Lecturer:
Yuanqing
Cheng

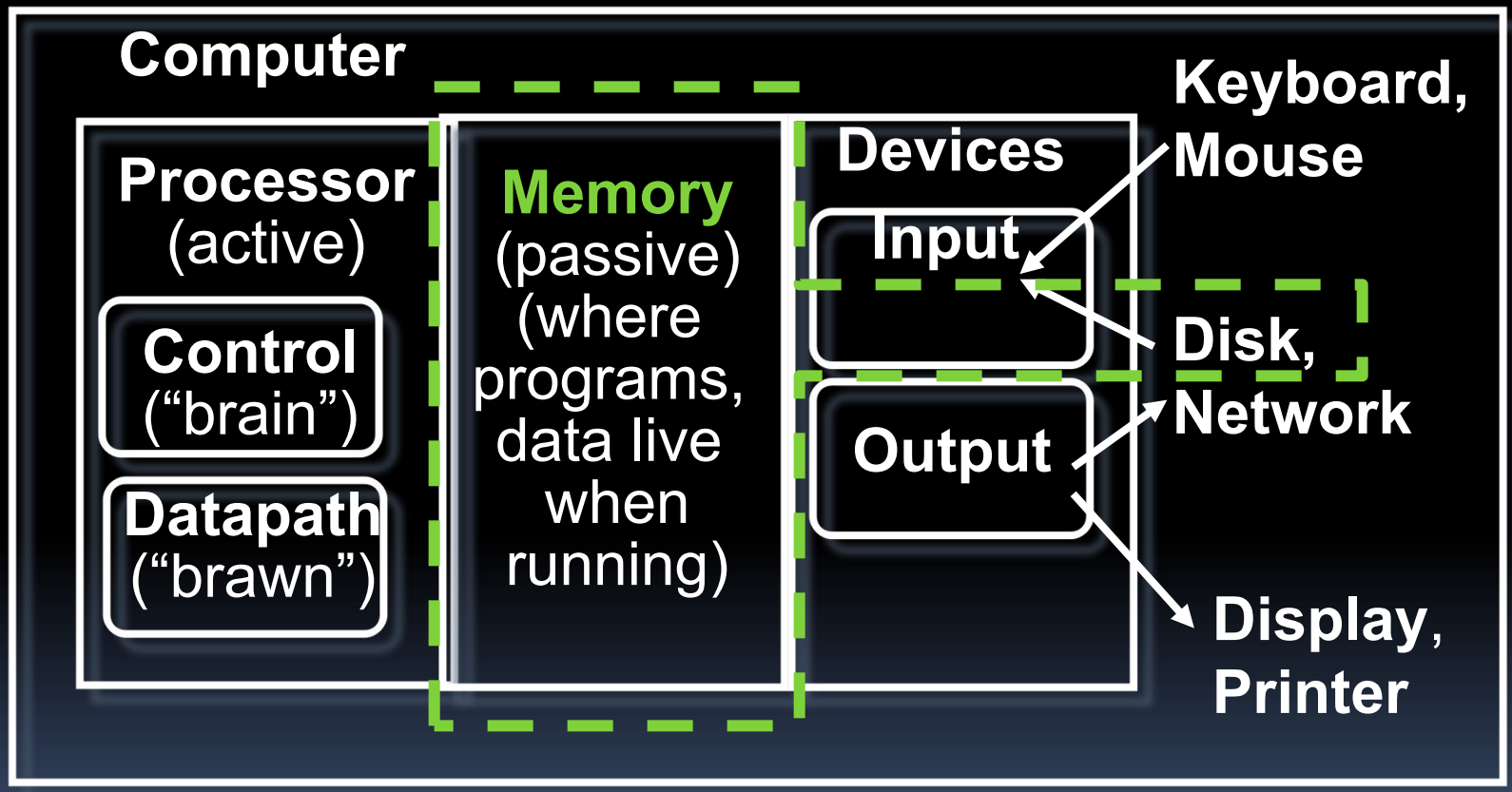
Computer Architecture
(计算机体系结构)

Lecture 25 – Caches I
2020-10-26

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 next week)**
  - **Superscalar (parallelism)**
  - **Out-of-order execution**

The Big Picture



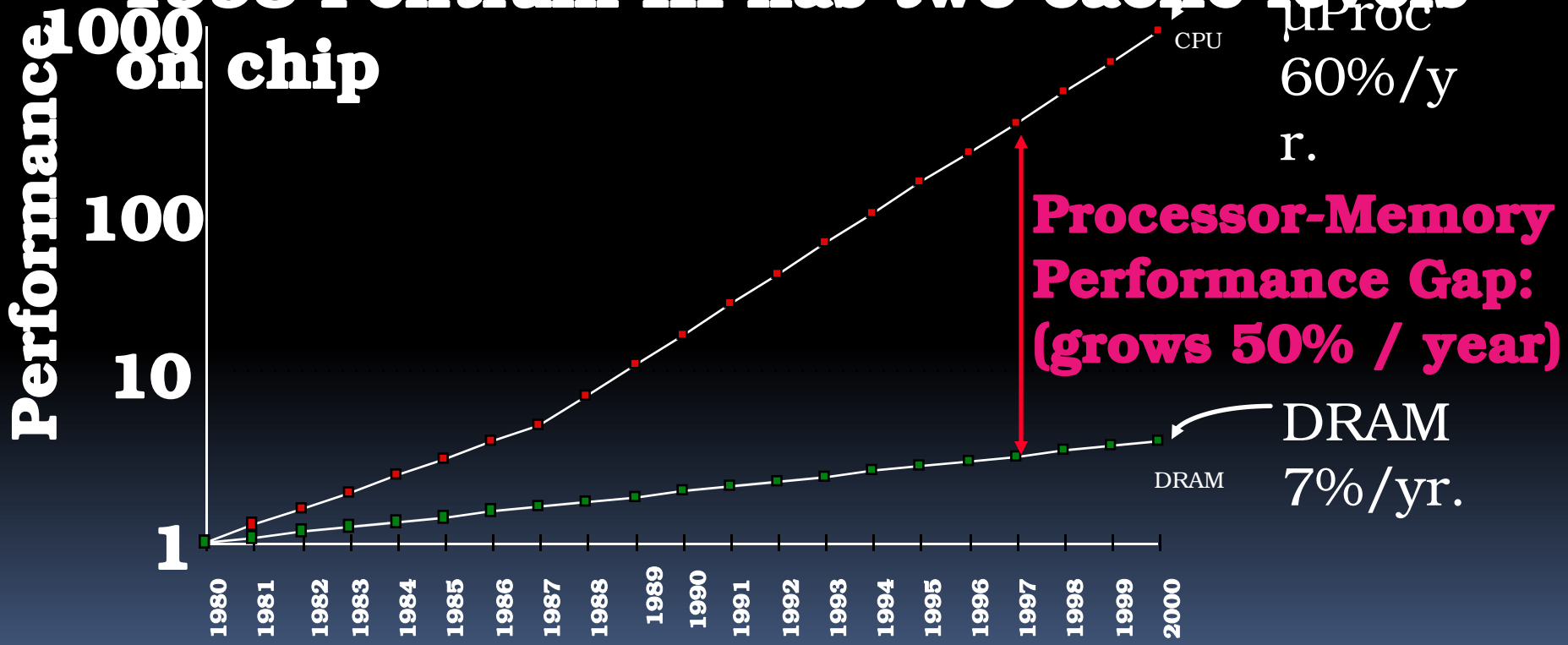
Memory Hierarchy

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**

Motivation: Why We Use Caches (written $)

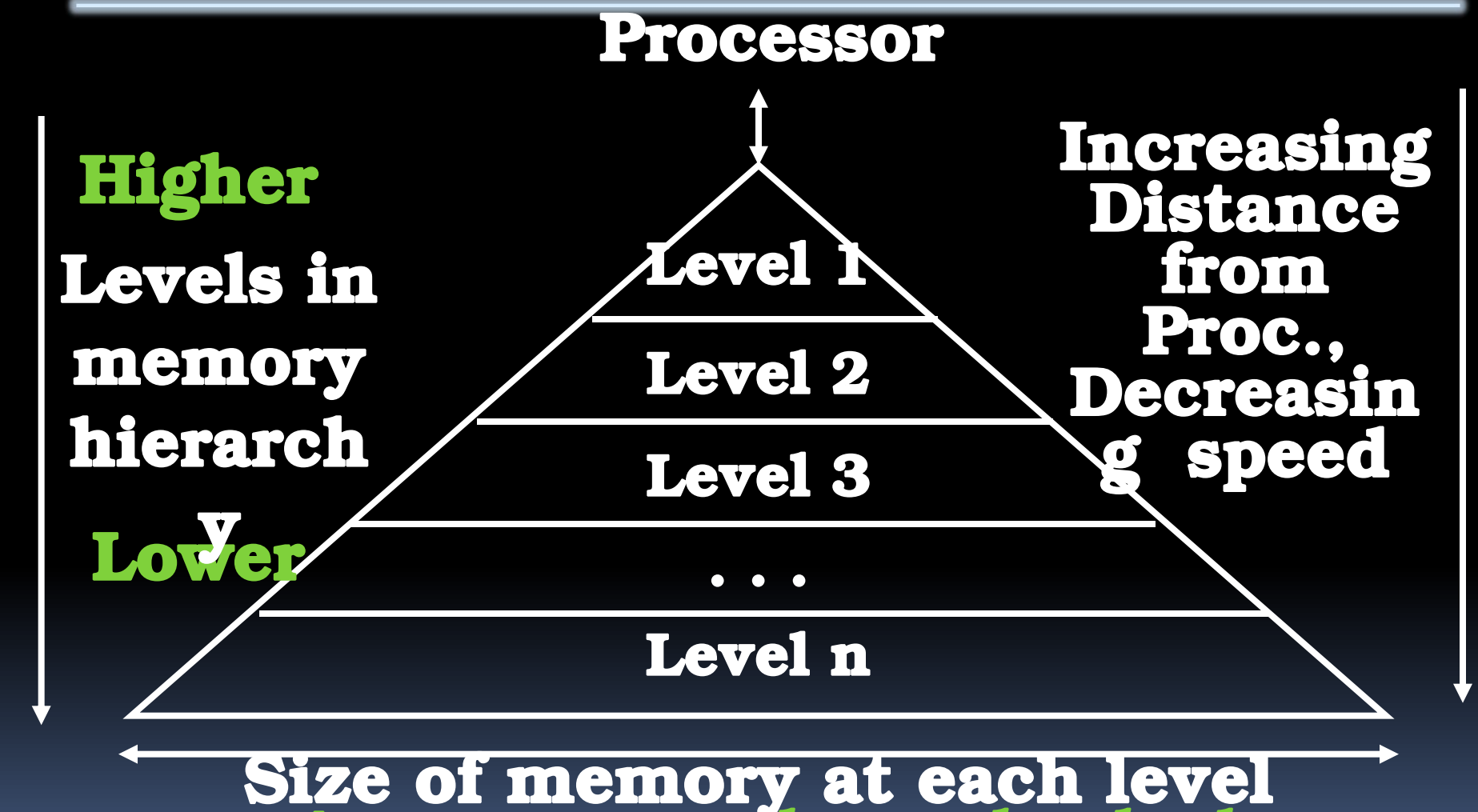
- **1989 first Intel CPU with cache on chip**
- **1998 Pentium III has two cache levels on chip**



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.**

Memory Hierarchy



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

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?)**
- **Memory Hierarchy presents the processor with the illusion of a very large & fast memory**

Memory Hierarchy Analogy: Library (1/2)

- You're writing a term paper (Processor) at a table in Doe
- Doe Library is equivalent to disk
  - essentially limitless capacity
  - very slow to retrieve a book
- Table is main 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 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 on table as possible, since faster than going to library**

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.**

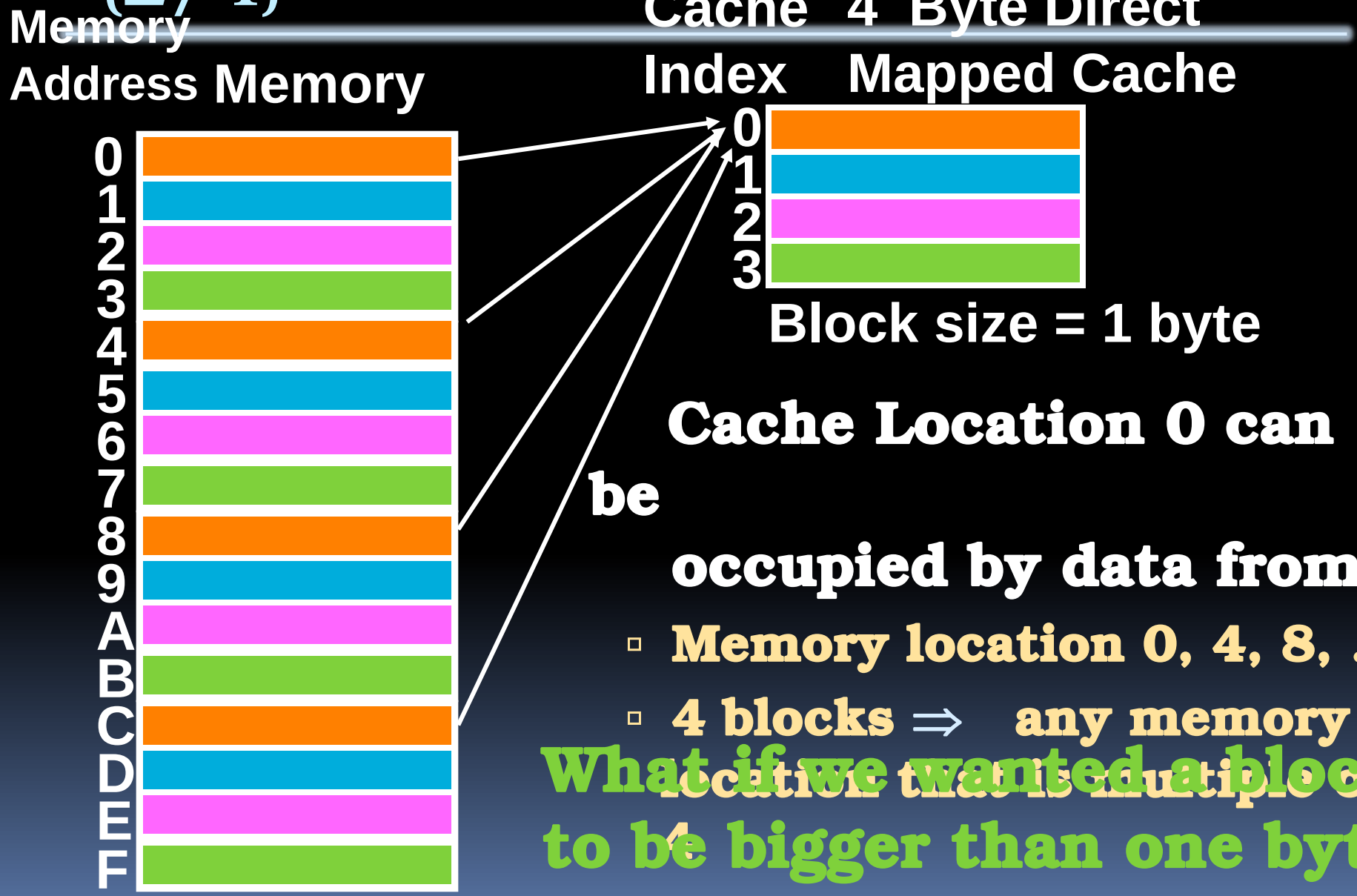
Cache Design

- **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?**

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)

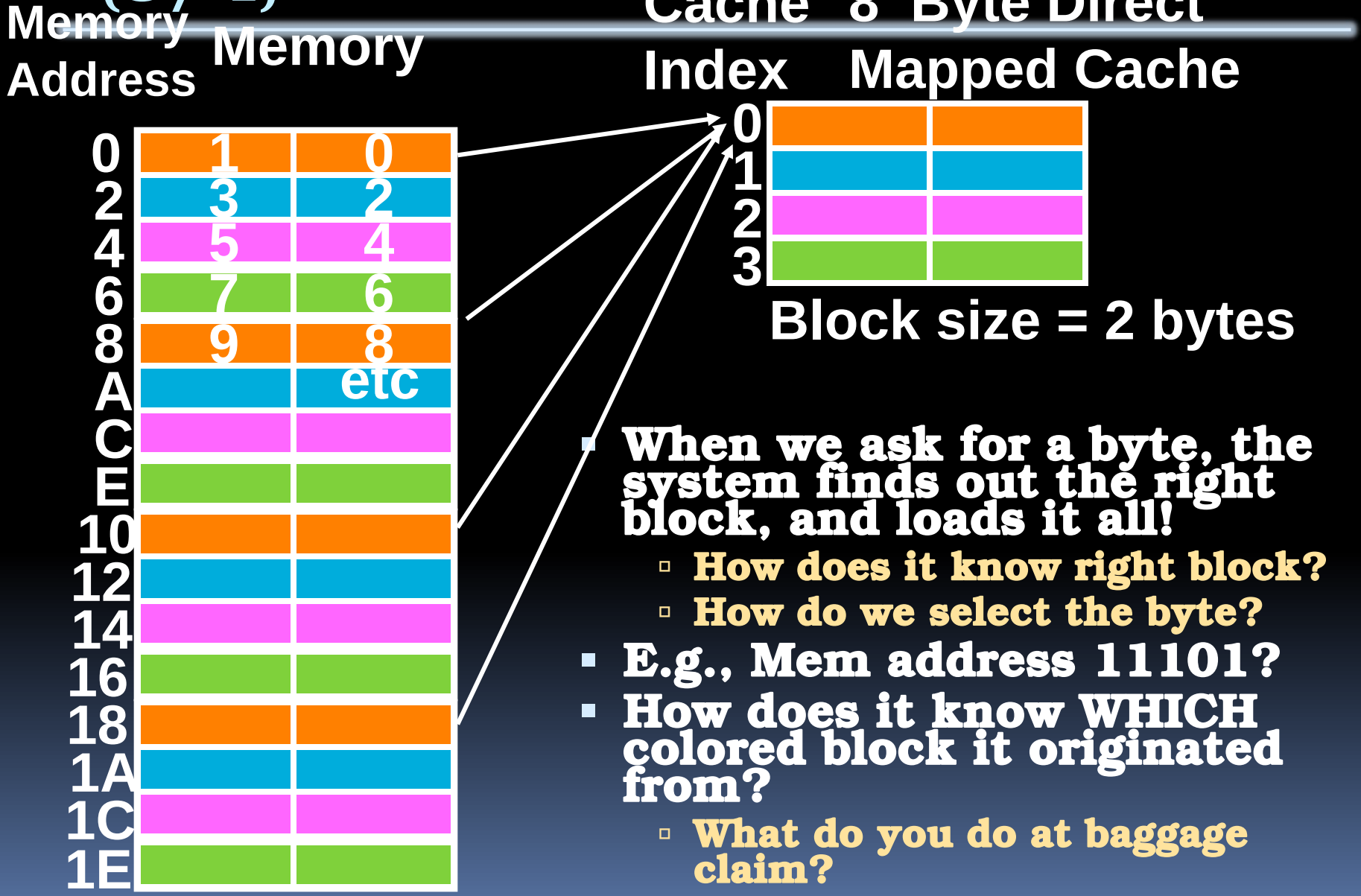


Cache Location 0 can be occupied by data from:

- **Memory location 0, 4, 8, …**
- **4 blocks ⇒ any memory location that is multiple of 4**

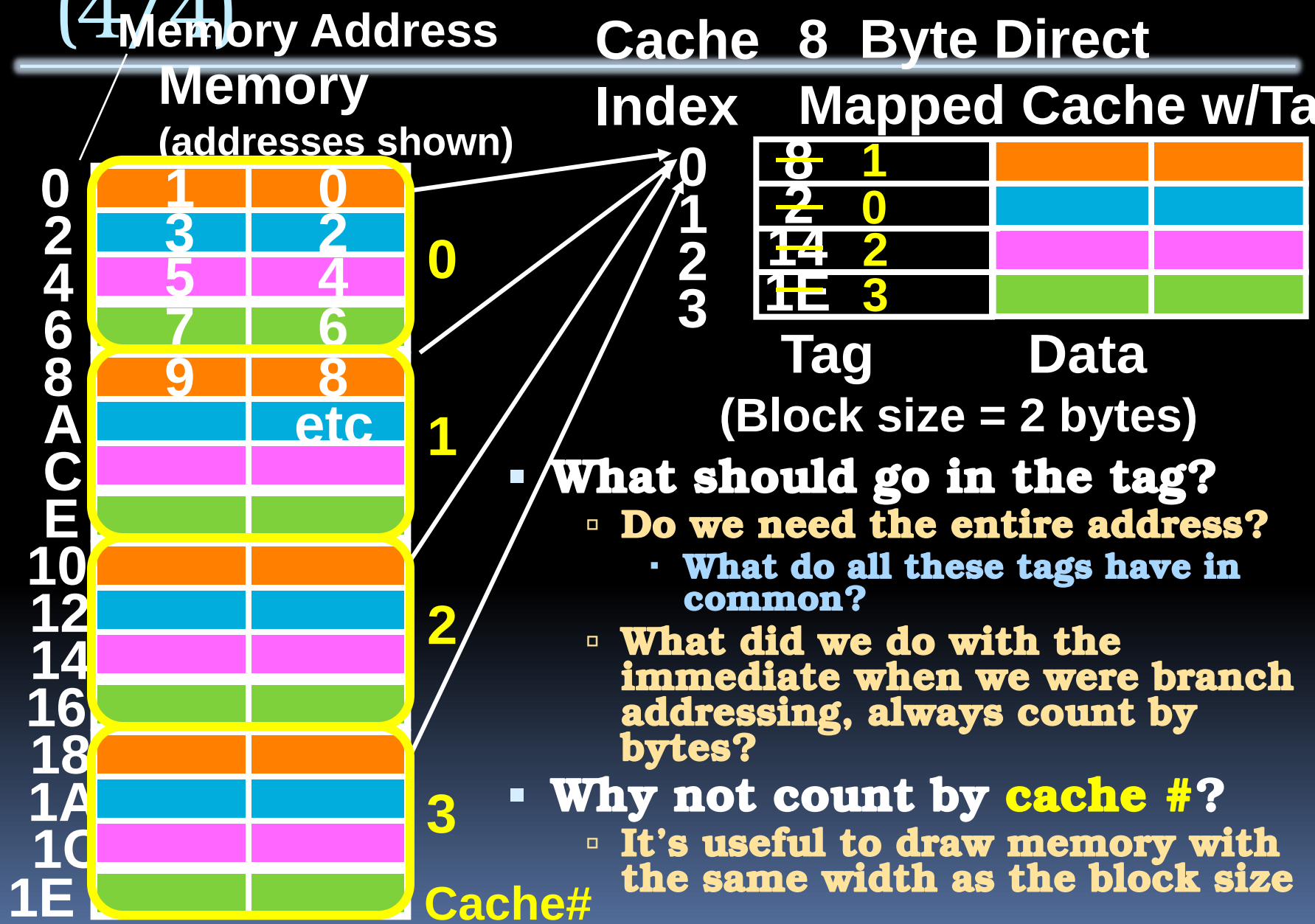
What if we wanted a block to be bigger than one byte?

Direct-Mapped Cache (3/4)



- When we ask for a byte, the 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?

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

Direct-Mapped Cache Terminology

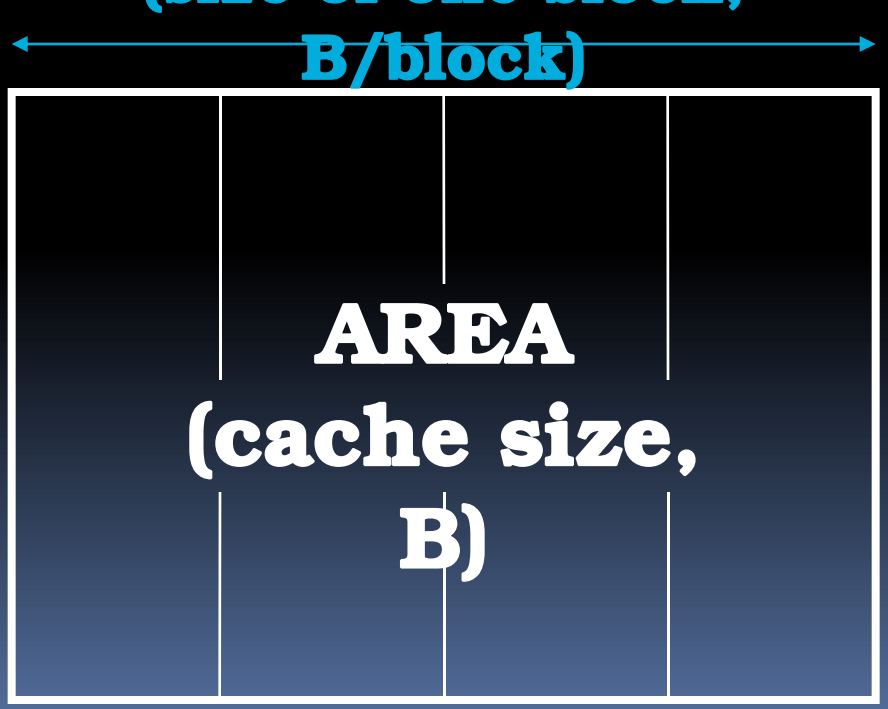
- **All fields are read as unsigned integers.**
- **Index**
  - **specifies the cache index (which "row"/block of the cache we should look in)**
- **Offset**
  - **once we've found correct block, specifies which byte within the block we want**
- **Tag**
  - **the remaining bits after offset and index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

TIO great cache mnemonic

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

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

Tag	Index	Offset



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

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**
- **Why not full 32 bit address as tag?**
  - **All bytes within block need same address (4b)**
  - **Index must be same for every address within a block, so it's redundant in tag check, thus can leave off to save memory**

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**