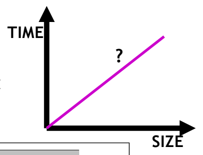
Lecture 14: Caches

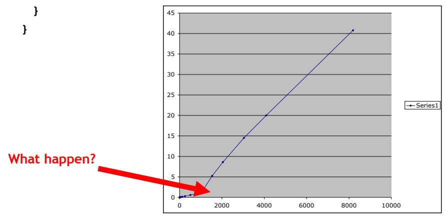
(CPEG323: Intro. to Computer System Engineering)

How will execution time grow with SIZE?

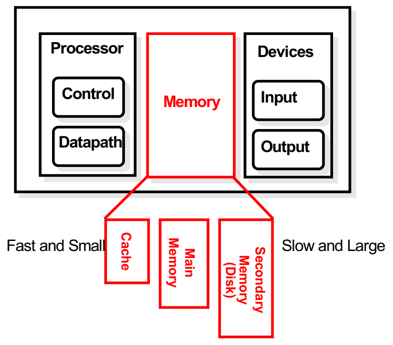
```
int array[SIZE];
int sum = 0;

for (int i = 0 ; i < 200000 ; ++ i) {
  for (int j = 0 ; j < SIZE ; ++ j) {
      sum += array[j];
  }
}
```



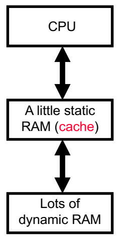


Components of a Computer



Introducing caches

- Caches help strike a balance
- A cache is a small amount of fast, expensive memory.
  - It 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.
- Principle used elsewhere: Networks, OS, Search…



The principle of locality

- Usually difficult or impossible to figure out what data will be "most frequently accessed" before a program actually runs
  - hard to know what to store into the small, precious cache memory.
- 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.

Locality in programs

- **Temporal locality:**
  - Loops are excellent examples
    - The loop body will be executed many times.
    - The computer will need to access those same few locations of the instruction memory repeatedly.
  - For example:

```
Loop:  lw    $t0, 0($s1)
       add   $t0, $t0, $s2
       sw    $t0, 0($s1)
       addi  $s1, $s1, -4
       bne   $s1, $0, Loop
```

- **Spatial locality:**
  - Nearly every program, because instructions are usually executed in sequence.

Locality in data

- **Temporal locality**
  - Programs often access the same variables over and over, especially within loops.

```
sum = 0;
for (i = 0; i < MAX; i++)
    sum = sum + f(i);
```

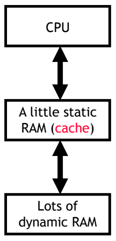
- **Spatial locality**
  - When reading location A from main memory, a copy of that data is placed in the cache but also copy A+1, A+2, …
    - Useful for arrays, records, multiple local variables

```
sum = 0;
for (i = 0; i < MAX; i++)
    sum = sum + a[i];
```

```
employee.name = "Homer Simpson";
employee.boss = "Mr. Burns";
employee.age = 45;
```

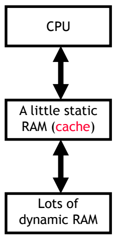
How caches take advantage of temporal locality

- The first time the processor reads from an address in main memory, a copy of that data is also stored in the cache.
  - The next time that same address is read, we can use the copy of the data in the cache *instead* of accessing the slower dynamic memory.
  - So the first read is a little slower than before since it goes through both main memory and the cache, but subsequent reads are much faster.
- This takes advantage of temporal locality—commonly accessed data is stored in the faster cache memory.



How caches take advantage of spatial locality

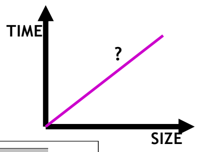
- When the CPU reads location *i* from main memory, a copy of that data is placed in the cache.
- But instead of just copying the contents of location *i*, we can copy *several* values into the cache at once, such as the four bytes from locations *i* through *i* + 3.
  - If the CPU later does need to read from locations *i* + 1, *i* + 2 or *i* + 3, it can access that data from the cache and not the slower main memory.
  - For example, instead of reading just one array element at a time, the cache might actually be loading four array elements at once.
- Again, the initial load incurs a performance penalty, but we're gambling on spatial locality and the chance that the CPU will need the extra data.

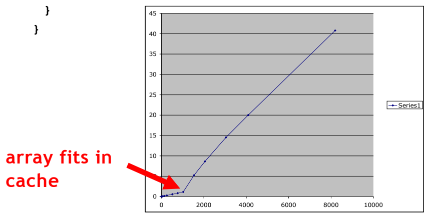


How will execution time grow with SIZE?

```
int array[SIZE];
int sum = 0;

for (int i = 0 ; i < 200000 ; ++ i) {
  for (int j = 0 ; j < SIZE ; ++ j) {
     sum += array[j];
  }
}
```





10

Definitions: Hits and misses

- A **cache hit** occurs if the cache contains the data that we're looking for. ☺
- A **cache miss** occurs if the cache does not contain the requested data. ☹

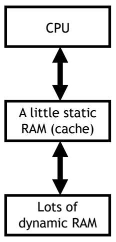
- Two basic measurements of cache performance.
  - the **hit rate** = the percentage of memory accesses that are handled by the cache. (**miss rate** =1 - hit rate)
  - The **miss penalty** = the number of cycles needed to access main memory on a cache miss
- Typical caches have a hit rate of 95% or higher.

- Caches organized in **levels** to reduce miss penalty

Memory System Performance

- Memory system performance depends on three important questions:
  - How long does it take to send data from the cache to the CPU?
  - How long does it take to copy data from memory into the cache?
  - How often do we have to access main memory?

- There are names for all of these variables:
  - The hit time is how long it takes data to be sent from the cache to the processor. This is usually fast, on the order of 1-3 clock cycles.
  - The miss penalty is the time to copy data from main memory to the cache. This often requires dozens of clock cycles (at least).
  - The miss rate is the percentage of misses.



12

Average memory access time

- The average memory access time, or AMAT, can then be computed

  AMAT = Hit time + (Miss rate x Miss penalty)

  This is just averaging the amount of time for cache hits and the amount of time for cache misses

- How can we improve the average memory access time of a system?
  - Obviously, a lower AMAT is better
  - Miss penalties are usually much greater than hit times, so the best way to lower AMAT is to reduce the miss penalty *or* the miss rate

- However, AMAT should only be used as a general guideline. Remember that execution time is still the best performance metric.

13

Performance example

- Assume that 33% of the instructions in a program are data accesses. The cache hit ratio is 97% and the hit time is one cycle, but the miss penalty is 20 cycles.

  AMAT = Hit time + (Miss rate x Miss penalty)

  =

- What is the length of pipeline stages that access memory (IF, MEM)?
  - Hit time
  - Hit time + Miss penalty
  - AMAT

- If the program has *N* dynamic instructions, how many cycles are lost due to stalls?

14

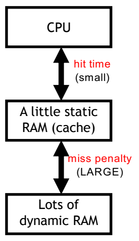
Caches: Quick review

- A cache is a small amount of fast memory located close to the CPU

- Caches take advantage of temporal and spatial locality in programs
  - Temporal: if a program accesses address *x*, it will likely access *x* again soon
  - Spatial: if a program accesses address *x*, it will likely access neighbor(*x*) soon

- AMAT (average memory access time)

  = hit-time + miss-rate × miss-penalty



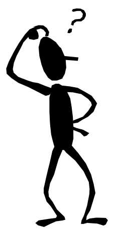
15

A simple cache design

- Caches are divided into blocks, which may be of various sizes.
  - The number of blocks in a cache is usually a power of 2.
  - For now we'll say that each block contains one byte. This won't take advantage of spatial locality, but we'll do that next time.
- Here is an example cache with eight blocks, each holding one byte.

Block index	8-bit data
000	
001	
010	
011	
100	
101	
110	
111	

Four important questions

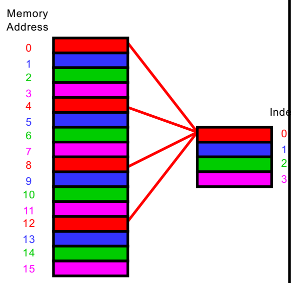


1. When we copy a block of data from main memory to the cache, where exactly should we put it?
2. How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?
3. Eventually, the small cache memory might fill up. To load a new block from main RAM, we'd have to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

- Questions 1 and 2 are related—we have to know where the data is placed if we ever hope to find it again later!

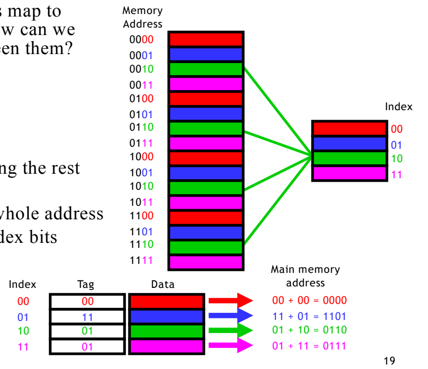
Direct-mapped cache

- Each main memory **block** is mapped to exactly one block in the cache.
  - For example, on the right is a 16-byte main memory and a 4-byte cache (four1-byte blocks).
- Many lower level blocks share a given cache block.
  - E.g., Memory locations 0, 4, 8 and 12 all map to cache block 0.
- How can we compute this mapping?



But wait a minute…

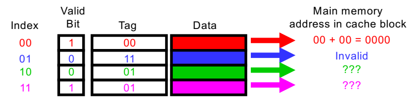
- Several addresses map to the same row. How can we distinguish between them?
- We add a tag, using the rest of the address
  - the tag is the whole address without the index bits



19

One more detail: the valid bit

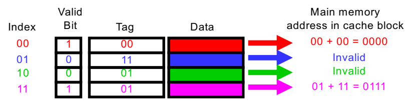
- When started, the cache is empty and does not contain valid data.
- We should account for this by adding a valid bit for each cache block.
  - When the system is initialized, all the valid bits are set to 0.
  - When data is loaded into a particular cache block, the corresponding valid bit is set to 1.



- So the cache contains more than just copies of the data in memory; it also has bits to help us find data within the cache and verify its validity.

One more detail: the valid bit

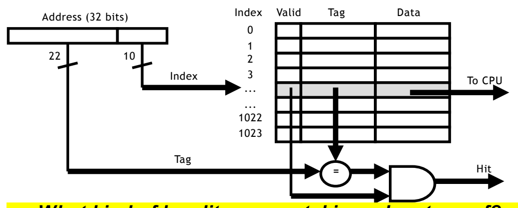
- When started, the cache is empty and does not contain valid data.
- We should account for this by adding a valid bit for each cache block.
  - When the system is initialized, all the valid bits are set to 0.
  - When data is loaded into a particular cache block, the corresponding valid bit is set to 1.



- So the cache contains more than just copies of the data in memory; it also has bits to help us find data within the cache and verify its validity.

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 kind of locality are we taking advantage of?

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).

Direct Mapped Cache

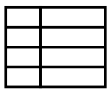
- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15

0000 0001 0010 0011 0100 0011 0100 1111

Address 0

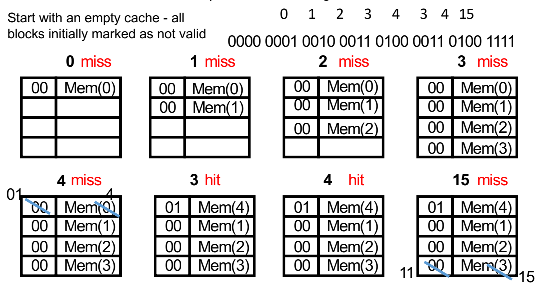


25

Direct Mapped Cache

- Consider the main memory reference string

Start with an empty cache - all blocks initially marked as not valid



0 miss

00	Mem(0)

1 miss

00	Mem(0)
00	Mem(1)

2 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)

3 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 miss

01 ~~00~~	~~Mem(0)~~ 4
00	Mem(1)
00	Mem(2)
00	Mem(3)

3 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

15 miss

01	Mem(4)
00	Mem(1)
00	Mem(2)
11 ~~00~~	~~Mem(3)~~ 15

- 8 requests, 6 misses

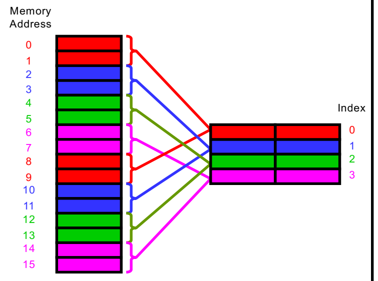
26

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?

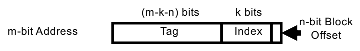
Spatial locality

- 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.

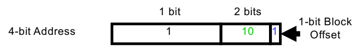


Locating data in the cache

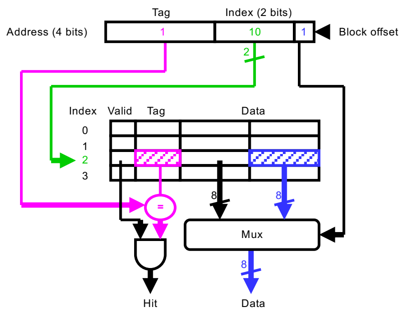
- Let's say we have a cache with 2^k blocks, each containing 2^n bytes.
- We can determine where a byte of data belongs in this cache by looking at its address in main memory.
  - k bits of the address will select one of the 2^k cache blocks.
  - The lowest n bits are now a block offset that decides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



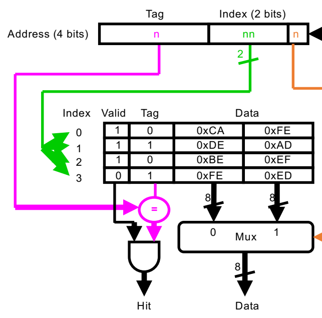
- Our example used a 2^2-block cache with 2^1 bytes per block. Thus, memory address 13 (1101) would be stored in byte 1 of cache block 2.



A picture



An exercise

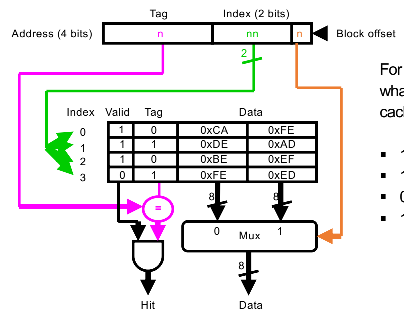


Index	Valid	Tag	Data	
0	1	0	0xCA	0xFE
1	1	1	0xDE	0xAD
2	1	0	0xBE	0xEF
3	0	1	0xFE	0xED

For the addresses below, what byte is read from the cache (or is there a miss)?

- 1010
- 1110
- 0001
- 1101

An exercise



For the addresses below, what byte is read from the cache (or is there a miss)?

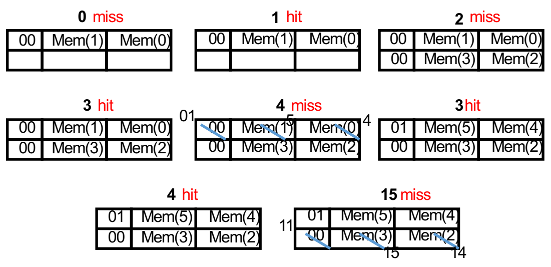
- 1010 (0xDE)
- 1110 (miss, invalid)
- 0001 (0xFE)
- 1101 (miss, bad tag)

Taking Advantage of Spatial Locality

- Let cache block hold more than one word

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15



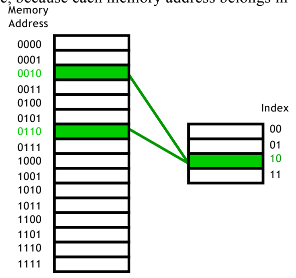
- 8 requests, 4 misses

33

34

Disadvantage of direct mapping

- The direct-mapped cache is easy: indices and offsets can be computed with bit operators or simple arithmetic, because each memory address belongs in exactly one block
- However, this isn't really flexible. If a program uses addresses 10, 110, 10, 110, ... then each access will result in a cache miss and a load into cache block 10
- This cache has four blocks, but direct mapping might not let us use all of them
- This can result in more misses than we might like



35

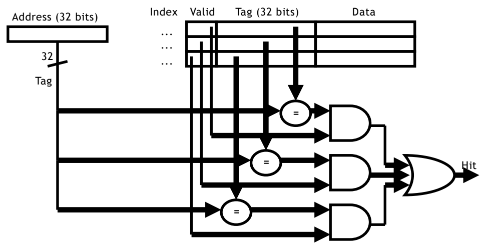
A fully associative cache

- A fully associative cache permits data to be stored in *any* cache block, instead of forcing each memory address into one particular block
  - when data is fetched from memory, it can be placed in *any* unused block of the cache
  - this way we'll never have a conflict between two or more memory addresses which map to a single cache block
- In the previous example, we might put memory address 10 in cache block 10, and address 110 in block 11. Then subsequent repeated accesses to 10 and 110 would all be hits instead of misses.
- If all the blocks are already in use, it's usually best to replace the least recently used one, assuming that if it hasn't used it in a while, it won't be needed again anytime soon.

36

The price of full associativity

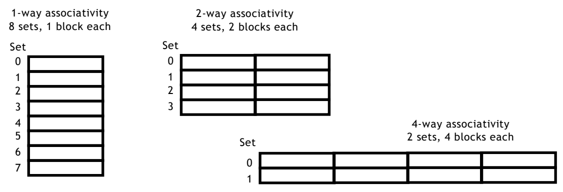
- However, a fully associative cache is expensive to implement.
  - Because there is no index field in the address anymore, the *entire* address must be used as the tag, increasing the total cache size.
  - Data could be anywhere in the cache, so we must check the tag of *every* cache block. That's a lot of comparators!



37

Set associativity

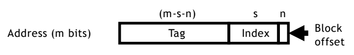
- An intermediate possibility is a set-associative cache
  - The cache is divided into *groups* of blocks, called sets
  - Each memory address maps to exactly one set in the cache, but data may be placed in any block within that set
- If each set has 2^x blocks, the cache is an 2^x-way associative cache
- Here are several possible organizations of an eight-block cache



38

Locating a set associative block

- We can determine where a memory address belongs in an associative cache in a similar way as before.
- If a cache has 2^s sets and each block has 2^n bytes, the memory address can be partitioned as follows.



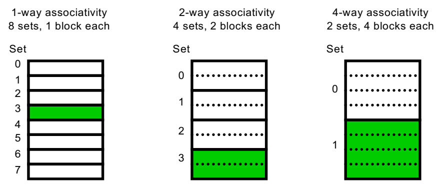
- Our arithmetic computations now compute a set index, to select a *set* within the cache instead of an individual block.

Block Offset = Memory Address mod 2^n

Block Address = Memory Address / 2^n
Set Index = Block Address mod 2^s

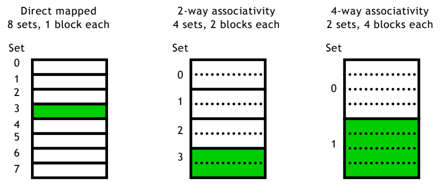
Example placement in set-associative caches

- Where would data from memory byte address 6195 be placed, assuming the eight-block cache designs below, with 16 bytes per block?
- 6195 in binary is 00...0110000 011 0011.
- Each block has 16 bytes, so the lowest 4 bits are the block offset.
- For the 1-way cache, the next three bits (011) are the set index.
  For the 2-way cache, the next two bits (11) are the set index.
  For the 4-way cache, the next one bit (1) is the set index.
- The data may go in *any* block, shown in green, within the correct set.



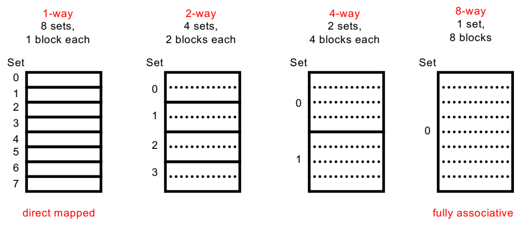
Block replacement

- Any empty block in the correct set may be used for storing data.
- If there are no empty blocks, the cache controller will attempt to replace the least recently used block, just like before.
- For highly associative caches, it's expensive to keep track of what's really the least recently used block, so some approximations are used. We won't get into the details.



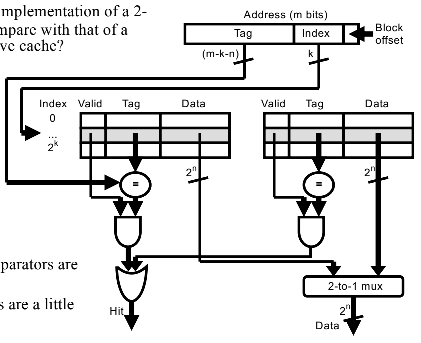
Set associative caches are a general idea

- 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.



2-way set associative cache implementation

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

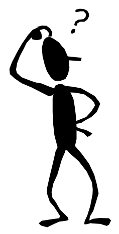


- Only two comparators are needed.
- The cache tags are a little shorter too.

Review So Far

- 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.

Four important questions

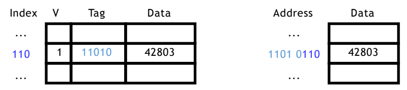


1. When we copy a block of data from main memory to the cache, where exactly should we put it?
2. How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?
3. Eventually, the small cache memory might fill up. To load a new block from main RAM, we' d have to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

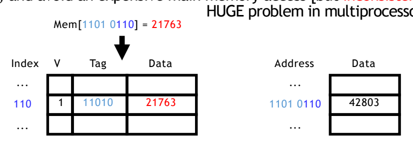
- Previous lectures answered the first 3. Today, we consider the 4th.

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:



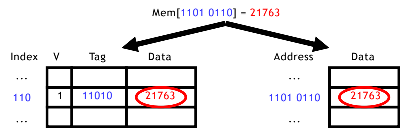
- If we write a new value to that address, we can store the new data in the cache, and avoid an expensive main memory access [but inconsistent]
  HUGE problem in multiprocessors



54

Write-through caches

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

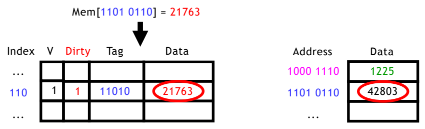


- This is simple to implement and keeps the cache and memory consistent
- Why is this not so good?

55

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

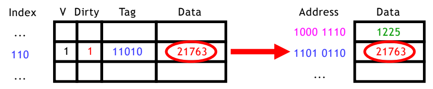


- Subsequent reads to the same memory address will be serviced by the cache, which contains the correct, updated data

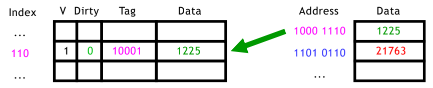
56

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



57

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)	138 Kbits	H teal)	148 Kbits

58

Answer

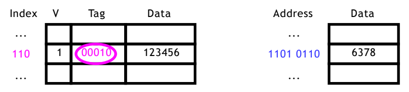
- How many total *bits* are required for that cache? (Round to nearest Kbits)
  - Direct-mapped, write-through, 16 KBytes of data, 4-word (16 Byte) blocks, 32-bit address
  - Tag <18 bits> | Index <10 bits> | Block Offset <4 bits>

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)	138 Kbits	H teal)	148 Kbits

59

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

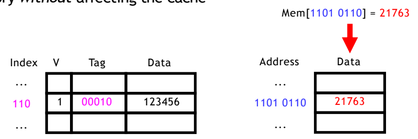


- When we update Mem[1101 0110], should we *also* load it into the cache?

60

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

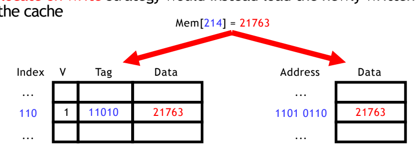


- This is good when data is written but not immediately used again, in which case there's no point to load it into the cache yet

61

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

62

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

63