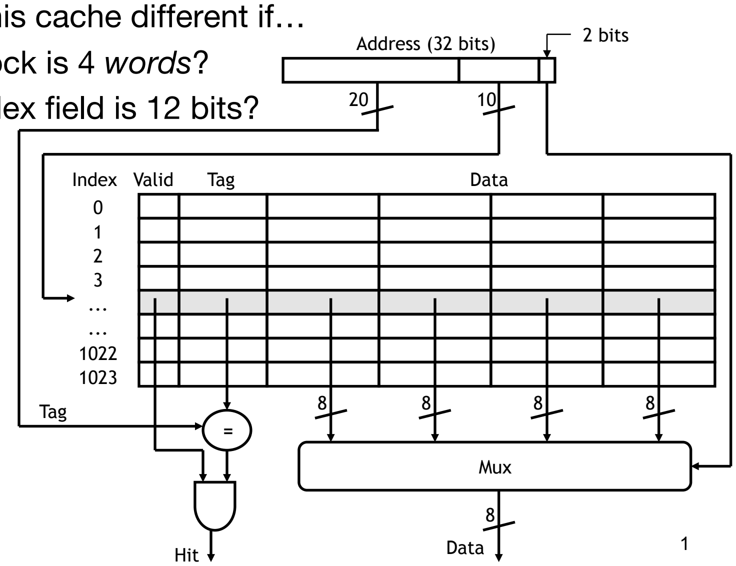
Review

How is this cache different if…

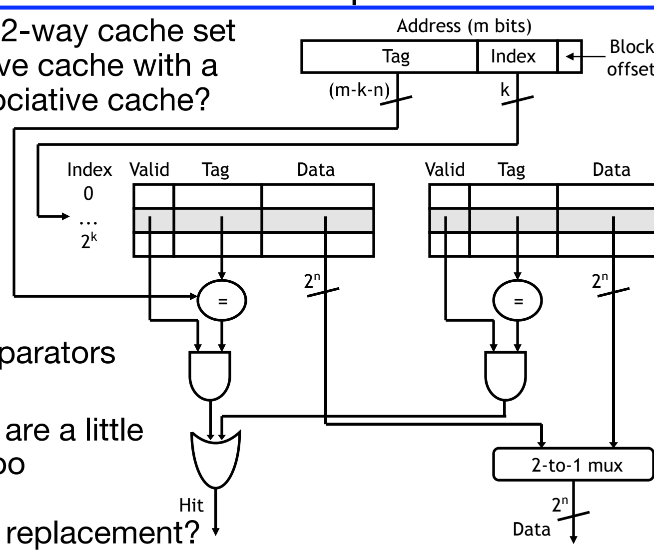
- the block is 4 *words*?
- the index field is 12 bits?



1

2-way set associative implementation

Compare a 2-way cache set associative cache with a fully-associative cache?



Only 2 comparators needed

Cache tags are a little shorter too

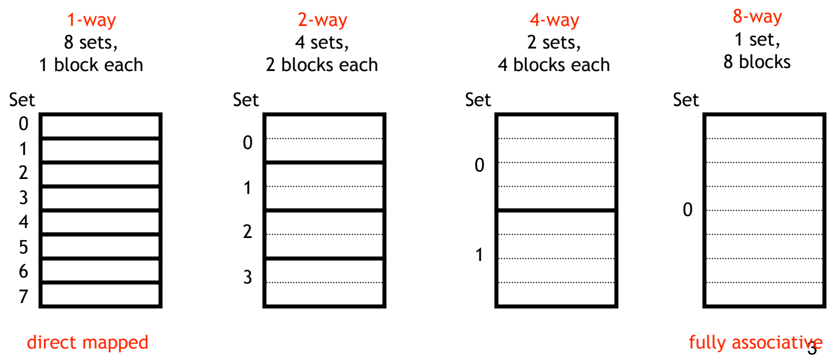
… deciding replacement?

2

Set associative caches are a general idea

By now you 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



3

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

4

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?

- Previous lectures answered the first 3. Today, 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

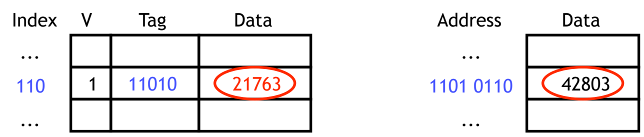
But now the cache and memory contain different, inconsistent data!

First Rule of Data Management: No inconsistent data

Second Rule: Don't Even Think About Violating 1st Rule

How can we ensure that subsequent loads will return the right value?

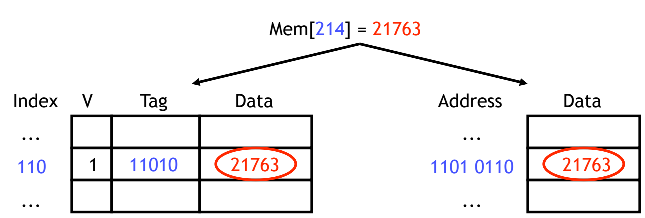
This is also problematic if other devices are sharing the main memory, as in I/O or a multiprocessor system



7

Write-through caches

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



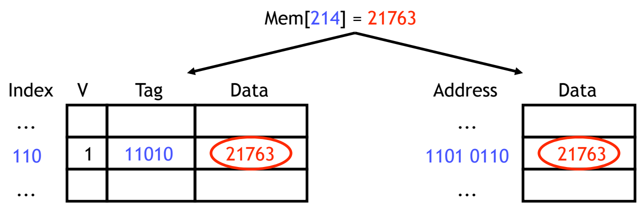
This is simple to implement and keeps the cache and memory consistent

Why might it be not so good?

8

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.

The bad thing is that forcing every write to go to main memory, we use up bandwidth between the cache and the memory.

9

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 generates 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

10

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 …

Notice that the write buffer allows the CPU to continue before the write is complete, but write-through has the problem: It uses memory bandwidth

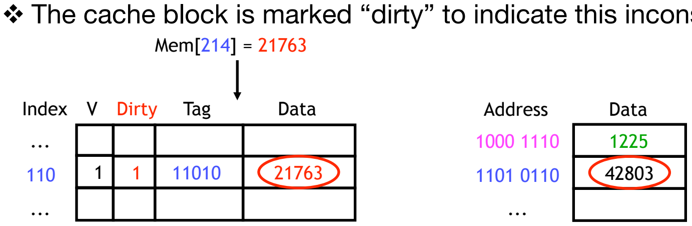
```
int sum_array_rows(int a[M][N])
{
  int i, j, sum = 0;

  for (i = 0; i < M; i++)
     for (j = 0; j < N; j++)
        sum += a[i][j];
  return sum;
}
```


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

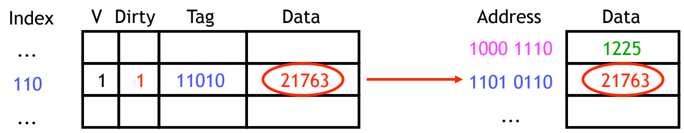


Subsequent reads to the same memory address will be serviced by the cache, which contains the correct, updated data

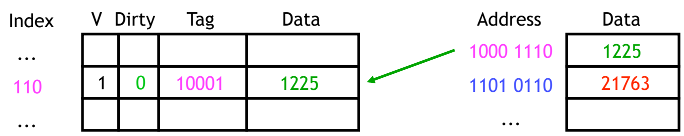
Finishing the write back

We don’t need to store the new value back to main memory unless 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



13

Write-back cache discussion

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

14

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

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.

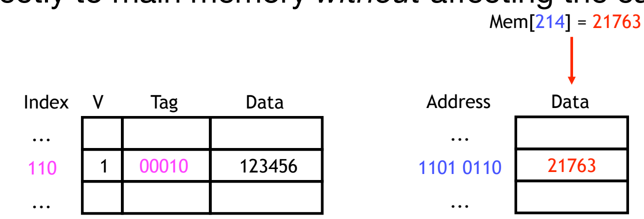
Index	V	Tag	Data
…			
110	1	00010	123456
…			

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

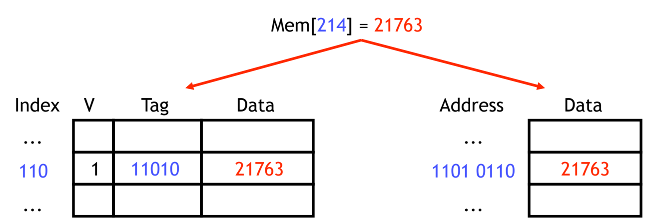


This is good when data is written but not immediately used again, in which case there's no point to load it into the cache yet

```
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 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

19

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

On a write-allocate cache this would be a hit (the arrow points to the "Miss Load B" line)

Answer: Write-no-allocate

20

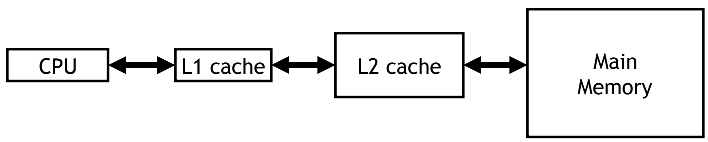
First Observations

Split Instruction/Data caches:

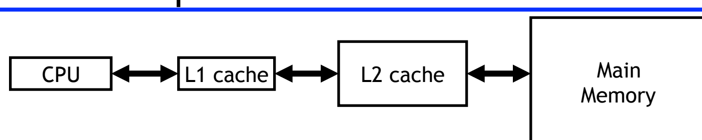
- Pro: No structural hazard between IF & MEM stages
  - A single-ported unified cache stalls fetch during load or store
- Con: Static partitioning of cache between instructions & data
  - Bad if working sets unequal: *e.g.,* code/**DATA** or **CODE**/data

Cache Hierarchies:

- Trade-off between access time & hit rate
  - L1 cache can focus on fast access time (okay hit rate)
  - L2 cache can focus on good hit rate (okay access time)
- Such hierarchical design is another “big idea”



Opteron Vital Statistics



L1 Caches: Instruction & Data

- 64 kB
- 64 byte blocks
- 2-way set associative
- 2 cycle access time

L2 Cache:

- 1 MB
- 64 byte blocks
- 4-way set associative
- 16 cycle access time (total, not just miss penalty)

Memory

- 200+ cycle access time