Fusion Shell
&
Retry Relay-Station

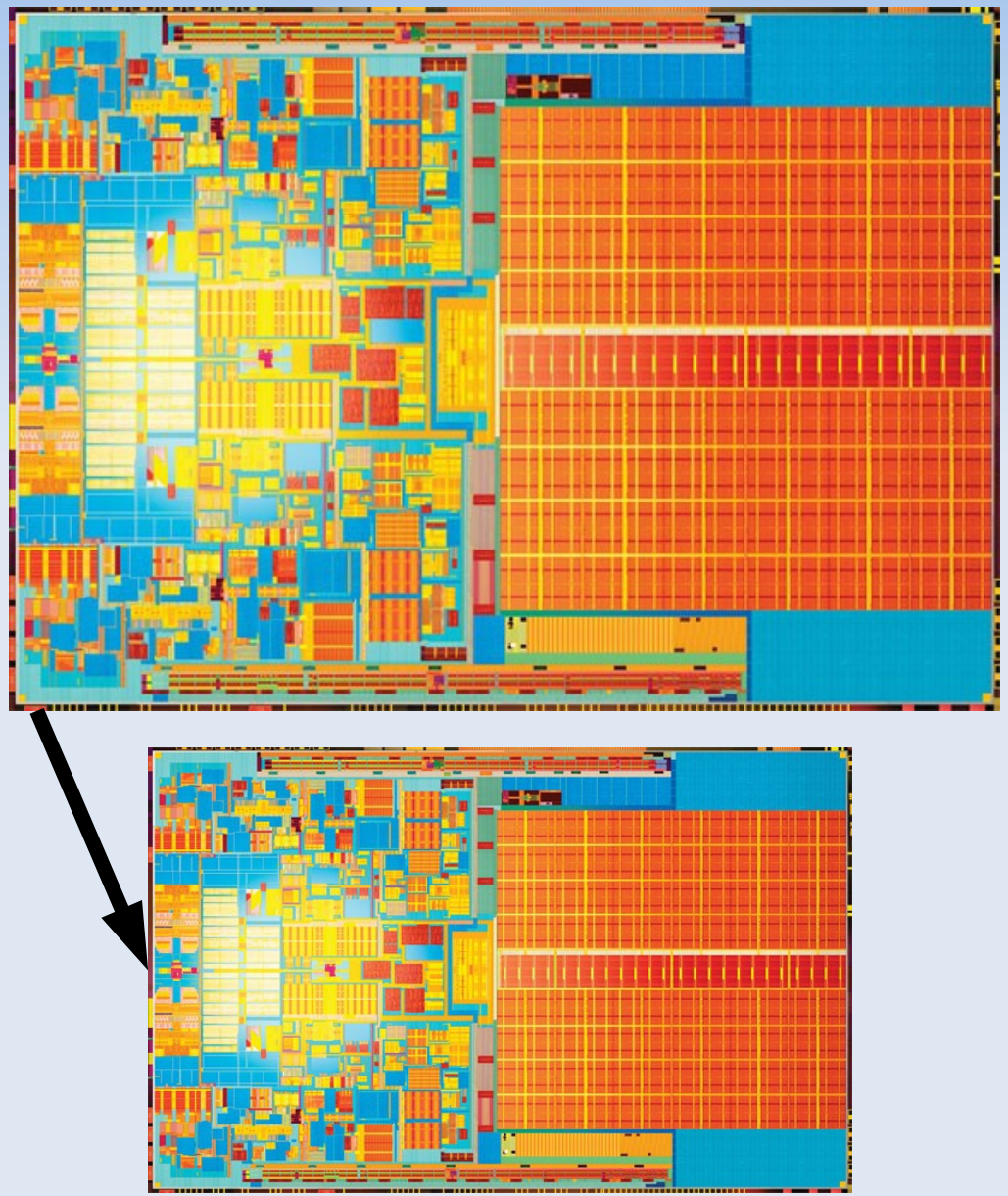
Yet Another Implementation
for
Latency Insensitive Design

Julien Boucaron, Anthony Coadou and Robert de Simone
INRIA, EPI AOSTE, Sophia-Antipolis

Agenda

- Introduction
  - Latency...
  - Latency Insensitive Design (LID)
  - How LID works ?
  - LID in a nutshell
  - Regular LID Implementation
- **Our Idea**
- **New Implementation**
  - **Retry Relay-Station**
  - **Fusion Shell**
- Results
  - FPGA
  - ASIC
- Conclusion

Latency... latency...

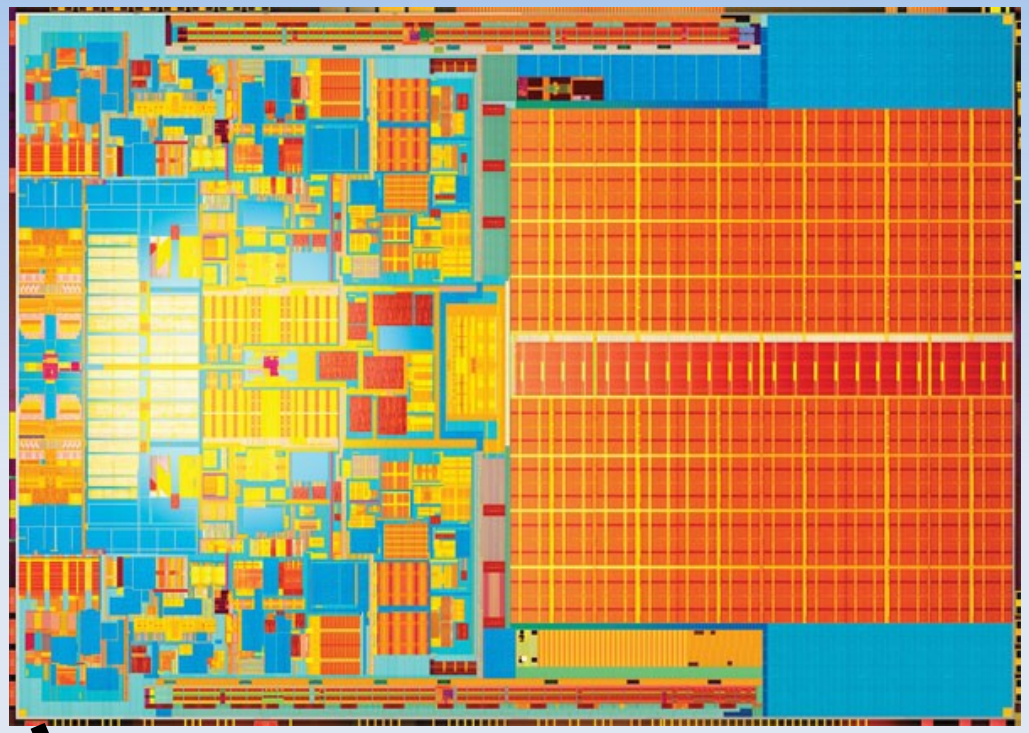


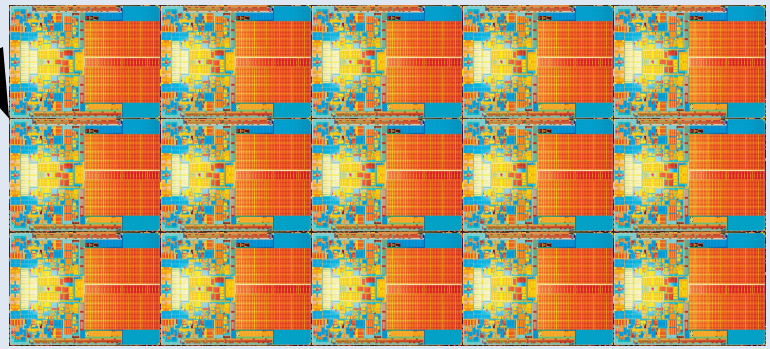
Core 2 Duo Woldale, Courtesy of Intel ®

- As die shrinks with newer processes
  - Gate Delay shorten
  - But... Interconnect Delay rise:

  Because as wires are getting smaller, resistivity rises whatever is your metal material

Latency... latency...

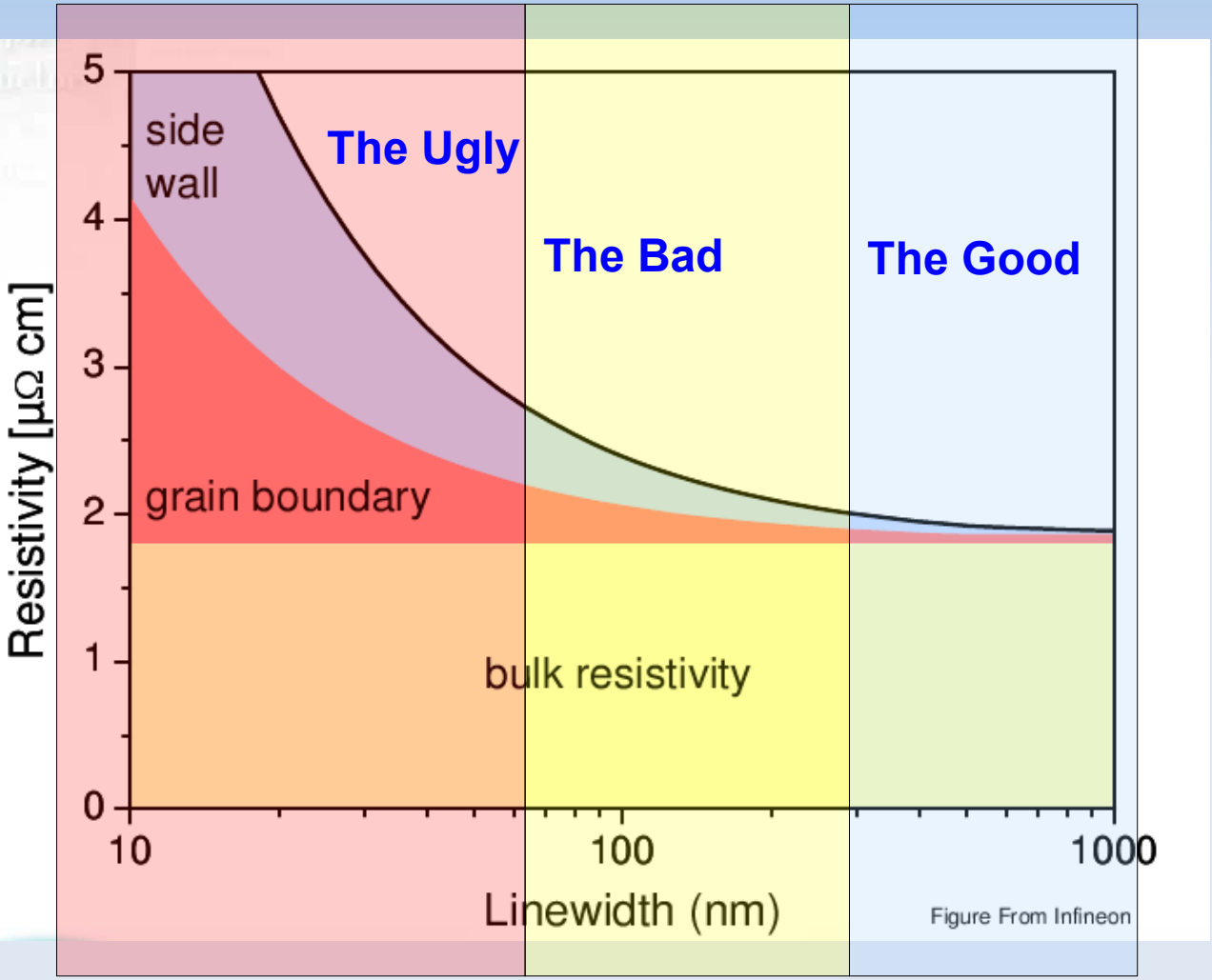




Core 2 Duo Woldale, Courtesy of Intel ®

- Yesterday, the die of the chip was big as this room: we were able to communicate together in « a clock cycle »
- But today, the die is big as the city or the whole country, it takes « more than a clock cycle »

Latency... latency...



Taken from ITRS RoadMap 2007
Courtesy of SEMAGROUP

- Latency is a main bottleneck for high-performance design
- **.... and it is getting worse !**

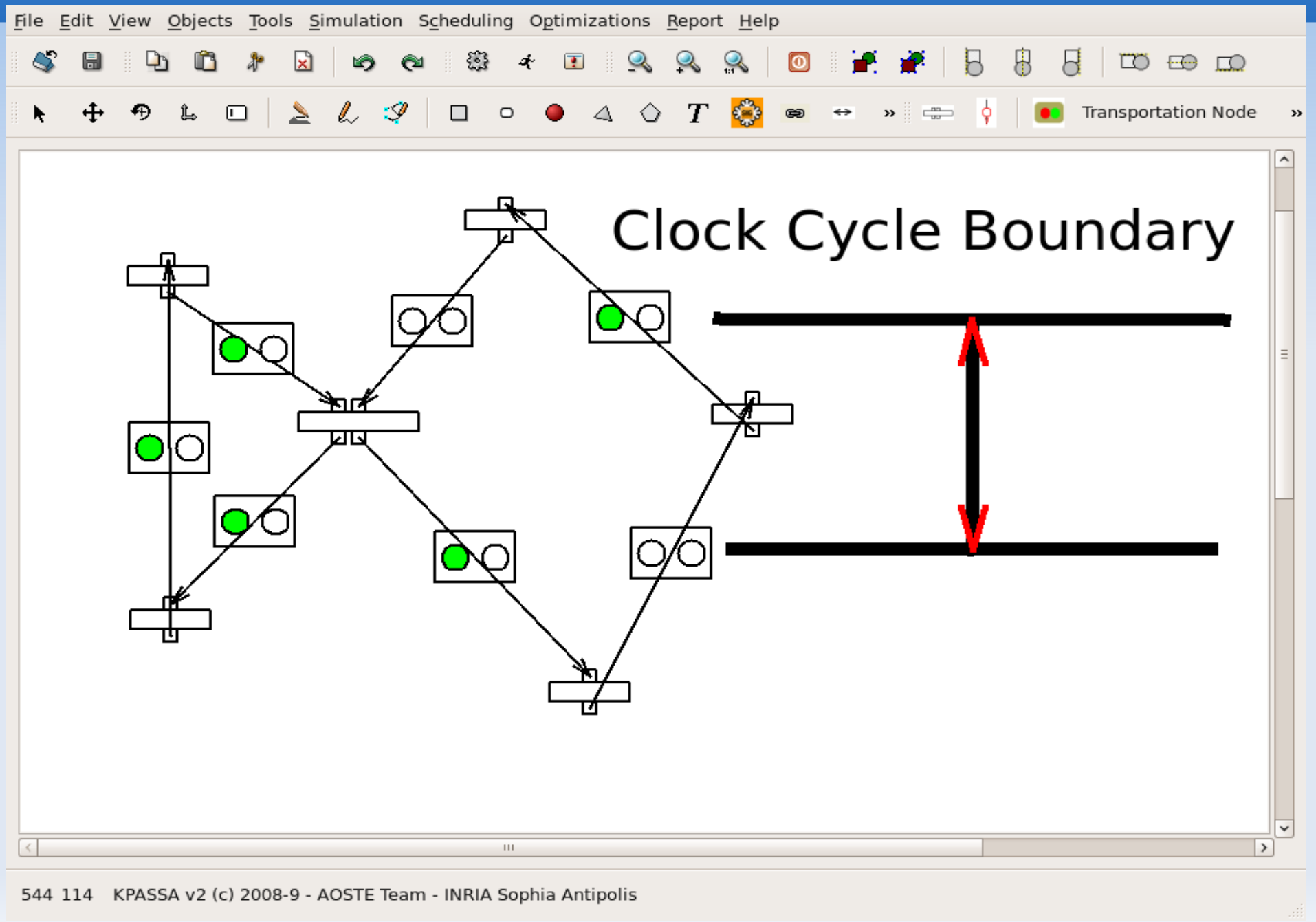
Latency... latency...

- **Moreover, there is correctness issues due to paths not having the same latencies, caused by placement and routing: data might not be « aligned »**
- We need an effective mean to cope with « synchronization » of all data at both input/output of IPs.
- The topic of this talk is to provide an implementation to cope with this correctness issue using **Latency Insensitive Design (LID)**

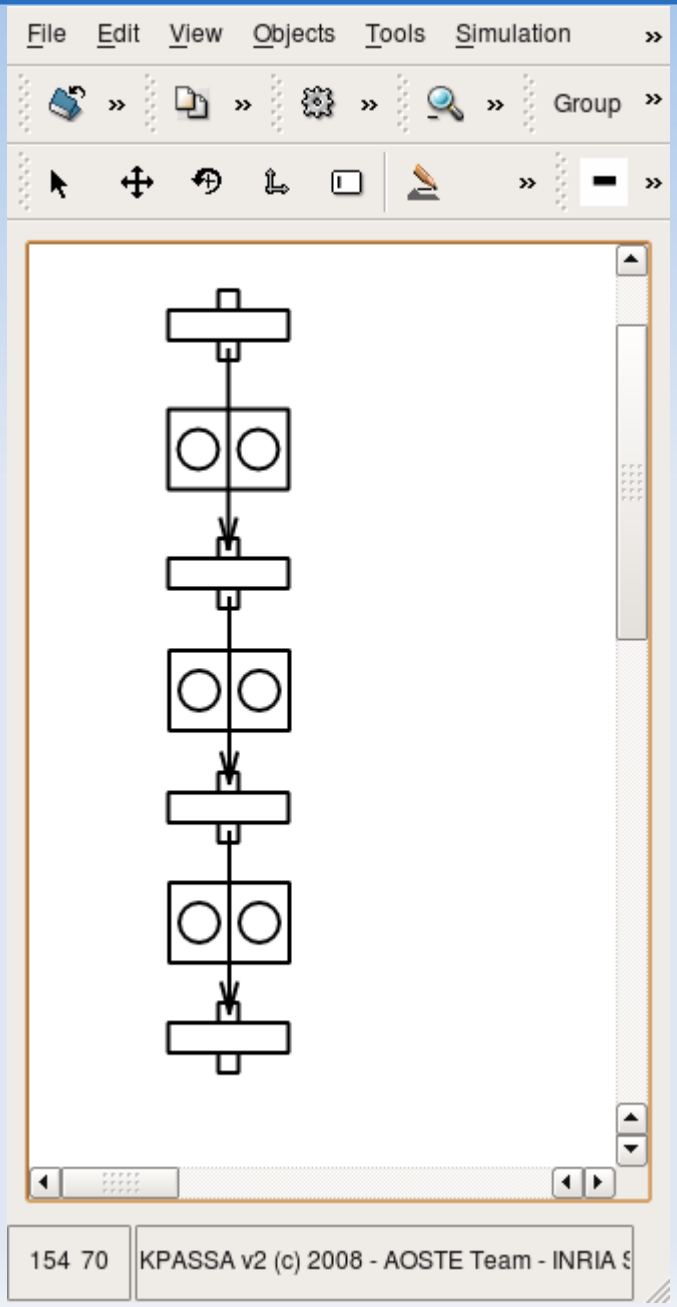
Latency Insensitive Design

- **Assume**: IP can be clock-gated in a clock cycle
- LI Protocol (back-pressure) between building-blocks
  - **Relay-Station**
    - **Smart repeater, implements part of back-pressure and wire-pipelining**
  - **Shell-wrapper**
    - **IP Clock-gating, data synchronization and implement part of back-pressure**
- **LID ensures same « behaviour » of the synchronous spec. modulo timing shifts**

Tiny Example



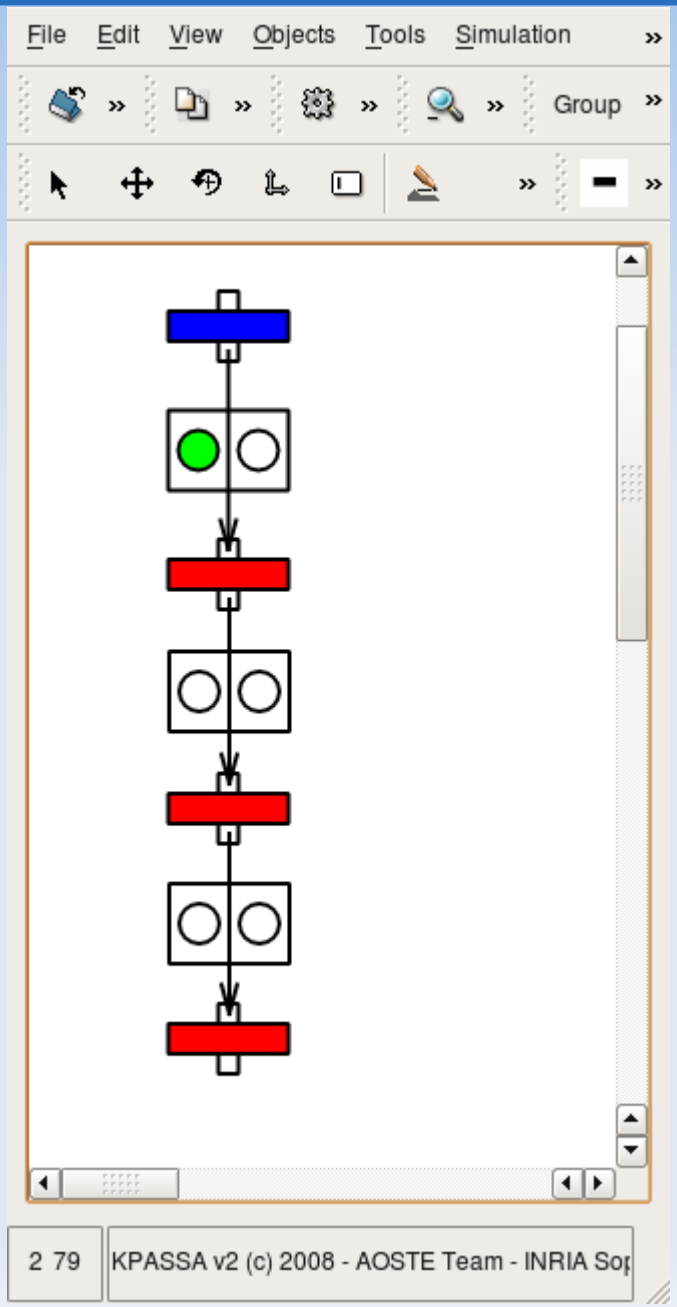
How LID works ?



- LID can be seen as both trafic lights and trafic jam congestion avoidance.
- We start with an empty chain of Relay Stations.

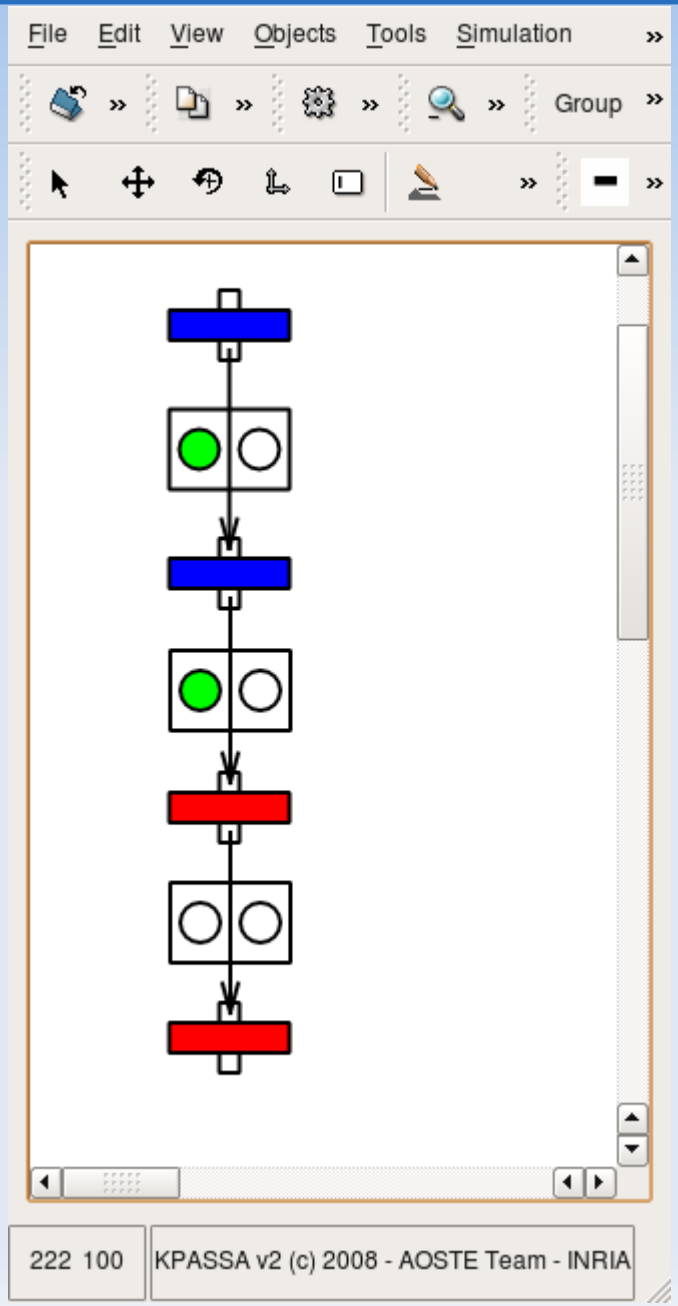
  (KPASSA2 DEMO AVAILABLE ON REQUEST)

How LID works ?



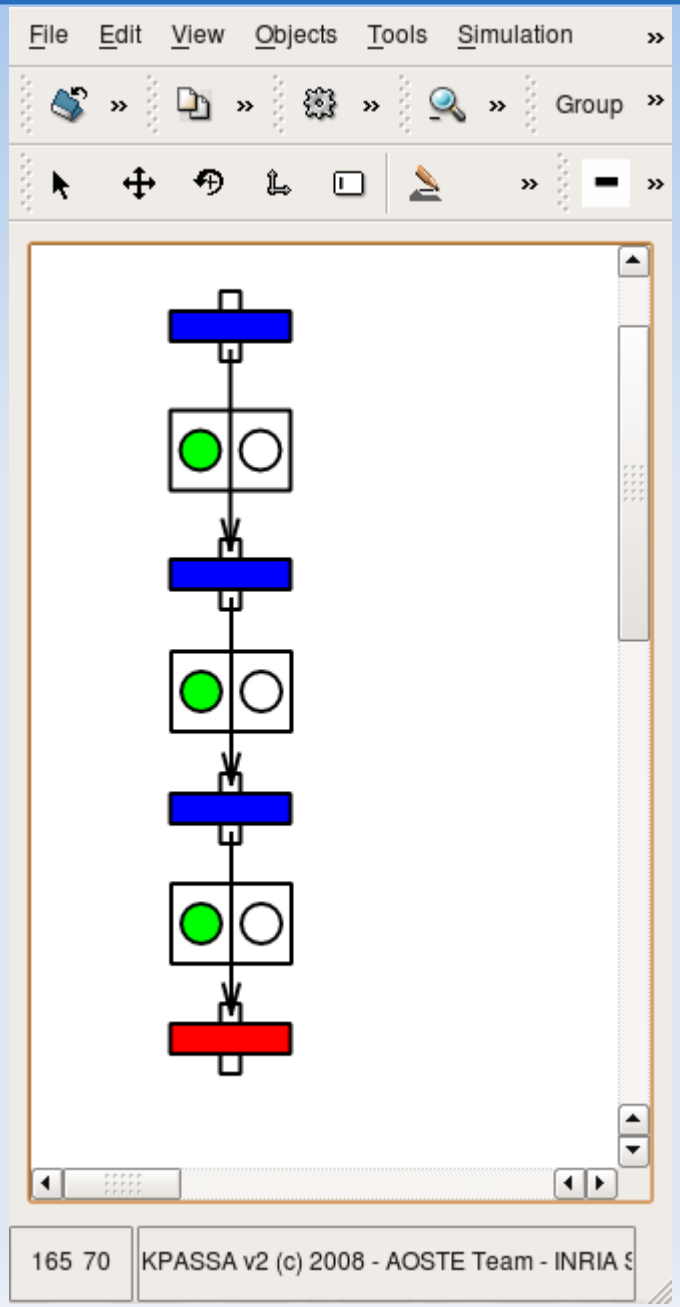
- We get an input token.
- Blue denotes execution, while red denotes stall of computation nodes.
- Green Circle denotes a token in regular buffer of RS.

How LID works ?



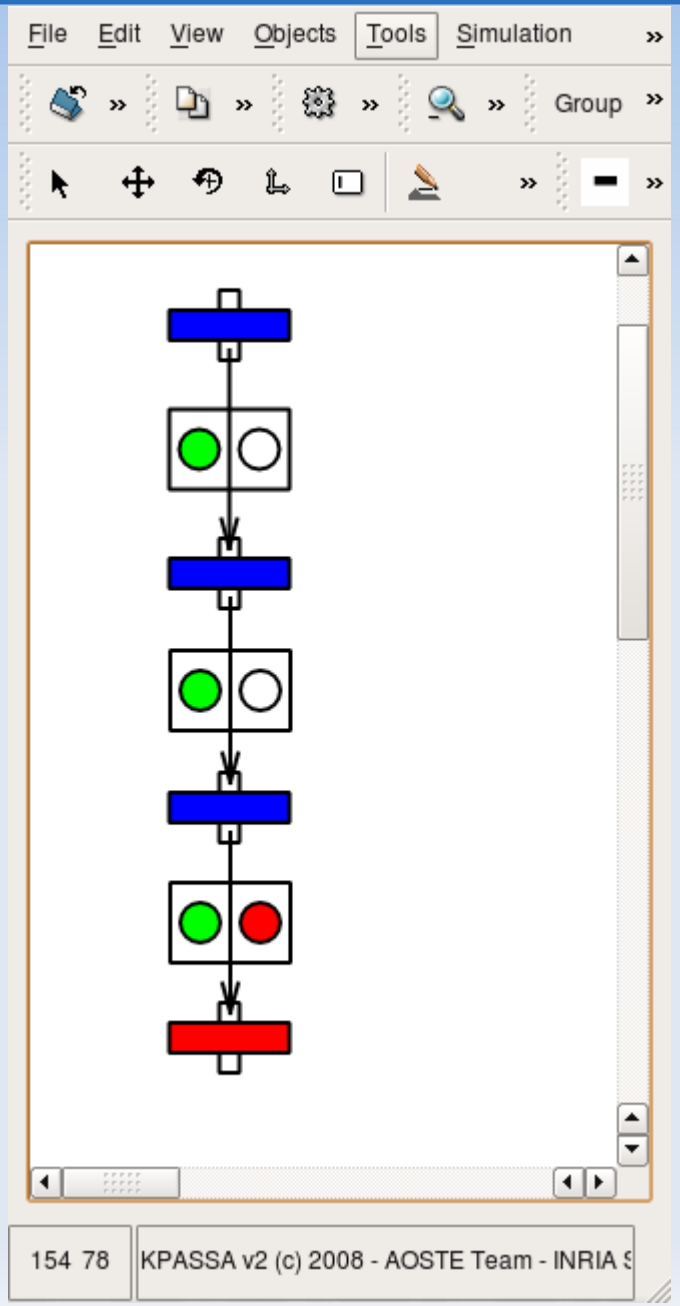
- We get another input token.
- There is no back-pressure down-side.
- So we just do **regular wire-pipelining** (store and forward).

How LID works ?



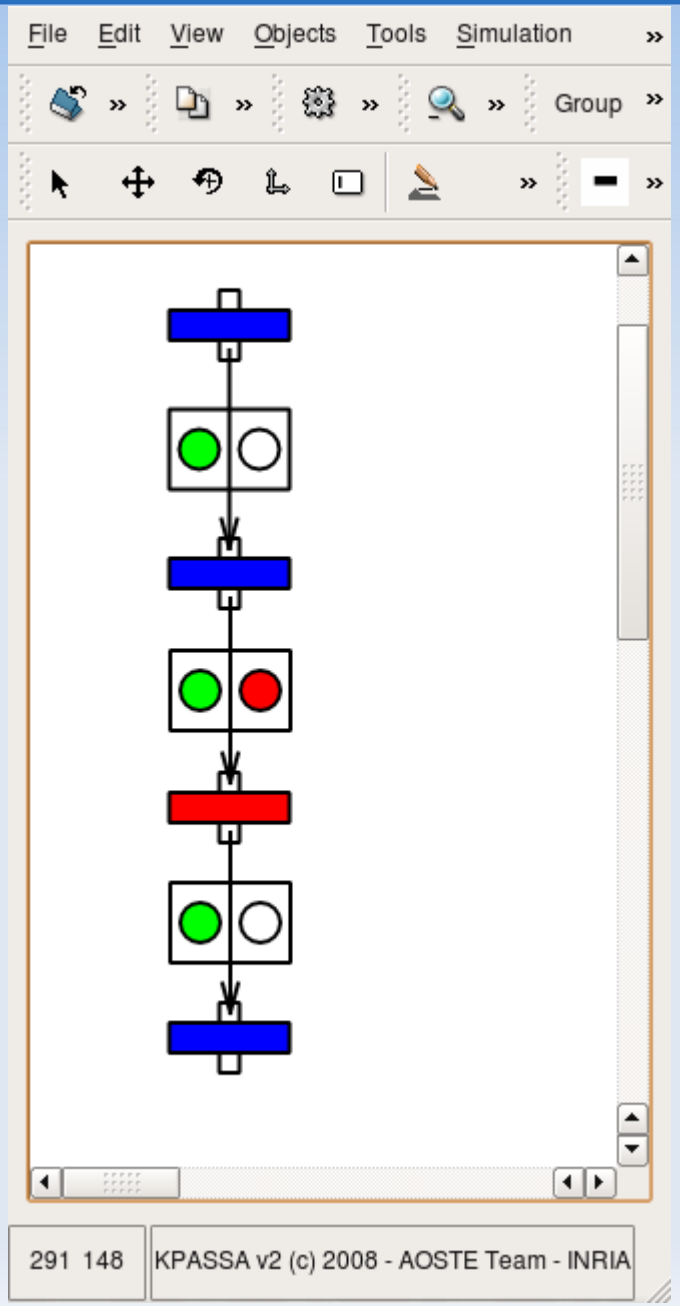
- Yet another input token, regular wire-pipelining mode.

How LID works ?



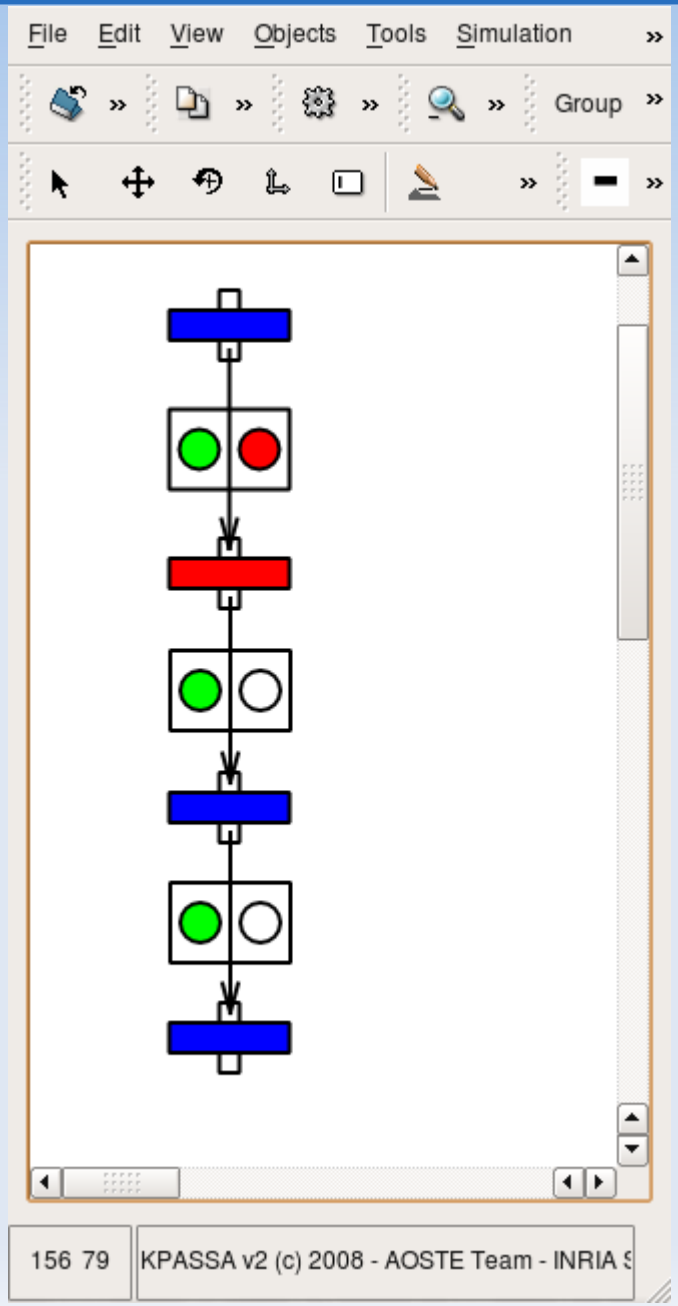
- Now, there is **congestion ahead**.
- The downward RS as to get the previous input token and the new arriving one, denoted by the Red Circle in the RS.

How LID works ?



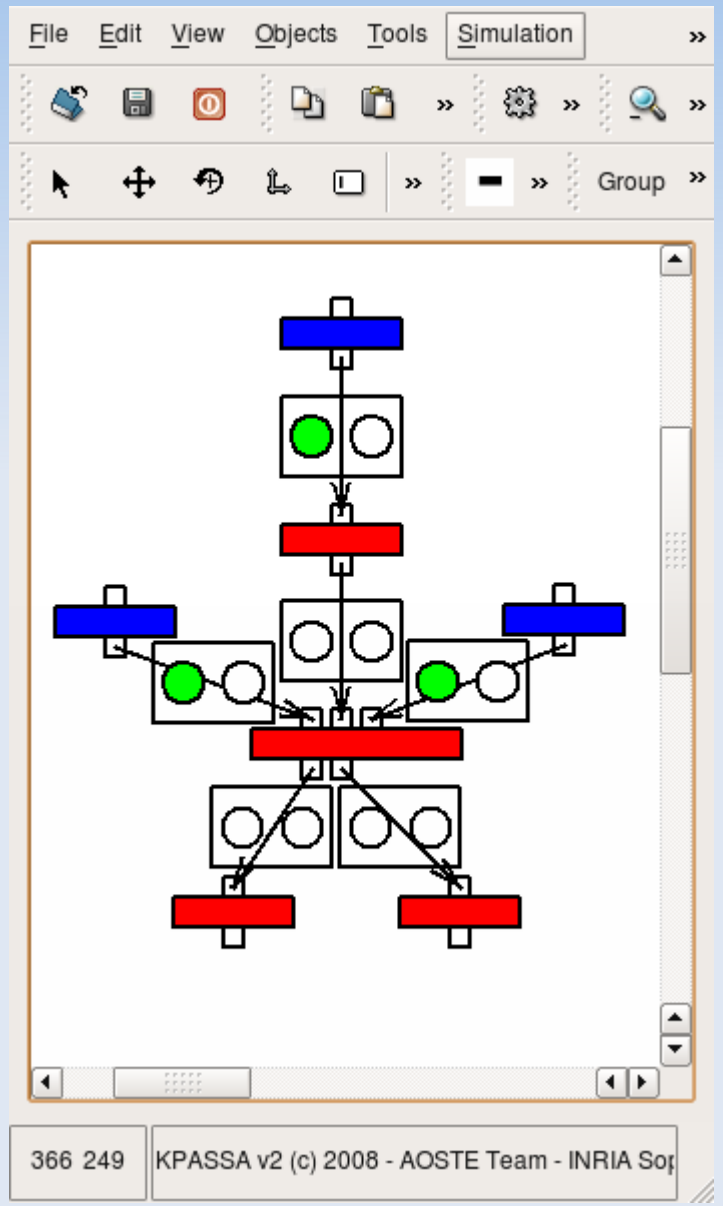
- Now, the downward computation node is ready, and consume the oldest data.
- While at the same time back-pressure is moving upward in the chain.

How LID works ?



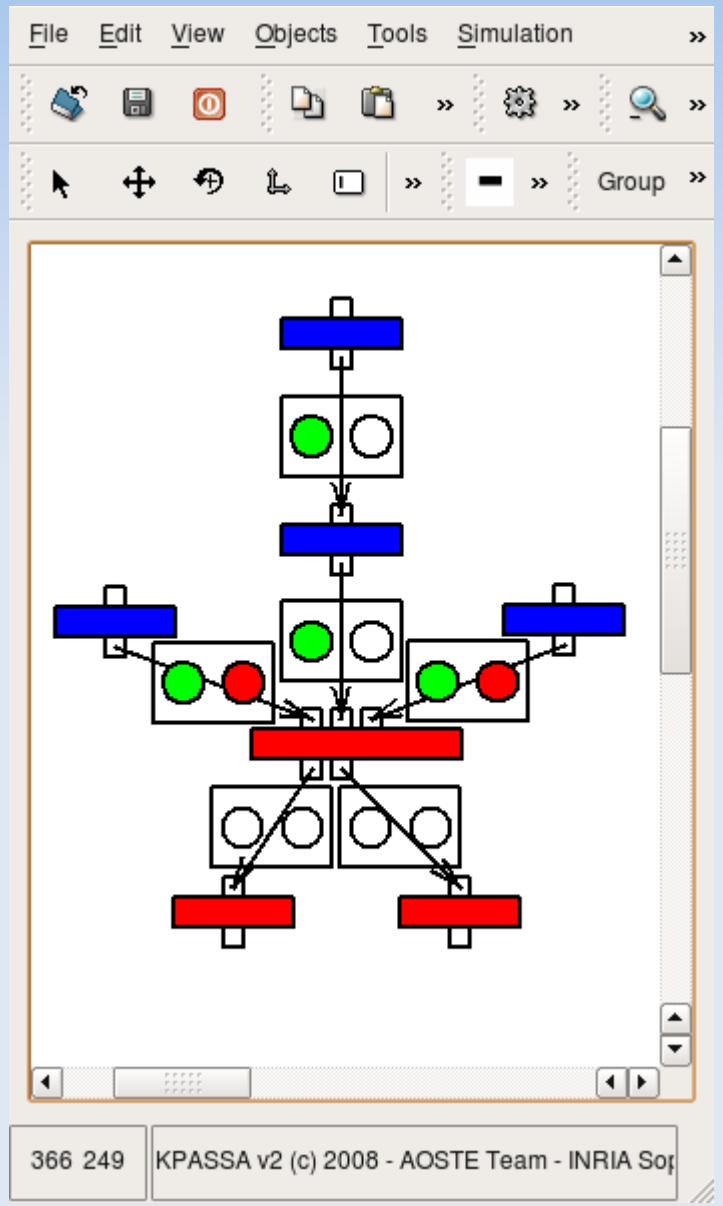
- Yet another back-pressure step.

How LID works ?



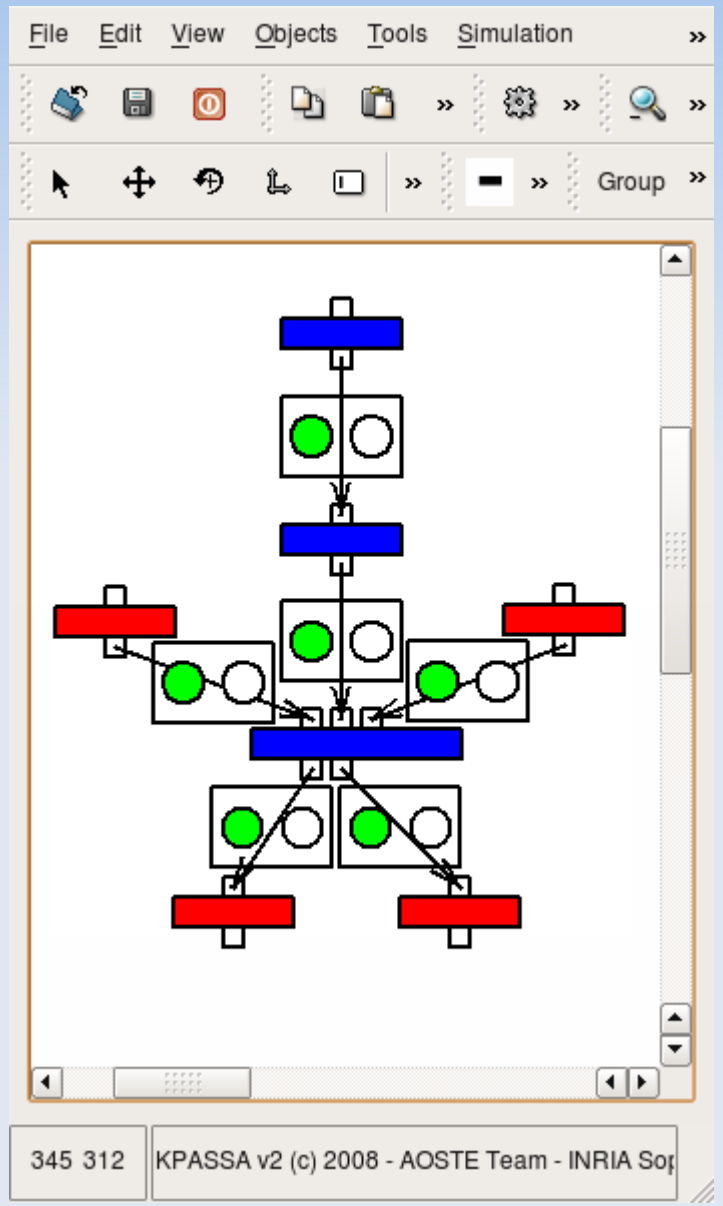
- Example to handle more than one input/output: the **Shell**.

How LID works ?



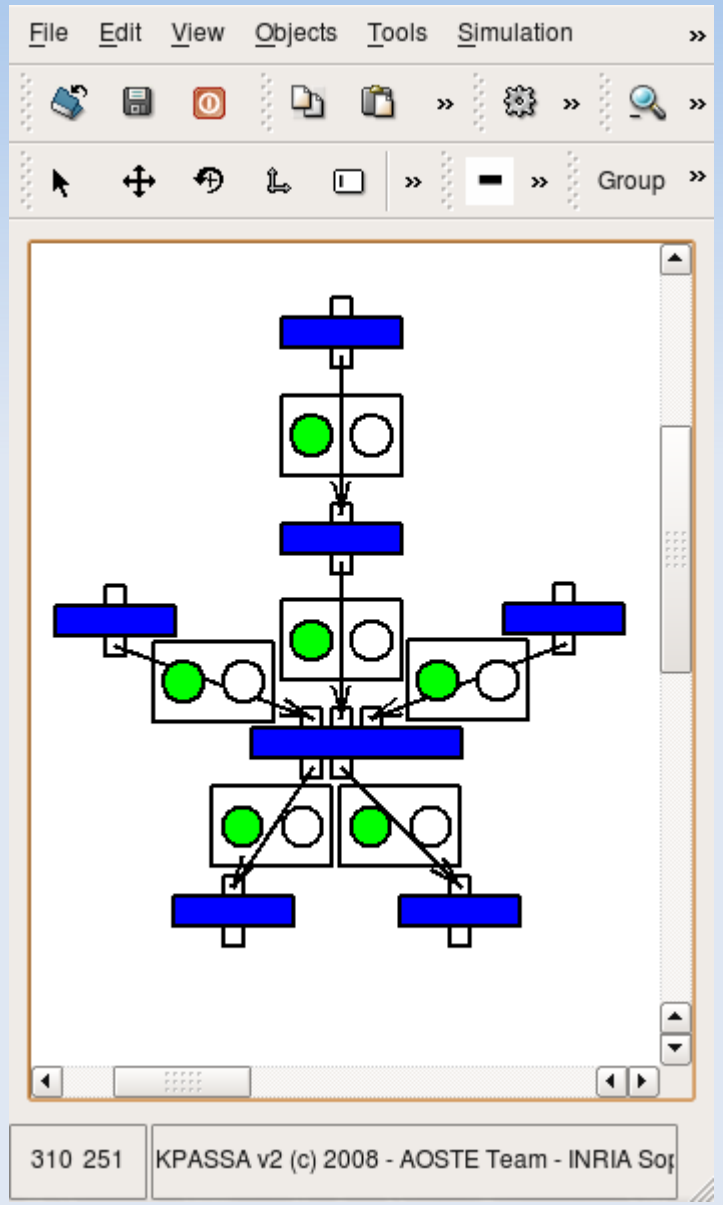
- Example to handle more than one input/output: the **Shell**.

How LID works ?



- Example to handle more than one input/output: the **Shell**.

How LID works ?



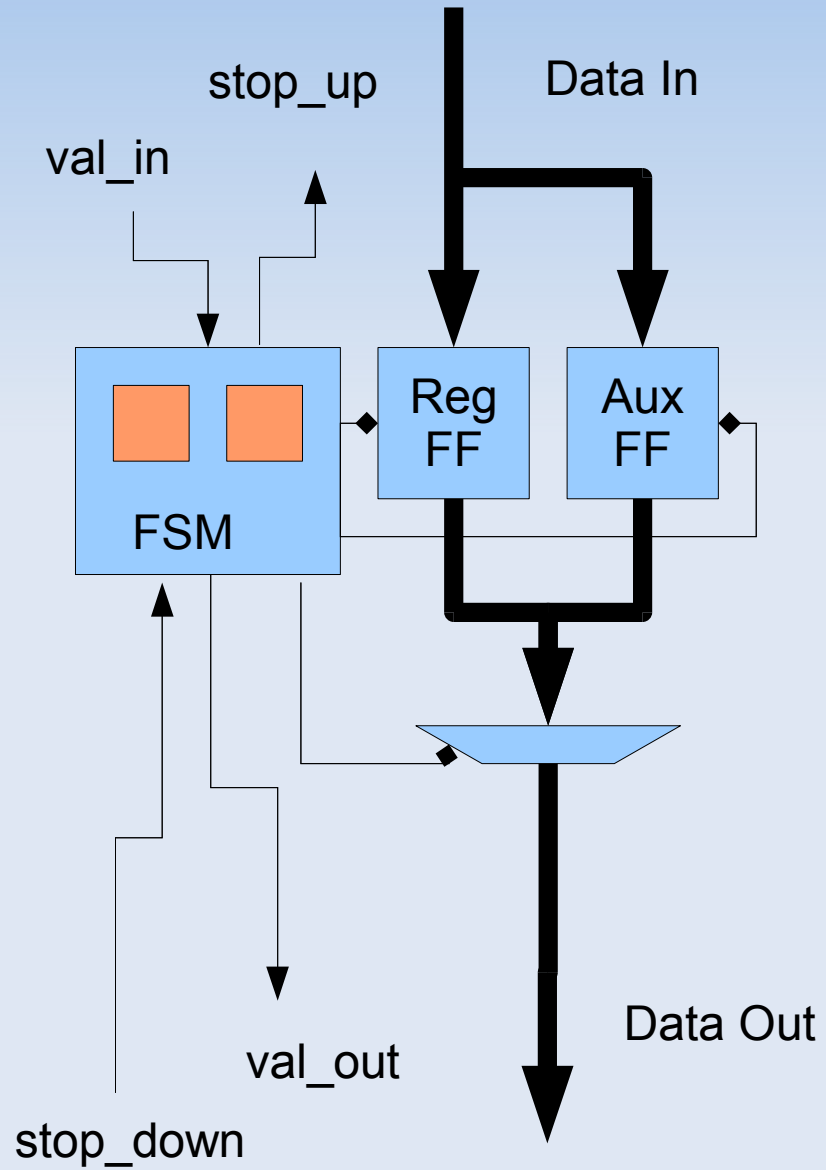
- Example to handle more than one input/output: the **Shell**.

LID in a Nutshell

- **Latency Insensitive Design**
  1) Start from a Synchronous Design where Timing Closure cannot be reached due to « **long wires** ».
  2) Encapsulate each IP with a **Shell-wrapper**.
  3) Insert **Relay-Stations** (smart buffer) on long wires.
  4) Place and Route, iterate on 3) until reaching Timing Closure.

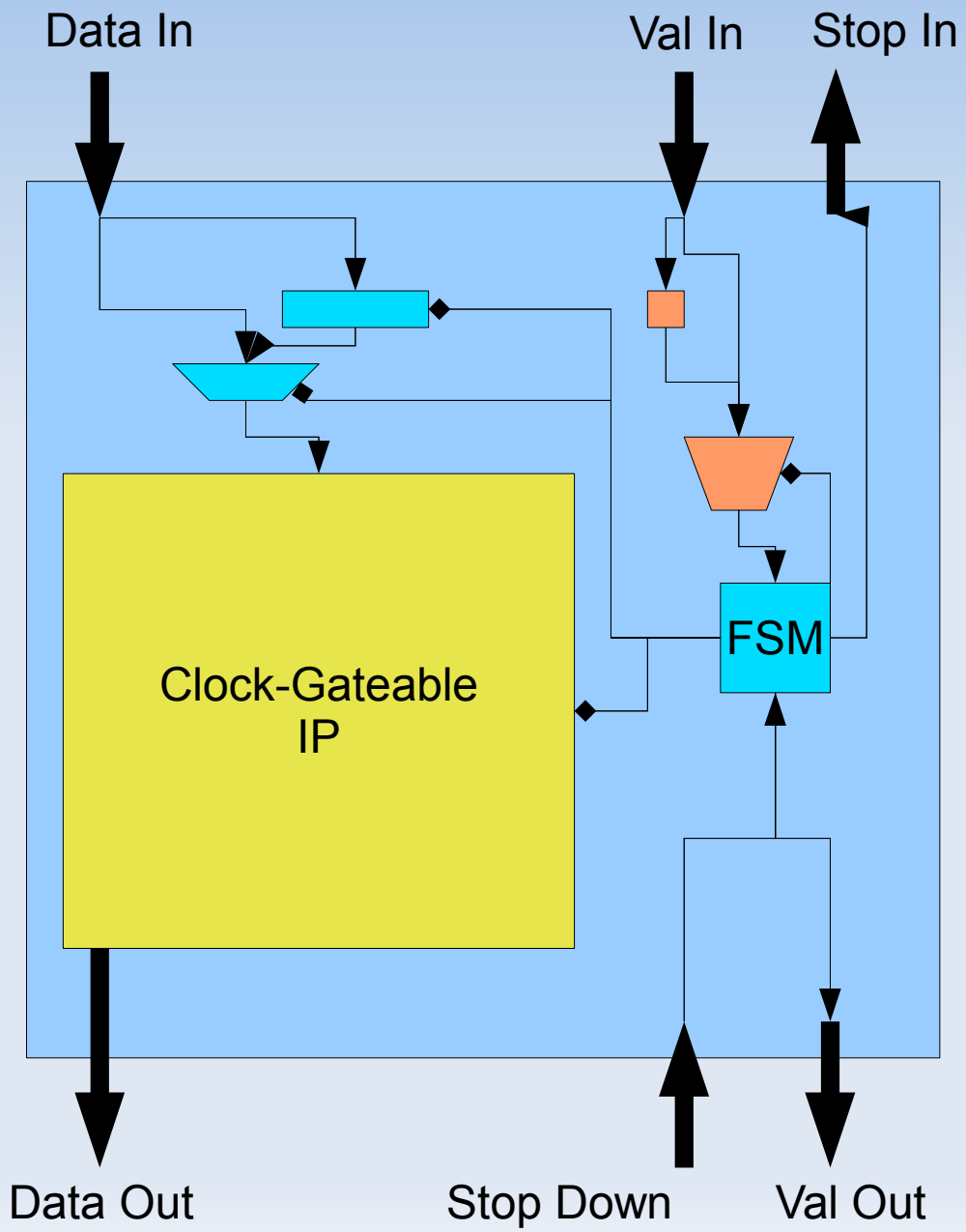
Let's go in the details of RS and Shell

Regular LID Implementation: RS



- Relay-Station
  - 2 FFs for Data
  - 2 FFs for Control
- We cannot remove any FFs without affecting the throughput, or correctness
- **Once data sent, no more present in any FF**

Regular LID Implementation: Shell



- Shell
  - 1 FF for each Input for Data
  - 1 FF for each Input for Control
- LID does not come for free.
- There is an area overhead, we want to minimize this

Our Idea

- **Problem**: once RS sent data, no more data in RS. Bypass buffers needed to take token(s), if the Shell does not execute the IP.
- **Data present in each RS, how to remove costly bypass buffers in the Shell ?**
- **The Trick: Modify behaviour of input RS of each Shell.**

  Repeat sending the same value kept inside until accepted by the Shell → **Repeat Relay Station (RRS)**
- Shell → **Fusion Shell with RRSs**

The Intuitive Way

- *Quiet* Child asks its Parent for something, while she/he is busy.
  The child waits *silently*.

  → Relay-Station

  - Hypothesis parent has memory.
- *Not so Quiet* Child asks ALL the time and annoys its Parent for something, until he/she gets it.

  → Retry Relay-Station

  - Hypothesis parent has not memory.

Retry Relay-Station

- « *Cook until done* » principle
- No change to data-flow of RS
- Less area overhead (ASIC)
- Simple change to the FSM (transitions):

  send oldest data if present whatever the FSM state, until accepted.
- Implements ASAP firing rule as in regular RS
  → no change to the throughput in this case

Fusion-Shell

- Only combinatorial logic for clock-gating logic:
  - &(val_in$_j$) & ~|(stop_down$_i$)
- No more any bypass on both control and data flow.
- Needs all inputs to be RRS for correctness.
- Less Area

Possible Throughput Degradation

- When ?
  - All data ready, back-pressure, input RRSs full
    → send backpressure on all channels at next clock cycle.
  - For regular LID depends on Shell implementation. Our implementation does not send backpressure and stores incoming tokens if bypasses are empty. But if next clock cycle we receive backpressure, we will send backpressure after.

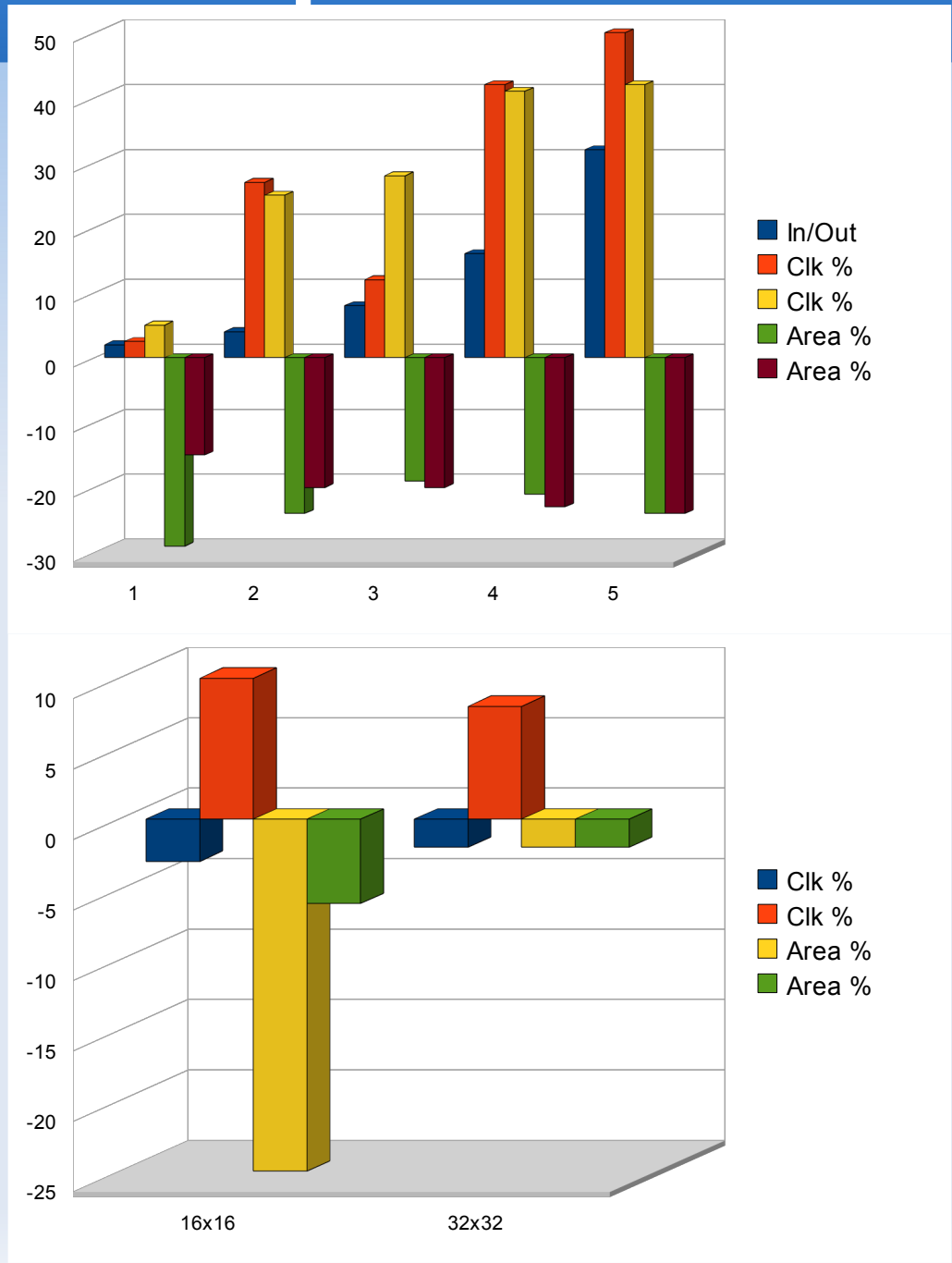
    RS + Shell in such case, means having RS with a capacity of 3.
- Check this through simulation or analytically

Results

- 2 Basic Tests
  - **Scalability** of control-paths between Shell-implementations versus number of inputs/outputs
  - Dummy data-path (MUL16,MUL32) to show area gain of the implementation
- FPGA & ASIC Results
  - FPGA: Xilinx Spartan3/Virtex5, ISE 10.2.1
  - ASIC: FreePDK 45nm from North Carolina SU & Oklahoma SU, Clock Rate 1Ghz, Synopsys DC

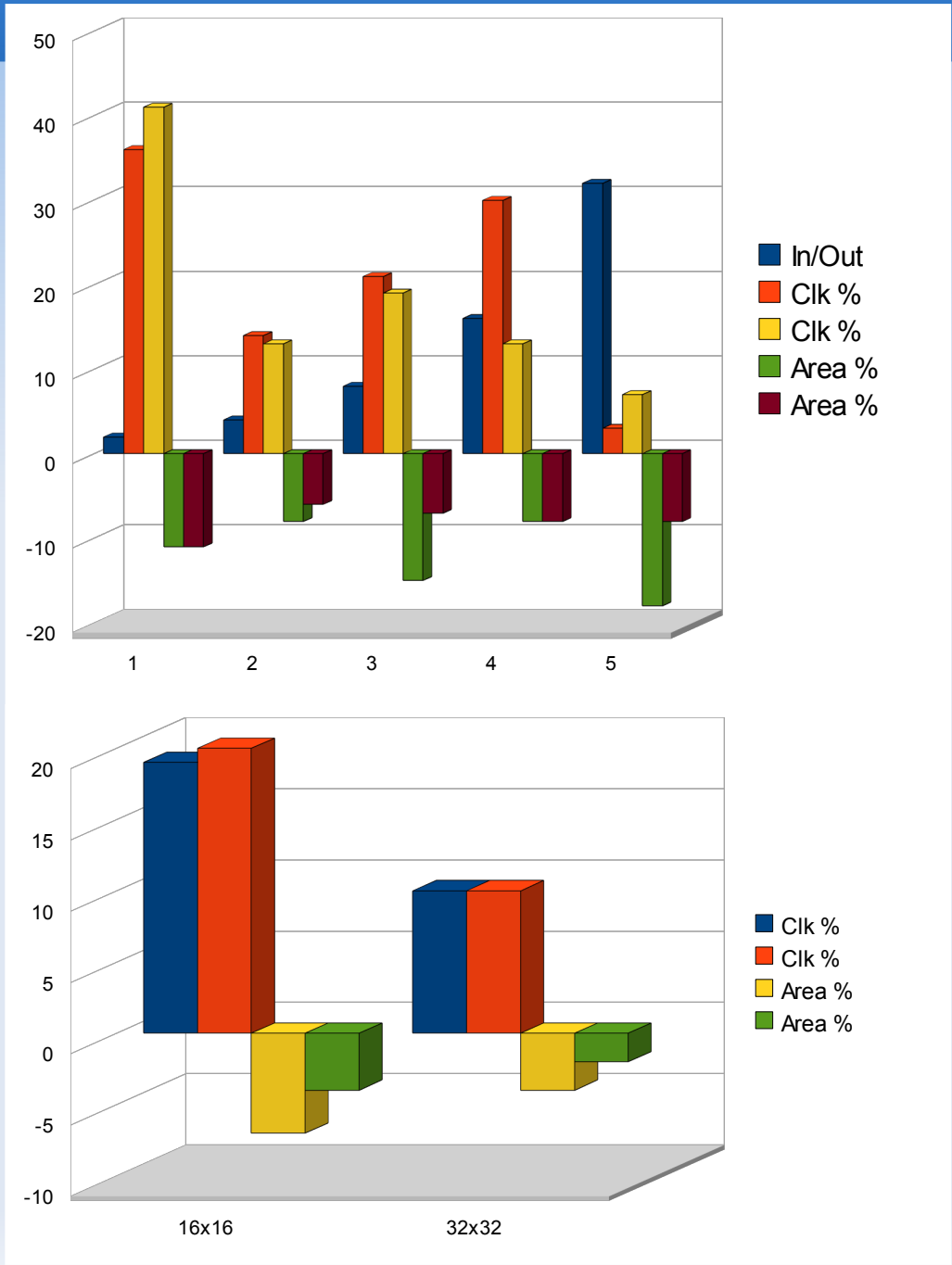
Results for FPGA: Spartan3

- Better Scalability
- Area as small or smaller
- Clock rate as fast or faster
- Power smaller
  - Both Dyn/Sta for V5.
  - Not for S3. (Big Sta.)



Results for FPGA: Virtex5

- Better Scalability
- Area as small or smaller
- Clock rate as fast or faster
- Power smaller
  - Both Dyn/Sta for V5.
  - Not for S3. (Big Sta.)

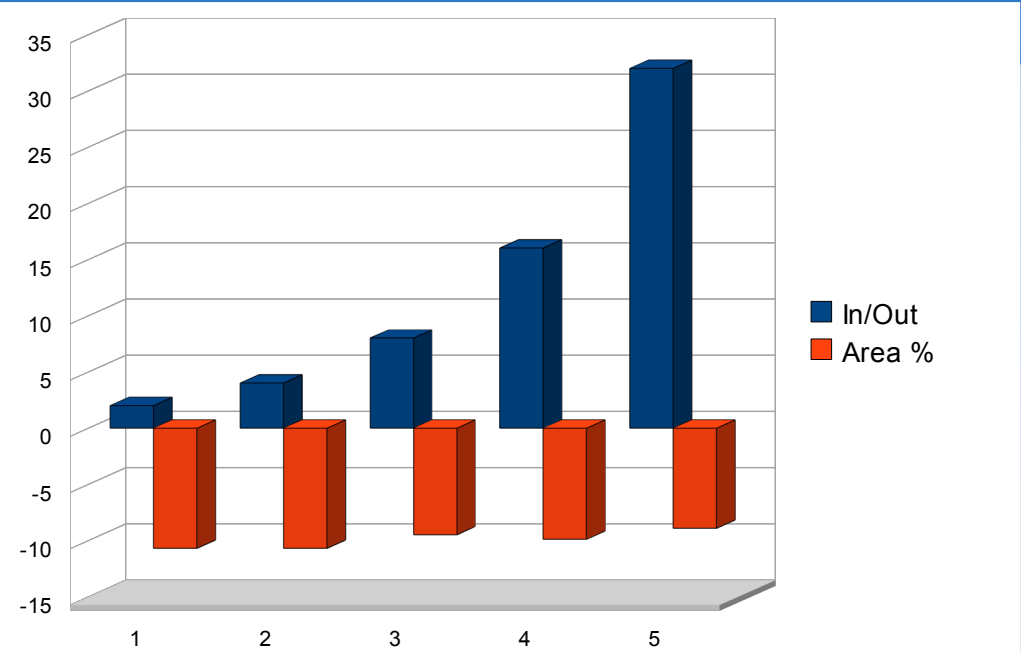


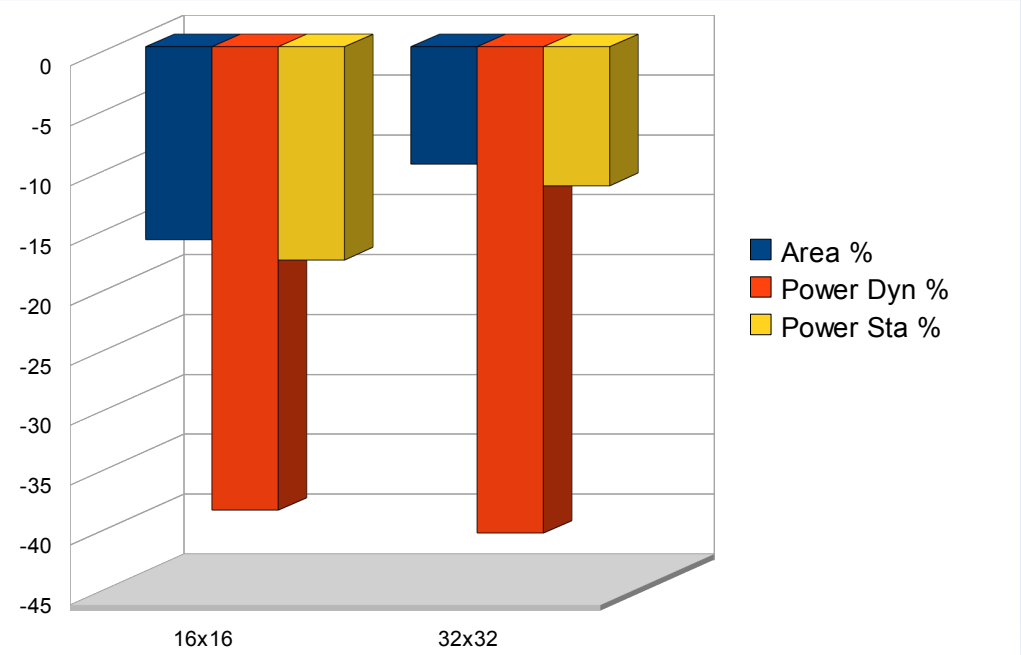
Results for ASIC

- Better Scalability
- Smaller Area
- Higher Clock Rate
- Power smaller
  - Dynamic: Less Ffs
  - Static: Less gates

But, our Design Kit does not have clock-gating cells...

Bart is energy consumming, so use him with care





Critical Assement on RRS + Fusion Shell

- You Should Use if
  - Area Critical
  - OK for Throughput Requirement
  - Still Margins in Dynamic Power
  - For instance, no dynamic power change when creating LID pipelines
- You Should Not Use if
  - KO for Throughput requirement
  - Creates Dynamic Power hotspot

Conclusion

- New Implementation for LID
  - Fusion Shell (no more any bypass buffers)
  - Retry RS: same or small area as RS, little protocol change (repeat sending data until accepted)
  - ***Backward compatible with Regular LID Implementation***
- Less Area, Equal or Faster Clock
  - Power to handle with care
  - Verilog Code available uppon request
- **On-going work**
  - Exercise: Can do better, using latches.
  - Long Term: Need tight Integration with P&R and (S)STA tools to be really effective.