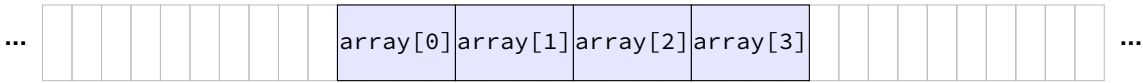
C and cache misses (warmup 1)

```
int array[4];
...
int even_sum = 0, odd_sum = 0;
even_sum += array[0];
odd_sum += array[1];
even_sum += array[2];
odd_sum += array[3];
```

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

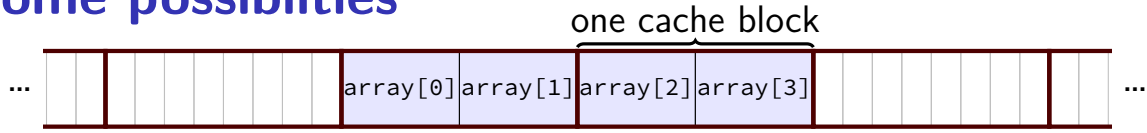
How many data cache misses on a 1-set direct-mapped cache with 8B blocks?

some possiblities



Q1: how do cache blocks correspond to array elements?
not enough information provided!

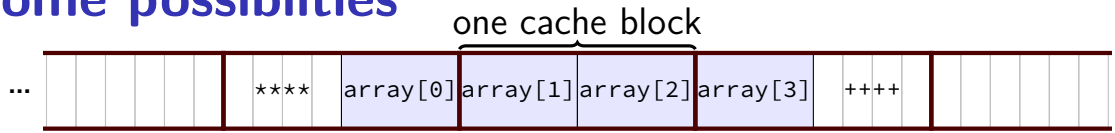
some possiblities



if array[0] starts at beginning of a cache block...
array split across two cache blocks

memory access	**cache contents afterwards**
—	`(empty)`
read `array[0]` (miss)	`{array[0], array[1]}`
read `array[1]` (hit)	`{array[0], array[1]}`
read `array[2]` (miss)	`{array[2], array[3]}`
read `array[3]` (hit)	`{array[2], array[3]}`

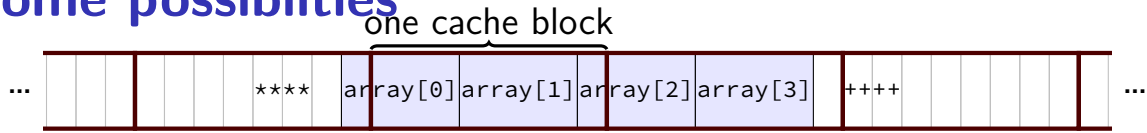
some possiblities



if array[0] starts right in the middle of a cache block
array split across three cache blocks

memory access	**cache contents afterwards**
—	`(empty)`
read `array[0]` (miss)	`{****, array[0]}`
read `array[1]` (miss)	`{array[1], array[2]}`
read `array[2]` (hit)	`{array[1], array[2]}`
read `array[3]` (miss)	`{array[3], ++++}`

some possiblities



if array[0] starts at an odd place in a cache block,
need to read two cache blocks to get most array elements

memory access	cache contents afterwards
—	(empty)
read `array[0]` byte 0 (miss)	{ ****, array[0] byte 0 }
read `array[0]` byte 1-3 (miss)	{ array[0] byte 1-3, array[2], array[3] byte 0 }
read `array[1]` (hit)	{ array[0] byte 1-3, array[2], array[3] byte 0 }
read `array[2]` byte 0 (hit)	{ array[0] byte 1-3, array[2], array[3] byte 0 }
read `array[2]` byte 1-3 (miss)	{part of array[2], array[3], ++++}
read `array[3]` (hit)	{part of array[2], array[3], ++++}

aside: alignment

compilers and malloc/new implementations usually try align values

align = make address be multiple of something

most important reason: don't cross cache block boundaries

C and cache misses (warmup 2)

```
int array[4];
int even_sum = 0, odd_sum = 0;
even_sum += array[0];
even_sum += array[2];
odd_sum += array[1];
odd_sum += array[3];
```

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

Assume array[0] at beginning of cache block.

How many data cache misses on a 1-set direct-mapped cache with 8B blocks?

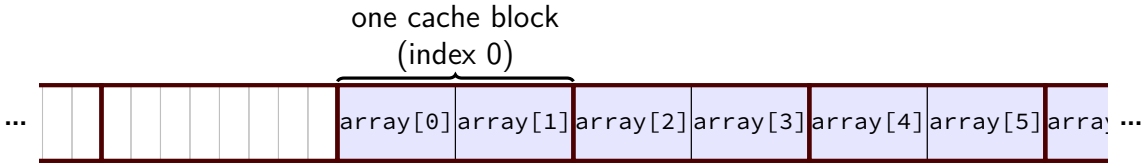
C and cache misses (warmup 3)

```
int array[8];
...
int even_sum = 0, odd_sum = 0;
even_sum += array[0];
odd_sum += array[1];
even_sum += array[2];
odd_sum += array[3];
even_sum += array[4];
odd_sum += array[5];
even_sum += array[6];
odd_sum += array[7];
```

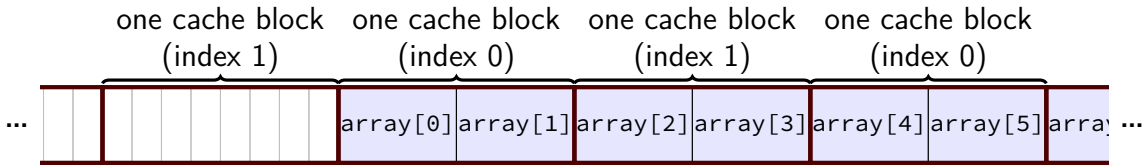
Assume everything but `array` is kept in registers (and the compiler does not do anything funny), and array[0] at beginning of cache block.

How many data cache misses on a **2**-set direct-mapped cache with 8B blocks?

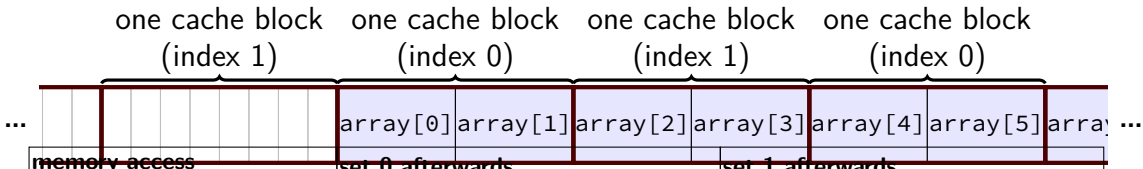
exercise solution



exercise solution

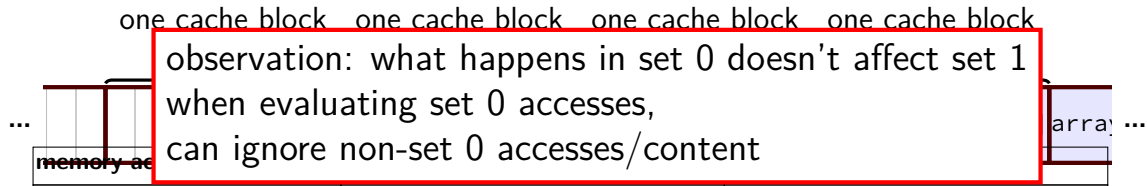


exercise solution



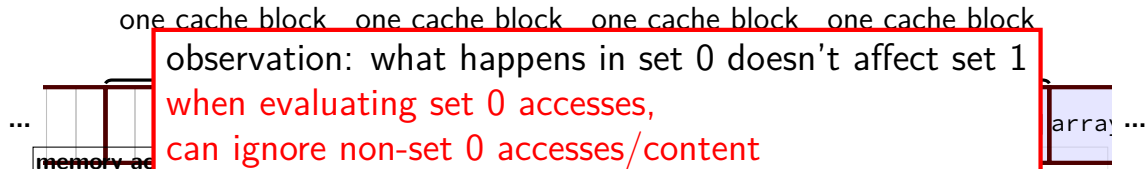
~~memory access~~	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read `array[0]` (miss)	{`array[0]`, `array[1]`}	(empty)
read `array[1]` (hit)	{`array[0]`, `array[1]`}	(empty)
read `array[2]` (miss)	{`array[0]`, `array[1]`}	{`array[2]`, `array[3]`}
read `array[3]` (hit)	{`array[0]`, `array[1]`}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

exercise solution



—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[1] (hit)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[3] (hit)	{array[0], array[1]}	{array[2], array[3]}
read array[4] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[5] (hit)	{array[4], array[5]}	{array[2], array[3]}
read array[6] (miss)	{array[4], array[5]}	{array[6], array[7]}
read array[7] (hit)	{array[4], array[5]}	{array[6], array[7]}

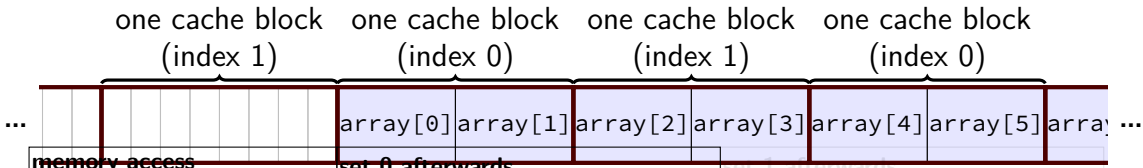
exercise solution



observation: what happens in set 0 doesn't affect set 1
when evaluating set 0 accesses,
can ignore non-set 0 accesses/content

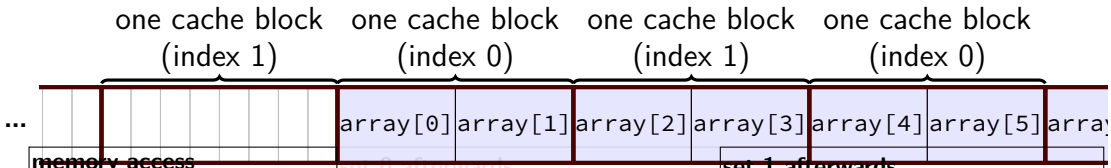
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[1] (hit)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[3] (hit)	{array[0], array[1]}	{array[2], array[3]}
read array[4] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[5] (hit)	{array[4], array[5]}	{array[2], array[3]}
read array[6] (miss)	{array[4], array[5]}	{array[6], array[7]}
read array[7] (hit)	{array[4], array[5]}	{array[6], array[7]}

exercise solution



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

exercise solution



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

arrays 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?

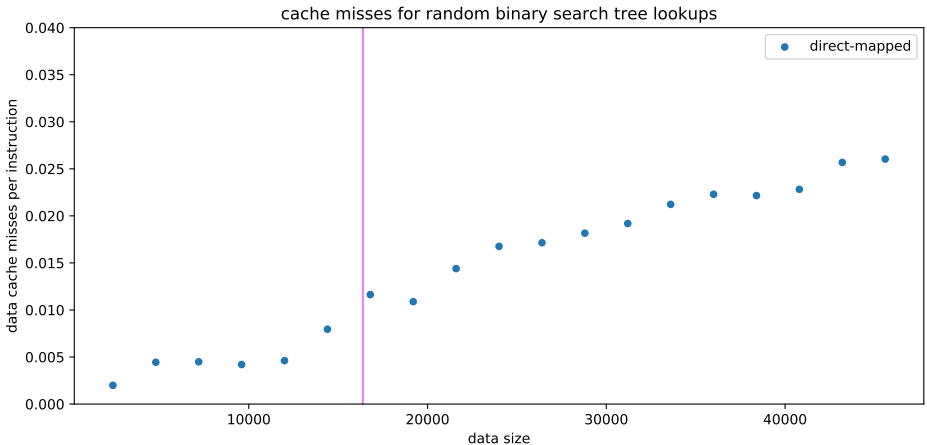
arrays and cache misses (2)

```
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 = 0; i < 1024; i += 2)
    odd_sum +=  array[i + 1];
```

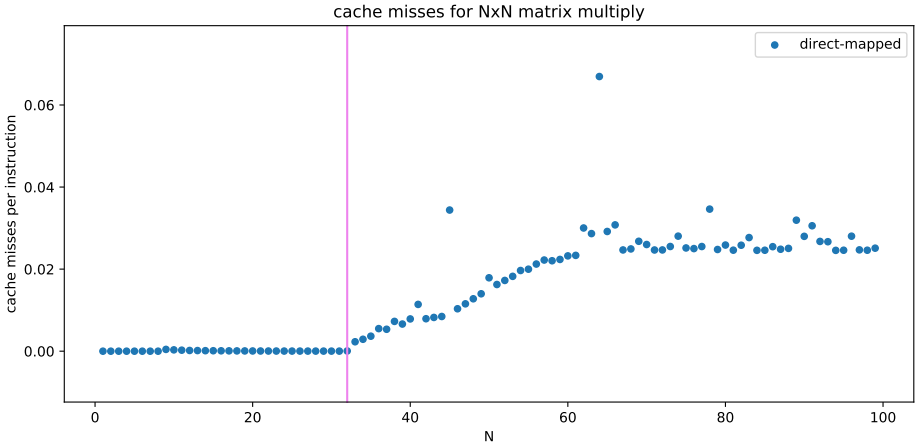
Assume everything but `array` is kept in registers (and the compiler does not do anything funny).

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

actual misses: BST lookups



actual misses: matrix multiplies



misses with skipping

```
int array1[512]; int array2[512];
...
for (int i = 0; i < 512; i += 1)
    sum += array1[i] * array2[i];
}
```

Assume everything but `array1`, `array2` is kept in registers (and the compiler does not do anything funny).

About how many *data cache misses* on a 2KB direct-mapped cache with 16B cache blocks?
Hint: depends on relative placement of array1, array2

best/worst case

`array1[i]` and `array2[i]` always different sets:

- 2 misses every 4 `i`
- = distance from array1 to array2 not multiple of # sets × bytes/set

`array1[i]` and `array2[i]` same sets:

- 2 misses every `i`
- = distance from array1 to array2 is multiple of # sets × bytes/set

worst case in practice?

two rows of matrix?

often sizeof(row) bytes apart

if the row size is multiple of number of sets $\times$ bytes per block, oops!

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

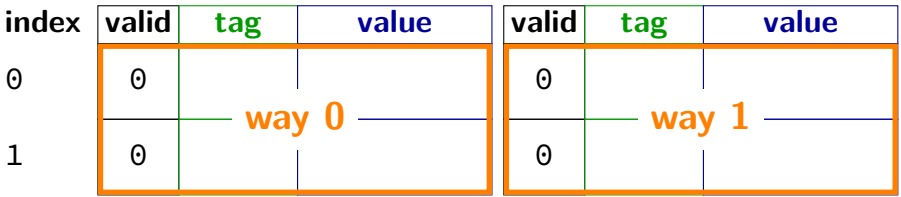
adding associativity

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



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)	

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)	

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)	

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)	

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)	

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)	

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!

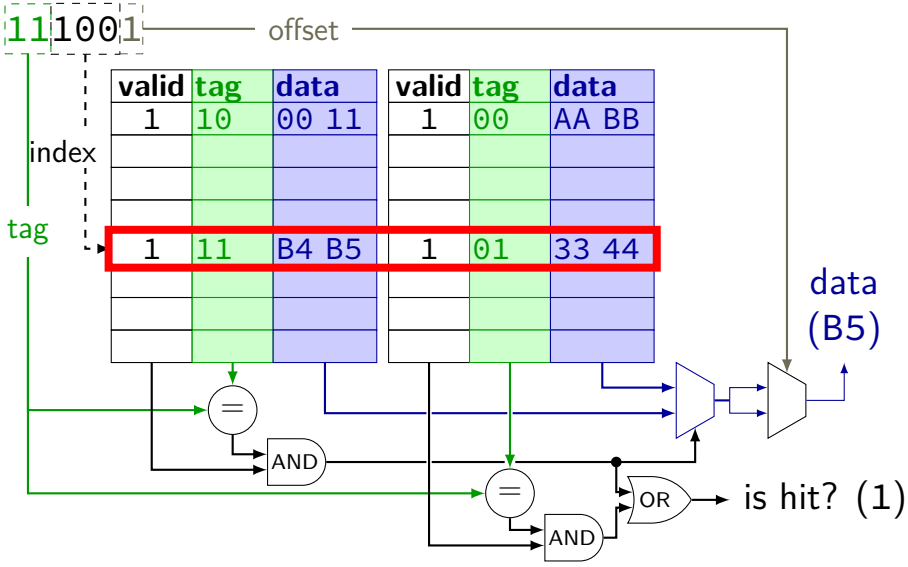
adding associativity

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

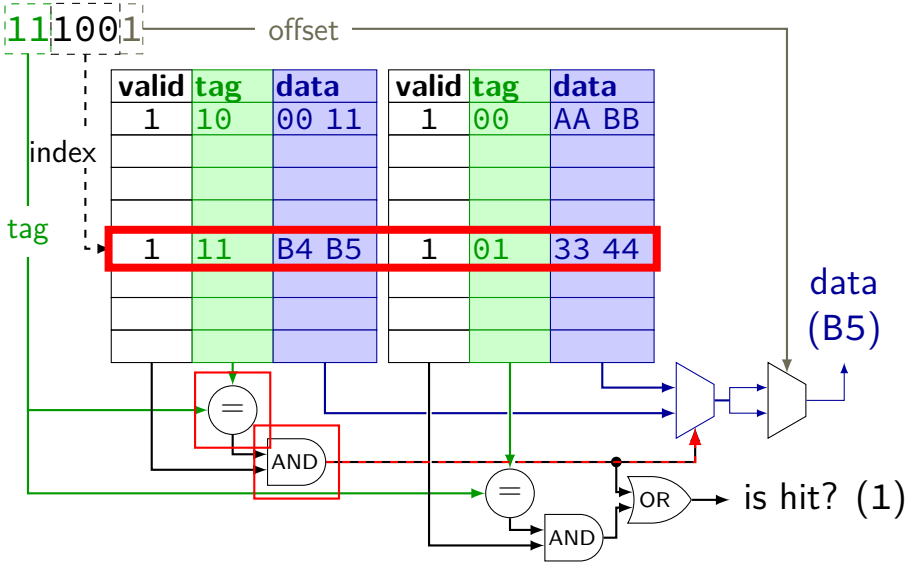
index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

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

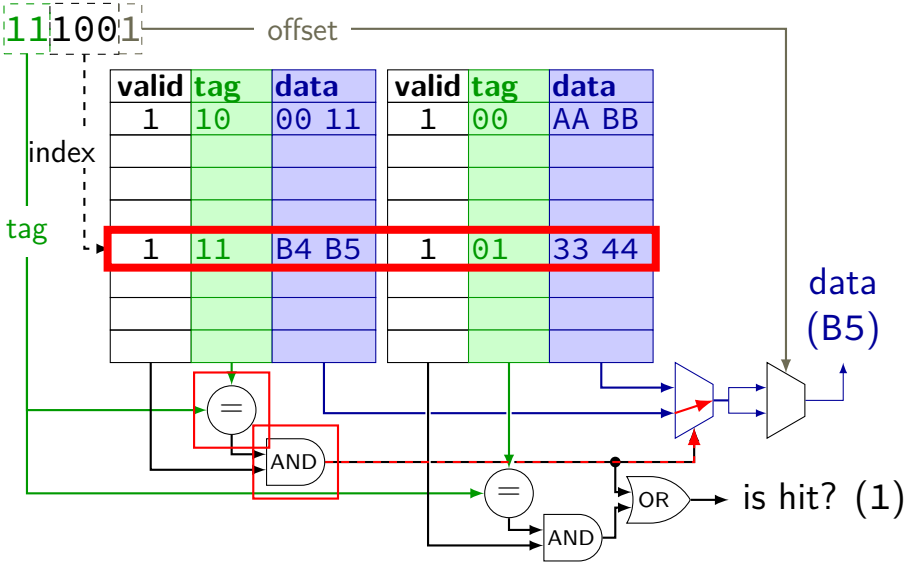
cache operation (associative)



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

m	memory addreses bits
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)

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)	$B = 64$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)	$B = 64$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)	$B = 64\text{Byte}$
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) $S = \frac{32\text{KB}}{64\text{Byte} \times 8} = 64$

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) $s = \log_2(64) = 6$

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`:

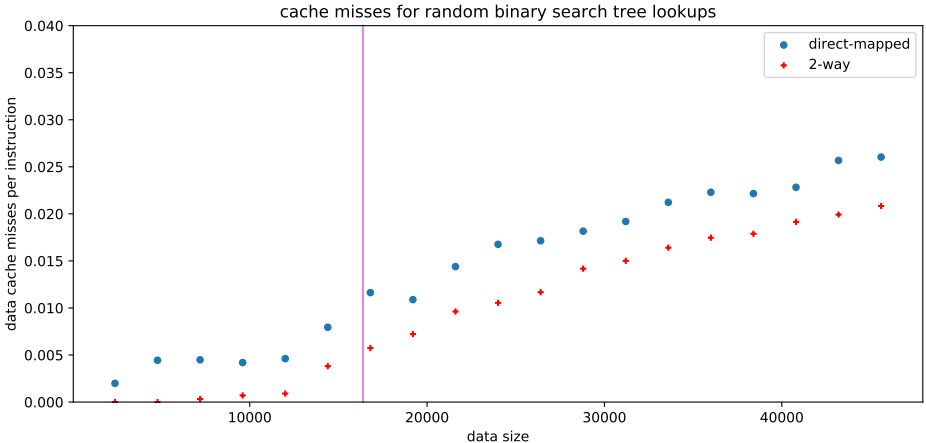
3	4	5	6	7
`0011`	`0100`	`0101`	`0110`	`0111`

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

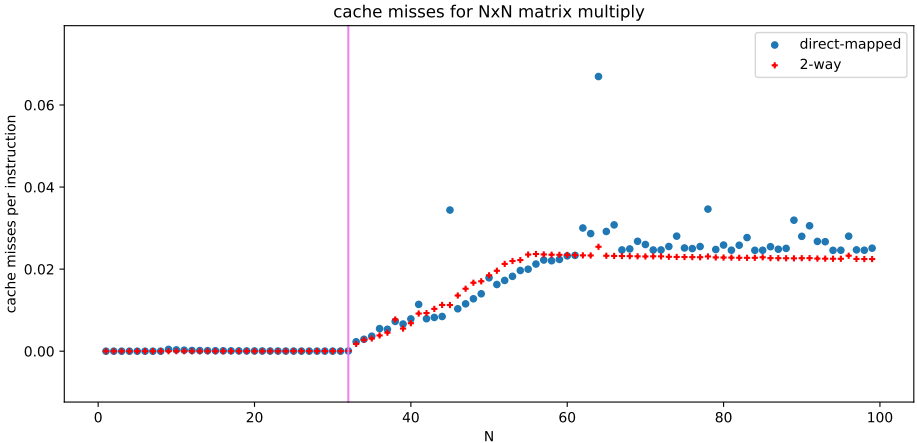
L3:

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

actual misses: BST lookups



actual misses: matrix multiplies



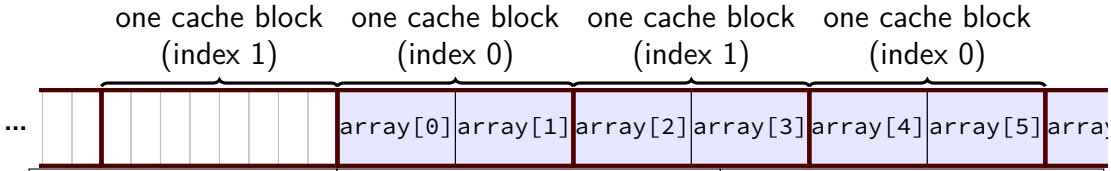
C and cache misses (warmup 4)

```
int array[8];
...
int even_sum = 0, odd_sum = 0;
even_sum += array[0];
even_sum += array[2];
even_sum += array[4];
even_sum += array[6];
odd_sum += array[1];
odd_sum += array[3];
odd_sum += array[5];
odd_sum += array[7];
```

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

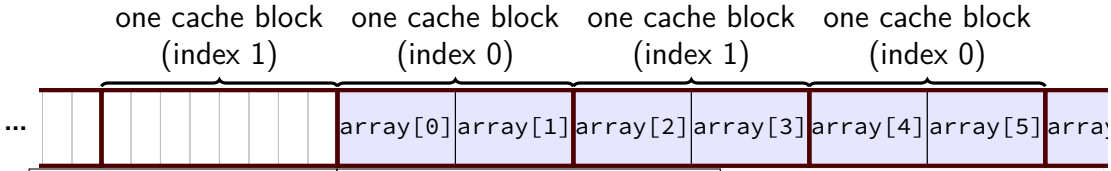
How many data cache misses on a **2**-set direct-mapped cache with 8B blocks?

exercise solution



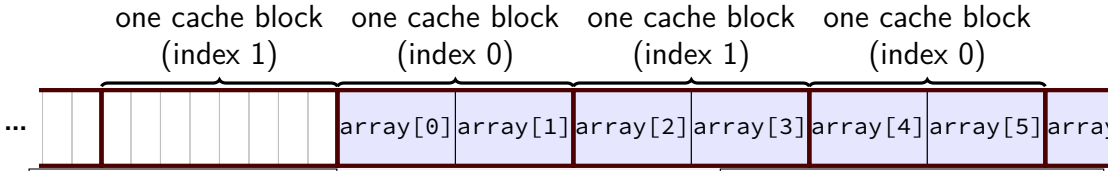
memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[4] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[6] (miss)	{array[4], array[5]}	{array[6], array[7]}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

exercise solution



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

exercise solution



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

arrays 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 = 0; 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?

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)	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

- 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
- really just need "avoid recently used" — much faster/simpler
- good approximations: E to $2E$ bits

first-in, first-out

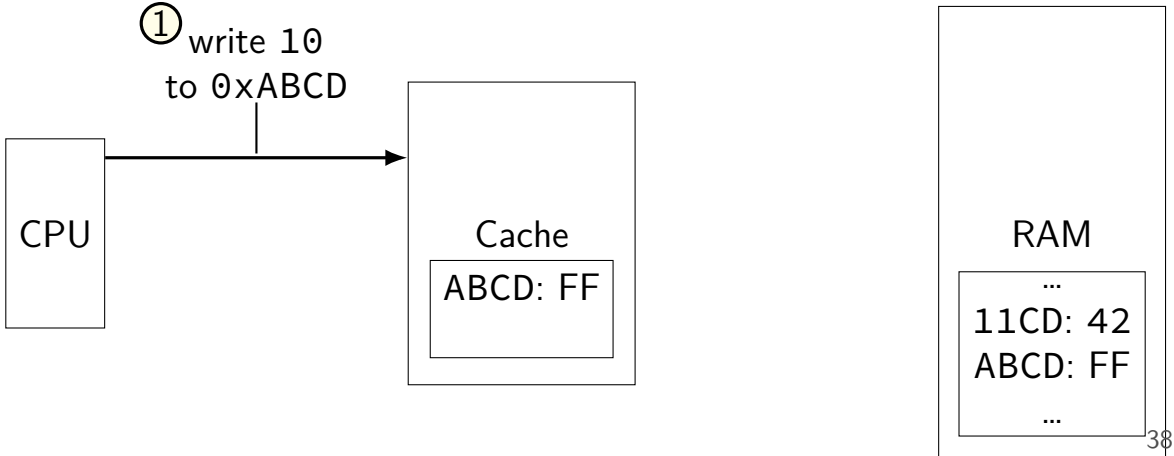
- counter per set — where to replace next

(pseudo-)random

- no extra information!
- actually works pretty well in practice

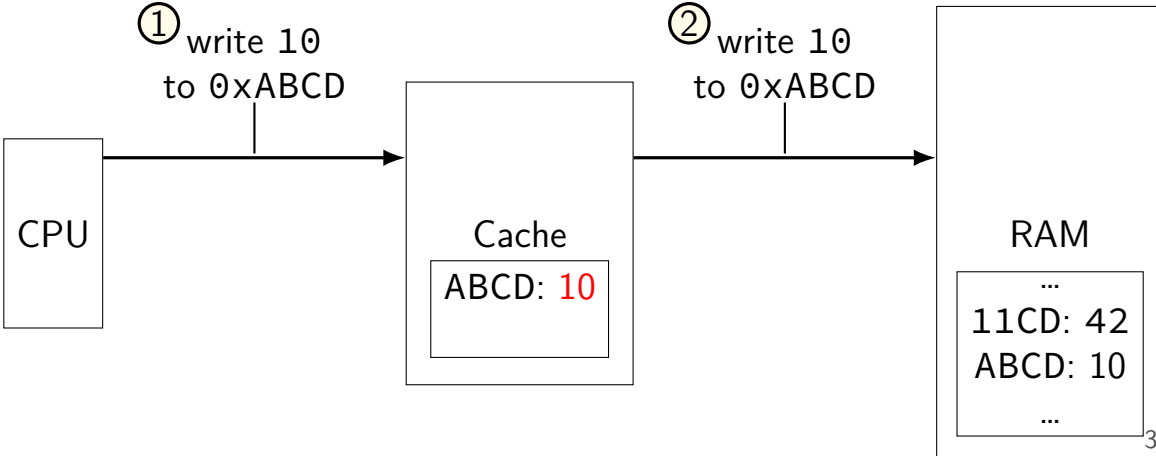
write-through v. write-back

option 1: write-through



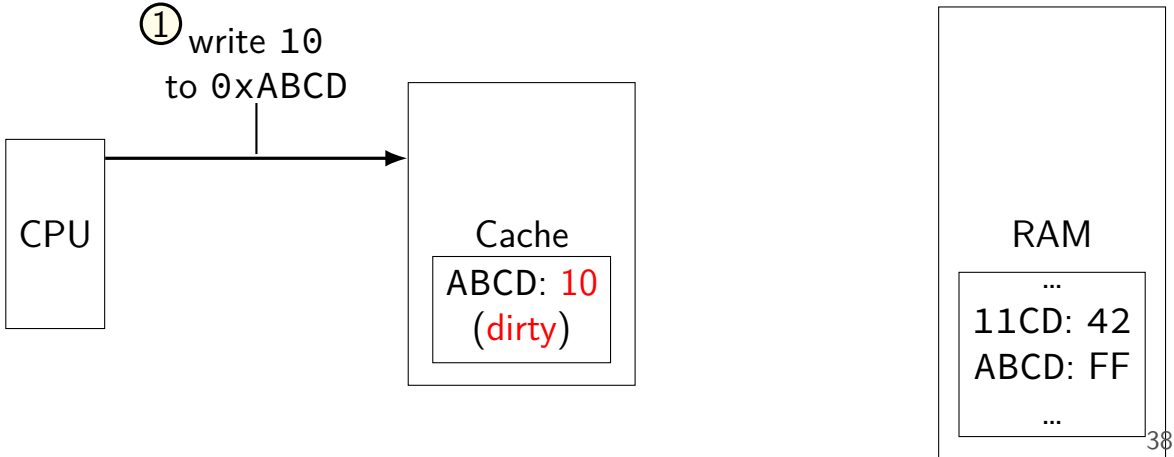
write-through v. write-back

option 1: write-through



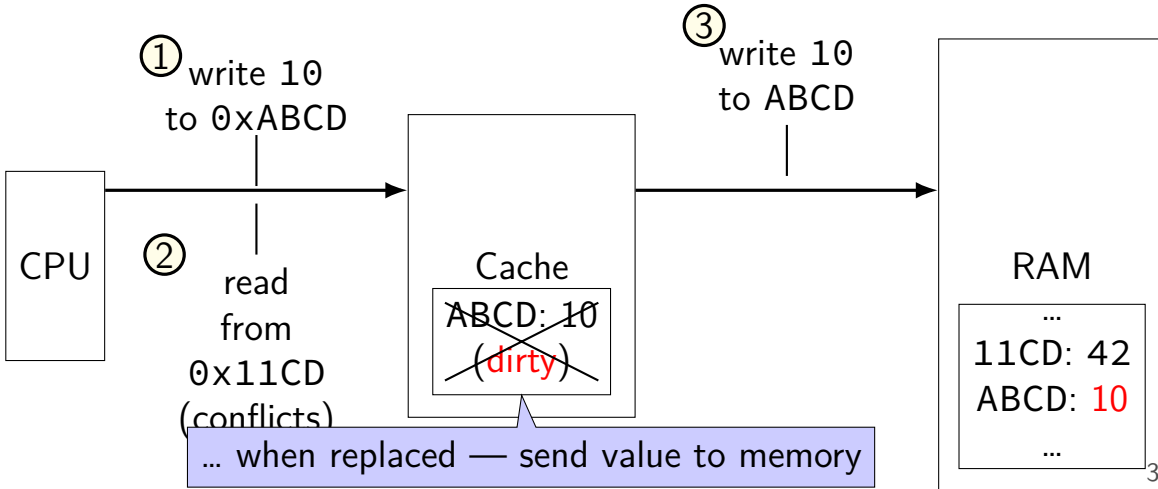
write-through v. write-back

option 2: write-back

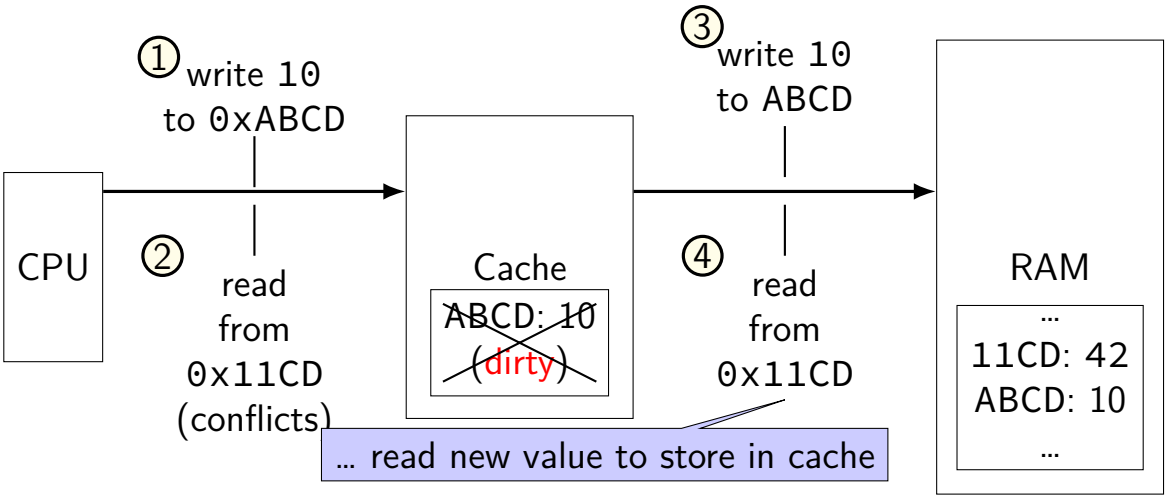


write-through v. write-back

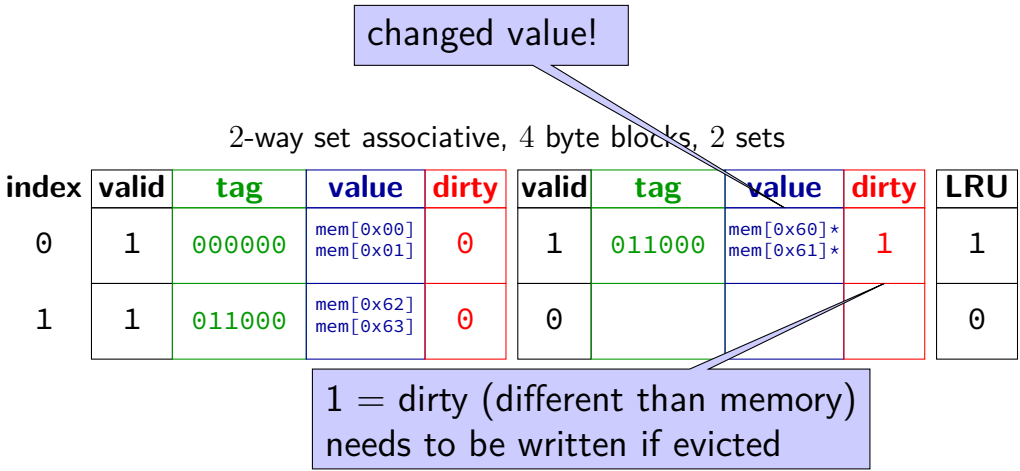
option 2: write-back



write-through v. write-back



writeback policy



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
(then follow write-through or write-back policy)

write-no-allocate

don't use cache at all (send write to memory *instead*)
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 $\widehat{0}$xFF 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 $\widehat{0}$xFF 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 $\widehat{0}$xFF 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	000001	0xFF mem[0x05]	1	0
1	1	011000	mem[0x62] mem[0x63]	0	0				0

writing $\widehat{0}$xFF 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 $\widehat{0}$xFF 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 (1, solution)

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~~0	1

writing 1 byte to 0x33: (set 1, offset 1) no read or write

reading 1 byte from 0x52:

reading 1 byte from 0x50:

exercise (1, solution)

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[0x50] mem[0x51]	~~0~~1	1

writing 1 byte to 0x33: (set 1, offset 1) no read or write

reading 1 byte from 0x52: (set 1, offset 0) **write** back 0x32-0x33; **read** 0x52-0x53

reading 1 byte from 0x50:

exercise (1, solution)

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

writing 1 byte to 0x33: (set 1, offset 1) no read or write

reading 1 byte from 0x52: (set 1, offset 0) **write** back 0x32-0x33; **read** 0x52-0x53

reading 1 byte from 0x50: (set 0, offset 0) replace 0x30-0x31 (no write back); **read** 0x50-0x51

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

exercise (2, solution)

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~~0

writing 1 byte to 0x33: (set 1, offset 1) write-through 0x33 modification

reading 1 byte from 0x52:

reading 1 byte from 0x50:

exercise (2, solution)

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

index	valid	tag	value	valid	tag	value	LRU
0	1	001100	mem[0x50] mem[0x51]	1	010000	mem[0x40] mem[0x41]	~~01~~
1	1	011000	mem[0x62] mem[0x63]	1	001100	mem[0x52] mem[0x53]	~~10~~

writing 1 byte to 0x33: (set 1, offset 1) write-through 0x33 modification

reading 1 byte from 0x52: (set 1, offset 0) replace 0x32-0x33; **read** 0x52-0x53

reading 1 byte from 0x50:

exercise (2, solution)

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

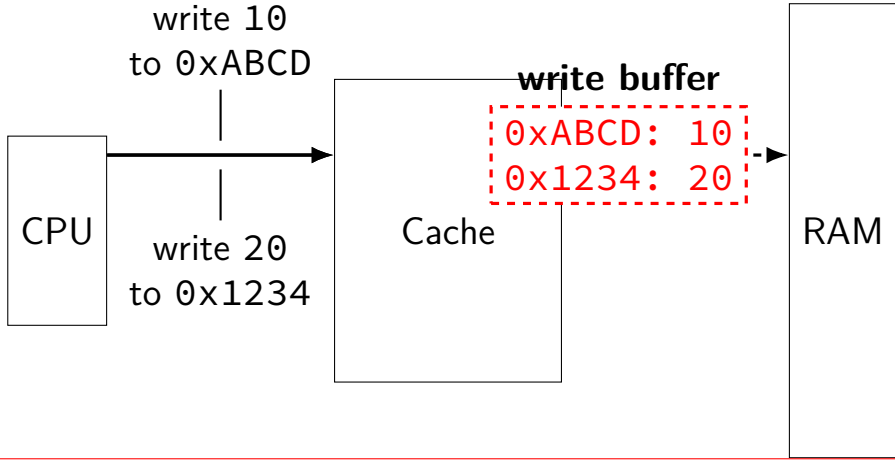
index	valid	tag	value	valid	tag	value	LRU
0	1	001100	mem[0x30] mem[0x31]	1	010000	mem[0x40] mem[0x41]	0
1	1	011000	mem[0x62] mem[0x63]	1	001100	mem[0x32] mem[0x33]	1

writing 1 byte to 0x33: (set 1, offset 1) write-through 0x33 modification

reading 1 byte from 0x52: (set 1, offset 0) replace 0x32-0x33; **read** 0x52-0x53

reading 1 byte from 0x50: (set 0, offset 0) replace 0x30-0x31; **read** 0x50-0x51

fast writes



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

cache miss types

common to categorize misses:

roughly "cause" of miss assuming cache block size fixed

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

capacity — cache was not big enough

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

(assuming typical locality + keeping cache size constant if possible...)

	miss rate	hit time	miss penalty
increase cache size	better	worse	—
increase associativity	better	worse	worse?
increase block size	depends	worse	worse
add secondary cache	—	—	better
write-allocate	better	—	?
writeback	—	—	?
LRU replacement	better	?	worse?
prefetching	better	—	—

prefetching = guess what program will use, access in advance

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

cache optimizations by miss type

(assuming other listed parameters remain constant)

	capacity	conflict	compulsory
increase cache size	fewer misses	fewer misses	—
increase associativity	—	fewer misses	—
increase block size	more misses?	more misses?	fewer misses
LRU replacement	—	fewer misses	—
prefetching	—	—	fewer misses

cache accesses and multi-level PTs

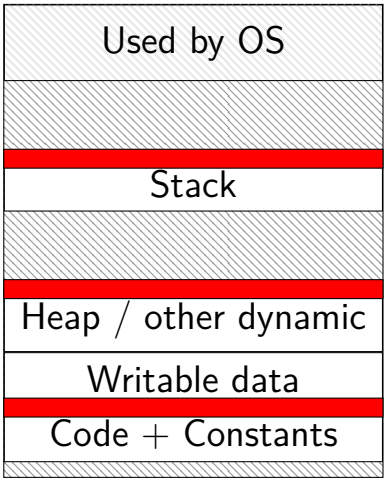
four-level page tables — five cache accesses per program memory access

L1 cache hits — typically a couple cycles each?

so add 8 cycles to each program memory access?

not acceptable

program memory active sets



0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F...

small areas of memory active at a time
one or two pages in each area?

0x0000 0000 0040 0000

page table entries and locality

page table entries have excellent temporal locality

typically one or two pages of the stack active

typically one or two pages of code active

typically one or two pages of heap/globals active

each page contains whole functions, arrays, stack frames, etc.

page table entries and locality

page table entries have excellent temporal locality

typically one or two pages of the stack active

typically one or two pages of code active

typically one or two pages of heap/globals active

each page contains whole functions, arrays, stack frames, etc.

needed page table entries are very small

page table entry cache

caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

page table entry cache

caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

only caches the page table lookup itself
(generally) just entries from the last-level page tables

page table entry cache

caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

not much spatial locality between page table entries
(they're used for kilobytes of data already)
(and if spatial locality, maybe use larger page size?)

page table entry cache

caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

few active page table entries at a time
enables highly associative cache designs

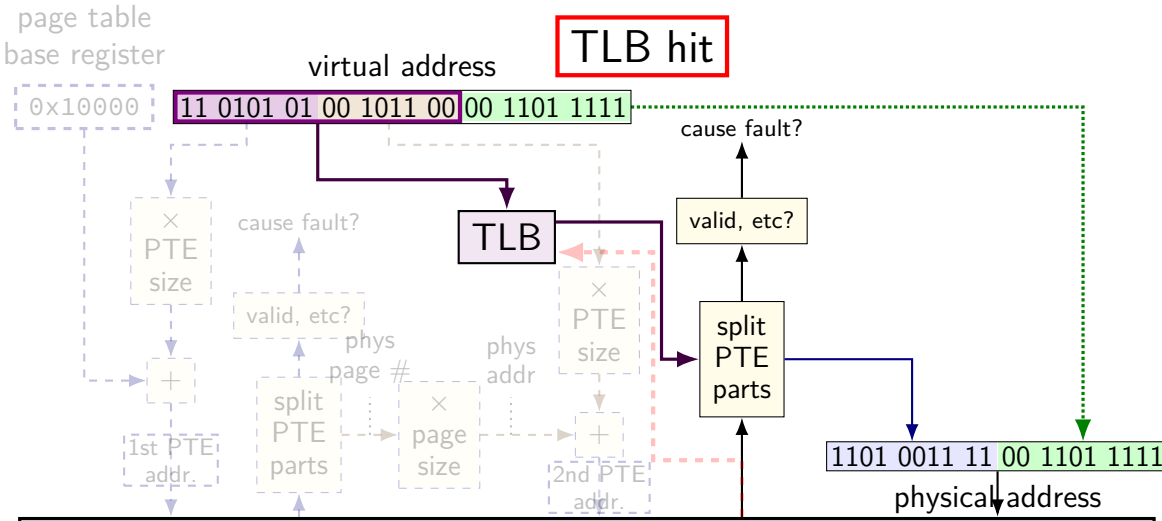
TLB and multi-level page tables

TLB caches valid last-level page table entries

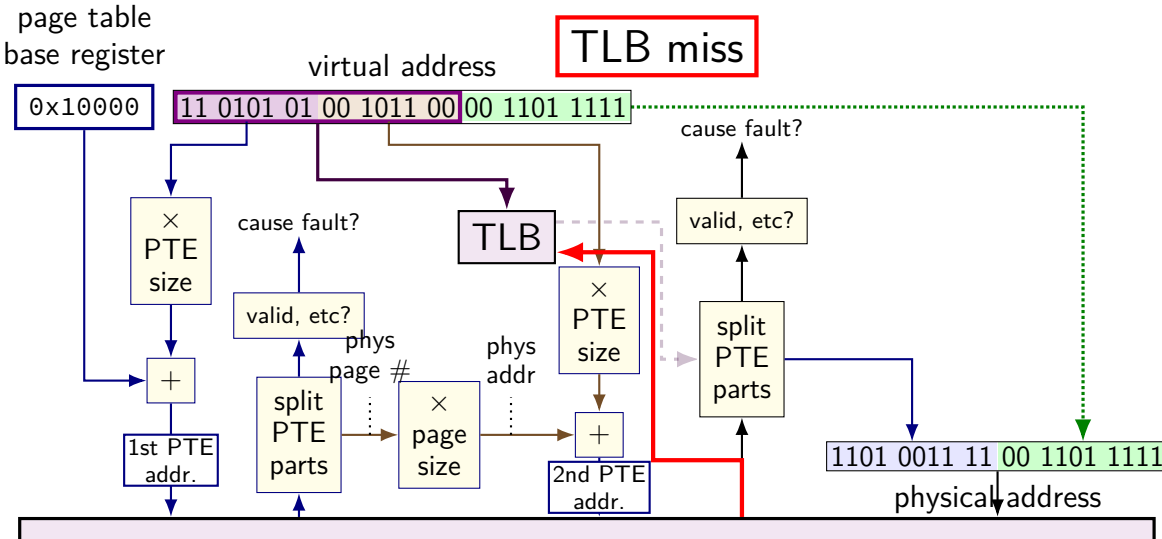
doesn't matter which last-level page table

means TLB output can be used directly to form address

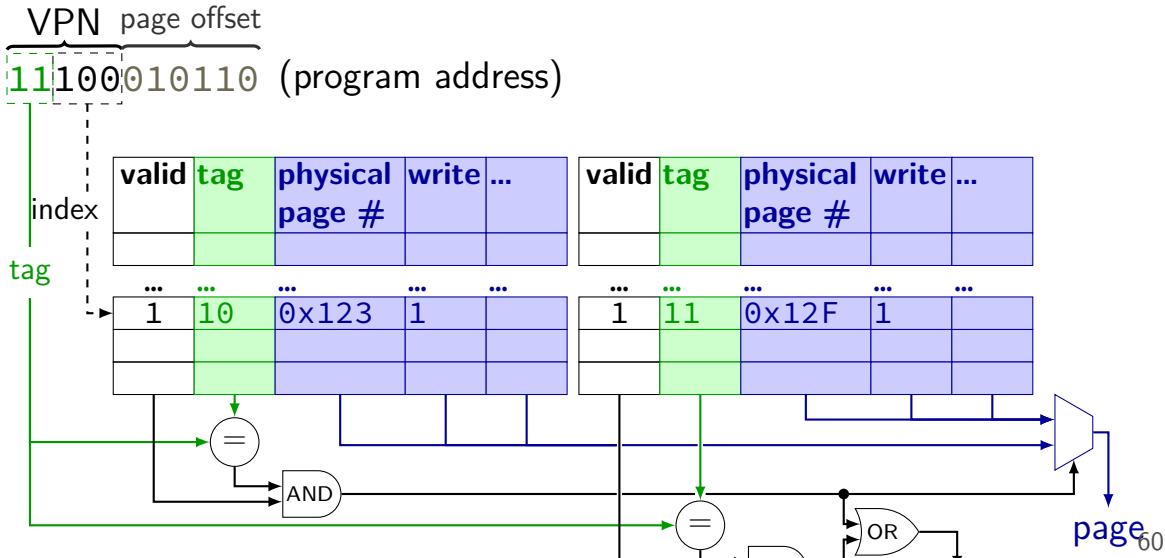
TLB and two-level lookup



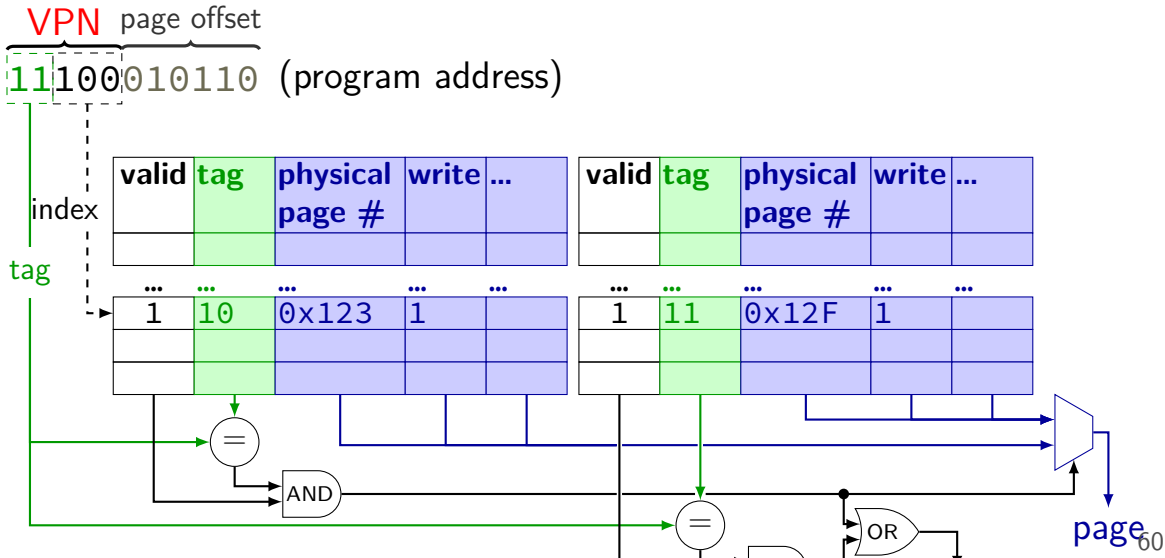
TLB and two-level lookup



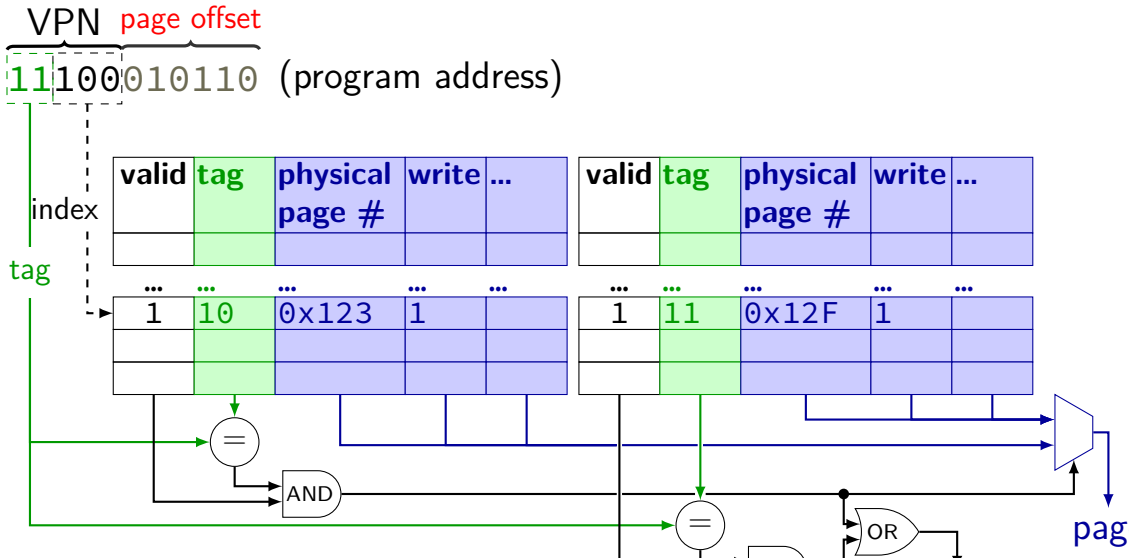
TLB organization (2-way set associative)



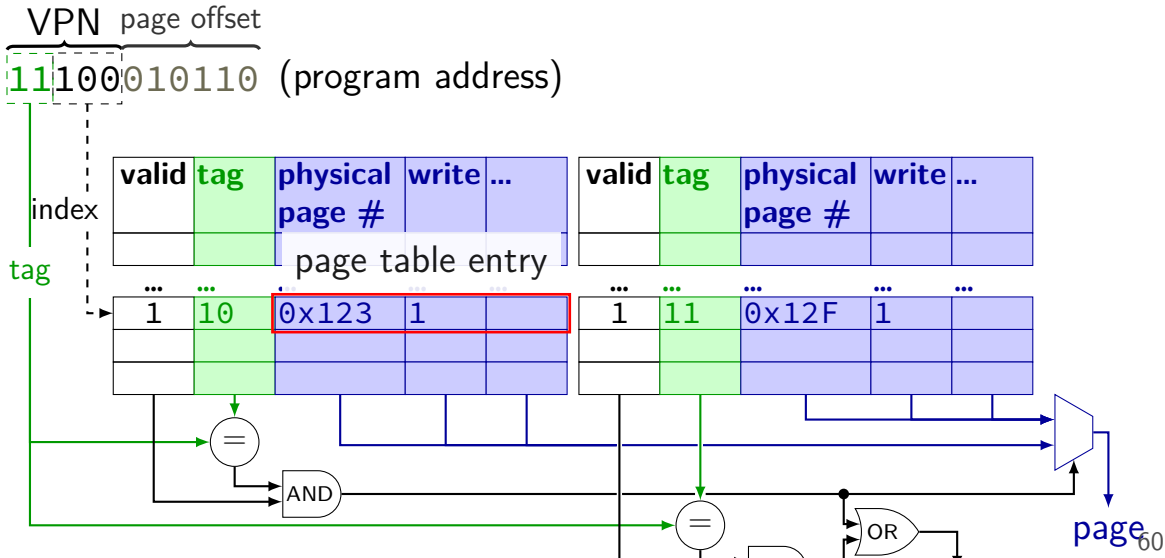
TLB organization (2-way set associative)



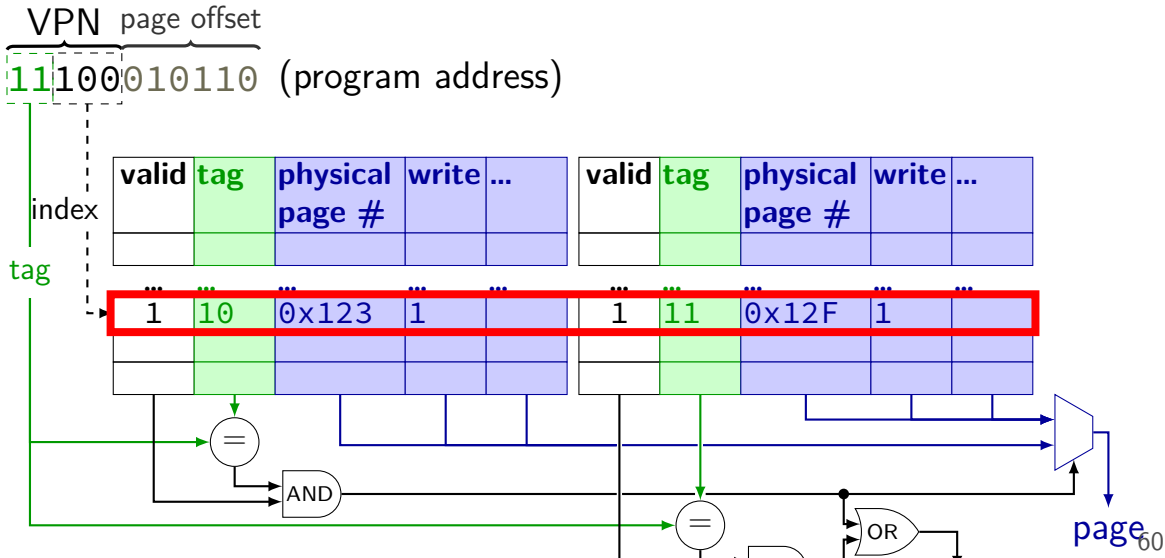
TLB organization (2-way set associative)



TLB organization (2-way set associative)



TLB organization (2-way set associative)



address splitting for TLBs (1)

my desktop:

4KB (2^{12} byte) pages; 48-bit virtual address

64-entry, 4-way L1 data TLB

TLB index bits?

TLB tag bits?

address splitting for TLBs (1)

my desktop:

4KB (2^{12} byte) pages; 48-bit virtual address

64-entry, 4-way L1 data TLB

TLB index bits?

$64/4 = 16$ sets — 4 bits

TLB tag bits?

$48 - 12 = 36$ bit virtual page number — $36 - 4 = 32$ bit TLB tag

address splitting for TLBs (2)

my desktop:

4KB (2^{12} byte) pages; 48-bit virtual address

1536-entry ($3 \cdot 2^9$), 12-way L2 TLB

TLB index bits?

TLB tag bits?

address splitting for TLBs (2)

my desktop:

4KB (2^{12} byte) pages; 48-bit virtual address

1536-entry ($3 \cdot 2^9$), 12-way L2 TLB

TLB index bits?

$1536/12 = 128$ sets — 7 bits

TLB tag bits?

$48 - 12 = 36$ bit virtual page number — $36 - 7 = 29$ bit TLB tag

exercise: TLB access pattern (setup)

4-entry, 2-way TLB, LRU replacement policy, initially empty

4096 byte pages

how many index bits?

TLB index of virtual address 0x12345?

exercise: TLB access pattern

4-entry, 2-way TLB, LRU replacement policy, initially empty

4096 byte pages

type	virtual	physical
`read`	`0x440030`	`0x554030`
`write`	`0x440034`	`0x554034`
`read`	`0x7FFFE008`	`0x556008`
`read`	`0x7FFFE000`	`0x556000`
`read`	`0x7FFFDFF8`	`0x5F8FF8`
`read`	`0x664080`	`0x5F9080`
`read`	`0x440038`	`0x554038`
`write`	`0x7FFFDFF0`	`0x5F8FF0`

which are TLB hits? which are TLB misses? final contents of TLB?

exercise: TLB access pattern

4-entry, 2-way TLB, LRU replacement policy, initially empty

4096 byte pages

				VPNs of PTEs held in TLB	
type	virtual	physical	result	set 0	set 1
read	0x440030	0x554030	miss	0x440	
write	0x440034	0x554034	hit	0x440	
read	0x7FFFE008	0x556008	miss	0x440	
read	0x7FFFE000	0x556000	hit	0x440, 0x7FFFE	
read	0x7FFFDFF8	0x5F8FF8	miss	0x440, 0x7FFFE	0x7FFFD
read	0x664080	0x5F9080	miss	0x664, 0x7FFFE	0x7FFFD
read	0x440038	0x554038	miss	0x664, 0x440	0x7FFFD
write	0x7FFFDFF0	0x5F8FF0	hit	0x664, 0x440	0x7FFFD

which are TLB hits? which are TLB misses? final contents of TLB?

exercise: TLB access pattern

4-entry, 2-way TLB, LRU replacement policy, initially empty

4096 byte pages

set idx	V	tag	physical page	write?	kernel?	···	LRU?
0	1	0x00220 (0x440 ≫ 1)	0x554	1	0	···	no
	1	0x00332 (0x00664 ≫ 1)	0x5F9	1	0	···	yes
1	1	0x3FFFF (0x7FFFD ≫ 1)	0x5F8	1	0	···	no
	0	---	---	–	–	···	yes

type					
read	0x440038	0x554038	miss	0x664, 0x440	0x7FFFD
write	0x7FFFDFF0	0x5F8FF0	hit	0x664, 0x440	0x7FFFD

which are TLB hits? which are TLB misses? final contents of TLB?

changing page tables

what happens to TLB when page table base pointer is changed?

- e.g. context switch

most entries in TLB refer to things from wrong process

- oops — read from the wrong process's stack?

changing page tables

what happens to TLB when page table base pointer is changed?

- e.g. context switch

most entries in TLB refer to things from wrong process

- oops — read from the wrong process's stack?

option 1: invalidate all TLB entries

- side effect on "change page table base register" instruction

changing page tables

what happens to TLB when page table base pointer is changed?

- e.g. context switch

most entries in TLB refer to things from wrong process

- oops — read from the wrong process's stack?

option 1: invalidate all TLB entries

- side effect on "change page table base register" instruction

option 2: TLB entries contain process ID

- set by OS (special register)
- checked by TLB in addition to TLB tag, valid bit

editing page tables

what happens to TLB when OS changes a page table entry?

most common choice: has to be handled in software

editing page tables

what happens to TLB when OS changes a page table entry?

most common choice: has to be handled in software

invalid to valid — nothing needed

- TLB doesn't contain invalid entries
- MMU will check memory again

valid to invalid — OS needs to tell processor to invalidate it

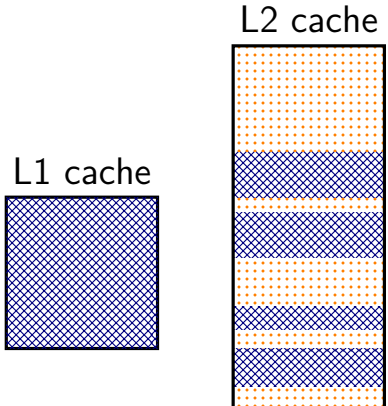
- special instruction (x86: `invlpg`)

valid to other valid — OS needs to tell processor to invalidate it

inclusive versus exclusive

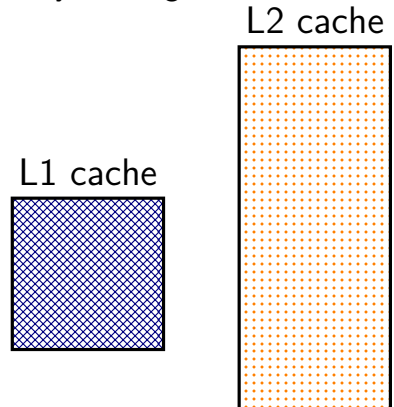
L2 inclusive of L1

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



L2 exclusive of L1

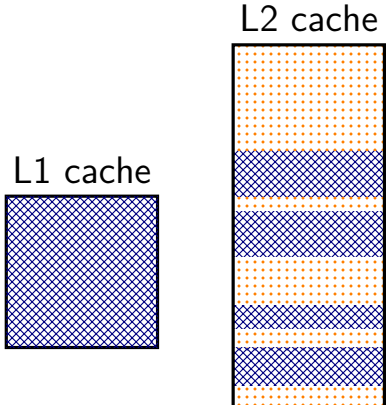
L2 contains different data than L1
adding to L1 must remove from L2
probably evicting from L1 adds to L2



inclusive versus exclusive

L2 inclusive of L1

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



inclusive policy:
no extra work on eviction
but duplicated data

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

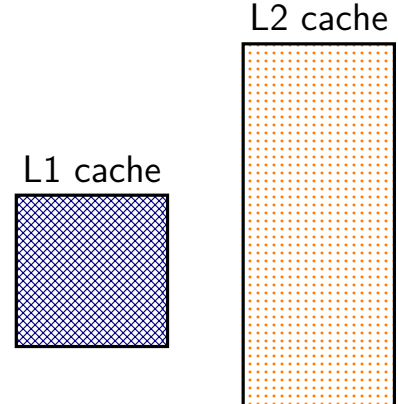
inclusive versus exclusive

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



Tag-Index-Offset formulas (direct-mapped)

(formulas derivable from prior slides)

$S = 2^s$	number of sets
s	(set) index bits
$B = 2^b$	block size
b	(block) offset bits
m	memory addreses bits
$t = m - (s + b)$	tag bits
$C = B \times S$	cache size (if direct-mapped)

Tag-Index-Offset formulas (direct-mapped)

(formulas derivable from prior slides)

$S = 2^s$	number of sets
s	(set) index bits
$B = 2^b$	block size
b	(block) offset bits
m	memory addreses bits
$t = m - (s + b)$	tag bits
$C = B \times S$	cache size (if direct-mapped)