Pipe Hazards (finish) / Caching

minor HCL update

new version of hclrs.tar:

removes poptest.yo from list of tests for seqhw, pipehw2

gives error if you try to assign to register output:

```
register xY { foo : 1 = 0; }
x_foo = 0; Y_foo = 1;
```

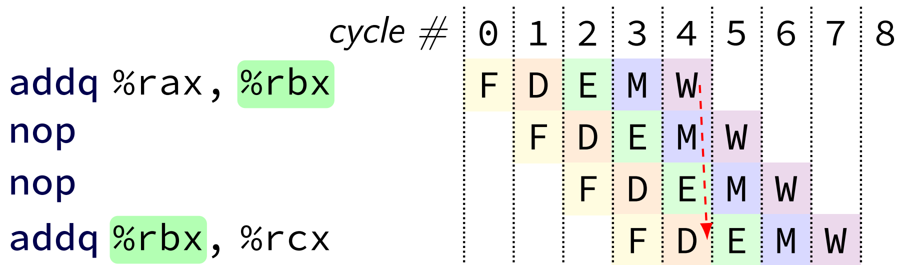
(previously: silently choose a value to give `Y_foo`)

post-quiz Q1 B

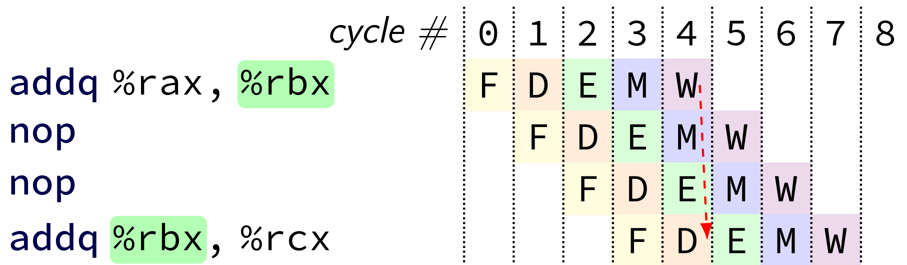
```
rmmovq %rcx, 0(%rdx)
    // reads: %rcx, %rdx
    // writes: data memory [no registers!]
nop
addq %rcx, %rdx
    // reads: %rcx, %rdx
    // writes: %rdx
```

read to read → no dependency

post-quiz Q1 D



post-quiz Q1 D



post-quiz Q 2

goal: squash instruction currently in execute

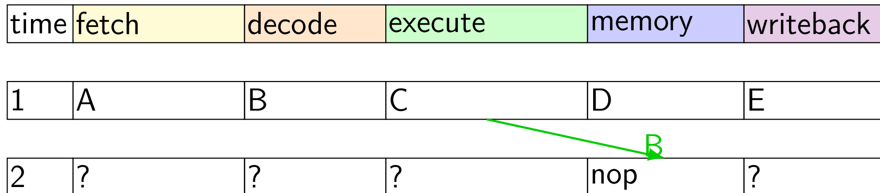
i.e. C disappears from the pipeline instead of advancing to memory

time	fetch	decode	execute	memory	writeback
1	A	B	C	D	E
2	?	?	?	?	?

post-quiz Q 2

goal: squash instruction currently in execute

i.e. C disappears from the pipeline instead of advancing to memory

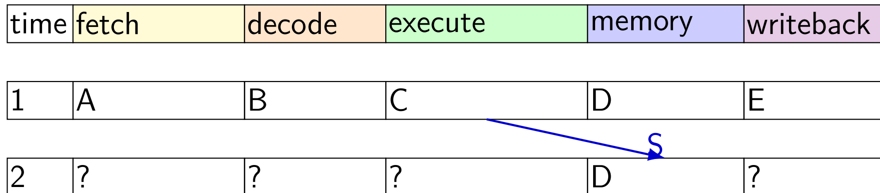


time	fetch	decode	execute	memory	writeback
1	A	B	C	D	E
2	?	?	?	nop	?

post-quiz Q 2

goal: squash instruction currently in execute

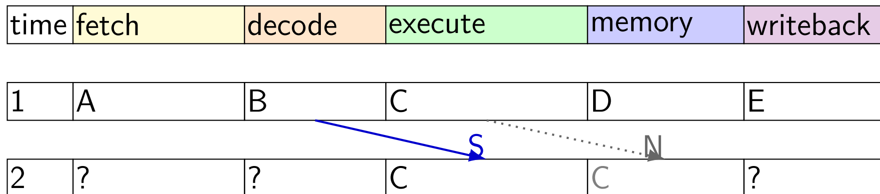
i.e. C disappears from the pipeline instead of advancing to memory



post-quiz Q 2

goal: squash instruction currently in execute

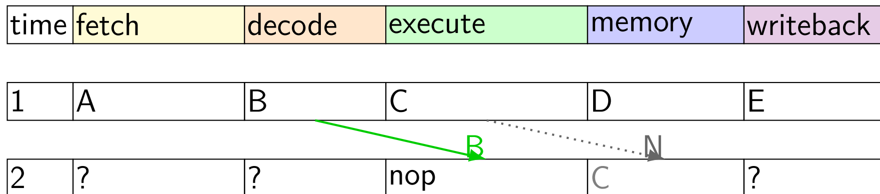
i.e. C disappears from the pipeline instead of advancing to memory



post-quiz Q 2

goal: squash instruction currently in execute

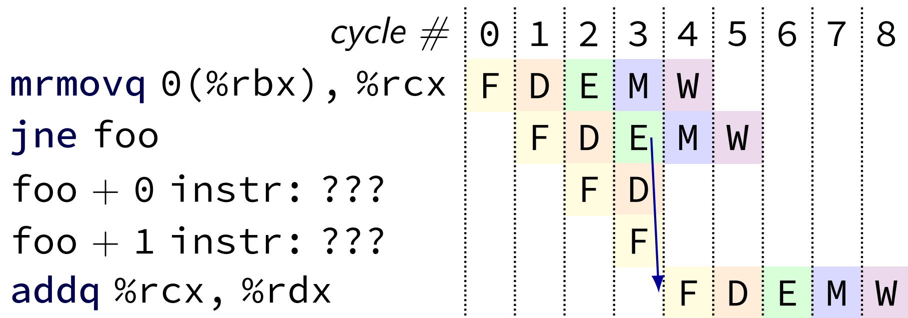
i.e. C disappears from the pipeline instead of advancing to memory



post-quiz Q 3

our/book's CPU predicts branches as always taken

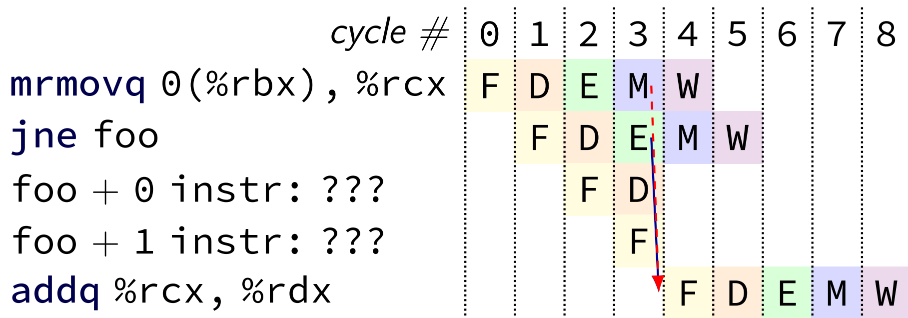
use branch result at beginning of jump's memory stage



post-quiz Q 3

our/book's CPU predicts branches as always taken

use branch result at beginning of jump's memory stage



post-quiz Q 4

irmovq → rmmovq: needs %rax

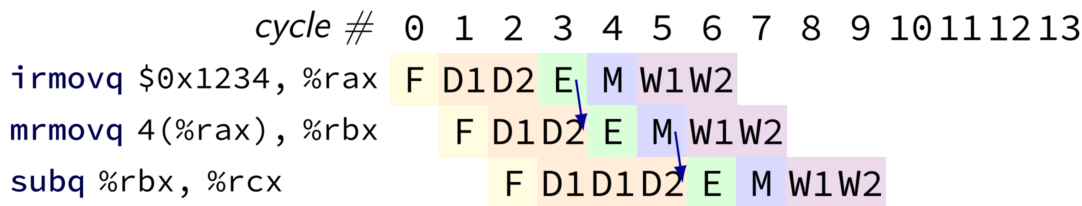
mrmovq → subq: needs %rbx

cycle #	0	1	2	3	4	5	6	7	8	9	10	11	12	13		
`irmovq $0x1234, %rax`	F	D1	D2	E	M	W1	W2									
`mrmovq 4(%rax), %rbx`		F	D1	D1	D1	D1	D1	D1	D2	E	M	W1	W2			
`subq %rbx, %rcx`			F	F	F	F	F	F	D1	D1	D1	D1	D1	D1	D2	E

post-quiz Q 5

irmovq → rmmovq: needs %rax

mrmovq → subq: needs %rbx



after forwarding/prediction

where do we still need to stall?

memory output needed in fetch

- `ret` followed by anything

memory output needed in exceute

- `mrmovq` or `popq` + use
  (in immediatelly following instruction)

overall CPU

5 stage pipeline

1 instruction completes every cycle — except hazards

most data hazards: solved by forwarding

load/use hazard: 1 cycle of stalling

jXX control hazard: branch prediction + squashing

- 2 cycle penalty for misprediction
- (correct misprediction after jXX finishes execute)

ret control hazard: 3 cycles of stalling

- (fetch next instruction after ret finishes memory)

pipeline with different hazards

example: 4-stage pipeline:
fetch/decode/execute+memory/writeback

```
                      // 4 stage   // 5 stage
addq %rax, %r8        //           // W
subq %rax, %r9        // W         // M
xorq %rax, %r10       // EM        // E
andq %r8,  %r11       // D         // D
```

pipeline with different hazards

example: 4-stage pipeline:
fetch/decode/execute+memory/writeback

```
                    // 4 stage  // 5 stage
addq %rax, %r8      //          // W
subq %rax, %r9      // W        // M
xorq %rax, %r10     // EM       // E
andq %r8,  %r11     // D        // D
```

addq/andq is hazard with 5-stage pipeline

addq/andq is **not** a hazard with 4-stage pipeline

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

result only available after second execute stage

where does forwarding, stalls occur?

cycle #	0	1	2	3	4	5	6	7	8
`addq %rcx, %r9`	F	D	E1	E2	M	W			
`addq %r9, %rbx`									
`addq %rax, %r9`									
`rmmovq %r9, (%rbx)`									

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

	cycle #	0	1	2	3	4	5	6	7	8
`addq %rcx, %r9`		F	D	E1	E2	M	W			
`addq %r9, %rbx`										
`addq %rax, %r9`										
`rmmovq %r9, (%rbx)`										

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

cycle #	0	1	2	3	4	5	6	7	8
addq %rcx, %r9	F	D	E1	E2	M	W			
addq %r9, %rbx		F	D	E1	E2	M	W		
addq %rax, %r9			F	D	E1	E2	M	W	
rmmovq %r9, (%rbx)				F	D	E1	E2	M	W

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

cycle #	0	1	2	3	4	5	6	7	8	
addq %rcx, %r9	F	D	E1	E2	M	W				
addq %r9, %rbx		F	D	E1	E2	M	W			
addq %r9, %rbx		F	D	D	E1	E2	M	W		
addq %rax, %r9			F	D	E1	E2	M	W		
addq %rax, %r9			F	F	D	E1	E2	M	W	
rmmovq %r9, (%rbx)				F	D	E1	E2	M	W	
rmmovq %r9, (%rbx)					F	D	E1	E2	M	W

missing pieces

multi-cycle memories

beyond pipelining: out-of-order, multiple issue

missing pieces

multi-cycle memories

beyond pipelining: out-of-order, multiple issue

multi-cycle memories

ideal case for memories: single-cycle

achieved with caches (next topic)

- fast access to small number of things

typical performance:

- 90+% of the time: single-cycle

sometimes many cycles (3–400+)

variable speed memories

memory is fast: (cache "hit"; recently accessed?)

cycle #	0	1	2	3	4	5	6	7	8
`mrmovq 0(%rbx), %r8`	F	D	E	M	W				
`mrmovq 0(%rcx), %r9`		F	D	E	M	W			
`addq %r8, %r9`			F	D	D	E	M	W	

memory is slow: (cache "miss")

cycle #	0	1	2	3	4	5	6	7	8			
`mrmovq 0(%rbx), %r8`	F	D	E	M	M	M	M	M	W			
`mrmovq 0(%rcx), %r9`		F	D	E	E	E	E	E	M	M	M	M
`addq %r8, %r9`			F	D	D	D	D	D	D	D	D	D

missing pieces

multi-cycle memories

beyond pipelining: out-of-order, multiple issue

beyond pipelining: multiple issue

start more than one instruction/cycle

multiple parallel pipelines; many-input/output register file

hazard handling much more complex

	cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`		F	D	E	M	W				
`subq %r10, %r11`		F	D	E	M	W				
`xorq %r9, %r11`			F	D	E	M	W			
`subq %r10, %rbx`			F	D	E	M	W			

…

beyond pipelining: out-of-order

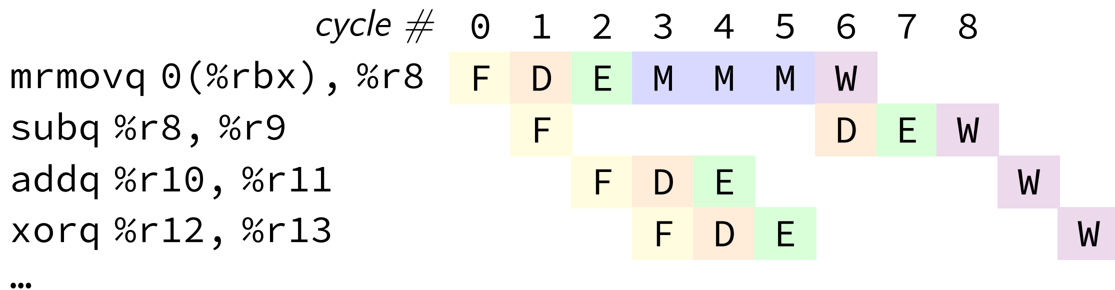
find later instructions to do instead of stalling

lists of available instructions in pipeline registers

- take any instruction with available values

provide illusion that work is still done in order

- much more complicated hazard handling logic



better branch prediction

forward (target > PC) not taken; backward taken

intuition: loops:

```
LOOP: ...
      ...
      je LOOP

LOOP: ...
      jne SKIP_LOOP
      ...
      jmp LOOP
SKIP_LOOP:
```

predicting ret: extra copy of stack

predicting ret — stack in processor registers

different than real stack/out of room? just slower

baz saved registers
baz return address
bar saved registers
bar return address
foo local variables
foo saved registers
foo return address
foo saved registers

stack in memory

baz return address
bar return address
foo return address

(partial?) stack in CPU registers

prediction before fetch

real processors can take multiple cycles to read instruction memory

predict branches before reading their opcodes

how — more extra data structures

- tables of recent branches (often many kilobytes)

2004 CPU

Floating Point Unit
Load/Store
Data Cache
Execution Units
Bus Unit
Fetch Scan Align Micro-code
Instruction Cache
L2 Cache 1MB
Memory Controller
Hyper Transport
DDR Memory Interface
Clock Generator

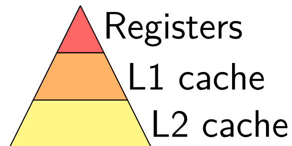


Image: approx 2004 AMD press image of Opteron die;
approx register/branch prediction location via chip-architect.org (Hans de Vries)

stalling/misprediction and latency

hazard handling where pipeline latency matters

longer pipeline — larger penalty

part of Intel's Pentium 4 problem (c. 2000)

- on release: 50% higher clock rate, 2-3x pipeline stages of competitors

out-of-order, multiple issue processor

first-generation review quote:

> For today's buyer, the Pentium 4 simply doesn't make sense. It's **slower** than the competition in just about every area, it's more expensive, it's

Review quote: Anand Lai Shimpi, "Intel Pentium 4 1.4 & 1.5 GHz", AnandTech, 20 November 2000

2004 CPU

Floating Point Unit
Load/Store
Data Cache
Execution Units
Bus Unit
Fetch Scan Align Micro-code
Instruction Cache
Memory Controller
L2 Cache IMB
Hyper Transport
DDR Memory Interface
Clock Generator

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

2004 CPU

Registers

Floating Point Unit

Load/Store

Data Cache

Execution Units

Bus Unit

L2 Cache

IMB

Hyper Transport

DDR Memory Interface

Fetch Scan Align Micro-code

Instruction Cache

Memory Controller

Clock Generator

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

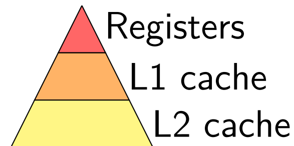
2004 CPU

Floating Point Unit
Load/Store
Data Cache
Execution Units
Bus Unit
L2 Cache
IMB
Fetch Scan Align
Micro-code
Instruction Cache
Memory Controller
Hyper Transport
DDR Memory Interface
Clock Generator

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

2004 CPU

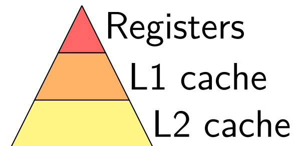
Floating Point Unit
Load/Store
Data Cache
Execution Units
Bus Unit
L2 Cache IMB
Fetch Scan Align Micro-code
Instruction Cache
Memory Controller
Hyper Transport
DDR Memory Interface
Clock Generator



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

2004 CPU

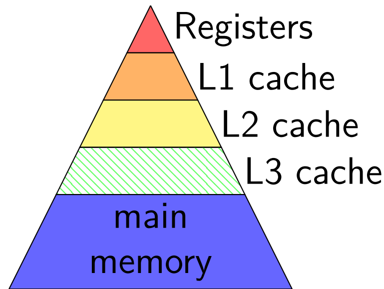
Floating Point Unit
Load/Store
Data Cache
Execution Units
Bus Unit
Fetch Scan Align Micro-code
Instruction Cache
L2 Cache IMB
Memory Controller
Hyper Transport
DDR Memory Interface
Clock Generator



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

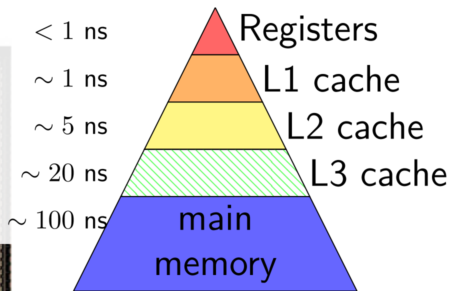
2004 CPU

Floating Point Unit
Load/Store
Data Cache
Execution Units
Bus Unit
L2 Cache IMB
Fetch Scan Align Micro-code
Instruction Cache
Memory Controller
Hyper Transport
DDR Memory Interface
Clock Generator



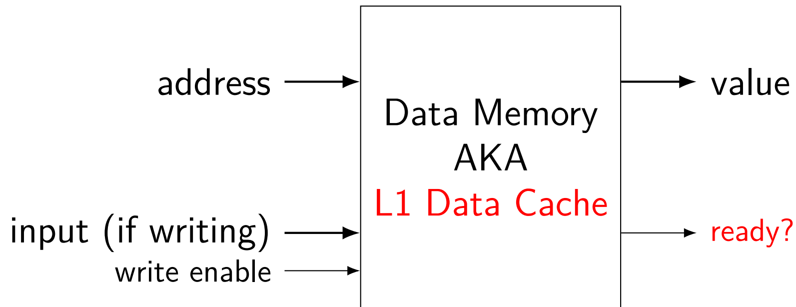
age: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU

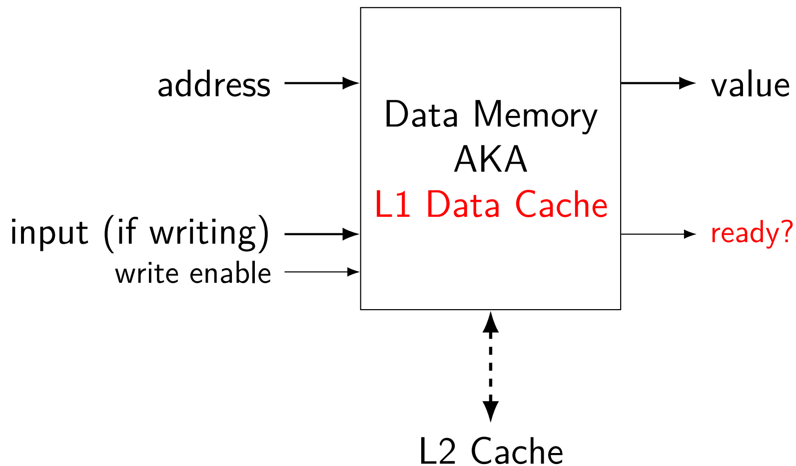


age: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

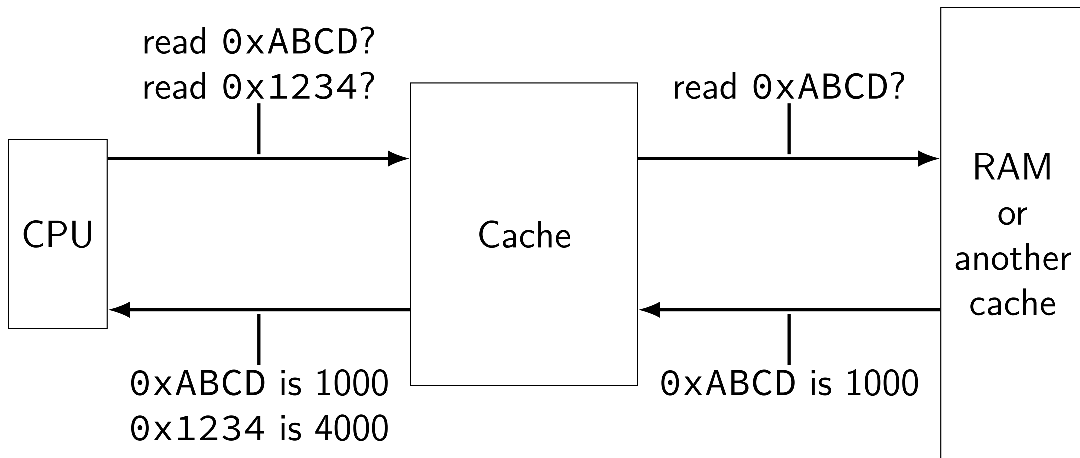
cache: real memory



cache: real memory



the place of cache



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]`

building a (direct-mapped) cache

Cache

value
00 00
00 00
00 00
00 00

cache block: 2 bytes

Memory

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

building a (direct-mapped) cache

read byte at 01011?

Cache

value
00 00
00 00
00 00
00 00

cache block: 2 bytes

Memory

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

building a (direct-mapped) cache

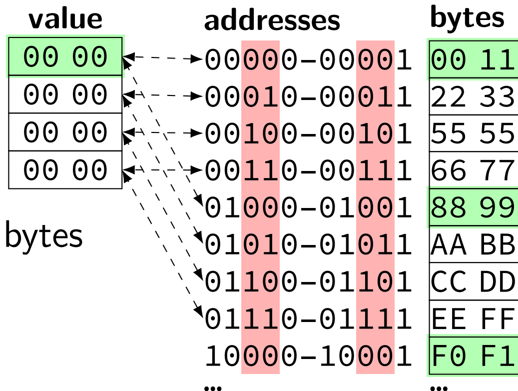
read byte at 01011?

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

Cache

Memory

index
00
01
10
11

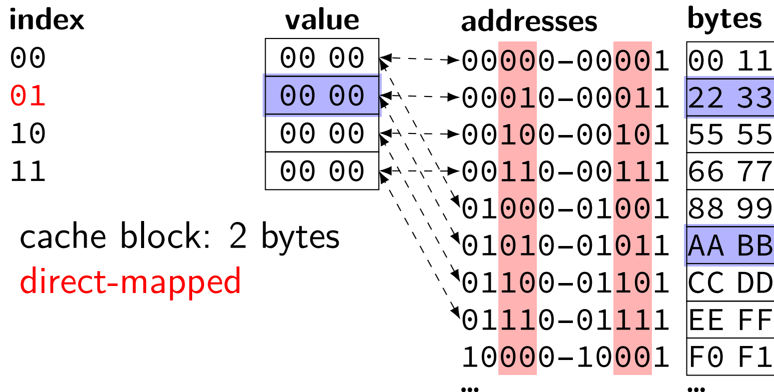


cache block: 2 bytes
direct-mapped

building a (direct-mapped) cache

read byte at 01011?

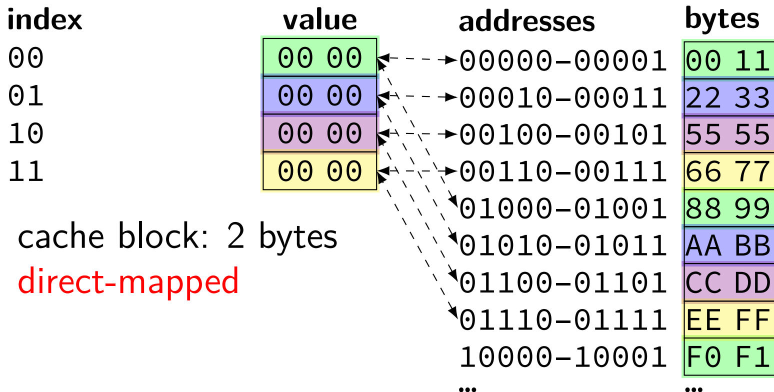
exactly one place for each address
spread out what can go in a block



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?

Cache

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

cache block: 2 bytes

direct-mapped

Memory

addresses	bytes
	1
00010–00011	22 33
0–00101	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
...	...

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

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

need tag to know

cache block: 2 bytes
direct-mapped

Memory

value from 01010 or 00010?

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

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
...	...

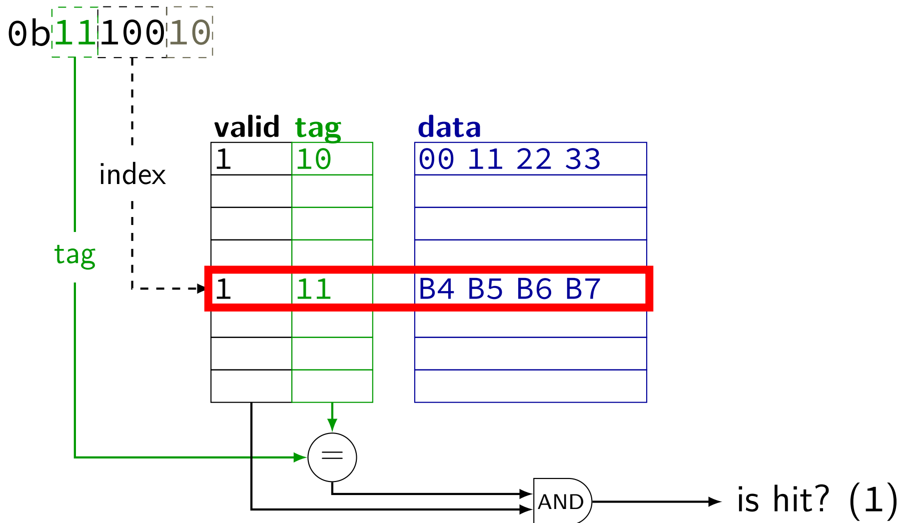
cache operation (read)

0b11100010

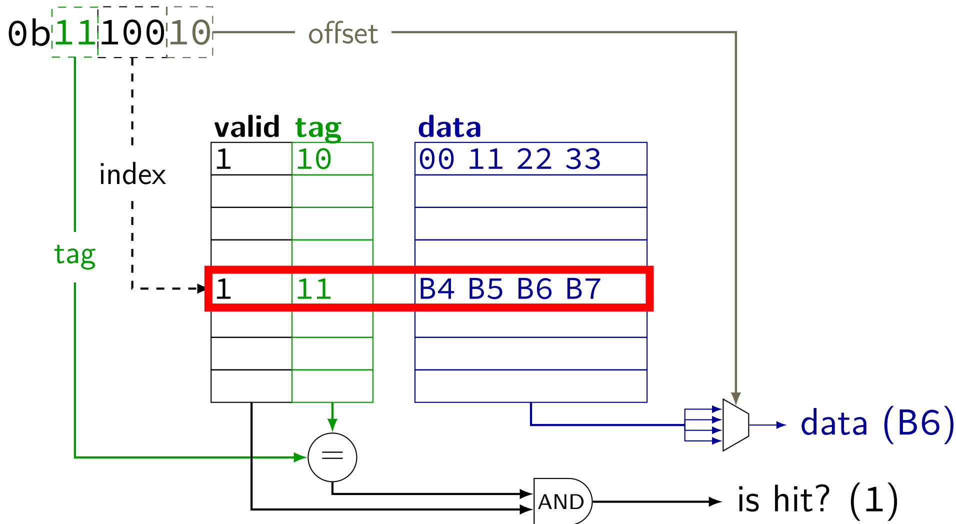
index

valid	tag	data
1	10	00 11 22 33
1	11	B4 B5 B6 B7

cache operation (read)



cache operation (read)



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	00	00 11 22 33
1	1	01	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

$2 = 2^1$ bytes in block

1 bit to say which byte

4 byte blocks, 2 sets

index	valid	tag	value
0	1	00	00 11 22 33
1	1	01	CC DD EE FF

2 byte blocks, 8 sets

index	valid	tag	value
			00 11
			F1 F2
			-- --
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	???	???	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[illegible]

index	valid	tag	va[illegible]ue
0	1	00	00 11 22 33
1	1	01	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
	0	--	-- --
	0	--	-- --
	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

$2^2 = 4$ sets

2 bits to index set

4 byte blocks, 2 sets

index	valid	tag	value
0	1	00	00 11 22 33
1	1	01	CC DD 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

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

4 byte blocks, 2 sets

index	valid	tag	value
0	1	00	00 11 22 33
1	1	01	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	00	00 11 22 33
1	1	01	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	--	-- --
01			--
10			--
10			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	00	00 11 22 33
1	1	01	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 formulas (direct-mapped only)

m	memory addreses bits (Y86-64: 64)
$S = 2^s$	number of sets
s	(set) index bits
$B = 2^b$	block size
b	(block) offset bits
$t = m - (s + b)$	tag bits
$C = B \times S$	cache size (if direct-mapped)

example access pattern (1)

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

2 byte blocks, 4 sets

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

example access pattern (1)

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

2 byte blocks, 4 sets

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

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

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

tag index offset

2 byte blocks, 4 sets

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

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

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

tag index offset

2 byte blocks, 4 sets

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

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

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

tag index offset

2 byte blocks, 4 sets

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

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

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

tag index offset

2 byte blocks, 4 sets

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

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

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

tag index offset

2 byte blocks, 4 sets

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

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

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

tag index offset

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

2 byte blocks, 4 sets

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

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

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

tag index offset

2 byte blocks, 4 sets

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

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

2 byte blocks, 4 sets

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

tag index offset

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

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

2 byte blocks, 4 sets

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

tag index offset

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

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

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

tag index offset

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

2 byte blocks, 4 sets

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
01	1	01100	mem[0x62] mem[0x63]
10	1	01100	mem[0x64] 65]
11	0		

miss caused by conflict

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ 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 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

4 byte blocks, 4 sets

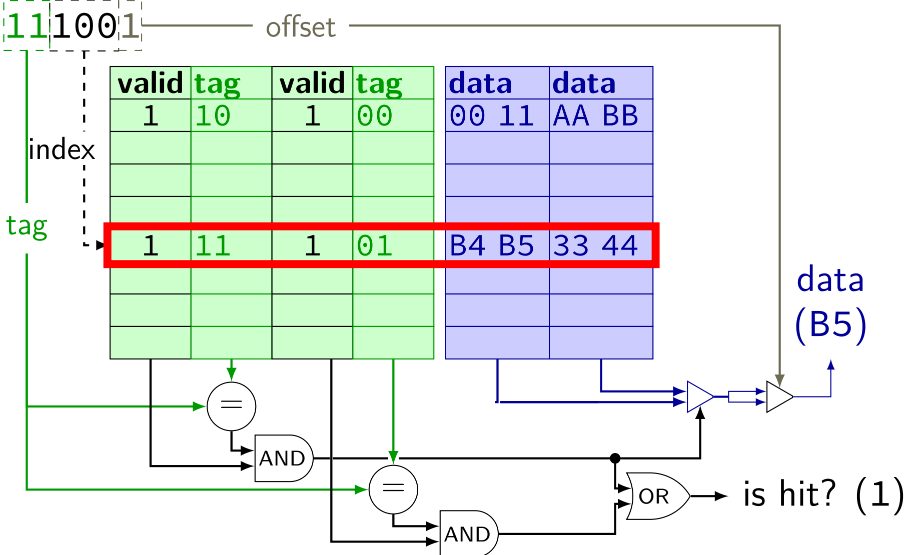
address (hex)	result
00000000 (00)	
00000001 (01)	
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

index	valid	tag	value
00			
01			
10			
11			

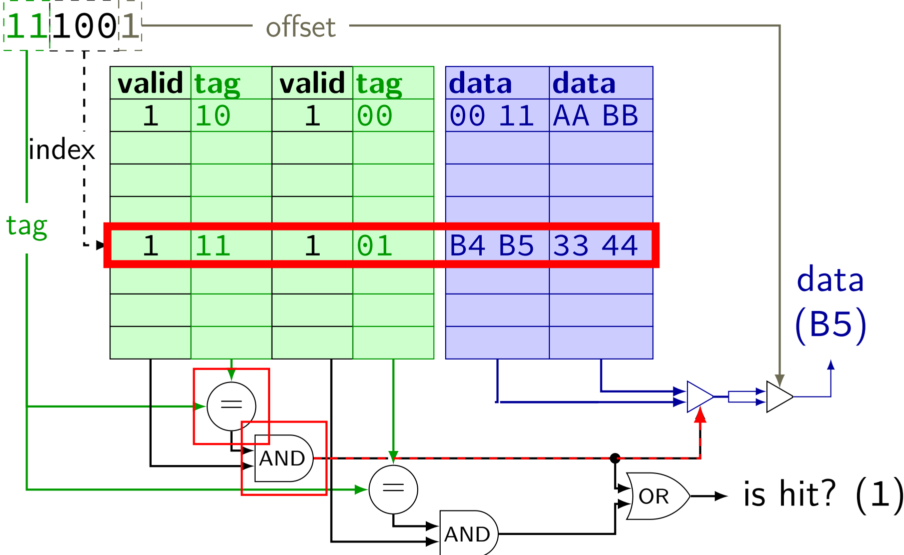
exercise: how many accesses are hits?

backup slides

cache operation (associative)



cache operation (associative)



cache operation (associative)