caching

Imagine you are working at Intel in the 1980s

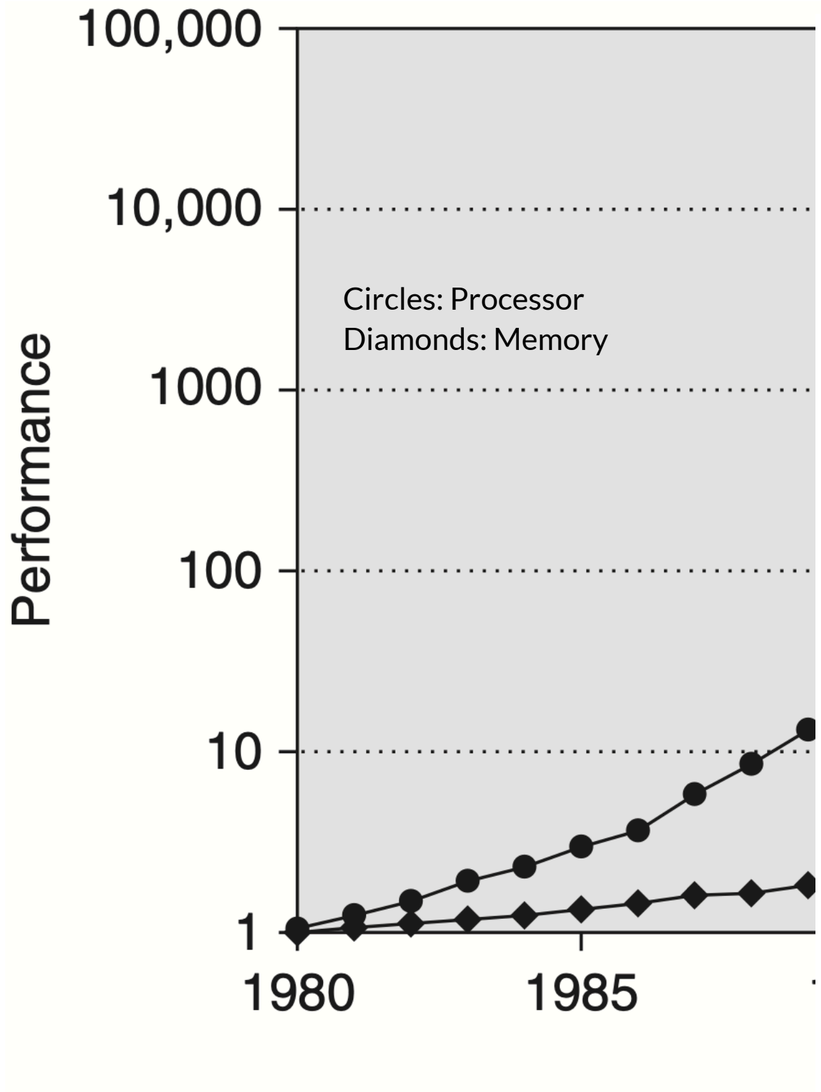


Image from “Computer Architecture: A Quantitative Approach”

Imagine you are working at Intel in the 1980s

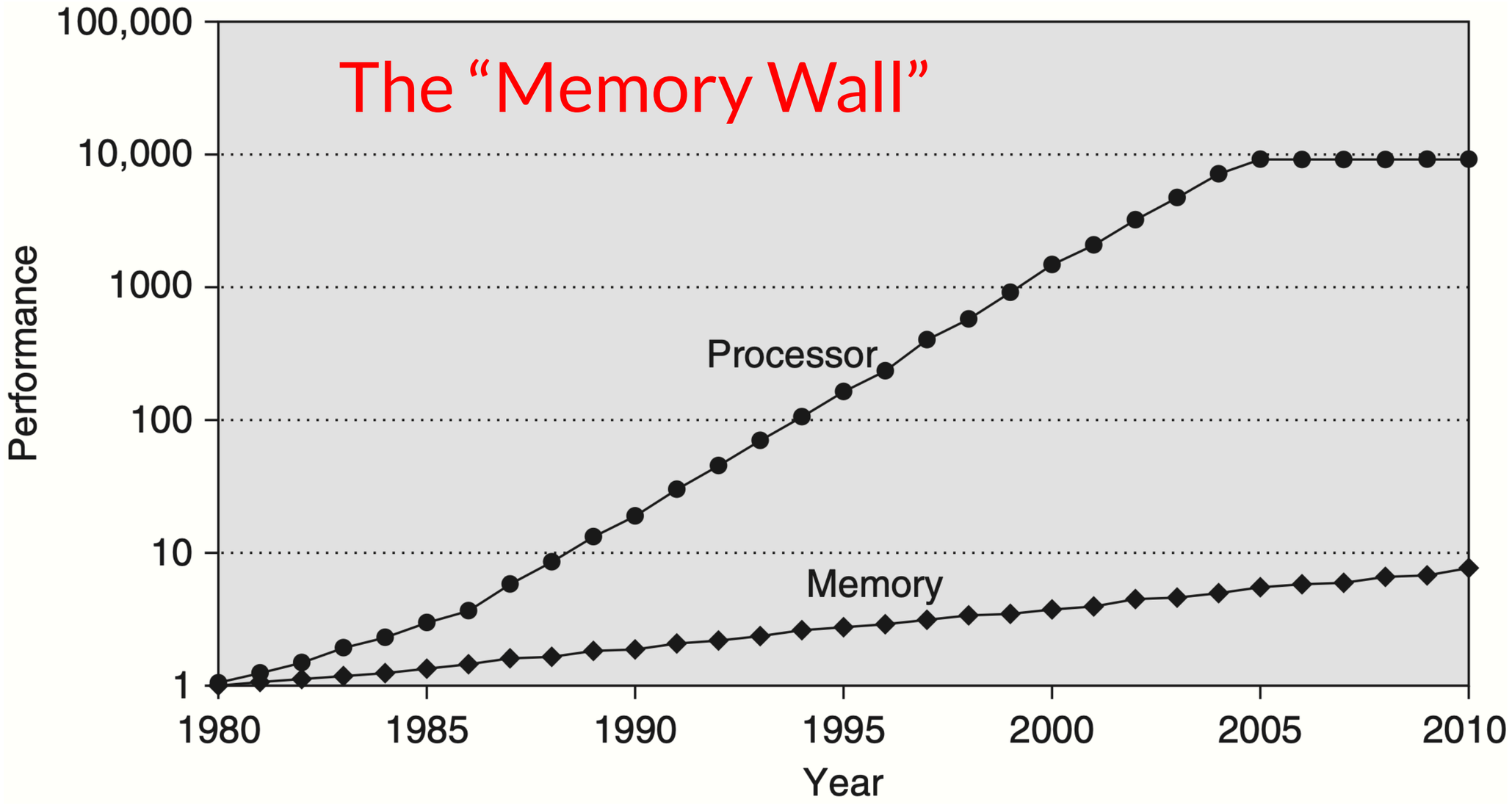


Image from “Computer Architecture: A Quantitative Approach”

Imagine you are working at Intel in the 1980s

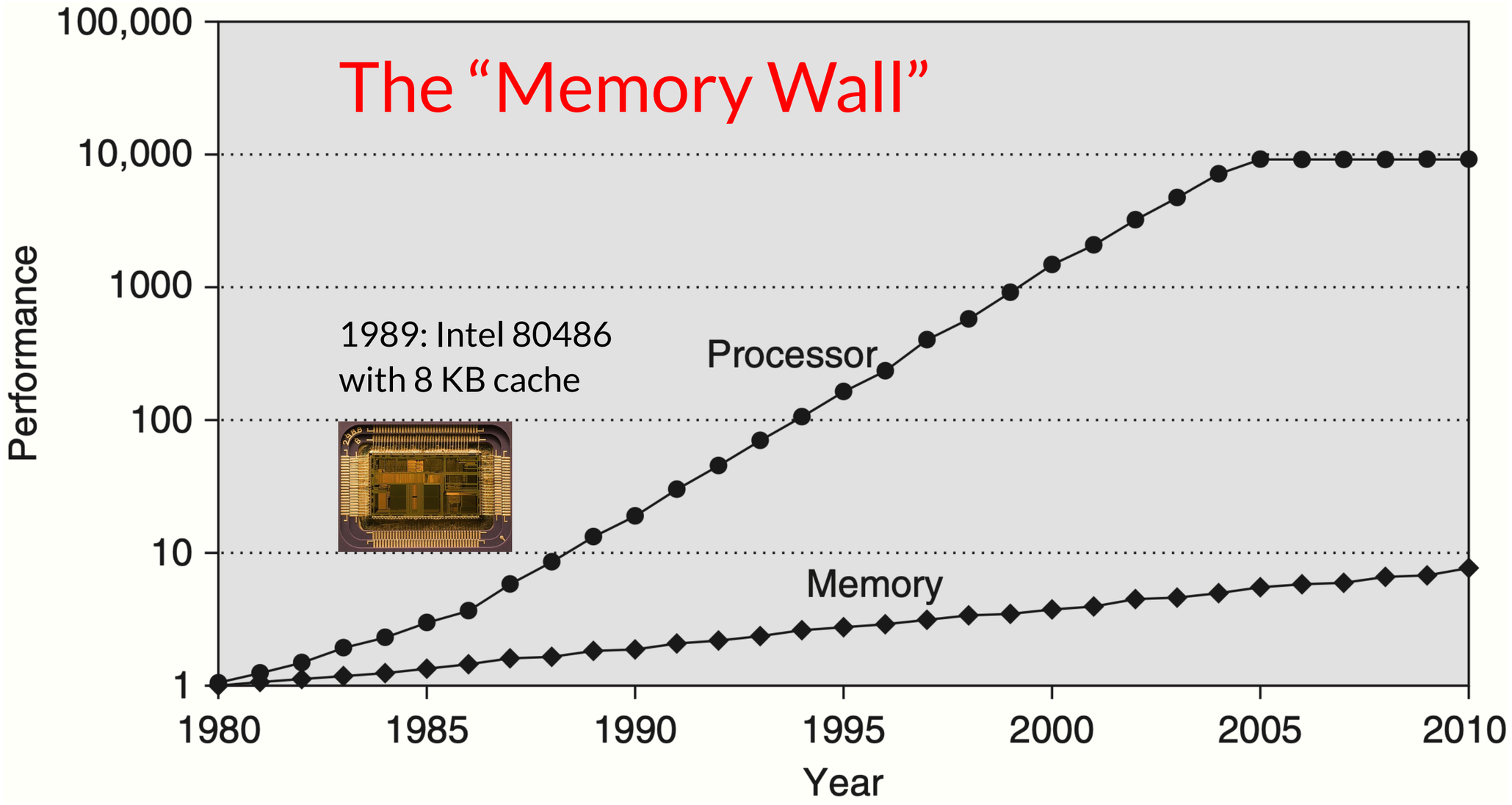
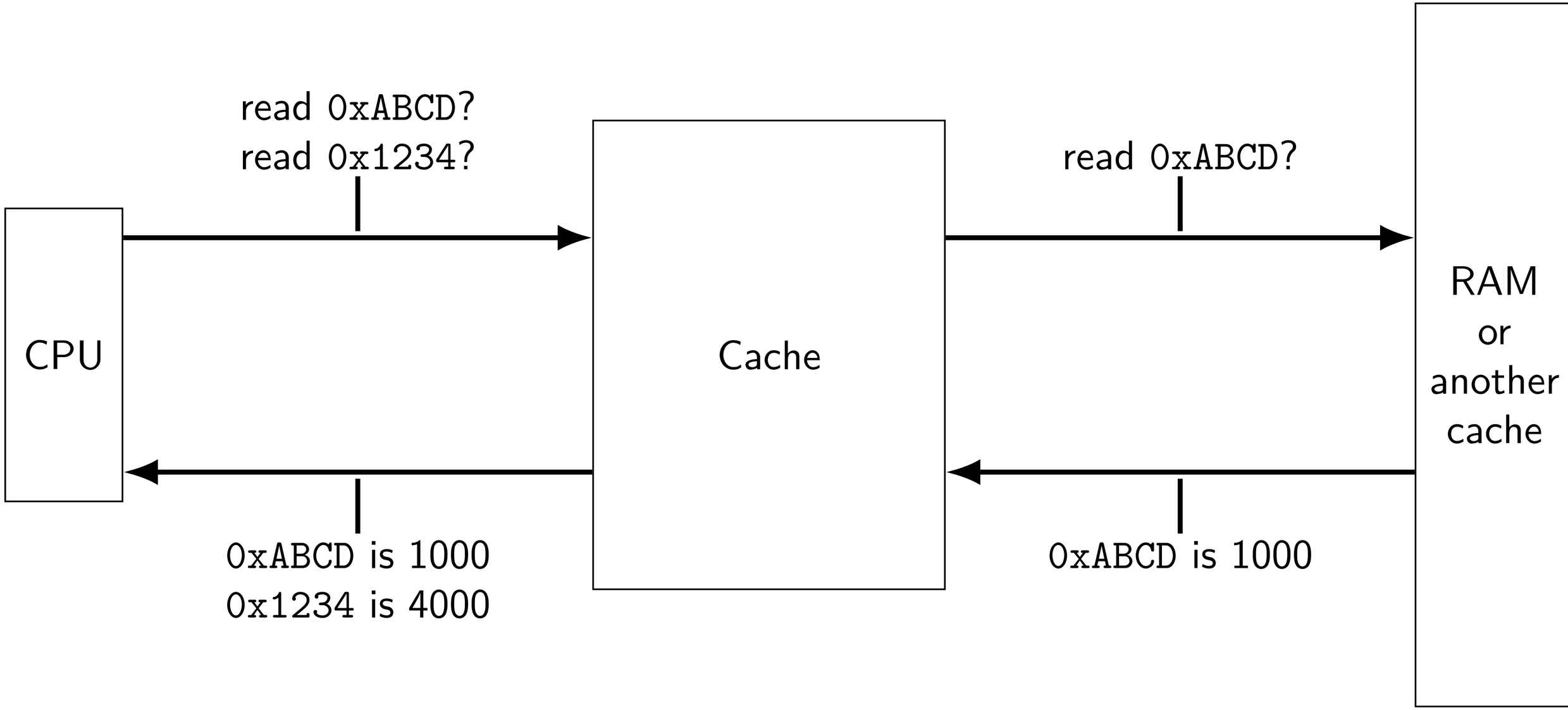
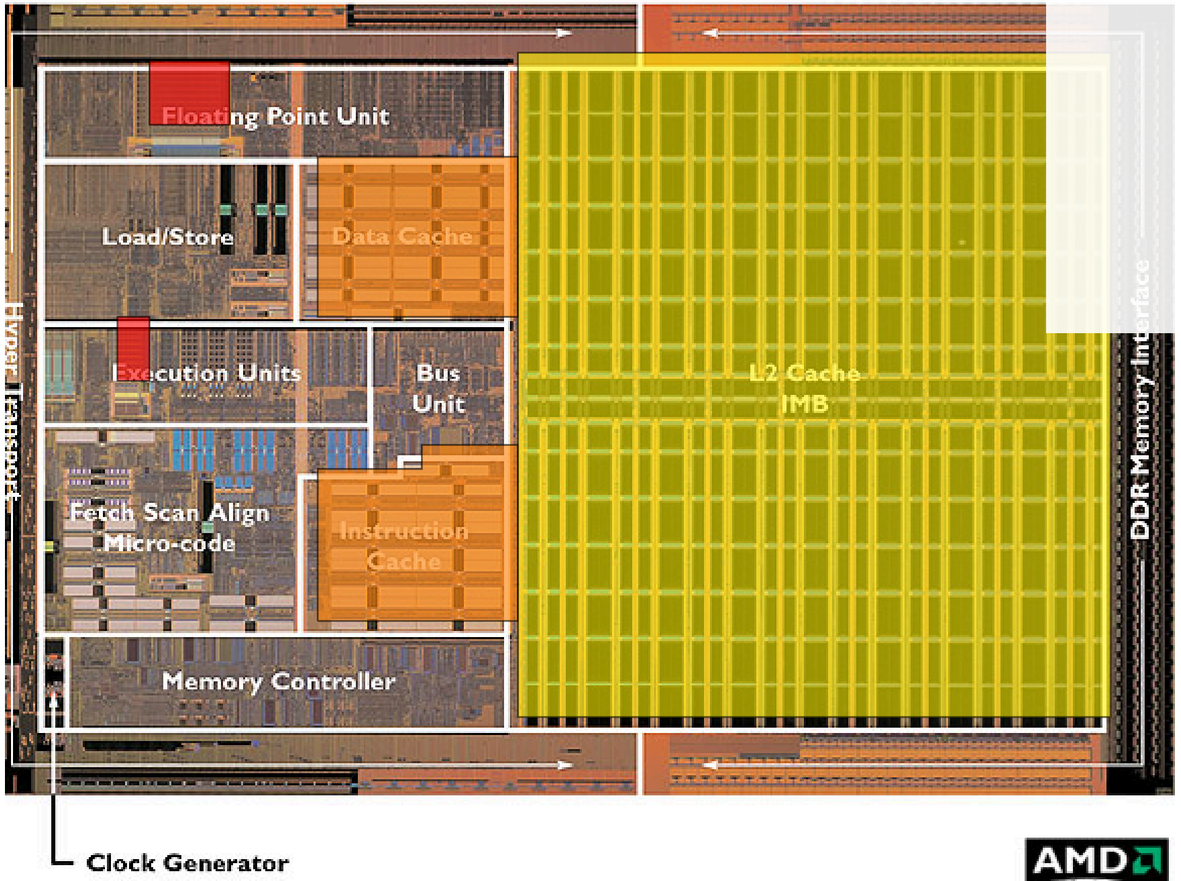


Image from “Computer Architecture: A Quantitative Approach”

the place of cache



memory hierarchy



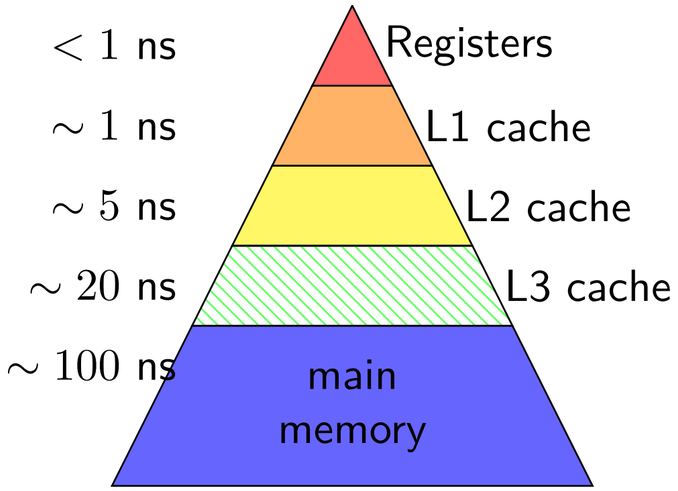


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries

memory hierarchy goals

performance of the fastest (smallest) memory

capacity of the largest (slowest) memory

how to hide 100x latency difference?

99+% hit rate ("hit" means value found in cache)

memory hierarchy assumptions

temporal locality

"if a value is accessed now, it will be accessed again soon"

caches should keep *recently accessed values*

spatial locality

"if a value is accessed now, adjacent values will be accessed soon"

caches should *store adjacent values at the same time*

natural properties of programs — think about loops

locality examples

```
double computeMean(int length, double *values) {
    double total = 0.0;
    for (int i = 0; i < length; ++i) {
        total += values[i];
    }
    return total / length;
}
```

temporal locality: machine code of the loop

spatial locality: machine code of most consecutive instructions

temporal locality: `total`, `i`, `length` accessed repeatedly

spatial locality: `values[i+1]` accessed after `values[i]`

locality exercise (1)

```
/* version 1 */
for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
        A[i] += B[j] * C[i * N + j]

/* version 2 */
for (int j = 0; j < N; ++j)
    for (int i = 0; i < N; ++i)
        A[i] += B[j] * C[i * N + j];
```

exercise: which has better temporal locality in A? in B? in C?

how about spatial locality?

locality exercise (2)

```
/* version 1 */
for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
        A[i] += B[j] * C[i * N + j]

/* version 3 */
for (int ii = 0; ii < N; ii += 32)
    for (int jj = 0; jj < N; jj += 32)
        for (int i = ii; i < ii + 32; ++i)
            for (int j = jj; j < jj + 32; ++j)
                A[i] += B[j] * C[i * N + j];
```

exercise: which has better temporal locality in A? in B? in C?

how about spatial locality?

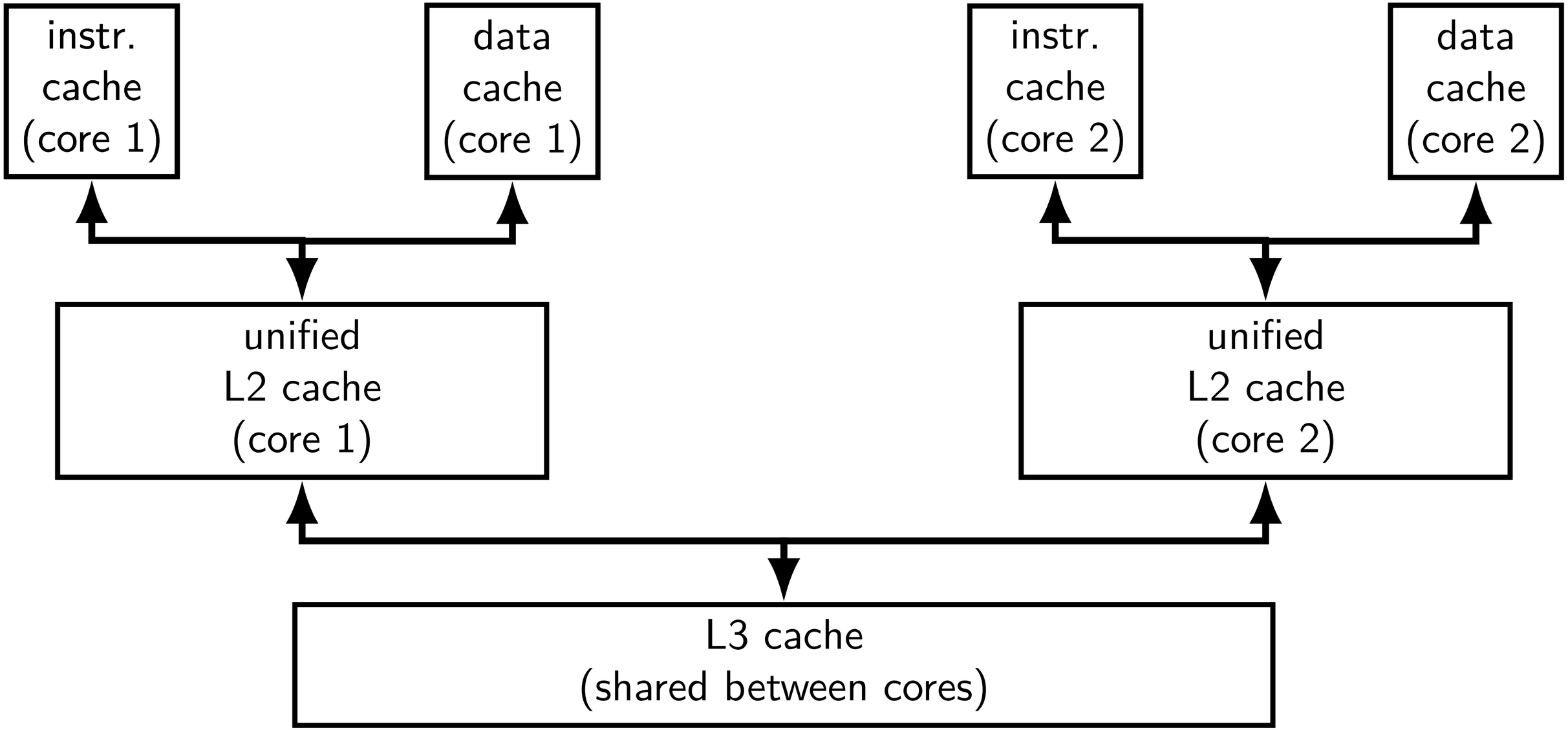
locality exercise (3)

```
struct Student {
    char name[128]; long id; float grade;
};
struct Student students[1000];

float averageGrade() {
    float sum = 0.0
    for (int i = 0; i < 1000; i += 1)
        sum += students[i].grade;
    return sum / 1000.0;
}
```

```
char studentNames[1000][128];
long studentIds[1000];
float studentGrades[1000];

float average() {
    float sum = 0.0
    for (int i = 0; i < 1000; i += 1)
        sum += studentGrades[i];
    return sum / 1000.0;
}
```

better spatial/temporal locality?

split caches; multiple cores (one design)



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: handle fewer accesses at a time

cache analogy: you finding words in books

library: main memory

- contains all the words in all books

copy of books you took home: L2 cache

- faster to look in books already at home

copy of individual book pages in your backpack: L1 cache

- fastest to look at book pages you have with you

one-block cache

one-block cache

Cache

value
00 00

Memory

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

one-block cache

decision: divide memory into two-byte blocks
put exactly one of these blocks in the cache

Cache

value
00 00

Memory

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

one-block cache

read byte at 01011?

Cache

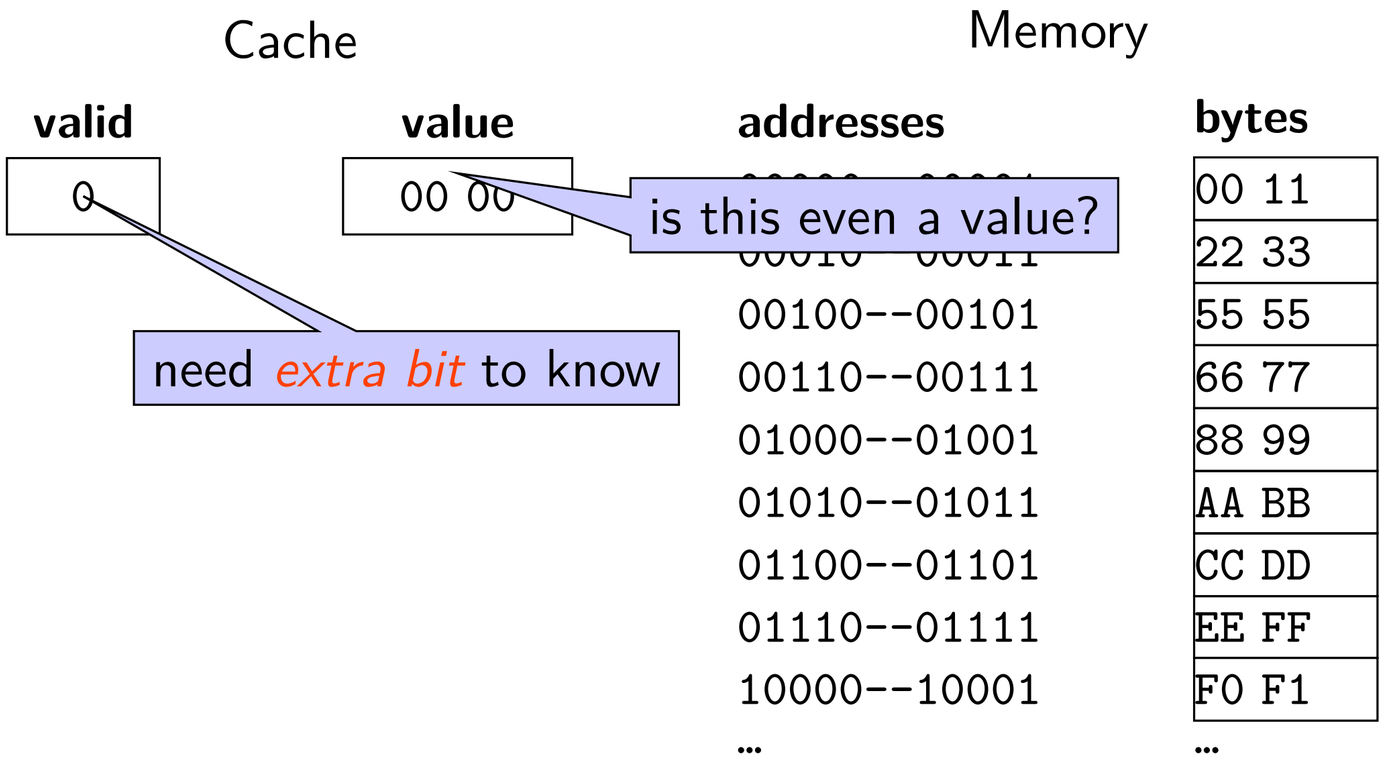
value
00 00

Memory

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

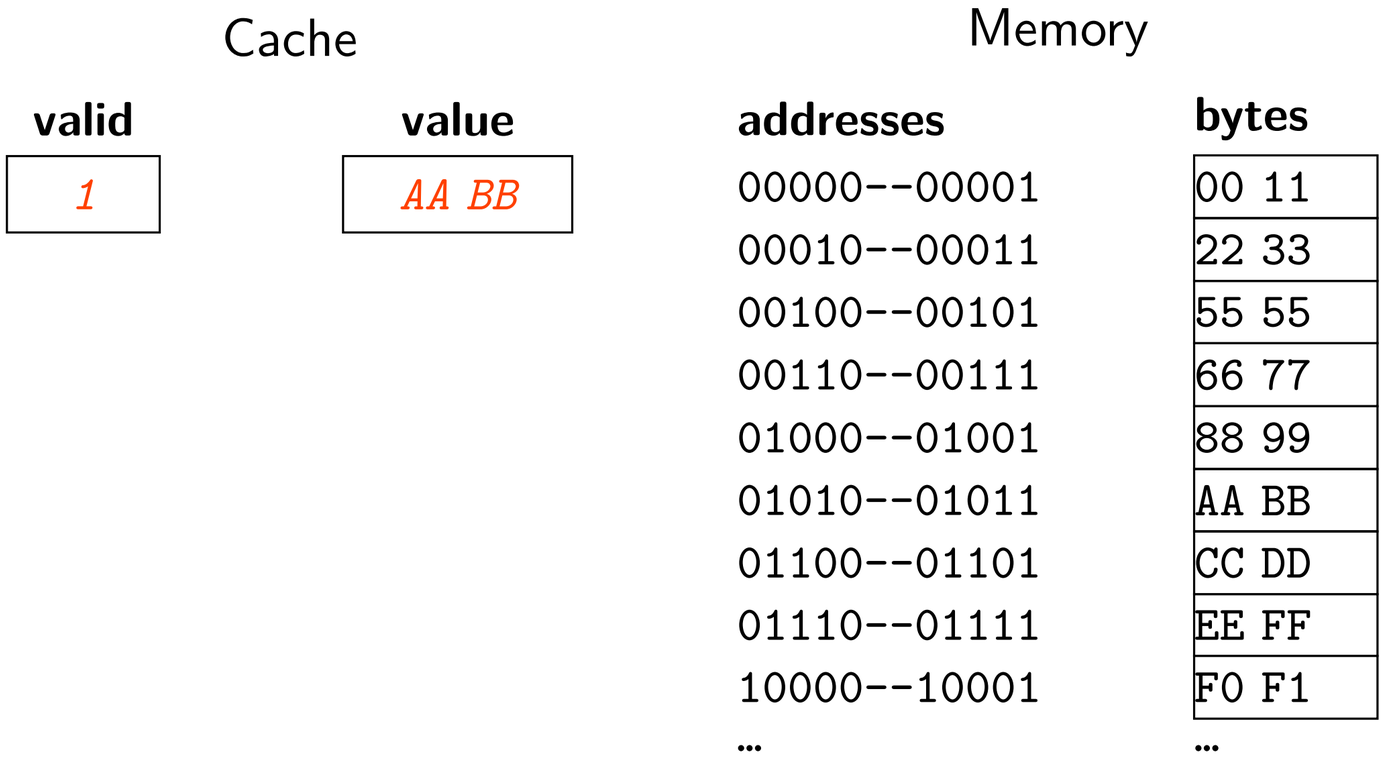
one-block cache

read byte at 01011?



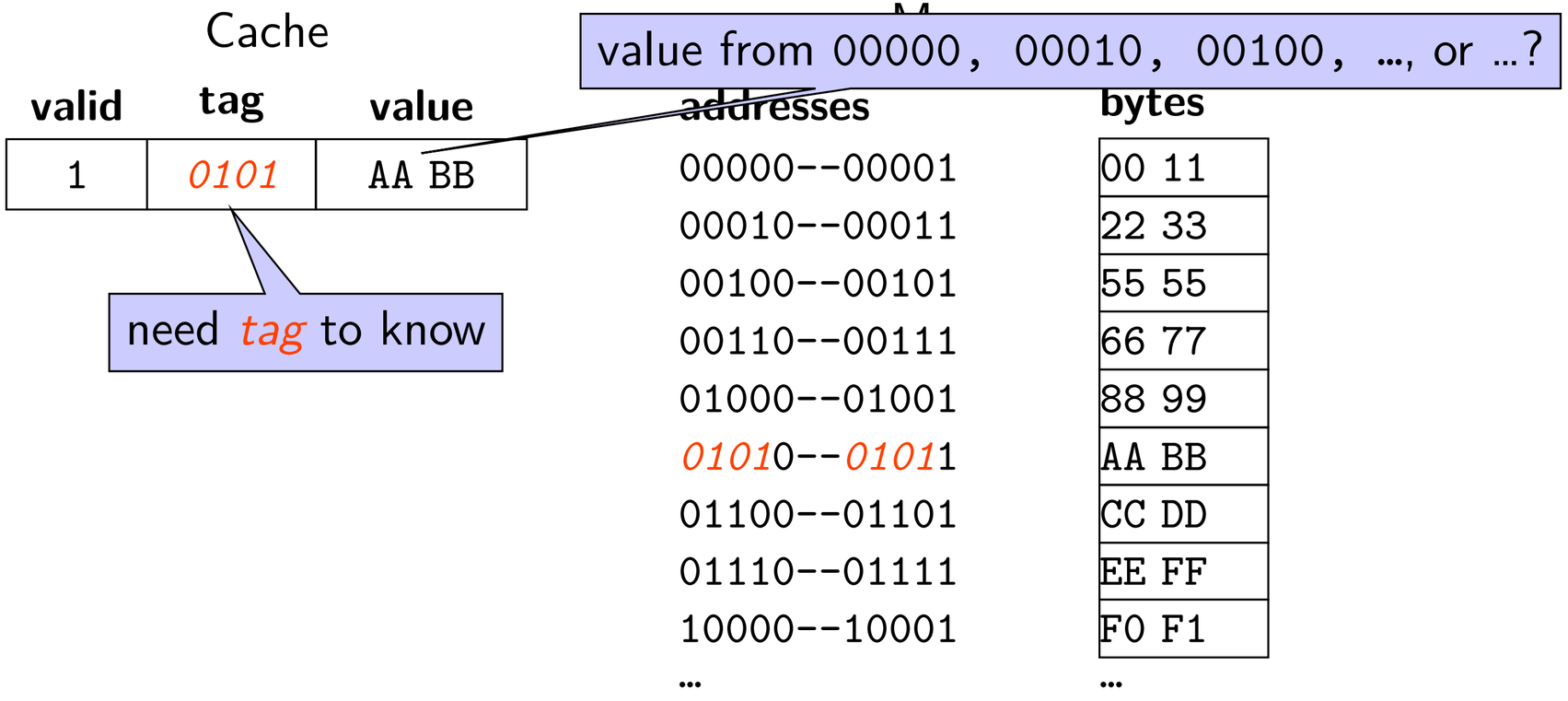
one-block cache

read byte at 01011?
invalid, fetch



one-block cache

read byte at *0101*1?



one-block cache

read byte at 01011*1*?

Cache

valid	tag	value
1	0101	AA *BB*

Memory

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

building a (direct-mapped) cache

building a (direct-mapped) cache

Cache

value
00 00
00 00
00 00
00 00

cache block: *2 bytes*

Memory

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

building a (direct-mapped) cache

read byte at 01011?

Cache

value
00 00
00 00
00 00
00 00

cache block: 2 bytes

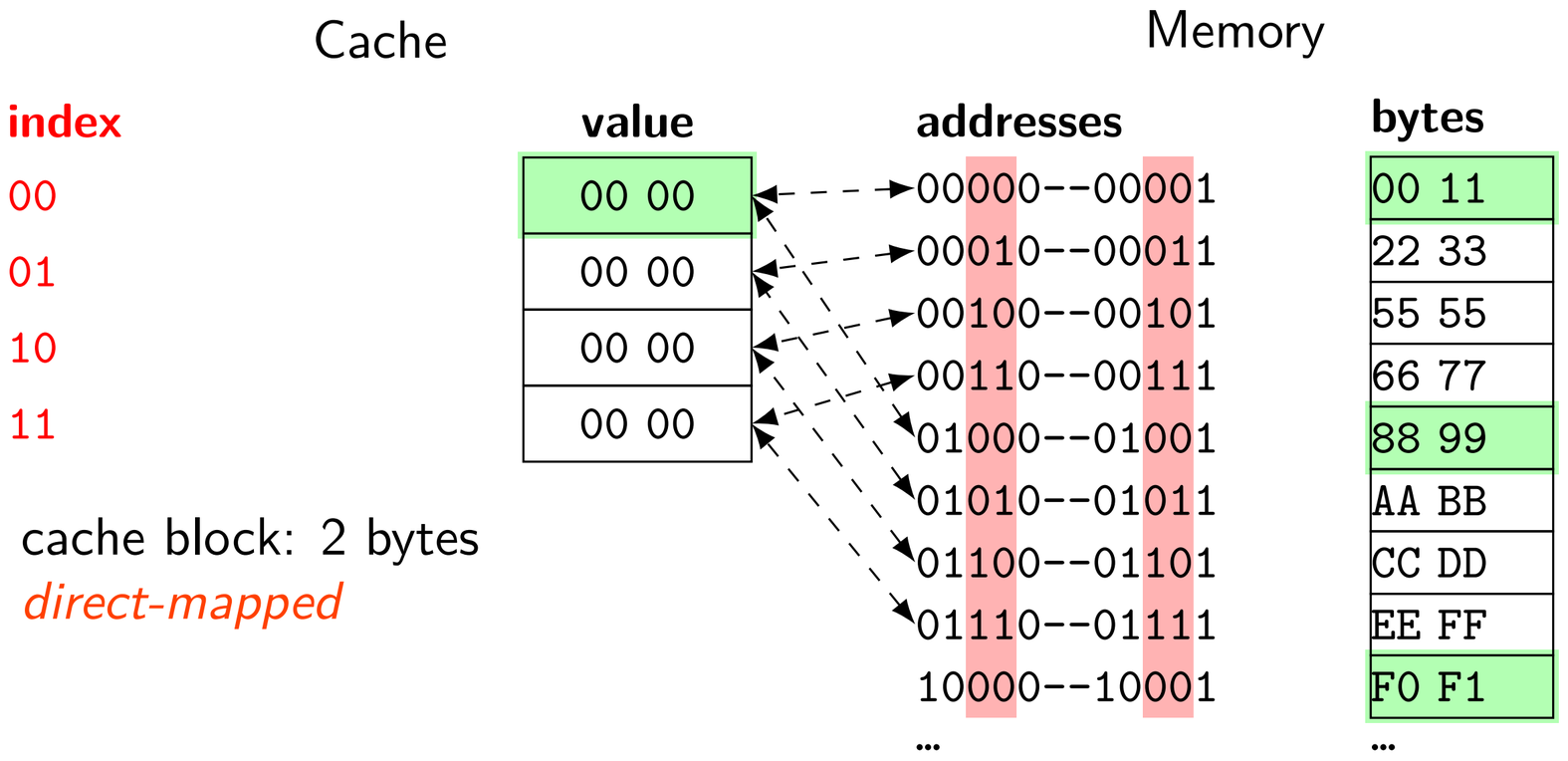
Memory

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

building a (direct-mapped) cache

read byte at 01011?

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

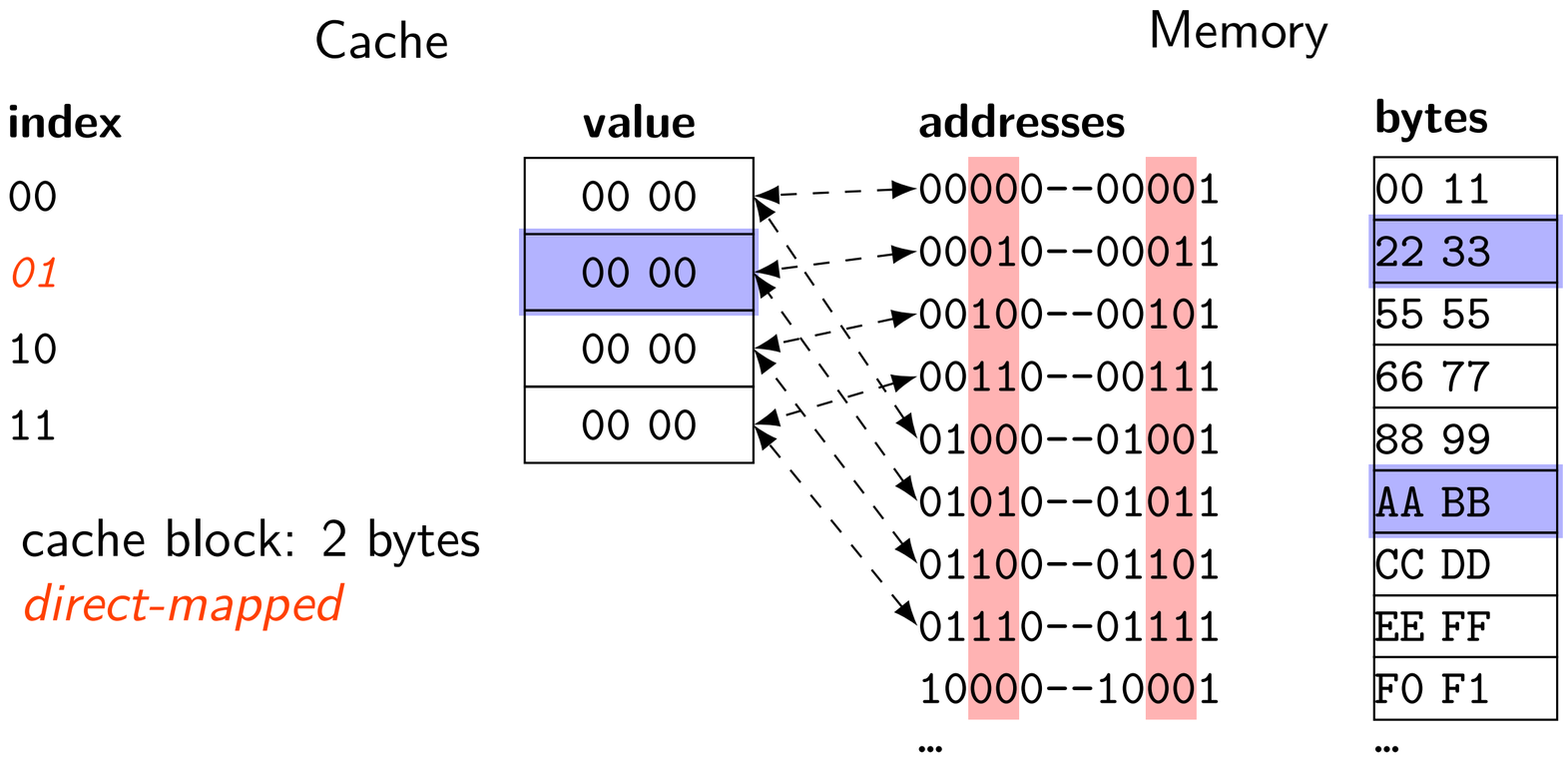


cache block: 2 bytes
direct-mapped

building a (direct-mapped) cache

read byte at 01*01*1?

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

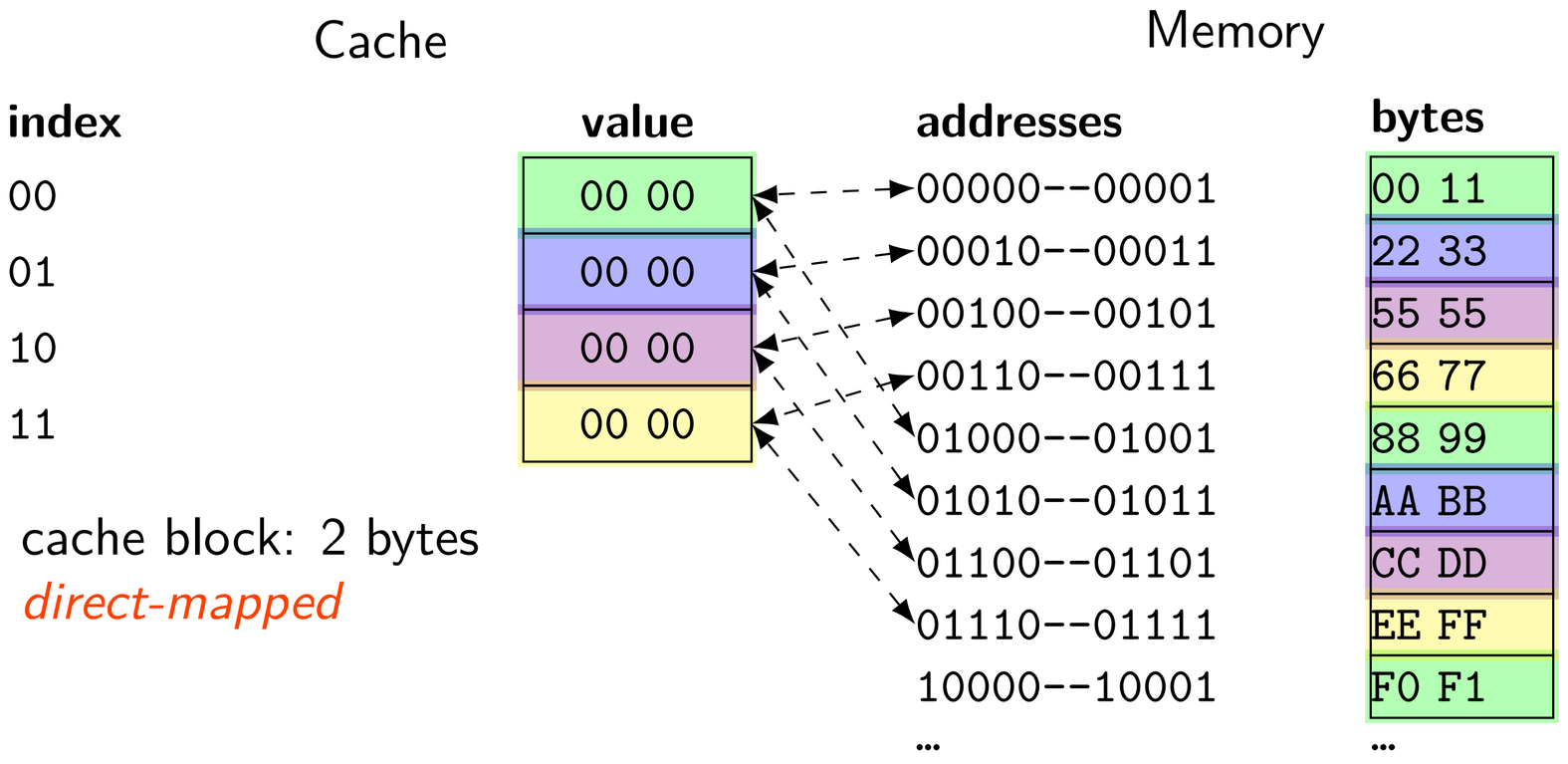


cache block: 2 bytes
direct-mapped

building a (direct-mapped) cache

read byte at 01*01*1?

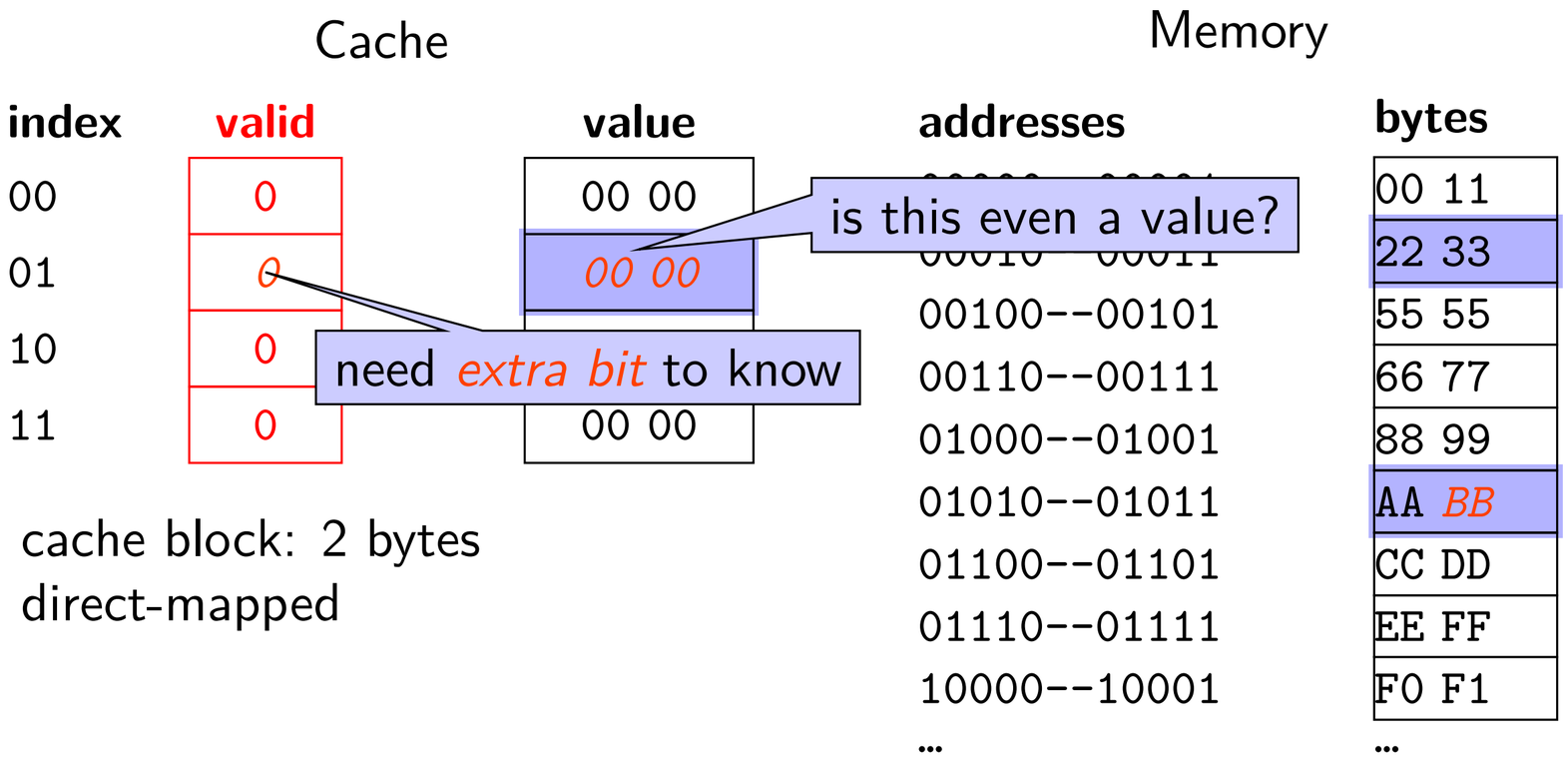
exactly *one place* for each address
spread out what can go in a block



cache block: 2 bytes
direct-mapped

building a (direct-mapped) cache

read byte at 01011?



cache block: 2 bytes
direct-mapped

building a (direct-mapped) cache

read byte at 01011?
invalid, fetch

Cache

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

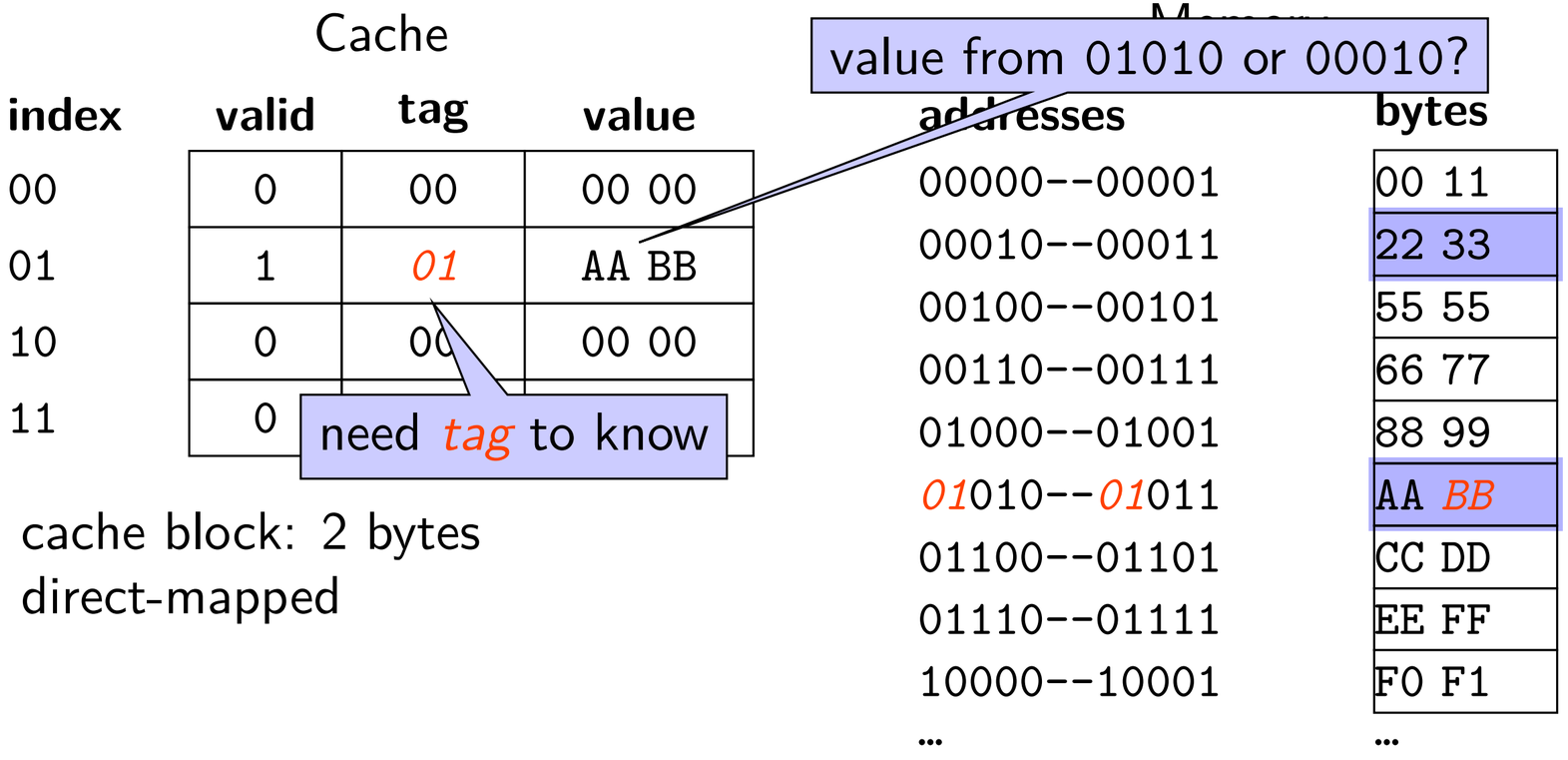
cache block: 2 bytes
direct-mapped

Memory

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

building a (direct-mapped) cache

read byte at 01011?
invalid, fetch



cache block: 2 bytes
direct-mapped

terminology

row = set

preview: change how much is in a row

cache analogy: you finding words in books

block size is 1 book page

book title: tag

page number: index

word index on book page: offset

Tag-Index-Offset (TIO)

Tag-Index-Offset (TIO)

address `001111` (stores value `0xFF`)

cache	tag	index	offset
2 byte blocks, 4 sets			
2 byte blocks, 8 sets			
4 byte blocks, 2 sets			

2 byte blocks, 4 sets

index	valid	tag	value
00	1	000	00 11
01	1	001	AA BB
10	0	--	-- --
11	1	001	EE *FF*

4 byte blocks, 2 sets

index	valid	tag	value
0	1	000	00 11 22 33
1	1	001	CC DD EE *FF*

2 byte blocks, 8 sets

index	valid	tag	value
000	1	00	00 11
001	1	01	F1 F2
010	0	--	-- --
011	0	--	-- --
100	0	--	-- --
101	1	00	AA BB
110	0	--	-- --
111	1	00	EE *FF*

Tag-Index-Offset (TIO)

address `00111`*1* (stores value `0xFF`)

cache	tag	index	offset
2 byte blocks, 4 sets			1
2 byte blocks, 8 sets			1
4 byte blocks, 2 sets			

2 byte blocks, 4 sets

index	valid	tag	value
00	1	000	00 11
01	1	001	AA BB
10	0	--	-- --
11	1	001	EE *FF*

$2 = 2^1$ bytes in block
1 bit to say which byte

2 byte blocks, 8 sets

index	valid	tag	value
000	1	00	00 11
001	1	01	F1 F2
010	0	--	-- --
011	0	--	-- --
100	0	--	-- --
101	1	00	AA BB
110	0	--	-- --
111	1	00	EE *FF*

4 byte blocks, 2 sets

index	valid	tag	value
0	1	000	00 11 22 33
1	1	001	CC DD EE *FF*

Tag-Index-Offset (TIO)

address 0011*11* (stores value 0xFF)

cache	tag	index	offset
2 byte blocks, 4 sets			1
2 byte blocks, 8 sets			1
4 byte blocks, 2 sets			11

2 byte blocks, 4 sets

index	valid	tag	value
00	1	000	00 11
01	1	001	AA BB
10	0	--	-- --
11	1	001	

4 byte blo

index	valid	tag	value
0	1	000	00 11 22 33
1	1	001	CC DD EE *FF*

$4 = 2^2$ bytes in block
2 bits to say which byte

2 byte blocks, 8 sets

index	valid	tag	value
000	1	00	00 11
001	1	01	F1 F2
010	0	--	-- --
11	0	--	-- --
00	0	--	-- --
101	1	00	AA BB
110	0	--	-- --
111	1	00	EE *FF*

Tag-Index-Offset (TIO)

address `001111` (stores value `0xFF`)

cache	tag	index	offset
2 byte blocks, *4 sets*		11	1
2 byte blocks, *8 sets*			1
4 byte blocks, *2 sets*		1	11

2 byte blocks, 4 sets

index	valid	tag	value
00	1	000	00 11
01	1	001	AA BB
10	0	--	-- --
11	1	001	EE *FF*

4 byte blocks, 2 sets

index	valid	tag	value
0	1	000	00 11 22 33
1	1	001	CC DD EE *FF*

$2^2 = 4$ sets
2 bits to index set

2 byte blocks, 8 sets

index	valid	tag	value
000	1	00	00 11
		01	F1 F2
		--	-- --
		--	-- --
100	0	--	-- --
101	1	00	AA BB
110	0	--	-- --
111	1	00	EE *FF*

Tag-Index-Offset (TIO)

address `001111` (stores value `0xFF`)

cache	tag	index	offset
2 byte blocks, 4 sets		11	1
2 byte blocks, 8 sets		111	1
4 byte blocks, 2 sets		1	11

2 byte blocks, 4 sets

index	valid	tag	value
00	1	000	00 11
01	1	001	AA BB
10	0	--	-- --
11	1	001	

4 byte bl

index	valid	tag	value
0	1	000	00 11 22 33
1	1	001	CC DD EE *FF*

$2^3 = 8$ sets

3 bits to index set

2 byte blocks, 8 sets

index	valid	tag	value
000	1	00	00 11
001	1	01	F1 F2
010	0	--	-- --
011	0	--	-- --
100	0	--	-- --
101	1	00	AA BB
110	0	--	-- --
111	1	00	EE *FF*

Tag-Index-Offset (TIO)

address `001111` (stores value `0xFF`)

cache	tag	index	offset
2 byte blocks, 4 sets		11	1
2 byte blocks, 8 sets		111	1
4 byte blocks, 2 sets		1	11

2 byte blocks, 4 sets

index	valid	tag	value
00	1	000	00 11
01	1	001	AA BB
10	0	--	-- --
11	1	001	EE *FF*

4 byte blocks, 2 sets

index	valid	tag	value
0	1	000	00 11 22 33
1	1	001	CC DD EE *FF*

2 byte blocks, 8 sets

index	valid	tag	value
000	1	00	00 11
001	1	01	F1 F2
010	0	--	-- --
011	0	--	-- --
100			-- --
101			AA BB
110	0	--	-- --
111	1	00	EE *FF*

$2^1 = 2$ sets
1 bit to index set

Tag-Index-Offset (TIO)

address `001111` (stores value `0xFF`)

cache	tag	index	offset
2 byte blocks, 4 sets	001	11	1
2 byte blocks, 8 sets	00	111	1
4 byte blocks, 2 sets	001	1	11

tag — whatever is left over

index	valid	tag	value
00	1	000	00 11
01	1	001	AA BB
10	0	--	-- --
11	1	001	EE *FF*

4 byte blocks, 2 sets

index	valid	tag	value
0	1	000	00 11 22 33
1	1	001	CC DD EE *FF*

2 byte blocks, 8 sets

index	valid	tag	value
000	1	00	00 11
001	1	01	F1 F2
010	0	--	-- --
011	0	--	-- --
100	0	--	-- --
101	1	00	AA BB
110	0	--	-- --
111	1	00	EE *FF*

cache size

cache size = amount of *data* in cache

not included metadata (tags, valid bits, etc.)

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)

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`

cache size exercise

A system uses a direct-mapped cache with 32 byte blocks.

Two memory accesses are made:

- Read `0x08249`
- Read `0x0c658`

What is the fewest number of cache sets this cache could have to prevent the second read from evicting the first?

What is the size of this cache?

example access pattern (1)

example access pattern (1)

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

2 byte blocks, 4 sets

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

example access pattern (1)

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

2 byte blocks, 4 sets

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

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$m = 8$ bit addresses
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

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

tag index offset

2 byte blocks, 4 sets

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

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

$b = 1$ (block) offset bits

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

$m = 8$ bit addresses

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

example access pattern (1)

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

tag index offset

2 byte blocks, 4 sets

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

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$m = 8$ bit addresses
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

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

tag index offset

2 byte blocks, 4 sets

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

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

$b = 1$ (block) offset bits

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

$m = 8$ bit addresses

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

example access pattern (1)

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

tag index offset

2 byte blocks, 4 sets

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

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$m = 8$ bit addresses
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

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

tag index offset

2 byte blocks, 4 sets

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

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$m = 8$ bit addresses
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

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

2 byte blocks, 4 sets

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

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$m = 8$ bit addresses
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

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

2 byte blocks, 4 sets

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

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$m = 8$ bit addresses
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

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

2 byte blocks, 4 sets

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		

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$m = 8$ bit addresses
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

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

2 byte blocks, 4 sets

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		

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$m = 8$ bit addresses
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

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

2 byte blocks, 4 sets

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*

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$m = 8$ bit addresses
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

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

2 byte blocks, 4 sets

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*

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$m = 8$ bit addresses
$t = m - (s + b) = 5$ tag bits

exercise

exercise

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

4 byte blocks, 4 sets

index	valid	tag	value
00			
01			
10			
11			

exercise

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

4 byte blocks, 4 sets

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

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

4 byte blocks, 4 sets

index	valid	tag	value
00			
01			
10			
11			

$B = 4 = 2^b$ byte block size
$b = 2$ (block) offset bits
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$m = 8$ bit addresses
$t = m - (s + b) = 4$ tag bits

exercise

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

tag index offset

4 byte blocks, 4 sets

index	valid	tag	value
00			
01			
10			
11			

$B = 4 = 2^b$ byte block size
$b = 2$ (block) offset bits
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$m = 8$ bit addresses
$t = m - (s + b) = 4$ tag bits

exercise

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

tag index offset

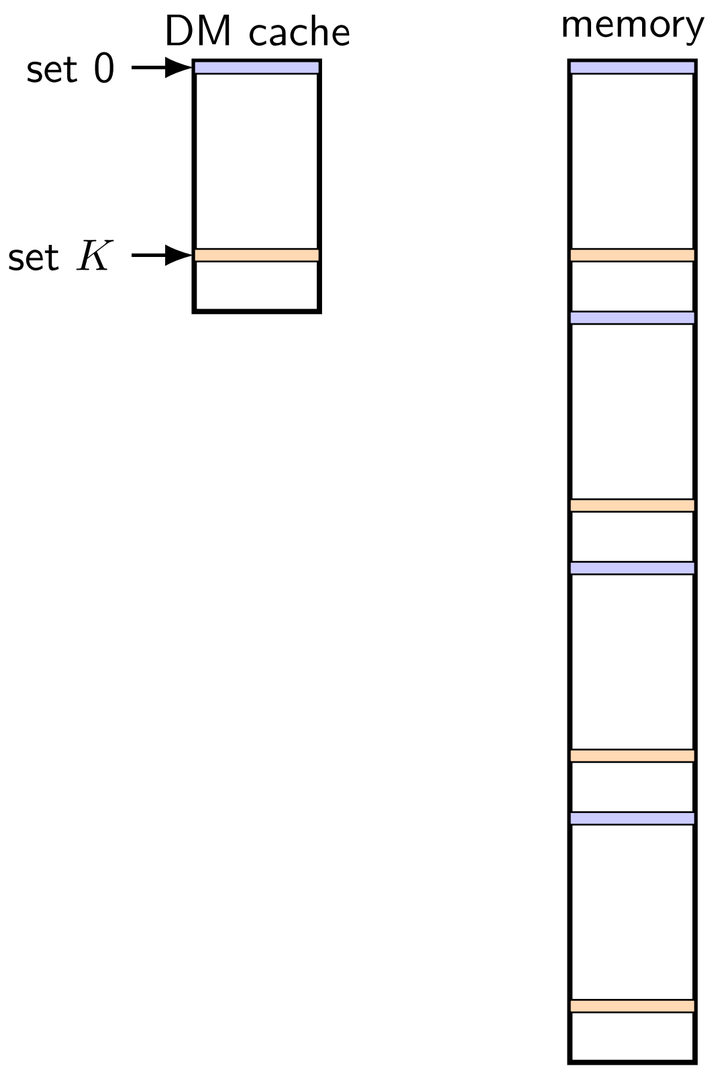
exercise: which accesses are hits?

4 byte blocks, 4 sets

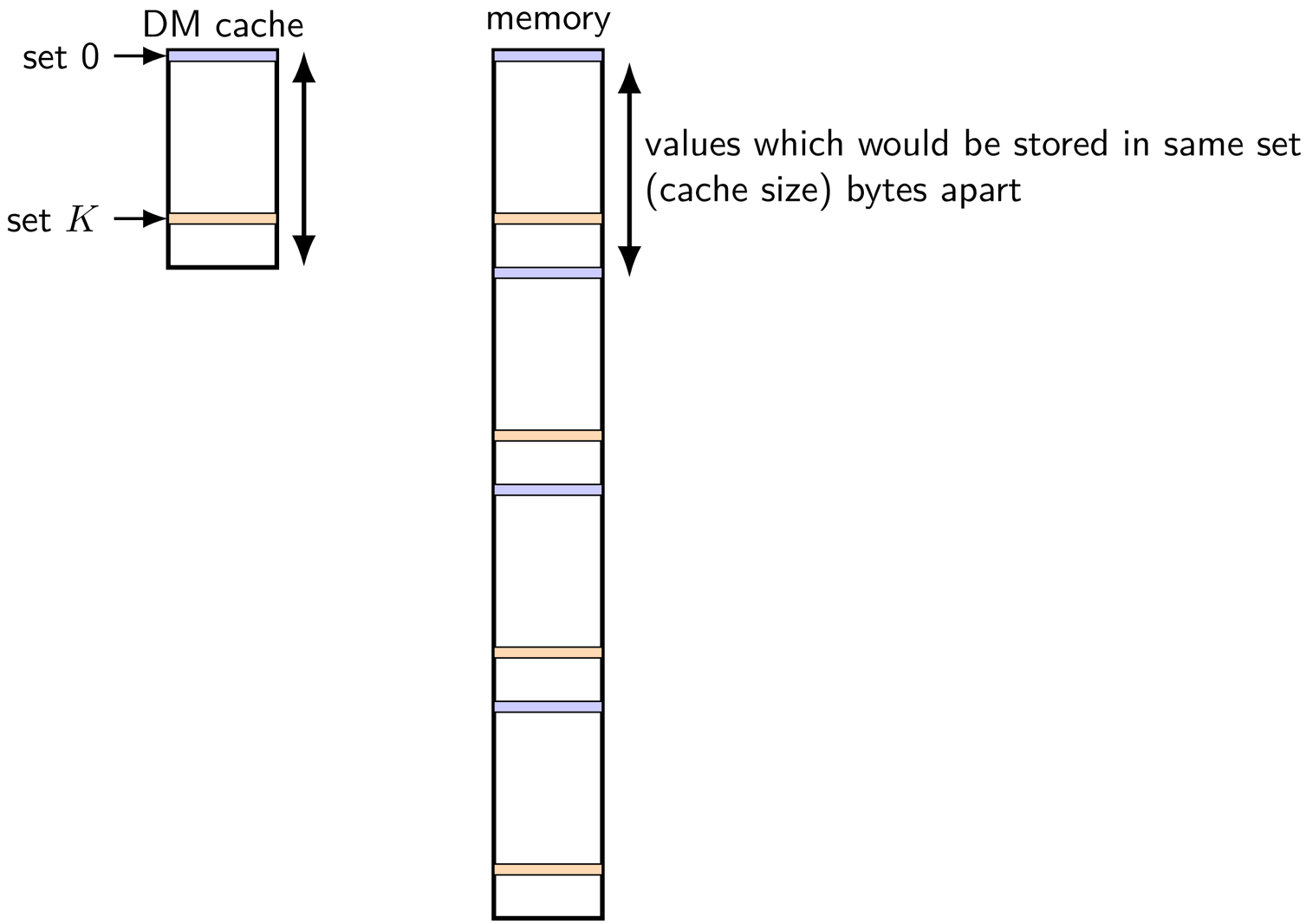
index	valid	tag	value
00			
01			
10			
11			

mapping of sets to memory (direct-mapped)

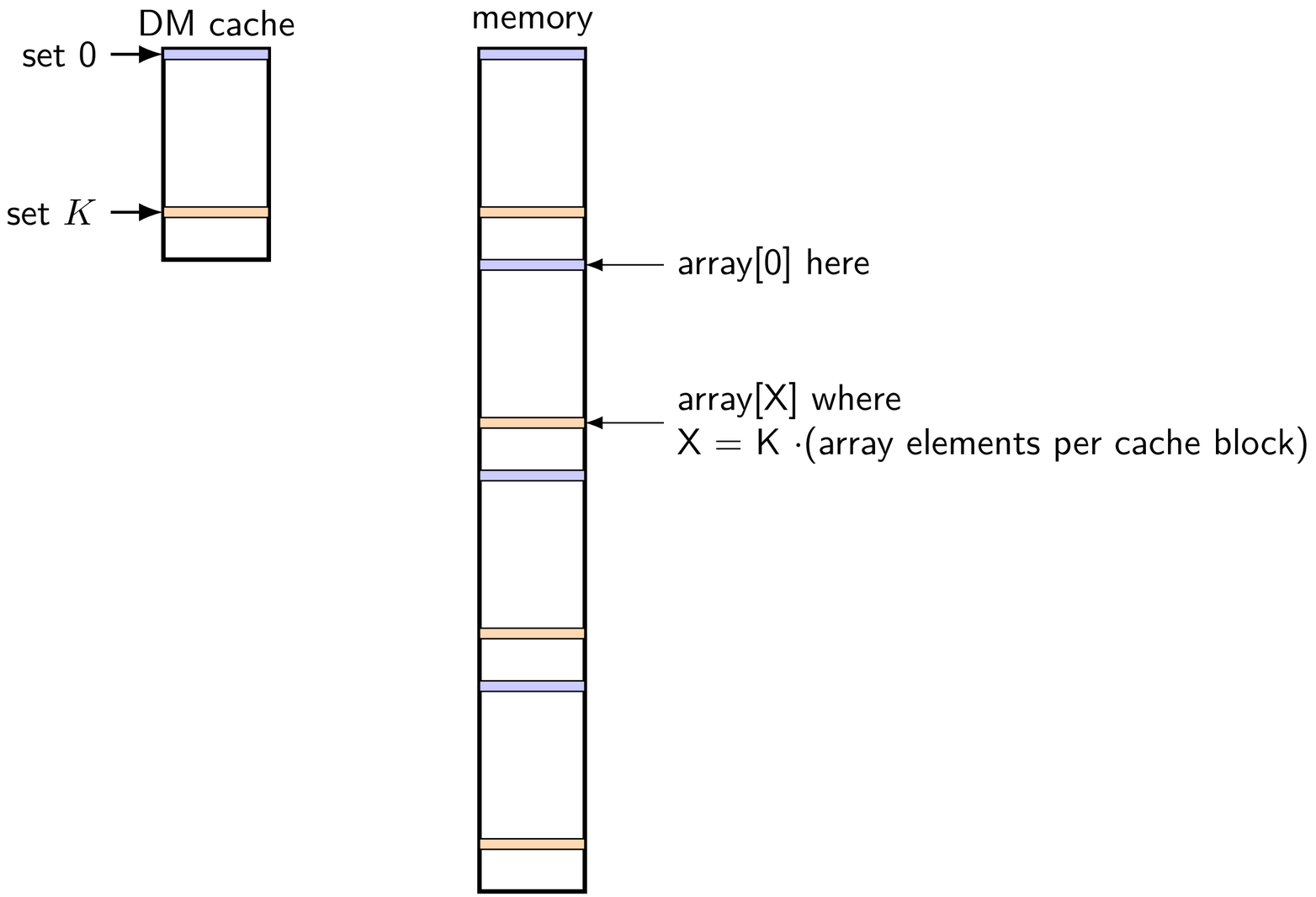
mapping of sets to memory (direct-mapped)



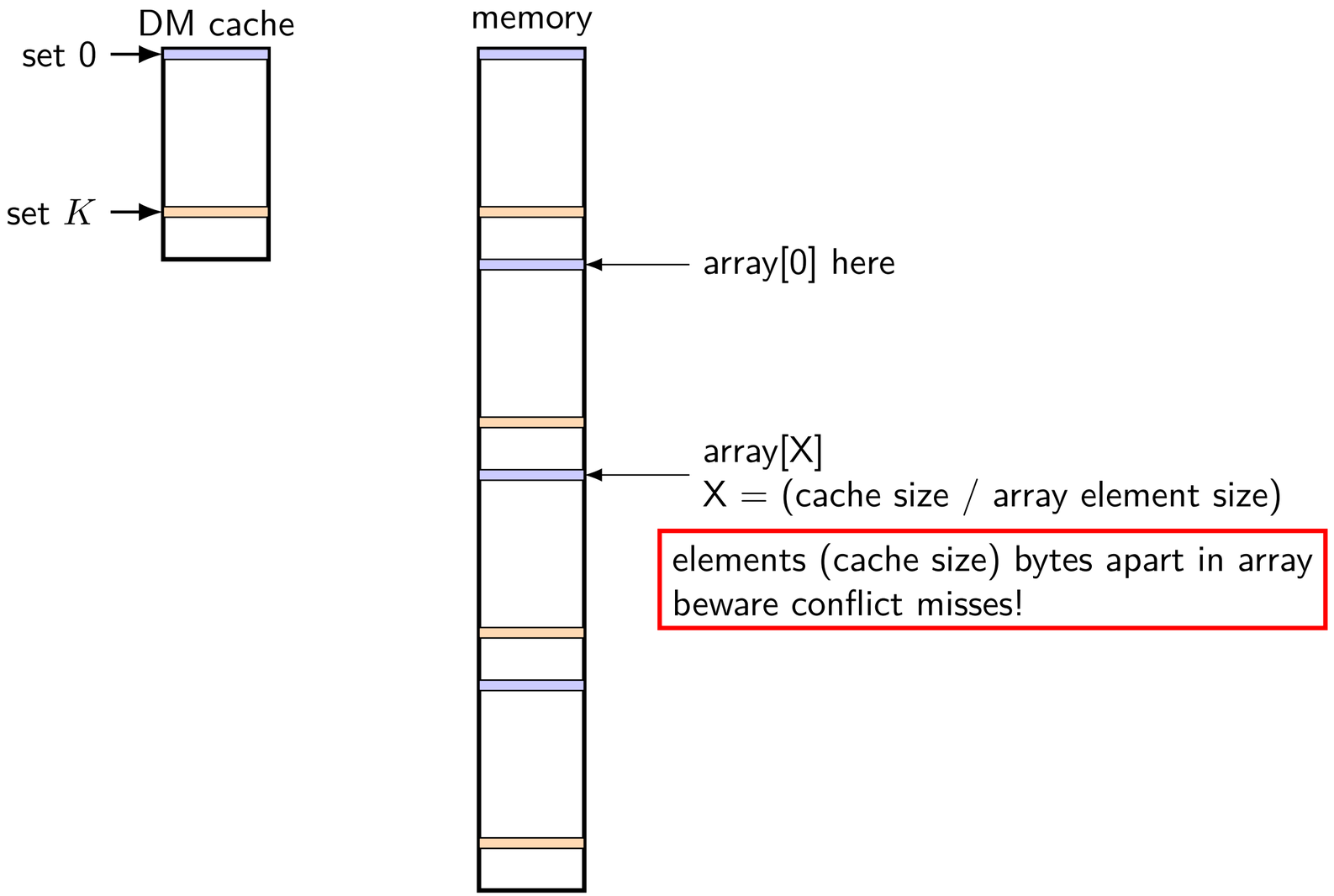
mapping of sets to memory (direct-mapped)



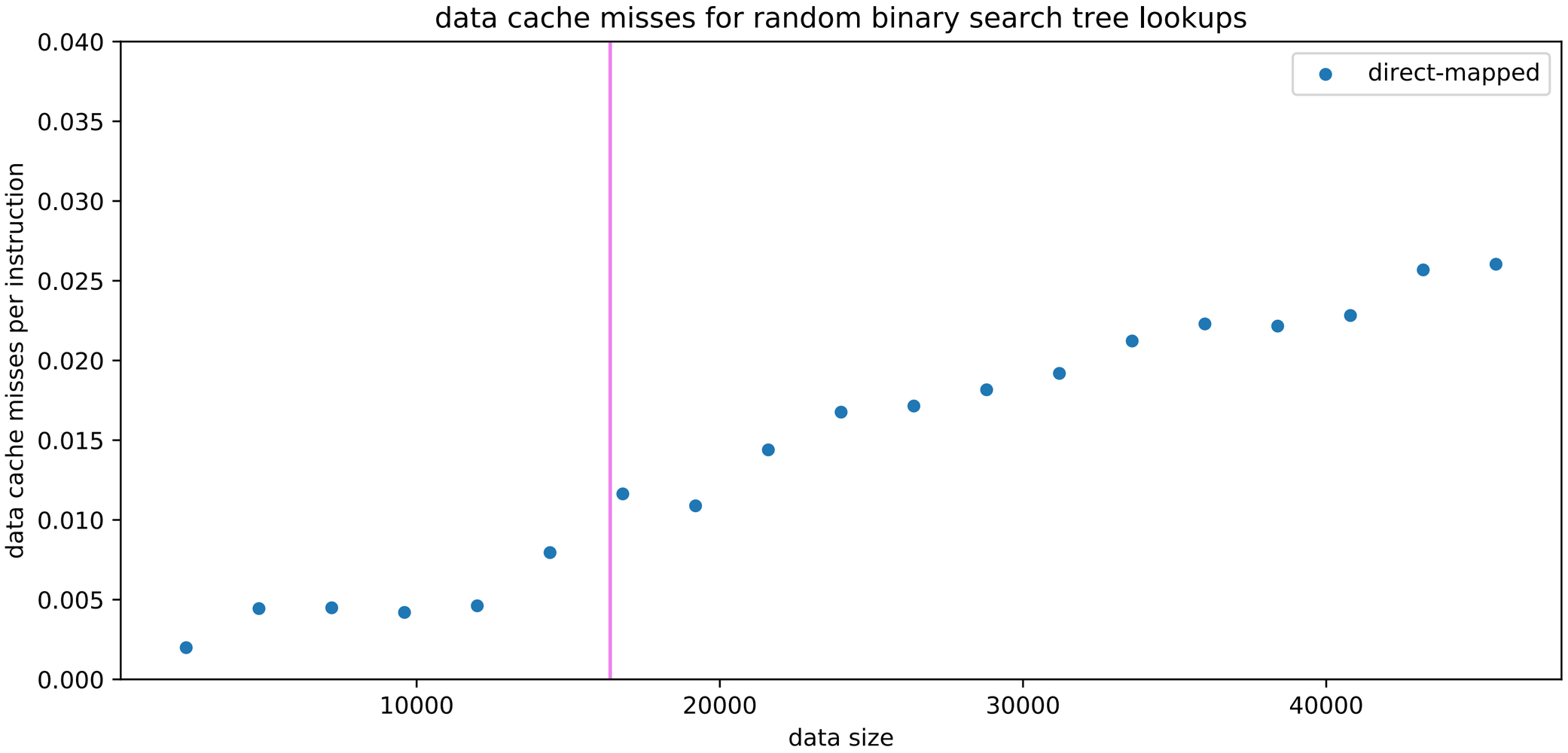
mapping of sets to memory (direct-mapped)



mapping of sets to memory (direct-mapped)

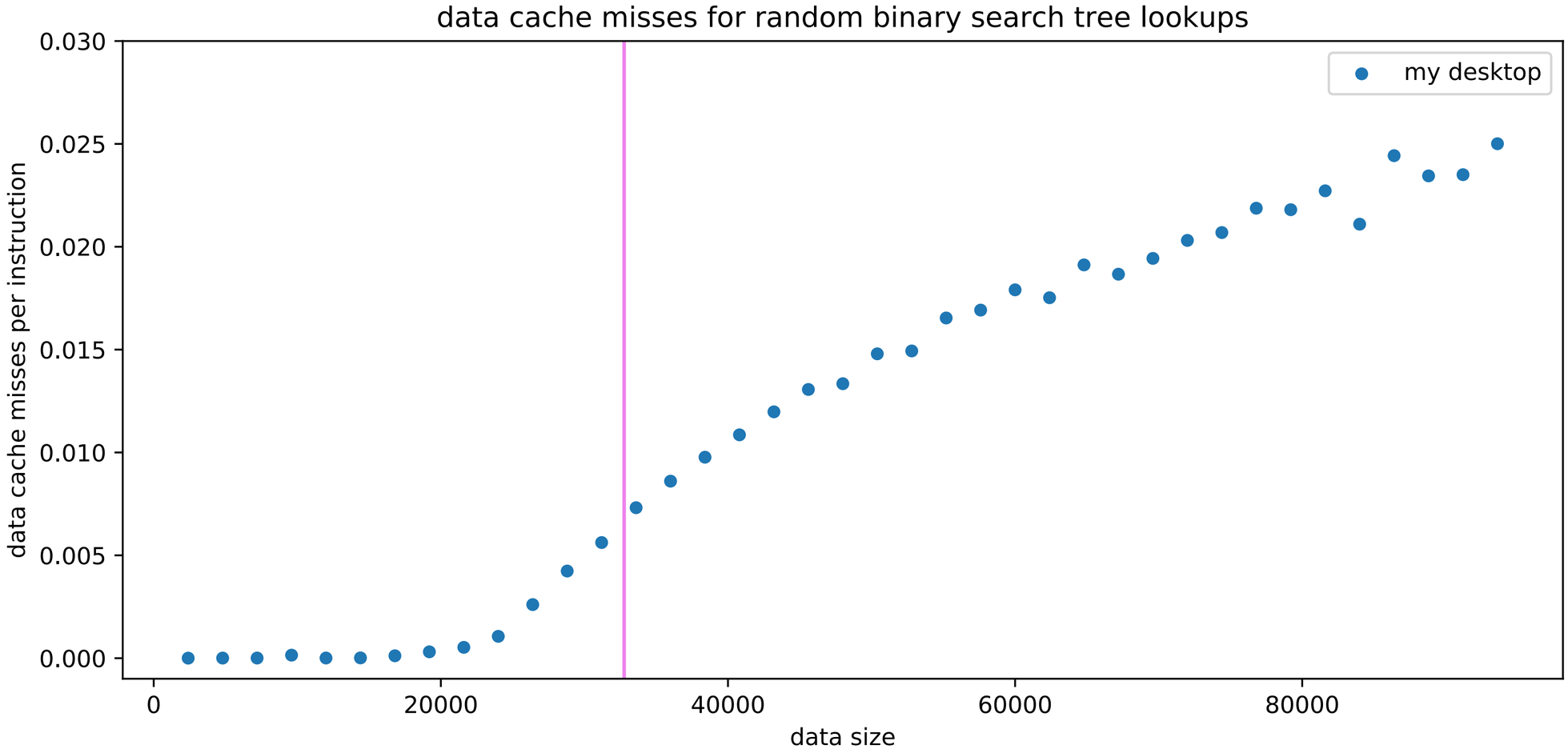


simulated misses: BST lookups



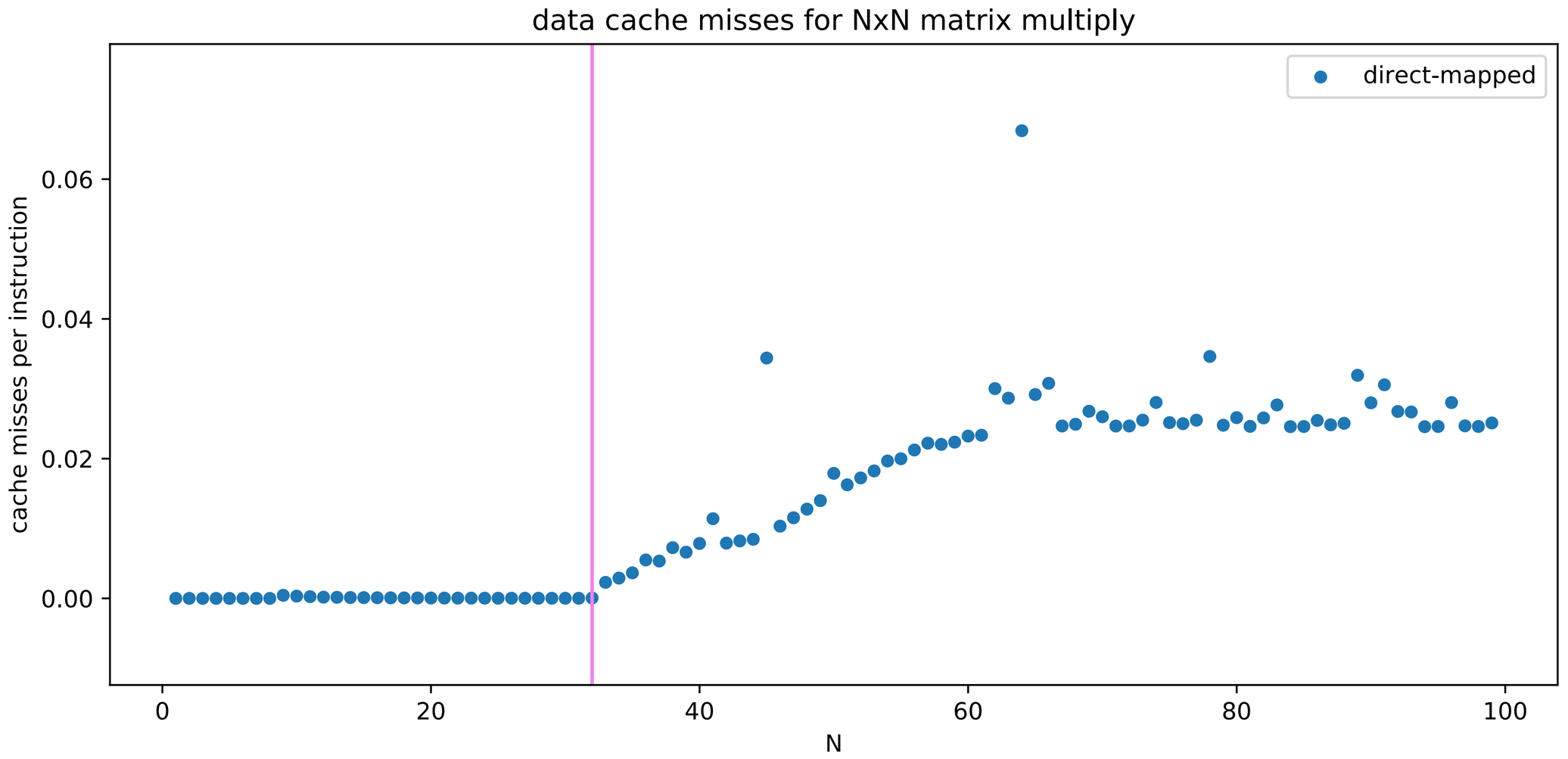
simulated 16KB direct-mapped cache; excluding BST setup

actual misses: BST lookups



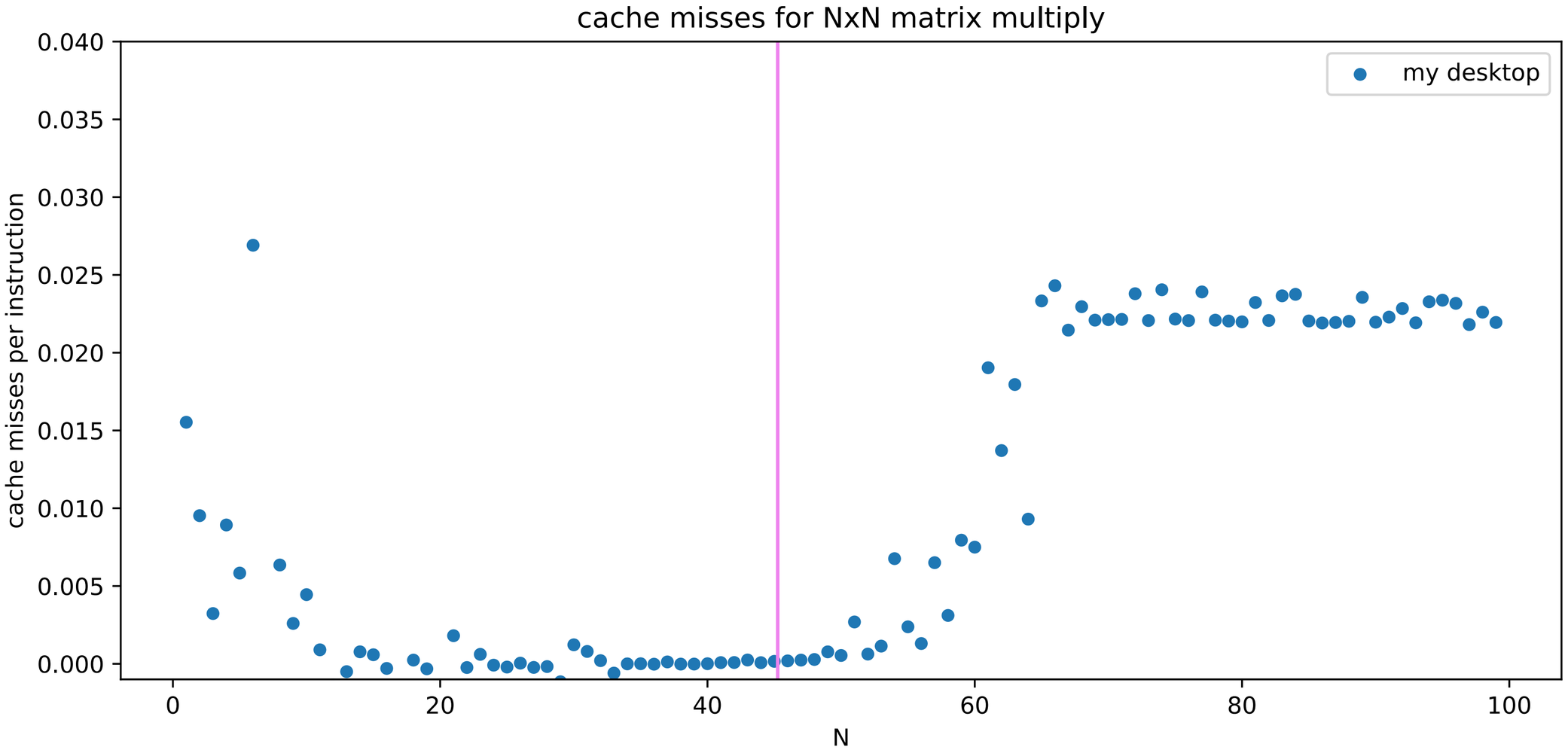
actual 32KB more complex cache (only one set of measurements + other things on the machine + excluding initial load)

simulated misses: matrix multiplies



simulated 16KB direct-mapped data cache; excluding initial load

actual misses: matrix multiplies



actual 32KB more complex cache; excluding initial matrix load

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 misses from two active values using same set
("conflict misses")

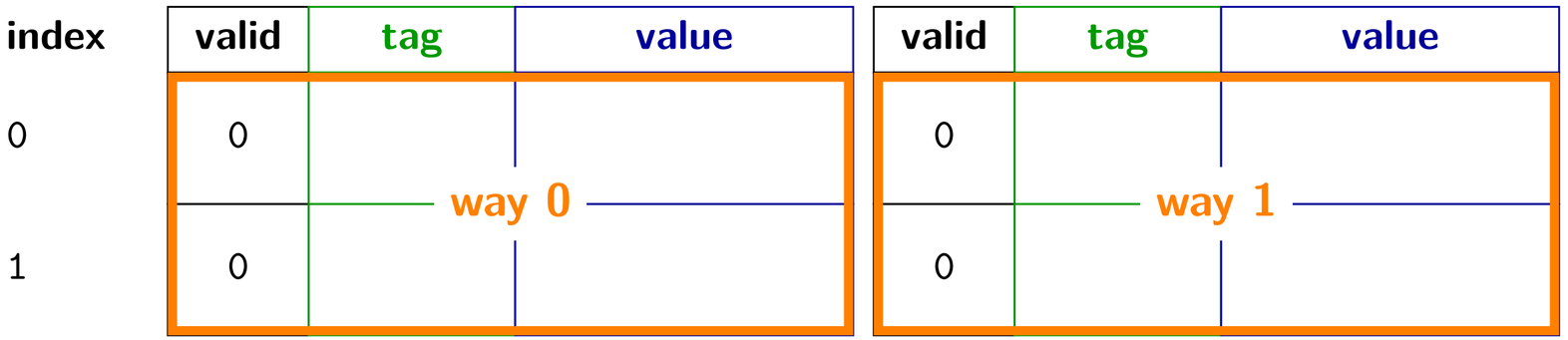
adding associativity

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)	

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	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 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)	

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

tag index offset

needs to replace block in set 0!

associative lookup possibilities

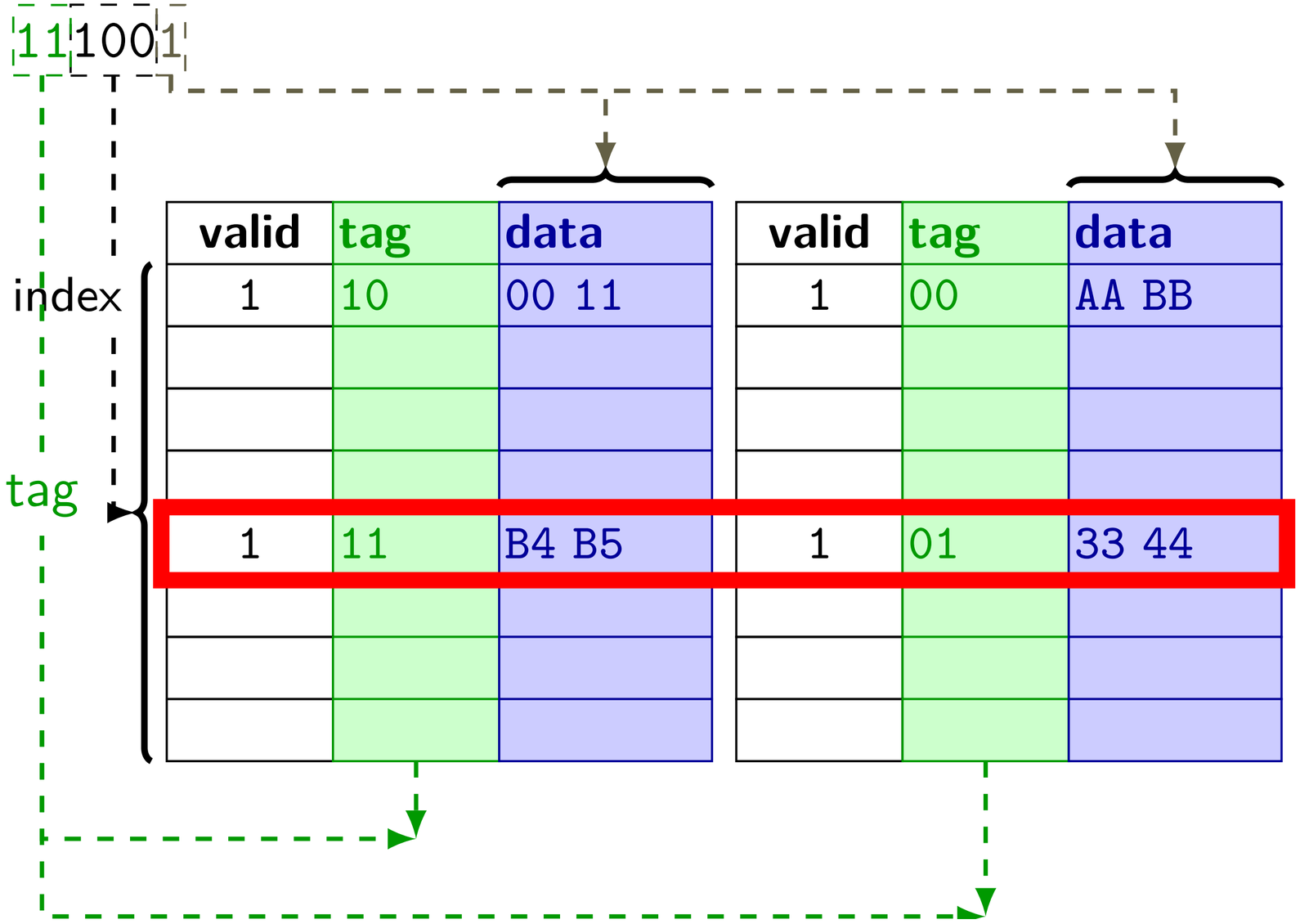
none of the blocks for the index are valid

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

cache operation (associative)



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

how to decide where to insert 0x64?

replacement policies

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

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

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

track which block was read least recently updated on *every access*

example replacement policies

least recently used

- take advantage of *temporal locality*
- at least $\lceil \log_2(E!) \rceil$ bits per set for E-way cache
  - (need to store order of all blocks)

approximations of least recently used

- implementing least recently used is expensive
- 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

LRU bit updating

LRU bit updating

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

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

address (hex)	result
...	...
01100001 (61)	miss
01100000 (60)	
01100000 (61)	
00000000 (00)	
11110000 (f0)	
11100000 (e0)	

LRU bit updating

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]*	*0*
1	1	011000	mem[0x62] mem[0x63]	0			1

address (hex)	result
...	...
01100001 (61)	miss
01100000 (60)	
01100000 (61)	
00000000 (00)	
11110000 (f0)	
11100000 (e0)	

LRU w/ more than two ways?

need to track total order

worst case: bunch of new accesses to same set

- first replaces least recently used
- next replaces next least recently used
- etc.

hard to track, so frequently only approximated

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)

C and cache misses (1)

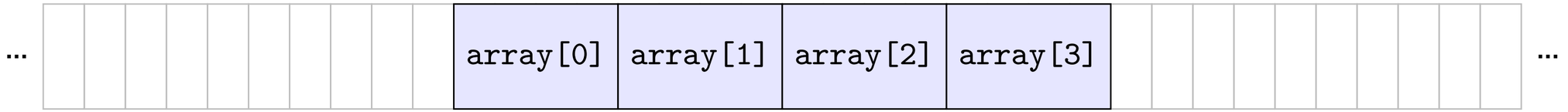
```
int array[4];
...
int even_sum = 0, odd_sum = 0;
even_sum += array[0];
odd_sum += array[1];
even_sum += array[2];
odd_sum += array[3];
```

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?

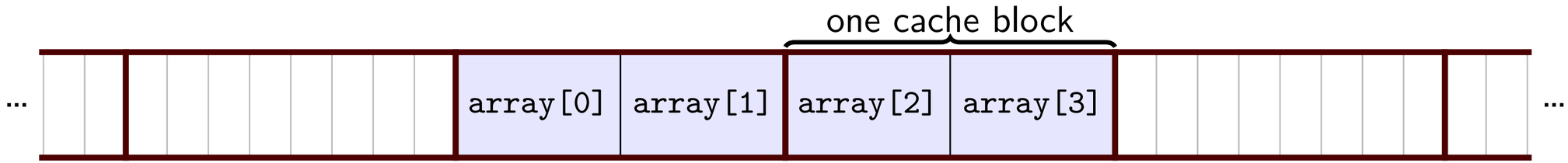
some possibilities

some possibilities



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

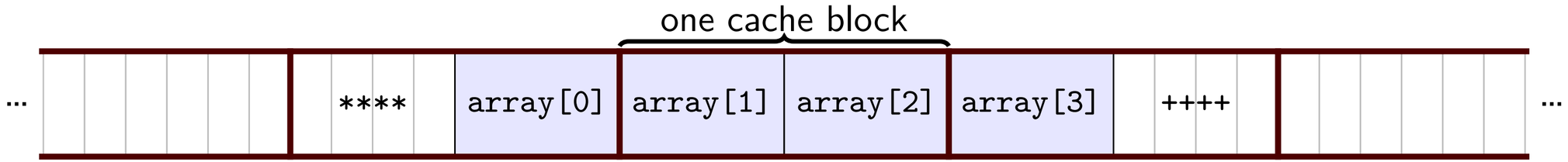
some possibilities



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]}`

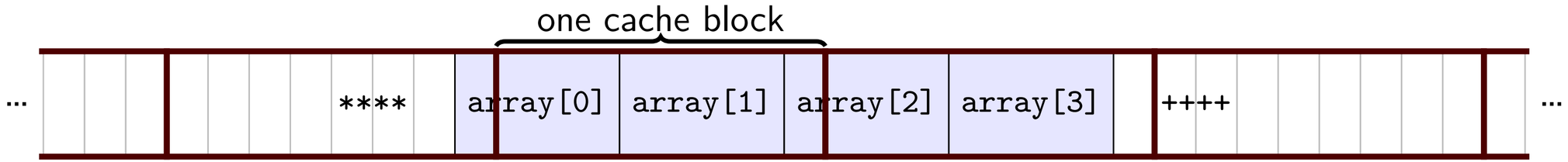
some possibilities



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 possibilities



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 to *align* values

align = make address be multiple of something

most important reason: don't cross cache block boundaries

C and cache misses (2)

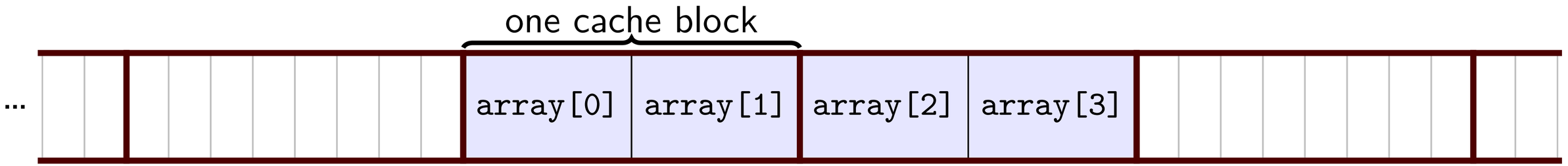
```
int array[4];
int even_sum = 0, odd_sum = 0;
even_sum += array[0];
even_sum += array[2];
odd_sum += array[1];
odd_sum += array[3];
```

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?

exercise solution



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

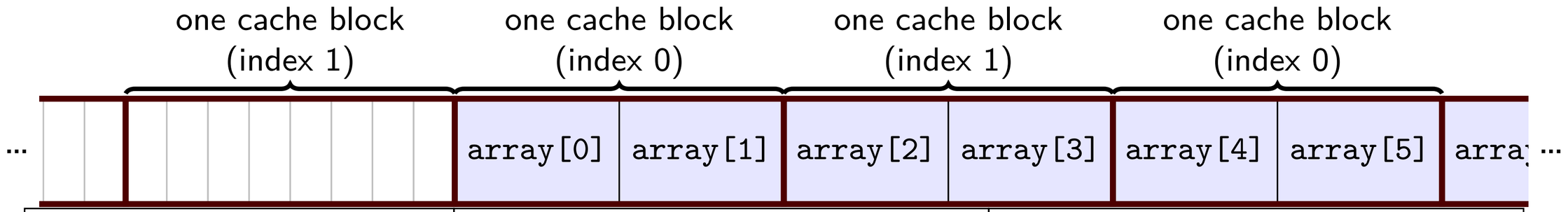
C and cache misses (3)

```
int array[8]; /* assume aligned */
...
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).

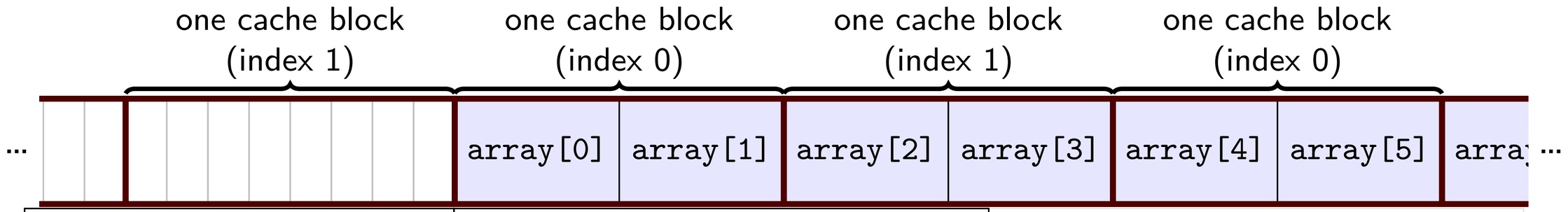
How many data cache misses on a 2-set direct-mapped

exercise solution



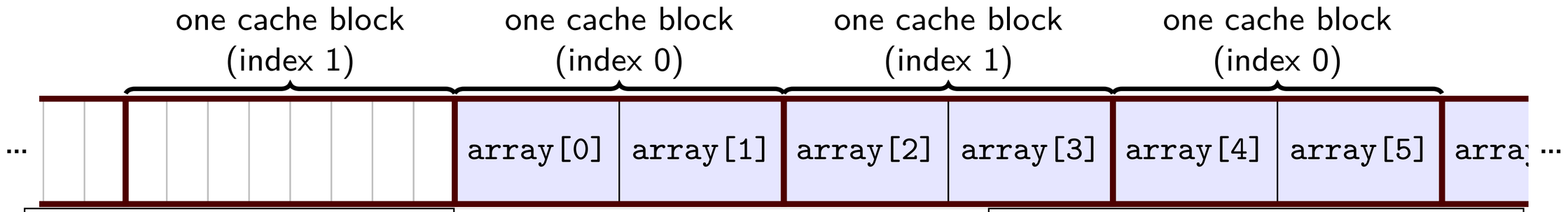
memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[4] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[6] (miss)	{array[4], array[5]}	{array[6], array[7]}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

C and cache misses (4)

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

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

C and cache misses (5)

```
int array[1024]; /* assume aligned */ int even = 0, odd = 0;
even += array[0];
even += array[2];
even += array[512];
even += array[514];
odd += array[1];
odd += array[3];
odd += array[511];
odd += array[513];
```

Assume everything but `array` is kept in registers (and the compiler does not do anything funny).
observation: `array[0]` and `array[512]` exactly 2KB apart

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

C and cache misses (6)

```
int array[1024]; /* assume aligned */ int even = 0, odd = 0;
even += array[0];
even += array[2];
even += array[500];
even += array[502];
odd += array[1];
odd += array[3];
odd += array[501];
odd += array[503];
```

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 blocks?

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:

= distance from `array1` to `array2` not multiple of `#sets` × `bytes/set`

2 misses every 4 `i`

blocks of 4 `array1[X]` values loaded, then used 4 times before loading next block

(and same for `array2[X]`)

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

= distance from `array1` to `array2` is multiple of `#sets` × `bytes/set`

2 misses every `i`

block of 4 `array1[X]` values loaded, one value used from it,

then, block of 4 `array2[X]` values replaces it, one value used from it, ...

worst case in practice?

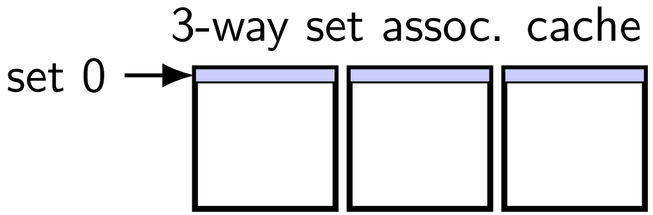
can have worst spacing happen by accident:

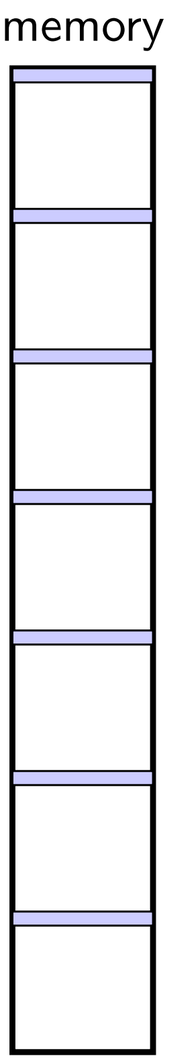
two structs/arrays placed at start of newly allocated page

- worst-case spacing likely small multiple of page size

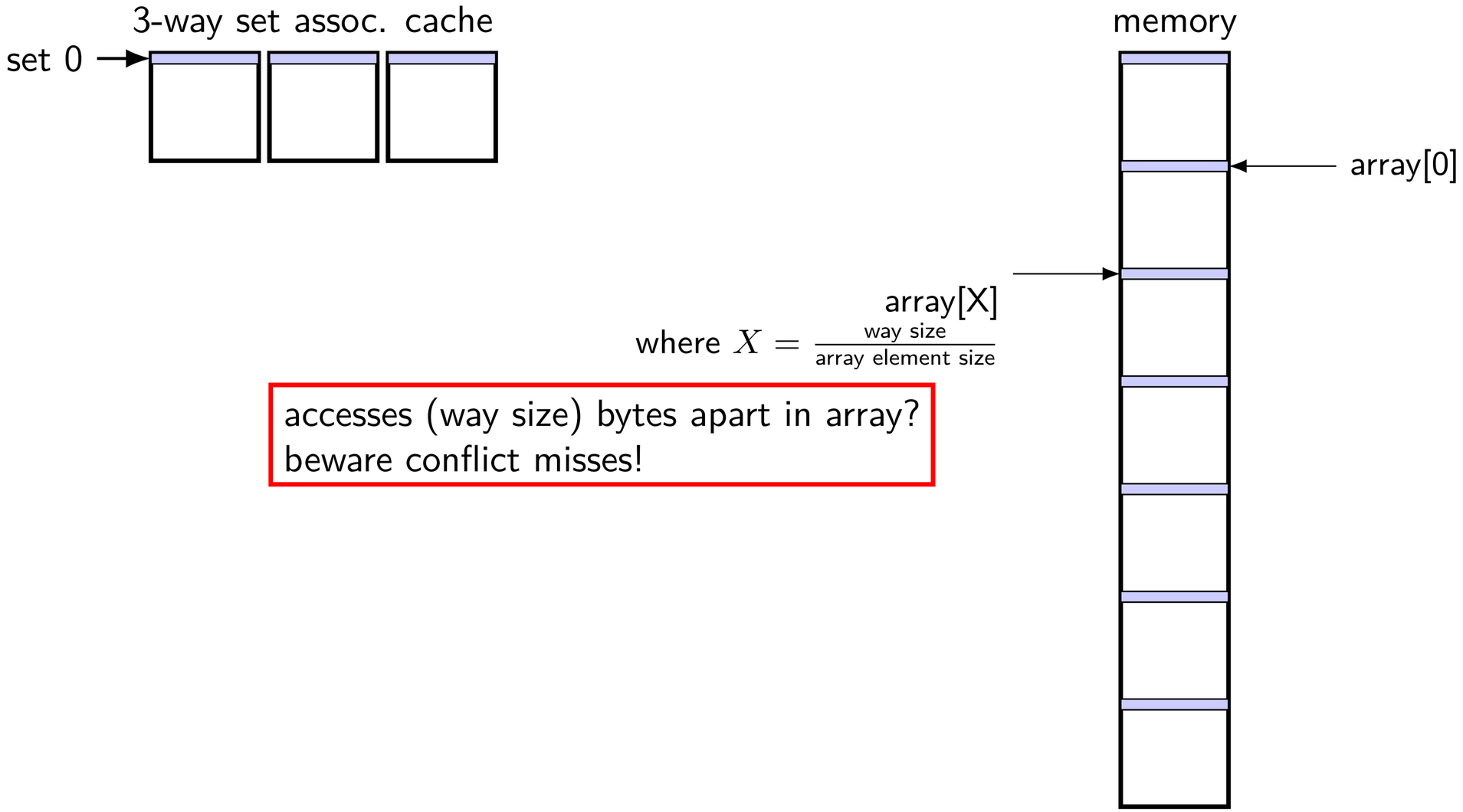
columns of matrix with power-of-two width

mapping of sets to memory (3-way)





mapping of sets to memory (3-way)



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

C and cache misses (assoc)

```
int array[1024]; /* assume aligned */
int even_sum = 0, odd_sum = 0;
even_sum += array[0];
even_sum += array[2];
even_sum += array[512];
even_sum += array[514];
odd_sum += array[1];
odd_sum += array[3];
odd_sum += array[511];
odd_sum += array[513];
```

Assume everything but `array` is kept in registers (and the compiler does not do anything funny).
observation: `array[0]`, `array[256]`, `array[512]`, `array[768]` in same set

How many data cache misses on a 2KB 2-way set associative cache with 16B blocks?

C and cache misses (assoc)

```
int array[1024]; /* assume aligned */
int even_sum = 0, odd_sum = 0;
even_sum += array[0];
even_sum += array[256];
even_sum += array[512];
even_sum += array[768];
odd_sum += array[1];
odd_sum += array[257];
odd_sum += array[513];
odd_sum += array[769];
```

Assume everything but `array` is kept in registers (and the compiler does not do anything funny).
observation: `array[0]`, `array[256]`, `array[512]`, `array[768]` in same set

How many data cache misses on a 2KB 2-way set associative cache with 16B blocks?

handling writes

what about writing to the cache?

two decision points:

if the value is not in cache, do we add it?

- if yes: need to load rest of block — *write-allocate*
- if no: missing out on locality? *write-no-allocate*

if value is in cache, when do we update next level?

- if immediately: extra writing *write-through*
- if later: need to remember to do so *write-back*

allocate on write?

say processor writes *less than whole* cache block

and 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?

allocate on write?

say processor writes *less than whole* cache block

and 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

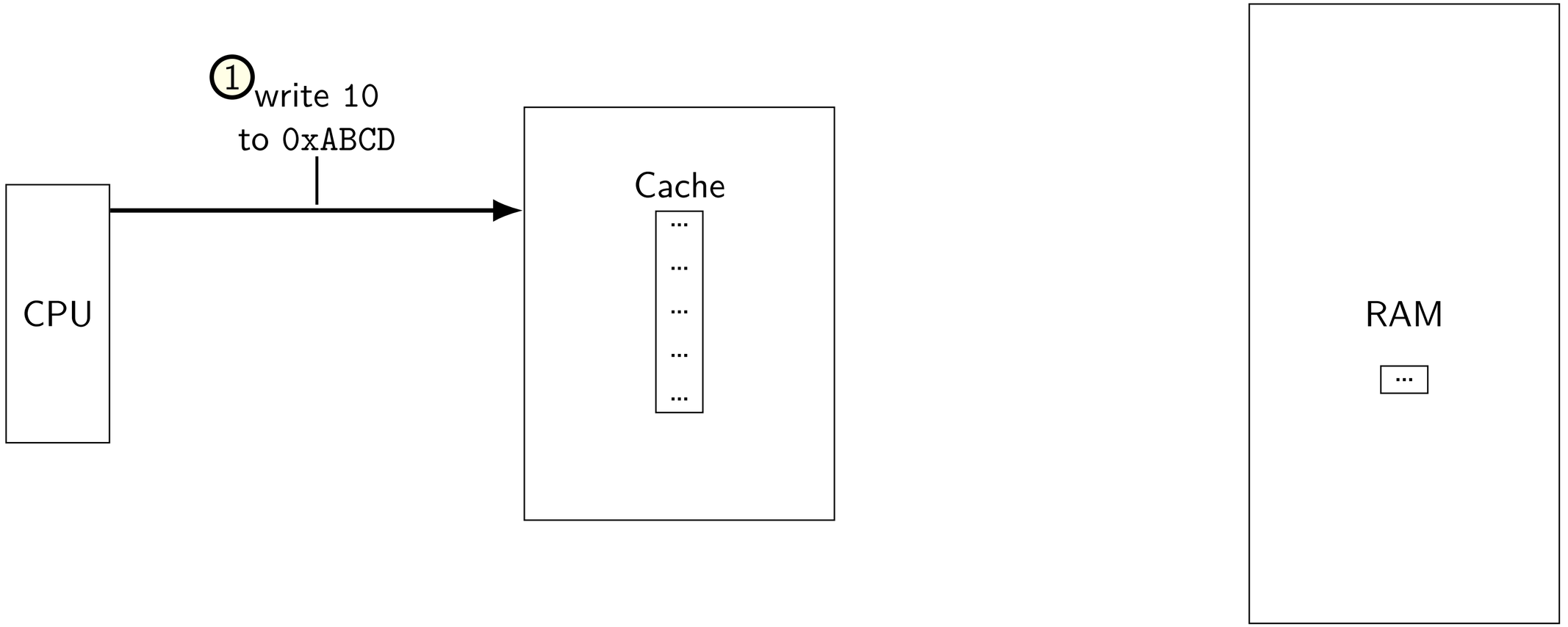
don't use cache at all (send write to memory *instead*)

guess: not read soon?

write-allocate v. write-no-allocate

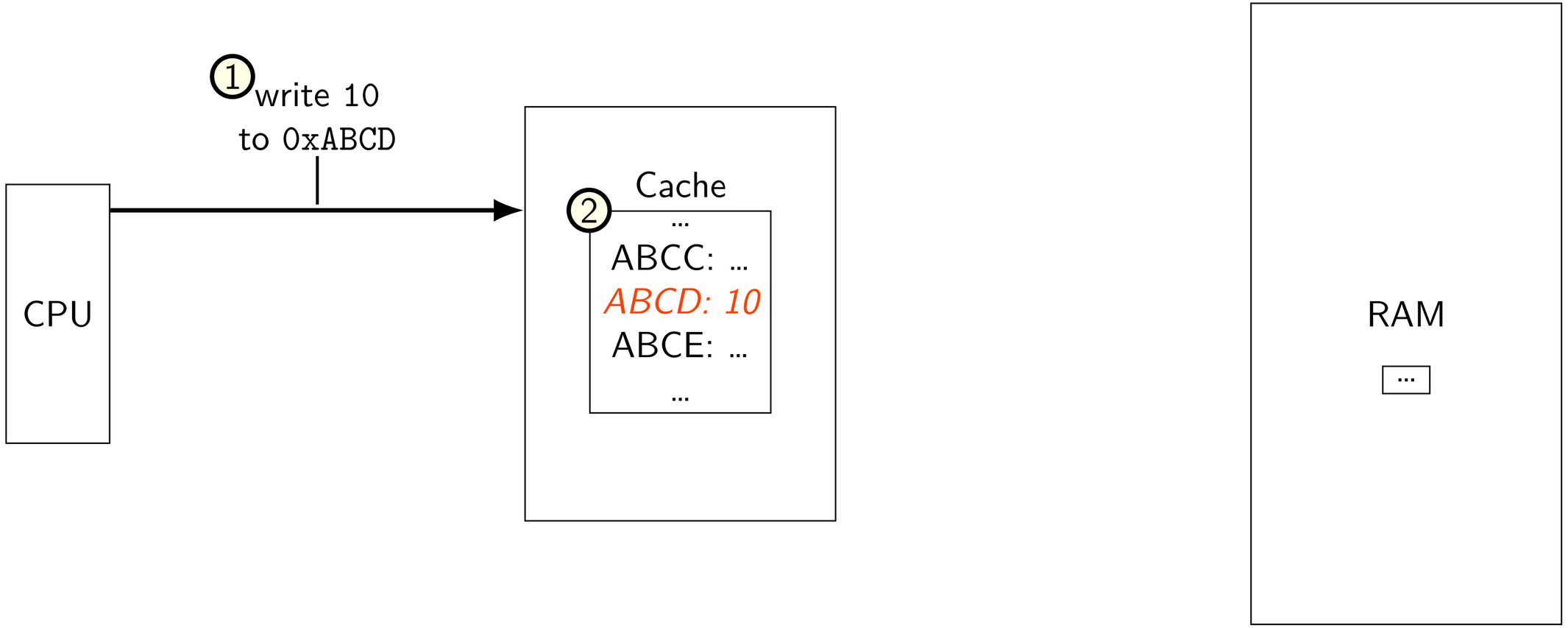
write-allocate v. write-no-allocate

option 1: write-allocate



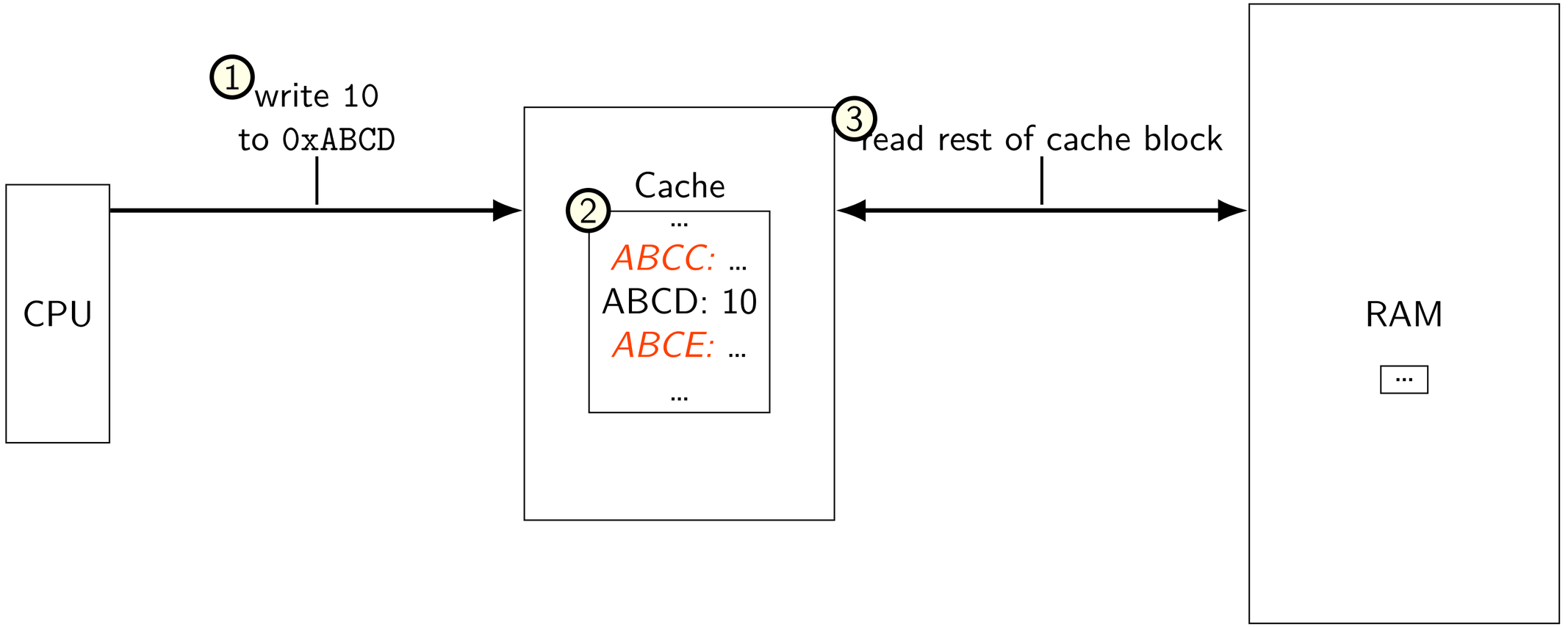
write-allocate v. write-no-allocate

option 1: write-allocate



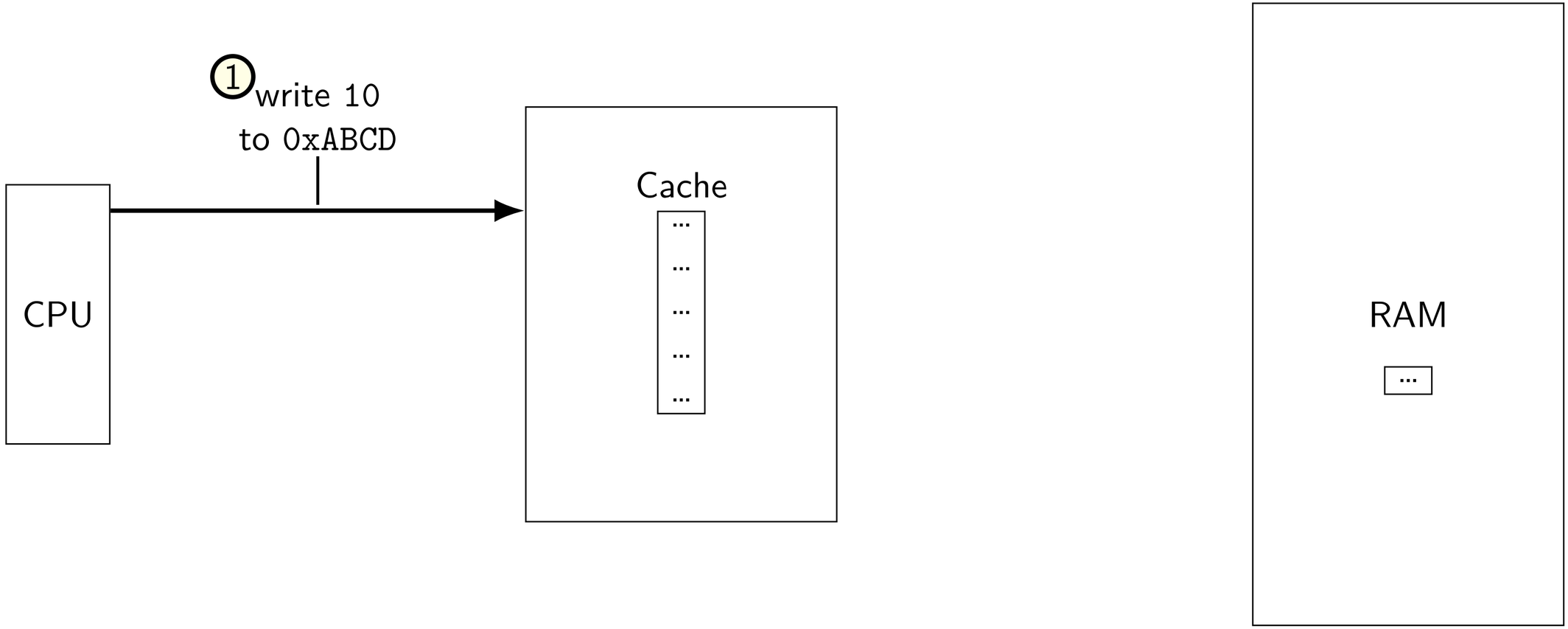
write-allocate v. write-no-allocate

option 1: write-allocate



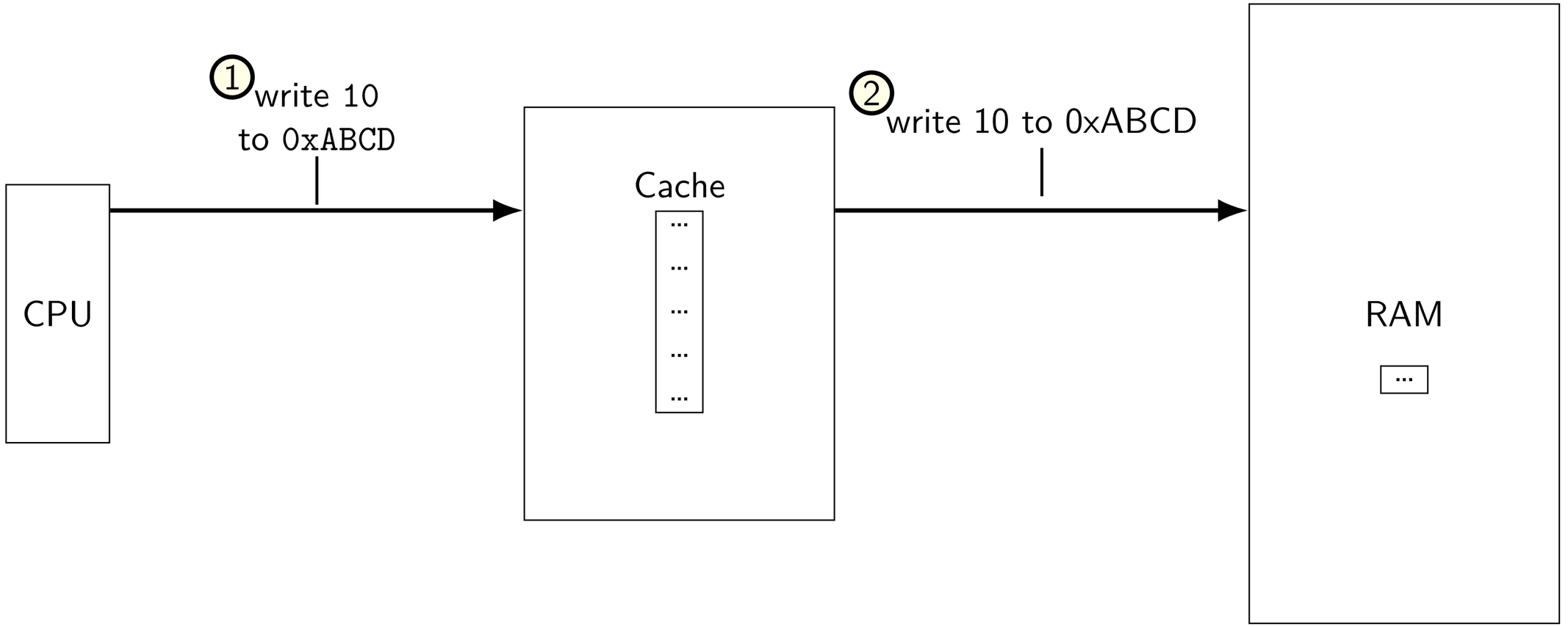
write-allocate v. write-no-allocate

option 2: write-no-allocate



write-allocate v. write-no-allocate

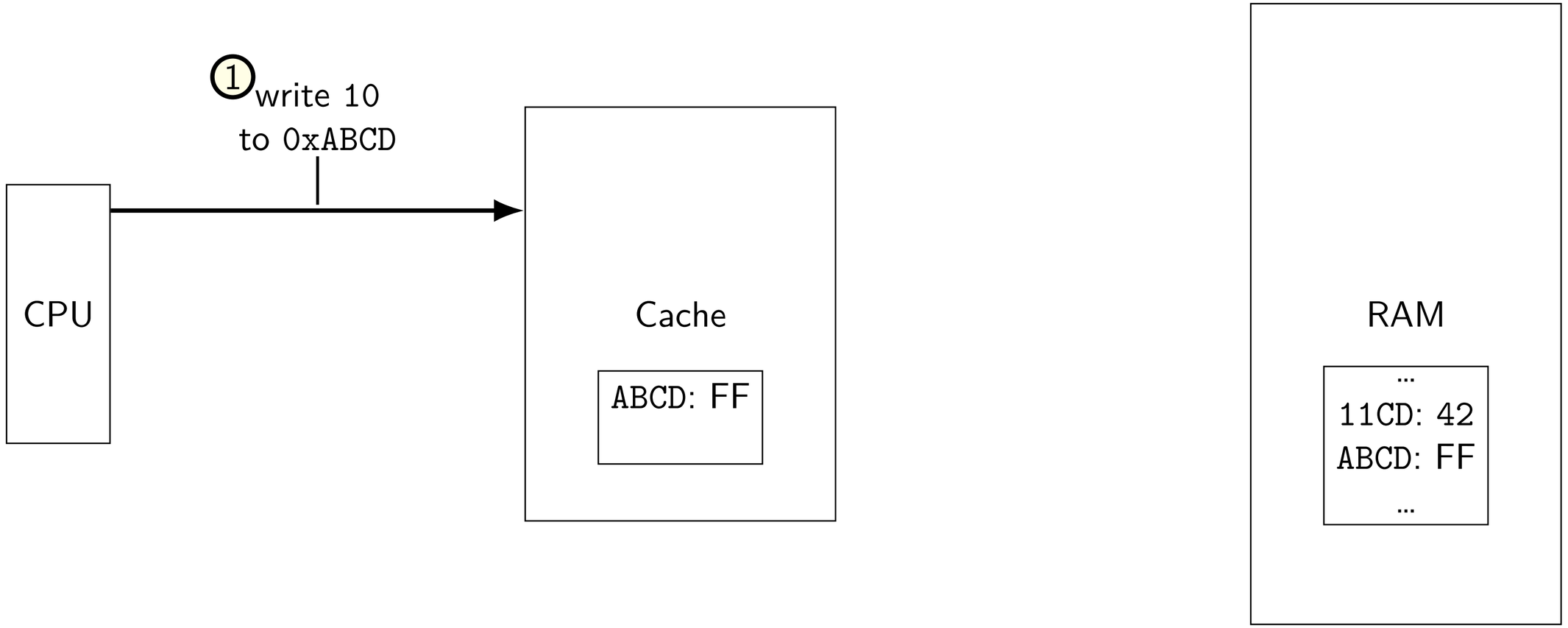
option 2: write-no-allocate



write-through v. write-back

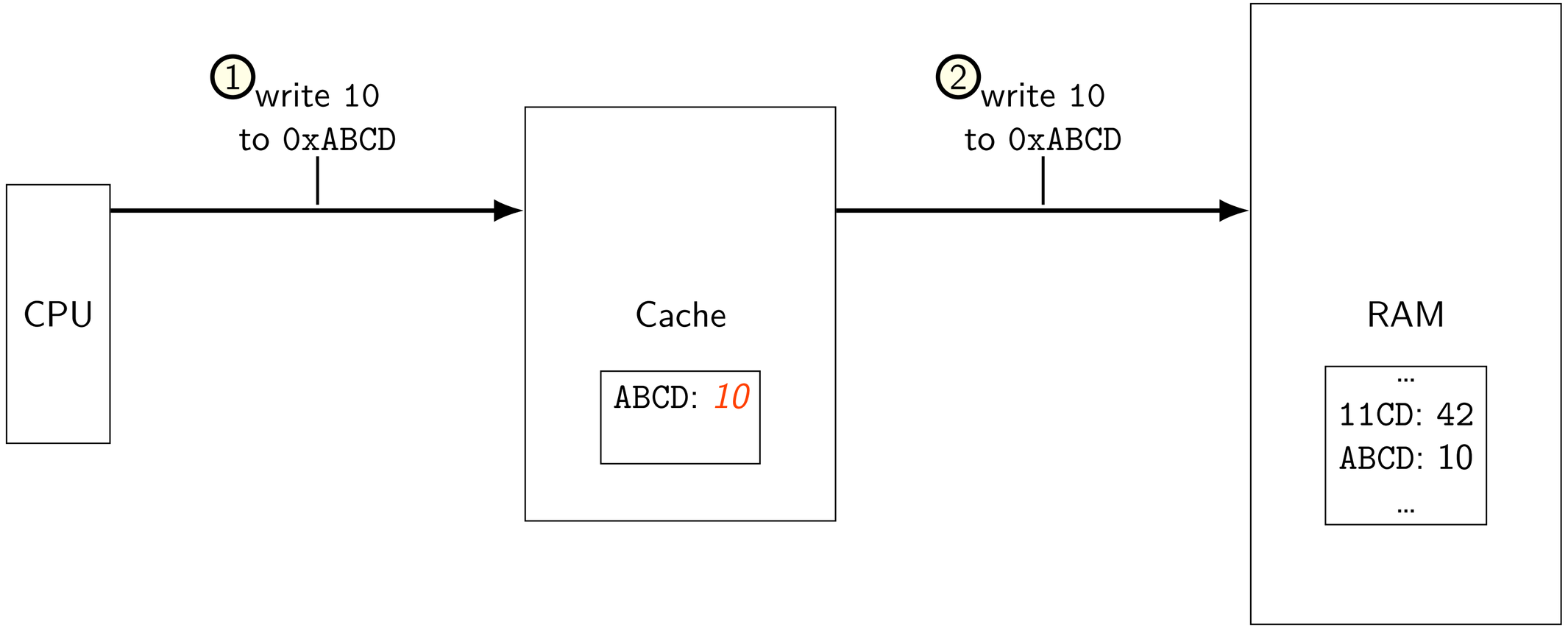
write-through v. write-back

option 1: write-through



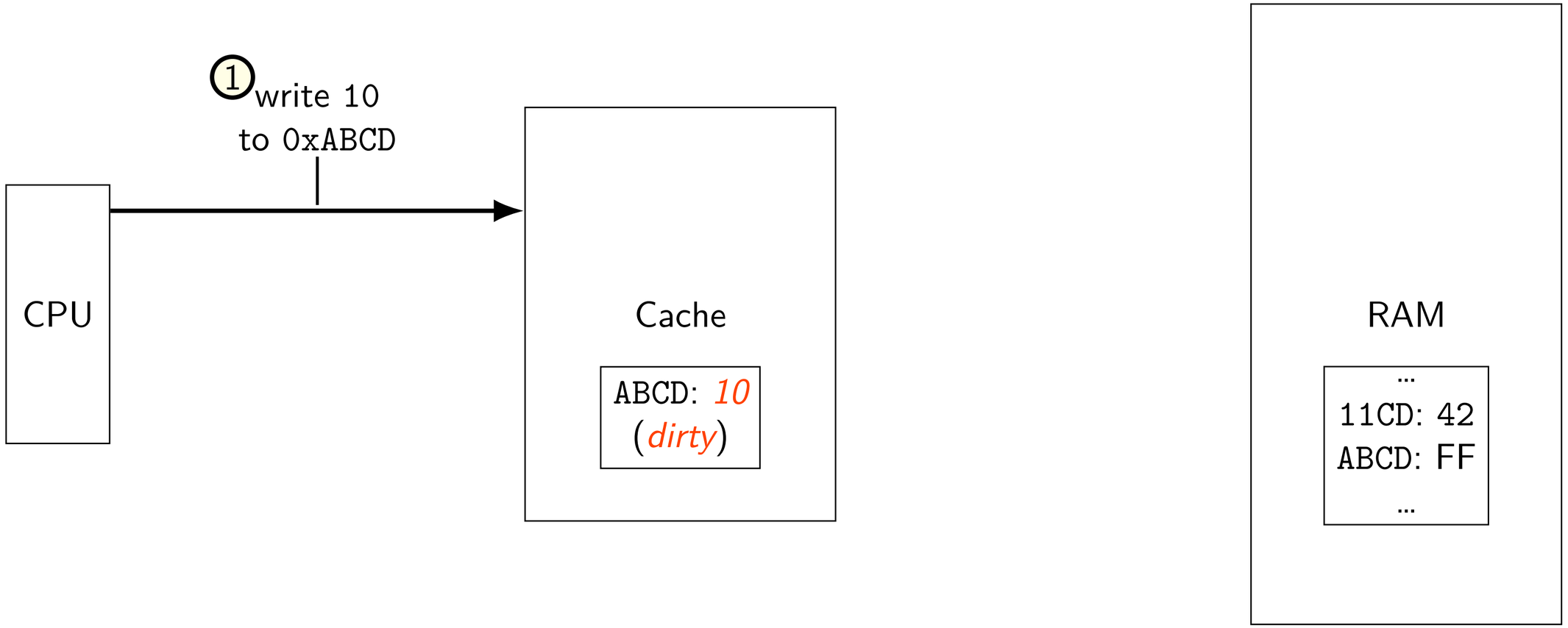
write-through v. write-back

option 1: write-through



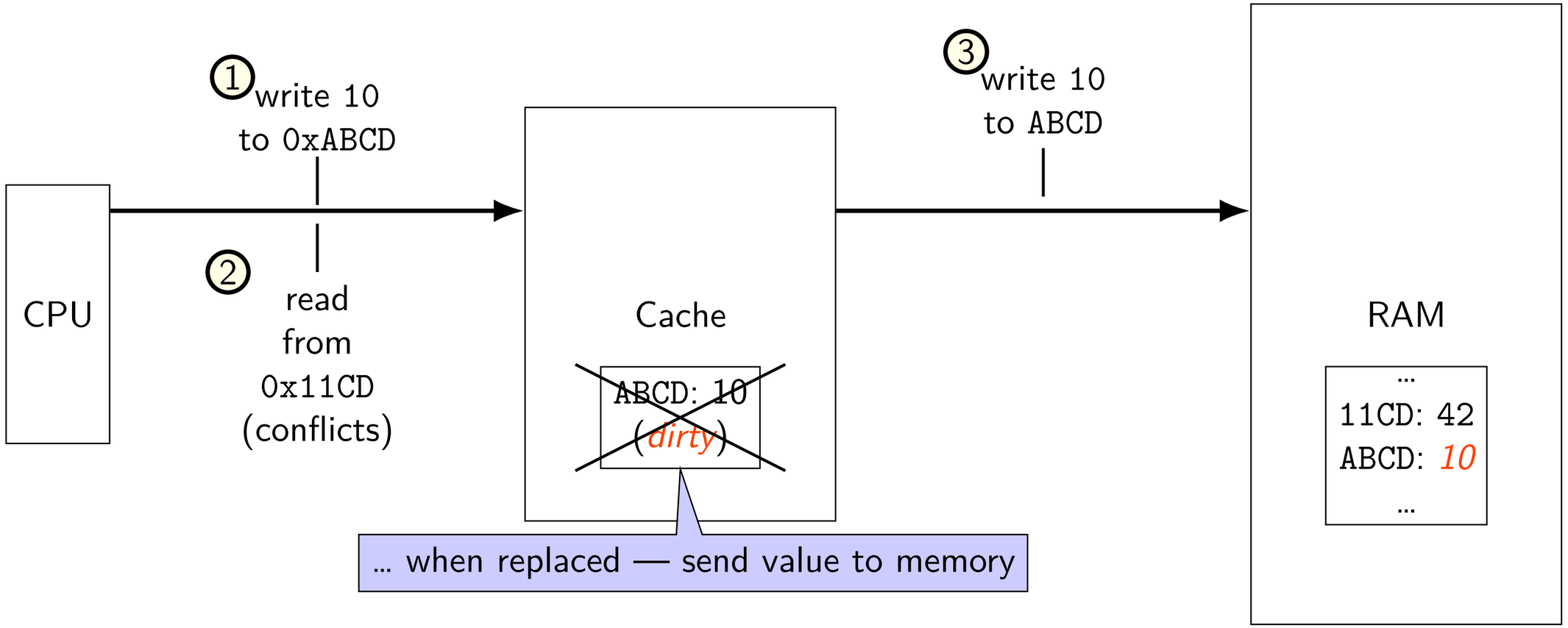
write-through v. write-back

option 2: write-back



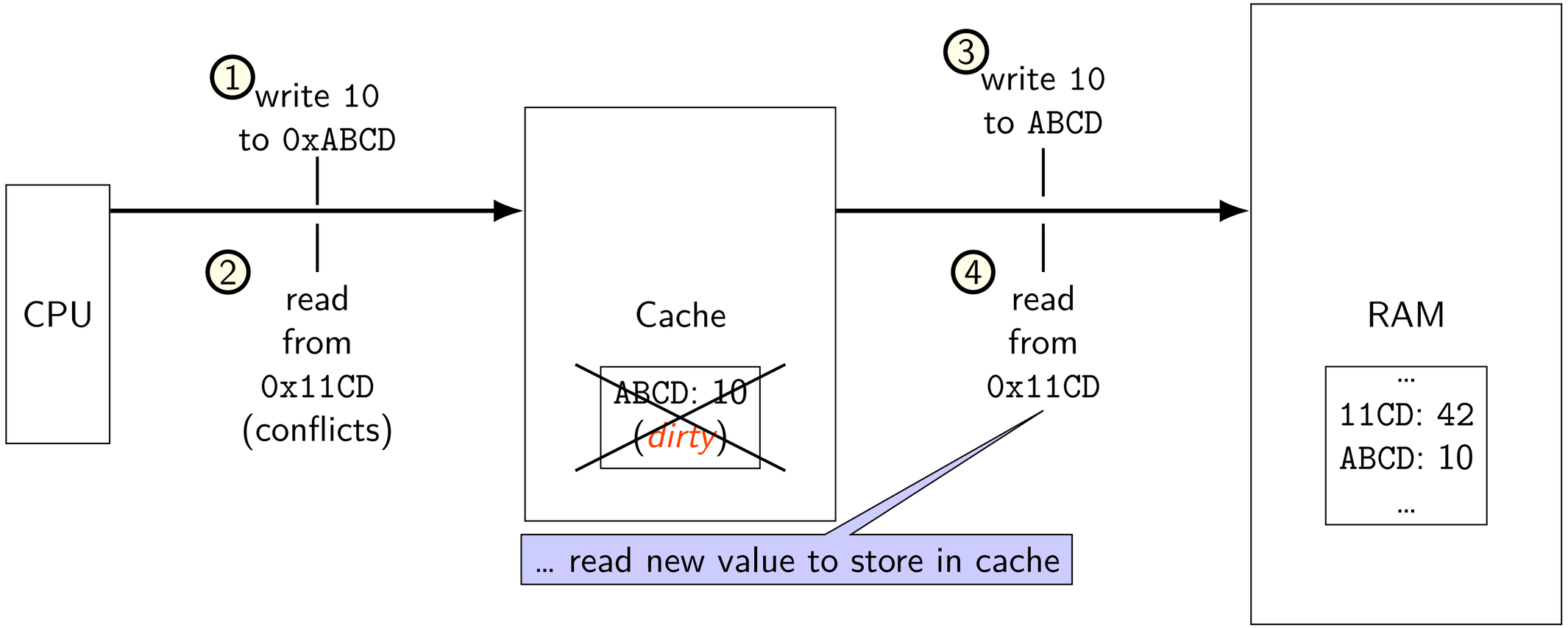
write-through v. write-back

option 2: write-back



write-through v. write-back

option 2: write-back



write-back policy

changed value!

2-way set associative, 2 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

write-allocate + write-back

write-allocate + write-back

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 + write-back

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 + write-back

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 + write-back

2-way set associative, LRU, writeback

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

writing `0xFF` into address `0x04`?

index 0, tag 000001

step 1: find *least recently used* block

step 2: possibly writeback old block

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

step 3b: update LRU information

write-no-allocate + write-back

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*

cache write 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

cache write 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:

reading 1 byte from 0x52:

reading 1 byte from 0x50:

cache write 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:

reading 1 byte from 0x52:

reading 1 byte from 0x50:

cache write 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~~0

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

reading 1 byte from 0x52:

reading 1 byte from 0x50:

cache write 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 next-level read or write

reading 1 byte from 0x52:

reading 1 byte from 0x50:

cache write 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	101000	mem[0x52] mem[0x53]	~~1~~0	~~1~~0

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

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

reading 1 byte from 0x50:

cache write 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 next-level read or write

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

reading 1 byte from 0x50:

cache write exercise (1, solution)

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

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	101000	mem[0x50] mem[0x51]	0	1	010000	mem[0x40]* mem[0x41]*	1	~~0~~1
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 next-level 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

cache write 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

cache write 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:

reading 1 byte from 0x52:

reading 1 byte from 0x50:

cache write 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:

reading 1 byte from 0x52:

reading 1 byte from 0x50:

cache write 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:

reading 1 byte from 0x50:

cache write 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	101000	mem[0x52] mem[0x53]	~~1~~0

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:

cache write 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:

cache write exercise (2, solution)

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

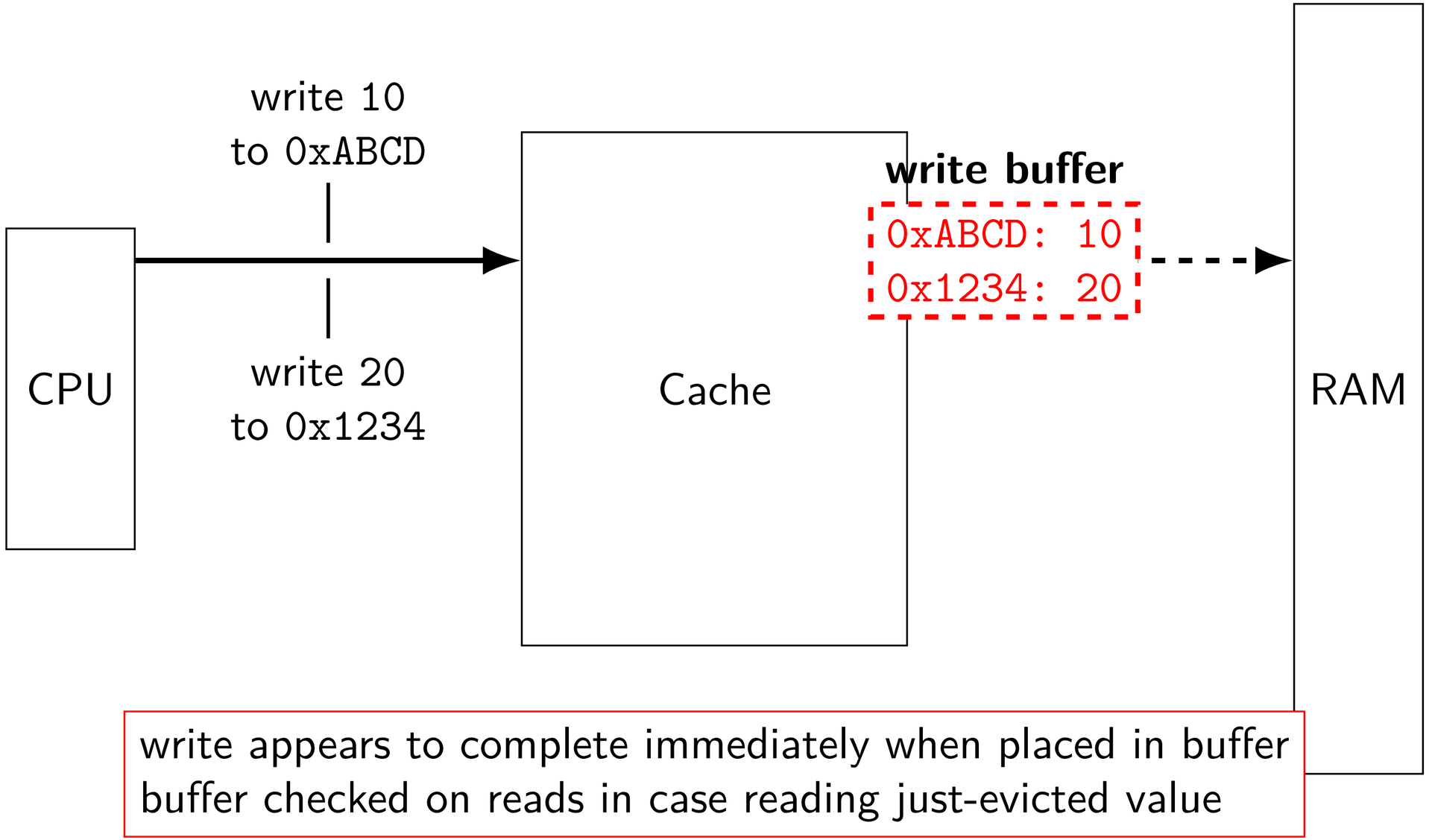
index	valid	tag	value	valid	tag	value	LRU
0	1	101000	mem[0x50] mem[0x51]	1	010000	mem[0x40] mem[0x41]	~~0~~1
1	1	011000	mem[0x62] mem[0x63]	1	001100	mem[0x32] mem[0x33]	1

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 with write-through caches



write appears to complete immediately when placed in buffer
buffer checked on reads in case reading just-evicted value

cache tradeoffs briefly

deciding cache size, associativity, etc.?

lots of tradeoffs:

- more cache hits v. slower cache hits?
- faster cache hits v. fewer cache hits?
- (N+1)th-level cache v. larger Nth level cache?
- …

details depend on programs run

- how often is same block used again?
- how often is same index bits used?
- how much {temporal,spatial} locality to take advantage of?

simulation to assess impact of designs

cache organization and miss rate

depends on program; one example:

SPEC CPU2000 benchmarks, 64B block size, LRU replacement

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%

average memory access time

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

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

effective speed of memory

AMAT exercise (1)

90% cache hit rate

hit time is 2 cycles

30 cycle miss penalty

what is the average memory access time?

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

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

AMAT exercise (1)

90% cache hit rate

hit time is 2 cycles

30 cycle miss penalty

what is the average memory access time?

5 cycles

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

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

AMAT exercise (1)

90% cache hit rate

hit time is 2 cycles

30 cycle miss penalty

what is the average memory access time?

5 cycles

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

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

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

two-level page table lookup

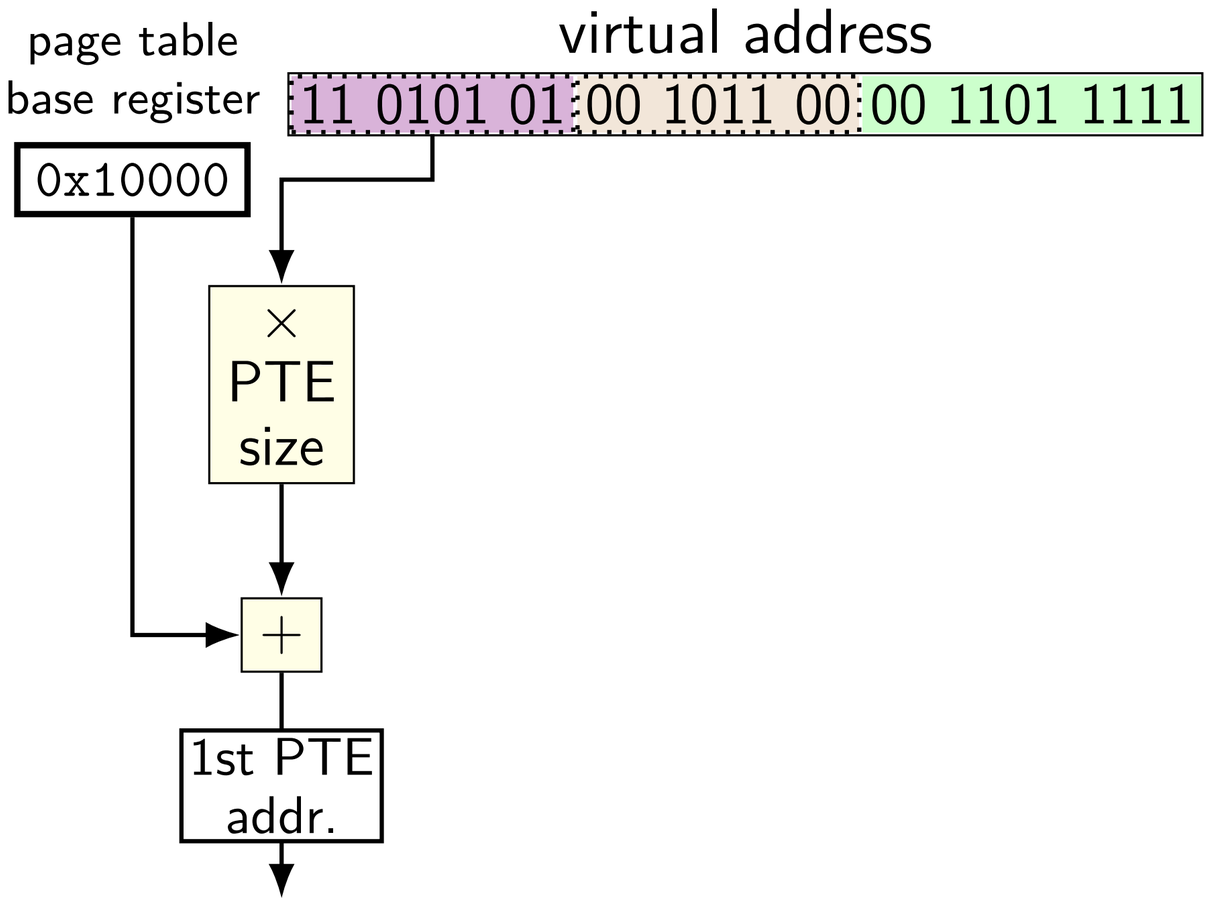
two-level page table lookup

virtual address

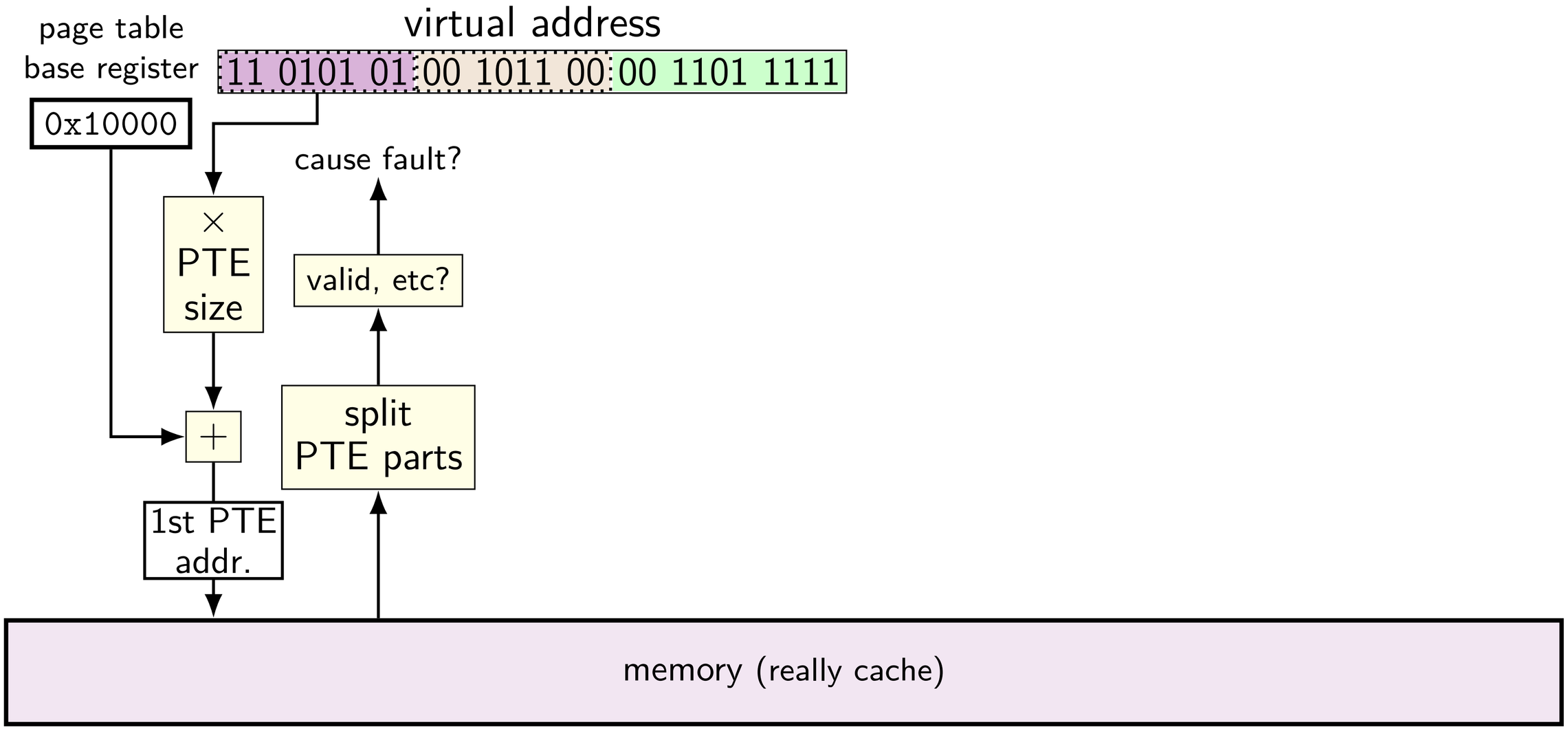
VPN — split into two parts (one per level)

this example: parts equal sized — common, but not required

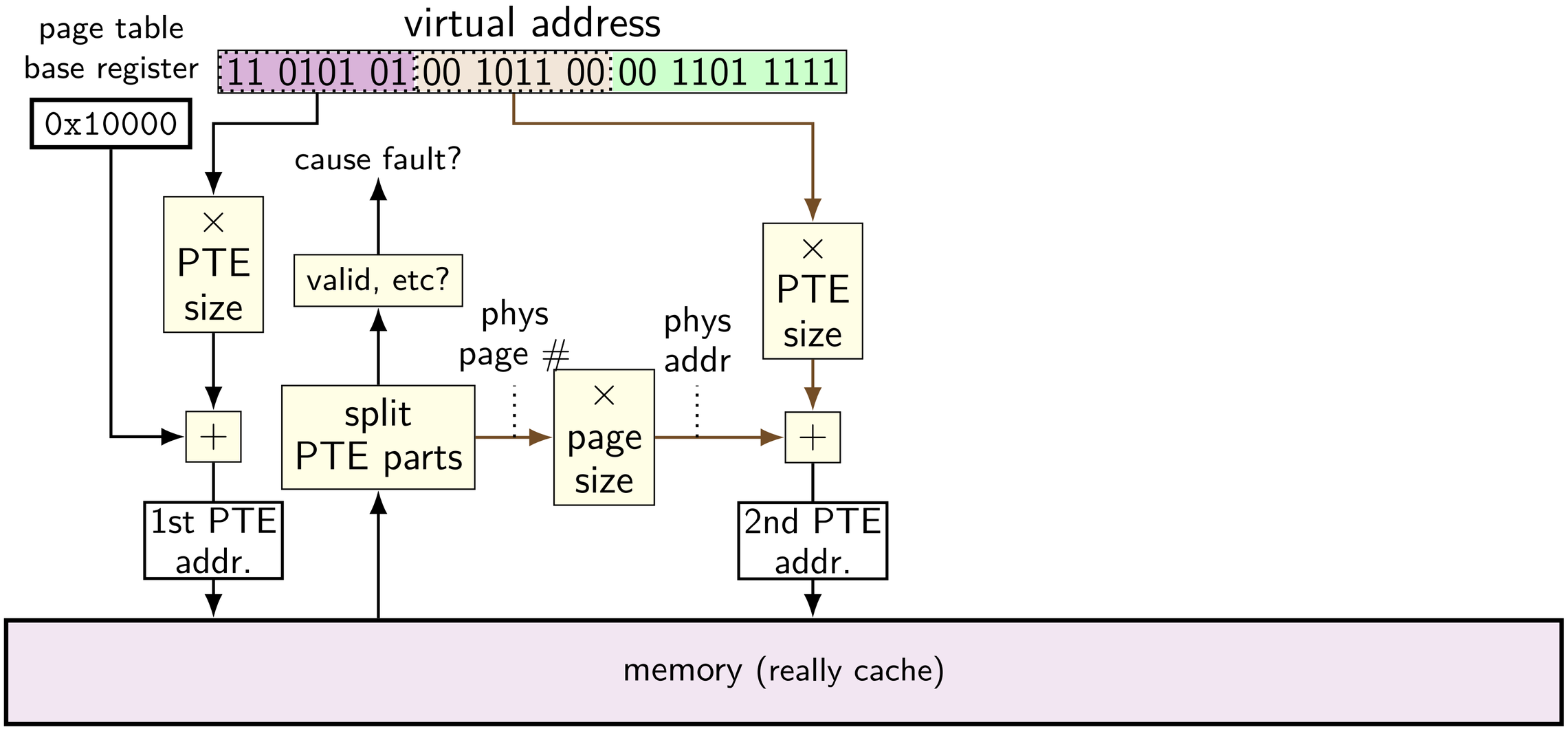
two-level page table lookup



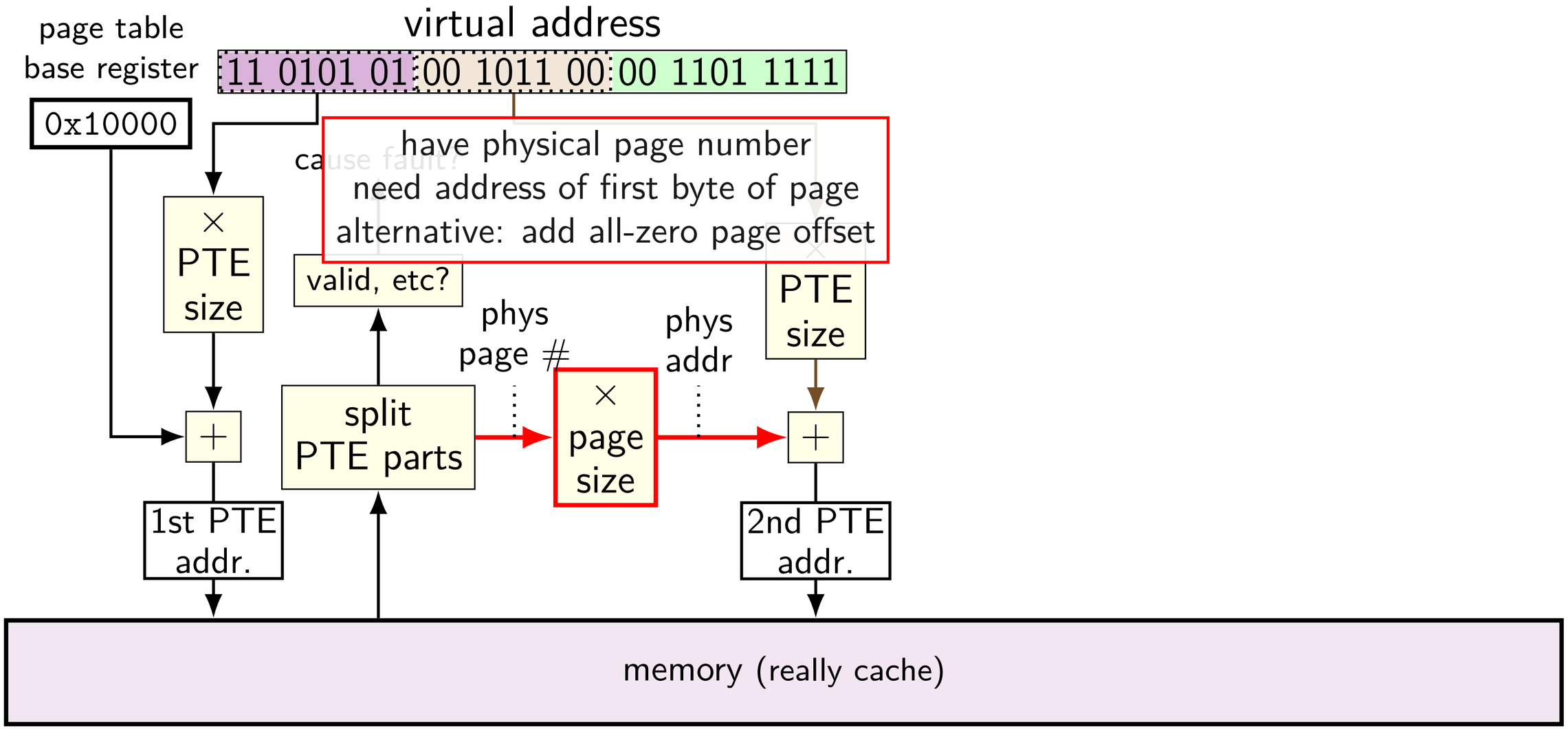
two-level page table lookup



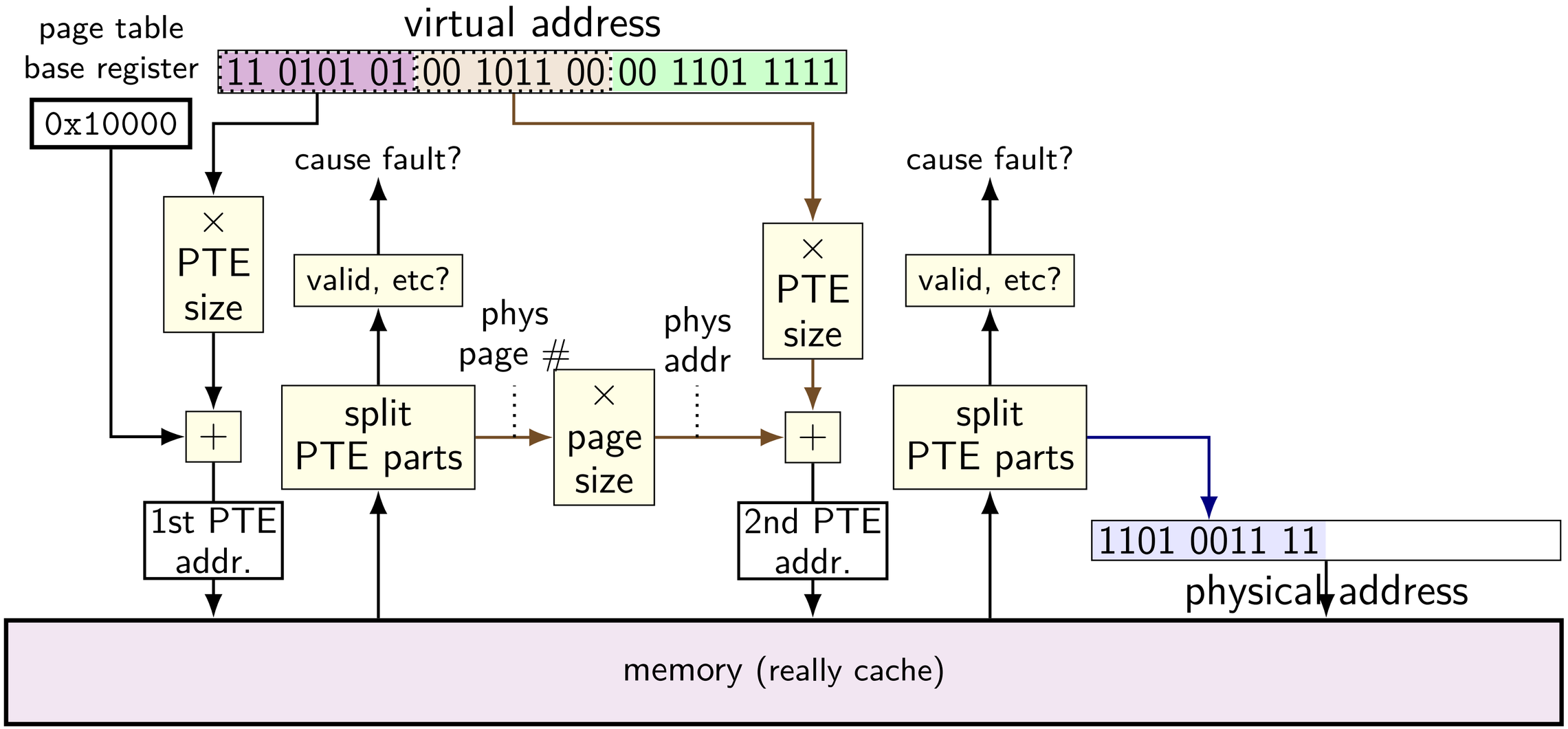
two-level page table lookup



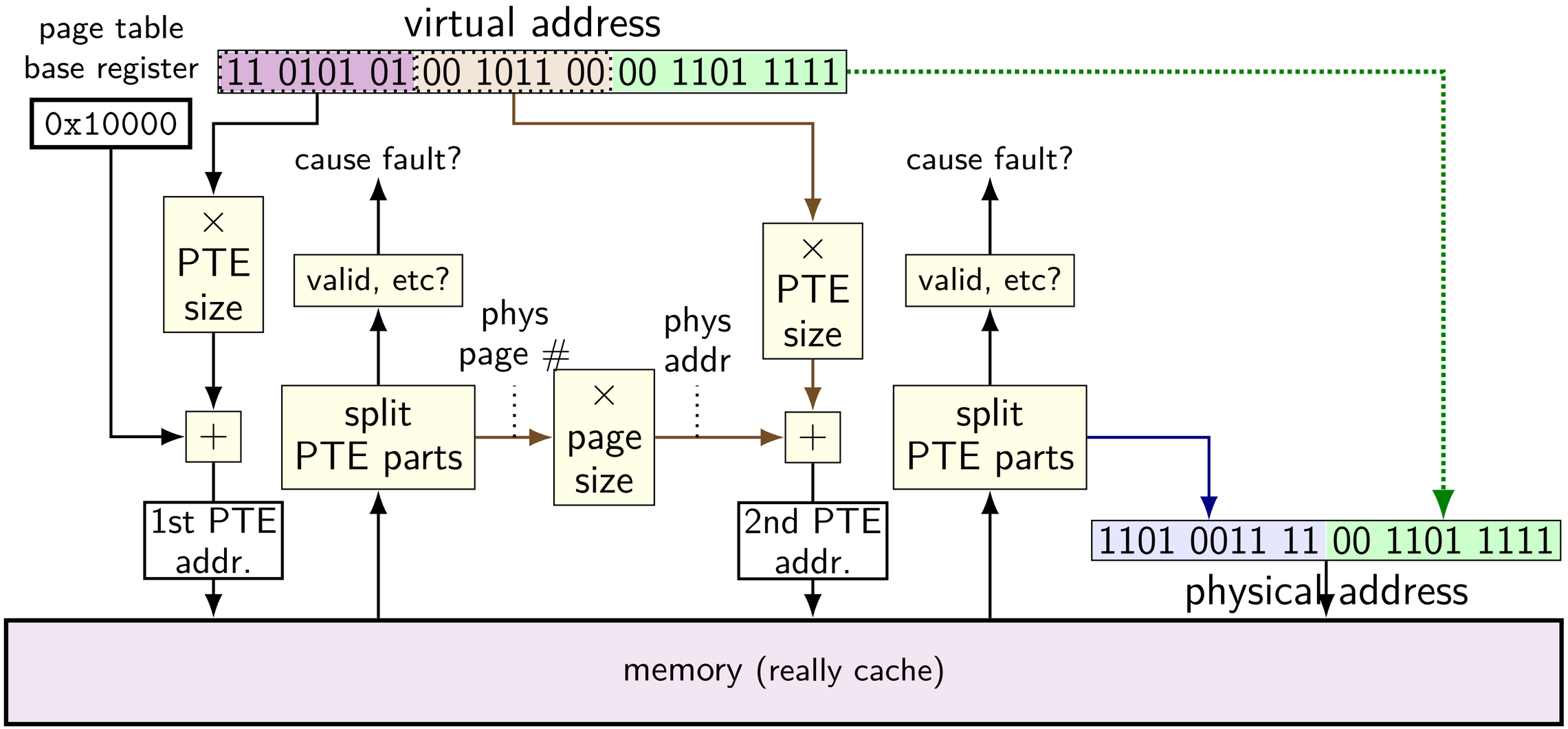
two-level page table lookup



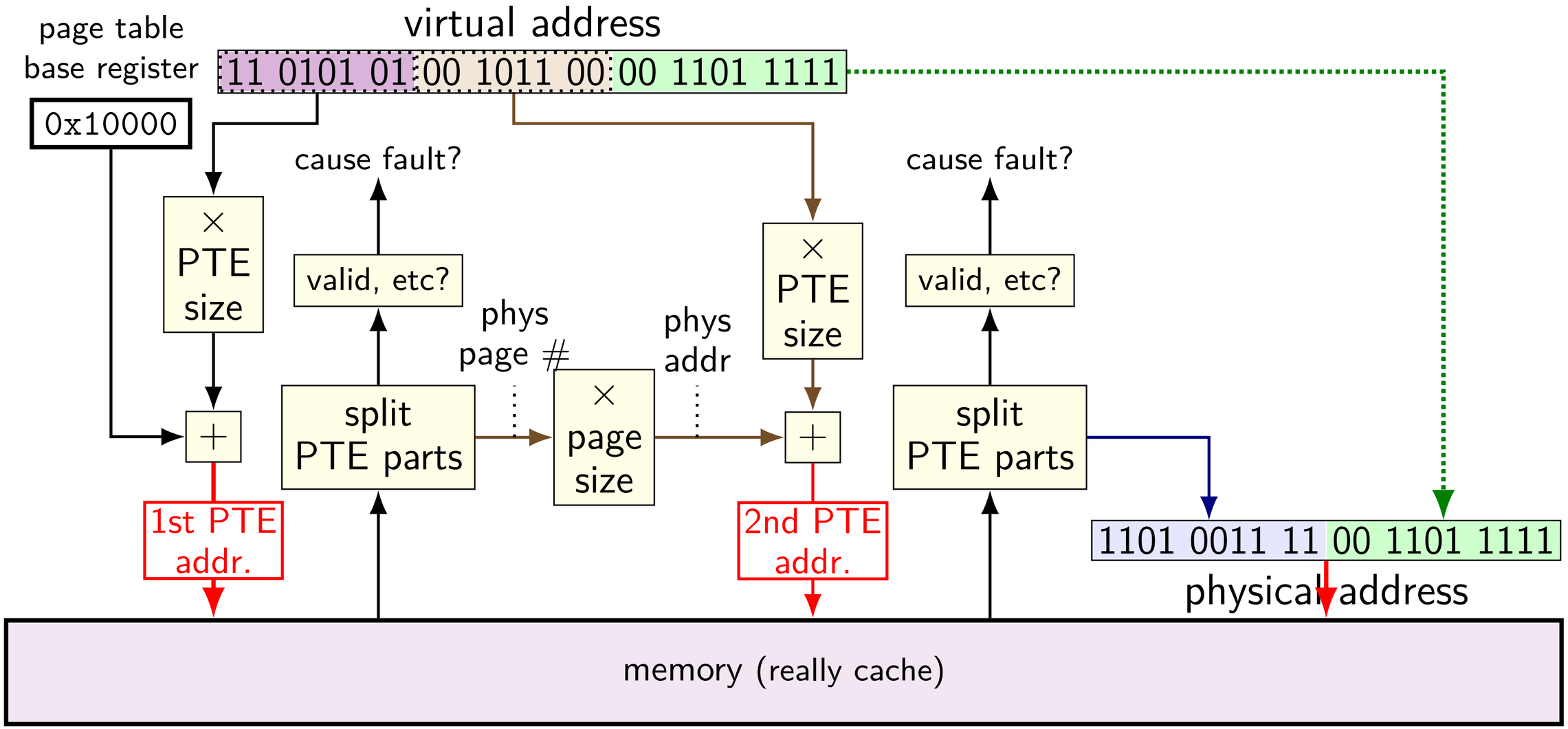
two-level page table lookup



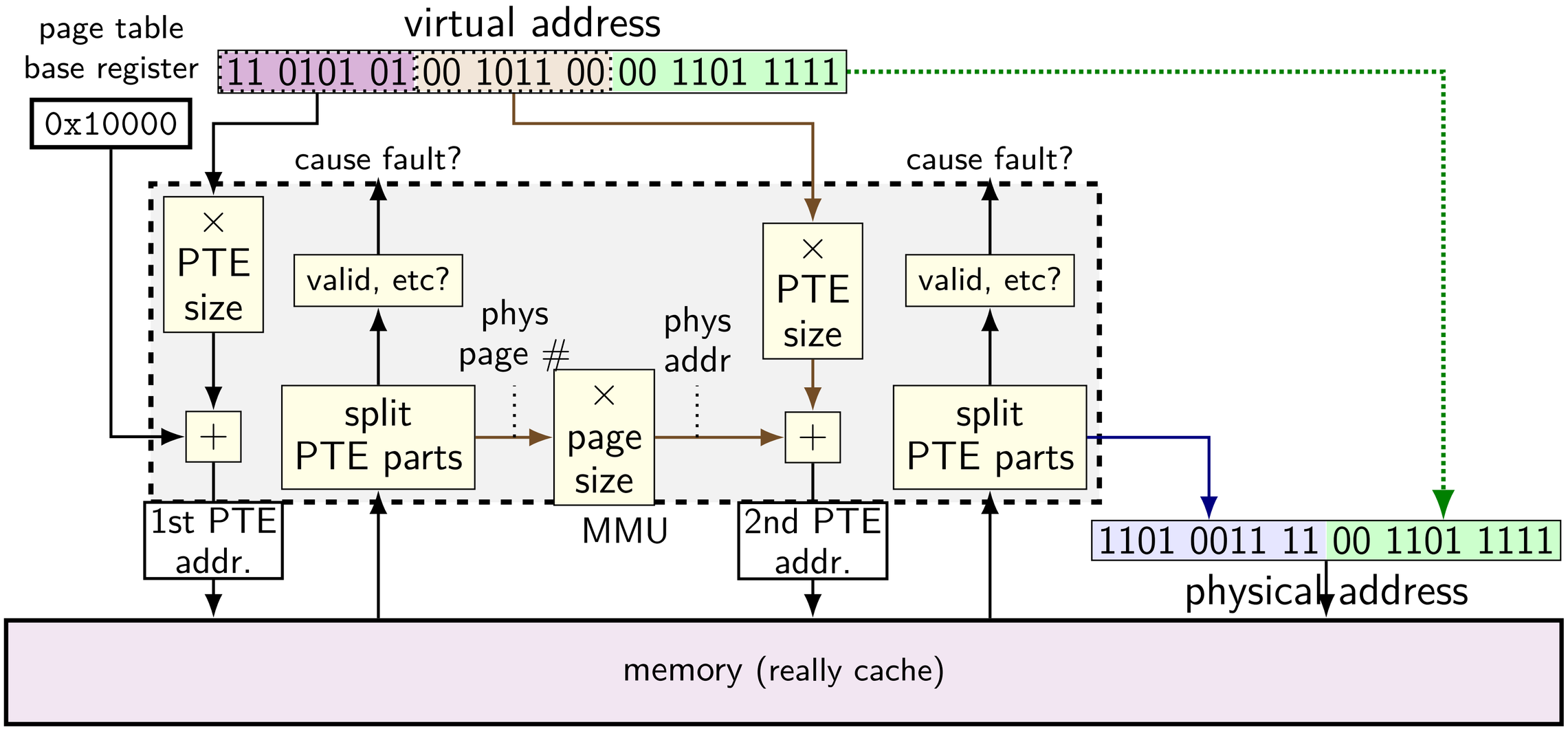
two-level page table lookup



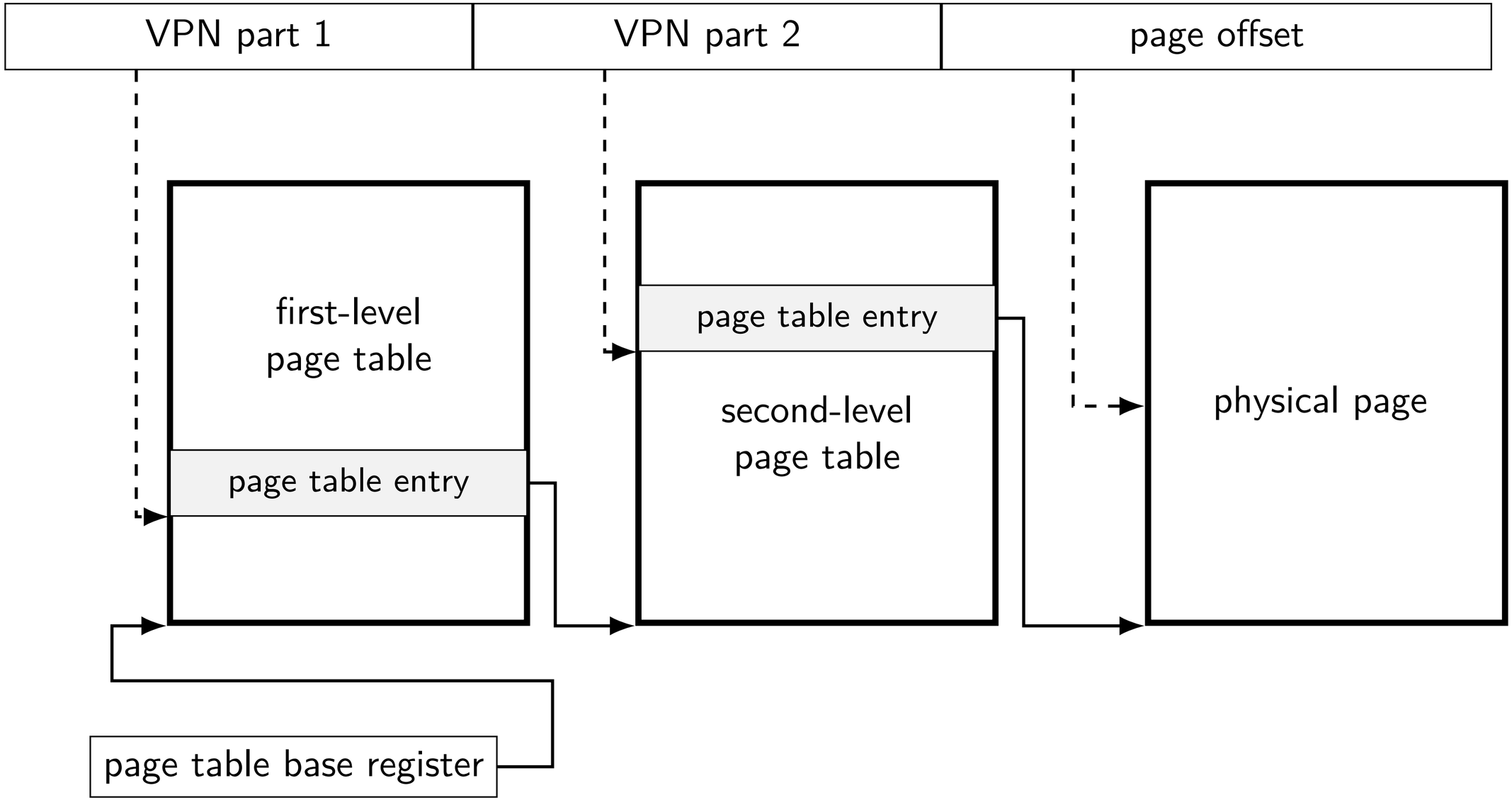
two-level page table lookup



two-level page table lookup



another view



cache accesses and multi-level PTs

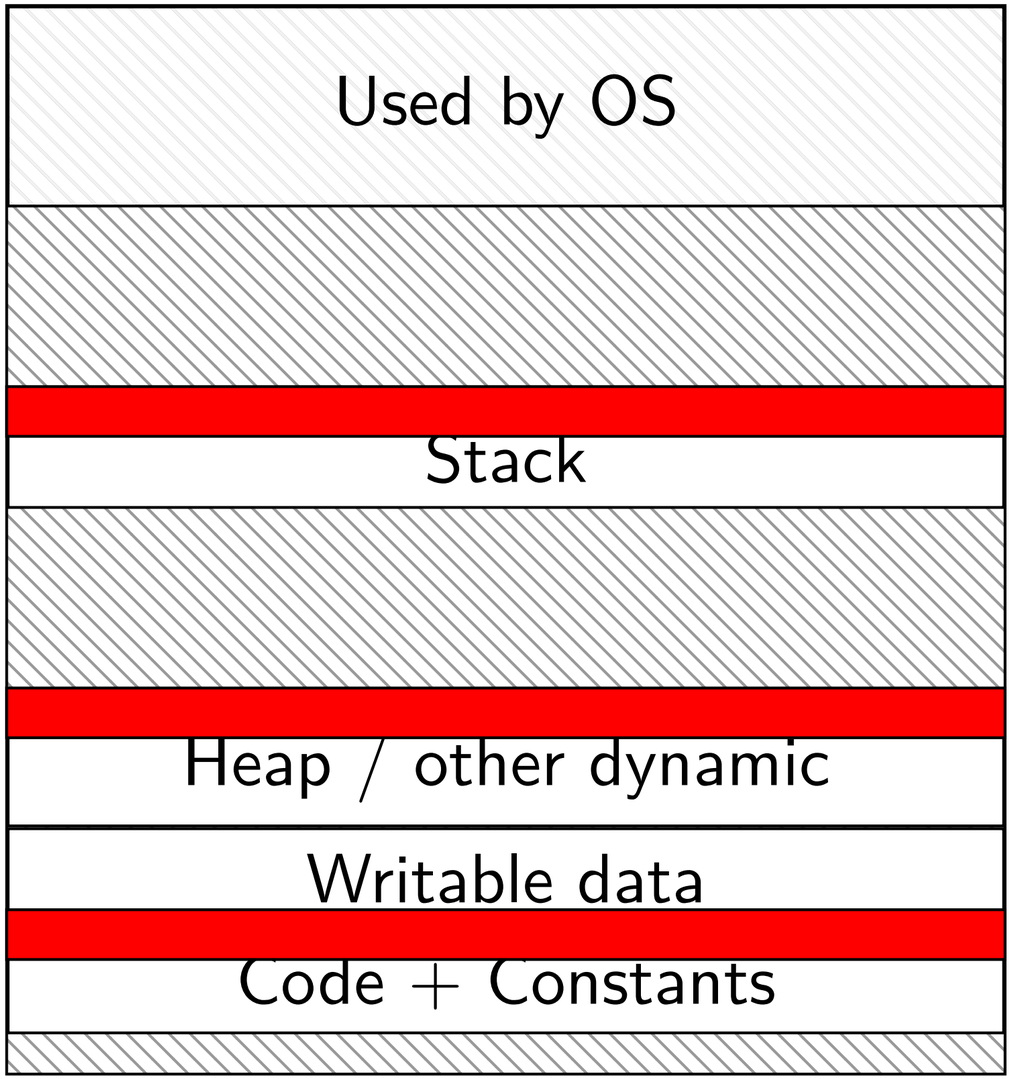
four-level page tables — five cache accesses per program memory access

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.

needed page table entries are *very small*

page table entry cache

caled a *TLB* (translation lookaside buffer)

(usually 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)

(usually 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)

(usually 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

virtual page number divided into index + tag

page table entry cache

caled a *TLB* (translation lookaside buffer)

(usually 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)

page table entry cache

caled a *TLB* (translation lookaside buffer)

(usually 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

`0` block offset bits

page table entry cache

caled a *TLB* (translation lookaside buffer)

(usually 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

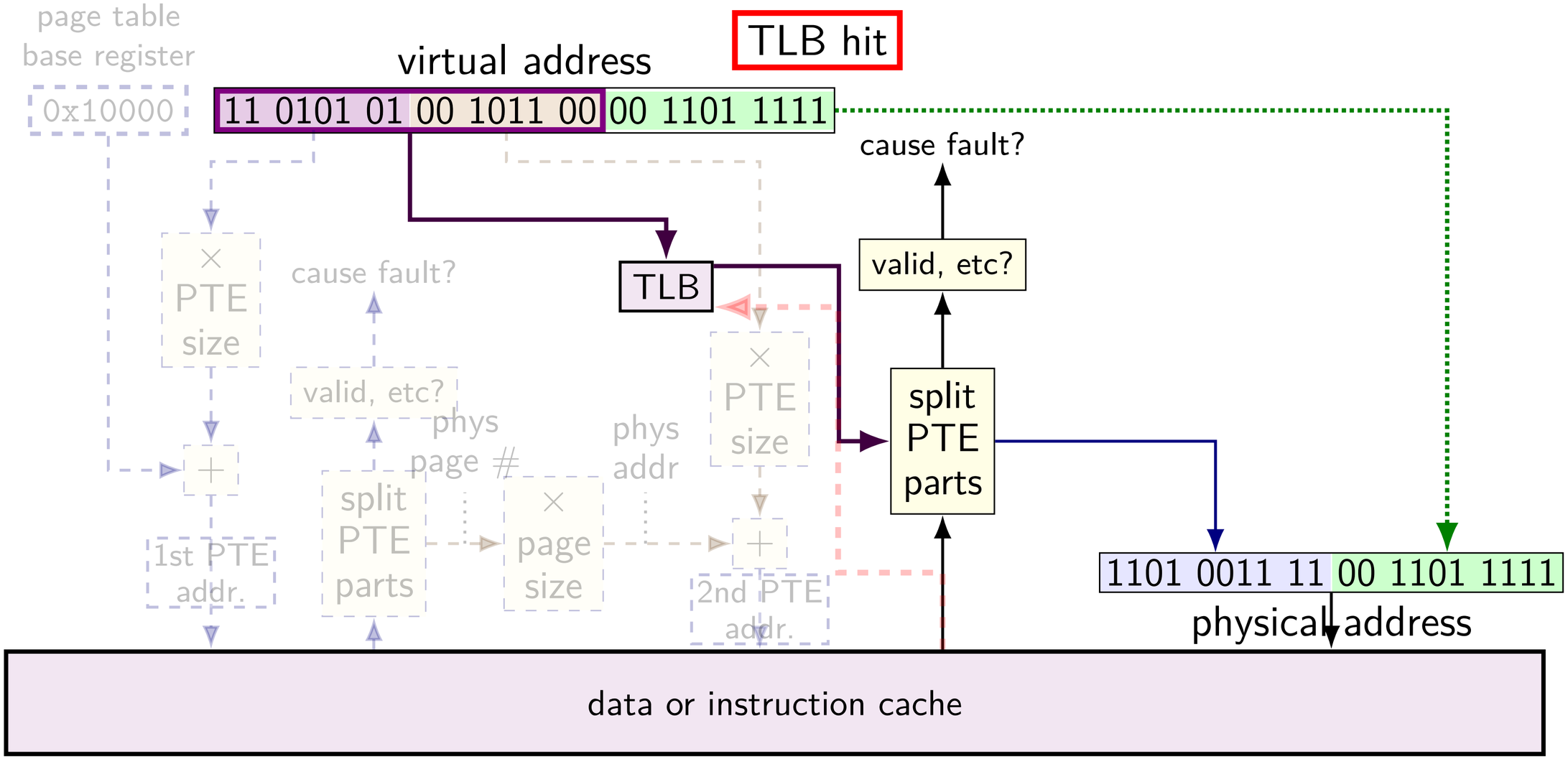
TLB caches *valid last-level page table entries*

doesn’t matter which last-level page table

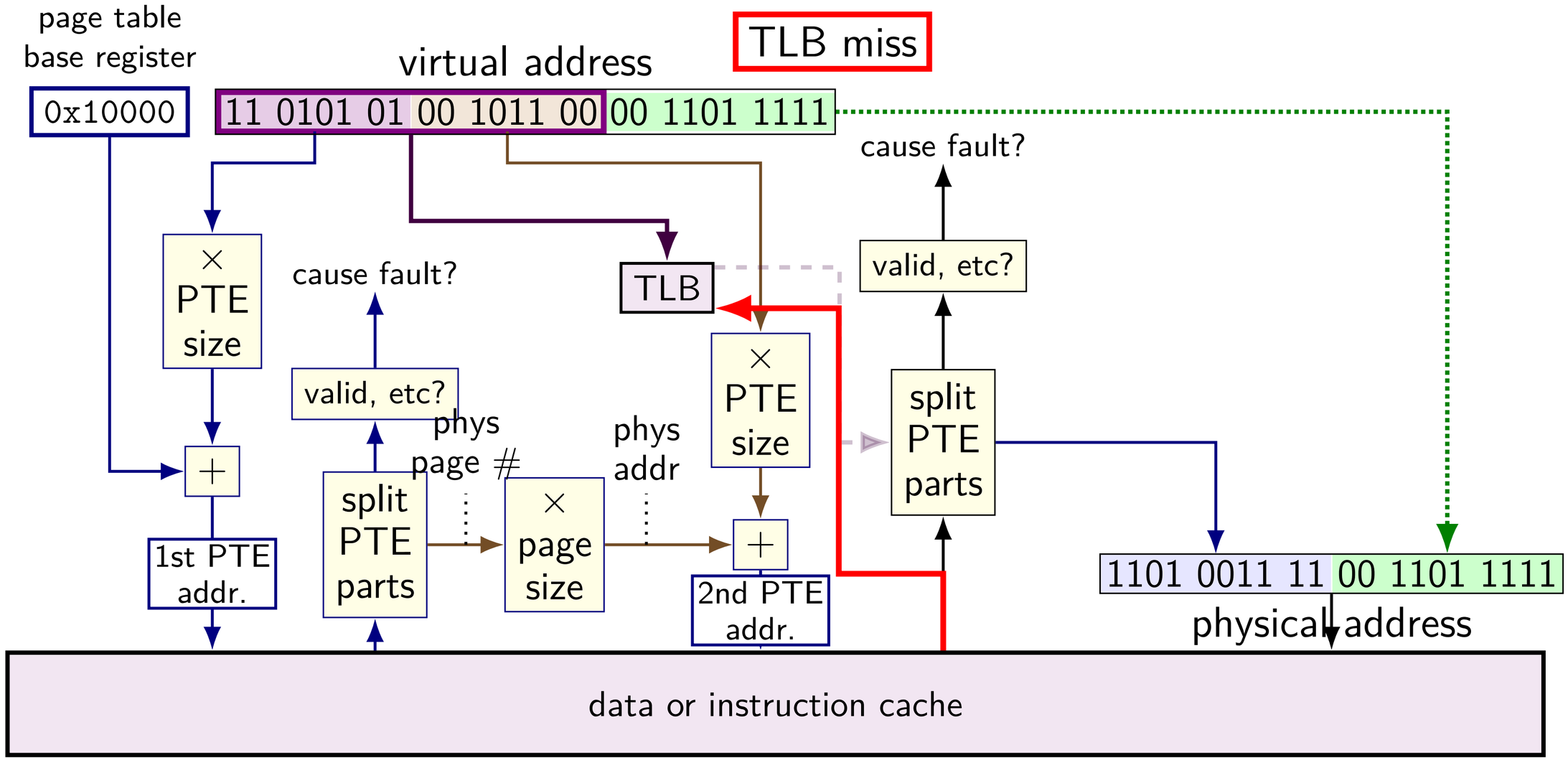
means TLB output can be used directly to form address

TLB and two-level lookup

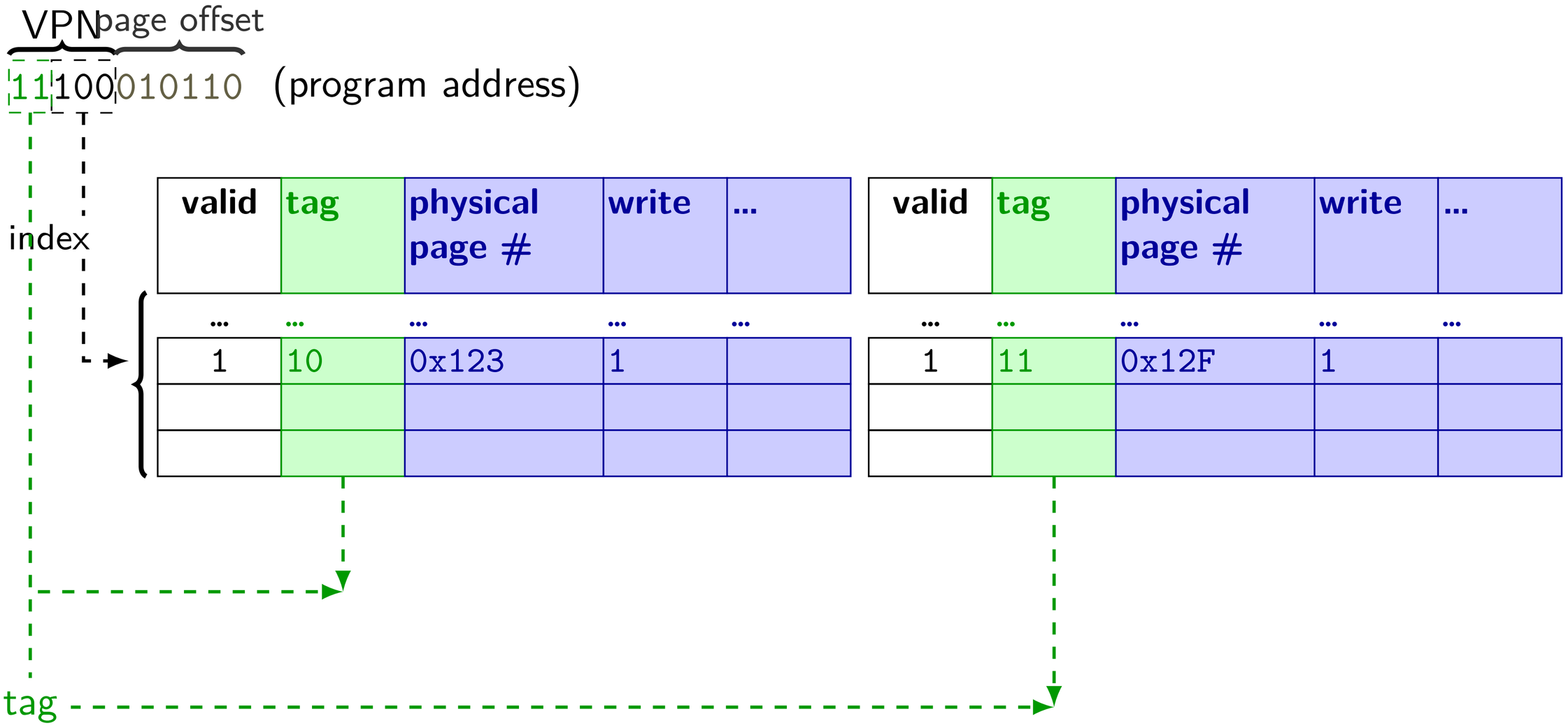
TLB and two-level lookup



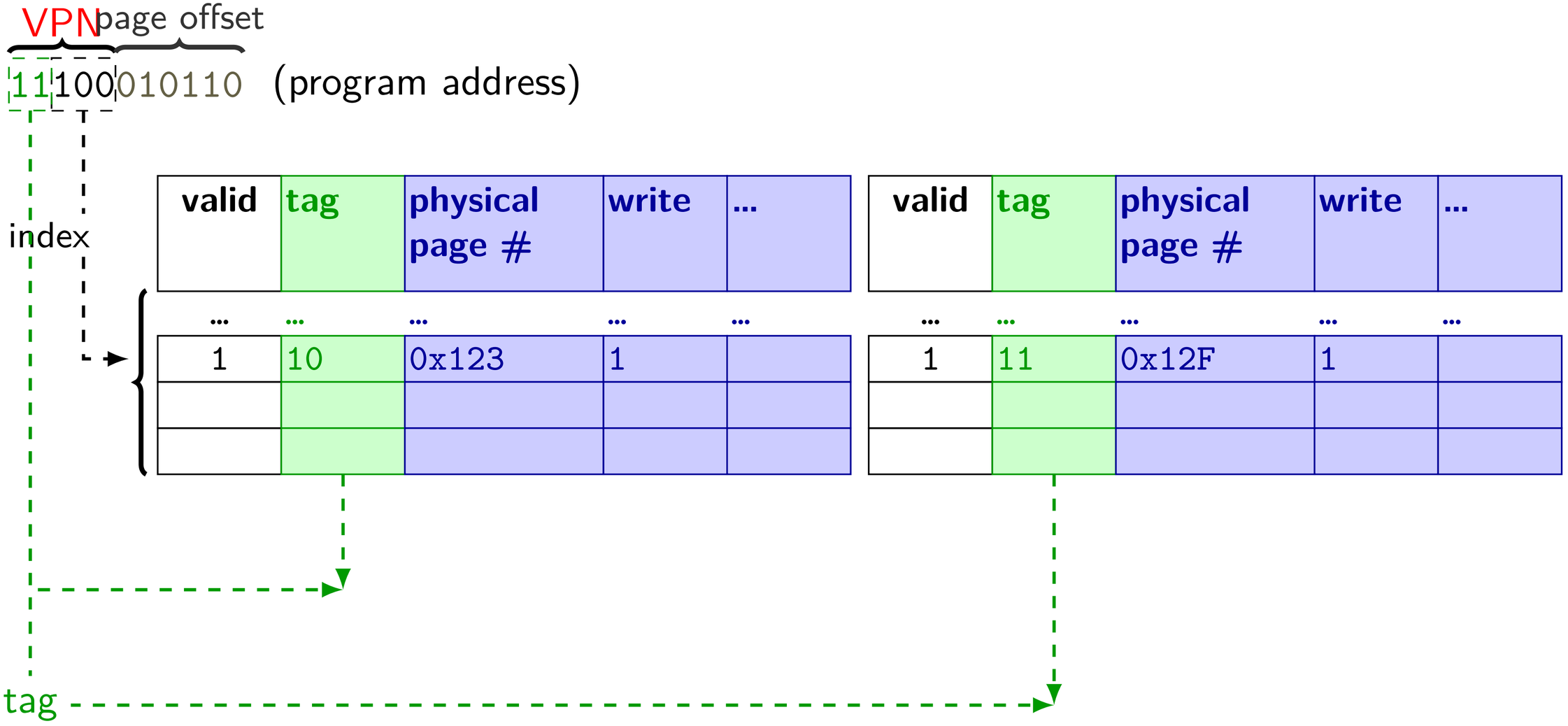
TLB and two-level lookup



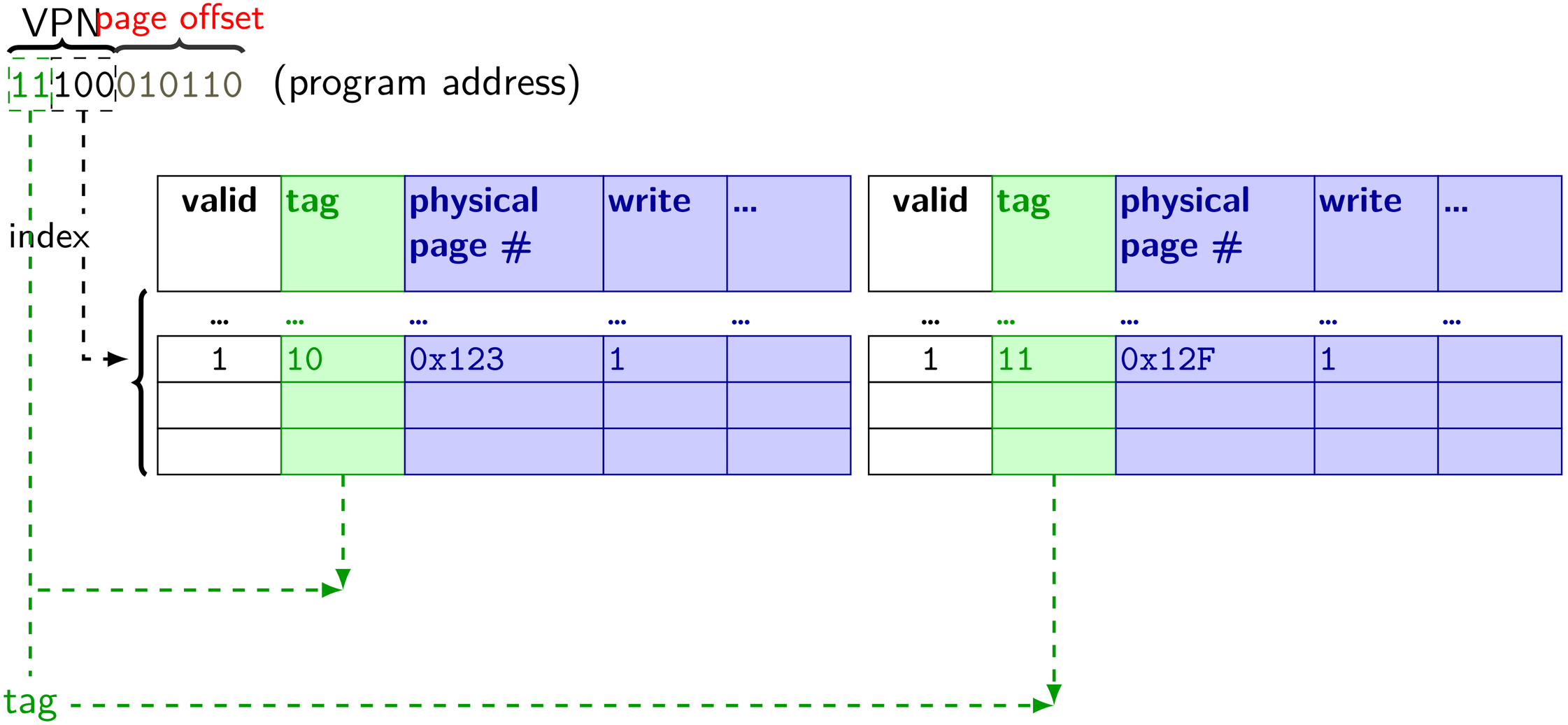
TLB organization (2-way set associative)



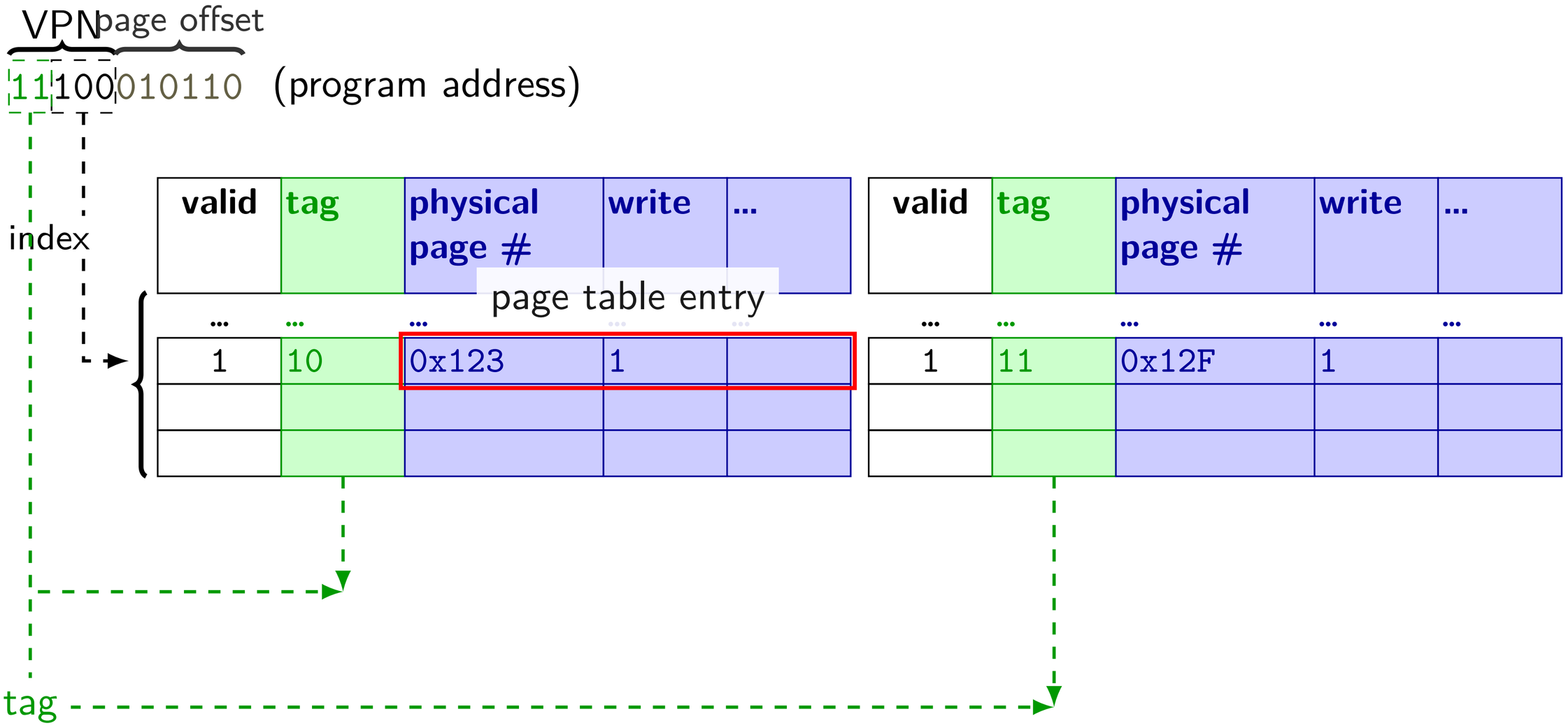
TLB organization (2-way set associative)



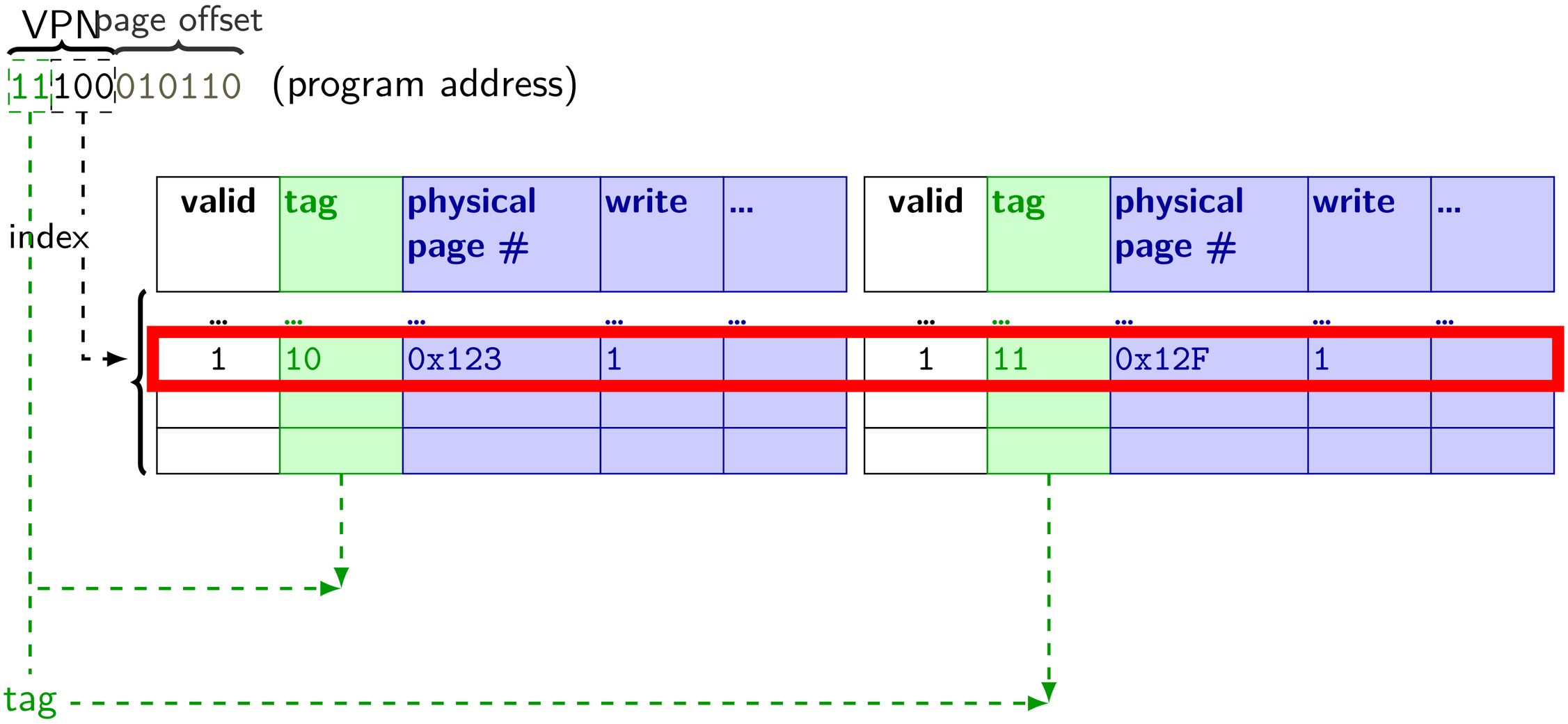
TLB organization (2-way set associative)



TLB organization (2-way set associative)



TLB organization (2-way set associative)



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	result	set 0	set 1
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?

solution: TLB access pattern

type	virtual	physical	result	set 0	set 1
read	0x440030	0x554030	miss	0x440	
write	0x440034	0x554034	hit	0x440	
read	0x7FFFE008	0x556008	miss	0x440, 0x7FFFE	
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

solution: TLB access pattern

type	virtual	physical	result	set 0	set 1
read	0x440030	0x554030	miss	0x440	
write	0x440034	0x554034	hit	0x440	
read	0x7FFFE008	0x556008	miss	0x440, 0x7FFFE	
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

set idx	V	tag	physical page	write?	user?	...	LRU?
0	1	0x00220 (0x00440 >> 1)	0x554	1	1	...	no
	1	0x00322 (0x00664 >> 1)	0x5F9	1	1	...	yes
1	1	0x3FFFF (0x7FFFD >> 1)	0x554	1	1	...	no
	0					...	yes

Backup slides

cache accesses and C code (1)

```
int scaleFactor;

int scaleByFactor(int value) {
    return value * scaleFactor;
}
```

```
scaleByFactor:
    movl scaleFactor, %eax
    imull %edi, %eax
    ret
```

---

exericse: what data cache accesses does this function do?

- 4-byte read of scaleFactor
- 8-byte read of return address

possible scaleFactor use

```
for (int i = 0; i < size; ++i) {
    array[i] = scaleByFactor(array[i]);
}
```

misses and code (2)

```
scaleByFactor:
    movl scaleFactor, %eax
    imull %edi, %eax
    ret
```

suppose each time this is called in the loop:

return address located at address `0x7fffffe43b8`

scaleFactor located at address `0x6bc3a0`

with direct-mapped 32KB cache w/64 B blocks, what is their:

	return address	scaleFactor
tag		
index		
offset		

misses and code (2)

```
scaleByFactor:
    movl scaleFactor, %eax
    imull %edi, %eax
    ret
```

suppose each time this is called in the loop:

return address located at address `0x7fffffe43b8`

scaleFactor located at address `0x6bc3a0`

with direct-mapped 32KB cache w/64 B blocks, what is their:

	return address	scaleFactor
tag	0xffffffc	0xd7
index	*0x10e*	*0x10e*
offset	0x38	0x20

conflict miss coincidences?

obviously I set that up to have the same index

- have to use exactly the right amount of stack space…

but one of the reasons we'll want something better than direct-mapped cache

C and cache misses (warmup 3)

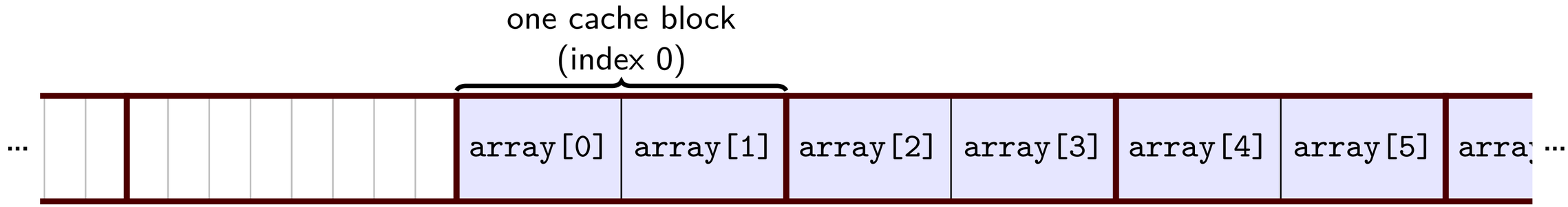
```
int array[8];
...
int even_sum = 0, odd_sum = 0;
even_sum += array[0];
odd_sum += array[1];
even_sum += array[2];
odd_sum += array[3];
even_sum += array[4];
odd_sum += array[5];
even_sum += array[6];
odd_sum += array[7];
```

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

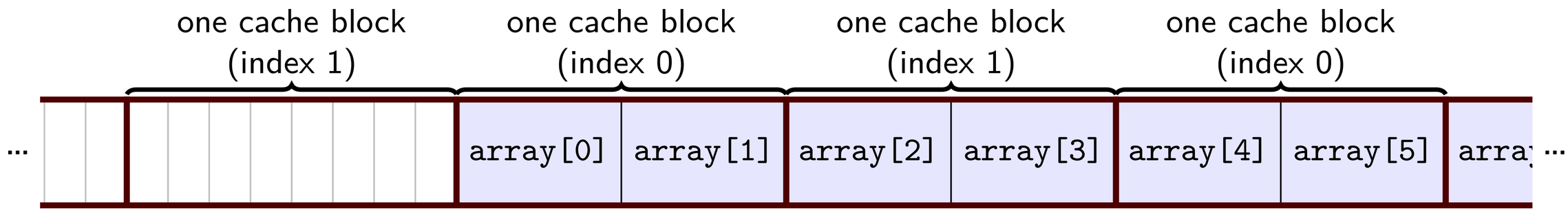
How many data cache misses on a 2-set direct-mapped cache with 8B blocks?

exercise solution

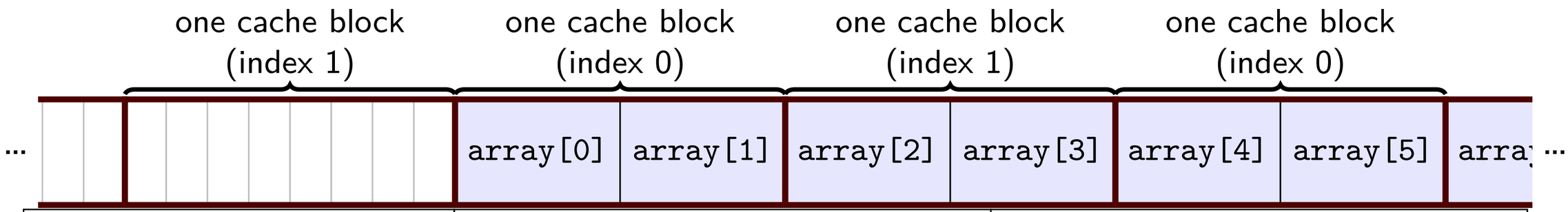
exercise solution



exercise solution

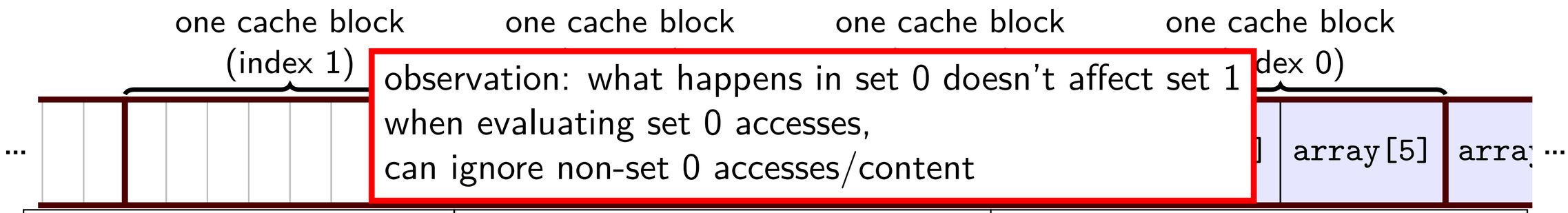


exercise solution



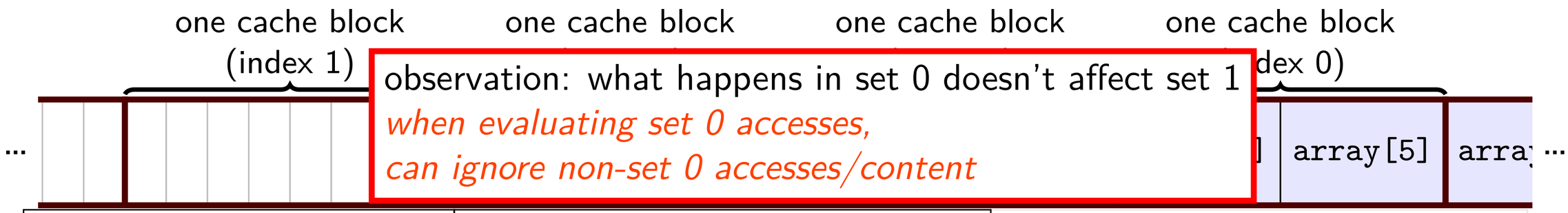
memory access	**set 0 afterwards**	**set 1 afterwards**
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[1] (hit)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[3] (hit)	{array[0], array[1]}	{array[2], array[3]}
read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



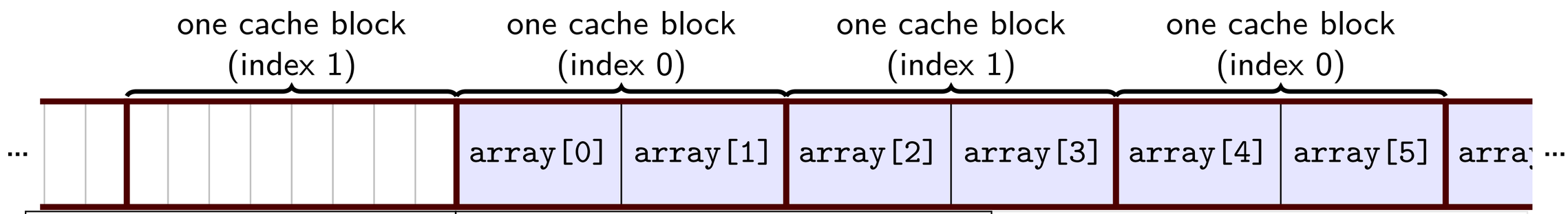
memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[1] (hit)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[3] (hit)	{array[0], array[1]}	{array[2], array[3]}
read array[4]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

exercise solution



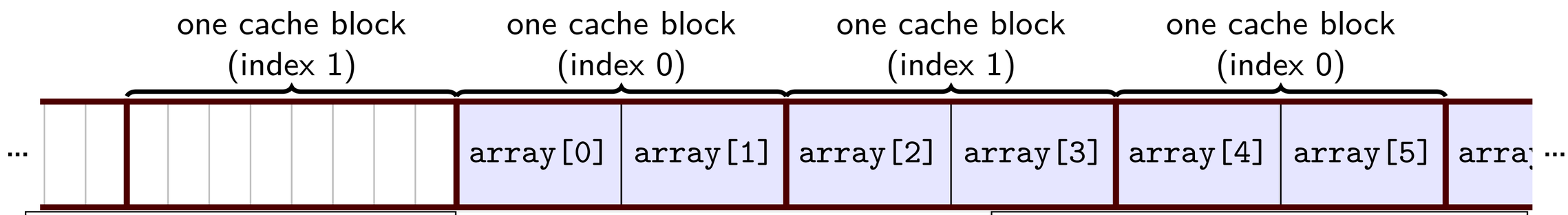
memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[1] (hit)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[3] (hit)	{array[0], array[1]}	{array[2], array[3]}
read array[4]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

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 initially empty 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 initially empty 2KB direct-mapped cache with 16B cache blocks?

explanation

2-way, 2KB set associative cache, 16B blocks

4 offset bits, 6 index bits

so addresses multiples $2^1 0$ bytes apart differ only in tag bits

example: array[0$\rightarrow{}3], $array[256\rightarrow{}259]$, $array[512 \rightarrow{}515]$, $array[768\rightarrow{}$771]

those all use the same set

but sets only holds 2 things

all misses

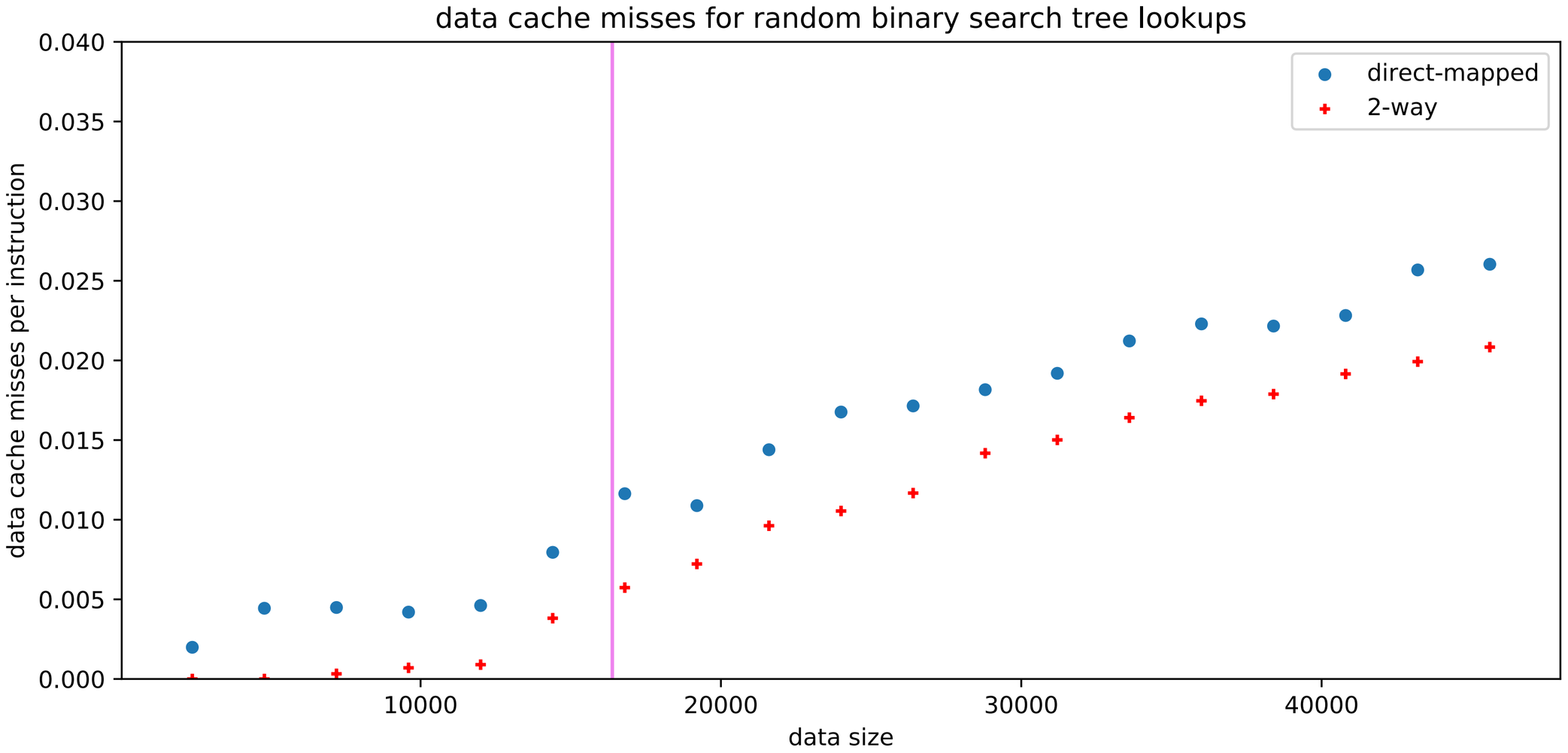
arrays and cache misses (2b)

```
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];
```

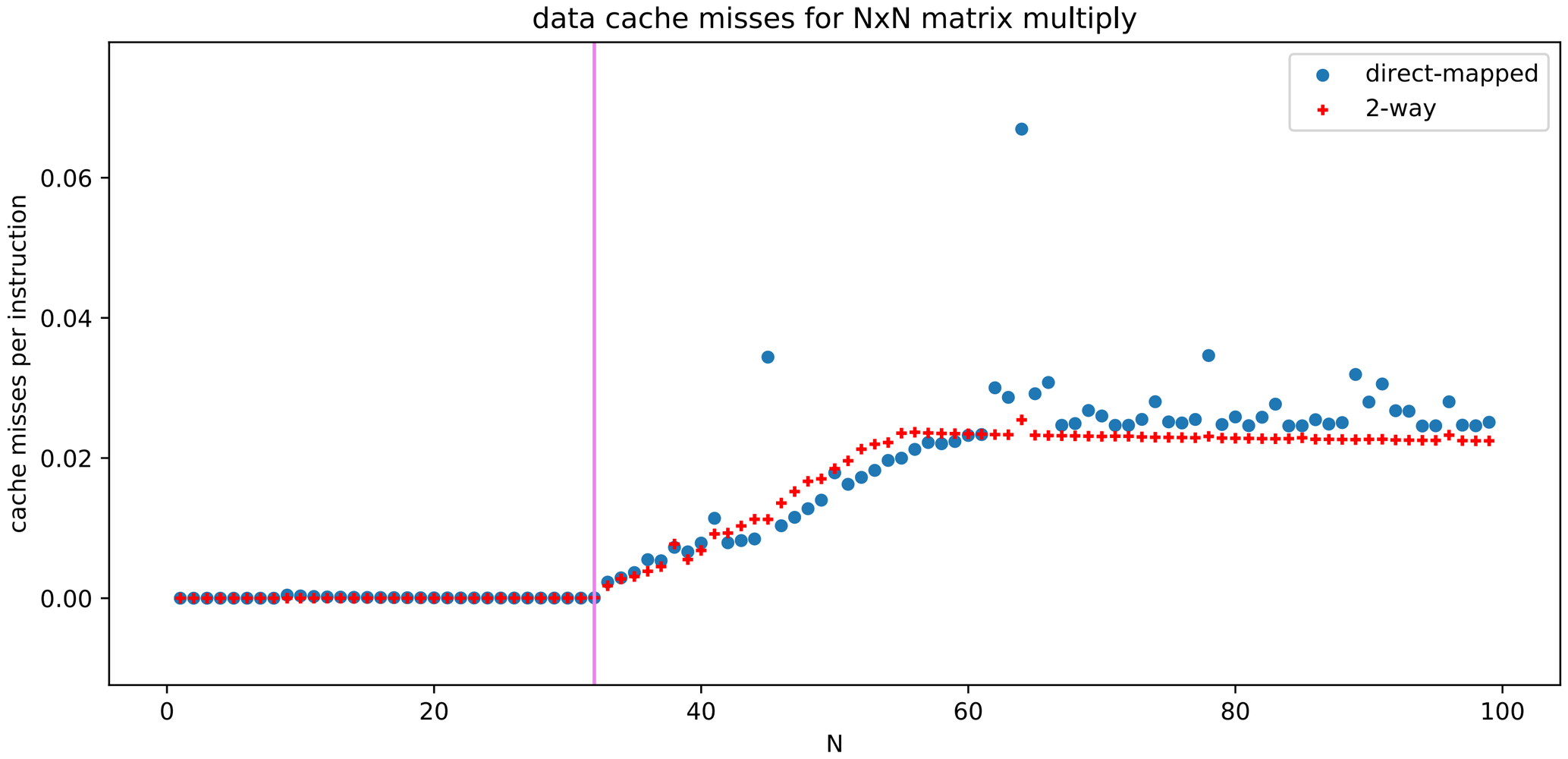
Assume everything but `array` is kept in registers (and the compiler does not do anything funny).

How many *data cache misses* on initially empty *4KB* direct-mapped cache with 16B cache blocks?

simulated misses: BST lookups



simulated misses: matrix multiplies

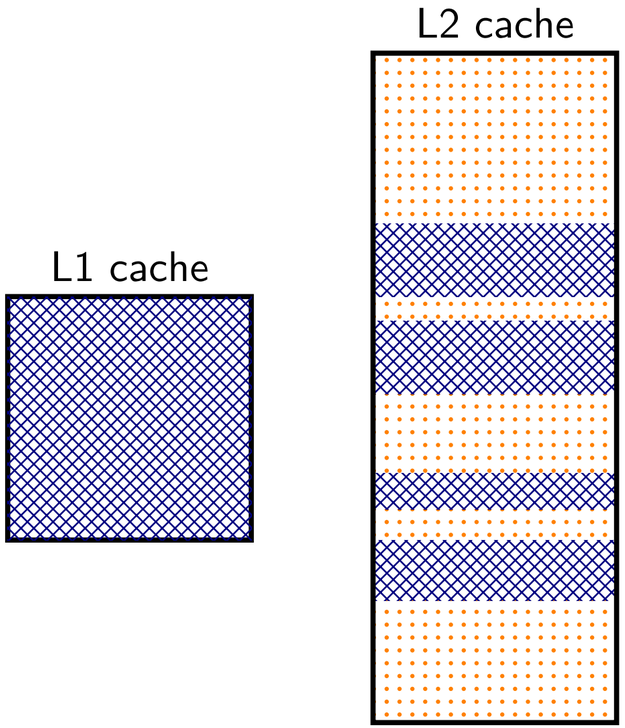


inclusive versus exclusive

inclusive versus exclusive

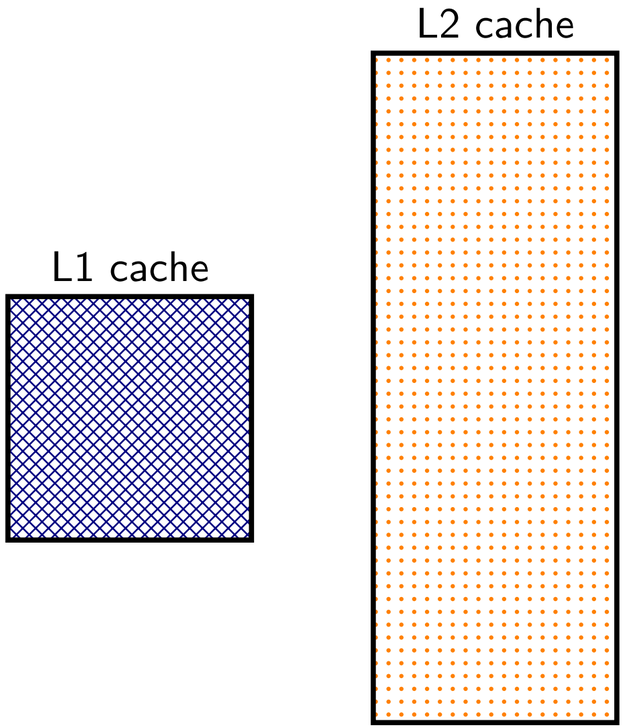
L2 inclusive of L1

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

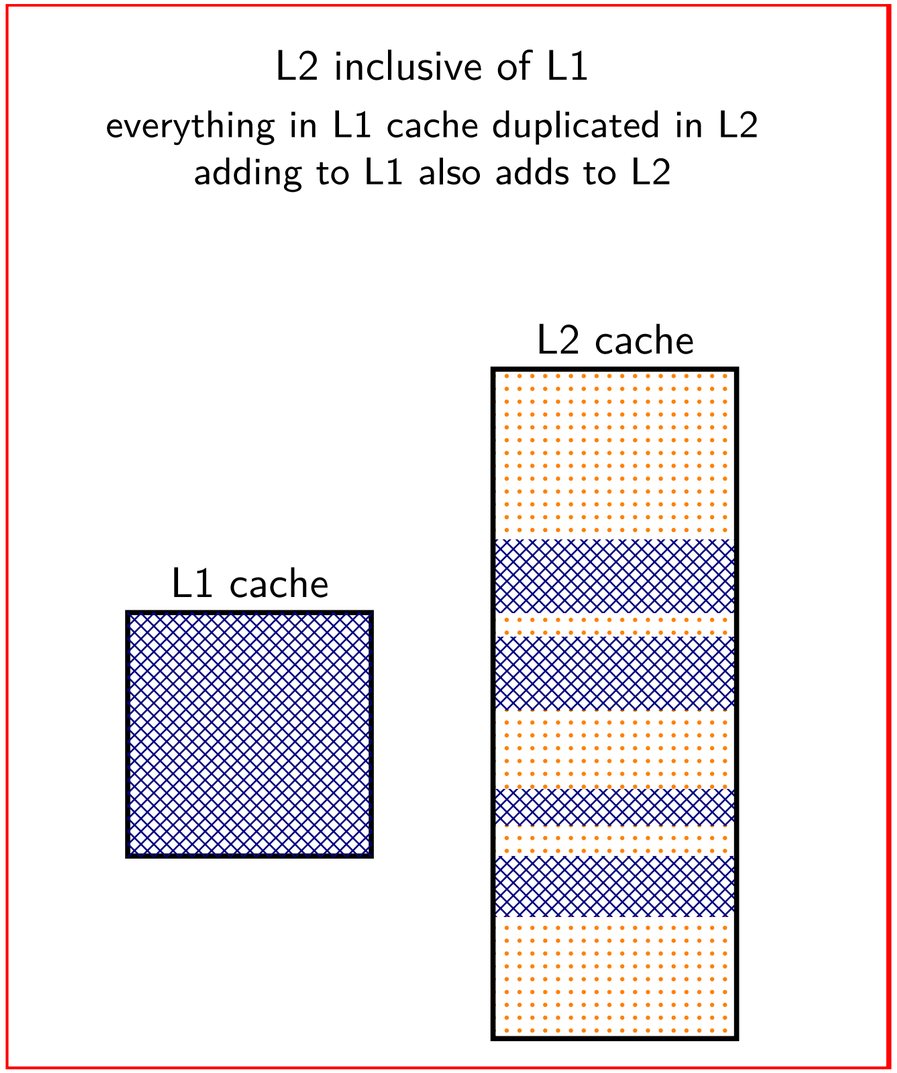


L2 exclusive of L1

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



inclusive versus exclusive



inclusive policy:
no extra work on eviction
but duplicated data

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

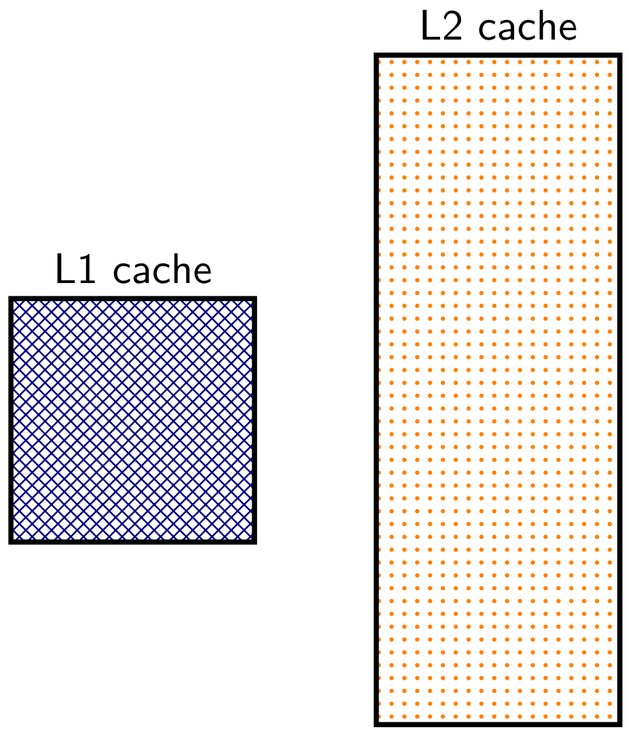
inclusive versus exclusive

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)

example access pattern (1)

example access pattern (1)

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

2 byte blocks, 4 sets

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

example access pattern (1)

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

2 byte blocks, 4 sets

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)

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

tag index offset

2 byte blocks, 4 sets

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)

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

tag index offset

2 byte blocks, 4 sets

index	valid	tag	value
00	*1*	*00000*	*mem[0x00]* *mem[0x01]*
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)

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

tag index offset

2 byte blocks, 4 sets

index	valid	tag	value
00	1	*00000*	*mem[0x00]* *mem[0x01]*
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)

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

tag index offset

2 byte blocks, 4 sets

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

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

2 byte blocks, 4 sets

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)

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

2 byte blocks, 4 sets

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

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

2 byte blocks, 4 sets

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

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

2 byte blocks, 4 sets

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		

$m = 8$ 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)

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

2 byte blocks, 4 sets

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		

$m = 8$ 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)

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

2 byte blocks, 4 sets

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*

$m = 8$ 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)

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

2 byte blocks, 4 sets

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*

$m = 8$ 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

cache organization and miss rate

depends on program; one example:

SPEC CPU2000 benchmarks, 64B block size, LRU replacement

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

prefetching

seems like we can't really improve cold misses…

have to have a miss to bring value into the cache?

solution: don't require miss: 'prefetch' the value before it's accessed

remaining problem: how do we know what to fetch?

common access patterns

suppose recently accessed 16B cache blocks are at:

0x48010, 0x48020, 0x48030, 0x48040

guess what's accessed next

common pattern with *instruction fetches* and *array accesses*

prefetching idea

look for sequential accesses

bring in guess at next-to-be-accessed value

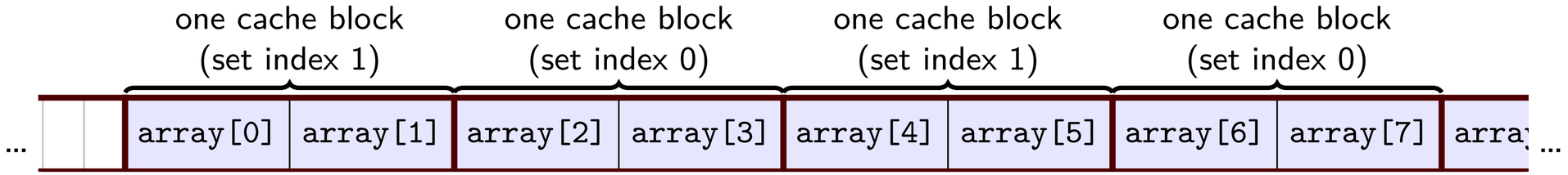
if right: no cache miss (even if never accessed before)

if wrong: possibly evicted something else — could cause more misses

- fortunately, sequential access guesses almost always right

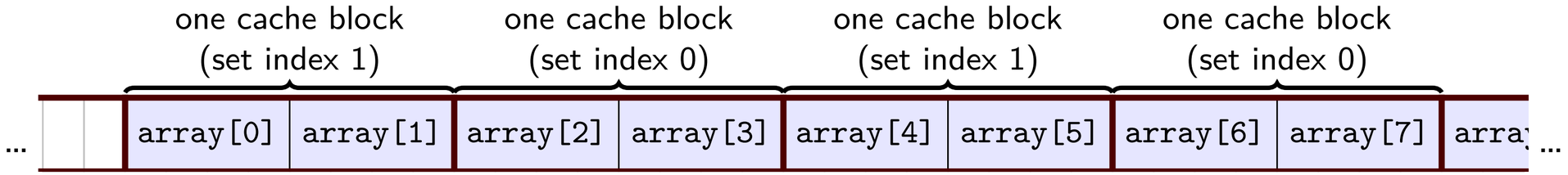
quiz exercise solution

quiz exercise solution



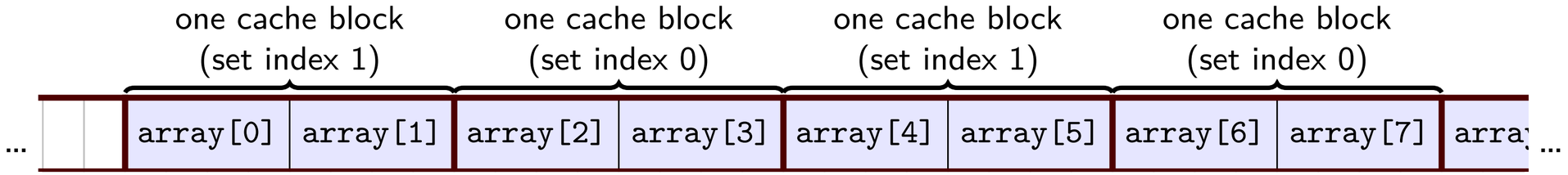
memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[3] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[6] (miss)	{array[0], array[1]}	{array[6], array[7]}
read array[1] (hit)	{array[0], array[1]}	{array[6], array[7]}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

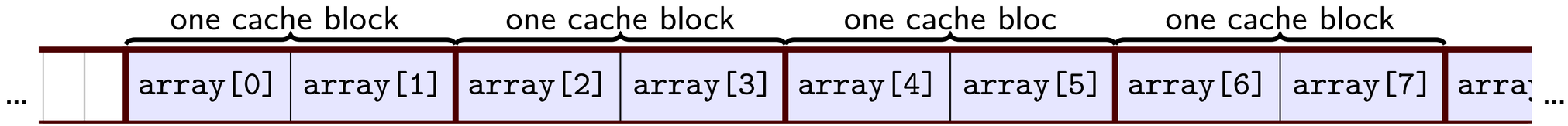
quiz exercise solution



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

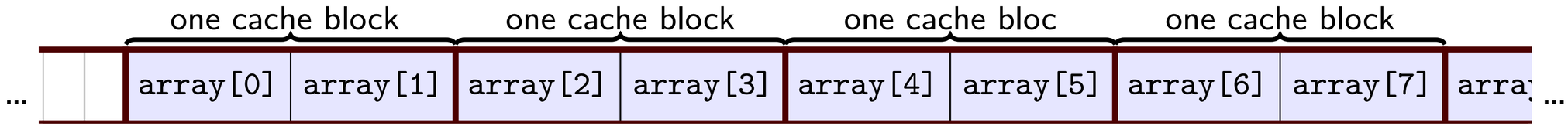
not the quiz problem



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]}`

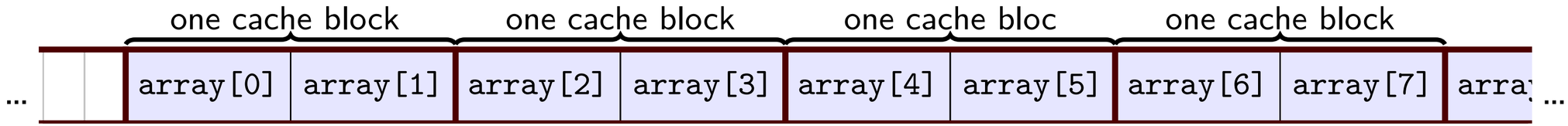
not the quiz problem



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]}`

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]}`

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

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
array1	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
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

2 misses for set 1

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 array1, 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 array1, 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

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 to 15) + 2KB, (0 to 15) + 4KB, …

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

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

set 1: address 16 to 31, (16 to 31) + 2KB, (16 to 31) + 4KB, …

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

block at 16+2KB: array[516] through array[519]

…

set 127: address 2032 to 2047, (2032 to 2047) + 2KB, …

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

block at 2032+2KB: array[1020] through array[1023]

thinking about cache storage (1)

2KB direct-mapped cache with 16B blocks —

set 0: address 0 to 15, (0 to 15) + 2KB, (0 to 15) + 4KB, …

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

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

set 1: address 16 to 31, (16 to 31) + 2KB, (16 to 31) + 4KB, …

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

block at 16+2KB: array[516] through array[519]

…

set 127: address 2032 to 2047, (2032 to 2047) + 2KB, …

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

block at 2032+2KB: 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 + 2KB, 0 + 4KB, …

- 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 + 2KB, 16 + 4KB, …

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

…

set 63: address 1008, 2032 + 2KB, 2032 + 4KB …

- 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 + 2KB, 0 + 4KB, …

- 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 + 2KB, 16 + 4KB, …

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

…

set 63: address 1008, 2032 + 2KB, 2032 + 4KB …

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

arrays and cache misses (3)

```
int sum; int array[1024]; // 4KB array
for (int i = 8; i < 1016; i += 1) {
    int local_sum = 0;
    for (int j = i - 8; j < i + 8; j += 1) {
        local_sum += array[i] * (j - i);
    }
    sum += (local_sum - array[i]);
}
```

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

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

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)	$B = 64\text{Byte}$
$B = 2^b$ (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`

L1:

- bits 6-11 (L1 set): `01 0101` = `0x15`
- bits 12- (L1 tag): 0x34

L2:

- bits 6-15 (set for L2): `01 0001 0101` = `0x115`
- bits 16-: `0x3`

L3:

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

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

L2:

- bits 6-15 (set for L2): `01 0001 0101` = `0x115`
- bits 16-: `0x3`

L3:

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

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

L2:

- bits 6-15 (set for L2): `01 0001 0101` = `0x115`
- bits 16-: `0x3`

L3:

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

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

L2:

- bits 6-15 (set for L2): `01 0001 0101` = `0x115`
- bits 16-: `0x3`

L3:

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

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*

L2:

- bits 6-15 (set for L2): `01 0001 0101` = `0x115`
- bits 16-: `0x3`

L3:

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

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

L2:

- bits 6-15 (set for L2): `01 0001 0101` = `0x115`
- bits 16-: `0x3`

L3:

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

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

L2:

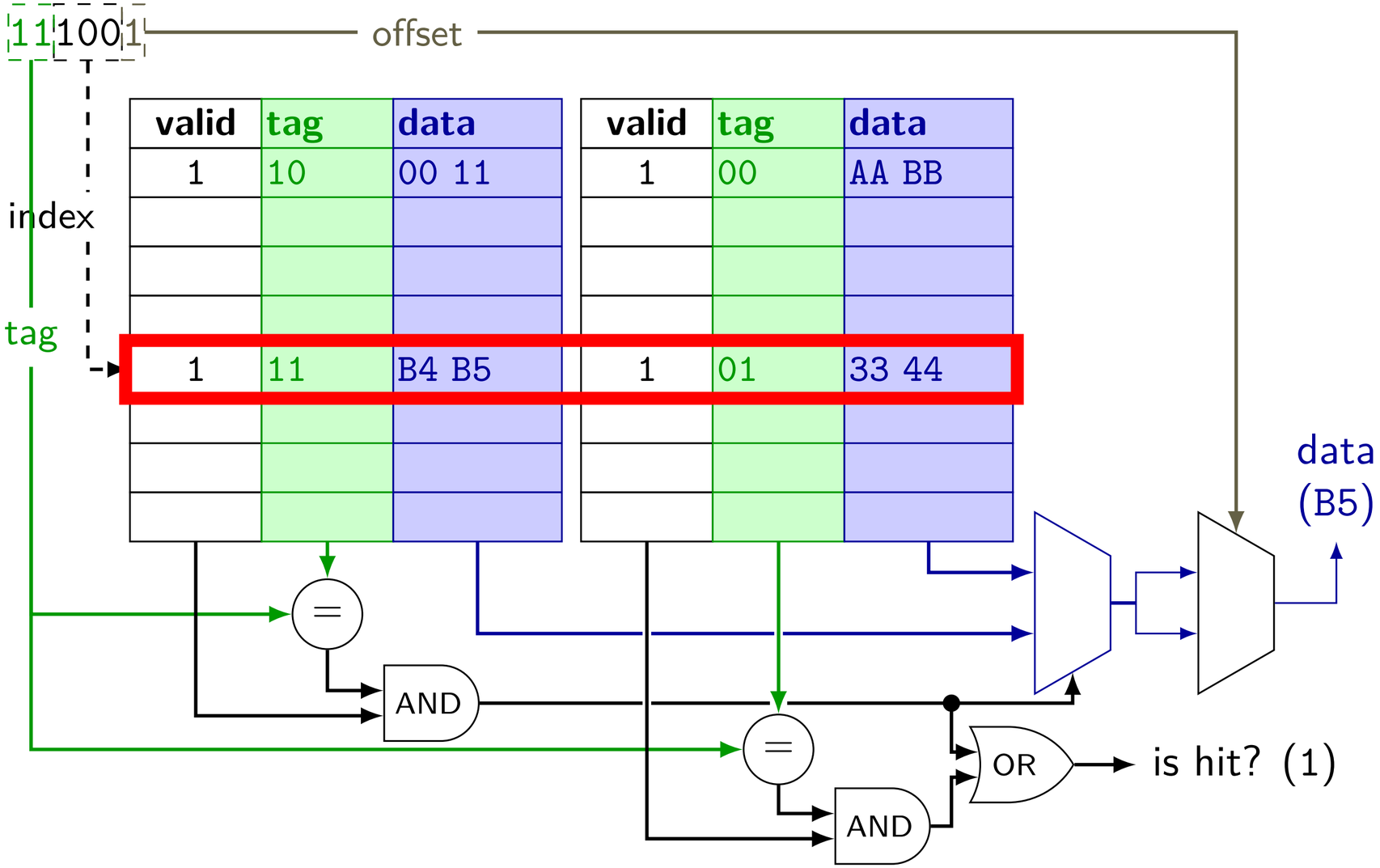
- bits 6-15 (set for L2): `01 0001 0101` = `0x115`
- bits 16-: `0x3`

L3:

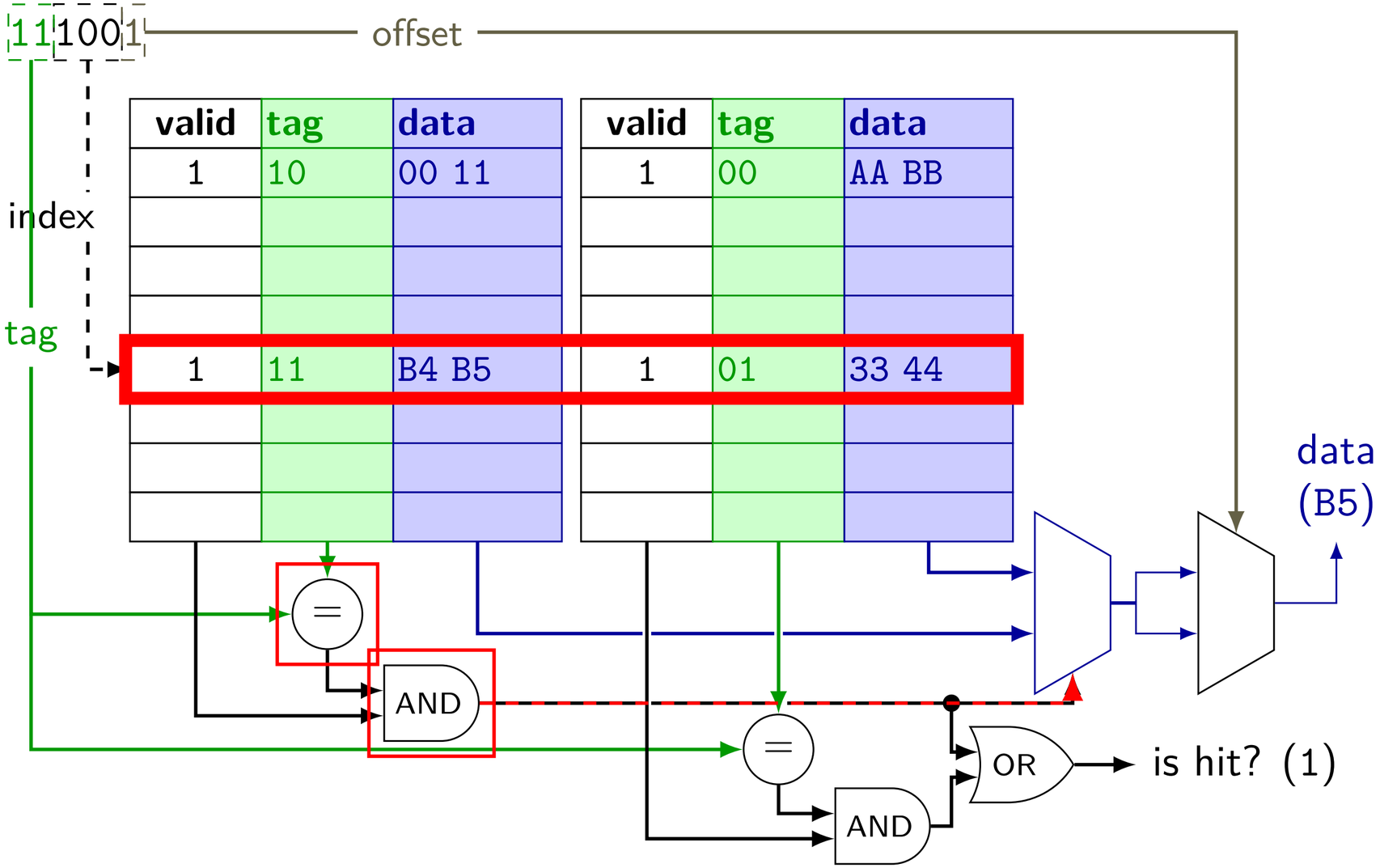
- bits 6-18 (set for L3): `0 1101 0001 0101` = `0xD15`
- bits 18-: `0x0`

cache operation (associative)

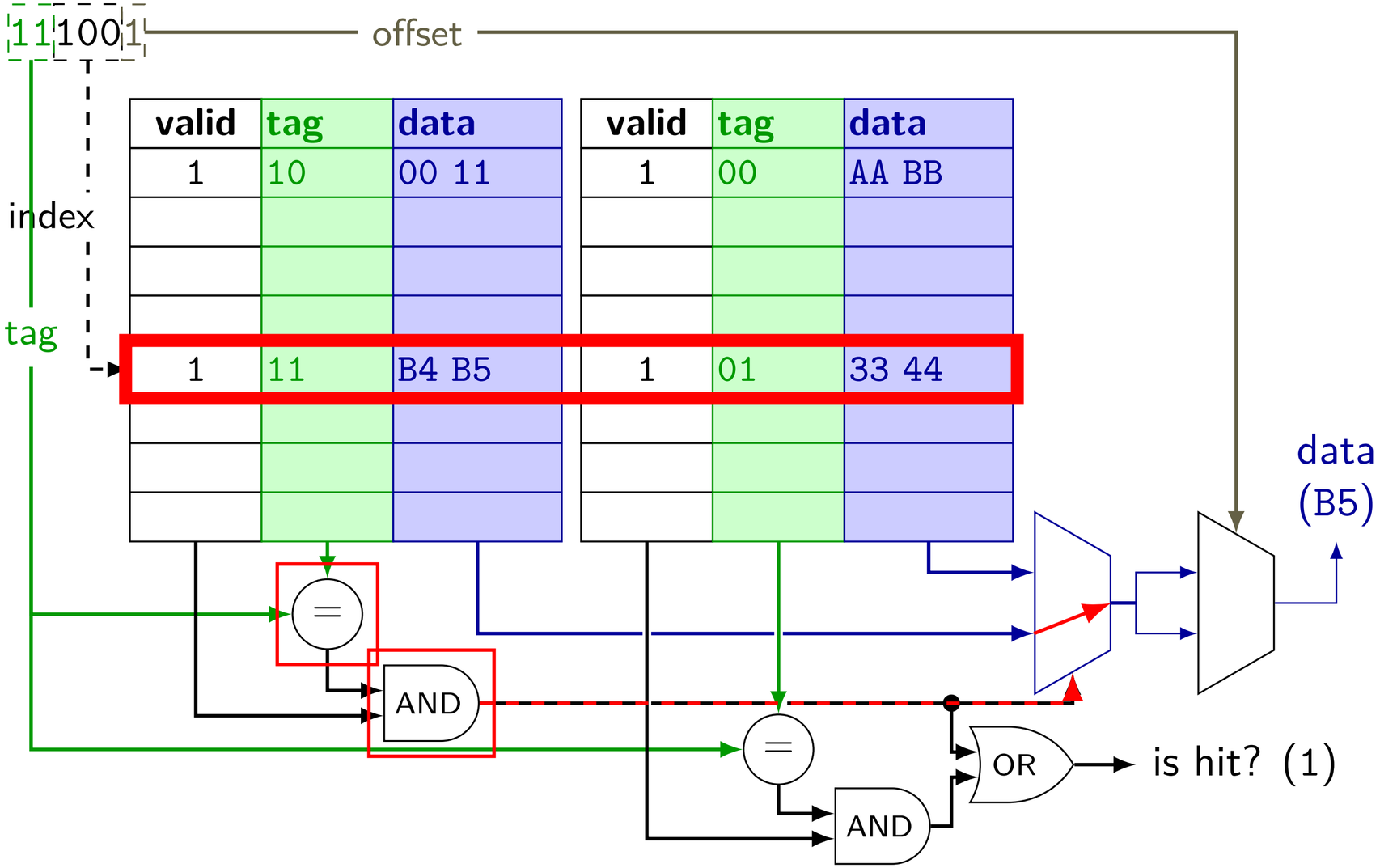
cache operation (associative)



cache operation (associative)



cache operation (associative)



backup slides — cache performance

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

coherence — from sync'ing cache with other caches

only issue with multiple cores

making any cache look bad

access enough blocks, to fill the cache

access an additional block, replacing something

access last block replaced

access last block replaced

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	good	bad	—
increase associativity	good	bad	bad?
increase block size	depends	bad	bad
add secondary cache	—	—	good
write-allocate	good	—	?
writeback	—	—	?
LRU replacement	good	?	bad?
prefetching	good	—	—

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	good	good	—
increase associativity	—	good	—
increase block size	bad?	bad?	good
LRU replacement	—	good	—
prefetching	—	—	good

average memory access time

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

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

effective speed of memory

AMAT exercise (1)

90% cache hit rate

hit time is 2 cycles

30 cycle miss penalty

what is the average memory access time?

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

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

AMAT exercise (1)

90% cache hit rate

hit time is 2 cycles

30 cycle miss penalty

what is the average memory access time?

5 cycles

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

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

AMAT exercise (1)

90% cache hit rate

hit time is 2 cycles

30 cycle miss penalty

what is the average memory access time?

5 cycles

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

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

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

exercise: AMAT and multi-level caches

suppose we have L1 cache with

- 3 cycle hit time, 90% hit rate

and an L2 cache with

- 10 cycle hit time, 80% hit rate (for accesses that make this far)
- (assume all accesses come via this L1)

and main memory has a 100 cycle access time

assume when there's an cache miss, the next level access starts after the hit time

- e.g. an access that misses in L1 and hits in L2 will take 10+3 cycles

what is the average memory access time for the L1 cache?

exercise: AMAT and multi-level caches

suppose we have L1 cache with

- 3 cycle hit time, 90% hit rate

and an L2 cache with

- 10 cycle hit time, 80% hit rate (for accesses that make this far)
- (assume all accesses come via this L1)

and main memory has a 100 cycle access time

assume when there's an cache miss, the next level access starts after the hit time

- e.g. an access that misses in L1 and hits in L2 will take 10+3 cycles

what is the average memory access time for the L1 cache?

- $3 + 0.1 \cdot (10 + 0.2 \cdot 100) = 6$ cycles
- L1 miss penalty is $10 + 0.2 \cdot 100 = 30$ cycles

example access pattern (1)

example access pattern (1)

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

2 byte blocks, 4 sets

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

example access pattern (1)

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

2 byte blocks, 4 sets

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)

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

tag index offset

2 byte blocks, 4 sets

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)

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

tag index offset

2 byte blocks, 4 sets

index	valid	tag	value
00	*1*	*00000*	*mem[0x00]* *mem[0x01]*
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)

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

tag index offset

2 byte blocks, 4 sets

index	valid	tag	value
00	1	*00000*	*mem[0x00]* *mem[0x01]*
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)

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

tag index offset

2 byte blocks, 4 sets

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

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

tag index offset

2 byte blocks, 4 sets

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

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

2 byte blocks, 4 sets

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

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

2 byte blocks, 4 sets

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

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

2 byte blocks, 4 sets

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		

$m = 8$ 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)

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

2 byte blocks, 4 sets

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		

$m = 8$ 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)

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

2 byte blocks, 4 sets

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*

$m = 8$ 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)

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

2 byte blocks, 4 sets

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*

$m = 8$ 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

cache organization and miss rate

depends on program; one example:

SPEC CPU2000 benchmarks, 64B block size, LRU replacement

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

prefetching

seems like we can't really improve cold misses…

have to have a miss to bring value into the cache?

solution: don't require miss: 'prefetch' the value before it's accessed

remaining problem: how do we know what to fetch?

common access patterns

suppose recently accessed 16B cache blocks are at:

0x48010, 0x48020, 0x48030, 0x48040

guess what's accessed next

common pattern with *instruction fetches* and *array accesses*

prefetching idea

look for sequential accesses

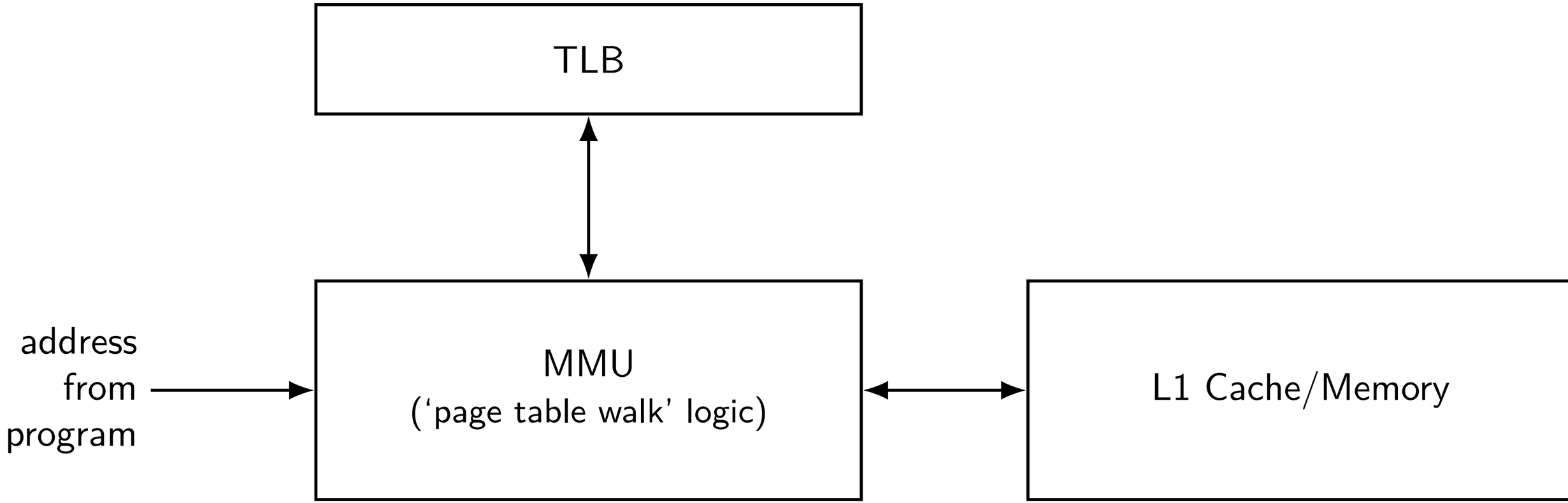
bring in guess at next-to-be-accessed value

if right: no cache miss (even if never accessed before)

if wrong: possibly evicted something else — could cause more misses

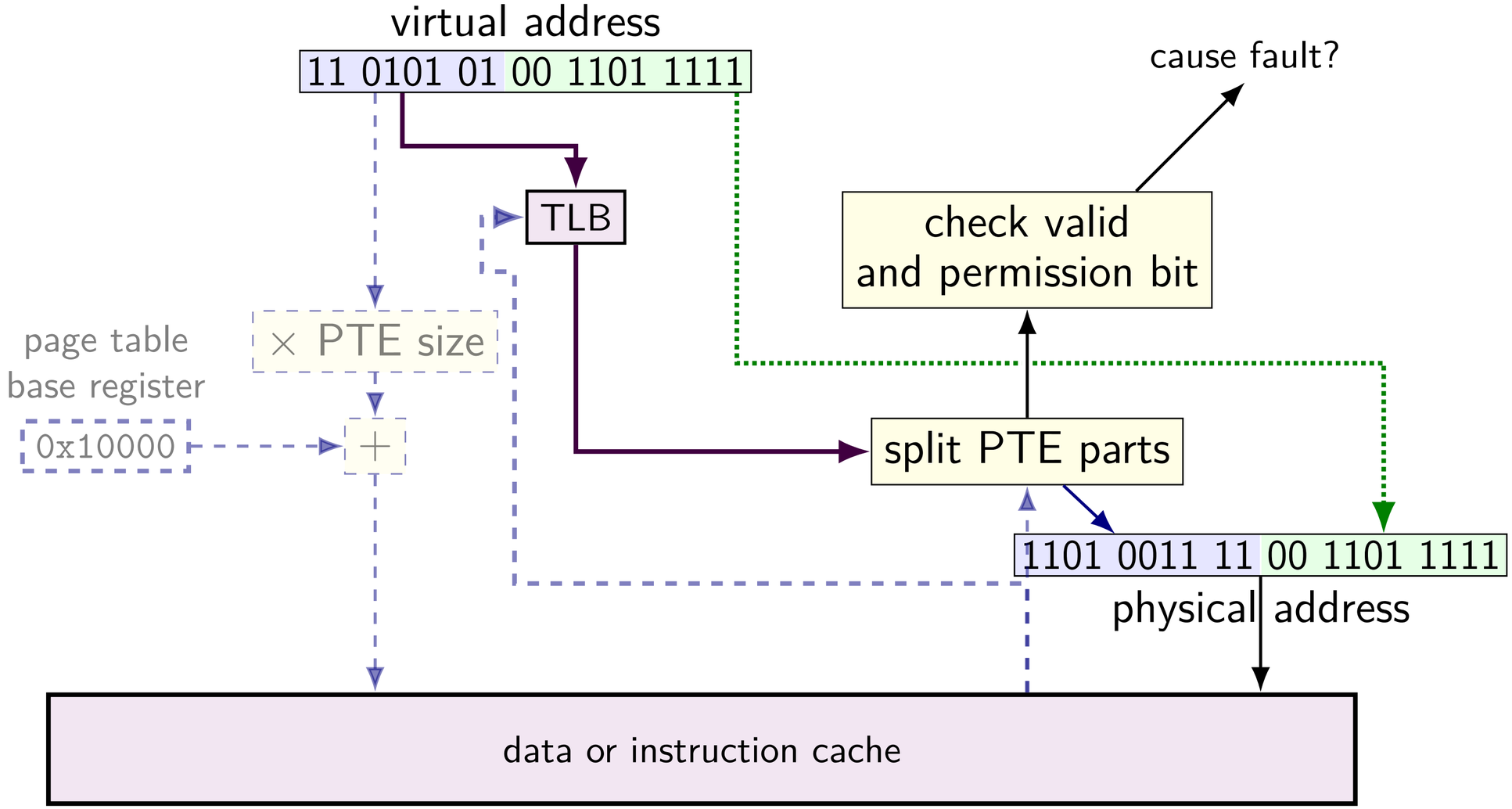
- fortunately, sequential access guesses almost always right

TLB and the MMU (1)

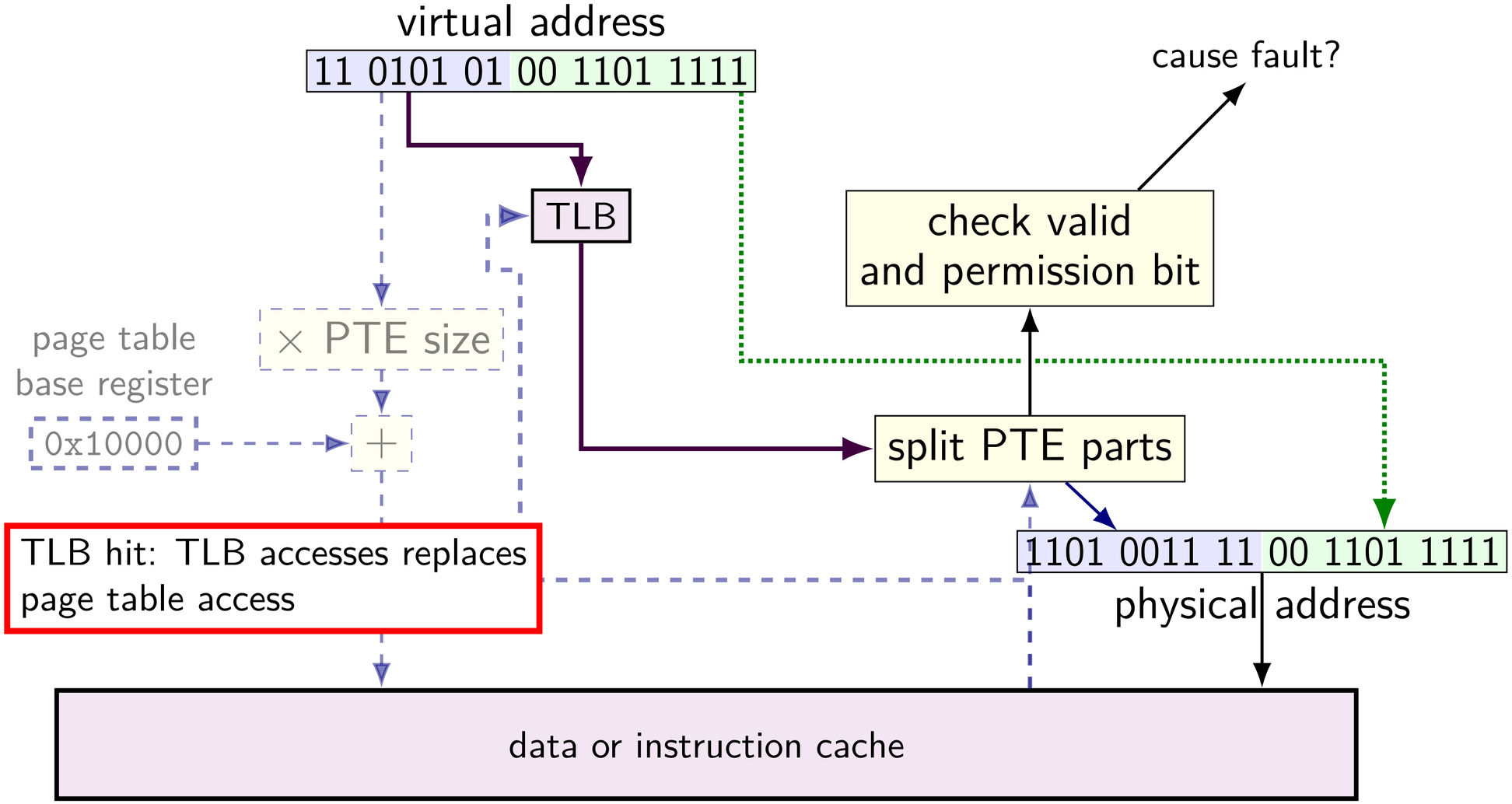


TLB and the MMU (2)

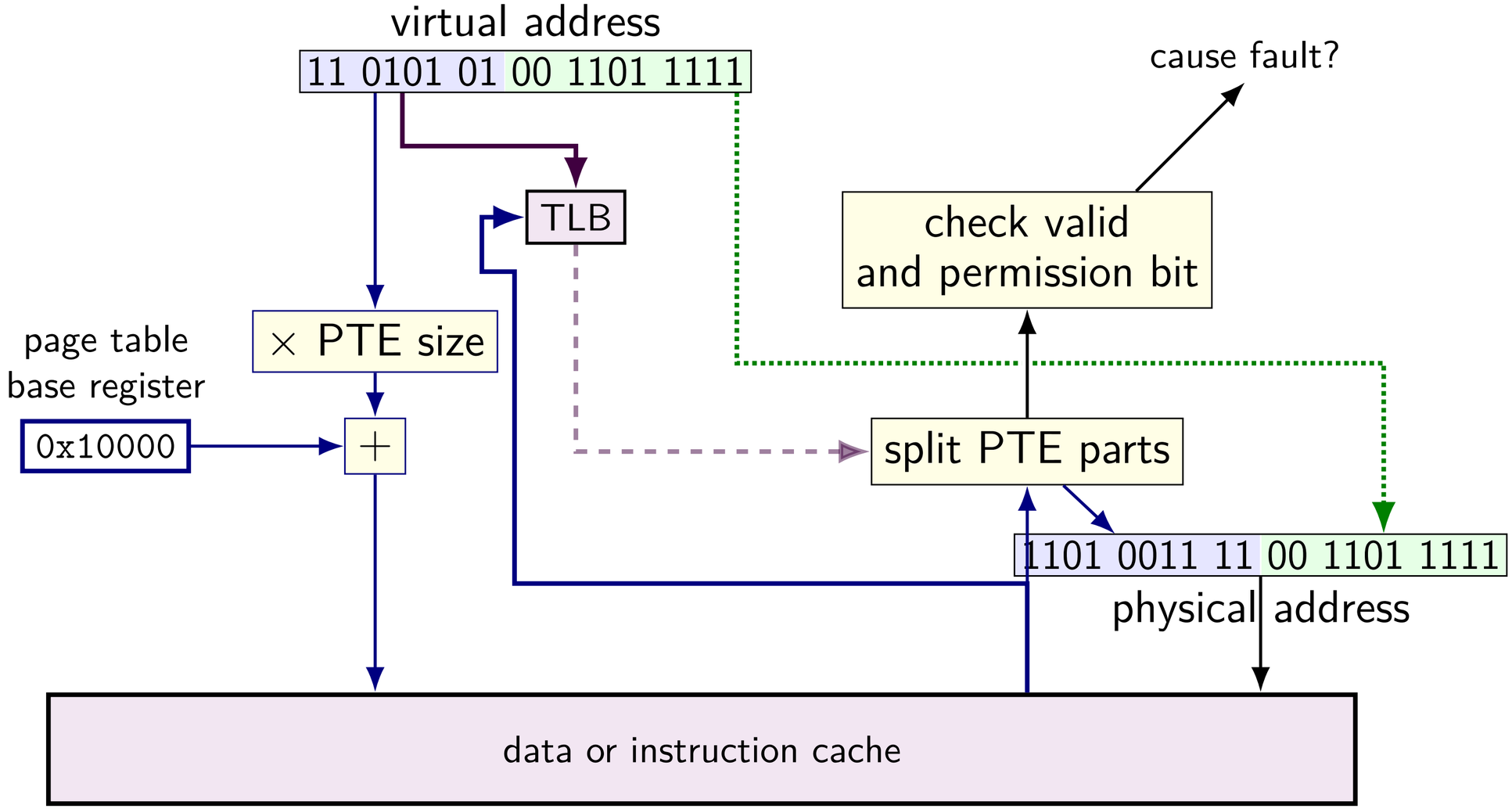
TLB and the MMU (2)



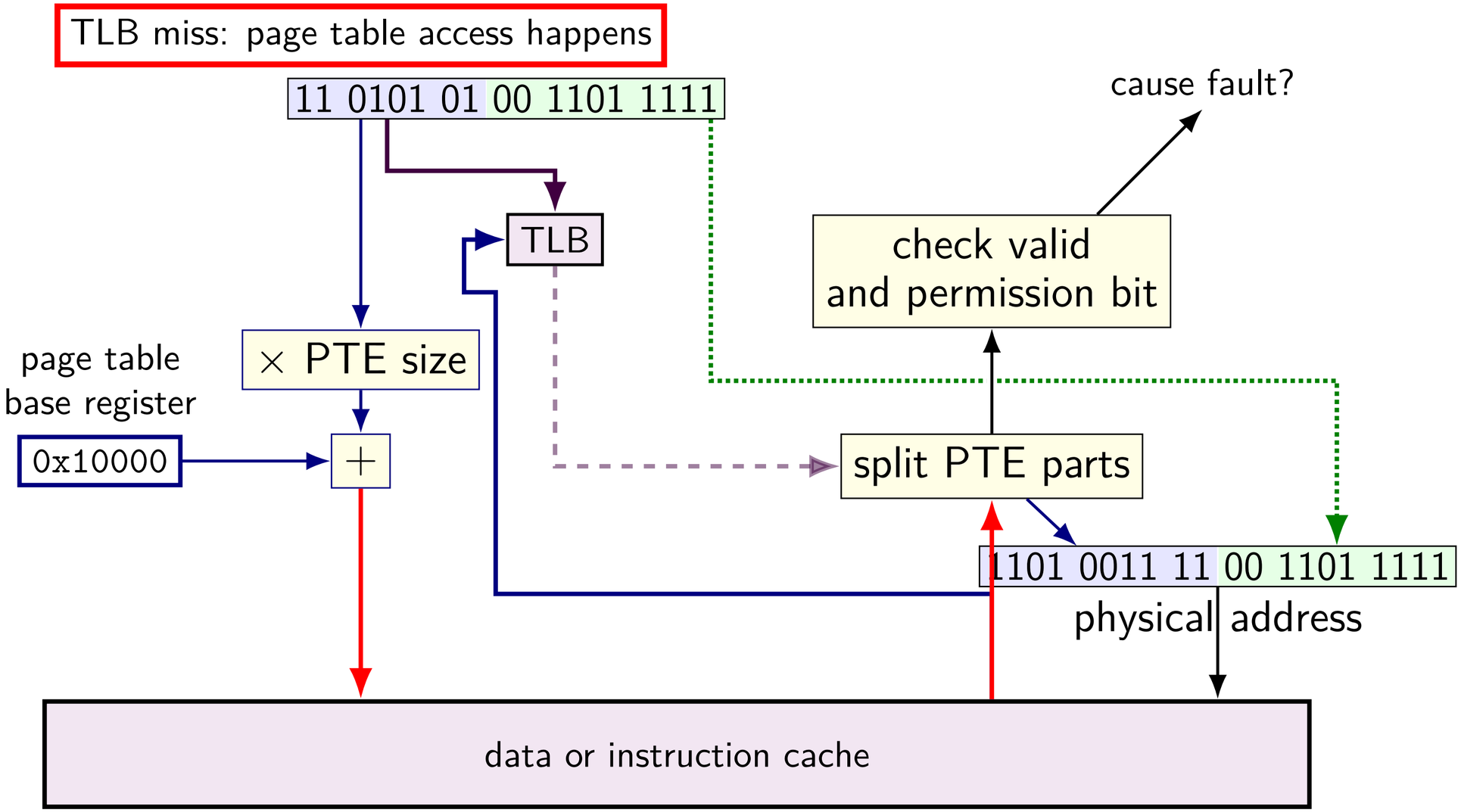
TLB and the MMU (2)



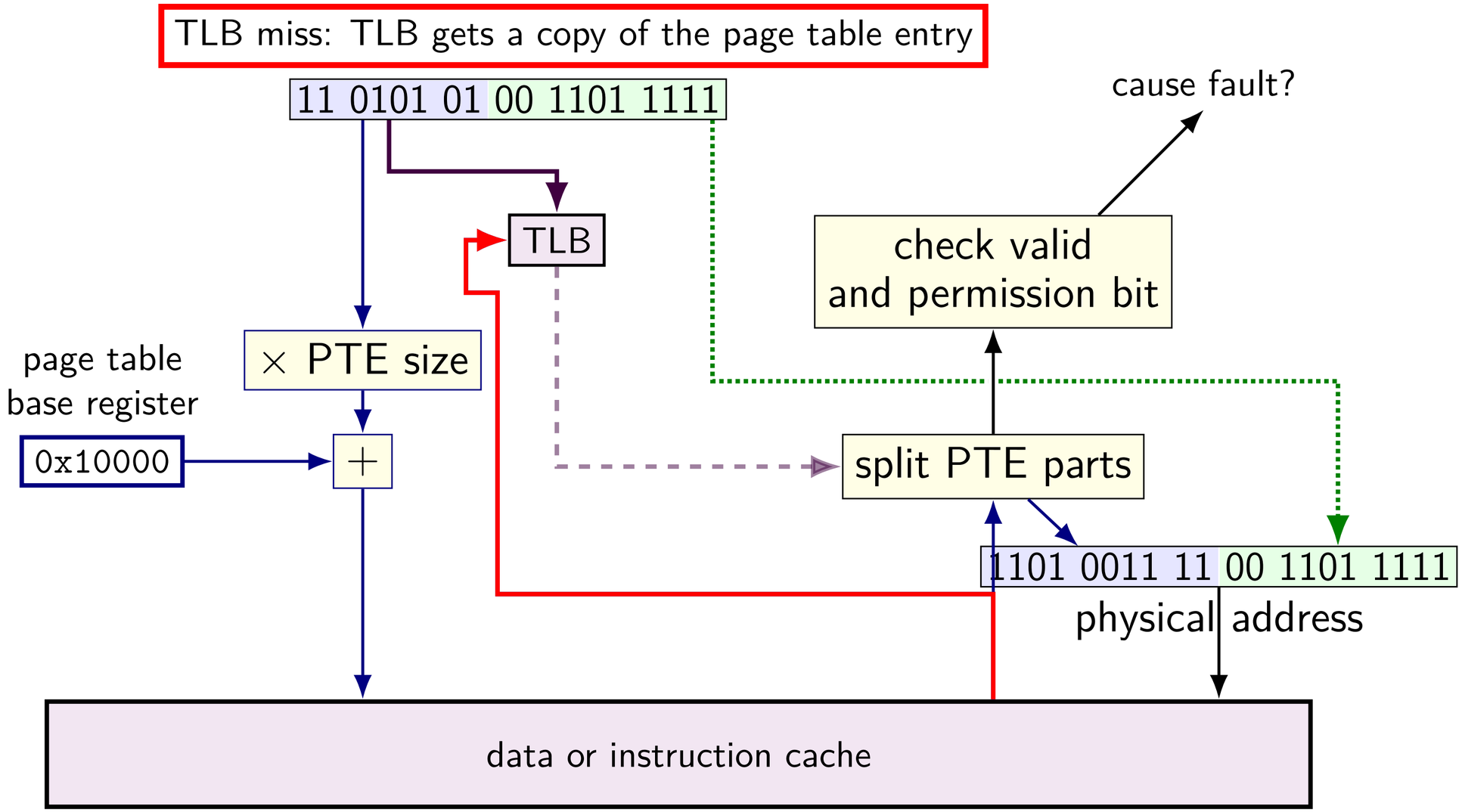
TLB and the MMU (2)



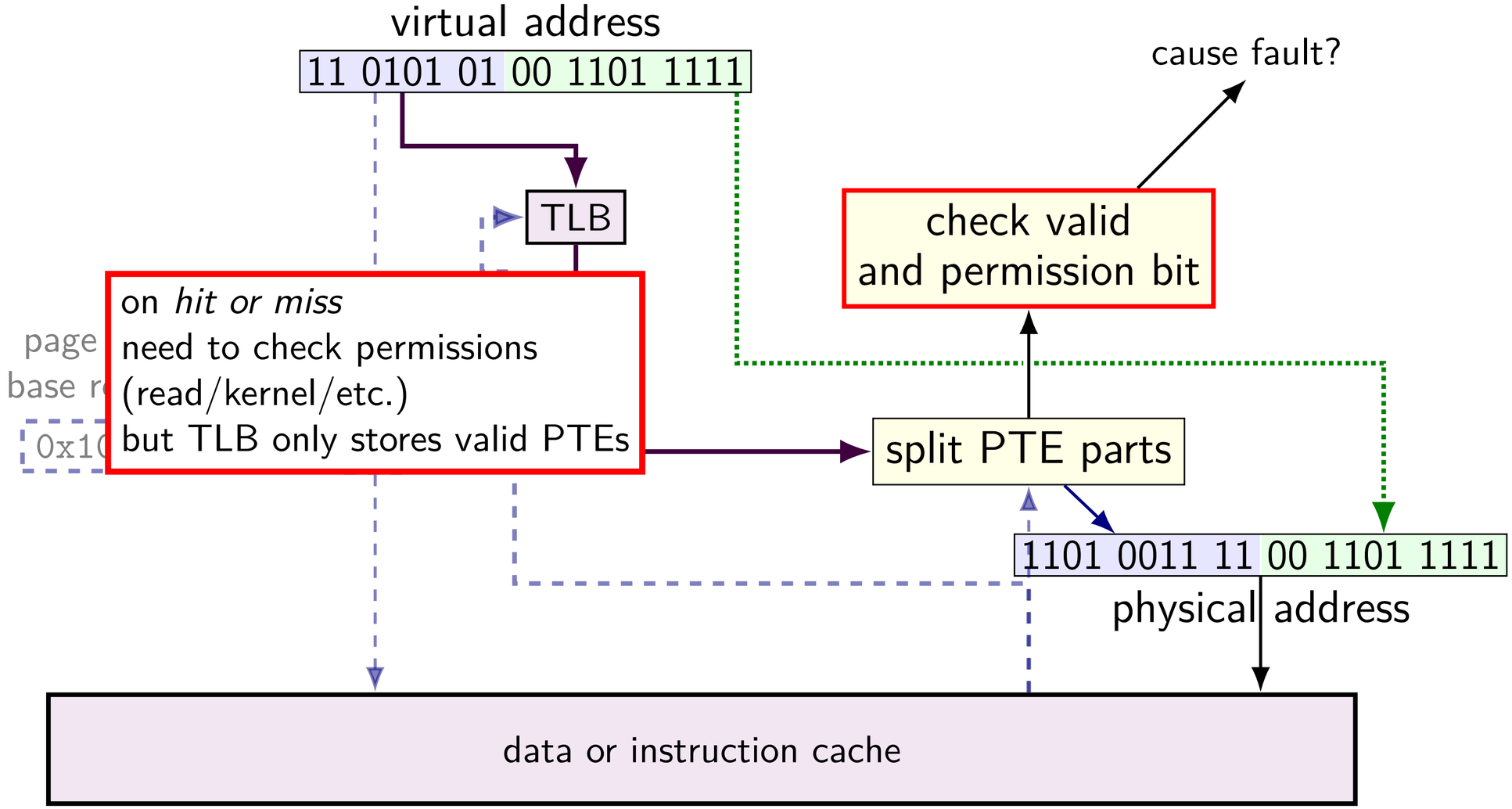
TLB and the MMU (2)



TLB and the MMU (2)



TLB and the MMU (2)



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*

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

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?

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?

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

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*

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