CS3350B Computer Architecture
Memory Hierarchy: How?

Marc Moreno Maza

http://www.csd.uwo.ca/~moreno/cs3350_moreno/index.html
Department of Computer Science
University of Western Ontario, Canada

Tuesday January 19, 2017

How is the hierarchy managed?

- registers $\leftrightarrow$ cache memory
  - the compiler (with the programmer's help) decides which values are stored in the registers
- cache $\leftrightarrow$ main memory
  - the cache controller hardware handles
- main memory $\leftrightarrow$ disks
  - the operating system (who controls the virtual memory)
  - virtual to physical address mapping is assisted by the hardware (TLB)
  - the programmer (who organizes the data into files)

Cache design questions

Q1 How best to organize the memory blocks (= lines) of the cache memories?

Q2 To which block (line) of the cache does a given main memory address map?

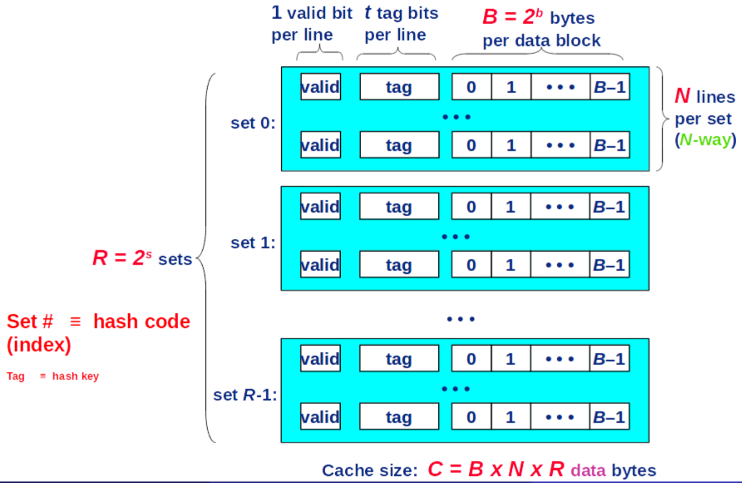
- Since the cache is a subset of the main memory, multiple memory addresses can map to the same cache location

Q3 How do we know if a block of the main memory currently has a copy in cache?

Q4 How do we find this copy quickly?

General organization of a cache memory

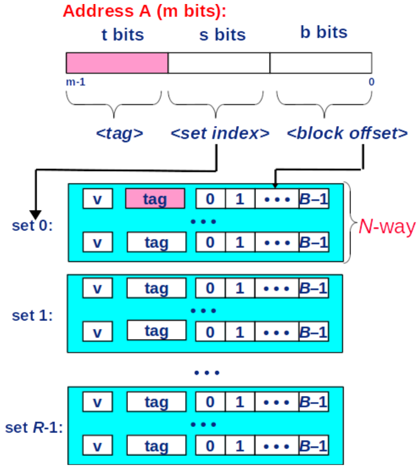
- Cache is an array of $R = 2^s$ sets
- Each set contains one or N lines
- Each line holds a block of $B = 2^b$ bytes of data



Addressing cache memories (memory-cache mapping)

```
lw $t0,0($s1) #$t0=Mem($s1)
sw $t0,0($s1) #Mem($s1)=$t0
```

- The data word at **address A** is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>
- The word contents begin at offset <block offset> bytes from the beginning of the block
- Address mapping:
  set# = (block address) modulo (R)
  block address = concatenated(<t bits>, <s bits>)



$b = \log_2(B)$, $R = C/(B * N)$
$s = \log_2(R)$, $t = m - s - b$

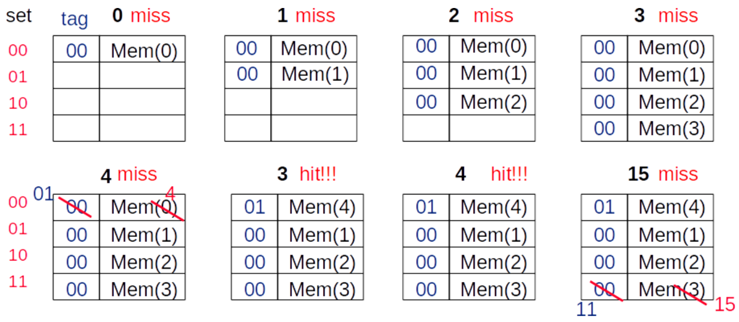
Types of cache organization

- **Direct-mapped**
  - $N = 1$
    - one line per set
    - each memory block is mapped to exactly one line in the cache)
  - $b = \log_2(B)$, $R = C/B$, $s = \log_2(R)$, $t = m - s - b$
- **Fully associative**
  - $R = 1$ (allow a memory block to be mapped to any cache block)
  - $b = \log_2(B)$, $N = C/B$, $s = 0$, $t = m - b$
- **n-way set associative**
  - N is typically 2, 4, 8, or 16
  - A memory block maps to a unique set (specified by the index field) and can be placed in any way of that set (so there are n choices)
  - $b = \log_2(B)$, $R = C/(B \times N)$, $s = \log_2(R)$, $t = m - s - b$

Direct mapped cache example (with 1 word data block)

- Assume that an address uses a 2-bit tag, a 2-bit set index and that we ignore the data block (or equivalently we set $B = 0$)
- Start with an empty cache – all blocks initially marked as not valid, then consider the sequence of memory address accesses:

0	1	2	3	4	3	4	15
0000	0001	0010	0011	0100	0011	0100	1111



0 miss

set	tag	
00	00	Mem(0)
01		
10		
11		

1 miss

set	tag	
00	00	Mem(0)
01	00	Mem(1)
10		
11		

2 miss

set	tag	
00	00	Mem(0)
01	00	Mem(1)
10	00	Mem(2)
11		

3 miss

set	tag	
00	00	Mem(0)
01	00	Mem(1)
10	00	Mem(2)
11	00	Mem(3)

4 miss

set	tag	
00	01 ~~00~~	~~Mem(0)~~ 4
01	00	Mem(1)
10	00	Mem(2)
11	00	Mem(3)

3 hit!!!

set	tag	
00	01	Mem(4)
01	00	Mem(1)
10	00	Mem(2)
11	00	Mem(3)

4 hit!!!

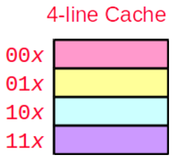
set	tag	
00	01	Mem(4)
01	00	Mem(1)
10	00	Mem(2)
11	00	Mem(3)

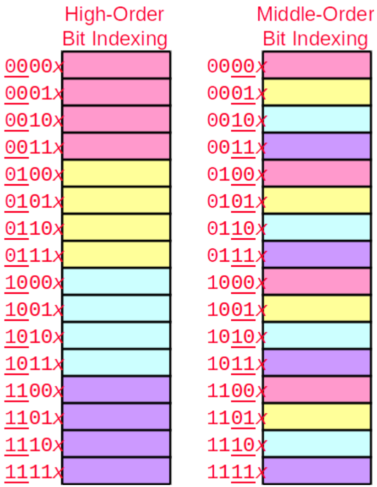
15 miss

set	tag	
00	01	Mem(4)
01	00	Mem(1)
10	00	Mem(2)
11	~~00~~ 11	~~Mem(3)~~ 15

8 requests, 2 hits, 6 misses = 25% hit rate

Why do we use middle bits as set index?

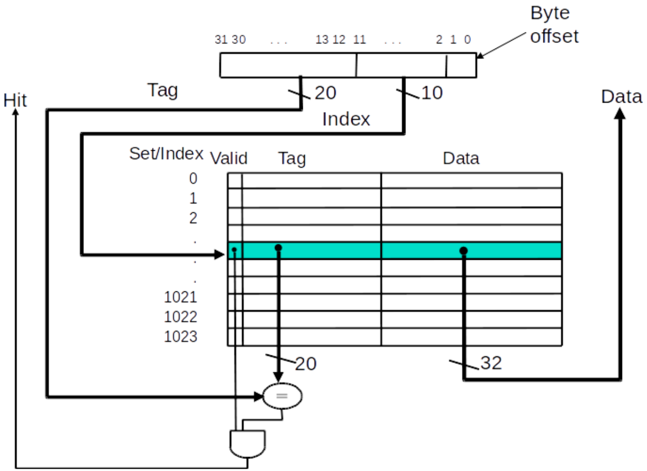




- High-Order Bit Indexing
  - Adjacent memory lines would map to same cache entry
  - Poor use of spatial locality
- Middle-Order Bit Indexing
  - Consecutive memory lines map to different cache lines
  - Can hold C-byte region of address space in cache at one time
  - What type of locality?

Direct mapped Cache example

- One word (4 Byte) data blocks, cache size = 1K words (or 4KB)

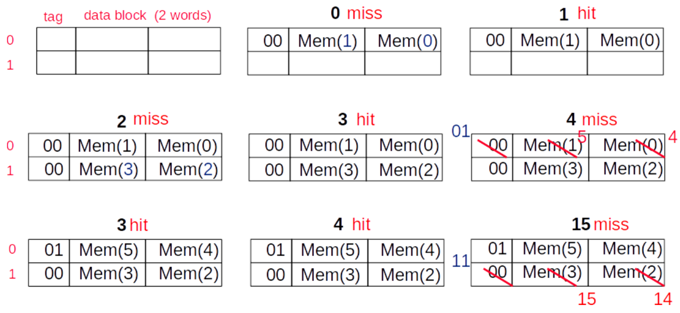


What kind of locality are we taking advantage of?

Taking advantage of spatial locality

- Let the cache block hold more than one word, say, two.
- Assume that an address uses 2 bits of tag, 1 bit of set index and 1-bit of word-in-block select.
- Start with an empty cache – all blocks initially marked as not valid; then consider the sequence of memory address accesses:

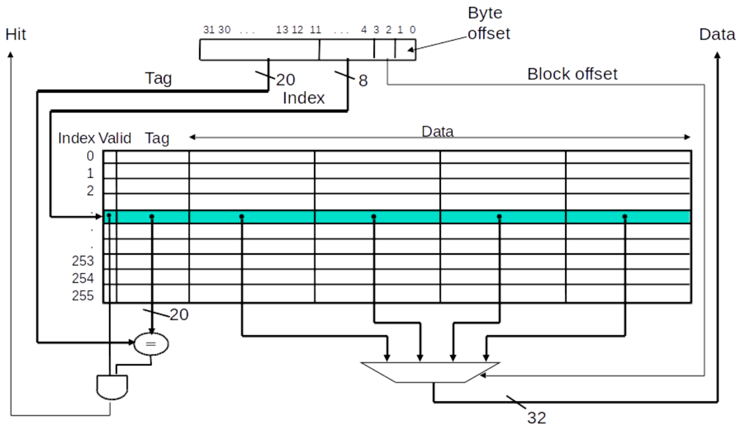
0	1	2	3	4	3	4	15
0000	0001	0010	0011	0100	0011	0100	1111



8 requests, 4 hits, 4 misses = 50% hit rate!

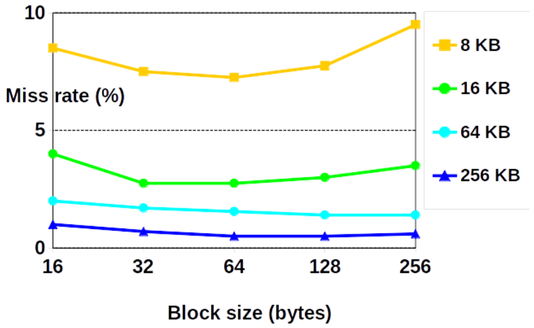
A Direct mapped cache with multiword block

- Four data words/block, cache size = 1K words (256 blocks, 4KB total data)



What kind of locality are we taking advantage of?

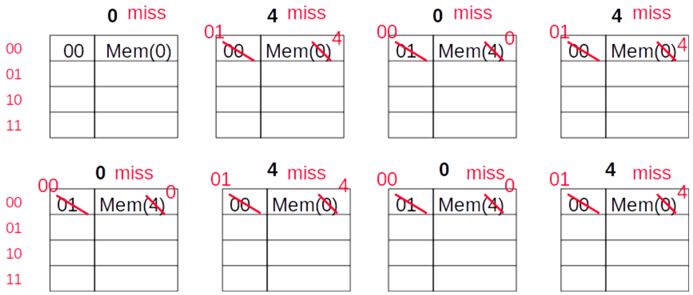
Miss rate vs block size vs cache size



- Miss rate goes up if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the same size cache is smaller (increasing capacity misses)

Exp: 4-word direct-mapped cache for a worst-case reference sequence

- Consider the main memory word reference sequence: 0, 4, 0, 4, 0, 4, 0, 4.

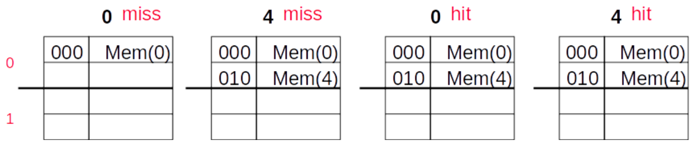


8 requests, 8 misses

- Ping pong effect due to **conflict** misses – two memory locations that map into the same cache block

Exp: 4-word 2-way associative cache for the same reference sequence

- Consider the same main memory word reference sequence: 0, 4, 0, 4, 0, 4, 0, 4.

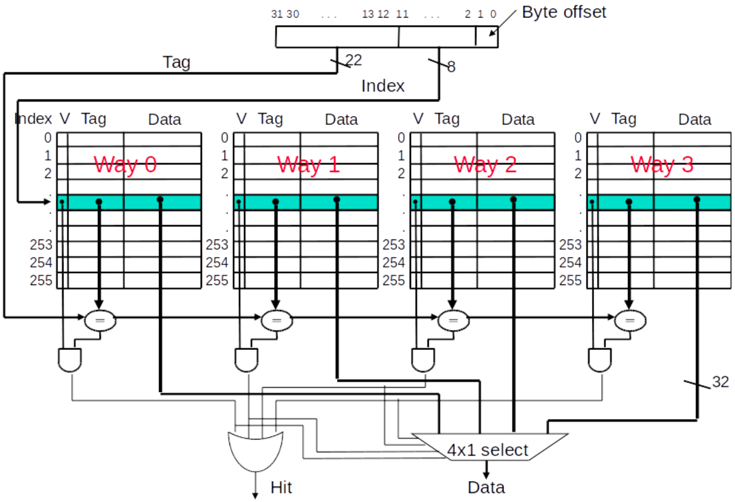


8 requests, 2 misses

- Solves the ping pong effect in a direct mapped cache due to conflict misses since now two memory locations that map into the same cache set can co-exist!

Four-way set associative cache

- $2^8 = 256$ sets each with four ways (each with one block)

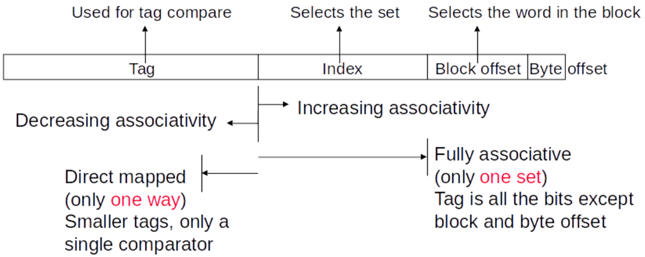


Extra costs of set associative caches

- When a miss occurs, which way do we pick for replacement?
  - **Least Recently Used (LRU)**: the block replaced is the one that has been unused for the longest time:
    - must have hardware to keep track of when each way was used relative to the other blocks in the set
    - for 2-way set associative, takes one bit per set → set the bit when a block is referenced (and reset the other way's bit)
- N-way set associative cache costs
  - N comparators (delay and area)
  - MUX delay (set selection) before data is available
  - Data available after set selection (and Hit/Miss decision). In a direct mapped cache, the cache block is available before the Hit/Miss decision
    - So its not possible to just assume a hit and continue and recover later if it was a miss

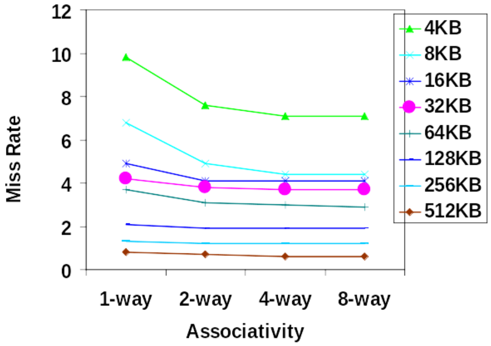
Range of set associative caches

- For a fixed size cache, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number of ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit



Benefits of set associative caches

- The choice between *direct mapped* and *set associative* depends on the cost of a miss versus the cost of implementation



- On popular benchmark test-programs, largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate)

What parameters do you not know by far?

	Intel Nehalem	**AMD Barcelona**
L1 cache size & organization	32KB for each per core; 64B blocks; Split I$ and D$	64KB for each per core; 64B blocks; Split I$ and D$
L1 associativity	4-way (I), 8-way (D) set assoc.; ~LRU replacement	2-way set assoc.; LRU replacement
L1 write policy	write-back, write-allocate	write-back, write-allocate
L2 cache size & organization	256MB (0.25MB) per core; 64B blocks; Unified	512KB (0.5MB) per core; 64B blocks; Unified
L2 associativity	8-way set assoc.; ~LRU	16-way set assoc.; ~LRU
L2 write policy	write-back, write-allocate	write-back, write-allocate
L3 cache size & organization	8192KB (8MB) shared by cores; 64B blocks; Unified	2048KB (2MB) shared by by cores; 64B blocks; Unified
L3 associativity	16-way set assoc.	32-way set assoc.; evict block shared by fewest cores
L3 write policy	write-back, write-allocate	write-back, write-allocate

Handling cache hits

- Read hits (I$ and D$)
  - this is what we want!
- Write hits (D$ only)
  - **Write-through**: require the cache and memory to be consistent
    - always write the data into both the cache block and the next level in the memory hierarchy
    - writes run at the speed of the next level in the memory hierarchy – so slow! – or can use a write buffer and stall only if the write buffer is full.
  - **Write-back**: allow cache and memory to be inconsistent
    - write the data into the cache block and write the modified cache block (dirty block) to the next level in the memory hierarchy only when that cache block is replaced
    - either require two cycles (a cycle to check for a hit followed by a cycle to actually perform the write) or require a write buffer to hold that data—effectively allowing the store to take only one cycle

Handling cache misses

- Read misses (I$ and D$)
  - Stall the execution, fetch the block from the next level in the memory hierarchy, install it in the cache and send the requested word to the processor, then let the processor resume.
- Write misses (D$ only)
  - Stall the execution, fetch the block from next level in the memory hierarchy, install it in the cache,
    - Write allocate – write the word into the cache updating both the tag and the portion of data block. In a write-back cache, we must first write the block back to memory if the data in the cache is modified
    - No-write allocate – skip the cache write and directly write the word to the write buffer (and eventually to the next memory level)

    then let the processor resume

Sources of cache misses

- **Compulsory**:
  - cause: (**cold** start: process migration or first reference)
  - how to reduce impact:
- **Capacity**:
  - cause: the cache cannot contain all blocks accessed by the program
  - how to reduce impact:
- **Conflict** (collision):
  - cause: multiple memory locations mapped to the same cache location
  - how to reduce impact:

Sources of cache misses

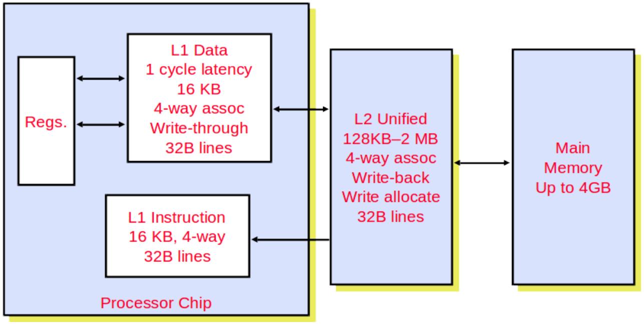
- **Compulsory**:
  - cause: (**cold** start: process migration or first reference)
  - how to reduce impact: increase block size (increases miss penalty; very large blocks could increase miss rate)
- **Capacity**:
  - cause: the cache cannot contain all blocks accessed by the program
  - how to reduce impact: increase cache size (may increase access time)
- **Conflict** (collision):
  - cause: multiple memory locations mapped to the same cache location
  - how to reduce impact: increase cache size and/or increase associativity (may increase access time)

Further reducing cache miss rates

Use multiple levels of caches

- With advancing technology, we have more and more room on the die for bigger L1 caches or for a second level of caches – normally a unified L2 cache (i.e., holding both instructions and data) and in some cases even a unified L3 cache.
- **New AMAT Calculation**:
  AMAT = L1 Hit Time + L1 Miss Rate * L1 Miss Penalty,
  L1 Miss Penalty = L2 Hit Time + L2 Miss Rate * L2 Miss Penalty,
  and so forth (final miss penalty is Main Memory access time)
- **Example**: 1 cycle L1 hit time, 2% L1 miss rate, 5 cycle L2 hit time, 5% L2 miss rate,100 cycle main memory access time
  - Without L2 cache:
    AMAT = 1 + .02*100 = 3
  - With L2 cache:
    AMAT = 1 + .02*(5 + .05*100) = 1.2

Intel Pentium cache hierarchy



Multilevel cache design considerations

- Design considerations for L1 and L2 caches are very different
  - Primary cache should focus on minimizing hit time in support of a shorter clock cycle
    - Smaller capacity with smaller block sizes
  - Secondary cache(s) should focus on reducing miss rate to reduce the penalty of long main memory access times
    - Larger capacity with larger block sizes
    - Higher levels of associativity
- The miss penalty of the L1 cache is significantly reduced by the presence of an L2 cache – so it can be smaller (i.e., faster) but have a higher miss rate
- For the L2 cache, hit time is less important than miss rate
  - The L2$ hit time determines L1$'s miss penalty
  - L2$ local miss rate $\gg$ than the global-cache-hierarchy miss rate.

Summary: improving cache performance

AMAT = Time for a Hit + Miss Rate * Miss Penalty

(1) Reduce the time to hit in the cache
- smaller cache
- direct mapped cache
- smaller blocks
- for writes, two possible strategies:
  - no-write allocate: no "hit" on cache, just write to write buffer
  - write allocate: to avoid two cycles (first check for hit, then write) pipeline writes via a delayed write buffer to cache

(2) Reduce the miss rate
- bigger cache
- more flexible placement (increase associativity)
- larger blocks (16 to 64 bytes typical)
- using a *victim-cache*, that is, a small buffer holding most recently discarded blocks.

Summary: improving cache performance

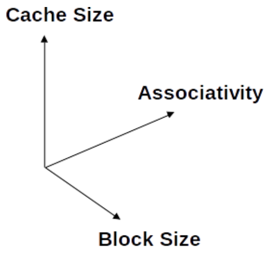
AMAT = Time for a Hit + Miss Rate * Miss Penalty

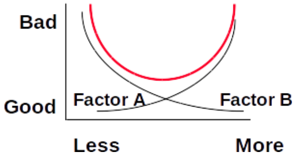
(3) Reduce the miss penalty

- smaller blocks
- use a write-buffer to hold dirty blocks being replaced so don't have to wait for the write to complete before reading
- check the write-buffer (and/or the victim-cache) on a read miss: we may get lucky!
- for large blocks, fetch the critical word first
- use multiple cache levels - (Note that the L2 cache is not tied to the CPU clock rate)
- faster backing store (= main memory) and improved memory bandwidth (e.g. wider buses).

Summary: the cache design space

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back
  - write allocation
- The optimal choice is a compromise
  - depends on access characteristics
    - workload
    - use (I-cache, D-cache, TLB)
  - depends on technology / cost





Takeaway

- The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time
    - Temporal locality: locality in Time
    - Spatial locality: locality in Space
- Three major categories of cache misses:
  - Compulsory misses: sad facts of life. Example: cold start misses
  - Conflict misses: increase cache size and/or associativity Nightmare Scenario: ping pong effect!
  - Capacity misses: increase cache size
- Cache design space
  - total size, block size, associativity (replacement policy)
  - write-hit policy (write-through, write-back)
  - write-miss policy (write allocate, write buffers)

Self-review questions for the memory hierarchy

Q1 Where can an entry be placed and how is an entry found if it is in the upper level?
Entry placement & identification

Q2 Which entry should be replaced on a miss?
Entry replacement

Q3 What happens on a write?
Write hit/miss strategy

Q1: where can an entry be placed/found?

	# of sets	Entries per set
Direct mapped	# of cache lines	1
Set associative	(# of cache lines) / associativity	Associativity (typically 2 to 16)
Fully associative	1	# of entries

	Location method	# of comparisons
Direct mapped	Index	1
Set associative	Index the set; compare set's tags	Degree of associativity
Fully associative	Compare all entries' tags	# of entries

Q2: which entry should be replaced on a miss?

- Easy for direct mapped - only one choice
- Set associative or fully associative
  - Random
  - LRU (Least Recently Used)
- For a 2-way set associative, random replacement has a miss rate about 1.1 times higher than LRU
- LRU is too costly to implement for high levels of associativity ($>$ 4-way) since tracking the usage information is costly