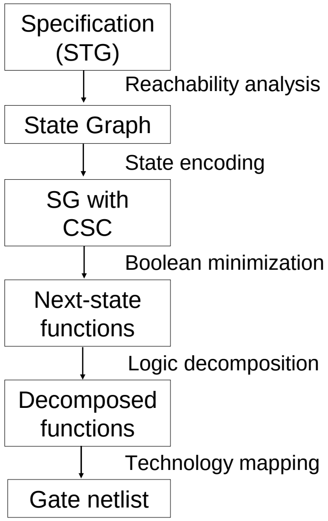
Petrify: Method and Tool for Synthesis of Asynchronous Controllers and Interfaces

Jordi Cortadella (UPC, Barcelona, Spain),
Mike Kishinevsky (Intel Strategic CAD Labs, Oregon),
Alex Kondratyev (Berkeley Cadence Research Labs, CA),
Luciano Lavagno (Politecnico di Torino, Italy),
Alex Yakovlev (University of Newcastle, UK)

Outline

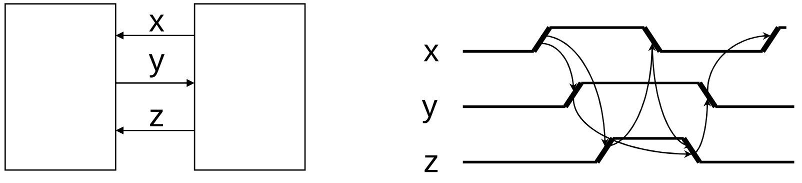
- Part 1: The Petrify Method
  - – Overview of the Petrify synthesis flow
  - – Specification: Signal Tranistion Graphs
  - – State graph and next-state functions
  - – State encoding
  - – Implementation conditions
  - – Speed-independent circuits
    - Complex gates
    - C-element architecture

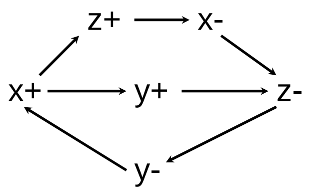
Design flow



Specification (STG)
Reachability analysis
State Graph
State encoding
SG with CSC
Boolean minimization
Next-state functions
Logic decomposition
Decomposed functions
Technology mapping
Gate netlist

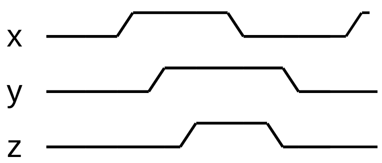
Specification

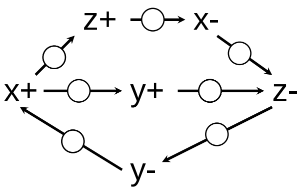




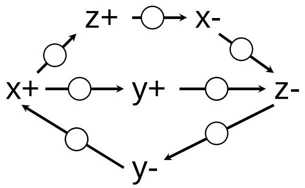
Signal Transition Graph (STG)

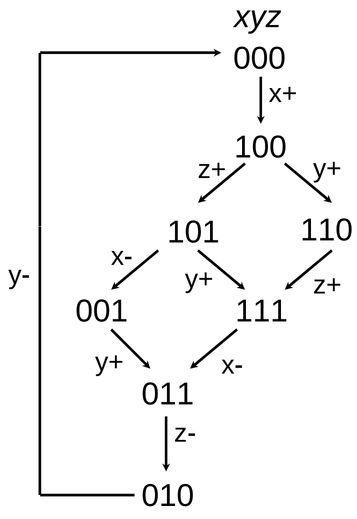
Token flow





State graph



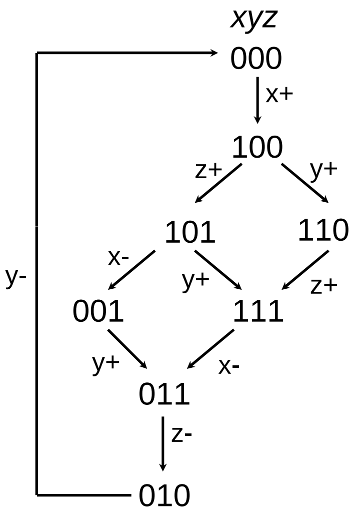


Next-state functions

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

$$y = z + x$$

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

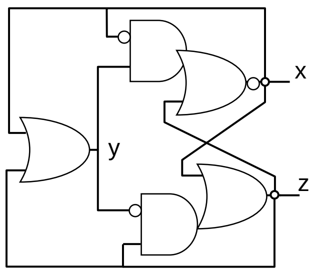


Gate netlist

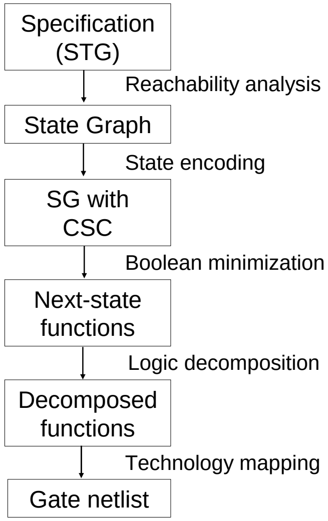
$$x = \bar{z} \cdot (x + \bar{y})$$

$$y = z + x$$

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



Design flow



Specification (STG)

Reachability analysis

State Graph

State encoding

SG with CSC

Boolean minimization

Next-state functions

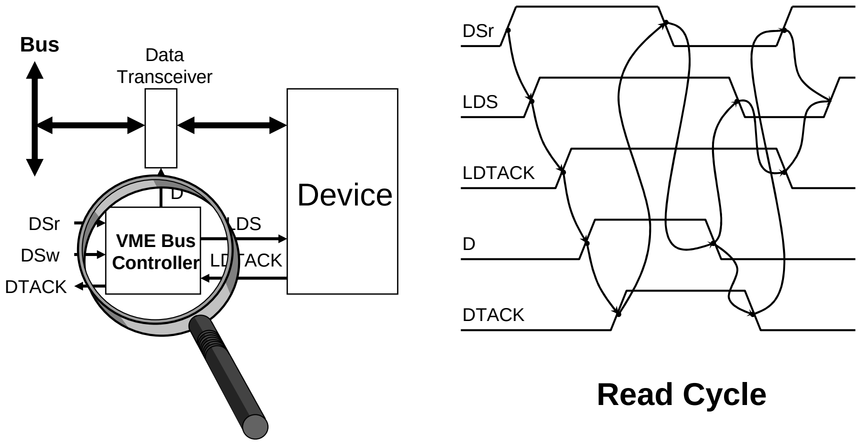
Logic decomposition

Decomposed functions

Technology mapping

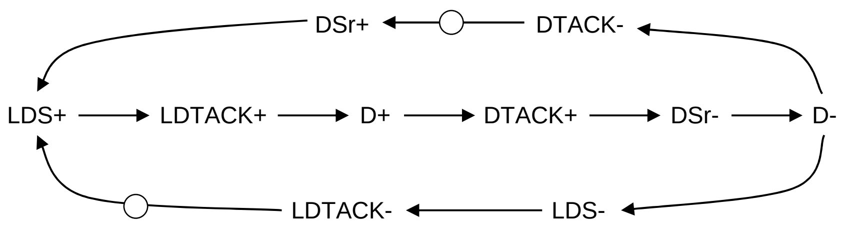
Gate netlist

VME bus



Read Cycle

STG for the READ cycle



DSr+

DTACK-

LDS+

LDTACK+

D+

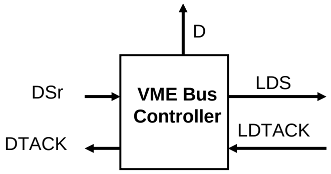
DTACK+

DSr-

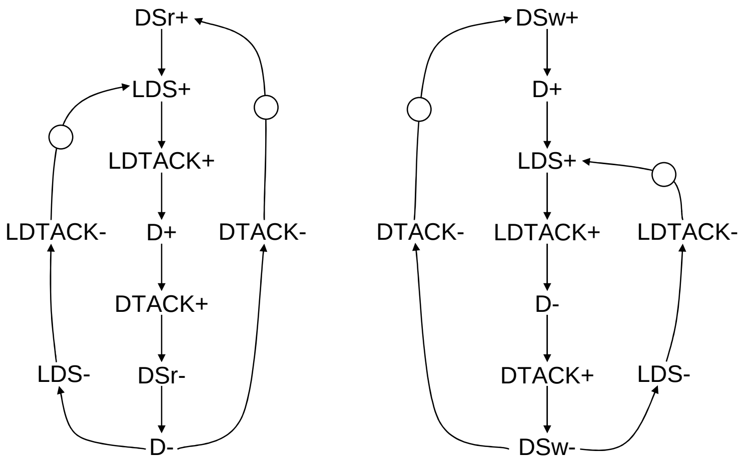
D-

LDTACK-

LDS-



Choice: Read and Write cycles



DSr+

LDS+

LDTACK+

LDTACK-

D+

DTACK-

DTACK+

LDS-

DSr-

D-

DSw+

D+

LDS+

DTACK-

LDTACK+

LDTACK-

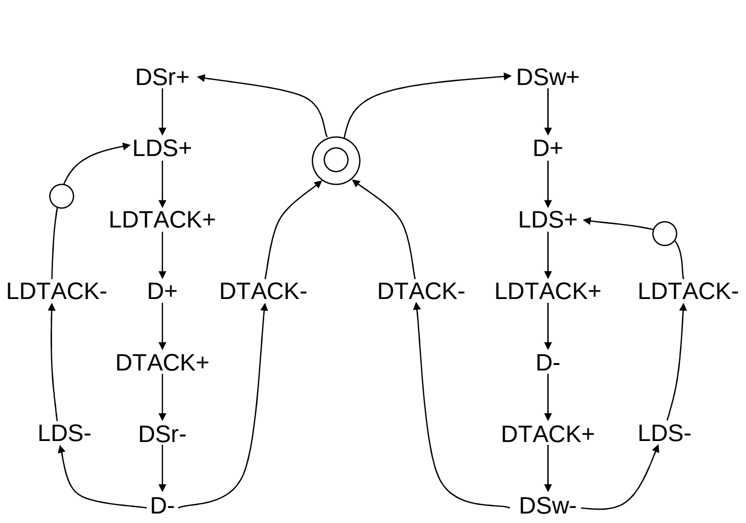
D-

DTACK+

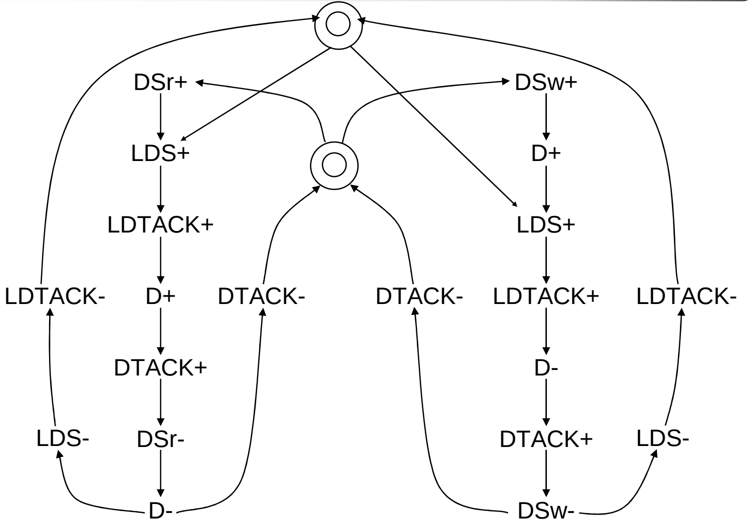
LDS-

DSw-

Choice: Read and Write cycles



Choice: Read and Write cycles



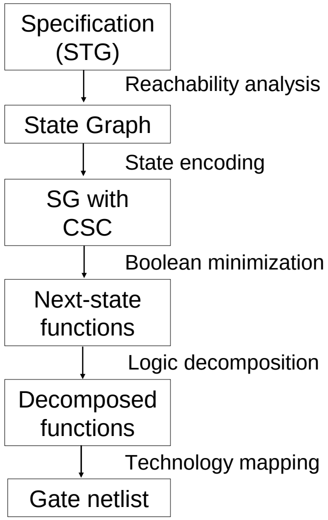
Circuit synthesis

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

Speed independence

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

Design flow



Specification (STG)

Reachability analysis

State Graph

State encoding

SG with CSC

Boolean minimization

Next-state functions

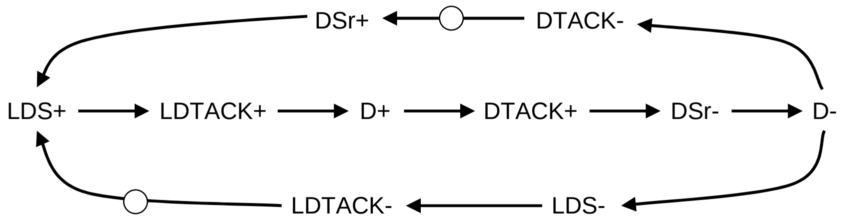
Logic decomposition

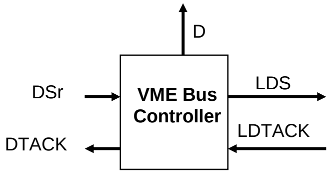
Decomposed functions

Technology mapping

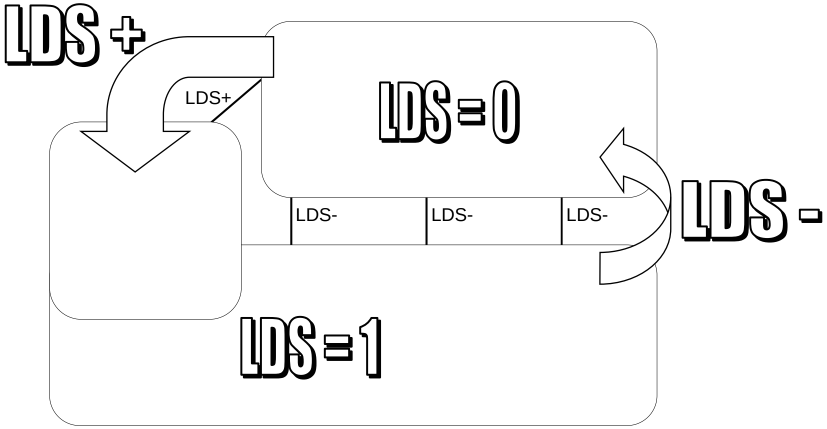
Gate netlist

STG for the READ cycle

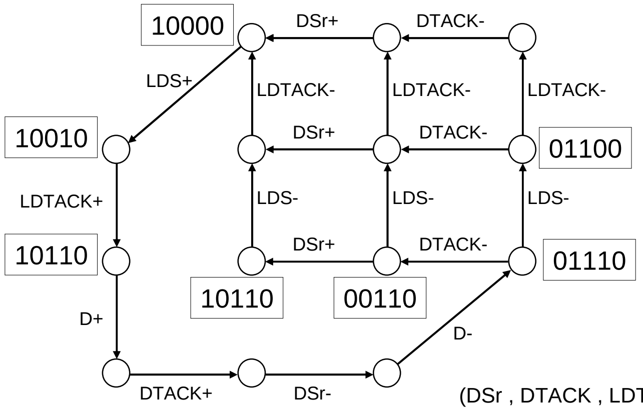




Binary encoding of signals

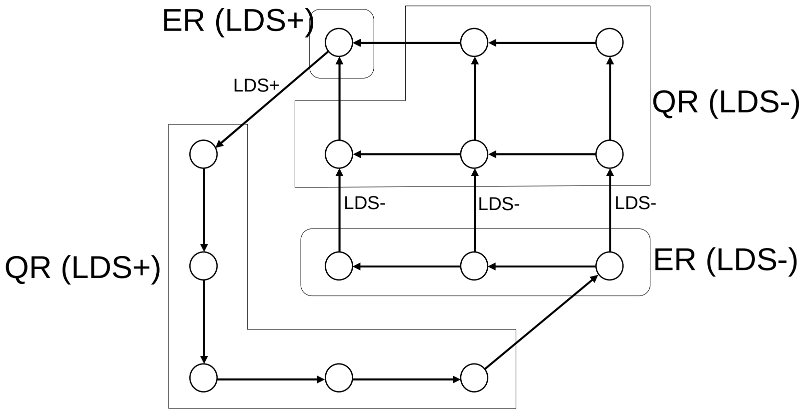


Binary encoding of signals

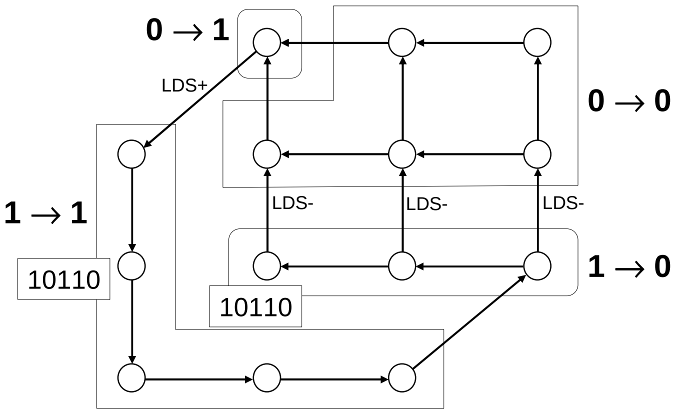


(DSr , DTACK , LDTACK , LDS , D)

Excitation / Quiescent Regions



Next-state function



Karnaugh map for LDS

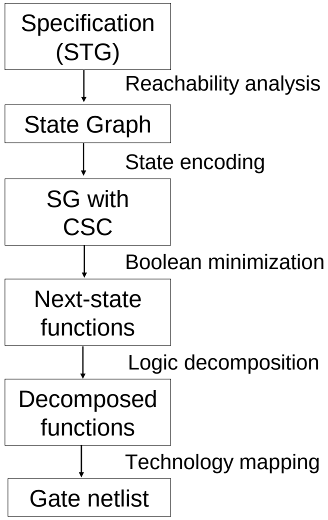
LDS = 0

D LDTACK \ DTACK DSr	00	01	11	10
00	0	0	-	1
01	-	-	-	-
11	-	-	-	-
10	0	0	-	0

LDS = 1

D LDTACK \ DTACK DSr	00	01	11	10
00	-	-	-	1
01	-	-	-	-
11	-	1	1	1
10	0	0	-	0/1?

Design flow



Specification (STG)

Reachability analysis

State Graph

State encoding

SG with CSC

Boolean minimization

Next-state functions

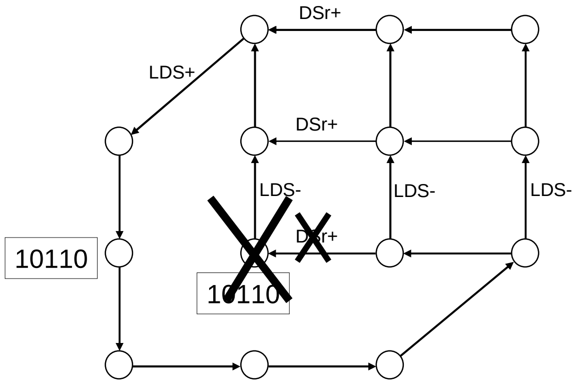
Logic decomposition

Decomposed functions

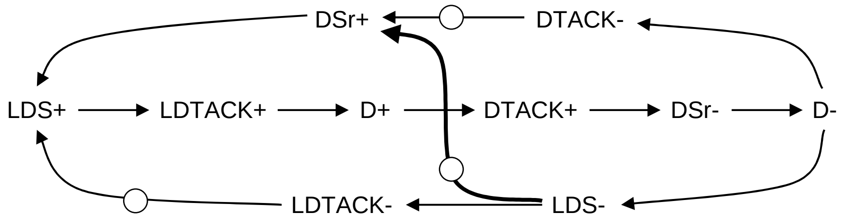
Technology mapping

Gate netlist

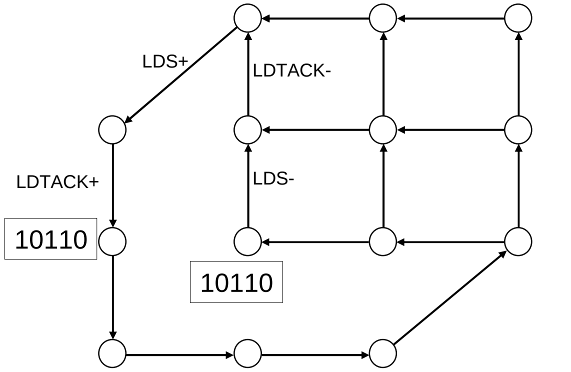
Concurrency reduction



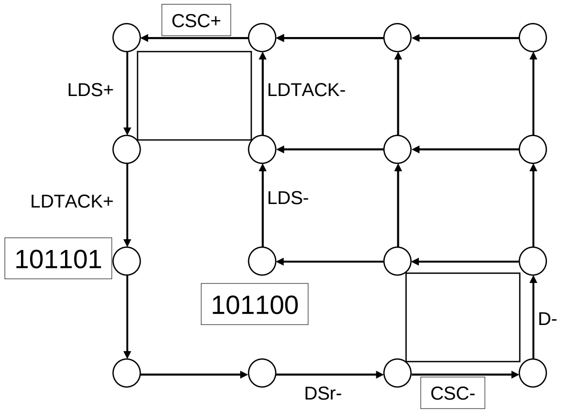
Concurrency reduction



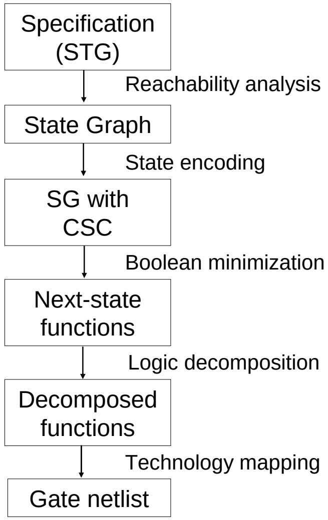
State encoding conflicts



Signal Insertion



Design flow



Specification (STG)

Reachability analysis

State Graph

State encoding

SG with CSC

Boolean minimization

Next-state functions

Logic decomposition

Decomposed functions

Technology mapping

Gate netlist

Complex-gate implementation

$$LDS = D + \mathrm{csc}$$

$$DTACK = D$$

$$D = LDTACK \cdot \mathrm{csc}$$

$$\mathrm{csc} = DSr \cdot (\mathrm{csc} + \overline{LDTACK})$$

Implementation conditions

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

Implementation conditions

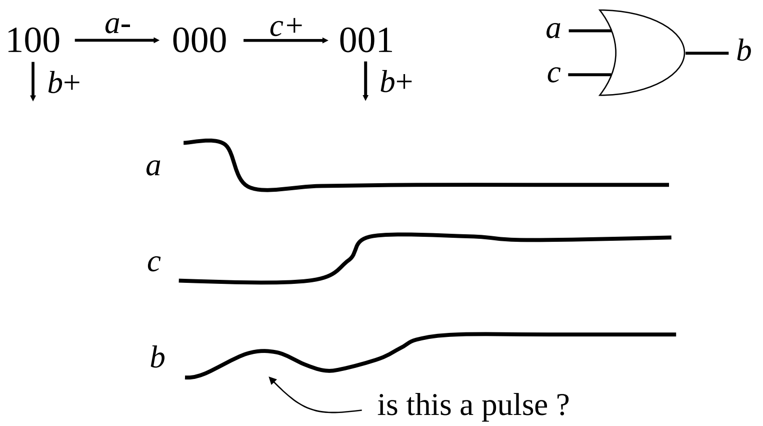
- Consistency + CSC + persistency

⇓

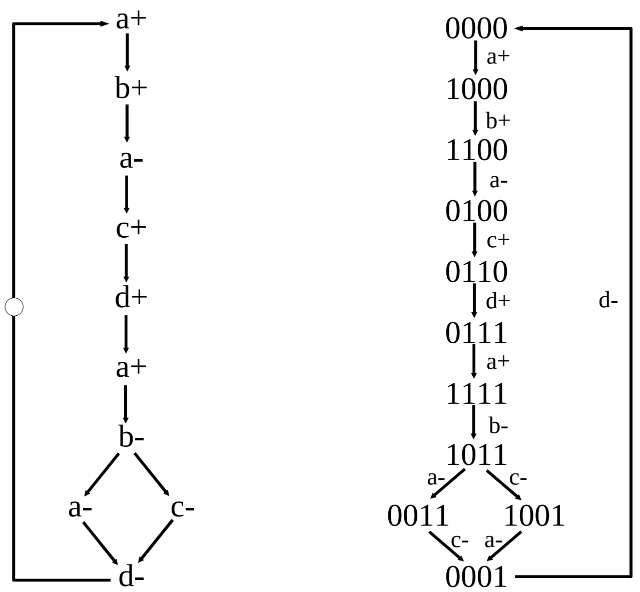
- There exists a speed-independent circuit that implements the behavior of the STG

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

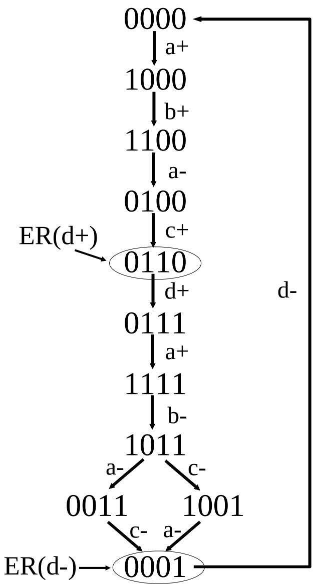
Persistency



Speed independence ⇒ glitch-free output behavior under any delay



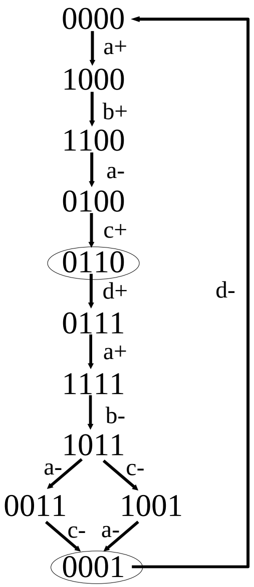
cd \ ab	00	01	11	10
00	0	0	0	0
01	0			1
11	1	1	1	1
10		1		



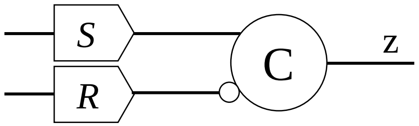
cd \ ab	00	01	11	10
00	0	0	0	0
01	0			1
11	1	1	1	1
10		1		

$$d = ad + \overline{a}c$$

Complex gate



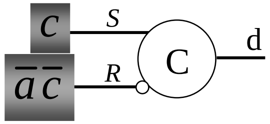
Implementation with C elements

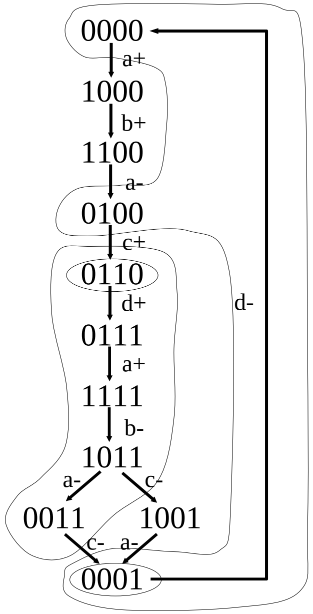


• • • → S+ → z+ → S- → R+ → z- → R- → • • •

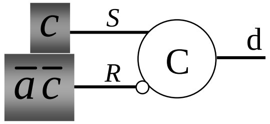
- *S* (set) and *R* (reset) must be mutually exclusive
- *S* must cover *ER(z+)* and must not intersect *ER(z-)* ∪ *QR(z-)*
- *R* must cover *ER(z-)* and must not intersect *ER(z+)* ∪ *QR(z+)*

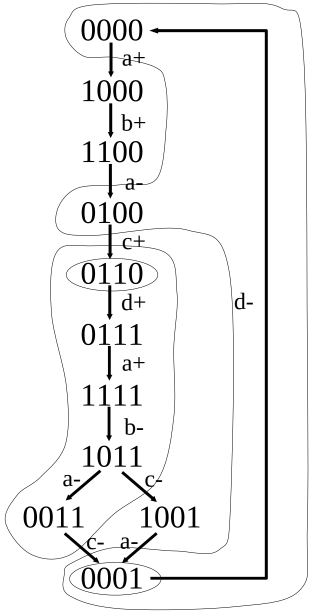
cd \ ab	00	01	11	10
00	0	0	0	0
01	0			1
11	1	1	1	1
10		1		





but ...





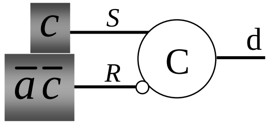
Assume that $R=\overline{a}\overline{c}$ has an unbounded delay

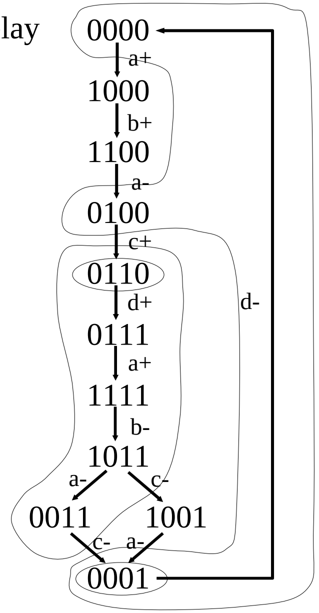
Starting from state 0000 (R=1 and S=0):

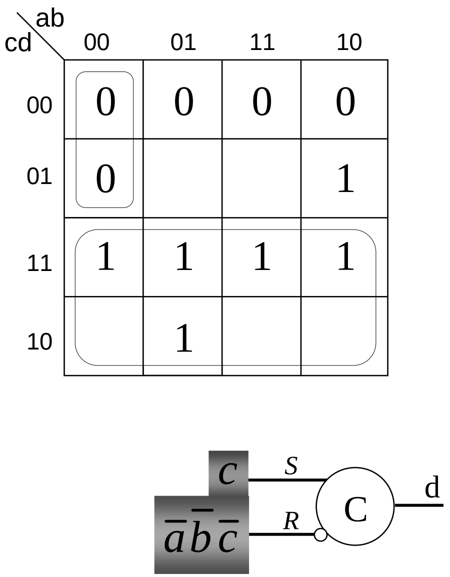
a+ ; R- ; b+ ; a- ; c+ ; S+ ; d+ ;

↑

R+ disabled (potential glitch)



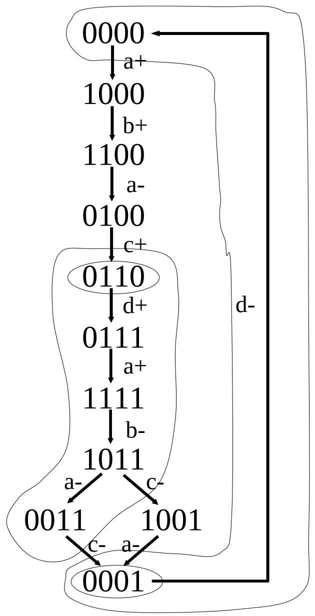




cd \ ab	00	01	11	10
00	0	0	0	0
01	0			1
11	1	1	1	1
10		1		

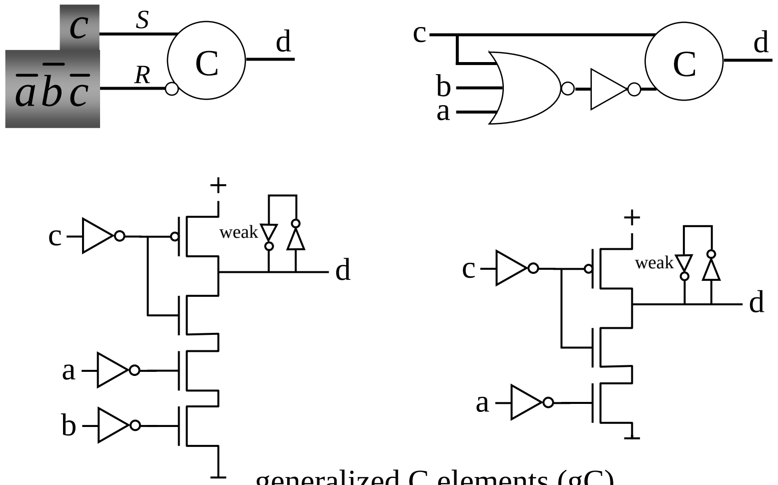
c

$\overline{a}\,\overline{b}\,\overline{c}$



Monotonic covers

C-based implementations



c

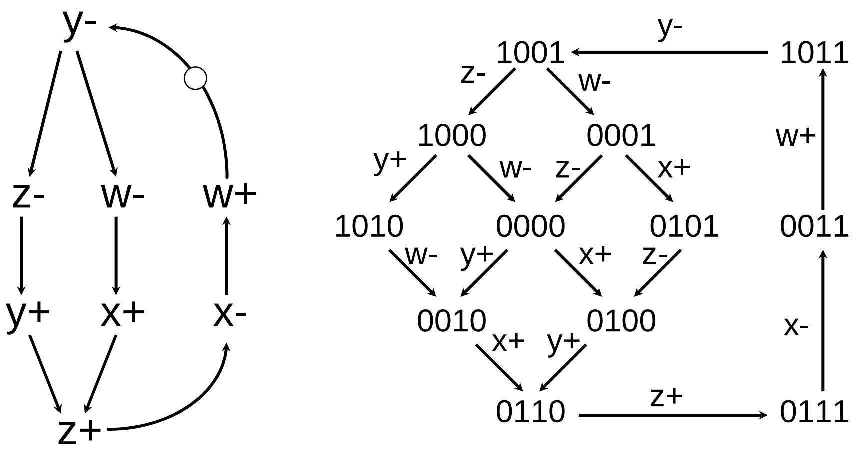
$\overline{a}\,\overline{b}\,\overline{c}$

generalized C elements (gC)

Speed-independent implementations

- Implementation conditions
  - Consistency
  - Complete state coding
  - Persistency
- Circuit architectures
  - Complex (hazard-free) gates
  - C elements with monotonic covers
  - ...

Synthesis exercise

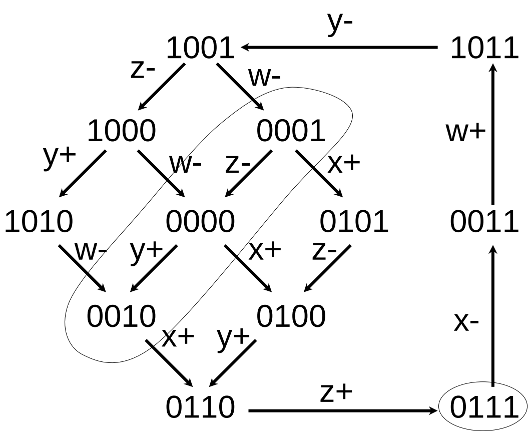


Derive circuits for signals *x* and *z* (complex gates and monotonic covers)

Synthesis exercise

yz \ wx	00	01	11	10
00	1	1	-	0
01	1	1	-	0
11	0	0	-	0
10	1	1	-	0

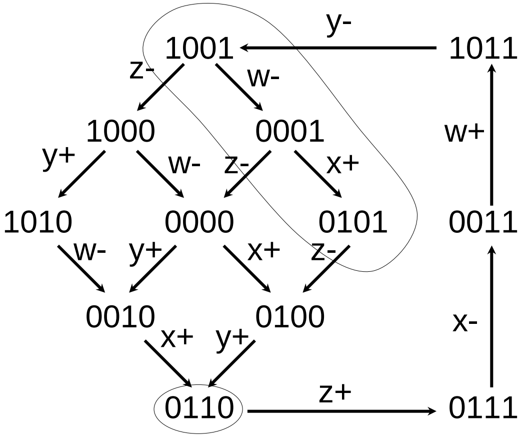
Signal *x*



Synthesis exercise

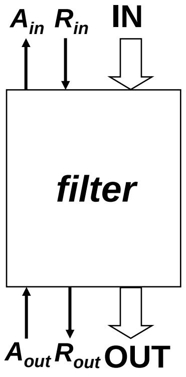
yz \ wx	00	01	11	10
00	0	0	-	0
01	0	0	-	0
11	1	1	-	1
10	0	1	-	0

Signal *z*

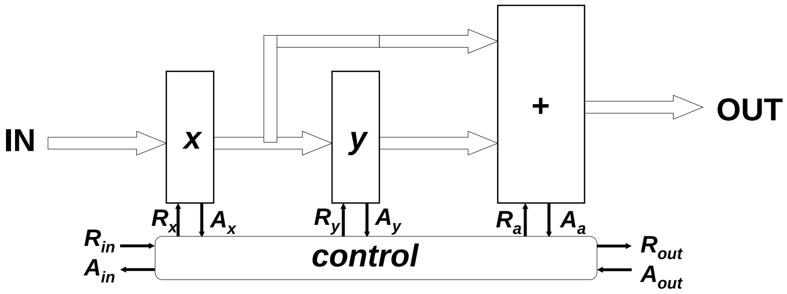


A simple filter: specification

```
y := 0;
loop
   x := READ (IN);
   WRITE (OUT, (x+y)/2);
   y := x;
end loop
```

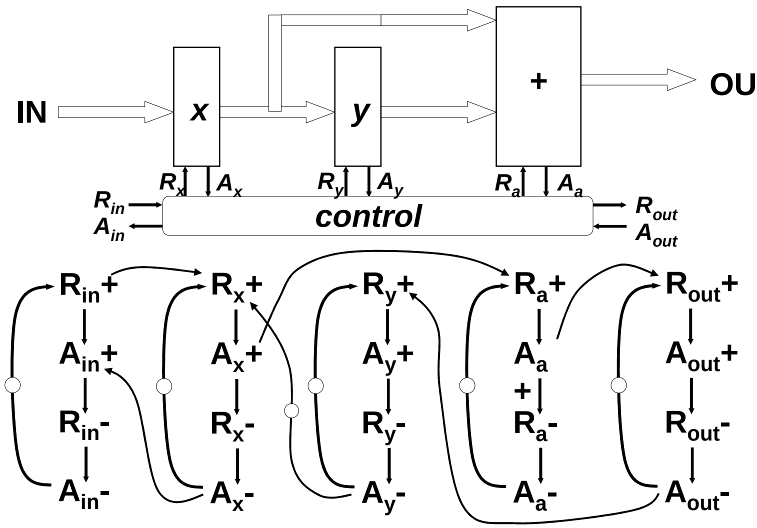


A simple filter: block diagram

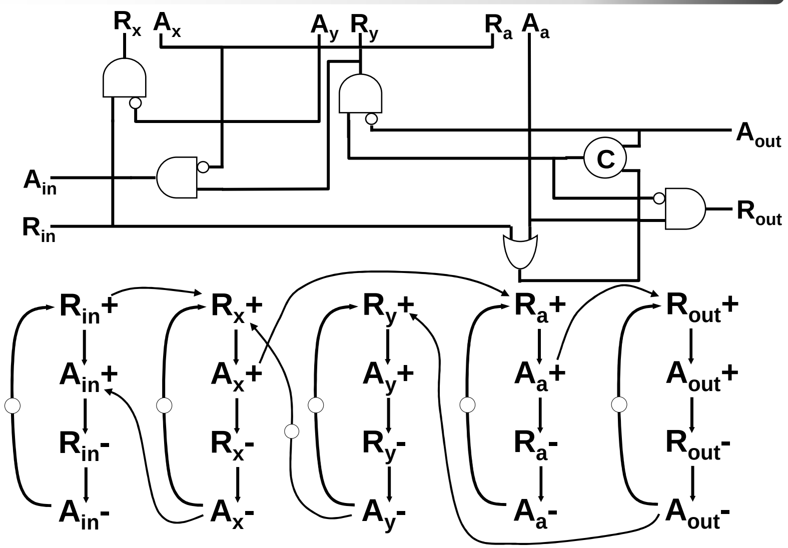


- ***x*** **and** ***y*** **are level-sensitive latches (transparent when R=1)**
- **+ is a bundled-data adder (matched delay between R_a and A_a)**
- **R_{in} indicates the validity of IN**
- **After A_{in}+ the environment is allowed to change IN**
- **(R_{out},A_{out}) control a level-sensitive latch at the output**

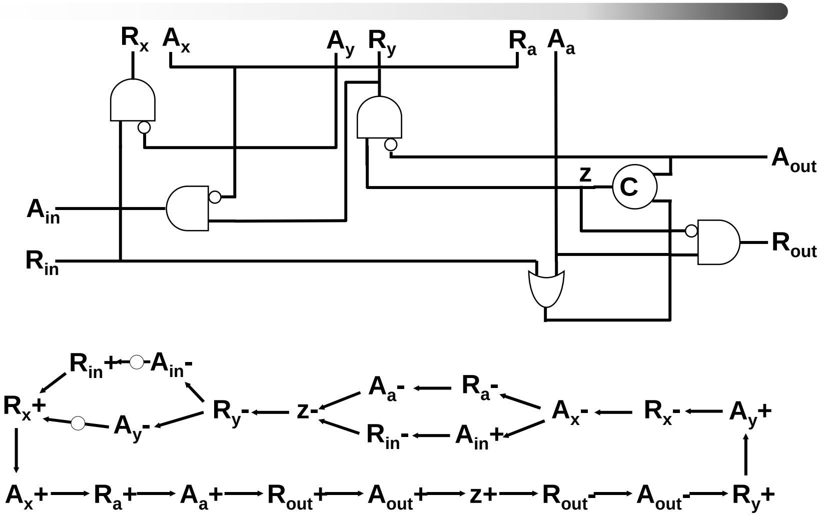
A simple filter: control spec.


IN
x
y
+
OUT
R_x A_x
R_y A_y
R_a A_a
R_{in}
A_{in}
control
R_{out}
A_{out}
R_{in}+ A_{in}+ R_{in}- A_{in}-
R_x+ A_x+ R_x- A_x-
R_y+ A_y+ R_y- A_y-
R_a+ A_a+ R_a- A_a-
R_{out}+ A_{out}+ R_{out}- A_{out}-

A simple filter: control impl.

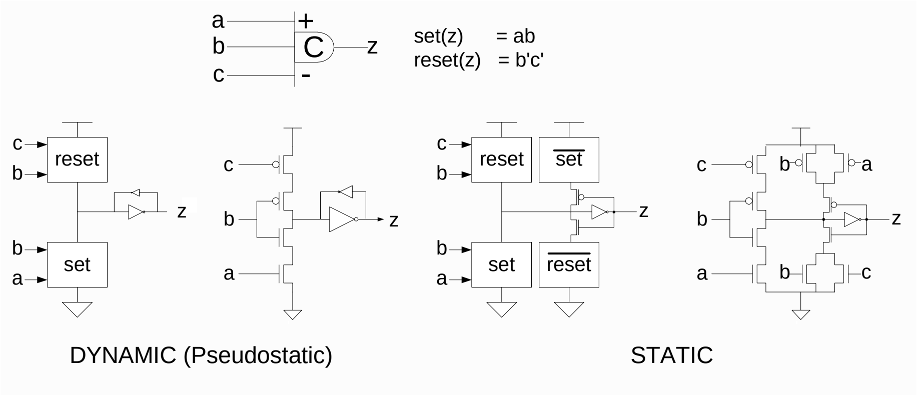

R_x A_x
A_y R_y
R_a A_a
A_{out}
C
A_{in}
R_{out}
R_{in}
R_{in}+ A_{in}+ R_{in}- A_{in}-
R_x+ A_x+ R_x- A_x-
R_y+ A_y+ R_y- A_y-
R_a+ A_a+ R_a- A_a-
R_{out}+ A_{out}+ R_{out}- A_{out}-

Control: observable behavior



Following slides borrowed from Ran Ginosar, Technion (VLSI Architectures course) with thanks

Generalized C Element

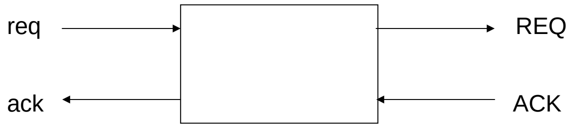


From Ran Ginosar's course

STG Rules

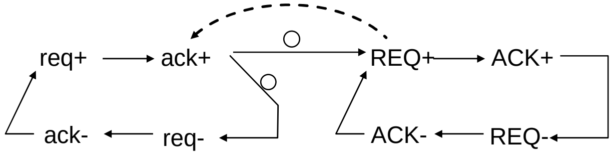
- Any STG:
  - Input free-choice—Only inputs may control the choice)
  - Bounded—Maximum k (given bound) token per place
  - Liveness—every signal transition can be activated
- STG for Speed Independent circuits:
  - Consistent state assignment—Signals strictly alternate between + and –
  - Persistency—Excited non-input signals must fire, namely they cannot be disabled by another transition
- Synthesizable STG:
  - Complete state coding—Different markings must represent different states

- We use the following circuit to explain STG rules:



1-Bounded (Safety)

- STG is *safe* if no place or arc can ever contain more than one token
- Often caused by one-sided dependency



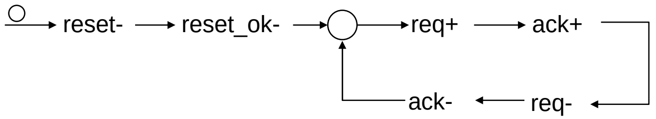
STG is not safe:

If left cycle goes fast and right cycle lags,
then arc ack+ → REQ+ accumulates tokens.
(REQ+ depends on both ack+ *and* ACK-)

Possible solution: stop left cycle by right cycle

Liveness

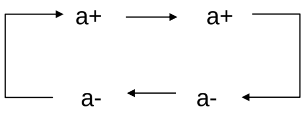
- STG is *live* if from every reachable marking, every transition can eventually be fired



  - The STG is not live:
    - Transitions reset, reset_ok cannot be repeated.
  - But non-liveness is useful for initialization

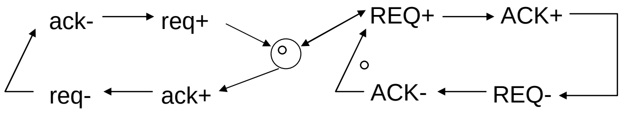
Consistent state assignment

- The following subgraph of STG makes no sense:



Persistency

- STG is *persistent* if for all non-input transitions once enabled the transition cannot be disabled by another transitions. Non-persistency may be caused by arbitration or dynamic conflict relations – STG must have places



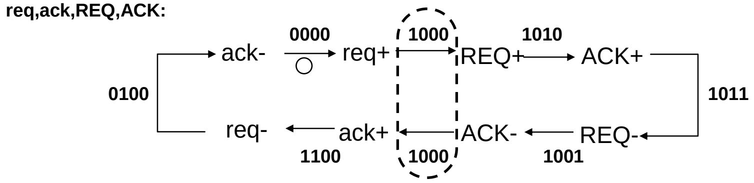
STG is not persistent:

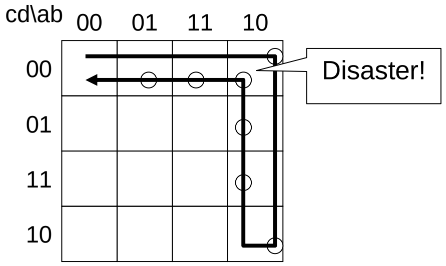
there is a place between req+ and ack+ in which a token is needed in order to fire REQ+ . So there is some sort of nondeterminism – either REQ+ manages to fire before ack+ or not

Possible solution: introduce proper dependence of the left cylce on the right one (e.g., an arc from req+ to REQ+, and from REQ+ to ack+)

Complete State Coding

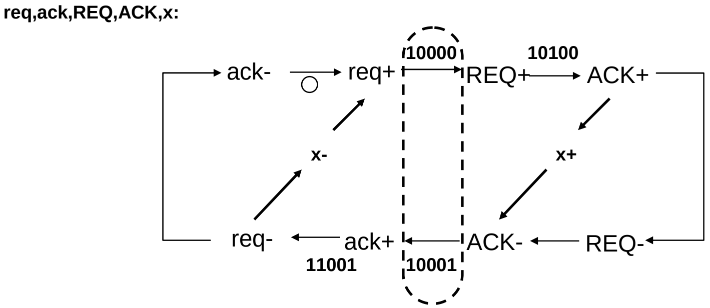
- STG has a *complete state coding* if no two different markings have identical values for all signals.





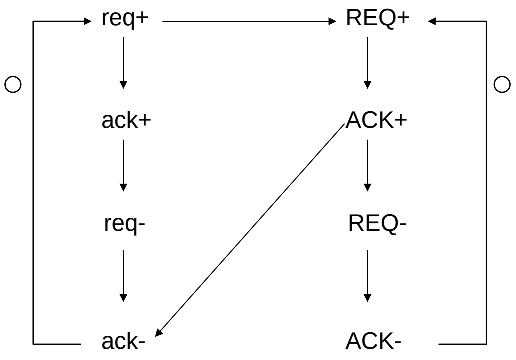
Complete State Coding

- Possible solution: Add an internal state variable



A faster STG?

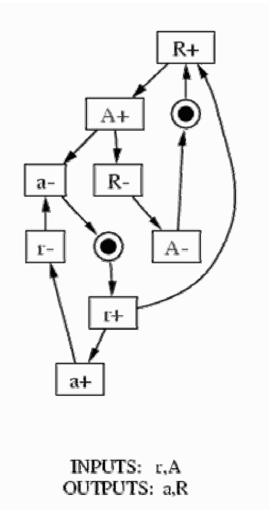
- Does it need an extra variable?



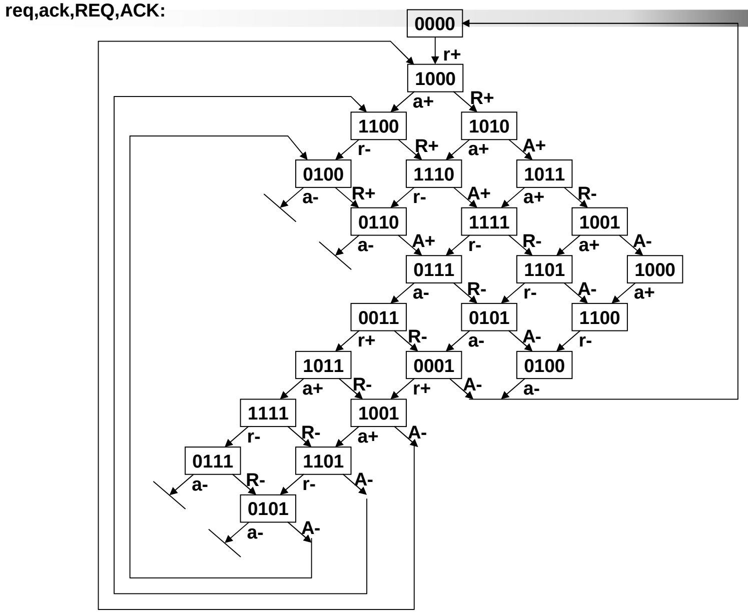
STG specification in .astg format

```
.model simple_buffer
.inputs r A
.outputs a R
.graph
# left handshake (r,a)
r+ a+
a+ r-
r- a-
a- r+
# right handshake (R,A)
R+ A+
A+ R-
R- A-
A- R+
# interaction between handshakes
r+ R+
A+ a-
.marking{<a-,r+><A-,R+>}
.end
```

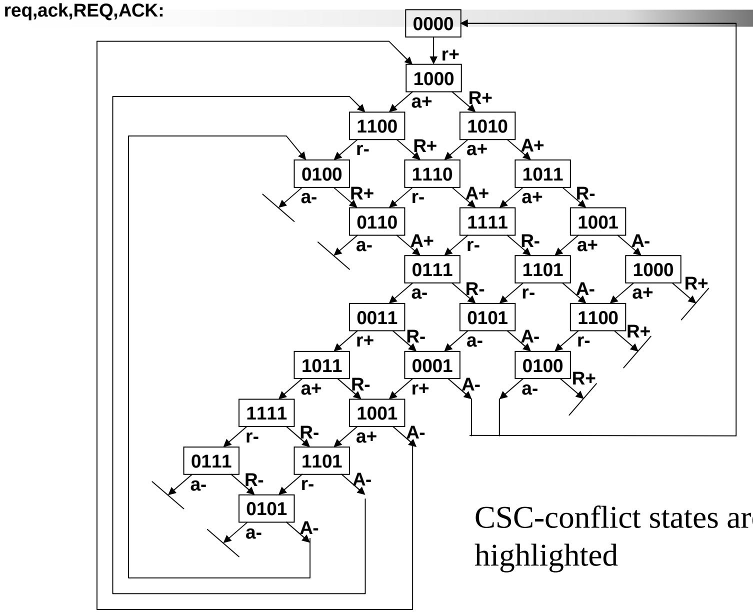
Drawn by `draw_astg`



The SG

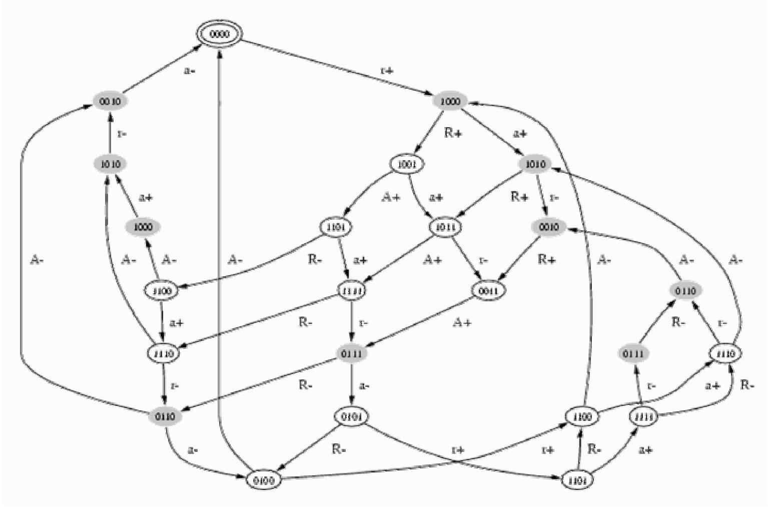


The SG

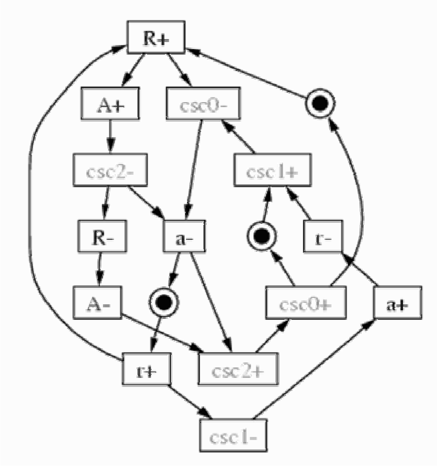


CSC-conflict states are highlighted

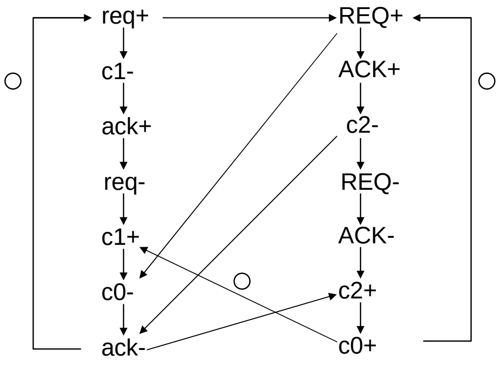
Drawn by `write_sg` & `draw_astg`



Extra states inserted by `petrify`



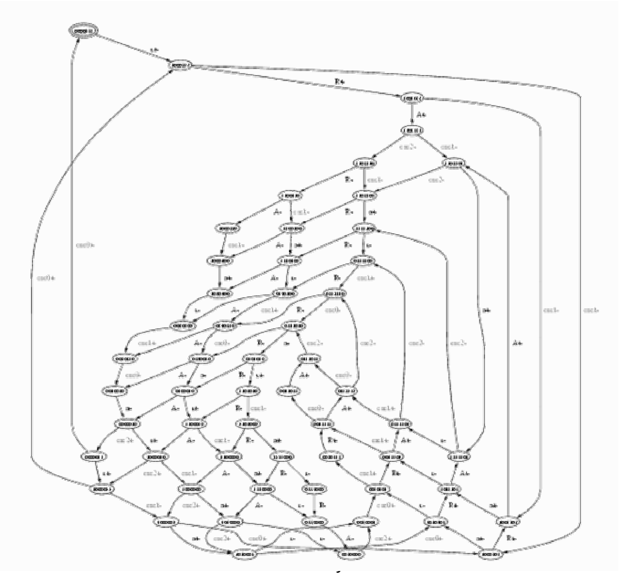
Rearranged STG



req+
c1-
ack+
req-
c1+
c0-
ack-
REQ+
ACK+
c2-
REQ-
ACK-
c2+
c0+

Initial Internal State: c0=c1=c2=1

The new State Graph…



The Synthesized Complex Gates Circuit

```
INORDER = r A a R csc0 csc1 csc2;
OUTORDER = [a] [R] [csc0] [csc1] [csc2];
[a] = a (csc2 + csc0) + csc1';
[R] = csc2 (csc0 (a + r) + R);
[csc0] = csc0 (csc1' + a') + R' csc2;
[csc1] = r' (csc0 + csc1);
[csc2] = A' (csc0' (csc1' + a') + csc2);
```

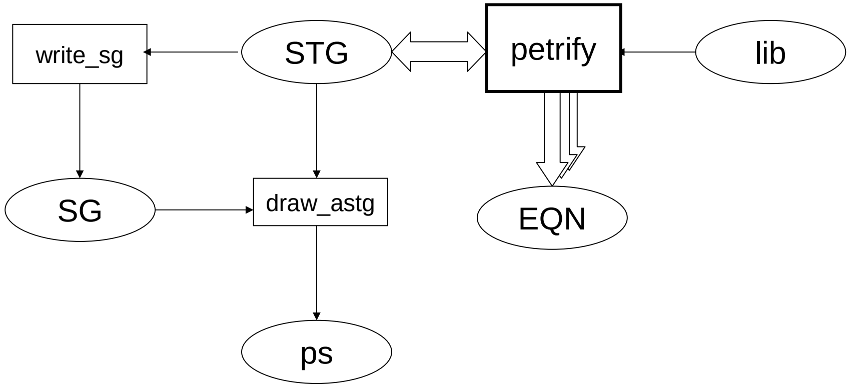
Technology Mapping

```
INORDER = r A a R csc0 csc1 csc2;
OUTORDER = [a] [R] [csc0] [csc1] [csc2];
[0] = R';                              # gate inv:      combinational
[1] = [0]' A' + csc2';                 # gate oai12:    combinational
[a] = a csc0' + [1];                   # gate sr_nor:   asynch
[3] = csc1';                           # gate inv:      combinational
[4] = csc0' csc2' [3]';                # gate nor3:     combinational
[5] = [4]' (csc1' + R');               # gate aoi12:    combinational
[R] = [5]';                            # gate inv:      combinational
[7] = (csc2' + a') (csc0' + A');       # gate aoi22:    combinational
[8] = csc0';                           # gate inv:      combinational
[csc0] = [8]' csc1' + [7]';            # gate oai12:    combinational
[csc1] = A' (csc0 + csc1);             # gate rs_nor:   asynch
[11] = R';                             # gate inv:      combinational
[12] = csc0' ([11]' + csc1');          # gate aoi12:    combinational
[csc2] = [12] (r' + csc2) + r' csc2;   # gate c_element1:asynch
```

The Synthesized Gen-C Circuit

```
INORDER = r A a R csc0 csc1 csc2;
OUTORDER = [a] [R] [csc0] [csc1] [csc2];
[0] = csc0' csc1 (R' + A);
[1] = csc0 csc2 (a + r);
[2] = csc2' A;
[R] = R [2]' + [1];                  # mappable onto gC
[4] = a csc1 csc2';
[csc0] = csc0 [4]' + csc2;           # mappable onto gC
[6] = r' csc0;
[csc1] = csc1 r' + [6];              # mappable onto gC
[8] = A' csc0' (csc1' + a');
[csc2] = csc2 R' + [8];              # mappable onto gC
[a] = a [0]' + csc1';                # mappable onto gC
```

Petrify Environment



Petrify

- Unix (Linux) command line tool
- **petrify –h** for help (flags etc.)
- **petrify –cg** for complex gates
- **petrify –gc** for generalized C-elements
- **petrify –tm** for tech mapping
- **draw_astg** to draw
- **write_sg** to create state graphs
- Documented on line, including tutorial

References

- See the attached Practical Exercise manual for various design examples using Petrify commands
- Additional references:
  - Petrify and all documentation can be downloaded from: http://www.lsi.upc.es/~jordic/petrify/petrify.html
  - J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno and A. Yakovlev, Logic Synthesis of Asynchronous Controllers and Interfaces, Springer, Berlin, 2002, ISBN3-540-43152-7