



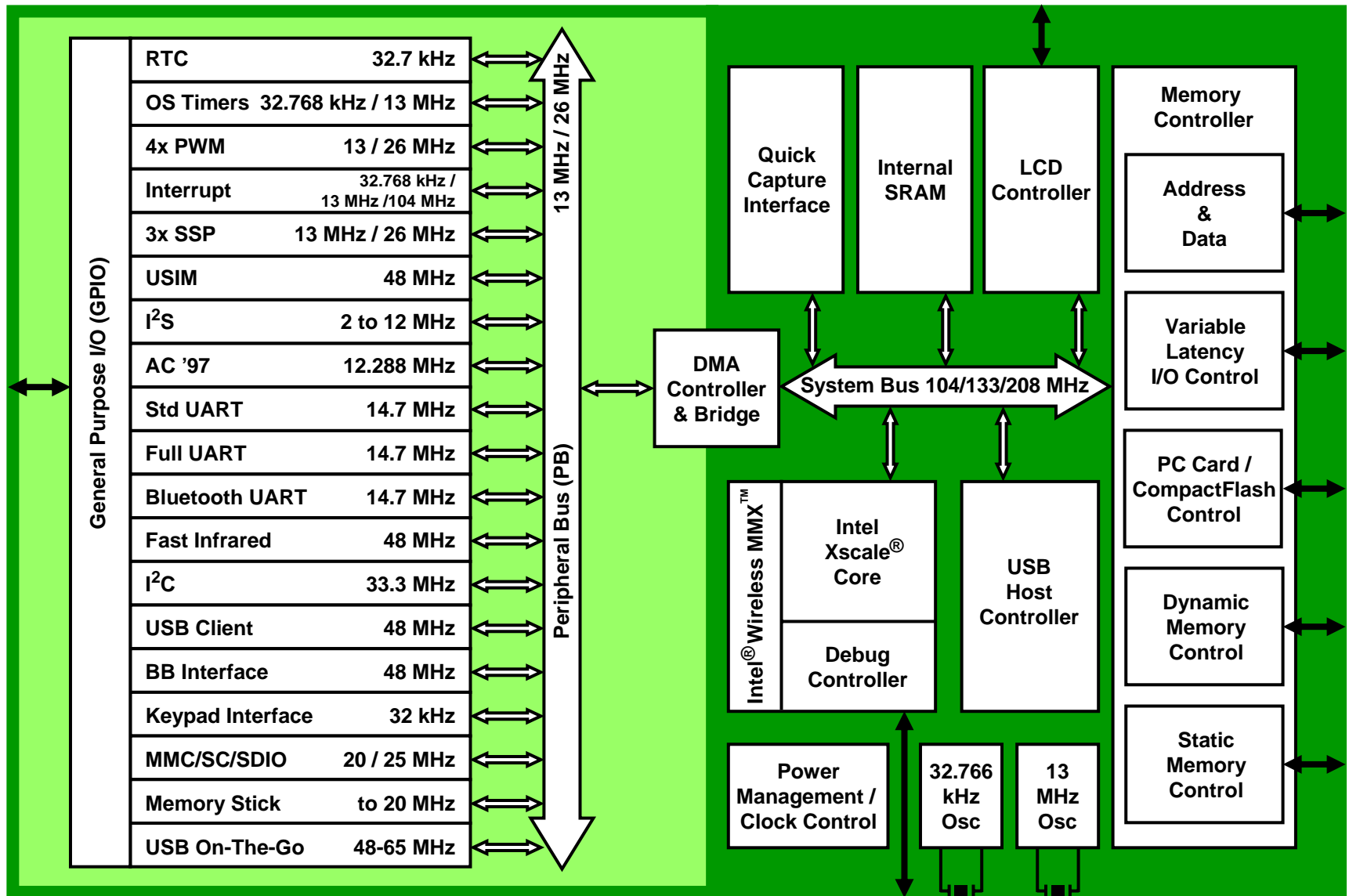
Asynchronous on-Chip Communication: Explorations on the Intel® PXA27x Peripheral Bus

Andrew M. Scott, Mark E. Schuelein, Marly Roncken, Jin-Jer Hwan
John Bainbridge, John R. Mawer, David L. Jackson, Andrew Bardsley

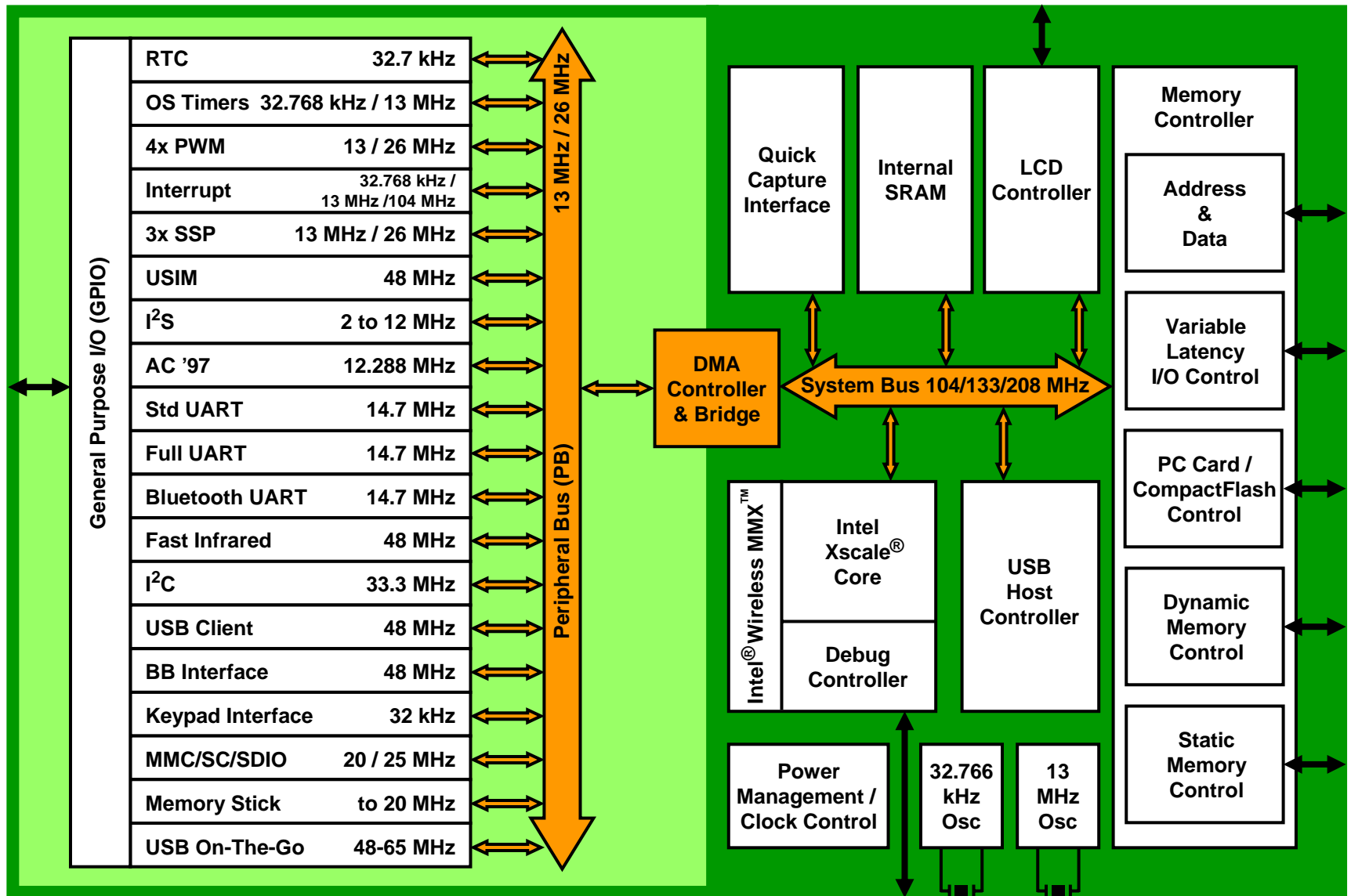


Introduction

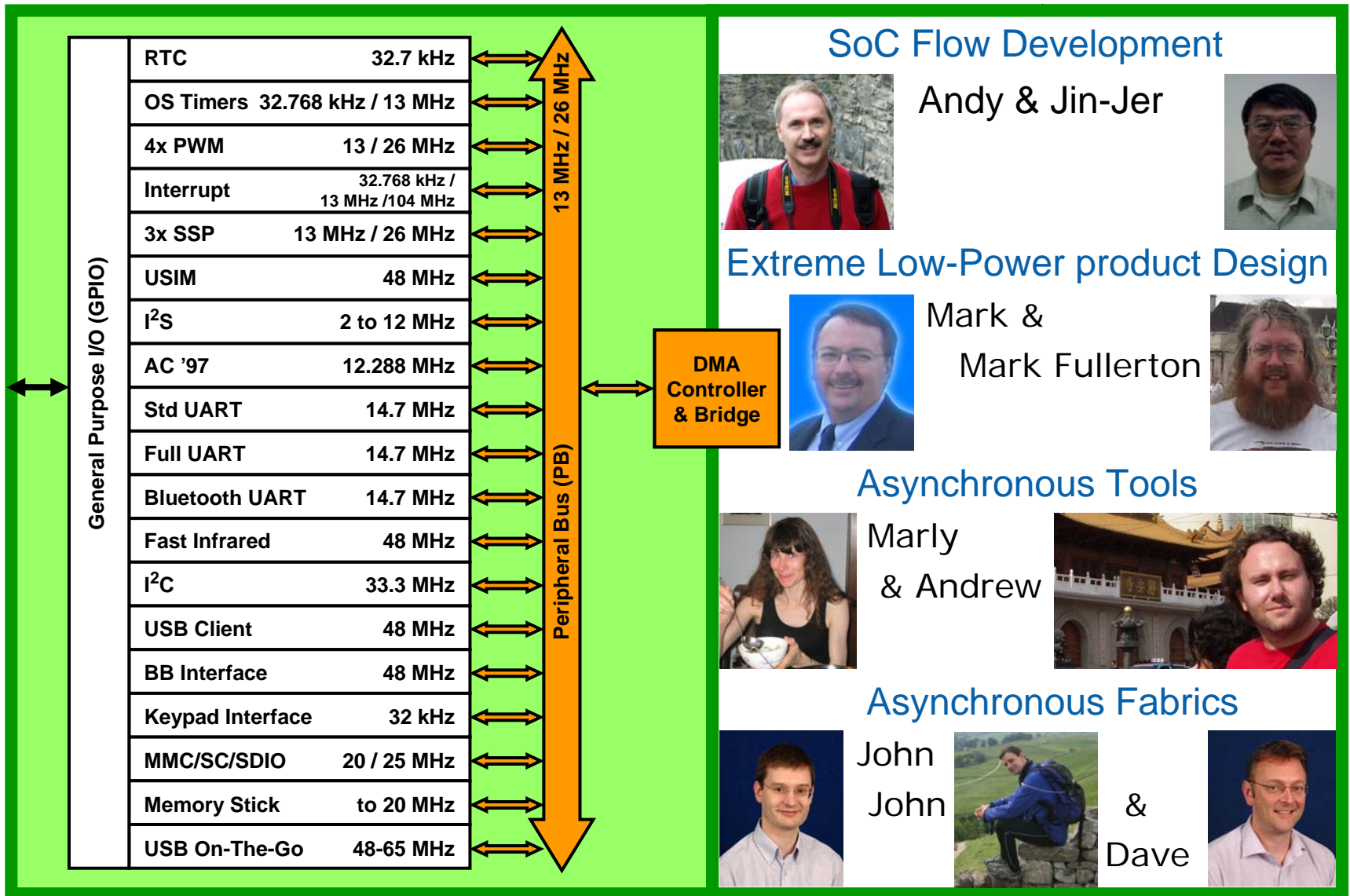
Intel® PXA27x Processor Design



Intel® PXA27x Processor Design



Async Peripheral Bus Team



Objectives

- **Build** an Asynchronous NoC in a Synchronous SoC flow
- **Assess** design tradeoffs from Product Developer's Perspective
- **Identify gaps** in current design capabilities

What do SoC Developers worry about?

- Inflexible product-introduction cycles
- Shorter product lead times & product life times
- Growing Complexity
 - Design & Manufacturing rules
 - Product & System design

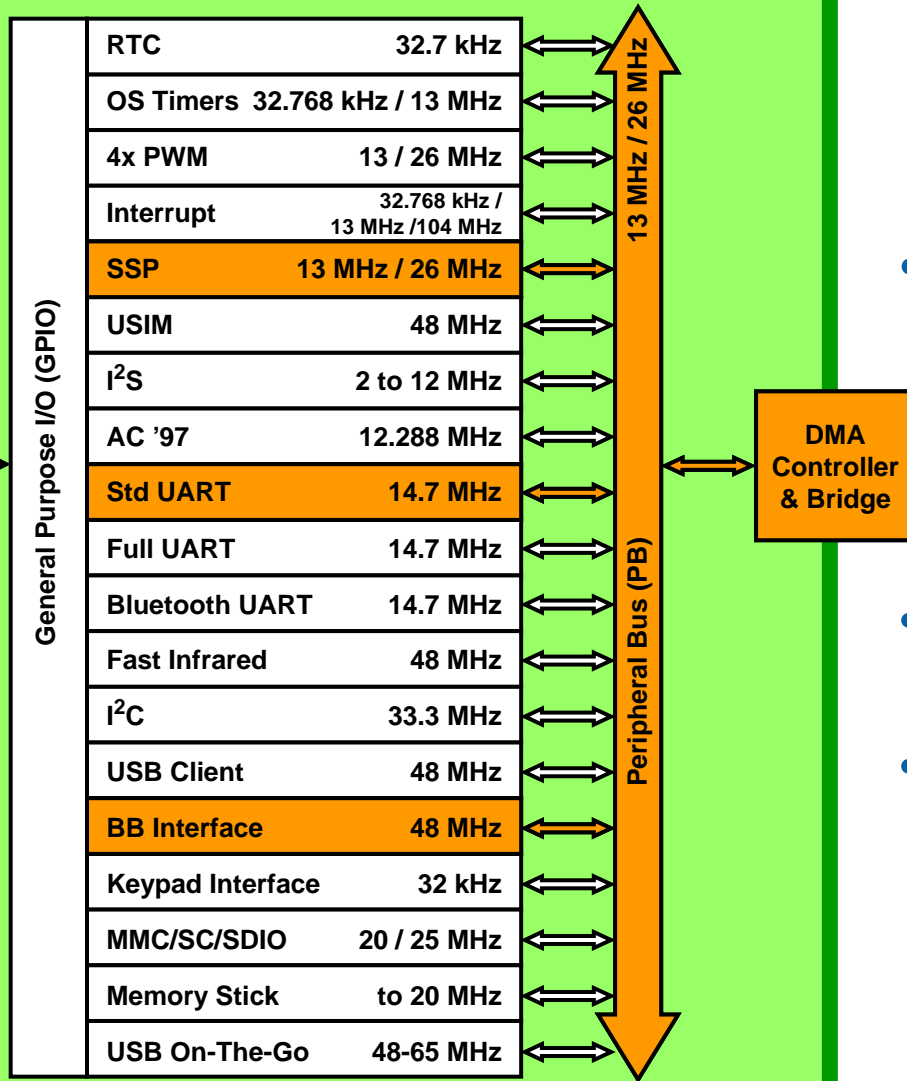
What do SoC Developers want from their flow?

- Fast integration and validation of IP
- Minimal IP and IP-collateral redesign
- Modular Design Flows
- Minimal disruption to their synchronous SoC flow



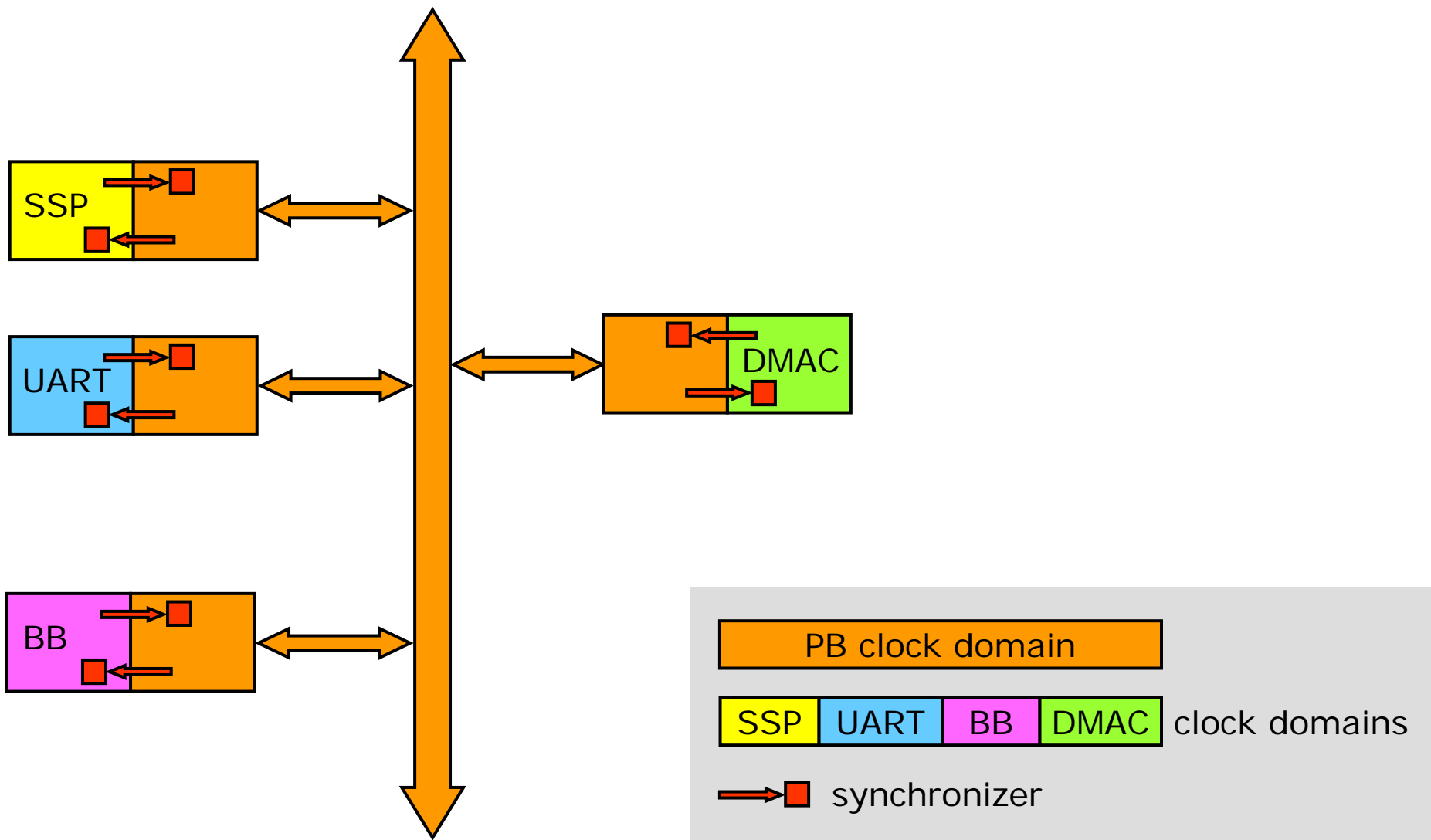
Exploration - What we did

Peripheral Bus – Baseline Design

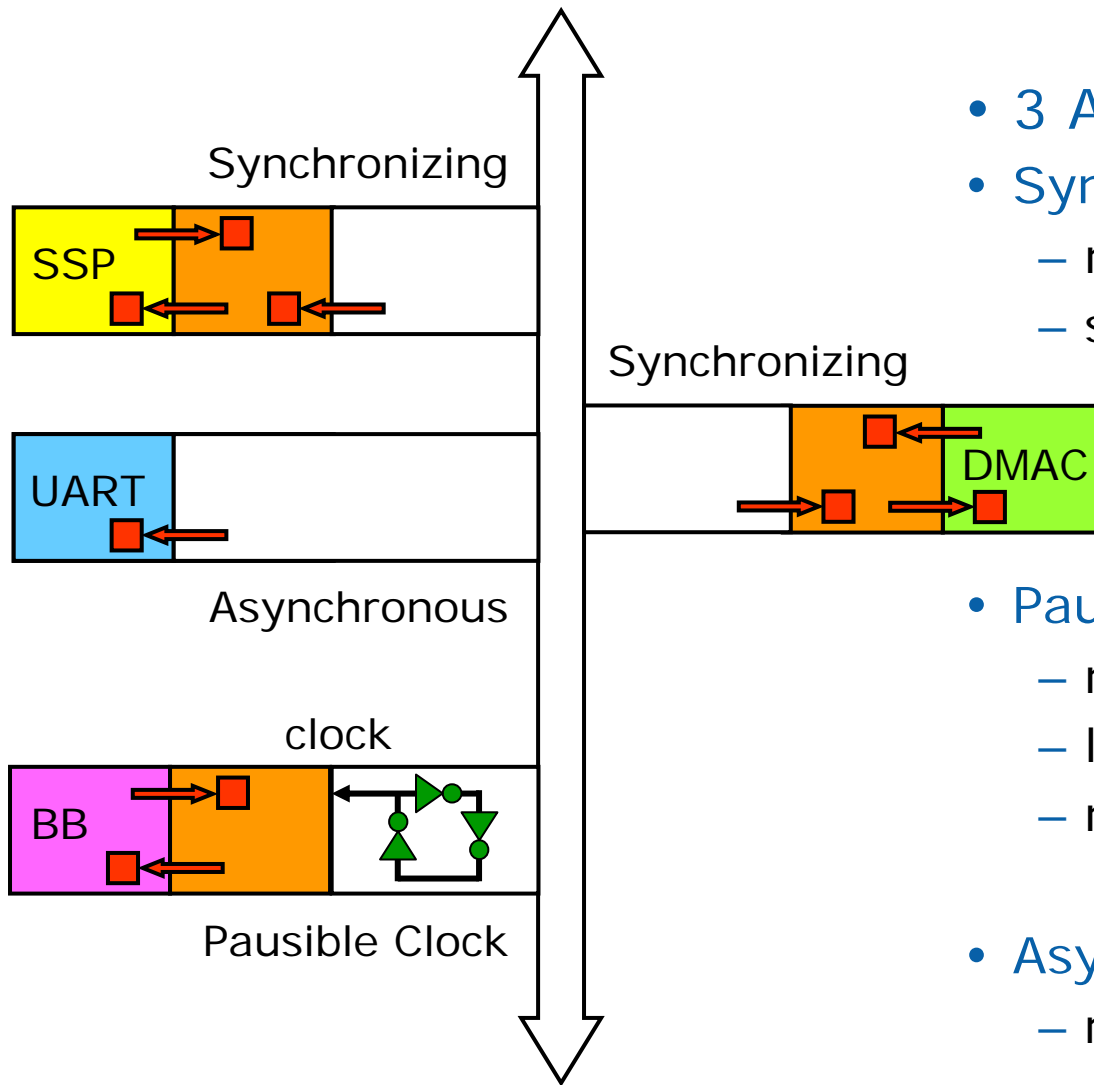


- 3 representative Bus Slaves:
 - SSP
 - UART
 - Baseband
- Peripheral Bus Fabric
- Bus Master
 - DMA Controller

Peripheral Bus – Synchronous Interface



Peripheral Bus – Asynchronous Interface



- 3 Async Interface Adaptations
- Synchronizing Adapter
 - no Master/Slave redesign
 - simplest, adds synchronizers

- Pausable Clock Adapter
 - no Master/Slave redesign
 - locally generated interface clock
 - no extra synchronizers

- Asynchronous Interface
 - requires redesign (UART)
 - removes synchronizers to PB



Transaction Level Testing

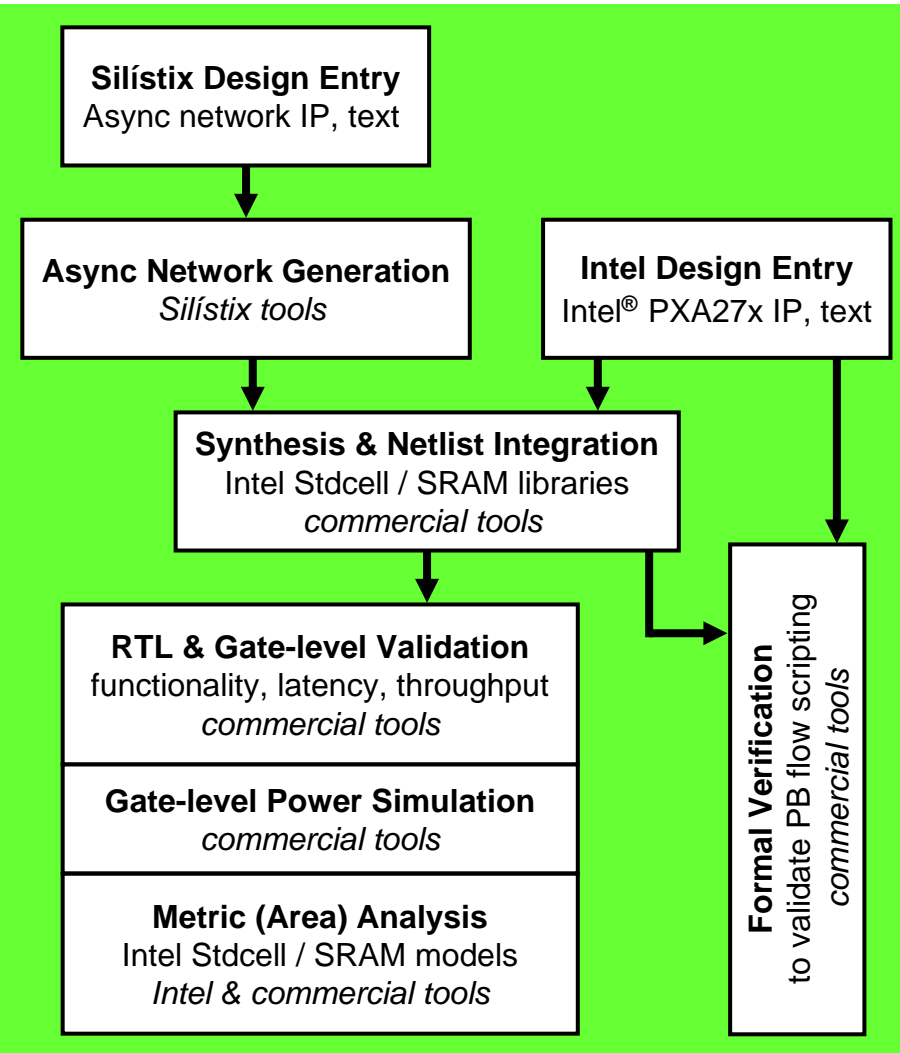
Transaction-Level Testing (TLT)

- *Test scope*
 - Functional coverage
 - Stress & Error Conditions
 - Multiple Use Models & Traffic Scenarios
 - Peripheral, Subsystem, and System Level
- How?
 - Specify Transactions
 - Automatic Protocol Adherence and Results Checking
- Strengths
 - Test Re-use at Peripheral, Subsystem & System level
 - Test Re-use for Synchronous & Asynchronous
 - Facilitated abstraction to higher-level traffic patterns
 - AND HENCE:
 - Highly portable & powerful !!!



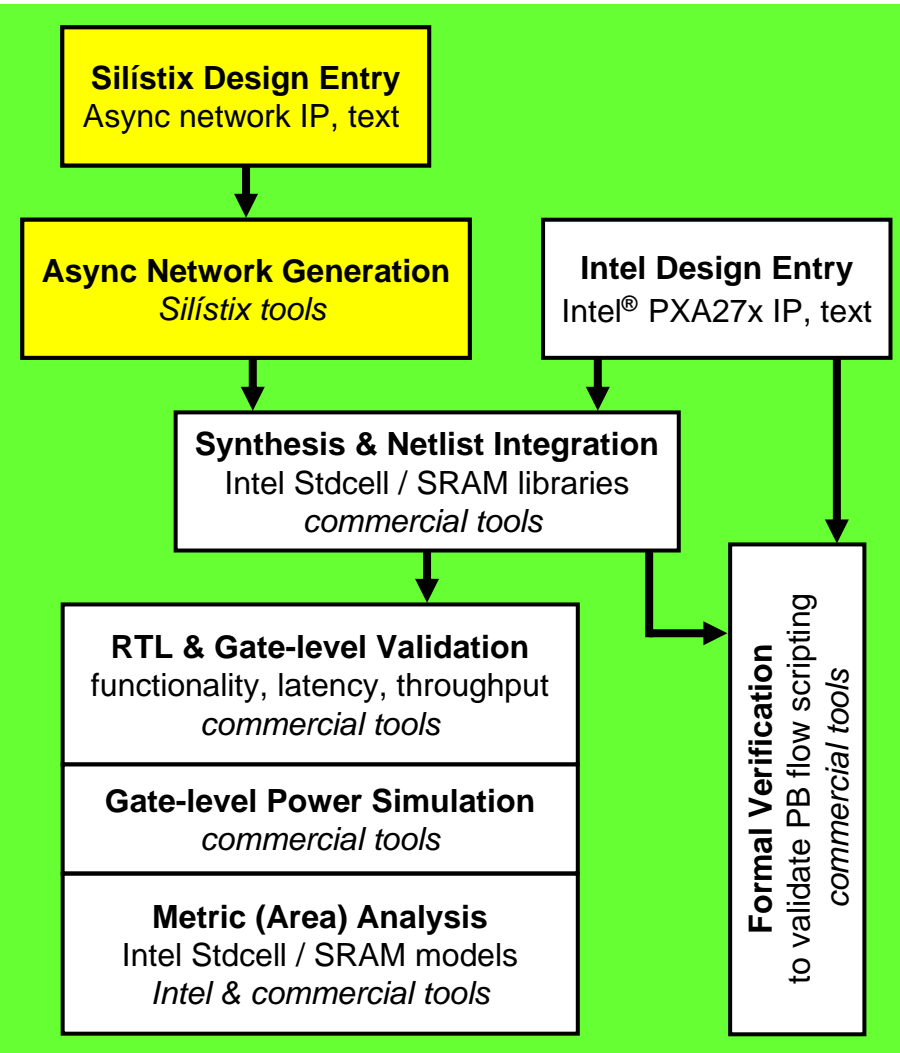
EDA Flow & Network Construction

Scope



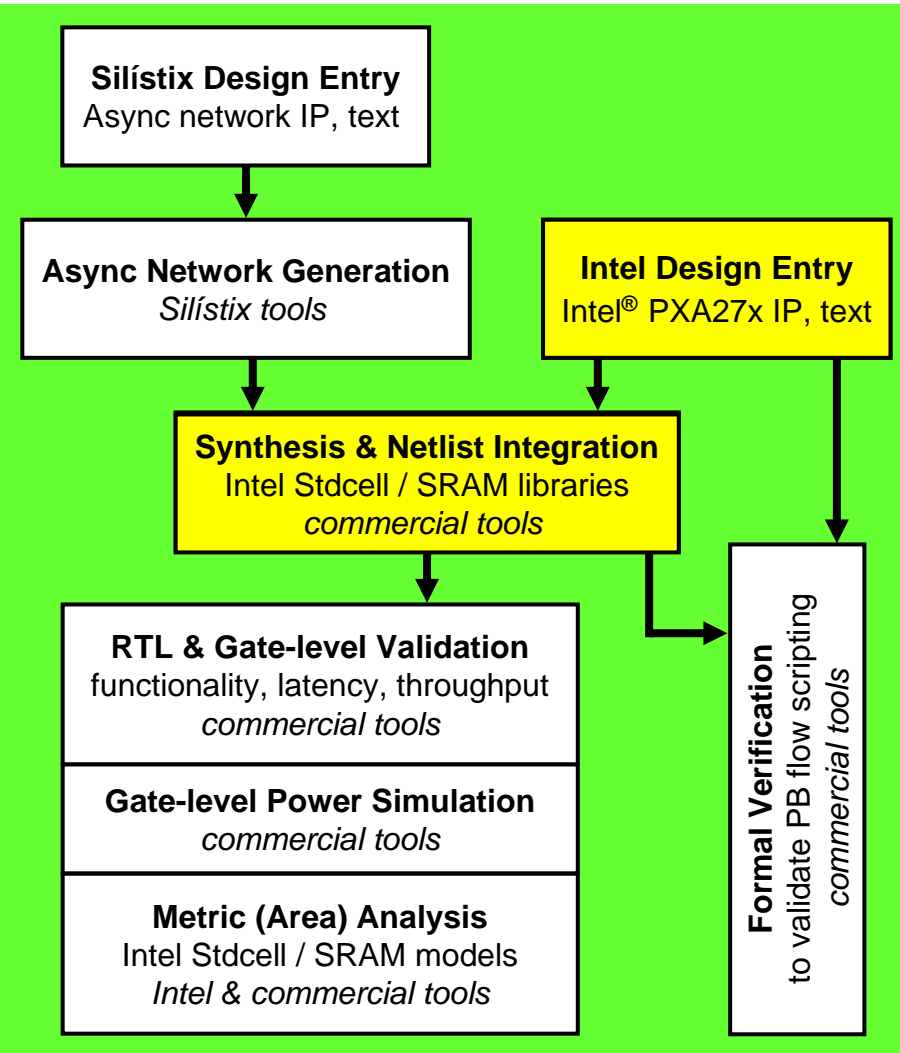
- *Build* 5 representative Synchronous & Asynchronous top-level networks
- *Evaluate*
 - Functionality
 - Timing & Realistic Power
 - Metrics
- *Assess* Asynchronous Design & EDA Flow integration issues

Silistix Design Entry & Asynchronous Network Construction



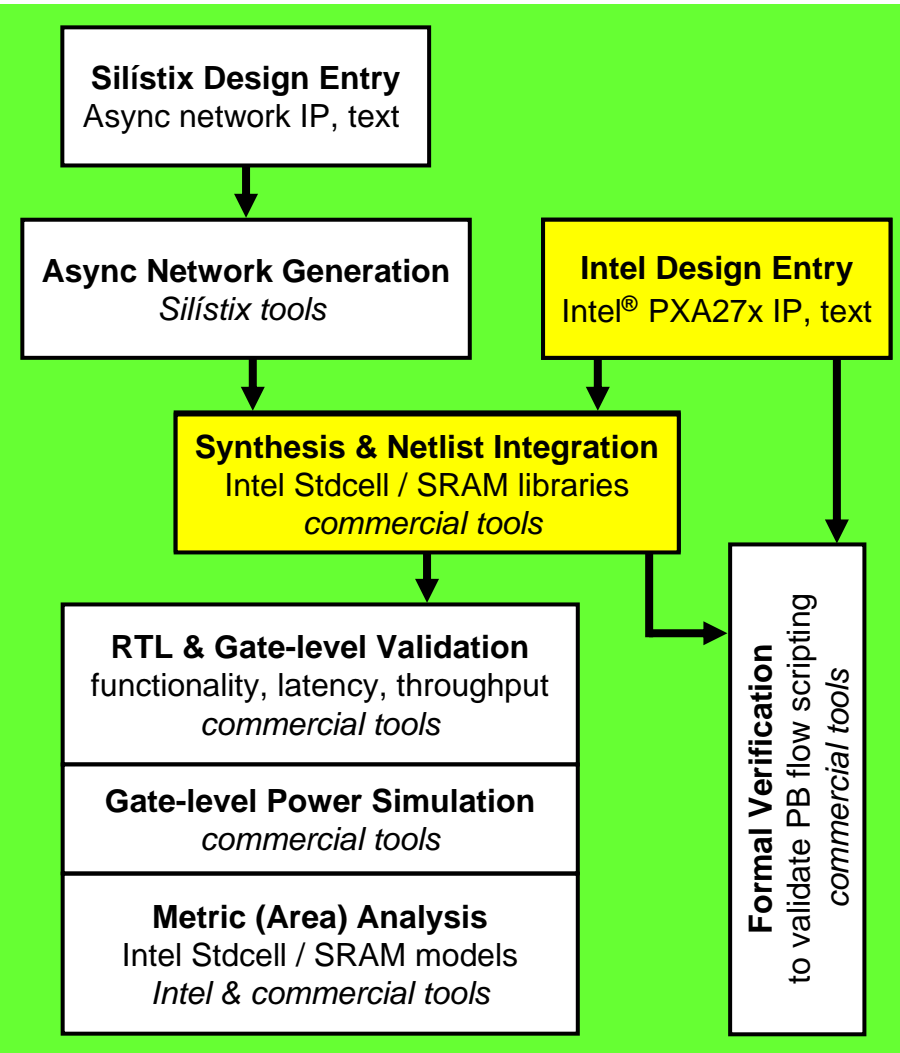
- *Enter* high-level description of self-timed NoC topology
- *Generate* hierarchical, structural Verilog netlists
- *Modify* UART PB-facing logic to attach directly to the asynchronous fabric

Intel Design Entry & Network Construction



- *Typical Low-Power SoC flow*
 - uses commercial EDA tools
 - used for wide product & process range (180, 130, 90nm etc.)

Intel Design Entry & Network Construction

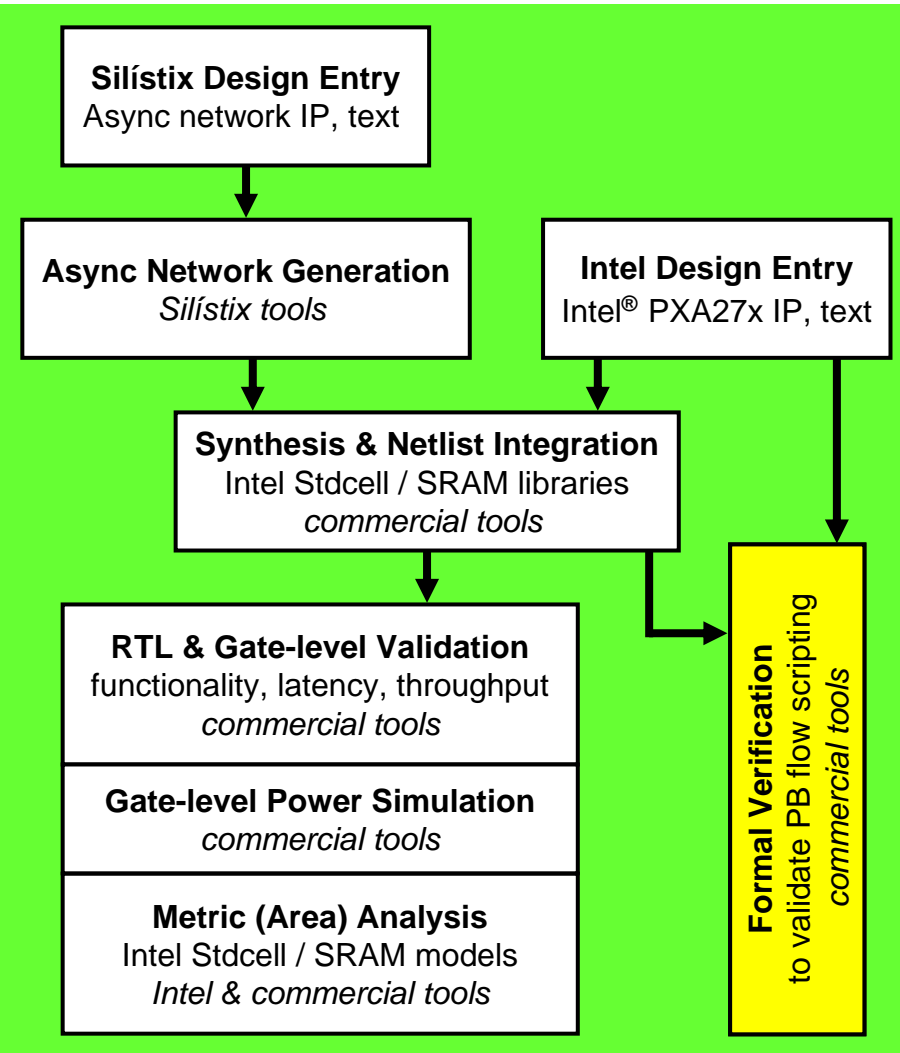


Our Usage Model:

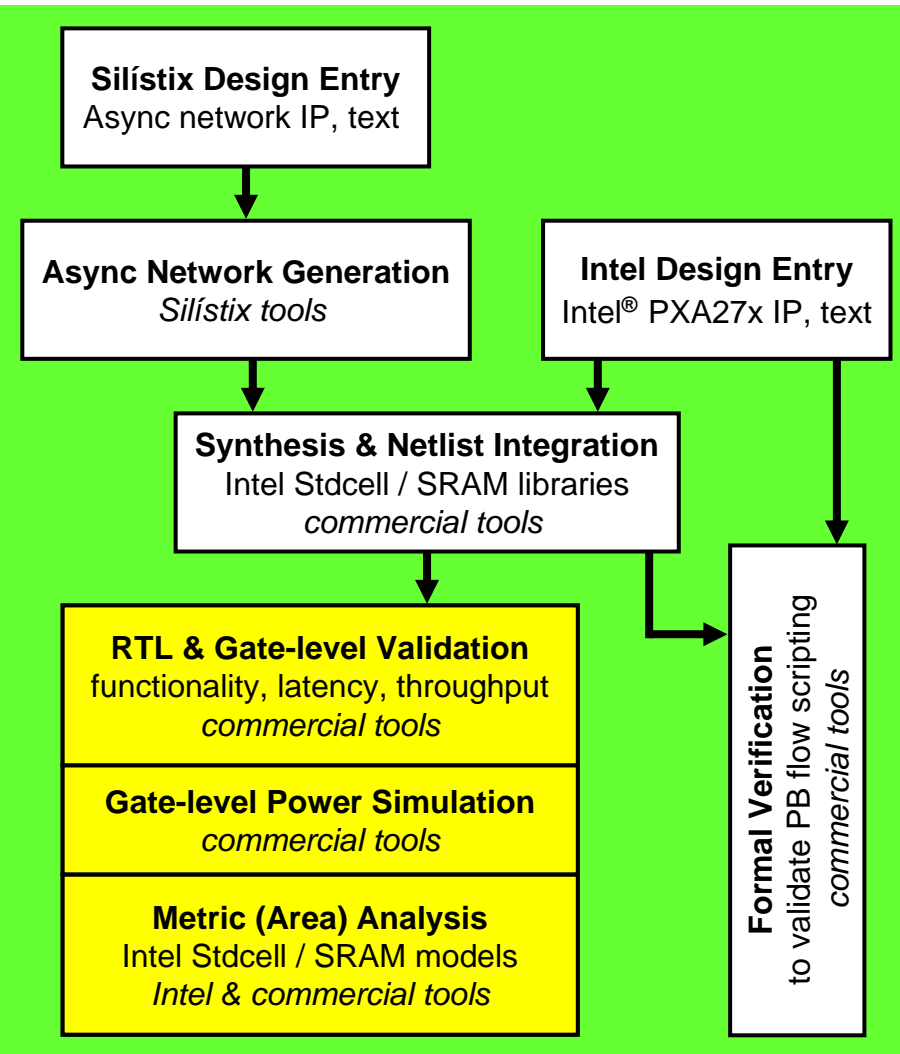
- *Synthesize synchronous blocks*
 - at 2 PVT corners
 - 1M-gate Wire-load model to match original 27-peripheral PB
 - No clock-gating, scan-insertion
- *Import Asynchronous blocks*
- *Stitch top-level networks*

Evaluation

- *Validate SoC flow usage*
 - Gate-to-gate FV
 - For key synchronous blocks



Evaluation



- *Dynamic Simulation*
 - Functionality, Timing & Power
 - Unit-delay models
 - Back-annotation, 2 PVT corners
- *Typical PB Traffic scenarios*
 - 0.5MB/s (PB idle)
 - 1MB/s (PB Normal)
 - 10MB/s (PB max)
- *Netlist-based Metric collection*

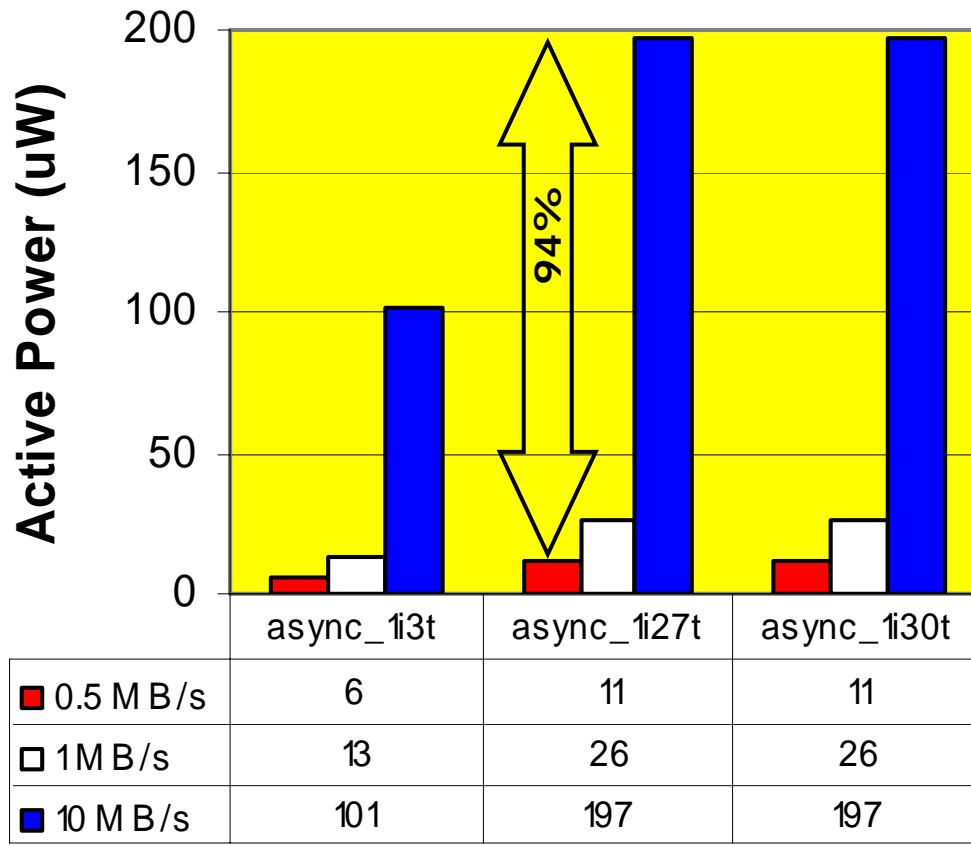
Top-Level Networks

- *Small test cases for debug: `sync_1i3t`, `async_1i3t`*
 - 1x (UART-sync, BB, SSP), 1x (UART-async, BB, SSP)
- *Primary test cases for Async-Sync comparisons:*
 - `async_127t`*
 - 9x (UART-async, BB, SSP)
 - UART-sync + Synchronizing Adapter substituted for Metrics
 - `sync_1i27t`*
 - 9x (UART-sync, BB, SSP) + up-scaled PB_MUX
- *Extra test case to check scaling properties: `async_1i30t`*
 - 10x (UART-async, BB, SSP)



Results

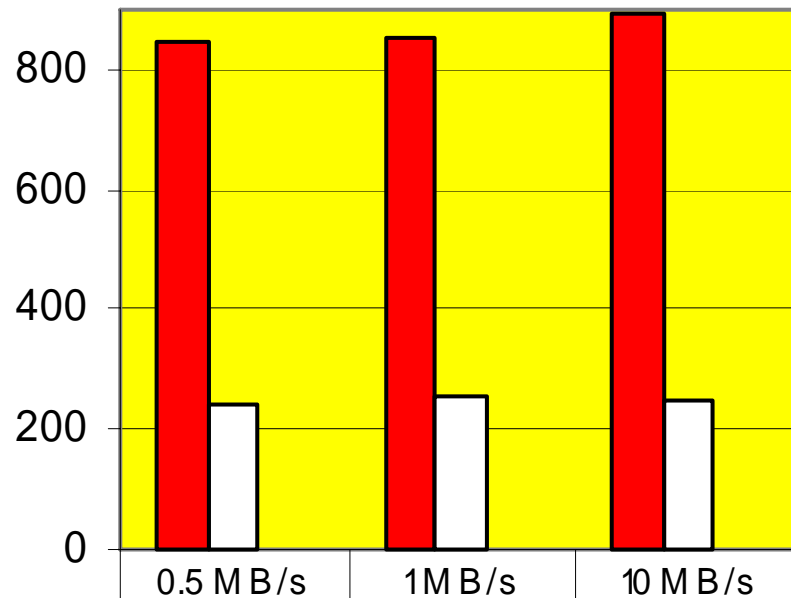
Active Power x Traffic: Async Fabric



- Async Fabric Power **SCALES** with traffic

Active Power x Traffic: **UART**

Active Power (uW)



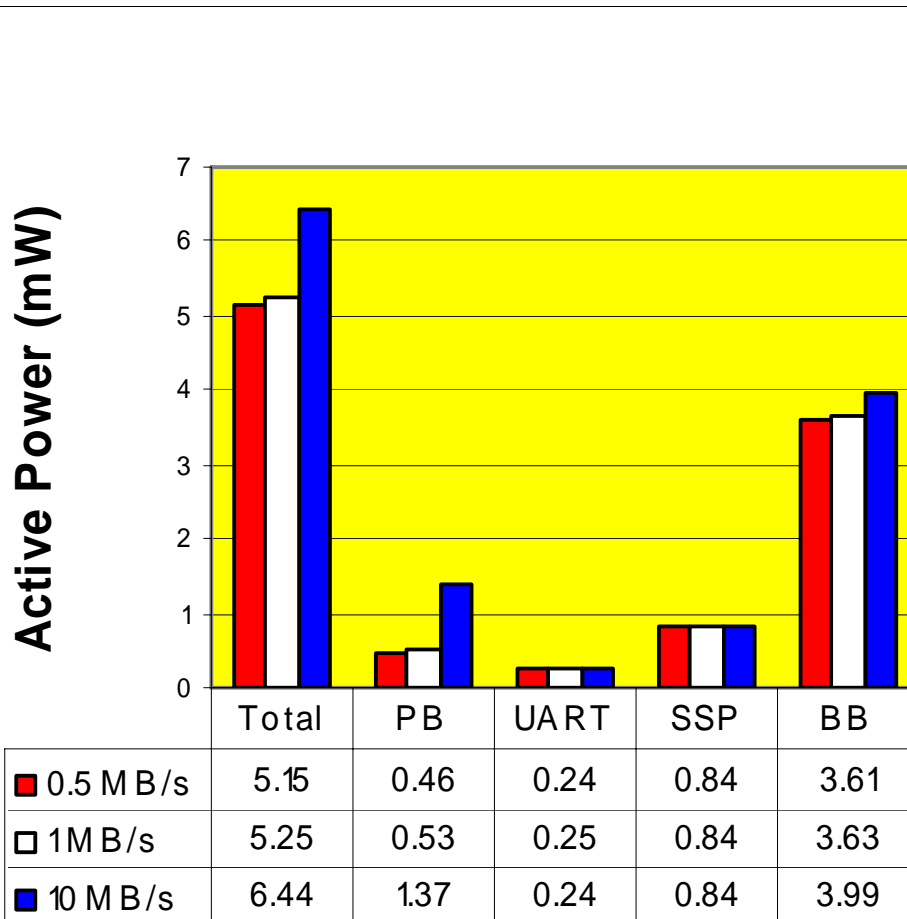
■ UART-sync	848	855	892
□ UART-async	242	253	248
Reduction	71%	70%	72%

- 70% Lower Power for Async Redesign

NOTE

- Async power scaling not visible for given TLT

Active Power x Data: **async_1i27t**



Synchronous Peripherals
dominate the Power spectrum

REASON:

- A small piece of Asynchronous
in a BIG synchronous World

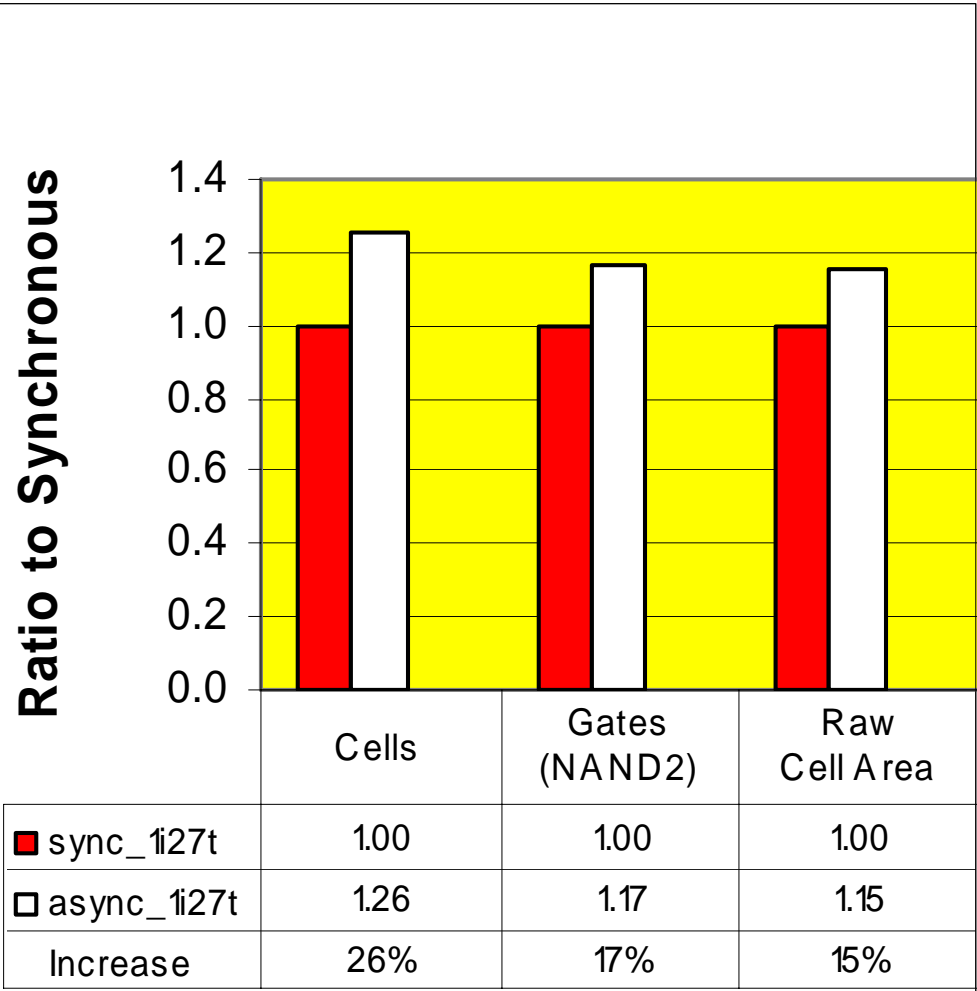
MEANS

- frequent interfacing

AND HENCE

- smaller up-scale of advantages

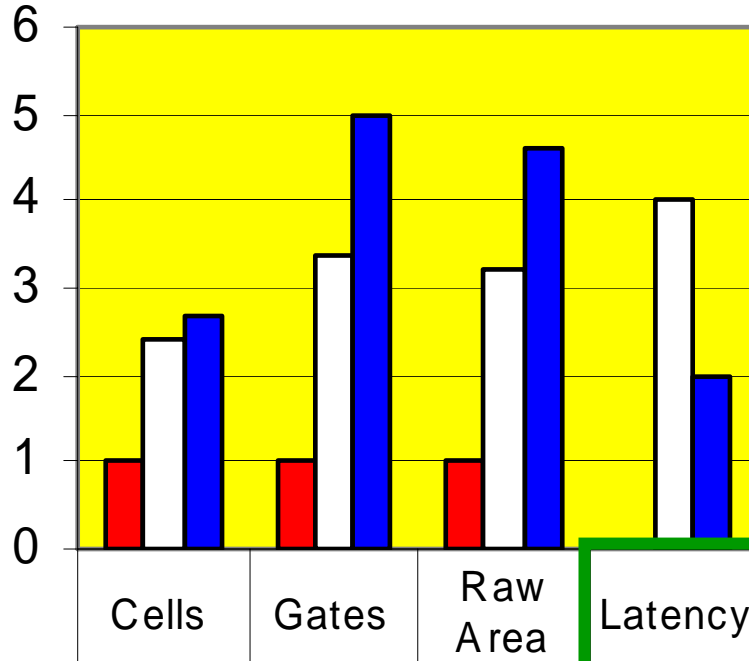
Metrics: Full Top-Level PB System



- Adapter overhead is small at the PB system level
- ~15% raw area
- add in WIRES:
 - 66% fewer wires
- which should result in:
 - better routing flexibility
 - better layout density

Interface Adaptation Metrics

Ratio to
Asynchronous Interface



- All 3 adaptation schemes worked!

■ Asynchronous	1.0	1.0	1.0	0
□ Synchronizing	2.4	3.4	3.2	4
■ Pausible Clock	2.7	5.0	4.6	2

**KEY learning
is HERE...**

Latency and bandwidth

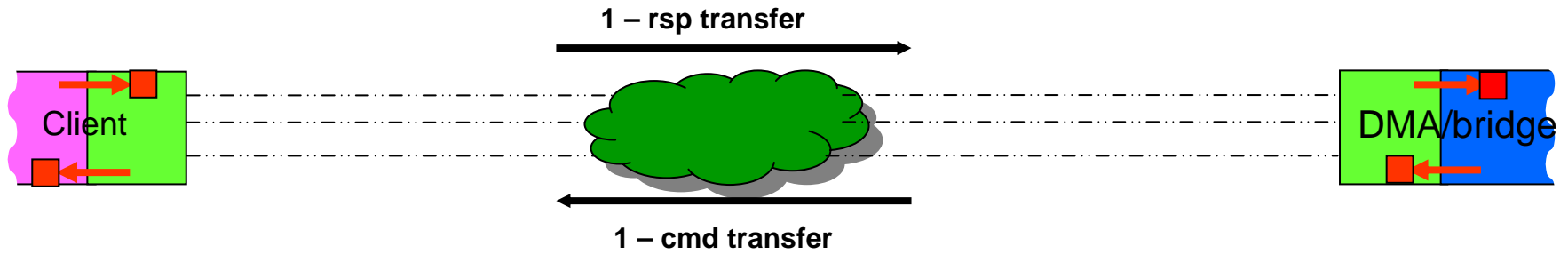
- PB had no latency requirement, but every transfer was 2 cycles, with no transfer overlapping or pipelining → all latency directly limits bandwidth.



- PB bus protocol requires 2 cycles per transfer

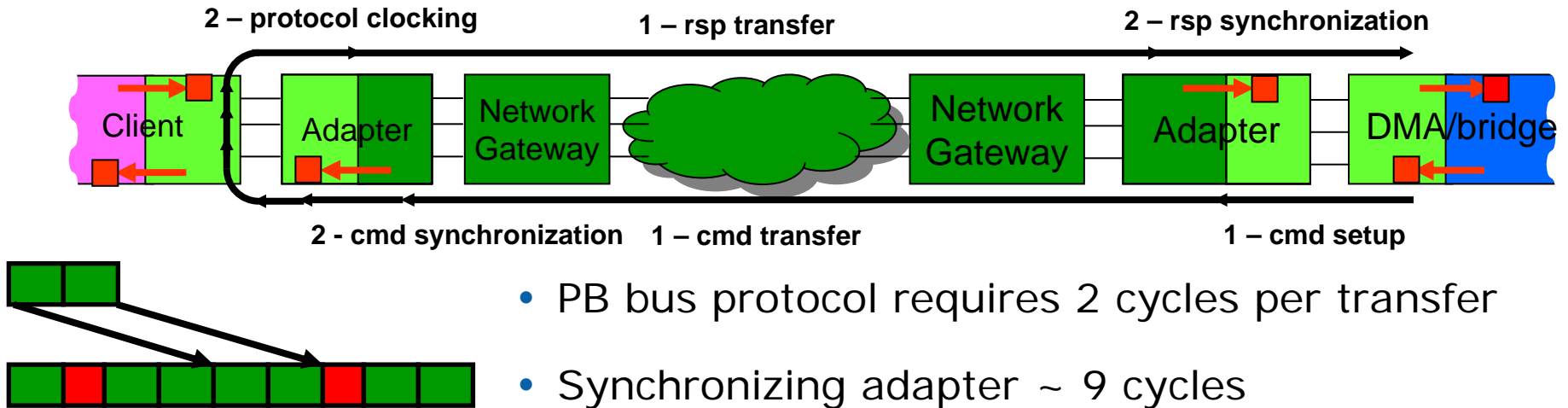


Latency and bandwidth

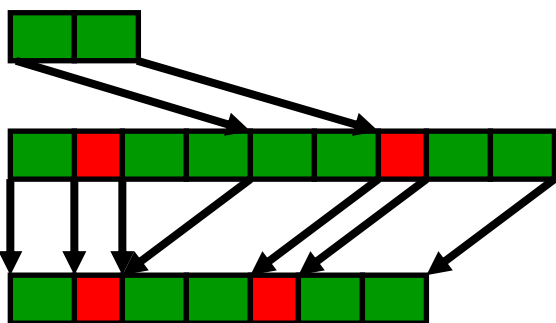
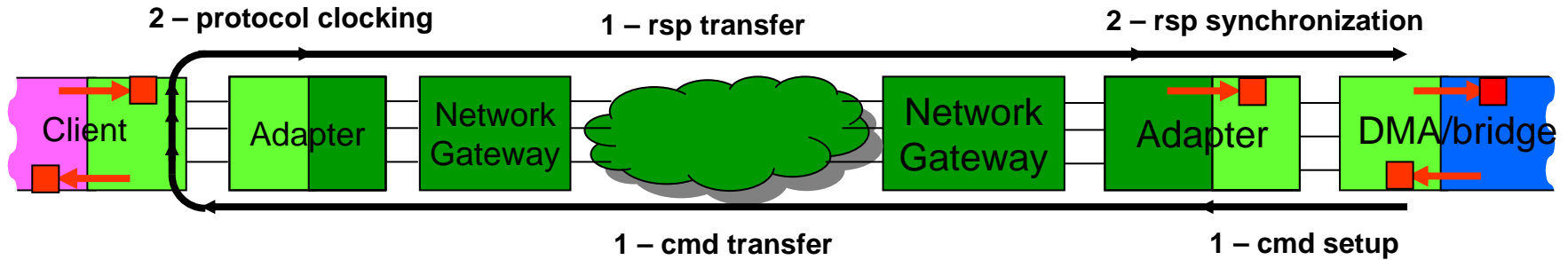


- PB bus protocol requires 2 cycles per transfer

Latency and bandwidth

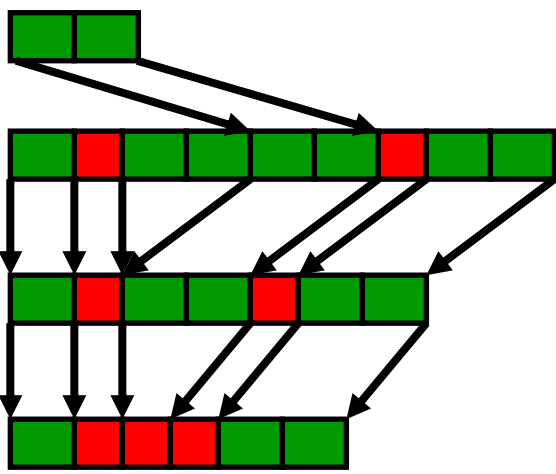
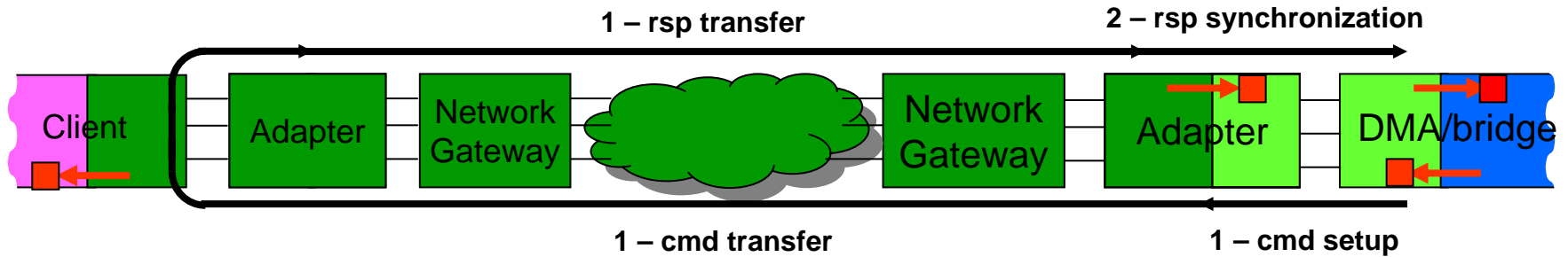


Latency and bandwidth



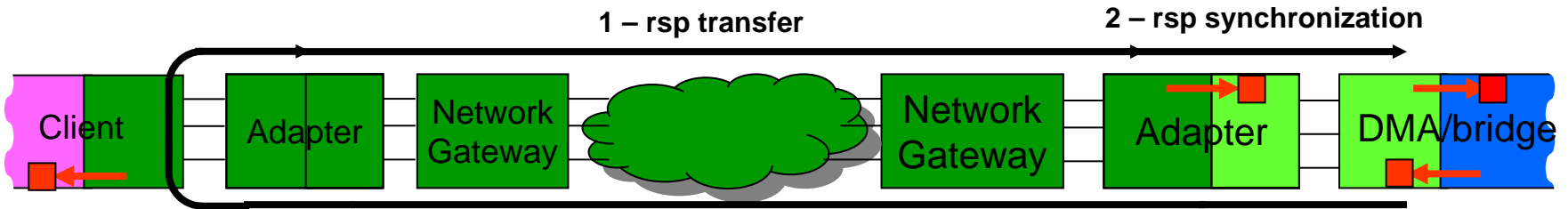
- PB bus protocol requires 2 cycles per transfer
- Synchronizing adapter ~ 9 cycles
 - Latency limits bandwidth - **only 90% of target**
- Pausable clock adapter ~ 7 cycles
 - Removes 2 cycles of command synchronization

Latency and bandwidth

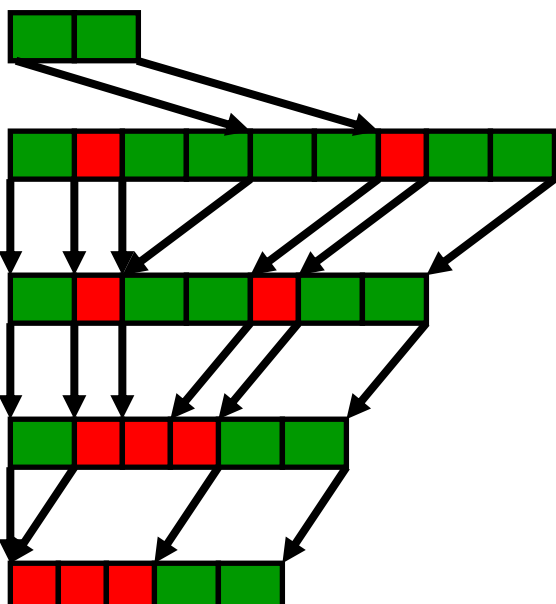


- PB bus protocol requires 2 cycles per transfer
- Synchronizing adapter ~ 9 cycles
 - Latency limits bandwidth - **only 90% of target**
- Pausable clock adapter ~ 7 cycles
 - Removes 2 cycles of command synchronization
- Async peripheral interface ~ 5 cycles
 - Removes ~2 cycles protocol clocking overhead

Latency and bandwidth

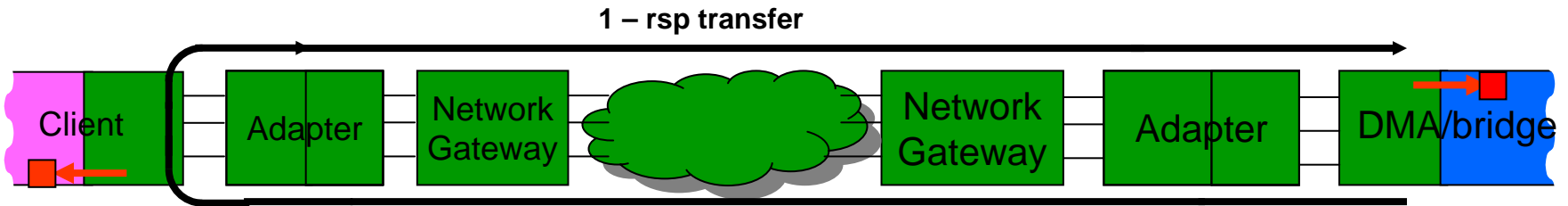


1 - cmd transfer



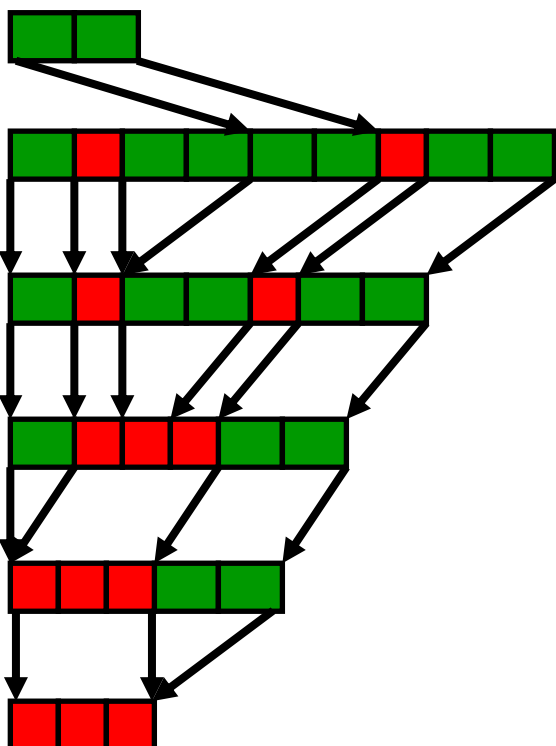
- PB bus protocol requires 2 cycles per transfer
- Synchronizing adapter ~ 9 cycles
 - Latency limits bandwidth - **only 90% of target**
- Pausable clock adapter ~ 7 cycles
 - Removes 2 cycles of command synchronization
- Async peripheral interface ~ 5 cycles
 - Removes ~2 cycles protocol clocking overhead
- Logic optimization ~ 4 cycles
 - Bus arbitration cycle – unnecessary for PB protocol

Latency and bandwidth



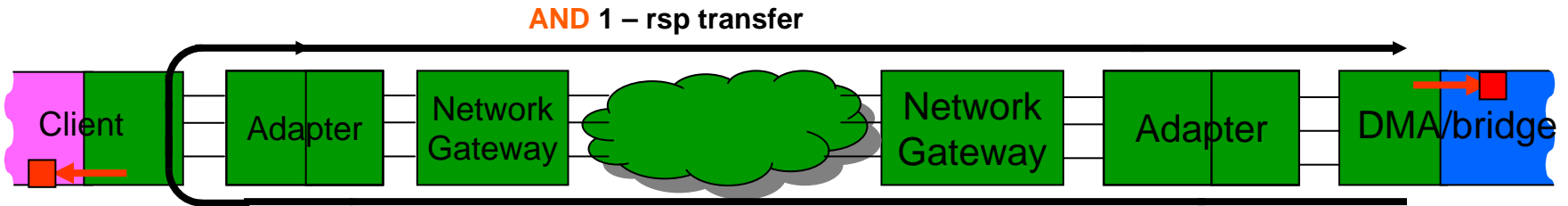
1 – rsp transfer

1 – cmd transfer

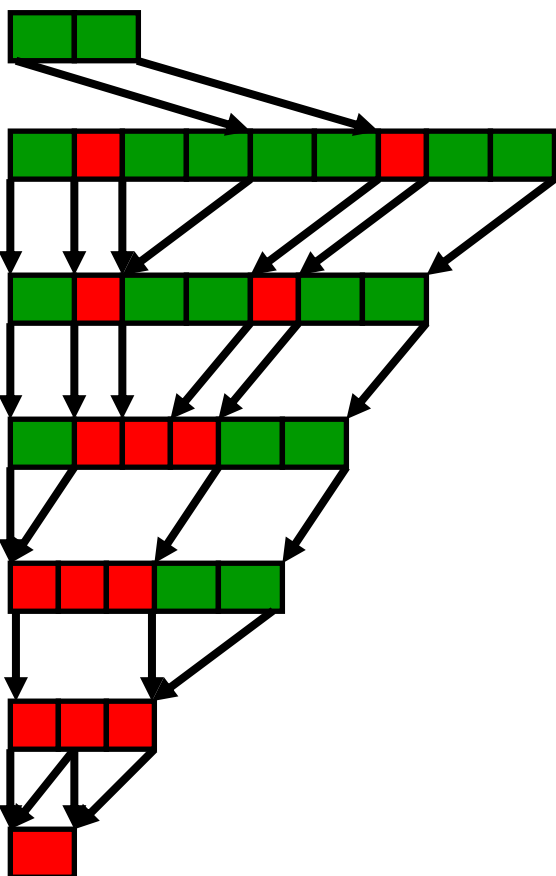


- PB bus protocol requires 2 cycles per transfer
- Synchronizing adapter ~ 9 cycles
 - Latency limits bandwidth - **only 90% of target**
- Pausable clock adapter ~ 7 cycles
 - Removes 2 cycles of command synchronization
- Async peripheral interface ~ 5 cycles
 - Removes ~2 cycles protocol clocking overhead
- Logic optimization ~ 4 cycles
 - Bus arbitration cycle – unnecessary for PB protocol
- Async bridge ~ 2 cycles
 - Removes 2 cycles of response synchronization

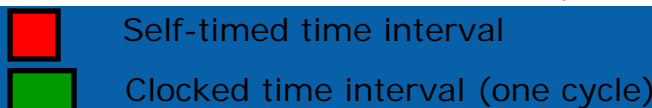
Latency and bandwidth



1 – cmd transfer



- PB bus protocol requires 2 cycles per transfer
- Synchronizing adapter ~ 9 cycles
 - Latency limits bandwidth - **only 90% of target**
- Pausable clock adapter ~ 7 cycles
 - Removes 2 cycles of command synchronization
- Async peripheral interface ~ 5 cycles
 - Removes ~2 cycles protocol clocking overhead
- Logic optimization ~ 4 cycles
 - Bus arbitration cycle – unnecessary for PB protocol
- Async bridge ~ 2 cycles
 - Removes 2 cycles of response synchronization
- Future: Concurrent command&response ~ 1 cycle
200% (2x improvement) of target



Key Learnings & Future Directions

- Partition with NoC in Mind
 - Minimize the number of timing domain crossings
 - Partition between NoC, Peripherals & Interface logic
 - Encapsulate asynchronous NoCs to simplify integration with mostly-synchronous tools
- Take advantage of NoC Strengths!
 - Exploit the layered communication approach
 - Concurrency can dramatically improve throughput & latency
 - Lower IP generation & validation costs
 - Self-timed NoC promotes faster timing closure and lower standby power
- Employ Transaction Level Test Suites
 - They were invaluable in testing, debugging, and benchmarking our NoCs
 - Enables portable, maintainable, flexible validation suites re-usable at multiple levels of abstraction
 - Real SoC traffic isn't homogeneous, and is much easier to model in a flexible, modular TLT

Key Learnings & Future Directions

- It's still a "mostly-synchronous" SoC world
 - New methods must seamlessly integrate with mostly-synchronous flows
 - Static Timing analysis flows & engines need to be enhanced to better handle complex multi-frequency and asynchronous design content (see SRC investigation by Beerel/Stevens)
- SoC Developers want flexibility in choosing Power, Latency, Bandwidth and Area
 - Our four-phase 1-hot QDI style was very robust, but a limiting factor in power reduction and achievable bandwidth
 - We see potential benefits in two-phase, single-rail and alternate QDI encodings
 - We expect that additional asynchronous cells and single-rail FIFOs will enable further improvements

Summary

- We built an asynchronous NoC in a synchronous SoC flow, today
- We demonstrated asynchronous NoC advantages
- We explored a number of tradeoffs
- We learned lessons & identified areas for further development

Asynchronous NoC in SoC.

Do it.