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

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`

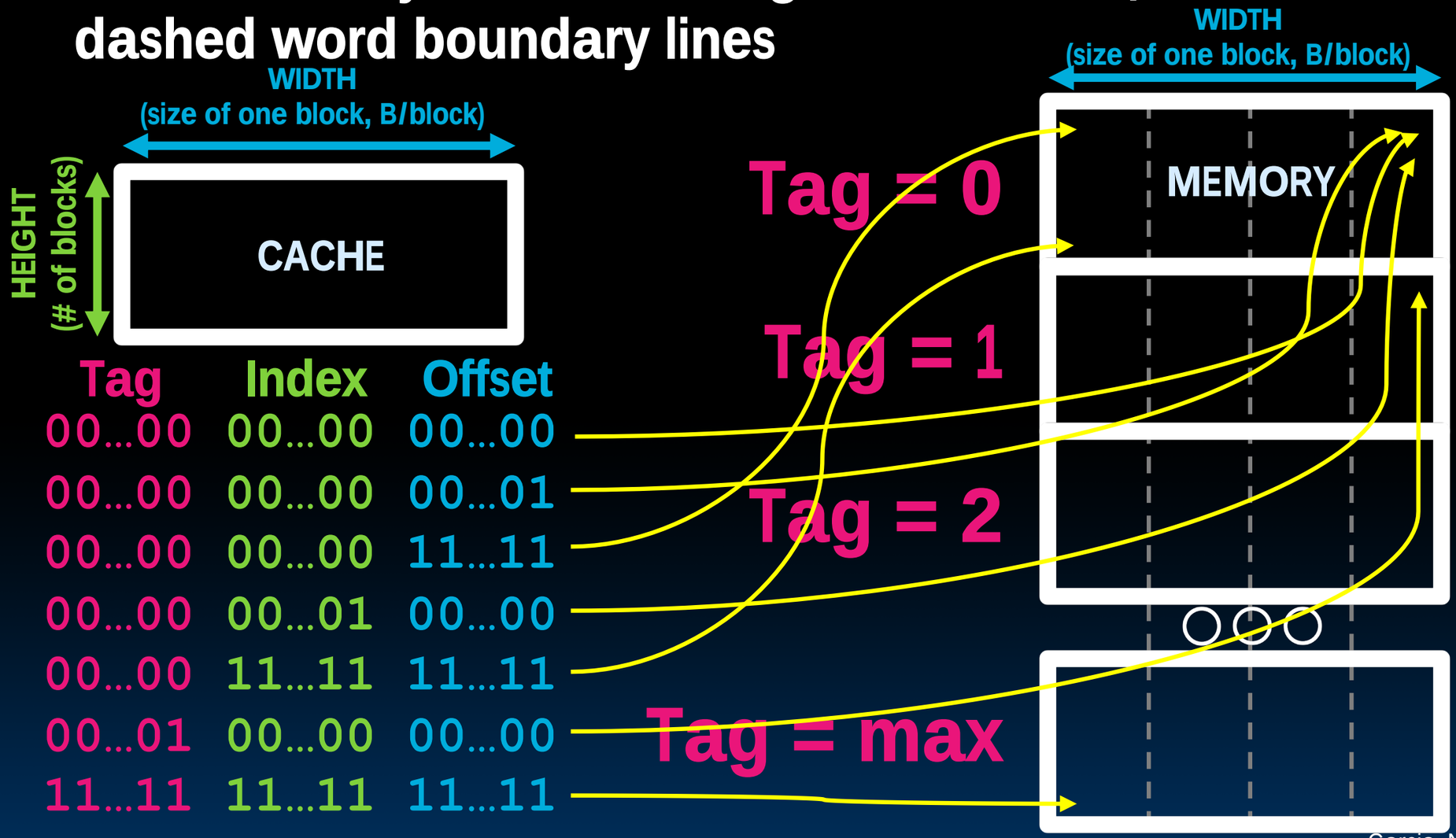
Memory Access with Cache

- **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)

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

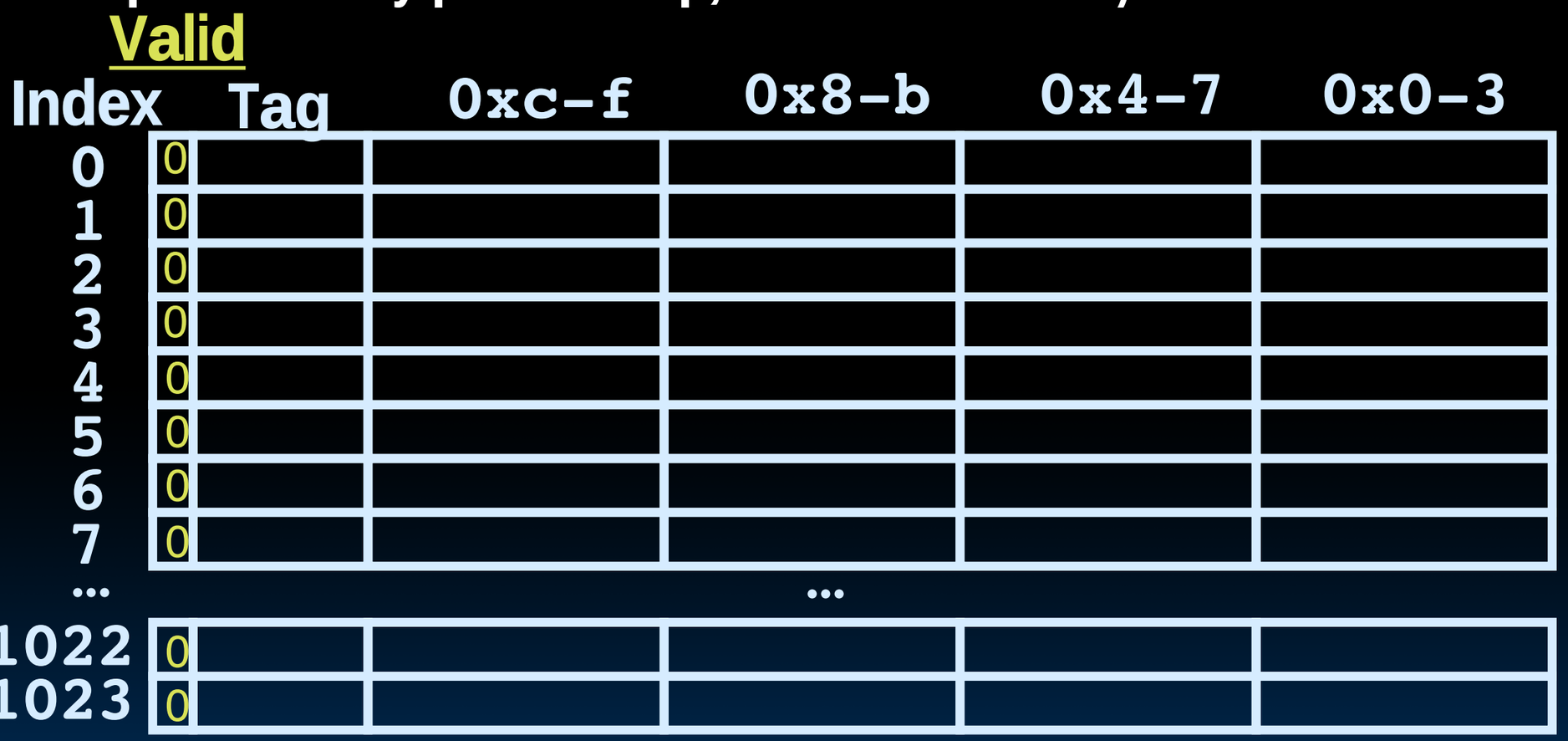
- **When start a new program, cache does not have valid information for this program**
- **Need an indicator whether this tag entry is valid for this program**
- **Add a "valid bit" to the cache tag entry**

  0 → cache miss, even if by chance, address = tag

  1 → cache hit, if processor address = tag

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, will investigate it some more!

"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

UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture

(a.k.a. Machine Structures)

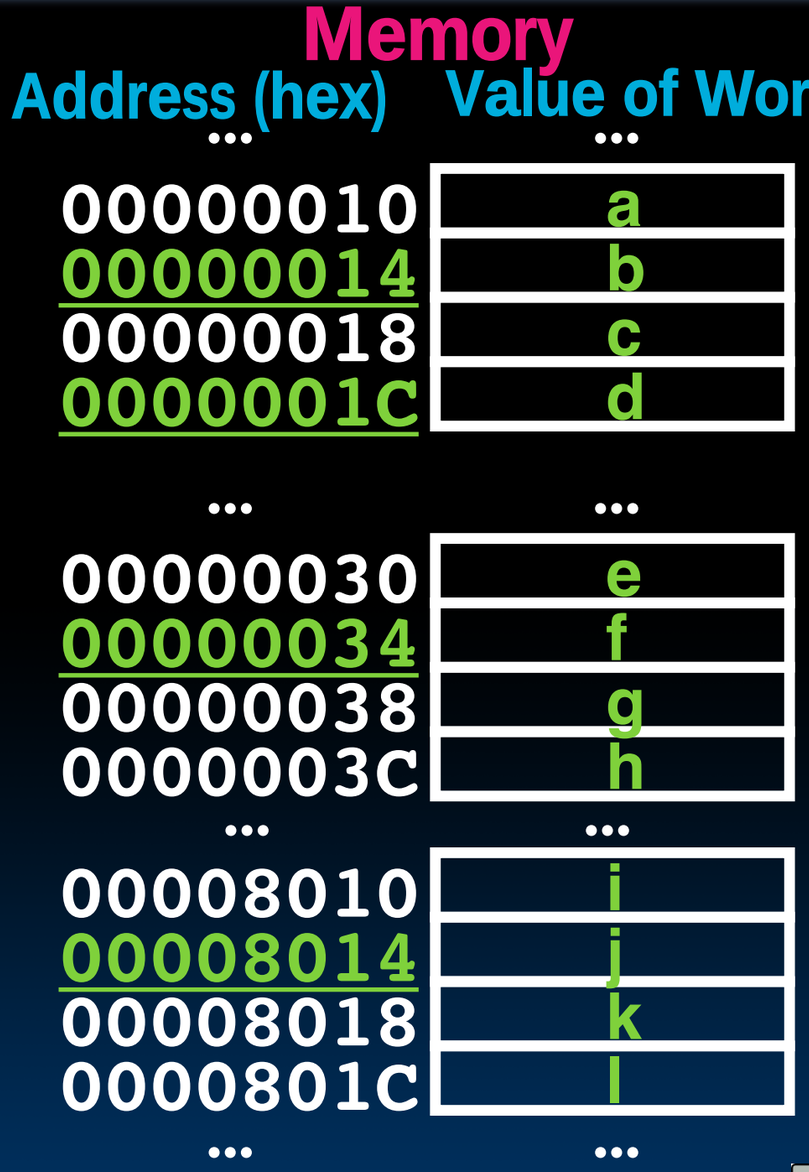
UC Berkeley
Professor
Bora Nikolić

Caches III

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

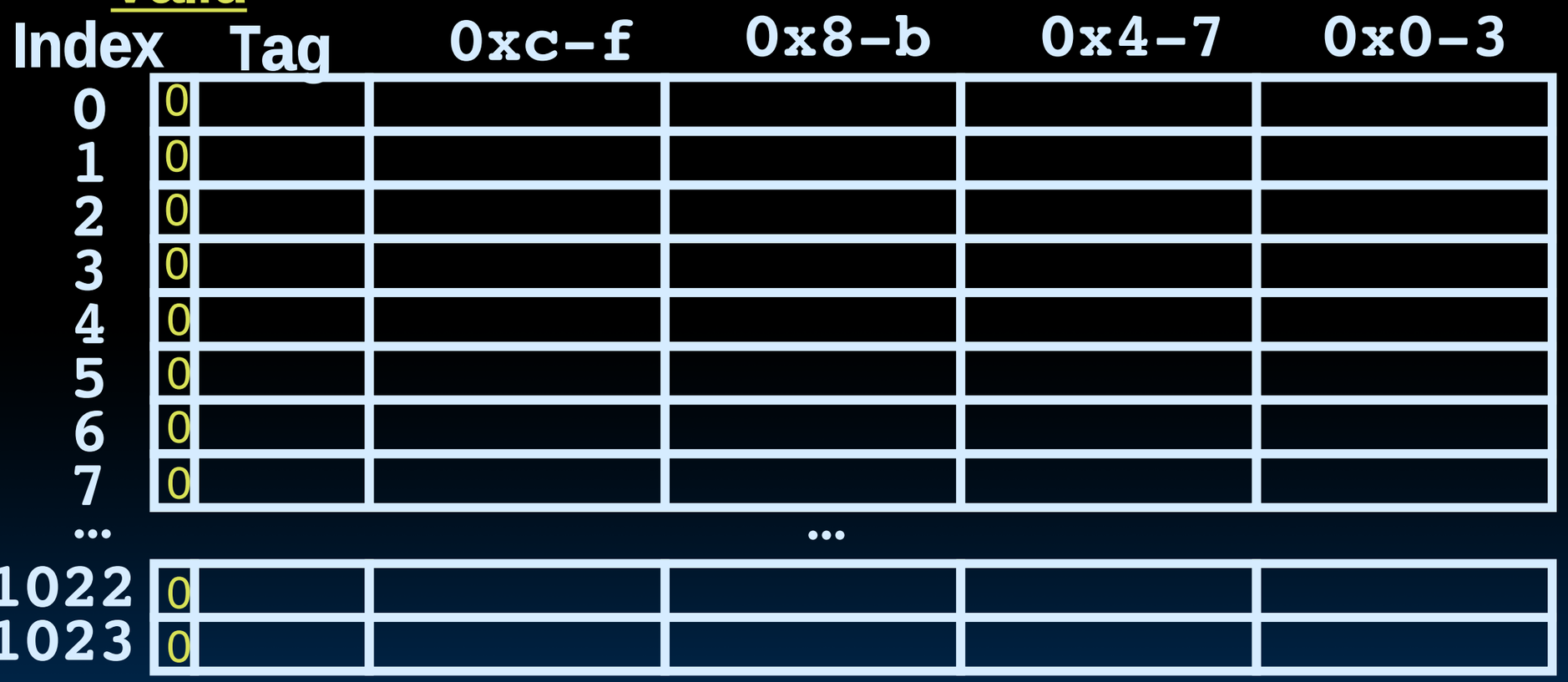
- 4 Addresses:
  - `0x00000014, 0x0000001C, 0x00000034, 0x00008014`
- 4 Addresses divided (for convenience) into Tag, Index, Byte Offset fields

```
00000000000000000000 0000000001 0100
00000000000000000000 0000000001 1100
00000000000000000000 0000000011 0100
00000000000000000010 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)



1. Read 0x00000014

- 00000000000000000000 0000000001 0100

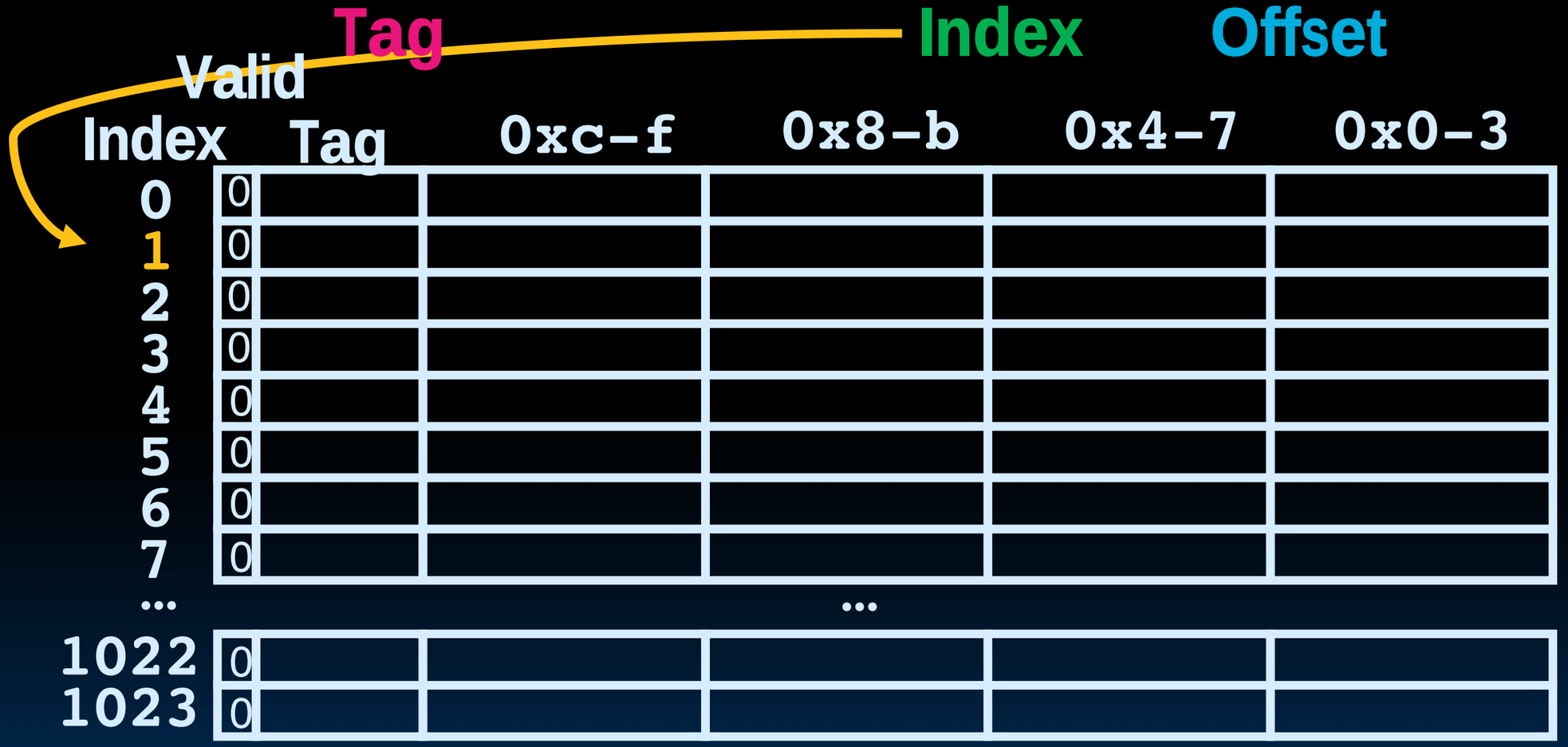
Tag Index Offset

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

So we read block 1 (0000000001)

- 00000000000000000000 0000000001 0100

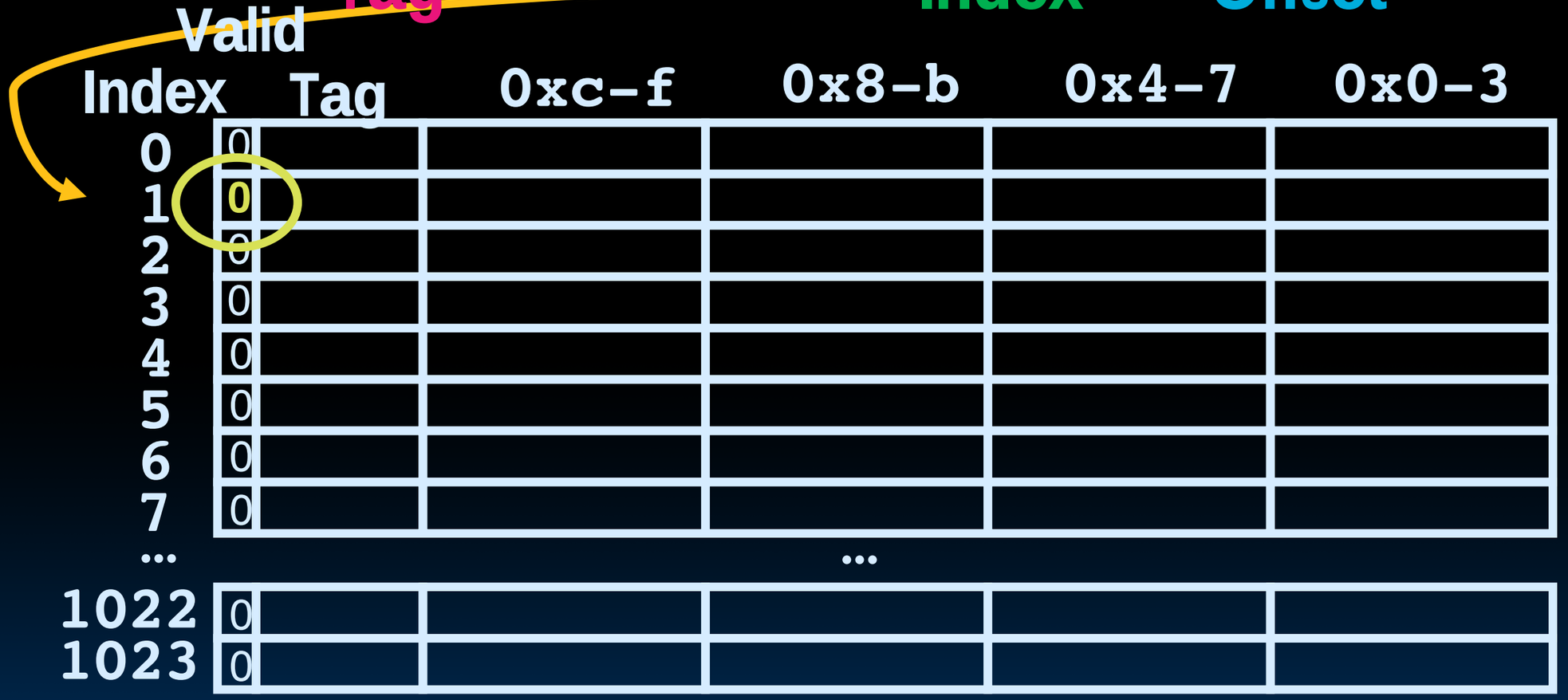
Tag Index Offset



No valid data

- 00000000000000000000 0000000001 0100

Tag Index Offset



So load that data into cache, setting tag, valid

- 00000000000000000000 0000000001 0100

Tag Index Offset

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

Read from cache at offset, return word b

- 00000000000000000000 0000000001 0100

Tag Index Offset

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

2. Read 0x0000001C = 0…00 0..001 1100

- 00000000000000000000 0000000001 1100

Tag Index Offset

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

Index is Valid

- 00000000000000000000 0000000001 1100

Tag Index Offset

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

Index is Valid, Tag Matches

- 00000000000000000000 0000000001 1100

Tag Index Offset

Index is Valid, Tag Matches, return d

- 00000000000000000000 0000000001 1100

Tag Index Offset

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

3. Read 0x00000034 = 0…00 0..011 0100

- 00000000000000000000 0000000011 0100

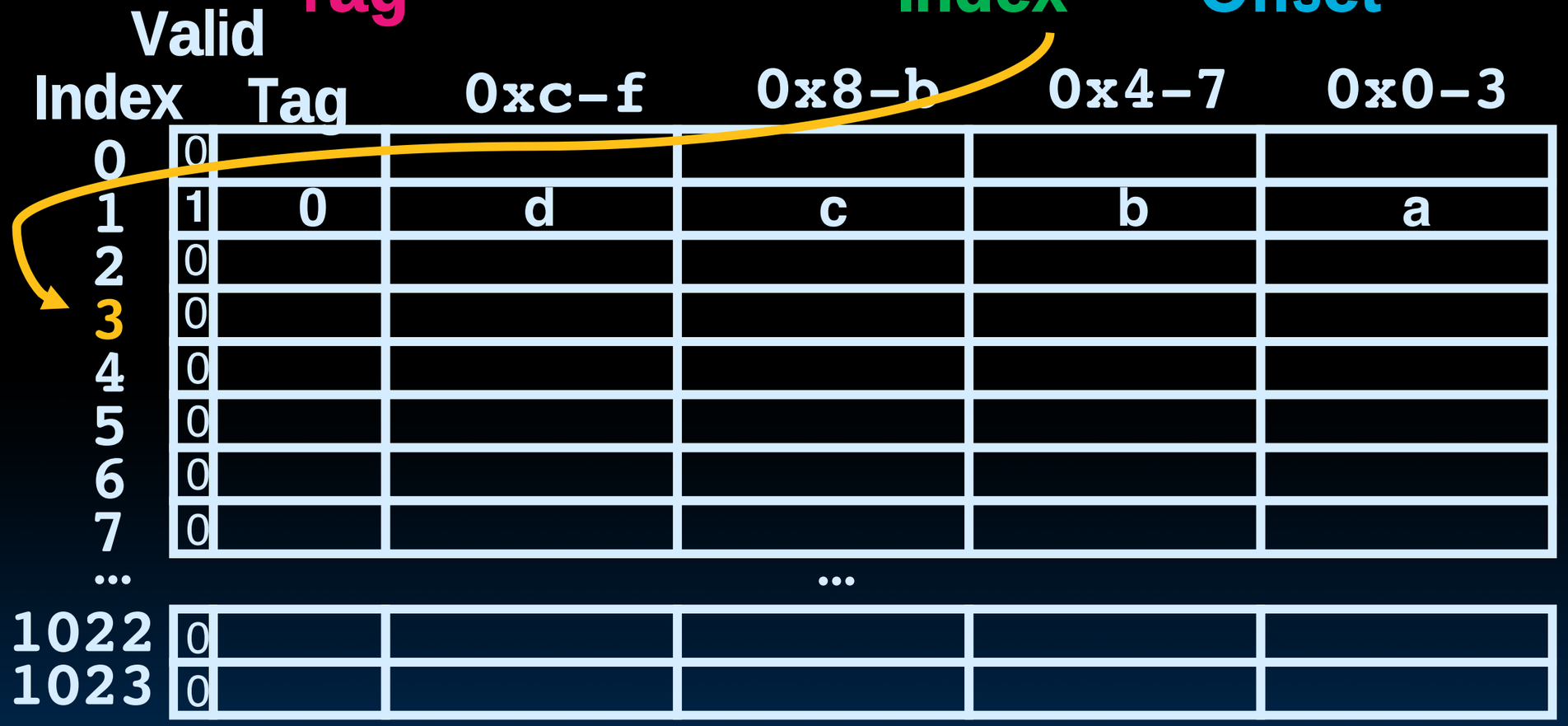
Tag Index Offset

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

So read block 3

- 00000000000000000000 0000000011 0100

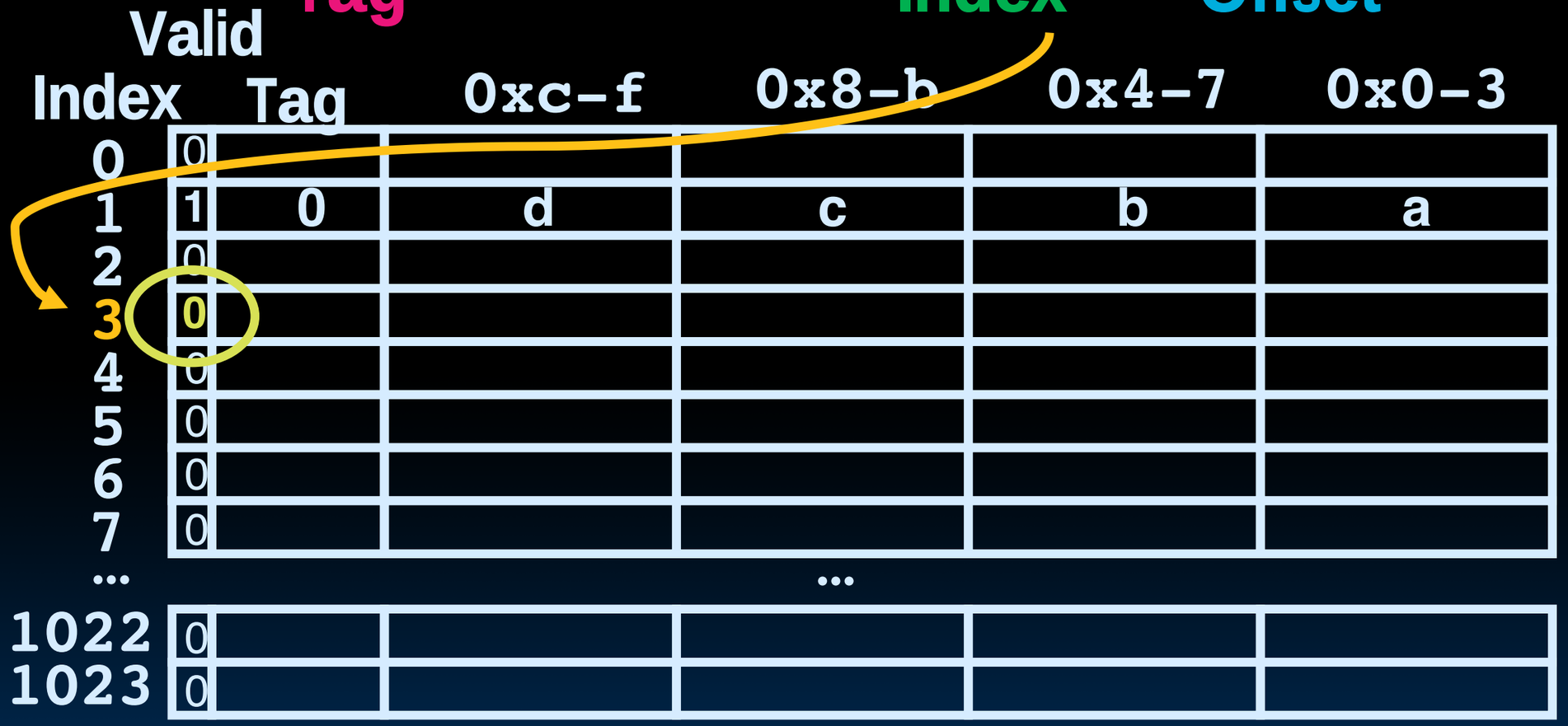
Tag Index Offset



No valid data

- 00000000000000000000 0000000011 0100

Tag Index Offset



Load that cache block, return word f

- 00000000000000000000 0000000011 0100

Tag Index Offset

Index	Valid	Tag	0xc–f	0x8–b	0x4–7	0x0–3
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

4. Read 0x00008014 = 0…10 0..001 0100

- 00000000000000010 0000000001 0100

Tag Index Offset

Index	Valid	Tag	0xc–f	0x8–b	0x4–7	0x0–3
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

So read Cache Block 1, Data is Valid

- 00000000000000010 0000000001 0100

Tag Index Offset

Index	Valid	Tag	0xc–f	0x8–b	0x4–7	0x0–3
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

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

- 0000000000000010 0000000001 0100

Tag Index Offset

Index	Valid	Tag	0xc–f	0x8–b	0x4–7	0x0–3
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

Miss, so replace block 1 with new data & tag

- 00000000000000010 0000000001 0100

Tag Index Offset

Index	Valid	Tag	0xc–f	0x8–b	0x4–7	0x0–3
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					
…				…		
1022	0					
1023	0					

And return word J

- 0000000000000010 0000000001 0100

Tag Index Offset

Index	Valid	Tag	0xc–f	0x8–b	0x4–7	0x0–3
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					
…				…		
1022	0					
1023	0					

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** ?
  `00000000000000000 0000000011 0000`
- Read address **0x0000001c** ?
  `00000000000000000 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

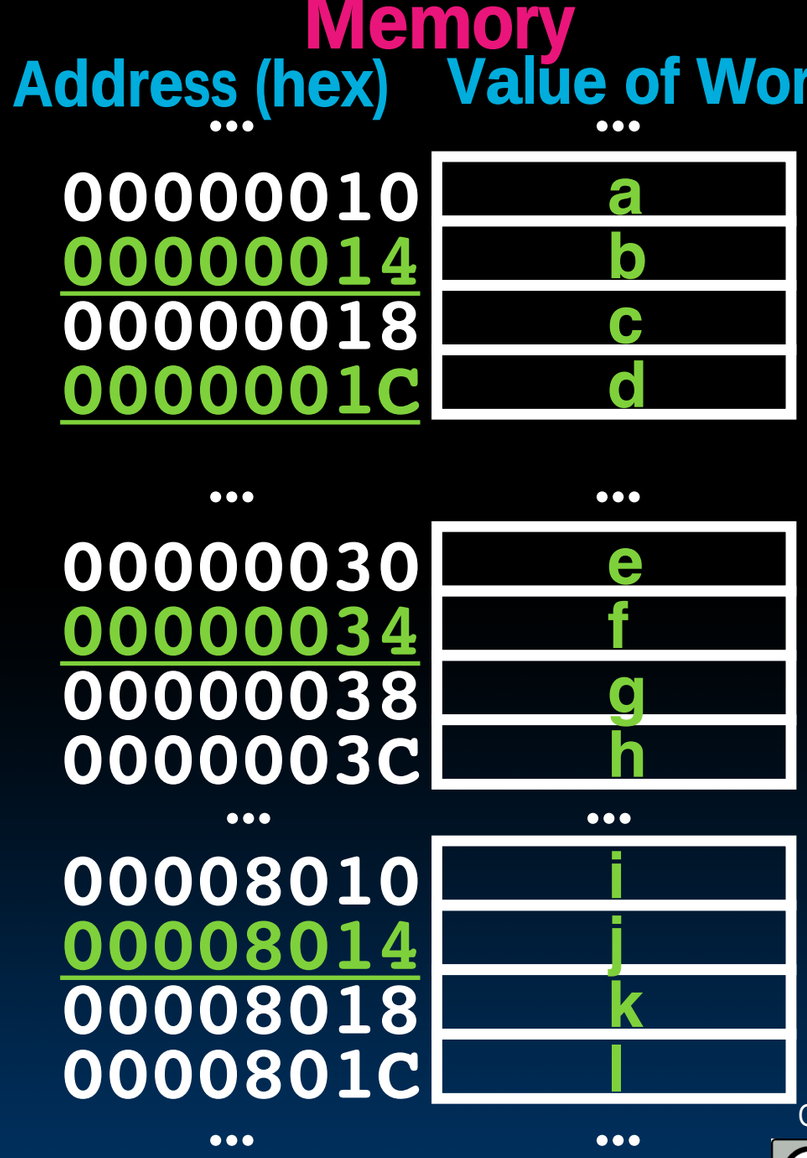
- **0x00000030 a hit**

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

- **0x0000001c a miss**

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

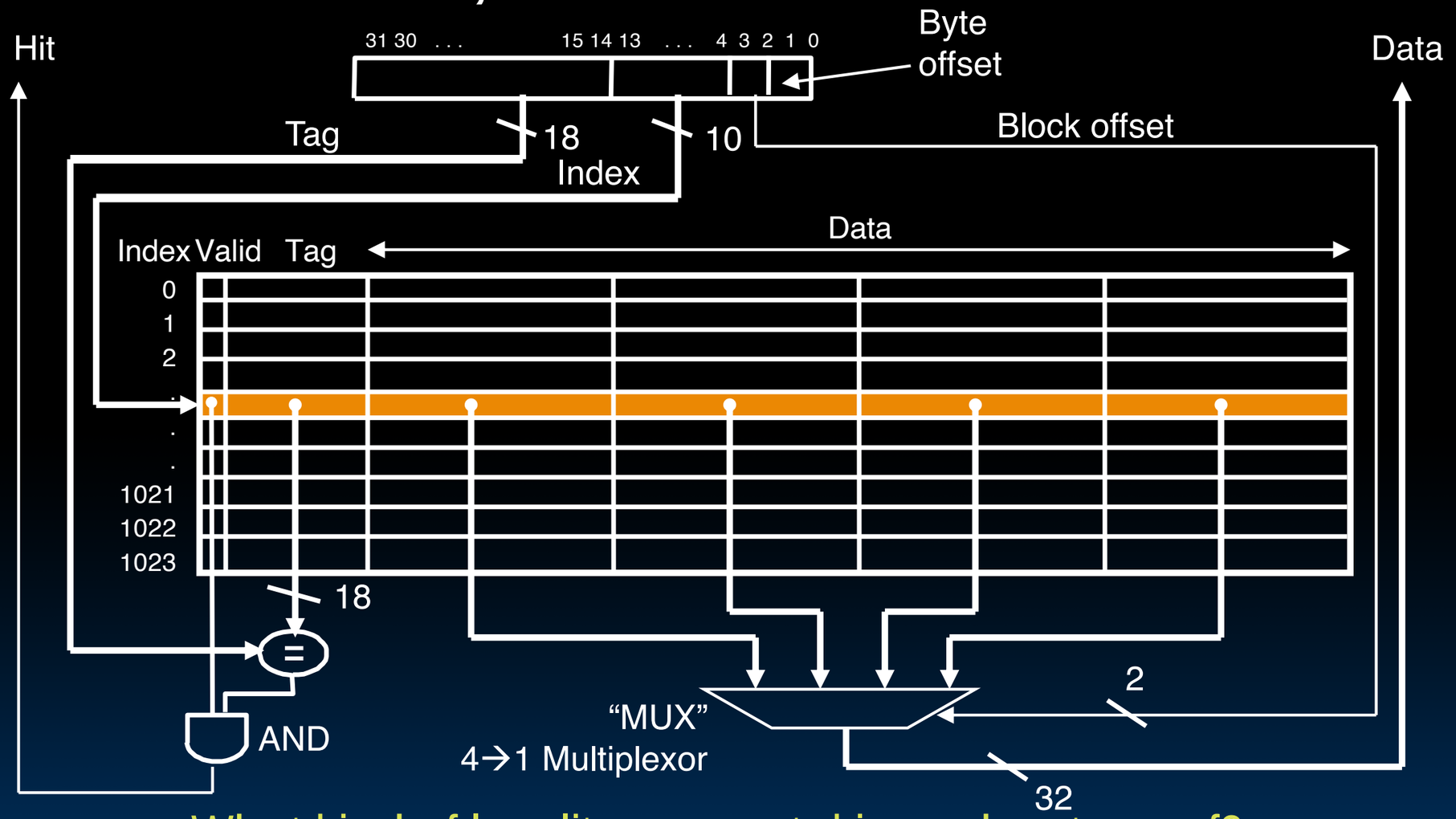
- **Since reads, values must = memory values whether or not cached:**
  - 0x00000030 = e
  - 0x0000001c = d



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?**

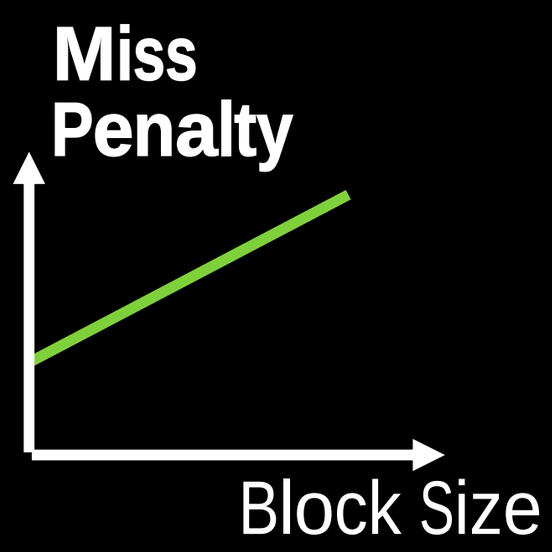
Block Size Tradeoff

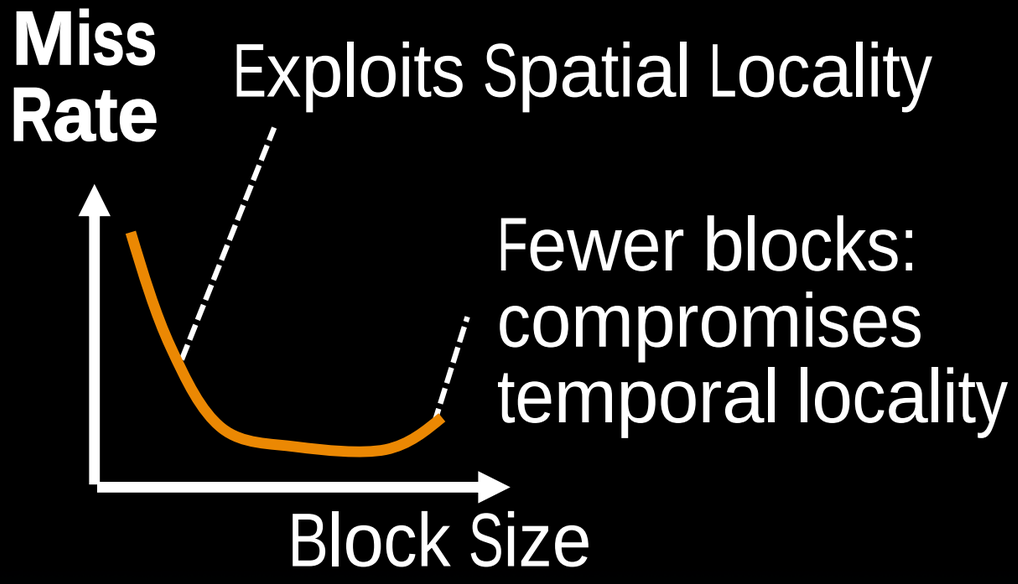
- **Benefits of Larger Block Size**
  - Spatial Locality: if we access a given word, we're likely to access other nearby words soon
  - Very applicable with Stored-Program Concept
  - Works well for sequential array accesses
- **Drawbacks of Larger Block Size**
  - Larger block size means larger miss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

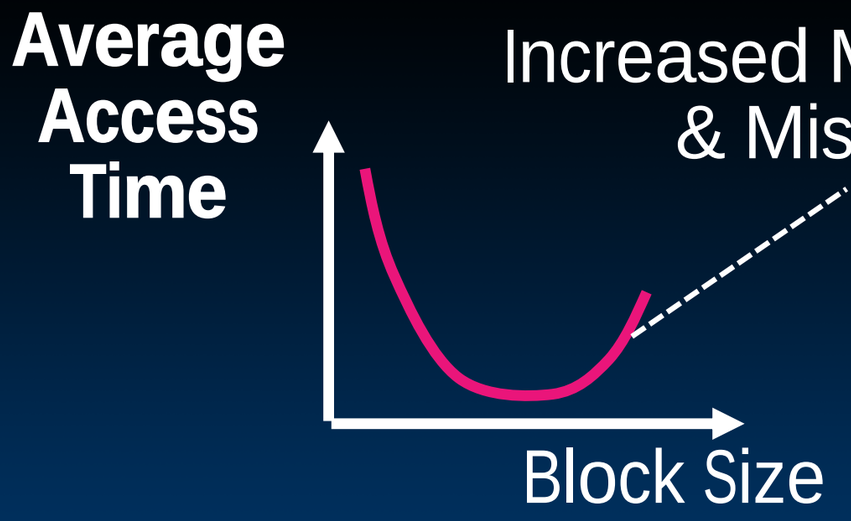
Extreme Example: One Big Block

- **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?