# 2004 CPU

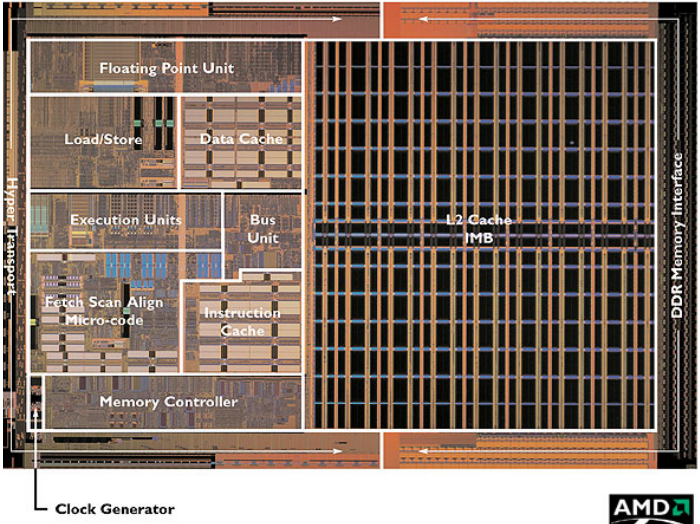
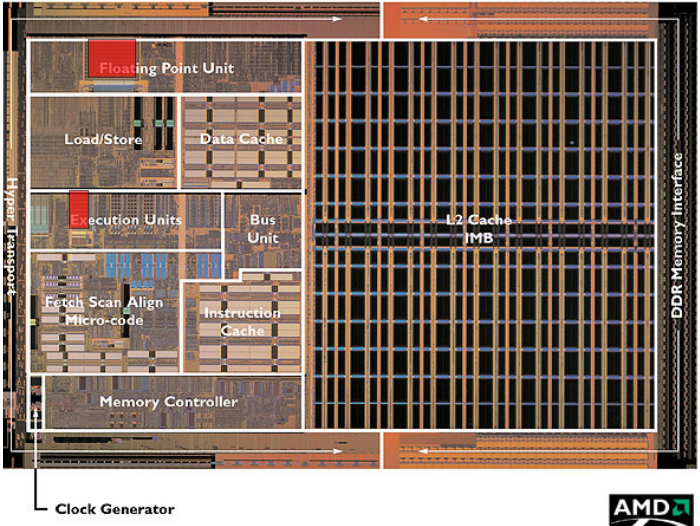



Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

# 2004 CPU




Registers

AMD 64 Opteron

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

# 2004 CPU

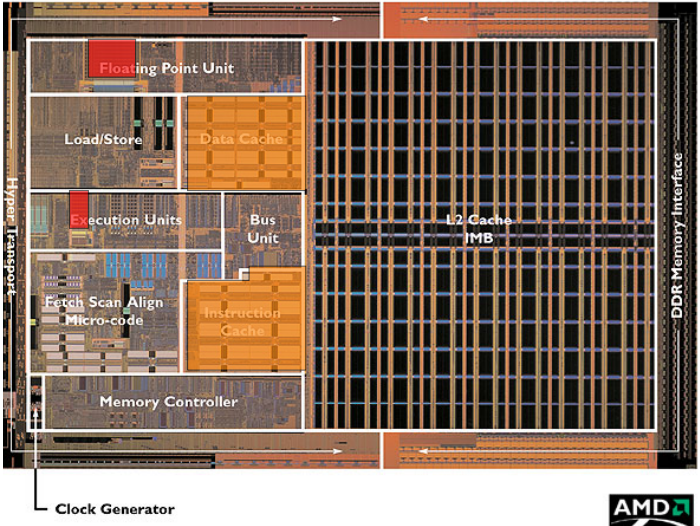
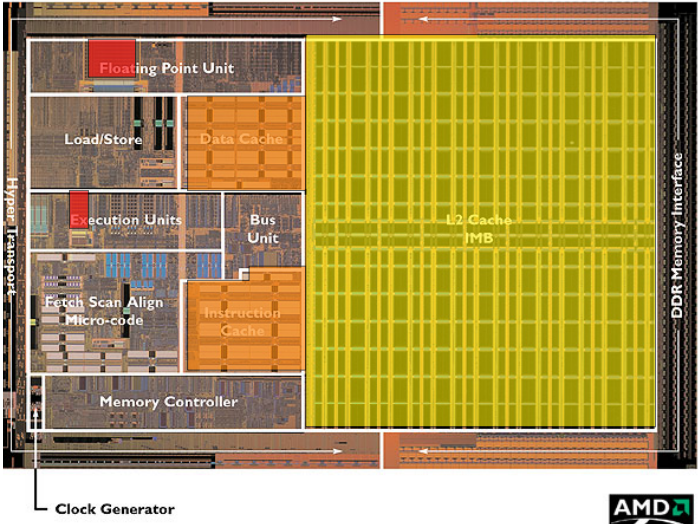
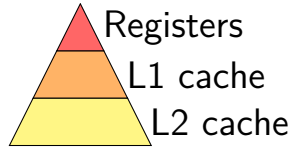




Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

# 2004 CPU

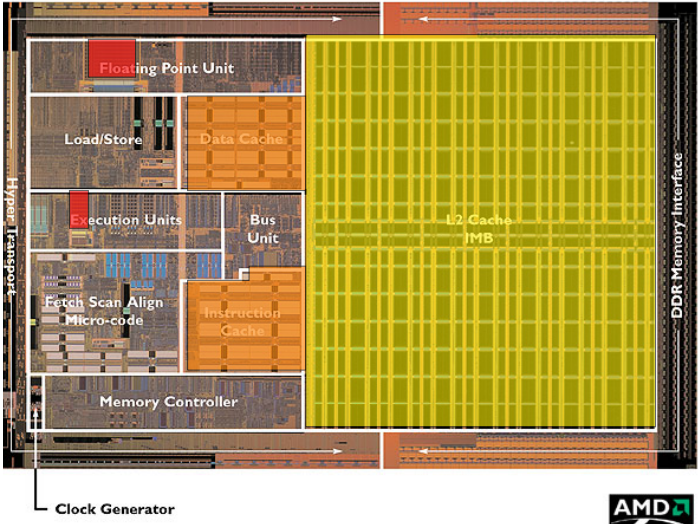


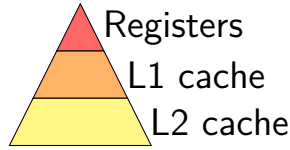




AMD 64 Opteron

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

# 2004 CPU

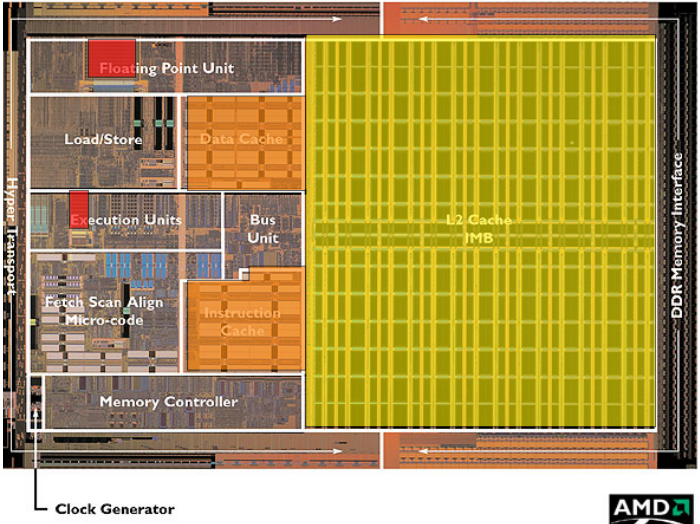



AMD 64 Opteron

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

# 2004 CPU



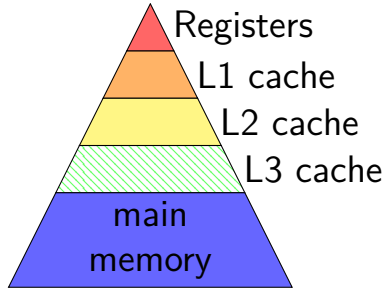
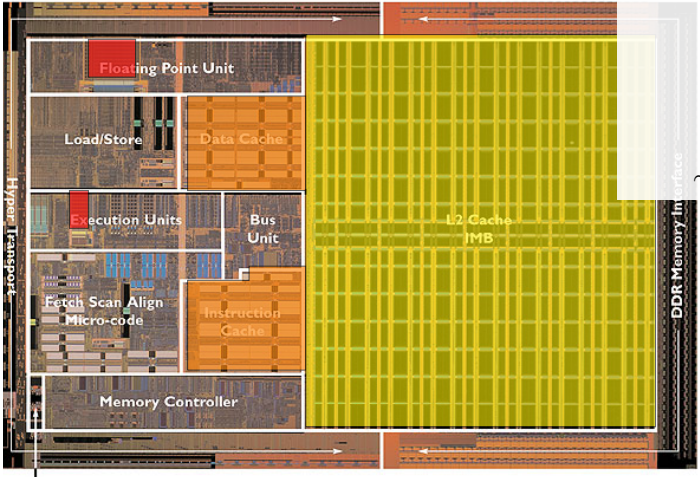




Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

# 2004 CPU



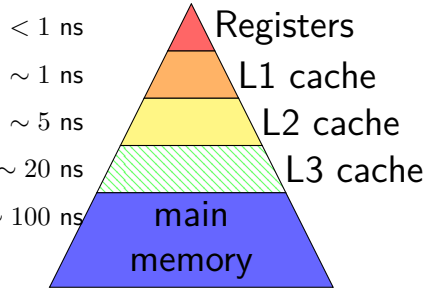
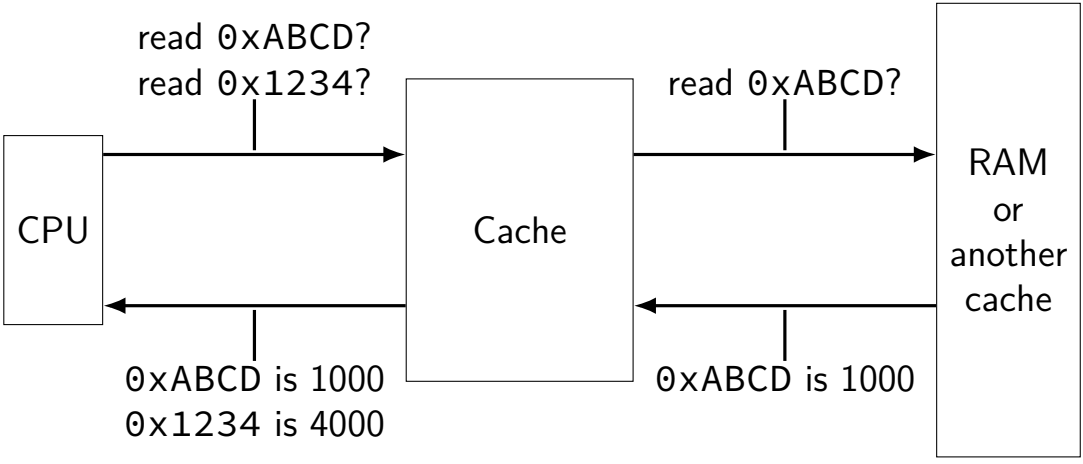




Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

# 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;
}
```

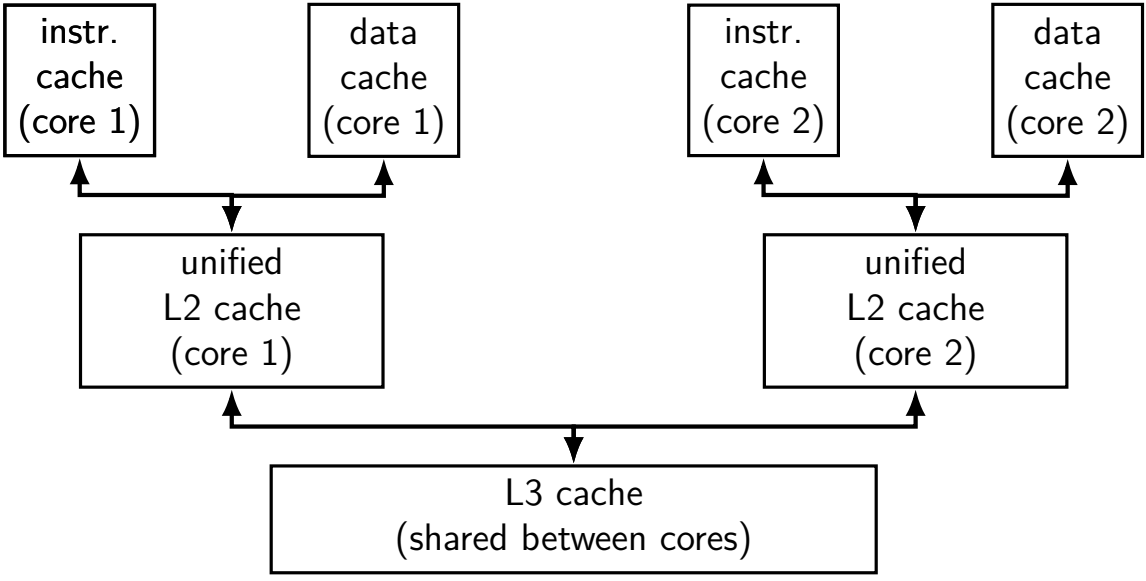
temporal locality: machine code of the loop

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

| value |
| --- |
| 00 00 |

Memory

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

# one-block cache

read byte at 01011?

Cache

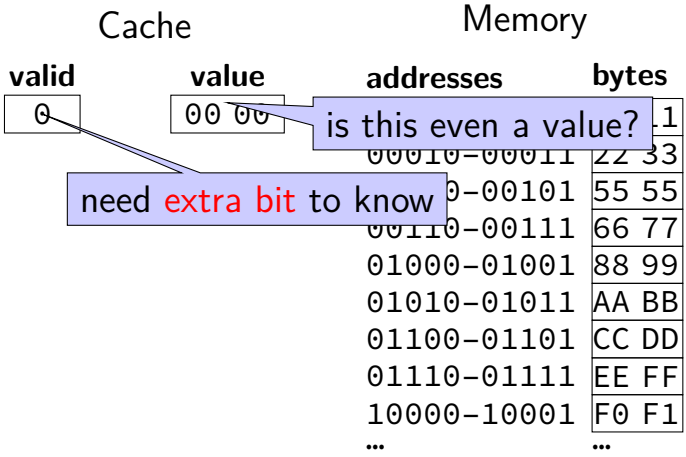
| value |
| --- |
| 00 00 |

Memory

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

# one-block cache

read byte at 01011?



# one-block cache

read byte at 01011?

invalid, fetch

Cache

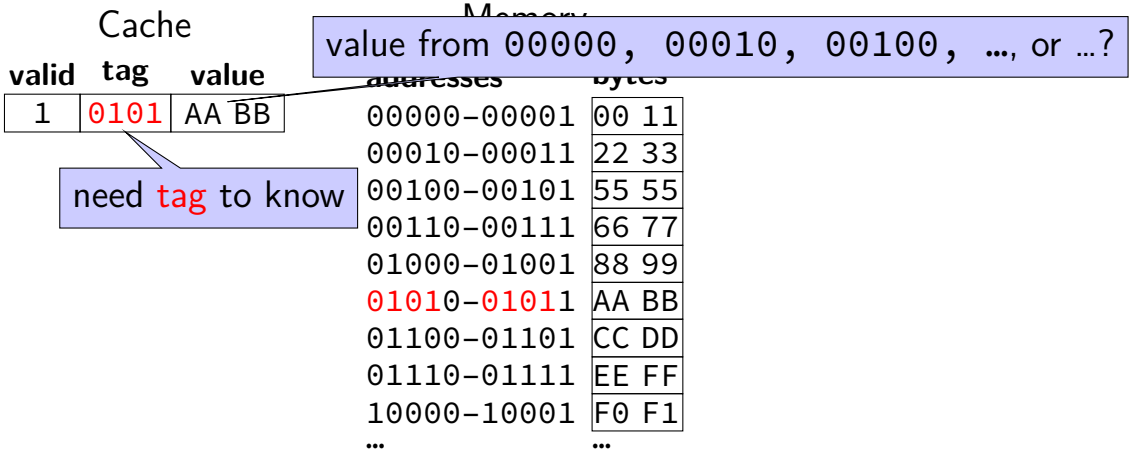
| valid | value |
|---|---|
| 1 | AA BB |

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

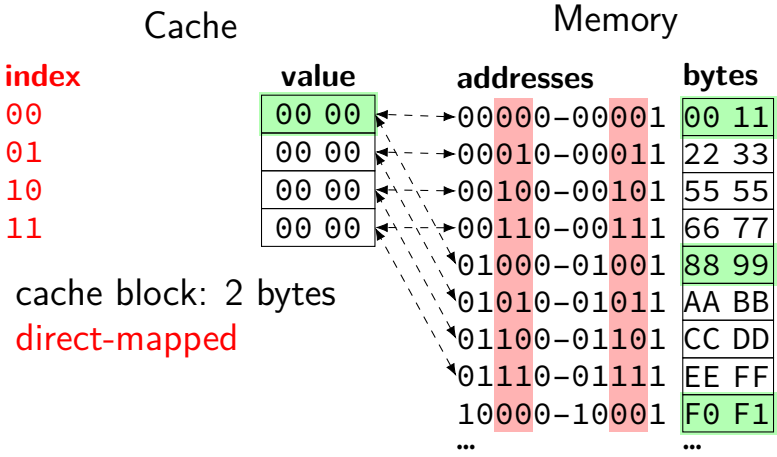
Memory

| 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?

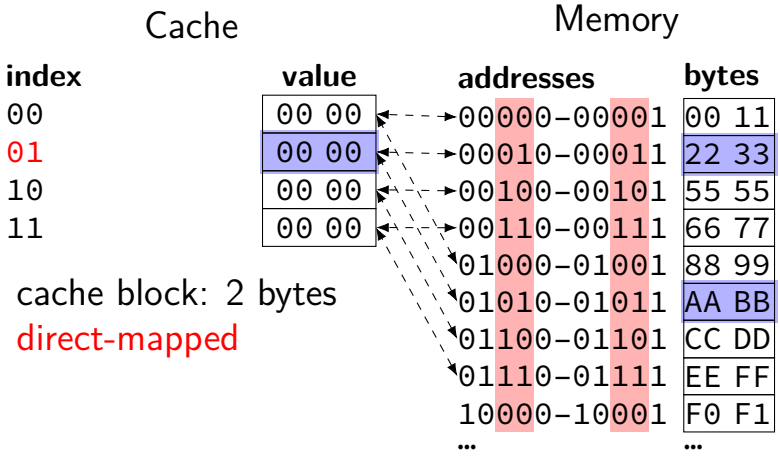
exactly one place for each address
spread out what can go in a block



# building a (direct-mapped) cache

read byte at 01011?

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



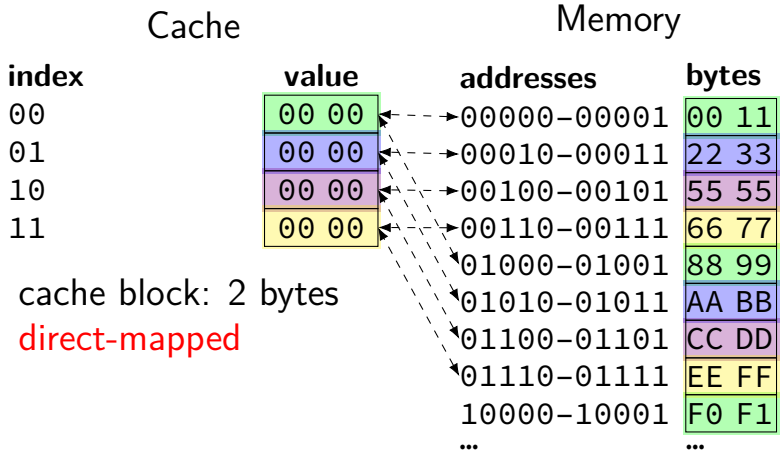

cache block: 2 bytes

direct-mapped

# building a (direct-mapped) cache

read byte at 01011?

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

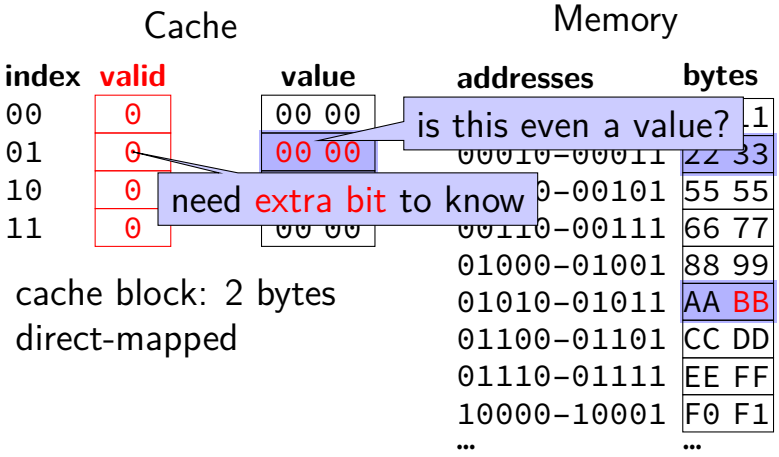



cache block: 2 bytes

direct-mapped

# building a (direct-mapped) cache

read byte at 01011?



# 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

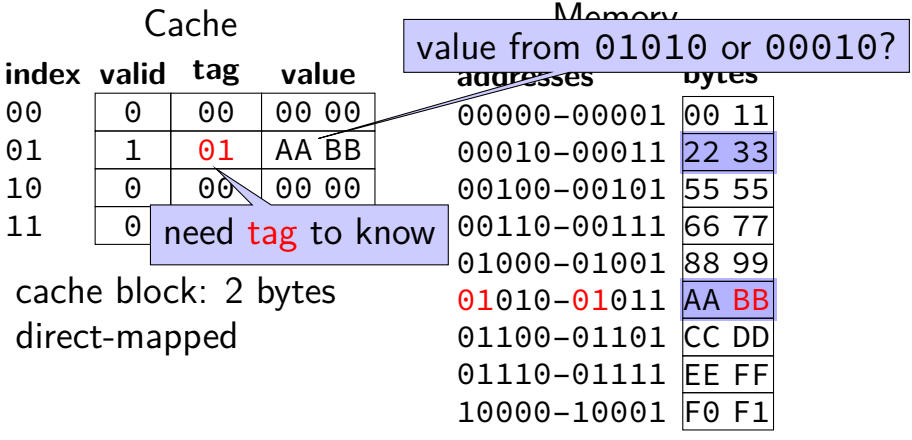
Memory

| 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



# 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 | | | |
| 2 byte blocks, 8 sets | | | |
| 4 byte blocks, 2 sets | | | |

2 byte blocks, 4 sets

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

4 byte blocks, 2 sets

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

2 byte blocks, 8 sets

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

# Tag-Index-Offset (TIO)

address `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 | | | |

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 | 000 | 00 11 |
| 001 | 1 | 001 | F1 F2 |
| 010 | 0 | -- | -- -- |
| 011 | 0 | -- | -- -- |
| 100 | 0 | -- | -- -- |
| 101 | 1 | 00 | AA BB |
| 110 | 0 | -- | -- -- |
| 111 | 1 | 00 | EE FF |

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

# 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 blocks, 2 sets

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

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

2 byte blocks, 8 sets

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

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

# 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 | | |

4 byte blocks, 2 sets

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

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

2 byte blocks, 8 sets

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

# Tag-Index-Offset (TIO)

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

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

2 byte blocks, 4 sets

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

4 byte blocks, 2 sets

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

2 byte blocks, 8 sets

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

$2^1 = 2$ sets

1 bit to index set

# Tag-Index-Offset (TIO)

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

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

tag — whatever is left over

| | | | |
|---|---|---|---|
| 00 | 1 | 000 | 00 11 |
| 01 | 1 | 001 | AA BB |
| 10 | 0 | -- | -- -- |
| 11 | 1 | 001 | EE FF |

4 byte blocks, 2 sets

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

2 byte blocks, 8 sets

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

# cache size

cache size = amount of *data* in cache

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

# Tag-Index-Offset formulas (direct-mapped)

(formulas derivable from prior slides)

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

# Tag-Index-Offset formulas (direct-mapped)

(formulas derivable from prior slides)

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

# TIO: exercise

64-byte blocks, 128 set cache

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

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

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

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

# example access pattern (1)

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

2 byte blocks, 4 sets

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

# example access pattern (1)

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

2 byte blocks, 4 sets

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

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

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

# example access pattern (1)

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

tag index offset

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

2 byte blocks, 4 sets

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

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

# example access pattern (1)

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

tag index offset

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

2 byte blocks, 4 sets

| index | valid | tag | value |
|---|---|---|---|
| 00 | 1 | 00000 | mem[0x00]<br>mem[0x01] |
| 01 | 0 | | |
| 10 | 0 | | |
| 11 | 0 | | |

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

# example access pattern (1)

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

tag index offset

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

2 byte blocks, 4 sets

| index | valid | tag | value |
|---|---|---|---|
| 00 | 1 | 00000 | mem[0x00]<br>mem[0x01] |
| 01 | 0 | | |
| 10 | 0 | | |
| 11 | 0 | | |

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

# example access pattern (1)

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

tag index offset

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

2 byte blocks, 4 sets

| index | valid | tag | value |
|---|---|---|---|
| 00 | 1 | 00000 | mem[0x00]<br>mem[0x01] |
| 01 | 1 | 01100 | mem[0x62]<br>mem[0x63] |
| 10 | 0 | | |
| 11 | 0 | | |

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

# example access pattern (1)

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

tag index offset

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

2 byte blocks, 4 sets

| index | valid | tag | value |
|---|---|---|---|
| 00 | 1 | 01100 | mem[0x60]<br>mem[0x61] |
| 01 | 1 | 01100 | mem[0x62]<br>mem[0x63] |
| 10 | 0 | | |
| 11 | 0 | | |

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

# example access pattern (1)

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

tag index offset

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

2 byte blocks, 4 sets

| index | valid | tag | value |
|---|---|---|---|
| 00 | 1 | 01100 | mem[0x60]<br>mem[0x61] |
| 01 | 1 | 01100 | mem[0x62]<br>mem[0x63] |
| 10 | 0 | | |
| 11 | 0 | | |

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

# example access pattern (1)

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

tag index offset

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

2 byte blocks, 4 sets

| index | valid | tag | value |
|---|---|---|---|
| 00 | 1 | 00000 | mem[0x00]<br>mem[0x01] |
| 01 | 1 | 01100 | mem[0x62]<br>mem[0x63] |
| 10 | 0 | | |
| 11 | 0 | | |

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

# example access pattern (1)

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

tag index offset

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

2 byte blocks, 4 sets

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

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

# example access pattern (1)

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

tag index offset

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

2 byte blocks, 4 sets

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

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

# example access pattern (1)

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

tag index offset

2 byte blocks, 4 sets

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

miss caused by conflict

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

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

# exercise

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

4 byte blocks, 4 sets

| index | valid | tag | value |
|---|---|---|---|
| 00 | | | |
| 01 | | | |
| 10 | | | |
| 11 | | | |

# exercise

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

4 byte blocks, 4 sets

| index | valid | tag | value |
|---|---|---|---|
| 00 | | | |
| 01 | | | |
| 10 | | | |
| 11 | | | |

how is the 8-bit address 61 (01100001) split up into tag/index/offset?

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

# exercise

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

4 byte blocks, 4 sets

| index | valid | tag | value |
|---|---|---|---|
| 00 | | | |
| 01 | | | |
| 10 | | | |
| 11 | | | |

# exercise

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

4 byte blocks, 4 sets

| index | valid | tag | value |
|---|---|---|---|
| 00 | | | |
| 01 | | | |
| 10 | | | |
| 11 | | | |

# exercise

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

4 byte blocks, 4 sets

| index | valid | tag | value |
|---|---|---|---|
| 00 | | | |
| 01 | | | |
| 10 | | | |
| 11 | | | |

exercise: which accesses are hits?

# cache accesses and C code (1)

```
int scaleFactor;

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

---

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

---

exericse: what data cache accesses does this function do?

# cache accesses and C code (1)

```
int scaleFactor;

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

---

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

---

exericse: what data cache accesses does this function do?

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

# possible scaleFactor use

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

# misses and code (2)

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

suppose each time this is called in the loop:

return address located at address `0x7ffffffe43b8`

scaleFactor located at address `0x6bc3a0`

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

| | return address | scaleFactor |
|---|---|---|
| tag | | |
| index | | |
| offset | | |

# misses and code (2)

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

suppose each time this is called in the loop:

return address located at address `0x7ffffffe43b8`

scaleFactor located at address `0x6bc3a0`

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

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

# misses and code (2)

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

suppose each time this is called in the loop:

return address located at address `0x7ffffffe43b8`

scaleFactor located at address `0x6bc3a0`

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

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

# conflict miss coincidences?

obviously I set that up to have the same index

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

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

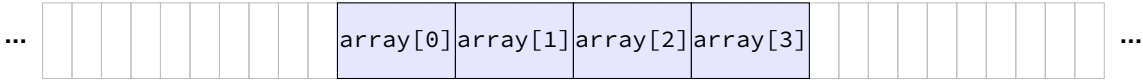
# C and cache misses (warmup 1)

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

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

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

# some possiblities




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

# aside: alignment

compilers and malloc/new implementations usually try align values

align = make address be multiple of something

most important reason: don't cross cache block boundaries

# C and cache misses (warmup 2)

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

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

Assume array[0] at beginning of cache block.

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

# C and cache misses (warmup 3)

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

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

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

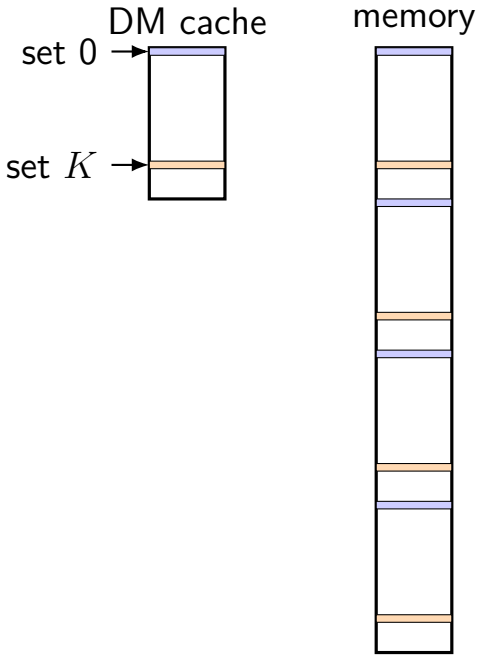
# C and cache misses (warmup 4)

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

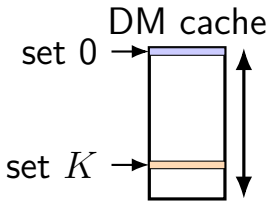
Assume everything but `array` is kept in registers (and the compiler does not do anything funny).

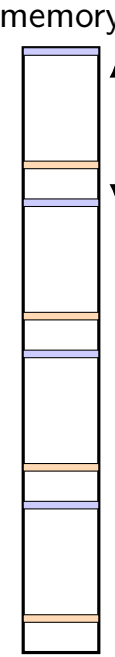
How many data cache misses on a **2**-set direct-mapped cache with 8B blocks?

# mapping of sets to memory (direct-mapped)



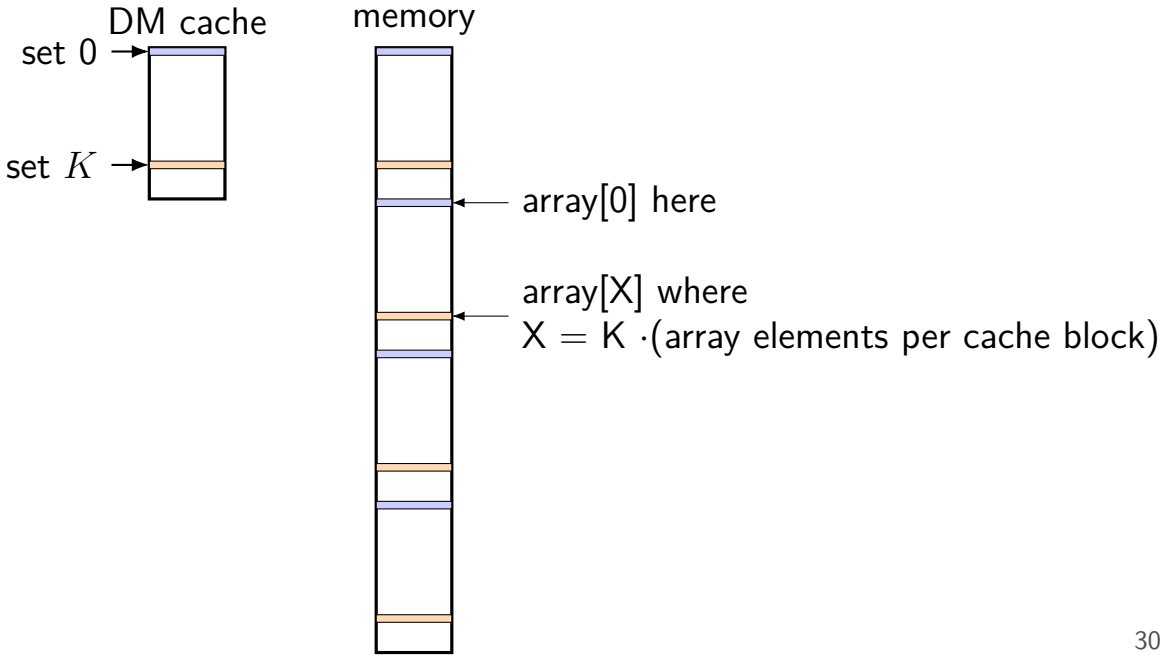
# mapping of sets to memory (direct-mapped)



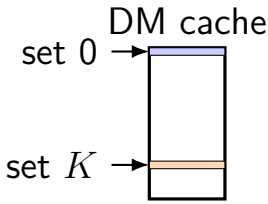



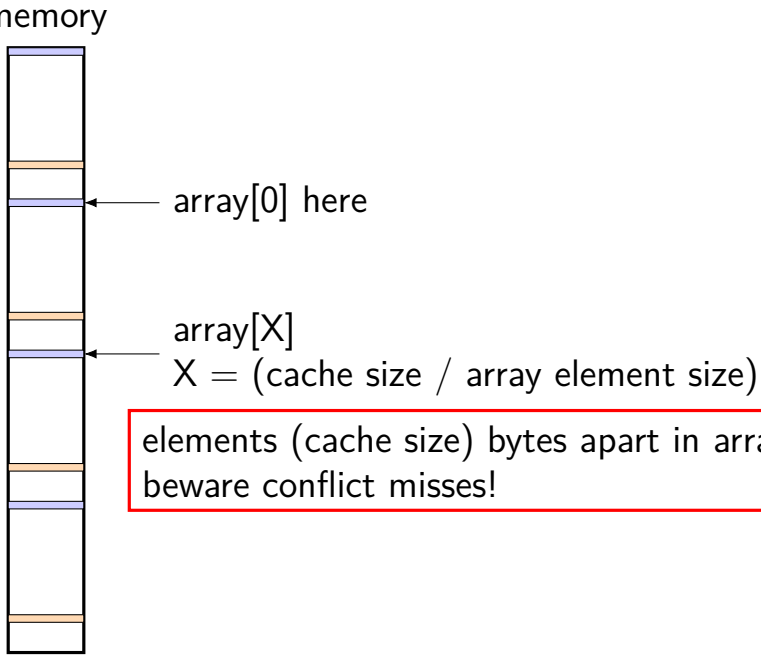

values which would be stored in same set (cache size) bytes apart

# mapping of sets to memory (direct-mapped)



# mapping of sets to memory (direct-mapped)



# C and cache misses (warmup 5)

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

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

observation: array[0] and array[512] exactly 2KB apart

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

# C and cache misses (warmup 6)

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

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

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

# misses with skipping

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

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

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

# best/worst case

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

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

2 misses every 4 `i`

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

(and same for array2[X])

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

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

2 misses every `i`

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

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

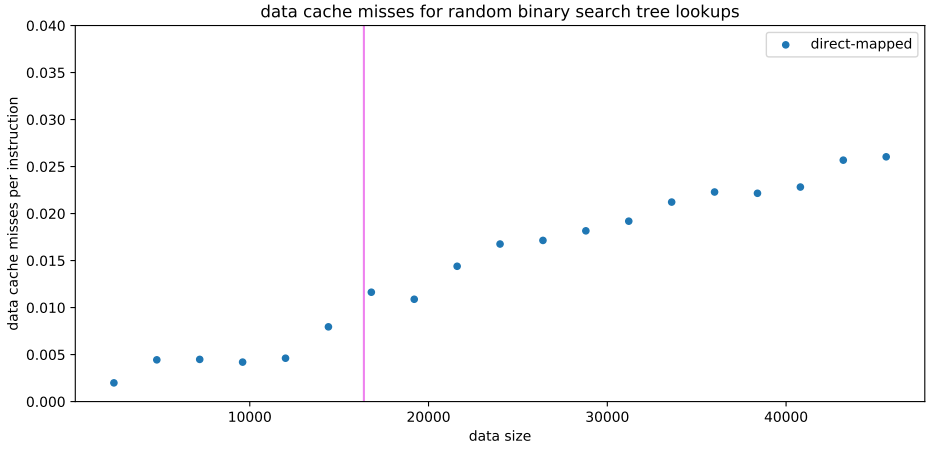
# worst case in practice?

two rows of matrix?

often sizeof(row) bytes apart

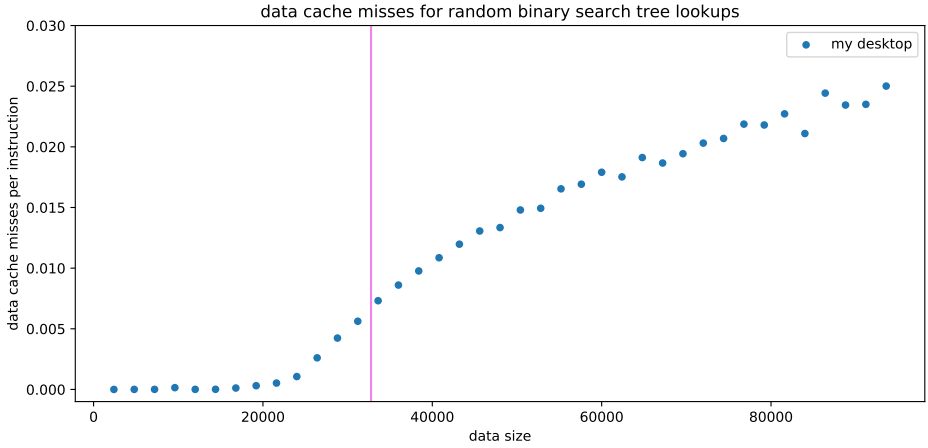
if the row size is multiple of number of sets $\times$ bytes per block, oops!

# simulated misses: BST lookups



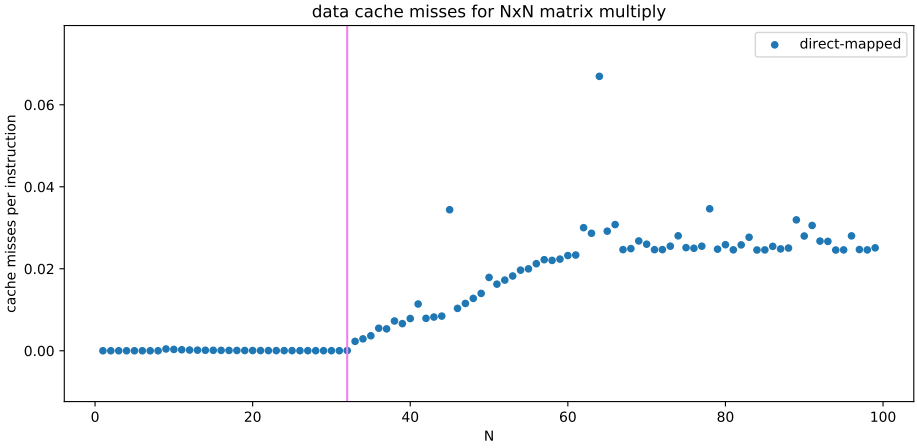

(simulated 16KB direct-mapped data cache; excluding BST setup)

# actual misses: BST lookups



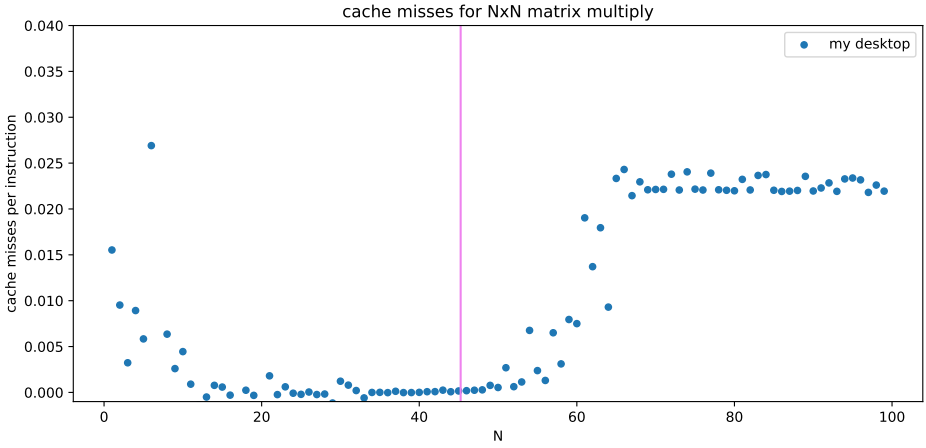

(actual 32KB more complex data cache)
(only one set of measurements + other things on machine + excluding initial load)

# simulated misses: matrix multiplies




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

# actual misses: matrix multiplies




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

# adding associativity

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

| index | valid | tag | value | valid | tag | value |
|---|---|---|---|---|---|---|
| 0 | 0 | | | 0 | | |
| 1 | 0 | | | 0 | | |

multiple places to put values with same index
avoid misses from two active values using same set
("conflict misses"))

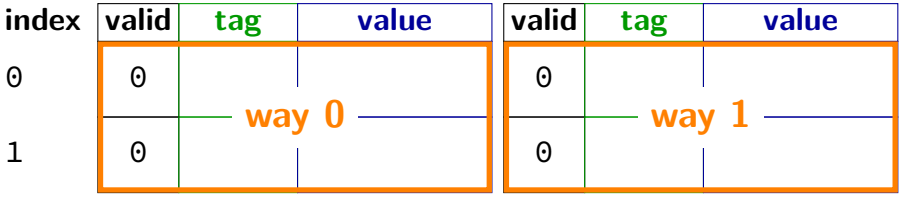
# adding associativity

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

| index | valid | tag | value | valid | tag | value |
|---|---|---|---|---|---|---|
| 0 | 0 | | set 0 | 0 | | |
| 1 | 0 | | set 1 | 0 | | |

# adding associativity

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



# adding associativity

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

| index | valid | tag | value | valid | tag | value |
|---|---|---|---|---|---|---|
| 0 | 0 | | | 0 | | |
| 1 | 0 | | | 0 | | |

$m = 8$ bit addresses
$S = 2 = 2^s$ sets
$s = 1$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 6$ tag bits

# adding associativity

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

| index | valid | tag | value | valid | tag | value |
|---|---|---|---|---|---|---|
| 0 | 1 | 000000 | mem[0x00]<br>mem[0x01] | 0 | | |
| 1 | 0 | | | 0 | | |

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

tag index offset

# adding associativity

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

| index | valid | tag | value | valid | tag | value |
|---|---|---|---|---|---|---|
| 0 | 1 | 000000 | mem[0x00]<br>mem[0x01] | 0 | | |
| 1 | 0 | | | 0 | | |

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

tag index offset

# adding associativity

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

| index | valid | tag | value | valid | tag | value |
|---|---|---|---|---|---|---|
| 0 | 1 | 000000 | mem[0x00]<br>mem[0x01] | 0 | | |
| 1 | 1 | 011000 | mem[0x62]<br>mem[0x63] | 0 | | |

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

tag index offset

# adding associativity

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

| index | valid | tag | value | valid | tag | value |
|---|---|---|---|---|---|---|
| 0 | 1 | 000000 | mem[0x00]<br>mem[0x01] | 1 | 011000 | mem[0x60]<br>mem[0x61] |
| 1 | 1 | 011000 | mem[0x62]<br>mem[0x63] | 0 | | |

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

tag index offset

# adding associativity

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

| index | valid | tag | value | valid | tag | value |
|---|---|---|---|---|---|---|
| 0 | 1 | 000000 | mem[0x00]<br>mem[0x01] | 1 | 011000 | mem[0x60]<br>mem[0x61] |
| 1 | 1 | 011000 | mem[0x62]<br>mem[0x63] | 0 | | |

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

tag index offset

# adding associativity

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

| index | valid | tag | value | valid | tag | value |
|---|---|---|---|---|---|---|
| 0 | 1 | 000000 | mem[0x00]<br>mem[0x01] | 1 | 011000 | mem[0x60]<br>mem[0x61] |
| 1 | 1 | 011000 | mem[0x62]<br>mem[0x63] | 0 | | |

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

tag index offset

# adding associativity

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

| index | valid | tag | value | valid | tag | value |
|---|---|---|---|---|---|---|
| 0 | 1 | 000000 | mem[0x00]<br>mem[0x01] | 1 | 011000 | mem[0x60]<br>mem[0x61] |
| 1 | 1 | 011000 | mem[0x62]<br>mem[0x63] | 0 | | |

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

tag index offset

needs to replace block in set 0!

# adding associativity

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

| index | valid | tag | value | valid | tag | value |
|---|---|---|---|---|---|---|
| 0 | 1 | 000000 | mem[0x00]<br>mem[0x01] | 1 | 011000 | mem[0x60]<br>mem[0x61] |
| 1 | 1 | 011000 | mem[0x62]<br>mem[0x63] | 0 | | |

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

tag index offset

# associative lookup possibilities

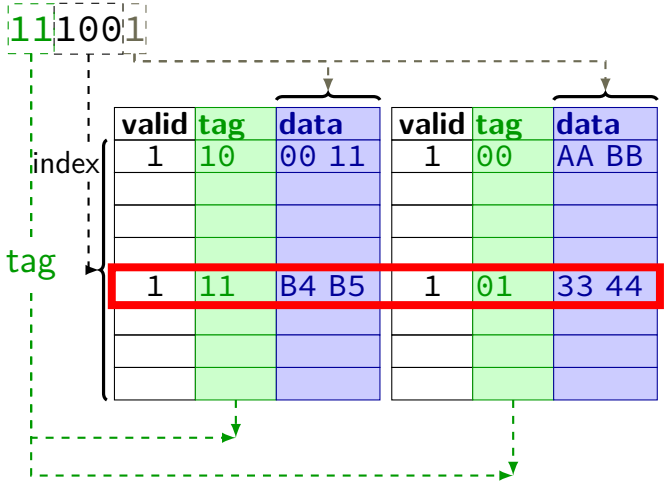
none of the blocks for the index are valid

none of the valid blocks for the index match the tag

- something else is stored there

one of the blocks for the index is valid and matches the tag

# cache operation (associative)



# replacement policies

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

| index | valid | tag | value | valid | tag | value |
|---|---|---|---|---|---|---|
| 0 | 1 | 000000 | mem[0x00]<br>mem[0x01] | 1 | 011000 | mem[0x60]<br>mem[0x61] |
| 1 | 1 | 011000 | mem[0x62]<br>mem[0x63] | 0 | | |

| address (hex) | result |
|---|---|
| 000 | |
| 00000001 (01) | hit |
| 01100011 (63) | miss |
| 01100001 (61) | miss |
| 01100010 (62) | hit |
| 00000000 (00) | hit |
| 01100100 (64) | miss |

how to decide where to insert 0x64?

# replacement policies

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

| index | valid | tag | value | valid | tag | value | LRU |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 000000 | mem[0x00]<br>mem[0x01] | 1 | 011000 | mem[0x60]<br>mem[0x61] | 1 |
| 1 | 1 | 011000 | mem[0x62]<br>mem[0x63] | 0 | | | 1 |

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

track which block was read least recently
updated on every access

# example replacement policies

least recently used

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

approximations of least recently used

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

first-in, first-out

- counter per set — where to replace next

(pseudo-)random

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

# associativity terminology

direct-mapped — one block per set

$E$-way set associative — $E$ blocks per set

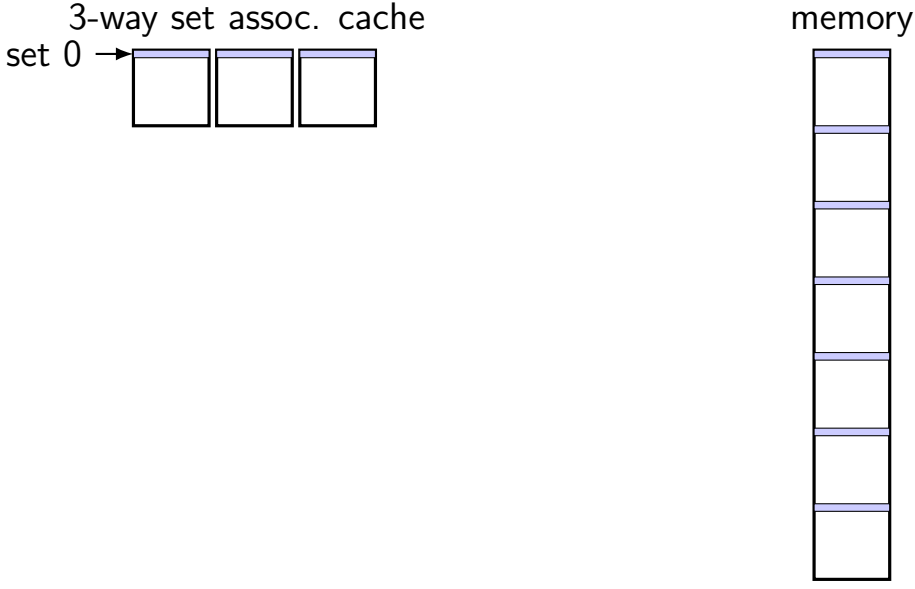
$E$ ways in the cache

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

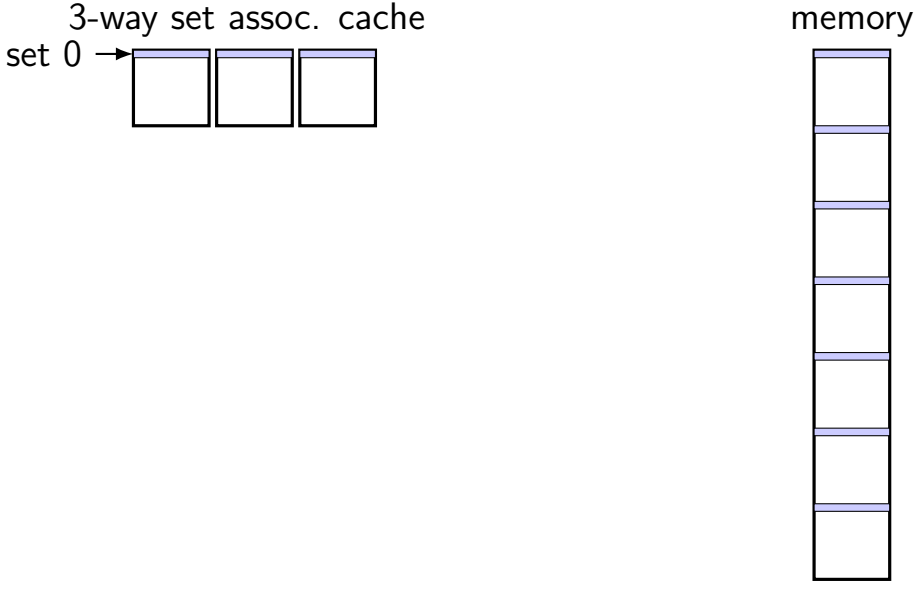
# Tag-Index-Offset formulas

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

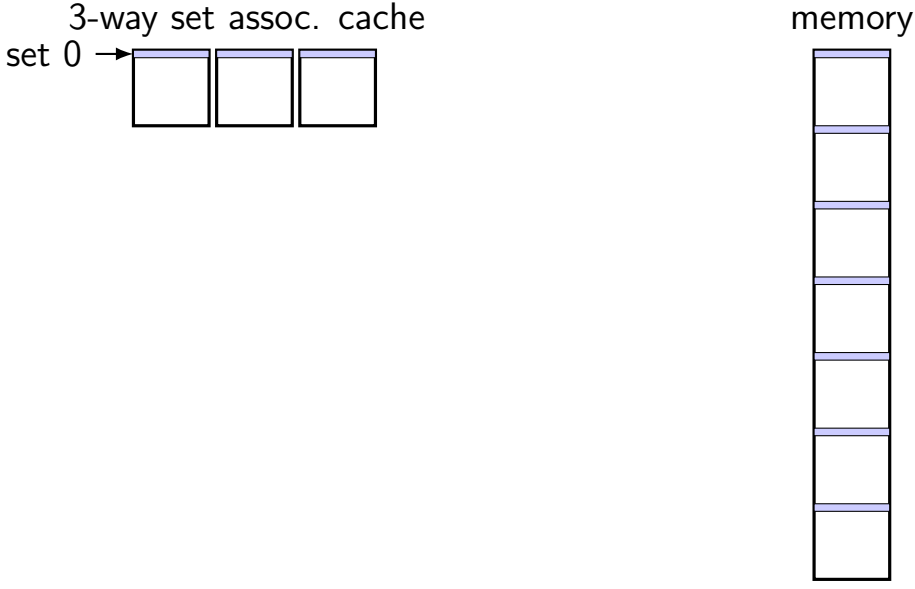
# mapping of sets to memory (3-way)



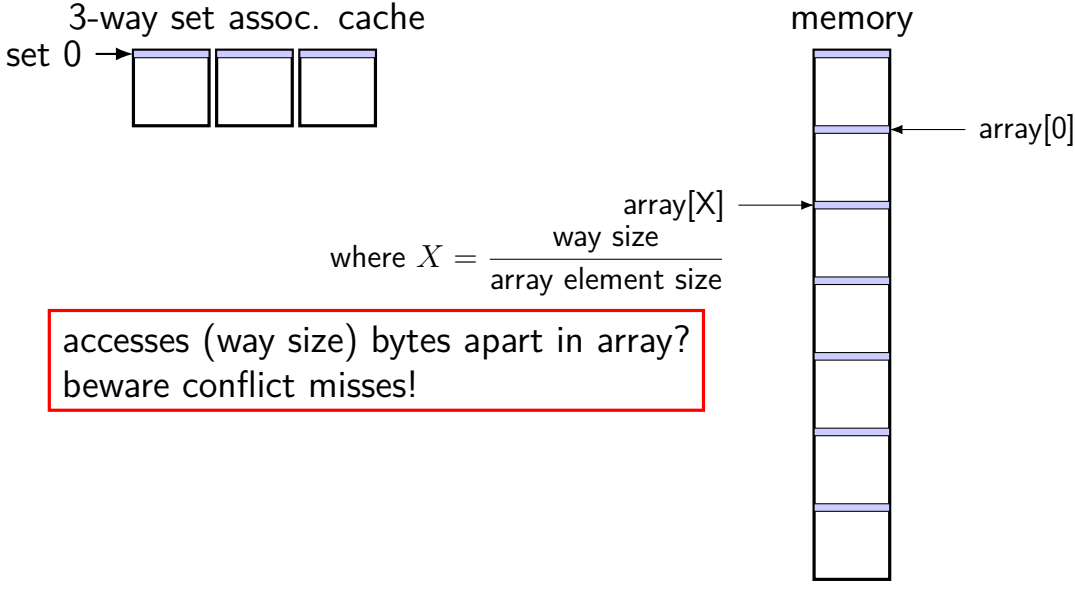
# mapping of sets to memory (3-way)



# mapping of sets to memory (3-way)



# mapping of sets to memory (3-way)




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

# 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

How about on a two-way set associative cache?

# C and cache misses (assoc)

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

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

opbservation: array[0], array[256], array[512], array[768] in same set

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

# C and cache misses (assoc)

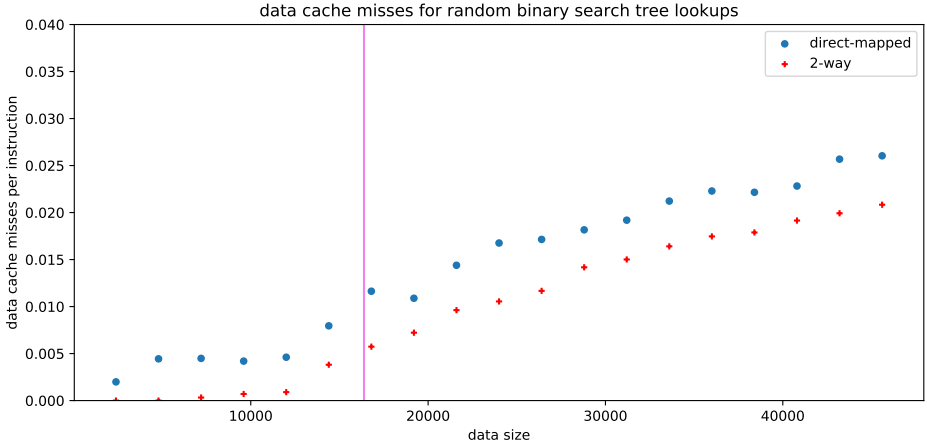
```
int array[1024]; /* assume aligned */
int even_sum = 0, odd_sum = 0;
even_sum += array[0];
even_sum += array[256];
even_sum += array[512];
even_sum += array[768];
odd_sum += array[1];
odd_sum += array[257];
odd_sum += array[513];
odd_sum += array[769];
```

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

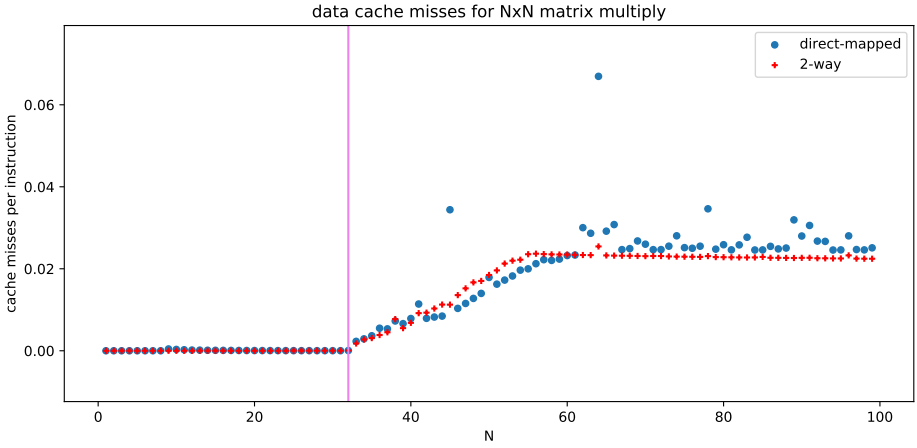
observation: array[0], array[256], array[512], array[768] in same set

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

# simulated misses: BST lookups



# simulated misses: matrix multiplies



# handling writes

what about writing to the cache?

two decision points:

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

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

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

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

# allocate on write?

processor writes less than whole cache block

block not yet in cache

two options:

write-allocate

> fetch rest of cache block, replace written part
> (then follow write-through or write-back policy)

write-no-allocate

> don't use cache at all (send write to memory *instead*)
> guess: not read soon?

# allocate on write?

processor writes less than whole cache block

block not yet in cache

two options:

write-allocate

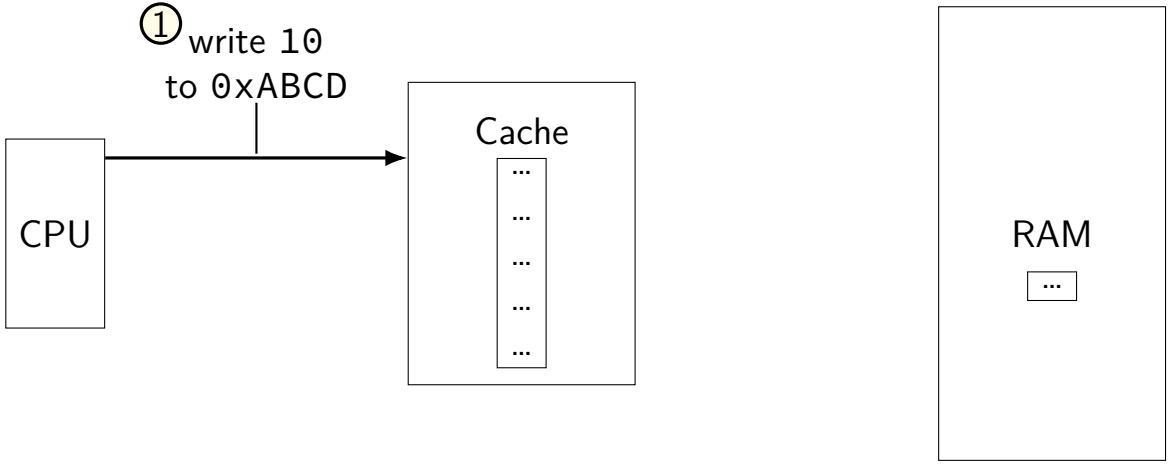
> fetch rest of cache block, replace written part
> (then follow write-through or write-back policy)

write-no-allocate

> don't use cache at all (send write to memory *instead*)
> guess: not read soon?

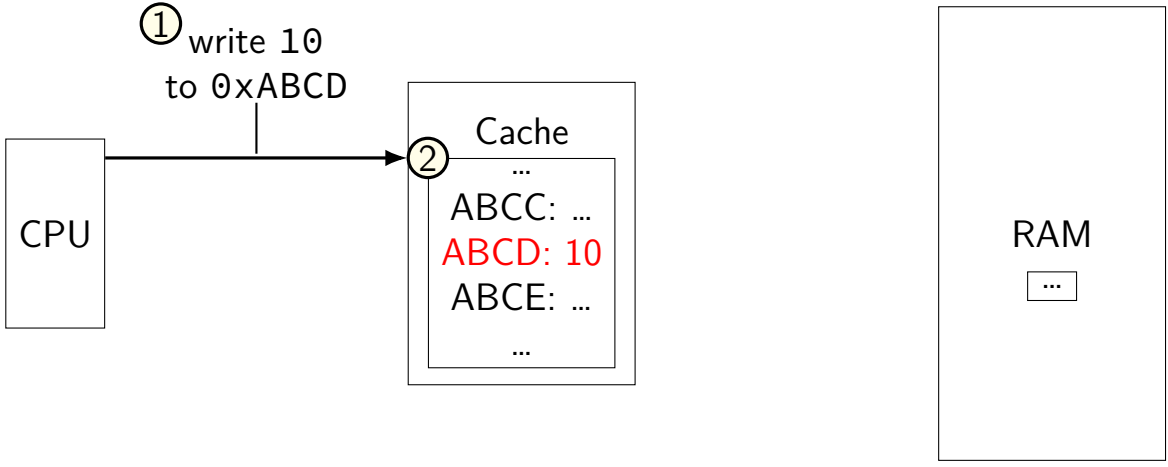
# write-allocate v. write-no-allocate

**option 1: write-allocate**

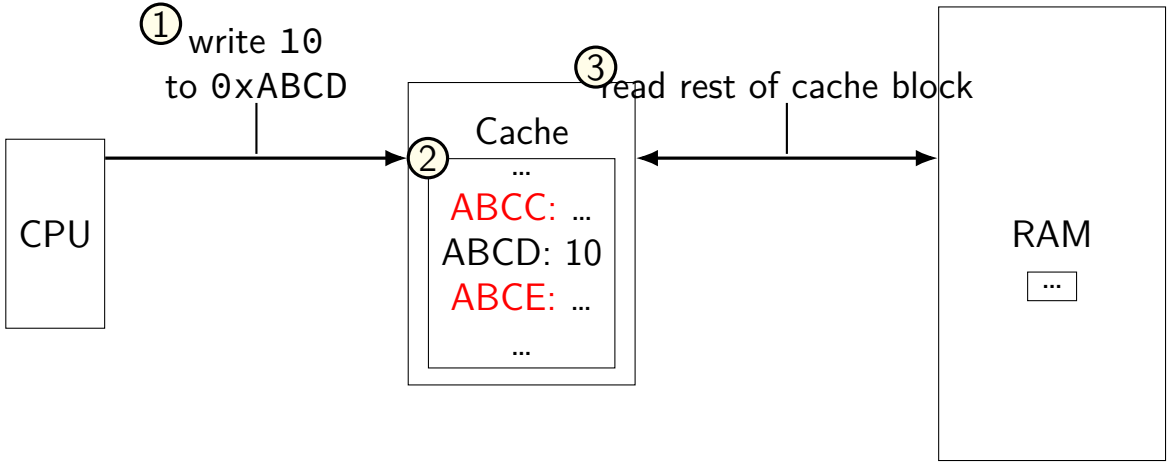


# write-allocate v. write-no-allocate

**option 1: write-allocate**

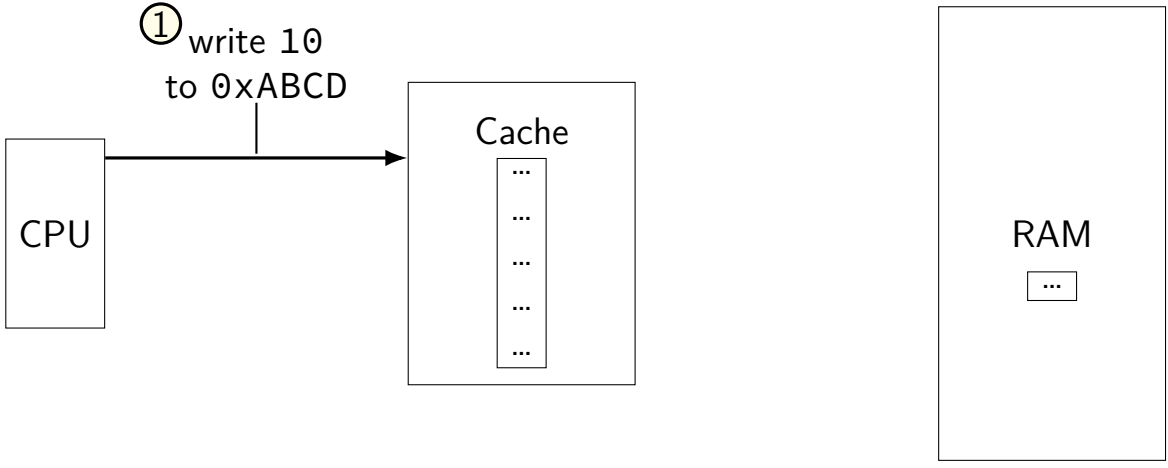


# write-allocate v. write-no-allocate



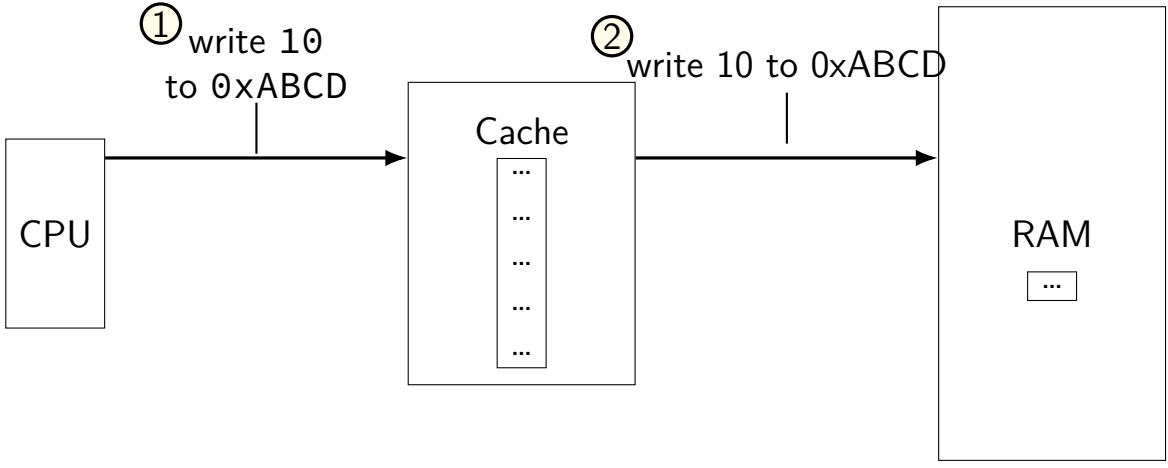
# write-allocate v. write-no-allocate

**option 2: write-no-allocate**



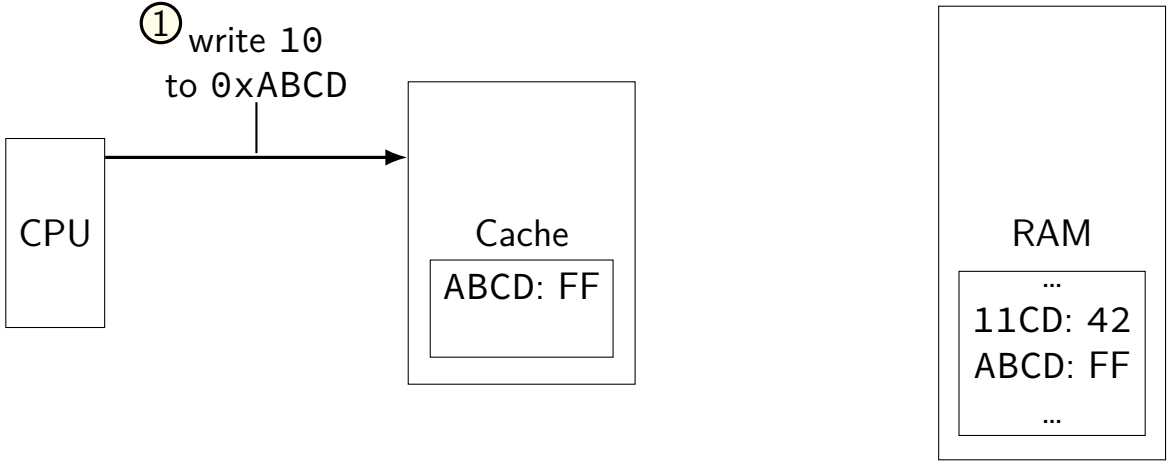
# write-allocate v. write-no-allocate

**option 2: write-no-allocate**



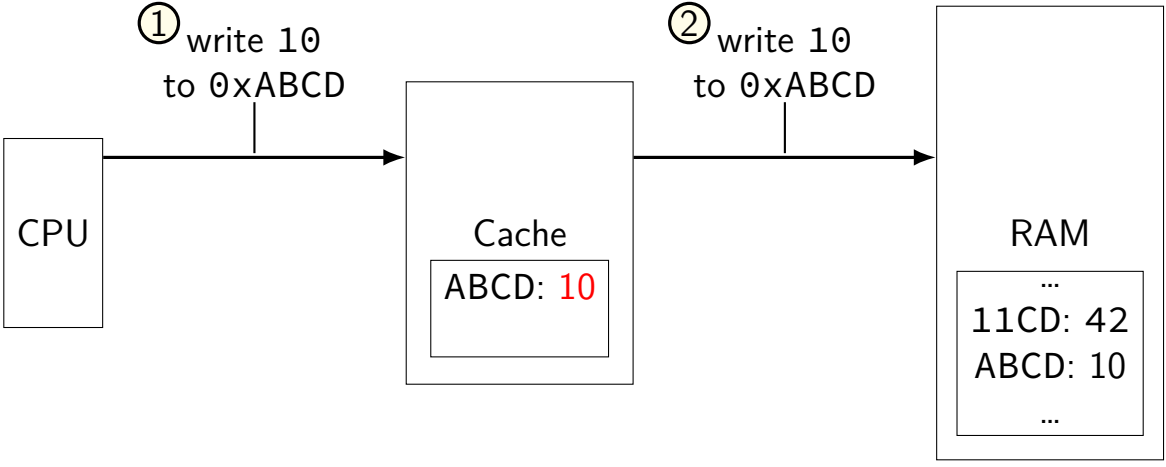
# write-through v. write-back

**option 1: write-through**



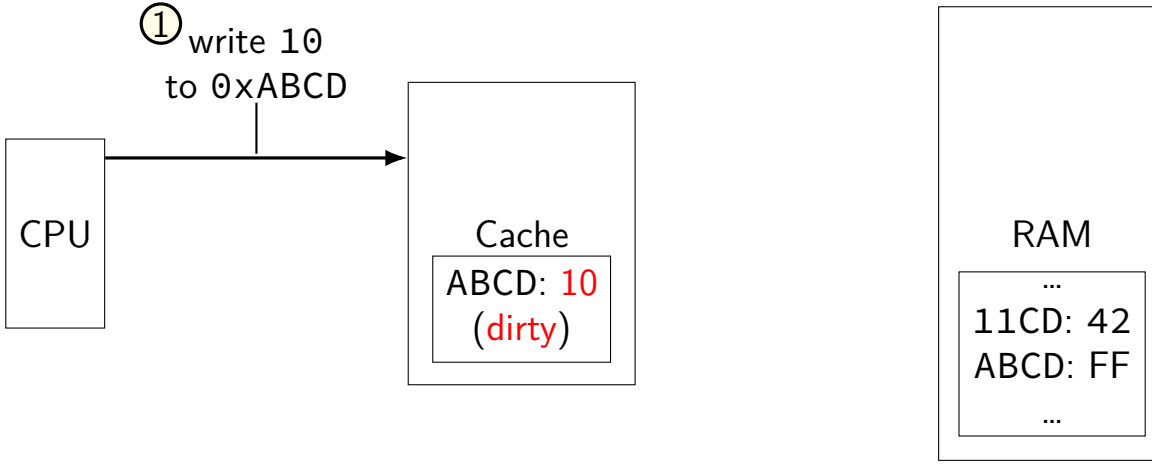
# write-through v. write-back

**option 1: write-through**



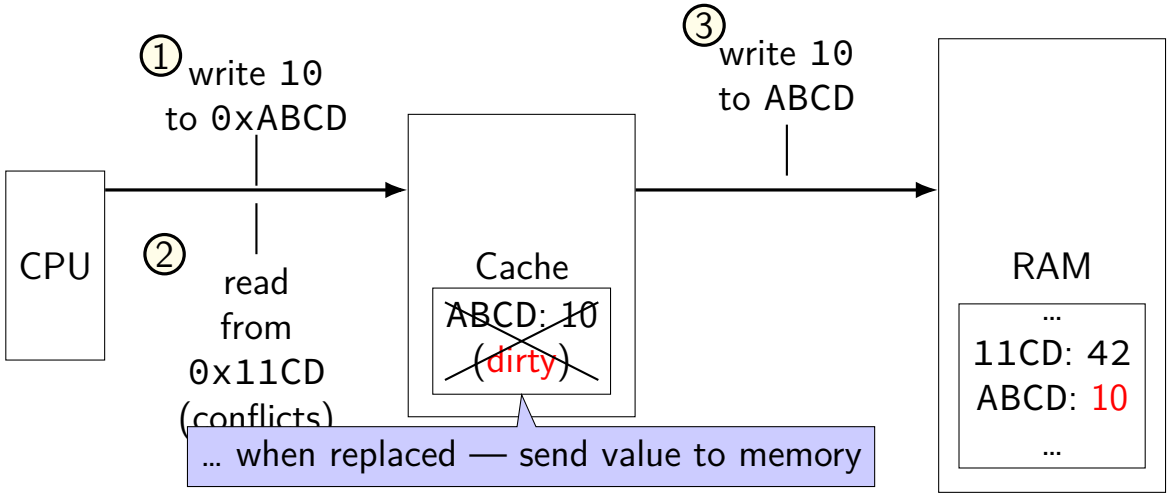
# write-through v. write-back

**option 2: write-back**

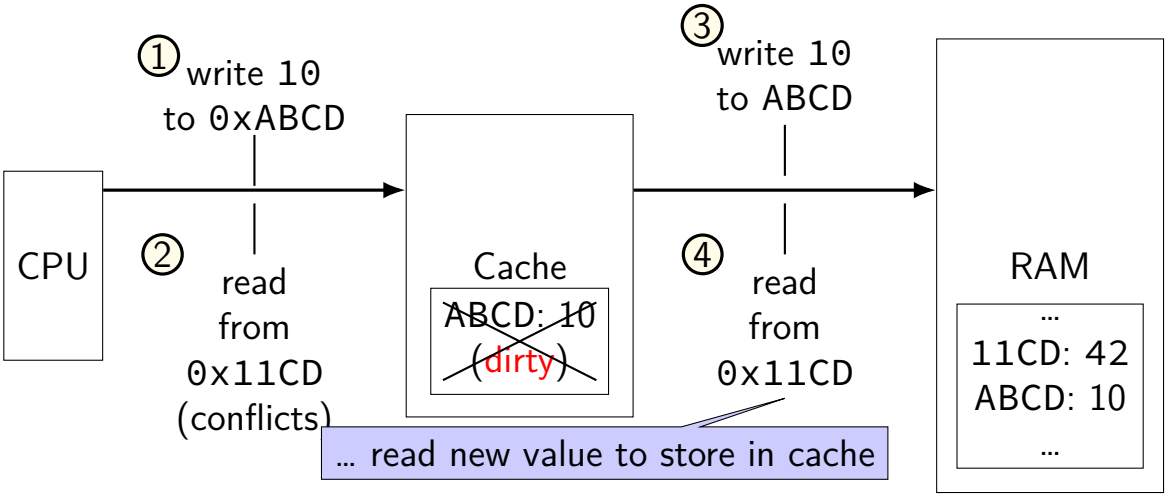


# write-through v. write-back

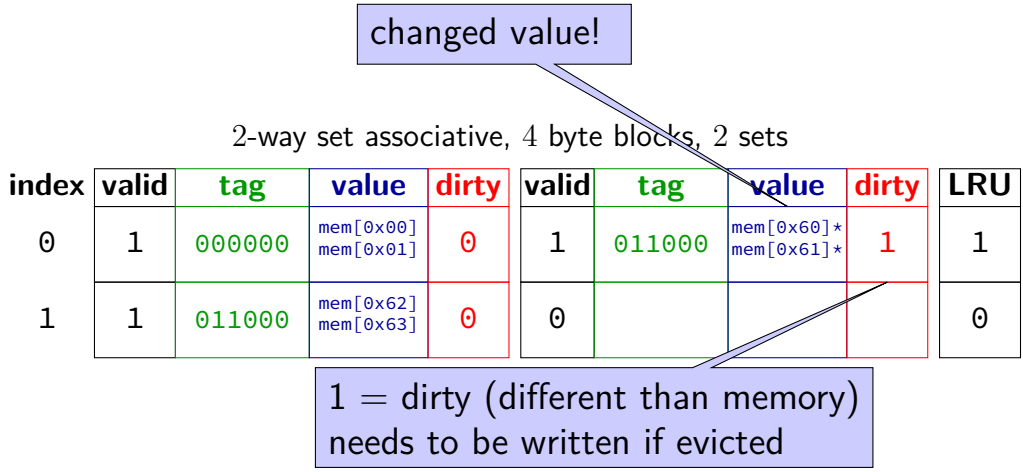
**option 2: write-back**



# write-through v. write-back



# writeback policy



# write-allocate + write-back

2-way set associative, LRU, writeback

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

writing $\widehat{0}$xFF into address 0x04?
index 0, tag 000001

# write-allocate + write-back

2-way set associative, LRU, writeback

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

writing $\widehat{0}$xFF into address 0x04?

index 0, tag 000001

step 1: find least recently used block

# write-allocate + write-back

2-way set associative, LRU, writeback

| index | valid | tag | value | dirty | valid | tag | value | dirty | LRU |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 000000 | mem[0x00]<br>mem[0x01] | 0 | 1 | 011000 | mem[0x60]*<br>mem[0x61]* | ~~1~~ | 1 |
| 1 | 1 | 011000 | mem[0x62]<br>mem[0x63] | 0 | 0 | | | | 0 |

writing $\widehat{0}$xFF into address 0x04?

index 0, tag 000001

step 1: find least recently used block

step 2: possibly writeback old block

# write-allocate + write-back

2-way set associative, LRU, writeback

| index | valid | tag | value | dirty | valid | tag | value | dirty | LRU |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 000000 | mem[0x00]<br>mem[0x01] | 0 | 1 | 000001 | 0xFF<br>mem[0x05] | 1 | 0 |
| 1 | 1 | 011000 | mem[0x62]<br>mem[0x63] | 0 | 0 | | | | 0 |

writing 0xFF into address 0x04?

index 0, tag 000001

step 1: find least recently used block

step 2: possibly writeback old block

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

step 3b: update LRU information

# write-no-allocate + write-back

2-way set associative, LRU, writeback

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

writing $\widehat{0}$xFF into address 0x04?

step 1: is it in cache yet?

step 2: no, just send it to memory

# exercise (1)

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

| index | valid | tag | value | dirty | valid | tag | value | dirty | LRU |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 001100 | mem[0x30]<br>mem[0x31] | 0 | 1 | 010000 | mem[0x40]*<br>mem[0x41]* | 1 | 0 |
| 1 | 1 | 011000 | mem[0x62]<br>mem[0x63] | 0 | 1 | 001100 | mem[0x32]*<br>mem[0x33]* | 1 | 1 |

for each of the following accesses, performed alone, would it require (a) reading a value from memory (or next level of cache) and (b) writing a value to the memory (or next level of cache)?

writing 1 byte to 0x33

reading 1 byte from 0x52

reading 1 byte from 0x50

# exercise (2)

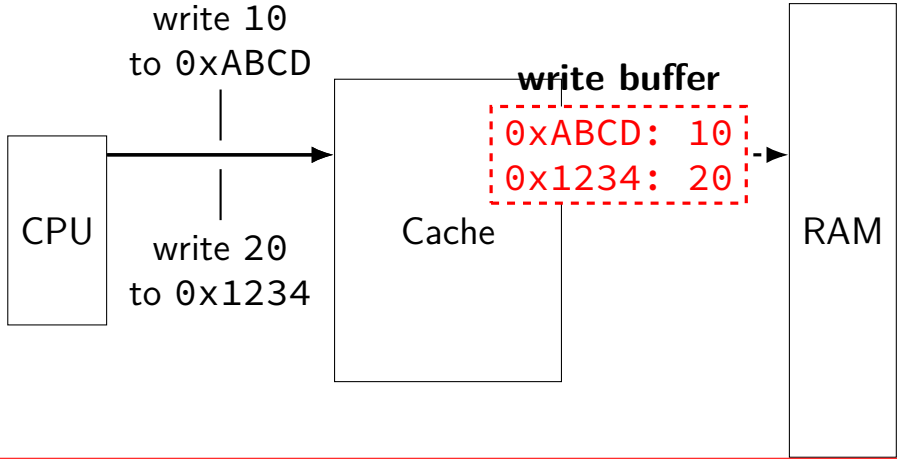
2-way set associative, LRU, write-no-allocate, write-through

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

for each of the following accesses, performed alone, would it require (a) reading a value from memory and (b) writing a value to the memory?

writing 1 byte to 0x33
reading 1 byte from 0x52
reading 1 byte from 0x50

# fast writes




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

# cache tradeoffs briefly

deciding cache size, associativity, etc.?

lots of tradeoffs:

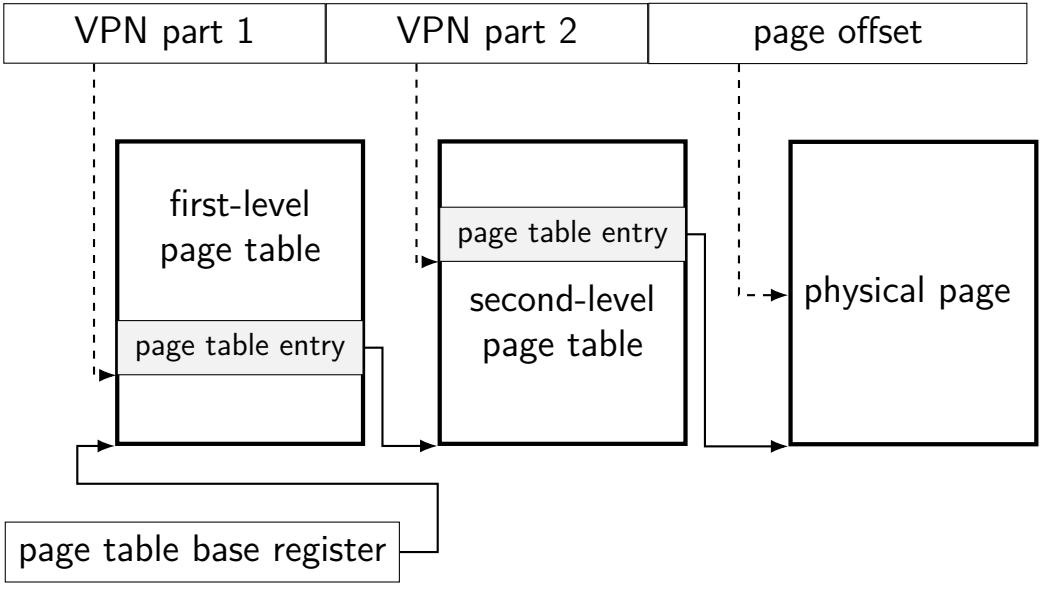
- more cache hits v. slower cache hits?
- faster cache hits v. fewer cache hits?
- more cache hits v. slower cache misses?
- ...

details depend on programs run

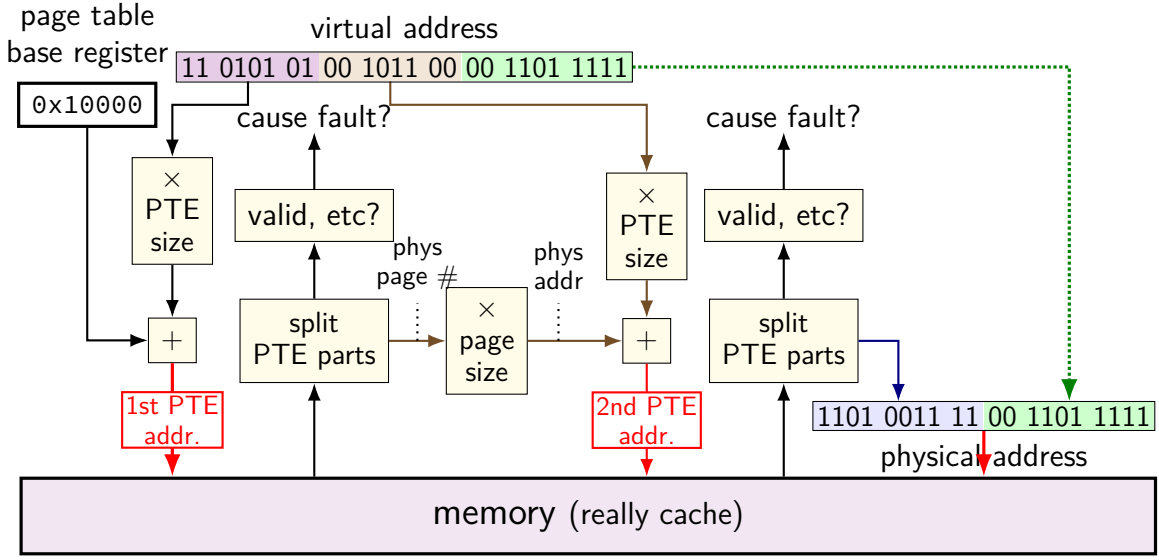
- how often is same block used again?
- how often is same index bits used?

simulation to assess impact of designs

# another view



# two-level page table lookup



# cache accesses and multi-level PTs

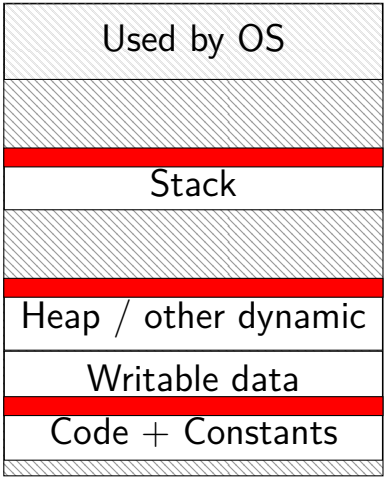
four-level page tables — five cache accesses per program memory access

L1 cache hits — typically a couple cycles each?

so add 8 cycles to each program memory access?

not acceptable

# program memory active sets




0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F…

small areas of memory active at a time

one or two pages in each area?

0x0000 0000 0040 0000

# page table entries and locality

page table entries have excellent temporal locality

typically one or two pages of the stack active

typically one or two pages of code active

typically one or two pages of heap/globals active

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

# page table entries and locality

page table entries have excellent temporal locality

typically one or two pages of the stack active

typically one or two pages of code active

typically one or two pages of heap/globals active

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

needed page table entries are very small

# page table entry cache

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

very small cache of page table entries

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

# page table entry cache

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

very small cache of page table entries

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

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

# page table entry cache

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

very small cache of page table entries

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

virtual page number divided into index + tag

# page table entry cache

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

very small cache of page table entries

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

not much spatial locality between page table entries
(they're used for kilobytes of data already)

# page table entry cache

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

very small cache of page table entries

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

0 block offset bits

# page table entry cache

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

very small cache of page table entries

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

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

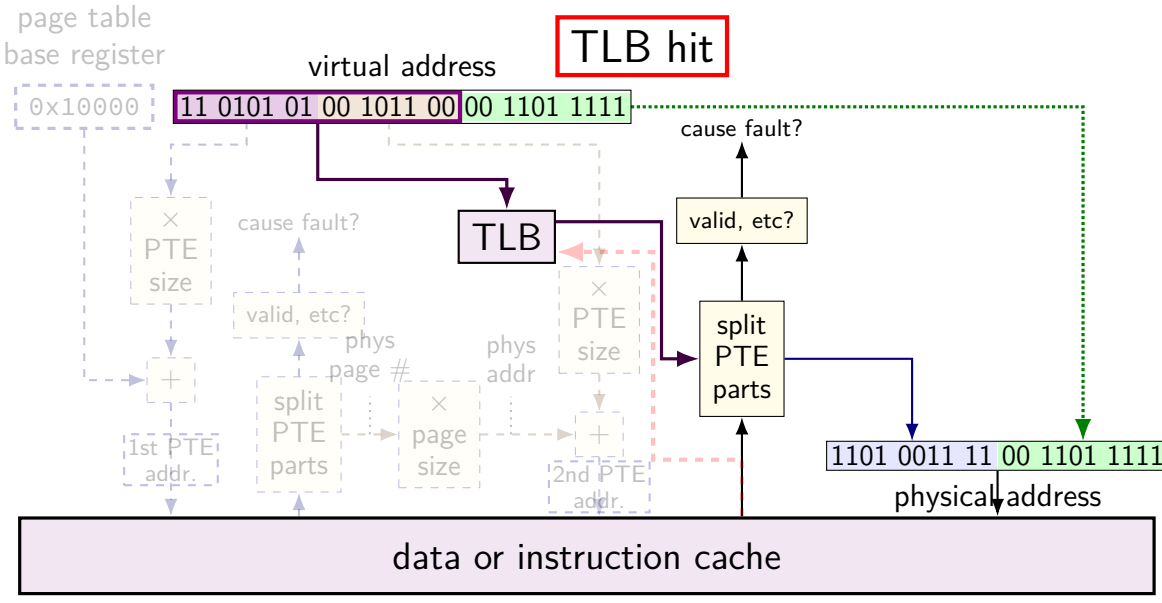
# TLB and multi-level page tables

TLB caches valid last-level page table entries

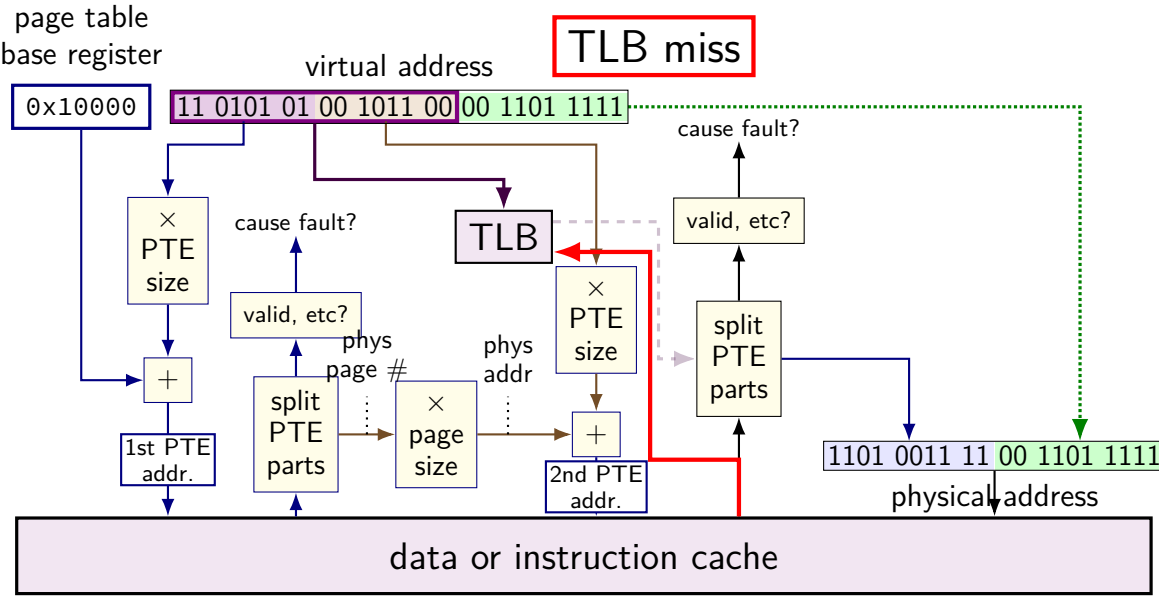
doesn't matter which last-level page table

means TLB output can be used directly to form address

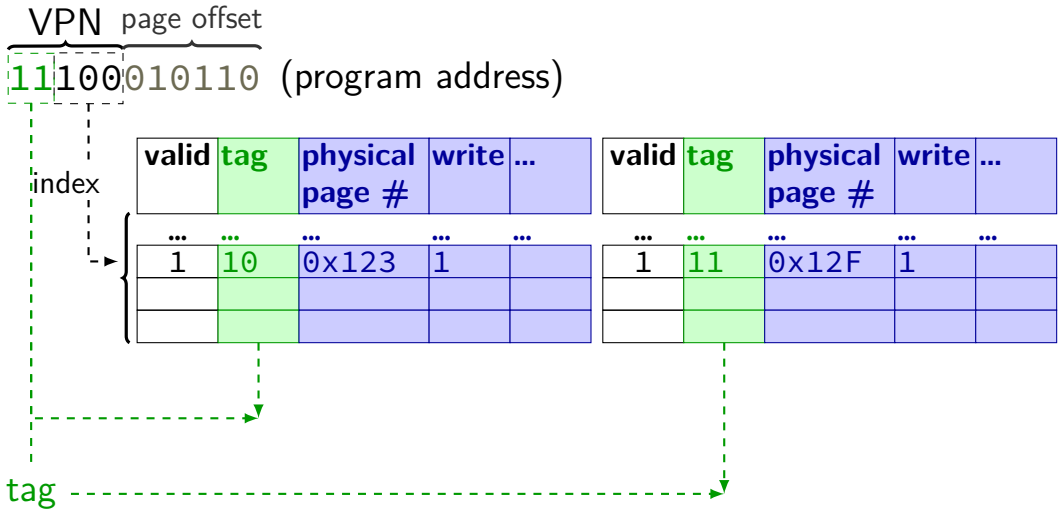
# TLB and two-level lookup



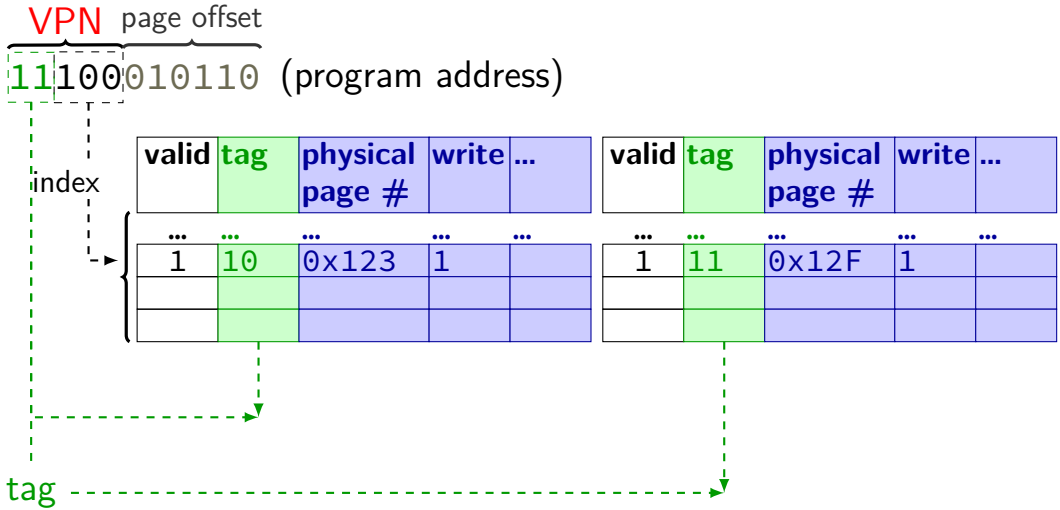
# TLB and two-level lookup



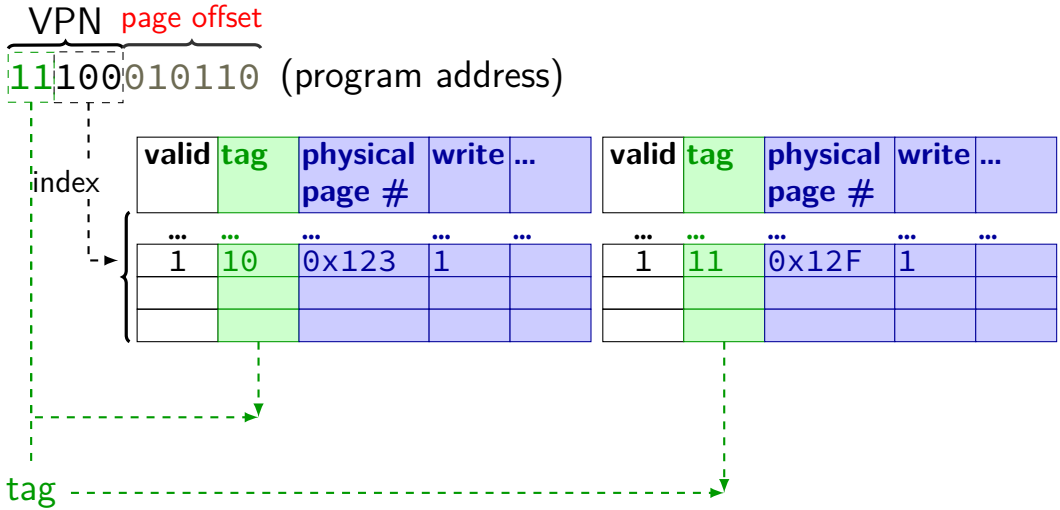
# TLB organization (2-way set associative)



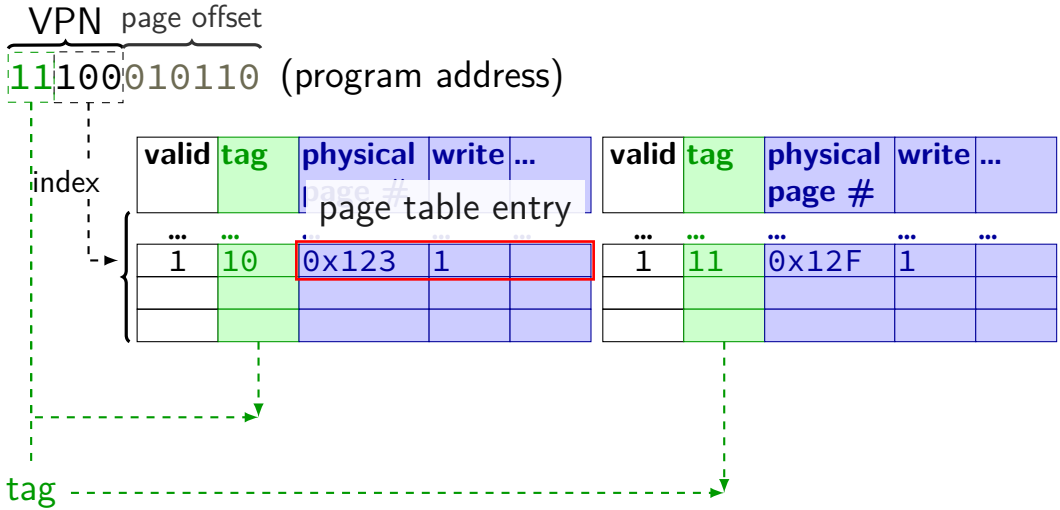
# TLB organization (2-way set associative)



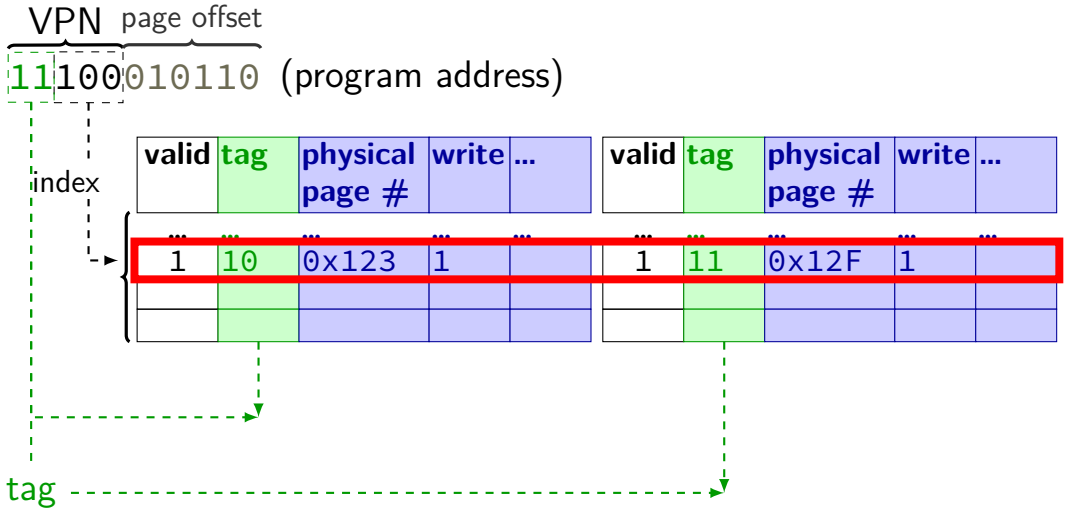
# TLB organization (2-way set associative)



# TLB organization (2-way set associative)



# TLB organization (2-way set associative)



# exercise: TLB access pattern (setup)

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

4096 byte pages

how many index bits?

TLB index of virtual address 0x12345?

# exercise: TLB access pattern

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

4096 byte pages

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

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