The Systematic Design of Asynchronous Circuits

Michael Kishinevsky
The University of Aizu, Japan

Luciano Lavagno
Politecnico di Torino, Italy
Cadence Berkeley Labs, USA

Peter Vanbekbergen
Synopsys, Inc., USA

Outline

- Introduction and basics
  What is asynchronous, advantages and disadvantages, design styles, hazards and races, control/data dichotomy
- Specification and synthesis methodologies
  Event based models, concurrent languages, FSMs
- Synthesis from Signal Transition Graphs
  Synthesis flow, state assignment, logic synthesis, design-for-testability, comparison with synchronous
- Verification and validation of asynchronous designs
  Formal verification, checking properties, validation in VHDL environment
- Practical applications
  Interface design, asynchronous processors, DSP circuits, CAD tools

Motivation

- Reliable design methodology for asynchronous is coming
- Asynchronous design sometimes unavoidable (asynchronous interfaces, arbiters, etc.)
- Asynchronous is getting more important:

"[In the microprocessor of the year 2000] we will see perhaps even asynchronous design..."

"Looking at the future"
Anant Agrawal, Vice President,
Engineering Sun Microsystems
SPARC Technology Business

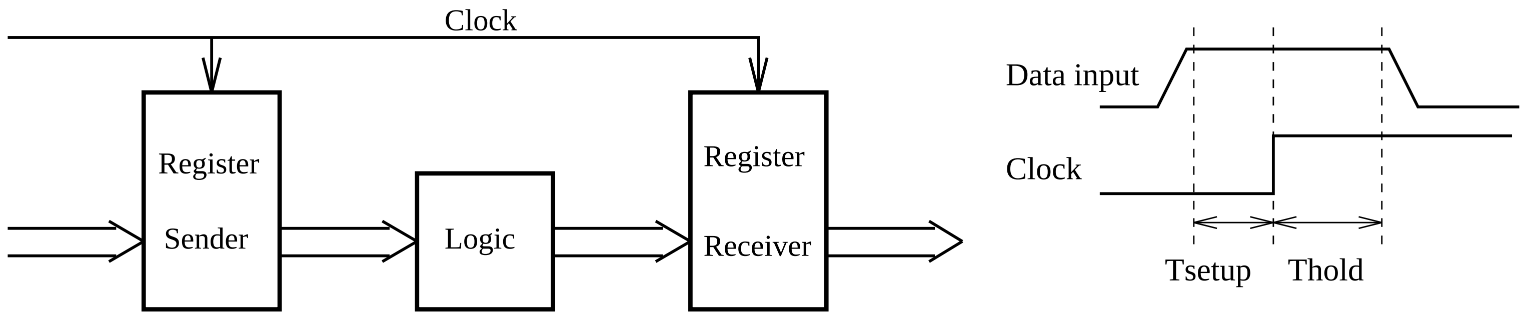
What is asynchronous?

Asynchronous = not synchronous $\Rightarrow$

- Synchronous: use an *external global* **clock** for observing system states
- Asynchronous: use *internal and external* **events** for observing system states

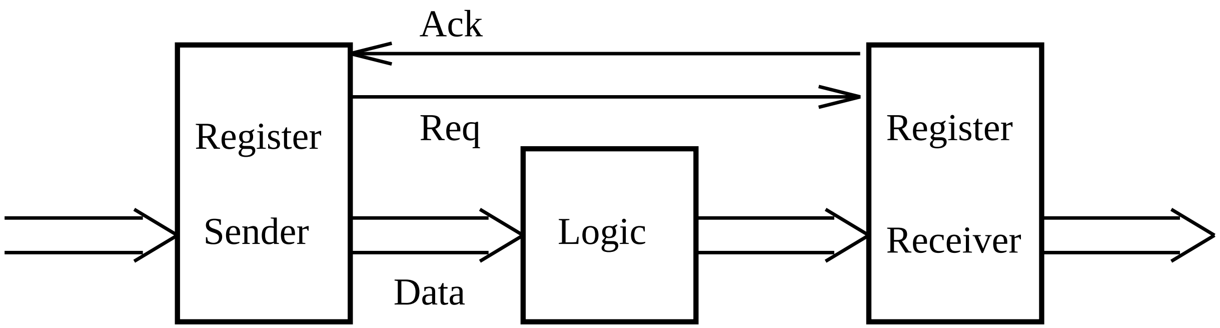
Asynchronous are also called self-timed

Synchronous communication



Timing constraint: input data must stay unchanged within a setup/hold window around a clock event

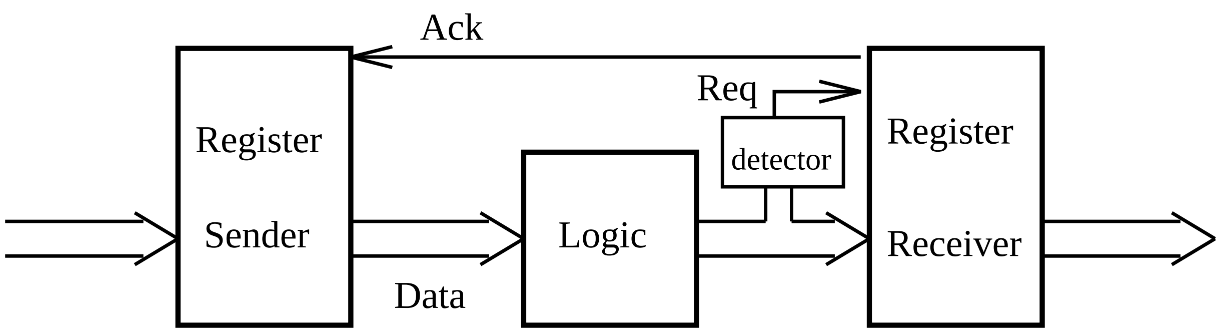
Asynchronous communication



- A signal *handshake* is used instead of a global clock

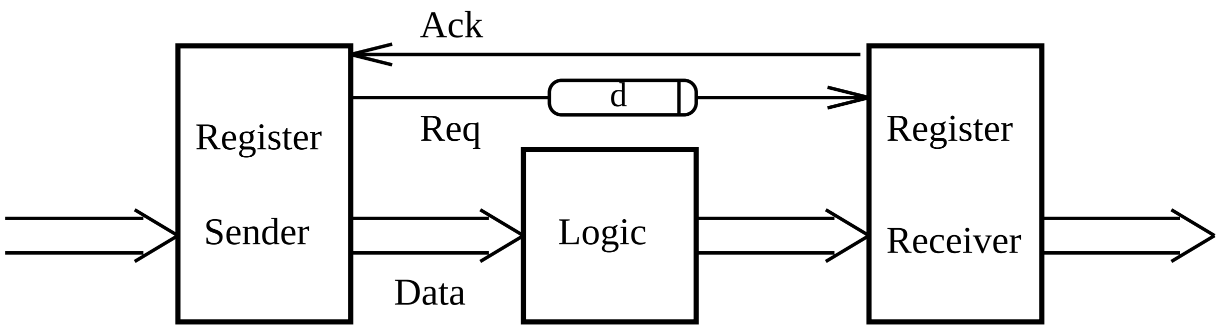
- Handshake can be implemented using *delay padding* or/and *completion detection*

Completion detection



- Special redundant data encoding ⇒ area overhead
- Completion detection forms a request ⇒ more area overhead
- Completion detection can be done inside a receiver
- Tolerates data skew

Delay padding



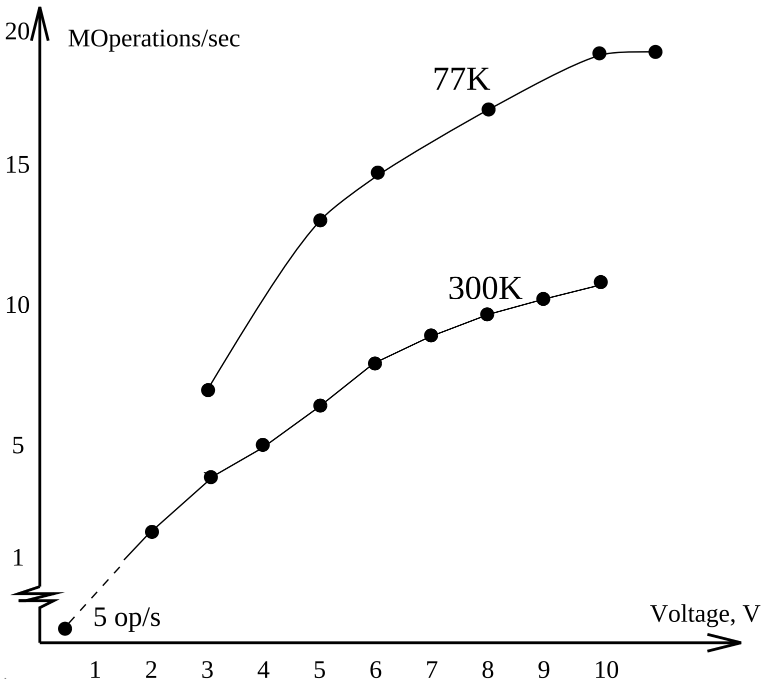
- Similar to synchronous
- Measure or estimate maximum logic delay
- Pad extra delay (if Req sets before $Logic$ stabilizes) for mimicking completion detection
- Time redundancy – timing margins are required

Potential advantages

Top ten lists at Async94 from Al Davis

- No clock distribution
- Average case performance
- Low power consumption
- Robustness to parameter variations
- Modular composition
- Metastability
  either does not occur or has time to resolve
- Global synchrony does not exist anyway
  (hardware/software codesign)

Robustness



- Alain Martin 1989
- Christian Nielsen 1994

Operating range:
5 op/s under 0.5V and 300K ↔ 19 Mop/s under 11V and 77K

Problems

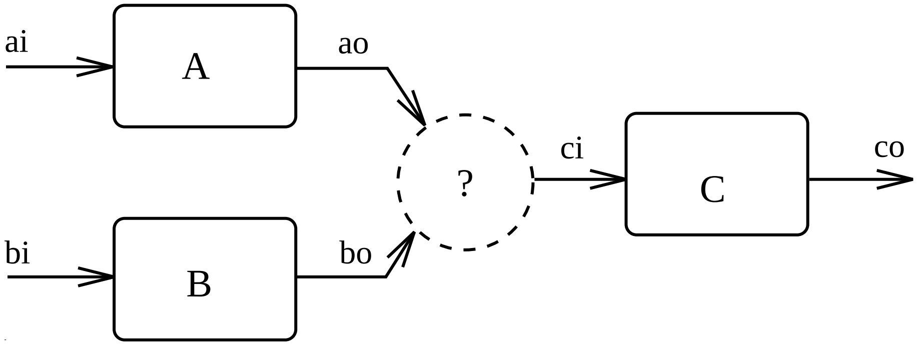
- Have you ever seen asynchronous circuits?
- Area overhead
- Testability?
- How to design them?

A systematic design methodology is needed

Part 1: Basics

- Example: C-element design
- Completion detection
- Design styles
  - delay models
  - signalling methods
  - models of environment
- Hazards and races: review
- Asynchronous data path

Synchronization of parallel processes

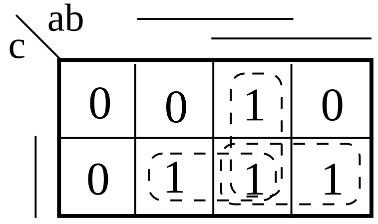


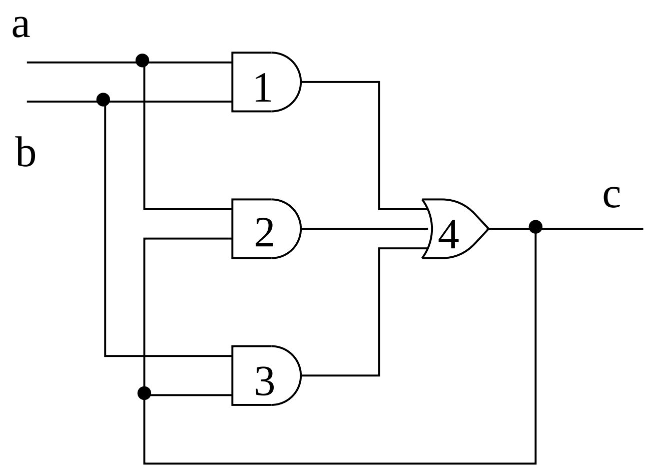
- After A and B do C.
- Both rising and falling transitions have to be synchronized:

  if $a = b = 1$ then $c := 1$
  if $a = b = 0$ then $c := 0$
  otherwise $c := c$
- AND-gate synchronizes only rising transitions

Synchronization can be done with a *C-element*

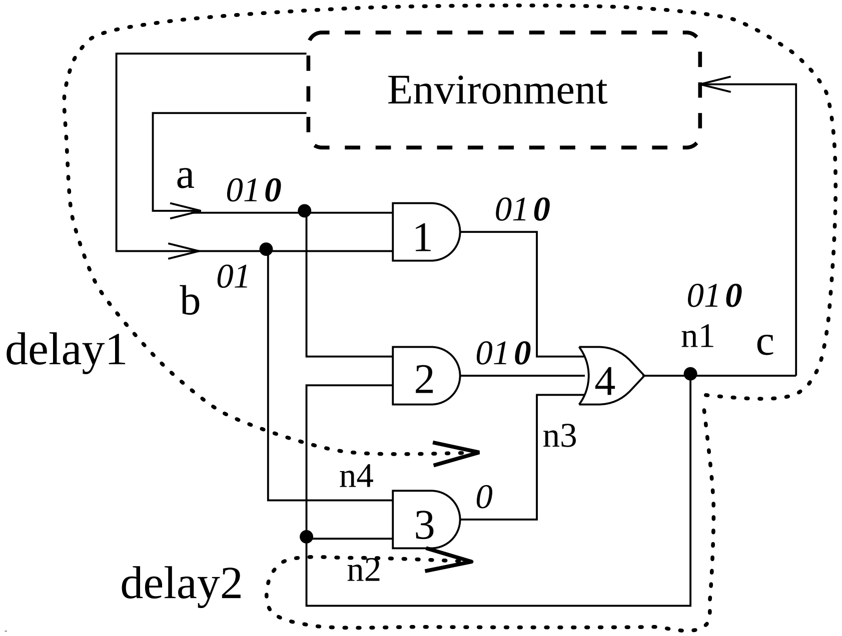
C-element on basic gates





Is this implementation correct?

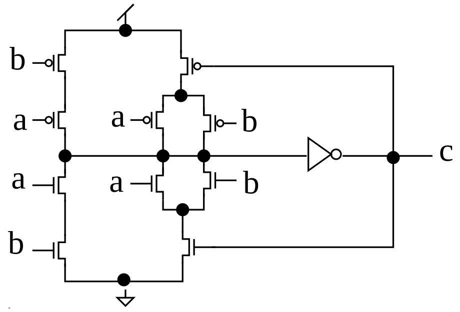
Delay assumptions



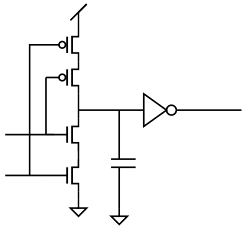
- *Delay 1* is a propagation delay from node 1 via environment to node 3
- *Delay 2* is a propagation delay from node 1 via internal feedback to node 3
- *Delay 1* must be greater than *Delay 2*

Delay assumptions influence correctness of asynchronous design

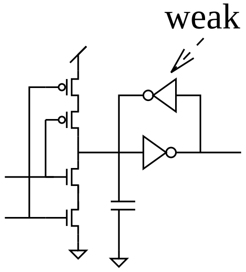
CMOS implementation of a C-element



Static



Dynamic

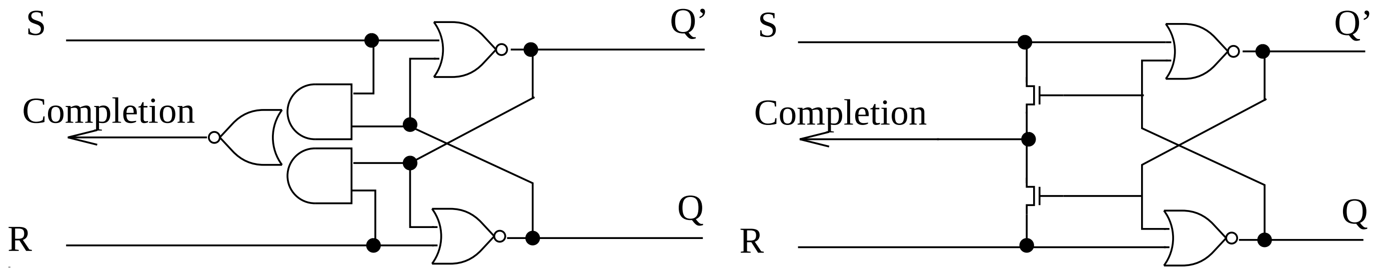


Quasi-static

Are these implementations correct?

Careful design is required (layout, transistor sizing, charge sharing)

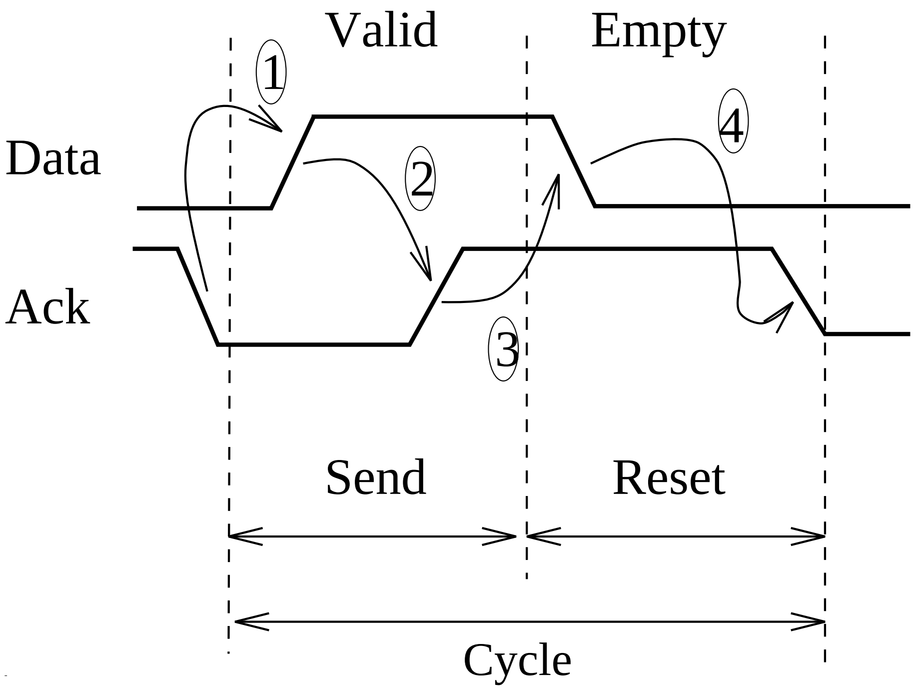
RS-latch with completion detection



$$Completion = 0 \Leftrightarrow < S, R > = < Q, Q' >$$

[Gillies,Swartwout'61 (in Miller '65), Martin, van Berkel 80's]

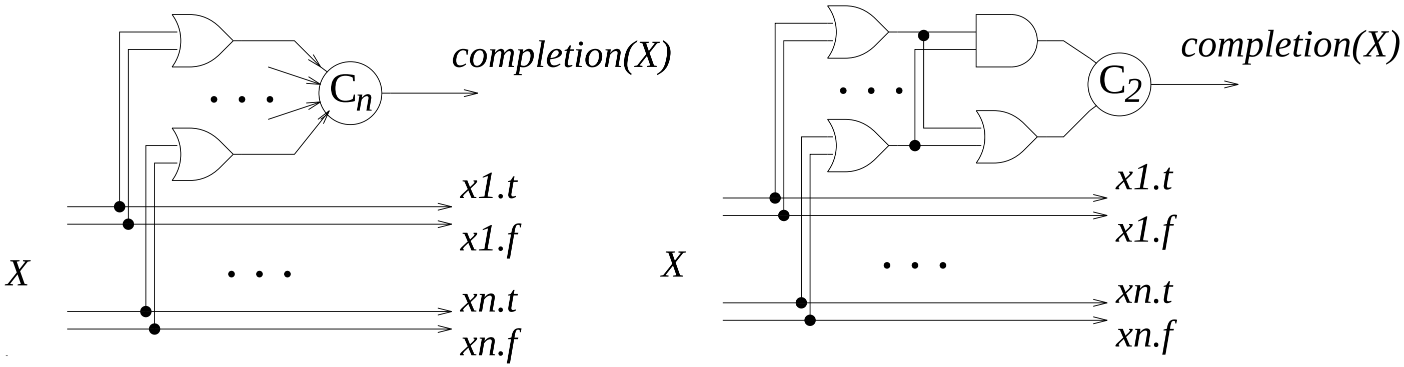
Dual-rail four-phase encoding



X	X.t	X.f
True	1	0
False	0	1
Empty	0	0

Valid = { True, False }

Completion detection



Completion detectors for dual-rail four-phase code

[Armstrong et al. 69] [Varshavsky 75]

Completion detection vs delay padding

- Completion detection
  - Completion detection is well developed
  - Requires an area overhead and an extra-delay.
    Not practical for wide data paths
- Bundled data
  - Padding delay allows to mimic the completion detection
  - Data path is synchronous
  - Requires timing redundancy for safe design

What influences the design methodology

- Delay model
- Signalling model
- Environment model
- Specification language

Delay models

- Place: gates, wires, gates + wires

- Timing properties: bounded and unbounded

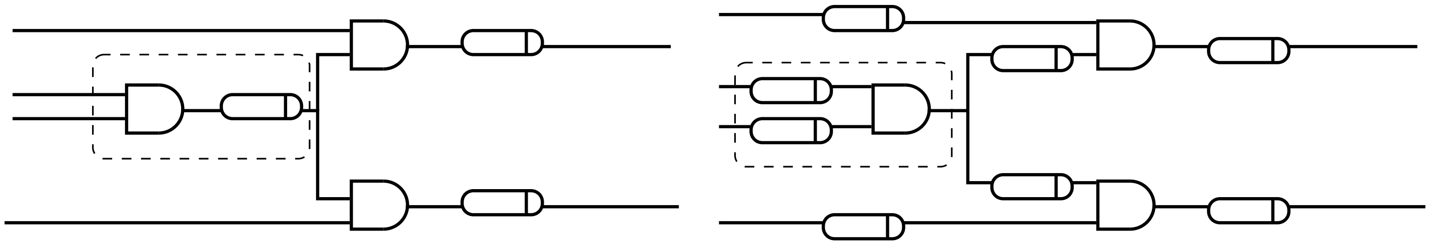
  Bounded = lower and upper bounds for circuit components are known
  Unbounded = delays are unknown but finite

- Memory properties: inertial and pure

  Apply a train of short pulses to the input of a delay element
  before the output of a delay has changed its value.

  Inertial = all pulses are filtered out
  Pure = no pulses are filtered out
  Intermediate models are possible

Gate vs wire delay models

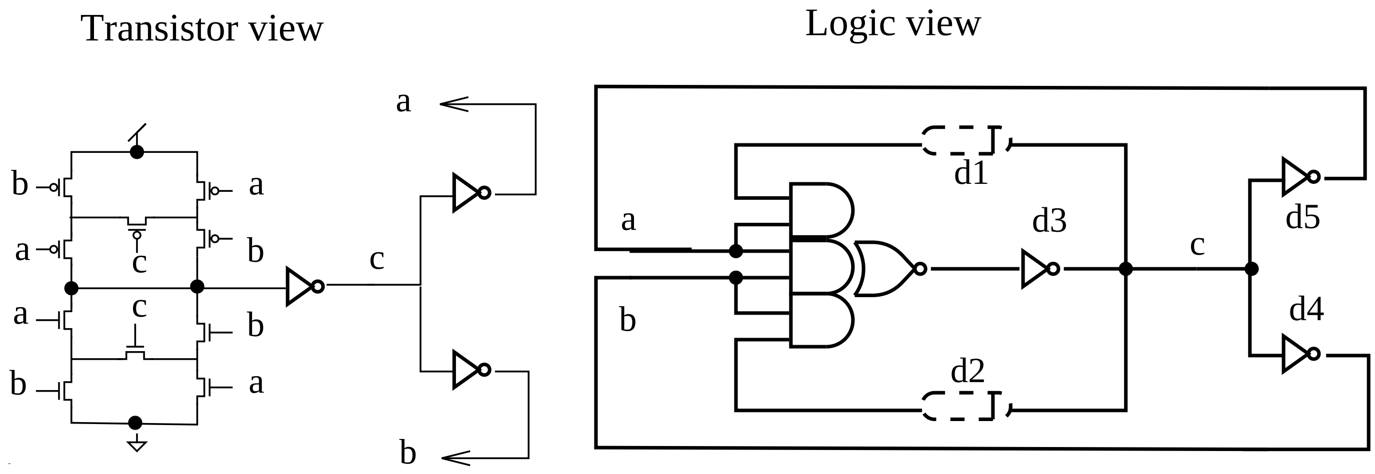


- Gate delay model = delays are in gates, no delays in wires

  More realistic version: gates may have arbitrary delays, wires *after fork* may have delays less than one gate delay

- Wire delay model = gate+wire delay model: wires (and gates) may have arbitrary delays

Delay models and correctness



- This design is correct for unbounded **gate** delay model
- This design is incorrect for unbounded **wire** delay model
  If $d1 > d3 + d5$ or $d2 > d3 + d4$ then the implementation does not satisfy the specification
- Correctness of this design under the bounded delay model depends on delay constraints.

Design styles for asynchronous control logic

- **Bounded delays:** realistic delay model for gates and wires.

  Technology mapping is easy.
  Verification is difficult since time must be included in the model.

- **Speed-independent:** pessimistic delay model for gates (unbounded) and often acceptable delay model for wires (delays after fork are less than one gate delay).

  Technology mapping is more difficult.
  Verification is easy since time is excluded from the model.

- **Delay-insensitive:** pessimistic delay model for gates and wires (both are unbounded).

  The class of such circuits (built out of basic gates) is almost empty.

- **Quasi-delay insensitive** = delay-insensitive except critical wire forks, called *isochronic forks*.

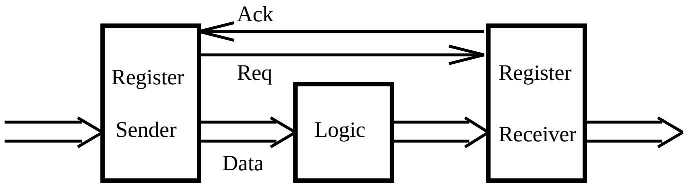
  In practice this is the same as speed-independent.

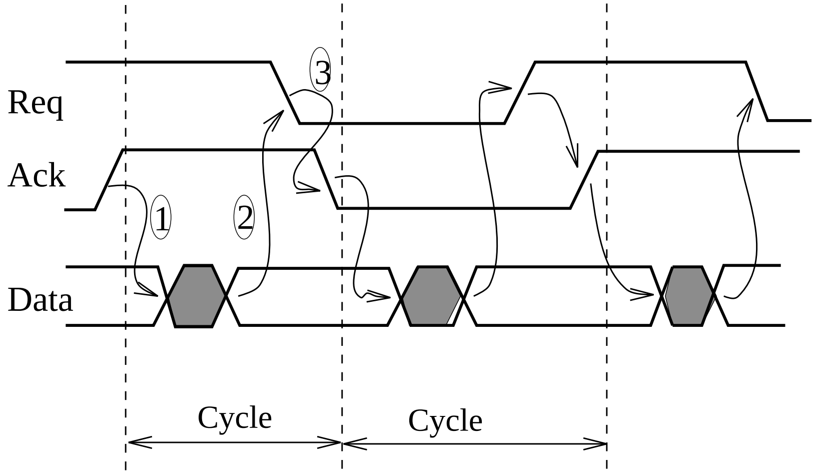
Signalling methods

- *Two-phase:*
  rising and falling signal transitions are equivalent

- *Four-phase:*
  rising and falling signal transitions are not equivalent

Two-phase signalling

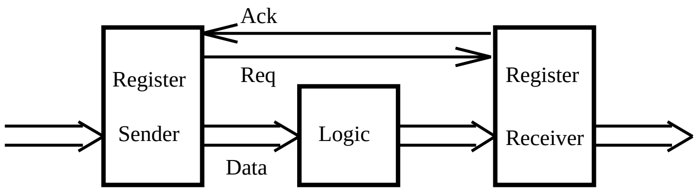


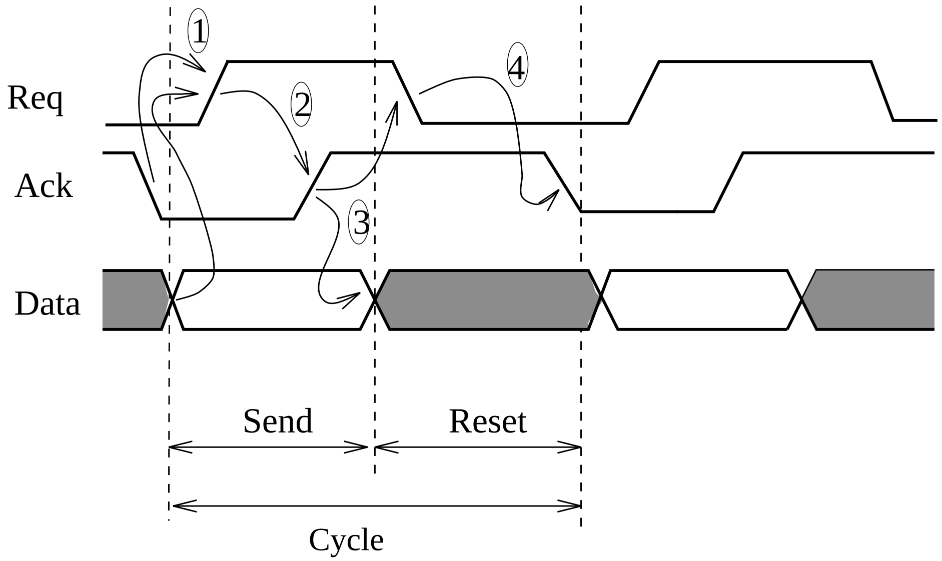


Two transitions at the control lines during one cycle

$acknowledge \Rightarrow$ data change $\Rightarrow request$

Four-phase signalling





Four transitions at the control lines during one cycle

Control is simpler

Reset phase: change data and reset control

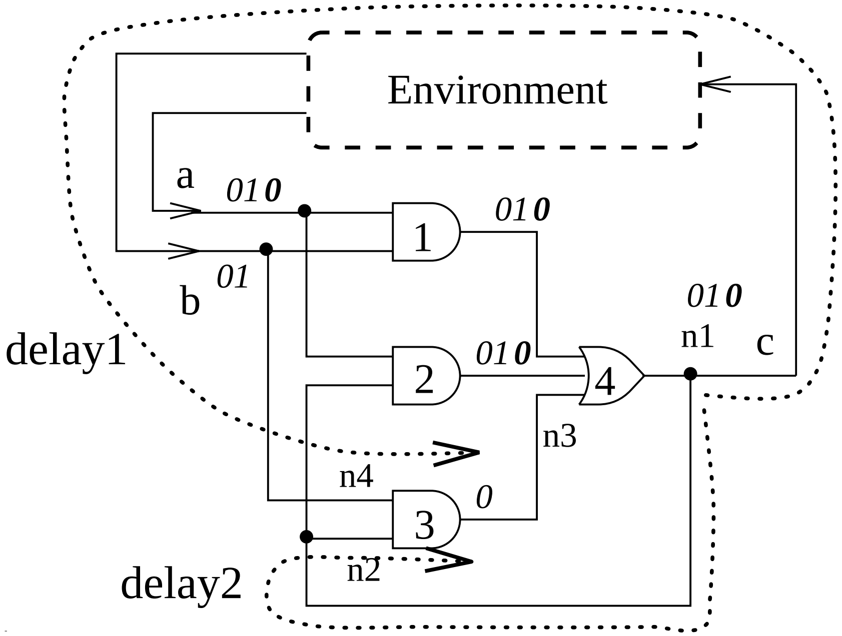
Models of the environment

Asynchronous circuits operate in the environment.

- Slow enough environment = Fundamental mode:
  inputs can change *after* the asynchronous system has settled into a stable state
- Reactive environment = Input/output mode:
  inputs can change as soon as the first output signal has changed due to the previous input change
- Constraints on input changes for fundamental mode and I/O-mode:
  - Single input change (SIC) - too slow
  - Multiple input change (MIC) - difficult to design using classical asynchronous FSM technique
  - Unrestricted input change (UIC) - very difficult to design using classical asynchronous FSM technique

Fundamental mode constraints are similar to setup and hold restrictions in synchronous design. Fundamental mode is often too restrictive.
Example: any pipeline operates in I/O-mode

Environment and correctness



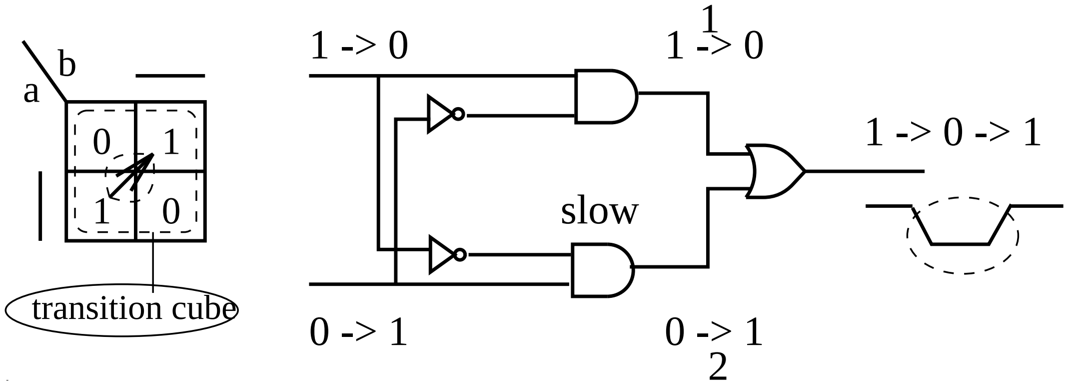
- For fundamental mode this design is correct (necessary delay constraints hold by definition)
- For I/O-mode this design may be incorrect unless "*Delay 1 > Delay 2*" type constraints hold

Assumptions on the environment influence the correctness of asynchronous design

Goals of tutorial

- **Will** present design methodology for
  - I/O-mode (most general)
  - Bounded and unbounded delay models
    (each of them may be useful for certain designs)
  - Four-phase signalling (more efficient and fast control).
    Two-phase signalling can be also handled.
- **Will not** present
  - Specific methods for Fundamental mode
  - Specific methods for two-phase signalling
  - All the methods that have been devised in history ...

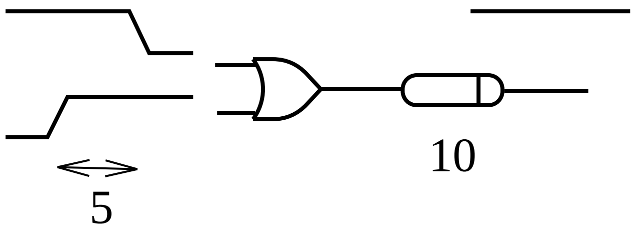
Hazards



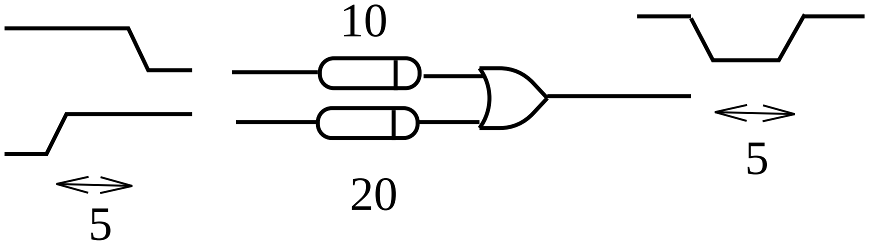
Hazard example:
function static 1-hazard
for $c = a \; xor \; b$

- Hazard: signal transition not specified by the designer
- Hazards make asynchronous design difficult
- Theory of hazards: [Unger, Kung, Nowick]

Hazards and delay model



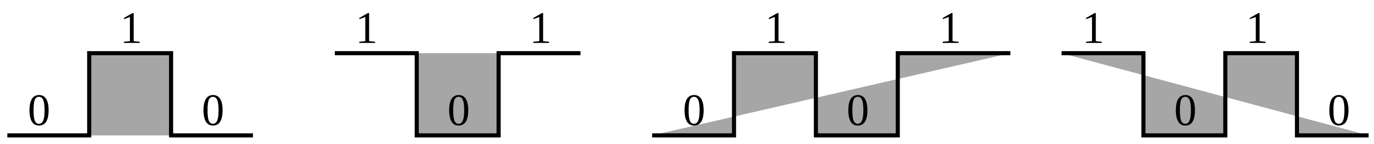
- Inertial gate delay model
  $\Rightarrow$ no hazards



- Wire delay model
  $\Rightarrow$ hazards

Hazard analysis depends on the delay model

Taxonomy of hazards



Static hazards   Dynamic hazards

- **Signal behavior**:
  static (no transition is required)
  or dynamic (monotonous transition is required)

- **Circuit type**:
  combinational (circuits without feedbacks) or
  sequential (circuits with feedbacks)

- **Due to specification or due to implementation**:
  function or logic; essential or not essential

- **Due to the I/O mode of operation**: delay hazards occurs when two input transitions are too close together

Combinational hazards

- Function hazards: inherent in the specification of the logic function.

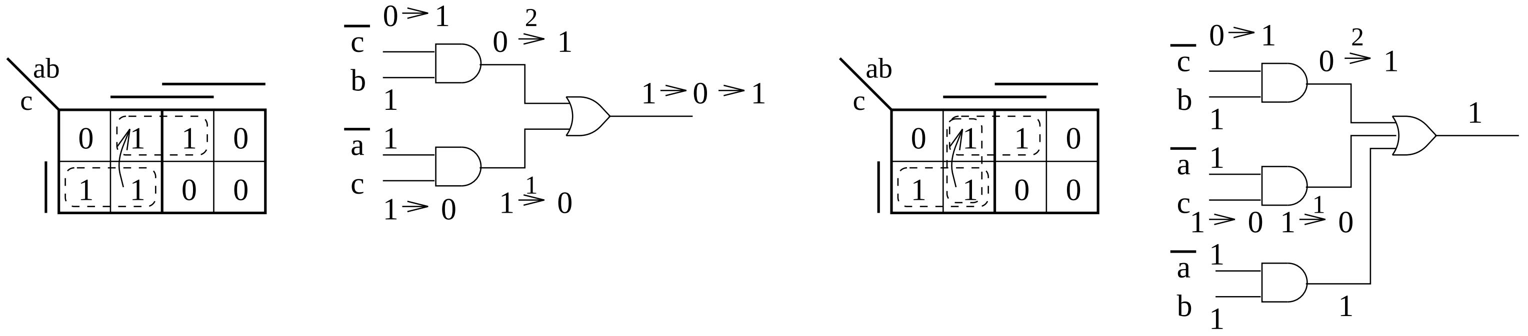
  Condition for 1-hazards: a transition cube contains 0

  Cannot be eliminated by changing the logic

- Logic hazards:
  depend on the particular implementation of the logic function

  Can be eliminated by changing the logic (or sometimes the delays)

Static logic hazard elimination

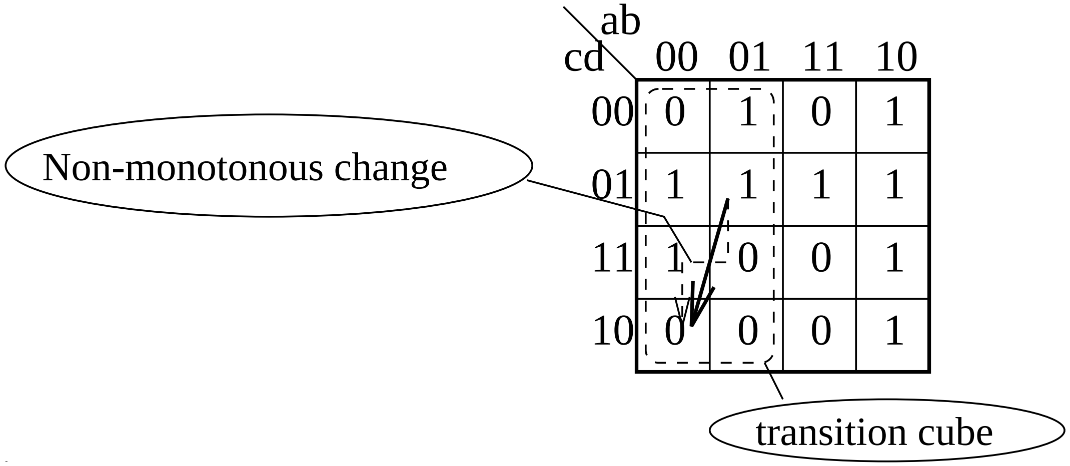


Static hazard: a signal which should remain constant changes

Static logic 1-hazard elimination condition:
each transition cube is contained inside a cube of the cover

Static logic 0-hazards never occur in a two-level Sum-of-products implementation

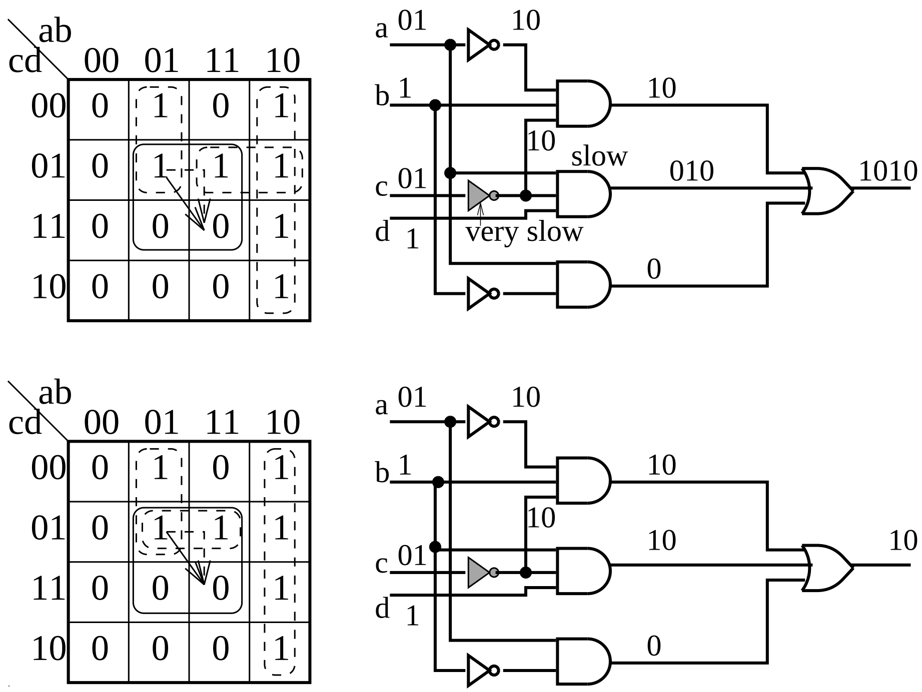
Dynamic function hazards



Dynamic hazard: a signal which should change once changes multiple times

Dynamic function hazard:
non-monotonous change of the function inside the transition cube

Dynamic logic hazard elimination



Dynamic logic 1to0-hazard elimination condition:
if transition cube intersects cube c of the cover,
then c contains the *start point* of the transition cube

0to1-hazards are symmetrical (the same for the end point)

Sequential hazards

Sequential circuits $\approx$ circuits with feedbacks

Sequential hazards occur because feedbacks propagate new values before the input net completes signal propagation to all inputs of the next state logic

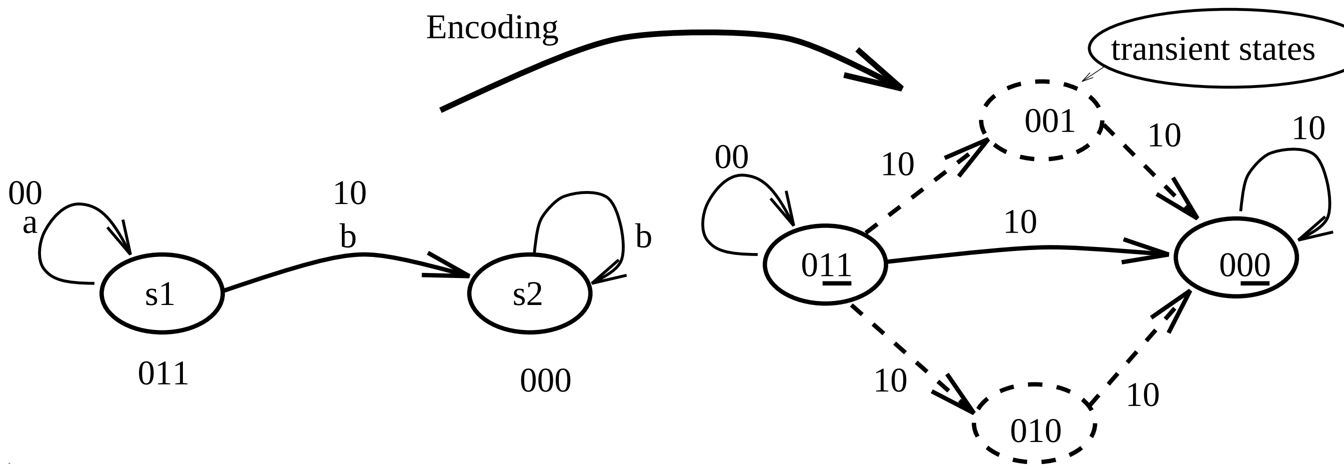
- **Essential hazard**: inherent in the FSM specification.

  cannot be eliminated by another state encoding $\Rightarrow$
  cannot be eliminated for unbounded delays

  can be masked out by controlling the delays

- Non-essential hazards (**races**):
  can be eliminated by a proper state encoding

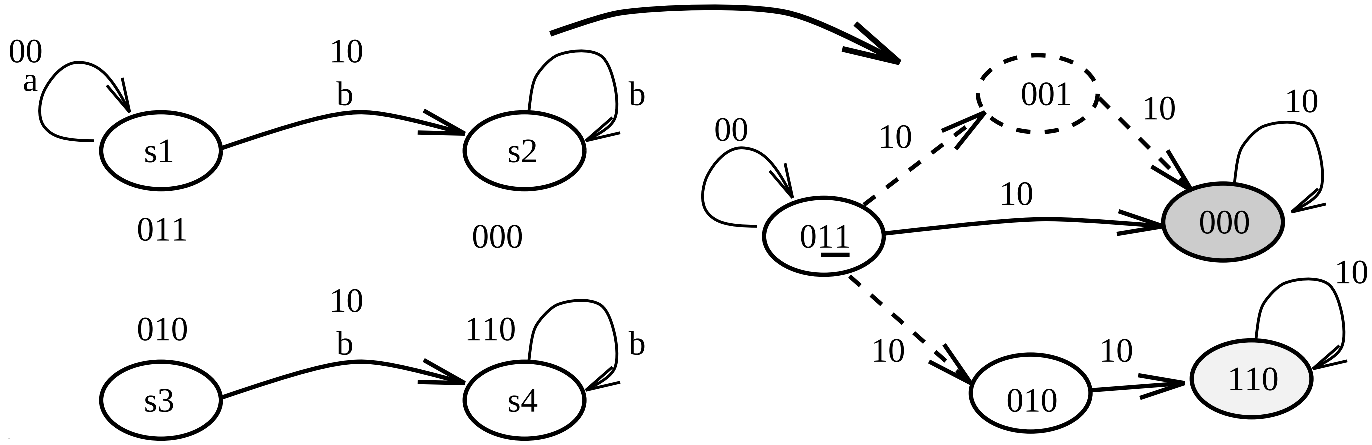
Non-critical races



Race: more than one state signal changes

Non-critical race:
all transient states drive the FSM into the same final state

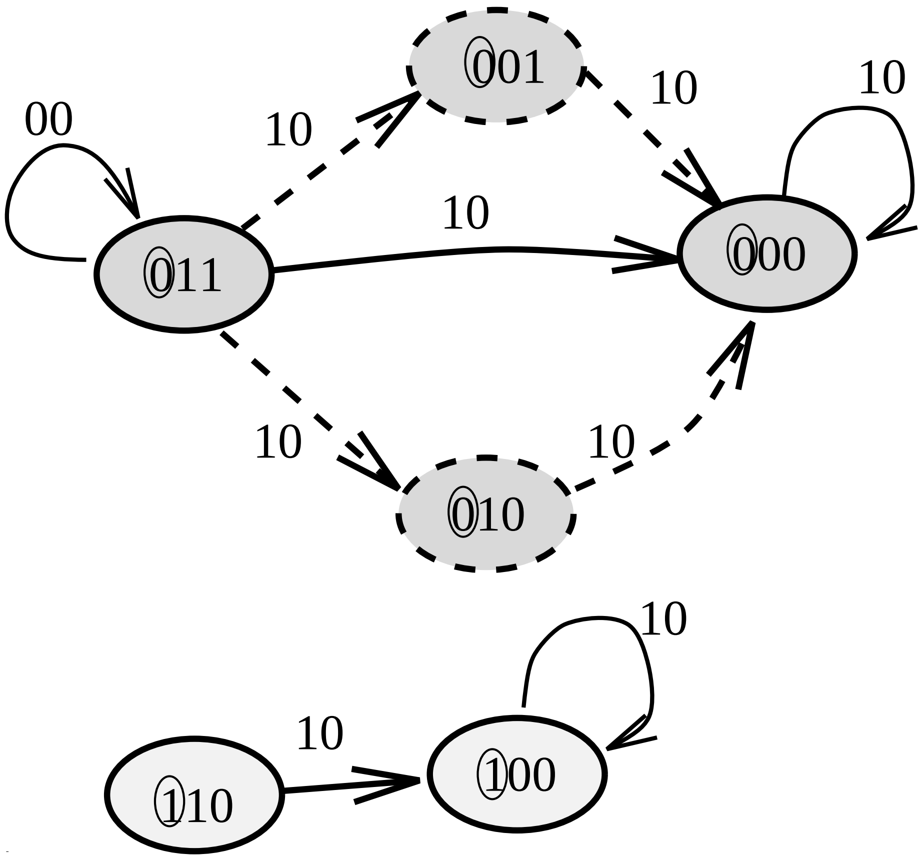
Critical races



Critical race:
different transient states drive the FSM into different final states.

Final state depends on the race winner

Race-free encoding



Tracey encoding:

Two state transitions under the same input
$s1 \overset{b}{\rightarrow} s2$ and $s3 \overset{b}{\rightarrow} s4$
must be distinguished by a stable state signal:
$\mathbf{0}11 \overset{10}{\rightarrow} \mathbf{0}00$ and $\mathbf{1}10 \overset{10}{\rightarrow} \mathbf{1}00$

Types of race-free encoding

- **Single bit change encoding:** simple, bad throughput

  adjacent states differ by exactly one bit
  may require a few FSM cycles to reach a stable state

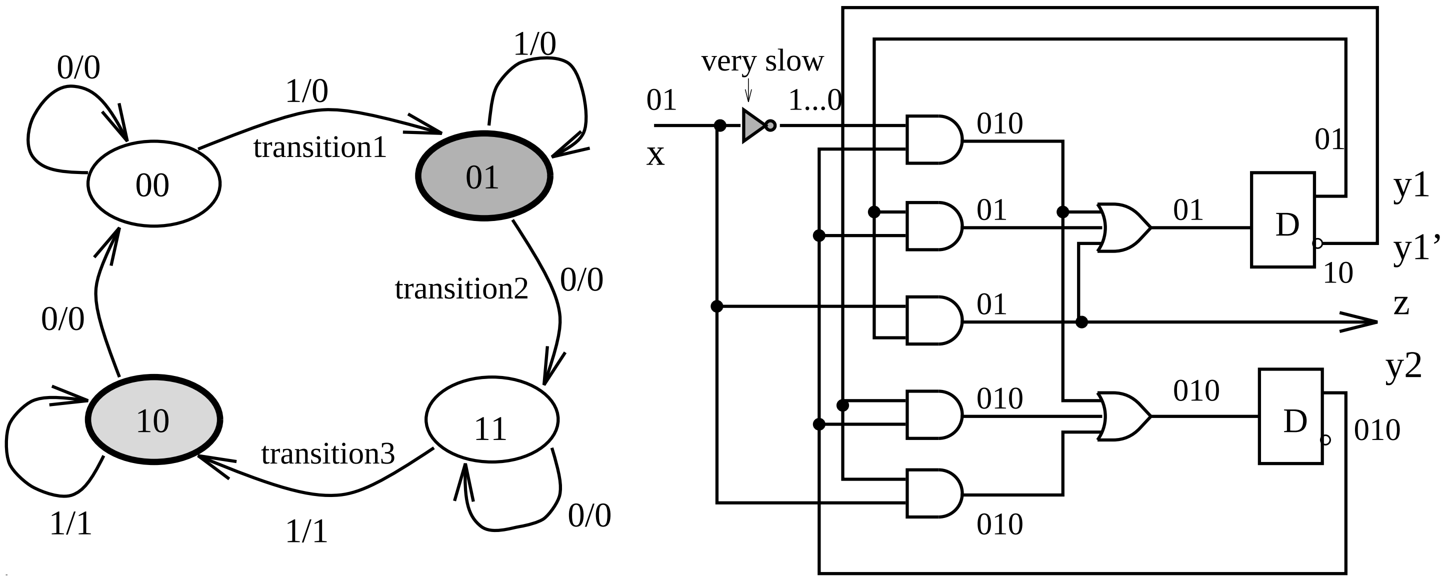
- **Single transition time encoding:**
  better throughput, more complex

  adjacent states may differ by multiple bits
  requires exactly one FSM cycle to reach a stable state

  Example: Tracey encoding

Race-free FSMs may still have **essential hazards**

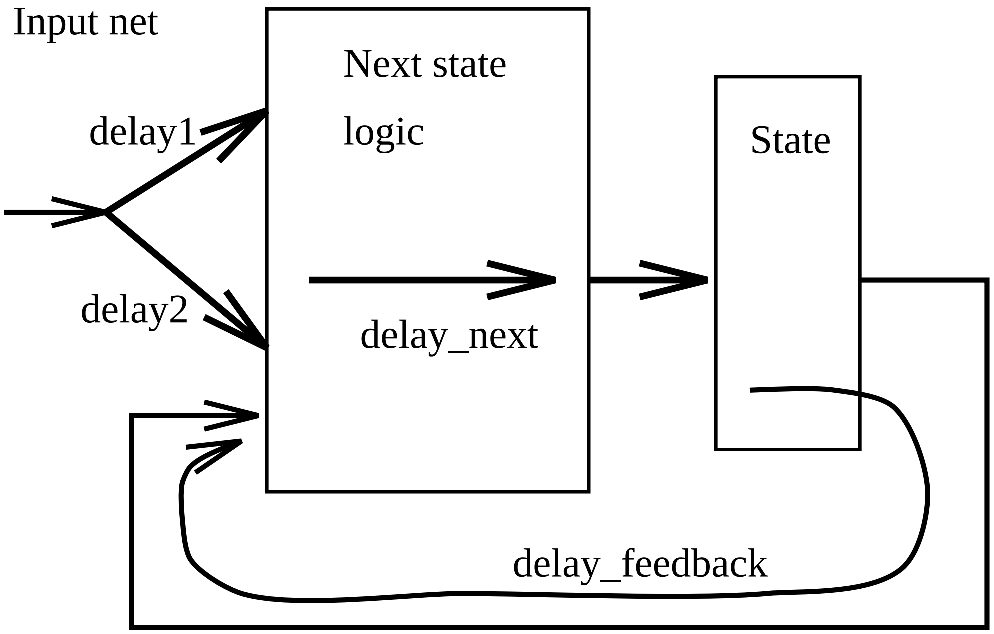
Mod-2 counter: essential hazards



Unger's condition for essential hazards:

state after **three** transitions of an input signal $\neq$
state after **one** transitions of the same input signal

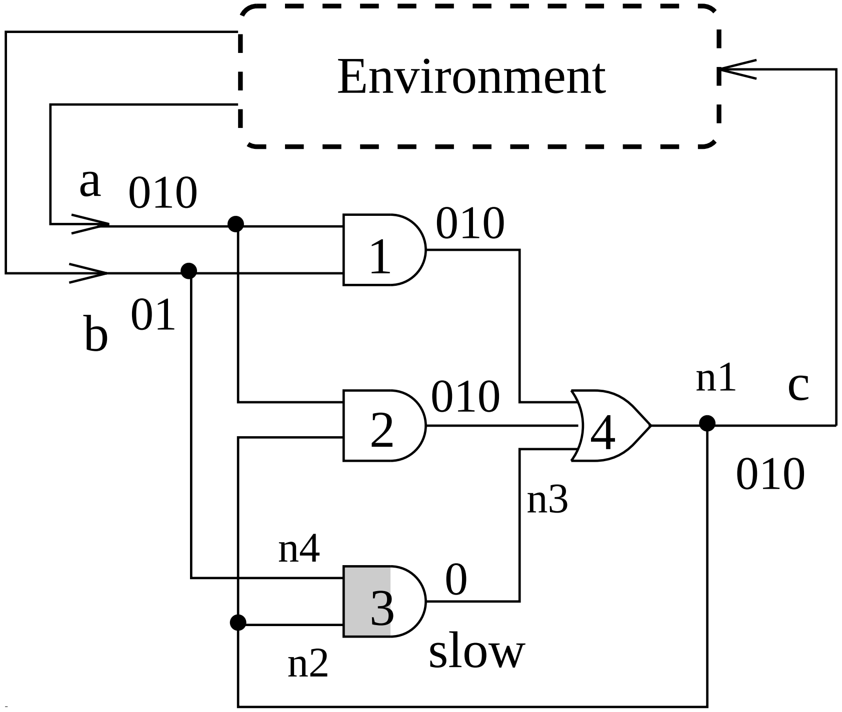
Why essential hazards occur



No essential hazards if and only if
$|delay1 - delay2| < mindelay_{next} + mindelay_{feedback}$

Essential hazards can be removed by controlling the delays

Delay hazards



- I/O-mode:
  input b changes
  after output c, but
  before the output of gate
  3 has settled
- This is a **delay hazard**

Delay hazards may occur under I/O-mode even in designs free from essential hazards

Hazard elimination methods

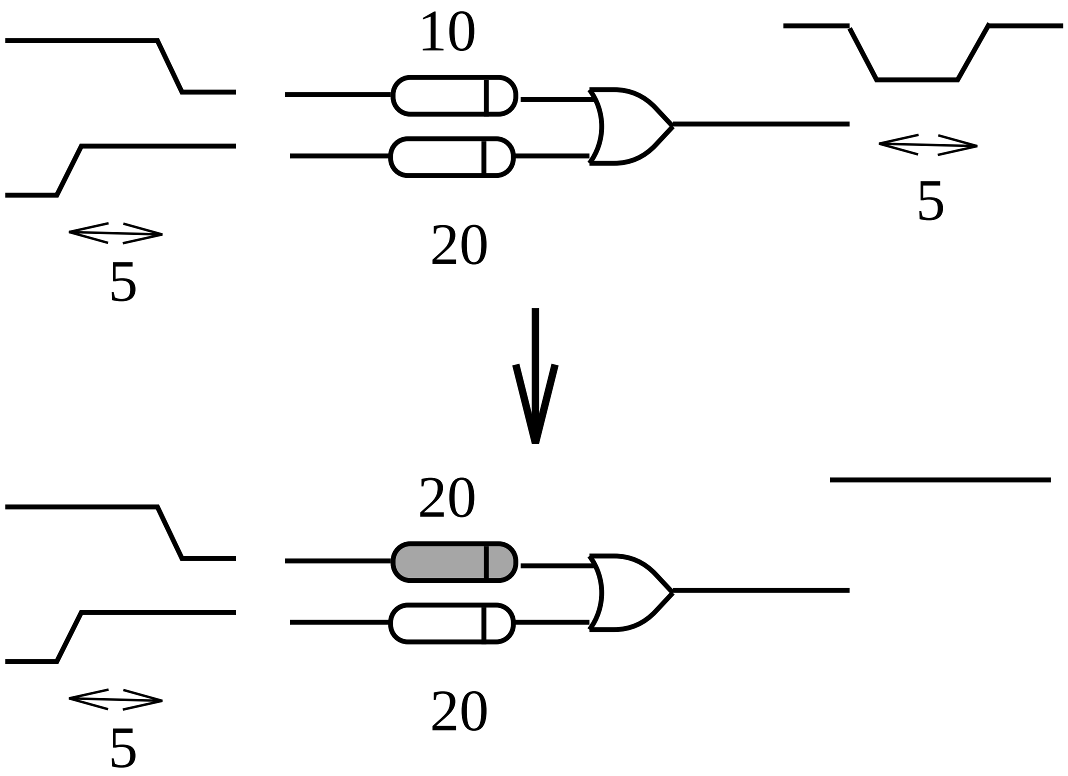
- Changing delays (delay padding).
  Works only for bounded delay model

- Logic transformations (circuit redundancy).
  Works both for bounded and unbounded delay models

- Encoding of inputs and outputs (code redundancy) and/or changing allowed input transitions.
  Works both for bounded and unbounded delay models

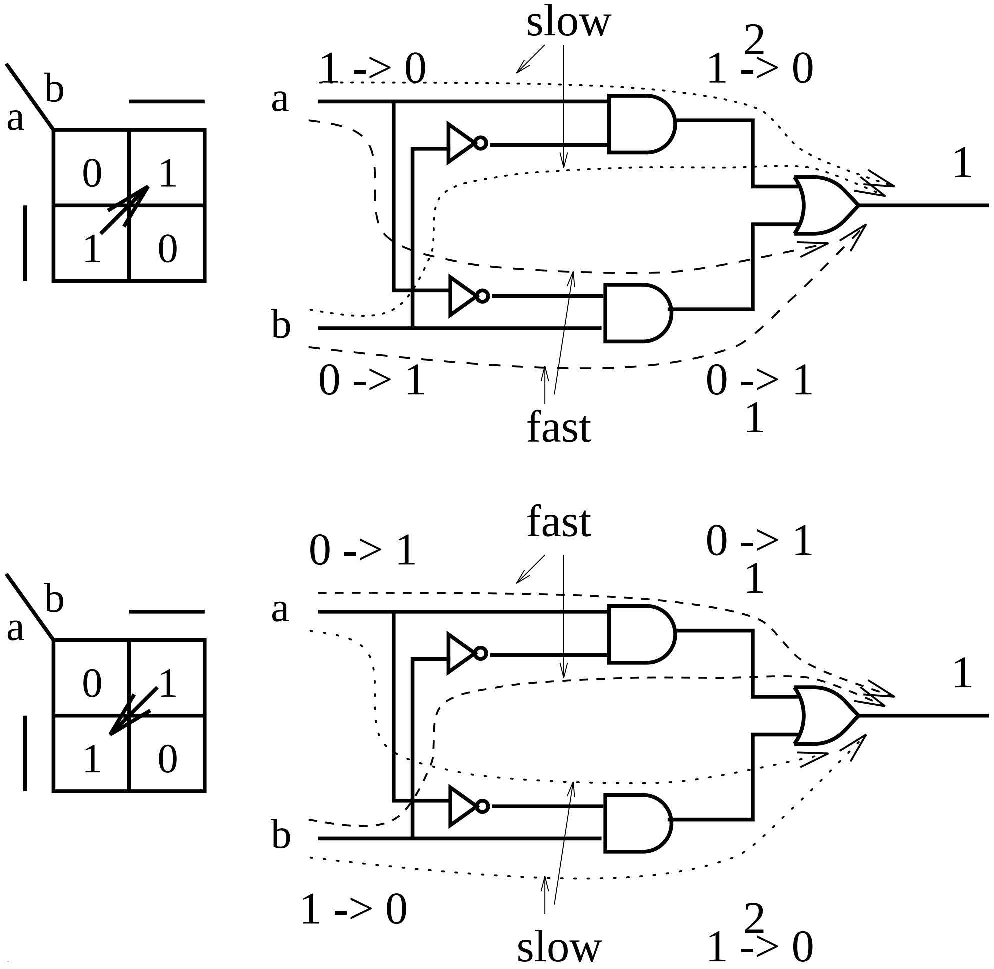
Fighting with hazards always requires some area and time overhead

Masking hazards by delay padding (1)



Controlling delays can eliminate hazards

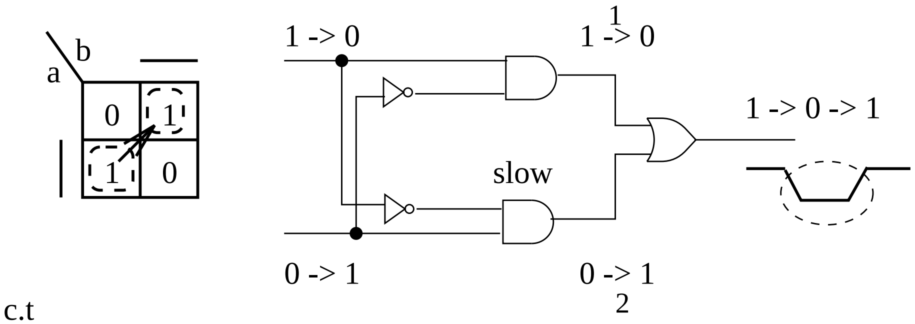
Masking hazards by delay padding (2)

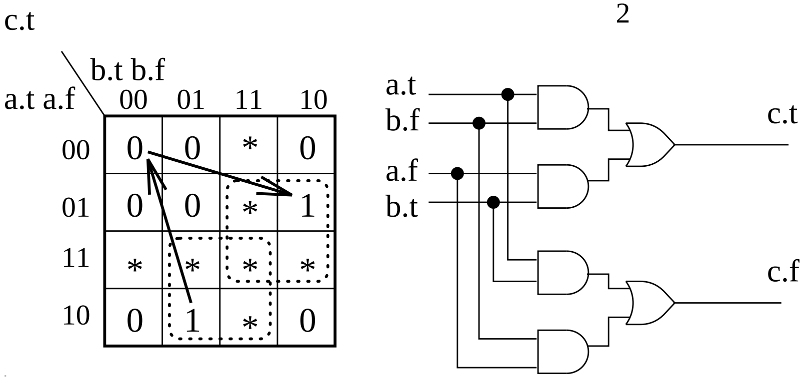


Contradictory delay constraints ⇒

controlling delays cannot eliminate hazards in general

Hazard elimination by redundant encoding





Dual-rail four-phase XOR-gate

- Function hazards cannot be eliminated by changing the logic $\Rightarrow$ change the *specification*
- Hazards depend on the allowed input transitions: XOR-gate is hazard-free under unique input changes in the fundamental mode
- Redundant encoding of input and output signals and four phase signalling protocol allows to eliminate even **functional** hazards

Modern view on hazards and races

Classical hazard/race taxonomy is complex for sequential circuits even in Fundamental Mode.

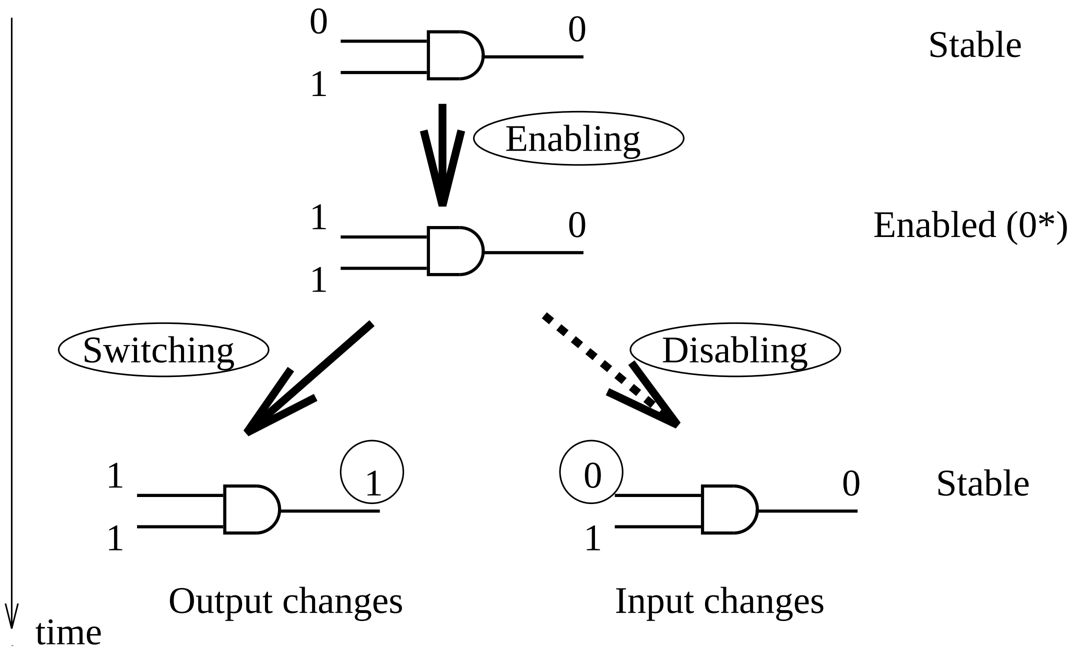
It is much more complex for the I/O-mode:

- Hazards may be viewed as signal transition *disabling*
- Critical races may be viewed as *incomplete* (ambiguous) specifications

Hazard avoidance:
design a circuit in which transition disabling cannot occur

Critical race avoidance:
make the specification complete and unambiguous

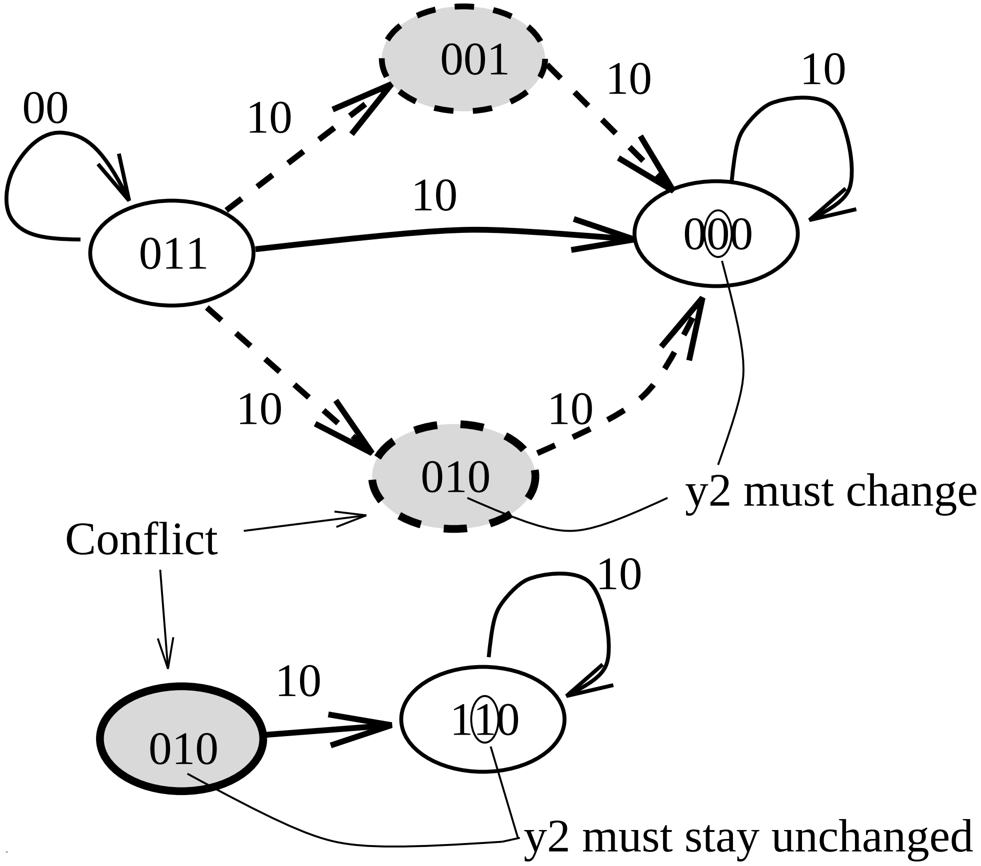
Speed-independent design



For unbounded gate delay model:
Disabling cannot occur $\Rightarrow$ no hazards

No need for a complex theory of hazards for this delay model

Critical races as incomplete state coding

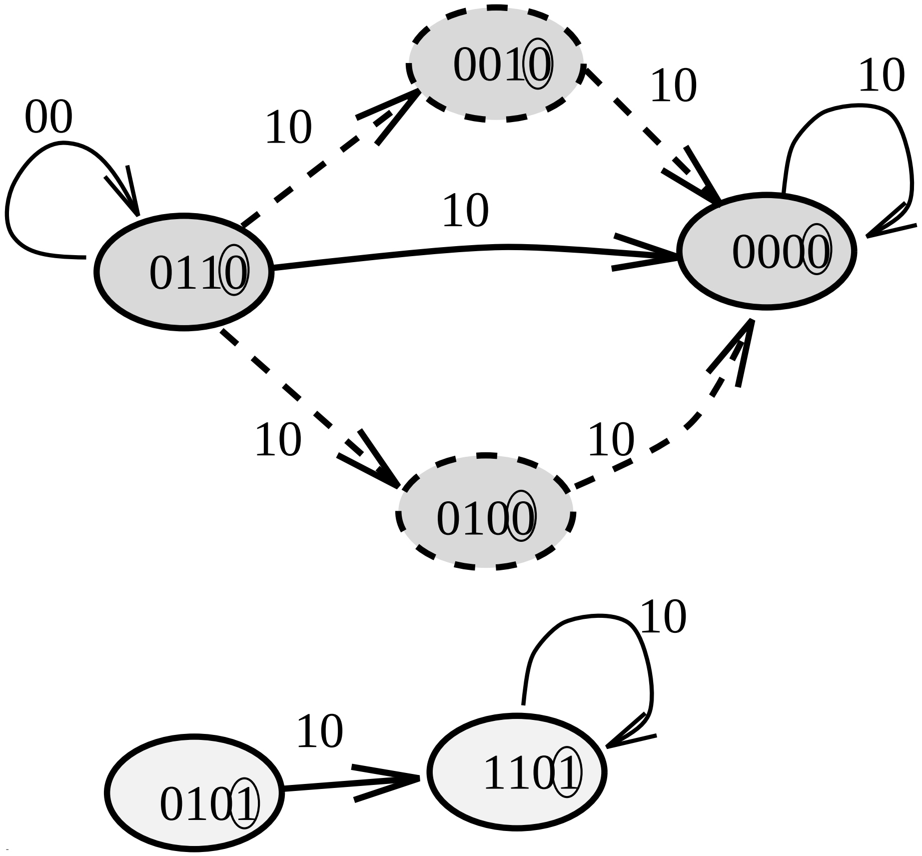


FSM spec is ambiguous, i.e. **incomplete**

State assignment conflict:

$< y1, y2, y3 >$	next state $y2'$
010	0
010	1
...	...

Complete state coding



Solving state assignment **conflicts:**

Add new state signals or

Remove one of the states (by re-encoding, reducing concurrency)

Data path design

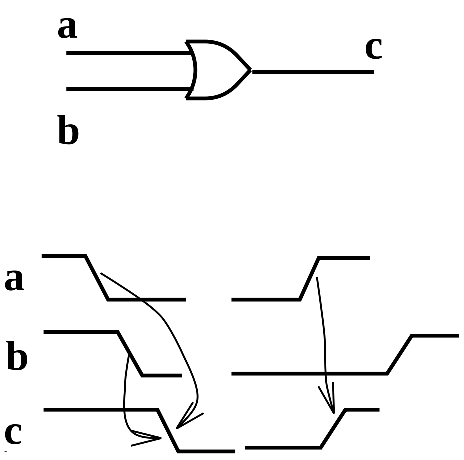
Two ways to design data path:

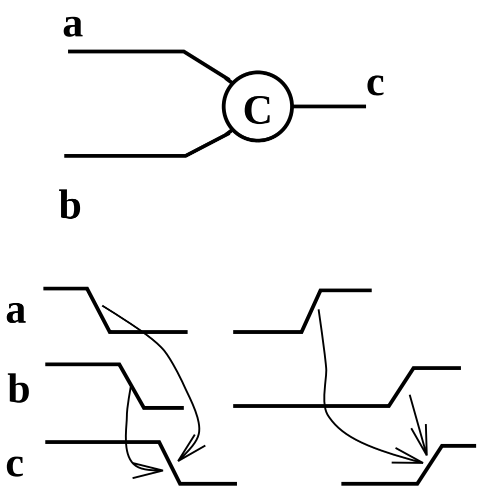
- Synchronous data path - bundled data
- Asynchronous data path
  (speed-independent = quasi delay-insensitive)

Asynchronous data path is useful if performance is data dependent and the average case computation delay is much less than the worst case delay

Two efficient methods for asynchronous data path: reduced direct logic (RDL) and differential cascode voltage switch logic (DCVSL)

Indication





- **OR-gate:** output indicates falling input transitions
- **C-element:** output indicates falling and rising input transitions

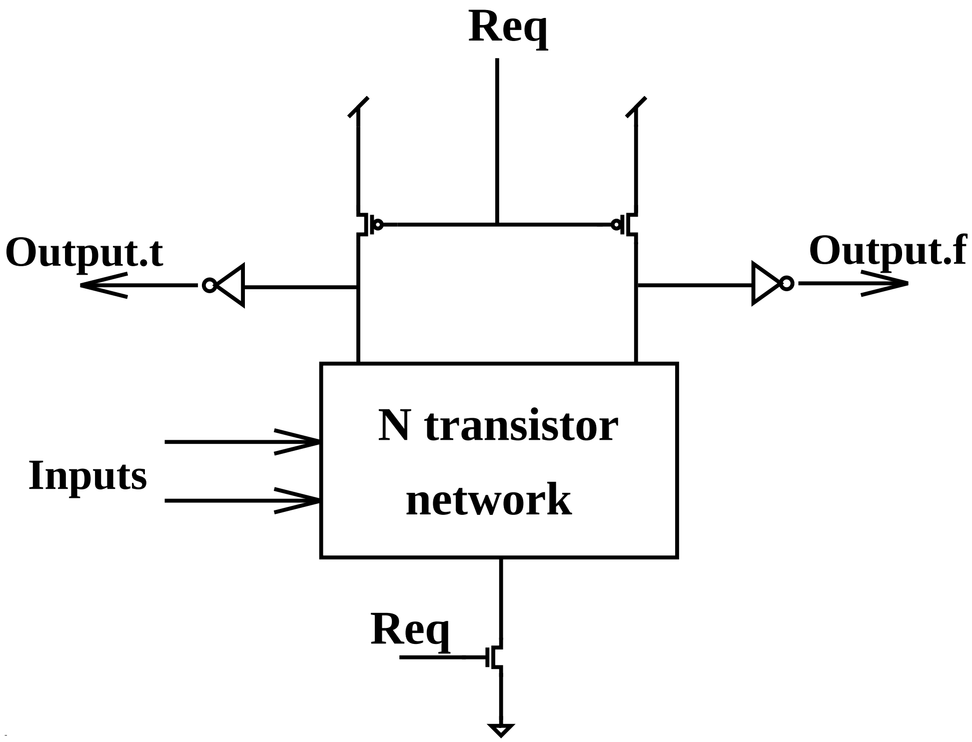
Indication: A. Taubin (1980)
Transition c indicates transition a if and only if:

- c cannot occur without a
- c is a "completion detector" for a.

Design idea:

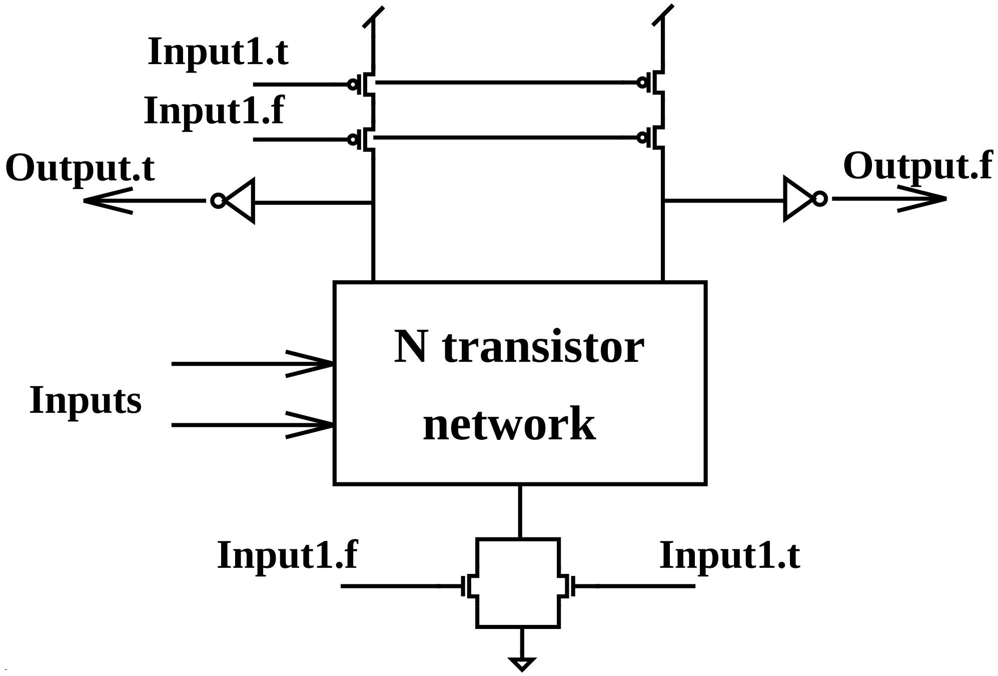
- All input and internal transitions must be indicated by some output transitions
- Choose indicating outputs to optimize performance/area

Differential Cascode Voltage Switch Logic



- Inputs are (typically) dual-rail
- Additional request signal is used for precharging
- Outputs are dual-rail four-phase (valid-empty protocol)
- Staticizers (cross-coupled inverters) may be needed for output signals
- Two cross-coupled pull-up transistors may improve performance

Reduced direct logic

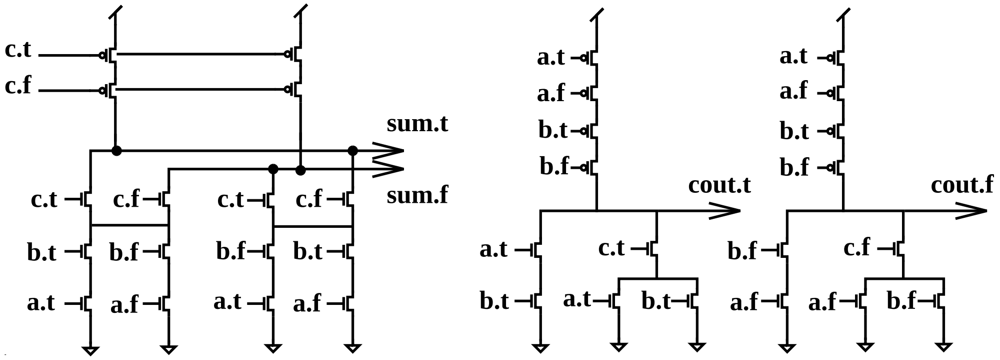


- Inputs and outputs are four-phase dual-rail
- No additional request signal
- Full custom design only

Design idea:

- Add a pair of p-transistors for each output and a pair of n-transistors for each dual rail input
- Optimize n-transistor logic
- Distribute indication of the inputs among the outputs

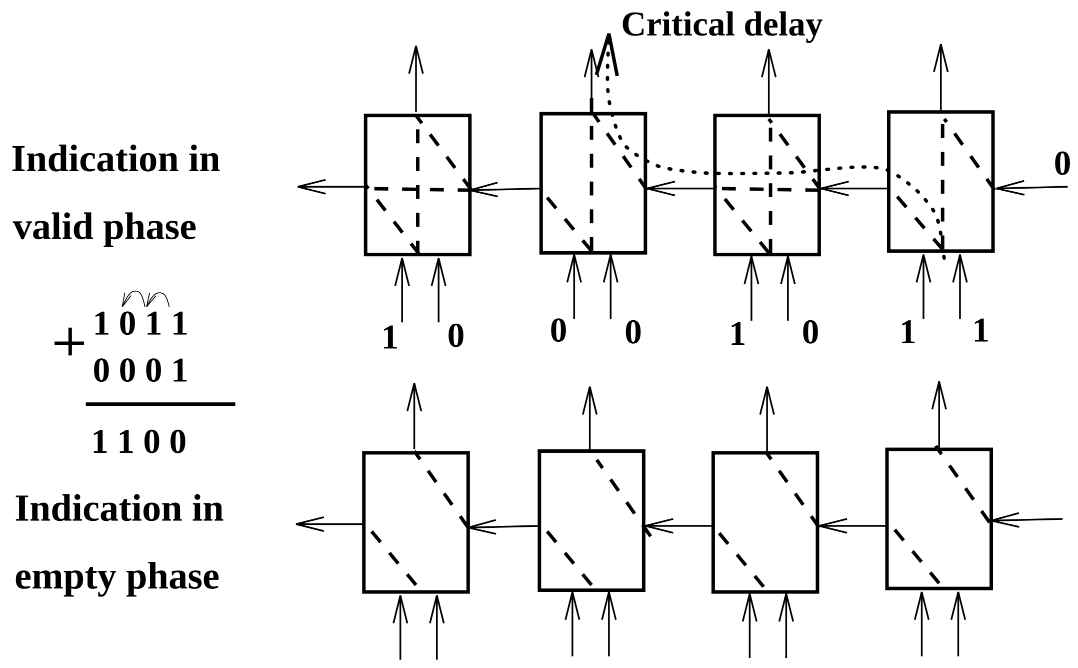
Asynchronous ripple-carry adder



A. Martin (1991)

- Asynchronous cell transistor count = 34
- Synchronous cell transistor count (without pass transistors) = 28
- Inputs and outputs are four-phase dual-rail

Critical delay in asynchronous adder



- Critical delay is proportional to $log_2 N$ (N = number of bits)
  It is determined by the maximal number of contiguous different bits
- 32-bit adder delay (1.6 MOSIS CMOS):
  asynchronous: 11 nsec   synchronous: 40 nsec

Part 2: Specification and synthesis

- Specification of control
  - CSP
  - FSM
  - Signal Transition Graphs (STG)
- STG in details: basics, limitation and extensions, timing
- Synthesis overview
- Simulation engine
- Complete state coding = race elimination

CSP

- CSP = communicating sequential processes.
- Concurrent processes communicating by input and output commands on channels.
- Semantics are based on trace theory. [Udding, van de Snepscheut, Rem]
- In general, techniques try to come up with quasi-delay-insensitive implementations
- Two main approaches: Production rule based [Martin, ...] and Syntax-directed decomposition [Ebergen, Berkel]

CSP PROGRAM

$$*[[l_i]; r_o \uparrow; [r_i]; r_o \downarrow; [\neg r_i]; l_o \uparrow; [\neg l_i]; l_o \downarrow]$$

- ";" = sequencing.
- $r_0 \uparrow$ = r_0 goes high, $r_0 \downarrow$ r_0 goes low.
- $[l_i]$ = wait until l_i is high, $[\neg l_i]$ wait until l_i is low.
- Program meaning: l_i has to be high before r_0 goes high. The circuit will wait until r_i goes high, before r_0 will go low again and so on ....

PRODUCTION RULES

$$
\begin{array}{r}
l_i \mapsto r_o\uparrow \\
r_i \mapsto r_o\downarrow \\
\neg r_i \mapsto l_o\uparrow \\
\neg l_i \mapsto l_o\downarrow
\end{array}
$$

- production rule = boolean condition $\mapsto$ transition.
- $l_i \mapsto r_o\uparrow$: when l_i is high, r_o will go high.
- $\neg r_i \mapsto l_o\uparrow$: when r_i is low, l_o will go high.
- conflict: $r_o\uparrow$ and $r_o\downarrow$ are not mutually exclusive.

PRODUCTION RULES(2)

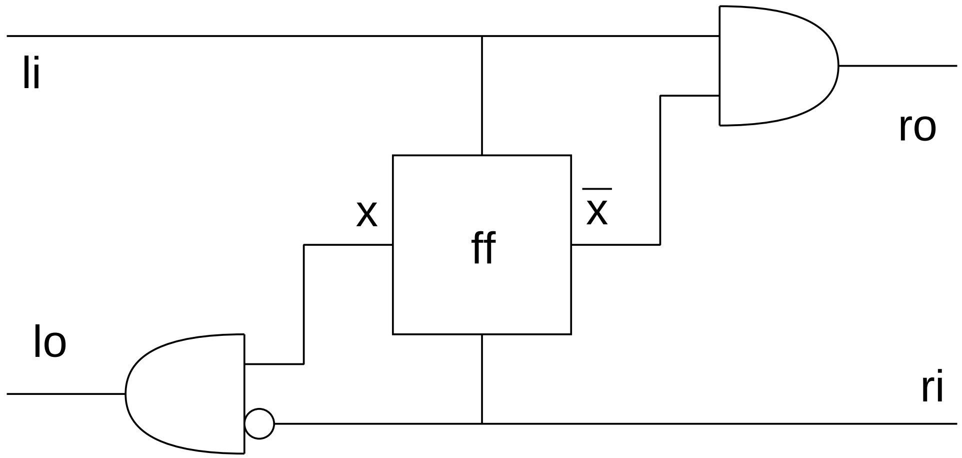
- Add a state signal to solve conflict:

$$*[[l_i]; r_o\uparrow; [r_i]; x\uparrow; [x]; r_o\downarrow; [\neg r_i]; l_o\uparrow; [\neg l_i]; x\downarrow; [\neg x]; l_o\downarrow]$$

- The new production rules:

$$\begin{aligned} \neg x \ \wedge \ l_i &\mapsto r_o\uparrow \\ r_i &\mapsto x\uparrow \\ x &\mapsto r_o\downarrow \\ x \ \wedge \ \neg r_i &\mapsto l_o\uparrow \\ \neg l_i &\mapsto x\downarrow \\ \neg x &\mapsto l_o\downarrow \end{aligned}$$

IMPLEMENTATION

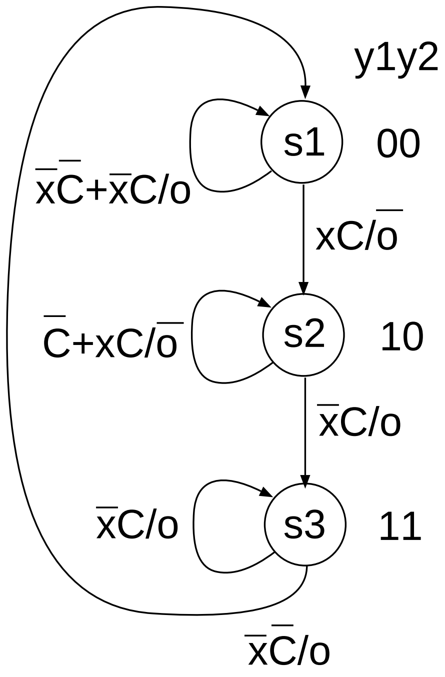


- The production rules can now be identified with operations.

- r_0 maps to an and-gate ($r_0 = \overline{x} l_i$).

$$\neg x \ \wedge \ l_i \mapsto r_o \uparrow$$
$$x \mapsto r_o \downarrow$$

FSM

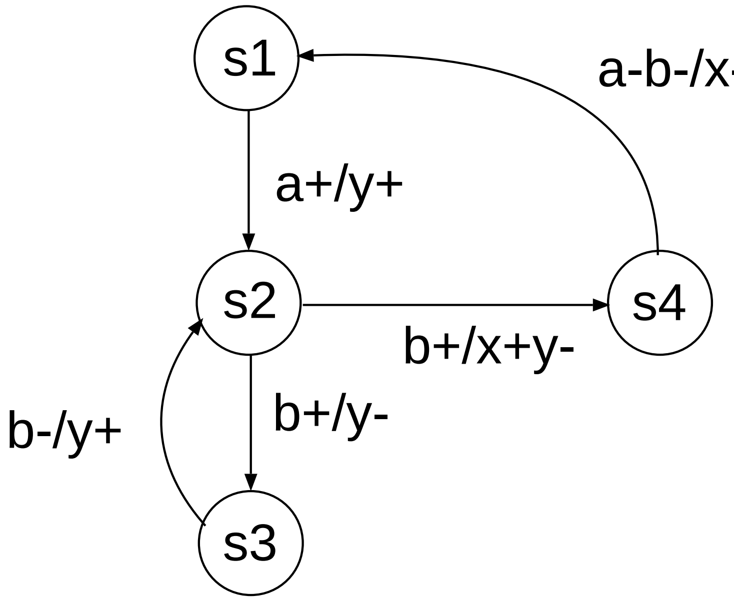


y1y2
s1
00
$\bar{x}\bar{C}+\bar{x}C/o$
$xC/\bar{o}$
s2
10
$\bar{C}+xC/\bar{o}$
$\bar{x}C/o$
s3
11
$\bar{x}C/o$
$\bar{x}\bar{C}/o$

FSM(2)

- Nodes = states, arcs = state transition
- Each arc is labeled with a pair of $input/output$ symbols.
- $input$ = condition for state transition to take place.
- $output$ = values for output signals after the state transition.
- Mostly limited to fundamental mode [Unger].

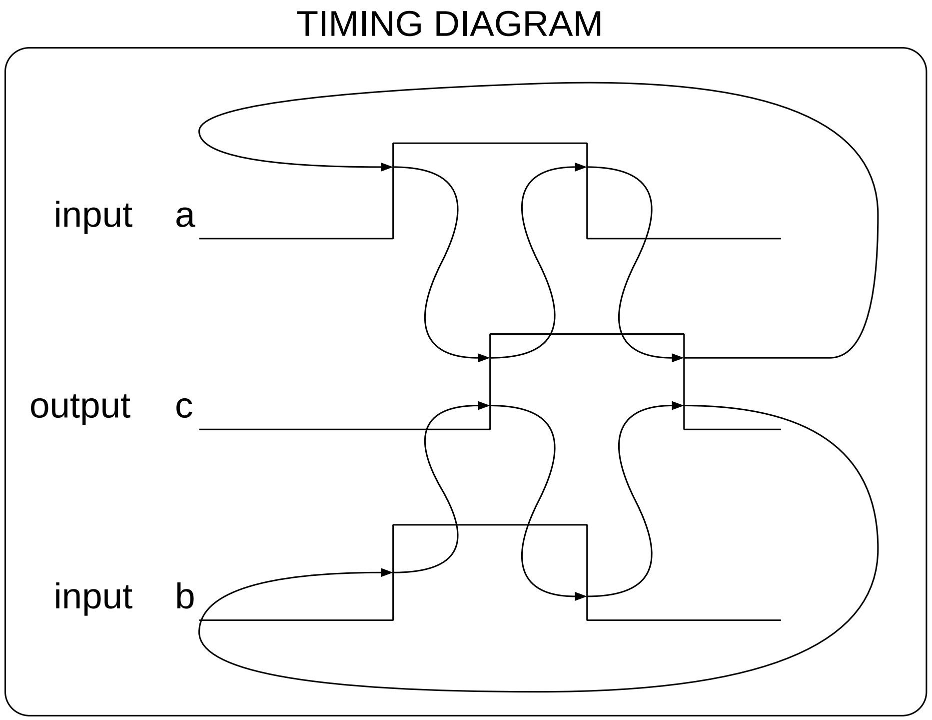
BURST MODE FSM

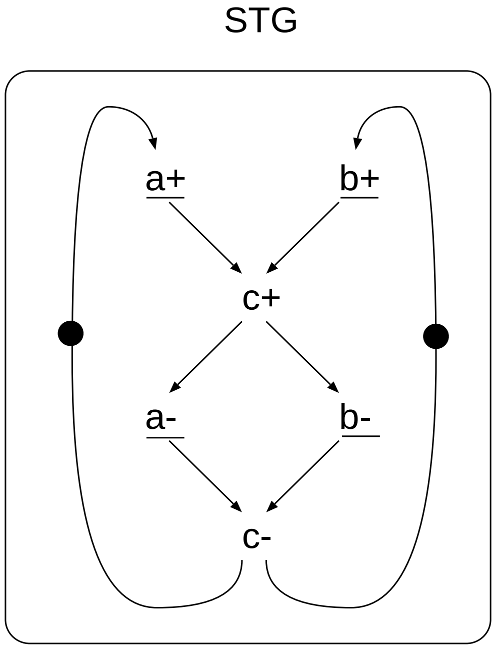


BURST MODE FSM

- based on signal transitions rather than signal levels [Nowick, Yun].
- First a set of input transitions fires.
- Second a set of output transitions fires.
- Finally state variables change and take the system to another state.
- Mostly limited to fundamental mode [Unger].

SIGNAL TRANSITION GRAPH

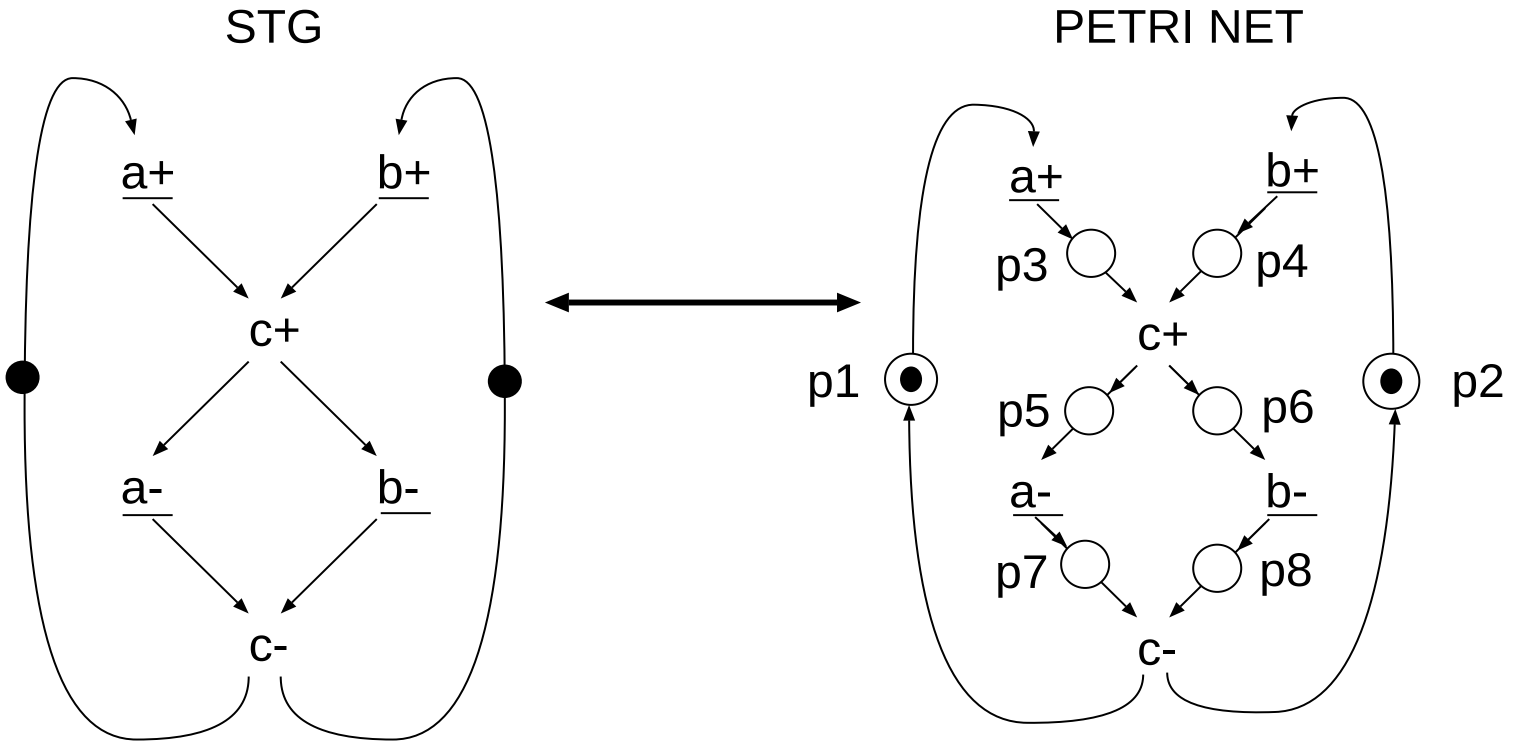




SIGNAL TRANSITION GRAPH(2)

- Vertices represent signal transitions.
- Arcs represent causal relations between transitions.
- Introduced by [Molnar 85] [Chu 87] [Rosenblum 85]

SIGNAL TRANSITION GRAPH (3)



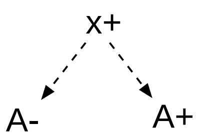
STG is an interpreted Petri net.

CONSISTENCY

INCONSISTENT

$A+ \dashrightarrow A+$ Two subsequent up-transitions without a down-transition in between

$A- \dashrightarrow A-$ Two subsequent down-transitions without an up-transition in between



Parallel transitions of the same signal

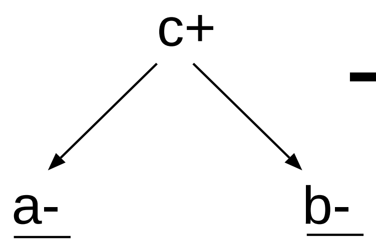
CONSISTENT

$A+ \dashrightarrow A- \dashrightarrow A+$

Two up-transitions with a down-transition in between (and vice versa)

MODELING POWER

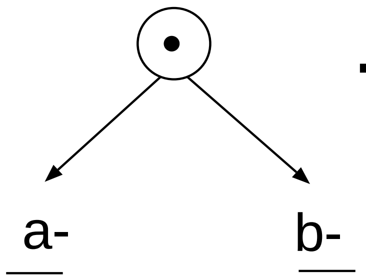
CONCURRENCY



a- concurrent with b-

EITHER a- fires before b-
OR b- fires before a-
OR a- and b- fire simultaneously

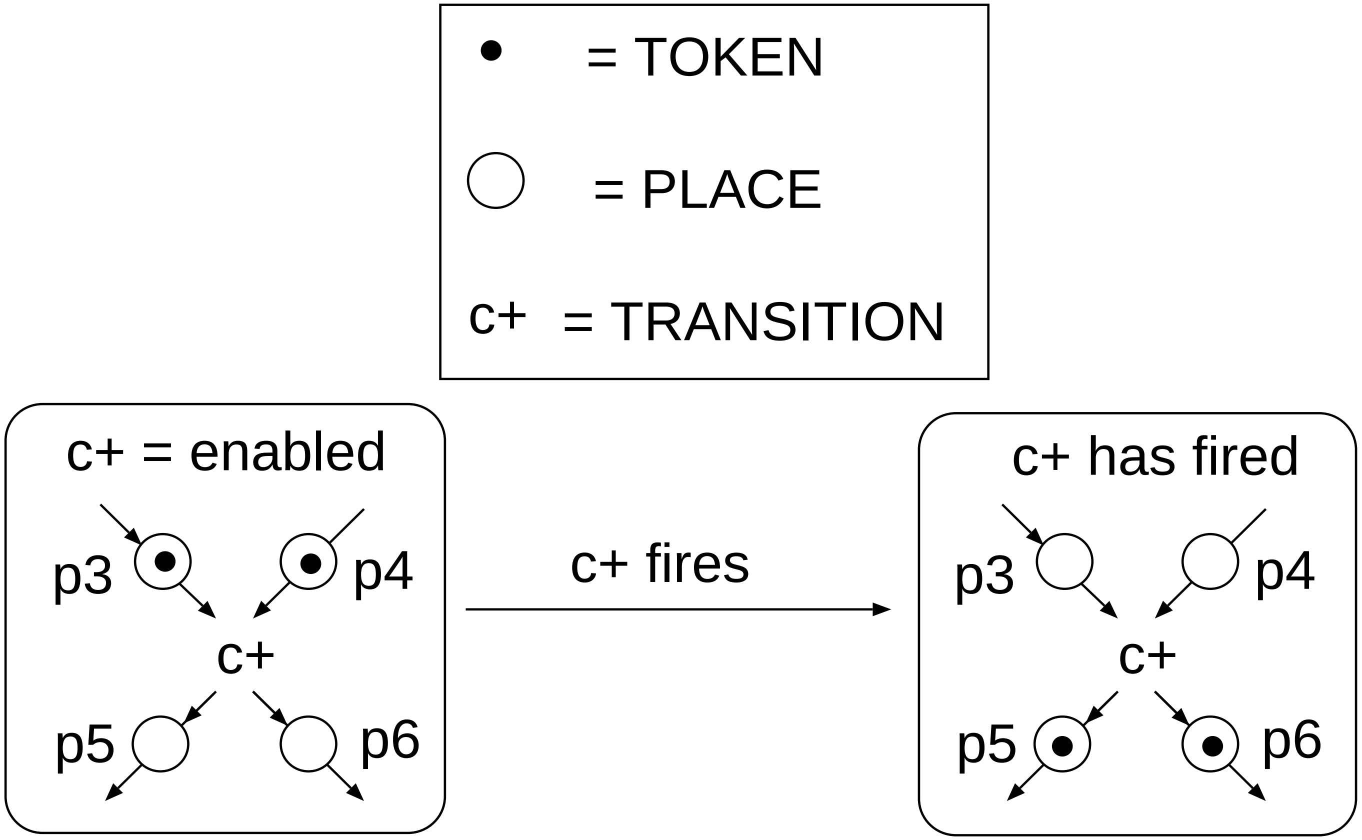
CHOICE



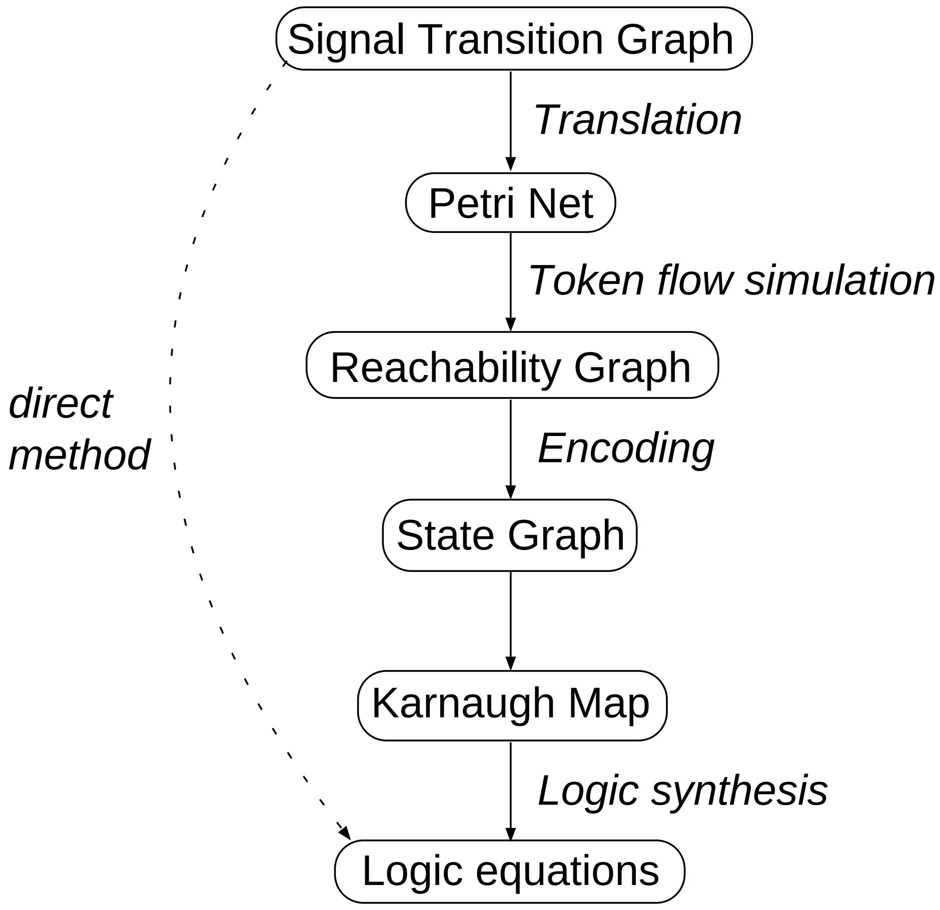
EITHER a- fires
OR b- fires
but not both

choice made by environment

FIRING SEMANTICS



SYNTHESIS

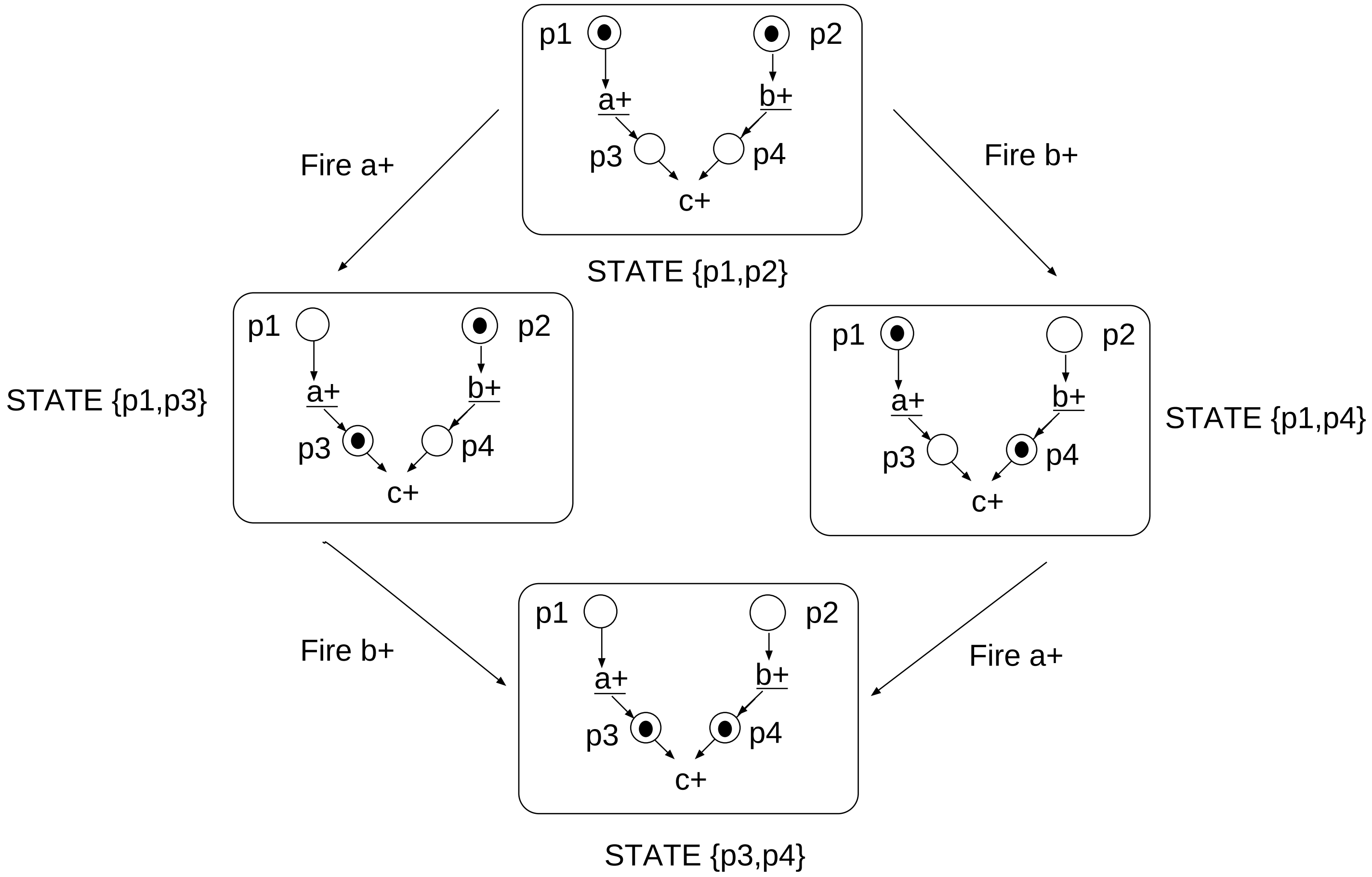


Signal Transition Graph
Translation
Petri Net
Token flow simulation
Reachability Graph
Encoding
State Graph
Karnaugh Map
Logic synthesis
Logic equations
direct
method

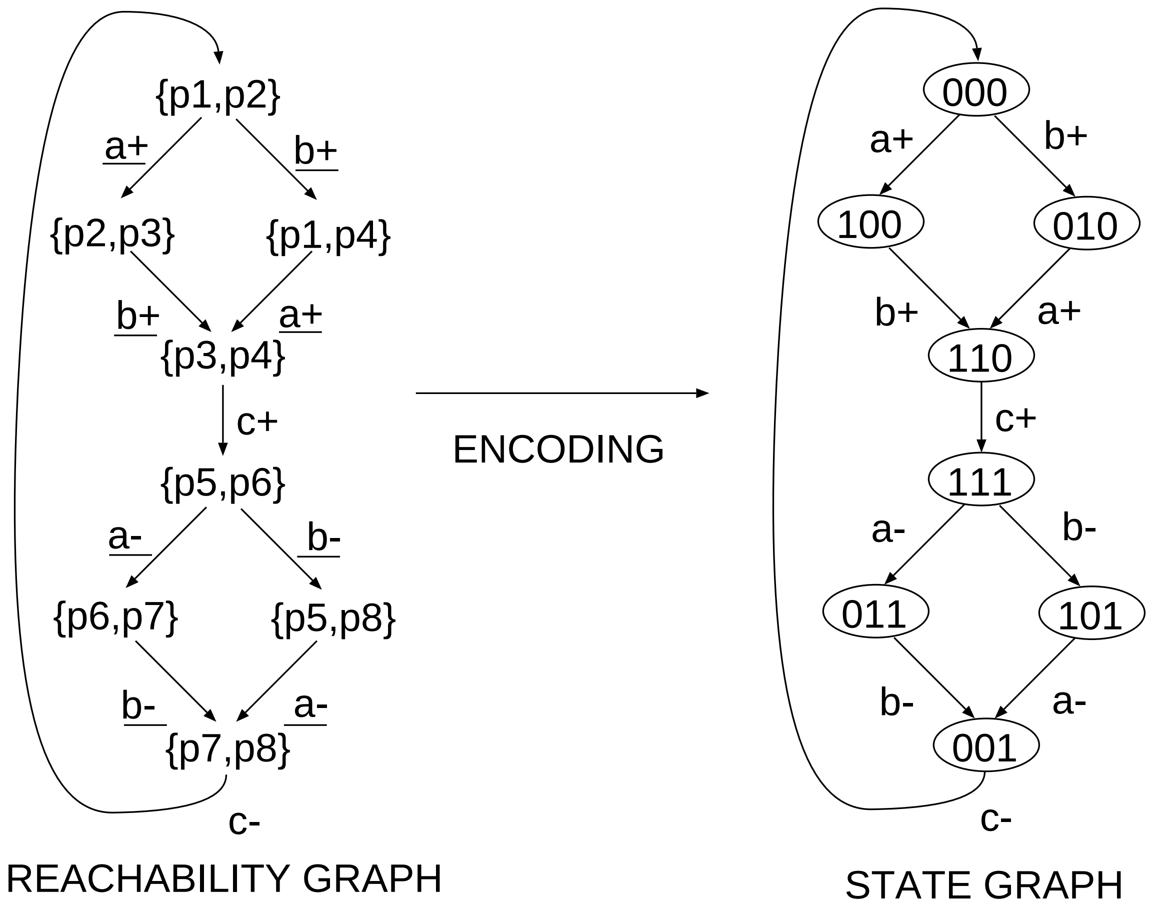
REACHABILITY GRAPH

- Vertices represent states the circuit can be in.
- Arcs represent transition from one state to another.
- Can be derived from the Petri net by simulating the token flow.
- Easiest representation to do analysis on.
- Size may be exponential in the number of transitions.
- Similar to trace structures [Dill], transition diagrams [Muller].

REACHABILITY GRAPH(2)

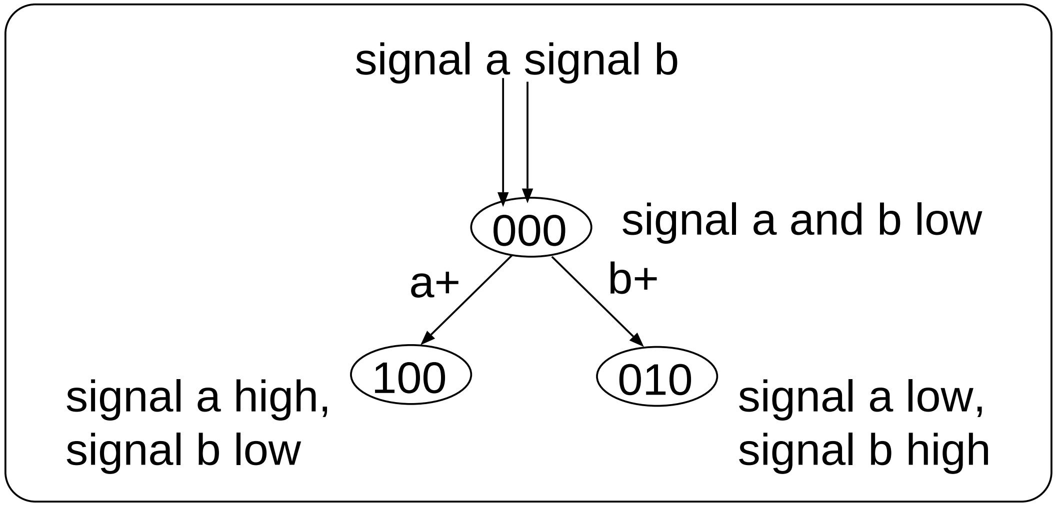


STATE GRAPH



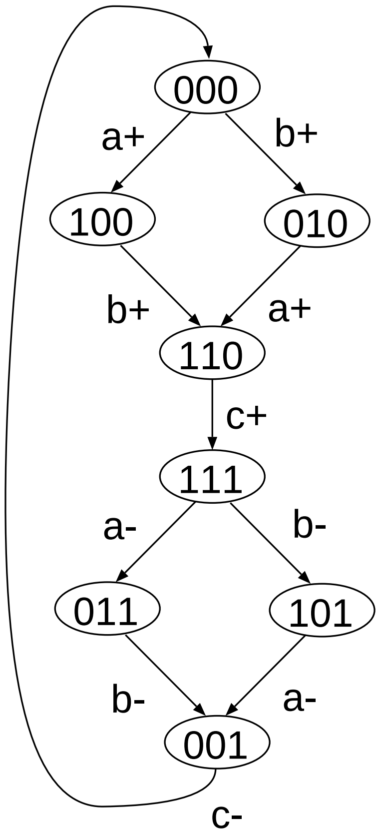
{p1,p2}
a+
b+
{p2,p3}
{p1,p4}
b+
a+
{p3,p4}
c+
{p5,p6}
a-
b-
{p6,p7}
{p5,p8}
b-
a-
{p7,p8}
c-
REACHABILITY GRAPH
ENCODING
000
a+
b+
100
010
b+
a+
110
c+
111
a-
b-
011
101
b-
a-
001
c-
STATE GRAPH

ENCODING



- Every state is assigned a code.
- This code is determined by signals in the *Petri net*.
- Based on this code, logical equations can be derived.

EQUATIONS



STATE GRAPH

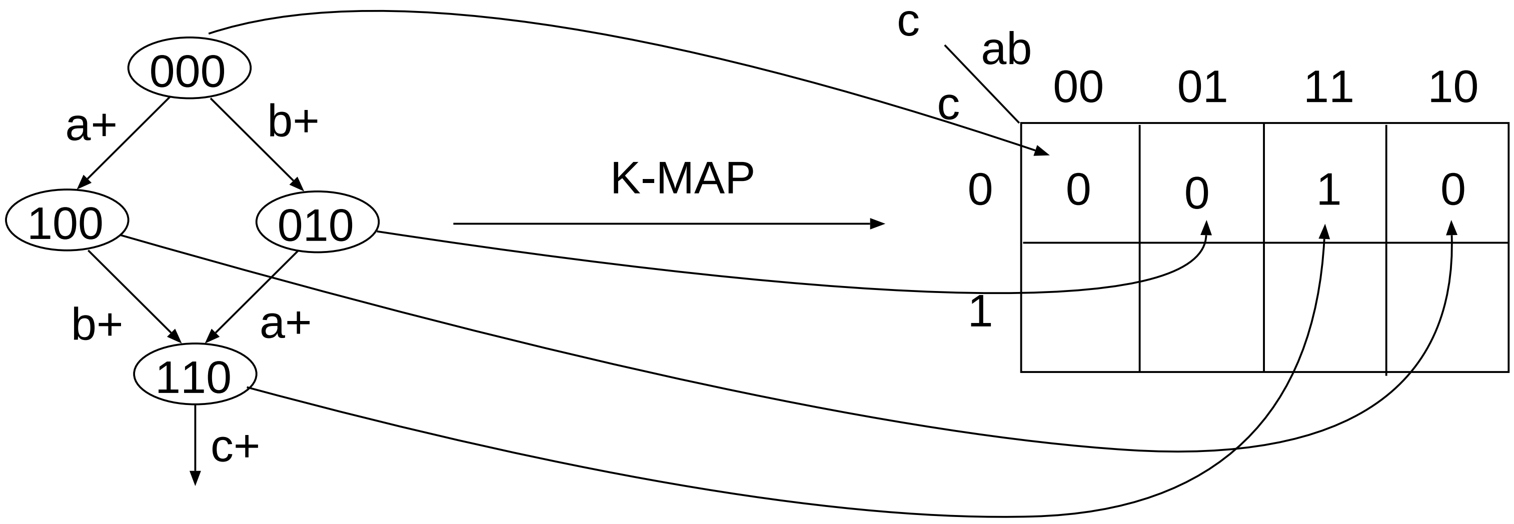
K-MAP

c \ ab	00	01	11	10
0	0	0	1	0
1	0	1	1	1

LOGIC EQUATIONS

$$c = ab + ac + bc$$

EQUATIONS(2)

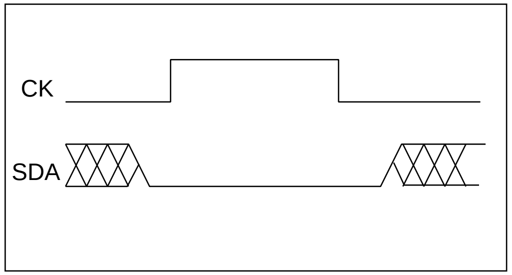


- State 000: c is assigned 0 because c stays low in this state.
- State 011: c is assigned 1 because c goes high in this state (c^+ is enabled).

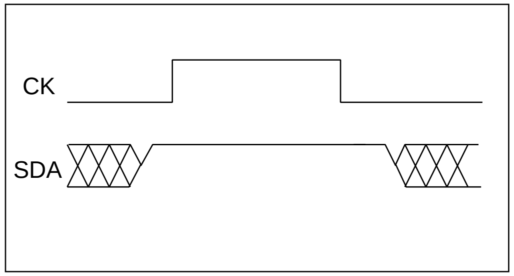
LIMITATIONS

- The advantages of STG.
  - Concurrency of signal transitions.
  - Conditional behavior based on signal transitions.
- The disadvantages of STG.
  - Conditional behavior based on signal levels.
  - No timing information just relative ordering of events.
  - No don't care behavior.
  - No or-node behavior.

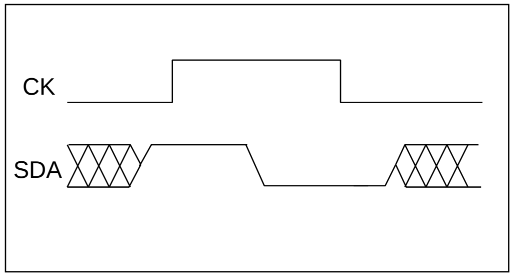
LEVELS AND TRANSITIONS



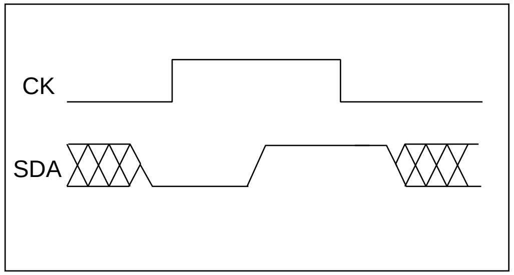
0 bit



1 bit



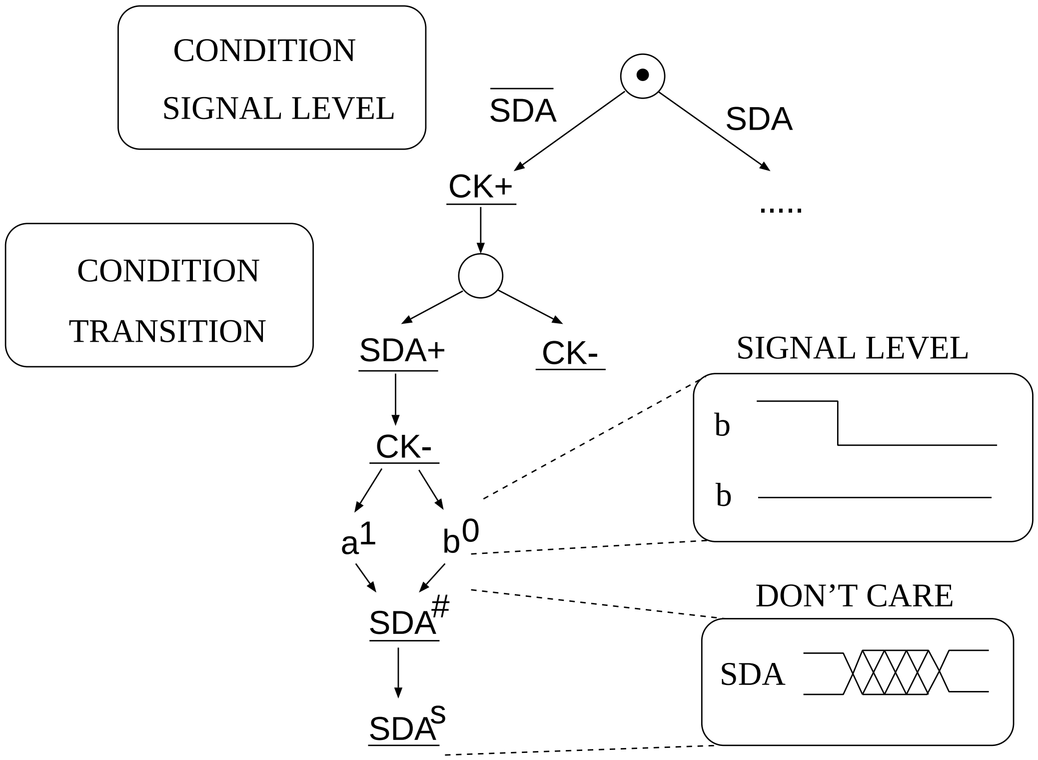
stop command



start command

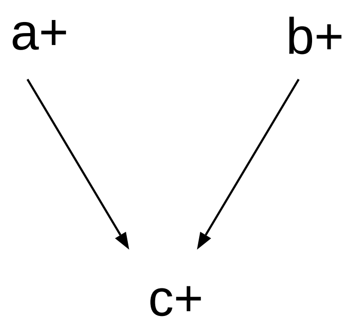
Protocol distinguishing different situations.

STG WITH EXTENSIONS

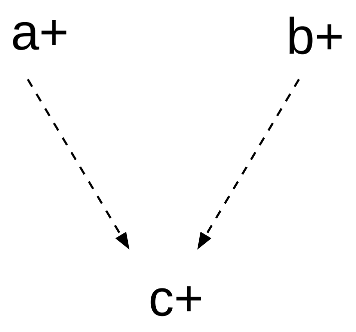


- extensions proposed by [Moon et al]

OR PRECEDENCE



c goes high after a and b have gone high

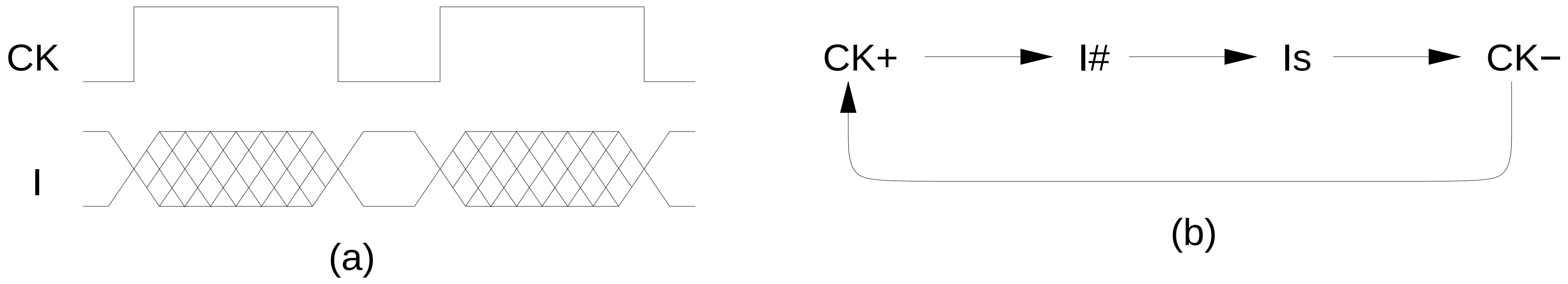


c goes high after a OR b have gone high

NO CHOICE!!

- This behavior is hard to model with classical Petri nets.

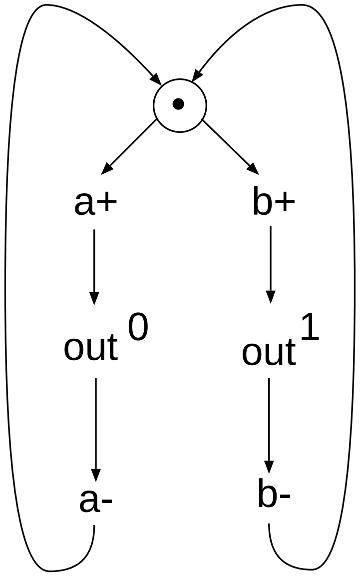
DON'T CARE & UNDEFINED BEHAVIOR

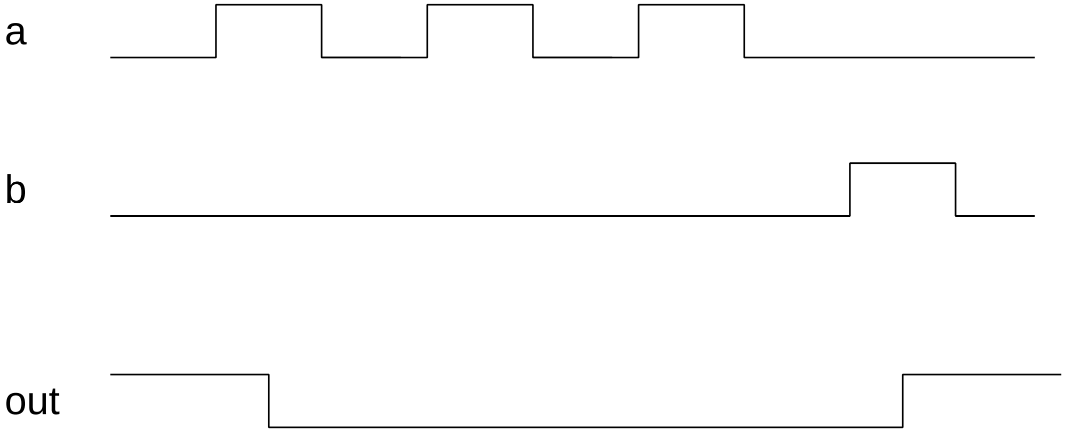


- $I^{\#}$: signal I becomes unstable.

- I^{s}: signal I becomes stable. The level is unknown.

LEVEL TRANSITIONS



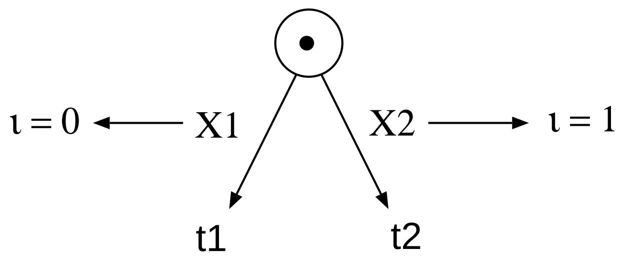


- Pulse on a sets out to 0, Pulse on b sets out to 1.
- Pulses may come in any order.

BOOLEAN GUARDS

- X_1, X_2 are Boolean characteristic functions.

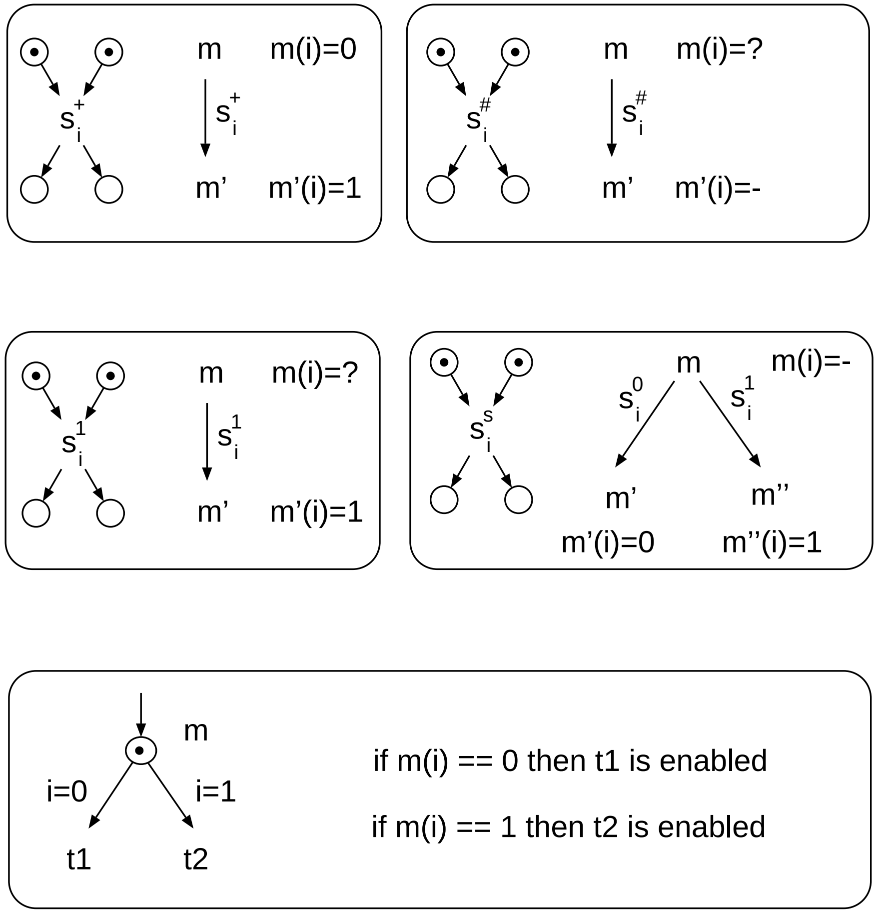
- Functions determine (together with tokens) which transition is enabled.



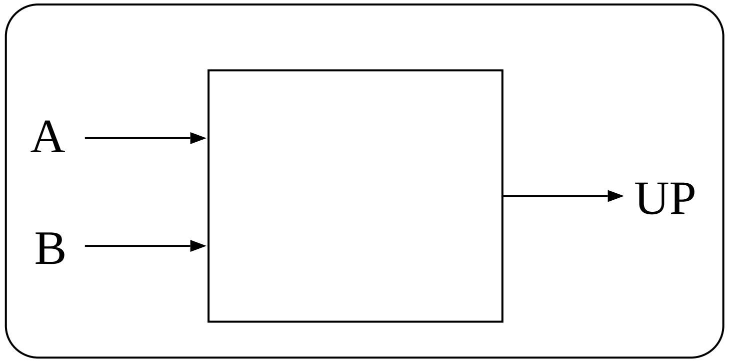
- t_1 is enabled if p contains a token AND signal $i = 0$.

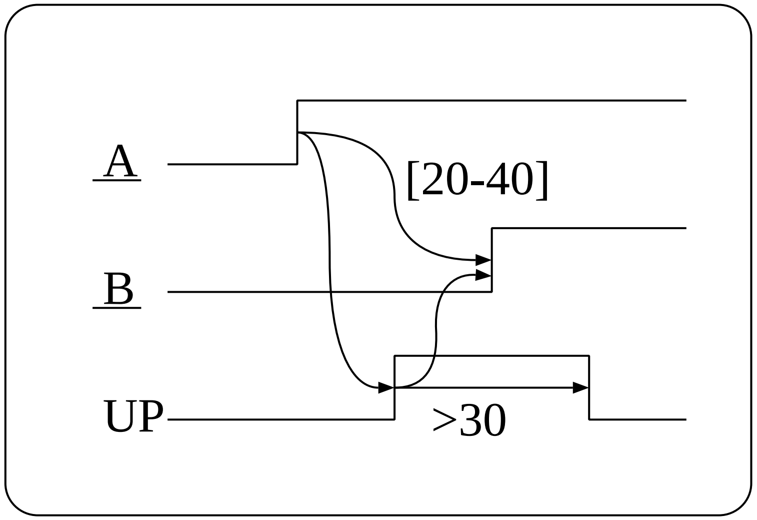
- t_2 is enabled if p contains a token AND signal $i = 1$.

TRANSLATION TO STATE GRAPH



TIMING: SPECIFICATION

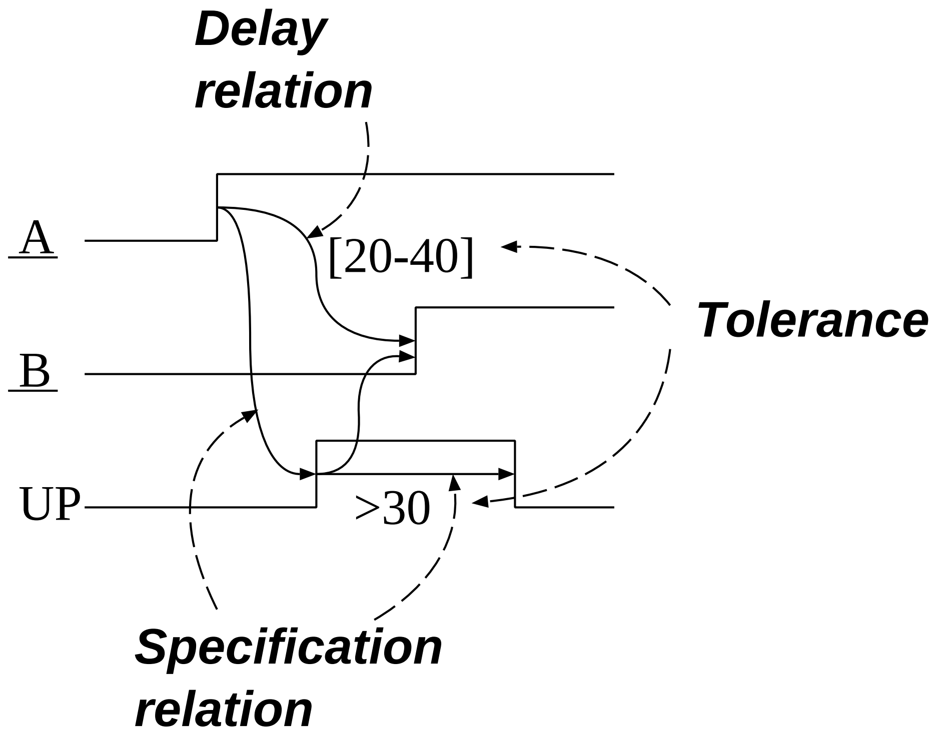




- Environment: if A goes high, B will go high 20 to 40 ns. later.

- Circuit: if A goes high UP needs to go high **before** B and needs to stay high at least 30 ns.

TIMING: SPECIFICATION(2)



TIMING: SPECIFICATION(3)

- Specification relations indicate how the circuit should react, if synthesized correctly.

- Delay relations indicate how an existing circuit (or environment) reacts.

- Tolerance: delays are not fixed numbers, but intervals (even more complex symbolic representation is possible).

TIMING CONSTRAINTS

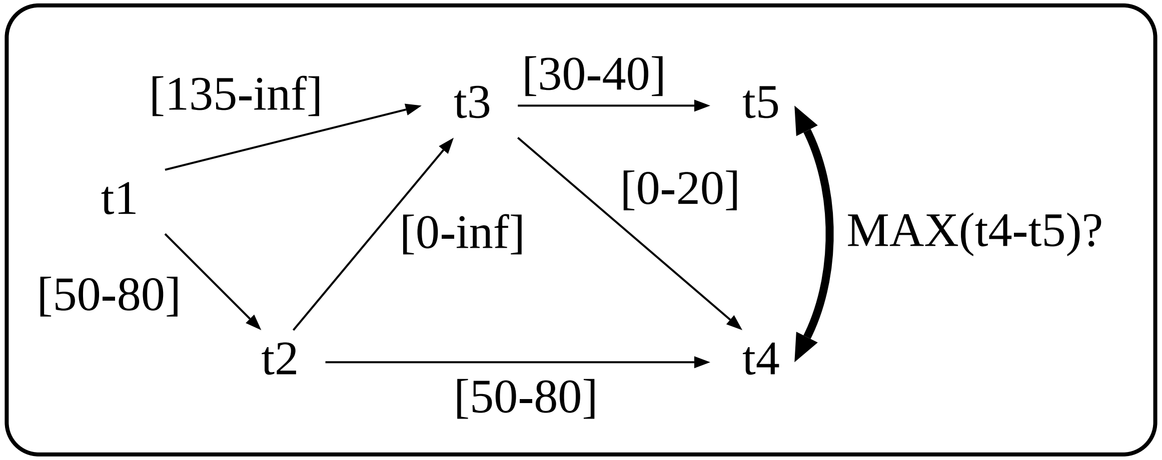
- Analysis of timing constraints [Rokicki, Hulgaard, Burns, Dill, Berthomieu, Amon, Walkup].

- Take into account analysis results [Myers].

- Active synthesis to satisfy timing constraints [Borriello, Vanbekbergen].

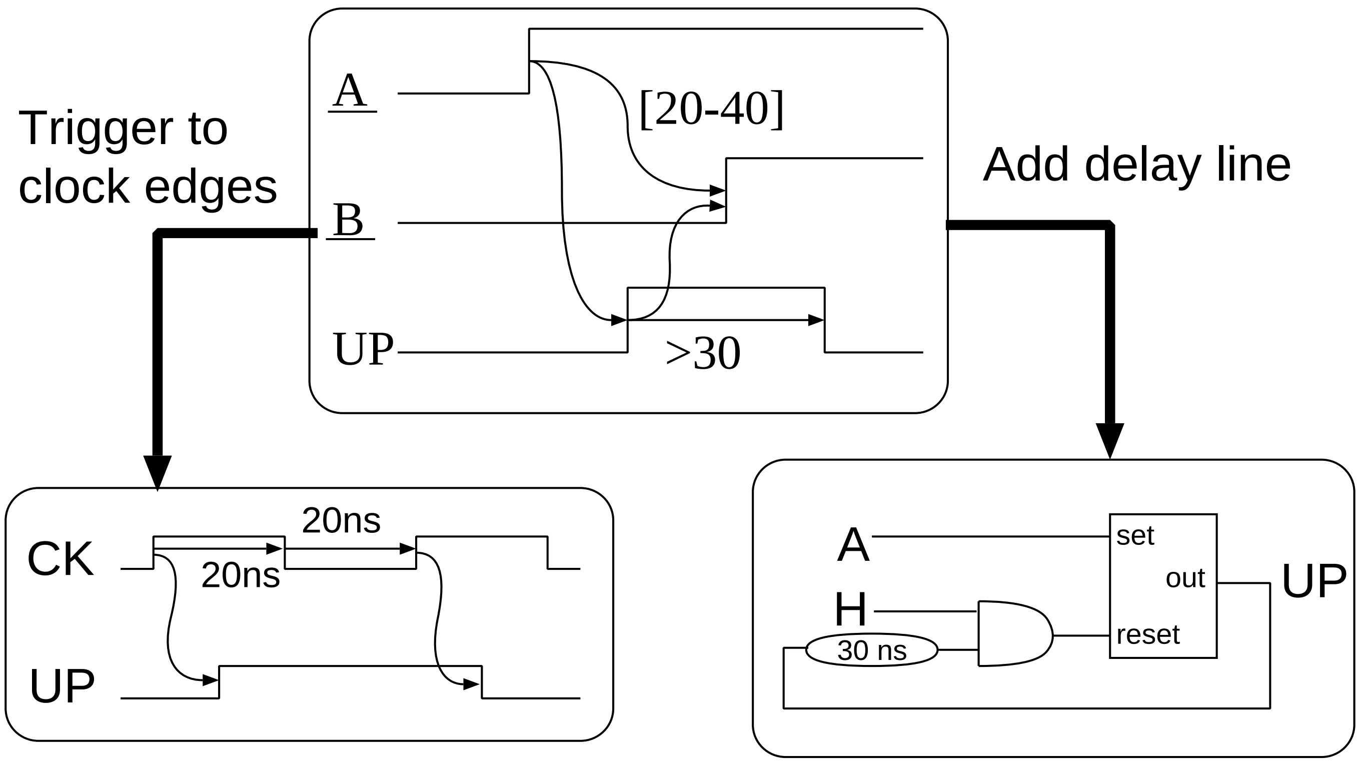
- Still a lot of research needed in this area.

TIMING: ANALYSIS



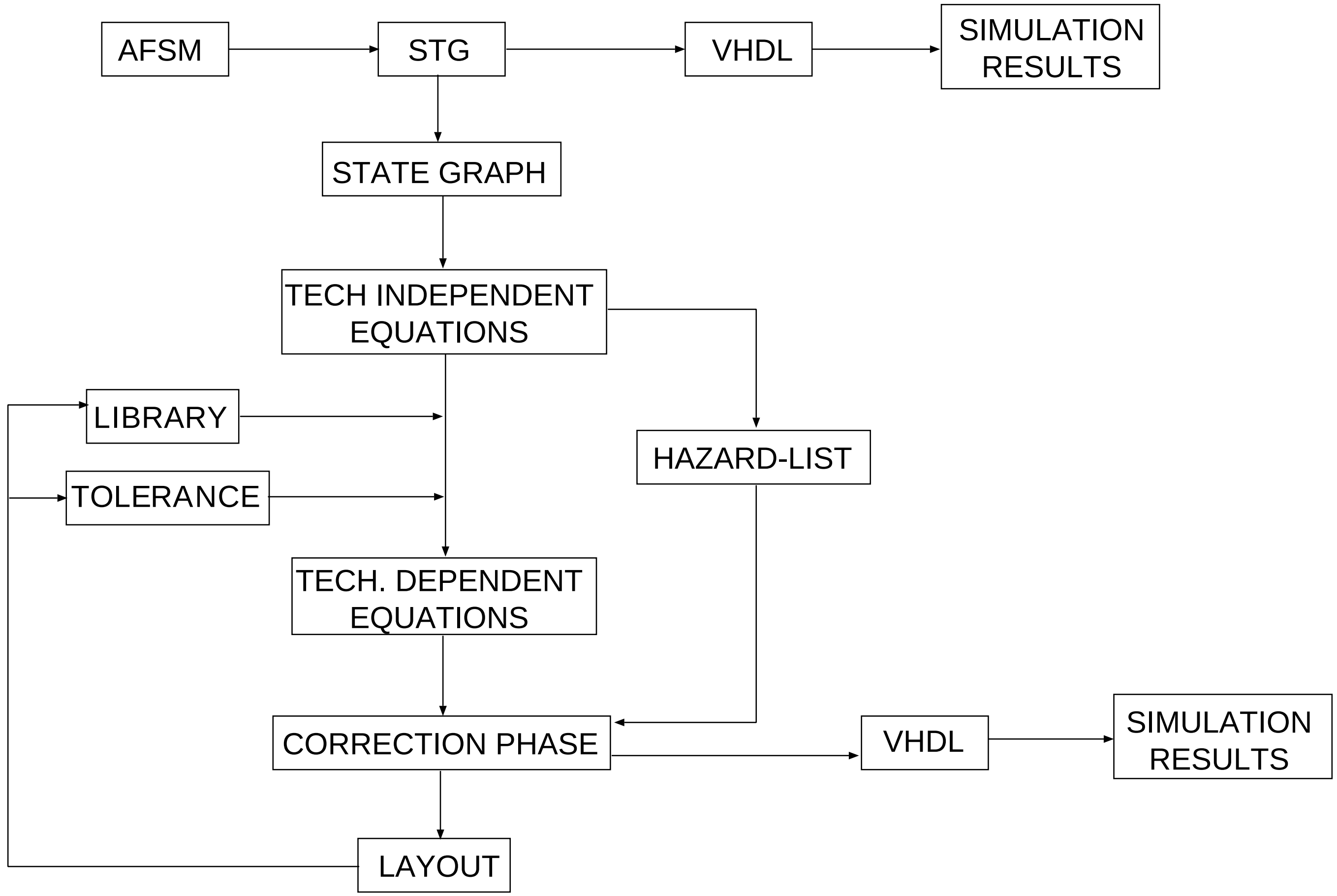
- Hard problem in its full generality [Berthomieu].
- Efficient solution by Rockiki (but might still be exponential).
- Polynomial solutions with various restrictions [Dill, Amon].

TIMING: SYNTHESIS



- Solutions with varying restrictions [Borriello, Vanbekbergen].

SYNTHESIS: OVERVIEW



SYNTHESIS: OVERVIEW(2)

- Simulation at high level (specification) and low level (circuit)
- Start from AFSM or STG
- Race-elimination by state graph transformation
- First step hazard-elimination (technology independent)
- Second step hazard-elimination (technology dependent)

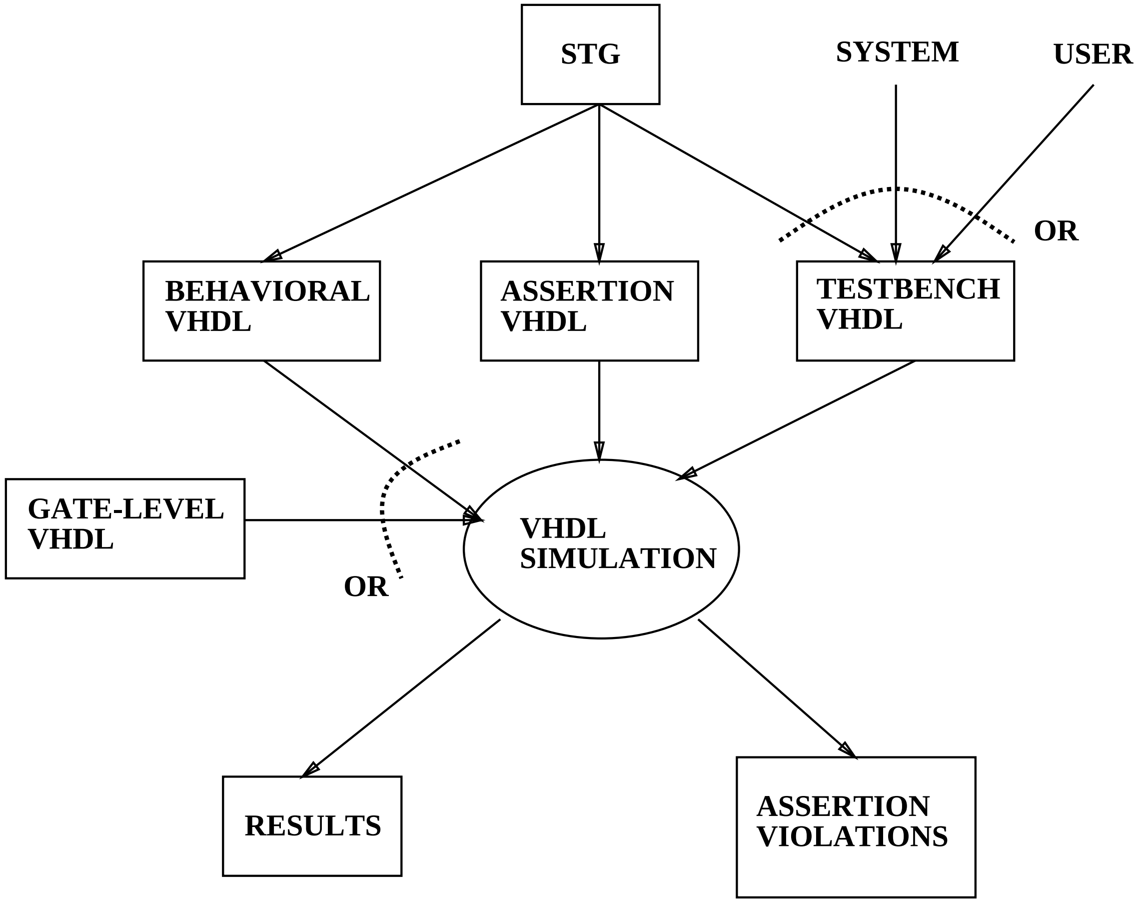
SYNTHESIS: OVERVIEW(3)

- Map to any library. The library may contain special elements to get better results, but synthesis should not rely on it.
- Library elements need to be hazard-free.
- Tolerance factor indicating how much delays of cells in library may vary.
- Back-annotation.

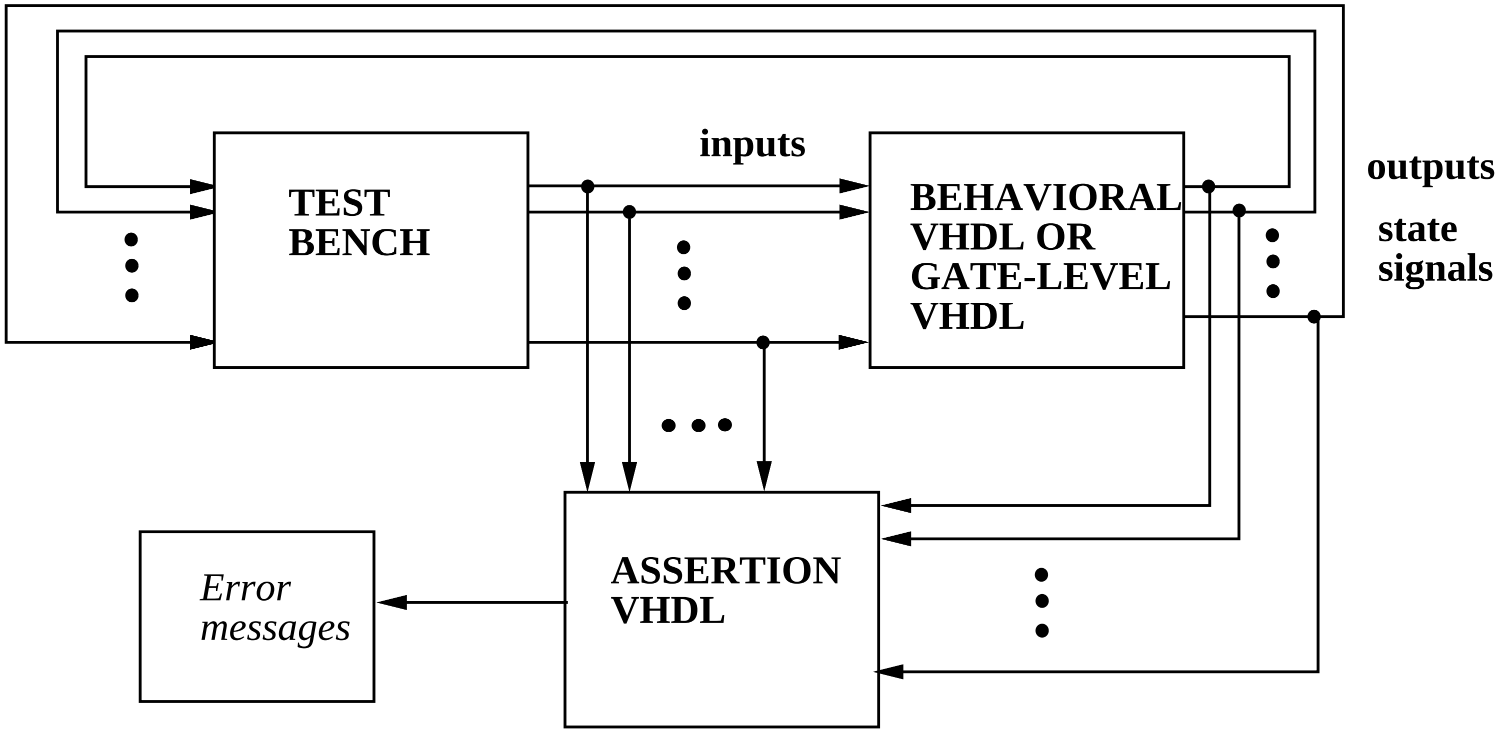
WHY VHDL?

- The need for a consistent language for all digital applications (synchronous and asynchronous).
- Tools (simulators) for free.
- The *complete* design can be validated at different levels of abstraction.
- Useful for debugging software prototypes ...

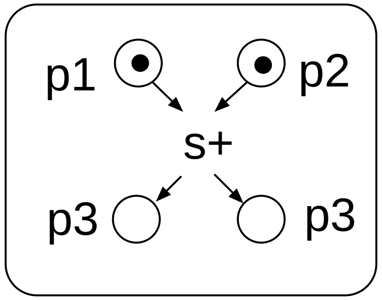
SIMULATION ENVIRONMENT



ARCHITECTURE

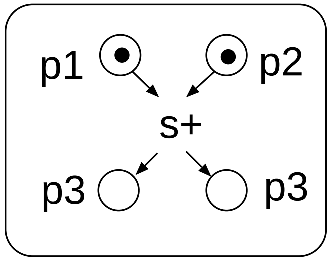


VHDL MODEL



- State = set of tokens in places
- A vector of Booleans: places
  - $places(1) = 1$ if place p_1 contains a token.
  - $places(1) = 0$ if place p_1 does not contain a token.

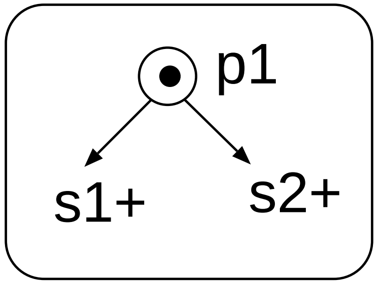
VHDL MODEL(2)




```
BLOCK (places(1) = '1' AND places(2) = '1')
BEGIN
     places(1) <= GUARDED '0';
     places(2) <= GUARDED '0';
     places(3) <= GUARDED '1';
     places(4) <= GUARDED '1';
     s <= GUARDED '1';
END;
```

- Every transition possibly concurrent with any other transition
- ONE block per transition
- Token flow modeled inside the block

CONDITIONS IN VHDL



pcond controls conditional execution

enables a different transition for each "pass"

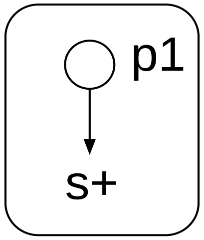
```
BLOCK (places(1) = '1' AND pcond = '0')
BEGIN
    places(1) <= GUARDED '0';
    s1 <= GUARDED '1';
    pcond <= GUARDED '1';
END;

BLOCK (places(1) = '1' AND pcond = '1')
BEGIN
    places(1) <= GUARDED '0';
    s2 <= GUARDED '1';
    pcond <= GUARDED '0';
END;
```

ASSERTION

- All blocks are possibly concurrent.
- In VHDL a resolution function is needed.
- STG is only correct if no two transitions of the same signal are enabled at the same time.
- Assertion in the resolution function that there are no two simultaneous assignments to the same signal.

ASSERTION(2)



ASSERT (s'STABLE = TRUE OR places(1) = '1' OR ....)

- Gives info about
  - where?
  - which signal?
  - which type of violation?

- Debugging circuit AND environment

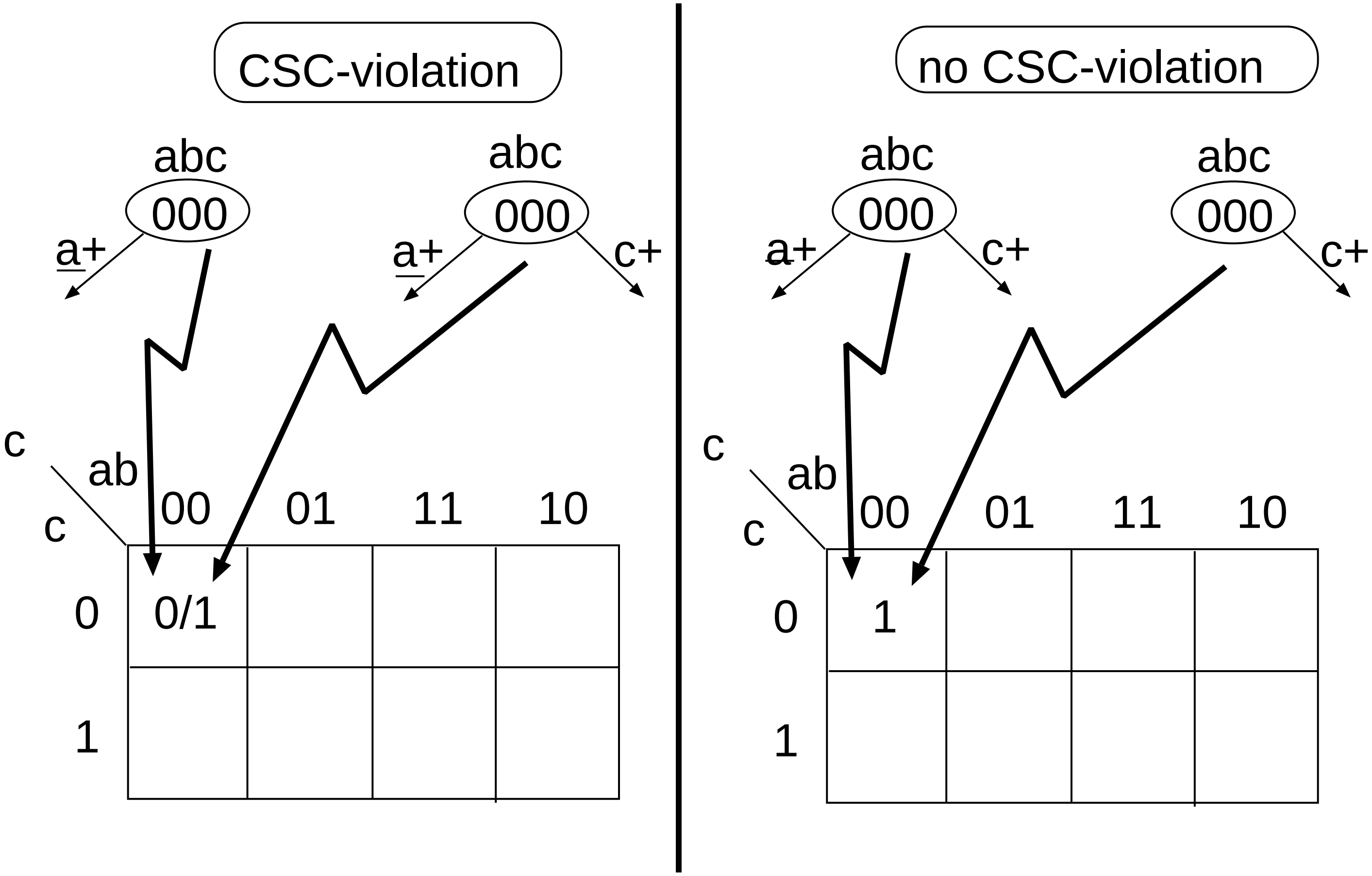
TESTBENCH

- Testbench = behavioral VHDL of "inverse" STG (inputs and outputs switched).
- Use of same procedure as the one generating behavioral VHDL (not written by designer)
- Assumes that the environment is the fastest possible.
- Not complete (like verification), but many errors found this way.

RACE: MODELING

- Races are most easily detected at state graph level.
- No race if the logic equations can be derived unambiguously.
- Sufficient condition:
  All states are assigned a unique code .
  Unique State Coding (USC) .
- A less tight sufficient condition:
  Complete State Coding (CSC) .

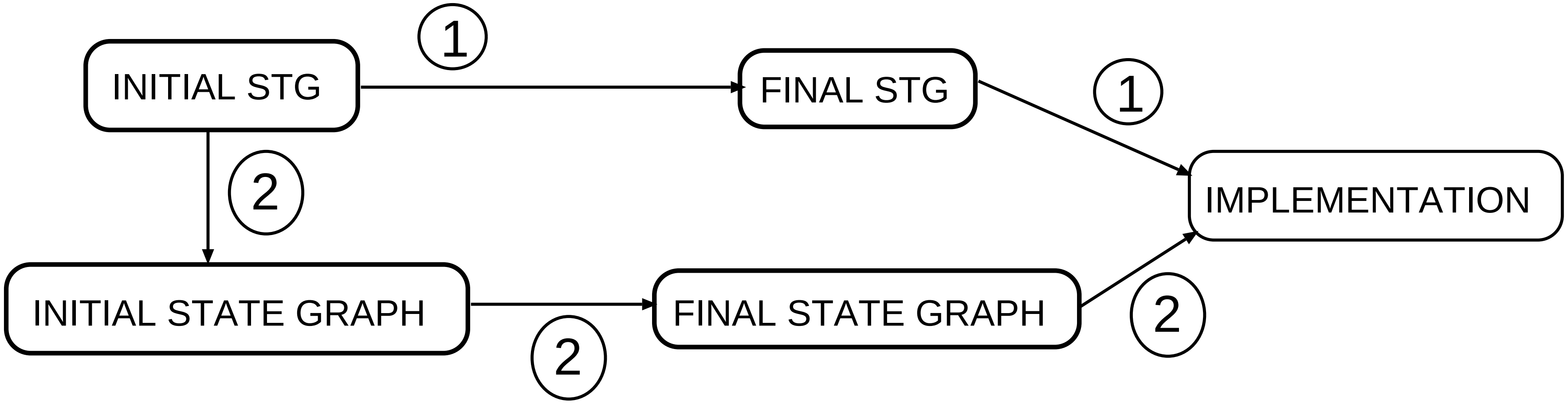
COMPLETE STATE CODING



RACE ELIMINATION

- Race-elimination is enforcing CSC (or USC).

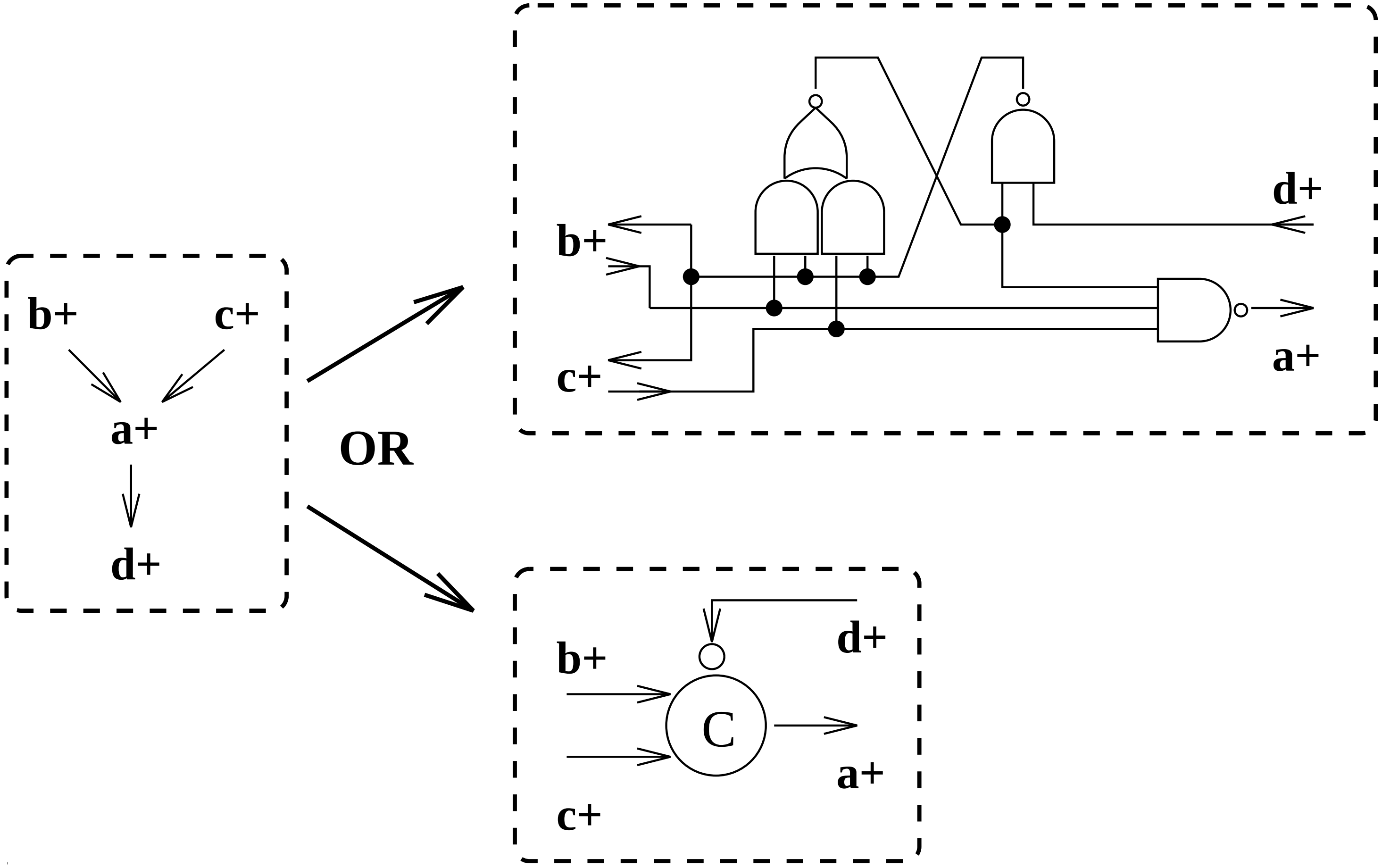
- Which route to take?



RACE ELIMINATION(2)

1. STG-LEVEL [Kondratyev, Yakovlev, Lin, Pastor]
   - Avoids the state graph level (run-time efficiency).
   - Circuits not always efficient [Kondratyev] or restrictions on type of specification handled [Yakovlev, Lin, Pastor].
2. STATE GRAPH LEVEL [Lavagno, Ykman, Puri, Vanbekbergen]
   - Can be very computation intensive.
   - Very general technique which can handle more general specifications than (1).

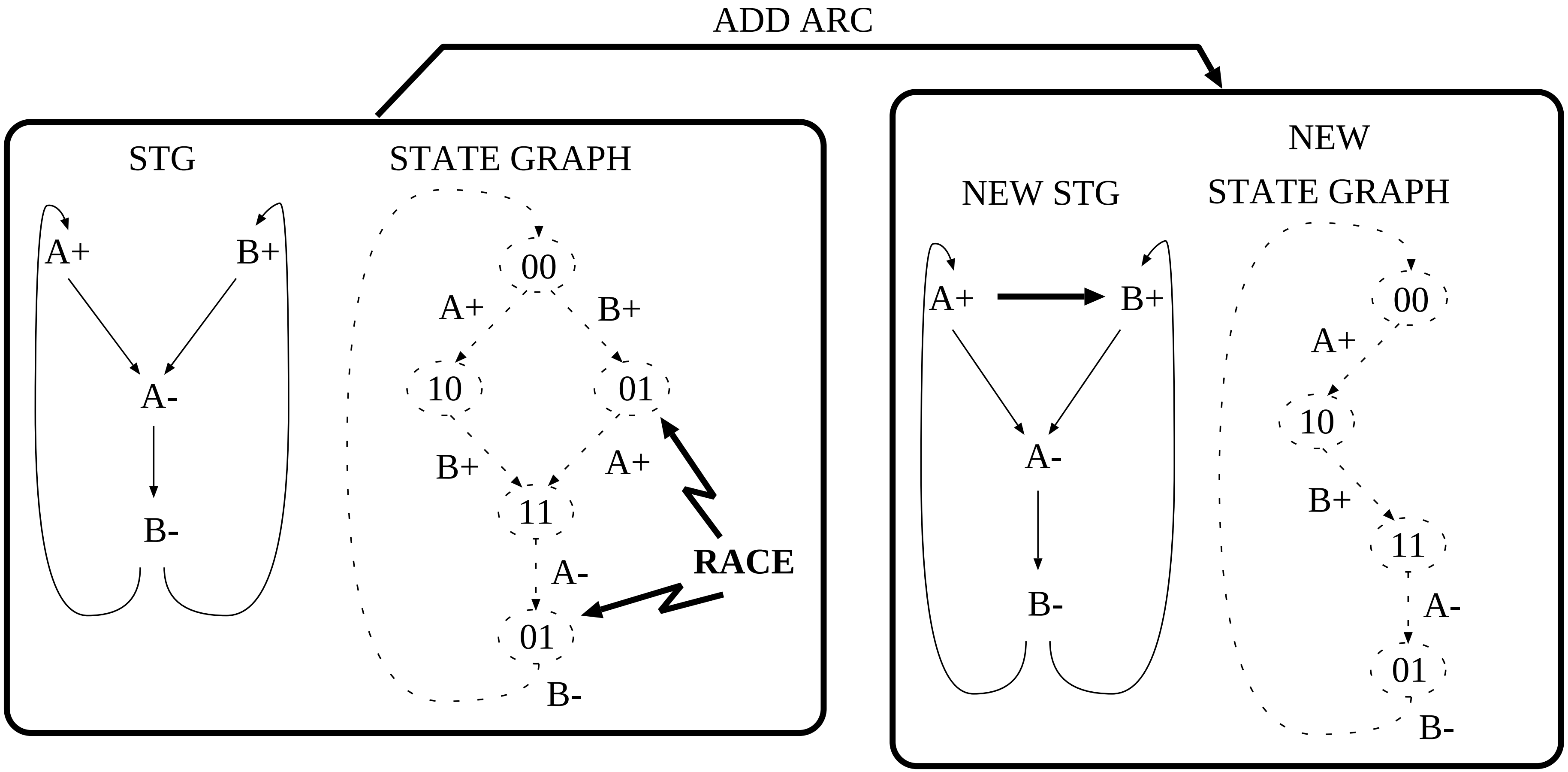
DIRECT METHOD



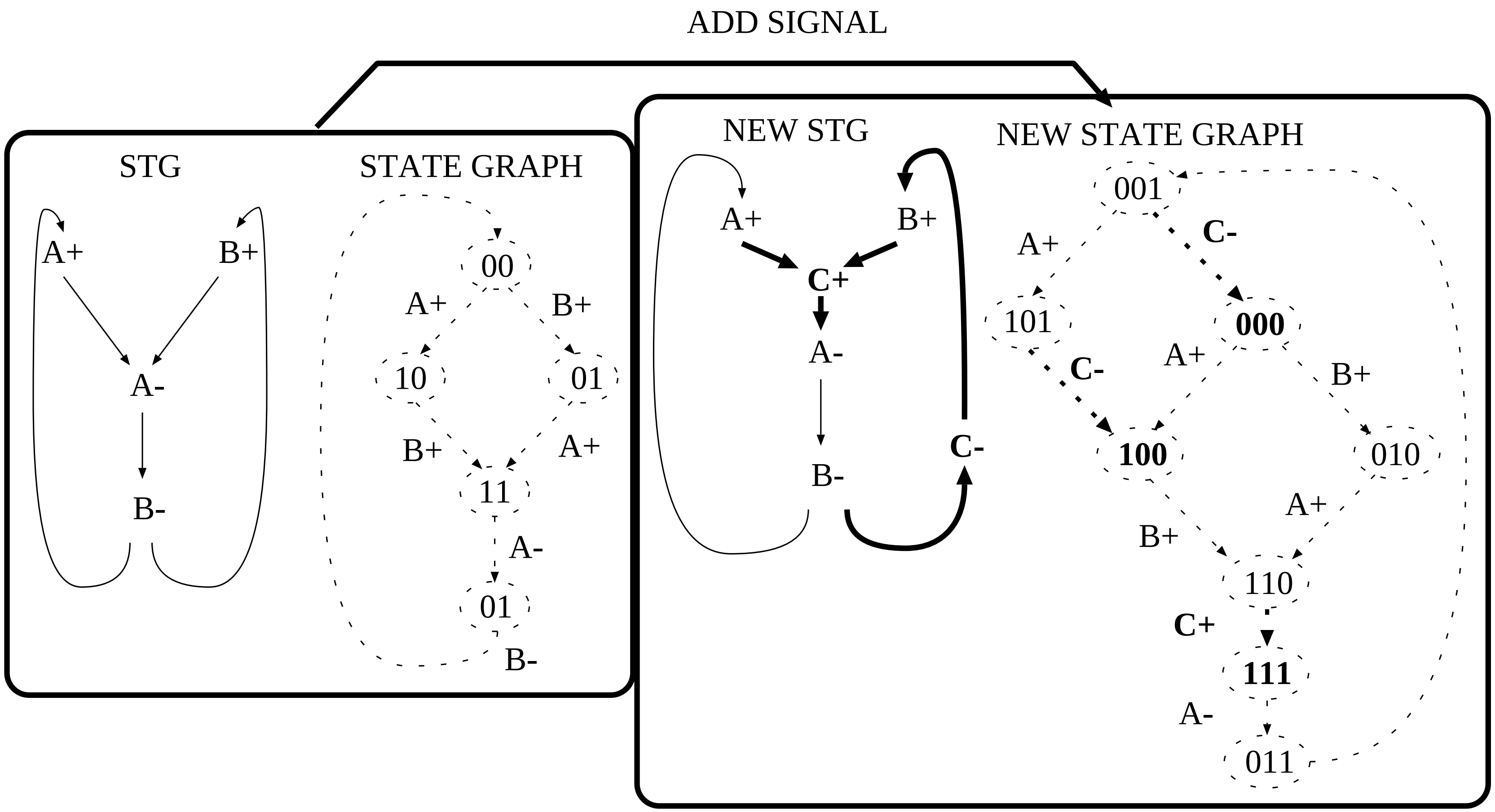
STG-LEVEL METHOD

- Works directly at STG-level and thus avoids state-explosion
- Restrictions:
  - No choices
  - Every signal has one up- and down-transition [Yakovlev]
- Extensions:
  - Choice [Lin, Pastor]
  - Signal may have more than one up- and down-transition [Lin, Ykman, Pastor]

STG-LEVEL METHOD(2)



STG-LEVEL METHOD(3)

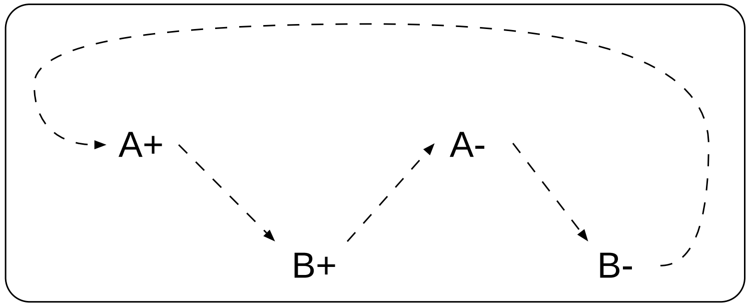


LOCK RELATION

- The main problem: we want to transform the STG to satisfy a property which is defined at state graph level.

- A theory is needed that transforms the CSC or USC property to the STG-level

Two signals A and B are locked
iff
the transitions of the two signals reside on a simple cycle in an alternating way

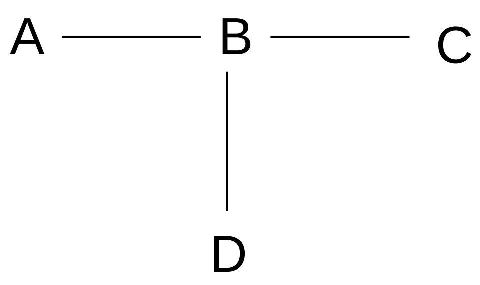


Corresponds to handshake protocol

LOCK GRAPH

The vertices correspond to the signals

There is an arc between two vertices if the corresponding signals in the STG are locked.



Lock Graph = connected

$\downarrow$

STG satisfies USC

LOCK THEOREM

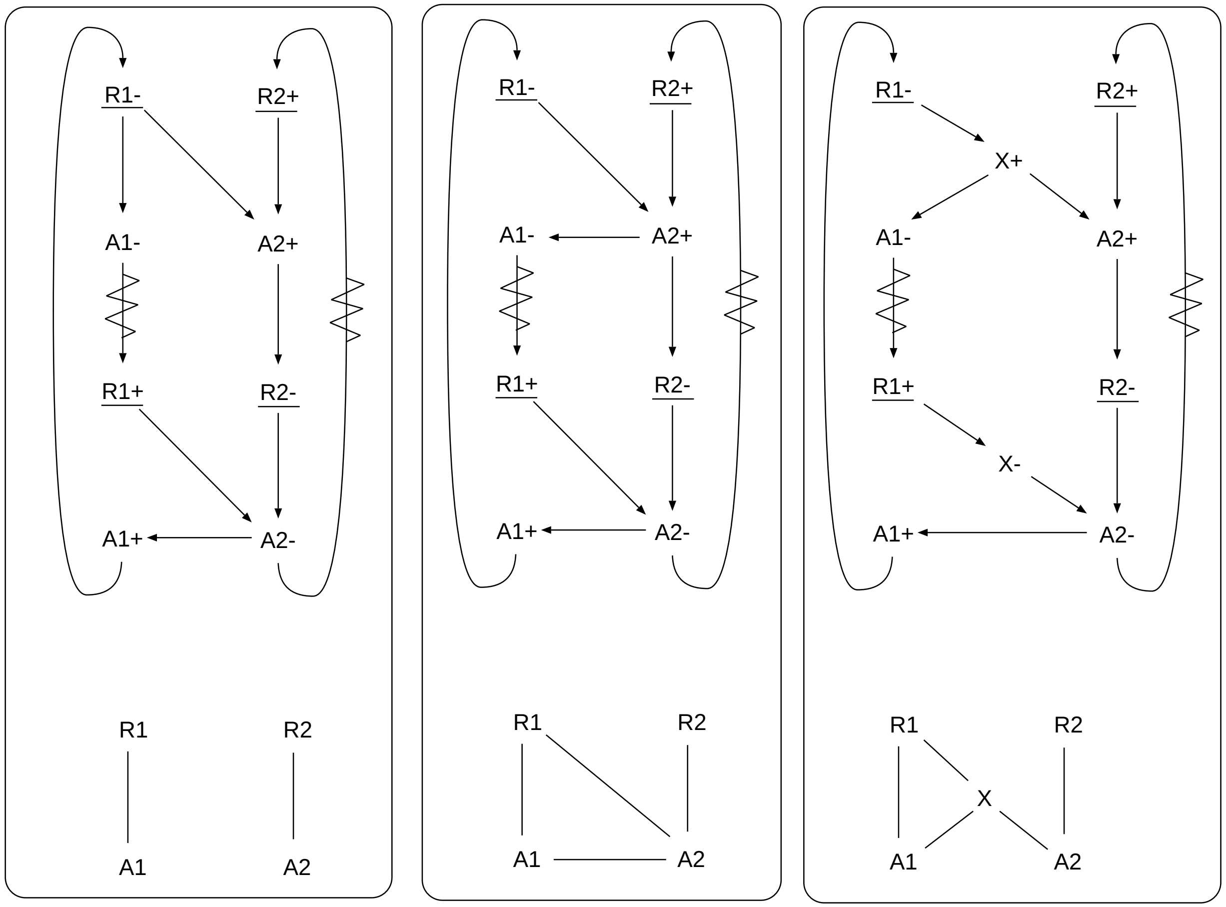
$$\begin{tikzcd}
\text{OLD STG} \arrow[r] \arrow[d, "\text{TRANSFORM}"] & \text{lock graph NOT connected} \\
\text{NEW STG} \arrow[r] & \text{lock graph connected}
\end{tikzcd}$$

- Adding arcs.
- Adding new signals (and transitions).
- If necessary, taking into account timing constraints.

INTUITION

- A complementary set is a set of transitions such that if a^+ is in the set then also a^- and vice versa.
- Firing a complementary set leads to states with the same code. (for instance $\{A_1^+, R_1^-, A_1^-, R_1^+\}$ in the next example (a)).
- The only allowed complementary set that you can fire is the set containing all transitions.
- Locking makes sure that no complementary subset of transitions is fireable.

LOCK EXAMPLE



STATE GRAPH LEVEL METHOD

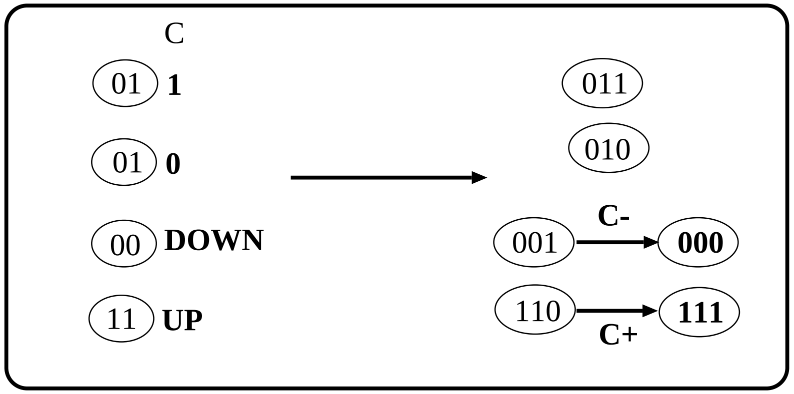
- Transforms the problem into a satisfiability problem.
- Very computation intensive.
- Very few restrictions:
  - General Petri nets (with all extensions).
  - General transformations (concurrent transitions).
- Partitioning [Puri] and Heuristics [Lavagno, Ykman] to improve efficiency of circuit and computation time.
- Concurrency reduction [Ykman].

STATE ASSIGNMENT

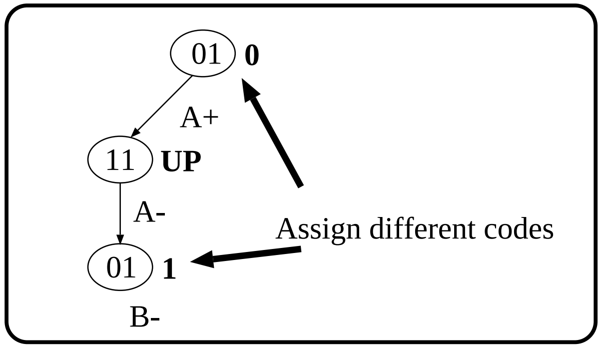
- "Expand" a state graph into a new state graph that satisfies CSC.
- 4-value assignment set: $\{0, 1, UP, DOWN\}$.
- Meaning of assigning a value to a state P for signal S.
  - P is assigned 0: S is low in P
  - P is assigned 1: S is high in P
  - P is assigned UP: P will be expanded in two states with S^+ connecting both states.
  - P is assigned $DOWN$: P will be expanded in two states with S^- connecting both states.

STATE ASSIGNMENT(2)

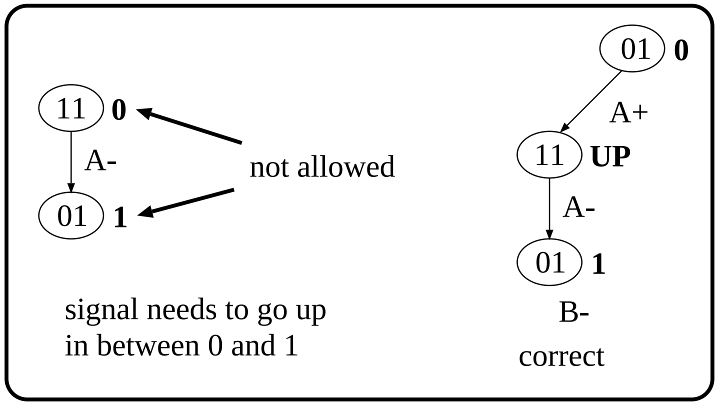
EXPANSION



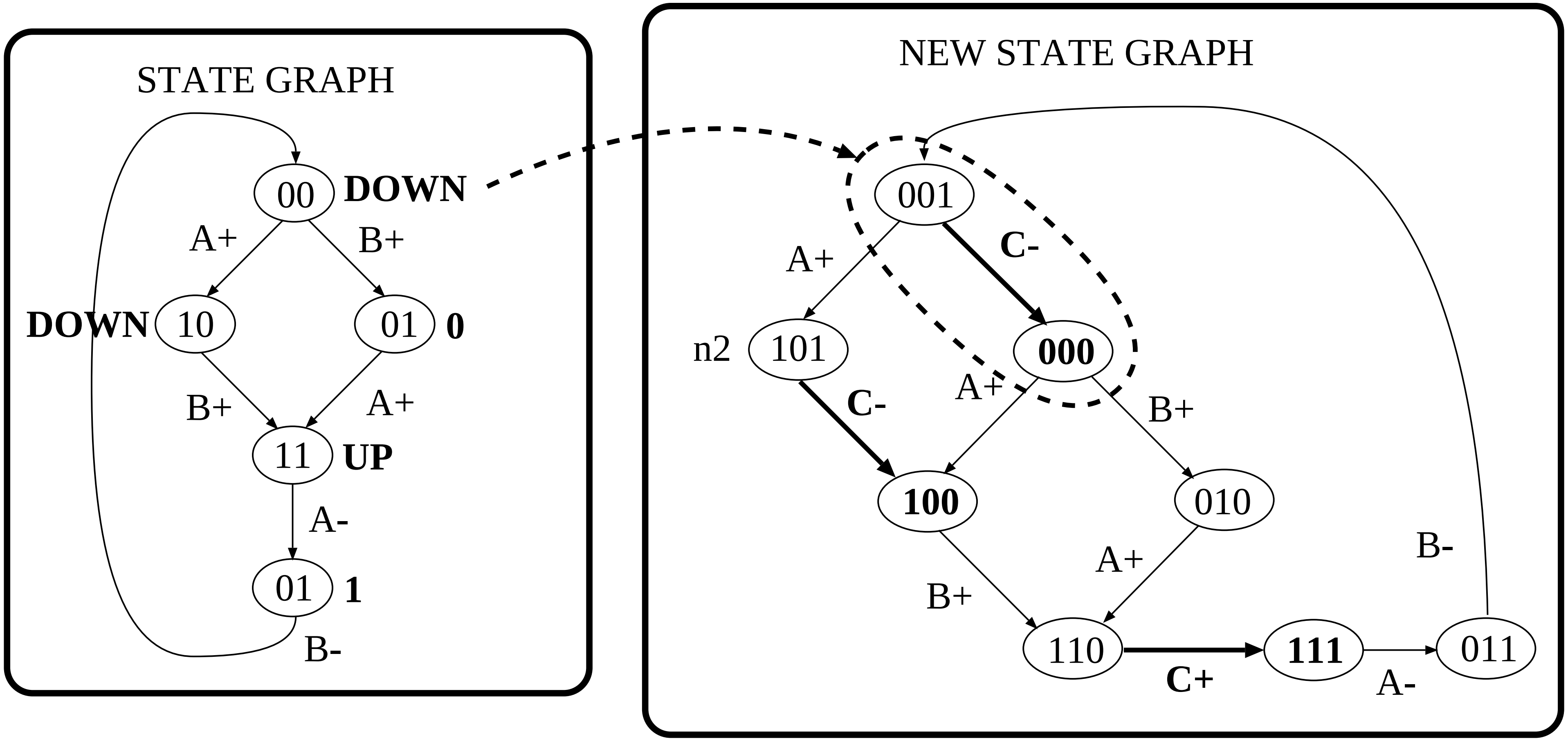
CSC-REQUIREMENT



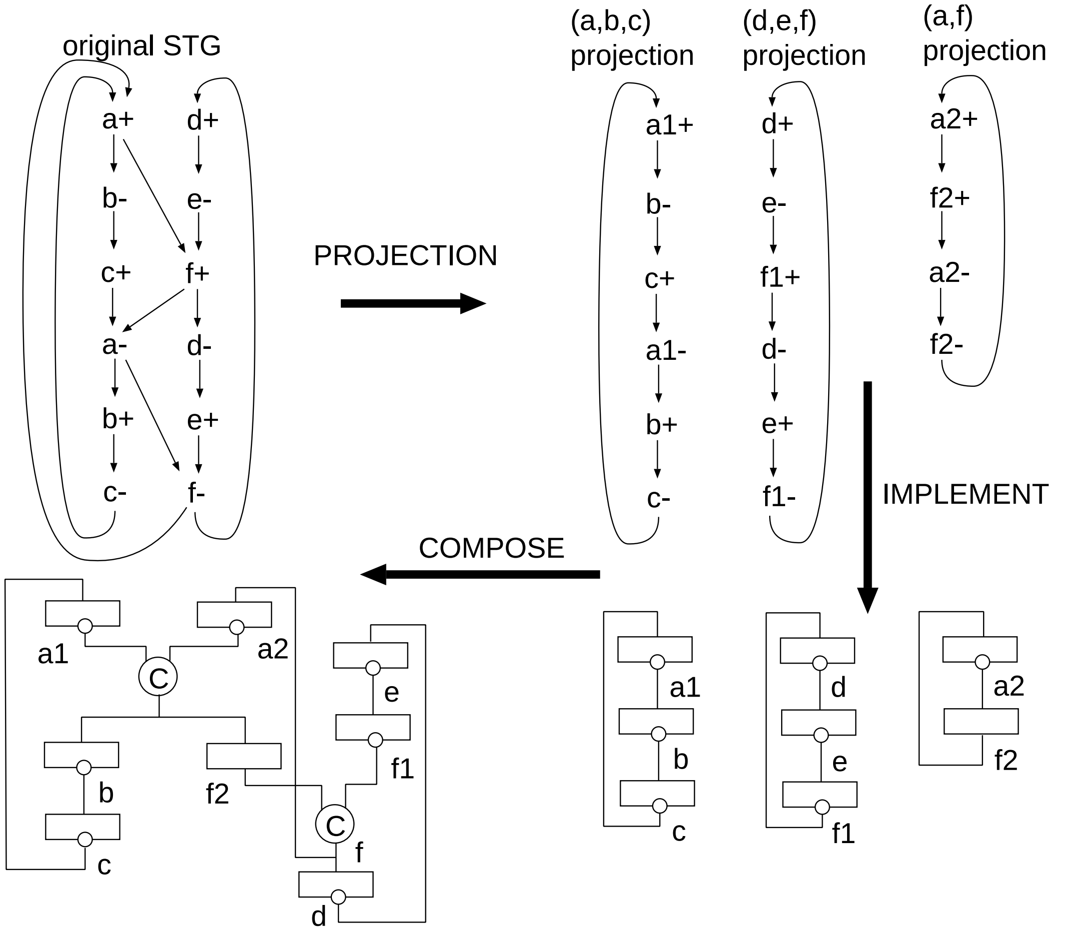
CONSISTENCY



STATE ASSIGNMENT(3)



DECOMPOSITION METHOD



Part 3: Hazard-free synthesis and testability

- Hazard-free synthesis based on logic transformations.
  Unbounded gate delays

- Hazard-free synthesis based on changing delays.
  Bounded wire delays

- Synthesis of non-deterministic asynchronous circuits.
  Meta-stability problem

- Testing

Hazard-free synthesis

- Input: STG (or SG) with CSC
- Output: logic circuit implementing the STG
  - correctly (no hazards, no races)
  - with minimal cost (area/delay/power/testability)
- Direct methods (syntax-driven from STG): fast but potentially suboptimal
- Synthesis-based methods

STG-based synthesis

- Absence of races guaranteed by construction (only STG signals can be fed back)

- Absence of hazards can be achieved:
  - by acknowledgment/indication (each gate has some other gate that "checks" when its transition is completed, to avoid hazards)
  - by insertion of appropriate delays

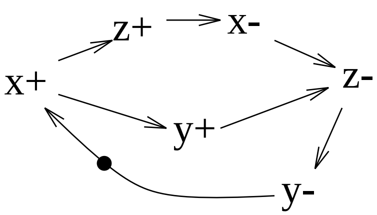
- Minimality is achieved by logic synthesis (special methods for two-level and multi-level implementations)

Synthesis with unbounded gate delays

- Muller (59): formal analysis method, but only design hints
- Martin (86, . . .): syntax-directed translation, then QDI-preserving optimization
- Varshavsky (90): syntax-directed translation from STG
- Beerel (92): improves with heuristic SG-level technique based on "monotonous switching" of cubes
- Kondratyev (93, . . .): necessary and sufficient conditions for monotonous switching, STG- and SG-level synthesis techniques

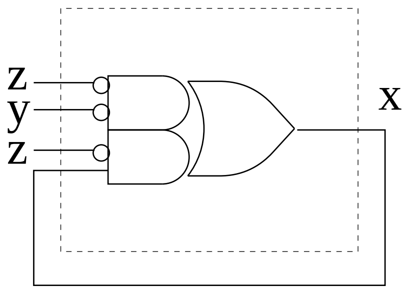
Driving and supporting gates

STG

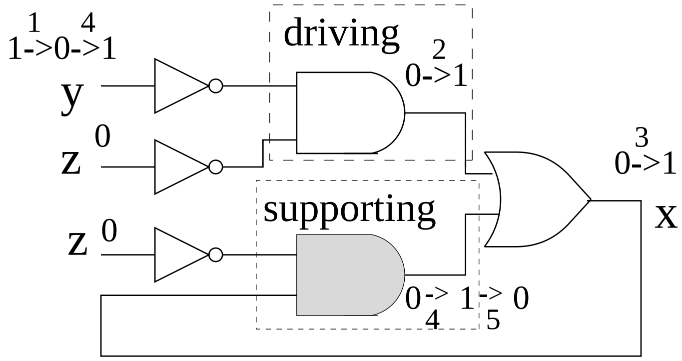


$x = \bar{z}\bar{y} + \bar{z}x$

Complex gate - no hazards



Simple gates - hazards

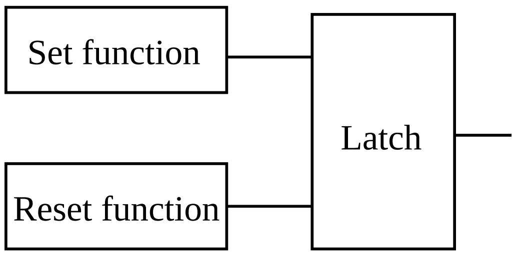


- All internal signal transitions must propagate to the output (acknowledgement principle applied to individual gates)

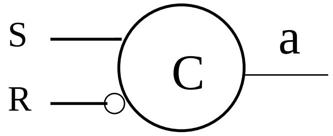
- Driving gates: “good”, supporting gates: “bad”

Implementation structures

Signal implementation



C-latch



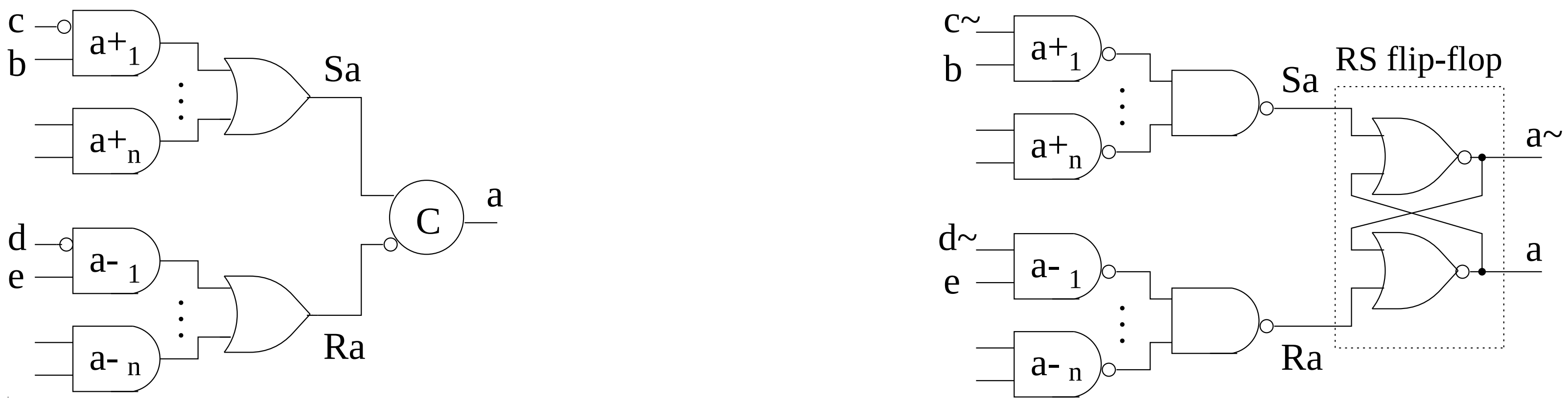
S=1, R=0 $\Rightarrow$ a+

S=0, R=1 $\Rightarrow$ a-

Each signal transition produces an AND gate

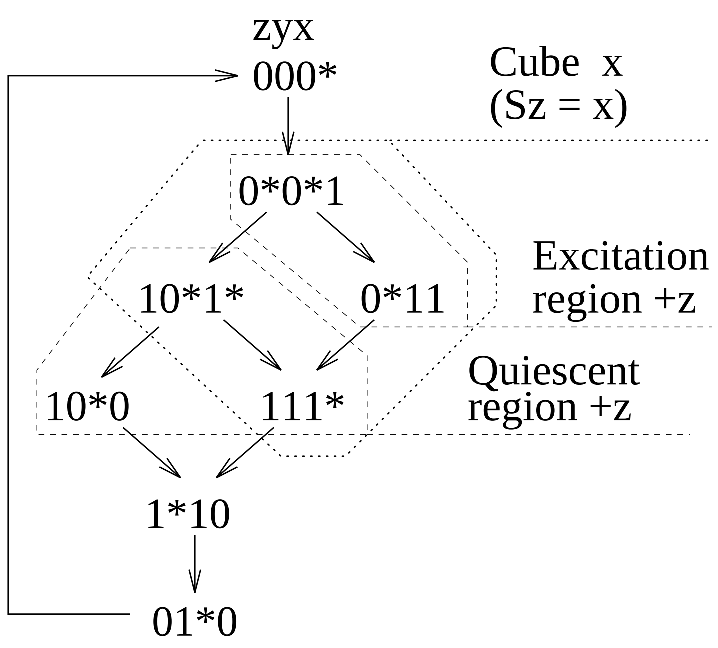
- $a+$ sets the latch
- $a-$ resets the latch

BASIC ARCHITECTURES



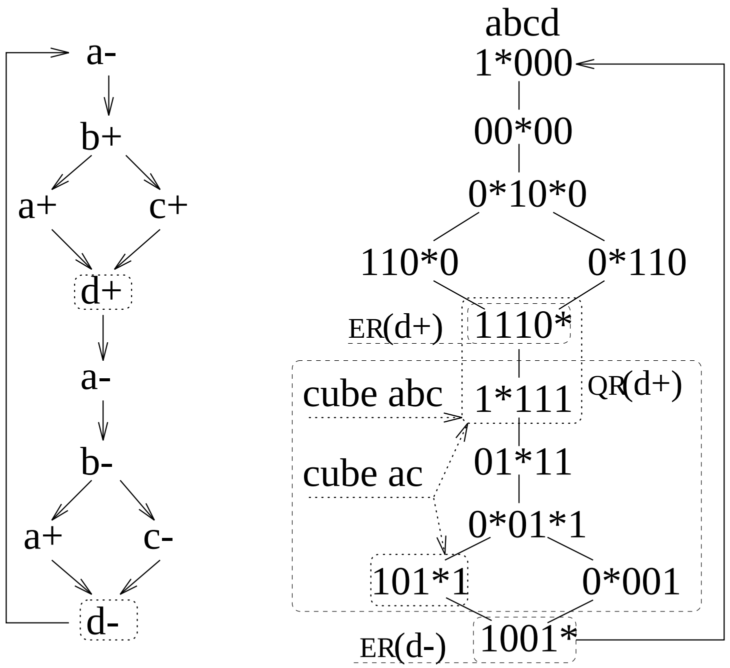
Regions of the state graph

- **Synthesis idea:** Set of concurrent transitions ⇒ cover cube in R or S function ⇒ driving gate



- Excitation region (ER) must be covered by **one** cube
- Quiescent region (QR) is a *don't care set* for a cover cube

Monotonous cover conditions

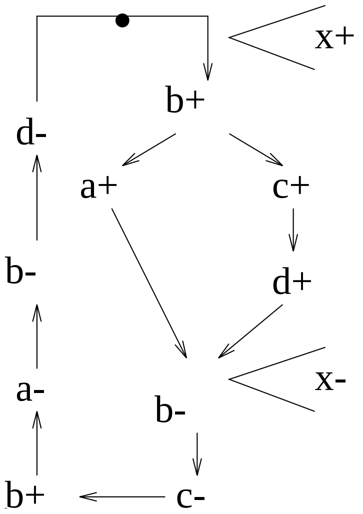


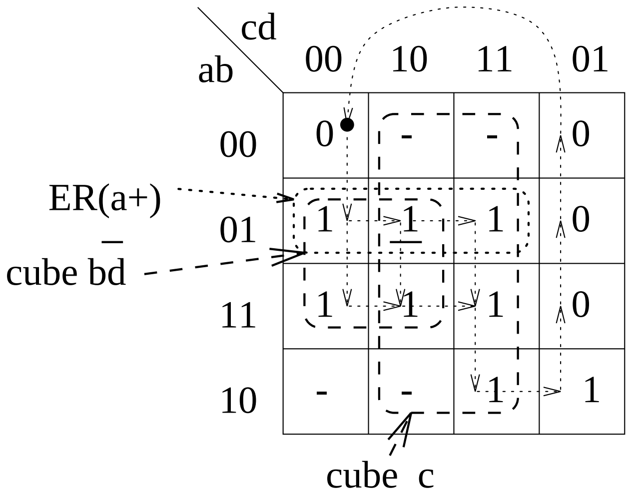
Cover cube C is a **monotonous cover** for $ER(a*)$ iff:

1. C covers all states $ER(a*)$
2. C covers no states outside $ER(a*) \cup QR(a*)$
3. C changes only once inside $QR(a*)$

- Monotonous cover $\Rightarrow$ SI-implementation
- Restricted (but useful and realistic) architecture

Reduction to monotonous cover form





- Cubes $\{b\,\overline{d},\ c\}$ form a correct non-monotonous cover of $ER(a+)$:
  - none covers all states of $ER(a+)$
  - due to non-persistency of c, d wrt $a+$
- After insertion of x: cube bx is a monotonous cover

- Externally observable behavior is preserved

- Sufficient condition: any MC violation can be eliminated with at most two added signals

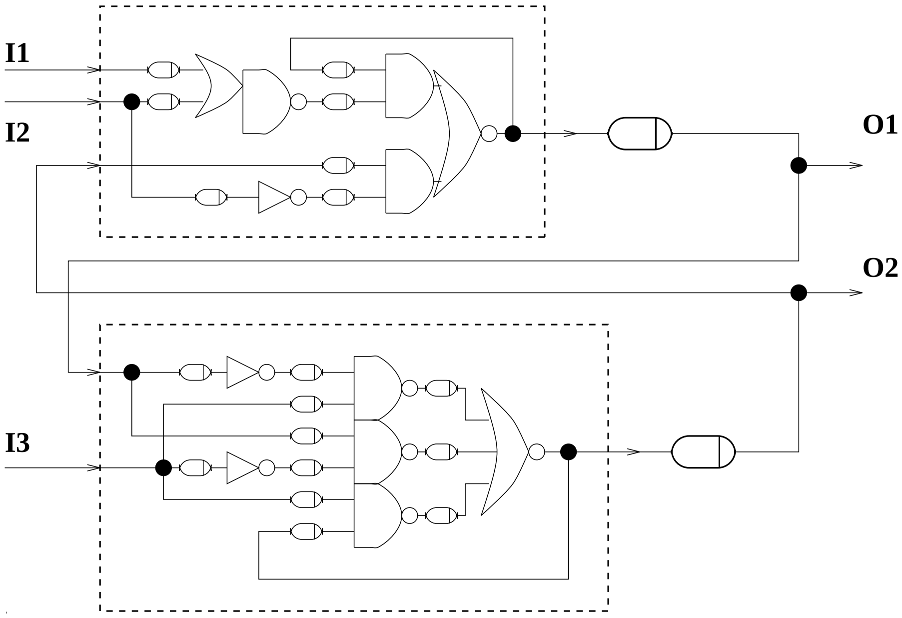
Extensions of monotonous cover theory

- Exact SG-based procedure based on state labeling (similar to state encoding)
- Theory can be extended to handle complex gates:
  - monotonicity conditions must hold for a *set* of cubes
  - each transition in ER must be covered by at least one cube of the gate

Synthesis with bounded wire delays

- Unger (69): summary of classical techniques for asynchronous FSMs
- Lavagno (91, ...): hazard-free synthesis from STGs
- Nowick (91): (92): synthesis of self-clocked asynchronous FSMs
- Yun (92): hazard-free synthesis of asynchronous FSMs

Synthesis of bounded-delay circuits from STG

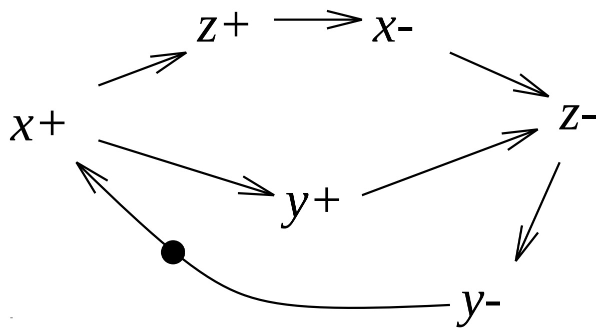


- One combinational logic block for each STG non-input signal

- Decouple hazards from races

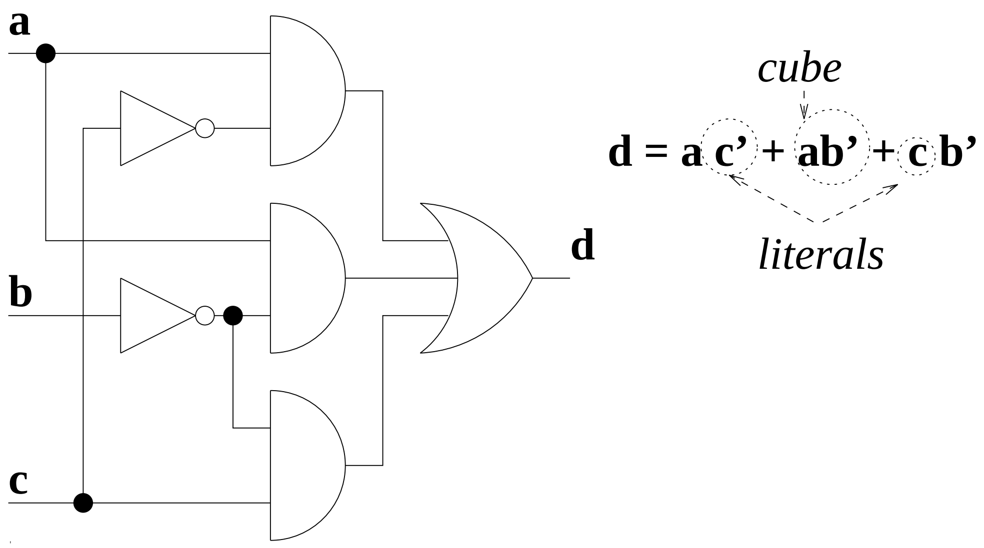
Synthesis algorithm properties

- Two classes of hazards:
  - from concurrent transitions
  - from ordered transitions



- The initial synthesis algorithm guarantees no hazards from *concurrent* transitions

Two-level circuit representation



- Prime cube: no literal can be removed
- Irredundant circuit: no connection can be removed

Initial synthesis algorithm

- build the SG from the STG

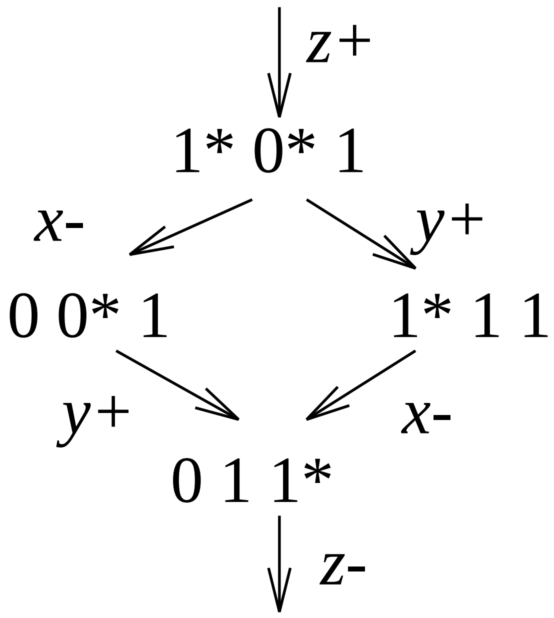
- for each non-input signal o
  - for each state s
    - for each maximal set T of transitions
      concurrently enabled in s

      whose firing does not change the implied value of o
      - generate a function cube

        that does not depend on signals in T

Synthesis algorithm example

- Synthesize signal z
- Current state $x\overline{y}z$
- Enabled transitions: $\{x-, y+\}$
- Enabled transitions that do not cause the implied value to change: $T_1 = \{x-\}$, $T_2 = \{y+\}$
- Added cubes: $\ldots + \overline{y}z + xz \ldots$



Properties of initial synthesis algorithm

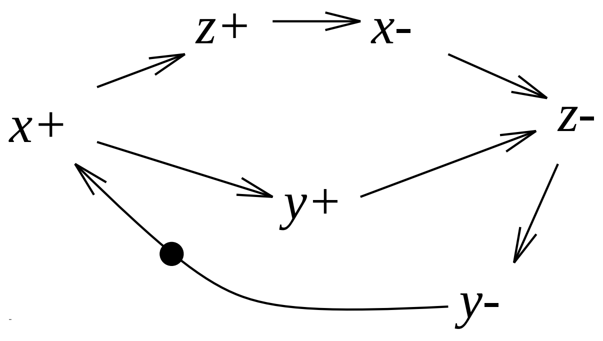
- Guarantees two-level implementation in which *and* gates are stable when concurrent transitions fire

- Circuit is hazard-free with respect to concurrent transitions

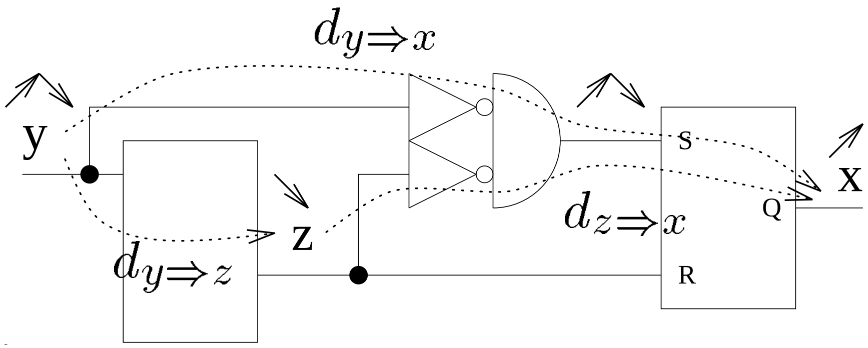
- Optimizing the circuit using only *distributivity, associativity and DeMorgan* preserves this property (Unger '67)

$\Rightarrow$ We can use standard (restricted) logic synthesis and technology mapping !

Remaining hazards



- Ordered transitions: $y+ \rightarrow z-$ (x should not be affected)
- Corresponding circuit:



- Hazard if $d_{y \Rightarrow x} > d_{y \Rightarrow z} + d_{z \Rightarrow x}$ (transition order **reversed** for x)

Hazard detection algorithm

- for each output signal o
  - for each pair (s', s'') of SG states
    between which the implied value of x does not change
    - for each pair of ordered transitions $(y* \rightarrow z*)$
      between s' and s''
      - if $d_{y \Rightarrow x} > d_{y \Rightarrow z} + d_{z \Rightarrow x}$ then report a hazard

Hazard detection example

0 0* 1

$\big\downarrow$ $y+$

0 1 1*

$\big\downarrow$ z-

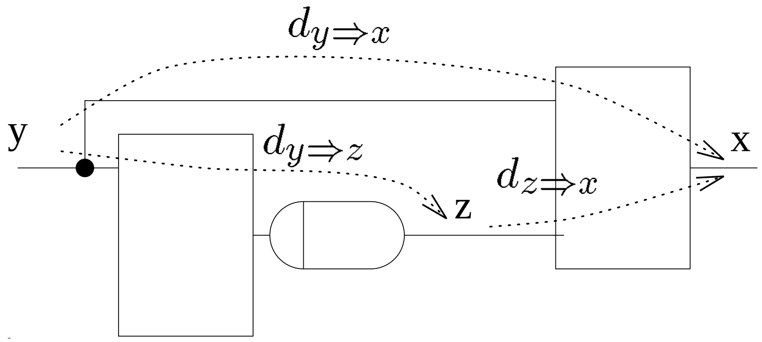
0 1 0

x \ yz	00	01	11	10
0	1	0	0	0
1				

(x must remain at 0)

Hazard elimination idea

- For each case in which a hazard exists if $d_{y \Rightarrow x} > d_{y \Rightarrow z} + d_{z \Rightarrow x}$
  - if possible, balance $d_{y \Rightarrow x}$ and $d_{z \Rightarrow x}$
  - otherwise, increase $d_{y \Rightarrow z}$



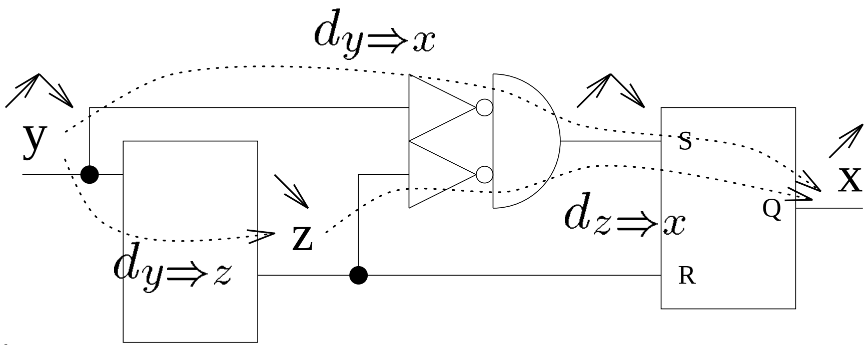
- Eliminating one hazard does not make any other hazard worse:
  - $d_{y \Rightarrow x}$ and $d_{z \Rightarrow x}$ are measured *before* the padded delay
  - $d_{y \Rightarrow z}$ is measured *after* the padded delay

Analogy with synchronous circuits

- In classical synchronous circuit synthesis:
  - slow down *the clock* until *all events* complete propagation
- In bounded delay asynchronous circuit synthesis:
  - slow down *each signal* until *all events that caused its transition* complete propagation
- We can use similar timing analysis techniques for the combinational logic blocks...

Hazard elimination procedure

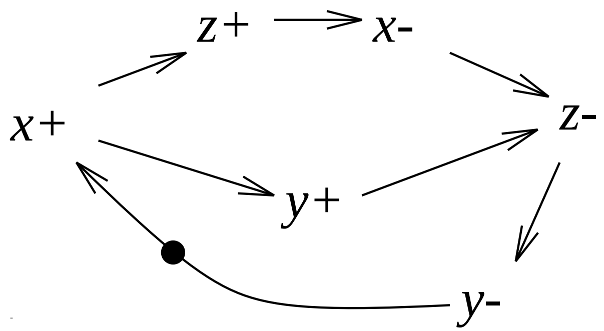
- Each hazard is associated with a triple of delays:



- $d_{y\Rightarrow x}$ and $d_{z\Rightarrow x}$ measured on the circuit (timing analysis)
- $d_{y\Rightarrow z}$ measured on the STG

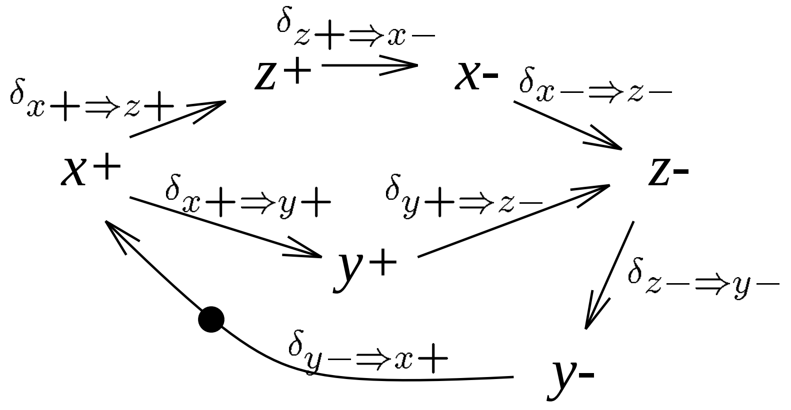
Delay estimation on the STG

- Example: delay between $x+$ and $z-$



- A transition $x*$ fires only when all its predecessors have fired
- Associate a variable D_{x*} with its minimal firing time
- The minimal firing time of $x*$ is
  the max among the minimal firing times of its predecessors
  plus its own delay

Delay estimation example



$$D_{x+} = 0$$

$$D_{z+} = D_{x+} + \delta_{x+\Rightarrow z+} + S_z$$

$$D_{y+} = D_{x+} + \delta_{x+\Rightarrow y+} + S_y$$

$$D_{x-} = D_{z+} + \delta_{z+\Rightarrow x-} + S_x$$

$$D_{z-} = \max \begin{pmatrix} D_{y+} + \delta_{y+\Rightarrow z-} \\ D_{x-} + \delta x- \Rightarrow z- \end{pmatrix} + S_z$$

Hazard elimination conditions

- For each hazard (y^*, z^*, x)

  $D_{y^*} = 0$

  for each transition i^* on an STG path between y^* and z^*

  $$D_{i^*} = MAX_{j^* \text{ precedes } i^*}(D_{j^*} + \delta_{j^* \Rightarrow i^*}) + S_i$$

  $D_{z^*} \geq d_{z \Rightarrow x} - d_{y \Rightarrow x}$

- Find a minimum cost assignment to the S_i variables that satisfies all the inequalities

Summary of logic synthesis

- Different algorithms for different delay models
  - neglecting wire delays, robust with respect to gate delays
  - considering wire delays but assuming bounds on them
- In any case, post-layout timing analysis is required
- Hierarchical decomposition:
  - bounded delays within blocks
  - unbounded wire delays between blocks

  What is the right granularity ?

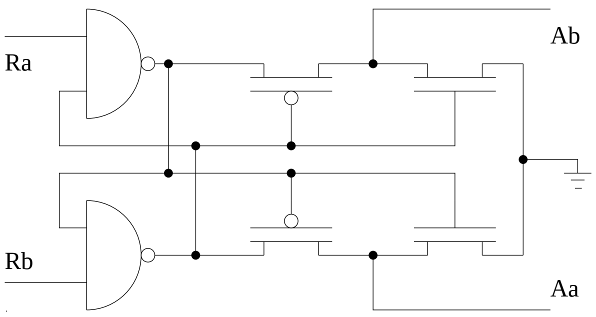
Meta-stability problem

- Feedback (output) transition disabled by an input or an output transition

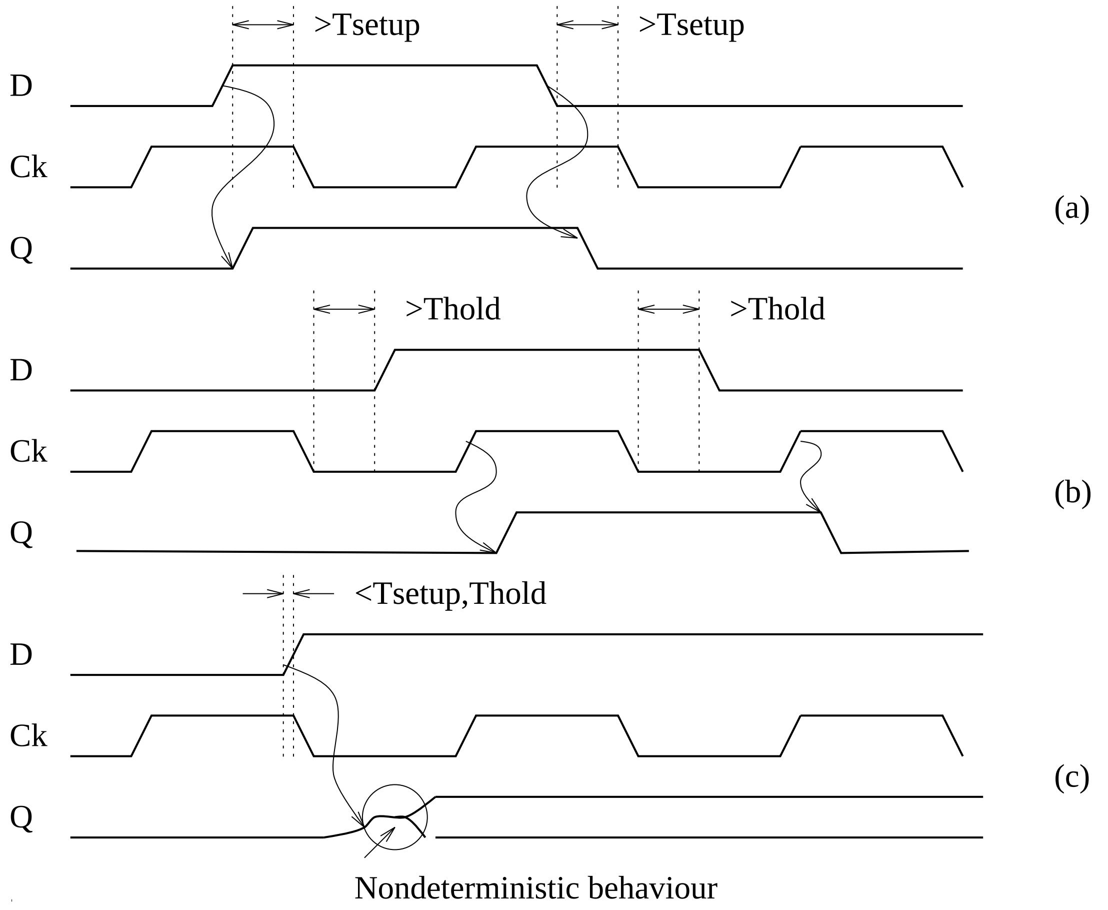
- May last for an unbounded amount of time

- Unavoidable with digital circuits (Molnar '73)

- Can be "hidden" with analog circuitry (Seitz '80)

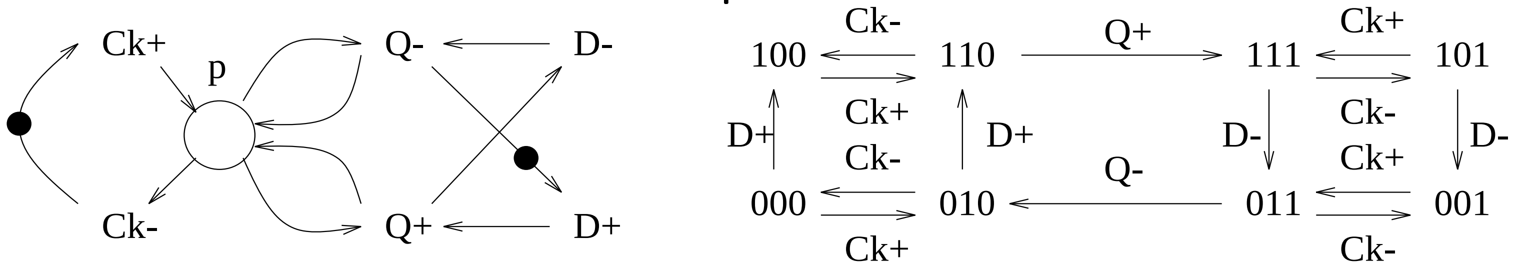


Example: a transparent latch



Meta-stability and STGs

- Manifestation: disabled output transition

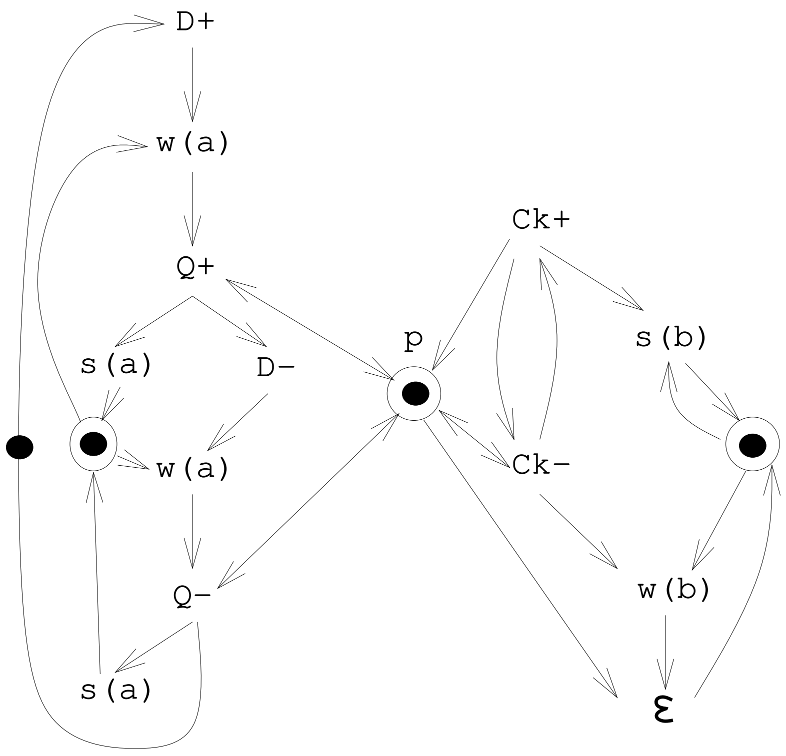


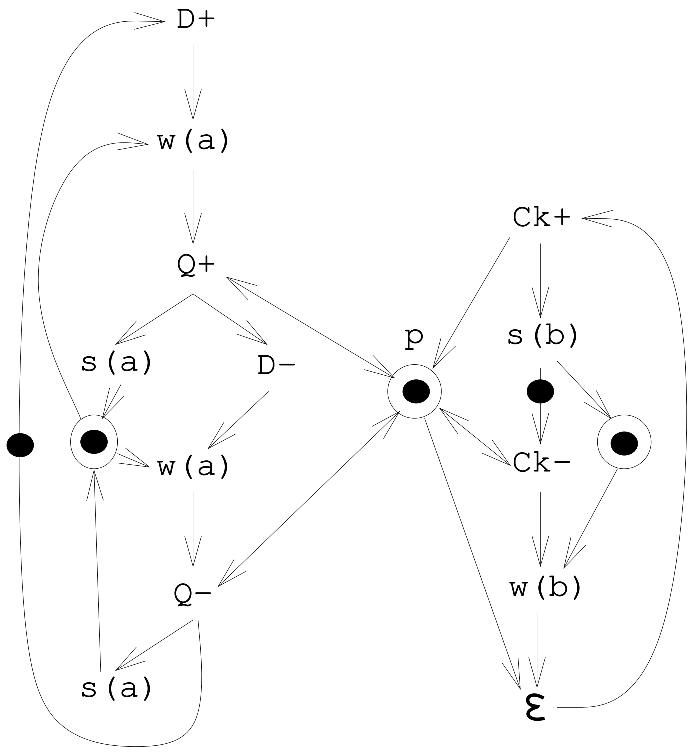
(a) (b)

- Solution: insert `wait` and `signal` primitive to allow
  - either the disabled transition
  - or the disabling transition

  to fire

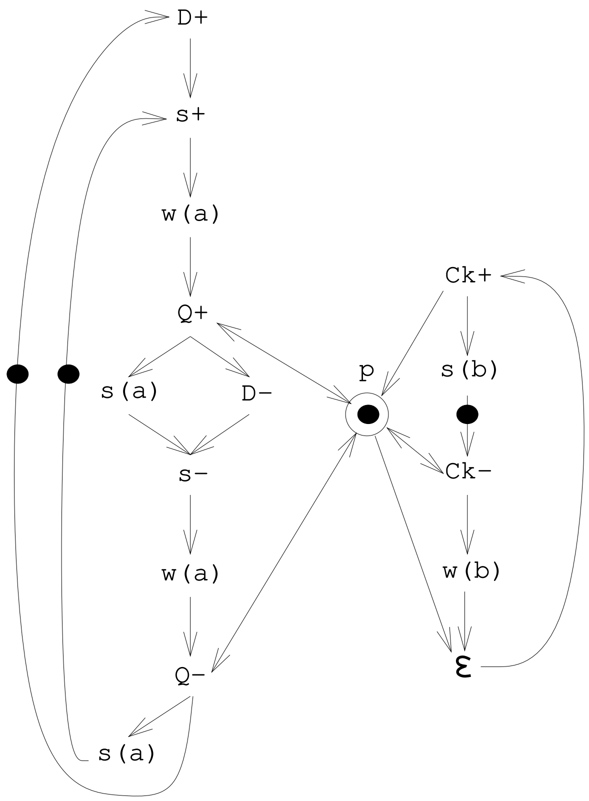
ME insertion in latch STG

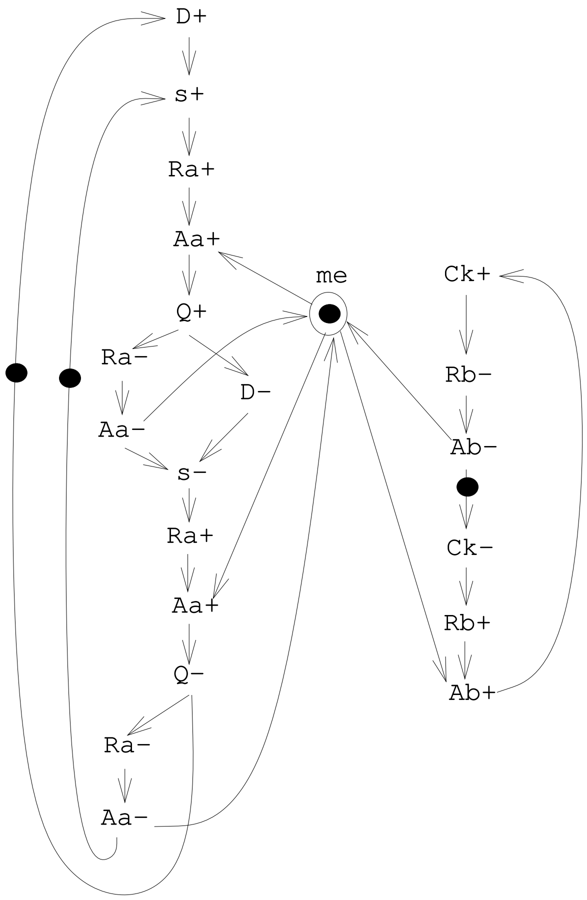




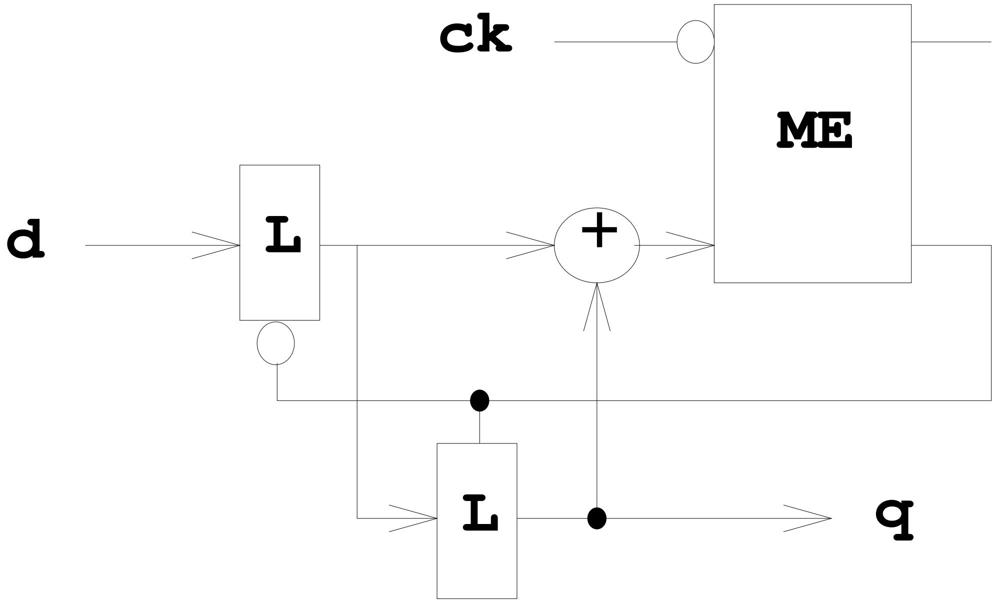
- Unsafeness avoided by causality or *timing* constraints

Request/acknowledge expansion





Final asynchronous latch implementation



- Transparent latch with stoppable clock...

Synthesis approach

- Detect non-persistent output transitions
- Define critical sections and insert ME Req/Ack signals (minimization, to cover only independent conflicts)
- Synthesize the conflict-free STG
- Add MUTEXes to the circuit, to implement Req/Ack pairs

Asynchronous circuit testing

- Traditionally considered *difficult*

- Not much worse than synchronous case if a *systematic design methodology* is followed

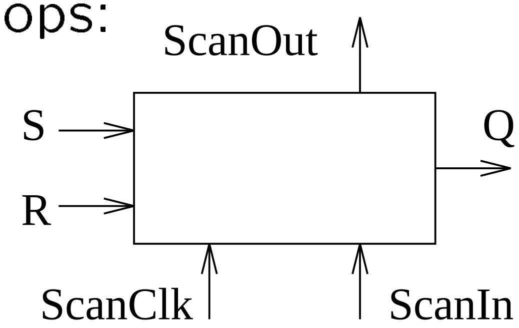
- Delay-fault testing is generally required:
  correctness only under specific delay hypotheses
  (zero or bounded wire delay)

Asynchronous circuit testing

- Speed-independent circuits are self-checking (Armstrong 69, Varshavsky 90, Beerel 91, ...), but only
  - for output stuck-at faults
  - if delay-faults do not invalidate the gate delay model
- Full-scan decomposition-based methodology (Keutzer 91)
- Partial-scan phase splitting-based methodology (Kishinevsky 94)

Hazard-free testing vs. operation

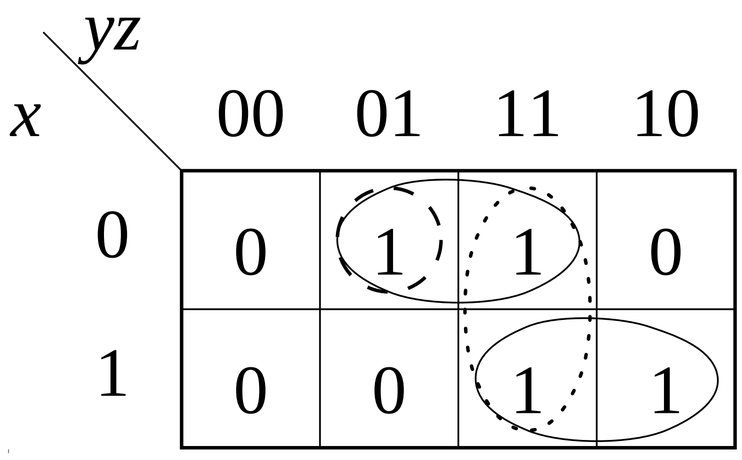
- Hazard-free testing: for each fault:
  - there exists a sequence of vectors that do not cause hazards and test the fault under every timing condition
  - the vectors are produced by scan flip-flops:



- Hazard-free operation: for each vector sequence:
  - that conforms to the circuit specification
  - the sequence cannot cause a hazard under any timing condition

Definitions

- Asynchronous circuits often require redundant
  - gates (bounded wire delay) or
  - gate inputs (unbounded gate delay)
- Redundancy vs. single cube containment:



Testing by phase splitting

- A circuit is *unate* if for all gates g for all inputs i, all paths from i to g have the same number of inversions

- **Unate circuits are easy to test**

- Full stuck-at and robust path-delay fault testability, with only two weak constraints:
  - no single-cube-containment
  - only algebraic operations can be used for optimization from two-level

- Use special phase-splitting flip-flops (independent outputs in test mode)

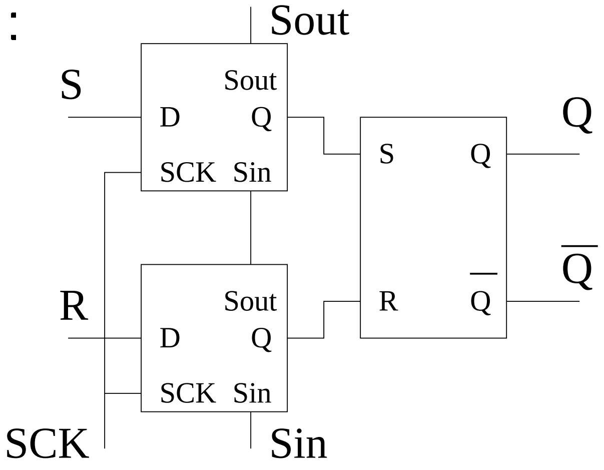
Phase-splitting flip-flops

- Only three combinations of Q and $\overline{Q}$ values are needed

  $\Rightarrow$ no need to break the feedback loop

  (reduces meta-stability probability)

- Example: phase-splitting S/R latch:



Design-for-testability procedure

- Design the circuit using any delay model
- Select paths that must be tested for hazard-free operation
- Full scan: every non-input signal implemented with a scan FF

  $\Rightarrow$ combinational test pattern generation
- Partial scan: every "external" feedback broken by at least one scan FF

  $\Rightarrow$ requires sequential ATPG to initialize non-scan FFs

Part 4: Verification

- What is verification
- Circuit analysis
- Checking specifications
- Comparing specifications and implementations
- Fighting with state explosion
- Timing verification

Verification

- Checking properties of a **specification**

  Example: Given an STG describing a bus protocol, check that

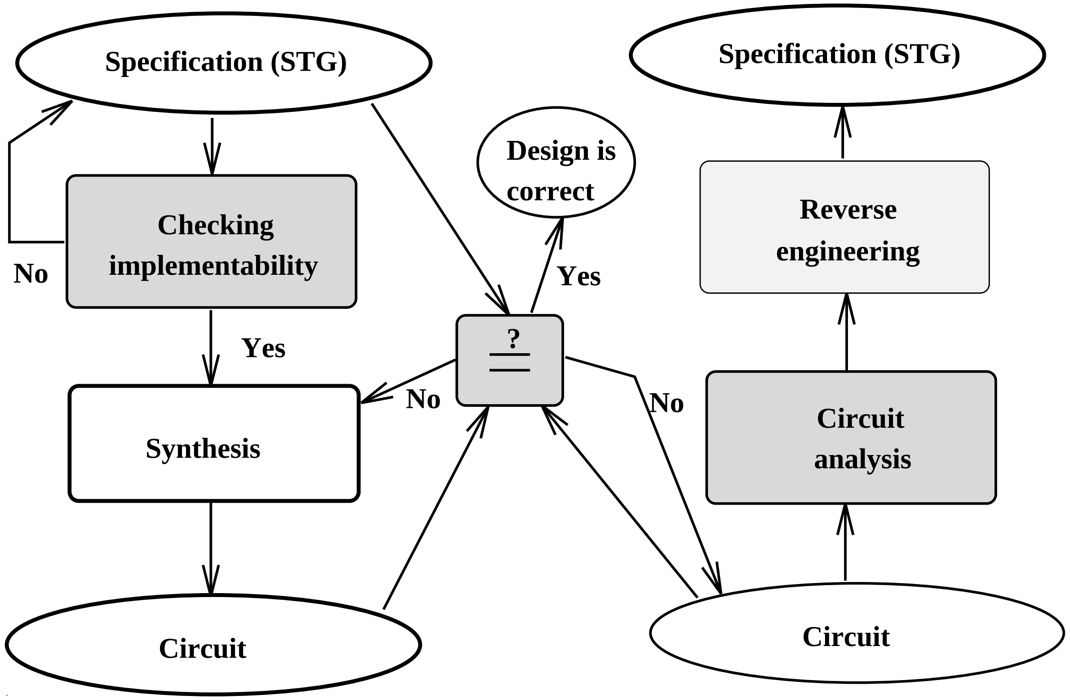
  - it does not deadlock
  - it can be implemented by an asynchronous circuit

- Checking properties of an **implementation**

  Example: Given a circuit check that

  - it is speed-independent
  - it correctly implements a given STG specification

Verification in design flow

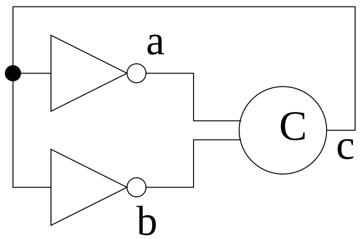


Verification is needed both for top-down and bottom-up design

Fight with state explosion

- Efficient state representation
  - BDD-based techniques (Bryant, Cortadella et al.)
  - State space reduction (Kishinevsky)
- Use of event-based models
  - Unfoldings of Petri Nets and STGs (McMillan, Kondratyev et al., Esparza)
  - Stubborn sets (Valmari)
  - Partial orders (Probst, Godefroid)
  - Reduction of nets (Murata, Esparza)

Speed-independence (D. Muller'56)



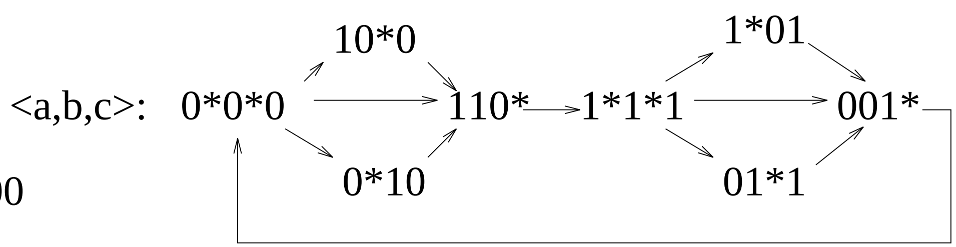
$a = \overline{c}$

$b = \overline{c}$

$c = a\,b + c\,(a+b)$

Initial state: $\langle a,b,c \rangle = 000$

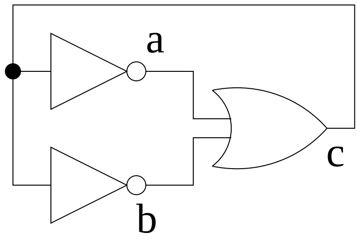
State Graph



There are no disabled transitions $1^* \to 1$ and $0^* \to 0$ in the state graph

Circuit is *speed-independent*

Checking speed-independence

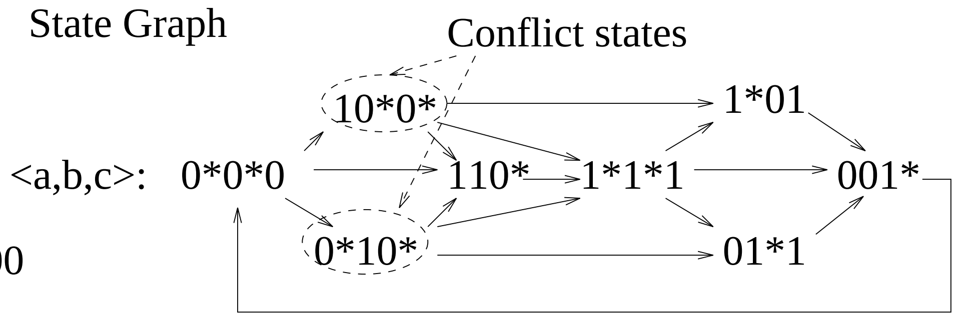


$a = \overline{c}$

$b = \overline{c}$

$c = a + b$

Initial state: <a,b,c> = 000

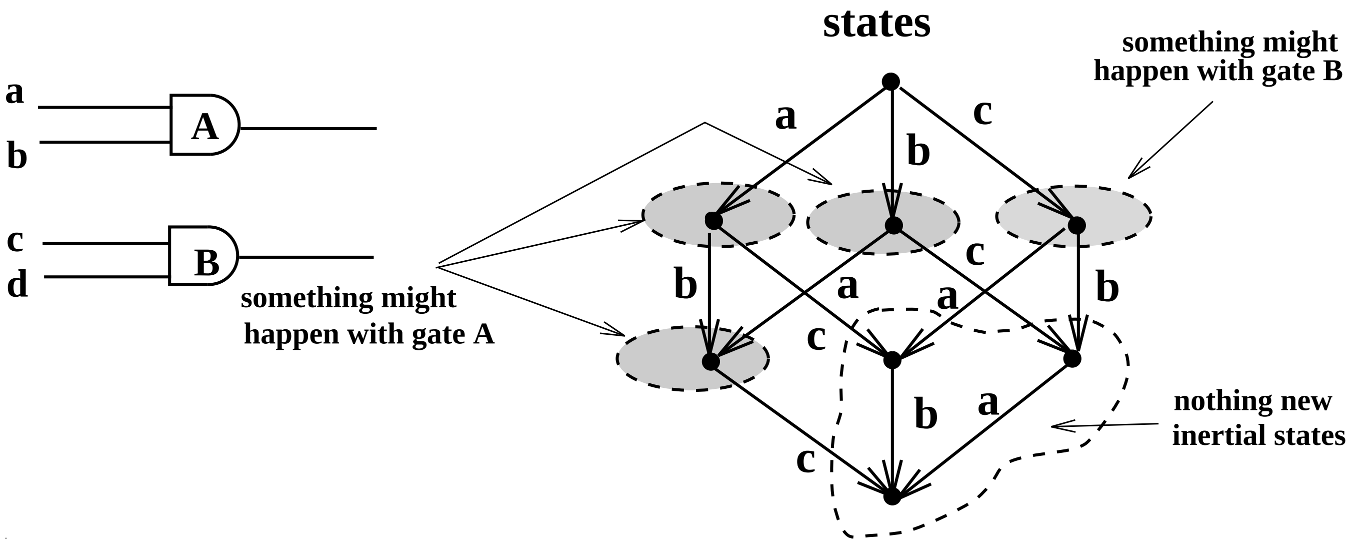


Conflict state: one signal transition is *disabled* by another signal transition, e.g., $1\mathbf{0^*}0^* \rightarrow 1^*\mathbf{0}1$

Circuit is *not* speed-independent (or, more precisely, not semi-modular)

Analysis for speed-independence:
(1) traverse states and
(2) check that no conflict states are reachable

State reduction

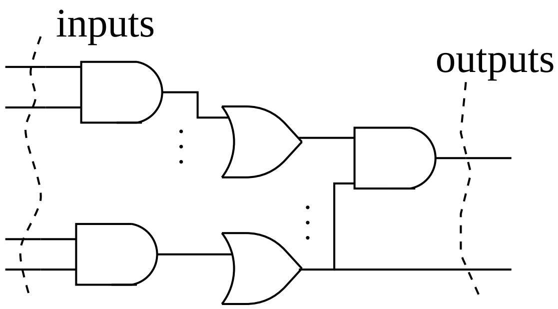


- Use actual gate interconnections
- Only *concurrent* input transitions may cause an output transition
- Micropipeline control queue: run time reduced from 1 hour to 1 sec
  $\Rightarrow$ in highly parallel circuits most of the states are useless for verification
- Efficient for fighting with concurrency. Choice and OR-causality are difficult to handle

What is implementation?

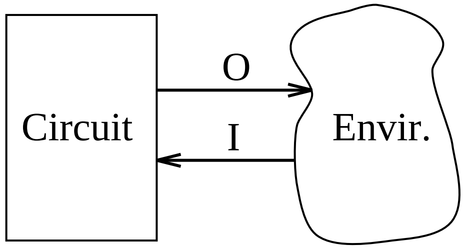
On $I \cup O$ circuit shows

the same language as STG



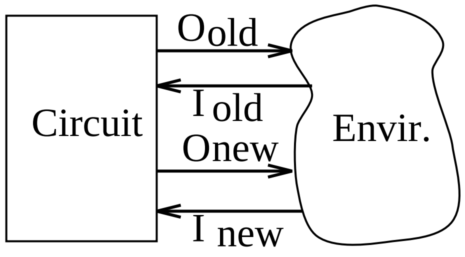
- **I/O-implementability**

  Circuit and STG have exactly the same $I \cup O$ set.
  Interface is preserved



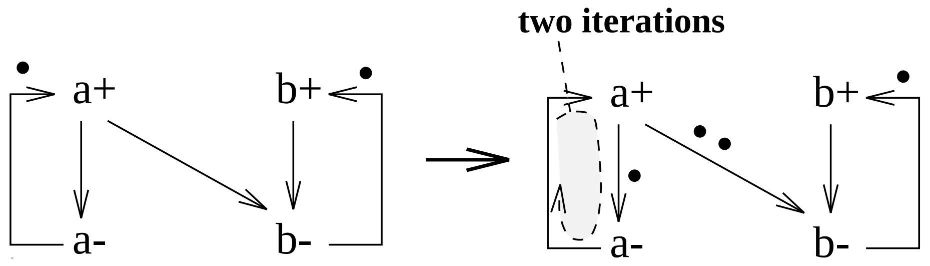
- **Implementability**

  New inputs and outputs can be added to a circuit.
  Interface is not preserved



Circuit synthesis from an STG

- **Boundedness**



- **Consistent state assignment (CSA)**

  (Chu'87)

$$\cdots \rightarrow a+ \rightarrow b+ \rightarrow a+ \rightarrow \cdots$$

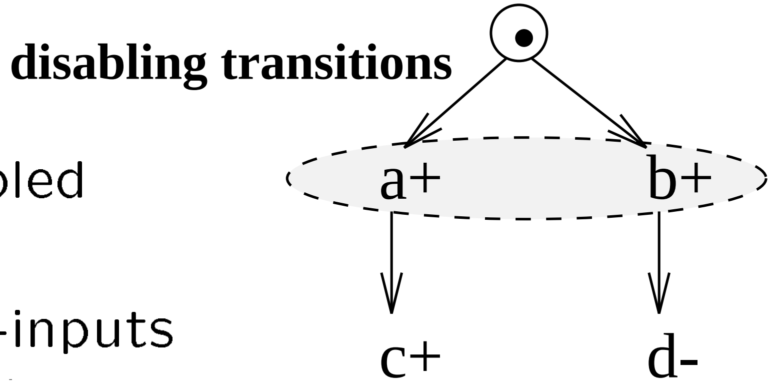
$$ab:\quad 0{*}0 \xrightarrow{a+} 10{*} \xrightarrow{b+} 11 \xrightarrow{a+} ?1$$

- **Complete state coding, CSC**

$$\boxed{\text{Boundedness} + \text{CSA} + \text{CSC} \Longleftrightarrow \text{circuit}}$$

Implementability by speed-independent circuit

Signal Persistency:



1. Non-input signals cannot be disabled

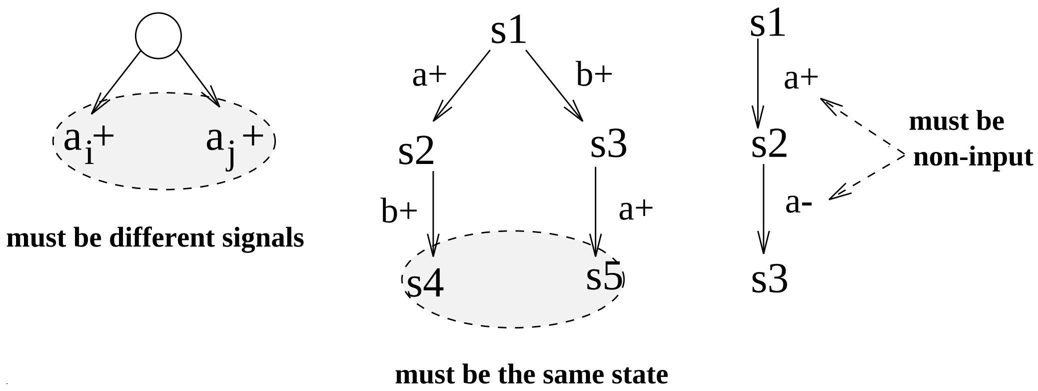
2. Inputs cannot be disabled by non-inputs

Boundedness + CSA + CSC + Persistency $\Longleftrightarrow$ SI-circuit

I/O implementability

- **Irreducible CSC conflicts:**
  cannot be solved without changing interface

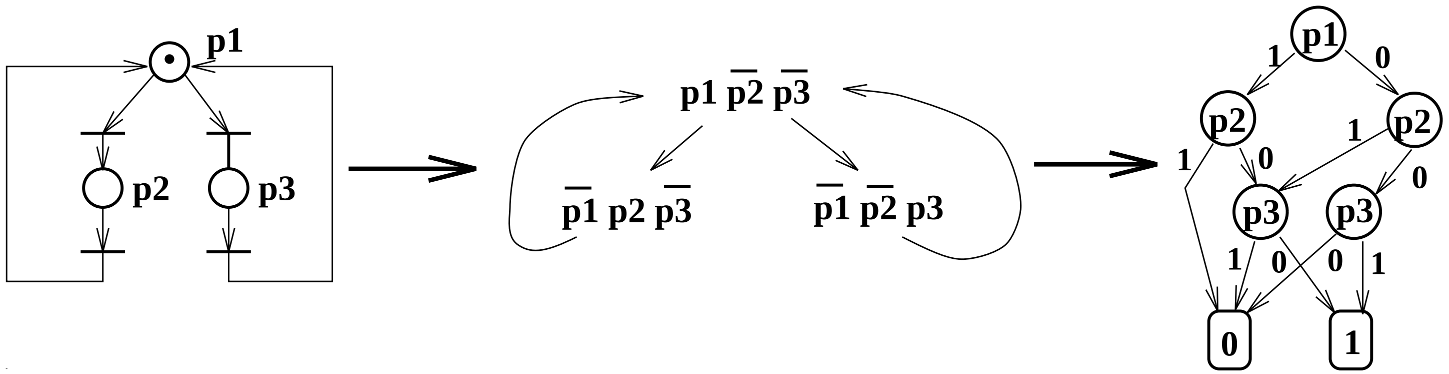
- All CSC-conflicts are reducible if STG is:
  - Deterministic
  - Commutative
  - No complementary input sequences



$$\text{Implementability} + \text{CSC-reducibility} \iff I/O\text{-implementability}$$

BDD representation of reachable states

All reachable markings (**reachability set**):
$p1\overline{p2}\,\overline{p3} + \overline{p1}p2\overline{p3} + \overline{p1}\,\overline{p2}p3$

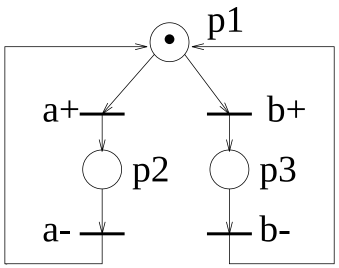


The characteristic function of the reachability set can be represented by a reduced ordered BDD

Symbolic traversal of STG

Each property is represented by a characteristic function

Full state = marking + binary state of signals



$$\longrightarrow$$

$$M_0 = p_1 \overline{p}_2 \overline{p}_3$$

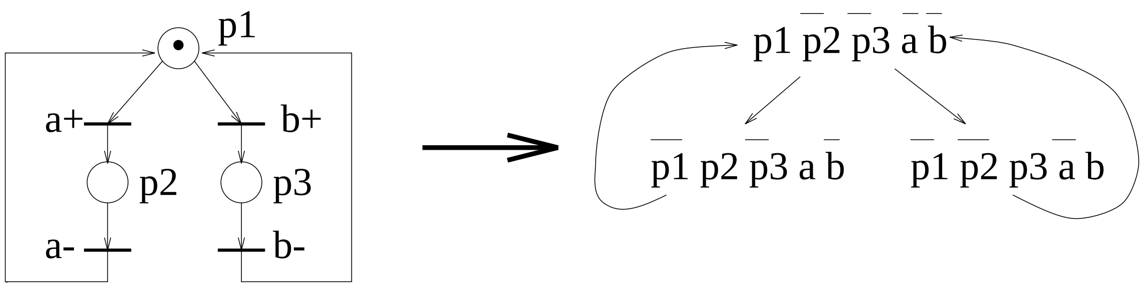
$$S_0 = p_1 \overline{p}_2 \overline{p}_3 \overline{a} \overline{b}$$

$$\mathrm{E}_t = \prod_{p_i \in {}^\bullet t} p_i \quad (t \text{ enabled}) \qquad \mathrm{ASM}_t = \prod_{p_i \in t^\bullet} p_i \quad (\text{succ. marked})$$

$$\mathrm{NPM}_t = \prod_{p_i \in {}^\bullet t} \overline{p_i} \quad (\text{pred. unmarked}) \qquad \mathrm{NSM}_t = \prod_{p_i \in t^\bullet} \overline{p_i} \quad (\text{succ.unmarked})$$

Checking properties via traversal

One-step reachability: $\delta_N(M,t) = (M_{\mathsf{E}_t} * \mathsf{NPM}_t)_{\mathsf{NSM}_t} * \mathsf{ASM}_t$



Consistency: $\mathsf{Inconsistent(a)} = E(a+) * a \; + \; E(a-) * \overline{a}$

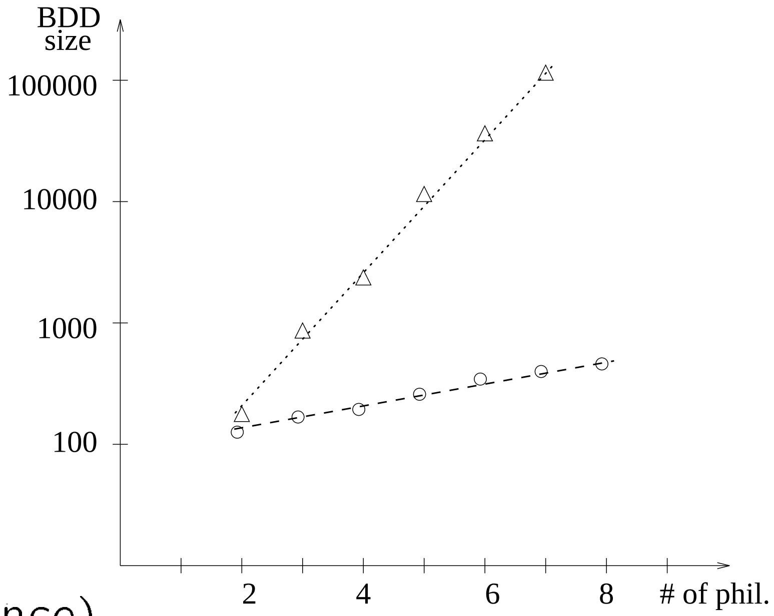
Persistency: $\delta_N(\ \mathsf{E}_{a+}, b+) * \overline{E_{a+}} \neq 0$

TOWARDS EFFICIENT TRAVERSAL

- Variables ordering in BDD
  - Place var. ⇒ by position in simple cycles

(Simple cycle ⇒ One-hot-encoding ⇒ Linear size BDD)

  - Signal variables ⇒ by preceding on places

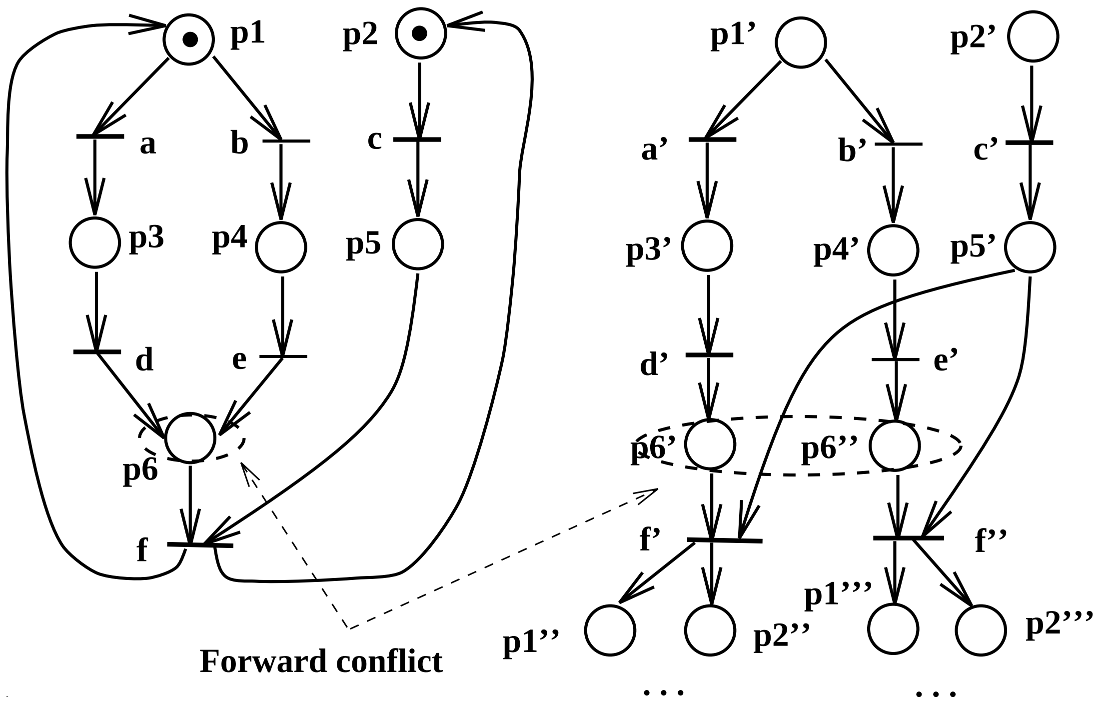


- The traversal order

(Transitions in traversal are chosen by precedence)

Unfolded Petri net

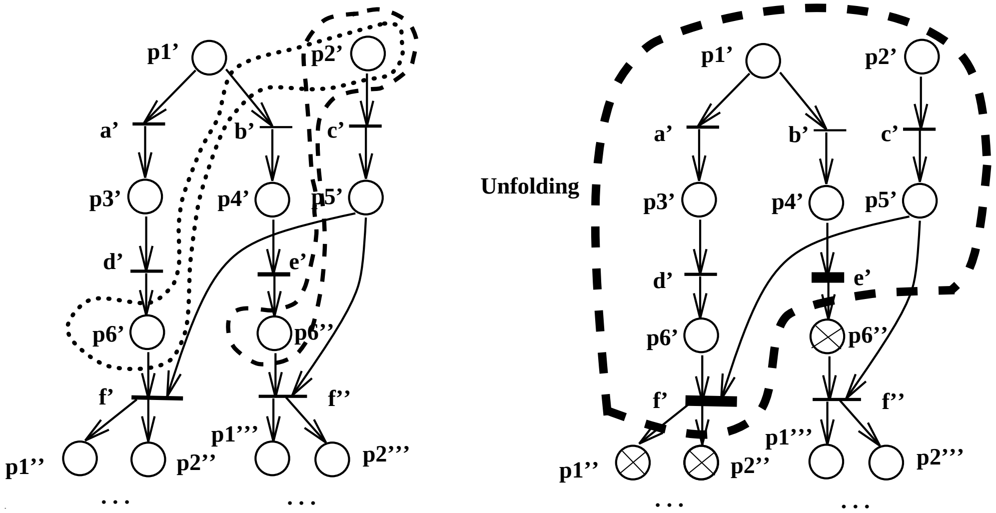
- Occurrence net = an *infinite* unfolded Petri Net
- All forward conflicts are split



Truncating occurrence nets

- **Unfolding** is a finite prefix of the occurrence net
- Unfolding is **complete**: no reachable marking of the initial net is missing
- **Cutoff** is a transition in the occurrence net whose successors can be safely removed. Cutoffs are chosen by observing local configurations.
- **Configuration** of t' = all transitions that must fire before t'
- **Basic marking** of t' = marking after all transitions from the local configuration of t' have fired
- Cutoffs:
  (1) **if** basic-marking(t')= basic-marking(t'') **and**
  size-of-configuration(t') $<$ size-of-configuration(t'') **then** t'' – cutoff
  (2) **if** configurations have the same size **then** check concurrent cutoffs

Unfoldings

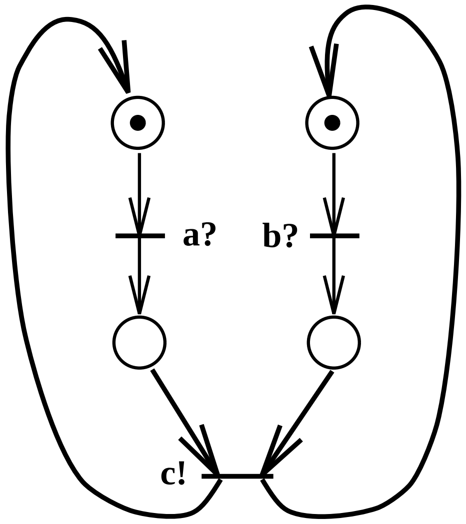


- Basic markings of d' and e' are equal
- Configuration of $f' = \{a', d', c'\}$

- **Ordering relations** are easy to derive:
  a' and c' are concurrent, a' precedes f', a' and e' are in conflict
- All reachable markings are present in unfolding $\Longleftrightarrow$ all properties of nets can be studied by unfoldings
- In many cases it is more efficient than BDD traversal since the ordering relations can be used
  (depends on the property and the net)

Verification in trace theory

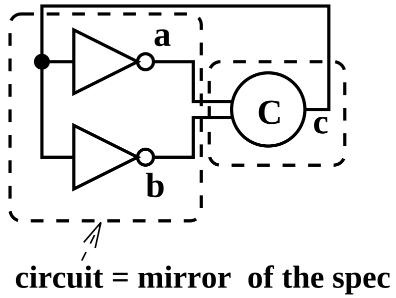
Dill, Rem, Udding, van de Snepscheut, Ebergen



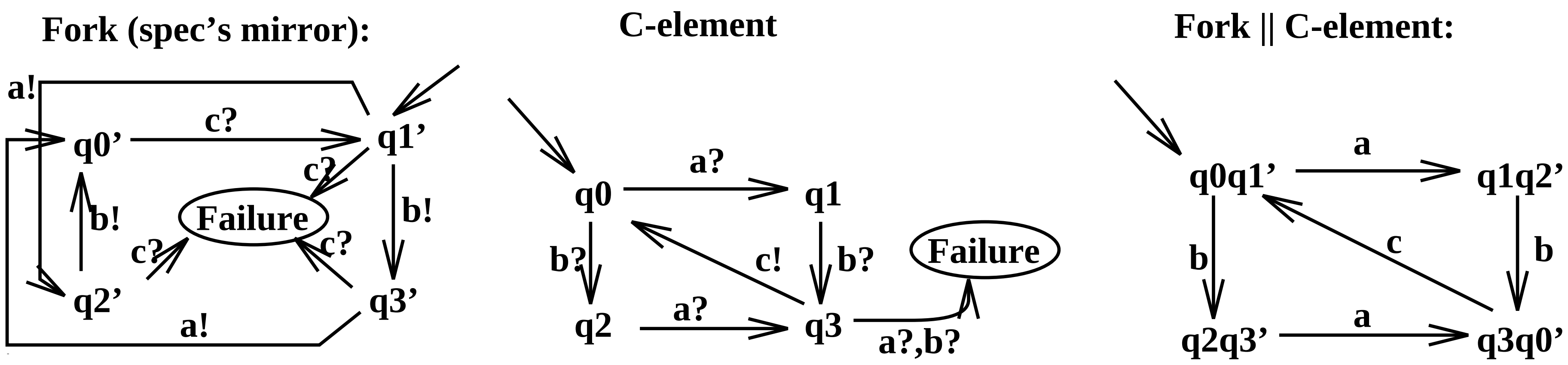
- Specification of Join:
  $pref * [a?; c!] || pref * [b?; c!]$
  $a?, b?$ are input; $c!$ is output

- Mirror of the spec = environment:
  inputs $\leftrightarrow$ outputs:
  $pref * [a!; c?] || pref * [b!; c?]$

- Does a C-element implement the Join?



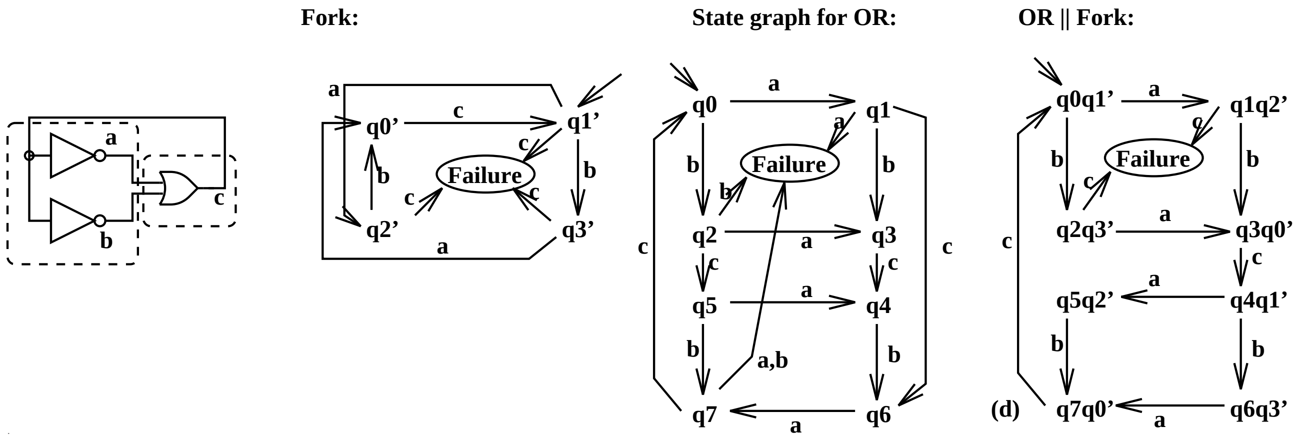
Failure states and hazardous behavior



Hazardous behavior (non speed-independence) =
failure states are reachable in the composite state graph

It also shows that the circuit does not implement the spec

Incorrect implementation



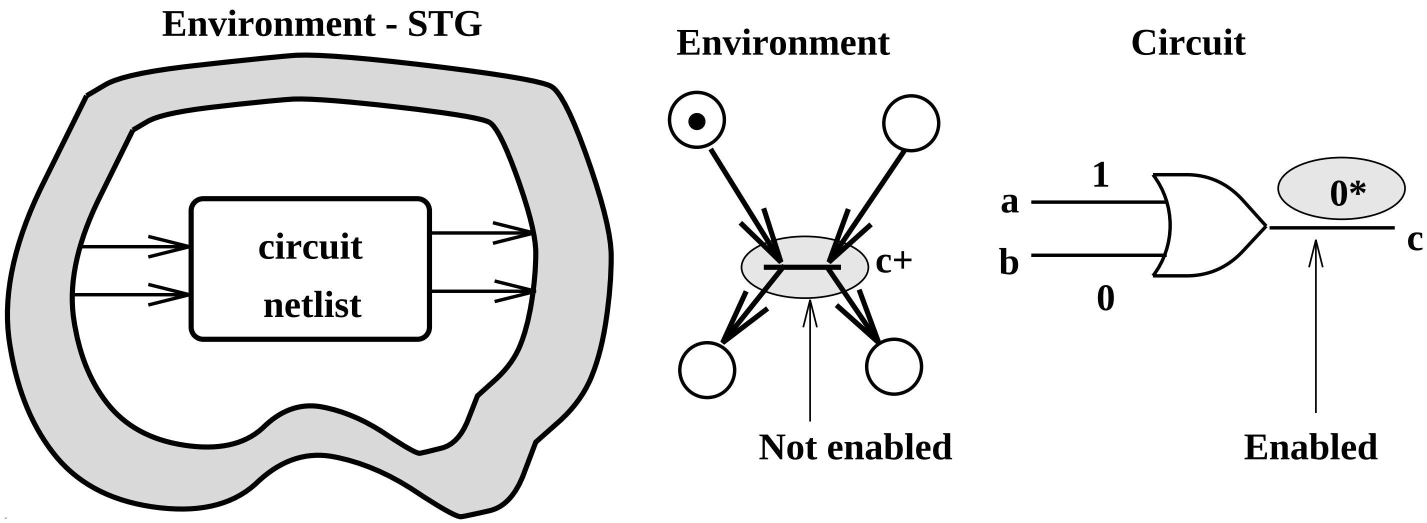
OR-gate is not an implementation for the Join

Failure state is reachable ⇒ an oscillator OR-gate + two inverters is not speed-independent

Compare with speed-independence analysis

Parallel composition of state graphs is inefficient

Parallel traversal on BDD

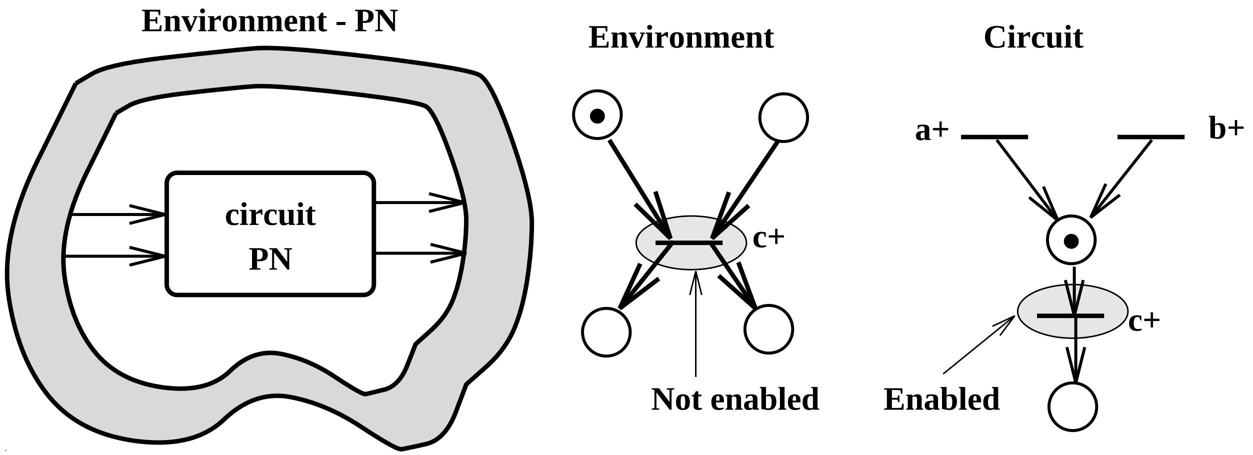


Signal c is enabled in the circuit but not in the environment

Failure detection:
$Reachable * Enabled(circuit) * not(Enabled(STG)) \neq 0$

To make it efficient do it on BDDs (Roig'94)

Parallel traversal on unfoldings

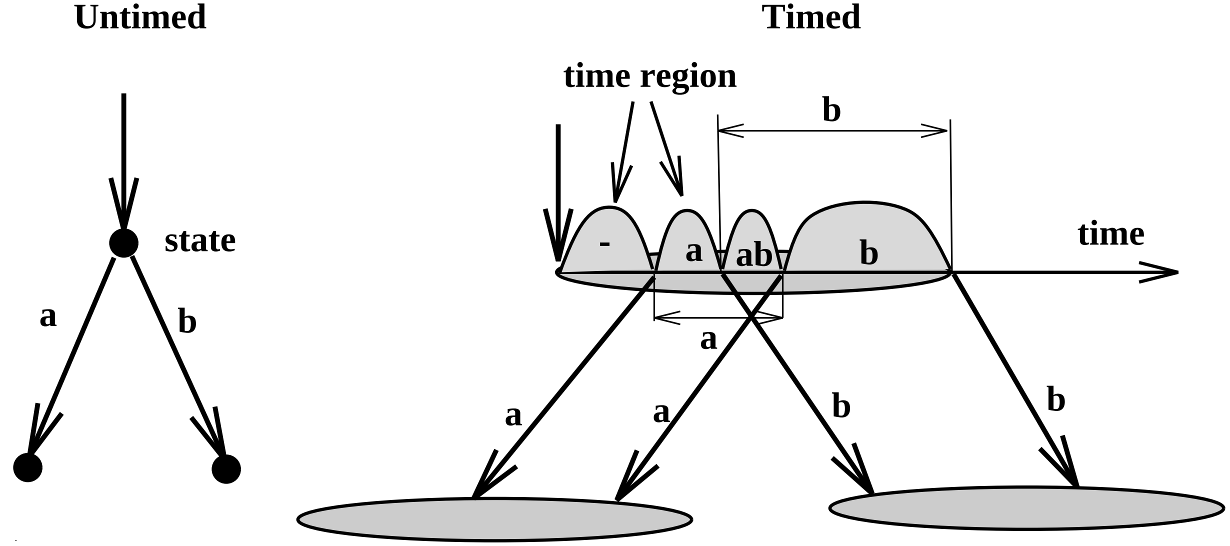


Unfold (1) PN-environment and (2) a PN-description of the circuit

Failure: output transition is enabled in the circuit **and** the corresponding input transition is not enabled in the environment (McMillan'95)

Check failure conditions by traversing the unfolding of the composition on-the-fly

Timing verification.1



- All states inside one time-region are equivalent
- How to partition timed state space into time regions?

Timing verification.2

Time is a real number $\Rightarrow$ the state space is infinite.
For finite-state verification:

- **Discrete time:** correct for important subclasses (orbital nets) (Rokicki, Henzinger)
- **Unit cube time:** partition time-states into equivalent classes (Alur)
- **Geometric time regions:** convex regions of timed states constrained by inequalities $clock_i - clock_j \leq constant$ (Dill, Rokicki, Lewis, Halbwachs)

Discrete time and unit cube time have better worst case state space size.
In practice, geometrical time works better

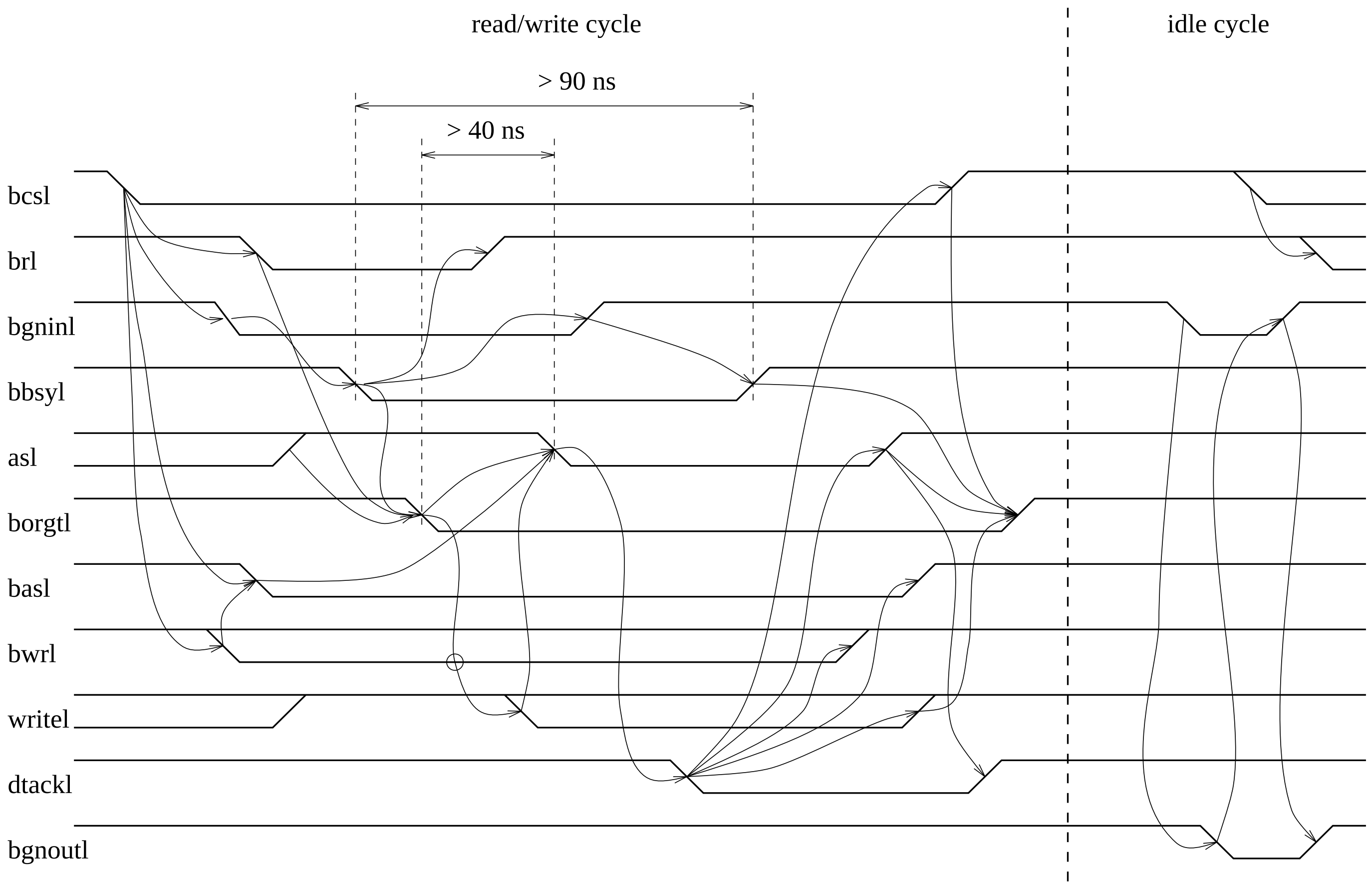
Geometric time regions + partial orders are the most efficient
(Orbital nets, Rokicki and Unique-choice nets, Hulgaard, Burns)

Use verification plus simulation (VHDL,Verilog)

Part5: Practical applications

- Interface design: VME-bus controller
- Processors:
  (1) AMULET1 and AMULET2
  (ARM group, Manchester Univ., Furber)
  (2) Counterflow Pipeline RISC
  (Sun Microsystem Labs, Sproull,Sutherland,Molnar)
- DSP-applications: DCC error corrector
  (Philips Research Labs, Kees van Berkel)

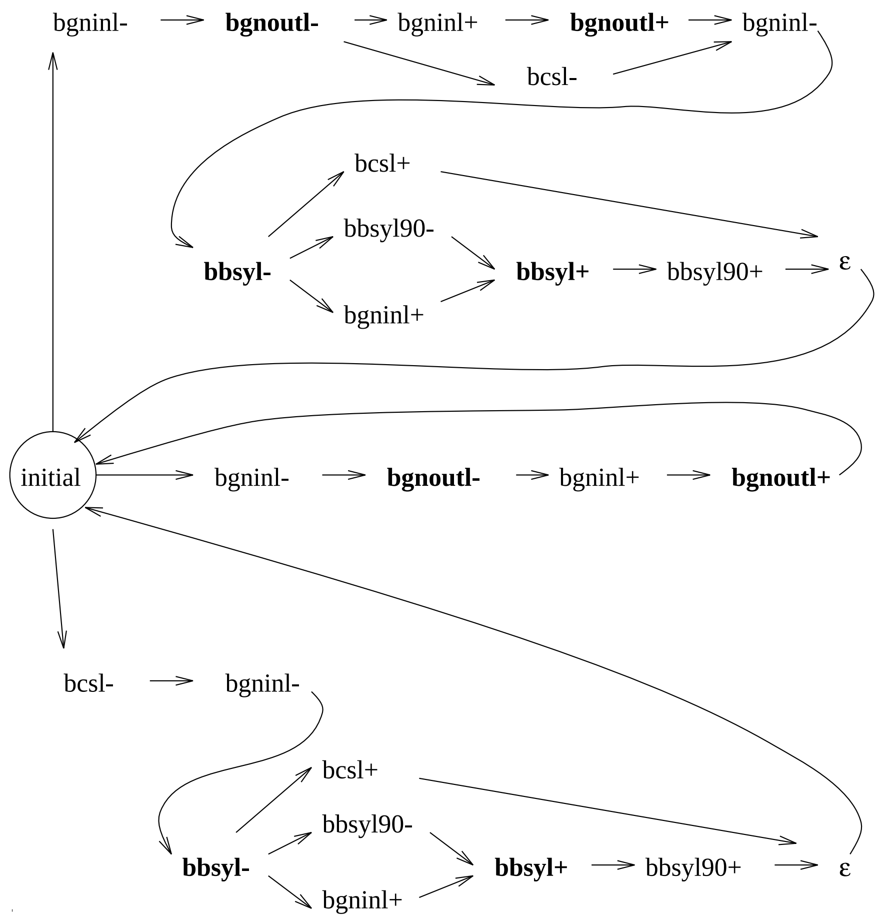
The *VMEbus* master interface protocol



The *VMEbus* master interface protocol

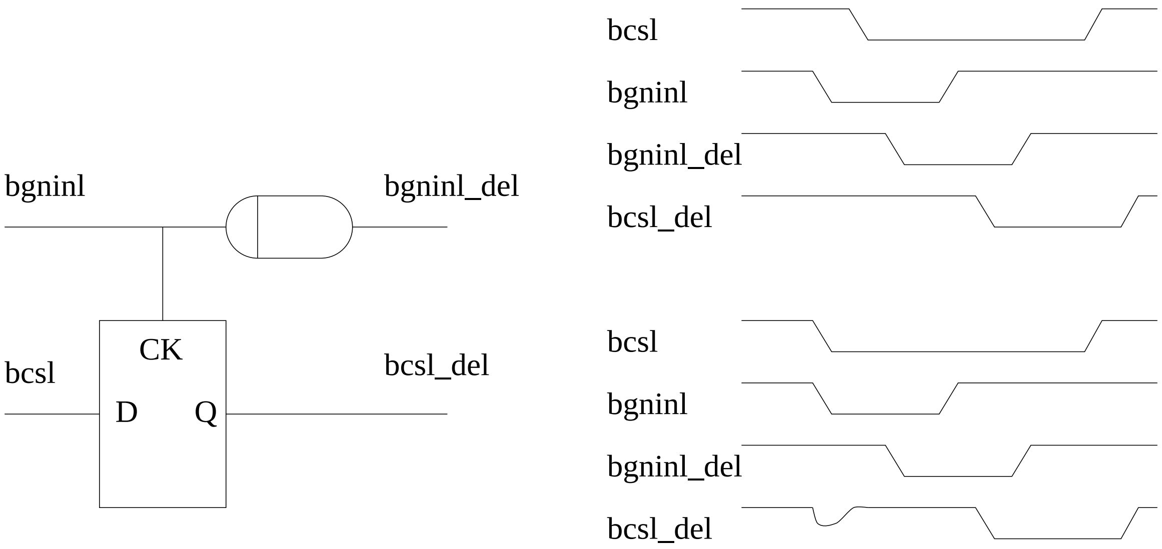
- Decompose into arbitration and read/write protocol:
  - simplify specification and synthesis tasks
  - separate the arbitration component (meta-stability)
- Both decomposed STGs must agree on the partial order of common transitions ("net contraction", Chu '87, or "hiding', Dill '88)

The arbitration STG

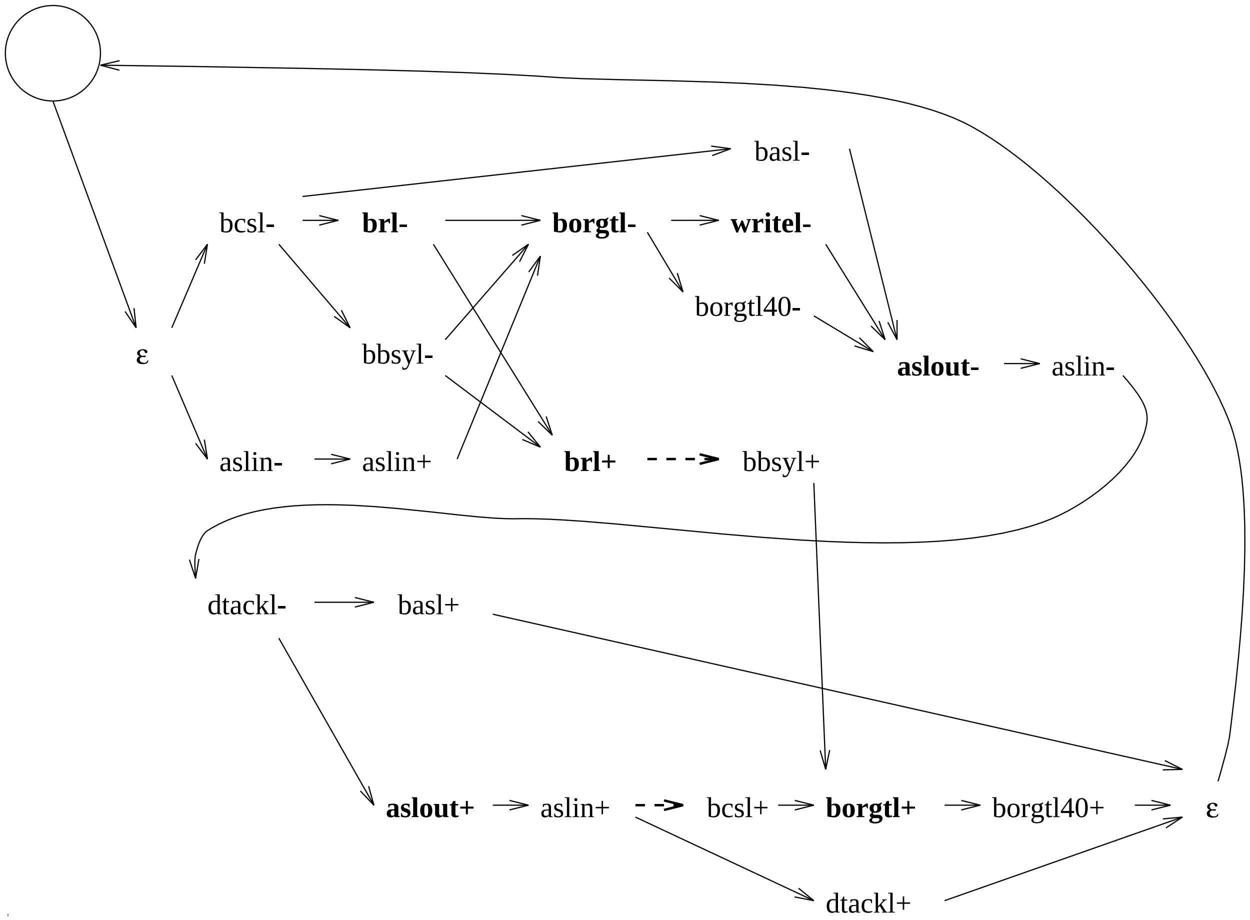


A meta-stability problem

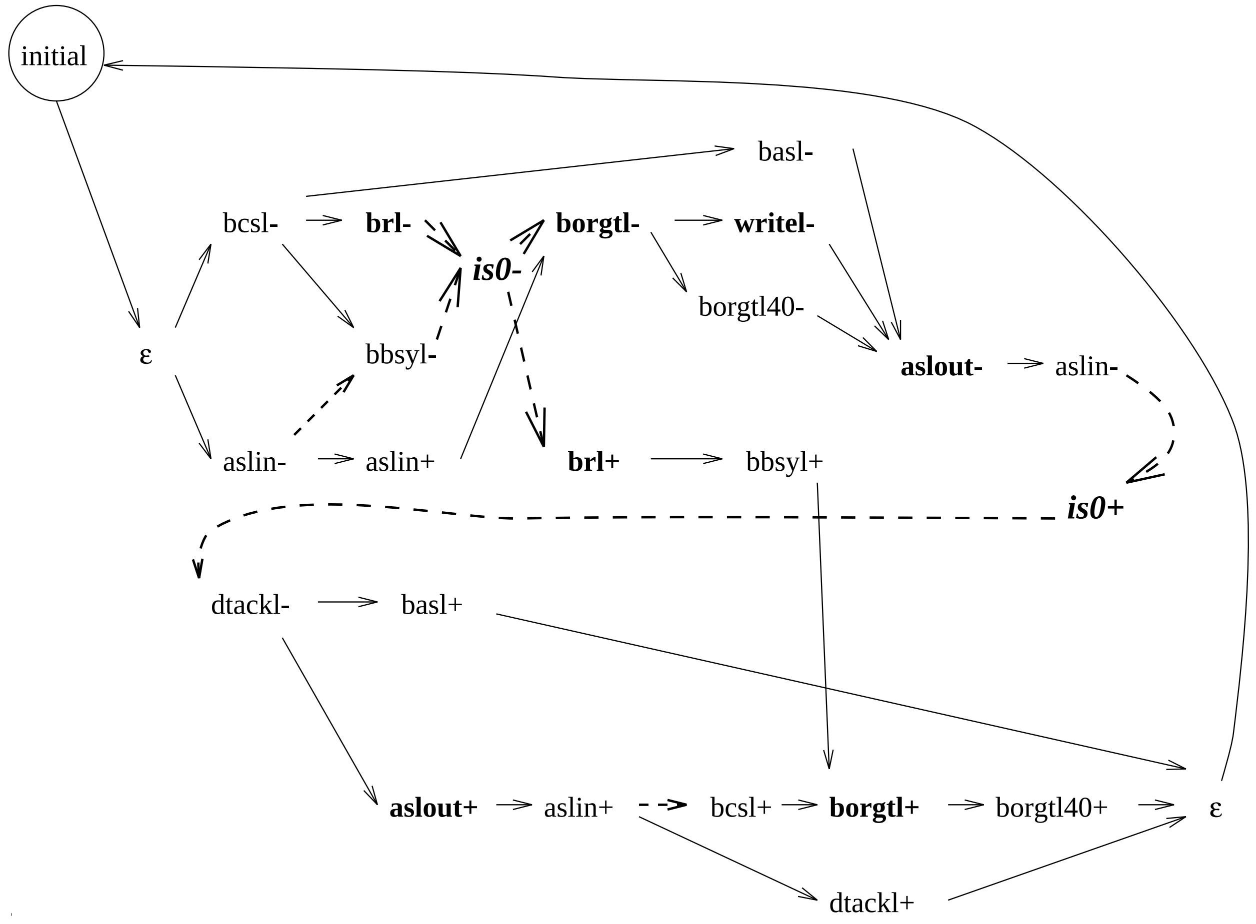
- We must latch and wait...
  (or use a *mutual exclusion element*)



(Half of) the read/write STG



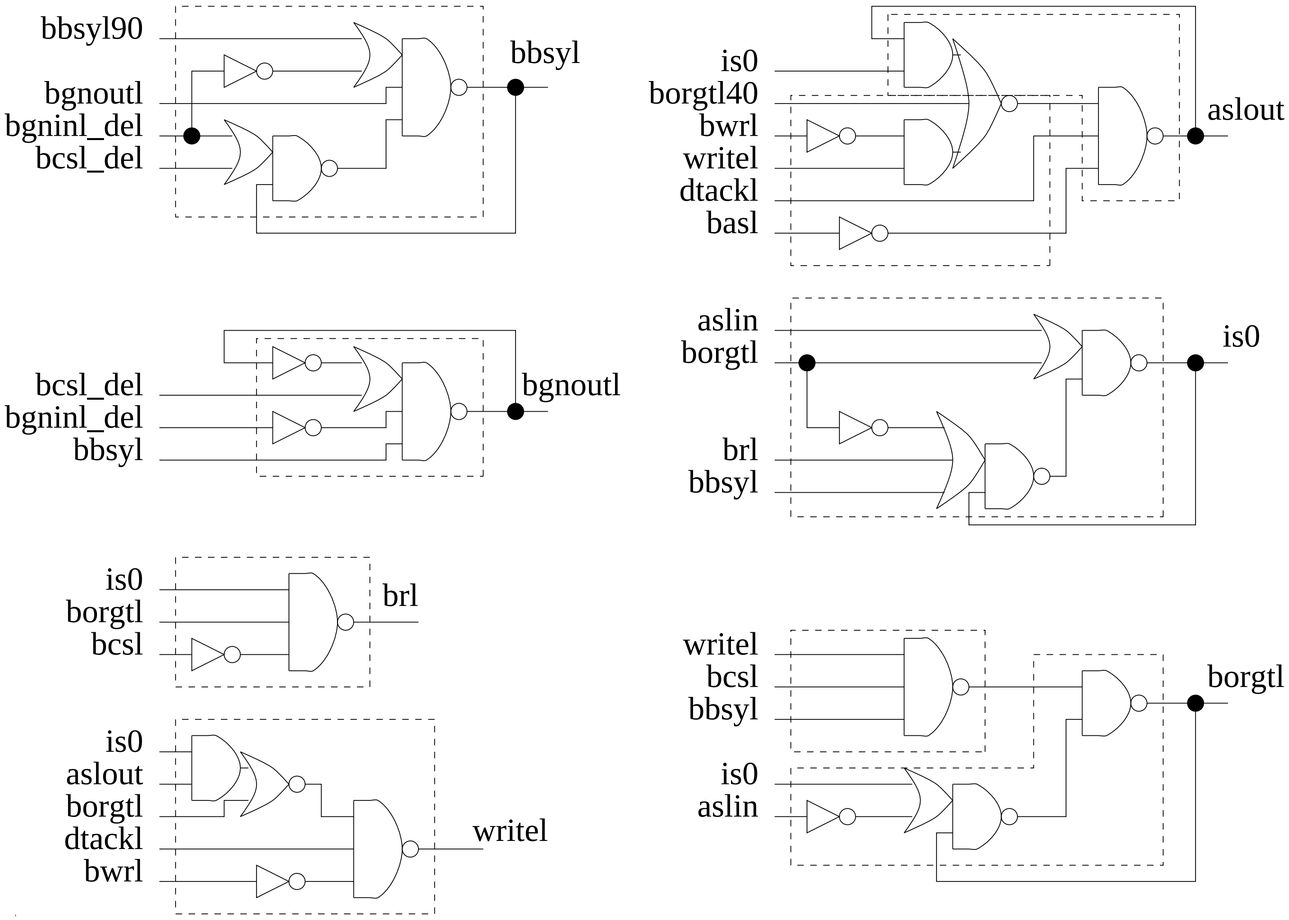
State encoding



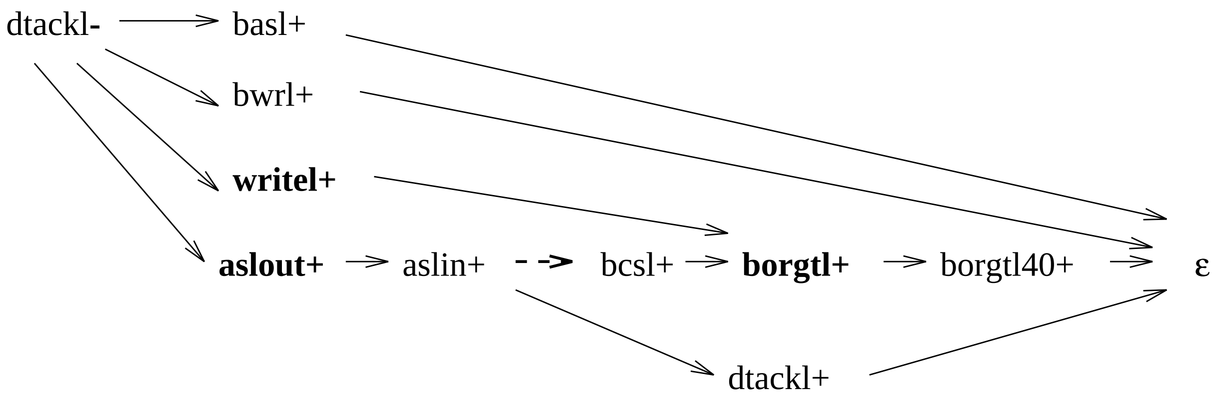
Initial implementation

$$
\begin{aligned}
bgnoutl &= \overline{bbsyl} + \overline{bcsl} \cdot bgnoutl + bgninl \\
bbsyl &= \overline{bbsyl90} \cdot bgninl + bbsyl \cdot bcsl + \\
&\quad bbsyl \cdot bgninl + \overline{bgnoutl} \\
brl &= bcsl + \overline{borgtl} + \overline{is0} \\
borgtl &= \overline{aslin} \cdot borgtl + bbsyl \cdot bcsl \cdot writel + \\
&\quad borgtl \cdot is0 \\
aslout &= aslout \cdot is0 + basl + borgtl40 + \\
&\quad \overline{bwrl} \cdot writel + \overline{dtackl} \\
writel &= aslout \cdot is0 + borgtl + bwrl + \overline{dtackl} \\
is0 &= \overline{aslin} \cdot \overline{borgtl} + bbsyl \cdot is0 + \\
&\quad \overline{borgtl} \cdot is0 + brl \cdot is0
\end{aligned}
$$

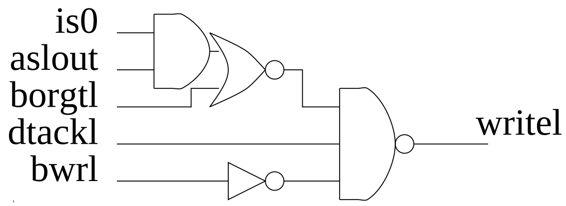
Final implementation (XILINX 3000 or SC)



Hazard example



- if $\delta_{aslout+, dtackl+} + \delta_{dtackl,\ writel} \leq \delta_{aslout, writel}$ then *writel* may have a $0 \to 1 \to 0 \to 1$ hazard



is0 $= 1$, *aslout* $= 0 \to 1$, *borgtl* $= 0\ (\to 1)$,
dtackl $= 0 \to 1$, *bwrl* $= 0\ (\to 1)$

AMULET processor

- AMULET is an asynchronous implementation of the ARM microprocessor

- Goal: microprocessor design with low power consumption (DEC Alpha - 30 watts)

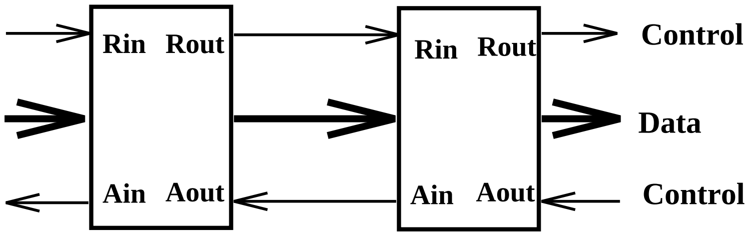
- AMULET1 – two-phase bundled data design $\approx$ two-phase speed-independent control + synchronous data path

- AMULET2 – four-phase bundled data design $\approx$ four-phase speed-independent control + synchronous data path

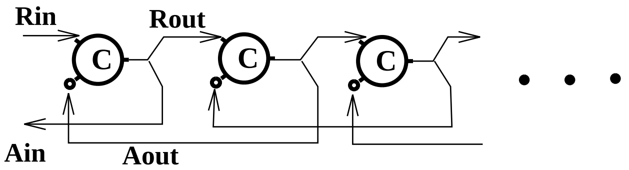
S. Furber 1995

Micropipelines

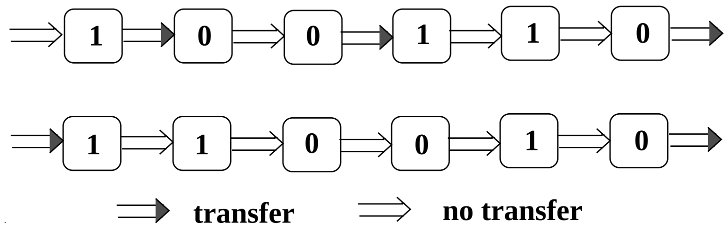
Sutherland's micropipeline:
Control path = Muller pipeline
Data path = Event controlled latches (Capture and Pass)



Micropipeline structure

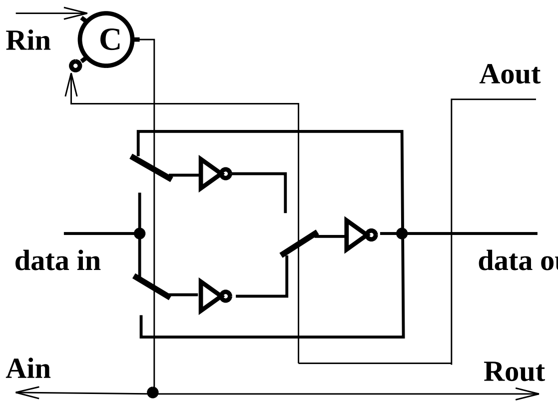


Muller control pipeline

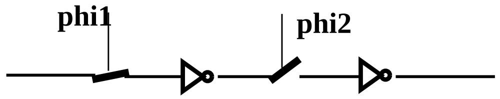


Control operation

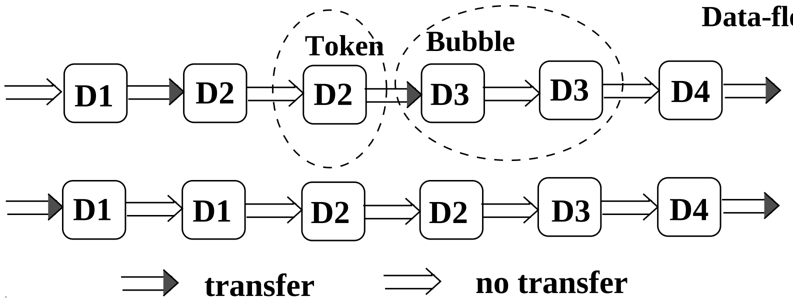
Two-phase control, two-phase data path



Sutherland's micropipeline with two-phase 'capture-pass' latch



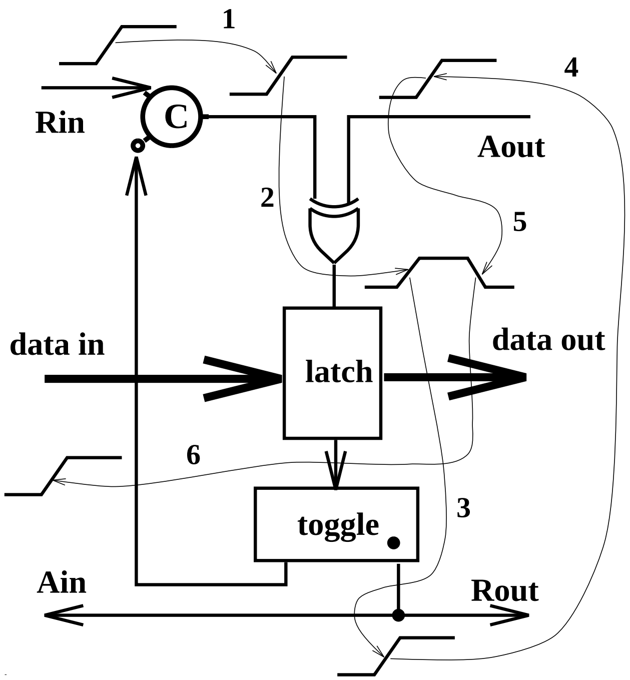
Two-phase clocking



Data-flow operation

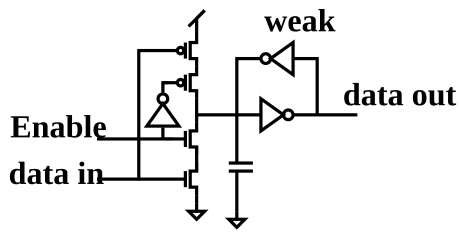
Two-phase register cell is too large and too slow
$\Rightarrow$ level sensitive (four-phase) transparent latches + two-phase to four-phase converters for controlling a register (Amulet1)

Two-phase control, four-phase data path



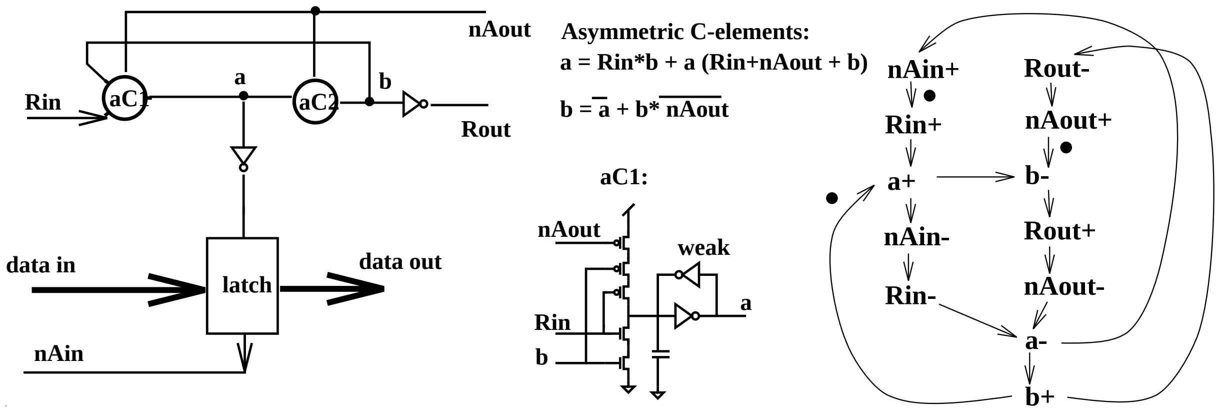
One transition at Rin
forces two transitions
at the latch control

Latch



Two-phase to four-phase control conversion is too expensive
$\Rightarrow$ four-phase control +
level sensitive (four-phase) transparent latches (Amulet2)

Four-phase control, four-phase data path



Only half of the cells of a Muller pipeline can be filled with data

With four-phase control a pipeline can be made dense
Extra memory is needed in the control to distinguish between the old and new value

Day pipeline (with hazards) ⇒ speed-independent pipeline was obtained by synthesis (Forcage)

Four-phase control: 30% faster and consumes 30% less power than two-phase

Amulet results

	Amulet1/ES2	ARM6
Process	1 μ m	1 μ m
Area(mm)	5.5$\times$ 4.1	4.1 $\times$ 2.7
Transistors	58,374	33,494
Performance	20.5 kDhry	33.5 kDhry
Multiplier	5.3 ns/bit	25ns/bit
MIPS/W	77	120

Amulet2 expectations: 3 times better than Amulet1 on the same fabrication process

Counterflow Pipeline Processor Architecture

- Conventional RISC pipeline:
  instructions and data flow in one direction

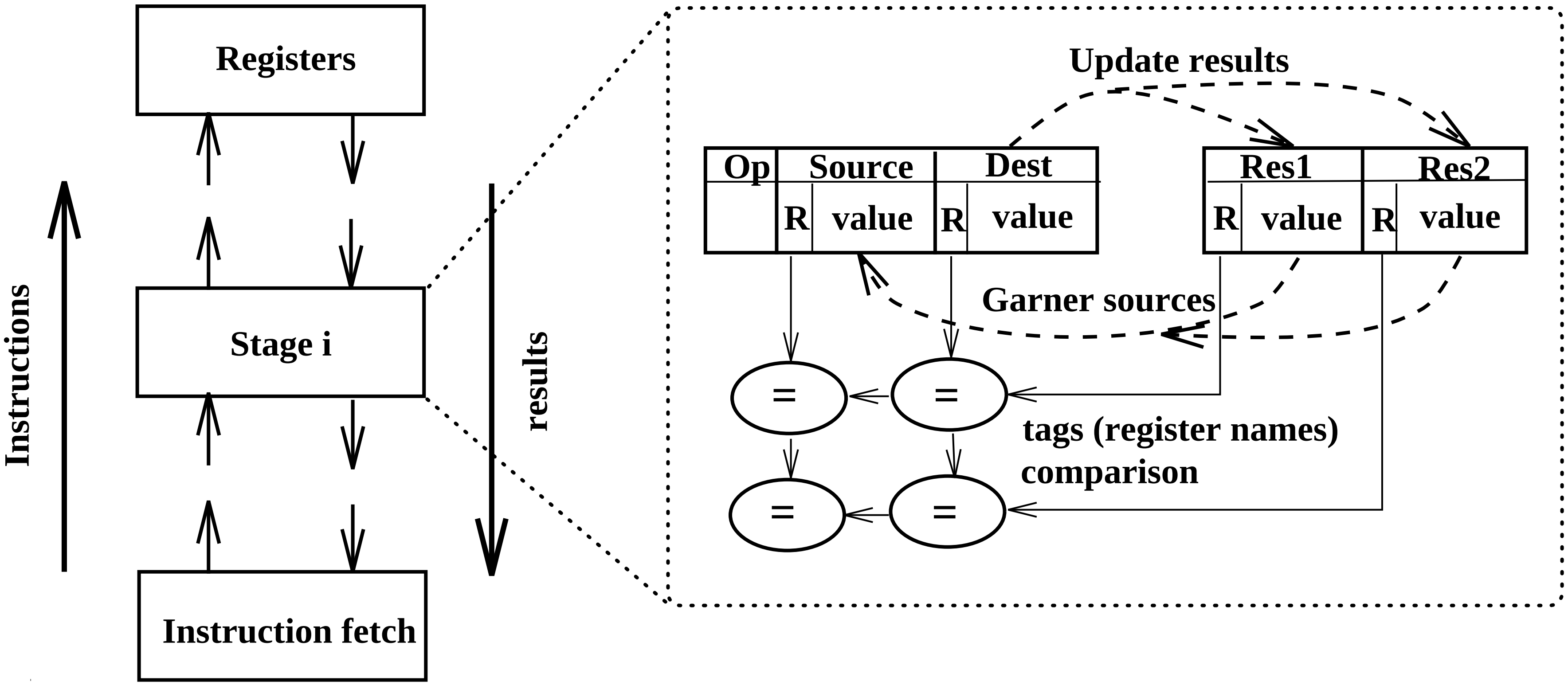
- Counterflow pipeline:
  instructions and data flow in opposite directions

- [Sproull, Sutherland, Molnar 94] Sun Microsystems Labs.

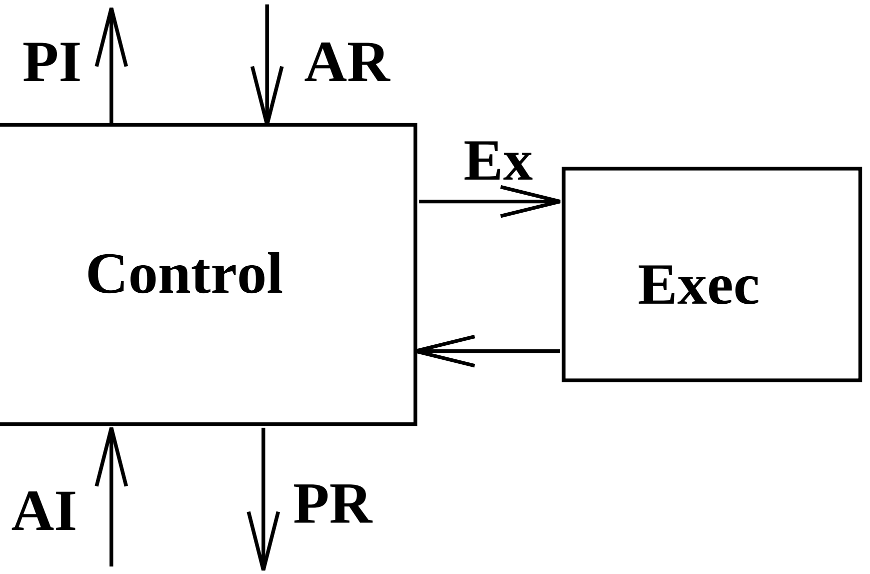
CFPP advantages and disadvantages

- Potential advantages:
  - Local control – no need in global "pipeline stall" signals
  - Regular and modular structure (proof of correctness is easy to get)
  - Local communication between pipeline stages
- Potential disadvantages:
  - Sinchronization between two flows requires arbiters
    (many points of metastability)
  - Low latency
  - Control hazards (traps, branches) require invalidating wrong results

A simplified counterflow pipeline structure



Control structure



Events:

AI - accept instruction

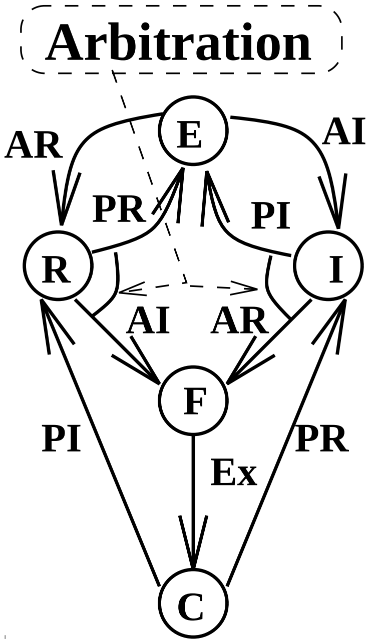
PI - pass instruction

AR - accept result

PR - pass result

Ex - execute instr.

Design challenge: CFPP state graph



States:

E - Empty

I - Instruction

R - Result

F - Full

C - Complete

Events:

AI - accept instruction

PI - pass instruction

AR - accept result

PR - pass result

Ex - execute instr.

How to synthesize this state graph *formally*?

A few solutions proposed based on STG and CSP

Tangram Approach: asynchronous for low power

- Tangram = CSP-based language

- Direct compilation to asynchronous circuits:
  one language construct $\Rightarrow$ one circuit component +
  peephole optimization

- Using STGs allows resynthesis and optimization [Peña 95]

- Low-power parts for portable audio equipment
  (DCC Error Corrector)

- [Van Berkel 93,94]

DCC Error Corrector results

- Average case switching activity (correct words – 98%) << worst case activity (incorrect words)

- Power consumption is determined:
  - For synchronous prototype – by the worst case
  - For asynchronous design – by the average case

- Asynchronous version is **five** times more economic in power consumption at normal $Vdd = 5V$.

- Supply voltage scaling allows futher power reduction

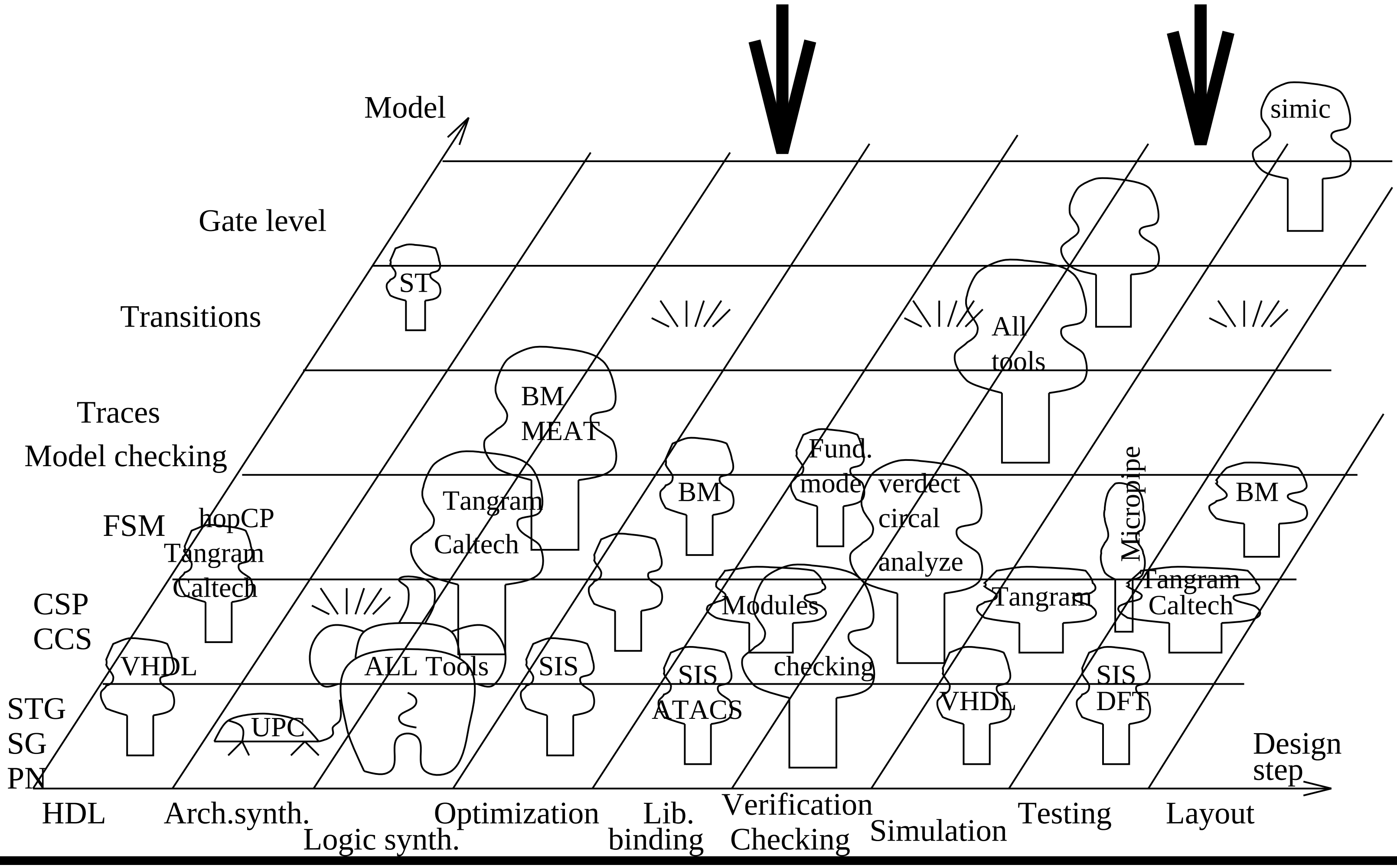
CAD tools for asynchronous circuits

- Synthesis: SIS (U.C.Berkeley), Tangram (Philips Research Labs), Petrify (UPC, Barcelona), Caltech tools, Forcage (Aizu), Assassin (IMEC), Syn (Stanford), MEAT (HP Labs), SHILPA (Univ. Utah), Burst Mode tools (Stanford/Columbia Univ.) etc.

- Verification: AVER (Stanford), Versify (UPC, Barcelona), Verdict (Waterloo), Unfold (Aizu), Forcage (Aizu), etc.

For a more complete list look at the asynchronous design home page:
http://www.cs.man.ac.uk/amulet/async/index.html
and at the Async96 symposium home page:
http://www.u-aizu.ac.jp/async96/

ACAD grove as a conclusion



More conclusions

- Asynchronous design is a challenging field
- Asynchronous synthesis is reaching maturity
- Asynchronous circuits take more area
  ... but area is becoming less important than speed, power and **correctness**
- Commercial CAD tools for asynchronous circuit design may be coming...