inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

Lecture 22
Caches I

2010-07-28

Instructor Paul Pearce

IT'S NOW LEGAL TO JAILBREAK YOUR PHONE!

On Monday the Library of Congress added 5 exceptions to the DMCA that, among other things, verify the legality of jail-breaking devices such as phones (details @ the URL). This means you can actually hack around with devices you already down. Ground breaking, isnt it?

http://preview.tinyurl.com/29sgmra

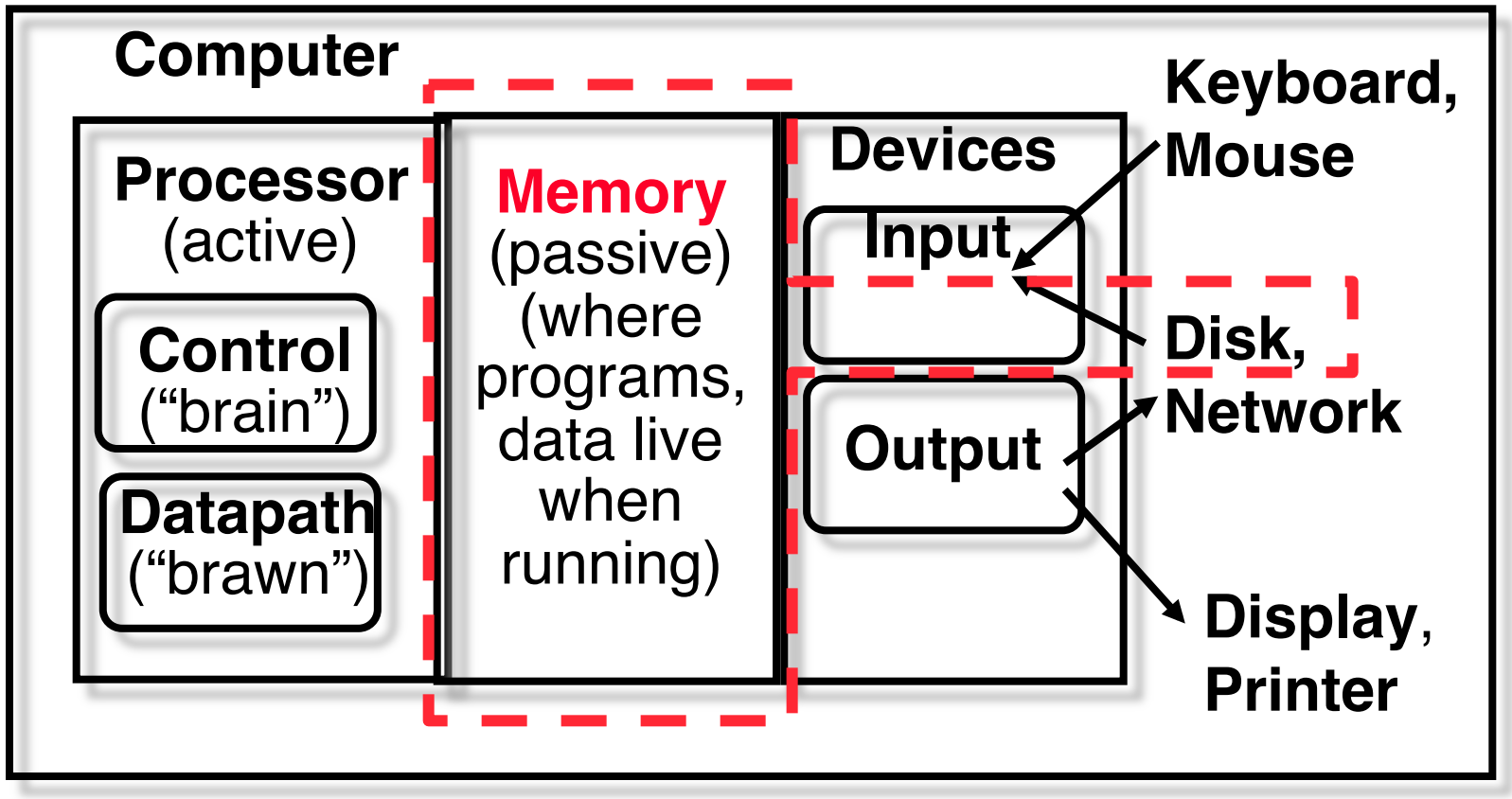
Pearce, Summer 2010 © UCB

Review : Pipelining

- Pipeline challenge is hazards
  - Forwarding helps w/many data hazards
  - Delayed branch helps with control hazard in our 5 stage pipeline
  - Data hazards w/Loads → Load Delay Slot
    - Interlock →"smart" CPU has HW to detect if conflict with inst following load, if so it stalls
- More aggressive performance (discussed in section a bit today)
  - Superscalar (parallelism)
  - Out-of-order execution

Pearce, Summer 2010 © UCB

The Big Picture



Pearce, Summer 2010 © UCB

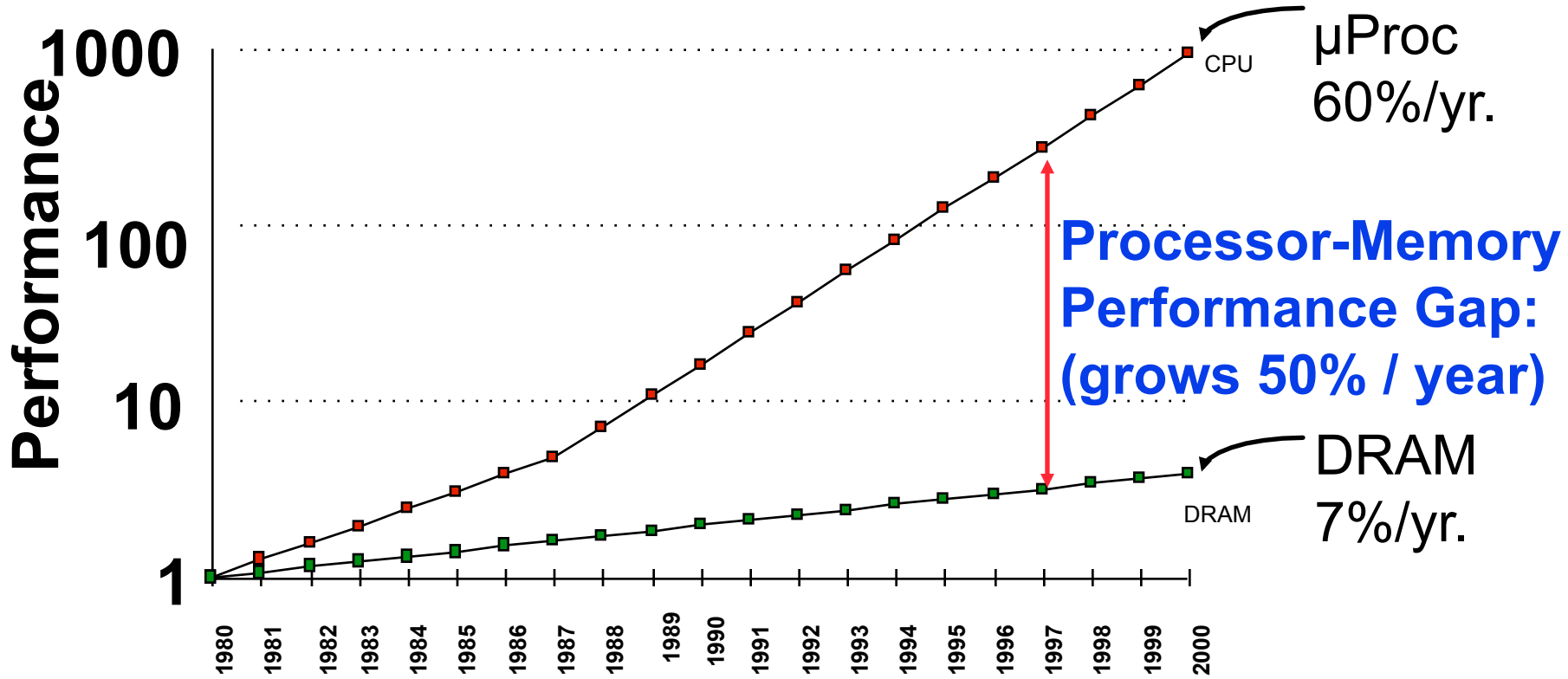
Memory Hierarchy *I.e., storage in computer systems*

- Processor
  - holds data in register file (~100 Bytes)
  - Registers accessed on nanosecond timescale
- Memory (we'll call "main memory")
  - More capacity than registers (~Gbytes)
  - Access time ~50-100 ns
  - Hundreds of clock cycles per memory access?!
- Disk
  - HUGE capacity (virtually limitless)
  - VERY slow: runs ~milliseconds

Pearce, Summer 2010 © UCB

Motivation: Why We Use Caches (written $)

- 1989 first Intel CPU with cache on chip
- 1998 Pentium III has two cache levels on chip



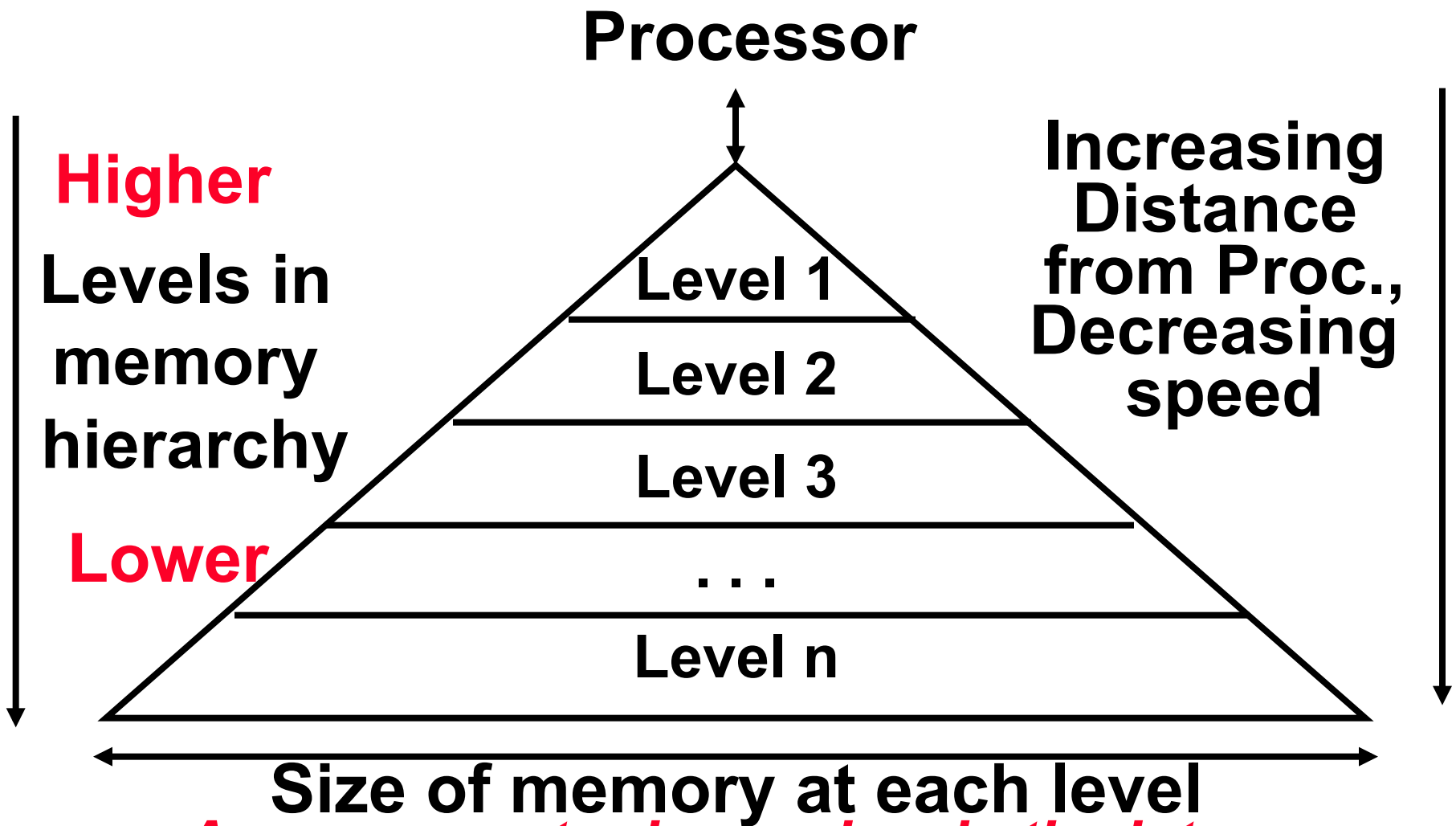
Pearce, Summer 2010 © UCB

Memory Caching

- Mismatch between processor and memory speeds leads us to add a new level: a memory cache
- Implemented with same IC processing technology as the CPU (usually integrated on same chip): faster but more expensive than DRAM memory.
- Cache is a copy of a subset of main memory.
- Most processors have separate caches for instructions and data. This is how we solved the single-memory structural hazard yesterday.

Pearce, Summer 2010 © UCB

Memory Hierarchy



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

Pearce, Summer 2010 © UCB

Memory Hierarchy

- If level closer to Processor, it is:
  - Smaller
  - Faster
  - More expensive
  - subset of lower levels (contains most recently used data)
- Lowest Level (usually disk) contains all available data (does it go beyond the disk? Is it networked? The cloud?)
- Memory Hierarchy presents the processor with the illusion of a very large & fast memory

Pearce, Summer 2010 © UCB

Memory Hierarchy Analogy: Library (1/2)

- You're writing a term paper (Processor) at a table in your dorm
- Doe Library is equivalent to disk
  - essentially limitless capacity
  - very slow to retrieve a book
- Dorm room is main memory
  - smaller capacity: means you must return book when dorm room fills up
  - easier and faster to find a book there once you've already retrieved it

Pearce, Summer 2010 © UCB

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, put it away (in your dorm)
  - much, much faster to retrieve data
- Illusion created: whole library open on the tabletop
  - Keep as many recently used books open on table as possible since likely to use again
  - Also keep as many books in your dorm as possible, since faster than going to library
- In reality, disk is SO slow, its more like having to drive to the Stanford Library
  - And who wants to do that?

Pearce, Summer 2010 © UCB

Memory Hierarchy Basis

- Cache contains copies of data in memory that are being used.
- Memory contains copies of data on disk that are being used.
- Caches work on the principles of temporal and spatial locality.
  - Temporal Locality: if we use it now, chances are we'll want to use it again soon.
  - Spatial Locality: if we use a piece of memory, chances are we'll use the neighboring pieces soon.

Pearce, Summer 2010 © UCB

Cache Design

- How do we organize cache?
- Where does each memory address map to?
  - (Remember that cache is subset of memory, so multiple memory addresses map to the same cache location.)
- How do we know which elements are in cache?
- How do we quickly locate them?

Pearce, Summer 2010 © UCB

Administrivia

- Homework 8 due tonight at midnight
- Project 2: Find a partner, get going. Due Monday.
- Project 1 and HW4 grades up now.
  - We had to regrade the project a few times to make sure the distribution was fair, and it took longer than expected. Sorry.
  - HW5 and 6 are in the pipeline. Should be done soon.
- Reminder: The drop deadline is this FRIDAY. I believe you have to drop in person, so don't wait.

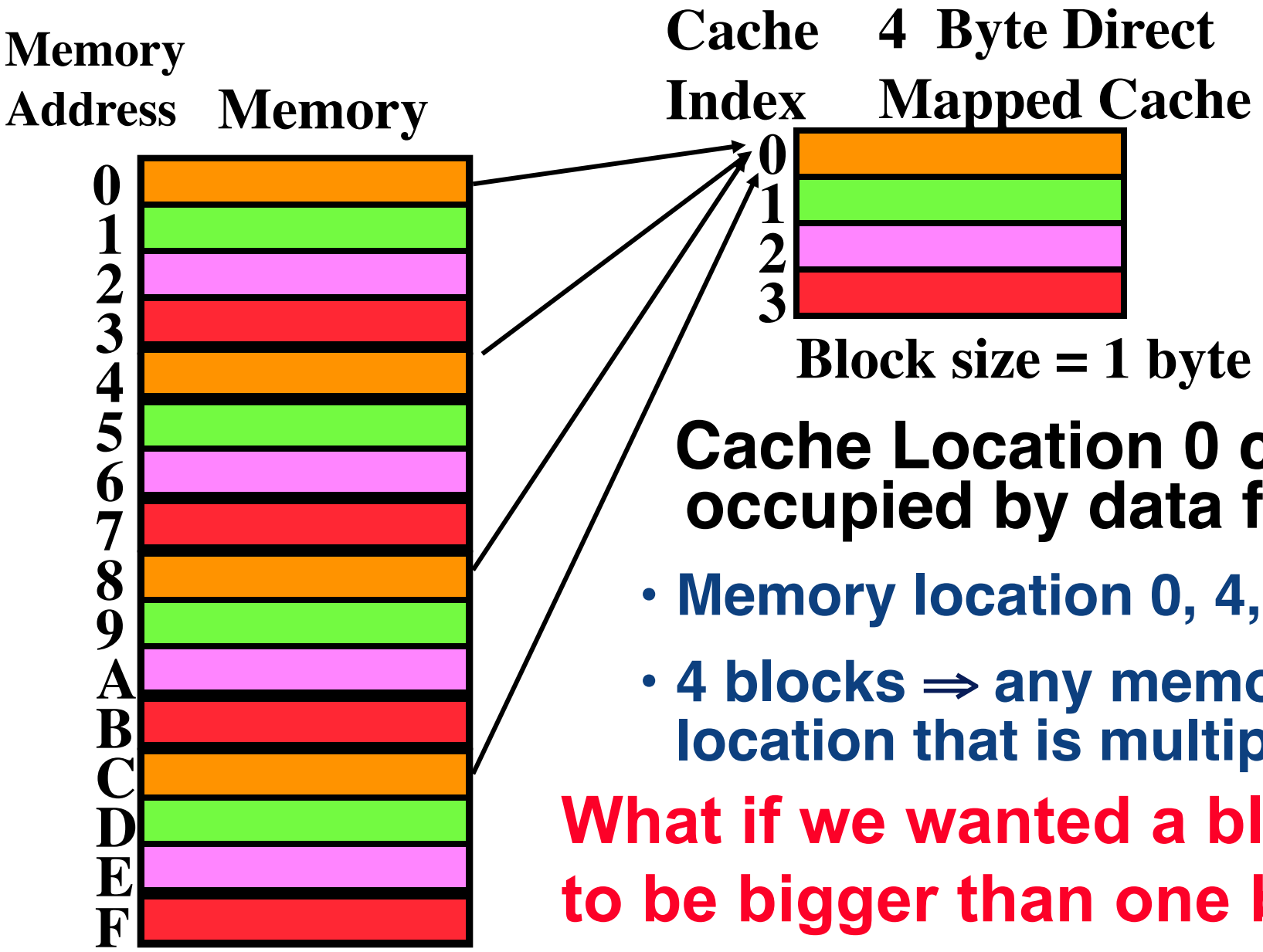
Pearce, Summer 2010 © UCB

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

Pearce, Summer 2010 © UCB

Direct-Mapped Cache (2/4)



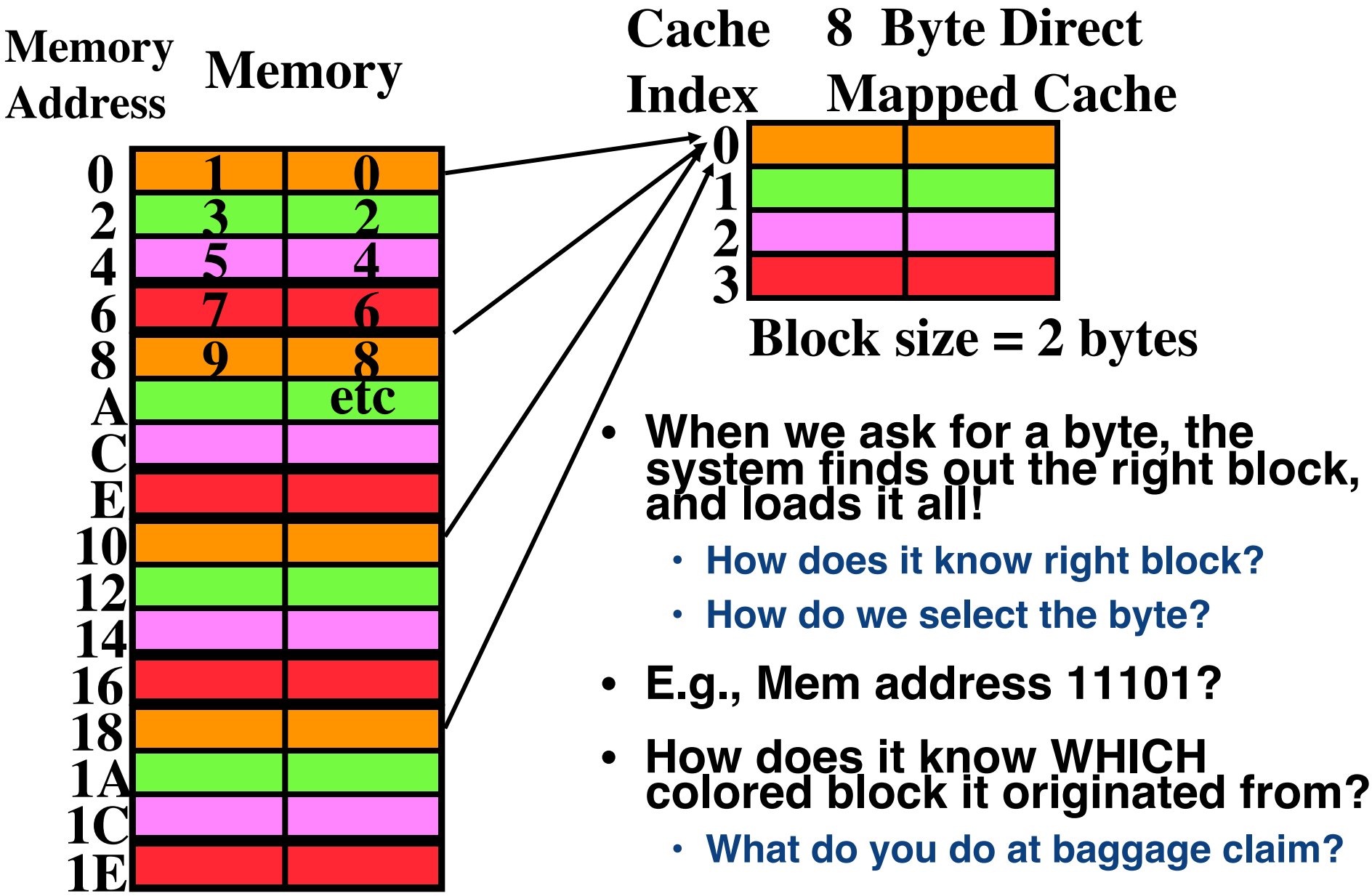
Cache Location 0 can be occupied by data from:

- 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?

Pearce, Summer 2010 © UCB

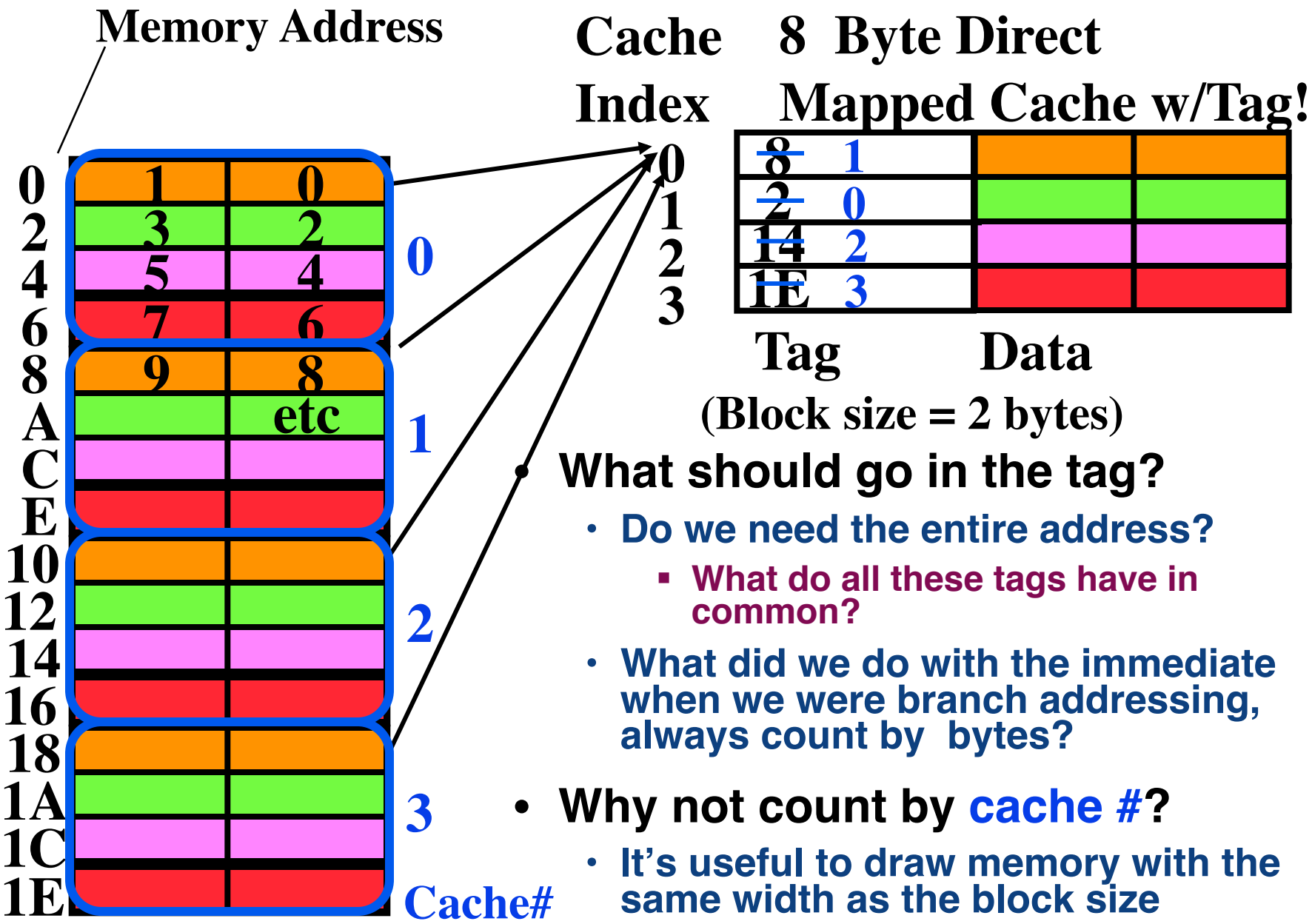
Direct-Mapped Cache (3/4)



- When we ask for a byte, the system finds out the right block, and loads it all!
  - How does it know right block?
  - How do we select the byte?
- E.g., Mem address 11101?
- How does it know WHICH colored block it originated from?
  - What do you do at baggage claim?

Pearce, Summer 2010 © UCB

Direct-Mapped Cache (4/4)

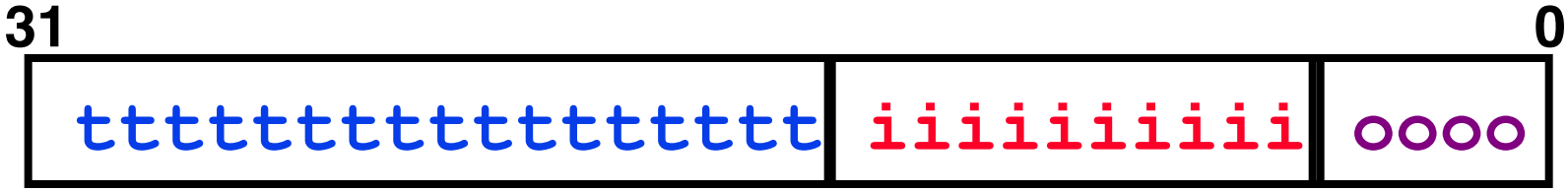


- What should go in the tag?
  - Do we need the entire address?
    - What do all these tags have in common?
  - What did we do with the immediate when we were branch addressing, always count by bytes?
- Why not count by cache #?
  - It's useful to draw memory with the same width as the block size

Pearce, Summer 2010 © UCB

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

Pearce, Summer 2010 © UCB

Direct-Mapped Cache Terminology

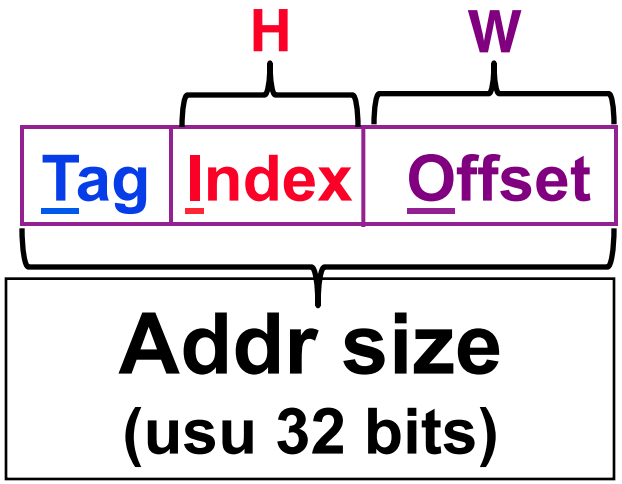
- All fields are read as unsigned integers.
- Index
  - specifies the cache index (which "row"/ block of the cache we should look in)
- Offset
  - once we've found correct block, specifies which byte within the block we want
- Tag
  - the remaining bits after offset and index are determined; these are used to distinguish between all the memory addresses that map to the same location

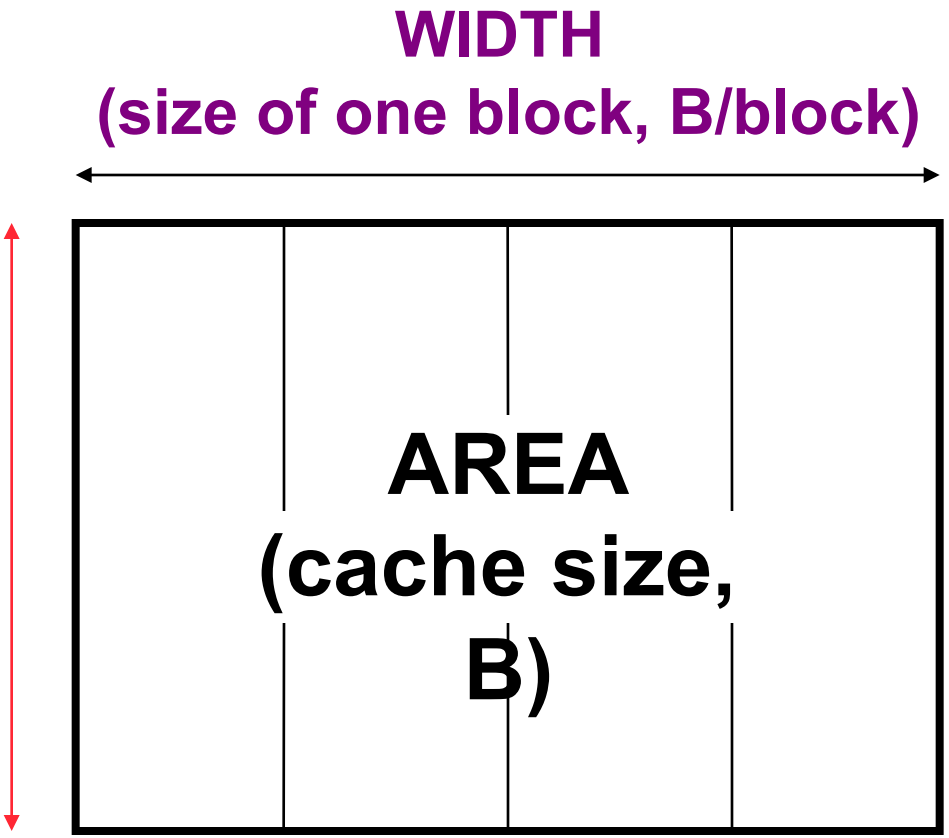
Pearce, Summer 2010 © UCB

TIO Dan's great cache mnemonic

AREA (cache size, B)
= HEIGHT (# of blocks)
*** WIDTH (size of one block, B/block)**

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





Pearce, Summer 2010 © UCB

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 we're using a 32-bit architecture
- Offset
  - need to specify correct byte within a block
  - block contains 2 bytes

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

Pearce, Summer 2010 © UCB

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

Pearce, Summer 2010 © UCB

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
- Why not full 32 bit address as tag?
  - Not all bytes within a block have the same address. So we shouldn't include offset
  - Index is the same for every address within a block, so it's redundant in tag check, thus can leave off to save memory (here 8 bits)

Pearce, Summer 2010 © UCB

Caching Terminology

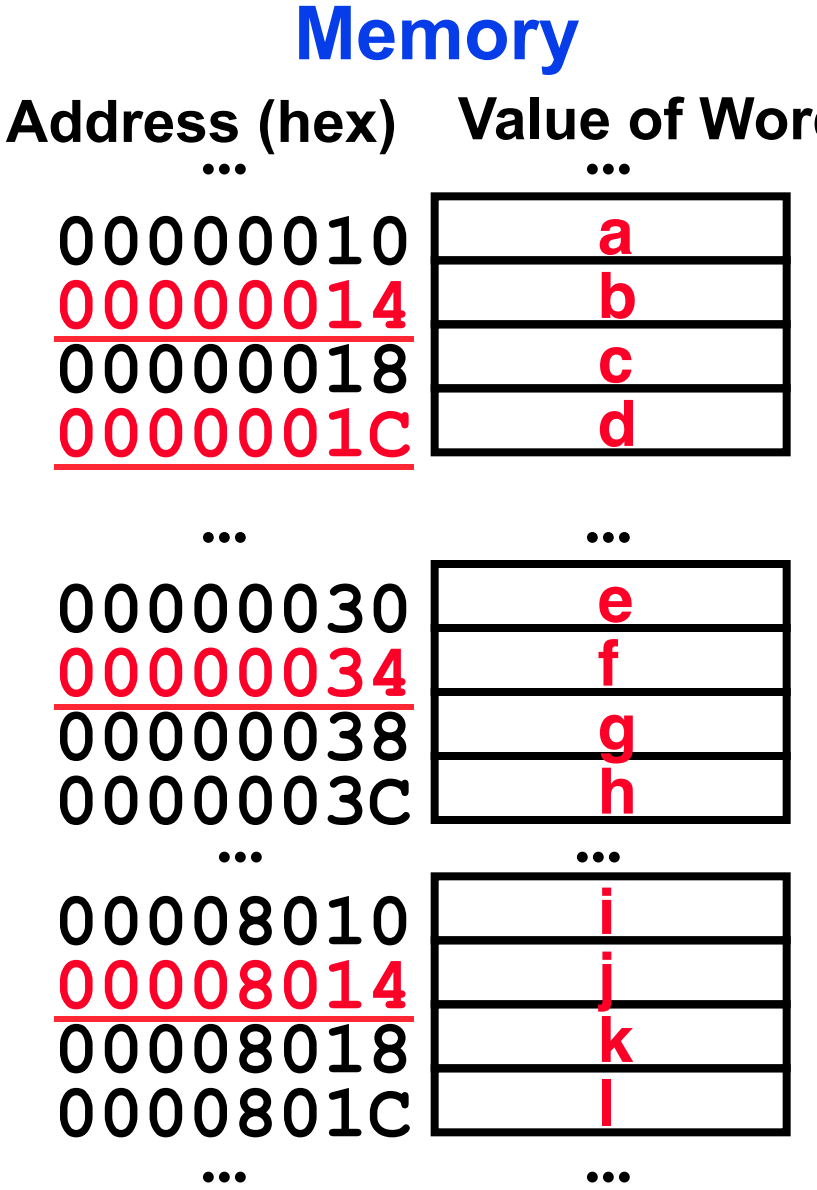
- When reading memory, 3 things can happen:
  - cache hit:
    cache block is valid and contains proper address, so read desired word
  - cache miss:
    nothing in cache in appropriate block, so fetch from memory
  - cache miss, block replacement:
    wrong data is in cache at appropriate block, so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

Pearce, Summer 2010 © UCB

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 vals here:



Pearce, Summer 2010 © UCB

Direct-Mapped Cache Example (2/3)

- **Offset**
  - block contains 16 bytes

    $= 2^4$ bytes → 4 bits for offset
- **Index:**
  - cache contains 16 KB = 2^{14} bytes
  - block contains 16 B = 2^4 bytes

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

    $= 2^{10}$ blocks/cache → 10 bits for index
- **Tag:**
  - 32 bit address → 10 bits for index, 4 for offset

    = 32 – 10 – 4 = 18 bits for tag

Pearce, Summer 2010 © UCB

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

Pearce, Summer 2010 © UCB

16 KB Direct Mapped Cache, 16B blocks

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

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					

Pearce, Summer 2010 © UCB

1. Load word from 0x00000014

- 00000000000000000000 0000000001 0100

Tag field | Index field | Offset

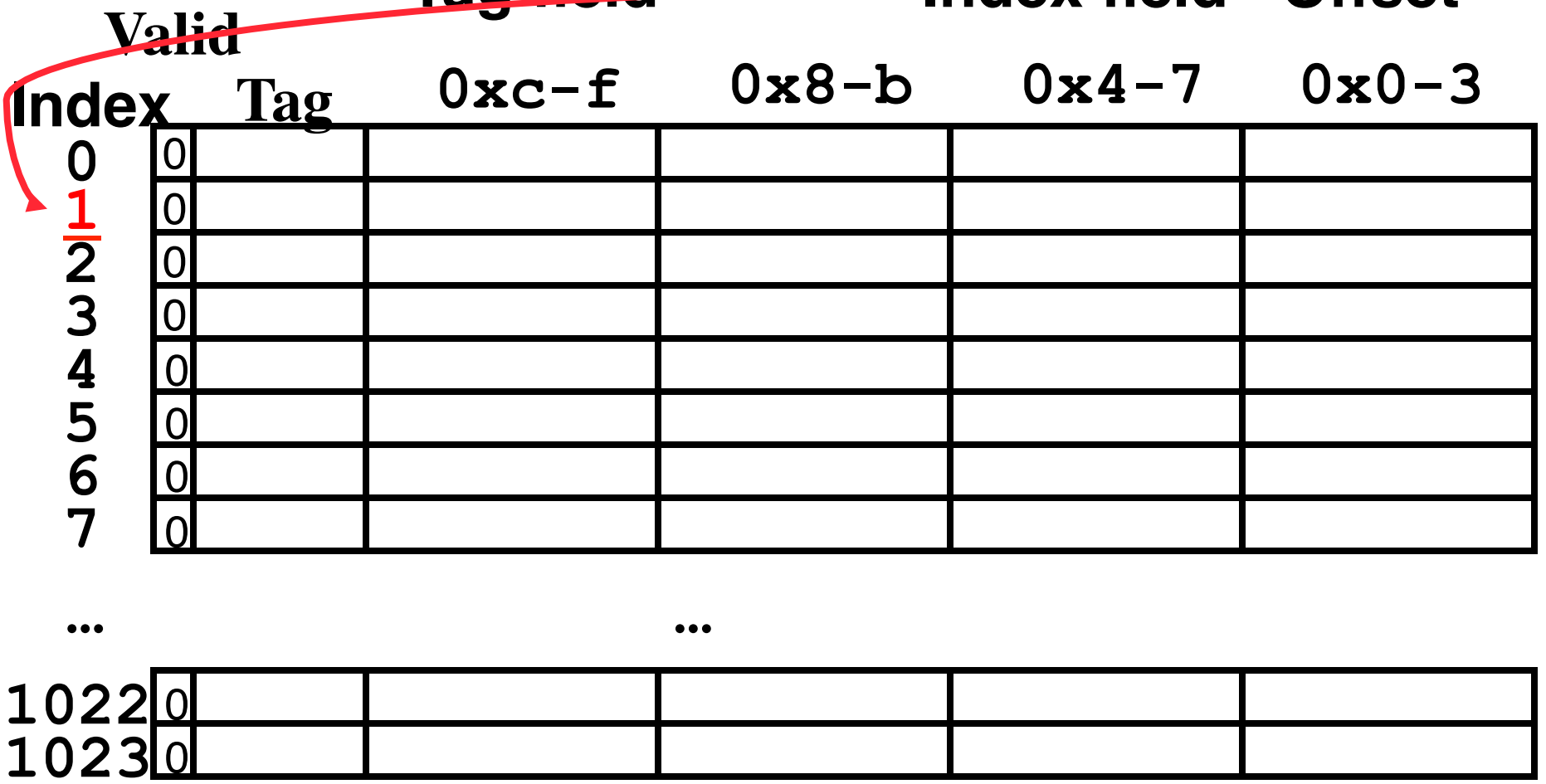
Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Pearce, Summer 2010 © UCB

So we read block 1 (0000000001)

- 00000000000000000000 0000000001 0100

Tag field | Index field | Offset

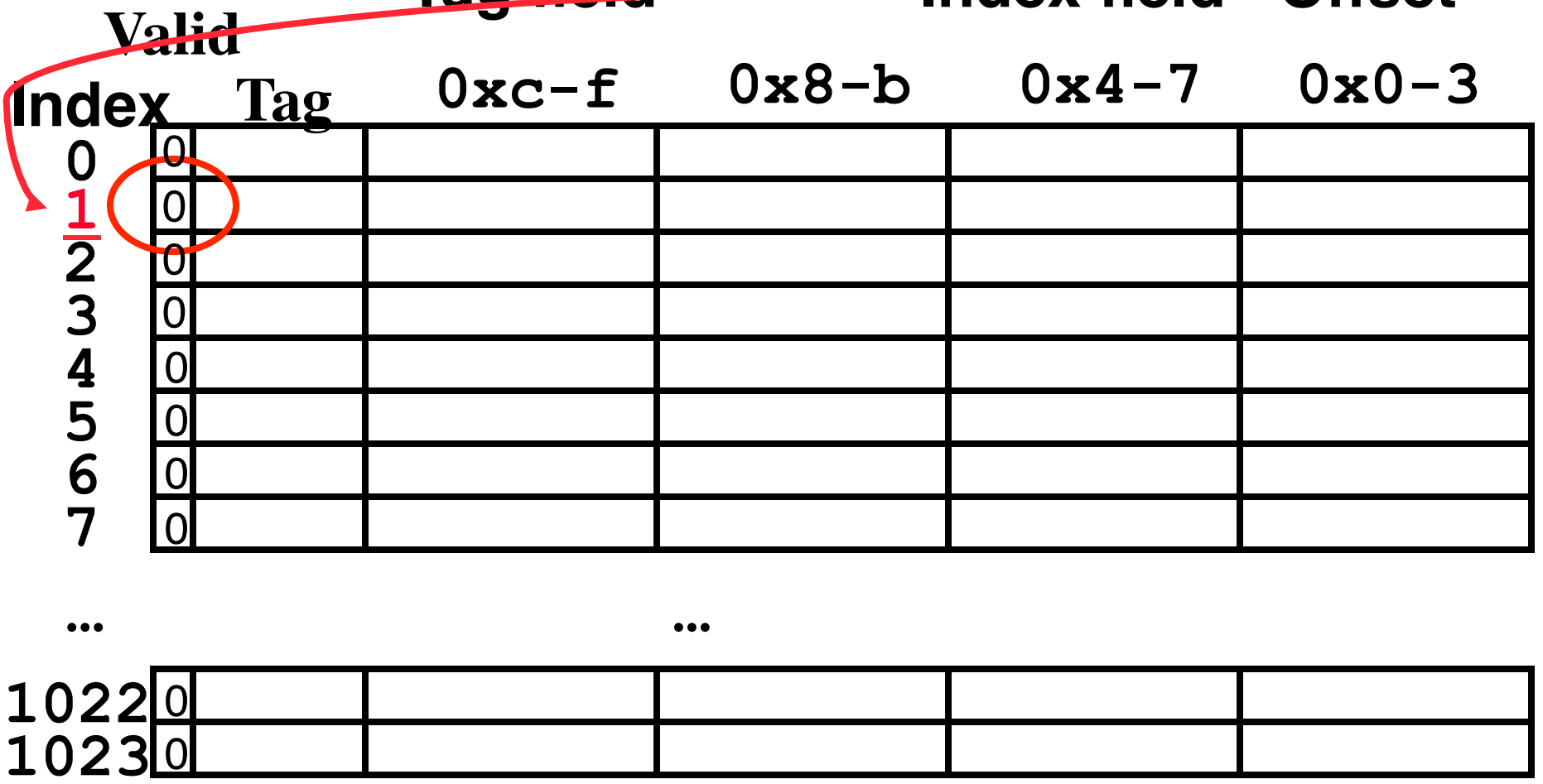


Pearce, Summer 2010 © UCB

No valid data

- 00000000000000000000 0000000001 0100

Tag field | Index field | Offset

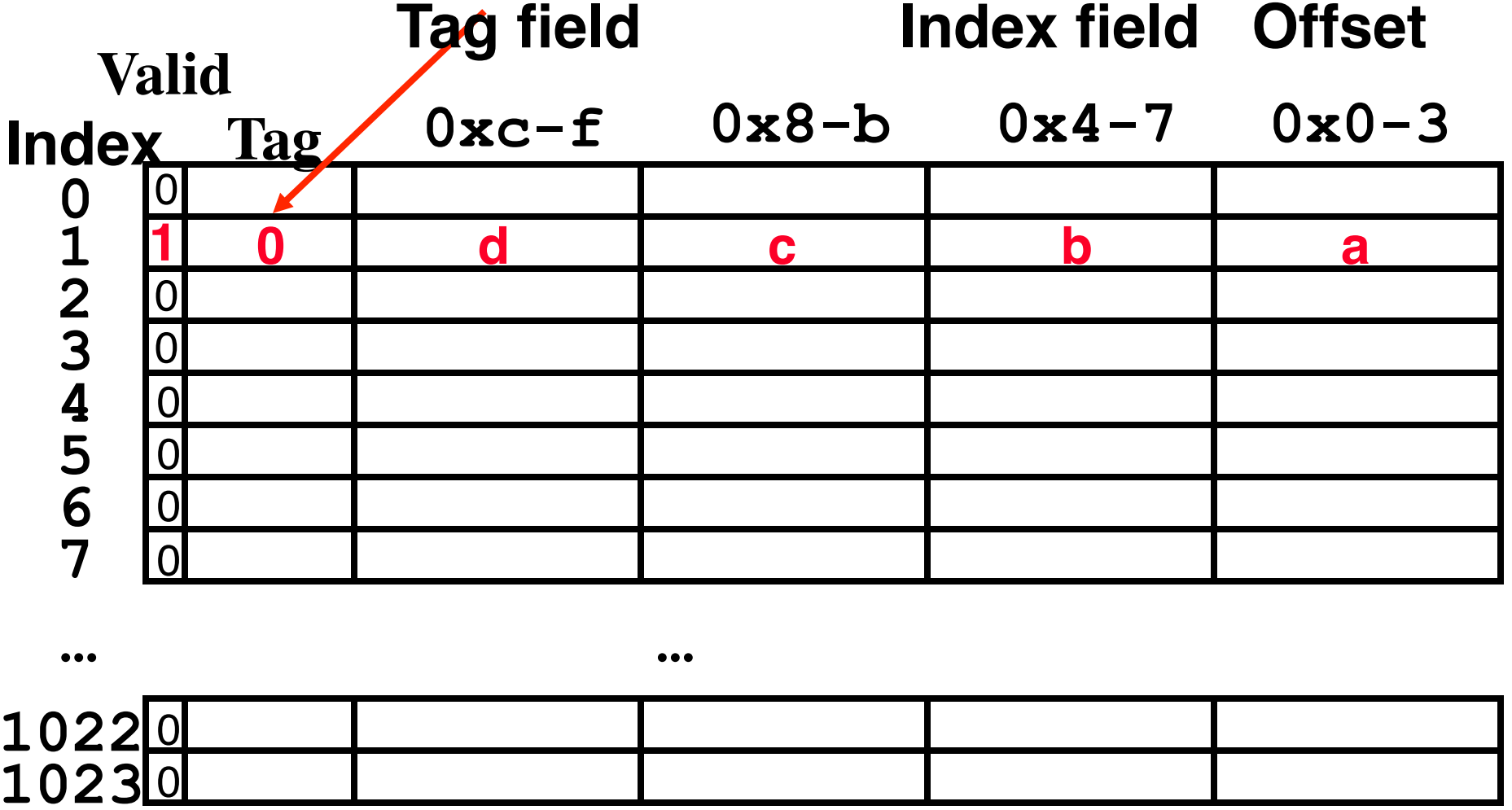


Pearce, Summer 2010 © UCB

So load that data into cache, setting tag, valid

- 00000000000000000000 0000000001 0100

Tag field | Index field | Offset



Pearce, Summer 2010 © UCB

Read from cache at offset, return word b

- 00000000000000000000 0000000001 0100

Tag field | Index field | 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					

Pearce, Summer 2010 © UCB

2. Load word from 0x0000001C

- 00000000000000000000 0000000001 1100

Tag field | Index field | 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					

Pearce, Summer 2010 © UCB

Index is Valid

- 00000000000000000000 0000000001 1100

Tag field | Index field | 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					

Pearce, Summer 2010 © UCB

Index valid, Tag Matches

- 00000000000000000000 0000000001 1100

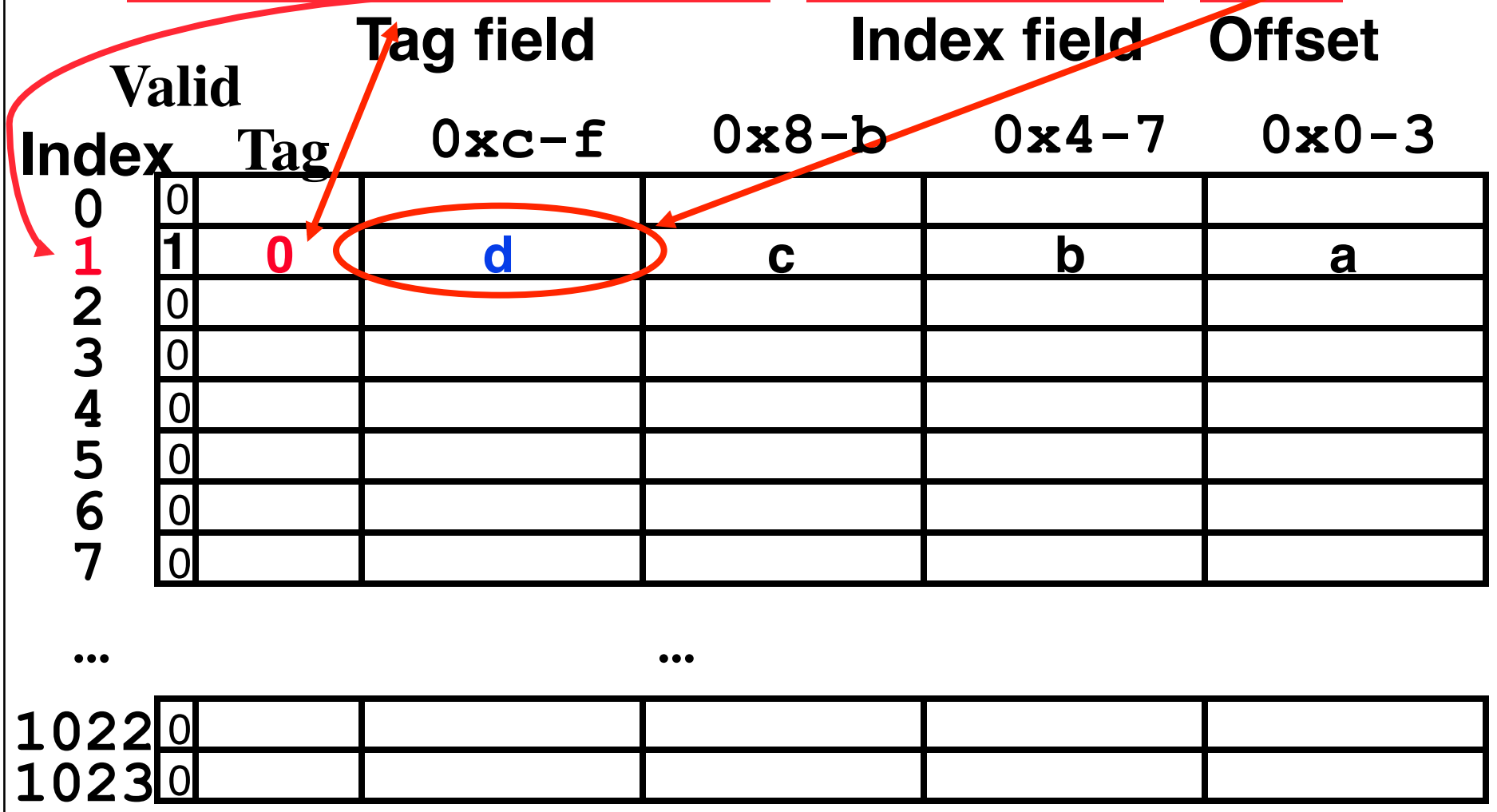
Tag field | Index field | Offset

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					

Pearce, Summer 2010 © UCB

Index Valid, Tag Matches, return d

- 00000000000000000000 0000000001 1100



Pearce, Summer 2010 © UCB

Peer Instruction

1. All caches take advantage of spatial locality.
2. All caches take advantage of temporal locality.
3. If you know your computer's cache size, you can often make your code run faster.

	123
a)	FFT
b)	FTF
c)	FTT
d)	TFT
e)	TTT

Pearce, Summer 2010 © UCB

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 lower level contains "most used" data from next higher level
  - exploits temporal & spatial locality
  - do the common case fast, worry less about the exceptions (design principle of MIPS)
- Locality of reference is a Big Idea

Pearce, Summer 2010 © UCB