

# ESP: The Open-Source SoC Platform

**Luca Carloni**

**Department of Computer Science  
Columbia University in the City of New York**



**International Conference on VLSI Design & International Conference on Embedded Design  
Bangalore, India, January 5<sup>th</sup>, 2020**



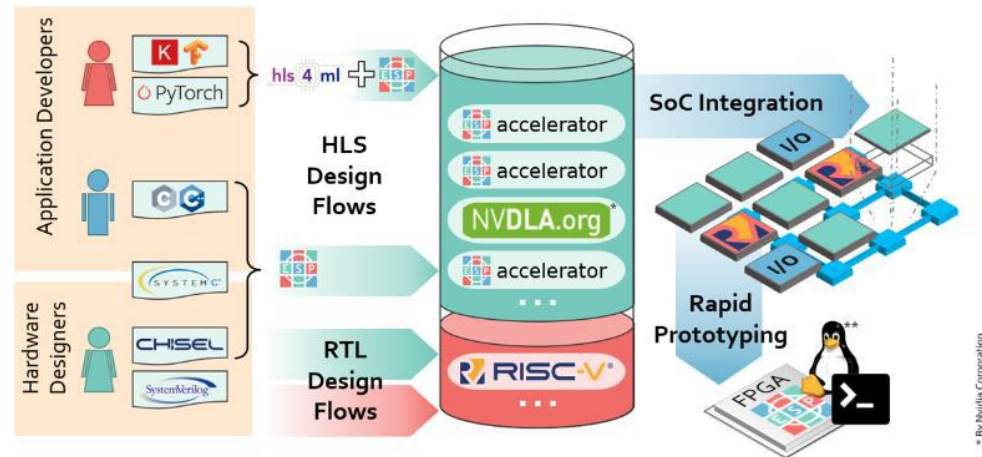
# Open Source Release of ESP

<https://www.esp.cs.columbia.edu>



## The ESP Vision

ESP is an open-source research platform for heterogeneous system-on-chip design that combines a flexible tile-based architecture and a modular system-level design methodology.



ESP provides three accelerator flows: RTL, high-level synthesis (HLS), machine learning frameworks. All three design flows converge to the ESP automated SoC integration flow that generates the necessary hardware and software interfaces to rapidly enable full-system prototyping on FPGA.

## Latest Posts



### Upcoming talk at VLSID & ES 2020 in Bangalore

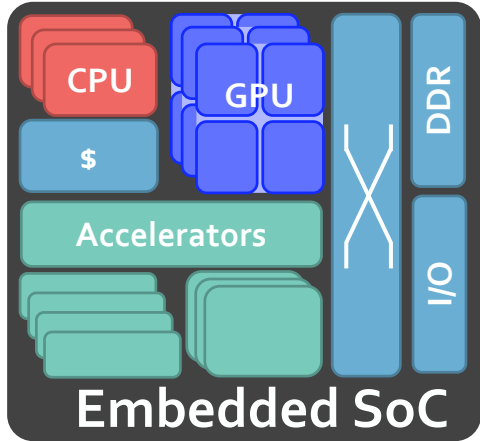
We will give a talk about ESP in Bangalore (India) on January 5th at the International Conference on VLSI Design and International Conference on Embedded Design (VLSID & ES).

[Read more](#)

Published: Jan 2, 2020

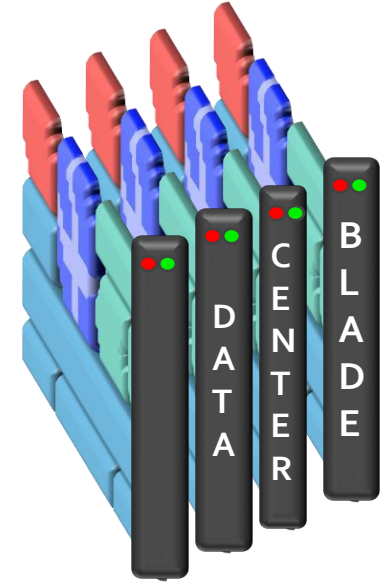


# Why ESP?



**Heterogeneous systems** are pervasive  
Integrating **accelerators** into a SoC is hard  
Doing so in a **scalable** way is very hard

Keeping the system **simple to program** while doing so is even harder



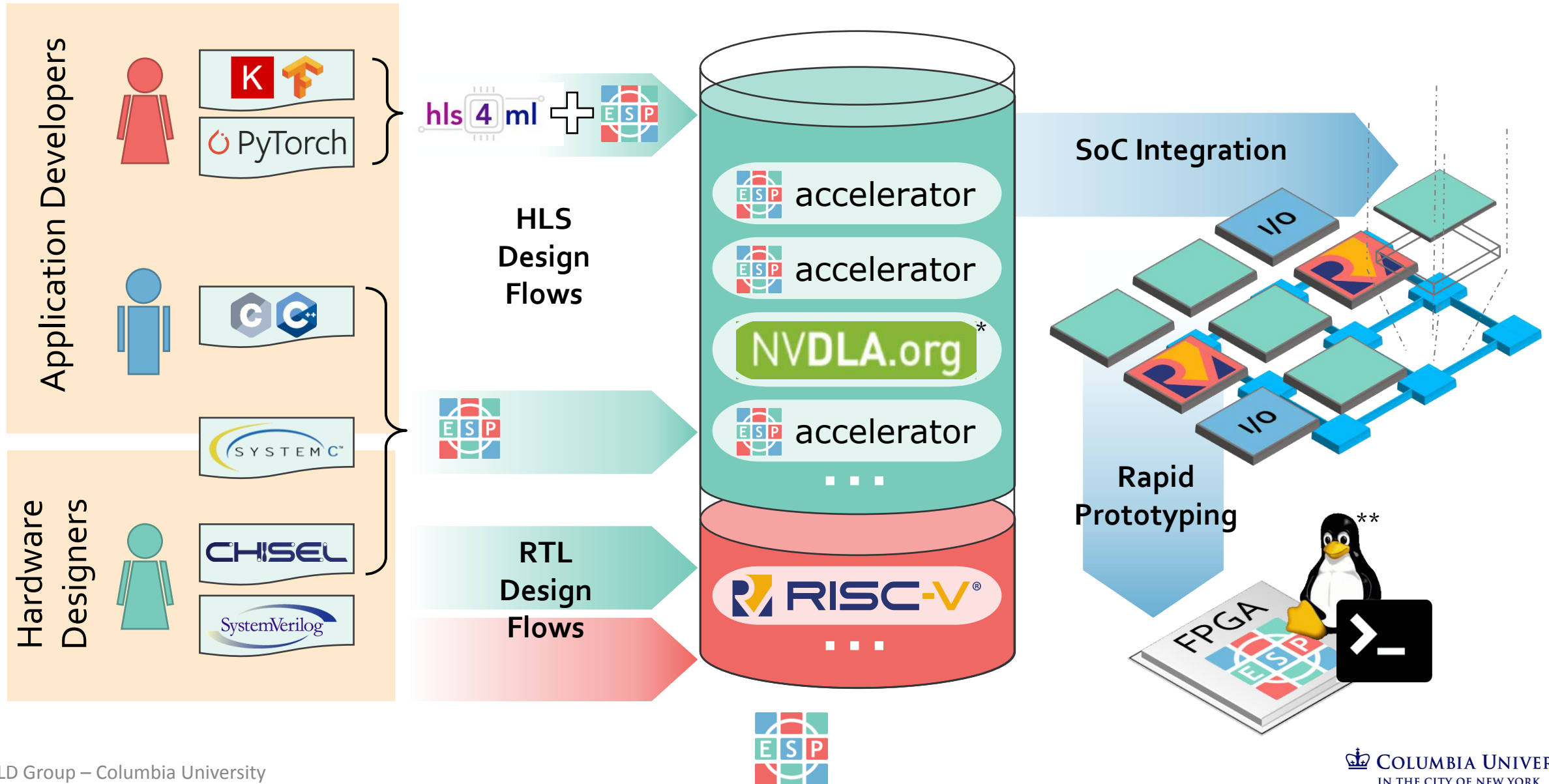
**ESP** makes it **easy**

ESP combines a **scalable architecture** with a **flexible methodology**

ESP enables **several accelerator design flows**  
and takes care of the hardware and software integration



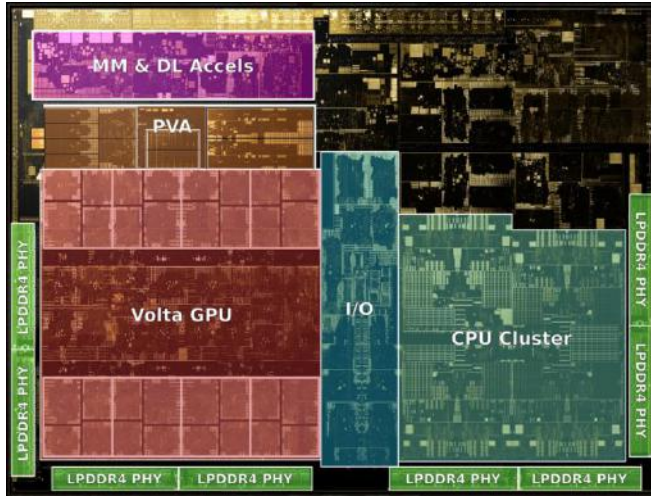
# ESP Vision: Domain Experts Can Design SoCs



\* By Nvidia Corporation  
\*\* By lewing@isc.tamu.edu Larry Ewing and The GIMP



# Outline

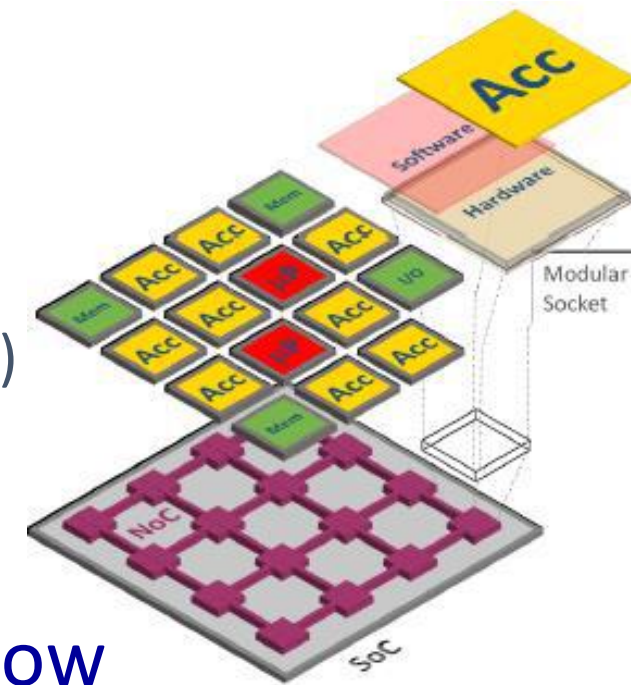


## 1. Motivation

- The Rise of Heterogeneous Computing

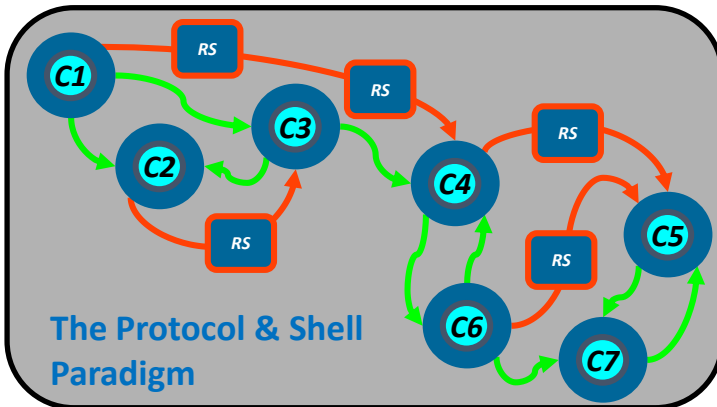
## 2. Proposed Architecture

- Embedded Scalable Platforms (ESP)

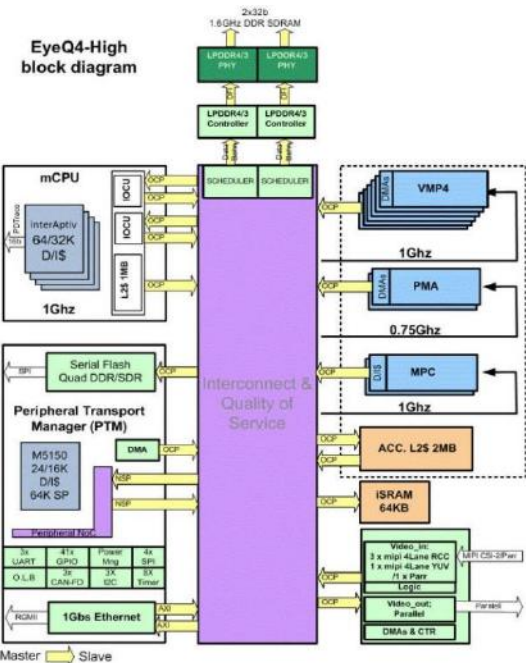


## 3. Methodology and Design Flow

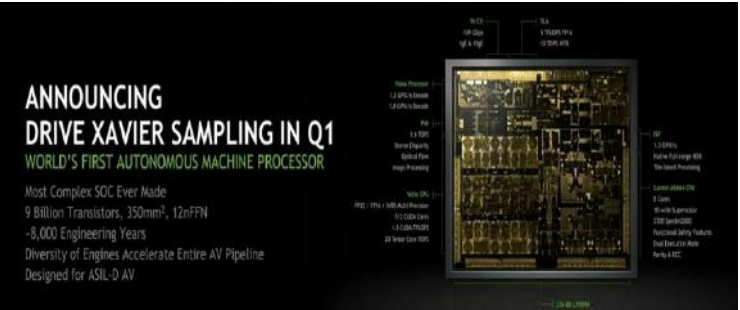
- with a Retrospective on Latency-Insensitive Design



# Heterogeneous Architectures Are Emerging Everywhere

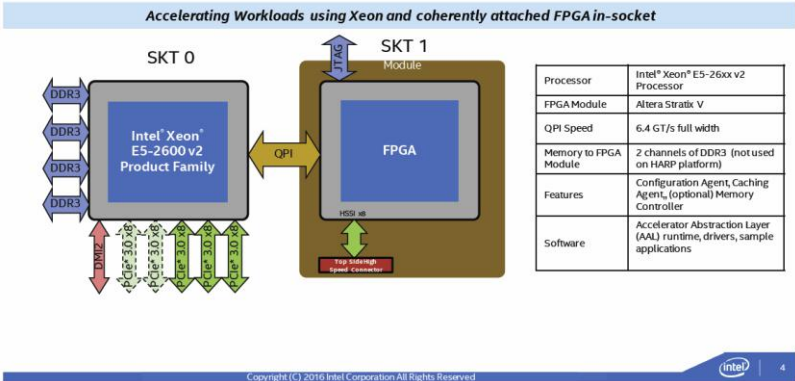


[ Source: [www.mobileye.com/](http://www.mobileye.com/) ]

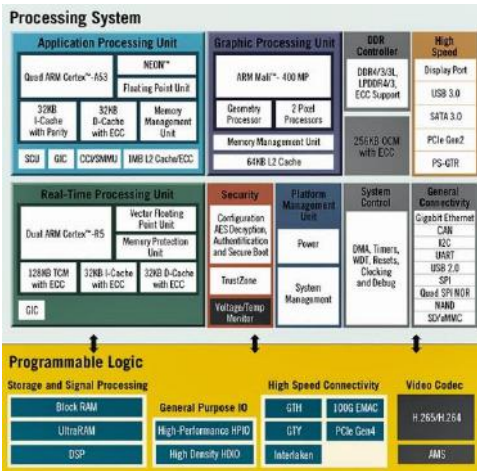


[ Source: <https://blogs.nvidia.com/> ]

## IvyTown Xeon + Stratix V FPGA



[ Source: "Xeon+FPGA Tutorial @ ISCA'16" ]

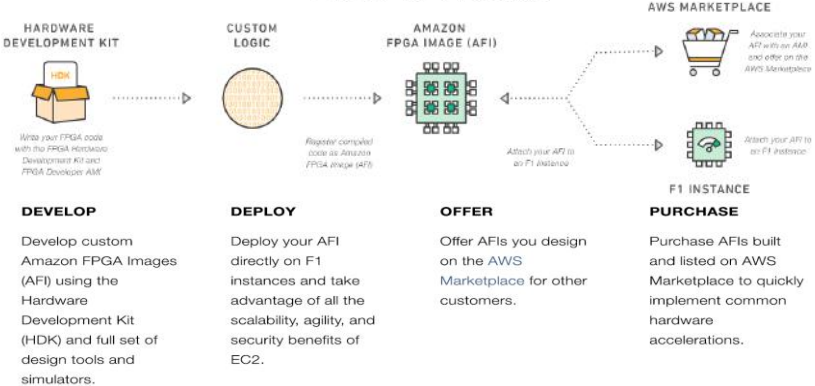


[ Source: [www.xilinx.com/](http://www.xilinx.com/) ]

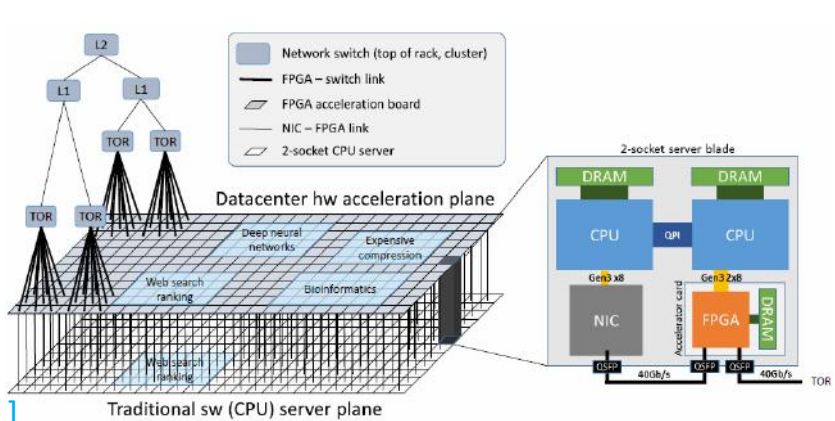


[ Source: <https://cloudplatform.googleblog.com/> ]

## How it Works



[ Source: <https://aws.amazon.com/ec2/instance-types/f1/> ]



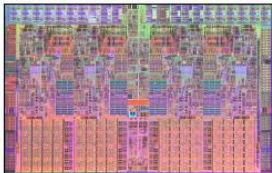
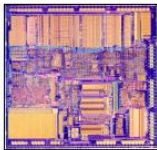
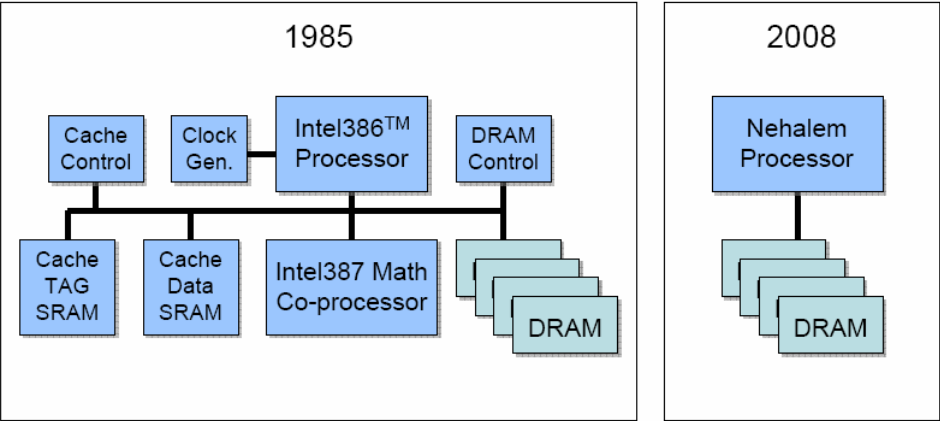
[ Source: [www.microsoft.com/](http://www.microsoft.com/) ]



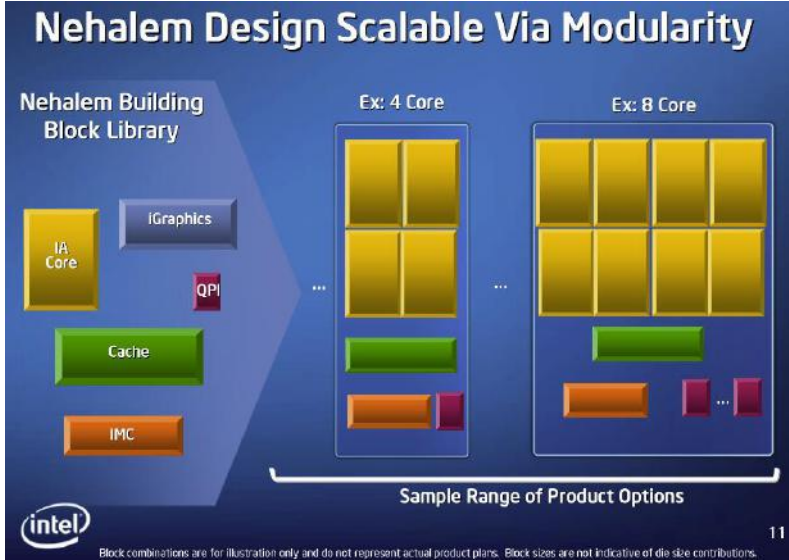


# From Microprocessors to Systems-on-Chip (SoC)

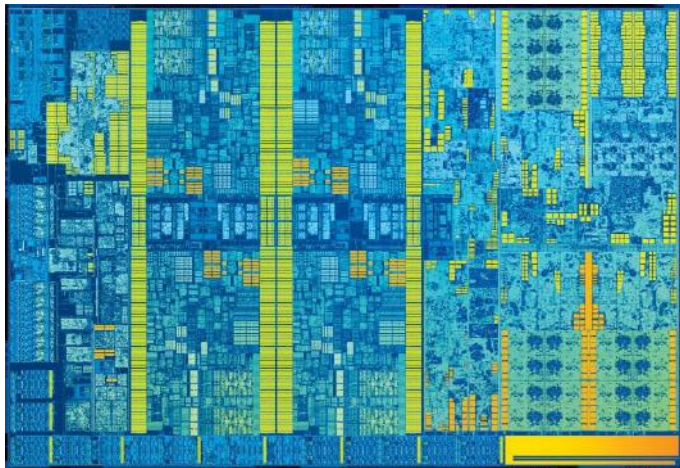
[ Source: M. Bohr 2009 ]



	Intel386™	Nehalem
Transistor Count:	280 thousand	731 million
Frequency:	16 MHz	>3.6 GHz
# Cores:	1	4
Cache Size:	None	8 MB
I/O Peak Bandwidth:	64 MB/sec	50 GB/sec
Adaptive Circuits:	None	Sleep Mode Turbo Mode Power Gating Adaptive Frequency Clocking



Source Intel.com [P. Gelsinger Press Briefing, Mar'08]



## 4.1 14nm 6th-Generation Core Processor SoC with Low Power Consumption and Improved Performance

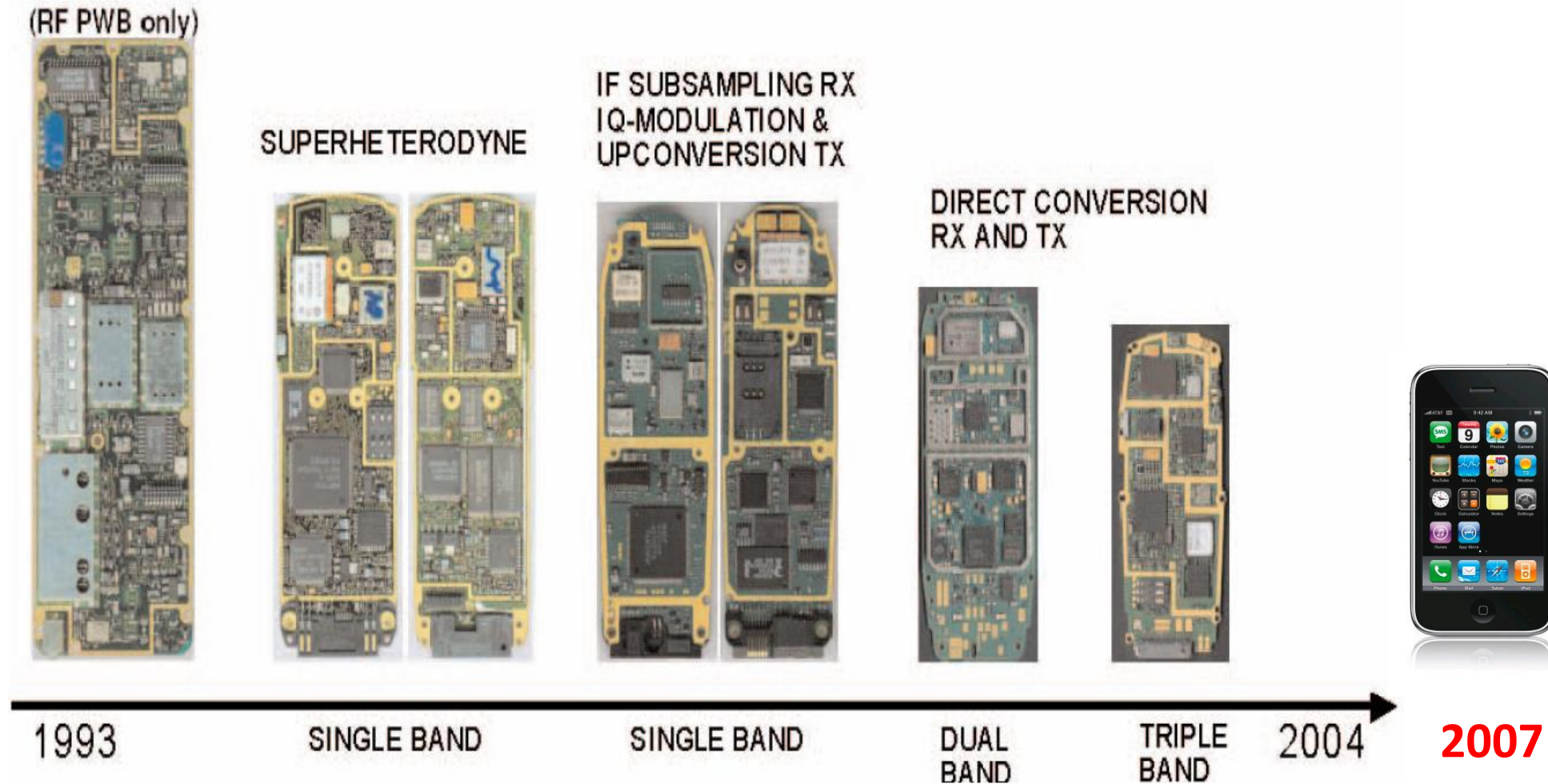
Eyal Fayneh, Marcelo Yuffe, Ernest Knoll, Michael Zelikson, Muhammad Abozaed, Yair Talker, Ziv Shmueli, Saher Abu Rahme

Intel, Haifa, Israel

Intel's 6<sup>th</sup> generation Core processor (code named "Skylake" or SKL) was designed



# The System Migrates into The Chip: Evolution of Mobile Phones

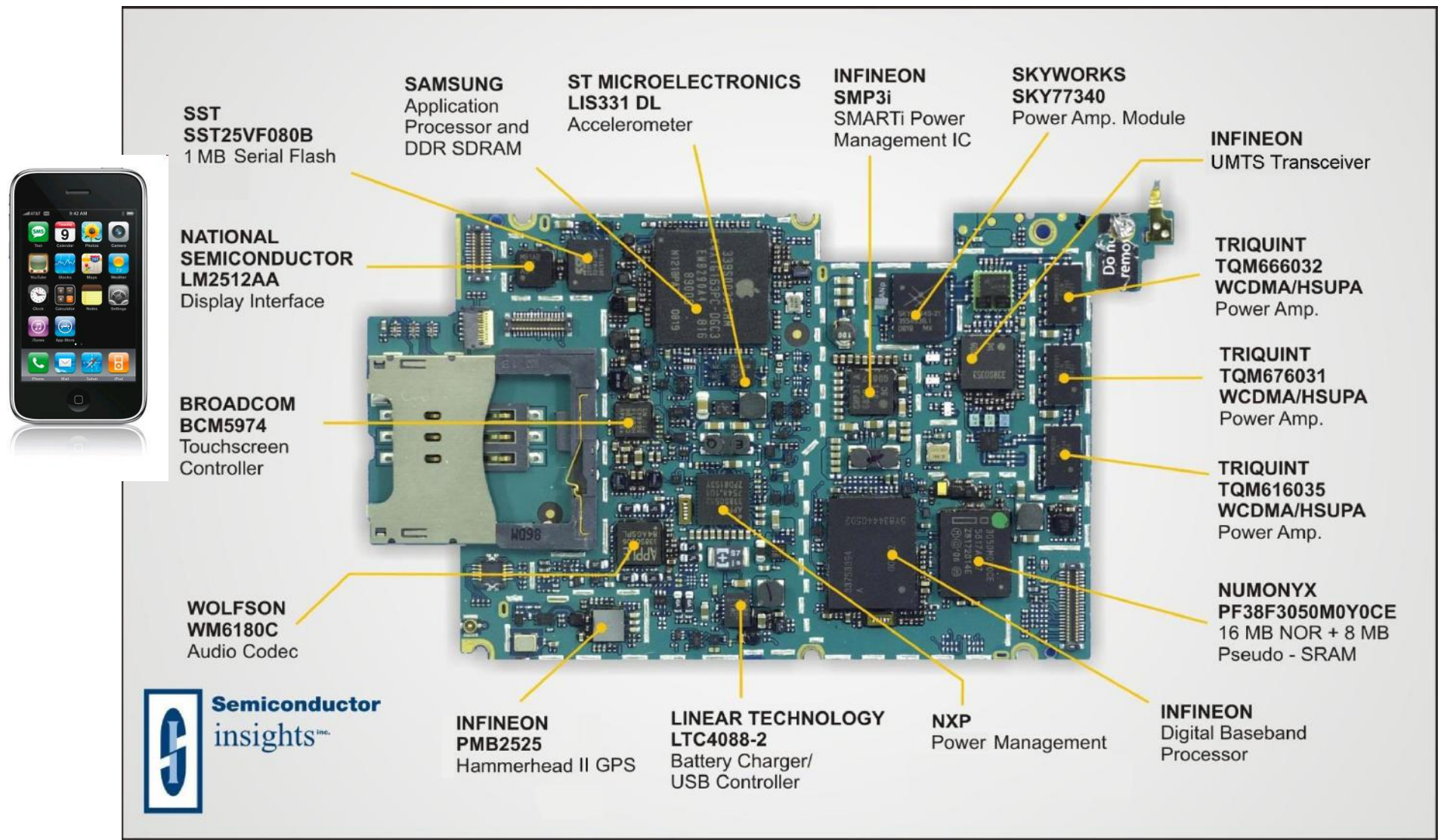


[ Source: Y. Neuvo, "Cellular phones as Embedded Systems", ISSCC 2004 ]





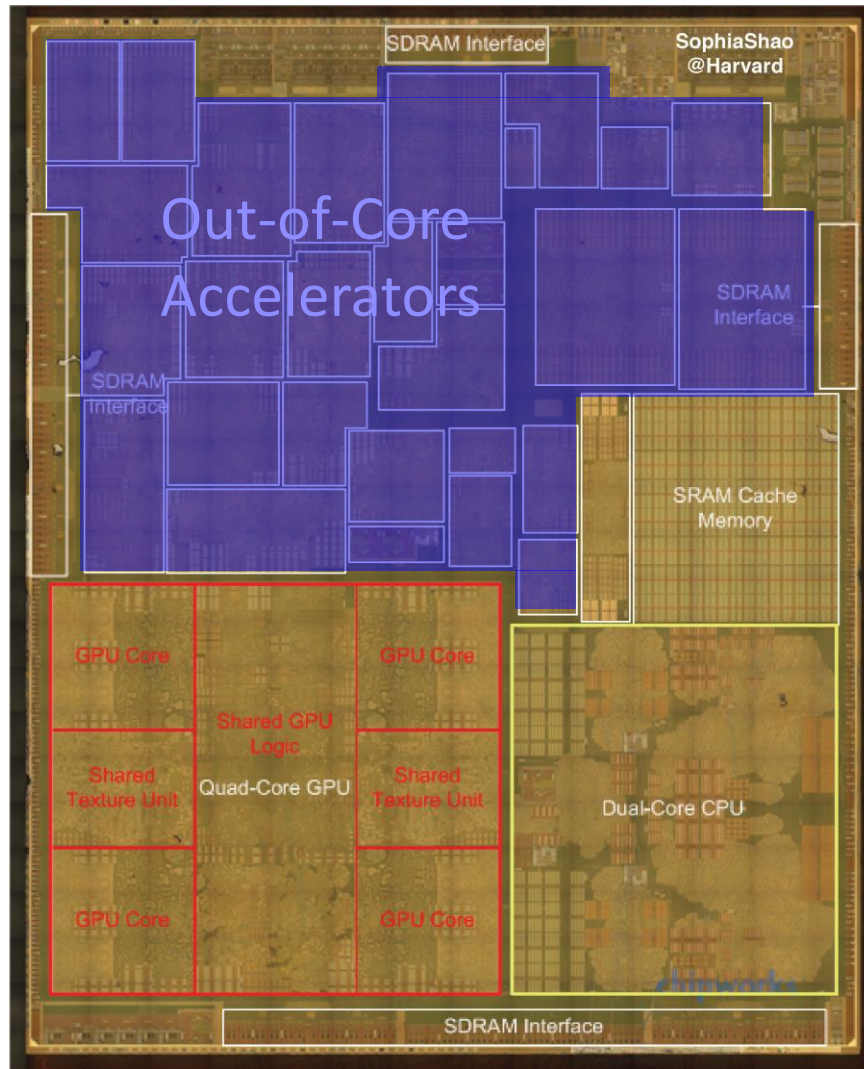
# Inside the SmartPhone Revolution: the Apple I-Phone 3G



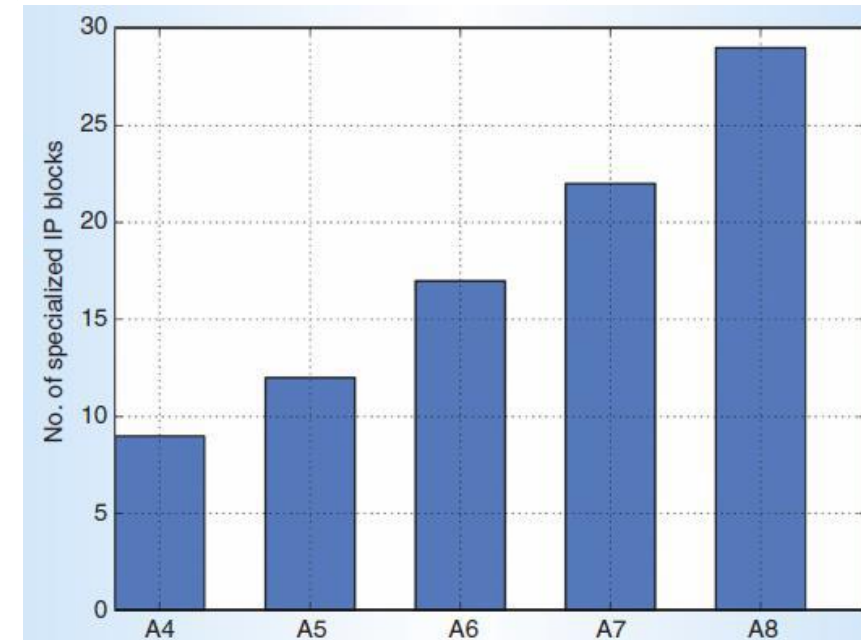
[ Source: Semiconductor Insights ]



# The Growth of Specialized IP Blocks: The Apple A8 SoC



Number of specialized IP blocks across five generations of Apple SoCs



[ Source: Shao et al. 2015]

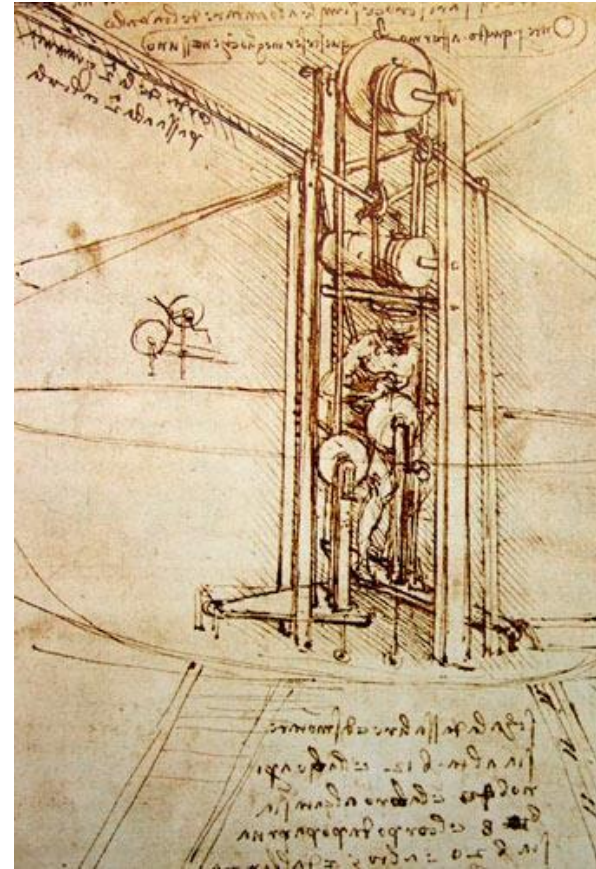
- The analysis of die photos from Apple's A6, A7, and A8 SoCs shows that more than half of the die area is dedicated to blocks that are neither CPUs nor GPUs, but rather specialized Intellectual Property (IP) blocks
- Many IP blocks are *accelerators*, i.e. specialized hardware components that execute an important computation more efficiently than software





# A (Perhaps Easy?) Prediction: No Single Architecture Will Emerge as the Sole Winner

- **The migration from homogeneous multi-core architectures to heterogeneous System-on-Chip architectures will accelerate, across almost all computing domains**
  - from IoT devices, embedded systems and mobile devices to data centers and supercomputers
- **A heterogeneous SoC will combine an increasingly diverse set of components**
  - different CPUs, GPUs, hardware accelerators, memory hierarchies, I/O peripherals, sensors, reconfigurable engines, analog blocks...
- **The set of heterogeneous SoCs in production in any given year will be itself heterogeneous!**
  - no single SoC architecture will dominate all the markets

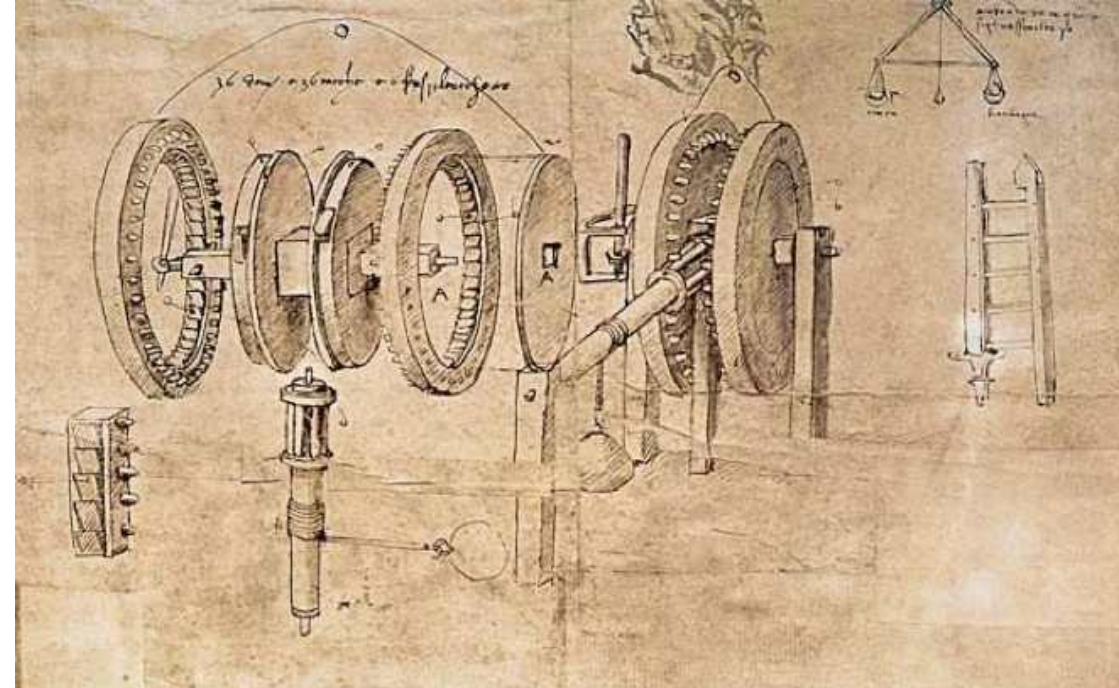




# Where the Key Challenges in SoC Design Are...

- The biggest challenges are (and will increasingly be) found in the **complexity of system integration**

- How to design, program and validate scalable systems that combine a very large number of heterogeneous components to provide a solution that is specialized for a target class of applications?



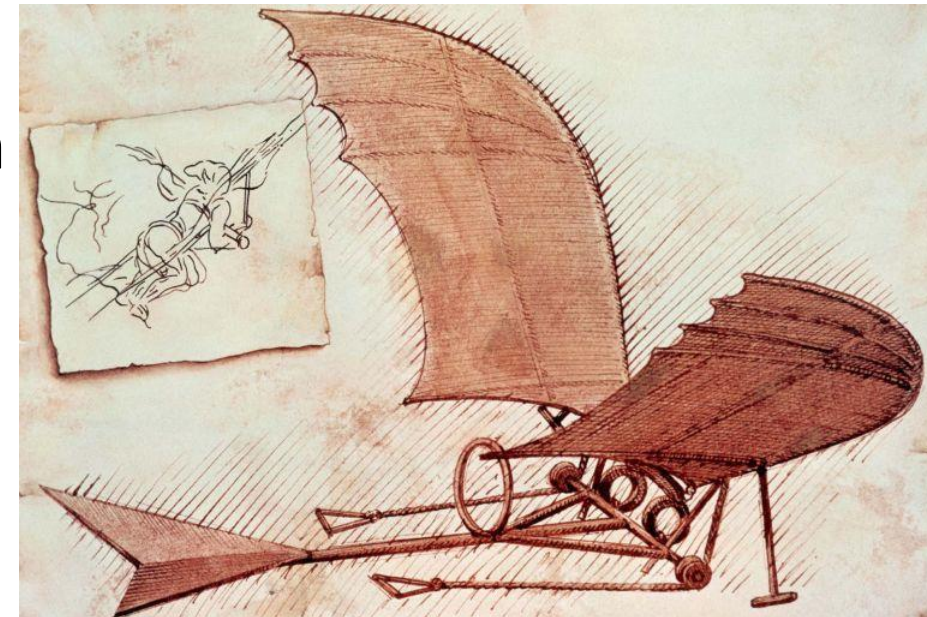
- How to handle this complexity?

- raise the level of abstraction to **System-Level Design**
- adopt compositional design methods with the **Protocol & Shell Paradigm**
- promote **Design Reuse**



# What is Needed? To Think at the System Level.

- **Move from a processor-centric to an SoC-centric perspective**
  - The processor core is just one component among many others
- **Develop platforms, not just architectures**
  - A platform combines an architecture and a companion design methodology
- **Raise the level of abstraction**
  - Move from RTL Design to System-Level Design
- **Promote Open-Source Hardware**
  - Build libraries of reusable components



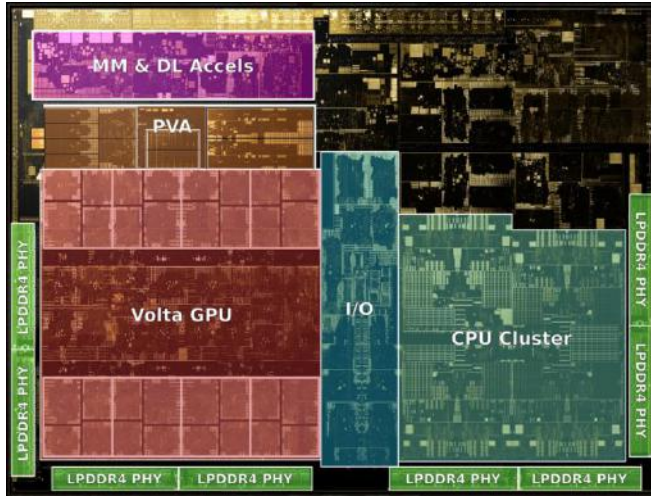
# Our System-Level Design Approach to Heterogeneous Computing: Key Ingredients

- **Develop Platforms, not just Architectures**
  - A **platform** combines an **architecture** and a companion **design methodology**
- **Raise the level of abstraction**
  - Move from RTL Design to **System-Level Design**
  - Move from Verilog/VHDL to high-level programming languages like **SystemC**
  - Move from ISA and RTL simulators to **Virtual Platforms**
  - Move from Logic Synthesis to **High-Level Synthesis** (both commercial and in-house tools), which is the key to enabling rich design-space exploration
- **Adopt compositional design methods**
  - Rely on **customizable libraries of HW/SW interfaces** to simplify the integration of heterogeneous components
- **Use formal metrics for design reuse**
  - Synthesize **Pareto frontiers** of optimal implementations from high-level specs
- **Build real prototypes (both chips and FPGA-based full-system designs)**
  - Prototypes **drive research** in systems, architectures, software and CAD tools





# Outline

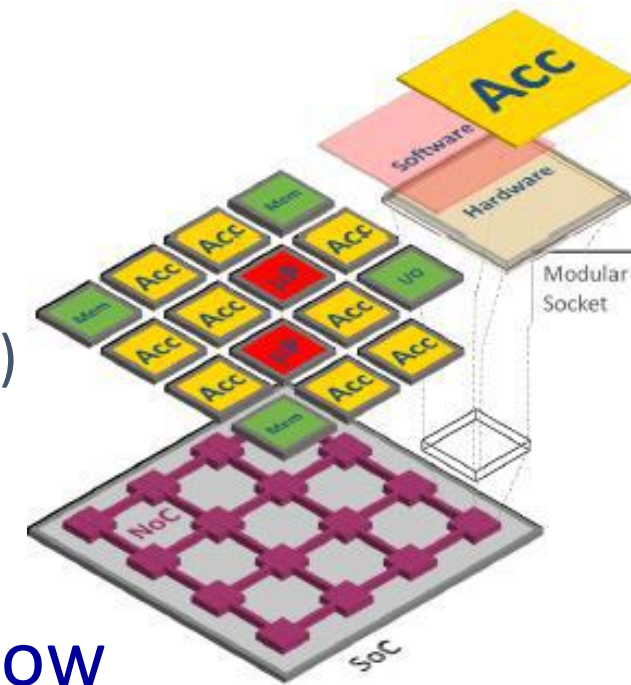


## 1. Motivation

- The Rise of Heterogeneous Computing

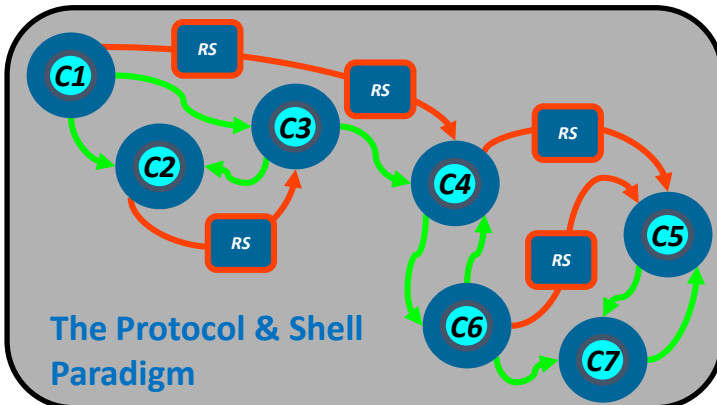
## 2. Proposed Architecture

- Embedded Scalable Platforms (ESP)

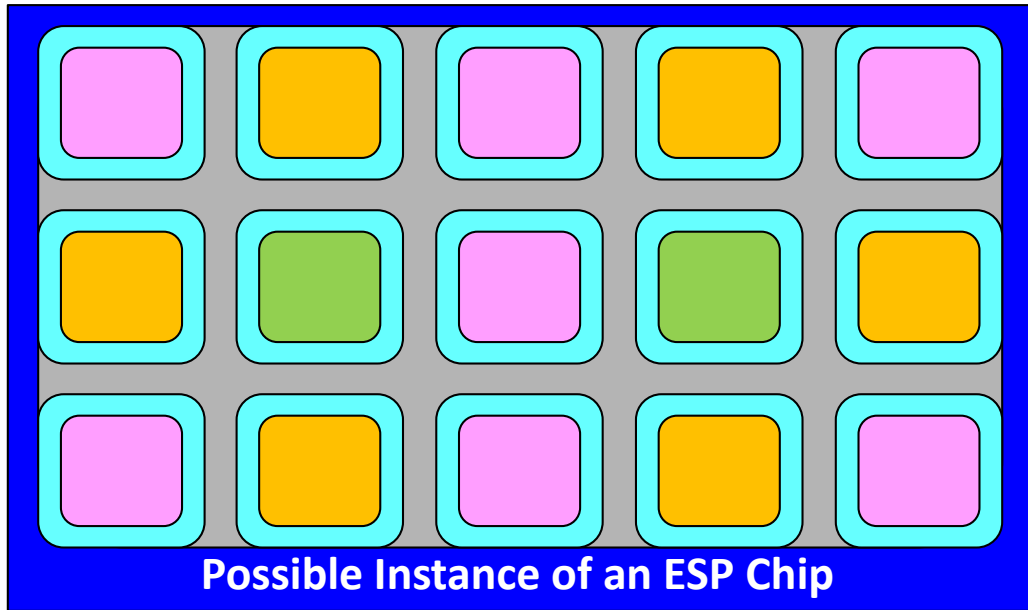


## 3. Methodology and Design Flow

- with a Retrospective on Latency-Insensitive Design



# The ESP Scalable Architecture Template



- **Processor Tiles**
  - each hosting at least one configurable processor core capable of running an OS
- **Accelerator Tiles**
  - synthesized from high-level specs
- **Other Tiles**
  - memory interfaces, I/O, etc.
- **Network-on-Chip (NoC)**
  - playing key roles at both design and run time

## Template Properties

- **Regularity**
  - tile-based design
  - pre-designed on-chip infrastructure for communication and resource management
- **Flexibility**
  - each ESP design is the result of a configurable mix of programmable tiles and accelerator tiles
- **Specialization**
  - with automatic high-level synthesis of accelerators for key computational kernels



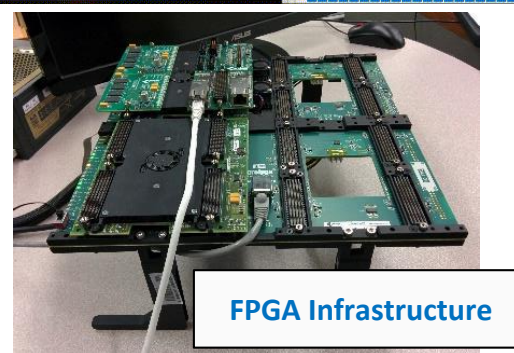
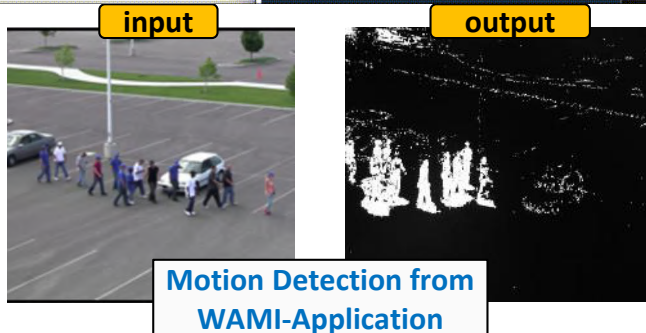


# Example of a System We Built: FPGA Prototype to Accelerate Wide-Area Motion Imagery



- **Design:** Complete design of WAMI-App running on an FPGA implementation of an ESP architecture

- featuring 1 embedded processor, 12 accelerators, 1 five-plane NoC, and 2 DRAM controllers
- SW application running on top of Linux while leveraging multi-threading library to program the accelerators and control their concurrent, pipelined execution
- **Five-plane, 2D-mesh NoC** efficiently supports multiple independent frequency domains and a variety of platform services



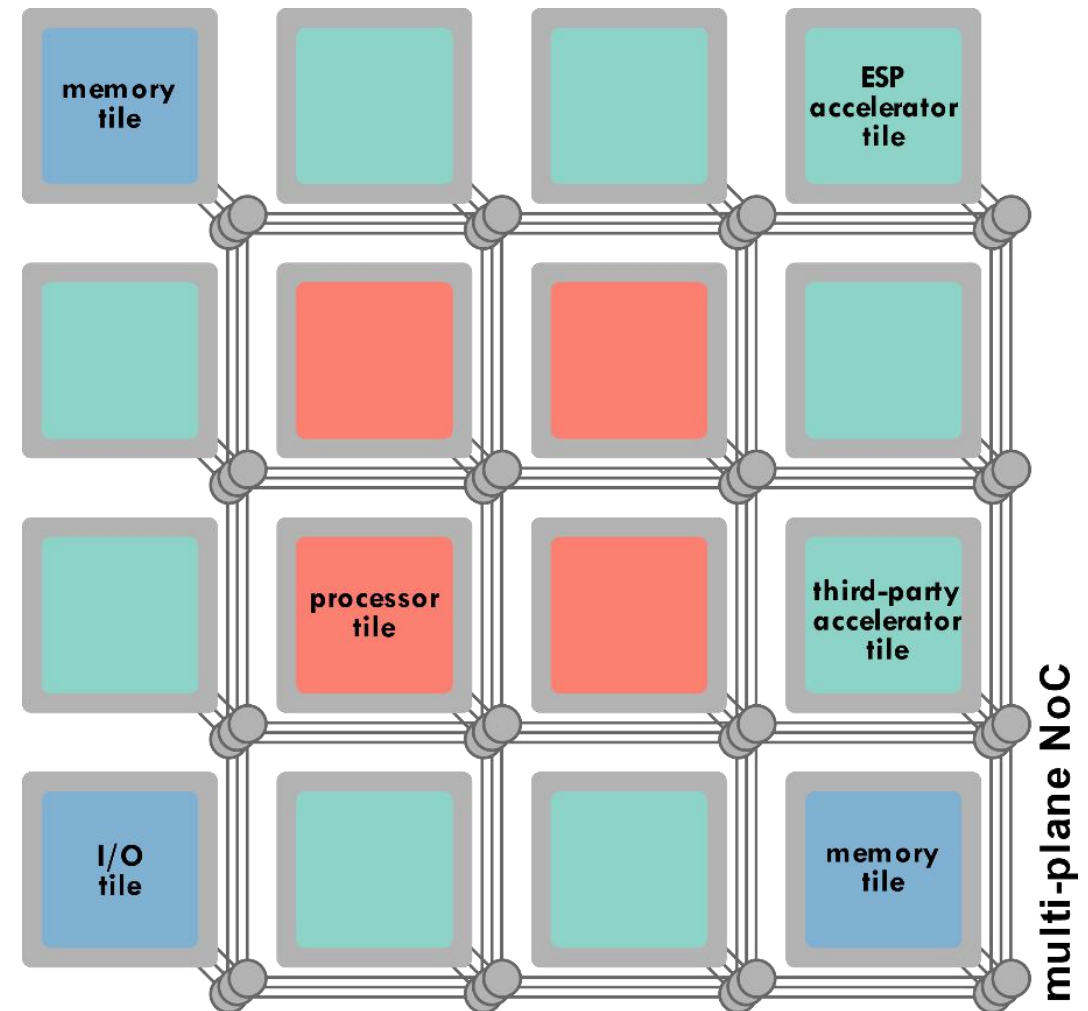
[P. Mantovani, L. P. Carloni et al., *An FPGA-Based Infrastructure for Fine-Grained DVFS Analysis in High-Performance Embedded Systems*, DAC 2016]



# ESP Architecture

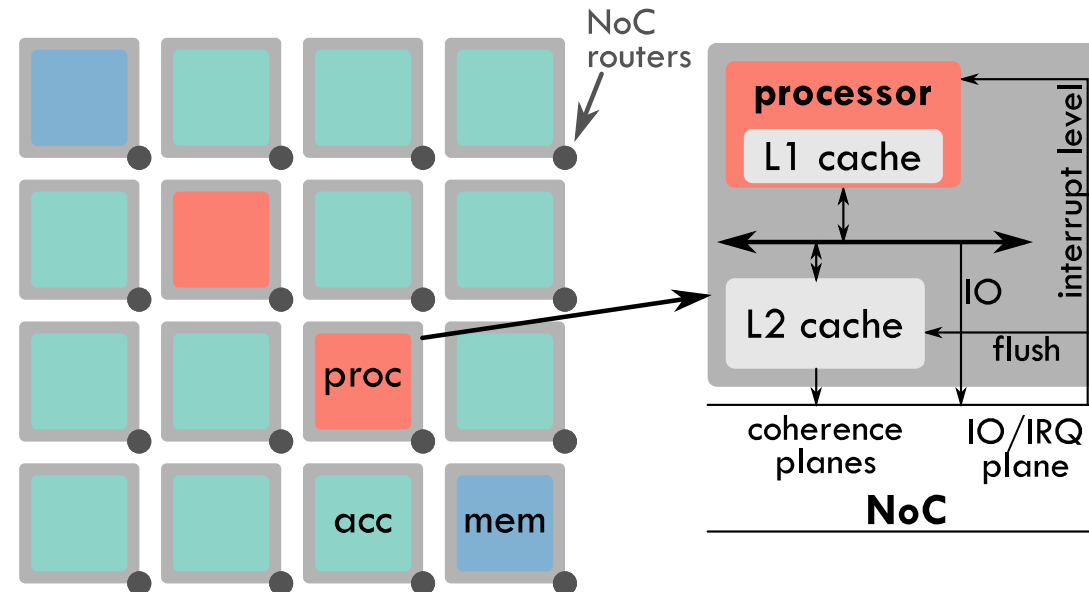
- RISC-V Processors
- Many-Accelerator
- Distributed Memory
- Multi-Plane NoC

The ESP architecture implements a **distributed** system, which is **scalable**, **modular** and **heterogeneous**, giving processors and accelerators similar weight in the SoC



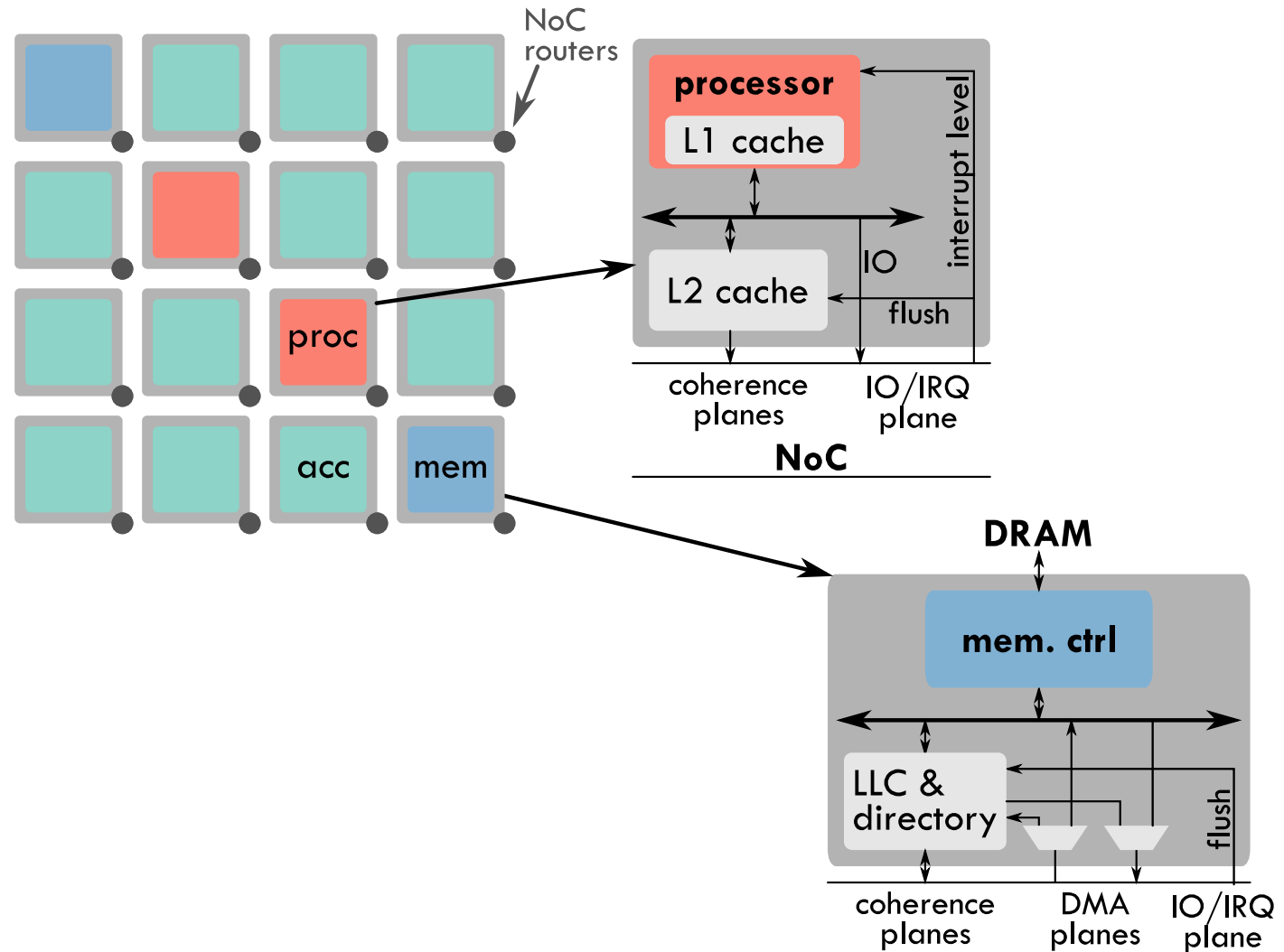
# ESP Architecture: Processor Tile

- Processor off-the-shelf
  - RISC-V Ariane (64 bit)
  - SPARC V8 Leon3 (32 bit)
  - L1 private cache
- L2 private cache
  - Configurable size
  - MESI protocol
- IO/IRQ channel
  - Un-cached
  - Accelerator config. registers, interrupts, flush, UART, ...



# ESP Architecture: Memory Tile

- External Memory Channel
- LLC and directory partition
  - Configurable size
  - Extended MESI protocol
  - Supports coherent-DMA for accelerators
- DMA channels
- IO/IRQ channel

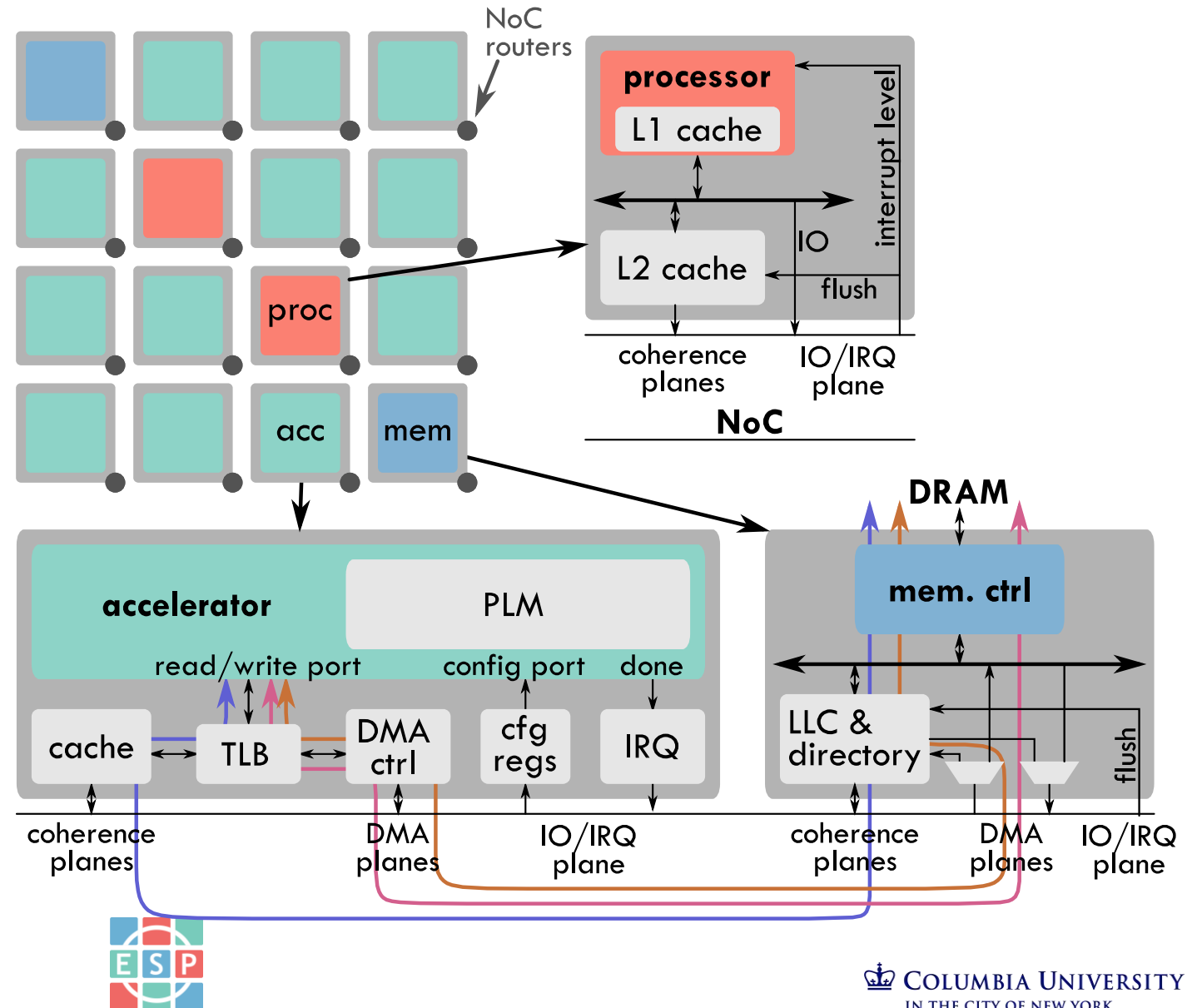




# ESP Architecture: Accelerator Tile

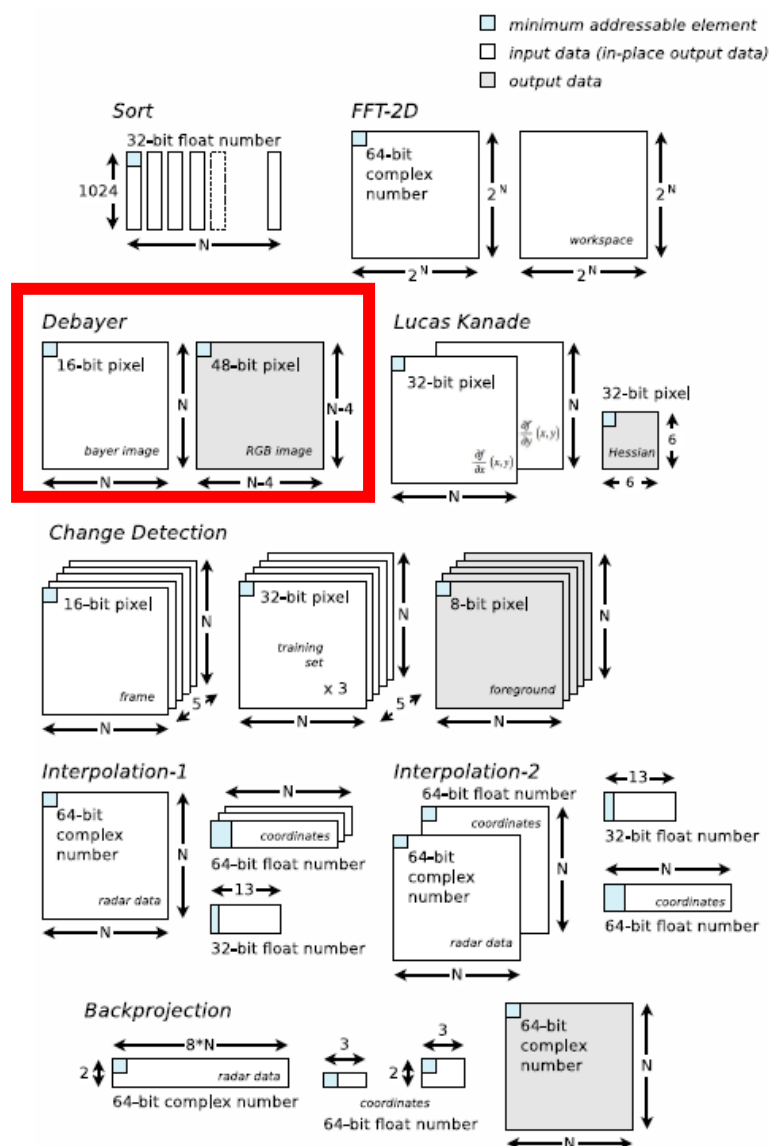
- Accelerator Socket w/ Platform Services

- Direct-memory-access
- Run-time selection of coherence model:
  - Fully coherent
  - LLC coherent
  - Non coherent
- User-defined registers
- Distributed interrupt

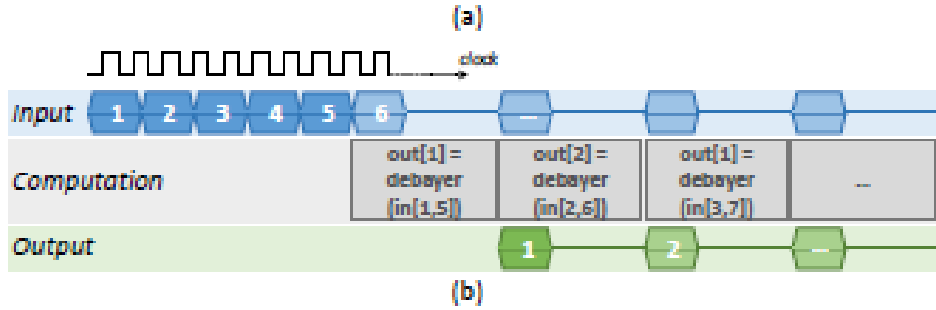
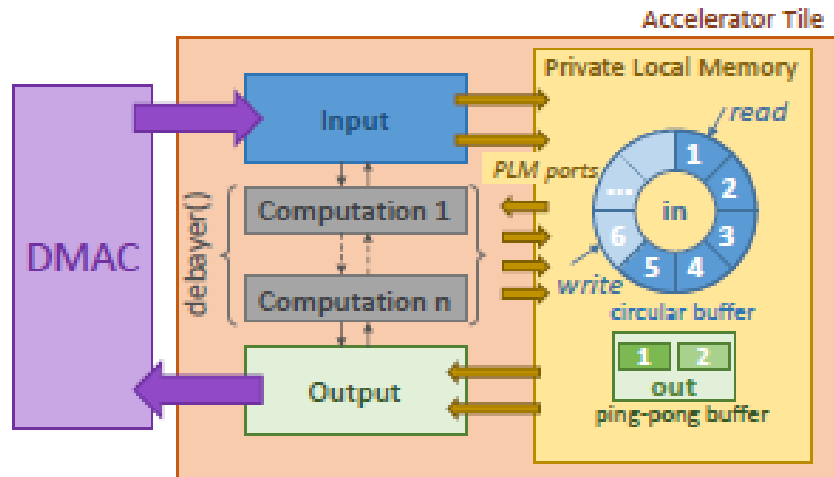


# Heterogeneous Applications Bring Heterogeneous Requirements

## Data Structures of the PERFECT TAV Benchmarks



## Structure and Behavior of the Debayer Accelerator



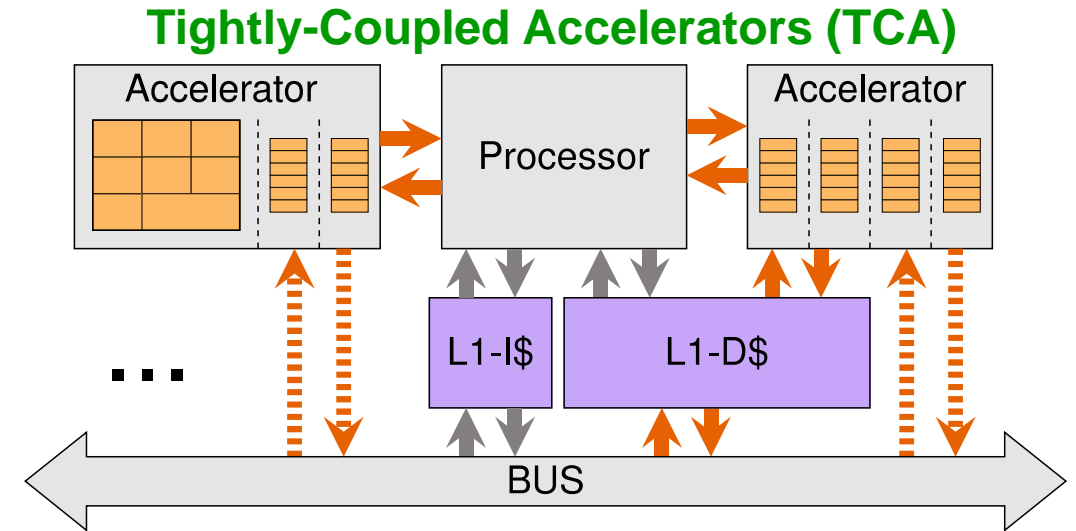
- While the Debayer structure and behavior is representative of the other benchmarks, the specifics of the actual computations, I/O patterns, and scratchpad memories vary greatly among them



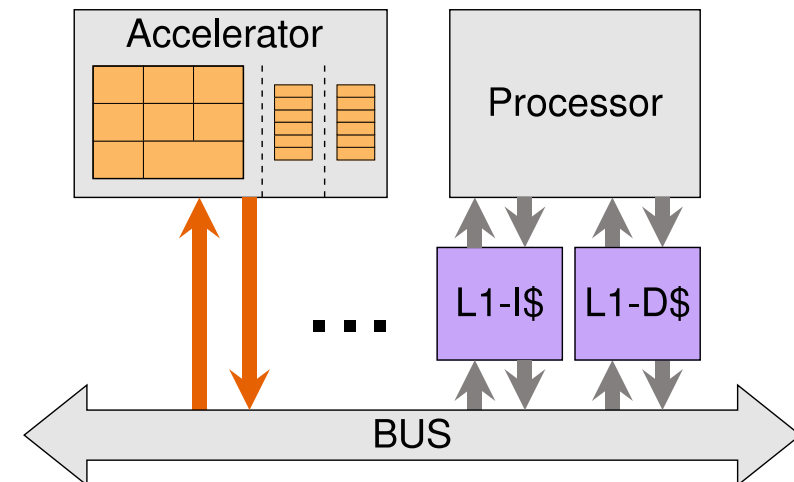
# How to Couple Accelerators, Processors and Memories?

- **Private local memories (aka scratchpads)** are key to performance and energy efficiency of accelerators
- There are two main models of coupling accelerators with processors, memories
  - Tightly-Coupled Accelerators
  - Loosely-Coupled Accelerators

[ E. G. Cota, P. Mantovani, G. Di Guglielmo, and L. P. Carloni,  
An Analysis of Accelerator Coupling in Heterogeneous  
Architectures, DAC'15]

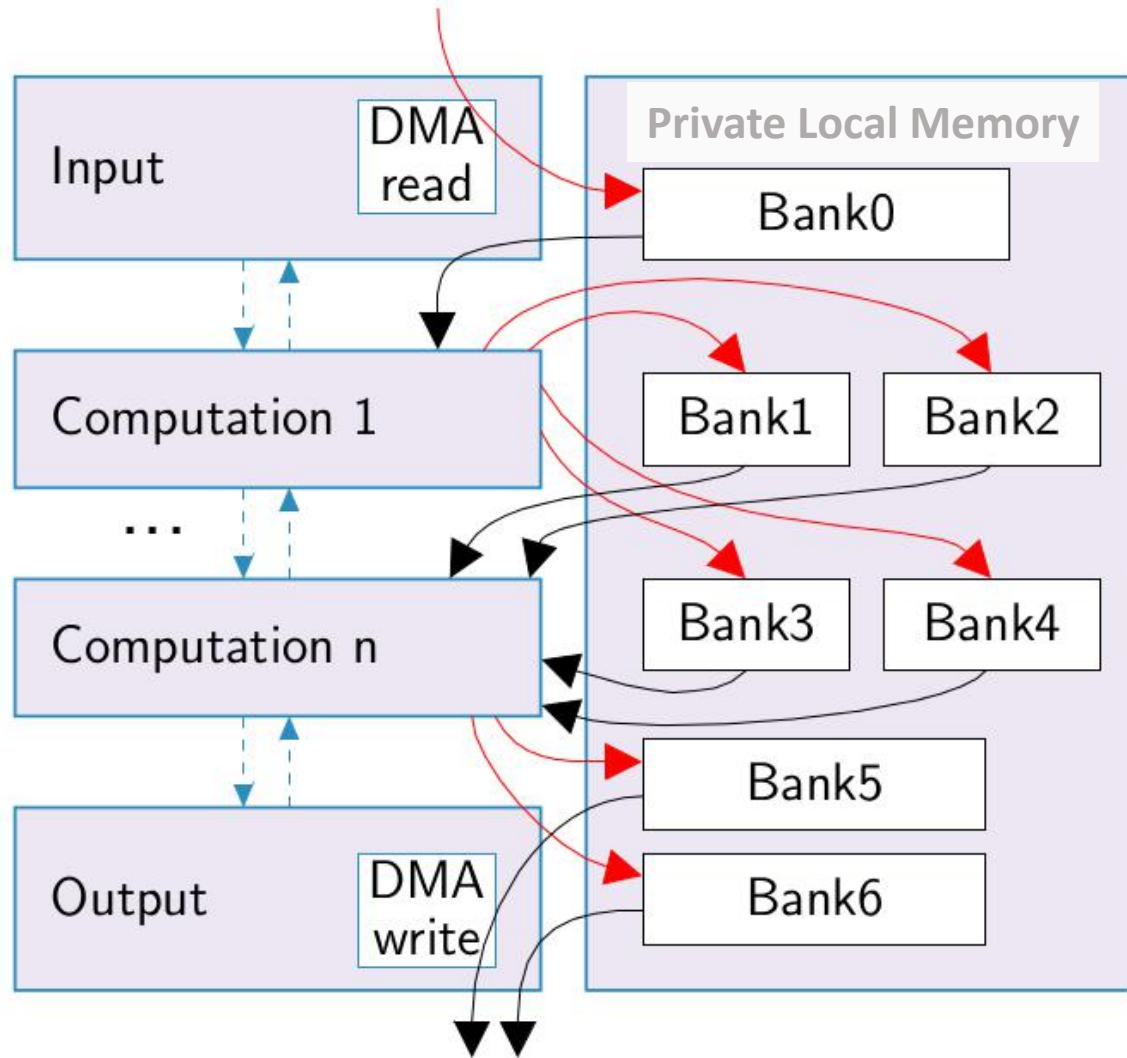


## Loosely-Coupled Accelerators (LCA)





# The Key Role of the Private Local Memories (PLM)



- Tailored, many-ported PLMs are **key to accelerator performance**
- A scratchpad features **aggressive SRAM banking** that provides multi-port memory accesses to match the multiple parallel blocks of the computation datapath
  - Level-1 caches cannot match this parallelism

[C. Pilato, P. Mantovani, G. Di Guglielmo, and L. P. Carloni, System-Level Optimization of Accelerator Local Memory for Heterogeneous Systems-on-Chip. IEEE Trans. on CAD of Integrated Circuits and Systems, 2017. ]



# Exploiting PLMs to Reduce the Opportunity Cost of Accelerator Integration

- **Two facts:**

1. Accelerators are made mostly of memory
2. Average utilization of accelerator PLMs is low

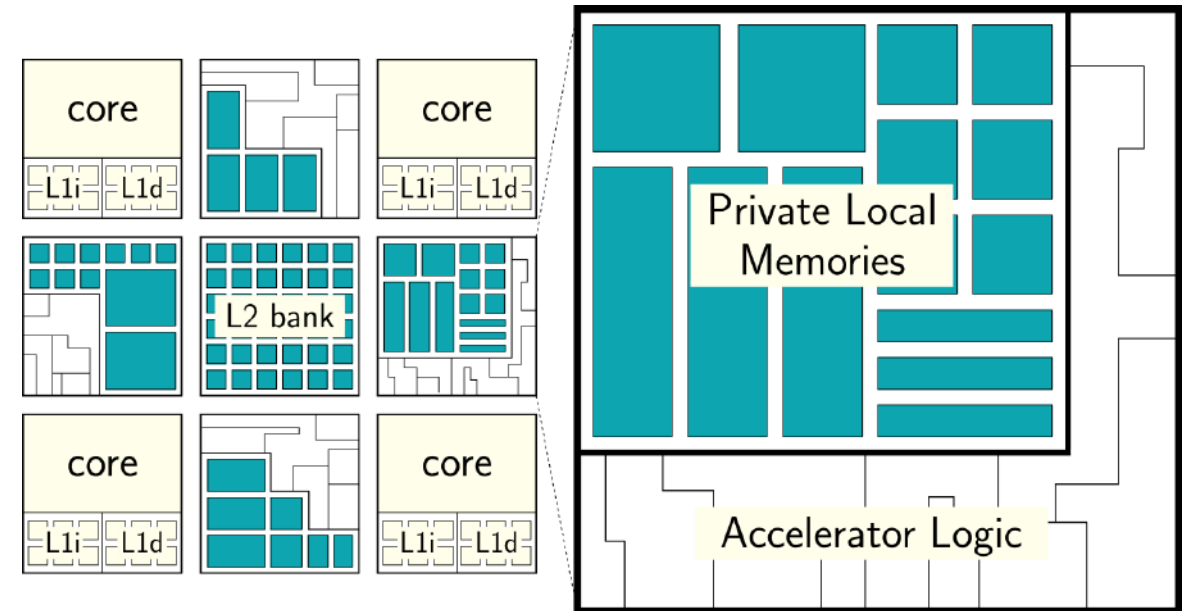
- **Main observation:**

- The accelerator PLM provide a de facto NUCA substrate

- **Key Idea:**

- Extend the last level cache with the PLMs of those accelerators that are not in use

[E. Cota, P. Mantovani, and L. P. Carloni, Exploiting Private Local Memories to Reduce the Opportunity Cost of Accelerator Integration, ICS '16]

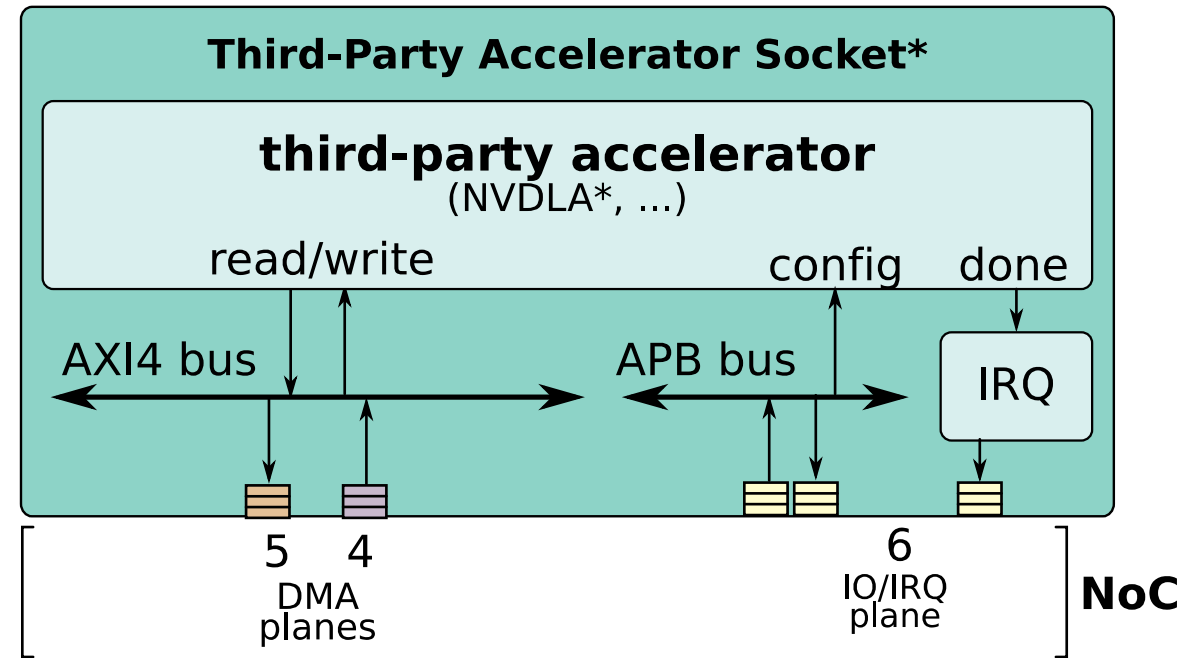
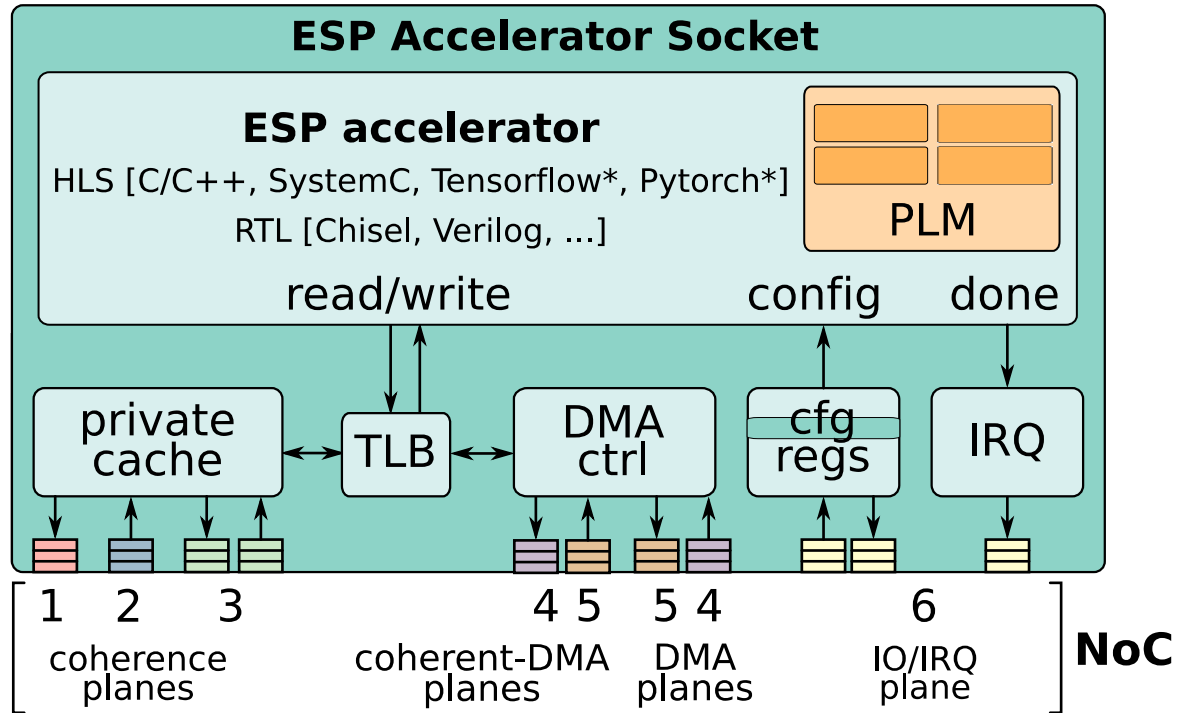


- **Implementation:**

- Minimal modification to accelerators
- Minimal area overhead
- Good Performance: a 6MB ROCA can realize ~70% of the performance/energy efficiency benefits of a same-area 8MB S-NUCA



# ESP Accelerator Socket

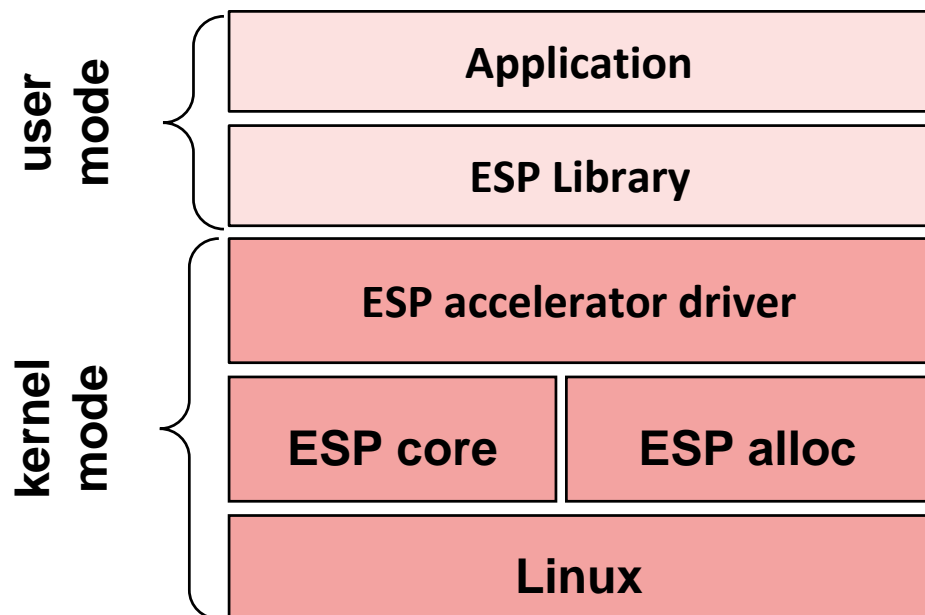




# ESP Software Socket

- **ESP accelerator API**

- Generation of device driver and unit-test application
- Seamless shared memory

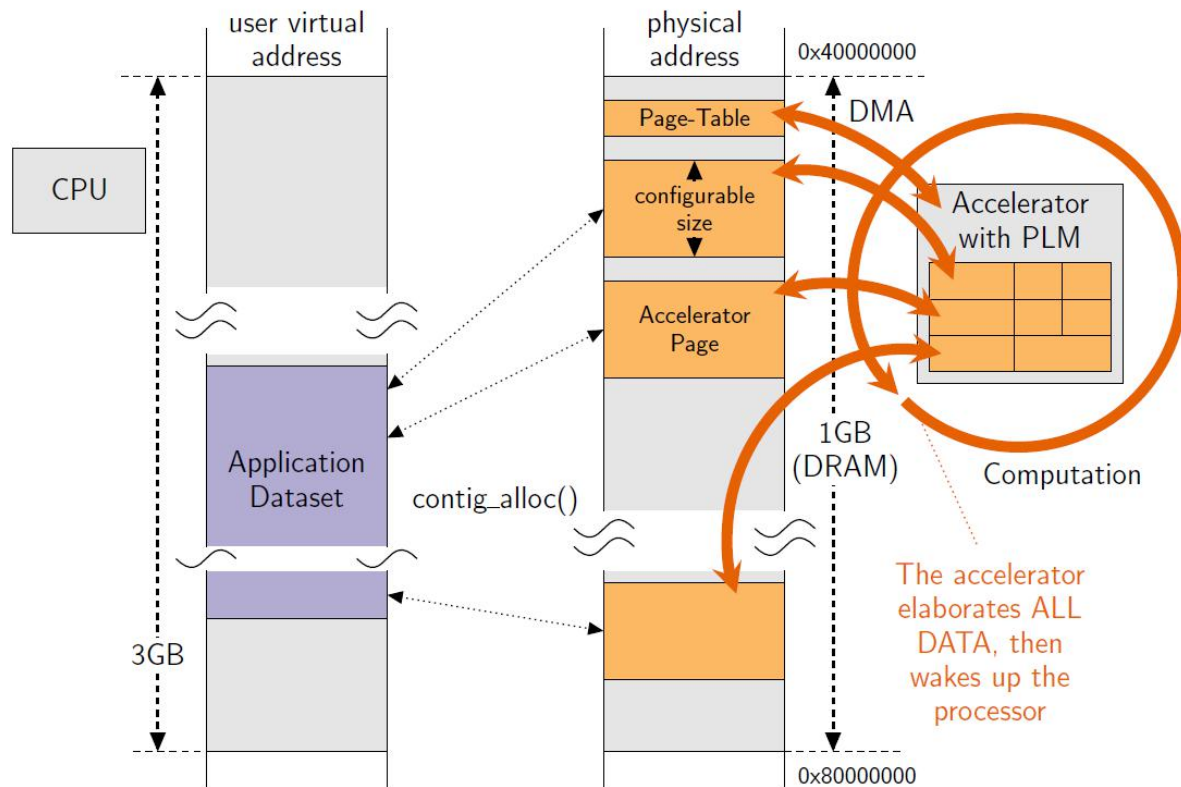


```
/*  
 * Example of existing C application  
 * with ESP accelerators that replace  
 * software kernels 2, 3 and 5  
 */  
{  
    int *buffer = esp_alloc(size);  
    for (...) {  
        kernel_1(buffer,...); /* existing software */  
        esp_run(cfg_k2);      /* run accelerator(s) */  
        esp_run(cfg_k3);  
        kernel_4(buffer,...); /* existing software */  
        esp_run(cfg_k5);  
    }  
    validate(buffer);          /* existing checks */  
    esp_cleanup();             /* memory free */  
}
```



# The Large Data Set Problem for SoC Accelerators

- Finding a high-performance and low-overhead mechanism that allows hardware accelerators to process large data sets without incurring penalties for data transfers



## • Solution :

- a low-overhead accelerator virtual address space, which is distinct from the processor virtual address space;
- direct sharing of physical memory across processors and accelerators;
- a dedicated DMA controller with specialized translation look aside buffer (TLB) per accelerator;
- hardware and software support for implementing run-time policies to balance traffic among available DRAM channels.

[ P. Mantovani, E. Cota, C. Pilato, G. Di Guglielmo and L. P. Carloni, Handling Large Data Sets for High-Performance Embedded Applications in Heterogeneous Systems-on-Chip. CASES 2016]



# ESP Platform Services

## Accelerator tile

DMA

Reconfigurable coherence

Point-to-point

ESP or AXI interface

DVFS controller

## Processor Tile

Coherence

I/O and un-cached memory

Distributed interrupts

DVFS controller

## Miscellaneous Tile

Debug interface

Performance counters access

Coherent DMA

Shared peripherals (UART, ETH, ...)

## Memory Tile

Independent DDR Channel

LLC Slice

DMA Handler





# The Twofold Role of the Network-on-Chip

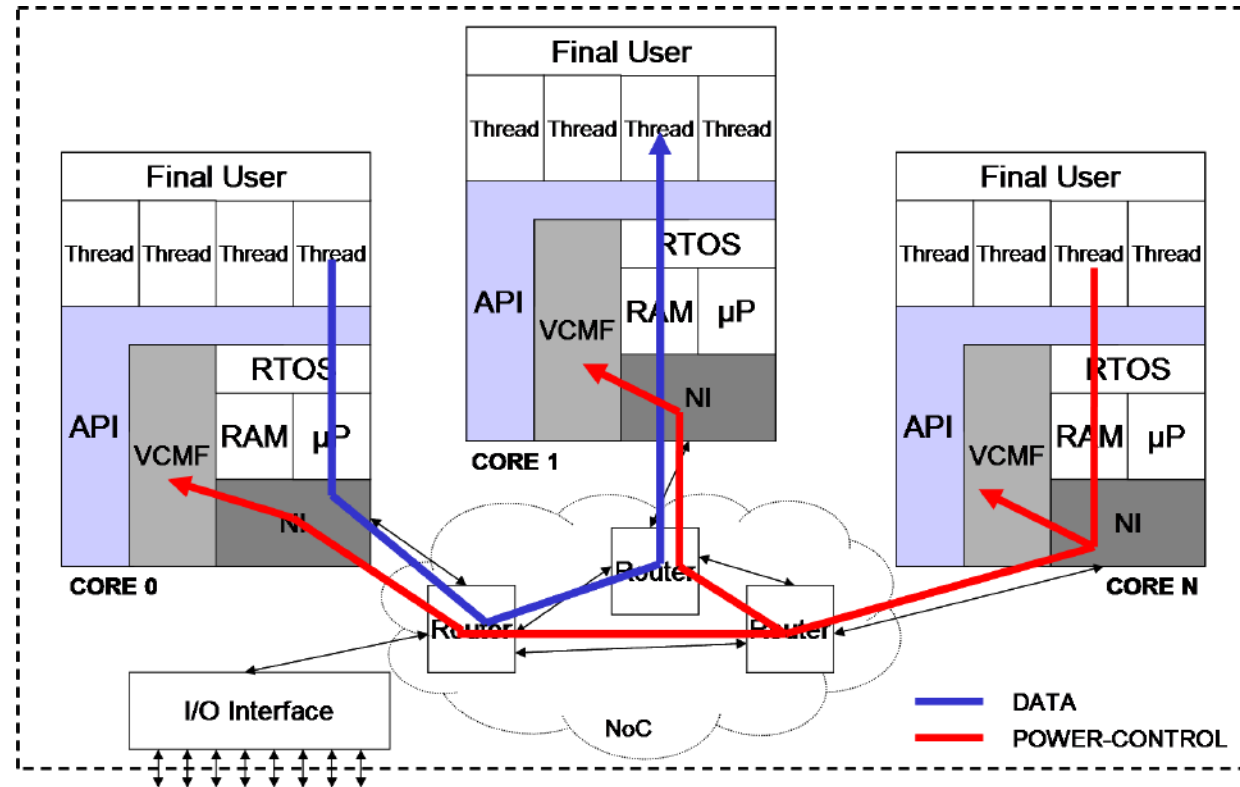
- A scalable NoC is instrumental to accommodate heterogeneous concurrency and computing locality in ESP

- **At Design Time**

- simplifies integration of heterogeneous tiles to balance regularity and specialization

- **At Run Time**

- energy efficient inter-tile data communication with integrated support for fine-grain power management and other services



- The NoC Interface interacts directly with the Tile Socket that supports the **ESP Platform Services**

- communication/synchronization channels among tiles
  - fine-grain power management with dynamic voltage-frequency scaling
  - seamless dynamic support for various accelerator coherence models



# Cache Coherence and Loosely-Coupled Accelerators

- An analysis of the literature indicates that there are three main cache-coherence models for loosely-coupled accelerators:

## 1. Non-Coherent Accelerator

- the accelerator operates through DMA bypassing the processor caches

## 2. Fully-Coherent Accelerator

- the accelerator issues main-memory requests that are coherent with the entire cache hierarchy
  - this approach can endow accelerators with a private cache, thus requiring no updates to the coherence protocol

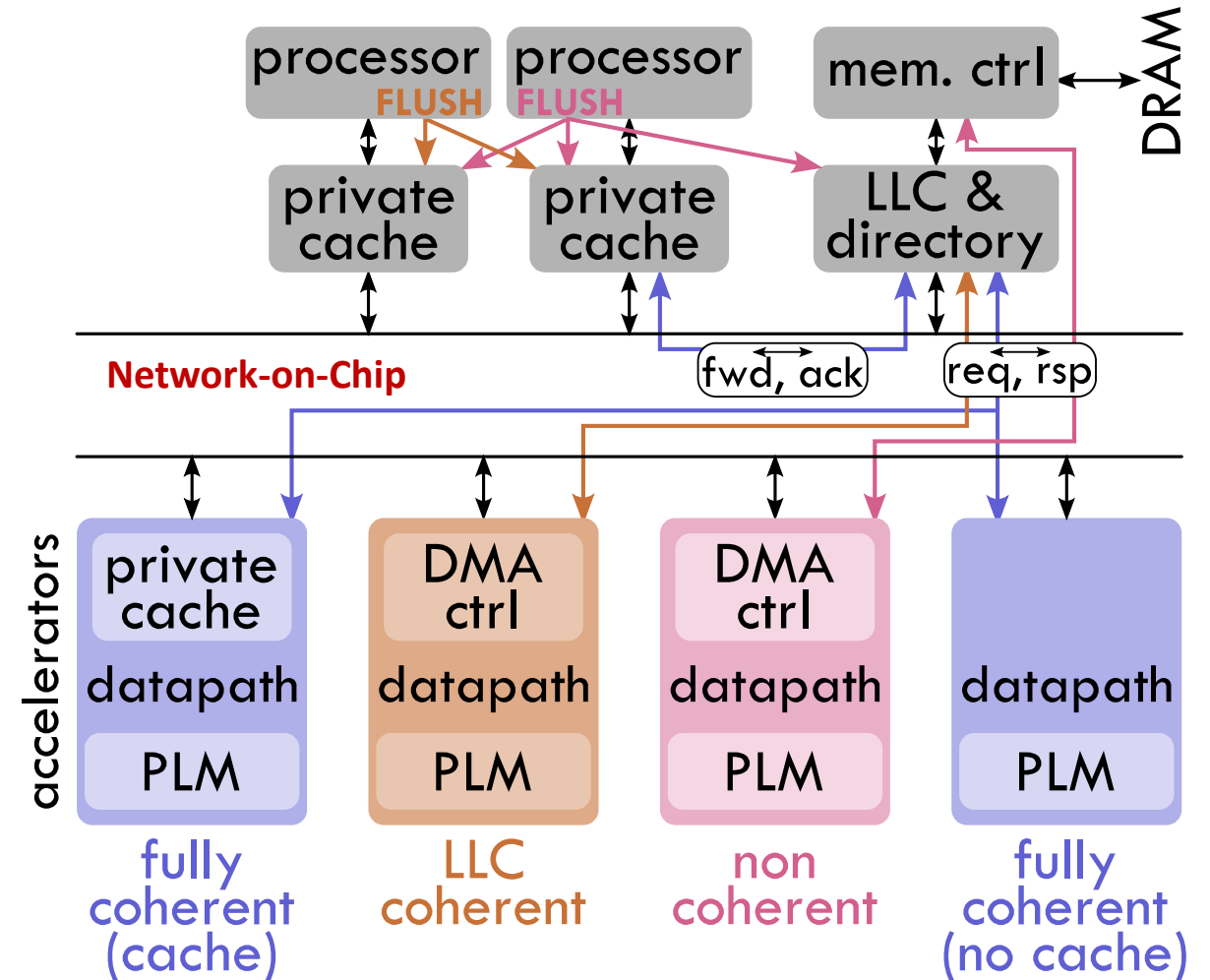
## 3. Last Level Cache (LLC)-Coherent Accelerator

- the accelerator issues main-memory requests that are coherent with the LLC, but not with the private caches of the processors
  - in this case, DMA transactions address the shared LLC, rather than off-chip main memory



# Example: NoC Services to Support Heterogeneous Cache-Coherence Models for Accelerators

- **Seamless dynamic support for 3 coherence models:**
  - Fully coherent accelerators
  - Non-coherent accelerators
  - Last-Level-Cache (LCC) coherent accelerators

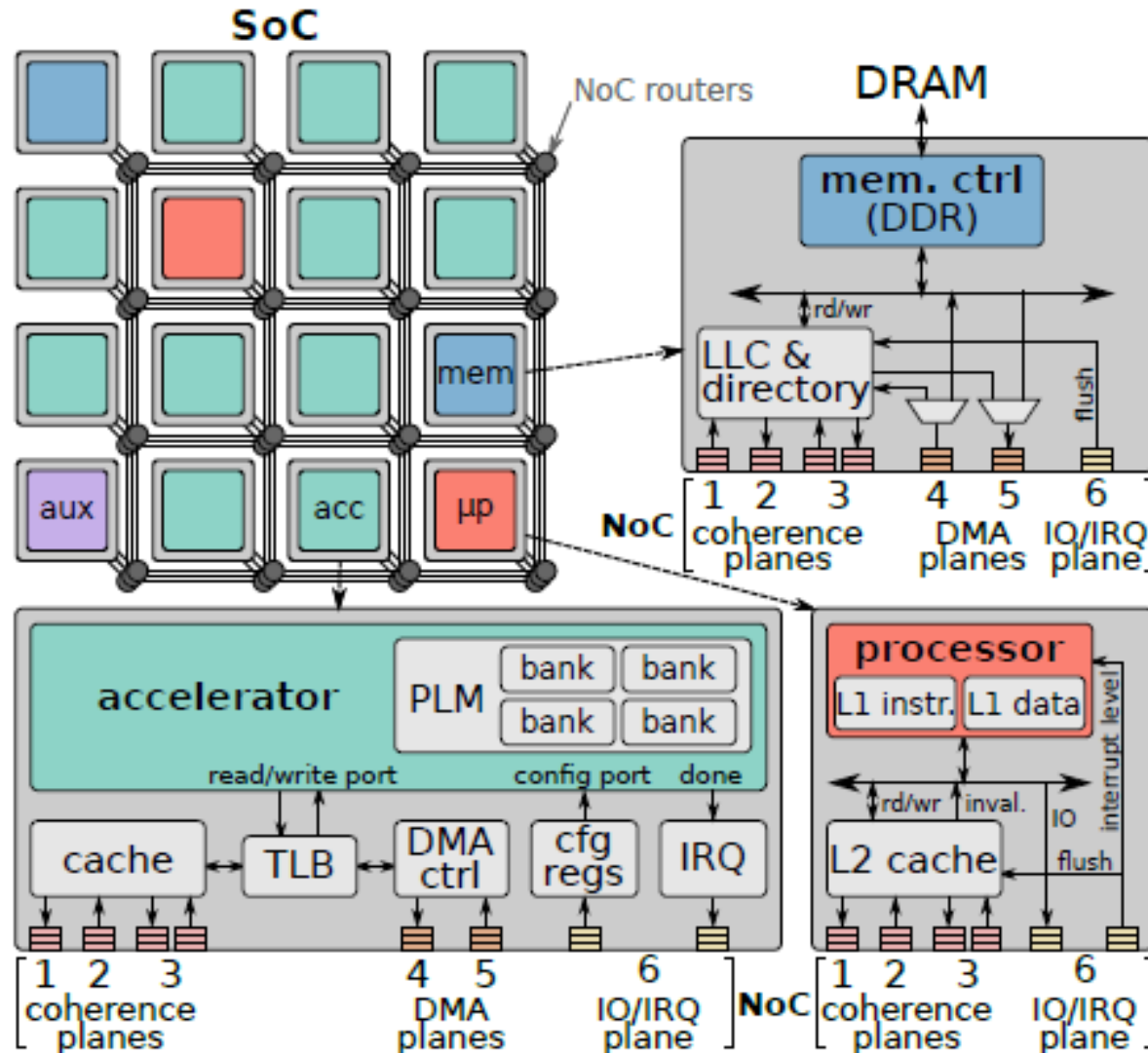


[D. Giri, P. Mantovani, and L. P. Carloni, *Accelerators & Coherence: An SoC Perspective*. IEEE MICRO, 2018. ]





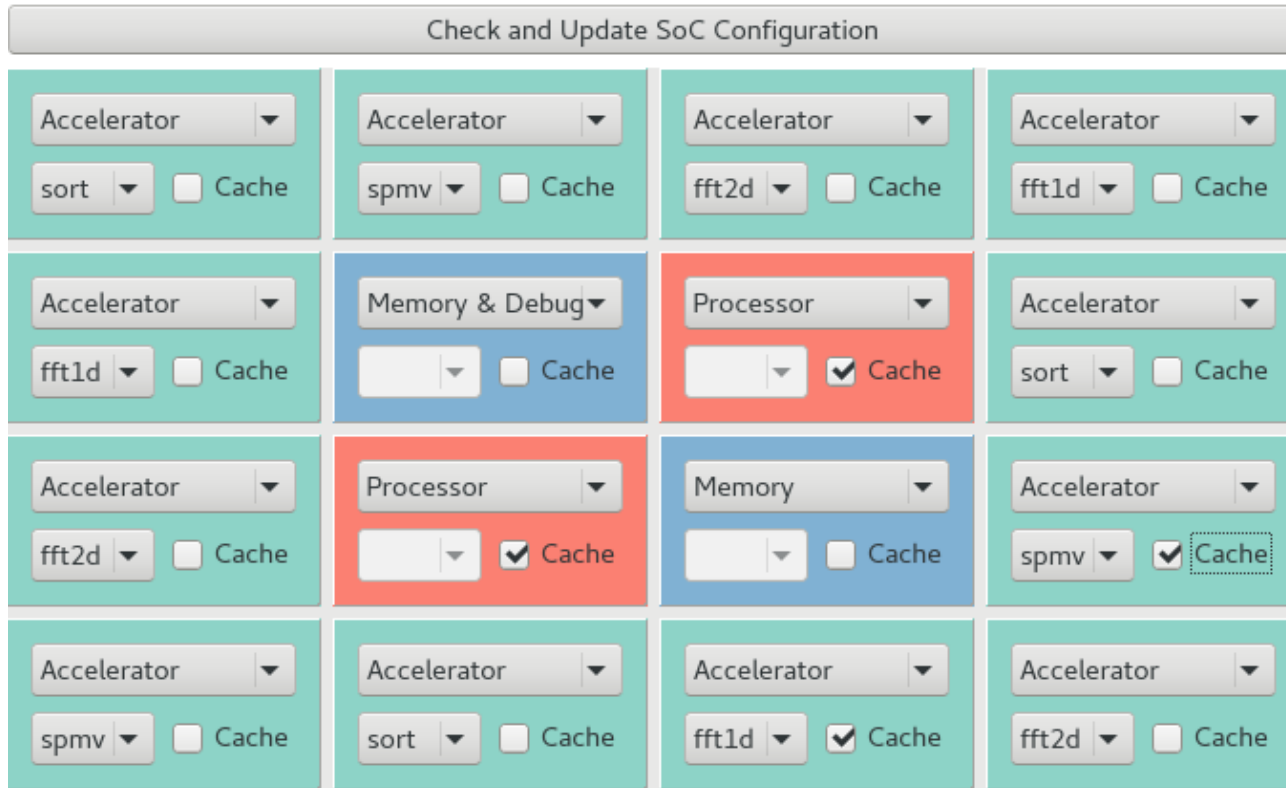
# Extending ESP to Support Heterogeneous Cache-Coherence Models for Accelerators



- First NoC-based system enabling the three models of coherence for accelerators to coexist and operate simultaneously through run-time selection in the same SoC
  - Design based on ESP Platform Services
- Extension of the MESI directory-based protocol to integrate LLC-coherent accelerators into an SoC
  - The design leverages the tile-based architecture of ESP over a packet-switched NoC to guarantee scalability and modularity



# Heterogeneous Coherence Implementation

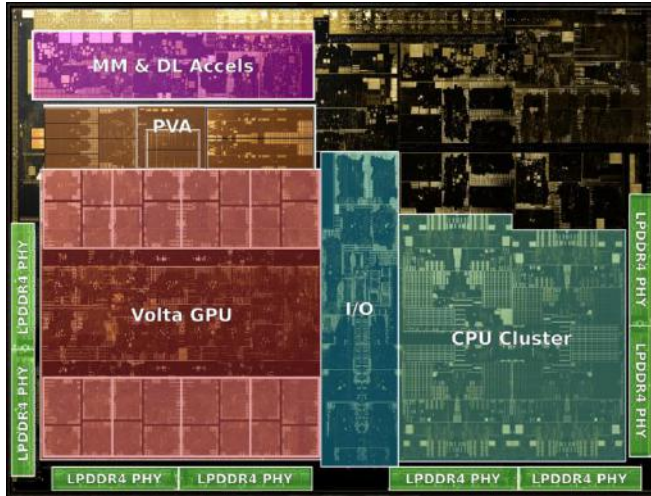


[D. Giri, P. Mantovani, L. P. Carloni, *Accelerators & Coherence: An SoC Perspective*, IEEE Micro, Nov/Dec 2018]

- **The CAD Infrastructure of ESP allows**
  - direct instantiation of heterogeneous configurable components from predesigned libraries
  - fully automated flow from the GUI to the bitstream for FPGAs
- **Extension of ESP to support atomic test-and-set and compare-and-swap operations over the NoC allows**
  - running multi-processor and multi-accelerator applications on top of Linux SMP



# Outline

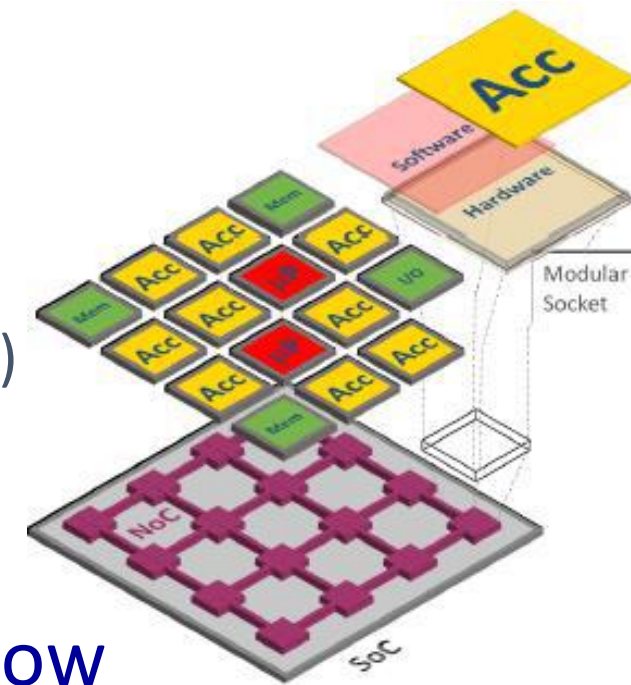


## 1. Motivation

- The Rise of Heterogeneous Computing

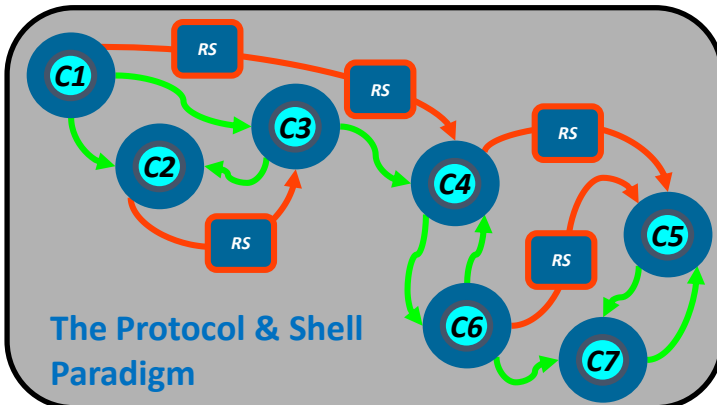
## 2. Proposed Architecture

- Embedded Scalable Platforms (ESP)

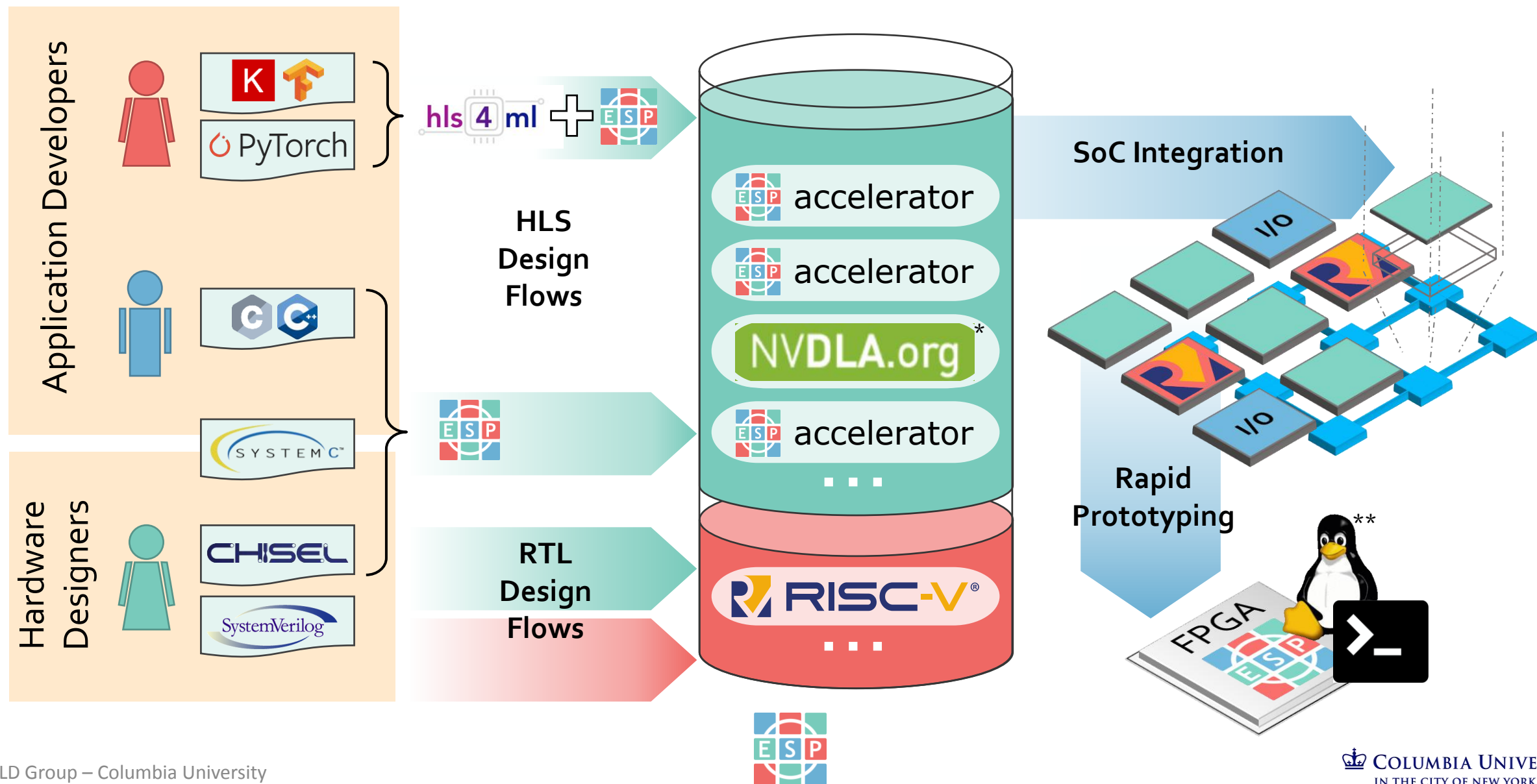


## 3. Methodology and Design Flow

- with a Retrospective on Latency-Insensitive Design

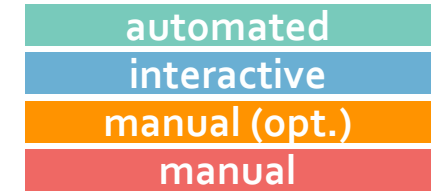


# ESP Vision: Domain Experts Can Design SoCs

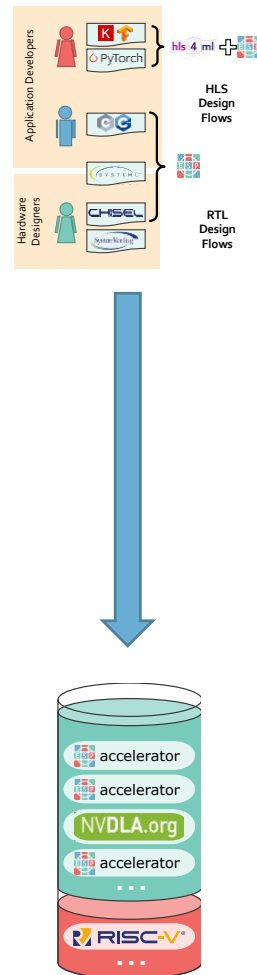
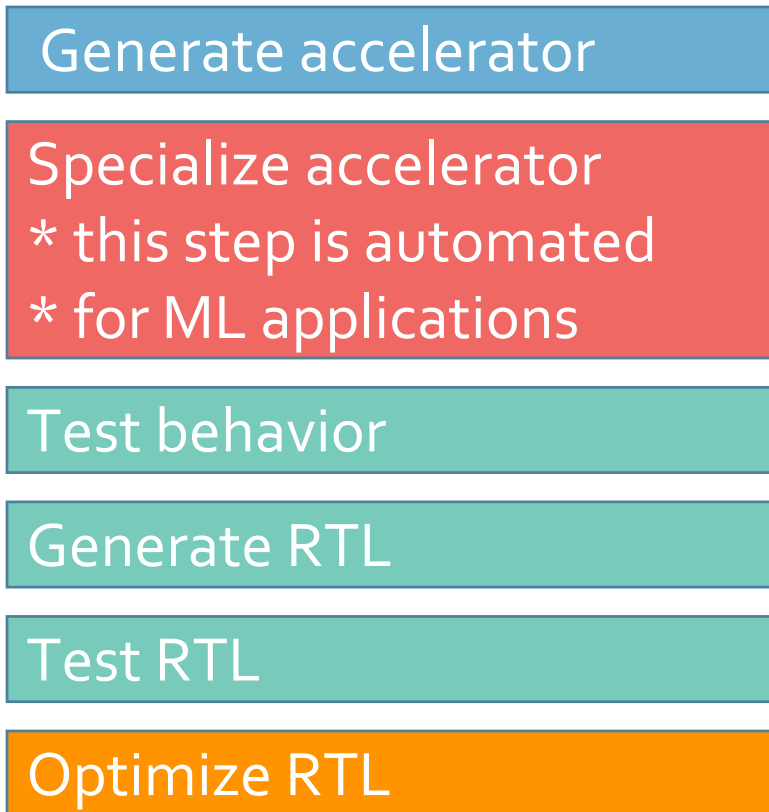




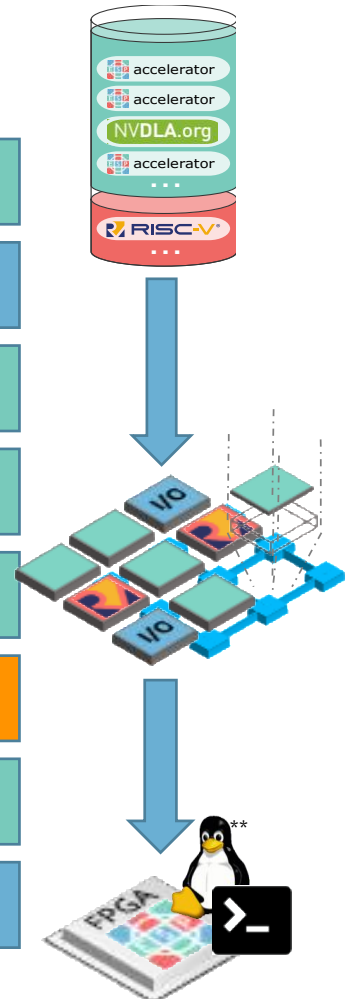
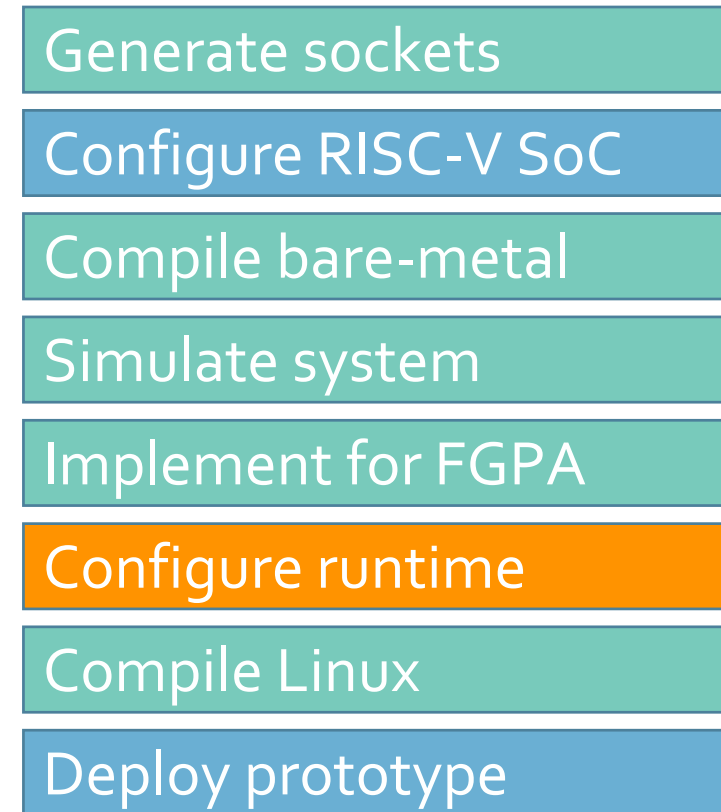
# ESP Methodology In Practice



## Accelerator Flow



## SoC Flow



# ESP Design Example: An Accelerator for WAMI

- The PERFECT WAMI-app is an image processing pipeline in behavioral C code

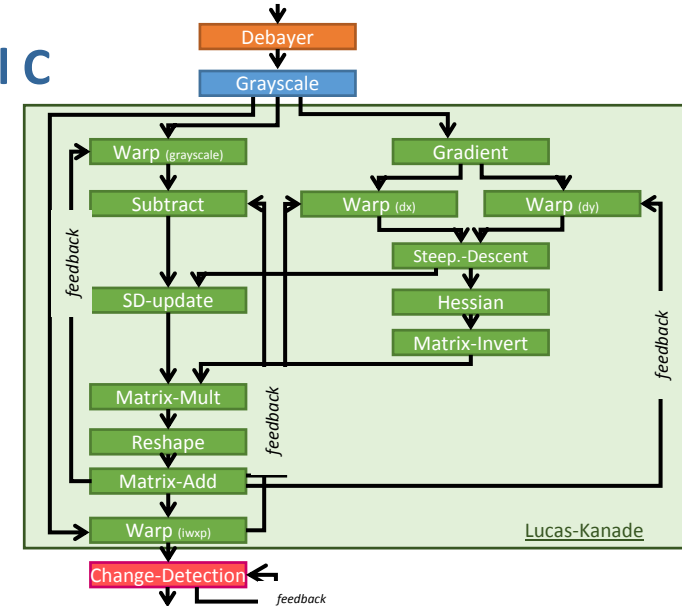
- From a sequence of frames it extracts masks of “meaningfully” changed pixels



- Complex data-dependency among kernels
- Computational intensive matrix operations
  - Global-memory access to compute ratio **45%**
  - Floating-point operation to compute ratio **15%**

- We designed 12 accelerators starting from a C “programmer-view” reference implementation

- Methodology to port C into synthesizable SystemC
- Automatic generation of customized RTL memory subsystems for each accelerator



Kernels	Lines of Code		
	C	SystemC	RTL
Debayer	195	664	8440
Grayscale	21	368	4079
Warp	88	571	6601
Gradient	65	540	12163
Subtract	36	379	4684
Steep.-Descent	34	410	8744
SD-Update	55	383	7864
Hessian	43	358	7042
Matrix-Invert	166	388	7392
Matrix-Mult	55	307	2708
Reshape	42	269	2160
Matrix-Add	36	287	2310
Change-Detect.	128	939	18416
Total	964	5863	92603

[P. Mantovani, G. Di Guglielmo, and L. P. Carloni, High-Level Synthesis of Accelerators in Embedded Scalable Platforms, ASPDAC 2016]



# Example of Accelerator Design with HLS: Debayer - 1

```

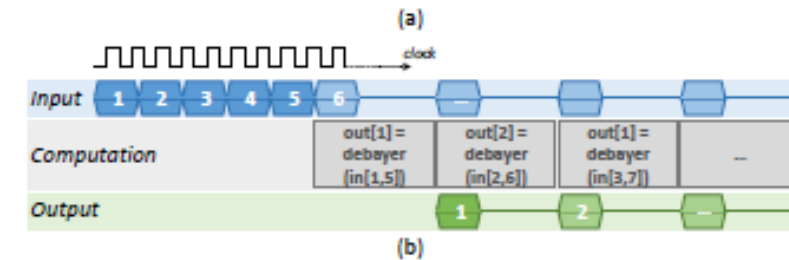
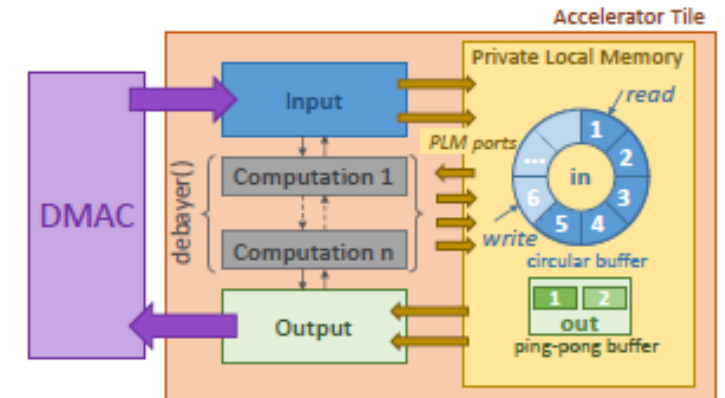
1 #include <systemc.h>
2 SC_MODULE(Debayer) {
3     sc_in<bool> clk, rst;
4 private:
5     sc_signal<bool> i_valid, i_ready, o_valid, o_ready;
6     int A0[6][2048]; // circular buffer
7     int B0[2048];
8     int B1[2048];
9 public:
10    //...
11    SC_CTOR(Debayer) {
12        SC_CTHREAD(input, clk.pos());
13        reset_signal_is(rst, false);
14        SC_CTHREAD(compute, clk.pos());
15        reset_signal_is(rst, false);
16        SC_CTHREAD(output, clk.pos());
17        reset_signal_is(rst, false);
18        //...
19    }
20    void input(void) {
21        // reset ...
22        unsigned circ = 0; // circular buffer write pointer
23        wait();
24        while(true) {
25            L0: for (int r=0; r<2048; r++) {
26                // DMA request
27                // read input ...
28                L1: for (int c=0; c<2048; c++)
29                    { A0[circ][c] = f(...); } //write to A0
30                // output ...
31                if (r >= 5) {
32                    // wait for ready from compute then notify as valid
33                }
34                circ++;
35                if (circ == 6)
36                    circ = 0;
37            }
38        }
39    }

```

```

40 void compute(void) {
41     int PAD = 2; bool flag = true;
42     int r_r = 0; // central row of the mask
43     // reset ...
44     wait();
45     while(true) {
46         L2: for (int r=0; r<2048-PAD; r++) {
47             // (wait for valid from input then notify as ready)
48             r_r = circ_buffer_row(r + 2);
49             L3: for (int j=PAD; j<2048-PAD; j++) {
50                 if (flag) B0[j] = g(A0[r_r][j-2], A0[r_r][j-1],
51                                     A0[r_r][j], A0[r_r][j+1], A0[r_r][j+2], ...);
52                 else B1[j] = g(A0[r_r][j-2], A0[r_r][j-1],
53                                 A0[r_r][j], A0[r_r][j+1], A0[r_r][j+2], ...);
54             }
55             // (valid to output, ready to compute)
56             flag = !flag;
57         }
58     }
59 }
60 void output(void) {
61     int PAD = 2; bool flag = true;
62     // reset ...
63     wait();
64     while(true) {
65         L4: for (int r=PAD; r<2048-PAD; r++) {
66             // (wait for valid from compute then notify as ready)
67             // prepare DMA request
68             // send data
69             L5: for (int c=PAD; c<2048-PAD; c++) {
70                 if (flag) h(B0[c], ...); //read from array B0
71                 else h(B1[c], ...); //read from array B1
72             }
73             // (ready to compute)
74             flag = !flag;
75         }
76     }
77 }
78 };

```

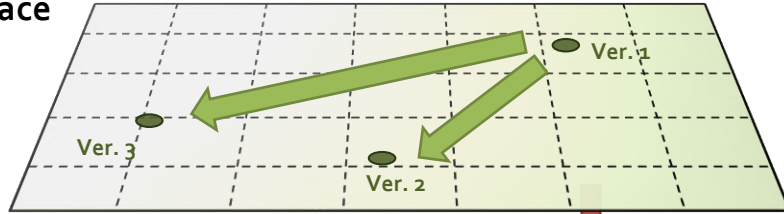


- The **3 processes** execute in pipeline
  - on a 2048×2048-pixel image, which is stored in DRAM, to produce the corresponding debayered version
- The **circular buffer** allows the reuse of local data, thus minimizing the data transfers with DRAM
- The **ping-pong buffer** allows the overlapping of computation and communication

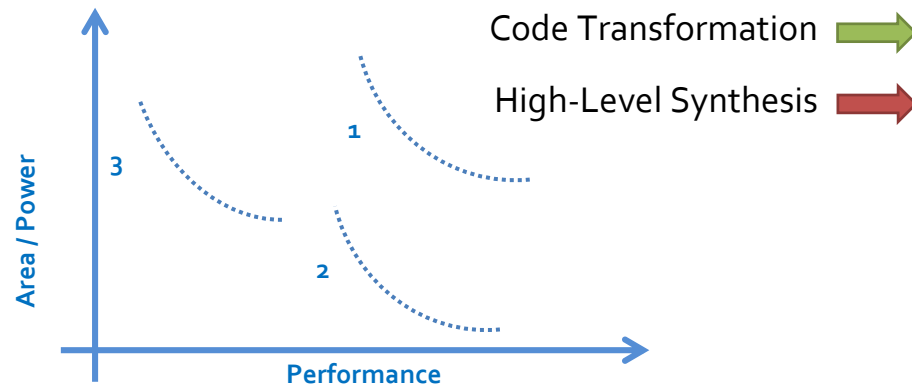
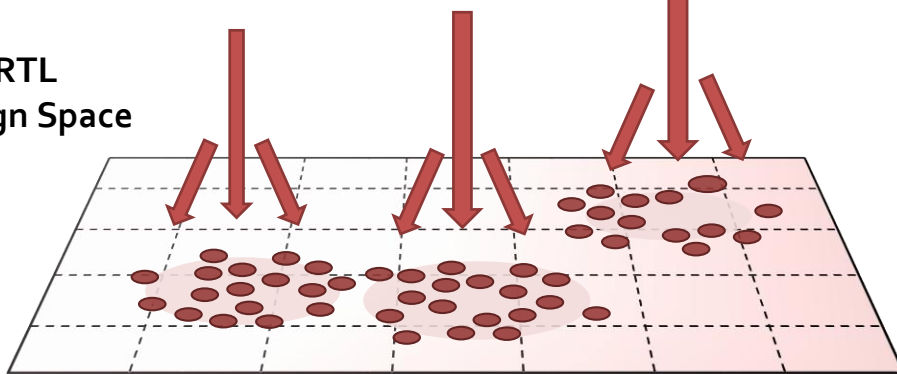


# High-Level Synthesis Drives Design-Space Exploration

Programmer View  
Design Space



RTL  
Design Space



- Given a SystemC specification, HLS tools provide a rich set of **configuration knobs** to synthesize a variety of RTL implementations
  - these implementations have different micro-architectures and provide different cost-performance trade-offs
- Engineers can focus on revising the high-level specification
  - to expose more parallelism, remove false dependencies, increase resource sharing...





# Example of Design-Space Exploration: Accelerator for the SAR Interp-1 Kernel

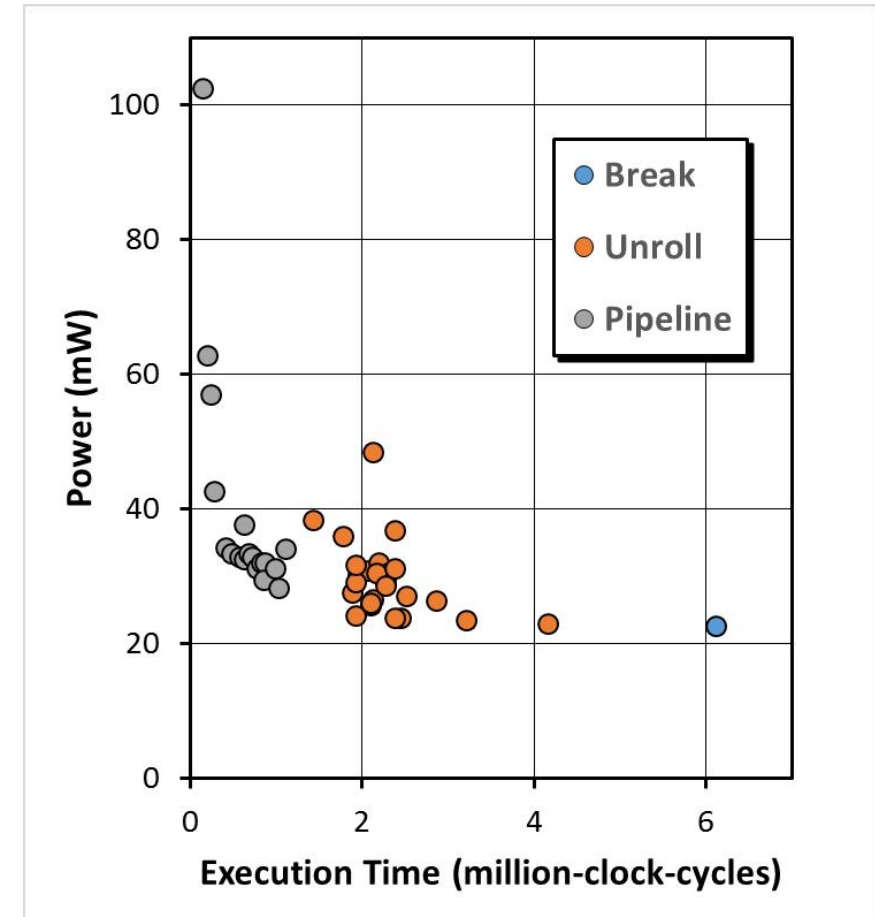
## Main loop in *Interpolation-1* kernel

```
function interp1()  
{  
  for(...)  
  {  
    accum = 0;  
    for(...)  
    {  
      accum += sinc(input);  
    }  
    store(accum);  
  }  
}
```

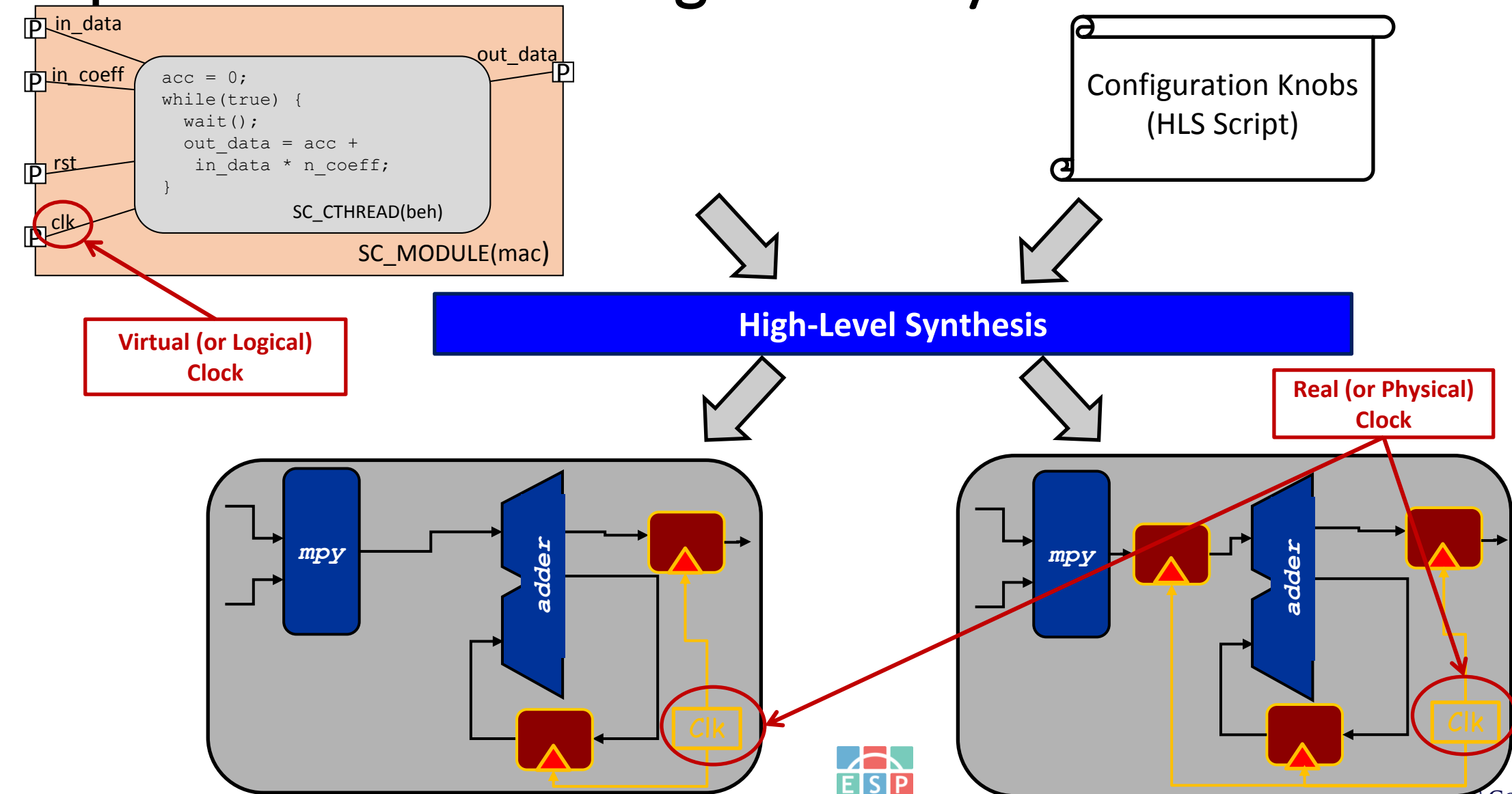


- Presence of expensive combinational function (sinc()) in the inner most loop
- Use of “loop knobs” provided by HLS tools to optimize for power and performance
- Derivation of Pareto set highlighting Power-Performance trade-offs

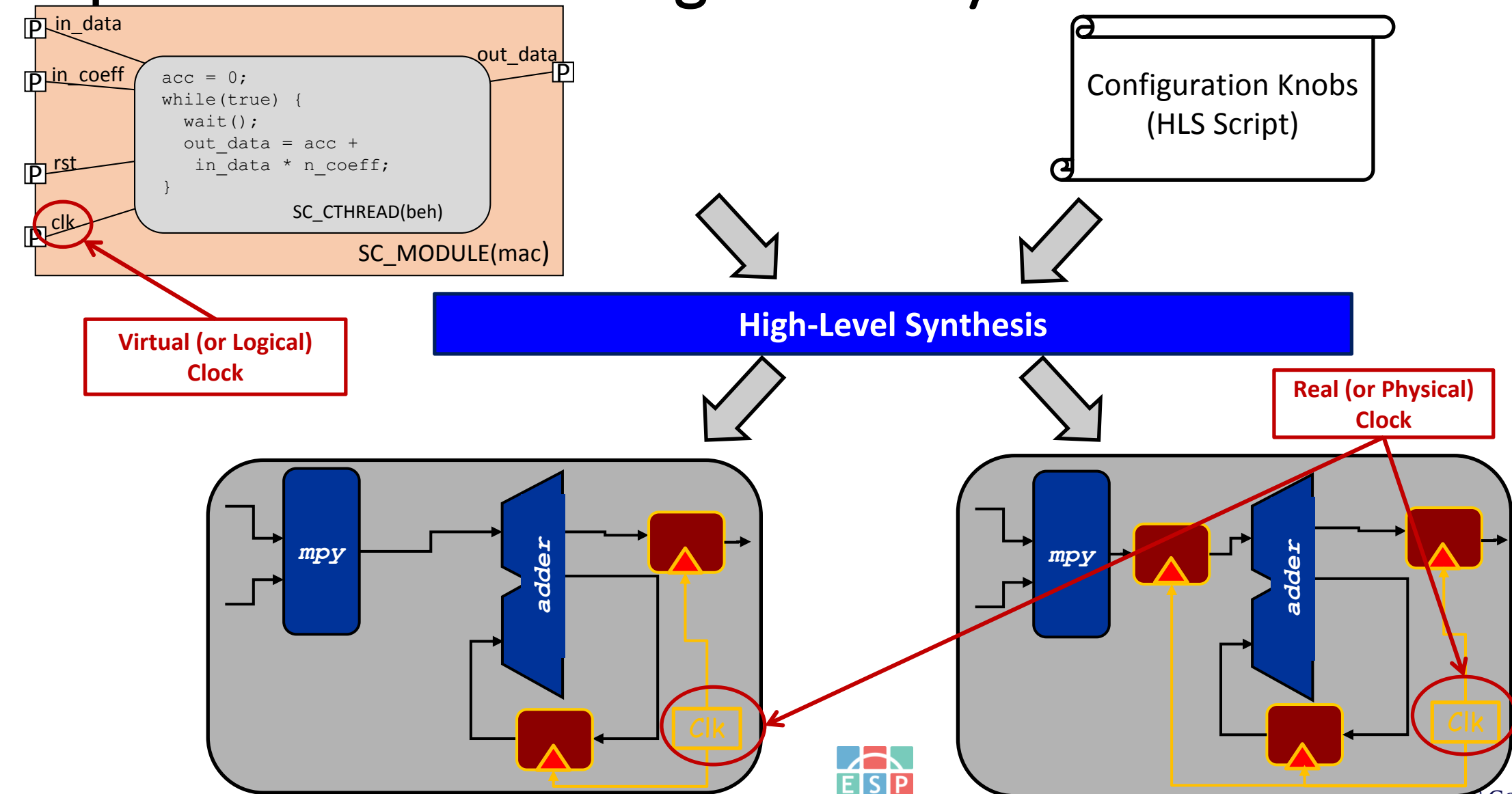
## Pareto Set Obtained with High-Level Synthesis (1GHz@1V, CMOS 32nm)



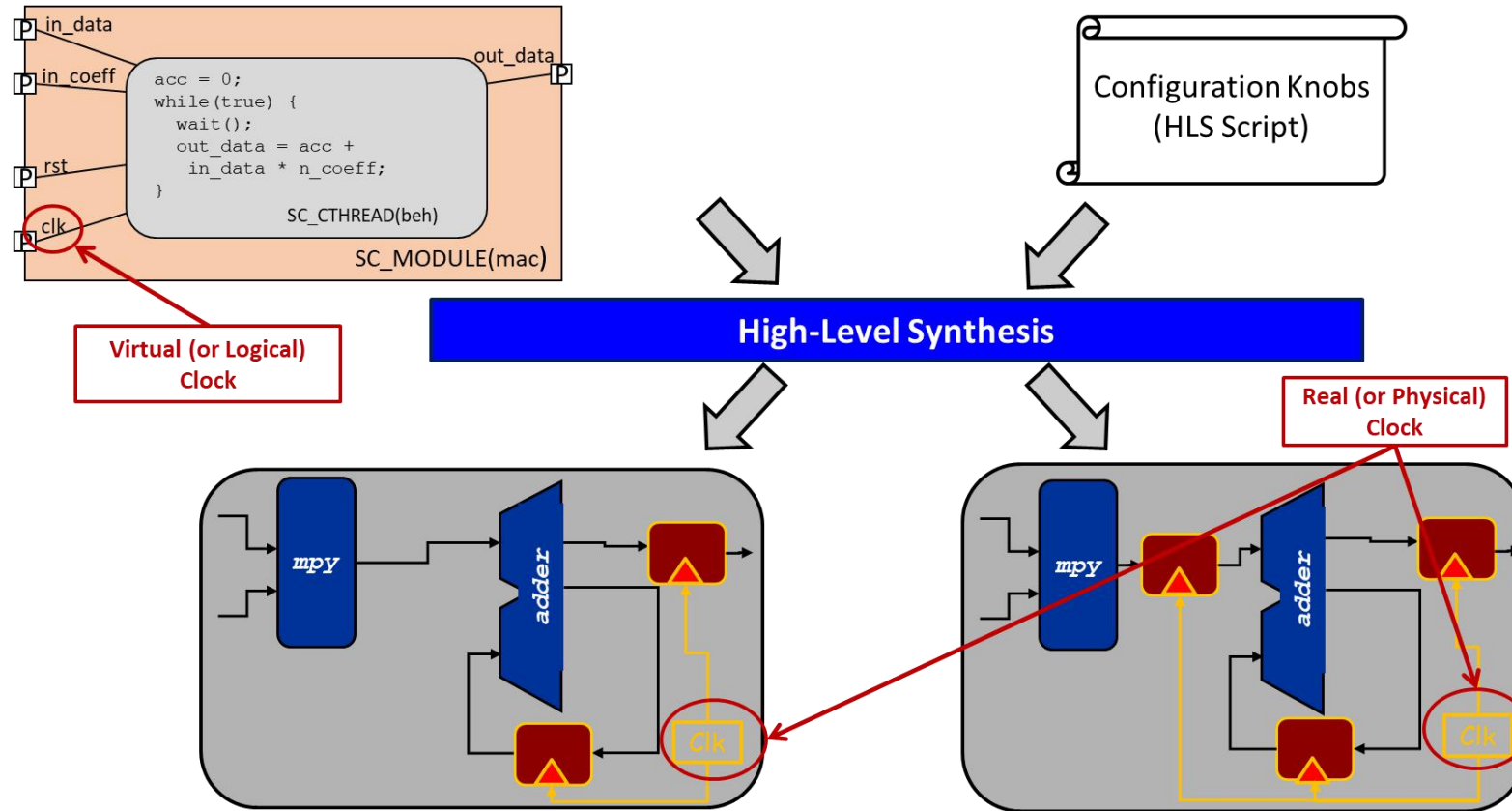
# From SystemC Specification to Alternative RTL Implementations via High-Level Synthesis



# From SystemC Specification to Alternative RTL Implementations via High-Level Synthesis



# From SystemC to RTL via HLS: Two Key Questions

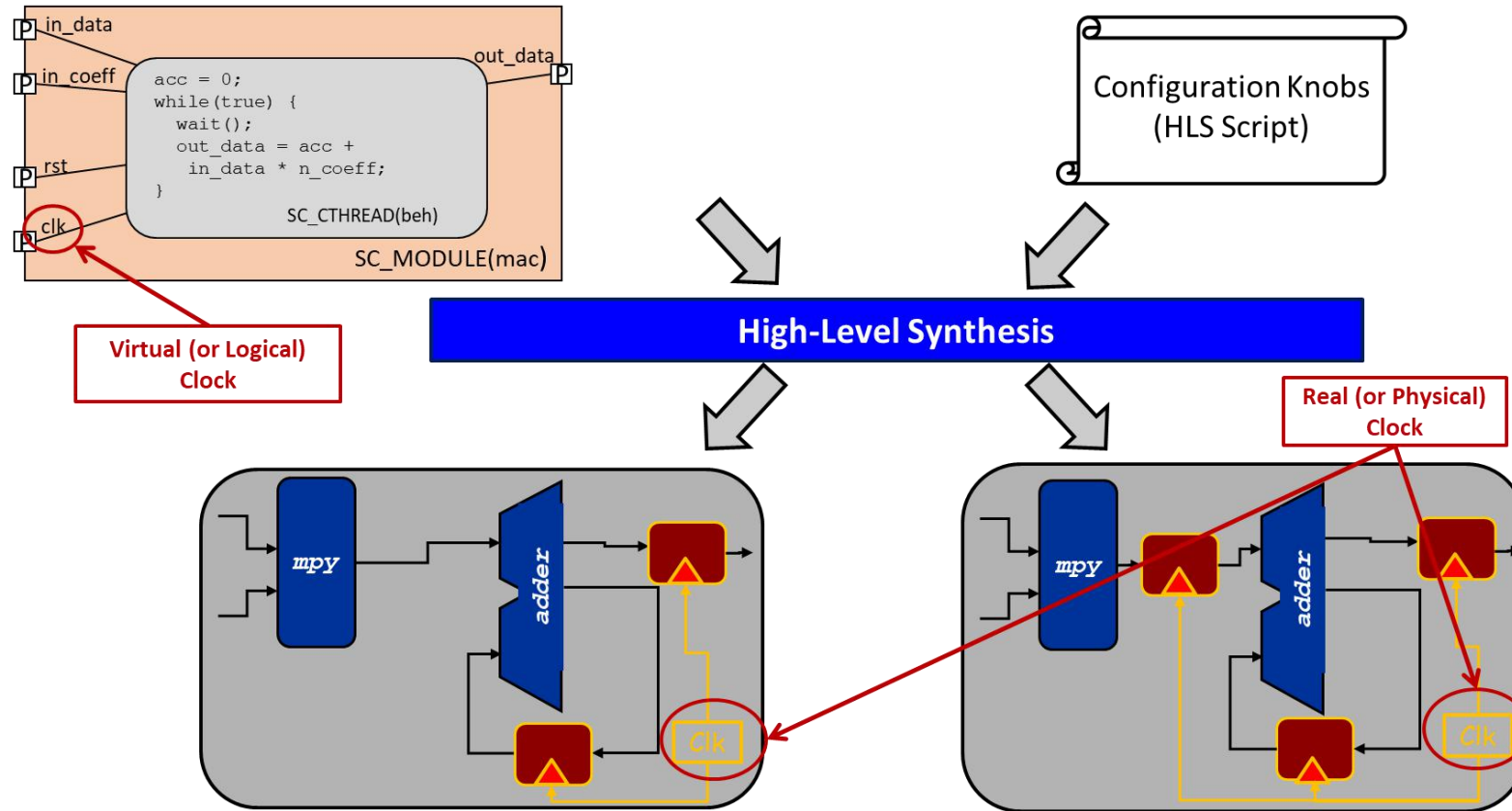


- In which sense each implementation is correct with respect to the original specification?
- How to find the best implementation?





# From SystemC to RTL via HLS: Optimality



- How to compare various synthesized implementations?

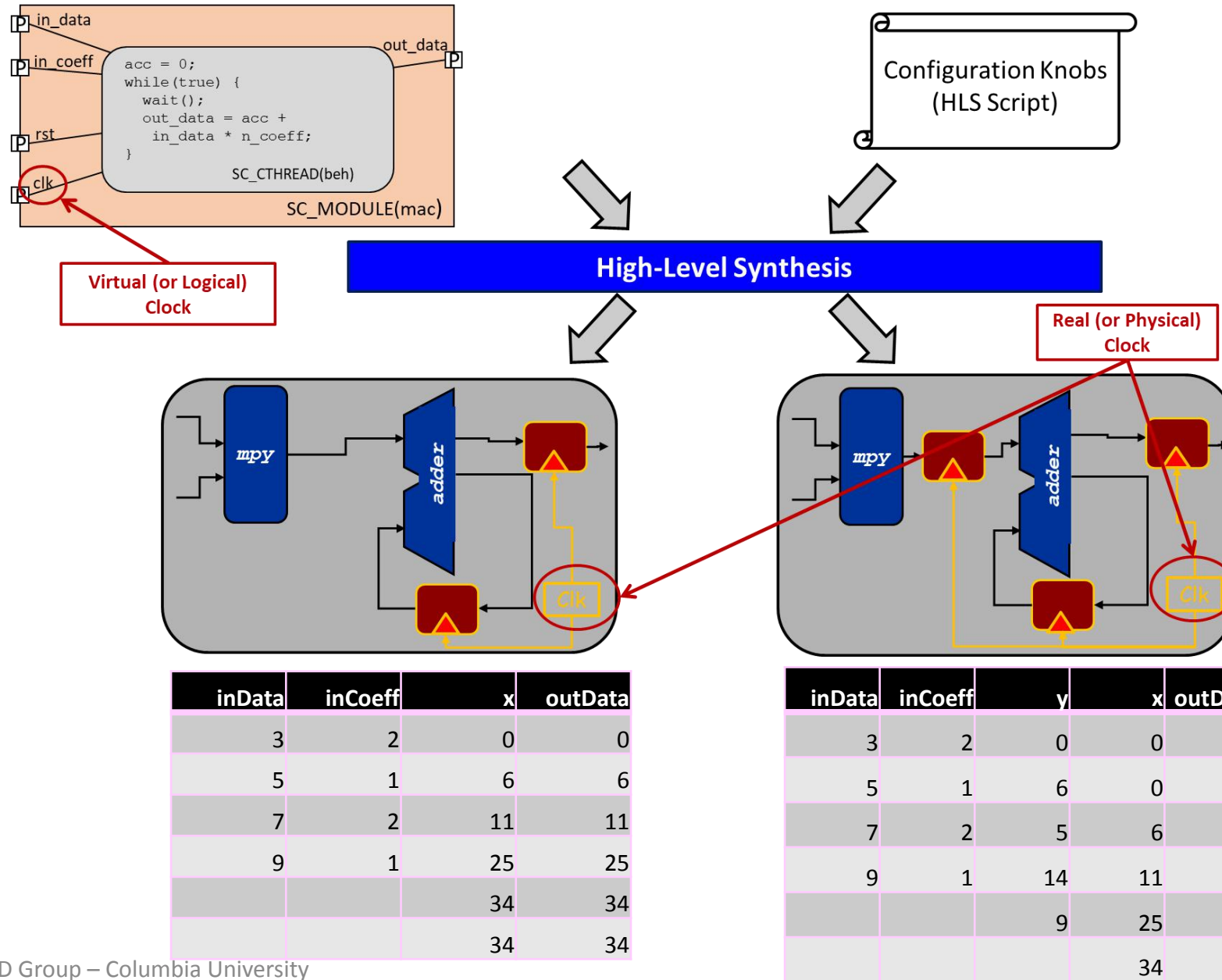
- in terms of cost
- in terms of performance

- This implementation has lower latency and lower area but also runs at lower (physical) clock frequency
- This implementation runs at higher (physical) clock frequency and offers higher data throughput but costs a bit more area

- Which implementation is better?



# From SystemC to RTL via HLS: Correctness



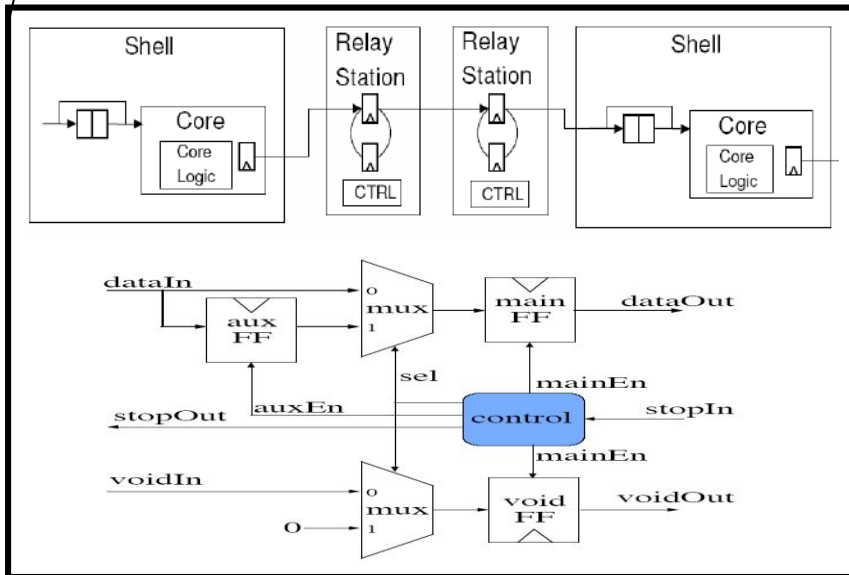
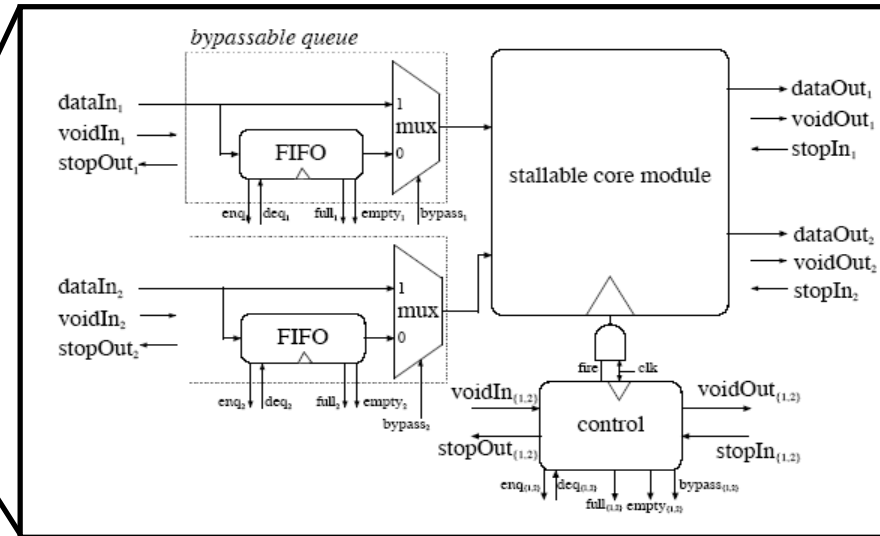
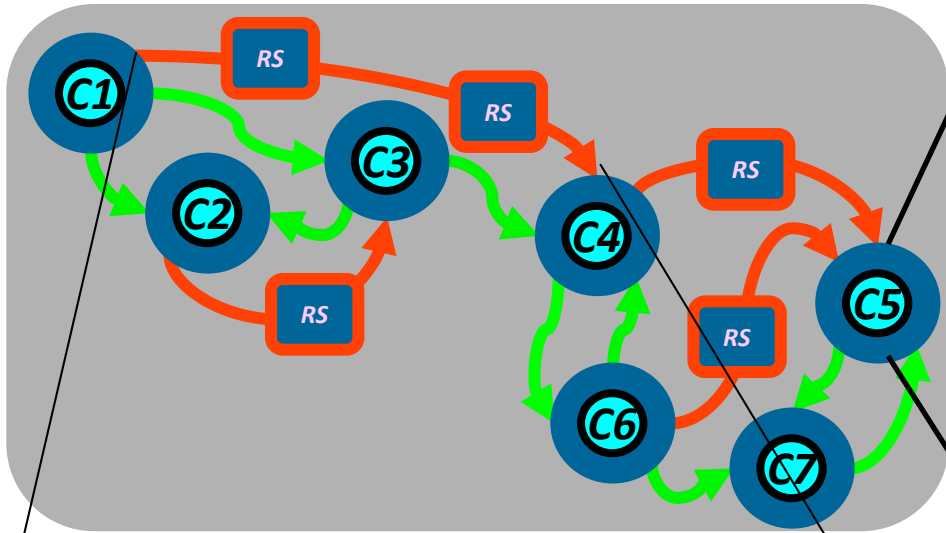
- Which notion of equivalence to use?

- between the synthesized implementation and the original specification
- among many alternative implementations?

- How to compare the I/O traces of the two implementations?

# Retrospective: Latency-Insensitive Design

[Carloni et al. '99]



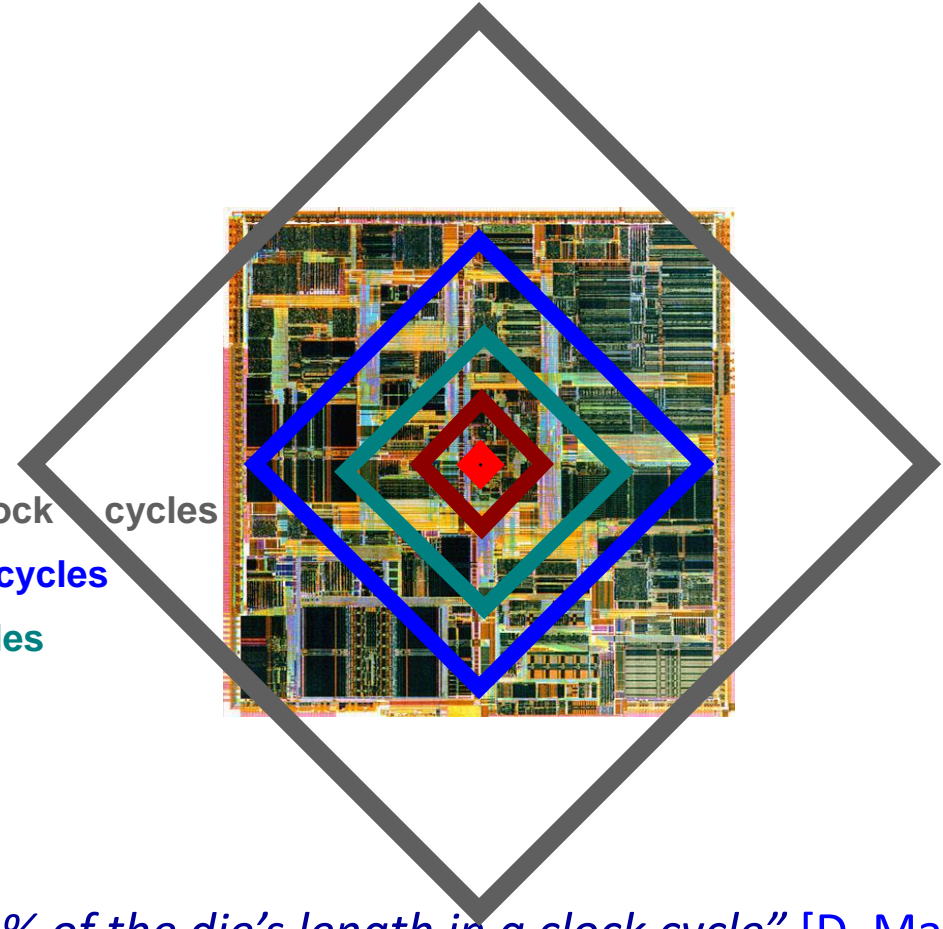
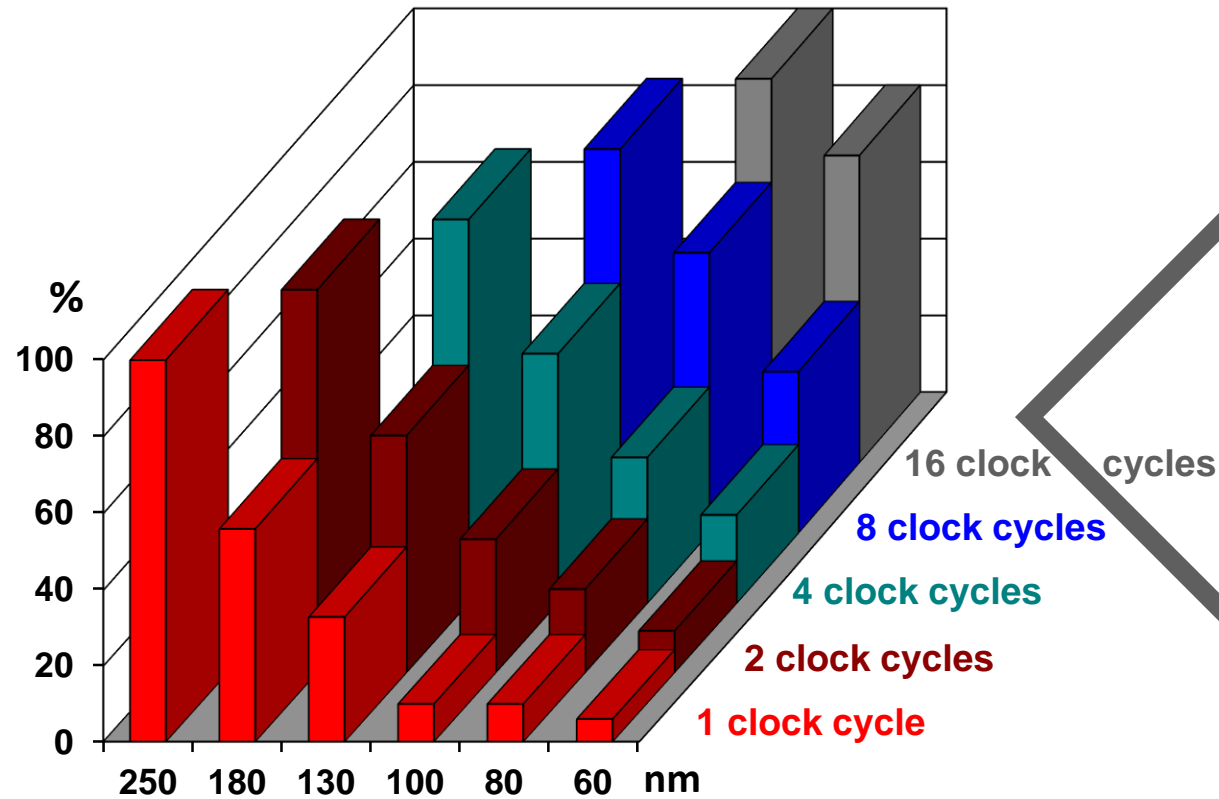
## Latency-Insensitive Design

- is the foundation for the *flexible synthesizable RTL representation*
- anticipates the separation of computation from communication that is proper of TLM with SystemC
  - through the introduction of the Protocols & Shell paradigm



# The Arrival of Nanometer Technologies in Mid Nineties

## Percentage of Reachable Die

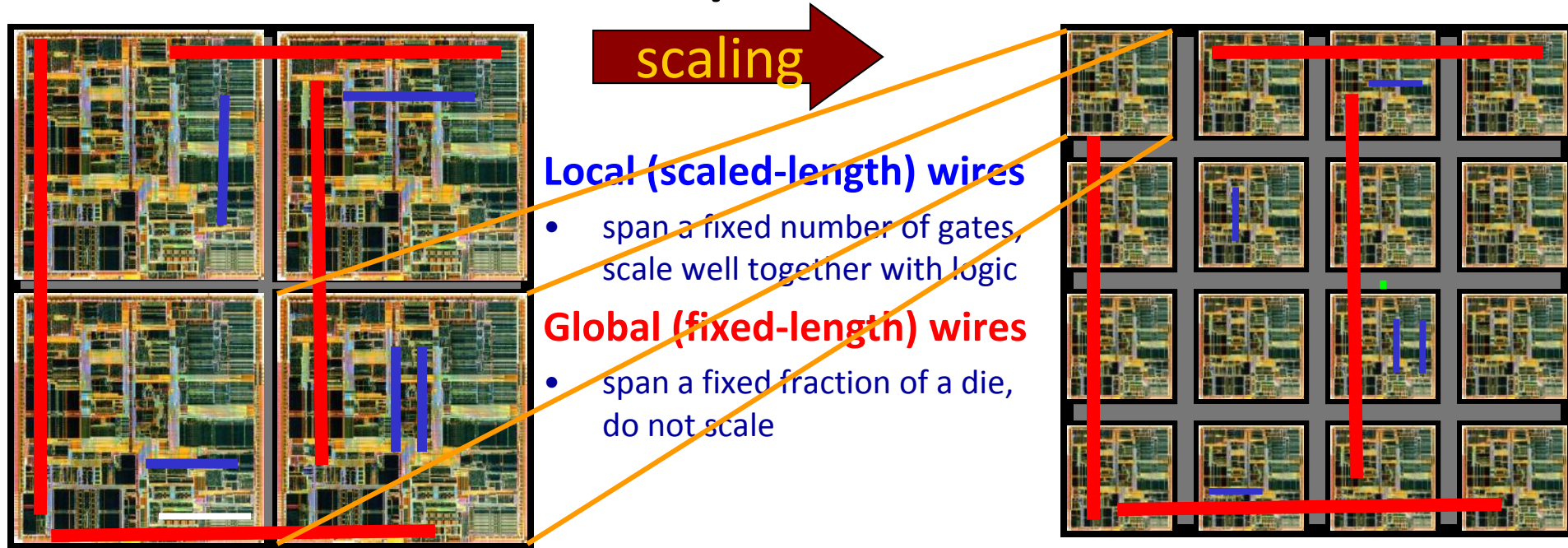


- “For a 60-nanometer process a signal can reach only 5% of the die’s length in a clock cycle” [D. Matzke, ‘97]
- Cause: Combination of higher clock frequencies and slower wires





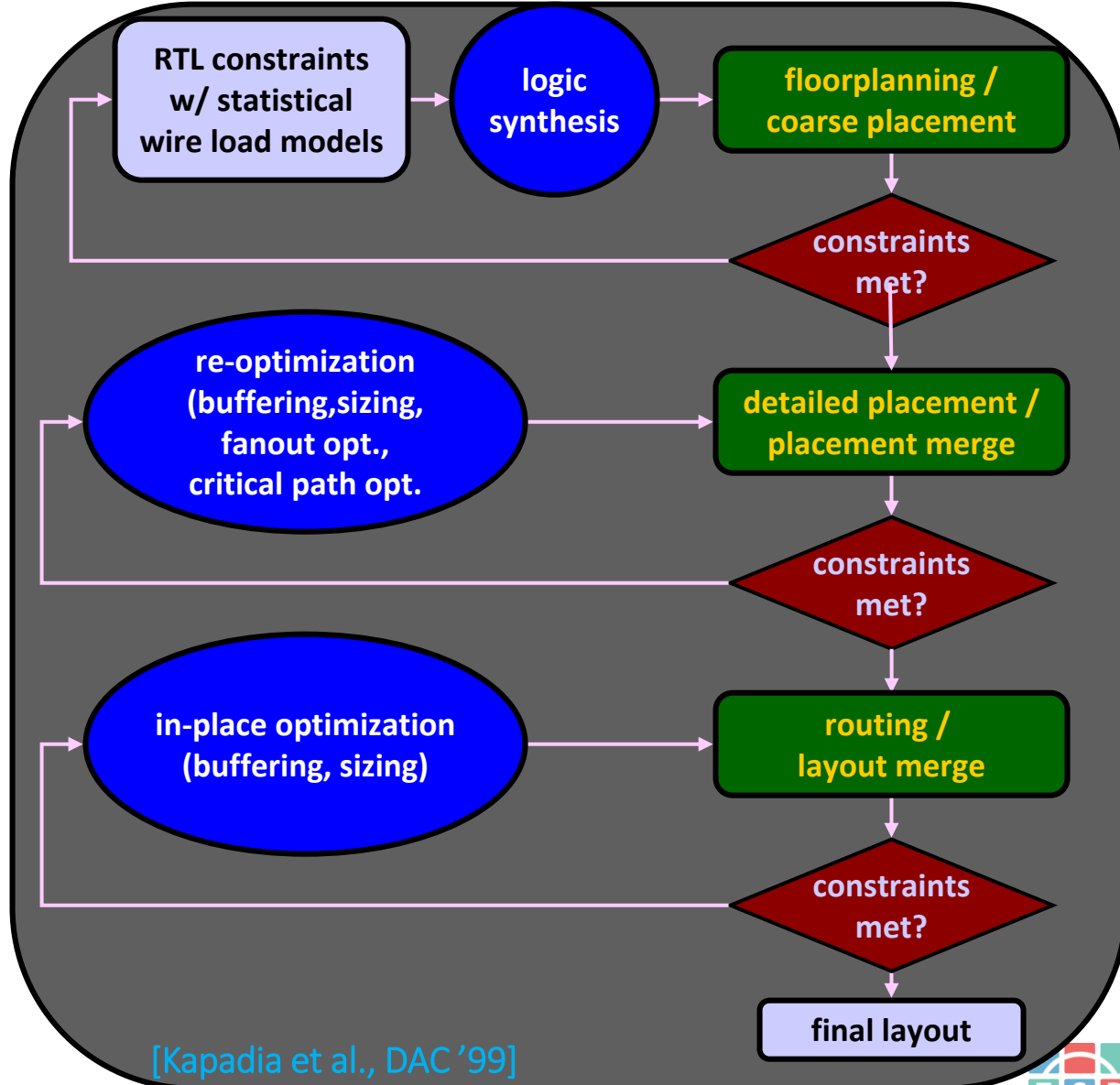
# Nanometer Technologies: Chips Become Distributed Systems



- Interconnect Latency
  - hard to estimate because affected by many phenomena
    - process variations, cross-talk, power-supply drop variations
  - breaks the synchronous assumption
    - that lies at the basis of design automation tool flows



# The Traditional Design Flow and the Timing Closure Problem

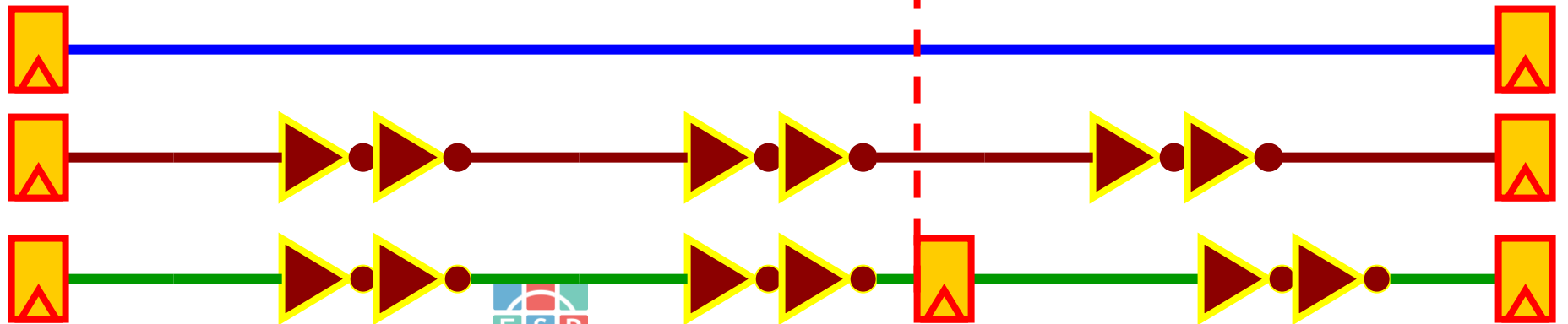
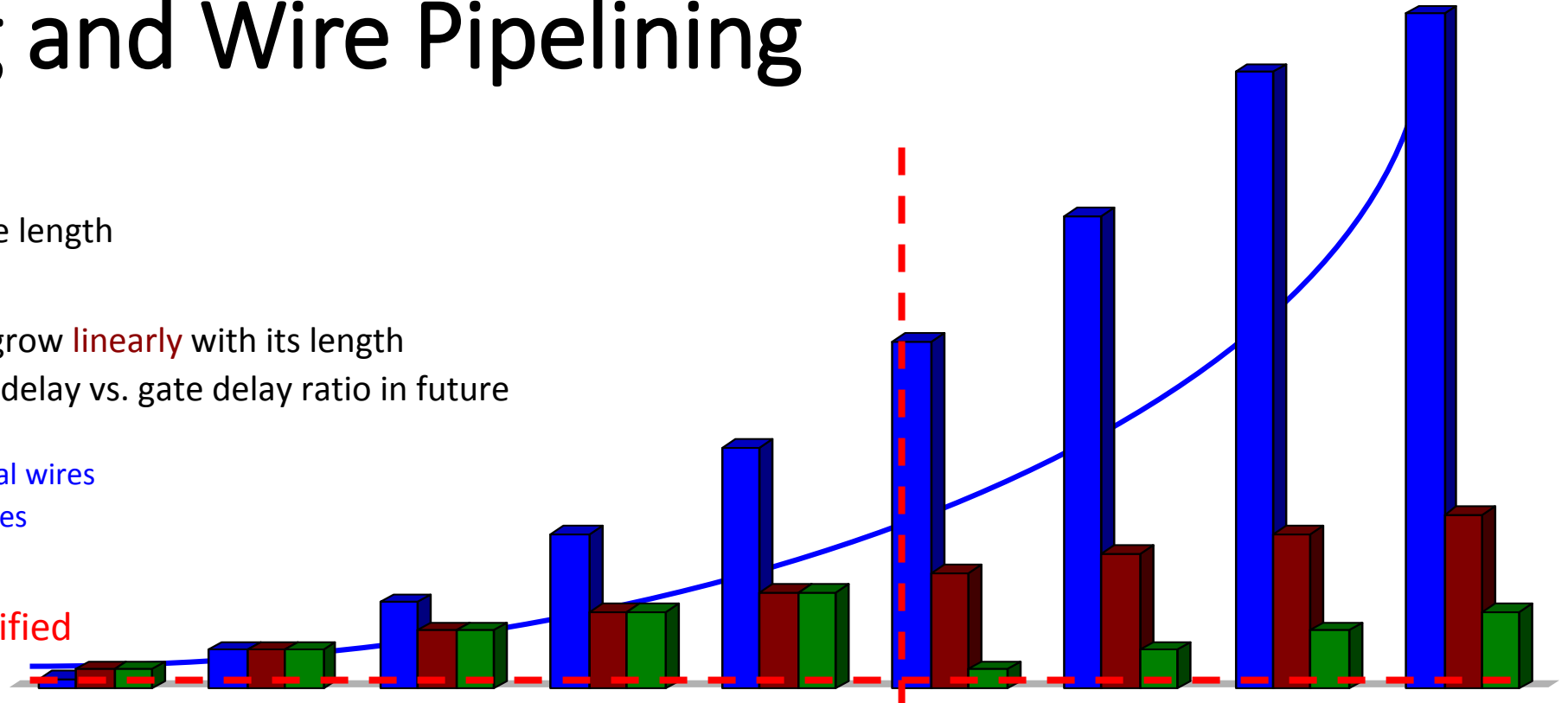


- Founded on the synchronous design methodology
  - longest combinational path (critical path) dictates the maximum operating frequency
  - operating frequency is often a design constraint
  - design exception: a path with delay larger than clock period
- **Many costly iterations between synthesis and layout because**
  - steps are performed independently
  - accurate estimations of global wire latencies are impractical
  - statistical delay models badly estimate post-layout wire load capacitance




# Wire Buffering and Wire Pipelining

- Wire Delay
  - grows quadratically with wire length
- Wire Buffering
  - if optimal makes wire delay grow linearly with its length
  - reduces the increase of wire delay vs. gate delay ratio in future process technologies
    - from 2000X to 40X for global wires
    - from 10X to 3X for local wires
- Wire Pipelining
  - is necessary to meet specified clock period



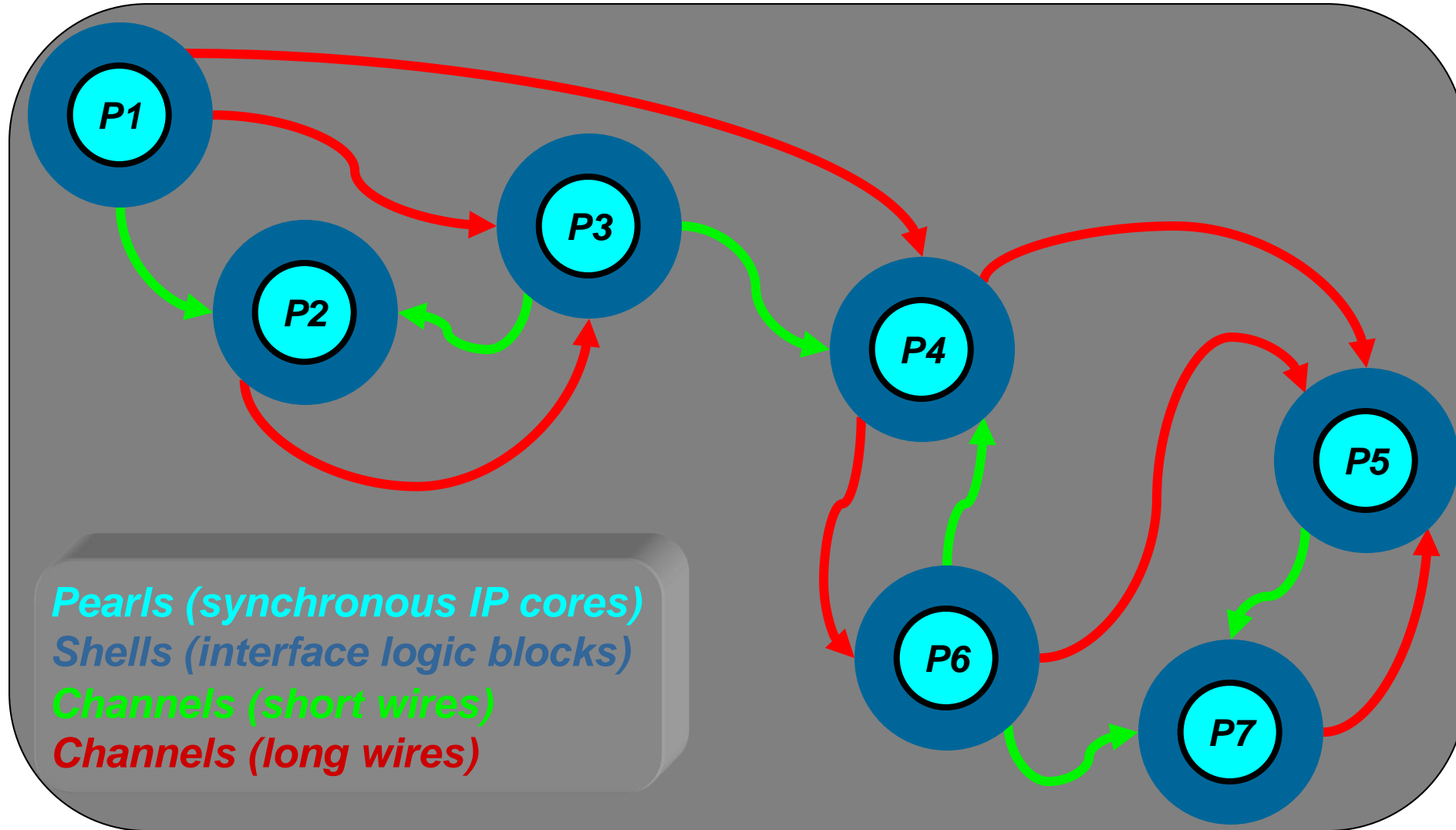
# Stateless Repeaters vs. Stateful Repeaters

- Both buffers and flip-flops are wire repeaters
  - regenerate the signals traveling on long wires
- **Stateful** repeaters 
  - storage elements, which carry a state
    - flip-flops, latches, registers, relay stations...
    - generally, the state must be initialized
- Inserting *stateful* repeaters impacts surrounding control logic
  - if the interface logic of two communicating modules assumed a certain latency, **then costly rework is necessary to account for additional pipeline stages**
  - necessary formal methods to enable automatic insertion

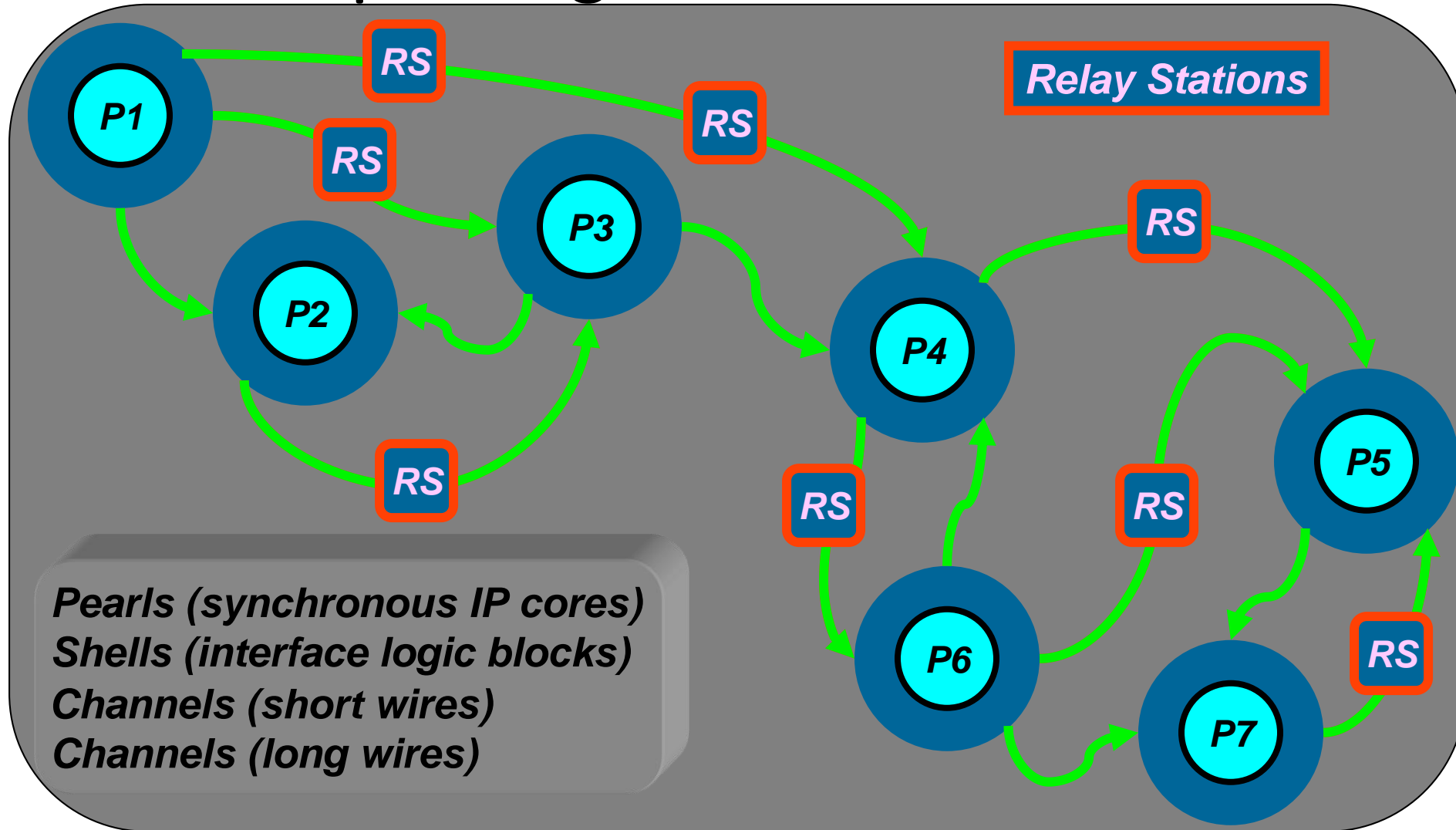




# Latency-Insensitive Design and the *Protocol & Shell* Paradigm [Carloni et al. '99]



# Correct-by-Construction Design Methodology Enables Automatic Wire Pipelining



**Relay Stations are sequential elements initialized with void data items**

# Compositionality & Theory of Latency-Insensitive Design [Carloni et al. '99]

- For patient processes the notion of latency equivalence is compositional

– Th.1:  $P1$  and  $P2$  patient  $\Rightarrow P1 \cap P2$  patient

– Th.2: for all patient  $P1, Q1, P2, Q2$   
 $P1 \equiv Q1$  and  $P2 \equiv Q2 \Rightarrow (P1 \cap P2) \equiv (Q1 \cap Q2)$

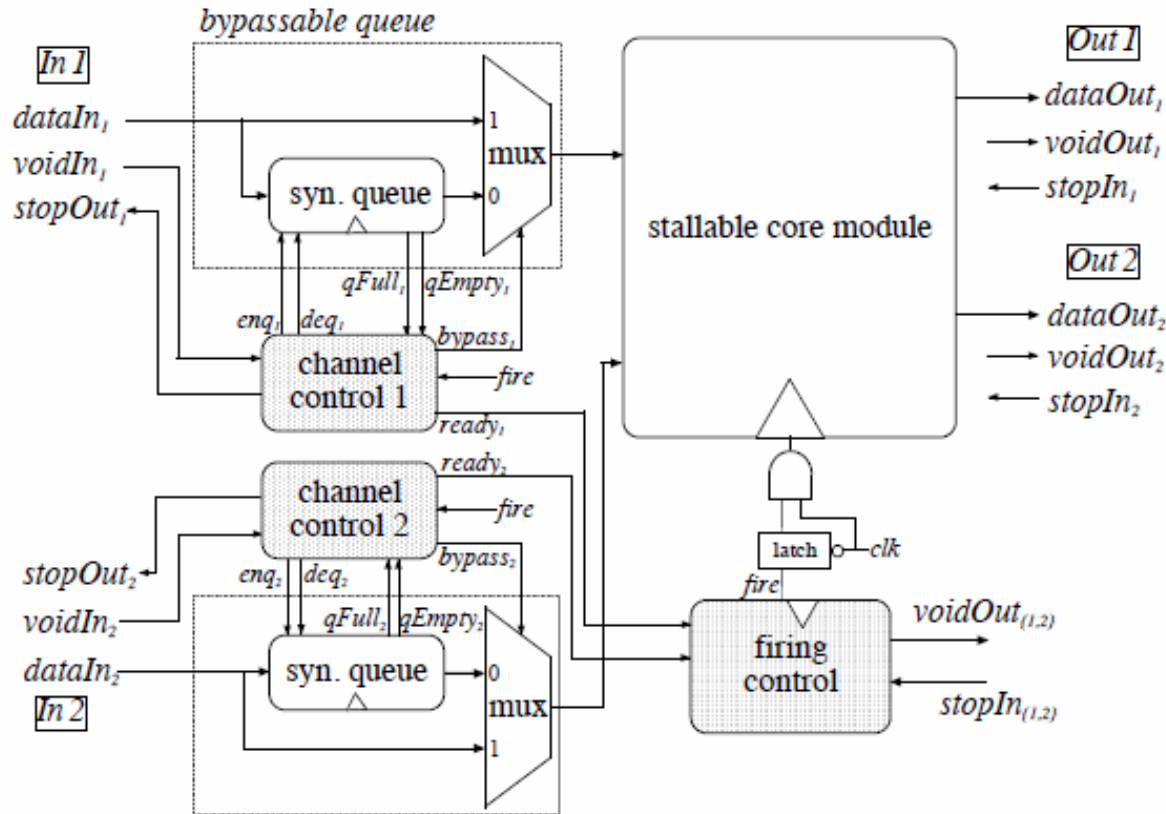
– Th.3: for all strict  $P1, P2$  and patient  $Q1, Q2$   
 $P1 \equiv Q1$  and  $P2 \equiv Q2 \Rightarrow (P1 \cap P2) \equiv (Q1 \cap Q2)$

- Major Theoretical Result

- if all processes in a strict system are replaced by corresponding patient processes **then the resulting system is latency equivalent to the original one**



# LID Building Blocks: Shell (with backpressure)



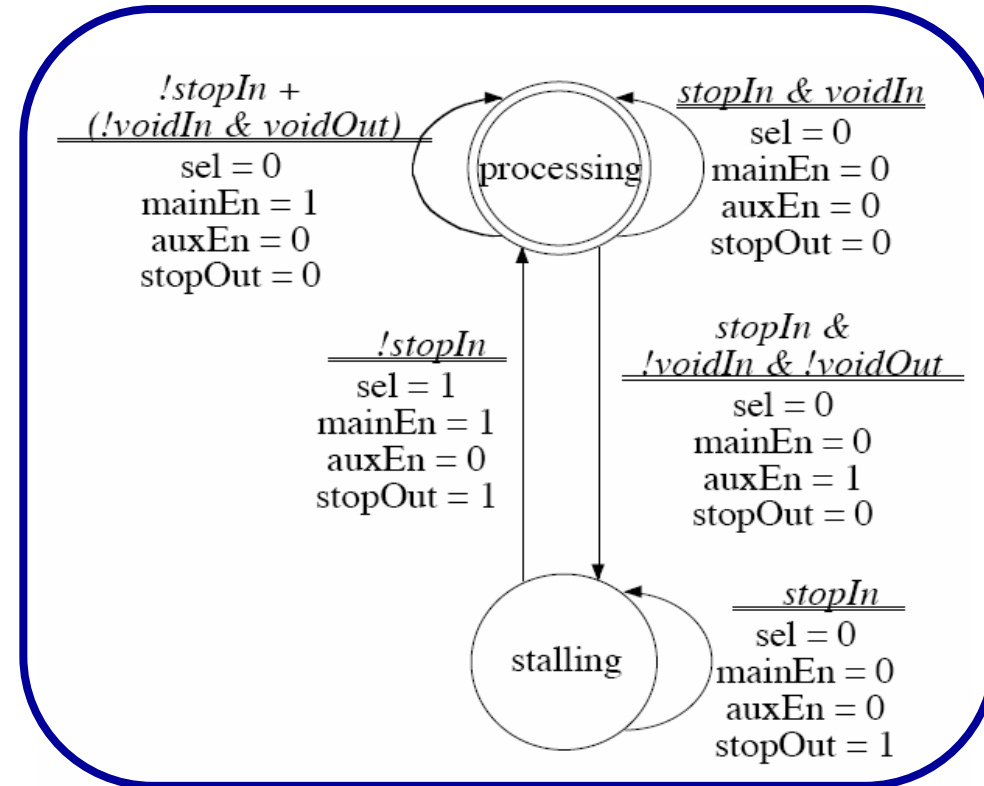
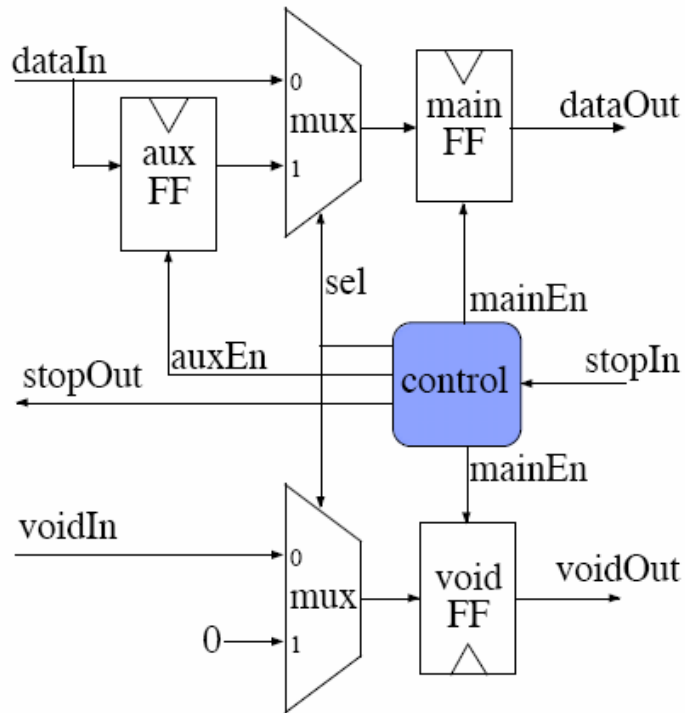
$$\begin{aligned}
 \text{fire} &= \bigwedge_{i \in \mathcal{I}} (\overline{\text{voidIn}_i} + \overline{\text{empty}_i}) \cdot \bigvee_{j \in \mathcal{O}} (\text{stopIn}_j \cdot \overline{\text{voidOut}_j}) \\
 \forall j \in \mathcal{O} \text{ voidOut}_j^+ &= \begin{cases} 0 & \text{if } \text{stopIn}_j \cdot \text{voidOut}_j \text{ is true} \\ \text{fire} & \text{otherwise} \end{cases} \\
 \forall i \in \mathcal{I} \text{ stopOut}_i &= \text{full}_i \\
 \forall i \in \mathcal{I} \text{ enq}_i &= \overline{\text{voidIn}_i} \cdot (\overline{\text{fire}} + \overline{\text{empty}_i}) \cdot \text{full}_i \\
 \forall i \in \mathcal{I} \text{ deq}_i &= \text{empty}_i \cdot \text{fire} \\
 \forall i \in \mathcal{I} \text{ bypass}_i &= \text{empty}_i
 \end{aligned}$$

- The theory of LID leaves open the possibility of developing various latency-insensitive protocols, each with a supporting implementation of the LID building blocks, i.e. shells and relay stations
- This is *a possible* implementation of a 2-input 2-output shell for a latency-insensitive protocol with *one-stop-to-stall* backpressure
  - the organization is general and can be easily scaled to an any I/O number
  - all output signals are clocked at the output of edge-triggered flip-flops
  - the minimum forward latency of the bypassable queue is zero





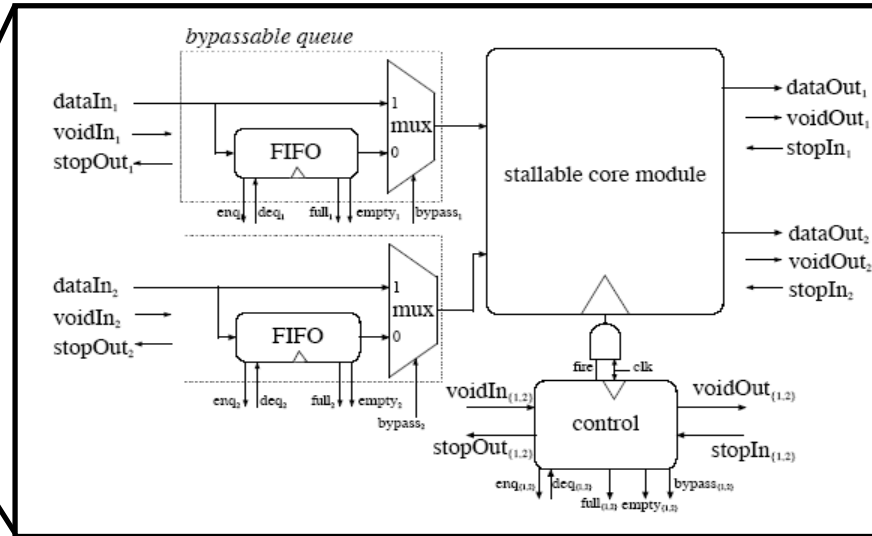
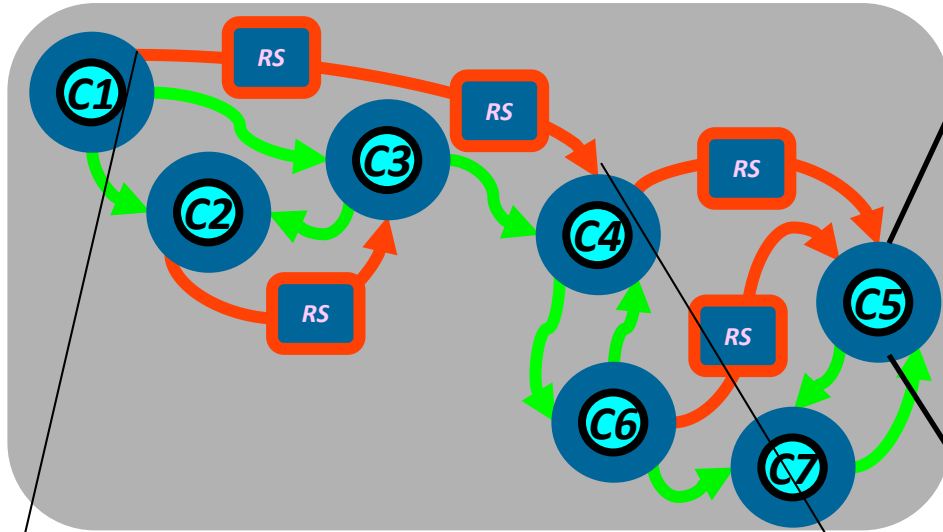
# LID Building Blocks: Relay Station



- A relay station is a clocked (stateful) buffer with
  - twofold storage capacity
  - simple control flow logic implemented as a 2-state Mealy FSM
    - Note that the value of the *stopOut* bit depends only on the current state of the controller, and thus no combinational path exists between *stopIn* and *stopOut*



# Benefits of the Protocols & Shells Paradigm

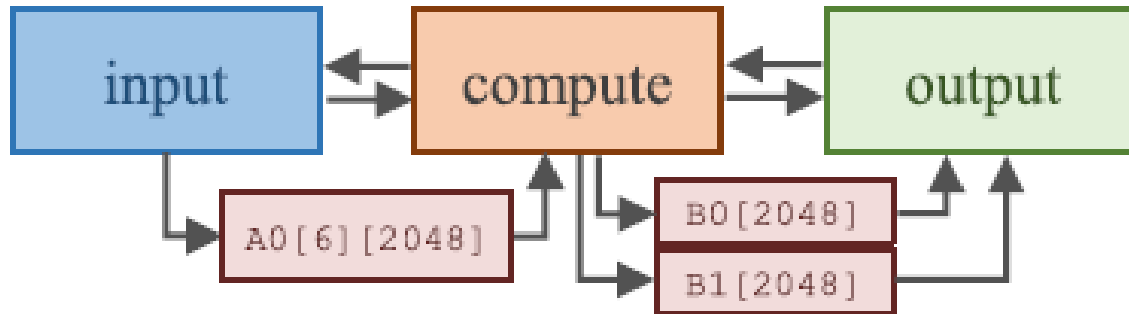


## The Protocol & Shells Paradigm

- preserves **modularity** of synchronous assumption in distributed environment
- guarantees **scalability** of global property by construction and through synthesis
- simplifies *integrated design & validation* by decoupling communication and computation, thus enabling **reusability**
- adds **flexibility** up to late stages of the design process



# Example: Combining LID and HLS in the Design of the Debayer Accelerator



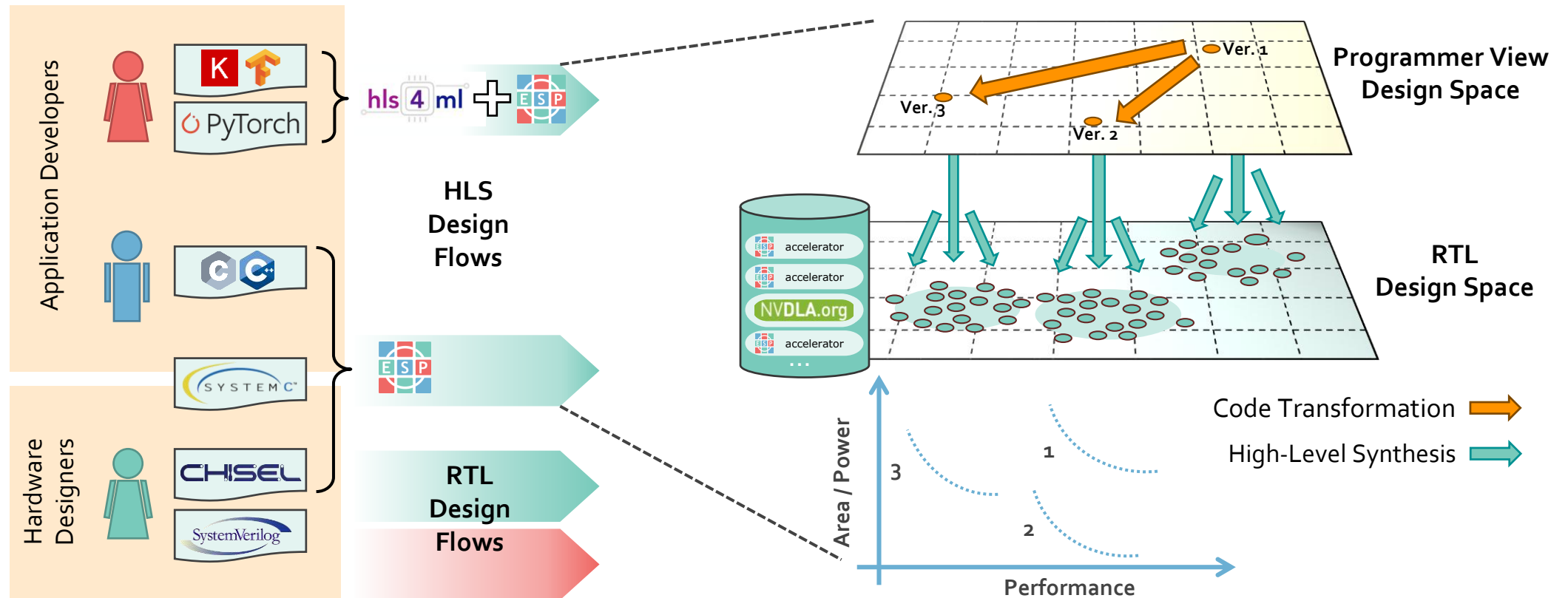
- The combination of the ESP interface and the latency-insensitive protocol enable a broad HLS-supported design-space exploration
- For example, for the compute process
  - Implementation E is obtained by unrolling loop L3 for 2 iterations, which requires 2 concurrent memory-read operations
  - Implementation F is obtained by unrolling L3 for 4 iterations to maximize performance at the cost of more area, but with only 2 memory-read interfaces; this creates a bottleneck because the 4 memory operations cannot be all scheduled in the same clock cycle
  - Implementation G, which Pareto-dominates implementation F, is obtained by unrolling L3 for 4 iterations and having 4 memory-read interfaces to allow the 4 memory-read operations to execute concurrently

[C. Pilato, P. Mantovani, G. Di Guglielmo, and L. P. Carloni, System-Level Optimization of Accelerator Local Memory for Heterogeneous Systems-on-Chip, TCAD '17]

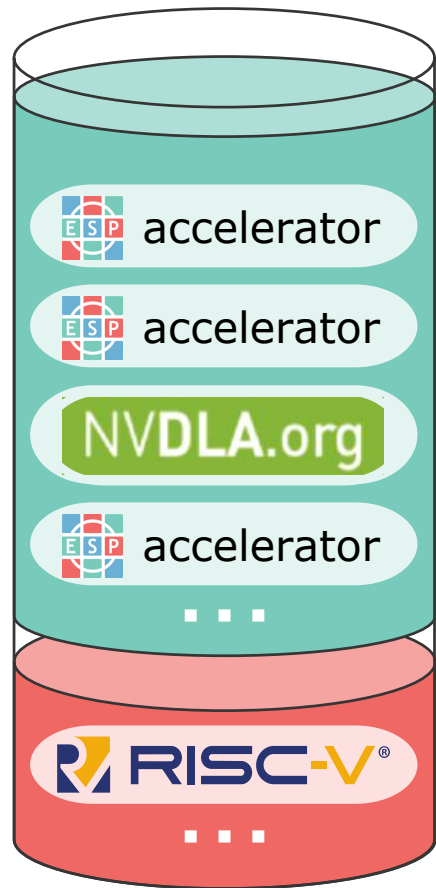


# ESP Accelerator Flow

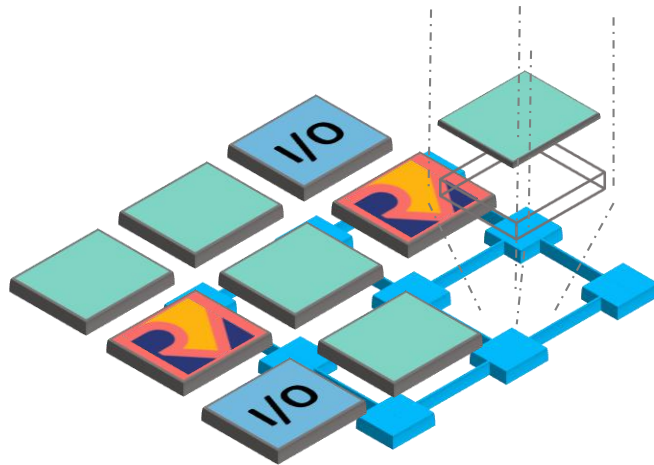
Developers focus on the **high-level specification, decoupled** from memory access, system communication, hardware/software interface



# ESP Interactive SoC Flow



## SoC Integration



ESP SoC Generator

**General SoC configuration:**  
virtexup  
ETH FPnew  
No JTAG  
Eth (192.168.1.2)  
Use SGMII  
No SVGA  
With synchronizers

**Data transfers:**  
☐ Bigphysical area  
☒ Scatter/Gather

**Cache Configuration:**  
Cache En.: ☐  
L2 SETS: 512  
L2 WAYS: 4  
LLC SETS: 1024  
LLC WAYS: 16  
ACC L2 SETS: 512  
ACC L2 WAYS: 4

**CPU Architecture:**  
Core: ariane

**NoC configuration**  
Rows: 2 Cols: 2  
Config

☐ Monitor DDR bandwidth  
☐ Monitor memory access  
☐ Monitor injection rate  
☐ Monitor router ports  
☐ Monitor accelerator status  
☐ Monitor L2 Hit/Miss  
☐ Monitor LLC Hit/Miss  
☐ Monitor DVFS

Num CPUs: 1  
Num memory controllers: 1  
Num I/O tiles: 1  
Num accelerators: 0

Num CLK regions: 1  
Num CLKBUF: 0  
VF points: 4

**NoC Tile Configuration**

(0,0)	(0,1)
mem	cpu
<input type="checkbox"/> Has L2 Clk Reg: 0 <input type="checkbox"/> Has PLL <input type="checkbox"/> CLK BUF	<input type="checkbox"/> Has L2 Clk Reg: 0 <input type="checkbox"/> Has PLL <input type="checkbox"/> CLK BUF
(1,0)	(1,1)
empty	IO
<input type="checkbox"/> Has L2 Clk Reg: 0 <input type="checkbox"/> Has PLL <input type="checkbox"/> CLK BUF	<input type="checkbox"/> Has L2 Clk Reg: 0 <input type="checkbox"/> Has PLL <input type="checkbox"/> CLK BUF

Generate SoC config





# “So, Why Most SoCs are Still Designed Starting from Manually-Written RTL Code?”

- **Difficult to pinpoint a single cause...**
  - Natural inertia of applying best practices
  - Organization of engineering divisions are based on well-established sign-off points of traditional CAD flows
  - Limitations of existing SLD tools (for HLS, verification, virtual platforms..)
  - Shortage of engineers trained to work at the SLD level of abstraction
- **Arguably, a chicken-and-egg problem**
  - the lack of bigger investments in developing SLD methodologies and tools is due to a lack of demand from engineers; conversely, the lack of this demand is due to the shortcomings of current SLD methodologies and tools
  - Academia should take the lead in breaking this vicious cycle



# CSEE-4868: System-on-Chip Platform

- *Foundation course on the programming, design, and validation of SoCs with emphasis on high-performance embedded applications*
- Offered at Columbia since 2011, moved to upper-level curriculum in Fall 2016
  - required course for CE BS program, elective for MS programs in CS and EE
- **Course Goals**
  - mastering the HW and SW aspects of integrating heterogeneous components into a complete system
  - designing new components that are reusable across different systems, product generations, and implementation platforms
  - evaluating designs in a multi-objective optimization space

**[L. P. Carloni et al. Teaching Heterogeneous Computing with System-Level Design Methods, WCAE 2019 ]**



# CSEE-4868 – Course Structure

- The course consists of **two main tracks** that run in parallel throughout the semester

## 1. Theory Track:

- Lectures on principles of system-level design, models of computation, latency-insensitive design, virtual prototyping, design-space exploration, HW/SW co-design, SoC architectures
  - Illustrated with case studies of recent SoCs from industry and academia

## 2. Practice Track

- Lectures on SystemC and transaction-level modeling, SW application and driver programming with virtual platforms, and HW accelerator design with HLS tools
  - extensive use of commercial tools (e.g. for HLS) and in-house tools (e.g. virtual platform, memory optimization)

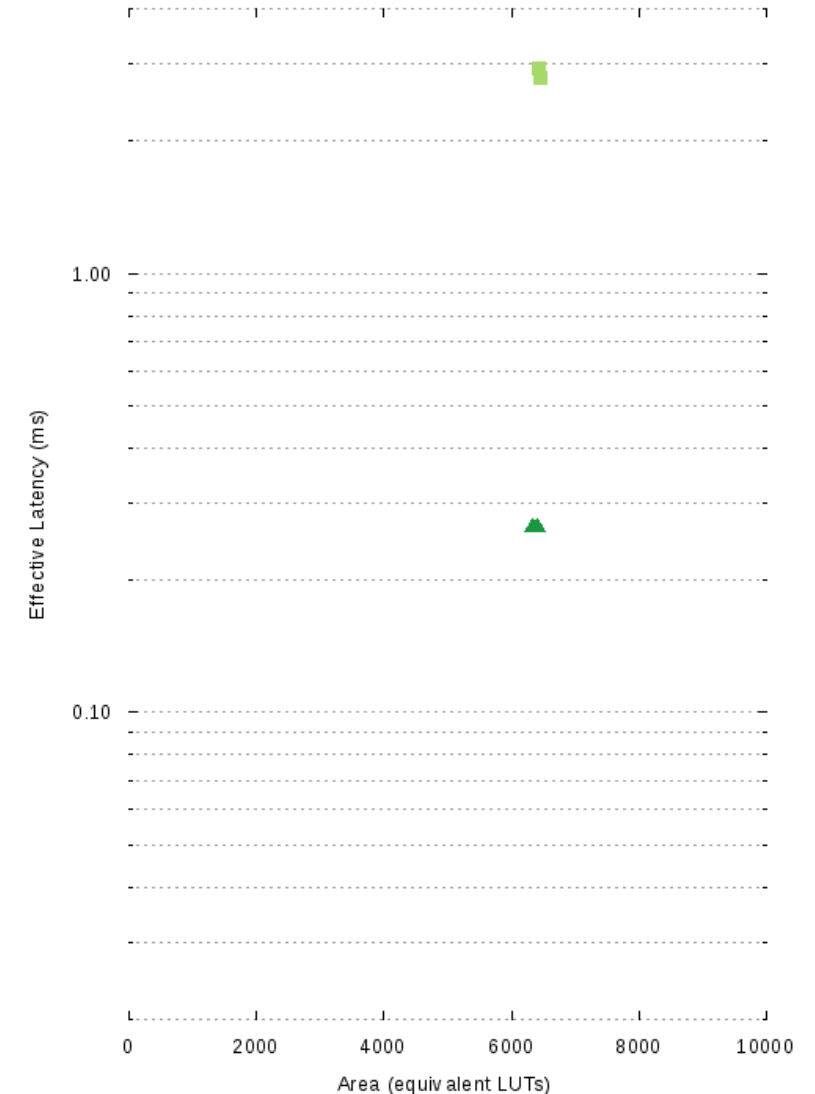


# Teaching System-on-Chip Platforms at Columbia:

## The Fall-2015 Course Project in Numbers

- At Columbia we developed the course '*CSEE-6868 System-on-Chip Platforms*' based on the ESP Design Methodology
- The Fall-2015 Project by Numbers
  - 21 student teams competed in designing a hardware accelerator for the WAMI Gradient kernel during a 1-month period
  - 661: Number of improved designs across all teams
  - 31.5: Average number of improved designs per team
  - 1.5: Average number of improved designs committed each day per team
  - 99: Total number of changes of the Pareto curve over the project period
  - 11: Final number of Pareto-optimal designs
  - 26X: Performance range of final Pareto curve
  - 10X: Area range of final Pareto curve

CSEE E6868 - Gradient - Pareto Curve Dec 02

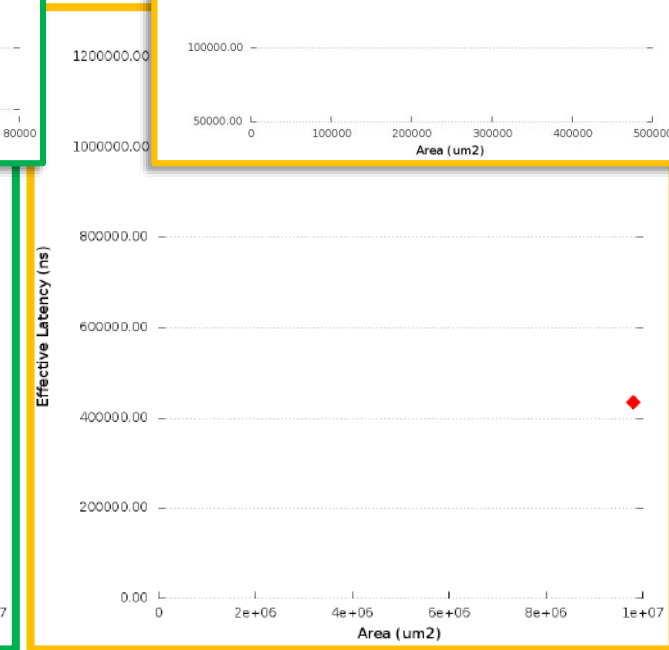
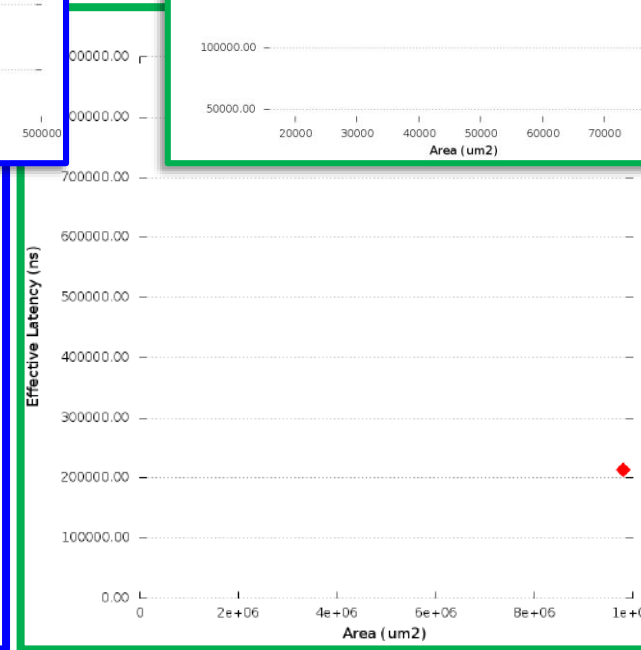
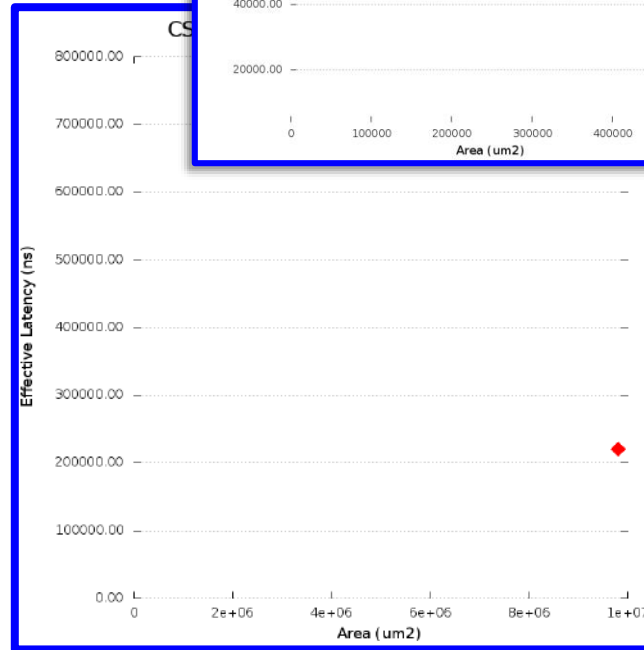
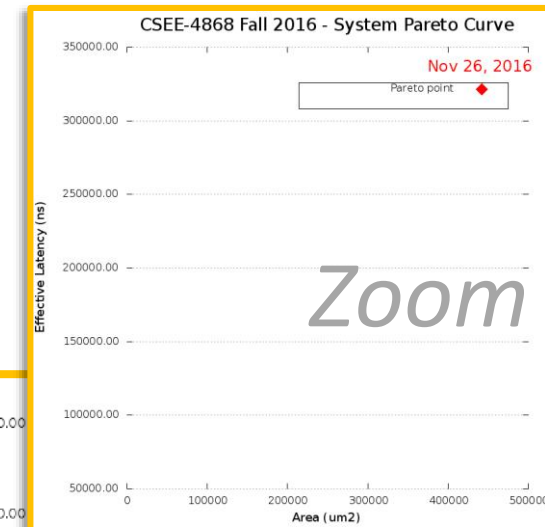
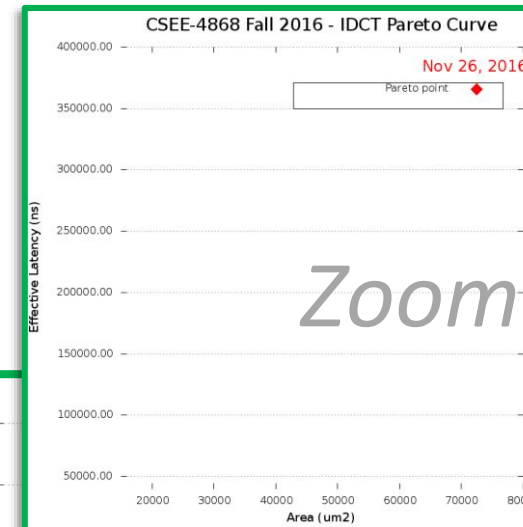
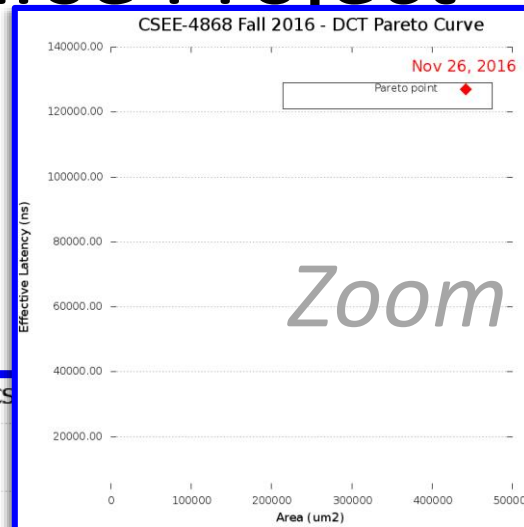


# Scaling Up the Design Complexity: The Fall-2016 Course Project

- **Fall-2016 New Features**
  - **Cloud-based** project environment
  - Introduction of **IP reuse** and compositional system-level design

- **The Fall-2016 Project by Numbers**

- 15 student teams competed in designing a system combining DCT and IDCT accelerators
- 302: Number of improved module designs across all teams
- 20.5: Average number of improved module designs per team
- 12.1: Average number of improved module designs per day
- 20: Total number of days when the Pareto curve of the system changed
- 20: Final number of Pareto-optimal designs
- 24X: System performance range
- 4X: System area range

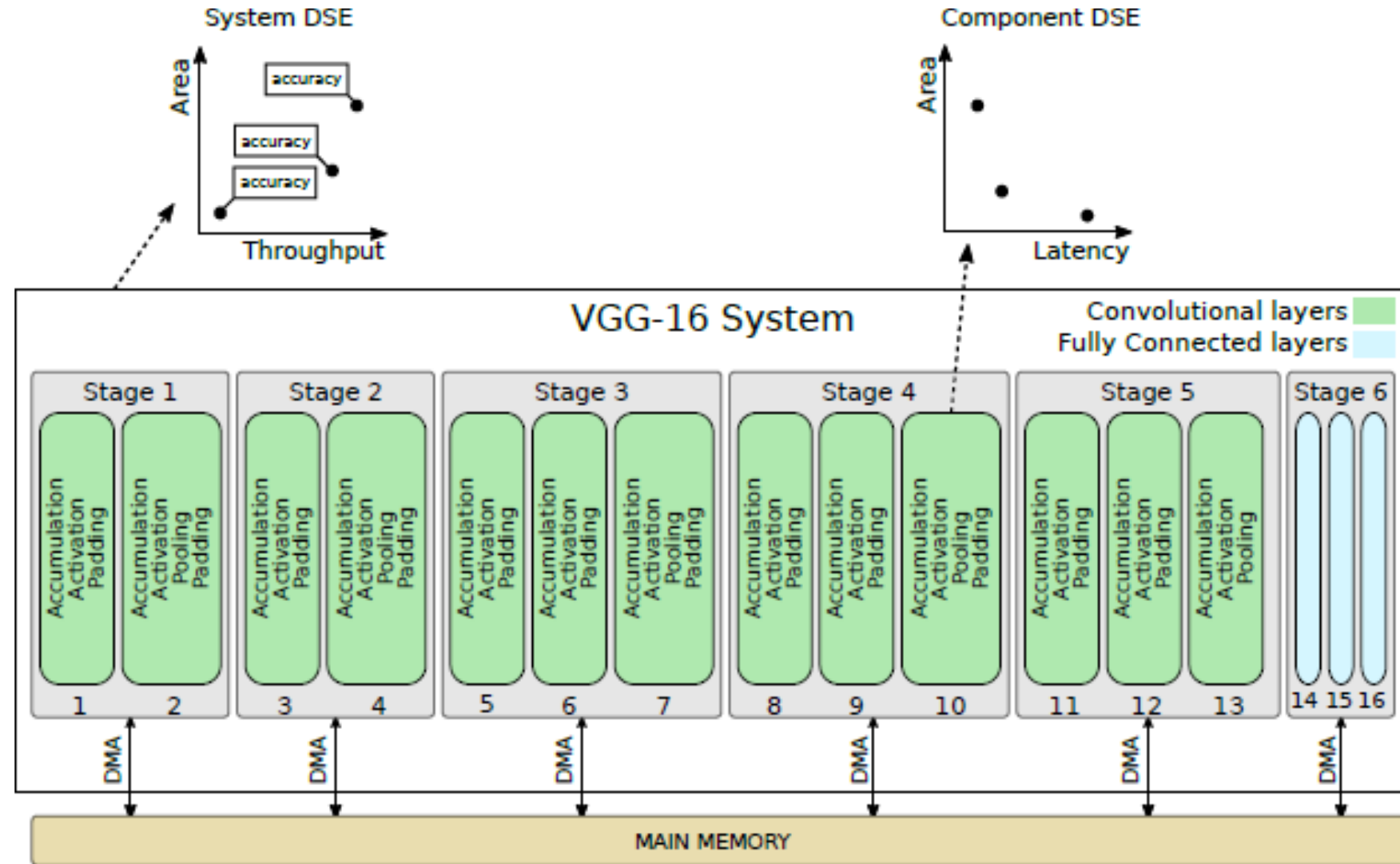




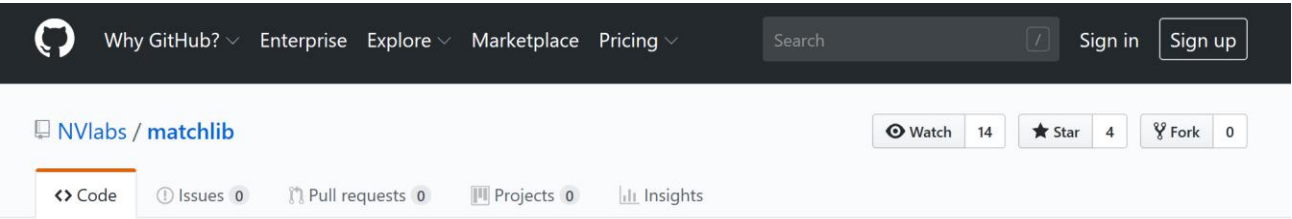
# Keep Scaling Up the Design Complexity: The Fall-2017 Course Project

- *Competitive and collaborative system-level design-space exploration of a CNN accelerator*

- partitions of the set of student teams compete on the reusable design of an given CNN stage
- all teams combine their stage design with the designs they “license” for the other stage to compete for the design of the overall CNN



# NVIDIA MatchLib



## MatchLib

build passing

MatchLib is a SystemC/C++ library of commonly-used hardware functions and components that can be synthesized by most commercially-available HLS tools into RTL.

Doxygen-generated documentation can be found [here](#). Additional documentation on the Connections latency-insensitive channel implementation can be found in the [Connections Guide](#).

## Getting Started

### Tool versions

MatchLib is regressed against the following tool/dependency versions:

- gcc - 4.9.3
- systemc - 2.3.1
- boost - 1.55.0
- doxygen - 1.8.11
- make - 3.82
- catapult - 10.3
- vcs - 2017.03-SP2-11
- verdi - 2017.12-SP2-2

## INVITED: A Modular Digital VLSI Flow for High-Productivity SoC Design

Brucek Khailany<sup>†</sup>, Evgeni Krimer<sup>†</sup>, Rangharajan Venkatesan<sup>†</sup>, Jason Clemons<sup>†</sup>, Joel S. Emer<sup>†◇</sup>, Matthew Fojtik<sup>†</sup>, Alicia Klinefelter<sup>†</sup>, Michael Pellauer<sup>†</sup>, Nathaniel Pinckney<sup>†</sup>, Yakun Sophia Shao<sup>†</sup>, Shreesha Srinath<sup>‡</sup>, Christopher Torng<sup>‡</sup>, Sam (Likun) Xi<sup>\*</sup>, Yanqing Zhang<sup>†</sup>, Brian Zimmer<sup>†</sup>

<sup>†</sup>NVIDIA, <sup>‡</sup>Cornell University, <sup>\*</sup>Harvard University, <sup>◇</sup>Massachusetts Institute of Technology



## Connections Guide

### Latency Insensitive Channel Library

## MATCHLIB, NVIDIA

MatchLib communications methodology is based on high-level synthesis (HLS) on the latency-insensitive design (LID) paradigm. Systems are fully designed in synthesizable SystemC and C++, and components are connected through synthesizable SystemC latency-insensitive (LI) channels. The approach is based on a library and API of latency-insensitive channels called Connections, which is the subject of this guide.

### 1 INTRODUCTION

MatchLib's *Connections* is a library and API of latency-insensitive channels. It was presented for the first time at DAC 2018 as part of a new modular digital VLSI methodology [Khailany et al. 2018]. To know the motivation behind *Connections* refer to section 2.3 of [Khailany et al. 2018].

All components of this library are HLS-able and they are designed to be synthesized with Mentor Catapult. Table 1 shows an overview of the most relevant components and API in *Connections*.

Table 1. API of Connections, reflecting unified terminals (ports) and types of channels

Port	Functions
In<T>	Pop(), PopNB()
Out<T>	Push(), PushNB()
InBuffered<T>	Pop(), PopNB(), Empty(), Peek()
OutBuffered<T>	Push(), PushNB(), Full()

Channel	Description
Combinational<T>	Combinationally connects ports
Bypass<T>	Enables DEQ when empty
Pipeline<T>	Enables ENQ when full
Buffer<T>	FIFO channel
OutNetwork<T>, InNetwork<T>	Network channels: packetizer and de-packetizer

### 2 PORTS

A module's latency-insensitive (LI) interface is composed of ports, with which it connects to other modules through channels. The port is the element through which a module enforces the LI protocol. The communication invariant

# In Summary

- Computer architectures are increasingly **heterogeneous**
- Heterogeneity raises design **complexity**
- Coping with complexity requires
  1. **raising the level of abstraction** in hardware design and
  2. **embracing design for reusability**
- **High-level synthesis** is a key technology to meet both requirements
- Flexible interfaces based on **LID Protocols & Shells Paradigm** are critical for composing circuits synthesized with HLS
- **ESP** is an open-source platform for heterogeneous SoC design that we developed based on these principles and practices



# ESP for Open-Source Hardware

- We contribute **ESP** to the OSH community in order to support the realization of
  - **more scalable** architectures for SoCs that integrate
  - **more heterogeneous** components, thanks to a
  - **more flexible** design methodology, which accommodates different specification languages and design flows
- ESP was conceived as a heterogeneous integration platform from the start and tested through years of teaching at Columbia University
- We invite you to **use ESP** for your projects and to **contribute to ESP!**

<https://www.esp.cs.columbia.edu>

The screenshot shows the ESP website interface. At the top is a navigation bar with links: Home, Resources, News, Team, and Contact. Below this is a dark blue header with the text "ESP the open-source SoC platform". Under the header are social media icons for GitHub, Twitter, YouTube, and a monitor icon. The main content area is titled "The ESP Vision" and describes ESP as an open-source research platform for heterogeneous system-on-chip design. Below the text is a diagram illustrating the design flows: Application Developers (using Keras, PyTorch, etc.) and Hardware Designers (using Chisel, SystemVerilog, etc.) feed into HLS Design Flows and RTL Design Flows, which then converge into a central "SoC Integration" flow. This flow leads to "Rapid Prototyping" on an FPGA. The diagram also shows a stack of accelerators (NVDLA.org, RISC-V, etc.) and a penguin icon representing Linux. To the right of the diagram is a sidebar titled "Latest Posts" featuring a post about an upcoming talk at VLSID & ES 2020 in Bangalore, published on Jan 2, 2020.

Home Resources News Team Contact

## ESP

the open-source SoC platform

Latest Posts

### The ESP Vision

ESP is an open-source research platform for heterogeneous system-on-chip design that combines a flexible tile-based architecture and a modular system-level design methodology.

Application Developers

Hardware Designers

HLS Design Flows

RTL Design Flows

SoC Integration

Rapid Prototyping

accelerator

accelerator

NVDLA.org

accelerator

RISC-V

Upcoming talk at VLSID & ES 2020 in Bangalore

We will give a talk about ESP in Bangalore (India) on January 5th at the International Conference on VLSI Design and International Conference on Embedded Design (VLSID & ES).

Read more

Published: Jan 2, 2020





# Some Recent Publications

Available at [www.cs.columbia.edu/~luca](http://www.cs.columbia.edu/~luca)

1. L. P. Carloni. From Latency-Insensitive Design to Communication-Based System-Level Design [The Proceedings of the IEEE, Vol. 103, No. 11, November 2015.](#)
2. L. P. Carloni. The Case for Embedded Scalable Platforms [DAC 2016. \(Invited Paper\).](#)
3. L. P. Carloni et al. Teaching Heterogeneous Computing with System-Level Design Methods, [WCAE 2019.](#)
4. E. G. Cota, P. Mantovani, G. Di Guglielmo, and L. P. Carloni. An Analysis of Accelerator Coupling in Heterogeneous Architectures. [DAC 2015.](#)
5. P. Mantovani, E. Cota, K. Tien, C. Pilato, G. Di Guglielmo, K. Shepard and L. P. Carloni. An FPGA-Based Infrastructure for Fine-Grained DVFS Analysis in High-Performance Embedded Systems. [DAC 2016.](#)
6. P. Mantovani, E. Cota, C. Pilato, G. Di Guglielmo and L. P. Carloni. Handling Large Data Sets for High-Performance Embedded Applications in Heterogeneous Systems-on-Chip. [CASES 2016.](#)
7. P. Mantovani, G. Di Guglielmo and L. P. Carloni. High-Level Synthesis of Accelerators in Embedded Scalable Platforms. [ASPDAC 2016.](#)
8. L. Piccolboni, P. Mantovani, G. Di Guglielmo, and L. P. Carloni. COSMOS: Coordination of High-Level Synthesis and Memory Optimization for Hardware Accelerators. [ACM Transactions on Embedded Computing Systems, 2017.](#)
9. C. Pilato, P. Mantovani, G. Di Guglielmo, and L. P. Carloni. System-Level Optimization of Accelerator Local Memory for Heterogeneous Systems-on-Chip. [IEEE Trans. on CAD of Integrated Circuits and Systems, 2017.](#)
10. D. Giri, P. Mantovani, and L. P. Carloni. Accelerators & Coherence: An SoC Perspective. [IEEE MICRO, 2018.](#)







Thank you from the **ESP** team!

<https://esp.cs.columbia.edu>

<https://github.com/sld-columbia/esp>



System Level Design Group



COMPUTER SCIENCE

