香港中文大學
The Chinese University of Hong Kong

CENG3420

Lecture 10: Cache

Bei Yu

byu@cse.cuhk.edu.hk

(Latest update: March 20, 2017)

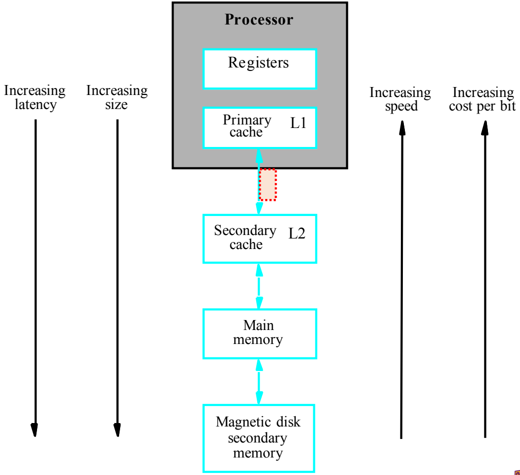
2017 Spring

Overview

Overview

Memory Hierarchy

- **Aim**: to produce fast, big and cheap memory
- L1, L2 cache are usually SRAM
- Main memory is DRAM
- Relies on locality of reference



Cache-Main Memory Mapping

- A way to record which part of the Main Memory is now in cache
- Synonym: Cache line == Cache block
- **Design concerns**:
  - Be Efficient: fast determination of cache hits/ misses
  - Be Effective: make full use of the cache; increase probability of cache hits

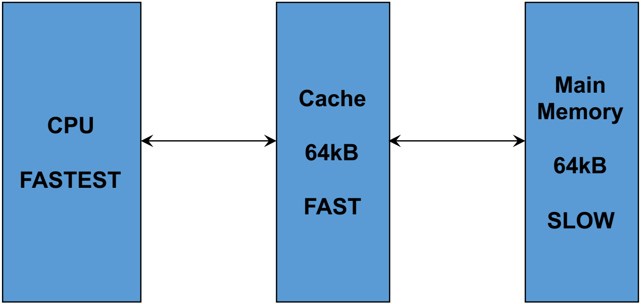
Two questions to answer (in hardware)

Q1 How do we know if a data item is in the cache?

Q2 If it is, how do we find it?

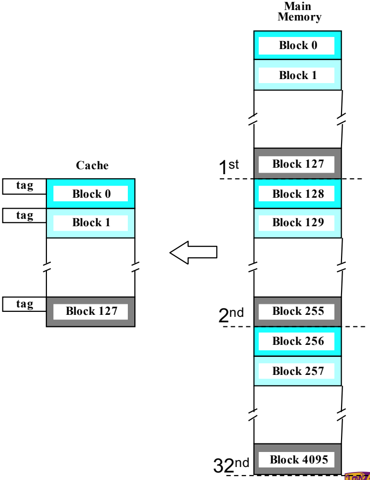
Imagine: Trivial Conceptual Case

- Cache size == Main Memory size
- Trivial one-to-one mapping
- Do we need Main Memory any more?



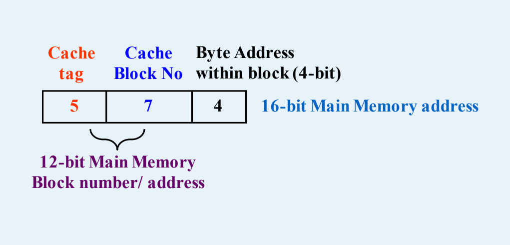
Reality: Cache Block / Cache Line

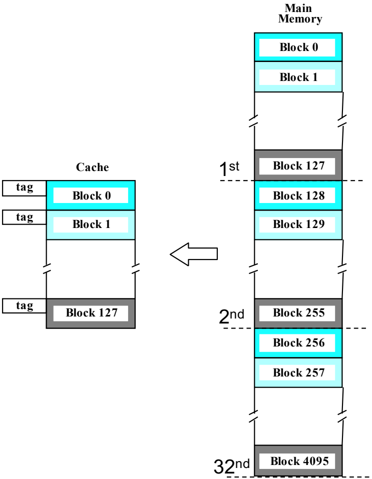
- Cache size is much smaller than the Main Memory size
- A block in the Main Memory maps to a block in the Cache
- Many-to-One Mapping



Overview

Direct Mapping





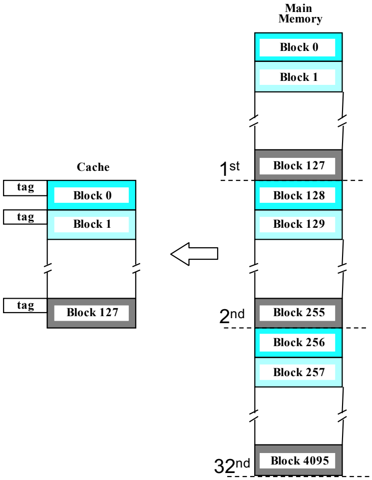
- $2^4 = 16$ bytes in a block
- $2^7 = 128$ Cache blocks
- $2^{(7+5)} = 4096$ main memory blocks

Direct Mapping

Cache tag	Cache Block No	Byte Address within block (4-bit)
5	7	4

16-bit Main Memory address

12-bit Main Memory Block number/ address



- $2^4 = 16$ bytes in a block
- $2^7 = 128$ Cache blocks
- $2^{(7+5)} = 4096$ main memory blocks

  - Block j of main memory maps to block (j mod 128) of Cache (same colour in figure)
  - Cache hit occurs if tag matches desired address

Direct Mapping

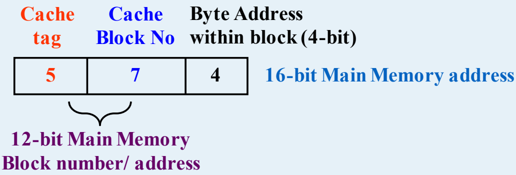
Memory address divided into 3 fields

- Main Memory Block number determines position of block in cache
- Tag used to keep track of which block is in cache (as many MM blocks can map to same position in cache)
- The **last bits** in the address selects target word in the block

Example: given an address (t,b,**w**) (16-bit)

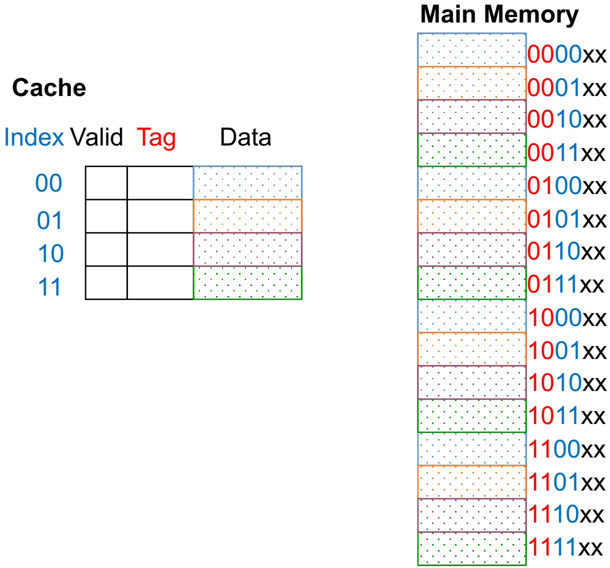
1. See if it is already in cache by comparing t with the tag in block b
2. If not, cache miss! Replace the current block at b with a new one from memory block (t,b) (12-bit)

Direct Mapping Example 1

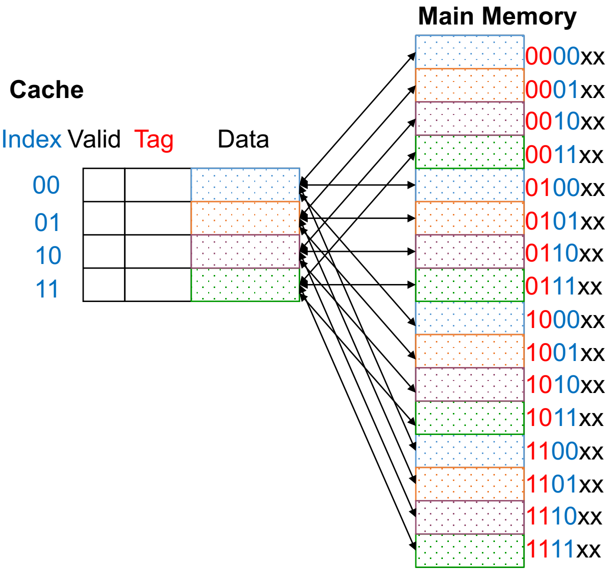


1. CPU is looking for [A7B4] MAR = 101001111011**0100**
2. Go to cache block 1111011, see if the tag is 10100
3. If YES, cache hit!
4. Otherwise, get the block into cache row 1111011

Direct Mapping Example 2



Direct Mapping Example 2



Question: Direct Mapping Cache Hit Rate

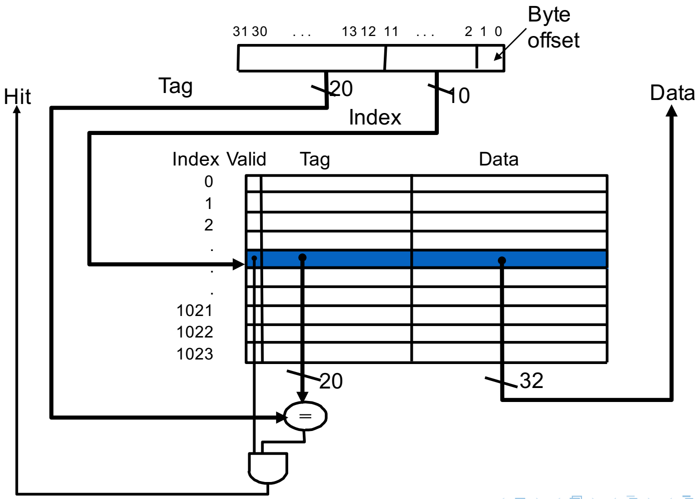
Consider a 4-block empty Cache, and all blocks initially marked as `not valid`), Given the main memory word addresses “0 1 2 3 4 3 4 15”, calculate Cache hit rate.

Cache

Index	Valid	Tag	Data
00			
01			
10			
11			

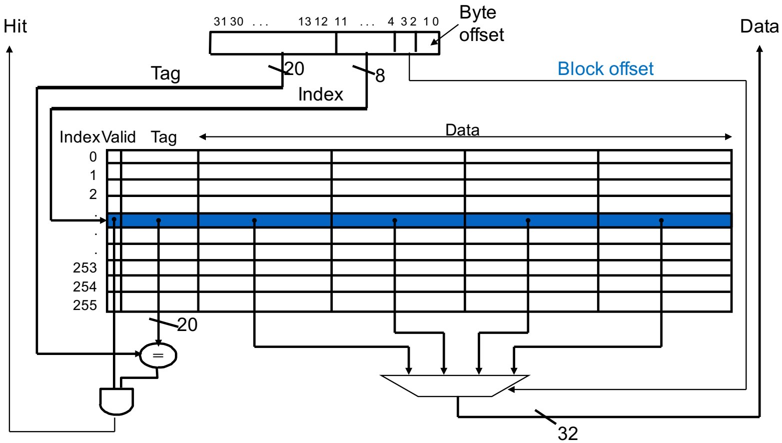
Example 3: MIPS

- One word blocks, cache size = 1K words (or 4KB)
- What kind of locality are we taking advantage of?



Example 4: MIPS w. Multiword Block

- Four words/block, cache size = 1K words
- What kind of locality are we taking advantage of?



Question: Multiword Direct Mapping Cache Hit Rate

Consider a 2-block empty Cache, and each block is with 2-words. All blocks initially marked as `not valid`). Given the main memory word addresses "0 1 2 3 4 3 4 15", calculate Cache hit rate.

Cache

Index	Tag	Data	
00			
01			

MIPS Cache Field Sizes

The number of bits includes both the storage for data and for the tags

- For a direct mapped cache with 2^n blocks, n bits are used for the index
- For a block size of 2^m words (2^{m+2} bytes), m bits are used to address the word within the block
- 2 bits are used to address the byte within the word

MIPS Cache Field Sizes

The number of bits includes both the storage for data and for the tags

- For a direct mapped cache with 2^n blocks, n bits are used for the index
- For a block size of 2^m words (2^{m+2} bytes), m bits are used to address the word within the block
- 2 bits are used to address the byte within the word

Size of the tag field?

$$32 - (n + m + 2)$$

MIPS Cache Field Sizes

The number of bits includes both the storage for data and for the tags

- For a direct mapped cache with 2^n blocks, n bits are used for the index
- For a block size of 2^m words (2^{m+2} bytes), m bits are used to address the word within the block
- 2 bits are used to address the byte within the word

Size of the tag field?

$$32 - (n + m + 2)$$

Total number of bits in a direct-mapped cache

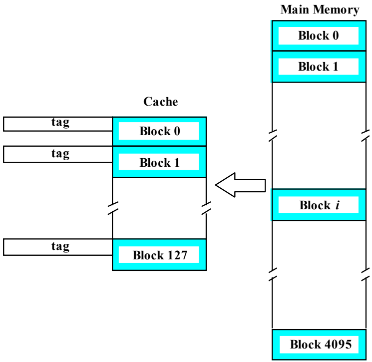
$$2^n \times (\text{block size} + \text{tag field size} + \text{valid field size})$$

Question: Bit number in a Cache

How many total bits are required for a direct mapped cache with 16KB of data and 4-word blocks assuming a 32-bit address?

Overview

Associative Mapping



- A MM block can be in **arbitrary** Cache block location
- In this example, all 128 tag entries must be compared with the address Tag in parallel (by hardware)

Associative Mapping Example

Tag	Byte
12	4

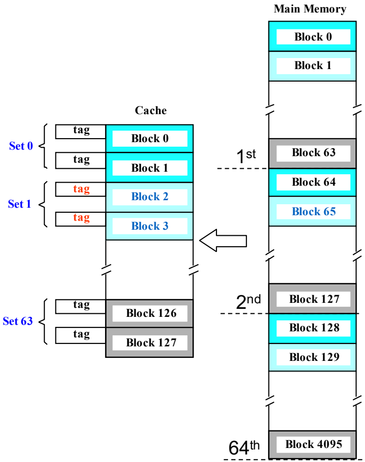
16-bit Main Memory address

1. CPU is looking for [A7B4] MAR = 101001111011**0100**
2. See if the tag 101001111011 matches one of the 128 cache tags
3. If YES, cache hit!
4. Otherwise, get the block into BINGO cache row

Set Associative Mapping

Tag	Set Number	Byte
6	6	4

16-bit Main Memory address



- Combination of direct and associative
- (j mod 64) derives the Set Number
- A cache with k-blocks per set is called a k-way set associative cache.

Example: 2-way set associative

Set Associative Mapping Example 1

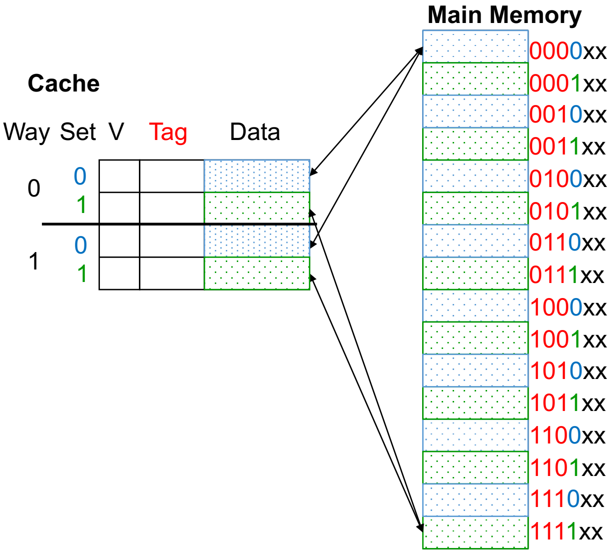
Tag	Set Number	Byte
6	6	4

16-bit Main Memory address

E.g. 2-Way Set Associative:

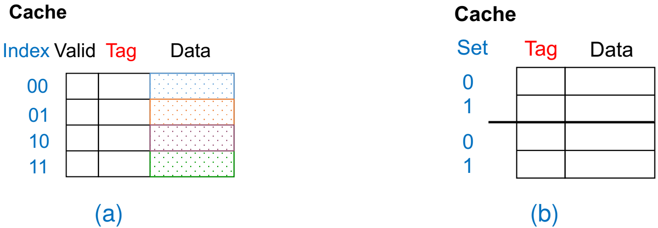
1. CPU is looking for [A7B4] MAR = 101001111011**0100**
2. Go to cache Set 111011 (59_{10})
   - Block 1110110 (118_{10})
   - Block 1110111 (119_{10})
3. See if ONE of the TWO tags in the Set 111011 is 101001
4. If YES, cache hit!
5. Get the block into BINGO cache row

Set Associative Mapping Example 2



Question: Direct Mapping v.s. 2-Way Set Associate

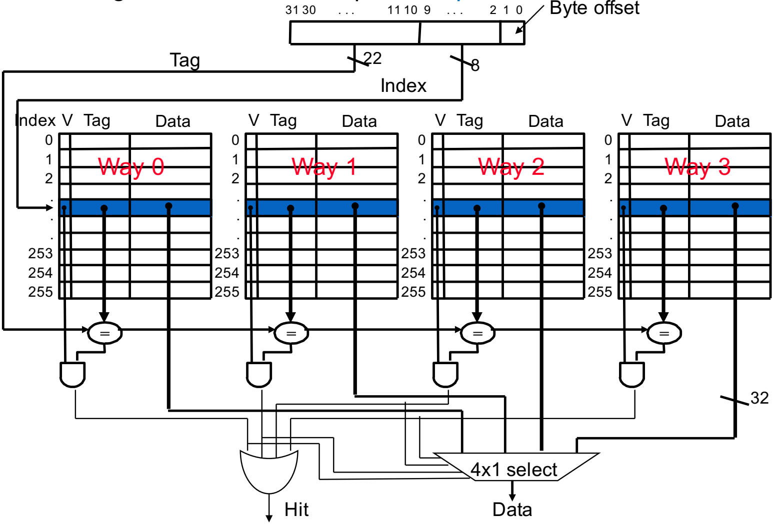
Consider the following two empty caches, calculate Cache hit rates for the reference word addresses: “0 4 0 4 0 4 0 4”



(a) Direct Mapping; (b) 2-Way Set Associative.

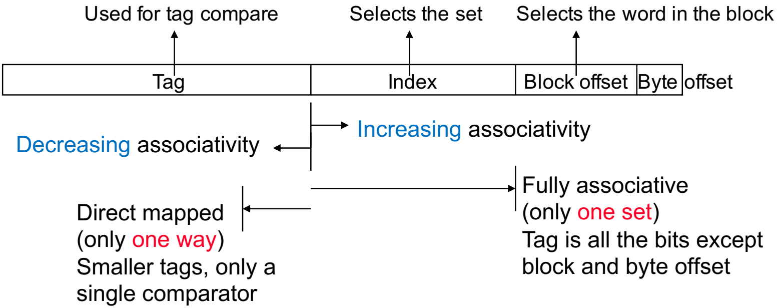
Set Associative Mapping Example 3: MIPS

- $2^8 = 256$ sets each with four ways (each with one block).
- four tags in the set are compared in parallel.



Range of Set Associative Caches

For a fixed size cache:



Overview

Handling Cache Read

- `I$` and `D$`
- Read hit: what we want!
- Read miss: stall the pipeline, 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 pipeline resume.

Handling Cache Write Hits

Only `D$`

Case 1: Write-Through

- Cache and memory to be consistent
- always write the data into both the cache block and the next level in the memory hierarchy
- Speed-up: use write buffer and stall only when buffer is full

Case 2: Write-Back

- Write the data only into the cache block
- Write to memory hierarchy when that cache block is "evicted"
- Need a dirty bit for each data cache block

Handling Cache Write Misses

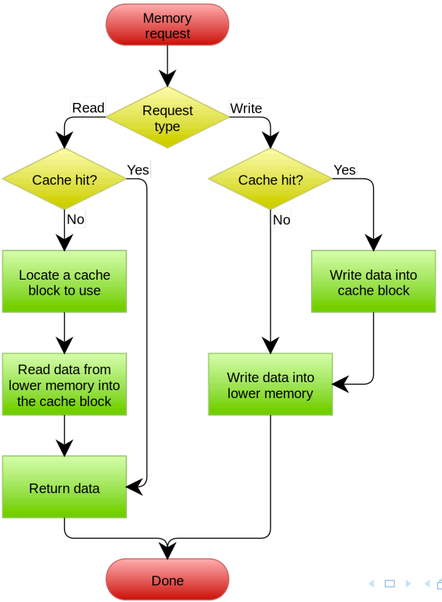
Case 1: Write-Through caches with a write buffer

- No-write allocate
- skip the cache write (but must invalidate that cache block since it will now hold stale data)
- just write the word to the write buffer (and eventually to the next memory level)
- no need to stall if the write buffer isn't full

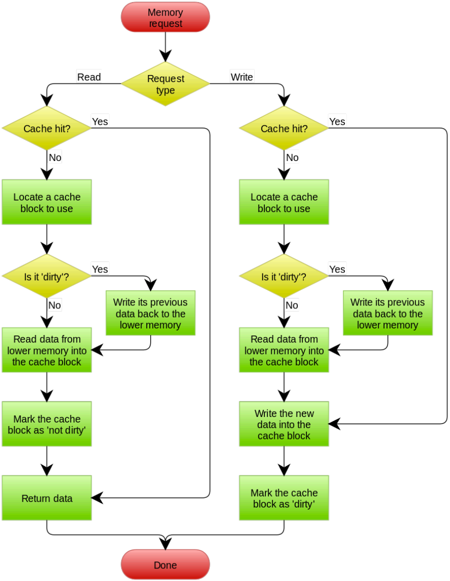
Case 2: Write-Back caches

- Write allocate
- Just write the word into the cache updating both the tag and data
- no need to check for cache hit
- no need to stall

Write-Through Cache with No-Write Allocation



Write-Back Cache with Write Allocation



Replacement Algorithms

Direct Mapping

- Position of each block fixed
- Whenever replacement is needed (i.e. cache miss $\rightarrow$ new block to load), the choice is obvious and thus no "replacement algorithm" is needed

Associative and Set Associative

- Need to decide which block to replace
- Keep/retain ones likely to be used in near future again

Associative & Set Associative Replacement

Strategy 1: Least Recently Used (LRU)

- e.g. for a 4-block/set cache, use a $log_2 4 = 2$ bit counter for each block
- Reset the counter to 0 whenever the block is accessed
- counters of other blocks in the same set should be incremented
- On cache miss, replace/ uncache a block with counter reaching 3

Associative & Set Associative Replacement

Strategy 1: Least Recently Used (LRU)

- e.g. for a 4-block/set cache, use a $log_2 4 = 2$ bit counter for each block
- Reset the counter to 0 whenever the block is accessed
- counters of other blocks in the same set should be incremented
- On cache miss, replace/ uncache a block with counter reaching 3

Strategy 2: Random Replacement

- Choose random block
- ☺Easier to implement at high speed

Cache Example

```
short  A[10][4];
int    sum = 0;
int    j, i;
double mean;

// forward loop
for (j = 0; j <= 9; j++)
    sum += A[j][0];

mean = sum / 10.0;

// backward loop
for (i = 9; i >= 0; i--)
    A[i][0] = A[i][0]/mean;
```

- Assume separate instruction and data caches
- So we consider only the data
- Cache has space for 8 blocks
- A block contains one word (byte)
- `A[10][4]` is an array of words located at `7A00-7A27` in row-major order

Cache Example

Memory word address in hex	Memory word address in binary	Array Contents (40 elements)
(7A00)	0 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0	A[0][0]
(7A01)	0 1 1 1 1 0 1 0 0 0 0 0 0 0 0 1	A[0][1]
(7A02)	0 1 1 1 1 0 1 0 0 0 0 0 0 0 1 0	A[0][2]
(7A03)	0 1 1 1 1 0 1 0 0 0 0 0 0 0 1 1	A[0][3]
(7A04)	0 1 1 1 1 0 1 0 0 0 0 0 0 1 0 0	A[1][0]
	⋮	
(7A24)	0 1 1 1 1 0 1 0 0 0 1 0 0 1 0 0	A[9][0]
(7A25)	0 1 1 1 1 0 1 0 0 0 1 0 0 1 0 1	A[9][1]
(7A26)	0 1 1 1 1 0 1 0 0 0 1 0 0 1 1 0	A[9][2]
(7A27)	0 1 1 1 1 0 1 0 0 0 1 0 0 1 1 1	A[9][3]

Tag for Direct Mapped

8 blocks in cache, 3 bits encodes cache block number

Tag for Set-Associative

4 blocks/ set, 2 cache sets, 1 bit encodes cache set number

Tag for Associative

To simplify discussion: 16-bit word (byte) address; i.e. 1 word = 1 byte.

Direct Mapping

- Least significant 3-bits of address determine location
- No replacement algorithm is needed in Direct Mapping
- When `i == 9` and `i == 8`, get a cache hit (2 hits in total)
- Only 2 out of the 8 cache positions used
- Very inefficient cache utilization

Cache Block number	Content of data cache after loop pass: (time line)																			
	j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8	j = 9	i = 9	i = 8	i = 7	i = 6	i = 5	i = 4	i = 3	i = 2	i = 1	i = 0
0	A[0][0]	A[0][0]	A[2][0]	A[2][0]	A[4][0]	A[4][0]	A[6][0]	A[6][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[6][0]	A[6][0]	A[4][0]	A[4][0]	A[2][0]	A[2][0]	A[0][0]
1																				
2																				
3																				
4		A[1][0]	A[1][0]	A[3][0]	A[3][0]	A[5][0]	A[5][0]	A[7][0]	A[7][0]	A[9][0]	A[9][0]	A[9][0]	A[7][0]	A[7][0]	A[5][0]	A[5][0]	A[3][0]	A[3][0]	A[1][0]	A[1][0]
5																				
6																				
7																				

Tags not shown but are needed.

Associative Mapping

- LRU replacement policy: get cache hits for $i = 9, 8, \ldots, 2$
- If `i` loop was a forward one, we would get no hits!

		Content of data cache after loop pass: (time line)																			
Cache Block number		j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8	j = 9	i = 9	i = 8	i = 7	i = 6	i = 5	i = 4	i = 3	i = 2	i = 1	i = 0
	0	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[0][0]
	1		A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[1][0]	A[1][0]
	2			A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]
	3				A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]
	4					A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]
	5						A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]
	6							A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]
	7								A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]

Tags not shown but are needed; LRU Counters not shown but are needed.

Set Associative Mapping

- Since all accessed blocks have even addresses (`7A00, 7A04, 7A08, ...`), only half of the cache is used, i.e. they all map to set 0
- LRU replacement policy: get hits for i = 9, 8, 7 and 6
- Random replacement would have better average performance
- If `i` loop was a forward one, we would get no hits!

		Content of data cache after loop pass: (time line)																			
		j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8	j = 9	i = 9	i = 8	i = 7	i = 6	i = 5	i = 4	i = 3	i = 2	i = 1	i = 0
Set 0	0	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[0][0]
	1		A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[1][0]	A[1][0]
	2			A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[2][0]	A[2][0]	A[2][0]
Cache Block number	3				A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]
	4																				
Set 1	5																				
	6																				
	7																				

Tags not shown but are needed; LRU Counters not shown but are needed.

Comments on the Example

- In this example, Associative is best, then Set-Associative, lastly Direct Mapping.
- What are the advantages and disadvantages of each scheme?
- In practice,
  - Low hit rates like in the example is very rare.
  - Usually Set-Associative with LRU replacement scheme is used.
- Larger blocks and more blocks greatly improve cache hit rate, i.e. more cache memory

Overview

Conclusion

- Cache Organizations:
  Direct, Associative, Set-Associative
- Cache Replacement Algorithms:
  Random, Least Recently Used
- Cache Hit and Miss Penalty