How High-Level Synthesis Enables Design for Reusability of Hardware Accelerators

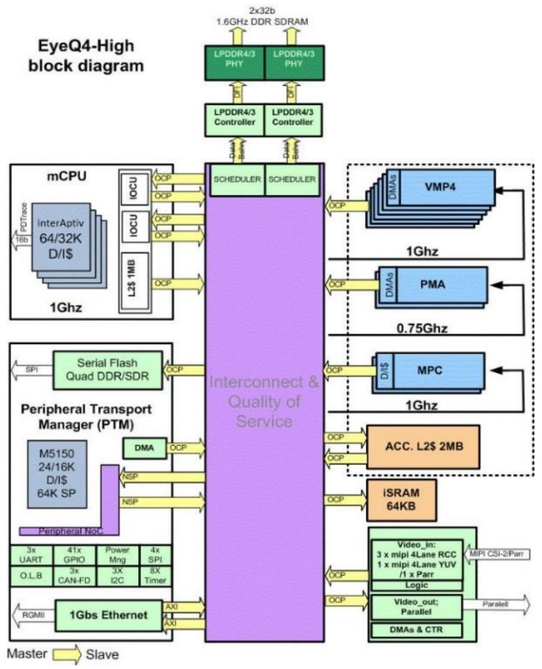
Luca Carloni

Department of Computer Science

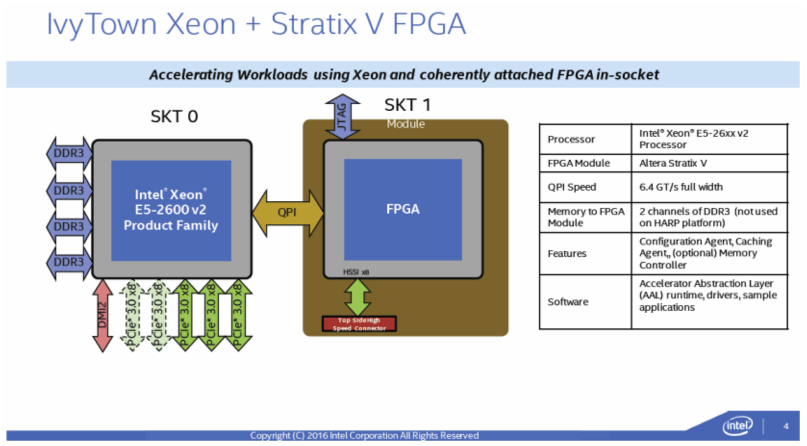
Columbia University in the City of New York

"Quo Vadis Logic Synthesis?", a Workshop at DATE'19 · Florence, Italy · March 29, 2019

Heterogeneous Architectures Are Emerging Everywhere



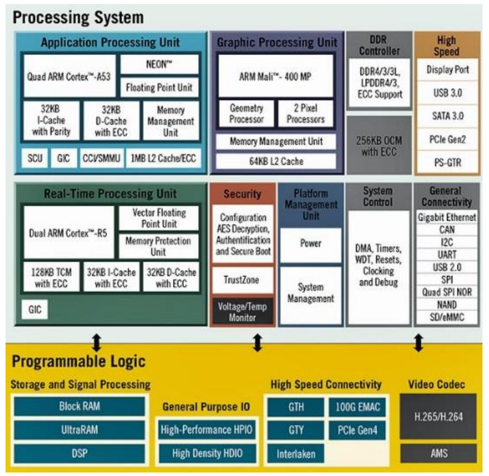
[Source: www.mobileye.com/]



IvyTown Xeon + Stratix V FPGA
Accelerating Workloads using Xeon and coherently attached FPGA in-socket
SKT 0
SKT 1
DDR3
Intel Xeon E5-2600 v2 Product Family
QPI
JTAG
FPGA
Module
Copyright (C) 2016 Intel Corporation All Rights Reserved

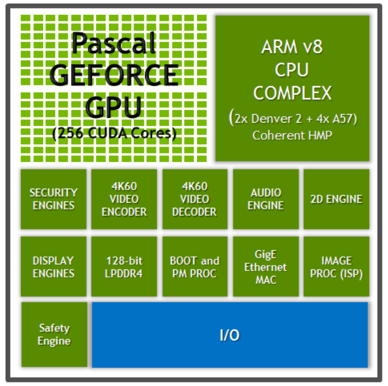
Processor	Intel® Xeon® E5-26xx v2 Processor
FPGA Module	Altera Stratix V
QPI Speed	6.4 GT/s full width
Memory to FPGA Module	2 channels of DDR3 (not used on HARP platform)
Features	Configuration Agent, Caching Agent, (optional) Memory Controller
Software	Accelerator Abstraction Layer (AAL) runtime, drivers, sample applications

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

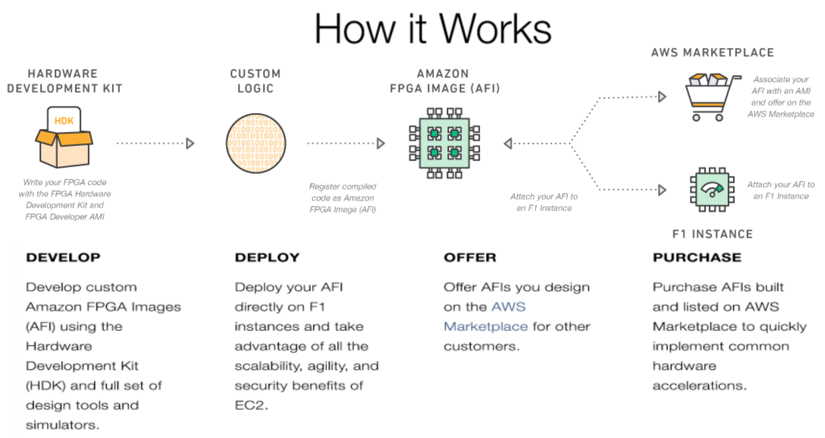


[Source: www.xilinx.com/]

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



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



How it Works

HARDWARE DEVELOPMENT KIT — Write your FPGA code with the FPGA Hardware Development Kit and FPGA Developer AMI

CUSTOM LOGIC — Register compiled code as Amazon FPGA Image (AFI)

AMAZON FPGA IMAGE (AFI)

AWS MARKETPLACE — Associate your AFI with an AMI and offer on the AWS Marketplace

F1 INSTANCE — Attach your AFI to an F1 Instance

DEVELOP

Develop custom Amazon FPGA Images (AFI) using the Hardware Development Kit (HDK) and full set of design tools and simulators.

DEPLOY

Deploy your AFI directly on F1 instances and take advantage of all the scalability, agility, and security benefits of EC2.

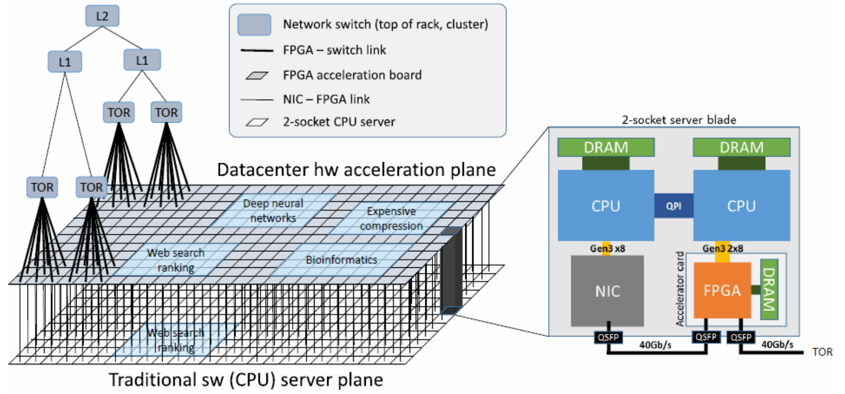
OFFER

Offer AFIs you design on the AWS Marketplace for other customers.

PURCHASE

Purchase AFIs built and listed on AWS Marketplace to quickly implement common hardware accelerations.

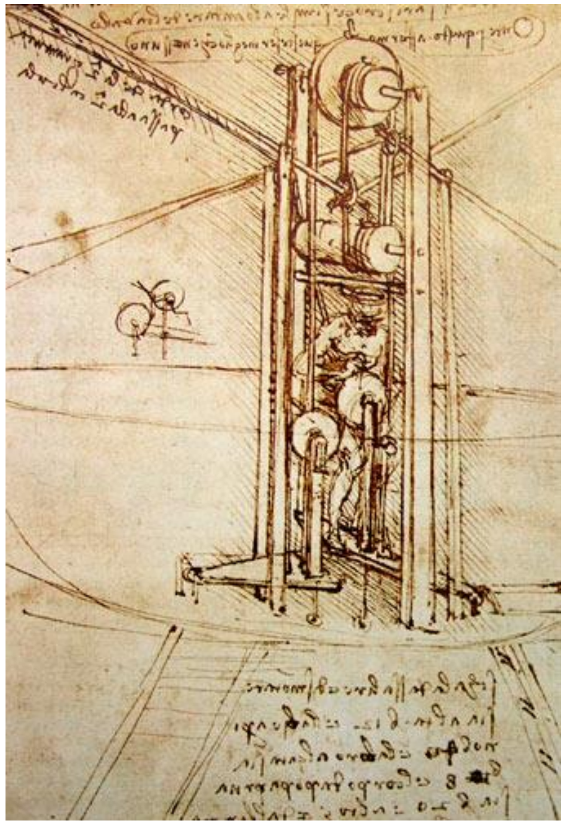
[Source: https://aws.amazon.com/ec2/instance-types/f1/]



[Source: www.microsoft.com/]

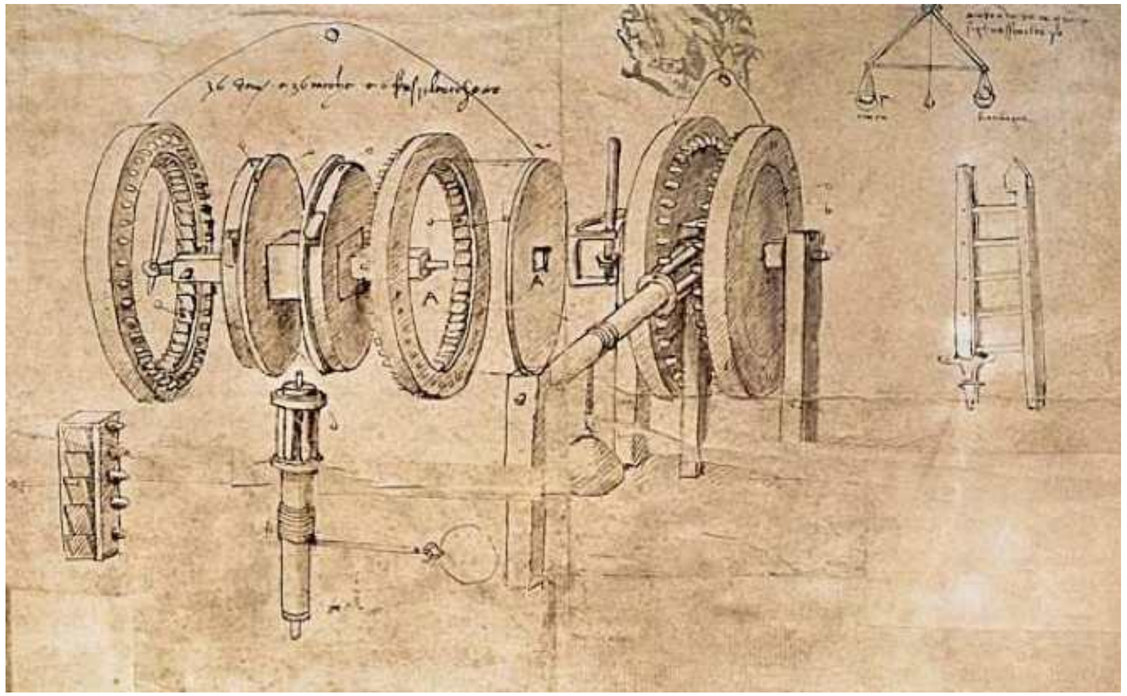
A (Perhaps Easy?) Prediction: No Single Architecture Will Emerge as the Sole Winner

- **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?

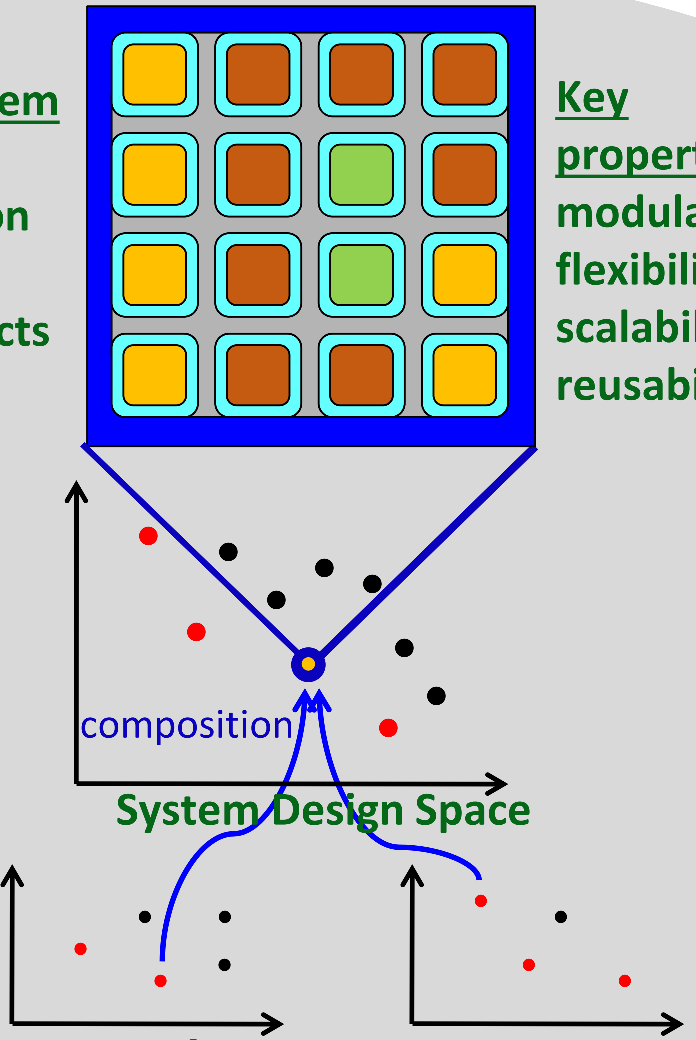


- How to handle this complexity?
  - raise the level of abstraction to System-Level Design
  - adopt compositional design methods with the Protocol & Shell Paradigm
  - promote Design Reuse

Our Vision: A System-Level Design Ecosystem

Architect
G005
1+
G
Select and integrate the best set of IP components to design and program an SoC for a target application domain

<u>SLD Ecosystem</u> promotes collaboration between SoC Architects and IP Core Designers



<u>Key properties:</u> modularity, flexibility, scalability reusability

Expert Builder
WM018
δ
3+
WM
Develop specialized components at a high level of abstraction based on unique intellectual property knowledge

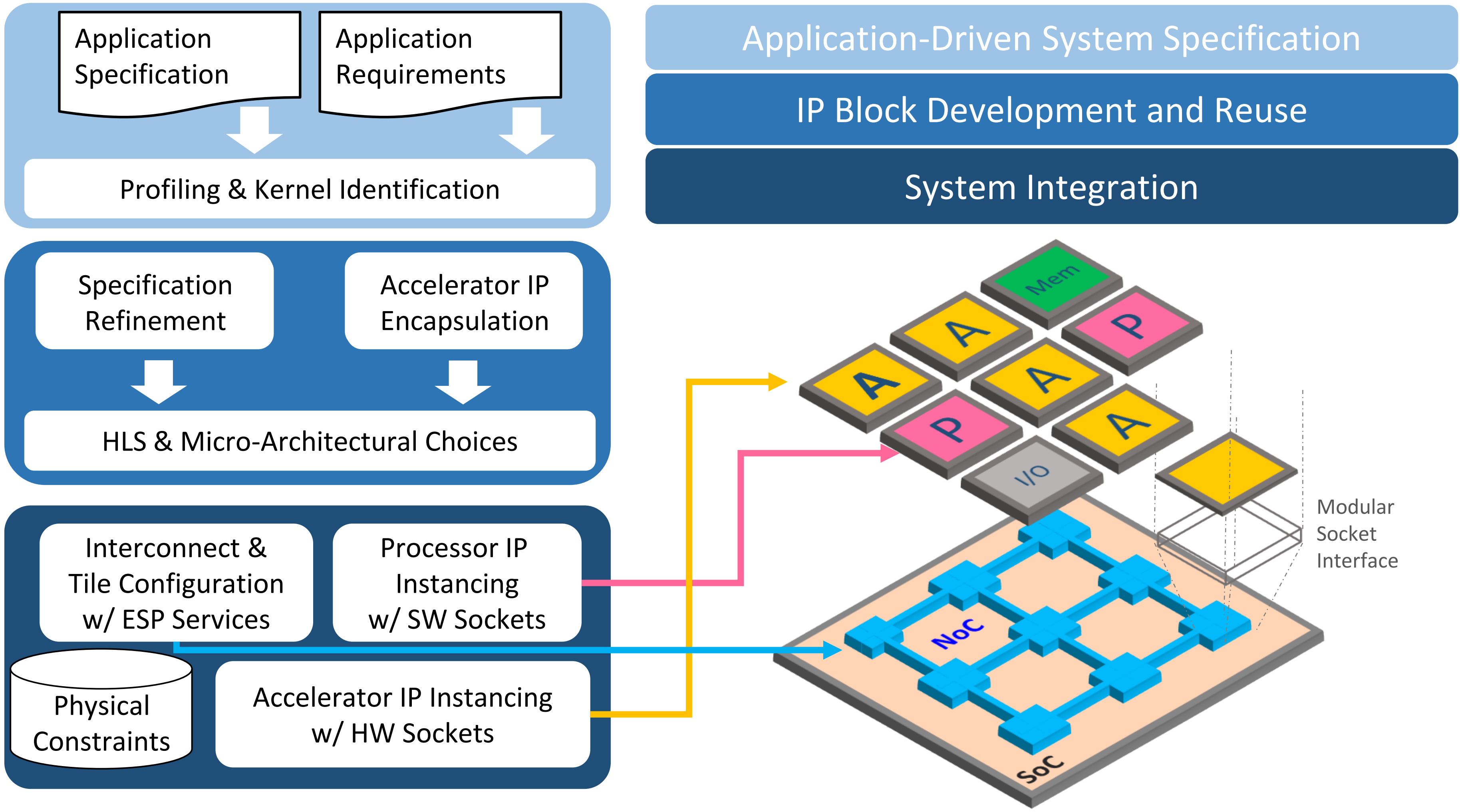
Soft IP Design Spaces

Our System-Level Design Approach: 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 ISA simulators to Virtual Platforms
  - Move from Verilog/VHDL to SystemC, also an IEEE standard
  - 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

Embedded Scalable Platforms: Architecture + Methodology

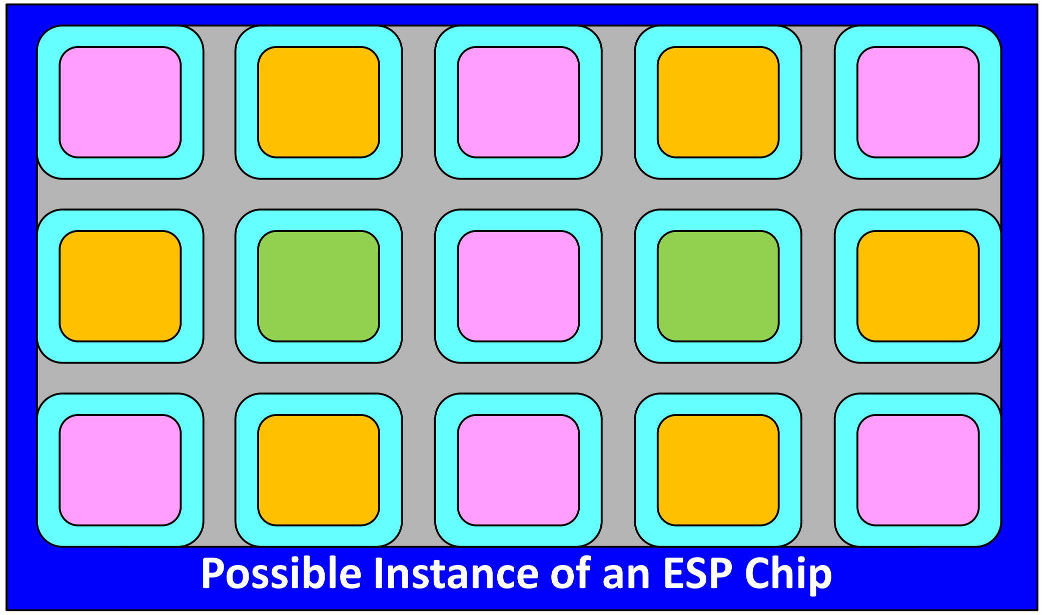
- The **flexible architecture** simplifies the integration of heterogeneous components by
  - balancing regularity and specialization
  - relying on the Protocol & Shell paradigm and scalable communication infrastructure
- The **system-level design methodology** promotes HW/SW co-design and is supported by
  - a mix of commercial and in-house CAD tools
  - a growing library of reusable IP blocks



Application Specification
Application Requirements
Profiling & Kernel Identification
Specification Refinement
Accelerator IP Encapsulation
HLS & Micro-Architectural Choices
Interconnect & Tile Configuration w/ ESP Services
Processor IP Instancing w/ SW Sockets
Physical Constraints
Accelerator IP Instancing w/ HW Sockets
Application-Driven System Specification
IP Block Development and Reuse
System Integration
Mem
A
P
A
A
P
A
I/O
NoC
SoC
Modular Socket Interface

[L. P. Carloni, *The Case for Embedded Scalable Platforms*, DAC 2016]

The ESP Scalable Architecture Template



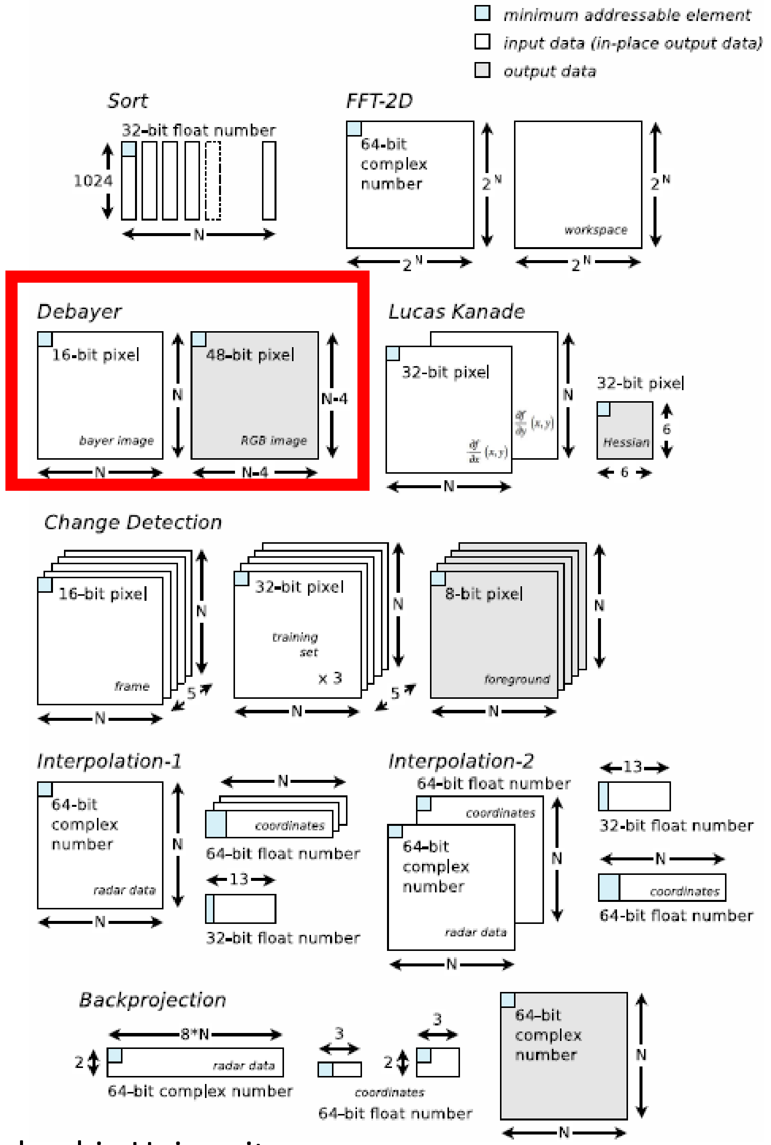
- **Processor Tiles**
  - each hosting at least one configurable processor core capable of running an OS
- **Accelerator Tiles**
  - synthesized from high-level specs
- **Other Tiles**
  - memory interfaces, I/O, etc.
- **Network-on-Chip (NoC)**
  - playing key roles at both design and run time

Template Properties

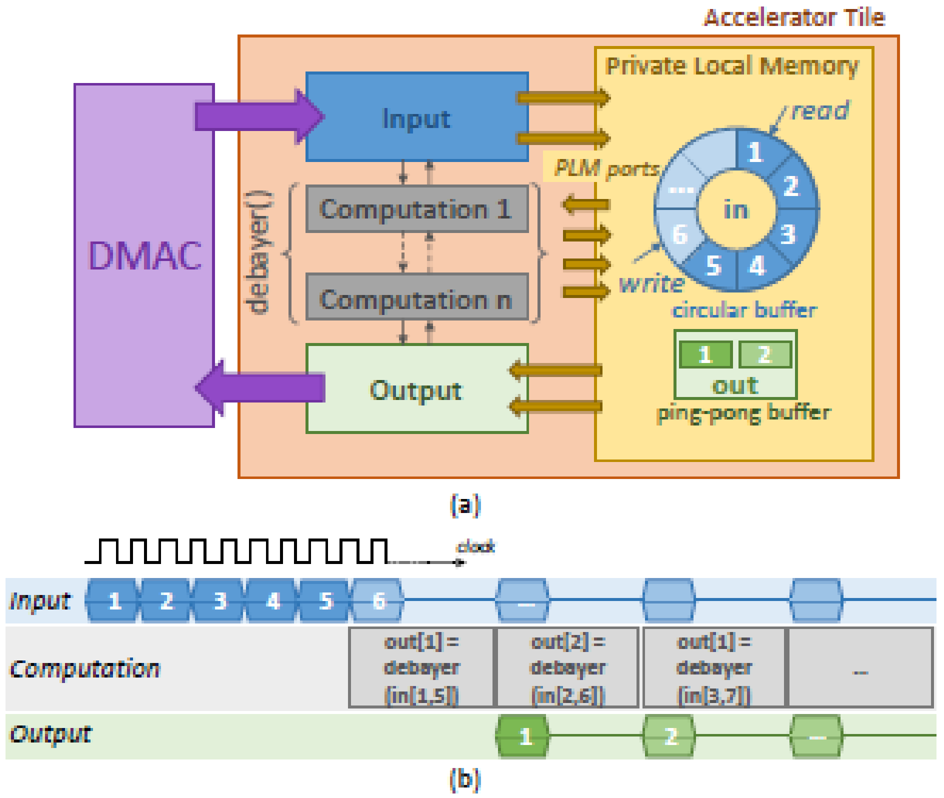
- **Regularity**
  - tile-based design
  - pre-designed on-chip infrastructure for communication and resource management
- **Flexibility**
  - each ESP design is the result of a configurable mix of programmable tiles and accelerator tiles
- **Specialization**
  - with automatic high-level synthesis of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

Heterogeneous Applications Bring Heterogeneous Requirements

Data Structures of the PERFECT TAV Benchmarks



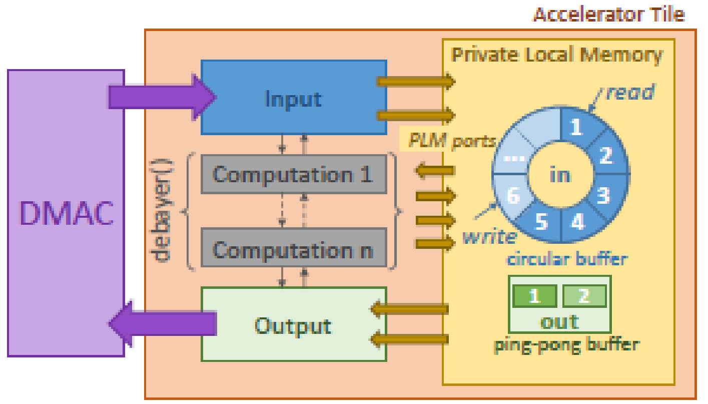
Structure and Behavior of the Debayer Accelerator



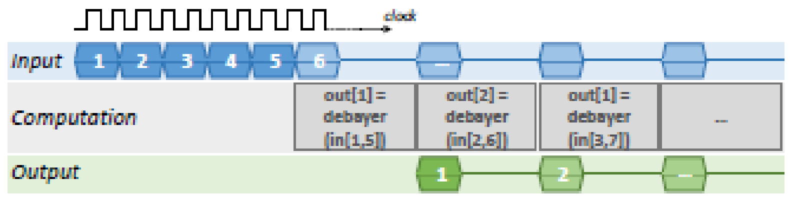
- **While the Debayer structure and behavior is representative of the other benchmarks, the specifics of the actual computations, I/O patterns, and scratchpad memories vary greatly among them**

Example of ESP Accelerator Design: Debayer - 1

```
#include <systemc.h>
SC_MODULE(Debayer) {
 sc_in<bool> clk, rst;
private:
 sc_signal<bool> i_valid, i_ready, o_valid, o_ready;
 int A0[6][2048];  // circular buffer
 int B0[2048];
 int B1[2048];
public:
 //...
 SC_CTOR(debayer) {
  SC_CTHREAD(input, clk.pos());
  reset_signal_is(rst, false);
  SC_CTHREAD(compute, clk.pos());
  reset_signal_is(rst, false);
  SC_CTHREAD(output, clk.pos());
  reset_signal_is(rst, false);
  //...
 }
 void input(void) {
  // reset ...
  unsigned circ = 0; // circular buffer write pointer
  wait();
  while(true) {
   L0: for (int r=0; r<2048; r++) {
    // DMA request
    // read input ...
    L1:  for (int c=0; c<2048; c++)
     { A0[circ][c] = f(...); } //write to A0
    // output ...
    if (r >=5){
      // wait for ready from compute then notify as valid
    }
    circ++;
    if (circ == 6)
     circ = 0;
   }
  }
 }
 void compute(void) {
  int PAD = 2; bool flag = true;
  int r_r = 0; // central row of the mask
  // reset ...
  wait();
  while(true) {
   L2: for (int r=0; r<2048-PAD; r++) {
    // (wait for valid from input then notify as ready)
    r_r = circ_buffer_row(r + 2);
    L3:  for (int j=PAD; j<2048-PAD; j++) {
      if (flag) B0[j] = g(A0[r_r][j-2], A0[r_r][j-1],
         A0[r_r][j], A0[r_r][j+1], A0[r_r][j+2], ...);
      else B1[j] = g(A0[r_r][j-2], A0[r_r][j-1],
         A0[r_r][j], A0[r_r][j+1], A0[r_r][j+2], ...);
     }
    // (valid to output, ready to compute)
    flag = !flag;
   }
  }
 }
 void output(void) {
  int PAD = 2; bool flag = true;
  // reset ...
  wait();
  while(true) {
   L4: for (int r=PAD; r<2048-PAD; r++) {
    // (wait for valid from compute then notify as ready)
    // prepare DMA request
    // send data
    L5:  for (int c=PAD; c<2048-PAD; c++) {
     if (flag) h(B0[c], ...);  //read from array B0
     else h(B1[c], ...);       //read from array B1
    }
    // (ready to compute)
    flag = !flag;
   }
  }
 }
};
```



(a)


clock
Input
1
2
3
4
5
6
Computation
out[1] = debayer (in[1,5])
out[2] = debayer (in[2,6])
out[1] = debayer (in[3,7])
...
Output
1
2

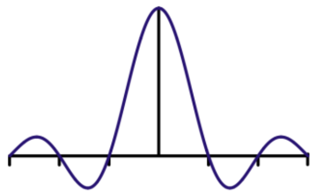
(b)

- 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

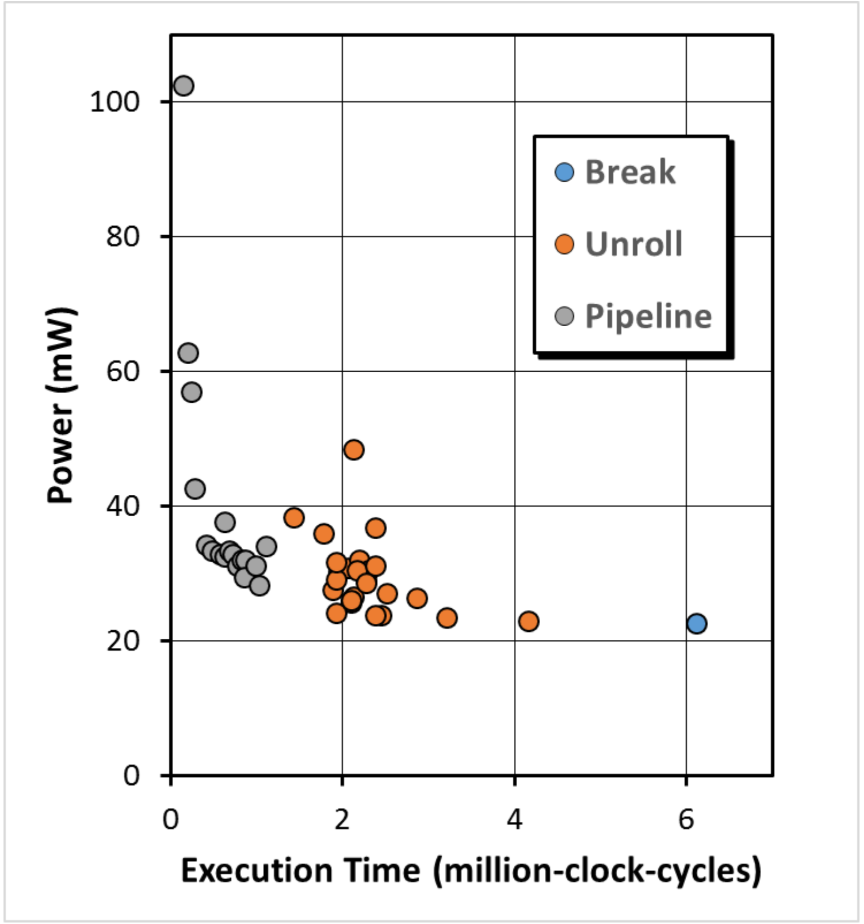
Example of Design-Space Exploration: Accelerator for the SAR Interp-1 Kernel

Main loop in *Interpolation-1* kernel

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

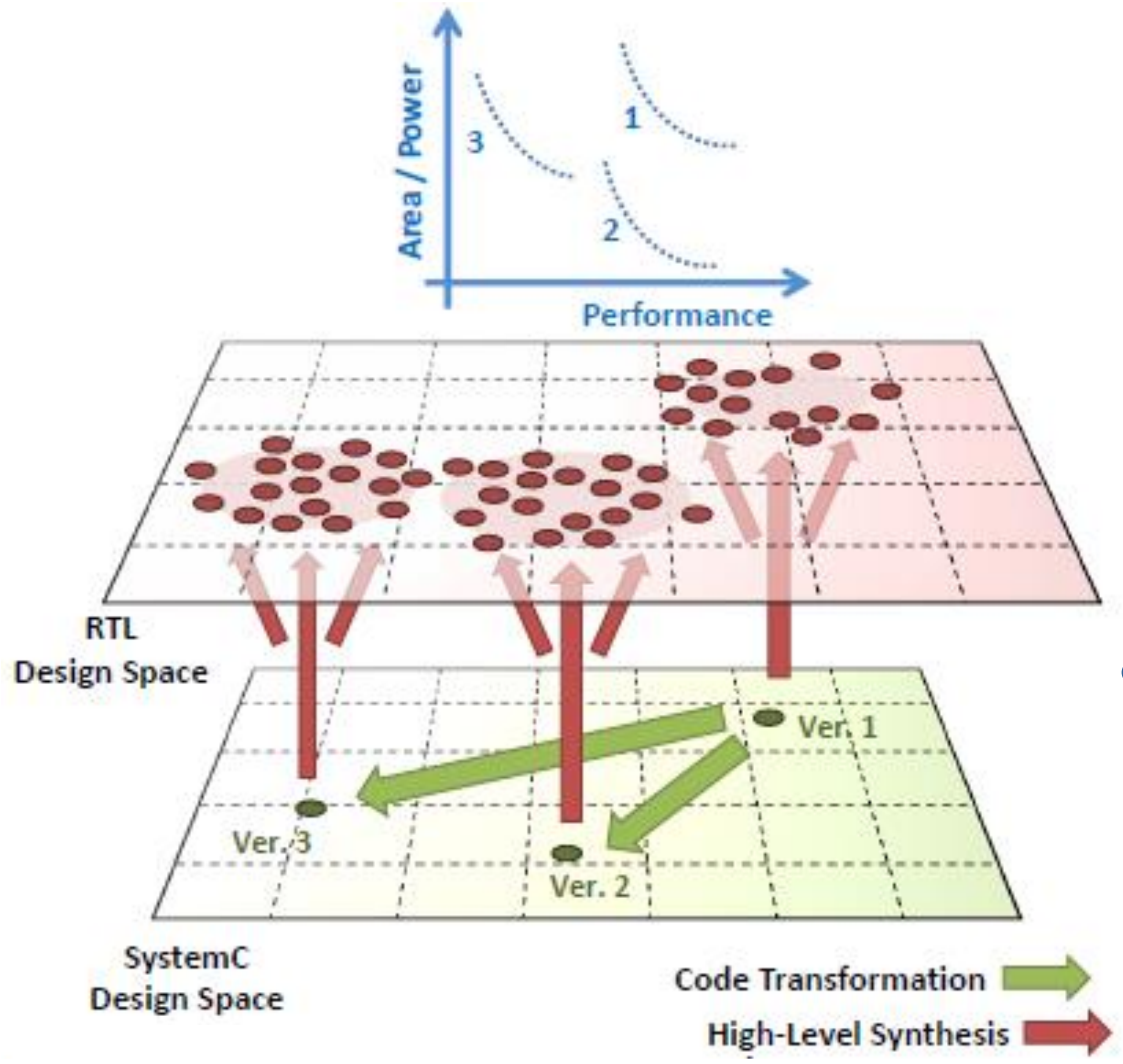


Pareto Set Obtained with High-Level Synthesis (1GHz@1V, CMOS 32nm)



- 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

High-Level Synthesis Drives Design-Space Exploration

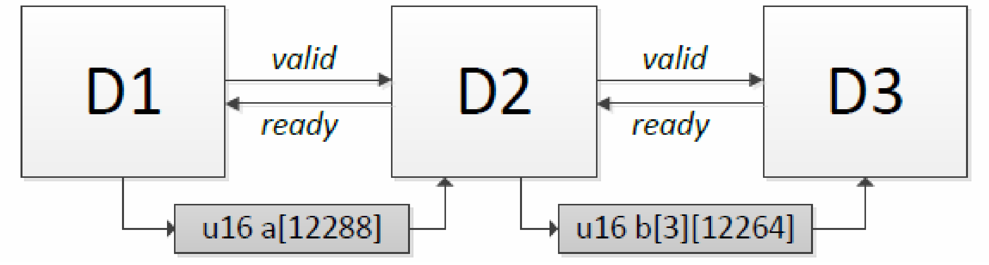


- **Given a SystemC specification, HLS tools provide a rich set of configuration knobs to synthesize a variety of RTL implementations**
  - these implementations have different micro-architectures and provide different cost-performance trade-offs
- **Engineers can focus on revising the high-level specification**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

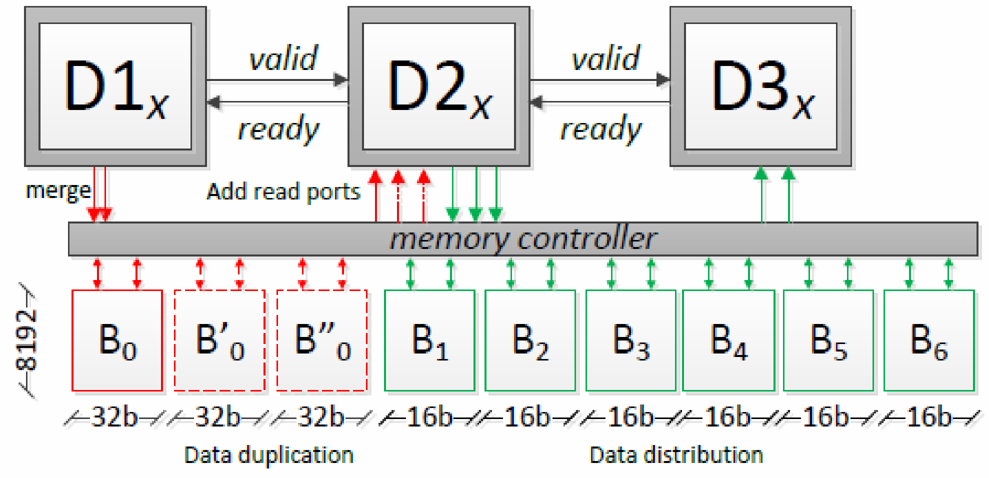
Accelerator Design Productivity: 1 Accelerator x Student x Month

- **In 3 months 3 students designed 9 TAV accelerators by performing these steps**
  1. Algorithm study and specification in synthesizable SystemC
  2. Design-space exploration with high-level synthesis
  3. Integration with LEON-3 processor and Linux OS
  4. Synthesis of two implementations
     - FPGA
     - 32nm standard cells to build accelerator tile
  5. Testing, validation, and performance/power evaluation

Dataflow of WAMI *Debayer* Kernel

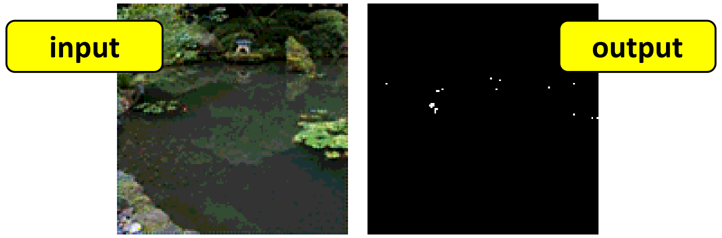


Optimization of Scratchpad Memory within Accelerator Tile for *Debayer* Kernel



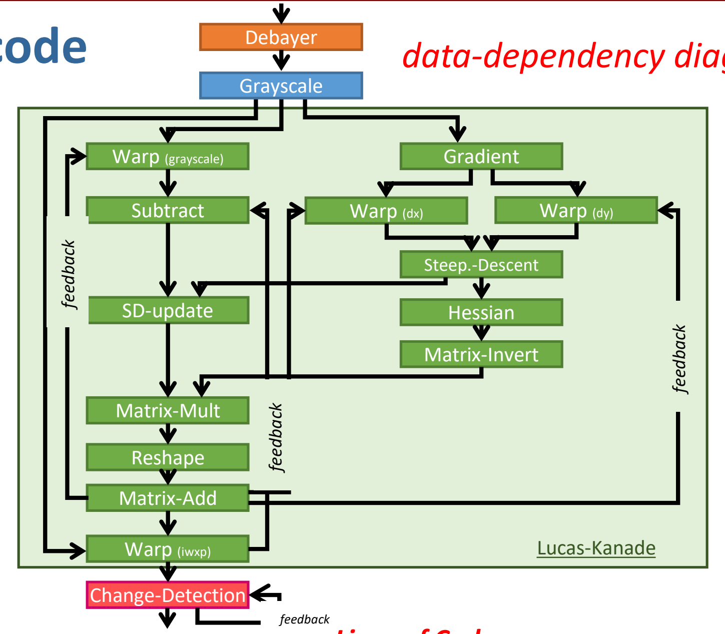
ESP Design Example: An Accelerator for Wide Area Motion Imagery

- **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

data-dependency diagram



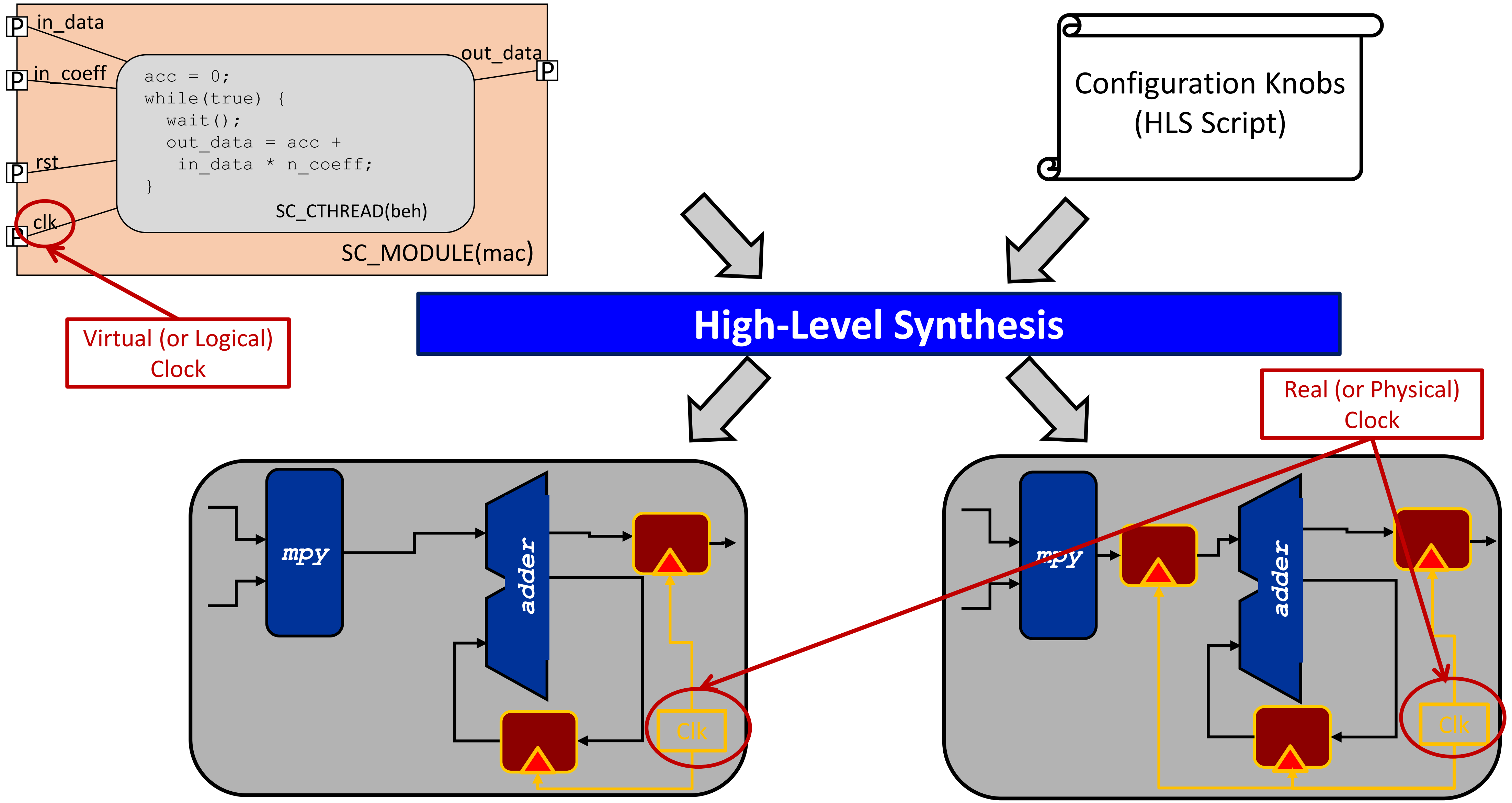
Lines of Code

Kernels	***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**

Lucas-Kanade

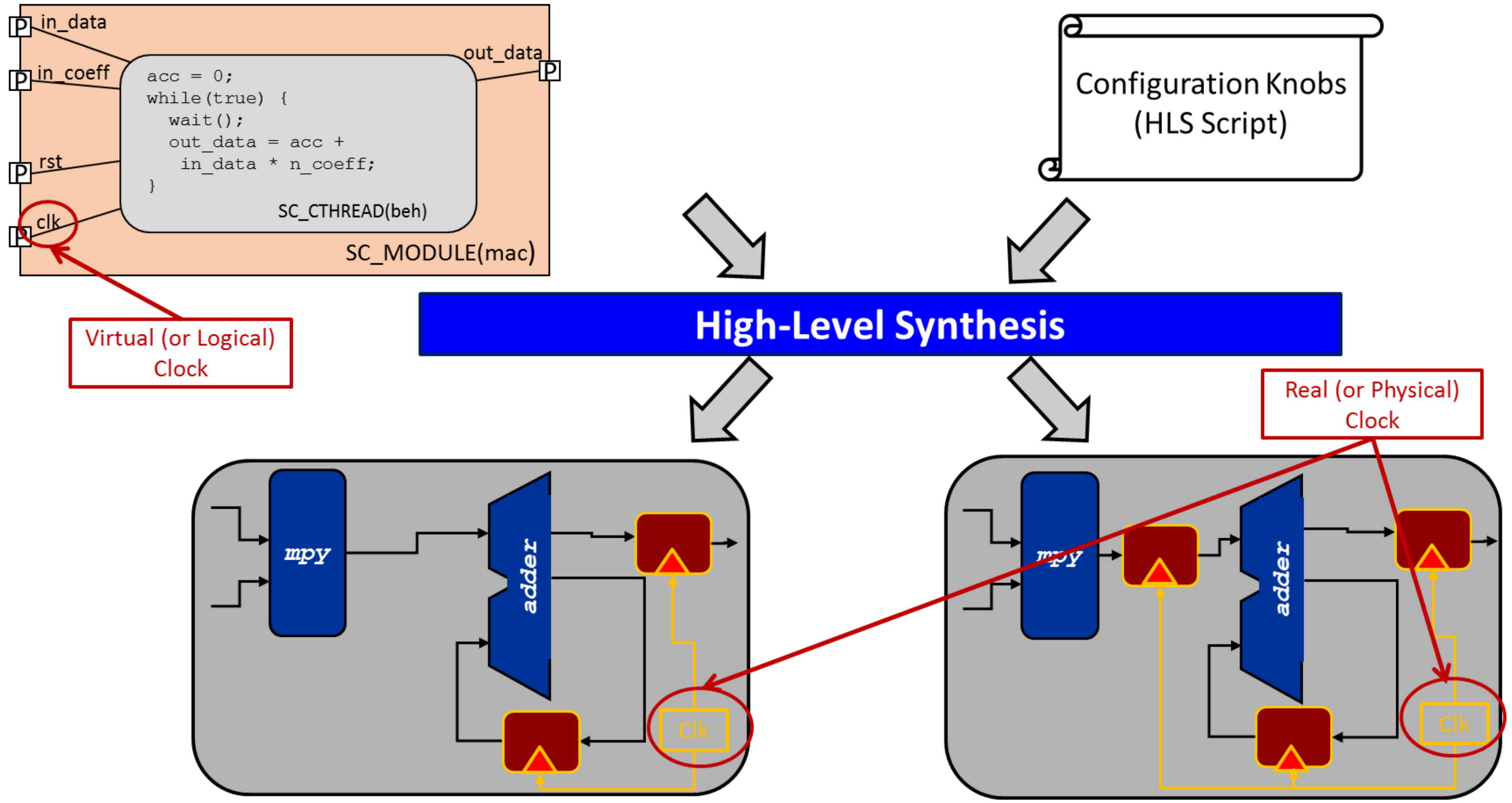
[P. Mantovani, G. Di Guglielmo, and L. P. Carloni, High-Level Synthesis of Accelerators in Embedded Scalable Platforms, ASPDAC 2016]

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



```
acc = 0;
while(true) {
  wait();
  out_data = acc +
   in_data * n_coeff;
}
```

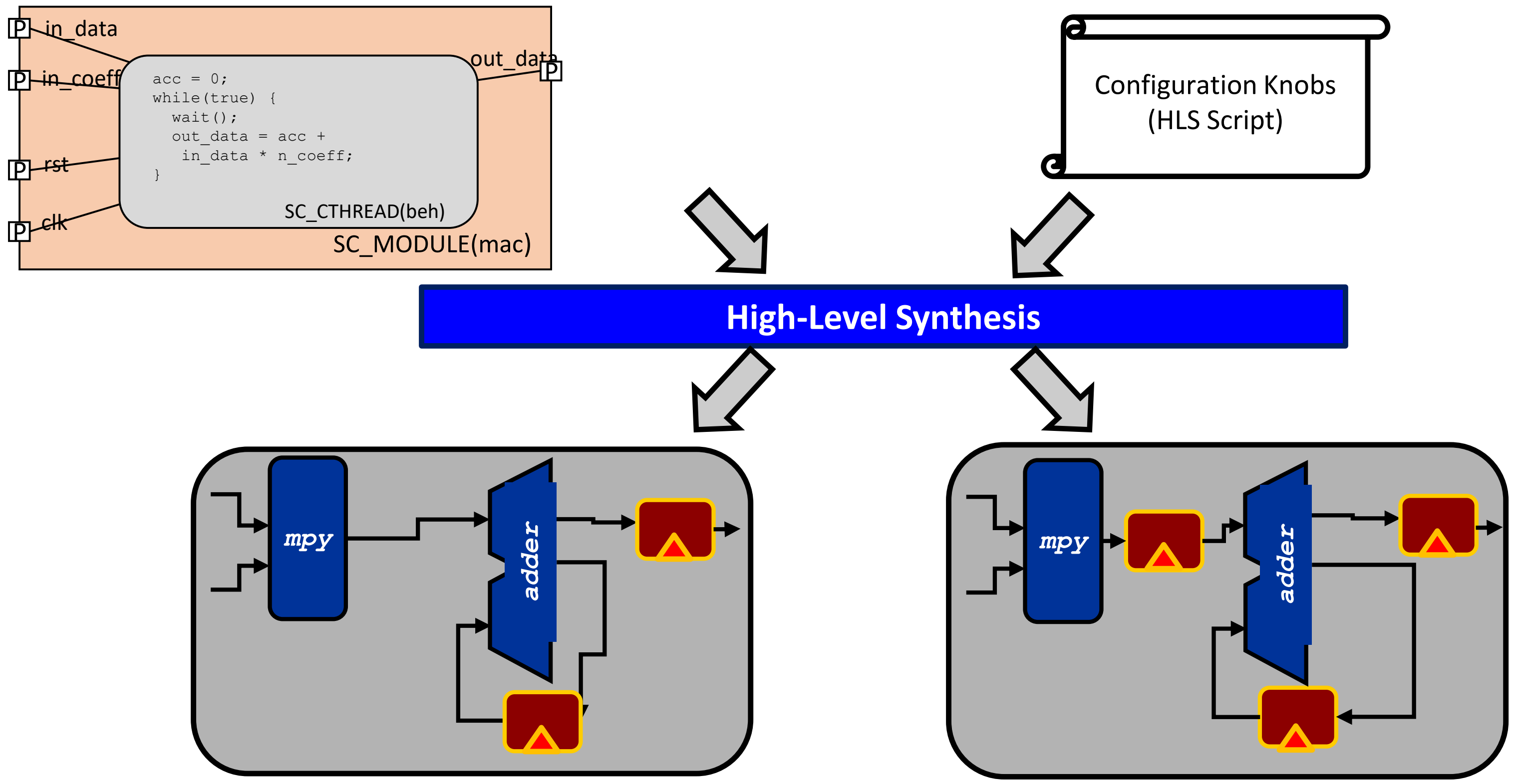
From SystemC to RTL via HLS: Two Key Questions



```
acc = 0;
while(true) {
  wait();
  out_data = acc +
   in_data * n_coeff;
}
```

- **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



```
acc = 0;
while(true) {
  wait();
  out_data = acc +
   in_data * n_coeff;
}
```

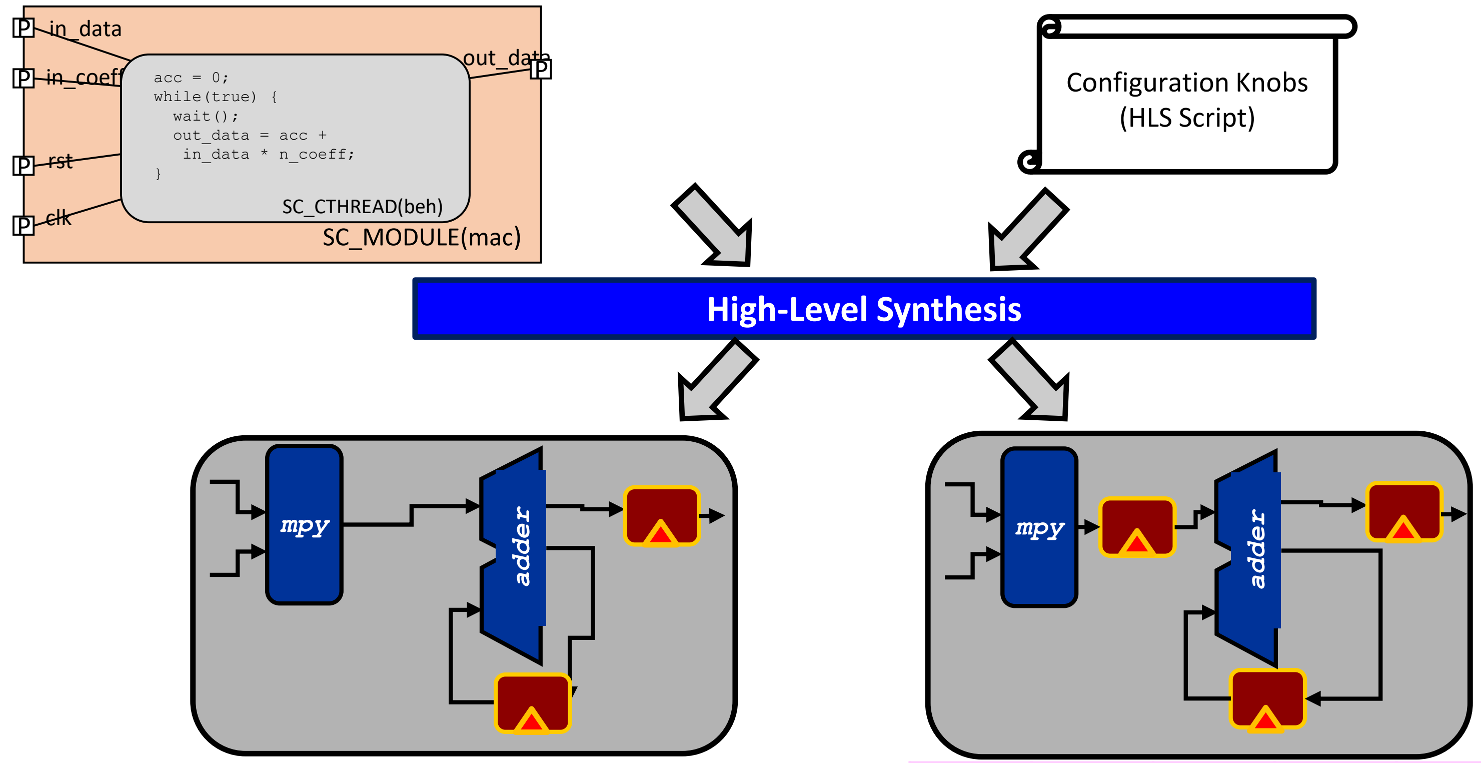
- This implementation has lower latency and lower area but also runs at lower (physical) clock frequency

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

- How to compare various synthesized implementations?
  - in terms of cost
  - in terms of performance

- Which implementation is better?

From SystemC to RTL via HLS: Correctness



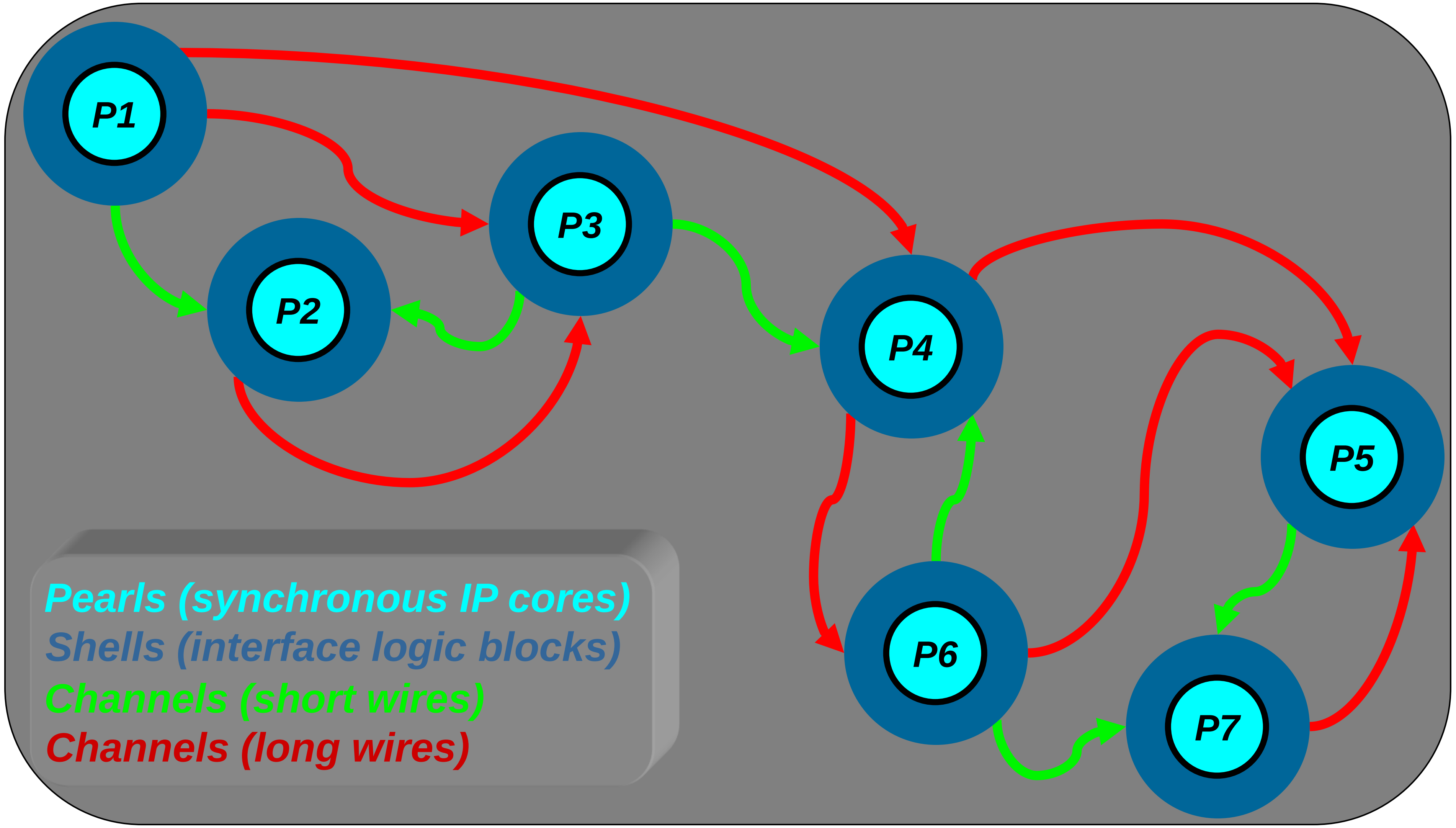
```
acc = 0;
while(true) {
  wait();
  out_data = acc +
   in_data * n_coeff;
}
```

inData	inCoeff	x	outData
3	2	0	0
5	1	6	6
7	2	11	11
9	1	25	25
		34	34
		34	34

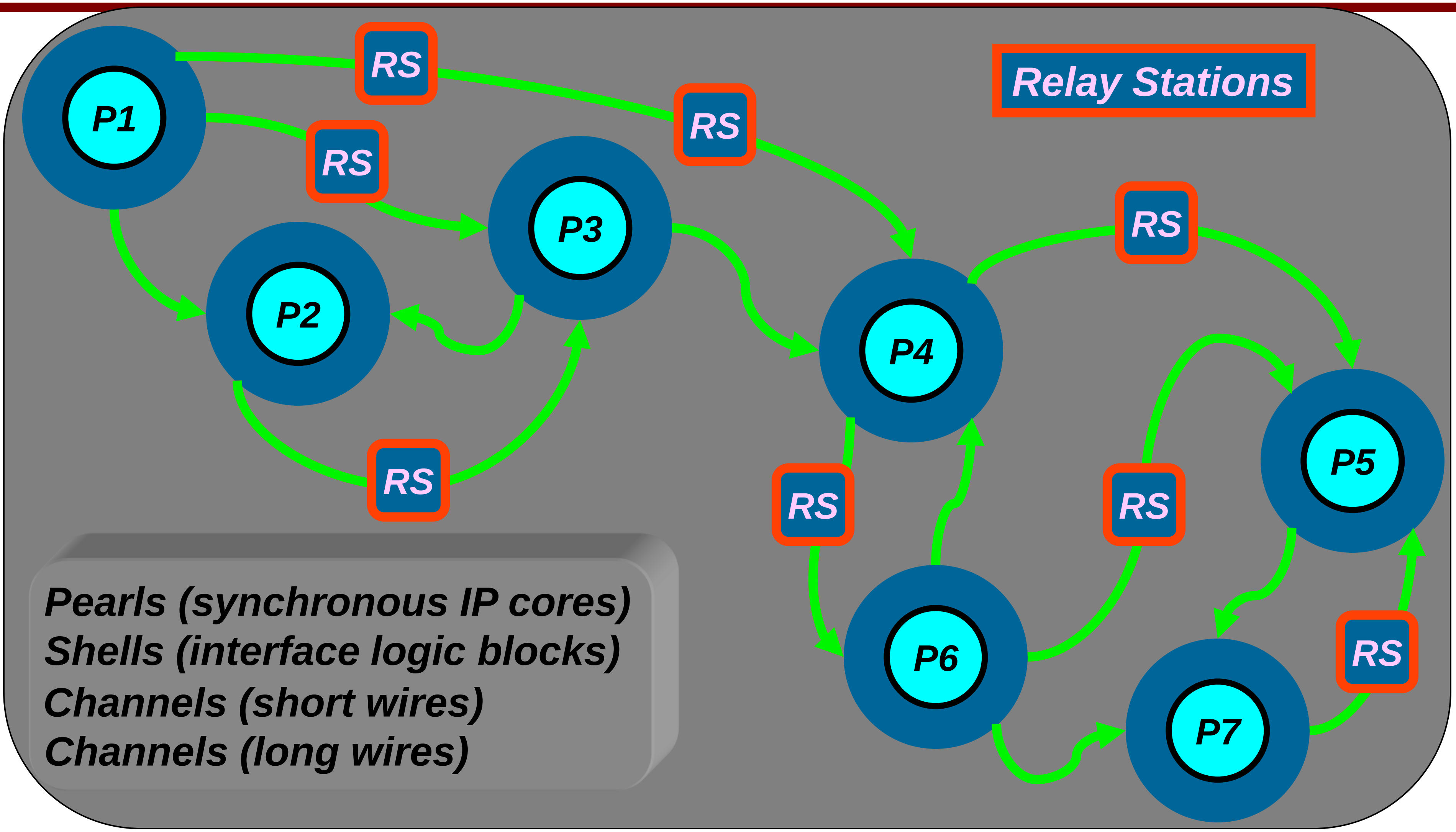
inData	inCoeff	y	x	outData
3	2	0	0	0
5	1	6	0	0
7	2	5	6	6
9	1	14	11	11
		9	25	25
			34	34

- **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?**

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


P1
P2
P3
P4
P5
P6
P7
Pearls (synchronous IP cores)
Shells (interface logic blocks)
Channels (short wires)
Channels (long wires)

Correct-by-Construction Design Methodology Enables Automatic Wire Pipelining

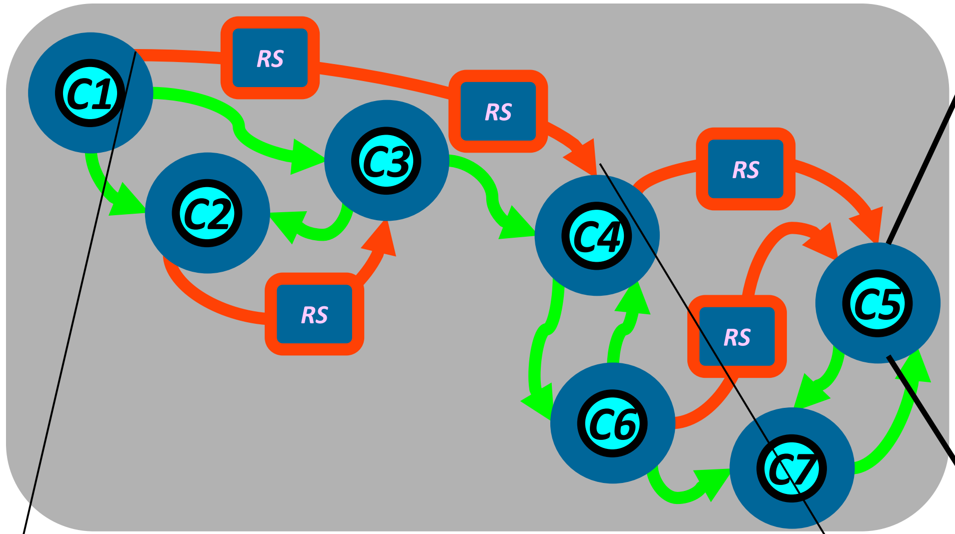


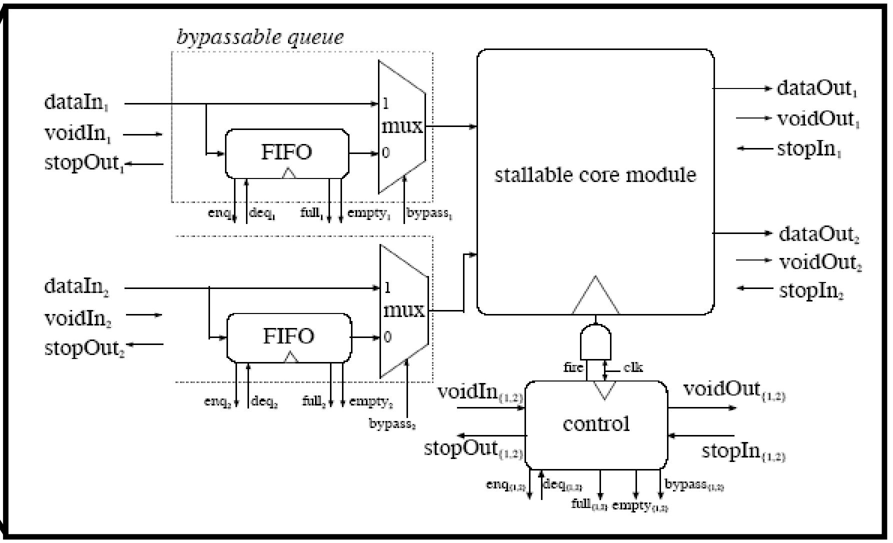
Pearls (synchronous IP cores)
Shells (interface logic blocks)
Channels (short wires)
Channels (long wires)

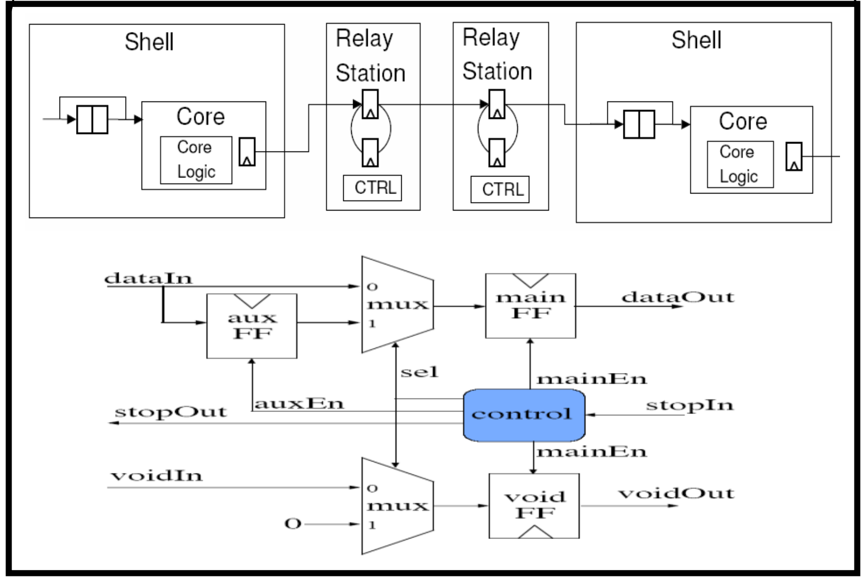
Relay Stations are sequential elements initialized with void data items

Retrospective: Latency-Insensitive Design

[Carloni et al. '99]



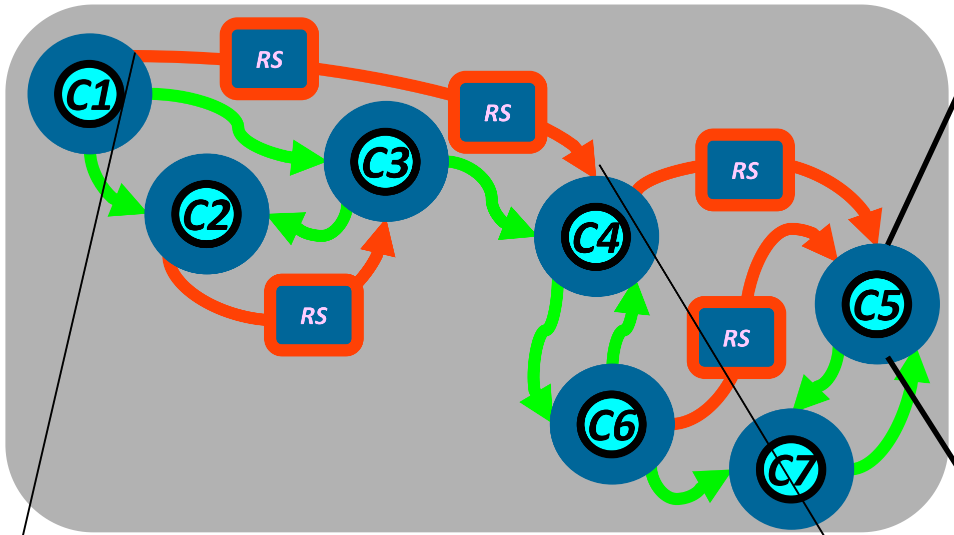


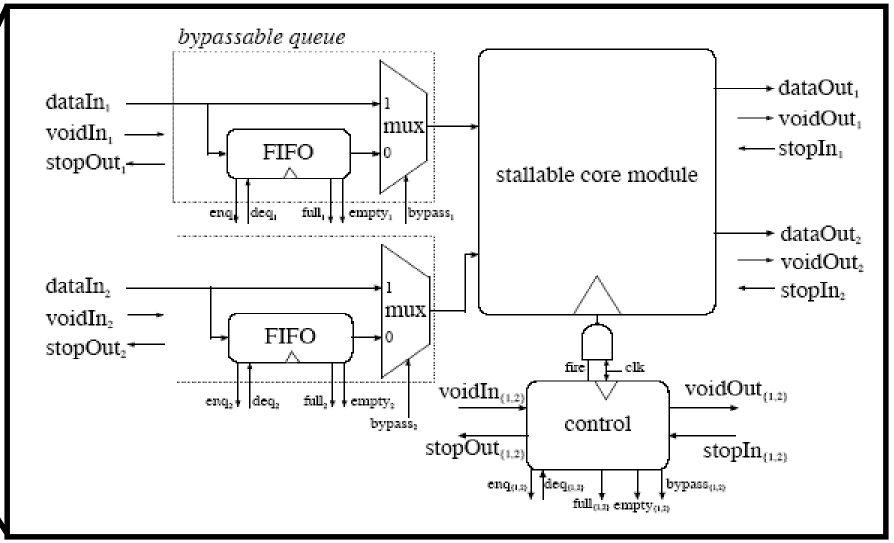


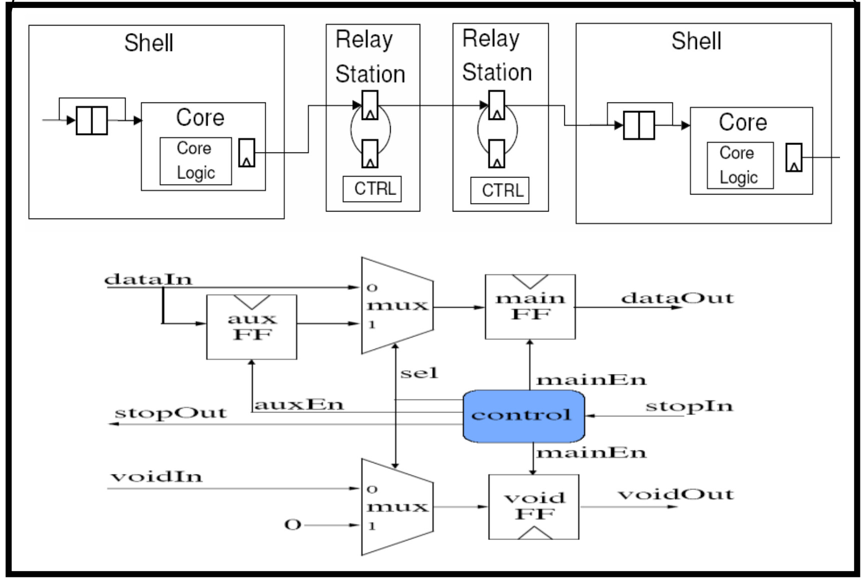
Latency-Insensitive Design

- 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

Benefits of the Protocols & Shells Paradigm



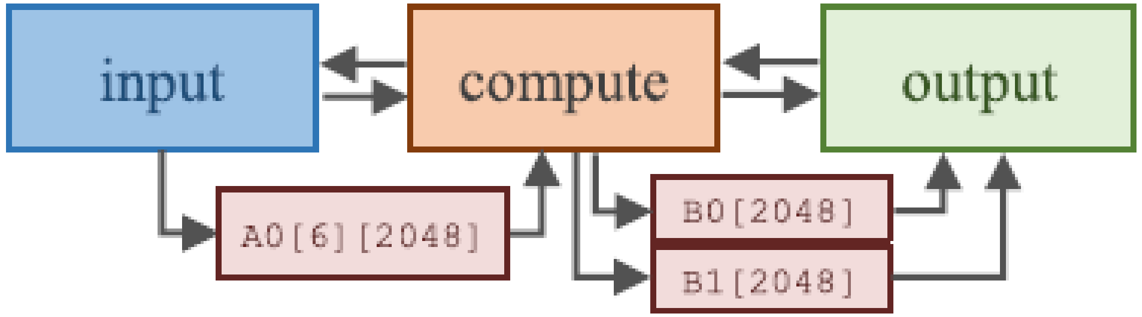




The Protocol & Shells Paradigm

- 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 design **flexibility** up to late stages of the design process

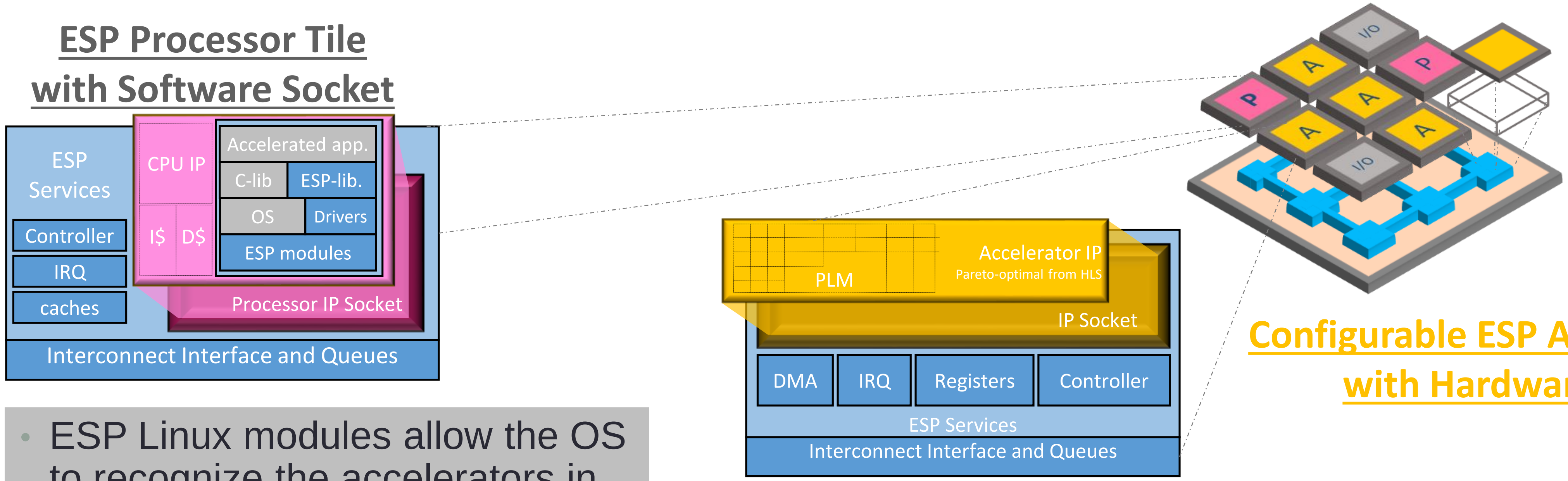
Example of ESP Accelerator Design: Debayer - 2



- **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 Accelerators: Hardware – Software Integration



- ESP Linux modules allow the OS to recognize the accelerators in the ESP tiles…
- …and simplify the programming of the ESP Linux device drivers for the accelerators
  - the user-provided accelerator-specific code is less than 2% for the WAMI-app device drivers

- Signal-level interface matches the accelerator interface of the general model
- Memory-mapped configuration registers match those defined in the accelerator model
- TLM abstractions and HLS allow the accelerator designer to be unaware of the particular specification of the SoC interconnection

[L. P. Carloni, *The Case for Embedded Scalable Platforms*, DAC 2016]

“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

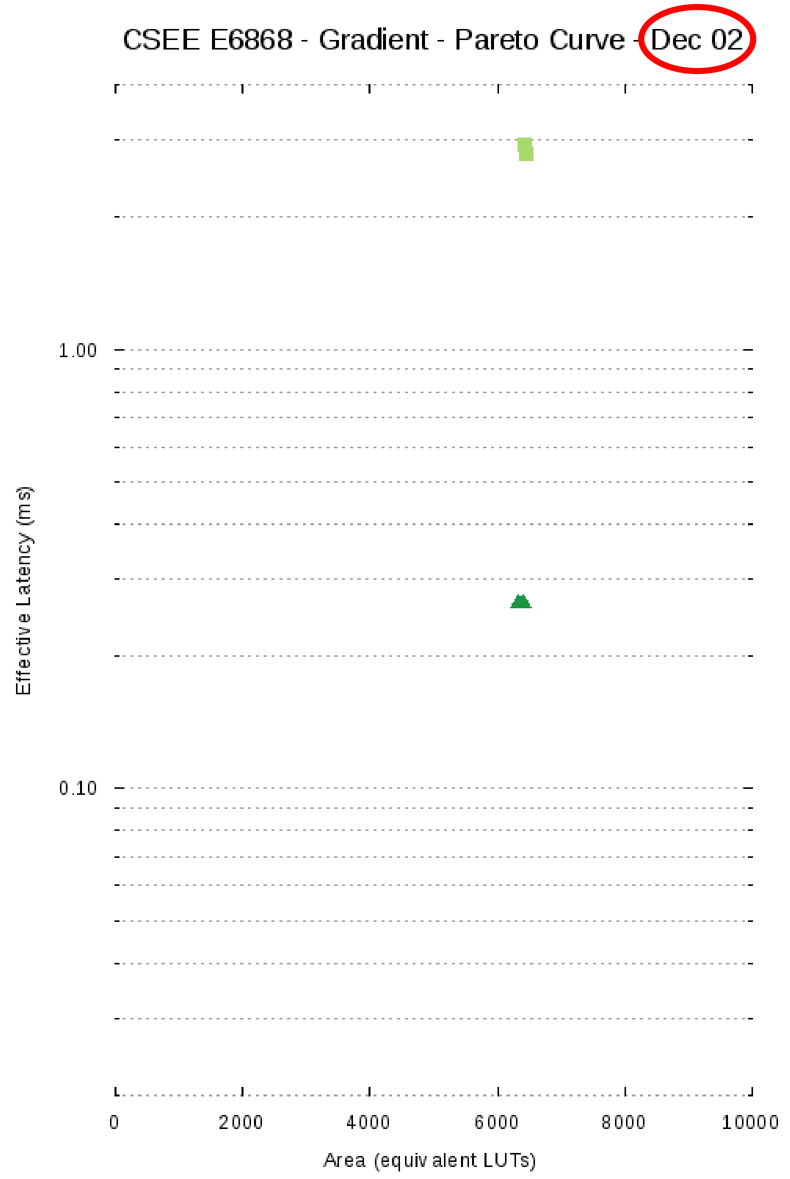
New Course

- CSEE-4868: System-on-Chip Platforms
  - *Foundation course on the programming, design, and validation of SoCs with emphasis on high-performance embedded applications*
  - Offered at Columbia since Spring 2011, part of both the CS and EE graduate programs
  - 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
  - Has moved to the upper-level curriculum in Fall 2016

[L. P. Carloni, The Case for Embedded Scalable Platforms, DAC 2016]

Teaching System-on-Chip Platforms at Columbia: The Fall-2015 Course Project in Numbers

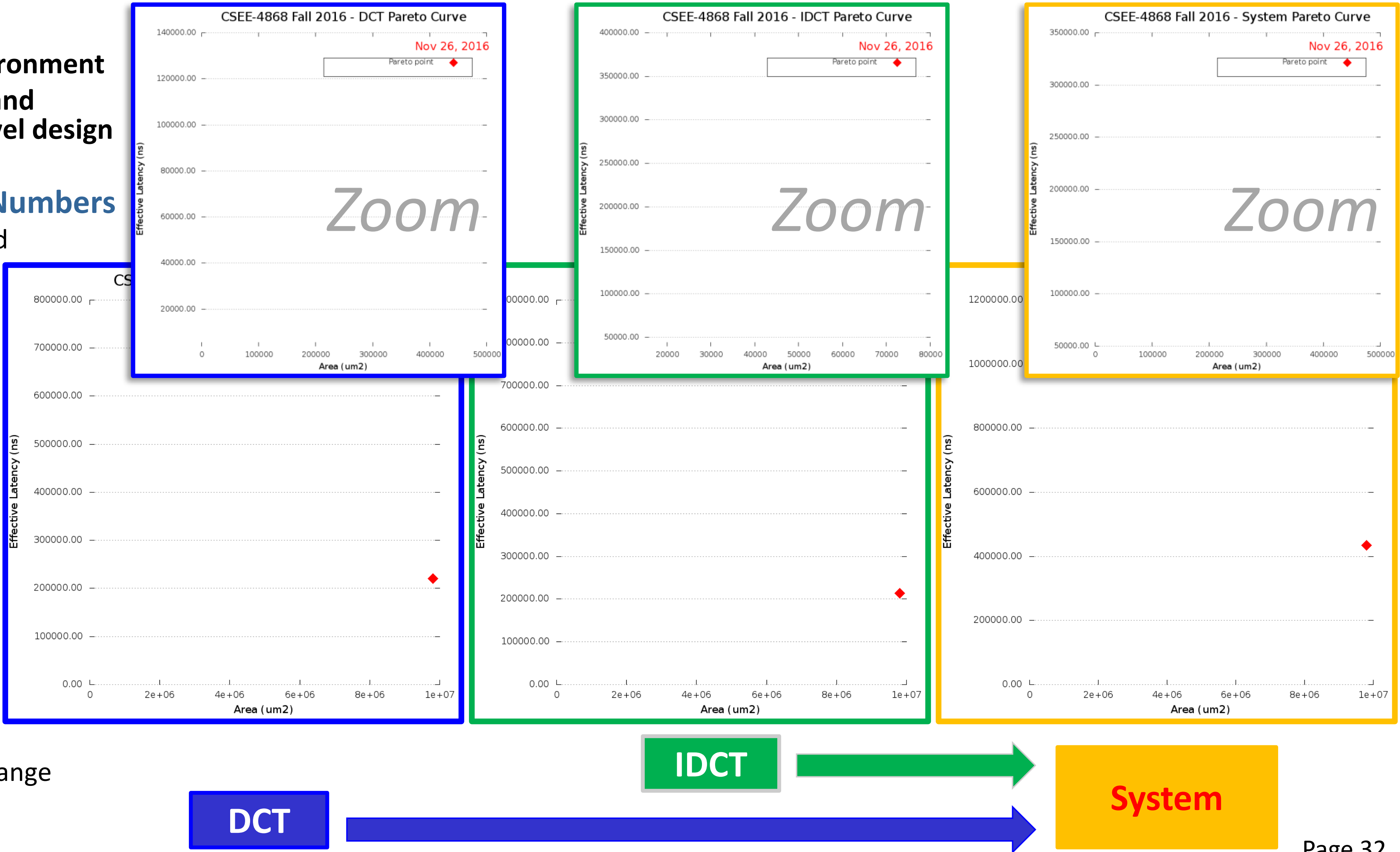
- At Columbia we developed the course '*CSEE-6868 System-on-Chip Platforms*' based on the ESP Design Methodology
- The Fall-2015 Project by Numbers
  - 21 student teams competed in designing a hardware accelerator for the WAMI Gradient kernel during a 1-month period
  - 661: Number of improved designs 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



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

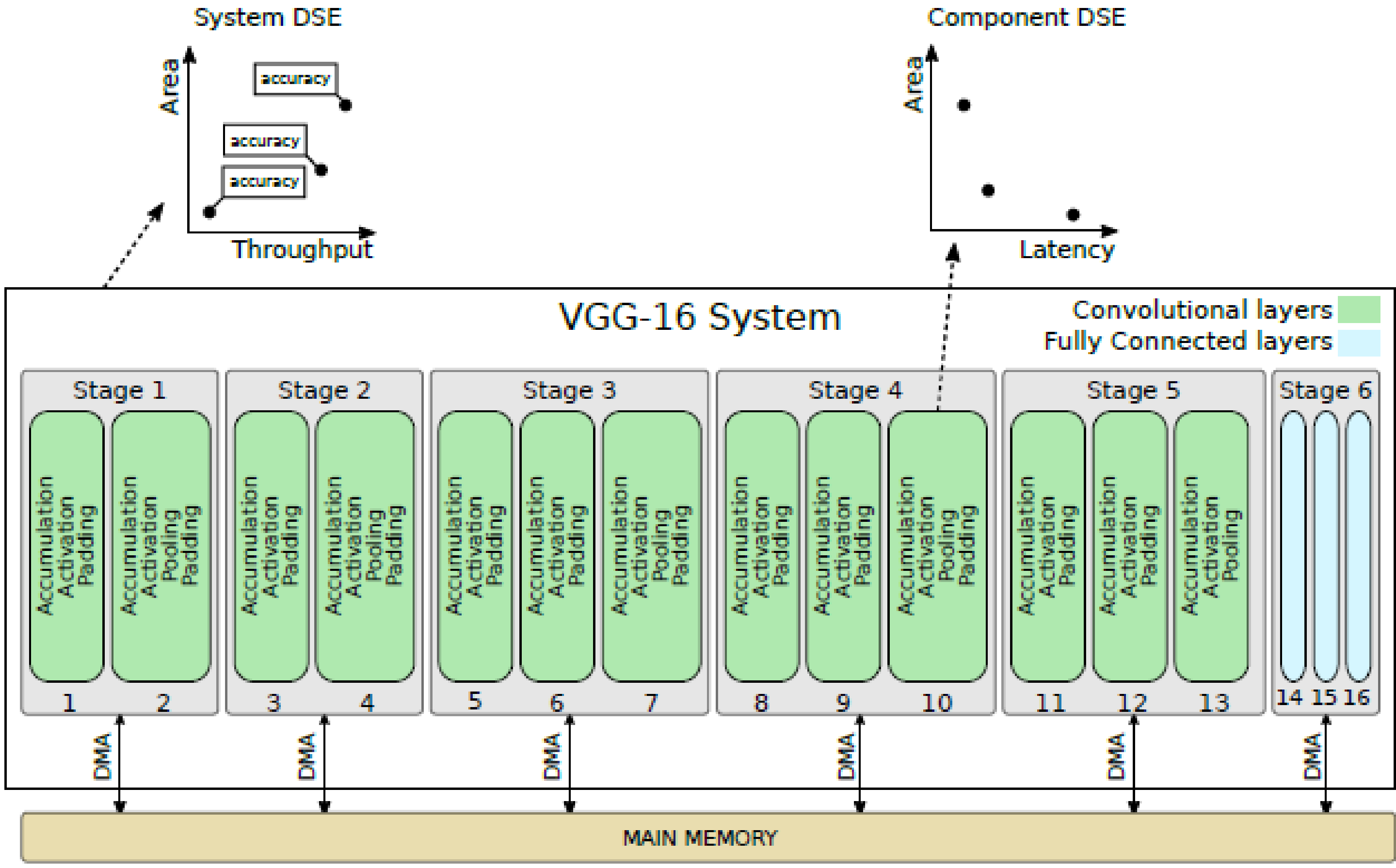
- **Fall-2016 New Features**
  - **Cloud-based project environment**
  - **Introduction of IP reuse and compositional system-level design**

- **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

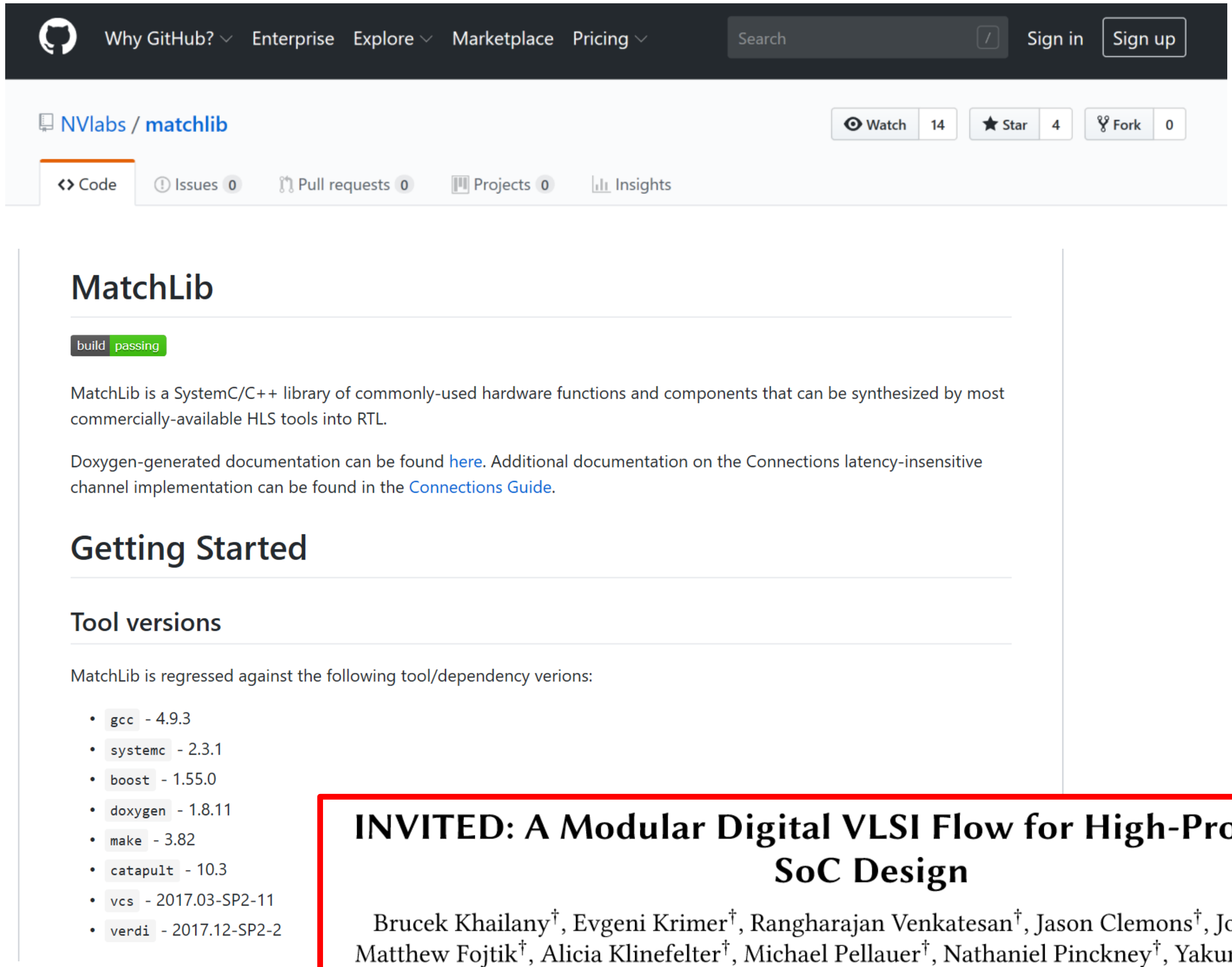


Keep Scaling Up the Design Complexity: The Fall-2017 Course Project

- ***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



Recent Industrial Advances: NVIDIA MatchLib



MatchLib

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.

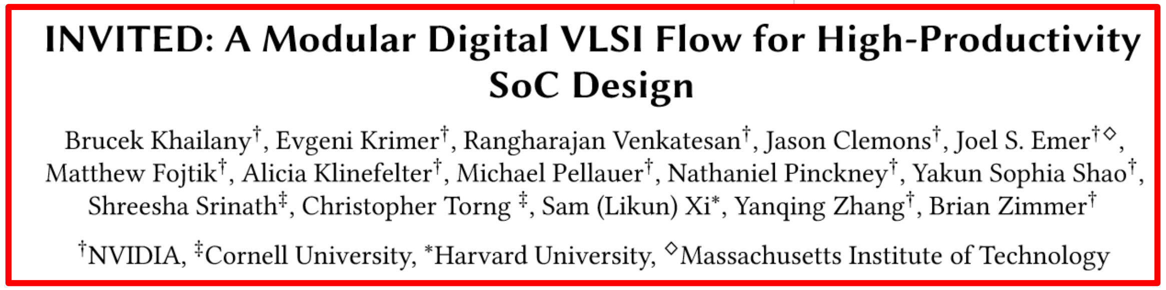
Doxygen-generated documentation can be found here. Additional documentation on the Connections latency-insensitive channel implementation can be found in the Connections Guide.

Getting Started

Tool versions

MatchLib is regressed against the following tool/dependency verions:

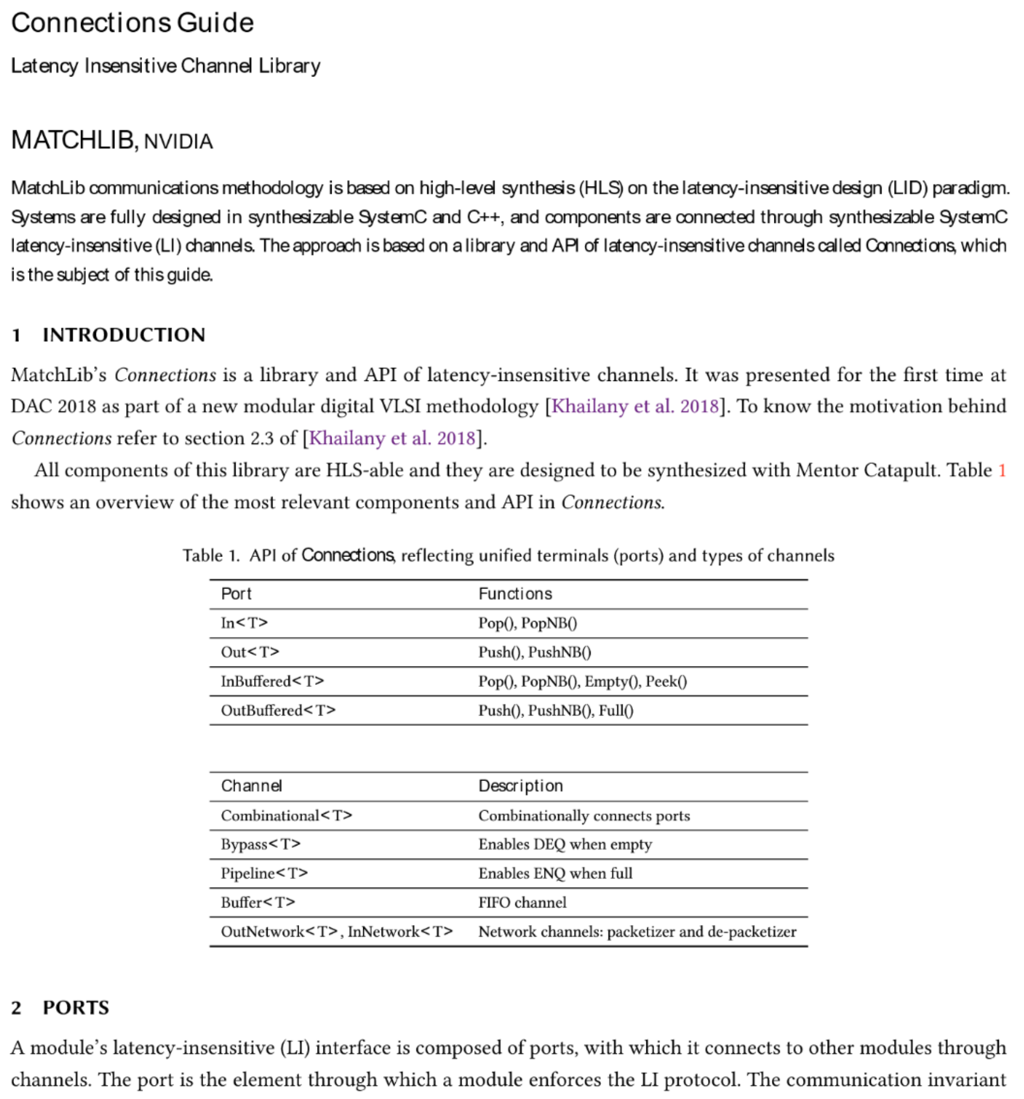
- `gcc` - 4.9.3
- `systemc` - 2.3.1
- `boost` - 1.55.0
- `doxygen` - 1.8.11
- `make` - 3.82
- `catapult` - 10.3
- `vcs` - 2017.03-SP2-11
- `verdi` - 2017.12-SP2-2



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

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

[†]NVIDIA, [‡]Cornell University, [*]Harvard University, [◇]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

Some Recent Publications

Available at www.cs.columbia.edu/~luca

1. L. P. Carloni. The Case for Embedded Scalable Platforms DAC 2016. (Invited Paper).
2. L. P. Carloni. From Latency-Insensitive Design to Communication-Based System-Level Design The Proceedings of the IEEE, Vol. 103, No. 11, November 2015.
3. E. Cota, P. Mantovani, and L. P. Carloni. Exploiting Private Local Memories to Reduce the Opportunity Cost of Accelerator Integration. ICS 2016.
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.