

caching — replacing / writing / tradeoffs

Changelog

20 October 2020: last time: more carefully distinguish addresses and values that come from addresses

20 October 2020: write-allocate: update tag in example

20 October 2020: AMAT exercise: correct “not decrease” to “not increase”

last time

direct-mapped caches

- divide cache data, memory into fixed-sized blocks

- each block of memory \rightarrow single block of cache

- store valid bits to remember if anything stored

- store tags to remember which address data came from

divide addresses in tag/index/offset

- offset: which byte of block to use

 - (each block starts with offset 0)

- index: which set of cache is address mapped to

- tag: rest of address

set-associative caches

- multiple blocks per 'set'

- use tag to identify which block is where

Tag-Index-Offset exercise

m	memory addresses bits (Y86-64: 64)
E	number of blocks per set (“ways”)
$S = 2^s$	number of sets
s	(set) index bits
$B = 2^b$	block size
b	(block) offset bits
$t = m - (s + b)$	tag bits
$C = B \times S \times E$	cache size (excluding metadata)

My desktop:

L1 Data Cache: 32 KB, 8 blocks/set, 64 byte blocks

L2 Cache: 256 KB, 4 blocks/set, 64 byte blocks

L3 Cache: 8 MB, 16 blocks/set, 64 byte blocks

Divide the address 0x34567 into **tag**, **index**, **offset** for each cache.

T-I-O exercise: L1

quantity	value for L1
----------	--------------

block size (given)	$B = 64\text{Byte}$
--------------------	---------------------

	$B = 2^b$ (b : block offset bits)
--	--------------------------------------

T-I-O exercise: L1

quantity	value for L1
----------	--------------

block size (given)	$B = 64\text{Byte}$
--------------------	---------------------

$$B = 2^b \text{ (} b: \text{ block offset bits)}$$

block offset bits	$b = 6$
-------------------	---------

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64\text{Byte}$
	$B = 2^b$ (b : block offset bits)
block offset bits	$b = 6$
blocks/set (given)	$E = 8$
cache size (given)	$C = 32\text{KB} = E \times B \times S$

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64\text{Byte}$
	$B = 2^b$ (b : block offset bits)
block offset bits	$b = 6$
blocks/set (given)	$E = 8$
cache size (given)	$C = 32\text{KB} = E \times B \times S$
	$S = \frac{C}{B \times E}$ (S : number of sets)

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64\text{Byte}$
	$B = 2^b$ (b : block offset bits)
block offset bits	$b = 6$
blocks/set (given)	$E = 8$
cache size (given)	$C = 32\text{KB} = E \times B \times S$
	$S = \frac{C}{B \times E}$ (S : number of sets)
number of sets	$S = \frac{32\text{KB}}{64\text{Byte} \times 8} = 64$

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64\text{Byte}$
	$B = 2^b$ (b : block offset bits)
block offset bits	$b = 6$
blocks/set (given)	$E = 8$
cache size (given)	$C = 32\text{KB} = E \times B \times S$
	$S = \frac{C}{B \times E}$ (S : number of sets)
number of sets	$S = \frac{32\text{KB}}{64\text{Byte} \times 8} = 64$
	$S = 2^s$ (s : set index bits)
set index bits	$s = \log_2(64) = 6$

T-I-O results

	L1	L2	L3
sets	64	1024	8192
block offset bits	6	6	6
set index bits	6	10	13
tag bits		(the rest)	

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

0x34567: 3 4 5 6 7
 0011 0100 0101 0110 0111

bits 0-5 (all offsets): 100111 = 0x27

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

0x34567: 3 4 5 6 7
 0011 0100 0101 0110 0111

bits 0-5 (all offsets): **100111** = 0x27

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

0x34567: 3 4 5 6 7
 0011 0100 0101 0110 0111

bits 0-5 (all offsets): 100111 = 0x27

L1:

bits 6-11 (L1 set): 01 0101 = 0x15

bits 12- (L1 tag): 0x34

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

0x34567: 3 4 5 6 7
 0011 0100 0101 0110 0111

bits 0-5 (all offsets): 100111 = 0x27

L1:

bits 6-11 (L1 set): 01 0101 = 0x15

bits 12- (L1 tag): 0x34

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

0x34567: 3 4 5 6 7
 0011 0100 0101 0110 0111

bits 0-5 (all offsets): 100111 = 0x27

L2:

bits 6-15 (set for L2): 01 0001 0101 = 0x115

bits 16-: 0x3

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

0x34567: 3 4 5 6 7
 0011 0100 0101 0110 0111

bits 0-5 (all offsets): 100111 = 0x27

L2:

bits 6-15 (set for L2): 01 0001 0101 = 0x115

bits 16-: 0x3

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

0x34567: 3 4 5 6 7
 0011 0100 0101 0110 0111

bits 0-5 (all offsets): 100111 = 0x27

L3:

bits 6-18 (set for L3): 0 1101 0001 0101 = 0xD15

bits 18-: 0x0

replacement policies

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
000	
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	hit
01100100 (64)	miss

how to decide where to insert 0x64?

replacement policies

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value	LRU
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]	1
1	1	011000	mem[0x62] mem[0x63]	0			1

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	hit
01100100 (64)	miss

track which block was read least recently updated on every access

example replacement policies

least recently used

take advantage of **temporal locality**

at least $\lceil \log_2(E!) \rceil$ bits per set for E -way cache

(need to store order of all blocks)

approximations of least recently used

implementing least recently used is expensive — lots of bookkeeping
bits+time

really just need “avoid recently used” — much faster/simpler

good approximations: E to $2E$ bits

first-in, first-out

counter per set — where to replace next

(pseudo-)random

no extra information!

actually works pretty well in practice

cache miss types

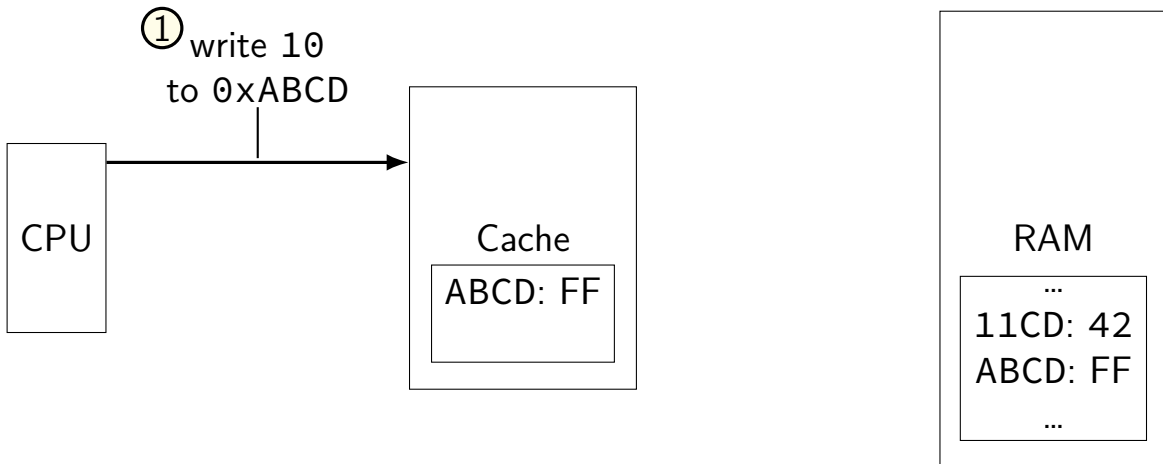
compulsory (or *cold*) — **first time** accessing something
adding more sets or blocks/set wouldn't change

conflict — sets aren't big/flexible enough
a fully-associative (1-set) cache of the same size would have done better

capacity — cache was not big enough

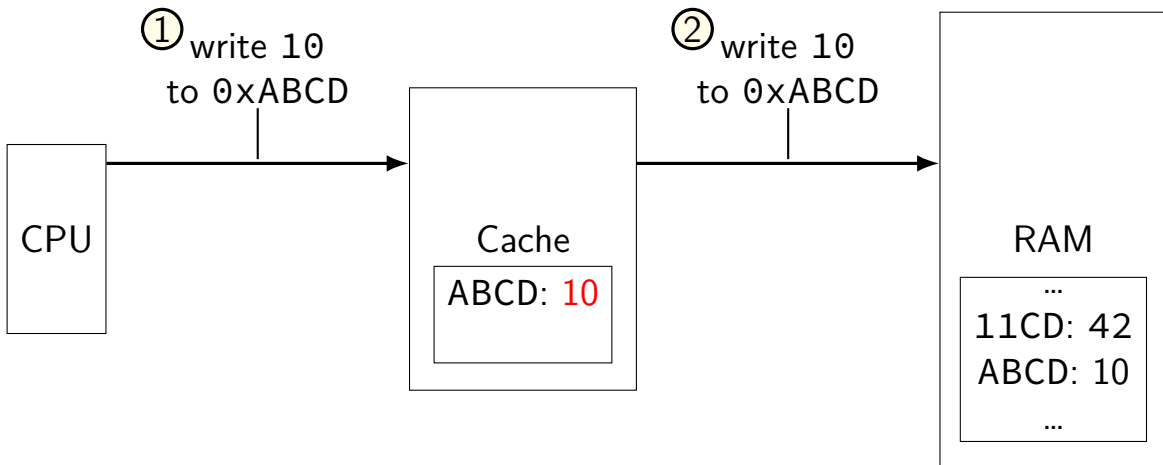
write-through v. write-back

option 1: write-through



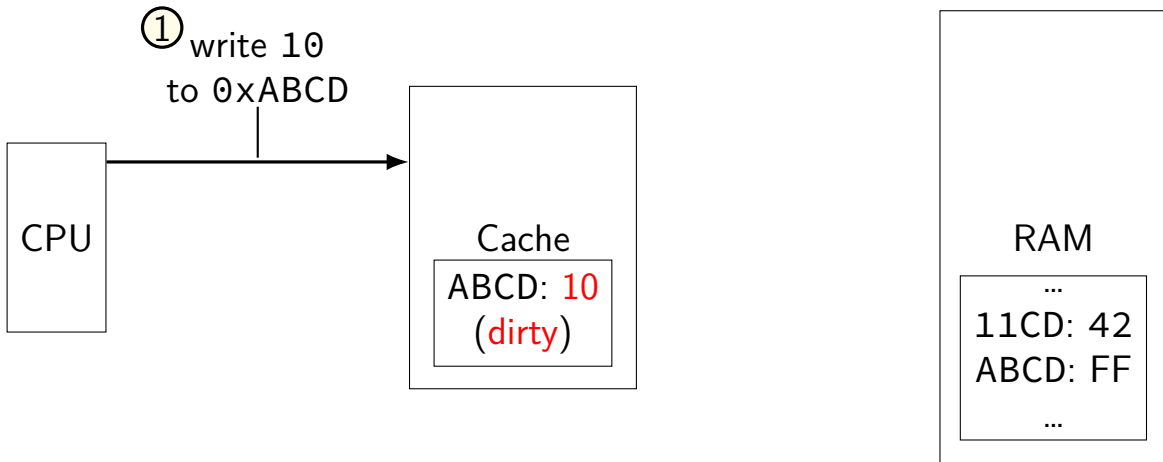
write-through v. write-back

option 1: write-through



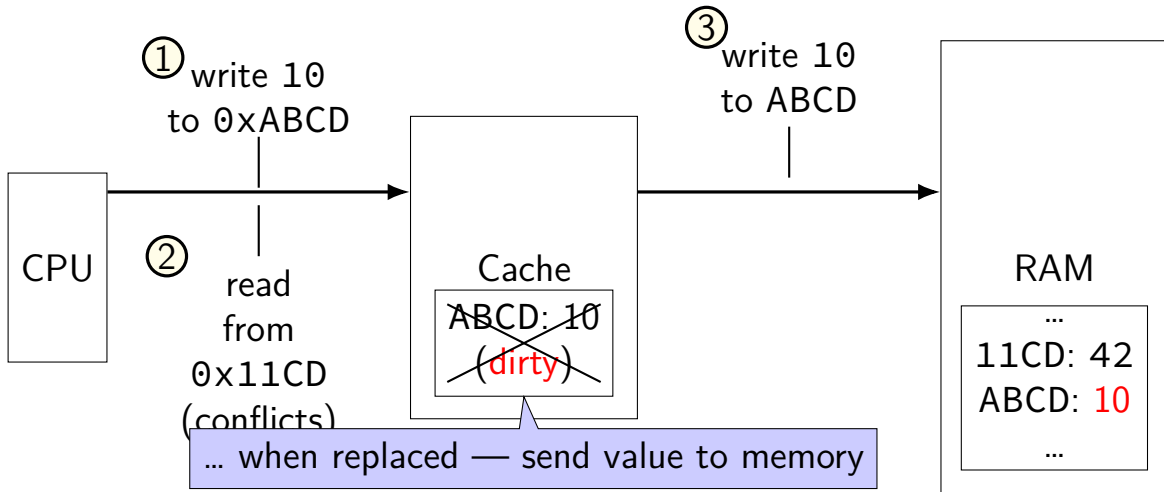
write-through v. write-back

option 2: write-back

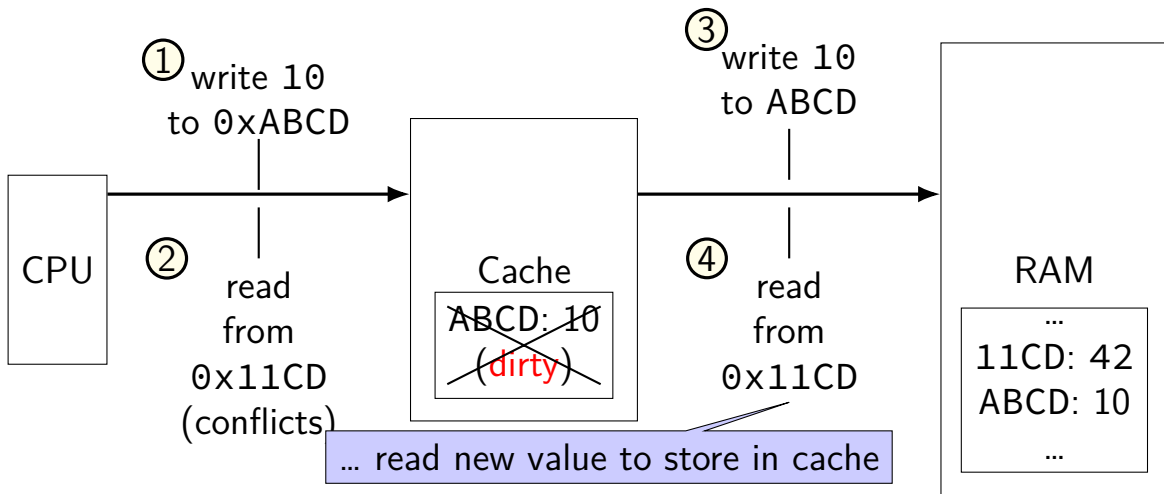


write-through v. write-back

option 2: write-back



write-through v. write-back



writeback policy

changed value!

2-way set associative, 4 byte blocks, 2 sets

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	000000	mem[0x00] mem[0x01]	0	1	011000	mem[0x60]* mem[0x61]*	1	1
1	1	011000	mem[0x62] mem[0x63]	0	0				0

1 = dirty (different than memory)
needs to be written if evicted

allocate on write?

processor writes **less than whole** cache block

block not yet in cache

two options:

write-allocate

fetch rest of cache block, replace written part

write-no-allocate

send write through to memory

guess: not read soon?

write-allocate

2-way set associative, LRU, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	000000	mem[0x00] mem[0x01]	0	1	011000	mem[0x60]* mem[0x61]*	1	1
1	1	011000	mem[0x62] mem[0x63]	0	0				0

writing 0xFF into address 0x04?

index 0, tag 000001

write-allocate

2-way set associative, LRU, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	000000	mem[0x00] mem[0x01]	0	1	011000	mem[0x60]* mem[0x61]*	1	1
1	1	011000	mem[0x62] mem[0x63]	0	0				0

writing $\hat{0}\text{xFF}$ into address $0\text{x}04$?

index 0, tag 000001

step 1: find **least recently used** block

write-allocate

2-way set associative, LRU, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	000000	mem[0x00] mem[0x01]	0	1	011000	mem[0x60]* mem[0x61]*	1	1
1	1	011000	mem[0x62] mem[0x63]	0	0				0

writing $\hat{0}\text{xFF}$ into address $0\text{x}04$?

index 0, tag 000001

step 1: find **least recently used** block

step 2: possibly writeback old block

write-allocate

2-way set associative, LRU, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	000000	mem[0x00] mem[0x01]	0	1	000001	0xFF mem[0x05]	1	0
1	1	011000	mem[0x62] mem[0x63]	0	0				0

writing 0xFF into address 0x04?

index 0, tag 000001

step 1: find **least recently used** block

step 2: possibly writeback old block

step 3a: read in new block – to get mem[0x05]

step 3b: update LRU information

write-no-allocate

2-way set associative, LRU, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	000000	mem[0x00] mem[0x01]	0	1	011000	mem[0x60]* mem[0x61]*	1	1
1	1	011000	mem[0x62] mem[0x63]	0	0				0

writing 0xFF into address 0x04?

step 1: is it in cache yet?

step 2: no, just send it to memory

exercise (1)

2-way set associative, LRU, write-allocate, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	001100	mem[0x30] mem[0x31]	0	1	010000	mem[0x40]* mem[0x41]*	1	0
1	1	011000	mem[0x62] mem[0x63]	0	1	001100	mem[0x32]* mem[0x33]*	1	1

for each of the following accesses, performed alone, would it require (a) reading a value from memory (or next level of cache) and (b) writing a value to the memory (or next level of cache)?

writing 1 byte to 0x33

reading 1 byte from 0x52

reading 1 byte from 0x50

exercise (2)

2-way set associative, LRU, write-no-allocate, write-through

index	valid	tag	value	valid	tag	value	LRU
0	1	001100	mem[0x30] mem[0x31]	1	010000	mem[0x40] mem[0x41]	0
1	1	011000	mem[0x62] mem[0x63]	1	001100	mem[0x32] mem[0x33]	1

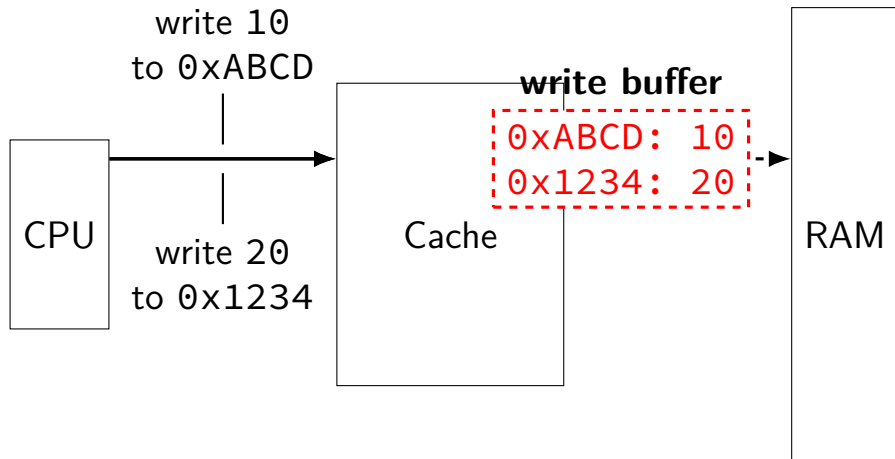
for each of the following accesses, performed alone, would it require (a) reading a value from memory and (b) writing a value to the memory?

writing 1 byte to 0x33

reading 1 byte from 0x52

reading 1 byte from 0x50

fast writes



write appears to complete immediately when placed in buffer
memory can be much slower

average memory access time

$AMAT = \text{hit time} + \text{miss penalty} \times \text{miss rate}$

effective speed of memory

AMAT exercise (1)

90% cache hit rate

hit time is 2 cycles

30 cycle miss penalty

what is the average memory access time?

suppose we could increase hit rate by increasing its size, but it would increase the hit time to 3 cycles

how much do we have to increase the hit rate for this to not increase AMAT?

AMAT exercise (1)

90% cache hit rate

hit time is 2 cycles

30 cycle miss penalty

what is the average memory access time?

5 cycles

suppose we could increase hit rate by increasing its size, but it would increase the hit time to 3 cycles

how much do we have to increase the hit rate for this to not increase AMAT?

AMAT exercise (1)

90% cache hit rate

hit time is 2 cycles

30 cycle miss penalty

what is the average memory access time?

5 cycles

suppose we could increase hit rate by increasing its size, but it would increase the hit time to 3 cycles

how much do we have to increase the hit rate for this to not increase AMAT?

miss rate of $2/30 \rightarrow$ approx 93% hit rate

backup slides

cache organization and miss rate

depends on program; one example:

SPEC CPU2000 benchmarks, 64B block size

LRU replacement policies

data cache miss rates:

Cache size	direct-mapped	2-way	8-way	fully assoc.
1KB	8.63%	6.97%	5.63%	5.34%
2KB	5.71%	4.23%	3.30%	3.05%
4KB	3.70%	2.60%	2.03%	1.90%
16KB	1.59%	0.86%	0.56%	0.50%
64KB	0.66%	0.37%	0.10%	0.001%
128KB	0.27%	0.001%	0.0006%	0.0006%

cache organization and miss rate

depends on program; one example:

SPEC CPU2000 benchmarks, 64B block size

LRU replacement policies

data cache miss rates:

Cache size	direct-mapped	2-way	8-way	fully assoc.
1KB	8.63%	6.97%	5.63%	5.34%
2KB	5.71%	4.23%	3.30%	3.05%
4KB	3.70%	2.60%	2.03%	1.90%
16KB	1.59%	0.86%	0.56%	0.50%
64KB	0.66%	0.37%	0.10%	0.001%
128KB	0.27%	0.001%	0.0006%	0.0006%

building a (direct-mapped) cache

Cache

value
00 00
00 00
00 00
00 00

cache block: 2 bytes

Memory

addresses	bytes
00000-00001	00 11
00010-00011	22 33
00100-00101	55 55
00110-00111	66 77
01000-01001	88 99
01010-01011	AA BB
01100-01101	CC DD
01110-01111	EE FF
10000-10001	F0 F1
...	...

building a (direct-mapped) cache

read byte at 01011?

Cache

value
00 00
00 00
00 00
00 00

cache block: 2 bytes

Memory

addresses	bytes
00000-00001	00 11
00010-00011	22 33
00100-00101	55 55
00110-00111	66 77
01000-01001	88 99
01010-01011	AA BB
01100-01101	CC DD
01110-01111	EE FF
10000-10001	F0 F1
...	...

building a (direct-mapped) cache

read byte at 01011?

exactly **one place** for each address
spread out what can go in a block

Cache

Memory

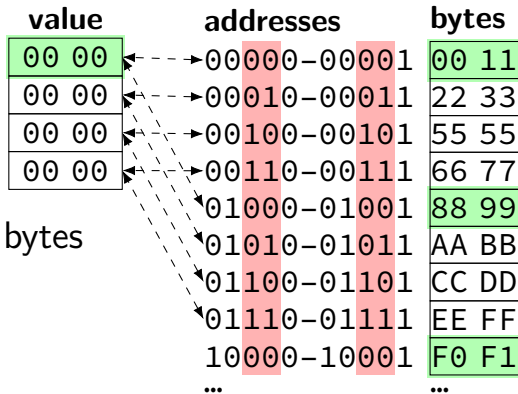
index

00

01

10

11



cache block: 2 bytes

direct-mapped

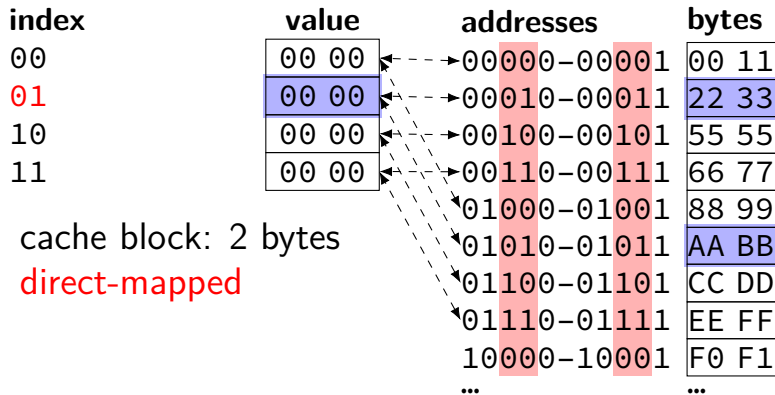
building a (direct-mapped) cache

read byte at 01011?

exactly **one place** for each address
spread out what can go in a block

Cache

Memory



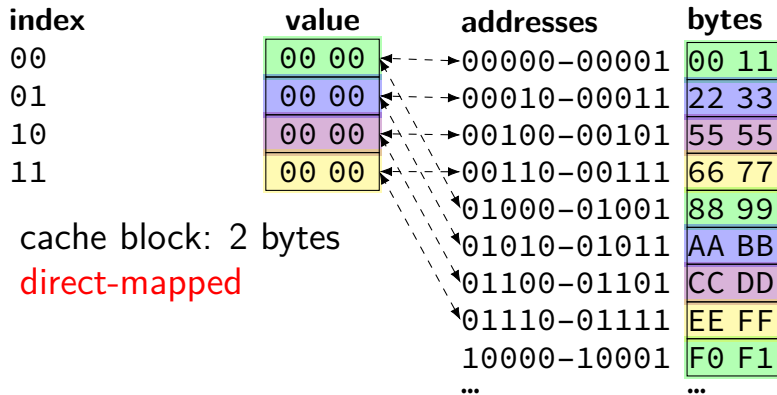
building a (direct-mapped) cache

read byte at 01011?

exactly **one place** for each address
spread out what can go in a block

Cache

Memory



building a (direct-mapped) cache

read byte at 01011?

Cache			Memory	
index	valid	value	addresses	bytes
00	0	00 00		1
01	0	00 00	00010-00011	22 33
10	0	00 00	0-00101	55 55
11	0	00 00	00110-00111	66 77
			01000-01001	88 99
			01010-01011	AA BB
			01100-01101	CC DD
			01110-01111	EE FF
			10000-10001	F0 F1
		

cache block: 2 bytes
direct-mapped

is this even a value?

need extra bit to know

building a (direct-mapped) cache

read byte at 01011?

invalid, fetch

Cache

index	valid	value
00	0	00 00
01	1	AA BB
10	0	00 00
11	0	00 00

cache block: 2 bytes

direct-mapped

Memory

addresses	bytes
00000-00001	00 11
00010-00011	22 33
00100-00101	55 55
00110-00111	66 77
01000-01001	88 99
01010-01011	AA BB
01100-01101	CC DD
01110-01111	EE FF
10000-10001	F0 F1
...	...

building a (direct-mapped) cache

read byte at 01011?

invalid, fetch

Cache				Memory	
index	valid	tag	value	addresses	bytes
00	0	00	00 00	00000-00001	00 11
01	1	01	AA BB	00010-00011	22 33
10	0	00	00 00	00100-00101	55 55
11	0			00110-00111	66 77
				01000-01001	88 99
				01010-01011	AA BB
				01100-01101	CC DD
				01110-01111	EE FF
				10000-10001	F0 F1
			

value from 01010 or 00010?

need tag to know

cache block: 2 bytes
direct-mapped

building a (direct-mapped) cache

read byte at 01011?

invalid, fetch

Cache

index	valid	tag	value
00	0	00	00 00
01	1	01	AA BB
10	0	00	00 00
11	0	00	00 00

cache block: 2 bytes

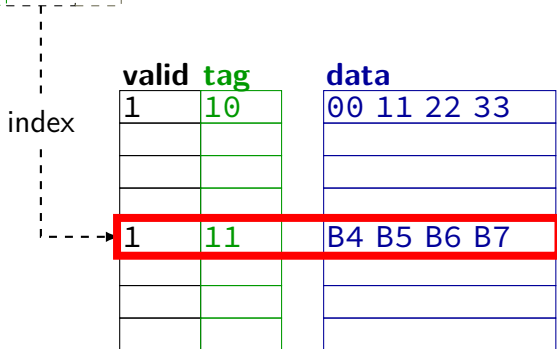
direct-mapped

Memory

addresses	bytes
00000-00001	00 11
00010-00011	22 33
00100-00101	55 55
00110-00111	66 77
01000-01001	88 99
01010-01011	AA BB
01100-01101	CC DD
01110-01111	EE FF
10000-10001	F0 F1
...	...

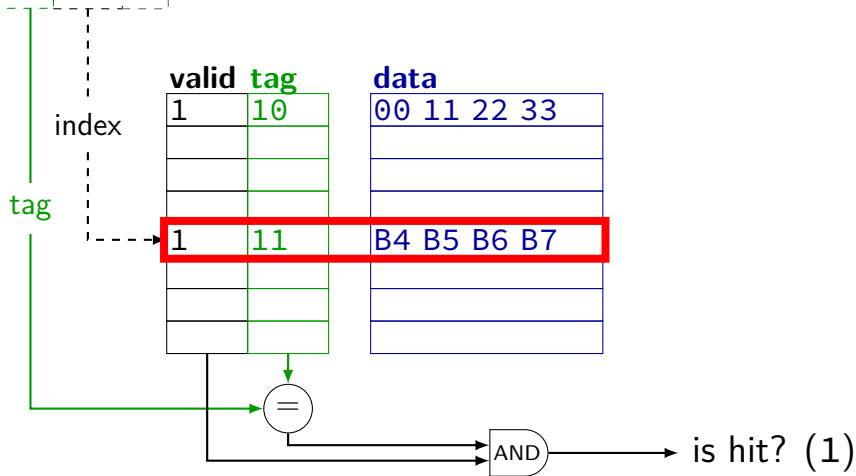
cache operation (read)

0b1110010

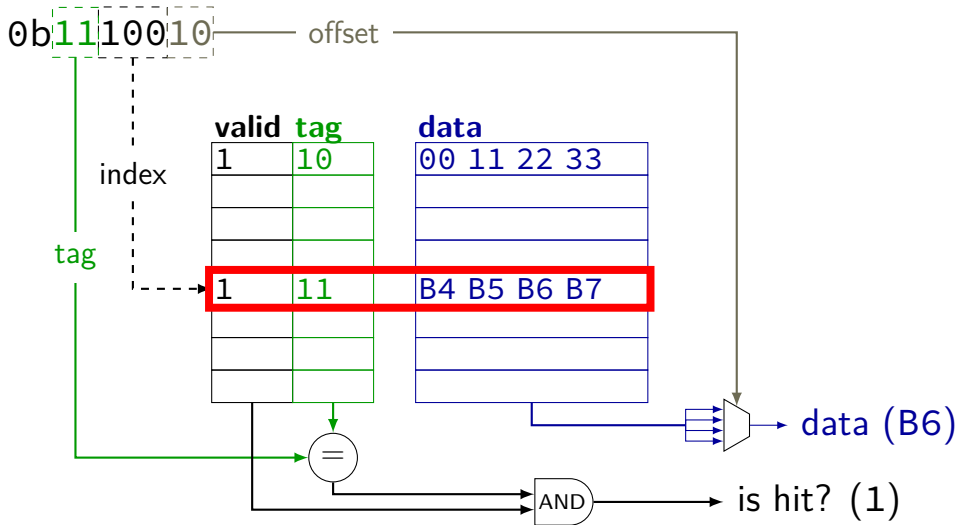


cache operation (read)

0b1110010



cache operation (read)



adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	0			0		
1	0			0		

multiple places to put values with same index
avoid conflict misses

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	0		set 0	0		
1	0		set 1	0		

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	0			0		
1	0			0		

Diagram illustrating a 2-way set associative cache structure with 2 byte blocks and 2 sets. The cache is divided into two sets, labeled "way 0" and "way 1". Each set contains two entries, indexed 0 and 1. The "valid" bit for all entries is 0, indicating they are currently invalid. The "tag" and "value" fields are empty.

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	0			0		
1	0			0		

$m = 8$ bit addresses

$S = 2 = 2^s$ sets

$s = 1$ (set) index bits

$B = 2 = 2^b$ byte block size

$b = 1$ (block) offset bits

$t = m - (s + b) = 6$ tag bits

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	0		
1	0			0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

tag indexoffset

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	0		
1	0			0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

tag indexoffset

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	0		
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

tag indexoffset

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	
00000000 (00)	
01100100 (64)	

tag indexoffset

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	
01100100 (64)	

tag indexoffset

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	hit
01100100 (64)	

tag index offset

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	hit
01100100 (64)	miss

needs to replace block in set 0!

tag indexoffset

adding associativity

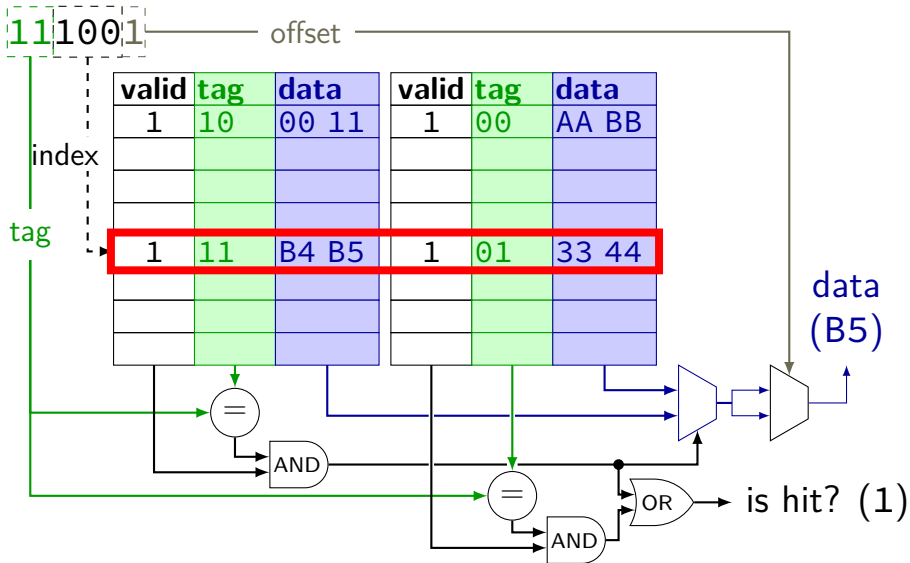
2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

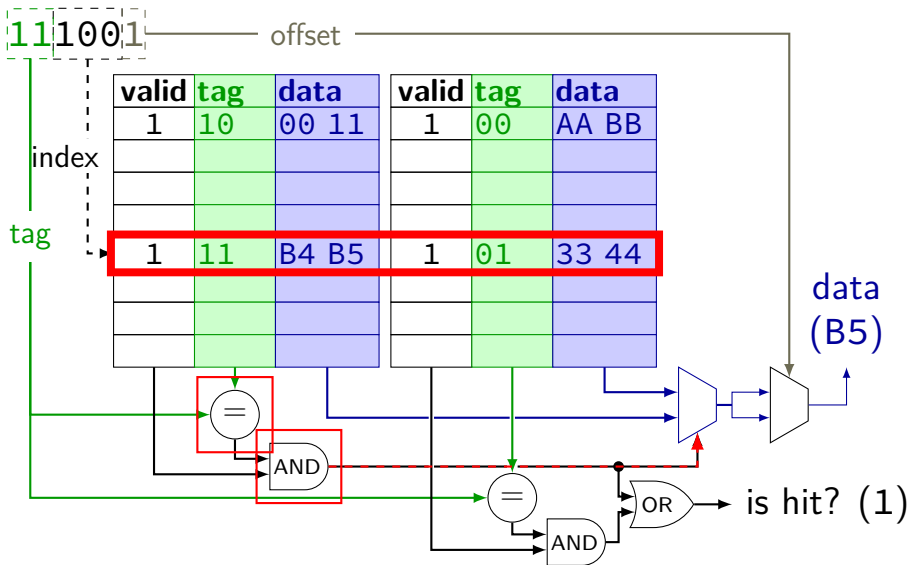
address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	hit
01100100 (64)	miss

tag indexoffset

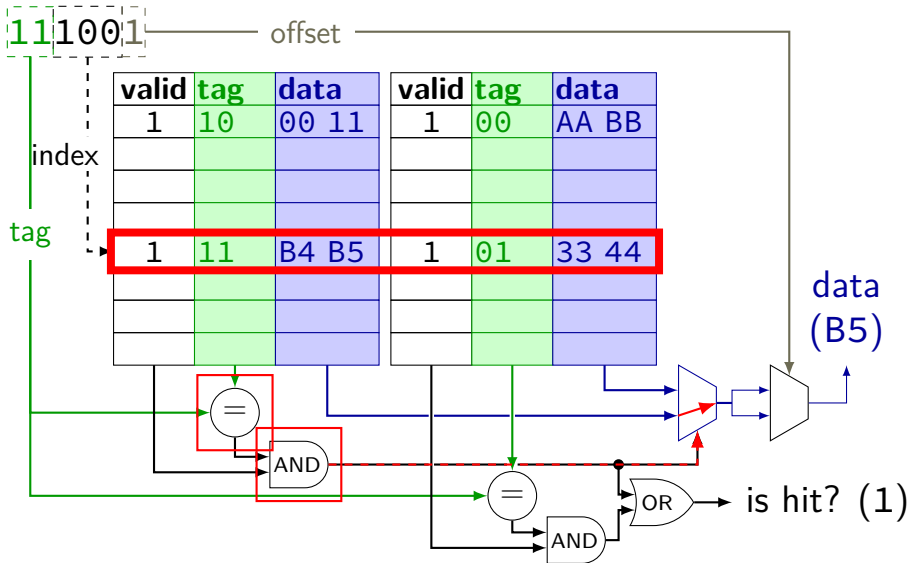
cache operation (associative)



cache operation (associative)



cache operation (associative)



associative lookup possibilities

none of the blocks for the index are valid

none of the valid blocks for the index match the tag
something else is stored there

one of the blocks for the index is valid and matches the tag