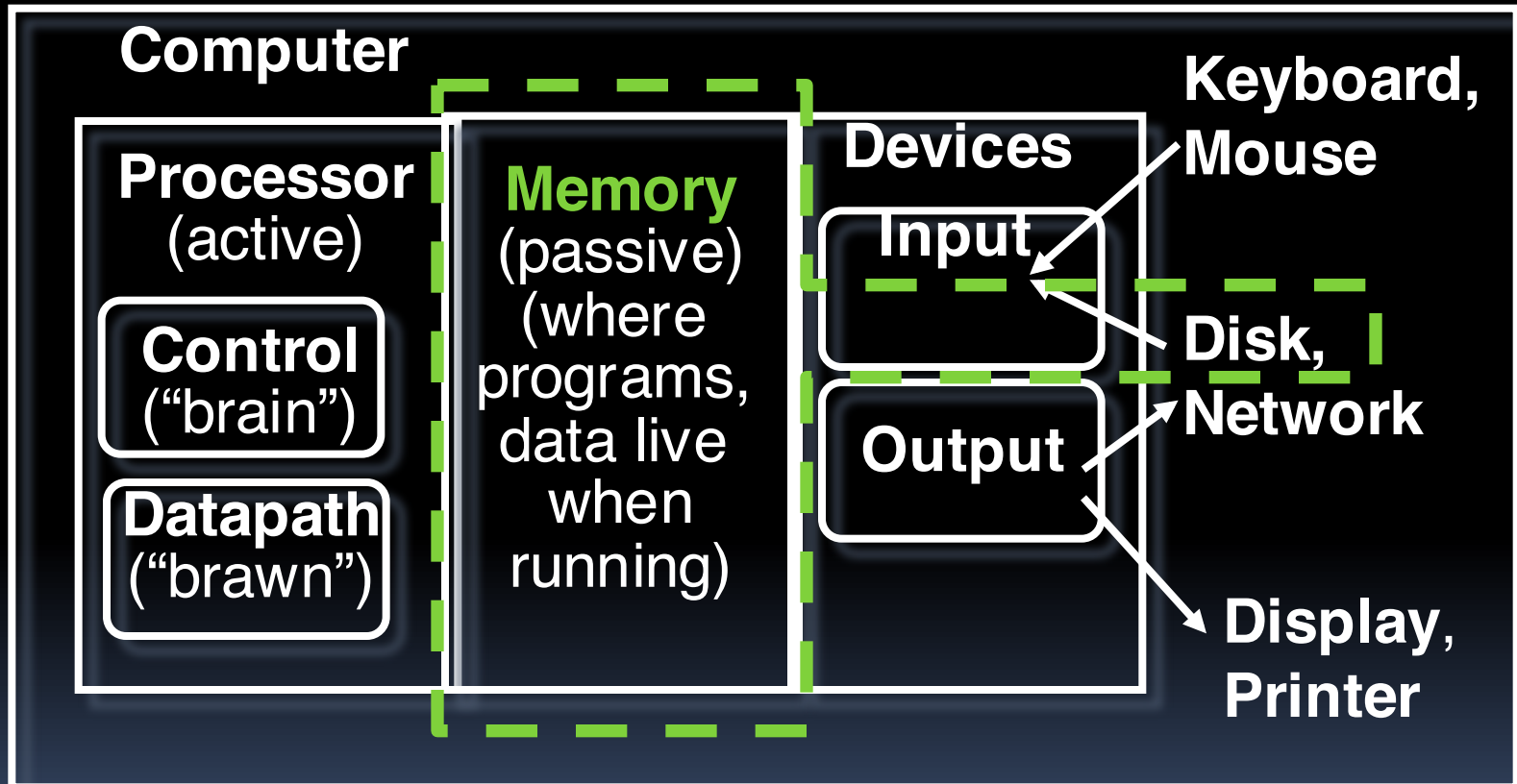


Review

- **Register Conventions:** Each register has a purpose and limits to its usage. Learn these and follow them, even if you're writing all the code yourself.
- **Logical and Shift Instructions**
 - Operate on bits individually, unlike arithmetic, which operate on entire word.
 - Use to isolate fields, either by masking or by shifting back and forth.
 - Use shift left logical, **sll**, for multiplication by powers of 2
 - Use shift right logical, **srl**, for division by powers of 2 of unsigned numbers (unsigned int)
 - Use shift right arithmetic, **sra**, for division by powers of 2 of signed numbers (int)
- **New Instructions:**
and, andi, or, ori, sll, srl, sra

The Big Picture



Memory Hierarchy

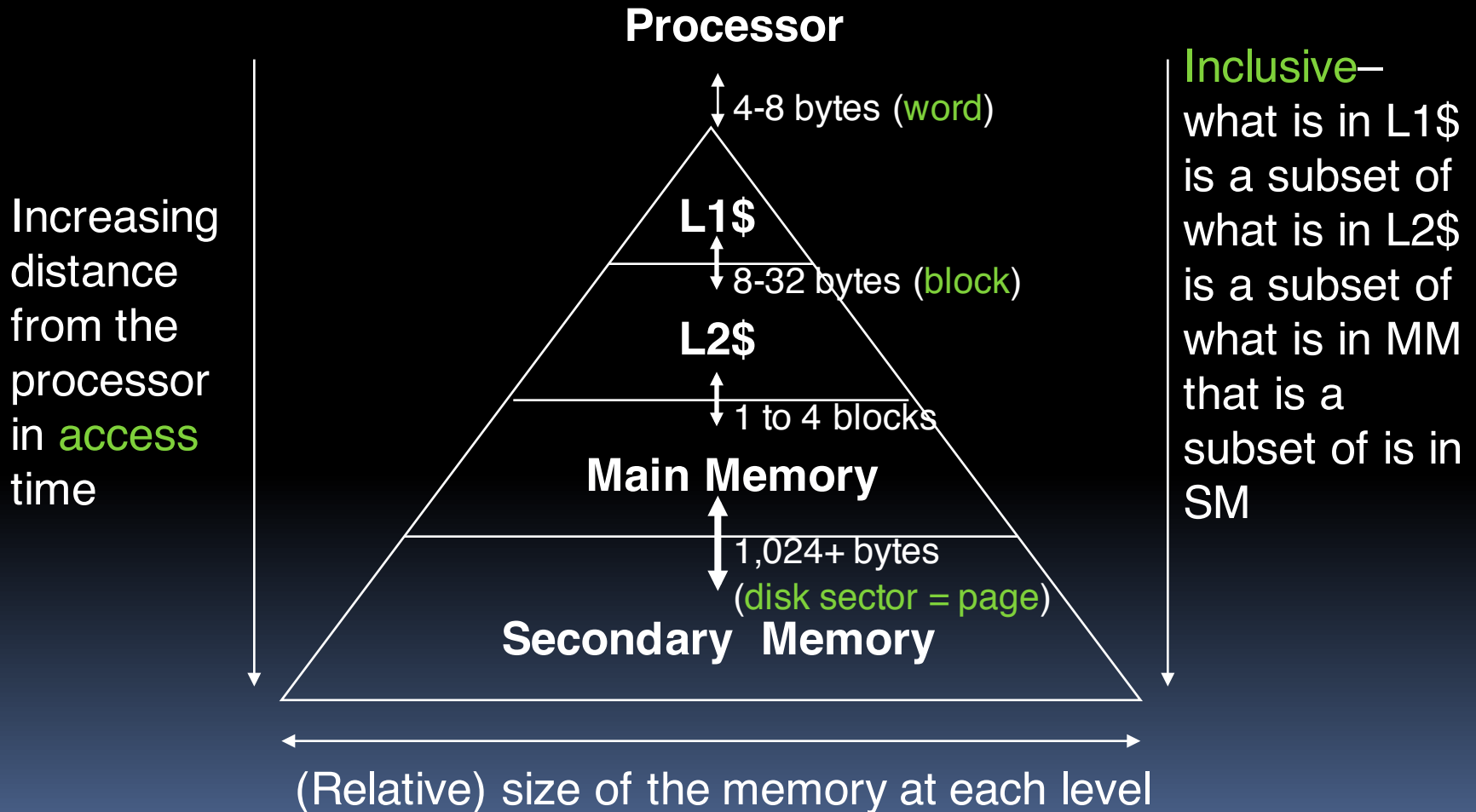
I.e., storage in computer systems

- **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

Memory Caching

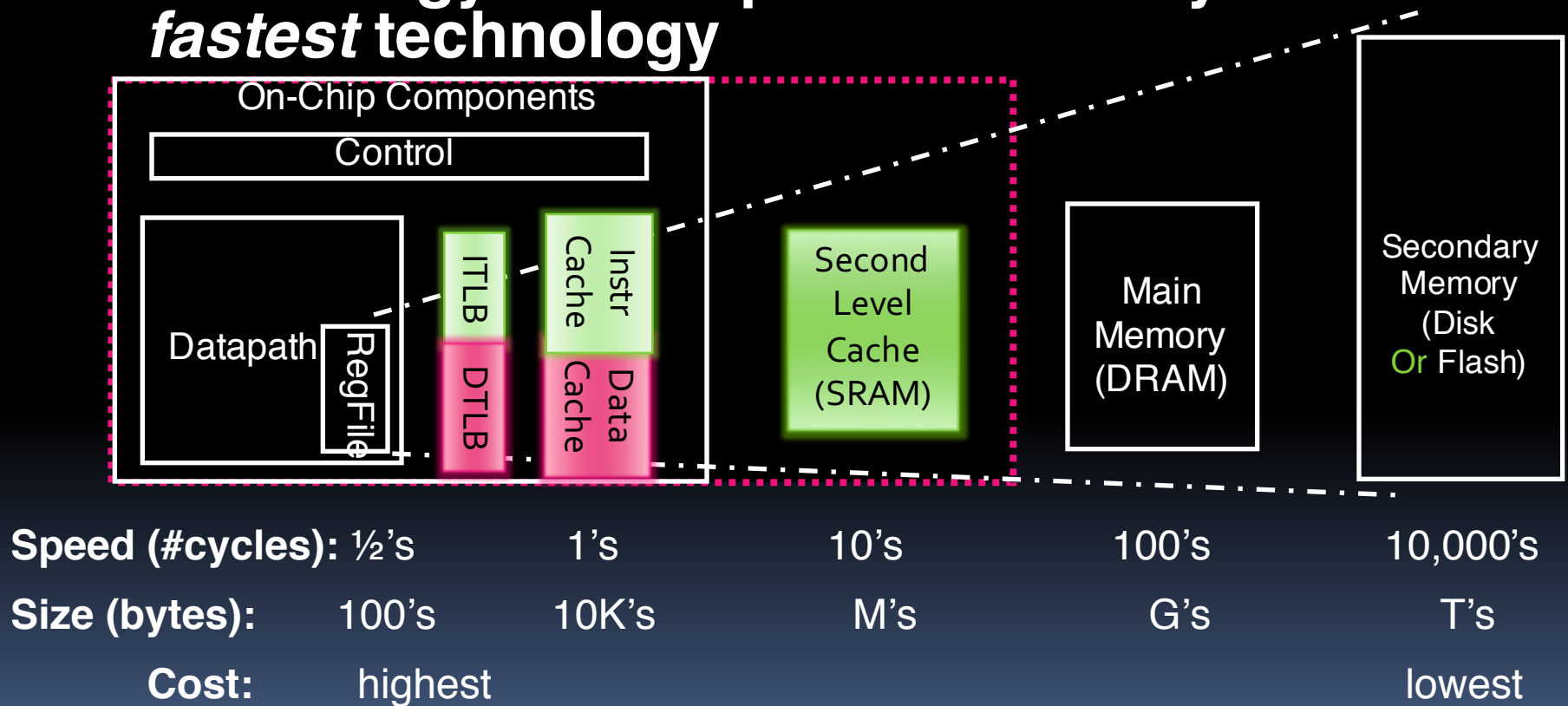
- Mismatch between processor and memory speeds leads us to add a new level: a memory **cache**
- Implemented with same IC processing technology as the CPU (usually integrated on same chip): faster but more expensive than DRAM memory.
- **Cache is a copy of a subset of main memory.**
- Most processors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

Characteristics of the Memory Hierarchy



Typical Memory Hierarchy

- **The Trick:** present processor with as much memory as is available in the *cheapest* technology at the speed offered by the *fastest* technology



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

- You're writing a term paper (Processor) at a **table** in **Stiern**
- **Stiern** 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
- 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.

Two Types of Locality

- ***Temporal Locality*** (locality in time)
 - If a memory location is referenced then it will tend to be referenced again soon
 - ⇒ Keep most recently accessed data items closer to the processor
- ***Spatial Locality*** (locality in space)
 - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon
 - ⇒ Move blocks consisting of contiguous words closer to the processor

Cache Design (for ANY cache)

- 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?

How is the Hierarchy Managed?

- **registers \leftrightarrow memory**
 - **By compiler (or assembly level programmer)**
- **cache \leftrightarrow main memory**
 - **By the cache controller hardware**
- **main memory \leftrightarrow disks (secondary storage)**
 - **By the operating system (virtual memory)**
 - **Virtual to physical address mapping assisted by the hardware (TLB)**
 - **By the programmer (files)**

Administrivia

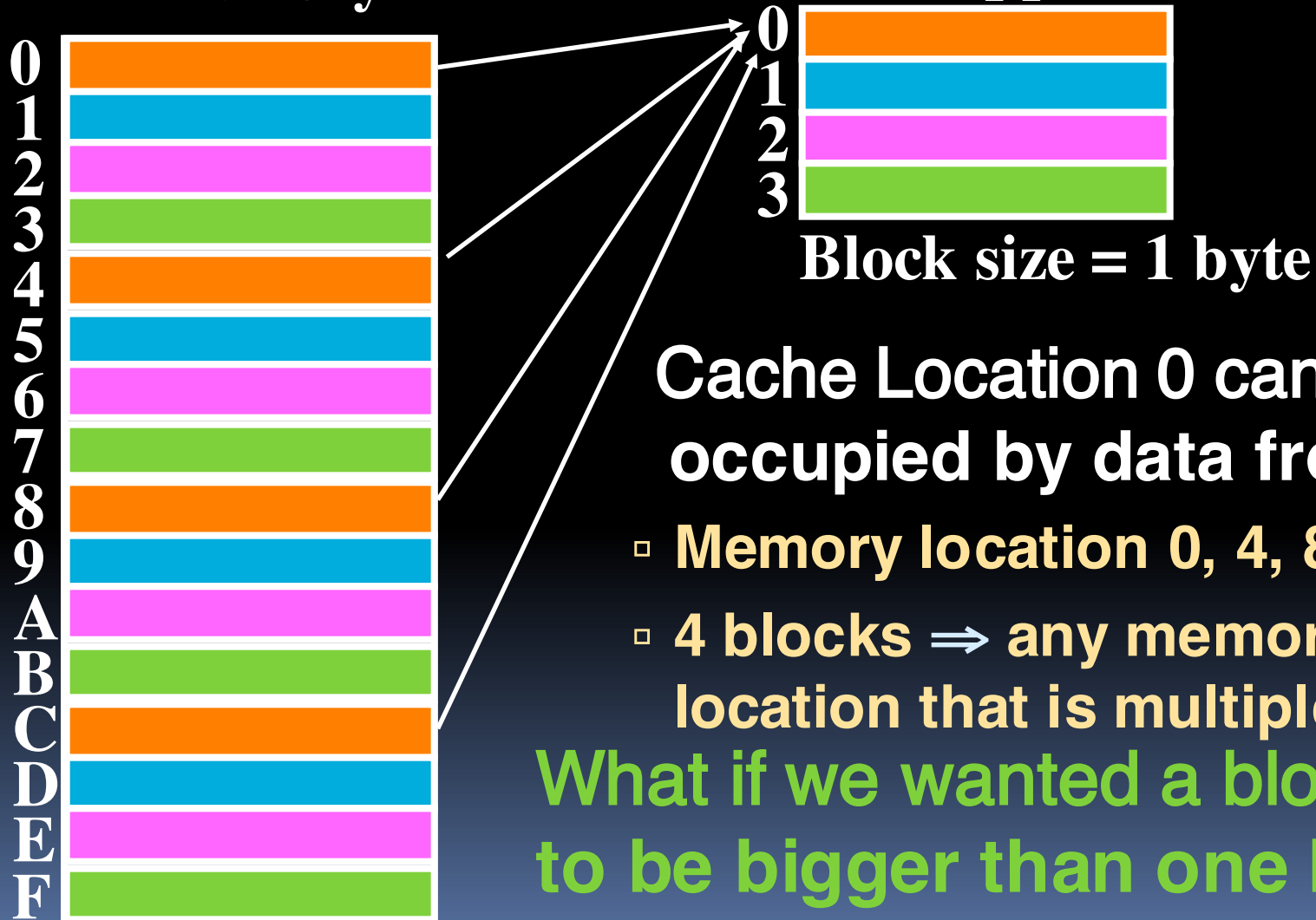
- **Midterm 1 Next Week**
- **HW 1 Solutions**
- **HW 2 Available**
- **Lab 3 Online**
- **Notes Updated**

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)

Memory Address Memory Cache Index 4 Byte Direct Mapped Cache



Cache Location 0 can be occupied by data from:

- Memory location 0, 4, 8, ...
- 4 blocks \Rightarrow 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)

Cache 8 Byte Direct
Index Mapped Cache

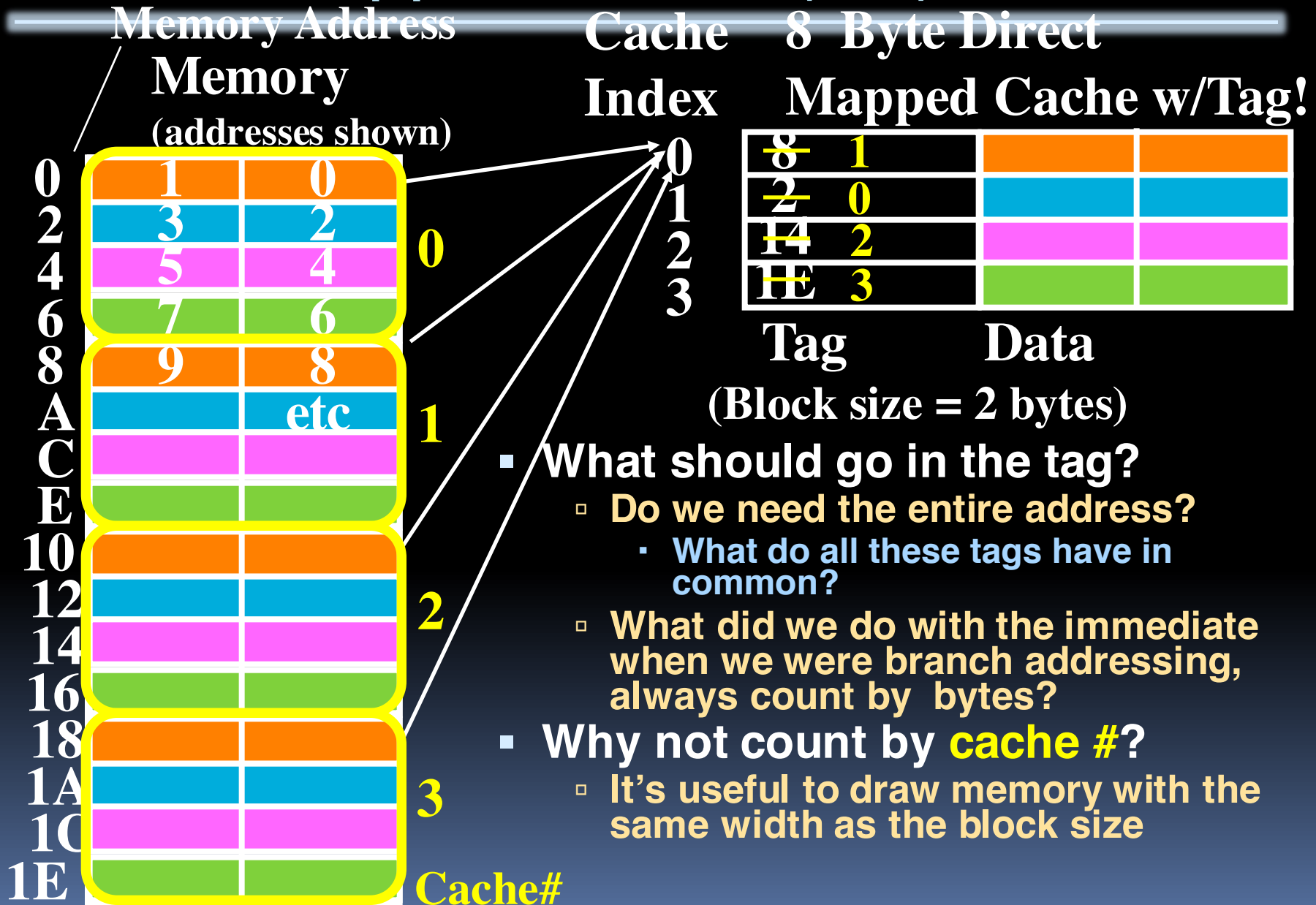
Memory Address	Memory	
0	1	0
2	3	2
4	5	4
6	7	6
8	9	8
A		etc
C		
E		
10		
12		
14		
16		
18		
1A		
1C		
1E		

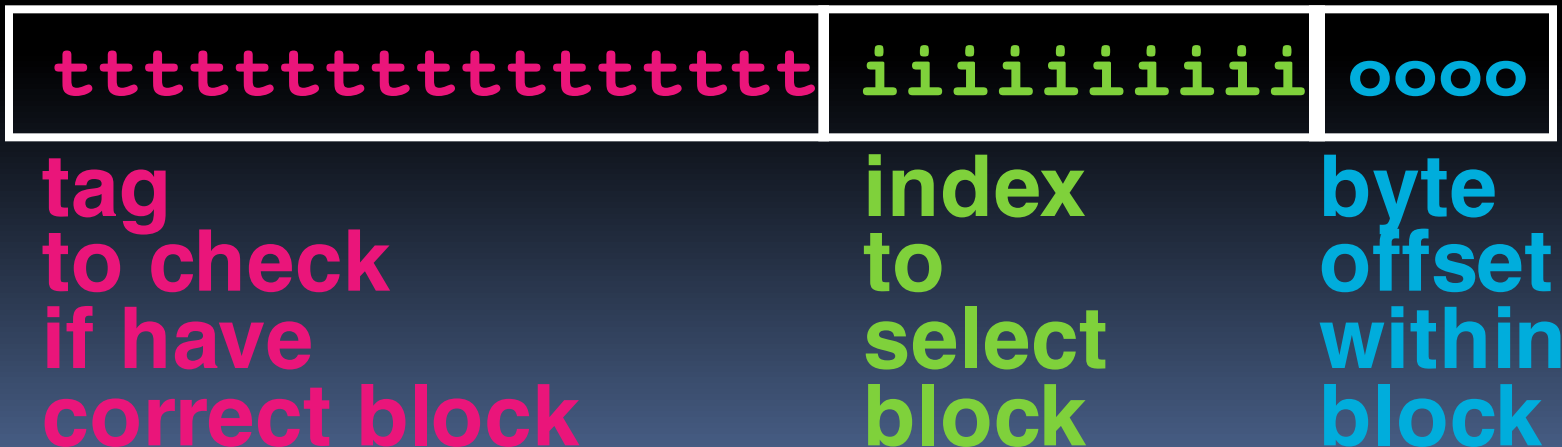
0		
1		
2		
3		

Block size = 2 bytes

- 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)





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

TIO Eddie's great cache mnemonic

AREA (cache size, B)

= HEIGHT (# of blocks)

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

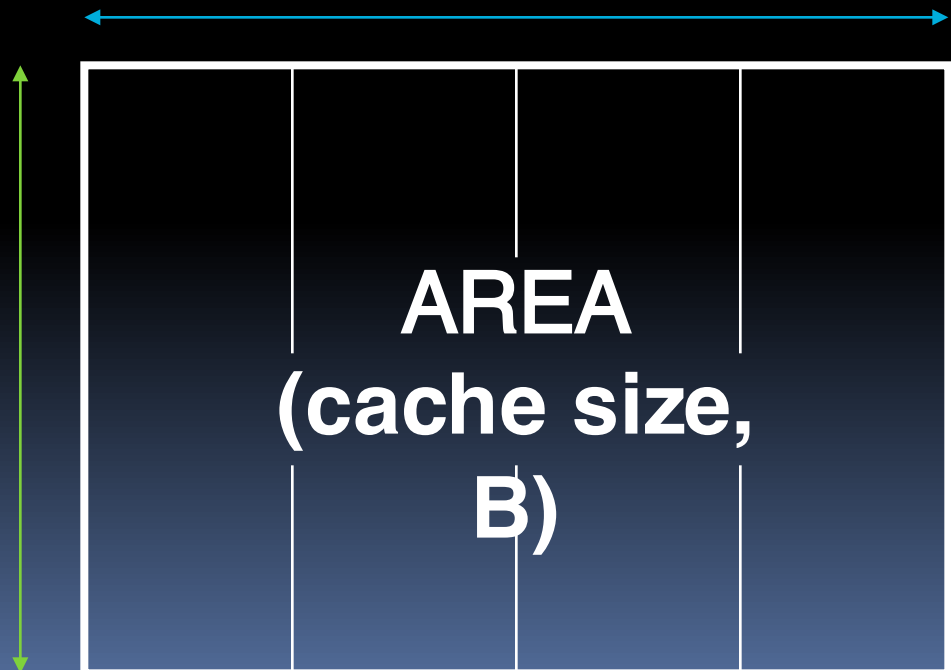
$$2^{(H+W)} = 2^H * 2^W$$

WIDTH

(size of one block, B/block)

Tag	Index	Offset
-----	-------	--------

HEIGHT
(# of blocks)



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

Direct-Mapped Cache Example (2/3)

- Index: (~index into an “array of blocks”)
 - need to specify correct block in cache
 - cache contains $8 \text{ B} = 2^3 \text{ bytes}$
 - block contains $2 \text{ B} = 2^1 \text{ bytes}$
 - # blocks/cache
 - = $\frac{\text{bytes/cache}}{\text{bytes/block}}$
 - = $\frac{2^3 \text{ bytes/cache}}{2^1 \text{ bytes/block}}$
 - = 2^2 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 (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 (here 10 bits)**

Peer Instruction

- A. For a given cache size: a larger block size can cause a lower hit rate than a smaller one.
- B. If you know your computer's cache size, you can often **make your code run faster**.
- C. Memory hierarchies take advantage of **spatial locality** by keeping the most recent data items **closer** to the processor.

	ABC
1:	FFF
1:	FFT
2:	FTF
2:	FTT
3:	TFF
3:	TFT
4:	TTF
5:	TTT

Peer Instruction Answer

- A. Yes – if the block size gets too big, fetches become more expensive and the big blocks force out more useful data.
- B. Certainly! That's call "tuning"
- C. "Most Recent" items \Rightarrow Temporal locality

- A. For a given cache size: a larger block size can cause a lower hit rate than a smaller one.
- B. If you know your computer's cache size, you can often make your code run faster.
- C. Memory hierarchies take advantage of spatial locality by keeping the most recent data items closer to the processor.

	ABC
1:	FFF
1:	FFT
2:	FTF
2:	FTT
3:	TFF
3:	TFT
4:	TTF
5:	TTT

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**