# Asynchronous Computing (Electronic Designer's perspective)

Alex Yakovlev
Newcastle University
Newcastle upon Tyne, U.K.
[http://async.org.uk](http://async.org.uk)
[http://workcraft.org](http://workcraft.org)

First School on Reaction Systems, Toruń, Poland, 3-5 June 2019

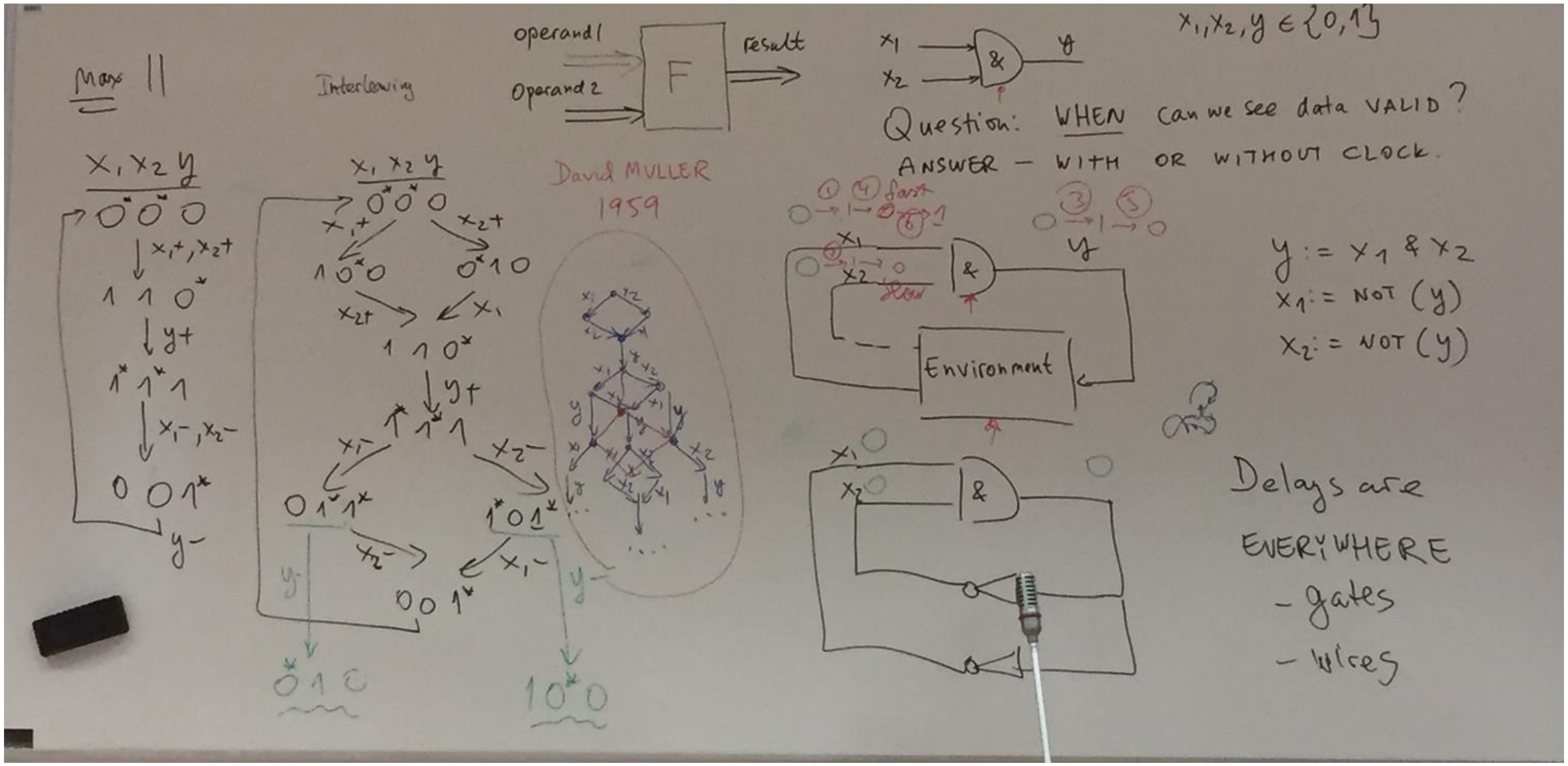
## Outline

- Simple intro examples on whiteboard
- 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 Design 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.**

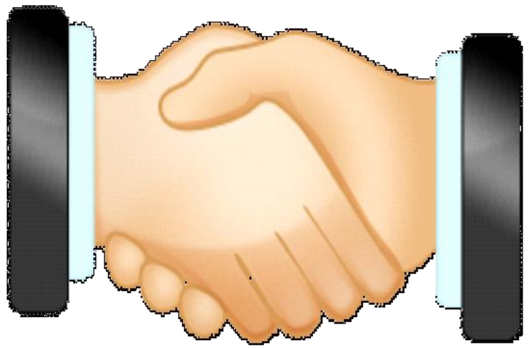
# Introduction/Motivation: Simple example



# Key Principles of Asynchronous Design

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

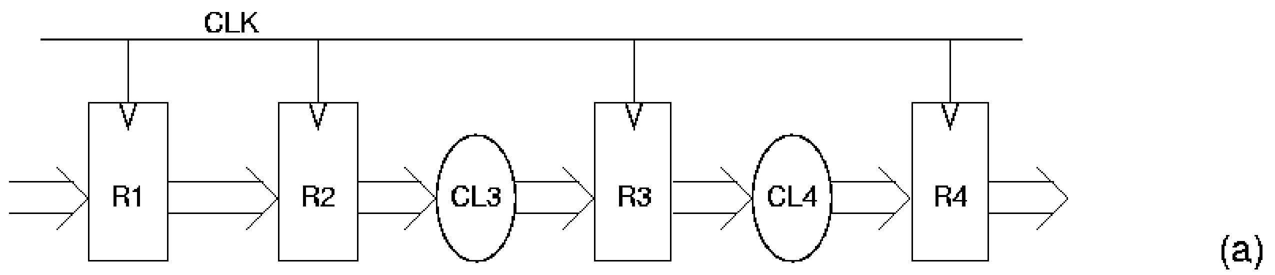
## Why and what is handshaking?



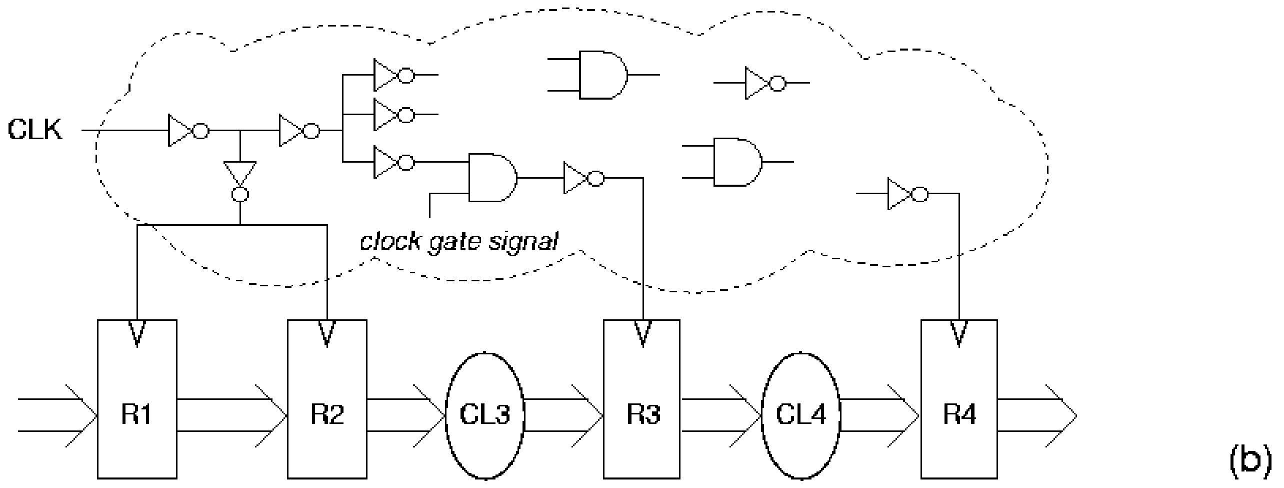
**Mutual Synchronisation is via Handshake**

# Synchronous clocking

**How we think**

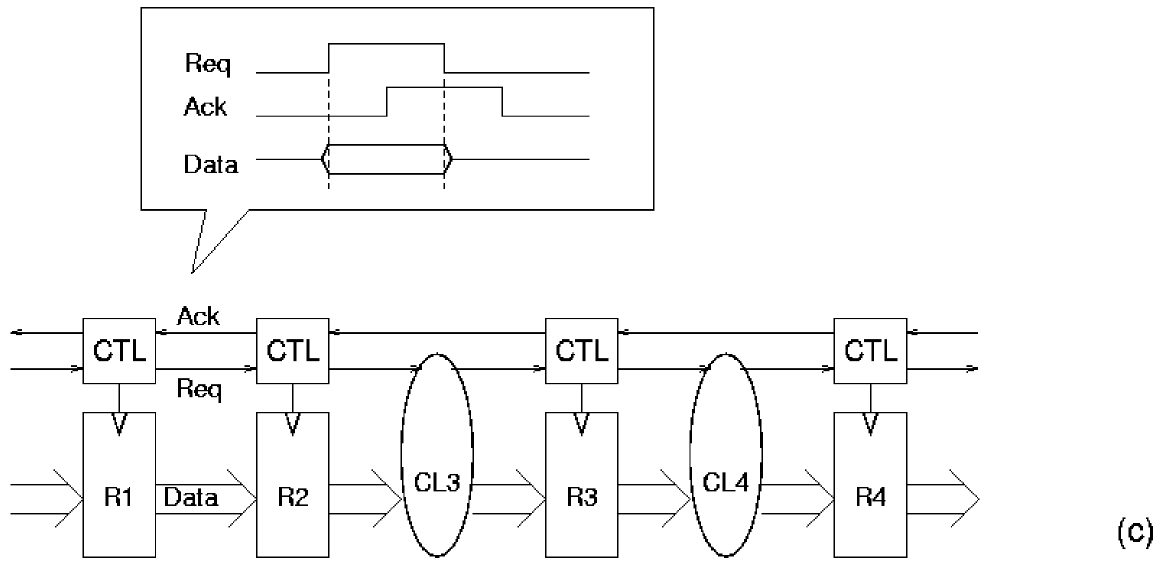


**What we design**

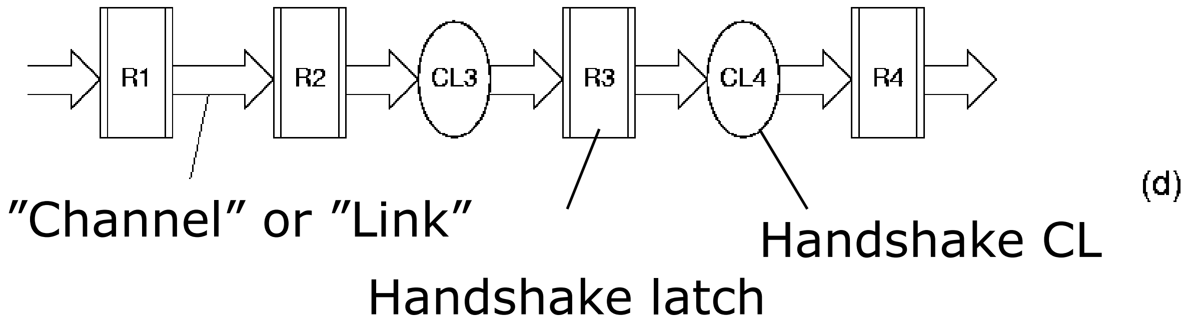


# Asynchronous handshaking

**What we design**

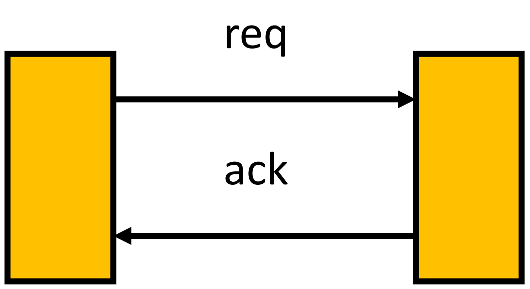


**How we think**

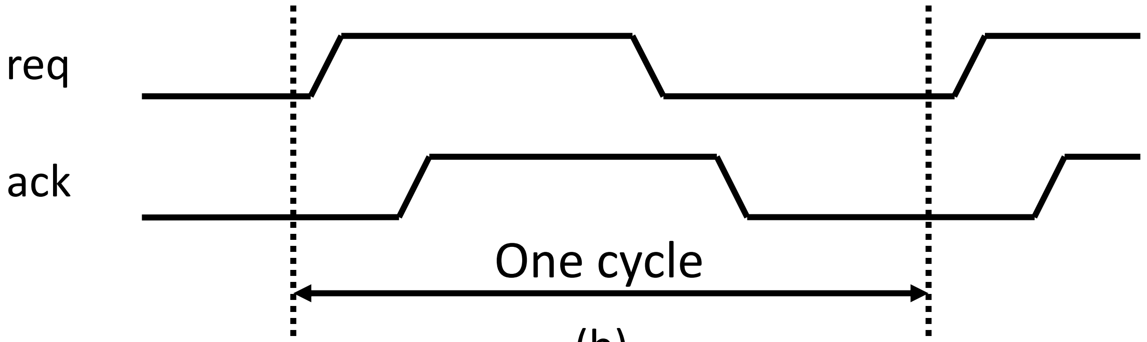


# Handshake Signalling Protocols

## Level Signalling (RTZ or 4-phase)

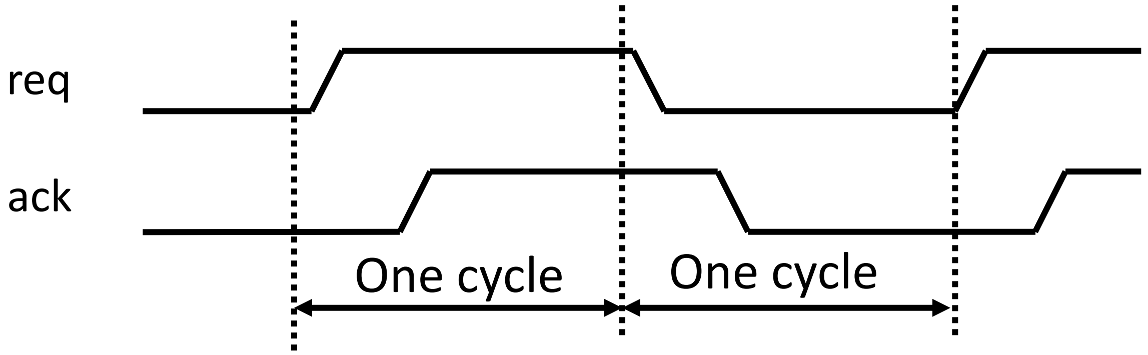


(a)

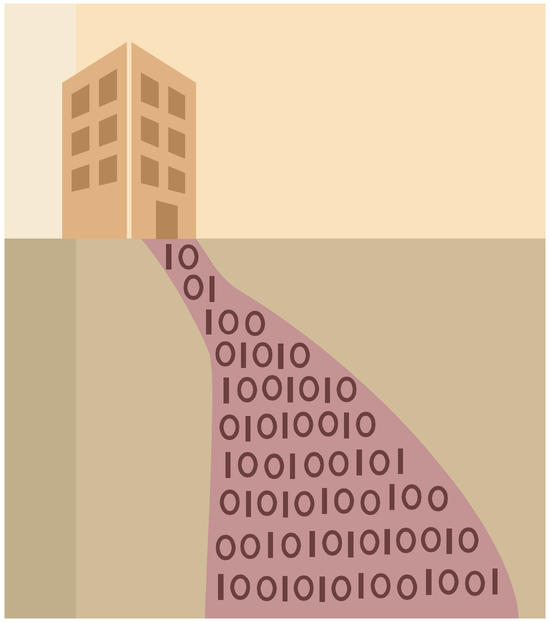


(b)

## Transition Signalling (NRZ or 2-phase)

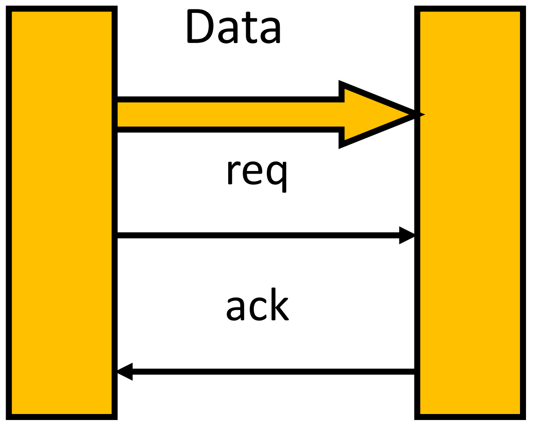


## Why and what is delay-insensitive coding?

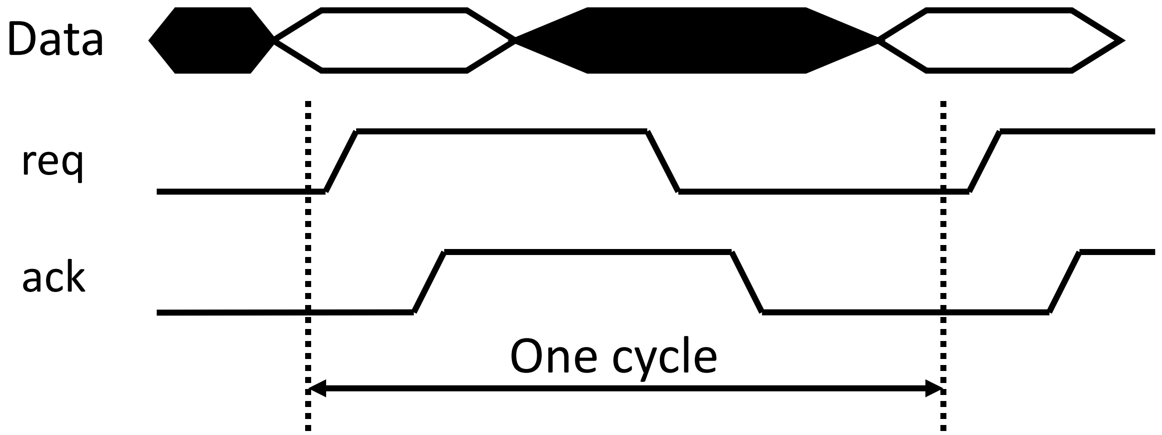


**Data Token = (Data Value, Validity Flag)**

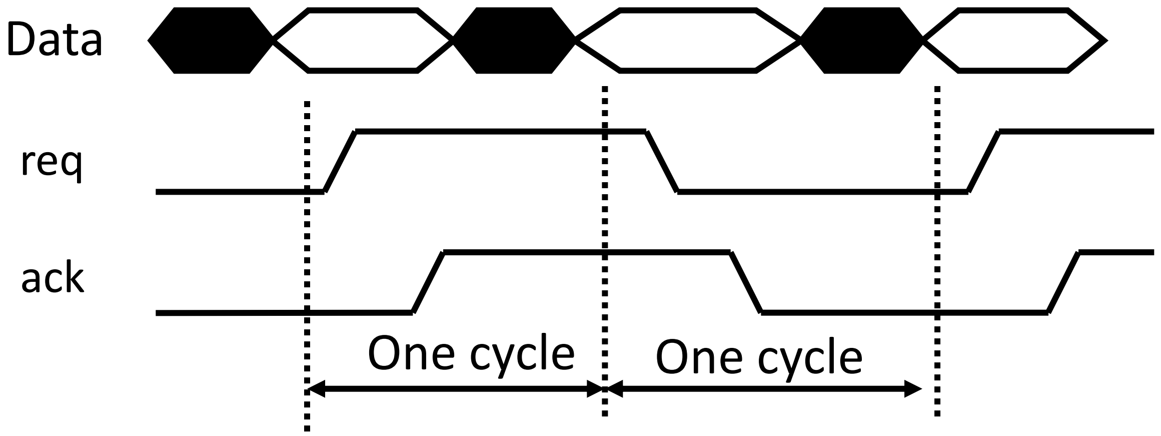
# Bundled Data



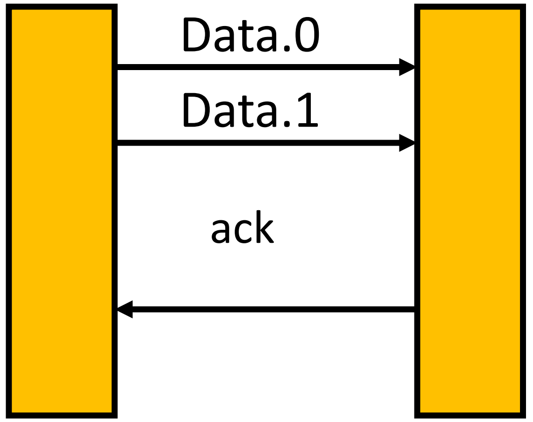
Return to Zero:

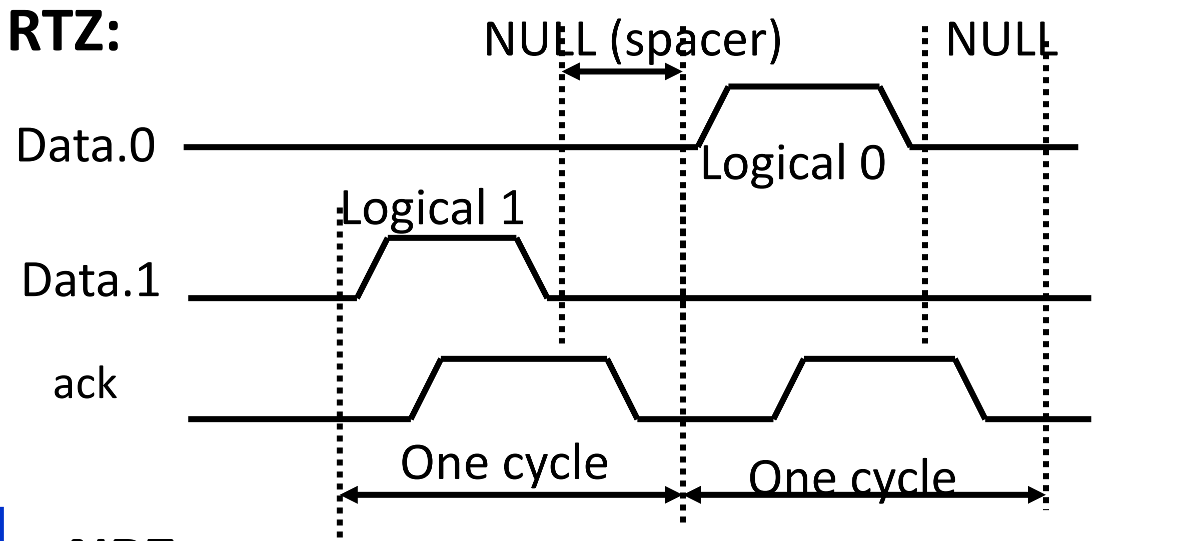


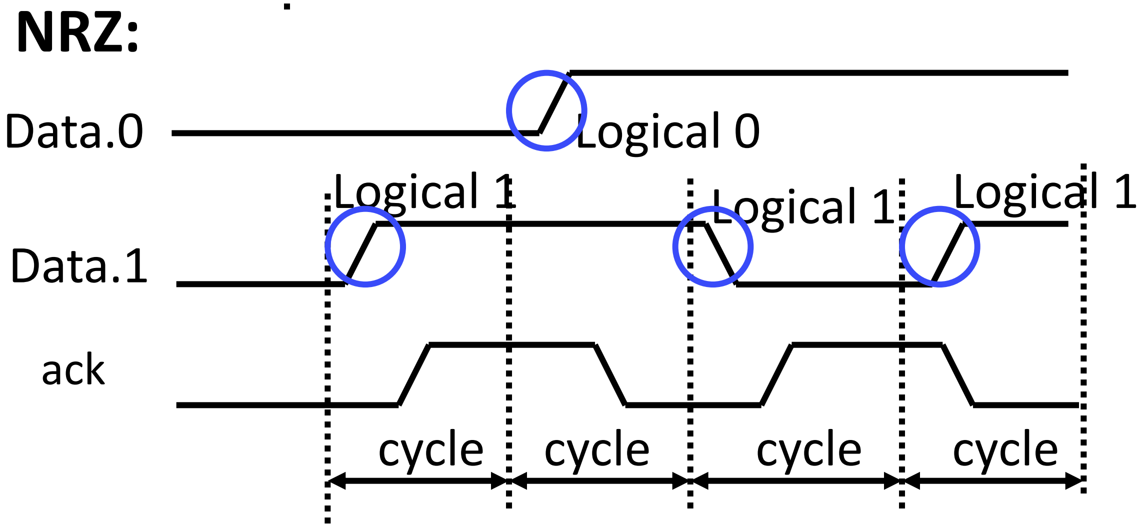
**Non-Return-to-Zero**



# Delay-Insensitive encoding (Dual-Rail)





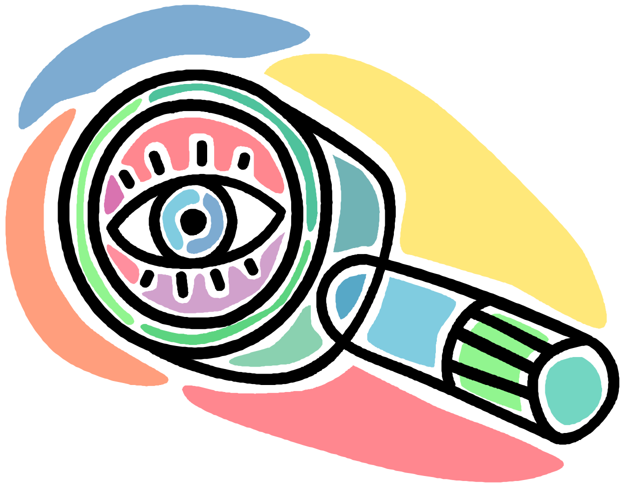


NRZ coding leads to complex logic implementation; special ways to track odd and even phases and logic values are usually needed

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

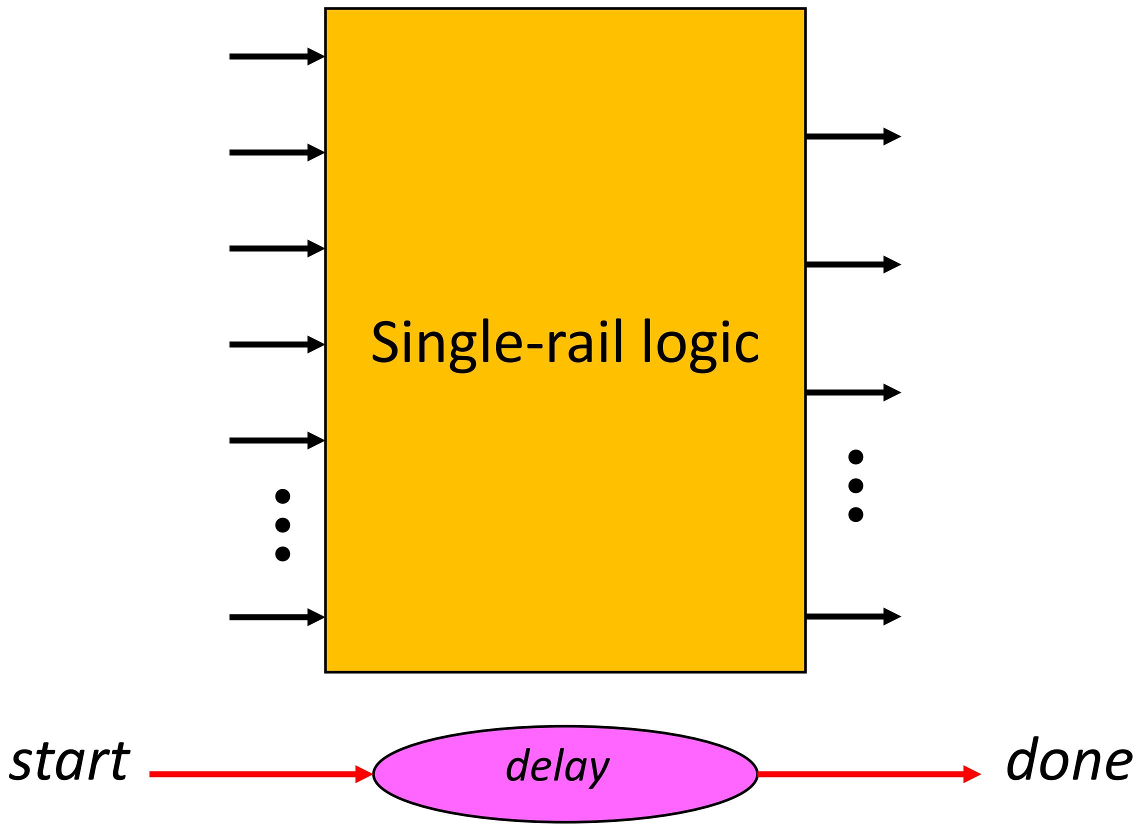
- 1-of-4:
  - 0001=> 00, 0010=>01, 0100=>10, 1000=>11
- 2-of-4:
  - 1100, 1010, 1001, 0110, 0101, 0011 – total 6 combinations (cf. 2-bit dual-rail – 4 comb.)
- 3-of-6:
  - 111000, 110100, …, 000111 – total 20 combinations (can encode 4 bits + 4 control tokens)
- 2-of-7:
  - 1100000, 1010000, …, 0000011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

## Why and what is completion detection?



**Signalling that the Transients are over**

# Bundled-data logic blocks

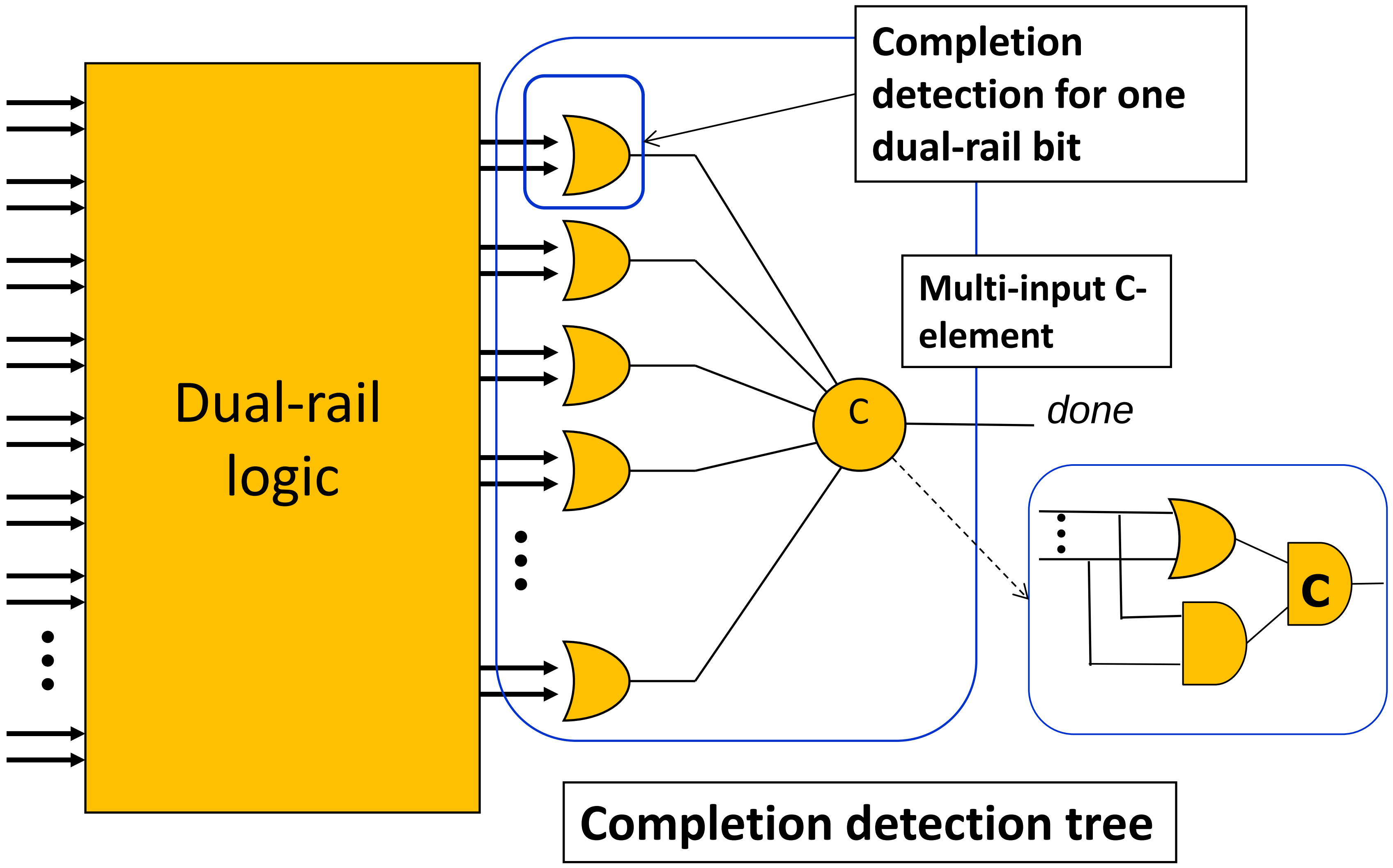


Completion is implicit: by done signal

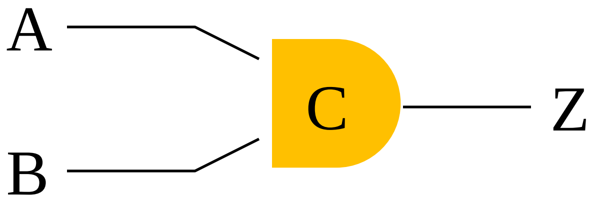
The delay must scale with the worst case delay path, So … not really self-timed

**Conventional logic + matched delay**

# True completion detection

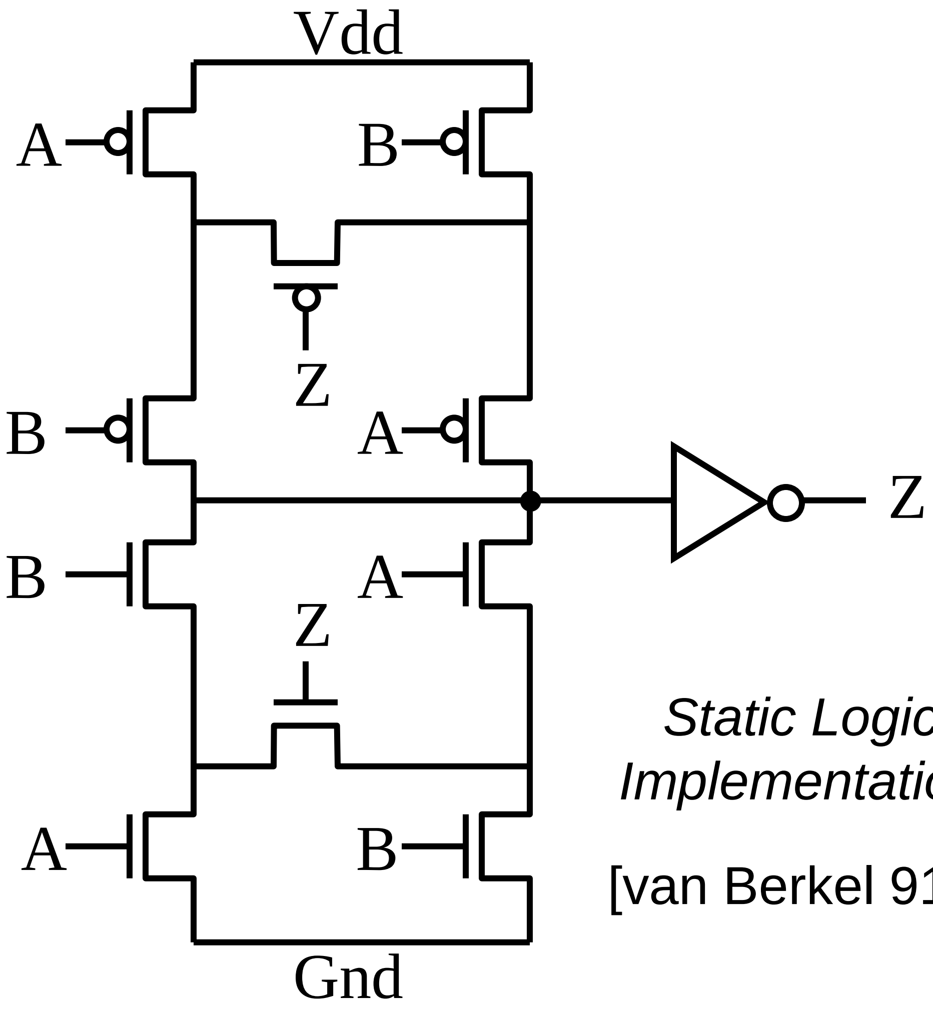

Dual-rail logic
Completion detection for one dual-rail bit
Multi-input C-element
C
done
C
Completion detection tree

# The Muller C element



Z_next=AB+Z(A+B)

| A | B | $Z^+$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | Z |
| 1 | 0 | Z |
| 1 | 1 | 1 |



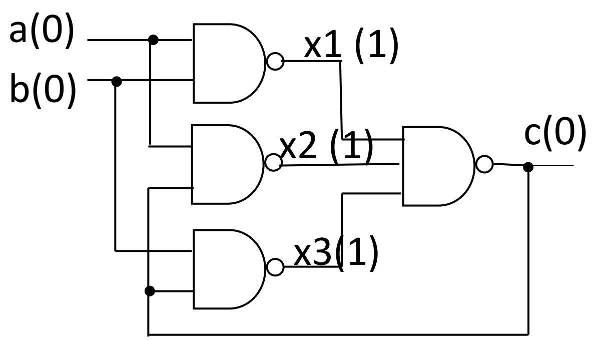
*Static Logic Implementation*

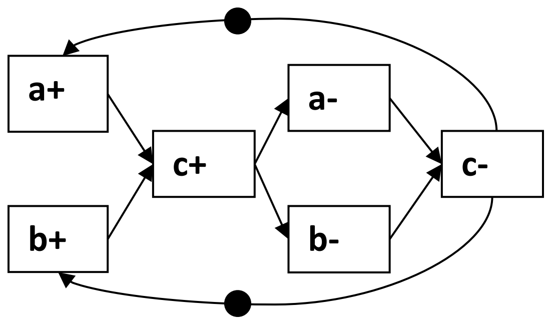
[van Berkel 91]

## Why and what is causal acknowledgment?

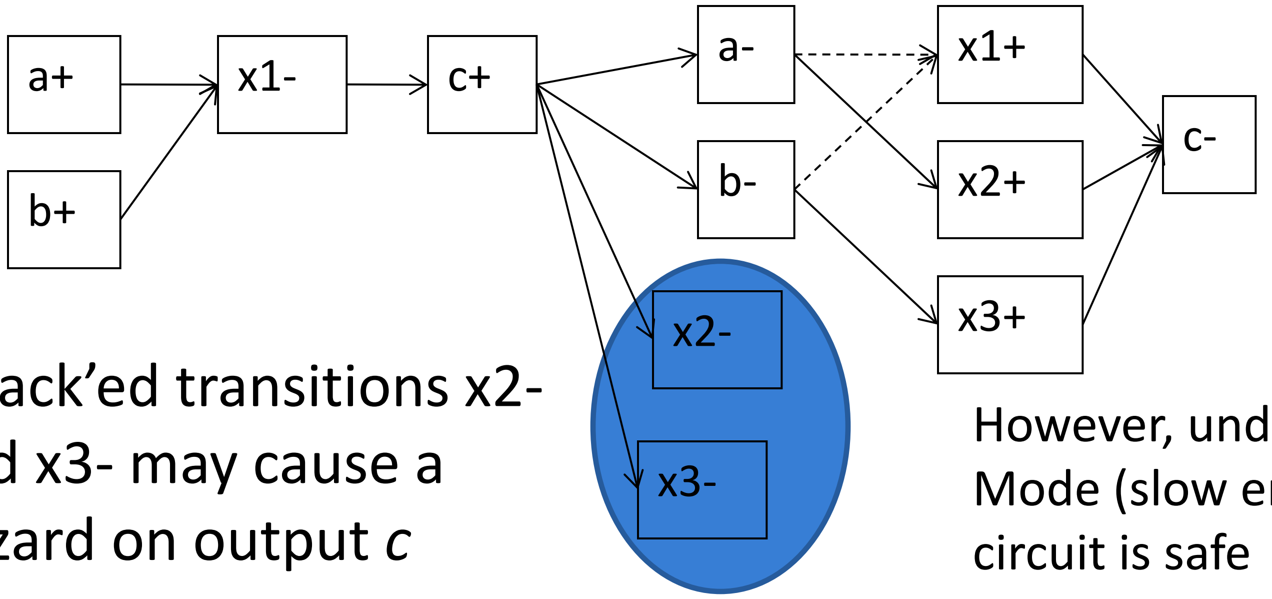
**Every signal event must be acknowledged by another event**

# Causal acknowledgment





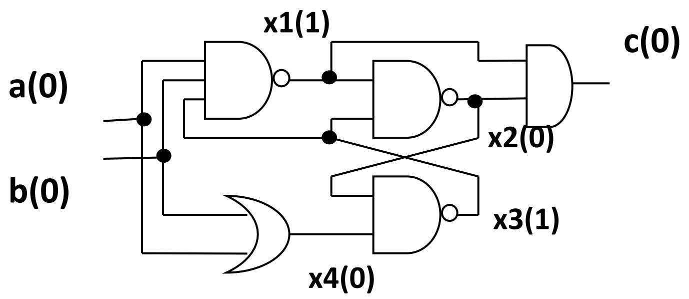
C-element implementation using simple gates

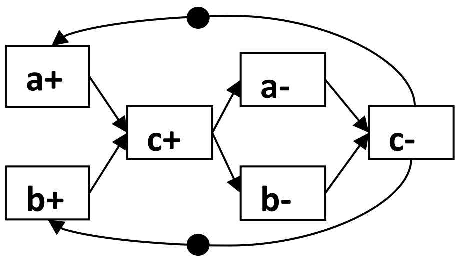


Unack'ed transitions x2- and x3- may cause a hazard on output *c*

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

# Principle of causal acknowledgement





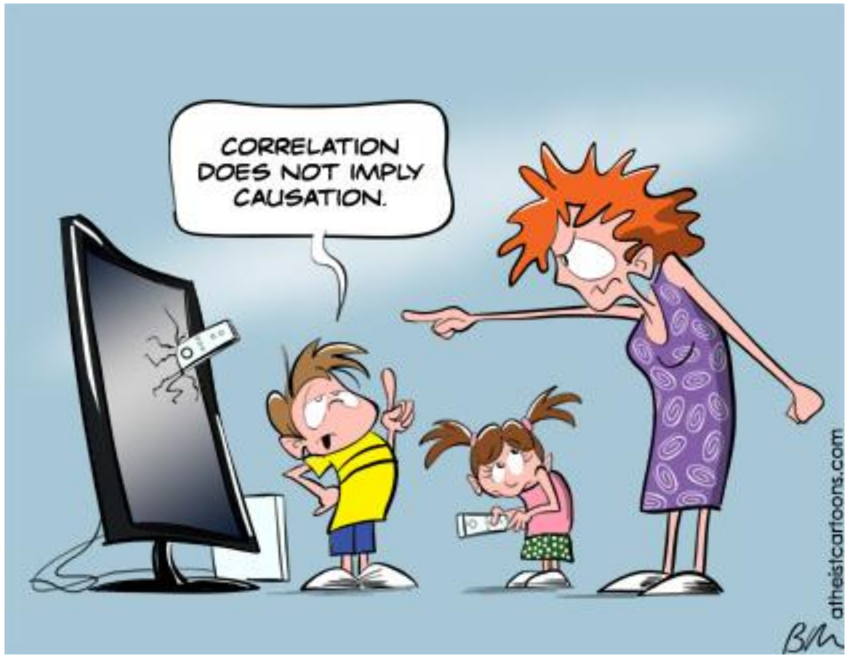
C-element implementation using simple gates

$$\begin{tikzcd}
\boxed{a+} \arrow[r] \arrow[rdd, dashed] & \boxed{x1-} \arrow[r] & \boxed{x2+} \arrow[rd] & & & \\
& & & \boxed{x3-} \arrow[r] & \boxed{x1+} \arrow[r] & \boxed{c+} \\
\boxed{b+} \arrow[ruu] \arrow[r, dashed] & \boxed{x4+} \arrow[rru] & & & &
\end{tikzcd}$$

$$\begin{tikzcd}
\boxed{a-} \arrow[rd] & & & & \\
& \boxed{x4-} \arrow[r] & \boxed{x3+} \arrow[r] & \boxed{x2-} \arrow[r] & \boxed{c-} \\
\boxed{b-} \arrow[ru] & & & &
\end{tikzcd}$$

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

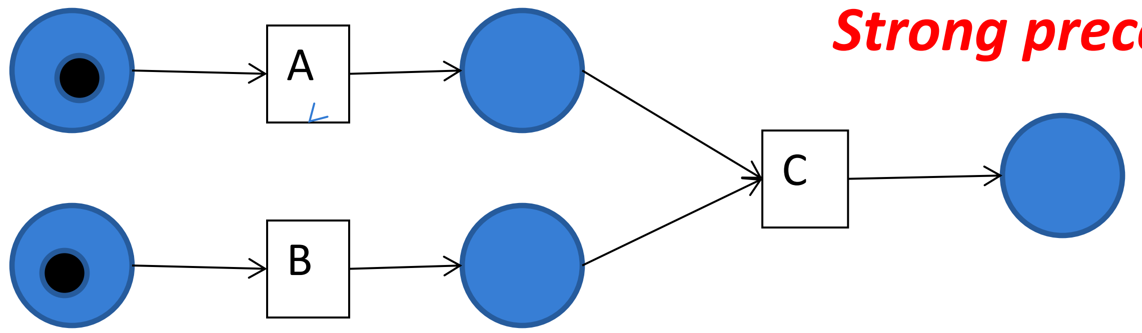
## Why and what are strong and weak causality ?



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

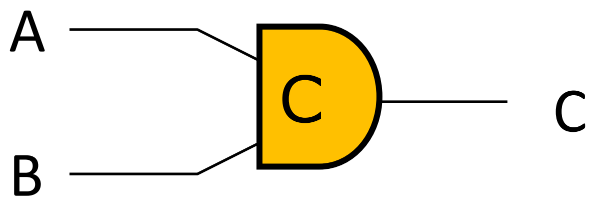
# Strong Causality

- **Petri net transitions synchronising as rendez-vous**



***Strong precedence***

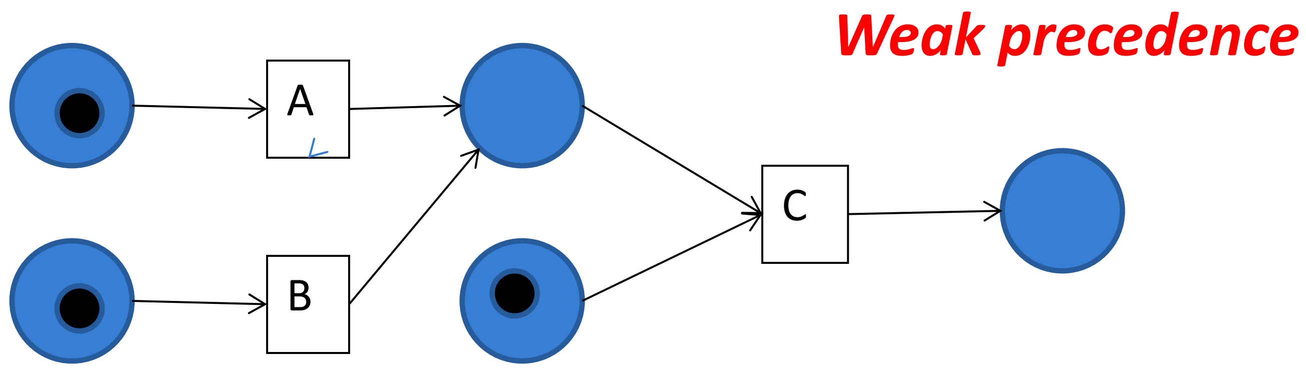
- **Logic circuits: Muller C-element (in 0-1 and 1-0 transitions), AND gate (in 0-1 transitions), OR gate (in 1-0 transitions)**



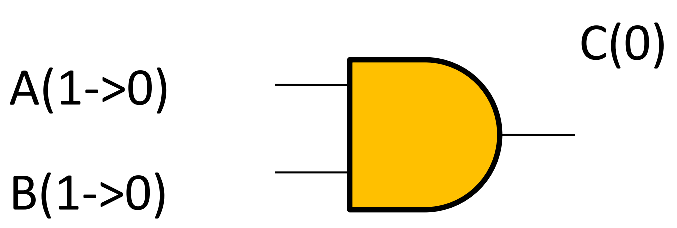
| A | B | $C^+$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | C |
| 1 | 0 | C |
| 1 | 1 | 1 |

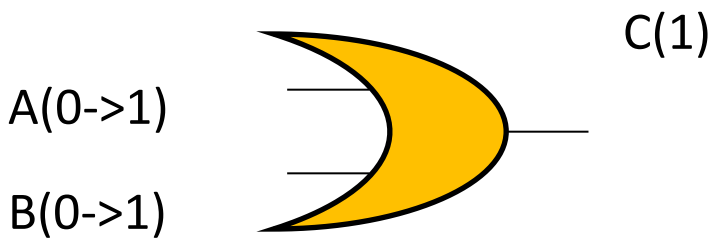
# Weak Causality

- **Petri net transitions communicating via places**

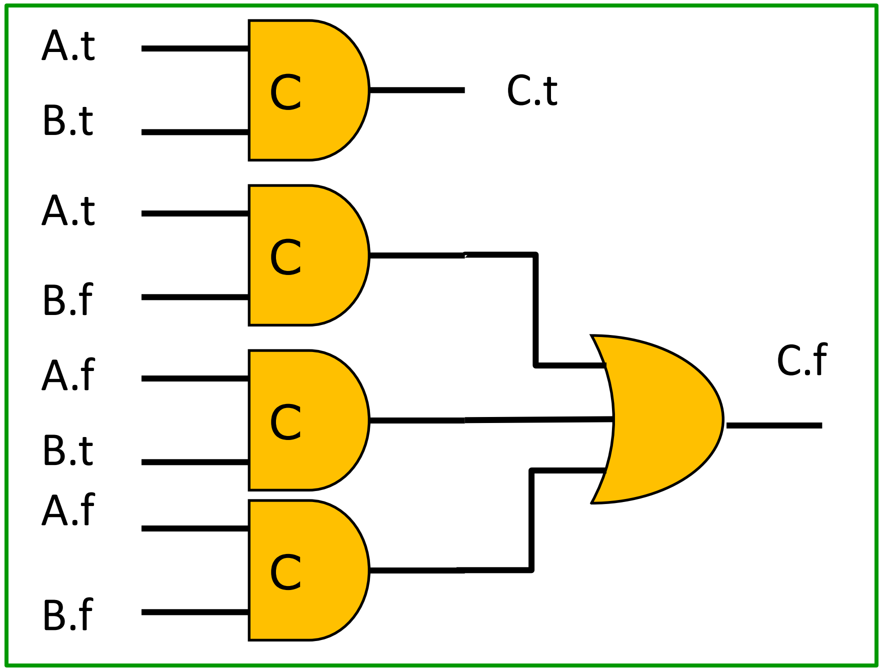


- **Logic circuits: AND gate (in 1-0 transitions), OR gate (in 0-1 transitions)**

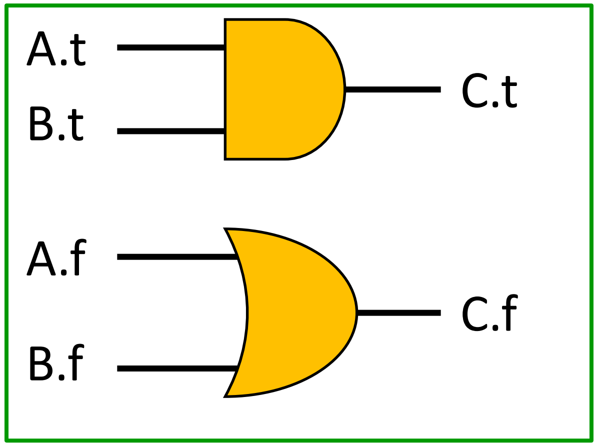




# Full indication versus Early Evaluation



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)**

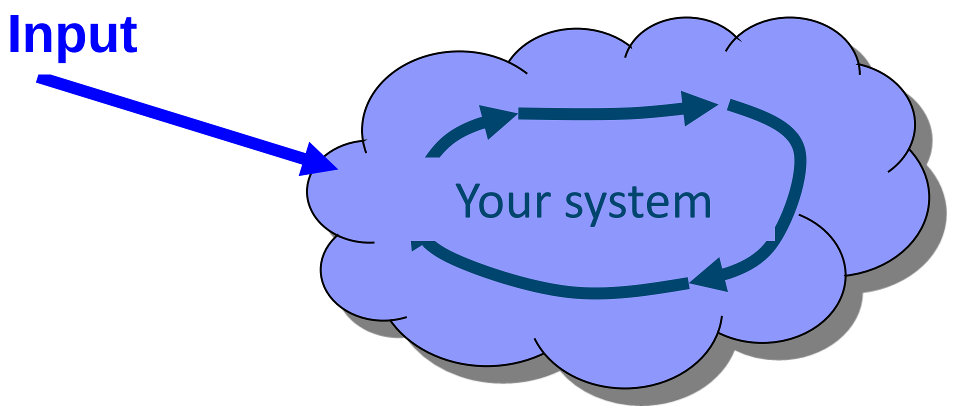
## Why and what is timing comparison?

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

## Synchronizers and arbiters

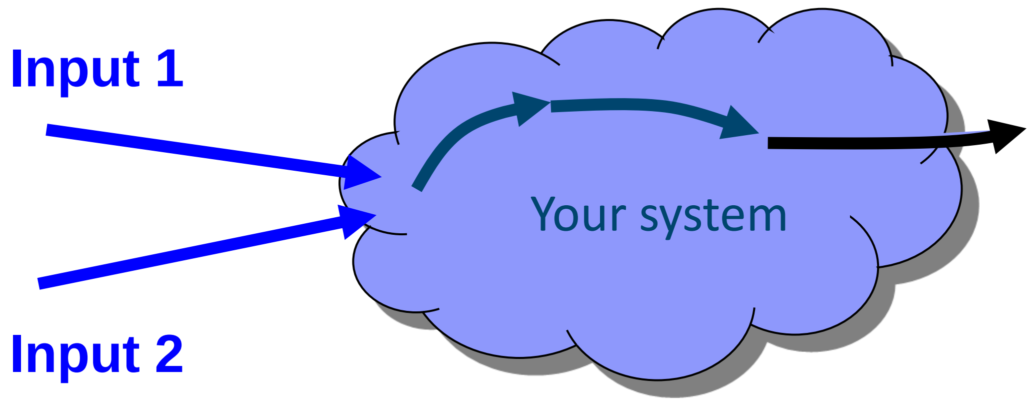
- Synchronizer

Decides which clock cycle to use for the input data

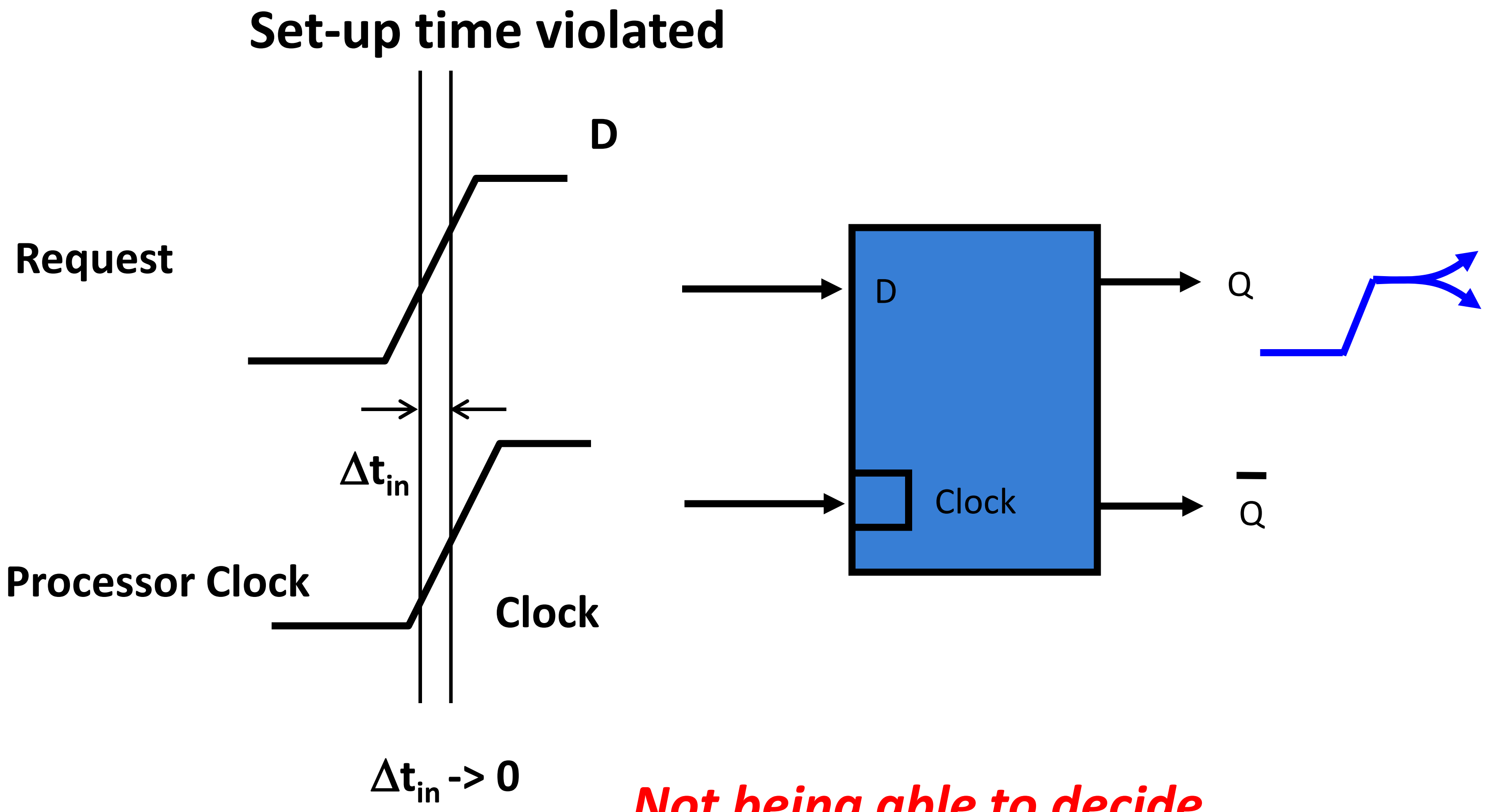


- Asynchronous arbiter

Decides the order of inputs

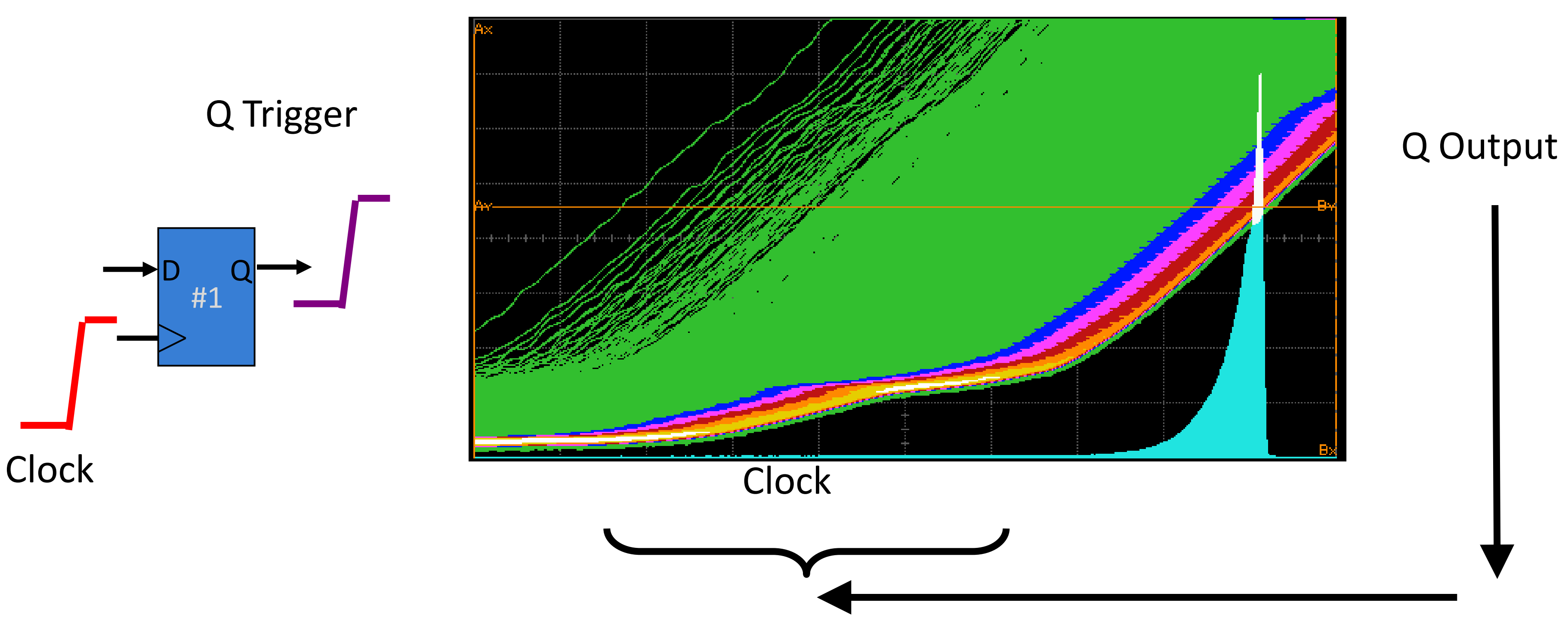


# Metastability is....



***Not being able to decide…***

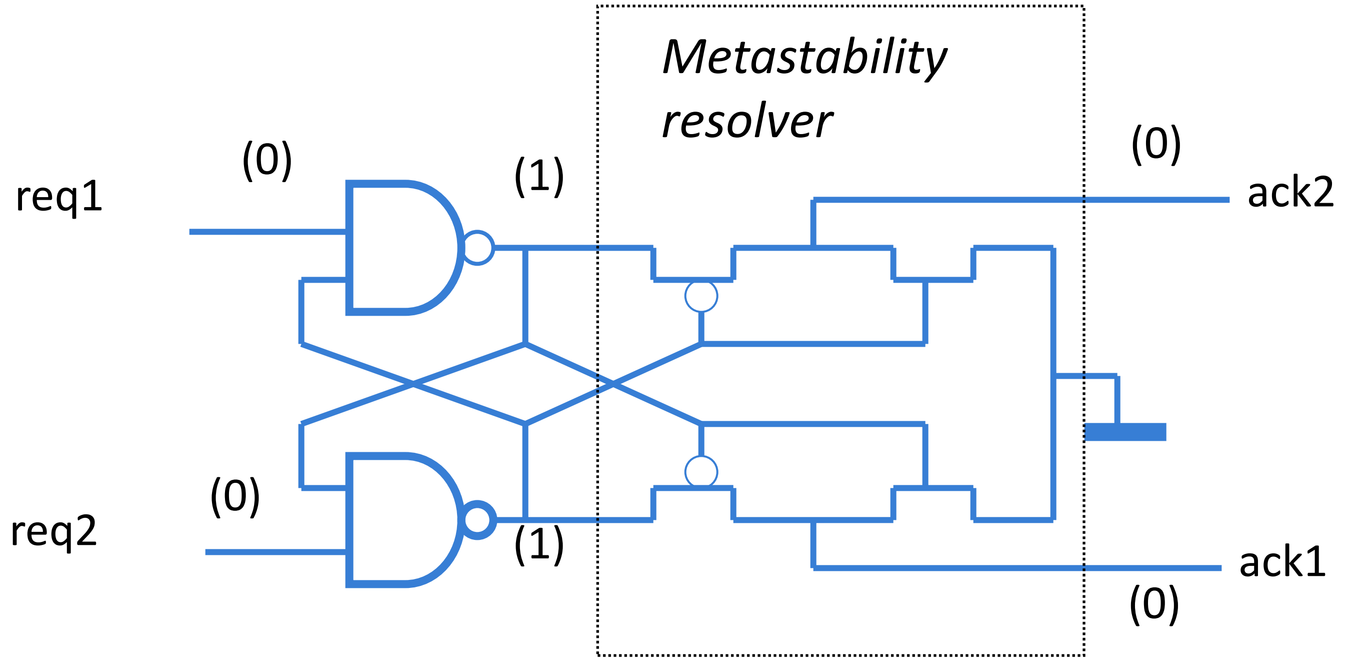
# Typical responses



- 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

## 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**

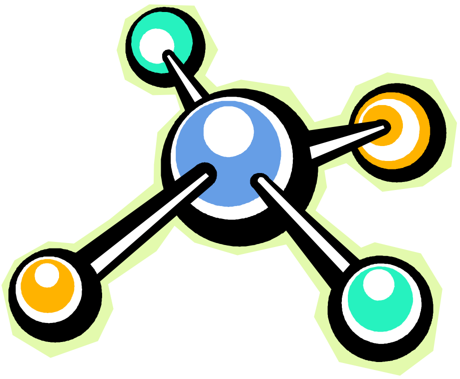
## Pros…

- 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 Idd (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

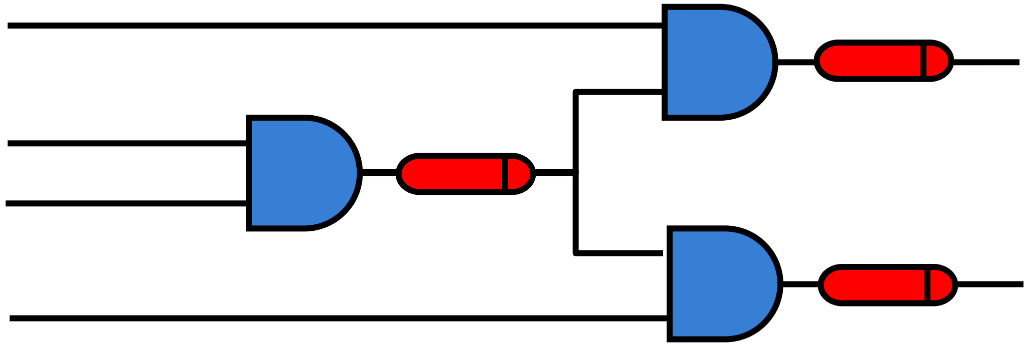


## Models and techniques for asynchronous design

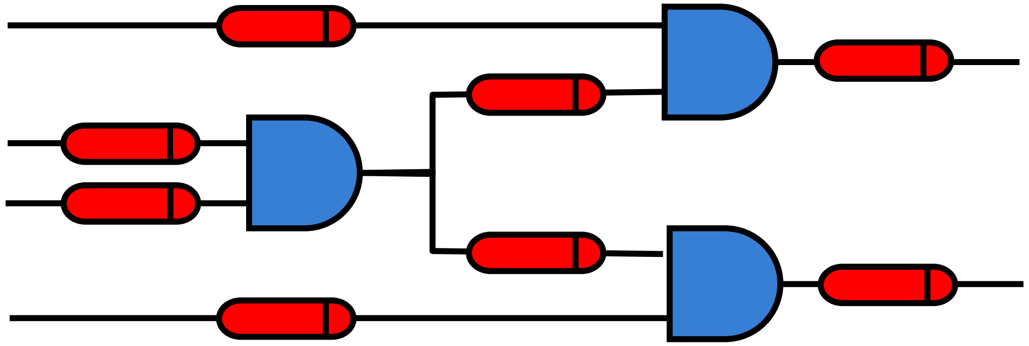
- Models:
  - Delay model (inertial, pure, gate delay, wire delay, bounded and unbounded delays)
  - Models of environment (fundamental mode, input-output)
  - Models of switching behaviour (state-based, event-based, hybrid)
- RTL level:
  - Data and control paths separate (data flow graphs, FSMs, Signal 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

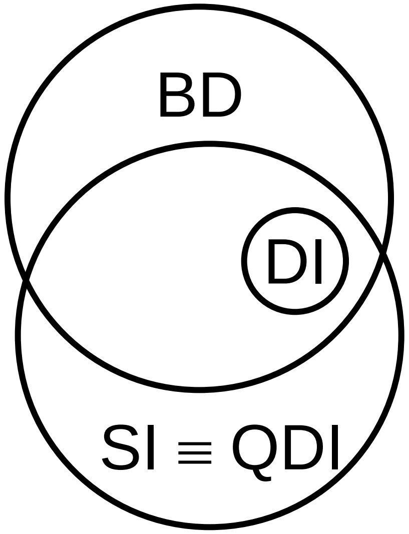


- Wire delay model: delays in gates and wires



## 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



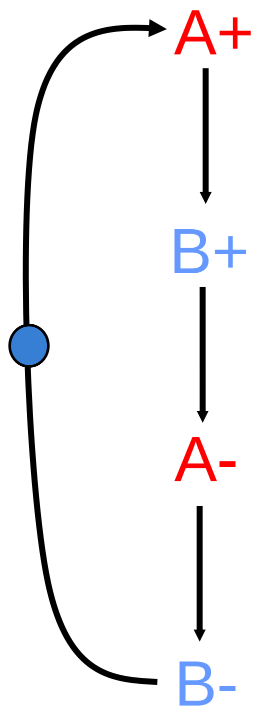
## Control Logic

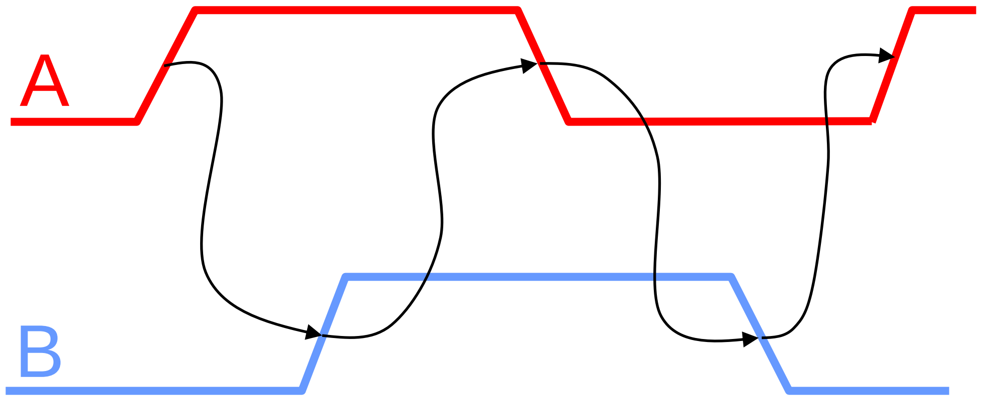
- **Control specification based on Petri nets (Signal Transition graphs)**

# Control specification

Signal Transition Graph (STG)

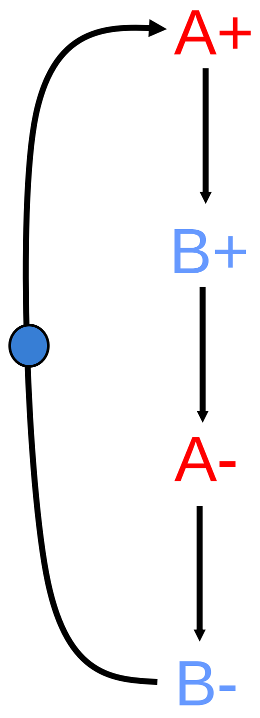
Timing Diagram

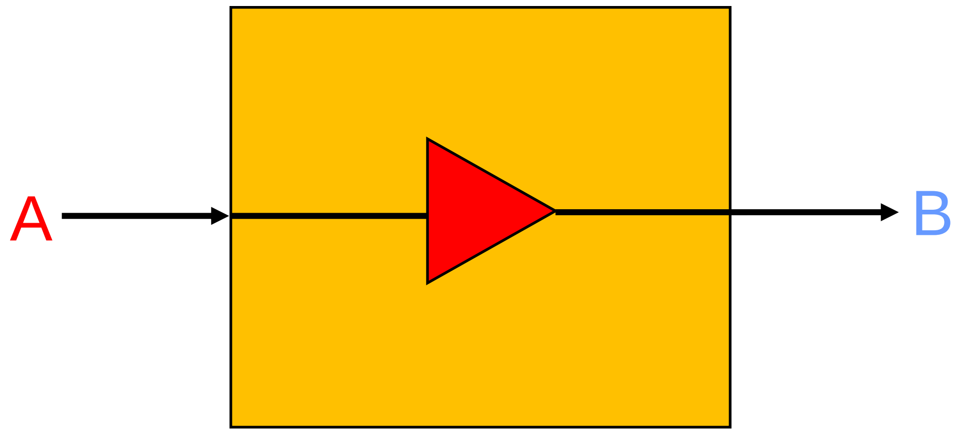

$$\begin{tikzcd} A+ \arrow[d] \\ B+ \arrow[d] \\ A- \arrow[d] \\ B- \arrow[uuu, bend left, "\bullet"] \end{tikzcd}$$



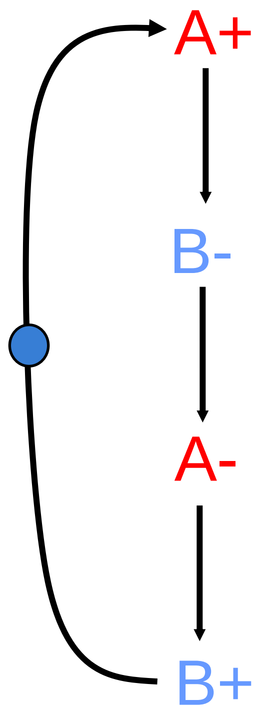
A input
B output

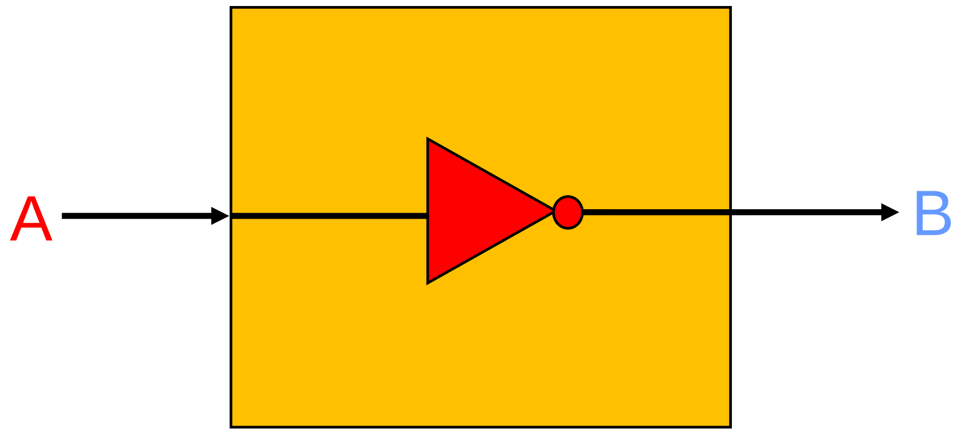
## Control specification



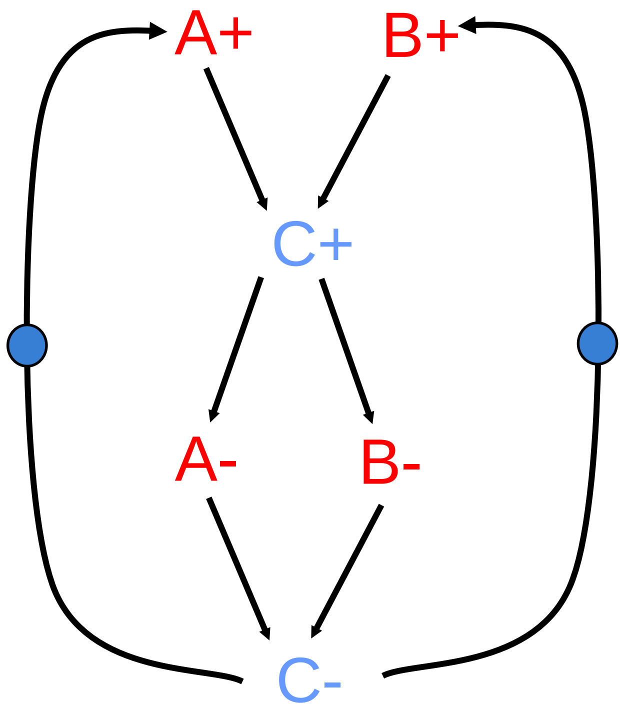


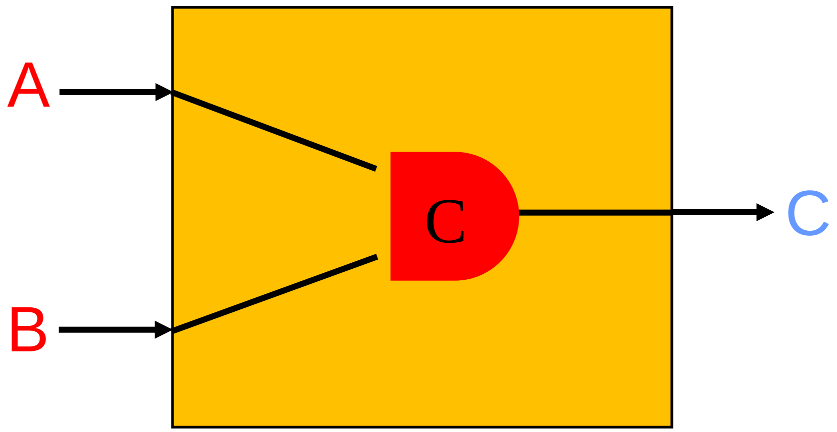
# Control specification



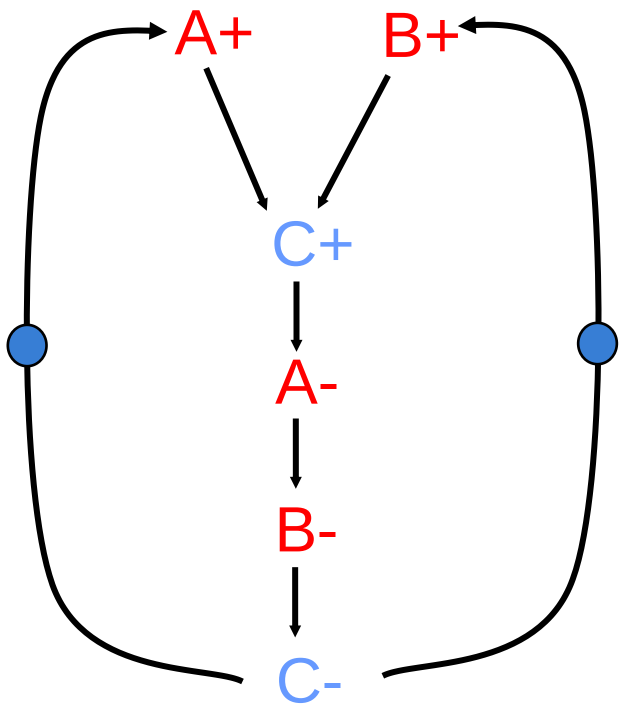


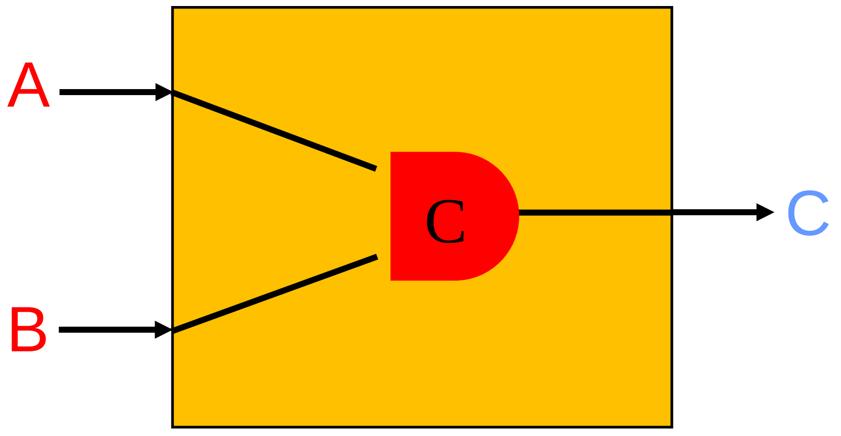
# Control specification



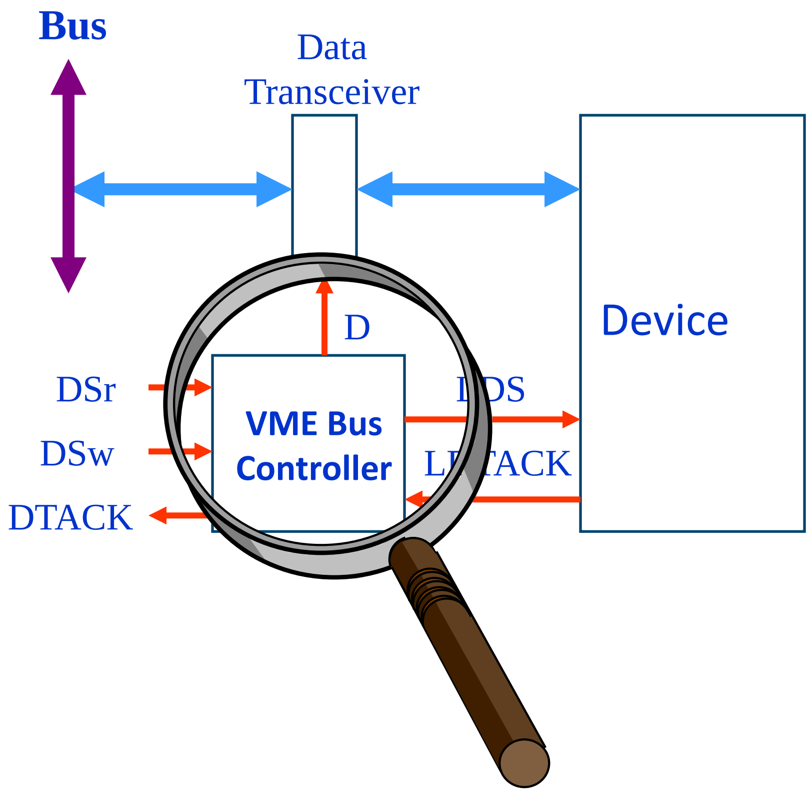


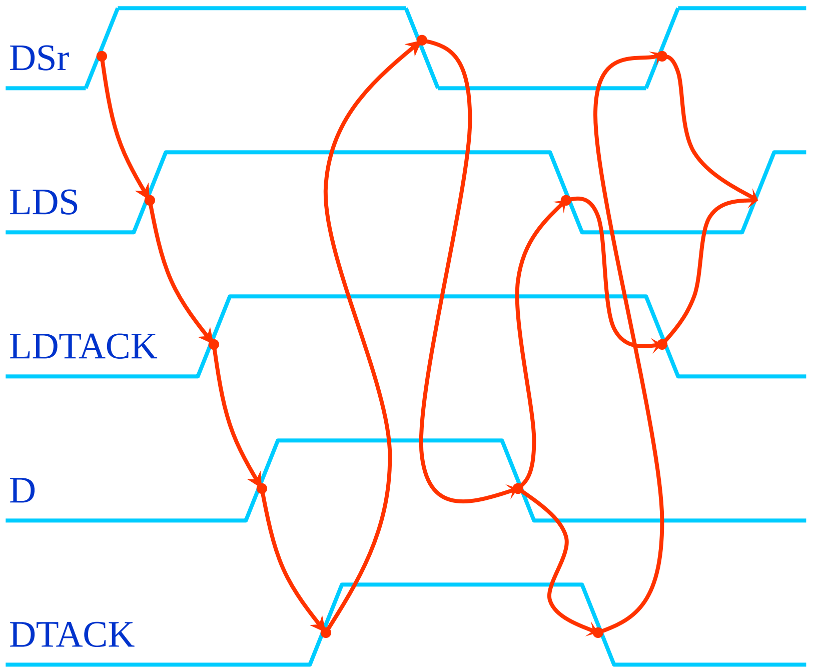
# Control specification





# VME bus example using Petri nets

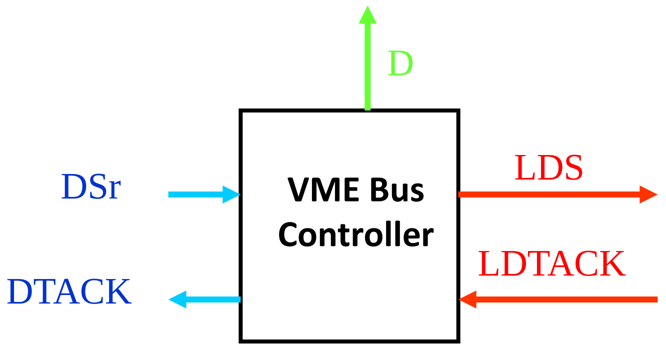




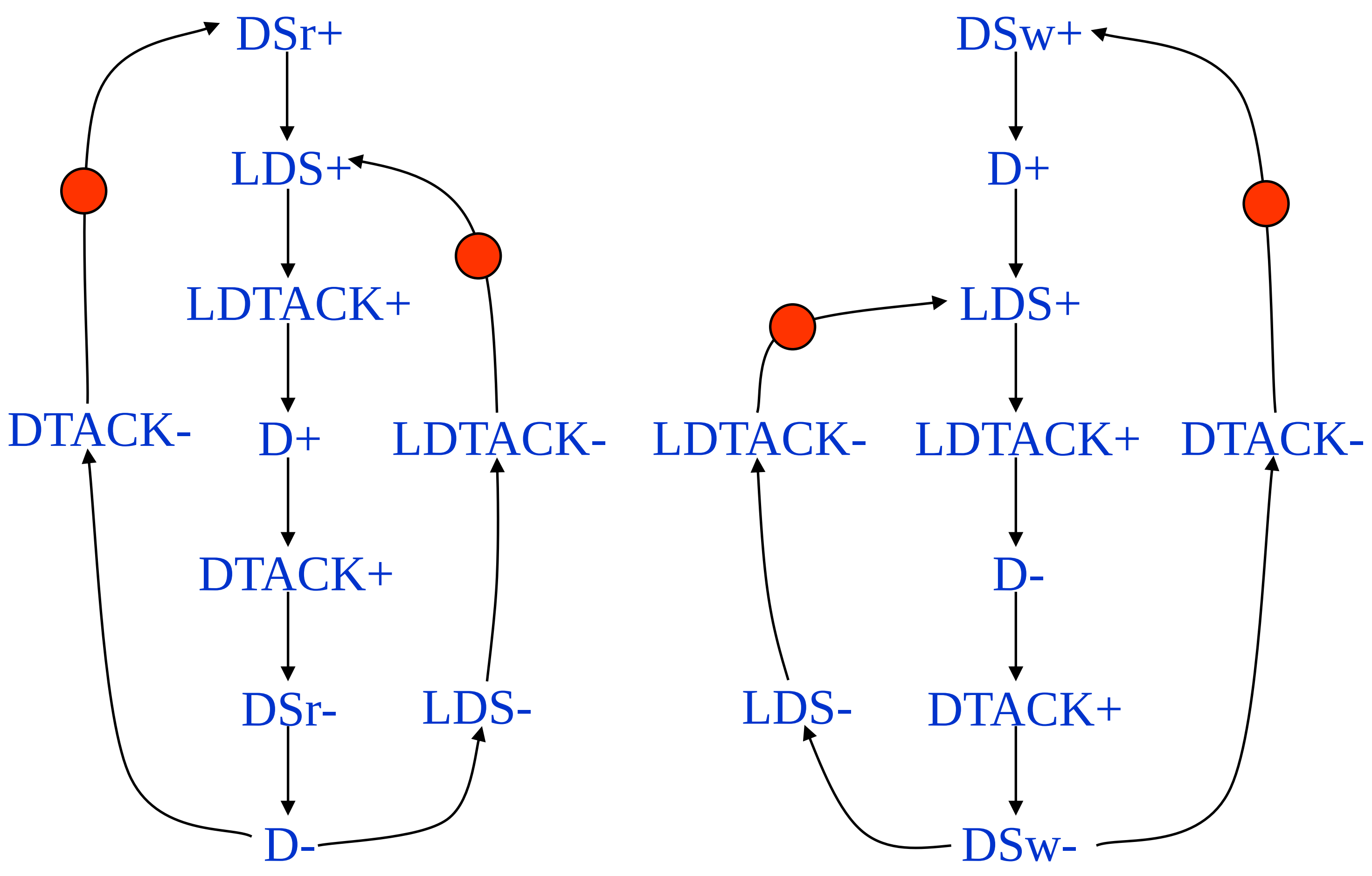
**Read Cycle**

# STG for the READ cycle

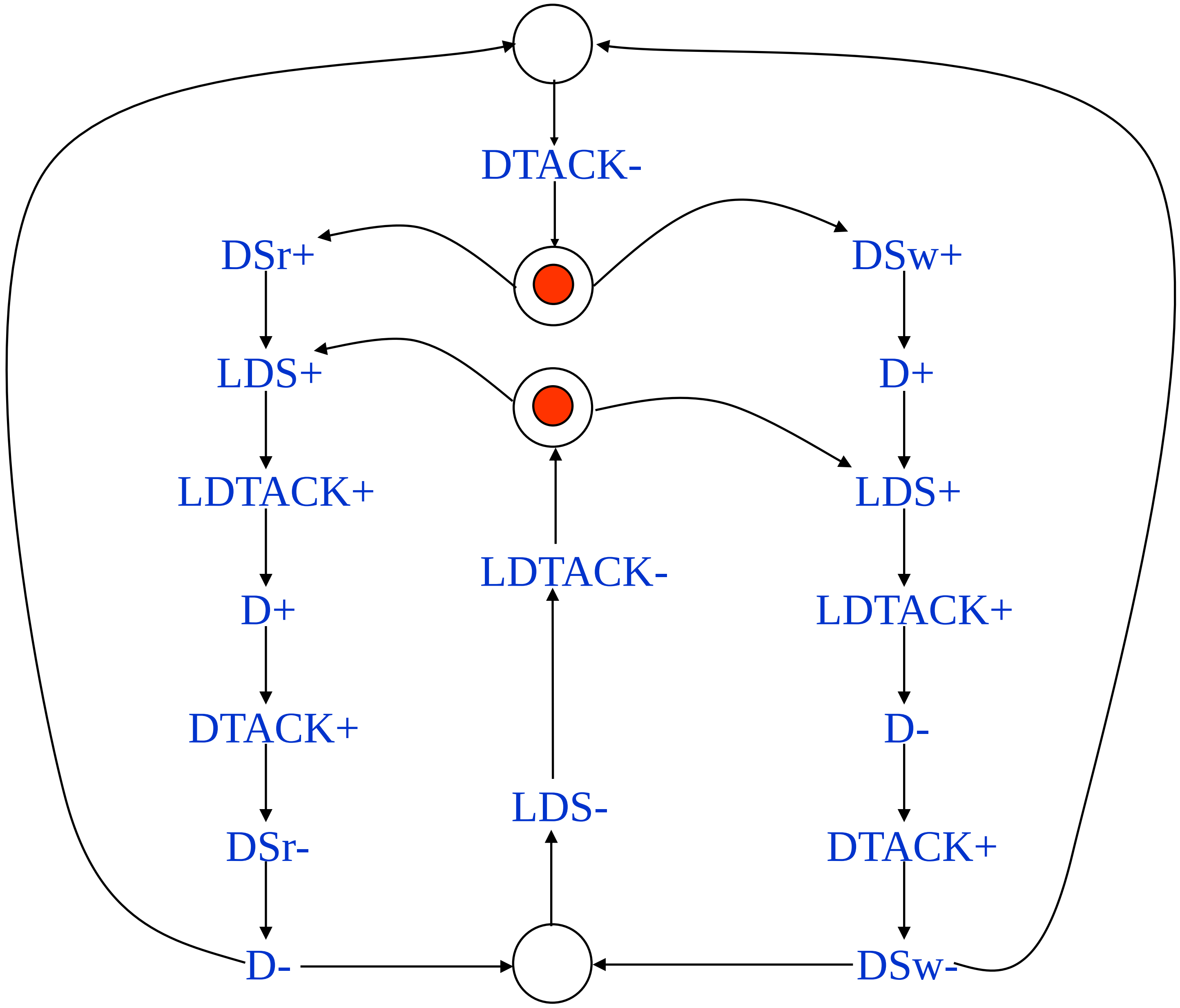
$$\begin{tikzcd}
 & \text{DSr+} \arrow[dl] & & \text{DTACK-} \arrow[ll, "\bullet"] & & \\
\text{LDS+} \arrow[r] & \text{LDTACK+} \arrow[r] & \text{D+} \arrow[r] & \text{DTACK+} \arrow[r] & \text{DSr-} \arrow[r] & \text{D-} \arrow[ull] \arrow[dll] \\
 & \text{LDTACK-} \arrow[ul, "\bullet"] & & \text{LDS-} \arrow[ll] & &
\end{tikzcd}$$



# Choice: Read and Write cycles



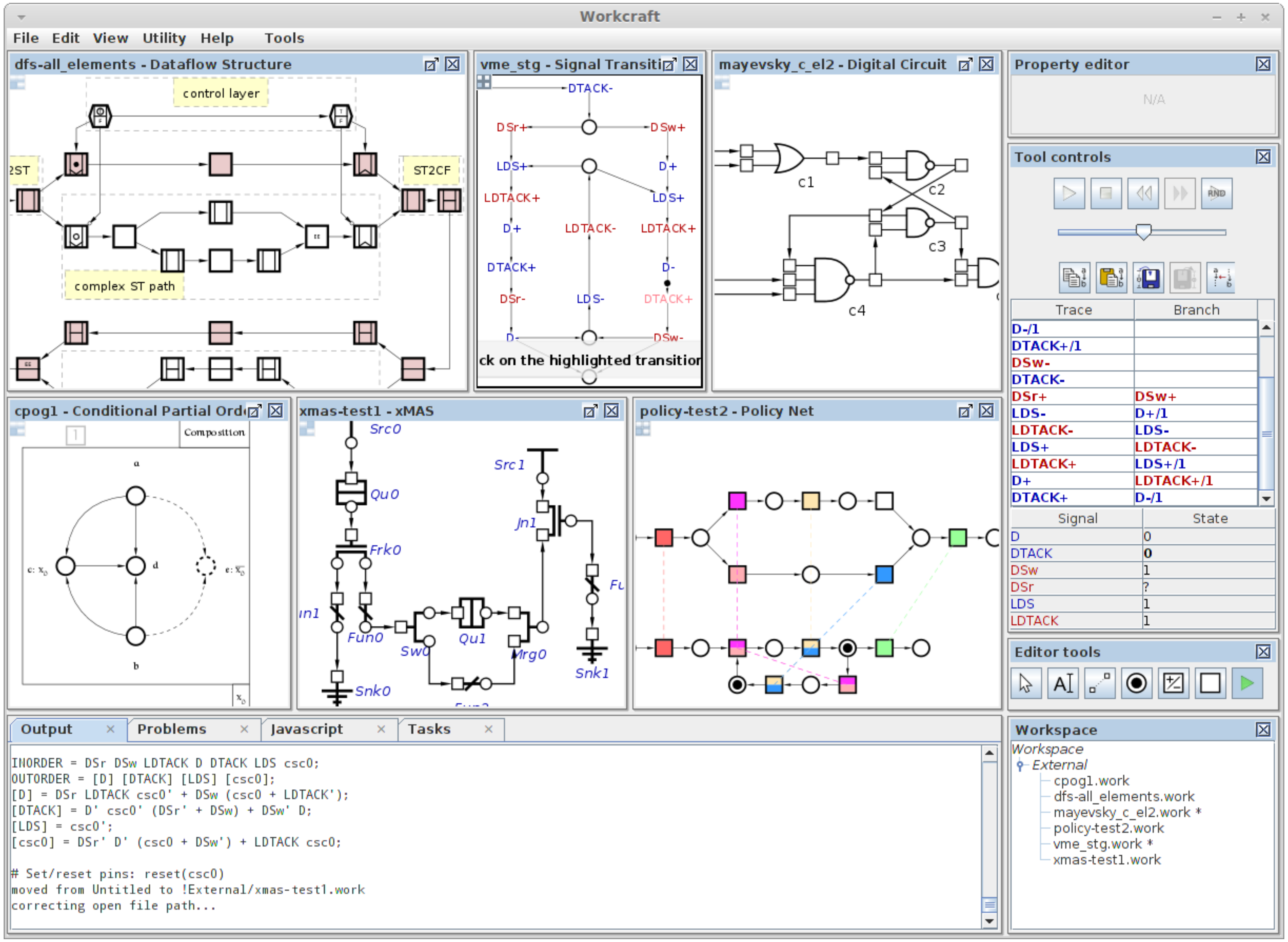
## Choice: Read and Write cycles



## Workcraft tool: workcraft.org

- Framework for interpreted graph models
  - Circuits, STGs, state graphs, dataflow structures, …
  - Interoperability between different abstraction levels
  - Consistency for users; convenience for developers
- Elaborate graphical user interface
  - Visual editing, analysis, and simulation
  - Easy access to common operations
  - Possibility to script specialised actions
- Interface to back-end tools for synthesis and verification
  - Reuse of established theory and tools (PETRIFY , MPSAT , PUNF )

# Synthesis & verification of async circuits



| Trace | Branch |
|---|---|
| D-/1 | |
| DTACK+/1 | |
| DSw- | |
| DTACK- | |
| DSr+ | DSw+ |
| LDS- | D+/1 |
| LDTACK- | LDS- |
| LDS+ | LDTACK- |
| LDTACK+ | LDS+/1 |
| D+ | LDTACK+/1 |
| DTACK+ | D-/1 |

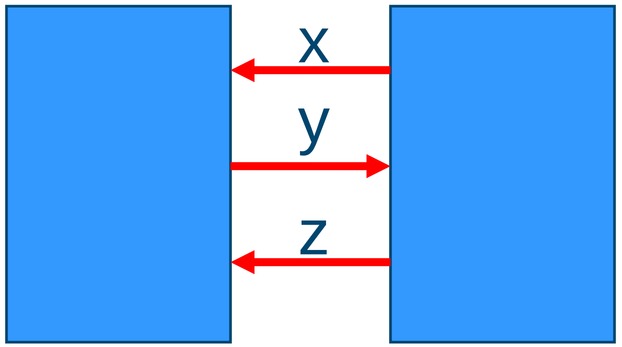
| Signal | State |
|---|---|
| D | 0 |
| DTACK | 0 |
| DSw | 1 |
| DSr | ? |
| LDS | 1 |
| LDTACK | 1 |

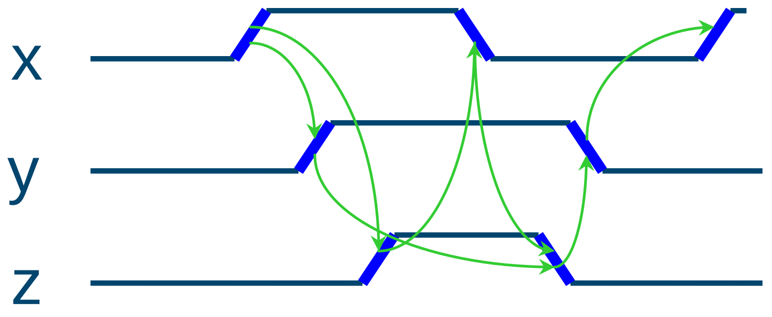
```
INORDER = DSr DSw LDTACK D DTACK LDS csc0;
OUTORDER = [D] [DTACK] [LDS] [csc0];
[D] = DSr LDTACK csc0' + DSw (csc0 + LDTACK');
[DTACK] = D' csc0' (DSr' + DSw) + DSw' D;
[LDS] = csc0';
[csc0] = DSr' D' (csc0 + DSw') + LDTACK csc0;

# Set/reset pins: reset(csc0)
moved from Untitled to !External/xmas-test1.work
correcting open file path...
```

Workspace
- External
  - cpog1.work
  - dfs-all_elements.work
  - mayevsky_c_el2.work *
  - policy-test2.work
  - vme_stg.work *
  - xmas-test1.work

# Logic synthesis: xyz-example

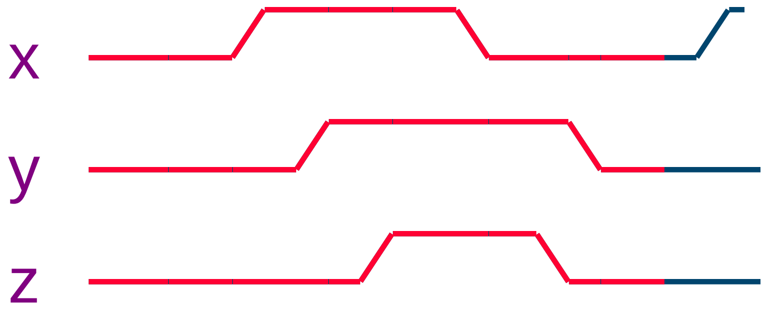


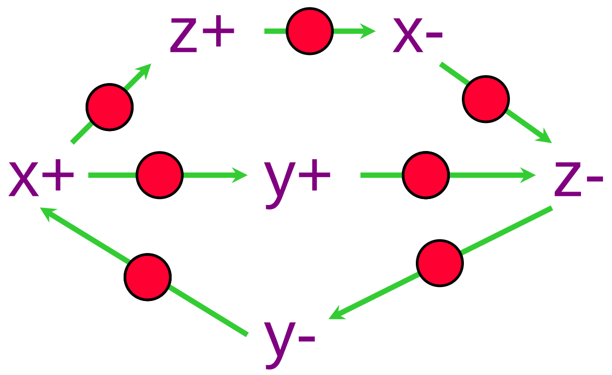


$$\begin{tikzcd}
 & z+ \arrow[r] & x- \arrow[dr] & \\
x+ \arrow[ur] \arrow[r] & y+ \arrow[rr] & & z- \arrow[dll] \\
 & y- \arrow[ul] & &
\end{tikzcd}$$

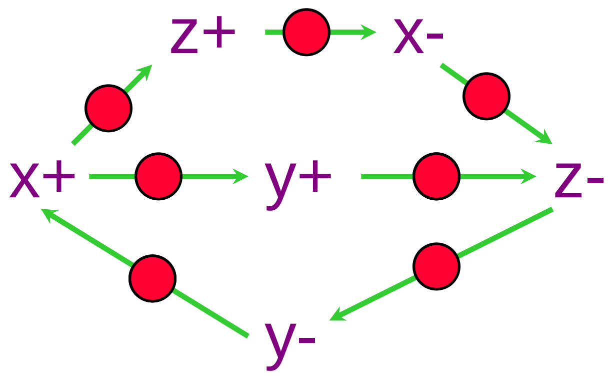
***Signal Transition Graph (STG)***

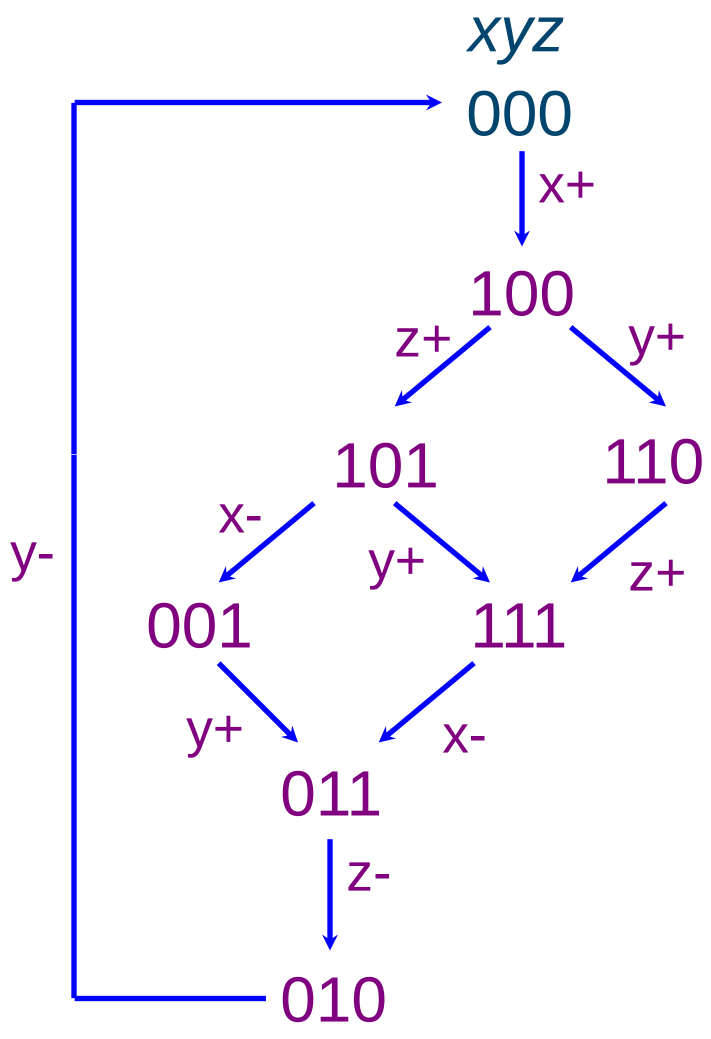
## Token flow





# State graph





# Next-state functions

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

$$y = z + x$$

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

$$\begin{tikzcd}
 & & xyz \\
 & & 000 \arrow[d, "x+"] \\
 & & 100 \arrow[dl, "z+"'] \arrow[dr, "y+"] \\
 & 101 \arrow[dl, "x-"'] \arrow[dr, "y+"] & & 110 \arrow[dl, "z+"] \\
001 \arrow[dr, "y+"'] & & 111 \arrow[dl, "x-"] \\
 & 011 \arrow[d, "z-"] \\
 & 010 \arrow[uuuuur, bend left=60, "y-"]
\end{tikzcd}$$

## 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**

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

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

Observations in this example:

**1) All combinations are used as states**

**2) All states appear uniquely**

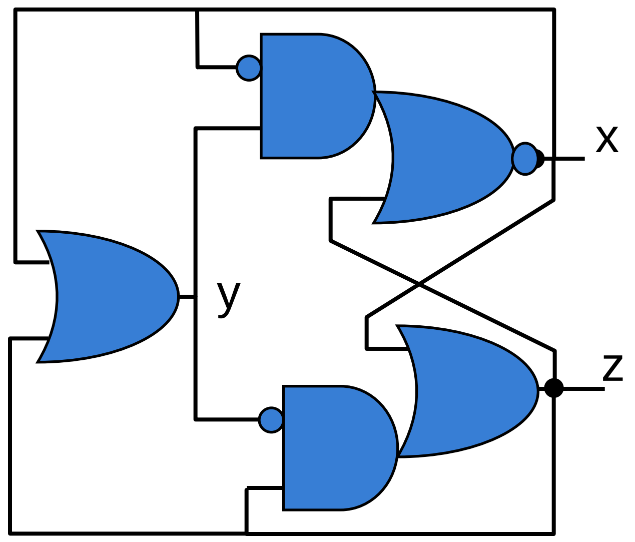
**Generally, this is not always the case!**

# 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

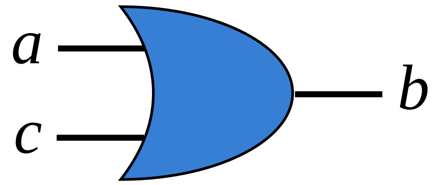
$$\Downarrow$$

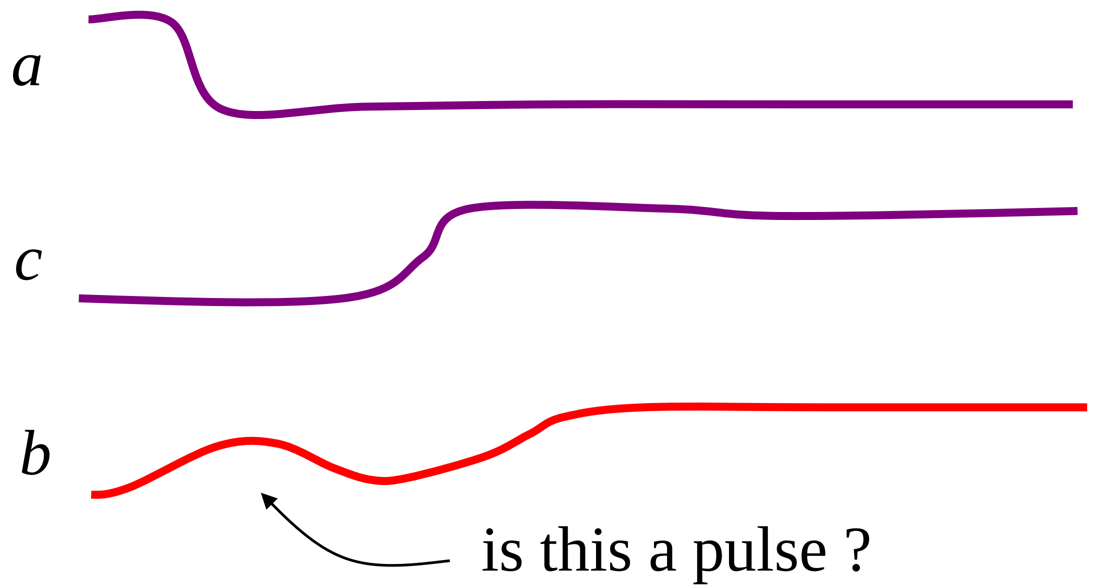
- There exists a speed-independent circuit that implements the behavior of the STG

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

# Persistency

$$\begin{tikzcd}
100 \arrow[r, "a-"] \arrow[d, "\color{red}{b+}"] & 000 \arrow[r, "c+"] & 001 \arrow[d, "\color{red}{b+}"] \\
{} & & {}
\end{tikzcd}$$





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

## Conclusion

- Asynchronous design is based on rigorous principles of causality and concurrency
- It allows electronic circuits to operate without centralised time constraints and adapt to any structural or behavioural variations and noise
- Nature is largely asynchronous as it is typically structured around energy supplies; synchronisation takes place in a distributed way
- Research question: Are reaction systems asynchronous? And in what way they are or can be asynchronous?
- Attend the workshop on "Bringing Asynchrony to Reaction Systems"!
- I encourage development of a Reaction Systems modelling plug-in under Workcraft.org