CprE 381: Computer Organization and Assembly Level Programming

Cache Design

Henry Duwe
Electrical and Computer Engineering
Iowa State University

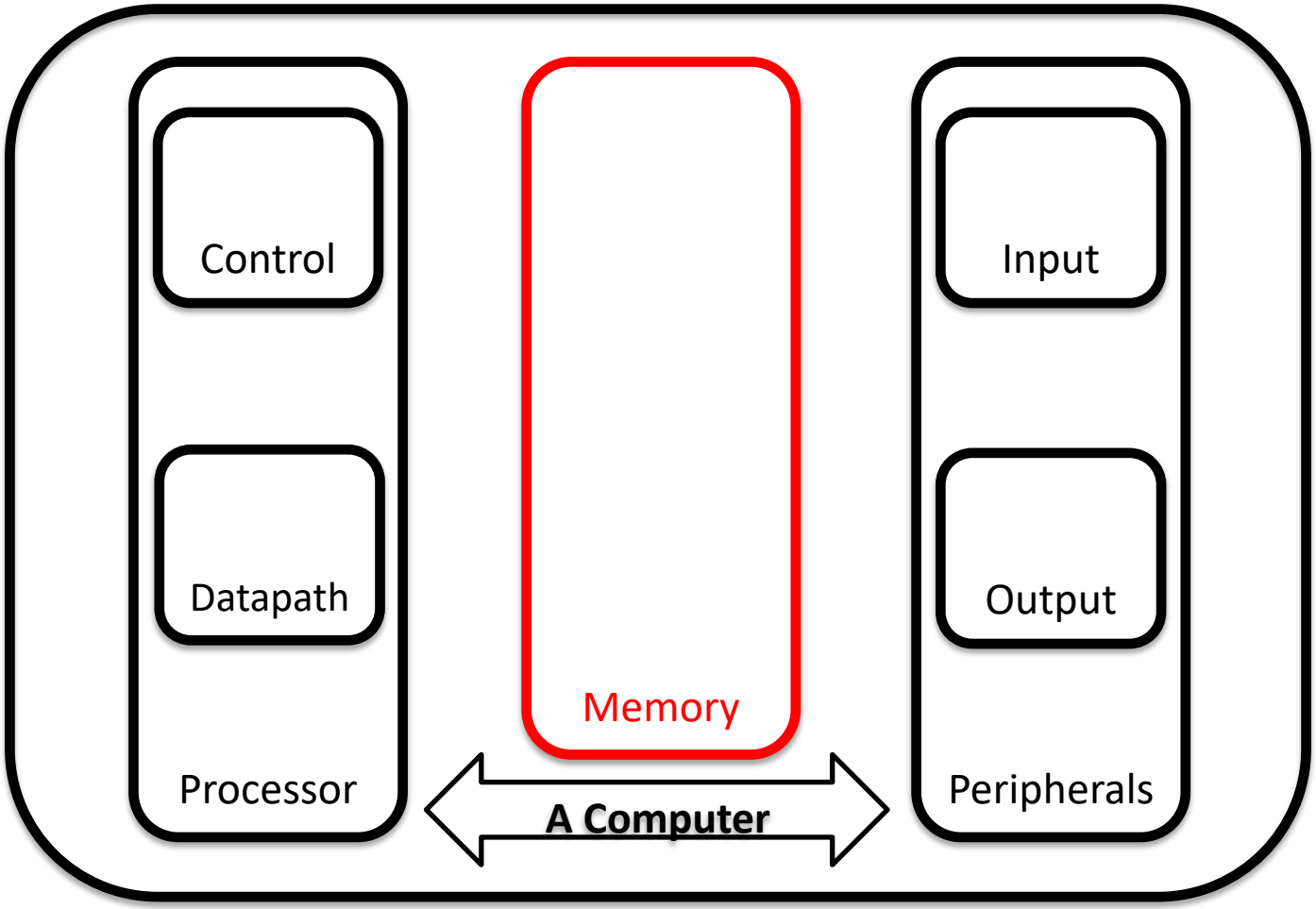
Administrative

- Project Part 3a
  - Due this week
  - **WARNING**: Much easier that Part 3b!

Administrative

- Snapshot of semester:
  - 3 weeks remain
  - In one week:
    - HW9 – Memory and caches
  - In two weeks:
    - HW10 – Caches some more
    - Project Part 3 due
  - In three weeks (Dead Week):
    - HW11 – Exam 3 question and advanced topics
    - Project Part 4 due – More in-depth report with analysis and benchmarking of processors
  - Final Exam:
    - May 6 at 7:30am-9:30am O_o
    - Please report any conflicts

Remember the System View!



Review: Small or Slow

- Unfortunately there is a tradeoff between speed, cost and capacity

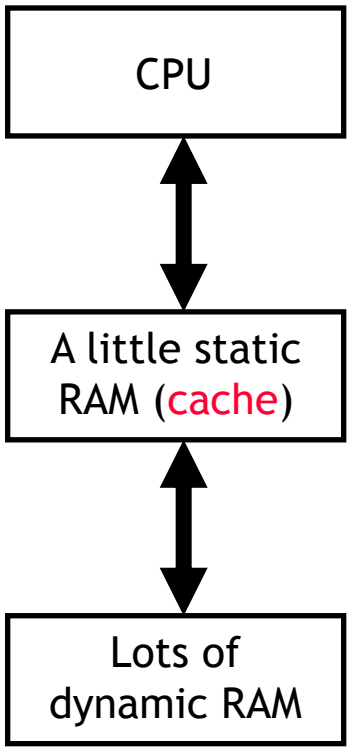
Storage	Speed	Cost	Capacity
Static RAM	Fastest	Expensive	Smallest
Dynamic RAM	Slow	Cheap	Large
Hard disks	Slowest	Cheapest	Largest

- Fast memory is too expensive for most people to buy a lot of
- But dynamic memory has a much longer delay than other functional units in a datapath. If every `lw` or `sw` accessed dynamic memory, we'd have to either increase the cycle time or stall frequently
- Here are ***rough*** estimates of some current storage parameters

Storage	Delay	Cost/MB	Capacity
Static RAM	1-10 cycles	~$1	128KB-128MB
Dynamic RAM	100-200 cycles	~$0.005	256MB-512GB
Hard disks	10,000,000 cycles	~$0.00005	512GB-10TB

Review: Introducing Caches

- Wouldn't it be nice if we could find a balance between fast and cheap memory?
- We do this by introducing a cache, which is a small amount of fast, expensive memory
  - The cache goes between the processor and the slower, dynamic main memory
  - It keeps a copy of the most frequently used data from the main memory
- Memory access speed increases overall, because we've made the common case faster
  - Reads and writes to the most frequently used addresses will be serviced by the cache
  - We only need to access the slower main memory for less frequently used data



Review: The Principle of Locality

- It's usually difficult or impossible to figure out what data will be "most frequently accessed" before a program actually runs, which makes it hard to know what to store into the small, precious cache memory.
- But in practice, most programs exhibit *locality*, which the cache can take advantage of.
  - The principle of temporal locality says that if a program accesses one memory address, there is a good chance that it will access the same address again.
  - The principle of spatial locality says that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses.

Definitions: Hits and Misses

- A cache hit occurs if the cache contains the data that we're looking for. Hits are good, because the cache can return the data much faster than main memory.
- A cache miss occurs if the cache does not contain the requested data. This is bad, since the CPU must then wait for the slower main memory.

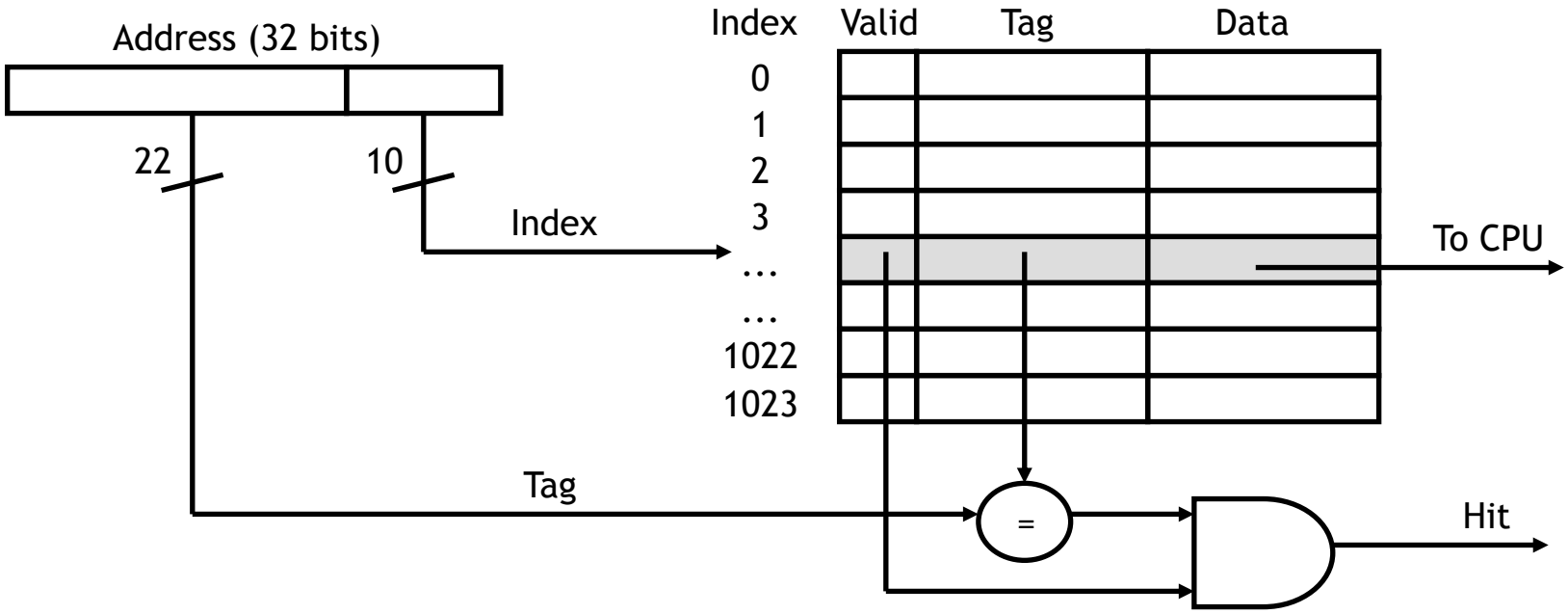
- There are two basic measurements of cache performance.
  - The hit rate is the percentage of memory accesses that are handled by the cache.
  - The miss rate (1 – hit rate) is the percentage of accesses that must be handled by the slower main RAM.
- Typical caches have a hit rate of 95% or higher, so in fact most memory accesses will be handled by the cache and will be dramatically faster.

Review: A Direct-Mapped Cache

- Each memory address maps to exactly one location in the cache
- Tag indicates which of the many memory addresses is currently mapped into cache location
- Valid bit indicates if the data stored in the cache entry has been loaded before

What Happens on a Cache Hit

- When the CPU tries to read from memory, the address will be sent to a cache controller.
  - The lowest *k* bits of the address will index a block in the cache.
  - If the block is valid and the tag matches the upper (*m* – *k*) bits of the *m*-bit address, then that data will be sent to the CPU.
- Here is a diagram of a 32-bit memory address and a 2^{10}-byte cache.

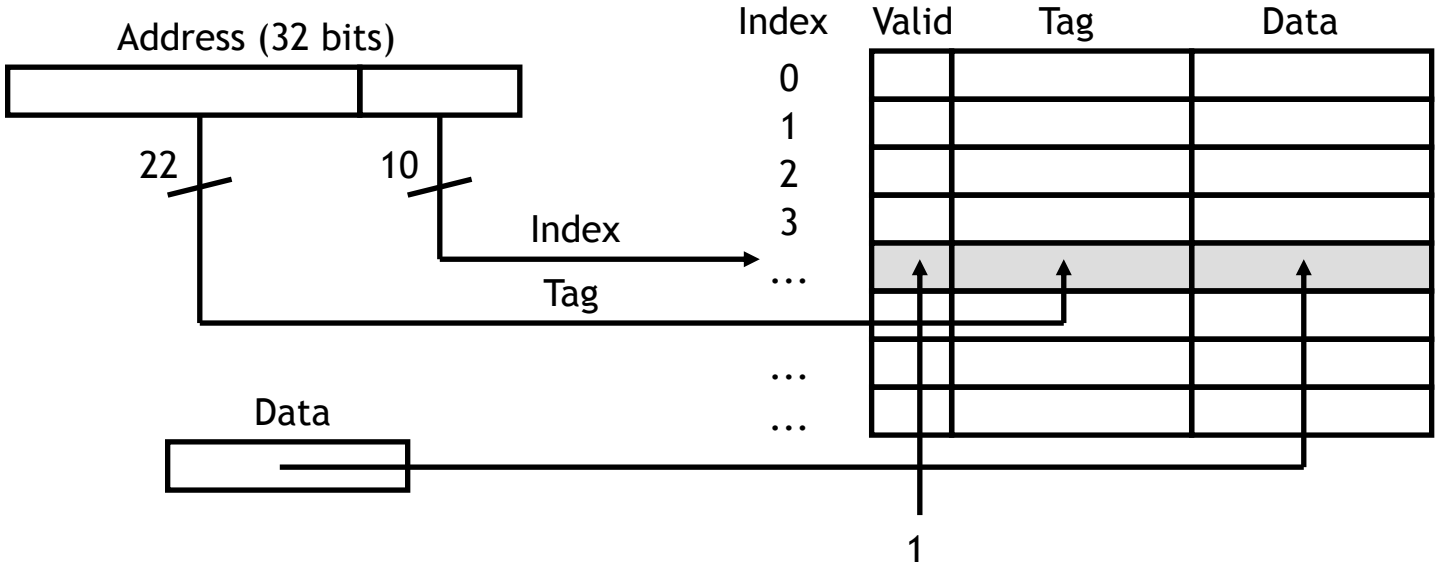


What Happens on a Cache Miss

- The delays that we've been assuming for memories (e.g., 2ns) are really assuming cache hits
  - If our CPU implementations accessed main memory directly, their cycle times would have to be much larger
  - Instead we assume that most memory accesses will be cache hits, which allows us to use a shorter cycle time
- However, a much slower main memory access is needed on a cache miss. The simplest thing to do is to stall the pipeline until the data from main memory can be fetched (and also copied into the cache)

Loading a Block into the Cache

- After data is read from main memory, putting a copy of that data into the cache is straightforward
  - The lowest *k* bits of the address specify a cache block
  - The upper (*m* – *k*) address bits are stored in the block's tag field
  - The data from main memory is stored in the block's data field
  - The valid bit is set to 1



What if the Cache Fills Up?

- Our third question was what to do if we run out of space in our cache, or if we need to reuse a block for a different memory address

- We answered this question implicitly on the last slide!
  - A miss causes a new block to be loaded into the cache, automatically overwriting any previously stored data.
  - This is a least recently used replacement policy, which assumes that older data is less likely to be requested than newer data.

How Big is the Cache?

- For a byte-addressable machine with 16-bit addresses with a cache with the following characteristics:
- How many blocks does the cache hold?

In-class Assessment!

Access Code: GEO

Note: sharing access code to those outside of classroom or using access code while outside of classroom is considered cheating

- How many bits of storage are required to build the cache (*e.g.*, for the data array, tags, etc.)?

How Big is the Cache?

- For a byte-addressable machine with 16-bit addresses with a cache with the following characteristics:
- It is direct-mapped (as discussed last time)
- Each block holds one byte
- The cache index is the four least significant bits

Two questions:

- How many blocks does the cache hold?

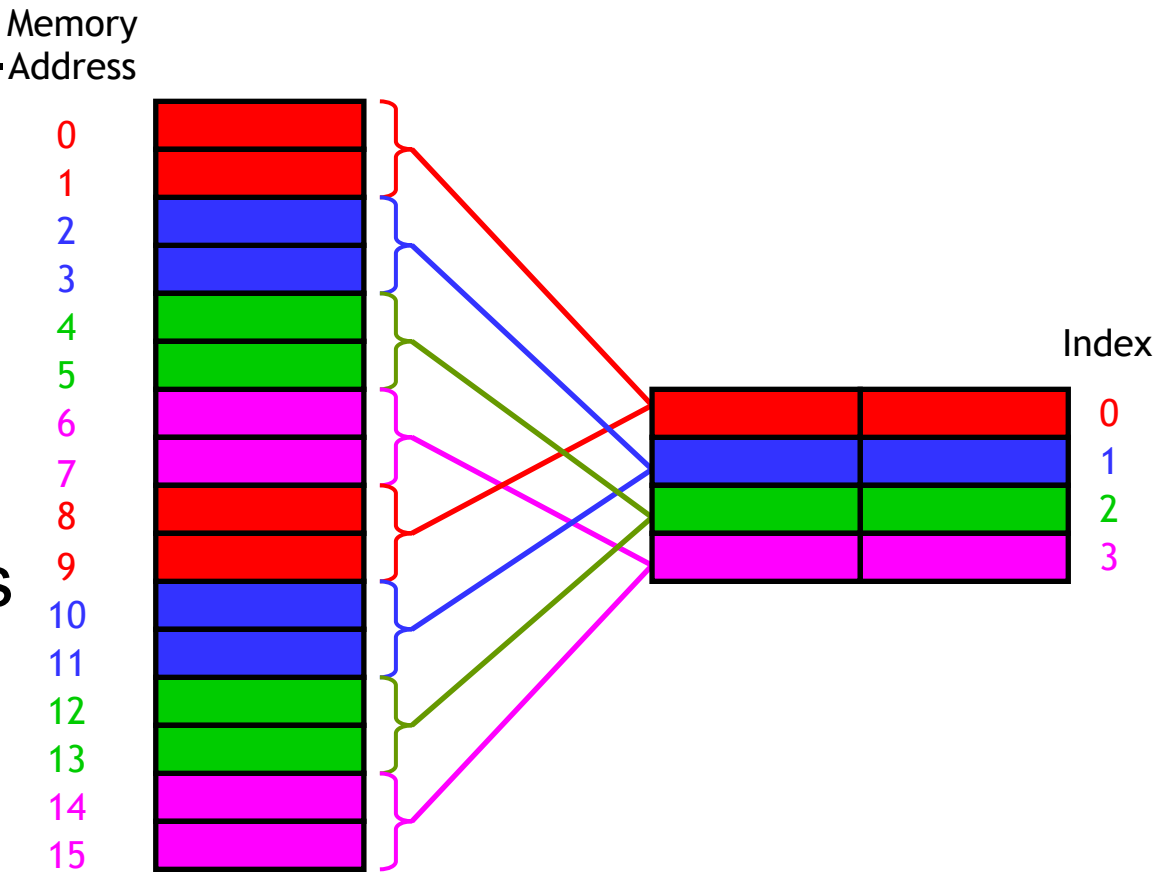
- How many bits of storage are required to build the cache (*e.g.*, for the data array, tags, etc.)?

Spatial Locality

- One-byte cache blocks don't take advantage of spatial locality, which predicts that an access to one address will be followed by an access to a nearby address
- What can we do?

Spatial Locality (cont.)

- What we can do is make the cache block size larger than one byte
- Here we use two-byte blocks, so we can load the cache with two bytes at a time
- If we read from address 12, the data in addresses 12 *and* 13 would both be copied to cache block 2

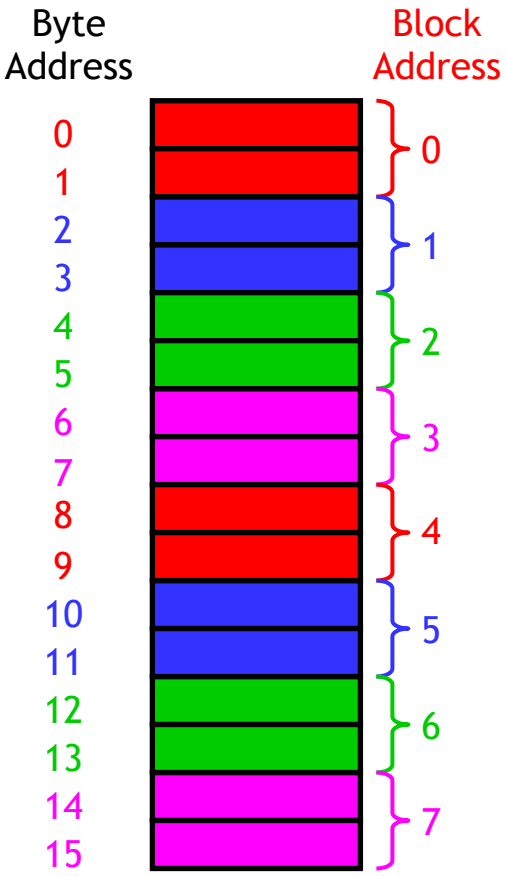


Block Addresses

- Now how can we figure out where data should be placed in the cache?
- It's time for block addresses! If the cache block size is 2^n bytes, we can conceptually split the main memory into 2^n-byte chunks too
- To determine the block address of a byte address *i*, you can do the integer division

$$i / 2^n$$

- Our example has two-byte cache blocks, so we can think of a 16-byte main memory as an "8-block" main memory instead
- For instance, memory addresses 12 and 13 both correspond to block address 6, since 12 / 2 = 6 and 13 / 2 = 6

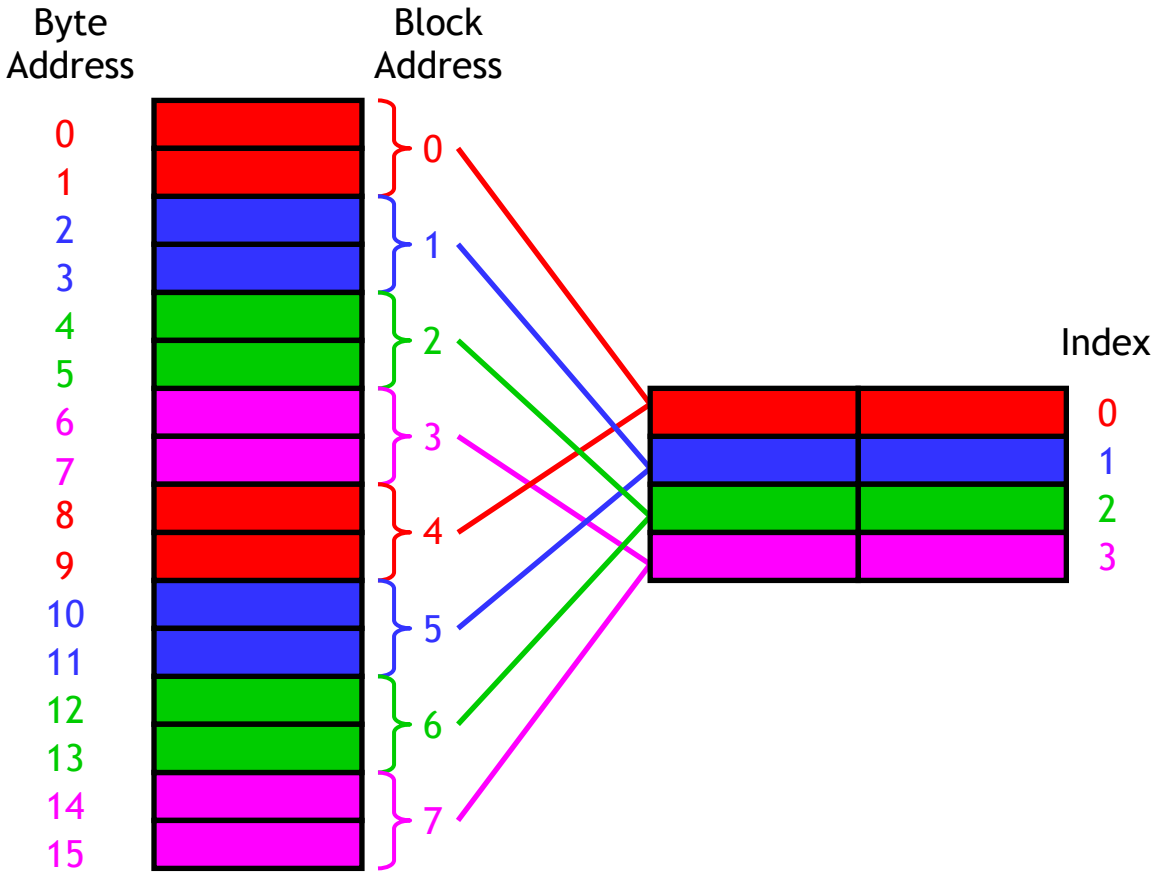


Cache Mapping

- Once you know the block address, you can map it to the cache as before: find the remainder when the block address is divided by the number of cache blocks

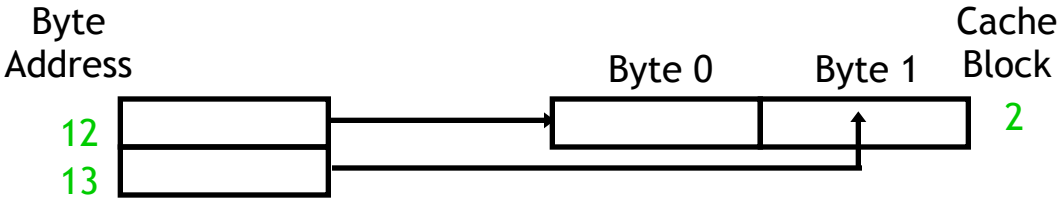
- In our example, memory block 6 belongs in cache block 2, since 6 mod 4 = 2

- This corresponds to placing data from memory *byte* addresses 12 and 13 into cache block 2



Data Placement within a Block

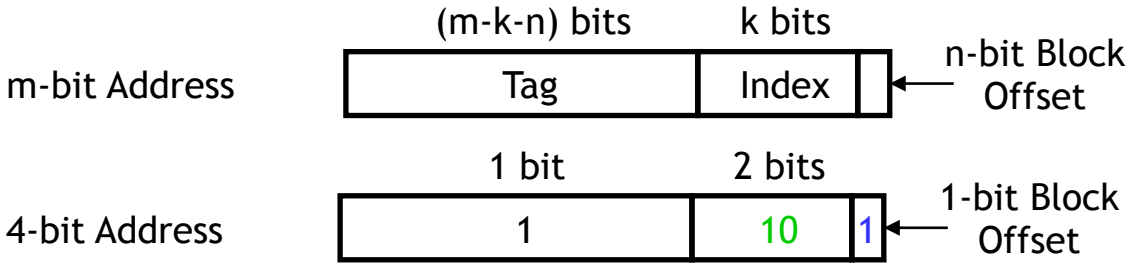
- When we access one byte of data in memory, we'll copy its entire *block* into the cache, to hopefully take advantage of spatial locality
- In our example, if a program reads from byte address 12 we'll load all of memory block 6 (both addresses 12 and 13) into cache block 2
- Note byte address 13 corresponds to the *same* memory block address! So a read from address 13 will also cause memory block 6 (addresses 12 and 13) to be loaded into cache block 2



- To make things simpler, byte *i* of a memory block is always stored in byte *i* of the corresponding cache block.

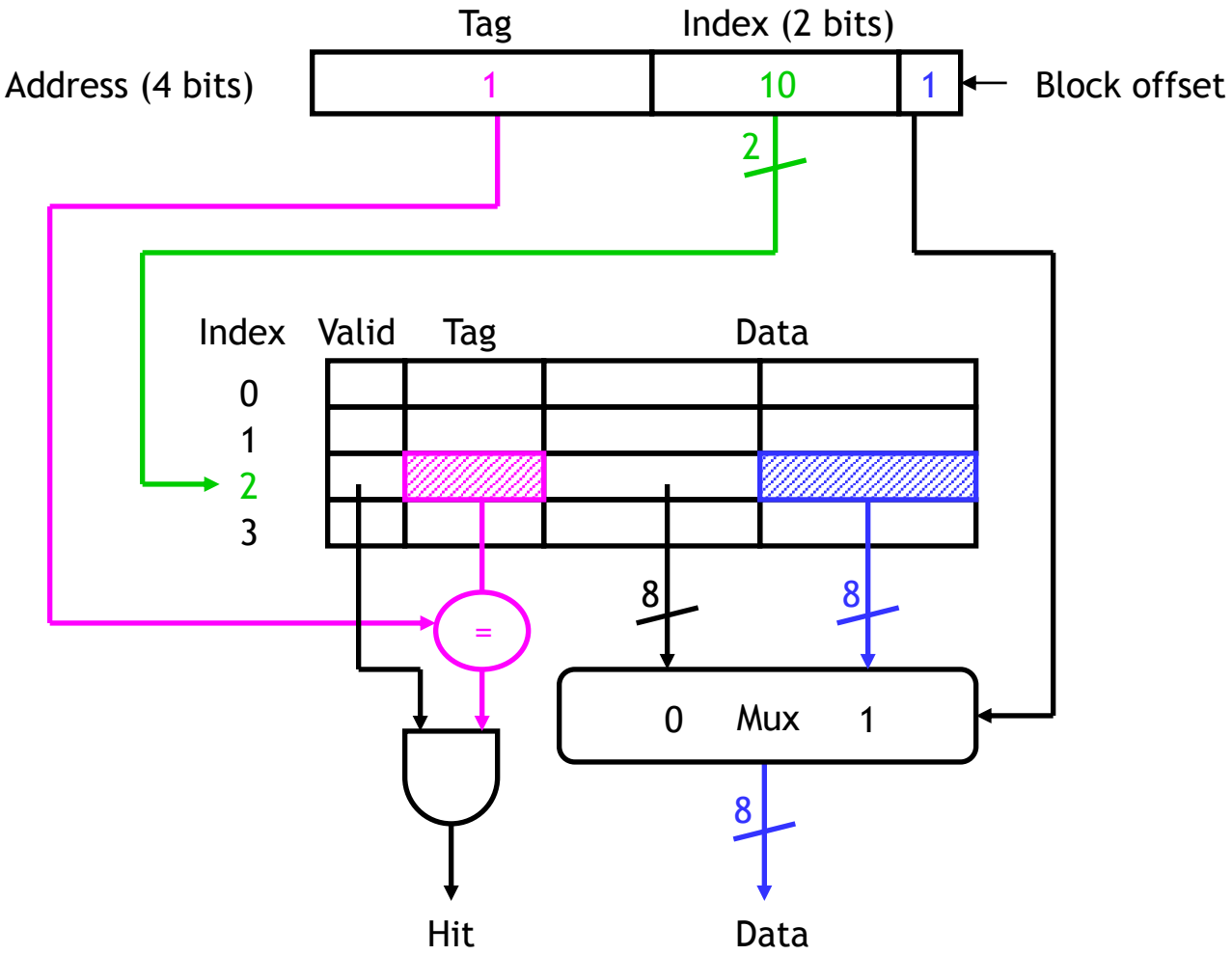
Locating Data in the Cache

- Let's say we have a cache with 2^k blocks, each containing 2^n bytes
- We can determine where a byte of data belongs in this cache by looking at its address in main memory
  - *k* bits of the address will select one of the 2^k cache blocks
  - The lowest *n* bits are now a block offset that decides which of the 2^n bytes in the cache block will store the data



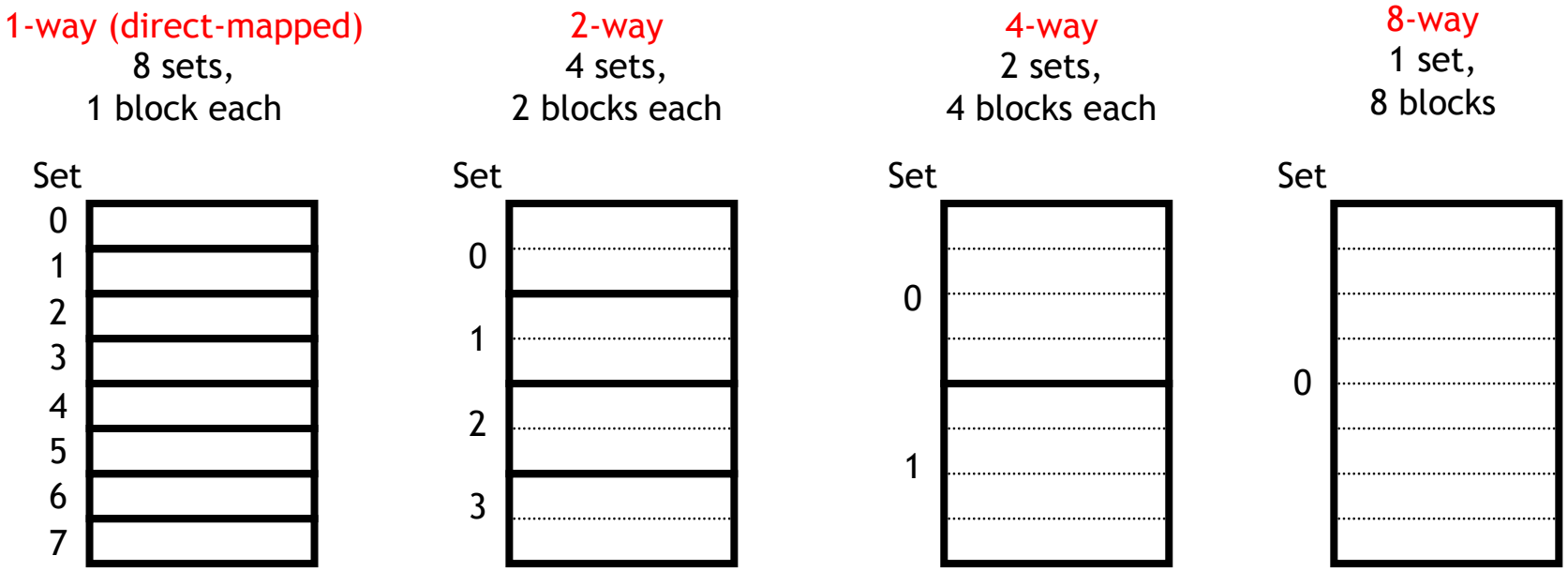
- Our example used a 2^2-block cache with 2^1 bytes per block. Thus, memory address 13 (1101) would be stored in byte 1 of cache block 2

A Picture


Tag
Index (2 bits)
Address (4 bits)
1
10
1
Block offset
2
Index
Valid
Tag
Data
0
1
2
3
=
8
8
0
Mux
1
8
Hit
Data

Preview: Set-Associative Cache

- By now you may have noticed the 1-way set associative cache is the same as a direct-mapped cache
- Similarly, if a cache has 2^k blocks, a 2^k-way set associative cache would be the same as a fully-associative cache



Acknowledgments

- These slides contain material developed and copyright by:
  - Joe Zambreno (Iowa State)
  - Akhilesh Tyagi (Iowa State)
  - David Patterson (UC Berkeley)
  - Mary Jane Irwin (Penn State)
  - Christos Kozyrakis (Stanford)
  - Onur Mutlu (Carnegie Mellon)
  - Krste Asanović (UC Berkeley)
  - Morgan Kaufmann