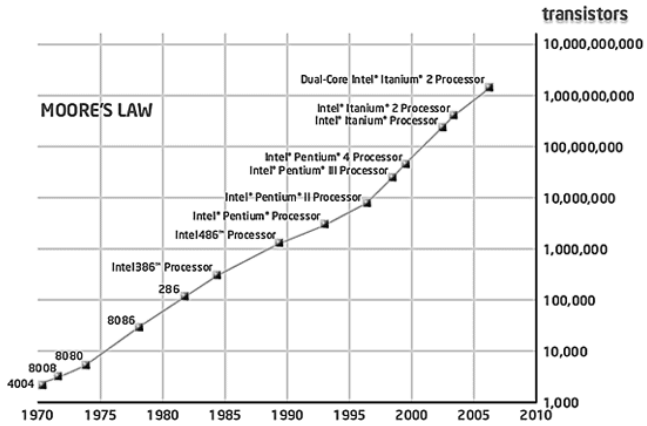
ELASTICITY AND PETRI NETS

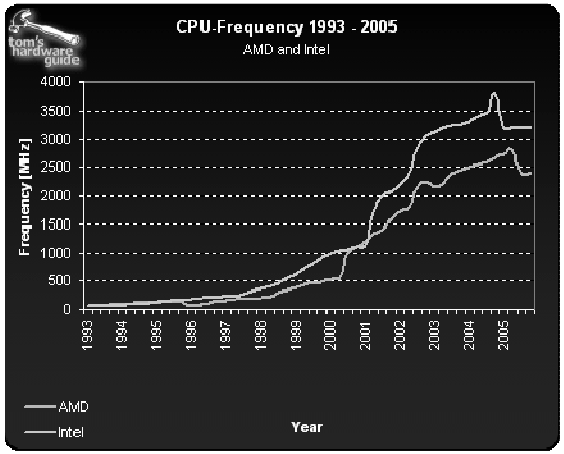
Jordi Cortadella, Universitat Politecnica de Catalunya, Barcelona
Mike Kishinevsky, Intel Corp., Strategic CAD Labs, Hillsboro

Moore's law

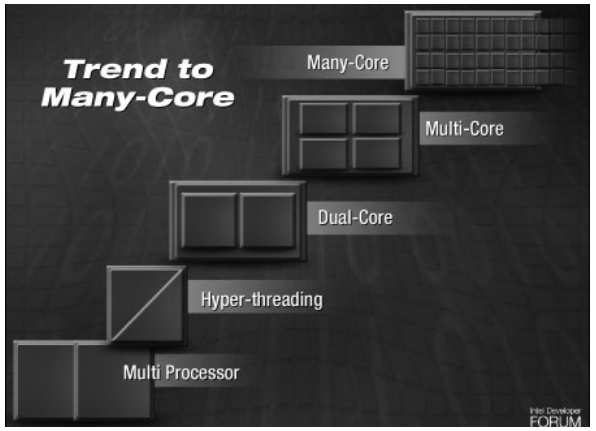


Source: Intel Corp.

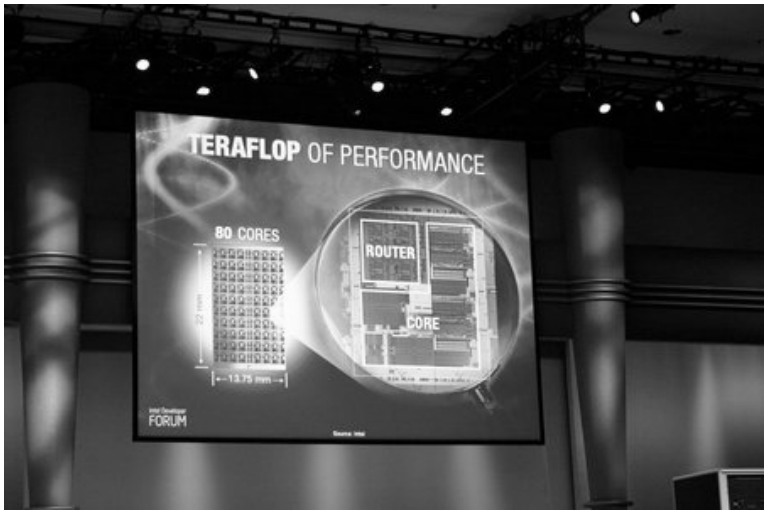
Is the GHz race over ?



Many-Core is here



Source: Intel Corp.



Why this tutorial ?

- Digital circuits are complex concurrent systems
- Variability and power consumption are key critical aspects in deep submicron technologies
- Multi (many)-core systems will become a novel paradigm:
  - System design
  - Applications
  - Concurrent programming
- Theory of concurrency may play a relevant role in this new scenario

Elasticity

- Tolerance to delay variability
- Different forms of elasticity
  - Asynchronous: no clock
  - Synchronous: variability synchronized with a clock
- In all forms of elasticity, token-based computations are performed (req/ack, valid/stop signals are used)

Outline

- Asynchronous elastic systems
  - The basics: circuits and elasticity
  - Synthesis of asynchronous circuits from Petri nets
  - Modern methods for the synthesis of large controllers
  - De-synchronization: from synchronous to asynchronous
- Synchronous elastic systems
  - Basics of synchronous elastic systems
  - Early evaluation and performance analysis
  - Optimization of elastic systems and their correctness

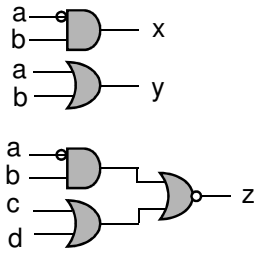
THE BASICS: CIRCUITS AND ELASTICITY

Outline

- Gates, latches and flip-flops. Combinational and sequential circuits.
- Basic concepts on asynchronous circuit design.
- Petri net models for asynchronous controllers. Signal Transition Graphs.

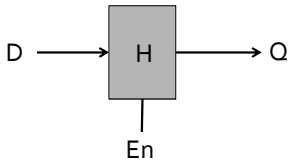
Boolean functions

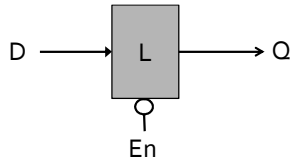
Composed from logic gates



$$x = \neg a \wedge b$$
$$y = a \vee b$$
$$z = \neg((\neg a \wedge b) \vee (c \vee d))$$

Memory elements: latches





Active high:

En = 0 (opaque): Q = prev(Q)

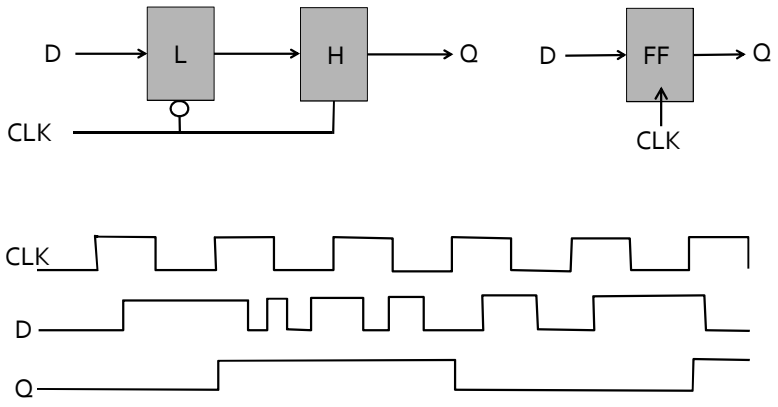
En = 1 (transparent): Q = D

Active low:

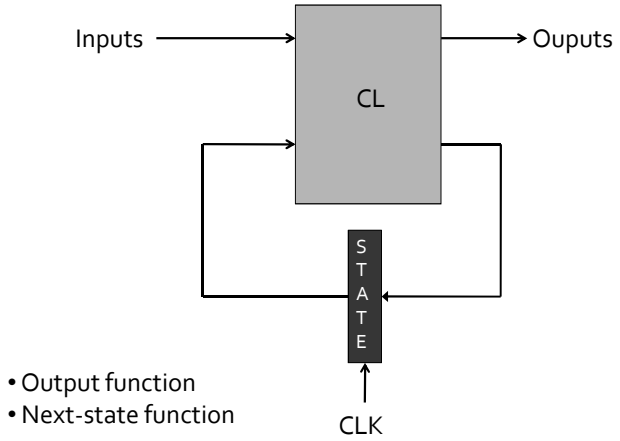
En = 1 (opaque): Q = prev(Q)

En = 0 (transparent): Q = D

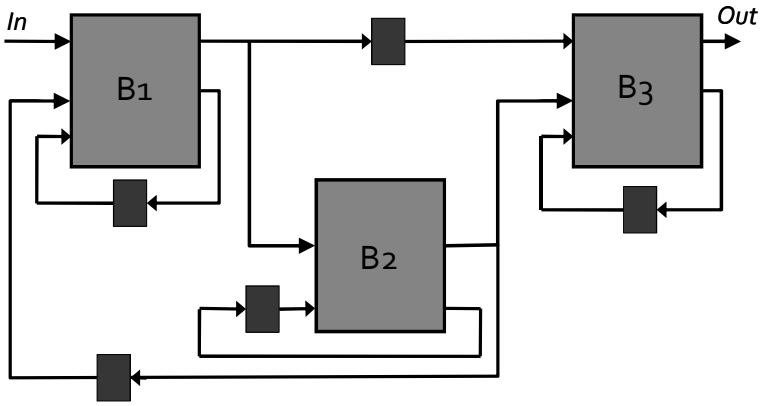
Memory elements: flip-flop



Finite-state automata

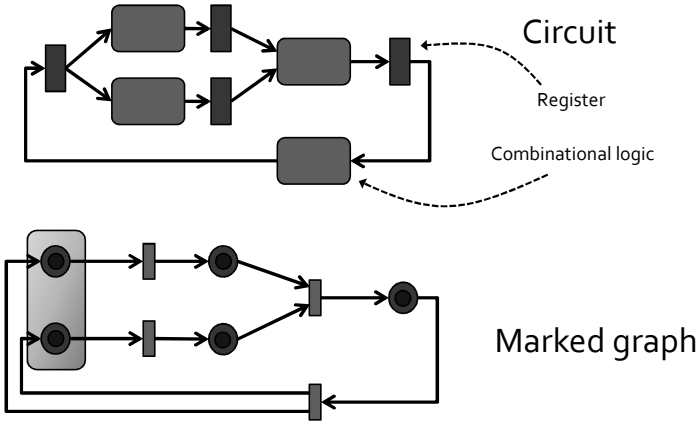


Network of Computing Units



No combinational cycles

Marked Graph Model

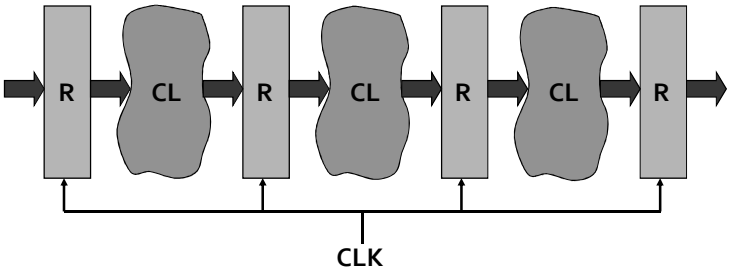


BASIC CONCEPTS ON ASYNCHRONOUS CIRCUIT DESIGN

Outline

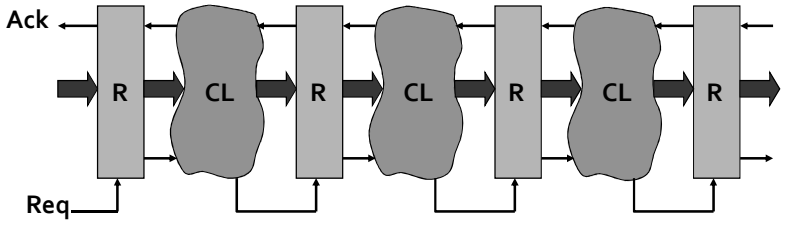
- What is an asynchronous circuit ?
- Asynchronous communication
- Asynchronous design styles (Micropipelines)
- Asynchronous logic building blocks
- Control specification and implementation
- Delay models and classes of async circuits
- Channel-based design
- Why asynchronous circuits ?

Synchronous circuit



Implicit (global) synchronization between blocks
Clock period > Max Delay (CL + R)

Asynchronous circuit

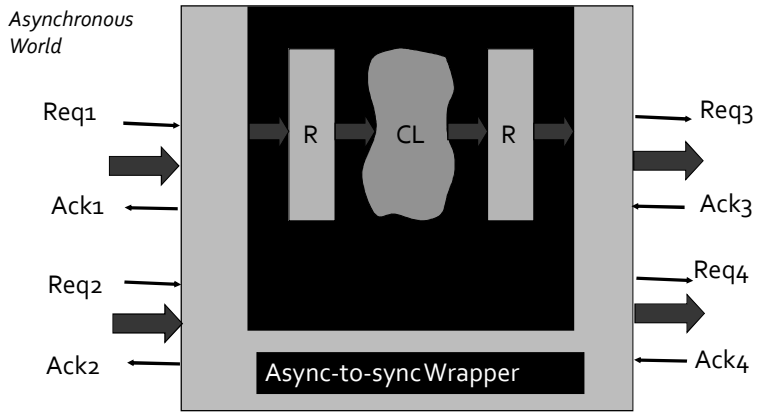


Explicit (local) synchronization:
Req / Ack handshakes

Motivation for asynchronous

- Asynchronous design is often unavoidable:
  - Asynchronous interfaces, arbiters etc.
- Modern clocking is multi–phase and distributed – and virtually 'asynchronous' (cf. GALS – next slide):
  - Mesachronous (clock travels together with data)
  - Local (possibly stretchable) clock generation
- Robust asynchronous design flow is coming (e.g. VLSI programming from Philips, Balsa from Univ. of Manchester, NCL from Theseus Logic …)

Globally Async Locally Sync (GALS)



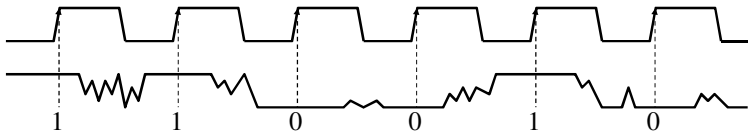
Key Design Differences

- Synchronous logic design:
  - proceeds without taking timing correctness (hazards, signal ack–ing etc.) into account
  - Combinational logic and memory latches (registers) are built separately
  - Static timing analysis of CL is sufficient to determine the Max Delay (clock period)
  - Fixed set–up and hold conditions for latches

Key Design Differences

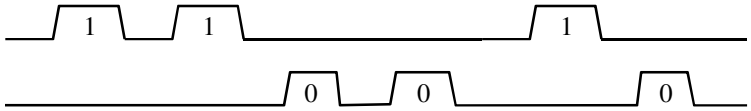
- Asynchronous logic design:
  - Must ensure hazard–freedom, signal ack–ing, local timing constraints
  - Combinational logic and memory latches (registers) are often mixed in "complex gates"
  - Dynamic timing analysis of logic is needed to determine relative delays between paths
- To avoid complex issues, circuits may be built as Delay-insensitive and/or Speed-independent (as discussed later)

Synchronous communication



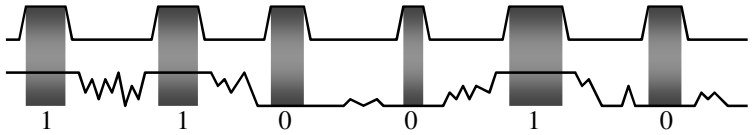
- Clock edges determine the time instants where data must be sampled
- Data wires may glitch between clock edges (set–up/hold times must be satisfied)
- Data are transmitted at a fixed rate (clock frequency)

Dual rail



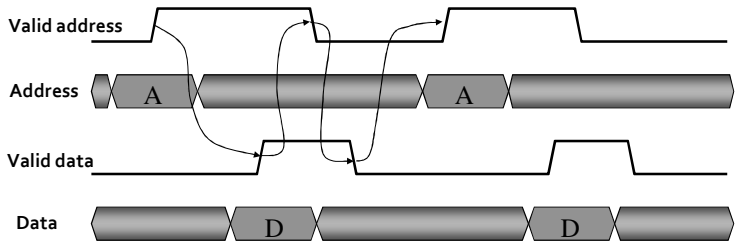
- Two wires with L(low) and H (high) per bit
  - "LL" = "spacer", "LH" = "0", "HL" = "1"
- *n*–bit data communication requires 2*n* wires
- Each bit is *self-timed*
- Other *delay-insensitive* codes exist (e.g. k-of-n) and event–based signalling (choice criteria: pin and power efficiency)

Bundled data



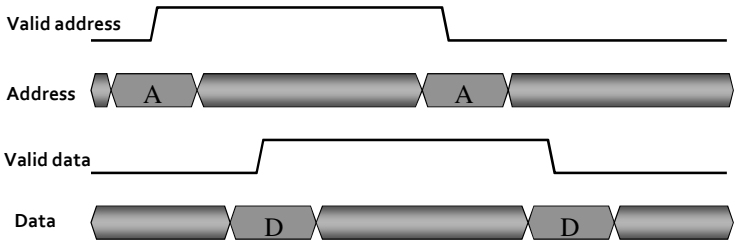
- Validity signal
  - Similar to an aperiodic local clock
- *n*–bit data communication requires *n*+1 wires
- Data wires may glitch when no valid
- Signaling protocols
  - level sensitive (latch)
  - transition sensitive (register): 2–phase / 4–phase

Example: memory read cycle



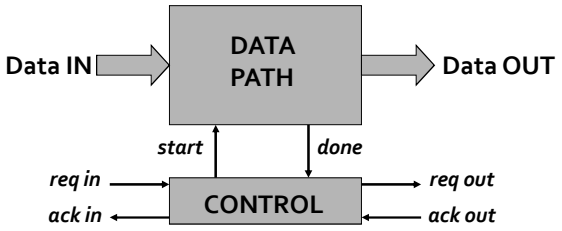
- Transition signaling, 4-phase

Example: memory read cycle



- Transition signaling, 2-phase

Asynchronous modules

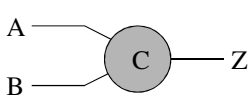


- **Signaling protocol:**

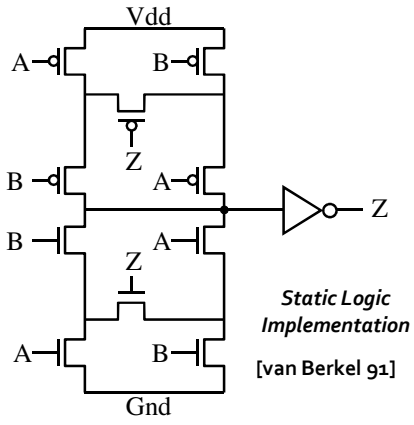
reqin+ start+ [*computation*] done+ reqout+ ackout+ ackin+
reqin- start- [*reset*] done- reqout- ackout- ackin-

(more concurrency is also possible)

Asynchronous latches: C element



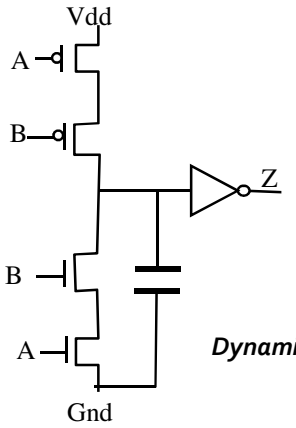
A	B	Z^+
0	0	0
0	1	Z
1	0	Z
1	1	1



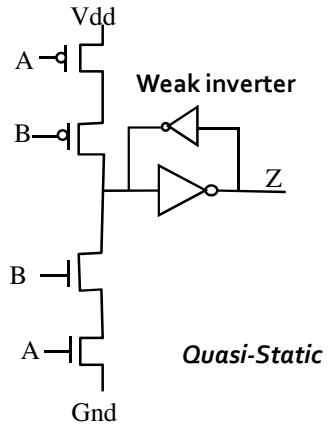
Static Logic Implementation

[van Berkel 91]

C-element: Other implementations

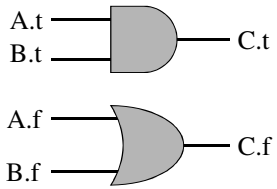


Dynamic



Quasi-Static

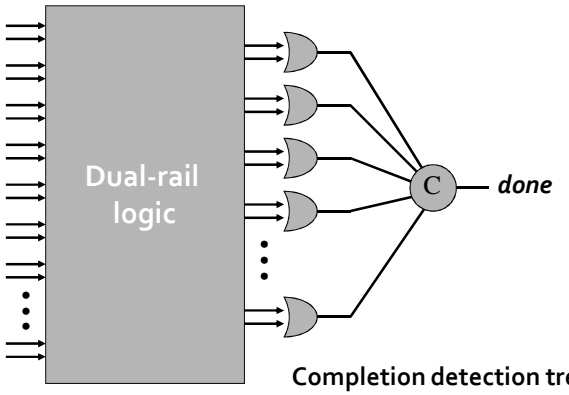
Dual-rail logic



Dual-rail AND gate

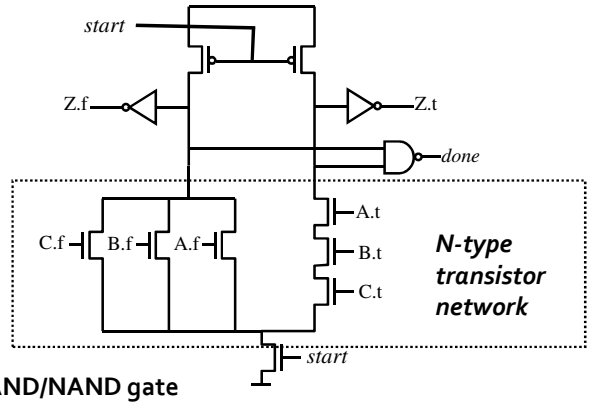
Valid behavior for monotonic environment

Completion detection



Completion detection tree

Differential cascode voltage switch logic

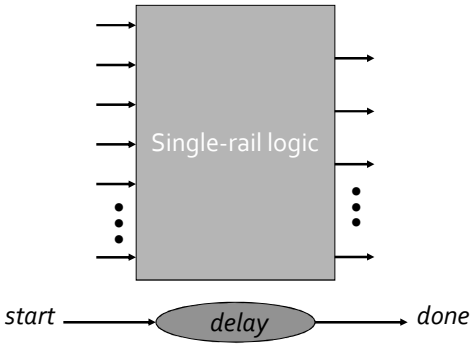


3–input AND/NAND gate

Example of dual-rail design

- Asynchronous dual-rail ripple-carry adder (A. Martin, 1991)
  - Critical delay is proportional to logN (N=number of bits)
  - 32–bit adder delay (1.6m MOSIS CMOS): 11 ns versus 40 ns for synchronous
  - Async cell transistor count = 34 versus synchronous = 28

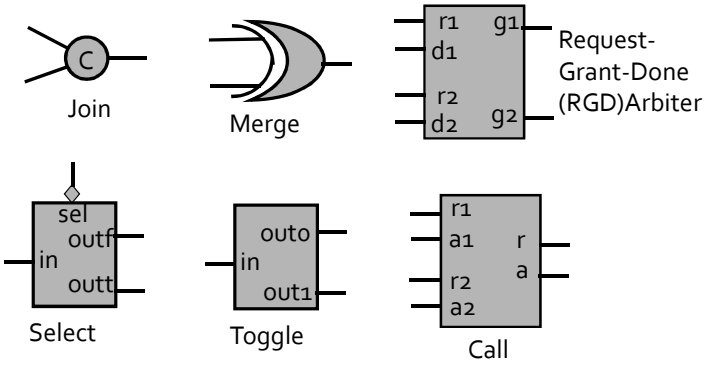
Bundled-data logic blocks



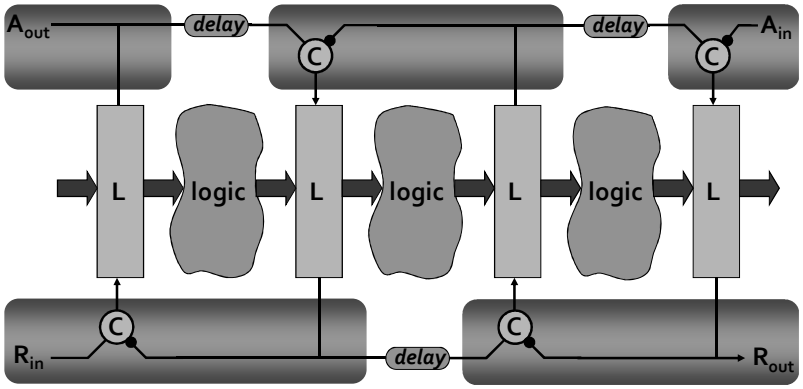
Conventional logic + matched delay

Micropipelines (Sutherland 89)

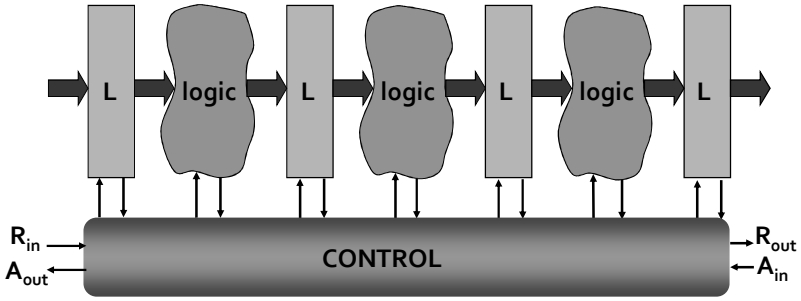
Micropipeline (2-phase) control blocks



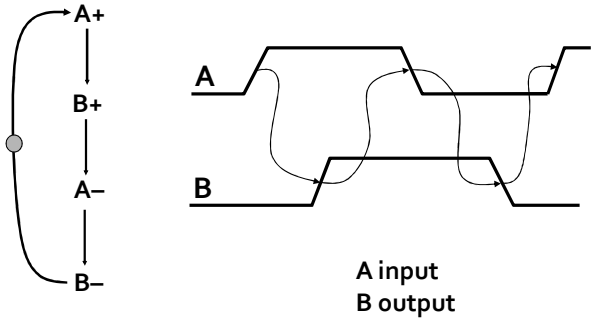
Micropipelines (Sutherland 89)



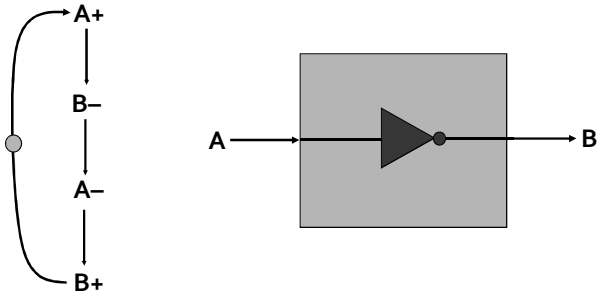
Data-path / Control



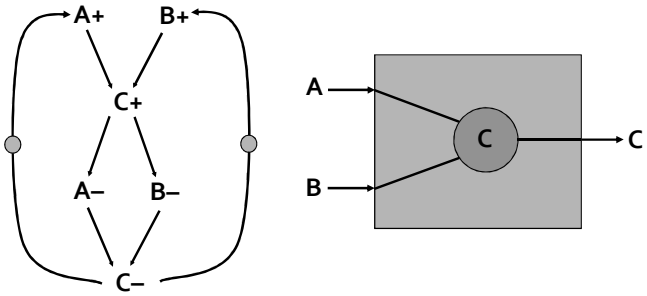
Control specification



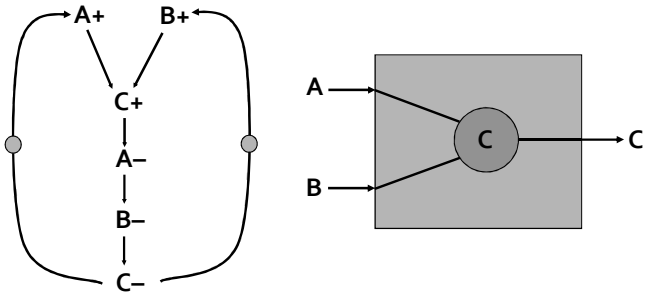
Control specification



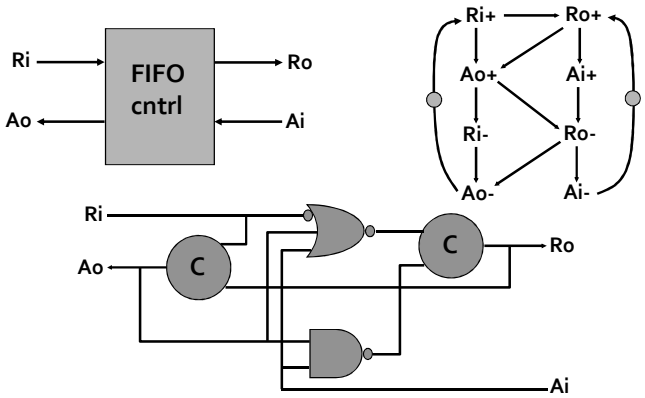
Control specification



Control specification

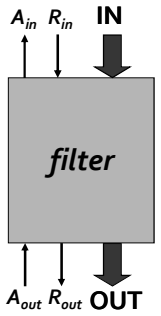


Control specification

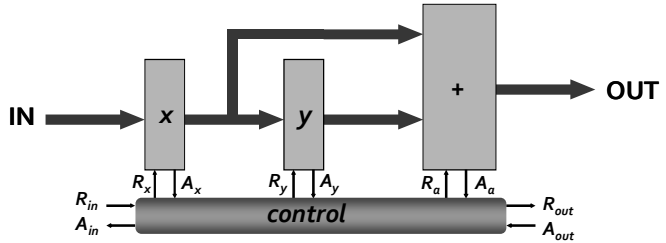


A simple filter: specification

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

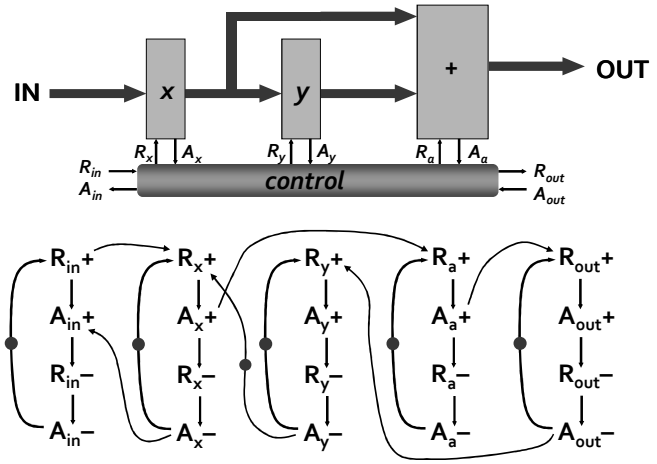


A simple filter: block diagram

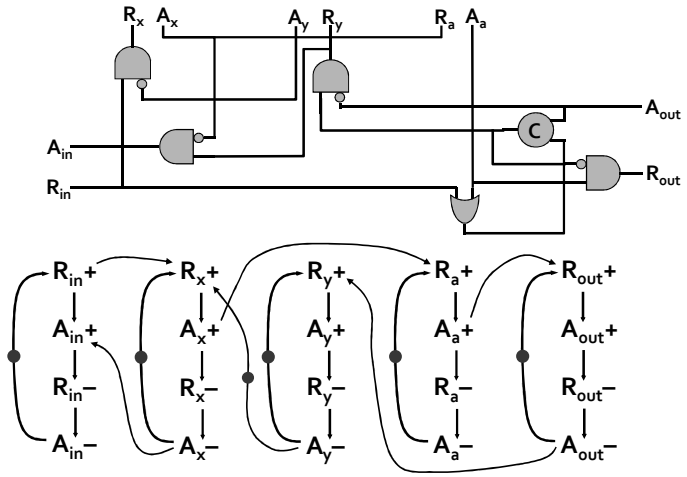


- *x* and *y* are level-sensitive latches (transparent when R=1)
- + is a bundled-data adder (matched delay between R_a and A_a)
- R_{in} indicates the validity of IN
- After A_{in}+ the environment is allowed to change IN
- (R_{out},A_{out}) control a level-sensitive latch at the output

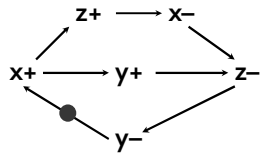
A simple filter: control spec.

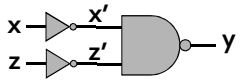


A simple filter: control impl.



Taking delays into account





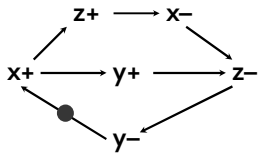
Delay assumptions:

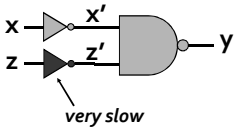
- Environment: 3 time units
- Gates: 1 time unit

events: $x+ \to x'- \to y+ \to z+ \to z'- \to x- \to x'+ \to z- \to z'+ \to y- \to$

time: 3 4 5 6 7 9 10 12 13 14

Taking delays into account





Delay assumptions: unbounded delays

events: $x+ \to x'- \to y+ \to z+ \to x- \to x'+ \to y-$ failure !

time: 3 4 5 6 9 10 11

WHY ASYNCHRONOUS ?

Motivation (designer's view)

- Modularity for system-on-chip design
  - Plug-and-play interconnectivity
- Average-case peformance
  - No worst-case delay synchronization
- Many interfaces are asynchronous
  - Buses, networks, …

Motivation (technology aspects)

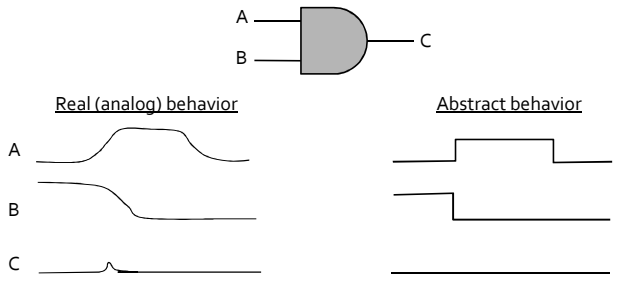
- Low power
  - Automatic clock gating
- Electromagnetic compatibility
  - No peak currents around clock edges
- Security
  - No 'electro–magnetic difference' between logical '0' and '1'in dual rail code
- Robustness
  - High immunity to technology and environment variations (temperature, power supply, …)

Dissuasion

- Concurrent models for specification
  - CSP, Petri nets, …: no more FSMs
- Difficult to design
  - Hazards, synchronization
- Complex timing analysis
  - Difficult to estimate performance
- Difficult to test
  - No way to stop the clock

SYNTHESIS OF ASYNCHRONOUS CIRCUITS FROM PETRI NETS

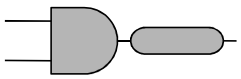
Delay models (I)



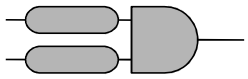
Abstractions are necessary to define delay models manageable for design, synthesis and verification. Abstractions introduce optimistic and pessimistic simplifications that must be carefully taken into account.

Delay models (II)

- Separation between functionality and timing [Muller]
  - Every gate has a zero-delay atomic evaluator (Boolean function)
  - A delay is associated to every output (gate delay model) or every input (wire delay model)
  - Delays can be:
    - Unbounded (arbitrary finite delays)
    - Bounded (within given min/max bounds)



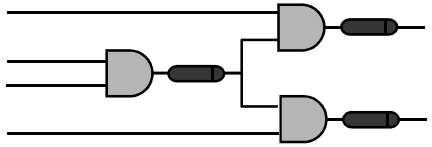
Gate delay model



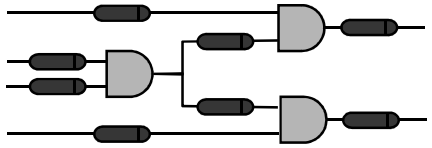
Wire delay model

Delay models (III)

- **Gate delay model: delays in gates, no delays in wires**



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

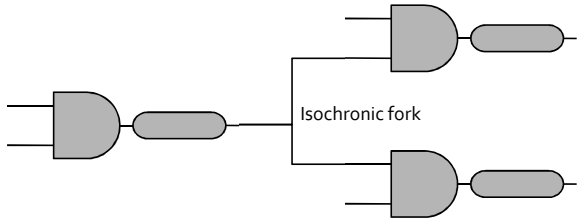


Delay models (IV)

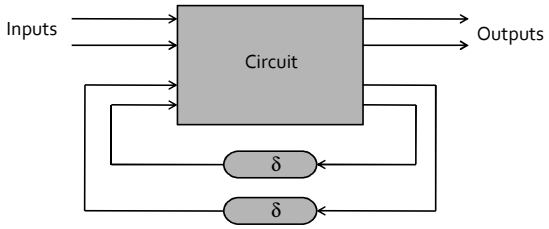
- ***Speed-independent circuit***:
  hazard-free under the unbounded gate delay model
- ***Delay-insensitive circuit***:
  hazard-free under the unbounded wire delay model
- ***Quasi-delay -insensitive circuit***:
  delay-insensitive with some isochronic forks

Speed-independent model

- Pessimistic, since delays are typically bounded
- Optimistic, since it assumes isochronic forks (negligible skew wrt the receiving gate delays)
- Efficient synthesis methods exist



Fundamental mode of operation



[Huffman 1964] : The circuit/environment interact with two phases
(1) The environment sends inputs to the circuit
(2) The circuit computes the outputs and the state signals

The environment does not send new inputs until the circuit stabilizes

Normal Fundamental Mode: Only one input changes at each communication cycle

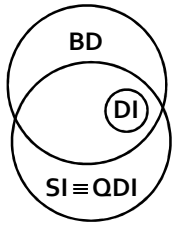
Input/Output mode of operation

- Computation and communication can overlap (according to some specified protocol)
- Event-based specification models are often used to describe the behavior (e.g., Petri nets).

This tutorial will cover the synthesis of speed-independent circuits that work under the I/O mode of operation and are specified using Petri nets.

Delay models for async. circuits

- Bounded delays (BD): realistic for gates and wires.
  - Technology mapping is easy, verification is difficult
- Speed independent (SI): Unbounded (pessimistic) delays for gates and "negligible" (optimistic) delays for wires.
  - Technology mapping is more difficult, verification is easy
- Delay insensitive (DI): Unbounded (pessimistic) delays for gates and wires.
  - DI class (built out of basic gates) is almost empty
- Quasi-delay insensitive (QDI): Delay insensitive except for critical wire forks (*isochronic forks*).
  - In practice it is the same as speed independent



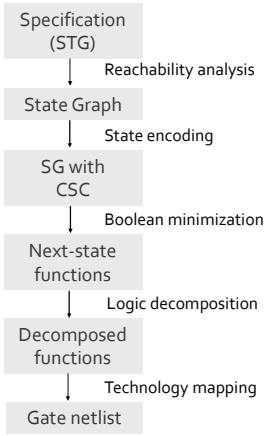
Outline

- Overview of the synthesis flow
- Specification
- State graph and next-state functions
- State encoding
- Implementability conditions
- Speed-independent circuit
  - Complex gates
  - C-element architecture
- Review of some advanced topics

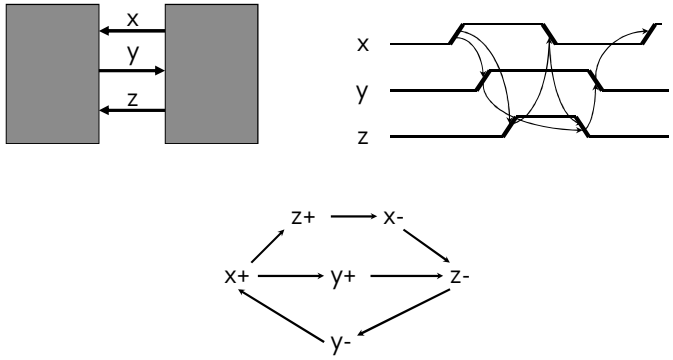
Book and synthesis tool

- J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno and A. Yakovlev, Logic synthesis for asynchronous controllers and interfaces, Springer-Verlag, 2002
- petrify: http://www.lsi.upc.es/petrify

Design flow

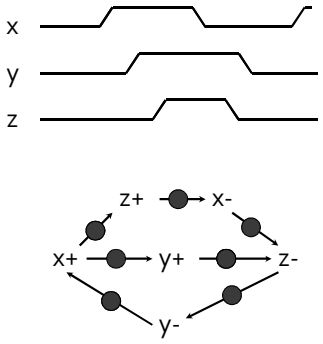


Specification

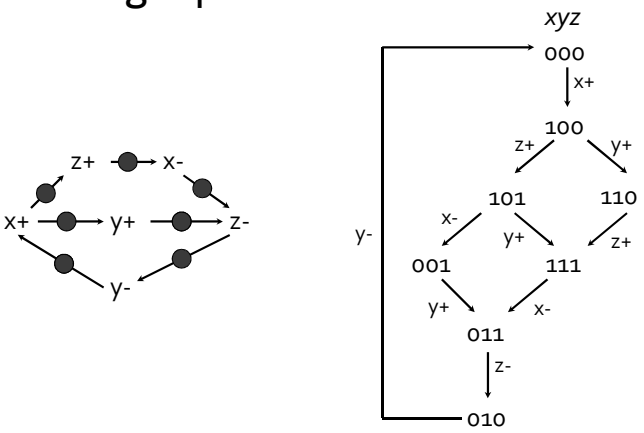


Signal Transition Graph (STG)

Token flow



State graph

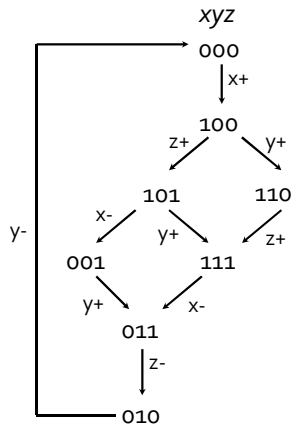


Next-state functions

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

$$y = z + x$$

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

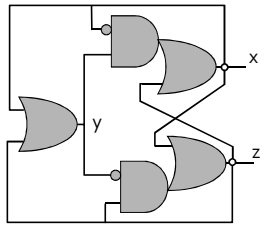


Gate netlist

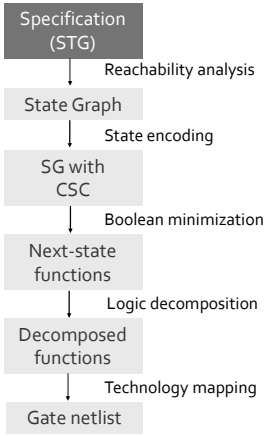
$$x = \overline{z} \cdot (x + \overline{y})$$

$$y = z + x$$

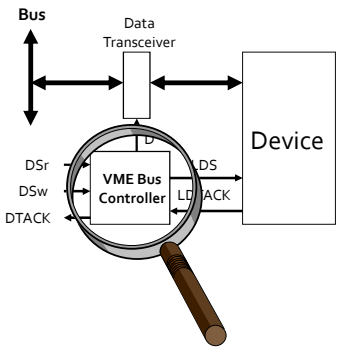
$$z = x + \overline{y} \cdot z$$

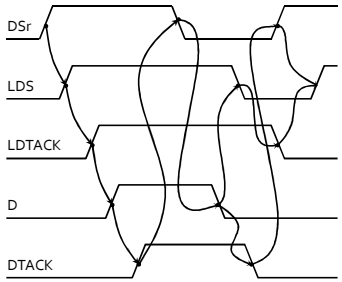


Design flow



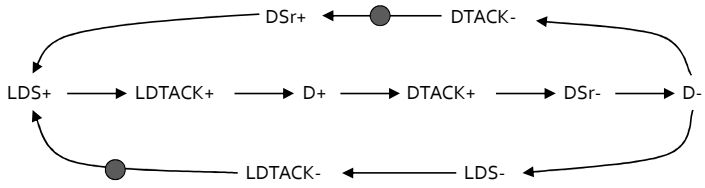
VME bus

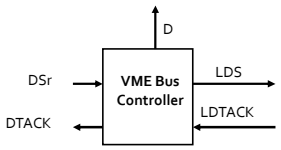




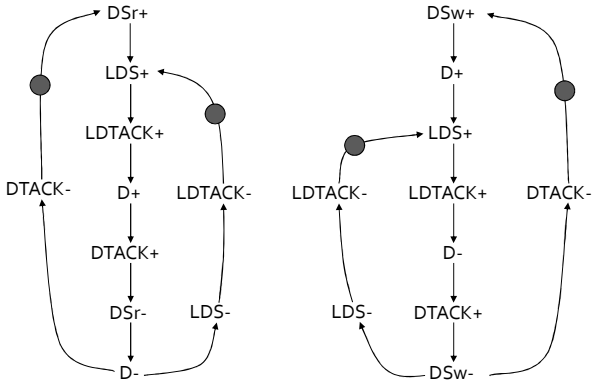
Read Cycle

STG for the READ cycle

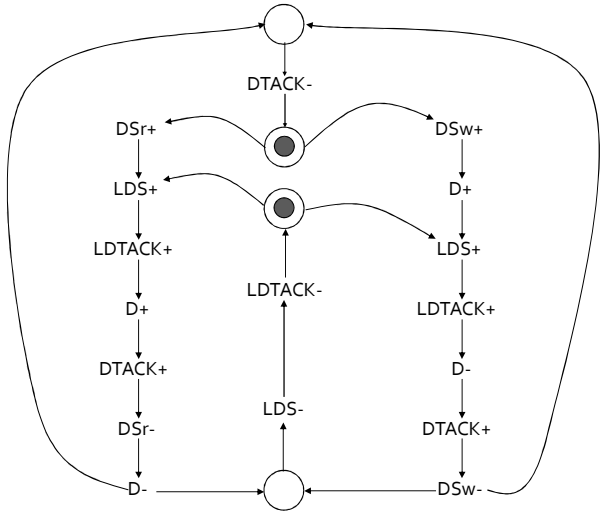




Choice: Read and Write cycles



Choice: Read and Write cycles



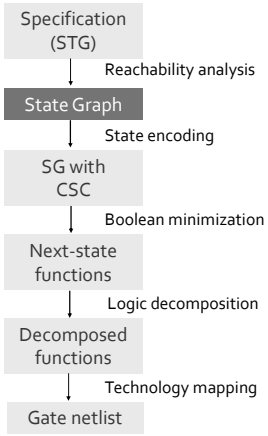
Circuit synthesis

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

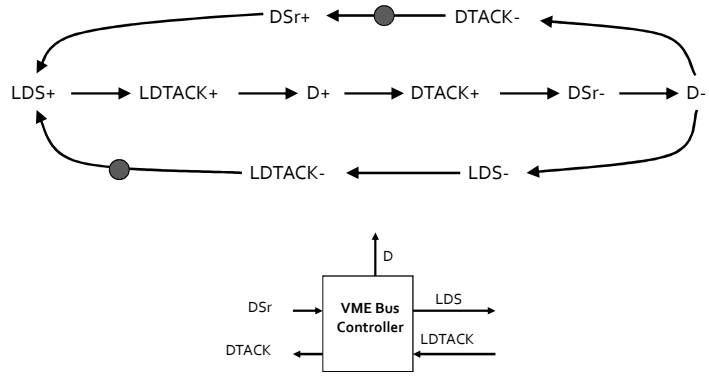
Speed independence

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

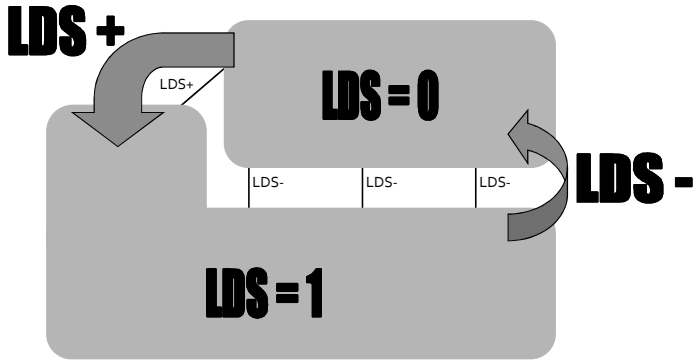
Design flow



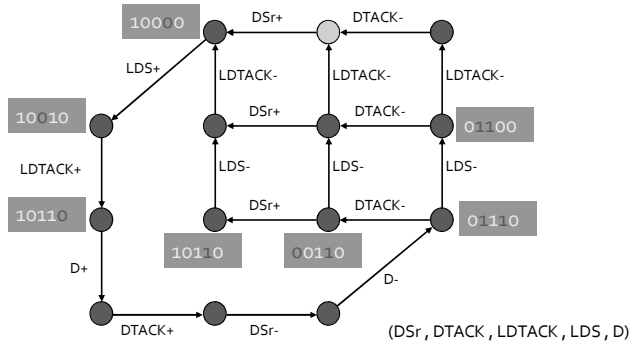
STG for the READ cycle



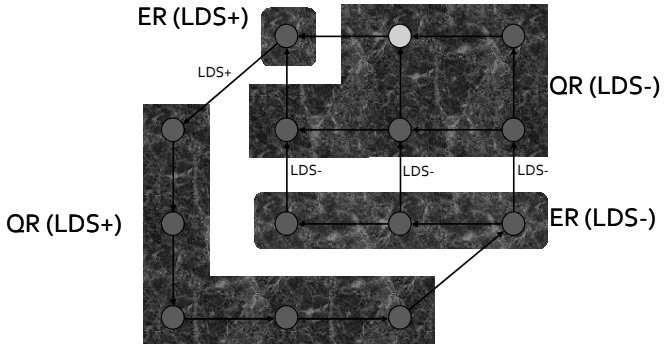
Binary encoding of signals



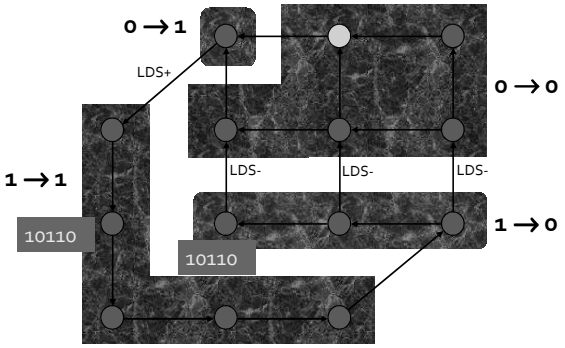
Binary encoding of signals



Excitation / Quiescent Regions



Next-state function



Karnaugh map for LDS

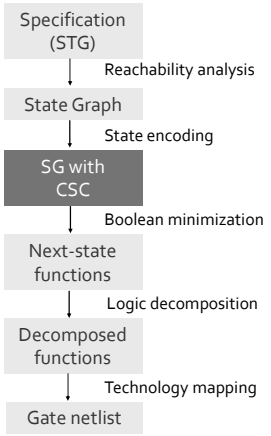
LDS = 0

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

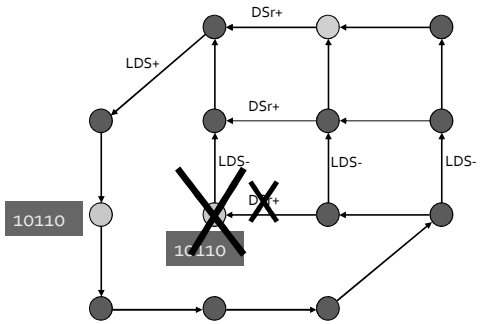
LDS = 1

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

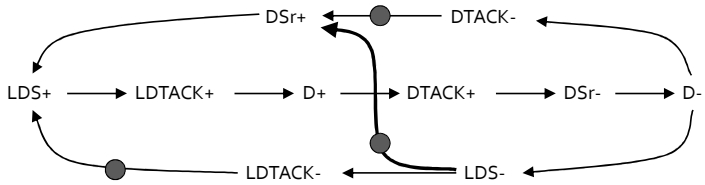
Design flow



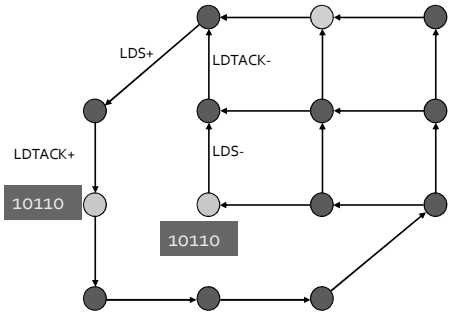
Concurrency reduction



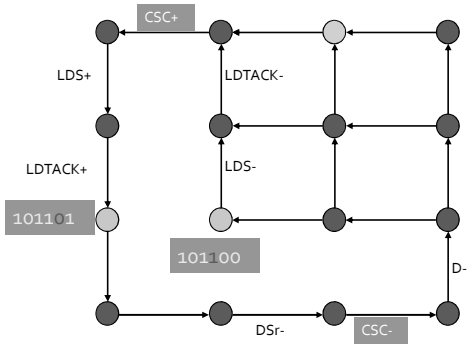
Concurrency reduction



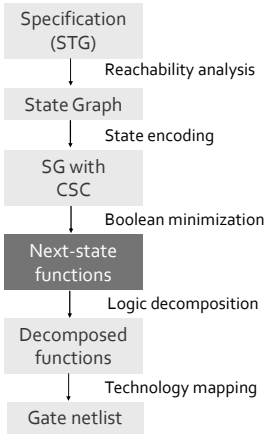
State encoding conflicts



Signal Insertion



Design flow



Complex-gate implementation

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

$$DTACK = D$$

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

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

Implementability 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)

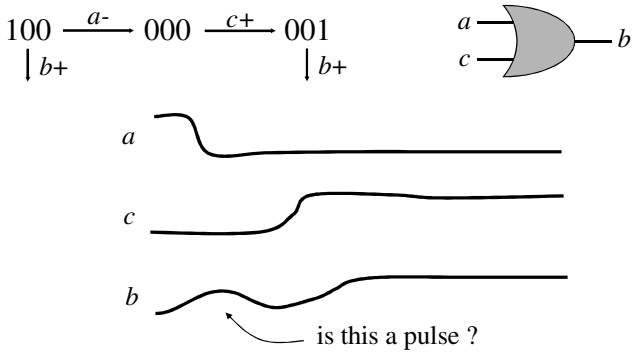
Implementability conditions

- Consistency + CSC + persistency

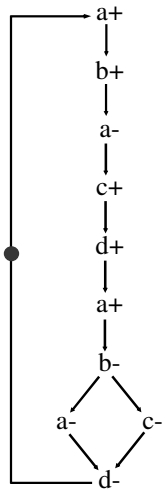
- There exists a speed-independent circuit that implements the behavior of the STG

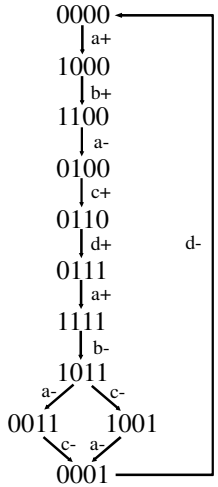
  (under the assumption that ay Boolean function can be implemented with one complex gate)

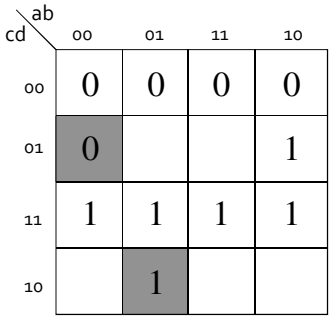
Persistency

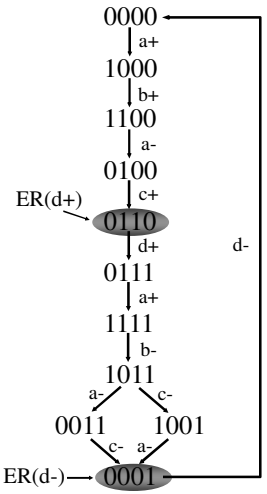


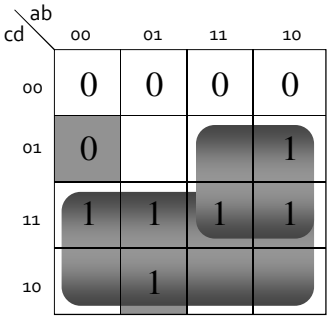
Speed independence ⇒ glitch-free output behavior under any delay





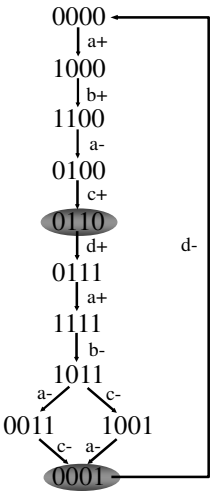




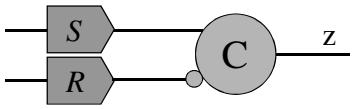


$$d = c + ad$$

Complex gate

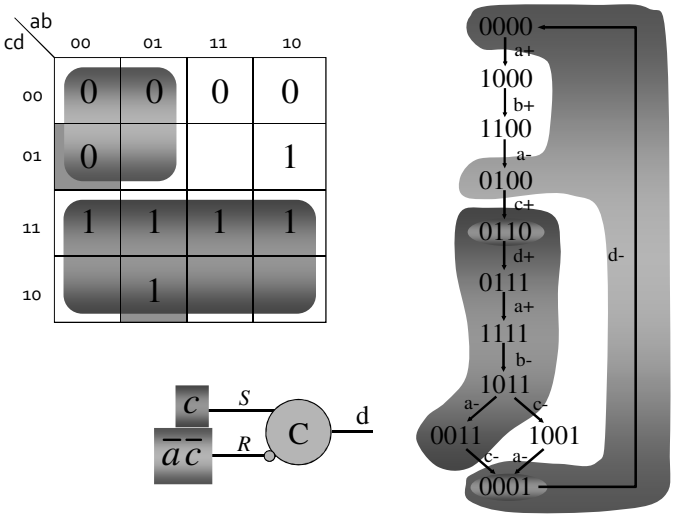


Implementation with C elements

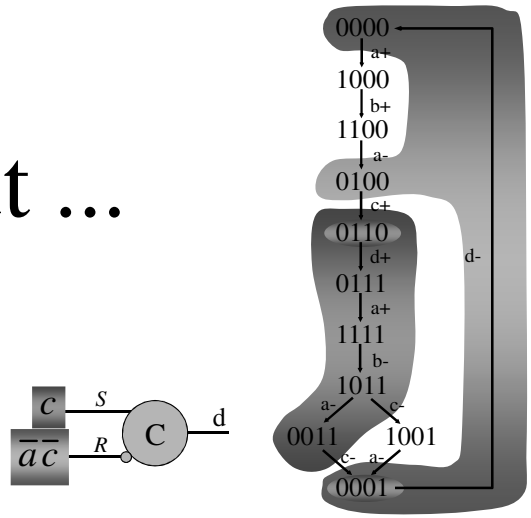


• • • → S+ → z+ → S- → R+ → z- → R- → • • •

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



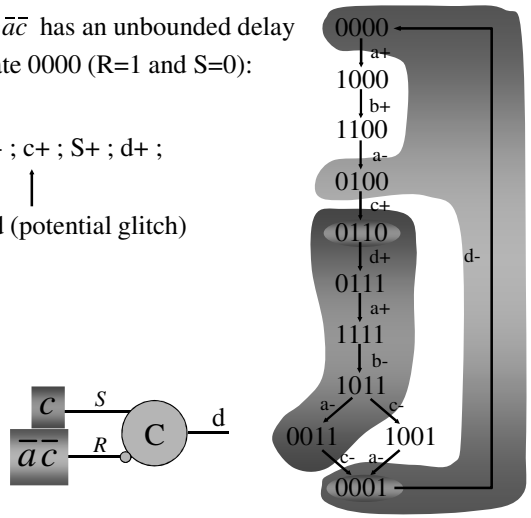
but …

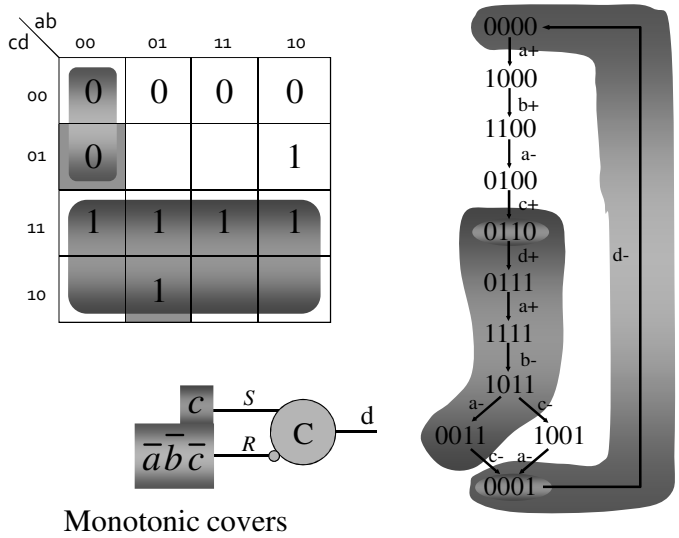


Assume that $R=\overline{a}\overline{c}$ has an unbounded delay
Starting from state 0000 (R=1 and S=0):

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

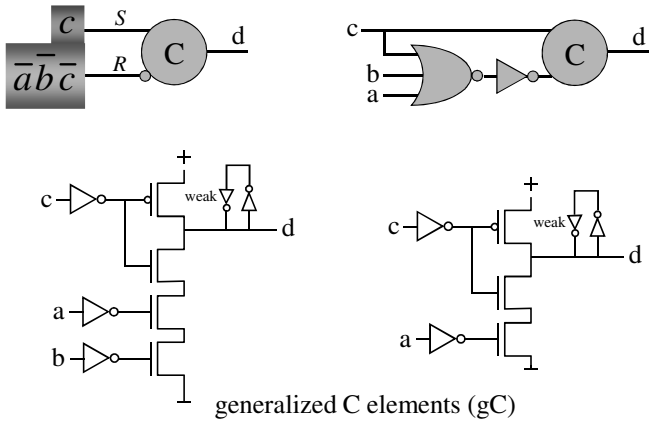
R+ disabled (potential glitch)





Monotonic covers

C-based implementations

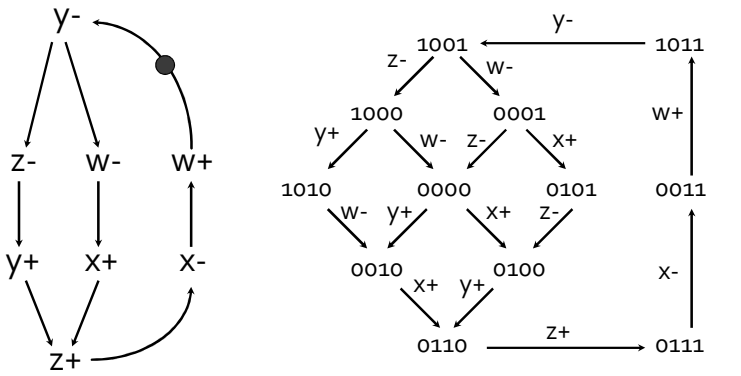


generalized C elements (gC)

Speed-independent implementations

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

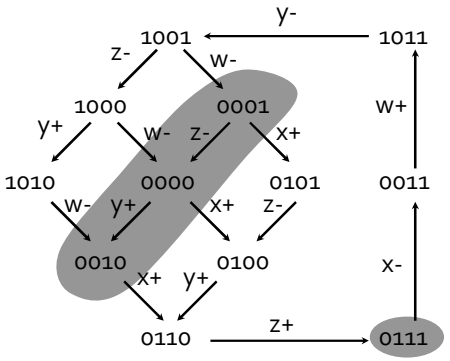
Synthesis exercise



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

Synthesis exercise

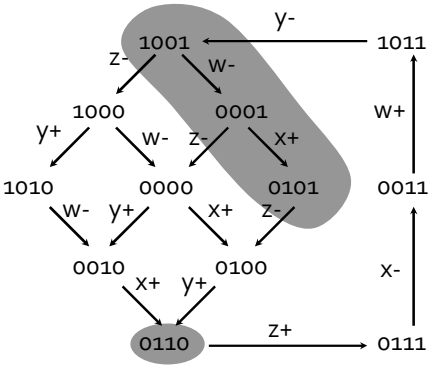
yz \ wx	00	01	11	10
00	1	1	-	0
01	1	1	-	0
11	0	0	-	0
10	1	1	-	0



Signal x

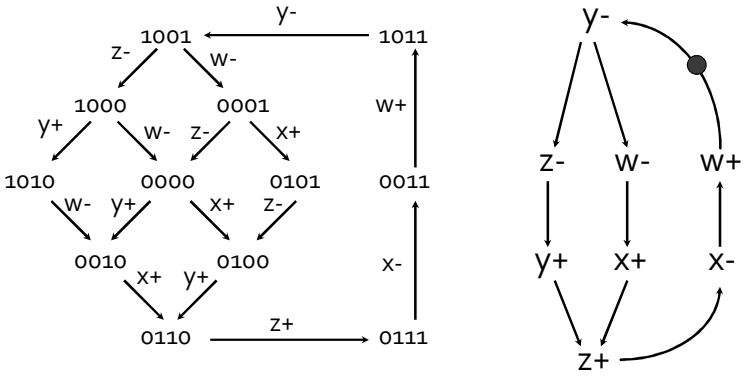
Synthesis exercise

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

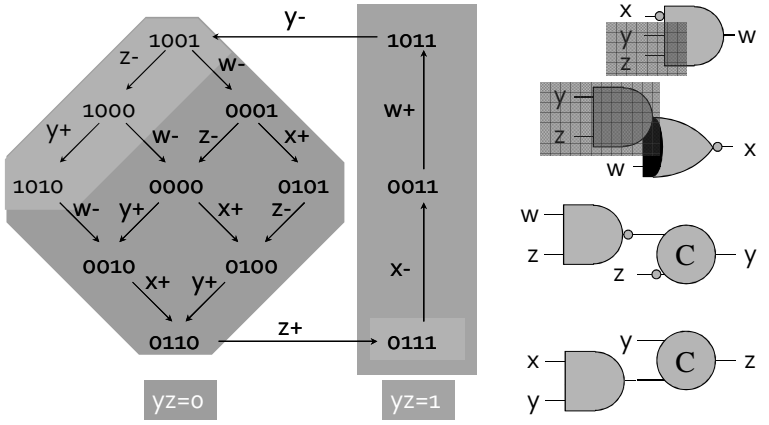


Signal z

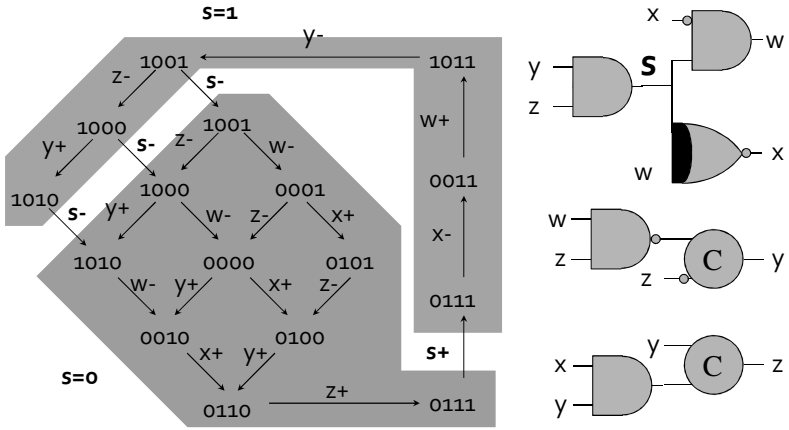
Logic decomposition: example



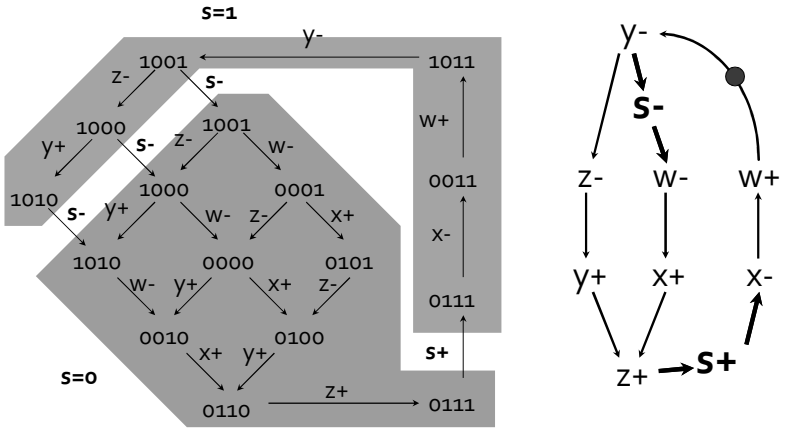
Logic decomposition: example



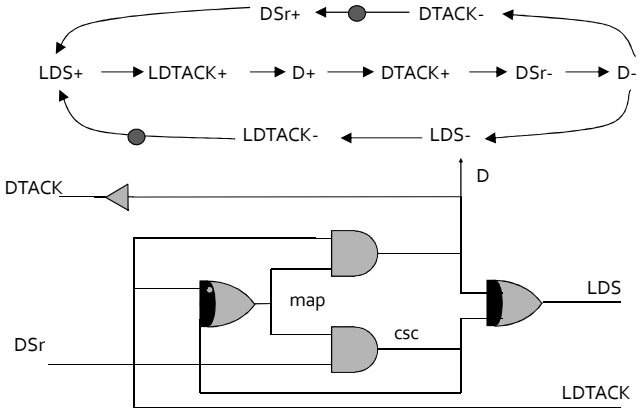
Logic decomposition: example



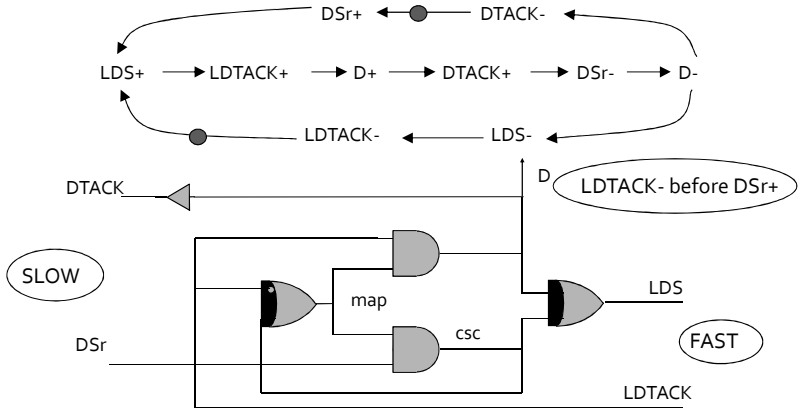
Logic decomposition: example



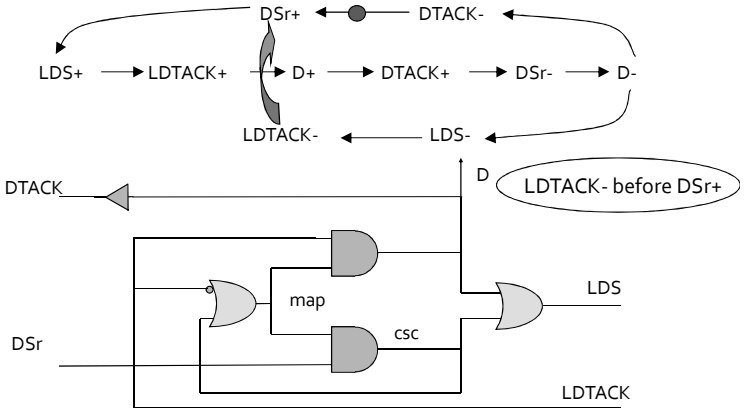
Speed-independent Netlist



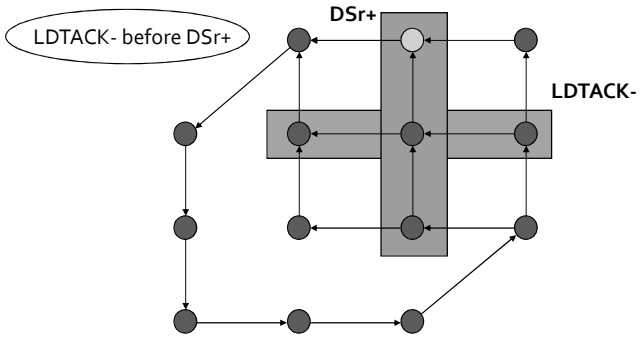
Adding timing assumptions



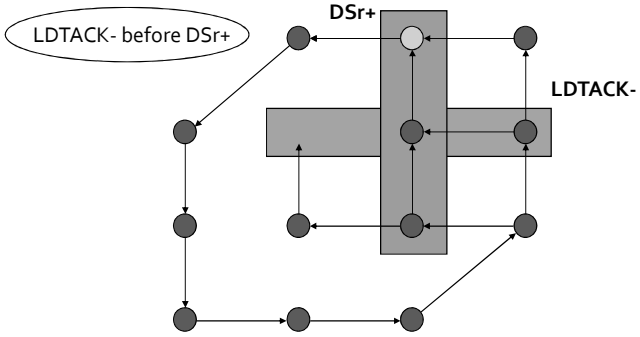
Adding timing assumptions



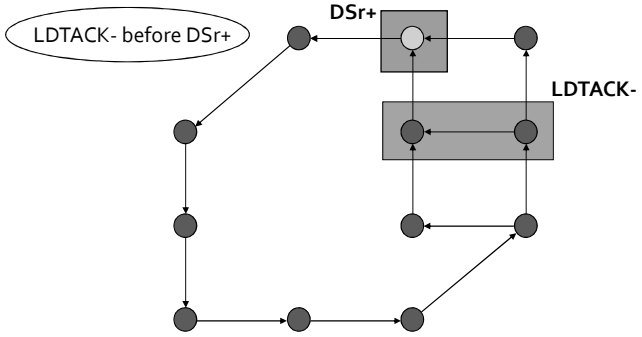
State space domain



State space domain

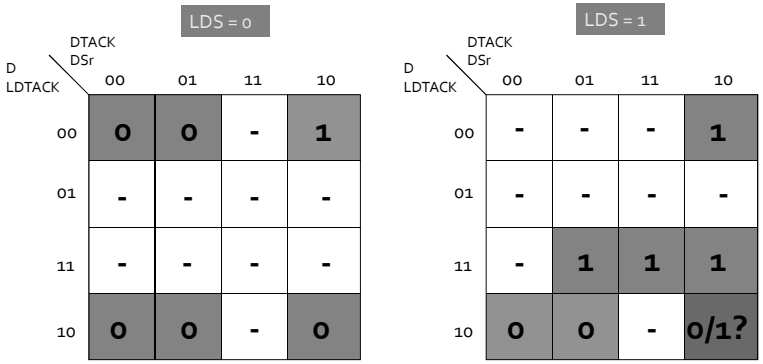


State space domain

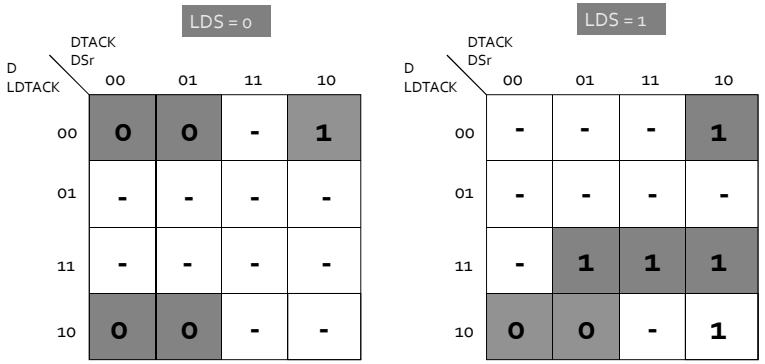


Two more unreachable states

Boolean domain



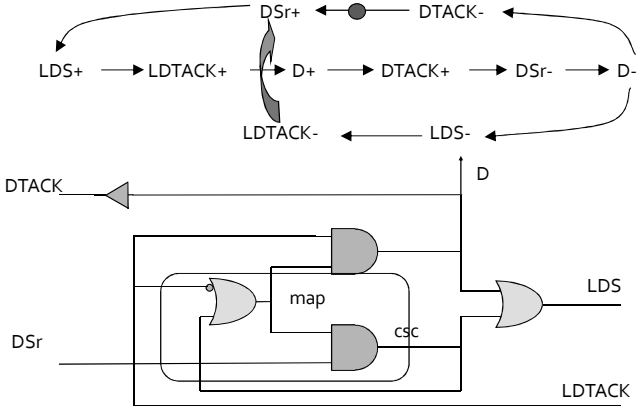
Boolean domain



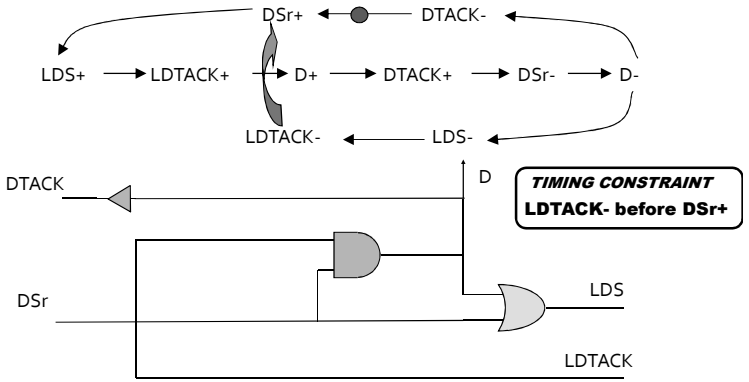
One more DC vector for *all* signals

One state conflict is removed

Netlist with one constraint



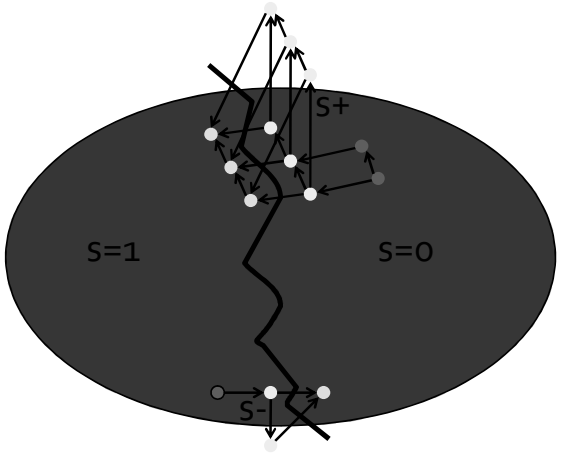
Netlist with one constraint



Signal insertion

- New signals need to be inserted to solve some synthesis problems (e.g., state encoding, logic decomposition)
- For each signal s, the events s+ and s- must be inserted while preserving certain behavioral properties (consistency, persistency).
- Each new signal determines a new partition of states (s=0, s=1)

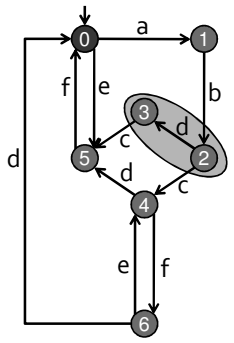
Signal insertion



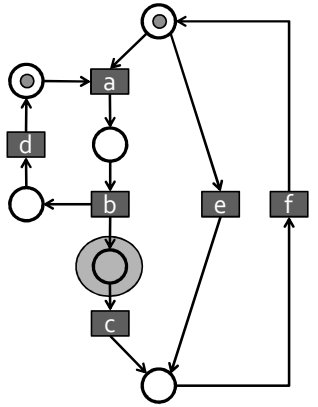
From state graphs to Petri nets

- A state graph may require transformations to meet certain properties (e.g., state encoding).
- The visualization of a state graph is not very informative. Event-based specifications explicitly represent the relations between events.
- Resort to the theory of regions

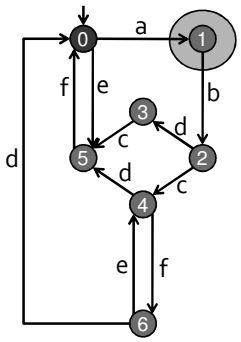
From state graphs to Petri nets

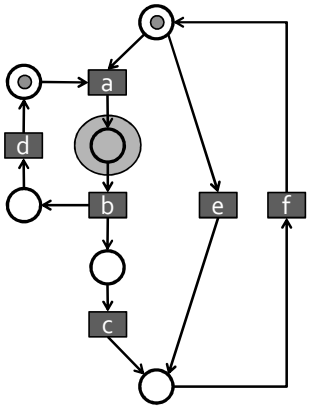


Region: {2,3}



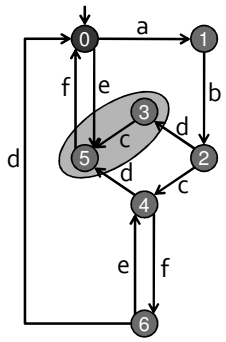
From state graphs to Petri nets

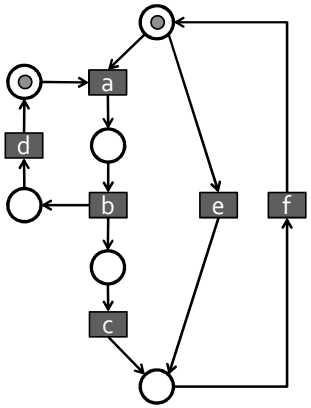




Region: {1}

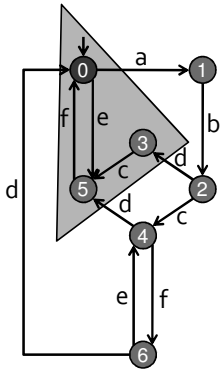
From state graphs to Petri nets

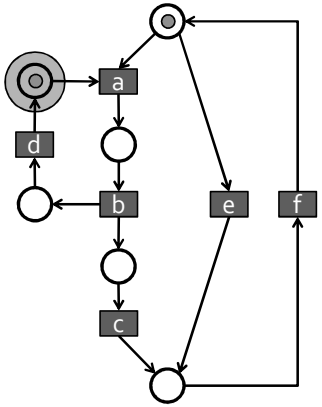




Not a region: {3,5}

From state graphs to Petri nets





Region: {0,3,5}

Theory of regions

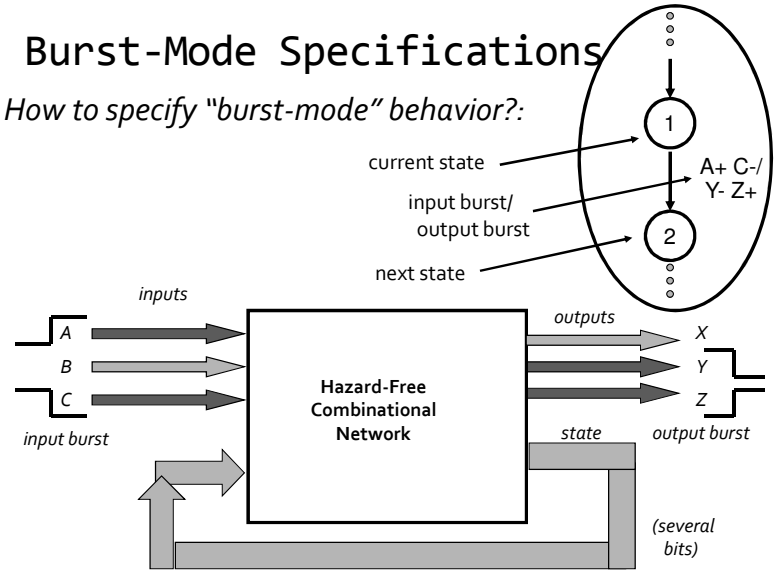
- Region: all arcs of any event have the same relationship with the region (enter, exit, no cross).
- Minimal region: not included in any other region
- Pre-/post-region of an event: region such that the event exits/enters the region
- Property: excitation closure
  - The intersection of all pre-regions of an event is the excitation region of the event

OTHER PARADIGMS

Thanks to Steve Nowick (Columbia Univ.)

Burst-Mode Specifications

How to specify "burst-mode" behavior?:



Burst-Mode Specifications

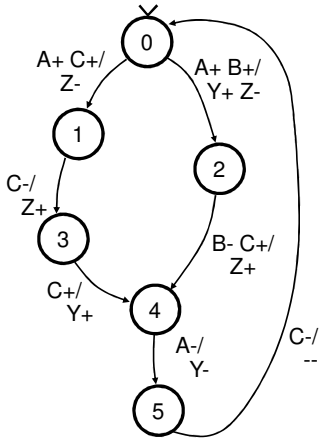
Example: Burst-Mode (BM) Specification:

Initial Values:
ABC = 000
YZ = 01

- Inputs in specified ***"input burst"*** can arrive <u>in any order</u> and <u>at any time</u>

- After all inputs arrive, generate ***"output burst"***

<u>Note:</u>
-input bursts: must be <u>non-empty</u>
(at least 1 input per burst)
-output bursts: may be empty
(0 or more outputs per burst)



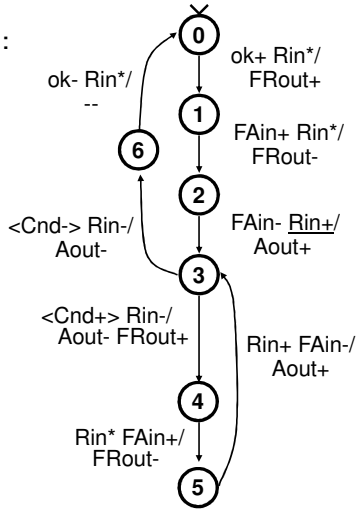
Burst-Mode Specifications

"Extended Burst-Mode" (XBM):
[Yun/Dill ICCAD-93/95]

<u>New Features:</u>

1. *"directed don't cares" (Rin*):*
allow concurrent inputs & outputs
2. *"conditionals" (<Cnd>):*
allow "sampling" of level signals

Handles glitchy inputs,
mixed sync/async inputs, etc.

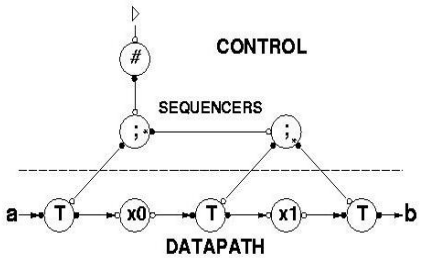


Syntax-directed translation

2-Place "Ripple Register" (= FIFO) [van Berkel]

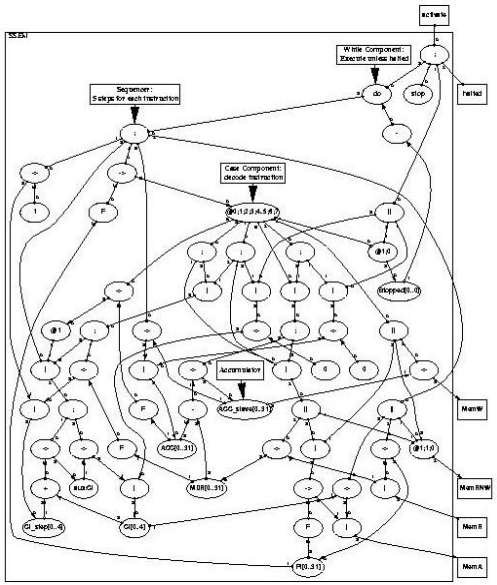
Tangram Program ⇒ Intermediate "Handshake Circuit"

```
proc (a?T  & b!T)
 begin
        x0, x1:  var T
 | forever do
        b! x1;
        x1 := x0;
        a? xo
 od
 end
```

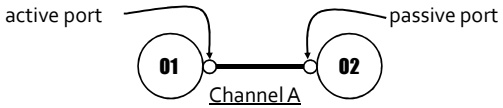


A Larger Example

Intermediate "Handshake Circuit"



Background: Channel-Based Communication



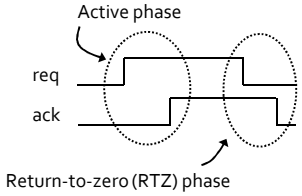
Components communicate using *"4-phase handshaking"*

- **O1:** *initiates* communication
- **O2:** *completes* communication

Channel implmtn. => use 2 wires:

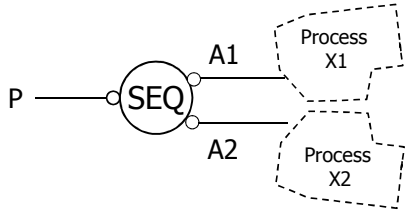
req => start operation
ack => operation done

(... can be extended to handle data)



Handshake Components: Sequencer

2-Way Sequencer: *activated* on channel P;
then *activates 2 processes in sequence* on channels A1 and A2

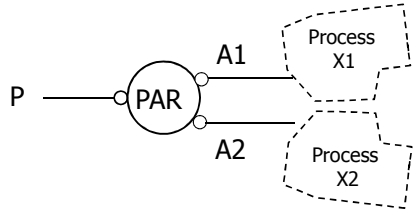


Operation
X1; X2

Goal: activate two sequential processes (i.e. operations)

Handshake Components: PAR Component

PAR Component: *activated* on channel P;
then *activates 2 processes in parallel* on channels A1 and A2

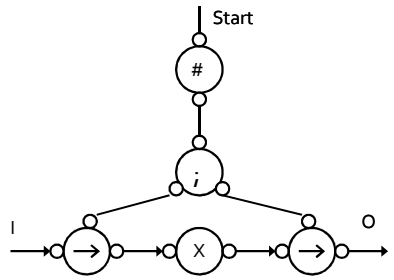


Operation
$X_1 \parallel X_2$

Goal: activate two parallel processes

Intermediate Representation

```
procedure Buf1 (
   input i: byte;
   output o: byte) is
local variable x : byte
begin
  loop begin
    i -> x ;
    o <- x
  end
end
```



Tangram Spec → Handshake Circuit (Syntax-Directed Translation; unoptimized)

Conclusions

- STGs have a high expressiveness power at a low level of granularity (similar to FSMs for synchronous systems)
- Synthesis from STGs can be fully automated
- Synthesis tools often suffer from the state explosion problem (symbolic techniques are used)
- The theory of logic synthesis from STGs can be found in:

J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno and A. Yakovlev, *Logic Synthesis of Asynchronous Controllers and Interfaces*, Springer Verlag, 2002.

Structural methods for synthesis of large specifications

Thanks to Josep Carmona and Victor Khomenko

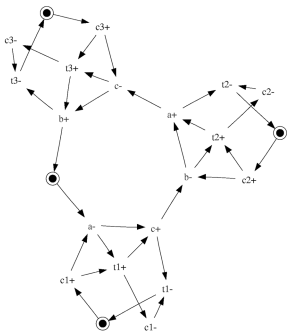
Outline

- Structural theory of Petri nets
  - Marking equation
  - Invariants
- ILP methods based on the marking equation
  - Detection of state encoding conflicts
  - Synthesis
- Methods based on unfoldings

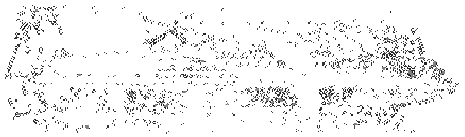
State space explosion problem

- Even in bounded nets, the state space can be exponential on the size of the net
  - Concurrency (explosion of interleavings)

Event-based vs. State-based model

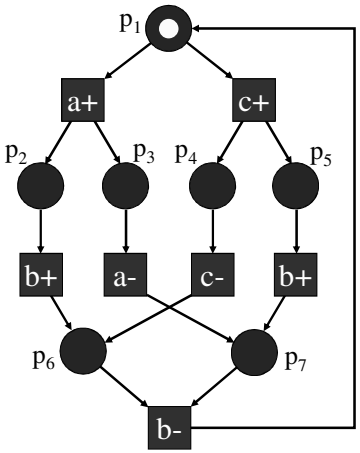


Petri Net



State Graph

Marking equation



Incidence matrix

	a+	a−	b+	b+	b−	c+	c−
p_1	−1	0	0	0	1	−1	0
p_2	1	0	−1	0	0	0	0
p_3	1	−1	0	0	0	0	0
p_4	0	0	0	0	0	1	−1
p_5	0	0	0	−1	0	1	0
p_6	0	0	1	0	−1	0	1
p_7	0	1	0	1	−1	0	0

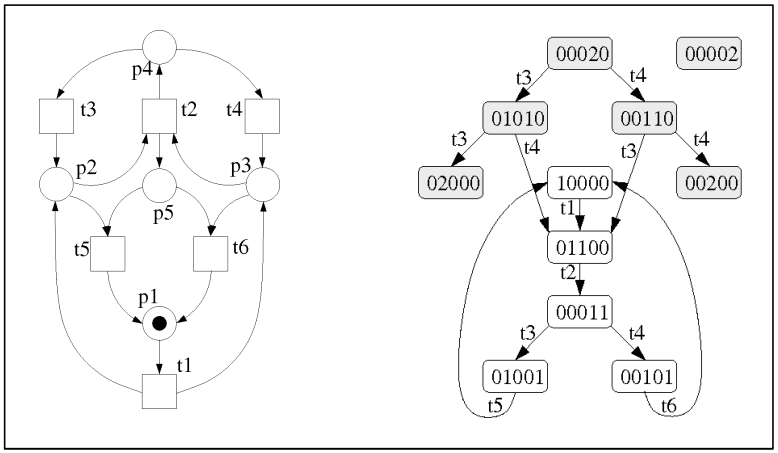
Marking equation

$$M' = M + Ax$$

$$\begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} + \begin{array}{c} \begin{matrix} a+ & a- & b+ & b+ & b- & c+ & c- \end{matrix} \\ \begin{pmatrix} -1 & 0 & 0 & 0 & 1 & -1 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & 1 & -1 & 0 & 0 \end{pmatrix} \end{array} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Necessary reachability condition, but not sufficient.

Spurious markings



Checking Unique State Coding

$$M_0 \xrightarrow{x} M_1 \xrightarrow{z = \{a+\ b+\ a-\ b-\}} M_2$$

- M_1 and M_2 have the same binary code (z must be a complementary set of transitions)
- M_1 and M_2 must be different markings (they must differ in at least one place)

Checking Unique State Coding

$$M_0 \xrightarrow{x} M_1 \xrightarrow{z = \{a+\ b+\ a-\ b-\}} M_2$$

ILP formulation:

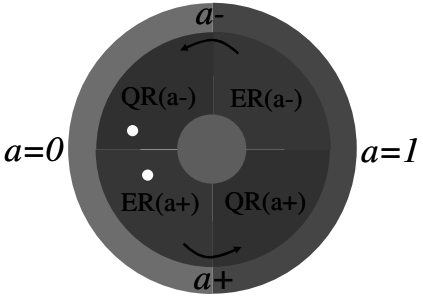
$M_1 = M_0 + Ax$
$M_2 = M_1 + Az$
$bal(z)$
$M_1 \neq M_2$
$x, z, M_1, M_2 \geq 0$

$bal(z) \equiv \forall a: \#(a+) - \#(a-) = 0$

Some experiments (USC)

benchmark	|P|	|T|	|signals|	CPU(s)
PpWk(3,9)	106	56	28	0.03
PpWk(3,12)	142	74	37	0.05
PpWkCsc(3,9)	108	56	28	0.67
PpWkCsc(3,12)	144	74	37	1.17
PpArb(3,9)	128	72	34	0.06
PpArb(3,12)	164	90	43	0.08
PpArbCsc(3,9)	131	72	34	1.05
PpArbCsc(3,12)	167	90	43	1.69

Checking Complete State Coding

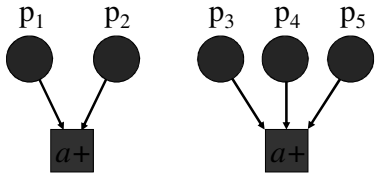


ILP formulation:

$M_1 = M_0 + Ax$
$M_2 = M_1 + Az$
$bal(z)$
$M_1 \in ER(a^*)$
$M_2 \notin ER(a^*)$
$x, z, M_1, M_2 \geq 0$

n ILP problems must be solved
(*n* is the number of transitions with label a^*)

Enabling conditions in ILP



$M \in ER(a+)$: $M(p_1)+M(p_2) \geq 2 \quad \vee \quad M(p_3)+M(p_4)+M(p_5) \geq 3$

$M \notin ER(a+)$: $M(p_1)+M(p_2) \leq 1 \quad \wedge \quad M(p_3)+M(p_4)+M(p_5) \leq 2$

(*formulation for safe nets only)

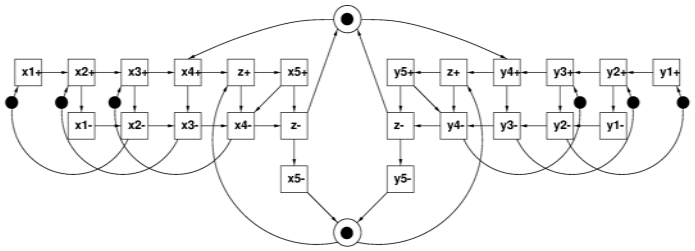
Some experiments (CSC)

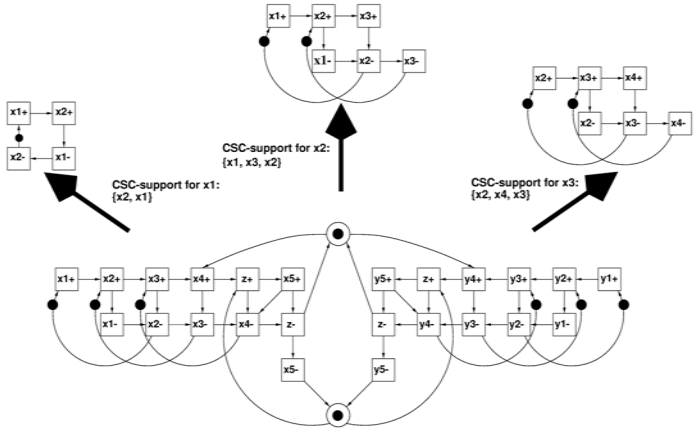
benchmark	|P|	|T|	|signals|	CLP	SAT	ILP
Tangram(3,2)	142	92	38	0.01	0.01	1.08
Tangram(4,3)	321	202	83	0.06	0.04	9.00
Art(10,9)	216	198	99	0.00	0.42	0.06
Art(20,9)	436	398	199	5.00	10.35	0.24
Art(30,9)	656	598	299	38.02	81.82	0.56
Art(40,9)	876	798	399	138.04	264.57	0.92
Art(50,9)	1096	998	499	377.00	630.41	1.46
ArtCsc(10,9)	752	630	315	time	14 m	3 m
ArtCsc(20,9)	1532	1270	635	time	mem	27 m
ArtCsc(30,9)	2312	1910	955	time	mem	1.5 h
ArtCsc(40,9)	3092	2550	1275	time	mem	3.5 h
ArtCsc(50,9)	3872	3190	1595	time	mem	7 h

Synthesis

- Each signal can be implemented with a subset of the STG signals in the support (typically less than 10)
- Synthesis of a signal:
  - Project the STG onto the support signals (hide the rest)
  - After projection, the STG still has CSC for the signal
  - Use state-based methods on each projection

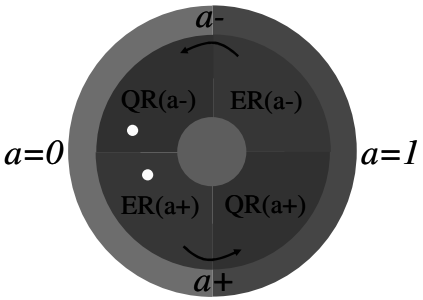
Synthesis example





Checking the support for a signal

Let $\sum$ be the set of signals and $\sum$' a potential support for a.
Let z' be the projection of z onto $\sum$'.
$\sum$' is a valid support for a if the following model has no solution:



ILP formulation:

$$M_1 = M_0 + Ax$$
$$M_2 = M_1 + Az$$
$$bal(z')$$
$$M_1 \in ER(a^*)$$
$$M_2 \notin ER(a^*)$$
$$x, z, M_1, M_2 \geq 0$$

Algorithm to find the support

z′ := {a} ∪ {trigger signals of a};

forever

    z″ := ILP_check_support (STG, a, z′);

    if z″ = 0 then return z′;

    z′ := z′ ∪ {unbalanced signals in z″};

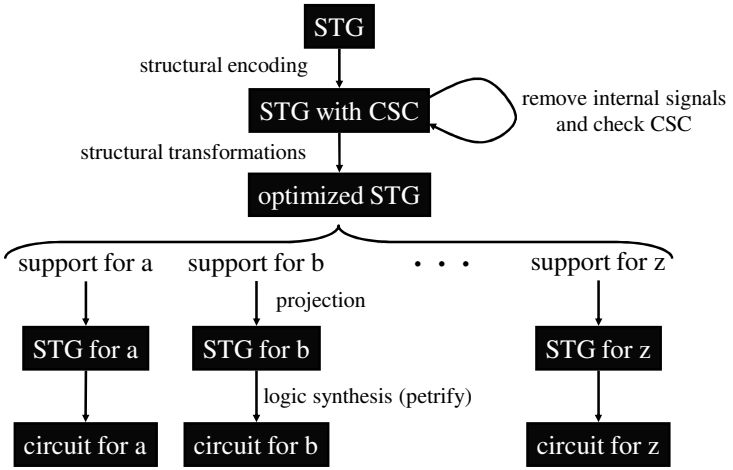
end forever

Experiments (Support + Synthesis)

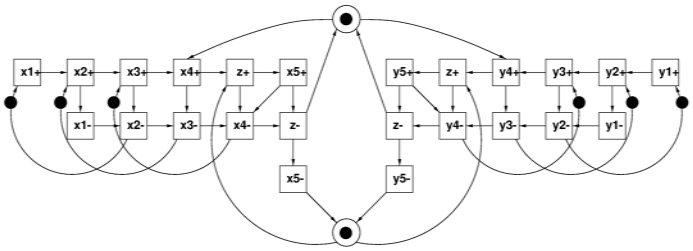
benchmark	States	|P|	|T|	|signals|	Literals		CPU	
					Petrify	ILP	Petrify	ILP
PpWkCsc(2,6)	8192	47	26	19	57	57	5	1
PpWkCsc(2,9)	524288	71	38	19	87	87	49	2
PpWkCsc(3,9)	2.7 x 10E7	106	56	28	?	130	mem	3
PpWkCsc(3,12)	2.2 x 10E11	142	74	37	?	117	time	3
PpArbCsc(2,6)	61440	62	36	17	77	77	21	83
PpArbCsc(2,9)	3.9 x 10E6	110	60	29	107	107	185	59
PpArbCsc(3,9)	3.3 x 10E9	131	72	34	163	165	10336	289
PpArbCsc(3,12)	1.7 x 10E12	167	90	43	?	210	time	608
TangramCsc(3,2)	426	142	92	38	97	103	56	146
TangramCsc(4,3)	9258	321	202	83	?	247	mem	2 h

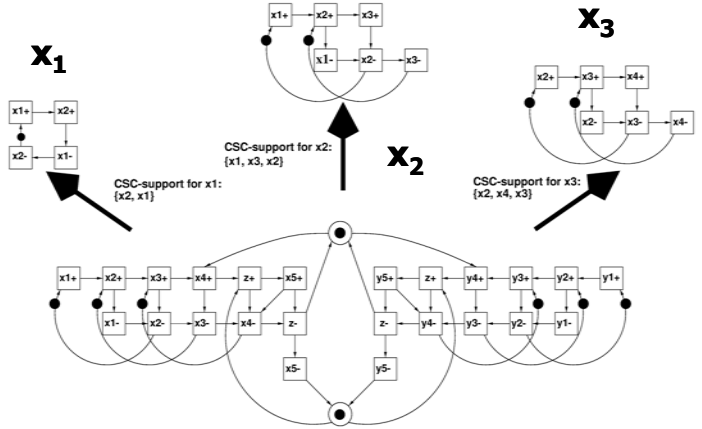
- Petrify (*Cortadella et al*):
  - State-based & technology mapping
  - BDD

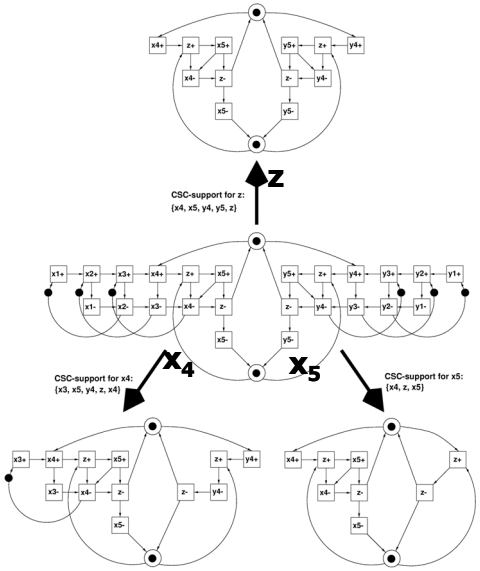
Design flow

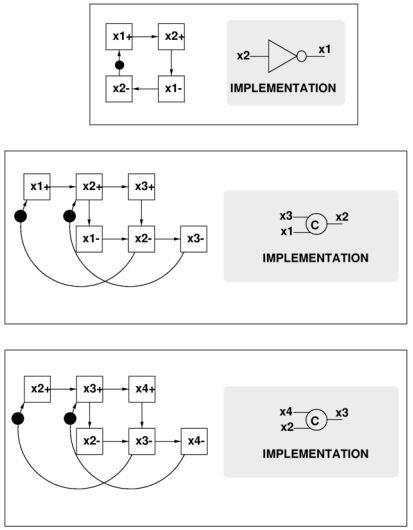


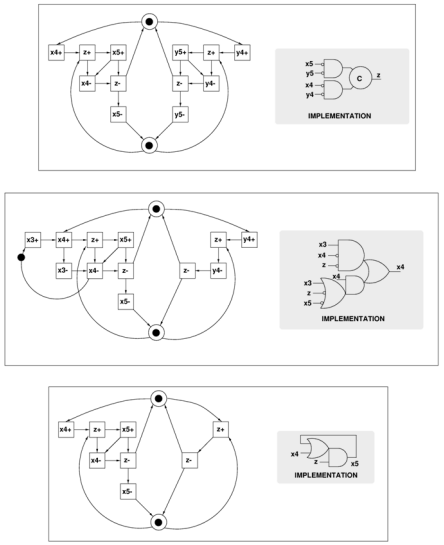
Synthesis example





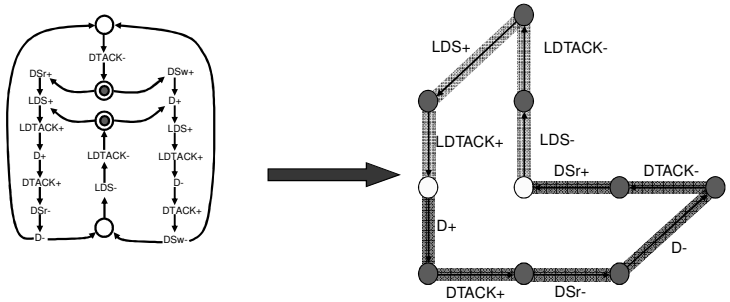






STATE ENCODING

Detection of conflicting states



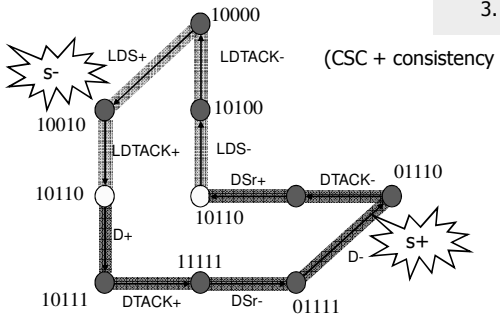
- **ILP**
  [Carmona & Cortadella, ICCAD'03]
- **SAT-UNFOLD**
  [Khomenko *et al.*, Fund. Informaticae]

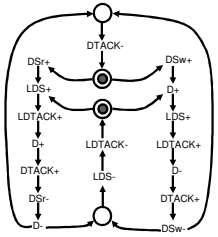
Disambiguation by consistent signal insertion

Disambiguate the conflicting states by introducing a new signal s:

STG Insertion of signal s must:
1. Solve conflict
2. Preserve consistency
3. Preserve persistency

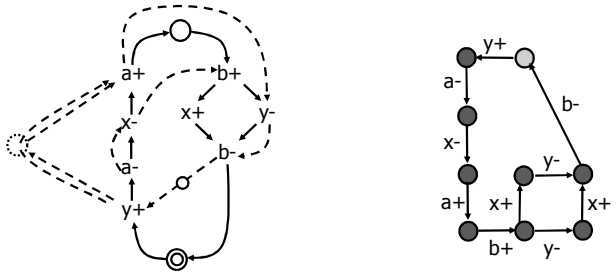
(CSC + consistency + persistency = SI-circuit)





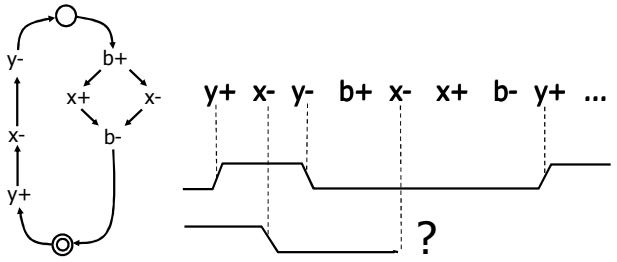
Implicit place

DEF1 (Behavior): The behavior of the net does not depend on the place.
DEF2 (Petri net): it never disables the firing of a transition.

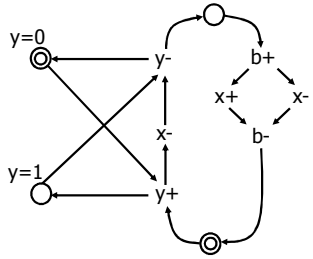


Consistency

Consecutive firings of a signal must alternate



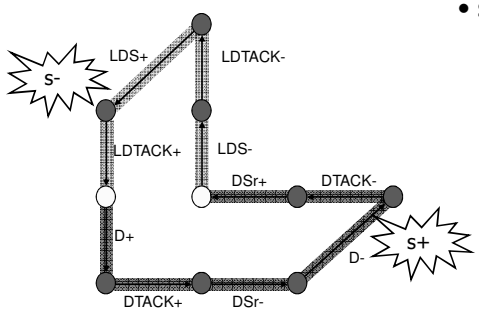
Implicit Places & Consistency



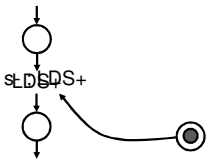
Theorem (Colom *et al.*)
Places y=0 and y=1 are implicit
if and only if signal y is consistent

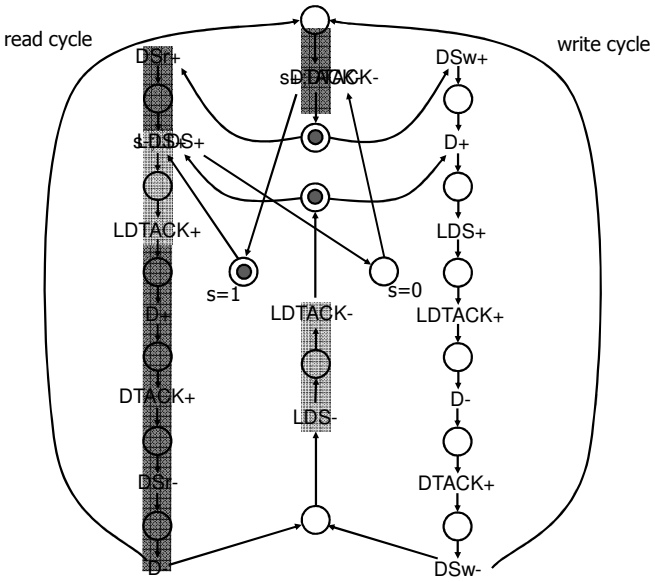
Disambiguation by consistent signal insertion

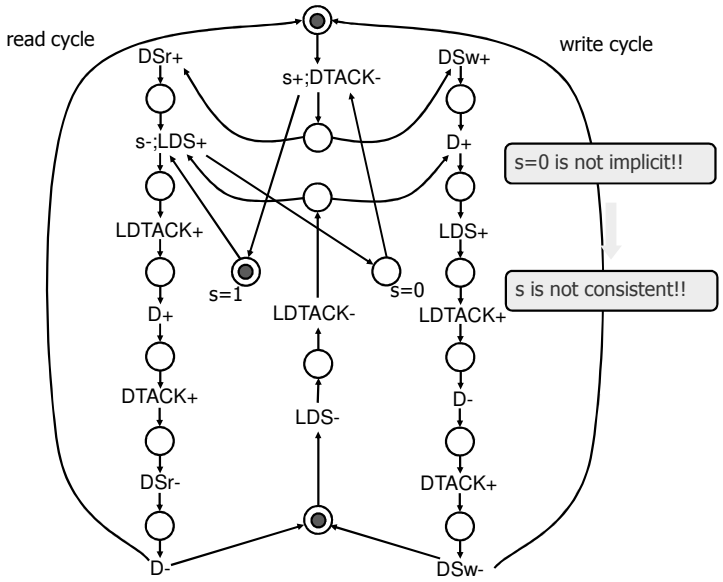
Disambiguate the conflicting states by introducing a new signal s:

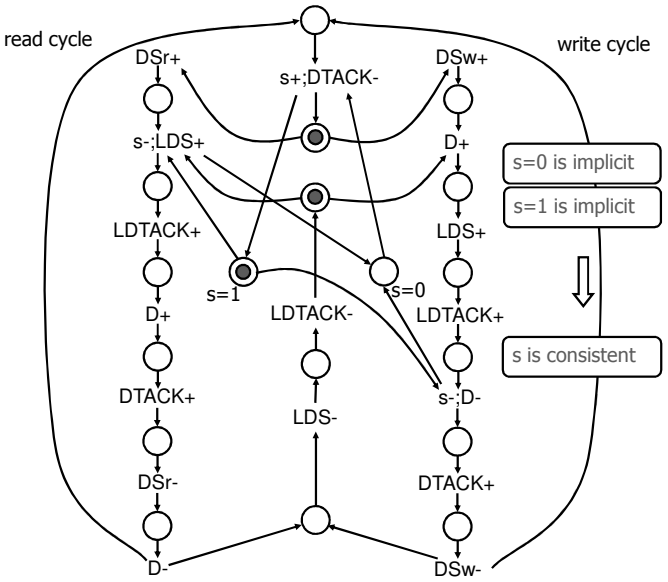


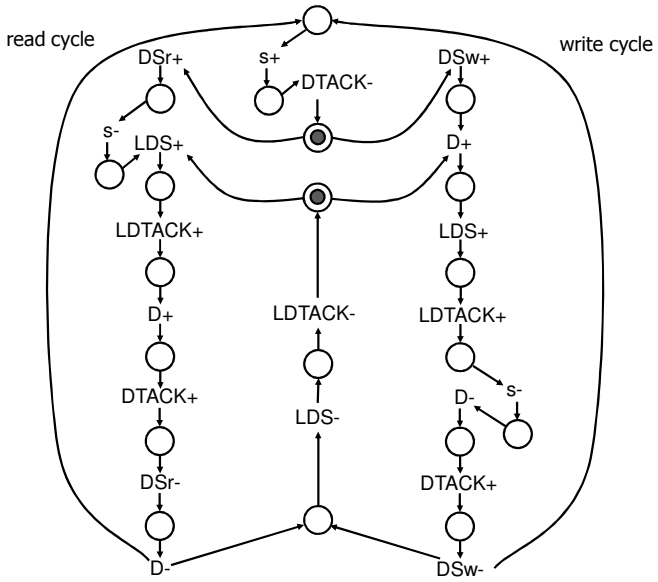
Insertion of s into the STG:
- s- will precede LDS+
- s+ will precede DTACK-




read cycle
write cycle
DSr+
DSw+
D+
LDTACK+
LDS+
s=1
s=0
LDTACK-
D+
DTACK+
LDS-
D-
DSr-
DTACK+
D-
DSw-


read cycle
write cycle
DSr+
s+;DTACK-
DSw+
s-;LDS+
D+
LDTACK+
LDS+
s=1
s=0
LDTACK-
D+
DTACK+
LDS-
D-
DSr-
DTACK+
D-
DSw-
s=0 is not implicit!!
s is not consistent!!


read cycle
write cycle
DSr+
s+;DTACK-
DSw+
s-;LDS+
D+
LDTACK+
LDS+
s=1
s=0
LDTACK-
D+
DTACK+
LDS-
s-;D-
DSr-
DTACK+
D-
DSw-
s=0 is implicit
s=1 is implicit
s is consistent


read cycle
write cycle
DSr+
s+
DTACK-
DSw+
s-
LDS+
D+
LDTACK+
LDS+
D+
LDTACK-
LDTACK+
DTACK+
LDS-
D-
s-
DSr-
DTACK+
D-
DSw-

Main algorithm for solving CSC conflicts

```
while CSC conflits exist do
  (σ1, σ2) := Find traces connecting conflict
  (s=0, s=1) := Find implicit places
                to break conflict
  Insert s+/s- transitions connected to
   (s=0) or (s=1)
endwhile
```

State space explosion problem

- Goal: avoid state enumeration to check implicitness of a place.
- Classical methods to avoid the explicit state space enumeration:
  - Linear Algebra (LP/MILP) } Structural methods
  - Graph Theory
  - Symbolic representation (BDDs)
  - Partiar order (Unfoldings)

LP model to check place implicitness

A place p is implicit if the following LP model is infeasible, where P' = P – {p}:

LP formulation:

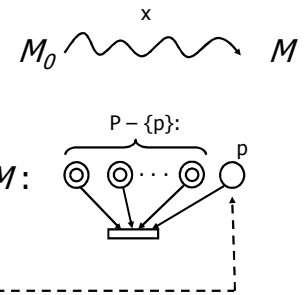
$M_0 + Ax = M$

$M[P'] - F[P',p\bullet]\cdot s \geq \mathbf{0}$

$M[p] - F[p,p\bullet]\cdot s < \mathbf{0}$

$s\cdot\mathbf{1} = 1$

$x, M, s \geq 0$



[Silva *et al.*]

LP model to check place implicitness

A place p is implicit if the following LP model is infeasible, where P' = P – {p}:

LP formulation:

$M_0 + Ax = M$

$M[P'] - F[P',p\bullet]\cdot s \geq \mathbf{0}$

$M[p] - F[p,p\bullet]\cdot s < \mathbf{0}$

$s\cdot\mathbf{1} = 1$

$x, M, s \geq 0$

DUAL

A place p is implicit if $M_0[p]$ is greater than or equal to the optimal value of the following LP, where P' = P – {p}:

LP formulation:

$\min y\cdot M_0$

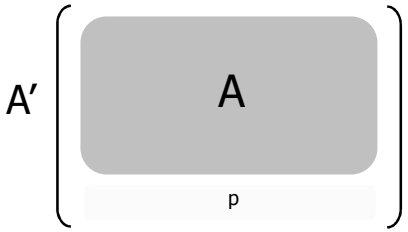
$y\cdot A[P'.T] \leq A[p,T]$

$y\cdot F[P', p\bullet] \geq F[p, p\bullet]$

$y \geq 0$

[Silva *et al.*]

MILP model to insert a implicit place



MILP formulation:

min y· M_0

y·A'[P'.T] ≤ A'[p,T]

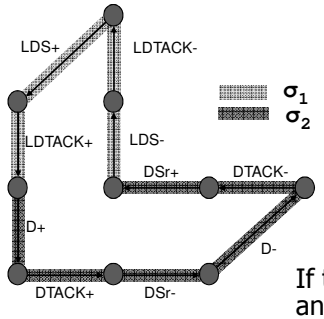
y· F'[P', p•] ≥ F[p, p•]

y≥ 0

$p \in \{-1,0,1\}^n$

MILP variables: y, **p**

MILP model to find insertion points that disambiguate the conflict



MILP formulation:

MILP "s=0 implicit"

MILP "s=1 implicit"

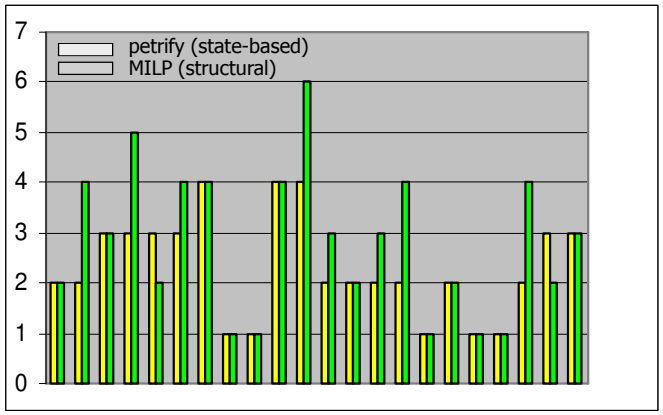
$\#(\sigma_1,s+) = \#(\sigma_1,s-) + 1$

$\#(\sigma_2,s-) = \#(\sigma_2,s+) + 1$

$M_0[s=0] + M_0[s=1] = 1$

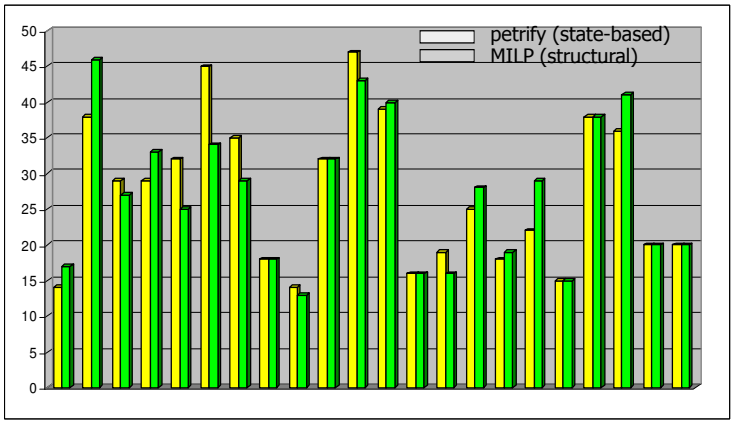
If there is a solution, rows in A' for s=0 and s=1 describe the insertion points (arcs in the net)

Number of inserted encoding signals



Benchmarks from [Cortadella *et al.*, IEEE TCAD'97]

Number of literals (area)



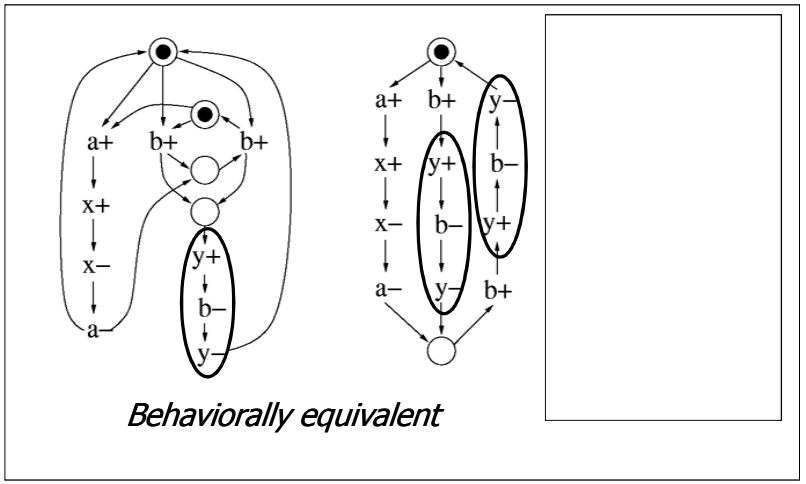
Benchmarks from [Cortadella *et al.*, IEEE TCAD'97]

Experimental results: large controllers

example	Places	Trans	Signals	CPU(min)	#sig	Lits	HDL
Art(10,9)	216	198	99	3.6	28	305	--
Art(20,9)	436	398	199	73.0	57	629	--
PpWk(3,12)	142	74	37	1.0	3	190	--
PpArb(3,12)	164	90	43	11.5	2	206	--
Var(9,5)	302	338	150	6.4	24	613	--
Var(12,1)	368	394	183	12.7	27	445	--
Par(12)	63	52	52	0.2	12	101	253
SeqPar(21,10)	160	128	64	2.2	23	269	398
SPM(7,16,18)	192	394	60	11.8	17	237	640

Synthesis with structural methods from
[Carmona & Cortadella, ICCAD'03]

It doesn't always work ...

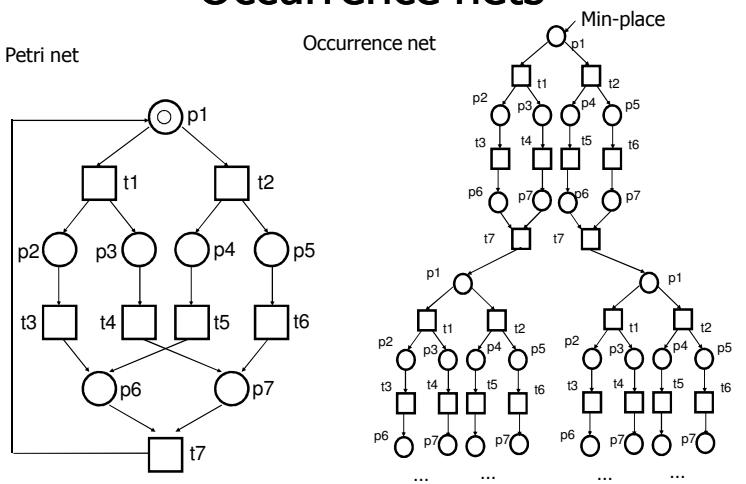


Behaviorally equivalent

SYNTHESIS USING UNFOLDINGS

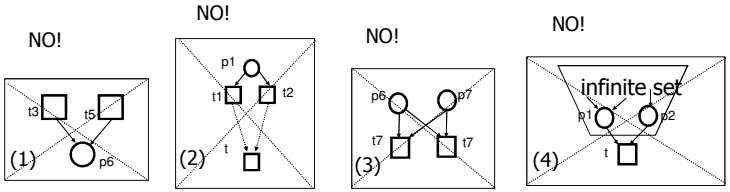
Work by Khomenko, Koutny and Yakovlev

Occurrence nets

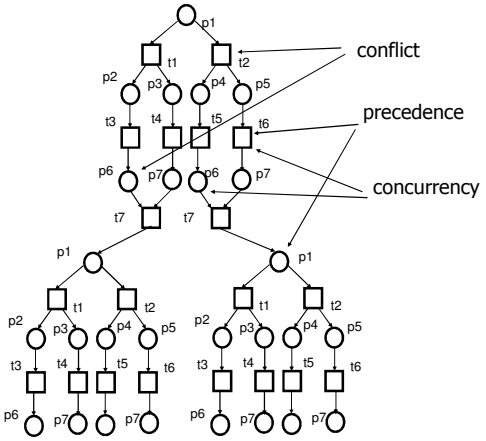


Occurrence nets

- The occurrence net of a PN N is a labelled (with names of the places and transitions of N) net (possibly infinite!) which is:
  - Acyclic
  - Contains no backward conflicts (1)
  - No transition is in self-conflict (2)
  - No twin transitions (3)
  - Finitely preceded (4)

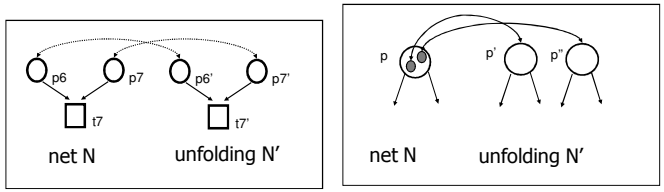


Relations in occurrence nets

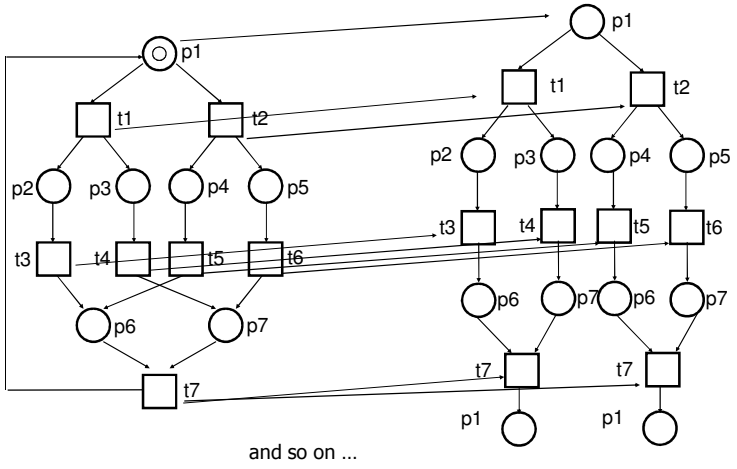


Unfolding of a PN

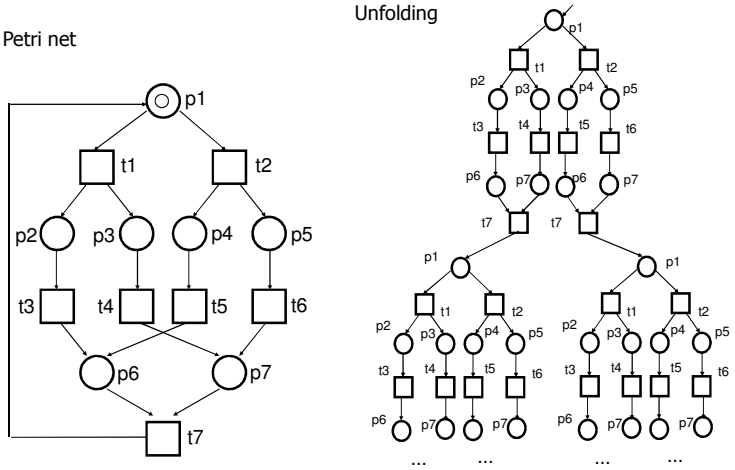
- The unfolding of Petri net N is a maximal labelled occurrence net (up to isomorphism) that preserves:
  - one-to-one correspondence (bijection) between the predecessors and successors of transitions with those in the original net
  - bijection between min places and the initial marking elements (which is multi-set)



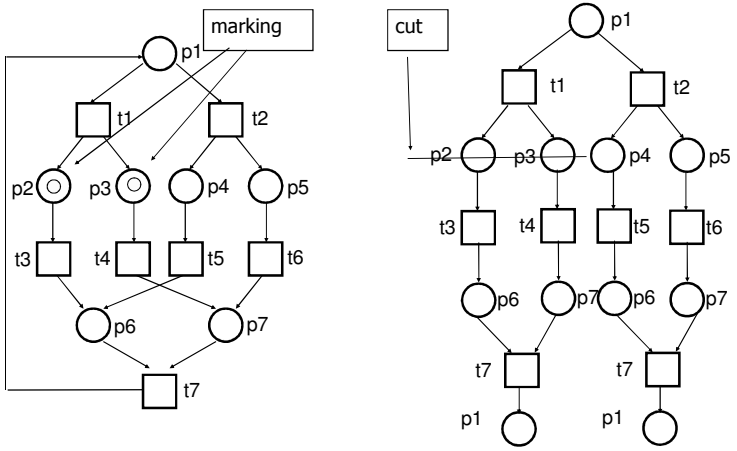
Unfolding construction



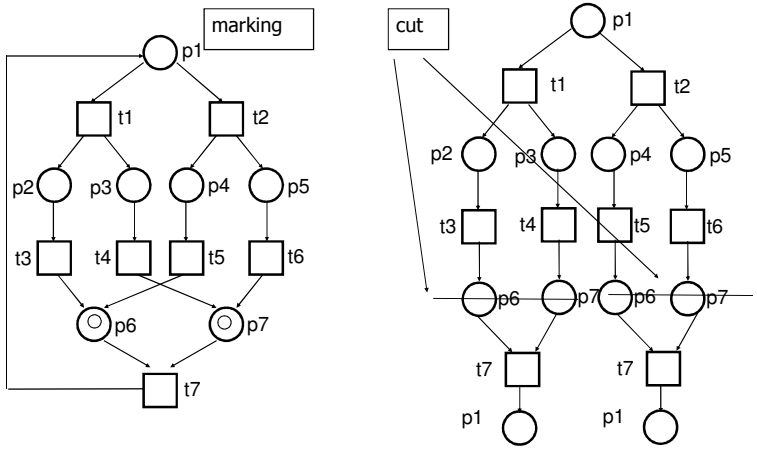
Unfolding construction



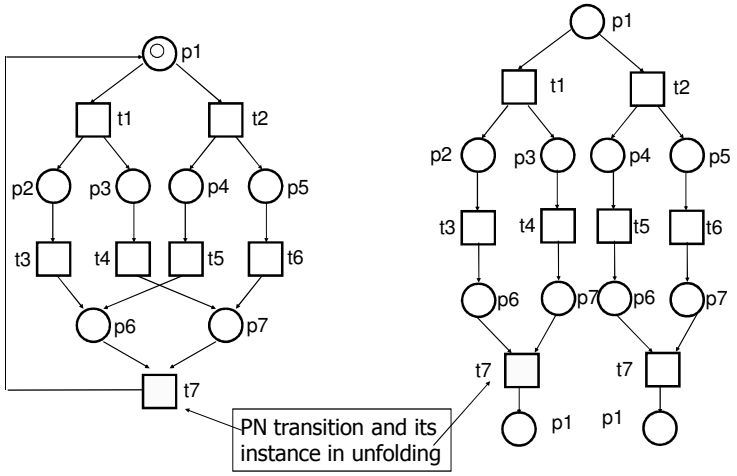
Petri net and its unfolding



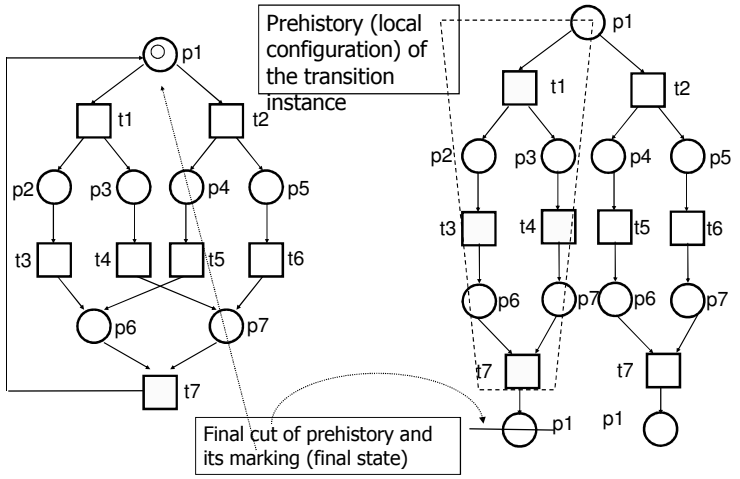
Petri net and its unfolding



Petri net and its unfolding



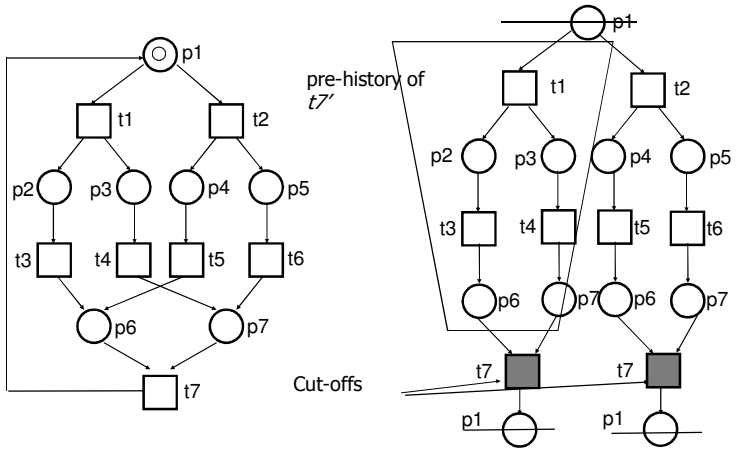
Petri net and its unfolding



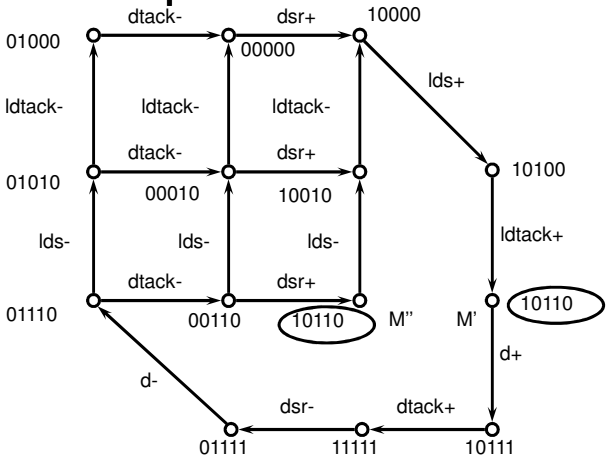
Truncation of unfolding

- At some point of unfolding the process begins to repeat parts of the net that have already been instantiated
- In many cases this also repeats the markings in the form of cuts
- The process can be stopped in every such situation
- Transitions which generate repeated cuts are called cut-off points or simply *cut-offs*
- The unfolding truncated by cut-off is called *prefix*

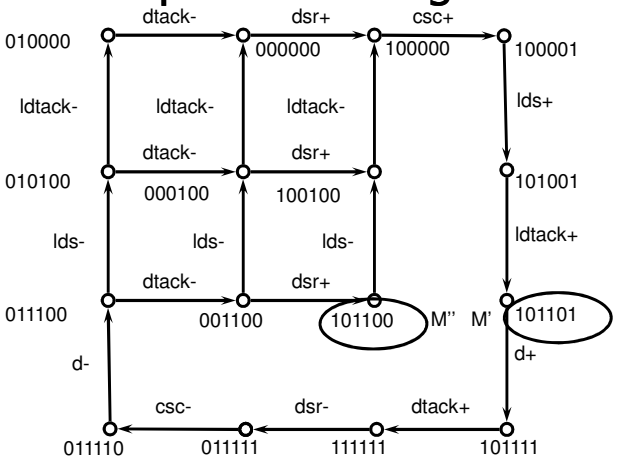
Cutoff transitions



Example: CSC Conflict



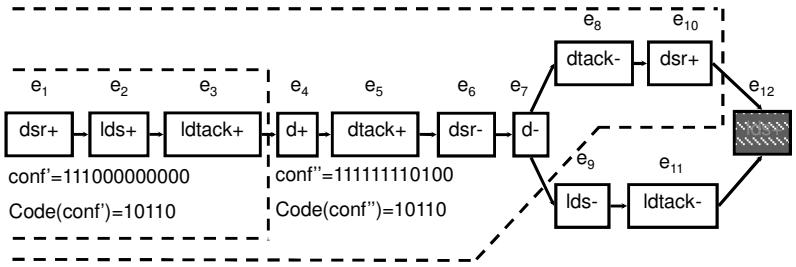
Example: enforcing CSC



Unfoldings

- ☺ Alleviate the state space explosion problem
- ☺ More visual than state graphs
- ☺ Proven efficient for model checking
- ☹ Quite complicated theory
- ☹ Not sufficiently investigated
- ☹ Relatively few algorithms

Translation Into a SAT Problem

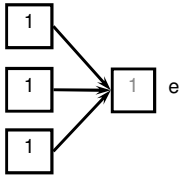


- ▸ Configuration constraint: conf′ and conf″ are configurations
- ▸ Encoding constraint: Code(conf′) = Code(conf″)
- ▸ Separating constraint: Out(conf′) ≠ Out(conf″)

Translation Into a SAT Problem

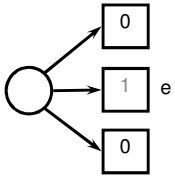
$$Conf(conf') \wedge Conf(conf'') \wedge$$
$$Code(conf',\ldots, val) \wedge Code(conf'',\ldots, val) \wedge$$
$$Out(conf',\ldots, out') \wedge Out(conf'',\ldots, out'') \wedge$$
$$out' \neq out''$$

Configuration constraint



causality

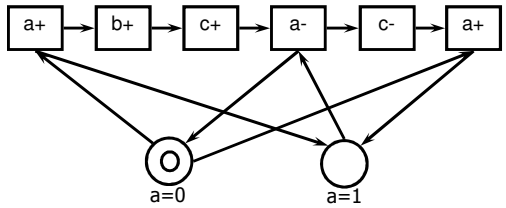
$$\bigwedge_{f \in {}^\bullet({}^\bullet e)} (conf_e \rightarrow conf_f)$$



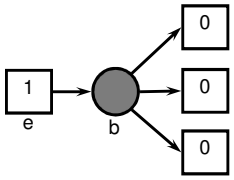
no conflicts

$$\bigwedge_{f \in ({}^\bullet e)^\bullet \setminus \{e\}} \neg(conf_e \wedge conf_f)$$

Tracing the value of a signal



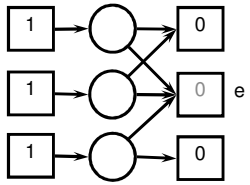
Computing the signals' values



$$cut_b \Leftrightarrow conf_e \wedge \bigwedge_{f \in b^\bullet} \neg conf_f$$

$$val_z \Leftrightarrow \bigvee_{h(b)=p_{z=1}} cut_b$$

Computing the enabled outputs



$$en_e \Leftrightarrow \bigwedge_{f \in {}^\bullet({}^\bullet e)} conf_f \wedge \bigwedge_{f \in ({}^\bullet e)^\bullet} \neg conf_f$$

$$out_z \Leftrightarrow \bigvee_{h(e)=z^\pm} en_e$$

Analysis of the Method

- ☺ A lot of clauses of length 2 – good for BCP
- ☺ The method can be generalized to other coding properties, e.g. *USC* and *normalcy*
- ☺ The method can be generalized to nets with dummy transitions
- ☺ Further optimization is possible for certain net subclasses, e.g. unique-choice nets

Synthesis using unfoldings

```
for each output signal z
  compute (minimal) supports of z
  for each 'promising' support X
    compute the projection of the set
    of reachable encodings onto X
    sorting them according to the
    corresponding values of Nxt_z
    apply Boolean minimization to the
    obtained ON- and OFF-sets
  choose the best implementation of z
```

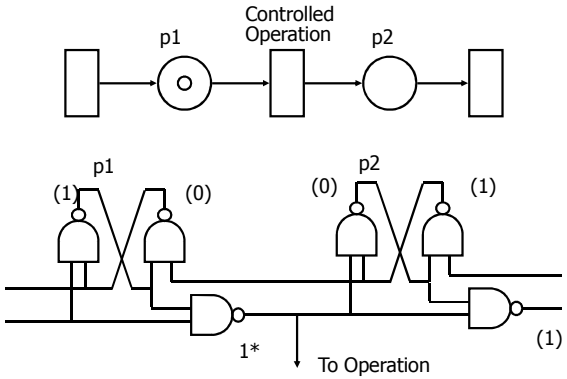
Experimental Results

- ► Unfoldings of STGs are almost always small in practice and thus well-suited for synthesis
- ► Huge memory savings
- ► Dramatic speedups
- ► Every valid speed-independent solution can be obtained using this method, so no loss of quality
- ► We can trade off quality for speed (e.g. consider only minimal supports): in our experiments, the solutions are the same as Petrify's (up to Boolean minimization)
- ► Multiple implementations produced

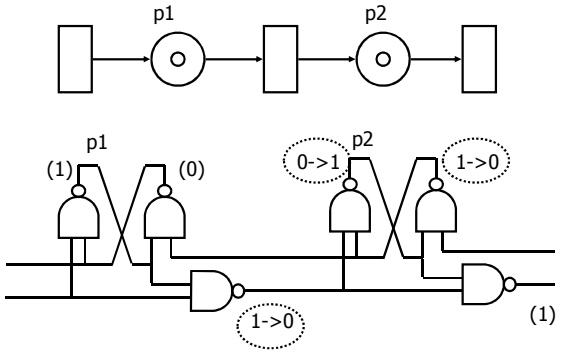
ILP vs. Unfoldings

- ► ILP:
  - Efficient in runtime
  - Incomplete (spurious markings)
- ► Unfoldings:
  - Efficient in runtime (but slower than ILP)
  - Complete
- ► Both methods give synthesis results similar to state-based methods

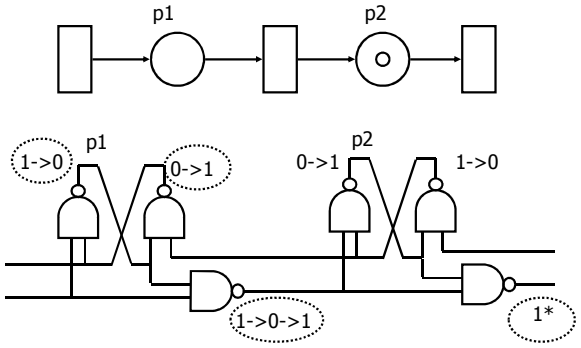
Direct synthesis (Varshavsky's Approach)



Direct synthesis



Direct synthesis

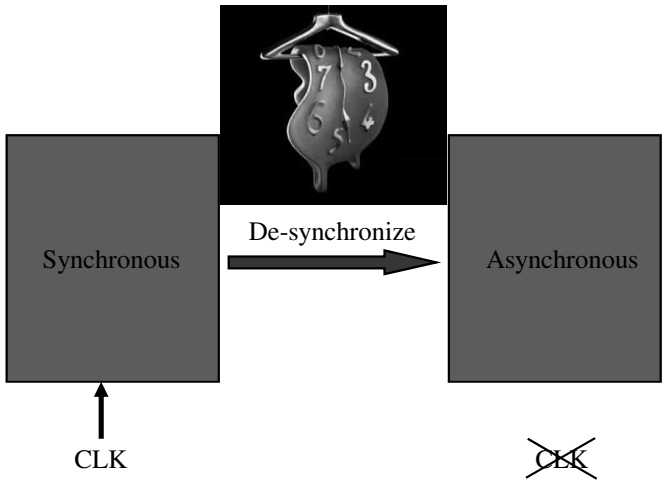


De-synchronization: from synchronous to asynchronous

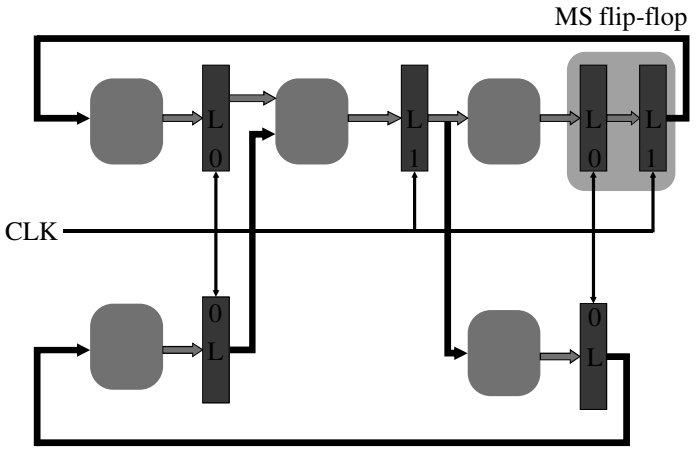
Based on the paper:
Blunno, Cortadella, Kondratyev, Lavagno, Lwin, Sotiriou,
Handshake protocols for de-synchronization,
ASYNC 2004.

Outline

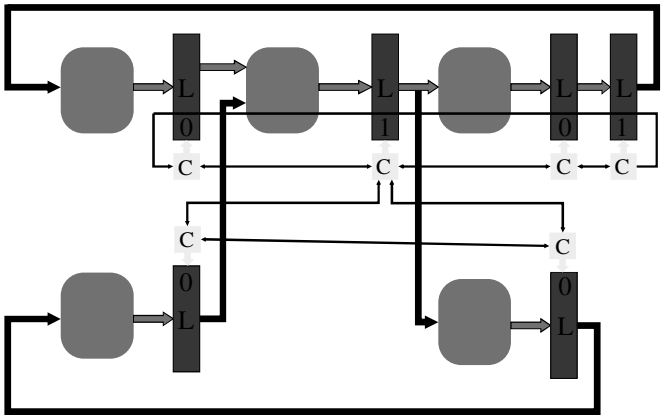
- What is de-synchronization ?
- Behavioral equivalence
- 4-phase protocols for de-synchronization
- Concurrency
- Correctness
- An example



Synchronous circuit

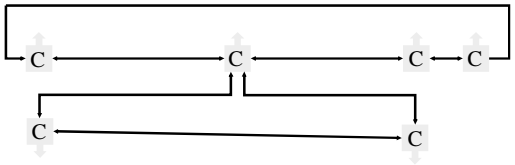


De-synchronization



De-synchronization

Distributed controllers substitute the clock network



The data path remains intact !

Design flow

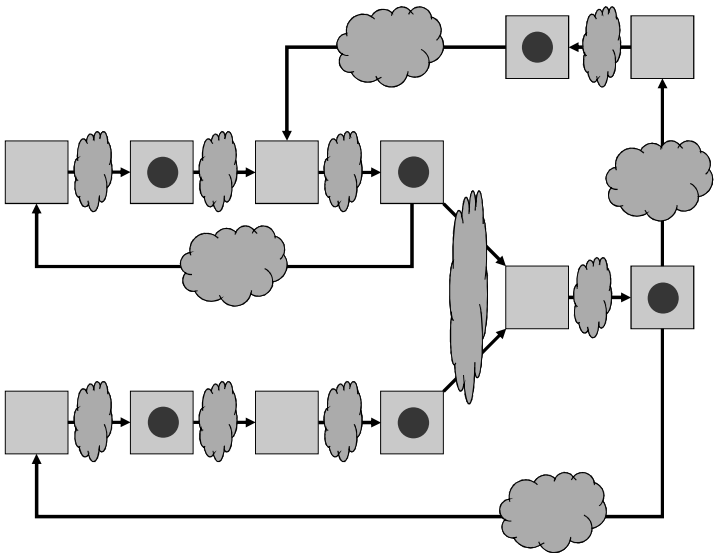
- Think synchronous
- Design synchronous:
  one clock and edge-triggered flip-flops
- De-synchronize (automatically)
- Run it asynchronously

Prior work

- Micropipelines (Sutherland, 1989)
- Local generation of clocks
  - Varshavsky et al., 1995
  - Kol and Ginosar, 1996
- Theseus Logic (Ligthart et al., 2000)
  - Commercial HDL synthesis tools
  - Direct translation and special registers
- Phased logic (Linder and Harden, 1996)
  (Reese, Thornton, Traver, 2003)
  - Conceptually similar
  - Different handshake protocol (2 phase vs. 4 phase)

Automatic de-synchronization

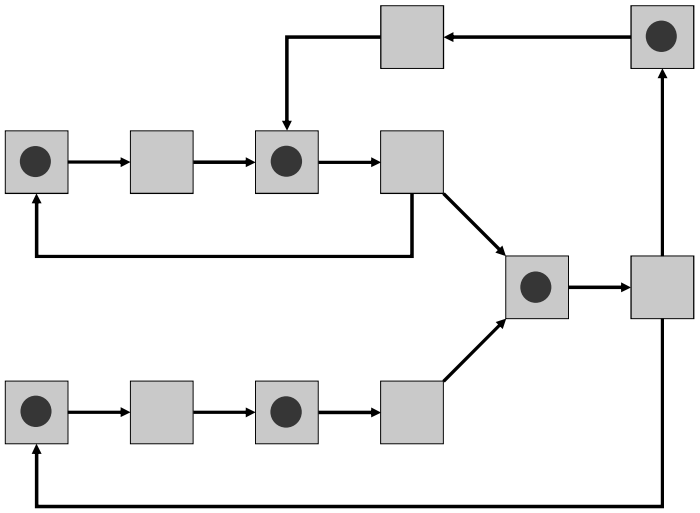
- Devise an *automatic method* for de-synchronization
- Identify a *subclass of synchronous circuits* suitable for de-synchronization
- *Formally prove correctness*



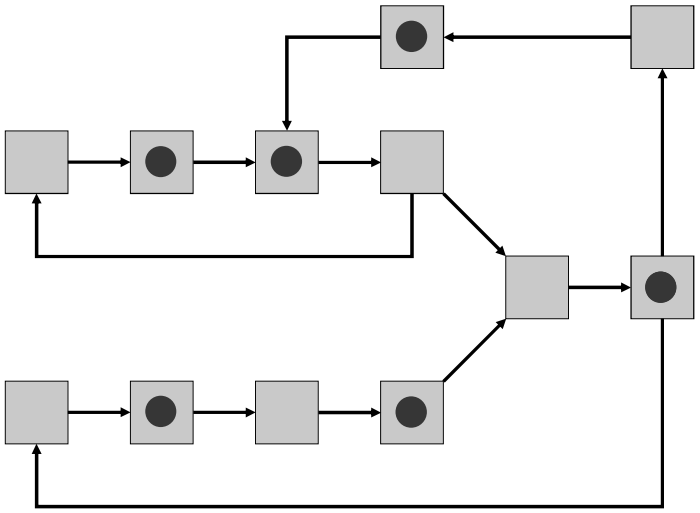
Outline

- What is de-synchronization ?
- ***Behavioral equivalence***
- 4-phase protocols for de-synchronization
- Concurrency
- Correctness
- An example

Synchronous flow

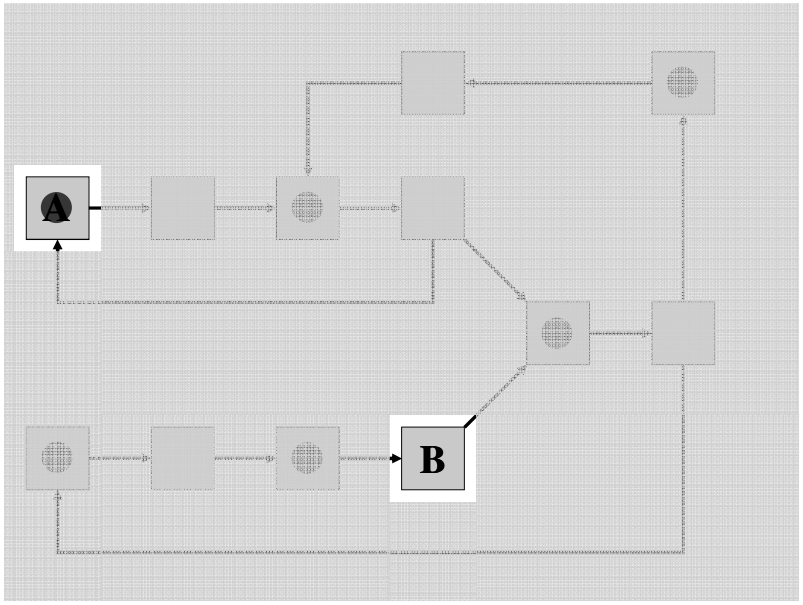


De-synchronized flow

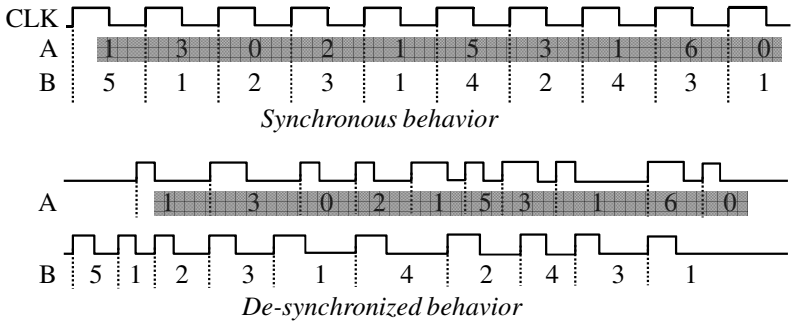


Flow equivalence

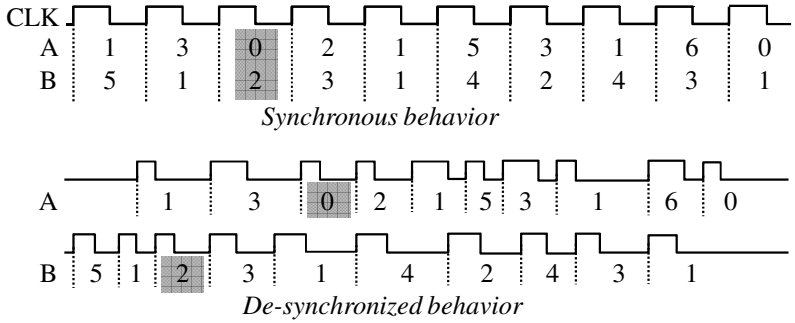
[Guernic, Talpin, Lann, 2003]



Flow equivalence

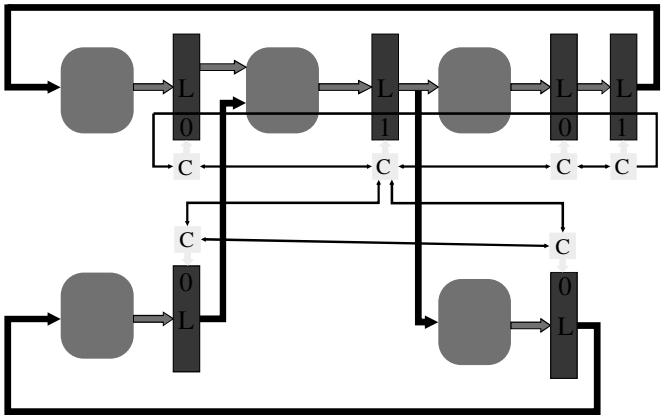


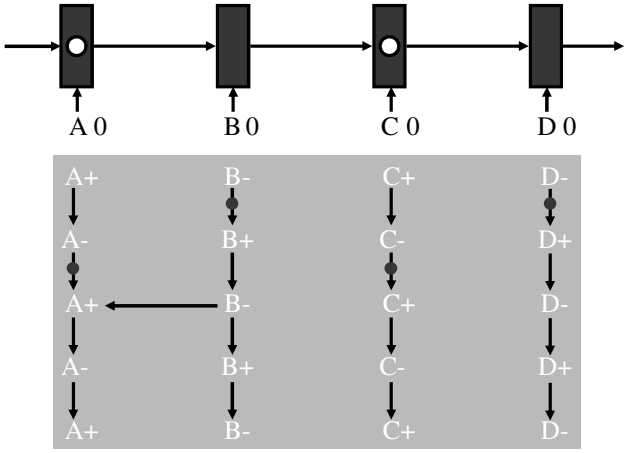
Flow equivalence



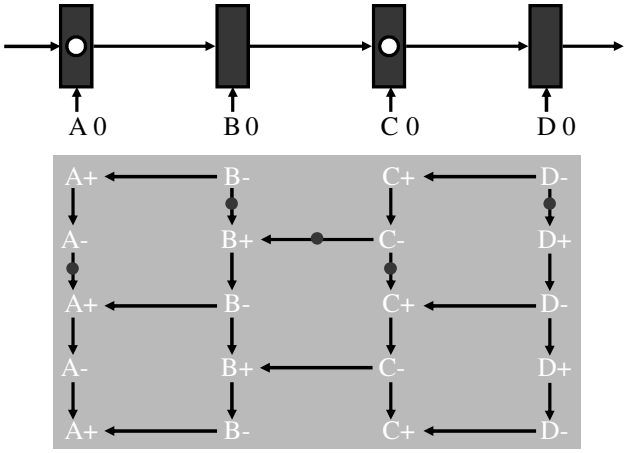
Outline

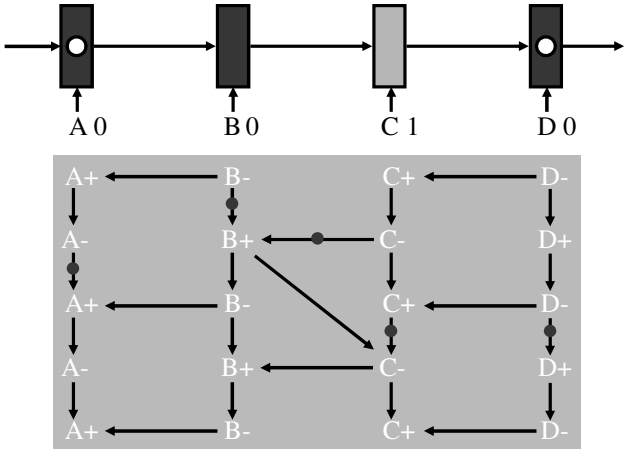
- What is de-synchronization ?
- Behavioral equivalence
- ***4-phase protocols for de-synchronization***
- Concurrency
- Correctness
- An example



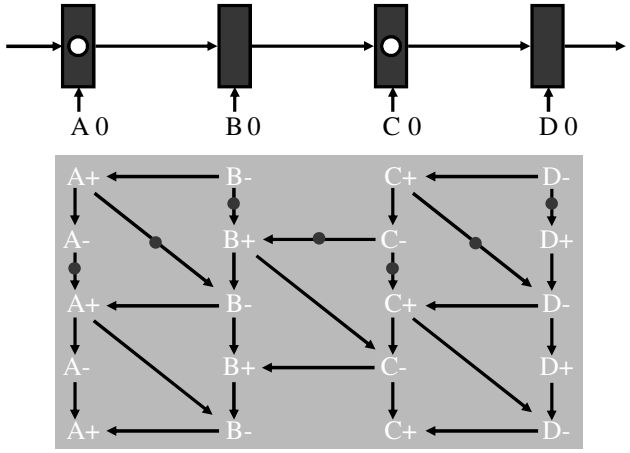


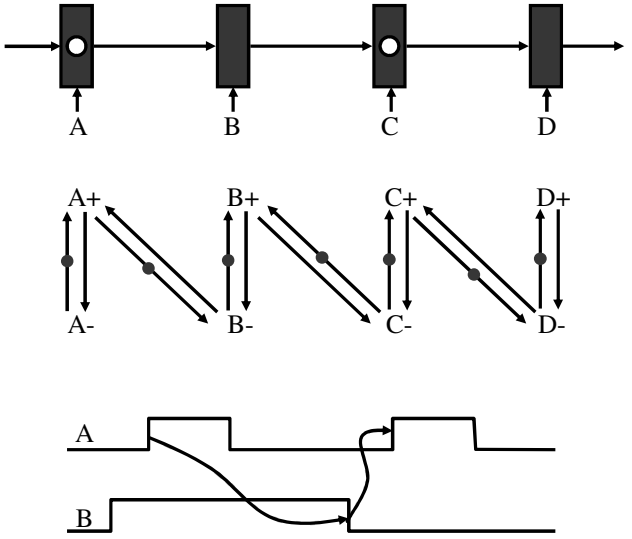
A latch cannot read another data item until the successor has captured the current one





A latch cannot become opaque before having captured the data item from its predecessor

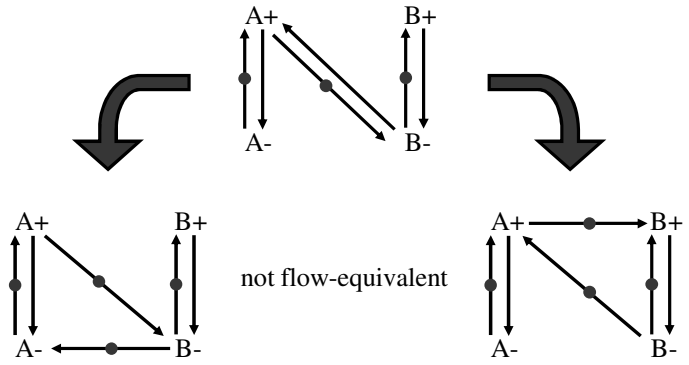


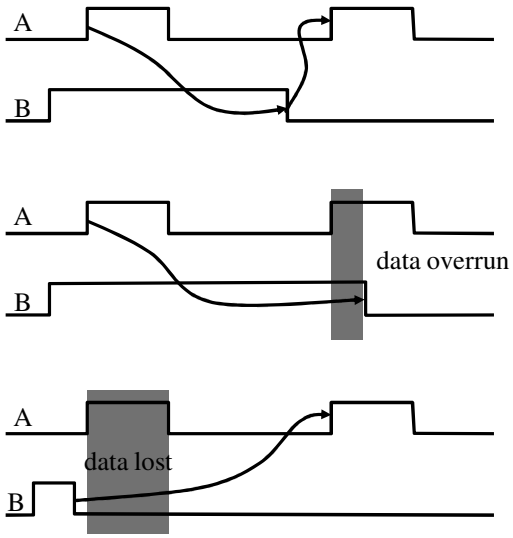


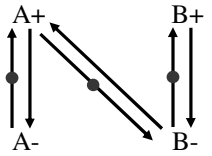
Outline

- What is de-synchronization ?
- Behavioral equivalence
- 4-phase protocols for de-synchronization
- ***Concurrency***
- Correctness
- An example

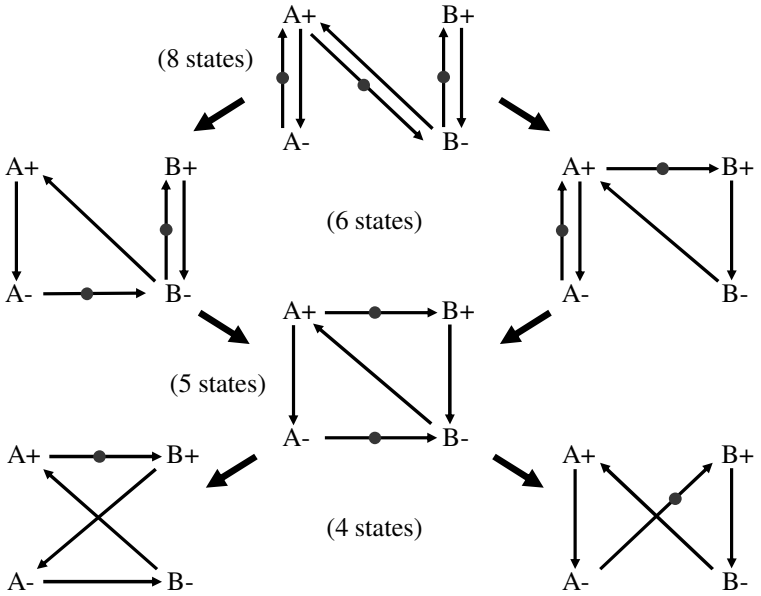
Can we increase concurrency ?

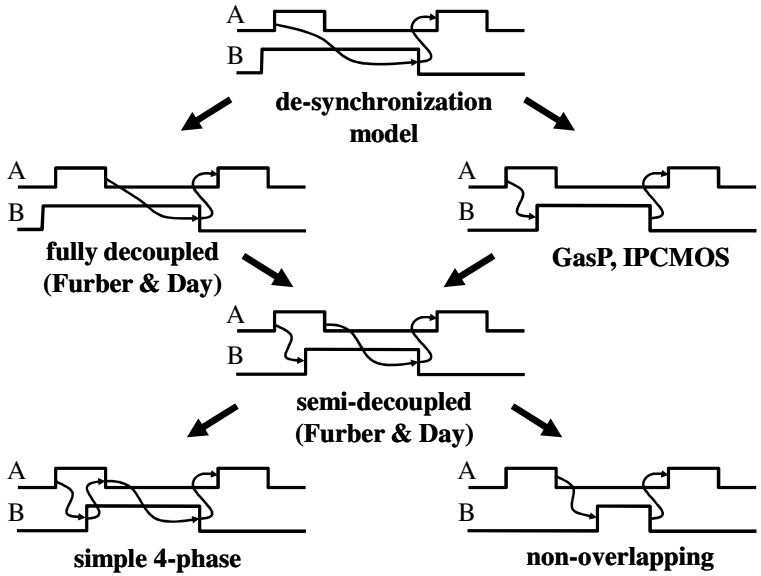


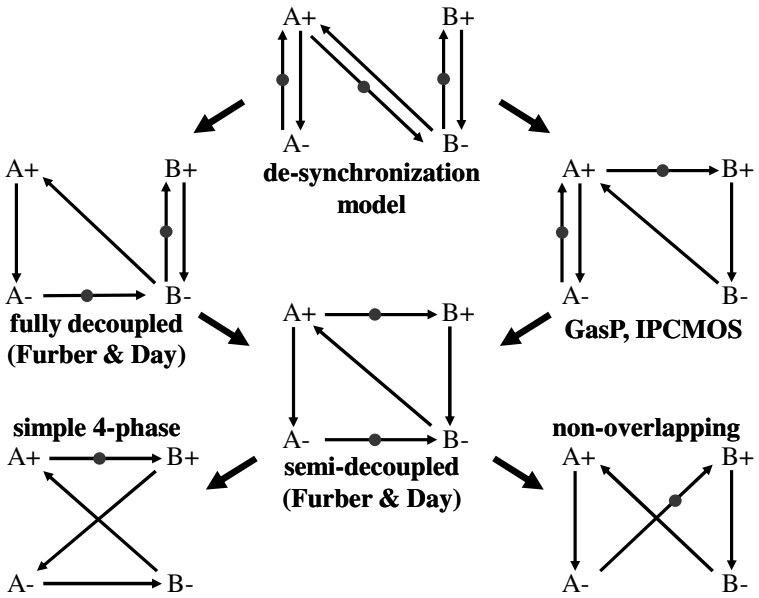




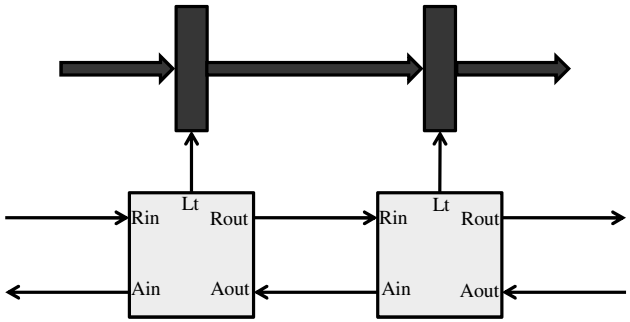
Can we reduce concurrency ? How much ?







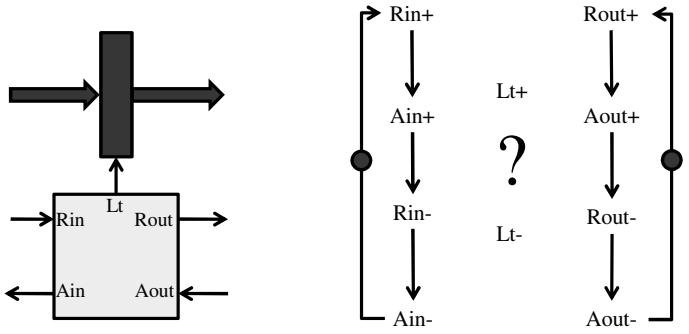
4-phase latch controllers



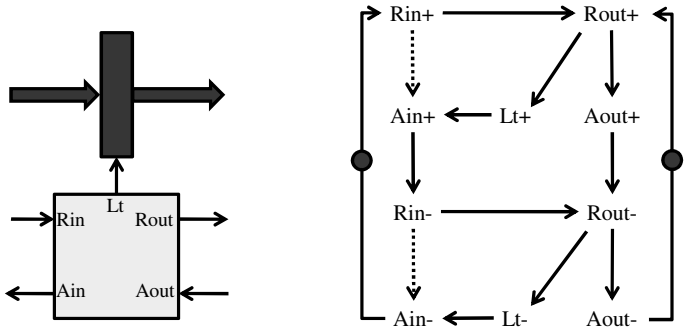
Furber and Day, IEEE Trans. VLSI, June 1996

Implementation note: Lt=0 (transparent), Lt=1 (opaque)

4-phase latch controllers

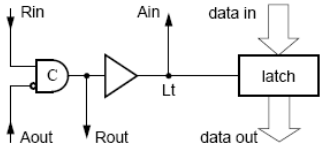


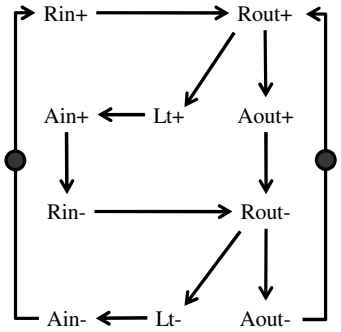
4-phase latch controllers



Simple 4-phase controller

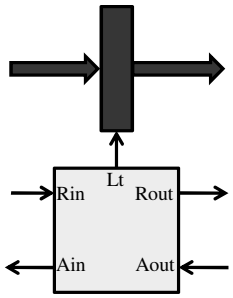
4-phase latch controllers

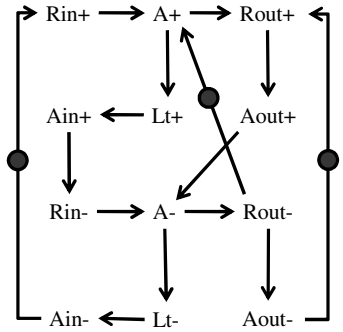




Simple 4-phase controller

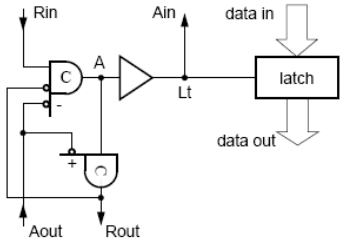
4-phase latch controllers

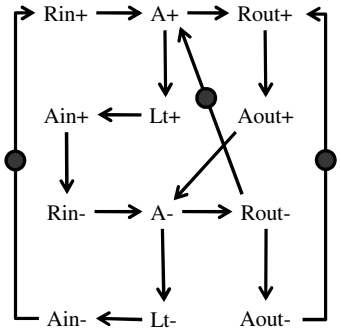




Semi-decoupled controller

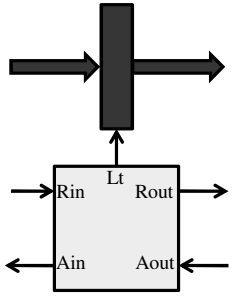
4-phase latch controllers

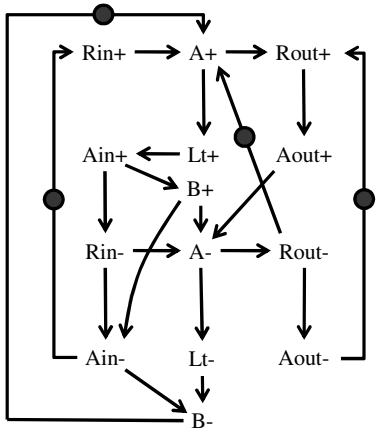




Semi-decoupled controller

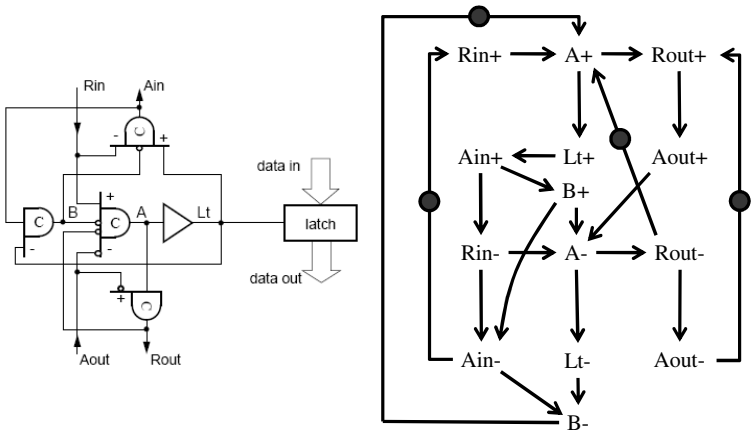
4-phase latch controllers





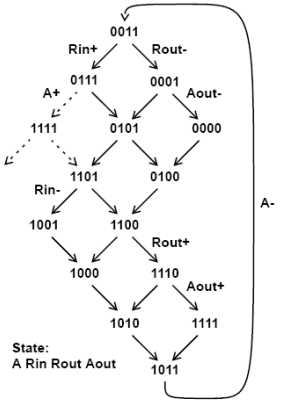
Fully decoupled controller

4-phase latch controllers

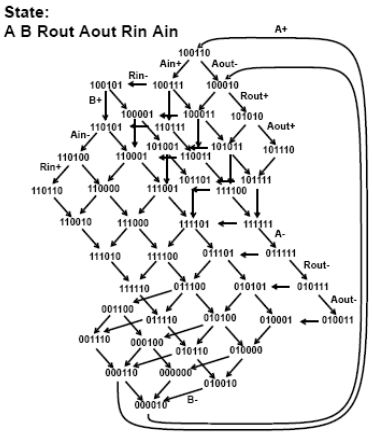


Fully decoupled controller

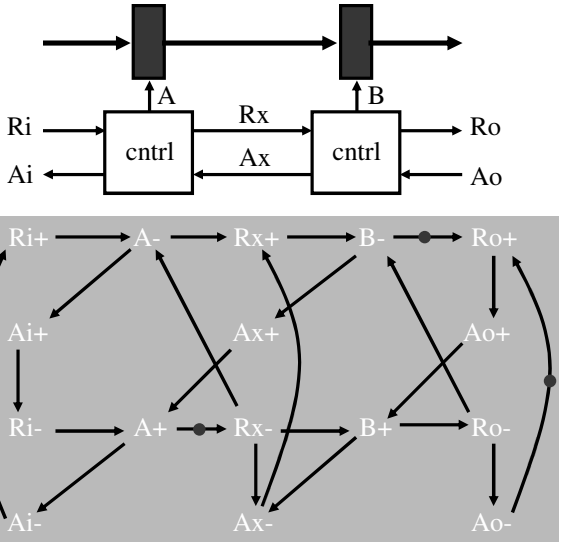
4-phase latch controllers (state graphs)



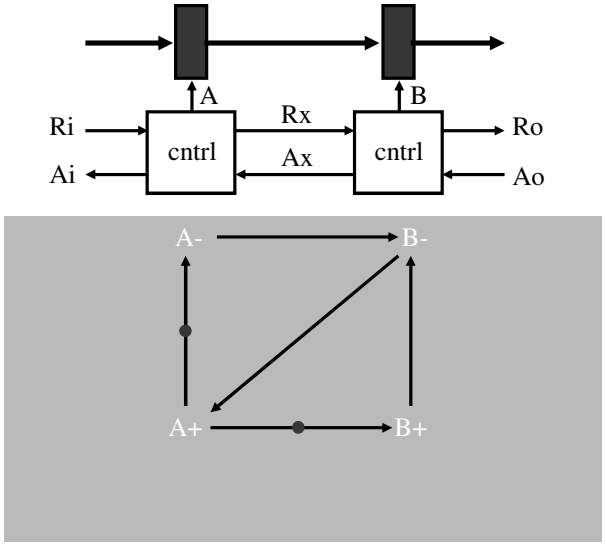
Semi-decoupled controller



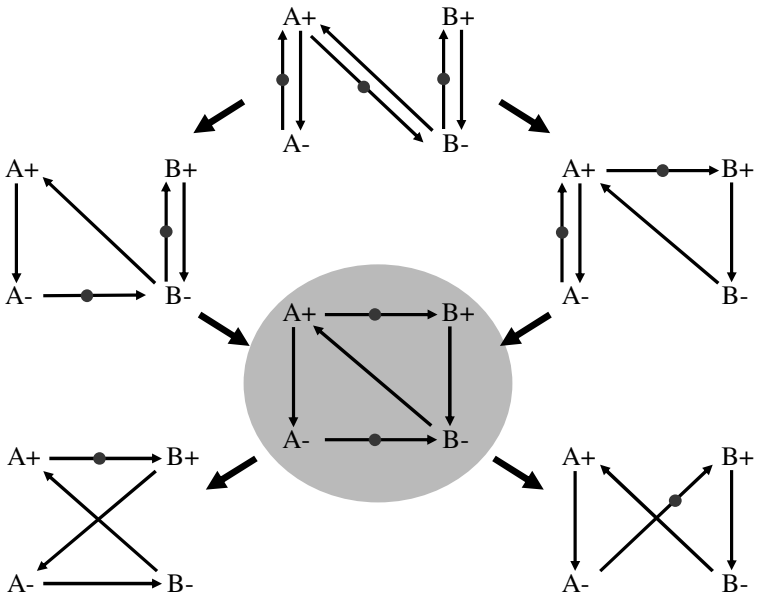
Fully decoupled controller



(semi-decoupled 4-phase protocol)

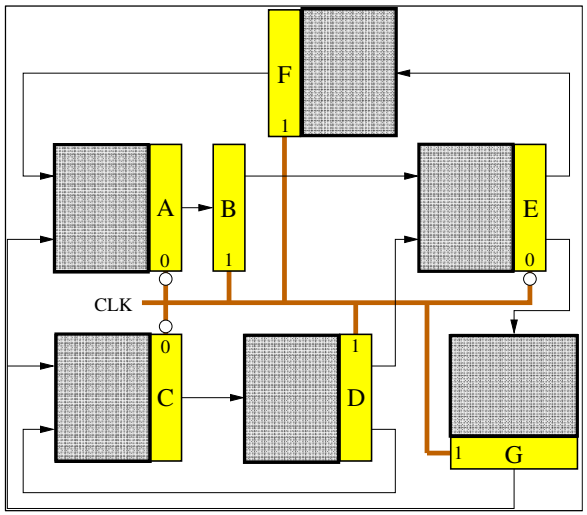


(semi-decoupled 4-phase protocol)

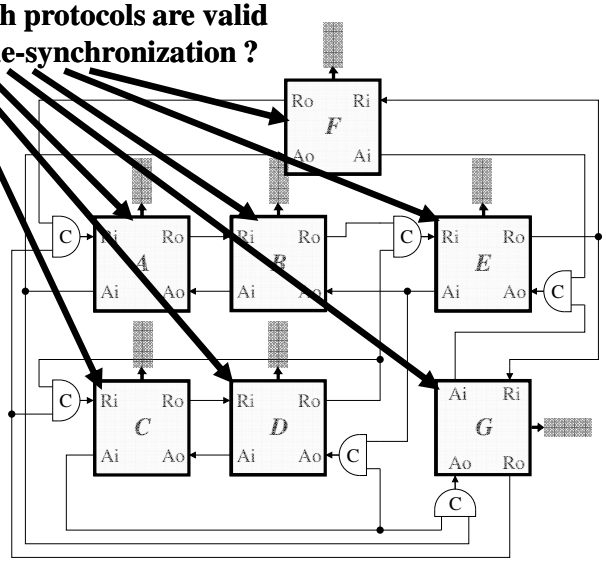


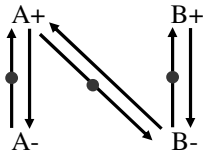
Outline

- What is de-synchronization ?
- Behavioral equivalence
- 4-phase protocols for de-synchronization
- Concurrency
- ***Correctness***
- An example



Which protocols are valid for de-synchronization ?

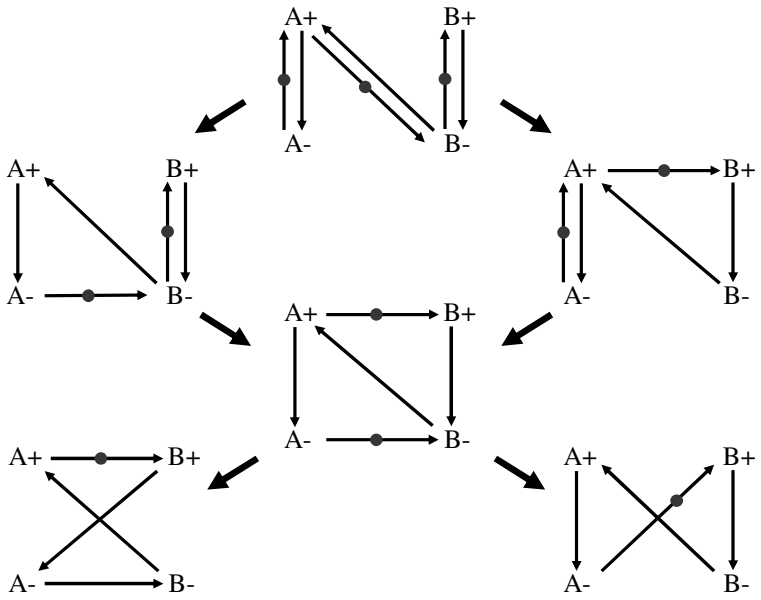




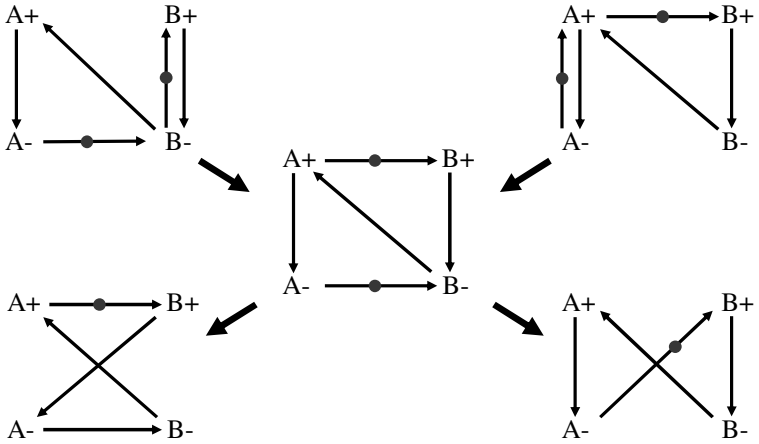
Theorem:
the de-synchronization protocol
preserves flow-equivalence

Proof: by induction on the length of the traces

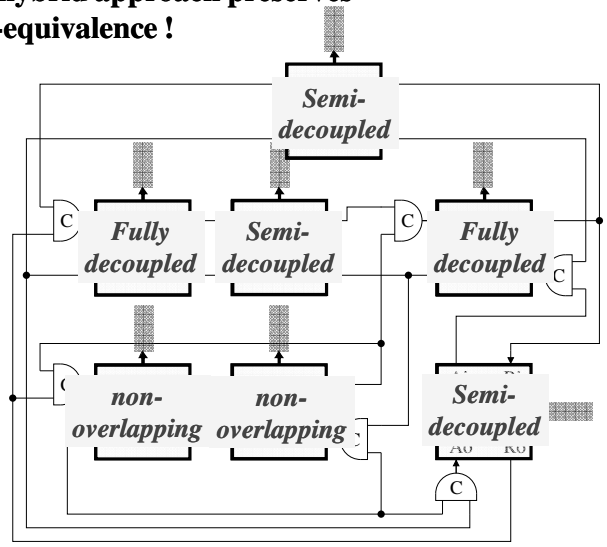
Induction hypothesis: same latch values at reset
Induction step:
same values at cycle *i* ➔ same values at cycle *i+1*

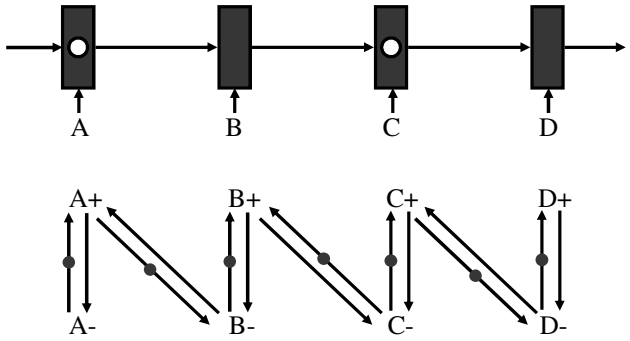


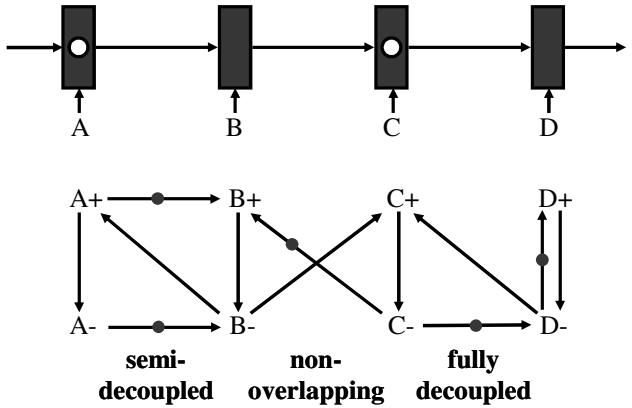
Theorem:
any reduction in concurrency preserves flow-equivalence



Any hybrid approach preserves flow-equivalence !







Flow-equivalence is preserved, … but …

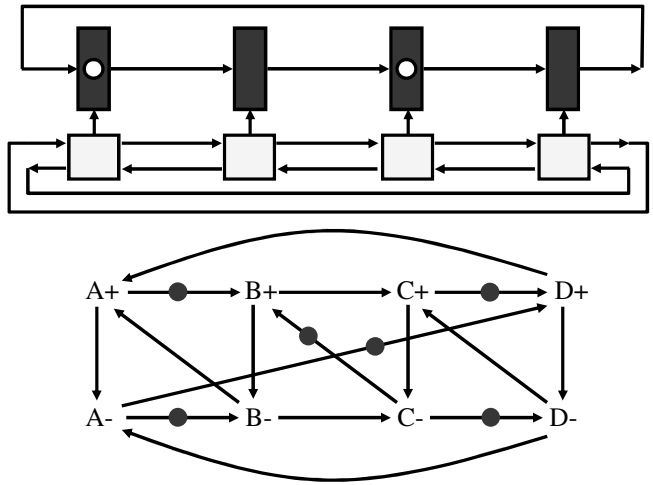
Liveness

- Preservation of flow-equivalence:

  all the generated traces are equivalent

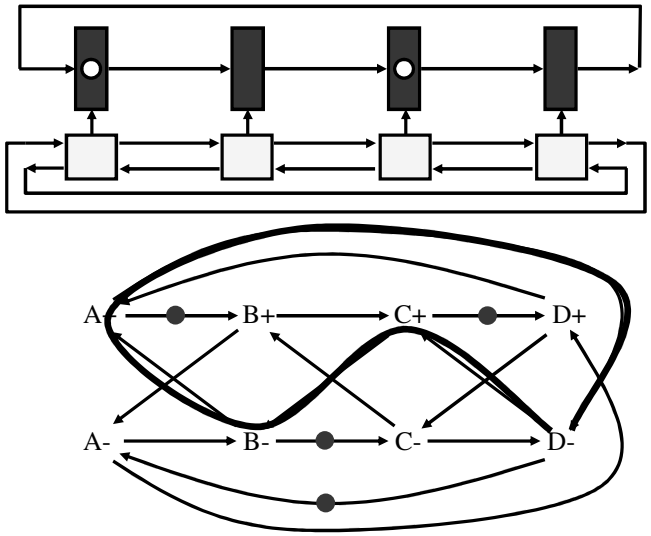
- Are all traces generated ?
  (Is the marked graph live ?)

  Not always !



Semi-decoupled 4-phase handshake protocol

Liveness: all cycles have at least one token [Commoner 1971]



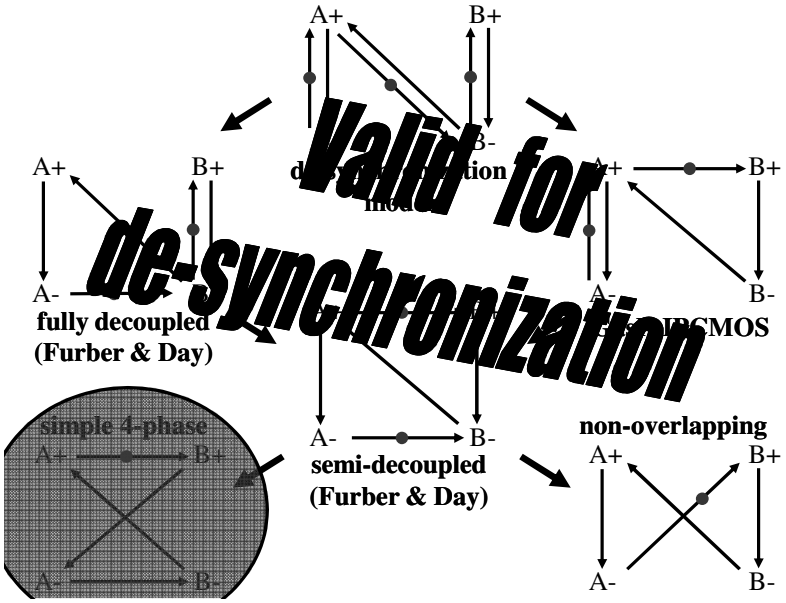
Simple 4-phase handshake protocol

Results about liveness

- At least three latches in a ring are required with only one data token circulating
  [*Muller 1962*]

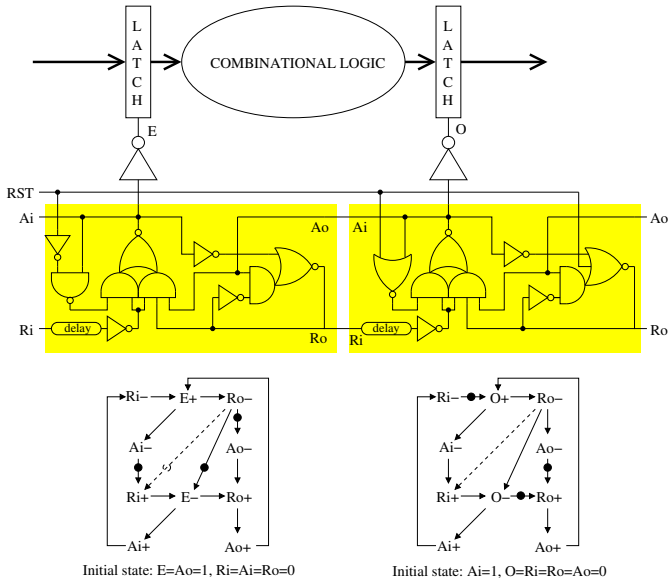
- **Theorem** (this paper):
  any hybrid combination of protocols is live if the simple 4-phase protocol is not used

  Proof: *any cycle has at least one token*

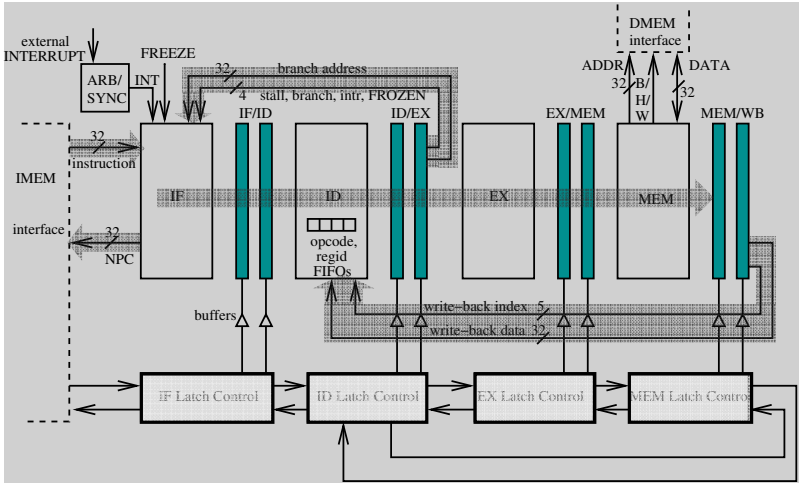


Outline

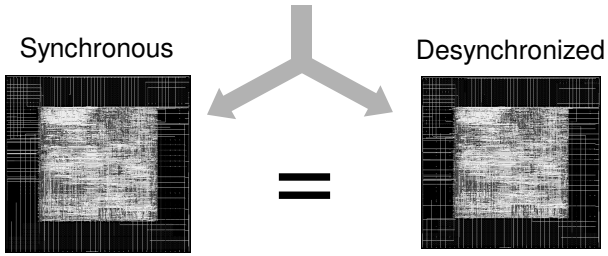
- What is de-synchronization ?
- Behavioral equivalence
- 4-phase protocols for de-synchronization
- Concurrency
- Correctness
- ***An example***



Async DLX block diagram



Synchronous RTL



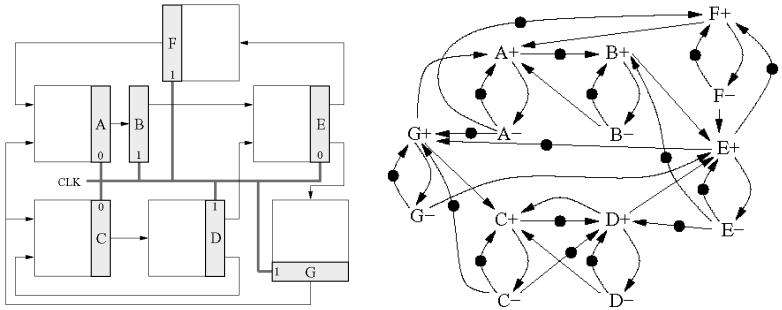
Cycle: 4.4ns
Power: 70.9mW
Area: 372,656µm

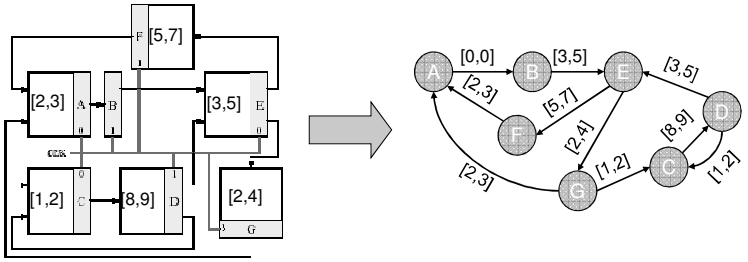
Cycle: 4.45ns
Power: 71.2mW
Area: 378,058µm

- All numbers are after Placement & Routing
- Total of 1500 flip-flops, 3000 latches
- DE-SYNC design includes 5 controllers, each driving 2 clock trees
- Power numbers include the clock tree
- Technology: UCM/Virtual Silicon 0.18 µm

Discussion

- The de-synchronization model provides an abstraction of the timing behavior





- Timing analysis
- Exploration of the design space

Conclusions

- EDA tools require a ***formal support*** (they must work for ***all*** circuits)
- A complete characterization of 4-phase protocols has been presented (partial order based on concurrency)
- Design flow developed at Cadence Berkeley Labs
  - Automated from gate netlist
  - Static timing analysis to derive matched delays
  - Constrained P&R to meet timing constraints

Part 2: Synchronous Elastic Systems

Jordi Cortadella and Mike Kishinevsky

- Synchronous elastic systems also called
  - Latency tolerant systems or
  - Latency insensitive systems
- We use term “synchronous elastic” since better linked to asynchronous elastic

Agenda of Part 2

I. Basics of elastic systems

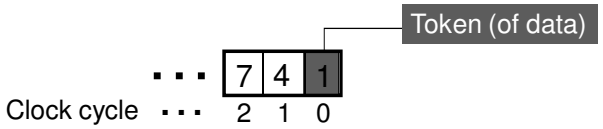
II. Early evaluation and performance analysis

III. Optimization of elastic systems and their correctness

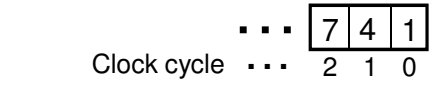
I

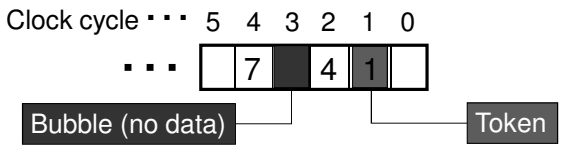
- What and Why
- Intuition
- How to design elastic systems
- Converting synchronous system to elastic
- Micro-arch opportunities
- Marked Graph models
- Performance evaluation

Synchronous Stream of Data

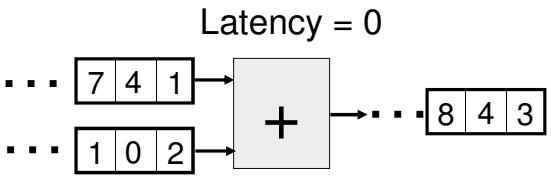


Synchronous Elastic Stream

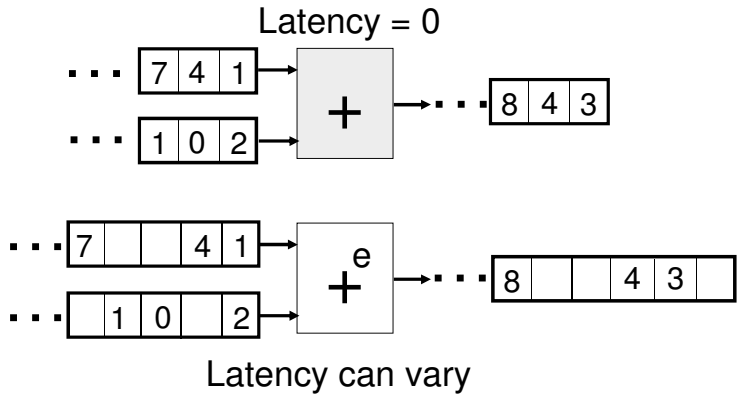




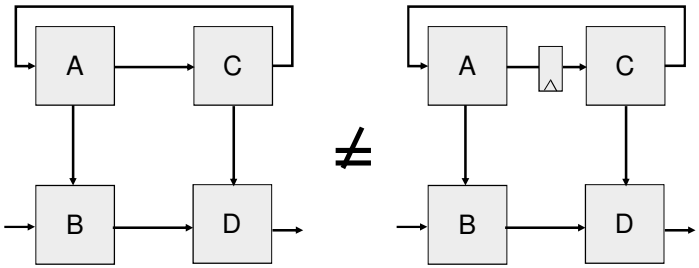
Synchronous Circuit



Synchronous Elastic Circuit

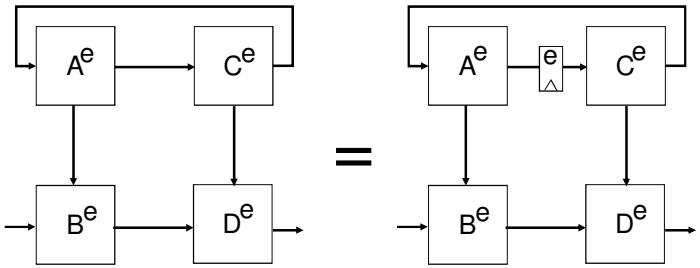


Ordinary Synchronous System



Changing latencies changes behavior

Synchronous Elastic (characteristic property)

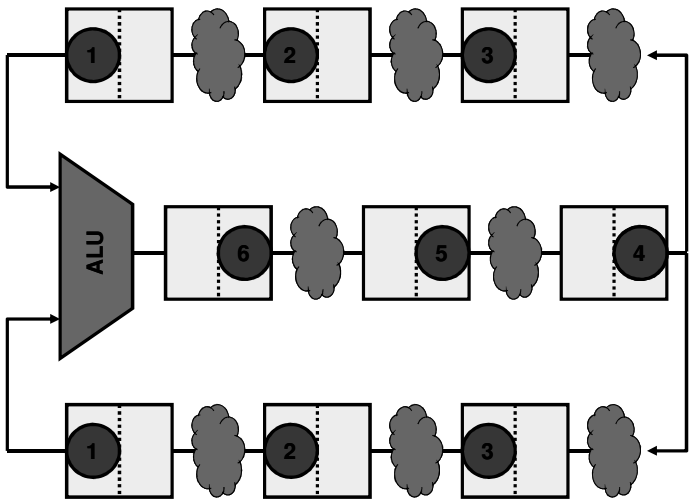


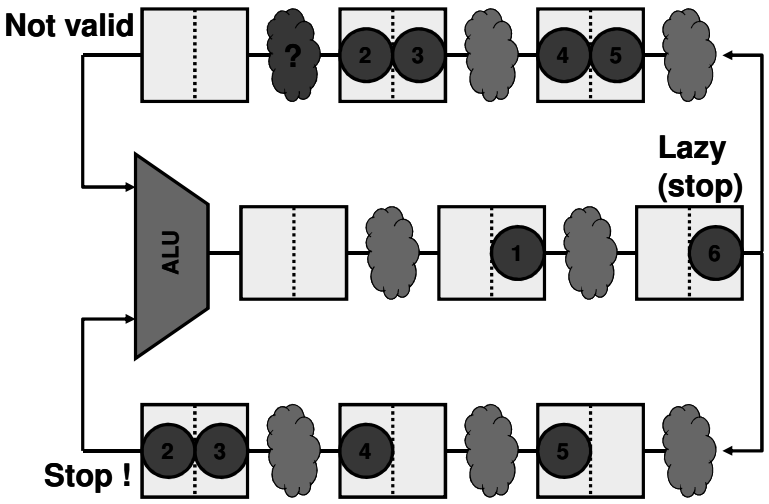
Changing latencies does NOT change behavior
= time elasticity

Why

- Scalable
- Modular (Plug & Play)
- Better energy-delay trade-offs
  (design for typical case instead of worst case)
- New micro-architectural opportunities
  in digital design
- Not asynchronous: use existing design
  experience, CAD tools and flows

Example of elastic behavior



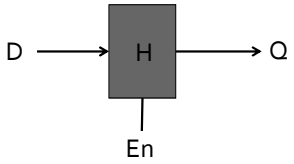


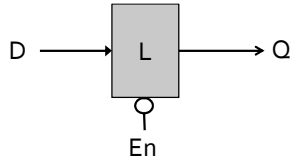
How to design elastic systems

We show an example of the implementation:
SELF = Synchronous Elastic Flow

Others are possible

Reminder:
Memory elements. Transparent latches

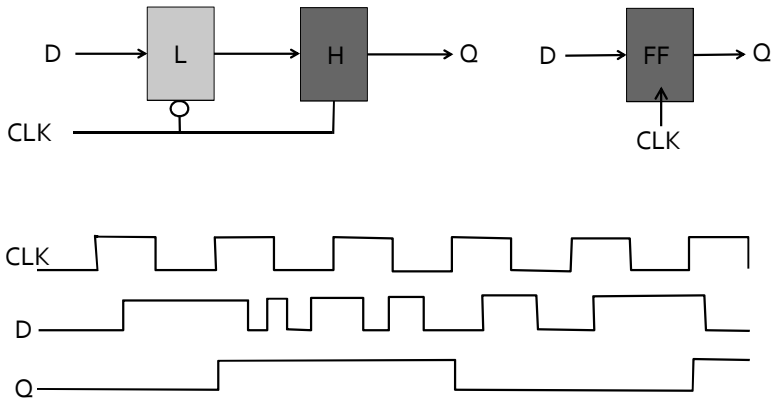




Active high:	Active low:
En = 0 (opaque): Q = prev(Q)	En = 1 (opaque): Q = prev(Q)
En = 1 (transparent): Q = D	En = 0 (transparent): Q = D

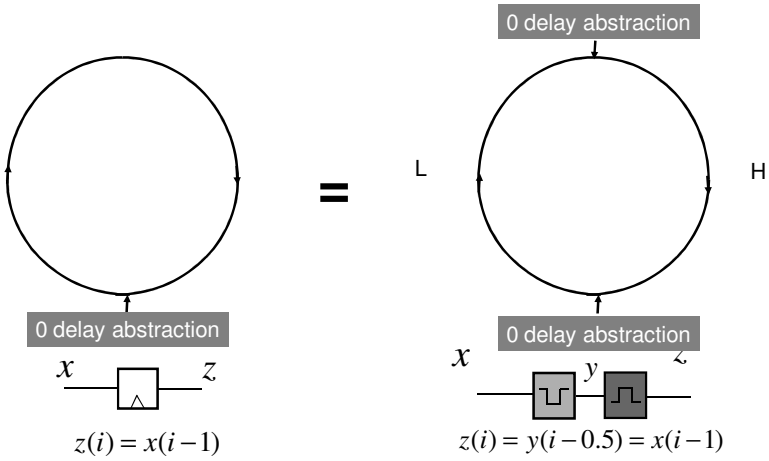
16

Reminder:
Memory elements. Flip-flop



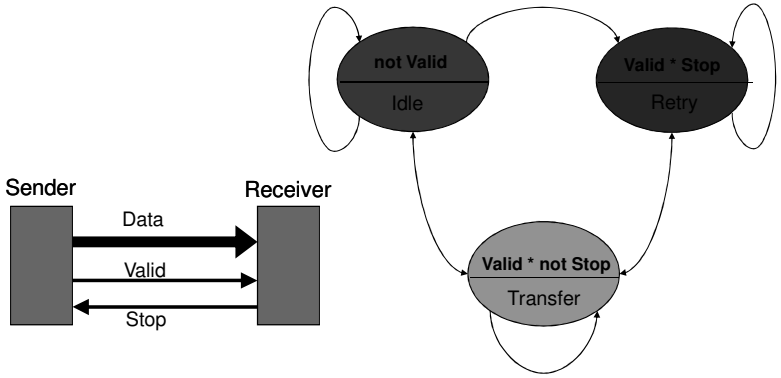
17

Reminder: Clock cycle = two phases

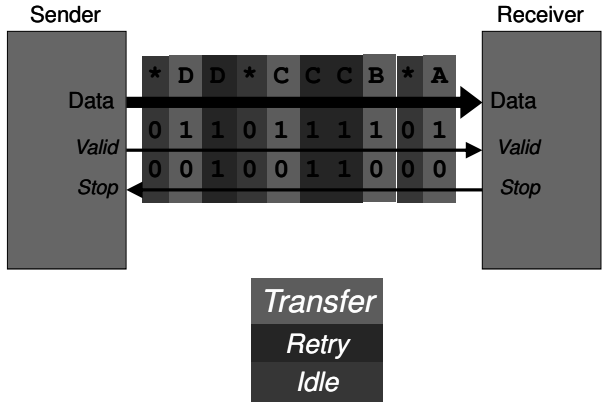


18

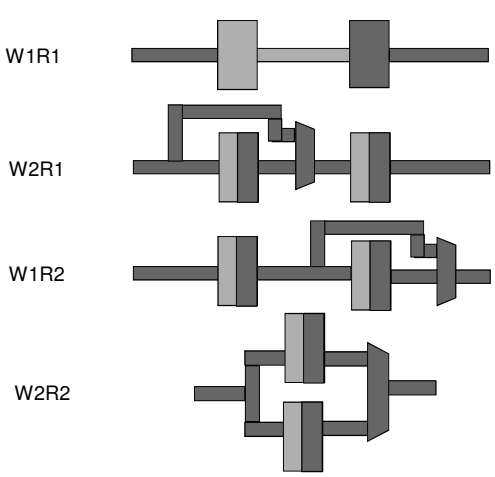
Elastic channel protocol



Elastic channel protocol



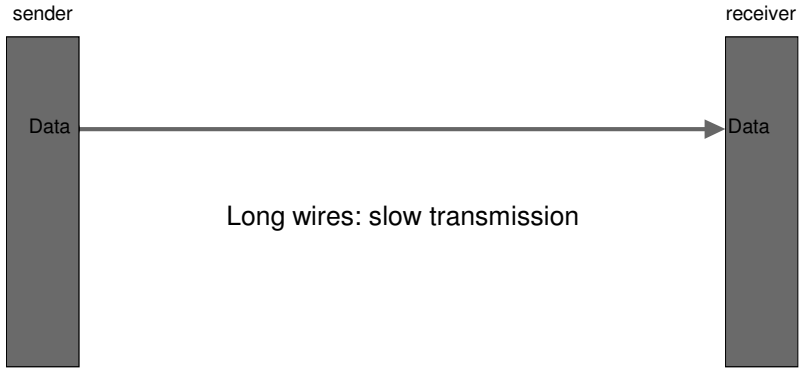
Elastic buffer keeps data while stop is in flight



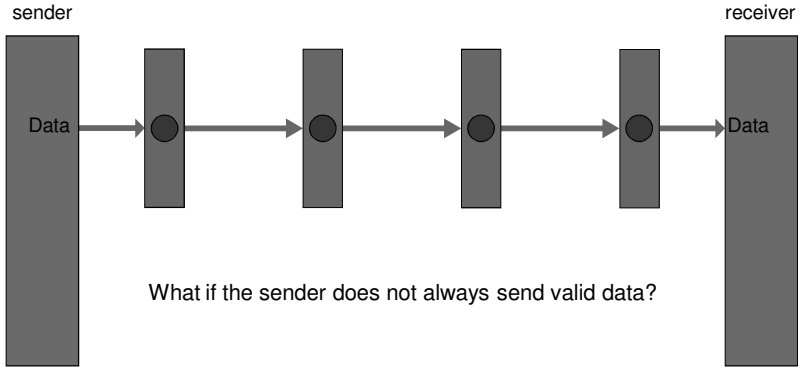
Cannot be done with Single Edge Flops without double pumping

Can use latches inside Master-Slave

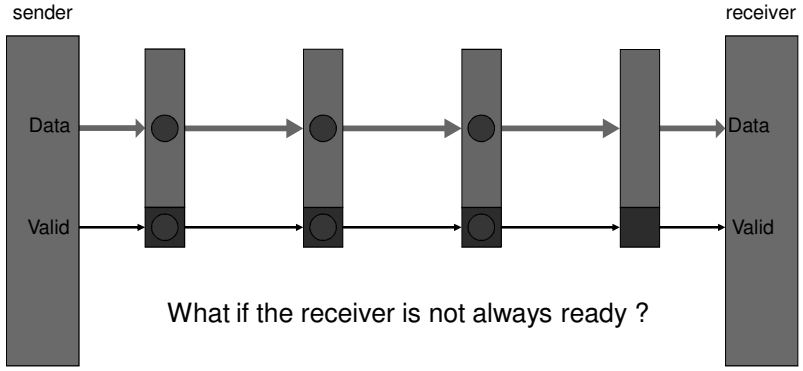
Communication channel



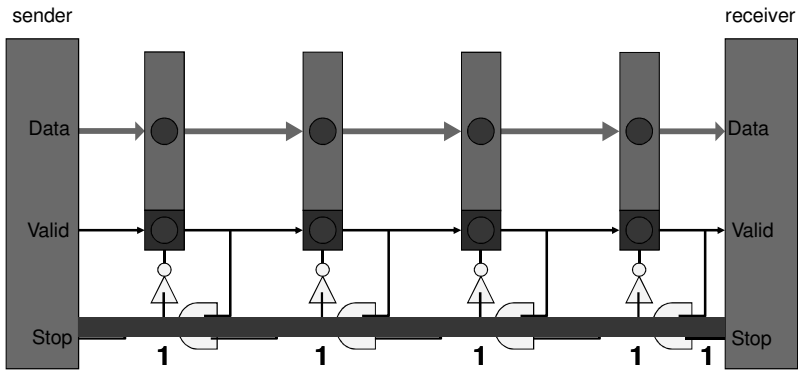
Pipelined communication



The Valid bit

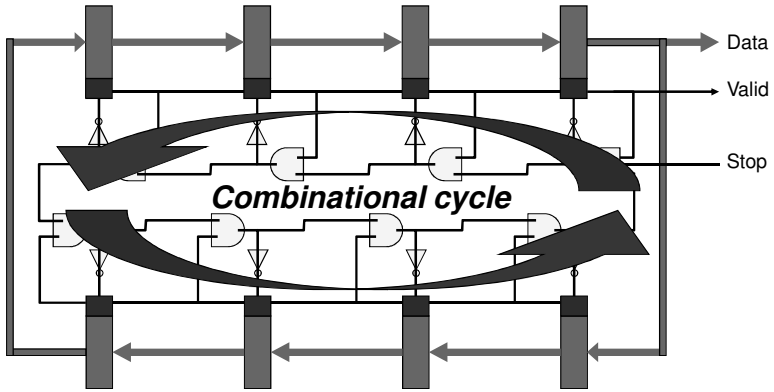


The Stop bit



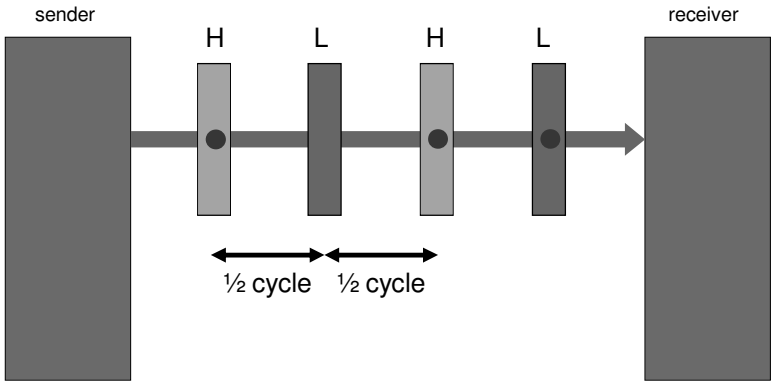
Back-pressure

Cyclic structures



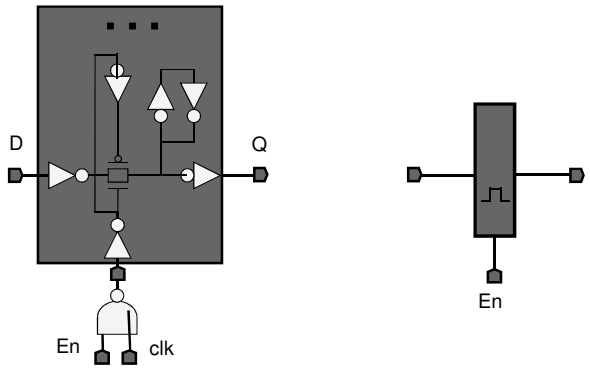
One can build circuits with combinational cycles (constructive cycles by Berry), but synthesis and timing tools do not like them

Example: pipelined linear communication chain with transparent latches

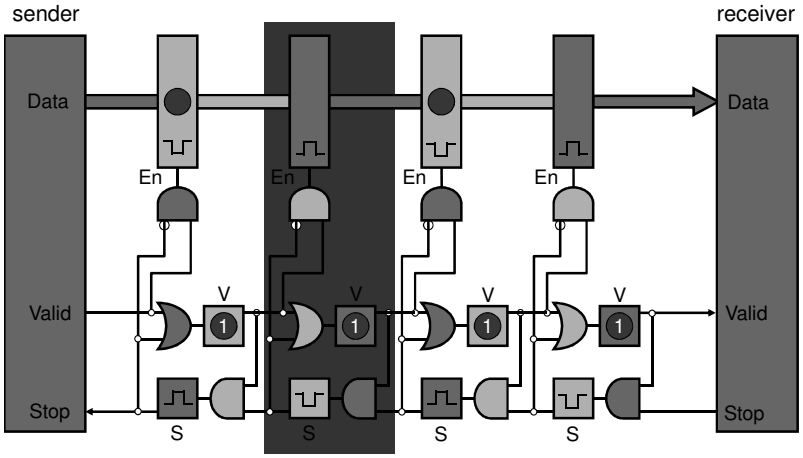


Master and slave latches with independent control

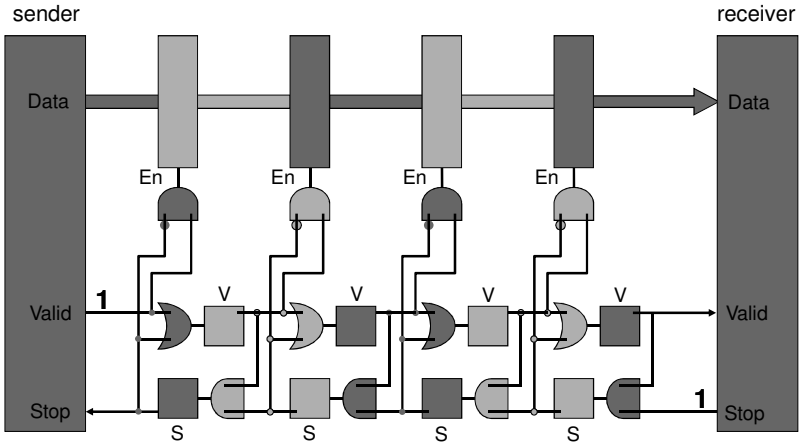
Shorthand notation (clock lines not shown)



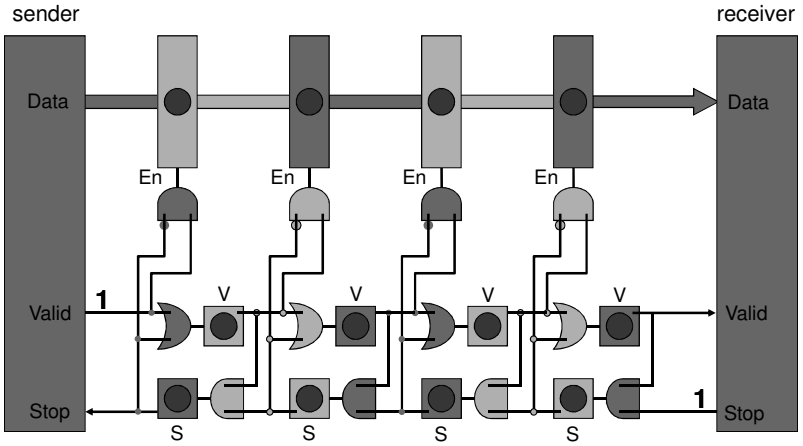
SELF (linear communication)



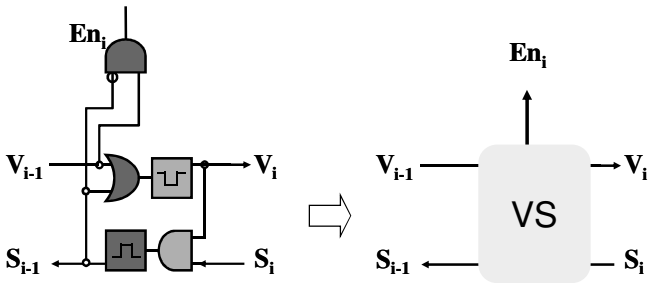
SELF



SELF

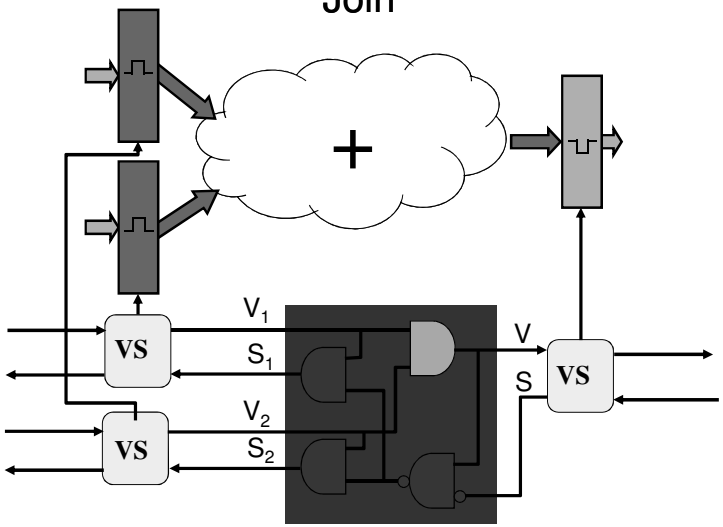


Basic VS block

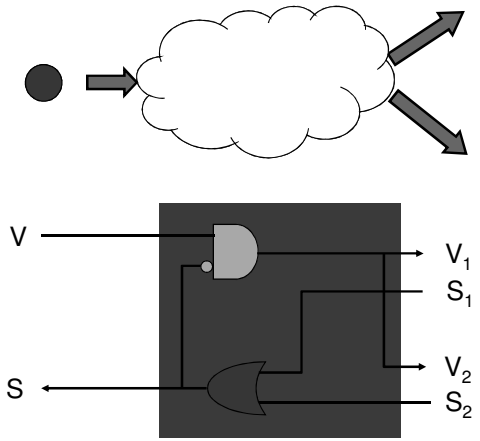


VS block + data-path latch = elastic HALF-buffer

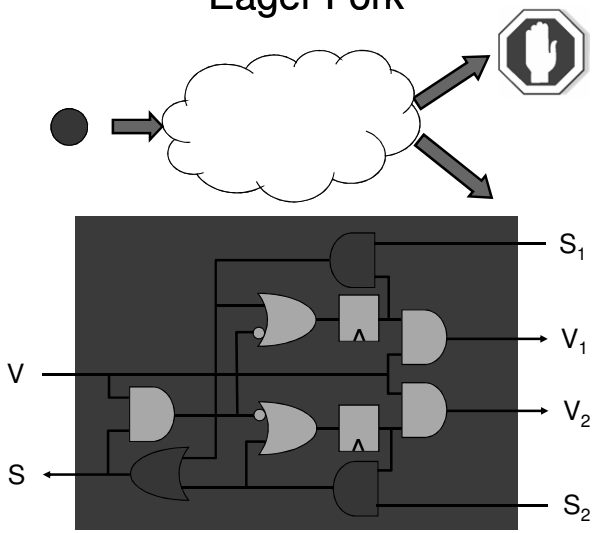
Join



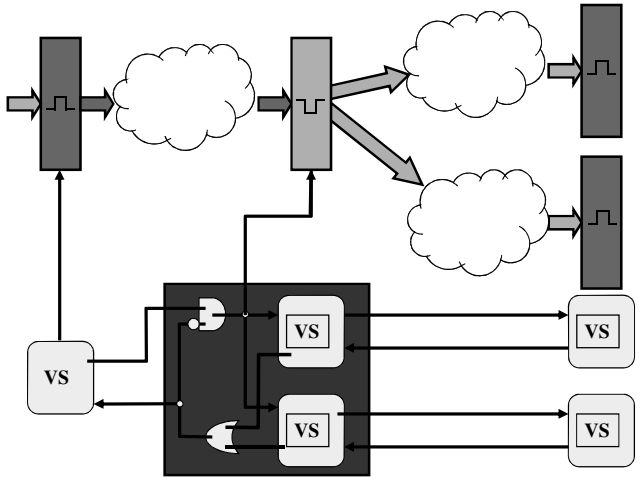
(Lazy) Fork



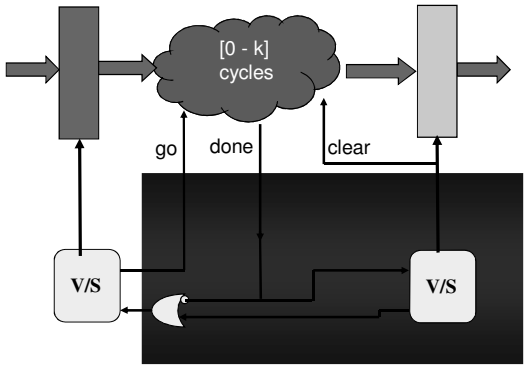
Eager Fork



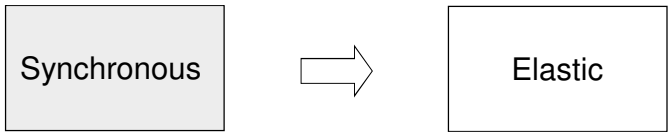
Eager fork (another implementation)

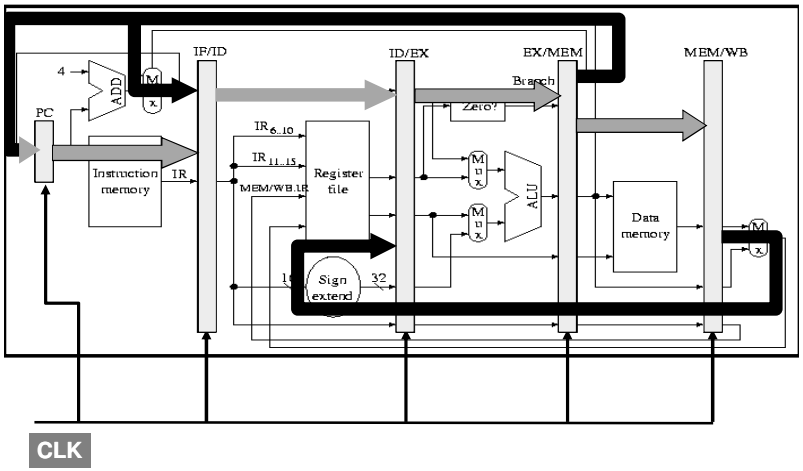


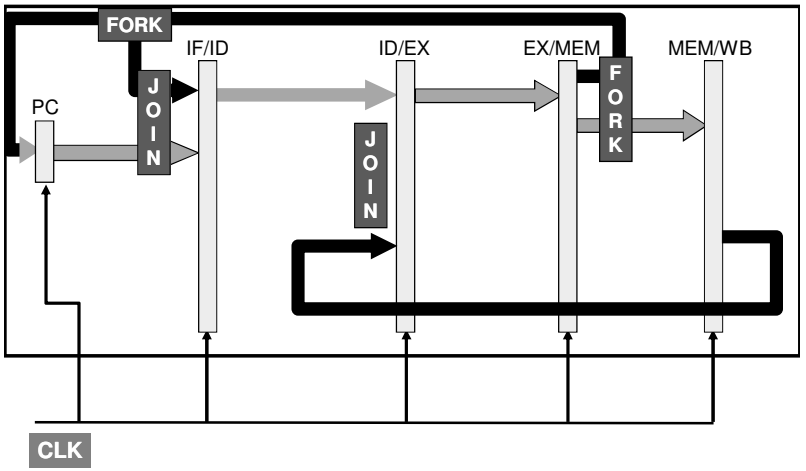
Variable Latency Units (to be changed)

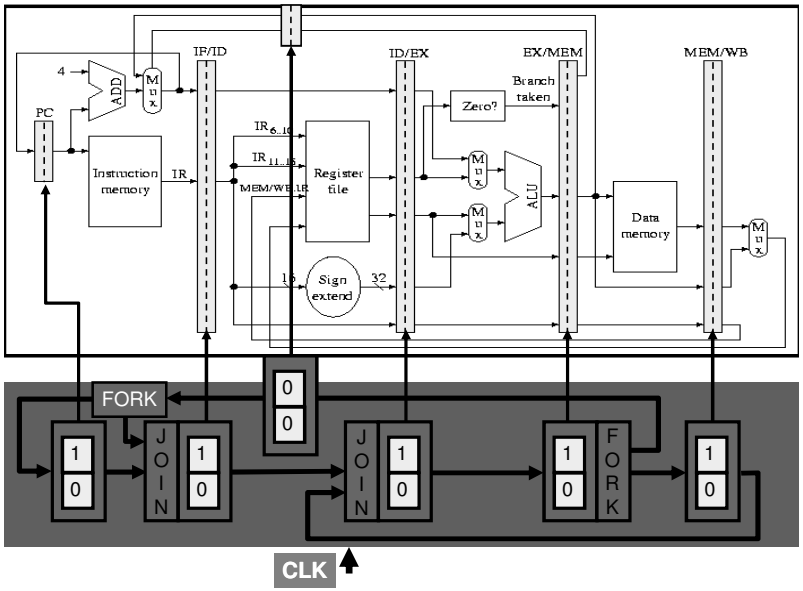


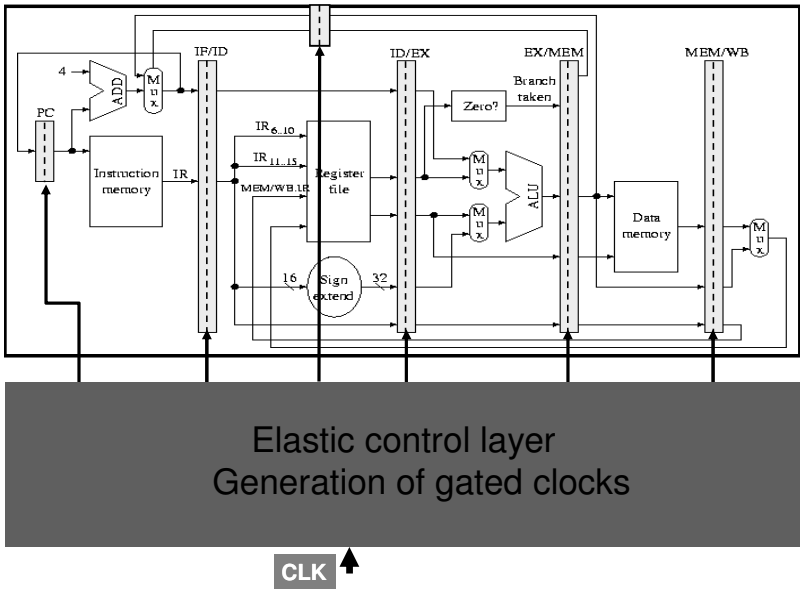
Elasticization





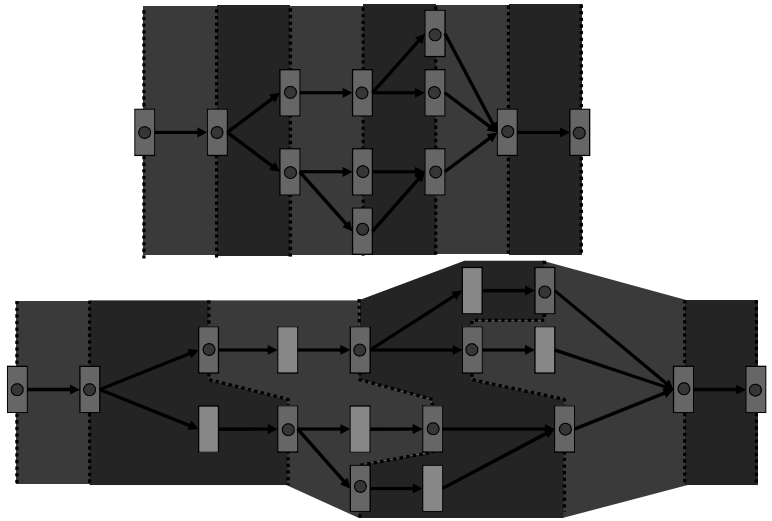




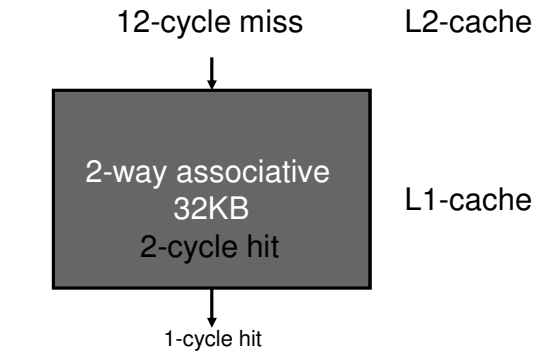


Micro-architectural opportunities

Circuit vs. μarchitectural cycles

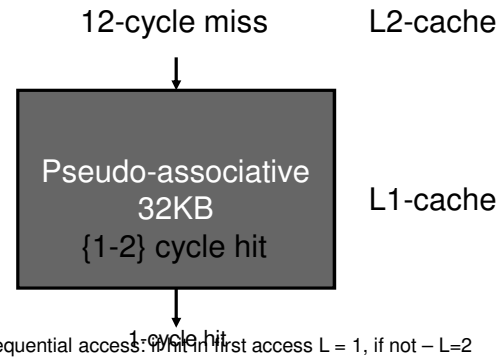


Variable-latency cache hits



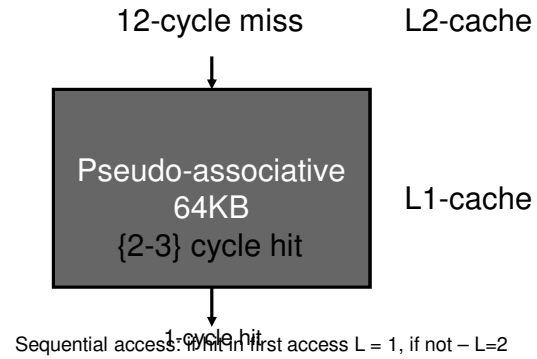
suggested by Joel Emer for ASIM experiment

Variable-latency cache hits



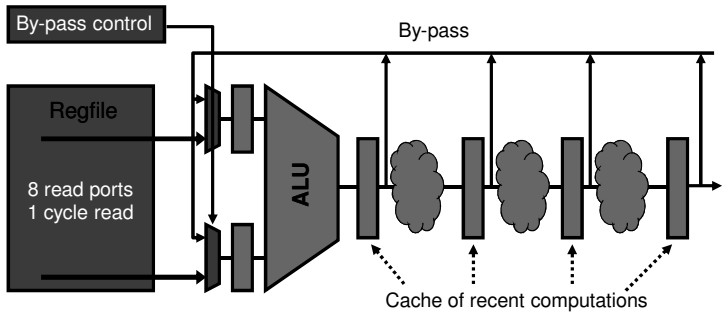
Sequential access: if hit in first access L = 1, if not – L=2
Trade-off: faster, or larger, or less power cache

Variable-latency cache hits



Sequential access: if hit in first access L = 1, if not – L=2
Trade-off: faster, or larger, or less power cache

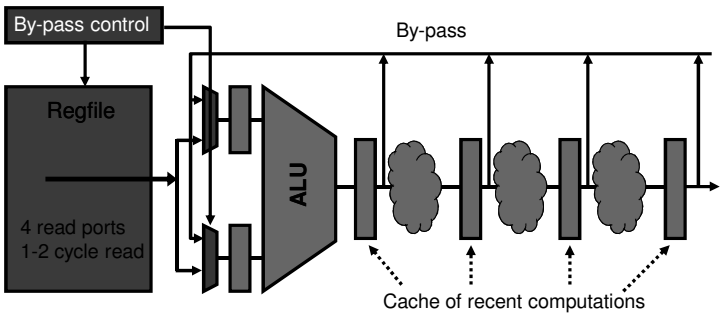
Variable-latency RF with less ports




```
LD   R2, A(R5)
ADD  R1, R2, R3   ← 1 by-pass, 1 read port
MUL  R4, R1, R6   ← 1 by-pass, 1 read port
CMP  R4, #100     ← 1 by-pass
```

4-way superscalar or 4 threads assumed, 1 way shown

Variable-latency RF with less ports

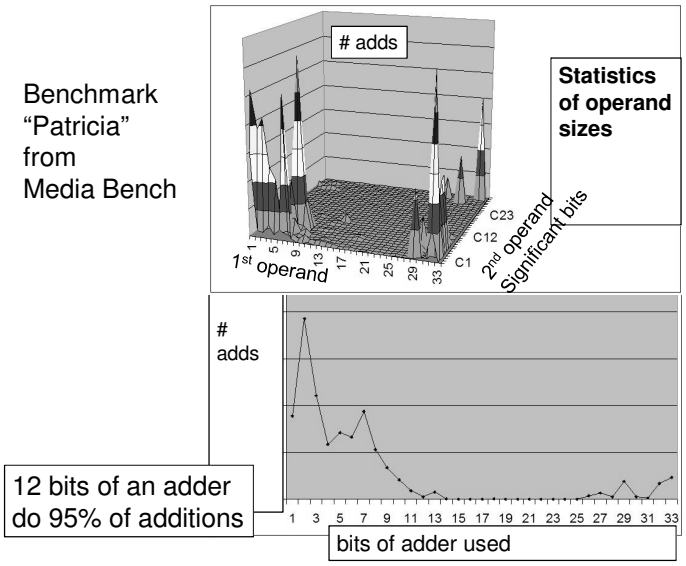



```
LD   R2, A(R5)
ADD  R1, R2, R3   <-- 1 by-pass, 1 read port
MUL  R4, R1, R6   <-- 1 by-pass, 1 read port
CMP  R4, #100     <-- 1 by-pass
```

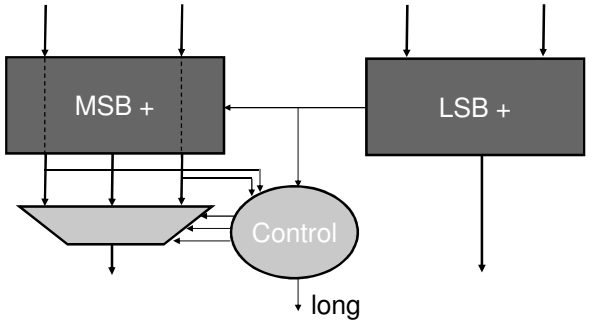
Variable-latency ALUs

- In most of the ADD/SUB operations, only few LSBs are used. The rest are merely for sign extension.
- Use the idea of telescopic units:
  - 1-cycle addition for 16 bits and sign extension
  - 2 or more cycles for 64-bit additions (rare case)
  - maybe there is no need for CLA adders …
  - or do all additions with 16-bit adders only

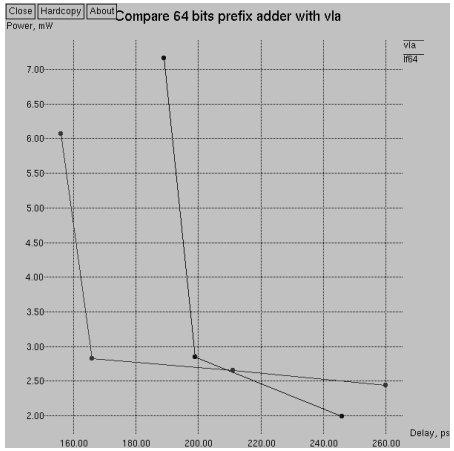
Benchmark "Patricia" from Media Bench



Variable Latency Adder



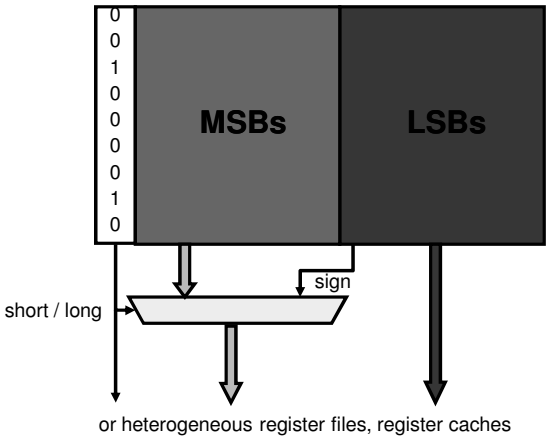
Power-delay [preliminary]



Pre-compute and tag size information

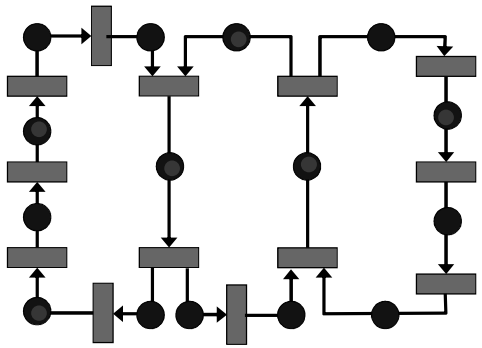
… and select functional unit according to the size of the data

Partitioned register file



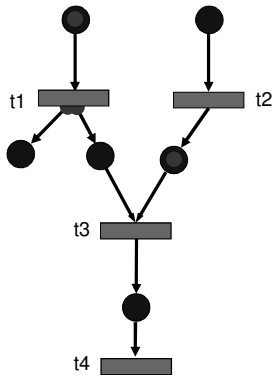
Reminder:
Petri Nets and Marked Graphs

Petri nets



- Petri net
  - Places {P}
  - Transitions {T}
  - Arcs
  - Tokens

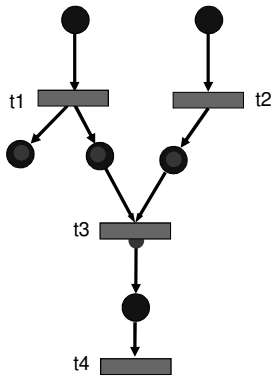
Petri nets. Token game



Enabling Rule:
- A transition is enabled if all its input places are marked
- Enabled transition can fire at any time

Firing Rule:
- One token is removed from every input place
- One token is added to every output place
- Change of marking is atomic

Petri nets. Token game

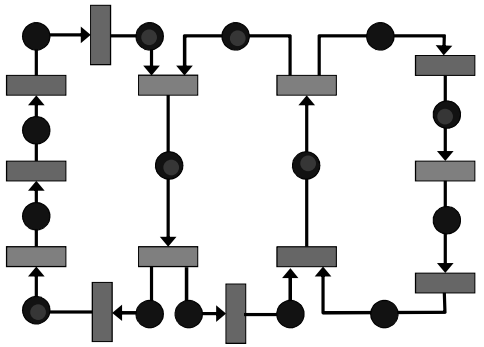


Enabling Rule:
- A transition is enabled if all its input places are marked
- Enabled transition can fire at any time

Firing Rule:
- One token is removed from every input place
- One token is added to every output place
- Change of marking is atomic

Timed Petri nets

- Assign a delay (δ) to every transition
  - An enabled transition fires δ time units after enabling

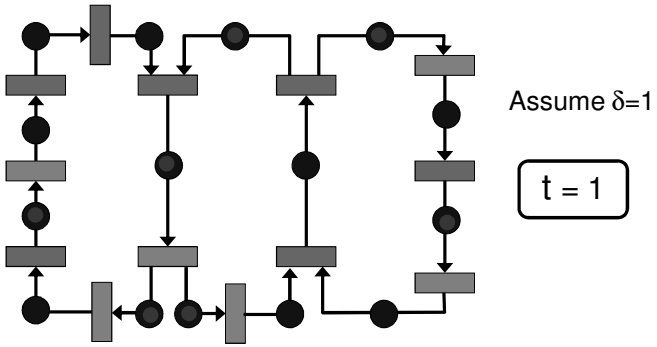


Assume δ=1

t = 0

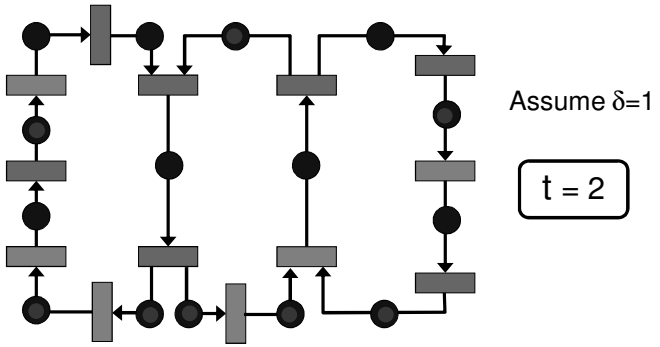
Timed Petri nets

- Assign a delay (δ) to every transition
  - An enabled transition fires δ time units after enabling



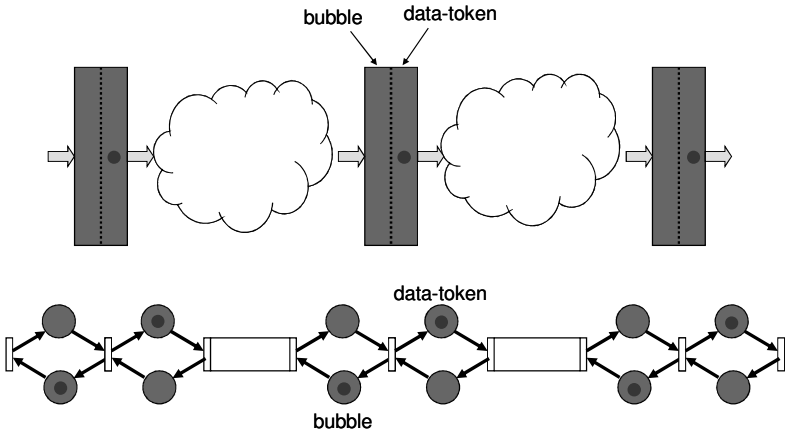
Timed Petri nets

- Assign a delay (δ) to every transition
  - A transition with marked input places will fire after δ time units

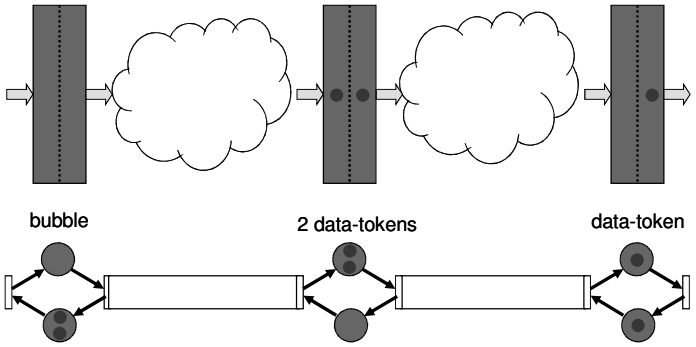


Marked Graph models
of elastic systems

Modelling elastic control with Petri nets

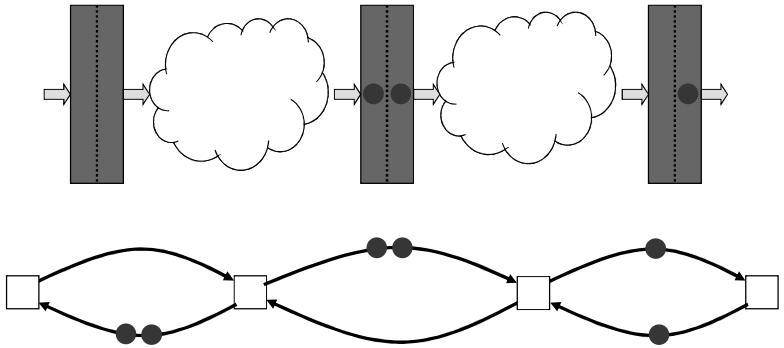


Modelling elastic control with Petri nets

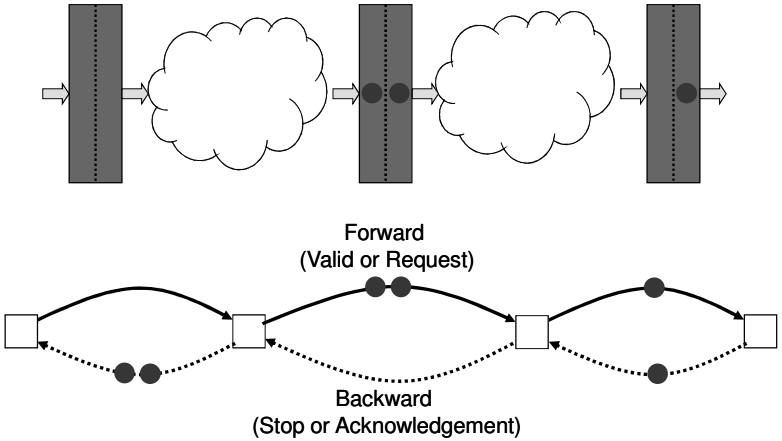


Hiding internal transitions of elastic buffers

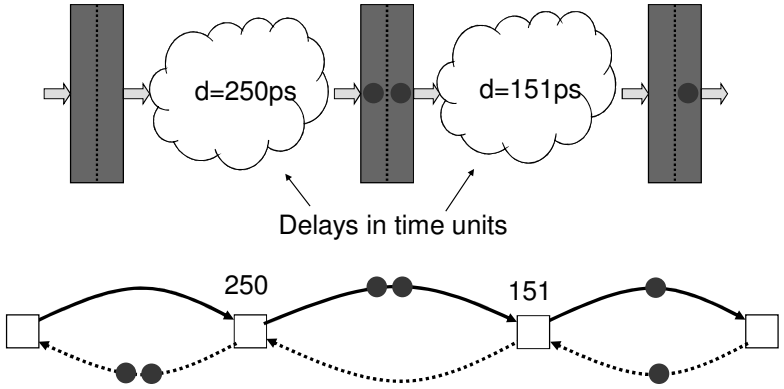
Modelling elastic control with Marked Graphs



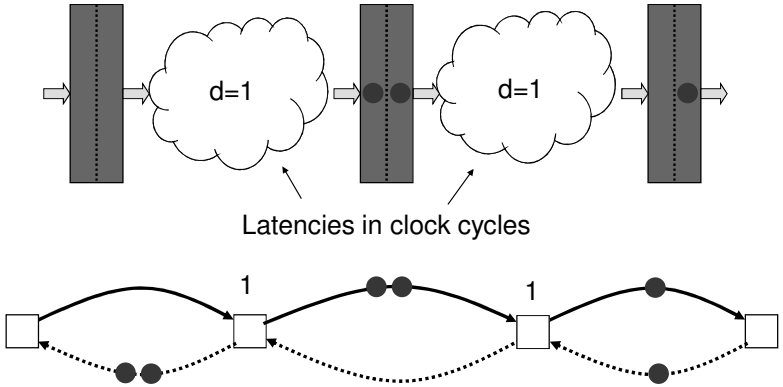
Modelling elastic control with Marked Graphs



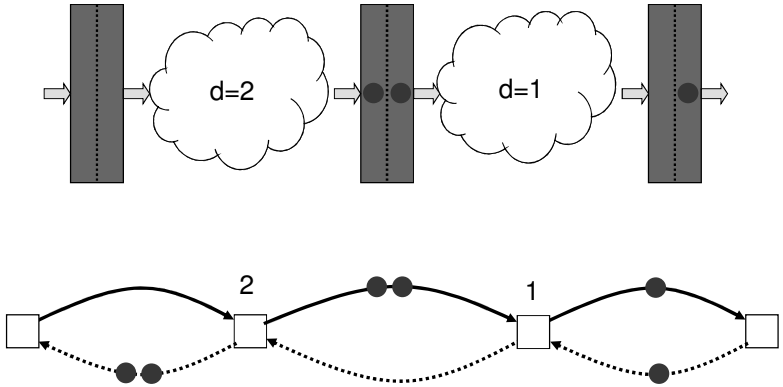
Elastic control with Timed Marked Graphs. Continuous time = asynchronous



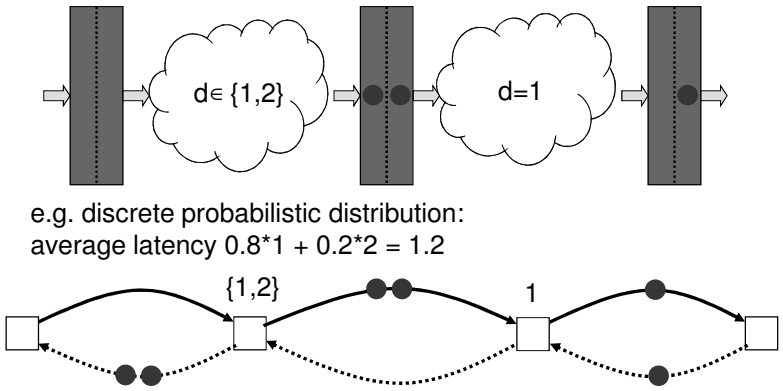
Elastic control with Timed Marked Graphs. Discrete time = synchronous elastic



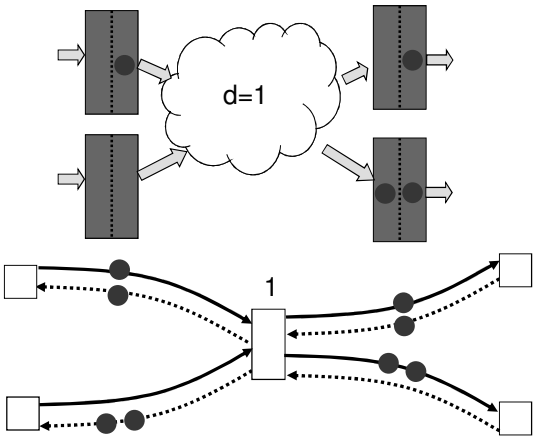
Elastic control with Timed Marked Graphs. Discrete time. Multi-cycle operation



Elastic control with Timed Marked Graphs. Discrete time. Variable latency operation



Modeling forks and joins

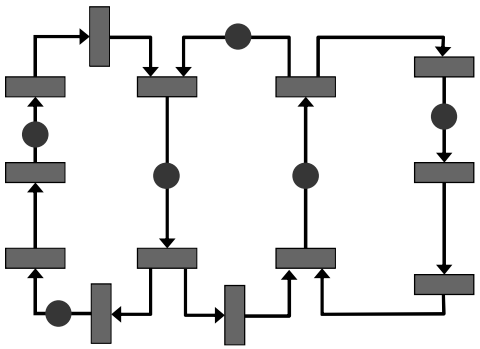


Elastic Marked Graphs

- An Elastic Marked Graph (EMG) is a Timed MG such that for any arc *a* there exists a complementary arc *a'* satisfying the following condition
  $\bullet a = a' \bullet$ and $\bullet a' = a \bullet$
- Initial number of tokens on *a* and *a'* $(M_0(a)+M_0(a'))$ = capacity of the corresponding elastic buffer
- Similar forms of "pipelined" Petri Nets and Marked Graphs have been previously used for modeling pipelining in HW and SW (e.g. Patil 1974; Tsirlin, Rosenblum 1982)

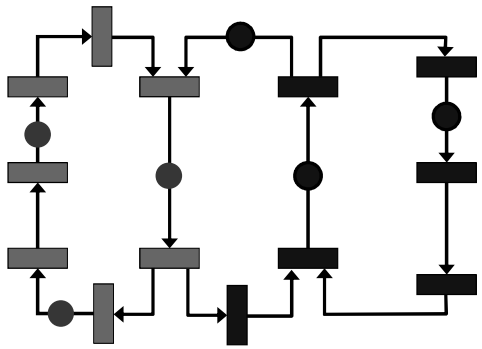
Performance analysis on Marked Graphs

Performance



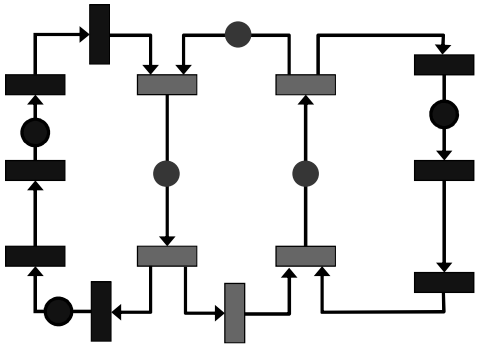
Th = operations / cycle

Performance



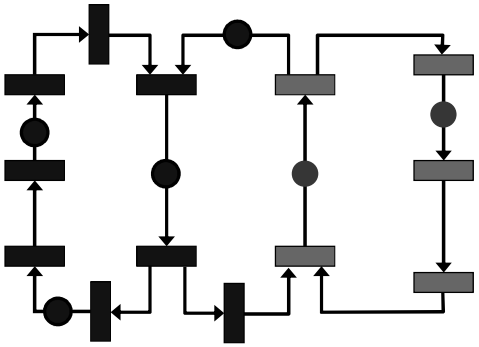
Th = 3 / 7

Performance



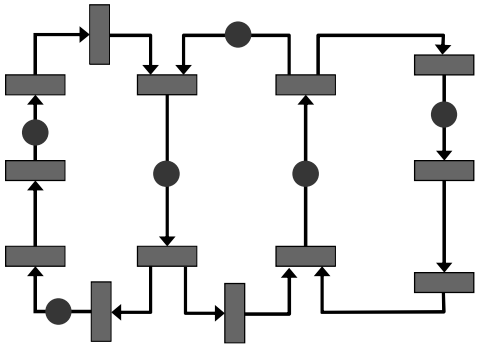
Th = 3 / 5

Performance



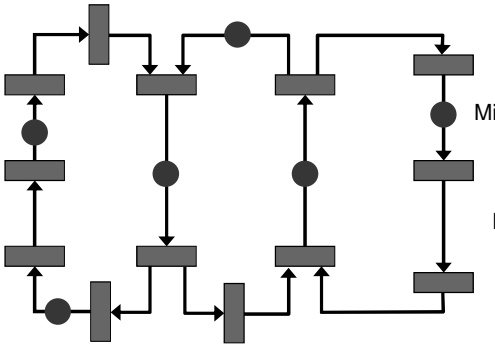
Th = 2 / 5

Performance



Th = min (0.43, 0.6, 0.4)

Performance



Minimum mean-weight cycle
(Karp 1978)

Many efficient algorithms
(some reviewed in
Dasdan,Gupta 1998)

*Th = min (0.43, 0.6, **0.4**)*

II

- Early evaluation
- Dual Marked Graphs
- Implementing early evaluation
- Performance analysis

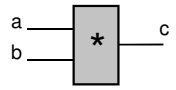
Early evaluation

- Naïve solution: introduce choice places
  - issue tokens at choice node only into one (some) relevant path
  - problem: tokens can arrive to merge nodes out-of-order
    later token can overpass the earlier one
- Solution: change enabling rule
  - early evaluation
  - issue negative tokens to input places without tokens, i.e. keep the same firing rule
  - Add symmetric sub-channels with negative tokens
  - Negative tokens kill positive tokens when meet
- Two related problems:
  Early evaluation and Exceptions (how to kill a data-token)

Examples of early evaluation

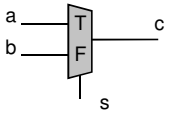
Goal: Improve system performance and power

MULTIPLIER



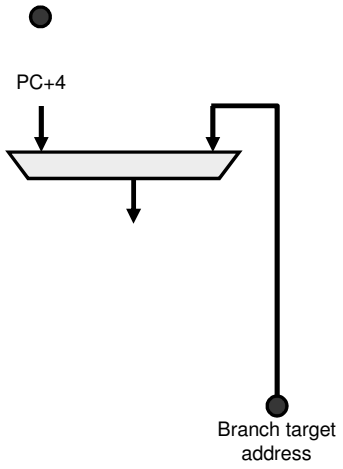
if a = 0 then c := 0 -- don't wait for b

MULTIPLEXOR



if s = T then c := a -- don't wait for b
else c := b -- don't wait for a

Example: next-PC calculation



Related work

- Petri nets
  - Extensions to model OR causality [Kishinevsky et al. 1994, Yakovlev et al. 1996]
- Asynchronous systems
  - Reese et al 2002: Early evaluation
  - Brej 2003: Early evaluation with anti-tokens
  - Ampalan & Singh 2006: preemption using anti-tokens

Dual Marked Graphs

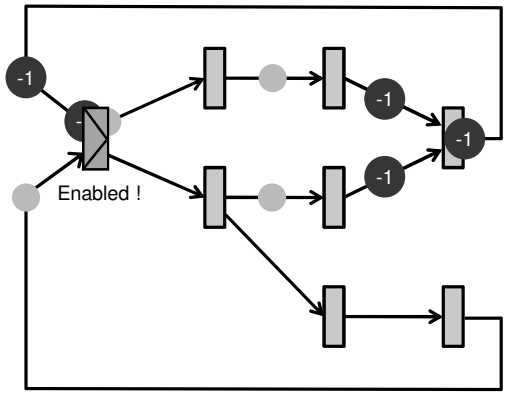
Dual Marked Graph

- Marking: Arcs (places) –> **Z**
- Some nodes are labeled as early-enabling
- Enabling rules for a node:
  - Positive enabling: $M(a) > 0$ for every input arc
  - Early enabling (for early enabling nodes): $M(a) > 0$ for some input arcs
  - Negative enabling: $M(a) < 0$ for every output arc
- Firing rule: the same as in regular MG

Dual Marked Graphs

- Early enabling is only defined for nodes labeled as early-enabled. Models computations that can start before all the incoming data available
- Early enabling can be associated with an external guard that depends on data variables (e.g., a select signal of a multiplexor)
- In DMG actual enabling guards are abstracted away
- Anti-token generation: When an early enabled node fires, it generates anti-tokens in the predecessor arcs that had no tokens
- Anti-token propagation counterflow: When negative enabled node fires, it propagates the anti-tokens from the successor to the predecessor arcs

Dual Marked Graph model



Properties of DMGs

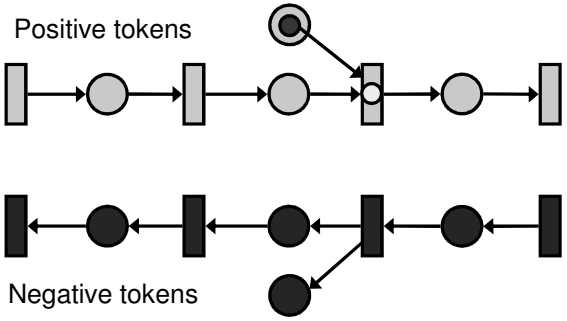
- Firing invariant: Let node *n* be simultaneously positive (or early) and negative enabled in marking *M.* Let M1 be the result of firing *n* from *M* due to positive (or early) enabling. Let M2 be the result of firing *n* from *M* due to negative enabling. Then, M1 = M2.
- Token preservation. Let *c* be a cycle of a strongly connected DMG. For every reachable marking M, $M(c) = M_0(c)$.
- Liveness. A strongly connected DMG is live iff for every cycle *c:* $M(c) > 0$.
- Repetitive behavior. In a SC DMG: a firing sequence *s* from *M* leads to the same marking iff every node fires in *s* the same number of times.
- DMGs have properties similar to regular MGs

Passive anti-token

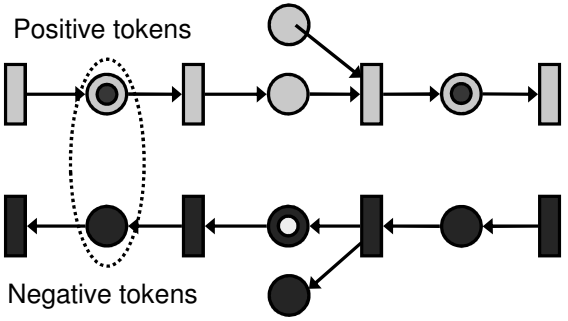
- Passive DMG = version of DMG without negative enabling
- Negative tokens can only be generated due to early enabling, but cannot propagate
- Let *D* be a DMG and D_p be a corresponding passive DMG. If environment (consumers) never generate negative tokens, then
  throughput (*D*) = throughput (D_p)
  - If capacity of input places for early enabling transitions is unlimited, then active anti-tokens do not improve performance
  - Active anti-tokens reduce activity in the data-path (good for power reduction)

Implementing early enabling

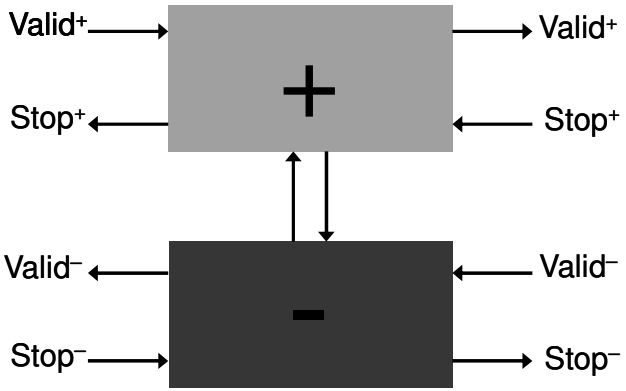
How to implement anti-tokens ?



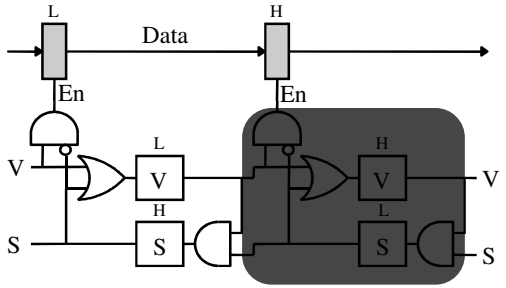
How to implement anti-tokens ?



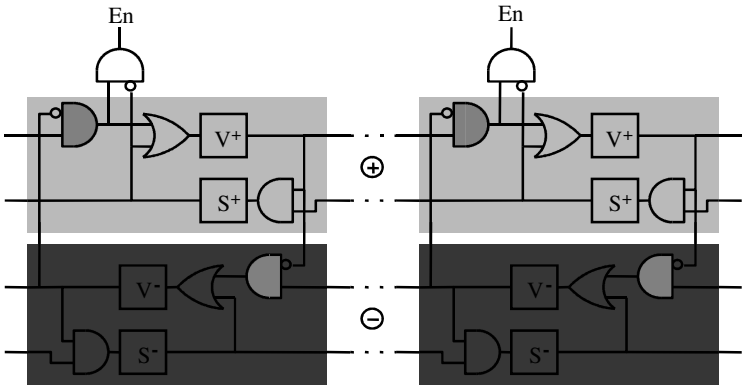
How to implement anti-tokens ?



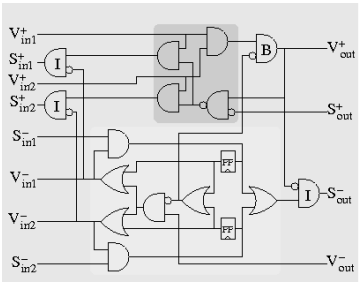
Controller for elastic buffer



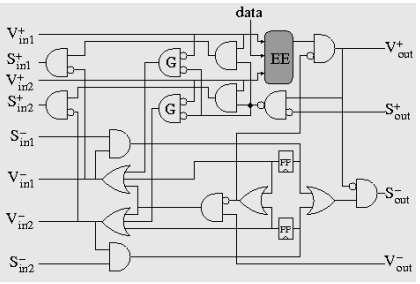
Dual controller for elastic buffer



Dual fork/join and early join

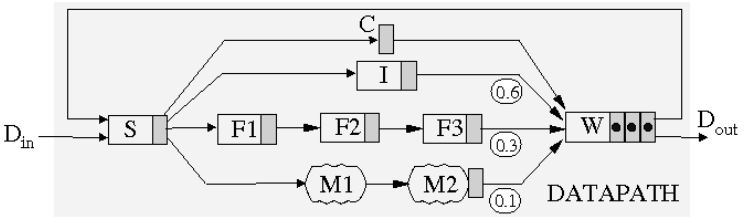


Dual fork/join



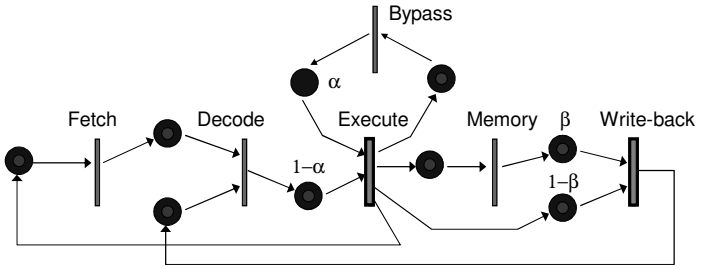
Join with early evaluation

Example



Evaluation	Throughput
No early evaluation	0.277
Passive anti-tokens M2 → W	0.280
Passive anti-tokens F3 → W	0.387
Active anti-tokens	0.400

DLX processor model with slow bypass



Throughput: *Th = operations / cycle*

System performance

Th=0.5

Applying early evaluation on "Execution" and "Write-back"

Th = 0.7 (α=0.3; β=0.3)

Conclusions

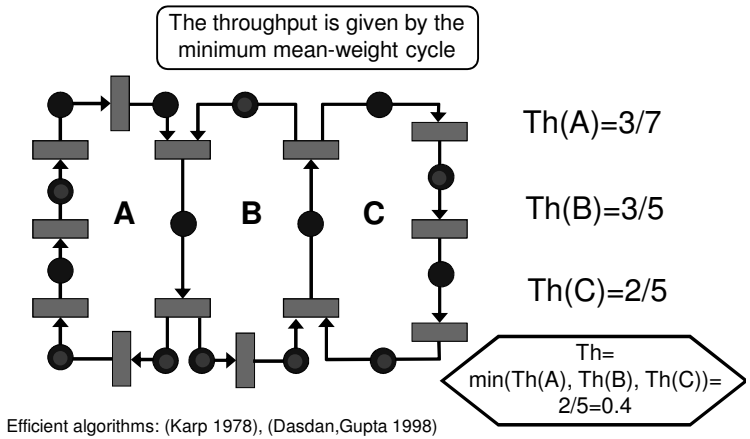
- Early evaluation can increase performance beyond the min cycle ratio
- The duality between tokens and anti-tokens suggests a clean and effective implementation

Performance analysis with early evaluation

(joint work with Jorge Júlvez)

Reminder: Performance analysis of Marked graphs

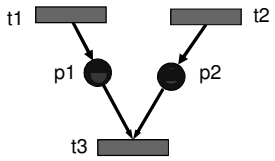
Th = operations / cycle = number of firings per time unit



Efficient algorithms: (Karp 1978), (Dasdan,Gupta 1998)

Marked graphs. Performance analysis

The throughput can also be computed by means of linear programming



$$th = \min(\overline{m}_{p1}, \overline{m}_{p2})$$

Average marking

$$\overline{m}_p = \lim_{t\to\infty} \frac{1}{t} \int_0^t m_p(\tau) d\tau$$

Throughput

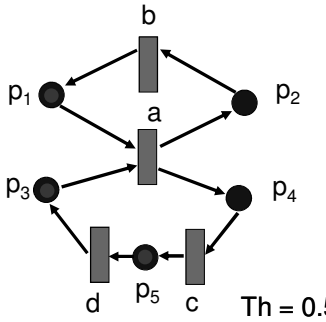
$$th = \min_p \overline{m}_p$$

Marked graphs. Performance analysis

The throughput can also be computed by means of linear programming

$$th = \min_p \overline{m}_p$$

max th



reachability

$\overline{m}_{p1} = 1 + t_b - t_a$
$\overline{m}_{p2} = 0 + t_a - t_b$
$\overline{m}_{p3} = 1 + t_d - t_a$
$\overline{m}_{p4} = 0 + t_a - t_c$
$\overline{m}_{p5} = 1 + t_c - t_d$

th constraints

$th \le \overline{m}_{p2}$ // transition b
$th \le \overline{m}_{p4}$ // transition c
$th \le \overline{m}_{p5}$ // transition d
$th \le \min(\overline{m}_{p1}, \overline{m}_{p3})$ // transition a

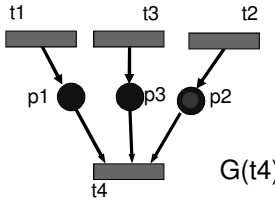
Th = 0.5

[Campos, Chiola, Silva 1991]

GMG = Multi-guarded Dual Marked Graph

- Refinement of passive DMGs
- Every node has a set of guards
- Every guard is a set of input arcs (places)
- *Simple* transition has one guard with all input places

Example:



G(t4)={{t1,t3},{t2,t3}}

Multi-guarded Dual Marked Graph

- Execution of transitions:
  - At the initial marking and each time *t* fires one of guards g*i* from *G(t)* is non-deterministically selected
    - Selection is persistent (cannot change between firings of *t*)
    - Accurate non-deterministic abstraction of the early evaluation conditions (e.g. multiplexor select signals)
  - *t* is enabled when for every place *p* in selected *gi: M(p)* > 0
  - enabled *t* can fire (regular firing rule)
- Single-server semantics: *no multiple-instances of the same transition can fire simultaneously*
  - Abstraction for systems that communicate through FIFO channels

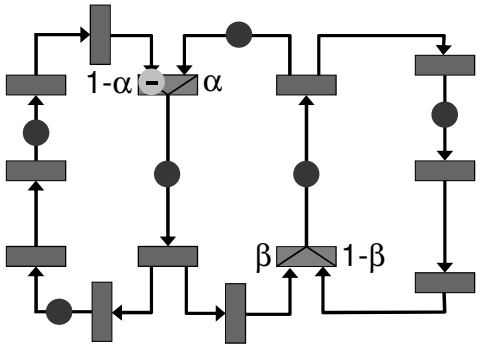
Timed GMG

- Every transition is assigned a non-negative delay $\delta(t)$
  - $\delta(t)=1$ *unless specified otherwise*
- Every guard *g* of every guarded transition is assigned *a strictly positive probability p(g)* such that for every *t:*

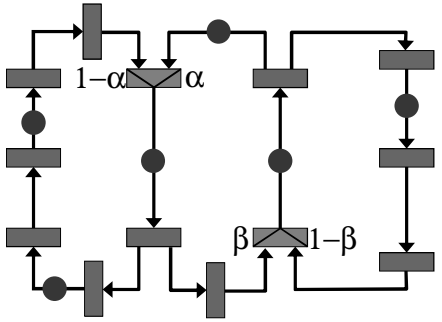
$$\sum_{g \in G(t)} p(g) = 1$$

- Guard selection for every transition is non-deterministic, but respects probabilities in the infinite executions
- Probabilities assumed to be independent (generous abstraction)
- Firing of transition *t* takes $\delta(t)$ time units, from the time it becomes enabled until the firing is completed

Early evaluation



Early evaluation



(0.43) (0.60) (0.40)

β \ α	0.0	0.2	0.4	0.6	0.8	1.0
0.0	0.40	0.40	0.40	0.40	0.40	0.40
0.2	0.42	0.42	0.42	0.42	0.43	0.43
0.4	0.43	0.44	0.44	0.45	0.45	0.45
0.6	0.43	0.44	0.45	0.47	0.48	0.49
0.8	0.43	0.44	0.46	0.48	0.51	0.54
1.0	0.43	0.44	0.46	0.49	0.54	0.60

Timed GMG

{Places, Transitions, δ, Prob} **Throughput?**

Marked graphs with early evaluation

stochastic dynamic system

Alternatives to compute the throughput:

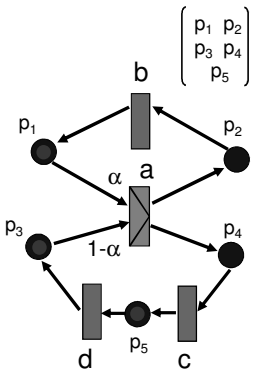
- Simulation
- Markov chain
- Linear programming

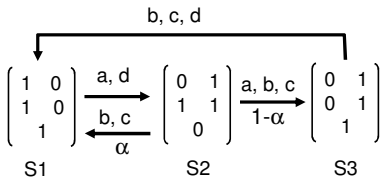
Throughput

$$Th = \lim_{\tau \to \infty} \sigma(\tau)/\tau$$

- τ – time, σ(τ) – firing vector
- This limit exists for every timed GMG
- It is the same for all transitions!

Markov chains





Solve Markov chain:

S2 = S1; S3 = (1-α)S2; S1 + S2 + S3 = 1;
Th = S1 + (1-α)S2

$$Th = (2 - \alpha) / (3 - \alpha)$$

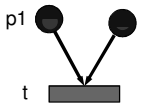
State explosion problem!

Linear programming formulation

Average marking: $\overline{m}_p$

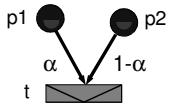
max th

$$\overline{m} = m_0 + t_{in} - t_{out}$$ reachability



th constraints

$$\delta(t) * th \le \min(\overline{m}_{p1}, \overline{m}_{p2}) \text{ for all “classical” t}$$

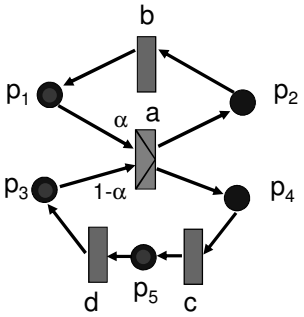


th constraints

$$\delta(t) * th = \alpha\, \overline{m}_{p1} + (1-\alpha)\, \overline{m}_{p2} \text{ for all “early” t}$$

Linear programming. Example

max th



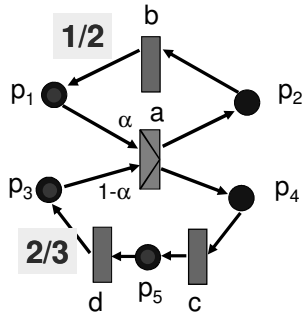
$$\overline{m}_{p1} = 1 + t_b - t_a$$
$$\overline{m}_{p2} = 0 + t_a - t_b$$
$$\overline{m}_{p3} = 1 + t_d - t_a$$
$$\overline{m}_{p4} = 0 + t_a - t_c$$
$$\overline{m}_{p5} = 1 + t_c - t_d$$

$$th \le \overline{m}_{p2}$$
$$th \le \overline{m}_{p4}$$
$$th \le \overline{m}_{p5}$$

$$th = \alpha\, \overline{m}_{p1} + (1-\alpha)\, \overline{m}_{p3}$$

$$Th = (2 - \alpha) / (3 - \alpha)$$

Averaging cycle throughput or cycle times does not work



$$Th = (2 - \alpha) / (3 - \alpha)$$

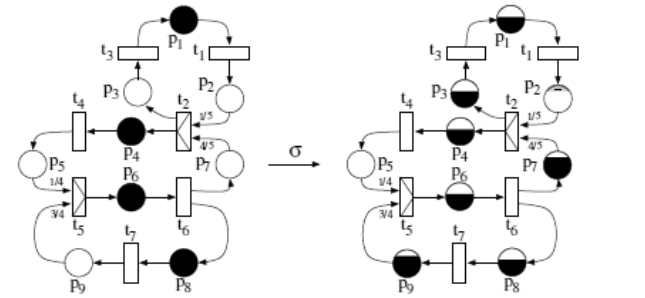
Averaging throughput of individual cycles

$$Th' = \alpha\, 1/2 + (1- \alpha)\, 2/3 = (4 - \alpha) / 6$$

Averaging effective cycle times of individual cycles

$$1/Th'' = 2\alpha + (1- \alpha)\, 3/2 = (3 + \alpha) / 2$$
$$Th'' = 2/(3+ \alpha)$$

Example



Initial marking Average (steady state) marking

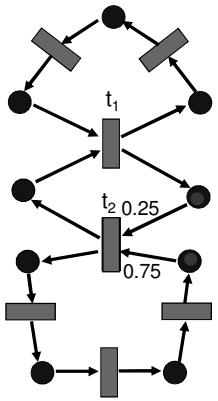
Linear programming

In general the LP yields a throughput upper bound

Particular cases of exact throughput:

- No early joins (i.e. MGs)
- All joins are early evaluation

Throughput estimation



- Throughput estimation:
  1. Up = throughput obtained with LP
  2. Low = throughput without early enabling
  3. Th = (Up + Low)/2

Results

Circuits from MCNC

2-input gates

a latch for each gate

75% tokens, 25% bubbles

25% muxes

Random select probability

Name	Nodes	Edges	MG=Low	Real	Up	ΔTh	Err
s27			0.333	0.333	0.333	0 %	
S208			0.500	0.571	0.594	14 %	
s298			0.091	0.120	0.129	32 %	
s349			0.333	0.333	0.333	0 %	
s382			0.250	0.284	0.294	14 %	
s386			0.400	0.400	0.400	0 %	
s400			0.400	0.438	0.470	10 %	
S444			0.200	0.261	0.287	31 %	
S510			0.167	0.167	0.167	0 %	
S526			0.333	0.333	0.333	0 %	
S641			0.333	0.393	0.432	18 %	
S713			0.250	0.333	0.333	33 %	
S820			0.143	0.201	0.230	41 %	
S832			0.286	0.310	0.342	8 %	
S953			0.286	0.295	0.333	3 %	
S1423			0.100	0.184	0.189	84 %	
S1488			0.188	0.236	0.271	26 %	
S1494			0.154	0.222	0.277	44 %	
S5378			0.235	0.250	0.250	6 %	
s9234			0.200	0.219	0.248	10 %	

Summary

- Early evaluation to improve system throughput
  - Evaluate expressions as soon as possible
  - Generate antitokens to erase “don’t care” bits
- Analytical model to estimate the throughput
  - Useful for architectural exploration
    - Which muxes must have early evaluation ?
    - Where do we put our by-passes ?
  - Faster than simulation
  - Simulation can be used at later design stages

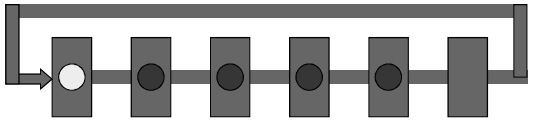
III

- Optimization
  - Slack matching and buffer sizing
  - Retiming and recycling
  - Clustering controllers
- Correctness
  - Theory of elastic machines
  - Formal verification

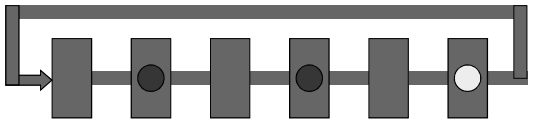
Buffer sizing

(joint work with Dmitry Bufistov)

How many bubbles do we need?

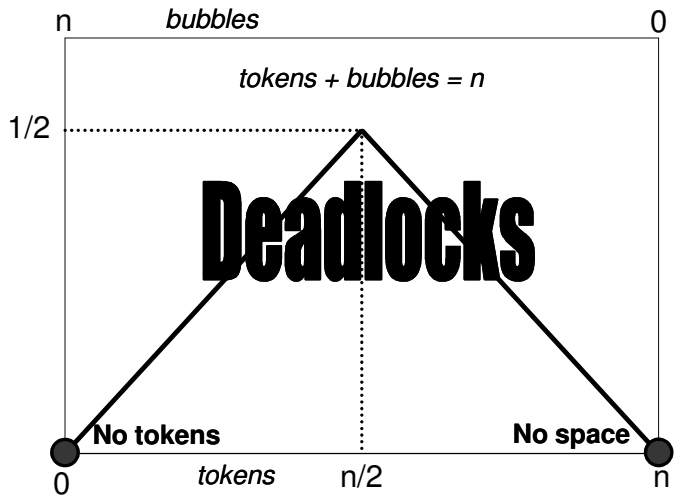


$O(n^2)$ *cycles*



$O(n)$ *cycles*

Throughput of an *n*-stage ring



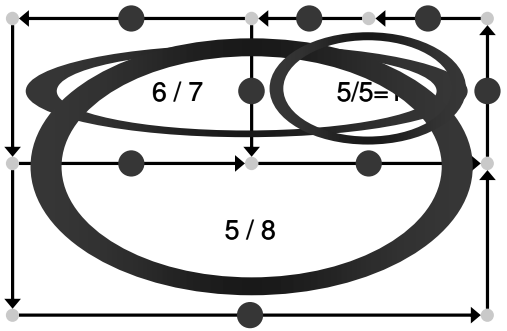
What flexibility do we have ?

- Number of tokens on a loop cannot be easily changed (inherent to the computation)
- Bubbles can always be added (as many as necessary), but may decrease throughput
- Buffer sizes can always be increased (provided forward latency of the buffer does not change)
- Tokens determine the maximum achievable throughput (assuming infinite buffer sizes)

Optimization techniques

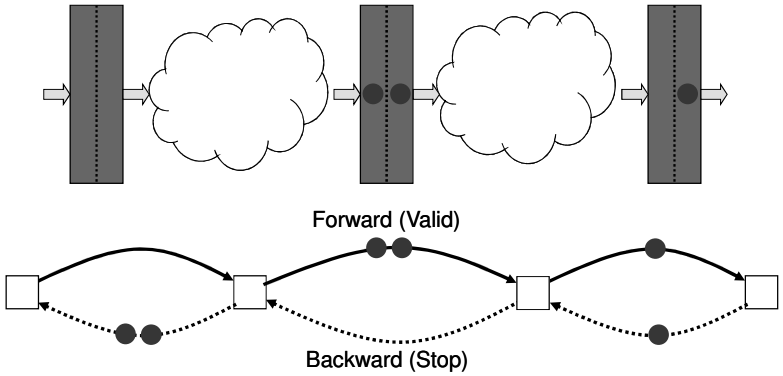
- Buffer sizing: select optimal capacity of elastic buffers without increasing forward latency for propagating data-tokens
- Slack matching: insert additional empty elastic buffers
  - increases buffer capacity
  - but, typically increases forward latency as well
  - also called recycling in the context of synchronous elastic (latency-tolerant) designs

Buffer optimization

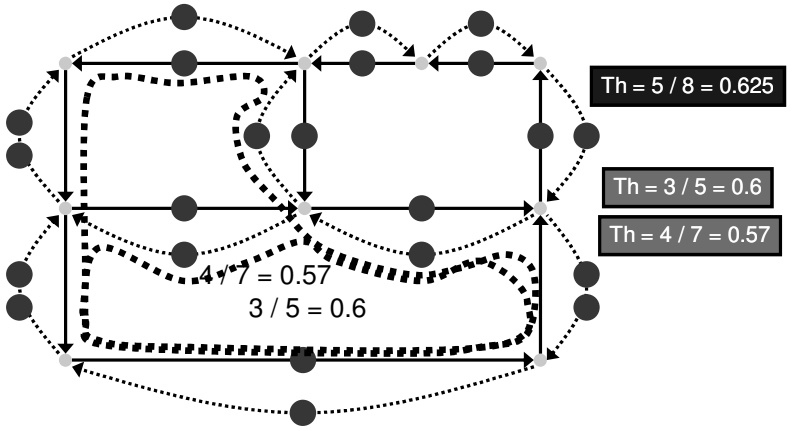


Th = 5 / 8 = 0.625

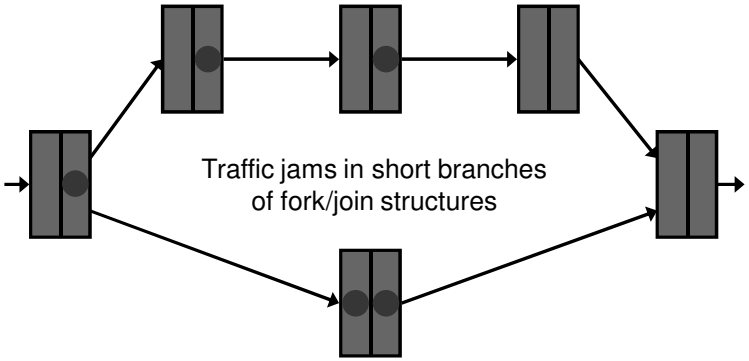
Buffer optimization



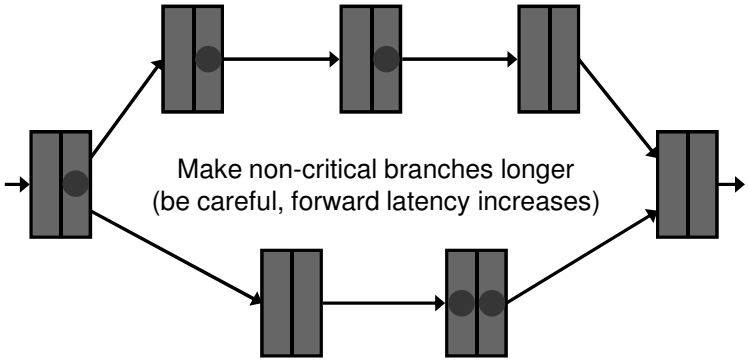
Buffer optimization



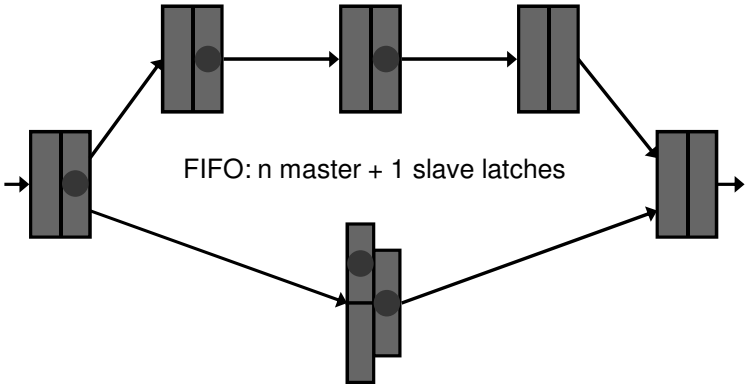
Why ?



Solution 1: slack matching/recycling



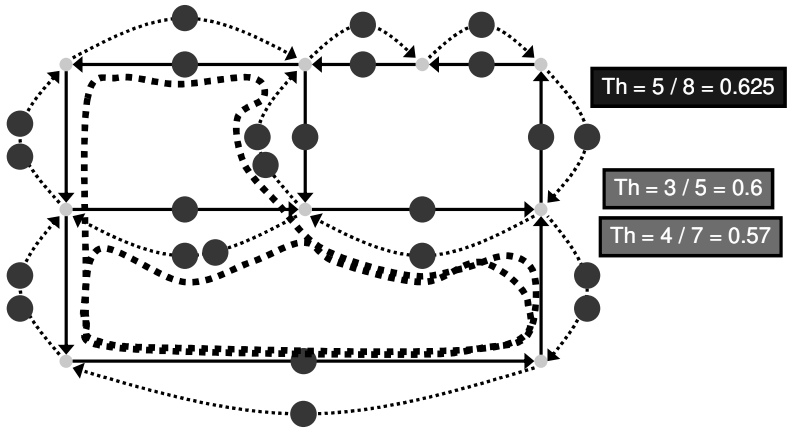
Solution 2: increase buffer size



Slack matching vs. buffer capacity

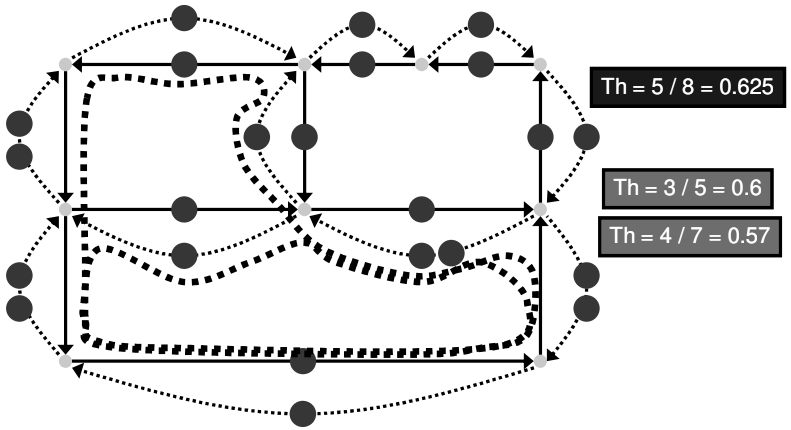
- Not equivalent (slack matching cannot always achieve the forward latency, while buffer capacity can)
- Slack matching is a well-studied problem in asynchronous design
- Slack matching = inc buffer capacity + split

Buffer optimization



Increase buffer capacity = Put token in backward edge

Buffer optimization



Increase buffer capacity = Put token in backward edge

Buffer sizing

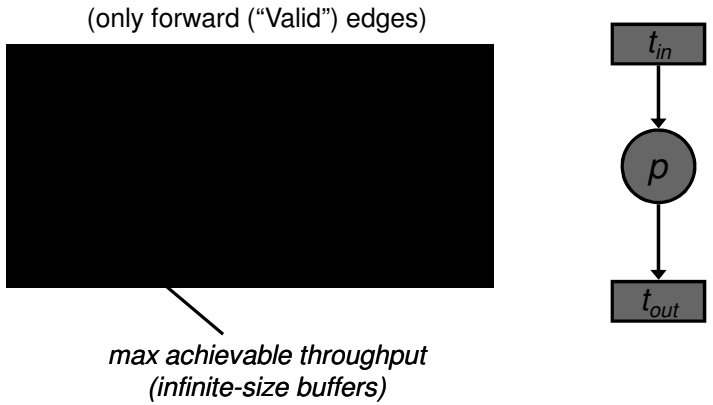
- Find min possible increase in buffer sizes such that the throughput is equal to the throughput of a system with infinite size buffers
- Combinatorial problem
- We found an exact ILP formulation, but …
- ILP is exponential
- Can we do better (polynomial time) ?

Buffer sizing is NP-complete

- NP-hardness: reduction of
  "min edges that cut all cycles in a digraph"
  to buffer sizing
- NP: Checking validity of solution can be done in polynomial time (e.g. Karp's algorithm)
- Therefore,

  No polynomial algorithm exists, unless P = NP

LP performance model



ILP model for buffer sizing

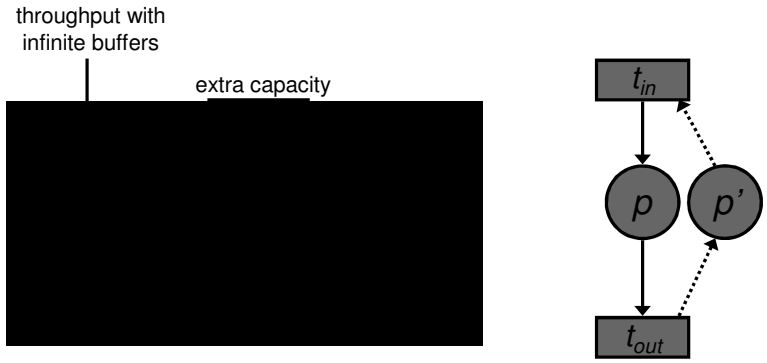


Table of results

Circuit	|V|	|E|	Th	Max Th	ΔTok	CPU (sec)	Mem (Mb)
s1423	484	942	0.33	0.33	0	<1	81
s1488	321	1662	0.5	0.5	0	<1	95
s1494	341	1775	0.5	0.5	0	1	108
s208	**36**	**100**	**0.5**	**1**	**26**	**1**	**1**
s27	**31**	**78**	**0.5**	**0.75**	**18**	**<1**	**1**
s298	823	7154	0.5	0.5	0	5	946
s349	**139**	**241**	**0.5**	**0.6**	**3**	**<1**	**7**
s386	86	339	0.5	0.5	0	<1	7
s400	119	273	0.33	0.33	0	<1	7
s444	132	298	0.33	0.33	0	<1	8
s510	149	571	0.5	0.5	0	<1	16
s526	145	382	0.33	0.33	0	<1	11
s5378	**1138**	**2484**	**0.42**	**0.55**	**30**	**4708**	**500**
s641	182	298	0.5	0.67	6	<1	11
s713	208	350	0.42	0.5	1	<1	15
s820	183	919	0.5	0.5	0	<1	31
s832	191	972	0.5	0.5	0	<1	34
s9234	1023	1992	0.25	0.25	0	<1	350
s953	**373**	**704**	**0.45**	**0.64**	**10***	**>21600**	**60**
s38417	**8315**	**16440**	**0.25**	**0.33**	**-**	**-**	**>2Gb**

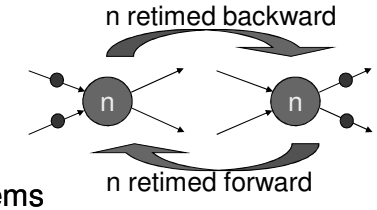
* - Non optimal integral solution with time limit 120 seconds

Retiming and Recycling

(joint work with Dmitry Bufistov and Sachin Sapatnekar)

Retiming

- Retiming: moving registers across combinational blocks or (equivalently) moving combinational blocks across registers
  - forward retiming
  - backward retiming



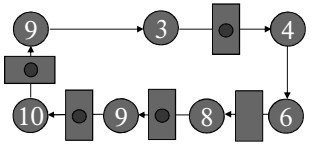
- Retiming in elastic systems
  - all registers participating in the retiming move should be labeled with the same number of data-tokens
  - use of negative tokens can remove the above constraint (will not discuss here)

Recycling

- Recycling: insert (or remove) empty elastic buffers (empty registers for short) on any edge
  - possible only in elastic systems
- We will ignore initialization and consider only steady state behavior
  - Initialization to an equivalent state almost always possible, but may require extra logic

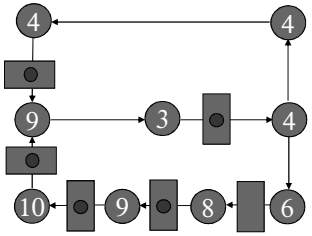
Effective Cycle Time

- Cycle time: c = max {path delay between registers}
- Throughput: Θ = min {tokens/cycle}
  Was formally defined before
- Effective cycle: **C** = c / Θ



c=12 Θ=4/5 **C** =15

R&R graph (RRG)



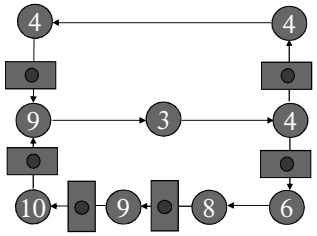
10 combinational block with a delay of 10 units

register (EB) with one data token

empty register (EB with no data tokens)

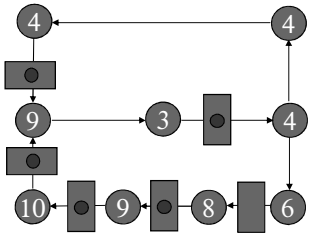
R&R is more powerful than retiming

Min delay retiming



C =16

Min delay R&R



C =15

Analogy between circuit retiming and reachability in MGs

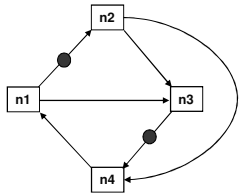
- Retiming graph of a circuit = MG:
  - combinational block = node
  - connection = edge
  - register = token
  - firing rules = backward retiming rules: each time a node is retimed, registers are removed from the input edges and added to the output edges
- MGs: A live marking M of an SCMG is reachable iff $M(\phi) = M0(\phi)$ for every cycle ϕ.

  Retiming interpretation.
  - => valid retiming preserves the number of registers at each cycle
  - <= if an assignment of registers has the same number of registers at each cycle as the initial circuit, then the assignment is a valid retiming.

Analogy between circuit retiming and reachability in MGs

- Non-negative marking M is reachable iff the marking equation holds:
  $M = M0 + A \times \sigma$
- Retiming interpretation:
  - $M0$ - initial assignment of registers to edges
  - M -assignment after retiming
  - A - retiming matrix
  - σ – retiming vector
- Rename M to R: $R = R0 + A \times \sigma$
- ILP formulation (A is totally unimodular. Polynomial problem)

Example of marking equation



$$\begin{array}{c} \\ e_{12} \\ e_{13} \\ e_{23} \\ e_{24} \\ e_{34} \\ e_{41} \end{array} \begin{bmatrix} 0 \\ 1 \\ 2 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + \begin{array}{c} \begin{matrix} n_1 & n_2 & n_3 & n_4 \end{matrix} \\ \begin{bmatrix} 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ -1 & 0 & 0 & 1 \end{bmatrix} \end{array} \times \begin{bmatrix} 1 \\ 2 \\ 0 \\ 1 \end{bmatrix}$$

$$M = M0 + A \times \sigma$$

Valid R&R solutions

Any integer solution for R and R' :

$$R \geq R' = R0 + A \times \sigma$$

is a valid R&R solution

- R' retiming subset (registers with data-tokens)
- R represents the R&R solution (registers with data-tokens or bubbles)
- (R – R') registers with bubbles (recycling)

Combinational path constraints

$$t_{in}(e) \geq t_{out}(e') + \delta(u), \forall e' = (w,u)$$
$$t_{out}(e) \geq t_{in}(e) - \tau' \times R(e)$$
$$t_{out} \geq 0$$
$$t_{in}(e) \leq \tau$$

τ – desired cycle time
τ' – original cycle time or any other constant $> \tau$
$\delta(u)$ – node u delay
R(e) – number of register on edge e=(u,v)
Register delays can be taken into account

Throughput constraints

$$R \le (R_0 + A \times \sigma) / \Theta$$

Let R be a valid R&R register assignment. There is a nonnegative real vector σ that fulfils the above inequality iff $\Theta(R) > \Theta$

ILP formulation for R&R

$$RR(\tau, \Theta) = \begin{cases} R \ge R_0 + A \times \sigma_1 \ge 0 \\ R \le (R_0 + A \times \sigma_2) / \Theta \\ Path\_constr(R, \tau) \\ R, \sigma_1 \in \text{int} \end{cases}$$

Given a cycle time τ and a throughput Θ, R is a valid R&R register assignment of an RRG (N,E,R0, δ) with $\tau(R) < \tau$ and $\Theta(R) > \Theta$ iff there exists a feasible solution of the above ILP

Min period R&R

Given an RRG and a throughput $\Theta > 0$, find a register assignment R that minimizes the cycle time τ and has throughput $\Theta(R) => \Theta$.

$$MIN\_PER(\Theta) = \begin{cases} \min \tau \\ subject \quad to \quad RR(\tau, \Theta) \end{cases}$$

Max throughput R&R

Given an RRG and a cycle period τ , find a register assignment R with $\tau(R) <= \tau$ that maximizes the throughput $\Theta(R)$.

$$MAX\_THR\_NL(\tau) = \begin{cases} \max \Theta \\ subject \quad to \quad RR(\tau, \Theta) \end{cases}$$

This problem is not linear (and not convex): Θ is a variable of the model and the second constraint of RR(τ, Θ) is not linear.

Use binary search on different Θ

Size of the interval for binary search

- Binary search explores $[\Theta_L,\Theta_U]$
  Θ_L has feasible R&R solution, Θ_U – does not
- What is the size of the interval not to miss an optimal solution?

$$|\Theta_L - \Theta_U| \geq 1/(|R0| + |E|)^2$$

$|R_0|$ – number of initial registers
$|E|$ – number of edges

Min effective cycle time R&R

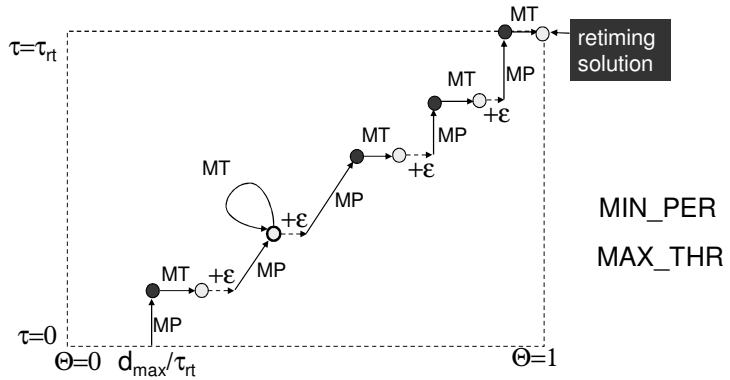
Given an RRG, find a register assignment R_{min} with a minimal effective cycle time $C(R_{min})$

```
MIN_ECYC_STEP(Θ):
  R1 := MIN_PER(Θ)
  R2 := MAX_THR(τ(R1))
return R2
```

```
MIN_ECYC(RRG):
  C := τrt;  Θ := d max/ τrt;  ε := 1/(|R0| + |E|)^2
  while Θ < 1
    R := MAX_ECYC_STEP(Θ)
    if C(R) < C then C := C(R)
    Θ := Θ(R) + ε
return C
```

Search for min effective cycle

Initialization: $\Theta(R_{min}) >= d_{max}/\tau_{rt}$, where d_{max} is the maximum delay of a node and τ_{rt} is the cycle period obtained by min-delay retiming



This search does not miss any solution with a better effective cycle time

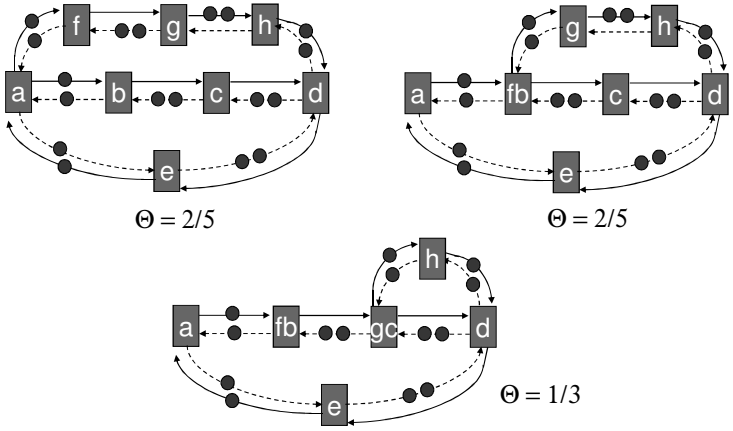
Results

- R&R can provide better effective cycle time than regular retiming if
  - long cycles and
  - unbalanced delays
- Useful for micro-architectural problems, not for well balanced circuit graphs

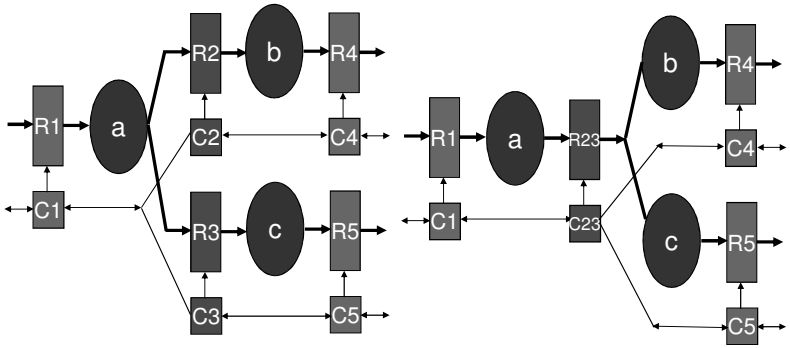
Clustering controllers

(joint work with Josep Carmona and Jorge Júlvez)

Merging nodes in elastic MG



Sharing controllers and elastic buffers



Mergeable transitions

- Definition:Transitions *ti* and *t j* of EMG *G* are called mergeable if *Th(G) = Th(G'),* where *G'* obtained by merging *ti* and *t j* in *G*
- Idea: Merge transitions with the same critical average marking at their input arcs
  - If transitions are not critical, then explore slack at the non-critical input arcs: check if the same throughput can be achieved with critical average marking

Correctness and Verification

Correctness (long story) = theory of Elastic Machines

(joint work with Sava Krstic and John O'Leary)

Correctness (short story)

- Developed theory of elastic machines
- Verify correctness of any elastic implementation = check conformance with the definition of elastic machine
- All SELF controllers are verified for conformance
- Elasticization is correct-by-construction

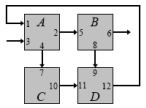
Systems

- A stream a over a set A is an infinite sequence $a[0], a[1], \ldots$ of elements of A
  - $a \sim_n b$ iff a and b have a common prefix of length n
- W is a set of wires
- W-behavior σ: a stream $\sigma.w$ for each $w \in W$
  - $[\![W]\!]$ = the set of all W-behaviors
  - $\sim_n$ extends to $[\![W]\!]$: $\sigma \sim_n \tau$ iff $(\forall w \in W)\ \sigma.w \sim_n \tau.w$
- A W-system is a subset of $[\![W]\!]$
  - Example: $\mathsf{Conn}(X, Y) = \{\sigma \mid \sigma.X = \sigma.Y\} \subseteq [\![\{X, Y\}]\!]$

Operations

- Projection $\sigma \mapsto \sigma.V \colon [\![W]\!] \to [\![V]\!]$ defined for $V \subseteq W$
- $\mathsf{hide}_V(\mathcal{S}) = \{\sigma.(W - V) \mid \sigma \in \mathcal{S}\} \quad \subseteq [\![W - V]\!]$
- $\mathcal{S}_1 \sqcup \mathcal{S}_2 = \{\sigma \mid \sigma.W_1 \in \mathcal{S}_1 \wedge \sigma.W_2 \in \mathcal{S}_2\} \quad \subseteq [\![W_1 \cup W_2]\!]$
- Networks of systems:

$$\langle \mathcal{S}_1, \ldots, \mathcal{S}_m \mid u_1 = v_2, \ldots, u_n = v_n \rangle \;=\; \mathsf{hide}_{\{u_1,\ldots,u_n,v_1,\ldots,v_n\}}(\mathcal{S}_1 \sqcup \cdots \sqcup \mathcal{S}_m \sqcup \mathsf{Conn}(u_1, v_1) \sqcup \cdots \sqcup \mathsf{Conn}(u_n, v_n))$$

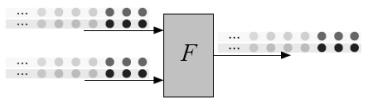


$= \langle A, B, C, D \mid 2 = 5, 4 = 7, 10 = 11, 8 = 9, 12 = 1 \rangle$

Machines = abstract circuits

Definition An (I, O)-machine is an $(I \cup O)$-system given by a function $F \colon [\![I]\!] \to [\![O]\!]$ satisfying the causality property

$$(\forall \sigma, \sigma' \in [\![I]\!])(\forall k \geq 0) \quad \sigma \sim_k \sigma' \implies F(\sigma) \sim_k F(\sigma')$$

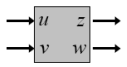


Outputs at the first k cycles are determined by inputs at the first k cycles.

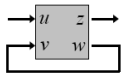
When is a network a machine?

- Feedback

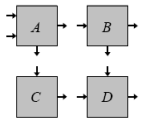
$\mathcal{S}$:

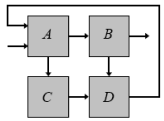


$\langle \mathcal{S} \mid v = w \rangle$:



- A network is a multiple feedback



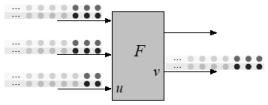


- Must preclude "combinational cycles"—but what are they?

Sequential and combinational dependency

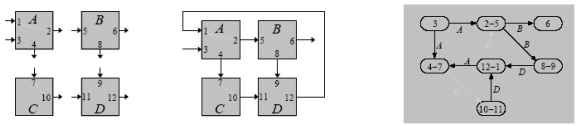
Definition An input-output pair (u, v) is sequential if

$$\left(\begin{smallmatrix} \forall \sigma, \sigma' \in [\![I]\!] \\ \forall k \geq 0 \end{smallmatrix} \right) \quad \begin{smallmatrix} \sigma.u \sim_{k-1} \sigma'.u \\ \wedge \\ (\forall x \neq u)\ \sigma.x \sim_k \sigma'.x \end{smallmatrix} \implies F(\sigma).v \sim_k F(\sigma').v$$



Feedback Lemma If (u, v) is sequential for a machine $\mathcal{S}$, then $\langle \mathcal{S} \mid u = v \rangle$ is a machine.

Detecting combinational loops

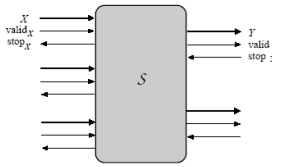


Definition $\Gamma(\mathcal{N})$: Vertices are wires of $\mathcal{N}$; directed edges drawn for non-sequential wire pairs.

Theorem If $\Gamma(\mathcal{N})$ is acyclic, then $\mathcal{N}$ is a machine.

[I,O] - Elastic machine

- Input-output structure
  - inputs: $I \cup \{\mathsf{valid}_X \mid X \in I\} \cup \{\mathsf{stop}_Y \mid Y \in O\}$
  - outputs: $O \cup \{\mathsf{valid}_Y \mid Y \in O\} \cup \{\mathsf{stop}_X \mid X \in I\}$



- Persistence
  - $\mathcal{S} \models \mathsf{G}\,(\mathsf{valid}_Y \wedge \mathsf{stop}_Y \Rightarrow (\mathsf{valid}_Y)^+)$ for every $Y \in O$

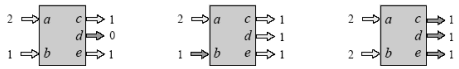
Liveness

- Liveness

  $(\forall Y \in O)$ $\mathcal{S} \models \mathsf{G}\,(\mathsf{min\_tct}_O \geq \mathsf{tct}_Y \wedge \mathsf{min\_tct}_I > \mathsf{tct}_Y \Rightarrow \mathsf{F}\,\mathsf{valid}_Y)$

  $(\forall X \in I)$ $\mathcal{S} \models \mathsf{G}\,(\mathsf{min\_tct}_{I \cup O} \geq \mathsf{tct}_X \Rightarrow \mathsf{F}\,\neg\mathsf{stop}_X)$

  - Serve only the most hungry channels:



  - Liveness guarantees that all transfer behaviors $\omega^{\mathsf{T}}.Z$ are infinite (in an "elastic environment")

$\therefore$ The transfer system $\mathcal{S}^{\mathsf{T}} = \{\omega^{\mathsf{T}} \mid \omega \in \mathcal{S} \sqcup \mathsf{Env}_{I,O}\}$

Elastic machine

- Determinism

  $(\forall \omega_1, \omega_2 \in \mathcal{S})$ $\omega_1^{\mathsf{T}}.I = \omega_2^{\mathsf{T}}.I \quad \Rightarrow \quad \omega_1^{\mathsf{T}}.O = \omega_2^{\mathsf{T}}.O$

Definition $\mathcal{S}$ is an $[I,O]$-elastic machine if it has the input-output structure as described, and satisfies the persistence, liveness, and determinism conditions.

Theorem If $\mathcal{S}$ is an $[I,O]$-elastic machine, then $\mathcal{S}^{\mathsf{T}}$ is an (I,O)-machine.

- $\mathcal{S}$ is an elasticization of $\mathcal{M}$ when $\mathcal{M} = \mathcal{S}^{\mathsf{T}}$

Elastic networks

Suppose $\mathcal{S}_1, \ldots, \mathcal{S}_m$ are elastic machines.

$$\mathcal{N} = \langle\!\langle \mathcal{S}_1, \ldots, \mathcal{S}_m \,[\![\, X_1 = Y_1, \ldots, X_n = Y_n \rangle\!\rangle$$

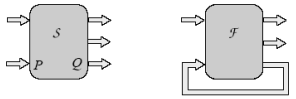
$$\triangleq$$

$$\langle \mathcal{S}_1, \ldots, \mathcal{S}_m \,|\, X_i = Y_i, \mathsf{valid}_{X_i} = \mathsf{valid}_{Y_i}, \mathsf{stop}_{X_i} = \mathsf{stop}_{Y_i}\ (1 \le i \le n) \rangle$$

- Is $\mathcal{N}$ an elastic machine?
- Do we have $\mathcal{N}^\mathsf{T} = \langle \mathcal{S}_1^\mathsf{T}, \ldots, \mathcal{S}_m^\mathsf{T} \,|\, X_1 = Y_1, \ldots, X_n = Y_n \rangle$?

Elastic feedback

$$\mathcal{F} = \langle\!\langle \mathcal{S} \,[\![\, P = Q \rangle\!\rangle = \langle \mathcal{S} \,|\, P = Q, \mathsf{valid}_P = \mathsf{valid}_Q, \mathsf{stop}_P = \mathsf{stop}_Q \rangle$$



Definition An i/o channel pair (P, Q) sequential for $\mathcal{S}$ if

$$\mathcal{S} \models \mathsf{G}\,(\mathsf{min\_tct}_{I \cup O} \ge \mathsf{tct}_Q \wedge \mathsf{min\_tct}_{I - \{P\}} > \mathsf{tct}_Q \Rightarrow \mathsf{F}\,\mathsf{valid}_Q)$$

and the graph $\Gamma(\mathcal{F})$ is acyclic.

Theorem If the channel pair (P, Q) is sequential for $\mathcal{F}$, then

- the wire pair (P, Q) is sequential for $\mathcal{S}^\mathsf{T}$
- $\mathcal{F}$ is an elastic machine
- $\mathcal{F}^\mathsf{T} = \langle \mathcal{S}^\mathsf{T} \,|\, P = Q \rangle$

Elastic network theorem

- $\mathcal{N} = \langle\!\langle \mathcal{S}_1, \ldots, \mathcal{S}_m \,[\![\, X_1 = Y_1, \ldots, X_n = Y_n \rangle\!\rangle$
- δ_i : a sequentiality interface for $\mathcal{S}_i$ ($\delta_i(Z)$ is a set of input wires "jointly sequential" wrt Z)

Definition $\Delta(\mathcal{N})$: Vertices are channels of $\mathcal{N}$ ($\because X_j$ and Y_j are identified); a directed edge drawn for each pair $(P, Q) \in I_i \times O_i$ such that $P \notin \delta_i(Q)$.

- $\mathcal{N}' = \langle \mathcal{S}_1^\mathsf{T}, \ldots, \mathcal{S}_m^\mathsf{T} \,|\, X_1 = Y_1, \ldots, X_n = Y_n \rangle$

Theorem If $\Delta(\mathcal{N})$ is acyclic, then $\mathcal{N}$ is an elastic machine, $\mathcal{N}'$ is a machine, and $\mathcal{N}^\mathsf{T} = \mathcal{N}'$.

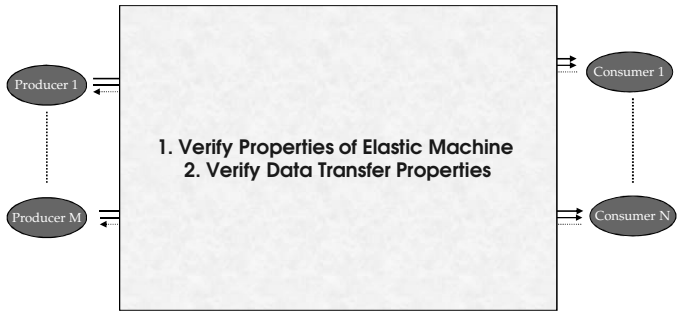
Inserting empty buffers

Theorem Suppose $\mathcal{N}_1$ and $\mathcal{N}_2$ are elastic networks obtainable from each other by insertion and deletion of empty elastic buffers. If one of $\Delta(\mathcal{N}_1)$, $\Delta(\mathcal{N}_2)$ is acyclic, then the other is acyclic too, and one has $\mathcal{N}_1^\mathsf{T} = \mathcal{N}_2^\mathsf{T}$.

Verification of elastic systems

(joint work with Syed Suhaib and Sava Krstic)

Implementation of Elastic Module



Problem

- Infinite domain transfer counters (tct)
- Model checking requires finite domain counters

Finite domain is sufficient

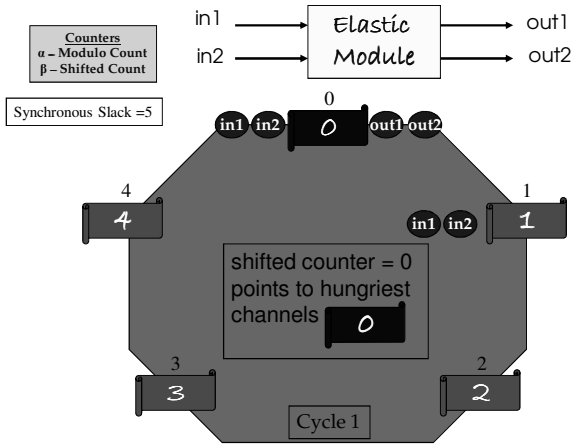
- Any implementation has finite sequential depth between any input channel and any output channel
- Model the tct variable as integers modulo (k+1) in some finite range [0,k] sufficient to cover maximal sequential depth
- Reset the tct when range reaches k, and restart from 0

How do you compute k?

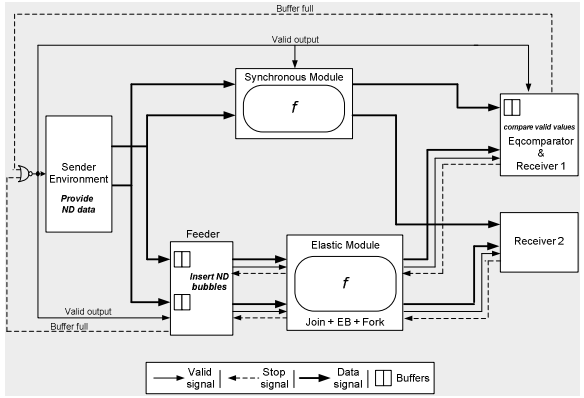
Synchronous Slack

- Capacity: C(i,j) is defined as the maximum number of data storage elements between channels i and j, where i ∈ I and j ∈O
- For an [I,O]-system S, its synchronous slack $\varepsilon = \min_{\{i,j\}} \{k: G(k \geq \max(|tct_i - tct_j|))\}$
- $\varepsilon <= \max_{\{i,j\}} C(i,j)$

Modeling of Counters



Validating Data Correctness



Use uninterpreted function

- Symbolic terms and uninterpreted function
  - Proposed by Burch and Dill '94
- We employ similar procedure
  - Encode all possible terms
  - Combinational logic modeled as a single function
- Consider, e.g., a two input uninterpreted function

Model checking

- SPIN Model Checker [Bell Labs]
- NuSMV Model Checker [IRST and CMU]

Summary

- SELF gives a low cost implementation of elastic machines
- Functionality correct when latencies change
- New micro-architectural opportunities
- Compositional theory proving correctness
- Early evaluation - mechanism for performance and power optimization
- Retiming and recycling, buffer optimization and other optimization opportunities
- To read on this work: see list of references

Research directions

- How to specify elastic machines
  - Asynchronous specification (e.g., CSP) discretized asynchrony view
  - Elastic synchronous specification (extend Esterel, Lustre, PBS with controlled asynchrony)
- Compilers
- Improve bounds on analytical perf. analysis for early evaluation
- Formal methods for micro-architectural optimization
- R&R and buffer optimization for systems with early evaluation
- More on optimization for elastic machines

Some of Related work

- **Async**
  - Rings (T. Williams, J.Sparso)
  - Caltech CHP and slack-elasticity (A. Martin, S.Burns, R.Manohar et al.)
  - Micropipelines (I. Sutherland)
  - Many others
- **Latency insensitive design**
  - L. Carloni and a few follow-ups (large overhead)
  - C. Svensson (Linköping U.) - wire-pipelining
- **Interlock pipelines**
  - H. Jacobson et al.
- **Desynchronization**
  - J. Cortadella et al.
  - V. Varshavsky
- **Performance analysis**
  - S.Burns
  - H. Hulgaard
  - C.Nielsen/M.Kishinevsky, etc.
- **Synchronous implementation of CSP**
  - J. O'Leary et al.
  - A. Peeters et al.
- **Telescopic units** – Benini et al.
- See a list of refrerences