

caching 2

last time (1)

PC update logic

- book changes PC register to predicted PC register
- grabs corrections to prediction at beginning of fetch
- care needed for stalling — take into account corrections!

cache hierarchy

- fast, small memories + slow, large ones
- goal: speed of small memories + capacity of large
- assumes we can choose what to place in fast memory correctly

locality

- temporal: accessed now = accesses again soon
- spatial: accessed now = access nearby addresses soon

last time (2)

direct-mapped cache:

divide memory + cache into *blocks*

fixed power-of-two size

least sig. bits of address = location in block (“offset”)

each block of memory can be stored in exactly one cache block

power of two number of cache blocks (S)

memory block 0 to cache block 0, block 1 to 1, ..., block $S-1$ to $S-1$

block S to 0, $S+1$ to 1, ...

bits of address just before offset indicate which cache block (“index”)

metadata for each cache block

valid bit = is something here

tag = part of address not in offset + index

quiz Q1

```
addq %rax, %rcx
jle foo
rmmovq 8(%rax), %rax
subq %rcx, %rdx
halt
```

foo:

```
xorq %r8, %r9
xorq %r10, %r11
xorq %r12, %r13
xorq %r8, %r9
xorq %r10, %r11
xorq %r12, %r13
```

1	2	3	4	5	6	7
F	D1	D2	E1	E2	M	W
	F	D1	D2	E1	E2	M W
						F

F	D1	D2	E1
F	D1	D2	
	F	D1	
		F	

quiz Q2

10% cond jump

40% taken, target less (predicted correctly, +0 cycle)

20% taken, target more (predicted incorrectly, +2 cycle)

10% not taken, target less (predicted incorrectly, +2 cycle)

30% not taken, target more (predicted correctly, +0 cycle)

10% memory $\rightarrow$ execute data dependency (+1 cycle)

$$1 \text{ cycle/instr (normal)} + 10\% \cdot 20\% \cdot 2 + 10\% \cdot 10\% \cdot 2 + 10\% \cdot 1 = 1.16$$

quiz Q3

```
for (int i = 0; i < 10000; ++i) {  
    A[i] += B[i];  
    A[i] *= C[i];  
}  
std::cout << "last A is " << A[9999] << std::endl;
```

elements of $A[i]$

temporal — some, (each element) accessed a few times, then not accessed again

spatial — some, when $A[i]$ accessed $A[i+1]$, etc. accessed soon

i

temporal — excellent, accessed tens of thousands of times

spatial — irrelevant, only one thing

quiz Q4-5

```
for (int i = 0; i < 10000; ++i) {  
    A[i] += B[i];  
    A[i] *= C[i];  
}  
std::cout << "last A is " << A[9999] << std::endl;
```

machine code for loop

temporal — excellent, tens of thousands of times

spatial — good, several instructions likely at adjacent addresses

machine code for cout

temporal — nope, used only once

spatial — good, several instructions likely at adjacent addresses

quiz Q6

4 bytes/block = 2 offset bits

index	valid	tag
00	✓	001
01	✓	110
10	✓	001
11	✓	010

111 — tag 000, index 01, offset 11 — miss

11010 — tag 001, index 10 , offset 10 — hit

100010 — tag 010, index 00, offset 10 — miss

101100 — tag 010, index 11, offset 00 — hit

111000 — tag 011, index 10, offset 00 — miss

TIO: exercise

64-byte blocks, 128 set cache

stores $64 \times 128 = 8192$ bytes (of data)

if addresses 32-bits, then how many tag/index/offset bits?

which bytes are stored in the same block as byte from 0x1037?

- A. byte from 0x1011
- B. byte from 0x1021
- C. byte from 0x1035
- D. byte from 0x1041

example access pattern (1)

2 byte blocks, 4 sets

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

index	valid	tag	value
00	0		
01	0		
10	0		
11	0		

example access pattern (1)

2 byte blocks, 4 sets

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

index	valid	tag	value
00	0		
01	0		
10	0		
11	0		

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

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

$b = 1$ (block) offset bits

$t = m - (s + b) = 5$ tag bits

example access pattern (1)

2 byte blocks, 4 sets

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

tag index offset

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

index	valid	tag	value
00	0		
01	0		
10	0		
11	0		

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

$b = 1$ (block) offset bits

$t = m - (s + b) = 5$ tag bits

example access pattern (1)

2 byte blocks, 4 sets

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

tag index offset

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

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

$b = 1$ (block) offset bits

$t = m - (s + b) = 5$ tag bits

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
01	0		
10	0		
11	0		

example access pattern (1)

2 byte blocks, 4 sets

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

tag index offset

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

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

$b = 1$ (block) offset bits

$t = m - (s + b) = 5$ tag bits

index	valid	tag	value
00	1	000000	mem[0x00] mem[0x01]
01	0		
10	0		
11	0		

example access pattern (1)

2 byte blocks, 4 sets

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

tag index offset

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

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

$b = 1$ (block) offset bits

$t = m - (s + b) = 5$ tag bits

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
01	1	01100	mem[0x62] mem[0x63]
10	0		
11	0		

example access pattern (1)

2 byte blocks, 4 sets

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

tag index offset

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

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

$b = 1$ (block) offset bits

$t = m - (s + b) = 5$ tag bits

index	valid	tag	value
00	1	01100	mem[0x60] mem[0x61]
01	1	01100	mem[0x62] mem[0x63]
10	0		
11	0		

example access pattern (1)

2 byte blocks, 4 sets

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

tag index offset

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

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

$b = 1$ (block) offset bits

$t = m - (s + b) = 5$ tag bits

index	valid	tag	value
00	1	01100	mem[0x60] mem[0x61]
01	1	01100	mem[0x62] mem[0x63]
10	0		
11	0		

example access pattern (1)

2 byte blocks, 4 sets

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

tag index offset

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

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

$b = 1$ (block) offset bits

$t = m - (s + b) = 5$ tag bits

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
01	1	01100	mem[0x62] mem[0x63]
10	0		
11	0		

example access pattern (1)

2 byte blocks, 4 sets

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

tag index offset

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

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

$b = 1$ (block) offset bits

$t = m - (s + b) = 5$ tag bits

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
01	1	01100	mem[0x62] mem[0x63]
10	1	01100	mem[0x64] mem[0x65]
11	0		

example access pattern (1)

2 byte blocks, 4 sets

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

tag index offset

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

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

$b = 1$ (block) offset bits

$t = m - (s + b) = 5$ tag bits

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
01	1	01100	mem[0x62] mem[0x63]
10	1	01100	mem[0x64] mem[0x65]
11	0		

example access pattern (1)

2 byte blocks, 4 sets

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

tag index offset

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

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

$b = 1$ (block) offset bits

$t = m - (s + b) = 5$ tag bits

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
01	1	01100	mem[0x62] mem[0x63]
10	1	01100	mem[0x64] mem[0x65]
11	0		

miss caused by conflict

exercise

4 byte blocks, 4 sets

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

index	valid	tag	value
00			
01			
10			
11			

exercise

4 byte blocks, 4 sets

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

index	valid	tag	value
00			
01			
10			
11			

how is the 8-bit address 61 (01100001) 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, 4 sets

address (hex)	result
000000000 (00)	
000000001 (01)	
01100011 (63)	
01100001 (61)	
01100010 (62)	
000000000 (00)	
01100100 (64)	

index	valid	tag	value
00			
01			
10			
11			

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

$B = 4 = 2^b$ byte block size

$b = 2$ (block) offset bits

$t = m - (s + b) = 4$ tag bits

exercise

4 byte blocks, 4 sets

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

tag index offset

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

$B = 4 = 2^b$ byte block size

$b = 2$ (block) offset bits

$t = m - (s + b) = 4$ tag bits

index	valid	tag	value
00			
01			
10			
11			

exercise

4 byte blocks, 4 sets

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

tag index offset

index	valid	tag	value
00			
01			
10			
11			

exercise: which accesses are hits?

example access pattern (1)

2 byte blocks, 4 sets

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

tag index offset

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

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

$b = 1$ (block) offset bits

$t = m - (s + b) = 5$ tag bits

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
01	1	01100	mem[0x62] mem[0x63]
10	1	01100	mem[0x64] mem[0x65]
11	0		

miss caused by conflict

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		
		way 0			way 1	
1	0			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		

$m = 8$ 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 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]	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 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)	
00000000 (00)	
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)	
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

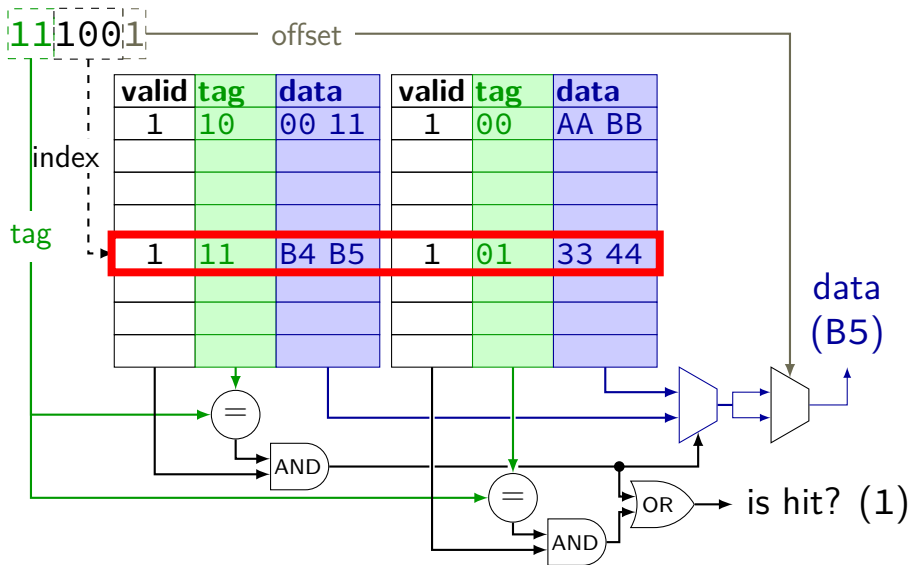
2-way set associative, 2 byte blocks, 2 sets

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		

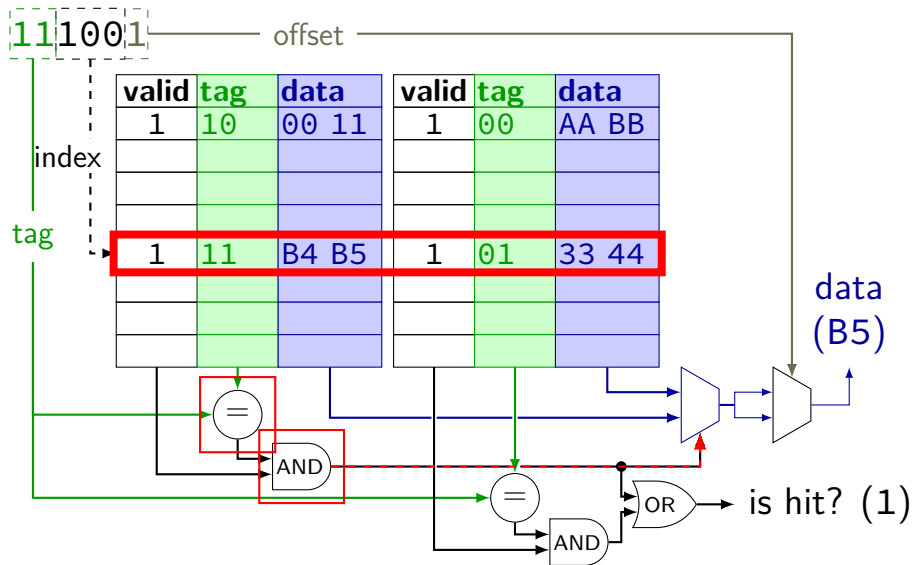
address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	hit
01100100 (64)	miss

tag index offset

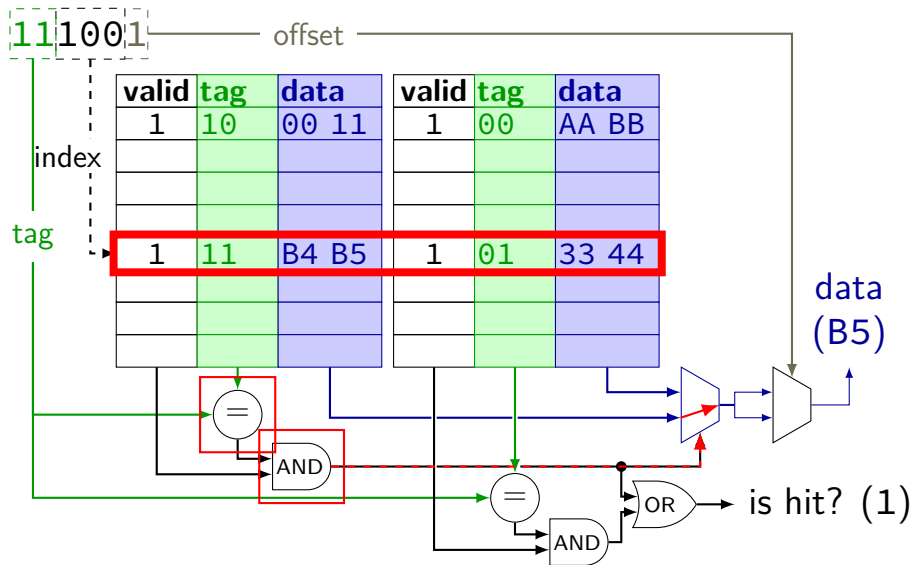
cache operation (associative)



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

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 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)

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)	$B = 64\text{Byte}$
	$B = 2^b$ (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)	$C = 32\text{KB} = E \times B \times S$

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)		

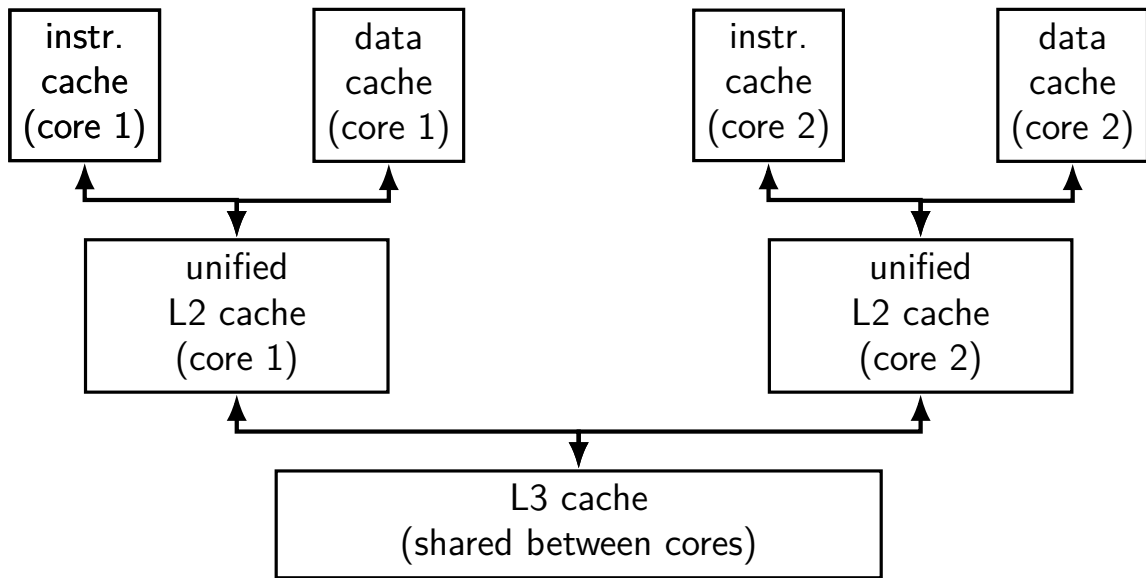
0x34567: 3 4 5 6 7
 0 011 0100 0101 0110 0111

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

L3:
bits 6-18 (set for L3): 0 1101 0001 0101 = 0xD15
bits 18-: 0x0

backup slides

split caches; multiple cores



hierarchy and instruction/data caches

typically separate data and instruction caches for L1

(almost) never going to read instructions as data or vice-versa

avoids instructions evicting data and vice-versa

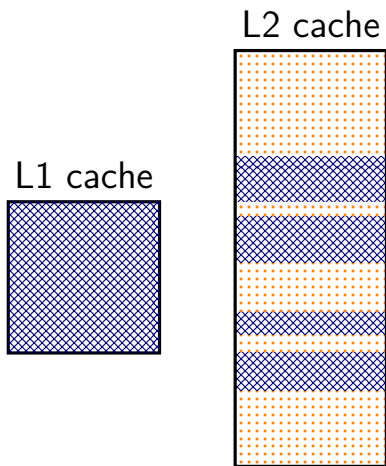
can optimize instruction cache for different access pattern

easier to build fast caches: that handles less accesses at a time

inclusive versus exclusive

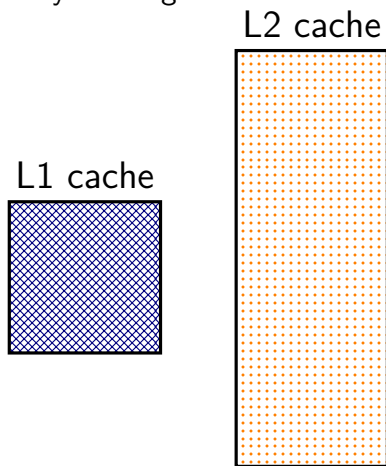
L2 inclusive of L1

everything in L1 cache duplicated in L2
adding to L1 also adds to L2



L2 exclusive of L1

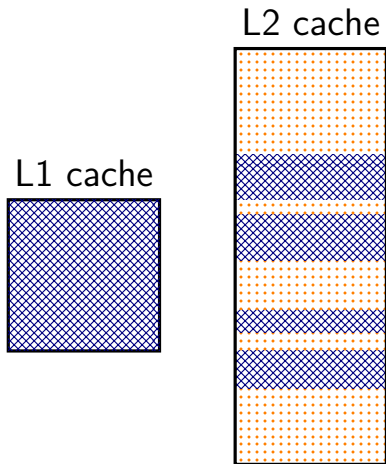
L2 contains different data than L1
adding to L1 must remove from L2
probably evicting from L1 adds to L2



inclusive versus exclusive

L2 inclusive of L1

everything in L1 cache duplicated in L2
adding to L1 also adds to L2

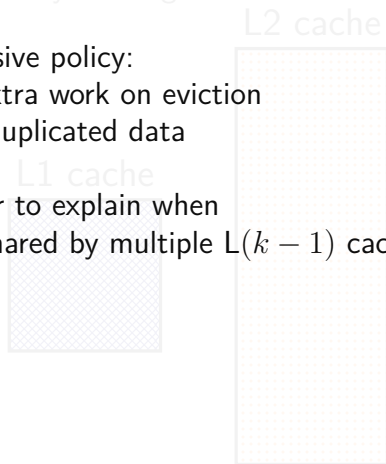


L2 exclusive of L1

L2 contains different data than L1
adding to L1 must remove from L2
probably evicting from L1 adds to L2

inclusive policy:
no extra work on eviction
but duplicated data

easier to explain when
 L_k shared by multiple $L(k-1)$ caches?



inclusive versus exclusive

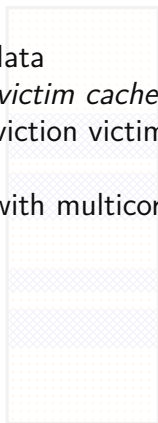
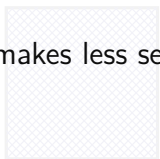
L2 inclusive of L1

everything in L1 cache duplicated in L2
adding to L1 also adds to L2

L2 cache

exclusive policy:
avoid duplicated data
sometimes called *victim cache*
(contains cache eviction victims)

makes less sense with multicore

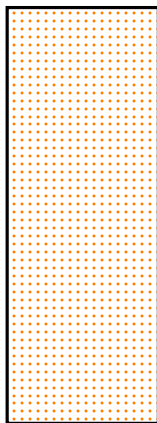
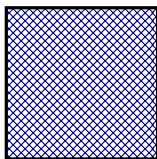


L2 exclusive of L1

L2 contains different data than L1
adding to L1 must remove from L2
probably evicting from L1 adds to L2

L2 cache

L1 cache



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%

exercise (1)

initial cache: 64-byte blocks, 64 sets, 8 ways/set

If we leave the other parameters listed above unchanged, which will probably reduce the number of **capacity misses** in a typical program? (Multiple may be correct.)

- A. quadrupling the block size (256-byte blocks, 64 sets, 8 ways/set)
- B. quadrupling the number of sets
- C. quadrupling the number of ways/set

exercise (2)

initial cache: 64-byte blocks, 8 ways/set, 64KB cache

If we leave the other parameters listed above unchanged, which will probably reduce the number of **capacity misses** in a typical program? (Multiple may be correct.)

- A. quadrupling the block size (256-byte block, 8 ways/set, 64KB cache)
- B. quadrupling the number of ways/set
- C. quadrupling the cache size

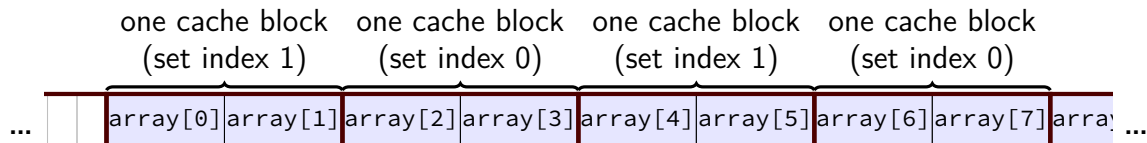
exercise (3)

initial cache: 64-byte blocks, 8 ways/set, 64KB cache

If we leave the other parameters listed above unchanged, which will probably reduce the number of **conflict misses** in a typical program? (Multiple may be correct.)

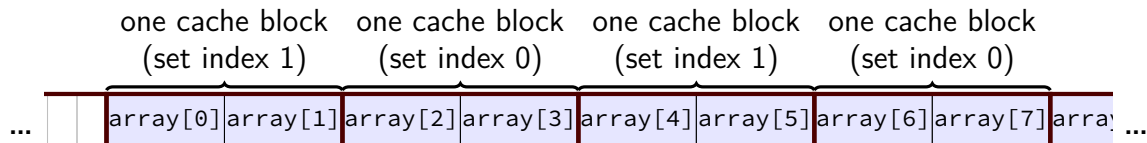
- A. quadrupling the block size (256-byte block, 8 ways/set, 64KB cache)
- B. quadrupling the number of ways/set
- C. quadrupling the cache size

quiz exercise solution



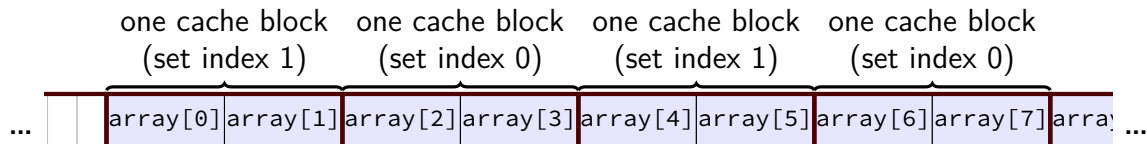
memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[3] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[6] (miss)	{array[0], array[1]}	{array[6], array[7]}
read array[1] (hit)	{array[0], array[1]}	{array[6], array[7]}
read array[4] (miss)	{array[4], array[5]}	{array[6], array[7]}
read array[7] (hit)	{array[4], array[5]}	{array[6], array[7]}
read array[2] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[5] (hit)	{array[4], array[5]}	{array[6], array[7]}
read array[8] (miss)	{array[8], array[9]}	{array[6], array[7]}

quiz exercise solution



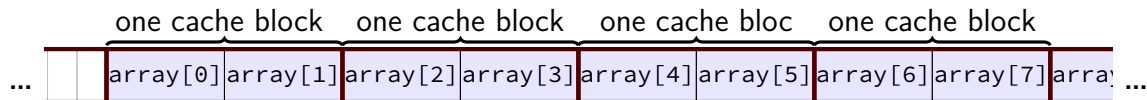
memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[3] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[6] (miss)	{array[0], array[1]}	{array[6], array[7]}
read array[1] (hit)	{array[0], array[1]}	{array[6], array[7]}
read array[4] (miss)	{array[4], array[5]}	{array[6], array[7]}
read array[7] (hit)	{array[4], array[5]}	{array[6], array[7]}
read array[2] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[5] (hit)	{array[4], array[5]}	{array[6], array[7]}
read array[8] (miss)	{array[8], array[9]}	{array[6], array[7]}

quiz exercise solution



memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[3] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[6] (miss)	{array[0], array[1]}	{array[6], array[7]}
read array[1] (hit)	{array[0], array[1]}	{array[6], array[7]}
read array[4] (miss)	{array[4], array[5]}	{array[6], array[7]}
read array[7] (hit)	{array[4], array[5]}	{array[6], array[7]}
read array[2] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[5] (hit)	{array[4], array[5]}	{array[6], array[7]}
read array[8] (miss)	{array[8], array[9]}	{array[6], array[7]}

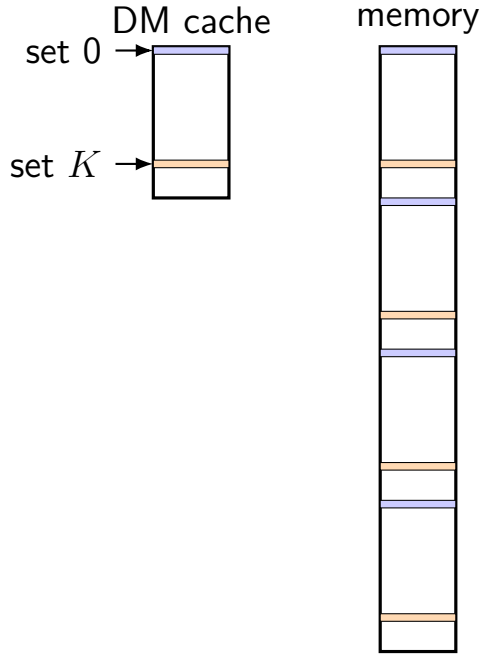
not the quiz problem



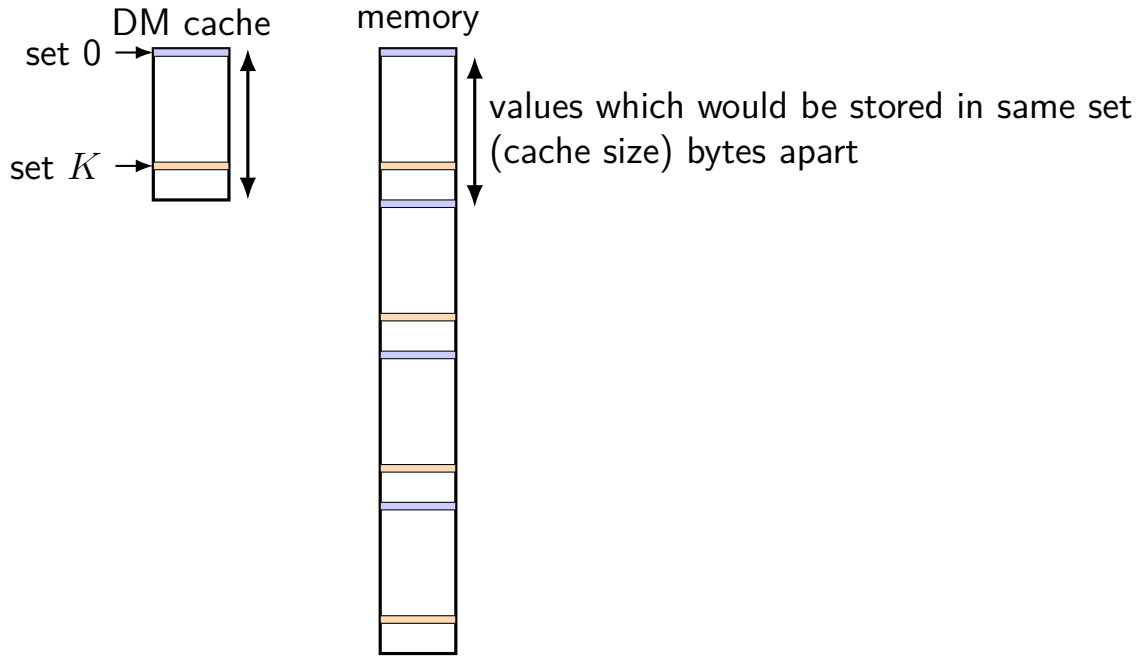
if 1-set 2-way cache instead of 2-set 1-way cache:

memory access	single set with 2-ways, LRU first
—	---, ---
read array[0] (miss)	---, {array[0], array[1]}
read array[3] (miss)	{array[0], array[1]}, {array[2], array[3]}
read array[6] (miss)	{array[2], array[3]}, {array[6], array[7]}
read array[1] (miss)	{array[6], array[7]}, {array[0], array[1]}
read array[4] (miss)	{array[0], array[1]}, {array[3], array[4]}
read array[7] (miss)	{array[3], array[4]}, {array[6], array[7]}
read array[2] (miss)	{array[6], array[7]}, {array[2], array[3]}
read array[5] (miss)	{array[2], array[3]}, {array[5], array[6]}
read array[8] (miss)	{array[5], array[6]}, {array[8], array[9]}

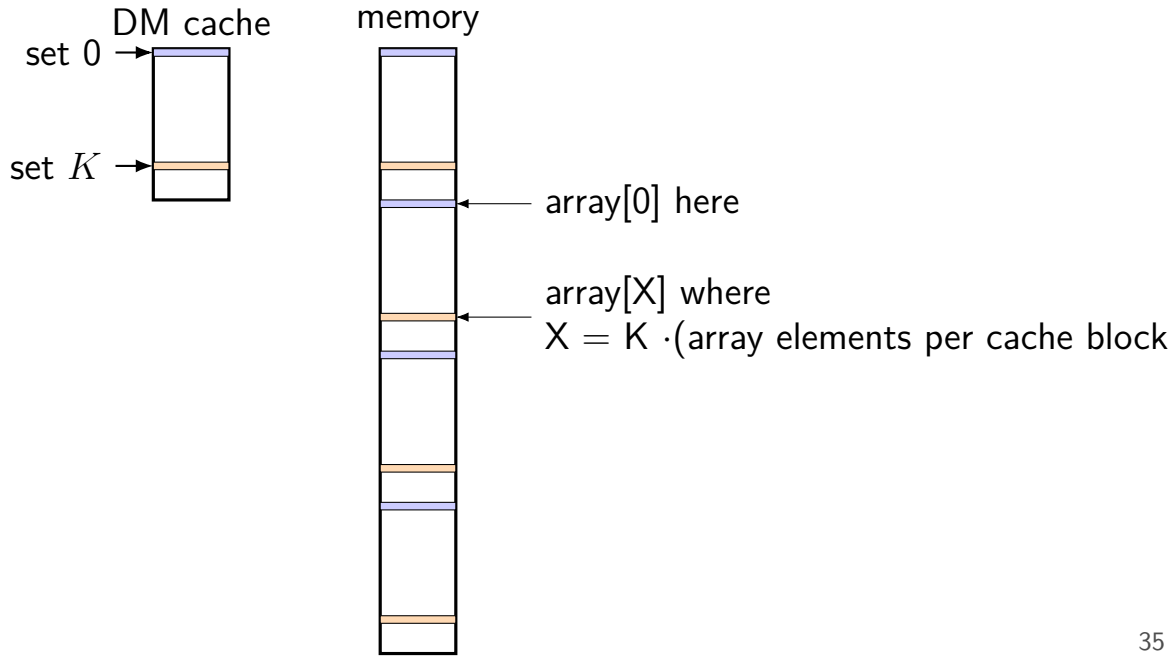
mapping of sets to memory (direct-mapped)



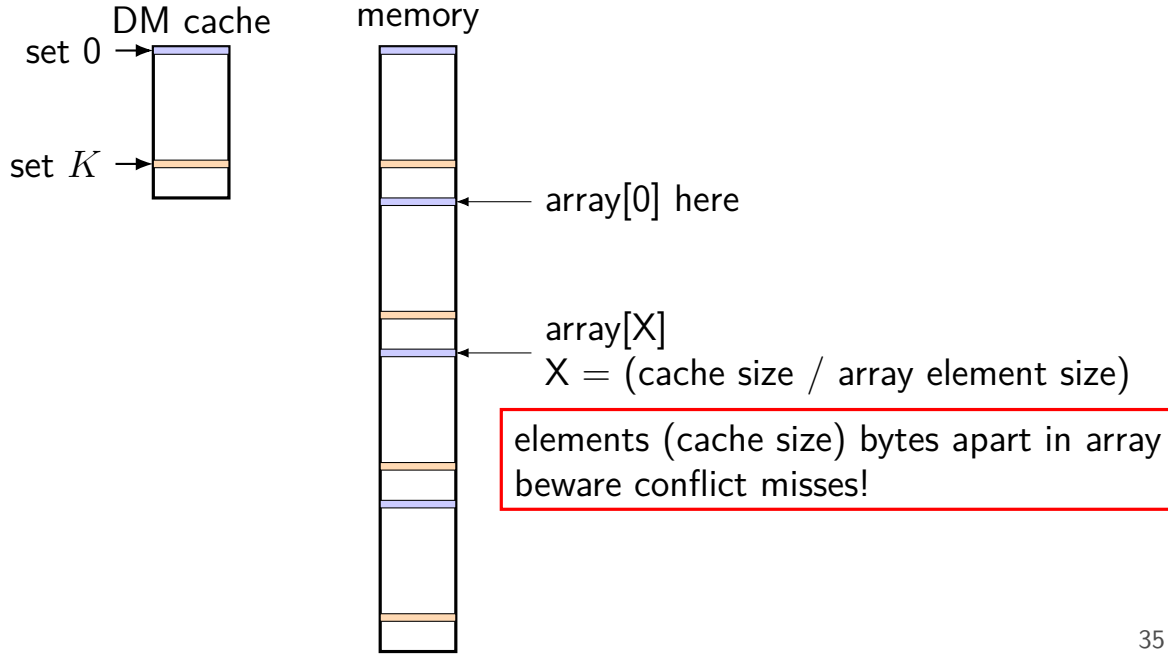
mapping of sets to memory (direct-mapped)



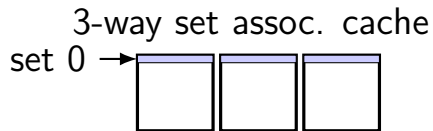
mapping of sets to memory (direct-mapped)



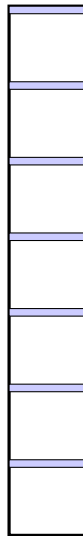
mapping of sets to memory (direct-mapped)



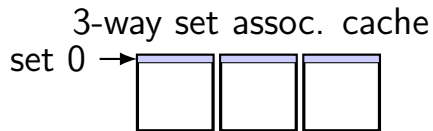
mapping of sets to memory (3-way)



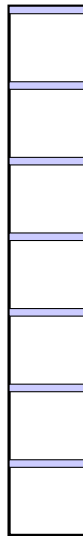
memory



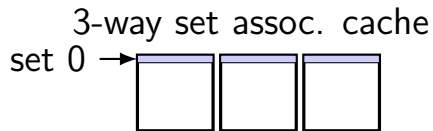
mapping of sets to memory (3-way)



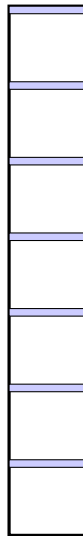
memory



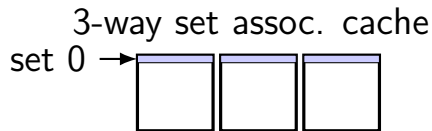
mapping of sets to memory (3-way)



memory



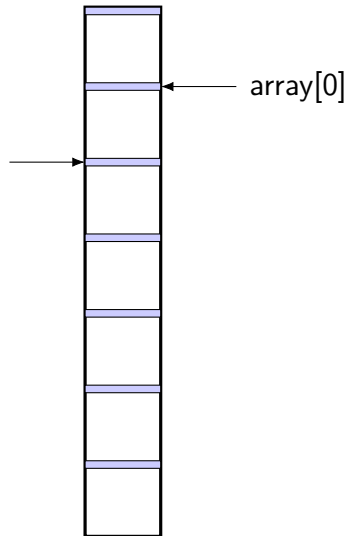
mapping of sets to memory (3-way)



$$\text{where } X = \frac{\text{way size}}{\text{array element size}}$$

accesses (way size) bytes apart in array?
beware conflict misses!

memory



C and cache misses (4)

```
typedef struct {  
    int a_value, b_value;  
    int other_values[6];  
} item;  
item items[5];  
int a_sum = 0, b_sum = 0;  
for (int i = 0; i < 5; ++i)  
    a_sum += items[i].a_value;  
for (int i = 0; i < 5; ++i)  
    b_sum += items[i].b_value;
```

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

C and cache misses (4, rewrite)

```
int array[40]
int a_sum = 0, b_sum = 0;
for (int i = 0; i < 40; i += 8)
    a_sum += array[i];
for (int i = 1; i < 40; i += 8)
    b_sum += array[i];
```

Assume everything but array is kept in registers (and the compiler does not do anything funny) and array starts at beginning of cache block.

How many *data cache misses* on a 2-way set associative 128B cache with 16B cache blocks and LRU replacement?

C and cache misses (4, solution pt 1)

ints 4 byte $\rightarrow$ array[0 to 3] and array[16 to 19] in same cache set

64B = 16 ints stored per way

4 sets total

accessing 0, 8, 16, 24, 32, 1, 9, 17, 25, 33

C and cache misses (4, solution pt 1)

ints 4 byte $\rightarrow$ array[0 to 3] and array[16 to 19] in same cache set

64B = 16 ints stored per way

4 sets total

accessing 0, 8, 16, 24, 32, 1, 9, 17, 25, 33

0 (set 0), 8 (set 2), 16 (set 0), 24 (set 2), 32 (set 0)

1 (set 0), 9 (set 2), 17 (set 0), 25 (set 2), 33 (set 0)

C and cache misses (4, solution pt 2)

access	set 0 after (LRU first)	result
—	—, —	
array[0]	—, array[0 to 3]	miss
array[16]	array[0 to 3], array[16 to 19]	miss
array[32]	array[16 to 19], array[32 to 35]	miss
array[1]	array[32 to 35], array[0 to 3]	miss
array[17]	array[0 to 3], array[16 to 19]	miss
array[32]	array[16 to 19], array[32 to 35]	miss

6 misses for set 0

C and cache misses (4, solution pt 3)

access	set 2 after (LRU first)	result	
—	—, —		
array[8]	—, array[8 to 11]	miss	2 misses for set 1
array[24]	array[8 to 11], array[24 to 27]	miss	
array[9]	array[8 to 11], array[24 to 27]	hit	
array[25]	array[16 to 19], array[32 to 35]	hit	

C and cache misses (3)

```
typedef struct {  
    int a_value, b_value;  
    int other_values[10];  
} item;  
item items[5];  
int a_sum = 0, b_sum = 0;  
for (int i = 0; i < 5; ++i)  
    a_sum += items[i].a_value;  
for (int i = 0; i < 5; ++i)  
    b_sum += items[i].b_value;
```

observation: 12 ints in struct: only first two used

equivalent to accessing array[0], array[12], array[24], etc.

...then accessing array[1], array[13], array[25], etc.

C and cache misses (3, rewritten?)

```
int array[60];  
int a_sum = 0, b_sum = 0;  
for (int i = 0; i < 60; i += 12)  
    a_sum += array[i];  
for (int i = 1; i < 60; i += 12)  
    b_sum += array[i];
```

Assume everything but array is kept in registers (and the compiler does not do anything funny) and array at beginning of cache block.

How many *data cache misses* on a 128B two-way set associative cache with 16B cache blocks and LRU replacement?

observation 1: first loop has 5 misses — first accesses to blocks

observation 2: array[0] and array[1], array[12] and array[13], etc. in same cache block

C and cache misses (3, solution)

ints 4 byte $\rightarrow$ array[0 to 3] and array[16 to 19] in same cache set

64B = 16 ints stored per way

4 sets total

accessing array indices 0, 12, 24, 36, 48, 1, 13, 25, 37, 49

so access to 1, 21, 41, 61, 81 all hits:

set 0 contains block with array[0 to 3]

set 5 contains block with array[20 to 23]

etc.

C and cache misses (3, solution)

ints 4 byte $\rightarrow$ array[0 to 3] and array[16 to 19] in same cache set

64B = 16 ints stored per way

4 sets total

accessing array indices 0, 12, 24, 36, 48, 1, 13, 25, 37, 49

so access to 1, 21, 41, 61, 81 all hits:

set 0 contains block with array[0 to 3]

set 5 contains block with array[20 to 23]

etc.

C and cache misses (3, solution)

ints 4 byte $\rightarrow$ array[0 to 3] and array[16 to 19] in same cache set

64B = 16 ints stored per way

4 sets total

accessing array indices 0, 12, 24, 36, 48, 1, 13, 25, 37, 49

0 (set 0, array[0 to 3]), 12 (set 3), 24 (set 2), 36 (set 1), 48 (set 0)

each set used at most twice

no replacement needed

so access to 1, 21, 41, 61, 81 all hits:

set 0 contains block with array[0 to 3]

set 5 contains block with array[20 to 23]

etc.

C and cache misses (3)

```
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?

C and cache misses (3, rewritten?)

```
item array[1024]; // 4 KB array  
int a_sum = 0, b_sum = 0;  
for (int i = 0; i < 1024; i += 128)  
    a_sum += array[i];  
for (int i = 1; i < 1024; i += 128)  
    b_sum += array[i];
```

C and cache misses (4)

```
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 4-way set associative 2KB direct-mapped cache with 16B cache blocks?

thinking about cache storage (1)

2KB direct-mapped cache with 16B blocks —

set 0: address 0 to 15, $(0 \text{ to } 15) + 2\text{KB}$, $(0 \text{ to } 15) + 4\text{KB}$, ...

set 1: address 16 to 31, $(16 \text{ to } 31) + 2\text{KB}$, $(16 \text{ to } 31) + 4\text{KB}$, ...

...

set 127: address 2032 to 2047, $(2032 \text{ to } 2047) + 2\text{KB}$, ...

thinking about cache storage (1)

2KB direct-mapped cache with 16B blocks —

set 0: address 0 to 15, $(0 \text{ to } 15) + 2\text{KB}$, $(0 \text{ to } 15) + 4\text{KB}$, ...

set 1: address 16 to 31, $(16 \text{ to } 31) + 2\text{KB}$, $(16 \text{ to } 31) + 4\text{KB}$, ...

...

set 127: address 2032 to 2047, $(2032 \text{ to } 2047) + 2\text{KB}$, ...

thinking about cache storage (1)

2KB direct-mapped cache with 16B blocks —

set 0: address 0 to 15, $(0 \text{ to } 15) + 2\text{KB}$, $(0 \text{ to } 15) + 4\text{KB}$, ...
block at 0: array[0] through array[3]

set 1: address 16 to 31, $(16 \text{ to } 31) + 2\text{KB}$, $(16 \text{ to } 31) + 4\text{KB}$, ...
block at 16: array[4] through array[7]

...

set 127: address 2032 to 2047, $(2032 \text{ to } 2047) + 2\text{KB}$, ...
block at 2032: array[508] through array[511]

thinking about cache storage (1)

2KB direct-mapped cache with 16B blocks —

set 0: address 0 to 15, $(0 \text{ to } 15) + 2\text{KB}$, $(0 \text{ to } 15) + 4\text{KB}$, ...

block at 0: `array[0]` through `array[3]`

block at $0+2\text{KB}$: `array[512]` through `array[515]`

set 1: address 16 to 31, $(16 \text{ to } 31) + 2\text{KB}$, $(16 \text{ to } 31) + 4\text{KB}$, ...

block at 16: `array[4]` through `array[7]`

block at $16+2\text{KB}$: `array[516]` through `array[519]`

...

set 127: address 2032 to 2047, $(2032 \text{ to } 2047) + 2\text{KB}$, ...

block at 2032: `array[508]` through `array[511]`

block at $2032+2\text{KB}$: `array[1020]` through `array[1023]`

thinking about cache storage (2)

2KB 2-way set associative cache with 16B blocks: block addresses

—

set 0: address 0, $0 + 2\text{KB}$, $0 + 4\text{KB}$, ...

set 1: address 16, $16 + 2\text{KB}$, $16 + 4\text{KB}$, ...

...

set 63: address 1008, $2032 + 2\text{KB}$, $2032 + 4\text{KB}$...

thinking about cache storage (2)

2KB 2-way set associative cache with 16B blocks: block addresses

set 0: address 0, $0 + 2\text{KB}$, $0 + 4\text{KB}$, ...
block at 0: array[0] through array[3]

set 1: address 16, $16 + 2\text{KB}$, $16 + 4\text{KB}$, ...
address 16: array[4] through array[7]

...

set 63: address 1008, $2032 + 2\text{KB}$, $2032 + 4\text{KB}$...
address 1008: array[252] through array[255]

thinking about cache storage (2)

2KB 2-way set associative cache with 16B blocks: block addresses

set 0: address 0, $0 + 2\text{KB}$, $0 + 4\text{KB}$, ...

block at 0: array[0] through array[3]

block at $0+1\text{KB}$: array[256] through array[259]

block at $0+2\text{KB}$: array[512] through array[515]

...

set 1: address 16, $16 + 2\text{KB}$, $16 + 4\text{KB}$, ...

address 16: array[4] through array[7]

...

set 63: address 1008, $2032 + 2\text{KB}$, $2032 + 4\text{KB}$...

address 1008: array[252] through array[255]

thinking about cache storage (2)

2KB 2-way set associative cache with 16B blocks: block addresses

set 0: address 0, $0 + 2\text{KB}$, $0 + 4\text{KB}$, ...

block at 0: array[0] through array[3]

block at 0+1KB: array[256] through array[259]

block at 0+2KB: array[512] through array[515]

...

set 1: address 16, $16 + 2\text{KB}$, $16 + 4\text{KB}$, ...

address 16: array[4] through array[7]

...

set 63: address 1008, $2032 + 2\text{KB}$, $2032 + 4\text{KB}$...

address 1008: array[252] through array[255]