



# **Asynchronous Design: Introduction to Principles and Models**

Alex Yakovlev

Newcastle University

Newcastle upon Tyne, U.K.

# Outline

- **Six Asynchronous Design Principles:**
  - **Asynchronous Handshaking**
  - **Delay-insensitive Encoding**
  - **Completion Detection**
  - **Causal Acknowledgement**
  - **Full Indication and Early Evaluation**
  - **Time Comparison**
- **Pros and Cons**
- **(Some of the) Models, Techniques and Tools for Asynchronous Design**
- **Asynchronous control logic synthesis from Signal Transition Graphs**

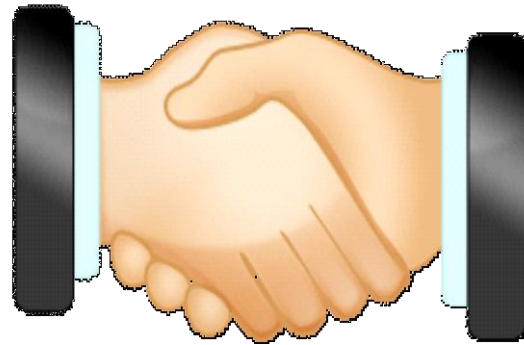
# Asynchronous Behaviour

- **Synchronous vs Asynchronous behaviour in general terms, examples:**
  - **Orchestra playing with vs without a conductor**
  - **Party of people having a set menu vs a la carte**
- **Synchronous means all parts of the system acting globally in tact, even if some or all part 'do nothing'**
- **Asynchronous means parts of the system act on demand rather than on global clock tick**
- **Acting in computation and communication is, generally, changing the system state**
- **Synchrony and Asynchrony can be in found in CPUs, Memory, Communications, SoCs, NoCs etc.**

# Key Principles of Asynchronous Design

- **Asynchronous handshaking**
- **Delay-insensitive encoding**
- **Completion detection**
- **Causal acknowledgment (aka indication or indicatability)**
- **Strong and weak causality (full indication and early evaluation)**
- **“Time comparison” (synchronisation, arbitration)**

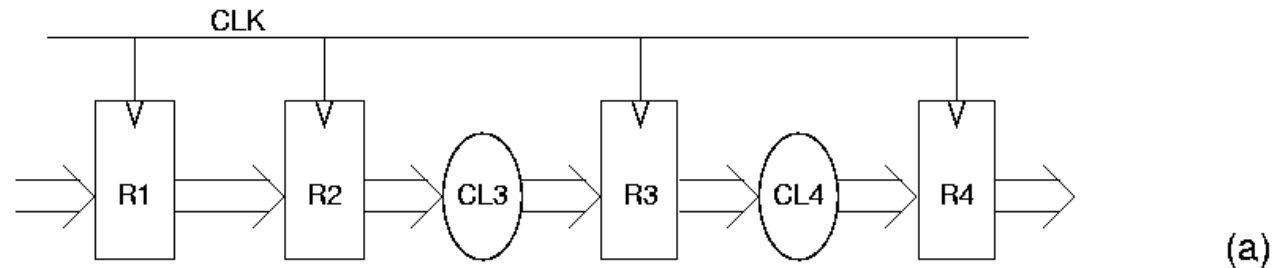
# Why and what is handshaking?



**Mutual Synchronisation is via Handshake**

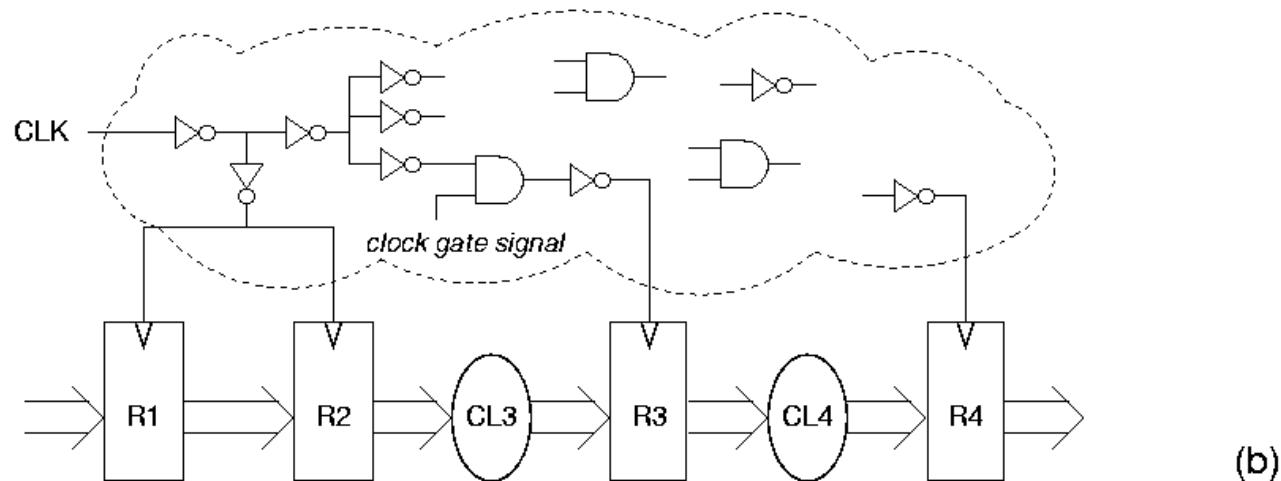
# Synchronous clocking

**How we think**



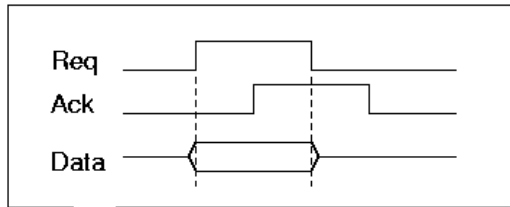
(a)

**What we design**

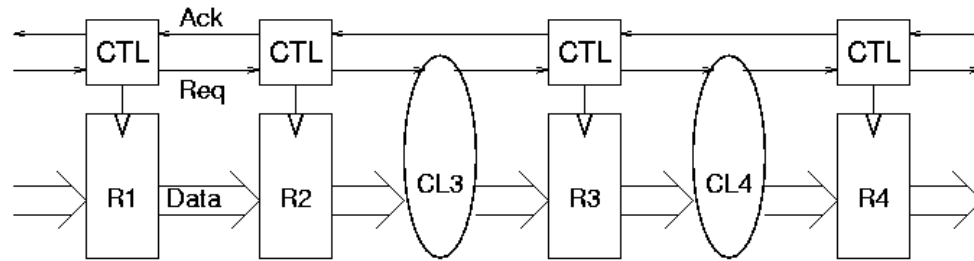


(b)

# Asynchronous handshaking

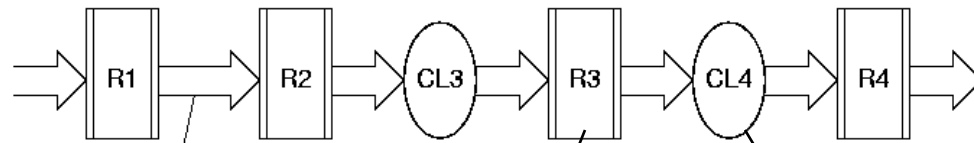


**What we design**



(c)

**How we think**



"Channel" or "Link"

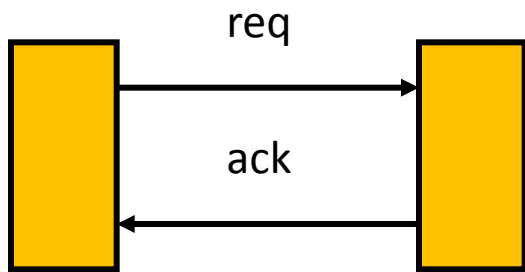
Handshake latch

Handshake CL

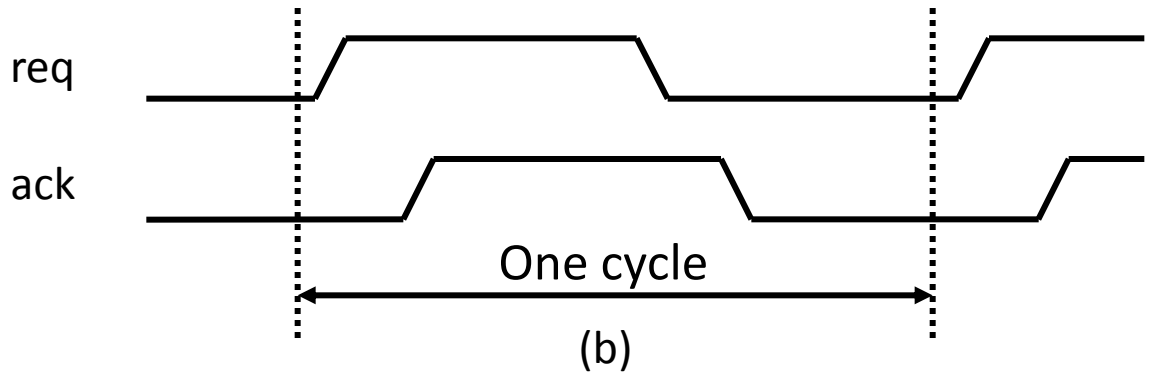
(d)

# Handshake Signalling Protocols

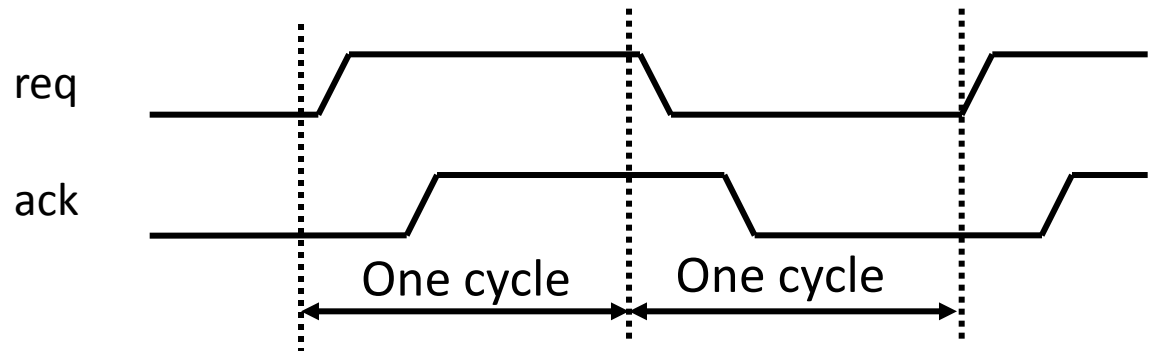
## Level Signalling (RTZ or 4-phase)



(a)

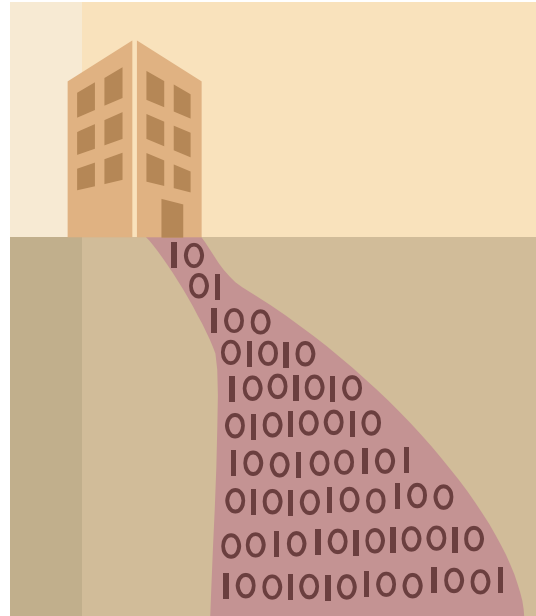


## Transition Signalling (NRZ or 2-phase)



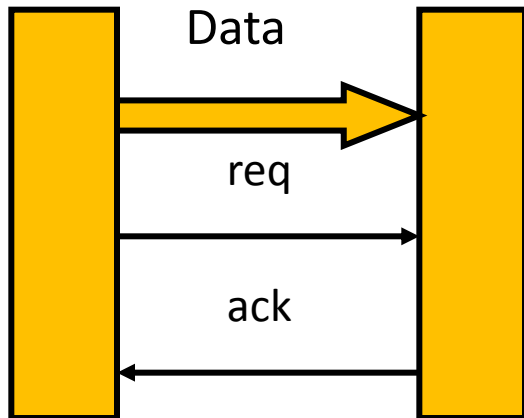


# Why and what is delay-insensitive coding?

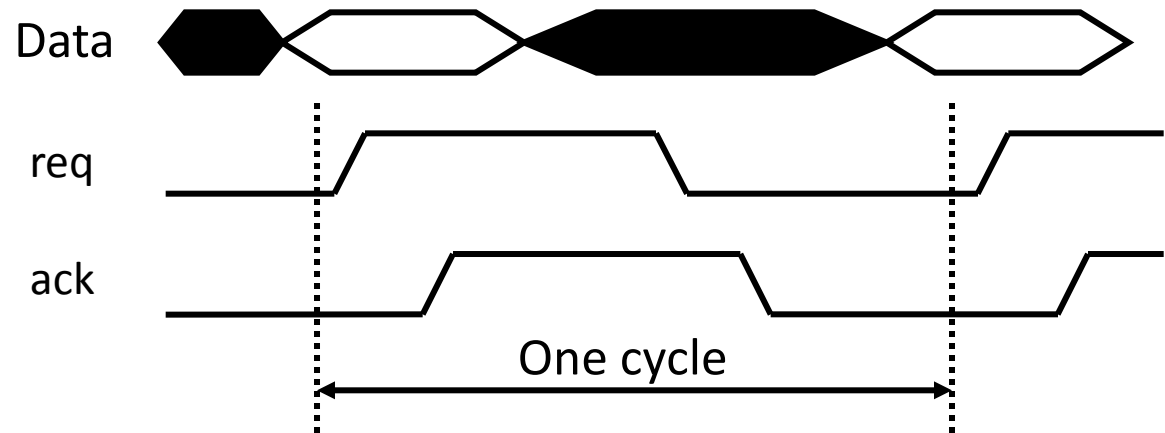


**Data Token = (Data Value, Validity Flag)**

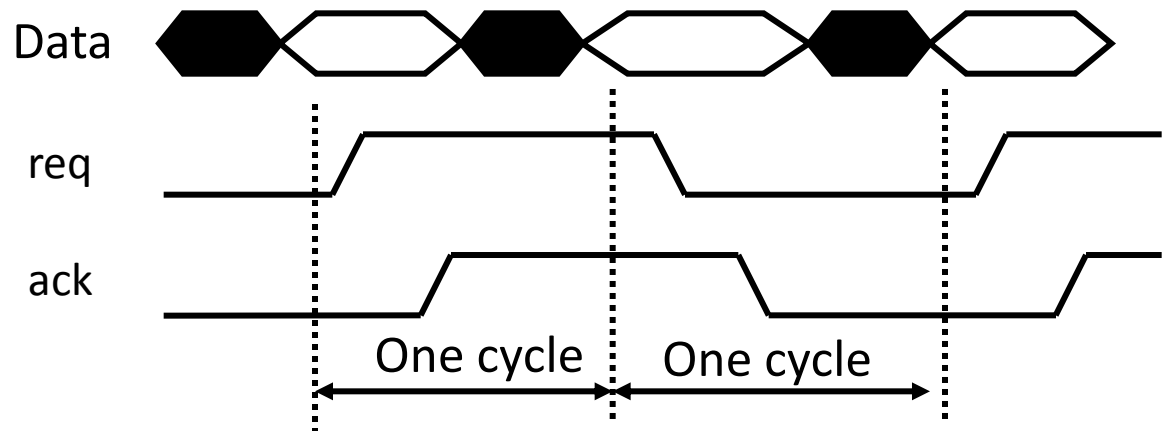
# Bundled Data



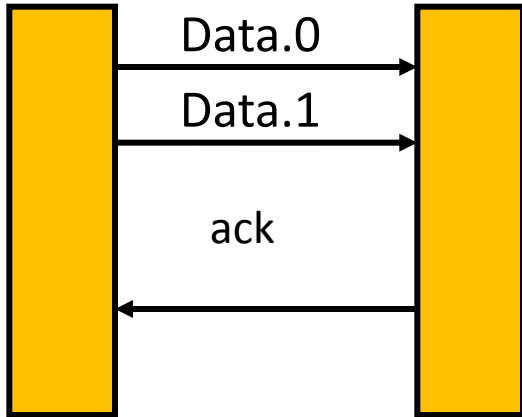
Return to Zero:



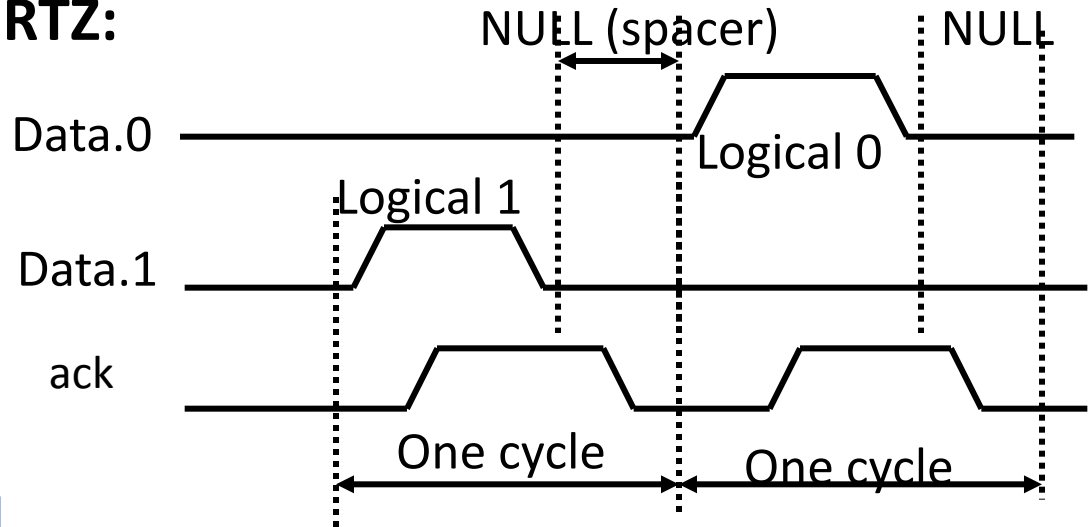
**Non-Return-to-Zero**



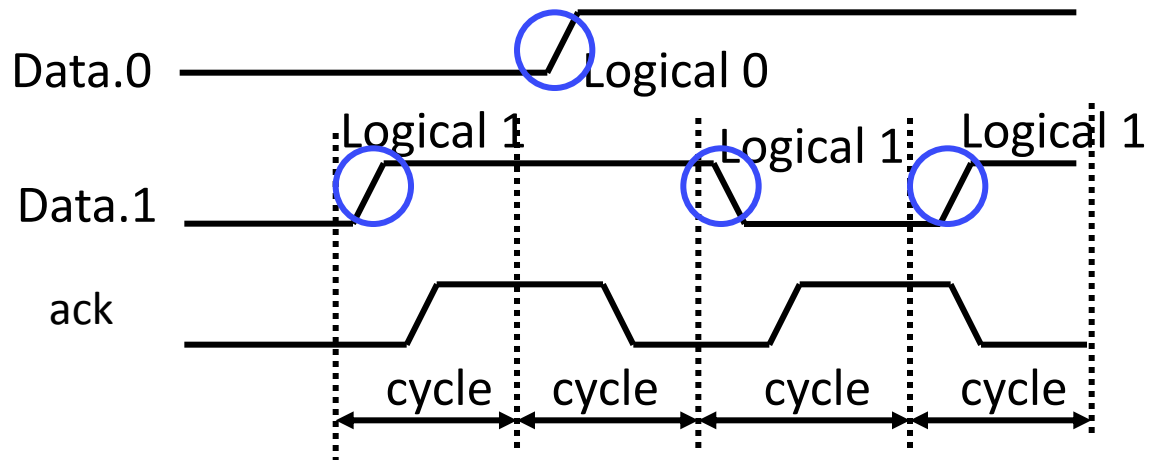
# DI encoded data (Dual-Rail)



**RTZ:**



**NRZ:**

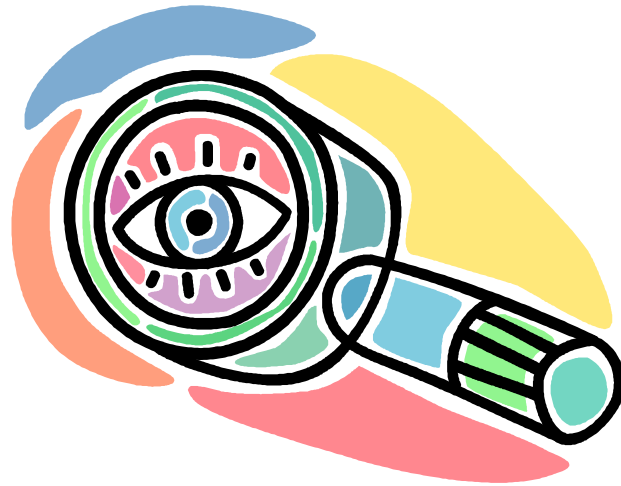


NRZ coding leads to complex logic implementation; special ways to track odd and even phases and logic values are needed, such as LEDR

# DI codes (1-of-n and m-of-n)

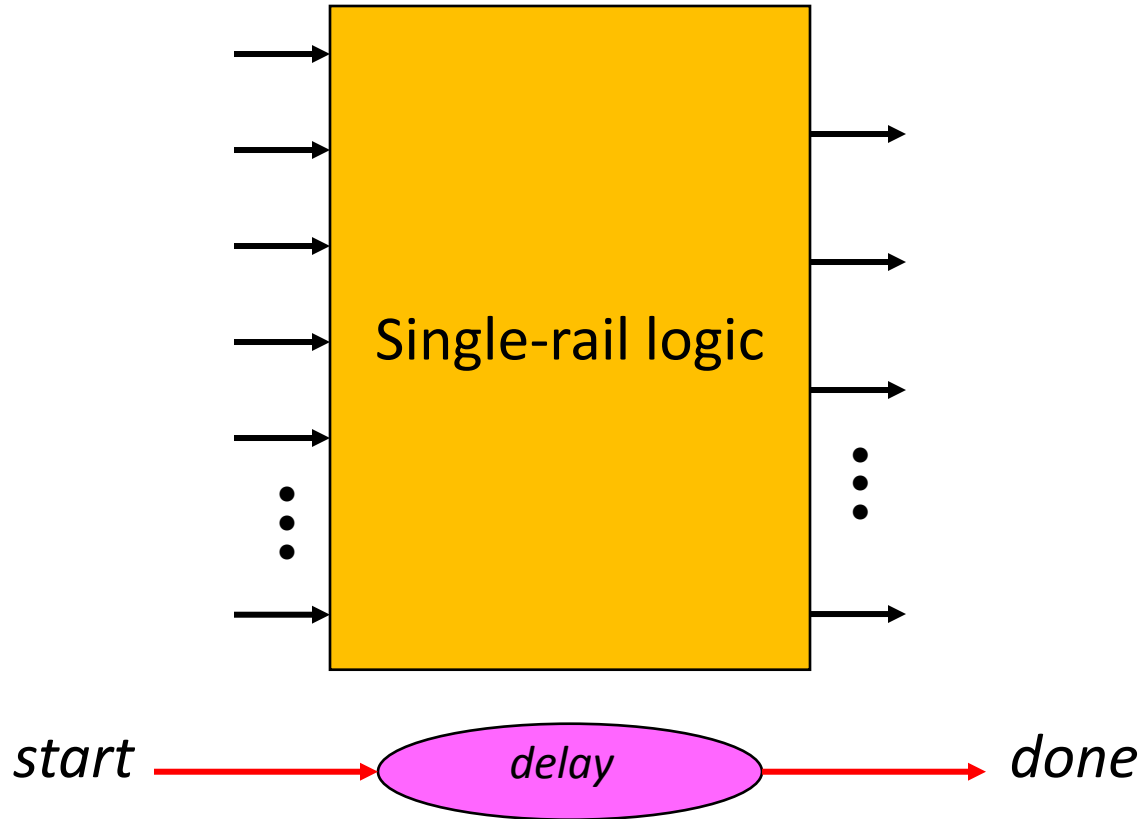
- **1-of-4:**
  - **0001=> 00, 0010=>01, 0100=>10, 1000=>11**
- **2-of-4:**
  - **1100, 1010, 1001, 0110, 0101, 0011 – total 6 combinations (cf. 2-bit dual-rail – 4 comb.)**
- **3-of-6:**
  - **111000, 110100, ..., 000111 – total 20 combinations (can encode 4 bits + 4 control tokens)**
- **2-of-7:**
  - **1100000, 1010000, ..., 0000011 – total 21 combinations (4 bits + 5 control tokens)**

# Why and what is completion detection?



**Signalling that the Transients are over**

# Bundled-data logic blocks

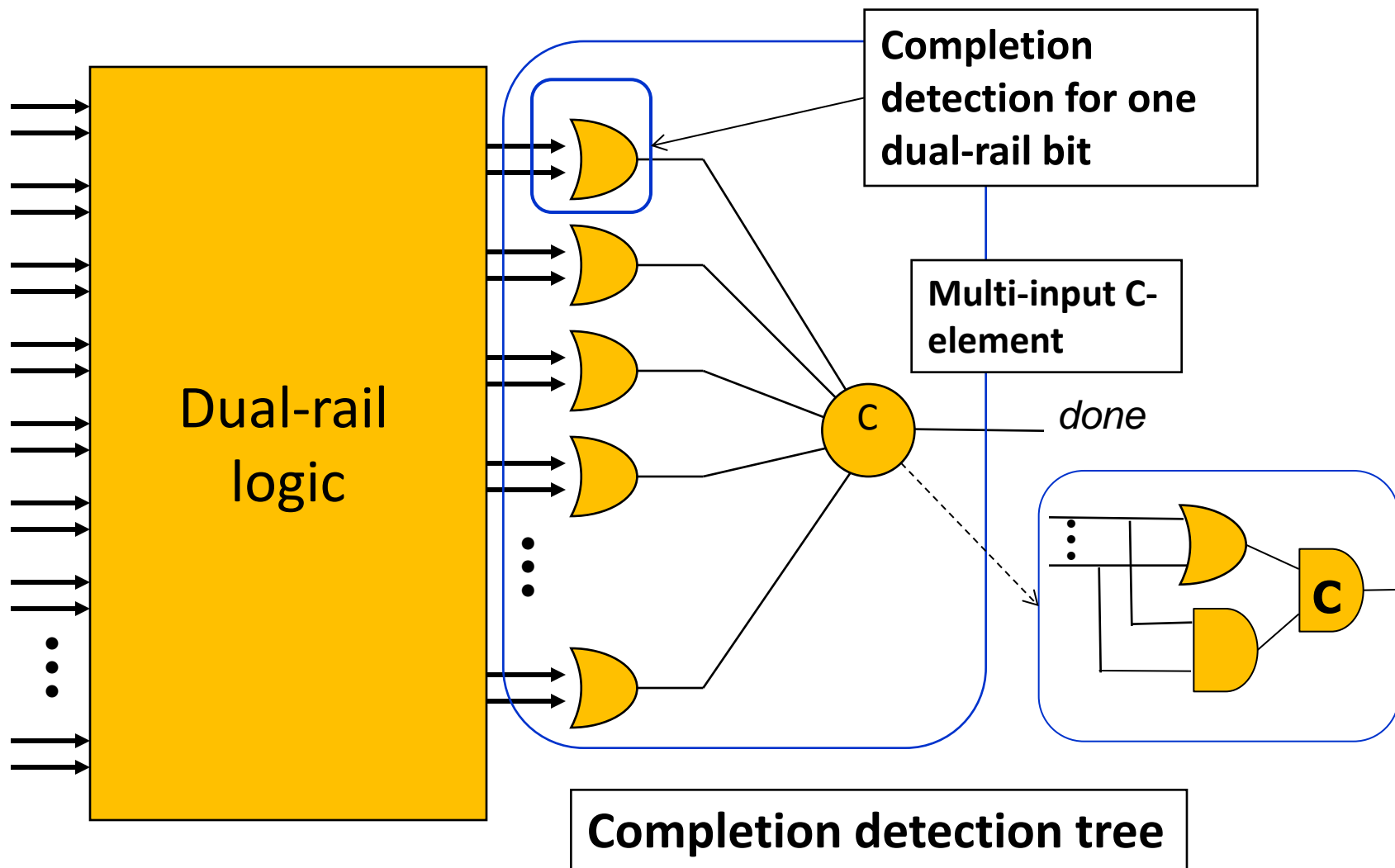


Completion  
is implicit:  
by done  
signal

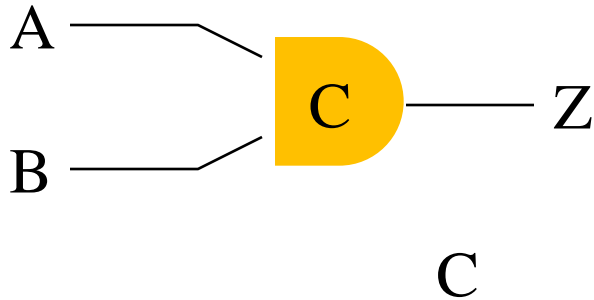
The delay must  
scale with the worst  
case delay path,  
So ... not really self-  
timed

**Conventional logic + matched delay**

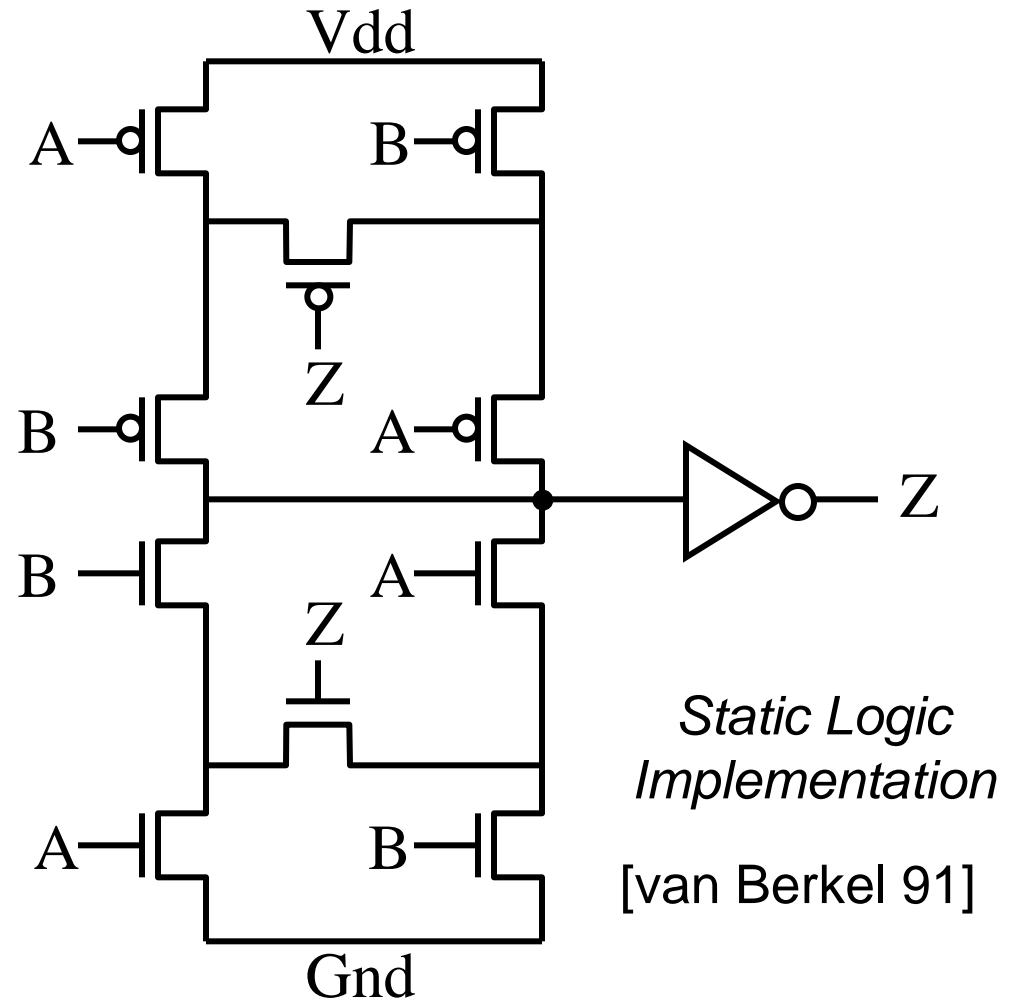
# True completion detection



# The Muller C element

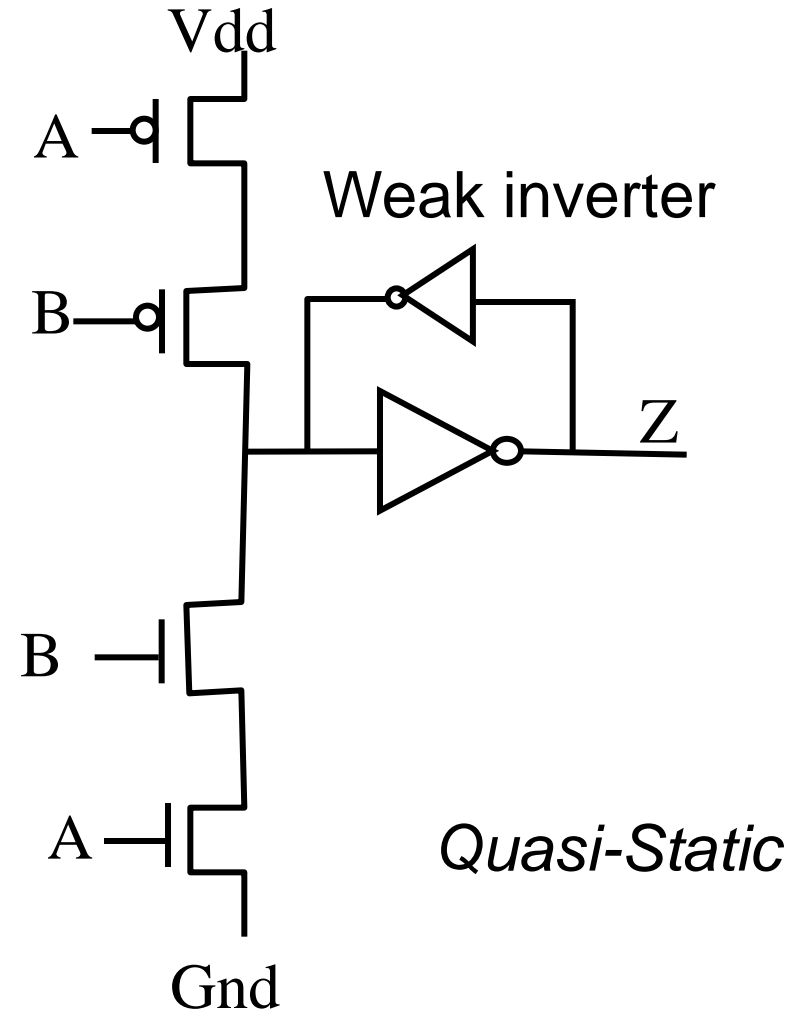
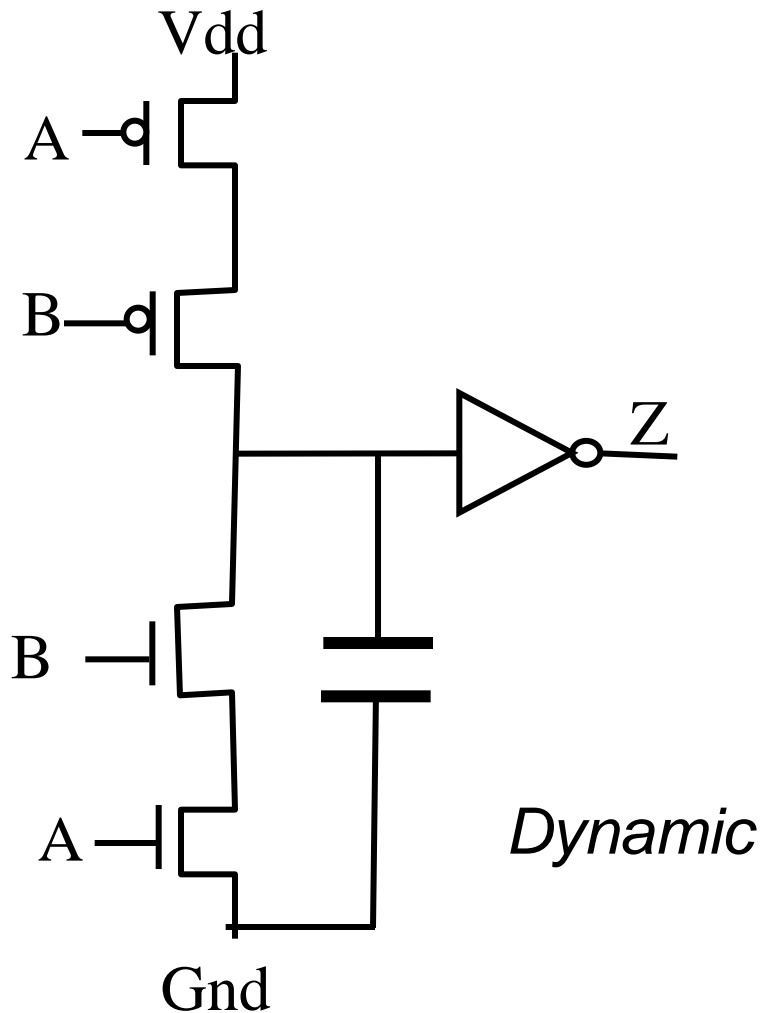


| A | B | Z <sup>+</sup> |
|---|---|----------------|
| 0 | 0 | 0              |
| 0 | 1 | Z              |
| 1 | 0 | Z              |
| 1 | 1 | 1              |





# C-element: Other implementations

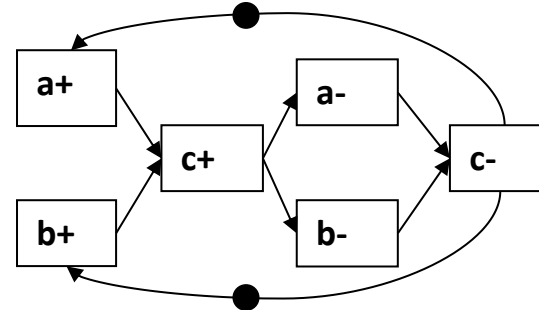
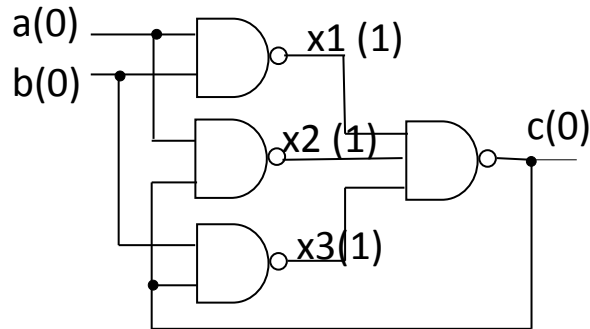


# Why and what is causal acknowledgment?

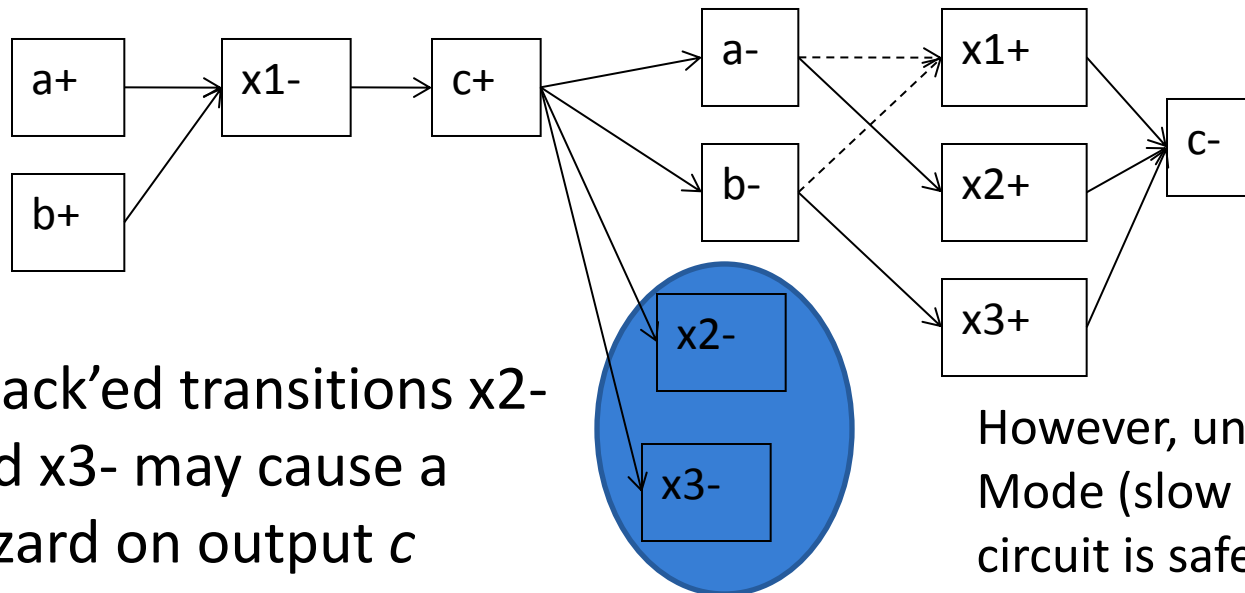


**Every signal event must be acknowledged  
by another event**

# Causal acknowledgment



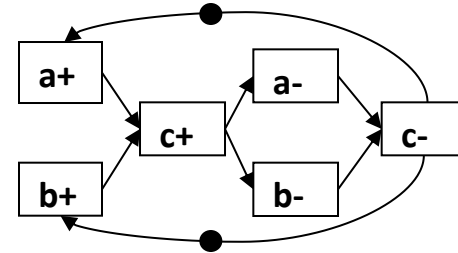
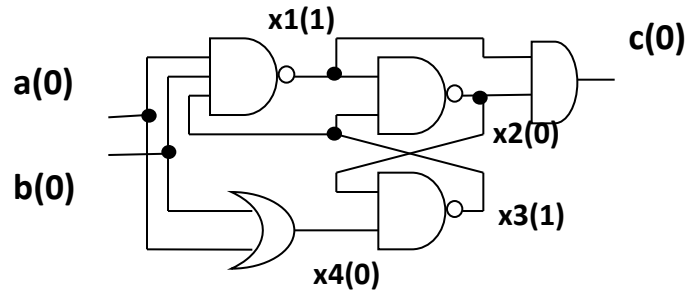
C-element implementation using simple gates



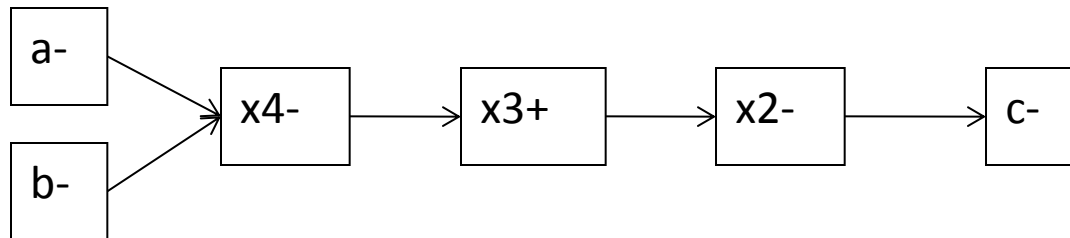
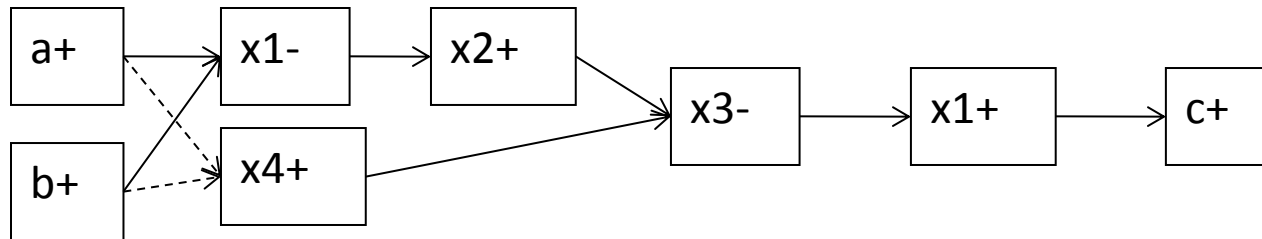
Unack'ed transitions  $x2-$  and  $x3-$  may cause a hazard on output  $c$

However, under Fundamental Mode (slow environment) the circuit is safe

# Principle of causal acknowledgement

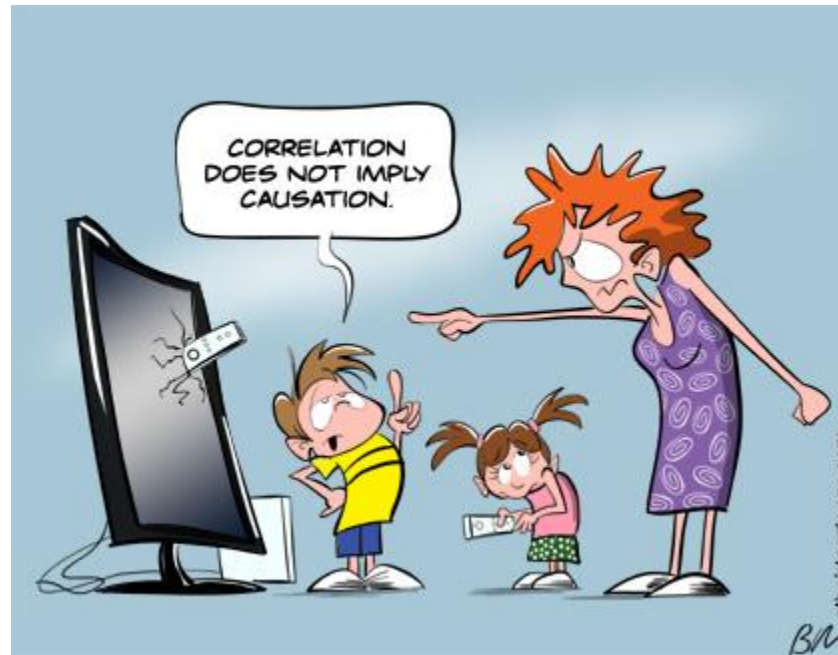


C-element implementation using simple gates



Each transition is causally ack'ed, hence no hazards can appear

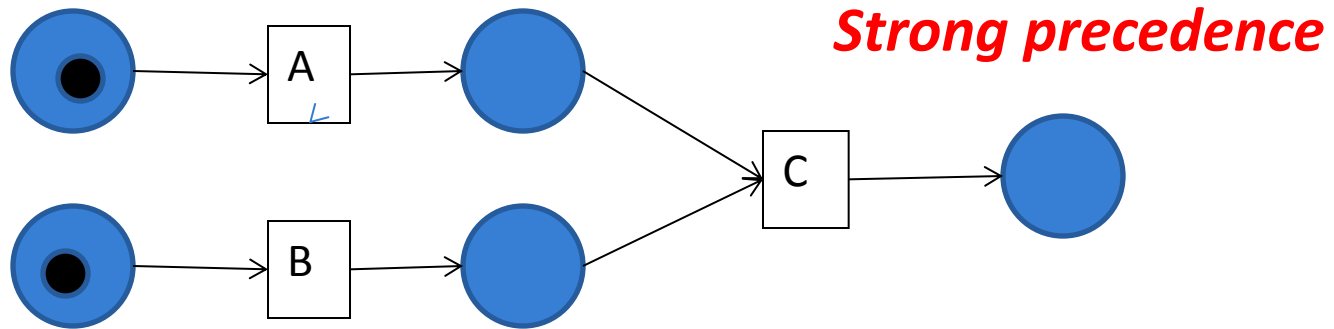
# Why and what are strong and weak causality ?



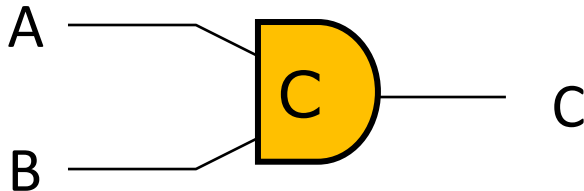
**Degree of necessity of precedence of some events for other events**

# Strong Causality

- Petri net transitions synchronising as rendez-vous



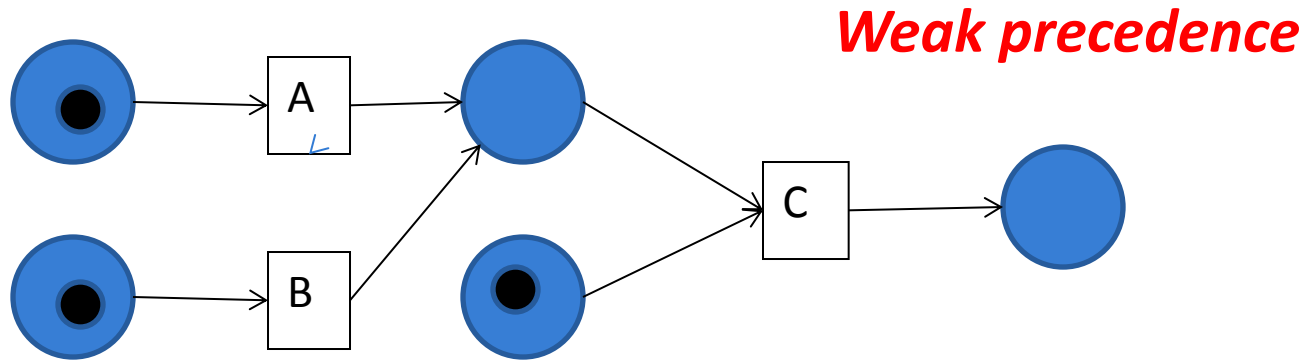
- Logic circuits: Muller C-element (in 0-1 and 1-0 transitions), AND gate (in 0-1 transitions), OR gate (in 1-0 transitions)



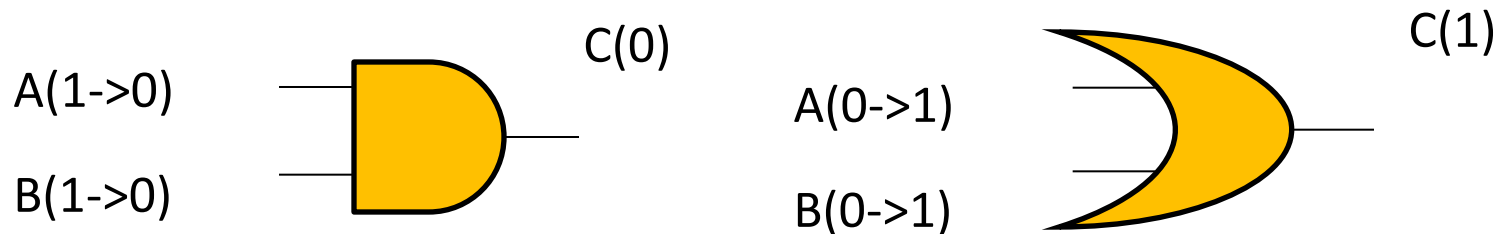
| A | B | C <sup>+</sup> |
|---|---|----------------|
| 0 | 0 | 0              |
| 0 | 1 | C              |
| 1 | 0 | C              |
| 1 | 1 | 1              |

# Weak Causality

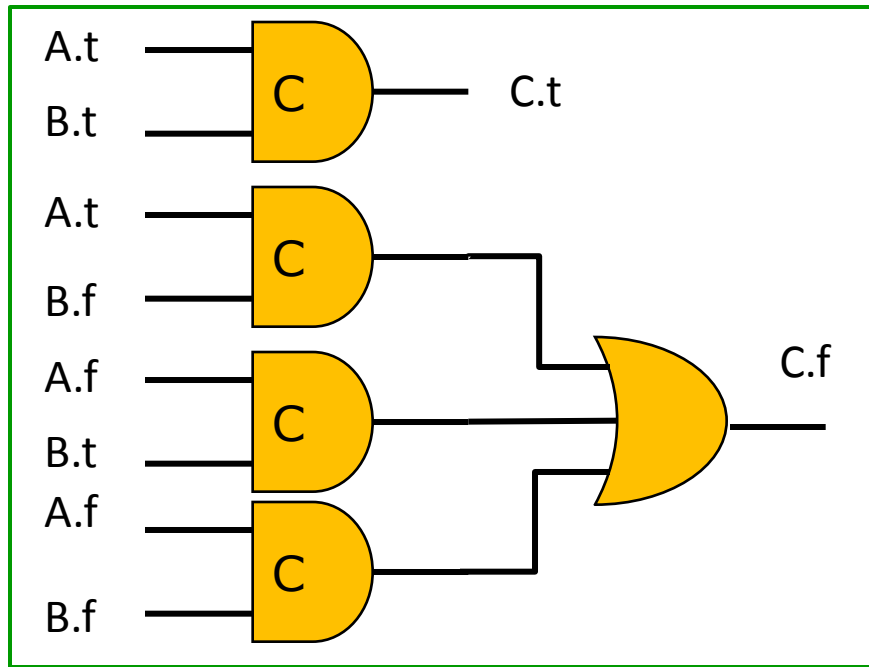
- Petri net transitions communicating via places



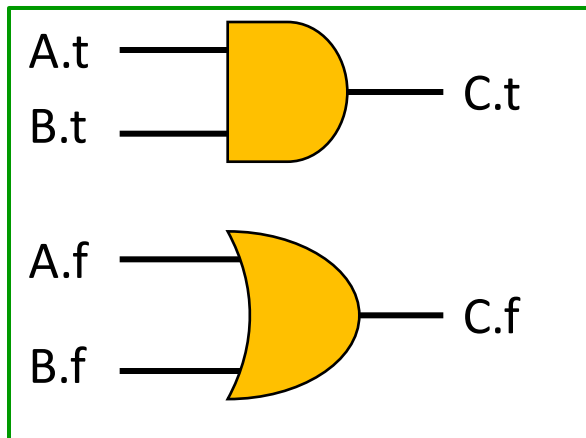
- Logic circuits: AND gate (in 1-0 transitions), OR gate (in 0-1 transitions)



# Full indication versus Early Evaluation



Dual-rail AND gate  
with full input  
acknowledgement



Dual-rail AND gate  
with “early propagation”

**Allows outputs to be produced from NULL to Codeword only when some (required) inputs have transitioned from NULL to Codeword (similar for Codeword to NULL)**



# Why and what is timing comparison?

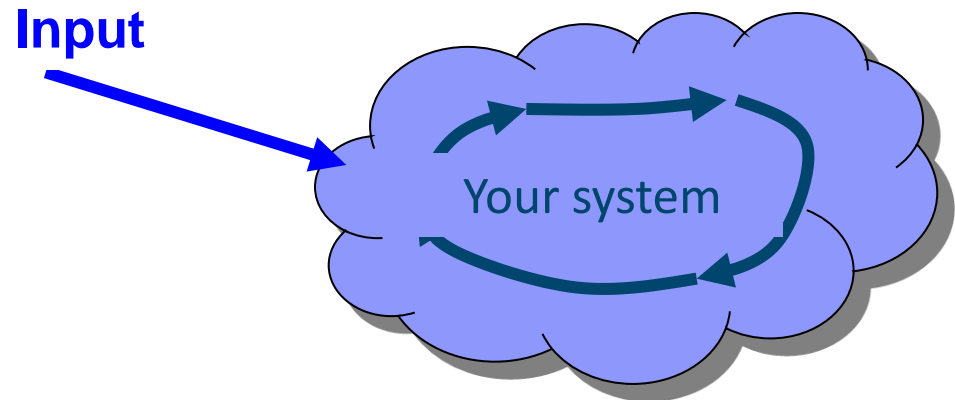


**Telling if some event happened before another event**

# Synchronizers and arbiters

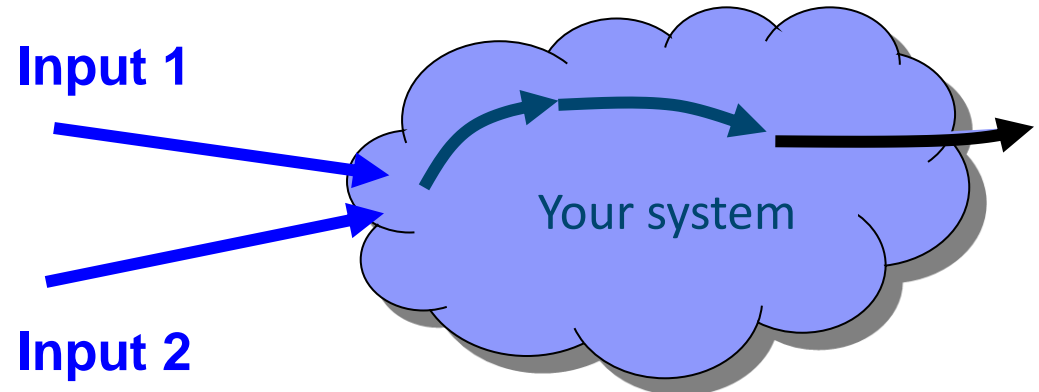
- Synchronizer

Decides which clock cycle to use for the input data

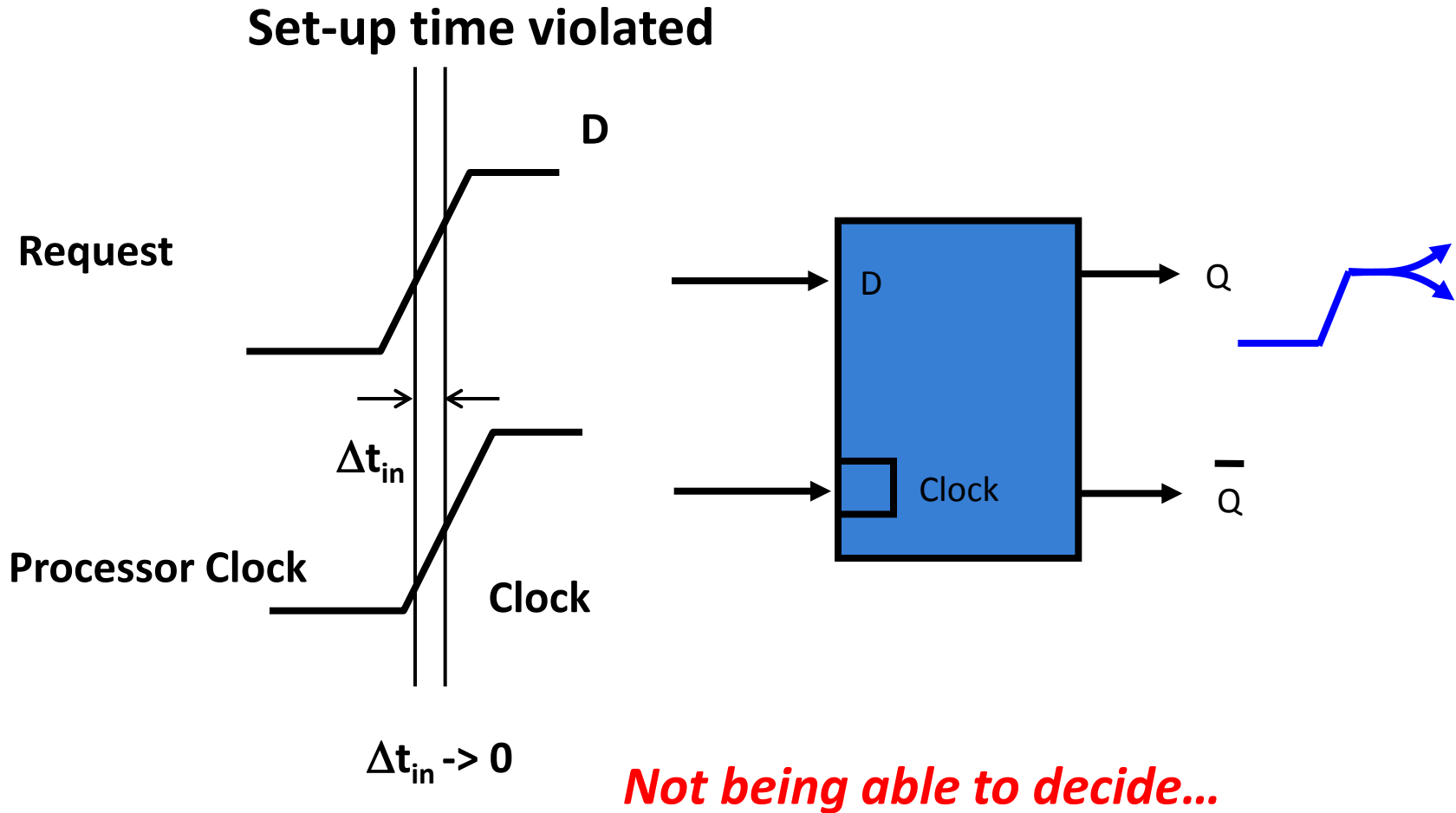


- Asynchronous arbiter

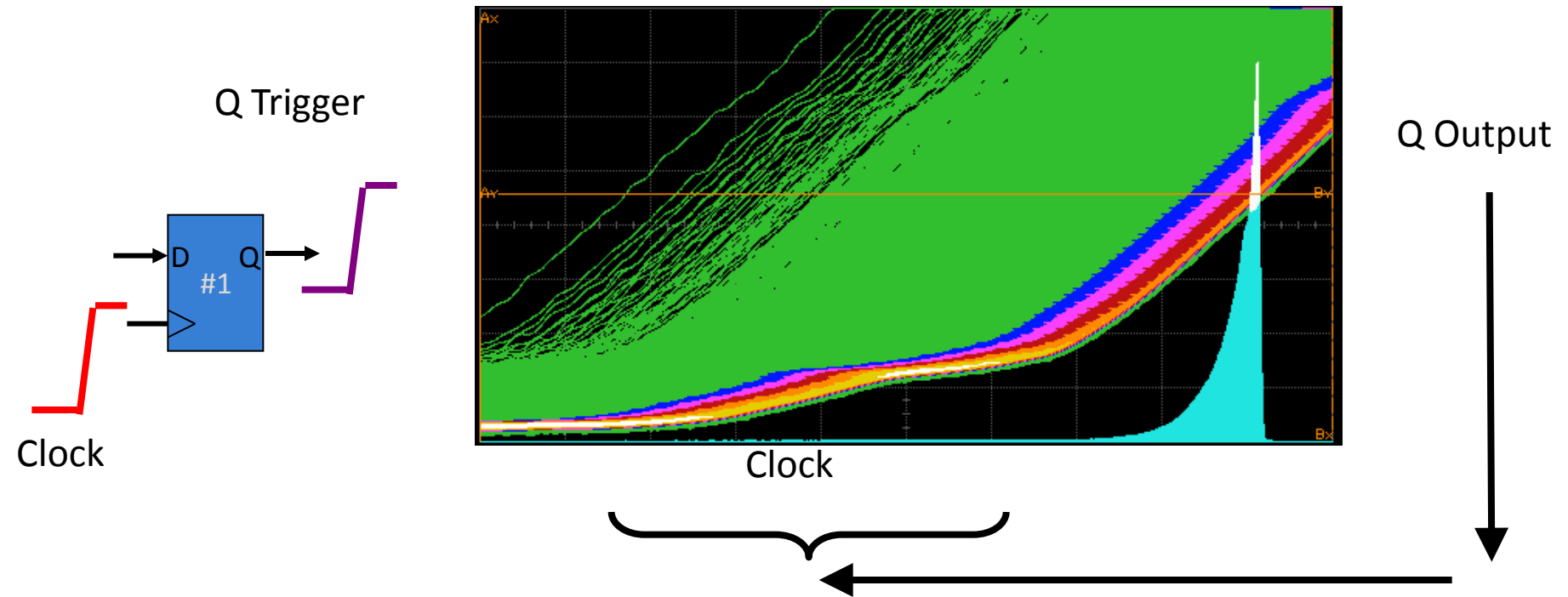
Decides the order of inputs



# Metastability is....



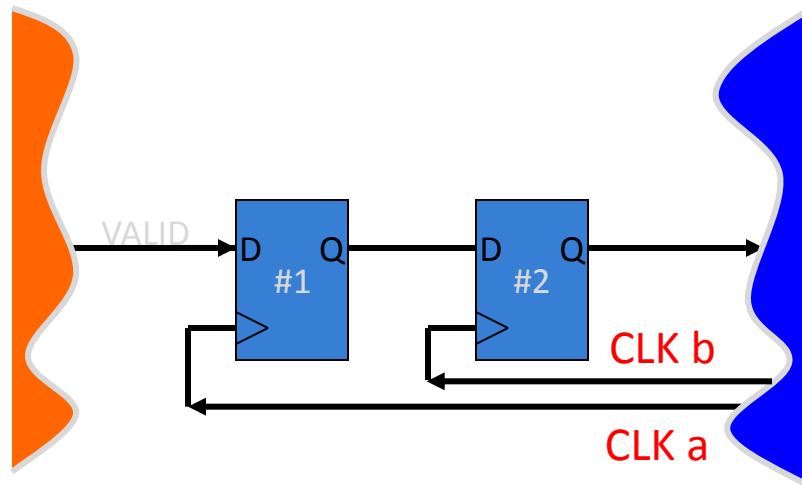
# Typical responses



- We assume all starting points are equally probable
- Most are a long way from the “balance point”
- A few are very close and take a long time to resolve

# Synchronizer

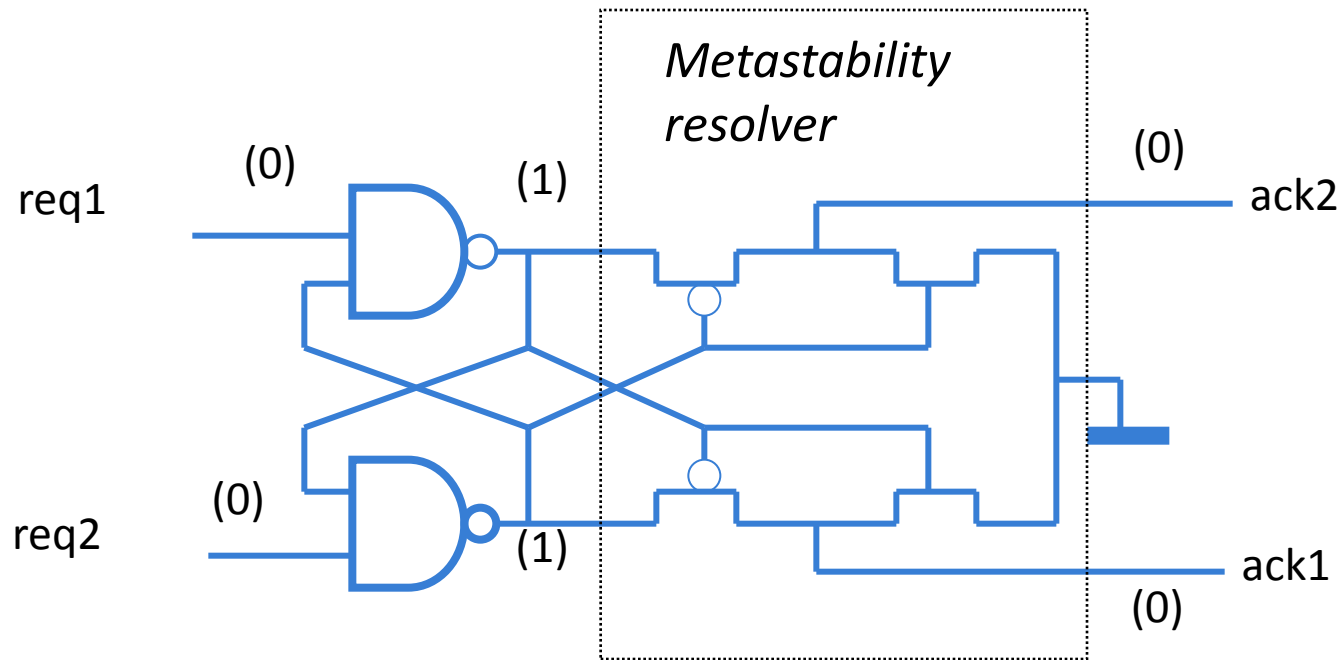
- $t$  is time allowed for the Q to change between CLK a and CLK b
- $\tau$  is the recovery time constant, usually the gain-bandwidth of the circuit
- $T_w$  is the “metastability window” (aperture around clock edge in which the capture of data edge causes a delay that is greater than normal propagation delay of the FF)
- $\tau$  and  $T_w$  depend on the circuit
- We assume that all values of  $\Delta t_{in}$  are equally probable



$$MTBF = \frac{e^{t/\tau}}{T_w \cdot f_c \cdot f_d}$$

# Two-way arbiter (Mutual exclusion element)

Basic arbitration element: Mutex (due to Seitz, 1979)



An asynchronous data latch with metastability resolver can be built similarly

# Importance of Timing Comparison

- **Understanding metastability is becoming very important as analogue and digital domains get closer, and timing uncertainty and PVT variations increase**
- **Arbitration and synchronization are increasing their importance due to many-core, timing domains, NoCs, GALS**
- **Design automation for metastability and synchronization is turning from research to practice (Blendix)**

# Pros...

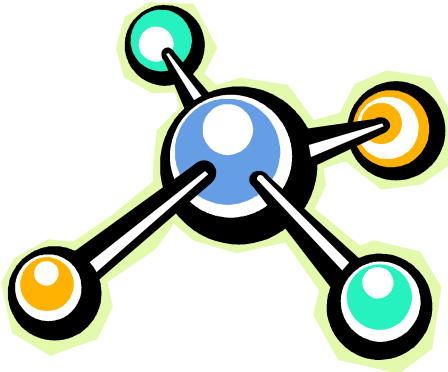
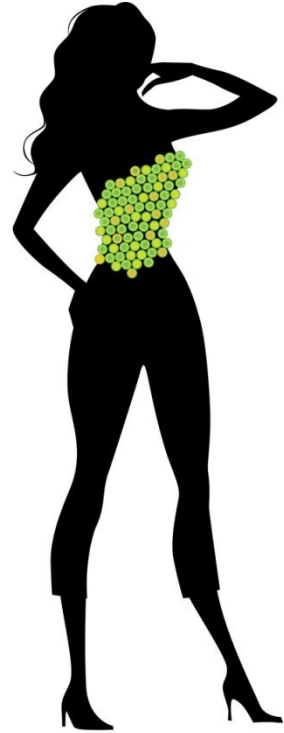
- People have always been excited by asynchronous design, and motivated by:
  - **Higher performance** (work on average not worst case delays)
  - **Lower power consumption** (automatic fine-grain “clock” gating; automatic instantaneous stand-by at arbitrary granularity in time and function; distributed localized control; more architectural options/freedom; more freedom to scale the supply voltage)
  - **Modularity** (Timing is at interfaces)
  - **Lower EMI and smoother I<sub>dd</sub>** (the local “clocks” tend to tick at random points in time)
  - **Low sensitivity to PVT variations** (timing based on matched delays or even *delay insensitive*)
  - **Secure chips** (white noise current spectrum)
  - **Plus, ... a lot of scope and fun for research (there are many unexplored paths in this forest!)**



# ... Cons

- So why have async designers been often “crucified” in the past?
  - **Overhead** (area, speed, power)
    - Control and handshaking
    - Dual-rail and completion detection costs
  - **Hard to design**
    - yes and no, ... It’s different – **there are very many styles and variants to go and one can easily get confused which is better**
  - **Very few \*\*practical\*\* CAD tools** (but many academic tools)
    - Tools are quite specific to particular design styles and design niches; hence don’ t cover the whole spectrum
    - Complexity of timing and performance models
    - Difficulty with sign-off (for particular frequency requirements)
    - ... But the situation is improving
  - **Hard to Test**
    - Possible, but not as mature as sync

# Models and techniques for design

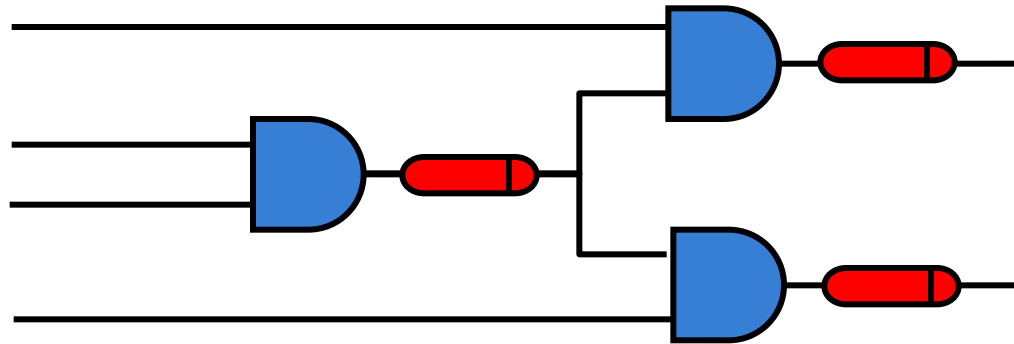


# Models and techniques for asynchronous design

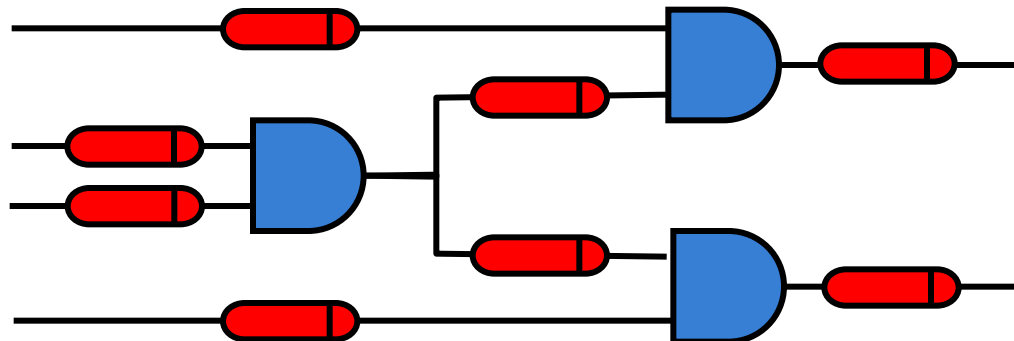
- **Models:**
  - Delay model (inertial, pure, gate delay, wire delay, bounded and unbounded delays)
  - Models of environment (fundamental mode, input-output)
  - Models of switching behaviour (state-based, event-based, hybrid)
- **RTL level:**
  - Data and control paths separate (data flow graphs, FSMs, Signal Transition Graphs, Synchronised Transitions)
  - Pipeline based (Combinational logic plus registers and latch controllers, e.g. micropipelines, gate-level pipelining)
  - Process-based (CSP-like, Balsa, Haste, Communicating Hardware Processes)
- **High-level models**
  - Flow graphs (Marked graphs, extended MGs), Petri nets, Markov Chains
  - Behavioural HDLs (C, SystemC)

# Gate vs wire delay models

- **Gate delay model: delays in gates, no delays in wires**

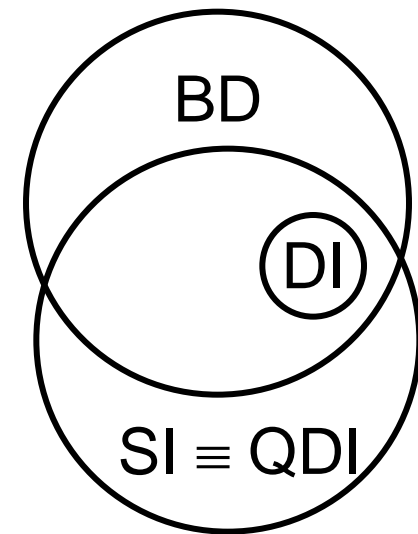


- **Wire delay model: delays in gates and wires**



# Delay models for async. circuits

- **Bounded delays (BD)**: realistic for gates and wires.
  - Technology mapping is easy, verification is difficult
- **Speed independent (SI)**: Unbounded (pessimistic) delays for gates and “negligible” (optimistic) delays for wires.
  - Technology mapping is more difficult, verification is easy
- **Delay insensitive (DI)**: Unbounded (pessimistic) delays for gates and wires.
  - DI class (built out of basic gates) is almost empty
- **Quasi-delay insensitive (QDI)**: Delay insensitive except for critical wire forks (*isochronic forks*).
  - In practice it is the same as speed independent

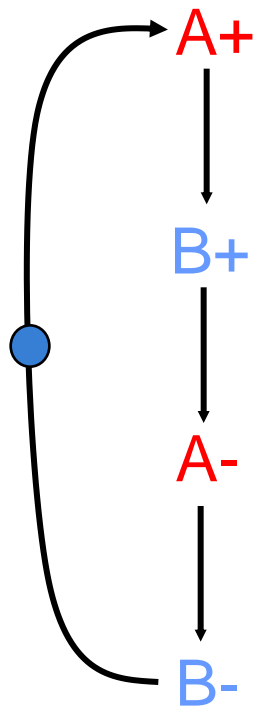


# Control Logic

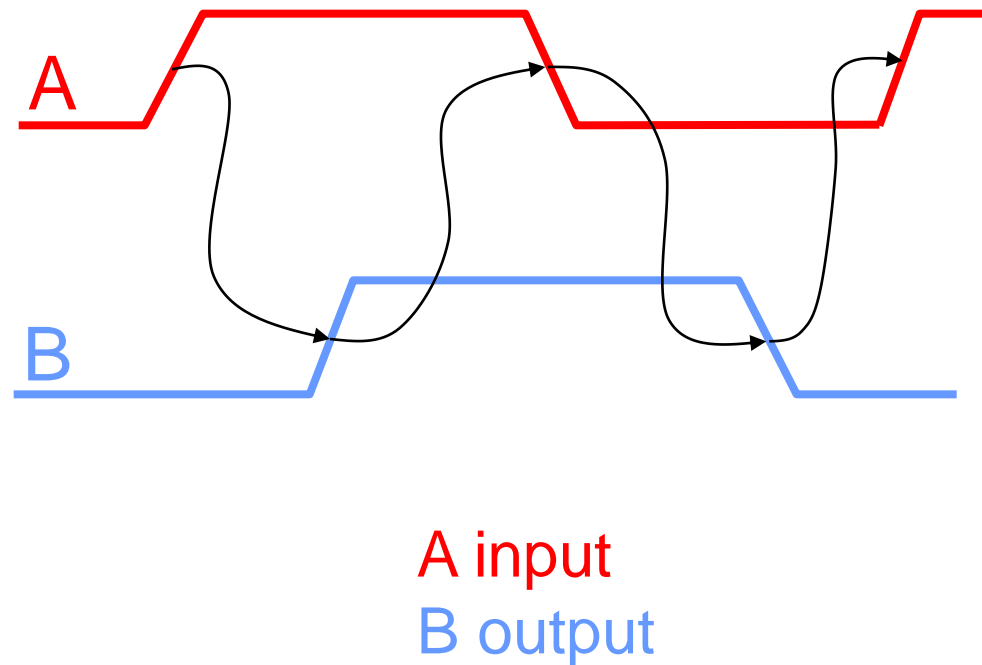
- **Control specification based on Petri nets (Signal Transition graphs)**

# Control specification

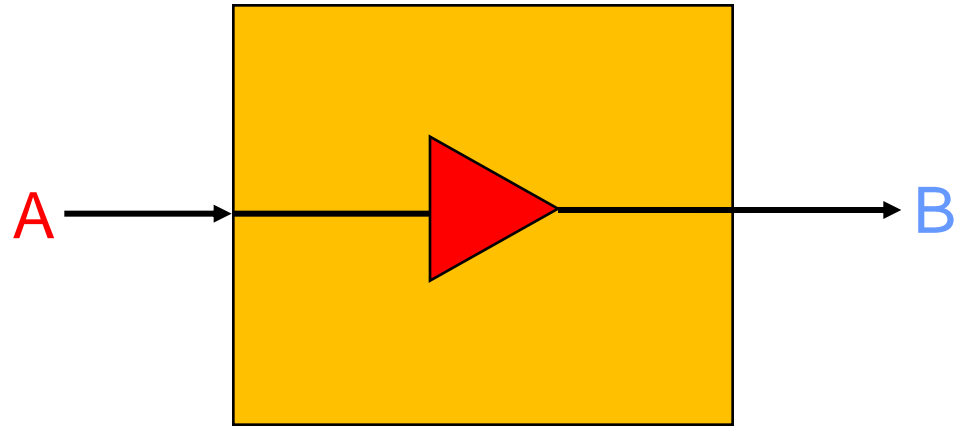
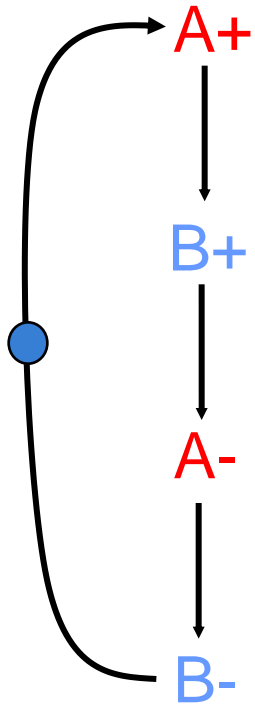
Signal Transition Graph  
(STG)



Timing Diagram

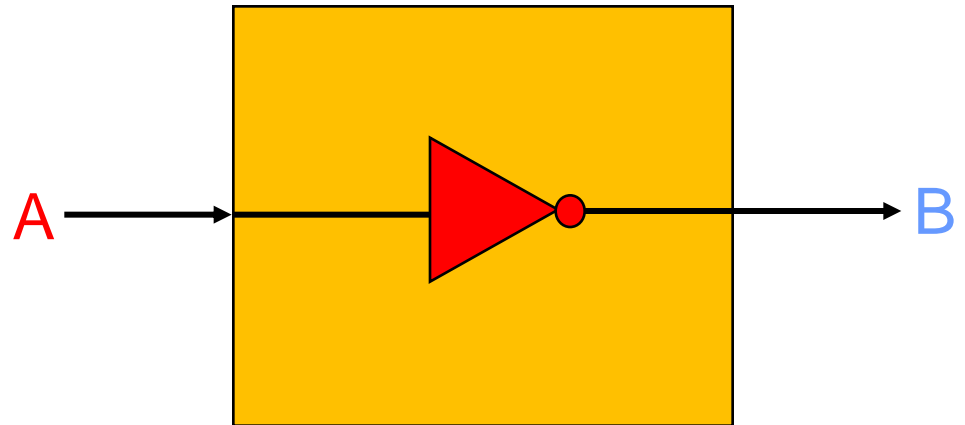
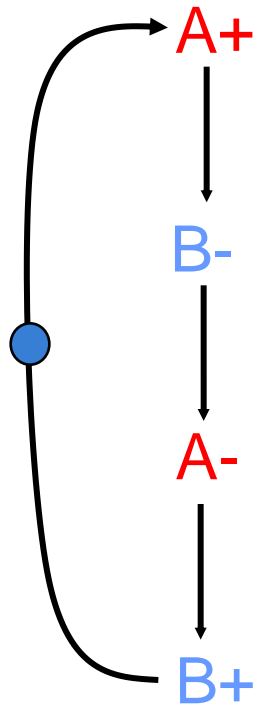


# Control specification

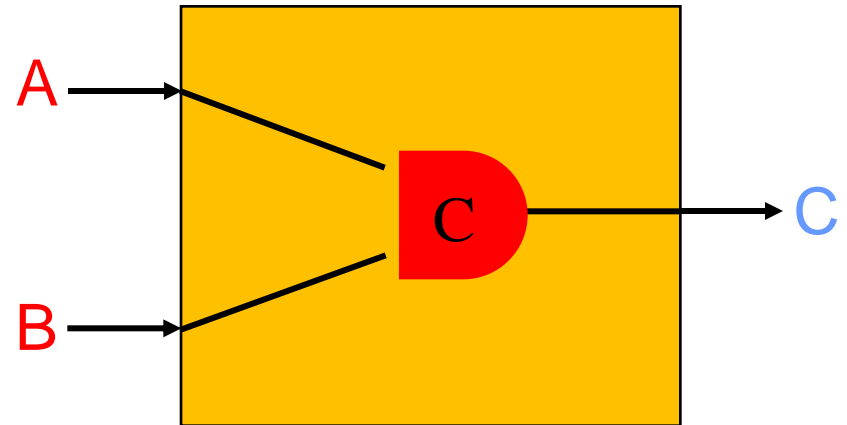
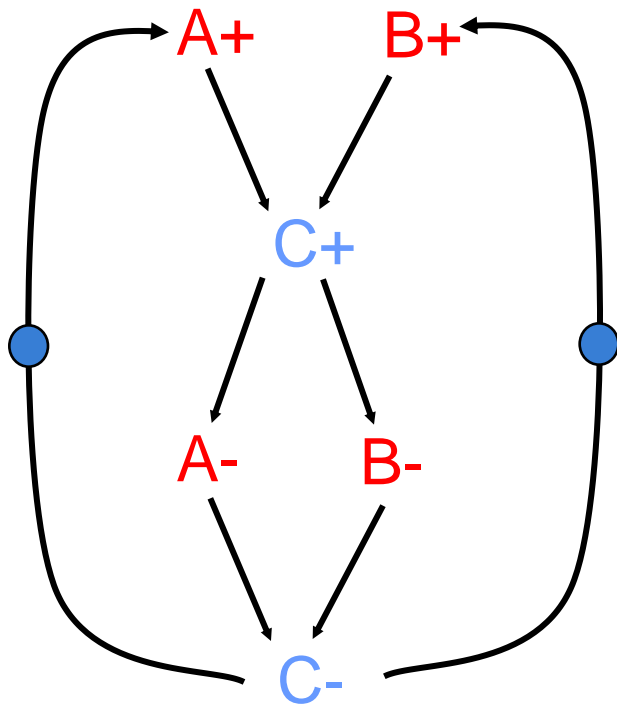




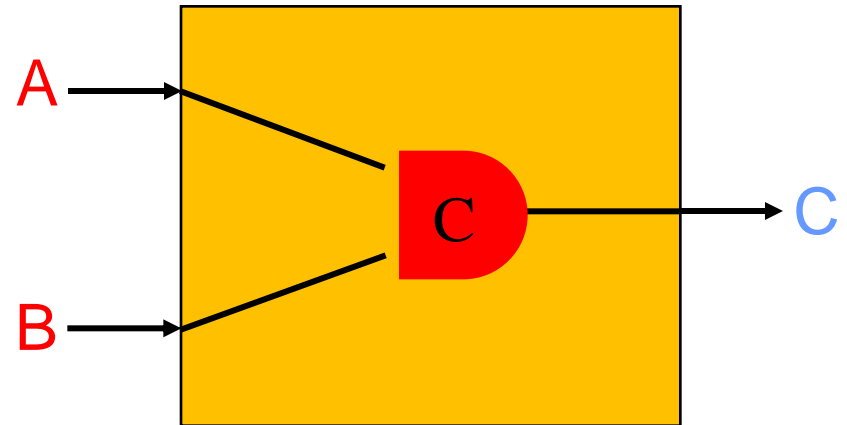
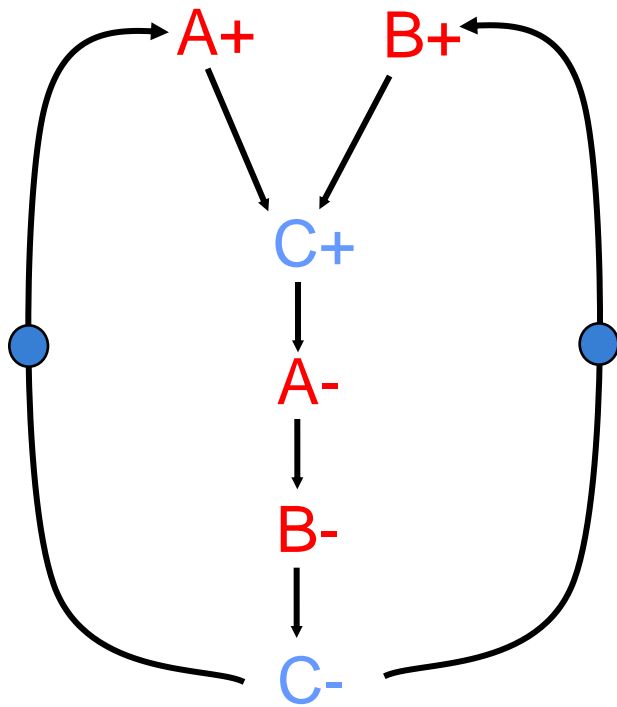
# Control specification



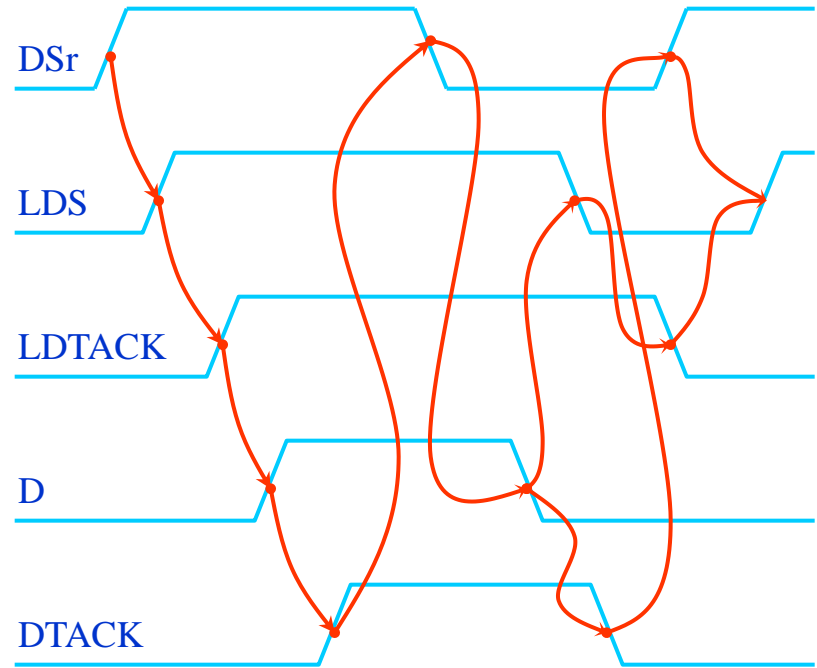
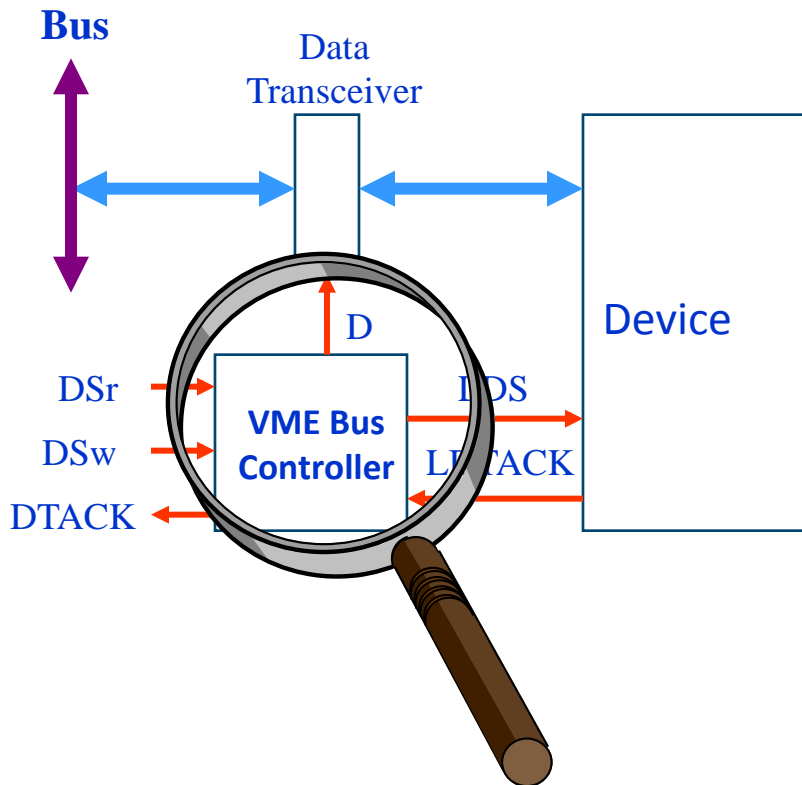
# Control specification



# Control specification

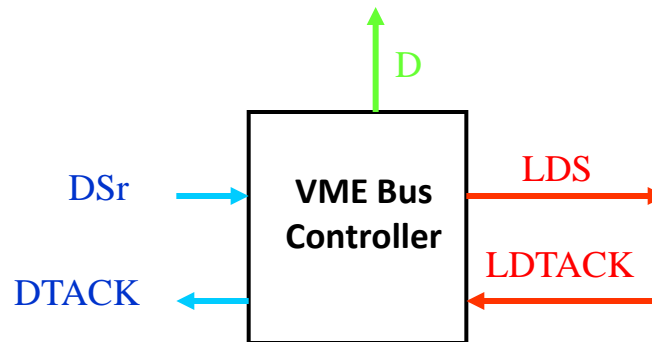
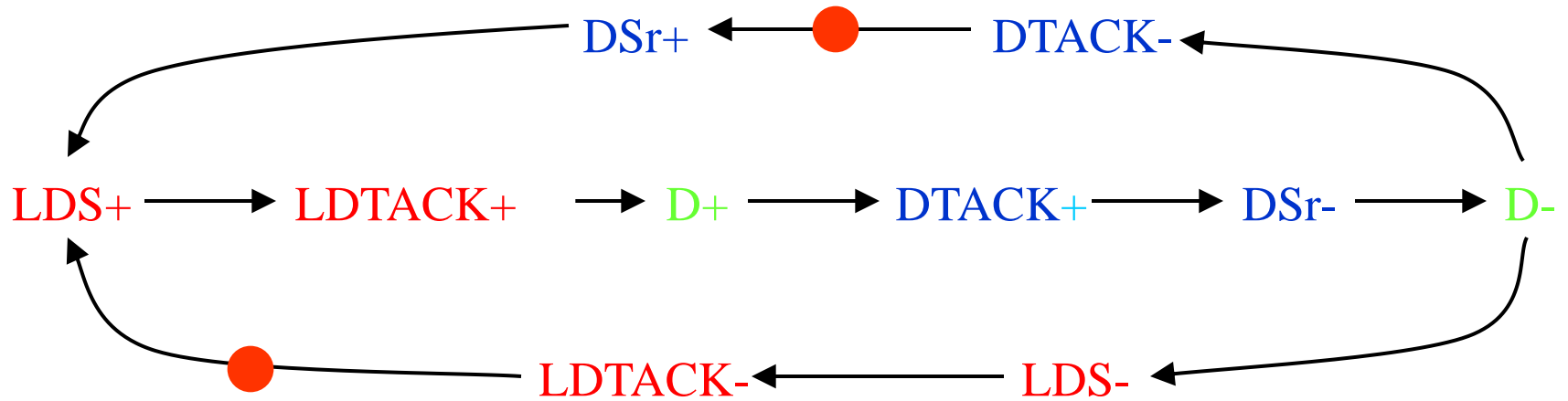


# VME bus example using Petri nets

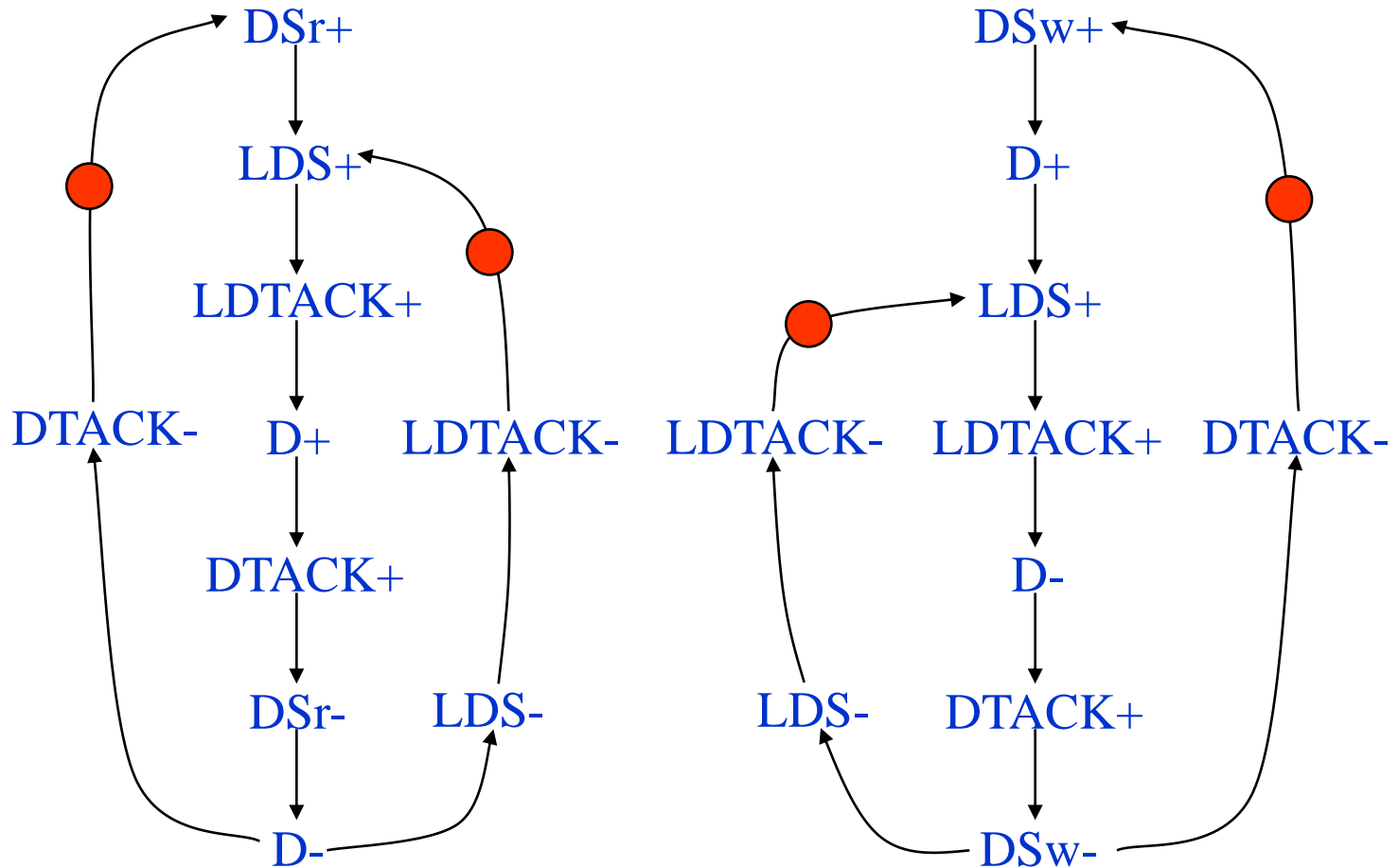


**Read Cycle**

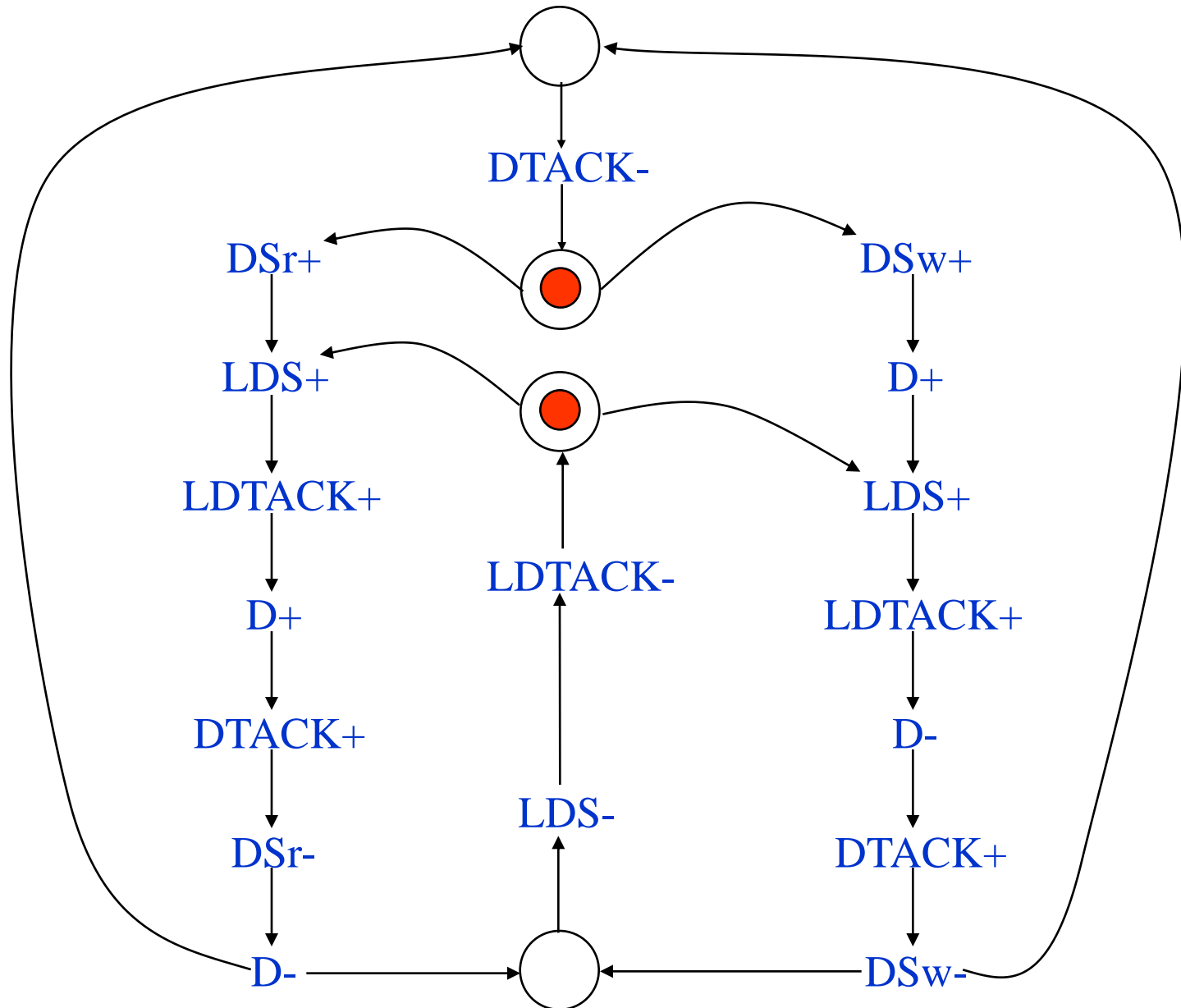
# STG for the READ cycle



# Choice: Read and Write cycles



# Choice: Read and Write cycles



# Workcraft tool

- Workcraft is a software package for **graphical edit**, analysis, synthesis and visualisation of asynchronous circuit behaviour
- Petrify plus a few other tools are part of it as plug-ins
- It is based in Java tools
- Can be downloaded from <http://workcraft.org/wiki/doku.php?id=download>
- And installed in 10 minutes.
- There is a simple to use tutorial for that



# Some references

- **General Async Design:** J. Sparsø and S.B. Furber, editors. *Principles of Asynchronous Circuit Design*, Kluwer Academic Publishers, 2001. (electronic version of a tutorial based on this book can be found on: [http://www2.imm.dtu.dk/pubdb/views/edoc\\_download.php/855/pdf/imm855.pdf](http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/855/pdf/imm855.pdf))
- **Async Control Synthesis:** J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis of Asynchronous Controllers and Interfaces*. Springer-Verlag, 2002. (Petrify software can be downloaded from: <http://www.lsi.upc.edu/~jordicf/petrify/>)
- **Arbiters and Synchronizers:** D.J. Kinniment, *Synchronization and Arbitration in Digital Systems*, Wiley and Sons, 2007 (a tutorial on arbitration and synchronization from ASYNC/NOCS 2008 can be found: <http://async.org.uk/async2008/async-nocs-slides/Tutorial-Monday/Kinniment-ASYNC-2008-Tutorial.pdf>)
- **Asynchronous on-chip interconnect:** John Bainbridge, *Asynchronous System-on-Chip Interconnect*, BCS Distinguished Dissertations, Springer-Verlag, 2002 (electronic version of the PhD thesis can be found on: [http://intranet.cs.man.ac.uk/apt/publications/thesis/bainbridge00\\_phd.php](http://intranet.cs.man.ac.uk/apt/publications/thesis/bainbridge00_phd.php))

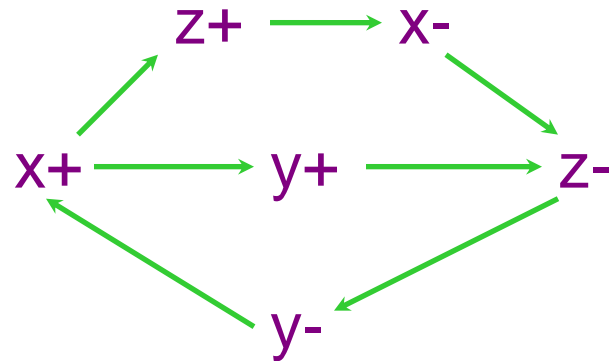
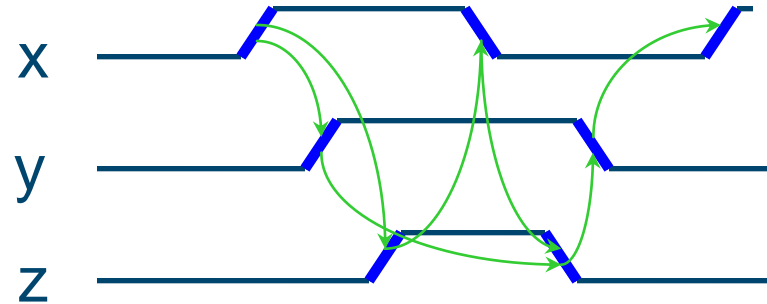
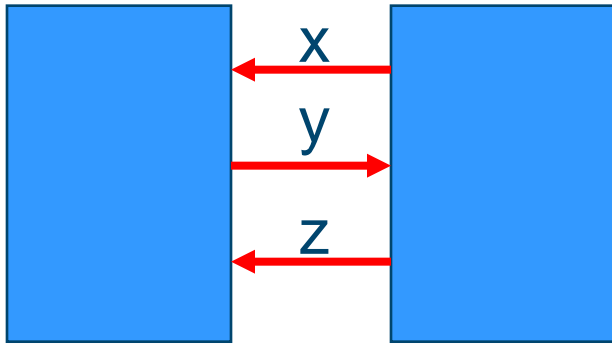
# Looking inside Logic Synthesis

- **Simple examples**

- 1) xyz-controller

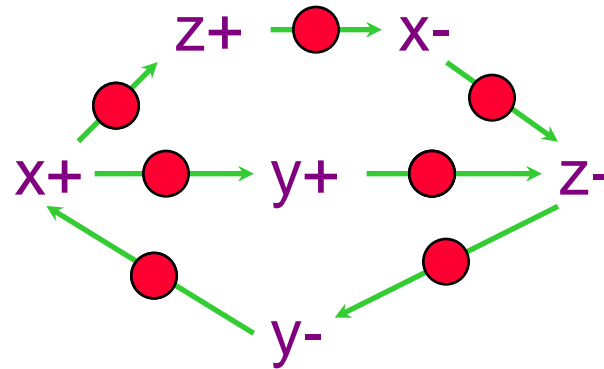
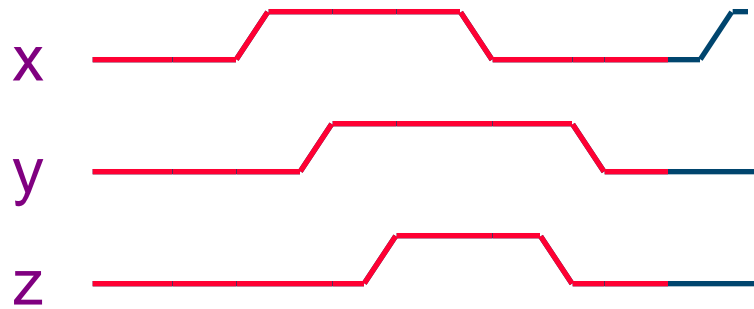
- 2) Non-overlapping clock generator

# xyz-example: Specification

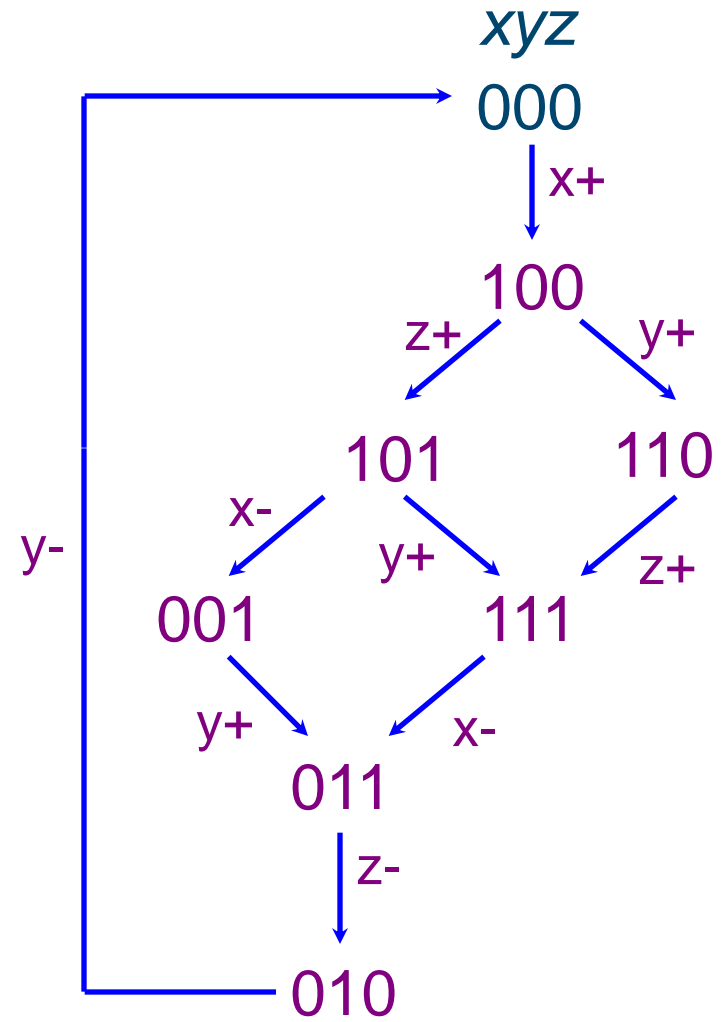
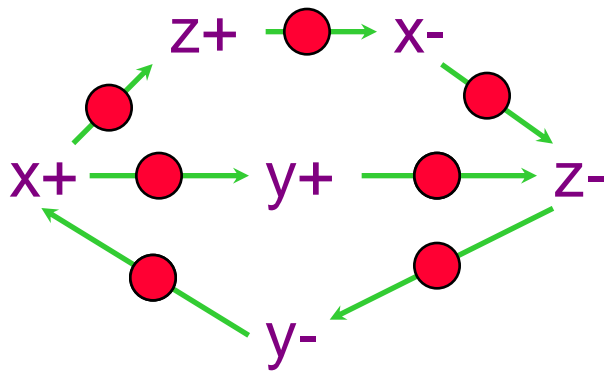


**Signal Transition Graph (STG)**

# Token flow



# State graph

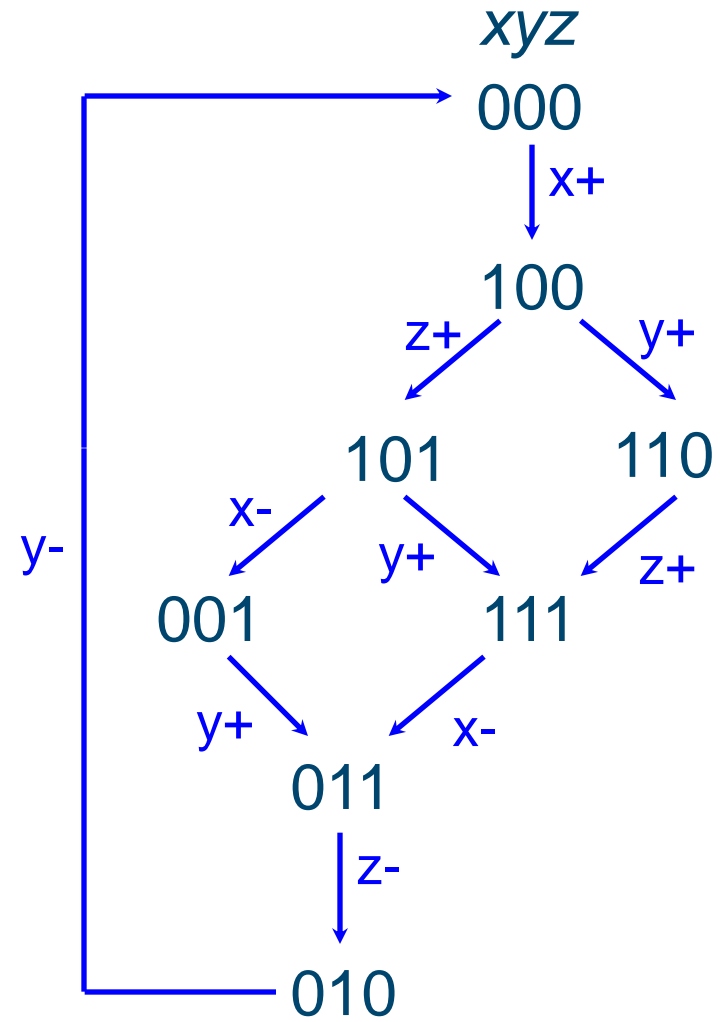


# Next-state functions

$$x = \bar{z} \cdot (x + \bar{y})$$

$$y = z + x$$

$$z = x + \bar{y} \cdot z$$



# Deriving next state functions

## 1) Truth Table

| Previous state | Next State |
|----------------|------------|
| 0*0 0          | 1 0 0      |
| 1 0*0*         | 1 1 1      |
| 0 1*0          | 0 0 0      |
| 1 1 0*         | 1 1 1      |
| 0 0*1          | 0 1 1      |
| 1*0*1          | 0 1 1      |
| 0 1 1*         | 0 1 0      |
| 1*1 1          | 0 1 1      |

## 2) Boolean Minimization

| xy | 00 | 10 | 11 | 01 |
|----|----|----|----|----|
| z  |    |    |    |    |
| 0  | 1  | 1  | 1  | 0  |
| 1  | 0  | 0  | 0  | 0  |

$$x = \bar{z} \cdot (x + \bar{y})$$

Observations in this example:

- 1) All combinations are used as states
- 2) All states appear uniquely

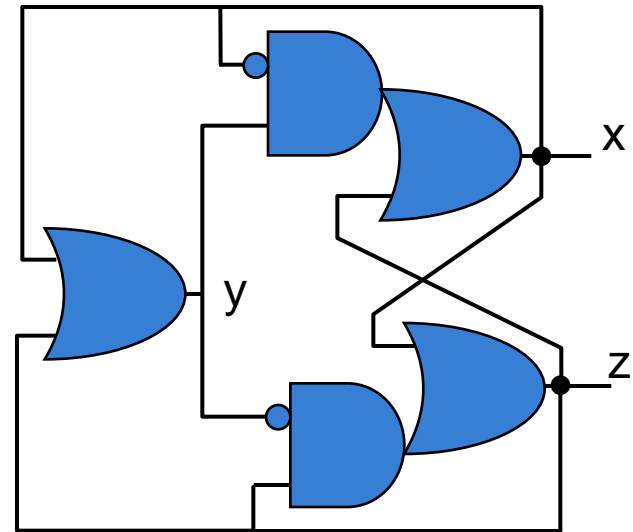
**Generally, this is not always the case!**

# Complex Gate netlist

$$x = \bar{z} \cdot (x + \bar{y})$$

$$y = z + x$$

$$z = x + \bar{y} \cdot z$$





# Circuit synthesis

- **Goal:**
  - **Derive a hazard-free circuit under a given delay model and mode of operation**

# Speed independence

- **Delay model**
  - Unbounded gate / environment delays
  - Certain wire delays shorter than certain paths in the circuit
- **Conditions for implementability:**
  - Consistency
  - Complete State Coding
  - Persistency

# Implementability conditions

- **Consistency**
  - Rising and falling transitions of each signal alternate in any trace
- **Complete state coding (CSC)**
  - Next-state functions correctly defined
- **Persistency**
  - No event can be disabled by another event (unless they are both inputs)

# Implementability conditions

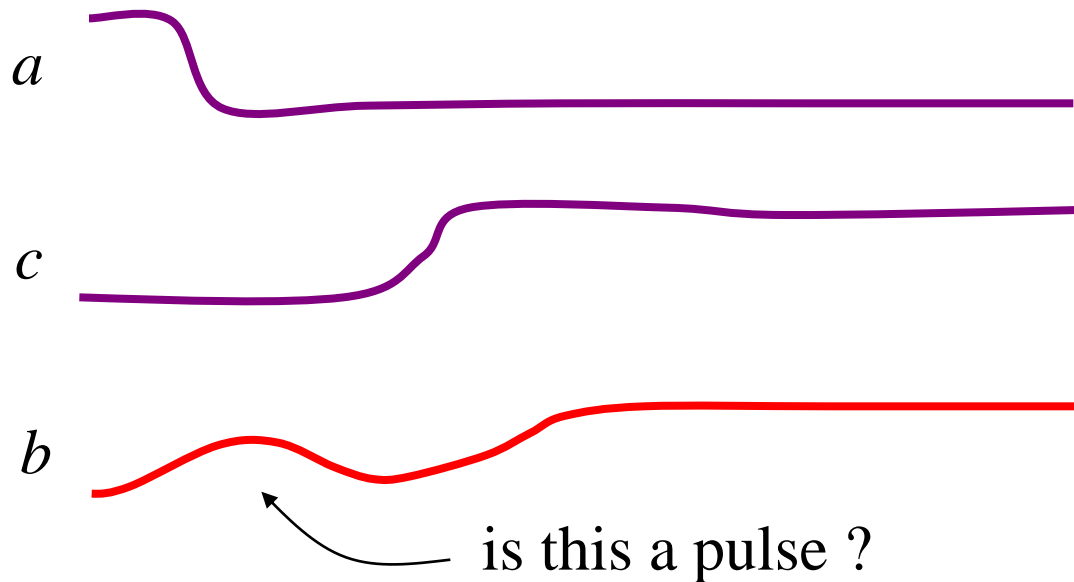
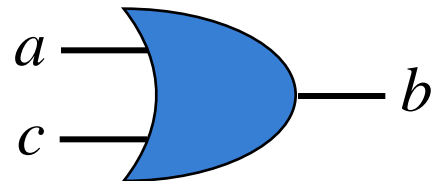
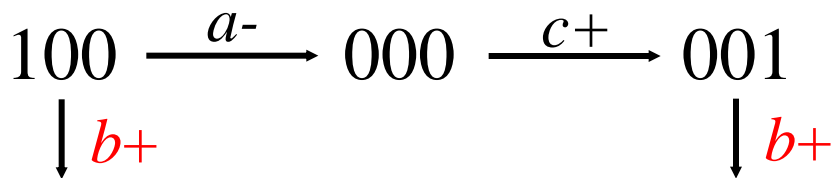
- **Consistency + CSC + persistency**



- **There exists a speed-independent circuit that implements the behavior of the STG**

**(under the assumption that any Boolean function can be implemented with one complex gate)**

# Persistence



60

Speed independence  $\Rightarrow$  glitch-free output behavior under any delay