

# Using periodic static schedule information in Latency-Insentive Design

SYNCHRON'06 - 29th November – Alpes d'Huez

INSTITUT NATIONAL  
DE RECHERCHE  
EN INFORMATIQUE  
ET EN AUTOMATIQUE



Julien  
BOUCARON,  
Jean Vivien MILLO  
Robert de SIMONE,  
*AOSTE* Team

# General Motivations

*Single clock* design of SoC soon *no longer feasible*.

- **Due to** increasing size and density
  - *Long wires latencies.*
  - *Clock Tree propagation issues.*
  - *Reaching Timing Closure when assembling IPs.*

*GALS Models & Latency Insensitive Design (LID)*

- Former step: hardware implementation for **dynamic scheduling**.
- Second step: hardware optimization using **periodic static scheduling** results.

# Outline

- Preliminaries
  - Basic Synchronous/Asynchronous Models (FreeChoice)
  - Introducing latencies
    - Dynamic Scheduling mechanisms for LID
    - Static Scheduling & K-Periodic behavior:

## Central Repetitive Problem

### ➤ *Our contributions*

#### ➤ *Static Periodic Scheduling Mechanisms*

◆ *Latency-based Equalization process.*

◆ *Fractional Registers for residual latencies.*

#### ➤ *KPassa tool & Experiments.*

- Conclusion & Further Topics

# Common Basis: Computation Network Scheme

Computation Nodes and Data Link communication Arcs.

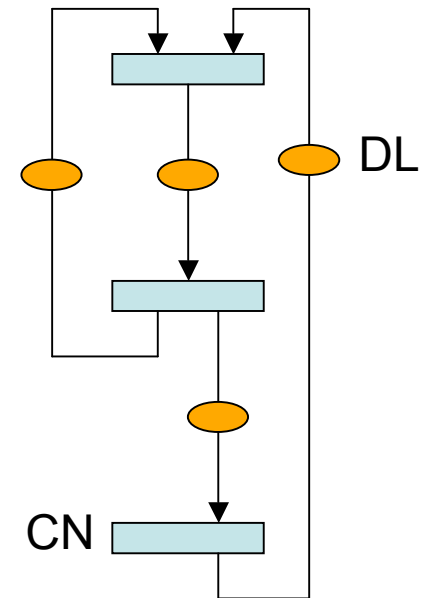
**Intuitive (incomplete) semantics:**

**CN** nodes consume/produce data on all input/output **DL** arcs

Data values abstracted as *tokens*  
(data present/absent)

**No conflict choice or alternative:**

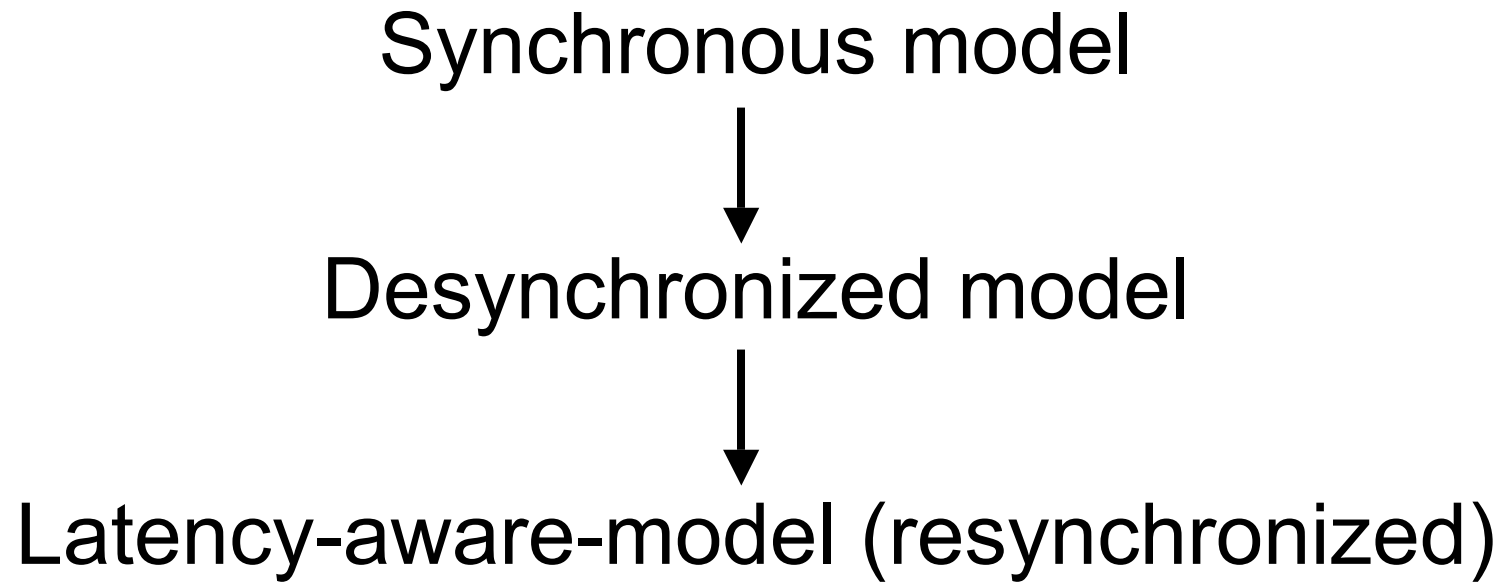
each link has one source and one target.



**Various Models** obtained by specializing

- Firing Rule (Sync, Async)
- Nature of DL Buffering (Capacity, Latency)

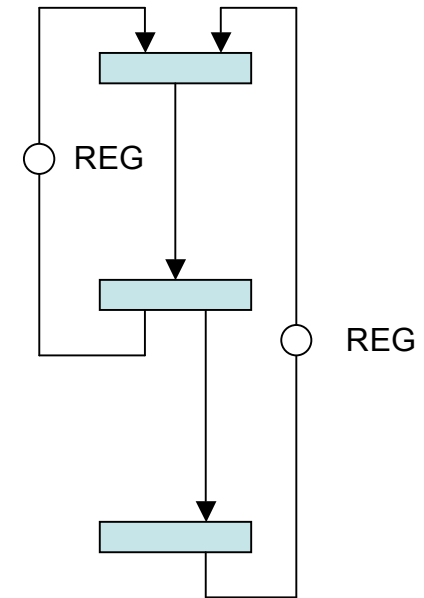
# A “flow” of intermediate models



# First Model

## 1. Synchronous

- All computations simultaneous
- Data links: **wires** or **unit delay/capacity** registers
- **Correct** if at least one register in each cycle



## 2. Asynchronous (Marked Graphs)

- Independent computation triggering
- Data links: unbounded buffers
- Correct if at least one token in each cycle

# Second Model

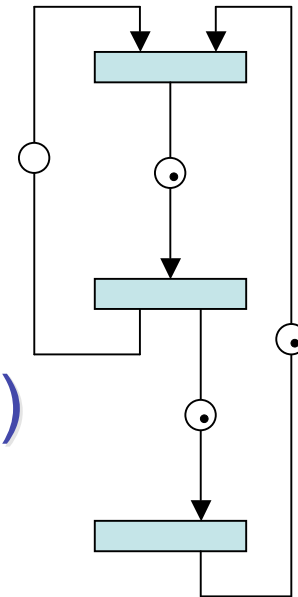
## 1. Synchronous

- All computations simultaneous
- Data links: wires or unit delay/capacity registers
- Correct if at least one latch in each loop

## 2. Asynchronous (Marked/Event Graphs)

- Independent computation triggering
- Data links: **unbounded buffers**
- **Correct** if at least one token in each cycle

[Commoner, Holt, Even & Pnueli 1971]



# Second Model

## 1. Synchronous

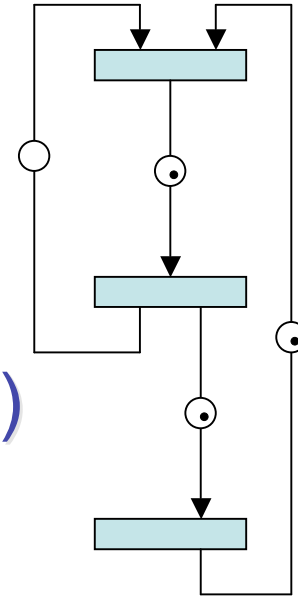
- All computations simultaneous
- Data links: wires or unit delay/capacity registers
- Correct if at least one latch in each loop

## 2. Asynchronous (Marked/Event Graphs)

- Independent computation triggering
- Data links: unbounded buffers
- Correct if at least one token in each cycle

[Commoner, Holt, Even & Pnueli 1971]

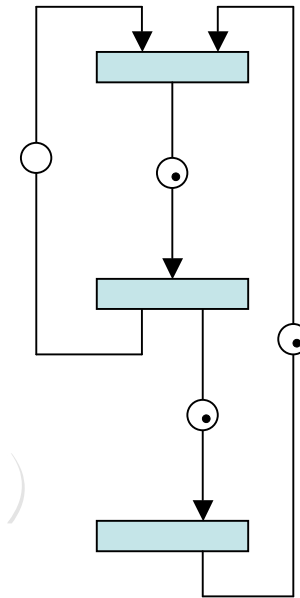
**Conflict Freeness: Same behaviour as synchronous,  
different timing.**



# Second Model

## 1. Synchronous

- All computations simultaneous
- Data links: wires or unit delay/capacity registers/latches
- **Correct** if at least one register value in each cycle



## 2. Asynchronous (Marked/Event Graphs)

- Independent computation triggering
- Data links: unbounded buffers
- **Correct** if at least one token in each cycle

## 3. SAME CORRECTNESS !!!

# Variation on Async Model

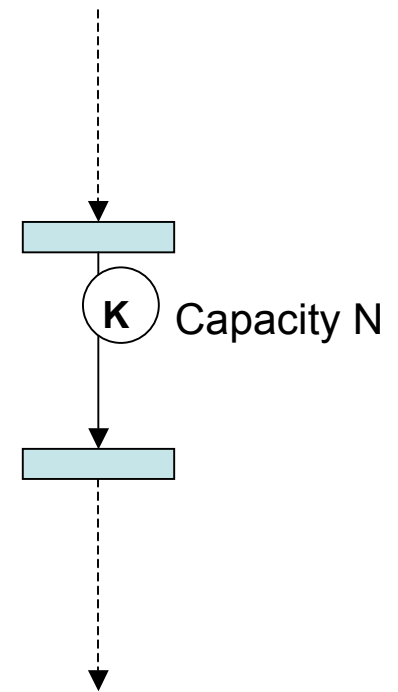
## Marked Graphs with capacities

- Data Links: **finite capacity buffers**

New issue: buffer congestion

Can be reduced to previous asynchronous **unbounded buffers case** by adding reverse back-pressure arcs.

- Correct if at least one token in each loop of the completed graph.



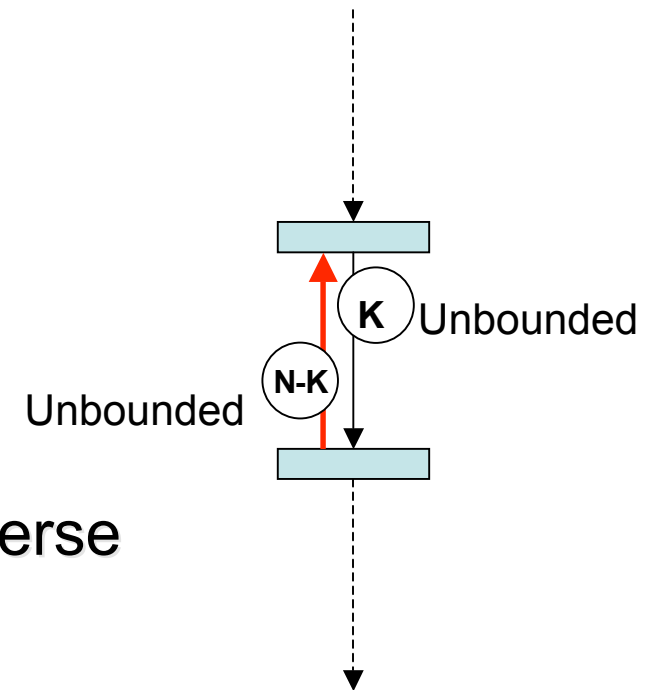
# Variation on Async Model

## Marked Graphs with capacities

- Data Links: **finite capacity buffers**

New issue: buffer congestion

Can be reduced to previous asynchronous **unbounded** buffers case by adding reverse **back-pressure** arcs.



- **Correct** if at least one token in each cycle of the **completed** graph.

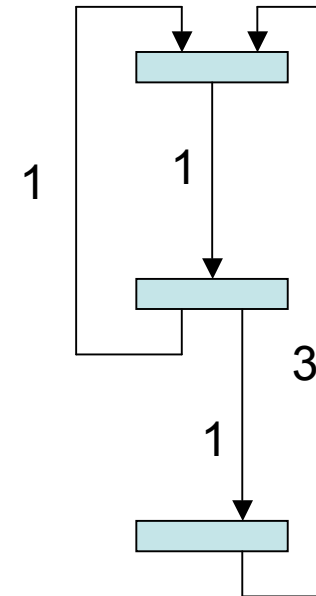
# Introducing arc latencies

**Latency = Duration  $\leftrightarrow$  Capacity**

Defined from

- **Asynchronous models**
  - **Timed Marked Graph theory**  
(RAMCHANDANI 1973)
- **Synchronous models**
  - **Latency Insensitive Design theory**  
(CARLONI, SANGIOVANNI-VINCENTELLI, MCMILLAN 1999)
- **ASAP semantics** (*Synchronous in Nature*)
  - All CN that may fire, do so.

*Some CN may idle because some tokens unavailable due to different latencies.*

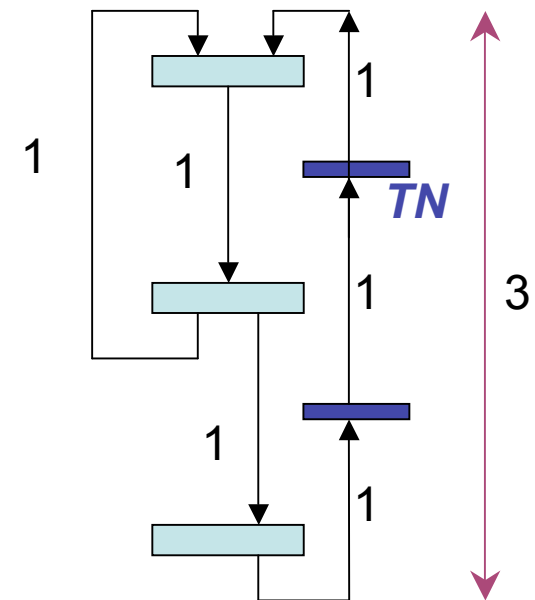


# Expanding latencies

Extra *Transportation Nodes* can be explicitly introduced to expand latencies in between *unit time* travel sections

Tokens only travel from a buffer to the next in one unit of time

Transportation Nodes similar to Computation Nodes



# Latency Insensitive Design

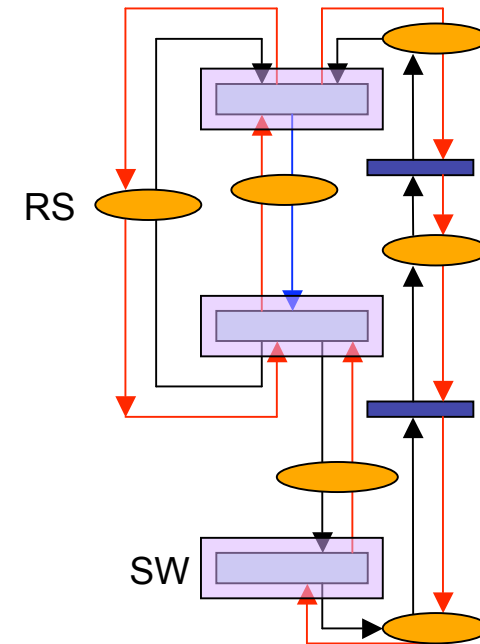
- **Relay Stations**

- Buffering between sections of latency one.
- Places of Capacity 2.

Correctness: initially at most one token.

- **Shell wrappers** around CN

- Activate the CN when input tokens and output slots are available
- **Clock gating.**



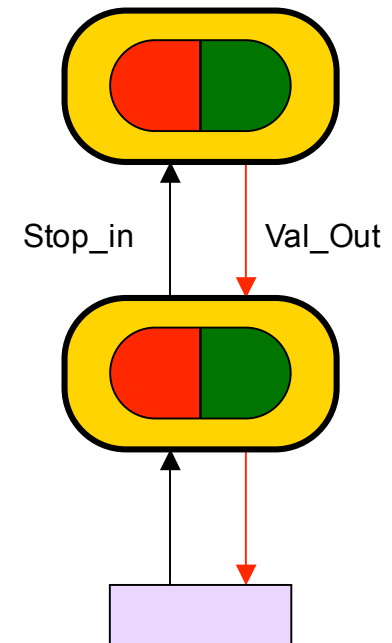
# Relay Station

## Buffer of capacity 2

3 States: Full, Half, Empty

## Behavior:

- IDEA: hold its token (if any) when congestion ahead.
- BUT: cannot warn its predecessor.
- THUS may simultaneously receive a second token.
- THEN: will signal congestion at next instant (no new token).



NEEDS **buffers of capacity 2** (RS) in between CN/TN  
→ Timed Marked Graph of capacity 2

# Efficient Hardware Implementation for LID

CARLONI, SANGIOVANNI-VINCENTELLI, MCMILLAN [2001]

CASU, MACCHIARULO [2003]

CHELSEA, NOWICK [2004]

BOUCARON, MILLO, DE SIMONE [2005]

CORTADELLA, KISHINEVSKY, GRUNDMANN [2006]

Later CASU, MACCHIARULO[2004] and ourselves[2006] recognize the periodicity of behaviours and the connection with Timed Event Graph, which provides useful known classical results.

# Second part: Now what about static scheduling ?

**Fact:** in Strongly Connected Timed Event Graphs, ASAP semantics leads to **k**-periodic static schedules [[Carlier/Chretienne, Baccelli et al](#)]

- After an initial phase, the token distribution becomes repetitive with **k** activation over a period of length **p**
- Graph Throughput =  $k/P$

# Outline

- Preliminaries
  - Basic Synchronous/Asynchronous Models (FreeChoice)
  - Introducing latencies
    - Dynamic Scheduling mechanisms for LID
    - Static Scheduling & K-Periodic behavior:  
Central Repetitive Problem

## ➤ ***Our contributions***

### ➤ ***Static Periodic Scheduling Mechanisms***

- ◆ ***Latency-based Equalization process.***
- ◆ ***Fractional Registers for residual latencies.***

### ➤ ***KPassa tool & Experiments.***

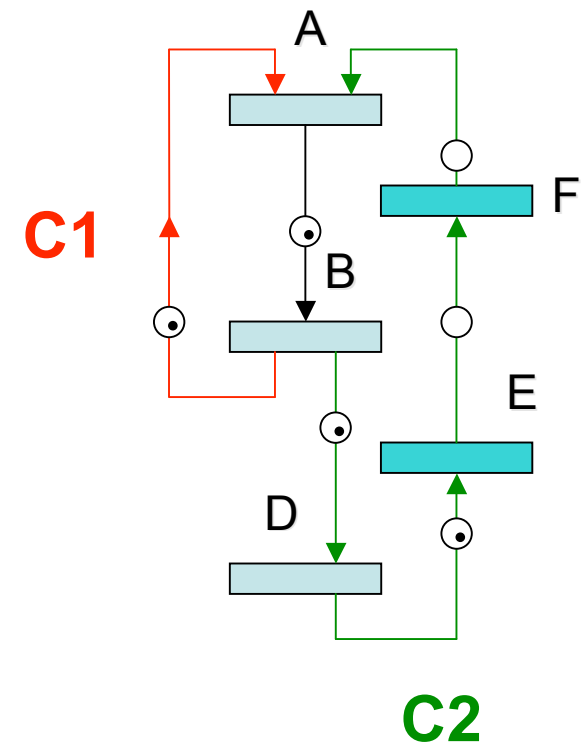
- Conclusion & Further Topics

# Our goal

- *Provide a “better” hardware implementation by exploiting this static schedule information*
  - Preserve original throughput of LID Spec.
  - Remove back-pressure mechanism
  - RS → simple register
    - + Fractional Registers only in precise locations
  - Shell → K-periodic schedule (computed off-line)
- *Do this by “Equalization”*
  - Add as much “virtual” integer latencies as possible to equalize throughputs
  - Fractional Registers for “residual differences”

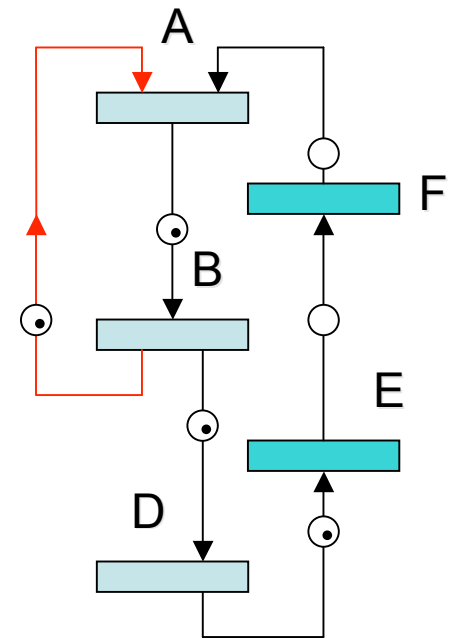
# Equalization example (1/3)

- 1) *List all elementary cycles*
- 2) *Compute all throughputs*
  - Find critical cycles
- Here : 2 cycles
  - **C1** : 2 tokens / 2 latencies
  - **C2** : 3 tokens / 5 latencies
  - $3/5 < 2/2$  so **C2** is *critical*.



# Equalization example (2/3)

- *3) Add virtual latencies to fast cycles, but not too much*
- **red arc** is the only possible location  
→NB: In general not only one solution !!!!

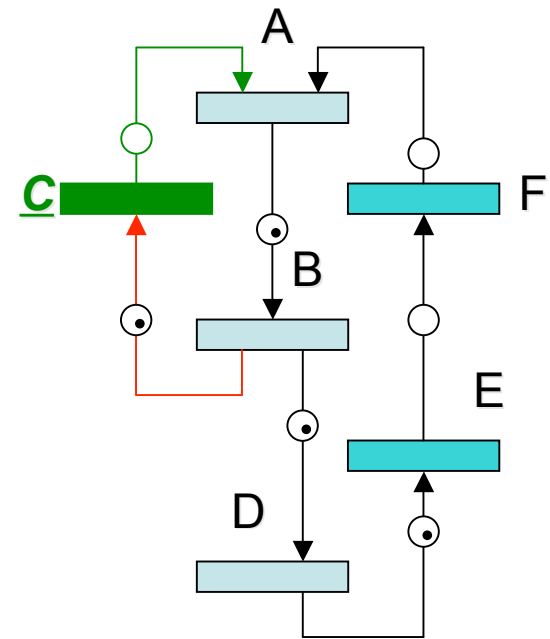


# Equalization example (2/3)

- *3) Add virtual latencies to fast cycles, but not too much*

- **red arc** is the only possible location
- Adding a unitary latency

$$2/(2+1) = 2/3 > 3/5 \text{ (still ok)}$$





# Equalization example (2/3)

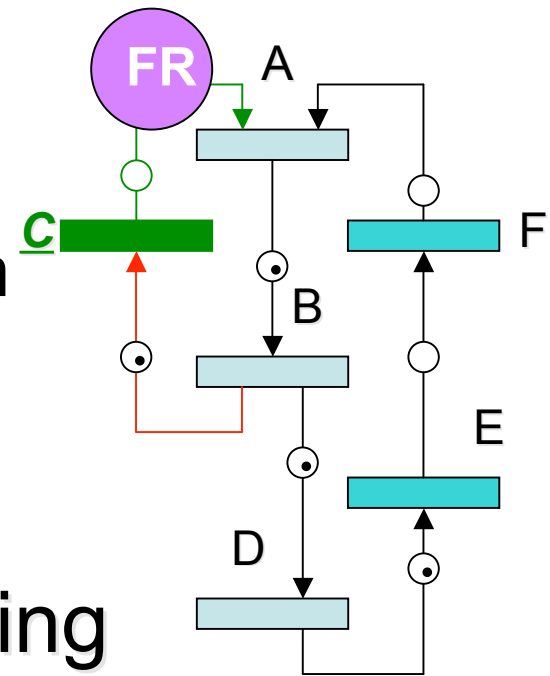
- 3) Add virtual latencies to fast cycles, but not too much

- **red** arc is the only possible location

- Adding a unitary latency

$$2/(2+1) = 2/3 > 3/5 \text{ (still OK)}$$

Still  $2/3 <> 3/5$  so “fractional” buffering needed at one place



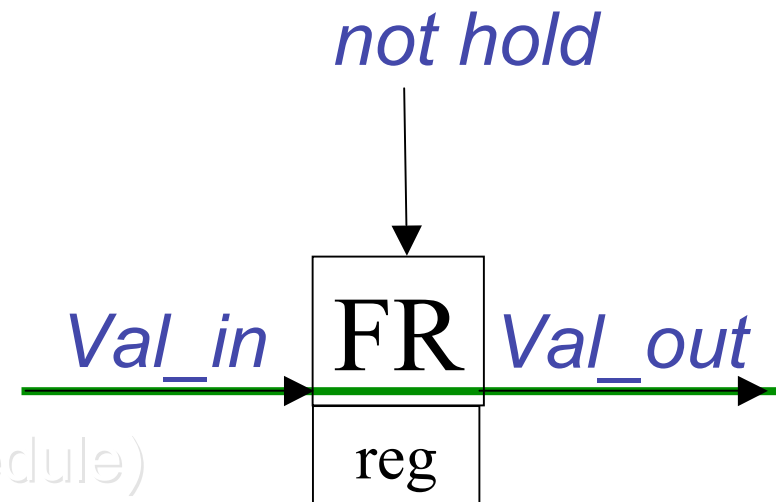
# Fractional Register behavior

- Acts as combinatorial **wire** when *not hold*.

- Acts as register when *hold*, keeping
  - a *single* token several steps, or
  - a token *sequence* in a row (each one once)

- *Correctness property:*

no *Val\_out* is sent when recipient not ready (by static schedule)



Registers + FRs provide expressiveness of Relay Stations.

But the goal is to add them only where needed

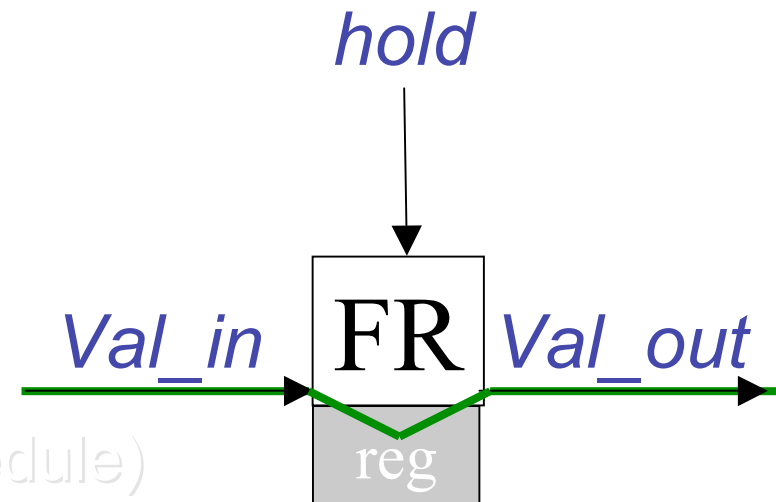
# Fractional Register behavior

• Acts as combinatorial wire when *not hold*.

- Acts as register when **hold**, keeping
  - a *single* token several steps, or
  - a token *sequence* in a row  
(each one once)

• *Correctness property:*

no *Val\_out* is sent when recipient not ready (by static schedule)

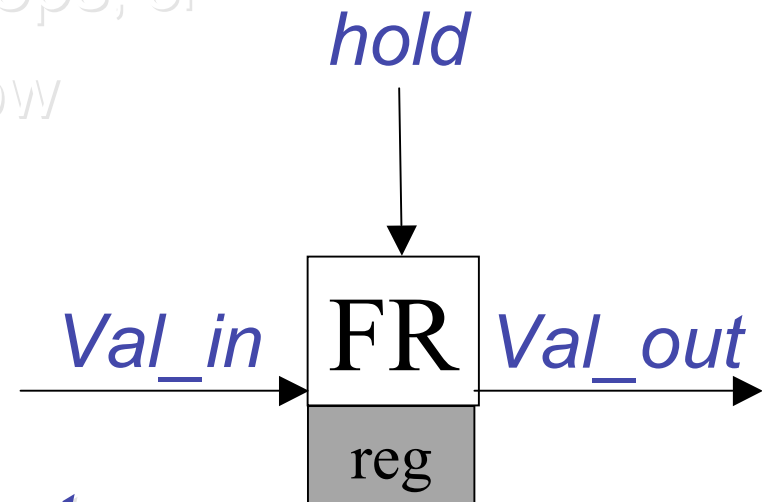


Registers + FRs provide expressiveness of Relay Stations.

But the goal is to add them only where needed

# Fractional Register behavior

- Acts as combinatorial wire when *not hold*.
- Acts as register latch when *hold*, keeping
  - a *single* token several steps, or
  - a token *sequence* in a row  
(each one once)



- **Global Correctness property**

(of static schedule) :

no *Val\_out* is sent when recipient not ready

# FR versus RS

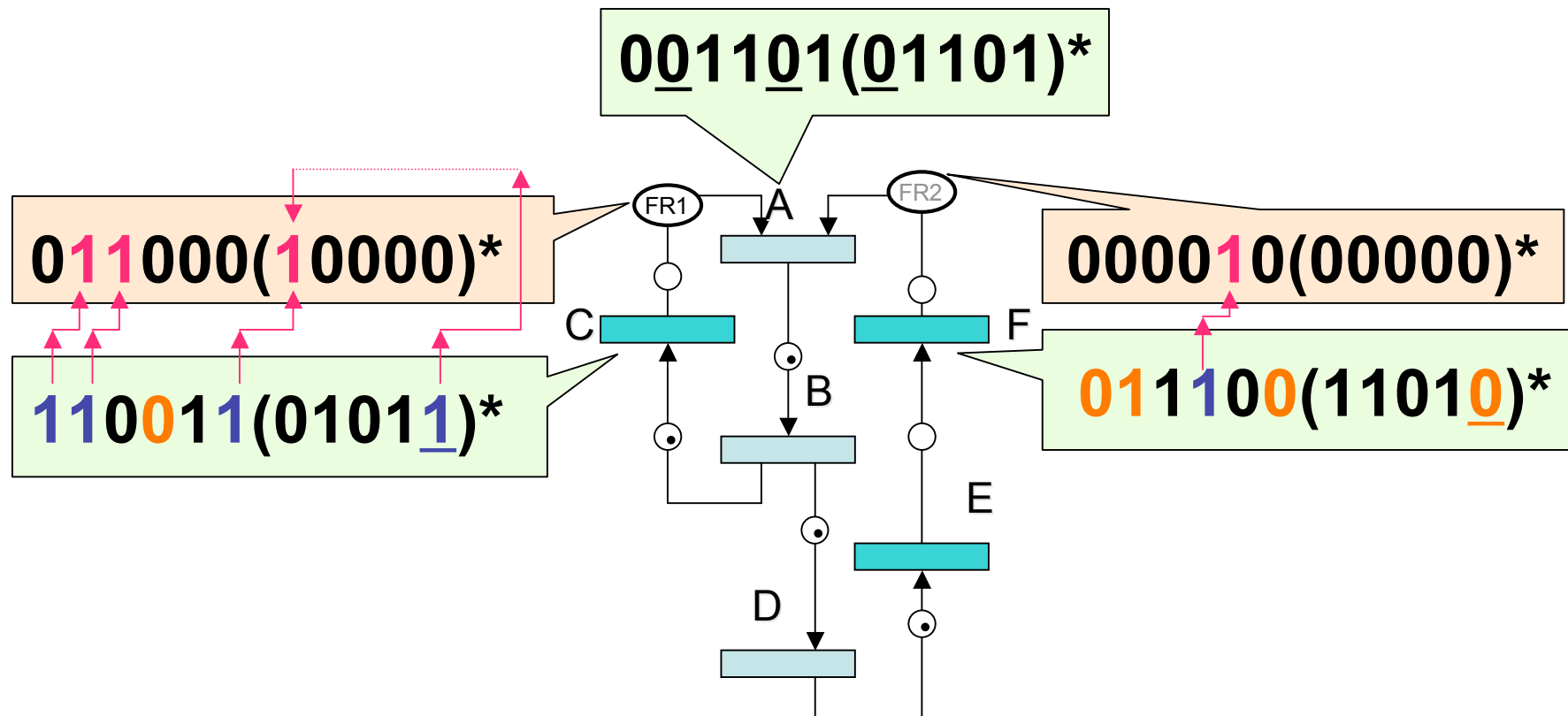
Adding one FR to a Register in each section provide expressiveness of Relay Stations.

But the goal is to add them only where needed after saturation by “virtual” integer latencies.

Todo this we compute schedules as first class citizen

# Equalization example (3/3)

- 4) *Symbolic Simulation provides explicit schedules and Fractional Registers*



# Equalization Algorithmic steps

1. *Enumerates* all *Elementary Cycles*.
2. *Compute* each cycle *throughput* (k/p)  
→ find Critical cycles.
3. *Add integer latencies to non-critical arcs*  
(as many as possible)  
→ Use a LP Solver (needs all cycles of step 1).
4. *Simulate* to build the *k-periodic* schedule  
and *place* the extra *Fractional Registers* and  
their schedules (*hold*).

# KPassa Tool

- Pronounced “*Que passa*”, for **K-Periodic As-Soon-as-possible Scheduling and Analysis**.
  - Written in *JAVA*®.
  - uses *ILOG*® *CPLEX*® LP Solver to add virtual latencies.
  - uses INRIA *Mascot Lib*: (Graph algorithms)  
[www-sop.inria.fr/mascotte/mascot/](http://www-sop.inria.fr/mascotte/mascot/)

# KPassa Tool

	#Nodes	#Cycles	#Critical Cycles	Max Cycle Latency	Throughput
MPEG2 Video Encoder	16	7	3	21	3/7
Encoder MultiStandard ADPCM	12	23	23	14	1/2
H264/AVC Encoder	20	12	3	27	4/9
29116a 16bits CAST MicroCPU	11	7	3	35	3/35
Abstract Stress Cycles	40	2295	1	1054	4/29
Abstract Stress Nodes	175	3784	1	1902	4/29

Table 1: Example sizes before equalization

	Perfect Eqn.	#FR init/periodic	#Added latencies	Time	Memory
MPEG2 Video Encoder	N	9/5	18	<1 sec	~11MB
Encoder MultiStandard ADPCM	Y	24/0	91	<1 sec	~11MB
H264/AVC Encoder	N	18/11	0	~ 1sec	~11MB
29116a 16bits CAST MicroCPU	Y	0/0	0	~ 1sec	~11MB
Abstract Stress Cycles	N	55/24	1577	~17 sec	~16MB
Abstract Stress Nodes	N	59/23	2688	~4 min	~43MB

Table 2: Equalization performances and results (Run on P4 3.4Ghz, 1GB RAM , Linux 2.6 and JDK 1.5)

Relatively Time Efficient, but Space Efficiency can be greatly improved.

Schedules representation need to be improved

# Optimization Problems

**Goal:** Schedule of a CN is the schedule of its predecessor shifted by one instant as much as possible → Not represent all schedules

**Naïve Idea:** Only one FR is needed where a fast cycle converge with a slower one.

FALSE! Counter-examples exist.

**Open question:** can we improve the distribution of tokens to make it true (asynchronous initialization)

# Work in Progress

- *Find Smooth Schedules*
- *Study efficient asynchronous initialization*
  - so that smooth periodic regimes are met fast
  - so number of required FR is minimized
- *Optimize allocation of virtual latencies*

**Thank you for your attention**

# Historical biblio recap

- [Reiter, 1968]: the system is limited to the throughput of its slowest cycle component.
- [Carlier, Chretienne 1987]: the system under ASAP rule is actually ultimately k-periodic with the throughput of its slowest cycle.
- [Baccelli, Cohen, Quadrat et. al 1992]: value of p
$$p = \text{lcm}_{\text{critical\_SCCs}} (\text{gcd}_{\text{cycles\_in\_critical\_SCC}} (\text{latency}))$$

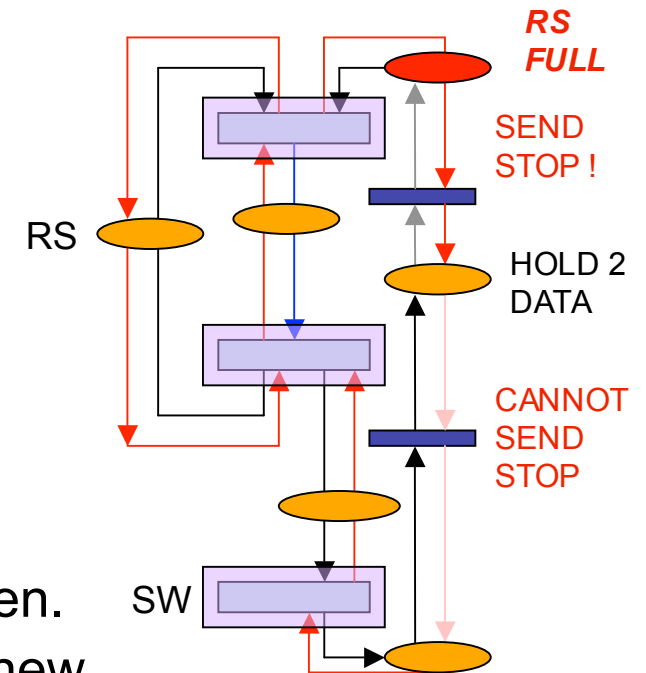
# Latency Insensitive Design

## Hardware Implementation:

- **Relay Stations** for buffering.
- **Shell wrappers** around CN  
→ local **clock gating**.

## RS Behavior:

- IDEA: hold its token (if any) when congestion.
- BUT: cannot warn its predecessor.
- THUS may simultaneously receive a second token.
- THEN: will signal congestion at next instant (no new token).



NEEDS **buffers of capacity 2** in between computation/transport nodes.

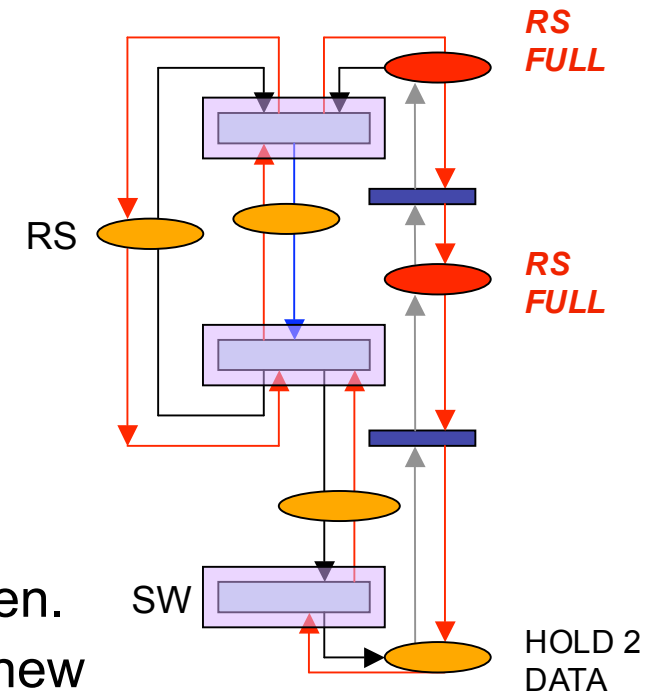
# Latency Insensitive Design

## Hardware Implementation:

- **Relay Stations** for buffering.
- **Shell wrappers** around CN  
→ local **clock gating**.

## RS Behavior:

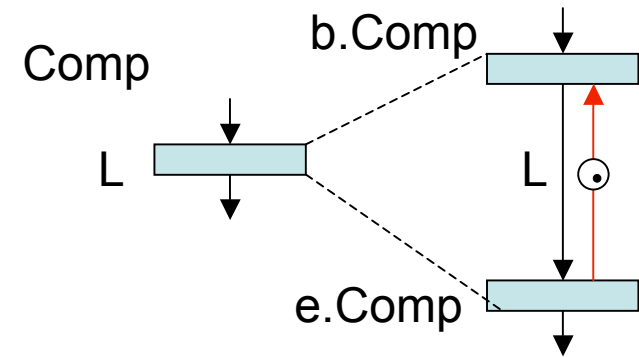
- IDEA: hold its token (if any) when congestion.
- BUT: cannot warn its predecessor.
- THUS may simultaneously receive a second token.
- THEN: will signal congestion at next instant (no new token).



NEEDS **buffers of capacity 2** in between computation/transport nodes.

# Variation on latencies

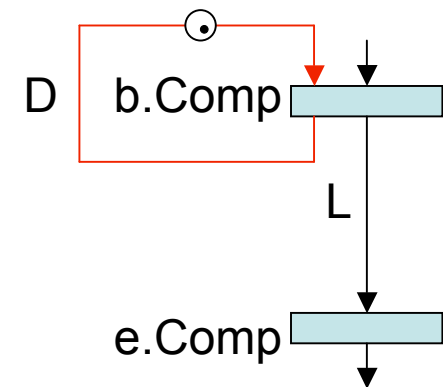
CN computation latency can be represented by transportation latency



Can also deal with pipelined computations

**Latency:** time needed from input to output

**Delay:** time needed between successive inputs



# Equalization example (3/3)

- 4) *Symbolic Simulation provides explicit schedules and Fractional Registers*

