CprE 381: Computer Organization and Assembly Level Programming

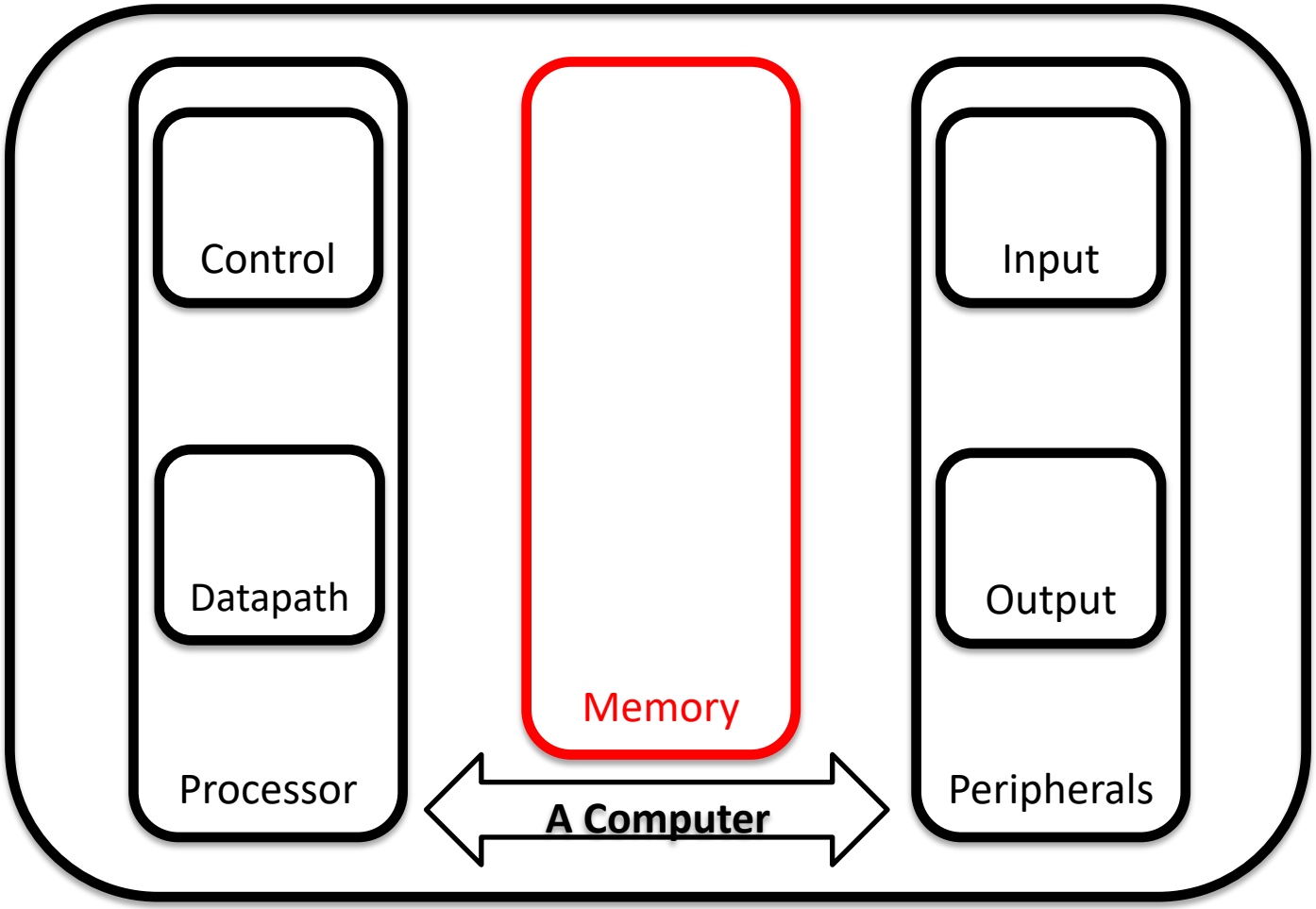
Cache Design

Henry Duwe
Electrical and Computer Engineering
Iowa State University

Administrative

- HW9 due Tonight!

Remember the System View!



Review: Small or Slow

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

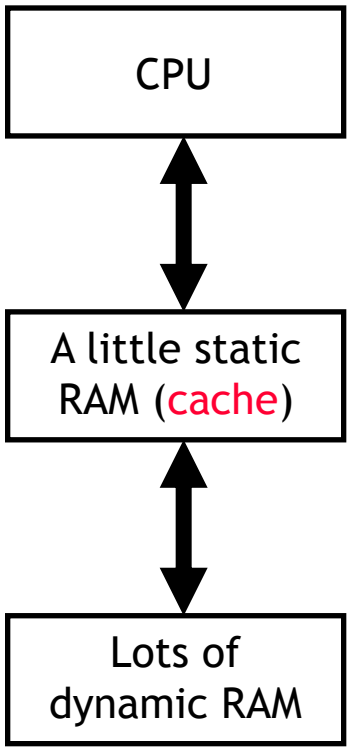
Storage	Speed	Cost	Capacity
Static RAM	Fastest	Expensive	Smallest
Dynamic RAM	Slow	Cheap	Large
Hard disks	Slowest	Cheapest	Largest

- 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-32GB
Hard disks	10,000,000 cycles	~$0.00005	256GB-4TB

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

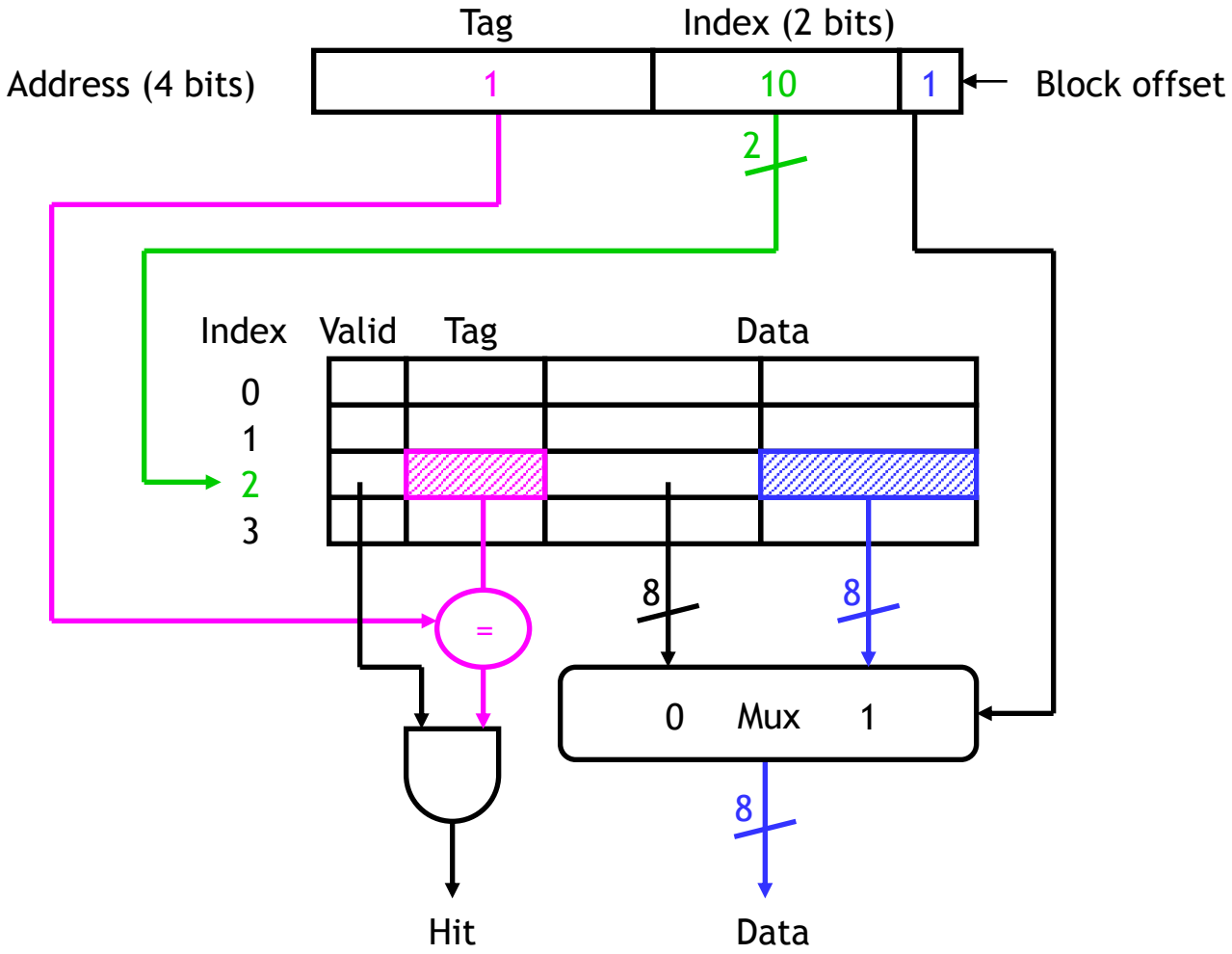
- A cache hit occurs if the cache contains the data that we're looking for. Hits are good, because the cache can return the data much faster than main memory.
- A cache miss occurs if the cache does not contain the requested data. This is bad, since the CPU must then wait for the slower main memory.

- 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

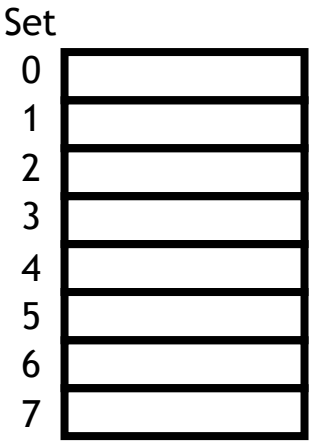
Review: Multi-Element Cache Block



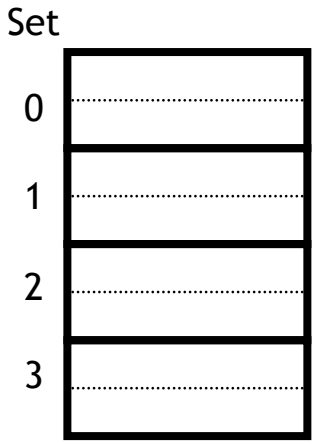
Review: Set-Associative Cache

- 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

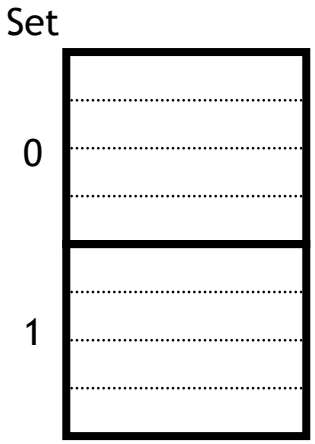
1-way (direct-mapped)
8 sets,
1 block each



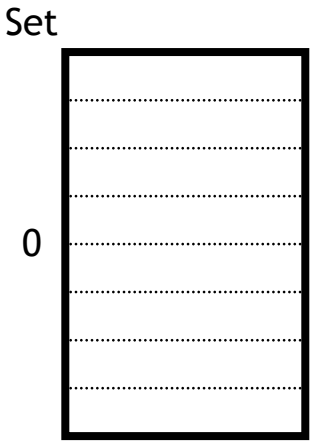
2-way
4 sets,
2 blocks each



4-way
2 sets,
4 blocks each



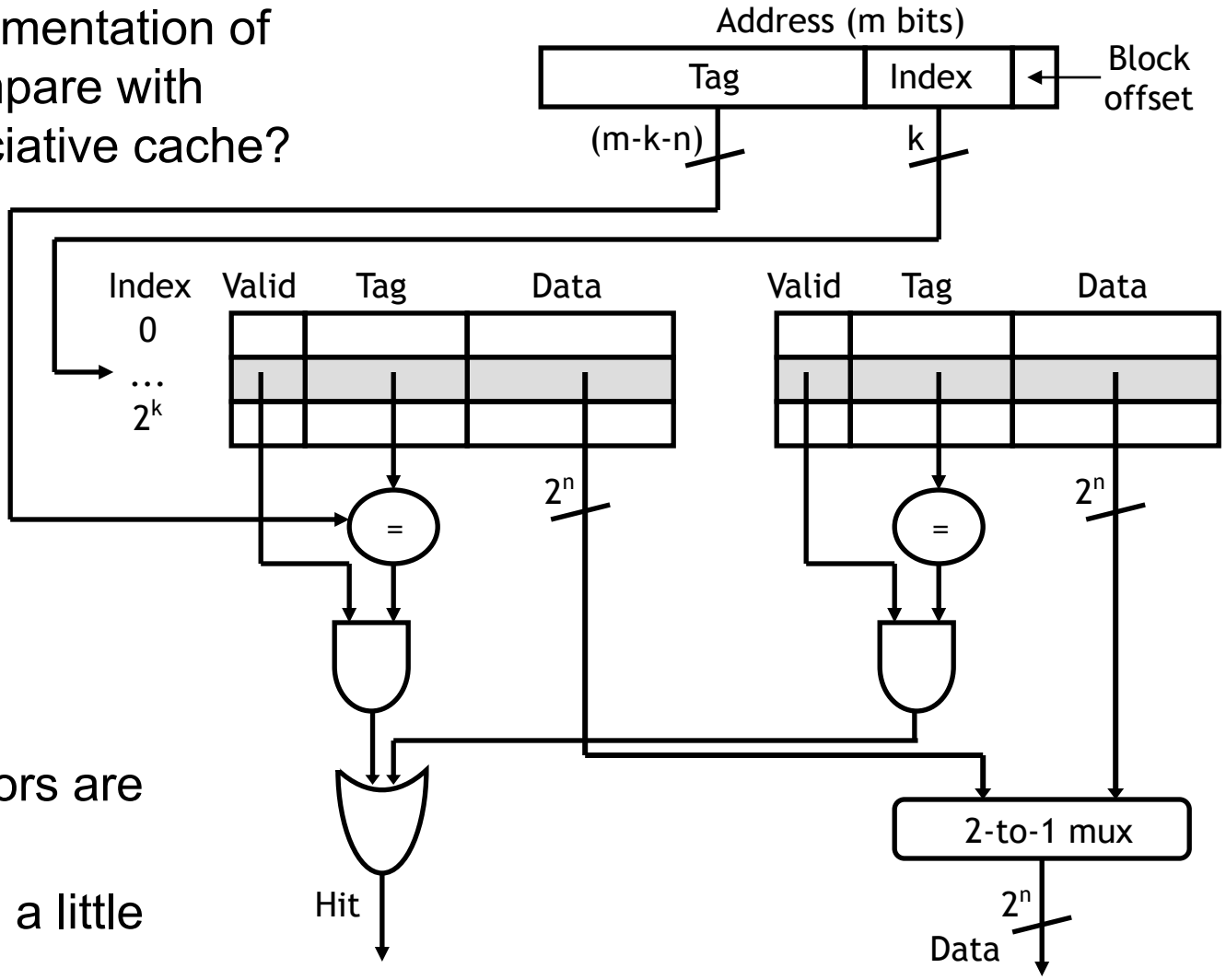
8-way
1 set,
8 blocks



fully associative

2-Way Set-Associative Implementation

- How does an implementation of a 2-way cache compare with that of a fully-associative cache?

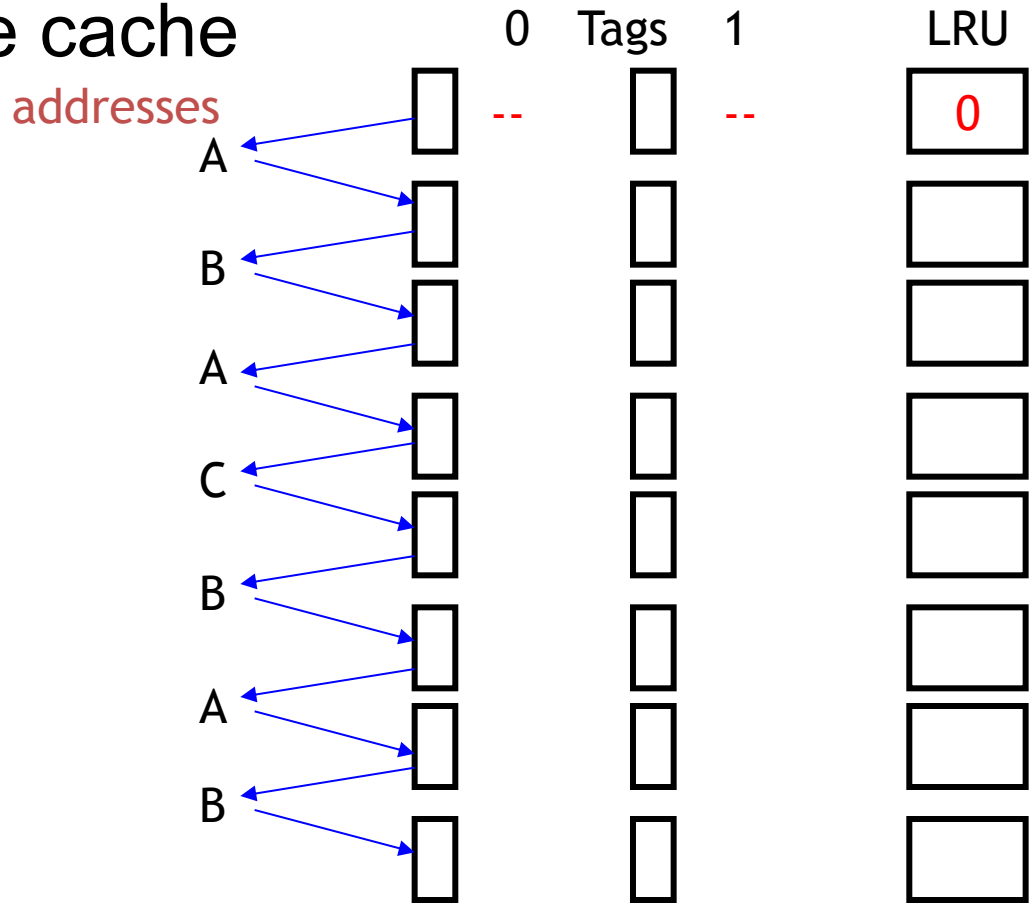


- Only two comparators are needed.
- The cache tags are a little shorter too.

LRU Example

- Assume a **fully-associative** cache with two blocks, which of the following memory references miss in the cache

  – Assume distinct addresses go to distinct blocks



Set-Associative Summary

- Larger block sizes can take advantage of spatial locality by loading data from not just one address, but also nearby addresses, into the cache
- Associative caches assign each memory address to a particular set within the cache, but not to any specific block within that set
  - Set sizes range from 1 (direct-mapped) to 2^k (fully associative)
  - Larger sets and higher associativity lead to fewer cache conflicts and lower miss rates, but they also increase the hardware cost
  - In practice, 2-way through 16-way set-associative caches strike a good balance between lower miss rates and higher costs
- Next, we'll talk more about measuring cache performance, and also discuss the issue of *writing* data to a cache

Four Important Questions

1. When we copy a block of data from main memory to the cache, where exactly should we put it?
2. How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?
3. Eventually, the small cache memory might fill up. To load a new block from main RAM, we'd have to replace one of the existing blocks in the cache... which one?
4. How can *write* operations be handled by the memory system?

- Previously we answered the first 3. Now, we consider the 4th.

Writing to a Cache

- Writing to a cache raises several additional issues
- First, let's assume that the address we want to write to is already loaded in the cache. We'll assume a simple direct-mapped cache

Index	V	Tag	Data
…			
110	1	11010	42803
…			

Address	Data
…	
1101 0110	42803
…	

- If we write a new value to that address, we can store the new data in the cache, and avoid an expensive main memory access

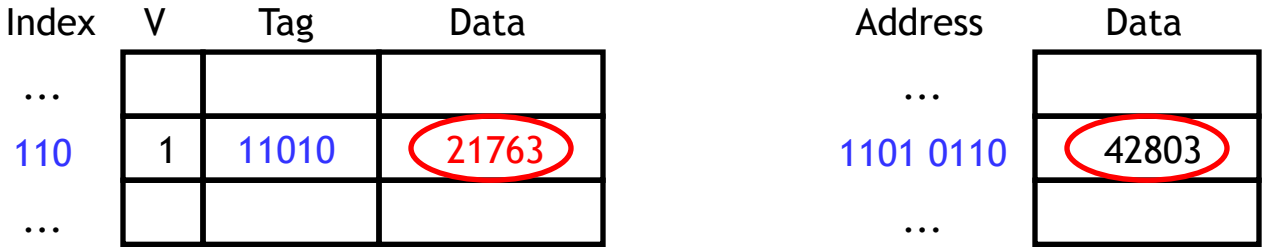
Mem[214] = 21763

Index	V	Tag	Data
…			
110	1	11010	21763
…			

Address	Data
…	
1101 0110	42803
…	

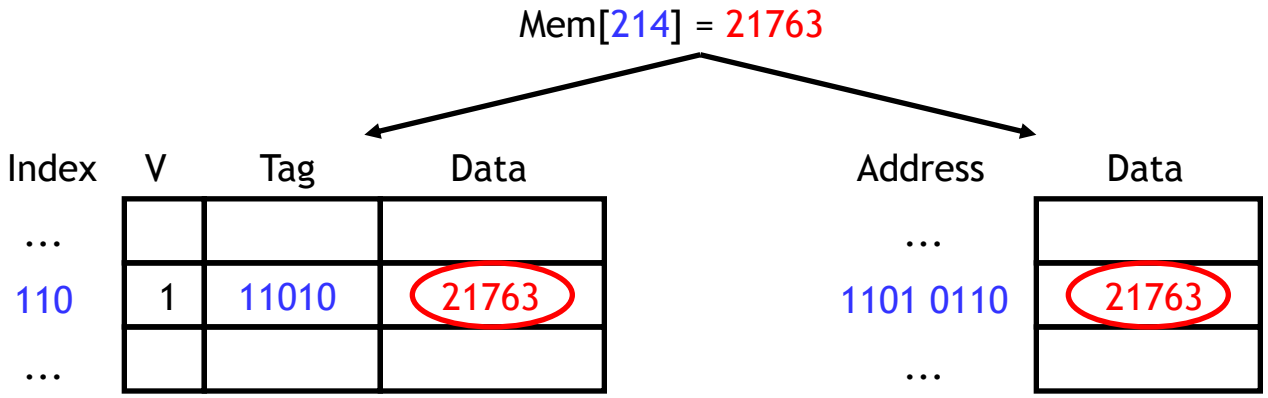
Inconsistent Memory

- But now the cache and memory contain different, inconsistent data!
- How can we ensure that subsequent loads will return the right value?
- This is also problematic if other devices are sharing the main memory, as in a multiprocessor system



Write-Through Caches

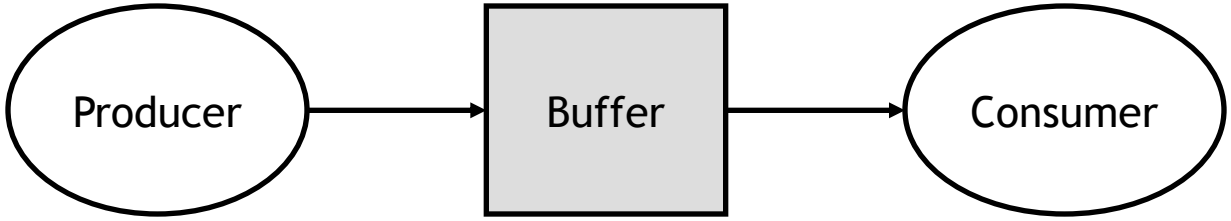
- A write-through cache solves the inconsistency problem by forcing all writes to update both the cache *and* the main memory



- This is simple to implement and keeps the cache and memory consistent
- Why is this not so good?

Write Buffers

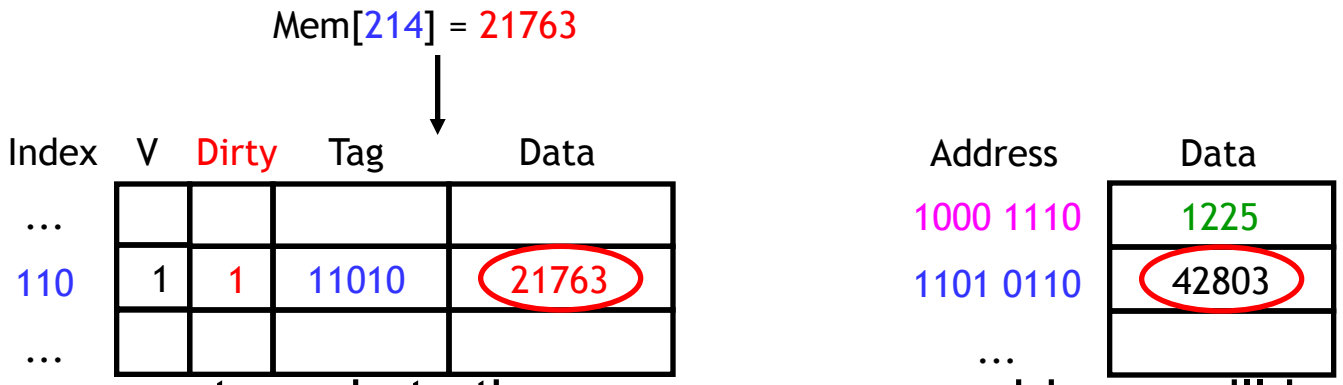
- Write-through caches can result in slow writes, so processors typically include a write buffer, which queues pending writes to main memory and permits the CPU to continue



- Buffers are commonly used when two devices run at different speeds
  - If a producer temporarily generates bursts of data too quickly for a consumer to handle, the extra data is stored in a buffer and the producer can continue on with other tasks, without waiting for the consumer
  - Conversely, if the producer slows down, the consumer can continue running at full speed as long as there is excess data in the buffer
- For us, the producer is the CPU and the consumer is the main memory

Write-Back Caches

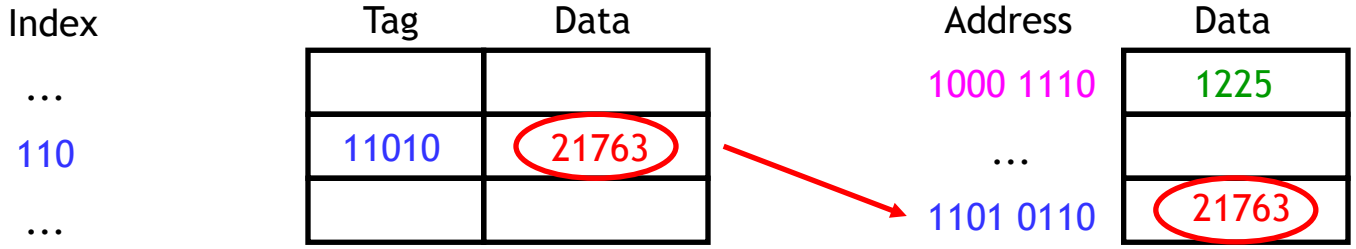
- In a write-back cache, the memory is not updated until the cache block needs to be replaced (*e.g.*, when loading data into a full cache set)
- For example, we might write some data to the cache at first, leaving it inconsistent with the main memory as shown before
  - The cache block is marked "dirty" to indicate this inconsistency



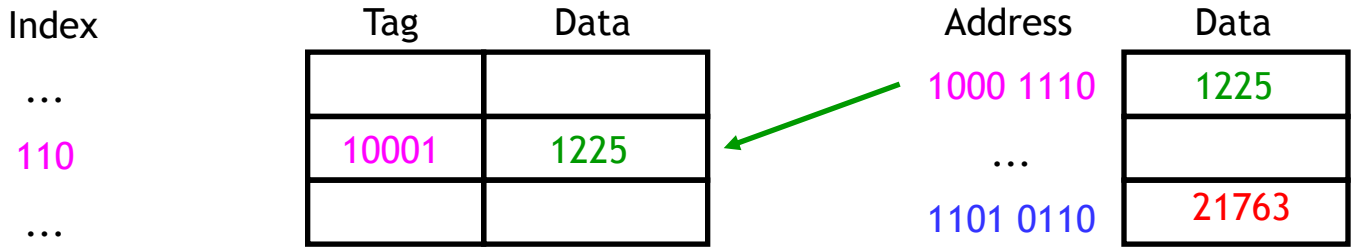
- Subsequent reads to the same memory address will be serviced by the cache, which contains the correct, updated data

Finishing the Write Back

- We don't need to store the new value back to main memory until the cache block gets replaced
- For example, on a read from Mem[142], which maps to the same cache block, the modified cache contents will first be written to main memory



- Only then can the cache block be replaced with data from address 142



Write-Back Cache Discussion

- Each block in a write-back cache needs a dirty bit to indicate whether or not it must be saved to main memory before being replaced—otherwise we might perform unnecessary writebacks
- Notice the penalty for the main memory access will not be applied until the execution of some *subsequent* instruction following the write
  - In our example, the write to Mem[214] affected only the cache.
  - But the load from Mem[142] resulted in *two* memory accesses: one to save data to address 214, and one to load data from address 142
    - The write can be "buffered" as was shown in write-through
- The advantage of write-back caches is that not all write operations need to access main memory, as with write-through caches
  - If a single address is frequently written to, then it doesn't pay to keep writing that data through to main memory
  - If several bytes within the same cache block are modified, they will only force one memory write operation at write-back time

Write Misses

- A second scenario is if we try to write to an address that is not already contained in the cache; this is called a write miss
- Let's say we want to store 21763 into Mem[1101 0110] but we find that address is not currently in the cache

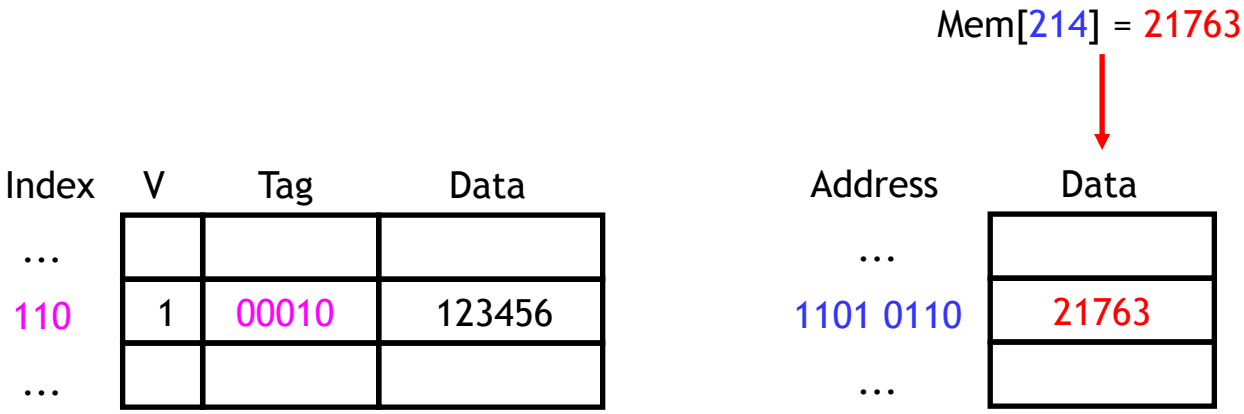
Index	V	Tag	Data
…			
110	1	00010	123456
…			

Address	Data
…	
1101 0110	6378
…	

- When we update Mem[1101 0110], should we *also* load it into the cache?

Write Around Caches (Write-no-Allocate)

- With a write around policy, the write operation goes directly to main memory *without* affecting the cache

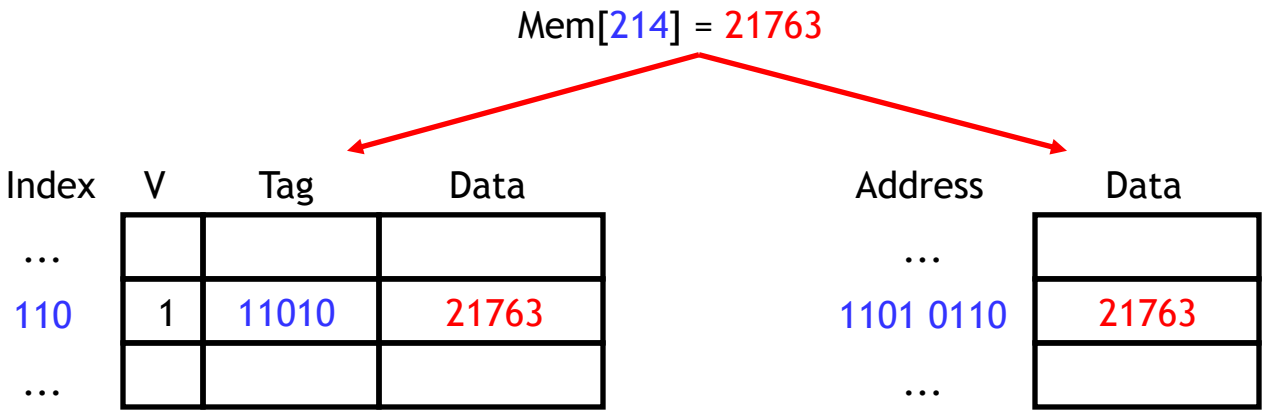


- This is good when data is written but not immediately used again, in which case there's no point to load it into the cache yet

```
for (int i = 0; i < SIZE; i++)
   a[i] = i;
```

Allocate on Write

- An allocate on write strategy would instead load the newly written data into the cache



- If that data is needed again soon, it will be available in the cache

Which is it?

- Given the following trace of accesses, can

In-class Assessment!

Access Code: !$Wr

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

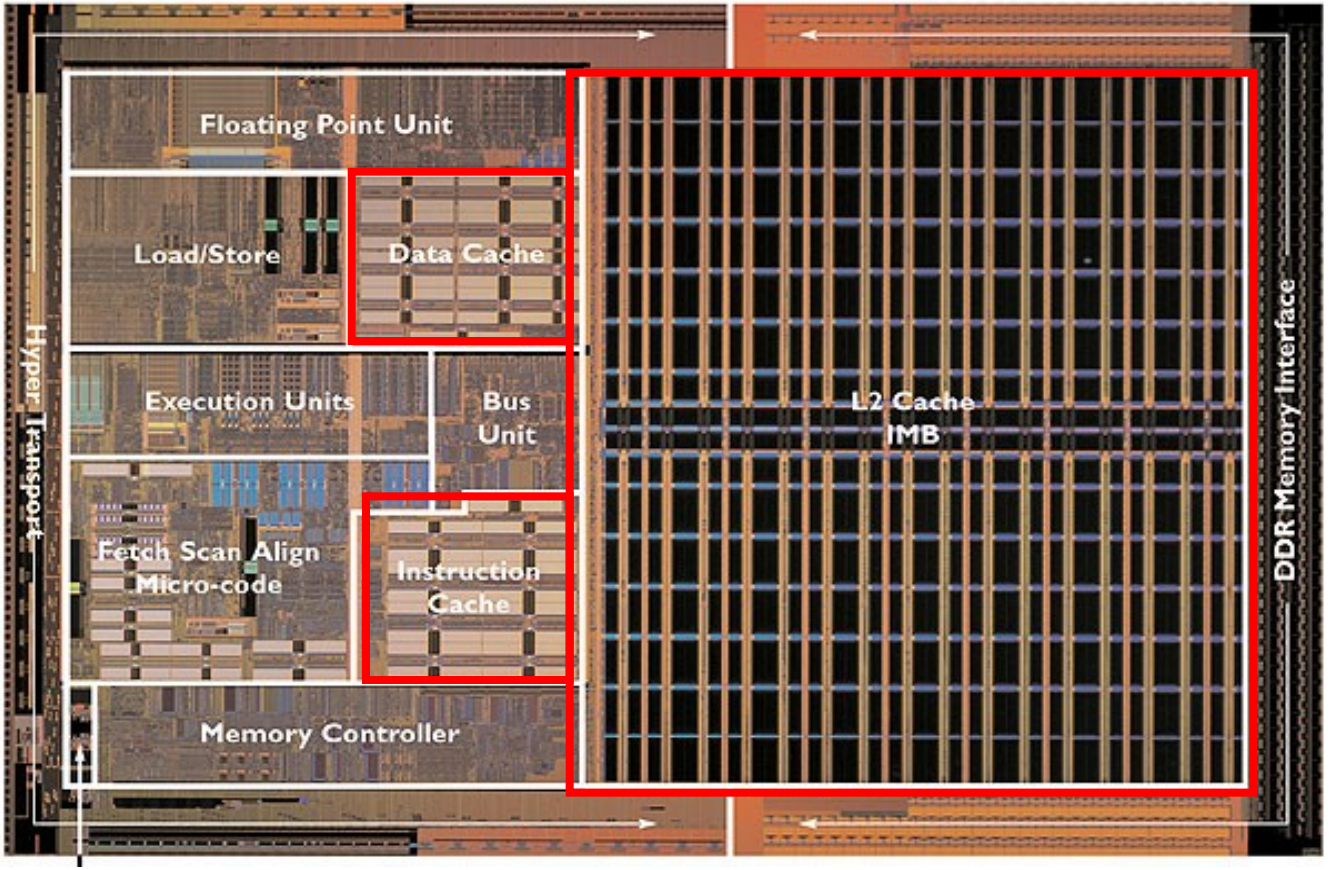
Miss	Load A
Miss	Store B
Hit	Store A
Hit	Load A
Miss	Load B
Hit	Load B
Hit	Load A

Which is it?

- Given the following trace of accesses, can you determine whether the cache is write-allocate or write-no-allocate?
  - Assume A and B are distinct and can be in the cache simultaneously

Miss	Load A
Miss	Store B
Hit	Store A
Hit	Load A
Miss	Load B
Hit	Load B
Hit	Load A

Preview: A Real Design (AMD Opteron)



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