Lecture 16: Caches (3)

(CPEG323: Intro. to Computer System Engineering)

Question

- Assuming a direct-mapped, write-through cache with 16 KB of data and 4-word blocks, how divide a 32-bit byte address to access a cache?

A red) Tag <14 bits> | Index <14 bits> | Block Offset <4 bits>

B orange) Tag <16 bits> | Index <14 bits> | Block Offset <2 bits>

C green) Tag <18 bits> | Index <10 bits> | Block Offset <4 bits>

D yellow) Tag <20 bits> | Index <10 bits> | Block Offset <2 bits>

E pink) Valid <1> | Tag <14 bits> | Index <14 bits> | Block Offset <4 bits>

F blue) Valid <1> | Dirty <1> |Tag <14 bits> | Index <14 bits> | Block Offset <4 bits>

G purple) Valid <1> | Tag <18 bits> | Index <10 bits> | Block Offset <4 bits>

H teal) Valid <1> | Dirty <1> |Tag <18 bits> | Index <10 bits> | Block Offset <4 bits>

Answer

- Assuming a direct-mapped, write-through cache with 16 KB of data and 4-word blocks, how divide a 32-bit byte address to access a cache?

A red) Tag <14 bits> | Index <14 bits> | Block Offset <4 bits>

B orange) Tag <16 bits> | Index <14 bits> | Block Offset <2 bits>

C green) Tag <18 bits> | Index <10 bits> | Block Offset <4 bits>

D yellow) Tag <20 bits> | Index <10 bits> | Block Offset <2 bits>

E pink) Valid <1> | Tag <14 bits> | Index <14 bits> | Block Offset <4 bits>

F blue) Valid <1> | Dirty <1> |Tag <14 bits> | Index <14 bits> | Block Offset <4 bits>

G purple) Valid <1> | Tag <18 bits> | Index <10 bits> | Block Offset <4 bits>

H teal) Valid <1> | Dirty <1> |Tag <18 bits> | Index <10 bits> | Block Offset <4 bits>

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 (but inconsistent)

Mem[1101 0110] = 21763

Index	V	Tag	Data
...			
110	1	11010	21763
...			

address	Data
...	
1101 0110	42803
...	

Write-through caches

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

Mem[1101 0110] = 21763

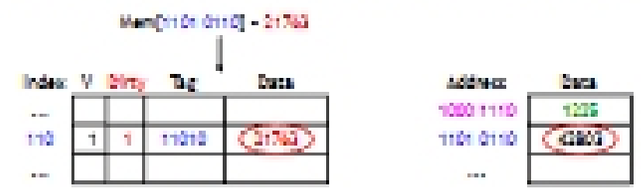
Index	V	Tag	Data
...			
110	1	11010	21763
...			

address	Data
...	
1101 0110	21763
...	

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

Write-back caches

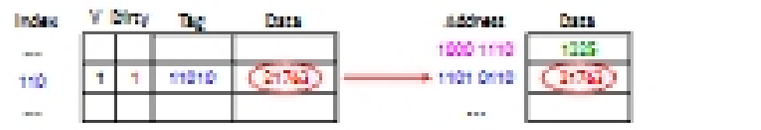
- In a write-back cache, the memory is not updated until the cache block needs to be replaced (e.g., when loading data into a full cache set)
- For example, we might write some data to the cache at first, leaving it inconsistent with the main memory as shown before
  - The cache block is marked "dirty"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



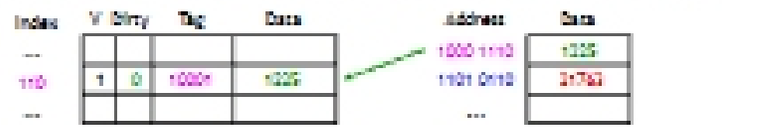
- Subsequent reads to the same memory address will be serviced by the cache, which contains the correct, updated data

Finishing the write back

- We don't need to store the new value back to main memory unless the cache block gets replaced
- E.g. on a read from Mem[1000 1110], 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



Question

- How many total *bits* are required for that cache?
  (Round to nearest Kbits)
  - Direct-mapped, write-through, 16 KBytes of data, 4-word (16 Byte) blocks, 32-bit address
  - Tag <18 bits> | Index <10 bits> | Block Offset <4 bits>

A red)	16 Kbits	E pink)	139 Kbits
B orange)	18 Kbits	F blue)	146 Kbits
C green)	128 Kbits	G purple)	147 Kbits
D yellow)	130 Kbits	H teal)	148 Kbits

Answer

- How many total *bits* are required for that cache?
  (Round to nearest Kbits)
  - Direct-mapped, write-through, 16 KBytes of data, 4-word (16 Byte) blocks, 32-bit address
  - Tag <18 bits> | Index <10 bits> | Block Offset <4 bits>

A red)	16 Kbits	E pink)	139 Kbits
B orange)	18 Kbits	F blue)	146 Kbits
C green)	128 Kbits	G purple)	147 Kbits
D yellow)	130 Kbits	H teal)	148 Kbits

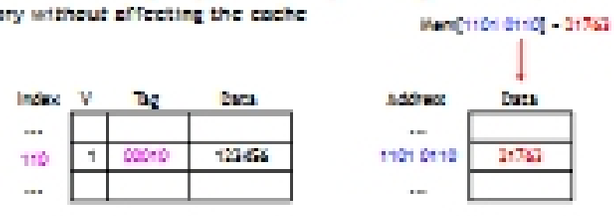
Write misses

- 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

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

Write around caches (a.k.a. write-no-allocate)

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

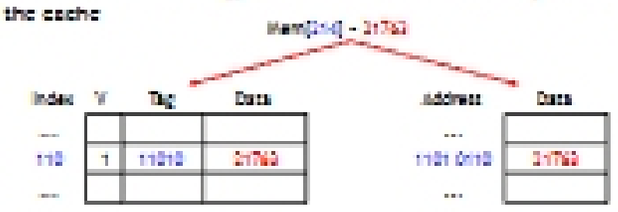


- This is good when data is written but not immediately used again, in which case there's no point to load it into the cache yet

```
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?

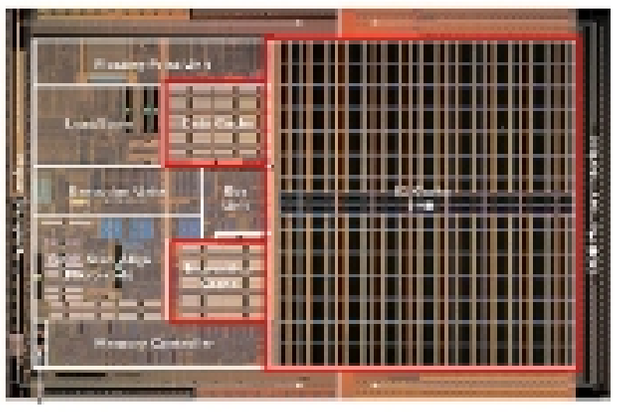
- Given the following trace of accesses, can you determine whether the cache is write-allocate or write-no-allocate?
  - Assume A and B are distinct, and can be in the cache simultaneously.

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

Answer: Write-no-allocate

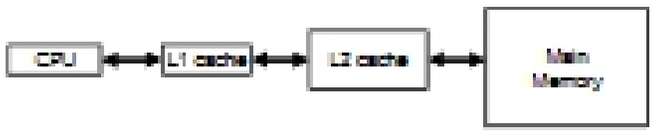
On a write-allocate cache this would be a hit

Real Designs



First Observations

- Cache Hierarchies:
  - Trade-off between access time & hit rate
    - L1 cache can focus on fast access time (with okay hit rate)
    - L2 cache can focus on good hit rate (with okay access time)
  - Such hierarchical design is another "big idea"
  - We saw this in section.



Comparing cache organizations

- Like many architectural features, caches are evaluated experimentally.
  - As always, performance depends on the actual instruction mix, since different programs will have different memory access patterns.
  - Simulating or executing real applications is the most accurate way to measure performance characteristics.
- The graphs on the next few slides illustrate the simulated miss rates for several different cache designs.
  - Again lower miss rates are generally better, but remember that the miss rate is just one component of average memory access time and execution time.

Associativity tradeoffs and miss rates

- Higher associativity means more complex hardware
- But a highly-associative cache will also exhibit a lower miss rate
  - Each set has more blocks, so there's less chance of a conflict between two addresses which both belong to the same set
  - Overall, this will reduce AMAT and memory stall cycles
- Figure from the textbook shows the miss rates decreasing as the associativity increases