Asynchronous Circuits:

Formal Verification and Synthesis

Victor Khomenko, Andrey Mokhov,
Danil Sokolov, Alex Yakovlev

PN/ACSD'15: Advanced Tutorial, Brussels, June 2015

Formal Verification of Asynchronous Circuits

2 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

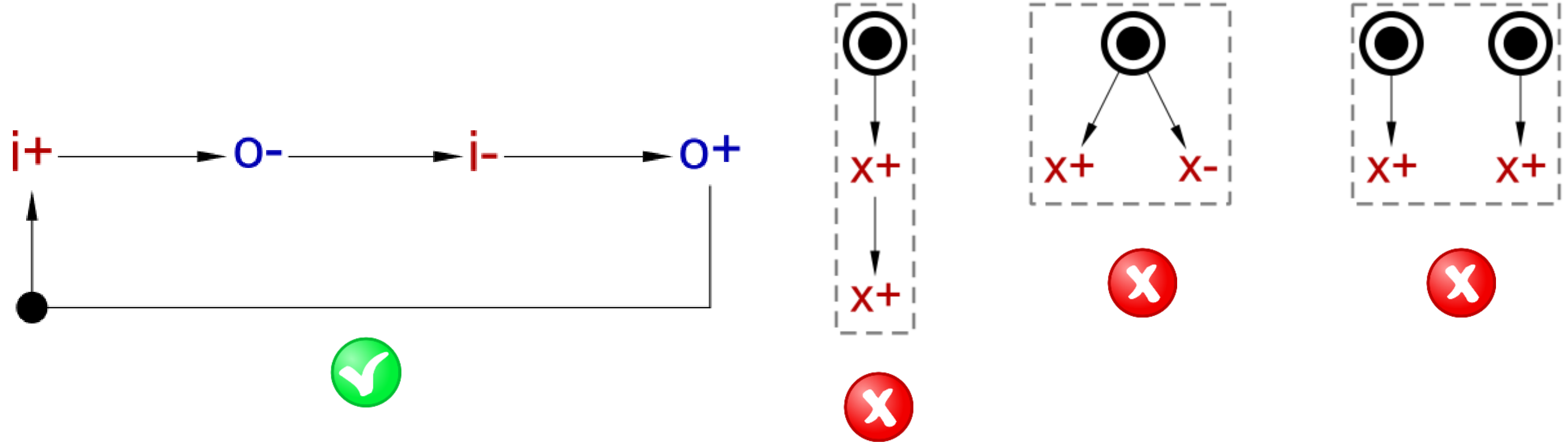
Standard PN properties:

- **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

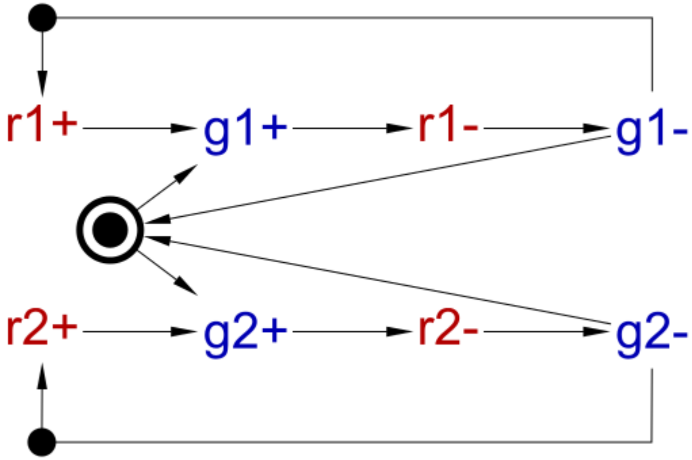


Verification of STG specification

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**

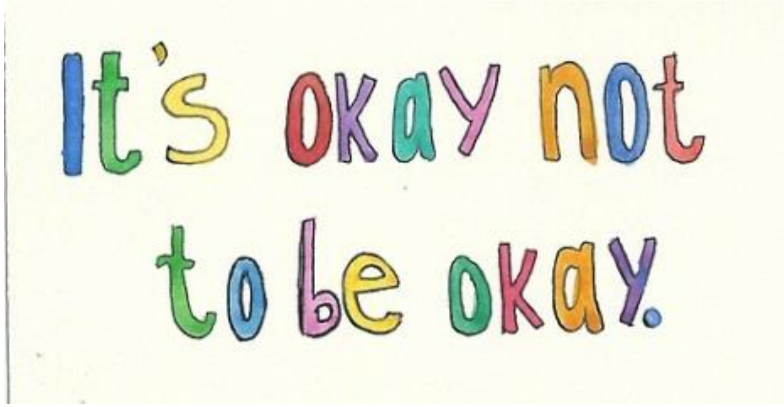


Verification of STG specification

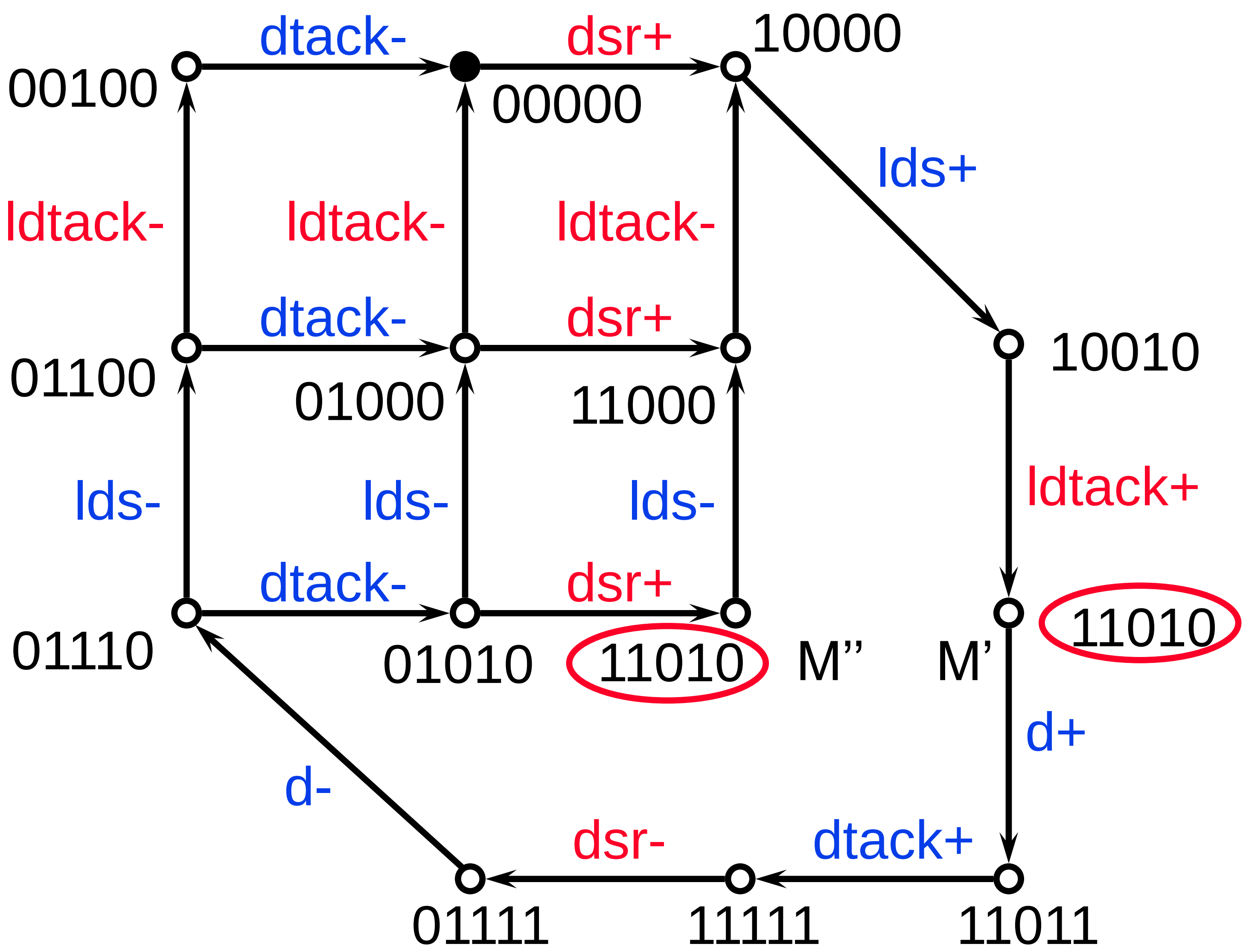
Complete State Coding (CSC): If two reachable states have the same values of all signals then they should enable the same outputs; two states violating this property are said to be in CSC conflict

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

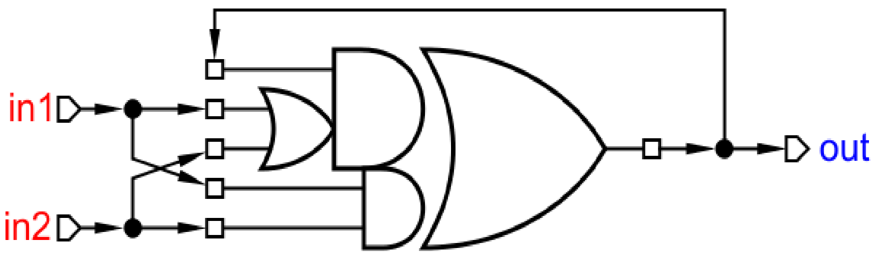


Verification of the circuit

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 $p_{\mathbf{i}=0} \rightarrow \mathbf{i+} \rightarrow p_{\mathbf{i}=1}$ and $p_{\mathbf{i}=1} \rightarrow \mathbf{i-} \rightarrow p_{\mathbf{i}=0}$
- For each output **o** with the next-state function $[\mathbf{o}]=E$, 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

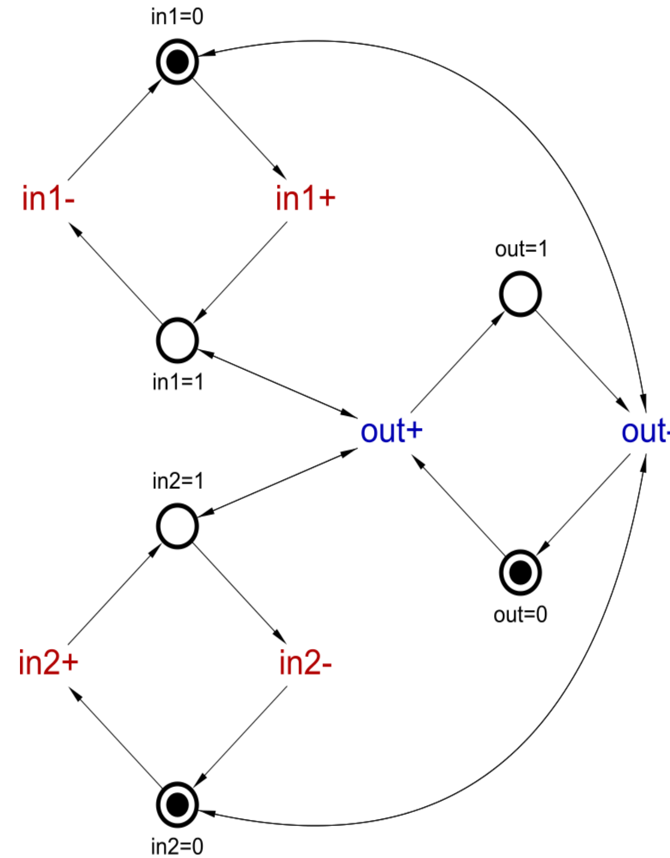


$$[out] = out \cdot (in1 + in2) + in1 \cdot in2$$

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

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

- 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

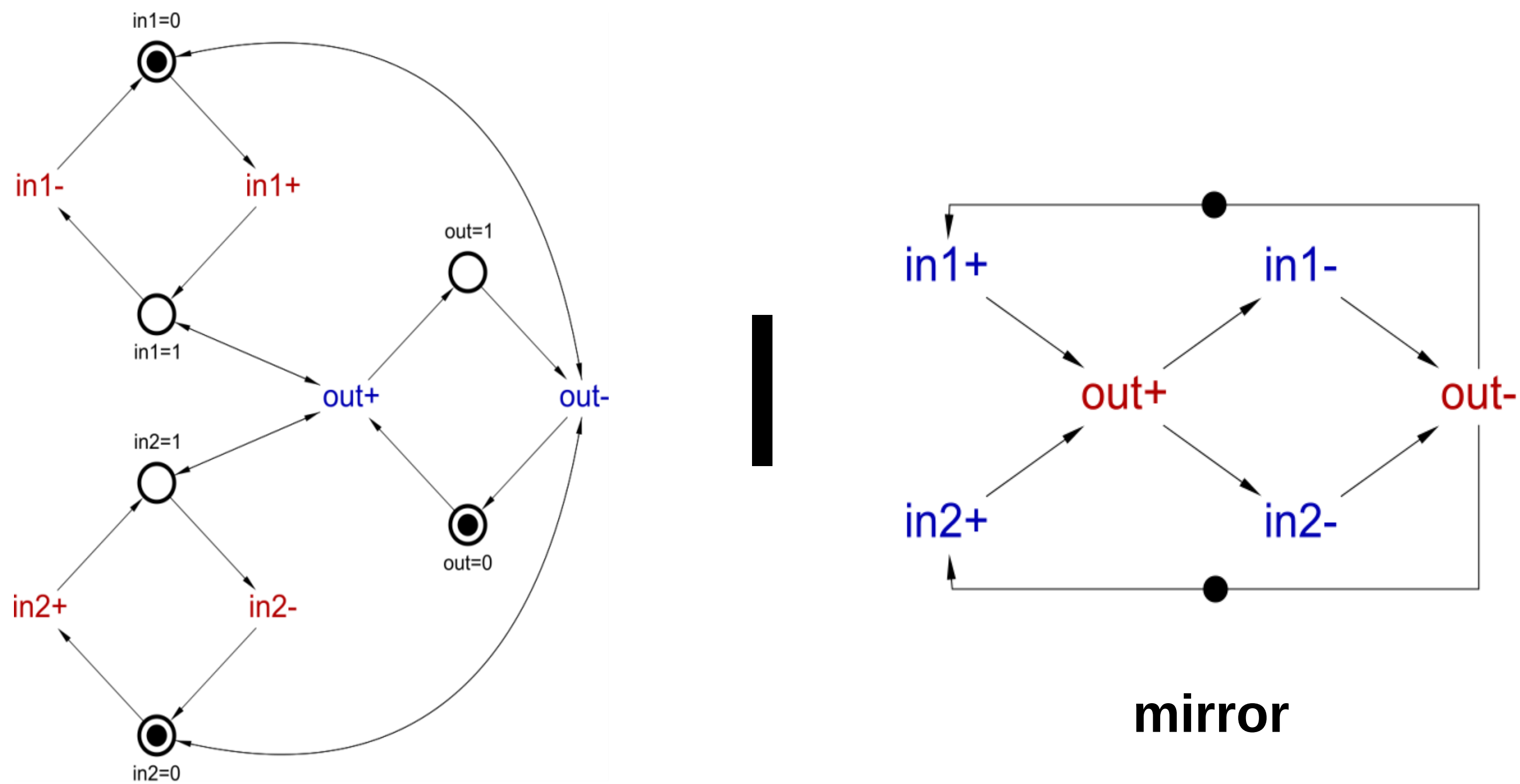
Gate-level circuit has no information about its environment, so naïve verification will always reveal hazards in any non-trivial circuit with inputs. Hence need to supply the environment's behaviour during verification: Assuming the environment fulfils the contract, the circuit must:

- 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

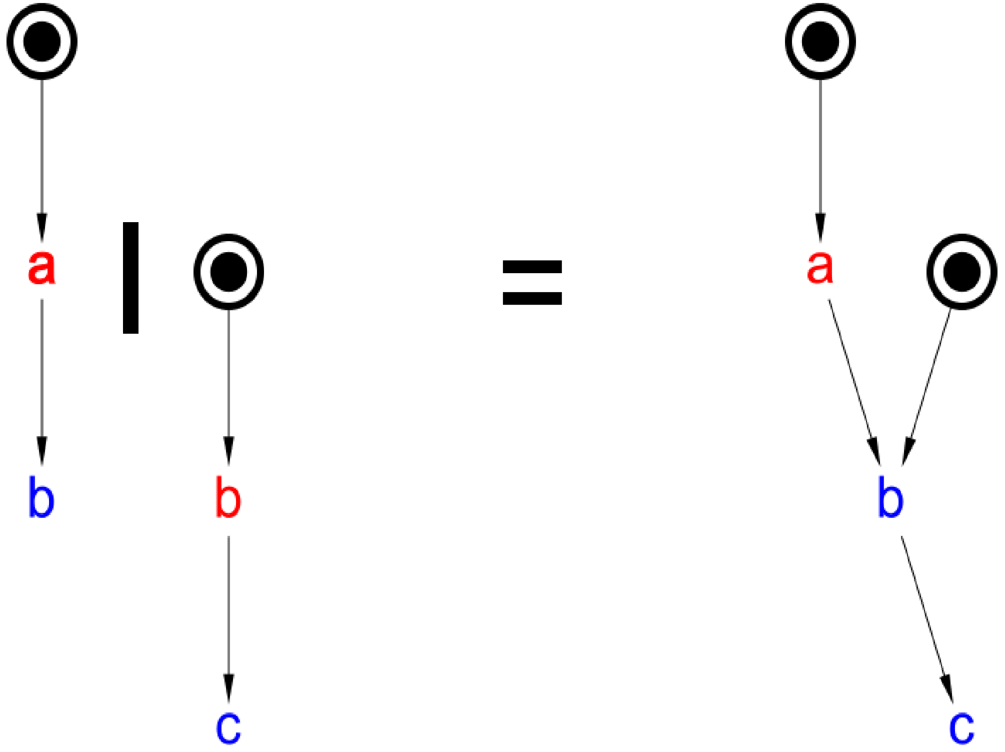
Problem: how to restrict the behaviour of the circuit by the behaviour of the environment to verify the properties?

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:



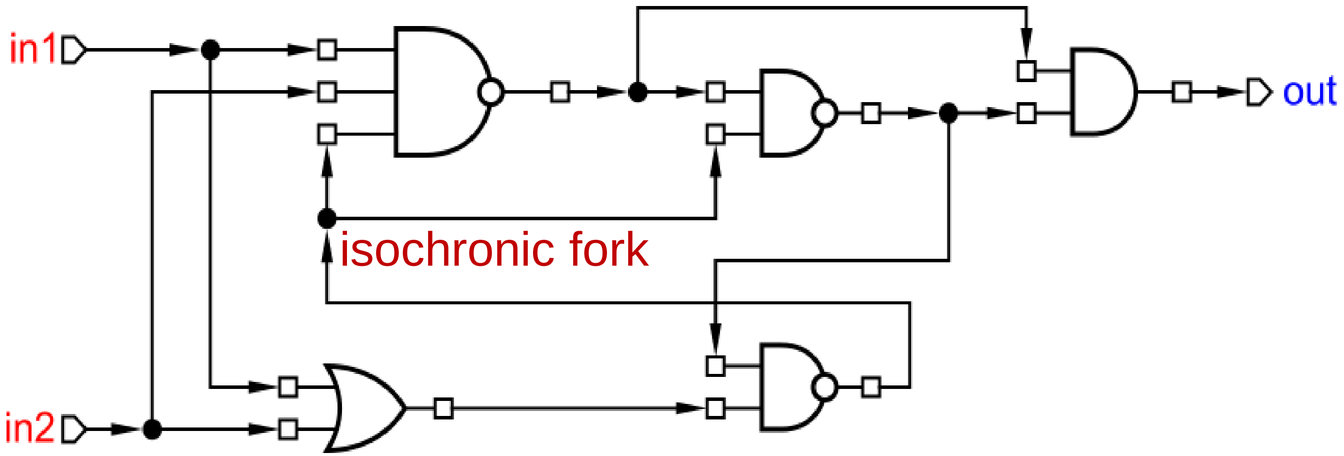
Parallel composition

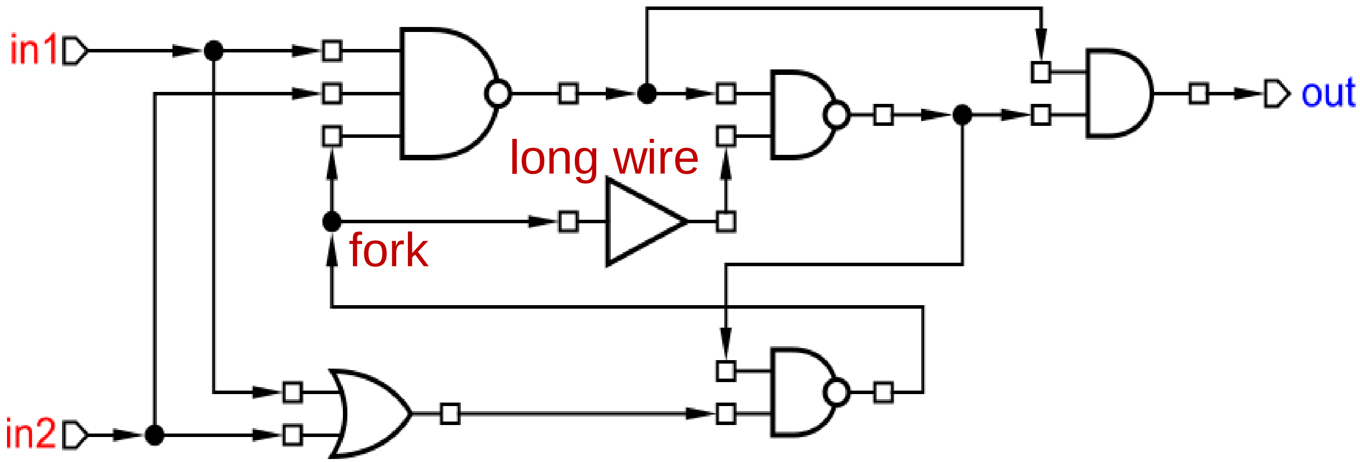
- **Idea:** **Fuse transitions from different STGs that have the same label (if STGs have several transitions with the same label, fuse each such transition in STG_1 with each such transition in STG_2)**
- **Example:**



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

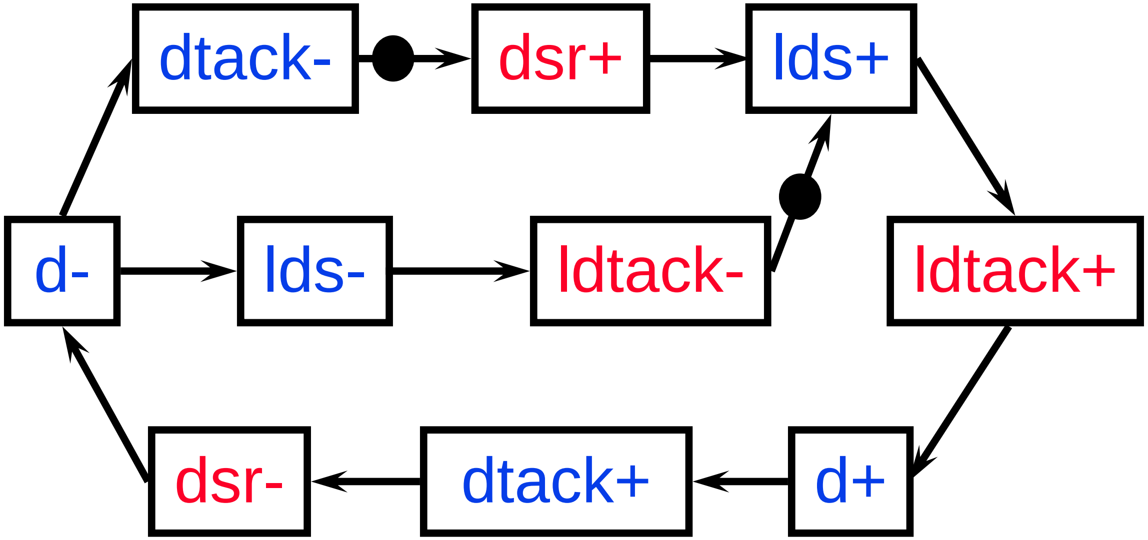
- **Uses PUNF-generated prefixes as an input – completely avoids state graph**
- **Employs a SAT solver for efficiency**
- **Verifies many relevant properties, like deadlocks, CSC, etc.**
- **Supports REACH – a language to specify custom properties**
- **Synthesis: CSC resolution, deriving complex-gate,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

PCOMP – parallel composition

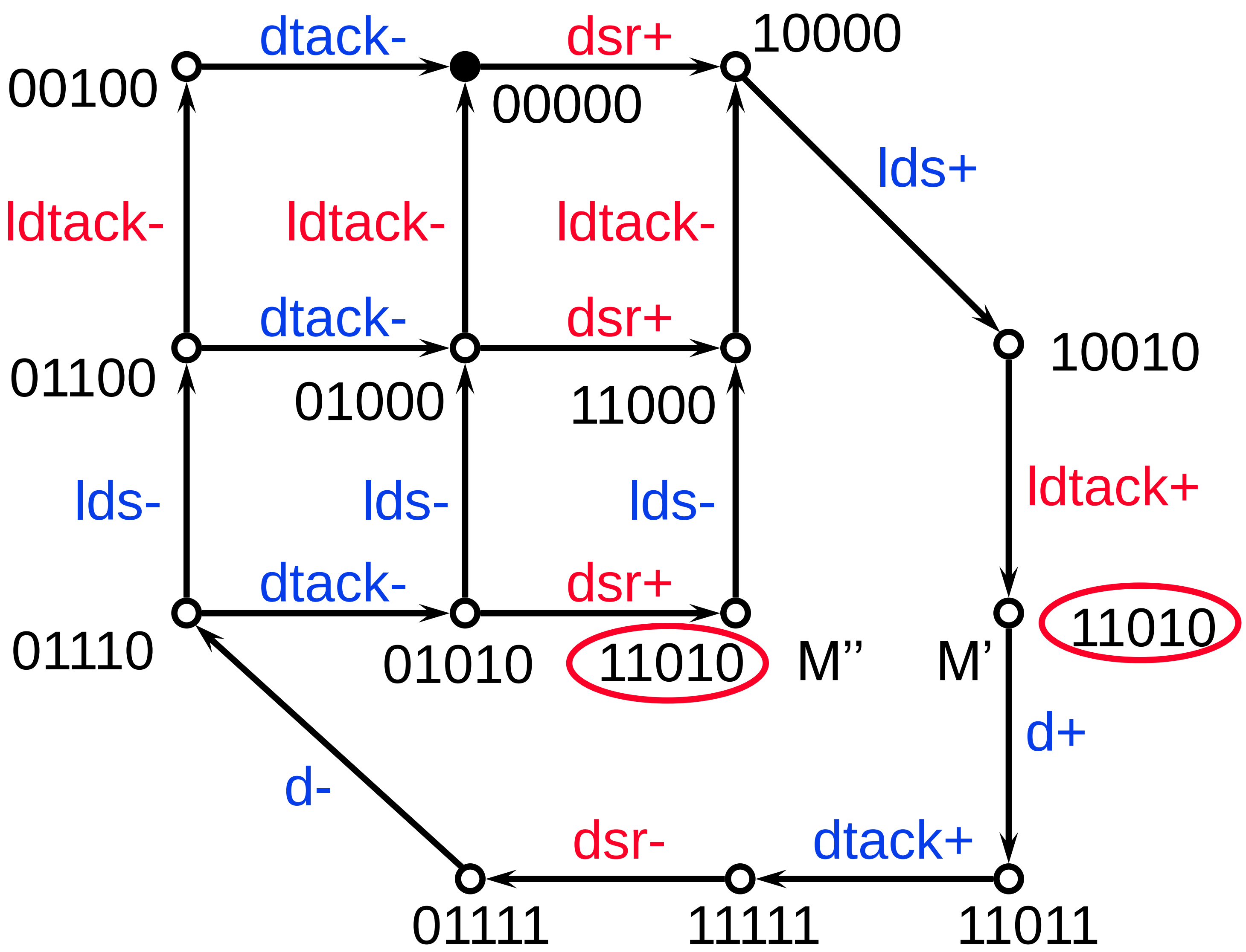
- Composes several STGs, optionally hiding the internal communication, e.g.:
  - to compose several modules into one
  - to compose a circuit with its environment for verification

CSC Conflict Resolution

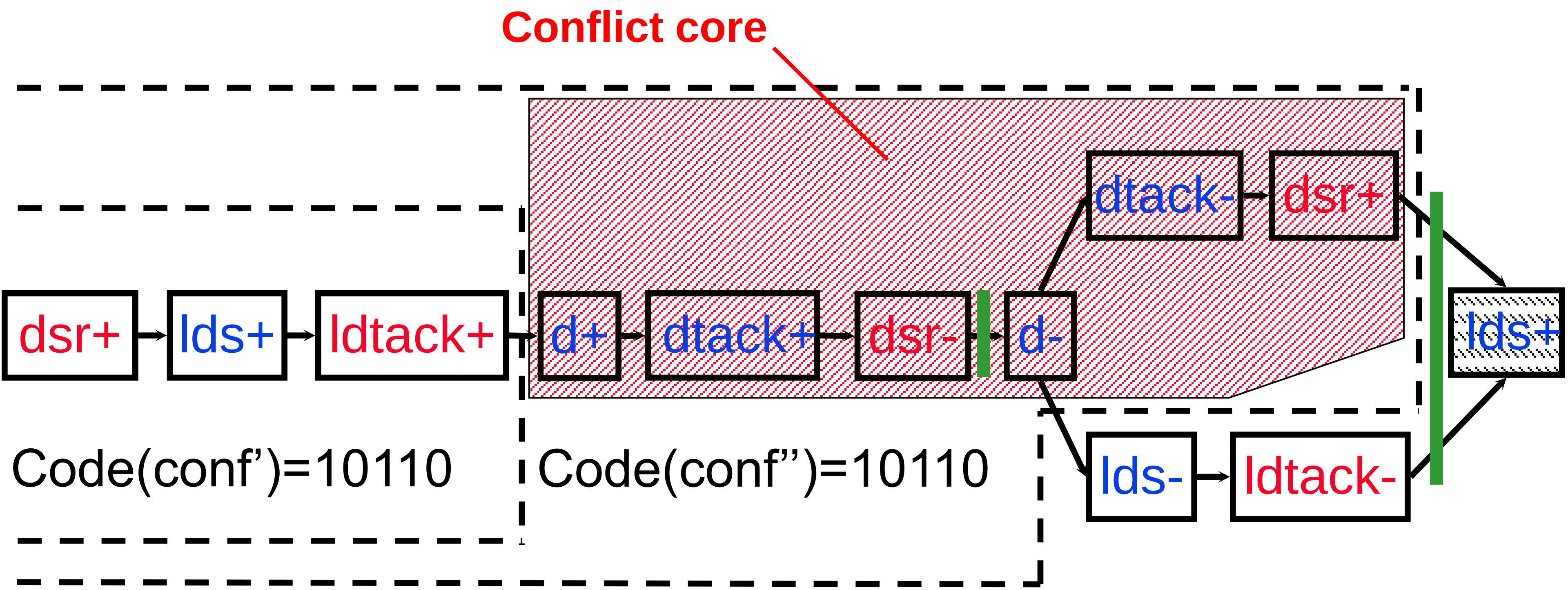
Example: VME Bus Controller



Example: CSC conflict



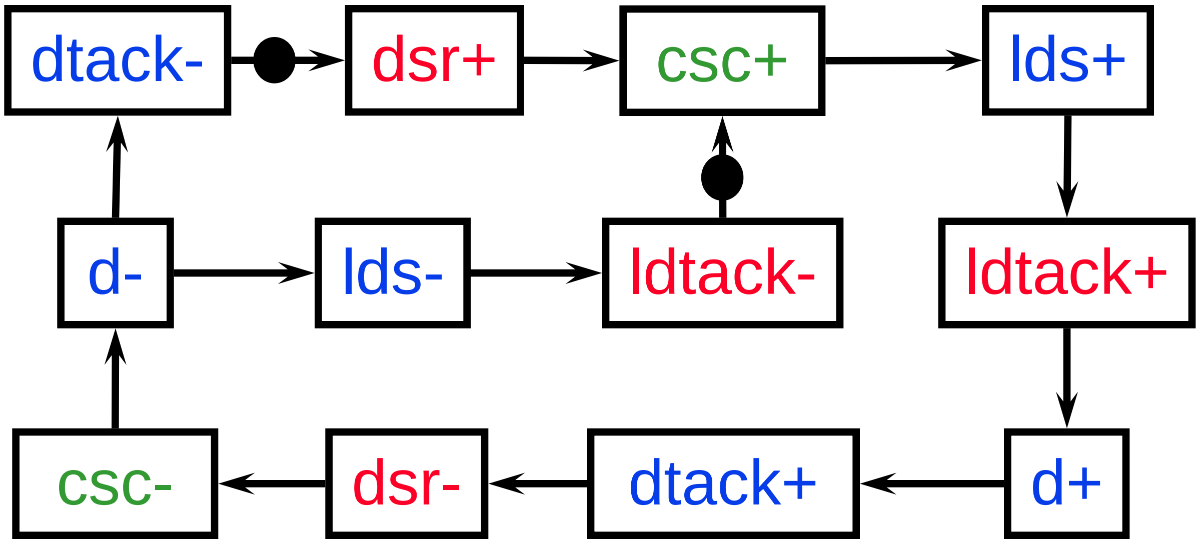
Example: Resolving the conflict



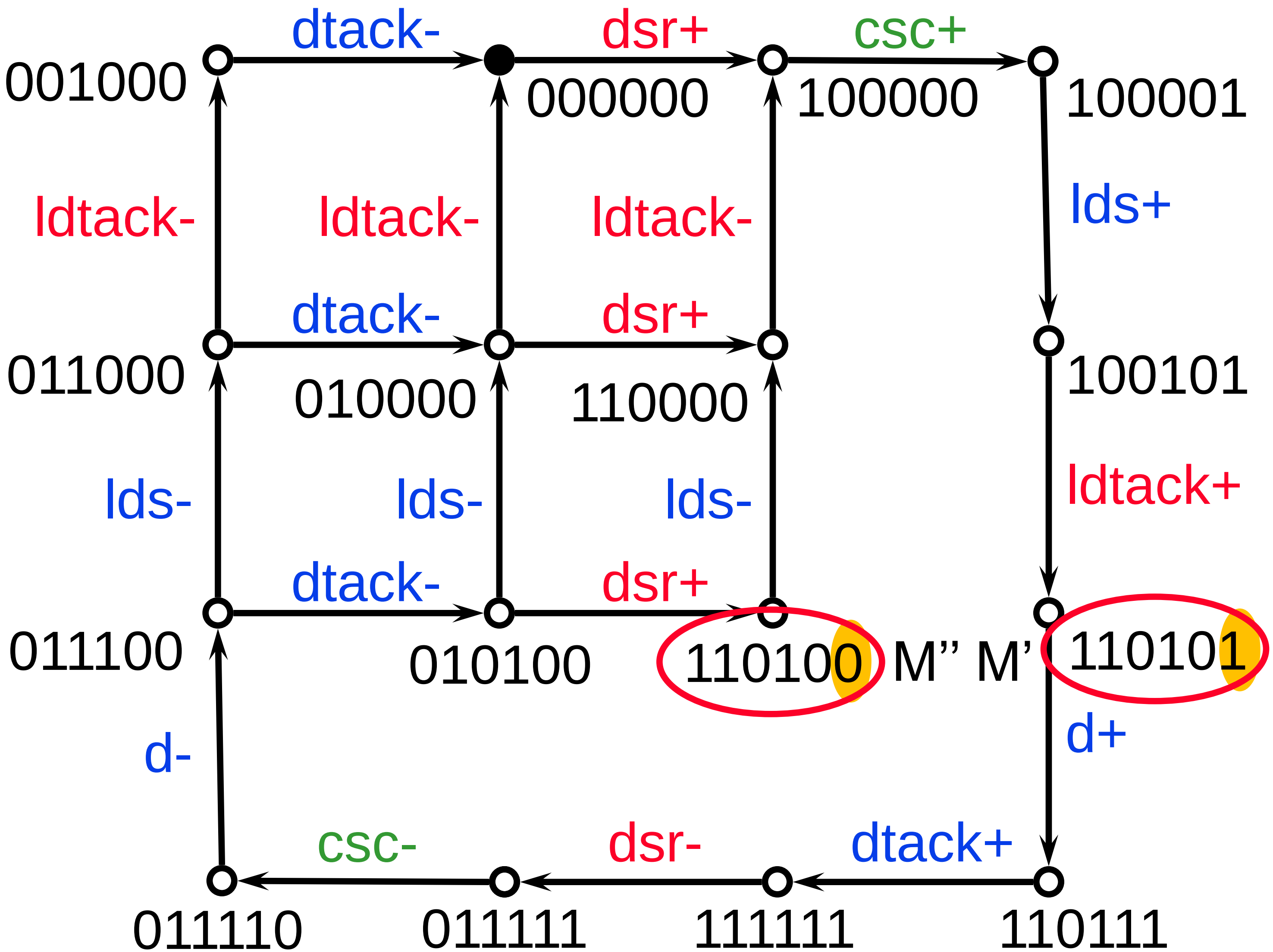
Idea: **Insert csc+ into the core and csc- outside the core to break the balance**

Note: **Cannot delay inputs!**

Example: Resolving the conflict

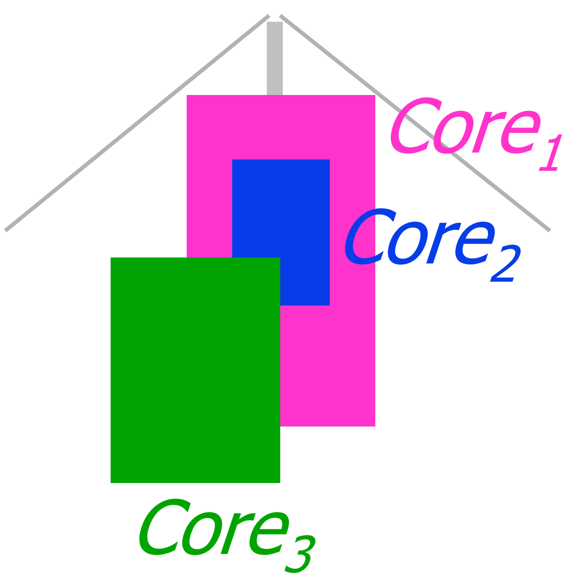


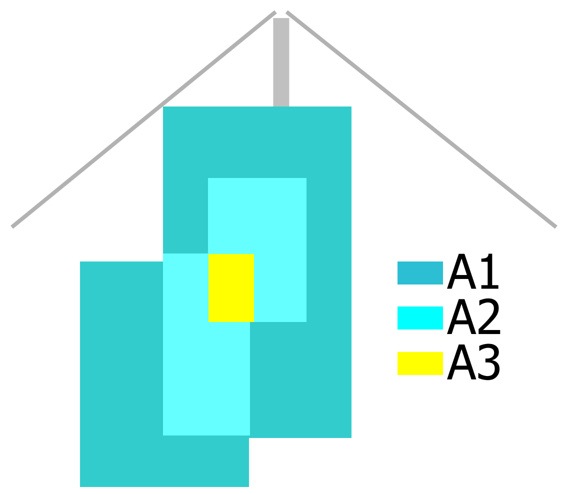
Example: Resolving the conflict

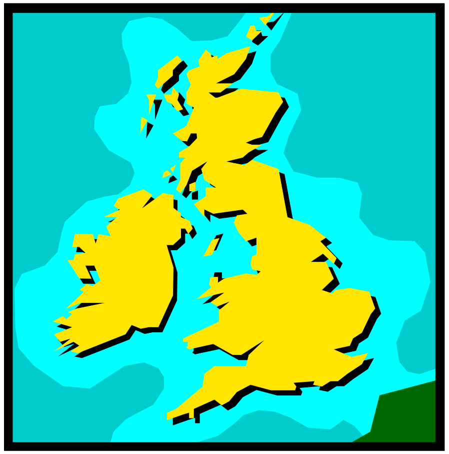


Core map

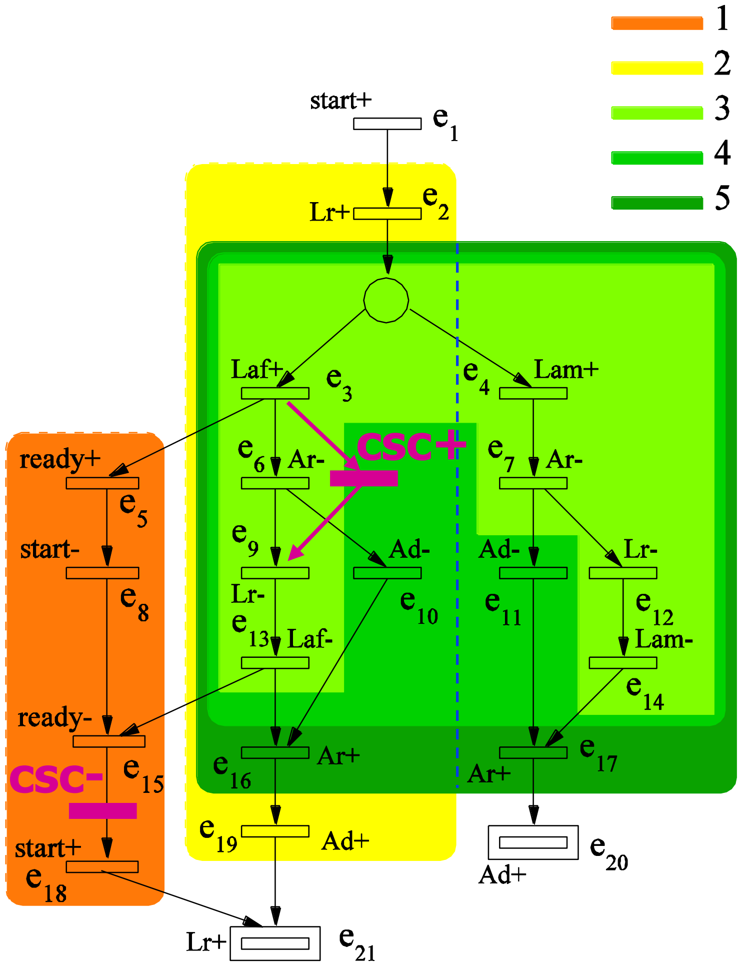
- **Cores often overlap**
- **High-density areas are good candidates for signal insertion**
- **Analogy with topographic maps**







Example: core map



Concurrency reduction

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

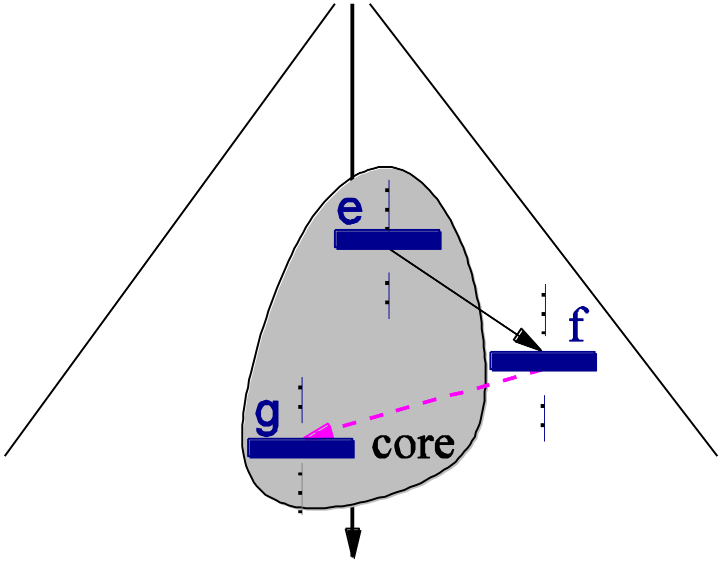
Note: Must not delay inputs, i.e. **b** cannot be an input!

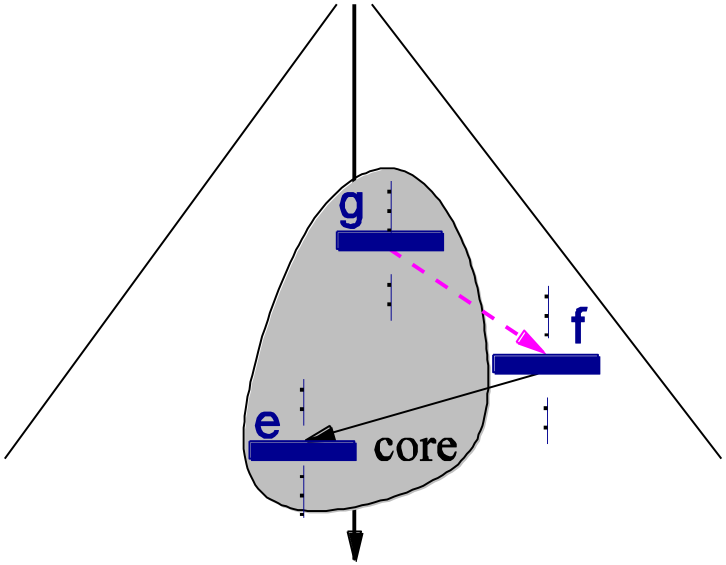
Note: Changes the behaviour, impacts the environment!

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

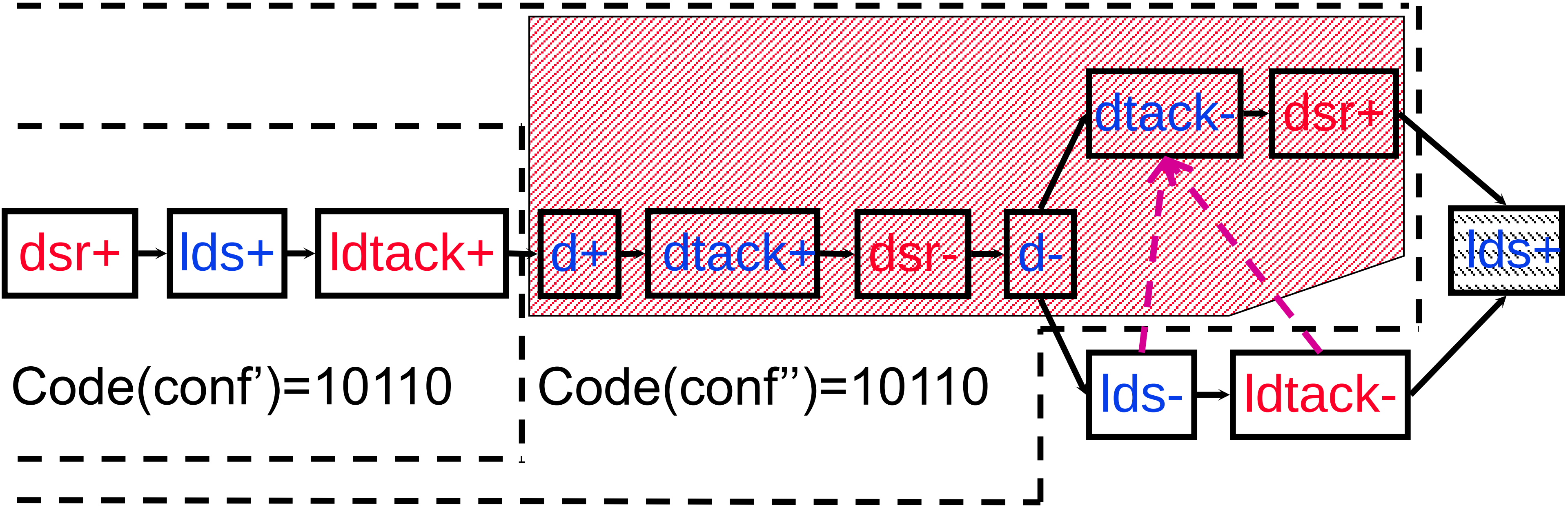
Used for resolving CSC conflicts and circuit simplification

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



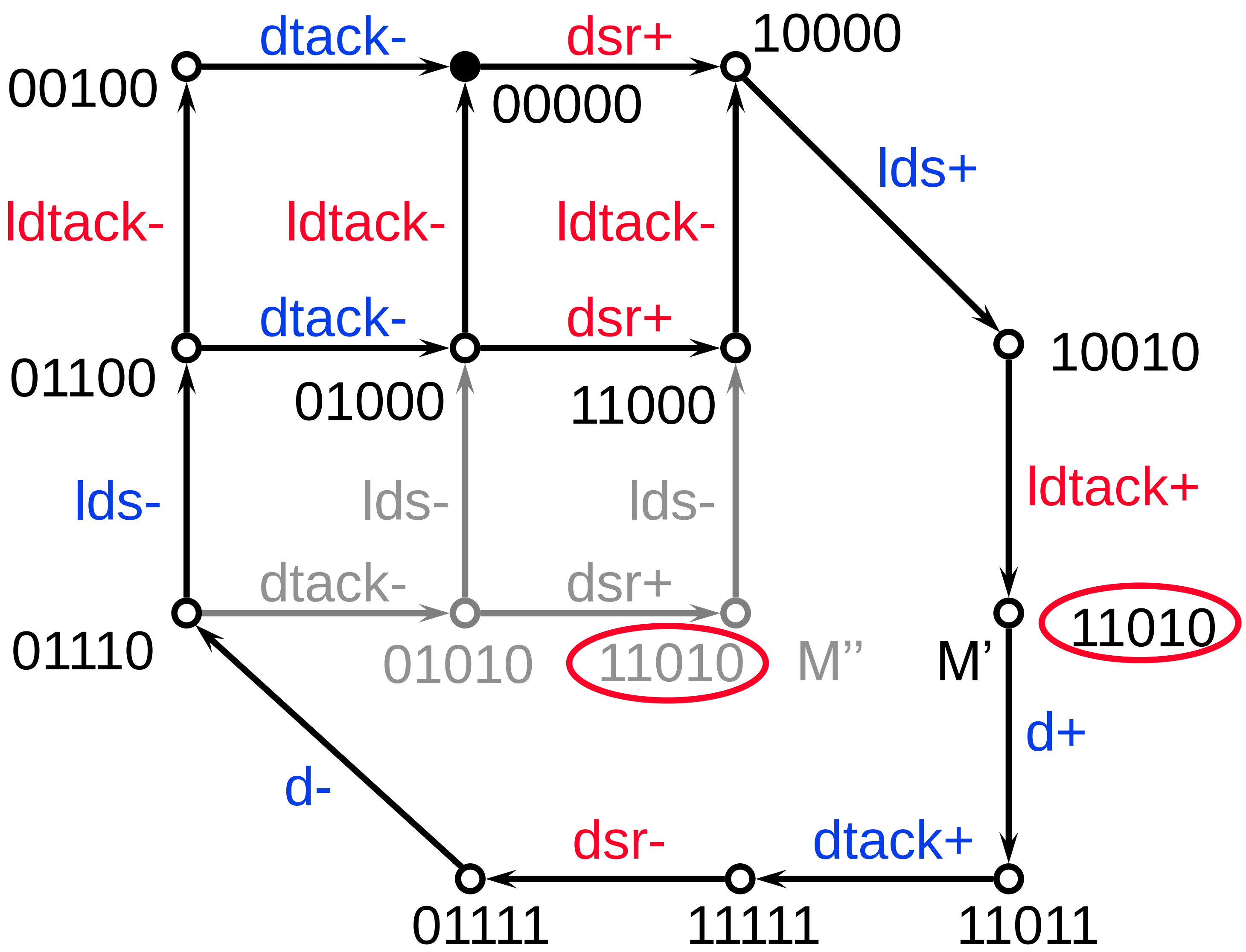


Example: Resolving the conflict



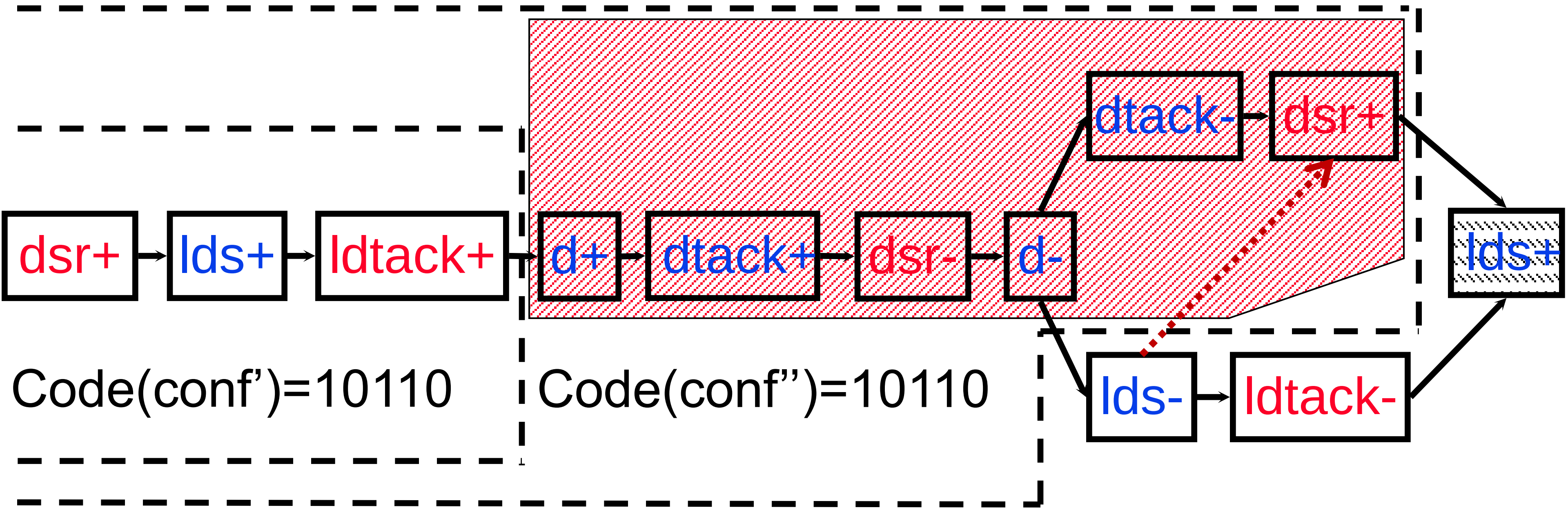
May be problematic!

Example: Resolving the conflict

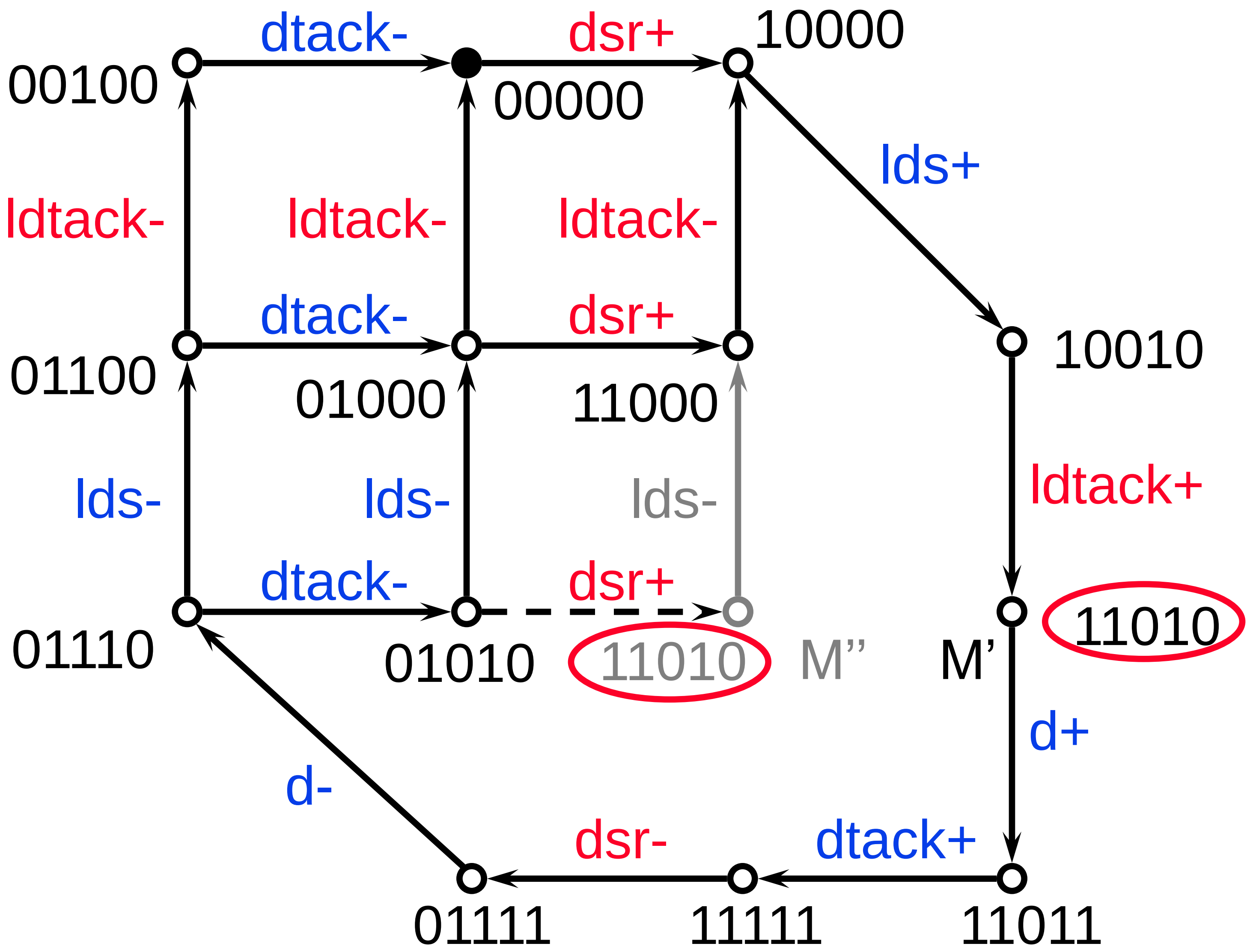


Relative timing assumptions

- "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



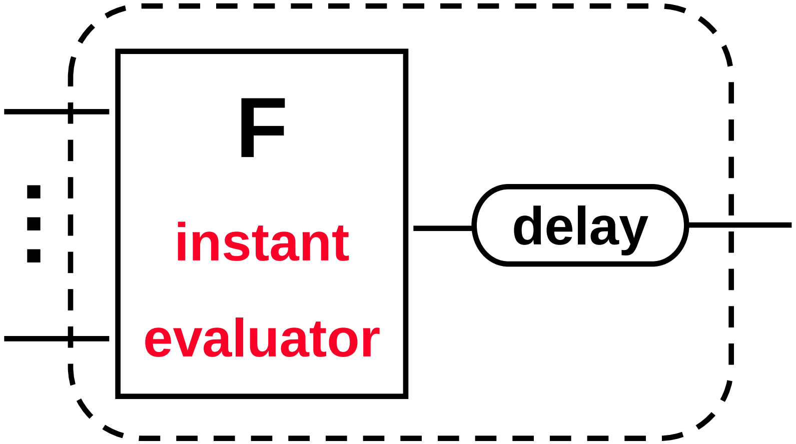
Comparison of the methods

- 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.)

Logic Synthesis and Implementation Styles in Asynchronous Circuits Design

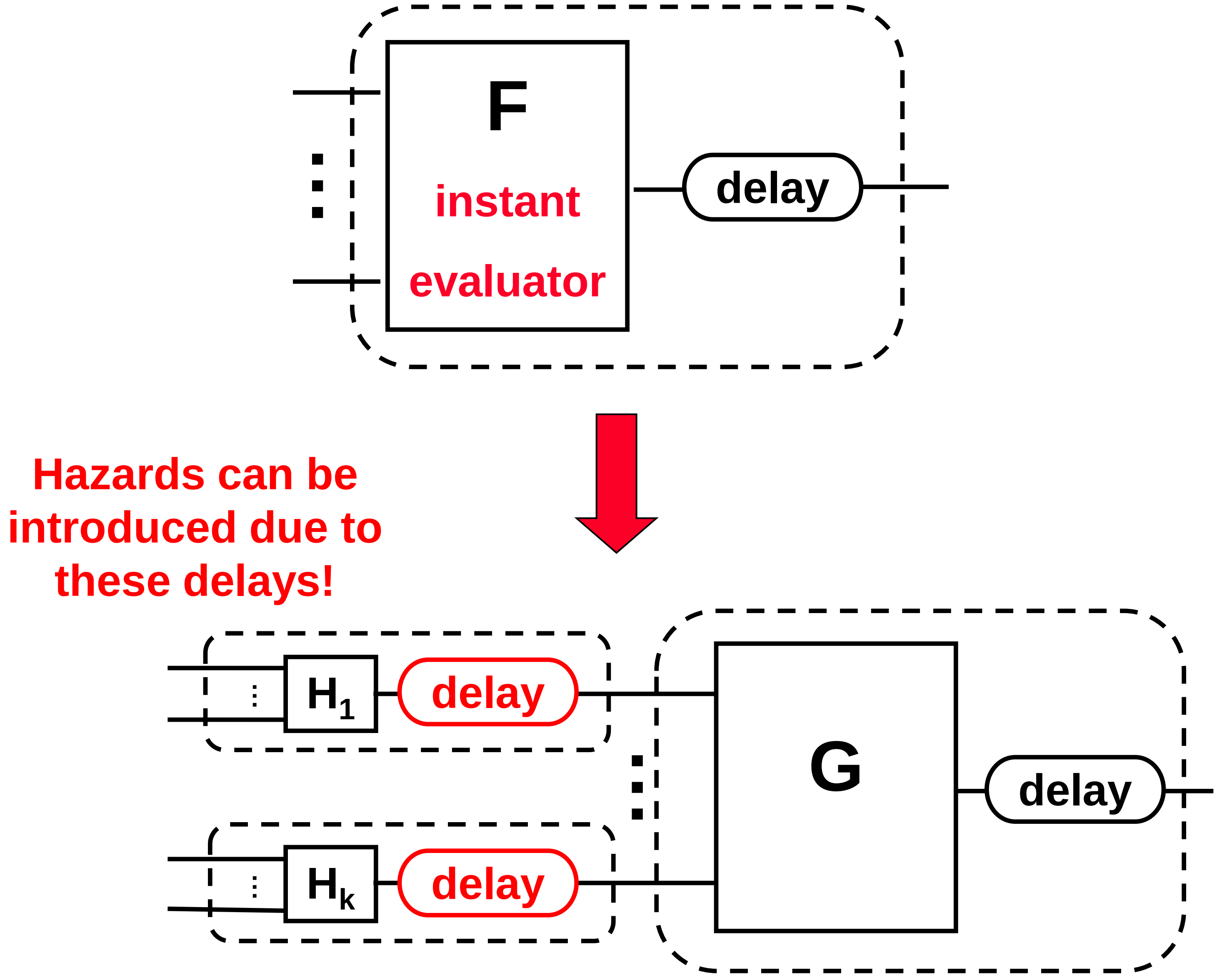
Speed-independence assumptions

- 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



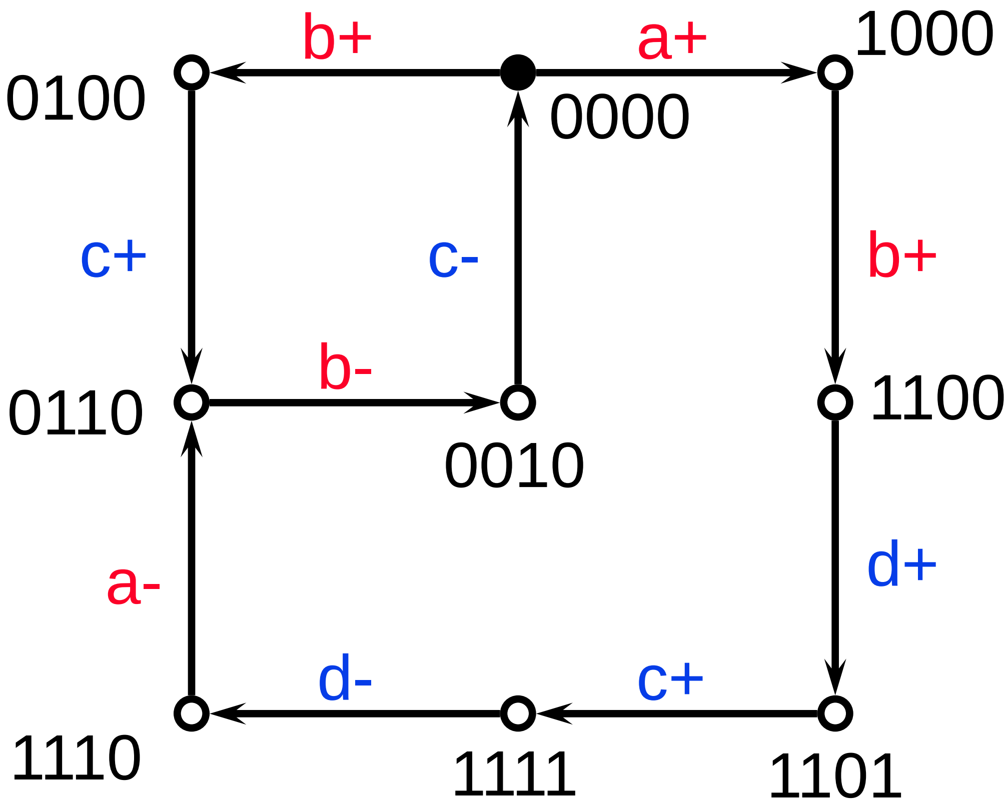
Gates & latches

- 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

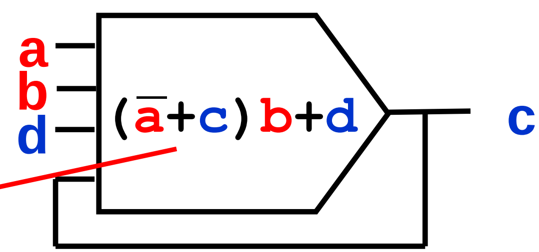
- Encoding (CSC) conflicts must be resolved first
- Several kinds of implementation can then be derived automatically:
  - complex-gate (CG)
  - generalised C-element (gC)
  - standard-C implementation (stdC)
- Can mix implementation styles on per-signal basis
- Logic decomposition may still be required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

Example: complex-gate synthesis



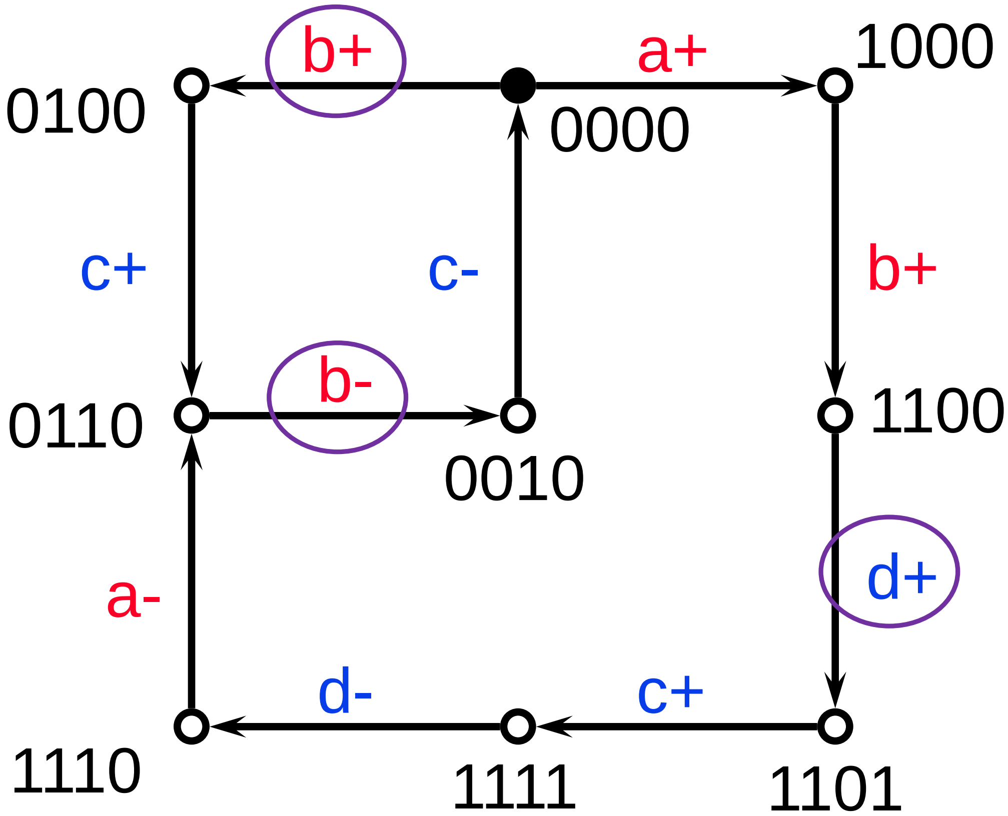
Code	Nxt_c
0100	1
0000	0
1000	0
0110	1
0010	0
1100	0
1110	1
1111	1
1101	1
else	-
Eqn	$(\bar{a}+c)b+d$

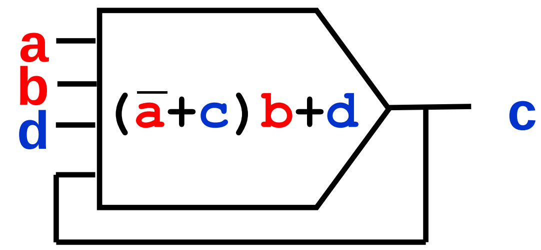
$$Nxt_z(s) = Code_z(s) \oplus Out_z(s)$$



The size of this Boolean expression is not limited!

Support, triggers and context





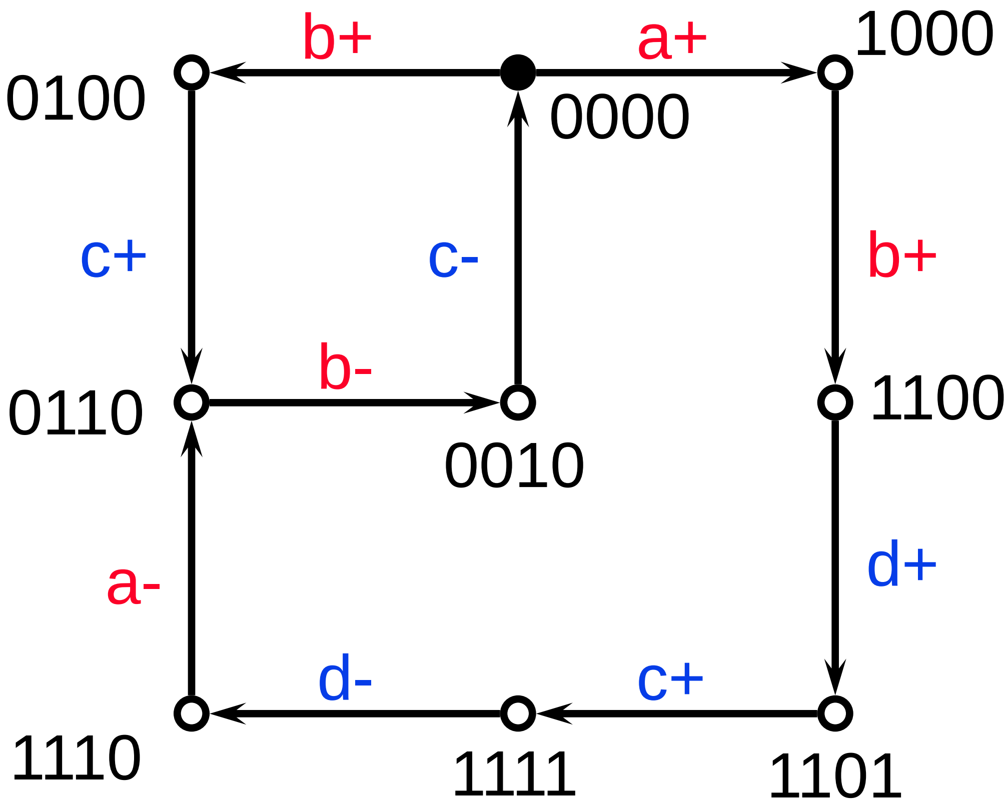
Signals that are the inputs of the gate producing a signal form its **support**, e.g. the support of c is {a,b,c,d}. Supports are not unique in general.

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.

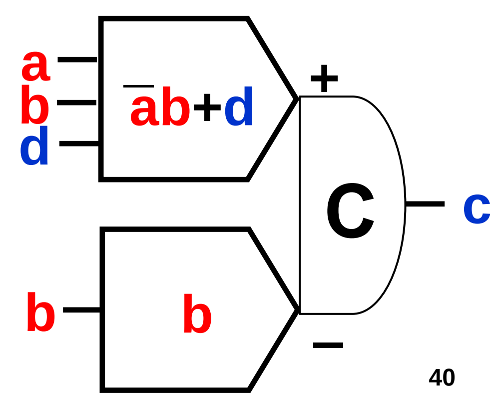
support = triggers + context

Example: gC implementation



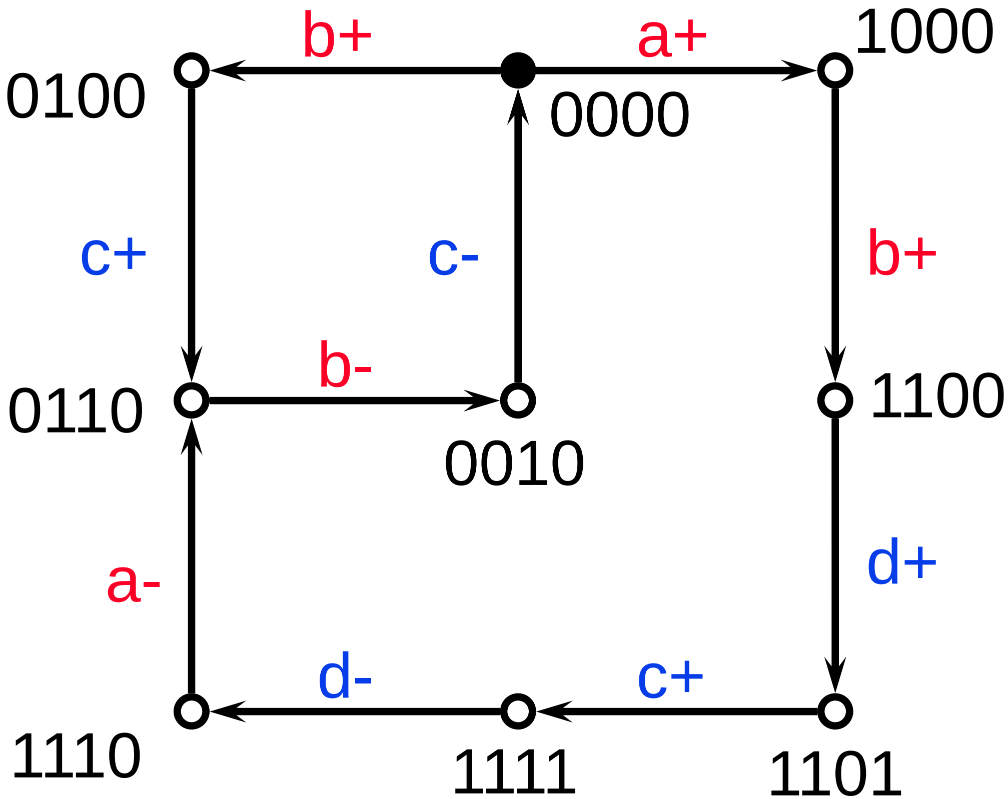
Code	Set_c	$Reset_c$
0100	1	0
0000	0	-
1000	0	-
0110	-	0
0010	0	1
1100	0	-
1110	-	0
1111	-	0
1101	1	0
else	-	-
Eqn	$\bar{a}b+d$	$\bar{b}$

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

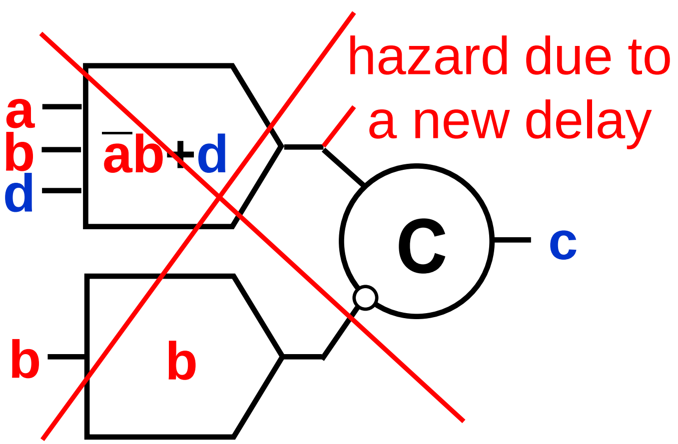


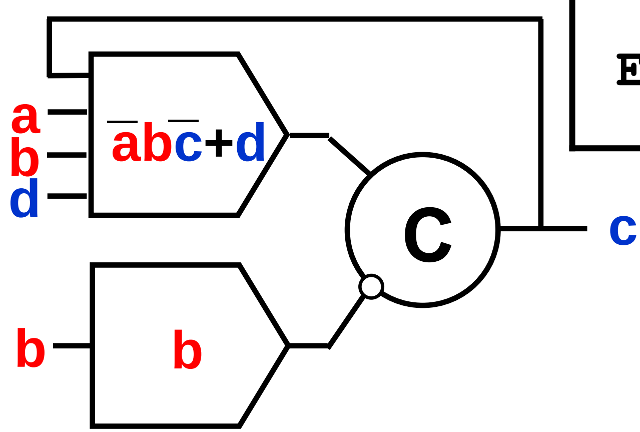
Implemented as pull-up and pull-down networks of transistors and a ‘keeper’; assumed to be **atomic**

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’