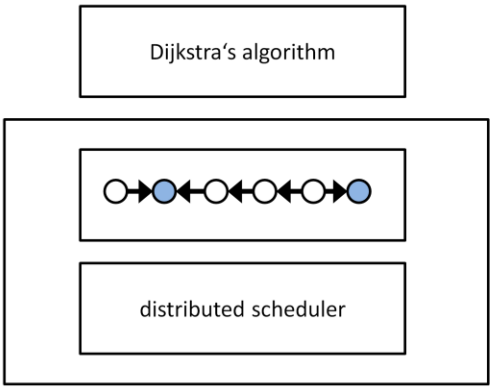
Beyond classical chip design
lecture 3

Self-stabilization (continued)

What we had...

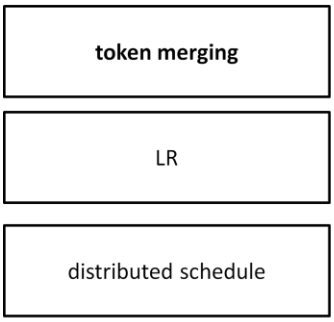
Distributed, weak-fair scheduler ->

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



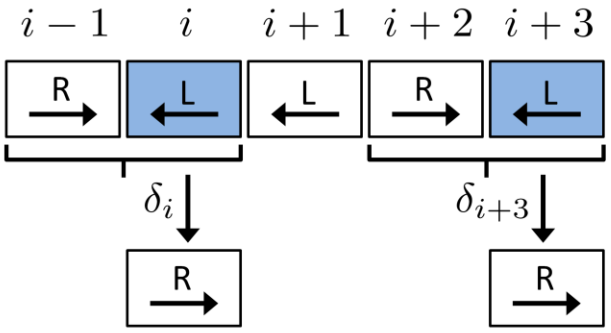
Self-stabilization

... link reversal almost solves the problem.



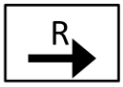
Self-stabilization

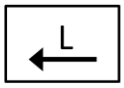
stable algorithm



Self-stabilization

adding direction





Self-stabilization

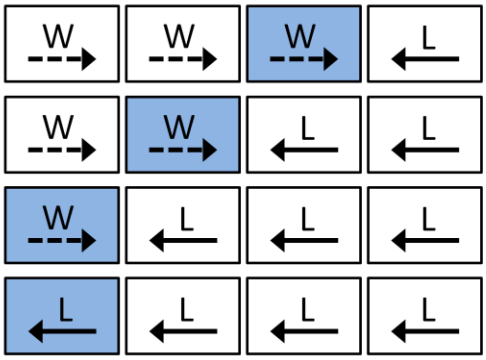
tokens turn only at borders ->

Prop 1. Mutex holds.

Prop 2. Weak fairness holds.

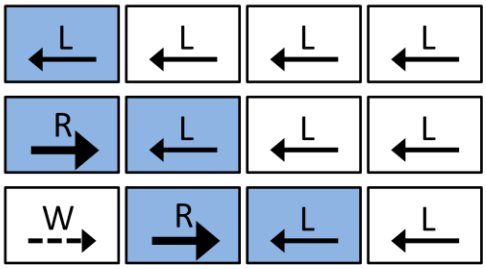
Self-stabilization

to left ...



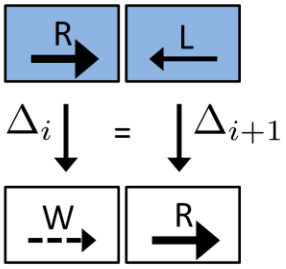
Self-stabilization

to right ...



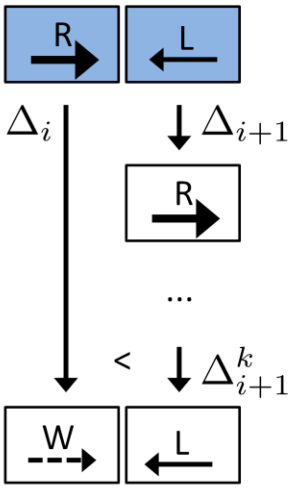
... well

Self-stabilization



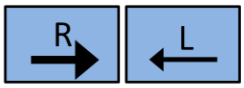
requires simultaneity: two sided constraint!

Self-stabilization

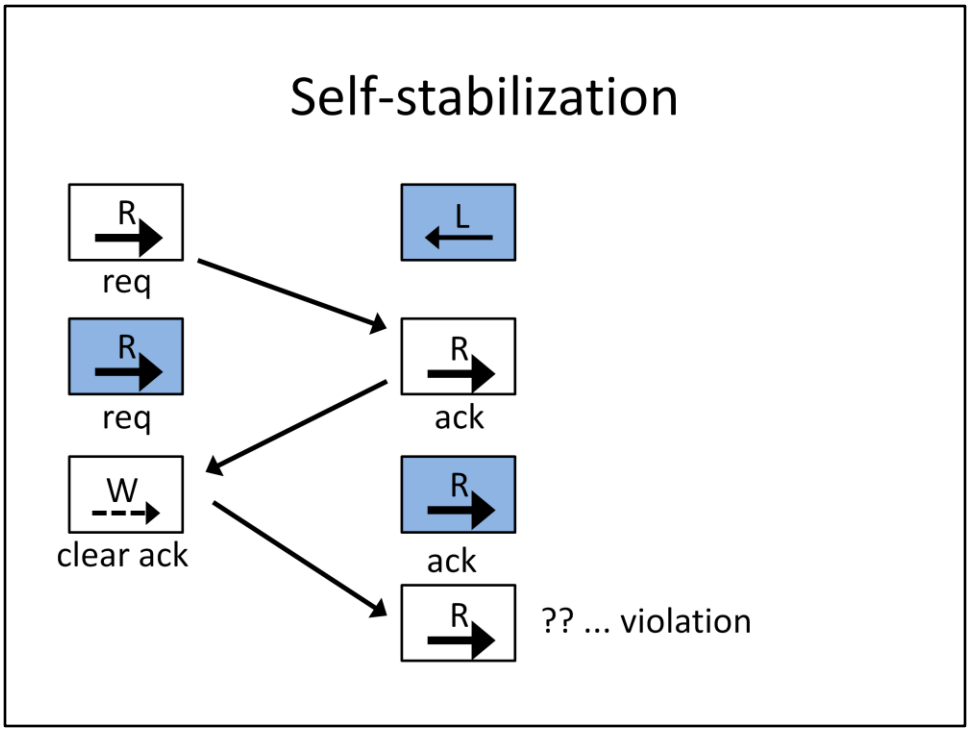


... one sided

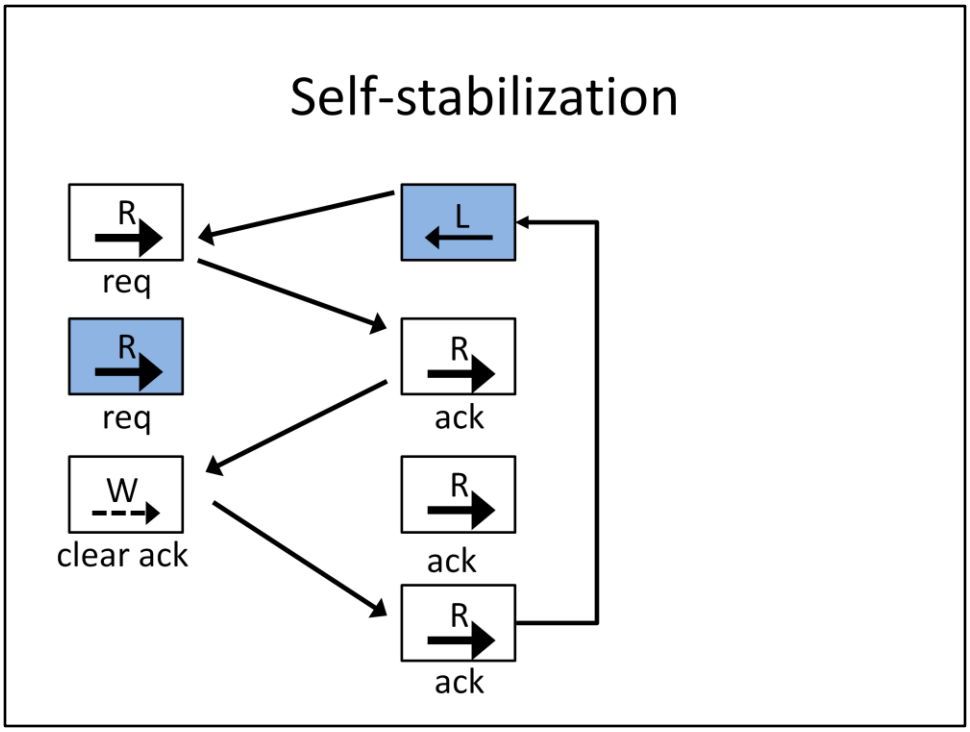
Self-stabilization



... without timing?



violation since right node with state „R ack“ makes a step to „R“ in response to left one, not looking to left one. This could result in unstable change.



no violation workaround: right node with „R ack“ waits with removing ack until it looks left again.

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 $\neg(Bu \wedge Bd)$

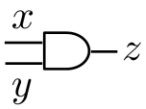
no self-reference: per rule

Gate

gate = rule-pair

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

$$Bu \leftrightarrow \neg Bd$$



$x \wedge y \rightarrow z \uparrow$

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

AOI = AND-OR-INVERT gate

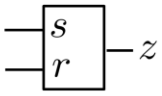
Gate

gate = rule-pair

state holding

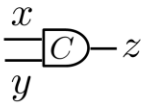
set-reset latch $s \rightarrow z\uparrow$

$r \rightarrow z\downarrow$



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

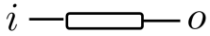
$\neg x \wedge \neg y \rightarrow z\downarrow$



set-reset latch: unspecified what happens if s=r=1. Is disallowed by non-interference. When using this gate: Make sure that non-interference is valid in all executions.

Wire

= special gate



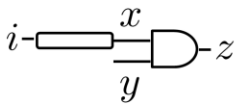
$i \to o \uparrow$

$\neg i \to o \downarrow$

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.

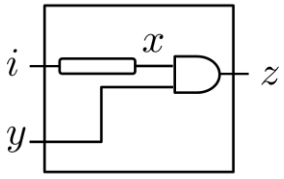
Circuit A implements circuit B

observable variables $\mathcal{O}$

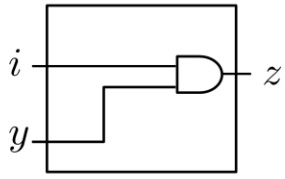
trace inclusion

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

A:



B:



if circuit A produces behavior that could be from circuit B as well, A implements B. Typically B is a circuit specification. The executions of B are the allowed executions. If A's executions all are in the set of the allowed executions we say A implements (specification) B.

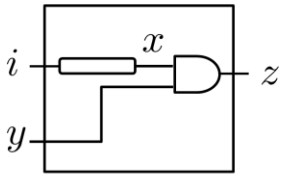
Circuit A implements circuit B

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

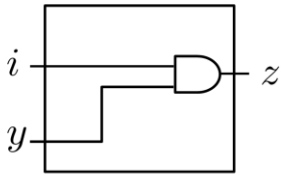
$i\uparrow i\downarrow y\uparrow z\uparrow$

-> A does not implement B

A:

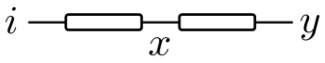


B:

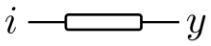


Mind...

wire + wire “is” not a (long) wire



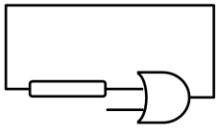
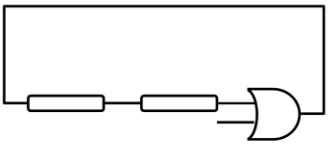
$i\uparrow, x\uparrow, i\downarrow, y\uparrow, x\downarrow, y\downarrow$



„is“ here means: implements in both directions: A is B if A implements B and B implements A.

Mind...

wire + wire “is” not a (long) wire

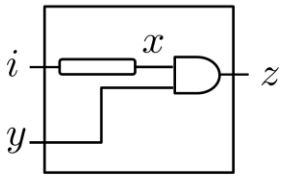
->  vs. 

oscillations?! [hw]

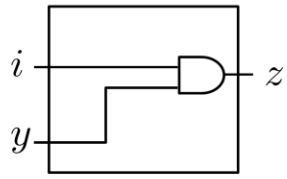
“implements”

Simulation.

A:



B:

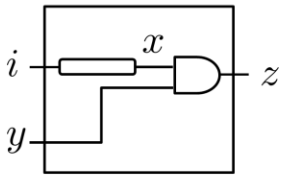


$i \uparrow$

“implements”

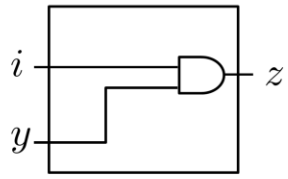
Simulation.

A:



$i \uparrow$

B:

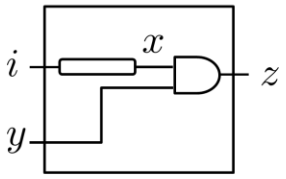


$i \uparrow$

“implements”

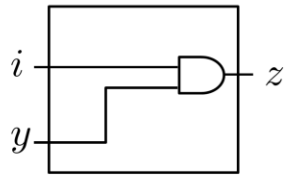
Simulation.

A:



$i \uparrow$

B:

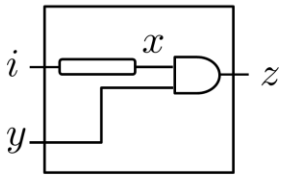


$i \uparrow i \downarrow$

“implements”

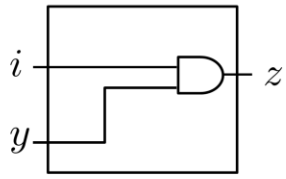
Simulation.

A:



$i \uparrow i \downarrow$

B:

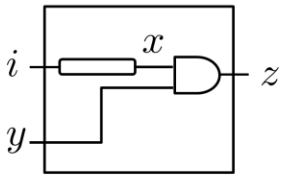


$i \uparrow i \downarrow$

“implements”

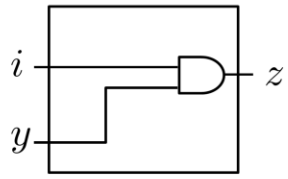
Simulation.

A:



$i \uparrow i \downarrow$

B:

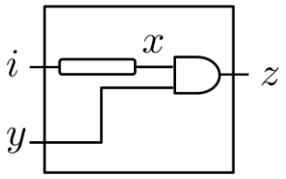


$i \uparrow i \downarrow y \uparrow$

“implements”

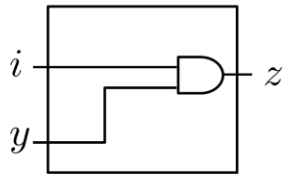
Simulation.

A:



$i \uparrow i \downarrow y \uparrow$

B:

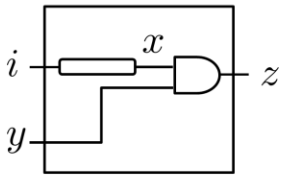


$i \uparrow i \downarrow y \uparrow$

“implements”

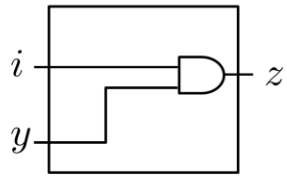
Simulation.

A:



$i \uparrow i \downarrow y \uparrow$

B:

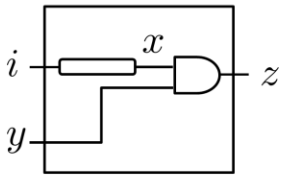


$i \uparrow i \downarrow y \uparrow i \uparrow$

“implements”

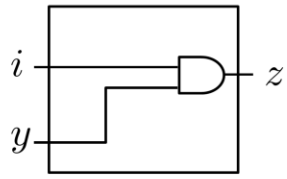
Simulation.

A:



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

B:

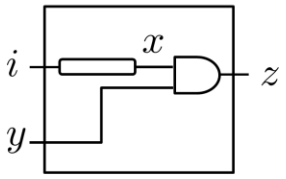


$i \uparrow i \downarrow y \uparrow i \uparrow$

“implements”

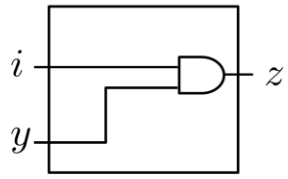
Simulation.

A:



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

B:

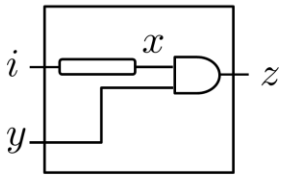


$i \uparrow i \downarrow y \uparrow i \uparrow z \uparrow$

“implements”

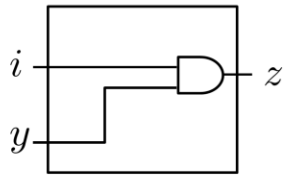
Simulation.

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:

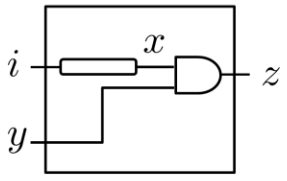
  non-observables with ending observable

- A makes a sequence of steps:

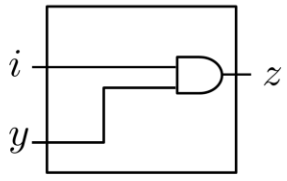
  non-observables with same ending observable

“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"

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

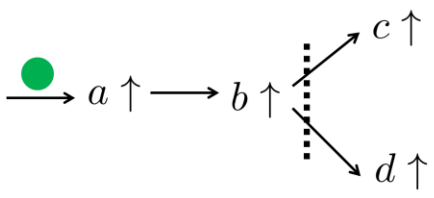
$\top \rightarrow a \uparrow$

$[\bot \rightarrow a \downarrow]$

$a \rightarrow b \uparrow$

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

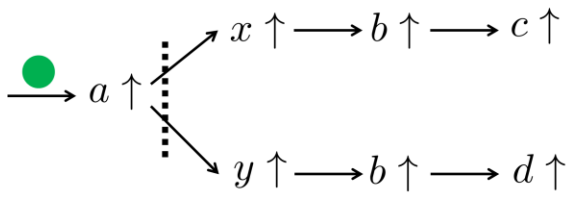
$b \wedge \neg c \rightarrow d \uparrow$



left side: circuit. right side: Petri-net-like graphical representation of the causal structure (what event causes what event) of the circuit. the green dot represents the initial event triggered. the dotted bar means that not both branchs can be taken, but only one of them (mutually exclusive)

“implements”

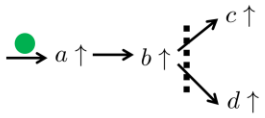
$\mathcal{O} = \{a, b, c, d\}$



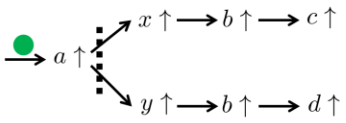
Note that again we could write the circuit for this graphical representation in our circuit model notation.

"implements"

Circuit A



Circuit B



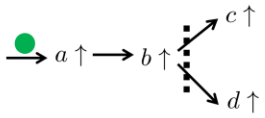
B implements A and A implements B.

A can simulate B

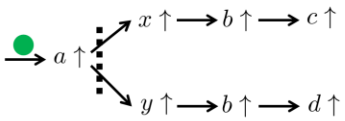
A implements B and B implements A: we know this since the only 2 executions projected to observable variables in both circuits are: a goes high, b goes high, c goes high and
a goes high, b goes high, d goes high.

“implements”

Circuit A



Circuit B



B implements A and A implements B.

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

The problem is that circuit A may decide later than B which branch it takes.

“implements”

-> other notions of “can simulate”

There are more powerfull notions where player A can reverse some of ist steps (e.g., it can reverse the last k steps). Still such simulation relations are not equivalent with the implementation relation.