Beyond classical chip design
lecture 5

Communication

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.

What we had...

- Communicating Hardware Processes as high-level spec.

- fundamental problem:

  Sequencing: a;b

  Multishot-sequencing: *[a;b]

... now look at communication

multishot-sequencing

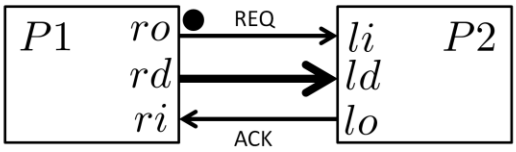
communication:

a <-> write(x)

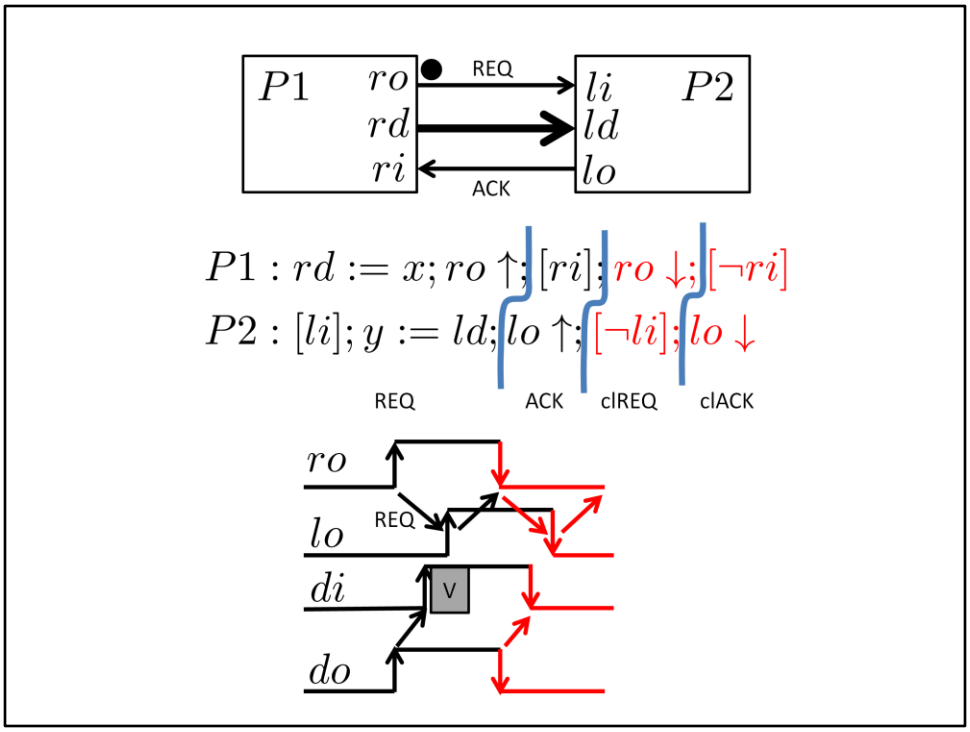
b <-> y = read(x)

communication: requires sequencing as sub-problem.

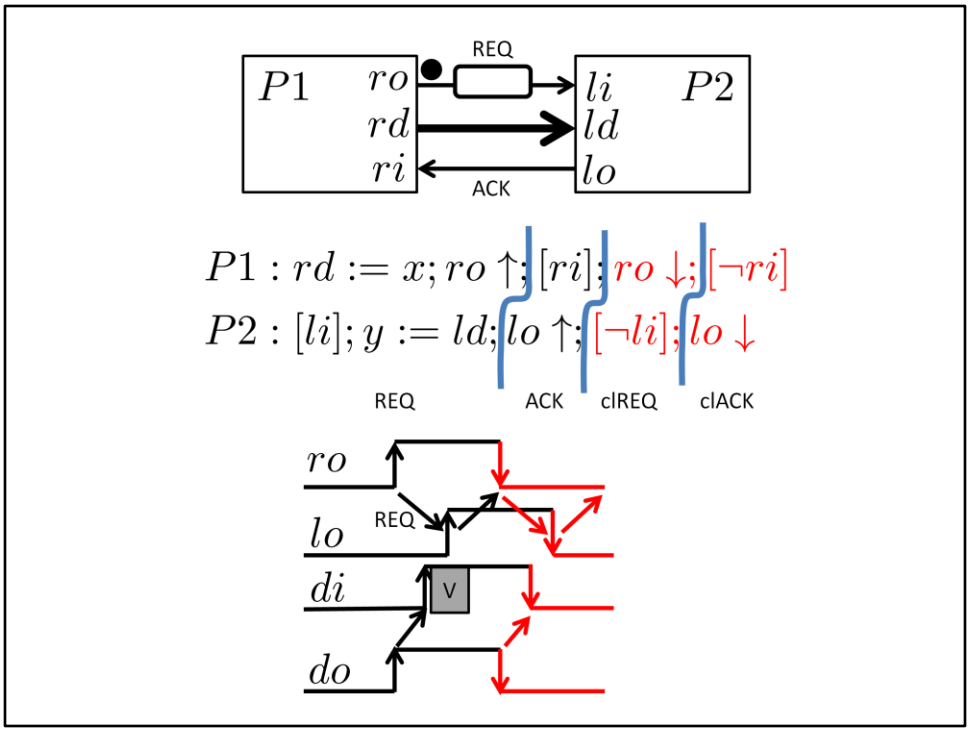
Bundled Data



until now: communication only for sequencing but not for transmitting data.
now: sequencing + data transfer
1st communication paradigm: bundled data. most similar to clocked circuit design communication

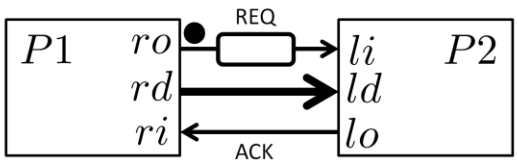


4 phase handshake + data
V = valid.
data has to be valid and stay valid for some time when req arrives at reader P2 -> timing conditions



making REQ line slower than data to ensure timing condition: REQ arrives after data valid at reader:
one sided constraint

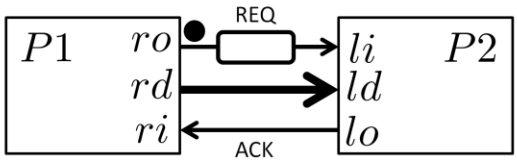
Bundled Data

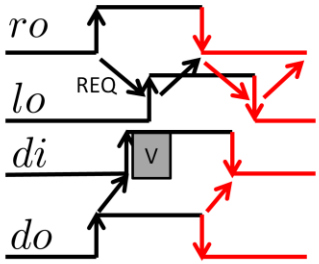


… requires one-sided constraint

Can we do without?

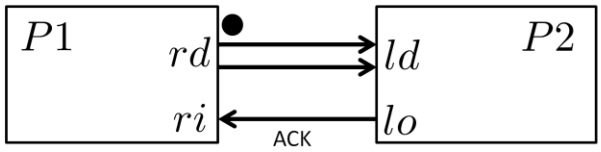
Bundled Data

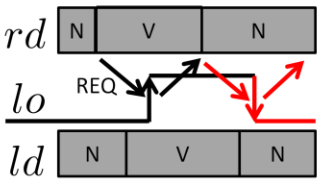




instead of sending REQ directly, encode it in data. The receiver must be able to detect when the data is valid, i.e., v(data) = true

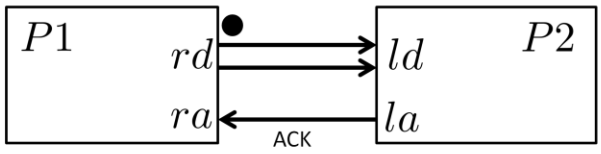
Delay-Insensitive Codes





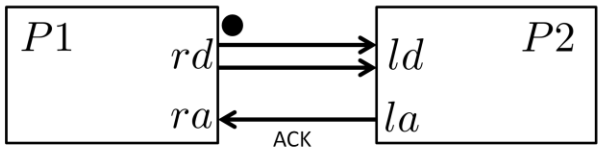
V(word), N(word)

instead of removing the request and data, encode this in data with a neutral predicate n(data).



$$P1 : rd := x; [ra]; rd := neutral; [\neg ra]$$
$$P2 : [v(ld)]; y := ld; la \uparrow; [n(ld)]; la \downarrow$$

-> transformed code with valid and neutral

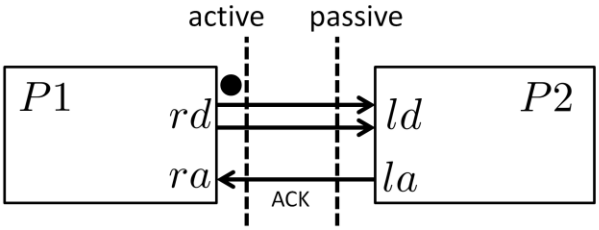


$$P1 : rd := x; [ra]; rd := neutral; [\neg ra]$$
$$P2 : [v(ld)]; y := ld; la \uparrow; [n(ld)]; la \downarrow$$

$$P1 : rd := x; [v(ra)]; rd := neutral; [n(ra)]$$
$$P2 : [v(ld)]; y := ld; la \uparrow; [n(ld)]; la \downarrow$$

transformation not only for data but also for ack-line. However, here only for theoretical purposes, as we do not transfer data
with ack.

Active vs Passive

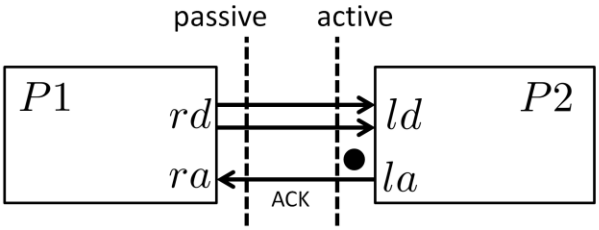


active $P1 : rd := x; [v(ra)]; rd := neutral; [n(ra)]$

passive $P2 : [v(ld)]; y := ld; la \uparrow; [n(ld)]; la \downarrow$

until now: writer active, reader passive (push based communication). However, can be vice versa, too. Important: always match active with passive interface.

Active vs Passive

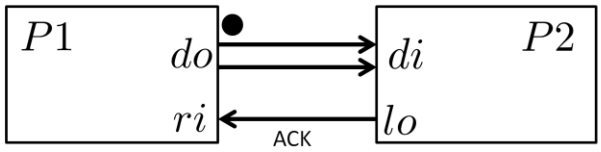


passive $P1 : [n(ra)]; rd := x; [v(ra)]; rd := neutral$

active $P2 : la \uparrow; [n(ld)]; la \downarrow; [v(ld)]; y := ld$

passive writer, active reader (pull based communication)

First code

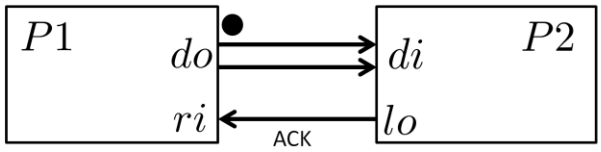


2-rails

neutral:	0 0	valid: is “0“ or “1“
0:	1 0	
1:	1 1	
transition:	0 1	

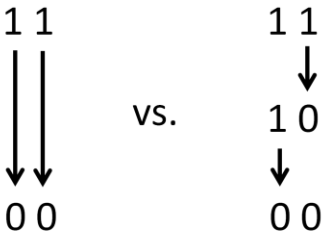
Our first code does not work:
transition from valid 1 to neutral over valid 0. The receiver could take this intermediate state as a valid 0

First code... hmm



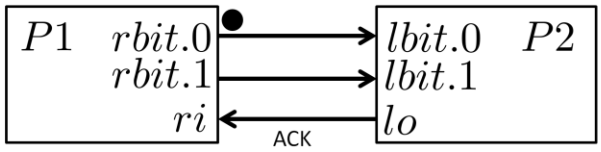
2-rails

neutral:	0 0
0:	1 0
1:	1 1
transition:	0 1



this is why it does not work

Dual-Rail



2-rails

neutral: 0 0

0: 1 0

1: 0 1

transition: 1 1

Now a working code: dual-rail encoding

Dual-Rail (n bit)

bit0.0 bit0.1 bit1.0 bit1.1 …

2n rails for n bit

generalization to n bit data

1-of-n / one-hot

n-rails for ld(n) bit

neutral:	0 0 0 0
0:	1 0 0 0
1:	0 1 0 0
2:	0 0 1 0
3:	0 0 0 1
transition:	else

Another code: 1-of-n code (also called on-hot code).
There are also more general k-of-n codes, which are however seldomly used.

Which one to choose...

… depends on

- efficient #wires
- fast & small n(), v() tests
- split & join

-> 1-of-4 splitting, dual-rail

the most common codes are 1-of-4 and dual-rail if the data is more than 1 bit. In case of 1 bit data, 1-of-2 and dual-rail are the same codes.

Important for codes are possibilities to split and join words of larger bitwidth. e.g. when building a decoding stage for a processor. Codes that do not allow easy split and joins are used seldom.

1-of-4 splitting: spit word in 2bit chunks and encode each one 1-of-4.

Beyond classical circuit design
lecture 5.5

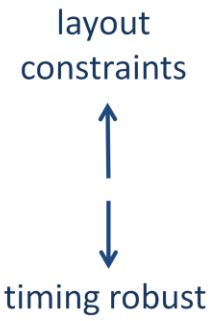
Circuit types

Further Reading

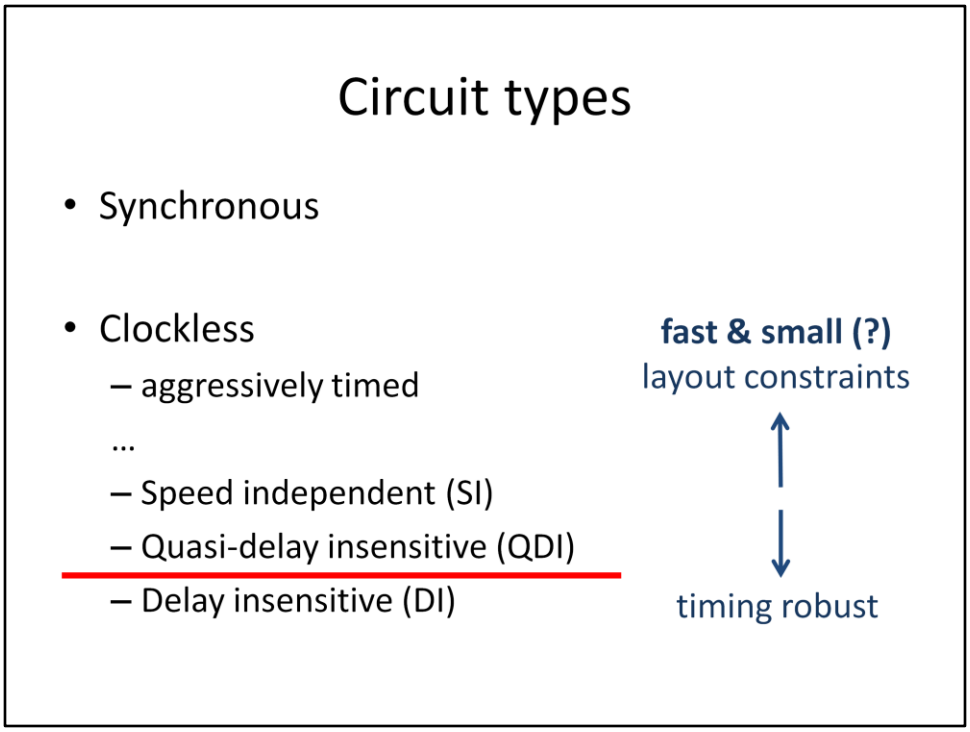
Keller, S.; Katelman, M.; Martin, A.J.: *A Necessary and Sufficient Timing Assumption for Speed-Independent Circuits.* Asynchronous Circuits and Systems (ASYNC'09). 15th IEEE Symposium on, pp. 65 - 76, 2009.

Circuit types

- Synchronous
- Clockless
  - aggressively timed
  - …
  - Speed independent (SI)
  - Quasi-delay insensitive (QDI)
  - Delay insensitive (DI)



trading layout (and thus timing) constraints versus timing robustness.

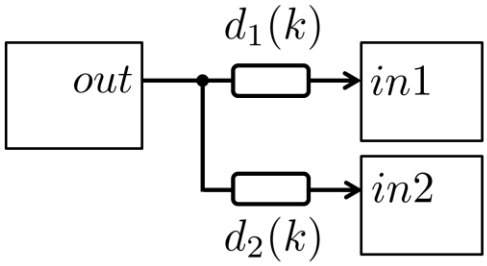


are aggressively timed circuits and synchronous circuits really fast & small?
Depends: Problem of conservative layouts as in synchronous design, see last homework

red border in expressiveness:
below the border DI circuits: only 2-input gate that exists there is the C-element -> highly resitricted class of circuits
that is not useful for many problems
above: circuits are Turing complete.

We will see what makes the difference.

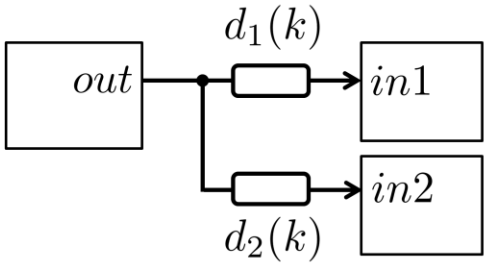
Isochronic fork



$$\forall k : d_1(k) \approx d_2(k)$$

from DI to QDI circuits: adding the isochronic fork assumption.
= the fundamental condition in clockless designs.

Isochronic fork

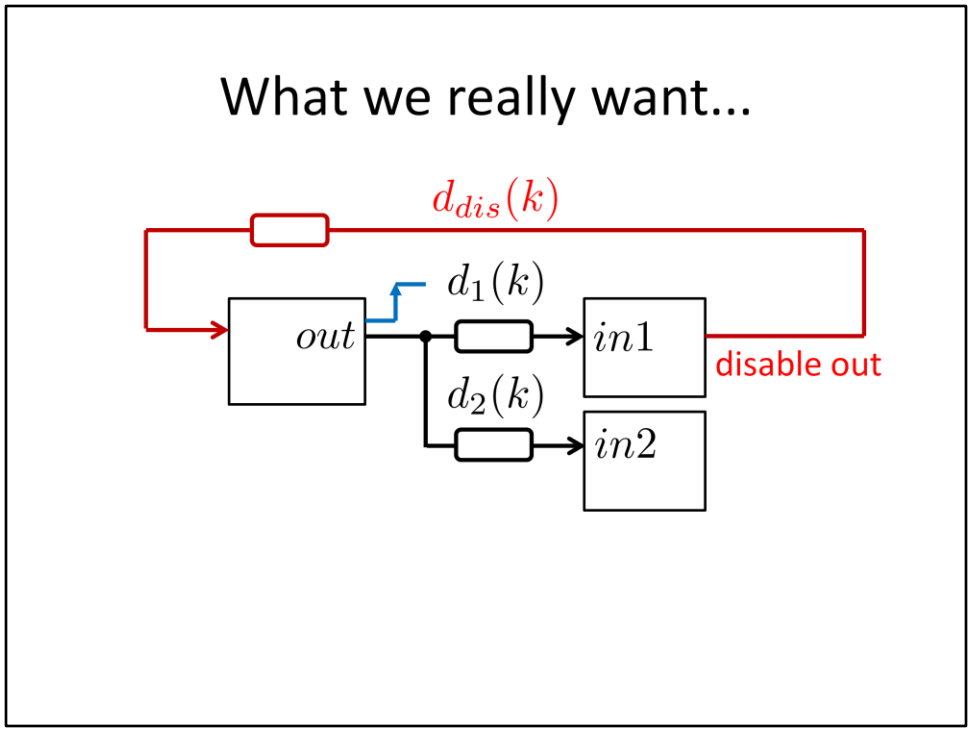


$\forall k : d_1(k) \approx d_2(k)$ ->

two-sided constraint [!]

ischronic fork :<->
all delays of the fork are (about) equal. [comment: Definition of „about equal“?! We will see later…]

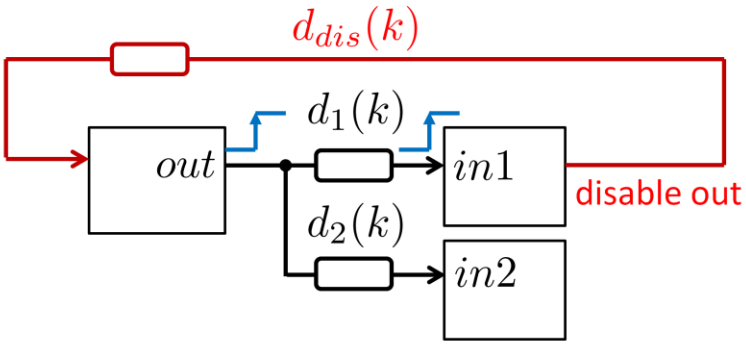
as such:
two sided-constraint that is difficult to reach (depending on what is meant by „about“)

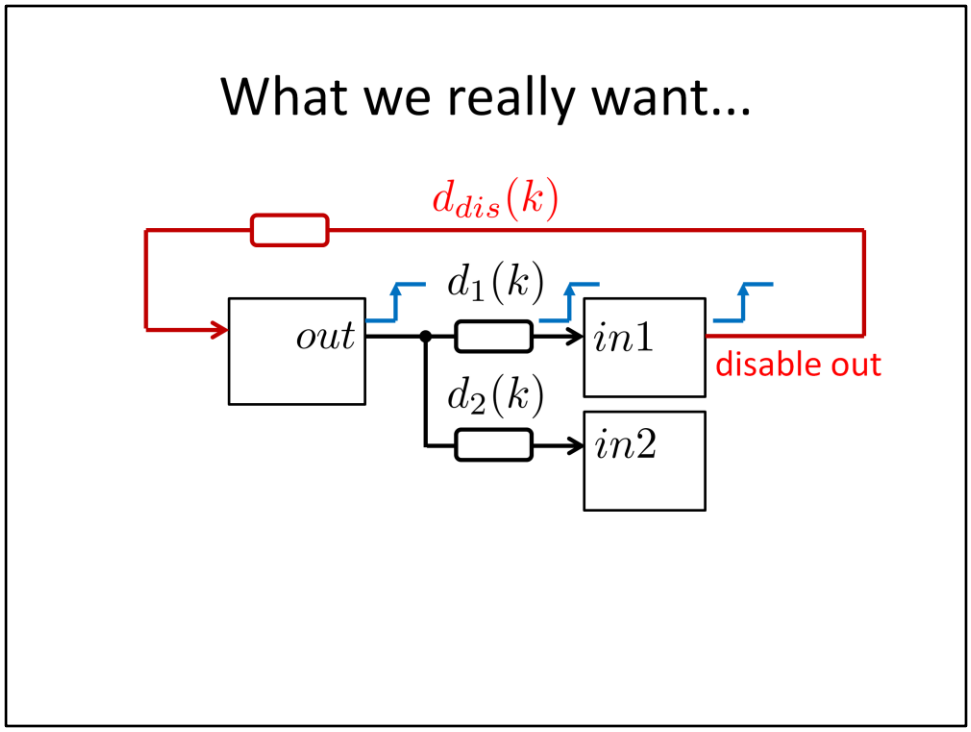


„about equal“ is a rule of thumb. What we really want is this:

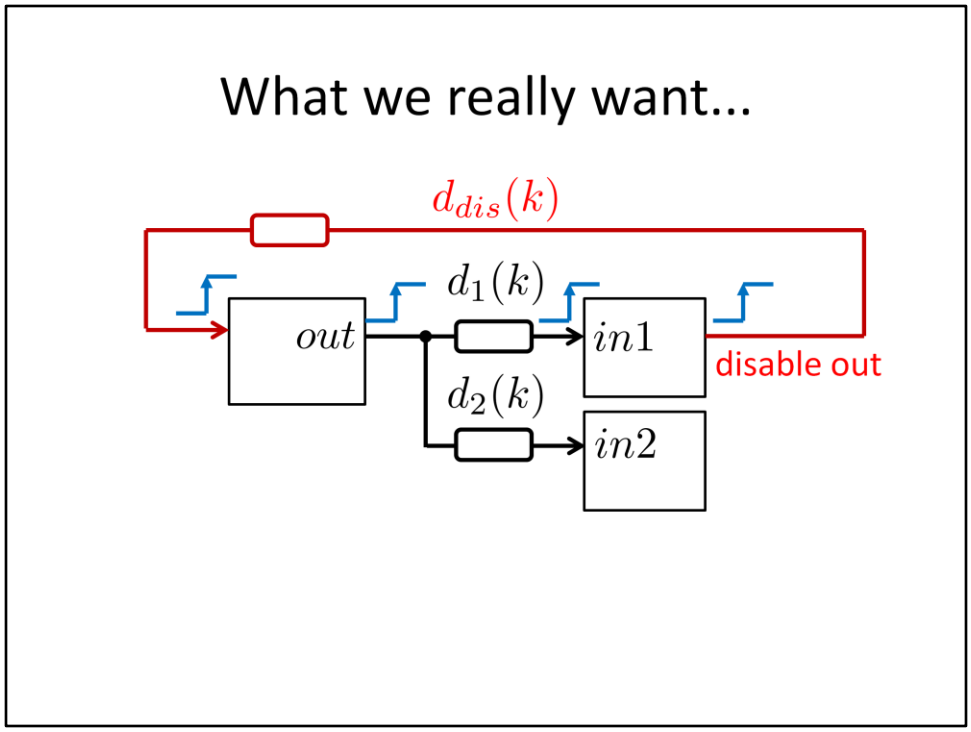
red: disabling path.

What we really want...

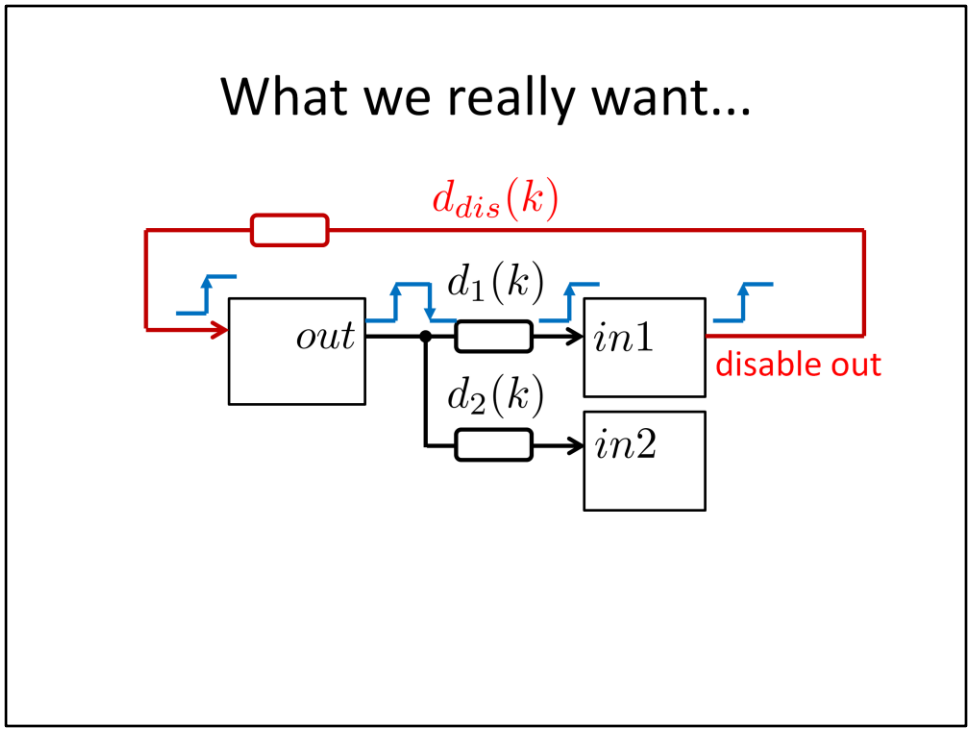




the fundamental condition in clockless designs.

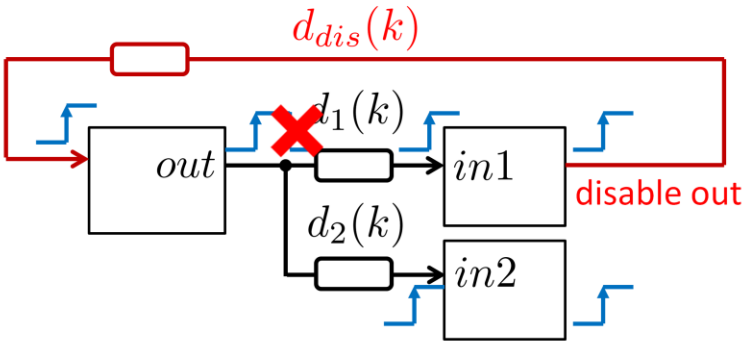


transition propagates along the disabling path



transition disables initial enabling

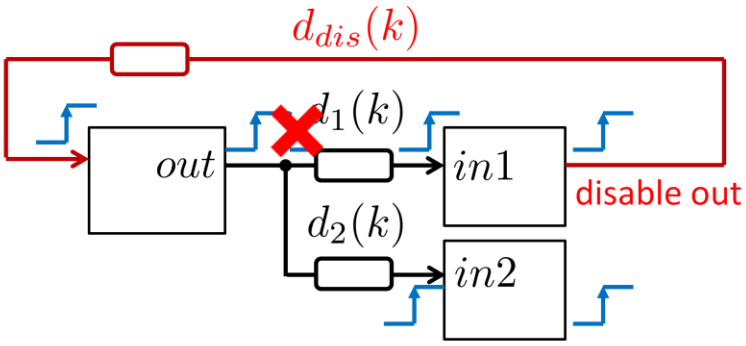
What we really want...



$$\forall k : d_2(k) \leq d_1(k) + d_{dis}(k)$$

the disabling should not happen before the other receiver did not receive the signal.

What we really want...



adversarial path condition

-> one-sided constraint

-> in fact it is a one-sided constraint that is far simpler to ensure.

Circuit types

- Synchronous
- Clockless
  - aggressively timed (beyond isochronic condition)
  - …
  - Speed independent (SI: all isochronic)
  - Quasi-delay insensitive (QDI: some isochronic)
  - Delay insensitive (DI)

isochronic fork as the fundamental condition:
DI: no isochronic fork assumptions
QDI: some forks are isochronic.

Examples

… soon from synthesis.

We will soon see isochronic forks in action.

Beyond classical circuit design
lecture 5.75

Synthesis

Synthesis

In: CHP

Out: Circuit (= Production rules + constraints)

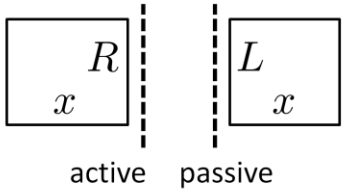
The snythesis problem

Example: 1-bit channel

Send $x \in \{0, 1\}$

active send,

passive receive



CHP:

sender: $Ps : *[\ldots R!x \ldots]$

receiver: $Pr : *[\ldots L?x \ldots]$

We want to synthesize a 1-bit communiction channel between a sender and receiver (push based).

Example: 1-bit channel

Choice: dual-rail encoding.

$$R!x \quad rd := x; [v(ra)]; rd := neutral; [n(ra)]$$

$$[x \rightarrow rd.1 \uparrow || \bar{x} \rightarrow rd.0 \uparrow]; [ra];$$
$$(rd.1 \downarrow || rd.0 \downarrow); [\bar{ra}]$$

Start with the abstract channel CHP as we learned it.
Our first choice, the encoding. -> modified CHP.
insert valid and neutral predicates for the chosen code.

Example: 1-bit channel

$L?x \quad [v(ld)]; y := ld; la\uparrow; [n(ld)]; la\downarrow$

$[ld.1 \vee ld.0]; [ld.1 \rightarrow x\uparrow \,||\, ld.0 \rightarrow x\downarrow]; la\uparrow;$
$[\neg(ld.1 \vee ld.0)]; la\downarrow$

$[ld.1 \rightarrow x\uparrow \,||\, ld.0 \rightarrow x\downarrow]; la\uparrow;$
$[\neg(ld.1 \vee ld.0)]; la\downarrow$

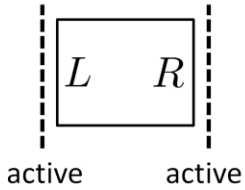
2nd CHP: the first predicate can be removed. It is redundant because the select is blocking anyway.

We finally obtained low-level CHP for sender and receiver. We will see at another example of how to fully get to PRs from CHP.

Example: active-active sequencer

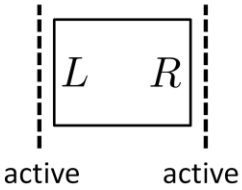
two active interfaces L, R.

CHP: $P1 : *[L; R]$



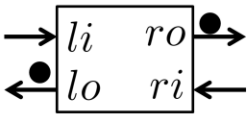
Example: active-active sequencer

two active interfaces L, R.



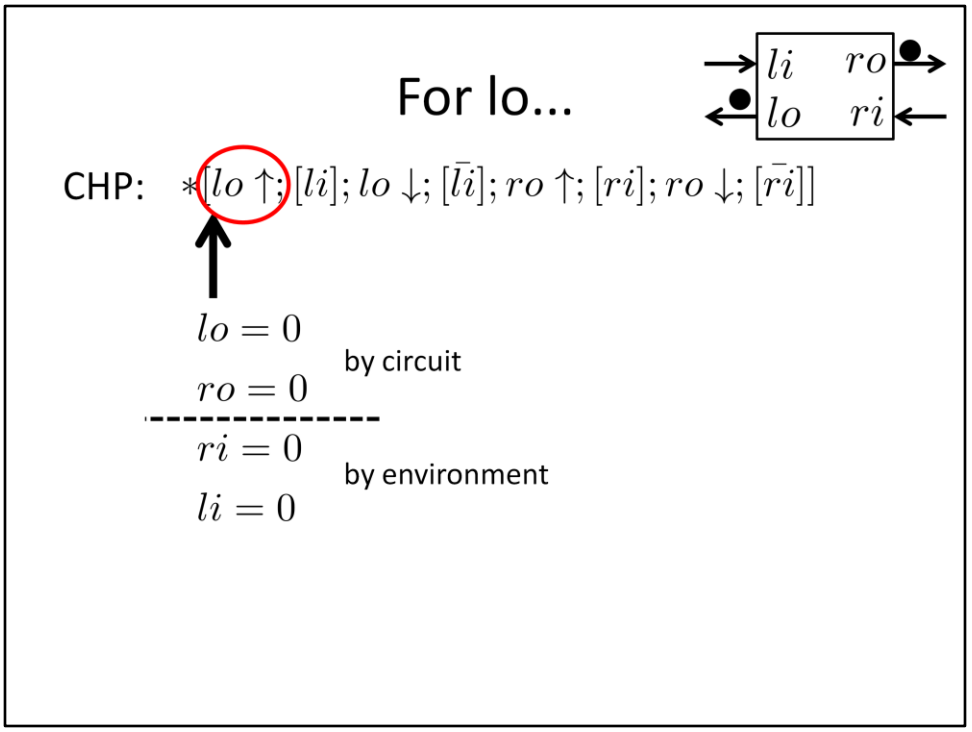
CHP: $P1 : *[L; R]$

Choice: 4-ph hs

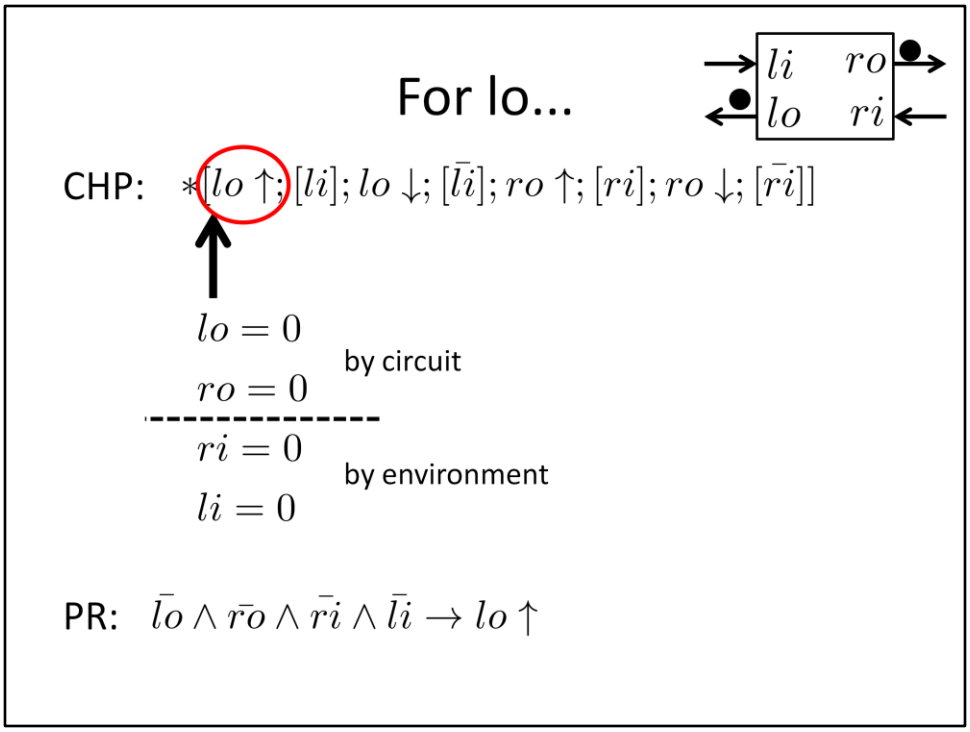


$$*[lo\uparrow; [li]; lo\downarrow; [\bar{li}]; ro\uparrow; [ri]; ro\downarrow; [\bar{ri}]]$$

Here: choice was for 4-phase handshaking as a sequencing protocol.

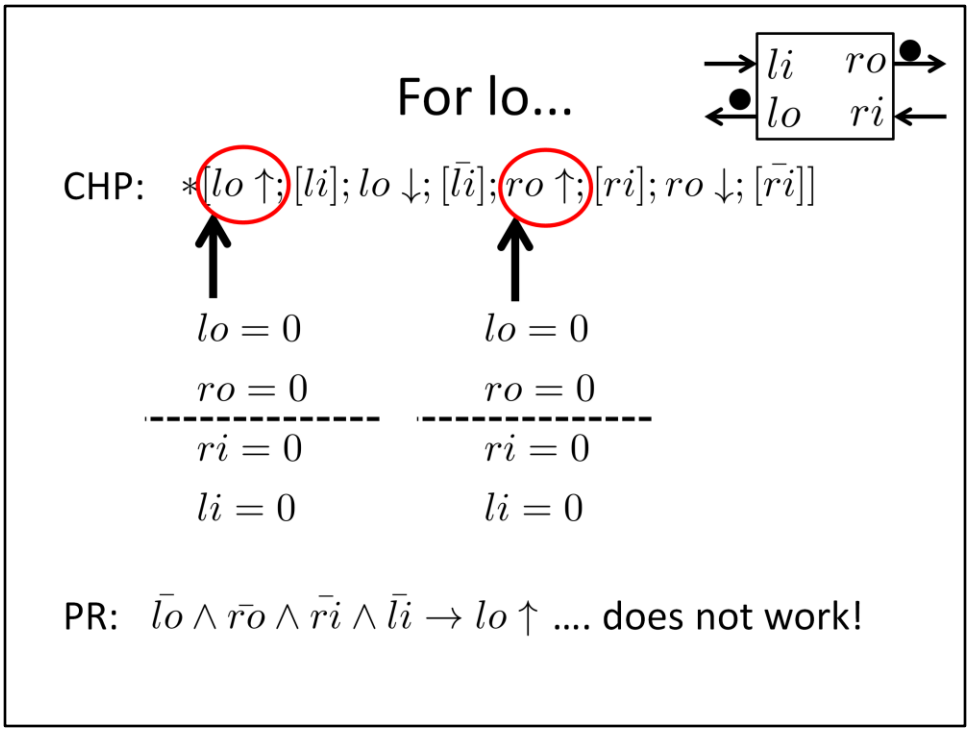


Synthesis process:
start with first command and note the variable states for it. Variables are both set from the circuit and from the environment.

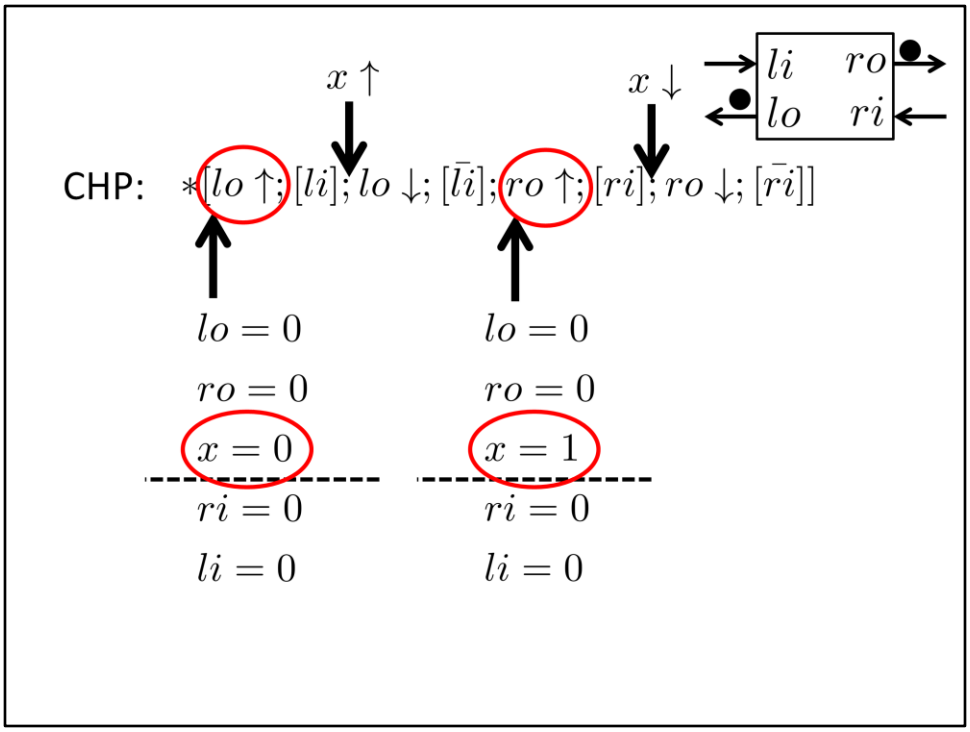


by circuit section: variable states derived from the CHP of the circuit alone
by environment section: variable states derived from the environmental behavior + circuit. e.g. [not ri] at end of CHP. Before executing lo-up still ri=0 since environment does not rise lo without a lo-up transition.

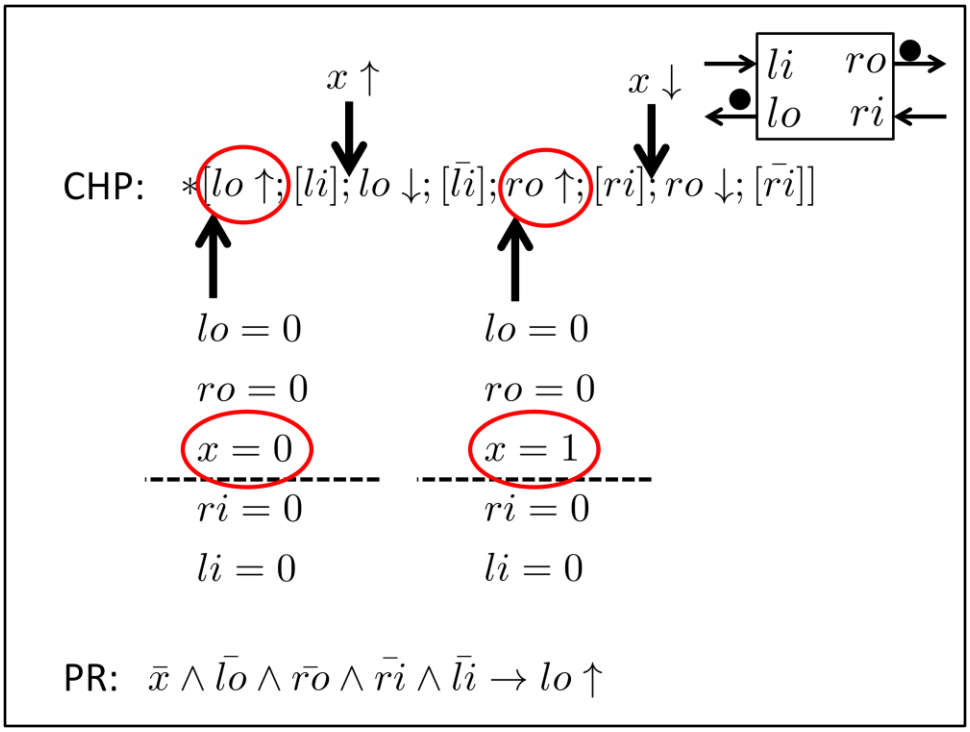
write a production rule from the state guarantees we get for the action. Here: predicate depending on *all* variables.



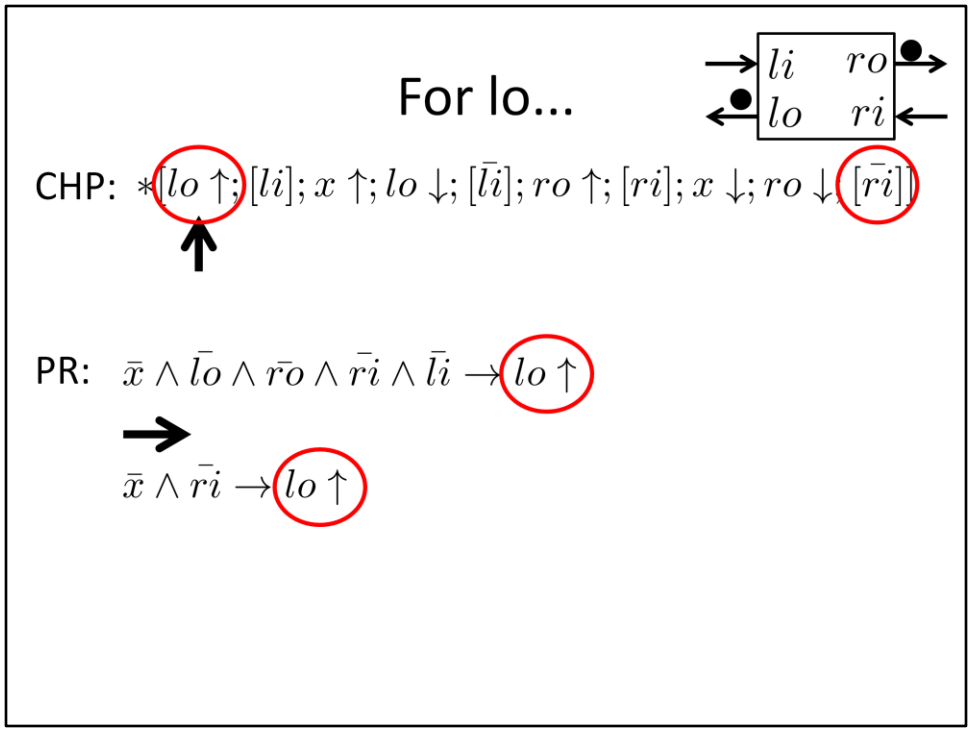
Problem: the action for ro-up needs the same predicate as lo-up.
But [!]: lo-up and ro-up should be executed at different times according to the CHP.
This cannot be expressed by the variables in the CHP!
-> we need to add helper variables.



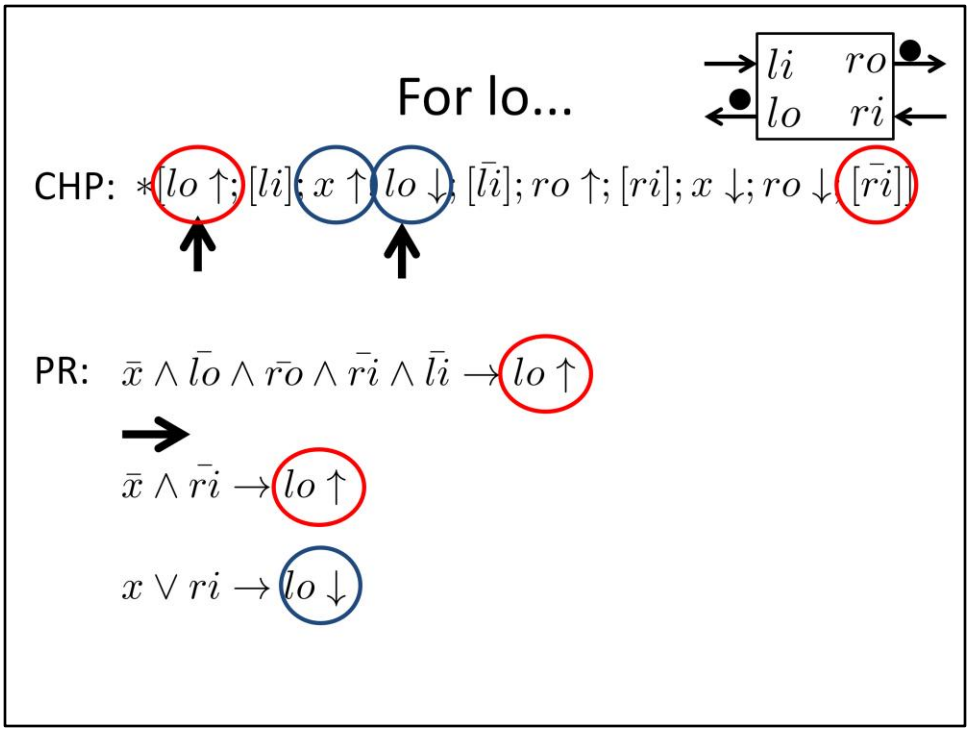
adding the helper variable x.



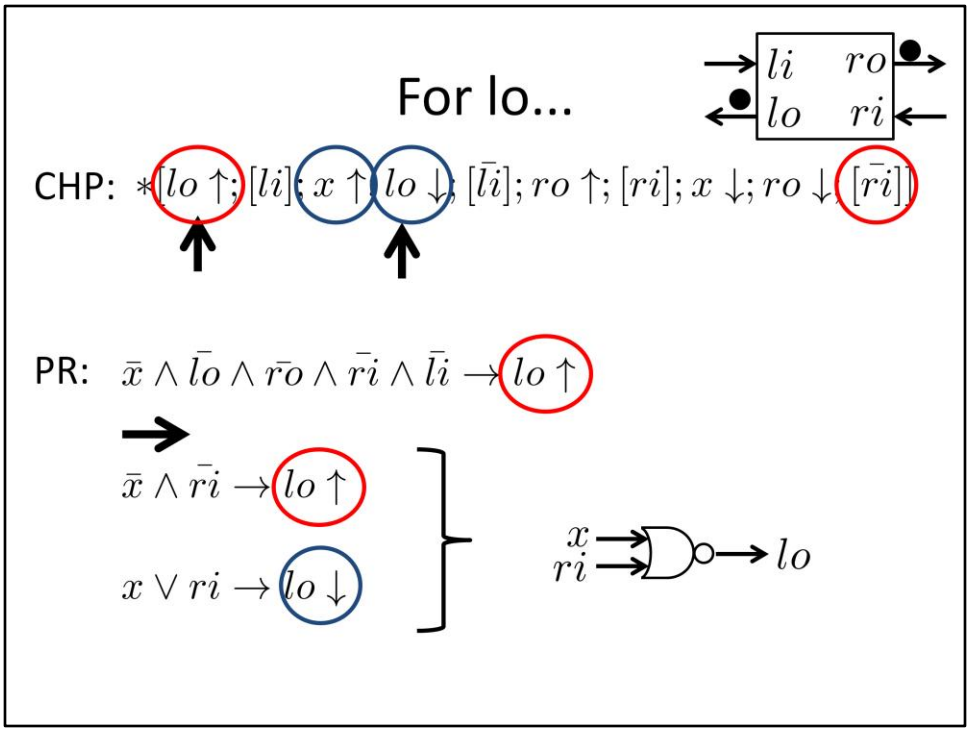
New PR with the helper variable.



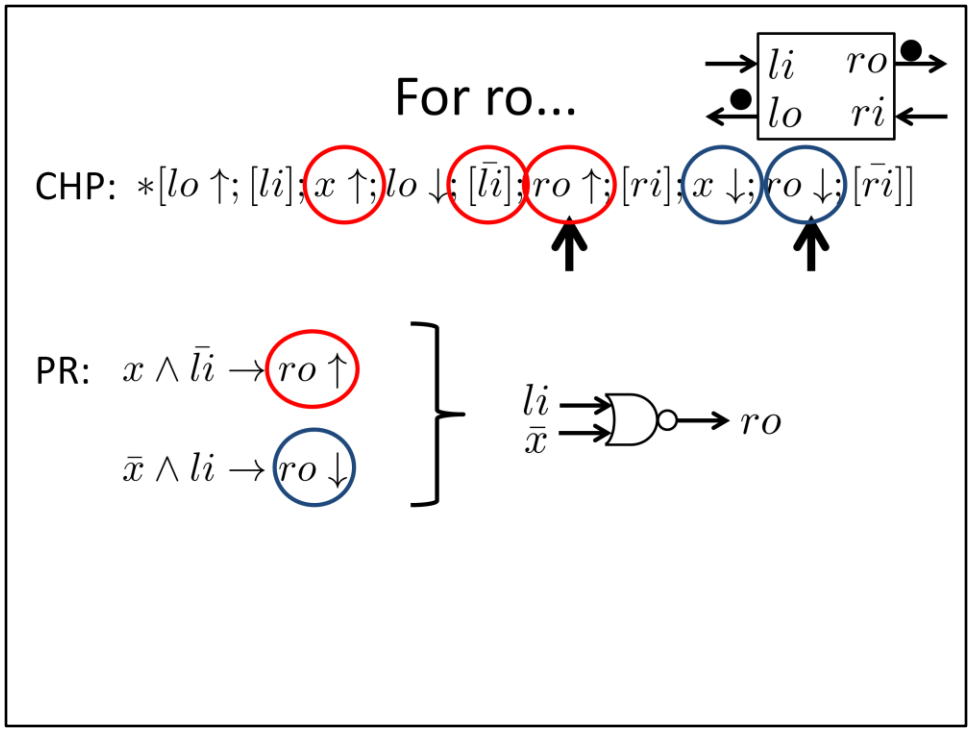
opimizing. minimum guard sufficient to decide when to trigger lo up. neg ri & helper variable are sufficient.



now write the guard when it should go down.

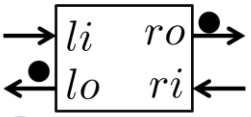


implementation as combinational gate is favorable since small and fast.
-> try to make gates combinational instead of state holding if possible.
here: 2OR.



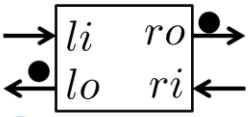
now for the ro signal.

For x...



CHP: $*[lo\uparrow;[li];x\uparrow;lo\downarrow;[\bar{li}];ro\uparrow;[ri];x\downarrow;ro\downarrow;[\bar{ri}]]$

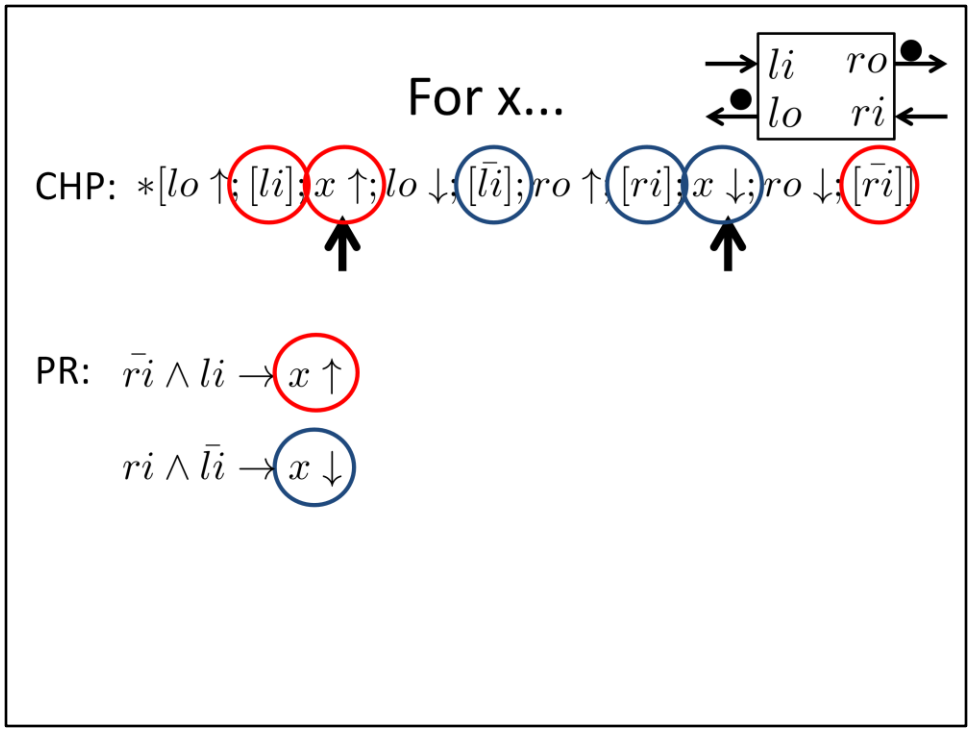
For x...



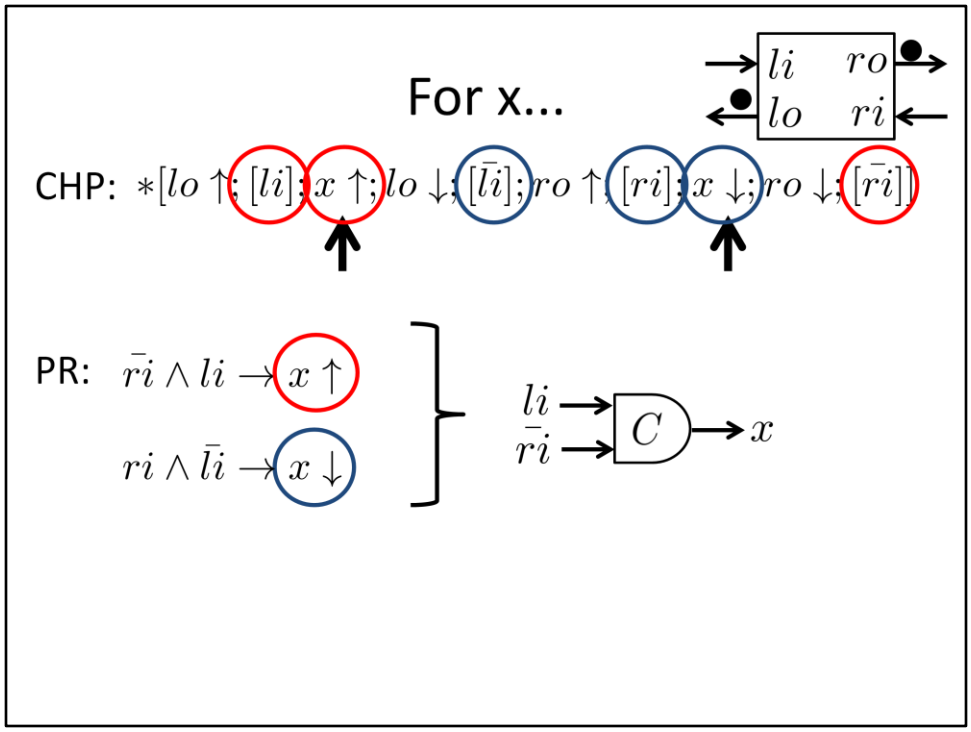
CHP: $*[lo\uparrow;[li];x\uparrow;lo\downarrow;[\bar{li}];ro\uparrow;[ri];x\downarrow;ro\downarrow;[\bar{ri}]]$

… What gate is this?

this time it doesn't look like we can do with a combinational gate.

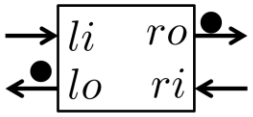


Writing it this way seems more favorable.



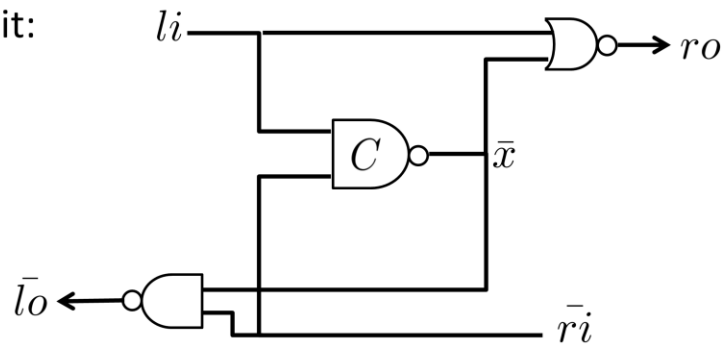
-> not combinational but a C-Element.

Putting it together



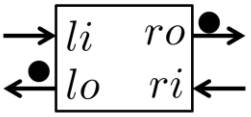
CHP: $*[lo\uparrow; [li]; x\uparrow; lo\downarrow; [\bar{li}]; ro\uparrow; [ri]; x\downarrow; ro\downarrow; [\bar{ri}]]$

Circuit:



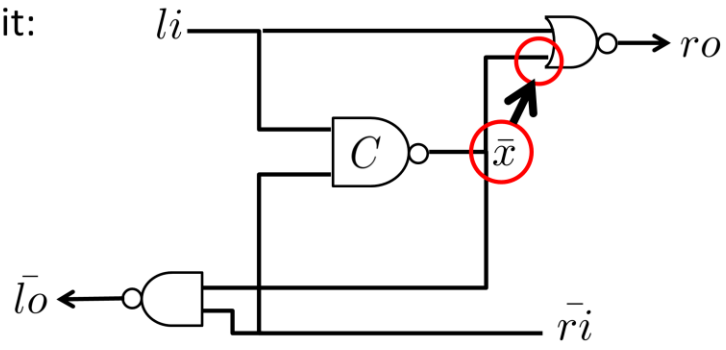
This is our result from synthesis. But can we simply put it together?

Mind wires...



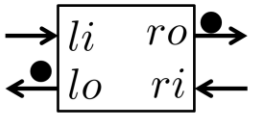
CHP: $*[lo\uparrow; [li]; x\uparrow; lo\downarrow; [\bar{li}]; ro\uparrow; [ri]; x\downarrow; ro\downarrow; [\bar{ri}]]$

Circuit:



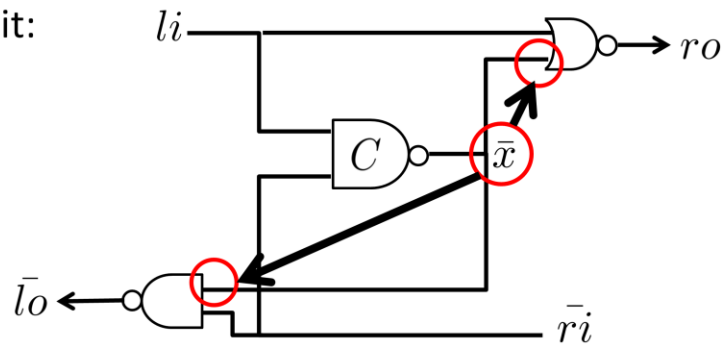
we accounted for delays with our PRs

Mind forks...



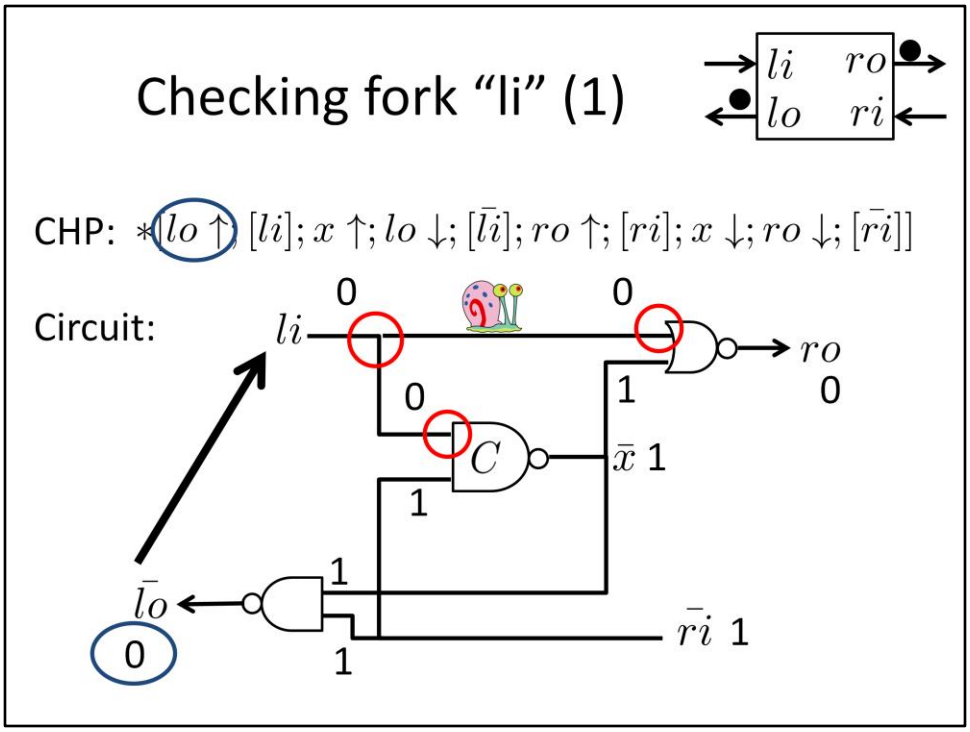
CHP: $*[lo\uparrow; [li]; x\uparrow; lo\downarrow; [\bar{li}]; ro\uparrow; [ri]; x\downarrow; ro\downarrow; [\bar{ri}]]$

Circuit:

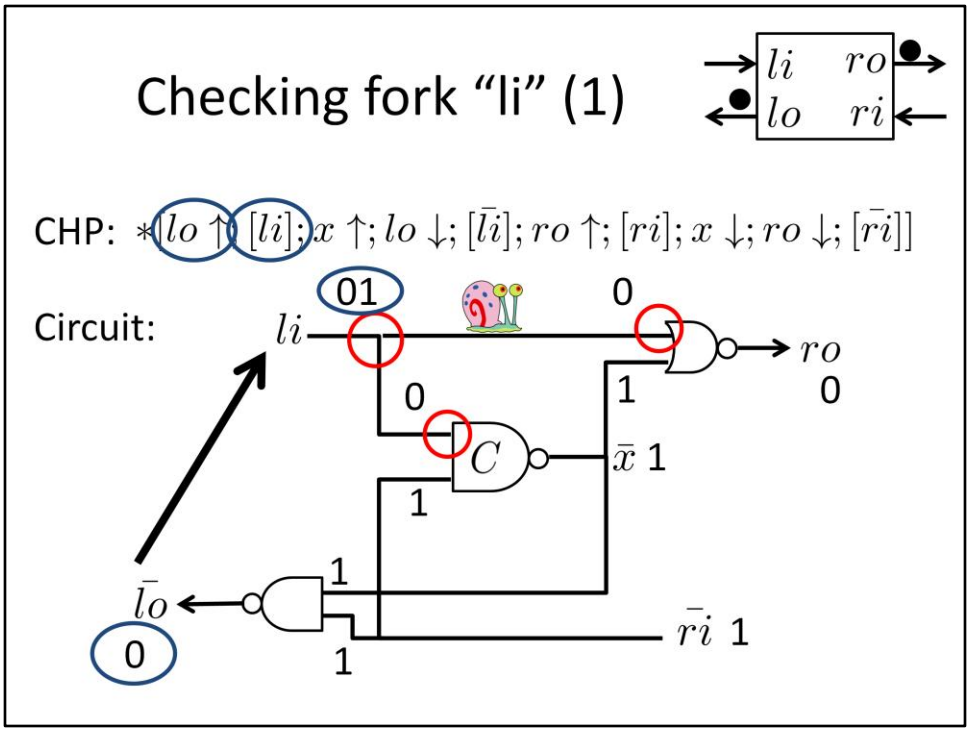


But: we assumed during synthesis that the views are the same. once x is updated, it is viewed the same at all gate inputs.

That is: we assumed *all* forks to have exactly the same delay. To be isochronic.
Do we really need all the be isochronic. Lets check…

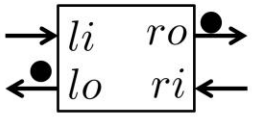


assume slow upper li teeth



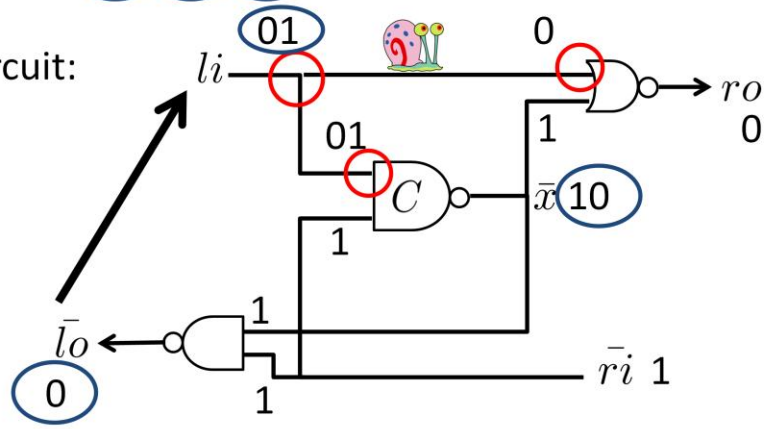
An example execution.

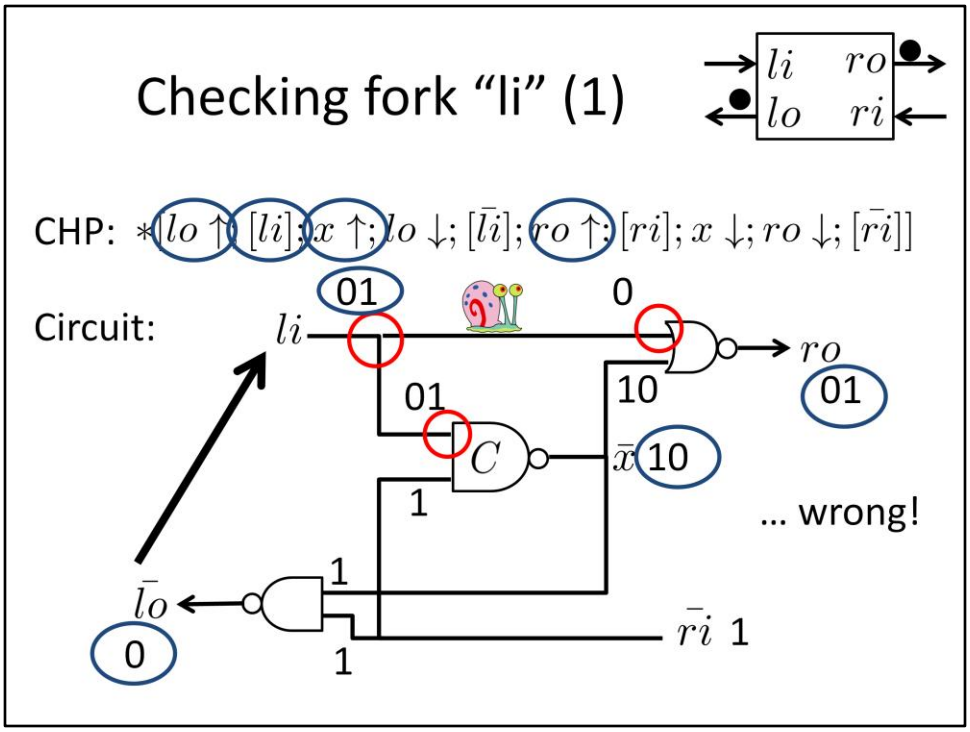
Checking fork “li” (1)



CHP: $*[lo\uparrow; [li]; x\uparrow; lo\downarrow; [\bar{li}]; ro\uparrow; [ri]; x\downarrow; ro\downarrow; [\bar{ri}]]$

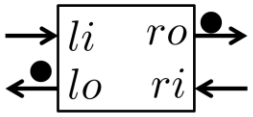
Circuit:





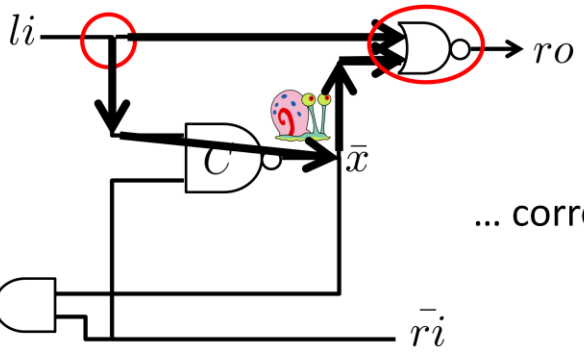
violation to our CHP specification.

+ one-sided Constraint



CHP: $*[lo\uparrow; [li]; x\uparrow; lo\downarrow; [\bar{li}]; ro\uparrow; [ri]; x\downarrow; ro\downarrow; [\bar{ri}]]$

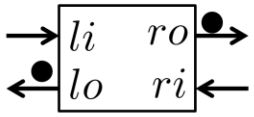
Circuit:



... correct.

prevent this from happening by one-sided delay constraint

+ one-sided Constraint



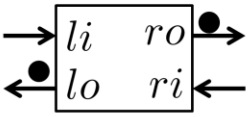
CHP: $*[lo\uparrow; [li]; x\uparrow; lo\downarrow; [\bar{li}]; ro\uparrow; [ri]; x\downarrow; ro\downarrow; [\bar{ri}]]$

Proving correct:

e.g. by induction.

base case: start with initial states and prove ordering of events from there for the first loop

+ one-sided Constraint



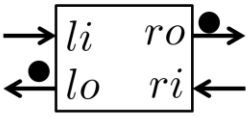
CHP: $*[lo\uparrow; [li]; x\uparrow; lo\downarrow; [\bar{li}]; ro\uparrow; [ri]; x\downarrow; ro\downarrow; [\bar{ri}]]$

proof:

examine ordering in time "<".

1) **lo-up < [li]:** guaranteed by environment: Initially li = 0. Can be set to li = 1 only by environment. Environment does this only after lo = 1.

+ one-sided Constraint

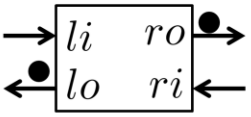


CHP: $*[lo\uparrow; [li]; x\uparrow; lo\downarrow; [\bar{li}]; ro\uparrow; [ri]; x\downarrow; ro\downarrow; [\bar{ri}]]$

proof:

2) **[li] < x-up:** x = 1 can happen only after both C-Element inputs are 1. This can happen only after li = 1 for the first time.

+ one-sided Constraint



CHP: $*[lo\uparrow; [li]; x\uparrow; lo\downarrow; [\bar{li}]; ro\uparrow; [ri]; x\downarrow; ro\downarrow; [\bar{ri}]]$

proof:

3) **x-up < lo-down:** this can happen only if one of the NAND inputs becomes 0 for the first time.

We first show that lo-down cannot happen because of the input connected to not ri becoming 0: .....

[hw]