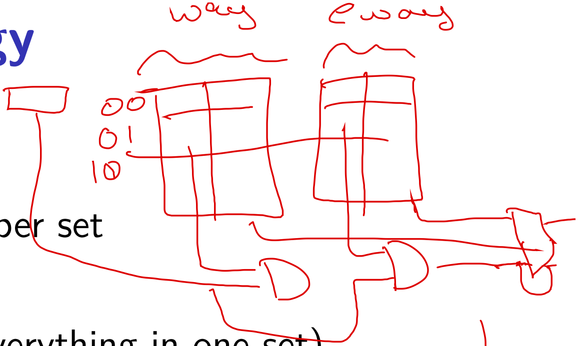
Caching

associativity terminology

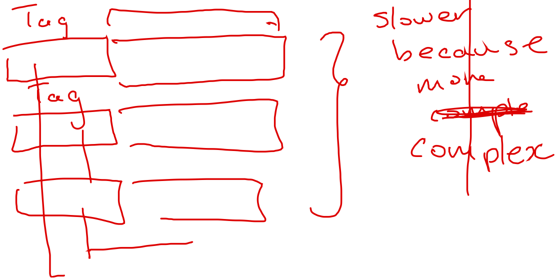


direct-mapped — one block per set

E-way set associative — E blocks per set

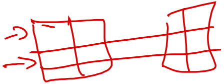
E ways in the cache

fully associative — one set total (everything in one set)



Tag-Index-Offset formulas (complete)

m	memory addreses 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)



Tag-Index-Offset exercise

m	memory addreses 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)	$B = 64$Byte
	$B = 2^b$ (b: block offset bits)

$\log_2(64) = 6$

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64$Byte
	$B = 2^b$ (b: 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)	$C = 32\text{KB} = E \times B \times S$

$$C = E \times B \times S = \frac{C}{E \times B}$$

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)	$C = 32\text{KB} = E \times B \times S$
	$S = \dfrac{C}{B \times E}$ (S: number of sets)
number of sets	$S = \dfrac{32\text{KB}}{64\text{Byte} \times 8} = 64$

$$KB = 2^{10}$$

$$\frac{32 \times 2^{10}}{64 \times 8} = \frac{2^5 \times 2^{10}}{2^6 \times 2^3}$$

$$\frac{2^{15}}{2^9} = 2^6 = 64$$

T-I-O exercise: L1

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

$\log_2(64) = 6$

$2^6 = 64$

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)		

10 A
11 B
12 C
13 D

`0x34567`:

```
   3    4    5    6    7
0 0011 0100 0101 0110 0111
```

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

L3:

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

13 1 5

bits 18-: `0x0`

cache miss types

compulsory (or *cold*) — first time accessing something

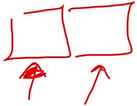
Never accessed before

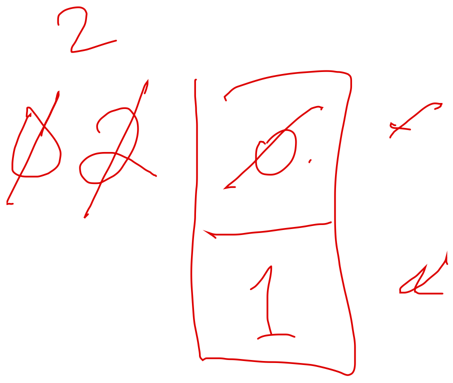
- doesn't matter how big/flexible the cache is

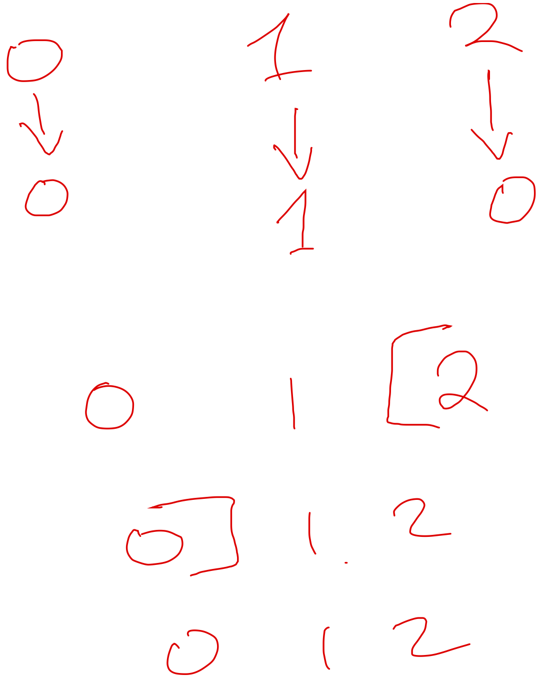
conflict — sets aren't big/flexible enough

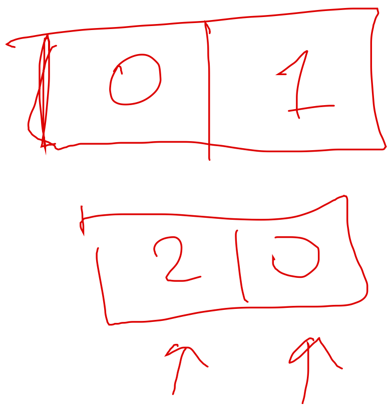
- a fully-associtive (1-set) cache of the same size would have done better

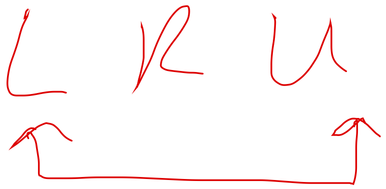
capacity — cache was not big enough

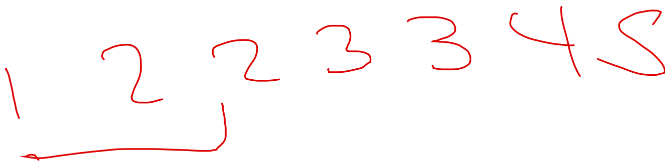









LRU


1 2 2 3 3 4 5

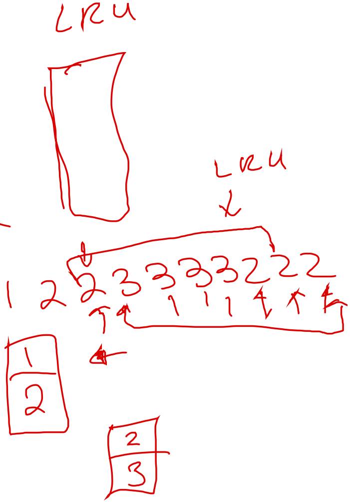
replacement policies

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[illegible]	[illegible]
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

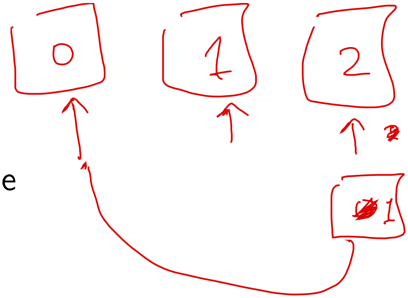
2-way set associative, 2 byte blocks, 2 sets

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)	mi
00000001 (01)	hit
01100011 (63)	mi
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

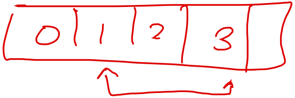


least recently used and approximations

- take advantage of temporal locality
- exact: $\lceil \log_2(E!) \rceil$ bits per set for E-way cache
- good approximations: E to $2E$ bits

first-in, first-out

- counter per set — where to replace next



(pseudo-)random

- no extra information!

exercise

4 byte blocks, 2 sets

index	V	tag	value	V	tag	value	LRU
0	0			0			
1	0			0			

address (hex)	hit?
00000000 (00)	
00000001 (01)	
00001010 (0A)	
00100001 (21)	
00001100 (0C)	
00000011 (02)	
00100011 (23)	

exercise

4 byte blocks, 2 sets

index	V	tag	value	V	tag	value	LRU
0	0			0			
1	0			0			

address (hex)	hit?
00000000 (00)	
00000001 (01)	
00001010 (0A)	
00100001 (21)	
00001100 (0C)	
00000011 (02)	
00100011 (23)	

how is the address 21 (00100001) split up into tag/index/offset?

b block offset bits;
$B = 2^b$ byte block size;
s set index bits; $S = 2^s$ sets;
$t = m - (s + b)$ tag bits (leftover)

exercise

4 byte blocks, 2 sets

index	V	tag	value	V	tag	value	LRU
0	0			0			
1	0			0			

address (hex)	hit?
00000000 (00)	
00000001 (01)	
00001010 (0A)	
00100001 (21)	
00001100 (0C)	
00000011 (02)	
00100011 (23)	

tag index offset

exercise

4 byte blocks, 2 sets

index	V	tag	value	V	tag	value	LRU
0	0			0			
1	0			0			

address (hex)	hit?
00000000 (00)	
00000001 (01)	
00001010 (0A)	
00100001 (21)	
00001100 (0C)	
00000011 (02)	
00100011 (23)	

tag index offset

exercise: how many accesses are hits?
what is the final state of the cache?

exercise

4 byte blocks, 2 sets

index	V	tag	value	V	tag	value	LRU
0	1	00000	M[0x00] M[0x01] M[0x02] M[0x03]	0			way 1
1	0			0			

address (hex)	hit?
00000000 (00)	miss
00000001 (01)	
00001010 (0A)	
00100001 (21)	
00001100 (0C)	
00000011 (02)	
00100011 (23)	

tag index offset

exercise: how many accesses are hits?
what is the final state of the cache?

exercise

4 byte blocks, 2 sets

index	V	tag	value	V	tag	value	LRU
0	1	00000	M[0x00] M[0x01] M[0x02] M[0x03]	1	00001	M[0x08] M[0x09] M[0x0A] M[0x0B]	way 0
1	0			0			

address (hex)	hit?
00000000 (00)	miss
00000001 (01)	hit
00001010 (0A)	miss
00100001 (21)	
00001100 (0C)	
00000011 (02)	
00100011 (23)	

tag index offset

exercise

4 byte blocks, 2 sets

index	V	tag	value	V	tag	value	LRU
0	1	00100	M[0x20] M[0x21] M[0x22] M[0x23]	1	00001	M[0x08] M[0x09] M[0x0A] M[0x0B]	way 1
1	0			0			

address (hex)	hit?
00000000 (00)	miss
00000001 (01)	hit
00001010 (0A)	miss
00100001 (21)	miss
00001100 (0C)	miss
00000011 (02)	
00100011 (23)	

tag index offset

exercise

4 byte blocks, 2 sets

index	V	tag	value	V	tag	value	LRU
0	1	00100	M[0x20] M[0x21] M[0x22] M[0x23]	1	00000	M[0x00] M[0x01] M[0x02] M[0x03]	way 0
1	1	00000	M[0x0C] M[0x0D] M[0x0E] M[0x0F]	0			way 1

address (hex)	hit?
00000000 (00)	miss
00000001 (01)	hit
00001010 (0A)	miss
00100001 (21)	miss
00001100 (0C)	miss
00000011 (02)	miss
00100011 (23)	

tag index offset

exercise

4 byte blocks, 2 sets

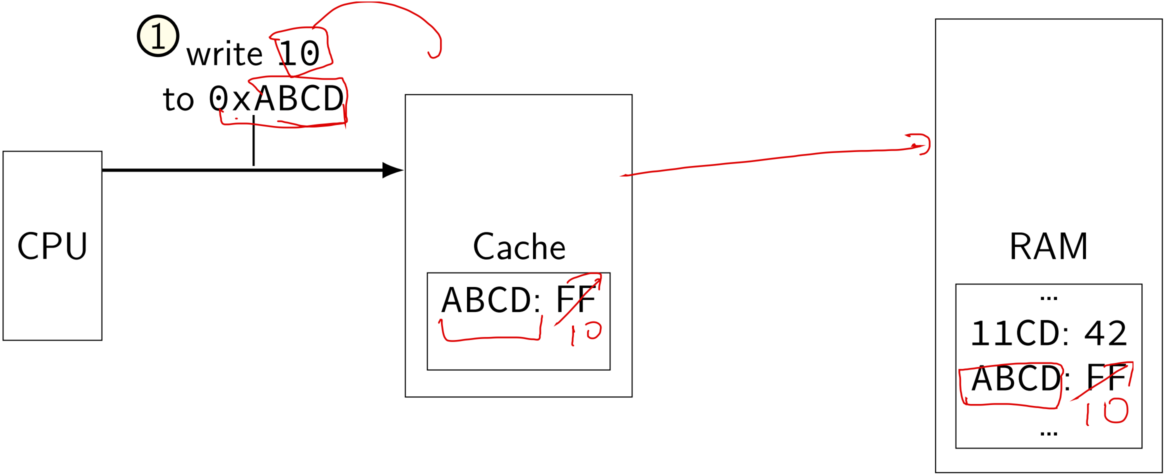
index	V	tag	value	V	tag	value	LRU
0	1	00100	M[0x20] M[0x21] M[0x22] M[0x23]	1	00000	M[0x00] M[0x01] M[0x02] M[0x03]	way 1
1	1	00000	M[0x0C] M[0x0D] M[0x0E] M[0x0F]	0			way 1

address (hex)	hit?
00000000 (00)	miss
00000001 (01)	hit
00001010 (0A)	miss
00100001 (21)	miss
00001100 (0C)	miss
00000011 (02)	miss
00100011 (23)	hit

tag index offset

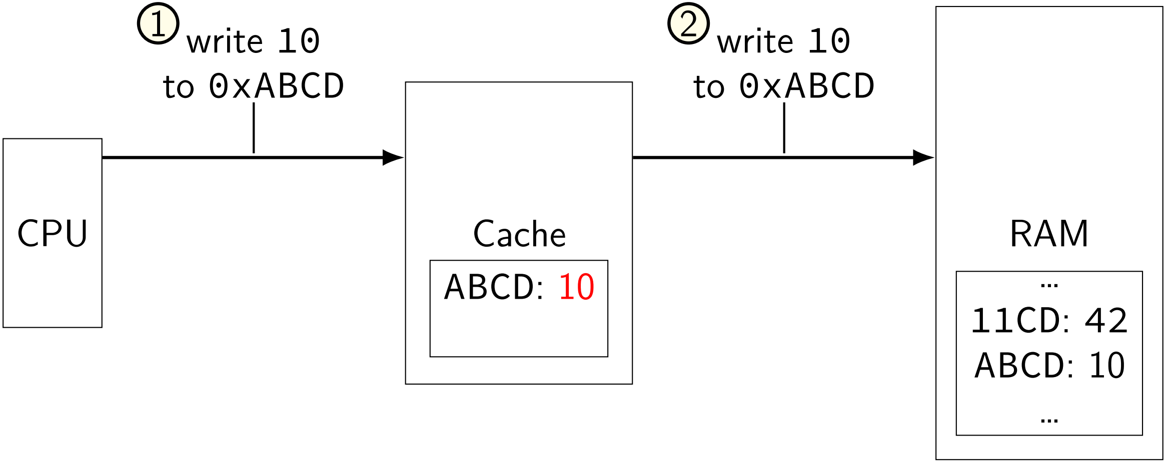
write-through v. write-back

option 1: write-through



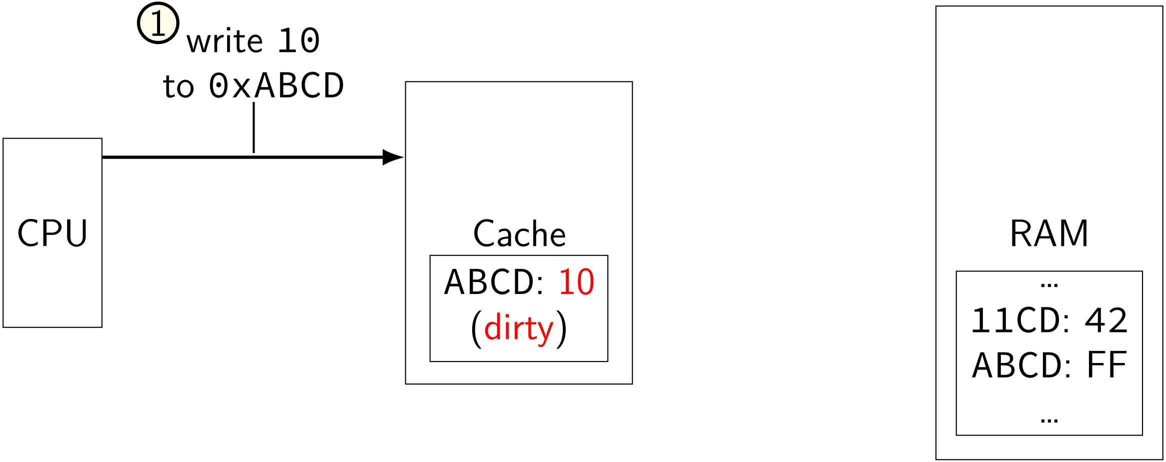
write-through v. write-back

option 1: write-through



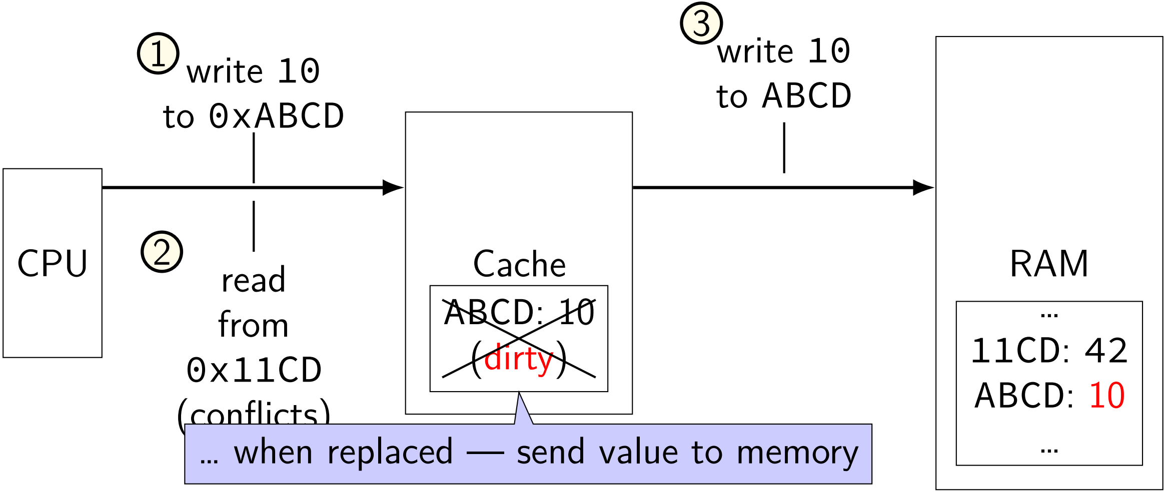
write-through v. write-back

option 2: write-back

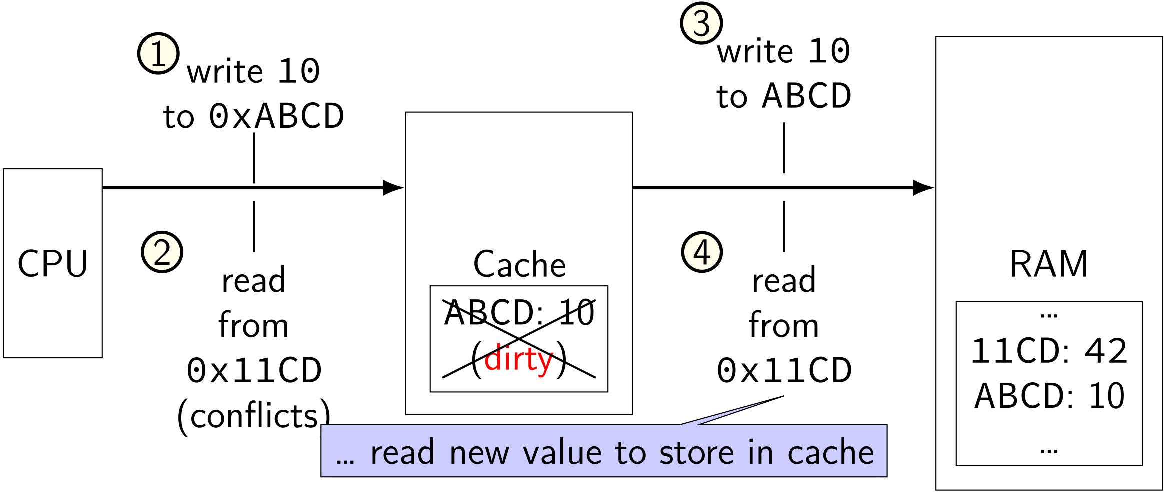


write-through v. write-back

option 2: write-back



write-through v. write-back



writeback policy

changed value!

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

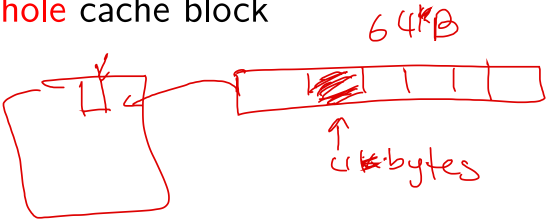
index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	000000	mem[0x00] mem[0x01]	0	1	011000	mem[0x60]* mem[0x61]*	1	1
1	1	011000	mem[0x62] mem[0x63]	0	0				0

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

allocate on write?

processor writes less than whole cache block

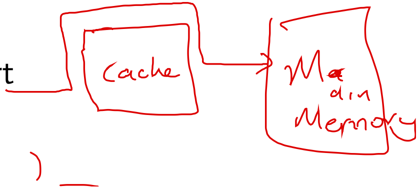
block not yet in cache



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 0xFF into address 0x04?

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 0xFF into address `0x04`?

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	011000	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

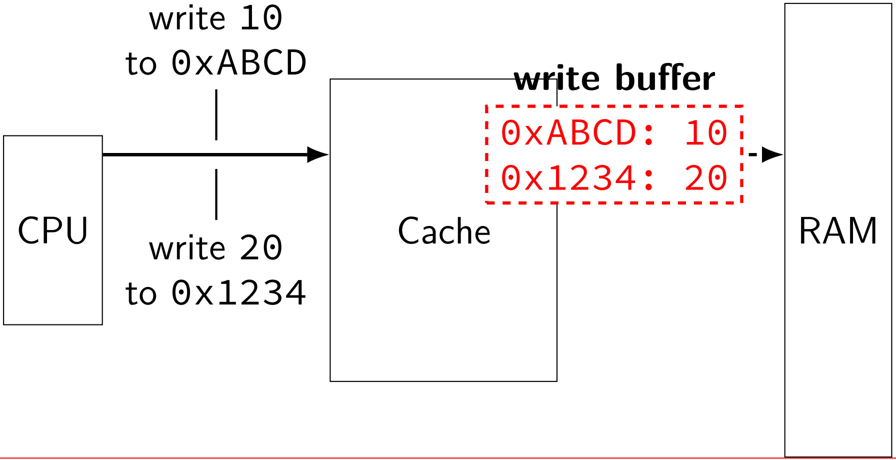
index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	000000	mem[0x00] mem[0x01]	0	1	011000	mem[0x60]* mem[0x61]*	1	1
1	1	011000	mem[0x62] mem[0x63]	0	0				0

writing 0xFF into address `0x04`?

step 1: is it in cache yet?

step 2: no, just send it to memory

fast writes



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

cache organization and miss rate

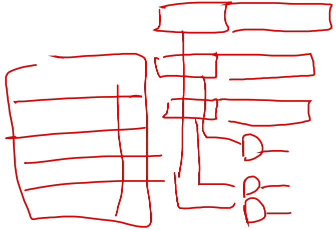
depends on program; one example:

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%



Data: Cantin and Hill, "Cache Performance for SPEC CPU2000 Benchmarks"
http://research.cs.wisc.edu/multifacet/misc/spec2000cache-data/

cache organization and miss rate

depends on program; one example:

SPEC CPU2000 benchmarks, 64B block size

LRU replacement policies

data cache miss rates:

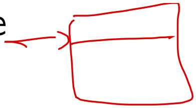
Cache size	direct-mapped	2-way	8-way	fully assoc.
1KB	8.63%	6.97%	5.63%	5.34%
2KB	5.71%	4.23%	3.30%	3.05%
4KB	3.70%	2.60%	2.03%	1.90%
16KB	1.59%	0.86%	0.56%	0.50%
64KB	0.66%	0.37%	0.10%	0.001%
128KB	0.27%	0.001%	0.0006%	0.0006%

Data: Cantin and Hill, "Cache Performance for SPEC CPU2000 Benchmarks"
http://research.cs.wisc.edu/multifacet/misc/spec2000cache-data/

reasoning about cache performance

hit time: time to lookup and find value in cache

L1 cache — typically 1 cycle?



miss rate: portion of hits (value in cache)

percentage of misses that you get in ca

miss penalty: extra time to get value if there's a miss

time to access next level cache or memory

hit time + MR × m

miss time: hit time + miss penalty

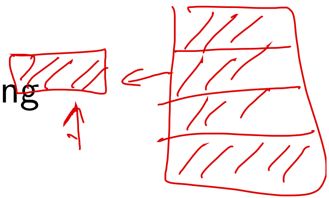
average memory access time

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

effective speed of memory

making any cache look bad

1. access enough blocks, to fill the cache
2. access an additional block, replacing something
3. access last block replaced
4. access last block replaced
5. access last block replaced



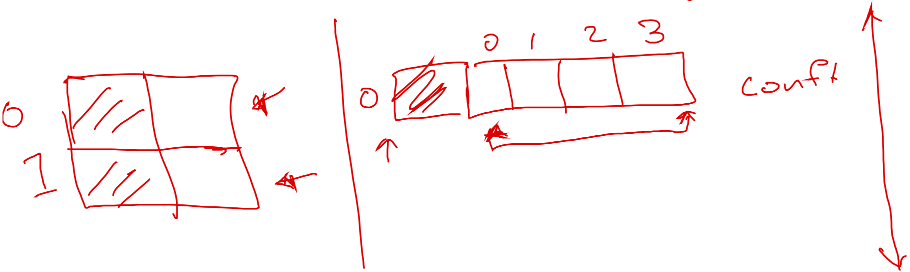
…

but — typical real programs have locality

cache optimizations by miss type

	capacity	conflict	compulsory
increase cache size	fewer misses	fewer misses	—
increase associativity	—	fewer misses	—
increase block size	—	more misses	fewer misses

(assuming other listed parameters remain constant)



C and cache misses (1)

```
int array[1024]; // 4KB array
int even_sum = 0, odd_sum = 0;
for (int i = 0; i < 1024; i += 2) {
    even_sum += array[i + 0];
    odd_sum +=  array[i + 1];
}
```

Assume everything but `array` is kept in registers (and the compiler does not do anything funny).

How many *data cache misses* on a 2KB direct-mapped cache with 16B cache blocks?

C and cache misses (2)

```
int array[1024]; // 4KB array
int even_sum = 0, odd_sum = 0;
for (int i = 0; i < 1024; i += 2)
    even_sum += array[i + 0];
for (int i = 1; i < 1024; i += 2)
    odd_sum +=  array[i + 1];
```

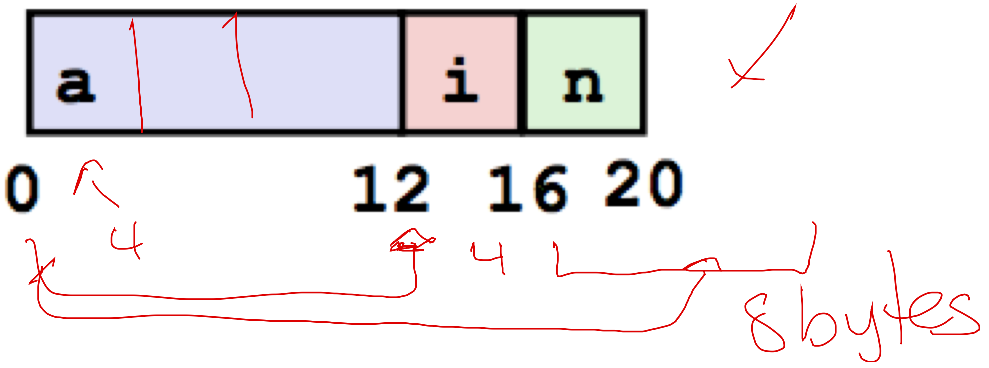
Assume everything but `array` is kept in registers (and the compiler does not do anything funny).

How many *data cache misses* on a 2KB direct-mapped cache with 16B cache blocks? Would a set-associtiave cache be better?

Structure Allocation

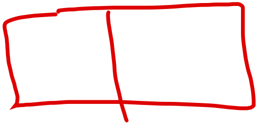
```
struct rec {
   int a[3];
   int i;
   struct rec *n;
};
```

Memory Layout



- **Concept**
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types

C and cache misses (3)



```c
typedef struct {
    int a_value, b_value;
    int boring_values[126];
} item;
item items[8]; // 4 KB array
int a_sum = 0, b_sum = 0;
for (int i = 0; i < 8; ++i)
    a_sum += items[i].a_value;
for (int i = 0; i < 8; ++i)
    b_sum += items[i].b_value;
```

Assume everything but `items` is kept in registers (and the compiler does not do anything funny).

How many *data cache misses* on a 2KB direct-mapped cache with 16B cache blocks?

backup slides