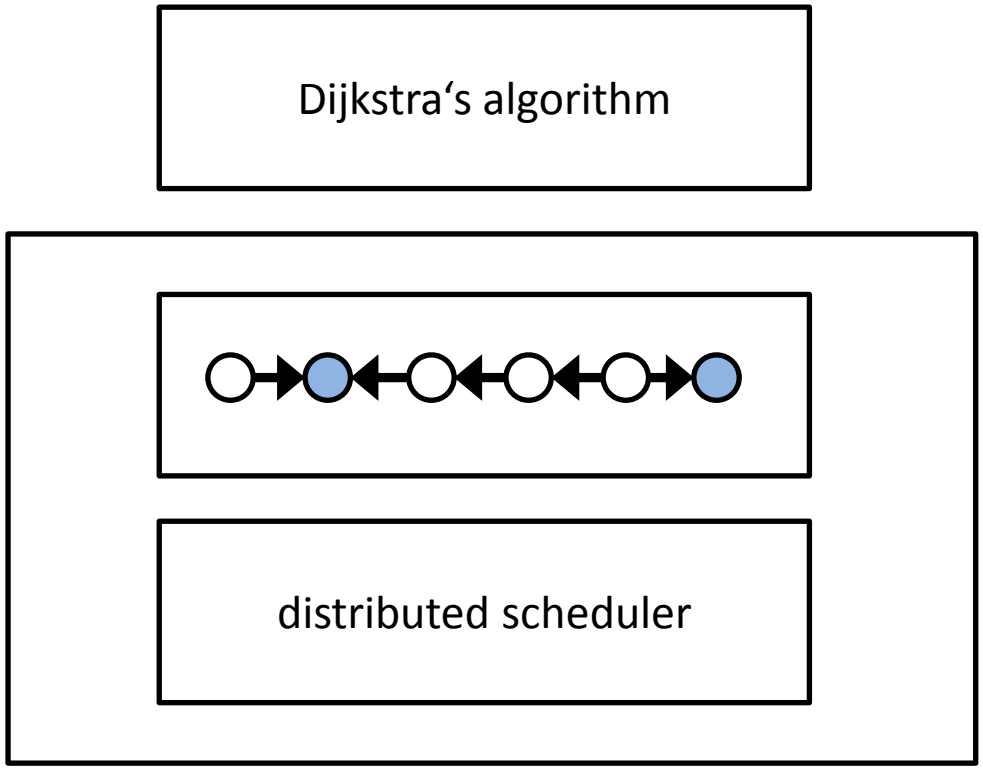
# Beyond classical chip design
# lecture 3

# Self-stabilization (continued)

# What we had...

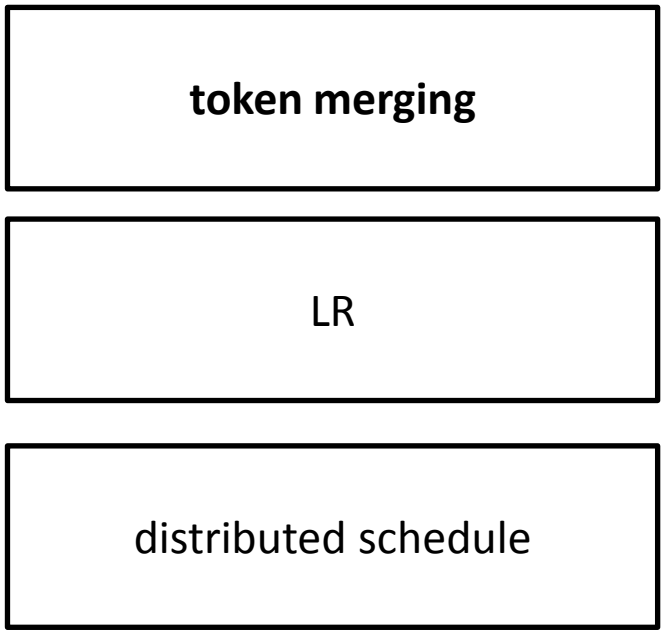
Distributed, weak-fair scheduler ->

Distributed, **neighbour-mutex**, weak fair scheduler.



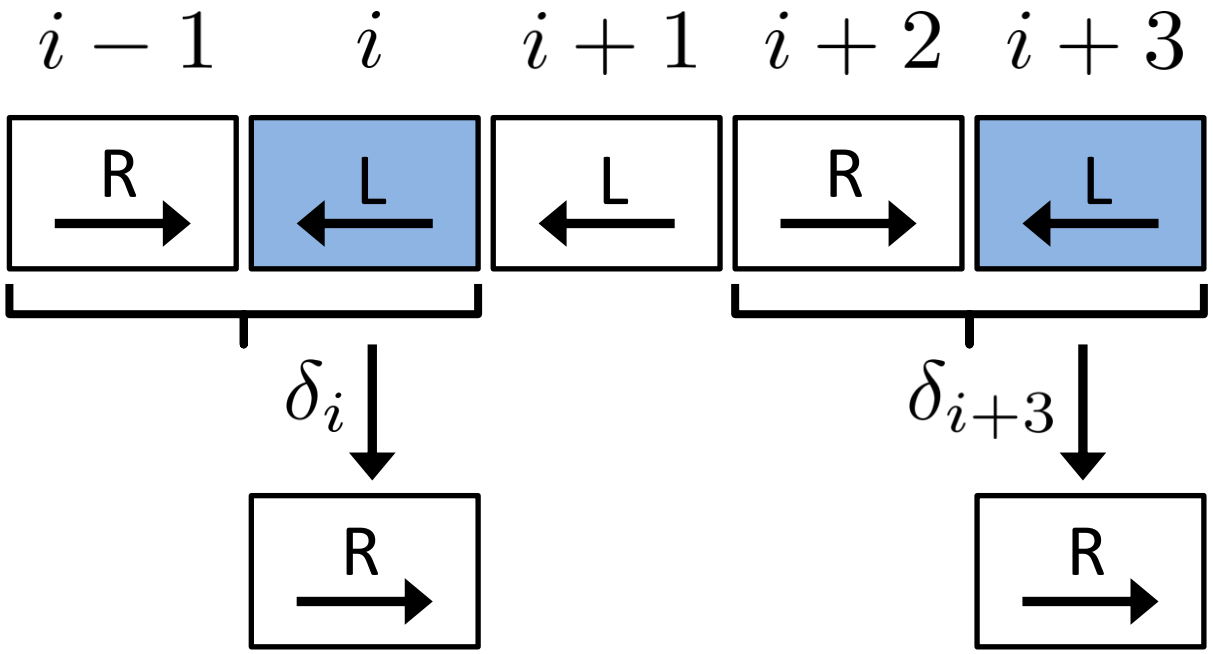
# Self-stabilization

... link reversal almost solves the problem.



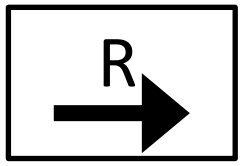
# Self-stabilization

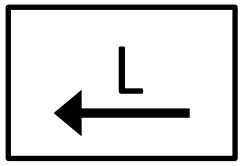
stable algorithm

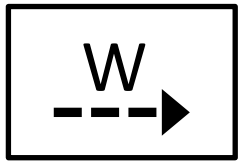


# Self-stabilization

adding direction



# Self-stabilization

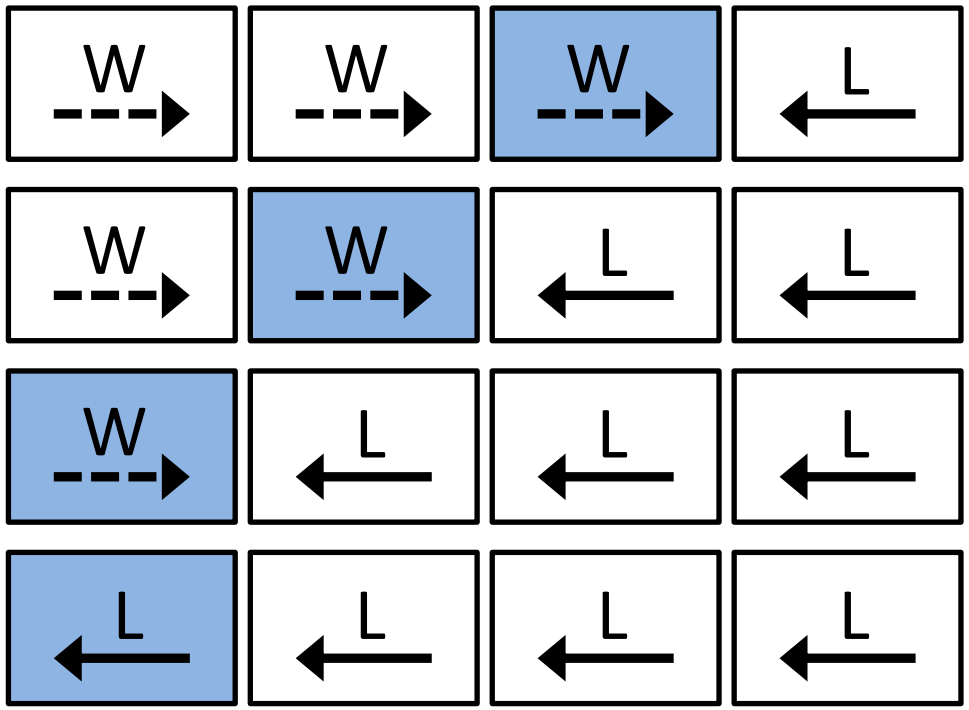
tokens turn only at borders ->

**Prop 1.** Mutex holds.

**Prop 2.** Weak fairness holds.

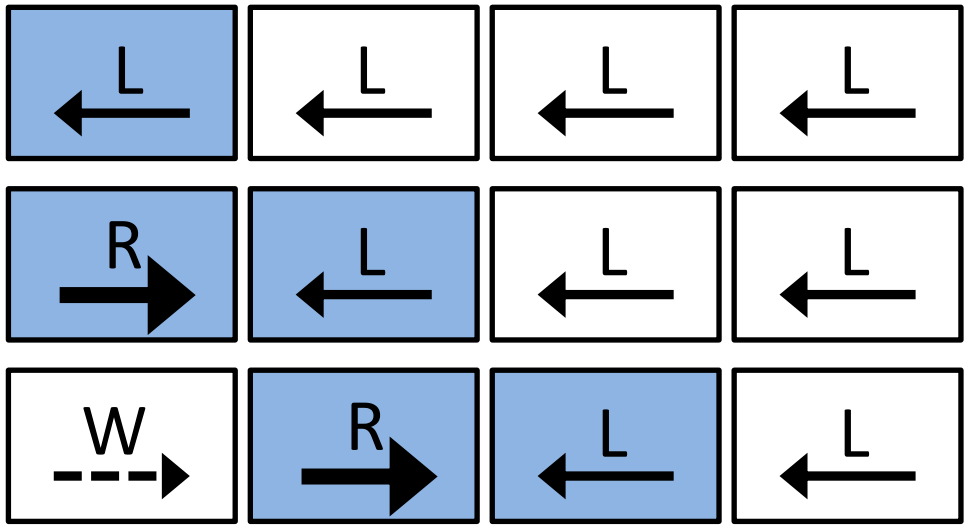
# Self-stabilization

to left ...



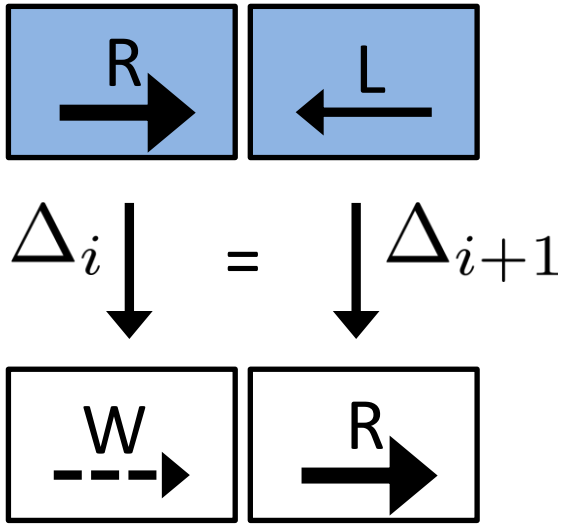
# Self-stabilization

to right ...



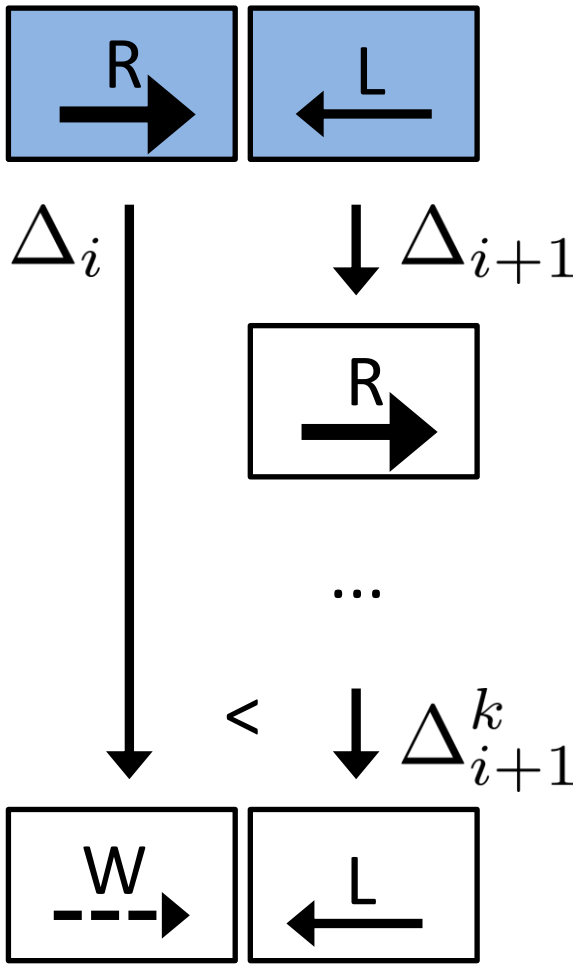

... well

# Self-stabilization



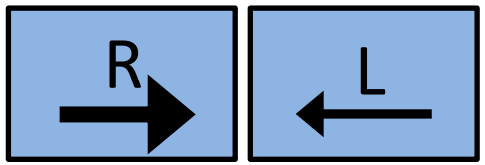

requires simultaneity: two sided constraint!

# Self-stabilization



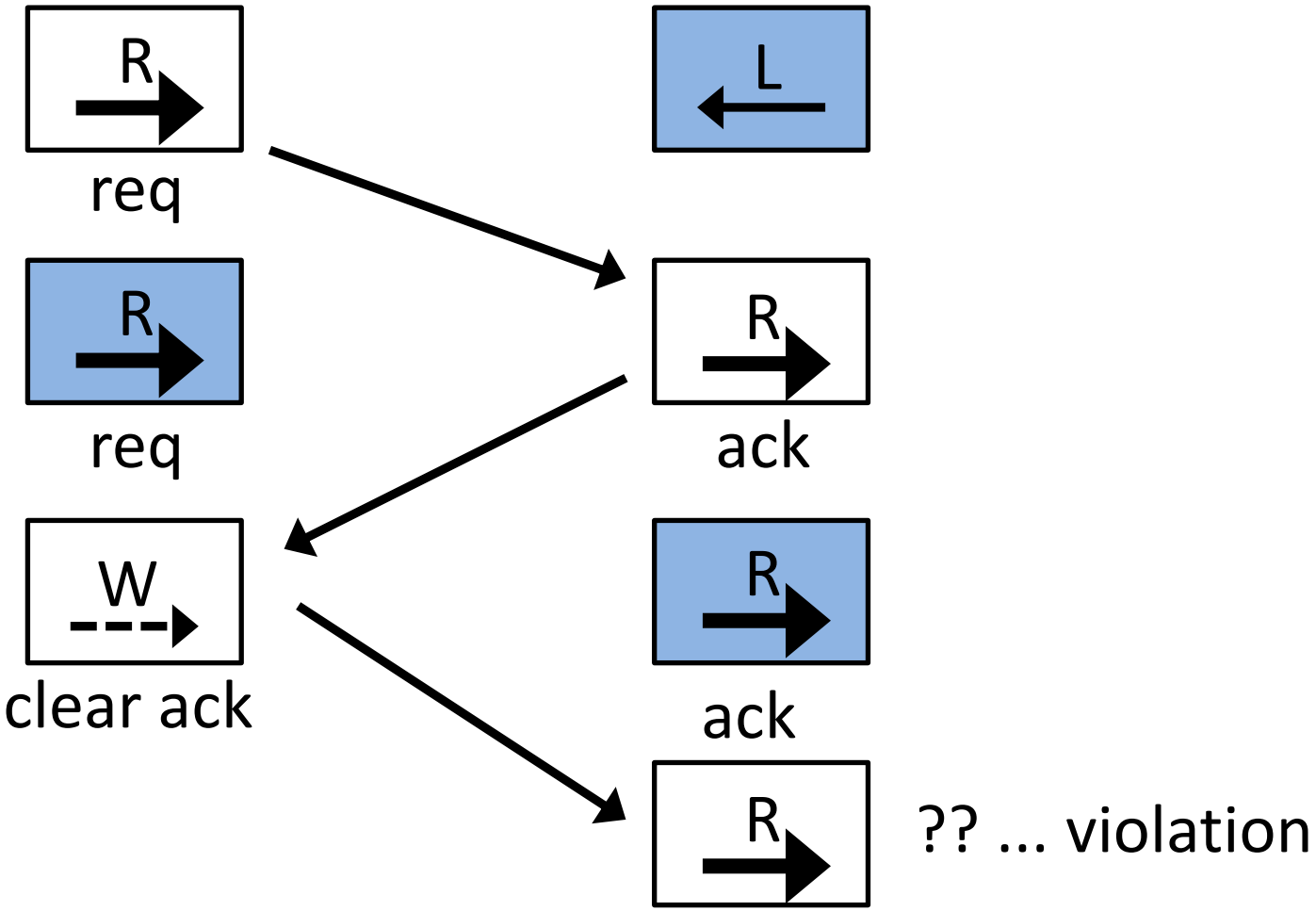

... one sided

# Self-stabilization

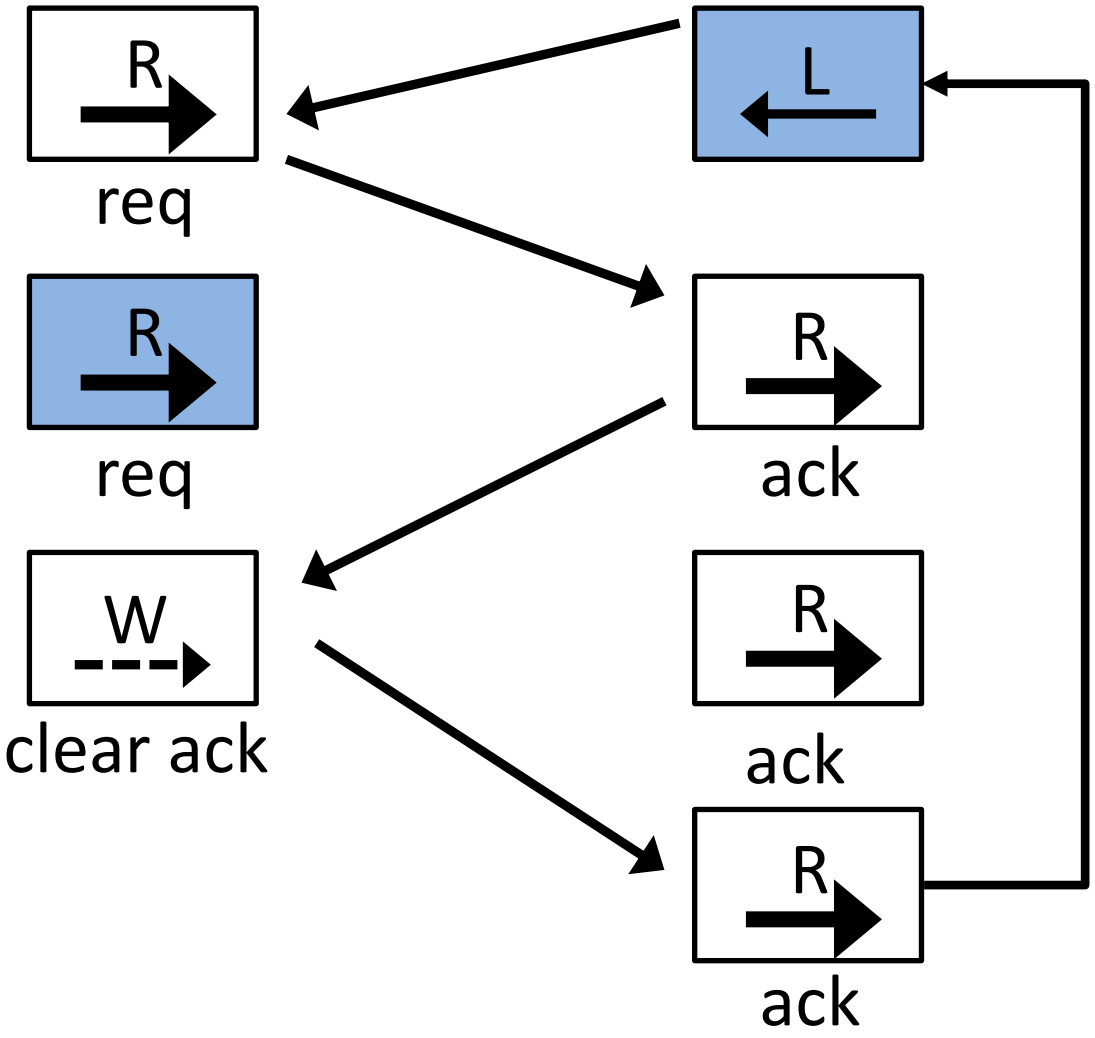



... without timing?

# Self-stabilization



# Self-stabilization



# Beyond classical circuit design
# lecture 3.5

## Circuit model

# Further Reading

Alain J. Martin: *Synthesis of Asynchronous VLSI Circuits.* Tech report California Institute of Technology, 1991.

Alain J. Martin and Mika Nyström: *Asynchronous techniques for system-on-chip design.* Proceedings of the IEEE Volume 94, Issue 6:1089 - 1120, June 2006.

# Binary, event based model

here [Alain Martin]:

low-level: production rules.

high-level: communicating hardware processes.

# Low-level Specifications

Production rules

# Production rules

variable/port: from a finite alphabet $V$

transition: variable + up/down

production rule: Boolean guard -> transition

$$x \wedge y \rightarrow z\uparrow$$
$$\neg(x \wedge y) \rightarrow z\downarrow$$

# Production rules

$x \wedge y \rightarrow z \uparrow$

$\neg(x \wedge y) \rightarrow z \downarrow$

typically rule-pairs

**non-interference**: per rule-pair $\neg(Bu \wedge Bd)$

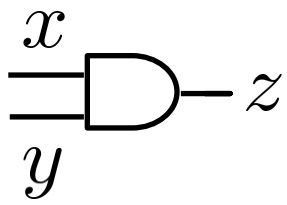
**no self-reference:** per rule

# Gate

gate = rule-pair

**combinational** (NOT, 2AND, 2OR, AOIs, ...)

$$Bu \leftrightarrow \neg Bd$$




$$x \wedge y \rightarrow z \uparrow$$

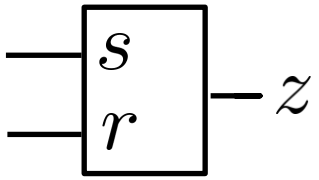
$$\neg(x \wedge y) \rightarrow z \downarrow$$

# Gate

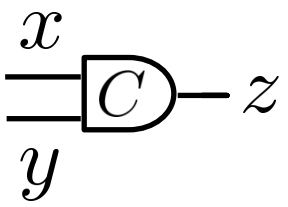
gate = rule-pair

**state holding**

set-reset latch $s \rightarrow z\uparrow$ $r \rightarrow z\downarrow$

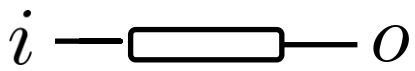



2C-Element $x \wedge y \rightarrow z\uparrow$ $\neg x \wedge \neg y \rightarrow z\downarrow$



# Wire

= special gate



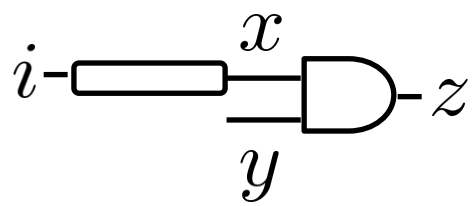

$i \to o \uparrow$

$\neg i \to o \downarrow$

# Production rules

circuit = algorithm = set of production rules

$$Au : x \wedge y \rightarrow z \uparrow$$
$$Ad : \neg(x \wedge y) \rightarrow z \downarrow$$
$$Du : i \rightarrow x \uparrow$$
$$Dd : \neg i \rightarrow x \downarrow$$




environment = set of production rules

# Execution

global state $s : V \mapsto \{0, 1\}$

enabled rule, step

execution $(s_n)_{n \geq 0}$

constraints: (weak) fairness, partial order, timed

# Hardware design

Given basic building blocks, implement the specification.

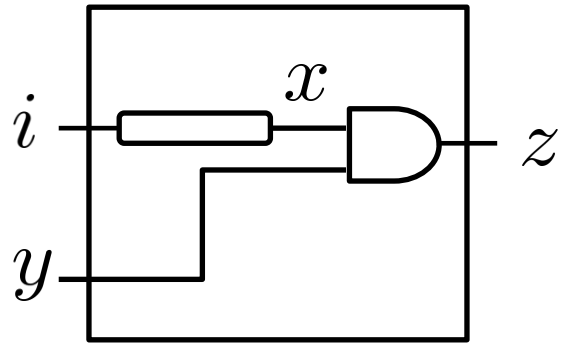
# Circuit A implements circuit B

observable variables $\mathcal{O}$

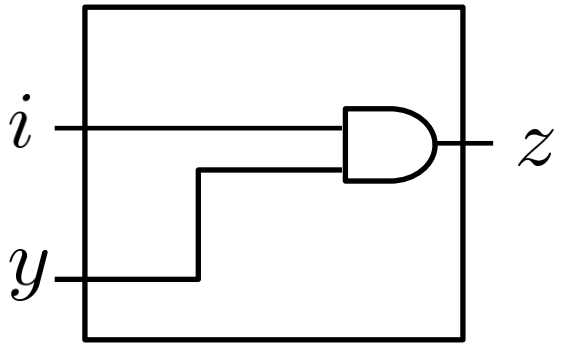
trace inclusion

$$E_A \restriction \mathcal{O} \subseteq E_B \restriction \mathcal{O}$$

A:




B:



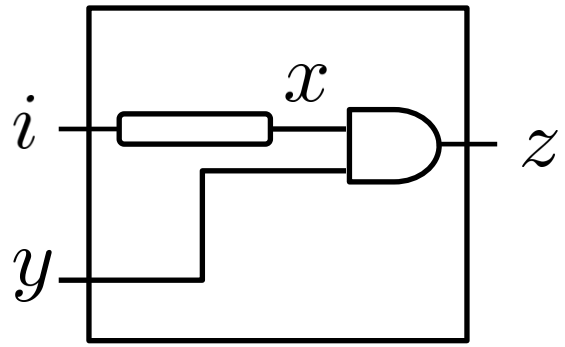
# Circuit A implements circuit B

$i\uparrow x\uparrow i\downarrow y\uparrow z\uparrow \mapsto$

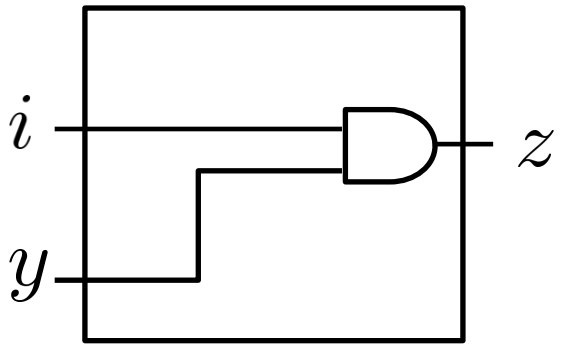
$i\uparrow i\downarrow y\uparrow z\uparrow$

-> A does not implement B

A:

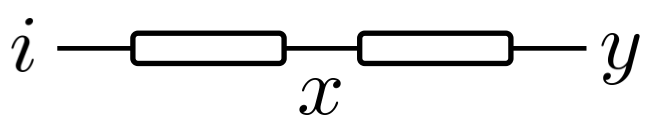



B:

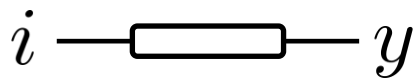


# Mind...

wire + wire “is” not a (long) wire

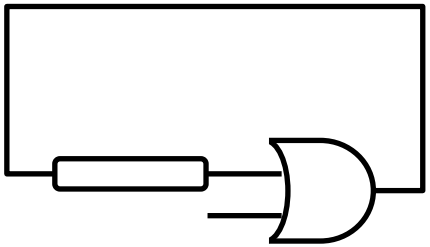


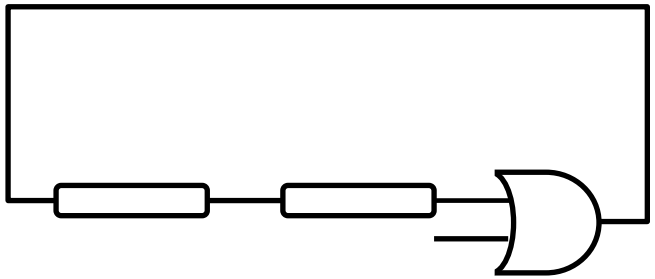

$$i\uparrow, x\uparrow, i\downarrow, y\uparrow, x\downarrow, y\downarrow$$



# Mind...

wire + wire “is” not a (long) wire

-> 

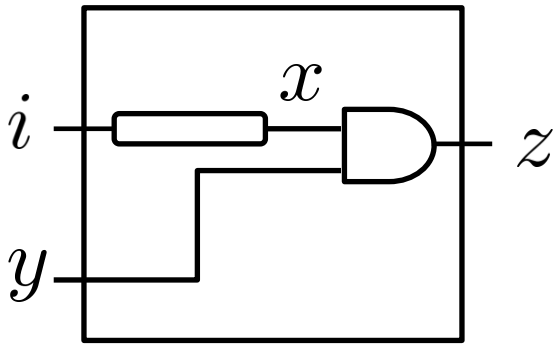
vs. 

oscillations?! [hw]

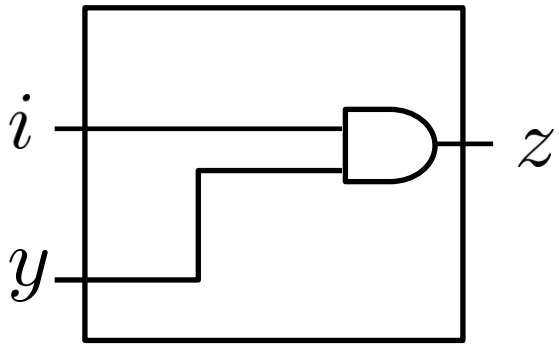
# “implements”

Simulation.

A:




B:

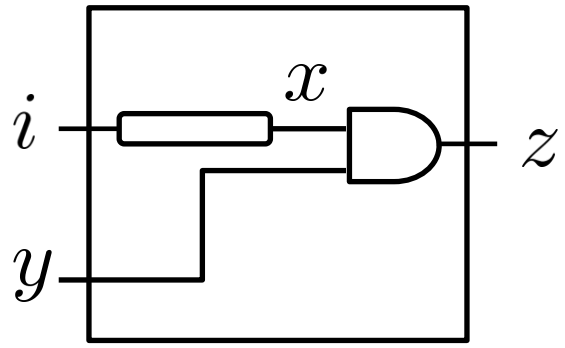



$i \uparrow$

# “implements”

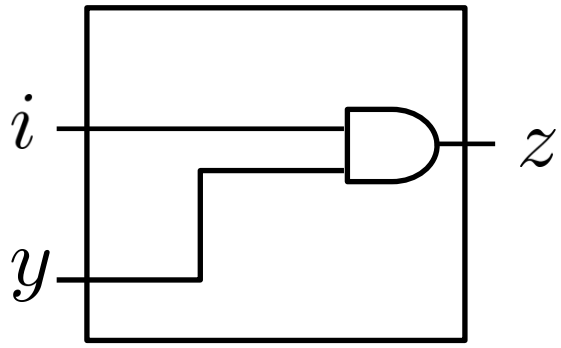
Simulation.

A:




$i \uparrow$

B:

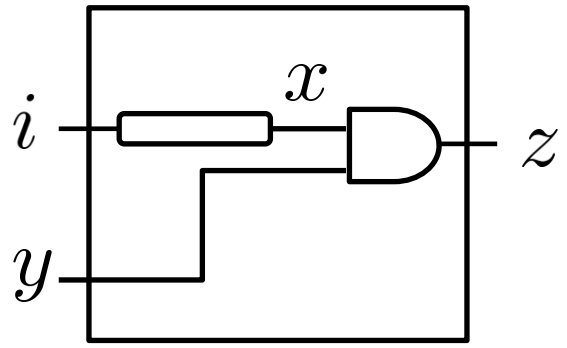



$i \uparrow$

# “implements”

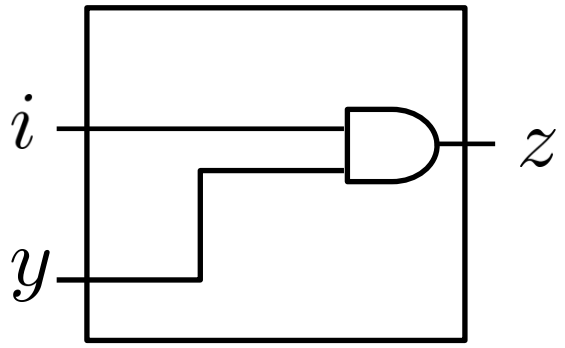
Simulation.

A:




$i\uparrow$

B:

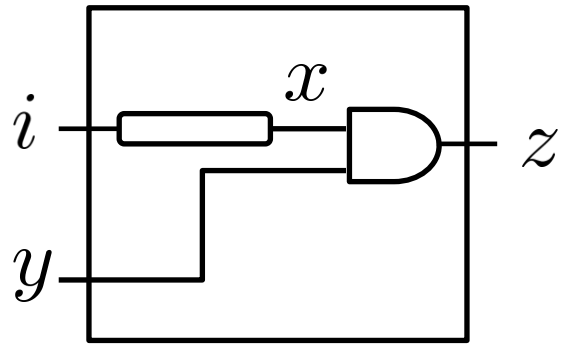



$i\uparrow i\downarrow$

# “implements”

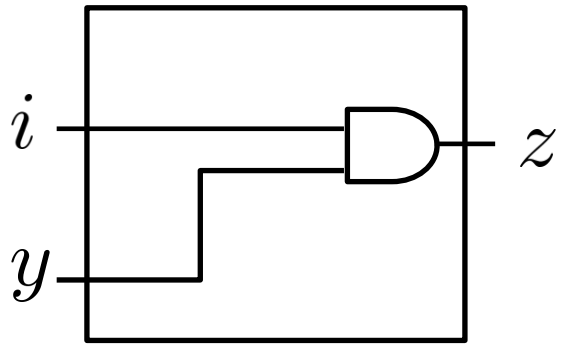
Simulation.

A:




$i \uparrow i \downarrow$

B:

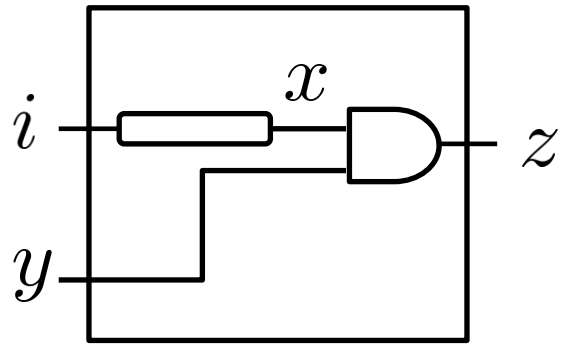



$i \uparrow i \downarrow$

# “implements”

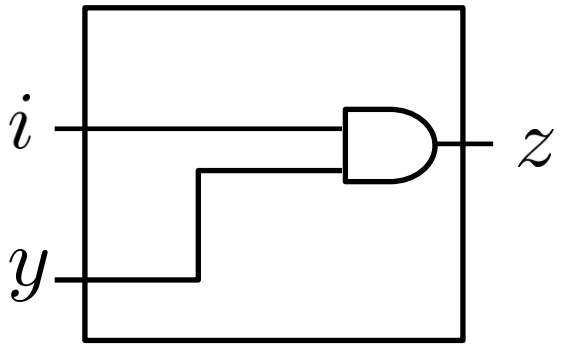
Simulation.

A:




$i \uparrow i \downarrow$

B:

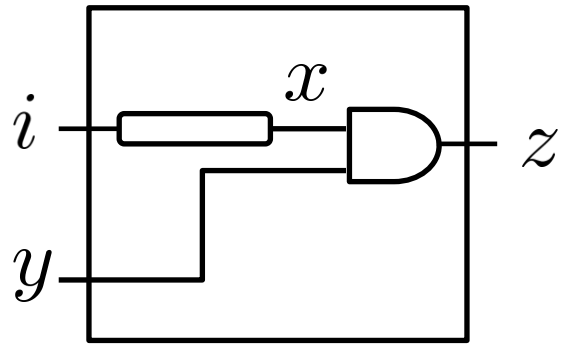



$i \uparrow i \downarrow y \uparrow$

# “implements”

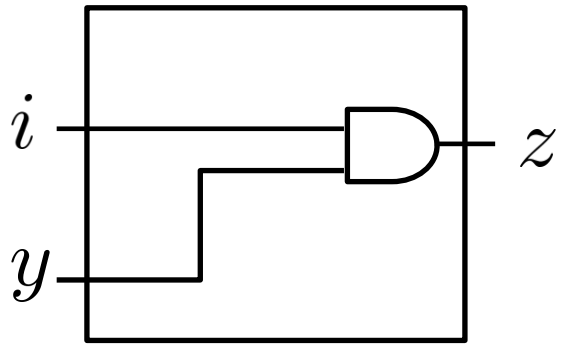
Simulation.

A:




$i \uparrow i \downarrow y \uparrow$

B:

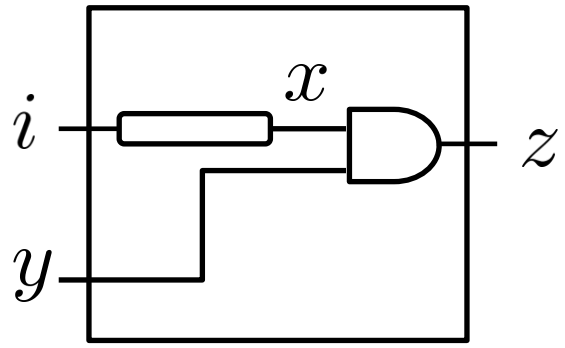



$i \uparrow i \downarrow y \uparrow$

# “implements”

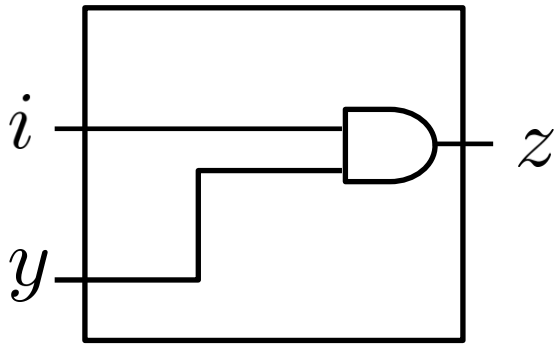
Simulation.

A:




$i \uparrow i \downarrow y \uparrow$

B:

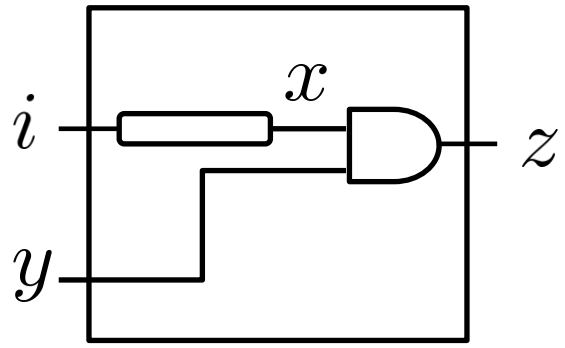



$i \uparrow i \downarrow y \uparrow i \uparrow$

# “implements”

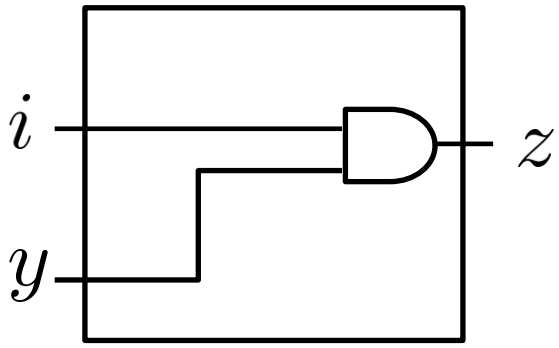
Simulation.

A:




$i\uparrow i\downarrow y\uparrow i\uparrow$

B:

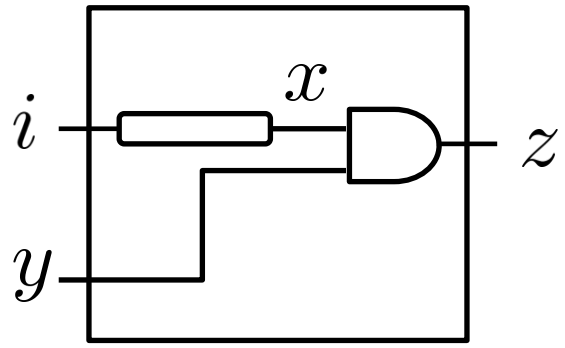



$i\uparrow i\downarrow y\uparrow i\uparrow$

# “implements”

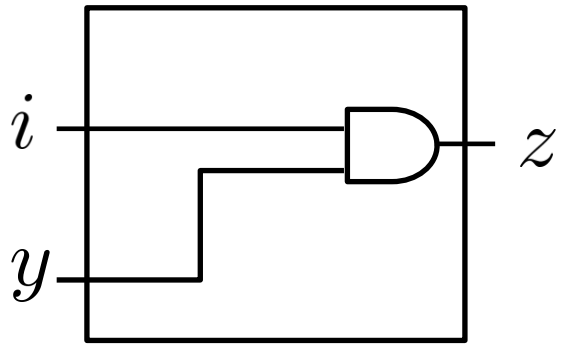
Simulation.

A:




$i\uparrow i\downarrow y\uparrow i\uparrow$

B:

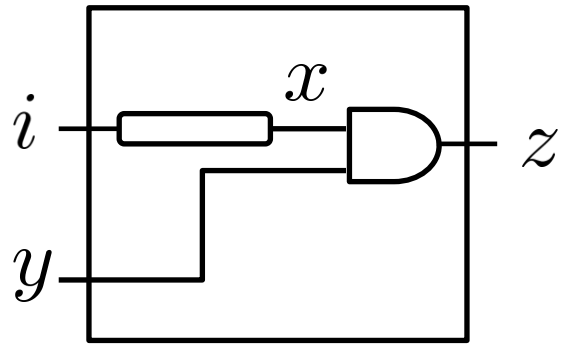



$i\uparrow i\downarrow y\uparrow i\uparrow z\uparrow$

# “implements”

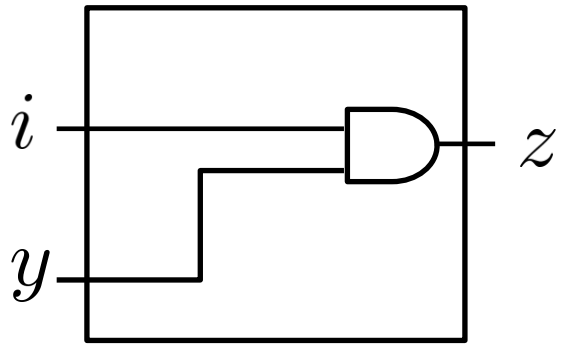
Simulation.

A:




$i\uparrow i\downarrow y\uparrow i\uparrow x\uparrow z\uparrow$

B:




$i\uparrow i\downarrow y\uparrow i\uparrow z\uparrow$

# “implements”

A can simulate B.

Game rules:

- B makes a sequence of steps:

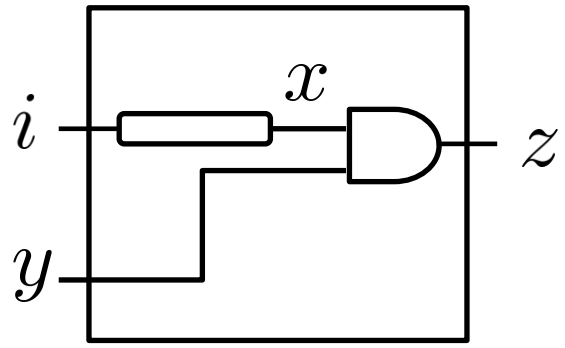
  non-observables with ending observable

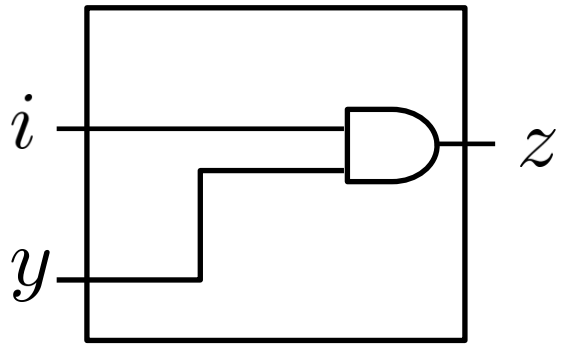
- A makes a sequence of steps:

  non-observables with same ending observable

# “implements”

A can simulate B -> B implements A [hw]




$i\uparrow i\downarrow y\uparrow i\uparrow x\uparrow z\uparrow$

$i\uparrow i\downarrow y\uparrow i\uparrow z\uparrow$

# “implements”

A can simulate B -> B implements A [hw]

Simulation is an efficient test for implementation.

# “implements”

A can simulate B -> B implements A [hw]

Simulation is an efficient test for implementation.

Is “can simulate” also necessary?

# “implements”

A can simulate B <- B implements A ?

# “implements”

$$\mathcal{O} = \{a, b, c, d\}$$

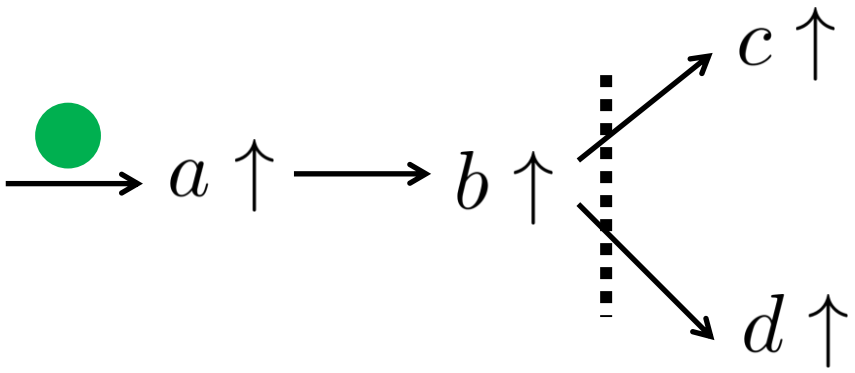
$$\top \rightarrow a \uparrow$$

$$[\bot \rightarrow a \downarrow]$$

$$a \rightarrow b \uparrow$$

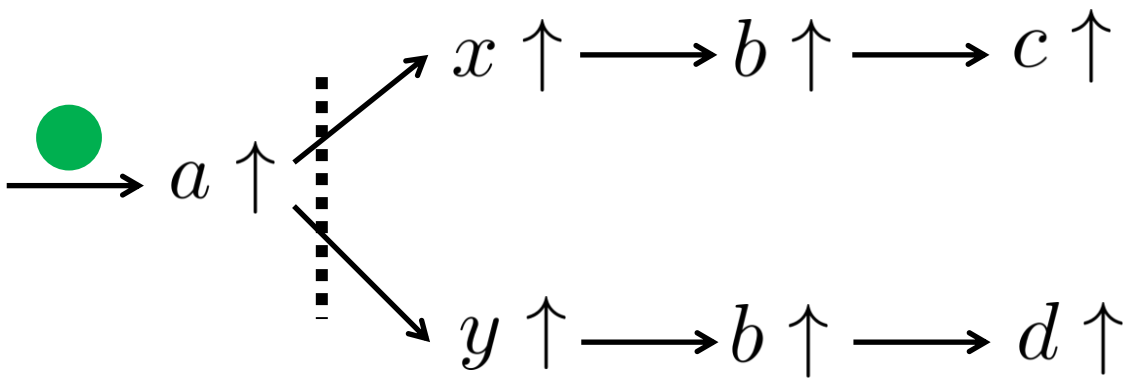
$$b \wedge \neg d \rightarrow c \uparrow$$

$$b \wedge \neg c \rightarrow d \uparrow$$



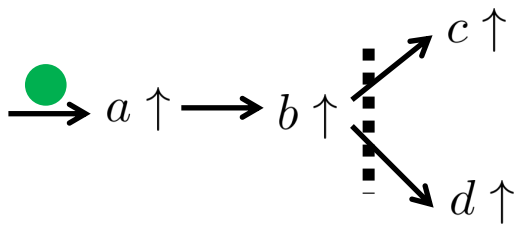
# “implements”

$$\mathcal{O} = \{a, b, c, d\}$$

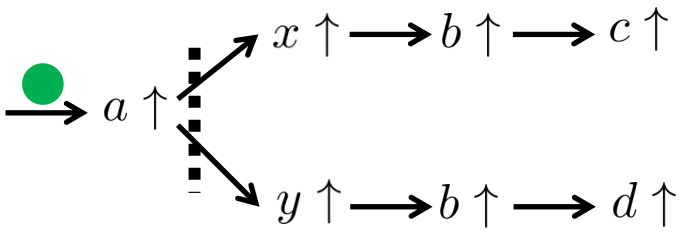


# “implements”

Circuit A




Circuit B

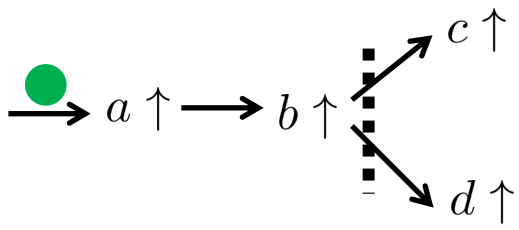



B implements A and A implements B.

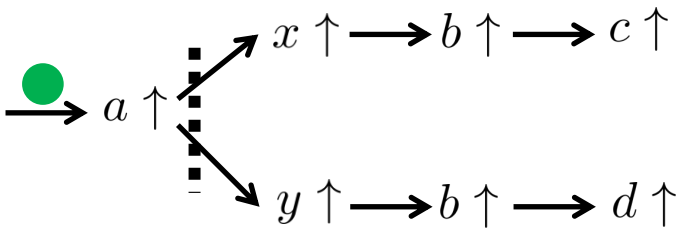
A can simulate B

# “implements”

Circuit A




Circuit B




B implements A and A implements B.

A can simulate B but B **cannot** simulate A.

# “implements”

-> other notions of “can simulate”