A4A: Asynchronous design for Analogue electronics

Victor Khomenko, Andrey Mokhov,
Danil Sokolov, Alex Yakovlev

Newcastle University, UK
async.org.uk

Outline

- Key Asynchronous Design Principles
- (Some of the) Models for Asynchronous Design
- Asynchronous control logic synthesis from Signal Transition Graphs
- Complete state coding (CSC) resolution
- Formal verification of asynchronous circuits
- Under the Bonnet of Workcraft tools
- Advanced Topic: Logic Synthesis and Implementation Styles in Asynchronous Circuits Design

Asynchronous Behaviour

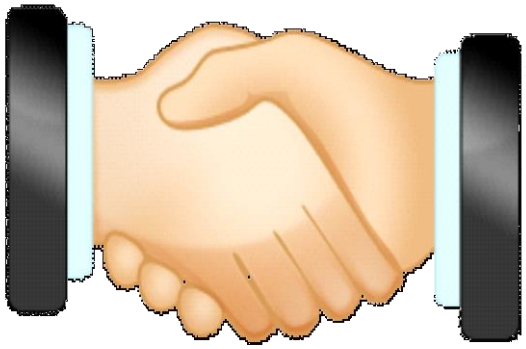
- Synchronous vs Asynchronous behaviour in general terms, examples:
  - Orchestra playing with vs without a conductor
  - Party of people having a set menu vs a la carte
- Synchronous means all parts of the system acting globally in tact, even if some or all part ‘do nothing’
- Asynchronous means parts 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 Circuit Design

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)

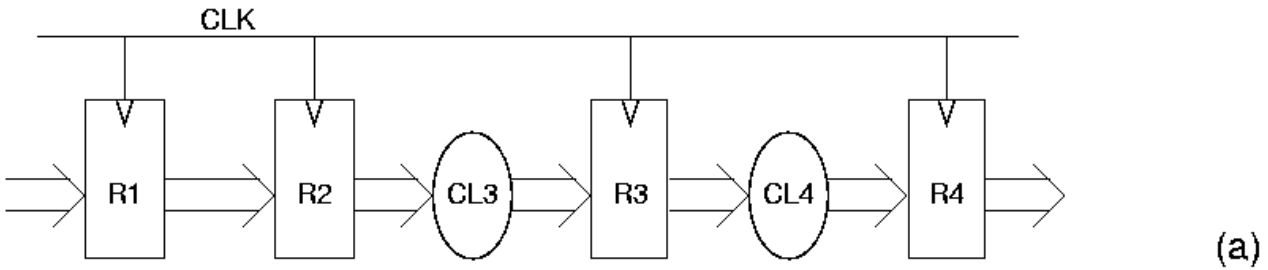
Why and what is handshaking?



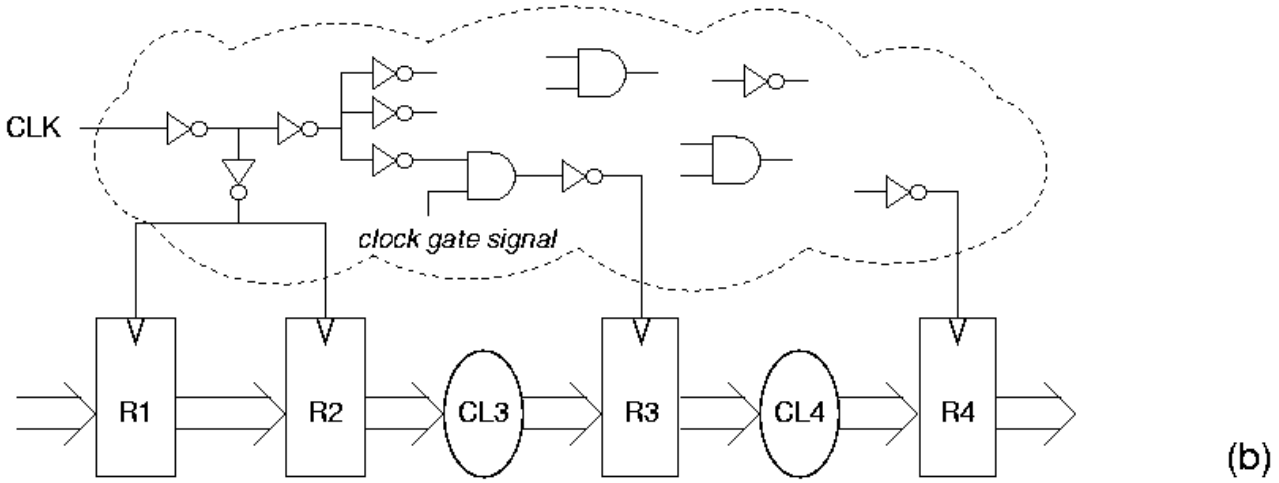
Mutual Synchronisation is via Handshake

Synchronous clocking

How we think

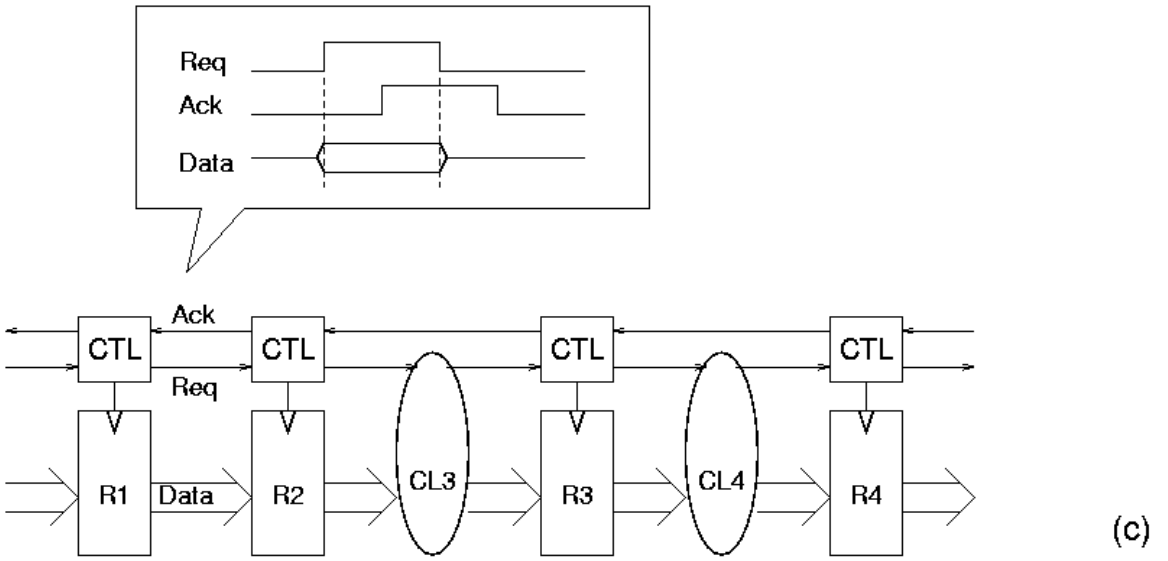


What we design

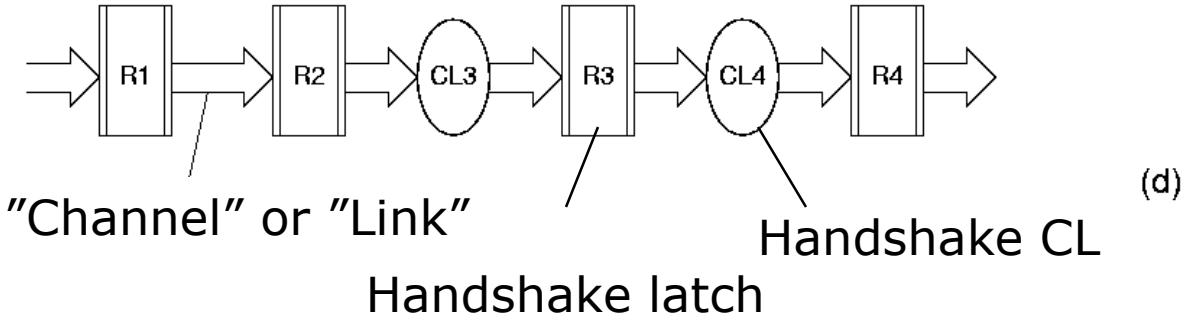


Asynchronous handshaking

What we design

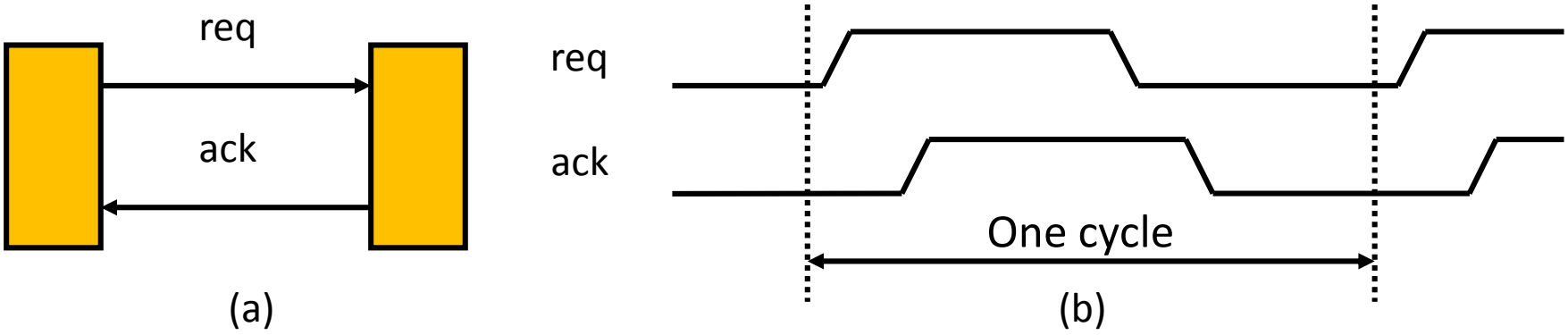


How we think

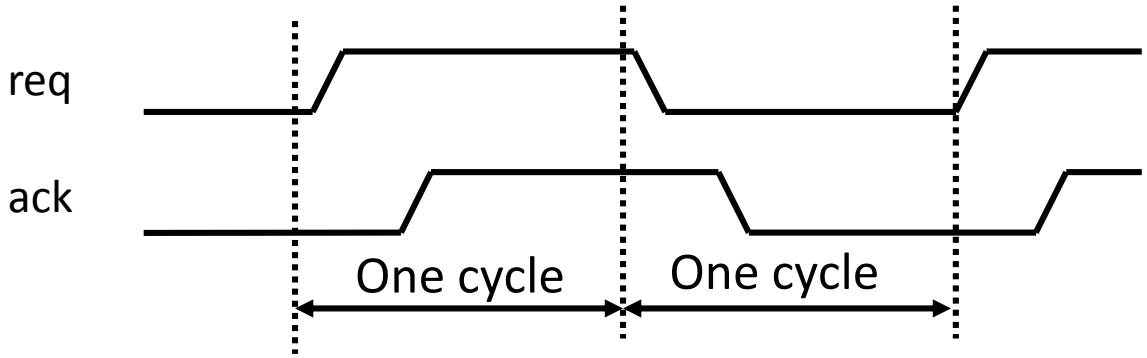


Handshake Signalling Protocols

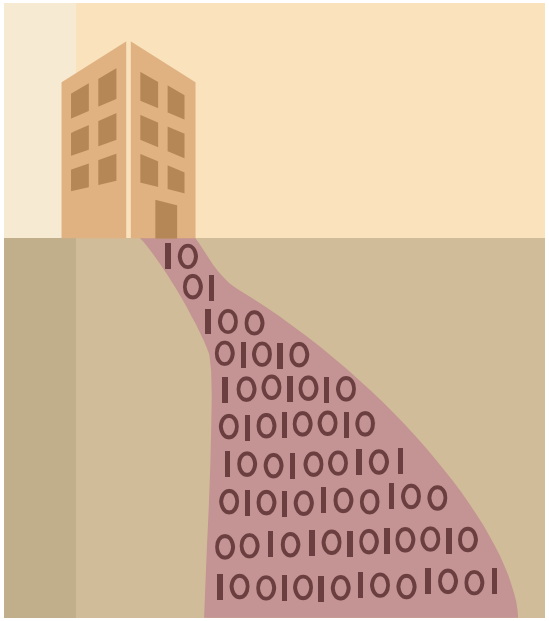
Level Signalling (RTZ or 4-phase)



Transition Signalling (NRZ or 2-phase)

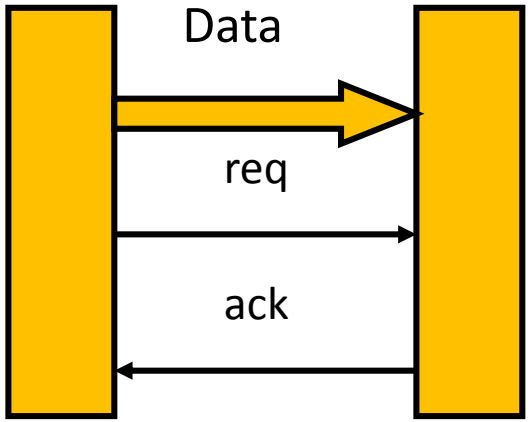


Why and what is delay-insensitive coding?

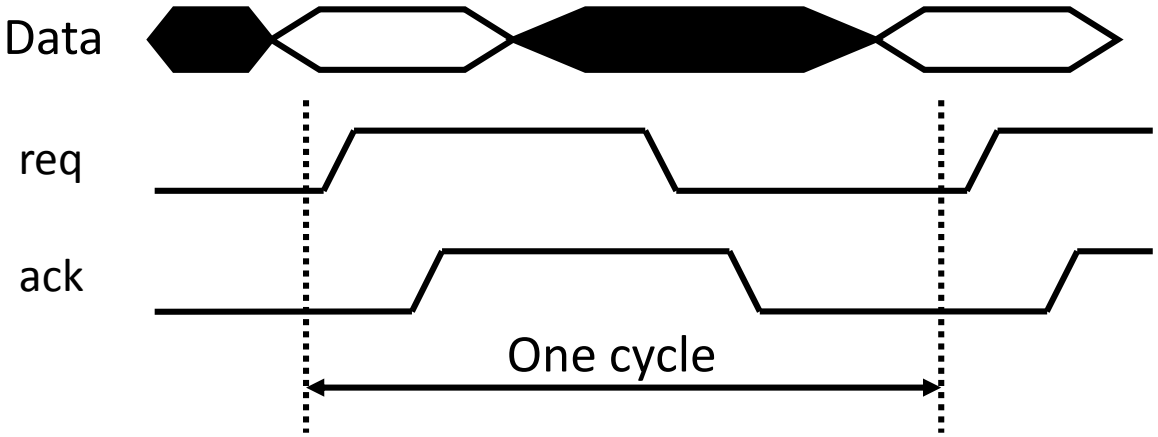


Data Token = (Data Value, Validity Flag)

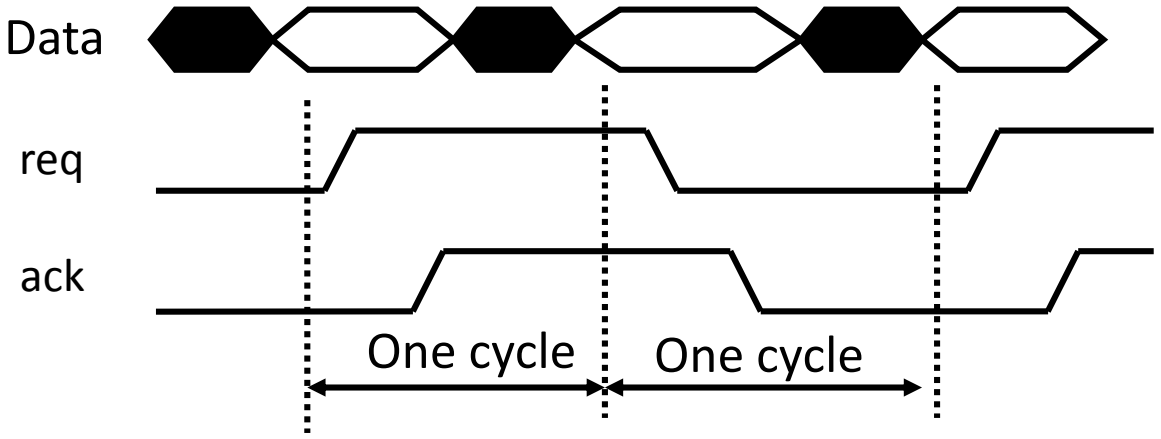
Bundled Data



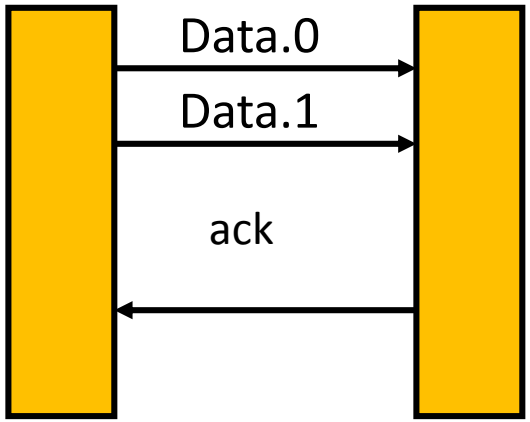
Return to Zero:



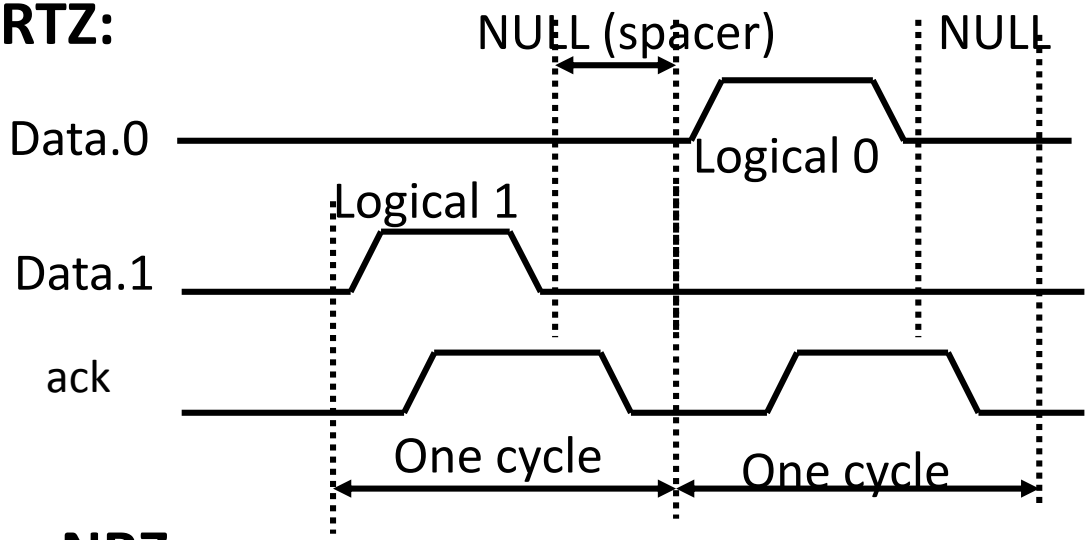
Non-Return-to-Zero



DI encoded data (Dual-Rail)

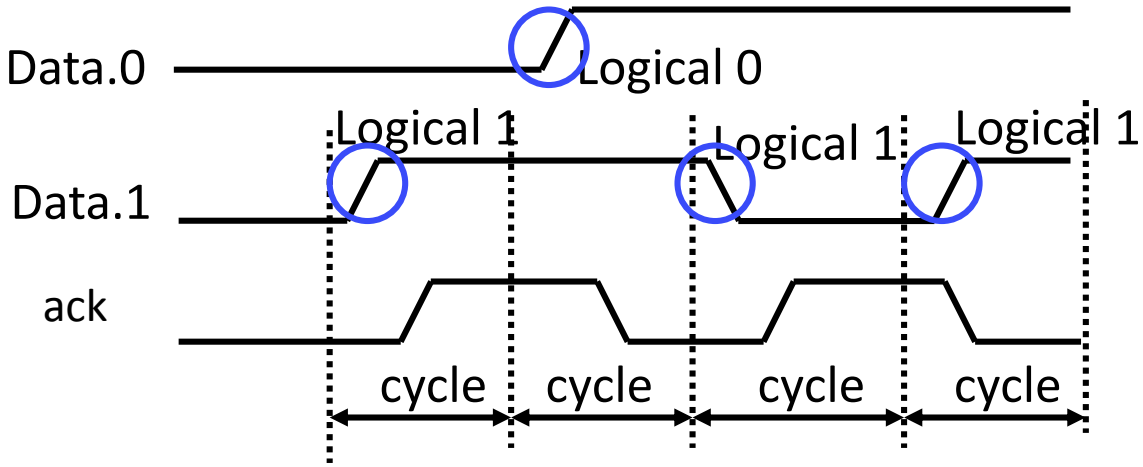


RTZ:



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

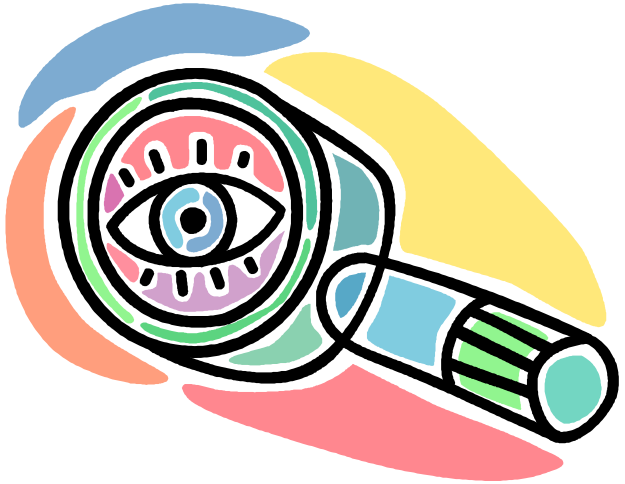
NRZ:



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

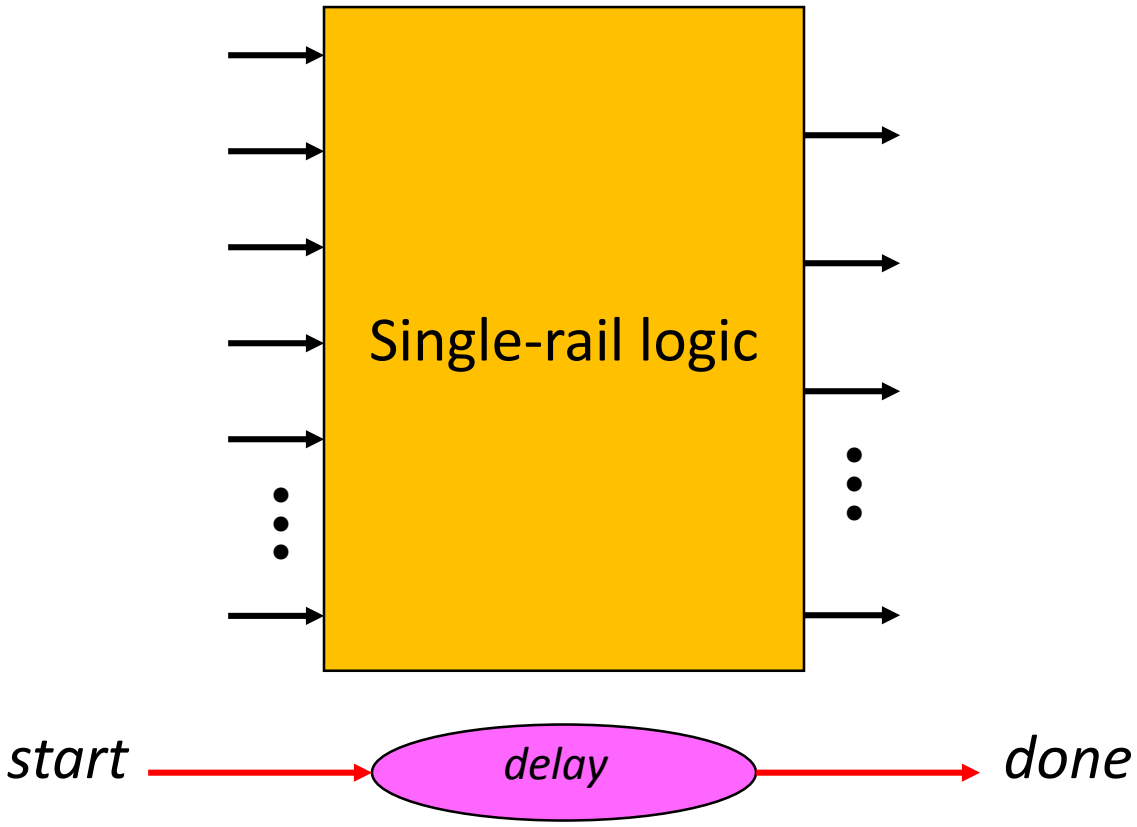
- **1-of-4:**
  - **0001=> 00, 0010=>01, 0100=>10, 1000=>11**
- **2-of-4:**
  - **1100, 1010, 1001, 0110, 0101, 0011 – total 6 combinations (cf. 2-bit dual-rail – 4 comb.)**
- **3-of-6:**
  - **111000, 110100, …, 000111 – total 20 combinations (can encode 4 bits + 4 control tokens)**
- **2-of-7:**
 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

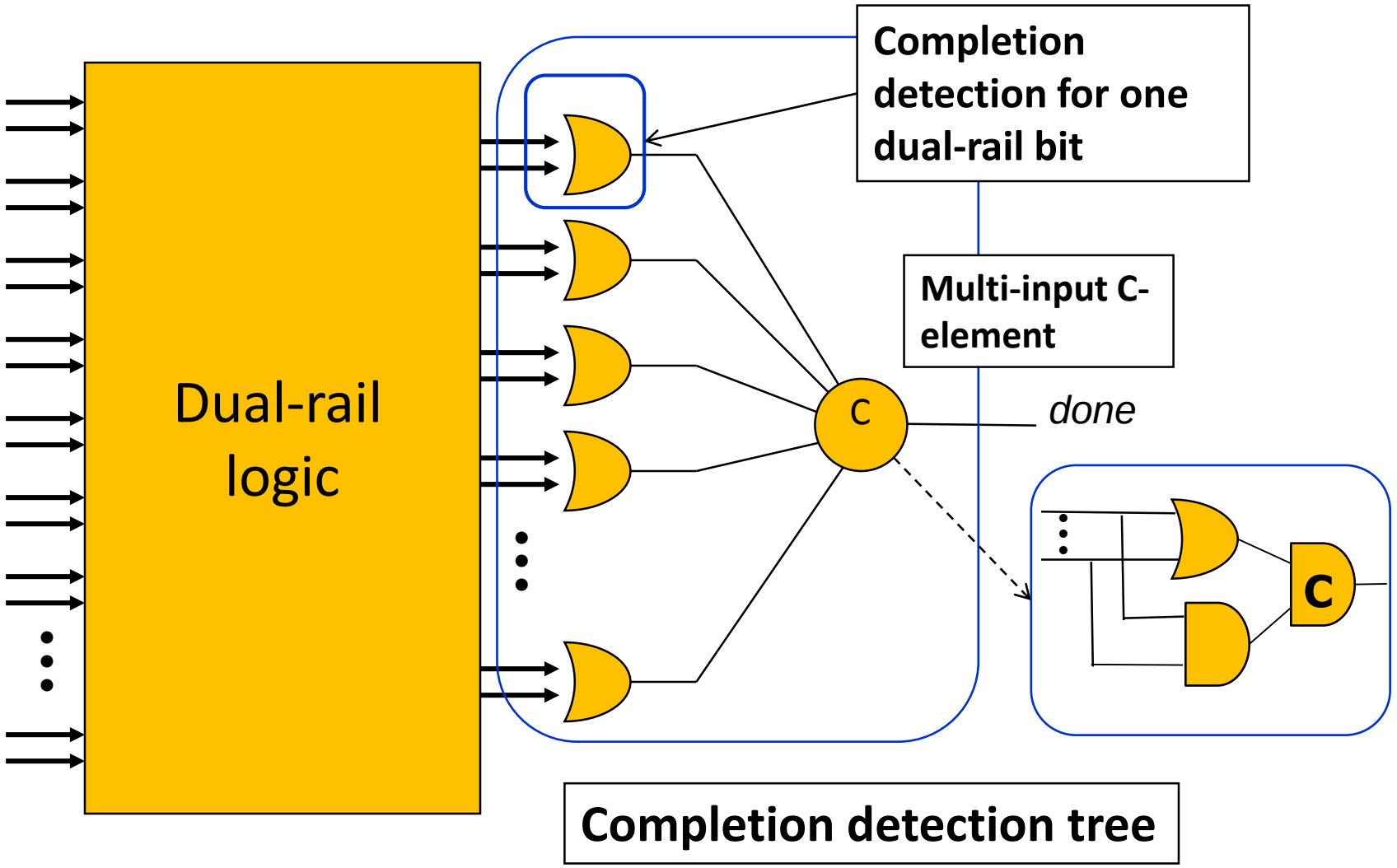


Conventional logic + matched delay

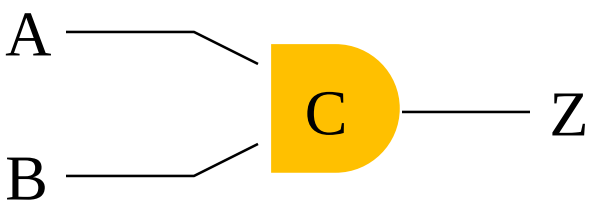
Completion is implicit: by done signal

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

True completion detection

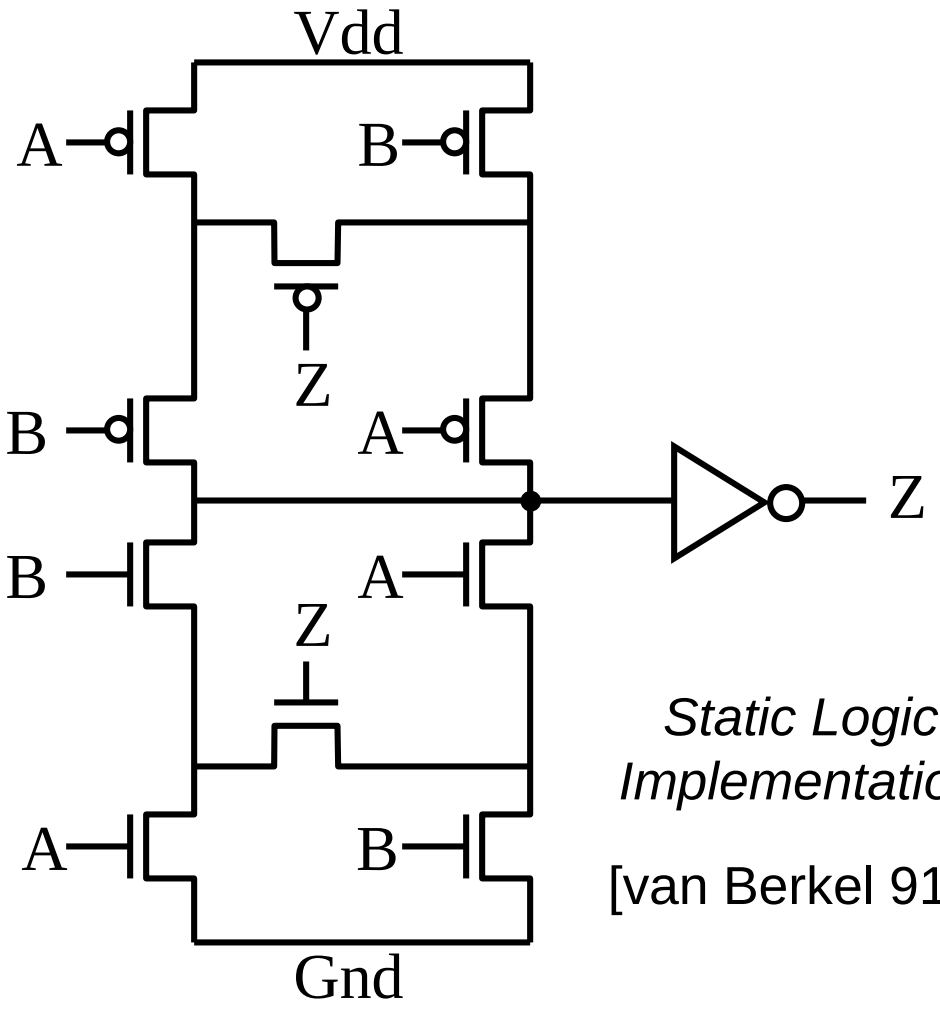


The Muller C element



C

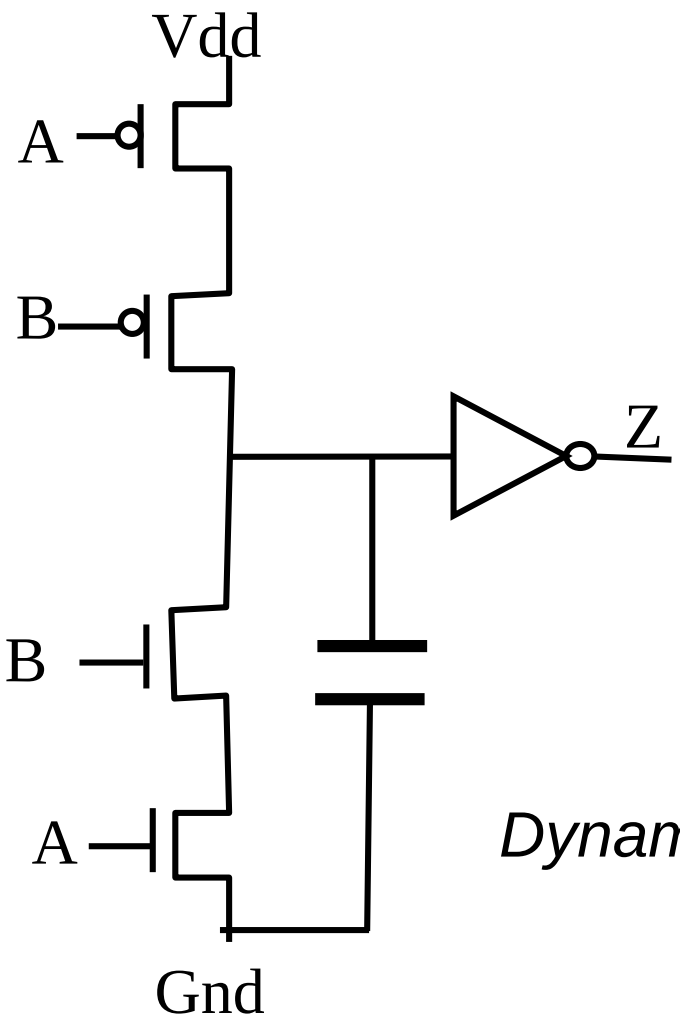
A	B	Z^+
0	0	0
0	1	Z
1	0	Z
1	1	1



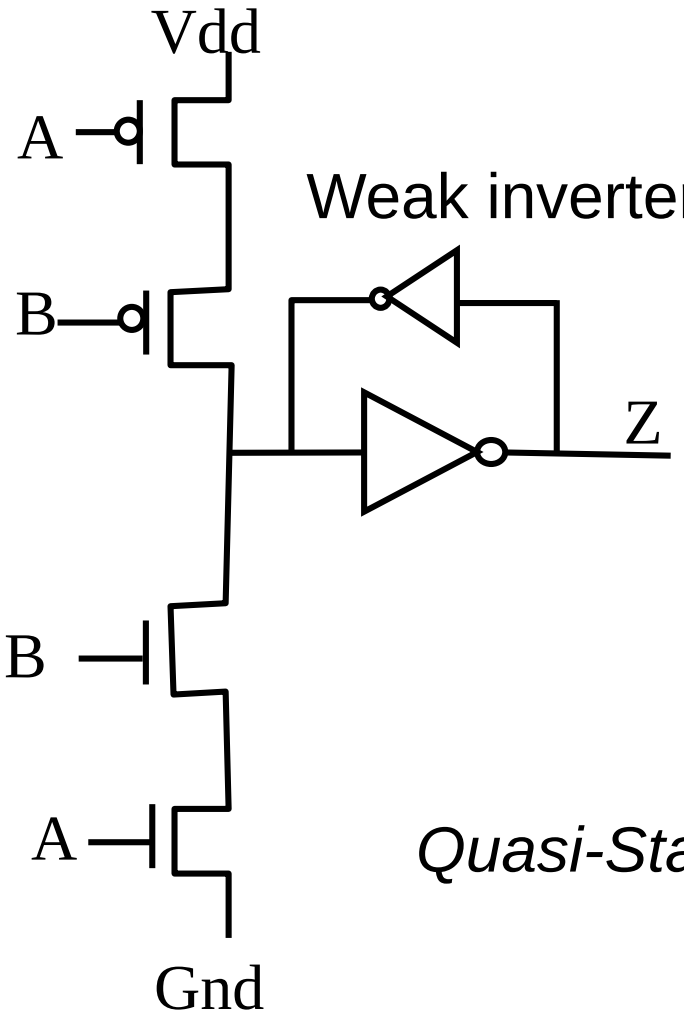
Static Logic Implementation

[van Berkel 91]

C-element: Other implementations



Dynamic

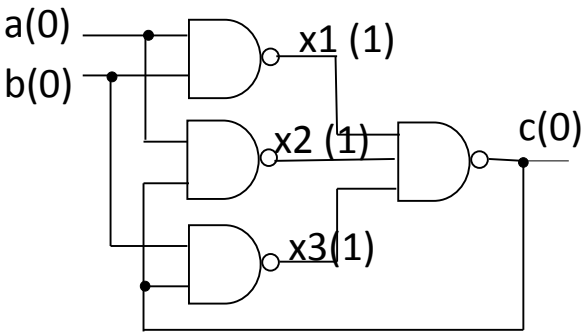


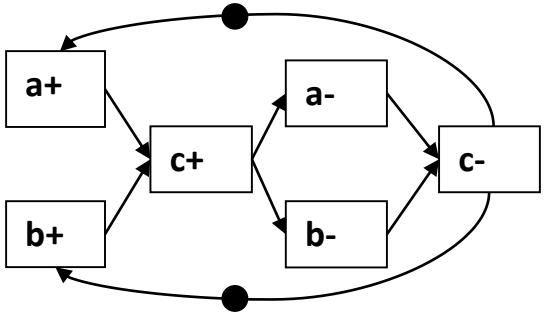
Quasi-Static

Why and what is causal acknowledgment?

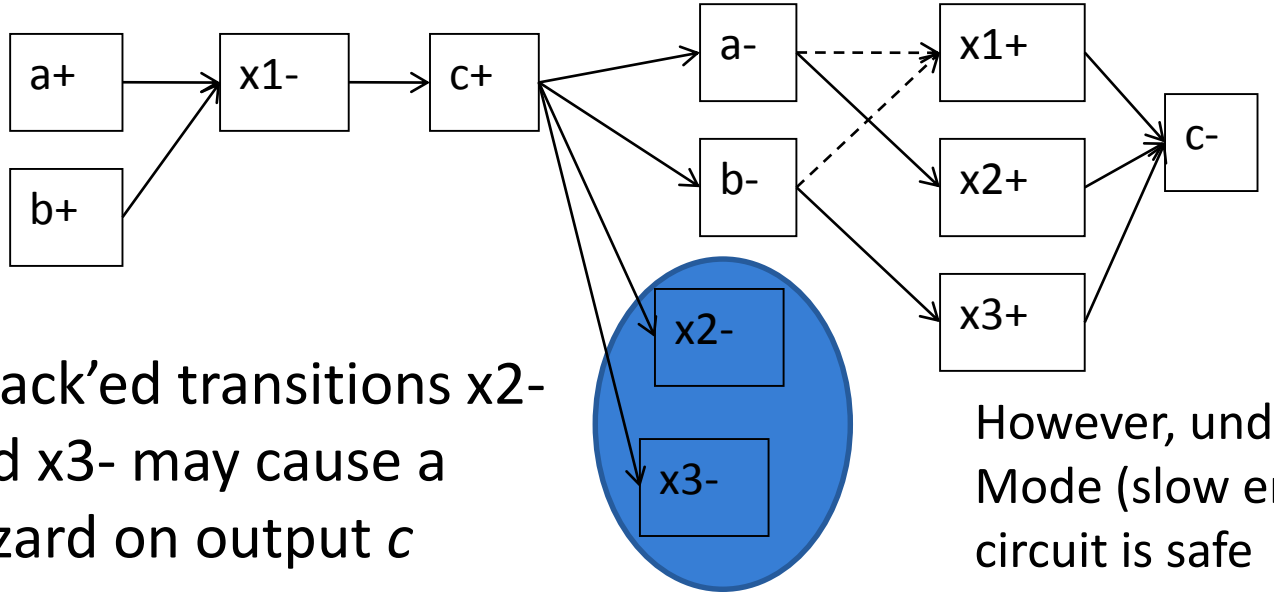
Every signal event must be acknowledged by another event

Causal acknowledgment





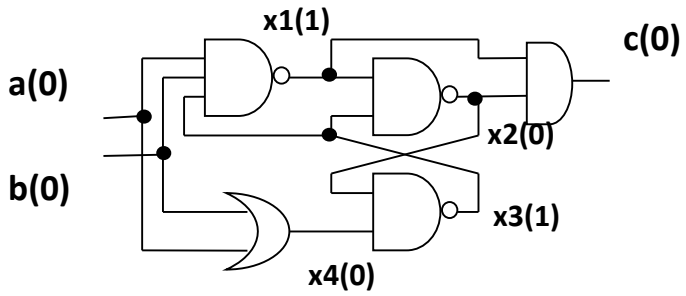
C-element implementation using simple gates

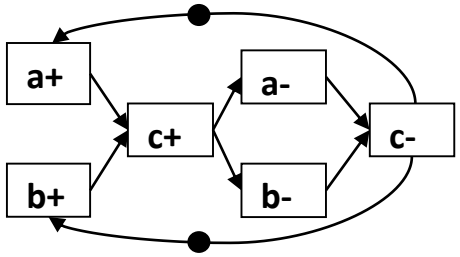


Unack'ed transitions x2- and x3- may cause a hazard on output *c*

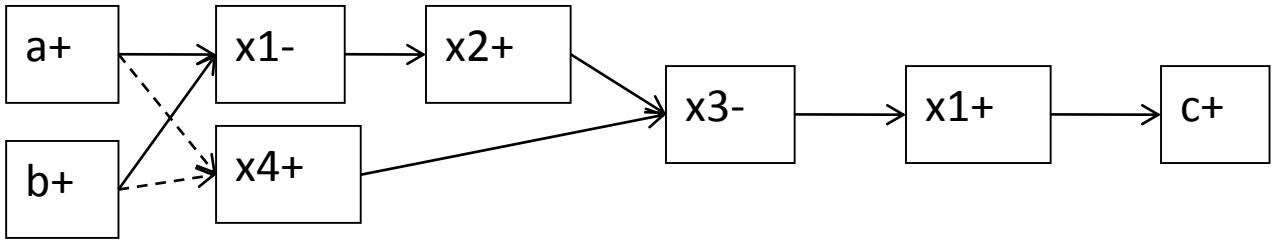
However, under Fundamental Mode (slow environment) the circuit is safe

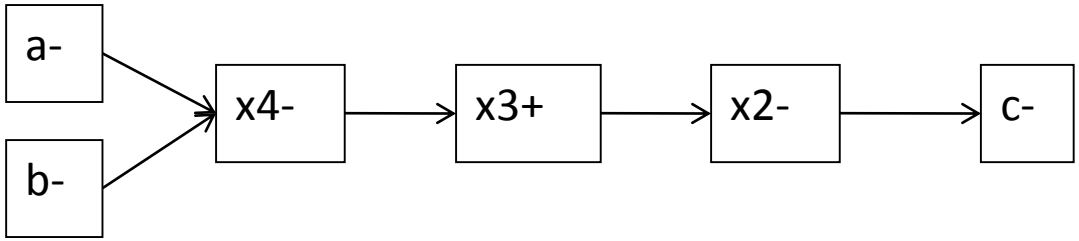
Principle of causal acknowledgement





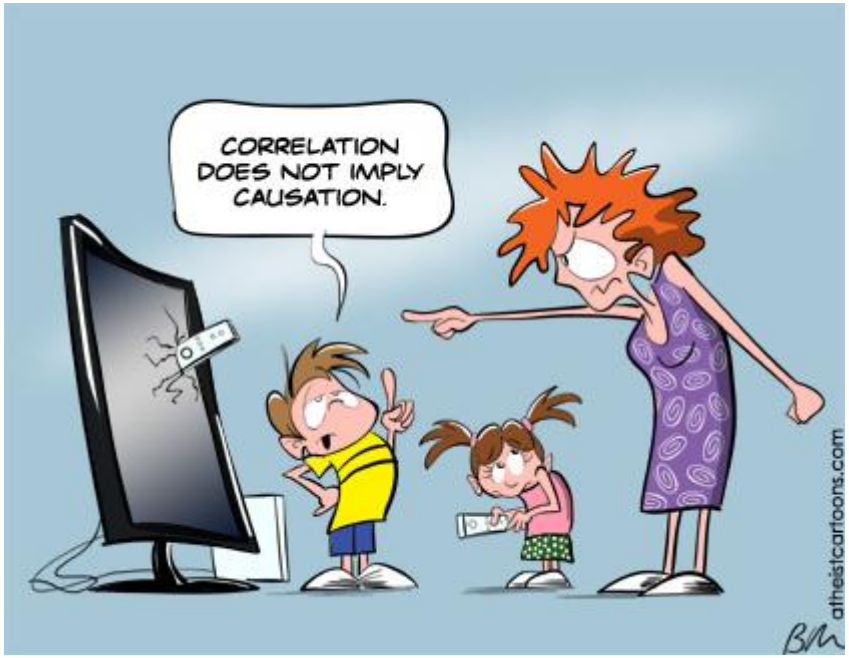
C-element implementation using simple gates





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

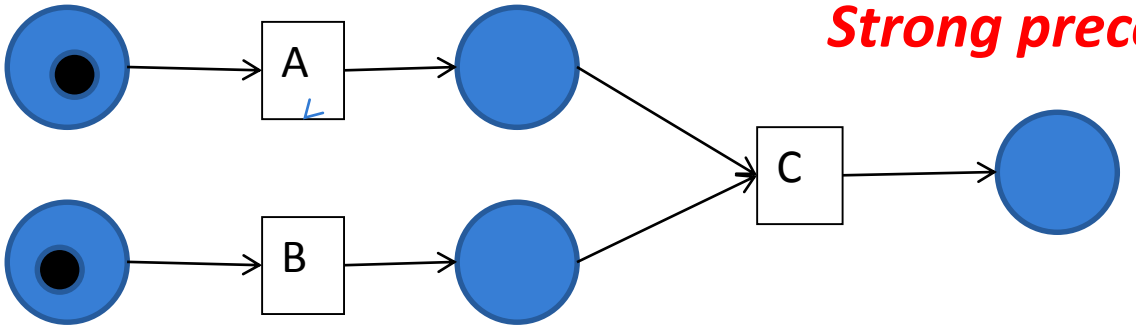
Why and what are strong and weak causality ?



Degree of necessity of precedence of some events for other events

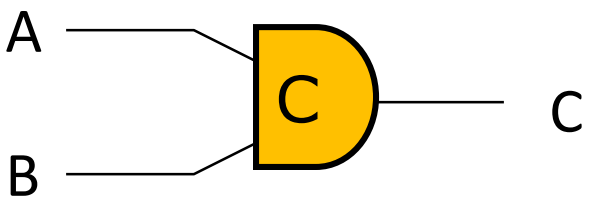
Strong Causality

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



Strong precedence

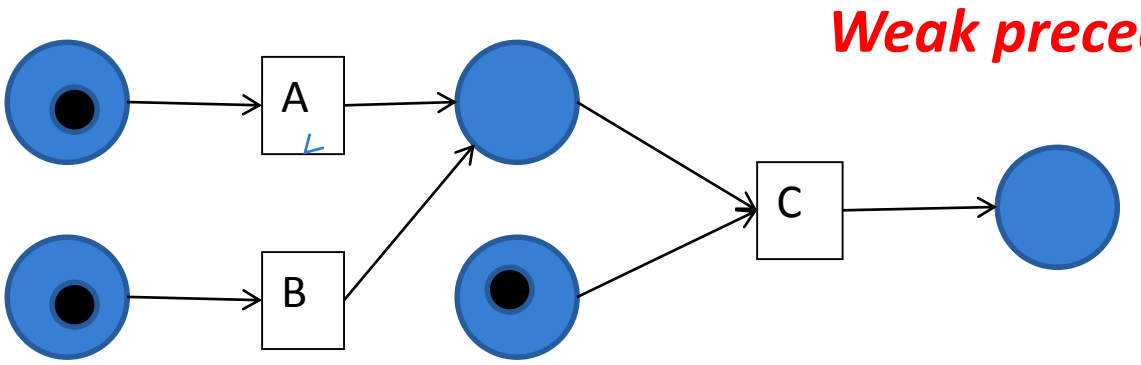
- **Logic circuits: Muller C-element (in 0-1 and 1-0 transitions), AND gate (in 0-1 transitions), OR gate (in 1-0 transitions)**



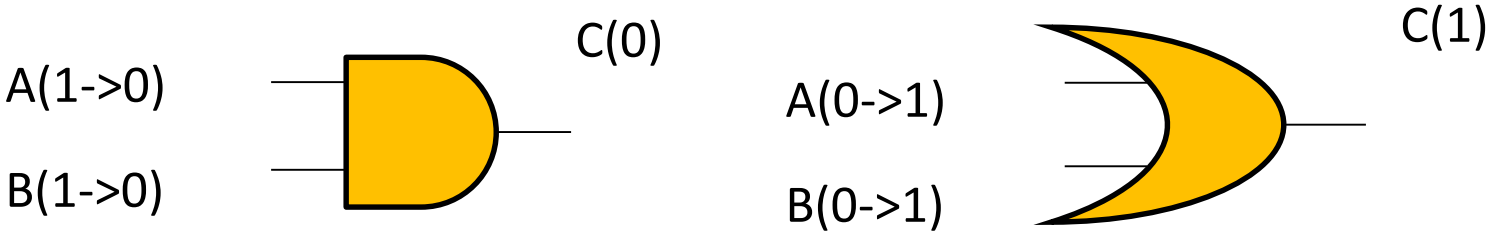
A	B	C^+
0	0	0
0	1	C
1	0	C
1	1	1

Weak Causality

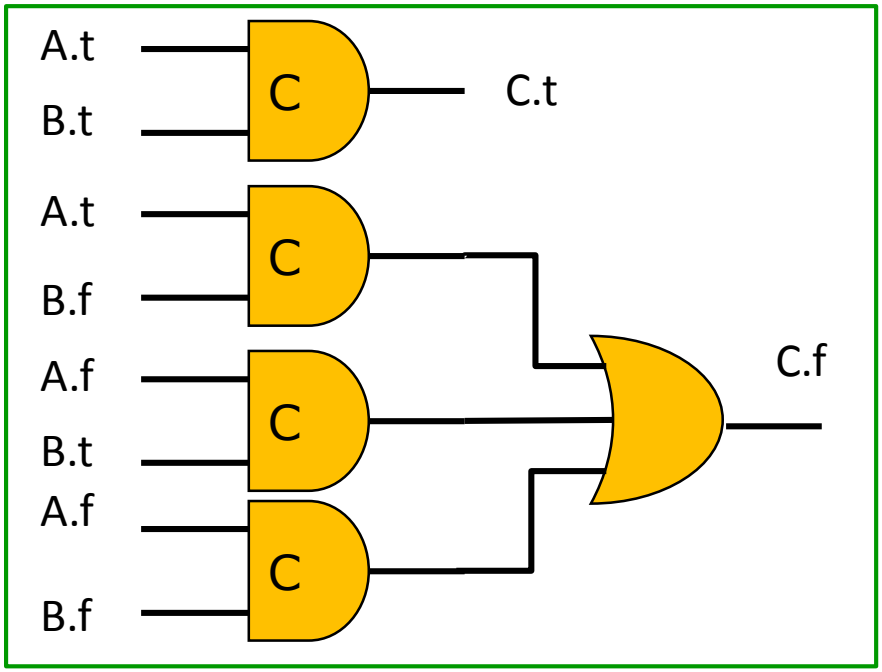
- **Petri net transitions communicating via places**



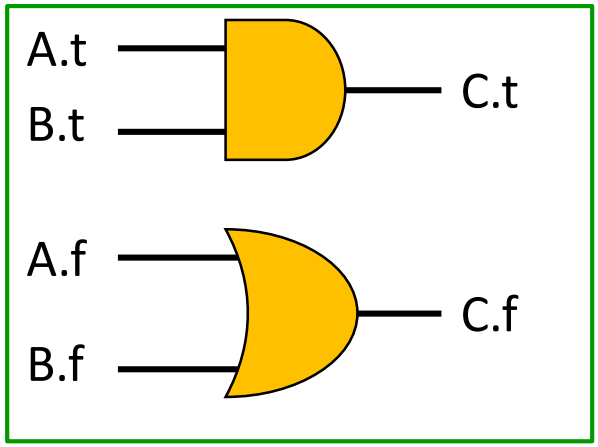
- **Logic circuits: AND gate (in 1-0 transitions), OR gate (in 0-1 transitions)**



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)

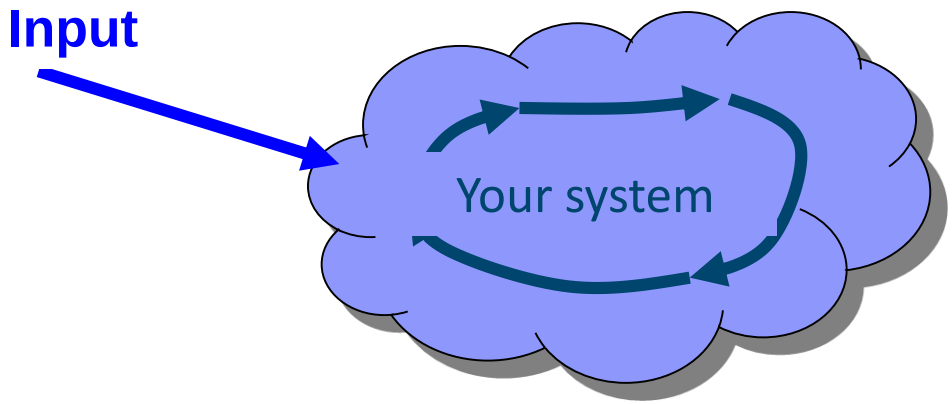
Why and what is timing comparison?

Telling if some event happened before another event

Synchronizers and arbiters

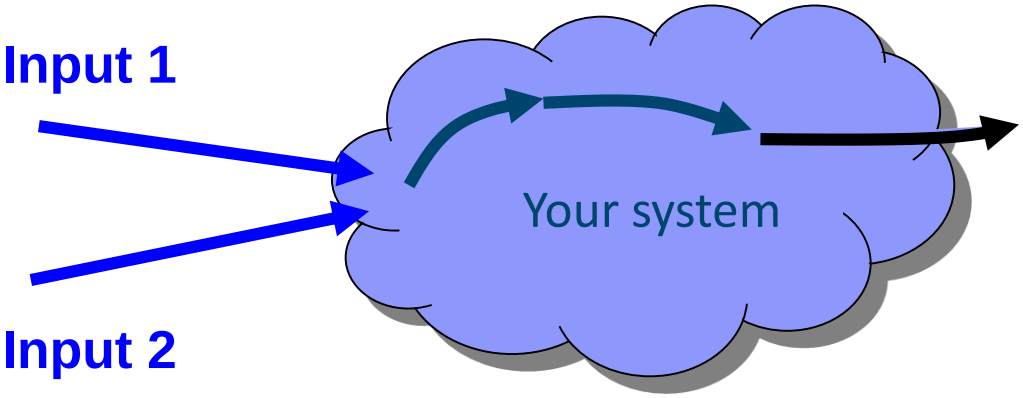
- Synchronizer

Decides which clock cycle to use for the input data

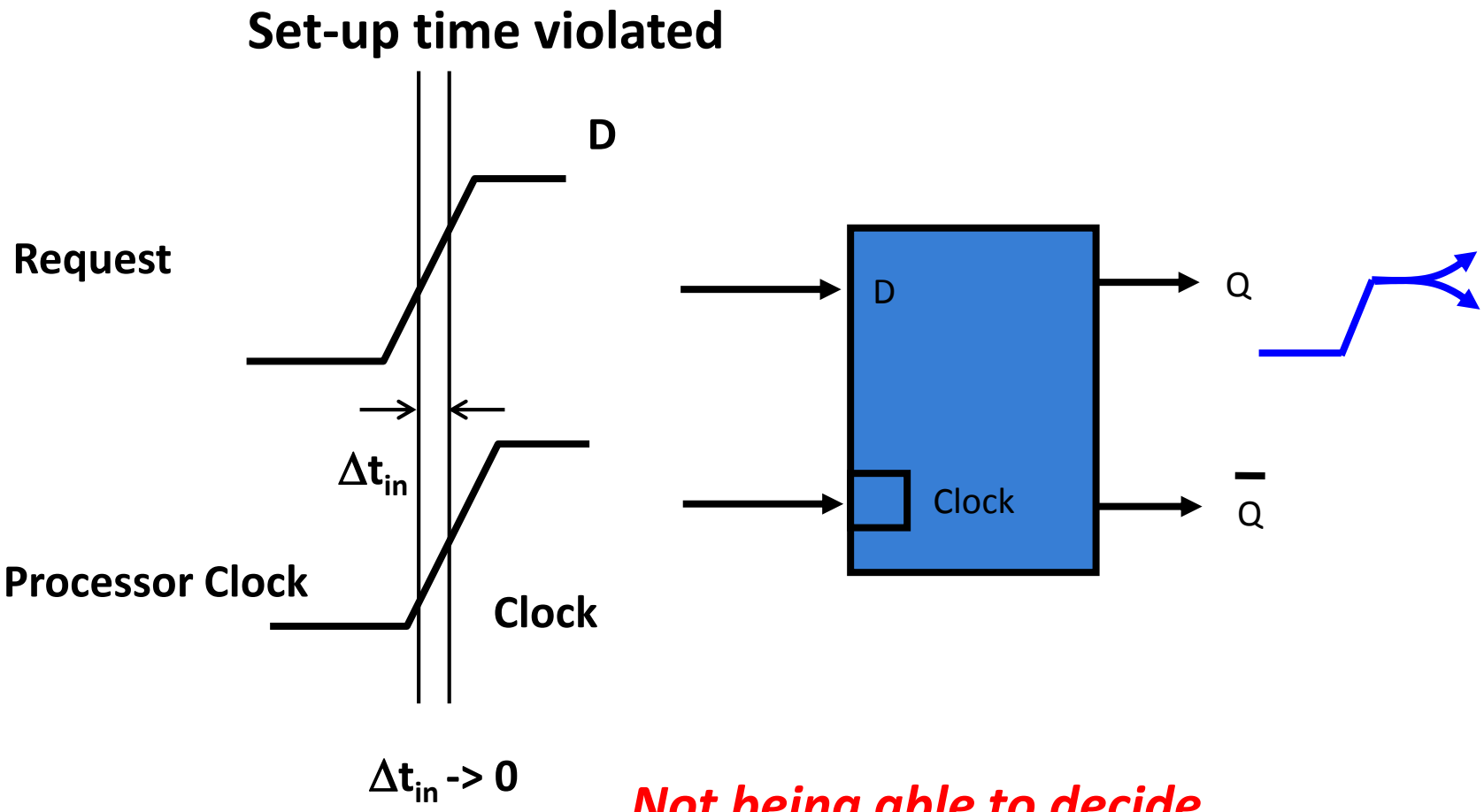


- Asynchronous arbiter

Decides the order of inputs

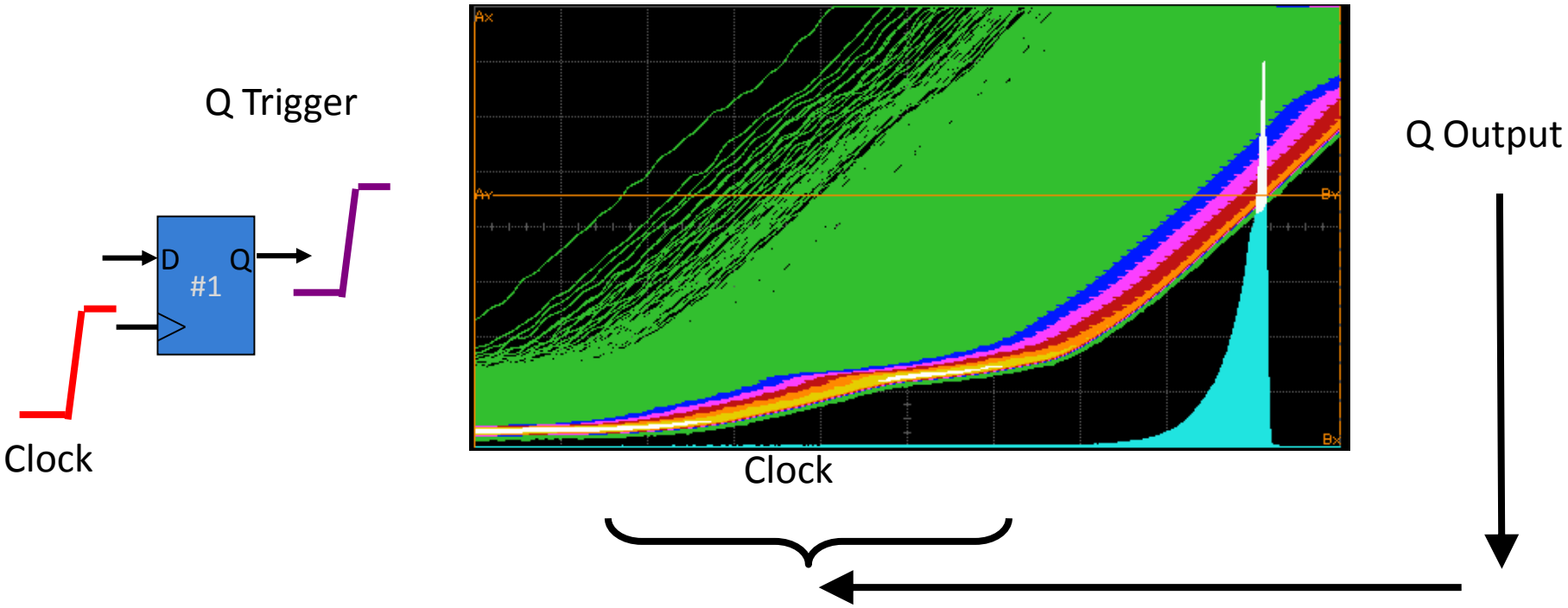


Metastability is....



Not being able to decide…

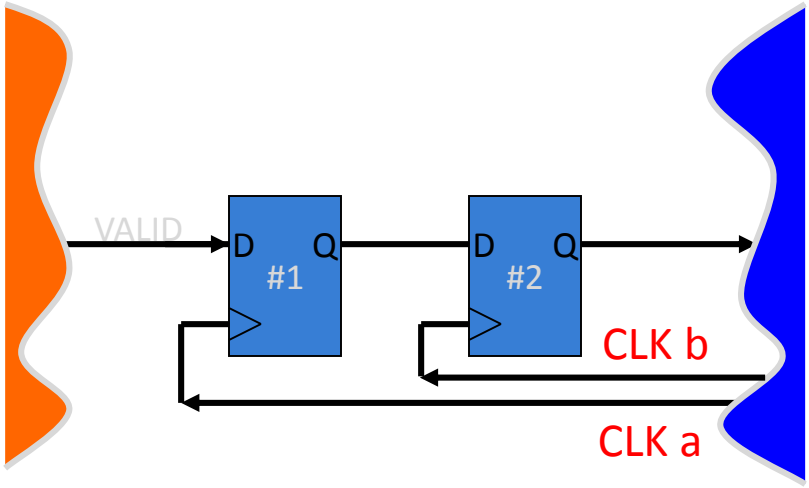
Typical responses



- **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**

Synchronizer

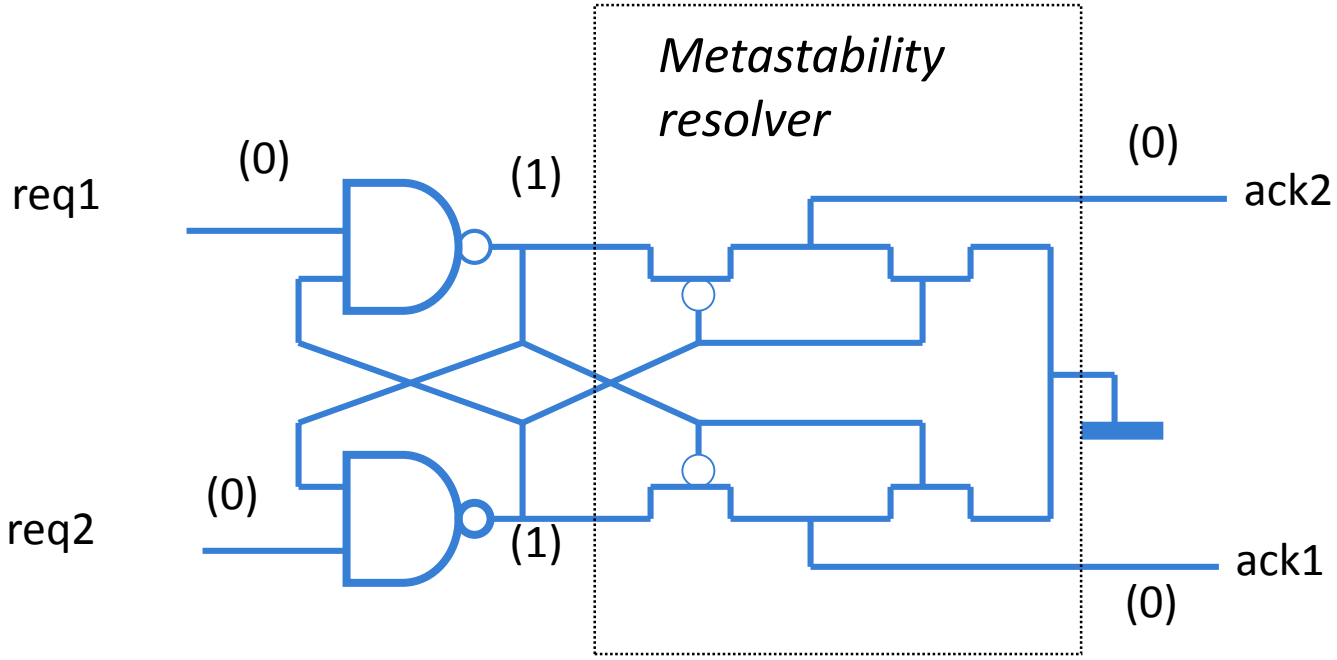
- t is time allowed for the Q to change between CLK a and CLK b
- τ is the recovery time constant, usually the gain-bandwidth of the circuit
- T_w is the "metastability window" (aperture around clock edge in which the capture of data edge causes a delay that is greater than normal propagation delay of the FF)
- τ and T_w depend on the circuit
- We assume that all values of Δt_{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 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

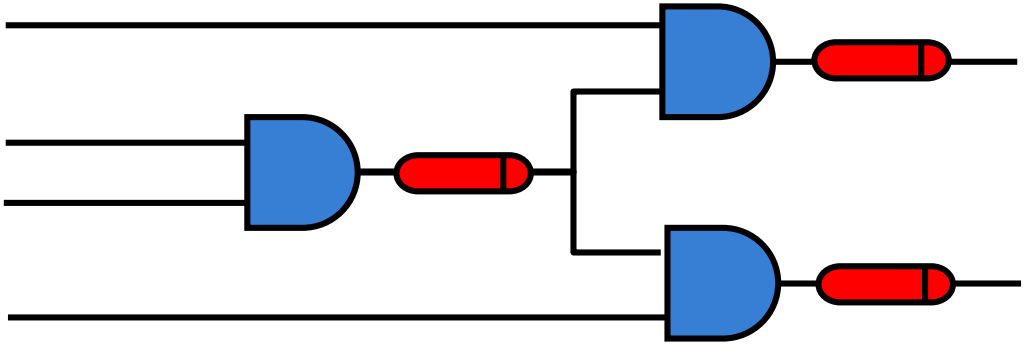
(Some) Models for Asynchronous Circuit Design

Models and techniques for asynchronous design

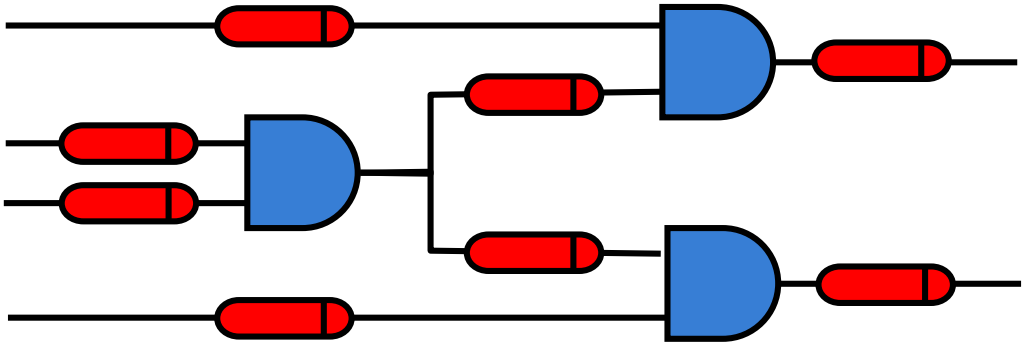
- **Models:**
  - **Delay model (inertial, pure, gate delay, wire delay, bounded and unbounded delays)**
  - **Models of environment (fundamental mode, input-output)**
  - **Models of switching behaviour (state-based, event-based, hybrid)**
- **RTL level:**
  - **Data and control paths separate (data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

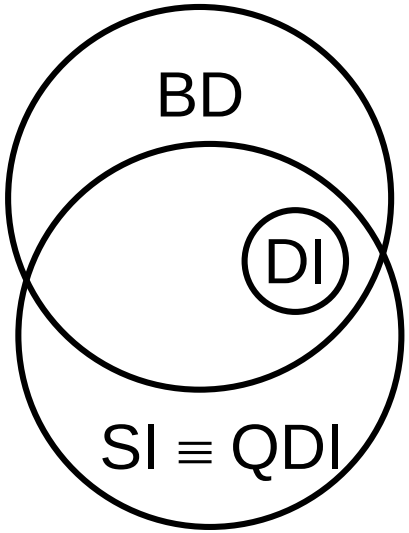


- **Wire delay model: delays in gates and wires**



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

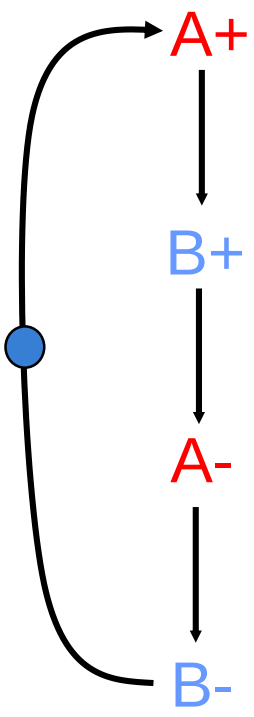


Control Logic

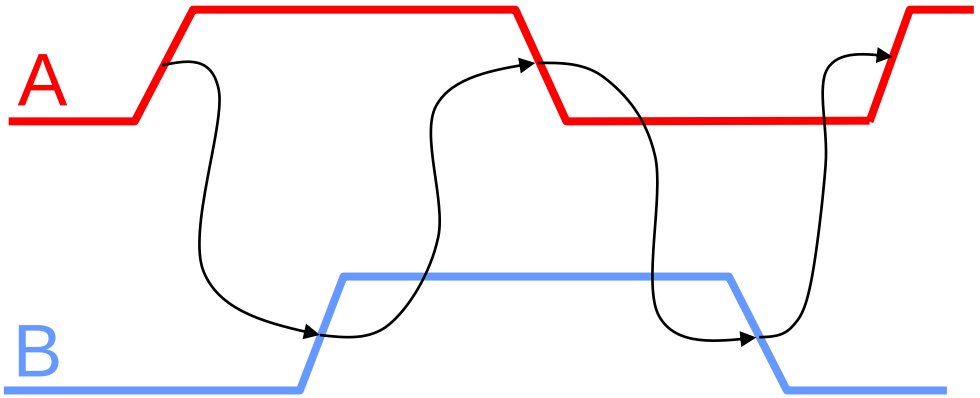
- **Control specification based on Petri nets (Signal Transition graphs)**

Control specification

Signal Transition Graph (STG)



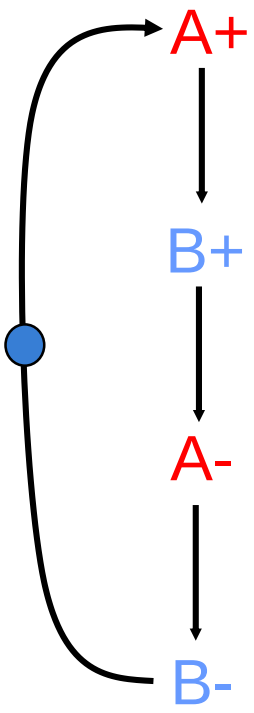
Timing Diagram

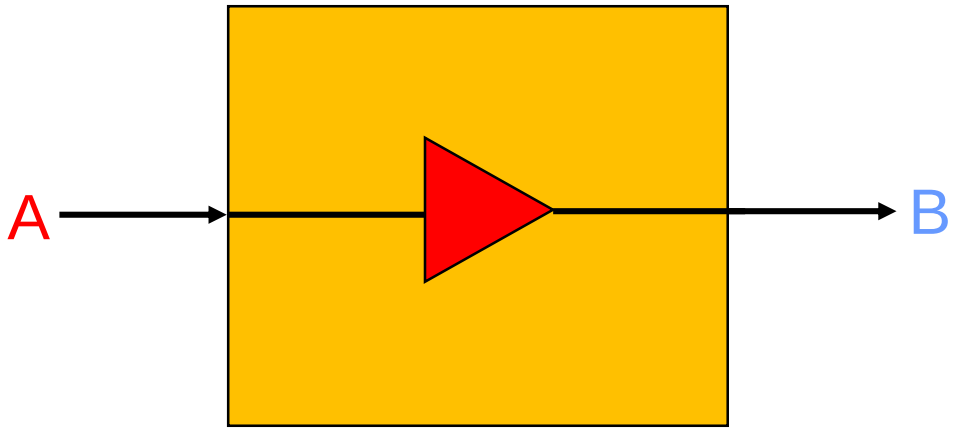


A input

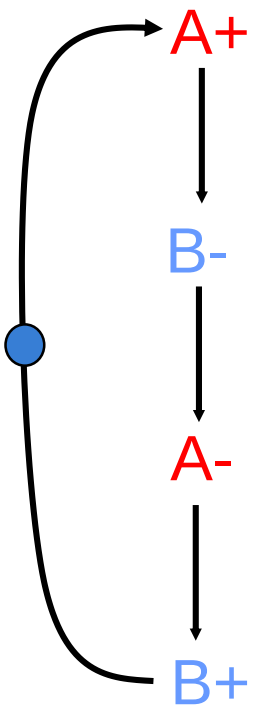
B output

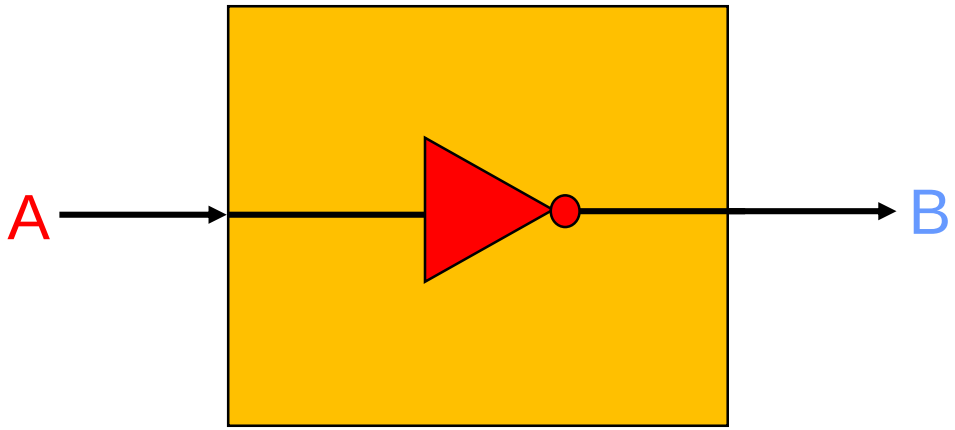
Control specification



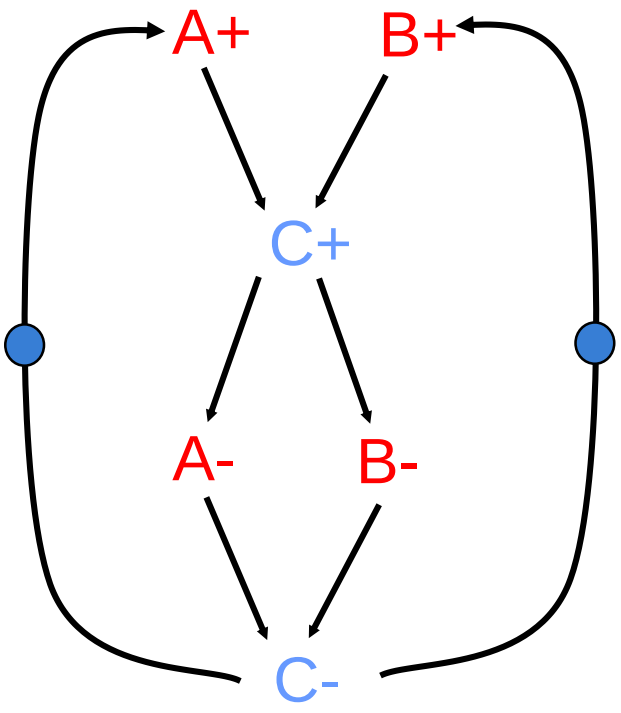


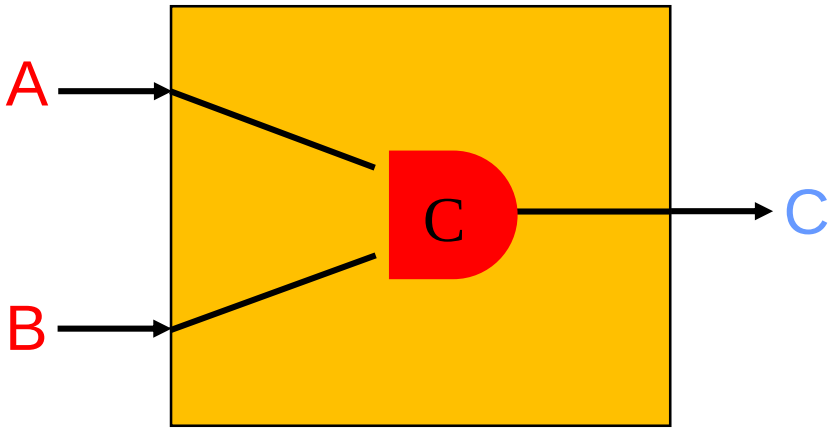
Control specification



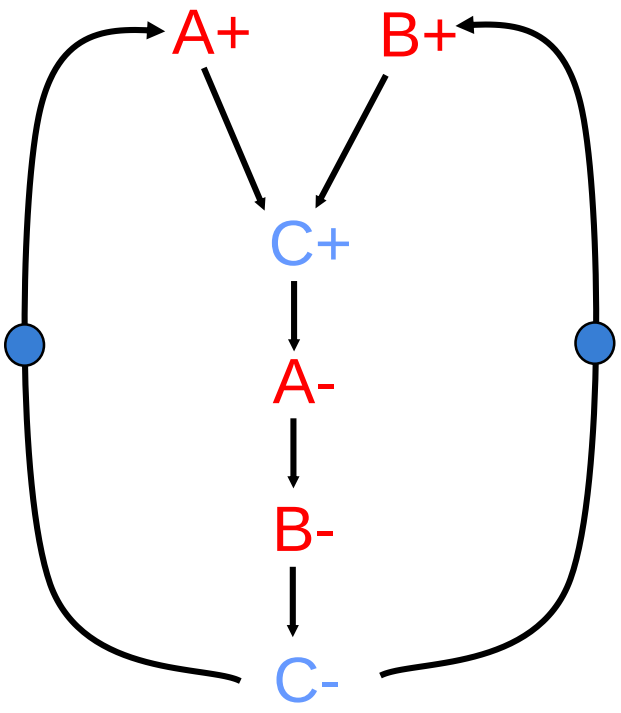


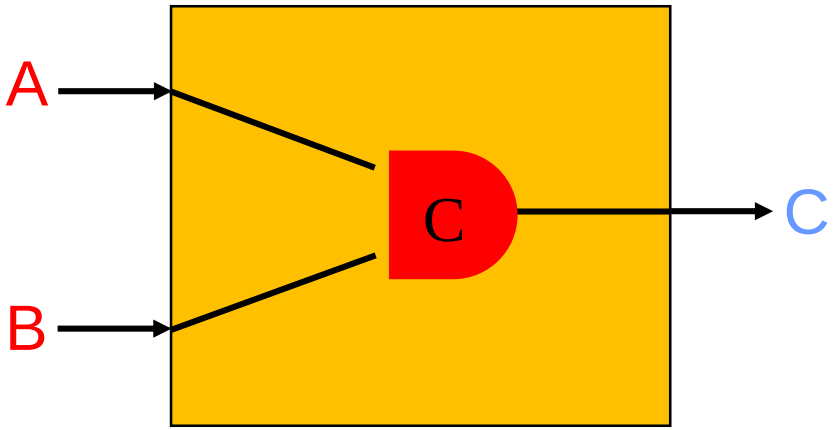
Control specification



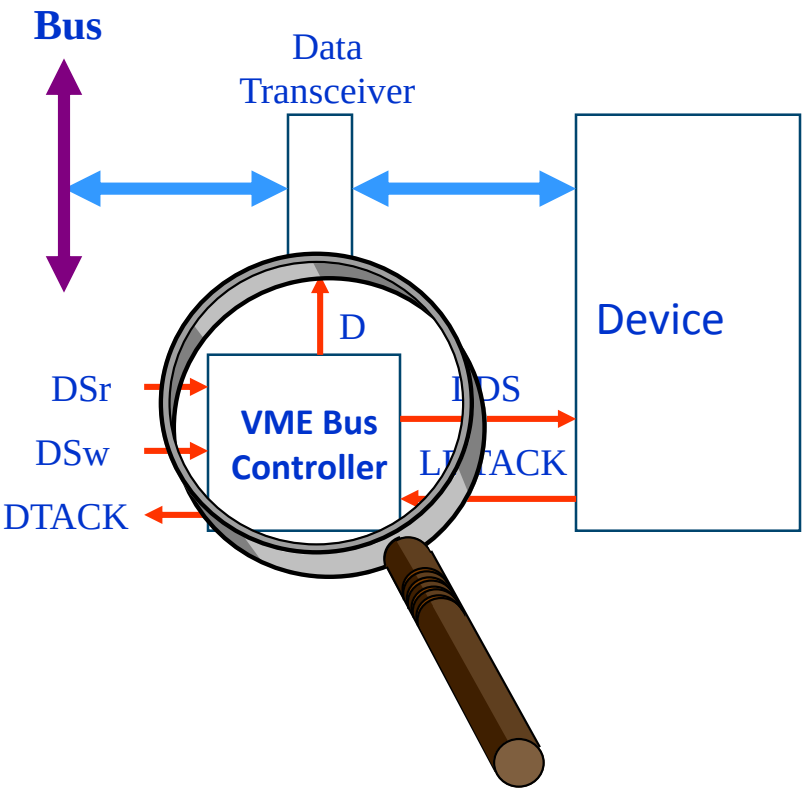


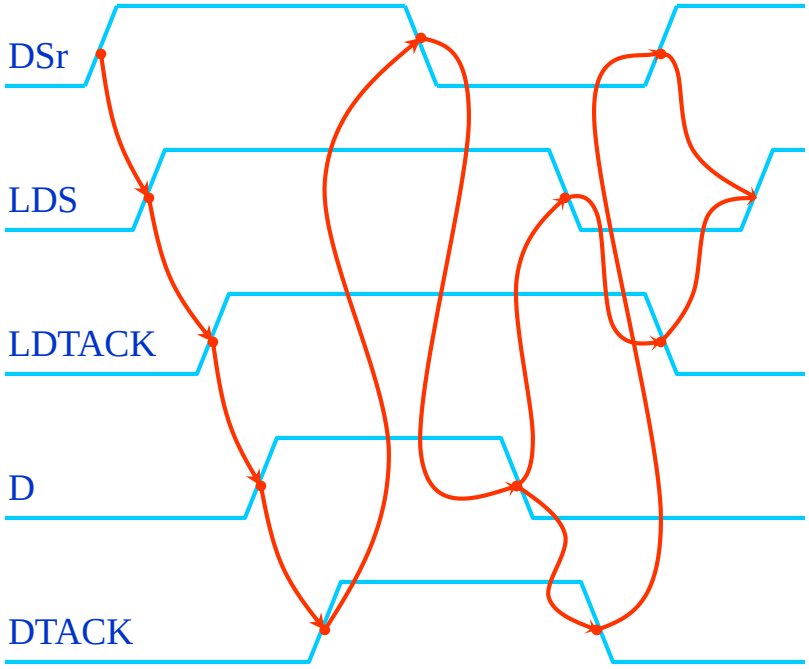
Control specification





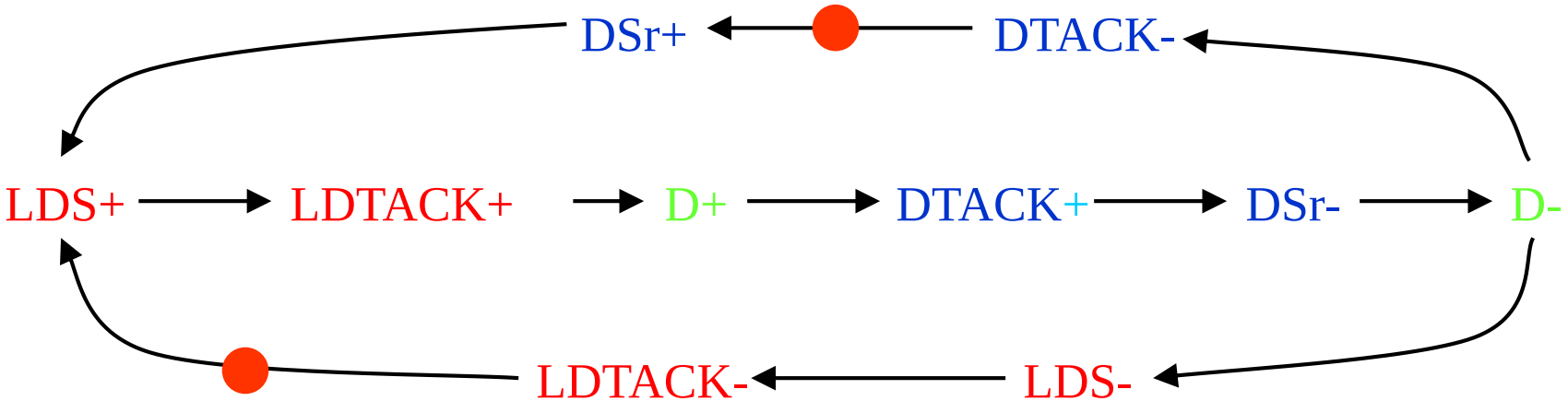
VME bus example using Petri nets

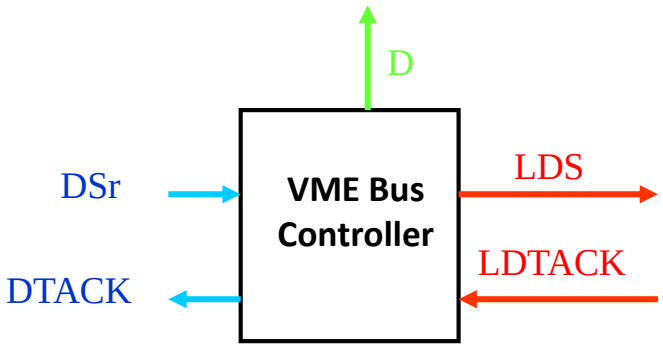




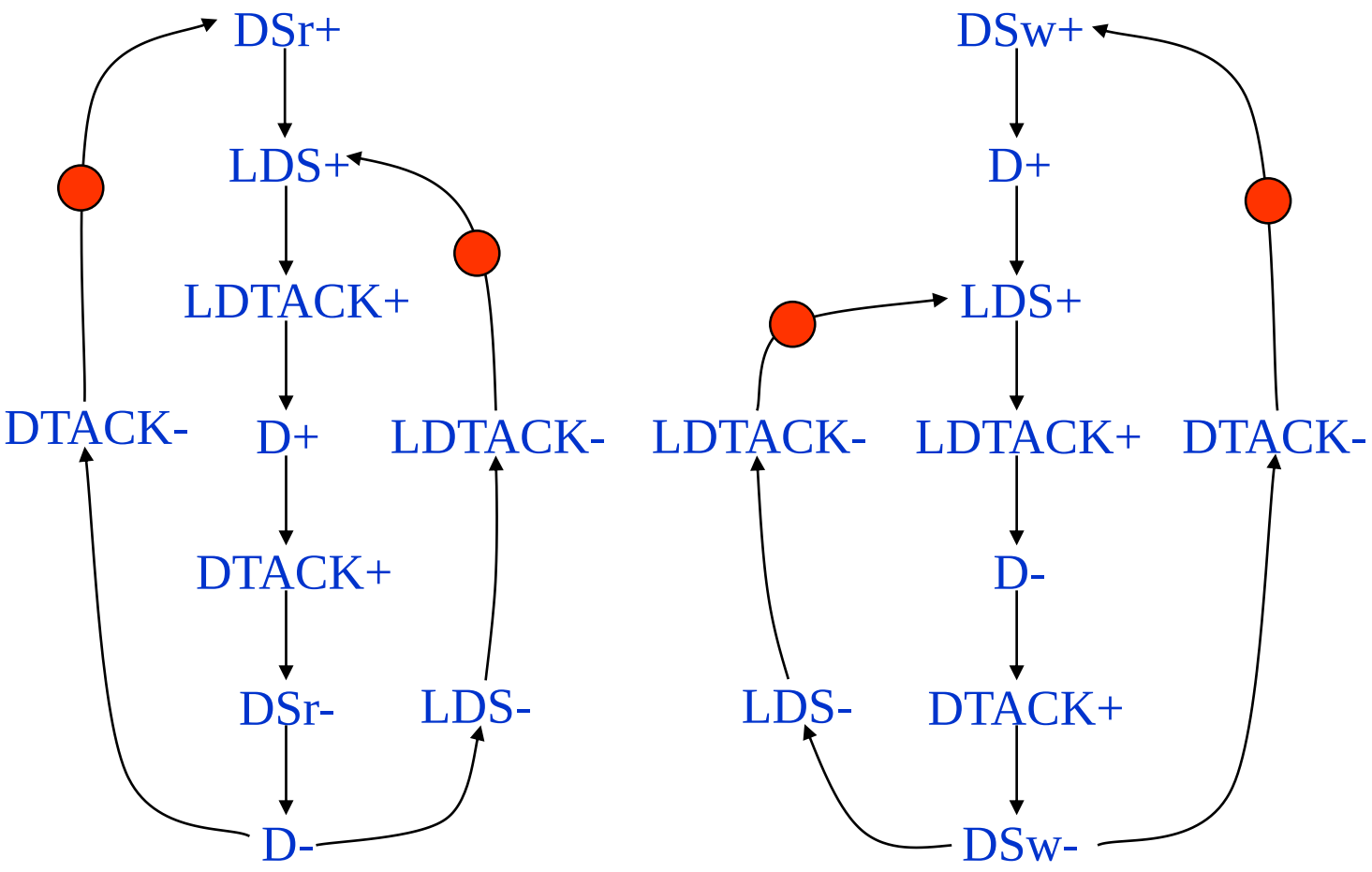
Read Cycle

STG for the READ cycle

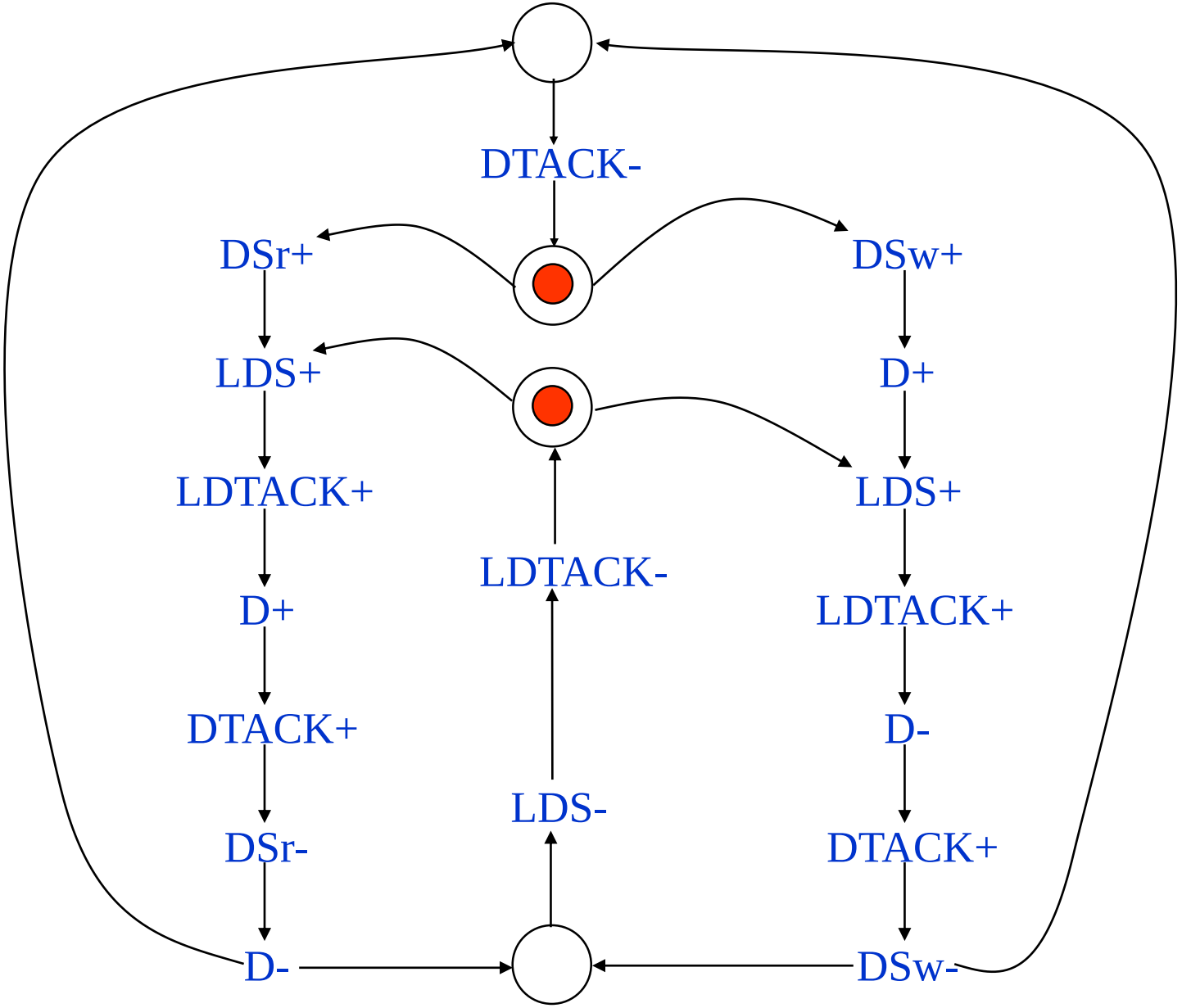




Choice: Read and Write cycles



Choice: Read and Write cycles



Workcraft tool

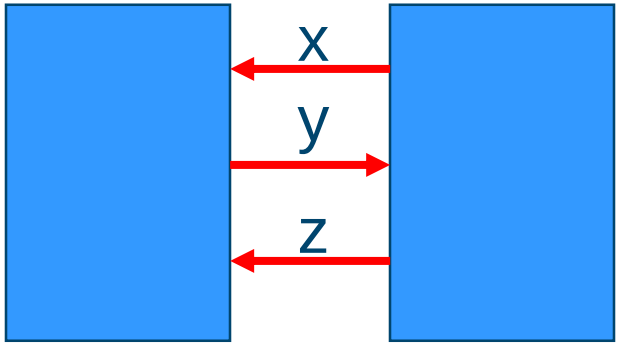
- Workcraft is a software package for graphical edit, analysis, synthesis and visualisation of asynchronous circuit behaviour
- Petrify plus a few other tools are part of it as plug-ins
- It is based in Java tools
- Can be downloaded from http://workcraft.org/wiki/doku.php?id=download
- And installed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

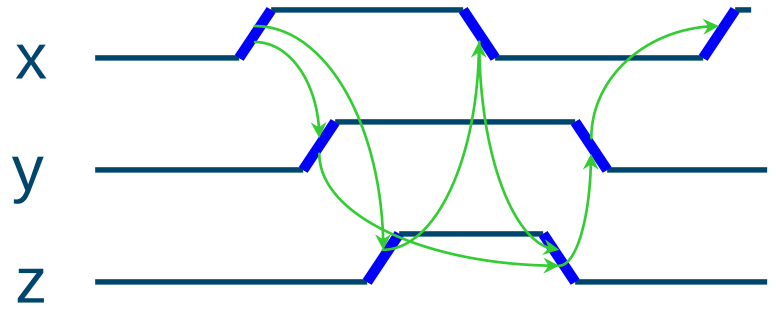
Some references

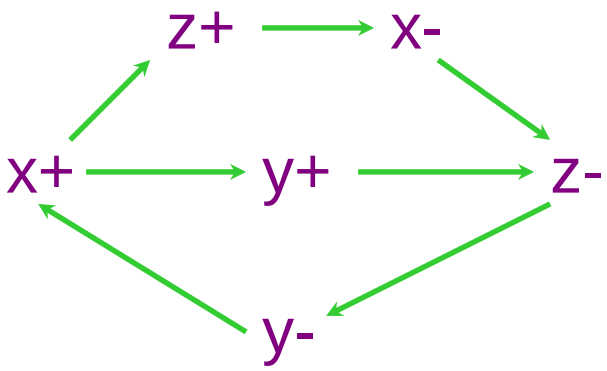
- **General Async Design:** J. Sparsø and S.B. Furber, editors. *Principles of Asynchronous Circuit Design*, Kluwer Academic Publishers, 2001. (electronic version of a tutorial based on this book can be found on: http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/855/pdf/imm855.pdf
- **Async Control 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)

Asynchronous Control Logic Synthesis from STGs: Basics

xyz-example: Specification

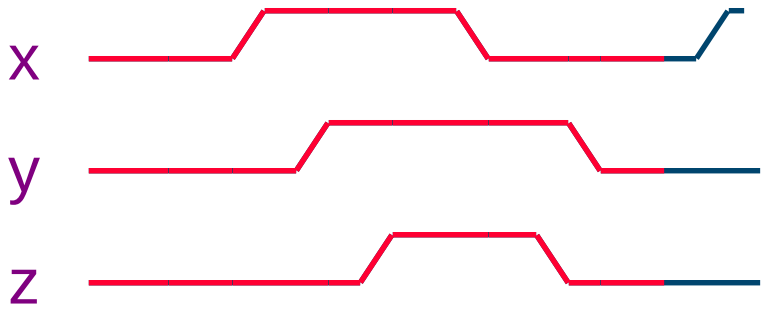


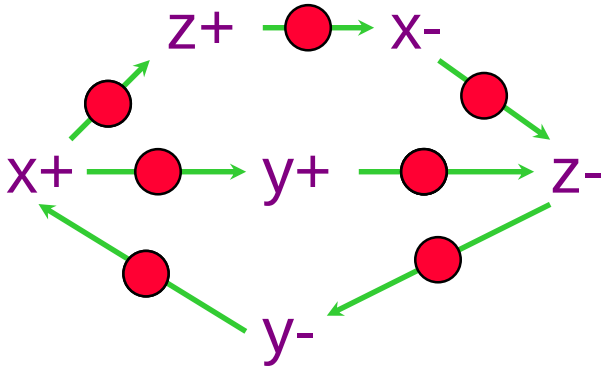




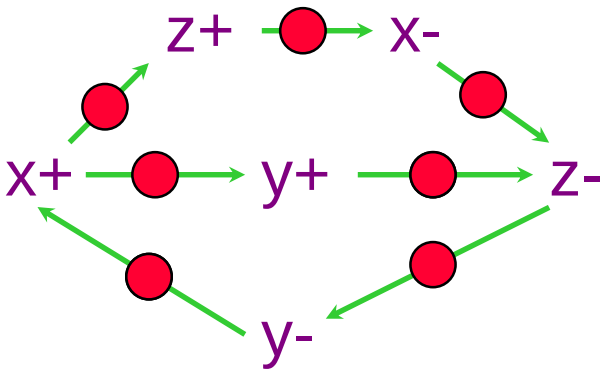
Signal Transition Graph (STG)

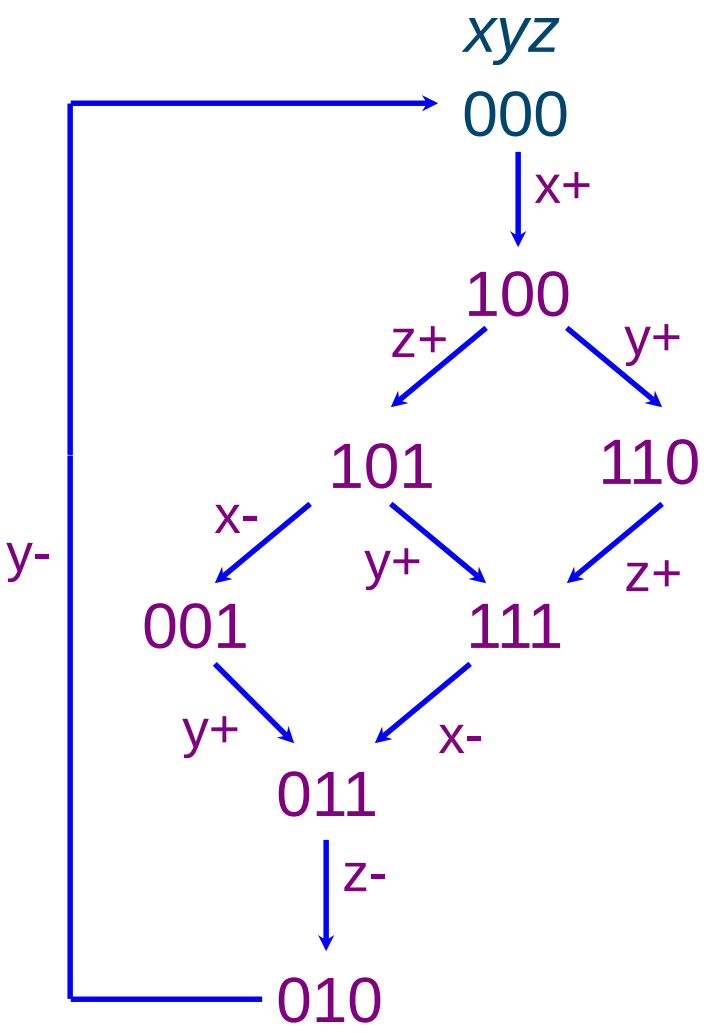
Token flow





State graph



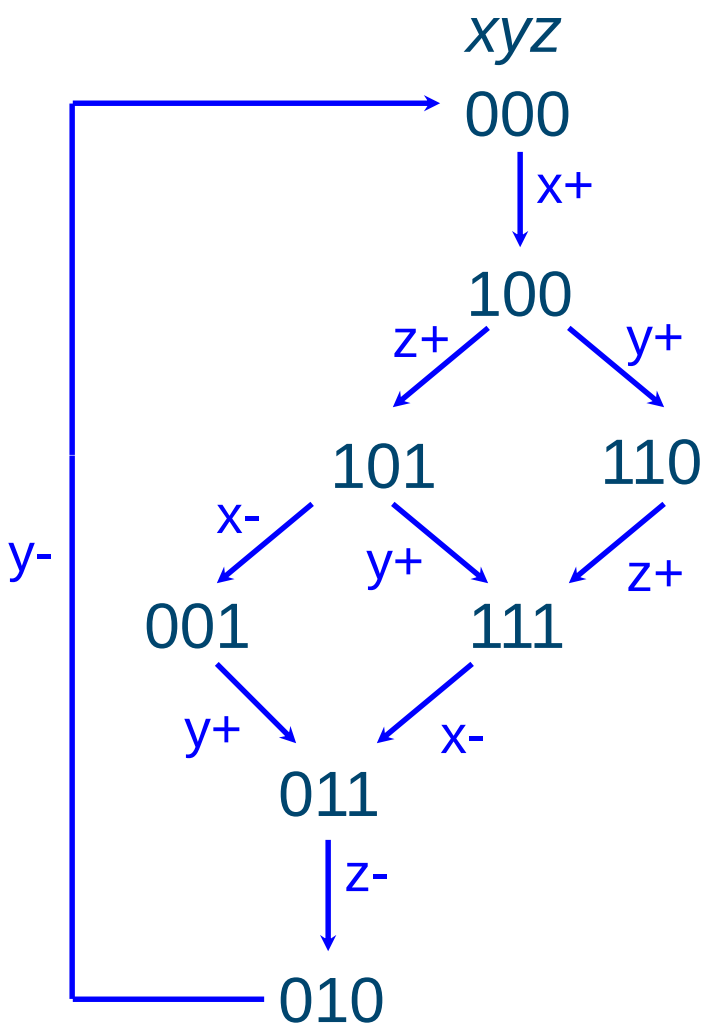


Next-state functions

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

$$y = z + x$$

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

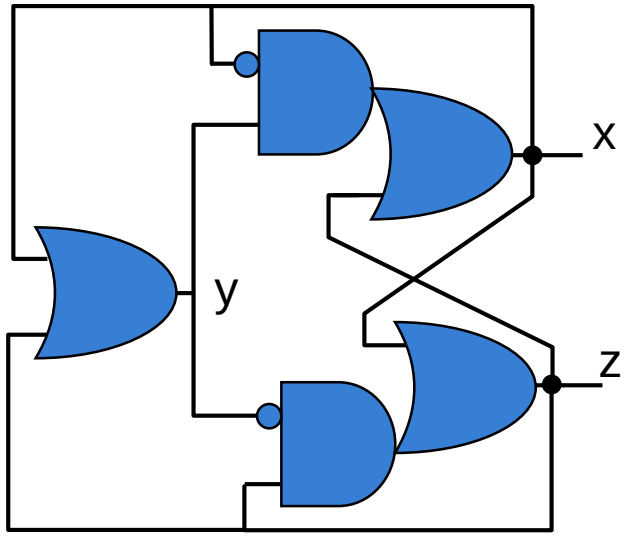


Complex Gate netlist

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

$$y = z + x$$

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



Circuit synthesis

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

Speed independence

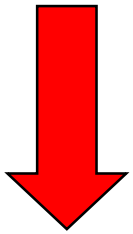
- **Delay model**
  - **Unbounded gate / environment delays**
  - **Certain wire delays shorter than certain paths in the circuit**
- **Conditions for implementability:**
  - **Consistency**
  - **Complete State Coding**
  - **Output-Persistency**

Implementability conditions

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

Implementability conditions

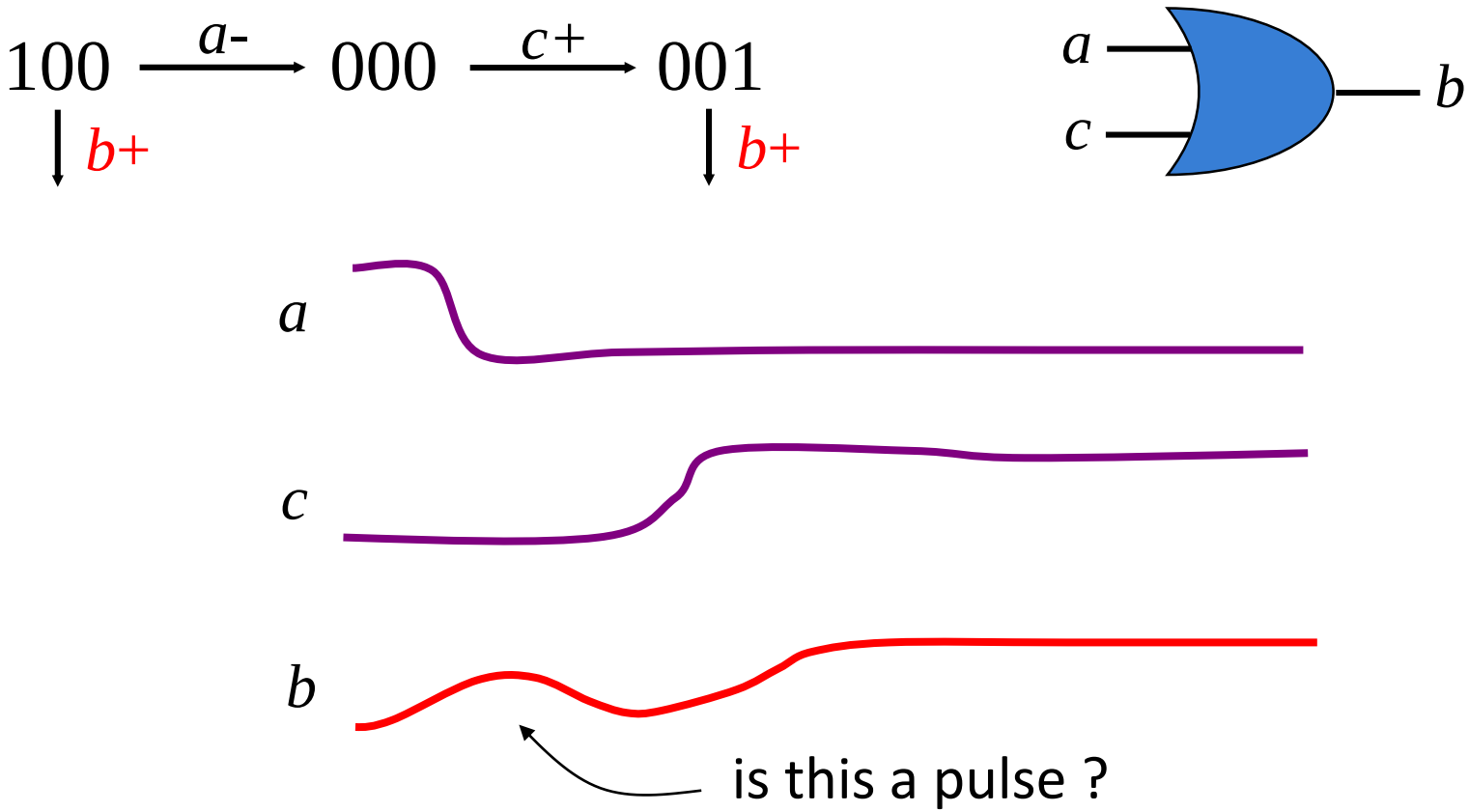
- **Consistency + CSC + output-persistency**



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

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

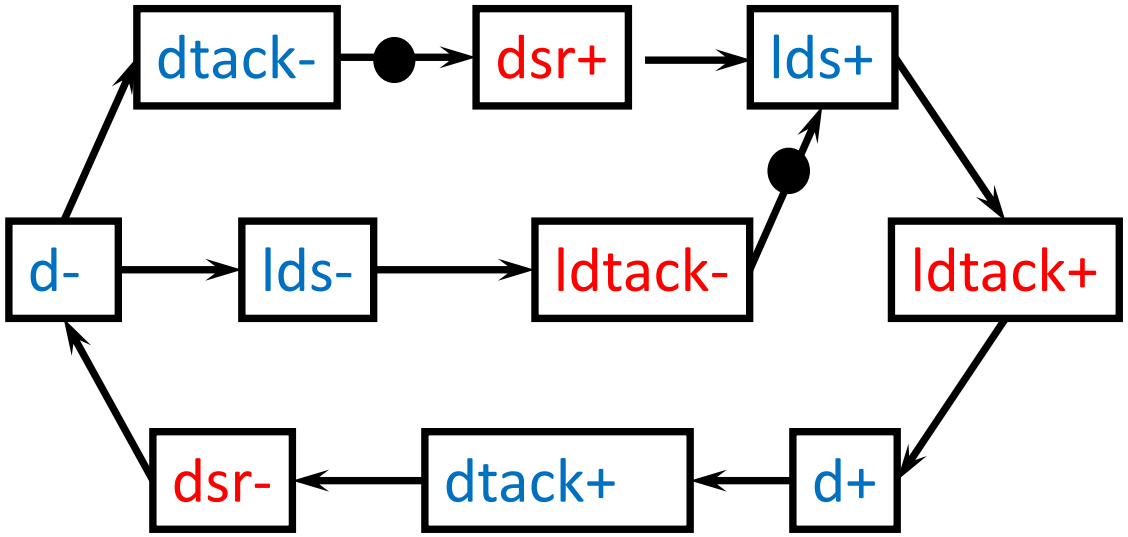
Persistency



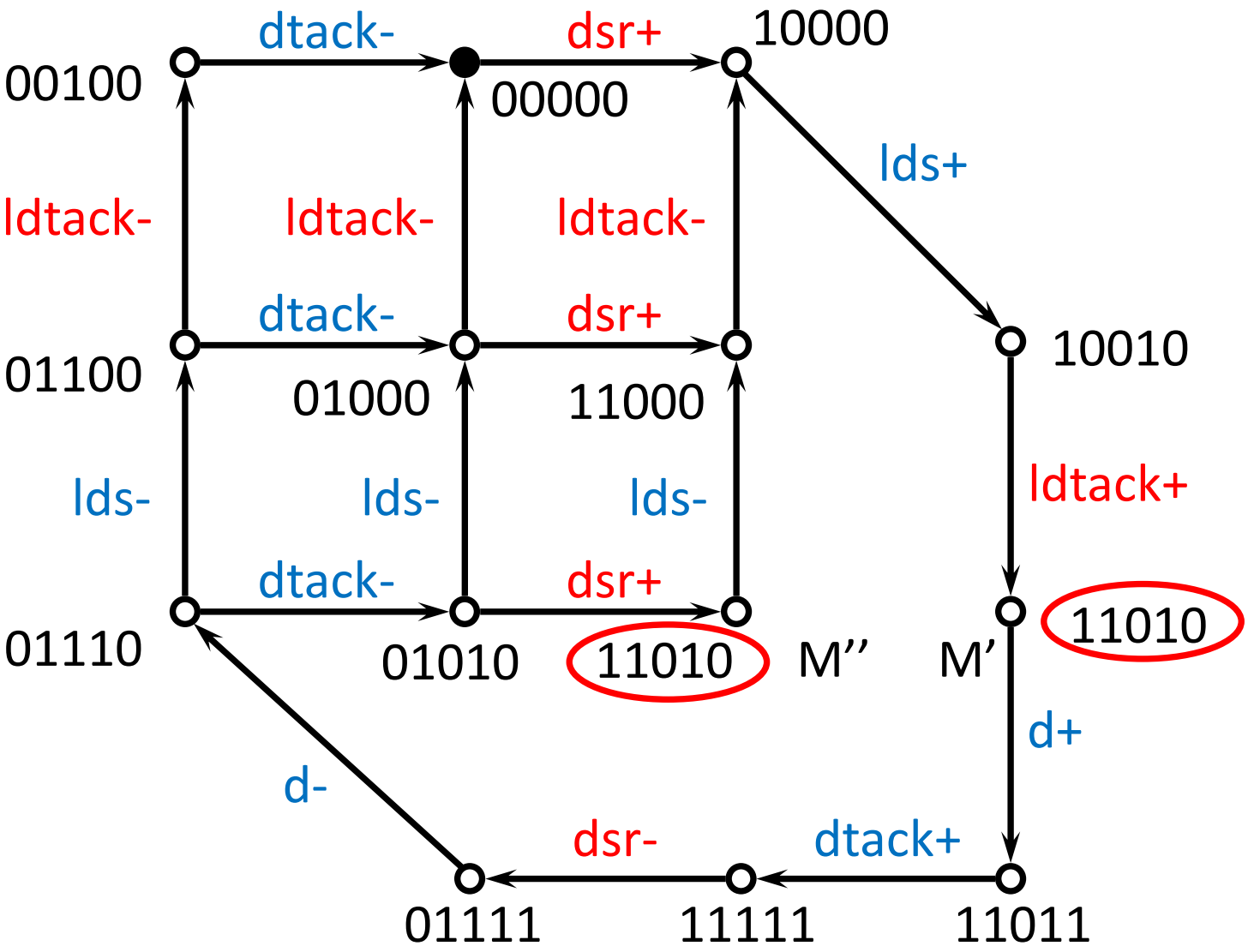
Speed independence ⇒ glitch-free output behaviour under any delay

Complete State Coding (CSC) Conflict Resolution

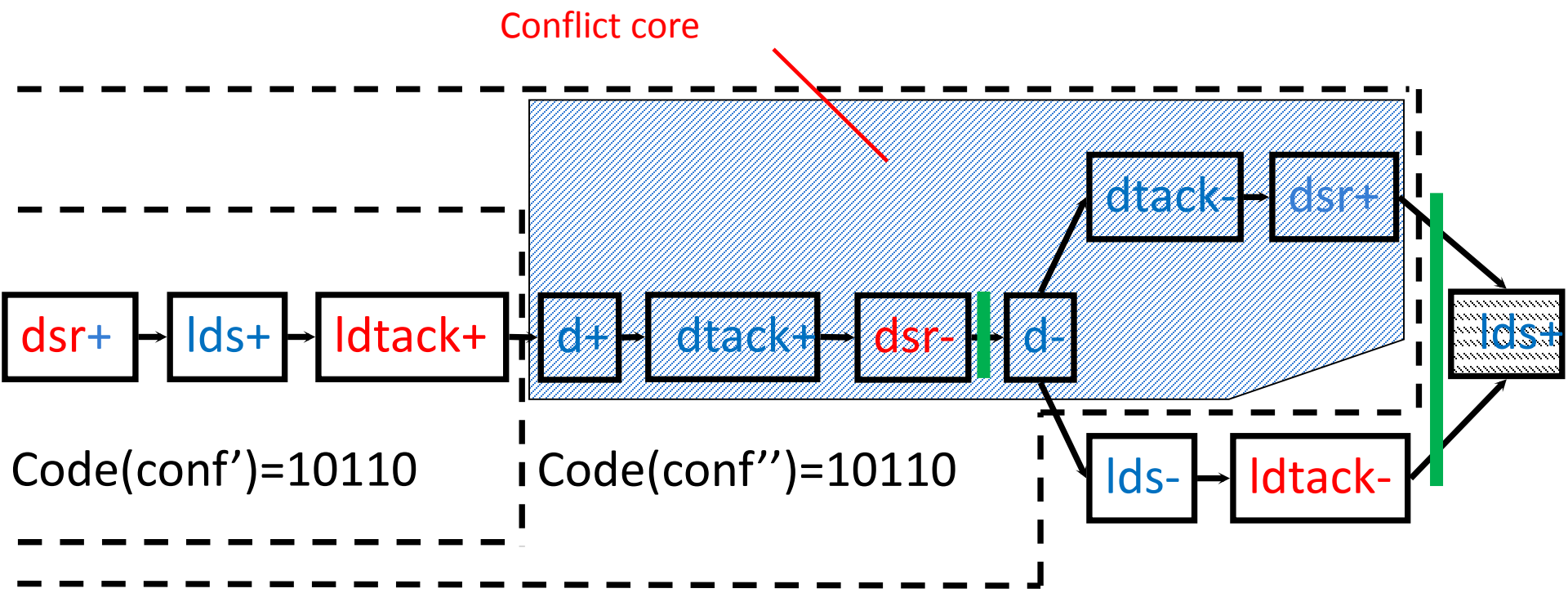
Example: VME Bus Controller



Example: CSC conflict



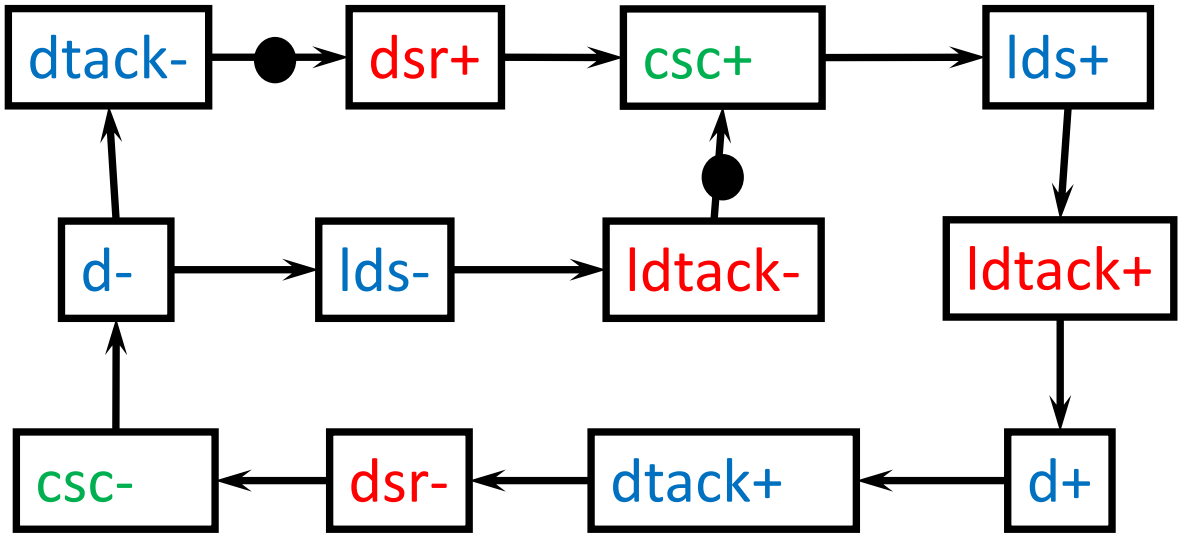
Example: Resolving the conflict



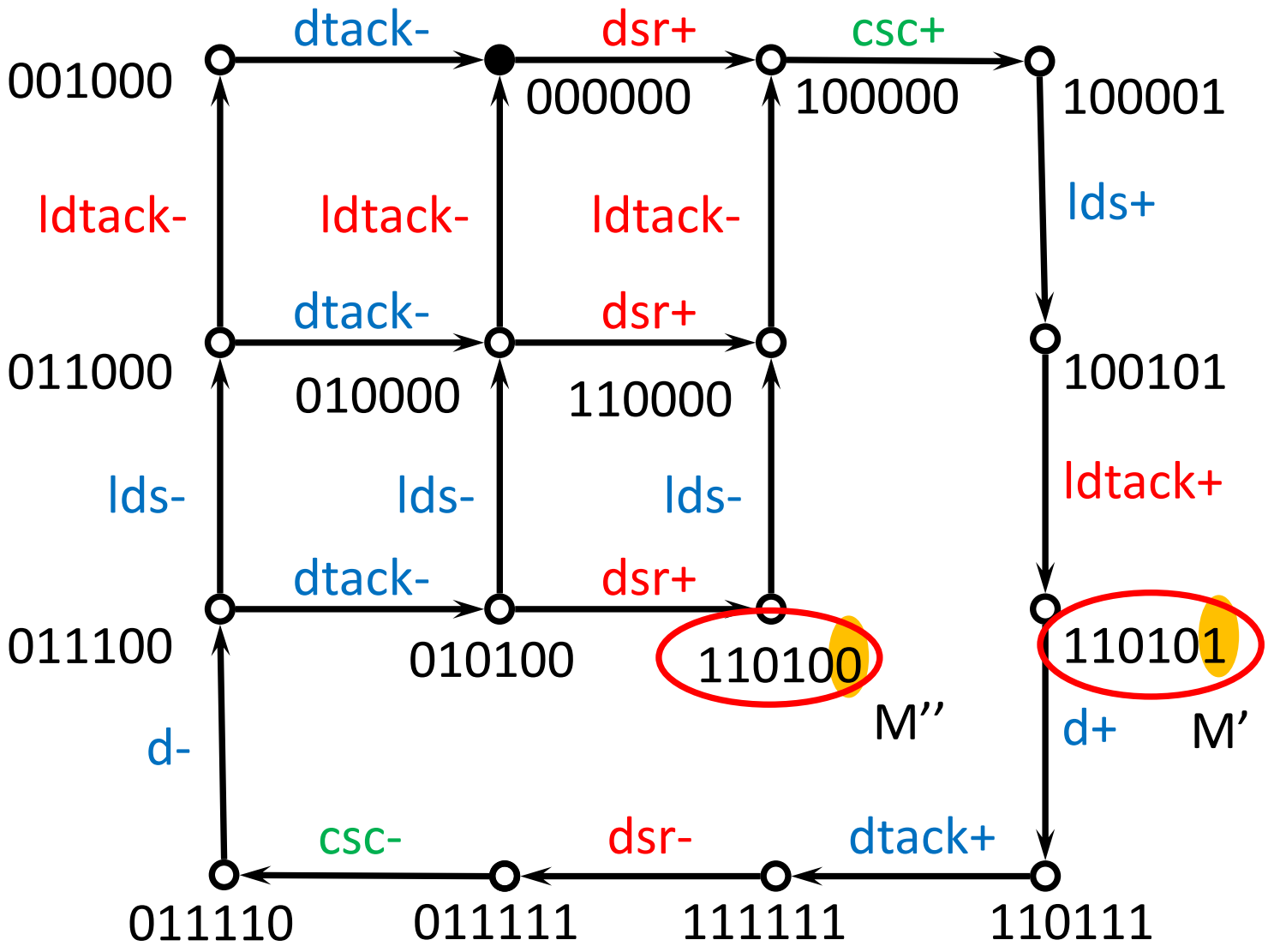
Idea: **Insert csc+ into the core and csc- outside the core to break the balance**

Note: **Cannot delay inputs!**

Example: Resolving the conflict

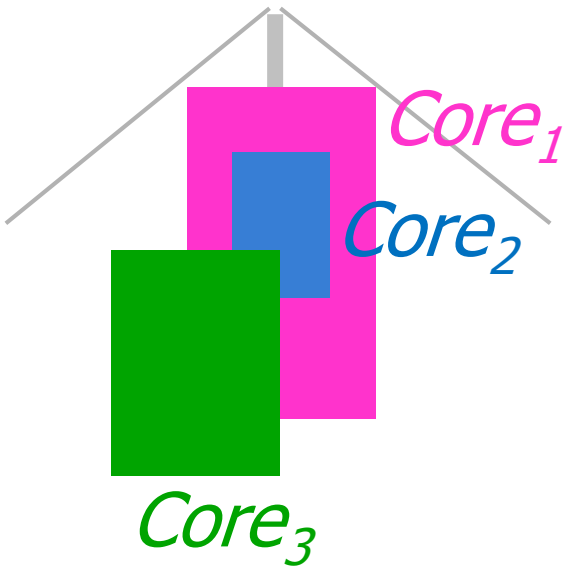


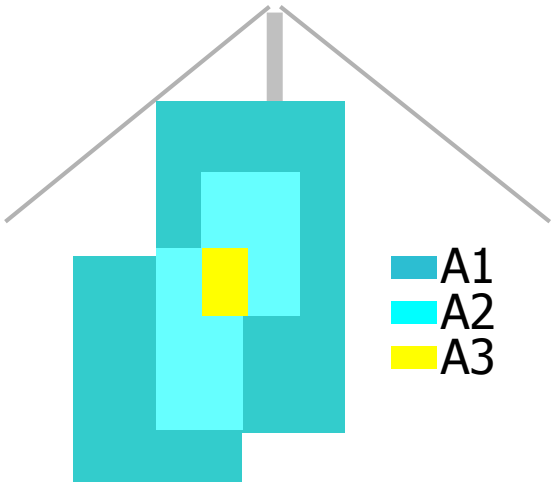
Example: Resolving the conflict

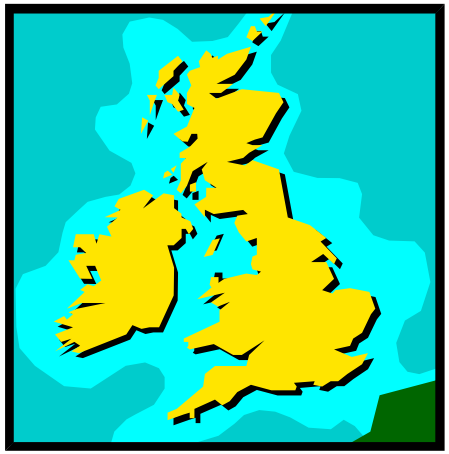


Core map

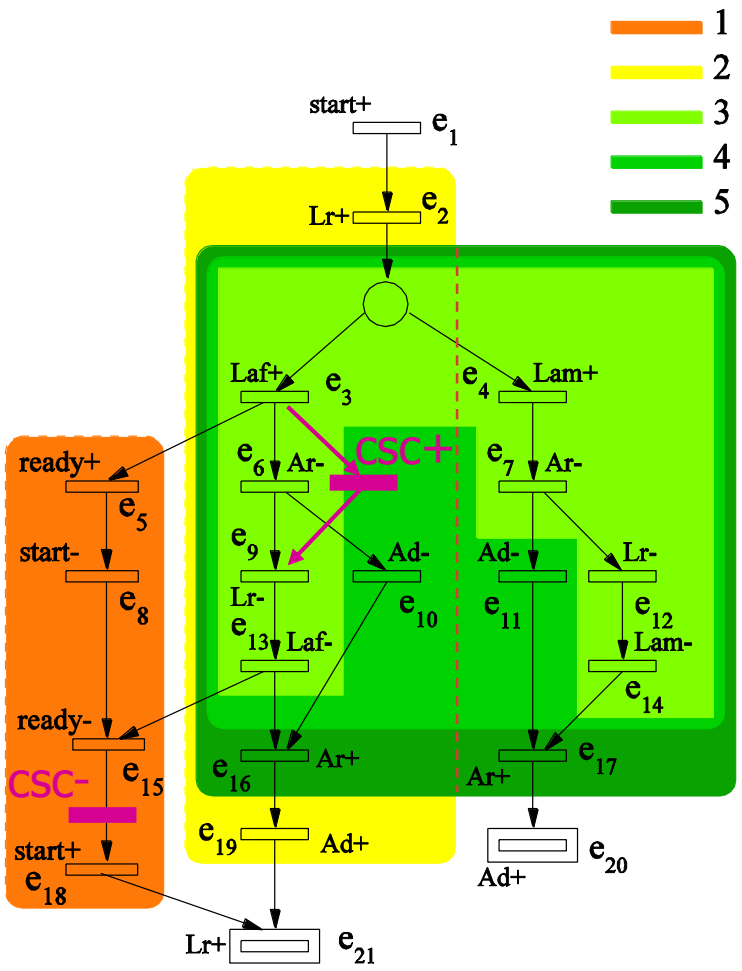
- **Cores often overlap**
- **High-density areas are good candidates for signal insertion**
- **Analogy with topographic maps**







Example: core map



Concurrency reduction

Introduces a new arc in the STG: **a** → **b**

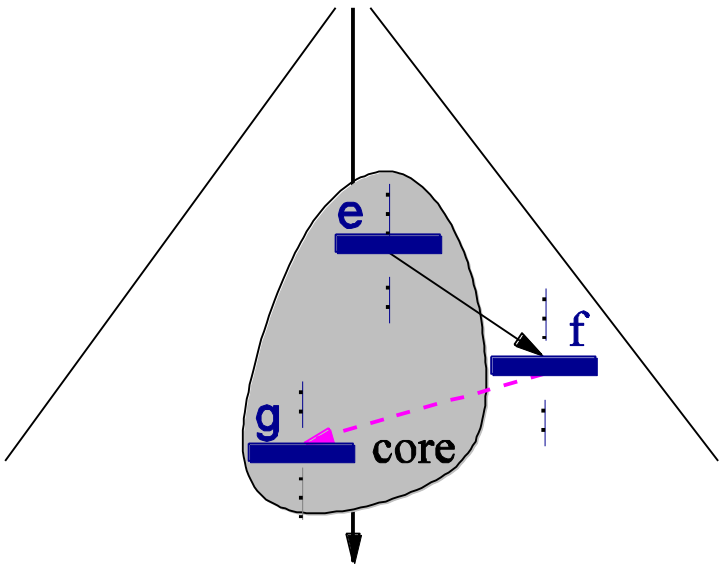
Note: Must not delay inputs, i.e. **b** cannot be an input!

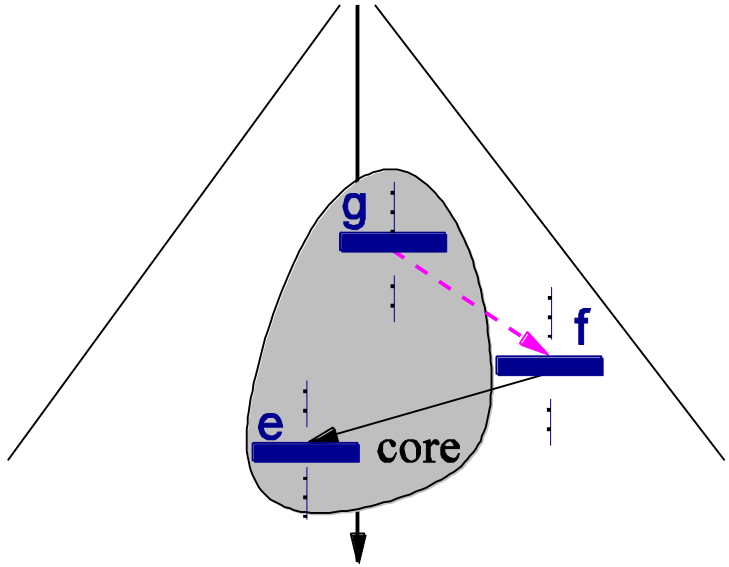
Note: Changes the behaviour, impacts the environment!

Heuristic: Try not to introduce new triggers of **b**, e.g. if there is an arc **a+** → **b+** then **a-** → **b-** is preferred

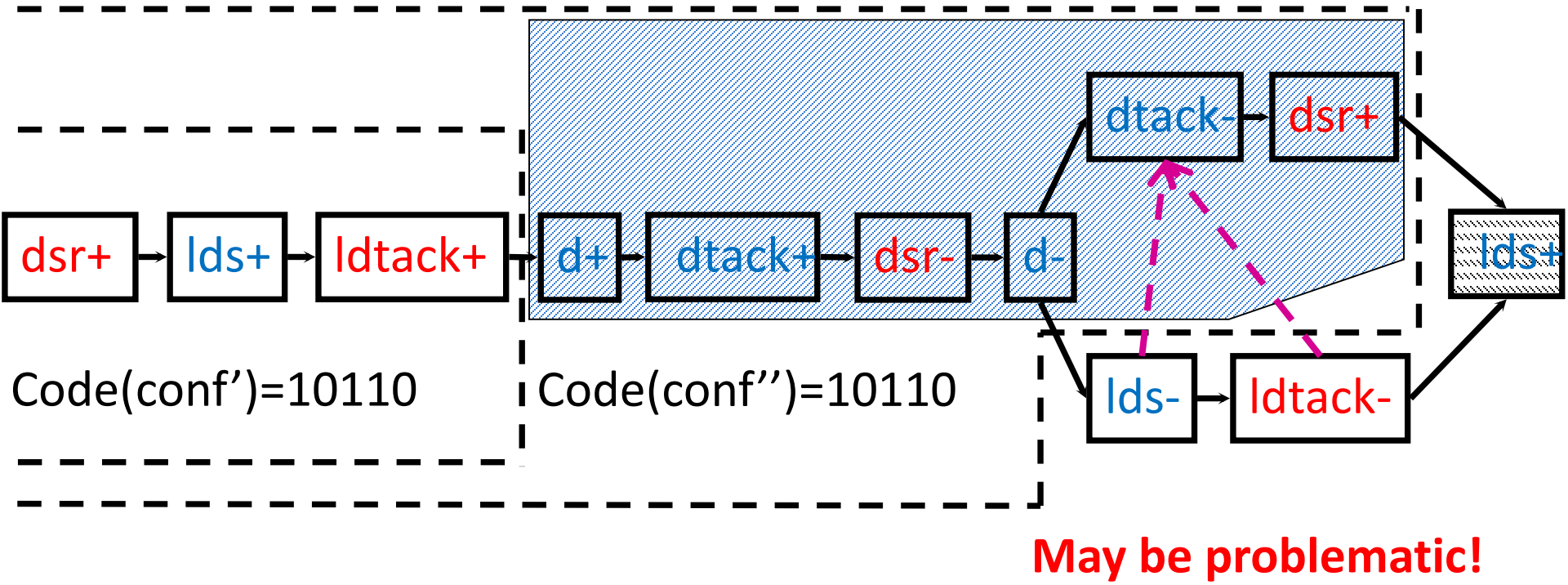
Used for resolving CSC conflicts and circuit simplification

'Drag' some events into the core to break the balance:



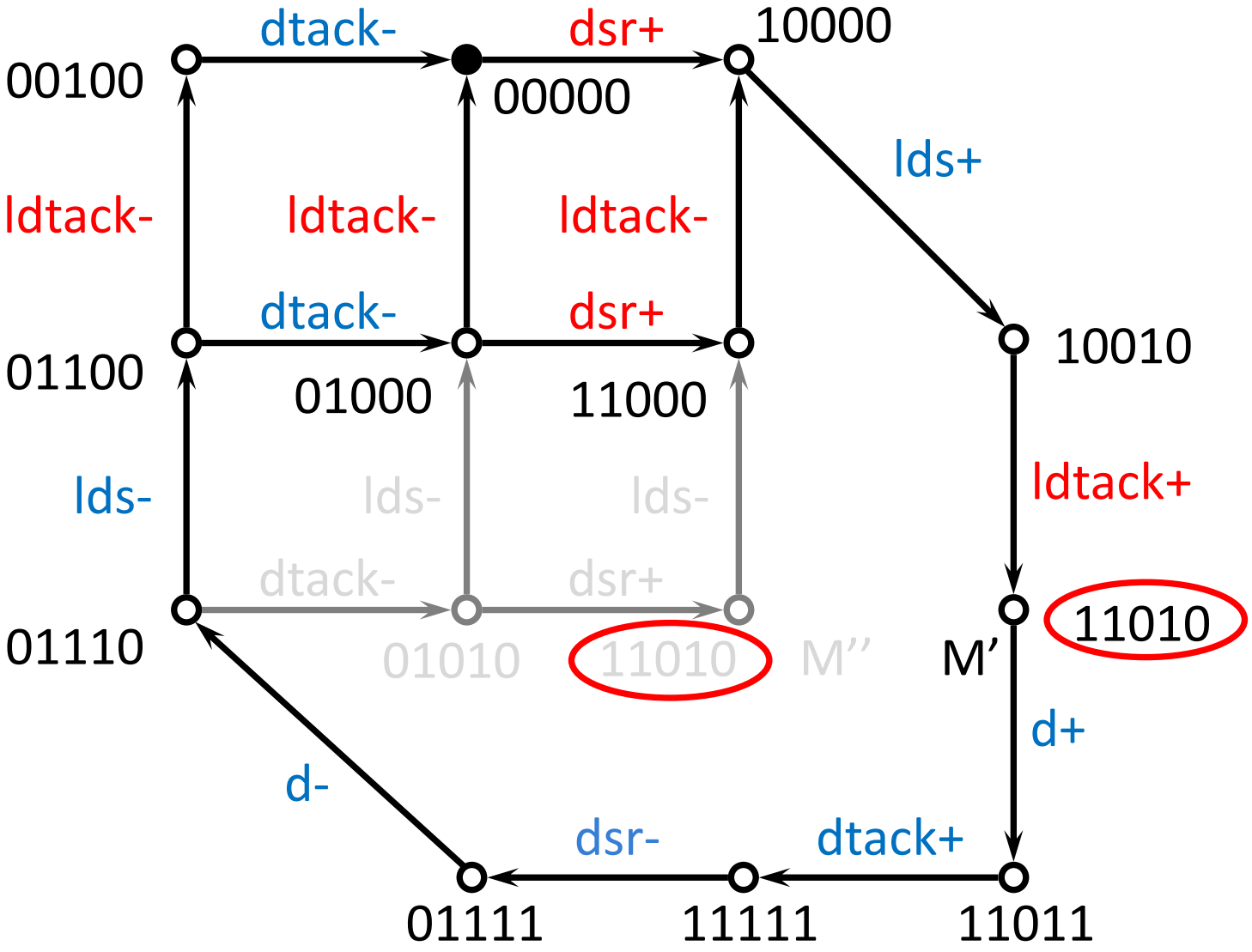


Example: Resolving the conflict



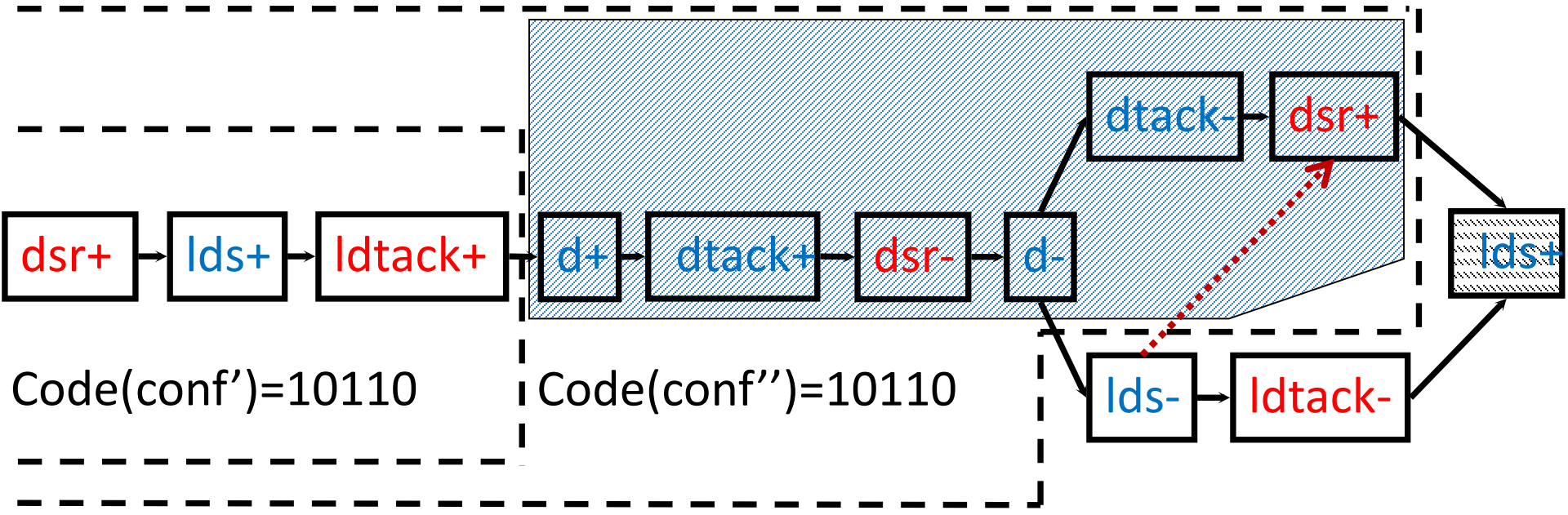
May be problematic!

Example: Resolving the conflict

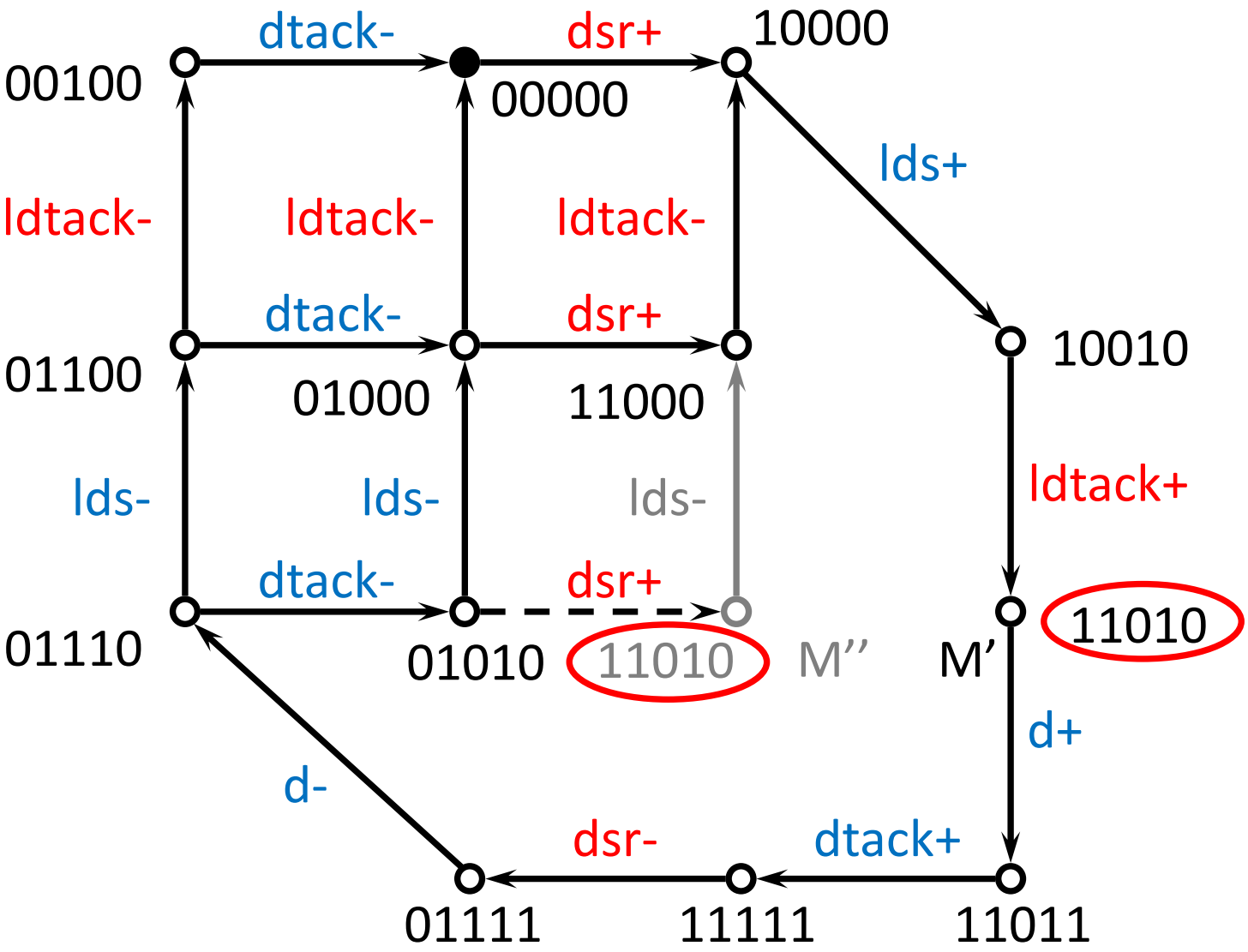


Relative timing assumptions

- **“This event will happen faster than that one”**
- **Break speed-independence, and generally problematic**
- **Similar to concurrency reductions, but the introduced arcs are special, in particular they don’t trigger signals**
- **Can “delay” inputs**



Example: Resolving the conflict



Comparison of the methods

- Signal insertions – paracetamol
  - ☺ behaviour is preserved
  - ☹ inserted signals have to be implemented
- Concurrency reductions – antibiotic
  - ☺ no new signals
  - ☺ reduced state graph and so more don't-cares in minimisation tables
  - ☹ change the behaviour: need to be careful if input → output (even indirectly) – this puts a new assumption on the environment!
  - ☹ can introduce deadlocks: Circuit: a → b & Environment: b → a
- Timing assumptions – surgery
  - ☺ no new signals
  - ☺ reduced state graph and so more don't-cares in minimisation tables
  - ☹ break speed-independence
  - ☹ require deep understanding of theory and the circuit's behaviour
  - ☹ introduce layout constraints, and need extensive validation
  - ☹ fragile due to variability (manufacturing, temperature, voltage, etc.)

Formal Verification of Asynchronous Circuits

TWO kinds of verification

1. Verification of the STG specification
   - there is no circuit yet, just an STG specification
   - check if the STG makes sense
   - check if the STG can be implemented as an SI circuit
2. Verification of the circuit
   - given a gate-level implementation of a circuit and an STG modelling the behaviour of the environment, check if the circuit is correct

Verification of STG specification

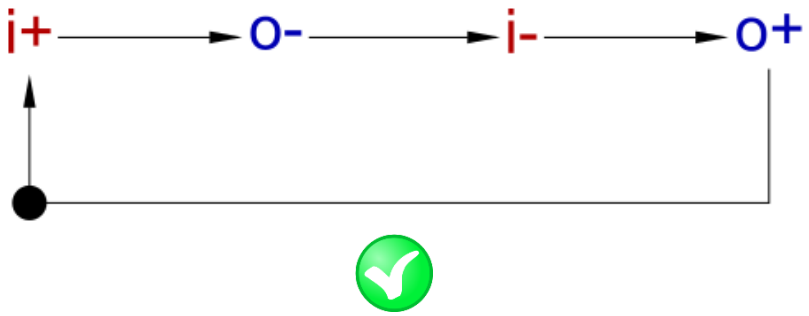
Standard PN properties:

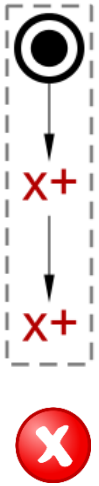
- **boundedness / safeness – a digital circuit has finitely many reachable states**
- **deadlock-freeness**
- **various custom reachability properties, e.g. mutual exclusion**

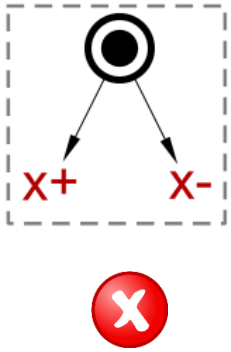
Verification of STG specification

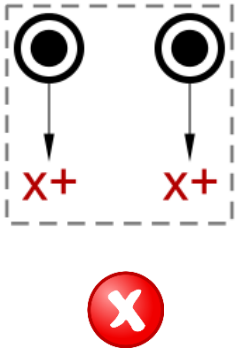
Consistency: in each execution, the rising and falling edges of each signal must alternate, always starting from the same edge – reduces to a reachability property

Intuition: at any reachable state the value of each signal is binary







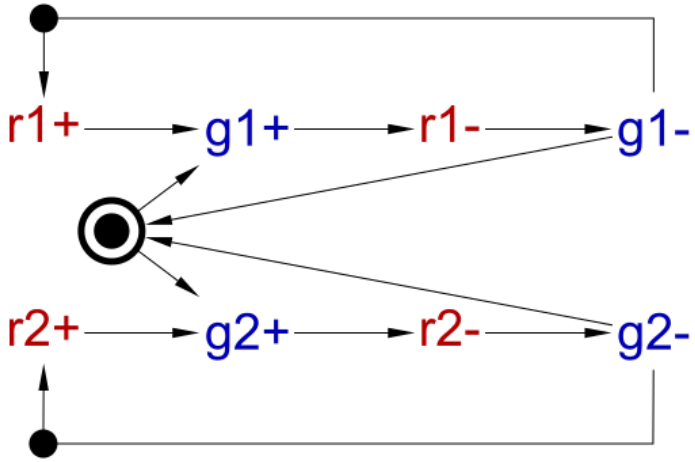


Verification of STG specification

Output-persistency: an enabled output must not be disabled by another signal firing first

Intuition: disabling and enabled output can lead to a non-digital pulse on the corresponding gate output

- **input / input choices: no OP violation, usually appear due to abstraction of the environment**
- **input / output choices: OP violation, very problematic – usually a mistake**
- output / output choices: OP violation, usually due to arbitration; implementable using a mutex – can be ‘factored out’ into the environment to ensure OP

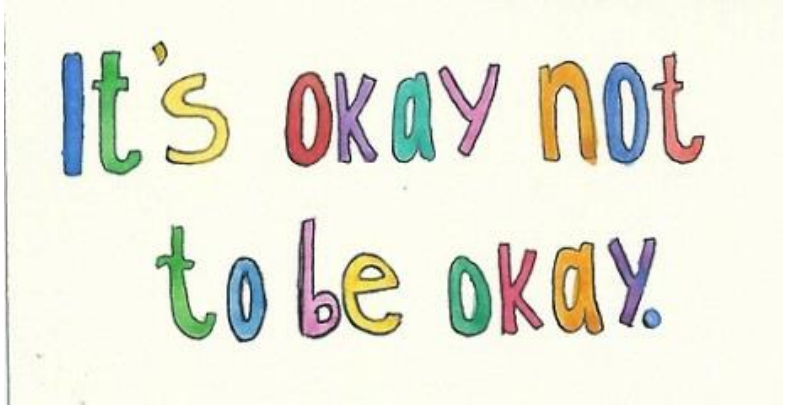


Verification of STG specification

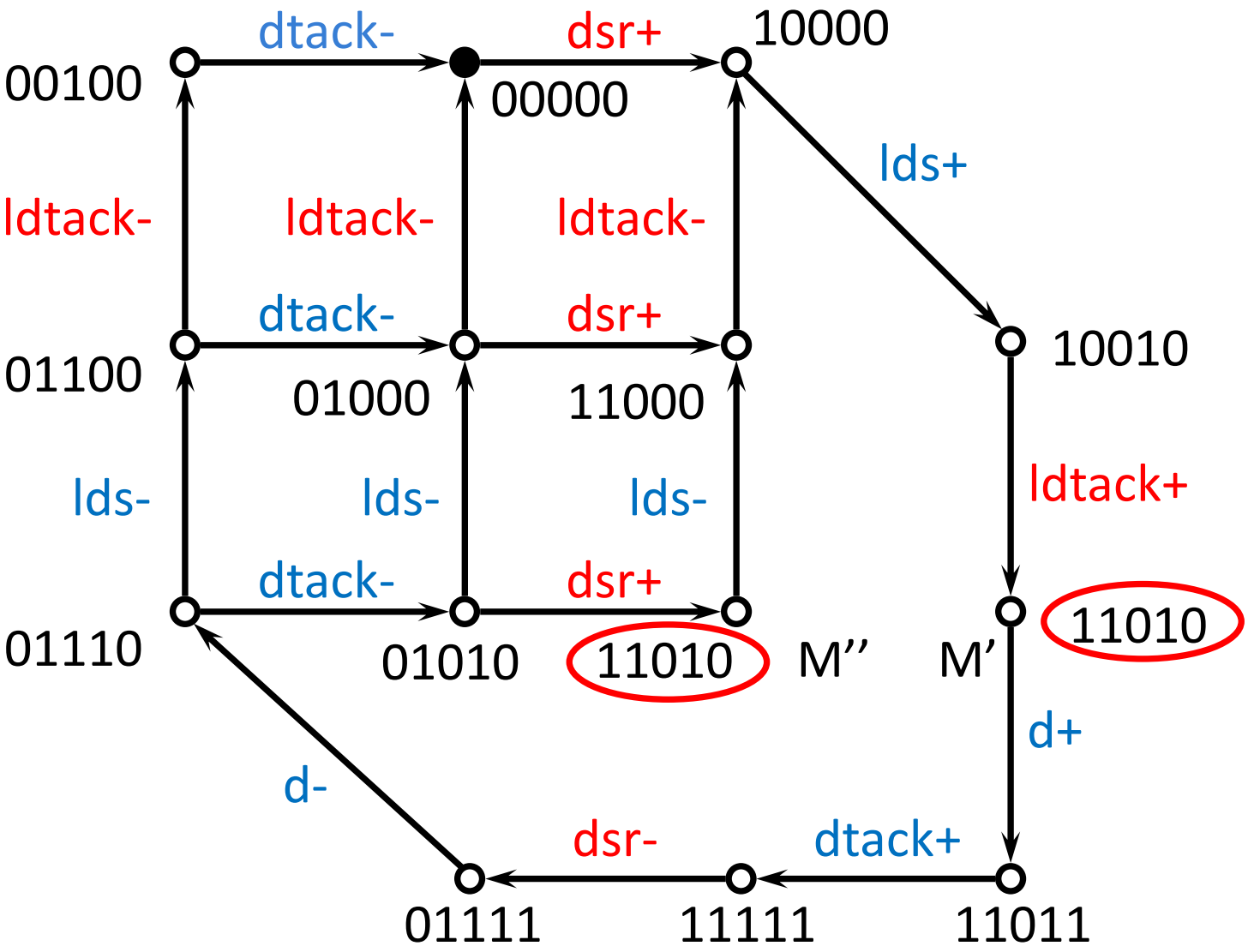
Complete State Coding (CSC): If two reachable states have the same values of all signals then they should enable the same outputs; two states violating this property are said to be in CSC conflict

Intuition: the circuit can only 'see' the signal values (not the tokens in the STG!), and these should be sufficient to determine which outputs to produce

Implementability property – CSC conflicts do not indicate that the STG is wrong; they can be resolved automatically



Example: CSC conflict

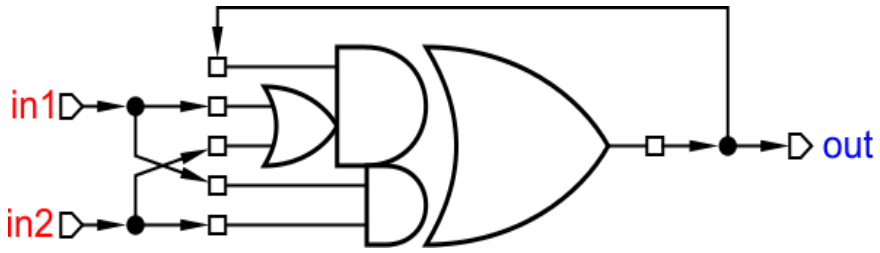


Verification of the circuit

Converting a gate-level circuit to an STG:

- Represent each signal **s** by two places, $p_{s=0}$ and $p_{s=1}$; exactly one of them is marked at any time, representing the current value of **s**
- Since there is no information about the environment's behaviour, it is taken to be the most general (i.e., it can always change the value of any input); this is modelled for each input signal **i** by adding transitions $p_{i=0} \rightarrow \mathbf{i+} \rightarrow p_{i=1}$ and $p_{i=1} \rightarrow \mathbf{i-} \rightarrow p_{i=0}$
- For each output **o** with the next-state function [**o**]=E, compute the set and reset functions $[\mathbf{o}\uparrow]=E|_{\mathbf{o}=0}$ and $[\mathbf{o}\downarrow]=\neg E|_{\mathbf{o}=1}$ as minimised DNF
- For each term m of the set function, add a transition $p_{\mathbf{o}=0} \rightarrow \mathbf{o+} \rightarrow p_{\mathbf{o}=1}$, and for each literal s (resp. $\neg$s) in m, connect **o+** to $p_{s=1}$ (resp. $p_{s=0}$) by a read arc; a similar process is used to define the transitions **o–** using the reset function

Example: modelling a C-element

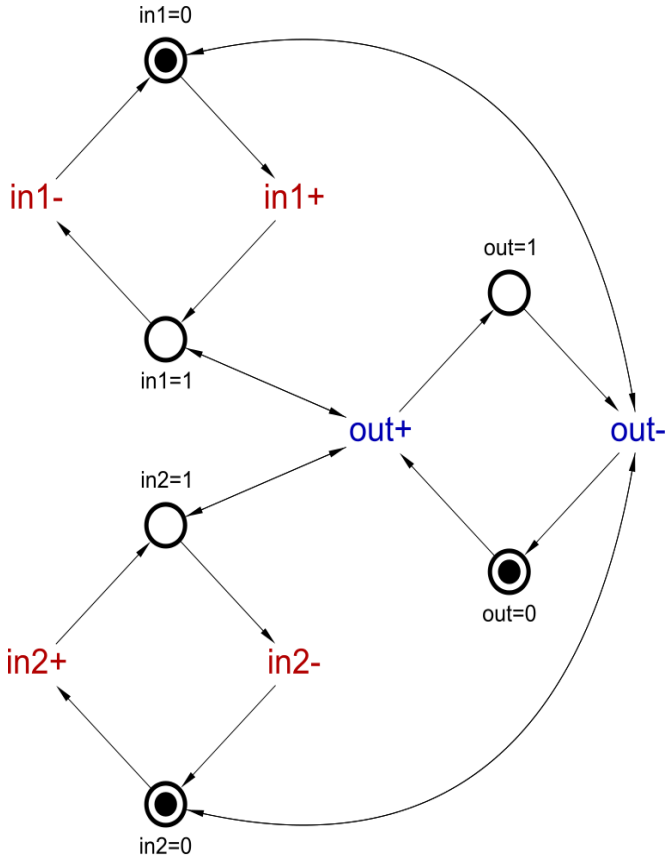


$[out] = out \cdot (in1 + in2) + in1 \cdot in2$

$[out\uparrow] = 0 \cdot (in1 + in2) + in1 \cdot in2 = in1 \cdot in2$

$[out\downarrow] = \neg(1 \cdot (in1 + in2) + in1 \cdot in2) =$
$\neg(in1 + in2 + in1 \cdot in2) = \neg(in1 + in2) =$
$\neg in1 \cdot \neg in2$

- This PN has more behaviour than the specification of C-element
- Not output-persistent: after in1+ in2+ the output **out+** can be disabled by in1- or in2-, i.e. there is a **hazard**
- This is because the circuit (and thus this STG) lacks information about the environment's behaviour!
- The circuit works correctly in an environment that fulfils the original contract



Gate-level modelling: Verification

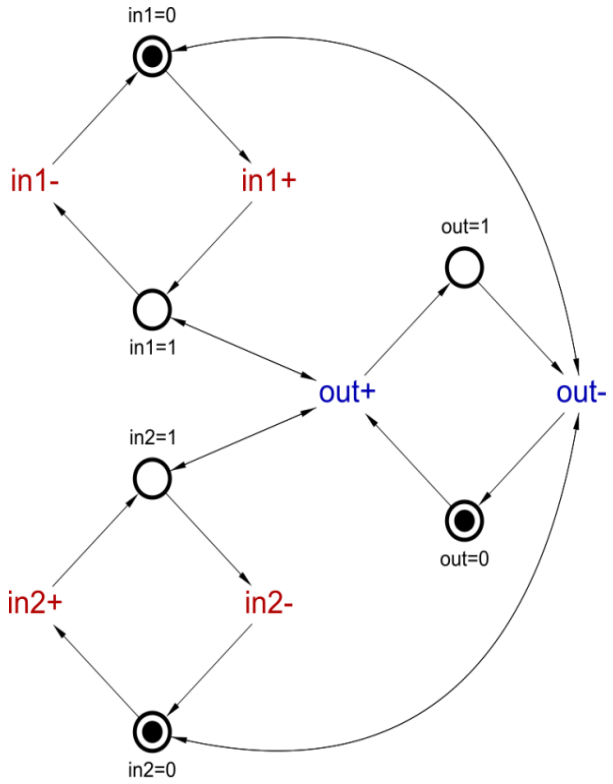
Gate-level circuit has no information about its environment, so naïve verification will always reveal hazards in any non-trivial circuit with inputs. Hence need to supply the environment's behaviour during verification: Assuming the environment fulfils the contract, the circuit must:

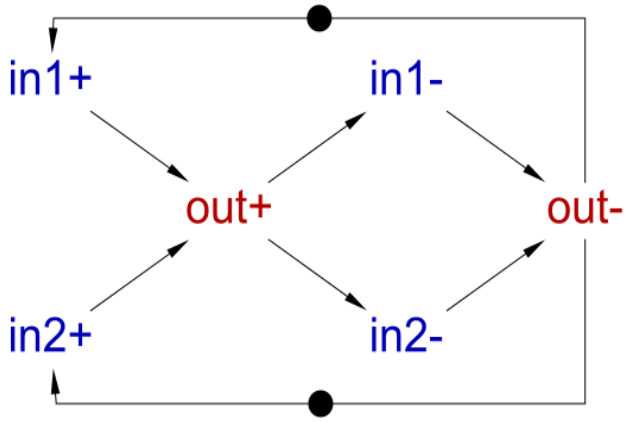
- be free from hazards: no output can be disabled by another signal (except in mutex)
- conform to its environment, i.e. never produce an unexpected output – the circuit must fulfil its contract too
- be deadlock-free
- etc.

Gate-level modelling: Verification

Problem: how to restrict the behaviour of the circuit by the behaviour of the environment to verify the properties?

Idea: use parallel composition! First, convert the circuit into an STG and then compose the latter with the **mirror** (i.e. inputs and outputs are swapped) of the original STG spec:

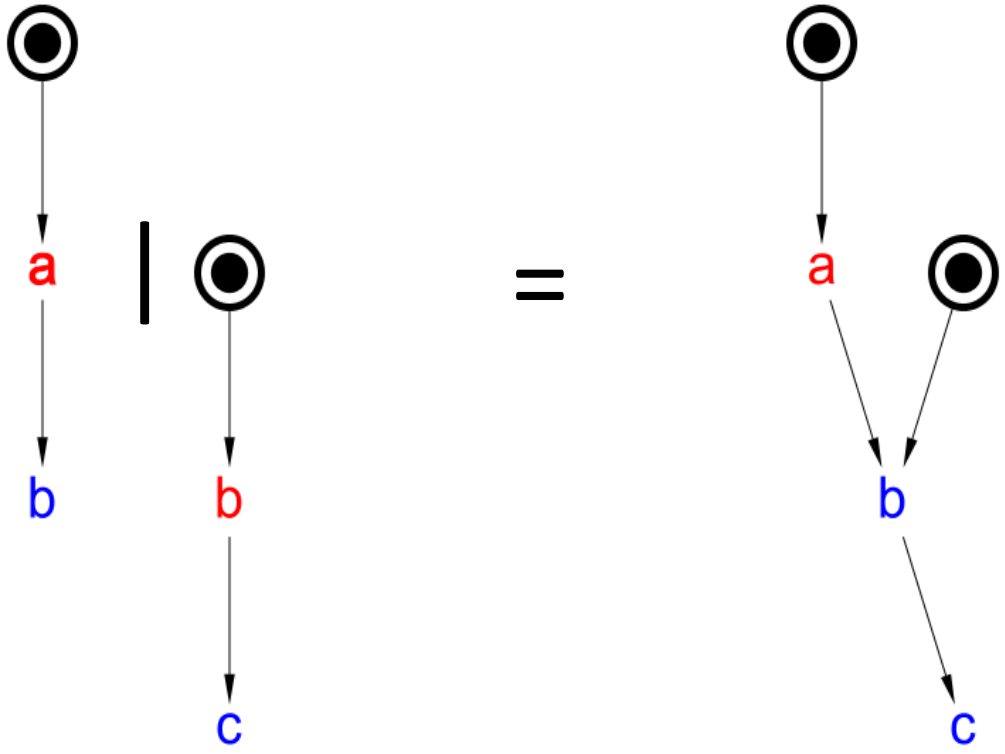




mirror

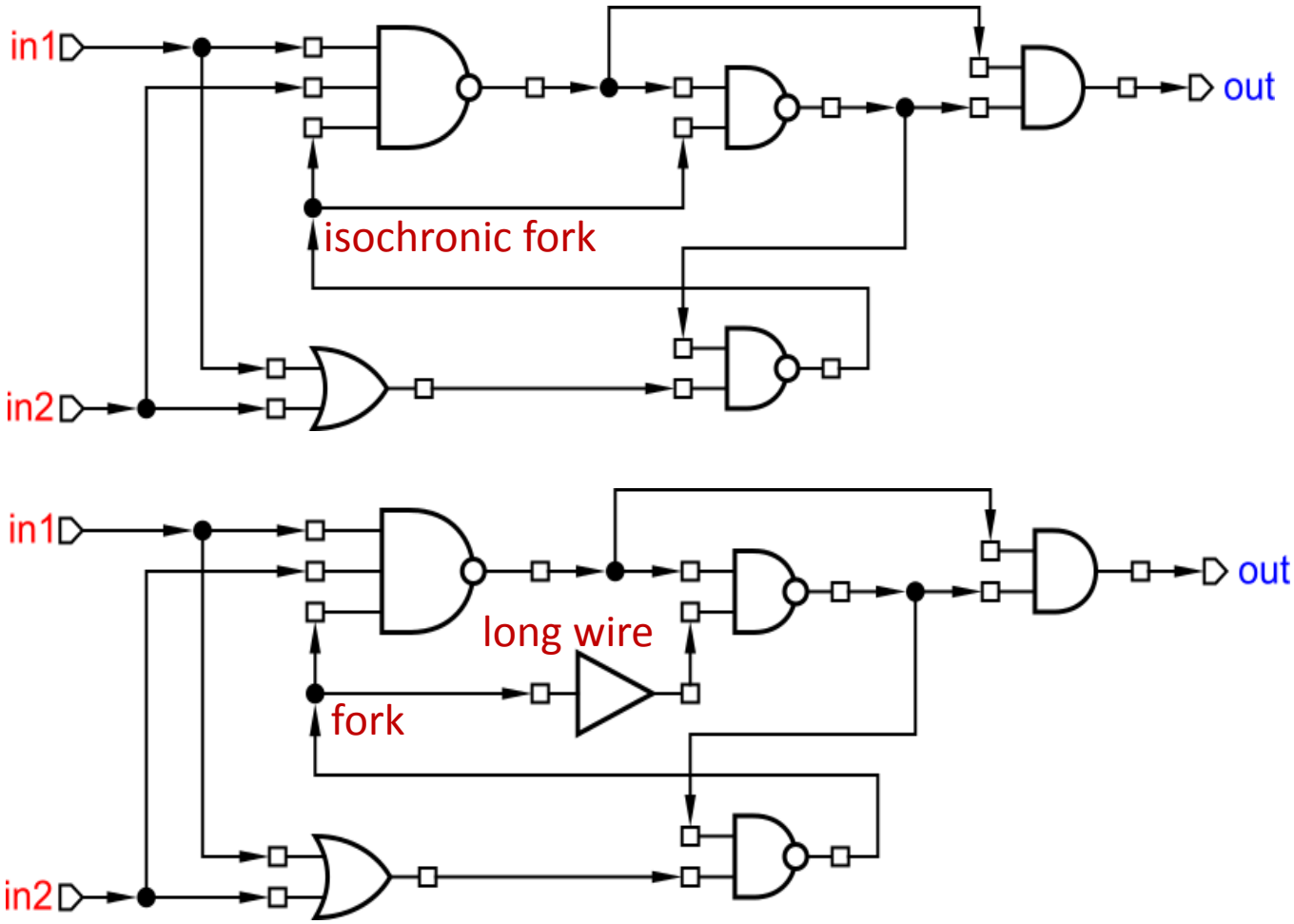
Parallel composition

- **Idea: Fuse transitions from different STGs that have the same label (if STGs have several transitions with the same label, fuse each such transition in STG_1 with each such transition in STG_2)**
- **Example:**



Example: C-element

Can a C-element be implemented by the following circuits?



Under the Bonnet of Workcraft

PUNF – parallel unfolder

- Tool for building Petri net unfoldings
- Utilises multiple processor cores
- Unfoldings alleviate the state space explosion problem – the number of reachable states is generally exponential in the size of the specification
- Works very well for asynchronous circuits due to high concurrency and small number of choices – an ideal case for unfoldings

MPSAT – verification and synthesis

- Uses PUNF-generated prefixes as an input – completely avoids state graph
- Employs a SAT solver for efficiency
- Verifies many relevant properties, like deadlocks, CSC, etc.
- Supports REACH – a language to specify custom properties
- Synthesis: CSC resolution, deriving complex-gate, gC, stdC implementations, logic decomposition

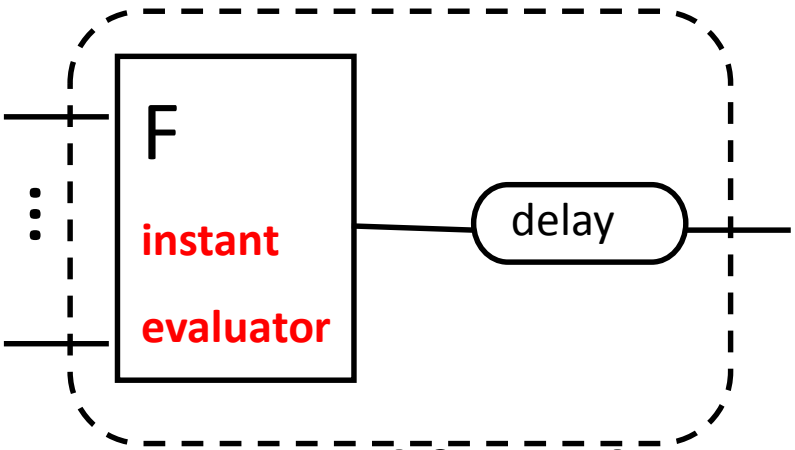
PCOMP – parallel composition

- **Composes several STGs, optionally hiding the internal communication, e.g.:**
  - **to compose several modules into one**
  - **to compose a circuit with its environment for verification**

Logic Synthesis and Implementation Styles in Asynchronous Circuits Design

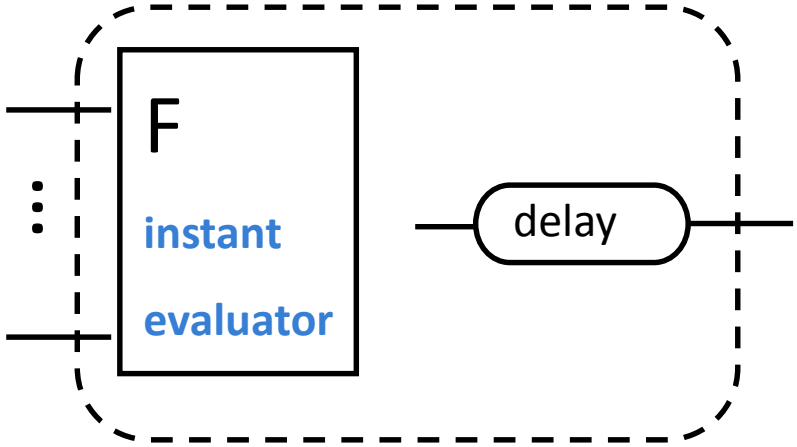
Speed-independence assumptions

- **Gates/latches are *atomic* (so no internal hazards)**

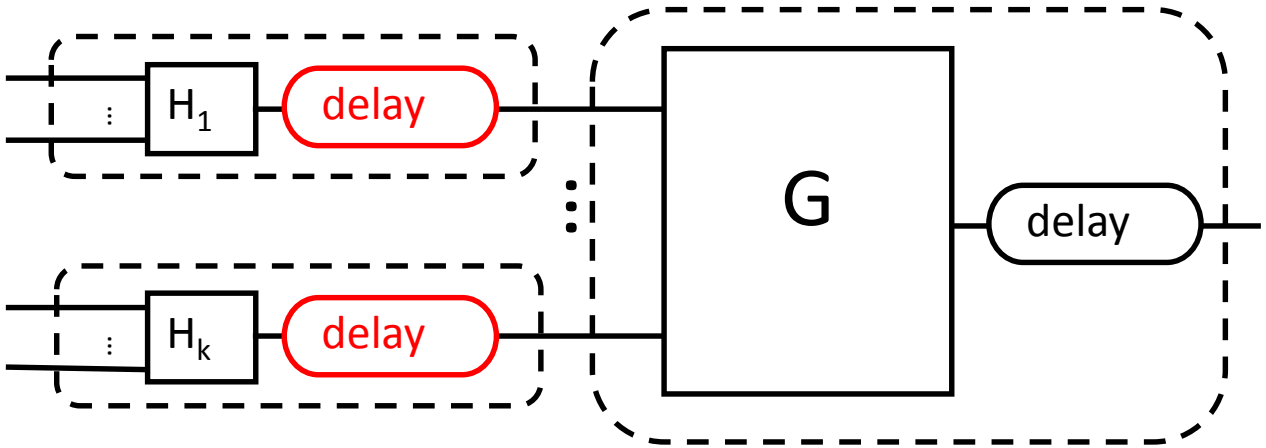


- **Gate delays are positive and finite, but variable and unbounded**
- **Wire delays are negligible (SI)**
- **Alternatively, [some] wire forks are isochronic (QDI), i.e. wire delays can be added to gate delays**

SI decomposition



Hazards can be introduced due to these delays!



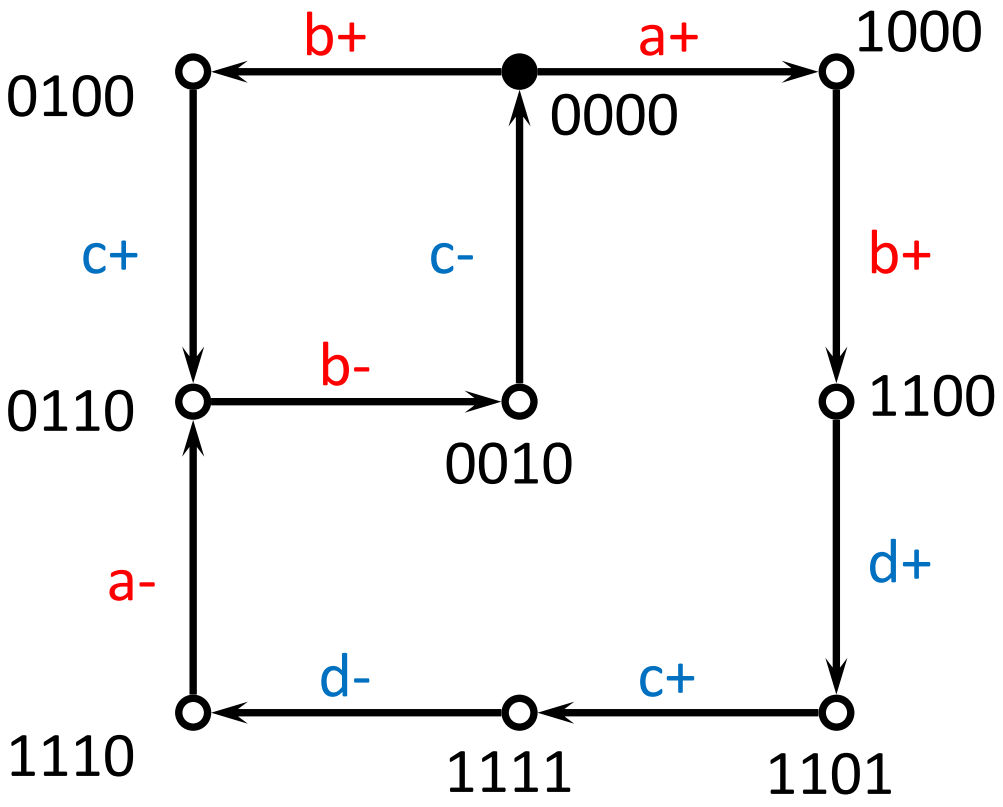
Gates & latches

- Good citizens: unate gates/latches, e.g. BUFFER, AND, OR, NAND, NOR, AND-OR, OR-AND, C-element, SR-latch, RS-latch
  - Output inverters ('bubbles') can be used liberally, e.g. NAND, NOR, as the invertor's delay can be added to the gate's delay
  - Input inverters are suspect as they introduce delays, but in practice are ok if the wire between the inverter and the gate is short
- Suspects: binate gates, e.g. XOR, NXOR, MUX, D-latch – may have internal hazards, but may still be useful

Logic synthesis

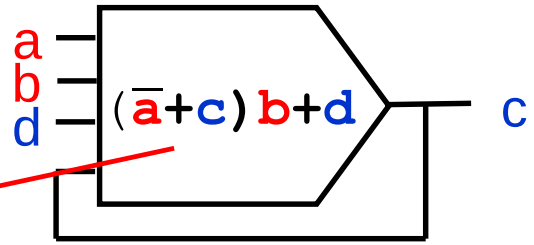
- Encoding (CSC) conflicts must be resolved first
- Several kinds of implementation can then be derived automatically:
  - complex-gate (CG)
  - generalised C-element (gC)
  - standard-C implementation (stdC)
- Can mix implementation styles on per-signal basis
- Logic decomposition may still be required if the gates are too complex

Example: complex-gate synthesis



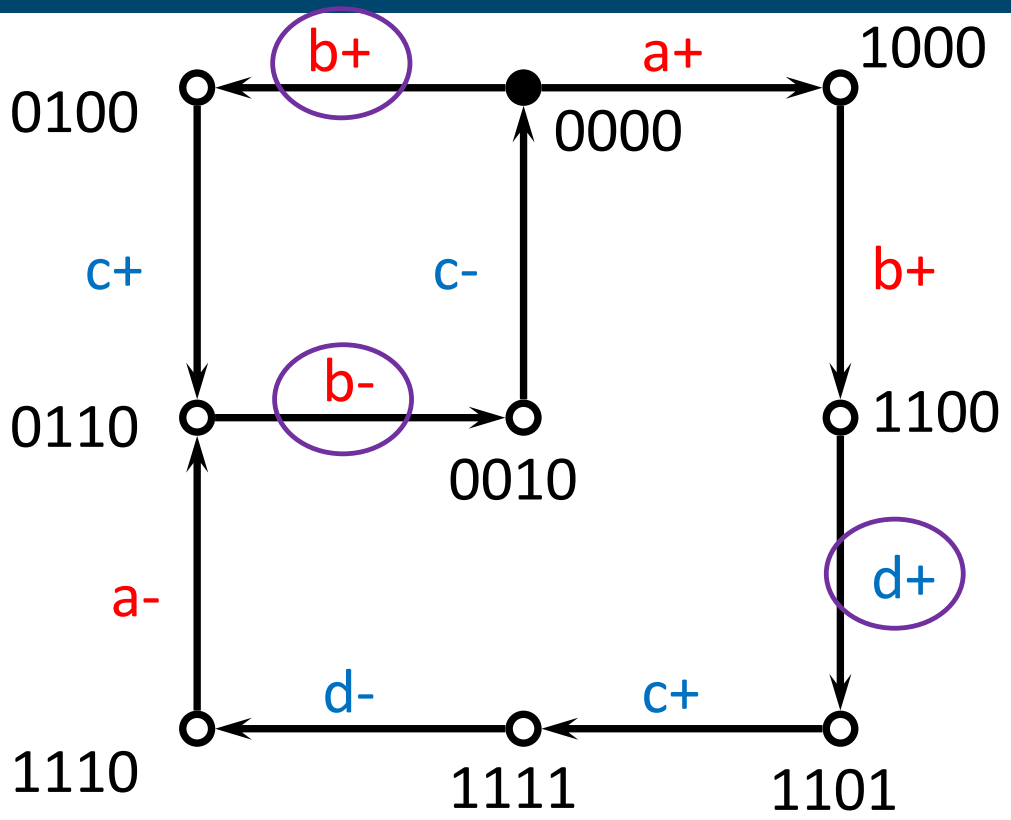
Code	Nxt_c
0100	1
0000	0
1000	0
0110	1
0010	0
1100	0
1110	1
1111	1
1101	1
else	-
Eqn	$(\bar{a}+c)b+d$

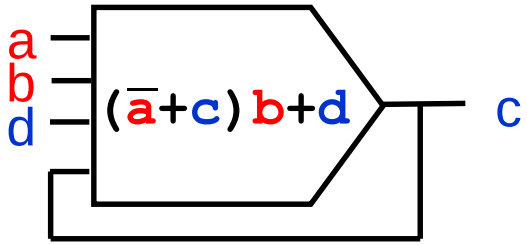
$$Nxt_z(s) = Code_z(s) \oplus Out_z(s)$$



The size of this Boolean expression is not limited!

Support, triggers and context





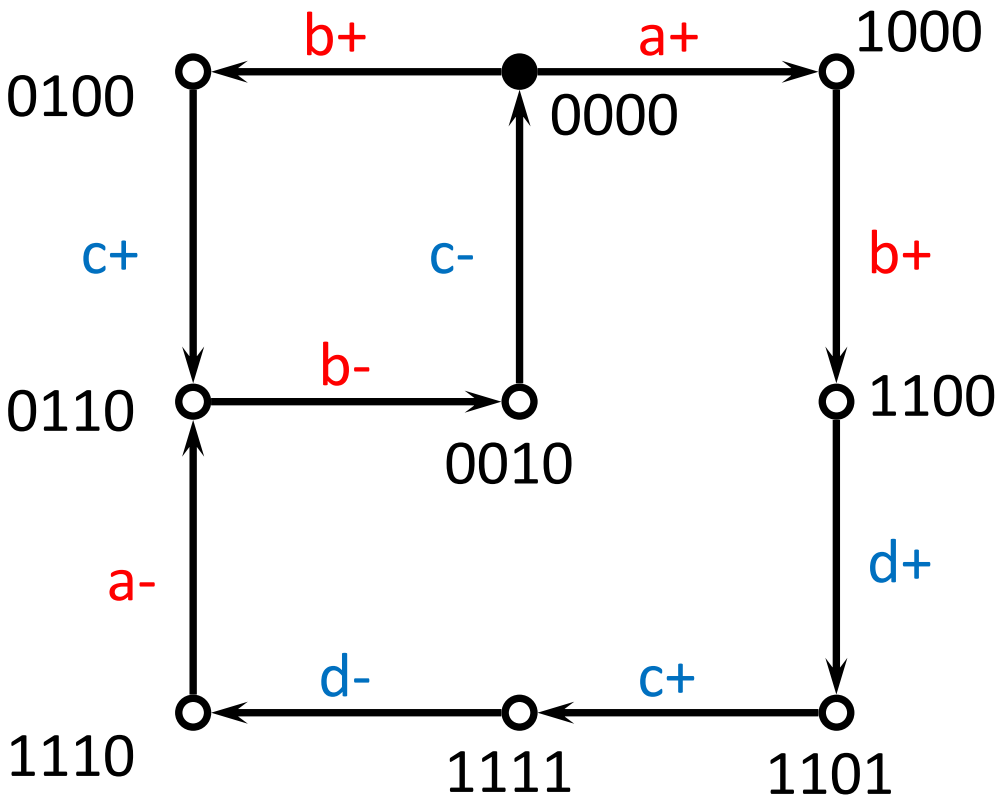
Signals that are the inputs of the gate producing a signal form its **support**, e.g. the support of c is {a,b,c,d}. Supports are not unique in general.

Signals whose occurrence can immediately enable a signal are called its **triggers**, e.g. the triggers of c are {b,d}. Triggers are unique, and are always in the support.

Signals in the support which are not triggers are called the **context**, e.g. the context of c is {a,c}. Context is not unique in general.

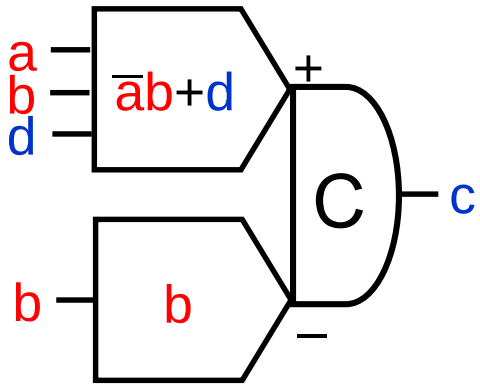
support = triggers + context

Example: gC implementation



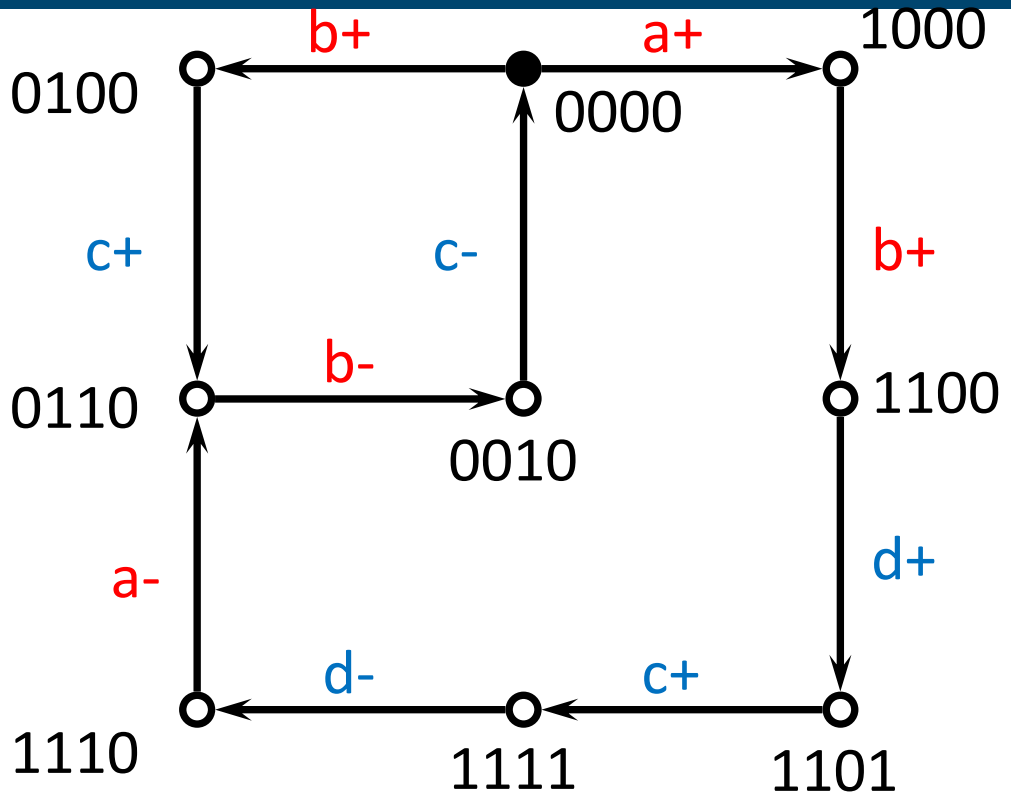
Code	Set_c	$Reset_c$
0100	1	0
0000	0	–
1000	0	–
0110	–	0
0010	0	1
1100	0	–
1110	–	0
1111	–	0
1101	1	0
else	–	–
Eqn	$\bar{a}b+d$	$\bar{b}$

$$Set_z(s) = \begin{cases} 1 & \text{if } Out_{z+}(s) = 1 \\ 0 & \text{if } Nxt_z(s) = 0 \\ - & \text{otherwise} \end{cases} \qquad Reset_z(s) = \begin{cases} 1 & \text{if } Out_{z-}(s) = 1 \\ 0 & \text{if } Nxt_z(s) = 1 \\ - & \text{otherwise} \end{cases}$$

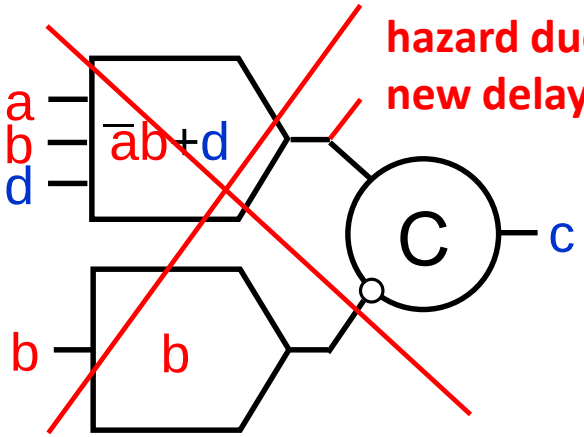


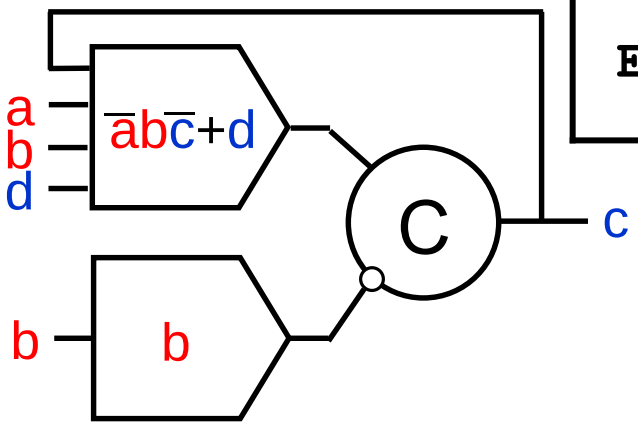
Implemented as pull-up and pull-down networks of transistors and a ‘keeper’; assumed to be atomic

Example: stdC implementation



Code	Set_c	$Reset_c$
0100	1	0
0000	0	–
1000	0	–
0110	–	0
0010	0	1
1100	0	–
1110	–	0
1111	–	0
1101	1	0
else	–	–
'Monotonic cover' constraints		
Eqn	$\bar{a}b\bar{c}+d$	$\bar{b}$





Logic Decomposition

- Often complex-gates are too complex to be mapped to a gate library, and so logic decomposition is required
- Cannot naïvely break up complex-gates – this is likely to introduce hazards (at least, timing assumptions are required)
- Decomposition is one of the most difficult tasks – no guarantee that automatic decomposition will succeed
- Manual changes in the STG may be required:
  - identify the most complex gates
  - try some concurrency reductions
  - try to decompose your circuit into smaller blocks
  - ‘be creative’