address 0x0000001c ?
  `0000000000000000 0000000001 1100`

Cache

Index	Valid	Tag	0		x8-b	0xc-f
0	0					
1	1	2	i	j	k	l
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
…				…		

Answers

- 0x00000030 a hit
  Index = 3, Tag matches,
  Offset = 0, value = e
- 0x0000001c
  Index = 1, Tag
  replace from memory,
  Offset = 0xc, value = d
- Therefore, returned values are:
  - 0x00000030 = e
  - 0x0000001c = d



“And in Conclusion…”

- We would like to have the capacity of disk at the speed of the processor: unfortunately this is not feasible.
- So we create a memory hierarchy:
  - each successively hi ost used” data from next lower level
  - exploits temporal locality and spat
  - do the common case fast, worry less about the exceptions (design principle of MIPS)
- Locality of reference is a Big Idea

Review and More Information

- Sections 5.1 - 5.3 of textbook