

changelog

27 Feb 2024: quiz Q4: correct swapped 1001/0000 in representing 0x90 as binary and resulting error in overall PTE value on slide

last time

multi-level page tables

- each level: single page table

- index = *part* of virtual page number

- use virtual page number from MSB to LSB

- check valid bits at each level

- later levels omitted if there would be no valid entries

quiz Q1B

OS needed to allocate whole pages of heap

allocated 0x500-0x5ff (virtual page 5) and 0x600-0x6ff (virtual page 6)

quiz Q2

child: got private copy of page 0x3 (0x300–0x3ff), 0x4, 0xe

parent: got private copy page 0xe

B: write to page 0x5 (0x500) — need private copy

D: can avoid copying 0xe twice by recognizing it's no longer shared

quiz Q4

$0x123450 = \text{virtual page } 0x12, \text{ page offset } 0x3450$

$0xA0000 + 0x12 \times 8 \text{ bytes per PTE} = 0xA0090$

quiz Q5

0x903450 = physical page 0x90, page offset 0x3450

page table entry for this:

24 unused bits (must be 0) = 0

24-bit physical page = 0x90

11 unused bits (must be 0) = 0

user-mode-accessible bit = 1

readable bit = 1

writable bit = ?

executable bit = ?

valid bit = 1

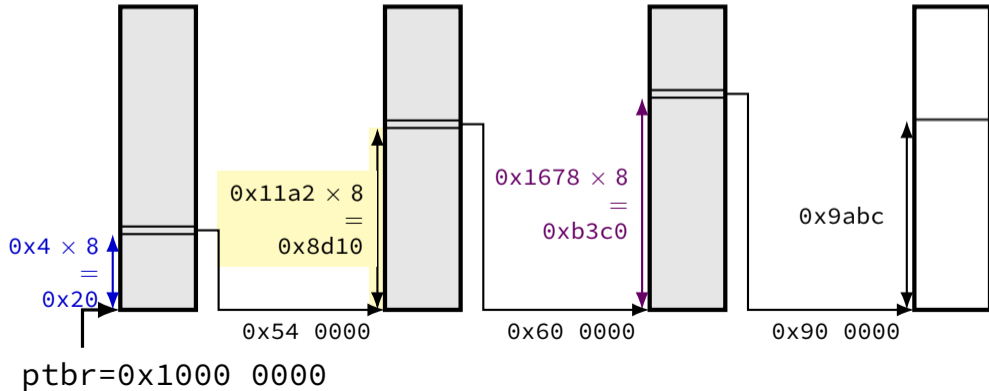
0000 ... 0000 0000 0000 0000 0000 1001 0000 0000 0000 000 1 1??1

0x 90 00 1 (9 or B or D or F)

quiz Q6

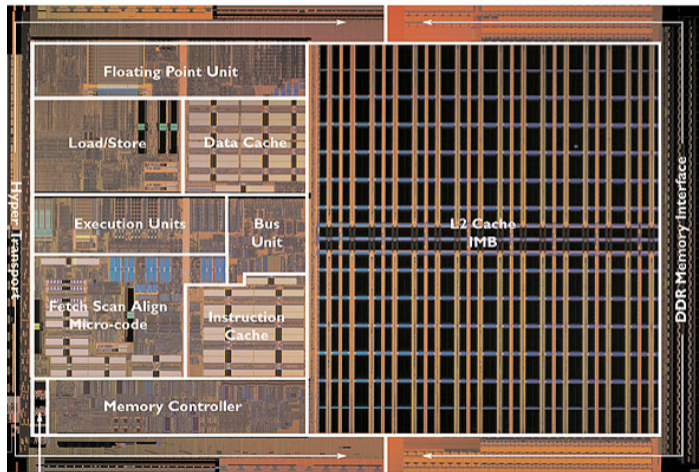
0x00123456789ABC

0000 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100
VPN part 1 VPN part 2 VPN part 3 page offset
0x4 0x11a2 0x1678



lab this week

2004 CPU



Clock Generator



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

2004 CPU

▲ Registers

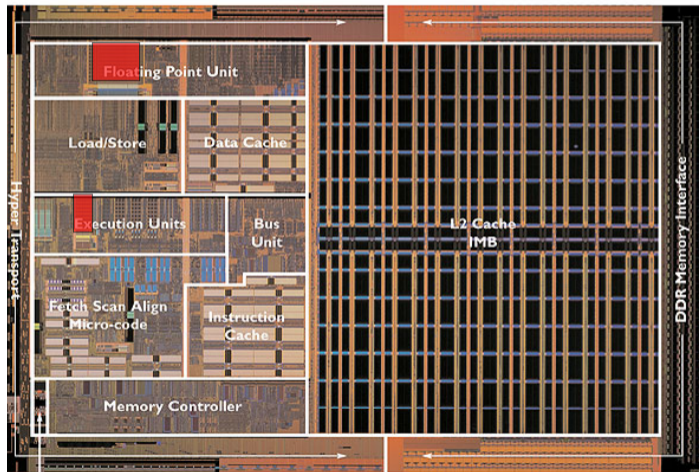


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU

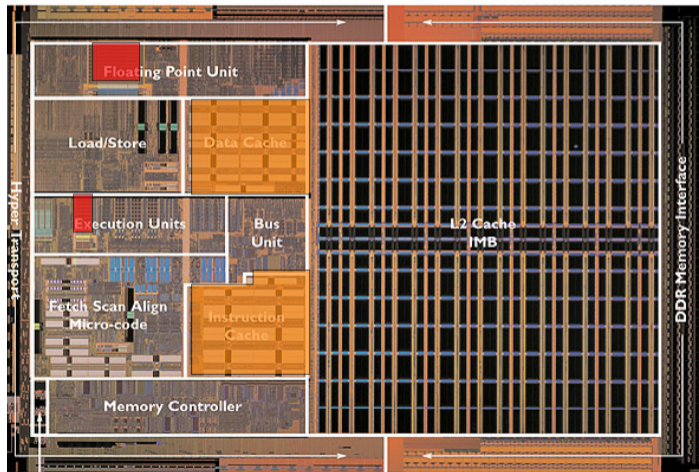


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU

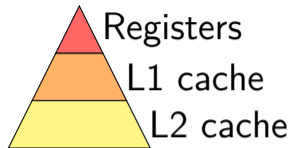
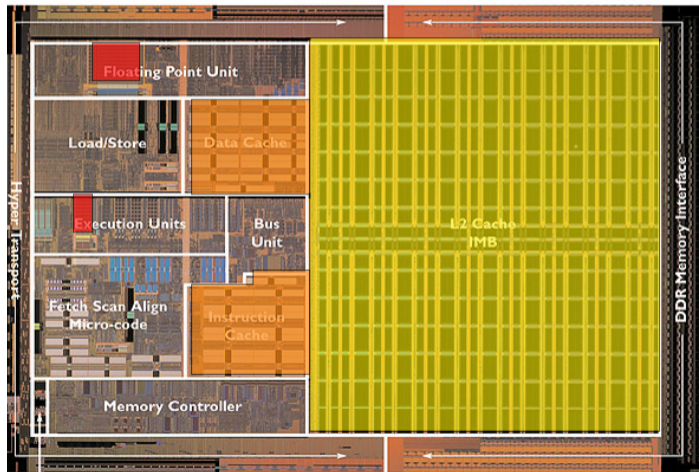


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU

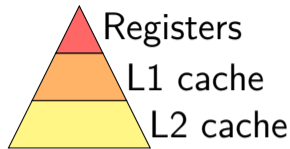
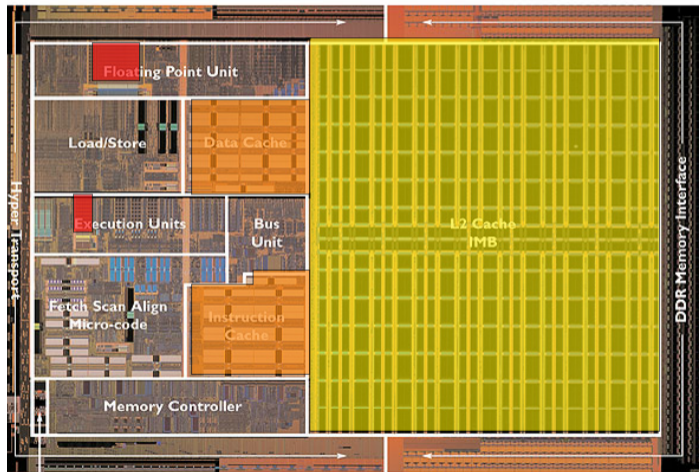
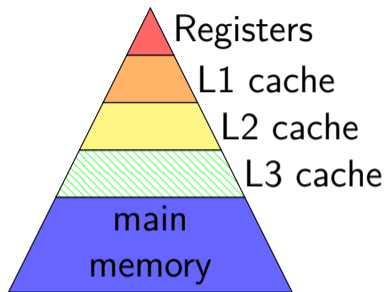
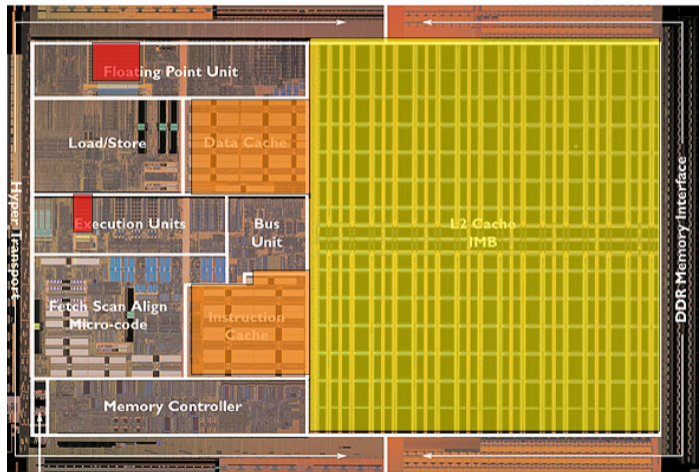
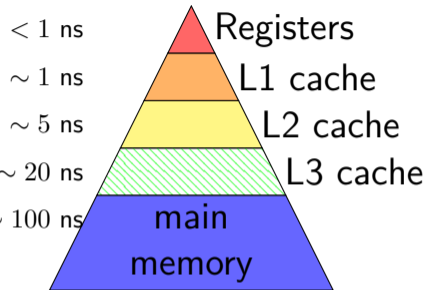
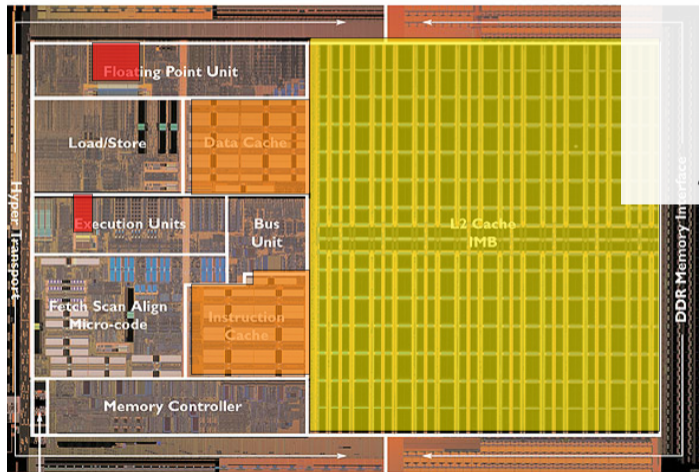


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU



2004 CPU

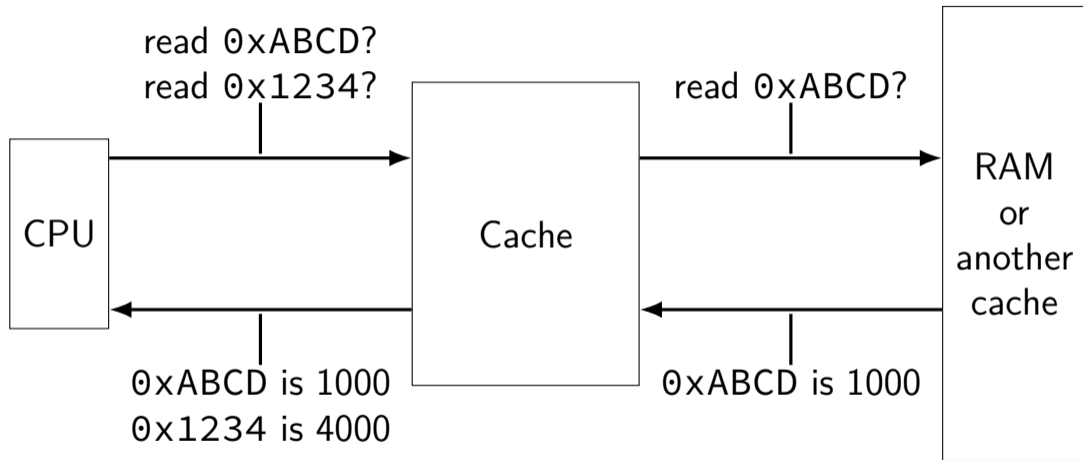


Clock Generator



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

the place of cache (1)



memory hierarchy goals

performance of the fastest (smallest) memory

hide 100x latency difference? 99+% hit (= value found in cache) rate

capacity of the largest (slowest) memory

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;  
}
```

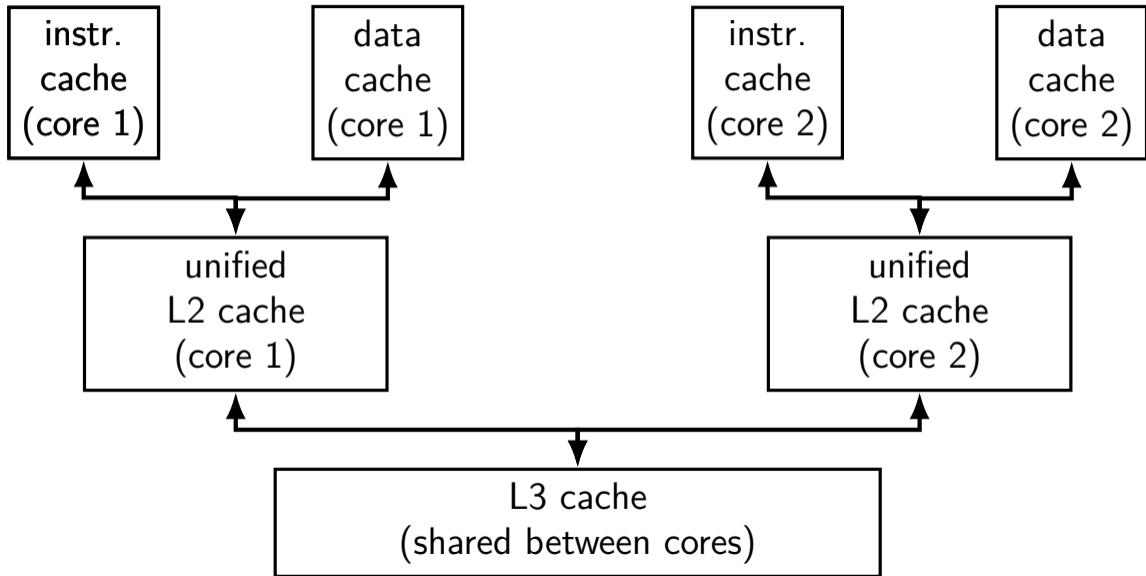
temporal locality: machine code of the loop

spatial locality: machine code of most consecutive instructions

temporal locality: total, i, length accessed repeatedly

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

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: that handles less accesses at a time

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	Memory	
value	addresses	bytes
00 00	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

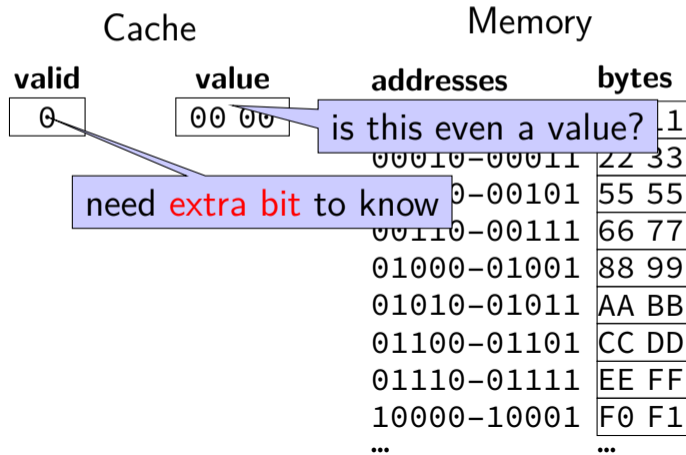
value
00 00

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

Cache

valid

1

value

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

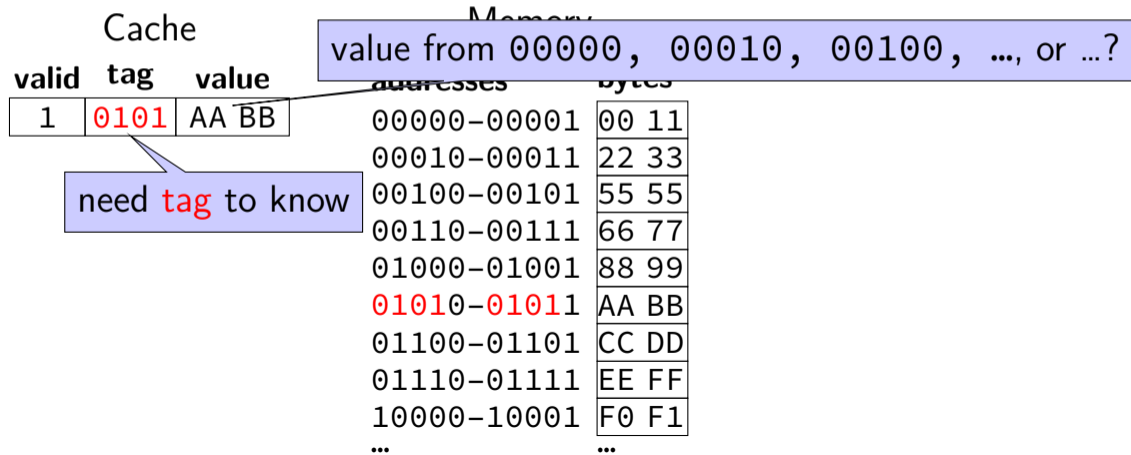
10000-10001 F0 F1

...

...

one-block cache

read byte at 01011?



one-block cache

read byte at 01011?

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
...	...

one-block cache

read byte at 01011?

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
...	...

one-block cache

read byte at 01011?

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

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

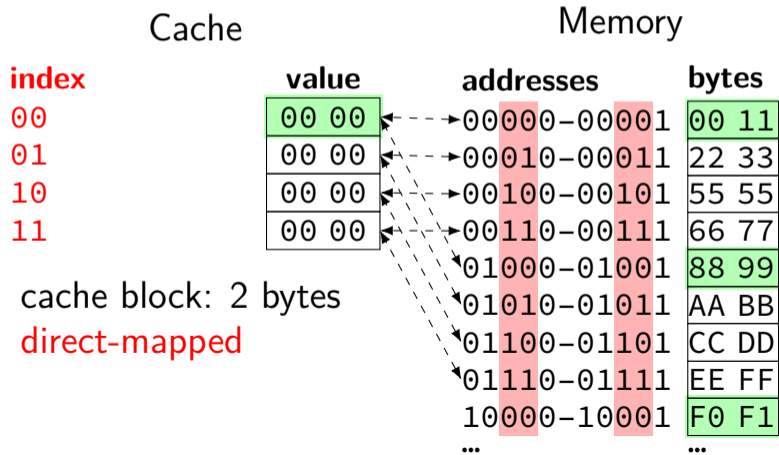
Memory

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?

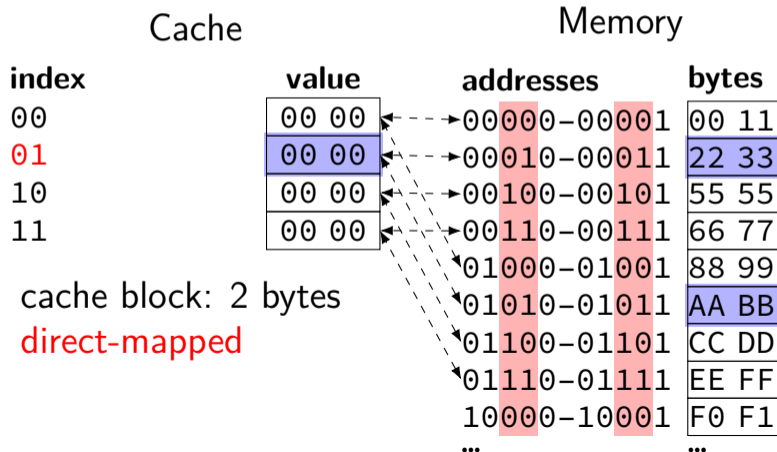
exactly **one place** for each address
spread out what can go in a block



building a (direct-mapped) cache

read byte at 01011?

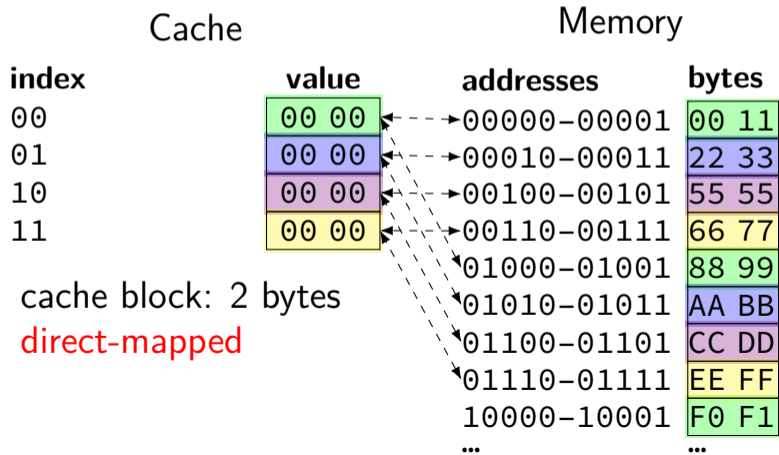
exactly **one place** for each address
spread out what can go in a block



building a (direct-mapped) cache

read byte at 01011?

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



building a (direct-mapped) cache

read byte at 01011?

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

cache block: 2 bytes
direct-mapped

is this even a value?

need extra bit to know

building a (direct-mapped) cache

read byte at 01011?

invalid, fetch

Cache

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

cache block: 2 bytes

direct-mapped

Memory

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

building a (direct-mapped) cache

read byte at 01011?

invalid, fetch

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

value from 01010 or 00010?

need tag to know

cache block: 2 bytes
direct-mapped

building a (direct-mapped) cache

read byte at 01011?

invalid, fetch

Cache

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

cache block: 2 bytes

direct-mapped

Memory

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

...

...

terminology

row = set

preview: change how much is in a row

Tag-Index-Offset (TIO)

address 001111 (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		

4 byte

4 = 2² bytes in block
2 bits to say which byte

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
	0	--	-- --
	0	--	-- --
	0	--	-- --
	1	00	AA BB
	0	--	-- --
	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

2 byte blocks, 8 sets

index	valid	tag	value
000	1	00	00 11
			F1 F2

100	0	--	----
101	1	00	AA BB
110	0	--	----
111	1	00	EE FF

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

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 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		

$2^3 = 8$ sets
3 bits to index set

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 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	0	--	----
101	0	--	----
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

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 addresses 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 addresses 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

backup slides

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?

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];
```

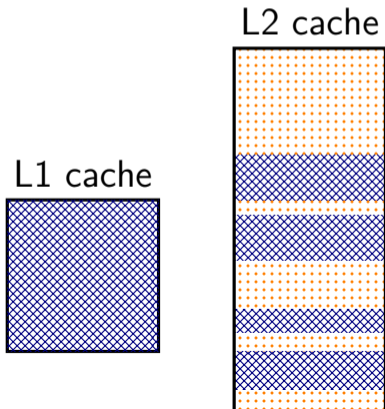
Assume everything but array is kept in registers (and the compiler does not do anything funny).

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

inclusive versus exclusive

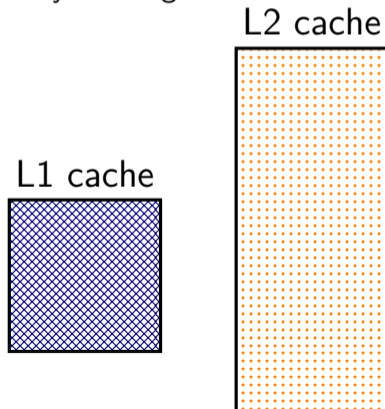
L2 inclusive of L1

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



L2 exclusive of L1

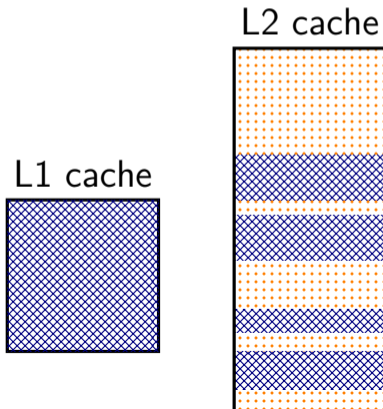
L2 contains different data than L1
adding to L1 must remove from L2
probably evicting from L1 adds to L2



inclusive versus exclusive

L2 inclusive of L1

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



L2 exclusive of L1

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

inclusive policy:

no extra work on eviction
but duplicated data

easier to explain when

L_k shared by multiple $L(k-1)$ caches?

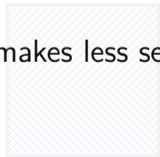
inclusive versus exclusive

L2 inclusive of L1

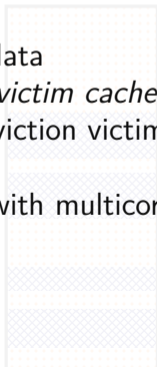
everything in L1 cache duplicated in L2
adding to L1 also adds to L2

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

makes less sense with multicore



L2 cache



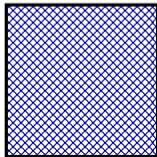
L2 exclusive of L1

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

L2 cache



L1 cache



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 addresses 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 addresses bits

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

$C = B \times S$ **cache size** (if direct-mapped)

cache organization and miss rate

depends on program; one example:

SPEC CPU2000 benchmarks, 64B block size

LRU replacement policies

data cache miss rates:

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

cache organization and miss rate

depends on program; one example:

SPEC CPU2000 benchmarks, 64B block size

LRU replacement policies

data cache miss rates:

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

exercise (1)

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

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

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

exercise (2)

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

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

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

exercise (3)

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

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

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

prefetching

seems like we can't really improve cold misses...

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

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 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

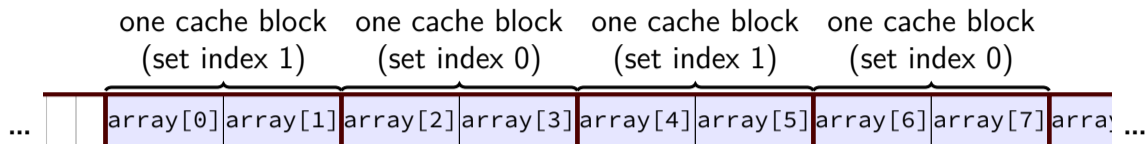
bring in guess at next-to-be-accessed value

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

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

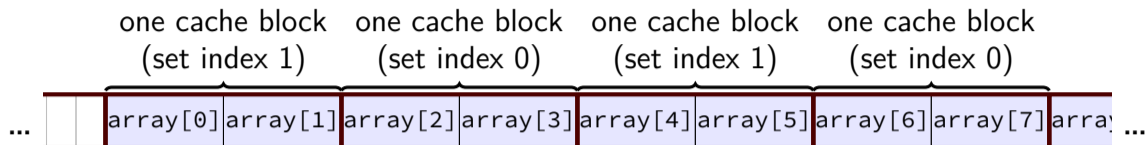
fortunately, sequential access guesses almost always right

quiz exercise solution



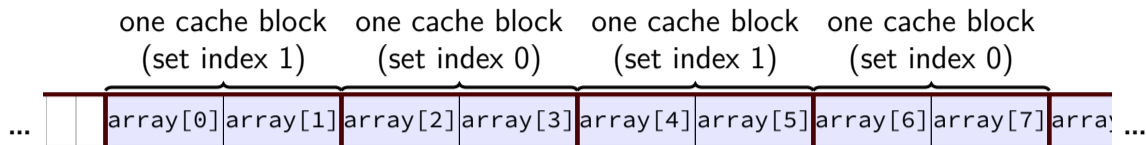
memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[3] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[6] (miss)	{array[0], array[1]}	{array[6], array[7]}
read array[1] (hit)	{array[0], array[1]}	{array[6], array[7]}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

quiz exercise solution



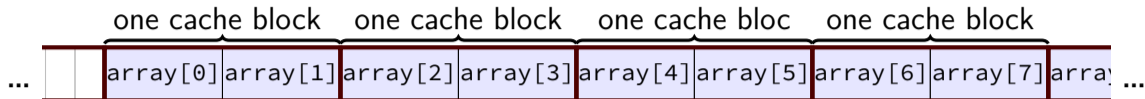
memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[3] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[6] (miss)	{array[0], array[1]}	{array[6], array[7]}
read array[1] (hit)	{array[0], array[1]}	{array[6], array[7]}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

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]}

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

C and cache misses (4, solution pt 1)

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

64B = 16 ints stored per way

4 sets total

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

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

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

C and cache misses (4, solution pt 2)

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

6 misses for set 0

C and cache misses (4, solution pt 3)

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

C and cache misses (3)

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

observation: 12 ints in struct: only first two used

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

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

C and cache misses (3, rewritten?)

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

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

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

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

observation 2: `array[0]` and `array[1]`, `array[12]` and `array[13]`, etc. in 41

C and cache misses (3, solution)

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

64B = 16 ints stored per way

4 sets total

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

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

set 0 contains block with array[0 to 3]

set 5 contains block with array[20 to 23]

etc.

C and cache misses (3, solution)

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

64B = 16 ints stored per way

4 sets total

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

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

set 0 contains block with array[0 to 3]

set 5 contains block with array[20 to 23]

etc.

C and cache misses (3, solution)

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

64B = 16 ints stored per way

4 sets total

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

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

each set used at most twice

no replacement needed

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

set 0 contains block with array[0 to 3]

set 5 contains block with array[20 to 23]

etc.

C and cache misses (3)

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

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

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

C and cache misses (3, rewritten?)

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

C and cache misses (4)

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

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

How many *data cache misses* on a 4-way set associative 2KB direct-mapped cache with 16B cache blocks?

thinking about cache storage (1)

2KB direct-mapped cache with 16B blocks —

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

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

...

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

thinking about cache storage (1)

2KB direct-mapped cache with 16B blocks —

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

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

...

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

thinking about cache storage (1)

2KB direct-mapped cache with 16B blocks —

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

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

...

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

thinking about cache storage (1)

2KB direct-mapped cache with 16B blocks —

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

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

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

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

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

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

...

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

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

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

thinking about cache storage (2)

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

—

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

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

...

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

thinking about cache storage (2)

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

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

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

...

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

thinking about cache storage (2)

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

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

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

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

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

...

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

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

...

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

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

thinking about cache storage (2)

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

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

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

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

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

...

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

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

...

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

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

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 \times 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 \times 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?

two rows of matrix?

often `sizeof(row)` bytes apart

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

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

My desktop:

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

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

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

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

T-I-O exercise: L1

quantity

value for L1

block size (given)

$B = 64\text{Byte}$

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

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64\text{Byte}$
	$B = 2^b$ (b : block offset bits)
block offset bits	$b = 6$

T-I-O exercise: L1

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

T-I-O exercise: L1

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

T-I-O exercise: L1

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

T-I-O exercise: L1

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

T-I-O results

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

T-I-O: splitting

	L1	L2	L3		
block offset bits	6	6	6		
set index bits	6	10	13		
tag bits	(the rest)				
0x34567:	3	4	5	6	7
	0011	0100	0101	0110	0111
bits 0-5 (all offsets):	100111 = 0x27				

T-I-O: splitting

	L1	L2	L3		
block offset bits	6	6	6		
set index bits	6	10	13		
tag bits	(the rest)				
0x34567:	3	4	5	6	7
	0011	0100	0101	0110	0111
bits 0-5 (all offsets):	100111 = 0x27				

T-I-O: splitting

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

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

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

L1:

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

bits 12- (L1 tag): 0x34

T-I-O: splitting

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

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

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

L1:

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

bits 12- (L1 tag): 0x34

T-I-O: splitting

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

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

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

L2:

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

bits 16-: 0x3

T-I-O: splitting

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

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

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

L2:

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

bits 16-: 0x3

T-I-O: splitting

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

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

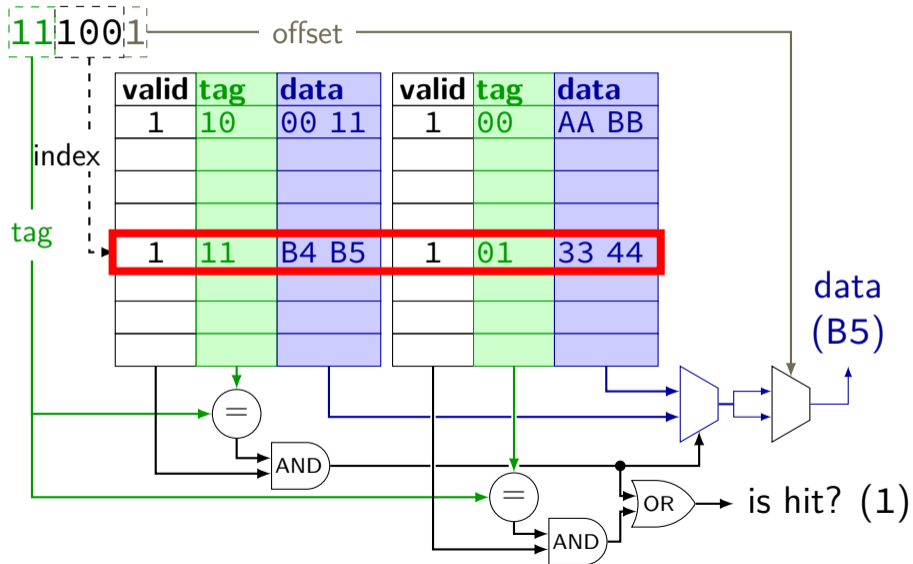
bits 0-5 (all offsets): 100111 = 0x27

L3:

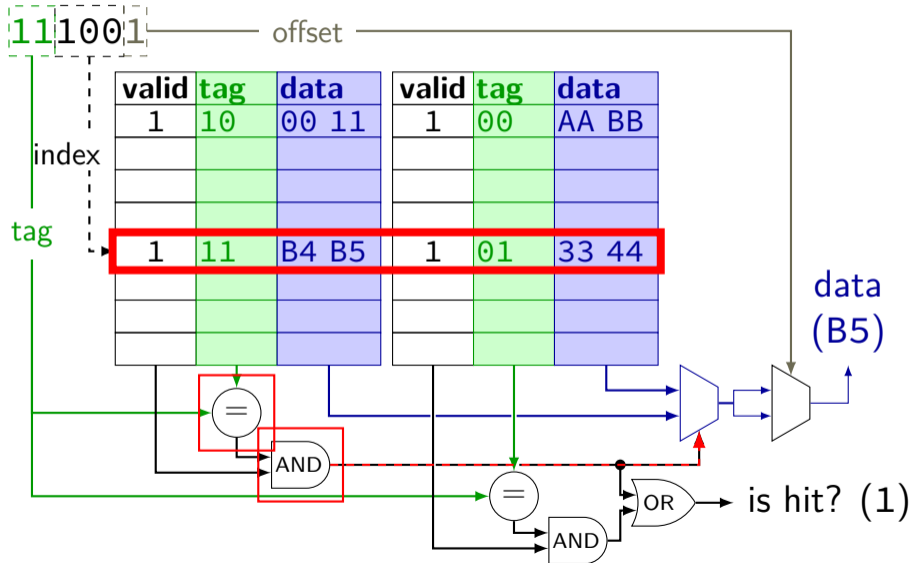
bits 6-18 (set for L3): 0 1101 0001 0101 = 0xD15

bits 18-: 0x0

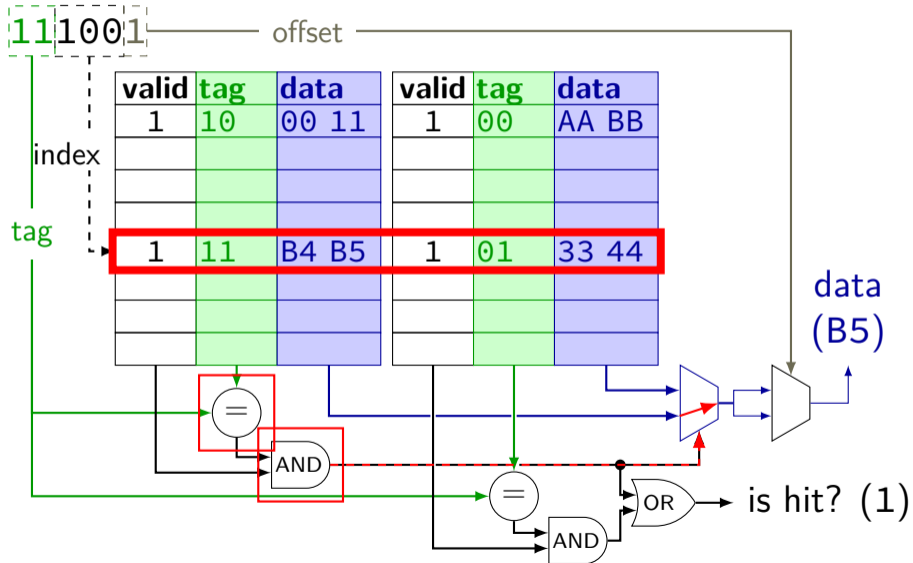
cache operation (associative)



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-associative (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

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

but — typical real programs have **locality**

cache optimizations

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

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

prefetching = guess what program will use, access in advance

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

cache optimizations by miss type

(assuming other listed parameters remain constant)

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

average memory access time

AMAT = hit time + miss penalty \times miss rate

or AMAT = hit time \times hit rate + miss time \times 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?

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?

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?

approximate miss analysis

very tedious to precisely count cache misses

even more tedious when we take advanced cache optimizations into account

instead, approximations:

good or bad temporal/spatial locality

good temporal locality: value stays in cache

good spatial locality: use all parts of cache block

with nested loops: what does inner loop use?

intuition: values used in inner loop loaded into cache once
(that is, once each time the inner loop is run)

...if they can all fit in the cache

approximate miss analysis

very tedious to precisely count cache misses

even more tedious when we take advanced cache optimizations into account

instead, approximations:

good or bad temporal/spatial locality

good temporal locality: value stays in cache

good spatial locality: use all parts of cache block

with nested loops: what does inner loop use?

intuition: values used in inner loop loaded into cache once
(that is, once each time the inner loop is run)

...if they can all fit in the cache

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?

exercise: miss estimating (1)

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

Assume: 4 array elements per block, N very large, nothing in cache at beginning.

Example: $N/4$ estimated misses for A accesses:

$A[i]$ should always be hit on all but first iteration of inner-most loop.
first iter: $A[i]$ should be hit about $3/4$ s of the time (same block as $A[i-1]$ that often)

Exercise: estimate # of misses for B , C

a note on matrix storage

A — $N \times N$ matrix

represent as **array**

makes dynamic sizes easier:

```
float A_2d_array[N][N];  
float *A_flat = malloc(N * N);
```

```
A_flat[i * N + j] == A_2d_array[i][j]
```

conversion re: rows/columns

going to call the first index rows

$A_{i,j}$ is A row i, column j

rows are stored together

this is an arbitrary choice

5x5 array and 4-element cache blocks

array[0*5 + 0]	array[0*5 + 1]	array[0*5 + 2]	array[0*5 + 3]	array[0*5 + 4]
array[1*5 + 0]	array[1*5 + 1]	array[1*5 + 2]	array[1*5 + 3]	array[1*5 + 4]
array[2*5 + 0]	array[2*5 + 1]	array[2*5 + 2]	array[2*5 + 3]	array[2*5 + 4]
array[3*5 + 0]	array[3*5 + 1]	array[3*5 + 2]	array[3*5 + 3]	array[3*5 + 4]
array[4*5 + 0]	array[4*5 + 1]	array[4*5 + 2]	array[4*5 + 3]	array[4*5 + 4]

5x5 array and 4-element cache blocks

array[0*5 + 0]	array[0*5 + 1]	array[0*5 + 2]	array[0*5 + 3]	array[0*5 + 4]
array[1*5 + 0]	array[1*5 + 1]	array[1*5 + 2]	array[1*5 + 3]	array[1*5 + 4]
array[2*5 + 0]	array[2*5 + 1]	array[2*5 + 2]	array[2*5 + 3]	array[2*5 + 4]
array[3*5 + 0]	array[3*5 + 1]	array[3*5 + 2]	array[3*5 + 3]	array[3*5 + 4]
array[4*5 + 0]	array[4*5 + 1]	array[4*5 + 2]	array[4*5 + 3]	array[4*5 + 4]

if array starts on cache block
first cache block = first elements
all together in one row!

5x5 array and 4-element cache blocks

array[0*5 + 0]	array[0*5 + 1]	array[0*5 + 2]	array[0*5 + 3]	array[0*5 + 4]
array[1*5 + 0]	array[1*5 + 1]	array[1*5 + 2]	array[1*5 + 3]	array[1*5 + 4]
array[2*5 + 0]	array[2*5 + 1]	array[2*5 + 2]	array[2*5 + 3]	array[2*5 + 4]
array[3*5 + 0]	array[3*5 + 1]	array[3*5 + 2]	array[3*5 + 3]	array[3*5 + 4]
array[4*5 + 0]	array[4*5 + 1]	array[4*5 + 2]	array[4*5 + 3]	array[4*5 + 4]

second cache block:

1 from row 0

3 from row 1

5x5 array and 4-element cache blocks

array[0*5 + 0]	array[0*5 + 1]	array[0*5 + 2]	array[0*5 + 3]	array[0*5 + 4]
array[1*5 + 0]	array[1*5 + 1]	array[1*5 + 2]	array[1*5 + 3]	array[1*5 + 4]
array[2*5 + 0]	array[2*5 + 1]	array[2*5 + 2]	array[2*5 + 3]	array[2*5 + 4]
array[3*5 + 0]	array[3*5 + 1]	array[3*5 + 2]	array[3*5 + 3]	array[3*5 + 4]
array[4*5 + 0]	array[4*5 + 1]	array[4*5 + 2]	array[4*5 + 3]	array[4*5 + 4]

5x5 array and 4-element cache blocks

array[0*5 + 0]	array[0*5 + 1]	array[0*5 + 2]	array[0*5 + 3]	array[0*5 + 4]
array[1*5 + 0]	array[1*5 + 1]	array[1*5 + 2]	array[1*5 + 3]	array[1*5 + 4]
array[2*5 + 0]	array[2*5 + 1]	array[2*5 + 2]	array[2*5 + 3]	array[2*5 + 4]
array[3*5 + 0]	array[3*5 + 1]	array[3*5 + 2]	array[3*5 + 3]	array[3*5 + 4]
array[4*5 + 0]	array[4*5 + 1]	array[4*5 + 2]	array[4*5 + 3]	array[4*5 + 4]

generally: cache blocks contain data from 1 or 2 rows
→ better performance from reusing rows

matrix multiply

$$C_{ij} = \sum_{k=1}^n A_{ik} \times B_{kj}$$

```
/* version 1: inner loop is k, middle is j */  
for (int i = 0; i < N; ++i)  
    for (int j = 0; j < N; ++j)  
        for (int k = 0; k < N; ++k)  
            C[i * N + j] += A[i * N + k] * B[k * N + j];
```

matrix multiply

$$C_{ij} = \sum_{k=1}^n A_{ik} \times B_{kj}$$

```
/* version 1: inner loop is k, middle is j */  
for (int i = 0; i < N; ++i)  
    for (int j = 0; j < N; ++j)  
        for (int k = 0; k < N; ++k)  
            C[i*N+j] += A[i * N + k] * B[k * N + j];
```

```
/* version 2: outer loop is k, middle is i */  
for (int k = 0; k < N; ++k)  
    for (int i = 0; i < N; ++i)  
        for (int j = 0; j < N; ++j)  
            C[i*N+j] += A[i * N + k] * B[k * N + j];
```

loop orders and locality

loop body: $C_{ij} += A_{ik}B_{kj}$

kij order: C_{ij} , B_{kj} have **spatial locality**

kij order: A_{ik} has **temporal locality**

... better than ...

ijk order: A_{ik} has spatial locality

ijk order: C_{ij} has temporal locality

loop orders and locality

loop body: $C_{ij} += A_{ik}B_{kj}$

kij order: C_{ij} , B_{kj} have spatial locality

kij order: A_{ik} has temporal locality

... better than ...

ijk order: A_{ik} has spatial locality

ijk order: C_{ij} has temporal locality

matrix multiply

$$C_{ij} = \sum_{k=1}^n A_{ik} \times B_{kj}$$

```
/* version 1: inner loop is k, middle is j */  
for (int i = 0; i < N; ++i)  
    for (int j = 0; j < N; ++j)  
        for (int k = 0; k < N; ++k)  
            C[i*N+j] += A[i * N + k] * B[k * N + j];
```

```
/* version 2: outer loop is k, middle is i */  
for (int k = 0; k < N; ++k)  
    for (int i = 0; i < N; ++i)  
        for (int j = 0; j < N; ++j)  
            C[i*N+j] += A[i * N + k] * B[k * N + j];
```

matrix multiply

$$C_{ij} = \sum_{k=1}^n A_{ik} \times B_{kj}$$

```
/* version 1: inner loop is k, middle is j */  
for (int i = 0; i < N; ++i)  
    for (int j = 0; j < N; ++j)  
        for (int k = 0; k < N; ++k)  
            C[i*N+j] += A[i * N + k] * B[k * N + j];
```

```
/* version 2: outer loop is k, middle is i */  
for (int k = 0; k < N; ++k)  
    for (int i = 0; i < N; ++i)  
        for (int j = 0; j < N; ++j)  
            C[i*N+j] += A[i * N + k] * B[k * N + j];
```

matrix multiply

$$C_{ij} = \sum_{k=1}^n A_{ik} \times B_{kj}$$

```
/* version 1: inner loop is k, middle is j */  
for (int i = 0; i < N; ++i)  
    for (int j = 0; j < N; ++j)  
        for (int k = 0; k < N; ++k)  
            C[i*N+j] += A[i * N + k] * B[k * N + j];
```

```
/* version 2: outer loop is k, middle is i */  
for (int k = 0; k < N; ++k)  
    for (int i = 0; i < N; ++i)  
        for (int j = 0; j < N; ++j)  
            C[i*N+j] += A[i * N + k] * B[k * N + j];
```

which is better?

$$C_{ij} = \sum_{k=1}^n A_{ik} \times B_{kj}$$

/ version 1: inner loop is k, middle is j*/*

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

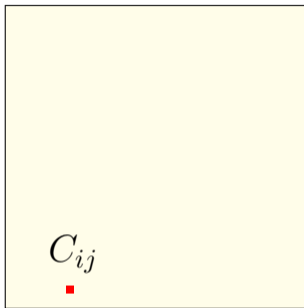
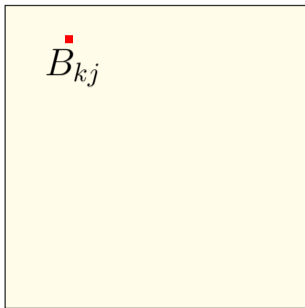
/ version 2: outer loop is k, middle is i */*

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

exercise: Which version has better spatial/temporal locality for...

accesses to C? accesses to A? accesses to B?

array usage: ijk order

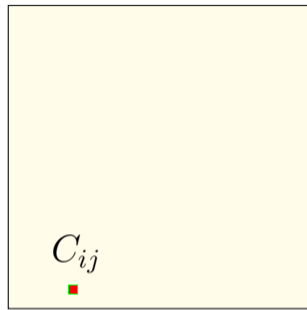
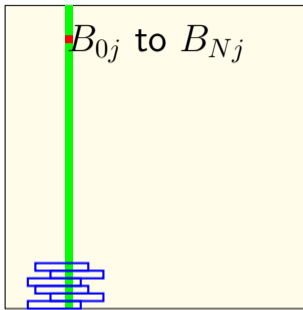
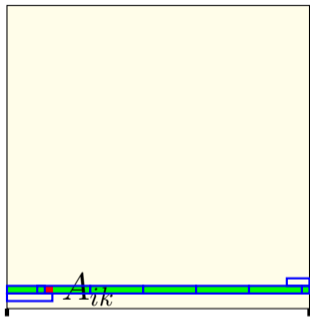


A_{x0} A_{xN}
for all i :
 for all j :
 for all k :
 $C_{ij} += A_{ik} \times B_{kj}$

if N large:

using C_{ij} many times per load into cache
using A_{ik} once per load-into-cache
(but using $A_{i,k+1}$ right after)
using B_{kj} once per load into cache

array usage: ijk order



A_{x0} A_{xN}

for all i :

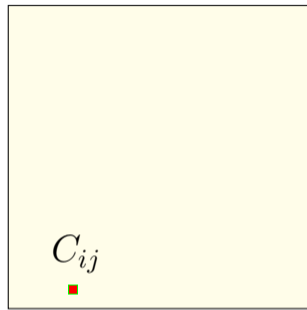
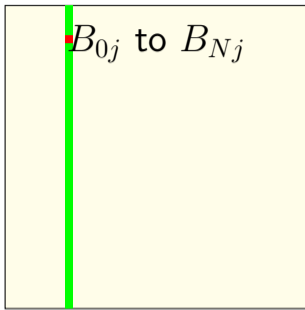
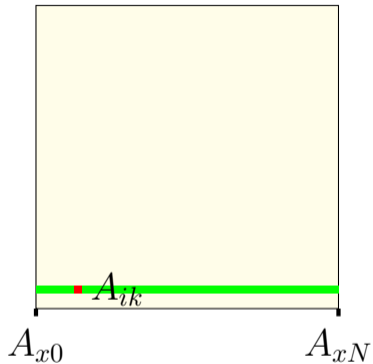
for all j :

for all k :

$$C_{ij} += A_{ik} \times B_{kj}$$

looking only at innermost loop:
good spatial locality in A
(rows stored together = reuse cache blocks)
bad spatial locality in B
(use each cache block once)
no useful spatial locality in C

array usage: ijk order



for all i :

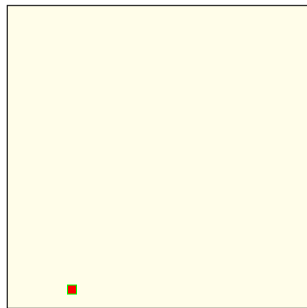
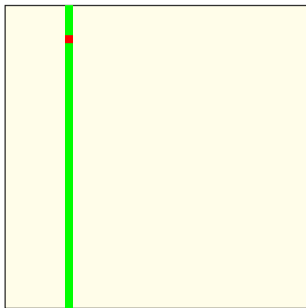
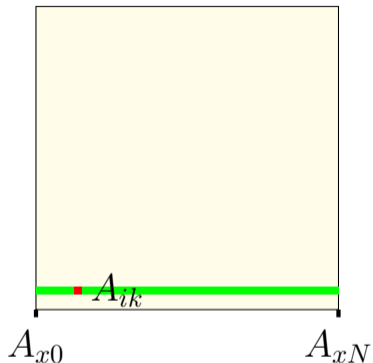
for all j :

for all k :

$$C_{ij} += A_{ik} \times B_{kj}$$

looking only at innermost loop:
temporal locality in C
bad temporal locality in everything else
(everything accessed exactly once)

array usage: *ijk* order



for all i :

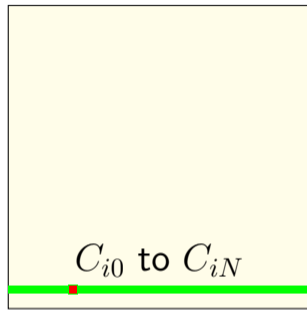
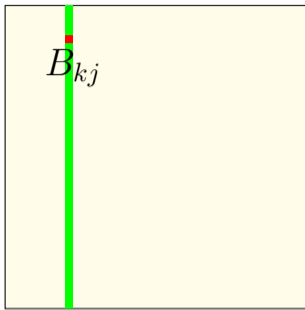
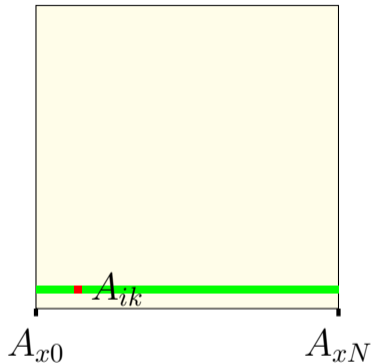
for all j :

for all k :

$$C_{ij} += A_{ik} \times B_{kj}$$

looking only at innermost loop:
row of A (elements used once)
column of B (elements used once)
single element of C (used many times)

array usage: ijk order



for all i :

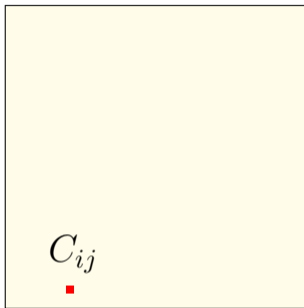
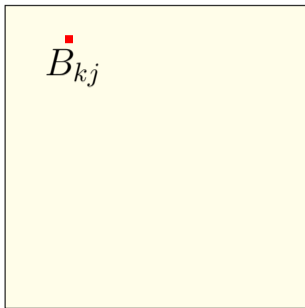
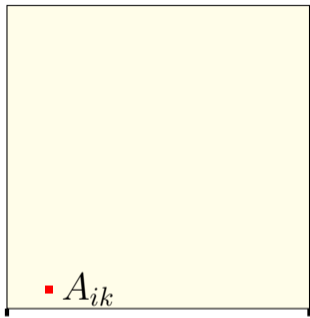
for all j :

for all k :

$$C_{ij+} = A_{ik} \times B_{kj}$$

looking only at two innermost loops together:
some temporal locality in A (column reused)
some temporal locality in B (row reused)
some temporal locality in C (row reused)

array usage: kij order



A_{x0} A_{xN}

for all k :

for all i :

for all j :

$$C_{ij} += A_{ik} \times B_{kj}$$

if N large:

using C_{ij} once per load into cache

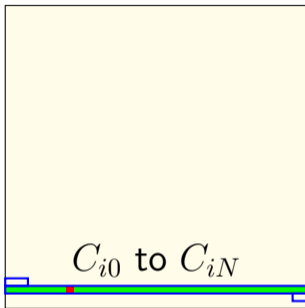
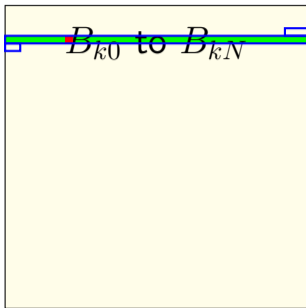
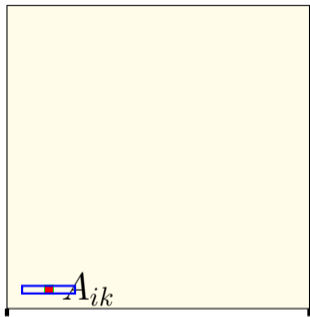
(but using $C_{i,j+1}$ right after)

using A_{ik} many times per load-into-cache

using B_{kj} once per load into cache

(but using $B_{k,j+1}$ right after)

array usage: *kij* order



A_{x0} A_{xN}

for all k :

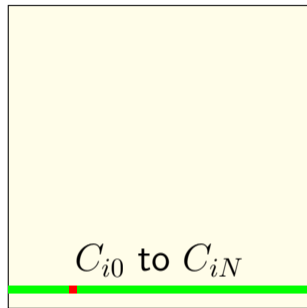
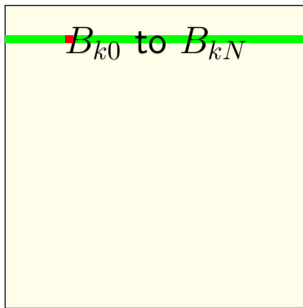
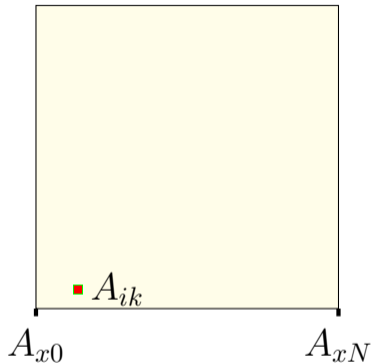
for all i :

for all j :

$$C_{ij+} = A_{ik} \times B_{kj}$$

looking only at innermost loop:
spatial locality in B, C
(use most of loaded B, C cache blocks)
no useful spatial locality in A
(rest of A's cache block wasted)

array usage: kij order



for all k :

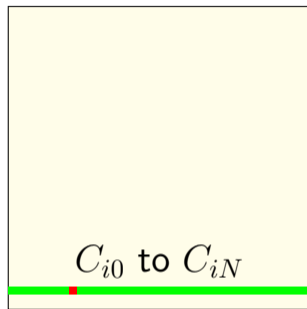
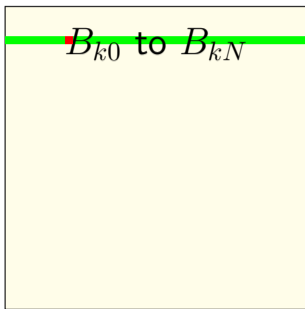
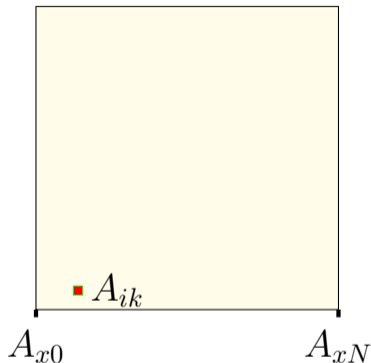
for all i :

for all j :

$$C_{ij+} = A_{ik} \times B_{kj}$$

looking only at innermost loop:
temporal locality in A
no temporal locality in B, C
(B, C values used exactly once)

array usage: kij order



for all k :

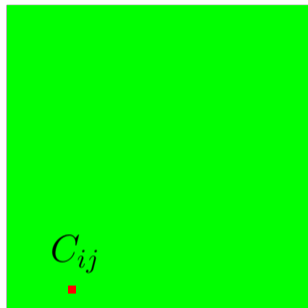
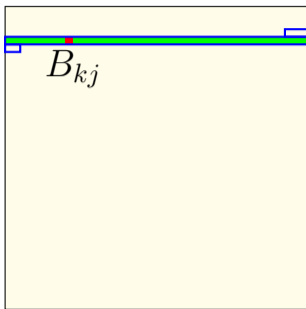
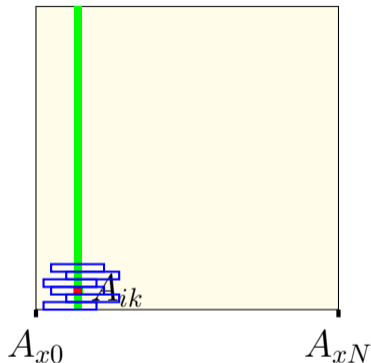
for all i :

for all j :

$$C_{ij} += A_{ik} \times B_{kj}$$

looking only at innermost loop:
processing one element of A (use many times)
row of B (each element used once)
column of C (each element used once)

array usage: kij order



for all k :

for all i :

for all j :

$$C_{ij} += A_{ik} \times B_{kj}$$

looking only at two innermost loops together:
good temporal locality in A (column reused)
good temporal locality in B (row reused)
bad temporal locality in C (nothing reused)

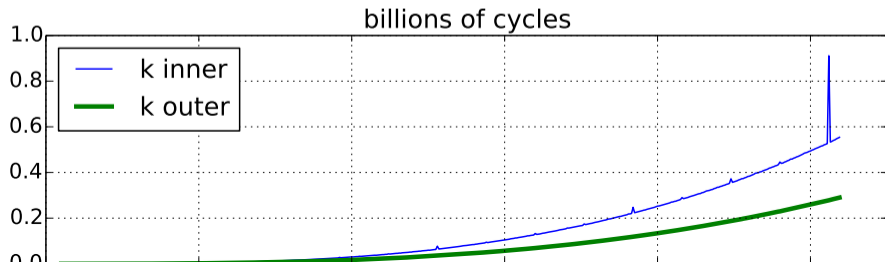
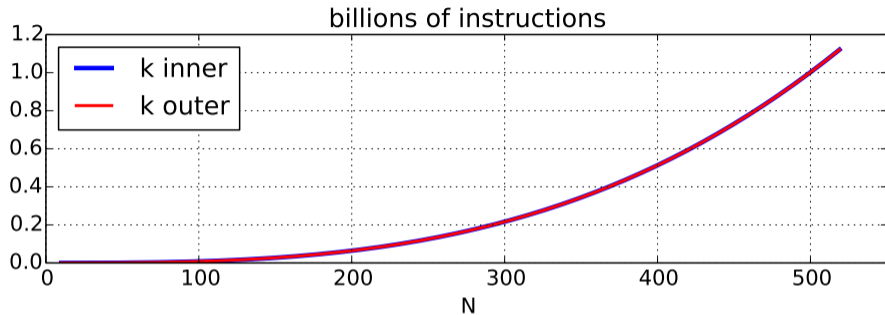
matrix multiply

$$C_{ij} = \sum_{k=1}^n A_{ik} \times B_{kj}$$

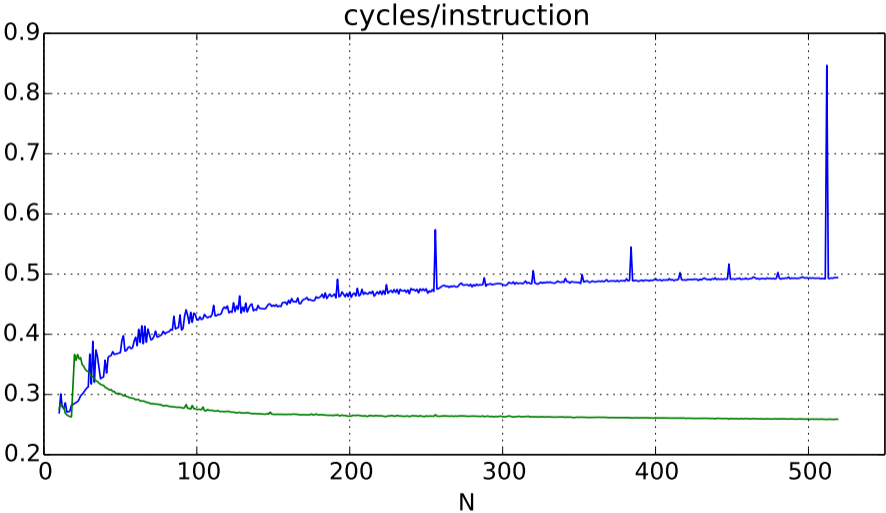
```
/* version 1: inner loop is k, middle is j */  
for (int i = 0; i < N; ++i)  
    for (int j = 0; j < N; ++j)  
        for (int k = 0; k < N; ++k)  
            C[i*N+j] += A[i * N + k] * B[k * N + j];
```

```
/* version 2: outer loop is k, middle is i */  
for (int k = 0; k < N; ++k)  
    for (int i = 0; i < N; ++i)  
        for (int j = 0; j < N; ++j)  
            C[i*N+j] += A[i * N + k] * B[k * N + j];
```

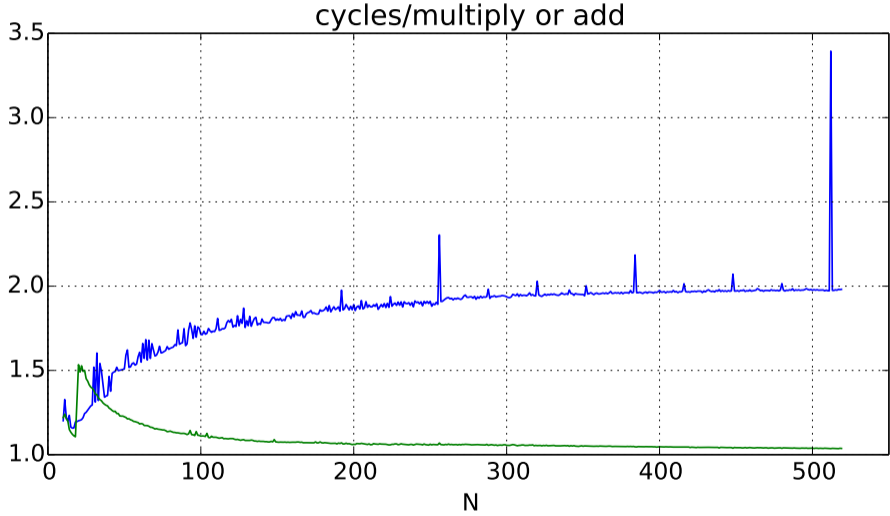

performance (with $A=B$)



alternate view 1: cycles/instruction



alternate view 2: cycles/operation



counting misses: version 1

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

if N really large

assumption: can't get close to storing N values in cache at once

for A: about $N \div$ block size misses per k-loop

total misses: $N^3 \div$ block size

for B: about N misses per k-loop

total misses: N^3

for C: about $1 \div$ block size miss per k-loop

total misses: $N^2 \div$ block size

counting misses: version 2

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

for A: about 1 misses per j-loop

total misses: N^2

for B: about $N \div$ block size miss per j-loop

total misses: $N^3 \div$ block size

for C: about $N \div$ block size miss per j-loop

total misses: $N^3 \div$ block size

exercise: miss estimating (2)

```
for (int k = 0; k < 1000; k += 1)
  for (int i = 0; i < 1000; i += 1)
    for (int j = 0; j < 1000; j += 1)
      A[k*N+j] += B[i*N+j];
```

assuming: 4 elements per block

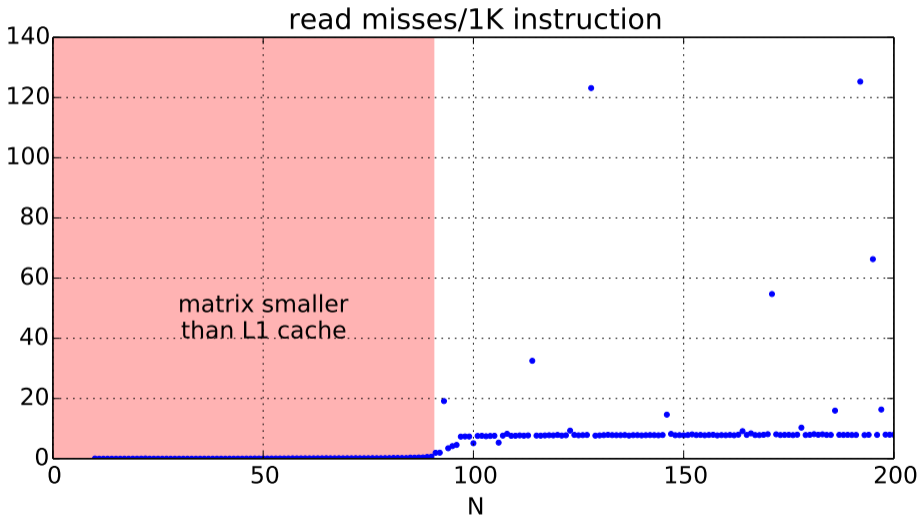
assuming: cache not close to big enough to hold 1K elements

estimate: *approximately* how many misses for A , B ?

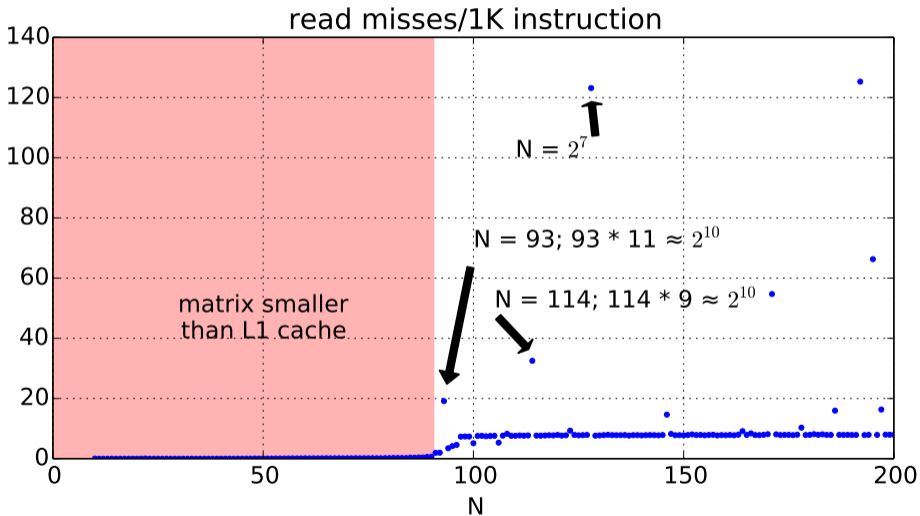
L1 misses (with $A=B$)



L1 miss detail (1)



L1 miss detail (2)



addresses

$B[k*114+j]$	is at	10	0000	0000	0100
$B[k*114+j+1]$	is at	10	0000	0000	1000
$B[(k+1)*114+j]$	is at	10	0011	1001	0100
$B[(k+2)*114+j]$	is at	10	0101	0101	1100
...					
$B[(k+9)*114+j]$	is at	11	0000	0000	1100

addresses

$B[k*114+j]$	is at	10	0000	0000	0100
$B[k*114+j+1]$	is at	10	0000	0000	1000
$B[(k+1)*114+j]$	is at	10	0011	1001	0100
$B[(k+2)*114+j]$	is at	10	0101	0101	1100
...					
$B[(k+9)*114+j]$	is at	11	0000	0000	1100

test system L1 cache: 6 index bits, 6 block offset bits

conflict misses

powers of two — lower order bits unchanged

$B[k*93+j]$ and $B[(k+11)*93+j]$:

1023 elements apart (4092 bytes; 63.9 cache blocks)

64 sets in L1 cache: usually maps to same set

$B[k*93+(j+1)]$ will not be cached (next i loop)

even if in same block as $B[k*93+j]$

how to fix? improve spatial locality
(maybe even if it requires copying)

locality exercise (2)

```
/* version 2 */  
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?

a transformation

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

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

split the loop over k — should be exactly the same
(assuming even N)

a transformation

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

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

split the loop over k — should be exactly the same
(assuming even N)

simple blocking

```
for (int kk = 0; kk < N; kk += 2)
  /* was here: for (int k = kk; k < kk + 2; ++k) */
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      /* load Aik, Aik+1 into cache and process: */
      for (int k = kk; k < kk + 2; ++k)
        C[i*N+j] += A[i*N+k] * B[k*N+j];
```

now **reorder** split loop — same calculations

simple blocking

```
for (int kk = 0; kk < N; kk += 2)
    /* was here: for (int k = kk; k < kk + 2; ++k) */
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            /* load Aik, Aik+1 into cache and process: */
            for (int k = kk; k < kk + 2; ++k)
                C[i*N+j] += A[i*N+k] * B[k*N+j];
```

now **reorder** split loop — same calculations

now handle B_{ij} for $k + 1$ right after B_{ij} for k

(previously: $B_{i,j+1}$ for k right after B_{ij} for k)

simple blocking

```
for (int kk = 0; kk < N; kk += 2)
  /* was here: for (int k = kk; k < kk + 2; ++k) */
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      /* load Aik, Aik+1 into cache and process: */
      for (int k = kk; k < kk + 2; ++k)
        C[i*N+j] += A[i*N+k] * B[k*N+j];
```

now **reorder** split loop — same calculations

now handle B_{ij} for $k + 1$ right after B_{ij} for k

(previously: $B_{i,j+1}$ for k right after B_{ij} for k)

simple blocking – expanded

```
for (int kk = 0; kk < N; kk += 2) {
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            /* process a "block" of 2 k values: */
            C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
            C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
        }
    }
}
```

simple blocking – expanded

```
for (int kk = 0; kk < N; kk += 2) {
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            /* process a "block" of 2 k values: */
            C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
            C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
        }
    }
}
```

Temporal locality in C_{ij} s

simple blocking – expanded

```
for (int kk = 0; kk < N; kk += 2) {  
    for (int i = 0; i < N; ++i) {  
        for (int j = 0; j < N; ++j) {  
            /* process a "block" of 2 k values: */  
            C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];  
            C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];  
        }  
    }  
}
```

More spatial locality in A_{ik}

simple blocking – expanded

```
for (int kk = 0; kk < N; kk += 2) {  
    for (int i = 0; i < N; ++i) {  
        for (int j = 0; j < N; ++j) {  
            /* process a "block" of 2 k values: */  
            C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];  
            C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];  
        }  
    }  
}
```

Still have good spatial locality in B_{kj} , C_{ij}

counting misses for A (1)

```
for (int kk = 0; kk < N; kk += 2)
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
    }
```

access pattern for A:

$A[0*N+0]$, $A[0*N+1]$, $A[0*N+0]$, $A[0*N+1]$... (repeats N times)

$A[1*N+0]$, $A[1*N+1]$, $A[1*N+0]$, $A[1*N+1]$... (repeats N times)

...

...

counting misses for A (1)

```
for (int kk = 0; kk < N; kk += 2)
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
    }
```

access pattern for A:

$A[0*N+0]$, $A[0*N+1]$, $A[0*N+0]$, $A[0*N+1]$... (repeats N times)

$A[1*N+0]$, $A[1*N+1]$, $A[1*N+0]$, $A[1*N+1]$... (repeats N times)

...

$A[(N-1)*N+0]$, $A[(N-1)*N+1]$, $A[(N-1)*N+0]$, $A[(N-1)*N+1]$...

$A[0*N+2]$, $A[0*N+3]$, $A[0*N+2]$, $A[0*N+3]$...

...

counting misses for A (1)

```
for (int kk = 0; kk < N; kk += 2)
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
    }
```

access pattern for A:

$A[0*N+0]$, $A[0*N+1]$, $A[0*N+0]$, $A[0*N+1]$... (repeats N times)

$A[1*N+0]$, $A[1*N+1]$, $A[1*N+0]$, $A[1*N+1]$... (repeats N times)

...

$A[(N-1)*N+0]$, $A[(N-1)*N+1]$, $A[(N-1)*N+0]$, $A[(N-1)*N+1]$...

$A[0*N+2]$, $A[0*N+3]$, $A[0*N+2]$, $A[0*N+3]$...

...

counting misses for A (2)

$A[0*N+0]$, $A[0*N+1]$, $A[0*N+0]$, $A[0*N+1]$... (repeats N times)

$A[1*N+0]$, $A[1*N+1]$, $A[1*N+0]$, $A[1*N+1]$... (repeats N times)

...

...

counting misses for A (2)

$A[0*N+0]$, $A[0*N+1]$, $A[0*N+0]$, $A[0*N+1]$... (repeats N times)

$A[1*N+0]$, $A[1*N+1]$, $A[1*N+0]$, $A[1*N+1]$... (repeats N times)

...

$A[(N-1)*N+0]$, $A[(N-1)*N+1]$, $A[(N-1)*N+0]$, $A[(N-1)*N+1]$...

$A[0*N+2]$, $A[0*N+3]$, $A[0*N+2]$, $A[0*N+3]$...

...

likely cache misses: only first iterations of j loop

how many cache misses per iteration? usually one

$A[0*N+0]$ and $A[0*N+1]$ usually in same cache block

counting misses for A (2)

$A[0*N+0]$, $A[0*N+1]$, $A[0*N+0]$, $A[0*N+1]$... (repeats N times)

$A[1*N+0]$, $A[1*N+1]$, $A[1*N+0]$, $A[1*N+1]$... (repeats N times)

...

$A[(N-1)*N+0]$, $A[(N-1)*N+1]$, $A[(N-1)*N+0]$, $A[(N-1)*N+1]$...

$A[0*N+2]$, $A[0*N+3]$, $A[0*N+2]$, $A[0*N+3]$...

...

likely cache misses: only first iterations of j loop

how many cache misses per iteration? usually one

$A[0*N+0]$ and $A[0*N+1]$ usually in same cache block

about $\frac{N}{2} \cdot N$ misses total

counting misses for B (1)

```
for (int kk = 0; kk < N; kk += 2)
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
    }
```

access pattern for B:

$B[0*N+0]$, $B[1*N+0]$, ... $B[0*N+(N-1)]$, $B[1*N+(N-1)]$

$B[2*N+0]$, $B[3*N+0]$, ... $B[2*N+(N-1)]$, $B[3*N+(N-1)]$

$B[4*N+0]$, $B[5*N+0]$, ... $B[4*N+(N-1)]$, $B[5*N+(N-1)]$

...

$B[0*N+0]$, $B[1*N+0]$, ... $B[0*N+(N-1)]$, $B[1*N+(N-1)]$

...

counting misses for B (2)

access pattern for B:

$B[0*N+0]$, $B[1*N+0]$, ... $B[0*N+(N-1)]$, $B[1*N+(N-1)]$

$B[2*N+0]$, $B[3*N+0]$, ... $B[2*N+(N-1)]$, $B[3*N+(N-1)]$

$B[4*N+0]$, $B[5*N+0]$, ... $B[4*N+(N-1)]$, $B[5*N+(N-1)]$

...

$B[0*N+0]$, $B[1*N+0]$, ... $B[0*N+(N-1)]$, $B[1*N+(N-1)]$

...

counting misses for B (2)

access pattern for B:

$B[0*N+0]$, $B[1*N+0]$, ... $B[0*N+(N-1)]$, $B[1*N+(N-1)]$

$B[2*N+0]$, $B[3*N+0]$, ... $B[2*N+(N-1)]$, $B[3*N+(N-1)]$

$B[4*N+0]$, $B[5*N+0]$, ... $B[4*N+(N-1)]$, $B[5*N+(N-1)]$

...

$B[0*N+0]$, $B[1*N+0]$, ... $B[0*N+(N-1)]$, $B[1*N+(N-1)]$

...

likely cache misses: any access, each time

counting misses for B (2)

access pattern for B:

$B[0*N+0]$, $B[1*N+0]$, ... $B[0*N+(N-1)]$, $B[1*N+(N-1)]$

$B[2*N+0]$, $B[3*N+0]$, ... $B[2*N+(N-1)]$, $B[3*N+(N-1)]$

$B[4*N+0]$, $B[5*N+0]$, ... $B[4*N+(N-1)]$, $B[5*N+(N-1)]$

...

$B[0*N+0]$, $B[1*N+0]$, ... $B[0*N+(N-1)]$, $B[1*N+(N-1)]$

...

likely cache misses: any access, each time

how many cache misses per iteration? equal to # cache blocks in 2 rows

counting misses for B (2)

access pattern for B:

$B[0*N+0]$, $B[1*N+0]$, ... $B[0*N+(N-1)]$, $B[1*N+(N-1)]$

$B[2*N+0]$, $B[3*N+0]$, ... $B[2*N+(N-1)]$, $B[3*N+(N-1)]$

$B[4*N+0]$, $B[5*N+0]$, ... $B[4*N+(N-1)]$, $B[5*N+(N-1)]$

...

$B[0*N+0]$, $B[1*N+0]$, ... $B[0*N+(N-1)]$, $B[1*N+(N-1)]$

...

likely cache misses: any access, each time

how many cache misses per iteration? equal to # cache blocks in 2 rows

about $\frac{N}{2} \cdot N \cdot \frac{2N}{\text{block size}} = N^3 \div \text{block size}$ misses

simple blocking – counting misses

```
for (int kk = 0; kk < N; kk += 2)
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
    }
```

$\frac{N}{2} \cdot N$ j-loop executions and (assuming N large):

about 1 misses from A per j-loop

$N^2/2$ total misses (before blocking: N^2)

about $2N \div$ block size misses from B per j-loop

$N^3 \div$ block size total misses (same as before blocking)

about $N \div$ block size misses from C per j-loop

$N^3 \div (2 \cdot \text{block size})$ total misses (before: $N^3 \div$ block size)

simple blocking – counting misses

```
for (int kk = 0; kk < N; kk += 2)
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
    }
```

$\frac{N}{2} \cdot N$ j-loop executions and (assuming N large):

about 1 misses from A per j-loop

$N^2/2$ total misses (before blocking: N^2)

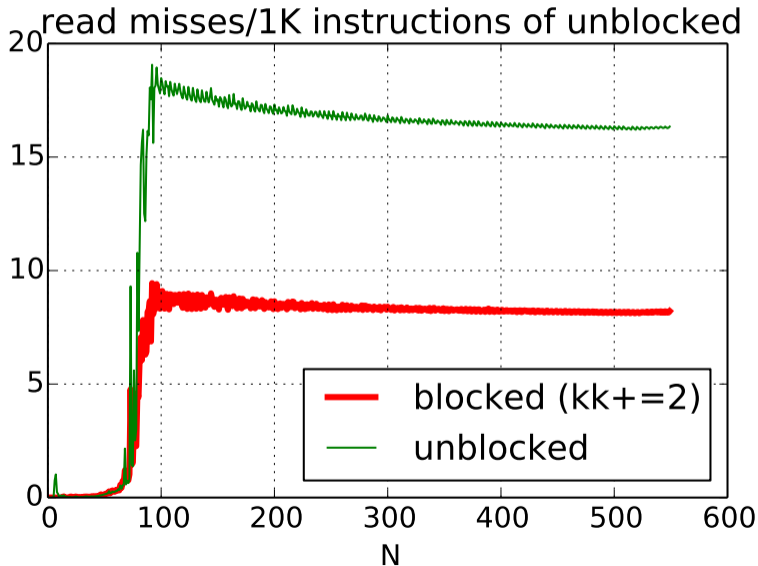
about $2N \div$ block size misses from B per j-loop

$N^3 \div$ block size total misses (same as before blocking)

about $N \div$ block size misses from C per j-loop

$N^3 \div (2 \cdot \text{block size})$ total misses (before: $N^3 \div$ block size)

improvement in read misses



simple blocking (2)

same thing for i in addition to k ?

```
for (int kk = 0; kk < N; kk += 2) {
    for (int ii = 0; ii < N; ii += 2) {
        for (int j = 0; j < N; ++j) {
            /* process a "block": */
            for (int k = kk; k < kk + 2; ++k)
                for (int i = 0; i < ii + 2; ++i)
                    C[i*N+j] += A[i*N+k] * B[k*N+j];
        }
    }
}
```

simple blocking — locality

```
for (int k = 0; k < N; k += 2) {  
  for (int i = 0; i < N; i += 2) {  
    /* load a block around Aik */  
    for (int j = 0; j < N; ++j) {  
      /* process a "block": */  
       $C_{i+0,j} += A_{i+0,k+0} * B_{k+0,j}$   
       $C_{i+0,j} += A_{i+0,k+1} * B_{k+1,j}$   
       $C_{i+1,j} += A_{i+1,k+0} * B_{k+0,j}$   
       $C_{i+1,j} += A_{i+1,k+1} * B_{k+1,j}$   
    }  
  }  
}
```

simple blocking — locality

```
for (int k = 0; k < N; k += 2) {  
  for (int i = 0; i < N; i += 2) {  
    /* load a block around Aik */  
    for (int j = 0; j < N; ++j) {  
      /* process a "block": */  
       $C_{i+0,j} += A_{i+0,k+0} * B_{k+0,j}$   
       $C_{i+0,j} += A_{i+0,k+1} * B_{k+1,j}$   
       $C_{i+1,j} += A_{i+1,k+0} * B_{k+0,j}$   
       $C_{i+1,j} += A_{i+1,k+1} * B_{k+1,j}$   
    }  
  }  
}
```

now: more temporal locality in B

previously: access B_{kj} , then don't use it again for a long time

simple blocking — counting misses for A

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    for (int j = 0; j < N; ++j) {
      Ci+0,j += Ai+0,k+0 * Bk+0,j
      Ci+0,j += Ai+0,k+1 * Bk+1,j
      Ci+1,j += Ai+1,k+0 * Bk+0,j
      Ci+1,j += Ai+1,k+1 * Bk+1,j
    }
```

$\frac{N}{2} \cdot \frac{N}{2}$ iterations of j loop

likely 2 misses per loop with A (2 cache blocks)

total misses: $\frac{N^2}{2}$ (same as only blocking in K)

simple blocking — counting misses for B

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    for (int j = 0; j < N; ++j) {
      Ci+0,j += Ai+0,k+0 * Bk+0,j
      Ci+0,j += Ai+0,k+1 * Bk+1,j
      Ci+1,j += Ai+1,k+0 * Bk+0,j
      Ci+1,j += Ai+1,k+1 * Bk+1,j
    }
```

$\frac{N}{2} \cdot \frac{N}{2}$ iterations of j loop

likely $2 \div$ block size misses per iteration with B

total misses: $\frac{N^3}{2 \cdot \text{block size}}$ (before: $\frac{N^3}{\text{block size}}$)

simple blocking — counting misses for C

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    for (int j = 0; j < N; ++j) {
      Ci+0,j += Ai+0,k+0 * Bk+0,j
      Ci+0,j += Ai+0,k+1 * Bk+1,j
      Ci+1,j += Ai+1,k+0 * Bk+0,j
      Ci+1,j += Ai+1,k+1 * Bk+1,j
    }
```

$\frac{N}{2} \cdot \frac{N}{2}$ iterations of j loop

likely $\frac{2}{\text{block size}}$ misses per iteration with C

total misses: $\frac{N^3}{2}$ (same as blocking only in K)

simple blocking — counting misses (total)

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    for (int j = 0; j < N; ++j) {
      Ci+0,j += Ai+0,k+0 * Bk+0,j
      Ci+0,j += Ai+0,k+1 * Bk+1,j
      Ci+1,j += Ai+1,k+0 * Bk+0,j
      Ci+1,j += Ai+1,k+1 * Bk+1,j
    }
```

before:

$$A: \frac{N^2}{2}; \quad B: \frac{N^3}{1 \cdot \text{block size}}; \quad C: \frac{N^3}{1 \cdot \text{block size}}$$

after:

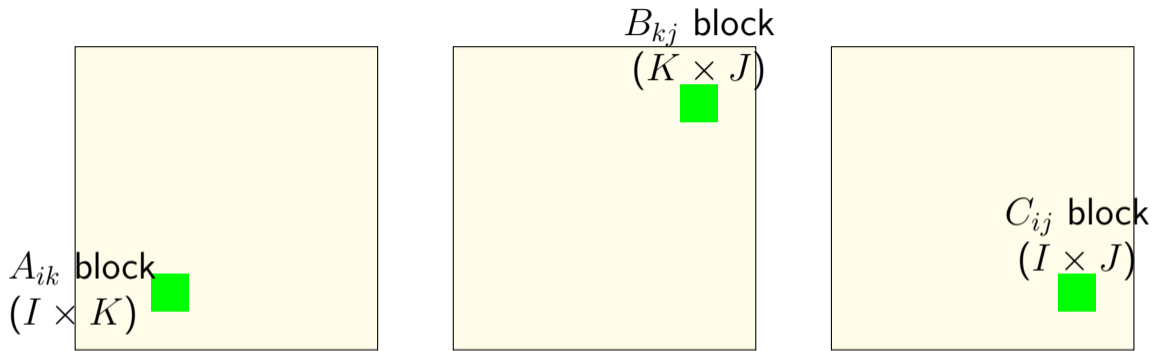
$$A: \frac{N^2}{2}; \quad B: \frac{N^3}{2 \cdot \text{block size}}; \quad C: \frac{N^3}{2 \cdot \text{block size}}$$

generalizing: divide and conquer

```
partial_matrixmultiply(float *A, float *B, float *C
                        int startI, int endI, ...) {
    for (int i = startI; i < endI; ++i) {
        for (int j = startJ; j < endJ; ++j) {
            for (int k = startK; k < endK; ++k) {
                ...
            }
        }
    }
}

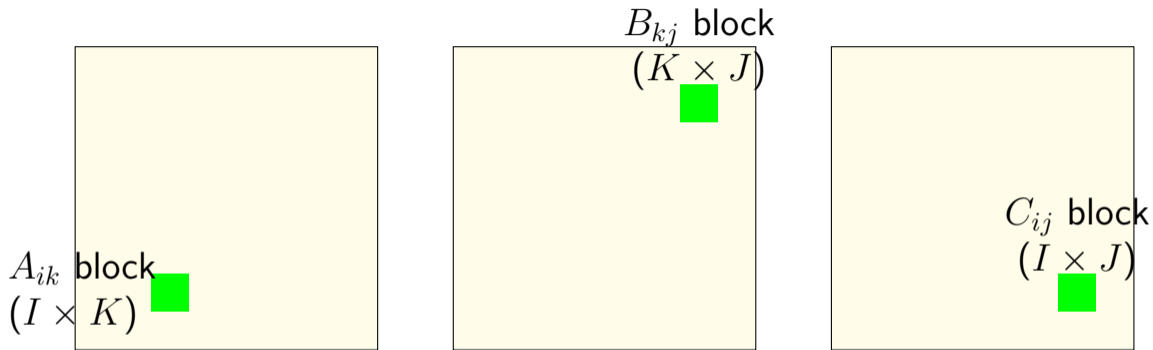
matrix_multiply(float *A, float *B, float *C, int N) {
    for (int ii = 0; ii < N; ii += BLOCK_I)
        for (int jj = 0; jj < N; jj += BLOCK_J)
            for (int kk = 0; kk < N; kk += BLOCK_K)
                ...
                /* do everything for segment of A, B, C
                   that fits in cache! */
                partial_matmul(A, B, C,
                               ii, ii + BLOCK_I,
                               jj, jj + BLOCK_J,
```

array usage: matrix block $C_{ij} += A_{ik} \cdot B_{kj}$



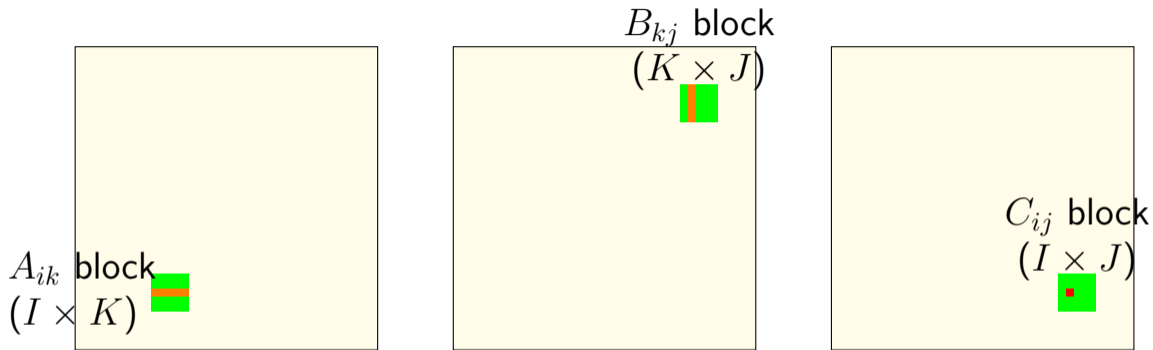
inner loops work on “matrix block” of A, B, C
rather than rows of some, little blocks of others
blocks fit into cache (b/c we choose I, K, J)

array usage: matrix block $C_{ij} += A_{ik} \cdot B_{kj}$



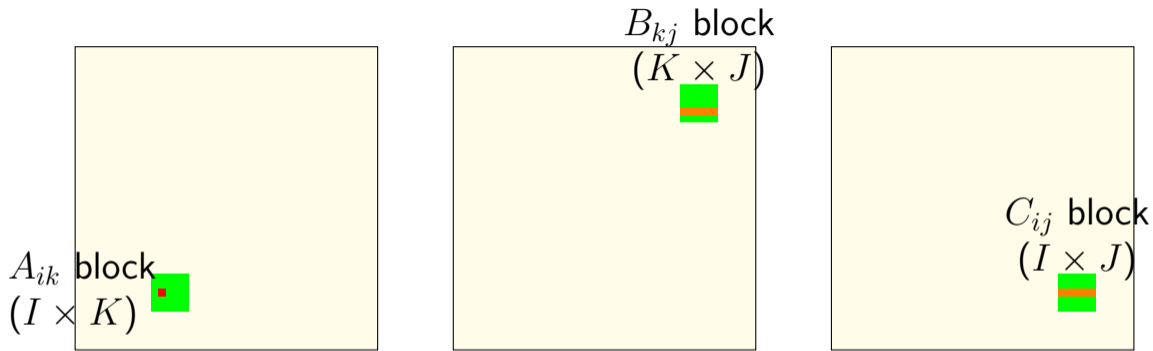
now (versus loop ordering example)
some spatial locality in A, B, and C
some temporal locality in A, B, and C

array usage: matrix block $C_{ij} += A_{ik} \cdot B_{kj}$



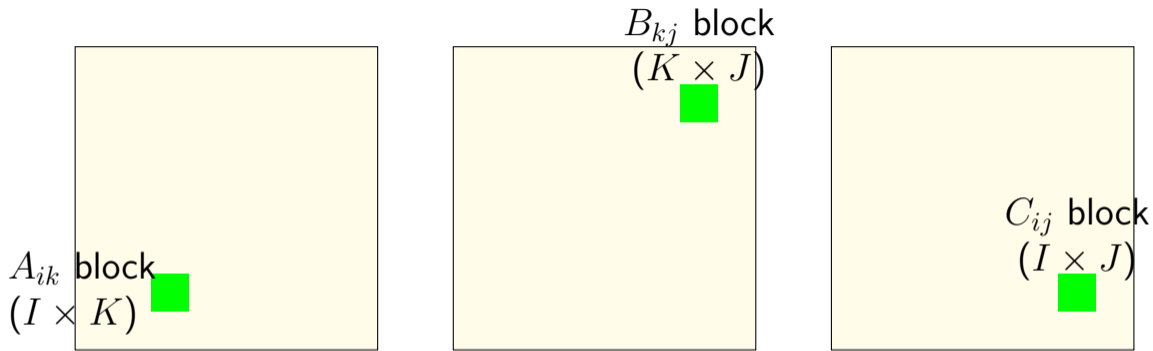
C_{ij} calculation uses strips from A , B
 K calculations for one cache miss
good temporal locality!

array usage: matrix block $C_{ij} += A_{ik} \cdot B_{kj}$



A_{ik} used with entire strip of B J calculations for one cache miss
good temporal locality!

array usage: matrix block $C_{ij} += A_{ik} \cdot B_{kj}$



(approx.) KIJ fully cached calculations
for $KI + IJ + KJ$ values need to be loaded per “matrix block”
(assuming everything stays in cache)

cache blocking efficiency

for each of N^3/IJK matrix blocks:

load $I \times K$ elements of A_{ik} :

$\approx IK \div$ block size misses per matrix block

$\approx N^3/(J \cdot \text{blocksize})$ misses total

load $K \times J$ elements of B_{kj} :

$\approx N^3/(I \cdot \text{blocksize})$ misses total

load $I \times J$ elements of C_{ij} :

$\approx N^3/(K \cdot \text{blocksize})$ misses total

bigger blocks — more work per load!

catch: $IK + KJ + IJ$ elements must fit in cache
otherwise estimates above don't work

cache blocking rule of thumb

fill the **most of the cache with useful data**

and do as much work as possible from that

example: my desktop 32KB L1 cache

$I = J = K = 48$ uses $48^2 \times 3$ elements, or 27KB.

assumption: conflict misses aren't important

systematic approach

```
for (int k = 0; k < N; ++k) {  
  for (int i = 0; i < N; ++i) {  
     $A_{ik}$  loaded once in this loop:  
    for (int j = 0; j < N; ++j)  
       $C_{ij}, B_{kj}$  loaded each iteration (if  $N$  big):  
       $B[i*N+j] += A[i*N+k] * A[k*N+j];$ 
```

values from A_{ik} used N times per load

values from B_{kj} used 1 times per load

but good spatial locality, so cache block of B_{kj} together

values from C_{ij} used 1 times per load

but good spatial locality, so cache block of C_{ij} together

exercise: miss estimating (3)

```
for (int kk = 0; kk < 1000; kk += 10)
  for (int jj = 0; jj < 1000; jj += 10)
    for (int i = 0; i < 1000; i += 1)
      for (int j = jj; j < jj+10; j += 1)
        for (int k = kk; k < kk + 10; k += 1)
          A[k*N+j] += B[i*N+j];
```

assuming: 4 elements per block

assuming: cache not close to big enough to hold 1K elements, but big enough to hold 500 or so

estimate: *approximately* how many misses for A, B?

loop ordering compromises

loop ordering forces compromises:

```
for k: for i: for j: c[i,j] += a[i,k] * b[j,k]
```

perfect temporal locality in $a[i,k]$

bad temporal locality for $c[i,j]$, $b[j,k]$

perfect spatial locality in $c[i,j]$

bad spatial locality in $b[j,k]$, $a[i,k]$

loop ordering compromises

loop ordering forces compromises:

```
for k: for i: for j: c[i,j] += a[i,k] * b[j,k]
```

perfect temporal locality in $a[i,k]$

bad temporal locality for $c[i,j]$, $b[j,k]$

perfect spatial locality in $c[i,j]$

bad spatial locality in $b[j,k]$, $a[i,k]$

cache blocking: work on blocks rather than rows/columns
have some temporal, spatial locality in everything

cache blocking pattern

no perfect loop order? work on rectangular matrix blocks

size amount used in inner loops based on cache size

in practice:

- test performance to determine 'size' of blocks

backup slides

cache organization and miss rate

depends on program; one example:

SPEC CPU2000 benchmarks, 64B block size

LRU replacement policies

data cache miss rates:

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

cache organization and miss rate

depends on program; one example:

SPEC CPU2000 benchmarks, 64B block size

LRU replacement policies

data cache miss rates:

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

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?

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 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

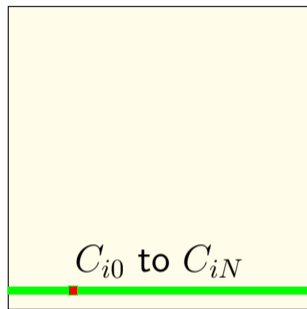
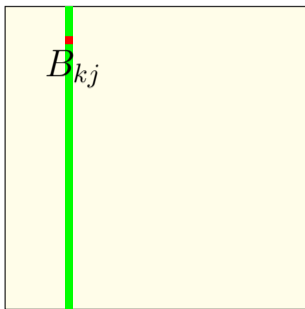
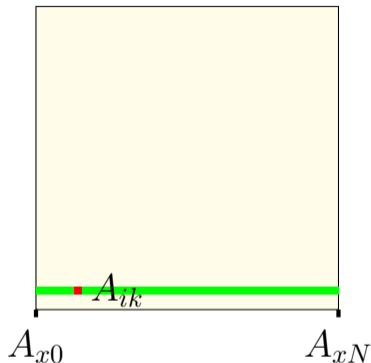
bring in guess at next-to-be-accessed value

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

array usage: ijk order



for all i :

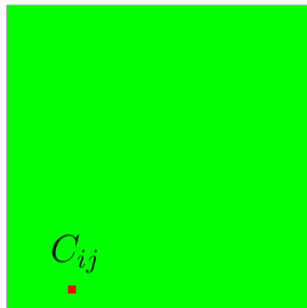
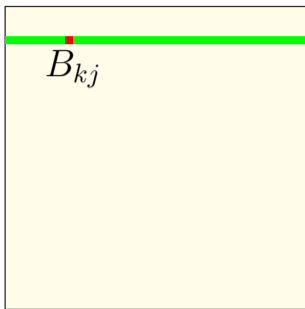
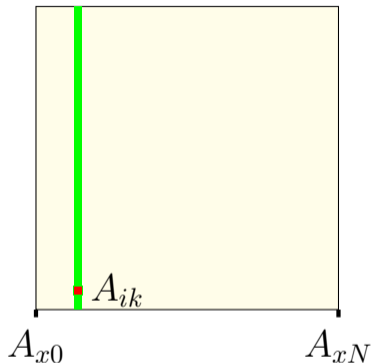
for all j :

for all k :

$$C_{ij+} = A_{ik} \times B_{kj}$$

looking only at two innermost loops together:
good spatial locality in A
poor spatial locality in B
good spatial locality in C

array usage: kij order



for all k :

for all i :

for all j :

$$C_{ij} += A_{ik} \times B_{kj}$$

looking only at two innermost loops together:
poor spatial locality in A
good spatial locality in B
good spatial locality in C

simple blocking – with 3?

```
for (int kk = 0; kk < N; kk += 3)
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
      C[i*N+j] += A[i*N+kk+2] * B[(kk+2)*N+j];
    }
```

$\frac{N}{3} \cdot N$ j-loop iterations, and (assuming N large):

about 1 misses from A per j-loop iteration

$N^2/3$ total misses (before blocking: N^2)

about $3N \div$ block size misses from B per j-loop iteration

$N^3 \div$ block size total misses (same as before)

about $3N \div$ block size misses from C per i-loop iteration

simple blocking – with 3?

```
for (int kk = 0; kk < N; kk += 3)
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
      C[i*N+j] += A[i*N+kk+2] * B[(kk+2)*N+j];
    }
```

$\frac{N}{3} \cdot N$ j-loop iterations, and (assuming N large):

about 1 misses from A per j-loop iteration

$N^2/3$ total misses (before blocking: N^2)

about $3N \div$ block size misses from B per j-loop iteration

$N^3 \div$ block size total misses (same as before)

about $3N \div$ block size misses from C per i-loop iteration

more than 3?

can we just keep doing this increase from 3 to some large X ? ...

assumption: X values from A would stay in cache

X too large — cache not big enough

assumption: X blocks from B would help with spatial locality

X too large — evicted from cache before next iteration

array usage (2 k at a time)

A_{ik} to $A_{i,k+1}$
—

■ B_{ki} to $B_{k+1,i}$

■ C_{ij}

for each kk :

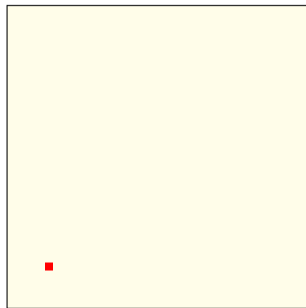
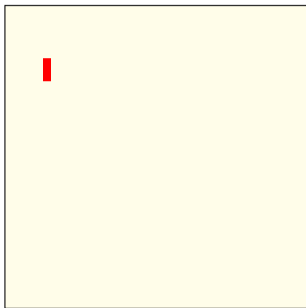
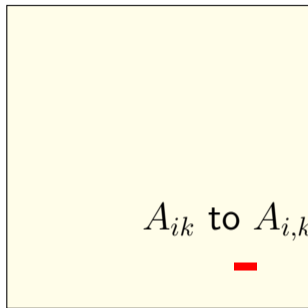
for each i :

for each j :

for $k=kk, kk+1$:

$$C_{ij} += A_{ik} \cdot B_{kj}$$

array usage ($2k$ at a time)



for each kk :

for each i :

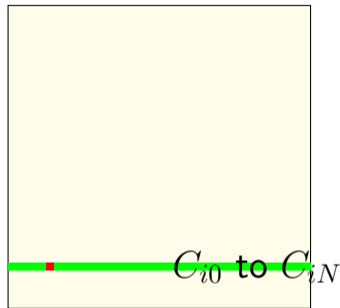
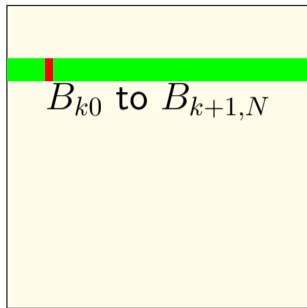
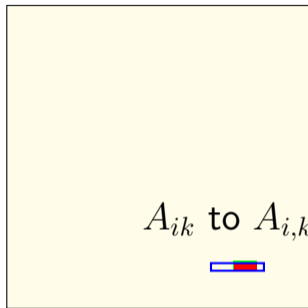
for each j :

for $k=kk, kk+1$:

$$C_{ij} += A_{ik} \cdot B_{kj}$$

within innermost loop
good spatial locality in A
bad locality in B
good temporal locality in C

array usage (2 k at a time)



for each kk :

for each i :

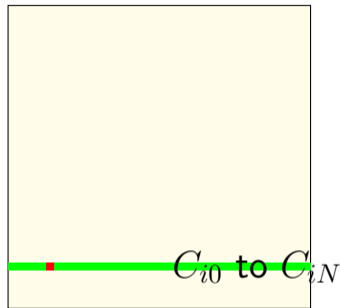
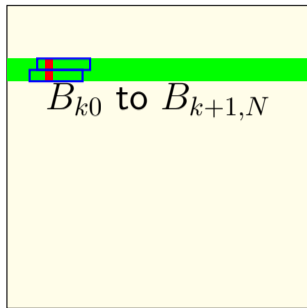
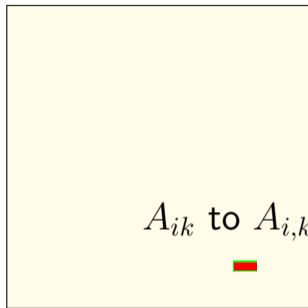
for each j :

for $k=kk, kk+1$:

$$C_{ij+} = A_{ik} \cdot B_{kj}$$

loop over j : better spatial locality
over A than before;
still good temporal locality for A

array usage (2 k at a time)



for each kk :

for each i :

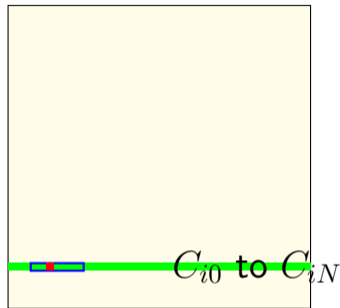
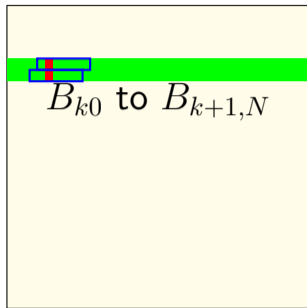
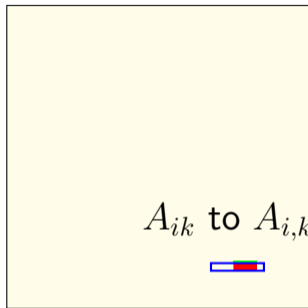
for each j :

for $k=kk, kk+1$:

$$C_{ij} += A_{ik} \cdot B_{kj}$$

loop over j : spatial locality over B is worse
but probably not more misses
cache needs to keep two cache blocks
for next iter instead of one
(probably has the space left over!)

array usage (2 k at a time)



for each kk :

for each i :

for each j :

for $k=kk, kk+1$:

$C_{ij} += A_{ik}$.

right now: only really care about
keeping 4 cache blocks in j loop

have more than 4 cache blocks?

increasing kk increment would use more of them

keeping values in cache

can't *explicitly* ensure values are kept in cache

...but reusing values *effectively* does this

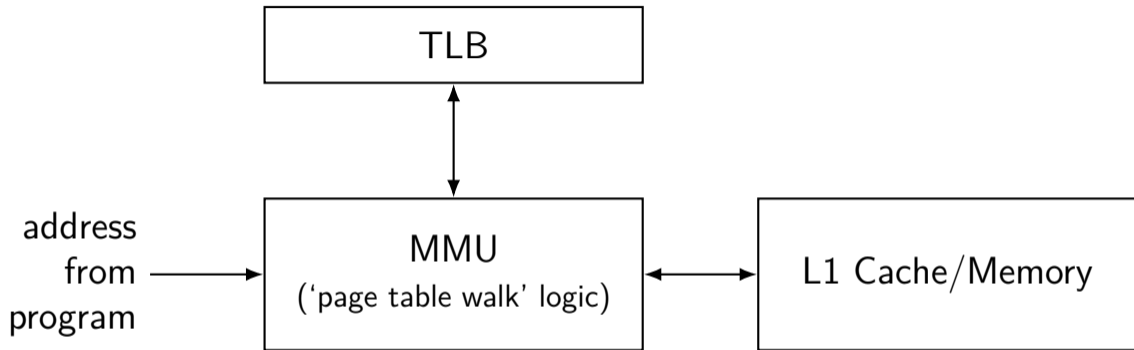
cache will try to keep recently used values

cache optimization ideas: choose what's in the cache

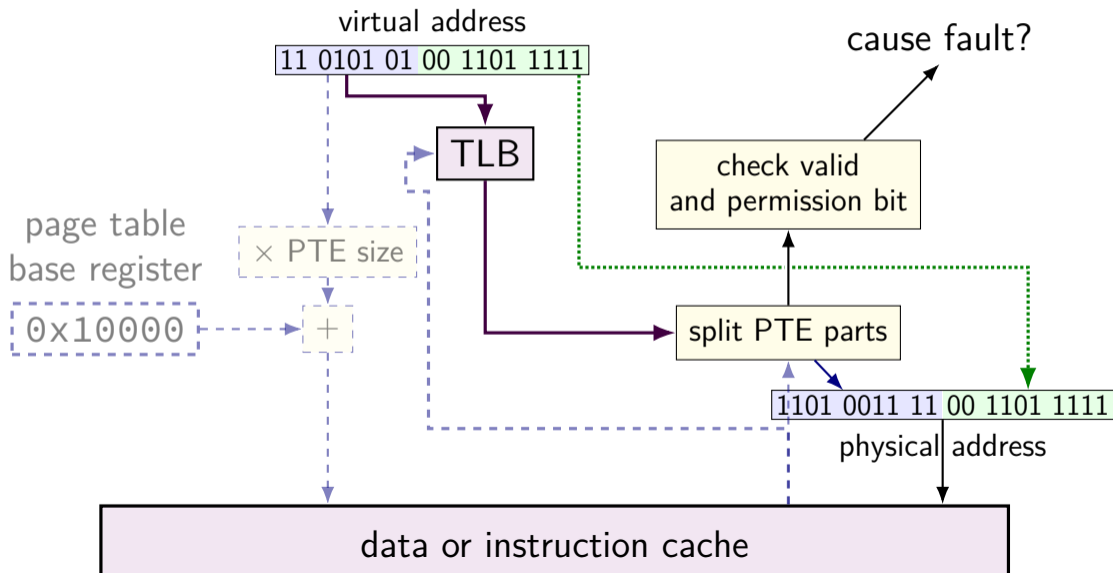
for thinking about it: load values explicitly

for implementing it: access only values we want loaded

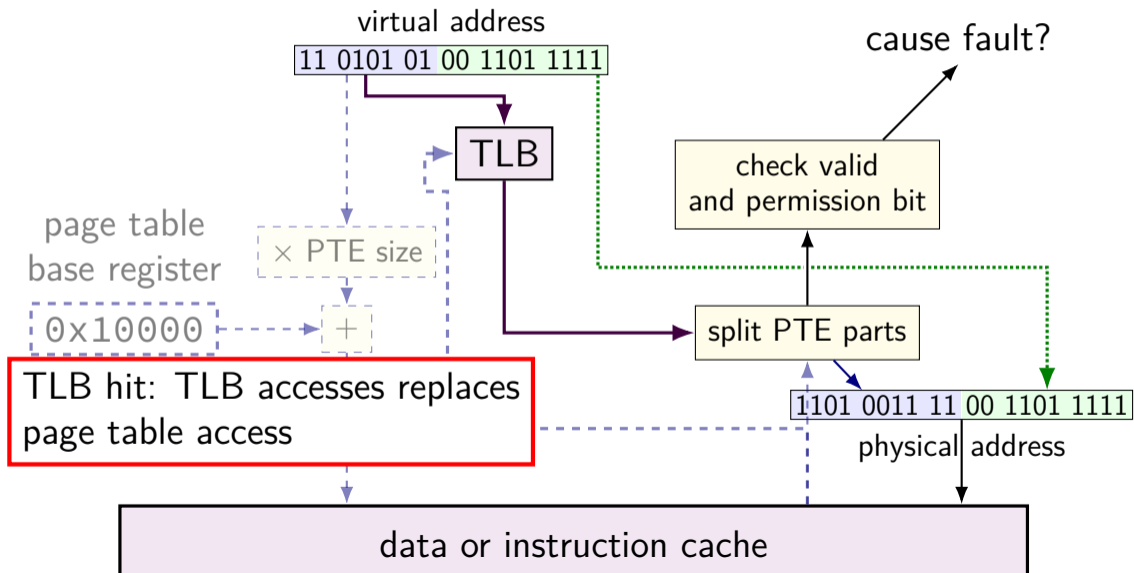
TLB and the MMU (1)



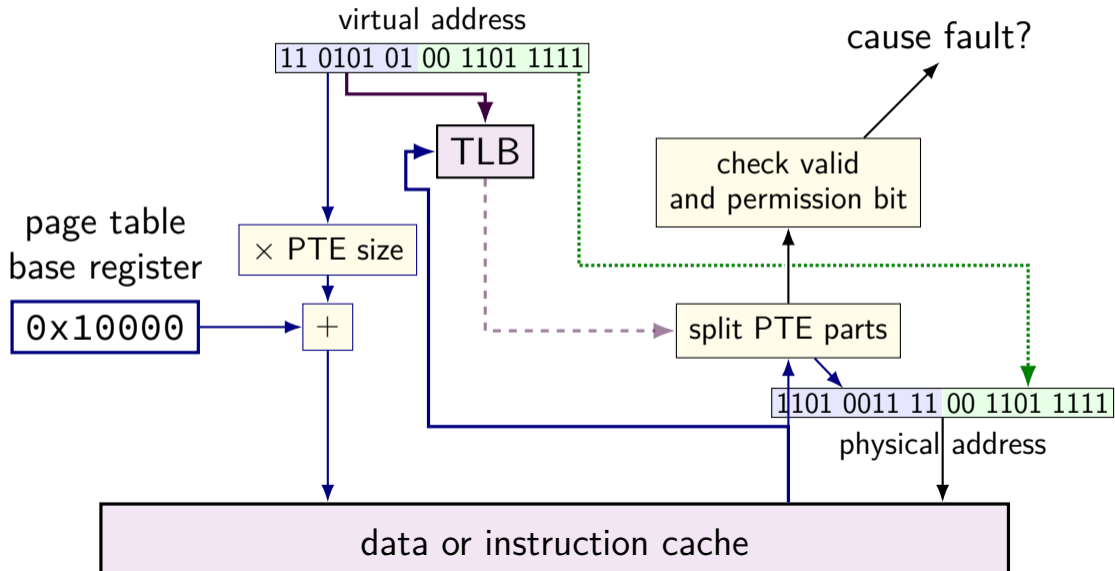
TLB and the MMU (2)



TLB and the MMU (2)

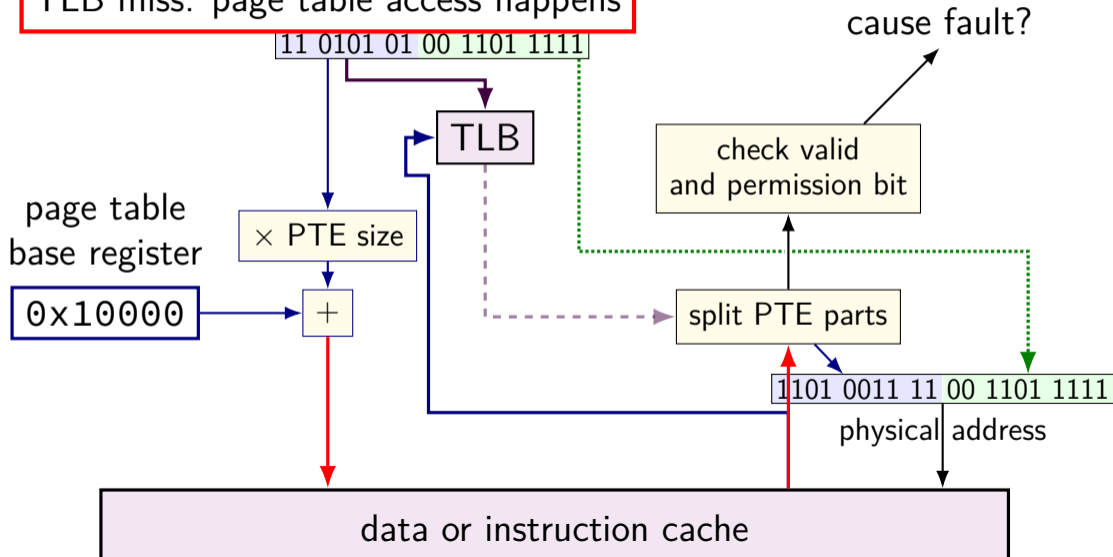


TLB and the MMU (2)



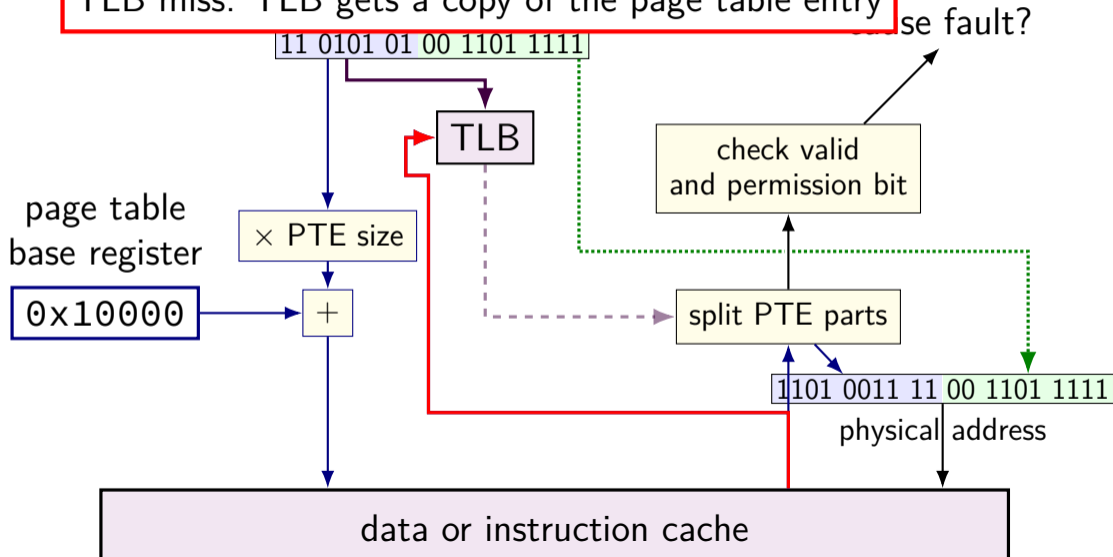
TLB and the MMU (2)

TLB miss: page table access happens

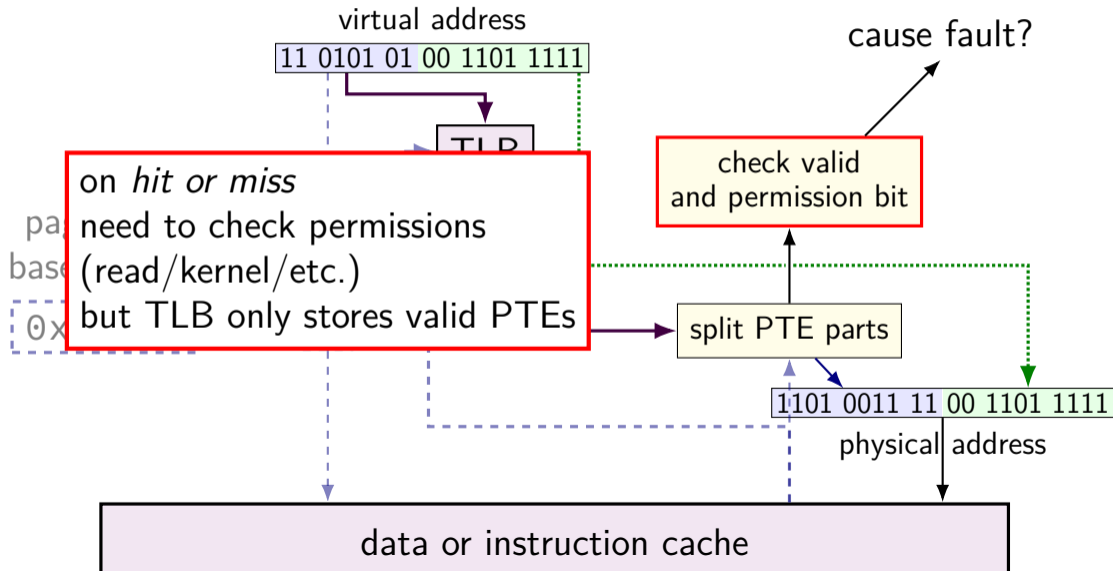


TLB and the MMU (2)

TLB miss: TLB gets a copy of the page table entry



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?

changing page tables

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

e.g. context switch

most entries in TLB refer to things from **wrong process**

oops — read from the wrong process's stack?

option 1: **invalidate** all TLB entries

side effect on “change page table base register” instruction

changing page tables

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

e.g. context switch

most entries in TLB refer to things from **wrong process**

oops — read from the wrong process's stack?

option 1: **invalidate** all TLB entries

side effect on “change page table base register” instruction

option 2: TLB entries contain process ID

set by OS (special register)

checked by TLB in addition to TLB tag, valid bit

editing page tables

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

most common choice: has to be handled **in software**

editing page tables

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

most common choice: has to be handled **in software**

invalid to valid — nothing needed

- TLB doesn't contain invalid entries

- MMU will check memory again

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

- special instruction (x86: `invlpg`)

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

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?

$$64/4 = 16 \text{ sets} \text{ — } 4 \text{ bits}$$

TLB tag bits?

$$48 - 12 = 36 \text{ bit virtual page number} \text{ — } 36 - 4 = 32 \text{ bit TLB tag}$$

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

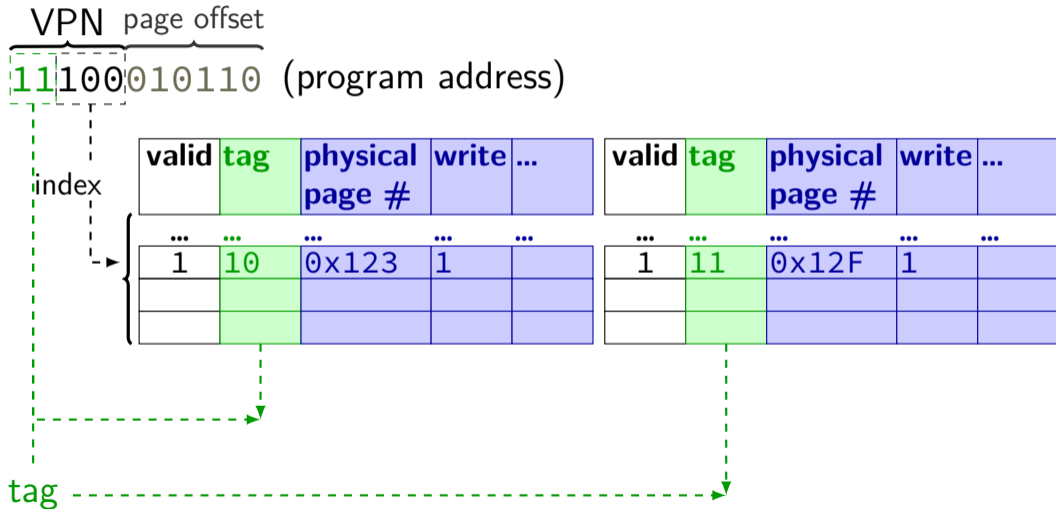
TLB index bits?

$$1536/12 = 128 \text{ sets} \text{ — } 7 \text{ bits}$$

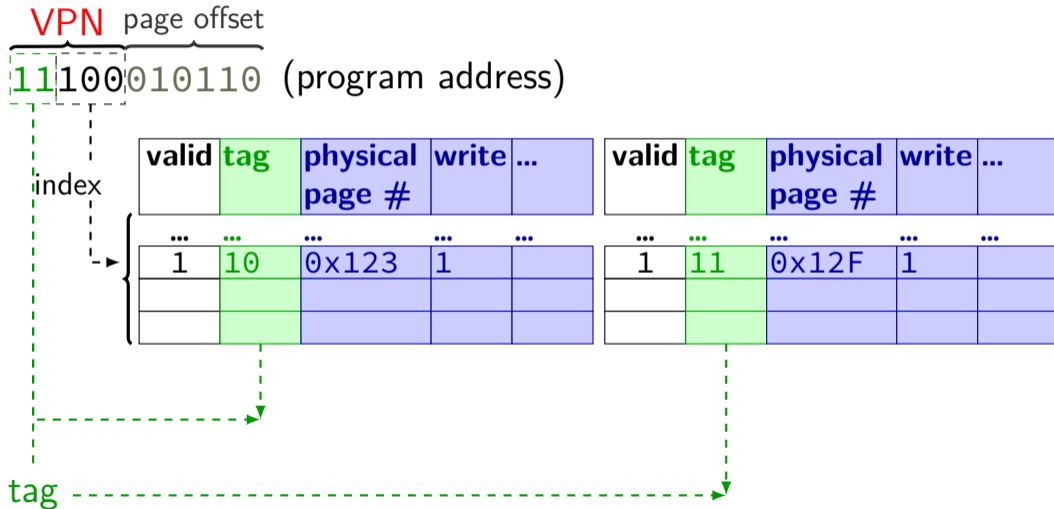
TLB tag bits?

$$48 - 12 = 36 \text{ bit virtual page number} \text{ — } 36 - 7 = 29 \text{ bit TLB tag}$$

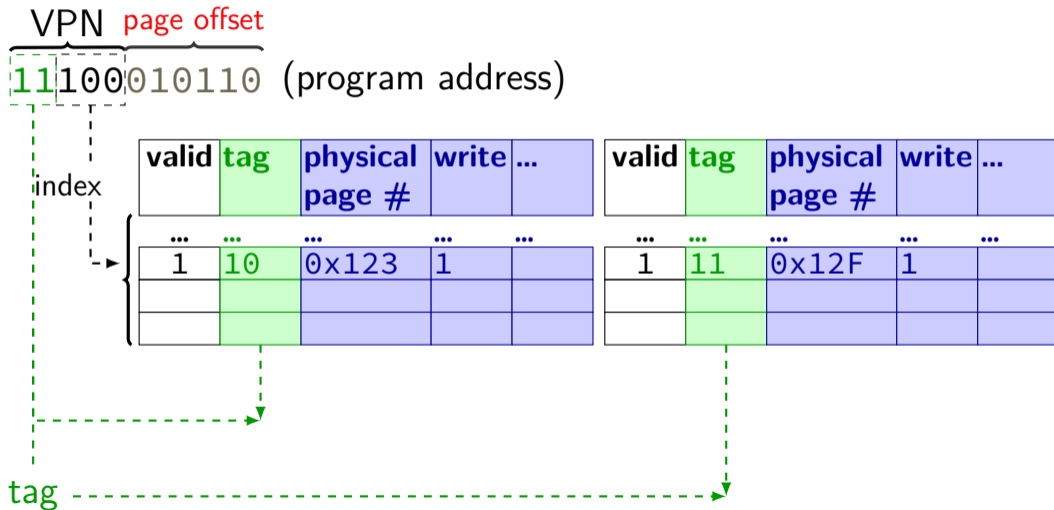
TLB organization (2-way set associative)



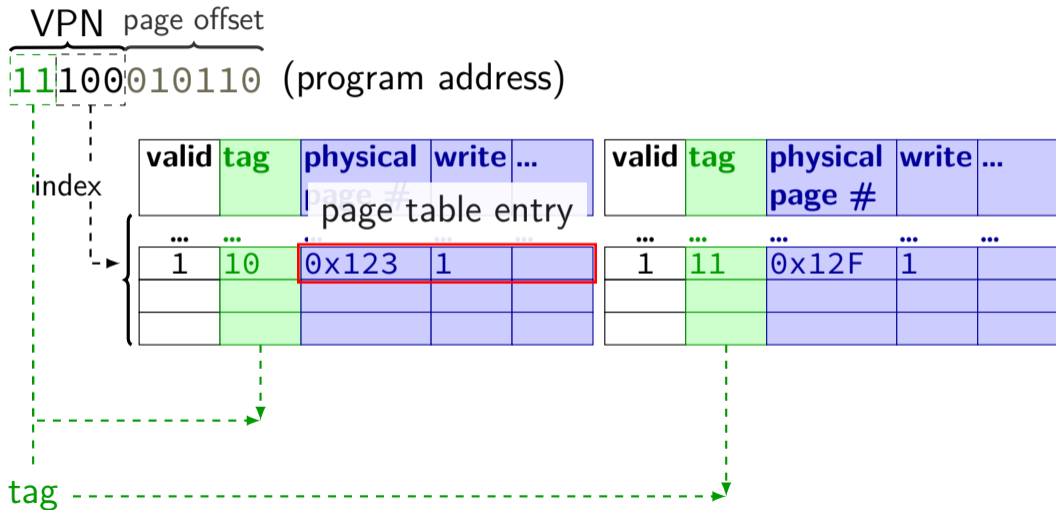
TLB organization (2-way set associative)



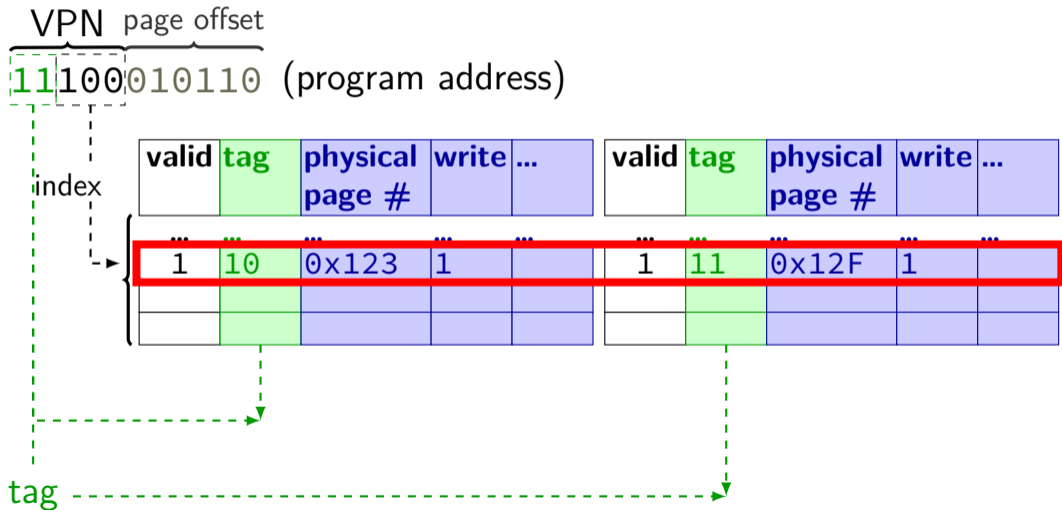
TLB organization (2-way set associative)



TLB organization (2-way set associative)



TLB organization (2-way set associative)



changing page tables

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

e.g. context switch

most entries in TLB refer to things from **wrong process**

oops — read from the wrong process's stack?

changing page tables

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

e.g. context switch

most entries in TLB refer to things from **wrong process**

oops — read from the wrong process's stack?

option 1: **invalidate** all TLB entries

side effect on “change page table base register” instruction

changing page tables

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

e.g. context switch

most entries in TLB refer to things from **wrong process**

oops — read from the wrong process's stack?

option 1: **invalidate** all TLB entries

side effect on “change page table base register” instruction

option 2: TLB entries contain process ID

set by OS (special register)

checked by TLB in addition to TLB tag, valid bit

editing page tables

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

most common choice: has to be handled **in software**

editing page tables

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

most common choice: has to be handled **in software**

invalid to valid — nothing needed

- TLB doesn't contain invalid entries

- MMU will check memory again

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

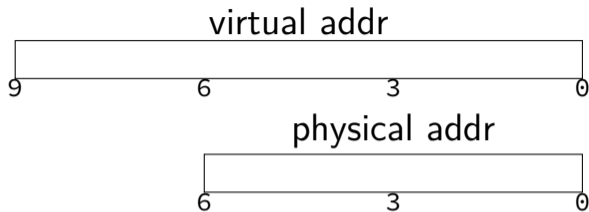
- special instruction (x86: `invlpg`)

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

2-level splitting

9-bit virtual address

6-bit physical address



2-level splitting

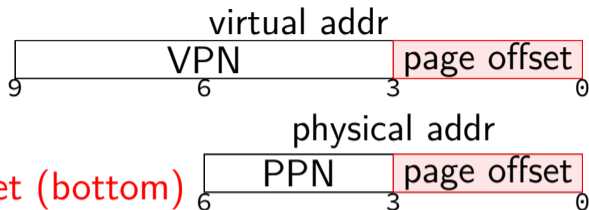
9-bit virtual address

6-bit physical address

8-byte pages \rightarrow 3-bit page offset (bottom)

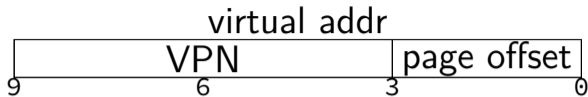
9-bit VA: 6 bit VPN + 3 bit PO

6-bit PA: 3 bit PPN + 3 bit PO

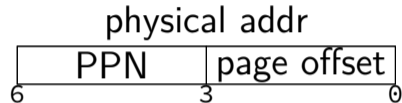


2-level splitting

9-bit virtual address



6-bit physical address



8-byte pages \rightarrow 3-bit page offset (bottom)

9-bit VA: 6 bit VPN + 3 bit PO

6-bit PA: 3 bit PPN + 3 bit PO

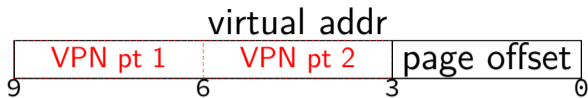
1 page page tables w/ 1 byte entry \rightarrow 8 entry PTs

page table (either level)

	valid? PPN	
0		
1		
2		
...
7		

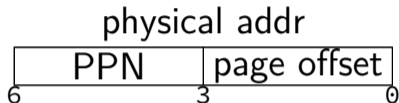
2-level splitting

9-bit virtual address



6-bit physical address

8-byte pages \rightarrow 3-bit page offset (bottom)



9-bit VA: 6 bit VPN + 3 bit PO

page table (either level)

6-bit PA: 3 bit PPN + 3 bit PO

1 page page tables w/ 1 byte entry \rightarrow 8 entry PTs

	valid? PPN	
0		
1		
2		
...
7		

8 entry page tables \rightarrow 3-bit VPN parts

9-bit VA: 3 bit VPN part 1; 3 bit VPN part 2

2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused
page table base register 0x20; translate virtual address 0x129

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	00 91 72 13
0x24-7	F4 A5 36 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	AC DC DC 0C

2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

page table base register $0x20$; translate virtual address $0x129$

physical addresses	bytes
$0x00-3$	00 11 22 33
$0x04-7$	44 55 66 77
$0x08-B$	88 99 AA BB
$0x0C-F$	CC DD EE FF
$0x10-3$	1A 2A 3A 4A
$0x14-7$	1B 2B 3B 4B
$0x18-B$	1C 2C 3C 4C
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	00 91 72 13
$0x24-7$	F4 A5 36 07
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	DB 0B DB 0B
$0x38-B$	EC 0C EC 0C
$0x3C-F$	AC DC DC 0C

$0x129 = 1\ 0010\ 1001$
 $0x20 + 0x4 \times 1 = 0x24$
PTE 1 value:
 $0xF4 = 1111\ 0100$
PPN 111, valid 1

2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

page table base register $0x20$; translate virtual address $0x129$

physical addresses	bytes
$0x00-3$	00 11 22 33
$0x04-7$	44 55 66 77
$0x08-B$	88 99 AA BB
$0x0C-F$	CC DD EE FF
$0x10-3$	1A 2A 3A 4A
$0x14-7$	1B 2B 3B 4B
$0x18-B$	1C 2C 3C 4C
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	00 91 72 13
$0x24-7$	F4 A5 36 07
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	DB 0B DB 0B
$0x38-B$	EC 0C EC 0C
$0x3C-F$	AC DC DC 0C

$0x129 = 1\ 0010\ 1001$
 $0x20 + 0x4 \times 1 = 0x24$

PTE 1 value:

$0xF4 = 1111\ 0100$

PPN 111, valid 1

PTE 2 addr:

$111\ 000 + 101 \times 1 = 0x3D$

PTE 2 value: $0xDC$

2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

page table base register $0x20$; translate virtual address $0x129$

physical addresses	bytes
$0x00-3$	00 11 22 33
$0x04-7$	44 55 66 77
$0x08-B$	88 99 AA BB
$0x0C-F$	CC DD EE FF
$0x10-3$	1A 2A 3A 4A
$0x14-7$	1B 2B 3B 4B
$0x18-B$	1C 2C 3C 4C
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	00 91 72 13
$0x24-7$	F4 A5 36 07
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	DB 0B DB 0B
$0x38-B$	EC 0C EC 0C
$0x3C-F$	AC DC DC 0C

$0x129 = 1\ 0010\ 1001$

$0x20 + 0x4 \times 1 = 0x24$

PTE 1 value:

$0xF4 = 1111\ 0100$

PPN 111, valid 1

PTE 2 addr:

$111\ 000 + 101 \times 1 = 0x3D$

PTE 2 value: $0xDC$

PPN **110**; valid 1

$M[110\ 001\ (0x31)] = 0x0A$

2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

page table base register $0x20$; translate virtual address $0x129$

physical addresses	bytes
$0x00-3$	00 11 22 33
$0x04-7$	44 55 66 77
$0x08-B$	88 99 AA BB
$0x0C-F$	CC DD EE FF
$0x10-3$	1A 2A 3A 4A
$0x14-7$	1B 2B 3B 4B
$0x18-B$	1C 2C 3C 4C
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	00 91 72 13
$0x24-7$	F4 A5 36 07
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	DB 0B DB 0B
$0x38-B$	EC 0C EC 0C
$0x3C-F$	AC DC DC 0C

$0x129 = 1\ 0010\ 1001$

$0x20 + 0x4 \times 1 = 0x24$

PTE 1 value:

$0xF4 = 1111\ 0100$

PPN 111, valid 1

PTE 2 addr:

$111\ 000 + 101 \times 1 = 0x3D$

PTE 2 value: $0xDC$

PPN 110; valid 1

$M[110\ 001\ (0x31)] = 0x0A$

2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused
page table base register 0x20; translate virtual address 0x129

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	00 91 72 13
0x24-7	F4 A5 36 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	AC DC DC 0C

0x129 = 1 0010 1001

0x20 + 0x4 × 1 = 0x24

PTE 1 value:

0xF4 = 1111 0100

PPN 111, valid 1

PTE 2 addr:

111 000 + 101 × 1 = 0x3D

PTE 2 value: 0xDC

PPN 110; valid 1

M[110 001 (0x31)] = 0x0A

2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;
page table base register 0x08; translate virtual address 0x0FB

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;
page table base register $0x08$; translate virtual address $0x0FB$

physical addresses	bytes
$0x00-3$	00 11 22 33
$0x04-7$	44 55 66 77
$0x08-B$	88 99 AA BB
$0x0C-F$	CC DD EE FF
$0x10-3$	1A 2A 3A 4A
$0x14-7$	1B 2B 3B 4B
$0x18-B$	1C 2C 3C 4C
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	D0 D1 D2 D3
$0x24-7$	D4 D5 D6 D7
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	DB 0B DB 0B
$0x38-B$	EC 0C EC 0C
$0x3C-F$	FC 0C FC 0C

$0x0F3 = 011\ 111\ 011$
(PTE 1 addr: $0x08 +$
PTE size times $011\ (3)$)
PTE 1: $0xBB$ at $0x0B$
PTE 1: PPN $101\ (5)$ valid 1
PTE 2: $0xF0$ at $0x2F$
PTE 2: PPN $111\ (7)$ valid 1
 $111\ 011 = 0x3B \rightarrow 0x0C$

2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;
page table base register $0x08$; translate virtual address $0x0FB$

physical addresses	bytes
$0x00-3$	00 11 22 33
$0x04-7$	44 55 66 77
$0x08-B$	88 99 AA BB
$0x0C-F$	CC DD EE FF
$0x10-3$	1A 2A 3A 4A
$0x14-7$	1B 2B 3B 4B
$0x18-B$	1C 2C 3C 4C
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	D0 D1 D2 D3
$0x24-7$	D4 D5 D6 D7
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	DB 0B DB 0B
$0x38-B$	EC 0C EC 0C
$0x3C-F$	FC 0C FC 0C

$0x0F3 = 011\ 111\ 011$
(PTE 1 addr: $0x08 +$
PTE size times $011\ (3)$)
*PTE 1: **0xBB** at $0x0B$*
PTE 1: PPN $101\ (5)$ valid 1
PTE 2: $0xF0$ at $0x2F$
PTE 2: PPN $111\ (7)$ valid 1
 $111\ 011 = 0x3B \rightarrow 0x0C$

2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;
page table base register 0x08; translate virtual address 0x0FB

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

0x0F3 = 011 111 011
(PTE 1 addr: 0x08 +
PTE size times 011 (3))
PTE 1: 0xBB at 0x0B
PTE 1: PPN 101 (5) valid 1
PTE 2: 0xF0 at 0x2F
PTE 2: PPN 111 (7) valid 1
111 011 = 0x3B → 0x0C

2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;
page table base register 0x08; translate virtual address 0x0FB

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

0x0F3 = 011 111 011
(PTE 1 addr: 0x08 +
PTE size times 011 (3))
PTE 1: 0xBB at 0x0B
PTE 1: PPN 101 (5) valid 1
PTE 2: 0xF0 at 0x2F
PTE 2: PPN 111 (7) valid 1
111 011 = 0x3B → 0x0C

2-level exercise (2)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;
page table base register 0x10; translate virtual address 0x109

physical
addresses bytes

0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 5A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical
addresses bytes

0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

2-level exercise (2)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;
page table base register 0x10; translate virtual address 0x109

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 5A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

0x109 = 100 011 001
(PTE 1 at:
0x10 + PTE size times 4 (100))
PTE 1: 0x1B at 0x14
PTE 1: PPN 000 (0) valid 1
(second table at:
0 (000) times page size = 0x00)
PTE 2: 0x33 at 0x03
PTE 2: PPN 001 (1) valid 1
001 001 = 0x09 → 0x99

2-level exercise (2)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;

page table base register $0x10$; translate virtual address $0x109$

physical addresses	bytes
$0x00-3$	00 11 22 33
$0x04-7$	44 55 66 77
$0x08-B$	88 99 AA BB
$0x0C-F$	CC DD EE FF
$0x10-3$	1A 2A 5A 4A
$0x14-7$	1B 2B 3B 4B
$0x18-B$	1C 2C 3C 4C
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	D0 D1 D2 D3
$0x24-7$	D4 D5 D6 D7
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	DB 0B DB 0B
$0x38-B$	EC 0C EC 0C
$0x3C-F$	FC 0C FC 0C

$0x109 = 100\ 011\ 001$
(PTE 1 at:
 $0x10 + \text{PTE size times } 4 (100))$
PTE 1: **0x1B** at $0x14$
PTE 1: PPN 000 (0) valid 1
(second table at:
0 (000) times page size = $0x00$)
PTE 2: $0x33$ at $0x03$
PTE 2: PPN 001 (1) valid 1
 $001\ 001 = 0x09 \rightarrow 0x99$

2-level exercise (2)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;

page table base register $0x10$; translate virtual address $0x109$

physical addresses	bytes
$0x00-3$	00 11 22 33
$0x04-7$	44 55 66 77
$0x08-B$	88 99 AA BB
$0x0C-F$	CC DD EE FF
$0x10-3$	1A 2A 5A 4A
$0x14-7$	1B 2B 3B 4B
$0x18-B$	1C 2C 3C 4C
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	D0 D1 D2 D3
$0x24-7$	D4 D5 D6 D7
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	DB 0B DB 0B
$0x38-B$	EC 0C EC 0C
$0x3C-F$	FC 0C FC 0C

$0x109 = 100\ 011\ 001$
(PTE 1 at:
 $0x10 + \text{PTE size times } 4 (100))$
PTE 1: $0x1B$ at $0x14$
PTE 1: PPN $000 (0)$ valid 1
(second table at:
 $0 (000)$ times page size = $0x00$)
PTE 2: $0x33$ at $0x03$
PTE 2: PPN $001 (1)$ valid 1
 $001\ 001 = 0x09 \rightarrow 0x99$

2-level exercise (2)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;

page table base register $0x10$; translate virtual address $0x109$

physical addresses	bytes
$0x00-3$	00 11 22 33
$0x04-7$	44 55 66 77
$0x08-B$	88 99 AA BB
$0x0C-F$	CC DD EE FF
$0x10-3$	1A 2A 5A 4A
$0x14-7$	1B 2B 3B 4B
$0x18-B$	1C 2C 3C 4C
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	D0 D1 D2 D3
$0x24-7$	D4 D5 D6 D7
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	DB 0B DB 0B
$0x38-B$	EC 0C EC 0C
$0x3C-F$	FC 0C FC 0C

$0x109 = 100\ 011\ 001$
(PTE 1 at:
 $0x10 + \text{PTE size times } 4 (100)$)
PTE 1: $0x1B$ at $0x14$
PTE 1: PPN $000 (0)$ valid 1
(second table at:
 $0 (000)$ times page size = $0x00$)
PTE 2: $0x33$ at $0x03$
PTE 2: PPN $001 (1)$ valid 1
 $001\ 001 = 0x09 \rightarrow 0x99$

2-level exercise (3)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused
page table base register 0x08; translate virtual address 0x00B

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

2-level exercise (3)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused
page table base register 0x08; translate virtual address 0x00B

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

0x0F3 = 000 001 011

PTE 1: 0x88 at 0x08

PTE 1: PPN 100 (5) valid 0
page fault!

2-level exercise (3)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused
page table base register $0x08$; translate virtual address $0x00B$

physical addresses	bytes
$0x00-3$	00 11 22 33
$0x04-7$	44 55 66 77
$0x08-B$	88 99 AA BB
$0x0C-F$	CC DD EE FF
$0x10-3$	1A 2A 3A 4A
$0x14-7$	1B 2B 3B 4B
$0x18-B$	1C 2C 3C 4C
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	D0 D1 D2 D3
$0x24-7$	D4 D5 D6 D7
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	DB 0B DB 0B
$0x38-B$	EC 0C EC 0C
$0x3C-F$	FC 0C FC 0C

$0x0F3 = 000\ 001\ 011$

PTE 1: $0x88$ at $0x08$

PTE 1: PPN 100 (5) valid 0
page fault!

2-level exercise (4)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused
page table base register 0x08; translate virtual address 0x1CB

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

2-level exercise (4)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused
page table base register 0x08; translate virtual address 0x1CB

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

0x1CB = 111 001 011
PTE 1: 0xFF at 0x0F
PTE 1: PPN 111 (7) valid 1
PTE 2: 0x0C at 0x39
PTE 2: PPN 000 (0) valid 0
page fault!

2-level exercise (4)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused
page table base register 0x08; translate virtual address 0x1CB

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

0x1CB = 111 001 011
PTE 1: 0xFF at 0x0F
PTE 1: PPN 111 (7) valid 1
PTE 2: 0x0C at 0x39
PTE 2: PPN 000 (0) valid 0
page fault!

2-level exercise (4)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused
page table base register 0x08; translate virtual address 0x1CB

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

0x1CB = 111 001 011
PTE 1: 0xFF at 0x0F
PTE 1: PPN 111 (7) valid 1
PTE 2: 0x0C at 0x39
PTE 2: PPN 000 (0) valid 0
page fault!

2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

page table base register 0x10; translate virtual address 0x376

physical
addresses bytes

0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	AC BC DC EC

physical
addresses bytes

0x20-3	D0 E1 D2 D3
0x24-7	D4 E5 D6 E7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

page table base register $0x10$; translate virtual address $0x376$

physical addresses	bytes
$0x00-3$	00 11 22 33
$0x04-7$	44 55 66 77
$0x08-B$	88 99 AA BB
$0x0C-F$	CC DD EE FF
$0x10-3$	1A 2A 3A 4A
$0x14-7$	1B 2B 3B 4B
$0x18-B$	1C 2C 3C 4C
$0x1C-F$	AC BC DC EC

physical addresses	bytes
$0x20-3$	D0 E1 D2 D3
$0x24-7$	D4 E5 D6 E7
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	DB 0B DB 0B
$0x38-B$	EC 0C EC 0C
$0x3C-F$	FC 0C FC 0C

$0x376 = 110\ 111\ 0110$

PTE 1: $0x10 + 6 \times 2 = 0x1C$:
AC BC

PTE 1: PPN 10 valid 1

PTE 2: $0x20 + 7 \times 2 = 0x2E$:
EF F0

PTE 2: PPN 11 valid 1

11 0110 = $0x36 \rightarrow$ DB

2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

page table base register $0x10$; translate virtual address $0x376$

physical addresses	bytes
$0x00-3$	00 11 22 33
$0x04-7$	44 55 66 77
$0x08-B$	88 99 AA BB
$0x0C-F$	CC DD EE FF
$0x10-3$	1A 2A 3A 4A
$0x14-7$	1B 2B 3B 4B
$0x18-B$	1C 2C 3C 4C
$0x1C-F$	AC BC DC EC

physical addresses	bytes
$0x20-3$	D0 E1 D2 D3
$0x24-7$	D4 E5 D6 E7
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	DB 0B DB 0B
$0x38-B$	EC 0C EC 0C
$0x3C-F$	FC 0C FC 0C

$0x376 = 110\ 111\ 0110$

PTE 1: $0x10 + 6 \times 2 = 0x1C$:
AC BC

PTE 1: PPN 10 valid 1

PTE 2: $0x20 + 7 \times 2 = 0x2E$:
EF F0

PTE 2: PPN 11 valid 1

11 0110 = $0x36 \rightarrow DB$

2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

page table base register $0x10$; translate virtual address $0x376$

physical addresses	bytes
$0x00-3$	00 11 22 33
$0x04-7$	44 55 66 77
$0x08-B$	88 99 AA BB
$0x0C-F$	CC DD EE FF
$0x10-3$	1A 2A 3A 4A
$0x14-7$	1B 2B 3B 4B
$0x18-B$	1C 2C 3C 4C
$0x1C-F$	AC BC DC EC

physical addresses	bytes
$0x20-3$	D0 E1 D2 D3
$0x24-7$	D4 E5 D6 E7
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	DB 0B DB 0B
$0x38-B$	EC 0C EC 0C
$0x3C-F$	FC 0C FC 0C

$0x376 = 110\ 111\ 0110$

PTE 1: $0x10 + 6 \times 2 = 0x1C$:

AC BC

PTE 1: PPN 10 valid 1

PTE 2: $0x20 + 7 \times 2 = 0x2E$:

EF F0

PTE 2: PPN 11 valid 1

11 0110 = $0x36 \rightarrow DB$

2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

page table base register $0x10$; translate virtual address $0x376$

physical addresses	bytes
$0x00-3$	00 11 22 33
$0x04-7$	44 55 66 77
$0x08-B$	88 99 AA BB
$0x0C-F$	CC DD EE FF
$0x10-3$	1A 2A 3A 4A
$0x14-7$	1B 2B 3B 4B
$0x18-B$	1C 2C 3C 4C
$0x1C-F$	AC BC DC EC

physical addresses	bytes
$0x20-3$	D0 E1 D2 D3
$0x24-7$	D4 E5 D6 E7
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	DB 0B DB 0B
$0x38-B$	EC 0C EC 0C
$0x3C-F$	FC 0C FC 0C

$0x376 = 110 \text{ } 111 \text{ } 0110$

PTE 1: $0x10 + 6 \times 2 = 0x1C$:
AC BC

PTE 1: PPN 10 valid 1

PTE 2: $0x20 + 7 \times 2 = 0x2E$:
EF F0

PTE 2: PPN 11 valid 1
 $11 \text{ } 0110 = 0x36 \rightarrow \text{DB}$

2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

page table base register $0x10$; translate virtual address $0x376$

physical addresses	bytes
$0x00-3$	00 11 22 33
$0x04-7$	44 55 66 77
$0x08-B$	88 99 AA BB
$0x0C-F$	CC DD EE FF
$0x10-3$	1A 2A 3A 4A
$0x14-7$	1B 2B 3B 4B
$0x18-B$	1C 2C 3C 4C
$0x1C-F$	AC BC DC EC

physical addresses	bytes
$0x20-3$	D0 E1 D2 D3
$0x24-7$	D4 E5 D6 E7
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	DB 0B DB 0B
$0x38-B$	EC 0C EC 0C
$0x3C-F$	FC 0C FC 0C

$0x376 = 110\ 111\ 0110$

PTE 1: $0x10 + 6 \times 2 = 0x1C$:
AC BC

PTE 1: PPN 10 valid 1

PTE 2: $0x20 + 7 \times 2 = 0x2E$:
EF F0

PTE 2: PPN 11 valid 1

11 0110 = $0x36 \rightarrow$ DB

2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

page table base register $0x10$; translate virtual address $0x376$

physical addresses	bytes
$0x00-3$	00 11 22 33
$0x04-7$	44 55 66 77
$0x08-B$	88 99 AA BB
$0x0C-F$	CC DD EE FF
$0x10-3$	1A 2A 3A 4A
$0x14-7$	1B 2B 3B 4B
$0x18-B$	1C 2C 3C 4C
$0x1C-F$	AC BC DC EC

physical addresses	bytes
$0x20-3$	D0 E1 D2 D3
$0x24-7$	D4 E5 D6 E7
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	DB 0B DB 0B
$0x38-B$	EC 0C EC 0C
$0x3C-F$	FC 0C FC 0C

$0x376 = 110\ 111\ 0110$

PTE 1: $0x10 + 6 \times 2 = 0x1C$:
AC BC

PTE 1: PPN 10 valid 1

PTE 2: $0x20 + 7 \times 2 = 0x2E$:
EF F0

PTE 2: PPN 11 valid 1

11 $0110 = 0x36 \rightarrow$ DB

2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

page table base register $0x10$; translate virtual address $0x376$

physical addresses	bytes
$0x00-3$	00 11 22 33
$0x04-7$	44 55 66 77
$0x08-B$	88 99 AA BB
$0x0C-F$	CC DD EE FF
$0x10-3$	1A 2A 3A 4A
$0x14-7$	1B 2B 3B 4B
$0x18-B$	1C 2C 3C 4C
$0x1C-F$	AC BC DC EC

physical addresses	bytes
$0x20-3$	D0 E1 D2 D3
$0x24-7$	D4 E5 D6 E7
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	DB 0B DB 0B
$0x38-B$	EC 0C EC 0C
$0x3C-F$	FC 0C FC 0C

$0x376 = 110\ 111\ 0110$

PTE 1: $0x10 + 6 \times 2 = 0x1C$:
AC BC

PTE 1: PPN 10 valid 1

PTE 2: $0x20 + 7 \times 2 = 0x2E$:
EF F0

PTE 2: PPN 11 valid 1

$11\ 0110 = 0x36 \rightarrow$ **DB**