# Design Automation for Analog-Mixed Signal Circuits with Asynchronous Control

Vladimir Dubikhin(*), Victor Khomenko (*),
Andrey Mokhov(*), Chris Myers (**),
Danil Sokolov(*), Alex Yakovlev (*)

(*)Newcastle University, UK
(**)The University of Utah, USA
Contact: Alex.Yakovlev@ncl.ac.uk
async.org.uk; workcraft.org

# Agenda

- Introduction:
  - Motivation, Challenges, Shortcomings of commercial flows
- Part 1. A4A: Asynchronous design for analogue electronics
  - Basics of asynchronous design
  - Design flow for A4A: formal specification, circuit synthesis, verification
  - Examples: multiphase buck, SRAM, ADC

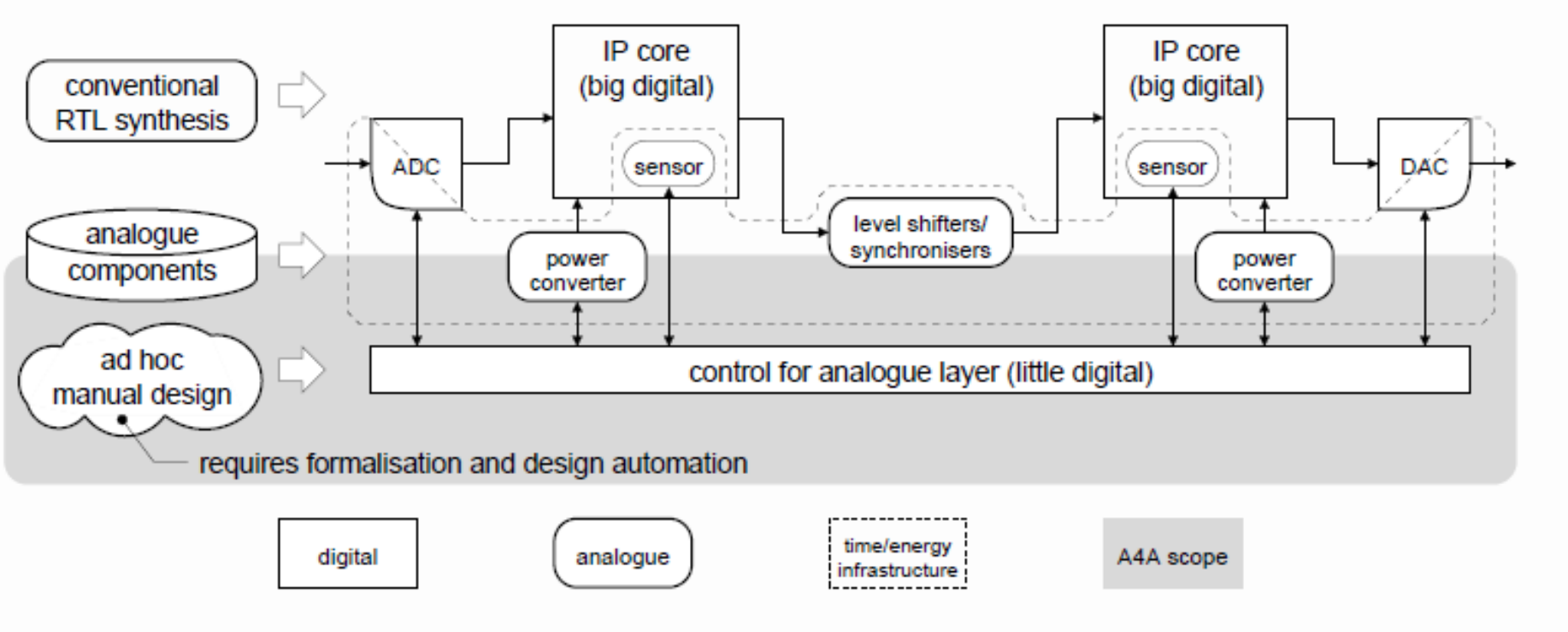
<Break>

- Part 2. AMS design with asynchronous control
  - Analogue verification with LEMA
  - Co-optimization flow: Workcraft and LEMA
  - Examples: C-element, Buck
- Part 3. Workcraft tools demo
- Discussion

# Motivation

- **Analog and Mixed Signal (AMS) design becomes more complex:**
  - **More functionality**
  - **Move to deep submicron after all!**
- **According to Andrew Talbot from Intel, recently speaking at the AMS workshop at RAL, *"transistors are very fast switches, netlists are huge, parasitics are phenomenally difficult to estimate, passives don't follow Moore's law, reliability is a brand new landscape"***

# Emergence of “little digital”




- **There is a strong drive for having more digital parts in AMS**
- **Analog and digital are often intertwined**
- **Asynchronous design appears good for little digital**

**–> A4A project (EPSRC, Dialog supports)**

# Motivation: power electronics context

- **Efficient implementation of power converters is paramount**
  - **Extending battery life for mobile gadgets**
  - **Reducing energy bill for PCs and data centres (5% and 3% of global electricity production, respectively)**
- **Need for responsive and reliable control circuits – *little digital***
  - **Millions of control decisions per second for years**
  - **A wrong decision may permanently damage the circuit**

# Motivation: EDA support is a challenge

- **Poor EDA support**
  - **Mostly supports flow from schematic capture; lacks flow from behavioural capture**
  - **Synthesis from behavioural (RTL) is optimized for data processing logic and supports only synchronous – *big digital***
  - ***Manual* and *ad hoc* solutions are prone to errors and hard to verify (weeks of simulations)**
- **Big challenge is EDA for asynchronous (hence our A4A project)**
- **What do the Industrial gurus say?**

# Industry quotes

- "...analog has to budget five or six respins. Silicon has become the validation vehicle for analog, and that's a problem."

  Sandipan Bhanot, CEO of Knowlent

- "If the digital designers did verification the way analog designers do verification, no chip would ever tape out."

  Sandipan Bhanot, CEO of Knowlent

- "...problems are being solved because we have very good analog engineers. But in the future, if we want to improve time-to-market we will have to improve the tools."

  James Lin, VP at National Semiconductor

(Source "Why is analog so difficult?" – DACezine, January 2008)

# Analog design in digital context is hard

- **If digital parts don't use clock, they are normally designed by hand and require massive simulations:**
  - **E.g. analogue designers cannot afford simulating power converters from start-up; Instead they force it into known state**
  - **More specifically: 50 us of Spectre simulation time takes approx. 10 hours using 8 CPU cores**
  - **Hence they can only verify cherry-picked corners of digital functionality**

**(from Dialog Semiconductor, 2016)**

# Intel's advice on AMS Design

**Intel's advice:**

- Partition the Design to separate Analog and Digital
  - Give digital circuits to digital tools
  - Give analog circuits to analog tools
  - Do not pollute the hierarchy with logic gates or analog components!

(Source: Intel's talk about Holistic AMS design in Ultra-DSM at the May 2016 NMI event on AMS)

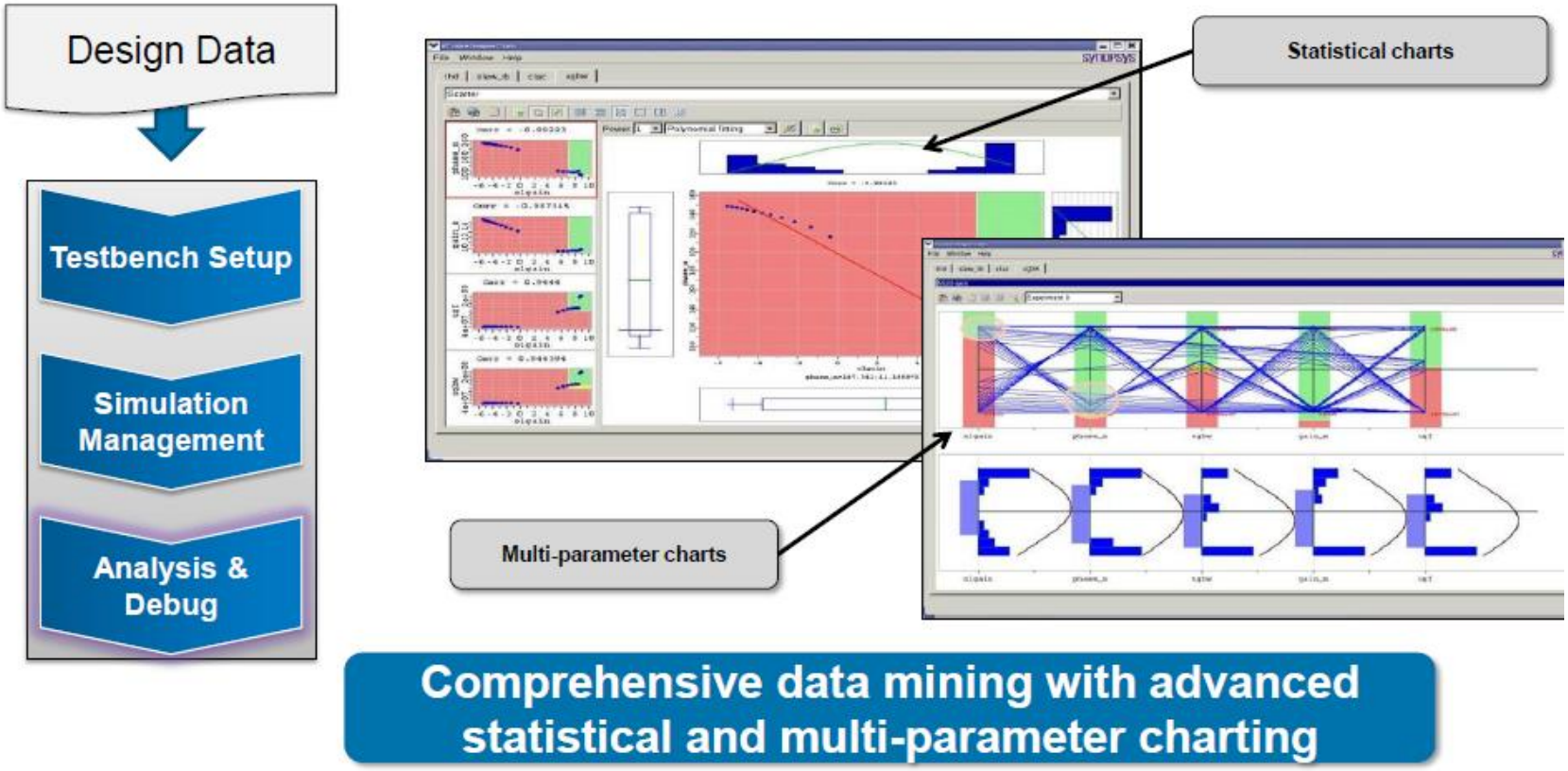
**But, how?**

**We must use Behavioural capture and drive verification from behavioural domain!**

# View from Synopsys

## Analysis and Debug

*Data mining*

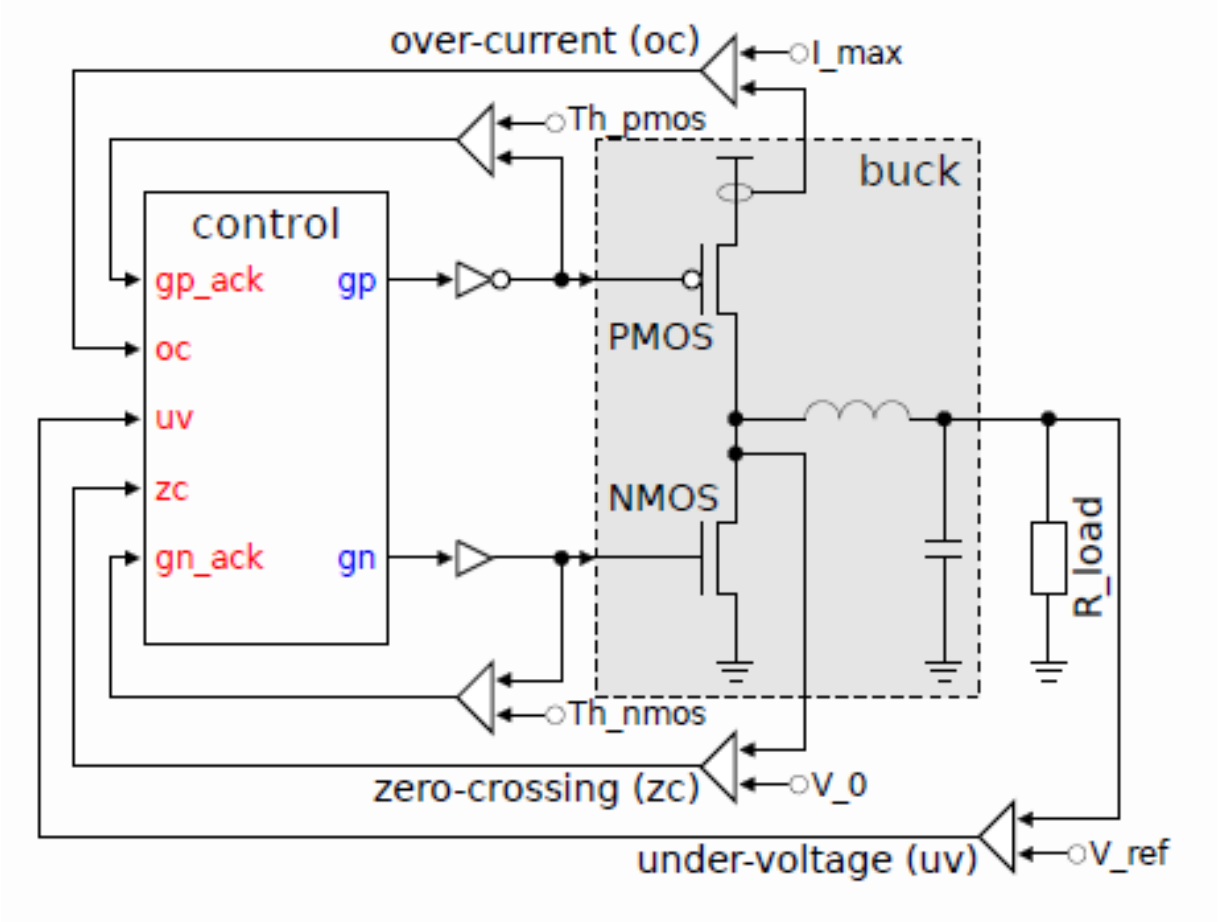



**Source: Damian Roberts, AMS Workshop, RAL, April 2016**

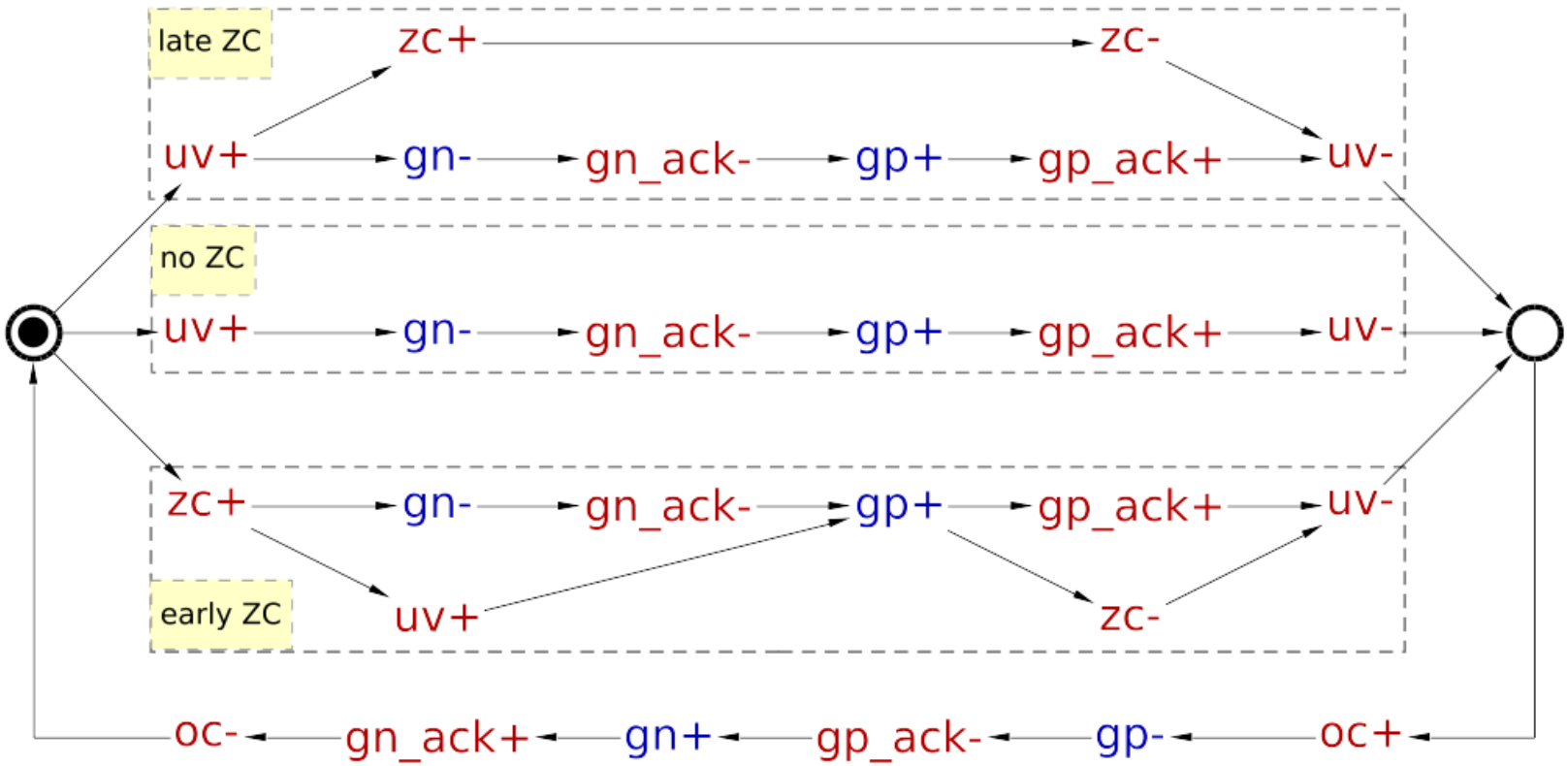
# Towards Async Design for Analog

- Asynchronous design offers many advantages for AMS control
- Challenges:
  - It requires behavioural capture and synthesis but commercial EDA tools don't support it
  - Verification of asynchronous designs as part of AMS
  - How to provide non-invasiveness with existing design practices – we need to work with SVA and SPICE simulation traces

# Buck example



# STG Specification of buck controller



# Synchronous design

- Two clocks: phase activation (~5MHz) and sampling (~100MHz)
  - ☺ Easy to design (RTL synthesis flow)
  - ☹ Response time is of the order of clock period
  - ☹ Power consumed even when idle
  - ☹ Non-negligible probability of a synchronisation failure
- Manual ad hoc design to alleviate the disadvantages
  - ☹ Verification by exhaustive simulation

# Asynchronous design

- Event-driven control decisions

  ☺ Prompt response (a delay of few gates)

  ☺ No dynamic power consumption when the buck is inactive

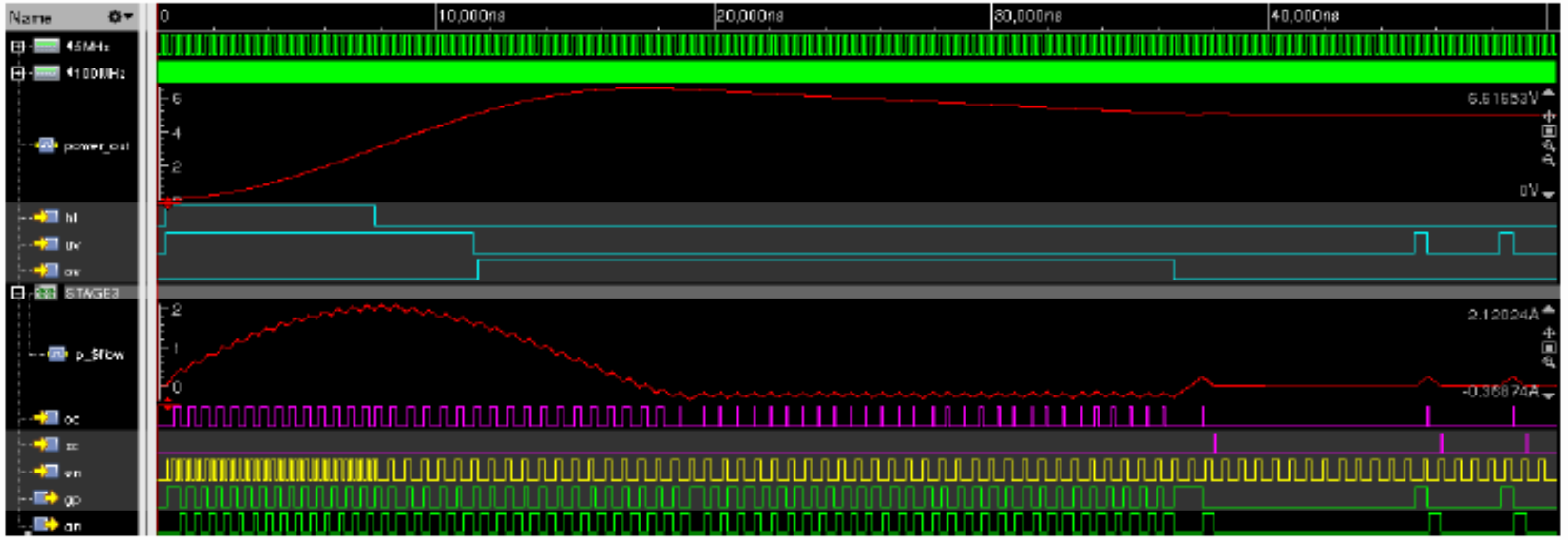
  ☺ Other well known advantages

  ☹ Insufficient methodology and tool support

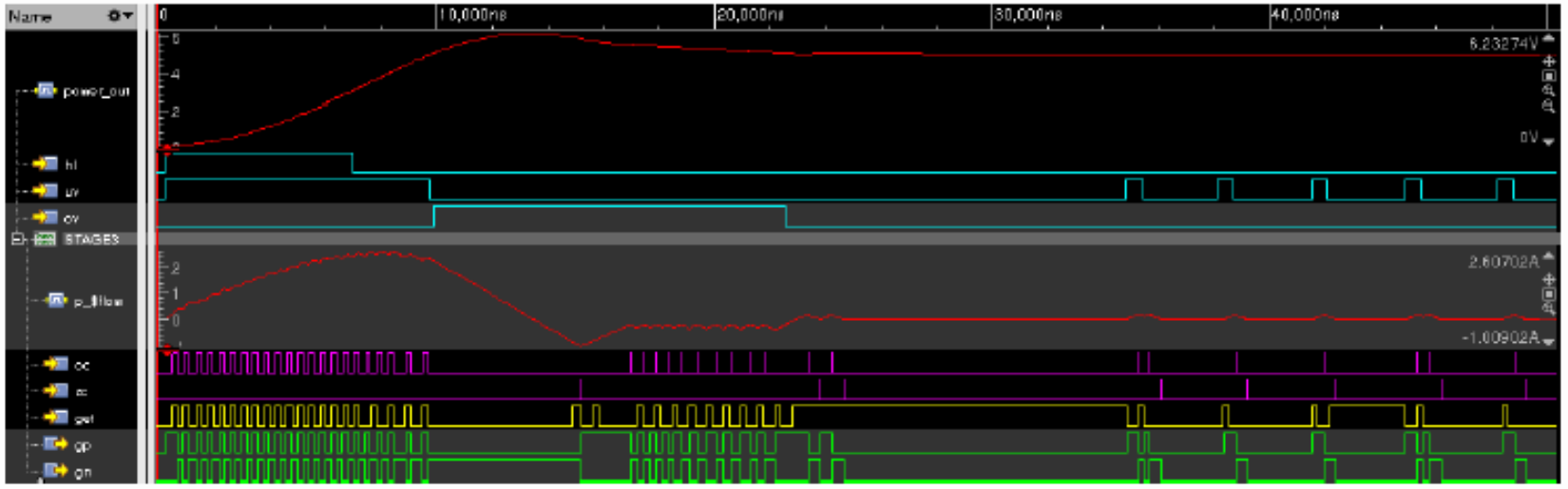
- Our goals
  - Formal specification of power control behaviour
  - Reuse of existing synthesis methods
  - Formal verification of the obtained circuits
  - Demonstrate new advantages for power regulation (power efficiency, smaller coils, ripple and transient response)

# Simulation results

synchronous




asynchronous



# Simulation results: Comparison

- Verilog-A model of the 3-phase buck
- Control implemented in TSMC 90nm
- AMS simulation in CADENCE NC-VERILOG
- Synchronous design
  - Phase activation clock – 5 MHz
  - Clocked FSM-based control – 100 MHz
  - Sampling and synchronisation
- Asynchronous design
  - Phase activation - token ring with 200 ns timer (= 5 MHz)
  - Event-driven control (input-output mode)
  - Waiting rather than sampling (A2A components)

# Specifics of Async Design for Bucks

- Needs to be to a large extent monolithic
- Has inputs that need to be sanitised
- Can have lots of timing assumptions for bounded delay implementation where solving coding and TM problems can be an issue
- I/O response times (constrained or optimised) drive the design and sign-off
- Different types of (de)compositions needed rather than (or not just) handshake ones


# 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

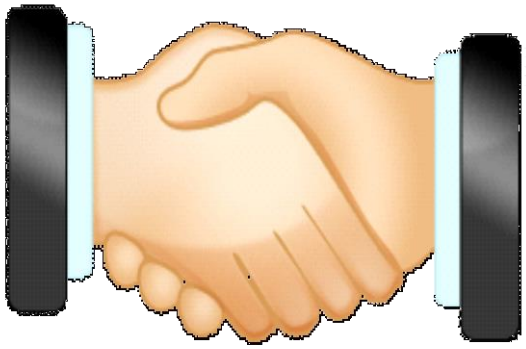
- Synchronous vs Asynchronous behaviour in general terms, examples:
  - Orchestra playing with vs without a conductor
  - Party of people having a set menu vs a la carte
- Synchronous means all parts of the system acting globally in tact, even if some or all part ‘do nothing’
- Asynchronous means parts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

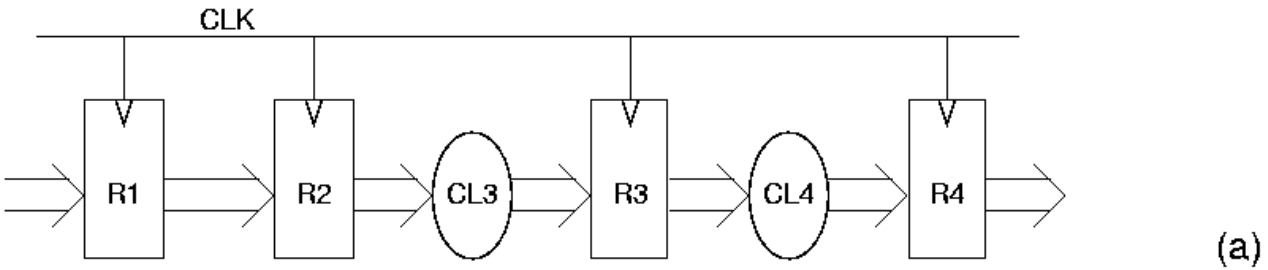
# Why and what is handshaking?



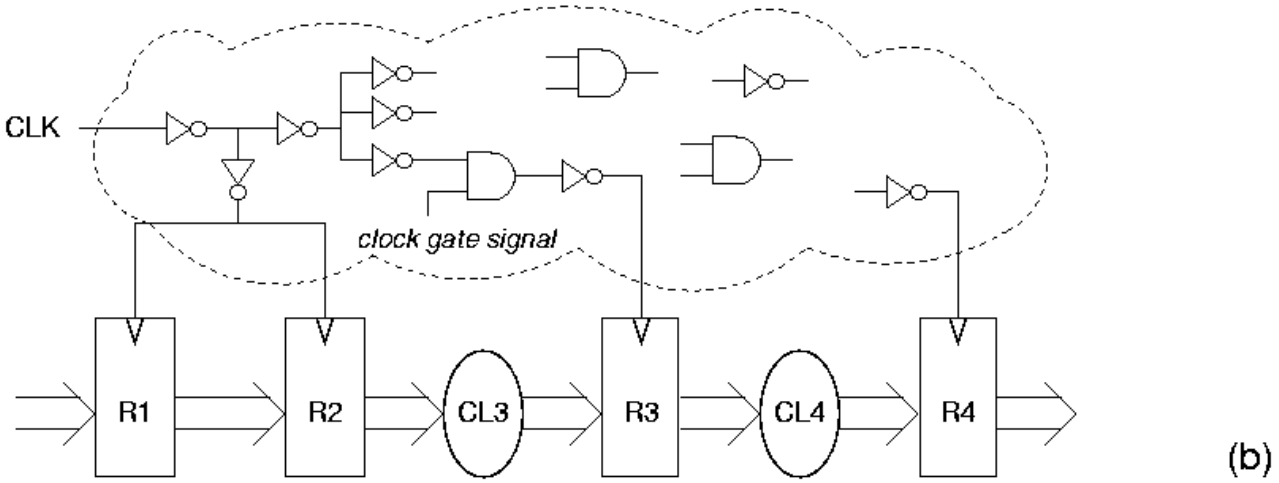
**Mutual Synchronisation is via Handshake**

# Synchronous clocking

**How we think**

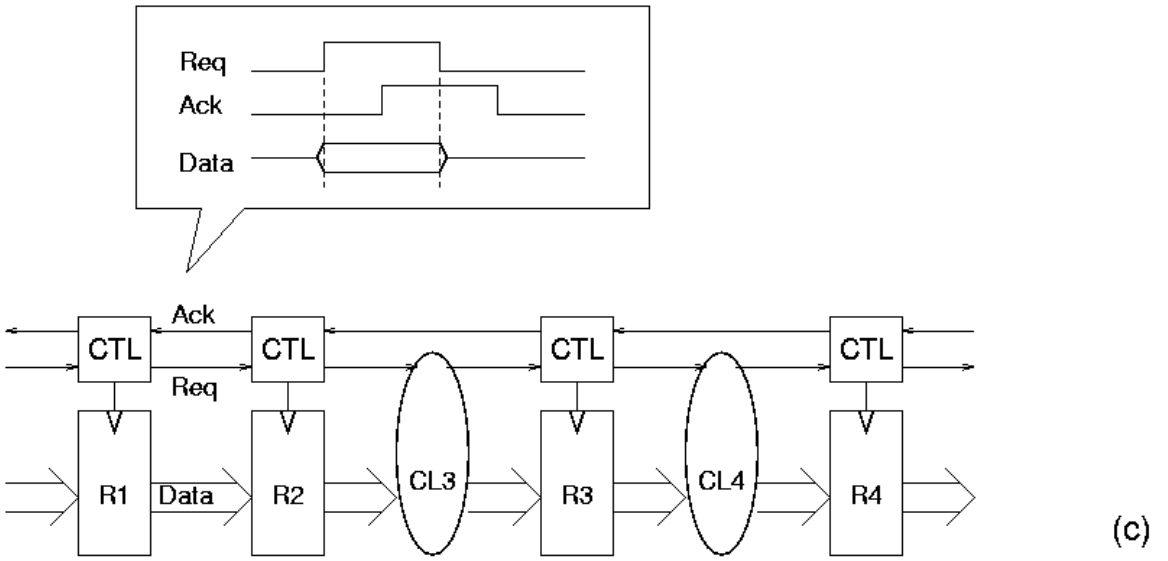



**What we design**

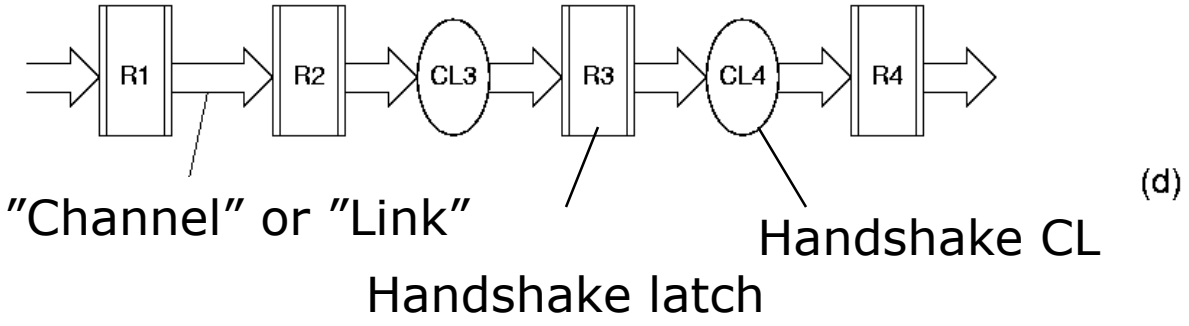


# Asynchronous handshaking

**What we design**

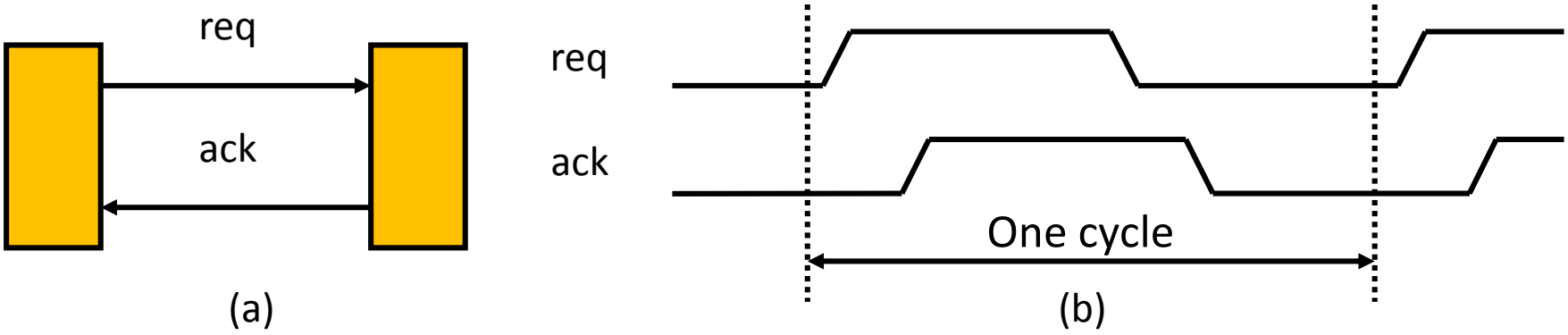



**How we think**

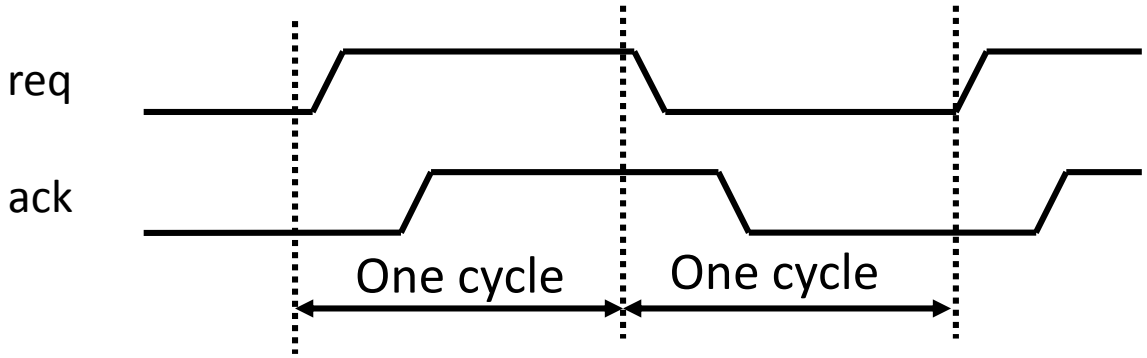


# Handshake Signalling Protocols

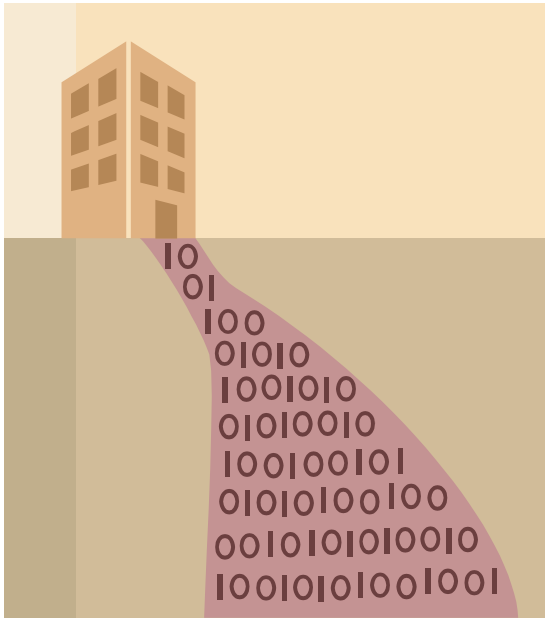
## Level Signalling (RTZ or 4-phase)




## Transition Signalling (NRZ or 2-phase)

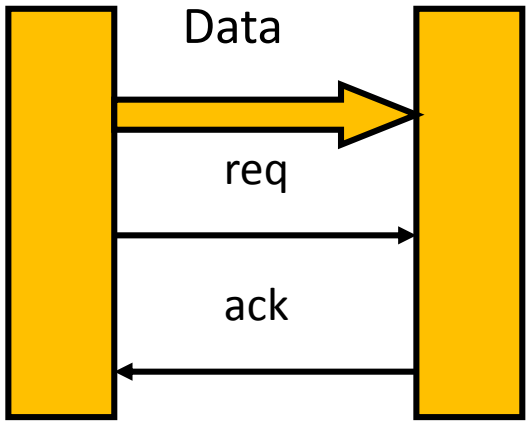


# Why and what is delay-insensitive coding?

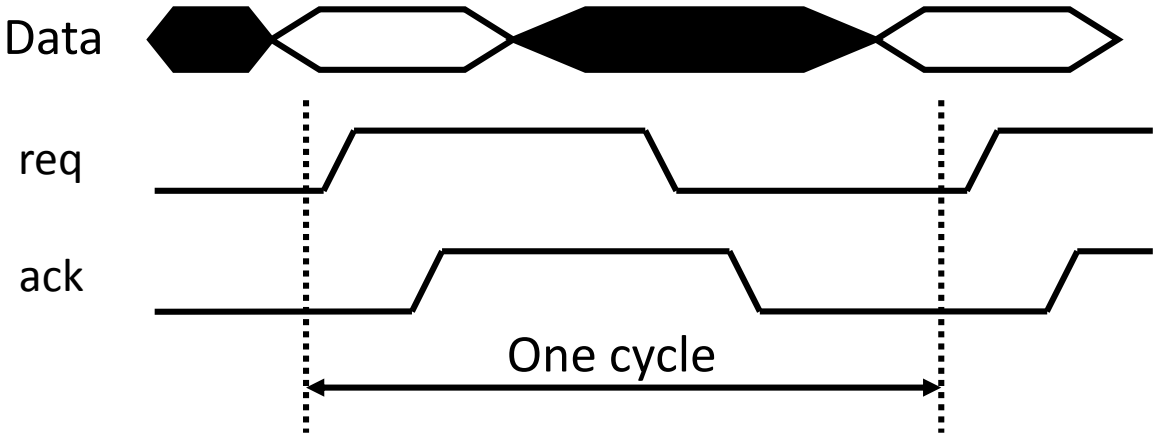


**Data Token = (Data Value, Validity Flag)**

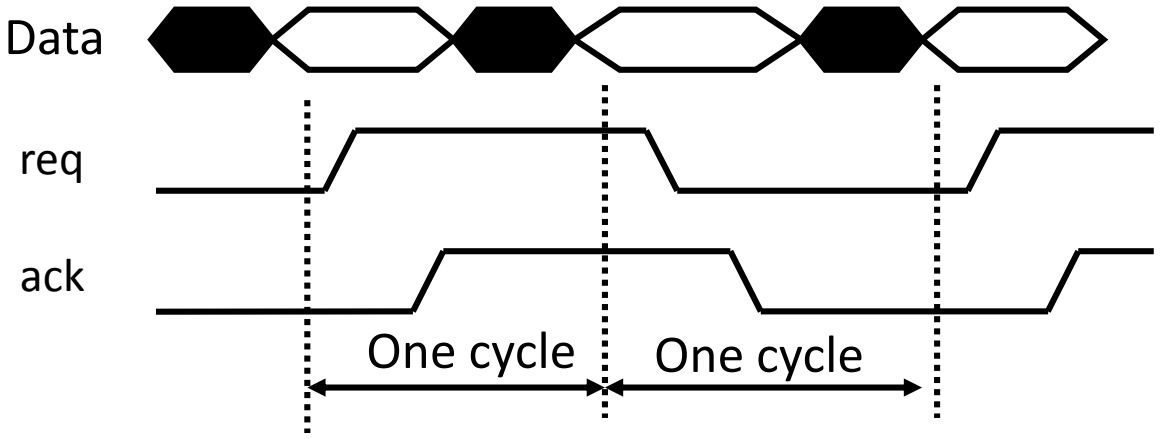
# Bundled Data



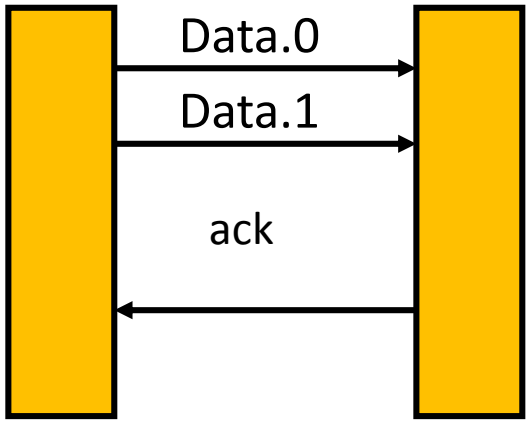

Return to Zero:



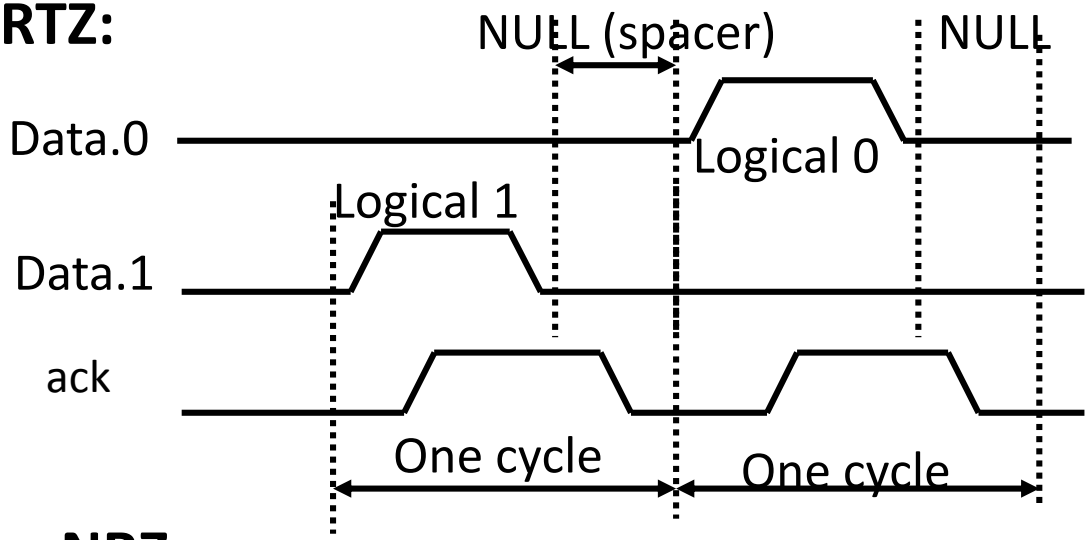

**Non-Return-to-Zero**



# DI encoded data (Dual-Rail)

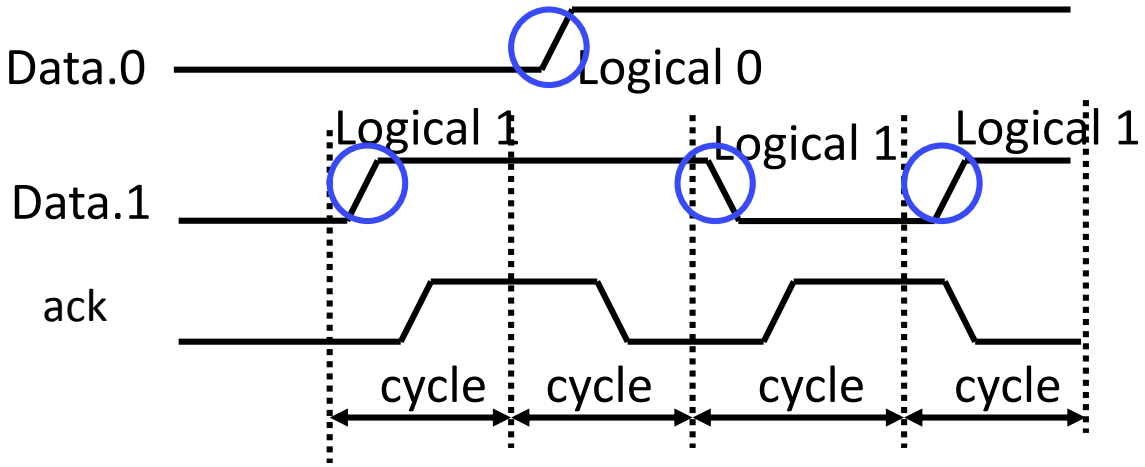



**RTZ:**




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

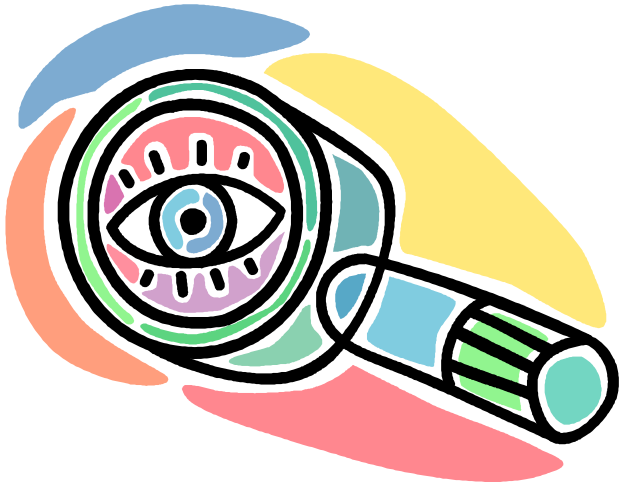
**NRZ:**



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

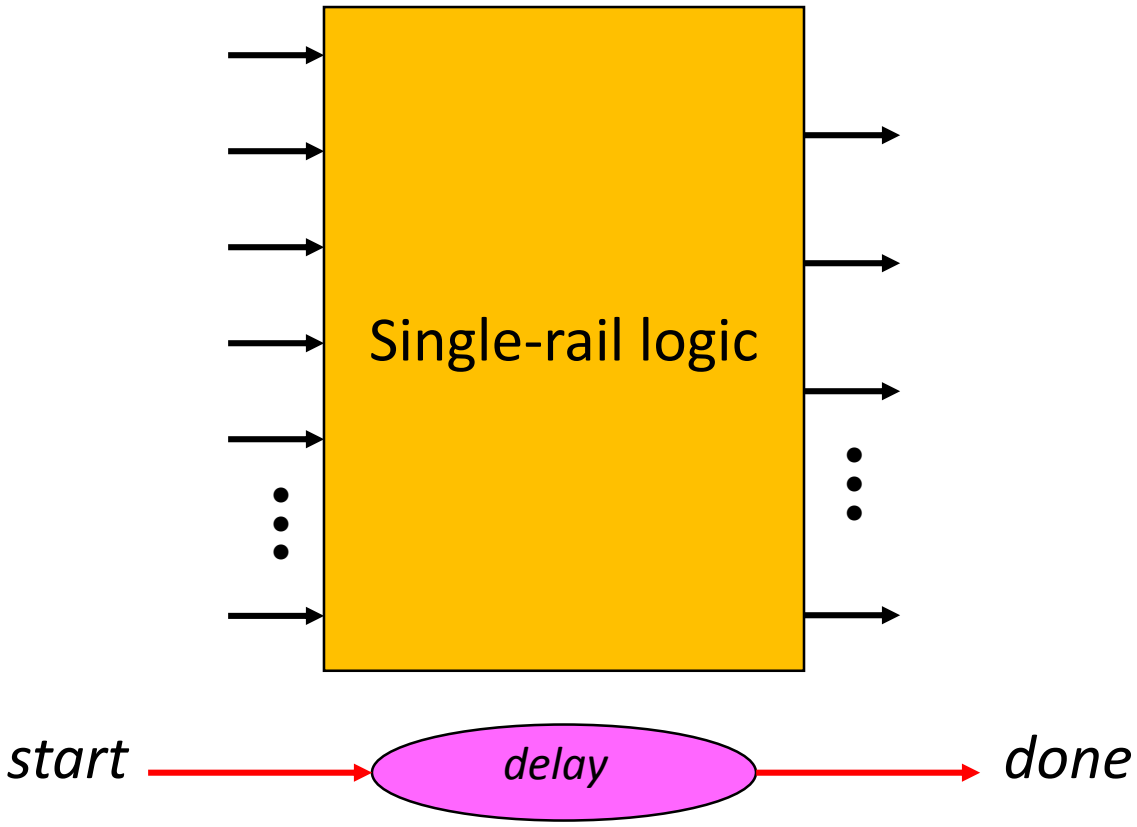
- **1-of-4:**
  - **0001=> 00, 0010=>01, 0100=>10, 1000=>11**
- **2-of-4:**
  - **1100, 1010, 1001, 0110, 0101, 0011 – total 6 combinations (cf. 2-bit dual-rail – 4 comb.)**
- **3-of-6:**
  - **111000, 110100, …, 000111 – total 20 combinations (can encode 4 bits + 4 control tokens)**
- **2-of-7:**
 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

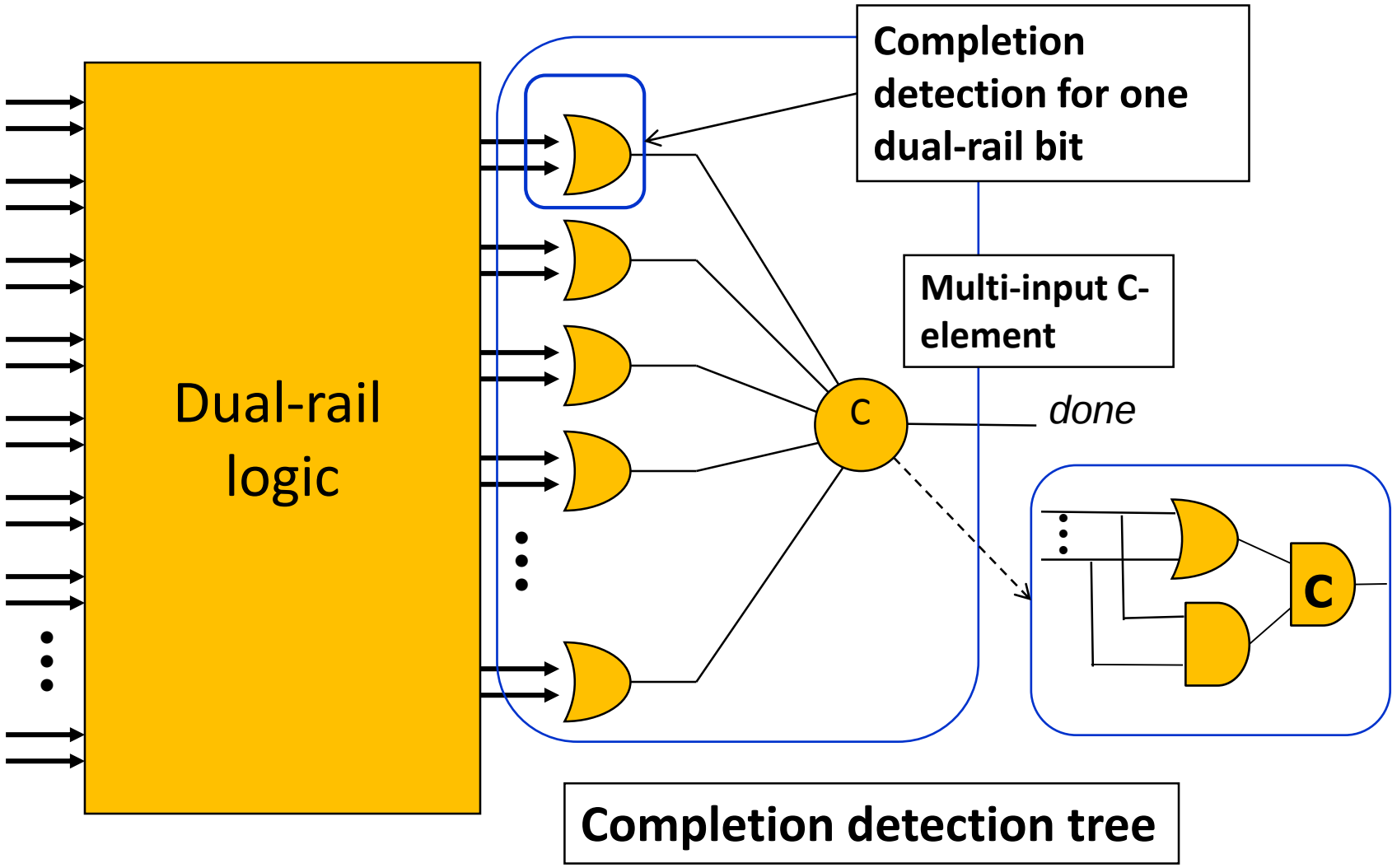



Completion is implicit: by done signal

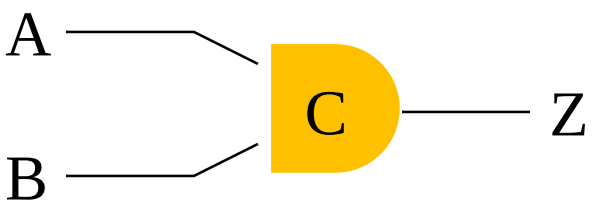
The delay must scale with the worst case delay path, So … not really self-timed

**Conventional logic + matched delay**

# True completion detection

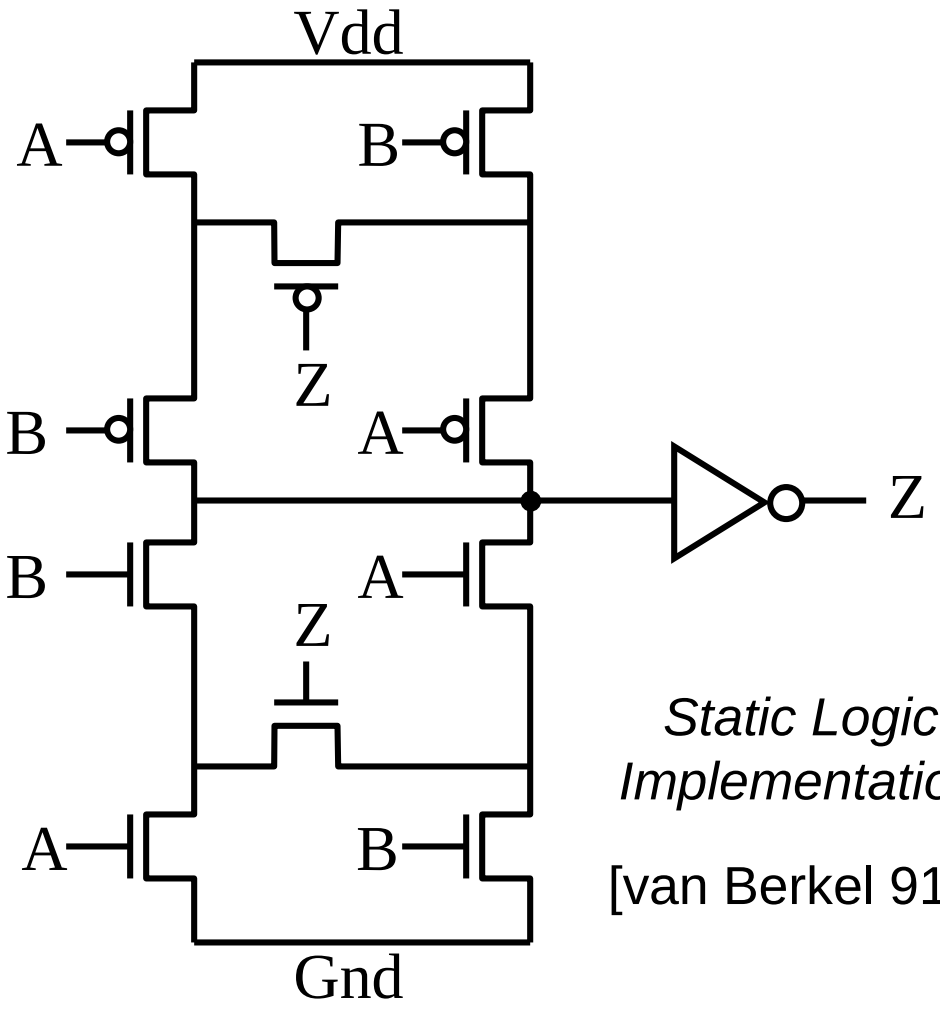


# The Muller C element




C

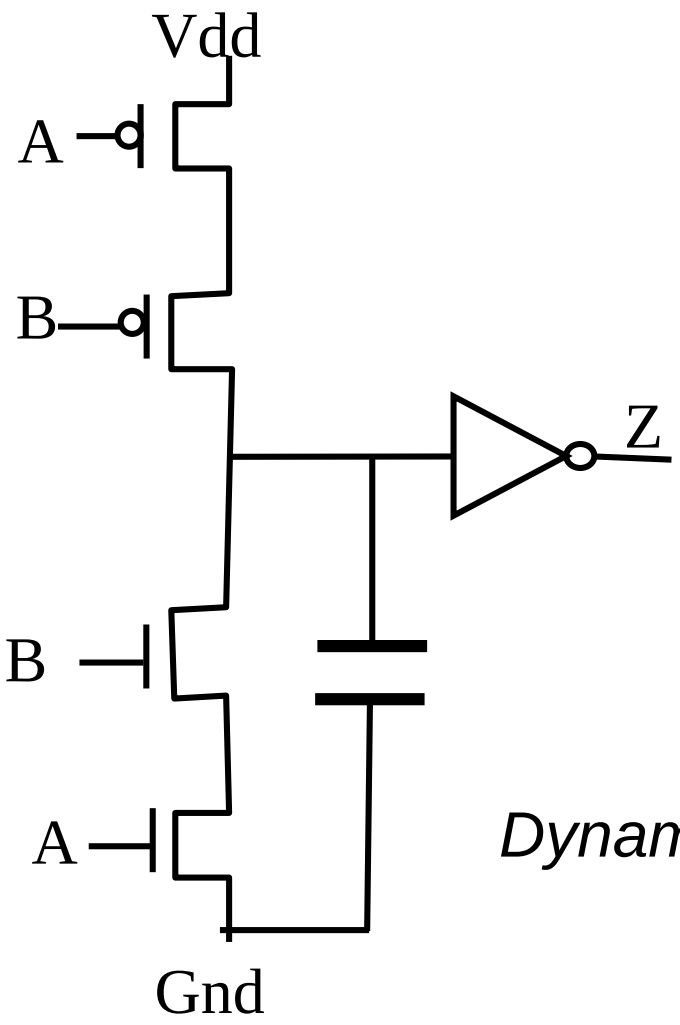
| A | B | $Z^+$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | Z |
| 1 | 0 | Z |
| 1 | 1 | 1 |



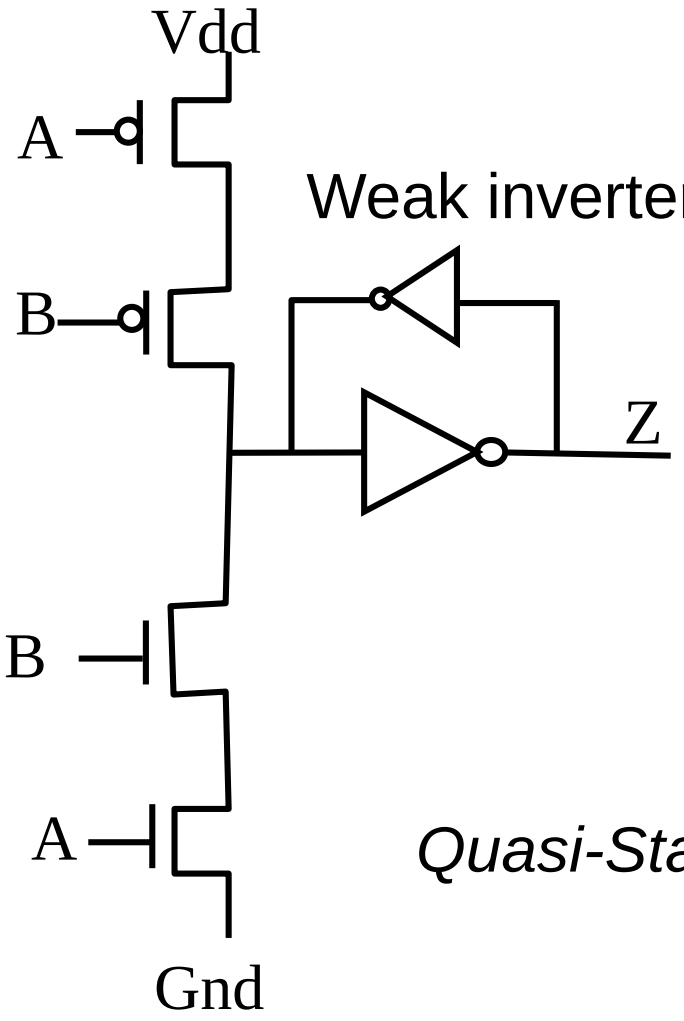

*Static Logic Implementation*

[van Berkel 91]

# C-element: Other implementations




*Dynamic*

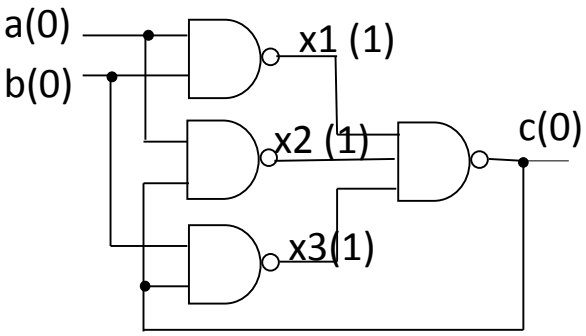


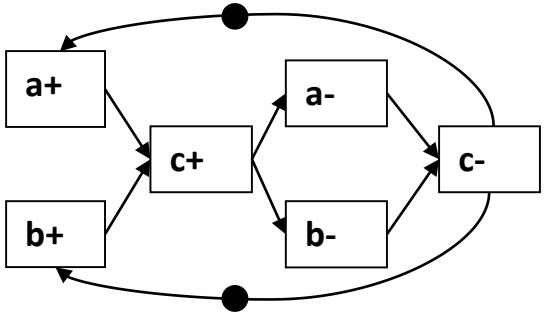

*Quasi-Static*

# Why and what is causal acknowledgment?

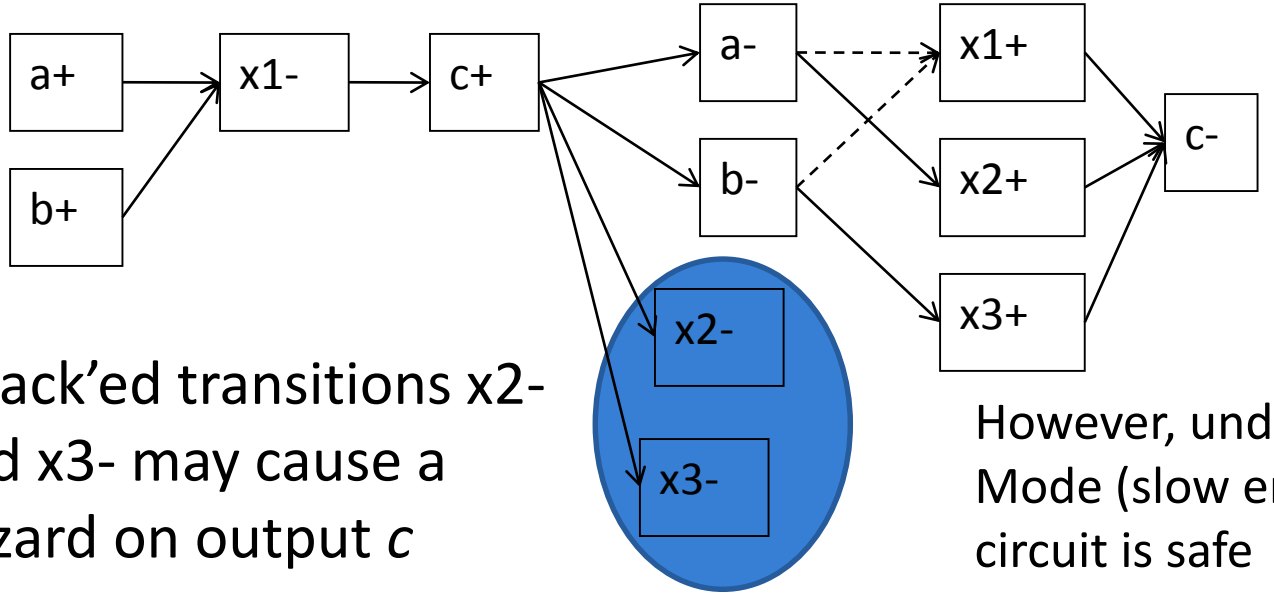

**Every signal event must be acknowledged by another event**

# Causal acknowledgment



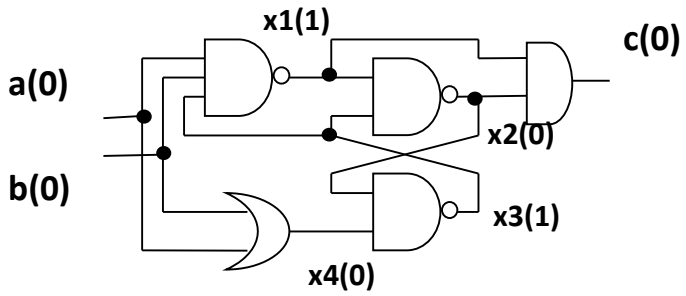

C-element implementation using simple gates

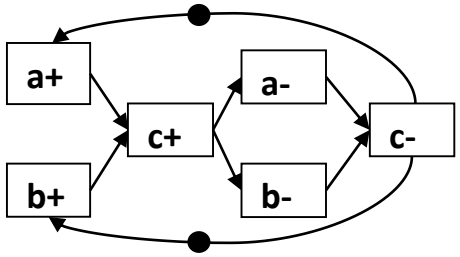



Unack'ed transitions x2- and x3- may cause a hazard on output *c*

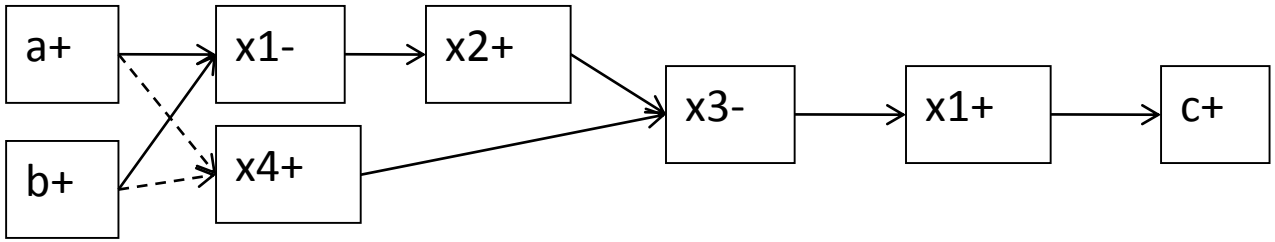
However, under Fundamental Mode (slow environment) the circuit is safe

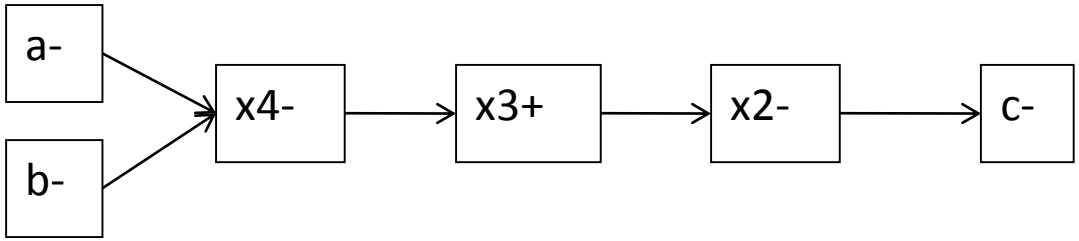
# Principle of causal acknowledgement



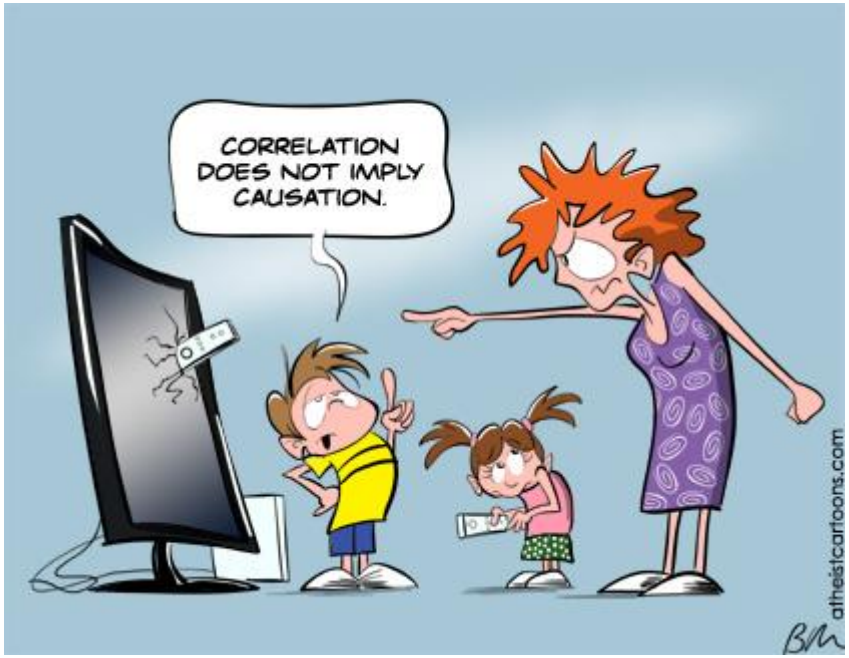

C-element implementation using simple gates




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

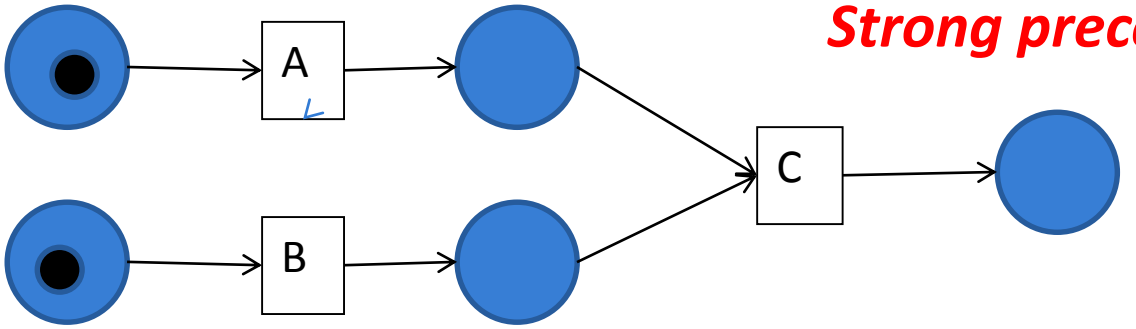
# Why and what are strong and weak causality ?




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

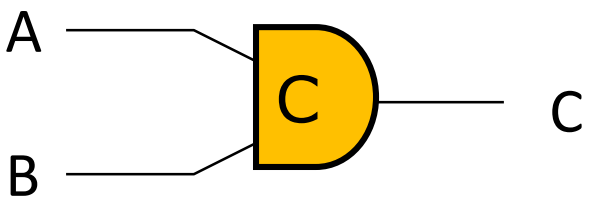
# Strong Causality

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




***Strong precedence***

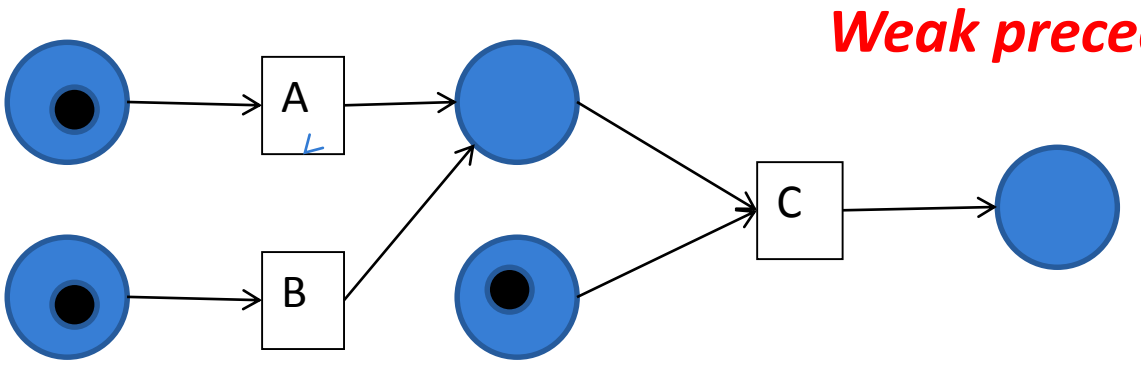
- **Logic circuits: Muller C-element (in 0-1 and 1-0 transitions), AND gate (in 0-1 transitions), OR gate (in 1-0 transitions)**



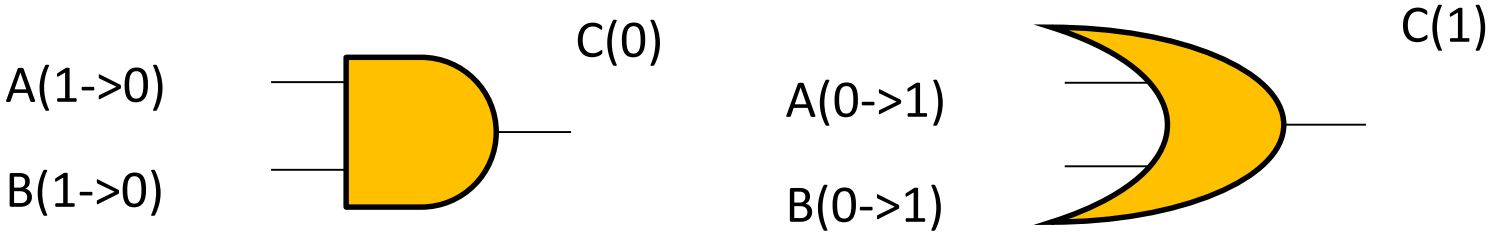

| A | B | $C^+$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | C |
| 1 | 0 | C |
| 1 | 1 | 1 |

# Weak Causality

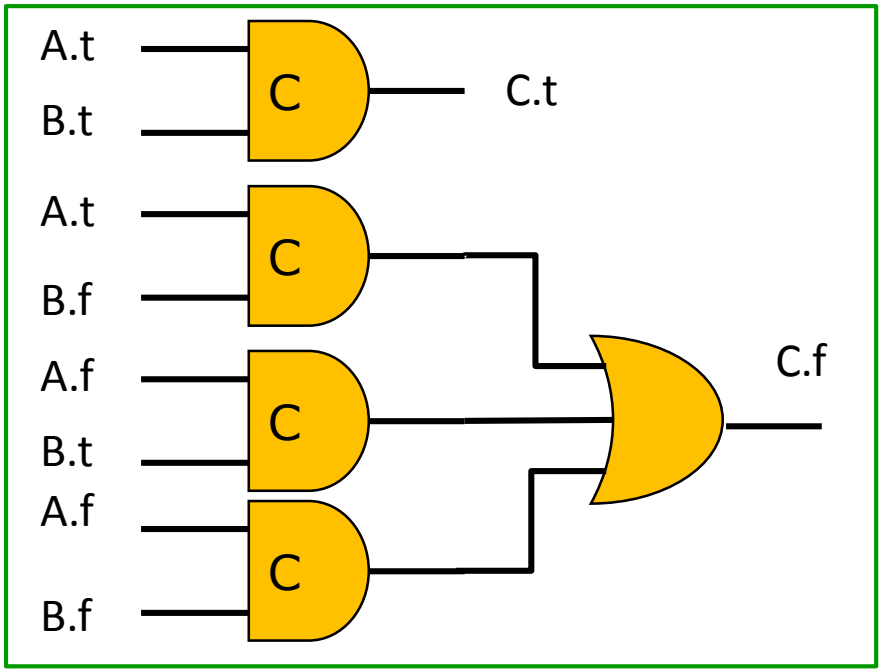
- **Petri net transitions communicating via places**



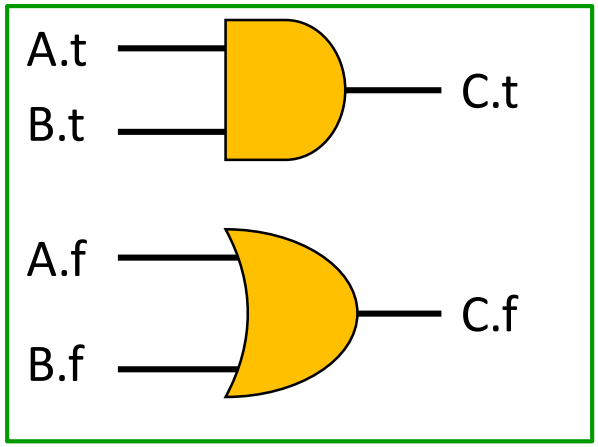

- **Logic circuits: AND gate (in 1-0 transitions), OR gate (in 0-1 transitions)**



# 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)**

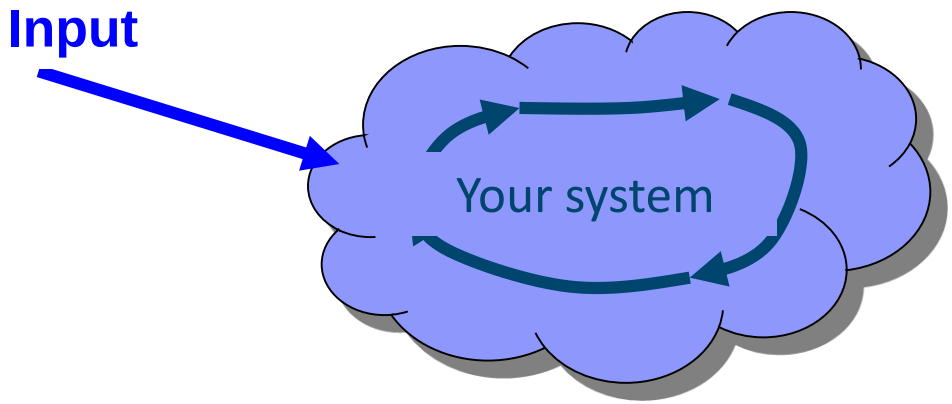
# Why and what is timing comparison?

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

# Synchronizers and arbiters

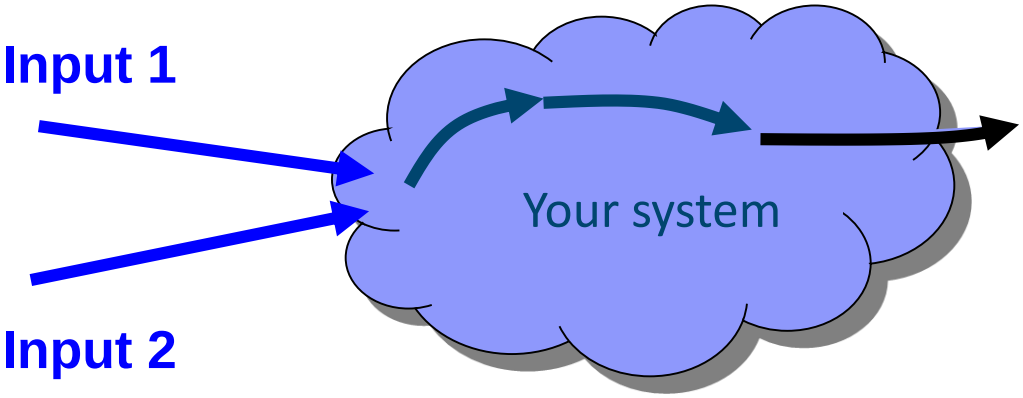
- Synchronizer

Decides which clock cycle to use for the input data

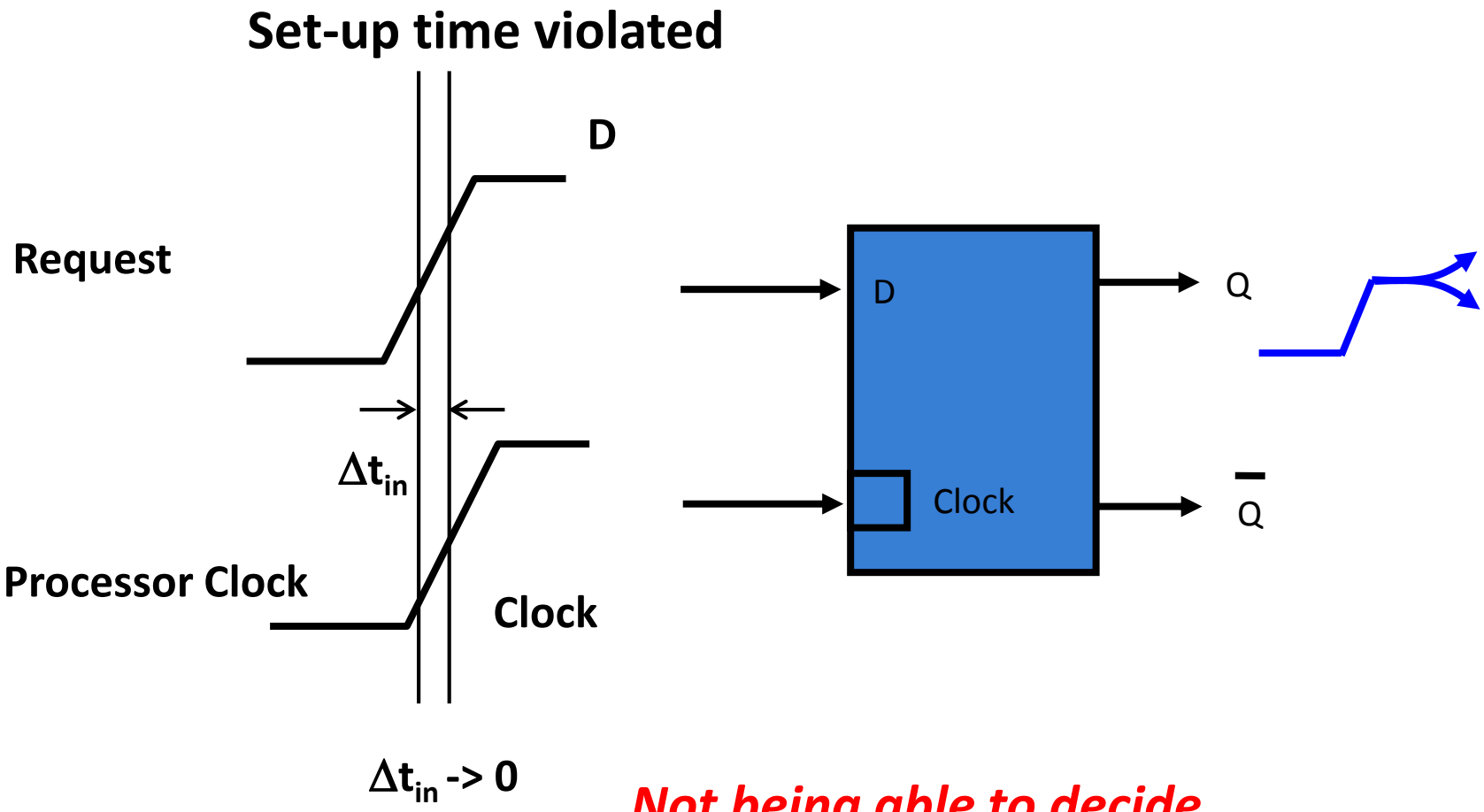



- Asynchronous arbiter

Decides the order of inputs

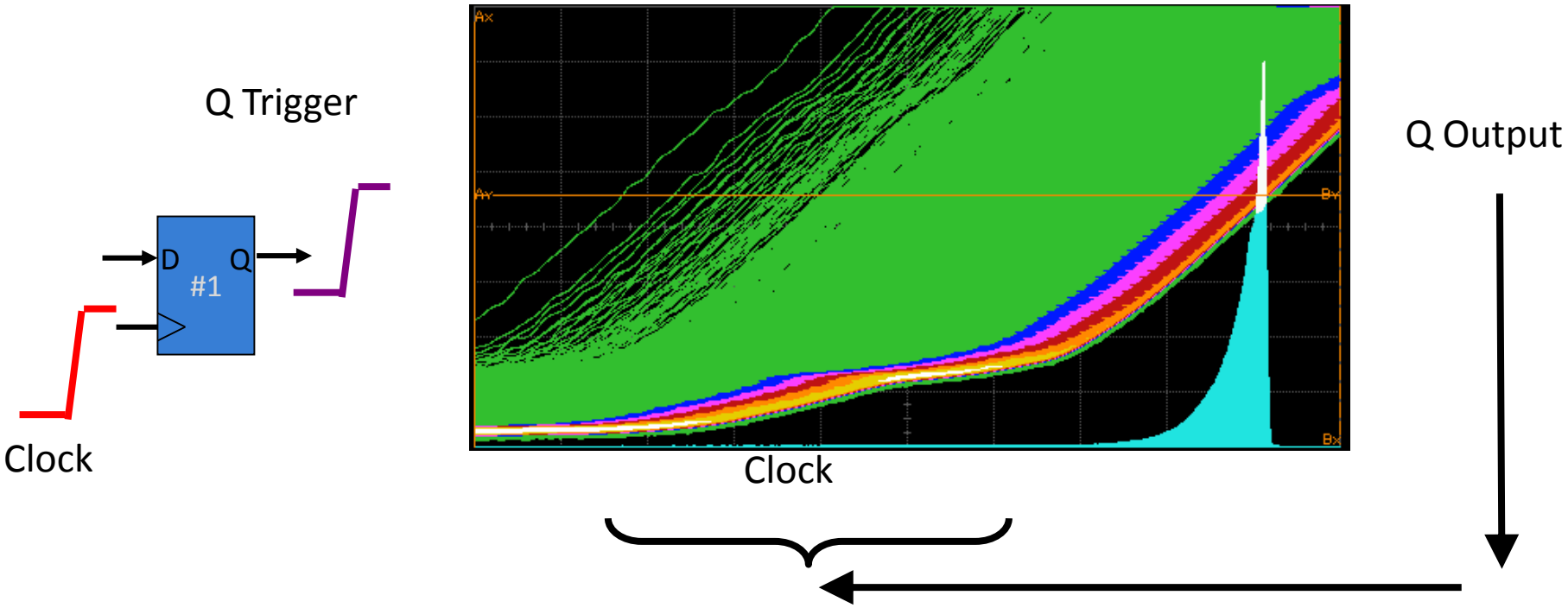


# Metastability is....




***Not being able to decide…***

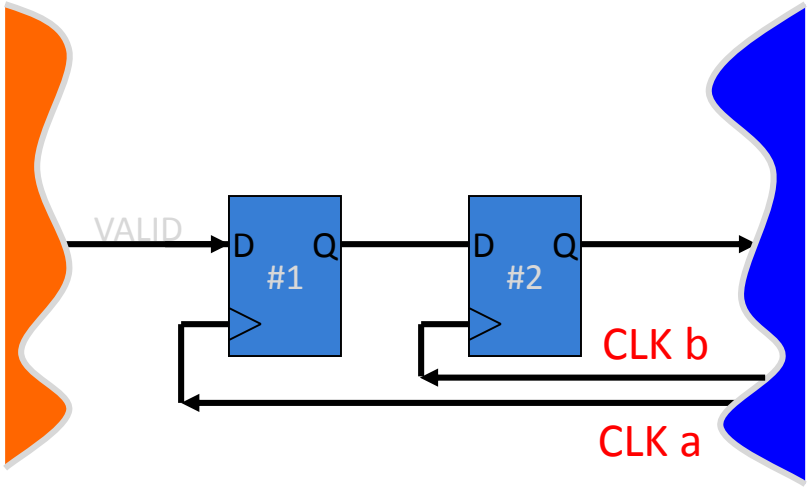
# Typical responses




- **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

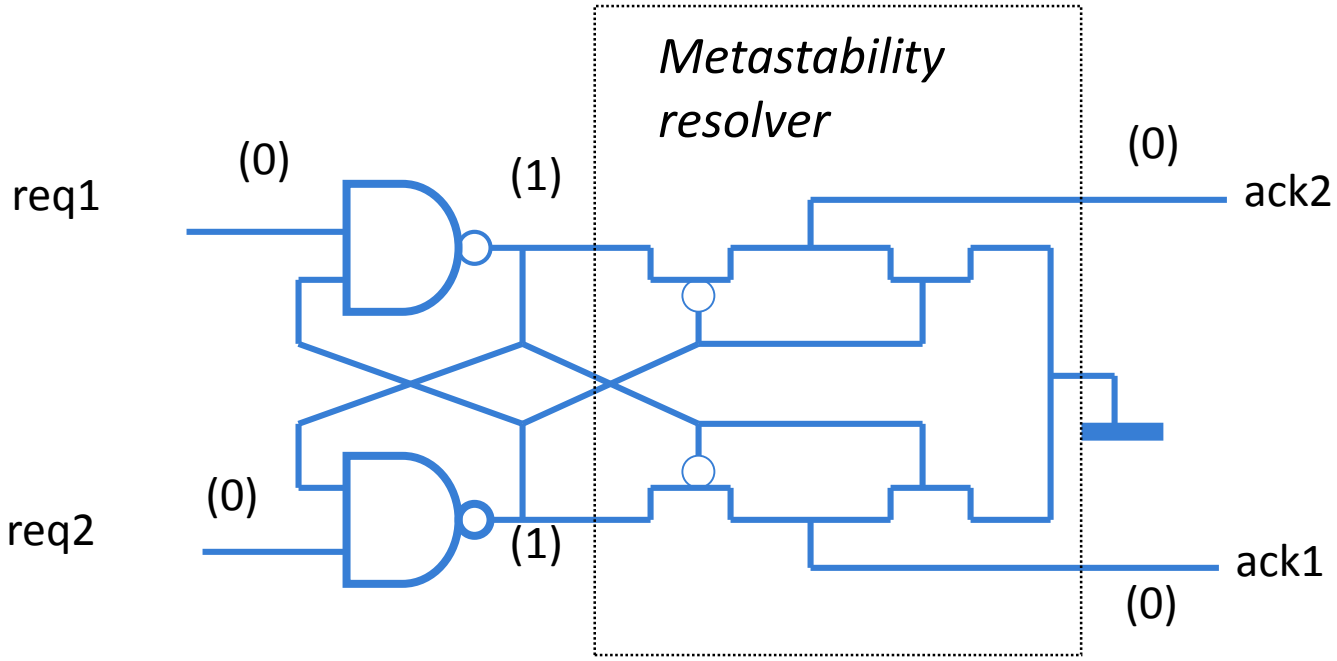
- $t$ is time allowed for the Q to change between CLK a and CLK b
- $\tau$ is the recovery time constant, usually the gain-bandwidth of the circuit
- $T_w$ is the "metastability window" (aperture around clock edge in which the capture of data edge causes a delay that is greater than normal propagation 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 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

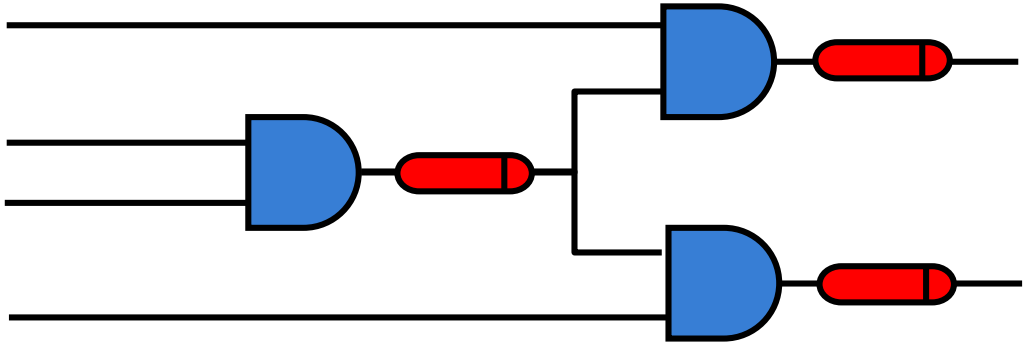
# (Some) Models for Asynchronous Circuit Design

# Models and techniques for asynchronous design

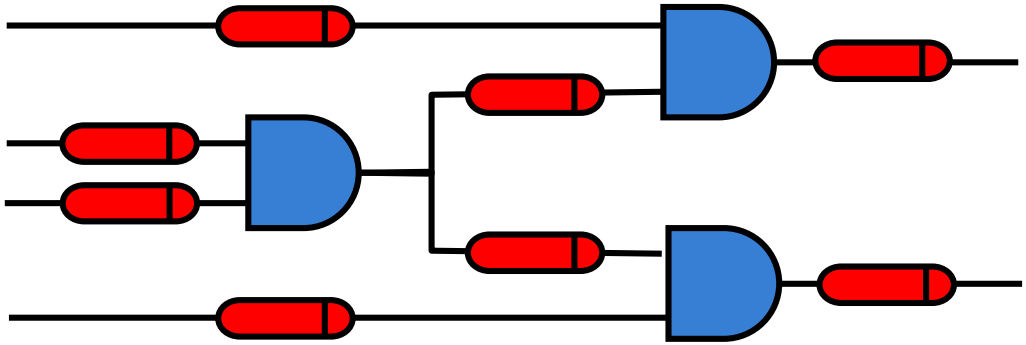
- **Models:**
  - **Delay model (inertial, pure, gate delay, wire delay, bounded and unbounded delays)**
  - **Models of environment (fundamental mode, input-output)**
  - **Models of switching behaviour (state-based, event-based, hybrid)**
- **RTL level:**
  - **Data and control paths separate (data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

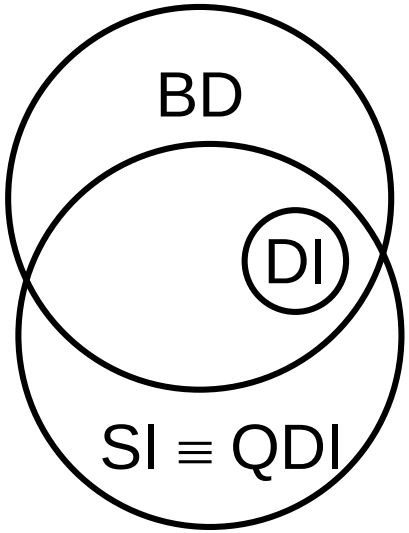


- **Wire delay model: delays in gates and wires**



# 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

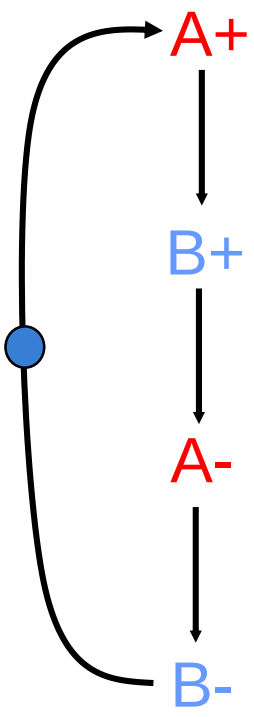


# Control Logic

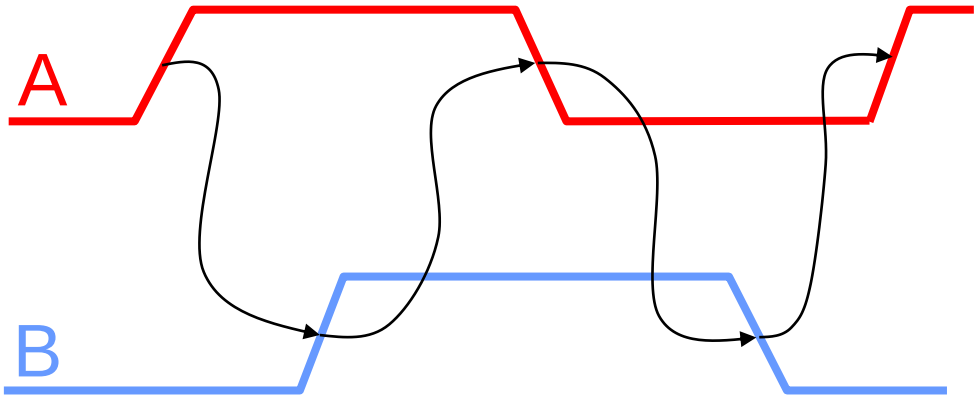
- **Control specification based on Petri nets (Signal Transition graphs)**

# Control specification

Signal Transition Graph (STG)



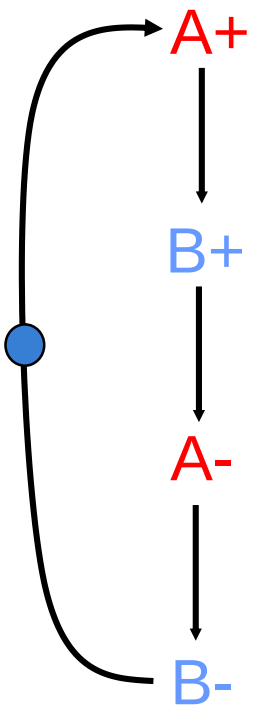

Timing Diagram

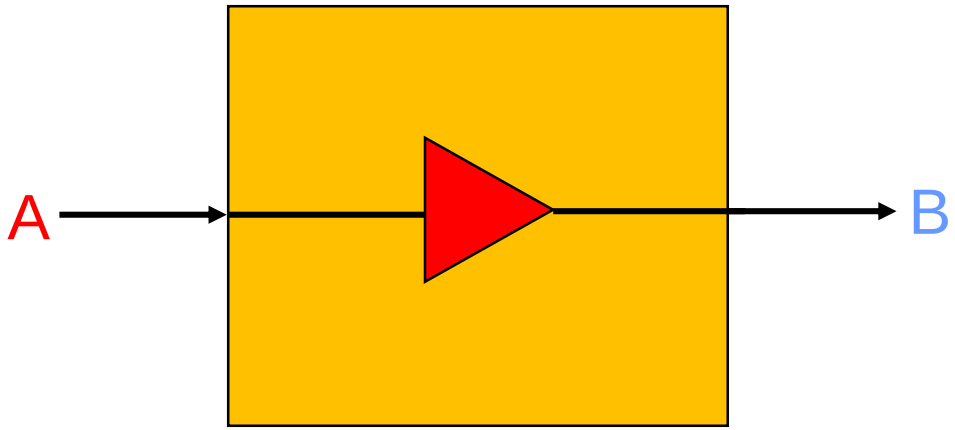



A input

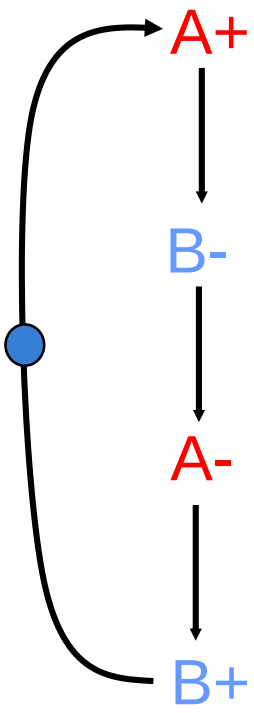
B output

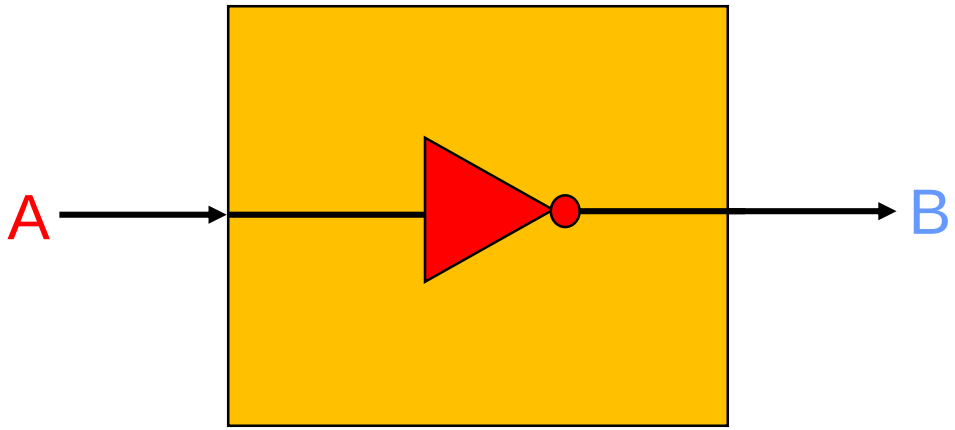
# Control specification



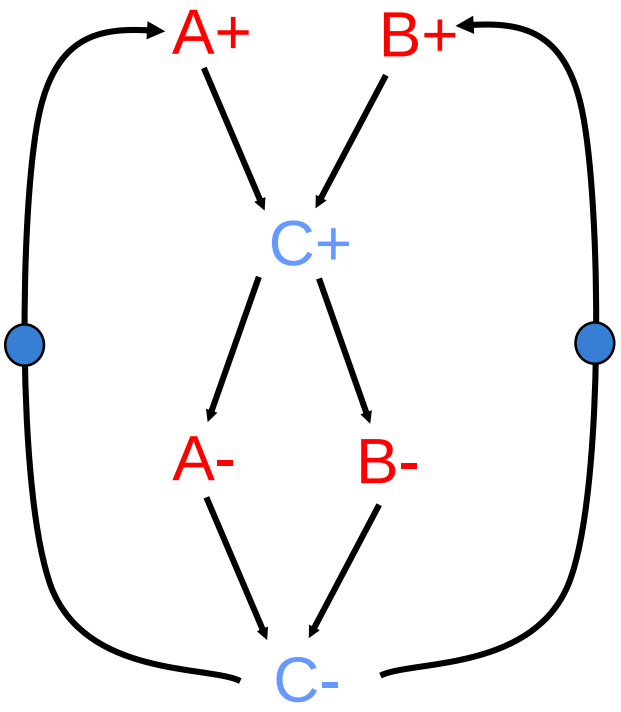



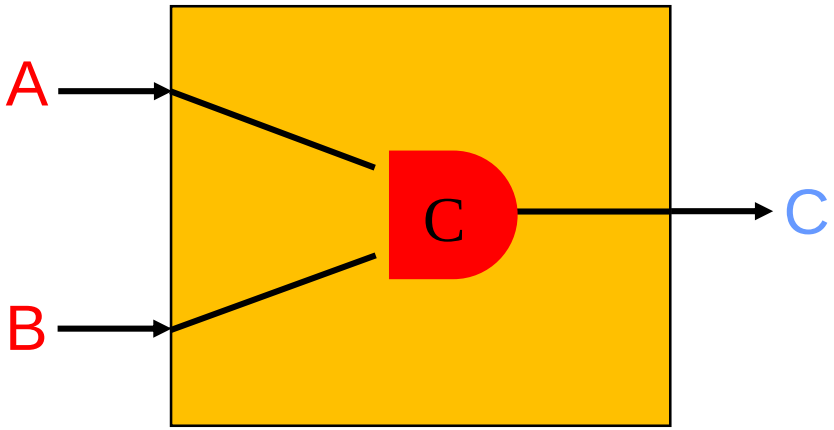
# Control specification



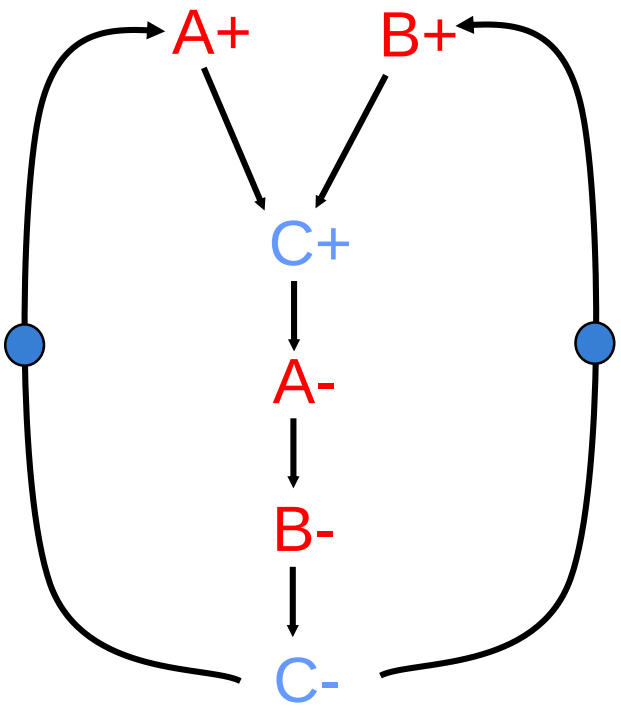



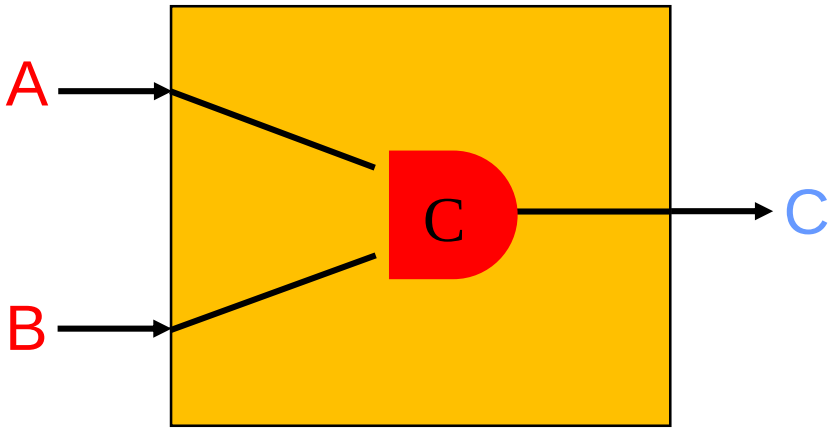
# Control specification



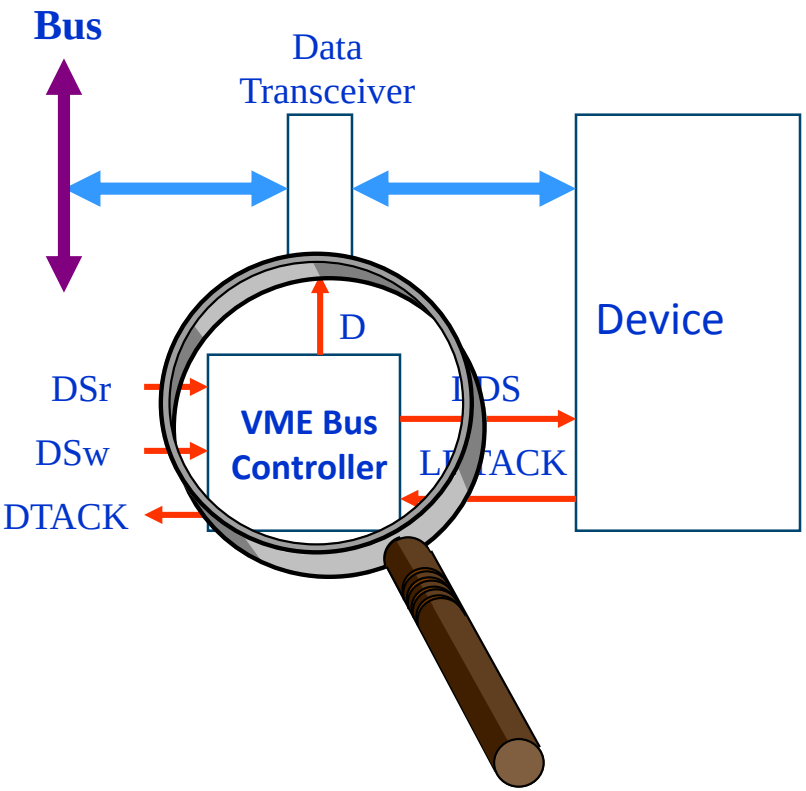



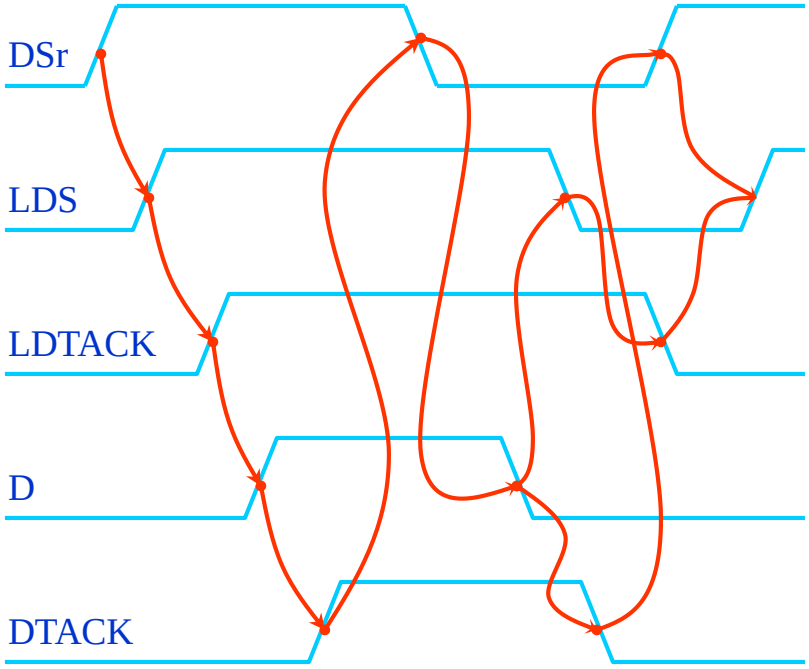
# Control specification



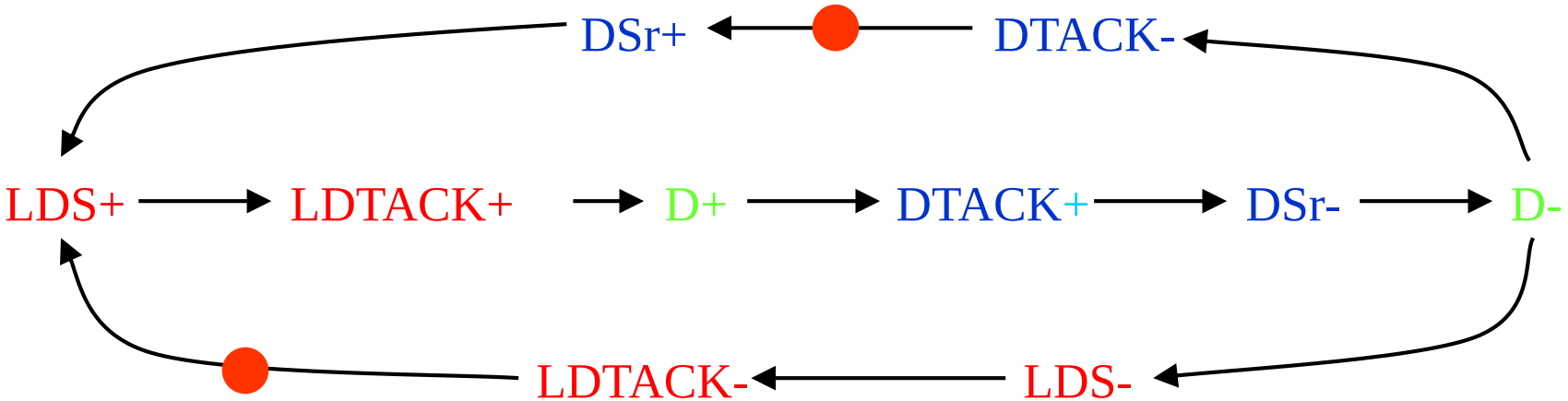
# VME bus example using Petri nets

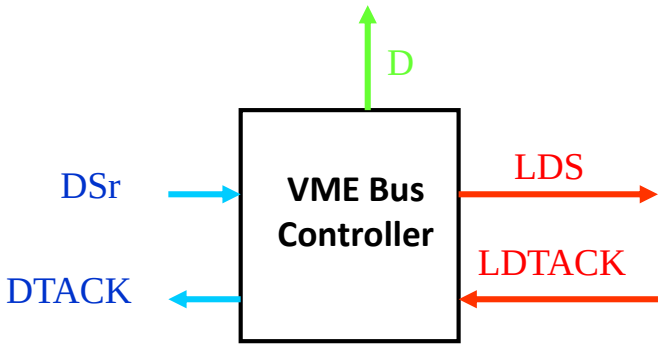


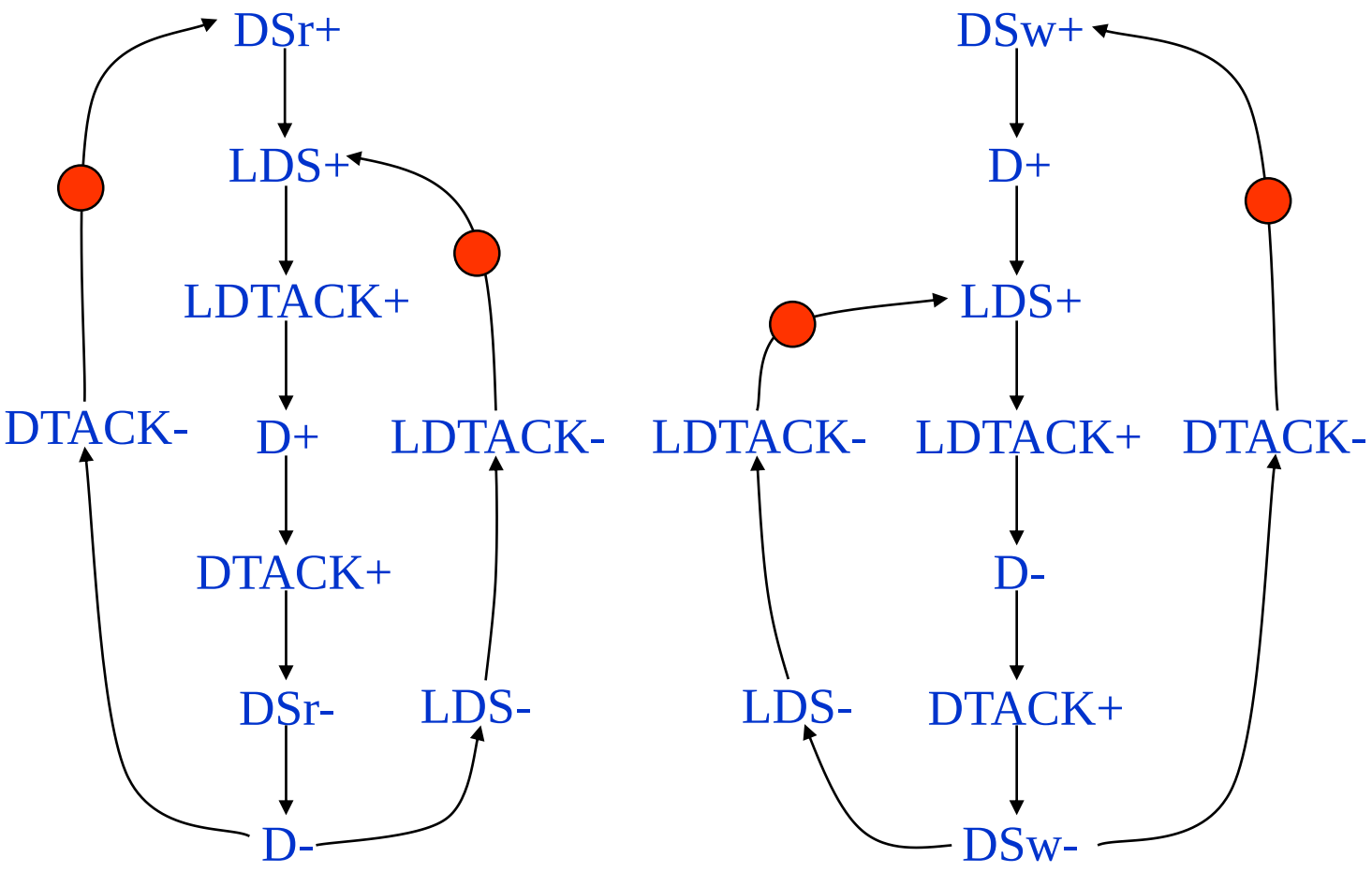

**Read Cycle**

# STG for the READ cycle

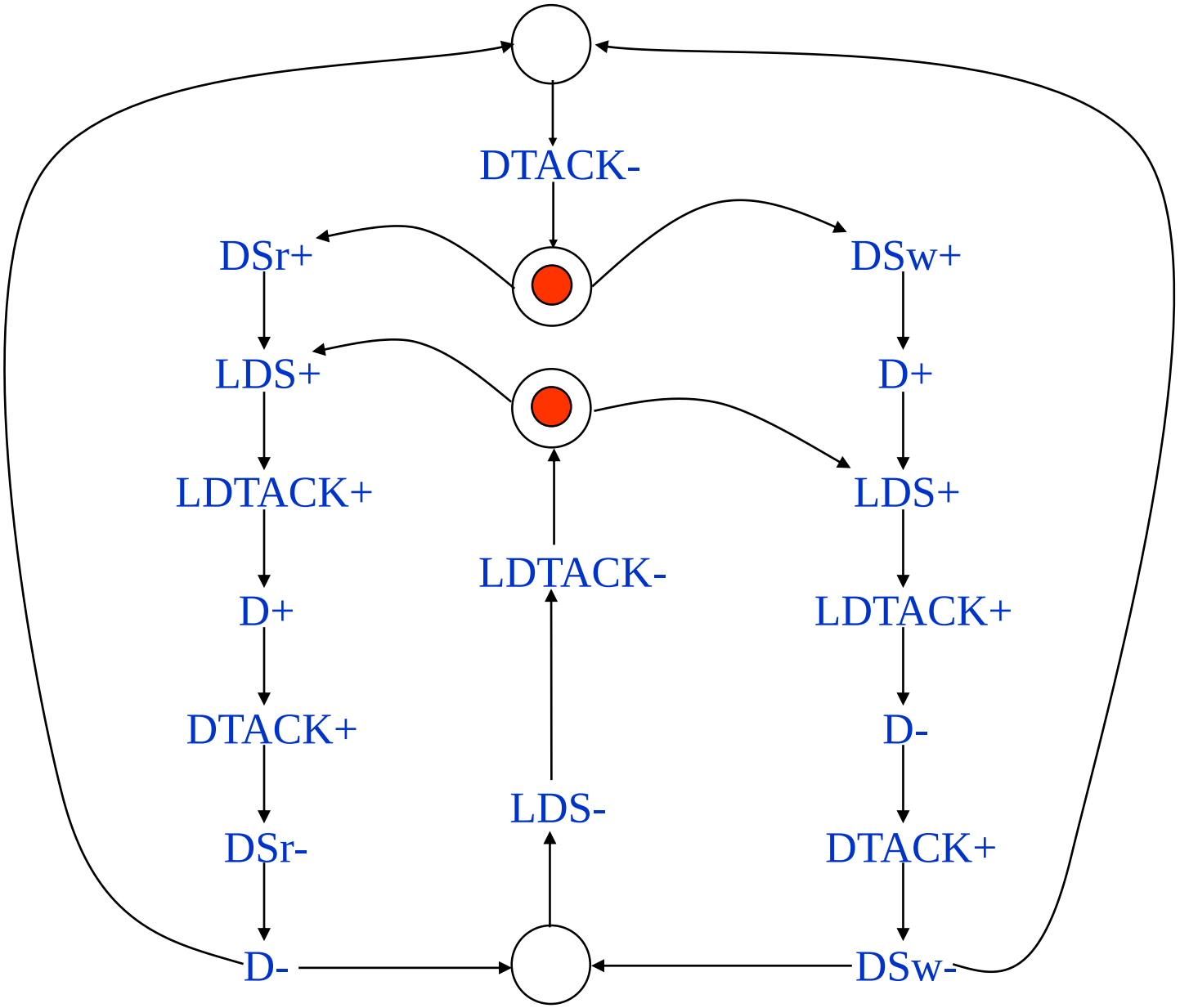


# Choice: Read and Write cycles



# Choice: Read and Write cycles



# Workcraft tool

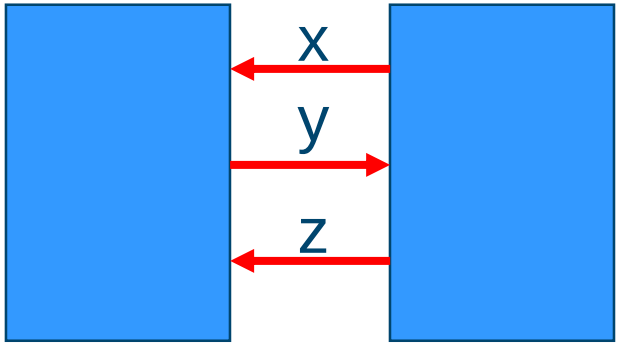
- **Workcraft is a software package for graphical edit, analysis, synthesis and visualisation of asynchronous circuit behaviour**
- **Petrify plus a few other tools are part of it as plug-ins**
- **It is based in Java tools**
- **Can be downloaded from http://workcraft.org/wiki/doku.php?id=download**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

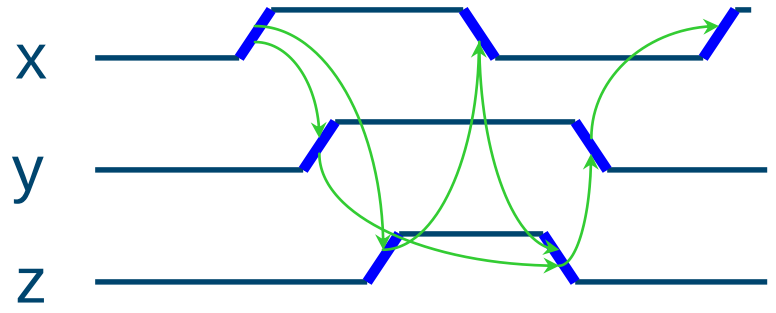
# Some references

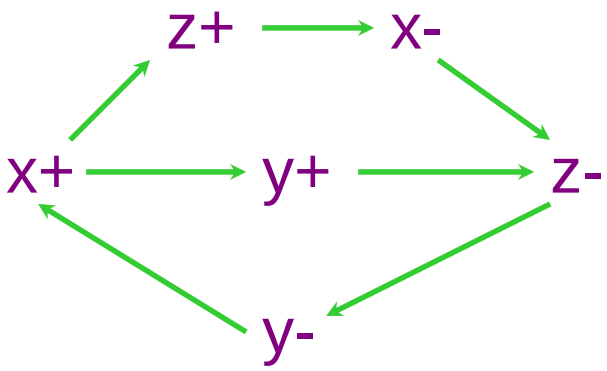

- **General Async Design:** J. Sparsø and S.B. Furber, editors. *Principles of Asynchronous Circuit Design*, Kluwer Academic Publishers, 2001. (electronic version of a tutorial based on this book can be found on: http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/855/pdf/imm855.pdf
- **Async Control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

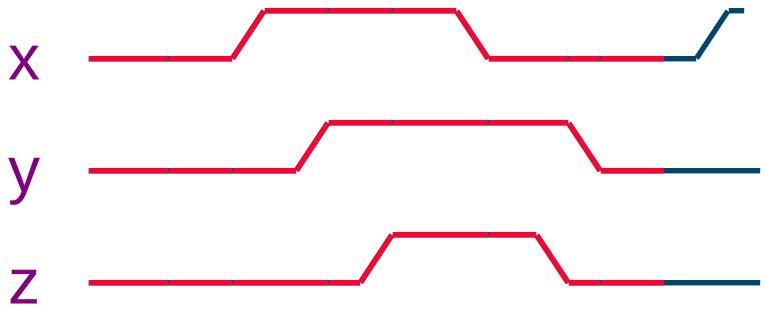


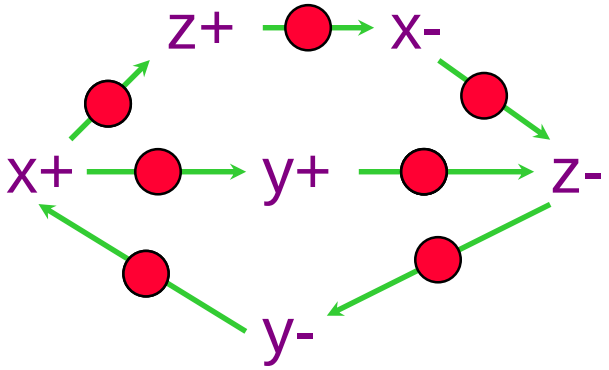



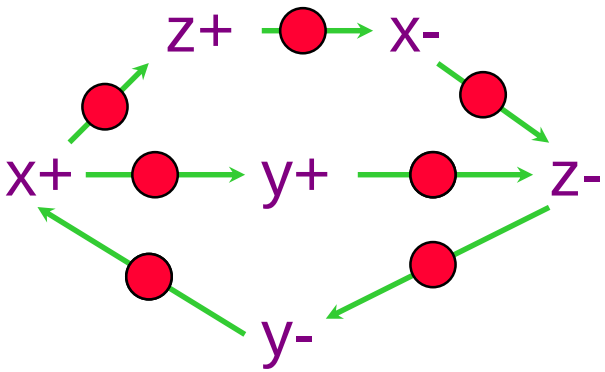

***Signal Transition Graph (STG)***

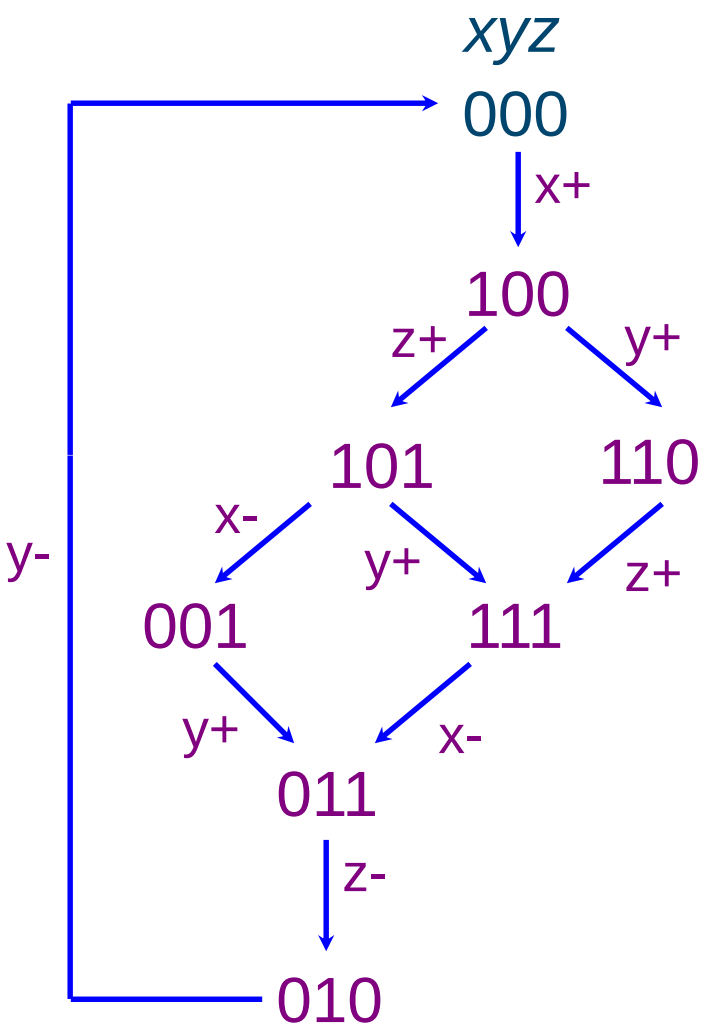
# Token flow



# State graph



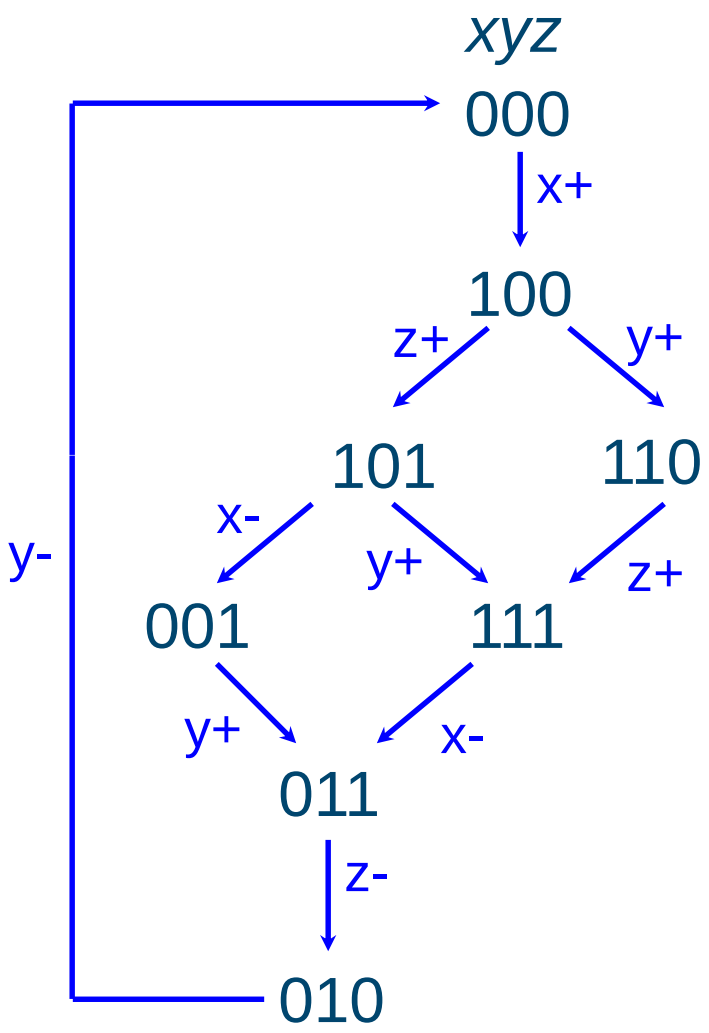



# Next-state functions

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

$$y = z + x$$

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

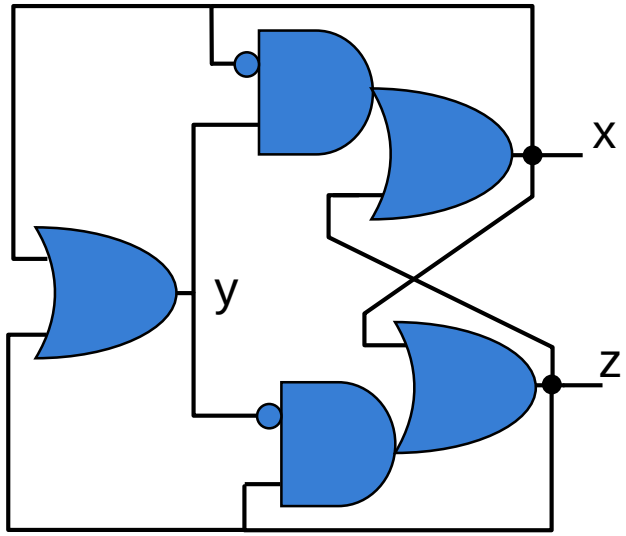


# Complex Gate netlist

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

$$y = z + x$$

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



# Circuit synthesis

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

# Speed independence

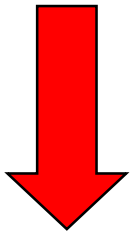
- **Delay model**
  - **Unbounded gate / environment delays**
  - **Certain wire delays shorter than certain paths in the circuit**
- **Conditions for implementability:**
  - **Consistency**
  - **Complete State Coding**
  - **Output-Persistency**

# 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

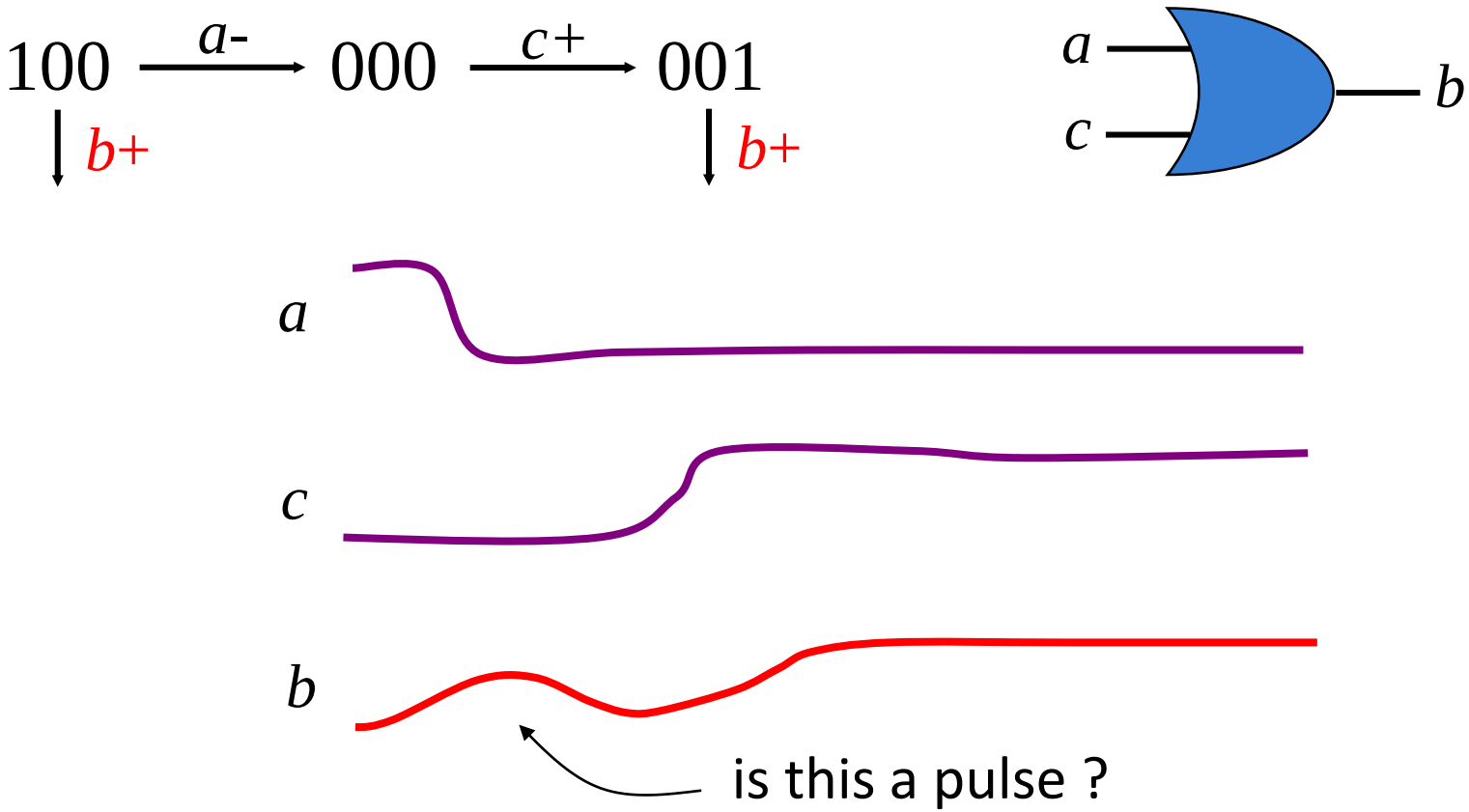
- **Consistency + CSC + output-persistency**



- **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)**

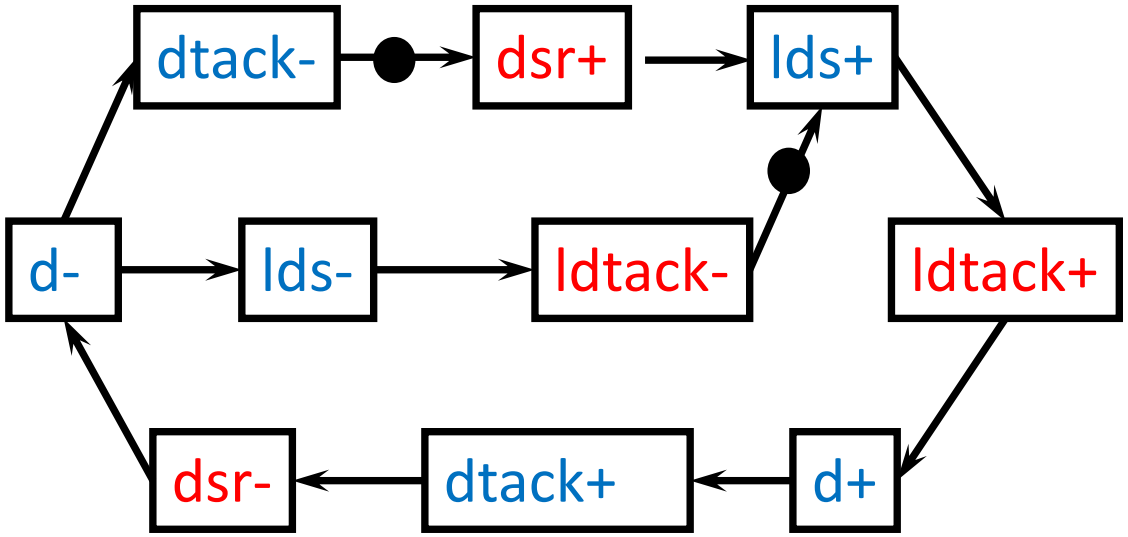
# Persistency



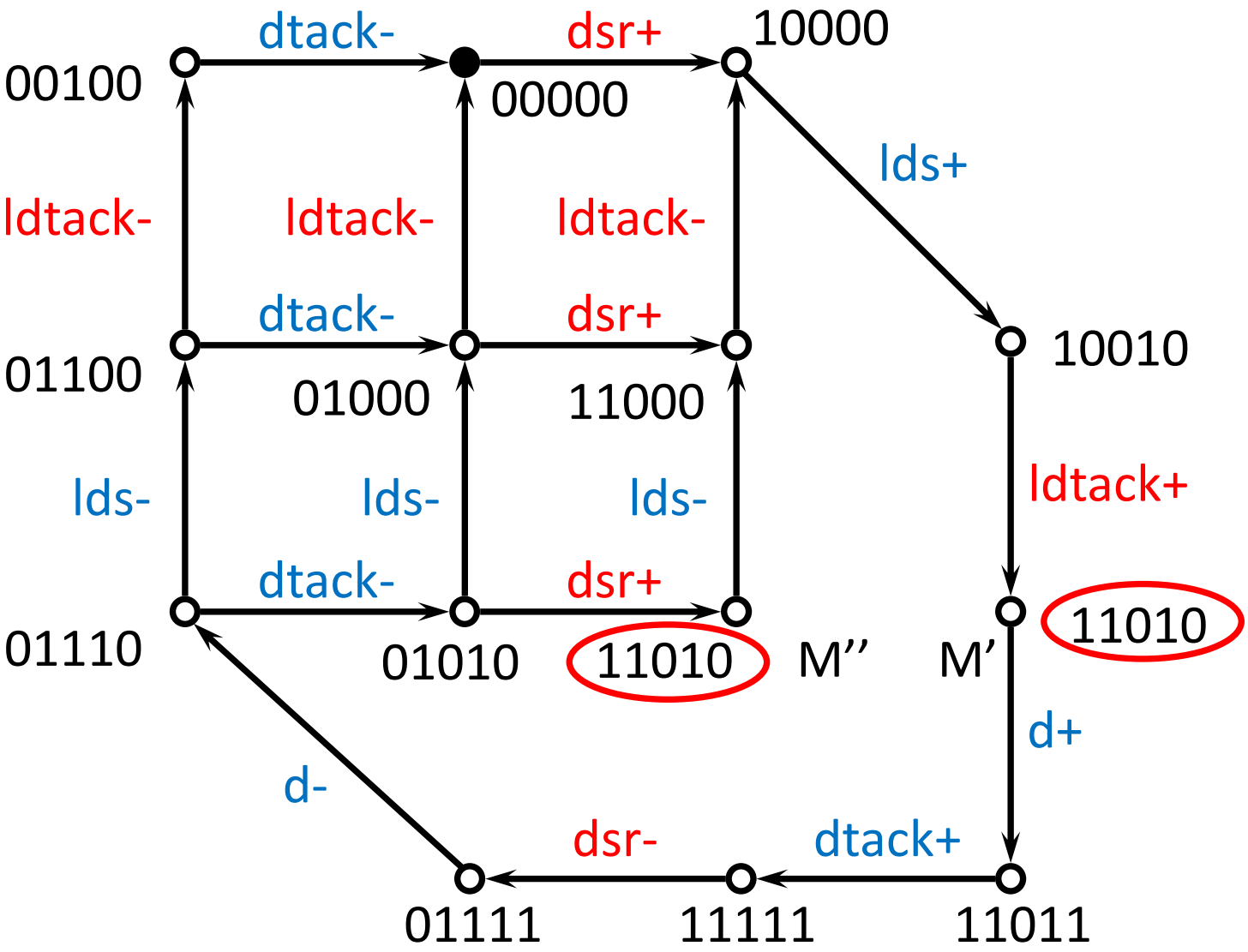

Speed independence ⇒ glitch-free output behaviour under any delay

# Complete State Coding (CSC) Conflict Resolution

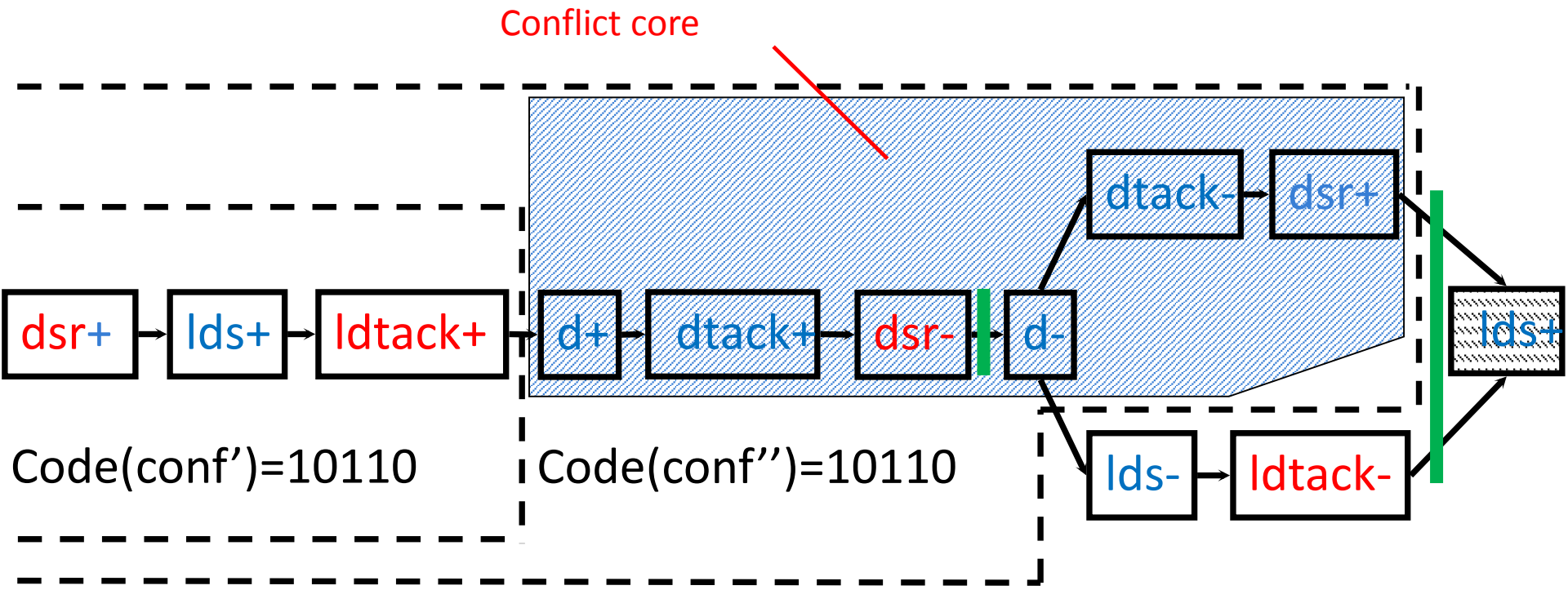
# Example: VME Bus Controller



# Example: CSC conflict



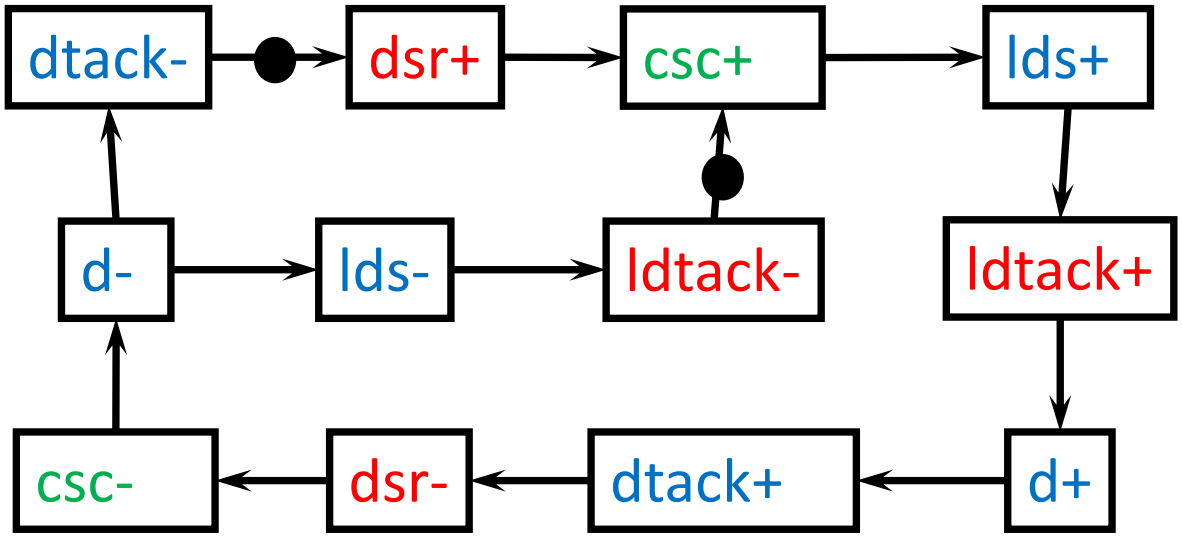
# Example: Resolving the conflict



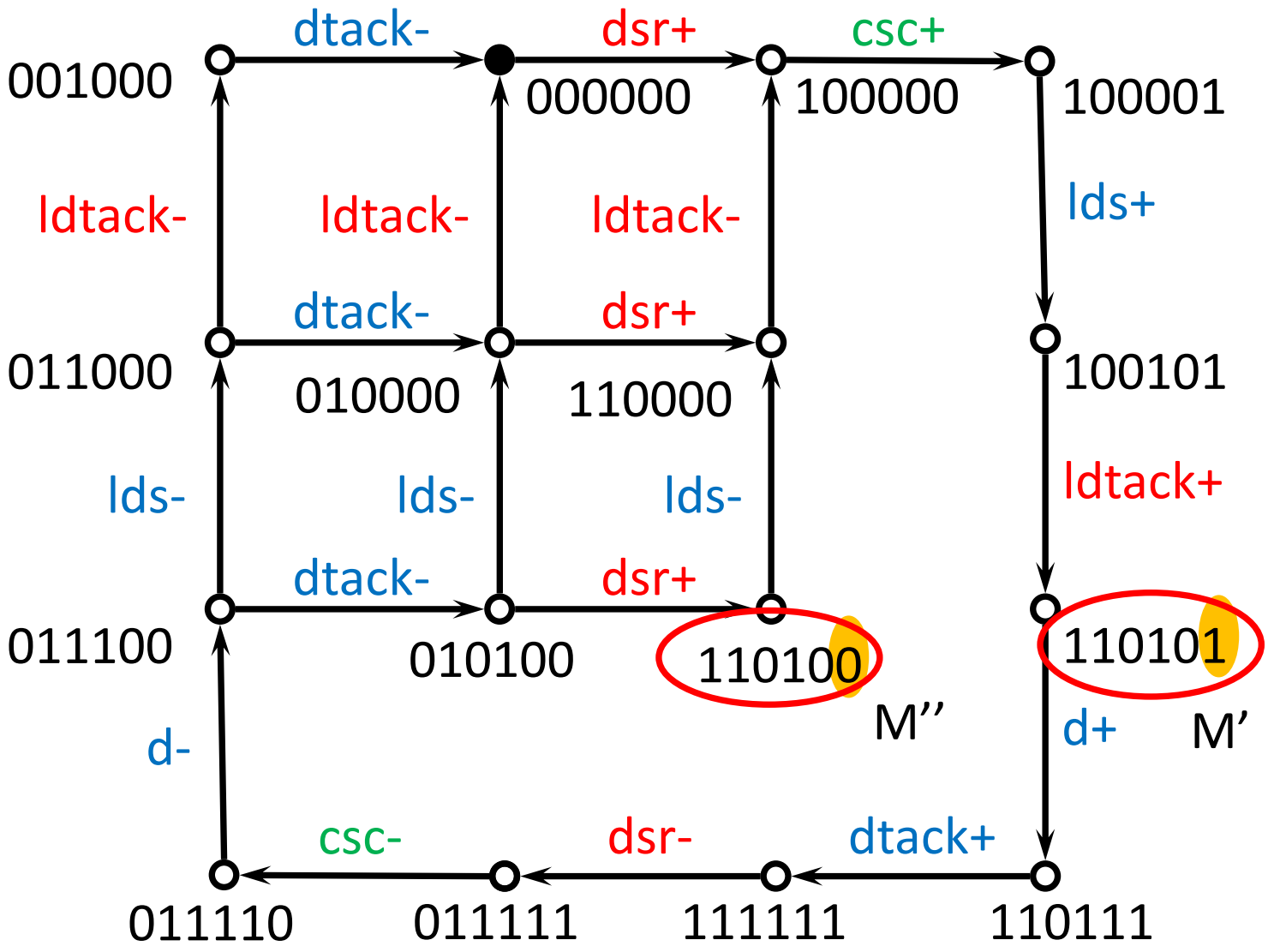

**Idea:** **Insert csc+ into the core and csc- outside the core to break the balance**

**Note:** **Cannot delay inputs!**

# Example: Resolving the conflict

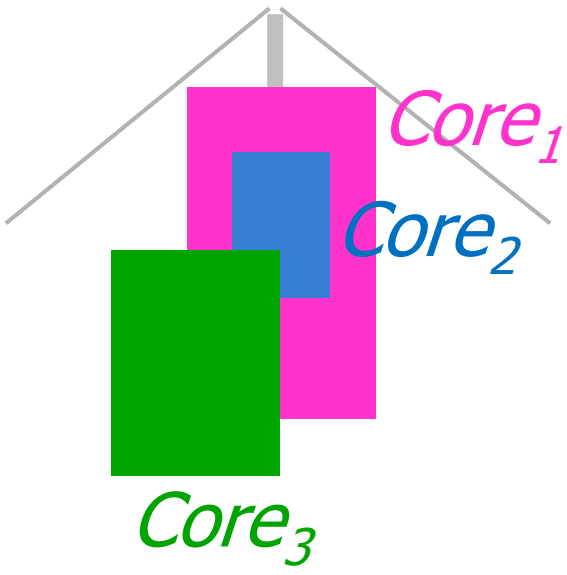


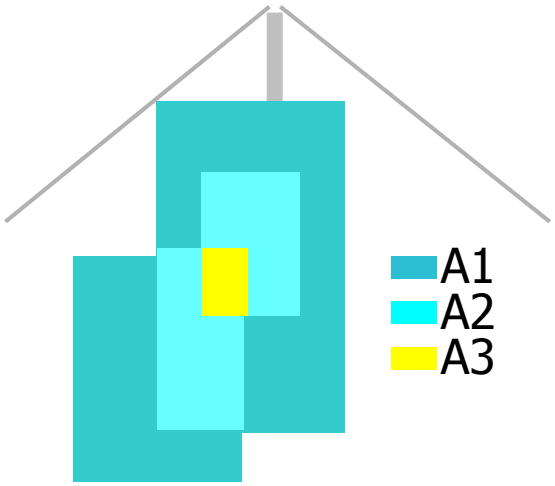
# Example: Resolving the conflict

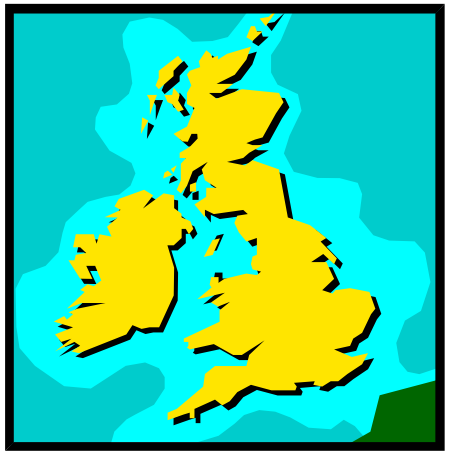


# Core map

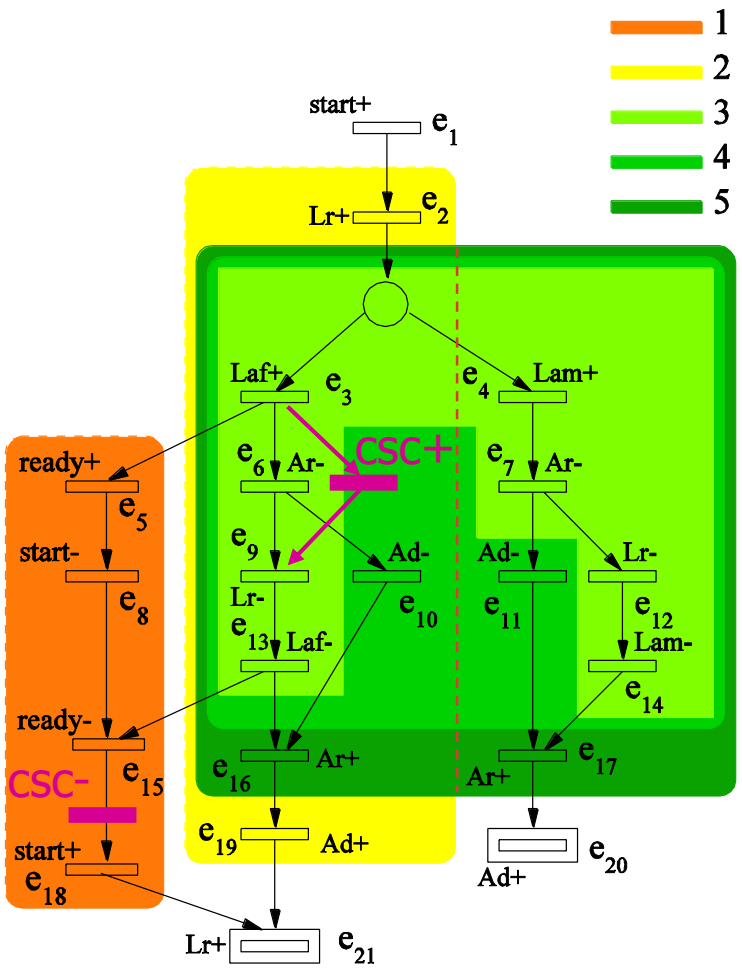
- **Cores often overlap**
- **High-density areas are good candidates for signal insertion**
- **Analogy with topographic maps**



# Example: core map



# Concurrency reduction

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

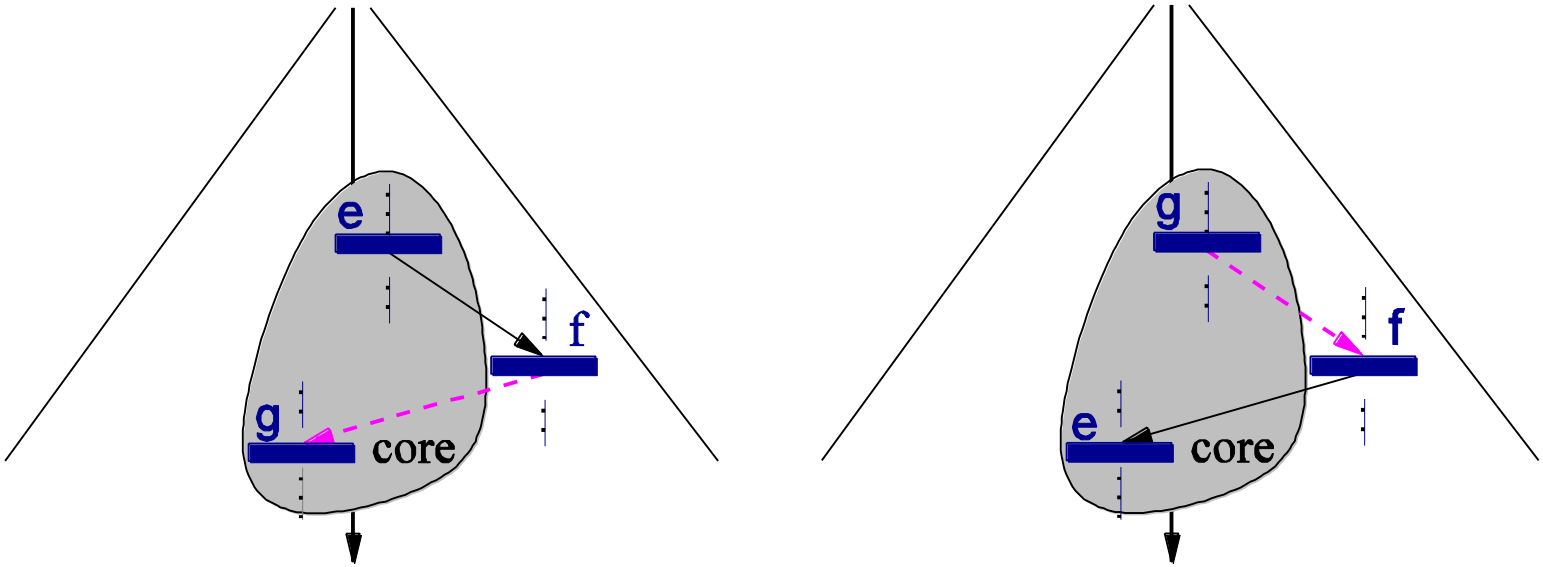
**Note:** Must not delay inputs, i.e. **b** cannot be an input!

**Note:** Changes the behaviour, impacts the environment!

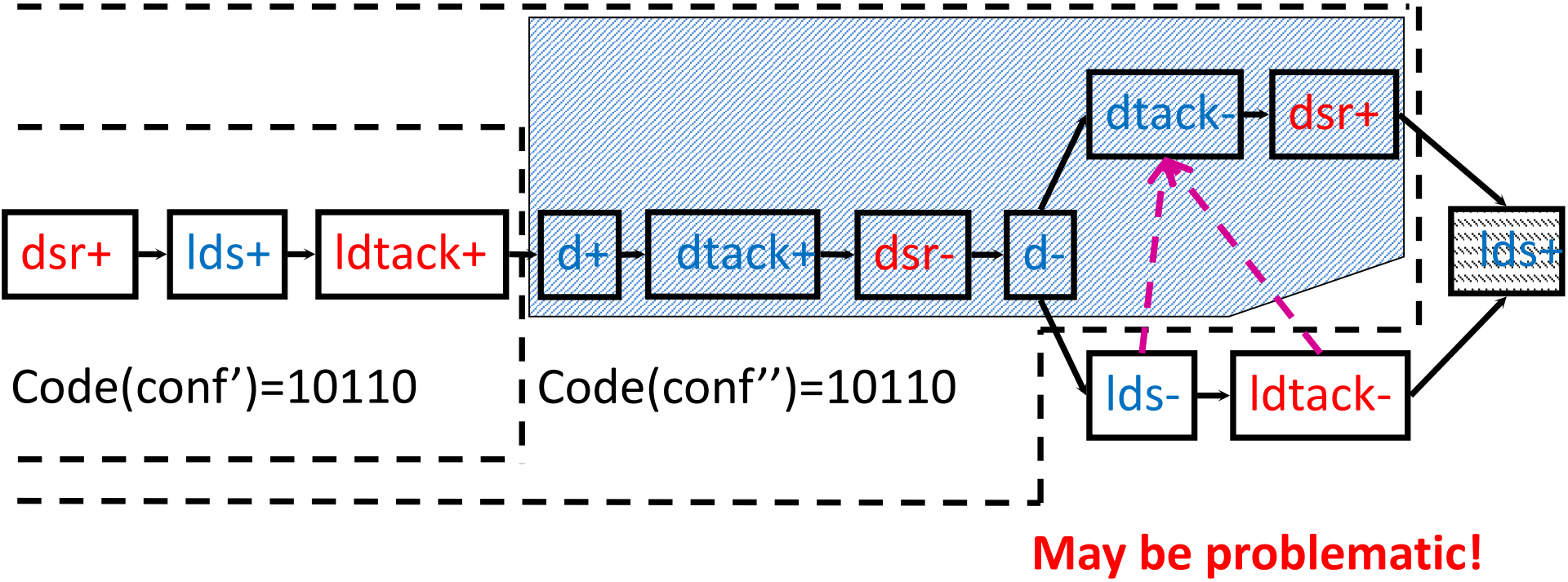
**Heuristic:** Try not to introduce new triggers of **b**, e.g. if there is an arc **a+** → **b+** then **a-** → **b-** is preferred

Used for resolving CSC conflicts and circuit simplification

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

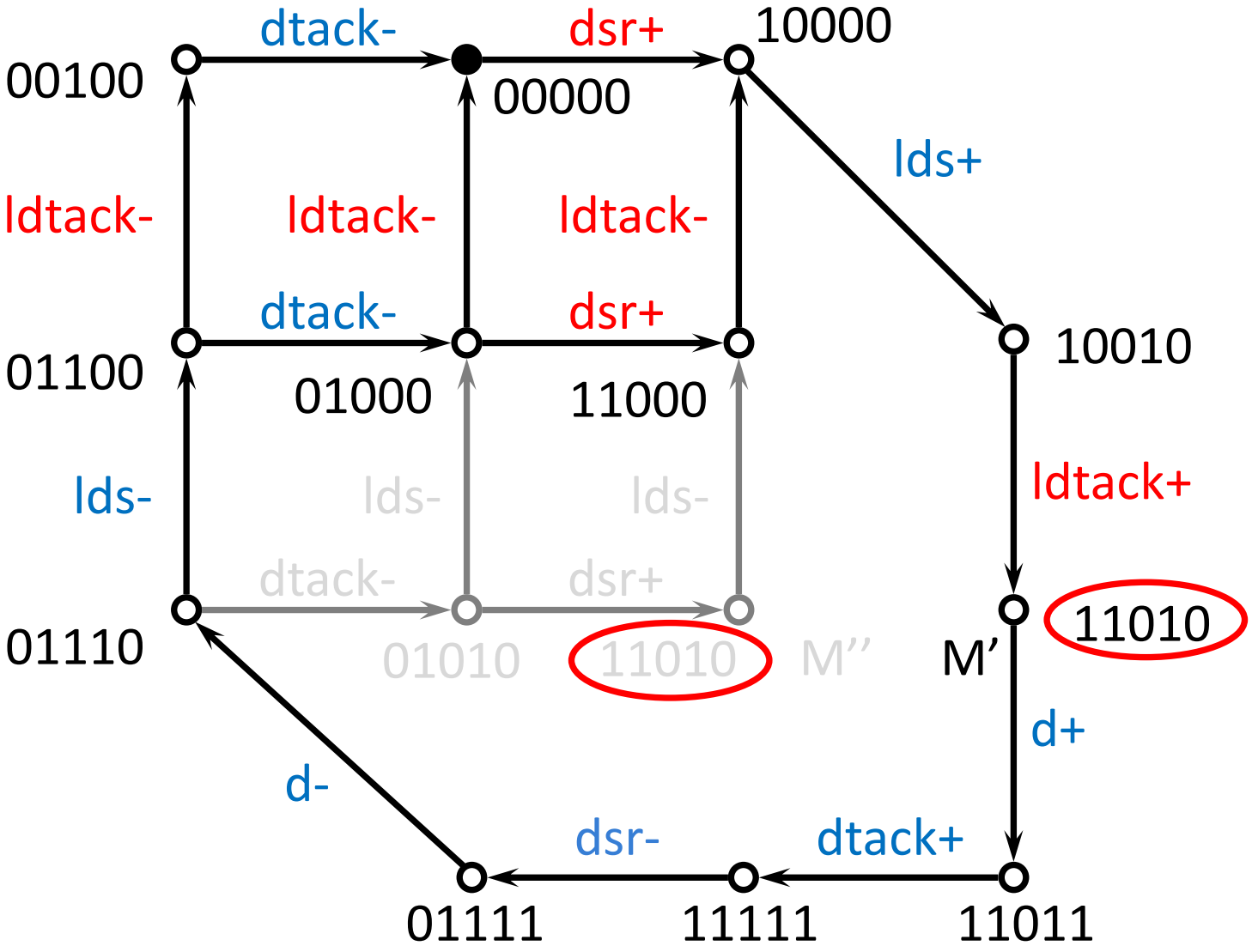


# Example: Resolving the conflict



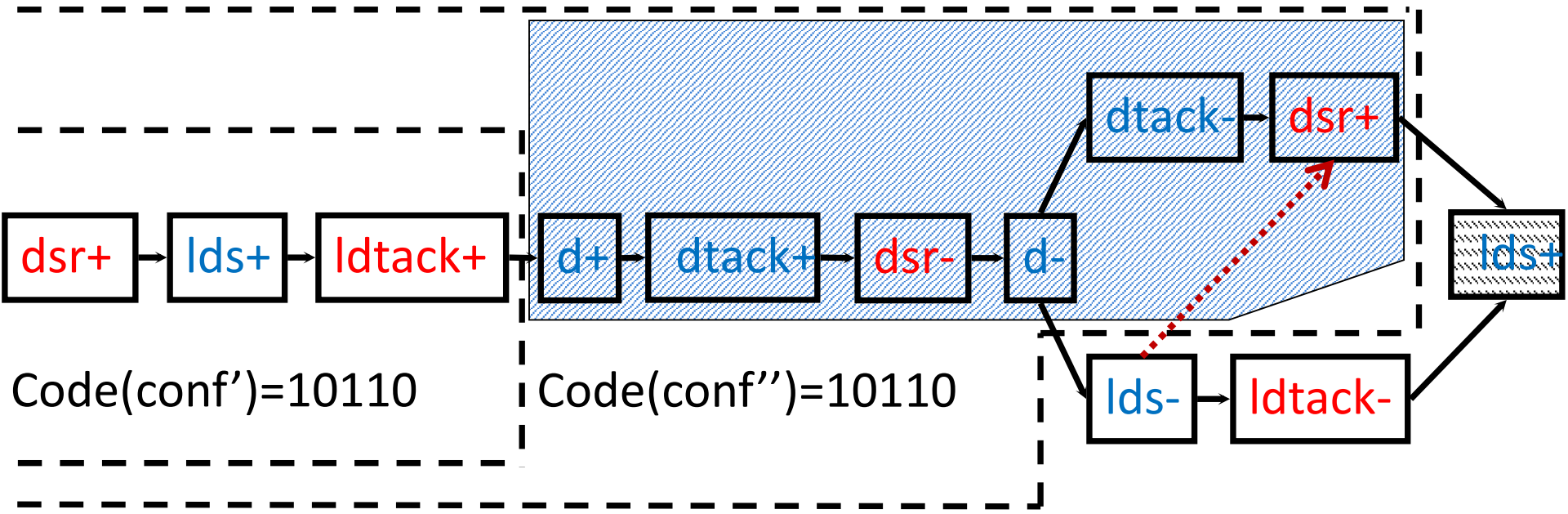

**May be problematic!**

# Example: Resolving the conflict

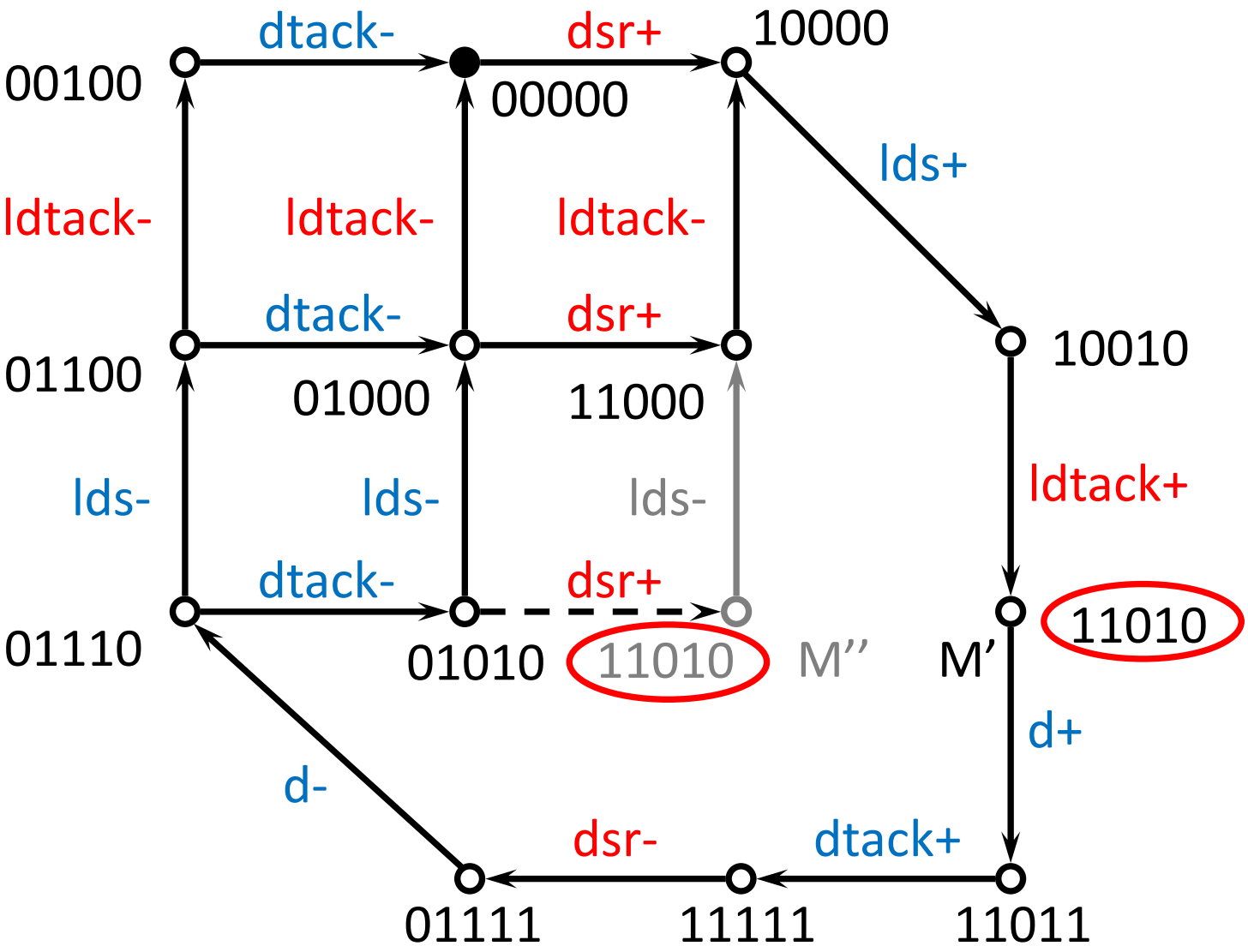


# Relative timing assumptions

- **“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

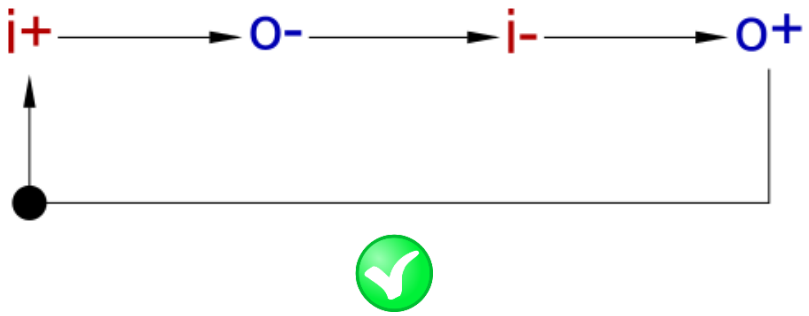
**Standard PN properties:**

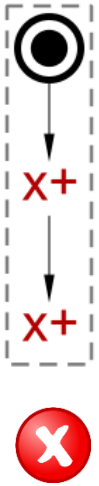
- **boundedness / safeness – a digital circuit has finitely many reachable states**
- **deadlock-freeness**
- **various custom reachability properties, e.g. mutual exclusion**

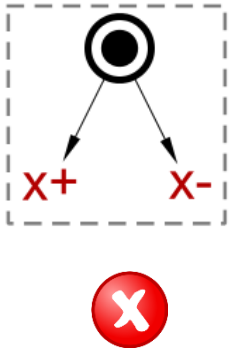
# Verification of STG specification

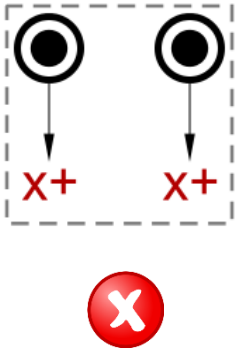
**Consistency:** in each execution, the rising and falling edges of each signal must alternate, always starting from the same edge – reduces to a reachability property

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



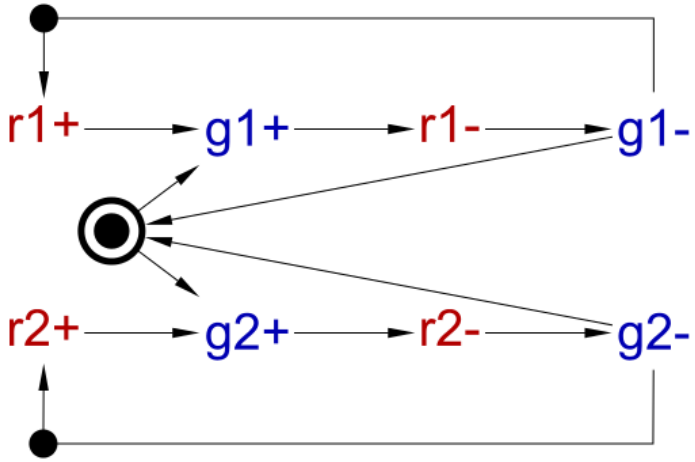



# Verification of STG specification

**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

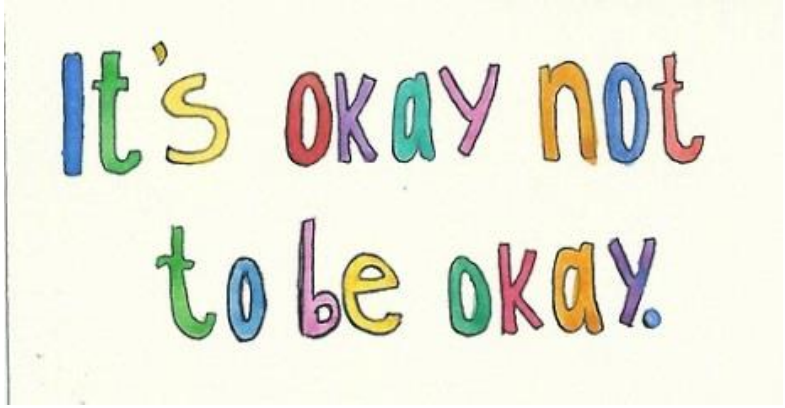


# Verification of STG specification

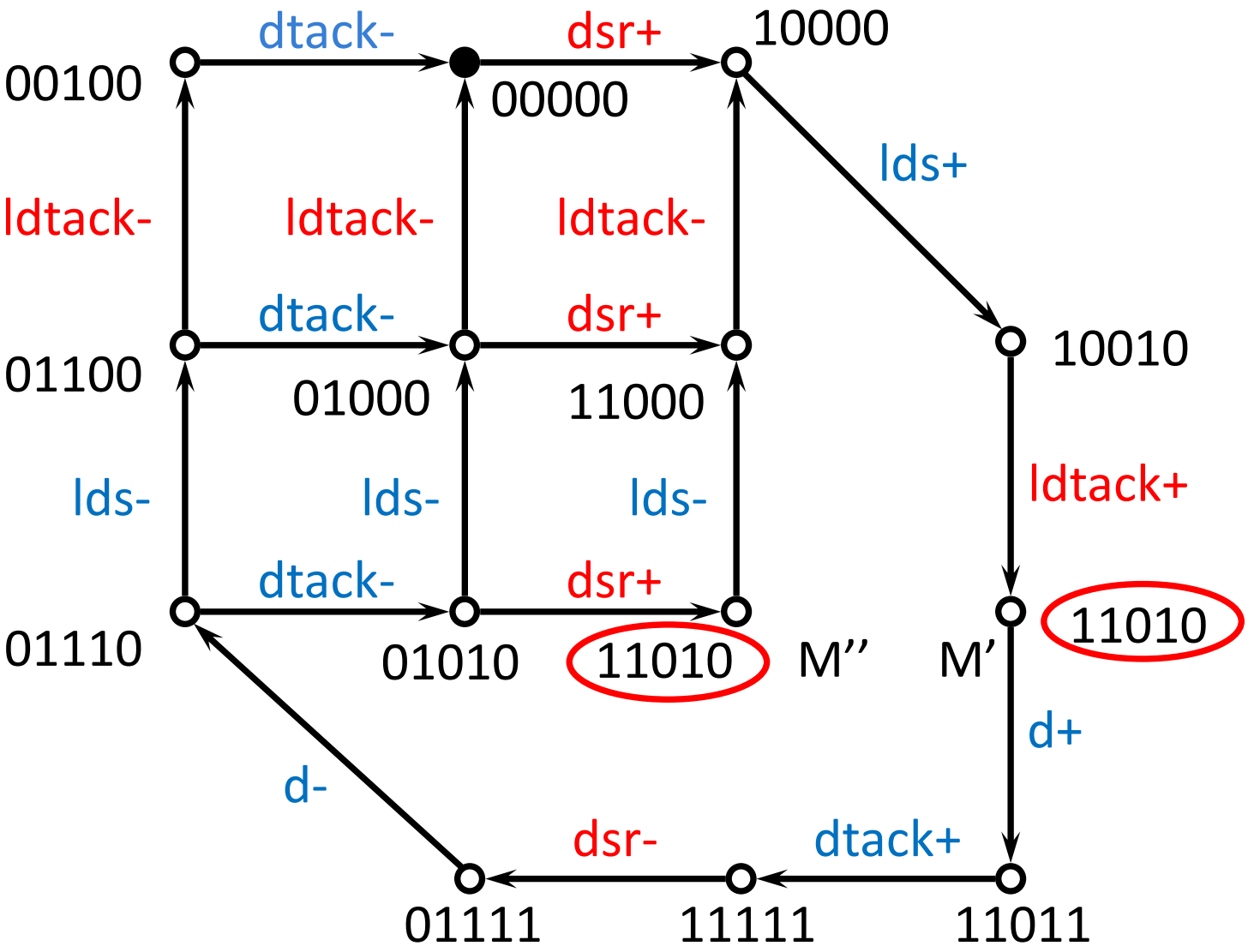
**Complete State Coding (CSC): If two reachable states have the same values of all signals then they should enable the same outputs; two states violating this property are said to be in CSC conflict**

**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

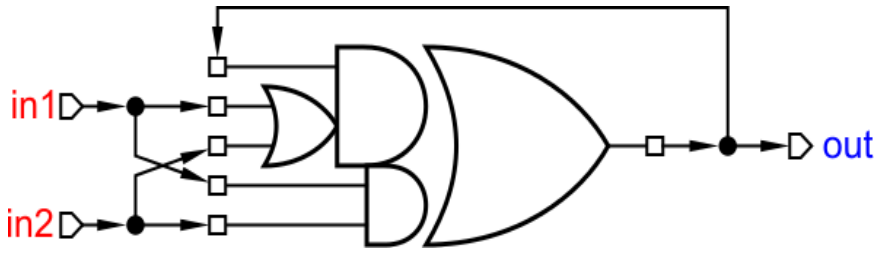


# Verification of the circuit

## 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

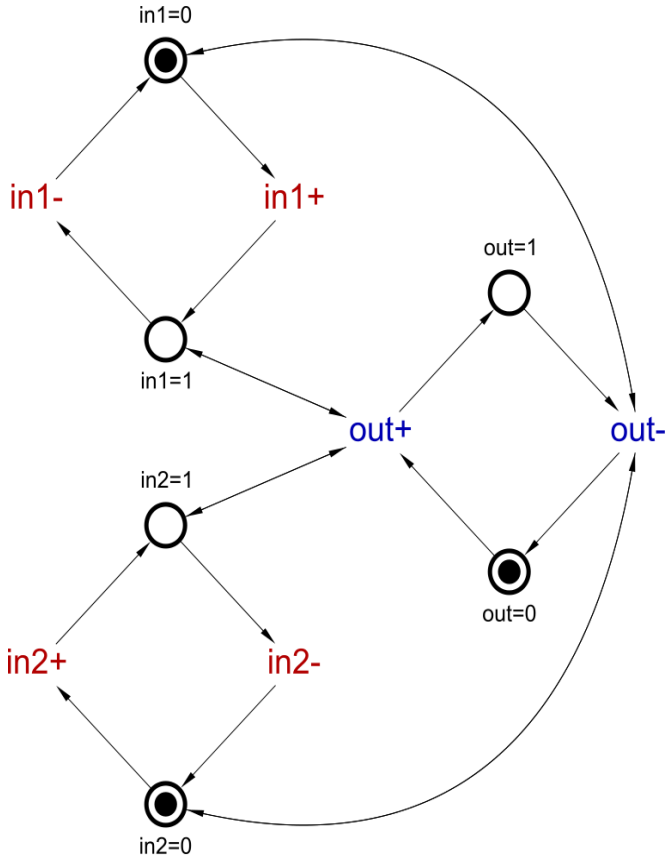



$[out] = out\cdot(in1 + in2) + in1\cdot in2$

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

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

- 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

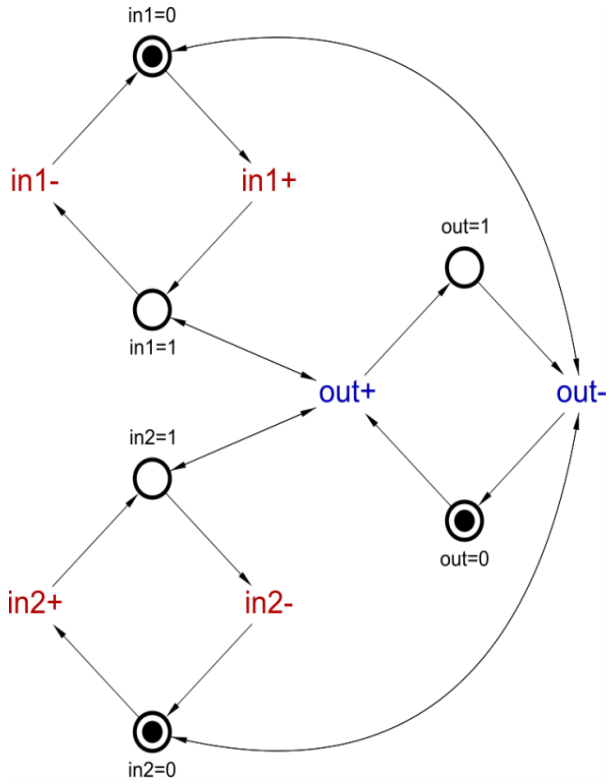
Gate-level circuit has no information about its environment, so naïve verification will always reveal hazards in any non-trivial circuit with inputs. Hence need to supply the environment's behaviour during verification: Assuming the environment fulfils the contract, the circuit must:

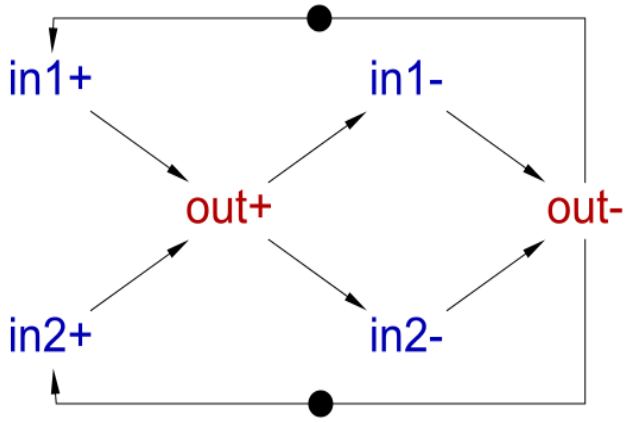
- be free from hazards: no output can be disabled by another signal (except in mutex)
- conform to its environment, i.e. never produce an unexpected output – the circuit must fulfil its contract too
- be deadlock-free
- etc.

# 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:

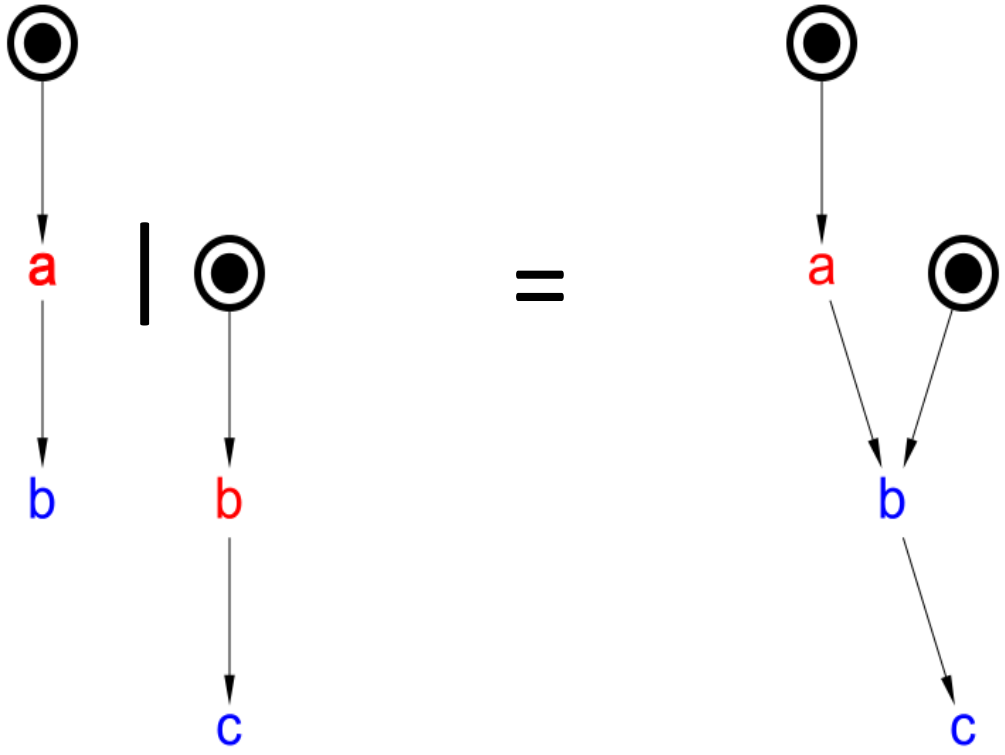



mirror

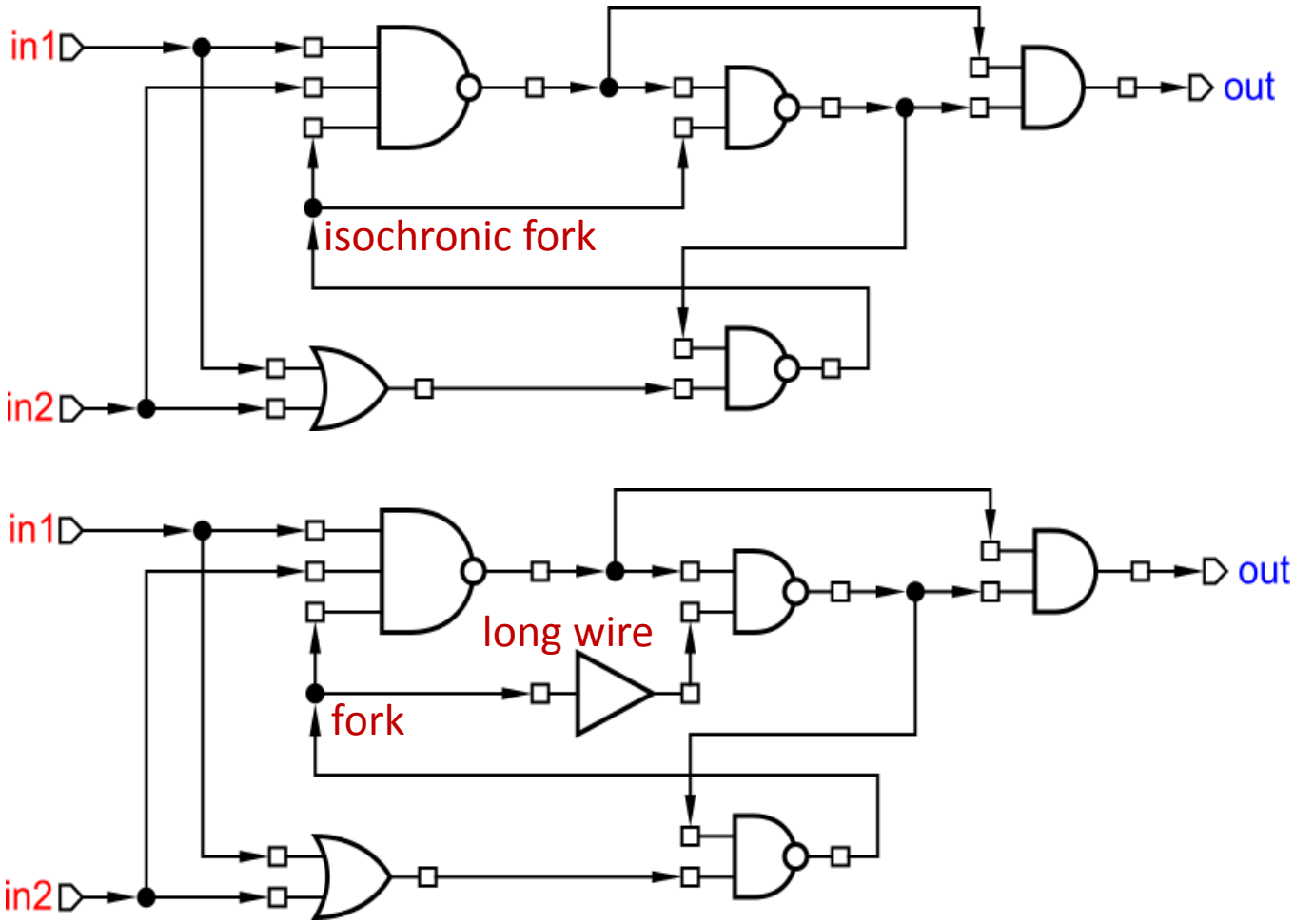
# Parallel composition

- **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

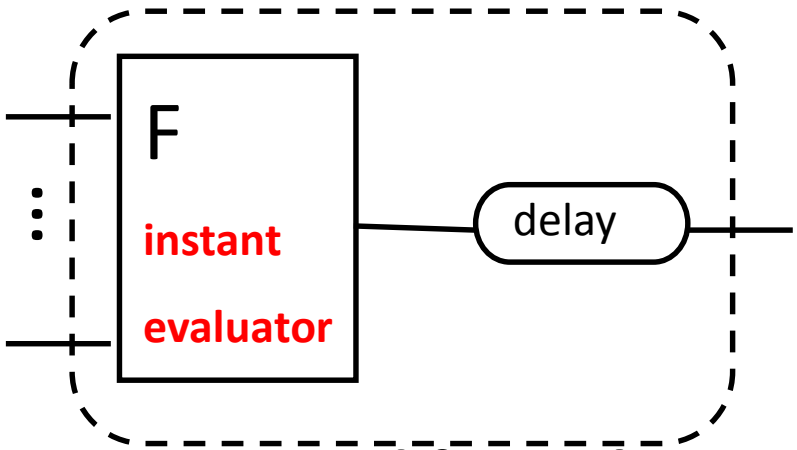
# PCOMP – parallel composition

- **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

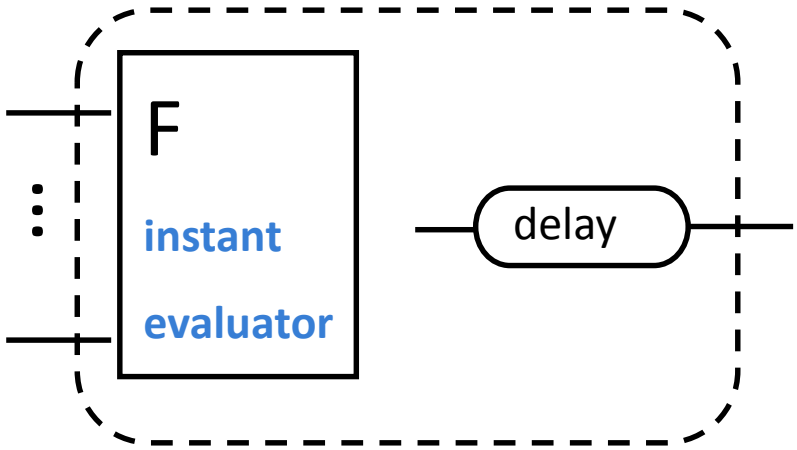
# Speed-independence assumptions

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

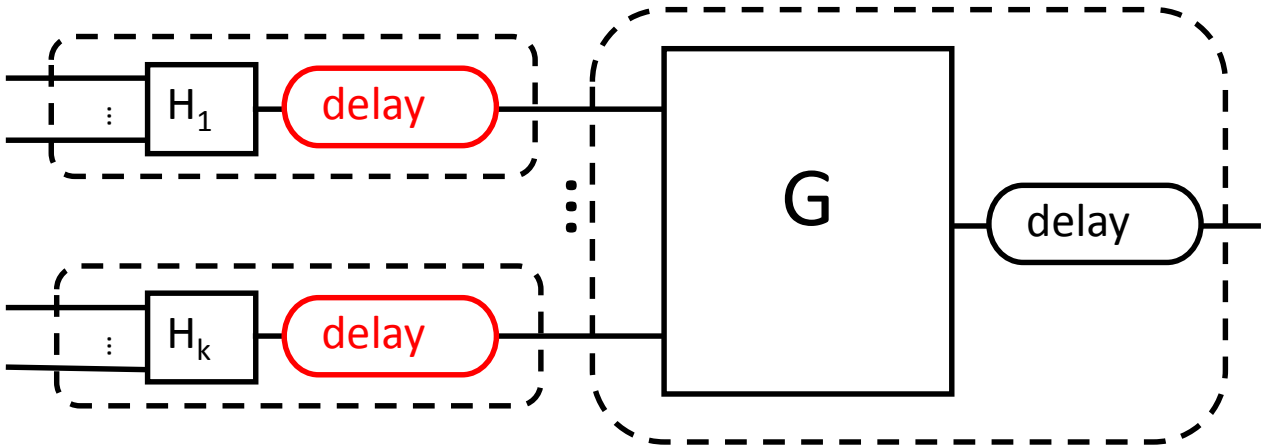



- **Gate delays are positive and finite, but variable and unbounded**
- **Wire delays are negligible (SI)**
- **Alternatively, [some] wire forks are isochronic (QDI), i.e. wire delays can be added to gate delays**

# SI decomposition




**Hazards can be introduced due to these delays!**



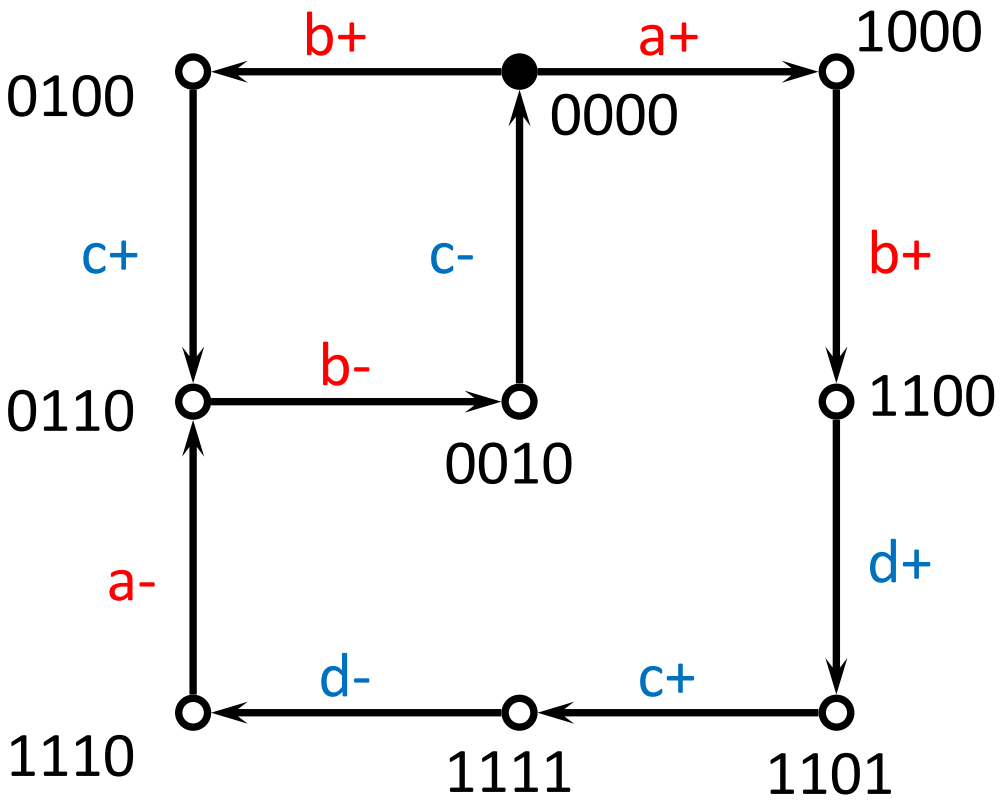
# Gates & latches

- 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

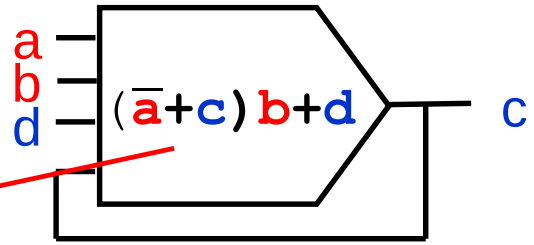
- Encoding (CSC) conflicts must be resolved first
- Several kinds of implementation can then be derived automatically:
  - complex-gate (CG)
  - generalised C-element (gC)
  - standard-C implementation (stdC)
- Can mix implementation styles on per-signal basis
- Logic decomposition may still be required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

# Example: complex-gate synthesis



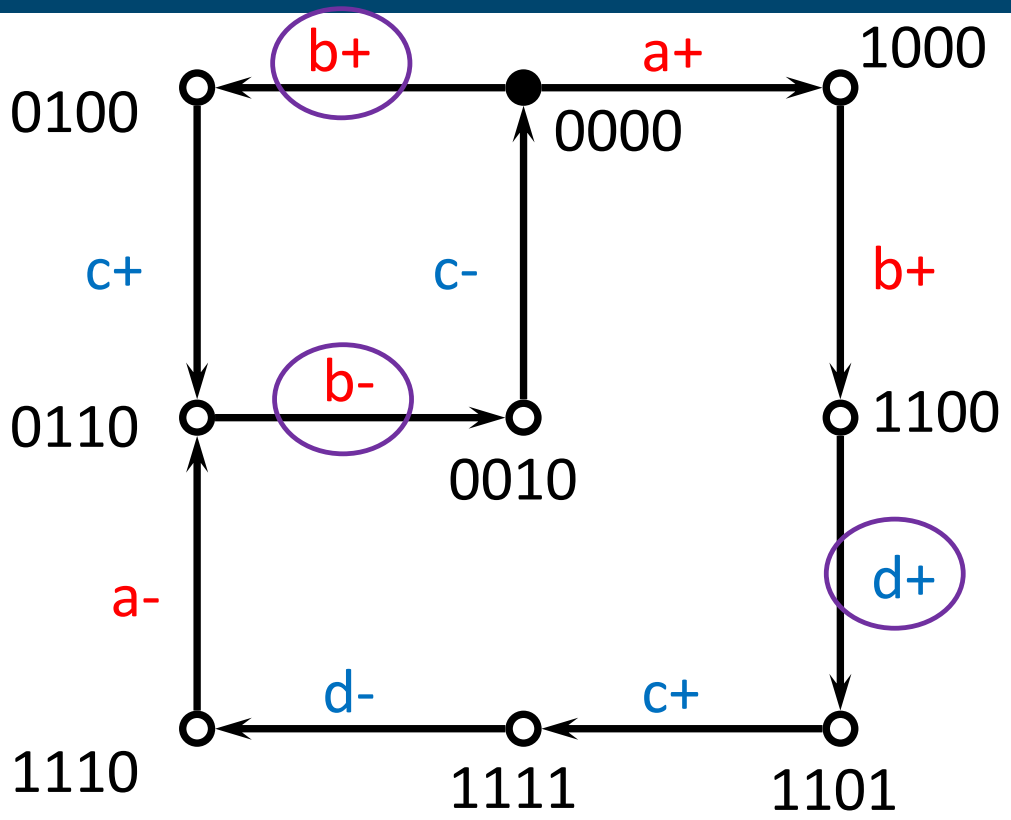

| Code | $Nxt_c$ |
|---|---|
| 0100 | 1 |
| 0000 | 0 |
| 1000 | 0 |
| 0110 | 1 |
| 0010 | 0 |
| 1100 | 0 |
| 1110 | 1 |
| 1111 | 1 |
| 1101 | 1 |
| else | - |
| Eqn | $(\bar{a}+c)b+d$ |

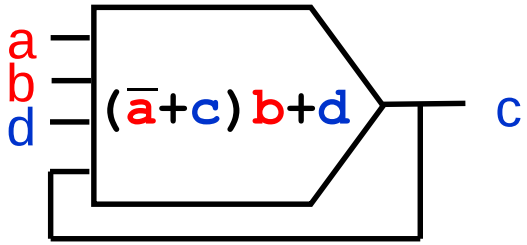
$$Nxt_z(s) = Code_z(s) \oplus Out_z(s)$$




The size of this Boolean expression is not limited!

# Support, triggers and context



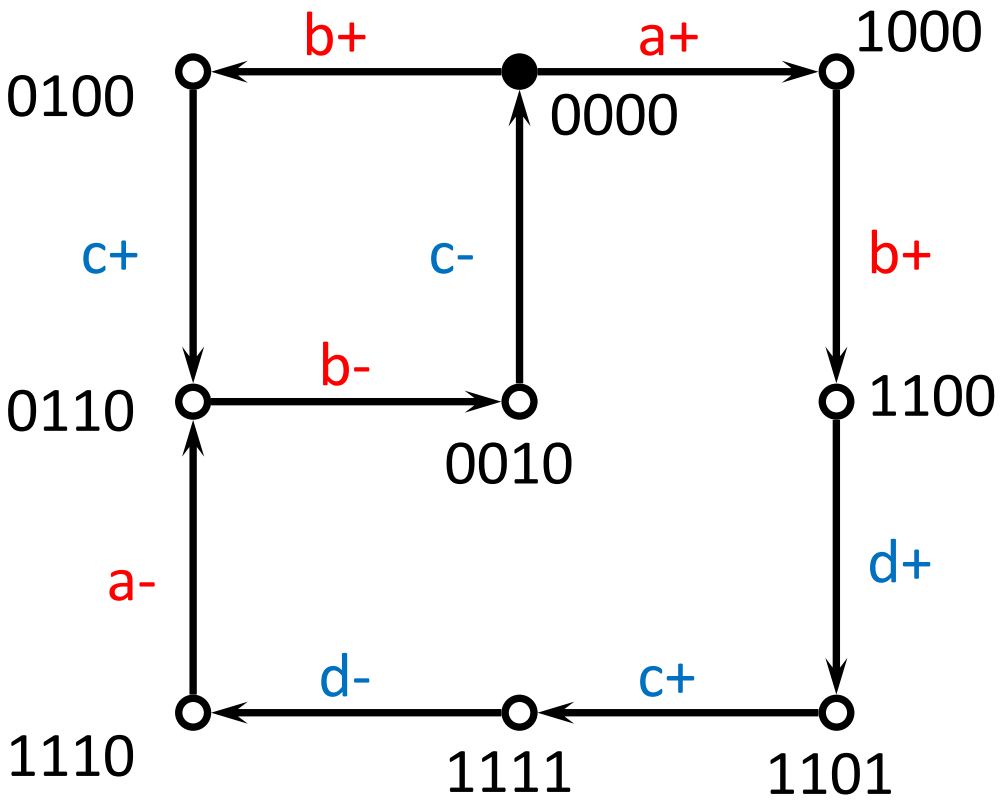

Signals that are the inputs of the gate producing a signal form its **support**, e.g. the support of c is {a,b,c,d}. Supports are not unique in general.

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.

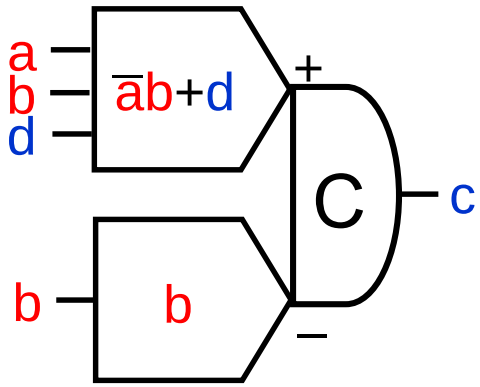
**support = triggers + context**

# Example: gC implementation



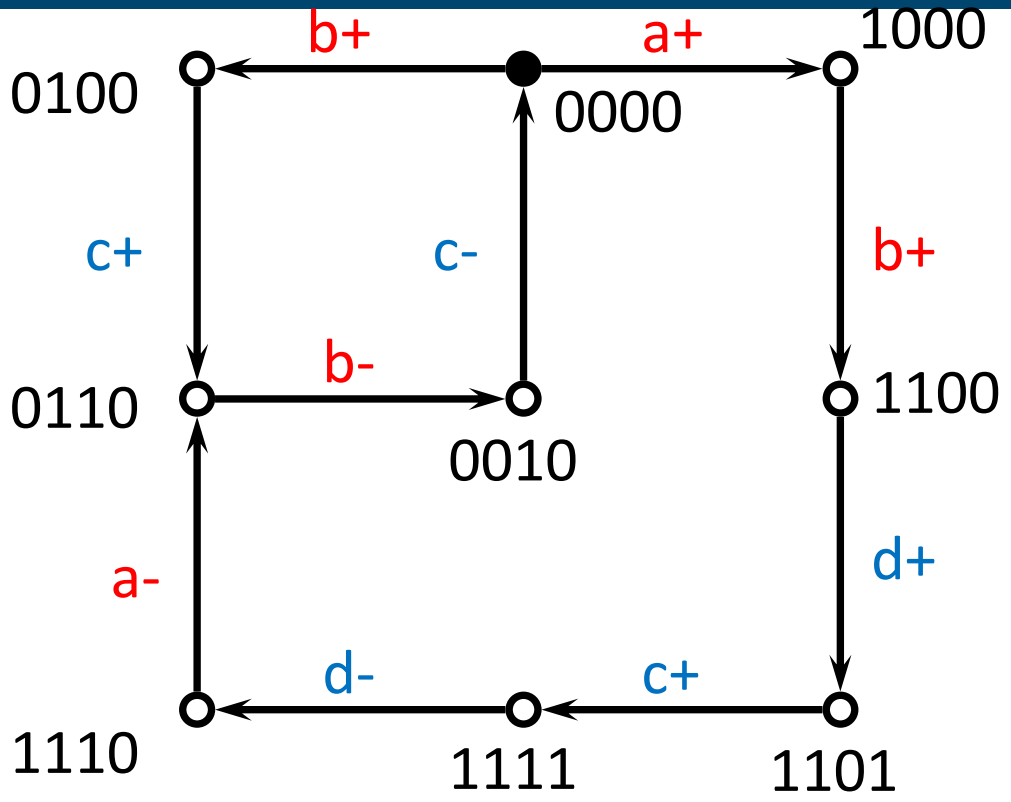

| Code | $Set_c$ | $Reset_c$ |
|---|---|---|
| 0100 | 1 | 0 |
| 0000 | 0 | – |
| 1000 | 0 | – |
| 0110 | – | 0 |
| 0010 | 0 | 1 |
| 1100 | 0 | – |
| 1110 | – | 0 |
| 1111 | – | 0 |
| 1101 | 1 | 0 |
| else | – | – |
| Eqn | $\bar{a}b+d$ | $\bar{b}$ |

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

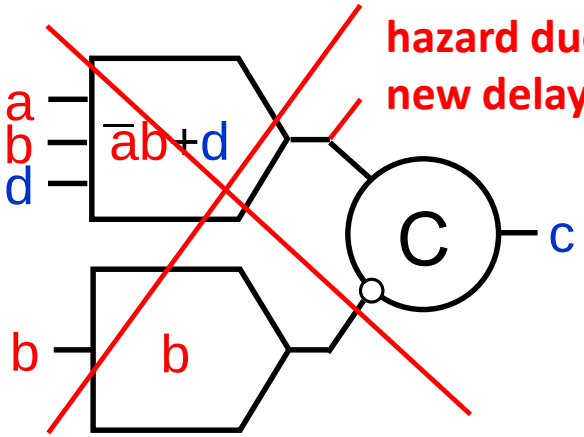


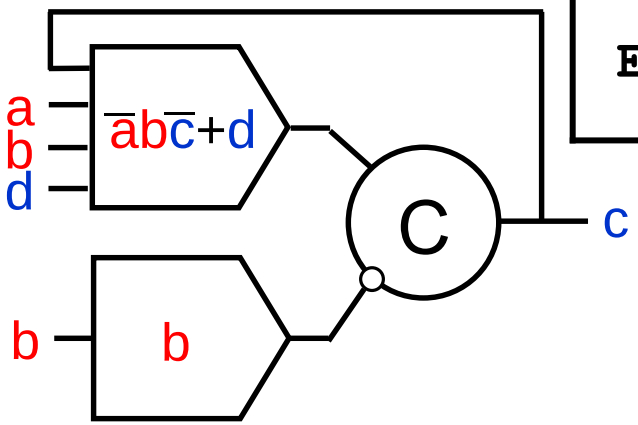

Implemented as pull-up and pull-down networks of transistors and a ‘keeper’; assumed to be atomic

# 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

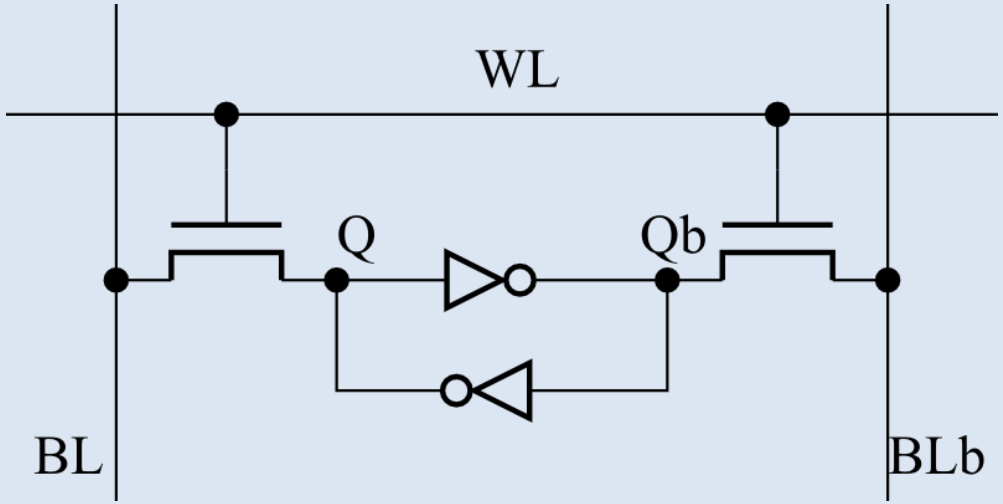
- Often complex-gates are too complex to be mapped to a gate library, and so logic decomposition is required
- Cannot naïvely break up complex-gates – this is likely to introduce hazards (at least, timing assumptions are required)
- Decomposition is one of the most difficult tasks – no guarantee that 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’

# Design examples: SRAM, buck, ADC

Andrey Mokhov, Alex Yakovlev
Danil Sokolov, Victor Khomenko

Newcastle University, UK
async.org.uk; workcraft.org

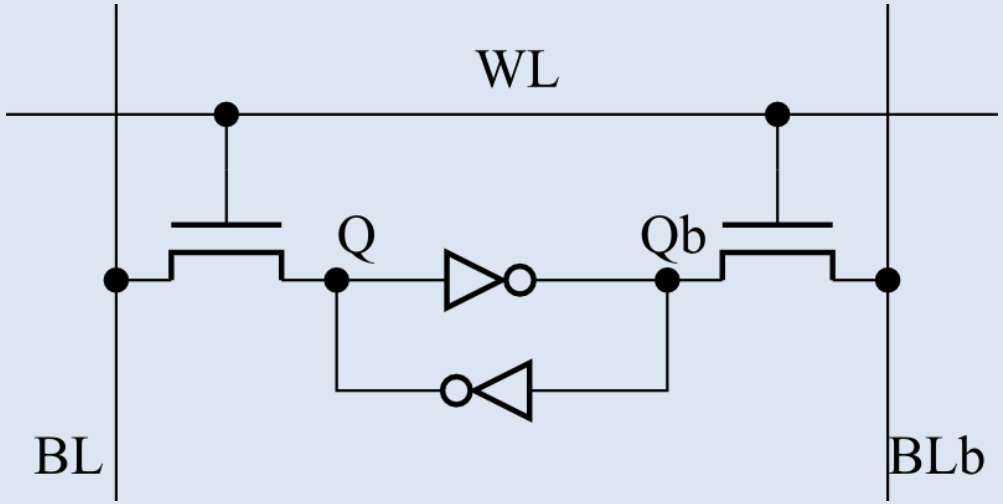
# Conventional 6T SRAM



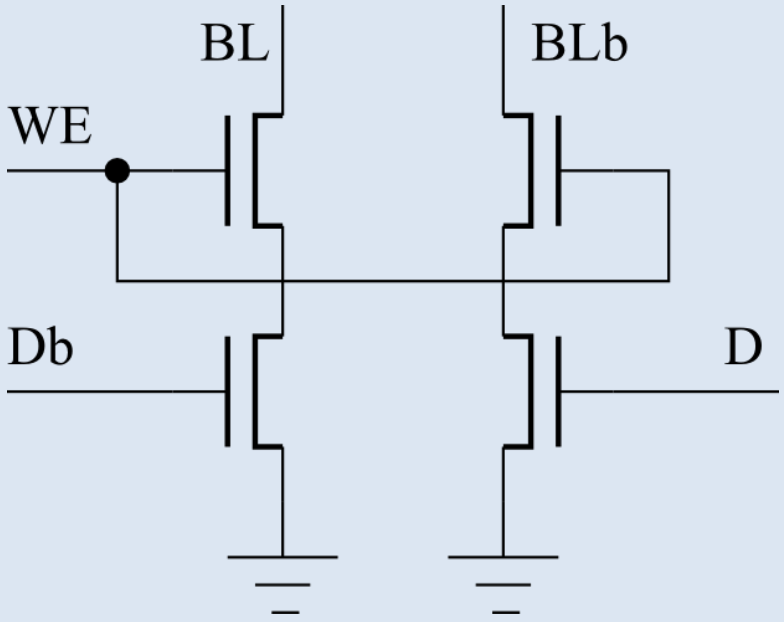

Bit cell

**Reading:** precharge bit lines, assert WL, **sense bit line changes**

# Conventional 6T SRAM



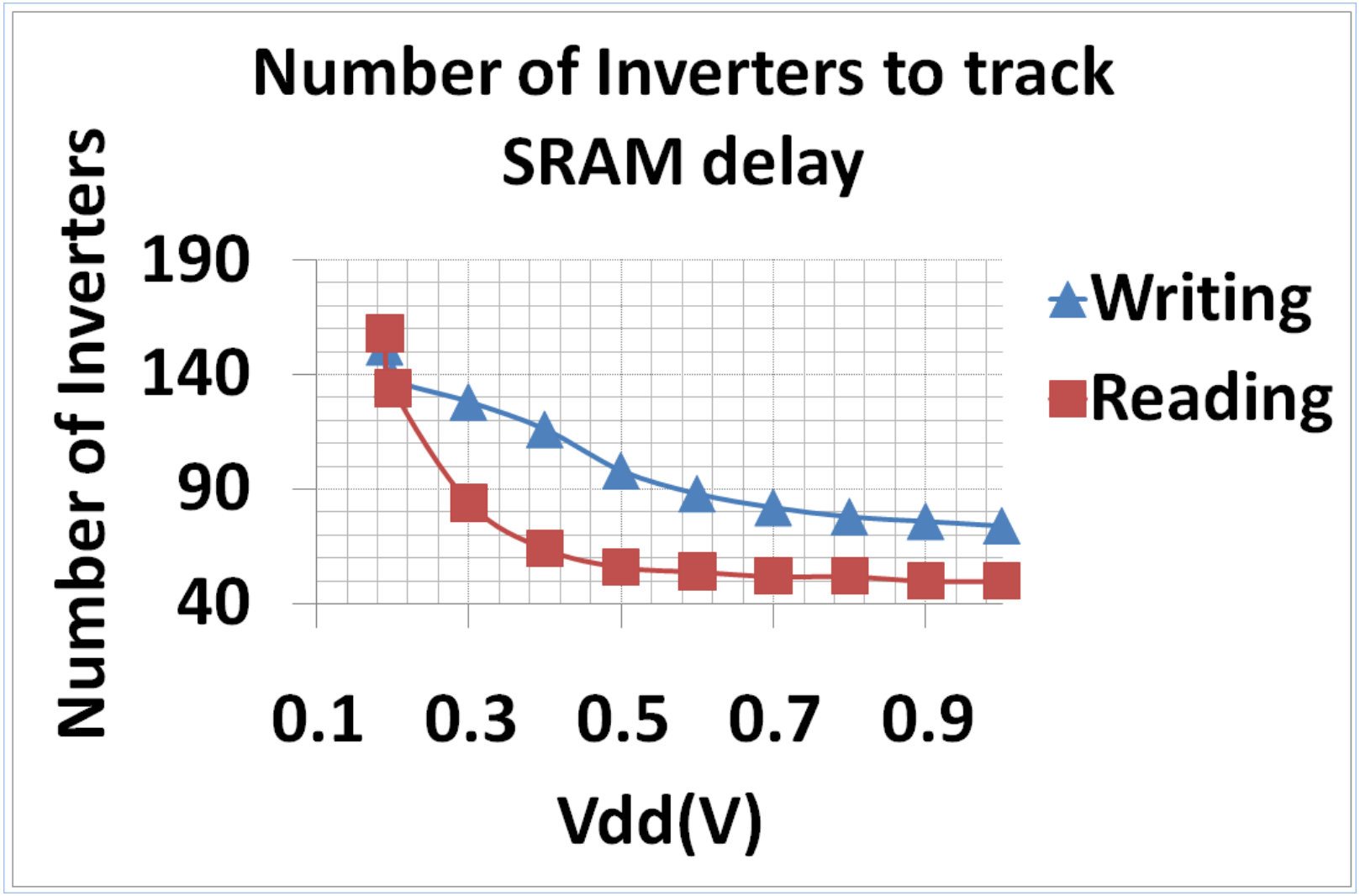

Bit cell




Write driver

**Writing:** set data lines, assert WL and WE, **wait for a while…**

# Problem: how long to wait?



# Problem: how long to wait?

SRAM 6T cell delays are difficult to match accurately

- When $V_{dd}$ = 1V, SRAM read delay is ≈50 inverters
- When $V_{dd}$ = 190mV, SRAM read delay is ≈158 inverters
- Read and write delays scale differently with $V_{dd}$

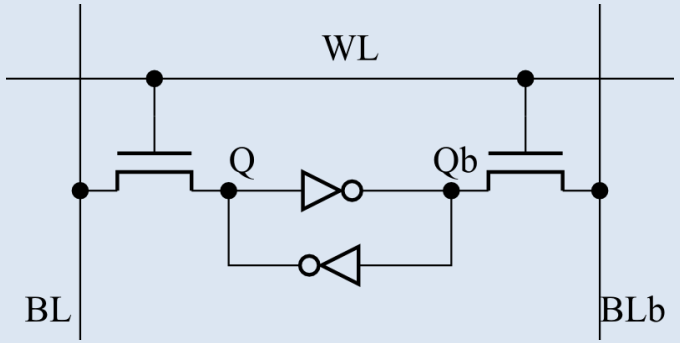
Conventional solutions

- Use different delay lines for different ranges of $V_{dd}$
- Duplicate an SRAM line to act as a reference delay line
- Need voltage references, costly in area and energy

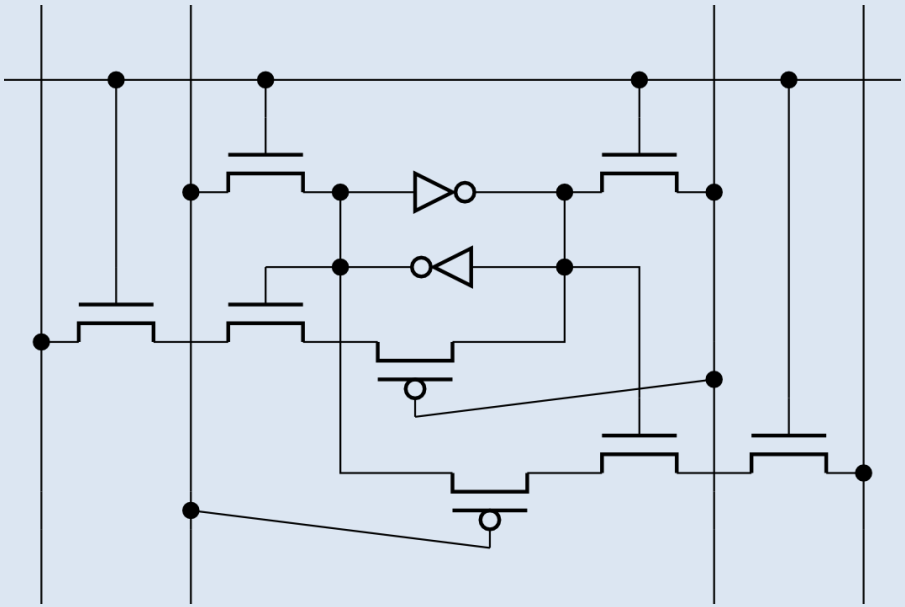
Asynchronous solution with completion detection

- Speed-independent, free from voltage references
- Developed by A. Baz *et al.* (PATMOS 2010, JOLPE 2011)

# Low-level completion detection




Bit cell

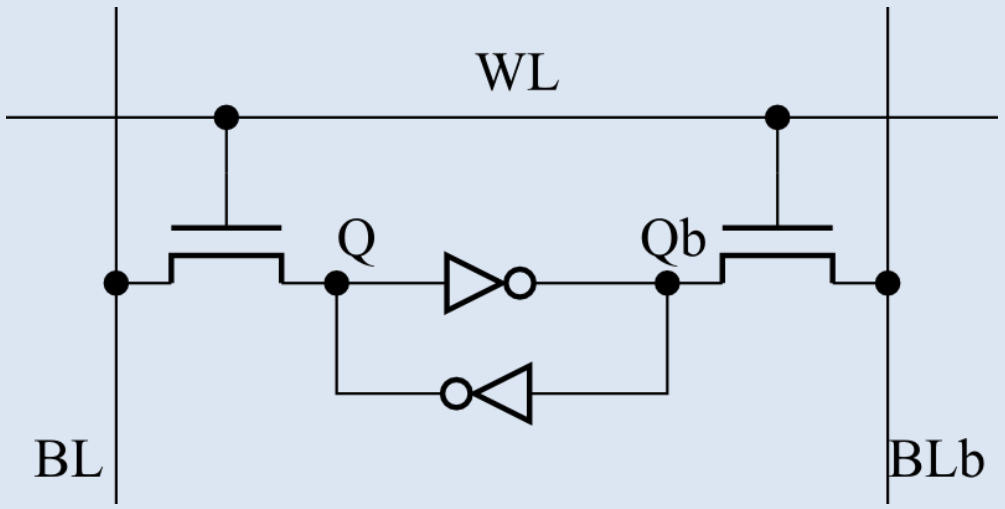


Bit cell with CD

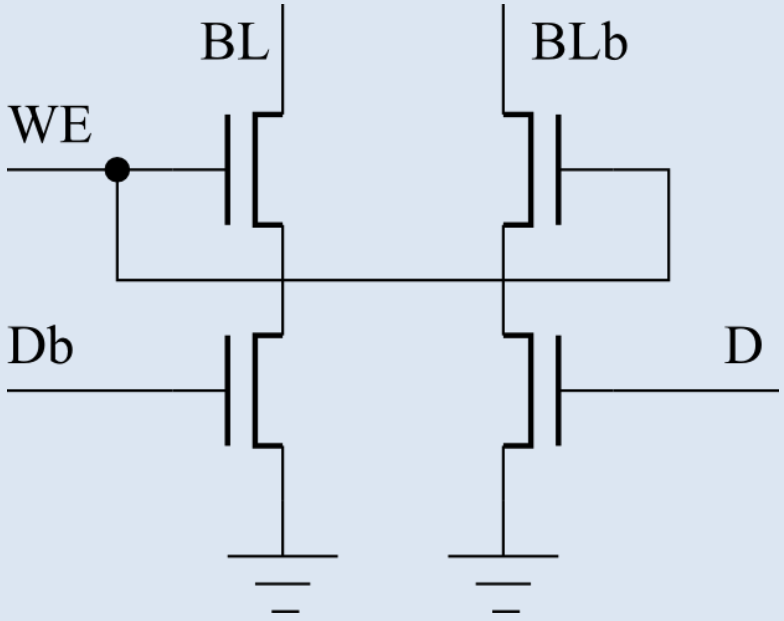
True completion detection **both for reading and writing**

**Too costly!**

# Back to conventional 6T SRAM



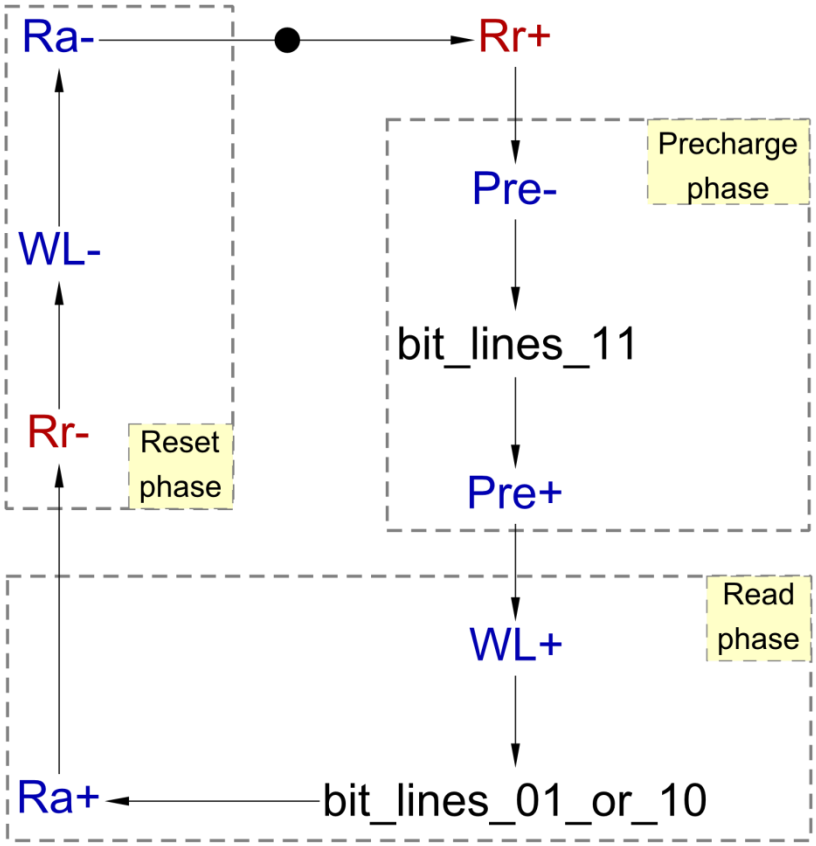

Bit cell



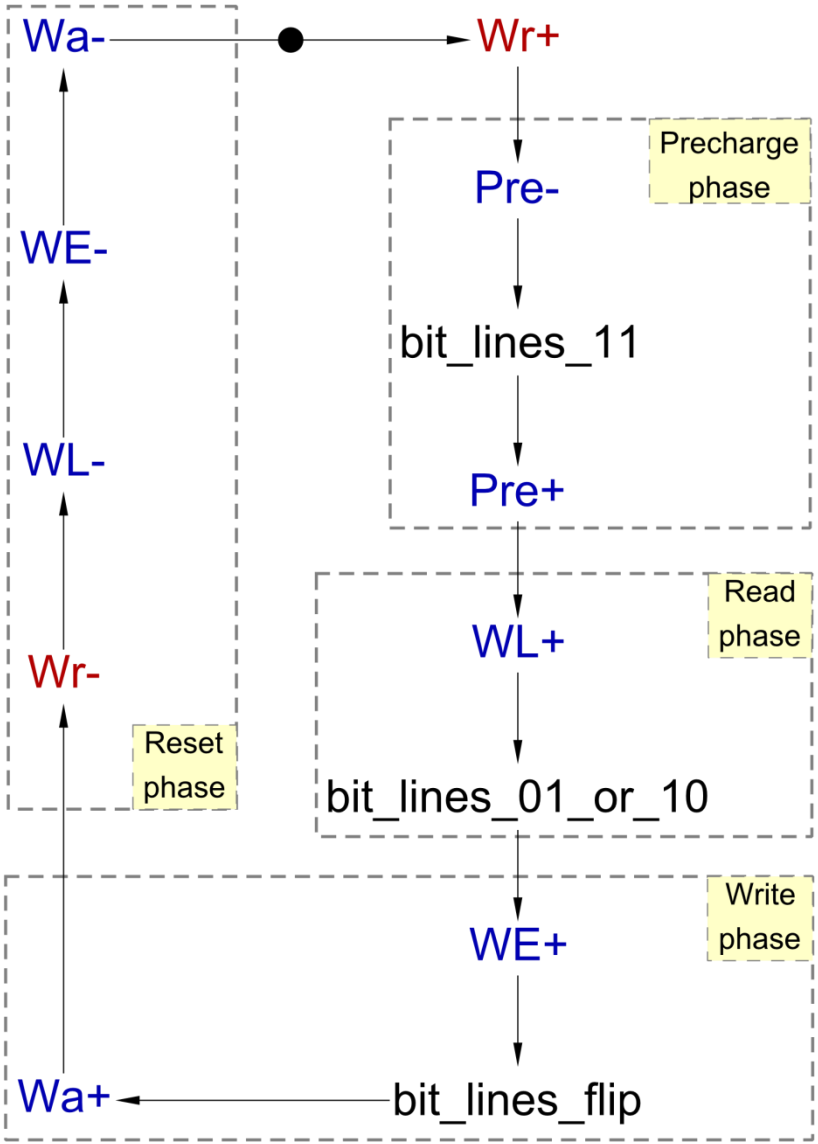

Write driver

**Idea 1:** completion detection is possible when the bit is flipped
**Idea 2:** read before writing to check if the bit will be flipped

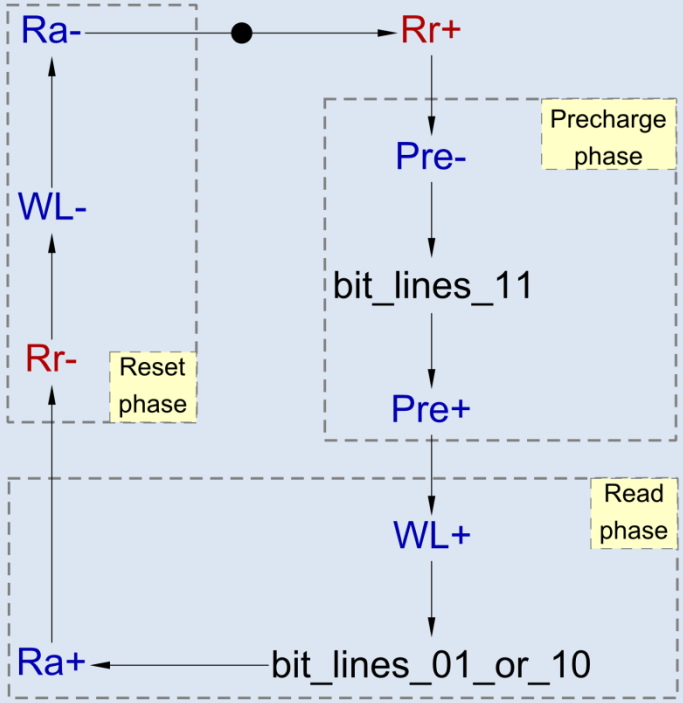
# Specification: read scenario



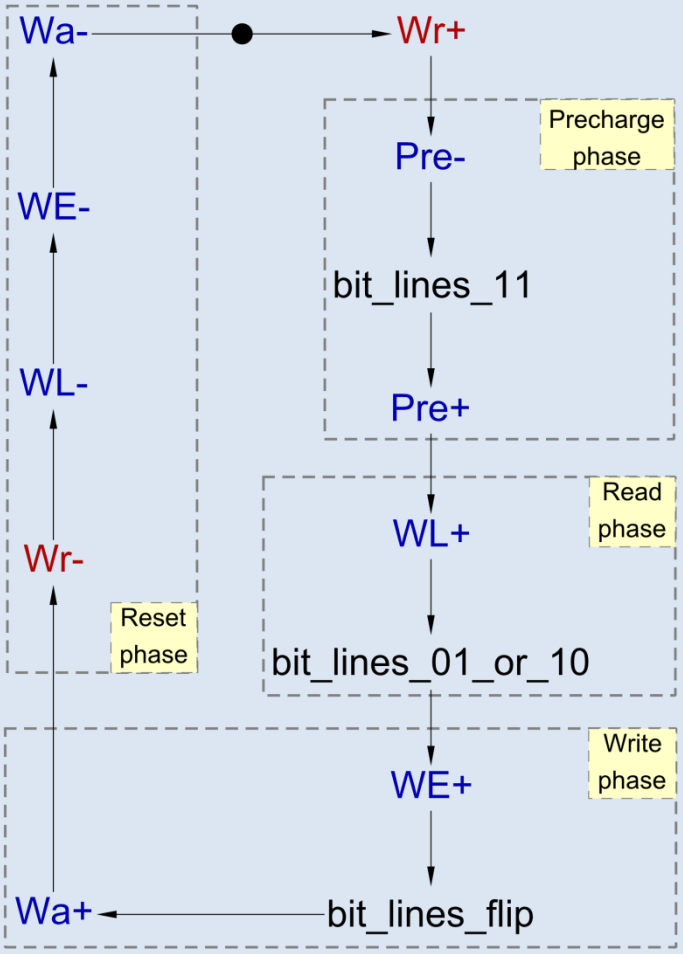
# Specification: write scenario



# Specification: composing scenarios

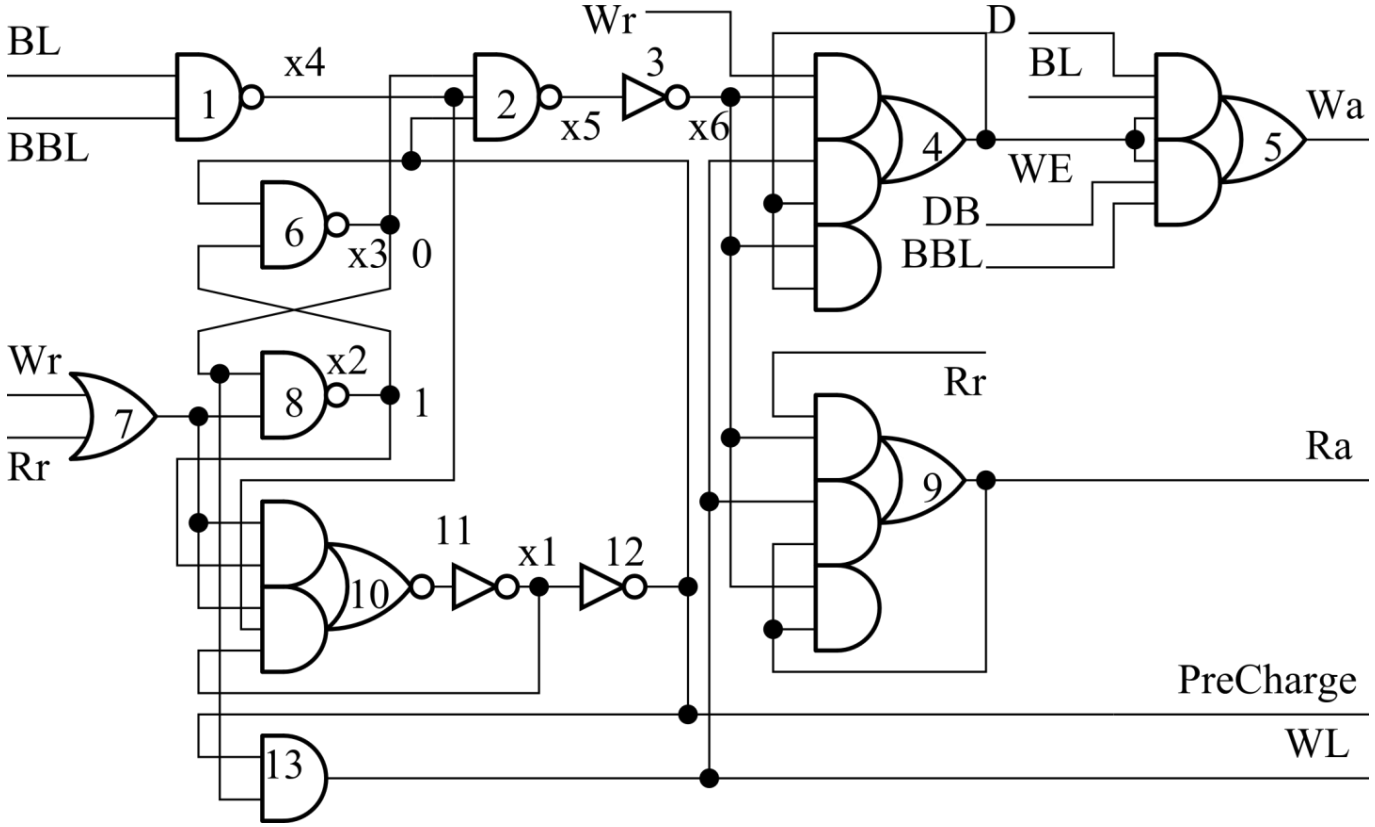



Read




Write

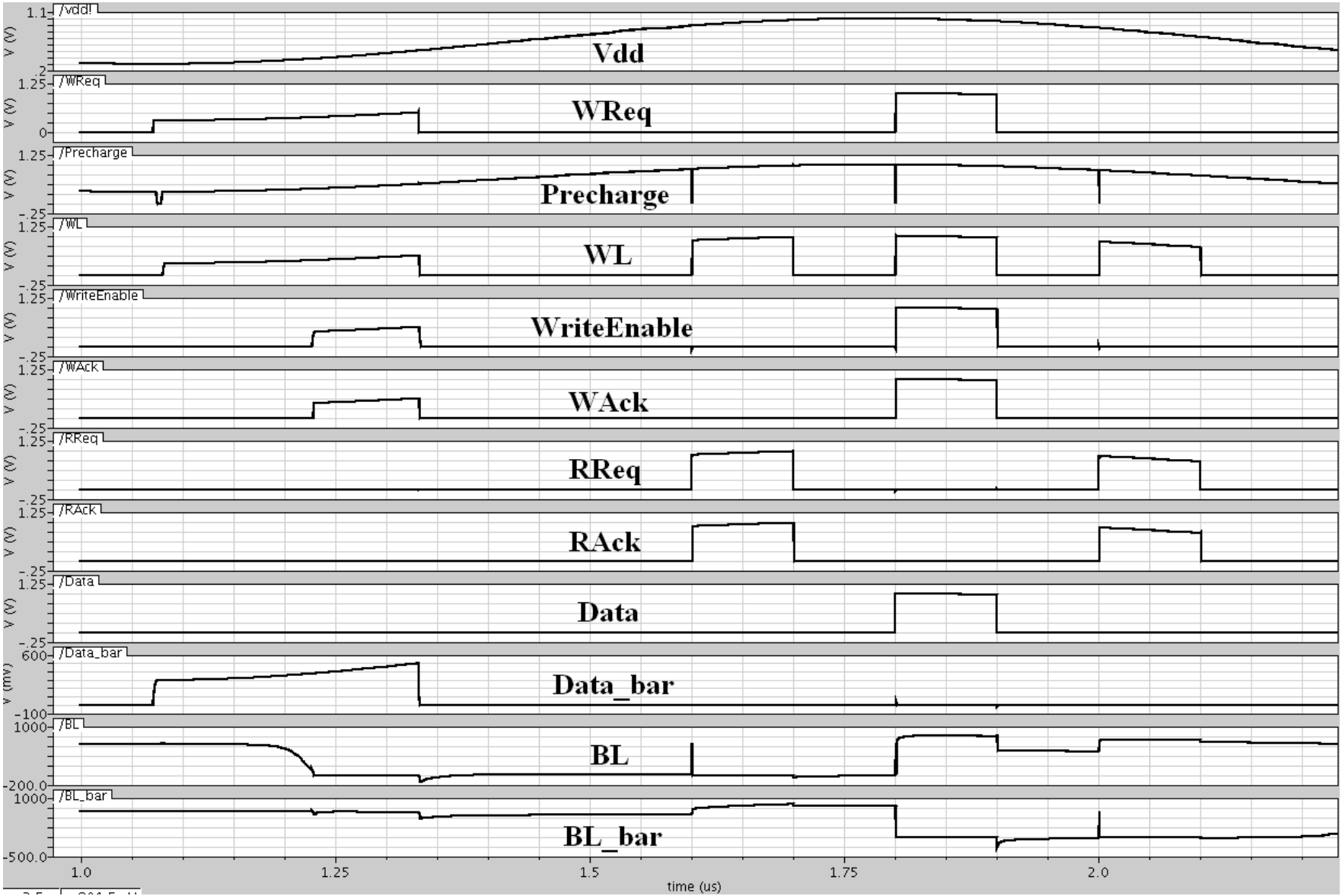
# Asynchronous SRAM controller



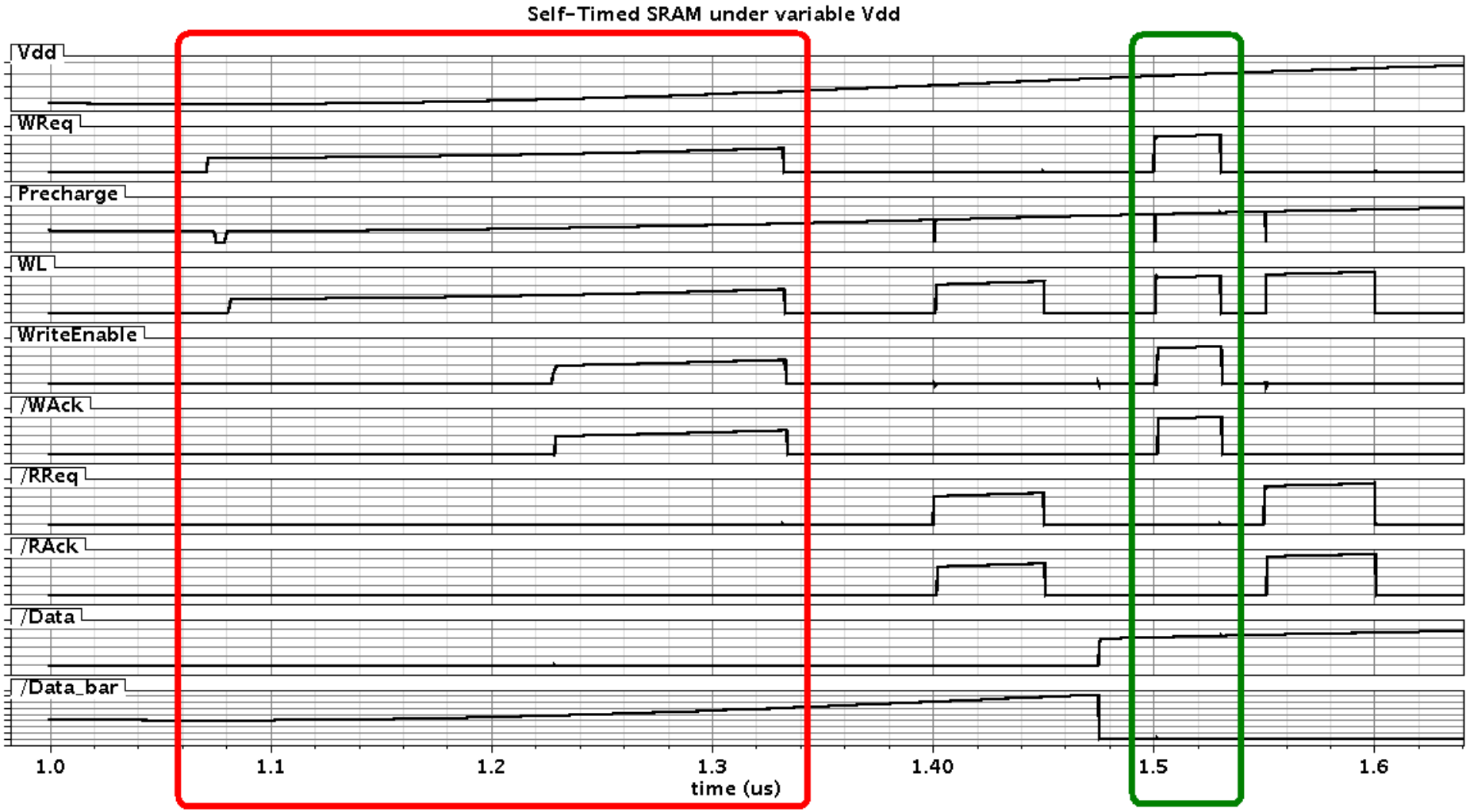

**Hand made**, hence not guaranteed to be speed-independent

We will design a provably correct implementation in Part II

# Tolerating variable voltage supply



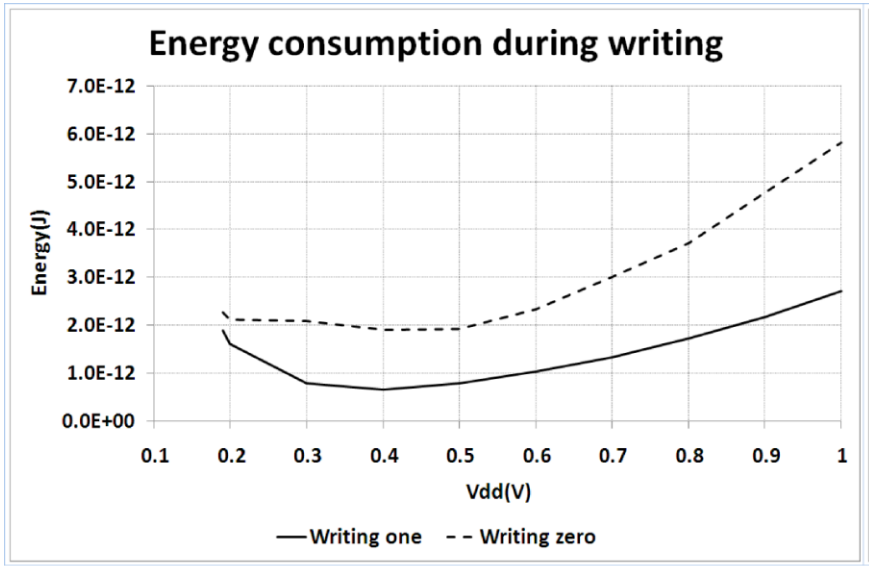
# Tolerating variable voltage supply

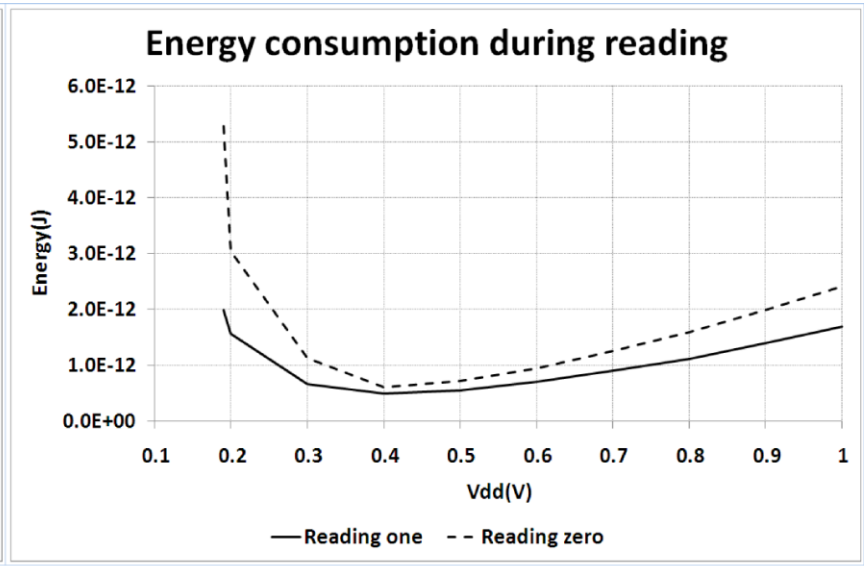


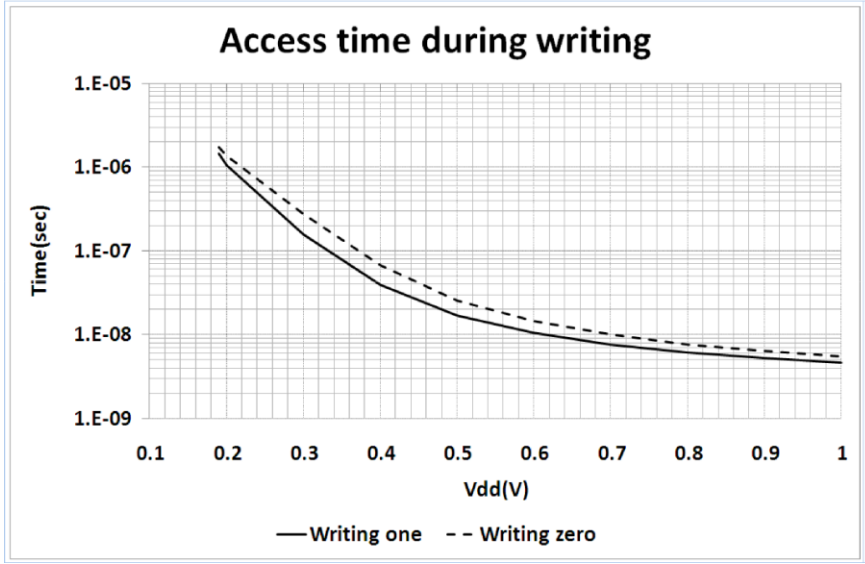

Low voltage, slow response

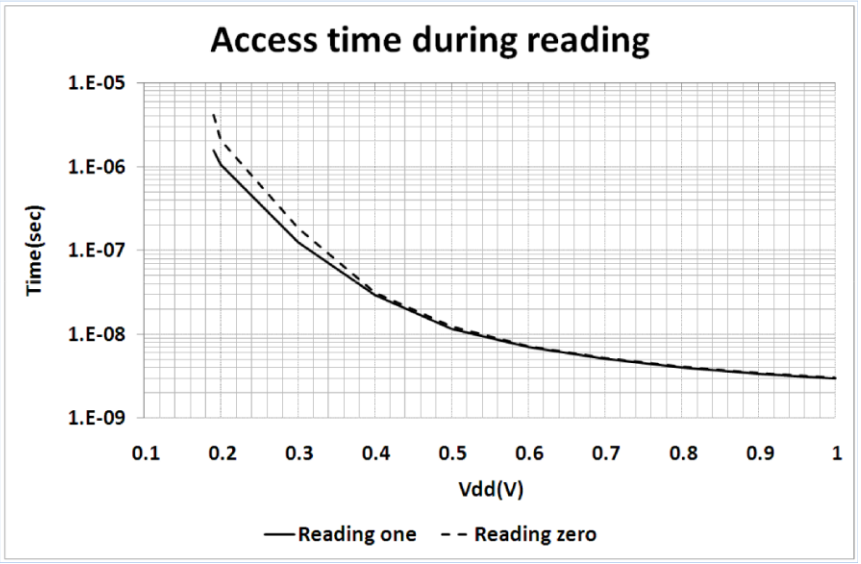
High voltage, fast response

# Trading energy for performance

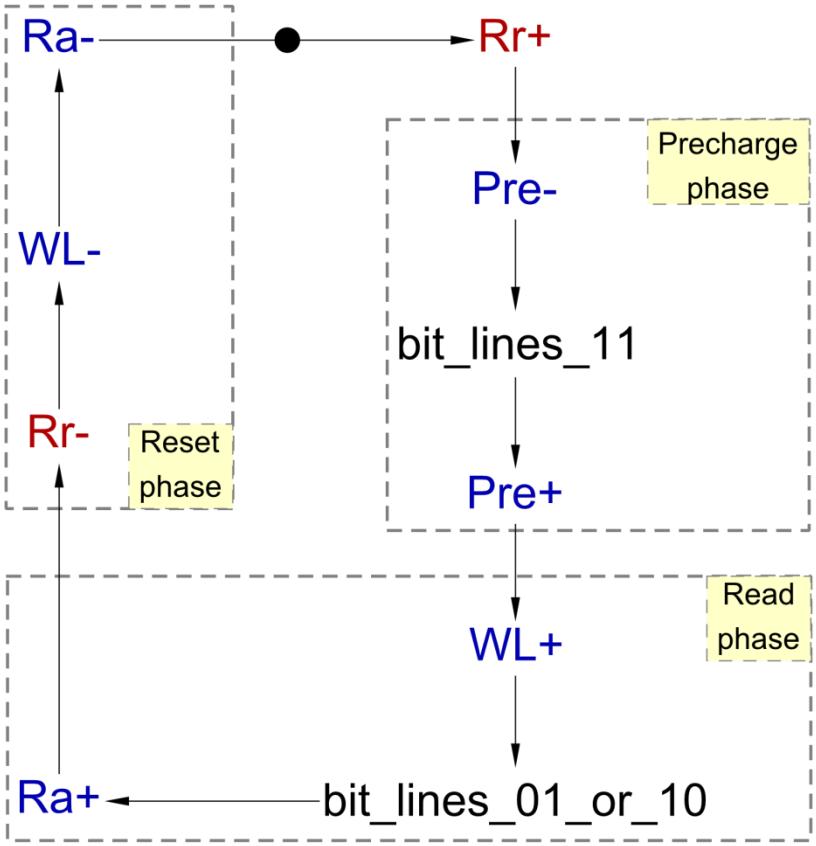


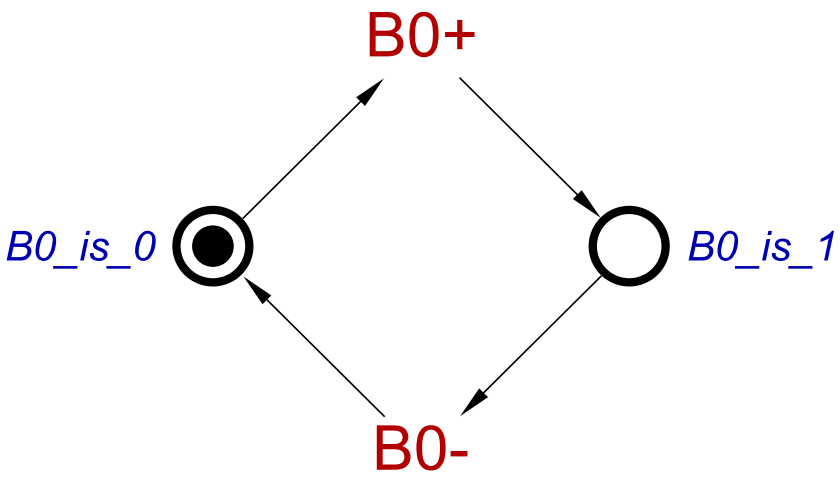



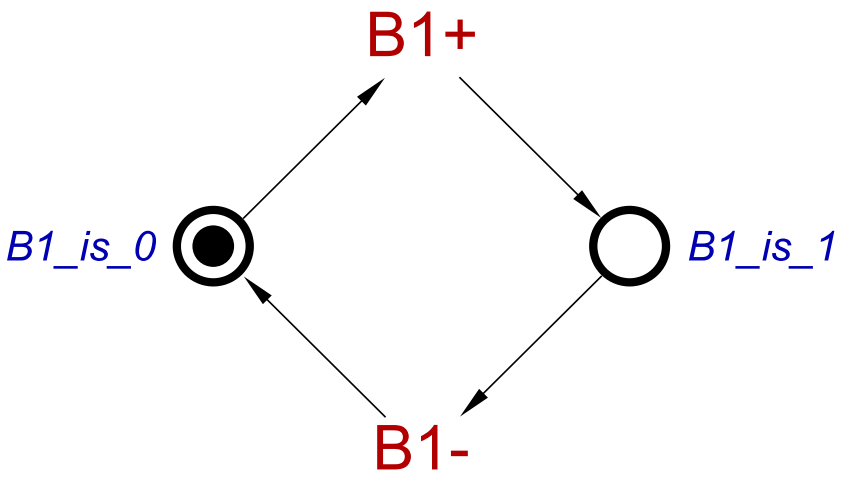



# Specification: read scenario

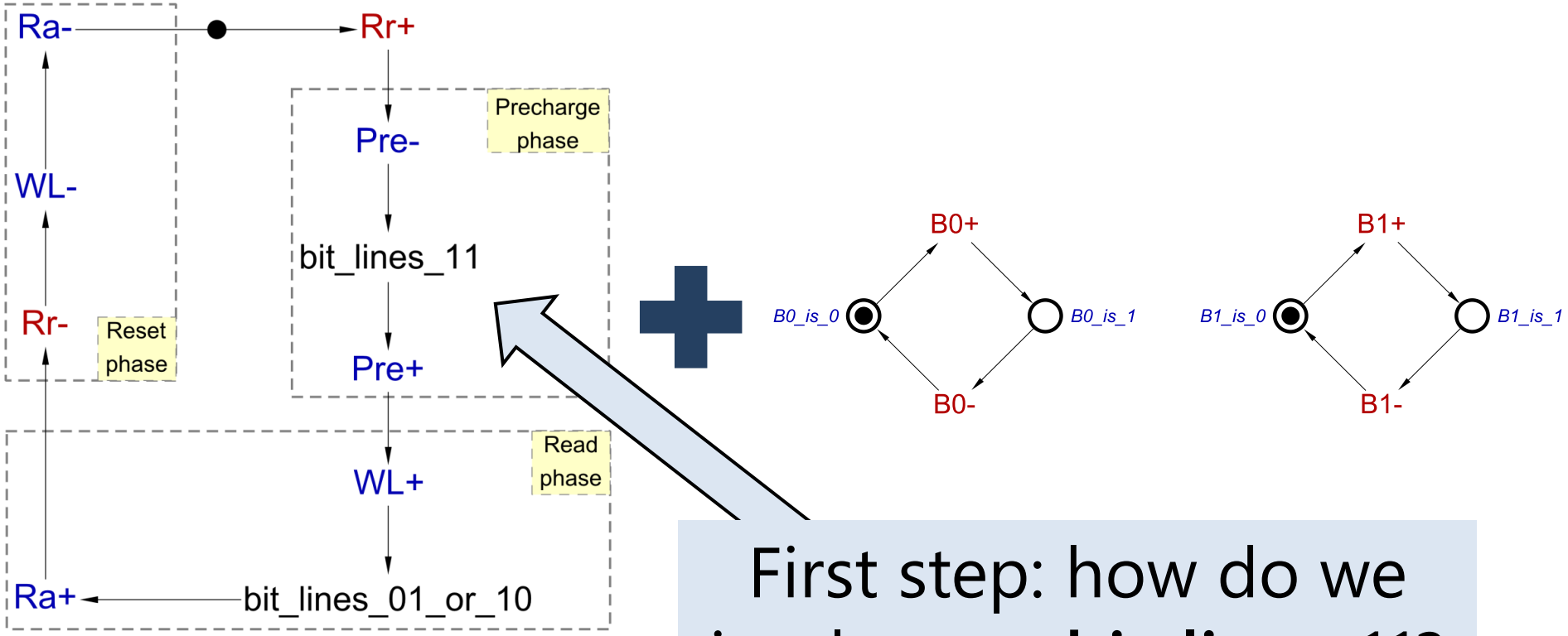


# Modelling bit line signals B0 and B1



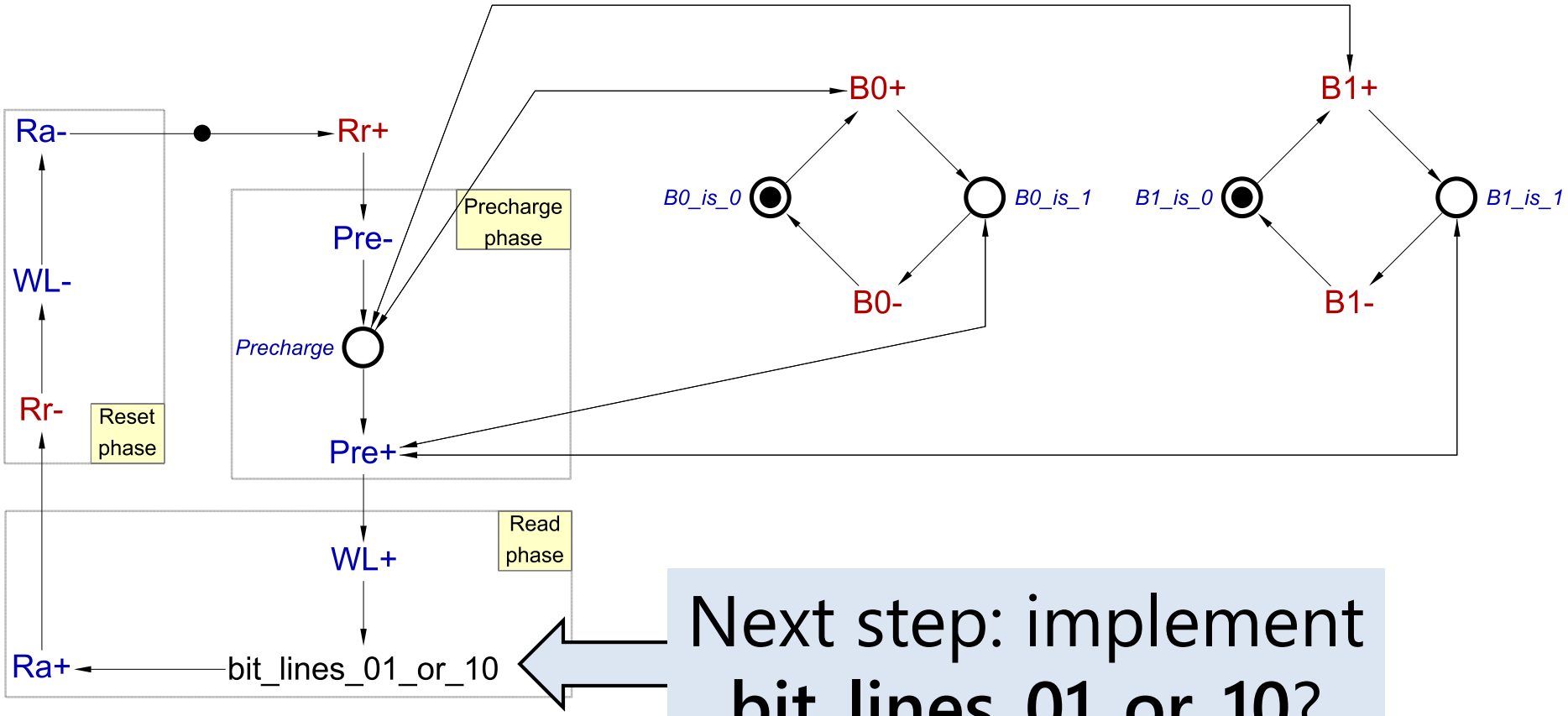



# Adding bit line signals to read scenario



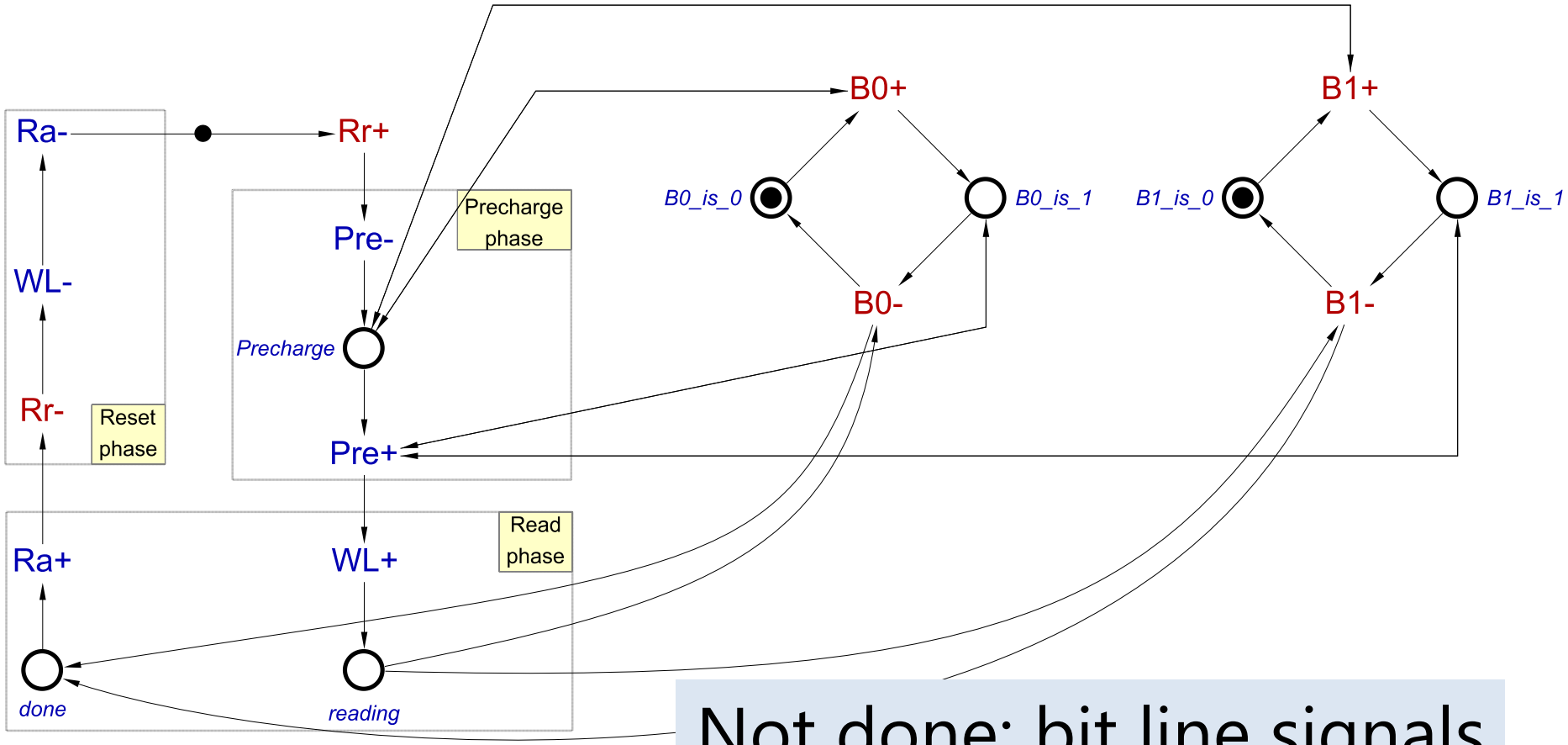

First step: how do we implement **bit_lines_11**?

# Adding bit line signals to read scenario




Next step: implement **bit_lines_01_or_10**?

# Adding bit line signals to read scenario




Not done: bit line signals may change when **Rr=0**

# Summary

Memory is inherently asynchronous

- Read & write completion can be reliably detected
- Conventional synchronous ‘handcuffs’ (matching delay lines) are clumsy and costly

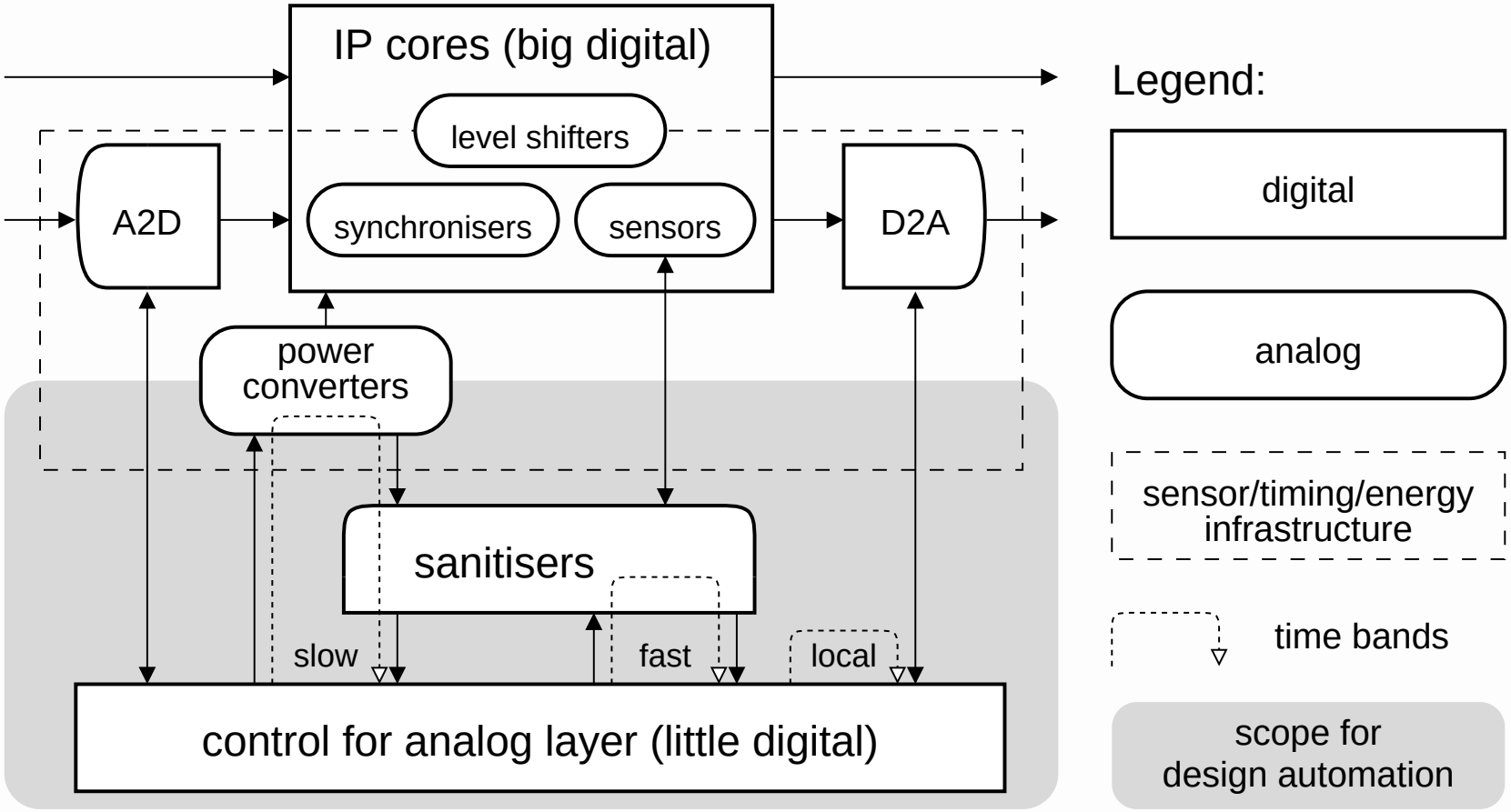
Typical ‘little digital’ control, fully supported by Workcraft design, synthesis and verification flow

Ongoing and future work – **you can contribute!**

- Integrate asynchronous SRAM into a real system
- Opportunity for new memory architectures

# Multiphase Buck Converter

# Motivation




- Analog and digital electronics are becoming more intertwined
- Analog domain becomes more complex and itself needs digital control

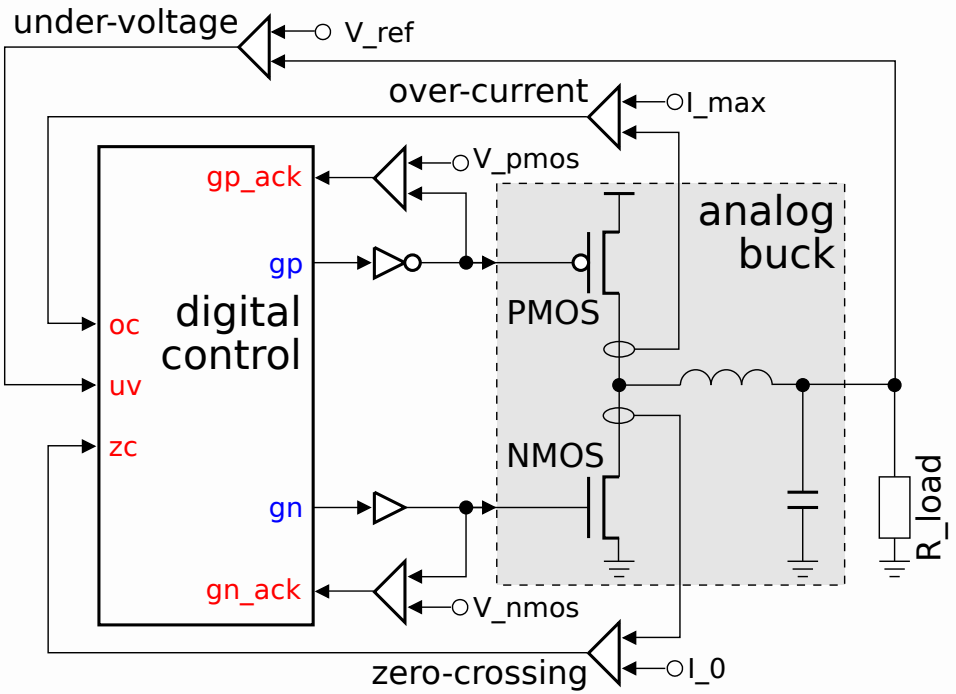
# Power electronics context

- Efficient implementation of power converters is paramount
  - Extending the battery life of mobile gadgets
  - Reducing the energy bill for PCs and data centres (5% and 3% of global electricity production respectively)
- Need for responsive and reliable control circuitry
  - Millions of control decisions per second for years
  - An incorrect decision may permanently damage the circuit
- Need for EDA (*little digital* vs *big digital* design flow)
  - RTL flow is optimised for synchronous data processing
  - *Ad hoc* asynchronous solutions are prone to errors and cannot be verified

# Synchronous vs asynchronous control

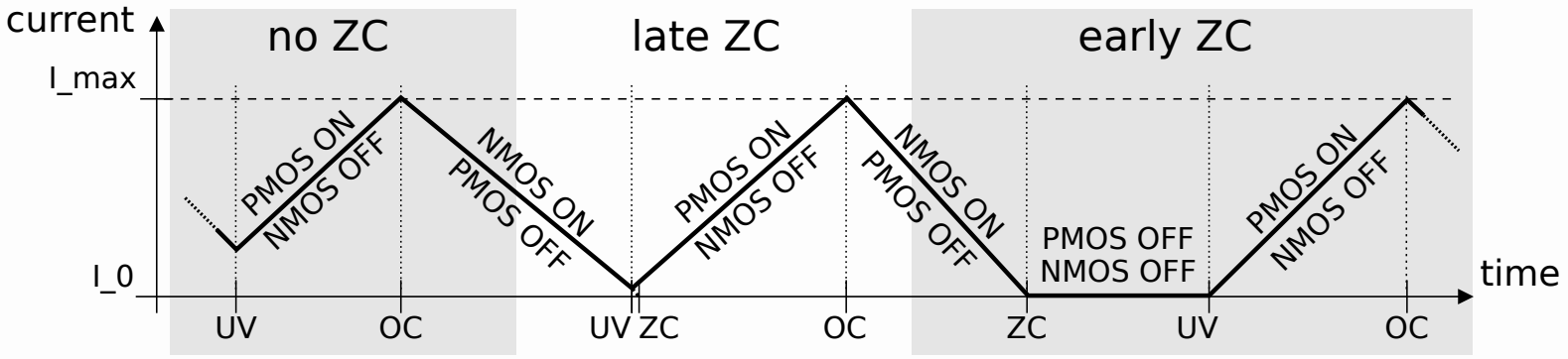
- Synchronous control
  - ☺ Conventional RTL design flow
  - ☹ Slow response (defined by the clock period)
  - ☹ Power consumed even when idle
  - ☹ Non-negligible probability of a synchronisation failure
- Asynchronous control
  - ☺ Prompt response (delay of few gates)
  - ☺ No dynamic power consumption when inactive
  - ☹ Non-conventional methodology and tool support

# Basic buck converter



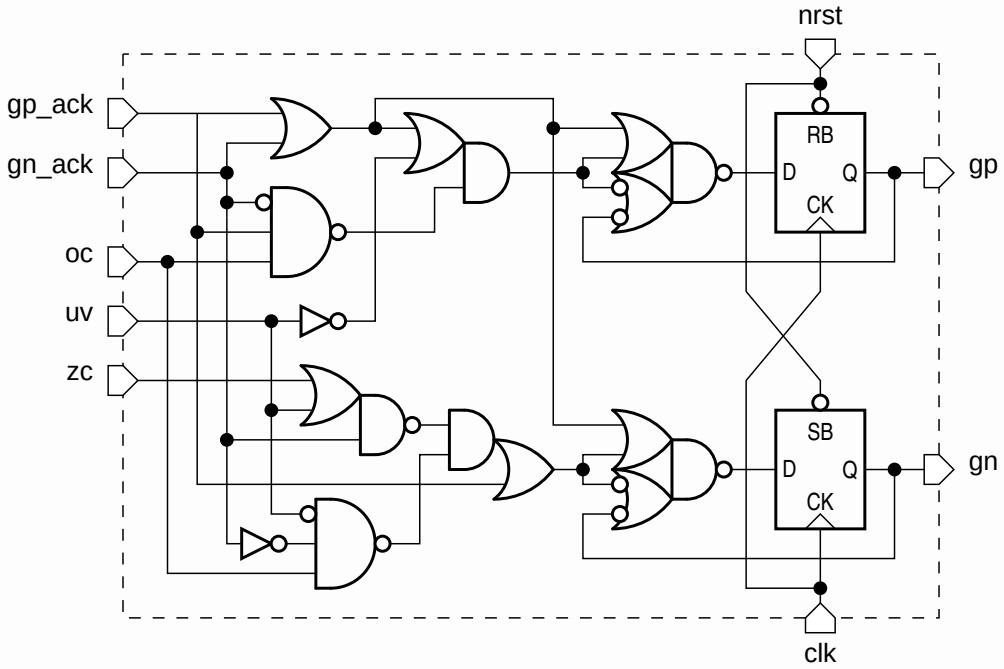

## Operating modes

- Under-voltage (UV)
- Over-current (OC)
- Zero-crossing (ZC)



# Synchronous design

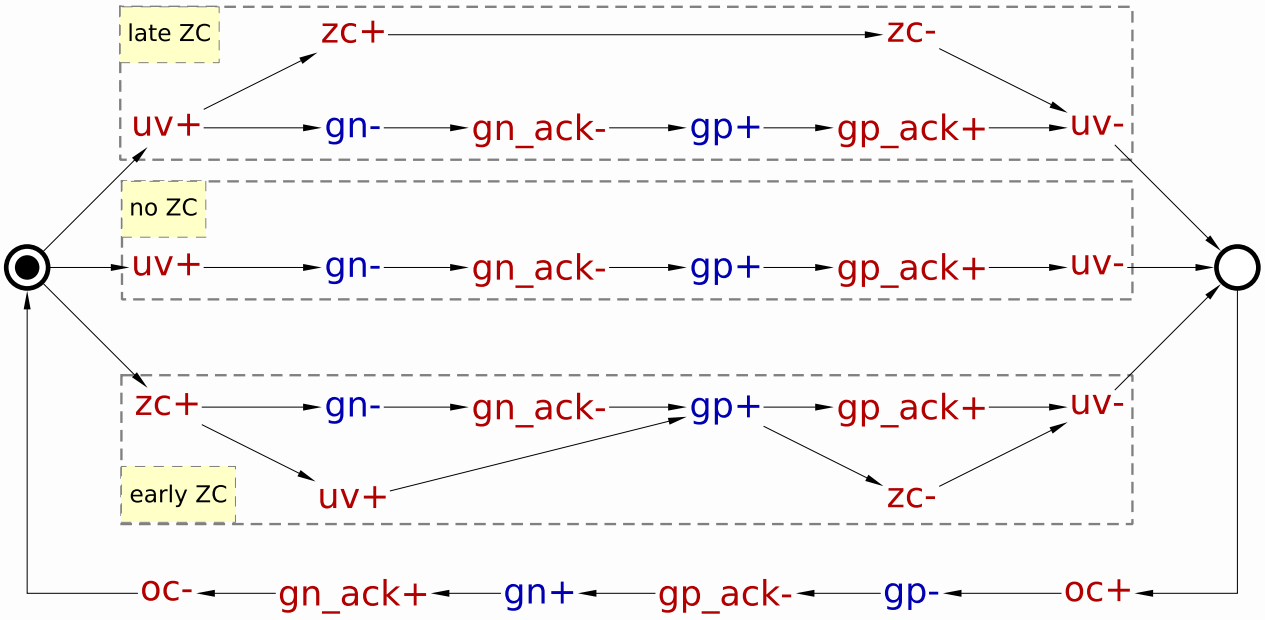
```
module control (clk, nrst, oc, uv, zc, gp_ack, gn_ack, gp, gn);
  input clk, nrst, uv, oc, zc, gp_ack, gn_ack;
  output reg gp, gn;
  always @(posedge clk or negedge nrst) begin
    if (nrst == 0) begin
      gp <= 0; gn <= 1;
    end else case ({gp_ack, gn_ack})
      2'b00: if (uv == 1) gp <= 1;
                else if (oc == 1) gn <= 1;
      2'b10: if (oc == 1) gp <= 0;
      2'b01: if (uv == 1 || zc == 1) gn <= 0;
    endcase
  end
endmodule
```



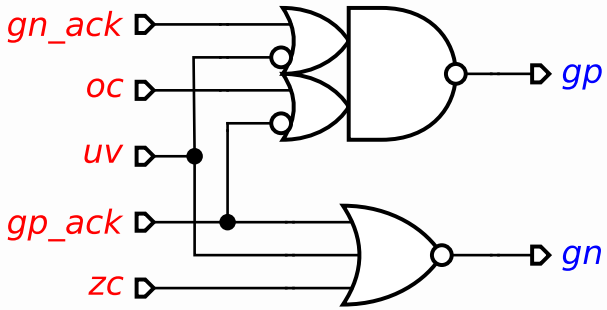

- If clock is slow, the control is unresponsive to the buck changes
- If clock is fast, it burns energy when the buck is inactive

# Asynchronous design

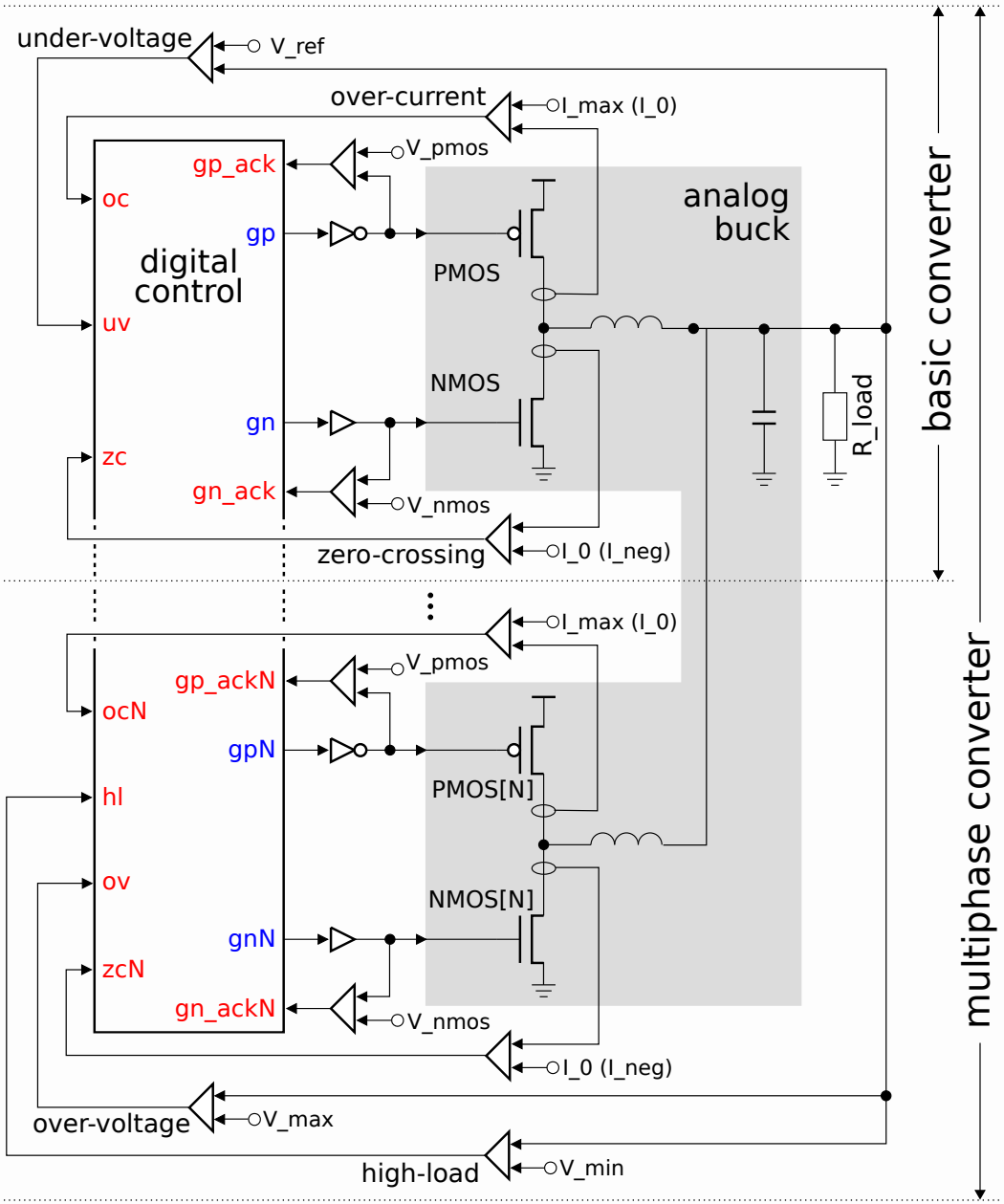
- STG specification




- Speed-independent implementation

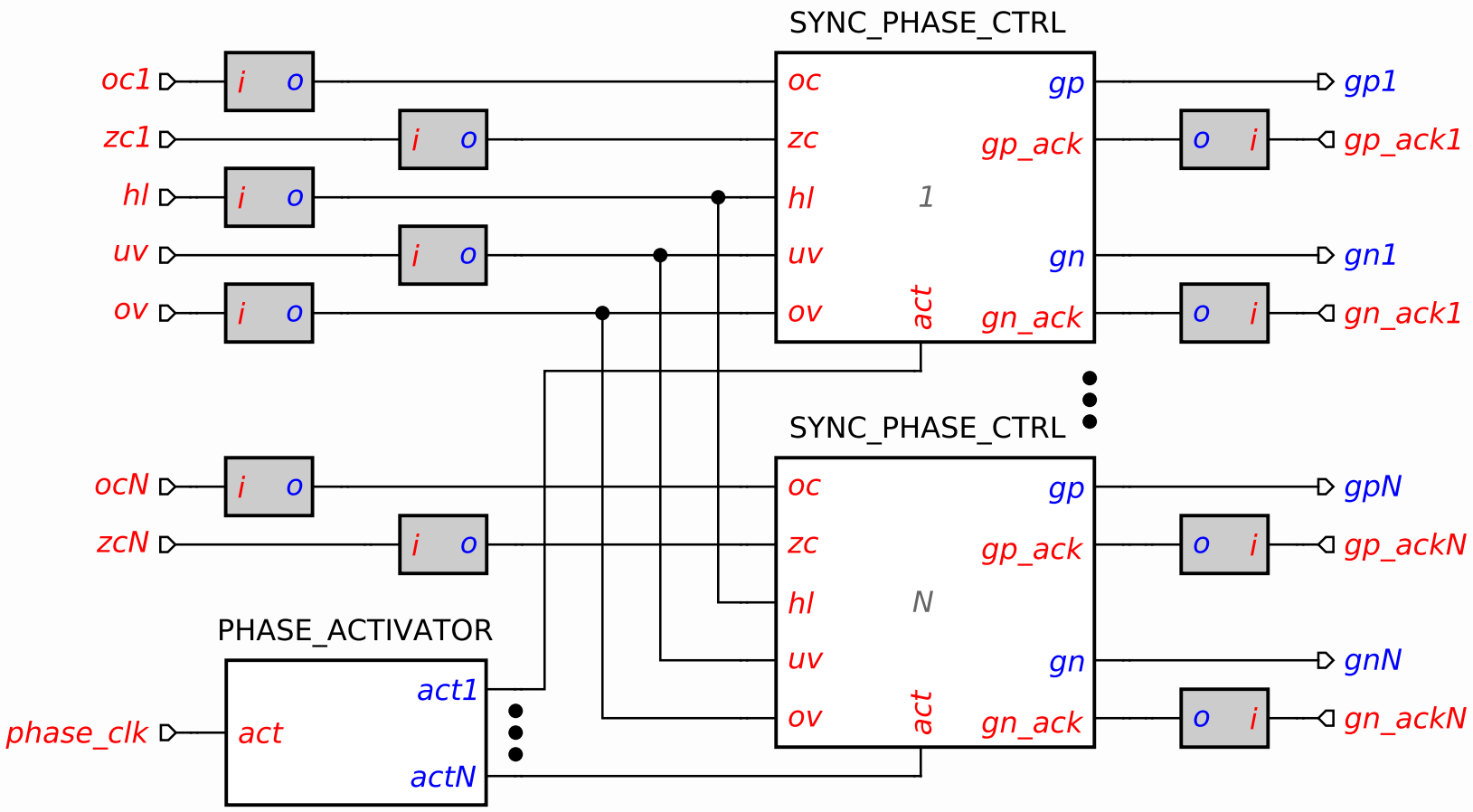


# Multiphase buck converter



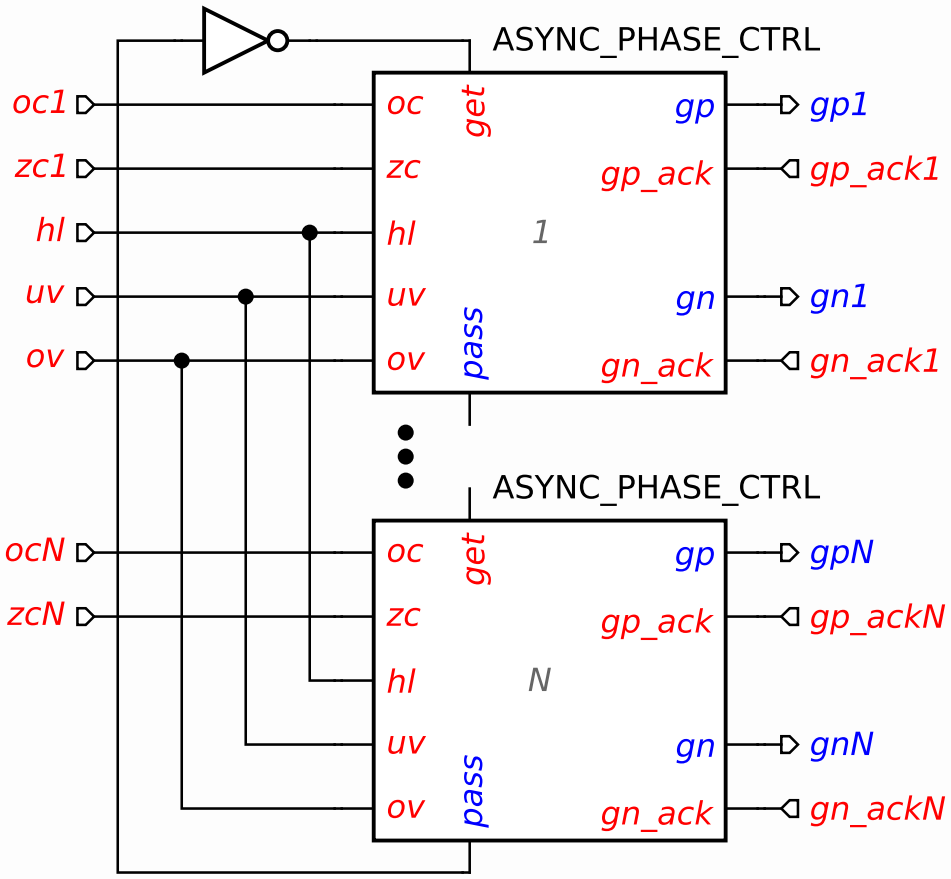

- Activation of phases
  - Sequential
  - May overlap
- More operating modes
  - High-load (HL)
  - Over-voltage (OV)
- Transistor min ON times
  - PMIN for PMOS
  - NMIN for NMOS
  - PMIN+PEXT for PMOS at first cycle

# Synchronous design



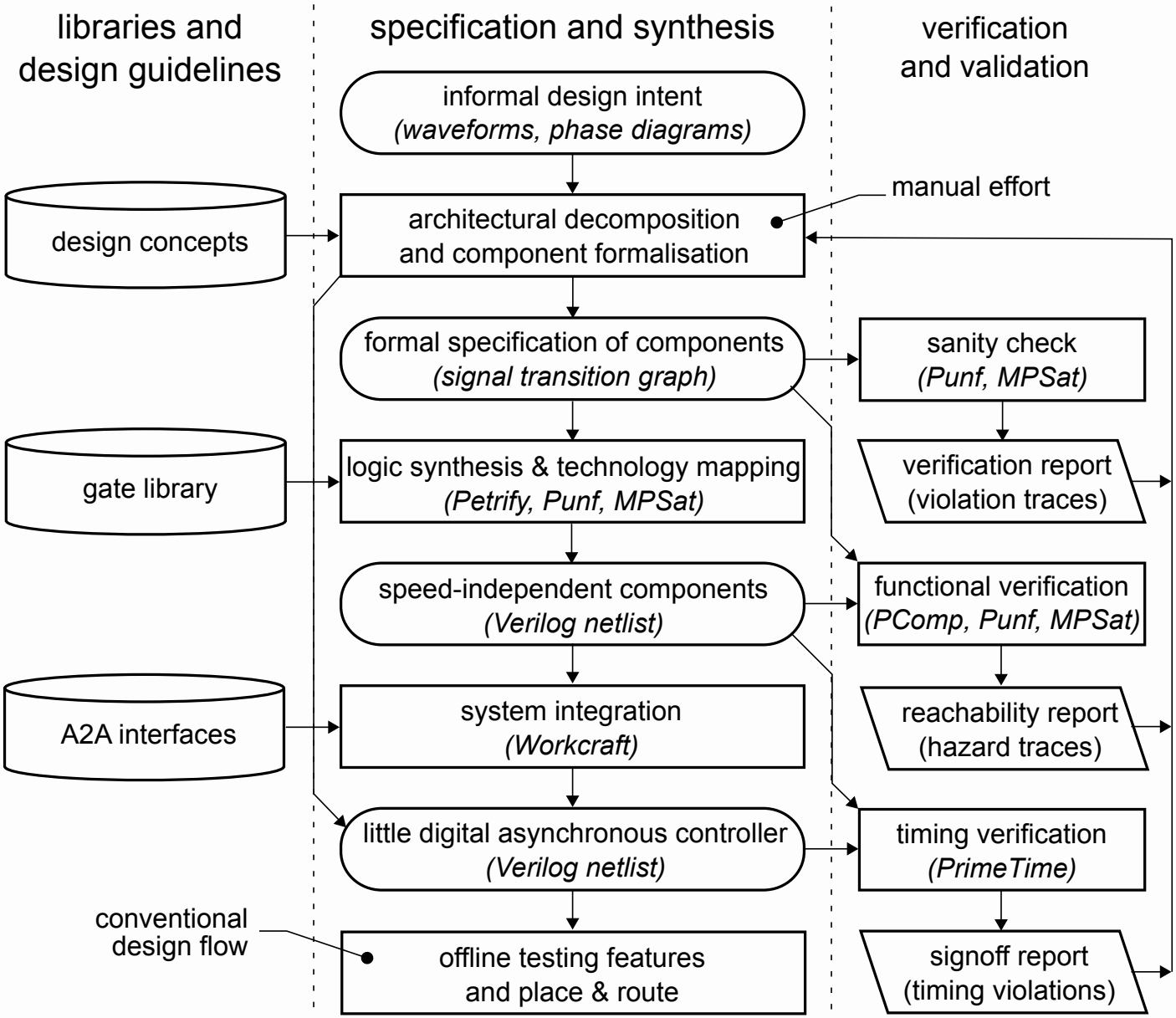

- Two clocks: phase activation (slow) and sampling (fast)
- Conventional RTL design flow for phase control
- Need for multiple synchronizers (grey boxes)

# Asynchronous design




- Token ring architecture, no need for phase activation clock
- No need for synchronisers
- A4A design flow for phase control

# A4A design flow

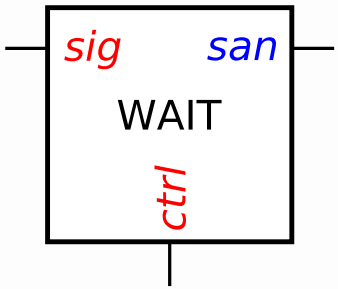


# A2A components

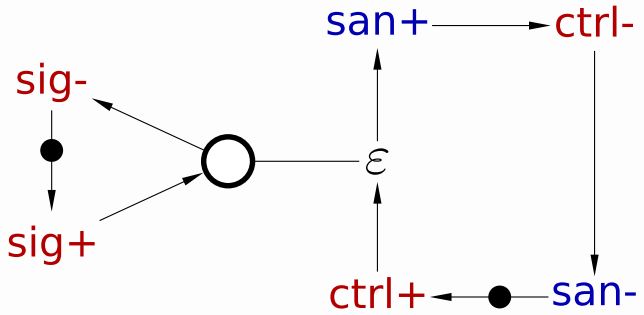
- Interface analog world of *dirty* signals
- Provide hazard-free *sanitised* digital signals
- Basic A2A components
  - **WAIT** / **WAIT0** – wait for analog input to become high / low and latch it until explicit release signal
  - **RWAIT** / **RWAIT0** – modification of WAIT / WAIT0 with a possibility to persistently cancel the waiting request
  - **WAIT01** / **WAIT10** – wait for a rising / falling edge
- Advanced A2A components
  - **WAIT2** – combination of WAIT and WAIT0 to wait for high and low input values, one after the other.
  - **WAITX** – arbitrate between two non-persistent analog inputs
  - **WAITX2** – behaves as WAITX in the rising phase and as WAIT0 in the falling phase

# WAIT element

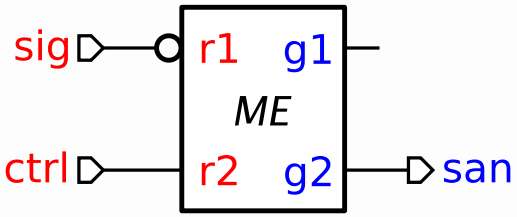
- Interface



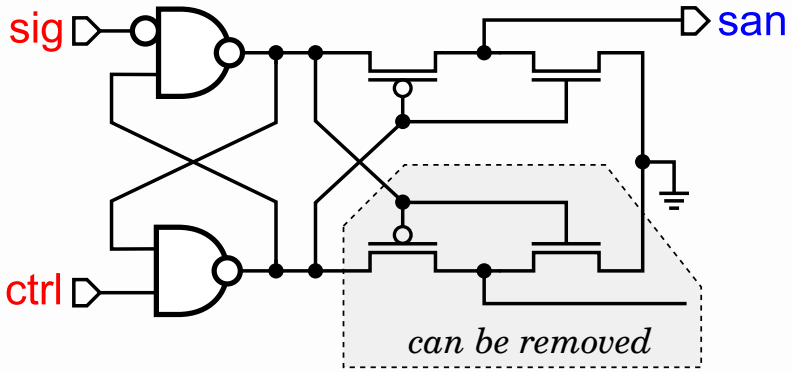

- STG specification



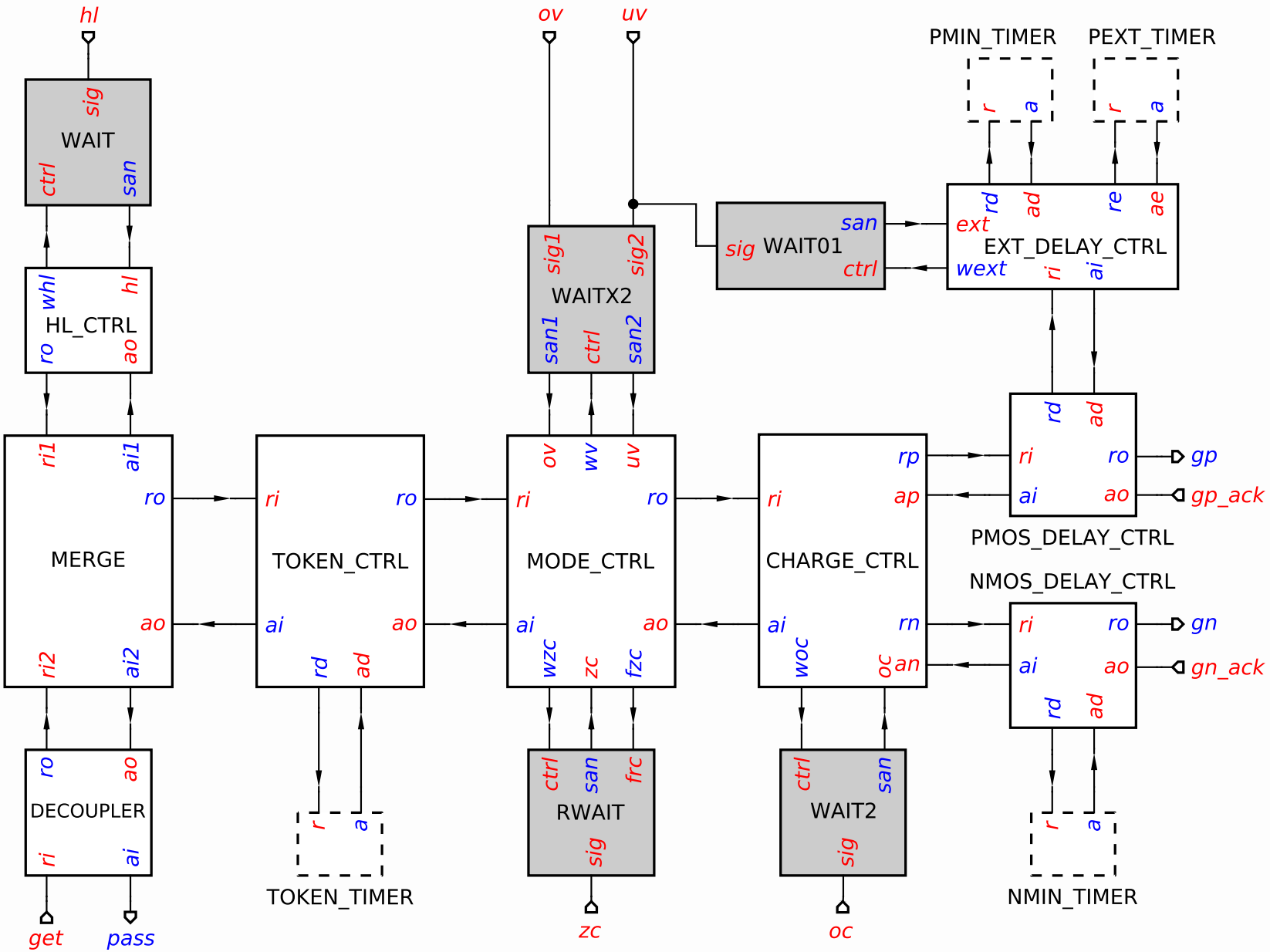

- ME-based solution




- Gate-level implementation

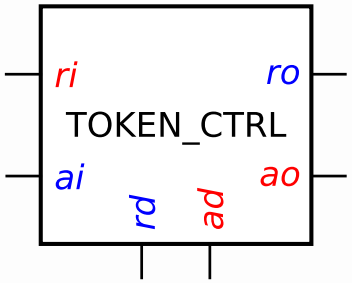


# Asynchronous phase control

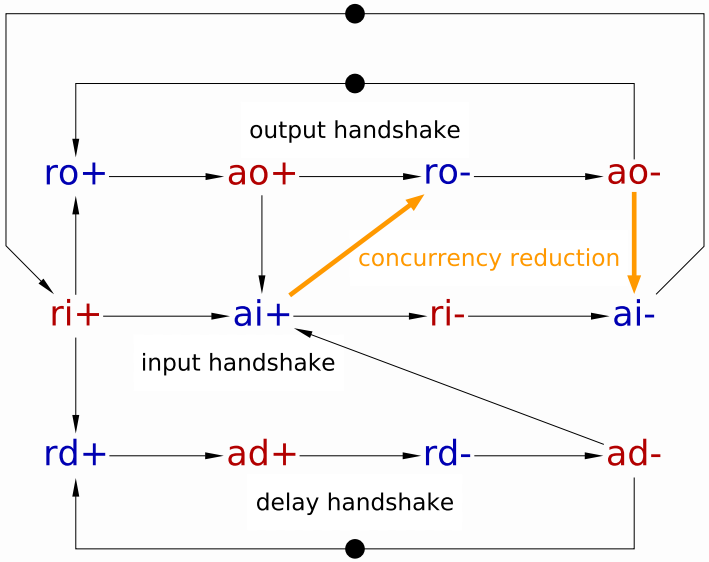


# Design of asynchronous components

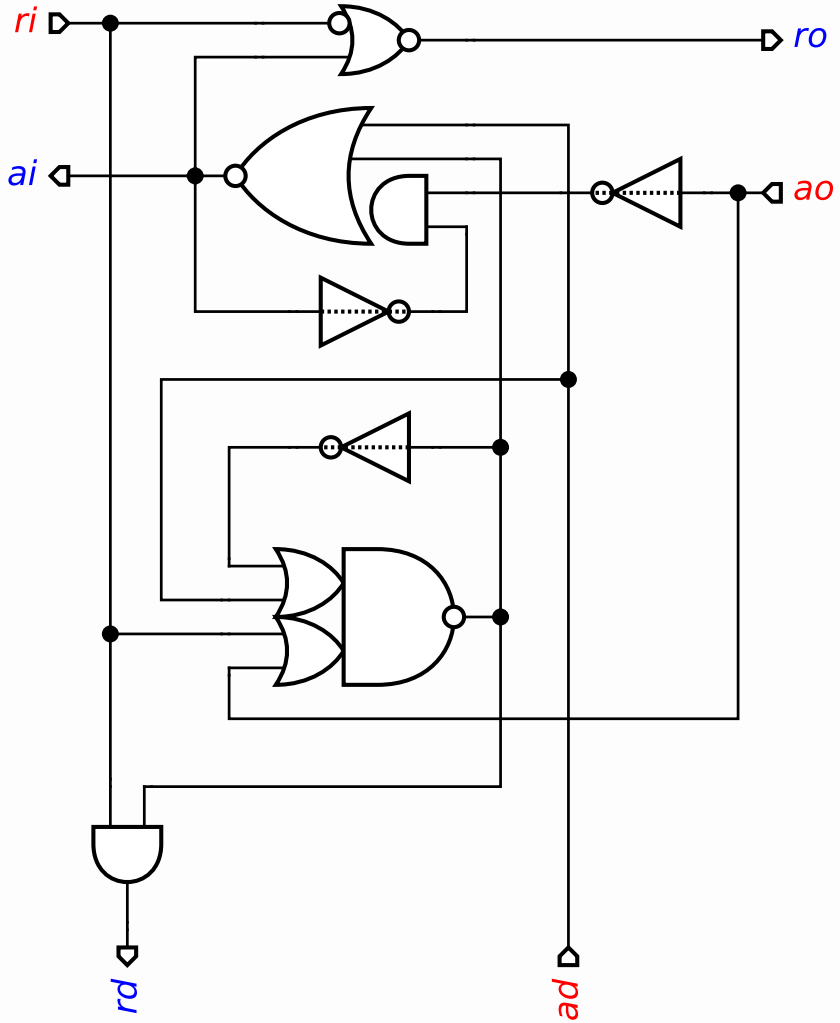
- Token control




- STG specification



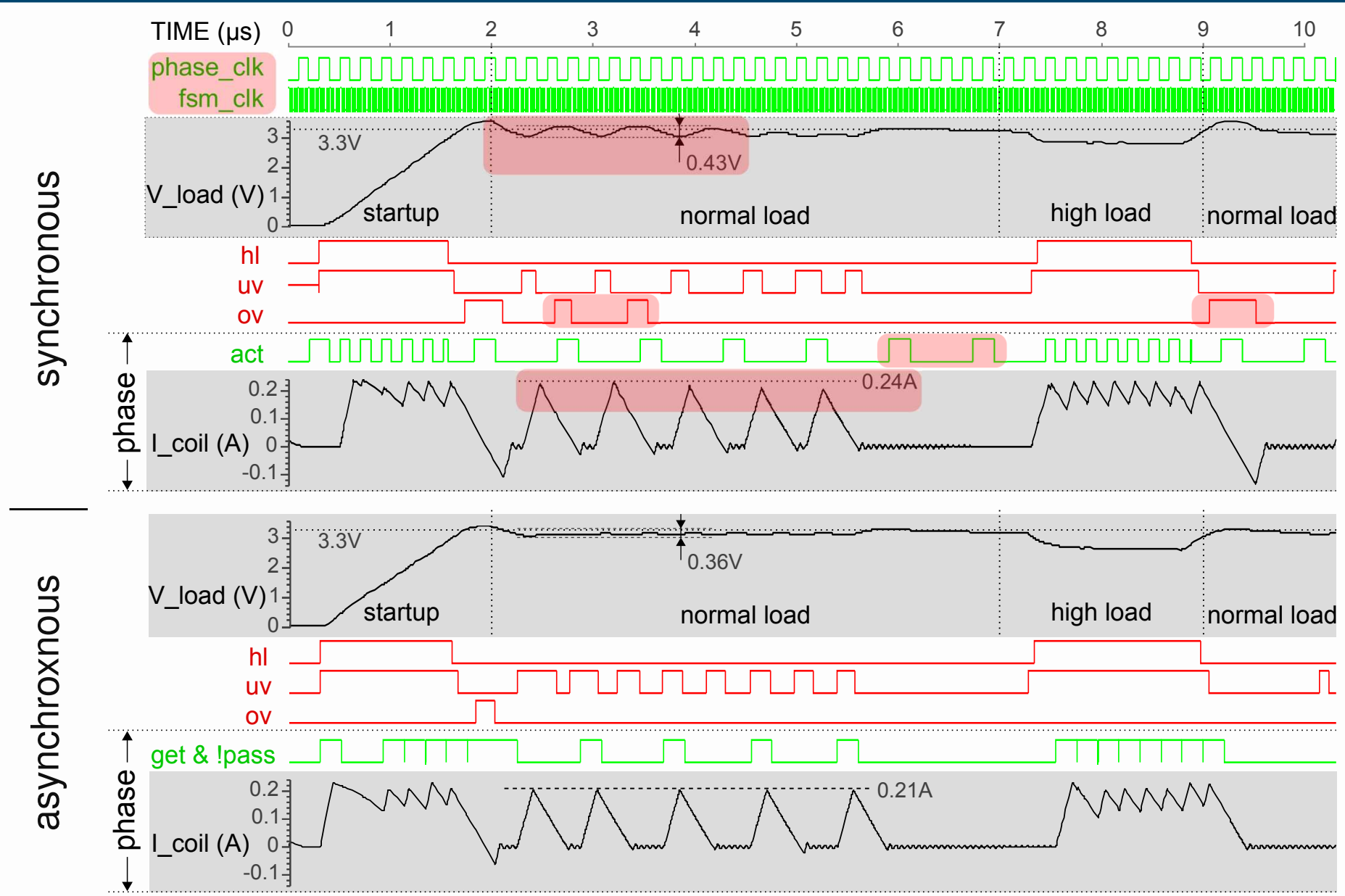

- Speed-independent implementation



# Simulation setup

- Verilog-A model of the 4-phase buck
- Control implemented in TSMC 90nm
- AMS simulation in CADENCE NC-VERILOG
- Synchronous design
  - Phase activation clock – 5MHz
  - Clocked FSM-based control – 100MHz, 333MHz, 666MHz, 1GHz
  - Sampling and synchronisation
- Asynchronous design
  - Phase activation – token ring with 200ns timer (= 5MHz)
  - Event-driven control (input-output mode)
  - Waiting rather than sampling (A2A components)

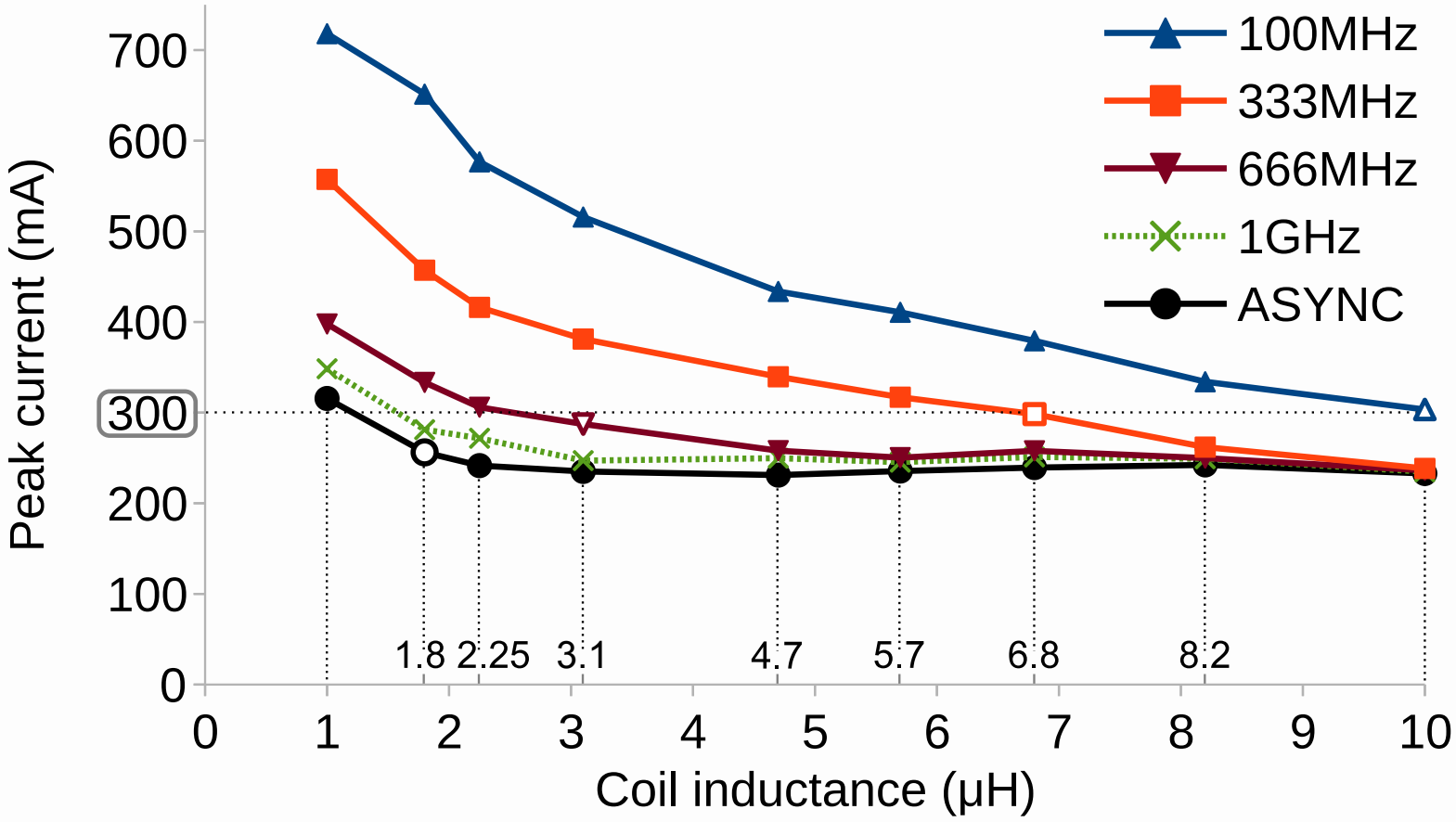
# Simulation waveforms



# Reaction time

| Controller | HL (ns) | UV (ns) | OV (ns) | OC (ns) | ZC (ns) |
|---|---|---|---|---|---|
| 100MHz | 25.00 | 25.00 | 25.00 | 25.00 | 25.00 |
| 333MHz | 7.50 | 7.50 | 7.50 | 7.50 | 7.50 |
| 666MHz | 3.75 | 3.75 | 3.75 | 3.75 | 3.75 |
| 1GHz | 2.50 | 2.50 | 2.50 | 2.50 | 2.50 |
| ASYNC | 1.87 | 1.02 | 1.18 | 0.75 | 0.31 |
| Improvement over 333MHz | 4x | 7x | 6x | 10x | 24x |

# Peak current



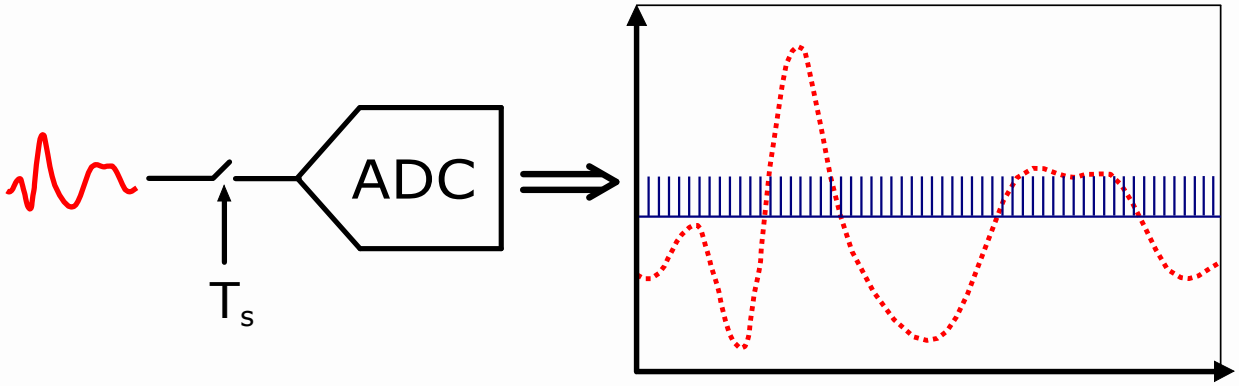
# Conclusions

- Design flow is automated to large extent
  - Library of A2A components
  - Automatic logic synthesis
  - Formal verification at the STG and circuit levels
- Benefits of asynchronous multiphase buck controller
  - Reliable, no synchronisation failures
  - Quick response time (few gate delays)
  - Reaction time can be traded off for smaller coils
  - Lower voltage ripple and peak current

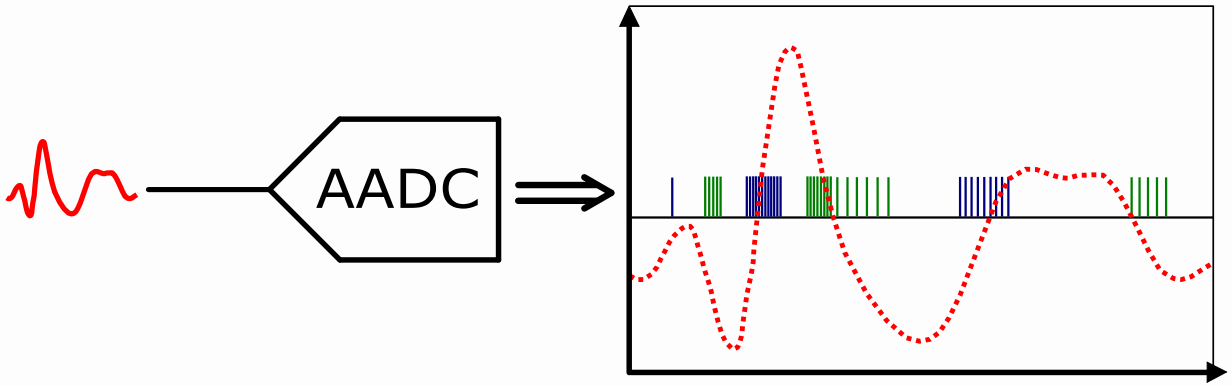
# Asynchronous ADC

# Sampling schemes

- Synchronous



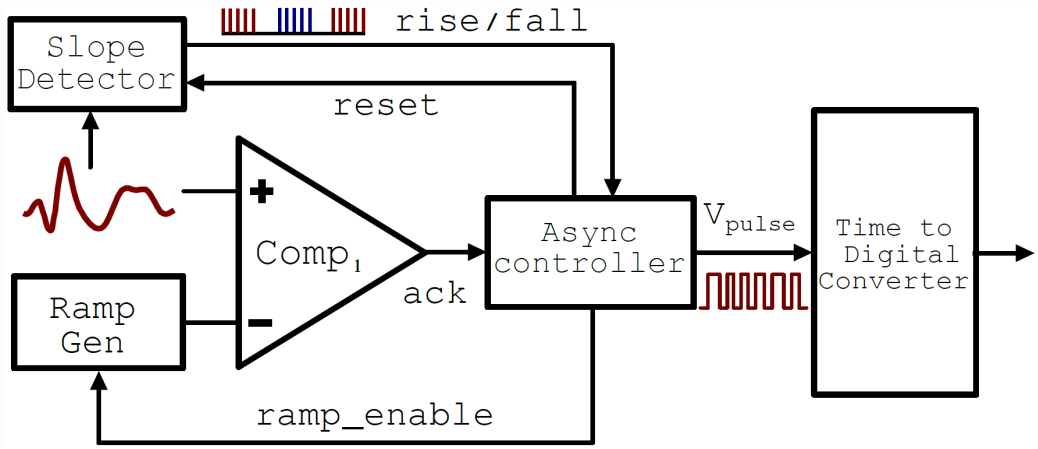

- Asynchronous

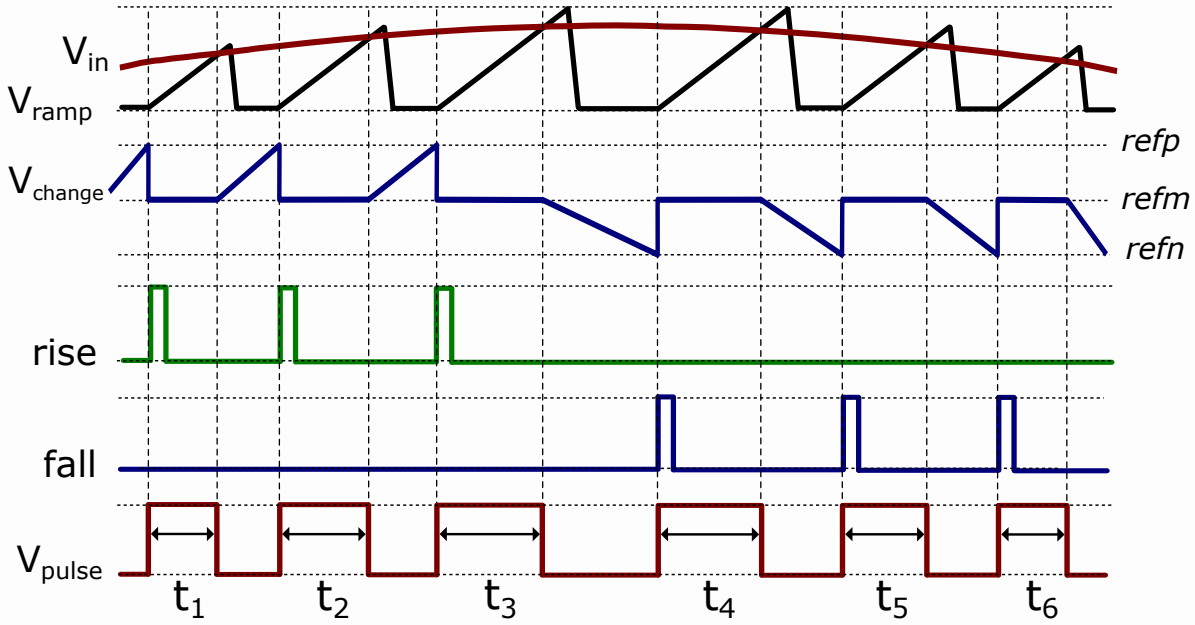



---

A. Ogweno, P. Degenaar, V. Khomenko and A. Yakovlev: *"A fixed window level crossing ADC with activity dependent power dissipation"*, accepted for NEWCAS-2016.

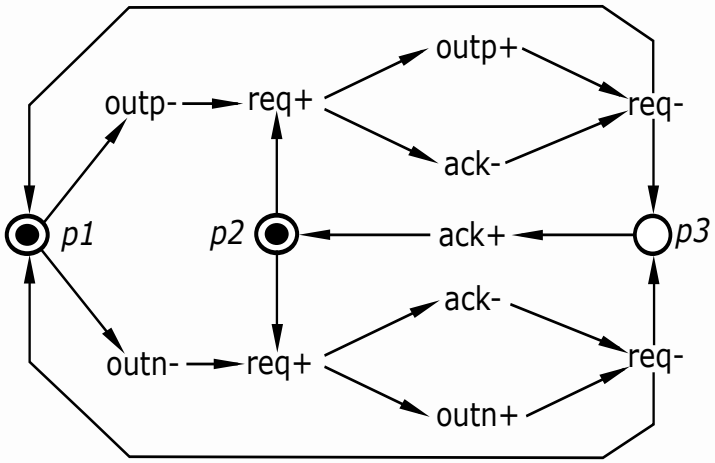
# ADC design


rise/fall
Slope Detector
reset
Comp1
Ramp Gen
ack
Async controller
Vpulse
Time to Digital Converter
ramp_enable

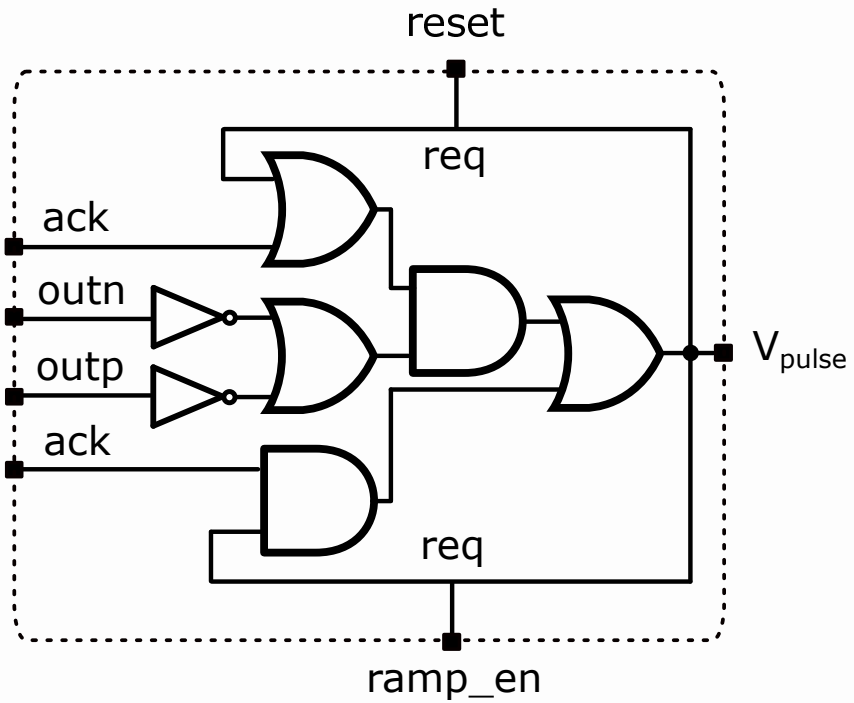

Vin
Vramp
Vchange
refp
refm
refn
rise
fall
Vpulse
t1
t2
t3
t4
t5
t6

# Asynchronous controller

- STG specification




- Speed-independent implementation




http://workcraft.org/

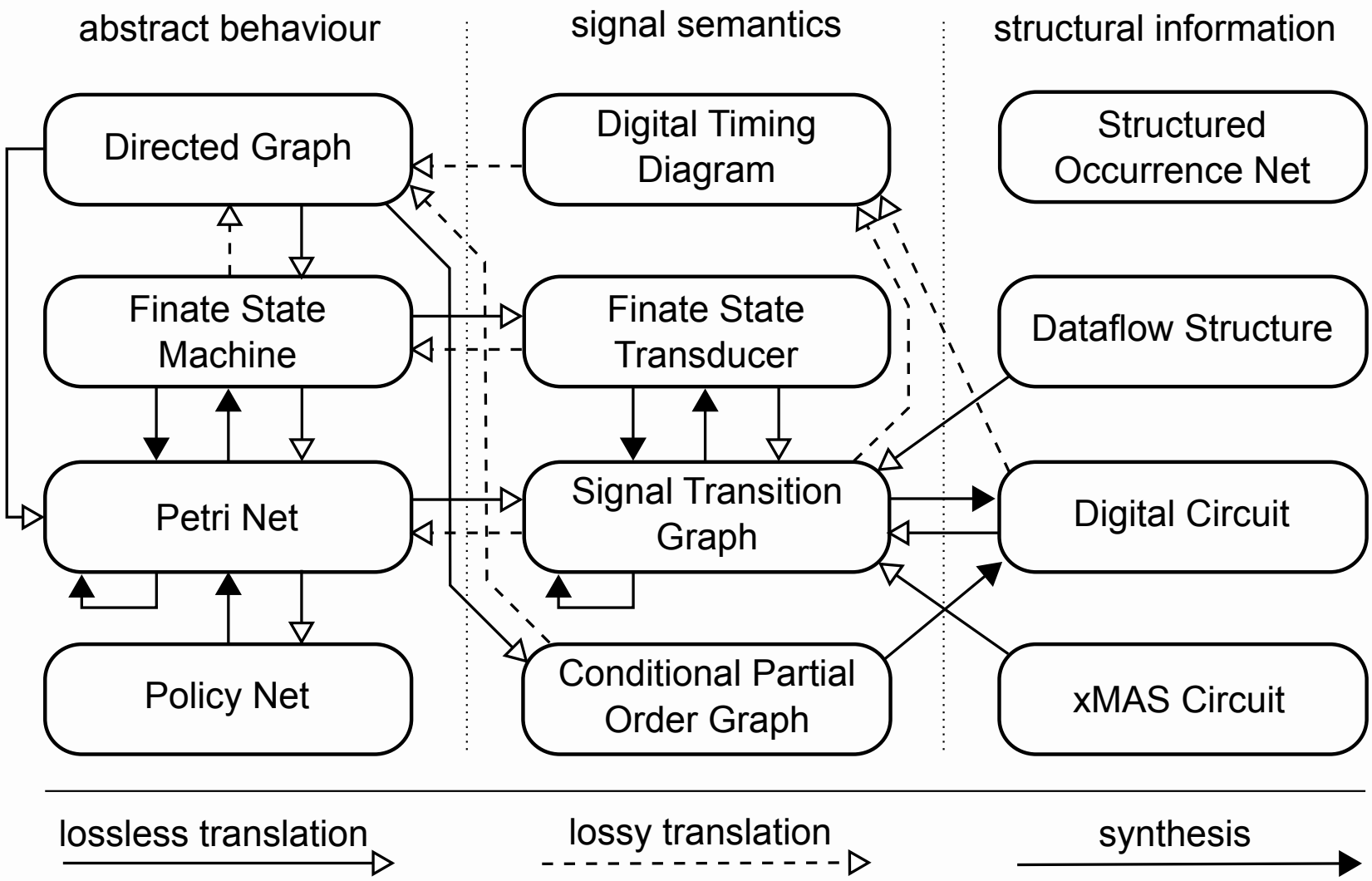
# What is WORKCRAFT?

- Framework for interpreted graph models
  - 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)
  - Command log for debugging and scripting

# Why to use WORKCRAFT?

- Availability
  - Open-source front-end and plugins
  - Permissive freeware licenses for back-end tools
  - Frequent releases (4-6 per year)
  - Specialised tutorials and online training materials
- Extendibility
  - Plugins for new formalisms
  - Import, export and converter plugins
  - Interface to back-end tools
- Usability
  - Elaborated GUI developed with much user feedback
- Portability
  - Distributions for Windows, Linux, and OS X

# Supported graph models

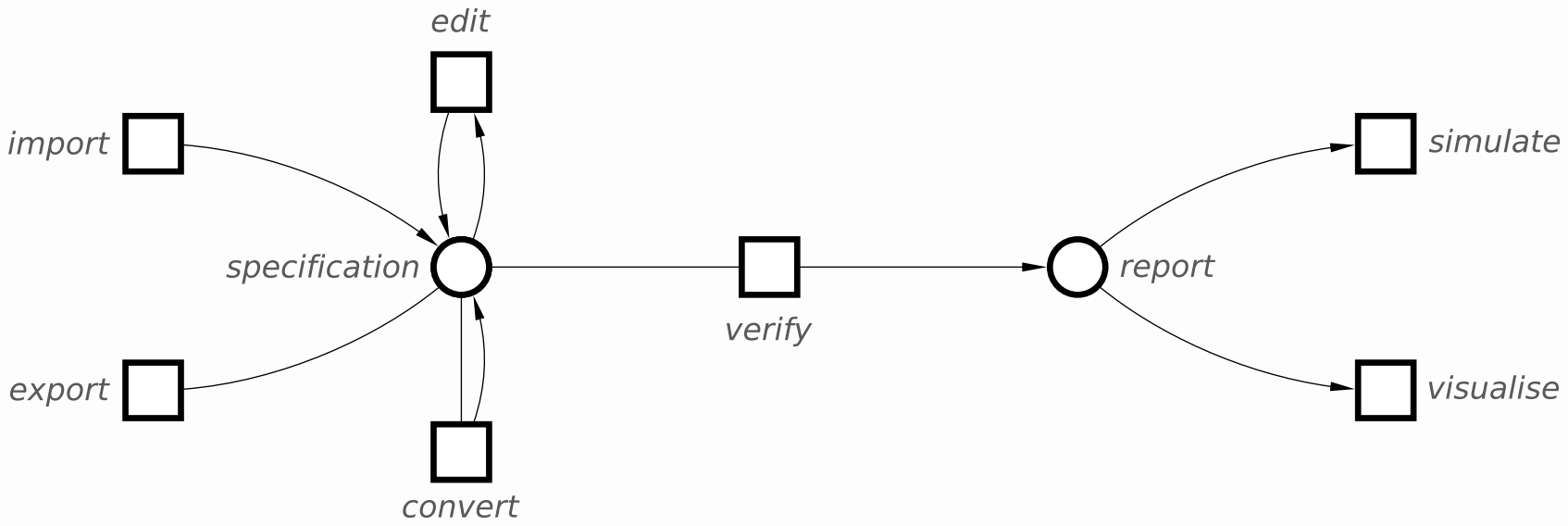


# Supported features

| Model | Supported features | | | |
|---|---|---|---|---|
| | Editing | Simulation | Verification | Synthesis |
| **abstract behaviour** | | | | |
| Directed Graph | Yes | Yes | Yes | n/a |
| Finite State Machine | Yes | Yes | Yes | Yes[1)] |
| Petri Net | Yes | Yes | Yes | Yes[2)] |
| Policy Net | Yes | Yes | Yes | n/a |
| **signal semantics** | | | | |
| Digital Timing Diagram | Yes | No | n/a | n/a |
| Finite State Transducer | Yes | Yes | Yes | Yes[3)] |
| Signal Transition Graph | Yes | Yes | Yes | Yes[4)] |
| Conditional Partial Order Graph | Yes | Some | No | Yes |
| **structural information** | | | | |
| Structured Occurrence Net | Yes | Yes | Yes | n/a |
| Dataflow Structure | Yes | Yes | Yes | No |
| Digital Circuit | Yes | Yes | Yes | n/a |
| xMAS Circuit | Yes | Yes | Some | No |

1) synthesis into Petri Net
2) re-synthesis into simpler Petri Net
3) synthesis into Signal Transition Graph
4) synthesis into Digital Circuit and re-synthesis into simpler Petri Net

# Design flow




- Import: ASTG, Verilog
- Export: ASTG, Verilog, SVG/Dot/PDF/EPS
- Convert: synthesis or translation
- Verify: reachability analysis (REACH predicates, SVA-like invariants)
- Visualise: CSC conflict cores, circuit initialisation, bottleneck

# Design flow: Asynchronous circuits

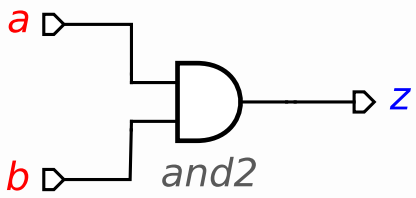
1. Specification of desired circuit behaviour with an STG model
2. Verification of the STG model
   (a) Standard implementability properties:
   consistency, deadlock freeness, output persistency
   (b) Design–specific custom properties
3. Resolution of complete state coding (CSC) conflicts
4. Circuit synthesis in one of the supported design styles
5. Manual tweaking and optimisation of the circuit
6. Verification of circuit against the initial specification
   (a) Synthesis tools are complicated and may have bugs
   (b) Manual editing is error-prone
7. Exporting the circuit as a Verilog netlist for conventional EDA backend

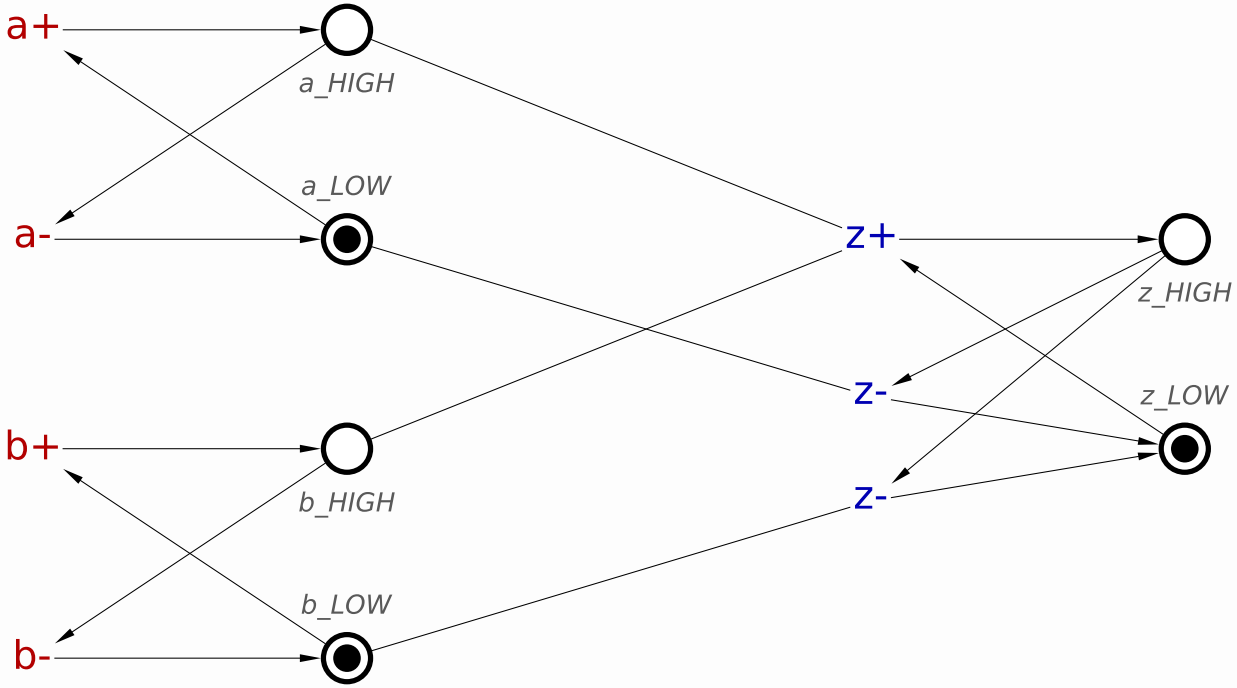
# What is hidden from the user?

Verification that the circuit conforms to its specification

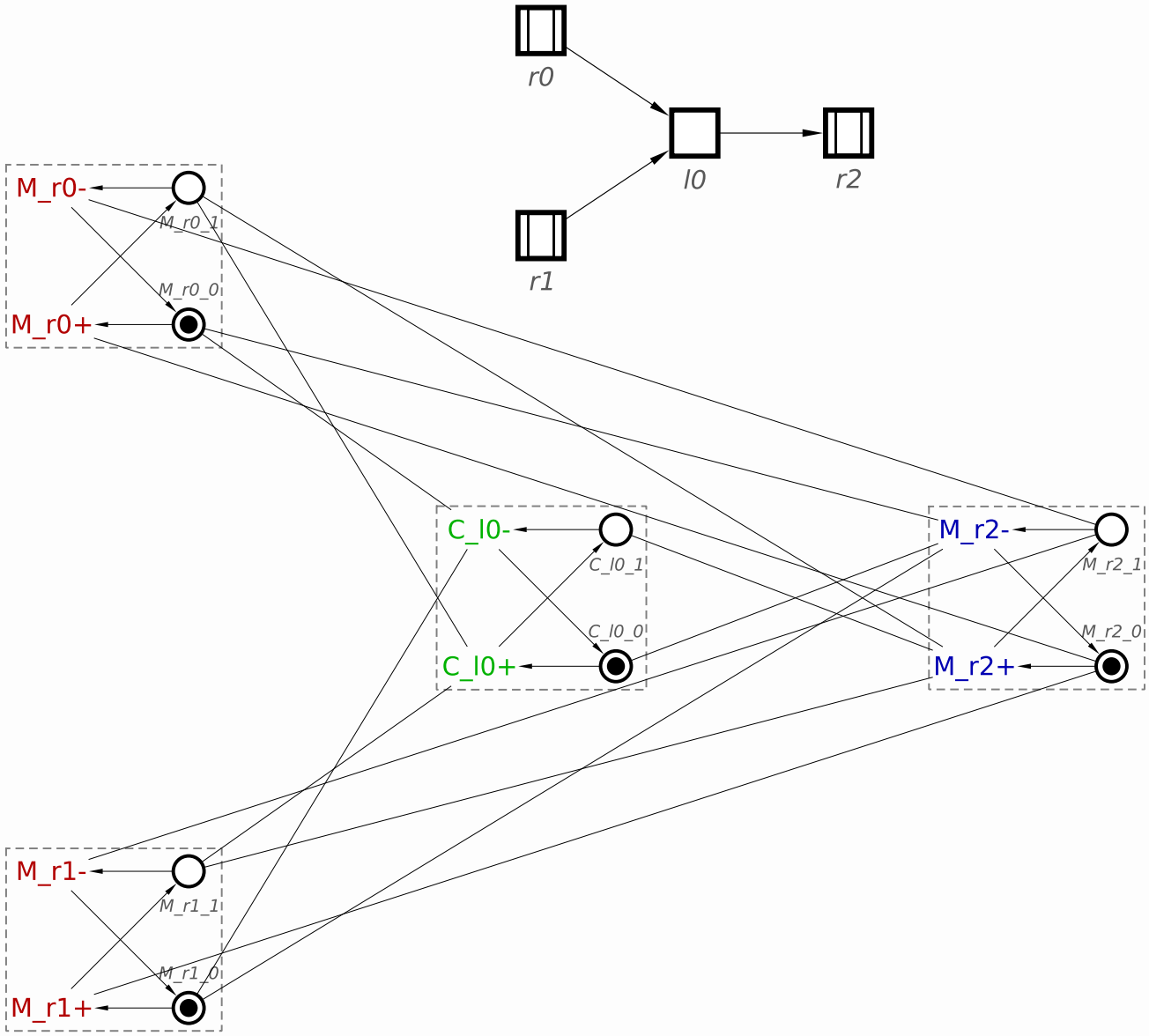
1. Circuit is converted to an equivalent STG – circuit STG
2. Internal signal transitions in the environment STG (contract between the circuit and its environment) are replaced by dummies
3. Circuit STG and environment STG are composed by PCOMP back-end
4. Conformation property is expressed in REACH language
5. Composed STG is unfolded by calling PUNF back-end
6. Unfolding prefix and REACH expression are passed to MPSAT back-end
7. Verification results are parsed by the front-end
8. Violation trace is projected to the circuit for simulation and debugging

# Circuit Petri nets as assembly language

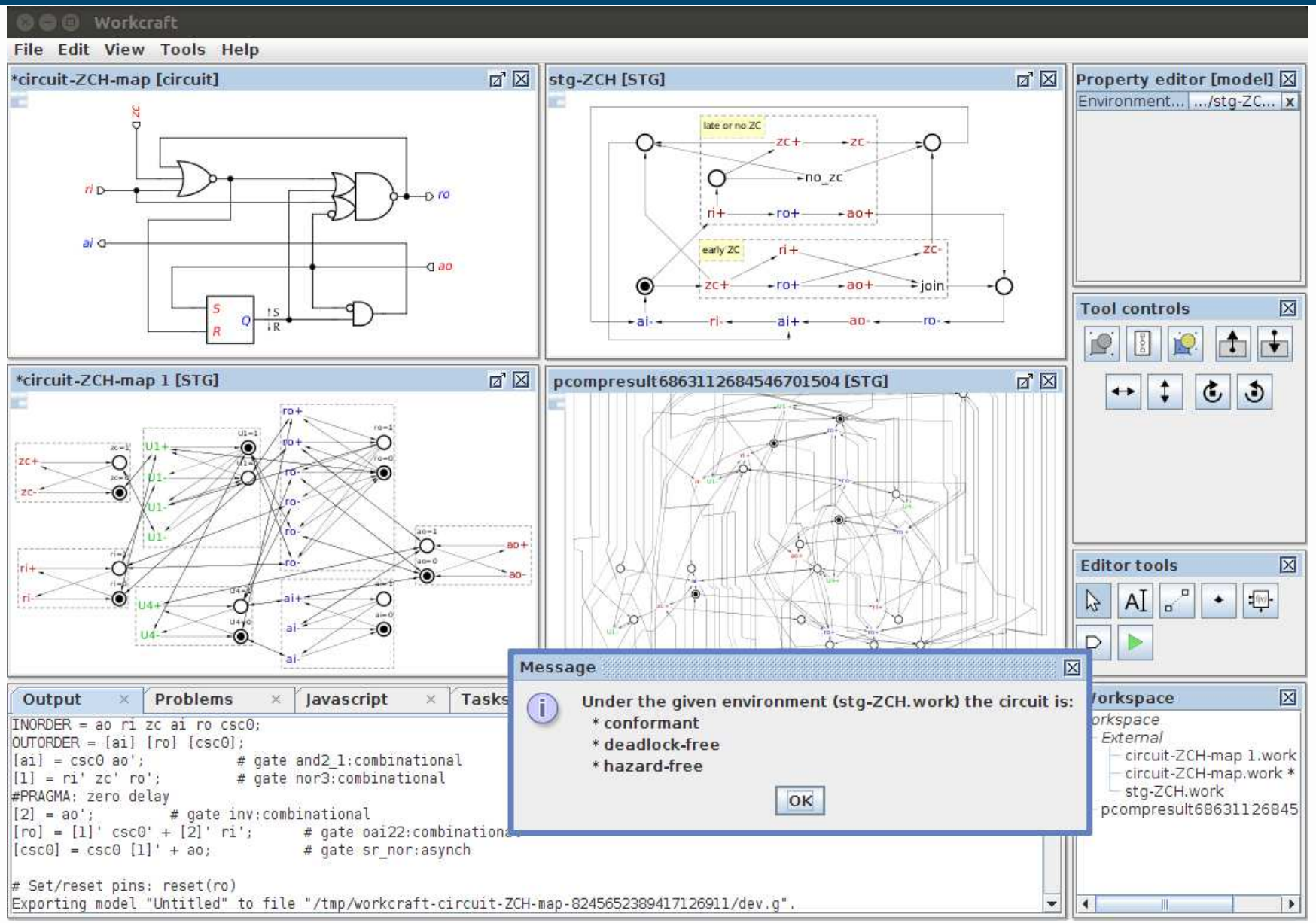


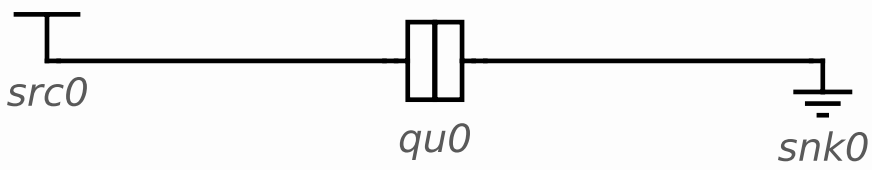
# Circuit Petri nets: Dataflow pipelines

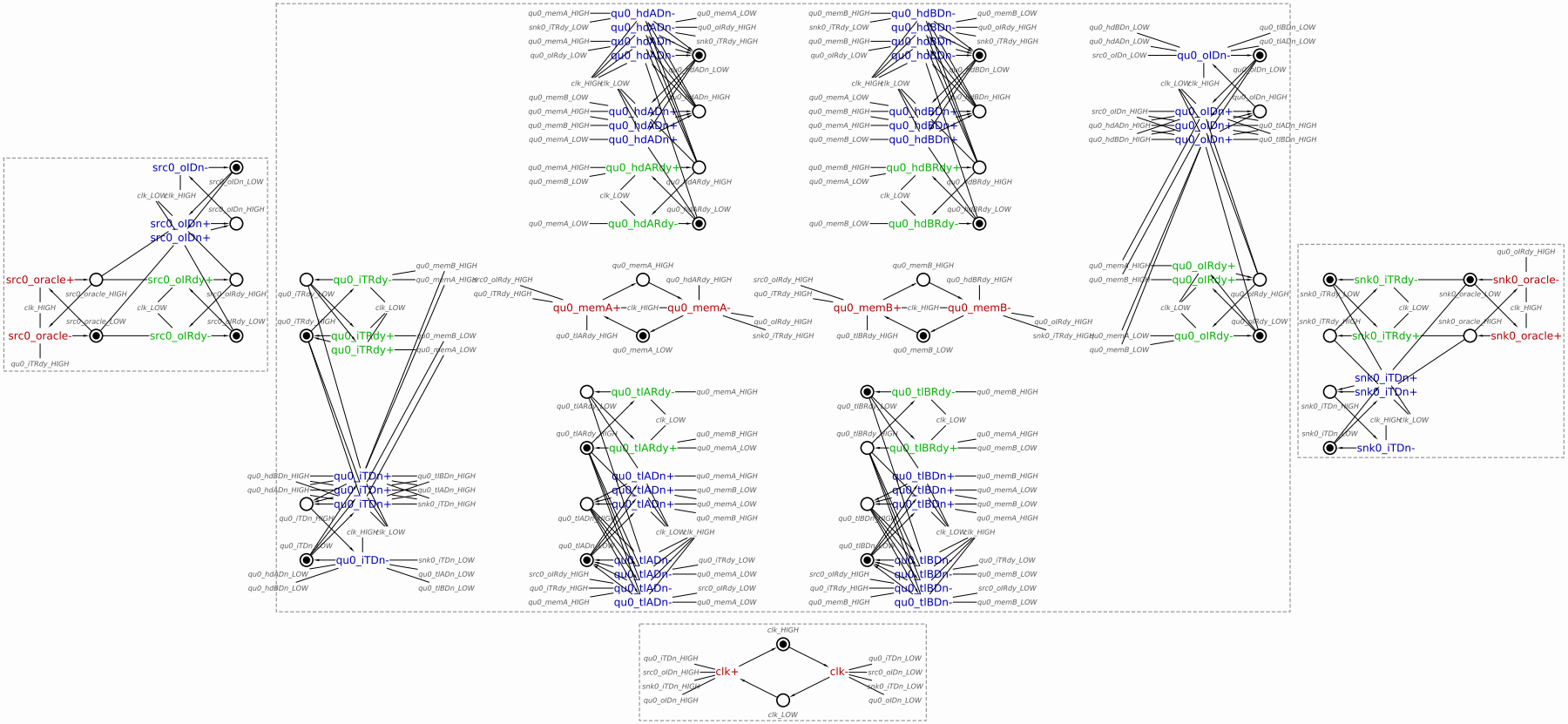


# WORKCRAFT live demo



# Circuit Petri nets: xMAS circuits



# A Workflow for the Design of Mixed-signal Systems with Asynchronous Control

**Vladimir Dubikhin**, Danil Sokolov, Alex Yakovlev, Chris J. Myers

# AMS Trends & Challenges

**Key Drivers**

- Internet of Things
- Mobile computing
- Automotive electronics

**Trends**

- Technology scaling
- Multiple power and time domains
- Analog and digital integration

**Challenges**

- Tighter reliability margins
- Concurrent analog and digital analysis
- Short development cycle

Based on slide from DAC2014 by ANSYS

# What this means for AMS?

- **Achieving better verification of *analog and digital blocks***
- ***Verifying the increasing amount of digital logic in analog designs***
- **Creating a *higher level of abstraction* for analog and mixed signal blocks**
- ***Automating* the manual custom design steps**
- **Adopting *circuit analytics* that tell *why* and *where* the circuit is failing to perform**

Based on slide from ISQED2013 by Mentor Graphics

# Why Asynchronous Logic?

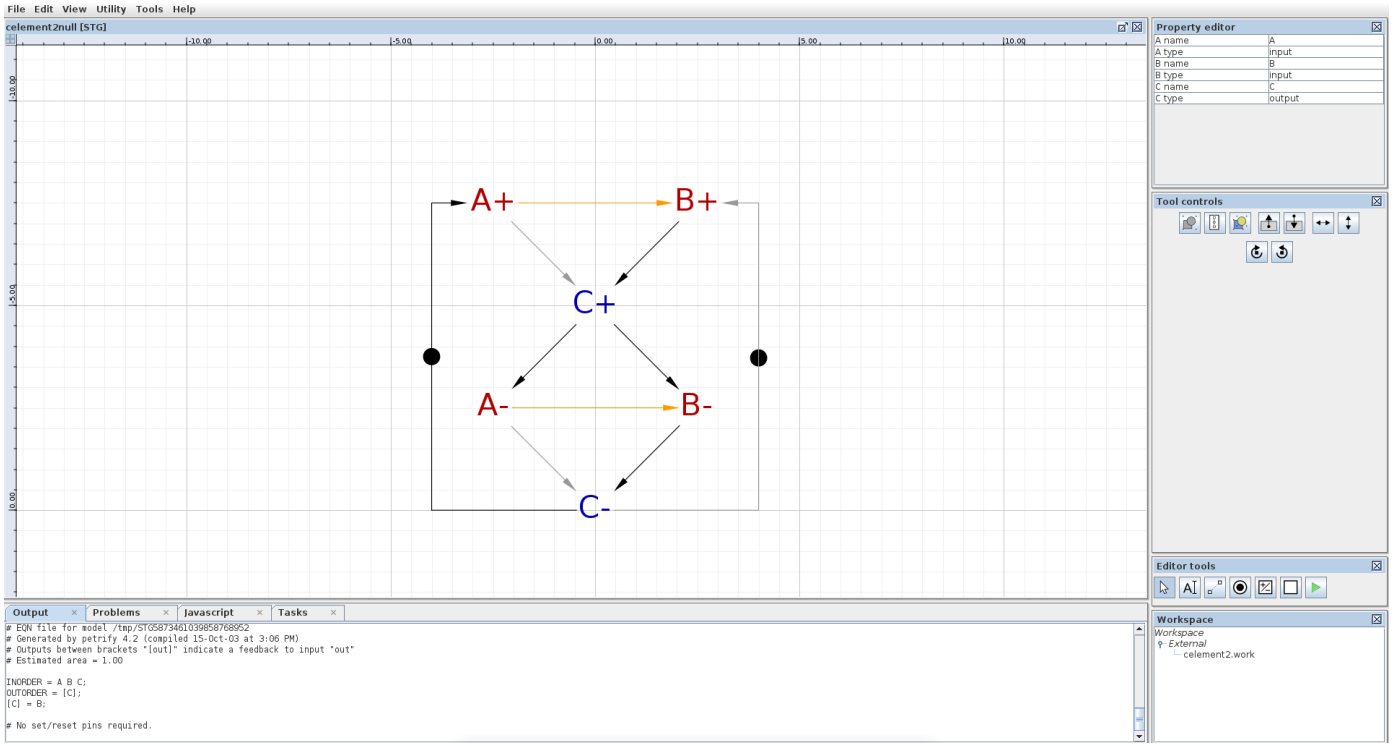
- Insensitive to delays
- Robust to process-voltage-temperature
- Average case performance
- Low power consumption and EMI

# Why Asynchronous Logic?

- Insensitive to delays
- Robust to process-voltage-temperature
- Average case performance
- Low power consumption and EMI
- Incompatible with commercial EDA tools

# Workcraft

- Modeling with signal transition graphs (STG)
- Formal verification of STG models
- Logic synthesis of asynchronous circuits




Available at http://workcraft.org/

# Why Formal Verification?

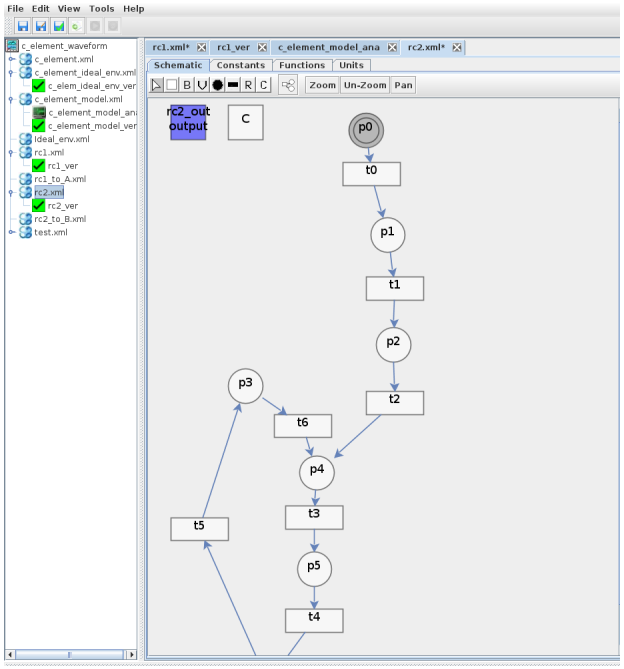
- Increased robustness of the system
- Abstract modeling
- Reduced need for conventional simulation

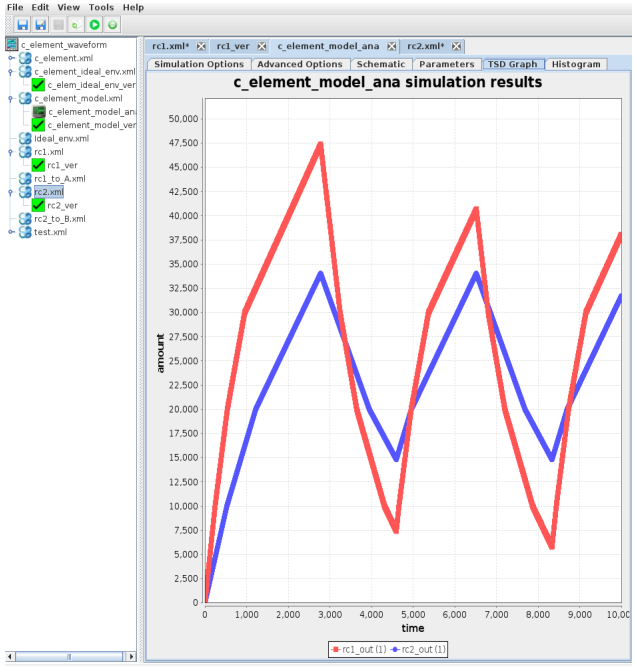
# Why Formal Verification?

- Increased robustness of the system
- Abstract modeling
- Reduced need for conventional simulation
- Limited tool support

# LEMA

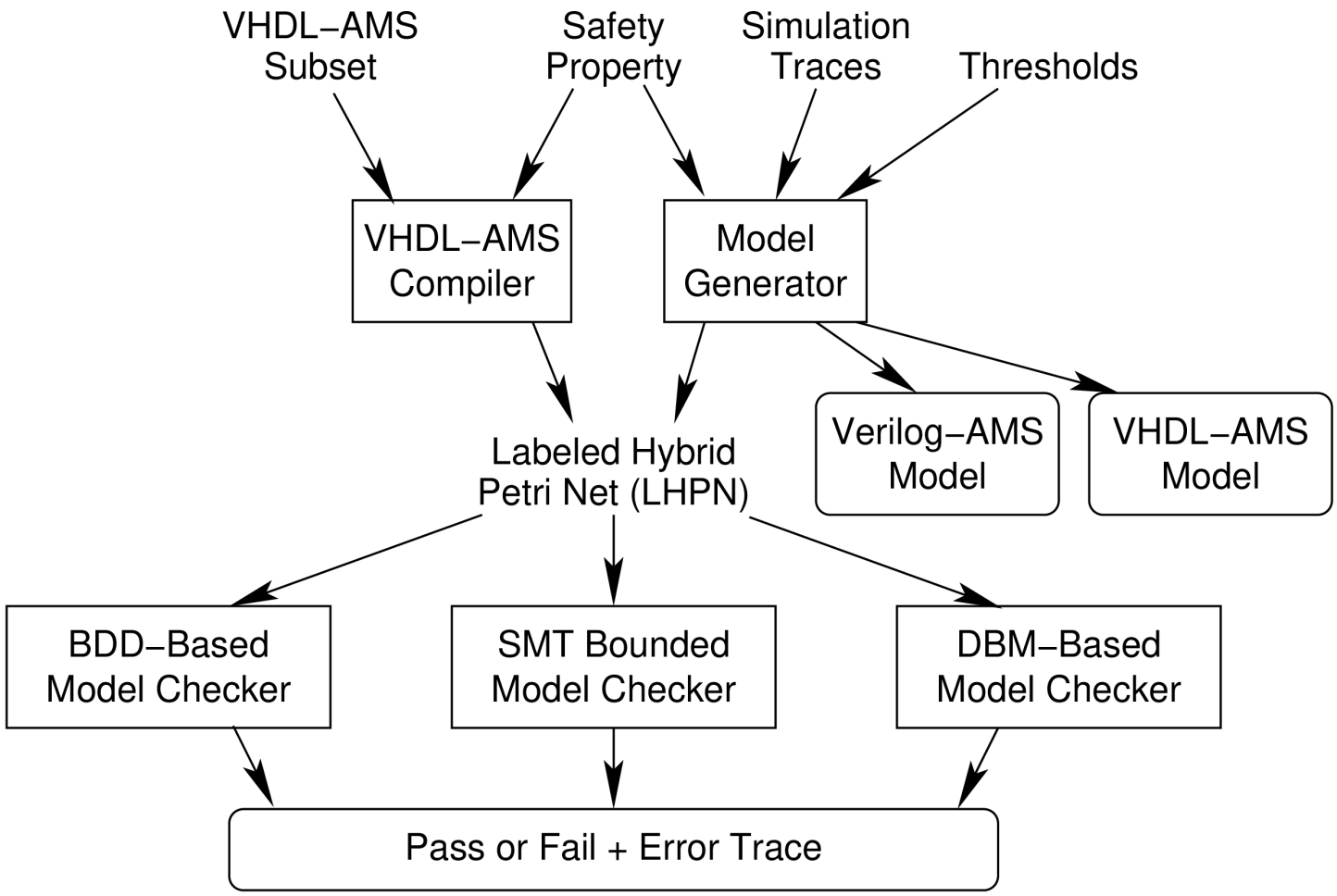
- Modeling with labeled Petri nets(LPN)
- Automatic model generation
- Property expression and checking
- Model extraction as SystemVerilog netlist



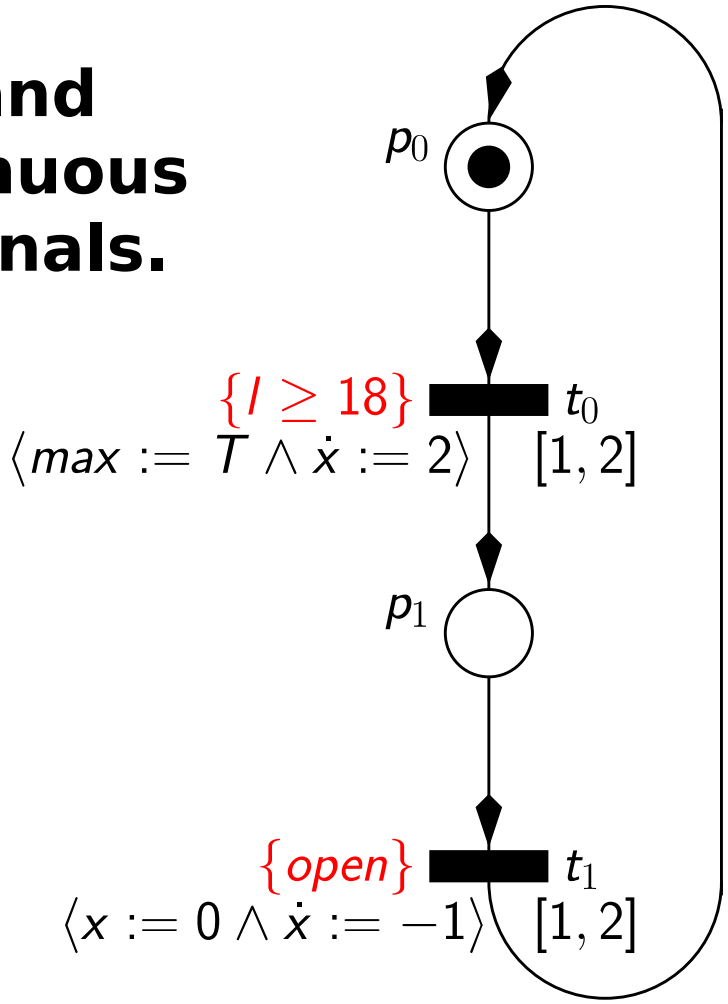

Available at http://async.ece.utah.edu/LEMA/

# LEMA Tool Flow



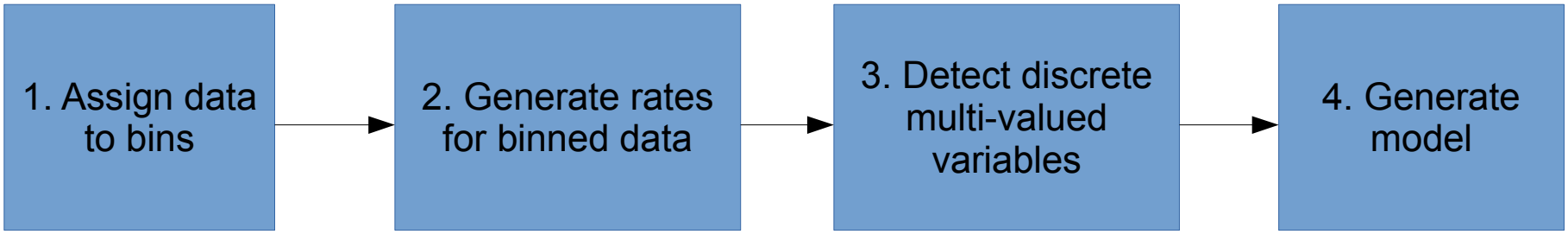
# Labeled Petri Nets (LPNs)

- **Composed of a Petri net and labels operating on continuous variables and Boolean signals.**
- **Label types are:**
  - **Enablings**
  - **Delay bounds**
  - **Boolean assignments**
  - **Value assignments**
  - **Rate assignments**

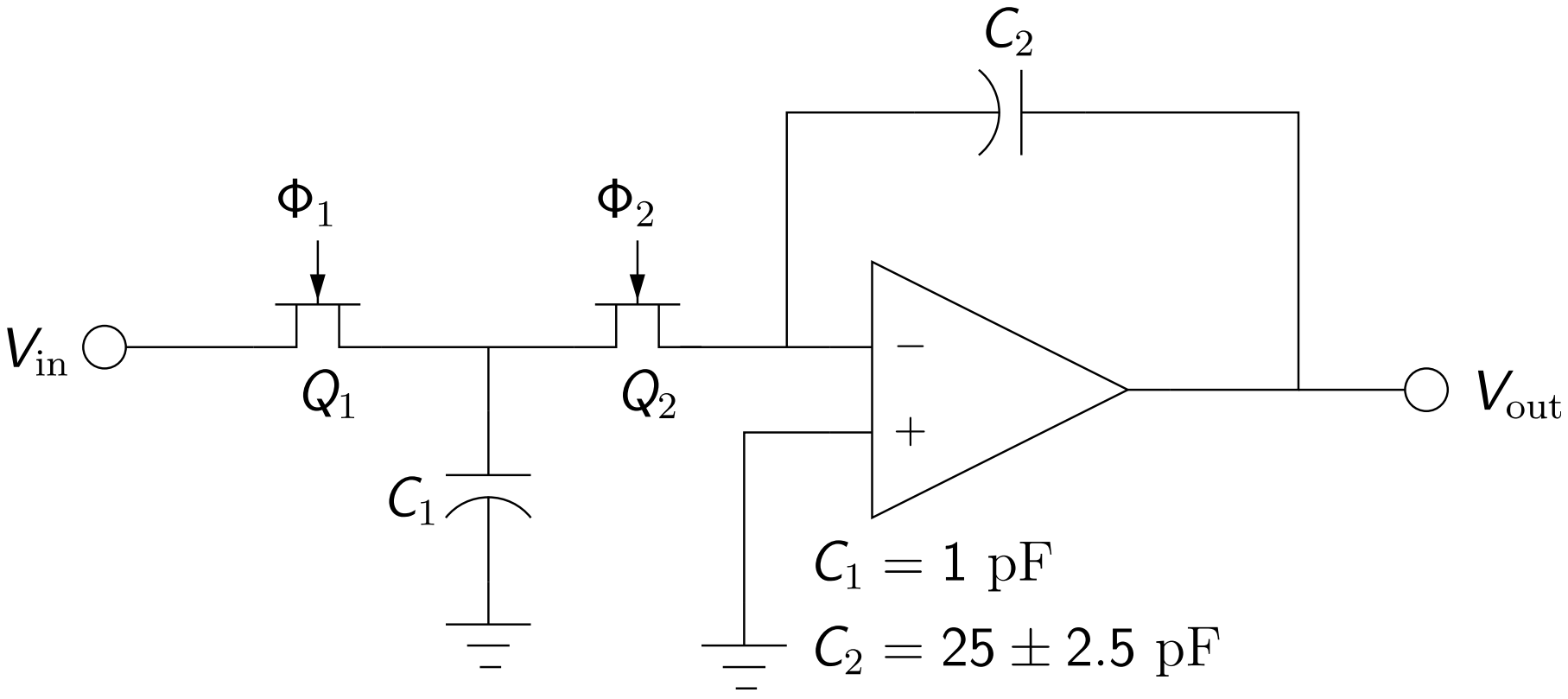


# LPN Model Generation

- **Build abstract models of the circuit using:**
  - **Simulation traces.**
  - **Thresholds on the design variables.**
  - **A property to verify.**



# Switched Capacitor




$C_1 = 1$ pF

$C_2 = 25 \pm 2.5$ pF

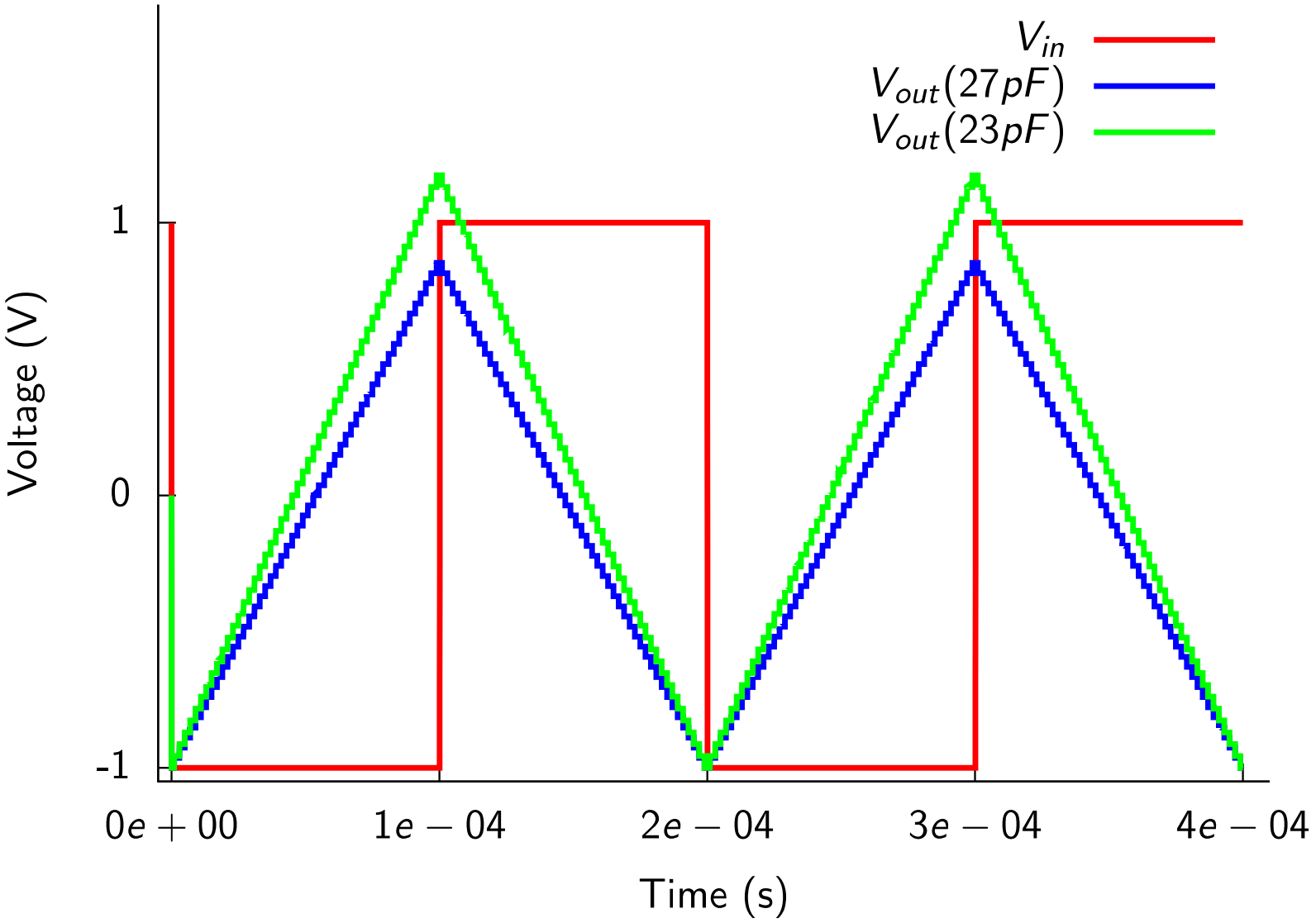
$V_{\text{in}} = \pm 1000$ mV

$freq(V_{\text{in}}) = 5$ kHz

$freq(\Phi_1) = freq(\Phi_2) = 500$ kHz

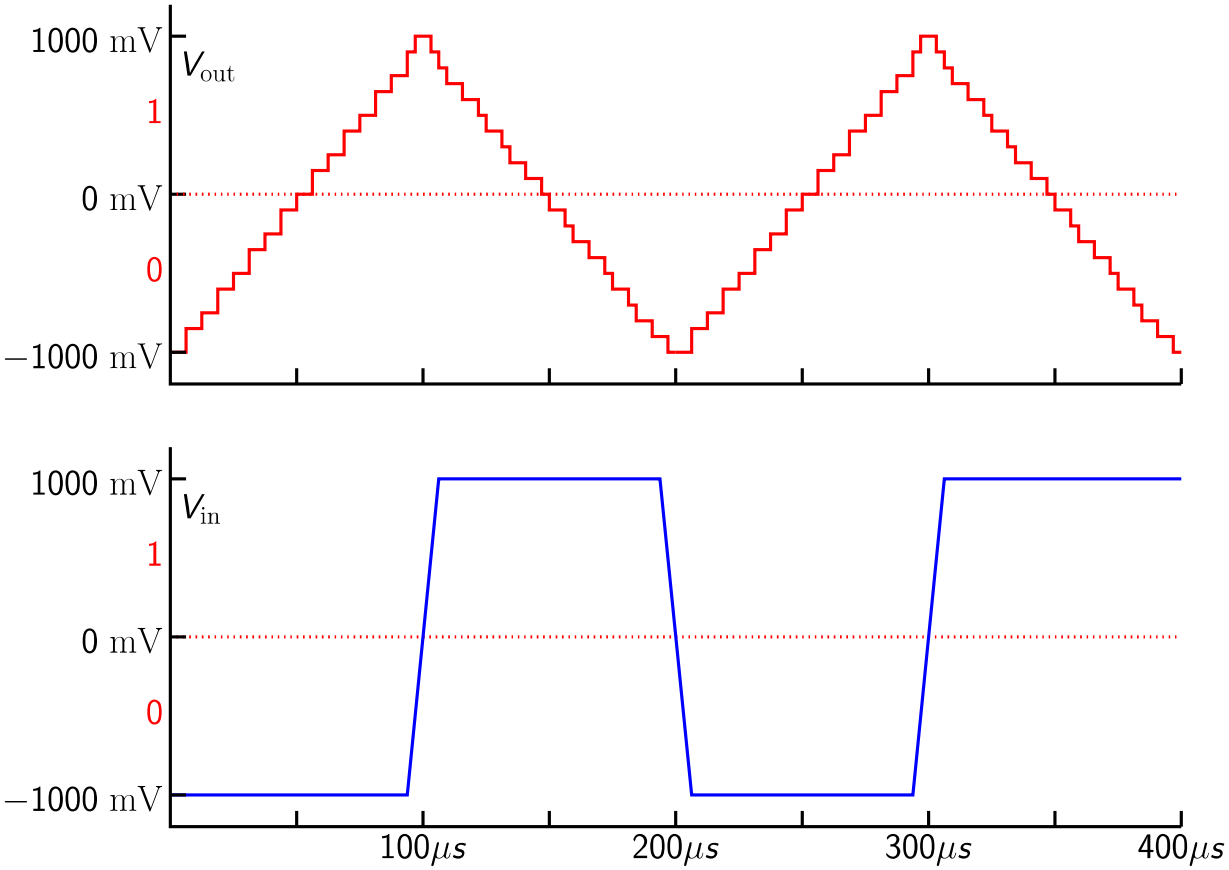
$dV_{\text{out}}/dt = \pm(18 \text{ to } 22)$ mV/$\mu$s

# Simulation Trace

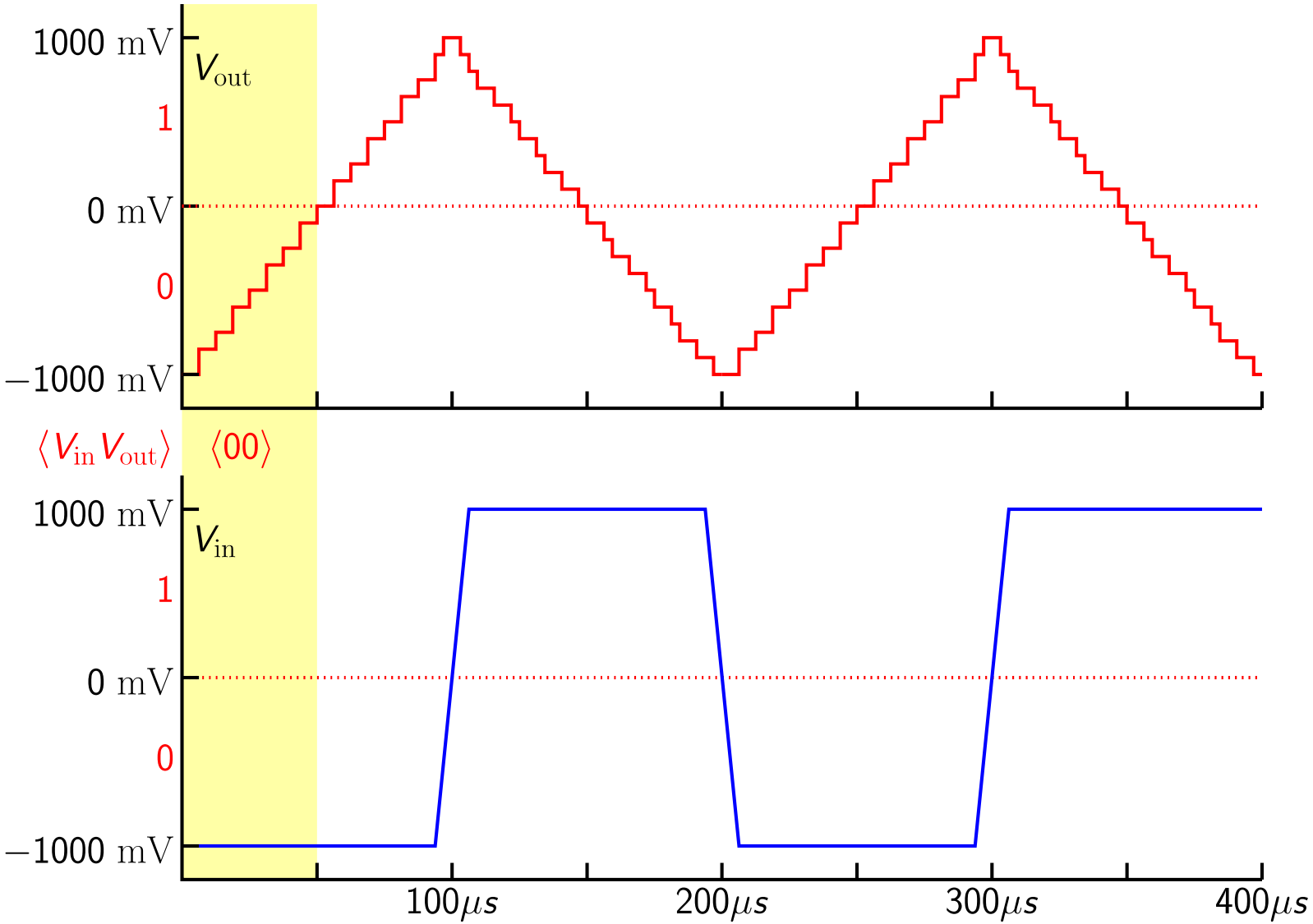


# Data Binning

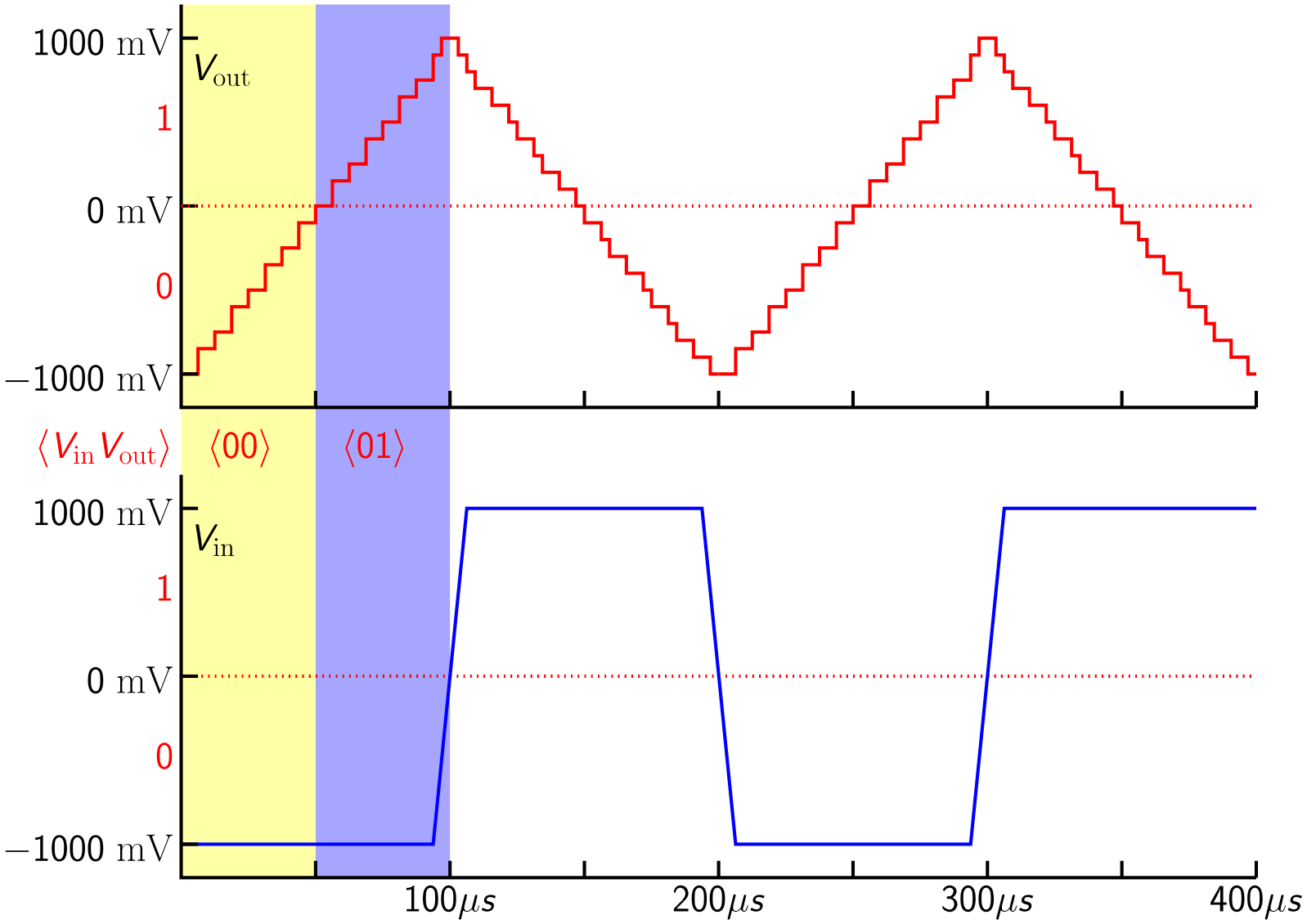
- **Each data point is assigned a bin based upon thresholds.**
- **Each bin represents an operating region of the system.**



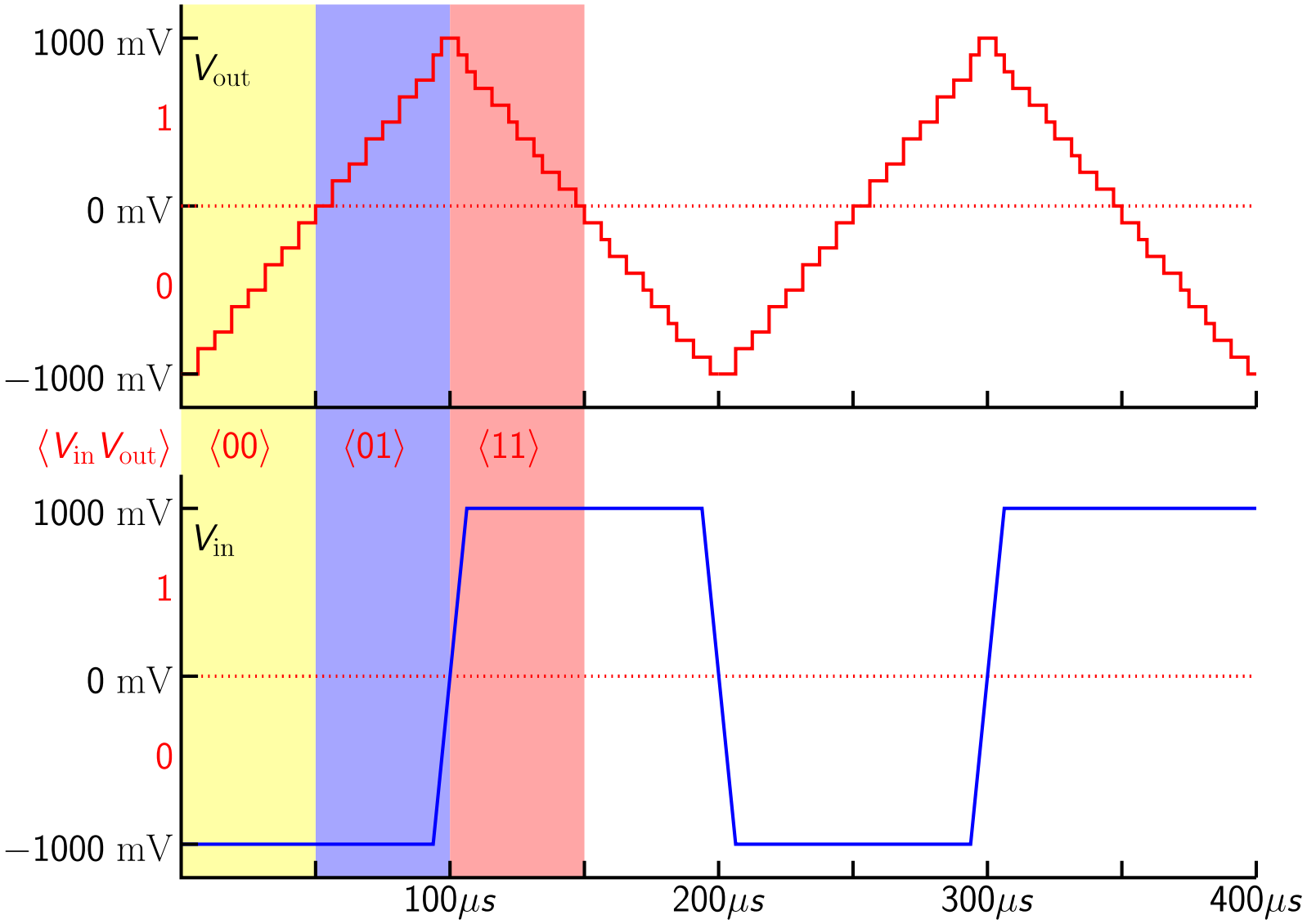
# Data Binning



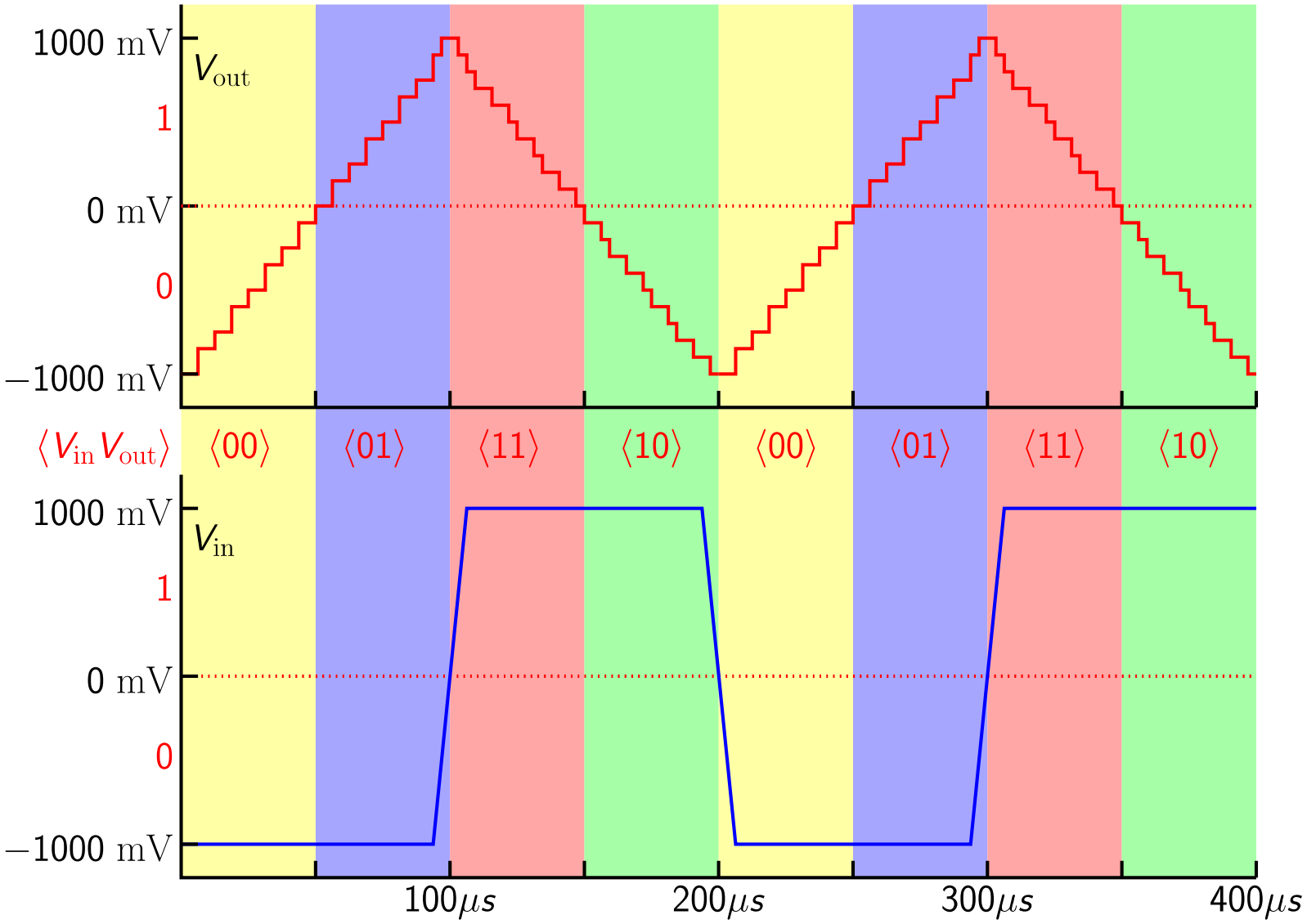
# Data Binning



# Data Binning

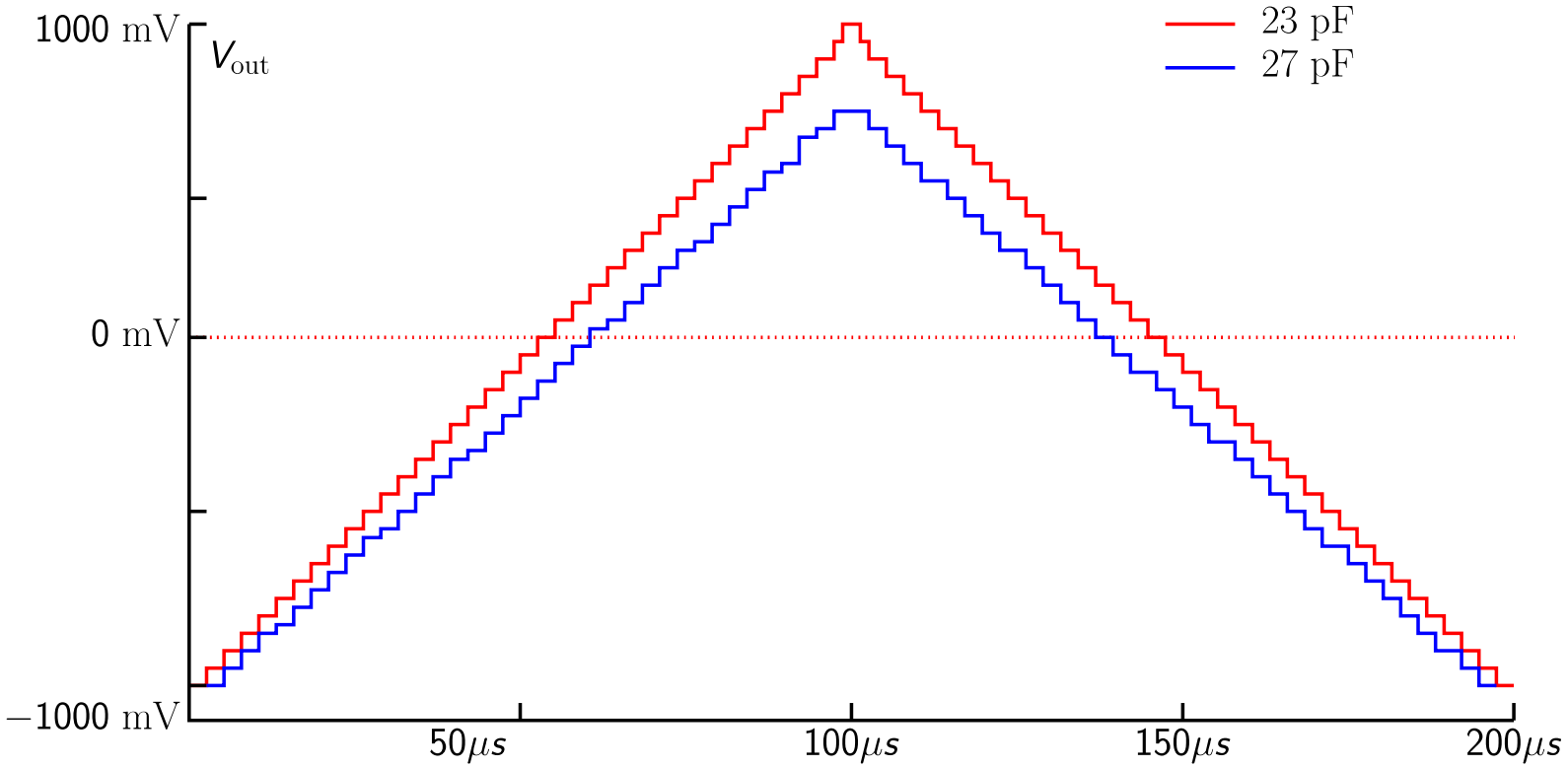


# Data Binning

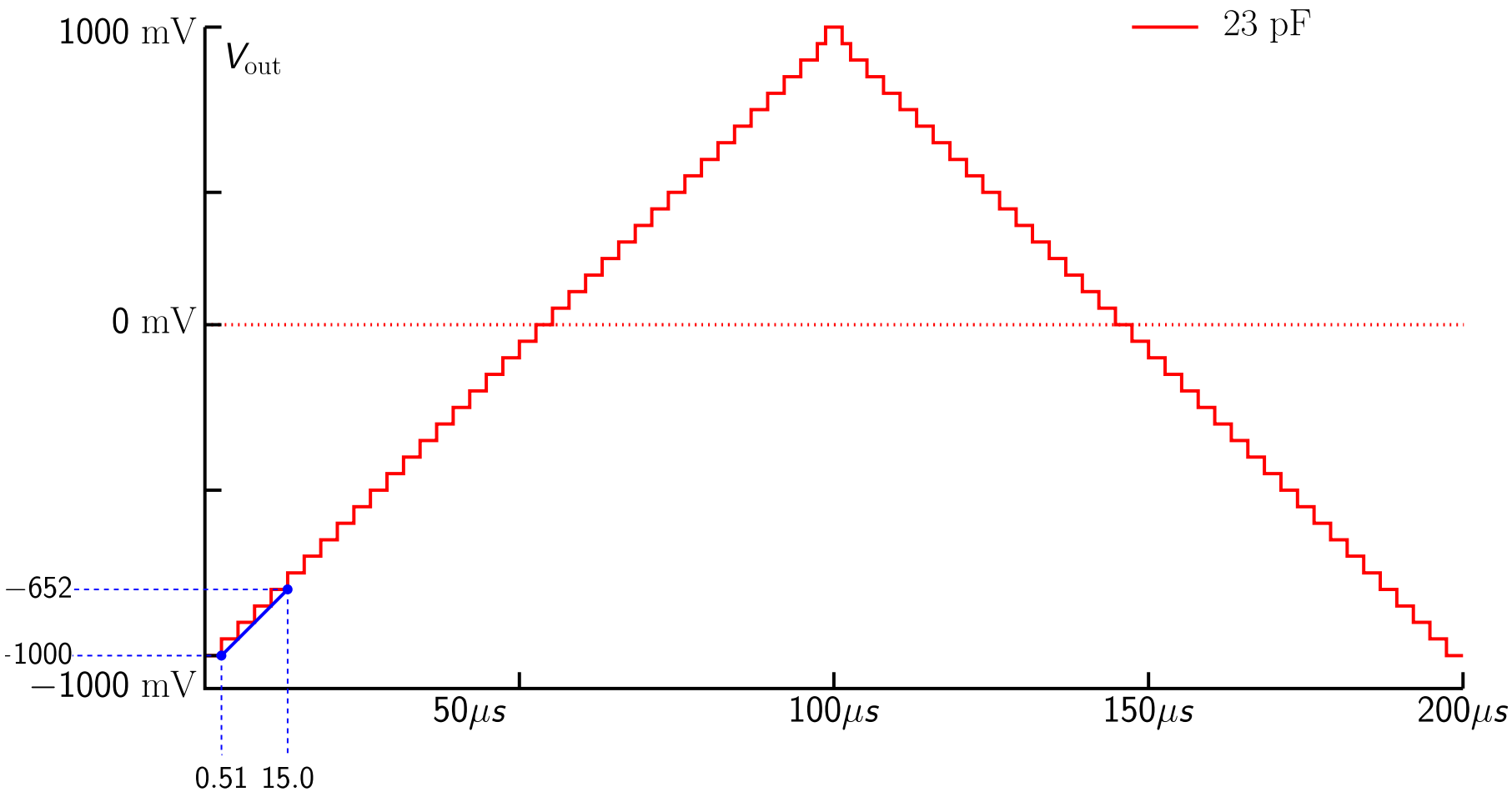


# Rate Calculation

- **Rates are calculated for each eligible data point in each bin.**
- **Low pass filtering smooths edge effects and transitory pulses.**
- **Minimum and maximum rates are tabulated for each bin.**



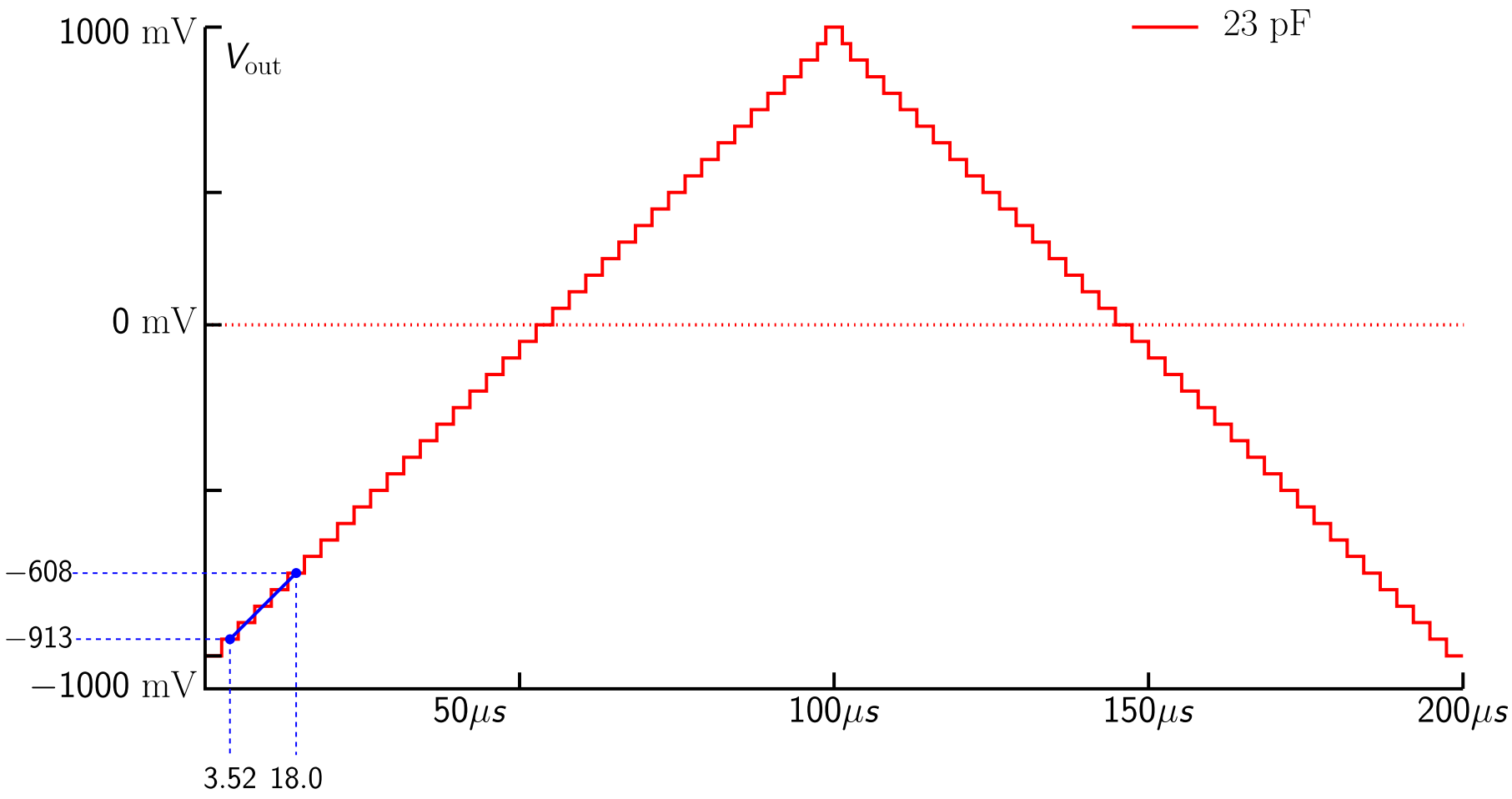
# Rate Calculation




$(-652--1000)/(15.0-0.51)=23.93\,mV/\mu s$

$V'_{out\,00}=[23,24]$

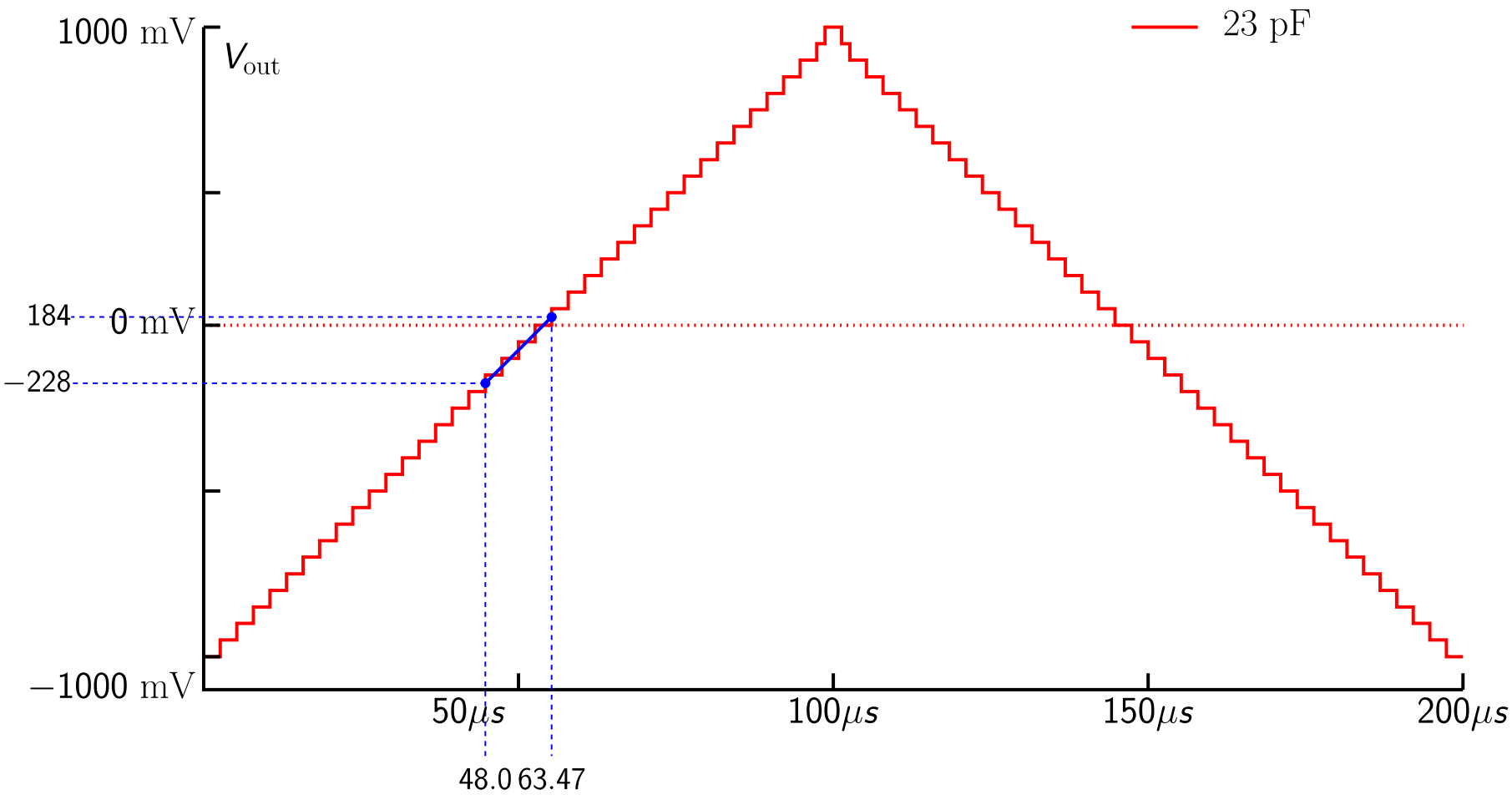
# Rate Calculation




$(-608--913)/(18.0-3.52)=21.06 mV/\mu s$

$V'_{out\,00}=[21,24]$

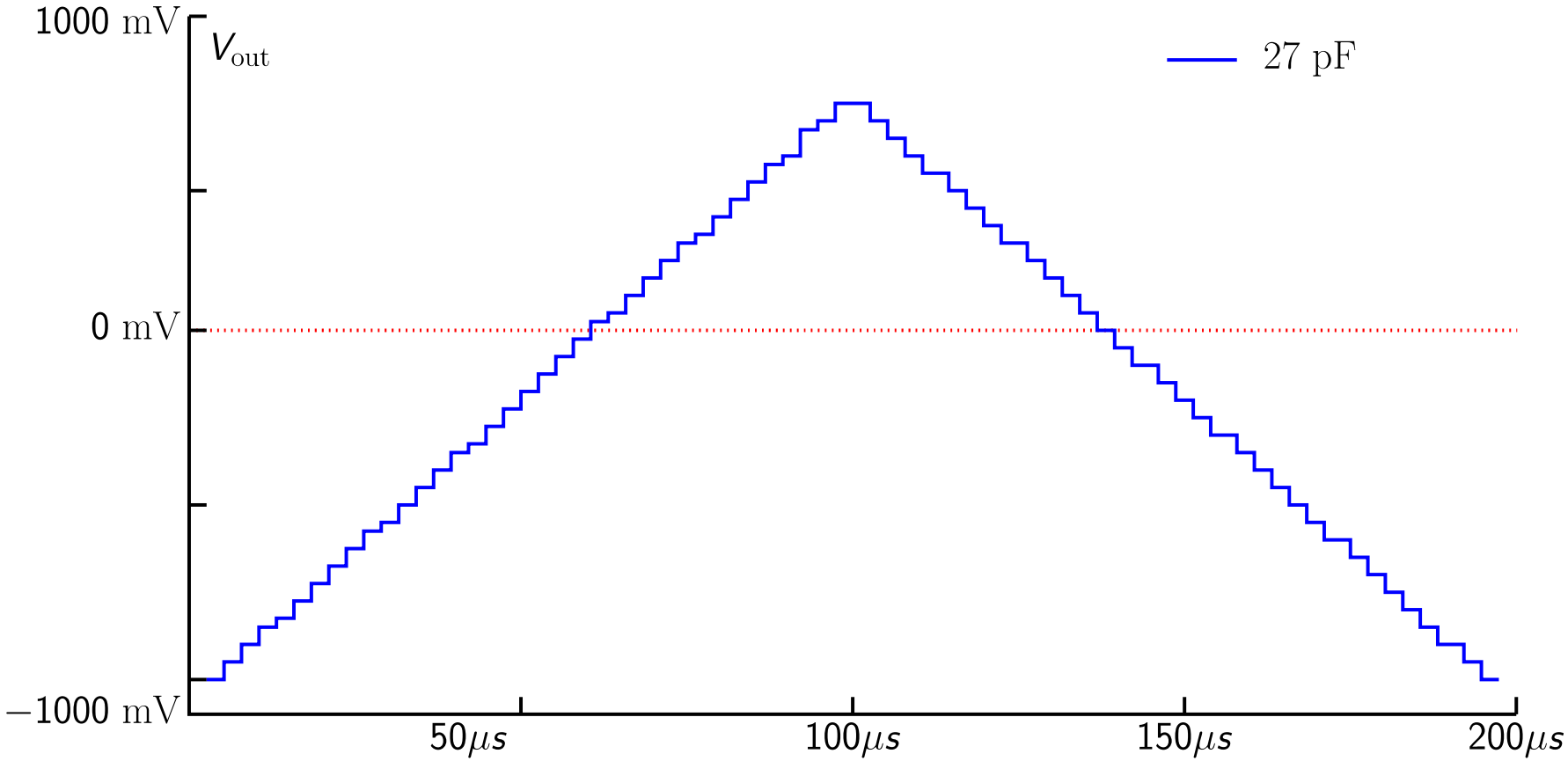
# Rate Calculation




*No rate calculated.*

$V'_{out\,00}=[19,24]$

# Rate Calculation




*Final rate calculations after* $C_2$=27$pF$.

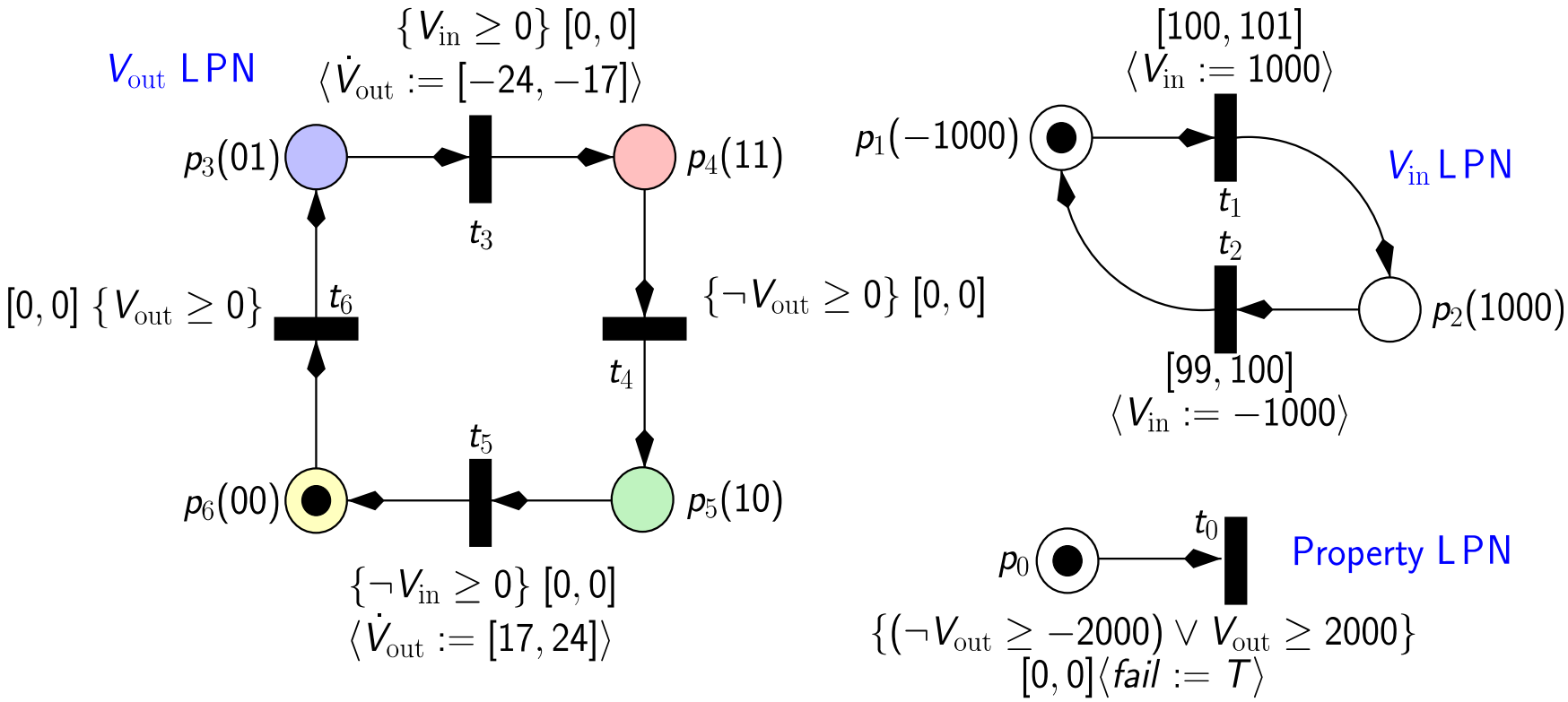
$V'_{out\,00}$=[17, 24]

# DMV Variables

- **Stable signals are handled differently to aid efficiency.**
- **Stability is determined by:**
  - **Remaining constant within an epsilon value for a specified time.**
  - **Total percent of the entire signal marked stable.**
- **Delay is calculated for each constant value.**
- **Min/max delay and constant values are extracted.**

# Generating an LPN

$$\textit{Initial values} = \{V_{out} = -1000\,mV, V_{\text{in}} = -1000\,mV, \textit{fail} = F\}; \textit{Initial rates} = \{V'_{\text{in}} = 0, V'_{out\,00} = [17, 24]\}$$

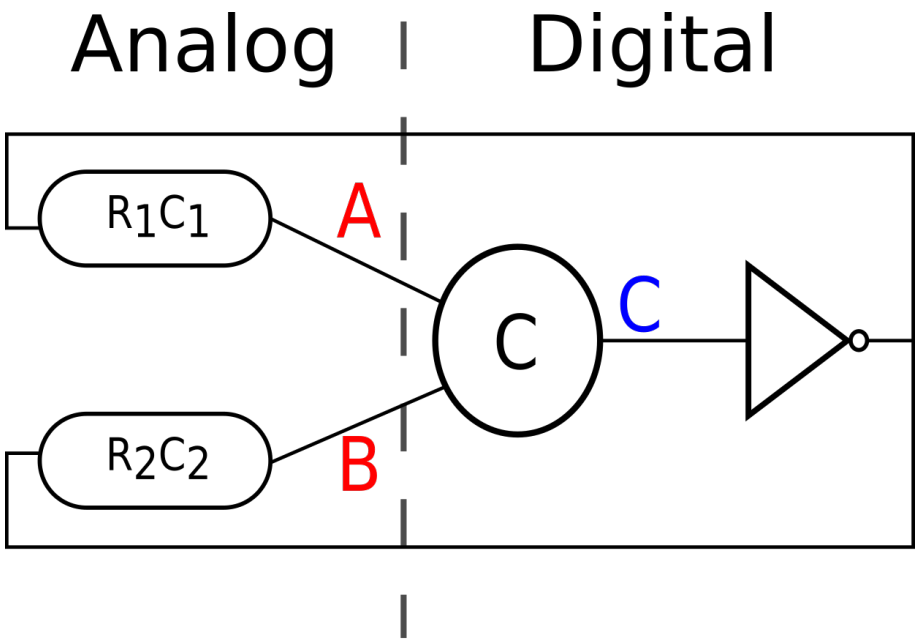


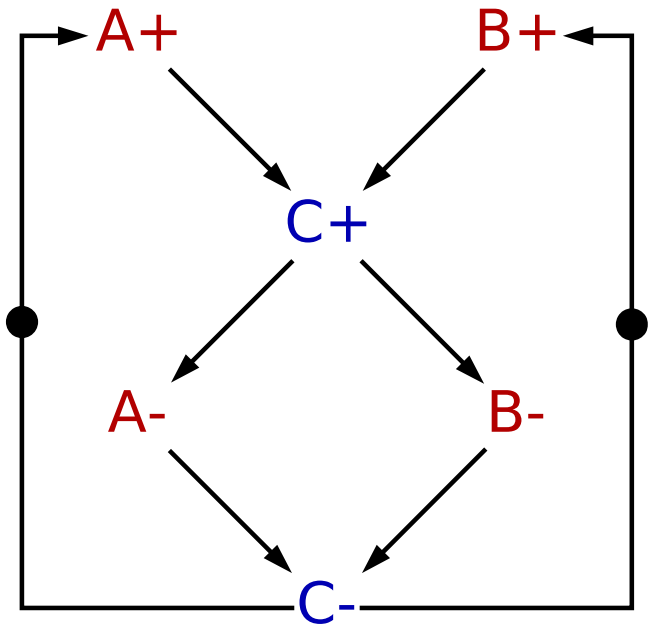
# Property Language

- **delay(d) - wait for d time units.**
- **wait(b) - wait until boolean expression, b, becomes true.**
- **waitPosedge(b) - wait for a positive edge on b.**
- **wait(b, d) - wait at most d time units for b to become true.**
- **assert(b, d) - ensure that b remains true for d time units.**
- **assertUntil(b1, b2 ) - ensure that b1 remains true until b2 is true.**
- **if-else - statement for selections.**
- **always(conditionsList){statements} - continue to execute statements until one of the signals in the list of variables condistionsList changes, then break out.**

# LEMA DEMO

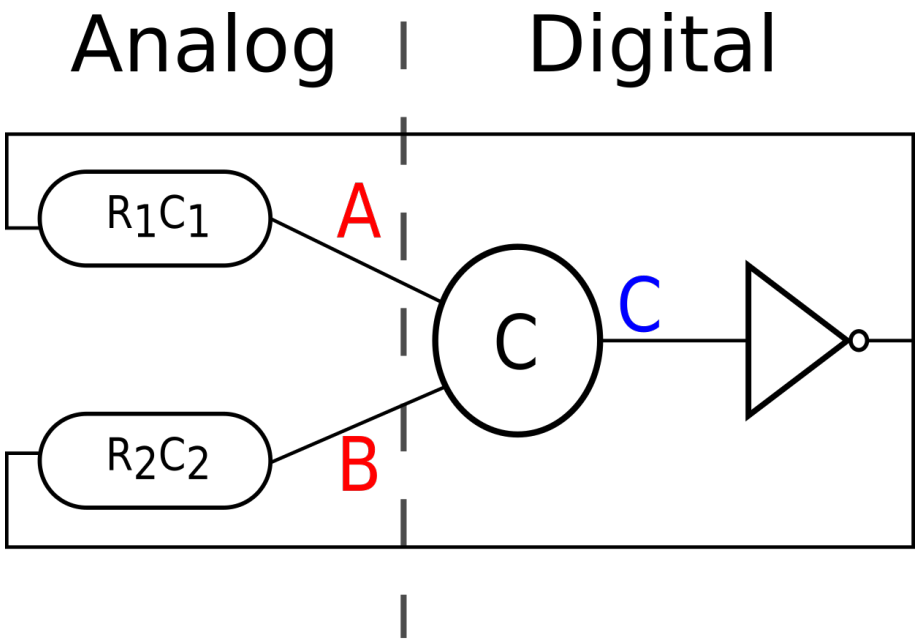
# C-element Example

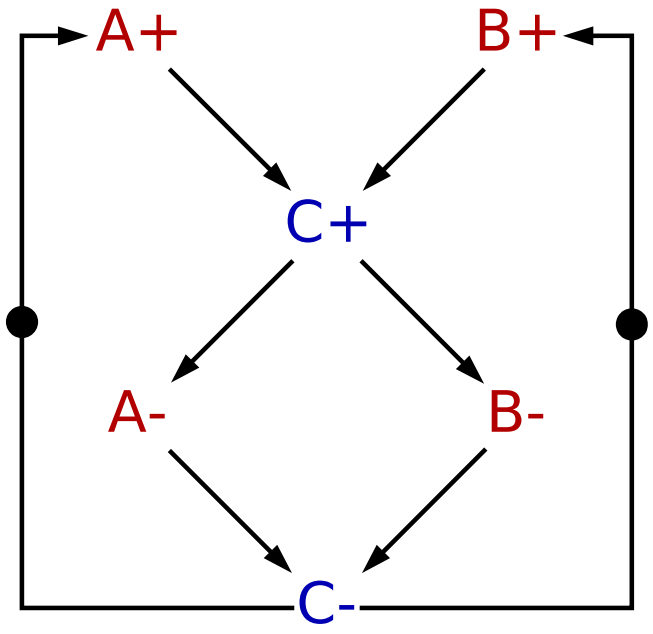



$$R_1 C_1 ? R_2 C_2$$

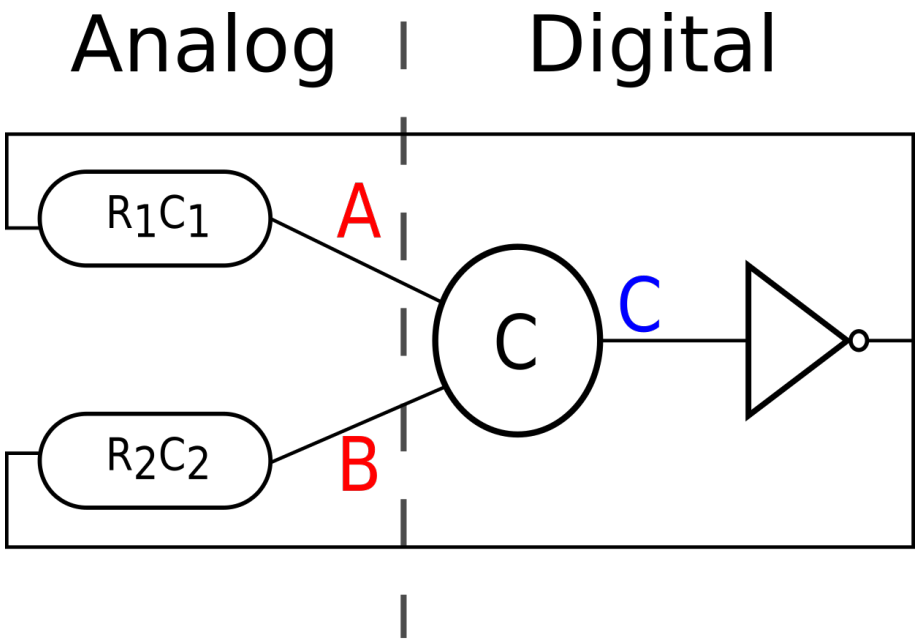
# C-element Example

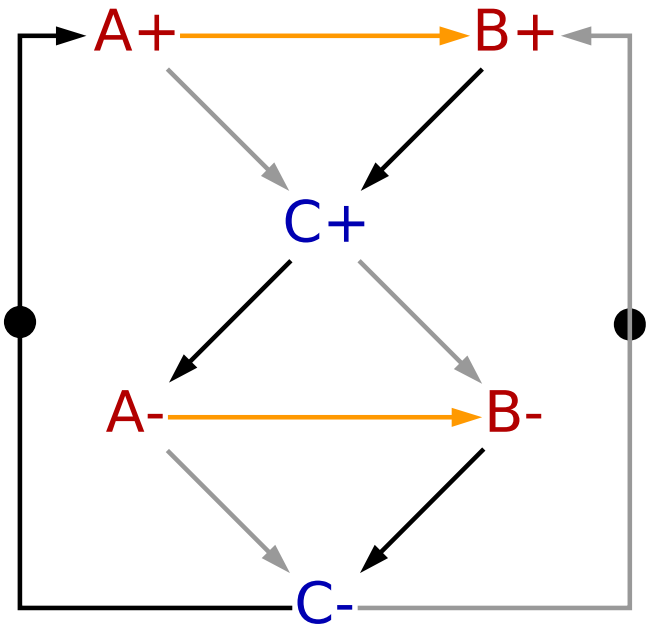



$$R_1 C_1 < R_2 C_2$$

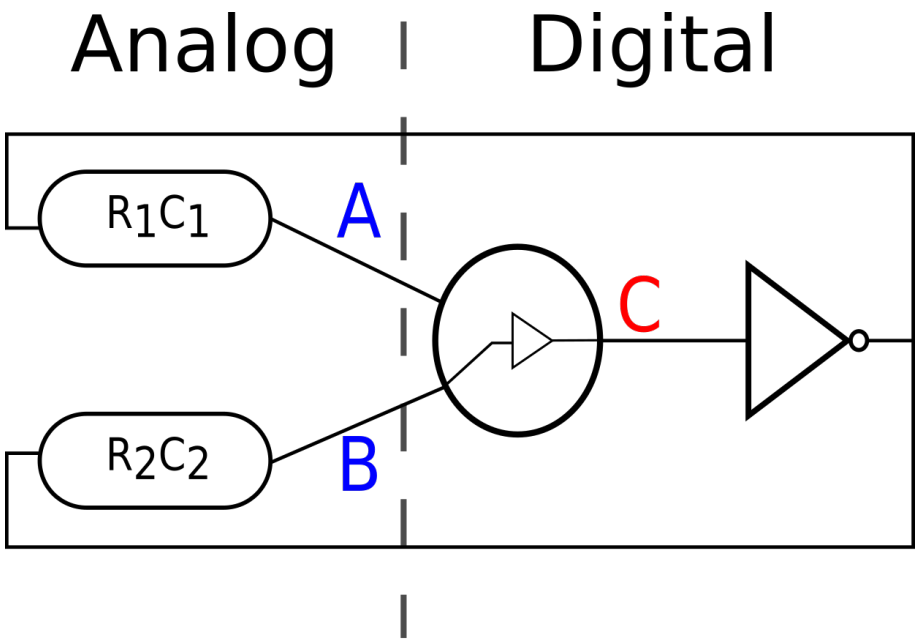
# C-element Example

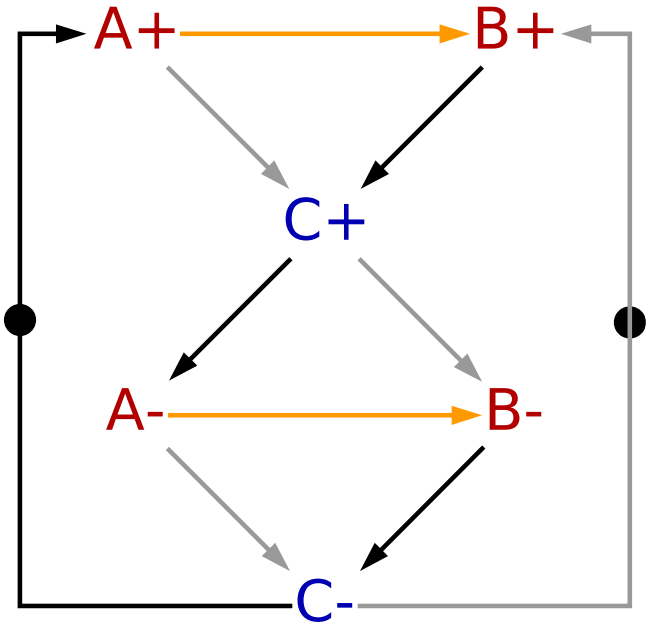



$$R_1 C_1 < R_2 C_2$$

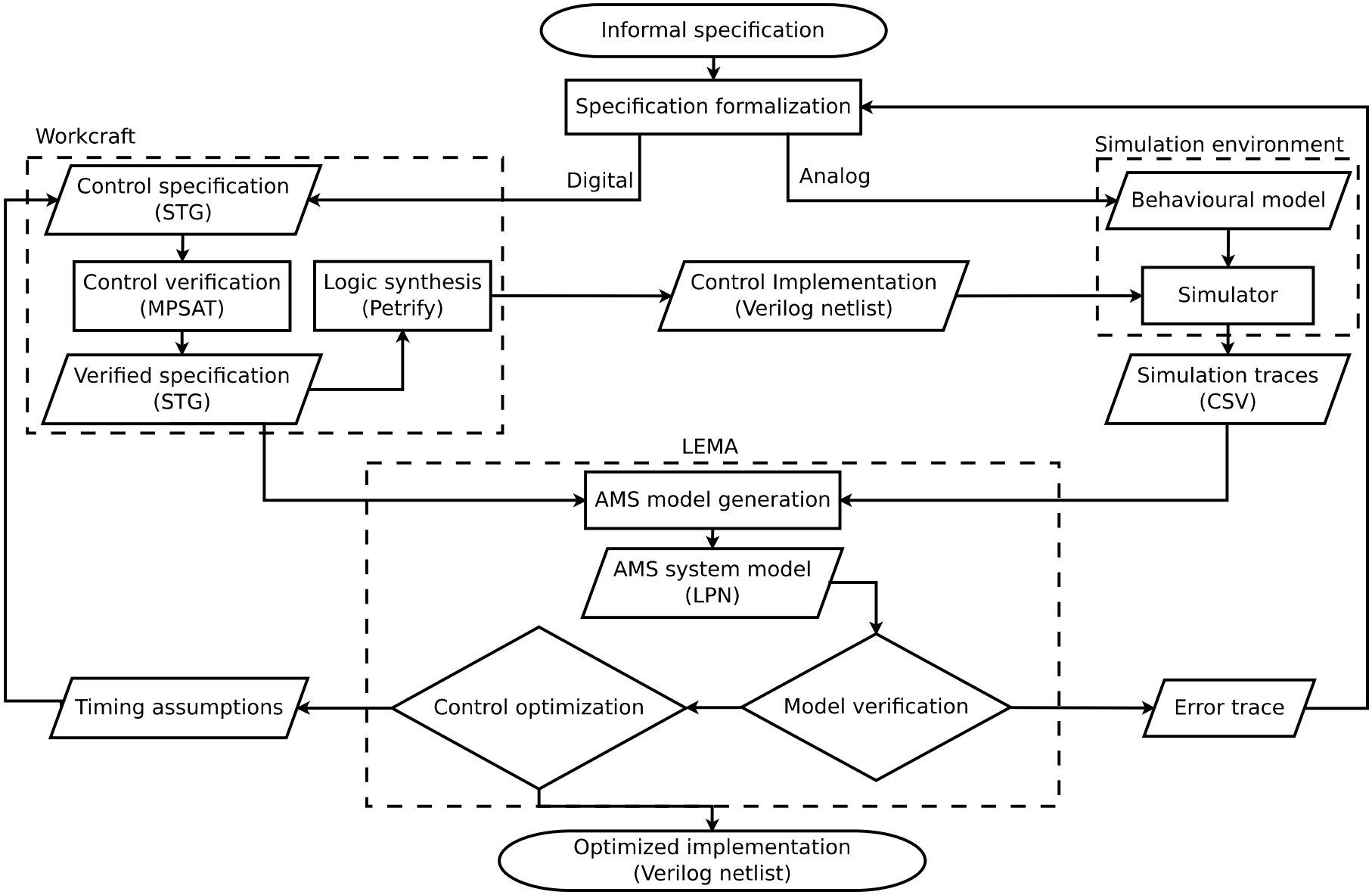
# C-element Example



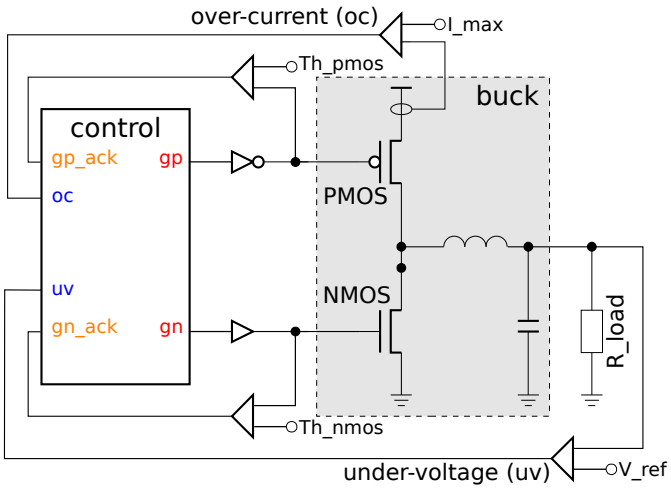



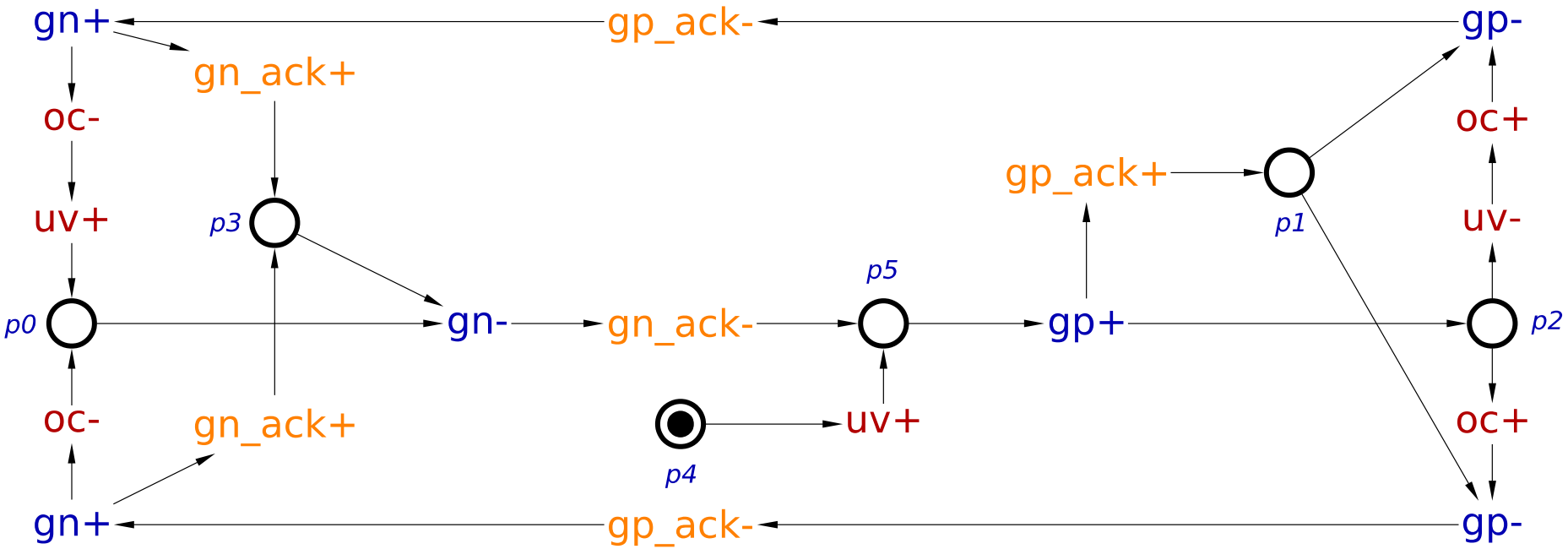

$$R_1 C_1 < R_2 C_2$$

# AMS verification workflow

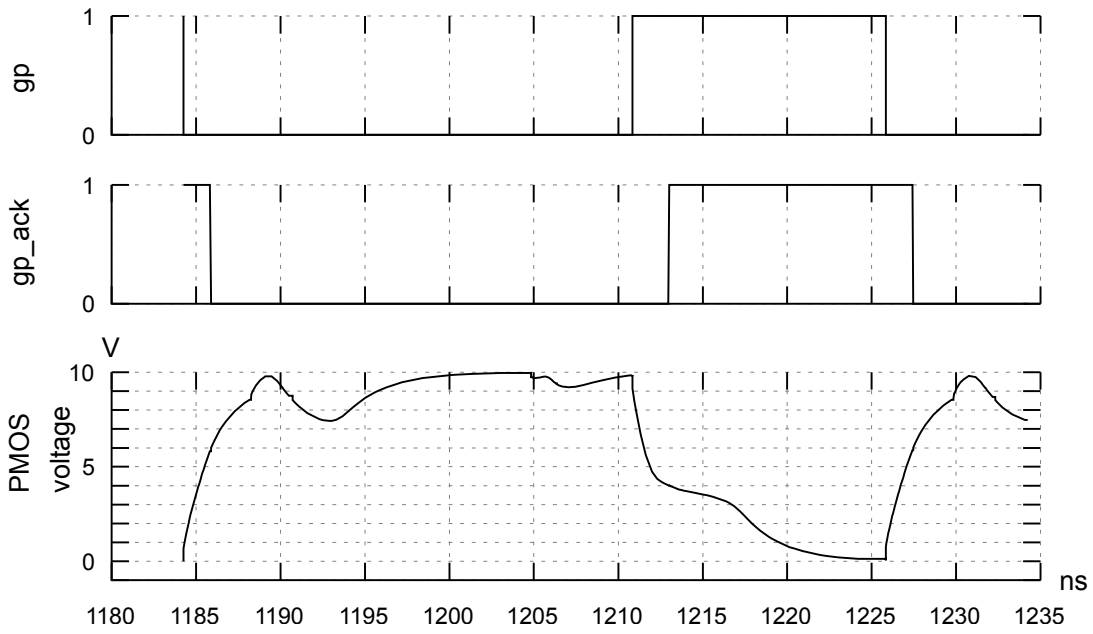


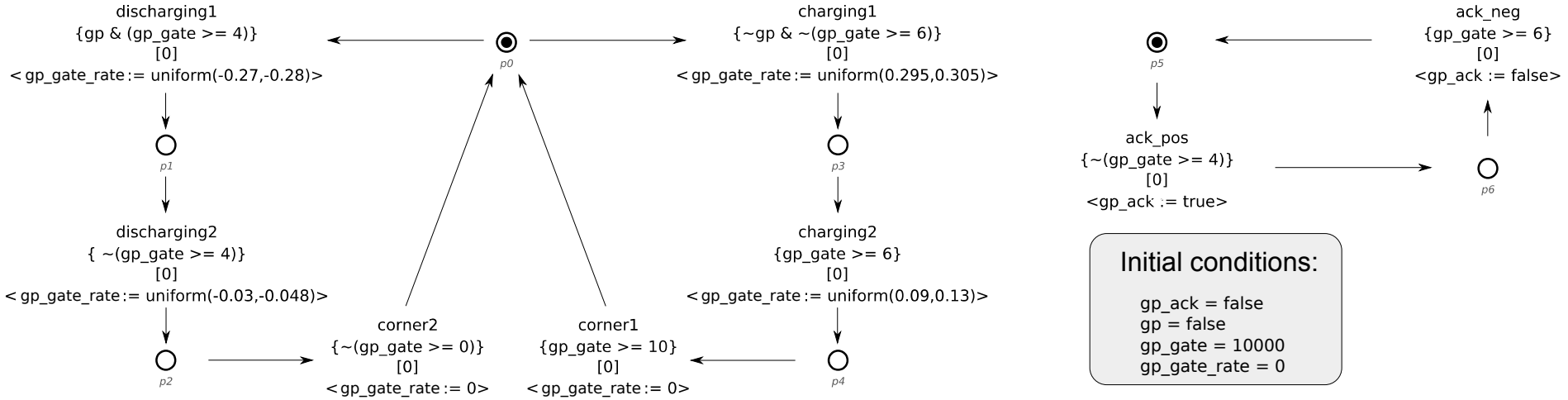
# Buck converter



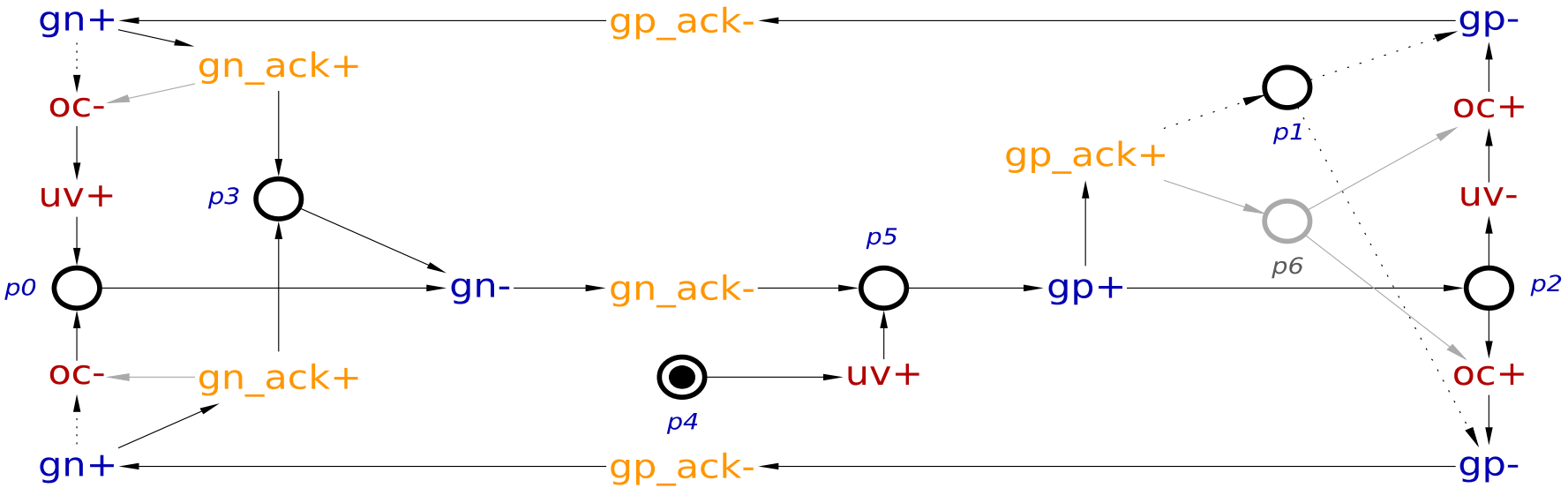
# Model generation example



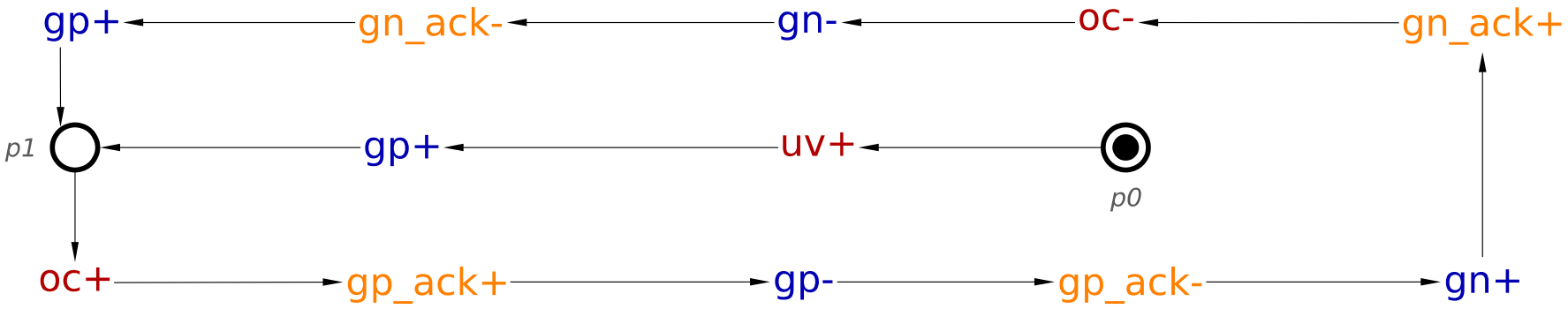



# Optimized specification

- **Concurrency reduction**




- **Scenario elimination**



# Verification challenges

- **Modules partitioning** – trade-off between model's accuracy and verification speed
- **False positives** – dealing with verification false fail states due to overapporximation
- **Properties expression** – models properties expressed via non-standard language