



**POLITECNICO**  
MILANO 1863

**Design of Hardware Accelerators**  
Academic Year 2021/2022

# Introduction to Heterogeneous Systems

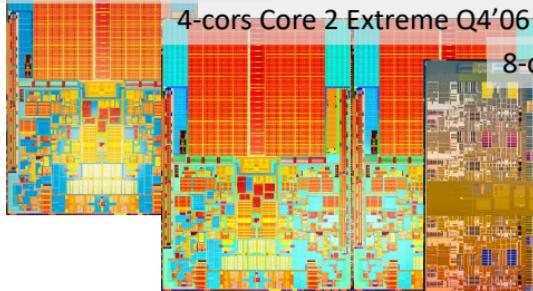
**Christian Pilato**

Dipartimento di Elettronica, Informazione e Bioingegneria

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)

# Increasing Number of Cores

2-cores Pentium D Q1'05  
2-cores Core Duo Q2'06

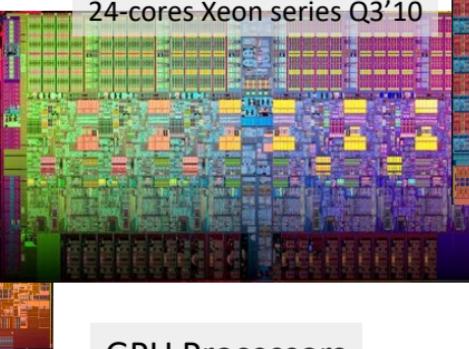


## General Purpose Processors

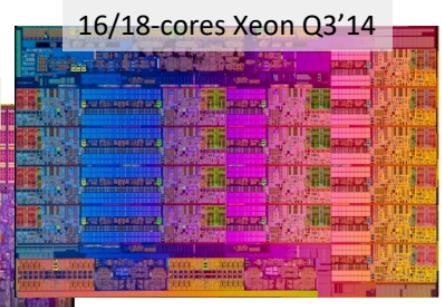
4-cores Core 2 Extreme Q4'06



8-cores Xeon Q1'10



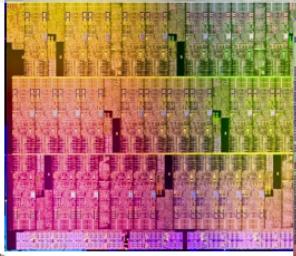
24-cores Xeon series Q3'10



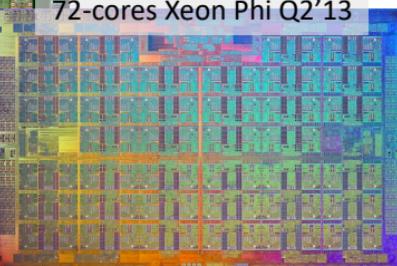
16/18-cores Xeon Q3'14

## Supercomputer Coprocessors

51/61-cores Xeon Phi Q2'13

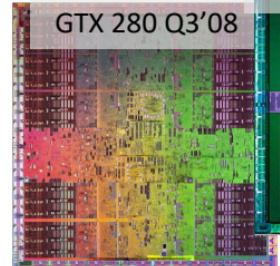


72-cores Xeon Phi Q2'13

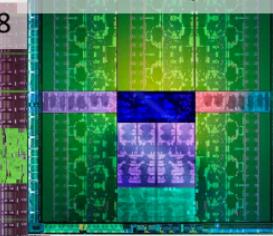


## GPU Processors

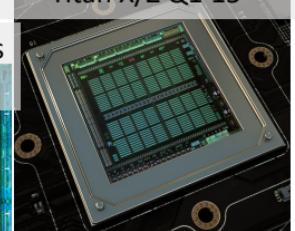
240-CUDA cores  
GTX 280 Q3'08



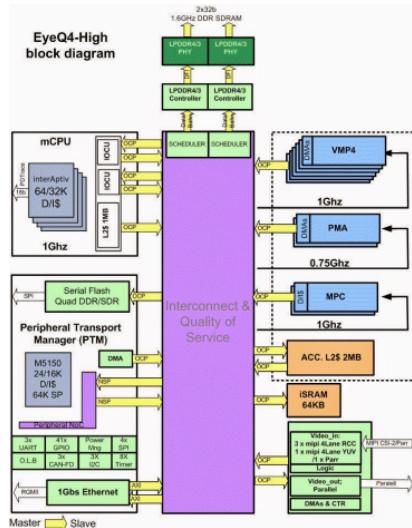
2x1536-CUDA cores  
GTX 690 Q2'12



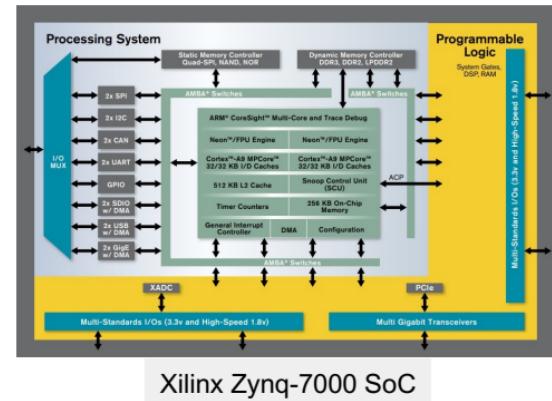
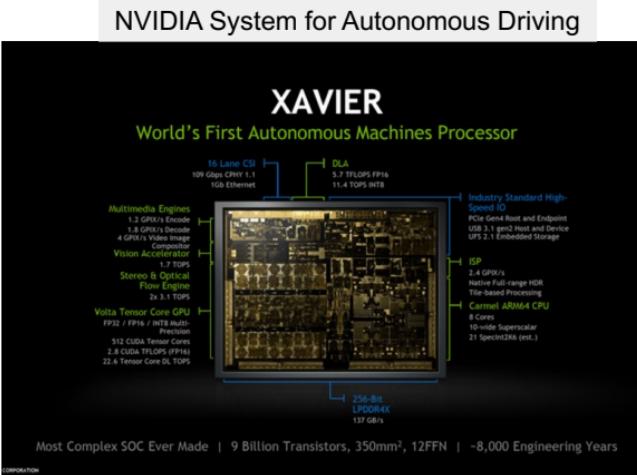
3072/5760-CUDA cores  
Titan X/Z Q1'15



# From Multicore to Heterogeneous Systems



Mobileye  
(US\$15.3 billion takeover by Intel)



Xilinx Zynq-7000 SoC

**Heterogeneous System:** system composed of several types of components

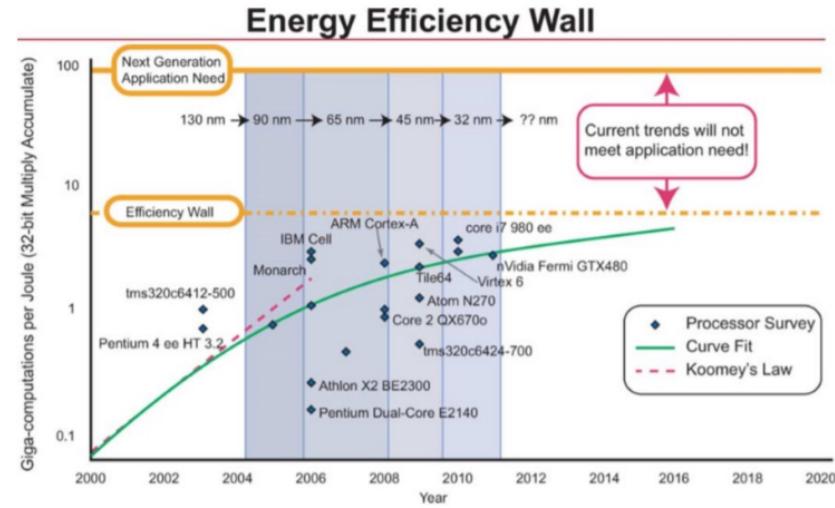
... but why?



# Moore's Law + Dennard Scaling

**Moore's Law:** "The number of transistors on an affordable CPU would double every two years" (G.E. Moore. 1976)

**Dennard scaling:** "If the transistor density doubles, the circuit becomes 40% faster\*, and power consumption (with 2x the number of transistors) stays the same" (R. H. Dennard, 1974)



[Marr et al. "Scaling Energy Per Operation via an Asynchronous Pipeline", TVLSI 2013]

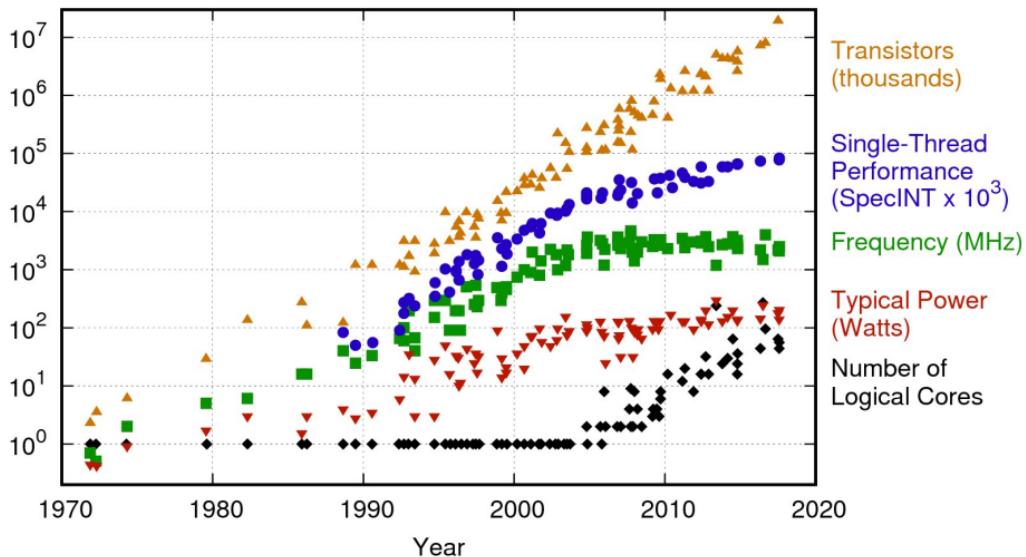
**Koomey's Law:** "at a fixed computing load, the amount of battery you need will fall by a factor of two every year and a half"

(\*Circuit delay is reduced)



# The Trend is not Infinite...

42 Years of Microprocessor Trend Data



20+ years of **CPU improvements** (pipeline stages, branch predictions, multicore, etc.), but we hit the **efficiency wall!**

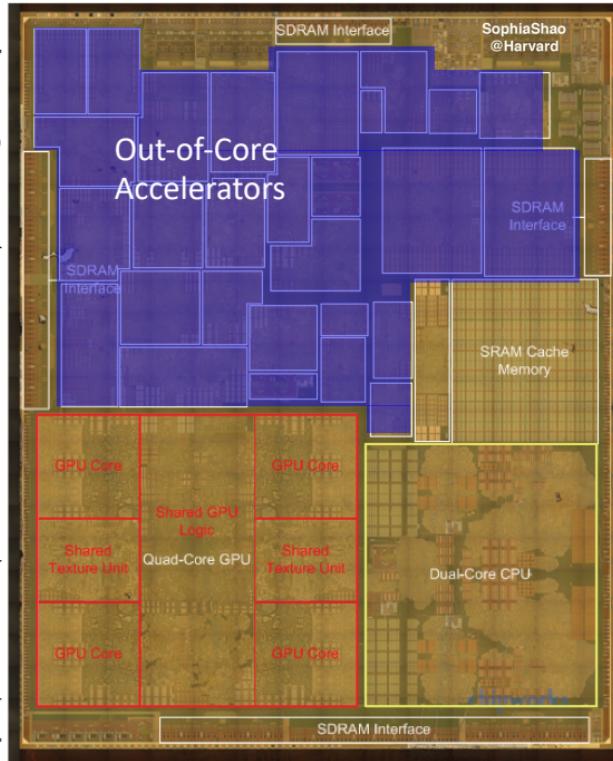
**Current leakage causes the chip to heat up!**

**“With each successive generation, the percentage of a chip that can actively switch drops exponentially due to power constraints”**



# Heterogeneous SoCs on the Market

[Die photo from Chipworks - Accelerators annotated by Y.S. Shao @ Harvard]



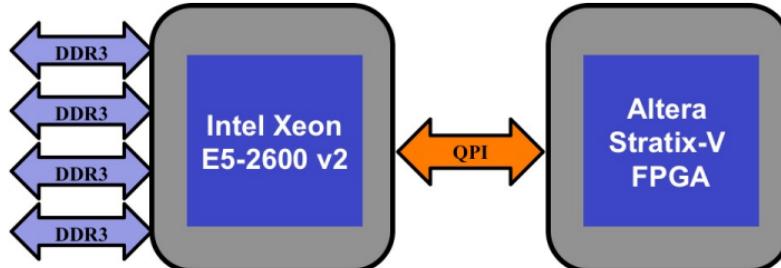
**iPhone 6 features the A8 SoC**

- 8.47 x 10.50 mm (20nm by TSMC)
  - **13% smaller** than A7 (28nm)
- dual-core ARM CPU at 1.40 GHz
  - **25% more CPU performance**
- four-cluster PowerVR GPU
  - **50% more graphics performance**
- 2 billions of transistors
  - **twice the number of transistors** compared to the A7
- almost 30 out-of-core accelerators
  - **50% more out-of-core accelerators** than A7 (~20 out-of-core accelerators)

# SoC is not Equal to Embedded System!

**HARP**: Heterogeneous platform with **Intel Xeon processor** (2.2 GHz) and coherently attached **Intel/Altera Stratix-V FPGA** (200 MHz)

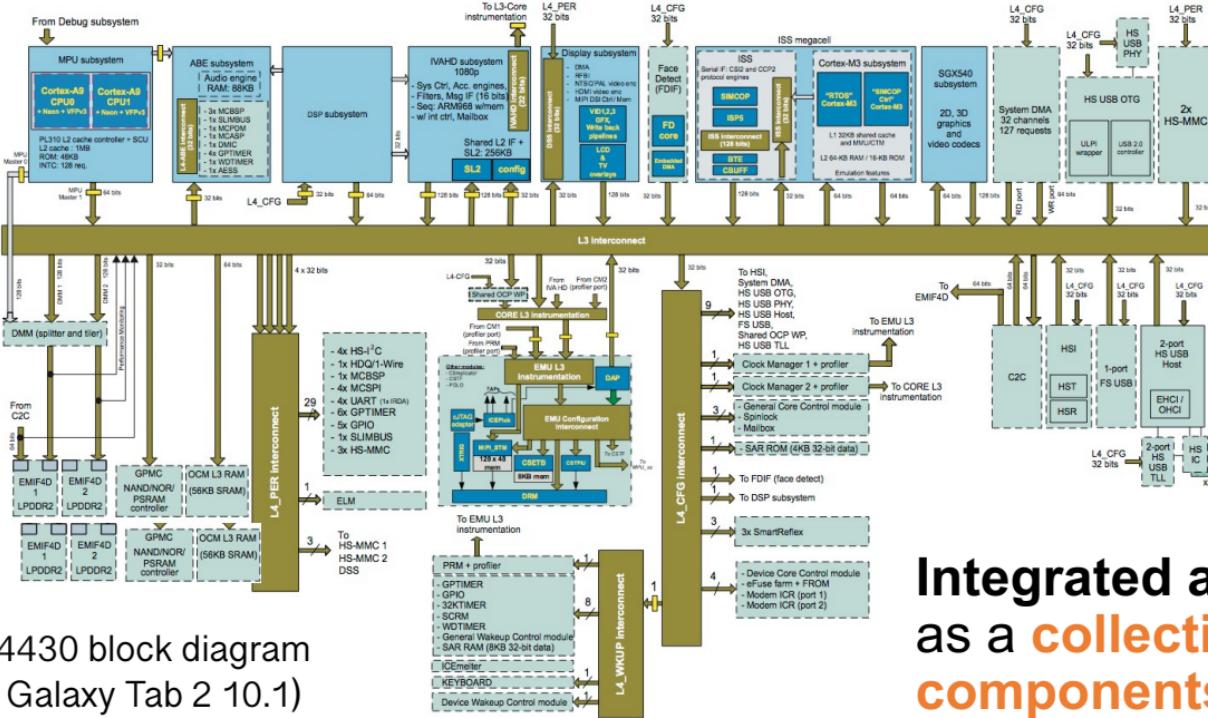
- 72 GB of DDR3 memory (possibility to store very large data sets)
- QPI interconnection link at 6.4 GT/s – low-latency from FPGA (~100 clock cycles per cache line at 200 MHz)
  - possibility of performing fast on-demand data accesses (off-chip)
- Software abstraction layer for application development and accelerator integration



**System-on-Chip** defines an architecture where the components are connected at the same physical level

# Today's SoC Architectures

Figure 1-2. OMAP4430 Block Diagram

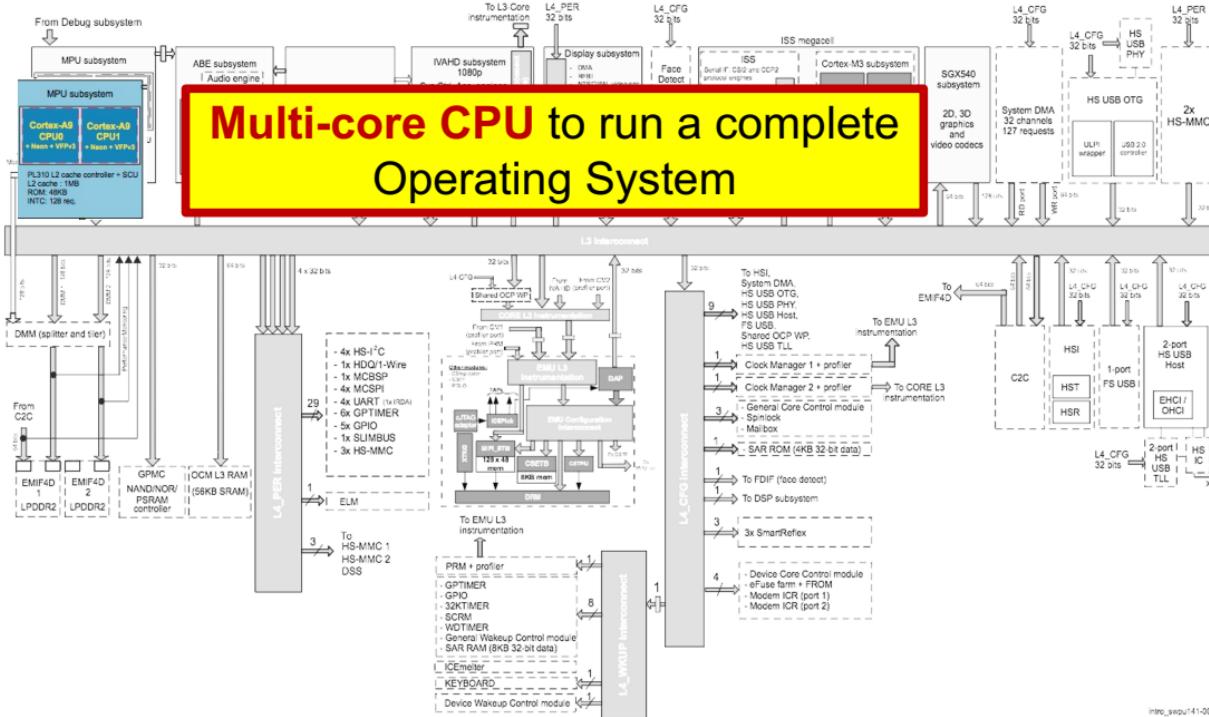


TI OMAP 4430 block diagram  
(Samsung Galaxy Tab 2 10.1)

Integrated architecture  
as a **collection of**  
**components**

# Today's SoC Architectures

Figure 1-2. OMAP4430 Block Diagram

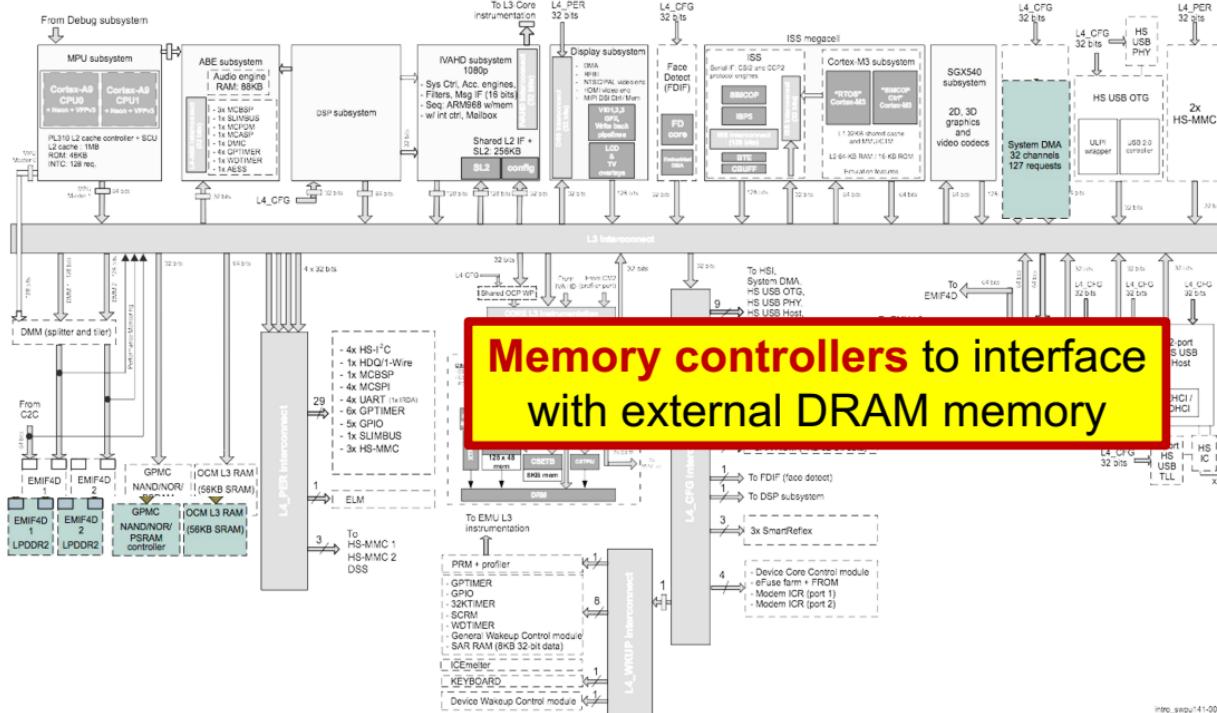


intro\_sequ141-001



# Today's SoC Architectures

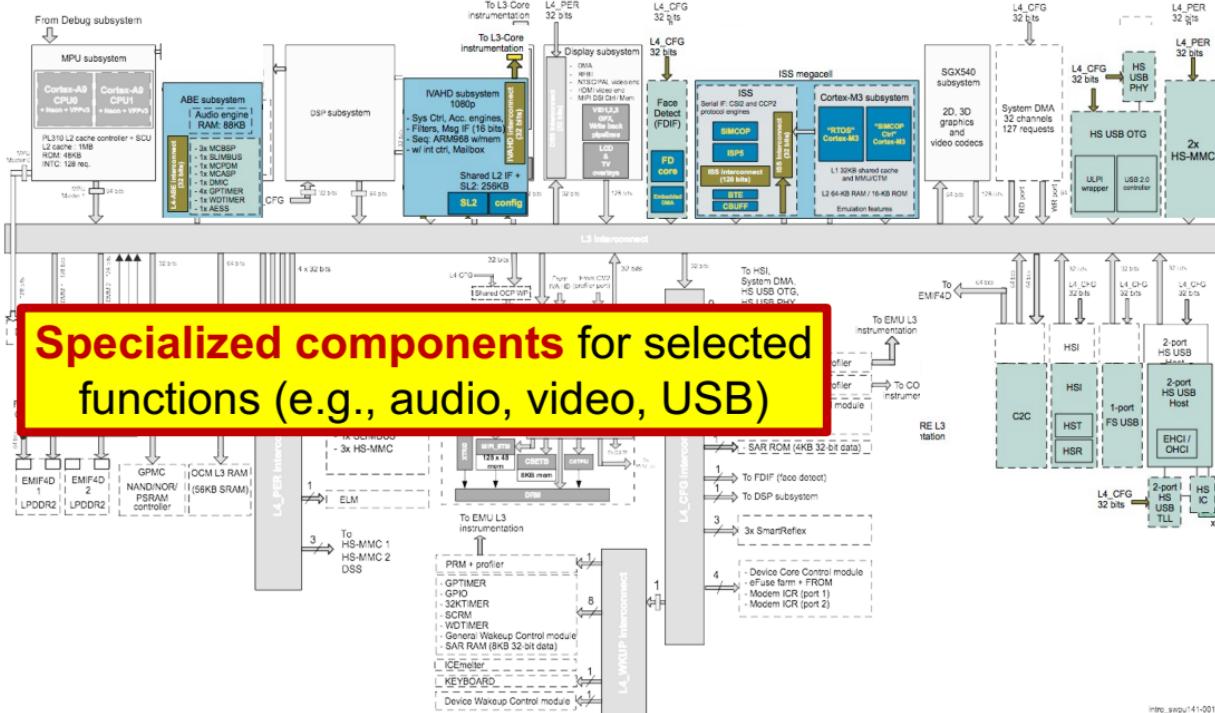
Figure 1-2. OMAP4430 Block Diagram



intro\_sequ141-001

# Today's SoC Architectures

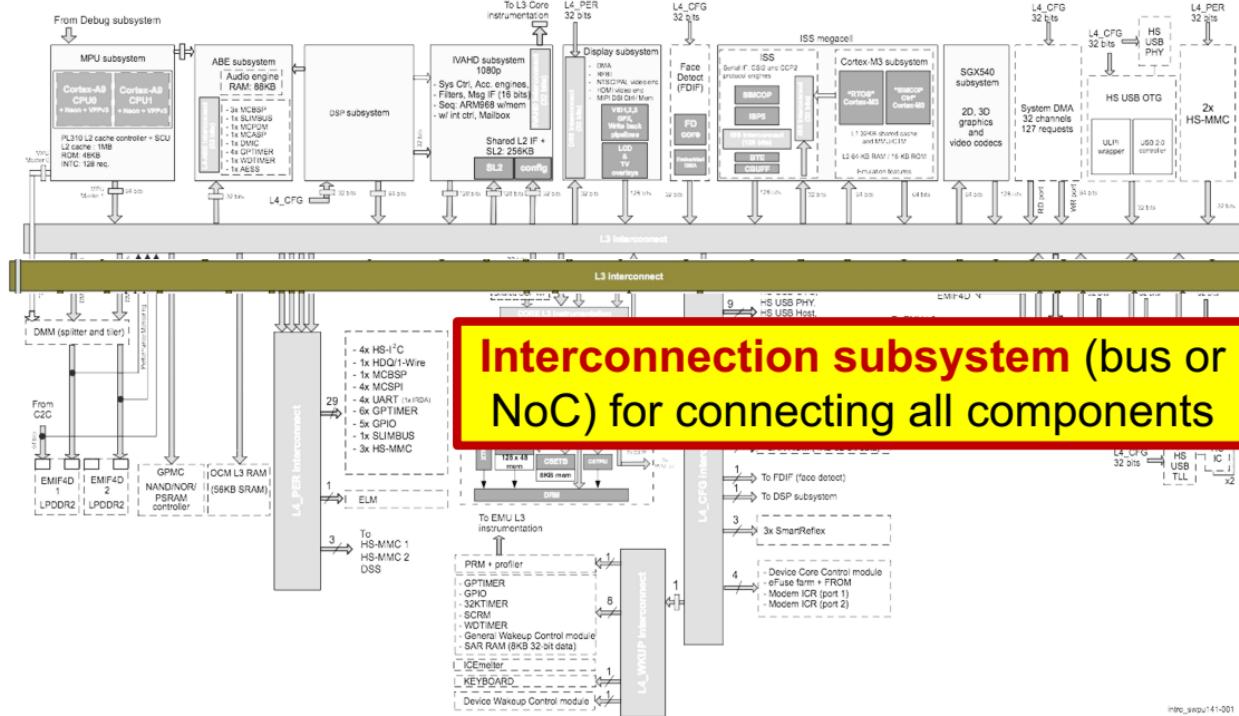
Figure 1-2. OMAP4430 Block Diagram



intro\_sequ/41-001

# Today's SoC Architectures

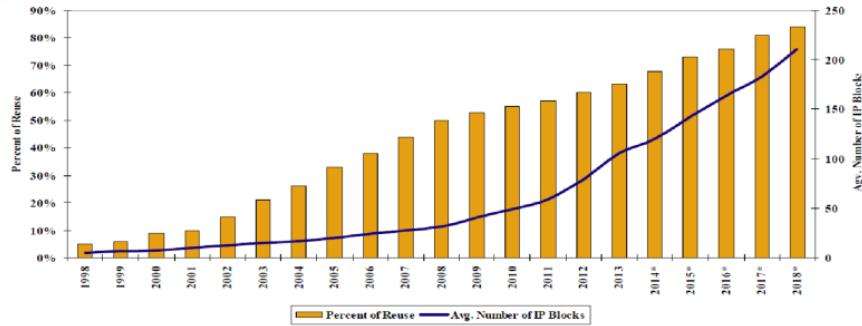
Figure 1-2. OMAP4430 Block Diagram



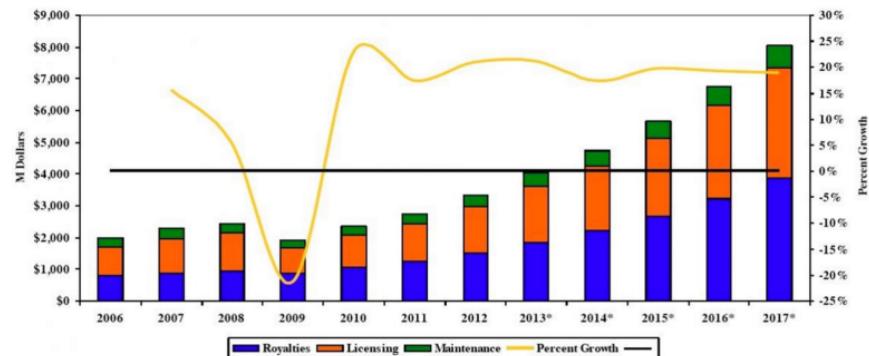
intro\_sequ/41-001



# Increasing Number of Integrated IPs



- Coping with **system complexity**
  - IP replication
  - IP reuse
- Estimated **60% reused IP blocks** per SoC (21x more IPs in less than 20 years)



- Design for reusability
  - Increased IP complexity
  - Increased IP cost
- Estimated **20% increased IP cost** every year

[SEMICO Research Corporation, reports from 2013/2014]

# Platform-Based Design

Predefined portion of the architecture (**platform template**)

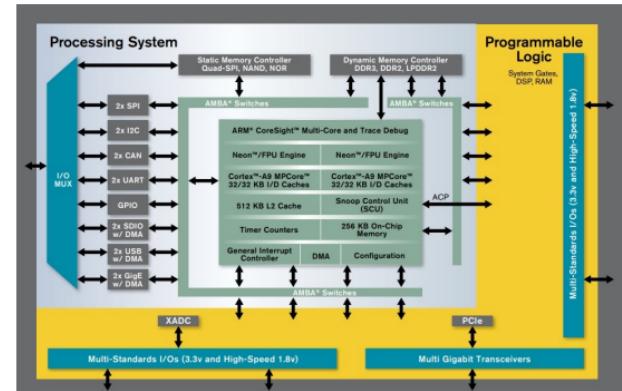
**General-purpose processor(s)**  
(with Operating System)

Interconnection infrastructure

Pre-defined IP blocks

Memory elements

**Customization:** adding hardware IP,  
programming FPGA logic or writing  
embedded software.



Xilinx Zynq-7000 SoC

Avoid redesigning each chip from scratch

# Processing Elements

**General-Purpose Processor(s)** are generic elements that can execute any functions, provided they are converted into microinstructions

- Used to run Operating System (OS)
- Necessary to coordinate hardware resources and manage I/O

**Accelerators** perform specific tasks more efficiently

- Programmable cores (GPU, specialized CPU)
- Configurable cores (specialized components)
- Dedicated cores

"more efficiently" does not necessarily mean "faster"

Processing Elements:  
collection of IP modules  
**(Processors and Coprocessors)**  
for executing functions



# Storage Elements

**On-chip:** temporary values stored for **direct/fast access**

**Cache:** component between main memory and component (transparent to execution)

**Scratchpad:** local memory programmed through software directives to move data

**Private Local Memory:** memory controlled by the component (not visible to the system)

**Off-chip:** large memories accessed through **memory controllers**

**Main Memory:** external memory that stores larger amounts of data

**SRAM/DRAM** are only types of technologies!

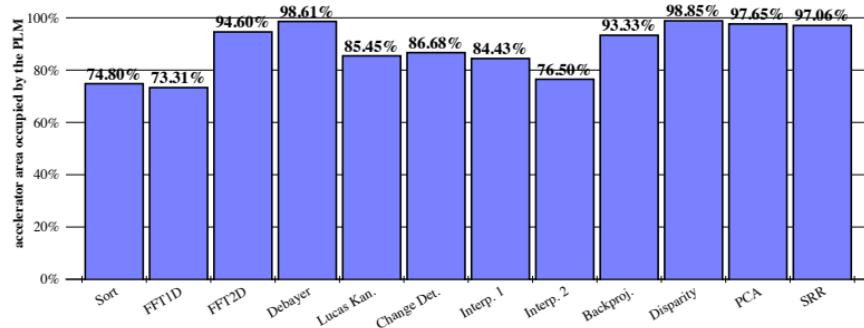
	Capacity	Latency	Cost/GB
Register	1000s of bits	20 ps	\$\$\$\$
SRAM	~10 KB-10 MB	1-10 ns	~\$1000
DRAM	~10 GB	80 ns	~\$10
Flash*	~100 GB	100 us	~\$1
Hard disk*	~1 TB	10 ms	~\$0.10

\* non-volatile (retains contents when powered off)



# Main Memory and PLM: A Huge Gap

Bench.	Main Mem Data Size (MB)	PLM Data Structures		Bench.	Main Mem Data Size (MB)	PLM Data Structures	
	(#)	(MB)			(#)	(MB)	
Sort	4.000	6	0.024	FFT1D	0.250	10	0.040
FFT2D	64.000	4	0.128	Debayer	16.000	4	0.096
Lucas Kan.	32.000	11	0.020	Change Det.	320.000	10	0.062
Interp. 1	32.040	6	0.048	Interp. 2	64.010	7	0.640
Backproj.	256.040	8	0.099	Dparity	15.820	11	0.146
PCA	20.190	3	0.117	SRR	4.760	21	0.076



Each accelerator is ~1mm<sup>2</sup>

- Comparable to Apple A8 accelerators

**PLM is from 75% to 98% of accelerator area**

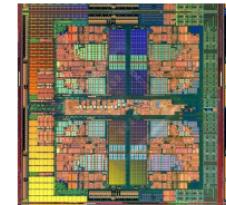
- With a lot of **data transfers**



# Memory Cost

**Memory leakage** is becoming more and more critical (<45nm)

- almost 70% of total power consumption
- SRAM leakage (caches and PLMs) contributes >75% to the total leakage



**DRAM and memory controllers**, as we know them today, are unlikely to satisfy all requirements

- Some emerging **non-volatile memory technologies** (e.g., PCM) enable new opportunities: **memory+storage merging**



Redesign of the memory hierarchy with application-centric approach



System programmability



# Interconnection System

**Bus:** a central crossbar responsible for arbitration between masters and slaves

**Network-on-chip:** packet switched network concepts between initiators and targets.

## Design time

- Abstraction to simplify reuse and integration of components

## Runtime

- Communication medium for energy-efficient exchange of massive data among cores
- Distributed mechanism to manage on-chip resources and control SoC operations

Must reach every chip corner with  
low latency and dissipation



# Relevant Research Projects

## Architectures



Embedded Scalable Platform (ESP)

*Columbia University*

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



Parallel Ultra-Low-Power (PULP)

*ETH Zurich & Univ. of Bologna*

<https://github.com/pulp-platform>

## Simulators



*AMD, ARM, HP, MIPS, Princeton, MIT, and the Universities of Michigan, Texas, and Wisconsin*  
<https://github.com/gem5>

<https://github.com/harvard-acc/gem5-aladdin>



# Embedded Scalable Platform (ESP)

## Architecture and Design Methodology

### • Regularity

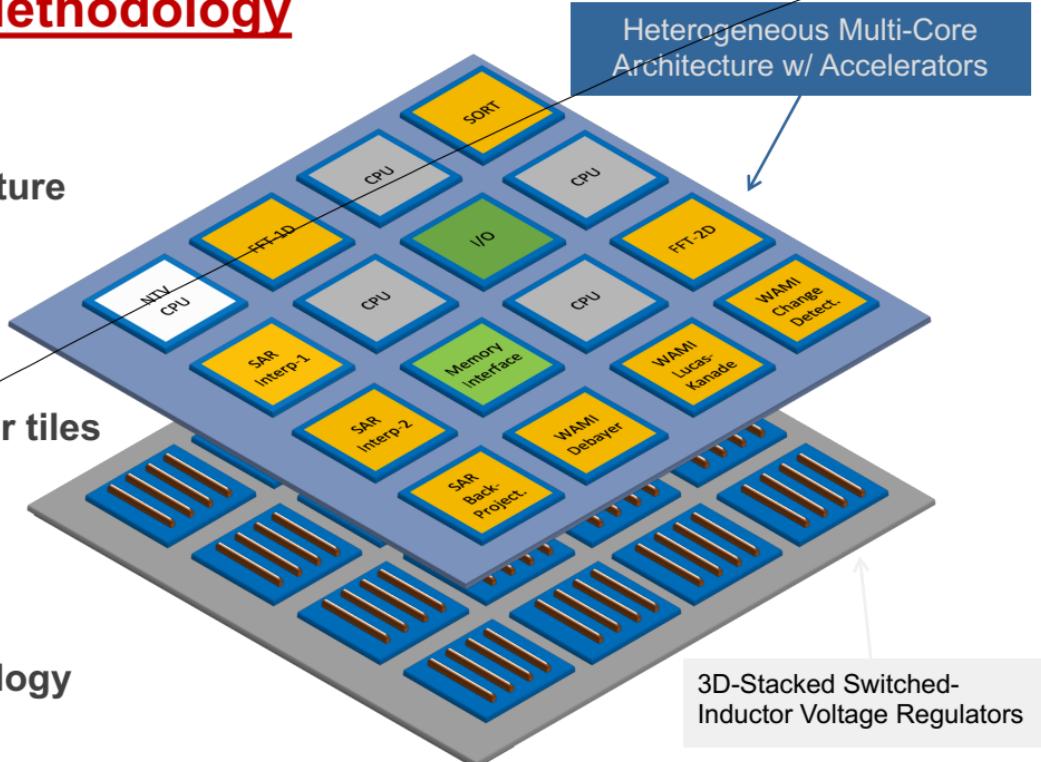
- tile-based design
- pre-designed **on-chip infrastructure** for communication and resource management

### • Flexibility

- each ESP design is the result of a **configurable mix of processor tiles and accelerator tiles**

### • Scalability

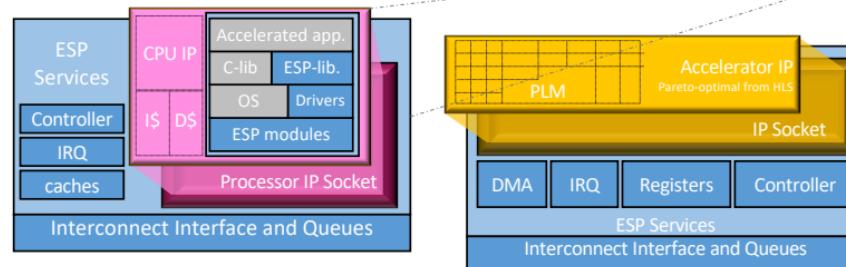
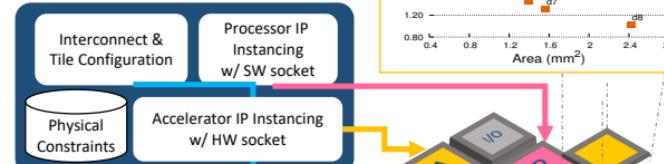
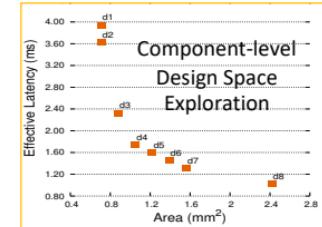
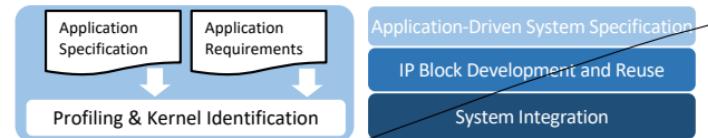
- at run-time via **fine-grain power management**
- at design-time via **ESP methodology**



# The ESP Design Methodology

## Speed, Flexibility, Reuse

- HLS-Based Design
- Component- and Application-Level DSE
- Automatic SoC Generation
  - “Socket-based” IP integration
  - Interconnect configuration
  - System memory mapping
- Software Layer Templates
  - ESP accelerator device driver



# Parallel Ultra-Low-Power (PULP) Platform

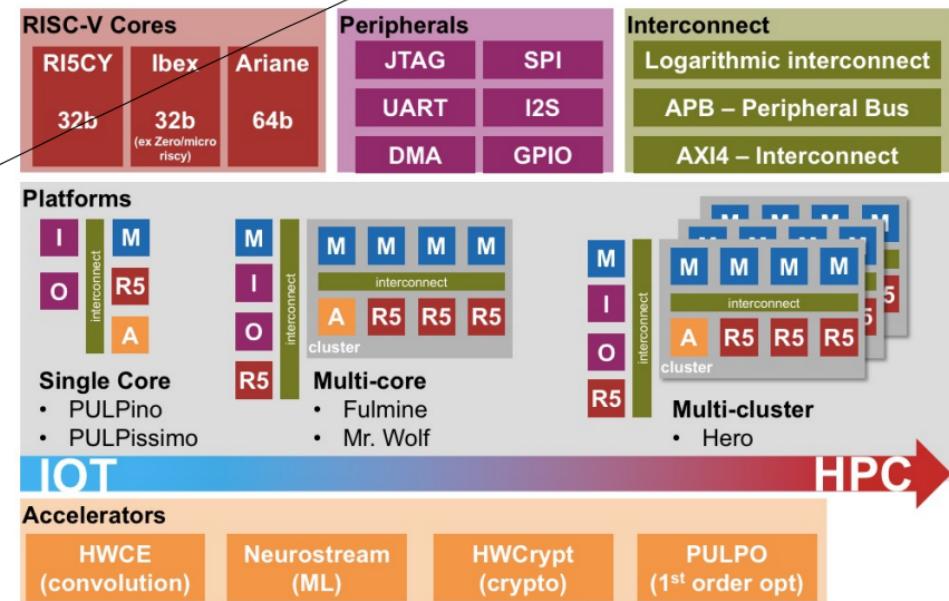
Ultra-low power system based on RISC-V

- **Efficiency** with multi-core clusters or custom accelerators
- **Silicon-proven architecture**
  - More than 20 certified chips

Complete platform integrating several research projects

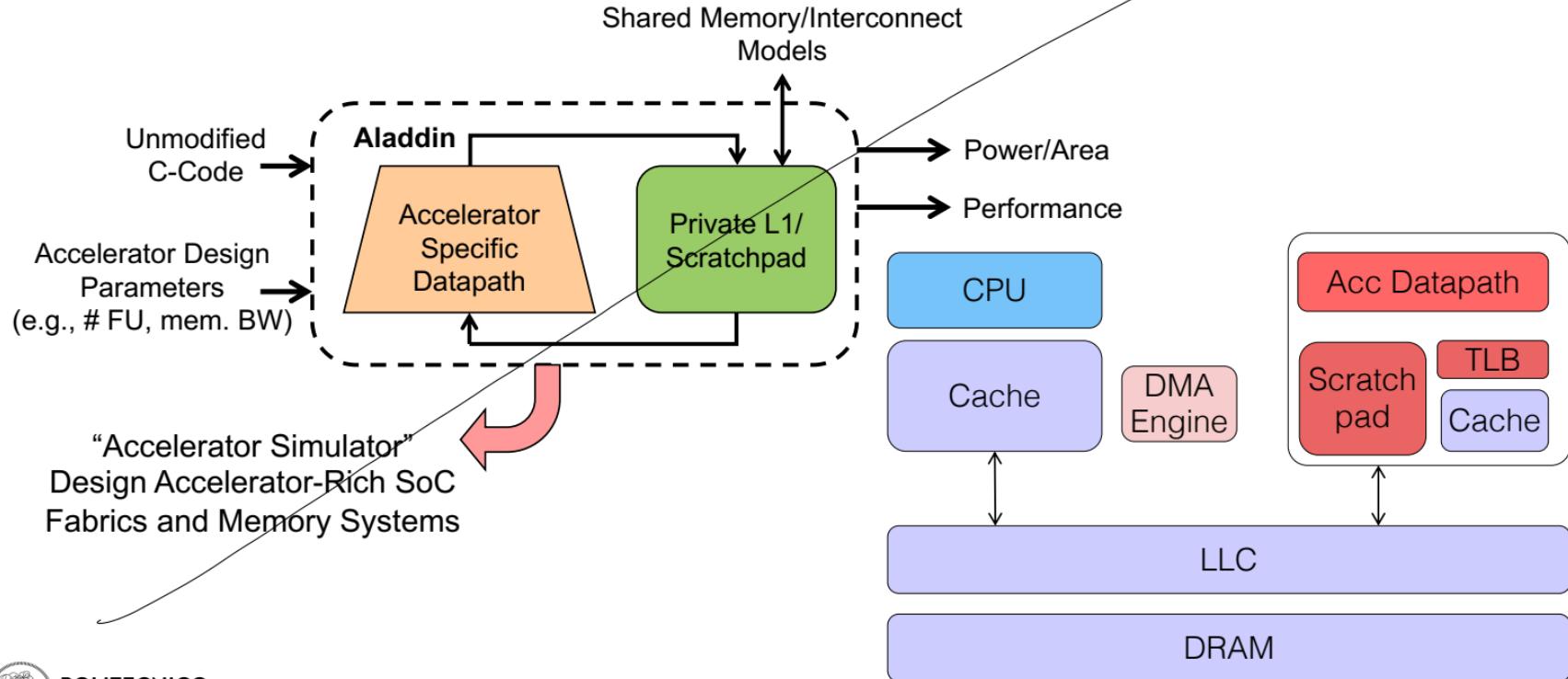
- Very productive community
- (Almost) everything is **open-source**

First complete  
**open hardware project**



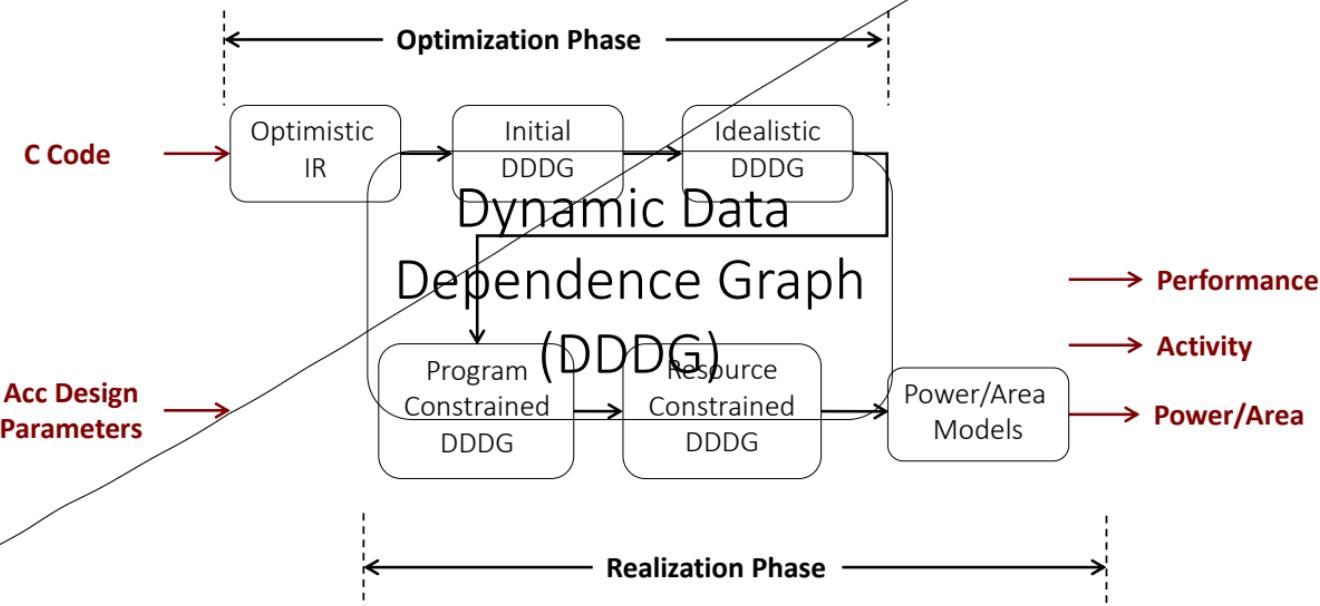
# Aladdin: A pre-RTL Accelerator Simulator

Power and performance estimation from C code



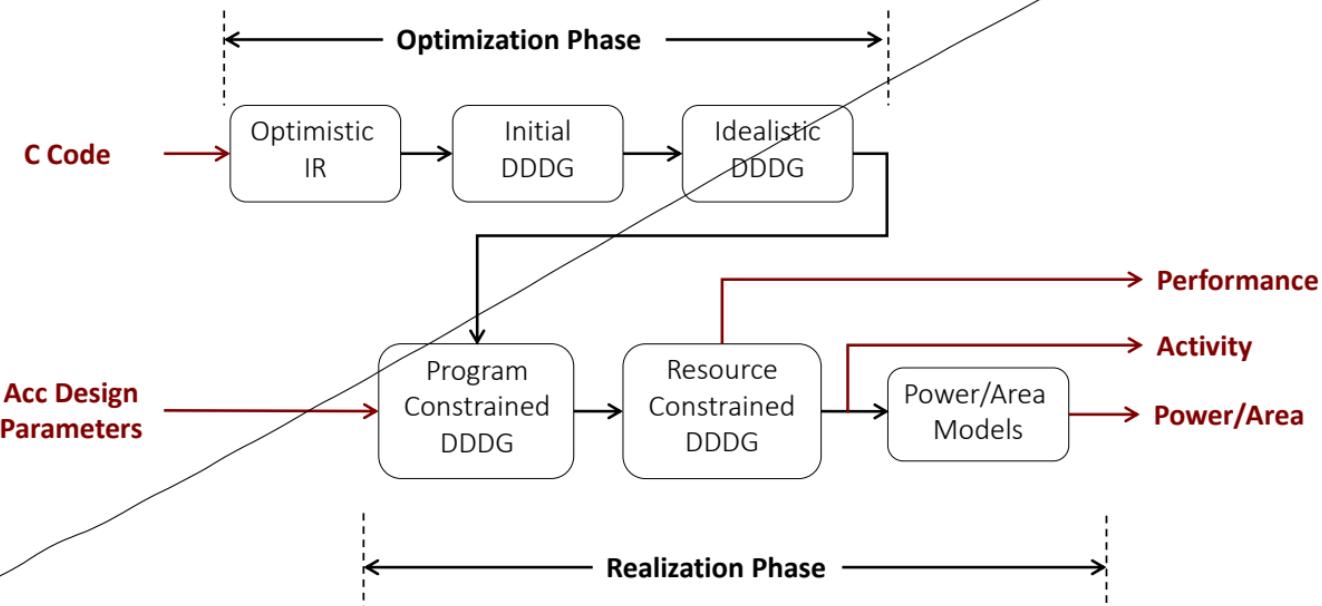
# Aladdin Overview

Analysis based on an **LLVM-based intermediate representation**



# Aladdin Overview

Analysis based on an **LLVM-based intermediate representation**



# **Design of Hardware Accelerators**

Academic Year 2021/2022

## Questions?

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)



**POLITECNICO**  
MILANO 1863

**Design of Hardware Accelerators**  
Academic Year 2021/2022

# Introduction to SoC Components

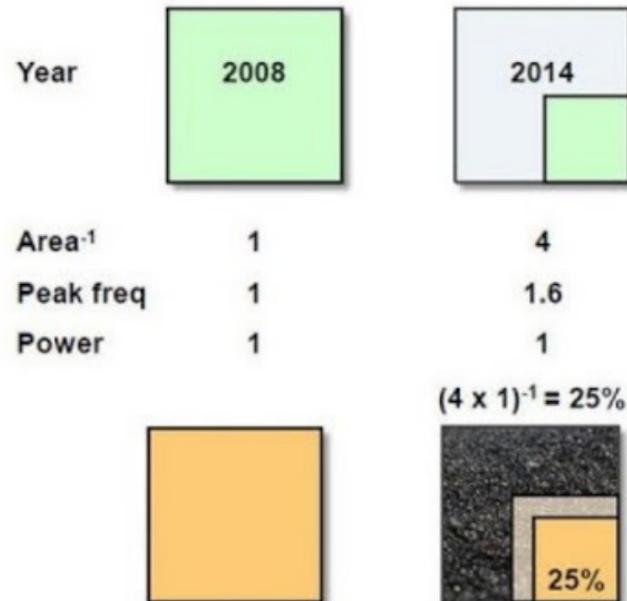
**Christian Pilato**

Dipartimento di Elettronica, Informazione e Bioingegneria

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)

# Dark Silicon and Specialization

- **Dark silicon** problems are progressively **reducing the fraction of the chip that can be active**
- **Specialization** allows designers to identify **components** that can be **turned off** when inactive
  - Requires the definition of a **scalable architecture** and a **communication infrastructure**
  - Requires also an **efficient power management system**



# Typical SoC Components

- General-Purpose Processors
- Memories (on- and off-chip)
- Co-processors (Accelerators)
- Communication Infrastructure



# Why Do We Need Processors in SoCs?

Hardwired systems are **inflexible**  
Lots of work to re-wire, or re-toggle

General purpose hardware can do different tasks  
Instead of re-wiring, supply **a new set of control signals**

Many applications share the **same (critical) kernels**  
**Specialization** is applied only to those kernels  
GPP executes the rest of the code

The diagram consists of four main text blocks arranged vertically. The first two are in light gray boxes, and the last two are in darker gray boxes. Red arrows point from the top two blocks down to the bottom two. A red double-headed arrow connects the middle two blocks.

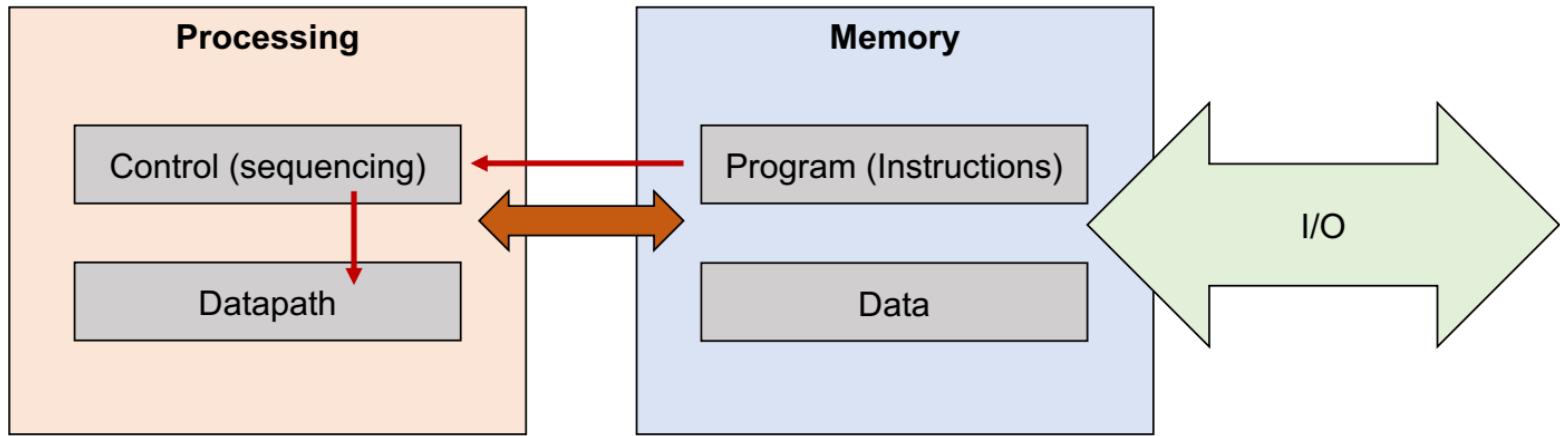
GPP allows for the **execution of different applications**

**SoCs**, not custom chips!



# What is a Processor?

Also called **stored program computer**



Instructions (sequence of control signals) in memory  
**sequential instruction processing**



# From Hardware to Software

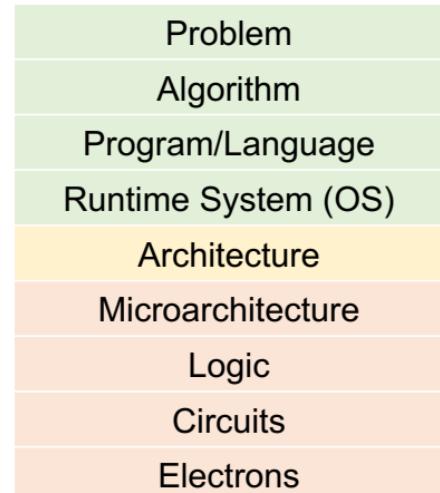
**Software** is a sequence of steps

- For each step, an arithmetic or logical operation is done
- For each operation, different control signals are needed – i.e., an instruction

**Hardware** is the physical implementation of the processing logic

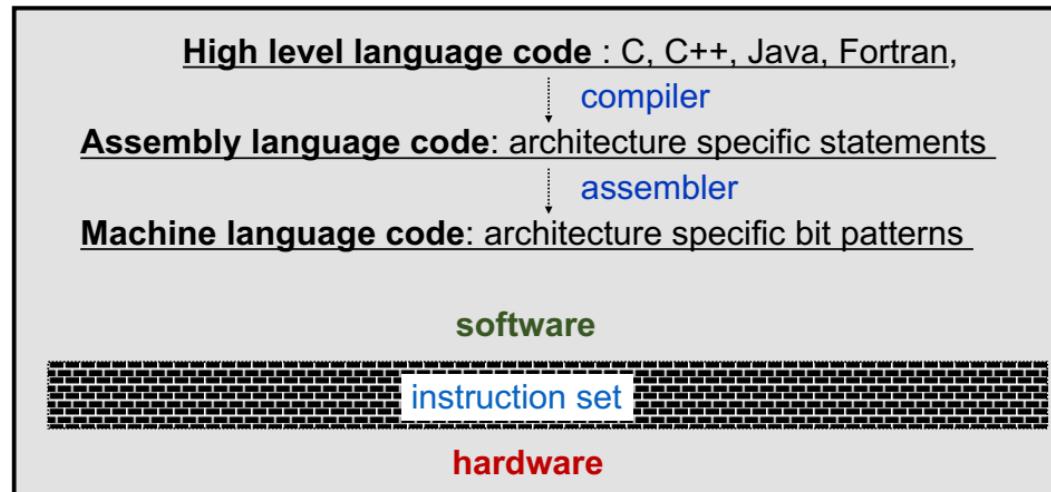
- The **processor microarchitecture** is (usually) a **proprietary design**

**(Instruction Set) Architecture**  
is used as an interface



# Instruction Set Architecture (ISA)

- Complete **collection of instructions** understood by a CPU
- Serves as an **interface** between software and hardware
- Provides a mechanism by which the software **tells the hardware what should be done**



# ISA - Processor Microarchitecture

**Instruction Set Architecture (ISA)** specifies how the programmer sees instructions to be executed (**programmer visible instruction set**)

- It defines how to specify the commands to the hardware logic

Often the ISA is identified with the  
**processor architecture**

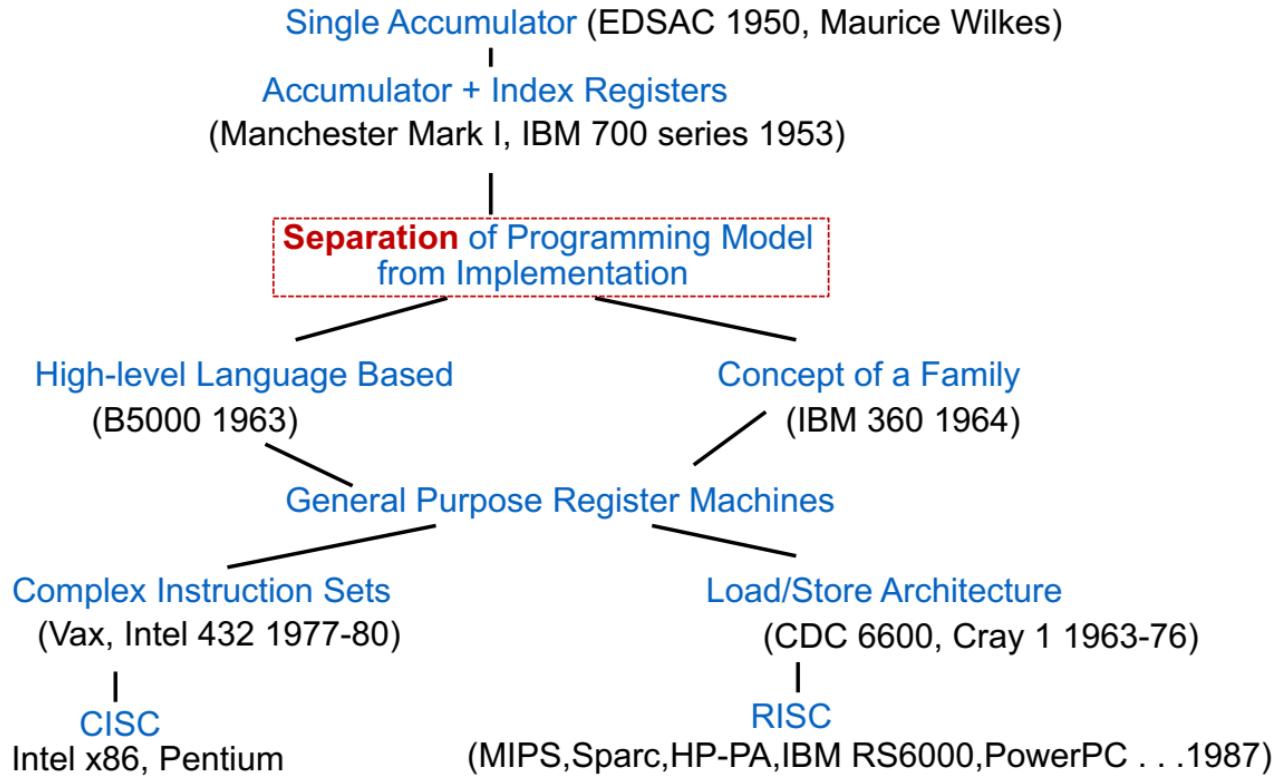
**Processor Microarchitecture** refers to the **internal organization of the processor**

- How the underlying implementation executes instructions
- So, several specific processors with differing microarchitectures may share the same architecture, i.e., the same ISA

**Consistency models:** programmers must see the order specified by ISA

- Microarchitecture can execute instructions in any order as long as it obeys the semantics specified by the ISA when making the instruction visible to software

# A Bit of History in Computer Programming



# CISC vs. RISC

## Complex Instruction Set Computer (CISC) [e.g., Intel x86]

> 1000 instructions, 1 to 15 bytes each

operands in dedicated/general-purpose registers memory, on stack, ...  
(can be 1, 2, 4, 8 bytes, signed or unsigned)

10s of addressing modes (e.g.  $\text{MEM}[\text{segment} + \text{reg} + \text{reg}^*\text{scale} + \text{offset}]$ )

About **80% of the computations** of a typical program required only about **20% of the instructions** in a processor's instruction set

## Reduced Instruction Set Computer (RISC) [e.g., MIPS]

≈ 200 instructions, 32 bits each, 3 formats

all operands in registers (almost all are 32 bits each)

≈ 1 addressing mode:  $\text{MEM}[\text{reg} + \text{imm}]$



# What is Better?

## Complex Instruction Set Computer (CISC) [e.g., Intel x86]

More operands and more complex (powerful?) instructions

More registers (Inter-register operations are quicker)

Fewer instructions per program (less memory)

## Reduced Instruction Set Computer (RISC) [e.g., MIPS]

Fewer operands and less complex (efficient?) instructions

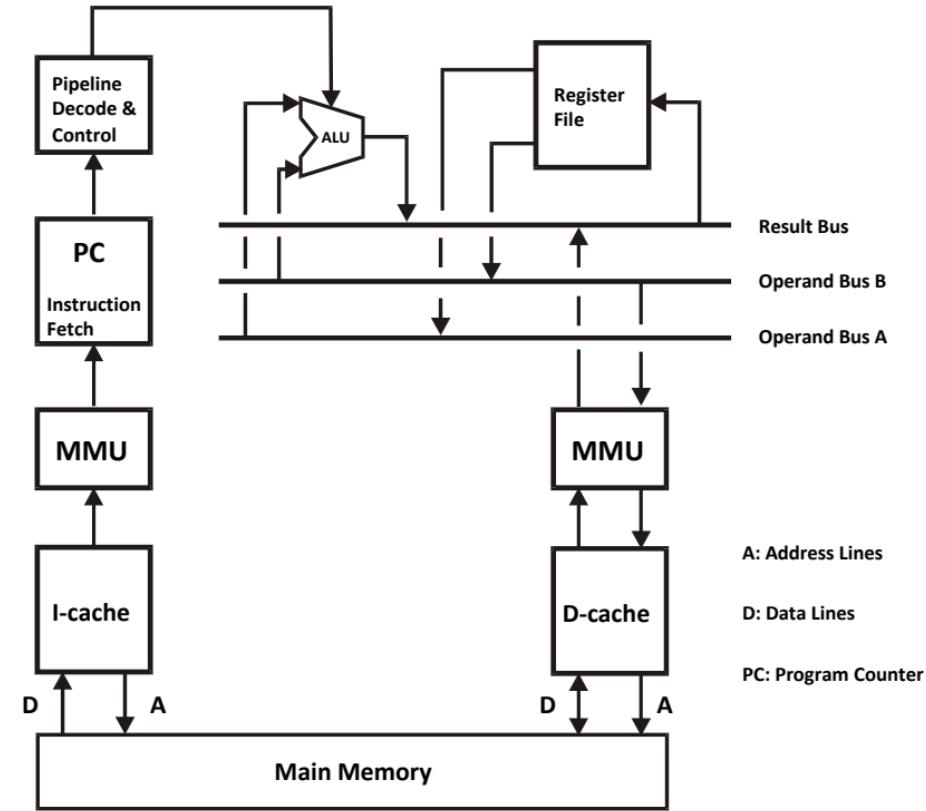
Faster fetch/execution of instructions

More instructions per program



# Microarchitecture of a Simple RISC CPU

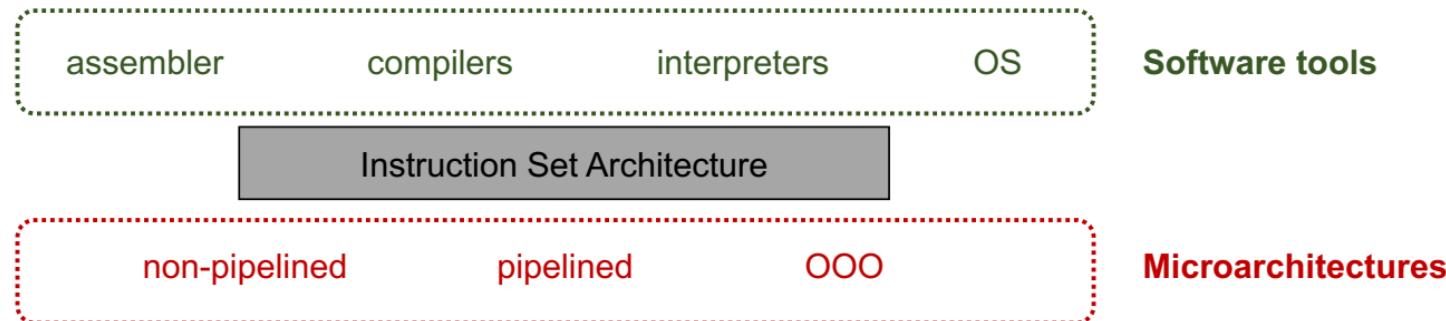
- Possibility to implement several optimizations **transparent to software execution**
  - E.g., pipelining w/- forwarding
- Simple (?) ALU** with basic arithmetic/logic operations
- Multiple execution stages** for an operation (**pipelining**)
  - Decode** stage decouples ISA from microarchitecture



# Instruction Set Architecture

ISA is a set of **instruction models**

- Each instruction defines **a way to transform the machine state**
- ISA is also **a «contract»** that an architect must follow **when designing a machine**



Is it possible to apply  
**CPU specialization? How?**



# Instruction Set Extensions

- Used for implementing **complex operations** not defined by the basic microarchitecture
  - Coprocessors **inside the CPU microarchitecture** (e.g., tightly-coupled accelerators)
  - Require an **extension of the compiler** and a **modification to the microarchitecture** to use them

Most ISAs (X86, ARM, Power, MIPS, SPARC) are **fixed and proprietary**

- Preventing practical efforts to reproduce the computer systems (**patents**)
- **Impossible to add special instructions** and apply **specialization with tightly-coupled accelerators**



# RISC-V

- Popular open-source ISA supported by many vendors

**Alibaba** releases its first RISC-V CPU as open source solution for 5G and AI [July 26, 2019]

**AWS** Announces RISC-V Support in the FreeRTOS Kernel [February 26, 2019]

**GreenWaves Technologies** Named 2019 Cool Vendor in AI Semiconductors [April 29, 2019]

- Microarchitecture must implement the given specification
  - No patent that would be required to implement a RISC-V-compatible processor
  - Low barrier to enter into (custom) processor design
- The RISC-V ISA was designed for research, and therefore includes **extra space for new instructions**
  - Possible (and encouraged) to provide **extensions for specific functions**

Enabling technology for  
**Custom SoC Design**



# RISC-V Popularity

2017



**RISC-V** Foundation: 65+ Members

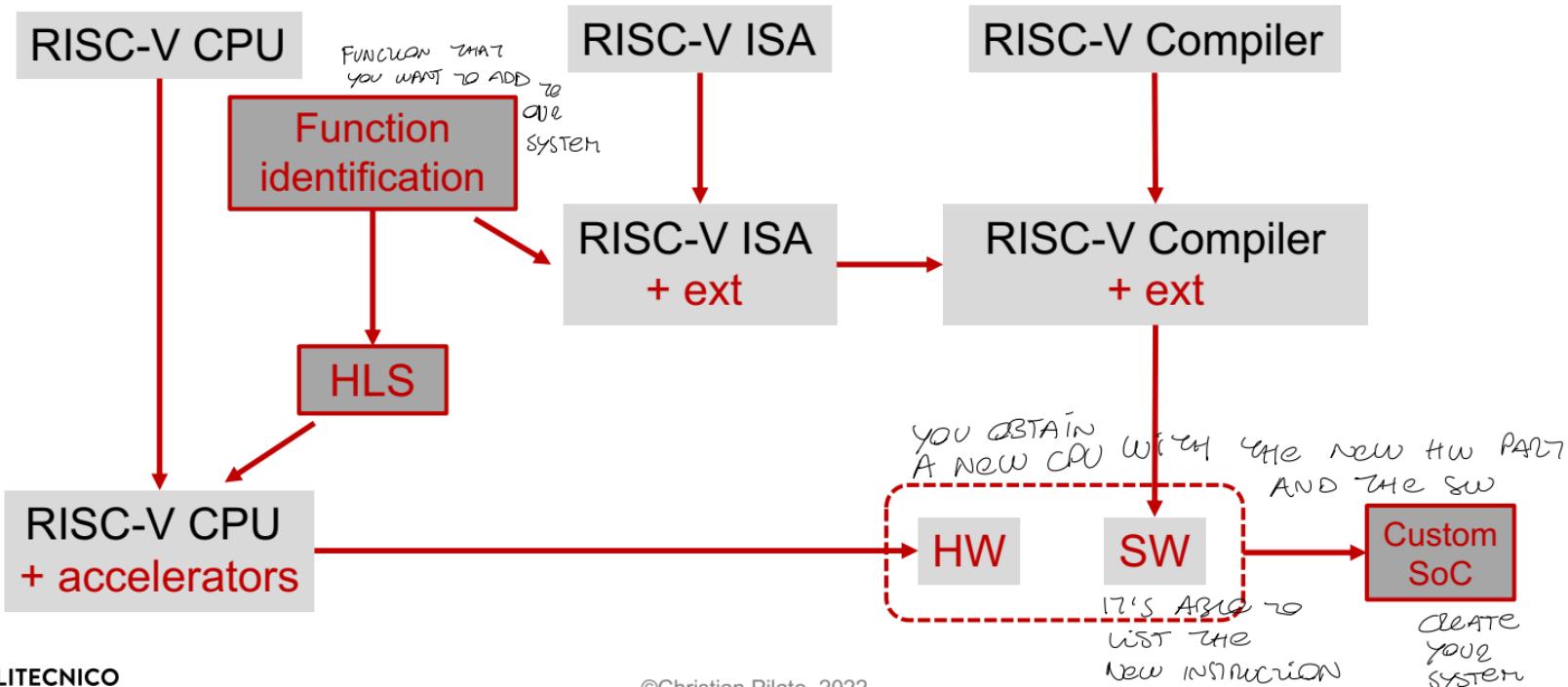


2019



# Custom SoCs with RISC-V

- RISC-V is increasingly supported by many open-source compilers



# CPU and Memory Hierarchy

## Ideal Memory:

Zero access time (latency)

Infinite capacity

Zero cost

Infinite bandwidth (for parallel accesses)

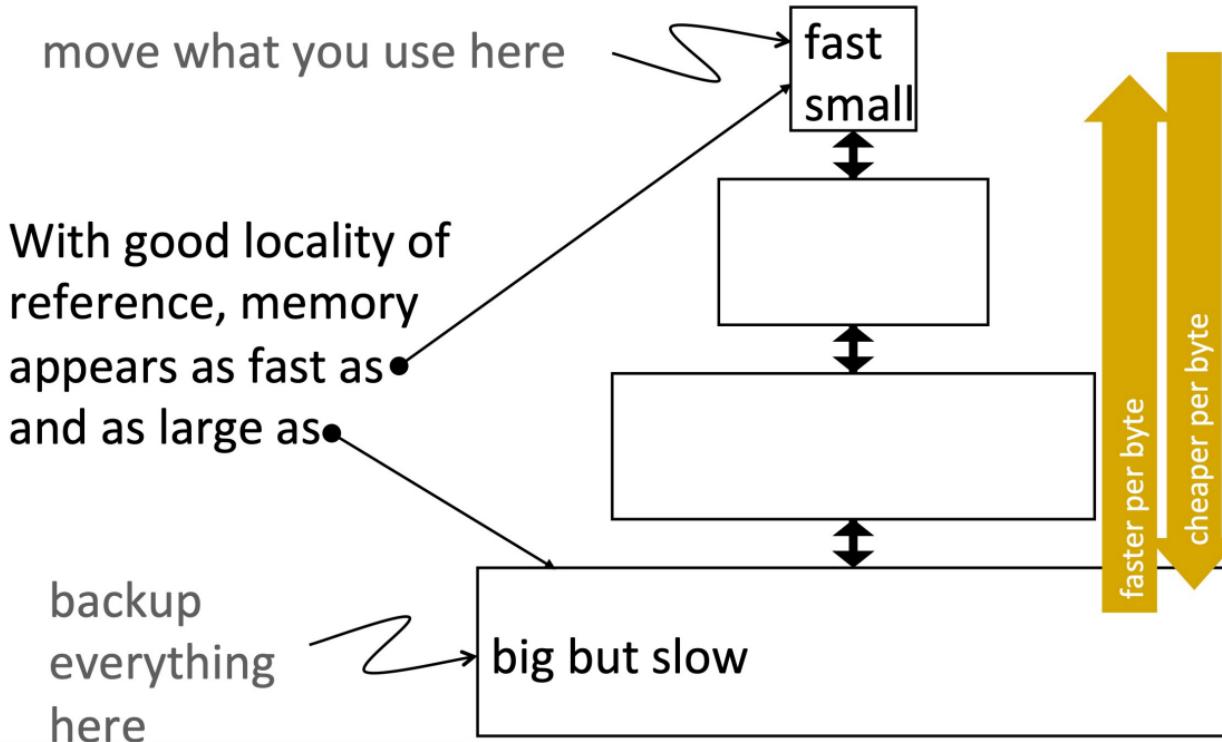
## Conflicting requirements!

**Idea:** Have multiple levels of storage

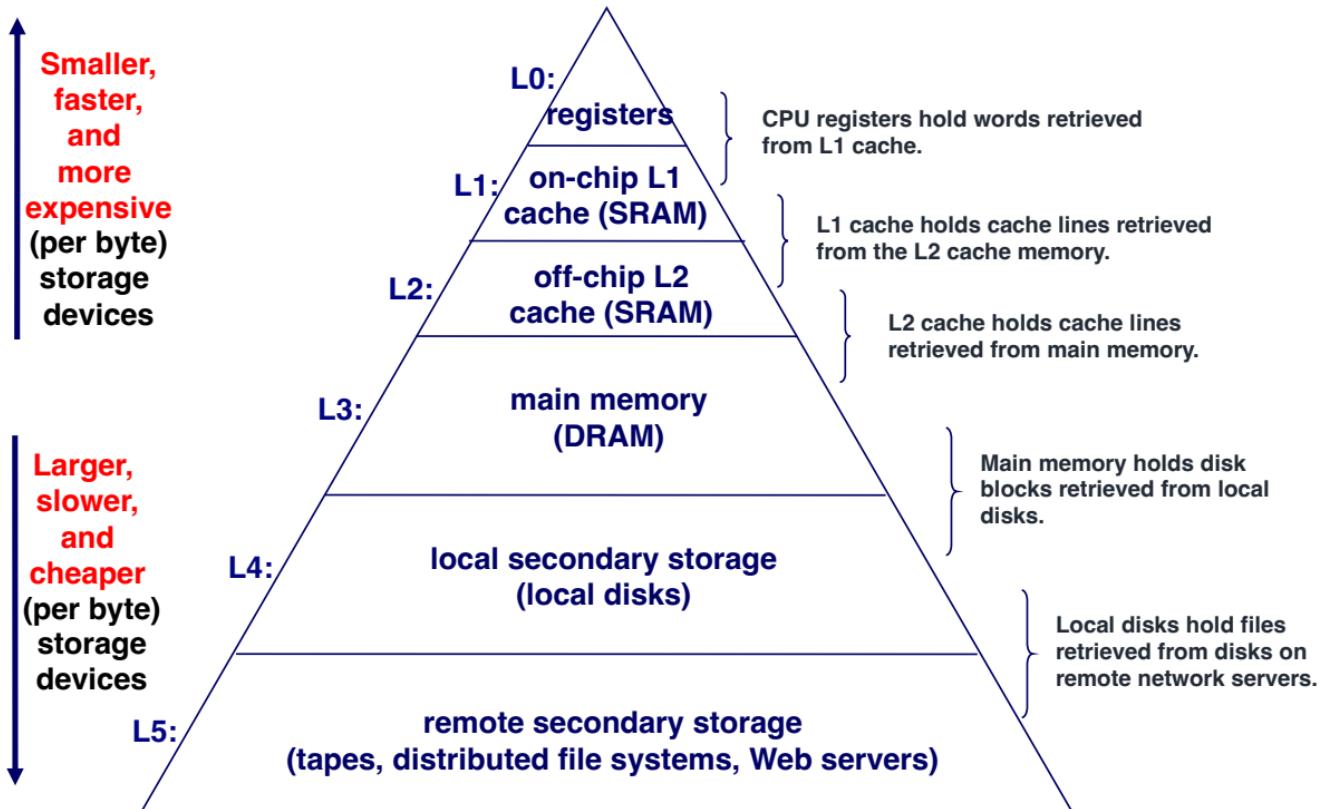
(progressively bigger and slower when far from the processor) and ensure most of the processor data is kept in the fast(er) level(s)



# Concept of Memory Hierarchy



# An Example of Memory Hierarchy



# Multi-CPU Architectures and Coherence

Writing parallel programs requires a **good hardware knowledge**

**Programmability issues** when there are multiple processing elements

Assume just single level caches and main memory

Processor (or coprocessor) writes to location in its cache

Other caches may hold shared copies - these will be out of date

Updating main memory alone is not enough

**Cache coherence:** consistency in the value of data between the versions in the (local) memories of several processing elements

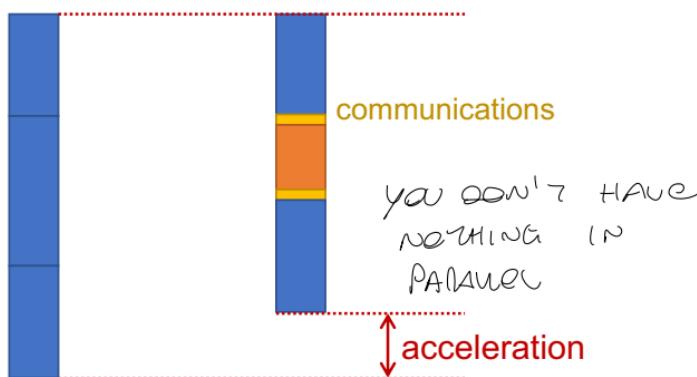
When do we need **cache coherence** in case of coprocessors?



# Heterogeneous Processing Elements

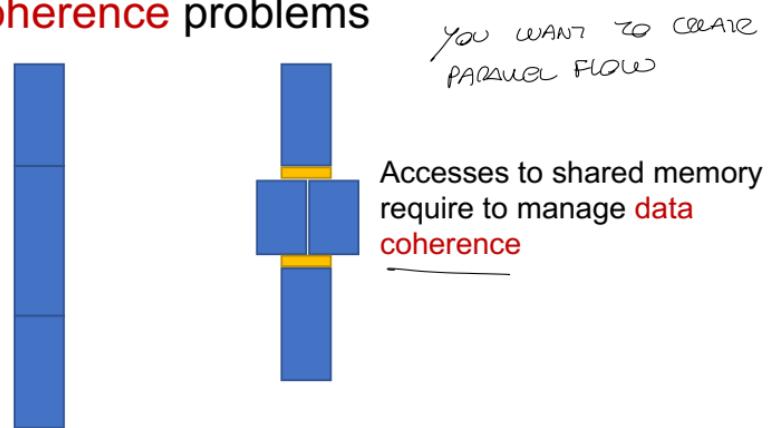
## Offloading Execution

- Master CPU is **on hold**
- Mostly to improve performance (**acceleration**) and/or energy consumption (**specialization**)



## Parallel Execution

- All units are **active** (equally?)
- Mostly to exploit **task-level parallelism**
- May introduce **synchronization** and **coherence** problems

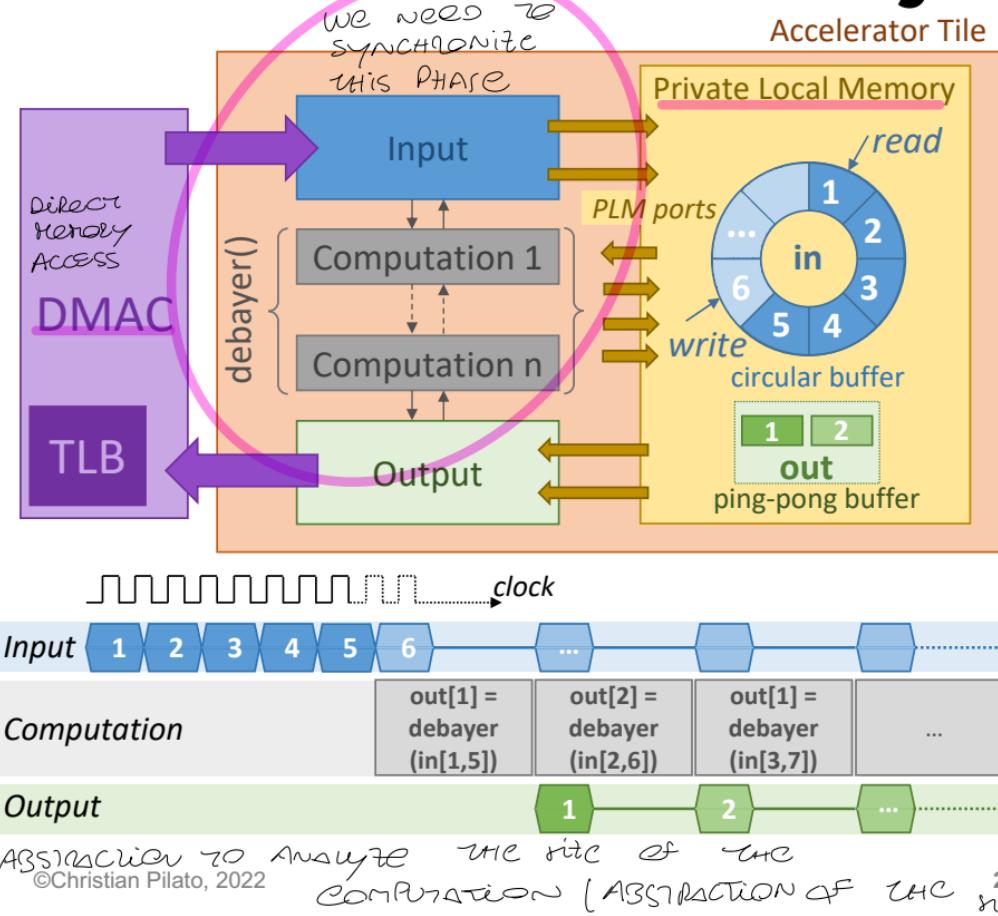


Especially in case of computation to be done on multiple independent data (massively parallel)



# Accelerators with Private Local Memory

- **Private Local Memory**
  - A **specialized multi-bank local storage** for highly-parallel data path
- **Autonomous DMA**
  - Without CPU intervention
  - Without system knowledge
- **Transaction-level abstraction**
  - Decouple input and output phases from computation
  - **Pipelining**

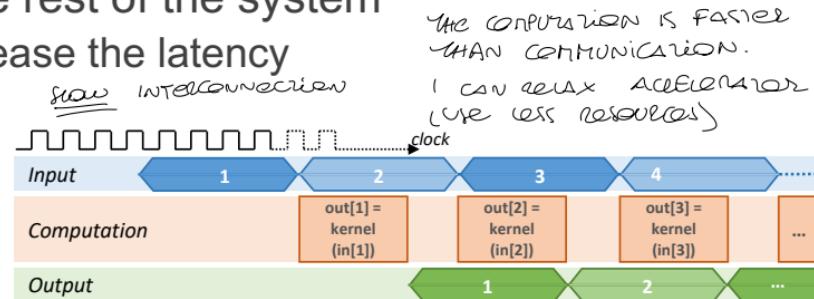
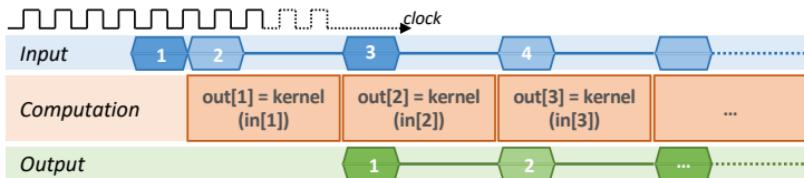


# What Happens with Multiple Accelerators?

Balancing communication and computation is crucial for performance optimization

- Optimizing microarchitecture reduces the **computation latency**
  - Combination of HLS transformations and PLM customization
- **Input and output phases** interact with the rest of the system
  - Backpressure due to congestion may increase the latency

Optimization: try to reuse the computation part



**Reduce the congestion or exploit the congestion**  
to optimize the execution at the system level

# SoC Communication

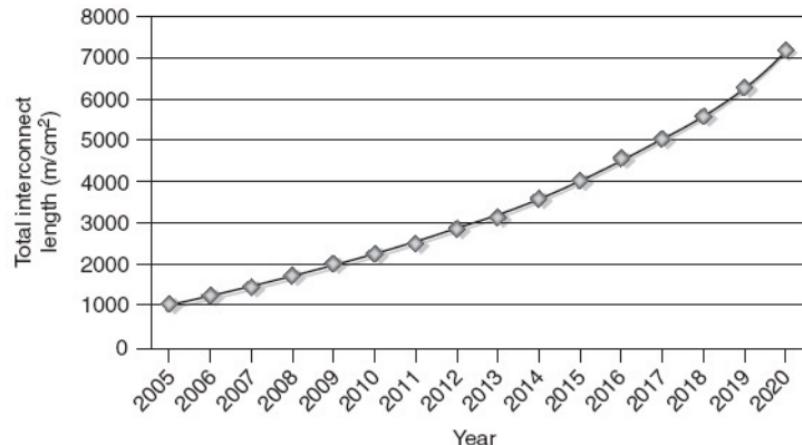
**Communication** is the most critical aspect affecting system performance

- **Communication architecture** consumes up to **50% of total on-chip power**
- Ever increasing number of wires, repeaters, bus components (arbiters, bridges, decoders etc.) increases system cost
- Design flow must include communication design, customization, exploration, verification and implementation

## Communication Architectures

in today's complex systems  
significantly affect performance,  
power, cost and time-to-market!

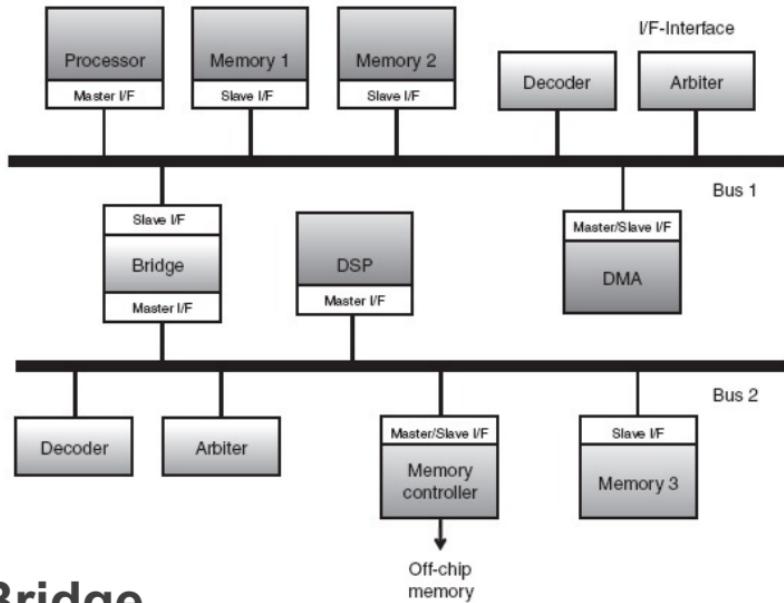
## Flexibility vs. Efficiency



# Bus Terminology

IN GENERAL THE MEMORY IS  
ALWAYS THE SLAVE

- **Master (or Initiator)**
  - IP component that initiates a read or write data transfer
- **Slave (or Target)**
  - IP component that only responds to incoming transfer requests
- **Arbiter**
  - Controls access to the shared bus
  - Uses arbitration scheme to select master to grant access to bus
- **Decoder**
  - Determines which component a transfer is intended for



## • Bridge

- Connects two busses
- Acts as slave on one side and master on the other



# Bus Signal Lines



- A bus typically consists of **three types of signal lines**
  - **Address**
    - Carry **address of destination** for which transfer is initiated
    - Can be shared or separate for read, write data
  - **Data**
    - Carry **information** between source and destination components
    - Can be shared or separate for read, write data
    - Choice of data width critical for application performance
  - **Control**
    - **Requests and acknowledgements**
    - Specify more information about type of data transfer
    - Byte enable, burst size, cacheable/bufferable, write-back/through, ...



# Physical Limitations

- Bus wires are implemented as **long metal lines** on a silicon wafer
  - Transmitting data using electromagnetic waves (**finite speed limit**)
- As application performance requirements increase, clock frequencies are also increasing
  - Greater bus clock frequency = shorter bus clock period
    - 100 MHz = 10 ns ; 500 MHz = 2 ns
- Time allowed for a signal on a bus to travel from source to destination in a single bus clock cycle is decreasing
- Can take **multiple cycles** to send **a signal across a chip**
  - 6-10 bus clock cycles @ 50 nm
  - unpredictability in signal propagation time has serious consequences for performance and correct functioning of synchronous digital circuits



# Bus pros (😊) and cons (😢)

- 😊 The silicon cost of a bus is small.
- 😊 Any bus is almost directly compatible with most available IPs, including software running on CPUs.
- 😊 The concepts are simple and well understood.
  
- 😢 Every unit attached adds parasitic capacitance, therefore electrical performance degrades with growth.
- 😢 Bus timing is difficult in a deep submicron process.
- 😢 Bus arbiter delay grows with the number of masters. The arbiter is also instance-specific.
- 😢 Bandwidth is limited and shared by all units attached.

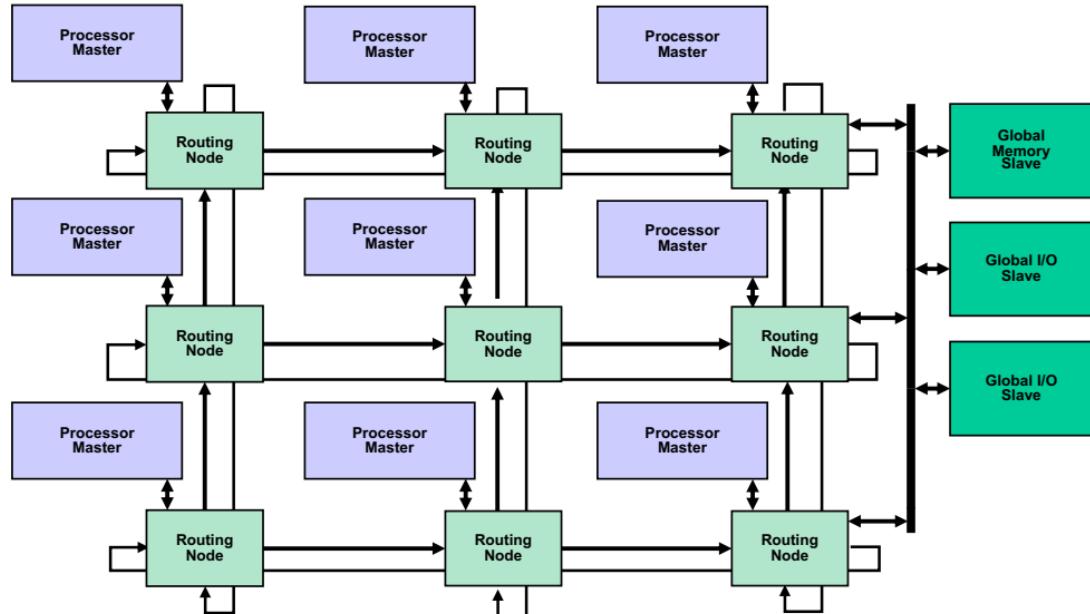


# Network-on-Chip

Wiring a network  
with routers

Leveraging existing  
**computer networking principles** to improve  
inter-component intra-chip  
communications

- Each on-chip component connected by an intelligent switch to particular communication wire(s)
- Improvement over standard bus-based interconnections for SoC architectures in terms of throughput



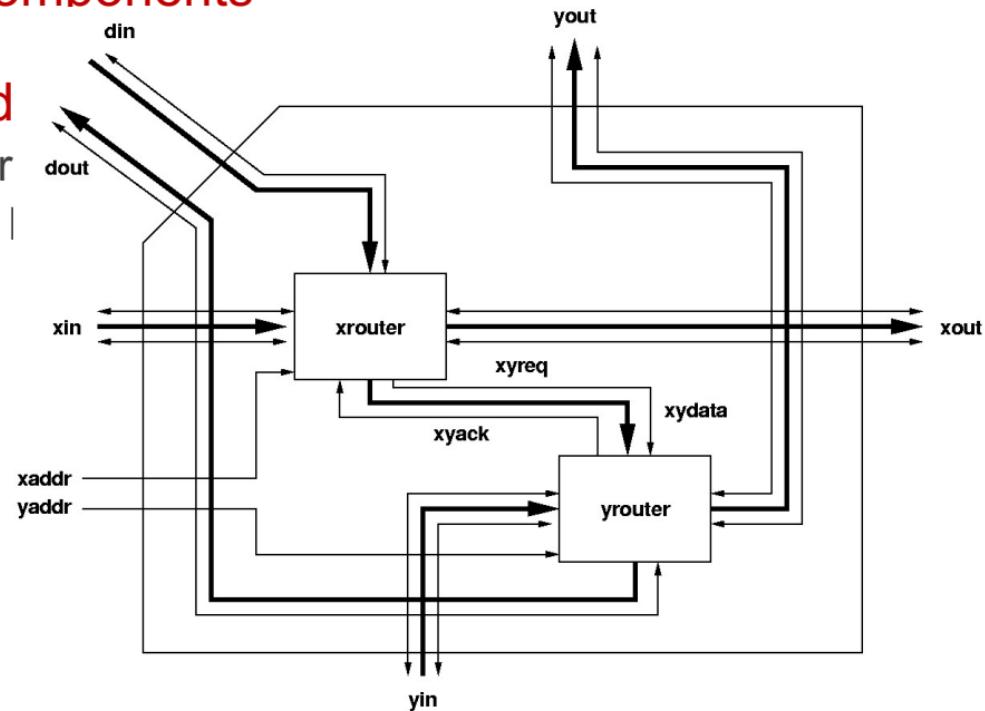
The complexity is inside  
the router.

Very scalable solution  
Very efficient solution

# NoC Routers

- Provide **interface** with **NoC components**
- Responsible for **routing** of **ind**
  - Provides for so-called best-effort
  - Sharing of resources allows for |

Multiple planes to support multiple services



# **Design of Hardware Accelerators**

Academic Year 2021/2022

## Questions?

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)



**POLITECNICO**  
MILANO 1863

**Design of Hardware Accelerators**  
Academic Year 2021/2022

# Design Flows and FPGA Technology

**Christian Pilato**

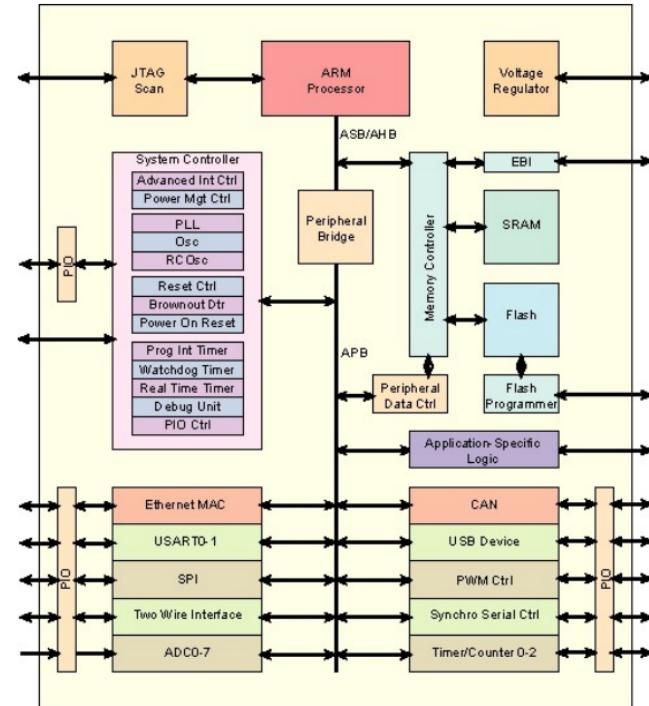
Dipartimento di Elettronica, Informazione e Bioingegneria

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)

# System-on-Chip Architectures

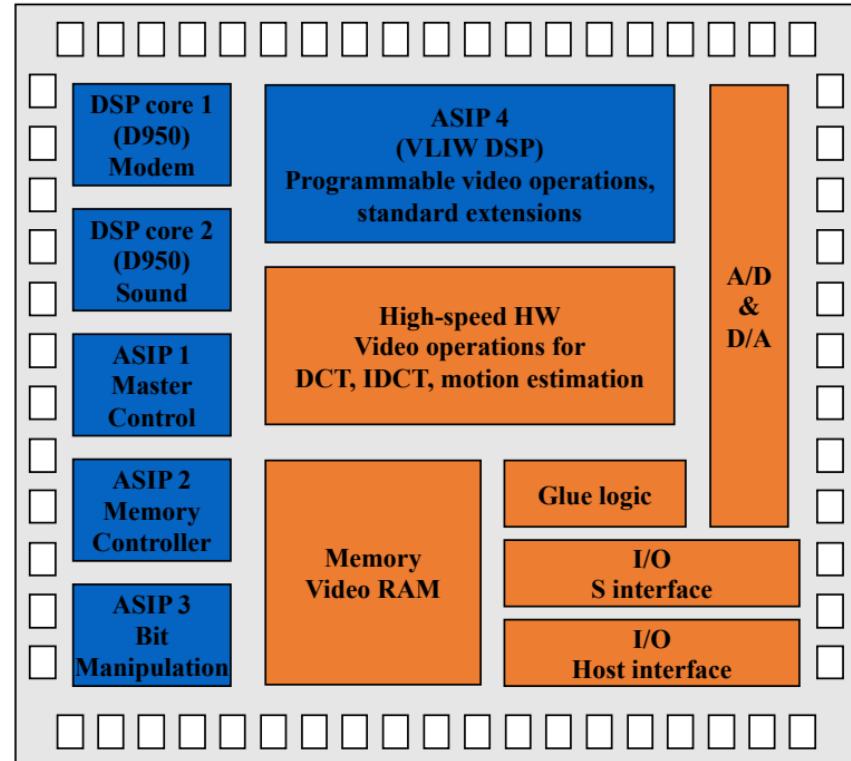
- **Specialized architectures** composed of several **IP components**

- 😊 Decreased power consumption
- 😊 Increased reliability
- 😊 Smaller board space
- 😊 Cheaper with ready-to-go components
- 😢 Extremely high design cost for the chip
- 😢 Large silicon space may be required
- 😢 Component testing may be difficult
- 😢 Prototyping may take longer
- 😢 Intellectual property (IP) issues



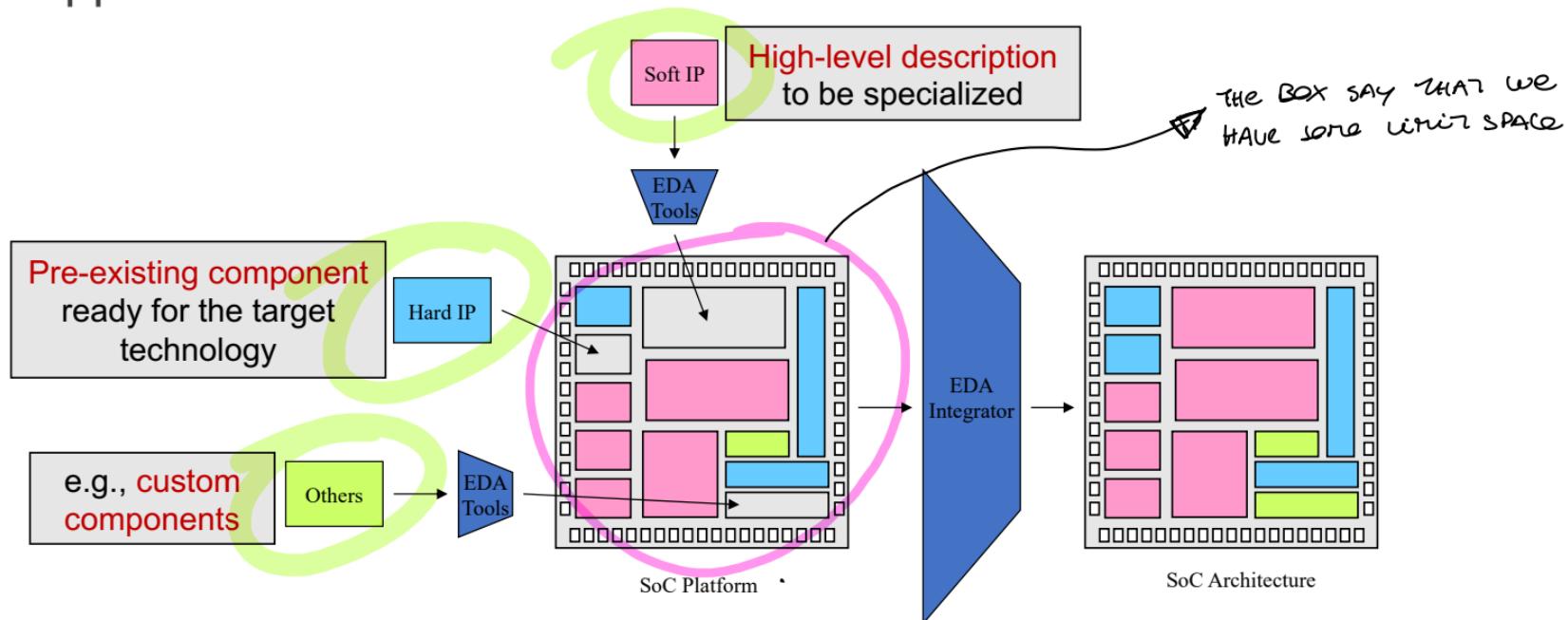
# From Specifications to Chips

```
for(i = 0; i < 18; i++) {  
    s = (mpffloat)0.0f;  
    k = 0;  
    do {  
        s += X[k] * v[k];  
        s += X[k+1] * v[k+1];  
        s += X[k+2] * v[k+2];  
        s += X[k+3] * v[k+3];  
        s += X[k+4] * v[k+4];  
        s += X[k+5] * v[k+5];  
        k += 6;  
    } while(k < 18);  
    v += 18;  
    ISCALE(s);  
    t[i] = s;  
}  
/* correct the transform into the 18x36 IMDCT we need */  
/* 36 muls */  
  
for(i = 0; i < 9; i++) {  
    x[i] = t[i+9] * Granule_imdct_win[gr->block_type][i];  
    ISCALE(x[i]);  
    x[i+9] = t[17-i] * Granule_imdct_win[gr->block_type][i+9];  
    ISCALE(x[i+9]);  
    x[i+18] = t[8-i] * Granule_imdct_win[gr->block_type][i+18];  
    ISCALE(x[i+18]);  
    x[i+27] = t[i] * Granule_imdct_win[gr->block_type][i+27];  
    ISCALE(x[i+27]);  
}
```



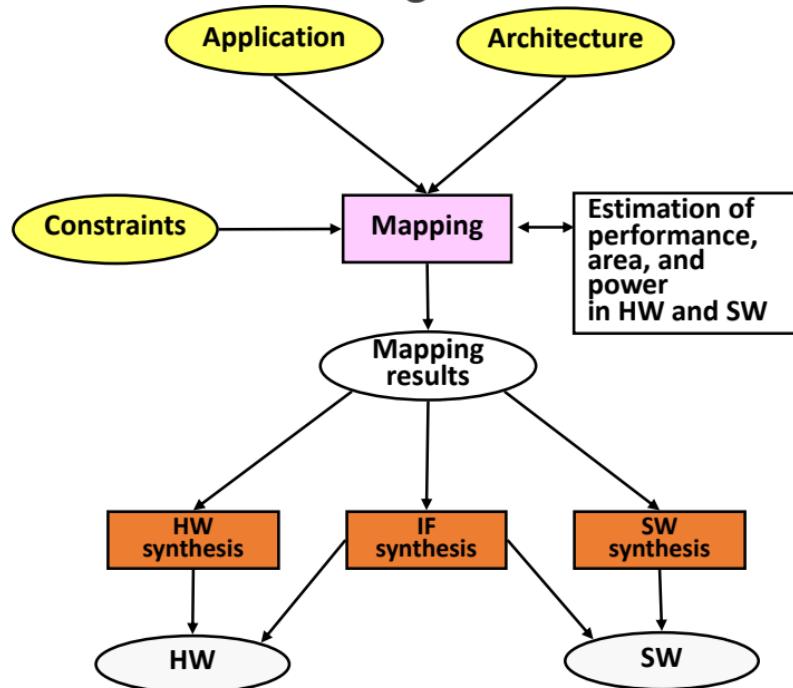
# Platform-based design

Design methodology to **customize** an **existing platform template** with the support of **EDA tools**



# Hardware/Software Co-Design

Refinement of an **application** and an **architecture template**  
based on the given **constraints**

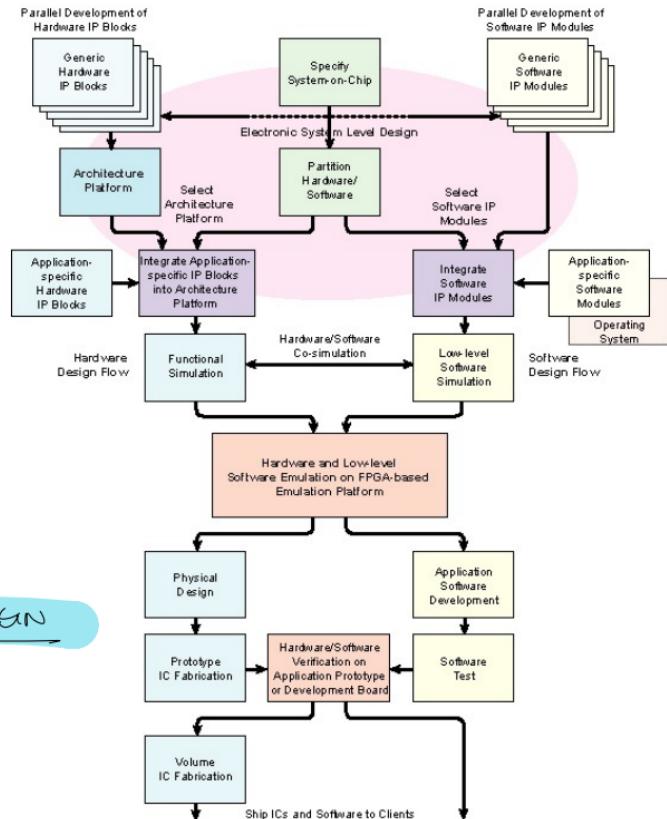
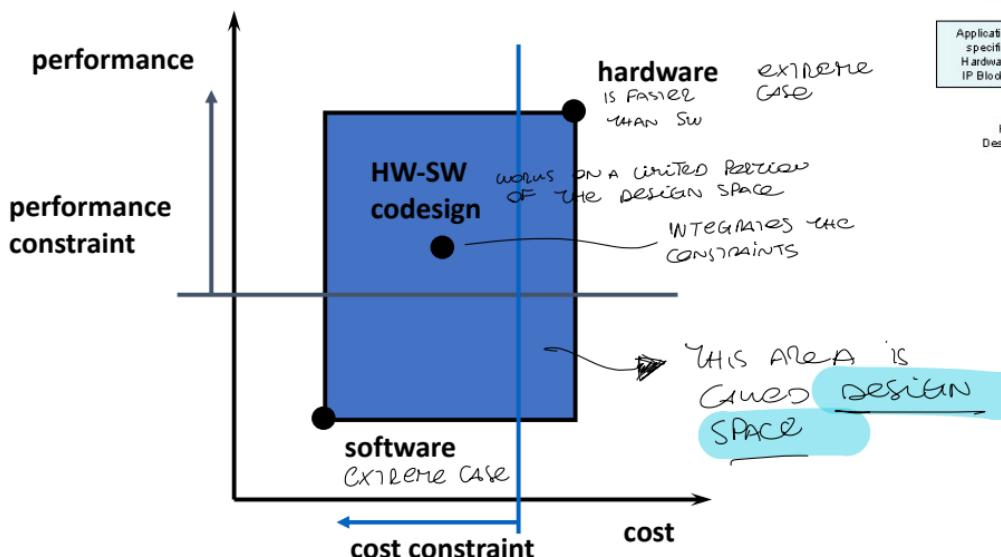


# Hardware-Software Co-Design

We will see A bottom-up APPROACH

## Balancing

- Performance of customized HW units
- Programmability of low-cost SW components



# Hardware Design

- As the complexity increases, initial design moves toward **more abstract levels**

- Top-down design**

- Behavioral-level design (Architecture synthesis tool)
- RT-level design (Logic synthesis tool) operates between registers
- Gate-level design (Layout synthesis tool)

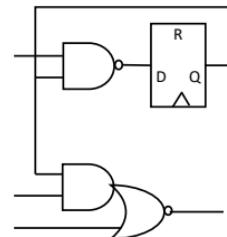
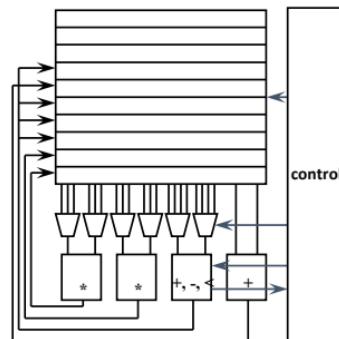
you define the system until you reach the **smallest component of the design**.

- Top-down + bottom-up** (Synthesis with libraries)

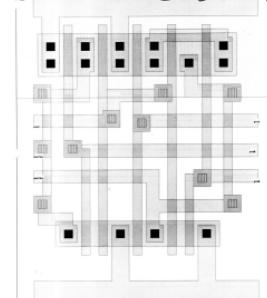
High level design

```
module DIFFEQ(x, y, u, dx, a, clock, start);
  input [7:0] a, dx;
  inout [7:0] x, y;
  input clock, start;
  reg [7:0] xl, ul, yl;
  always
  begin
    wait (start);
    while (x < a)
    begin
      xl = x + dx;
      ul = u - (3*x*u*dx) - (3*y*dx);
      yl = y + (u*dx);
      @(posedge clock);
      x = xl; u = ul; y = yl;
    end
  end
endmodule
```

Description of a slow filter

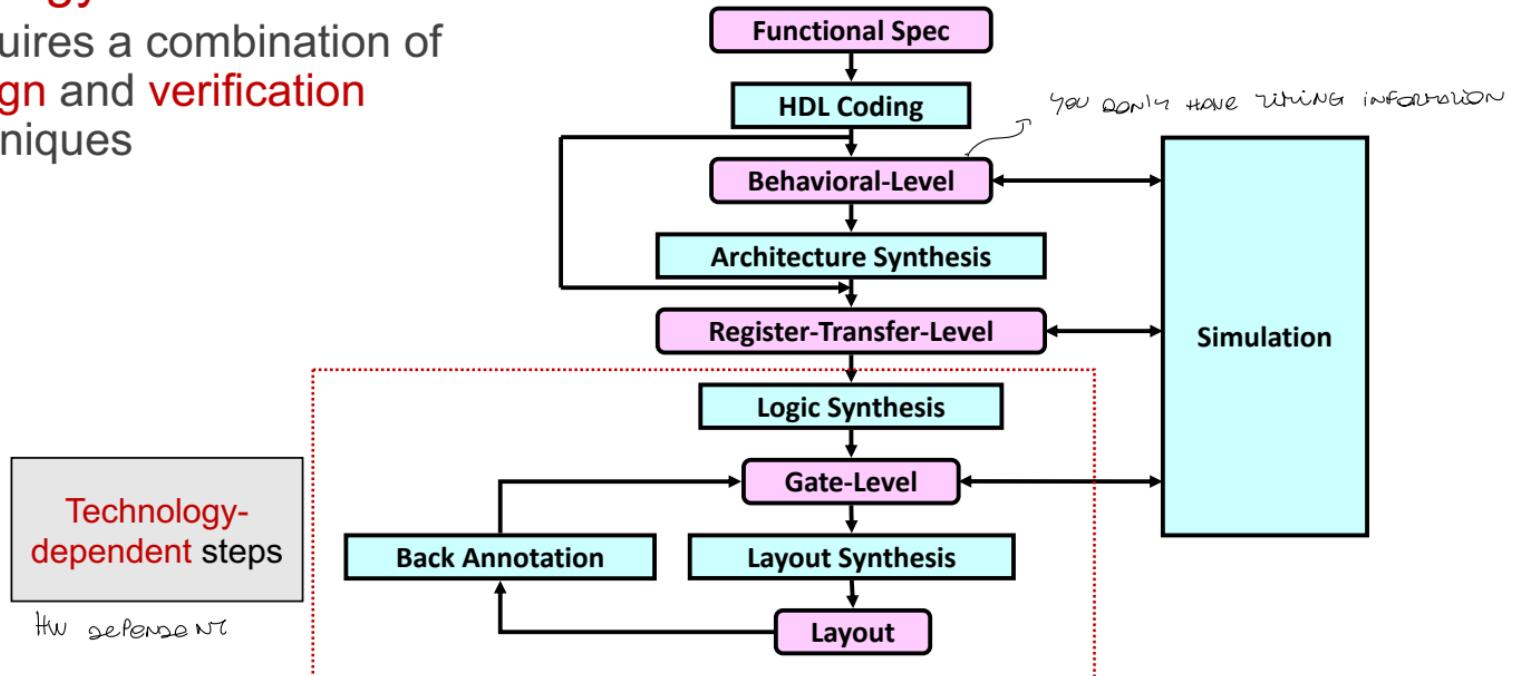


you create the actual circuit



# Hardware Design Flow

- Sequence of **passes** to **refine a specification** with respect to the **target technology**
  - Requires a combination of **design** and **verification** techniques



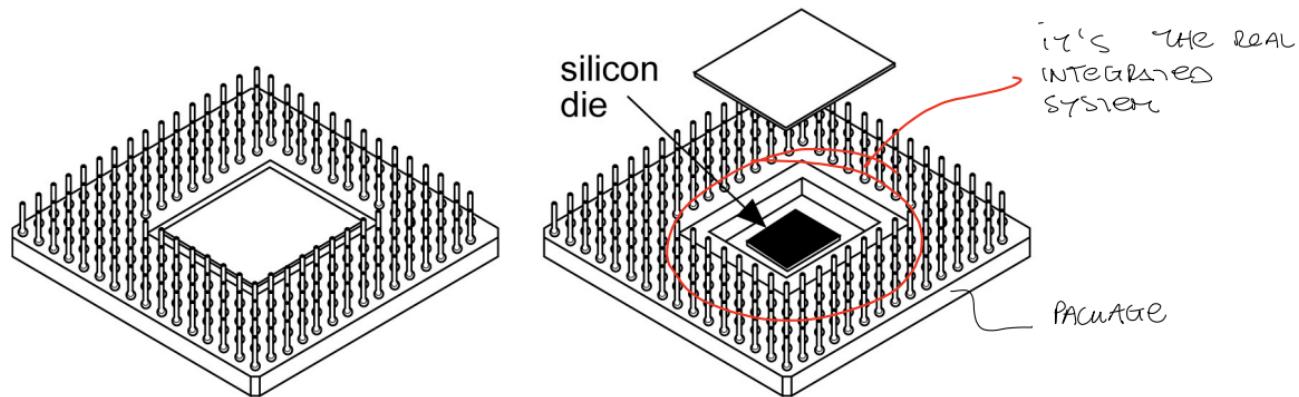
# Software Design

- Why is software still needed?
  - Flexibility, upgradability *ALSO BECAUSE SW IS LESS EXPENSIVE*
  - Low cost
- Design challenges
  - **Parallel programming** *→ there is the problem of coherence*
    - Software **development** for MP-SoC
    - **Partitioning** for load balancing, communication overhead, ...
  - **Code optimization**
    - Minimization of **memory size**
    - Maximization of **performance**
    - Minimization of **power consumption**
  - **Real-time constraints**
    - Hard real-time (worst case performance), Soft real-time (probabilistic performance),  
No real-time (average case performance)



# ASIC (*Application Specific Integrated Circuit*)

- An Integrated Circuit (IC) designed to perform a specific function for a specific application



→ 1 metric to use

- **Gate equivalent** - a **unit of size** measurement corresponding to a 4-transistor gate equivalent (e.g. a **2-input NOR gate**)
  - Used to compare different technologies

Functionally complete  
you can create all the other  
functions

# Types of ASIC

## • Fixed-function ASICs

- Full-Custom ASICs : you have a lot of time to place each single transistor  
Best performance but it will take forever
- Standard-Cell-Based ASICs : you have some predefined cell  
It's equivalent an a compiler.
- ...

## • Reprogrammable ASICs

you can reprogram the functionality

- Gate-Array–Based ASICs
- Channeled Gate Array
- Channel-less Gate Array
- Structured Gate Array
- Programmable Logic Devices
- Field-Programmable Gate Arrays
- ...

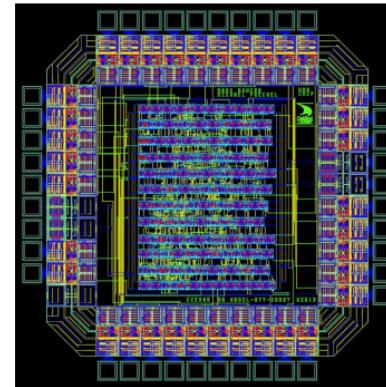


# Focus on Two Technologies

classic view of fixed-function ASICs

- **Fixed-function ASICs**

- Very integrated, yet very expensive
- I WANT TO REPROGRAM A FUNCTIONALITY YOU HAVE TO  
REDISIGN A NEW CHIP



- **FPGA – Field-Programmable Gate Array**

- Cheaper to implement, reprogrammable



# Full-Custom ASICs

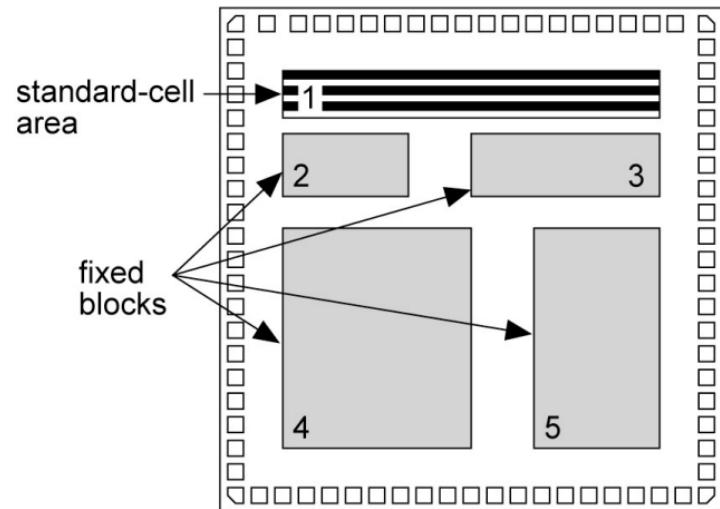
- Full-custom design offers the **highest performance** and lowest part cost (**smallest die size**) for a given design
  - All mask layers are customized
    - Generally, the designer lays out all cells by hand
    - Some automatic placement and routing may be done
    - Critical (timing) paths are usually laid out completely by hand
- The disadvantages of full-custom design include **increased design time, complexity, design expense, and highest risk**
- Microprocessors (strategic silicon) were exclusively full-custom
  - Designers are increasingly turning to semicustom ASIC techniques
  - Other examples of full-custom ICs or ASICs are requirements for high-voltage (automobile), analog/digital (communications), sensors and actuators, and memory (DRAM)
- Most of **Analog/Digital interfaces** are full-custom

**Not scalable!**



# Standard-Cell-Based ASICs

- Based on a set of **full-custom macros**
  - Standard cells, megacells , megafunctions, full-custom blocks, system-level macros (SLMs), fixed blocks, cores, Functional Standard Blocks (FSBs), ...
- All **mask layers** are customized, (both) transistors and interconnect
  - Automated buffer sizing, placement and routing
- Custom blocks can be embedded
- Manufacturing lead time is about eight weeks



# ASIC Cell Libraries

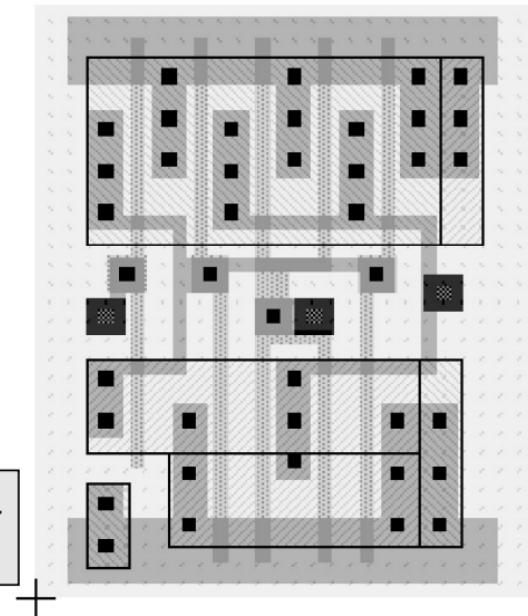
- A **library of cells** is used by the designer to design the **basic logic functions** for an ASIC (building blocks – equivalent to microinstructions)
- Options for cell library
  - (1) Use a **design kit** from the ASIC vendor
    - Usually requires the use of ASIC vendor approved tools
    - Cells are “phantoms” - empty boxes that get filled in by the vendor when you deliver, or "hand off" the netlist
    - Vendor may provide more of a “guarantee” that design will work
  - (2) Buy an **ASIC-vendor library** from a library vendor
    - Library vendor is different from fabricator (foundry)
    - Library may be approved by the foundry (qualified cell library)
    - Allows the designer to own the masks (tooling) for the part when finished
  - (3) You can build your own cell library (**application-specific cell libraries**)
    - Difficult and costly



# ASIC Library Development

- A complete ASIC library (suitable for commercial use) must include the following for each cell and macro:
  - A behavioral model (VHDL or Verilog model)
  - A detailed timing model
  - A physical layout
  - A test strategy
  - A circuit schematic
  - A cell icon (symbol)
  - A wire-load model
  - A routing model

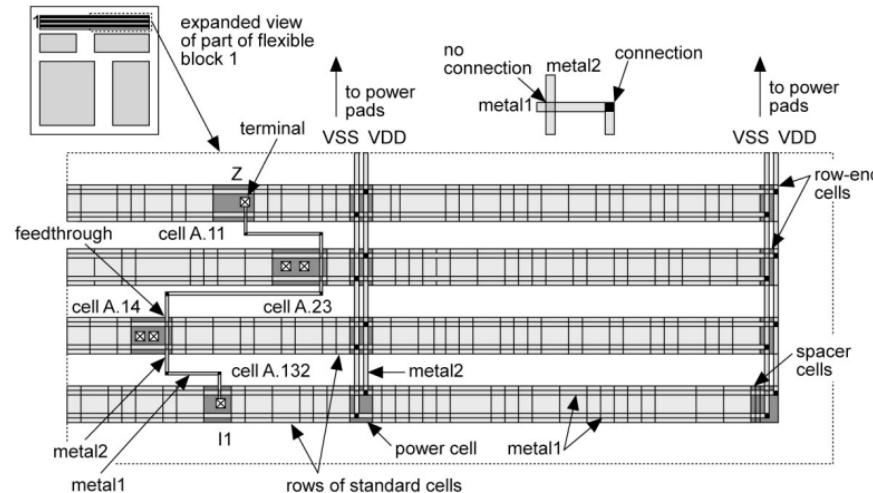
Example of standard-cell implementation



# Standard Cell ASIC Routing

- Standard cells are organized in "rows" (all cells have the same height)
  - Alignment of pins for wiring (only horizontal and vertical lines)
- Metal2 may be used to cross over cell rows that use metal1 for wiring
- Other wiring cells: buffer/filler cells, row-end cells, and power cells

Placement defines the position of the cells to simplify routing



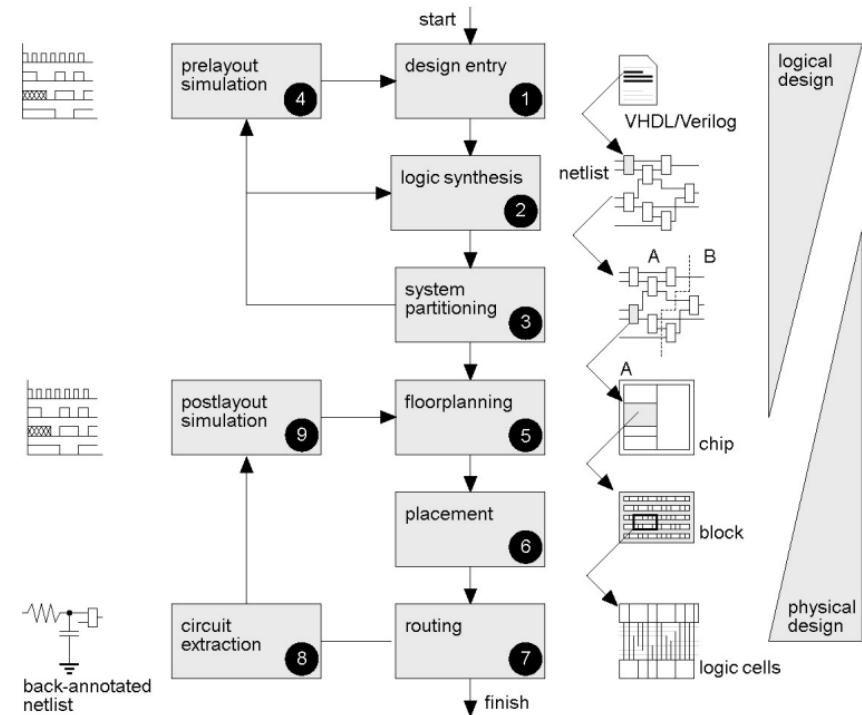
Complex routing can create violations to be solved manually or with iterations



# Design Flow

complete flow

1. **Design entry** - Using a hardware description language (HDL) or schematic entry
2. **Logic synthesis** - Produces a netlist - logic cells and their connections
3. **System partitioning** - Divide a large system into ASIC-sized pieces
4. **Prelayout simulation** - Check to see if the design functions correctly
5. **Floorplanning** - Arrange the blocks of the netlist on the chip
6. **Placement** - Decide the locations of cells in a block
7. **Routing** - Make the connections between cells and blocks (*including clock and power distribution*)
8. **Extraction** - Determine the resistance and capacitance of the interconnect (*based on resulting wirelength*)
9. **Postlayout simulation** - Check to see the design still works with the added loads of the interconnect



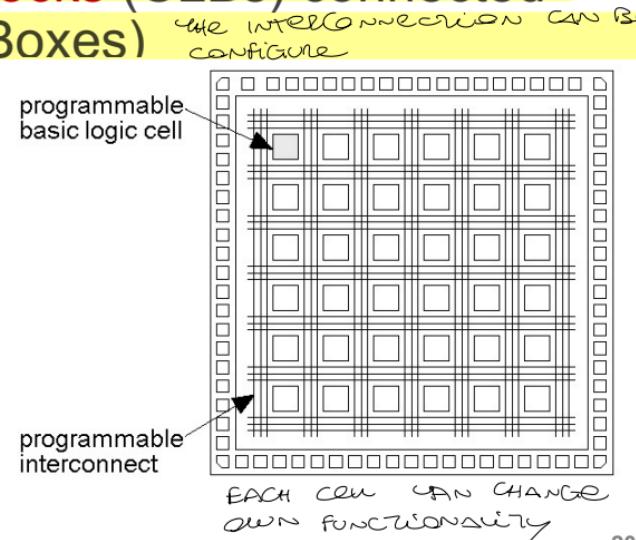
# Gate-Array-Based ASICs

- Transistors are **predefined** on the silicon wafer
  - The predefined pattern of transistors is called the *base array*
  - The smallest element that is replicated to make the base array is called the *base* or *primitive cell*
  - The top-level interconnect between the transistors is defined by the designer in custom masks - *Masked Gate Array (MGA)*
- Design is performed by **connecting predesigned and characterized logic cells** from a library (macros)
- After validation, automatic placement and routing are typically used to **convert the macro-based design into a layout** on the ASIC using primitive cells



# FPGA (*Field Programmable Gate Array*)

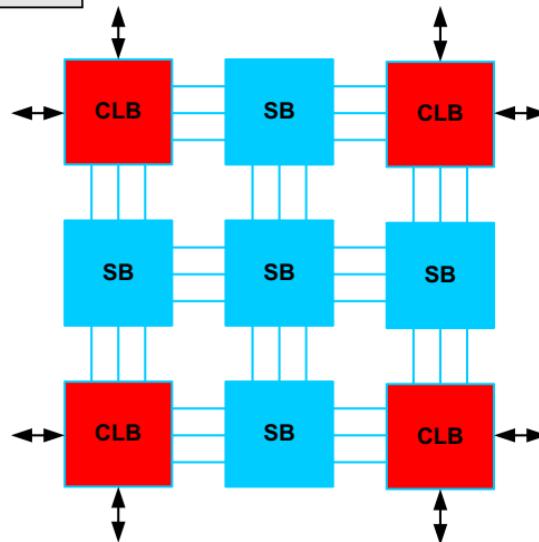
- An **integrated circuit** that can be **configured** by the **user** to emulate any digital circuit as long as there are enough resources
  - The core is a regular array of **programmable basic logic cells** that can implement **combinational** as well as **sequential** logic (flip-flops)
- An FPGA is an array of **Configurable Logic Blocks** (CLBs) connected through **programmable interconnect** (Switch Boxes)
  - None of the mask layers are customized
  - A method for programming the basic logic cells and the interconnect
  - A matrix of programmable interconnect surrounds the basic logic cells
  - Programmable **I/O cells** surround the core



# FPGA Structure

HIGH # OF PROGRAMMABLE CELLS

Very regular design allowing for high-density chips



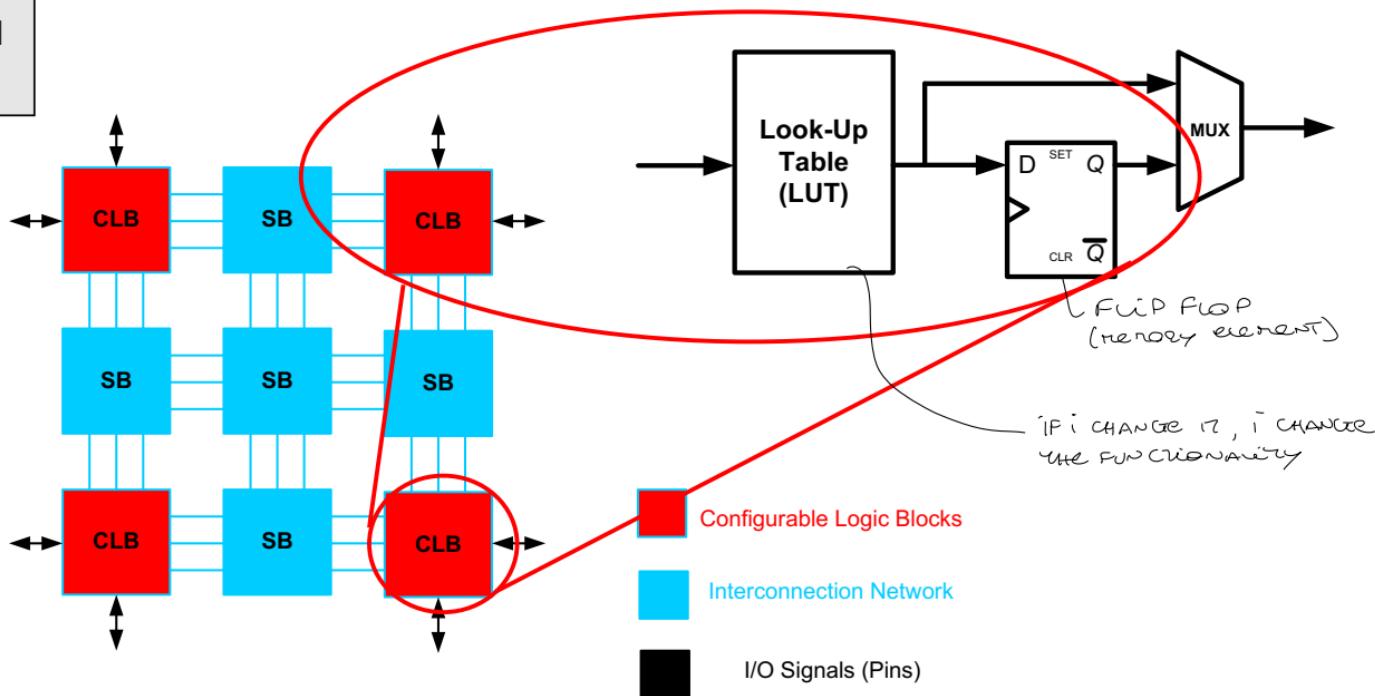
Can be complemented by specialized blocks (e.g., RAM or DSP)

- Configurable Logic Blocks
- Interconnection Network
- I/O Signals (Pins)



# Simplified CLB Structure

Both **combinational** and **sequential** logic

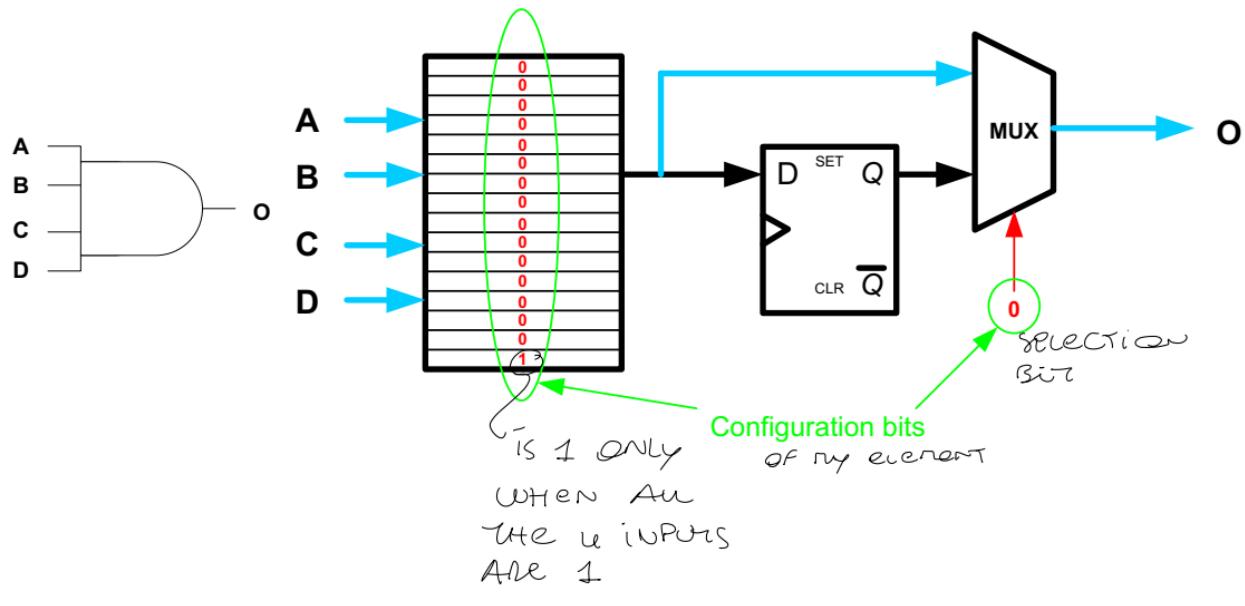


# Example: 4-input AND gate

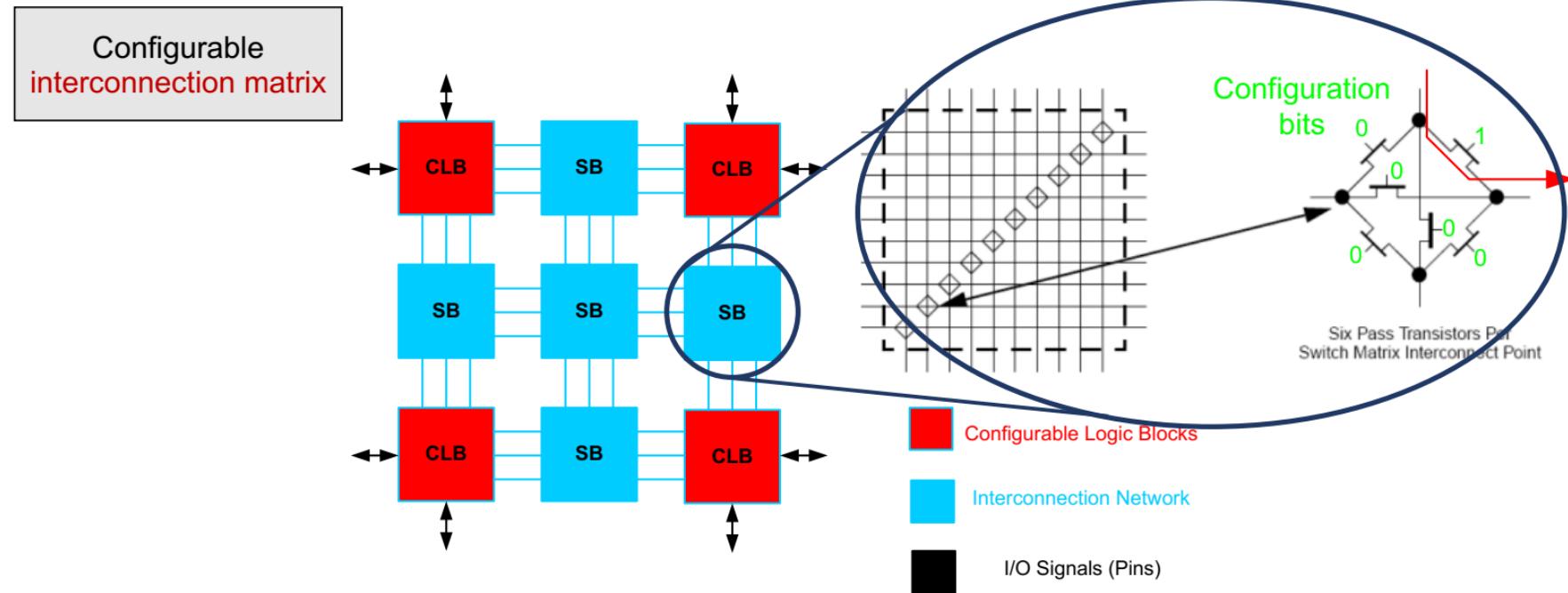
4 INPUT LUT

A	B	C	D	O
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

$2^4$  CONFIGURATION BITS

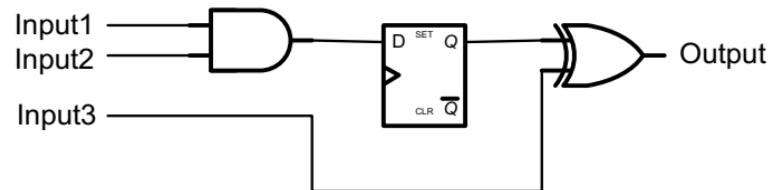
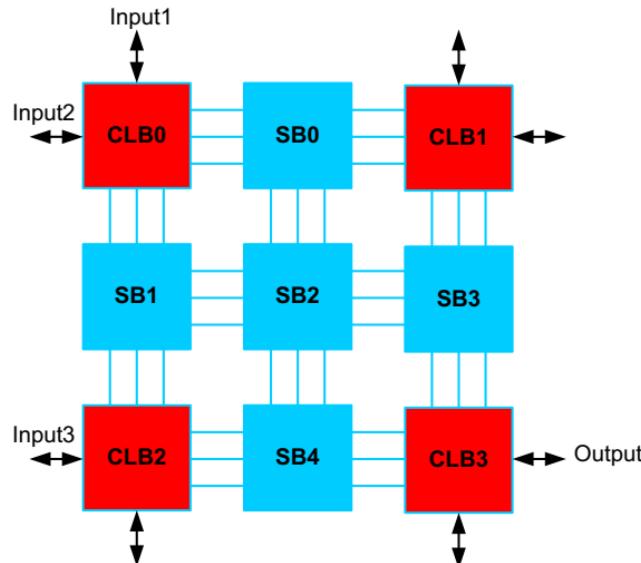


# Interconnection Network



# Example

Determine the configuration bits for the following circuit implementation in a **2x2 FPGA**, with **I/O constraints** as shown in the following figure. Assume 2-input LUTs in each CLB.

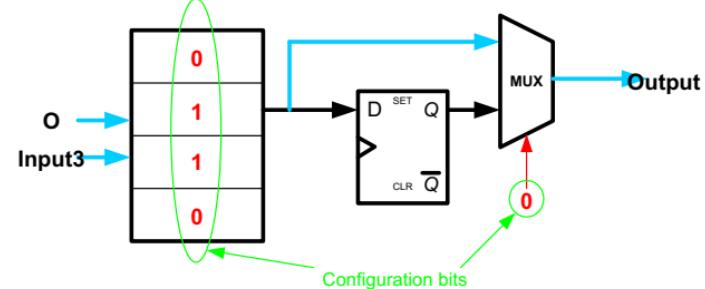
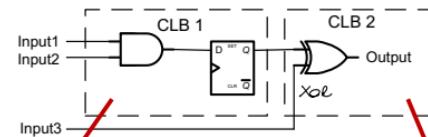
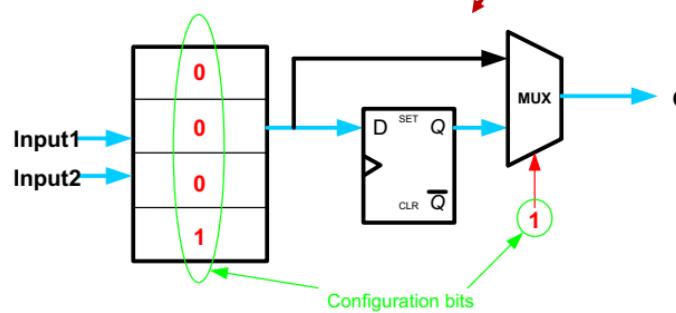


Requires separation  
into **multiple CLBs**

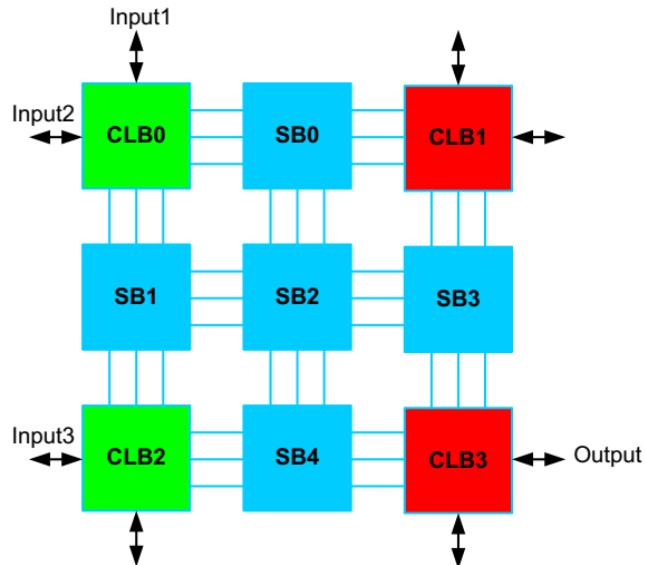


# CLBs Required

Creation of 2-input clusters (logic synthesis)



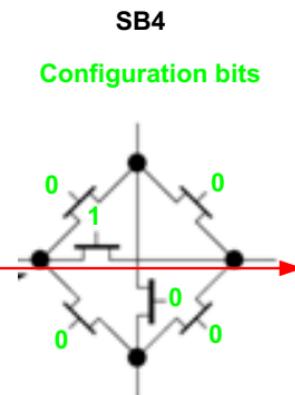
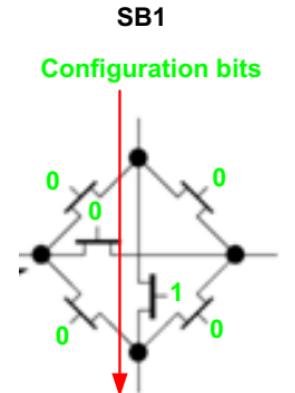
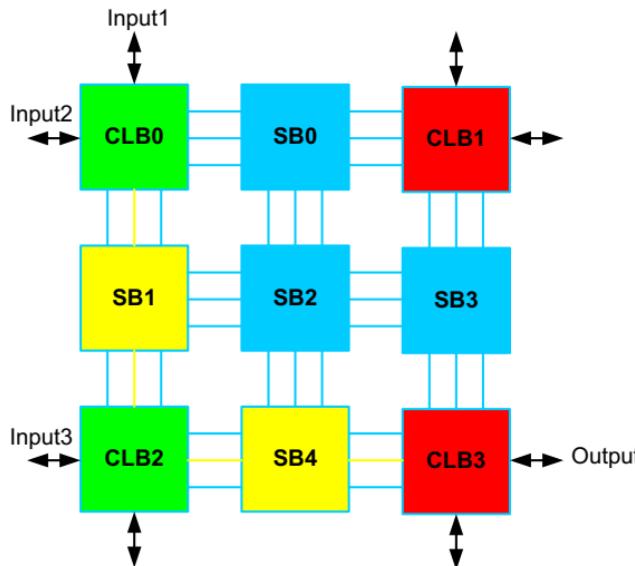
# Placement: Select CLBs



Dependent on I/O constraints



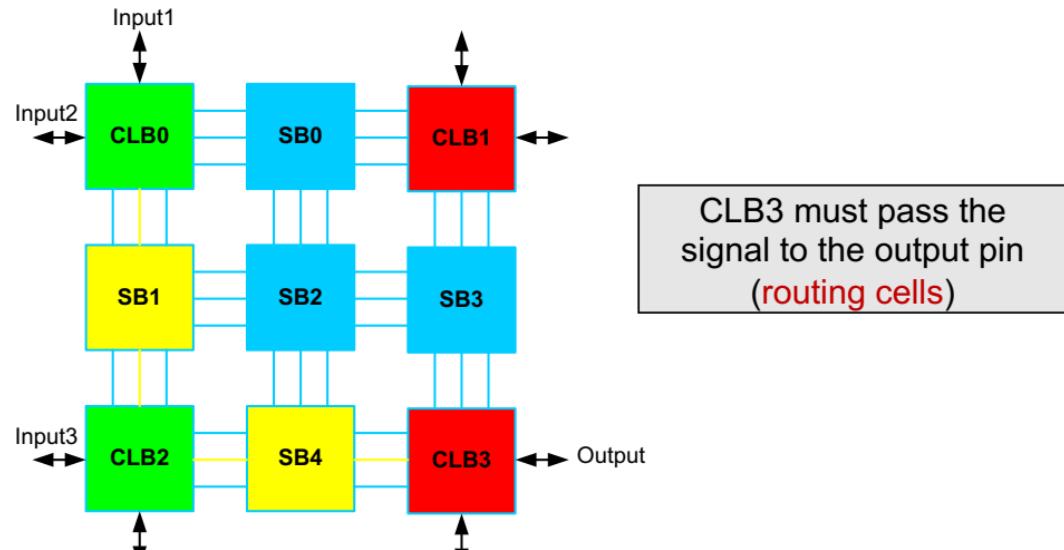
# Routing: Select Path



# Configuration Bitstream

- The configuration bitstream must include ALL CLBs and SBs, even unused ones

- CLB0: 00011
- CLB1: XXXX
- CLB2: 01100
- CLB3: ?????
- SB0: 000000
- SB1: 000010
- SB2: 000000
- SB3: 000000
- SB4: 000001

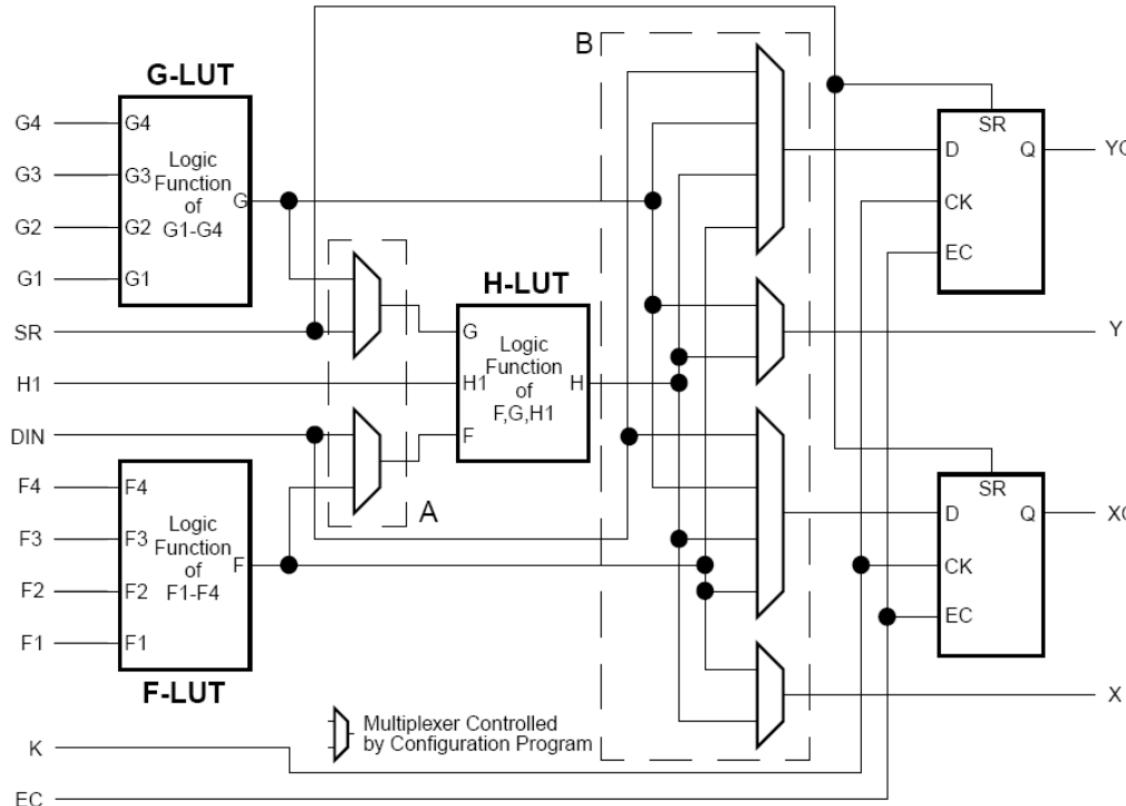


CLB3 MUST HAVE  
A PRECISE FUNCTIONALITY  
THAT I HAVE TO CONFIGURE

©Christian Pilato, 2022



# Realistic FPGA CLB: Xilinx



# Economics of ASICs

- Total product (or part) cost is a function of fixed cost, variable cost, and the number of products (parts) sold:

$$\text{total part cost} = \text{fixed part cost} + \text{variable cost per part} \times \text{volume of parts}$$

→ IS HIGHER IN FPGA

- On a parts only basis, an FPGA is more expensive per-gate than a CBIC
- Fixed cost of the CBIC is higher than the fixed cost of FPGA
  - Design cost
  - Fabrication cost
- Example, assume:
  - FPGA fixed cost is \$21,800, part cost is \$39
  - CBIC fixed cost is \$146,000, part cost is \$18

How many parts to  
get profit?



# SoC Design Flow Conclusions

Based on **platform-based design** and **hardware-software co-design**

- IP hardware blocks and software blocks are developed in parallel
  - Hardware with CAD tools
  - Software with development environments (e.g., SDK)
- Emulated and verified on FPGA
  - Fast creation of **executable implementations**
- Standard-cell implementation for **silicon creation**
  - Logic Synthesis, Technology Mapping, Placement and Routing based on a given set of library views



# **Design of Hardware Accelerators**

Academic Year 2021/2022

## Questions?

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)



**POLITECNICO**  
MILANO 1863

**Design of Hardware Accelerators**  
Academic Year 2021/2022

# Introduction to Hardware Descriptions

**Christian Pilato**

Dipartimento di Elettronica, Informazione e Bioingegneria

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)

10/03/2022

# What is Verilog?

↳ it's the equivalent of C for hardware

**Hardware Description Language (HDL)** to describe **digital circuits at many abstract levels**

- Developed in 1984 (Standard: IEEE 1364, Dec 1995) PRETTY OLD
- Used in almost all semiconductor companies and tools to specify components

Verilog is less verbose than VHDL

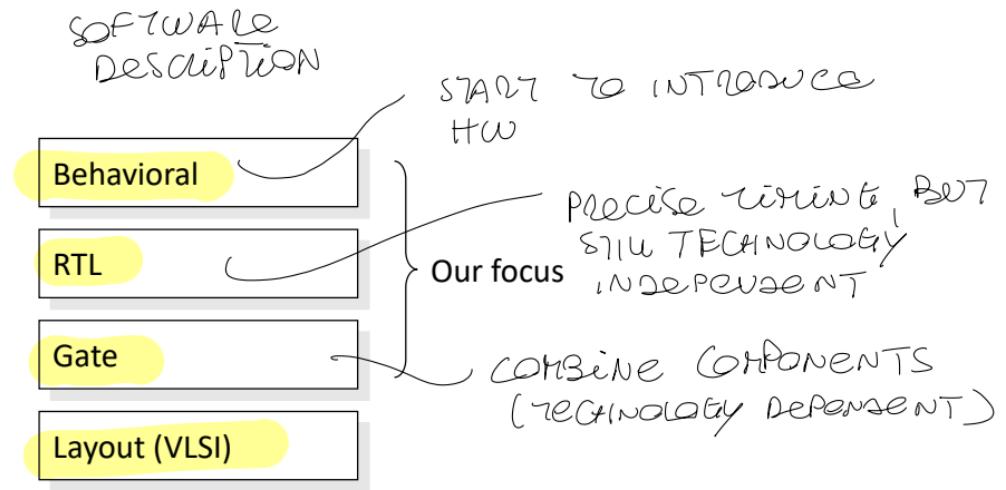
- **Why are we studying Verilog?** To become familiar with the hardware description language (HDL) approach for specifying designs

- Be able to read a simple Verilog HDL description
- Be able to write a simple Verilog HDL description using a limited set of syntax and semantics
- Understanding the need for a “hardware view” when creating an accelerator

TO DESIGN AN ACCELERATOR I WANT TO HAVE AN HW VIEW



# Abstraction Levels in Verilog



# Where can we use Verilog HDL?

You use Verilog for all phases (important property!)

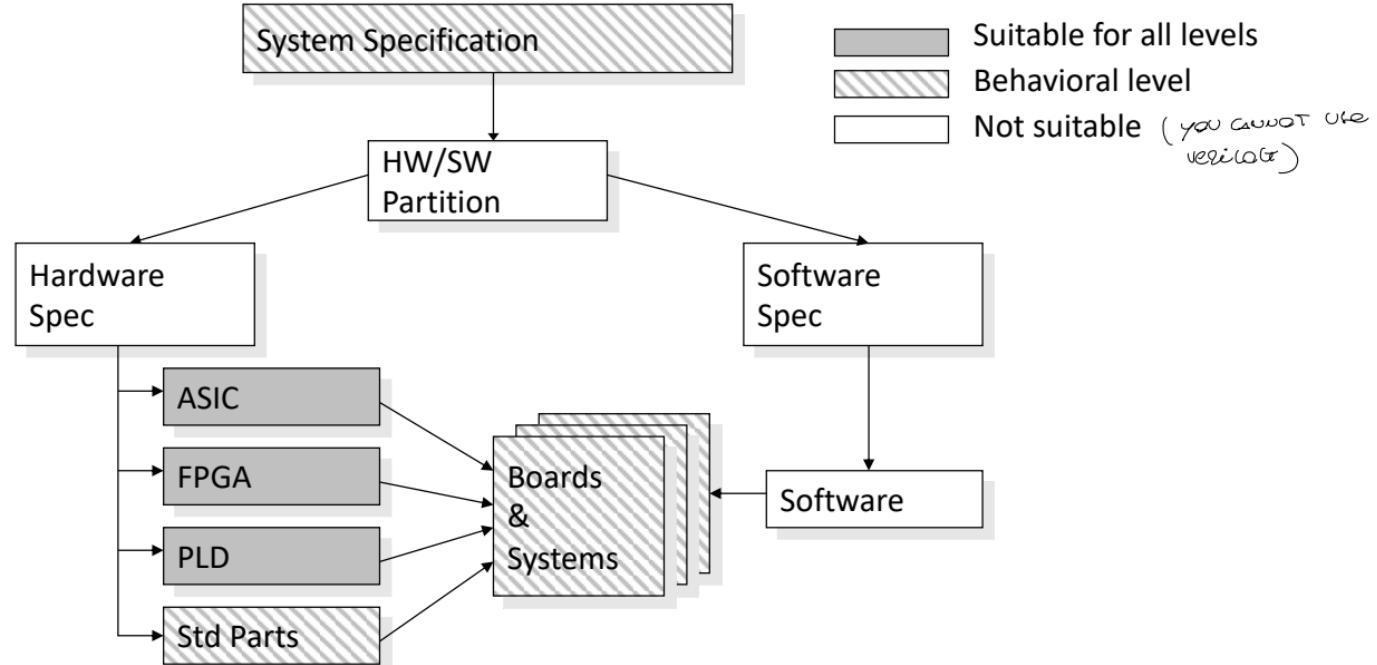
Verilog is designed for **circuit verification** and **simulation**, for **timing analysis**, for **test analysis** (testability analysis and fault grading) and for **logic synthesis**.

ACTUAL DEFINITION OF THE COMPONENT

For example, before you get to the structural level of your design, you want to make sure the logical paths of your design is faultless and meets the spec.



# Application Areas of Verilog



# User Identifiers and Notation

Formed from {[A-Z], [a-z], [0-9], \_, \$}, but ..

- .. cannot start with \$ or [0-9]

- myidentifier ✓
- m\_y\_identifier ✓
- 3my\_identifier ✗
- \$my\_identifier ✗
- \_myidentifier\$ ✓

- .. is case **case sensitive**

- myid ≠ Myid

List element separator: ,

Statement terminator: ;

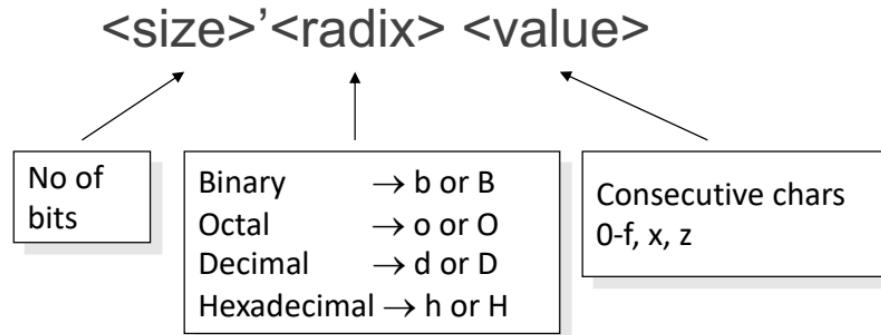


# Comments

- `//` The rest of the line is a comment
- `/*` Multiple line  
comment `*/`
- `/*` Nesting `/*` comments `*/` do **NOT** work `*/`



# Numbers in Verilog (i)



Examples:

- $8'h\ ax = 1010xxxx$       4 bits per letter
- $12'o\ 3zx7 = 011zzzxxx111$       3 bits per letter

- **0** represents **low logic level** or **false condition**
- **1** represents **high logic level** or **true condition**
- **x** represents **unknown logic level** – also X
- **z** represents **high impedance logic level** (open circuit) – also Z  
signal not driven correctly



# Numbers in Verilog (ii)

- You can insert “\_” for readability
  - `12'b 000111010100`
  - `12'b 000_111_010_100`
  - `12'o 07_24`
- Bit extension
  - MSB = 0, x or z  $\Rightarrow$  extend this
    - `4'b x1 = 4'b xx_x1`
  - MSB = 1  $\Rightarrow$  zero extension
    - `4'b 1x = 4'b 00_1x`



# Basic Syntax of a Module

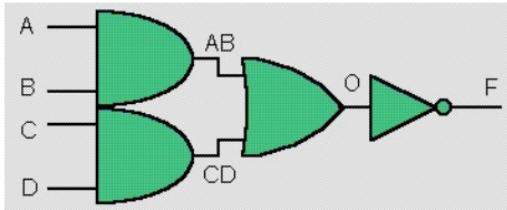
EQUIVALENT  
OF A C  
FUNCTION

**module** *module\_name* (*module\_port*, *module\_port*, ...);

Declarations:

list of ports for our design

*input, output, wire, reg, parameter.....*



System Modeling:

(describe the system in gate-level,

data-flow, or

behavioral style...)

**endmodule** ends a module – not a statement => no “;”



# Example

```
module full_adder (A, B, c_in, c_out, S);  
    // declarations contains the direction of the ports AND THE  
    types  
    // system description  
  
endmodule
```

*Direction can be omitted*



# Input/Output Declarations

## • Input Declaration

- Scalar ( $\leq$  Bit Ports)

- **input** list of input identifiers;
  - Example: input A, B, c\_in;

default 1 index Port

- Vector ( $\geq$  Bits Ports)

- **input[range]** list of input identifiers;
  - Example: input [15:0] A, B, data;

## • Output Declaration

- Scalar Example: **output** c\_out, OV, MINUS;

- Vector Example: **output** [7:0] ACC, REG\_IN, data\_out;



# Example

```
module full_adder (A, B, c_in, c_out, S);  
    // declarations  
  
    input [15:0] A, B;  
    input c_in; single bit value  
    output c_out; single bit value  
    output [15:0] S;  
  
    // system description  
  
endmodule
```



# Nets

- Can be thought as hardware wires driven by logic
- Equal z when unconnected
- Various types of nets
  - wire
  - wand (wired-AND)
  - wor (wired-OR)
  - tri (tri-state)

A NET CORRESPONDS TO AN HW WIRE.

WIRE ARE SIGNALS.



# Registers

Doesn't represent physically the

l implemented with RF in HW

- Variables that store values
- Do not represent real hardware but ..
- .. real hardware can be implemented with registers
- Only one type: `reg`

```
reg A, C; // declaration  
// assignments are always done inside a procedure  
A = 1;  
  
C = A; // C gets the logical value 1  
A = 0; // C is still 1  
C = 0; // C is now 0
```

- Register values are updated explicitly!!



# Vectors

- Represent buses

```
wire [3:0] busA;  
reg [1:4] busB;  
reg [1:0] busC; -> busC is 2 bits
```

- Left number is MSB (most significant bit)
- Slice management

busC = busA[2:1];

$$\Leftrightarrow \begin{array}{l} \text{busC[1]} = \text{busA[2]}; \\ \text{busC[0]} = \text{busA[1]}; \end{array}$$

MSB

- Vector assignment (**by position!!**)

$$\text{busB} = \text{busA}; \Leftrightarrow \left\{ \begin{array}{l} \text{busB[1]} = \text{busA[3]}; \\ \text{busB[2]} = \text{busA[2]}; \\ \text{busB[3]} = \text{busA[1]}; \\ \text{busB[4]} = \text{busA[0]}; \end{array} \right.$$



# Integer & Real Data Types

- Declaration

```
integer i, k; 32 bits  
real r;
```

- Use as registers (inside procedures)

```
i = 1; // assignments occur inside procedure  
r = 2.9;  
k = r; // k is rounded to 3
```

- Integers are not initialized!!
- Reals are initialized to 0.0



# Verilog Operators

- **Logical & Relational Operators**
- **Bitwise Operators**
- **Reduction Operators**
- **Shift Operators**
- **Concatenation:** {identifier\_1, identifier\_2, ...}
- **Replication Operators:** {n{identifier}}
- **Relational Operators**
- **Equality Operators**
- **Conditional Operator**
- **Arithmetic Operators**



# Logical Operators

- `&&` → logical AND
- `||` → logical OR
- `!` → logical NOT
- Operands evaluated to ONE bit value: `0`, `1` or `x`
- Result is ONE bit value: `0`, `1` or `x`

A = 6<sup>true</sup>;

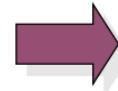
B = 0;

C = x;

A && B → 1 && 0 → 0

A || !B → 1 || 1 → 1

C || B → x || 0 → x



but C&B=0



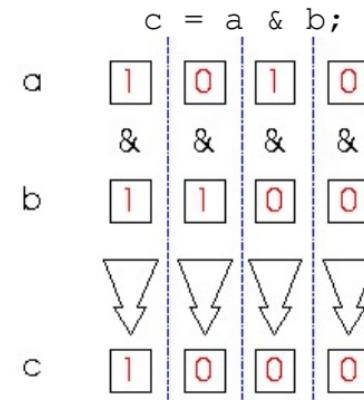
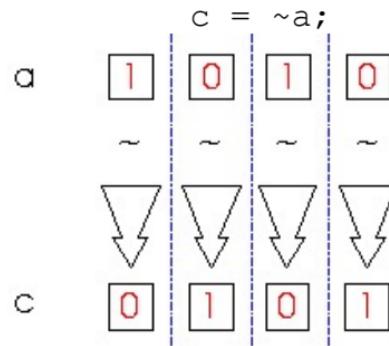
# Bitwise Operators (i)

- & → bitwise AND
  - | → bitwise OR
  - ~ → bitwise NOT
  - ^ → bitwise XOR
  - ~^ or ^~ → bitwise XNOR
- 
- Operation on a **bit-by-bit basis** *the operations are done bit to bit*



# Bitwise Operators (ii)

a = 4'b1010;  
b = 4'b1100;



# Reduction Operators

- `&` → AND
- `|` → OR
- `^` → XOR
- `~&` → NAND
- `~|` → NOR
- `~^ or ^~` → XNOR
- One multi-bit operand → One single-bit result

```
a = 4'b1001;  
..  
c = |a;           // c = 1|0|0|1 = 1    At least 1 bit true
```



# Shift Operators

- `>>` → shift right
- `<<` → shift left
- Result is same size as first operand, **always zero filled**

```
a = 4'b1010;  
...  
d = a >> 2; // d = 0010  
c = a << 1; // c = 0100
```



# Concatenation Operator

- `{op1, op2, ..}` → concatenates op1, op2, .. to single number
- Operands must be sized !!

```
reg a;  
reg [2:0] b, c;  
  
..  
a = 1'b 1;  
b = 3'b 010;  
c = 3'b 101;  
catx = {a, b, c};           // catx = 1_010_101  
caty = {b, 2'b11, a};       // caty = 010_11_1  
catz = {b, 1};              // WRONG !!
```

- Replication ..

```
catr = {4{a}, b, 2{c}};    // catr = 1111_010_101101
```



# Relational Operators

- $>$  → greater than
- $<$  → less than
- $\geq$  → greater or equal than
- $\leq$  → less or equal than
- Result is one bit value: 0, 1 or x

1 > 0                       $\rightarrow 1$   
'b1x1 <= 0               $\rightarrow x$   
10 < z                       $\rightarrow x$       The result is UNKNOWN



# Equality Operators

- `==` → logical equality } Return 0, 1 or  $x$
- `!=` → logical inequality }
- `====` → case equality }
- `!==` → case inequality }

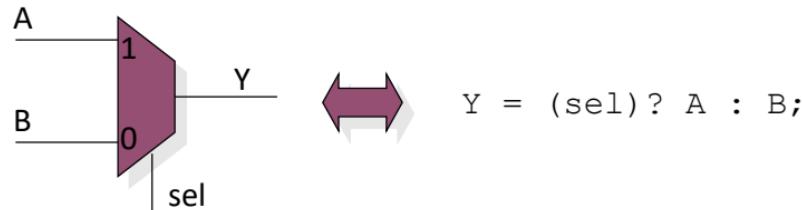
- $4'b\ 1z0x == 4'b\ 1z0x \rightarrow X$
- $4'b\ 1z0x != 4'b\ 1z0x \rightarrow X$
- $4'b\ 1z0x === 4'b\ 1z0x \rightarrow 1$
- $4'b\ 1z0x !== 4'b\ 1z0x \rightarrow 0$



# Conditional Operator

- `cond_expr ? true_expr : false_expr`

- Like a 2-to-1 mux ..



# Arithmetic Operators (i)

- $+, -, *, /, \%$
- If any operand is  $x$ , the result is  $x$
- Negative registers:
  - regs can be assigned negative but are treated as unsigned

```
reg [15:0] regA;  
.  
regA = -4'd12;           // stored as 216-12 = 65524  
regA/3                 evaluates to 21861
```



# Arithmetic Operators (ii)

- Negative integers:
  - can be assigned negative values
  - different treatment depending on base specification or not

```
reg [15:0] regA;  
  
integer intA;  
  
..  
  
intA = -12/3;      // evaluates to -4 (no base spec)  
  
intA = -'d12/3;   // evaluates to 1431655761 (base spec)
```



# Operator Precedence

+ - ! ~ unary	highest precedence
* / %	
+ - (binary)	
<< >>	
< <= => >	
== != === !==	
& ~ &	
^ ^~ ~^	
~	
&&	
? : conditional	lowest precedence

Use parentheses to  
enforce your  
priority



# Behavioral Modeling

- The behavior of a design is described using procedural constructs
  - initial statement: this statement executes only once.
  - always statement: this statement always executes in a loop forever.....
- Only register data type can be assigned in these statements



# Continuous Assignments

- Syntax:

```
assign #delay <id> = <expr>; delay is only for analysis (it's not synthesizable)
```

optional  
net type !!

- Where to write them:

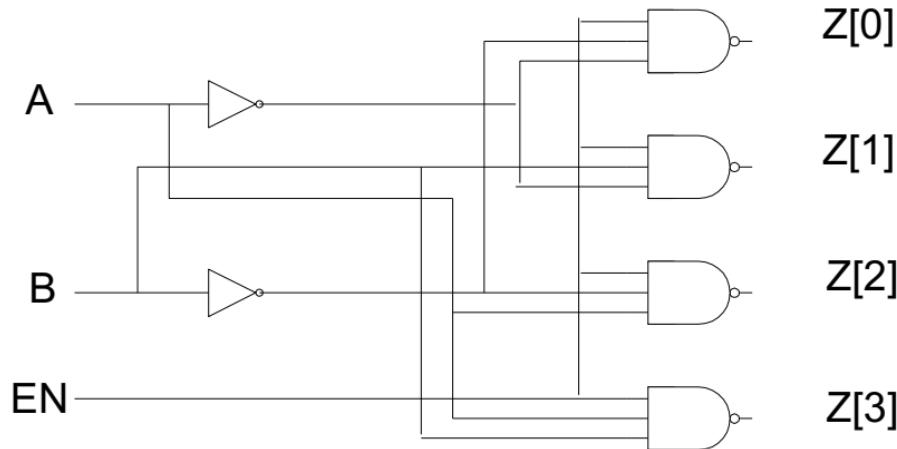
- inside a module
- outside procedures

- Properties:

- they all execute in parallel
- their order is independent (*irrelevant*)
- they are continuously active



# Example: 2 to 4 Decoder



# Example

```
module decoder_2_4(A,B,EN,Z);
    input          A,B,EN;
    output [0:3]   Z;
    wire           Ab, Bb;

    assign #1     Ab=~A;
    assign #1     Bb=~B;
    assign #2     Z[0]=~(Ab & Bb & EN);
    assign #2     Z[1]=~(Ab & B & EN);
    assign #2     Z[2]=~(A & Bb & EN);
    assign #2     Z[3]=~(A & B & EN);

endmodule
```



# Behavioral Model - Procedures (i)

- Procedures = sections of code that we know they execute sequentially
- Procedural statements = statements inside a procedure (they execute sequentially)
- e.g. another 2-to-1 mux implem:

Execution Flow



```
begin
    if (sel == 0)
        Y = B;
    else
        Y = A;
end
```

Procedural assignments:  
Y must be reg !!



# Behavioral Model - Procedures (ii)

- Modules can contain any number of procedures
- Procedures execute in parallel (in respect to each other) and ..
- .. can be expressed in two types of blocks:
  - **initial** → they execute only once
  - **always** → they execute for ever (until simulation finishes)



# “Initial” Blocks

- Start execution at the beginning of the execution (e.g., simulation) and finish when their last statement executes

```
module nothing;
```

```
initial
```

```
    $display("I'm first");
```

Will be displayed  
at sim time 0

```
initial begin
```

```
    #50;
```

```
    $display("Really?");
```

```
end
```

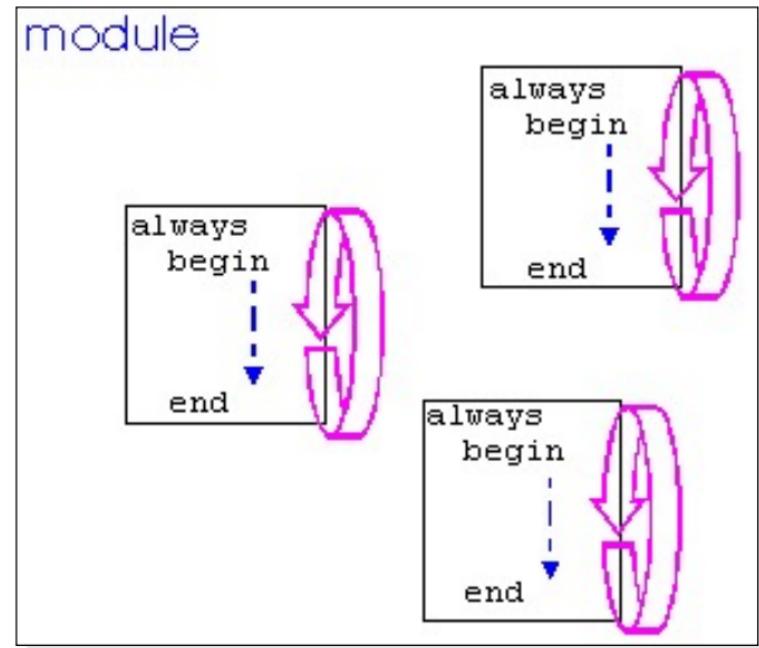
Will be displayed  
at sim time 50

```
endmodule
```



# “Always” Blocks

- Start execution at the beginning of the execution and continues indefinitely
- All always blocks execute in parallel



# Always Statement

- Syntax: **always**  
*#timing\_control procedural\_statement*

- Procedural statement is one of :

- Blocking Procedural\_assignment

**always**

**@ (A or B or Cin)**

**begin**

**T1=A & B;**

**T2=B & Cin;**

**T3=A & Cin;**

**Cout=T1 | T2 | T3;**

**end**

T1 assignment is occurs first, then T2, then T3....



# Events

- @

```
always @(signal1 or signal2 or ..) begin  
    ..  
end
```

execution triggers every time any signal changes

```
always @ (posedge clk) begin  
    ..  
end
```

execution triggers every time clk changes from *0* to *1*

```
always @ (negedge clk) begin  
    ..  
end
```

execution triggers every time clk changes from *1* to *0*



# Procedural Statements: if

```
if (expr1)
    true_stmt1;

else if (expr2)
    true_stmt2;

..
else
    def_stmt;
```

E.g. 4-to-1 mux:

```
module mux4_1(out, in, sel);
output out;
input [3:0] in;
input [1:0] sel;

reg out;
wire [3:0] in;
wire [1:0] sel;

always @ (in or sel)
    if (sel == 0)
        out = in[0];
    else if (sel == 1)
        out = in[1];
    else if (sel == 2)
        out = in[2];
    else
        out = in[3];
endmodule
```



# Procedural Statements: case

case (expr)

item\_1, .., item\_n: stmt1;

item\_n+1, .., item\_m: stmt2;

..

default: def\_stmt;

endcase

E.g. 4-to-1 mux:

```
module mux4_1(out, in, sel);
    output out;
    input [3:0] in;
    input [1:0] sel;

    reg out;
    wire [3:0] in;
    wire [1:0] sel;

    always @ (in or sel)
        case (sel)
            0: out = in[0];
            1: out = in[1];
            2: out = in[2];
            3: out = in[3];
        endcase
    endmodule
```



# Procedural Statements: for

```
for (init_assignment; cond;
      step_assignment)
      stmt;

E.g.

module count(Y, start);
output [3:0] Y;
input start;

reg [3:0] Y;
wire start;
integer i;

initial
    Y = 0;

always @ (posedge start)
    for (i = 0; i < 3; i = i + 1)
        #10 Y = Y + 1;
endmodule
```



# Procedural Statements: while

while (expr) stmt;

E.g.

```
module count(Y, start);
output [3:0] Y;
input start;

reg [3:0] Y;
wire start;
integer i;

initial
    Y = 0;

always @ (posedge start) begin
    i = 0;
    while (i < 3) begin
        #10 Y = Y + 1;
        i = i + 1;
    end
end
endmodule
```



# Procedural Statements: repeat

repeat (times) stmt;

Can be either an  
integer or a variable

E.g.

```
module count(Y, start);
    output [3:0] Y;
    input start;

    reg [3:0] Y;
    wire start;

    initial
        Y = 0;

    always @ (posedge start)
        repeat (4) #10 Y = Y + 1;
endmodule
```



# Procedural Statements: forever

It's useful to use a clock  
in a testbench

```
forever stmt;  
  
    ↓  
    Executes until sim  
    finishes
```

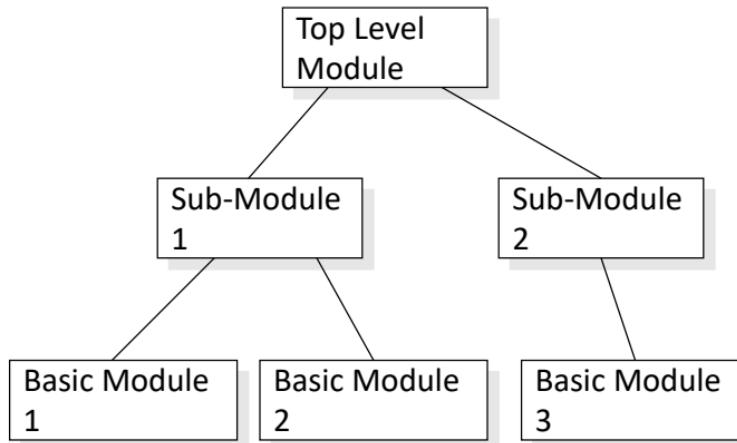
Typical example:  
*clock generation in test modules*

```
module test;  
  
    reg clk;  
  
    initial begin  
        clk = 0;  
        forever #10 clk = ~clk;  
    end  
  
    other_module1 o1(clk, ...);  
    other_module2 o2(..., clk, ...);  
  
endmodule
```

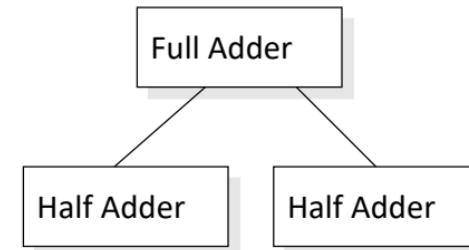
$T_{clk} = 20$  time units



# Hierarchical Design



E.g.



# Behavioral & Hierarchical Verilog

```
module addsub (A, B, R, sub) ;  
    input [3:0] A, B ;  
    output [3:0] R ;  
    input sub ;  
    wire [3:0] data_out;  
    add A1 (A, data_out, sub, R);  
    M1comp C1 (B, data_out, sub);  
endmodule
```

*Subcomponents*



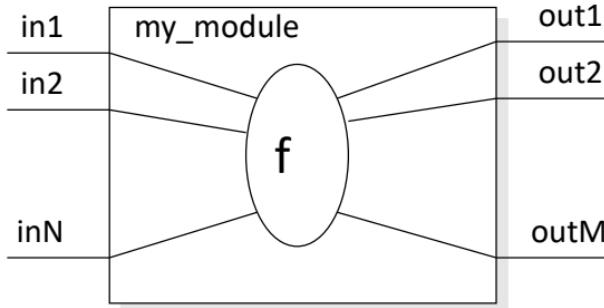
# Behavioral & Hierarchical Verilog

```
module add (X, Y, C_in, S);
    input [3:0] X, Y;
    input C_in;
    output [3:0] S;
    assign S = X + Y + {3'b0, C_in};
endmodule

module M1comp (data_in, data_out, comp);
    input[3:0] data_in;
    input comp;
    output [3:0] data_out;
    assign data_out = {4{comp}} ^ data_in;
endmodule
```



# Module



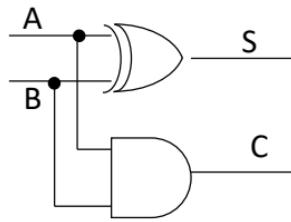
```
module my_module(out1, ..., inN);
    output out1, ..., outM;
    input in1, ..., inN;

    ... // declarations
    ... // description of f (maybe
    ... // sequential)

endmodule
```

Everything you write in Verilog must be inside a module  
exception: compiler directives

# Example: Half Adder



```
module half_adder(S, C, A, B);
output S, C;
input A, B;

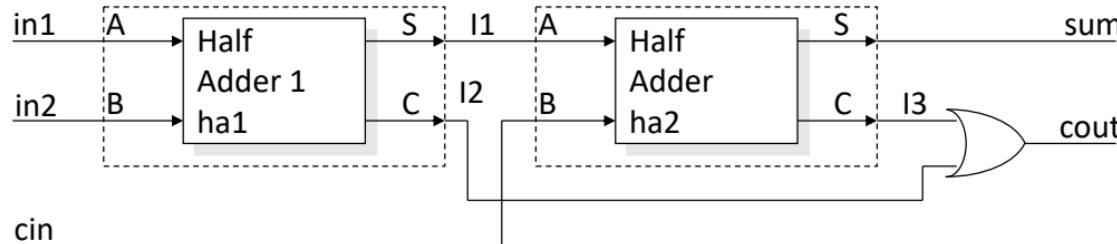
wire S, C, A, B;

assign S = A ^ B;
assign C = A & B;

endmodule
```



# Example: Full Adder



```
module full_adder(sum, cout, in1, in2, cin);
    output sum, cout;
    input in1, in2, cin;

    wire sum, cout, in1, in2, cin;
    wire I1, I2, I3;

    half_adder ha1(I1, I2, in1, in2);
    half_adder ha2(sum, I3, I1, cin);

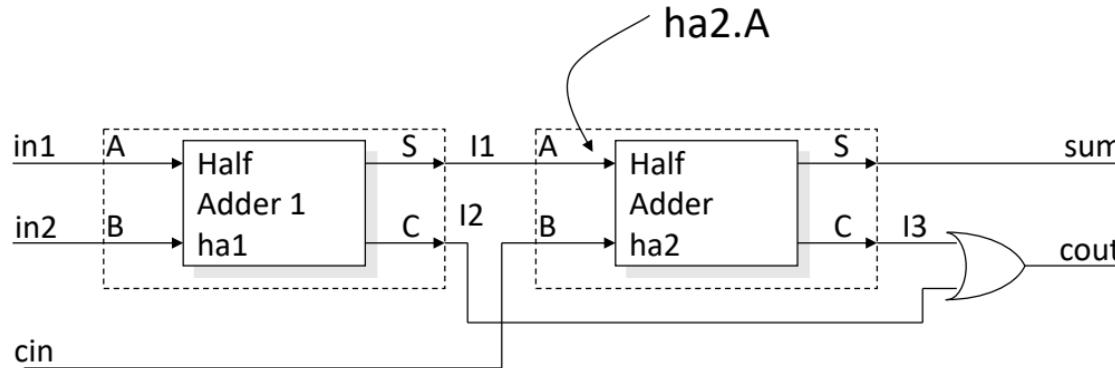
    assign cout = I2 || I3;

endmodule
```

Module name → half\_adder ha1(I1, I2, in1, in2);  
Instance name → half\_adder ha2(sum, I3, I1, cin);

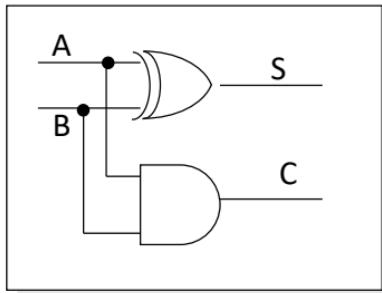


# Hierarchical Names



Remember to use instance names,  
not module names

# Example: Half Adder



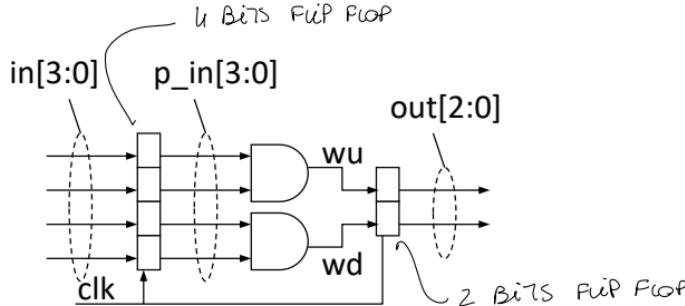
Assuming:

- XOR: 2 t.u. delay
- AND: 1 t.u. delay

```
module half_adder(S, C, A, B);  
    output S, C;  
    input A, B;  
  
    wire S, C, A, B;  
  
    xor #2 (S, A, B);  
    and #1 (C, A, B);  
  
endmodule
```



# Parameters



A. Implementation  
without parameters

```
module dff4bit(Q, D, clk);
output [3:0] Q;
input [3:0] D;
input clk;

reg [3:0] Q;
wire [3:0] D;
wire clk;

always @ (posedge clk)
Q = D;

endmodule
```

```
module dff2bit(Q, D, clk);
output [1:0] Q;
input [1:0] D;
input clk;

reg [1:0] Q;
wire [1:0] D;
wire clk;

always @ (posedge clk)
Q = D;

endmodule
```



# Parameters (ii)

A. Implementation  
without parameters (cont.)

```
module top(out, in, clk);
    output [1:0] out;
    input [3:0] in;
    input clk;

    wire [1:0] out;
    wire [3:0] in;
    wire clk;

    wire [3:0] p_in;      // internal nets
    wire wu, wd;

    assign wu = p_in[3] & p_in[2];
    assign wd = p_in[1] & p_in[0];

    dff4bit instA(p_in, in, clk);
    dff2bit instB(out, {wu, wd}, clk);
    // notice the concatenation!!

endmodule
```



# Parameters (iii)

## B. Implementation with parameters

```
moduledff(Q, D, clk);
parameter WIDTH = 4;
output [WIDTH-1:0] Q;
input [WIDTH-1:0] D;
input clk;

reg [WIDTH-1:0] Q;
wire [WIDTH-1:0] D;
wire clk;

always @ (posedge clk)
    Q = D;

endmodule
```

```
module top(out, in, clk);
output [1:0] out;
input [3:0] in;
input clk;

wire [1:0] out;
wire [3:0] in;
wire clk;

wire [3:0] p_in;
wire wu, wd;

assign wu = p_in[3] & p_in[2];
assign wd = p_in[1] & p_in[0];

dff instA(p_in, in, clk);
// WIDTH = 4, from declaration

dff instB(out, {wu, wd}, clk);
    defparam instB.WIDTH = 2;
// We changed WIDTH for instB only

endmodule
```



# Structural Model (Gate Level)

- Built-in gate primitives:

and, nand, nor, or, xor, xnor, buf, not, bufif0, bufif1, notif0, notif1

- Usage:

```
nand (out, in1, in2);  
and #2 (out, in1, in2, in3);  
not #1 N1(out, in);  
xor X1(out, in1, in2);
```

2-input NAND without delay

3-input AND with 2 t.u. delay

NOT with 1 t.u. delay and instance name

2-input XOR with instance name

- Write them inside module, outside procedures



# **Design of Hardware Accelerators**

Academic Year 2021/2022

## Questions?

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)



**POLITECNICO**  
MILANO 1863

**Design of Hardware Accelerators**  
Academic Year 2021/2022

# Open Challenges in the Design of Heterogeneous Systems

**Christian Pilato**

Dipartimento di Elettronica, Informazione e Bioingegneria

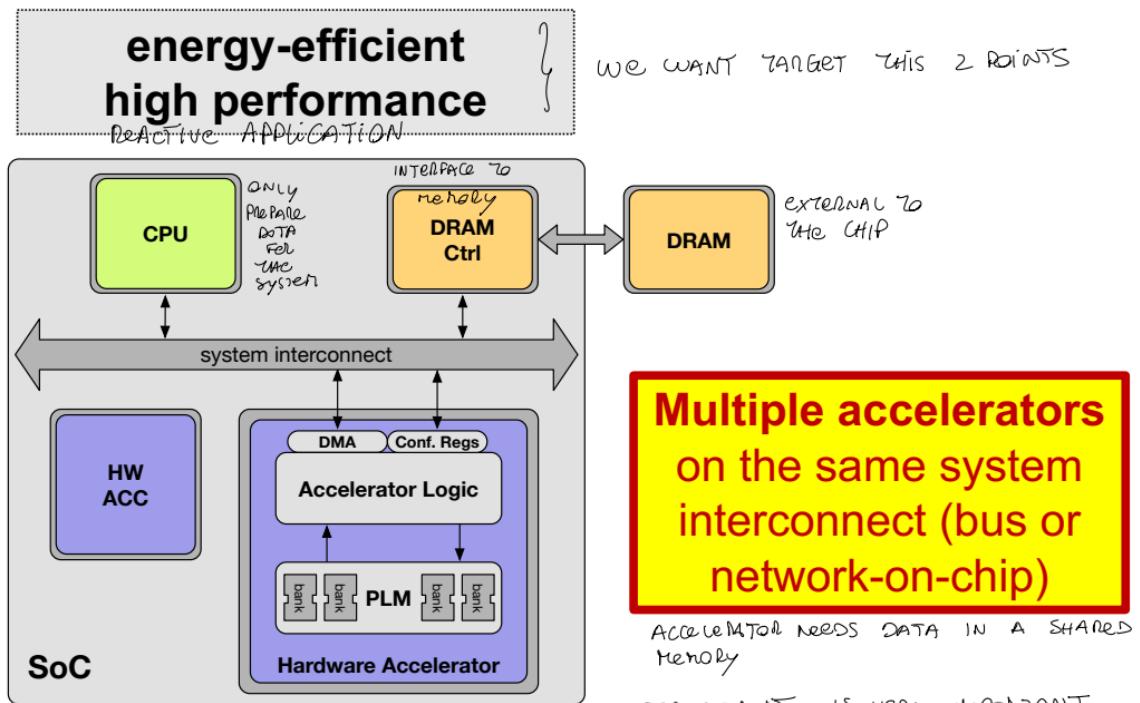
[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)

17 | 03 | 2022

# Abstraction of Heterogeneous SoCs

Specialized microarchitecture for both computation and storage

CPU to prepare the data in DRAM and control the accelerator (e.g., device driver)



Multiple accelerators on the same system interconnect (bus or network-on-chip)

Acceleration needs data in a shared memory

Scalability is very important  
More components and more issues

# Modern SoC Challenges

ARE INTERDEPENDENT

## System-Level Optimization

*Understand how to generate and optimize an efficient SoC architecture also given the technology constraints and the synthesis process*

UNDERSTAND THE OVERALL ARCHITECTURE

## Programmability

*Understand the interactions between hardware and software, and how to optimally control the component execution*

## Hardware Security

*Understand how to protect the effort done by a design house for creating a complex SoC*

PROTECTING AN ARCHITECTURE IS A COST.

TRade OFF Between DIFFERENT CHALLENGES



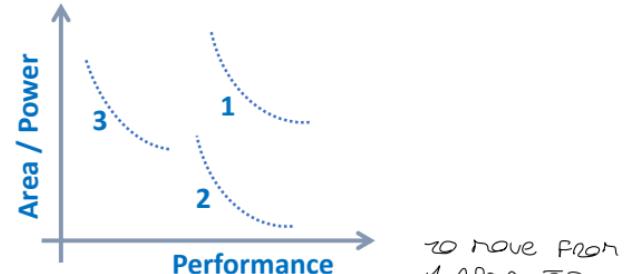
# From Software to Hardware

**Input language for system specification** is an open question

C/C++ are one of the possible sw languages

**High-Level Synthesis (HLS)** is the process of automatically generating hardware from a software description

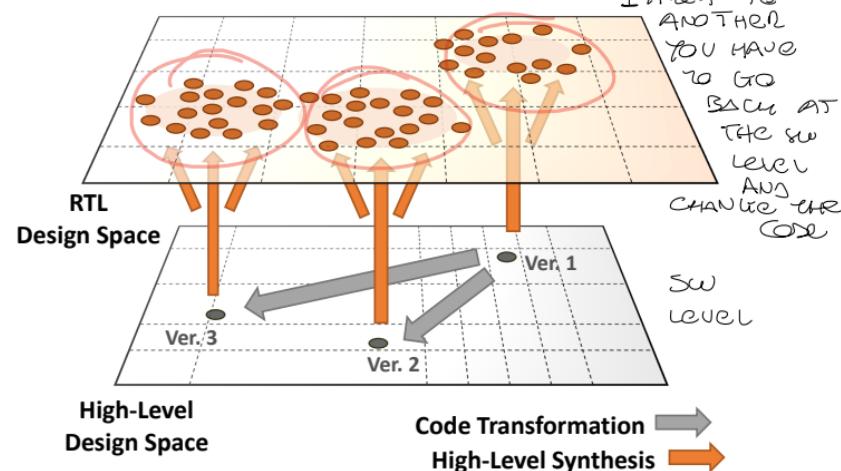
HLS HAS MANY ASSUMPTIONS AND CONSTRAINTS



Hardware generation is very dependent on **how the code is written**

- C does not express concurrency
- Arrays vs. pointers
- Multi-port interfaces
- .... IN HW YOU WANT CONCURRENCY

YOU HAVE TO KNOW C



# Programmer's View

## Combination of **Computational Functions** and **Data Structures**

- Different representations (and languages!) based on the **model of computation** and the **type of application**

easy to represent  
functions executed in series

### C FUNCTION

```
void Gsm_LPC_Analysis(word* so, word* LARc)
{
    longword L_ACF[9];
    Autocorrelation(so, L_ACF);
    Reflect_coeff(L_ACF, LARc);
    To_Log_Area_Rat(LARc);
    Quant_and_coding(LARc);
}
```

**Irregular applications:** pointer-based operations (arithmetic, dynamic resolutions, accesses to external memory, ...)

difficult to extract hw concepts  
easy to write

### C++ EXTENSION

you have the concept to represent concurrency  
INTRODUCTION OF HW DESIGN

```
SC_MODULE(debayer) {
    sc_in<bool> clk, rst;
private:
    int A0[6][2048];
    int B0[2048], B1[2048];
public:
    SC_CTOR(debayer) {
        SC_CTHREAD(Load, clk.pos());
        reset_signal_is(rst, false);
        SC_CTHREAD(Compute, clk.pos());
        reset_signal_is(rst, false);
        SC_CTHREAD(Store, clk.pos());
        reset_signal_is(rst, false);
    }
    //...
```

it is  
still  
SW but  
it's  
already  
hw  
design.

you can  
have  
concurrent  
functions.

no language  
is the  
best in  
general

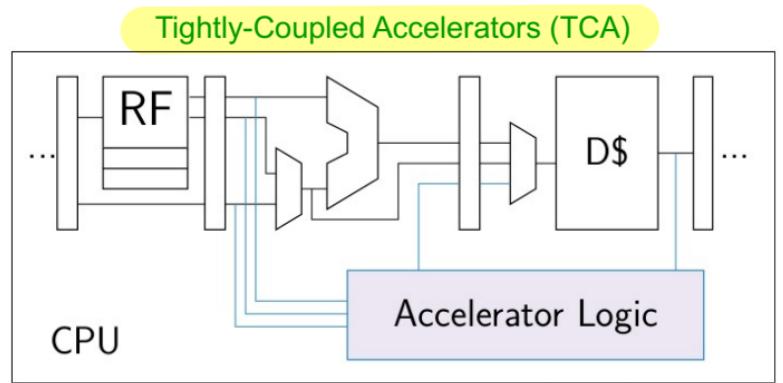
**Streaming applications:** data transfers to exchange data blocks with main memory



# Coprocessor Coupling in Heterogeneous SoCs

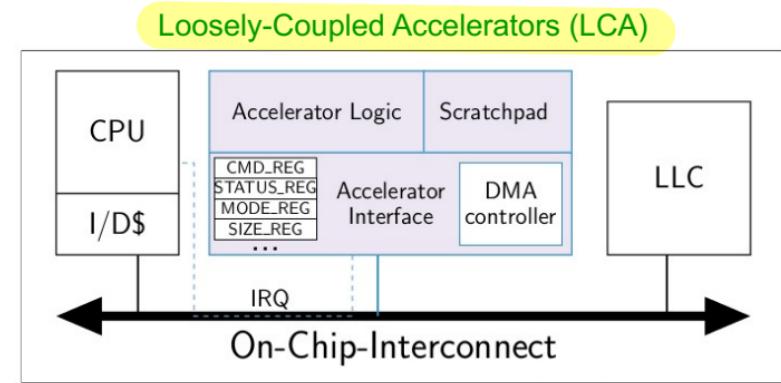
Two main models of **coupling coprocessors with processors**

- **Tightly-Coupled Accelerator (TCA)** shares key resources with the CPU: register file, MMU, and L1 cache
- **Loosely-Coupled Accelerator (LCA)** is outside the CPU and uses an integrated DMA controller to transfer data between their memory and the system memory



INTEGRATED IN THE MICRO ARCHITECTURE  
OF THE CPU.

it will be faster  
(2/3 cycles)



the Accelerator has its local  
memory and it's external to  
the CPU.

it can work independently and



LIMITED AMOUNT OF DATA  
Efficient for something that always happens

# DMA-Based Coprocessors

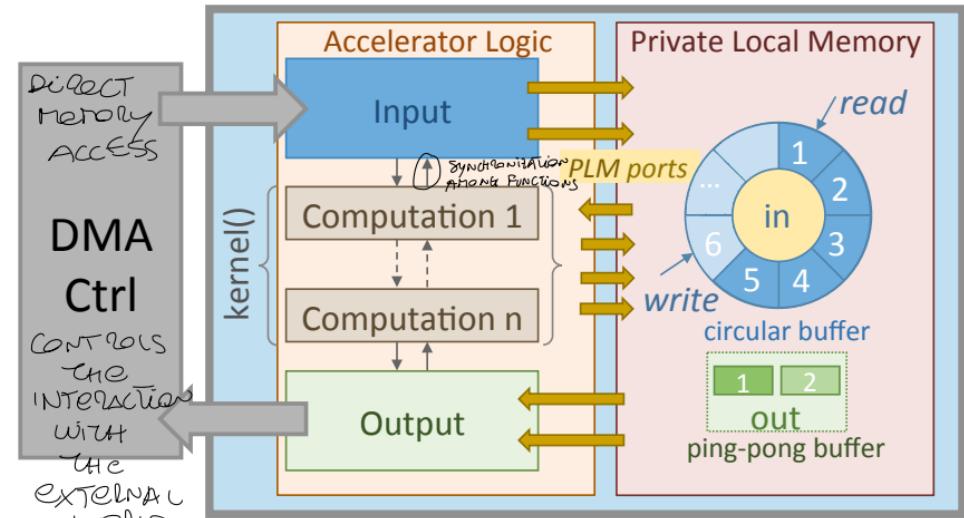
The CPU can work also if the Accelerator is stalled.

**Loosely-coupled accelerators** are more efficient in case of large data sets to be elaborated

- **Allocation:** How much data to store on-chip? Where to store the rest of the data?
- **Computation and Communication:** How to access the data efficiently? How to transfer the data efficiently?

When designing a component is **impossible to predict and optimize all situations** where it will be used

MAKING ASSUMPTIONS ALWAYS LEADS TO ERRORS



# Memory Accesses and Spatial Parallelism

Memory subsystem must be predesigned

## Distributed registers (e.g., flip-flops)

- Many ports at the cost of more area
- Good for small to medium data structures

1024x32 array in an industrial  
CMOS 32nm technology

Distributed registers

145,707.5  $\mu\text{m}^2$

## Memory Intellectual Property (IP) blocks

- Area-efficient macro blocks provided by the technology vendors
  - SRAMs for CMOS and BRAMs for FPGA
- Good for medium to large arrays
- Limited number of ports (usually no more than two!)

(SRAM)

Memory IP block

35,106.6  $\mu\text{m}^2$

(4x area reduction)

For your memory

Multi-bank architectures  
based on memory IPs

Technology constraints limit the parallelism

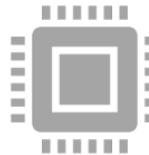


# Data Footprint Gap



**Algorithm data footprint** in rapidly increasing

Full HD image  
(1920x1080 pixels): ~33 Mb  
8K UHD image  
(7680x4320 pixels): ~530 Mb



**On-chip memory** is expensive and has limited capacity

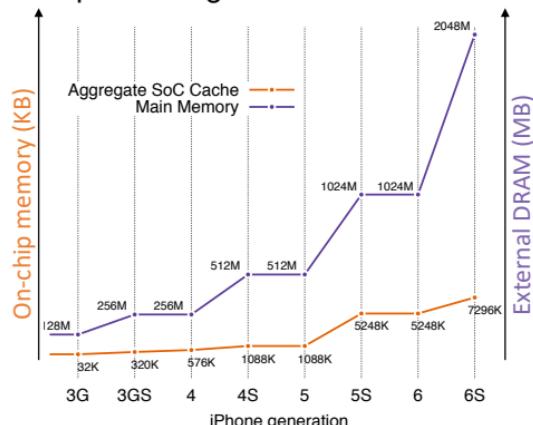
Largest available SRAM in our commercial 32nm CMOS technology is 8192x16 (~131 Kb)

1,030 available 32Kb BRAMs in the Xilinx Virtex-7 VC707 FPGA Evaluation Board (~33 Mb)



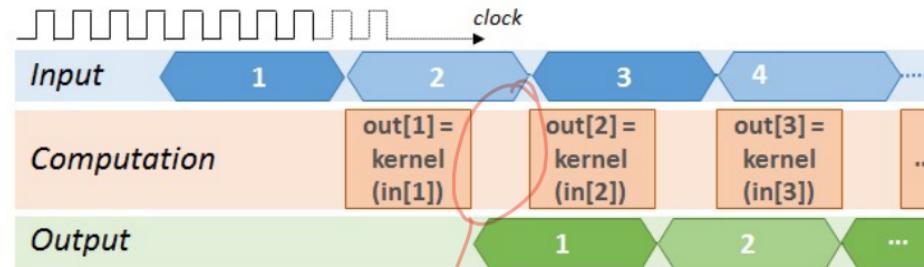
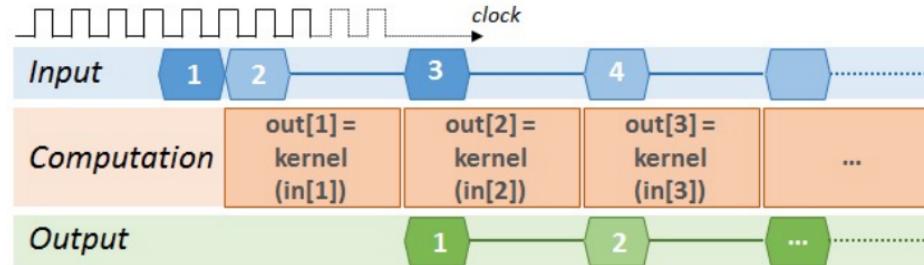
**Algorithm data footprint** is much larger than **on-chip data footprint**

Only part of the data can stay close to the processing elements



# Balancing Execution Phases

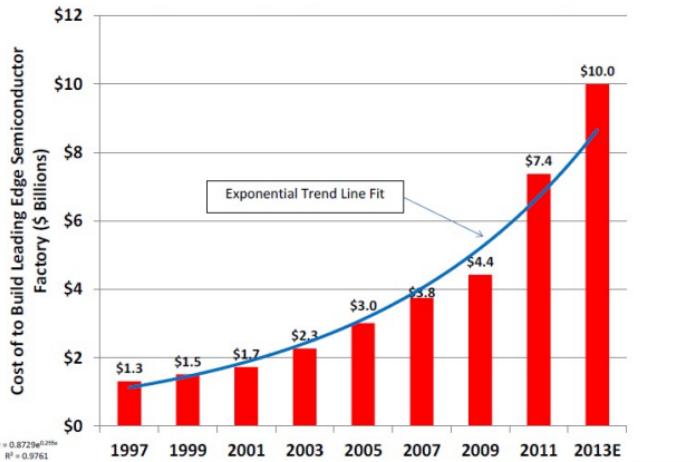
Each component with intense data accesses requires to  
**balance computation and communication**  
inefficient optimizations may create **execution stalls**



# Increasing Manufacturing Costs

Chips are becoming bigger and technology scaling is expensive!

- New lithographic instruments with large error margins



Chip design cycle becomes expensive with high variability

reduce design costs by reusing pre-designed components  
(programmability issues) *But you have programmability issues*

reduce manufacturing costs by outsourcing chip fabrication  
(security issues)

# Programmability Issues

You have 2 components:

Platform-based design: **orthogonalization of concerns**

## component generation



## system integration



### Key enabling properties

**modularity**  
**flexibility**  
**scalability**  
**reusability**

IF I USE HIGH LEVEL SYNTHESIS  
I DON'T WANT TO DO HW DESIGN.

INTEGRATE BUILDS THE ENTIRE SYSTEM

# Programmability Assets

**Modularity**: possibility of creating an SoC as a collection of components

- *Requires standardization of the interfaces, reducing cross-optimizations*

**Flexibility**: possibility of adapting one component to changes in the behavior of the others

- *Requires latency-insensitive protocols*

**Scalability**: possibility of creating larger SoCs without (significant performance degradations)

- *Requires scalable methods and transparent interconnection components*

**Reusability**: possibility of reusing pre-existing components

- *Requires system-level methods for retargeting components*

Concept of Re  
Synthesis

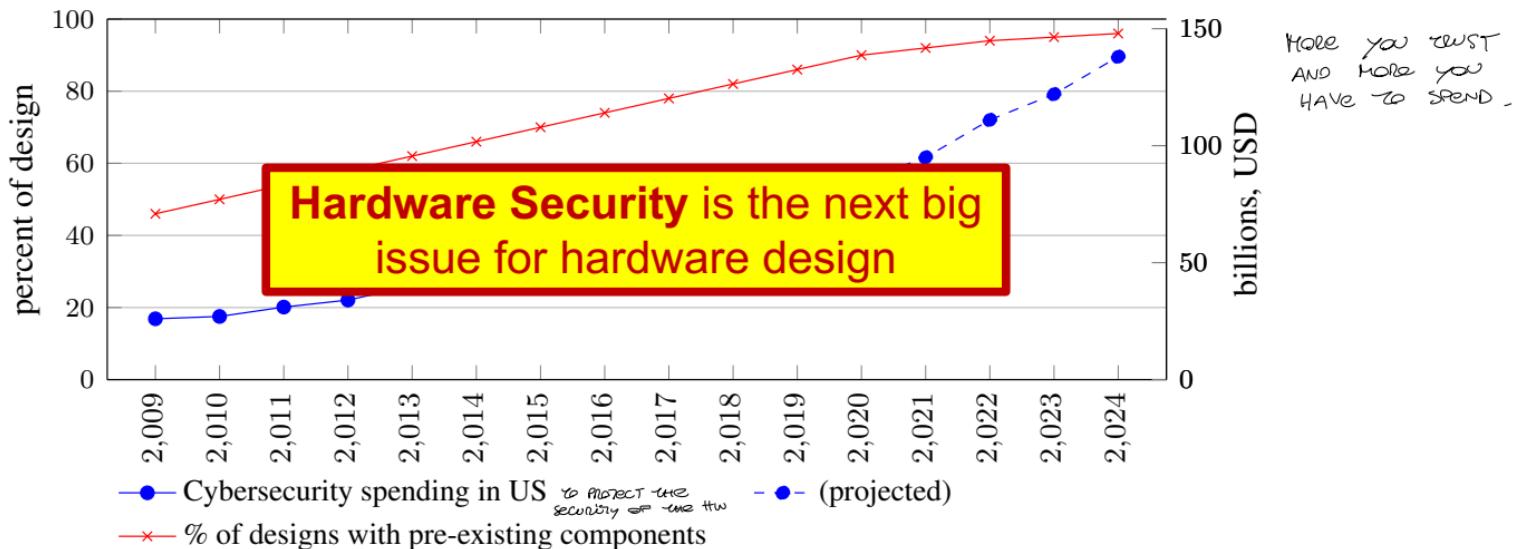


# System Complexity and Hardware Security

Components can be bought from other many vendors.

Increasing system complexity demands design & reuse approaches

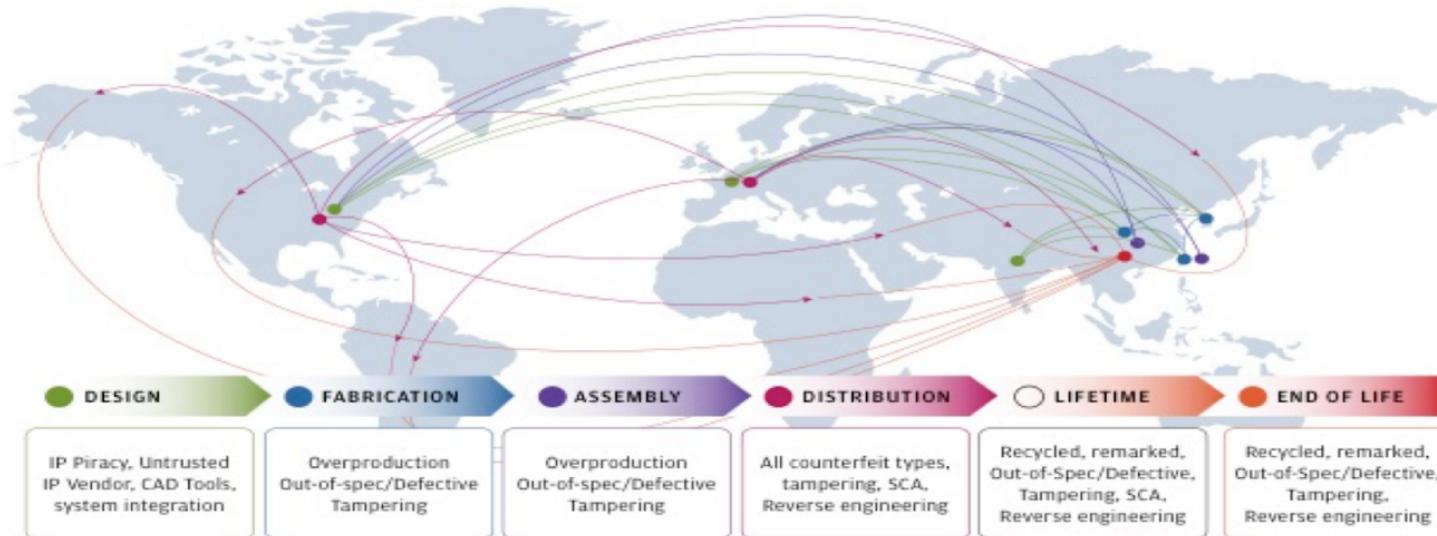
- IP components are coming from **many vendors** and assembled to create the SoC
- Most of the **design houses are fabless**



# Globalization of the Supply Chain

Supply chain is more and more distributed to **reduce costs**

- **Many security threats**
- Cost of addressing them is exponentially increasing from level to level



# IC/IP piracy and overbuilding

**Steal and claim ownership** of IC and/or illegal use

- Malicious SoC integration house
- Malicious foundry

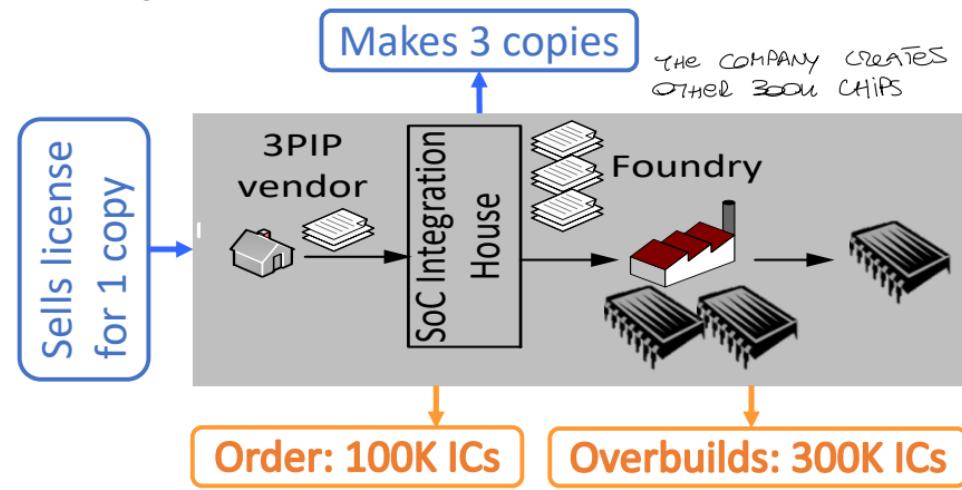
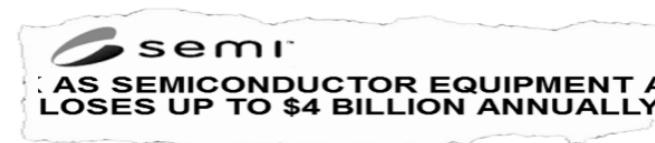
Real-life impact

- \$4,000,000,000 loss per year to IC industry
- ARM detected IP piracy in 2000

*They don't even know when they started losing money.*

**EE Times**

ARM files patent infringement suit against IP startup picoTurbo



# What to Protect?

YOU WANT TO PROTECT THE  
FINAL USER.



## How Sensitive Data is Elaborated by the System-on-Chip Architectures

Analysis of data elaboration to identify the hardware modifications to improve the overall security (also to prevent also software-based attacks)

**Hardware-assisted Security**

ASSISTED BY  
THE MODIFICATION  
OF THE HARDWARE

## Intellectual Property in the Design of Components and Architectures

Analysis of the digital design (component or architecture) to apply security protection methods against IP theft and counterfeiting

**Hardware Security**



# **Design of Hardware Accelerators**

Academic Year 2021/2022

## Questions?

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)



**POLITECNICO**  
MILANO 1863

**Design of Hardware Accelerators**  
Academic Year 2021/2022

# Latency-Insensitive Design

**Christian Pilato**

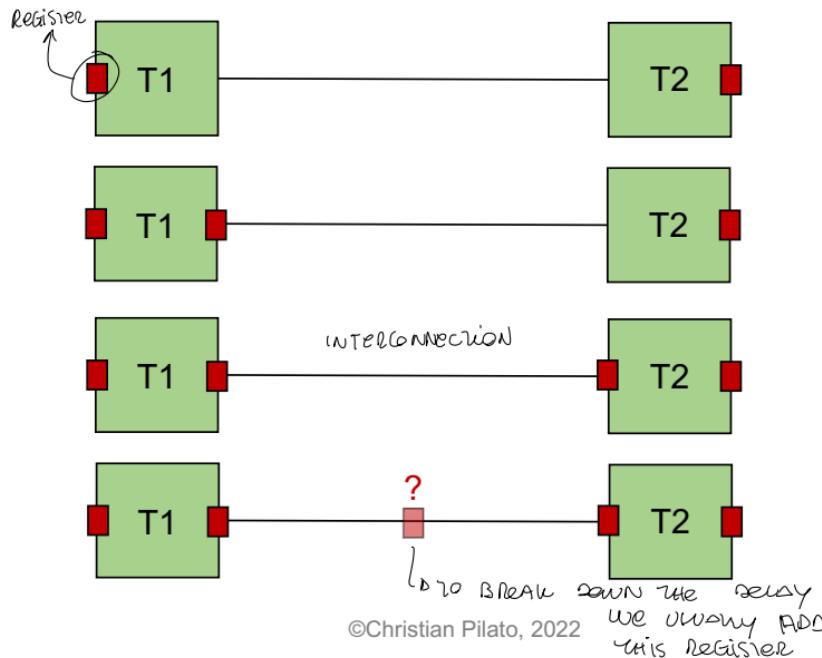
Dipartimento di Elettronica, Informazione e Bioingegneria

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)

We don't have the complete view of the system

# Circuit Timing

A system with **multiple components** works correctly as far as it is running with a **clock period** that is the **maximum** of the clock periods of the components (reg to reg)



# Communication Issues

In a **deep sub-micron process technology (<90nm)**, **process variability** is a serious concern

Technology is small and it's causes a lot of variability  
Components can have different delays

Technology improvements are on the transistors but not on the wires at the same level

- Inevitable dominance of wire delay

Wires is still something that you can reduce but it's not comparable to what you have on the transistor

**Long wire** will play significant role in **logic synthesis optimization**

- Interconnect Topology Optimization
- Optimal Buffer Insertion
- Optimal Wire Sizing
- But .. not everything can be rectified during interconnection optimization

try to reduce the distance between components  
more power consumption if you want faster interconnection

Need for a global «protocol» that is insensitive to delays

you don't have timing violation



# Latency-Insensitive Approach (i)

To design complex system in a correct way we need to:

- **relax time constraints** during early phases of the design
  - when correct measures of the inter-module delay paths are not available
- **simplify the composition** of sequential modules in pipeline mode
- **facilitate the insertion of extra pipeline stages** between one module and the next one with the purpose of buffering those signals which propagate on long wires



# Latency-Insensitive Approach (ii)

Idea of implementing a **latency-insensitive communication protocol**

Problem definition:

Given a **synchronous design** composed of communicating modules, how can we create a synchronous design that tolerates arbitrary communication latency

*it has the concept of clock  
you don't want to change the actual paradigm of the system*

No need to think of the digital system in a completely different way (e.g., as an asynchronous design)

- Possibility to reuse components designed for other systems



# Latency Insensitive vs. Asynchronous

Asynchronous systems require designer to think digital systems completely differently

- Remove the concept of global clock and create a complete event-based architecture with (complex) hand-shaking the second event starts when the first finishes

A latency-insensitive design is a **specified synchronous system**

- Components are still synchronous circuits

Delay insensitive circuit operates correctly regardless of delays on gates and wires

- Arbitrary delay is a multiple of the clock period

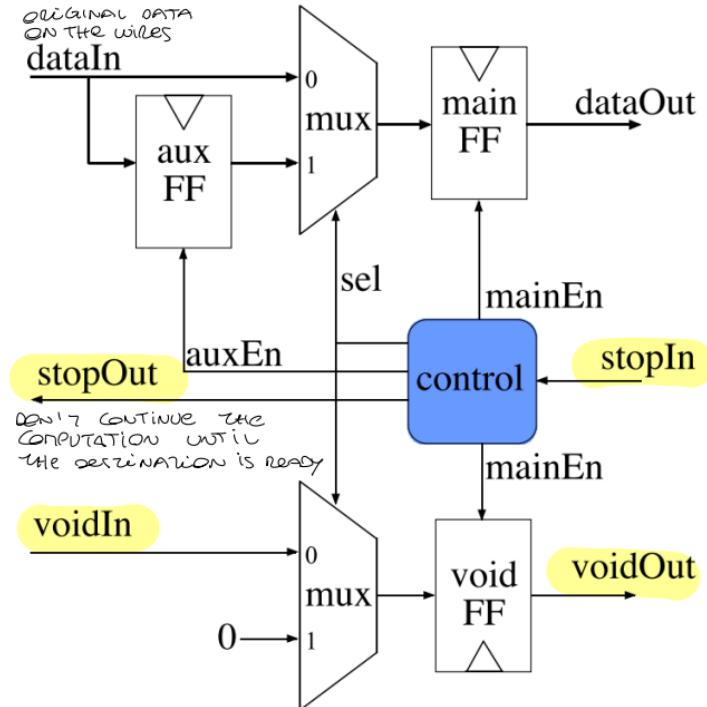
IF I ADD A STAGE IN THE MIDDLE I ADD A DELAY



# Latency-Insensitive Protocols

- Simple **communication protocol** for coping with transmission delays
  - Stalls the transmission if the data link is not ready (**stopIn**)
- Consecutive connection of control signals provides **backpressure**
  - Avoids computational error

Base element for scalable interconnections



# Latency-Insensitive Concepts

In a **Latency-Insensitive Design (LID)**, a design is correct if and only if **the sequence of the events (and not their timing) is correct**

- Timing is only a **non-functional metric** to evaluate the quality of the implementation

It introduces the following concepts:

- A **relay station (RS)** is a module that is inserted wherever it is necessary to tolerate delays *it's the concept of something that is in the middle*
- Each RS introduces one **stalling event** (non informative)  
*is event that tells you that the data that you have in that moment are valid or not (void)*
- A module receiving a stalling event as input emits stalling events as outputs at the next cycle



# Latency-Insensitive Theory

Given a complete synchronous specification of system composed a collection of modules, it aims at defining communication channels with relay stations

To manage exchanges of informative and stalling events between the relay stations, it encapsulates each module with a **shell**

It's connected not only  
to the DATA but it  
manages the STOP AND  
STAN connection

Traditional synchronous system:

- Layout obtained by standard Place&Route tools

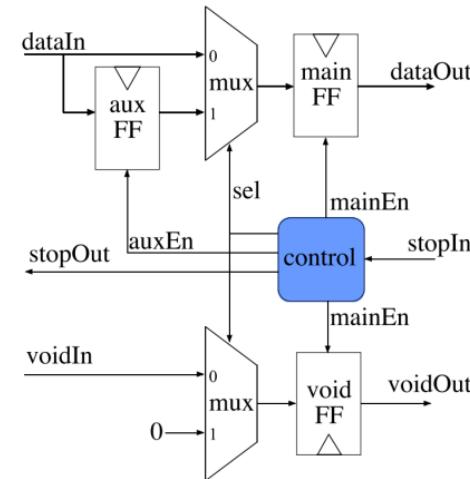
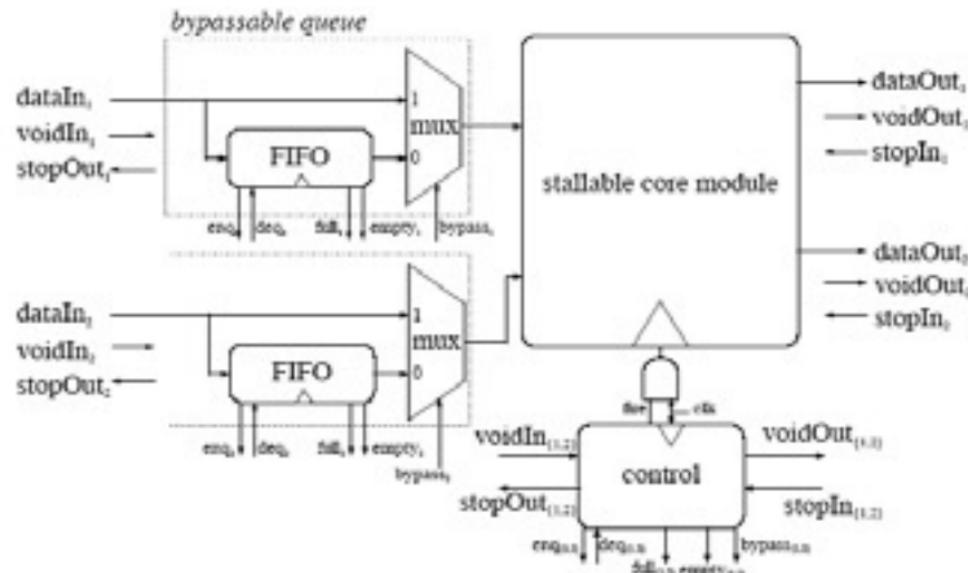
It can operate also as a **post-layout optimization**

- Necessary number of relay stations inserted into each critical channel



# Complete Latency-Insensitive System

Protocol that governs the exchange of information in a patient system



# Implementation Concepts (i)

- Channels are point-to-point unidirectional links between 2 components
  - Source/Sink Modules
    - source : produce the information
    - sink : receive information
  - More channels to connect more components (each of them may have a different latency)
- Packet Fields
  - Payload
  - Void
- True (Informative) Packets are the ones with **void = 0**
- Stalling Packets are the ones with **void = 1**



# Implementation Concepts (ii)

Data transmitted as **packets**

## Source

- Puts true packets (void0) or stalling packets (void1) on the channel

## Sink

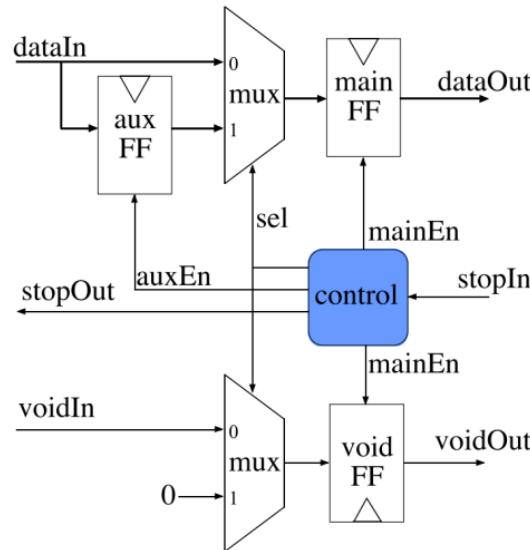
- Decides to store/discard the packet based on the void value
- If stalling (not ready to consume the packet), sends a **stop** flag
- Stop flag tells source that packet cannot be received (source becomes stalling)



# Relay Station

In some systems, **void (0/1)** is replaced by **valid (1/0)**

Source sends true/stallable packets only if sink is ready to receive ( $\neg stop$ )  
STOP IS zero



Sink consumes packets only if it is ready to receive them

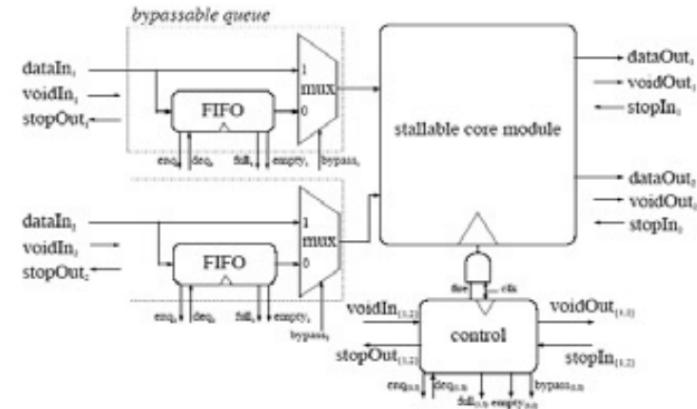
If one component is not ready, backpressure is propagated through the stop signals



# Shell

A shell is a **wrapper** that encapsulates **module M** and interfaces with channels so that M becomes a patient process

- Gets incoming packets from input channels
- Filters void packets
- After all input values are received, passes to M and activates computation
- Gets results of M when completed
- If no stop flag is received, sends result



Guarantee input synchronization and data propagation

- Internal computation fired only if all inputs have arrived (otherwise it is stalling)



# Abstraction Levels for LID

**Latency-insensitive design** is more a **paradigm** rather than an actual implementation

- It can be applied at different abstraction levels with the same concepts

During logic and physical design

- **Wire segmentation** to pipeline long wires

During system-level design

- System-level LID to match modules with unpredictable behavior



# Latency-Insensitive Physical Design

Similar to the procedure (and architecture) discussed before

Sequence of steps start from a collection of synchronous modules

- Synthesize layout and compute wirelength
- Extract parasitic capacitance to determine actual wire latency (delay of the wires)
- Segment every wire with latency greater than the clock period, and add relay stations
- Build shell around each module to obtain *patient processes*

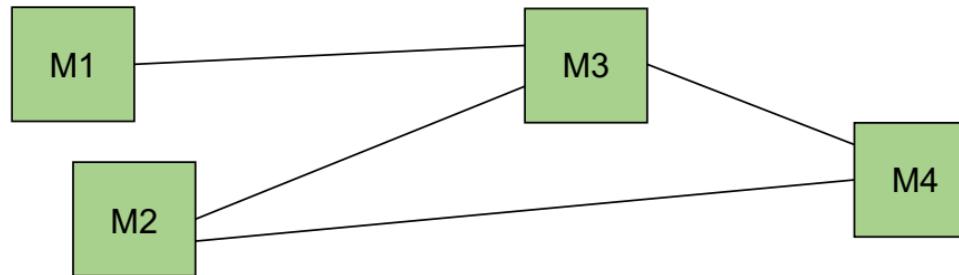
It only requires the **modules to be stallable** (i.e., *it can be interrupted and resumed in any moment with no side effects*)

you stop the execution and when you resume the execution there is no error



# System-Level LID

Propagating stalls through stop signals allows each module to start if and only if the data are available



Base concept to the creation of complex systems that can be optimized independently

- At design time, no need to know the latency of the others
- At run time, possibility to change the latency of one component without affecting the correctness of the overall computation

*You affect the wiring but not the correctness*

# LID and FIFO Buffers

A **First-in First-out (FIFO) buffer** is part of an **advanced latency-insensitive system**

- Enables the accumulation of more data tokens on a channel
- Empty/Full signals can be connected to the Void/Stop signals of the components

In particular:

- When a FIFO is empty, no data can be consumed by the sink module (stallable events)
- When the FIFO has at least one element, the sink module can consume a data token (informative event)
- When the FIFO is full, the source module sees the equivalent of a stop signal and cannot proceed



# Compositional System Level Design

Complex SoCs (or even single modules) can be designed to be synchronous but reactive to events (e.g., data availability)

- Paradigm used in

Design styles offer **synthesizable communication primitives** for data exchanges:

- A **blocking write** stalls the component until the operation can be performed
- It must match a corresponding **blocking read** on the other side

The tools implement these primitives with latency-insensitive logic (area overhead is generally less than 3% in both ASIC and FPGA technologies)

From now on we start to design a single component at a time  
Some one generates the components and someone uses the components



# **Design of Hardware Accelerators**

Academic Year 2021/2022

## Questions?

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)



**POLITECNICO**  
MILANO 1863

**Design of Hardware Accelerators**  
Academic Year 2021/2022

# Introduction to High-Level Synthesis

**Christian Pilato**

Dipartimento di Elettronica, Informazione e Bioingegneria

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)

26/03/2022 lecture

# Need for High-Level Design

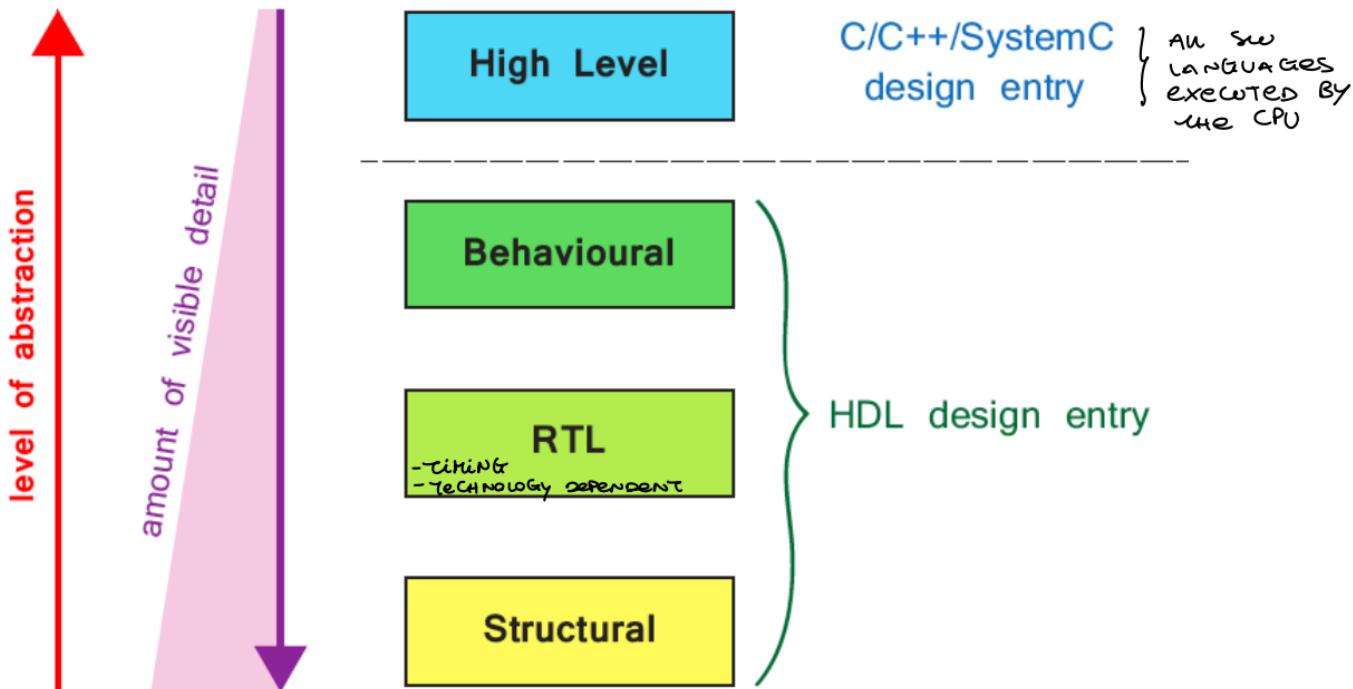
SW DESCRIPTION THAT  
CAN AUTOMATICALLY  
GENERATE THE HW

Working at **higher level of abstraction** allows designer to:

- Model complex designs
- Reduce the development time *writing sw is easy and faster than write at hw level*
- Simplify the code review (more experts at higher abstraction levels)
- Operate at technology independent level *you can write once the sw and then you can generate the hw for the technology that you want*
- Simplify (and explore) HW/SW partitioning



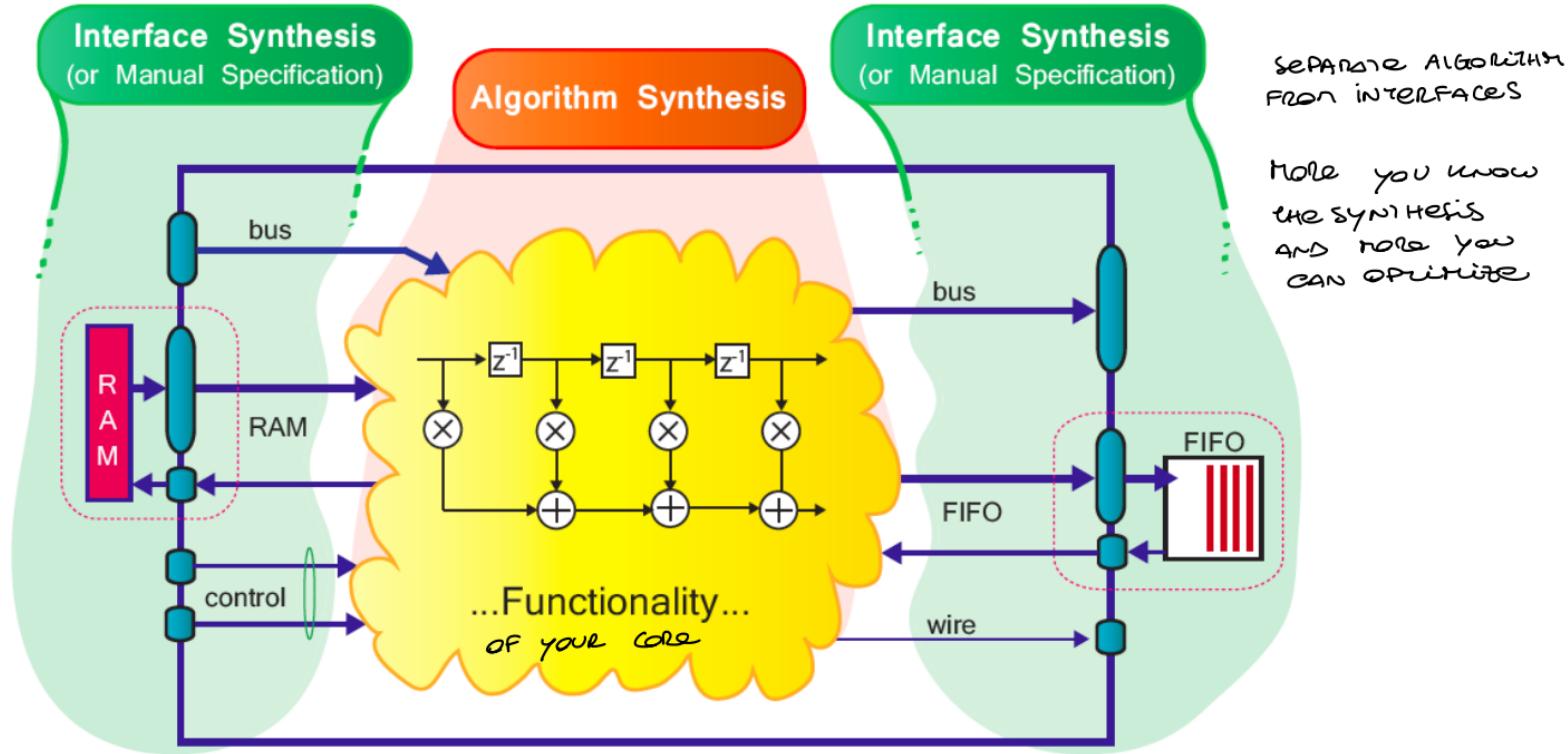
# Levels of Abstraction in FPGA Design



Source: The Zynq Book



# Algorithm and Interface Synthesis



Source: The Zynq Book

# High-Level Synthesis: HLS

## High-Level Synthesis

- Creates an RTL implementation from C, C++, System C, OpenCL API C kernel code *IS NOT really technology independent* *FROM THE SAME CODE YOU CAN DERIVE DIFFERENT IMPLEMENTATION ON'S*
- Extracts control and dataflow from the source code
- Implements the design based on defaults and user applied directives

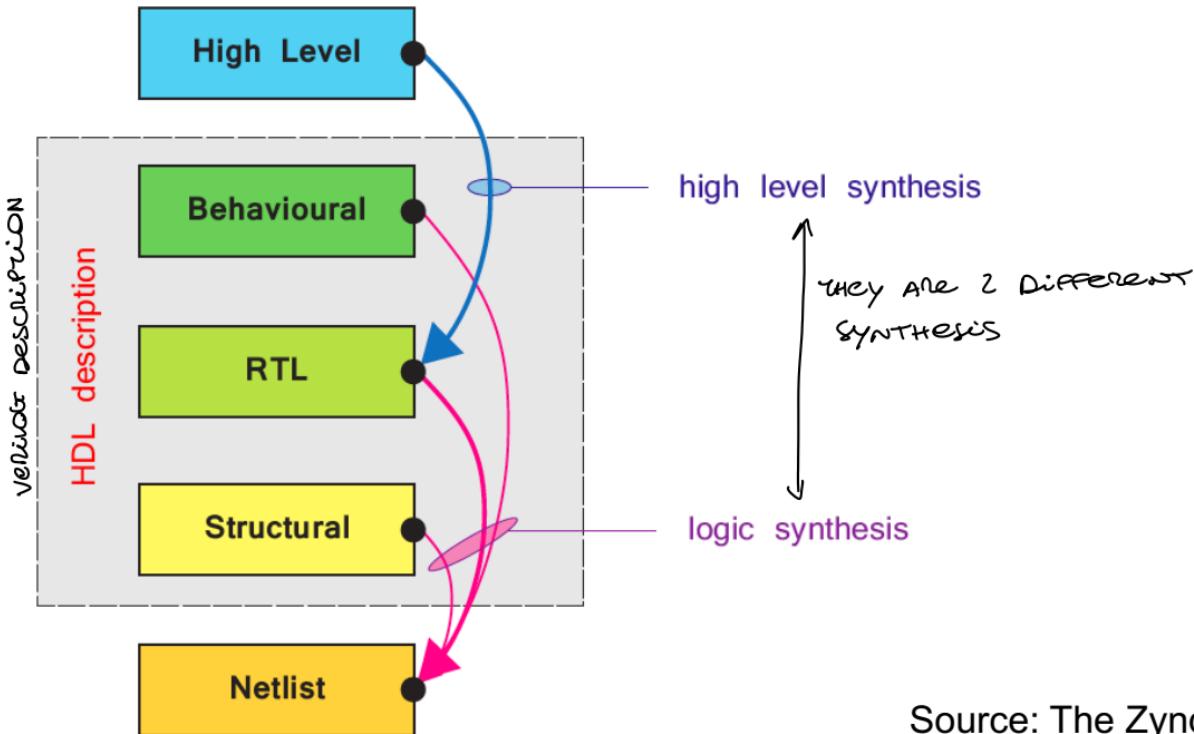
## Many implementation are possible from the same source description

- Smaller designs, faster designs, optimal designs – Enables design exploration

*UNDERSTAND THE FUNCTIONALITY OF THE ACCELERATOR THAT YOU WANT TO GENERATE  
// THE EVOLUTION OF THE CONTROL FLOW AND OF THE DATA*

*DEFAULT ASSUMPTION → INFINITE RESOURCES*

# High-Level Synthesis vs. Logic Synthesis



# Advantages of HLS

HIGH LEVEL SYNTHESIS

## Productivity

- Easy to model higher level of complexities
- Smaller in size source compared to RTL code
- Generates RTL much faster than manual method

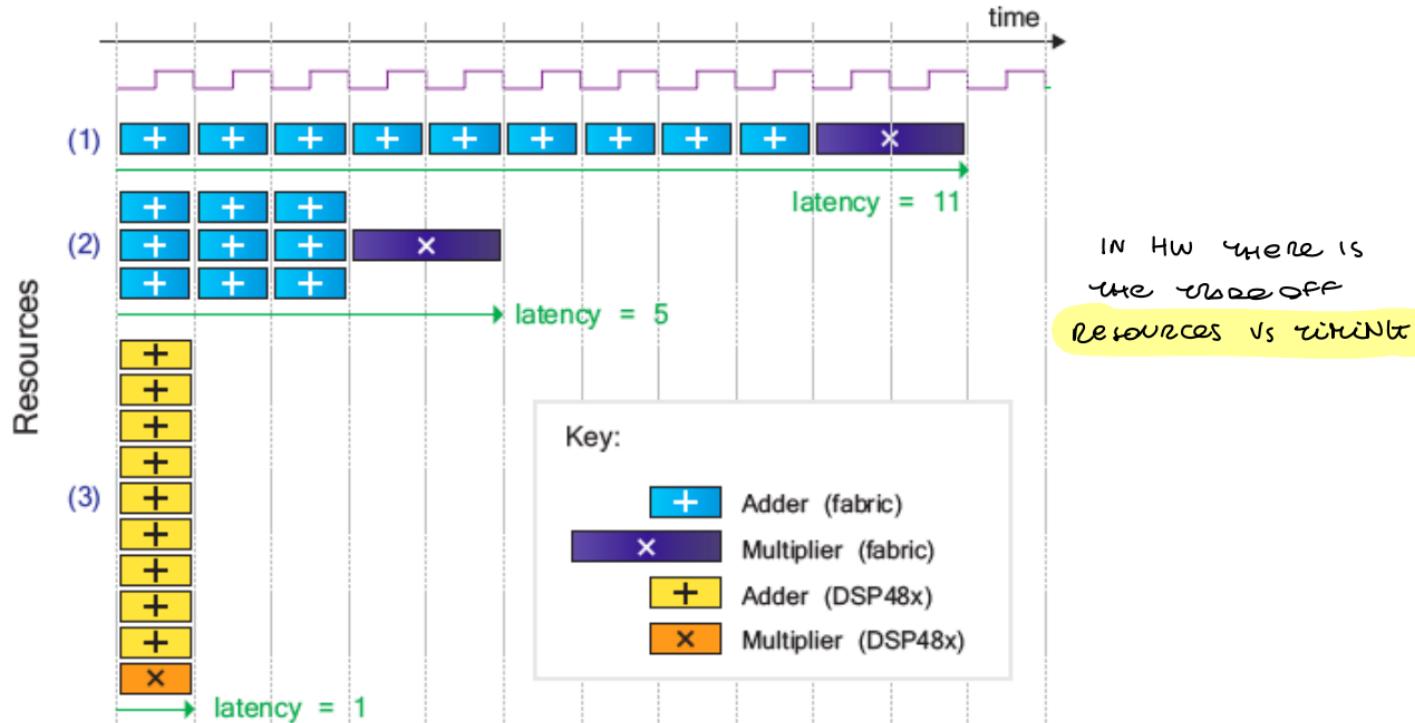
THE GENERATION OF RTL IS  
MADE BY A TOOL

## Quality of Results

- Automatic parallelism extraction
- Multi-cycle functionality
- Loop Optimization
- Optimization of Memory Access

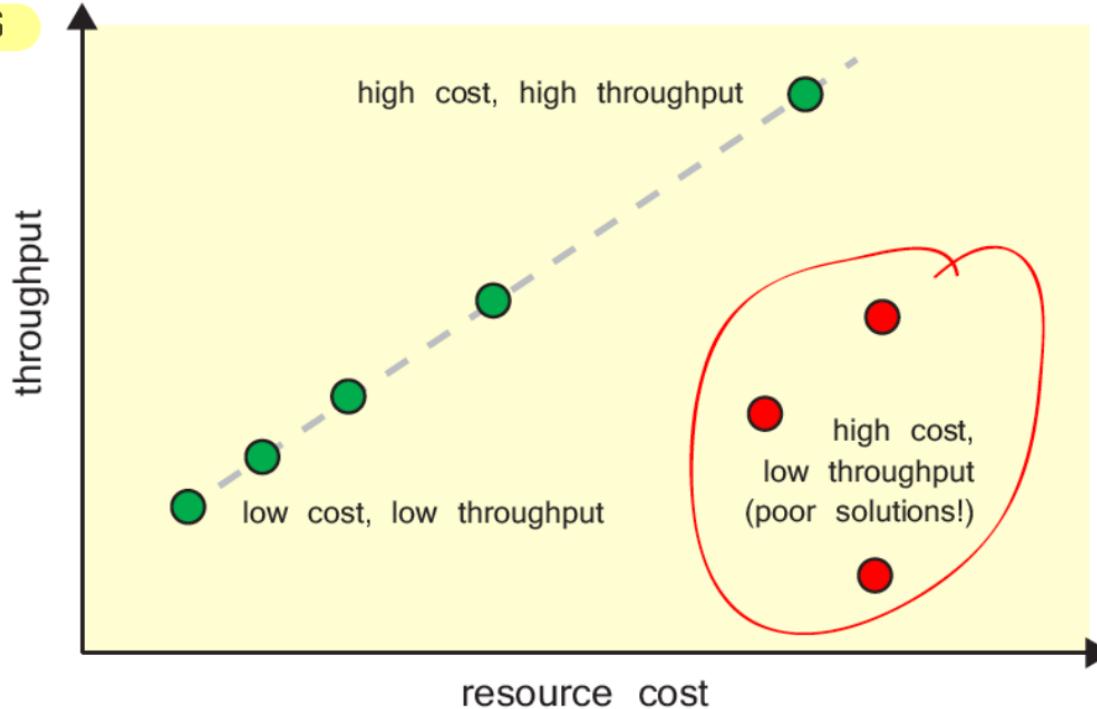


# Alternatives from HLS – Avg. of 10 numbers



# Design Trade-offs Explored Using HLS

**Pareto dominance:** one point dominates another if it is equal or better for all objectives



# Short History of High-Level Synthesis

- **Generation 1** (1980s-early 1990s): research period
- **Generation 2** (mid 1990s-early 2000s):
  - Commercial tools from Synopsys, Cadence, Mentor Graphics, etc.
  - **Input languages: behavioral HDLs**
  - Target: ASIC
  - Outcome: Commercial failure
- **Generation 3** (from early 2000s):
  - Domain oriented commercial tools: in particular for DSP
  - **Input languages: C, C++, C-like languages (Impulse C, Handel C, etc.), Matlab + Simulink, Bluespec**
  - Target: FPGA, ASIC, or both
  - Outcome: First success stories



# Vivado HLS: Cinderella Story

AutoESL Design Technologies, Inc. (25 employees)

Flagship product:

AutoPilot, translating **C/C++/System C** to **VHDL or Verilog**

- Acquired by the biggest FPGA company, Xilinx Inc., in 2011
- AutoPilot integrated into the primary Xilinx toolset, Vivado, as  
**Vivado HLS, released in 2012**

**“High-Level Synthesis for the Masses”**



# LegUp – Academic Tool for HLS

- **Open-source HLS Tool**
  - Developed at the University of Toronto
  - Faculty supervisors: Jason H. Anderson and Stephen Brown
  - FPL Community Award 2014
- **High-Level Synthesis from C to Verilog**
- **Targets Altera FPGAs (extension to Xilinx relatively simple)**
- **Two flows**
  - Pure Hardware
  - Hardware/Software Hybrid
    - = Tiger MIPS + hardware accelerator(s) + Avalon bus + shared on-chip and off-chip memory

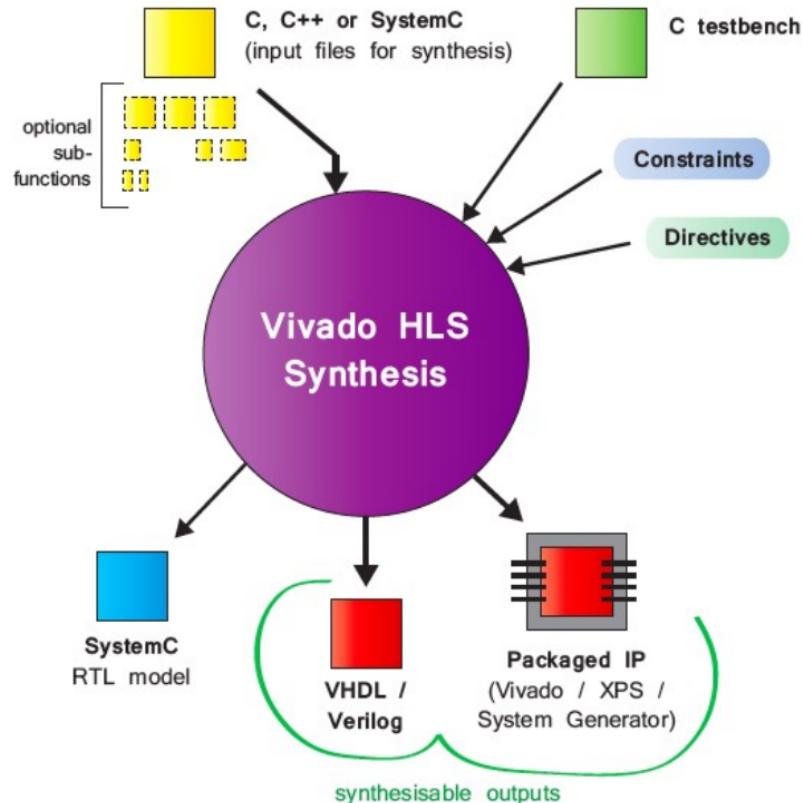


# Bambu – Academic Tool for HLS

- **Open-source HLS Tool**
  - Under development at Politecnico di Milano
  - Faculty supervisor: Fabrizio Ferrandi
- **High-Level Synthesis from C/C++ to Verilog/VHDL**
- **Targets both ASIC and FPGA**
  - Automatic generation of testbenches
- **Support for the implementation of custom “passes”**
  - Special algorithms for synthesis steps *CUSTOMIZATION OF THE FUNCTIONALITIES*
  - Hardware security
  - Debugging
  - ...



# Vivado HLS Synthesis Process

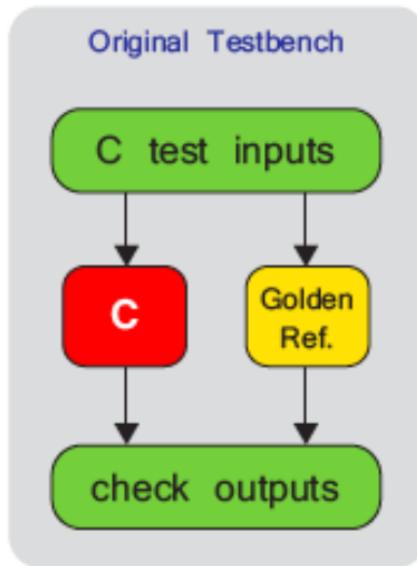


Source: The Zynq Book



# Functional Verification and Cosimulation

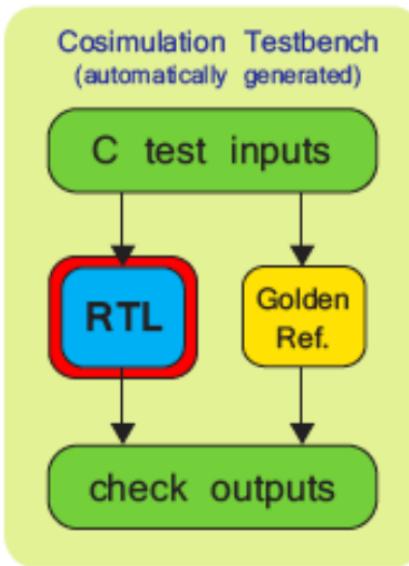
## Functional Verification



Vivado HLS  
C/RTL Cosimulation  
Process

A purple arrow points from the Functional Verification section to the C/RTL Cosimulation section.

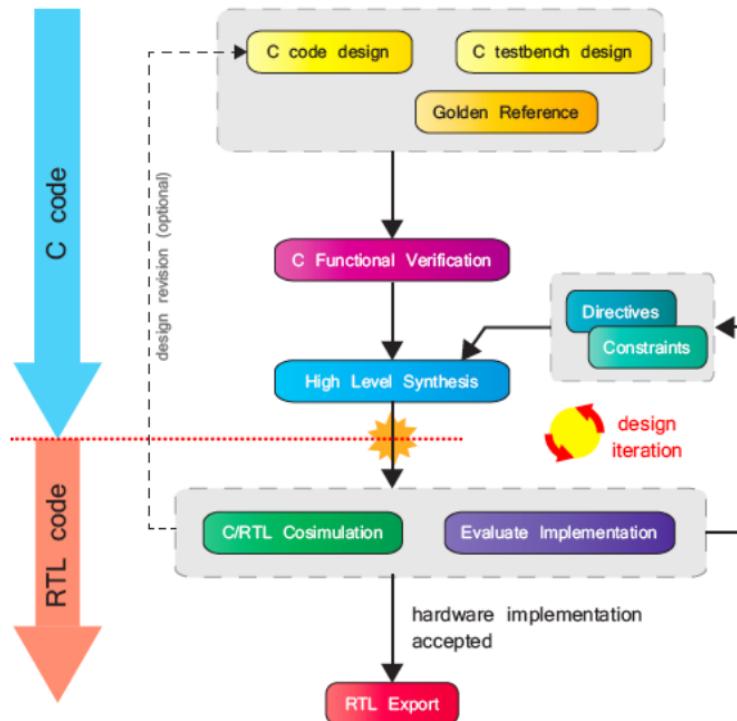
## C/RTL Cosimulation



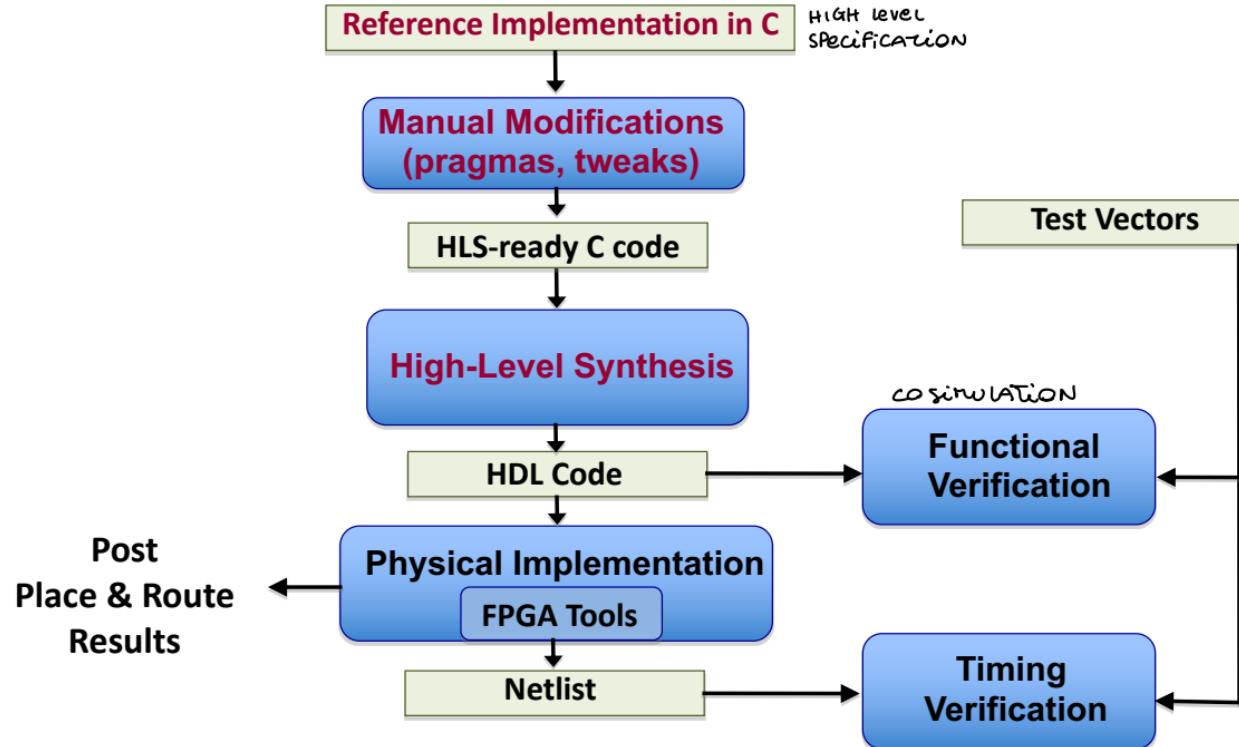
Source: The Zynq Book



# Vivado HLS Design Flow



# Development and Benchmarking Flow



# Vivado HLS

## ■ Starts at C

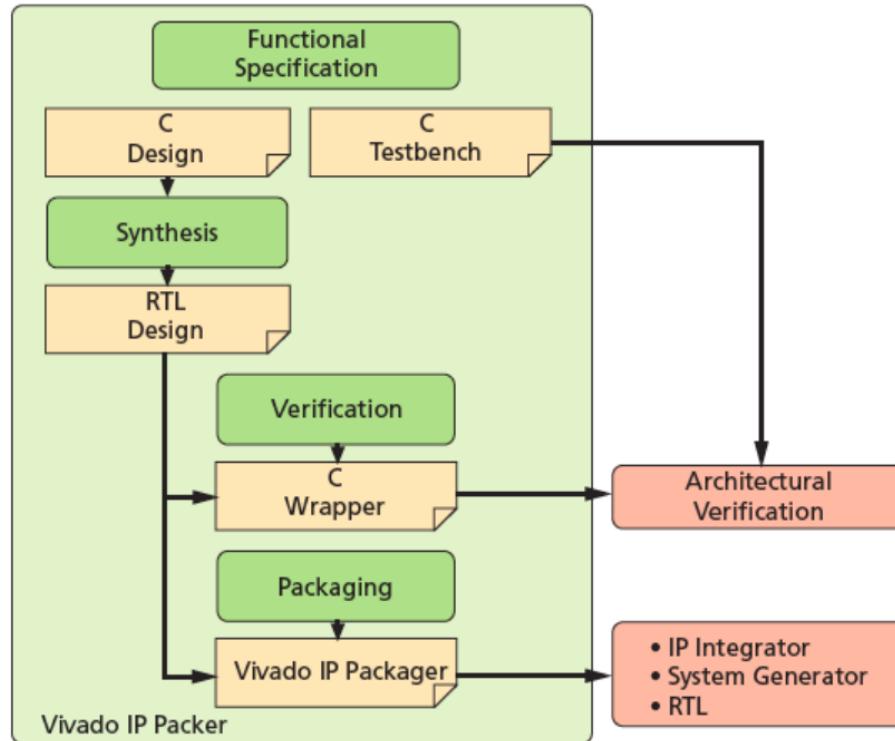
- C
- C++
- SystemC

## ■ Produces RTL

- Verilog
- VHDL
- SystemC

## ■ Automates Flow

- Verification
- Implementation



# Introduction to High-Level Synthesis

## How is hardware extracted from C code?

- Control and datapath can be extracted from C code for each function
- At some point in the top-level control flow, control is passed to a sub-function
- Sub-function may be implemented to execute concurrently with the top-level and or other sub-functions (e.g., Load/Compute/Store)

## How is this control and dataflow turned into a hardware design?

- Vivado HLS maps this to hardware through scheduling and binding processes

## How is my design created?

- How functions, loops, arrays and IO ports are mapped?



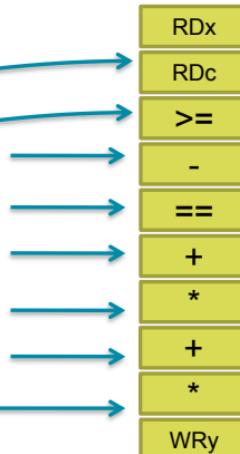
# HLS: Control & Datapath Extraction

## Code

```
void fir (
    data_t *y,
    coef_t c[4],
    data_t x
) {
    static data_t shift_reg[4];
    acc_t acc;
    int i;

    acc=0;
    loop: for (i=3;i>=0;i--) {
        if (i==0) {
            acc+=x*c[0];
            shift_reg[0]=x;
        } else {
            shift_reg[i]=shift_reg[i-1];
            acc+=shift_reg[i]*c[i];
        }
    }
    *y=acc;
}
```

## Operations



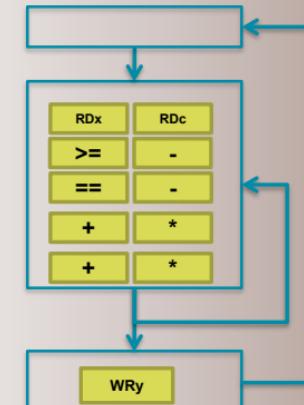
## Control Behavior

Finite State Machine (FSM)  
states



## Control & Datapath Behavior

Control Dataflow



From any C code example ..

Operations are extracted...

The control is known

A unified control dataflow behavior is created.



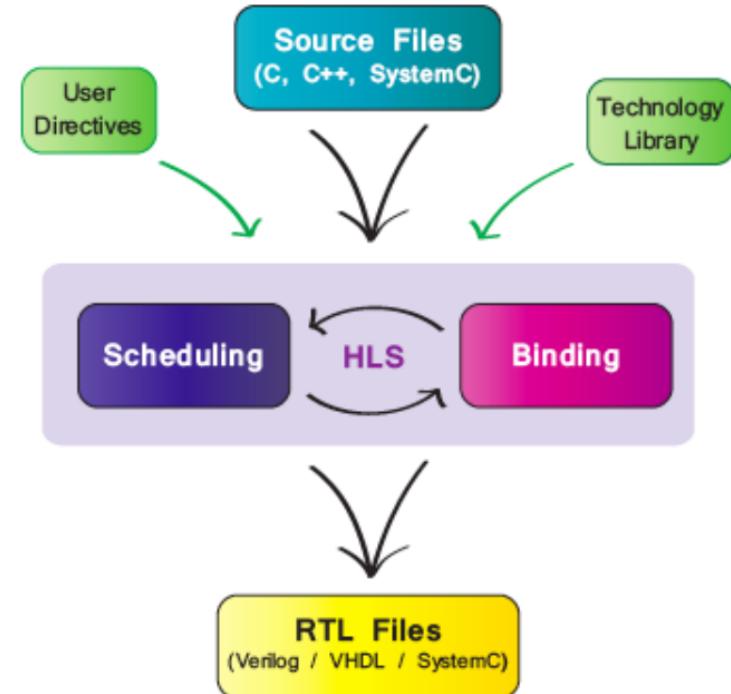
# Scheduling and Binding: Heart of HLS

**Scheduling** determines in which clock cycle an operation will occur

- Takes into account control, dataflow, and directives
- Resource allocation can be constrained

**Binding** determines which functional unit is used for each operation

- Takes into account component delays, directives
- Includes functional units, registers, etc.

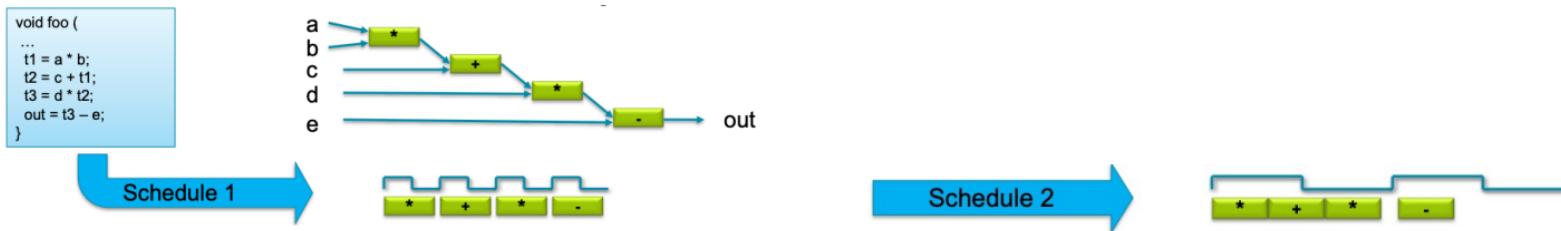


# Scheduling

Operations in the control flow graph are mapped into clock cycles

The technology and user constraints impact the schedule

- A faster technology (or slower clock) may allow more (or less) operations to occur in the same clock cycle



The code also impacts the schedule

- Code implications and data dependencies must be obeyed



# Binding

Binding is where operations are mapped onto physical library units

Binding Decision: to share or not to share

- Given this schedule:



- Binding must use 2 multipliers, since both are in the same cycle
- It can decide to use an adder and subtractor or *share* one addsub

- Given this schedule:



- Binding may decide to share the multipliers (each is used in a different cycle)
- Or it may decide the cost of sharing (muxing) would impact timing and it may decide *not to share* them
- It may make this same decision in the first example above too



# High-Level Concepts

- **Functions:** All code is made up of functions which represent the design hierarchy: the same in hardware
- **Top Level IO :** The arguments of the top-level function determine the hardware RTL interface ports
- **Types:** All variables are of a defined type (even custom). The type can influence the area and performance
- **Loops:** Functions typically contain loops. How these are handled can have a major impact on area and performance
- **Arrays:** Arrays are used often in C code. They can influence the device IO and become performance bottlenecks
- **Operators:** Operators in the C code may require sharing to control area or specific hardware implementations to meet performance

```
void fir (
    data_t *y,
    coef_t c[4],
    data_t x
){

    static data_t shift_reg[4];
    acc_t acc;
    int i;

    acc=0;
    loop: for (i=3;i>=0;i--) {
        if (i==0) {
            acc+=x*c[0];
            shift_reg[0]=x;
        } else {
            shift_reg[i]=shift_reg[i-1];
            acc+=shift_reg[i] * c[i];
        }
    }
    *y=acc;
}
```



# C, C++ and SystemC Support

The vast majority of C, C++ and SystemC is supported

- Provided it is statically defined at **compile time**
- If it's not defined until **run time**, it won't be synthesizable

HLS is a static analysis of the HW.

Any of the three variants of C can be used

- If C is used, Vivado HLS expects the file extensions to be .c
- For C++ and SystemC it expects file extensions .cpp



# **Comprehensive C/C++ Support**

## **A Complete C Validation & Verification Environment**

- Vivado HLS supports complete bit-accurate validation of the C model
- Vivado HLS provides a productive C-RTL co-simulation verification solution

## **Modeling with bit-accuracy**

- Supports arbitrary precision types for all input languages
- Allowing the exact bit-widths to be modeled and synthesized

## **Floating point support**

- Support for the use of float and double in the code



# **Design of Hardware Accelerators**

Academic Year 2021/2022

## Questions?

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)



**POLITECNICO**  
MILANO 1863

**Design of Hardware Accelerators**  
Academic Year 2021/2022

# High-Level Functionality Description

**Christian Pilato**

Dipartimento di Elettronica, Informazione e Bioingegneria

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)

# HLS Problem Definition

Generates HW description from an high level language

# Input

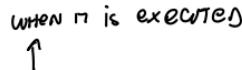
- An intermediate representation
  - A set of functional resources (with area/time characterization)  
    Assees / multiplets ...  
    Associated to the technology
  - A set of constraints WHAT you need to respect
  - One or more objectives

# Output

- Hardware description of the **data-path + controller**

# Tasks

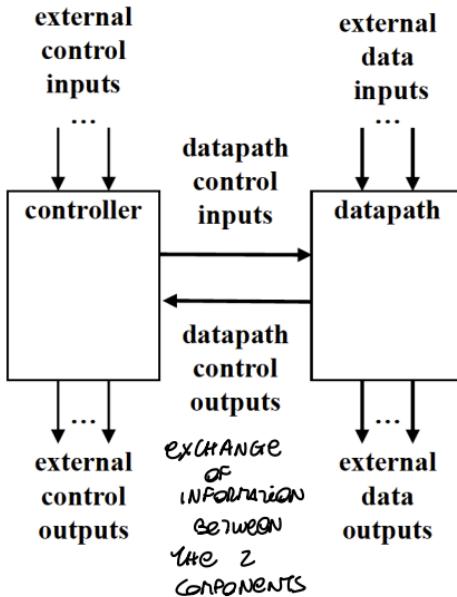
- Place operations in time (scheduling) and space (binding)
  - Determine detailed interconnection and control



WHERE EACH OPERATION IS EXECUTED (IN WHICH FUNCTIONAL UNIT?)

# Data-path and Controller

Model to organize the microarchitecture of each component



- DATA MANIPULATION →
- Datapath:** TAKES EXTERNAL DATA INPUT AND PROCESSES THE DATA OUTPUT
- **Functional resources:** Perform operations on data (arithmetic and logic blocks)
  - **Memory resources:** Store data (internal memories and registers)  
Memory is to store data
  - **Interconnection resources:** Connect all resources (muxes, busses and ports)

## Controller:

- Finite state machine (FSM)

Allows you to describe the behavior



# Constraints and Objectives

**Area:** number of modules/resources available or size of your silicon die

- Constraint: Maximum area something that you must respect (A solution that don't respect the constraints is considered invalid)
- Objective: Minimize area something that you want optimize (you can accept the worst solution)

Same for other cost metrics (e.g., **power**)

→ Number of cycles from the beginning to the end

**Latency**: number of cycles for input data to result in a solution or result.

**Throughput**: amount of data that can be processed in a given amount of time (usually involves dataflow/pipelining)

- Constraint: Maximum latency/minimum throughput
- Objective: Minimize latency/maximize throughput



# Intermediate Representations

An **intermediate representation** (IR) is the **internal format** used to represent a functionality to be synthesized

- Ideally language-agnostic (possibility to use the same IR for multiple input languages)  
*↳ Iden... there are no reference to the input language*
- Conducive for further processing, such as optimization and translation (every optimization step is usually an IR-to-IR transformation)

## Examples of intermediate representations for HLS

- Abstract syntax tree (AST)
- 3-address code
- Basic block and Control flow graph (CFG)
- Static single assignment form (SSA)
- Directed acyclic graph (DAG)

# Abstract Syntax Tree (AST)

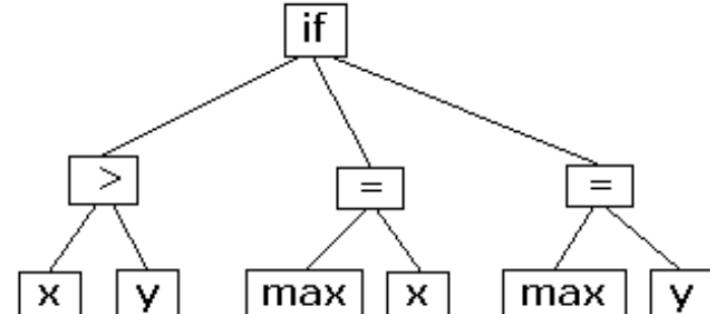
In computer science, an **abstract syntax tree** is a representation of the abstract syntactic structure of text written in a formal language

- Each node of the tree denotes a construct occurring in the text
- Still language dependent (based on the constructs in the language)
- Usually associated with a backend to reproduce the code

Easy to be extracted and manipulated

- Source-to-source transformations
- Extraction of the language constructs for transformation into low-level IR

```
if (x > y)
    max = x;
else
    max = y;
```



## 3-address code

Each statement is converted into the form:  $x = y \text{ op } z$

- single operator and at most three names

$$t = x - y + z; \quad \rightarrow \quad \begin{aligned} &\text{tmp} = x - y; \\ &t = \text{tmp} + z; \end{aligned}$$

## Simple and easy-to-read format

- Explicit names of the intermediate values (signals)
  - Standard format for the operators (possibility to design a library of components)



# Basic Block

A **basic block** is a maximal sequence of instructions with:

- no labels (except at the first instruction), and
- no jumps (except in the last instruction)

So:

- You can enter into the basic block only at the beginning
- You can exit from the basic block only at end

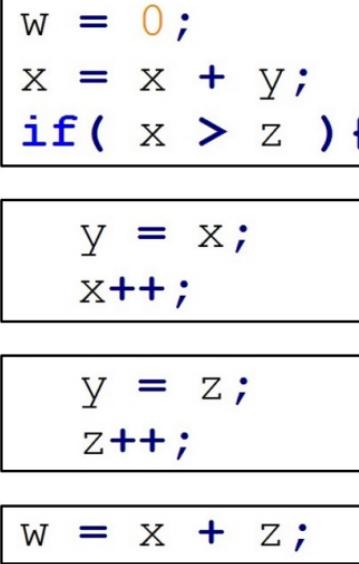
All operations inside the basic block must be executed before moving to the next one

IN THE BASIC BLOCK IT CAN BE ANYTHING WITHOUT IMPACTING THE  
OTHER



# Examples of Basic Blocks

```
w = 0;  
x = x + y;  
if( x > z ) {  
    y = x;  
    x++;  
} else {  
    y = z;  
    z++;  
}  
w = x + z;
```



Code

4 Basic Blocks



# How to Identify a Basic Block?

- Input: sequence of instructions  $instr(i)$ 
  - Any while/for/switch can be converted into a sequence of operations combined with `if/goto`
- Identify **leaders**. Leaders are:
  - The entry point of the function
  - Any instruction that is a target of a branch
  - Any instruction following a (conditional or unconditional) branch
- Iterate by adding subsequent instructions to the basic block until we reach another leader



# Control Flow Graph (CFG)

Graph representation of the control flow.

- Nodes are the basic blocks
- Directed Edges are the potential control flow paths

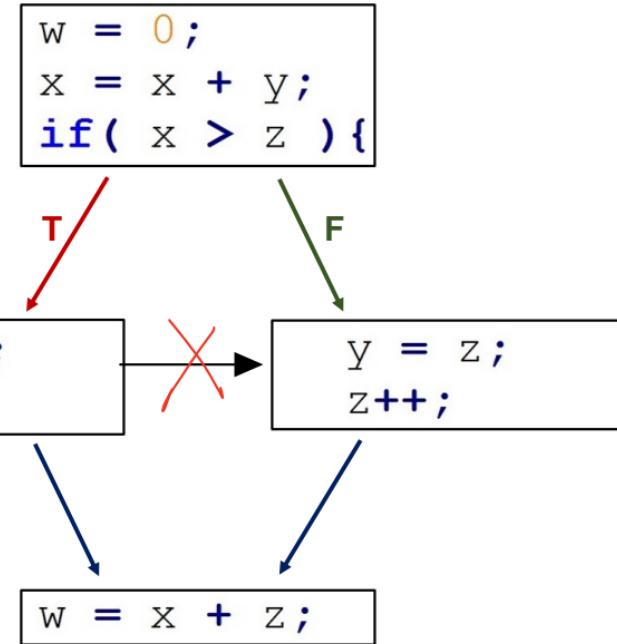
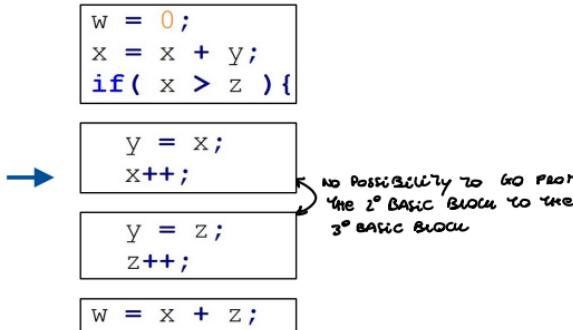
You have a directed edge between B1 and B2 if:

- BRANCH from last statement of B1 to first statement of B2 (B2 is a leader), or
- B2 immediately follows B1 in program order and B1 does NOT end with unconditional branch (goto)

Analysis of the (backward) edges allow the identification of **loops**

# Example of CFG

```
w = 0;  
x = x + y;  
if( x > z ) {  
    y = x;  
    x++;  
} else {  
    y = z;  
    z++;  
}  
w = x + z;
```



# Single Static Assingment (SSA)

A program is in Single Static Assingment (SSA) form if every variable is only assigned once

- Every new assignment to the same variable creates a new temporary value

Original

```
a := b + c  
b := c + 1  
d := b + c  
a := a + 1  
e := a + b
```

SSA

```
a1 := b1 + c1  
b2 := c1 + 1  
d1 := b2 + c1  
a2 := a1 + 1  
e1 := a2 + b2
```

Useful to directly identify the operation that creates my variable conceptually  $a_1$  and  $a_2$  are the same variable but they carry different values.



# SSA and $\phi$ -Functions

If the variable value can come from different path (e.g., different branches of an IF statement), it is necessary to determine the correct source

- Use of the  $\phi$ -functions

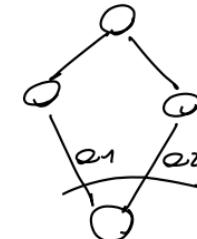
Original

```
if B then
    a := b
else
    a := c
end
... a ...
```

SSA

```
if B then
    a1 := b
else
    a2 := c
End
a3 :=  $\Phi(a_1, a_2)$ 
... a3 ...
```

CFG



Dynamically move the  
values  $a_1$  or  $a_2$  into  
 $a_3$  to use it later



# CFG and $\phi$ -Functions

$\phi$ -functions are always at the **beginning** of a basic block

Select between values depending on control-flow

$a_m := \phi(a_1 \dots a_k)$ : the block has  $k$  preceding blocks, the  $\phi$ -function defines a new variable

$\phi$ -functions are **all evaluated simultaneously**

- generally implemented as multiplexers or register transfers
- often they are automatically "absorbed" by (register) binding algorithms



# Dataflow Graphs (i)

Behavioral views of architectural models

Useful to represent data-paths

Each basic block have a data flow graph associated with it

Graph:

- Vertices = operations
- Edges = dependencies



# Data Flow Graphs (ii)

Used to model data dependencies in the code

Four types of data dependencies

- **Flow** or read-after-write
- **Anti** or write-after-read
- **Output** or write-after-write
- **Input** or read-after-read

Notes:

- Input dependencies does not affect scheduling so they can be executed in parallel
- Anti and Output dependencies can be removed by register renaming technique or SSA
- So, DFG is used to model only **RAW dependencies** (*target operation must be executed only after the source operation has written the data*)

# Data Flow Graph Example

Easy to identify **operations** that can **run in parallel**

single-assignment form:

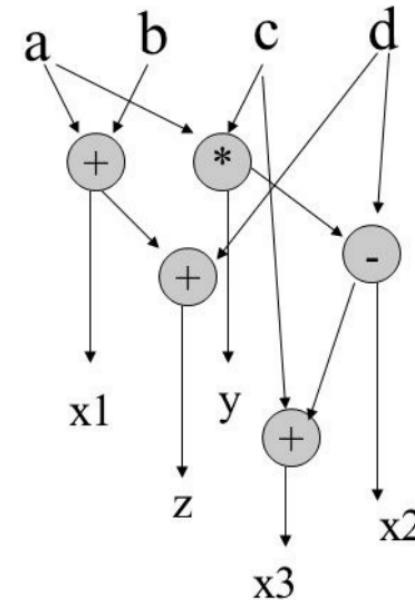
$x_1 \leftarrow a + b;$

$y \leftarrow a * c;$

$z \leftarrow x_1 + d;$

$x_2 \leftarrow y - d;$

$x_3 \leftarrow x_2 + c;$



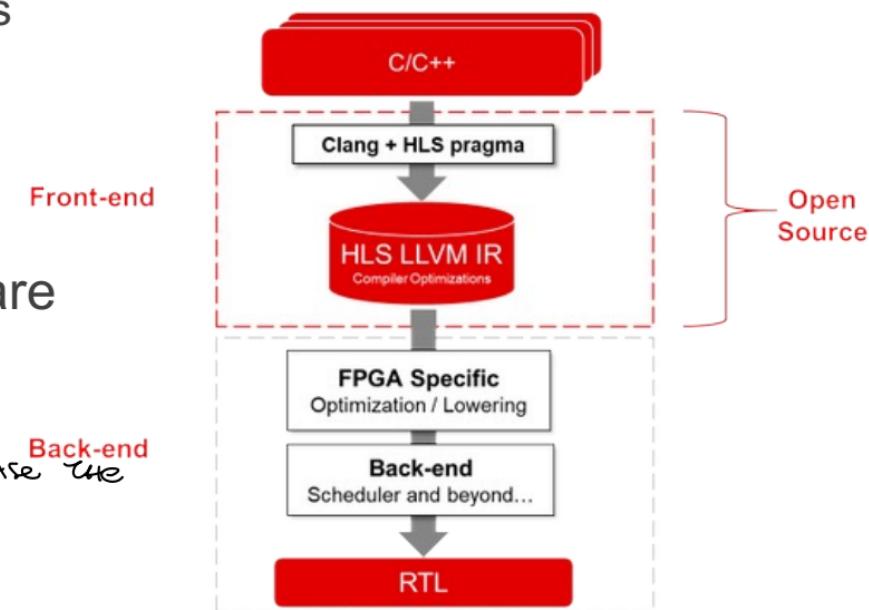
# Compiler Frontend

Usually based on **mature compilers** (GCC and LLVM)

- Many transformations are purely on the functionality
- Leveraging years of research in compilers
- Support for many input languages

FPGA/HLS **specific optimizations** are based on the resulting IR

- bitwidth analysis
- memory partitioning *MULTIPLE BANKS TO INCREASE THE  
NO OF PORTS*
- creation of custom resources



# Compiler Transformations

In Vivado/Vitis HLS, they can be selected with **pragma annotations** or **TCL directives**

- Pragmas are easier to read (directly into the code)
- Directives are easier to be “changed” (e.g., for DSE – same code but different files). They require identifiers into the code
- Always, **give a label to every loop**. It helps debugging

Usually applied to:

- Functions: inline
- Loops: unroll, pipeline
- Memories: partition
- Operations and dependencies



# Example

```
void block fir(int input[256], int output[256], int taps[NUM TAPS],  
              int delay_line[NUM TAPS]) {  
  
    int i, j;  
    U: for (j = 0; j < 256; j++) {  
        int result = 0;  
  
        loop THAT I  
WANT TO  
UNROLL }  
        U: for (i = NUM TAPS - 1; i > 0; i--) {  
            #pragma HLS unroll  
            delay_line[i] = delay line[i - 1];  
        }  
        delay line[0] = input[j];  
  
        loop THAT  
I WANT  
TO  
PIPELINE }  
        C: for (i = 0; i < NUM TAPS; i++) {  
            #pragma HLS pipeline  
            result += delay line[i] * taps[i];  
        }  
        output[j] = result;  
    }  
}
```

IMPORTANT TO GIVE A LABEL  
TO EACH LOOP



# Standard and Custom Data Types

For each data type, HLS assumes the **same bitwidth** of the corresponding **CPU versions**

- **int**: 32 bits / **char**: 8 bits
- **float**: 32 bits / **double**: 64 bits
- pointers: 32/75 bits (depending on the CPU memory addressing)

Simplify **hardware/software interfaces**: data can be simply copied as they are *THE INTEGER OF THE SW CAN BE USED AS IS BY THE ACCELERATOR*

In some cases, there are too many bits for the real range of the values

- E.g.: values between 0 and 1,000,000 only requires 20 bits



# Custom Data Types

In HLS, it is common to customize the ranges by using **synthesizable libraries** for declaring the variables with custom data types

```
unsigned int variable;
```

//32 bits



```
ap_uint<20> variable;
```

//20 bits

*SYNTAX FOR XILINX*

The variable can be later used as *it is*

→ the engine does the DATA bitwidth analysis to minimize the logic

```
variable = variable + 1; // operation with 20 bits
```

The HLS engine performs **data-range propagation** to minimize the logic



# Accelerator Interfaces (i)

**Top-level ports** must be connected to the **rest of the system**:

- preserving the **semantics** of the function
- enabling data **exchanges** between SW AND HW

Given a functionality, HLS always generate ports for each of the parameters as follows:

- Parameters **passed by copy** are converted into input ports (connected to registers written by the CPU)
- Parameters **passed by reference** are converted into memory interfaces (access to a memory external to the component)

HLS also adds **control ports** to manage **start**, **done**, and **reset**



# Accelerator Interfaces (ii)

HLS can also automatically generate **standardized interfaces** on top of the basic ones

- AXI-Lite for parameters (memory-mapped IO operations) IS A PROTOCOL AND YOU CAN MOVE THE CONTROL SIGNALS
- AXI-Master/AXI-Stream for memory accesses

When dealing with **external memories**, the scheduling phase must have assumptions on the **latency of the operations**

- **Local data** (PLM or scratchpad) have **fixed-latency access**
  - Simple interface with CE, R/W, ADDR, DIN, DOUT
  - Address bitwidth is customized with respect to the memory to be accessed
- **Remote data** (cache or off-chip memory) have **variable-latency access**
  - More complex interfaces with protocols to exchange data



# **Design of Hardware Accelerators**

Academic Year 2021/2022

## Questions?

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)



**POLITECNICO**  
MILANO 1863

**Design of Hardware Accelerators**  
Academic Year 2021/2022

# High-Level Synthesis Scheduling and Binding

**Christian Pilato**

Dipartimento di Elettronica, Informazione e Bioingegneria

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)

# What is Scheduling?

***Scheduling is the assignment of operations to time (control steps), possibly within given limits on hardware resources and latency***

What does scheduling do?

- Uses **data dependencies** to identify parallelism
- Exploits **mutual exclusion**
  - Code that is never executed at the same time can be scheduled in the same clock cycles (and share the units)
- Optimizes **loops**

Generally, one the first steps in the HLS core engine

# What is Binding?

***Binding is the assignment of operations to hardware resources (functional units) such that there are no conflicts in using them and the total number is minimized***

Naive approach: Assign each operation to a different functional unit

What does binding do?

- Uses **scheduling information** to identify sharing opportunities
- Exploits **mutual exclusion**
  - Operations in different basic blocks are never executed at the same time and can share hardware resources
- Can be defined before scheduling
  - Imposes constraints on the operation scheduling (e.g., operation serialization)

# What is Resource Sharing?

***Resource sharing is the possibility of using the same functional unit to implement two (or more) operations without any conflicts***

Sharing opportunities can be defined *before* or *after* scheduling

## Before:

- Pre-defined binding. Two operations that share the same resource cannot be executed in the same clock cycle and must be serialized
  - Additional scheduling constraints

## After:

- Binding algorithms on scheduled graph. Two operations that are not executed in the same clock cycle can share the same functional unit

# Scheduling Problem Definition

## Input

- **Intermediate representation** (DFG, CDFG, etc.)
- **Clock period** (or target frequency)
- **Functional unit latencies** expressed (in nanoseconds)

## Output

- Determine the **start time** for each operation
- Satisfy all the **data dependencies** and **resource constraints**

## Goal

- Primary: Optimize the **circuit latency**
- Secondary: Achieve **area/latency trade-off**

# Scheduling Effects

**Performance:** Scheduling determines the **timing evolution** of the circuit, so it has a direct impact on the latency (or throughput) of the implementation

- Identification of dependencies to exploit spatial parallelism

**Area:** Scheduling has an **indirect effect** on area. Operations in the same clock cycle require to be assigned to different physical units

- So, the maximum number of concurrent operations of the same type is a lower bound on the required number of hardware resources

# Scheduling Approaches

## Scheduling without constraints

- Assumption: infinite resources
- Applications: obtain lower bound on clock cycles

## Scheduling under resource constraints

- Idea: schedule operations (possibly serializing them) such that the overall number of used resources is within a given budget
- Applications: limit the use of resources

## Scheduling under timing constraints

- Idea: schedule operations such that the end time of the last ones are within a given time budget
- Applications: real-time scheduling

# Scheduling Algorithms

**Exact formulations:** variables to represent the assignment to units and clock cycles, constraints to represent dependencies and variables

- Linear Programming
- Integer Linear Programming

However, scheduling is an **NP-hard problem: heuristics**

- ASAP (As Soon As Possible) and ALAP (As Late As Possible)
- List-based scheduling
- Force-directed scheduling
- Path-based scheduling
- Percolation scheduling
- Meta-heuristics: (simulated annealing, tabu search, etc.)

*Borrowing concepts from compilers*

# ASAP: Definition

Each operation is scheduled in the first clock cycle in which is **available**

*An operation is available when all its predecessors have been scheduled and have completed their execution*

All operations have **bounded delays**

- Expressed in numbers of cycles (multiples of the clock period)

**No constraints** on resources or area

- Unconstrained Scheduling Problem

**Goal:** minimize latency

- *Lower bound:* Circuit cannot go faster (use less clock cycles) than ASAP scheduling

# ASAP: Algorithm

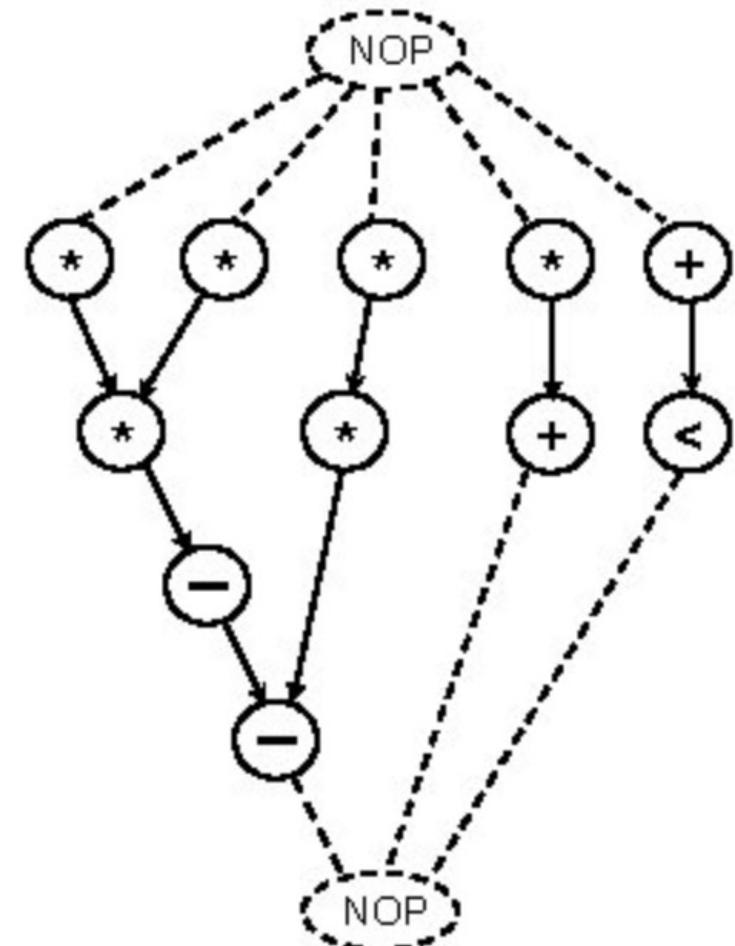
1. Initialize the set of **ready vertices** with the source node
2. Pick one node from the set of ready vertices and schedule it with the following equation

$$start\_time(op_i) = \max_{p \in Pred} end\_time(op_p)$$

3. Define the end time of the current node

$$end\_time(op_i) = start\_time(op_i) + delay(op_i)$$

4. Add all successors of the current node to the set of ready vertices
5. Repeat from step 2 until the set is empty



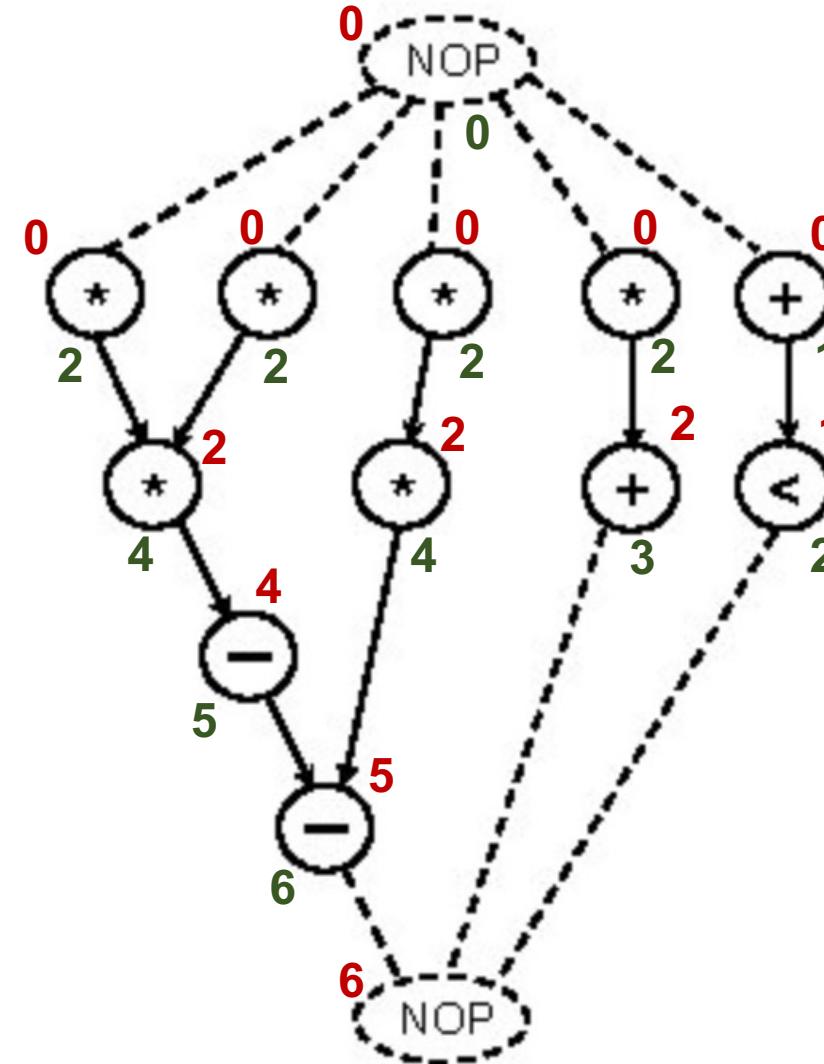
# ASAP: Example

Assumptions:

- latency of multipliers: 2 cycles
- latency of adders: 1 cycle
- latency of comparators: 1 cycle

- **Start time**
- **End time**

$$start\_time(op_i) = \max_{p \in Pred} end\_time(op_p)$$



# ALAP: Definition

Dual problem of ASAP: It solves a **latency-constrained problem**

- Latency bound is set to latency computed by ASAP algorithm

*Each operation is scheduled in last clock cycle where it can be scheduled without causing an extra delay*

All operations have **bounded delays**

- Expressed in numbers of cycles (multiples of the clock period)

**No constraints** on resources or area

- Unconstrained Scheduling Problem

# ALAP: Algorithm

1. Initialize the set of **ready vertices** with the sink node
2. Pick one node from the set of ready vertices and schedule it with the following equation

$$end\_time(op_i) = \min_{s \in succ} start\_time(op_p)$$

3. Define the start time of the current node

$$start\_time(op_i) = end\_time(op_i) - delay(op_i)$$

4. Add all predecessors of the current node to the set of ready vertices
5. Repeat from step 2 until the set is empty

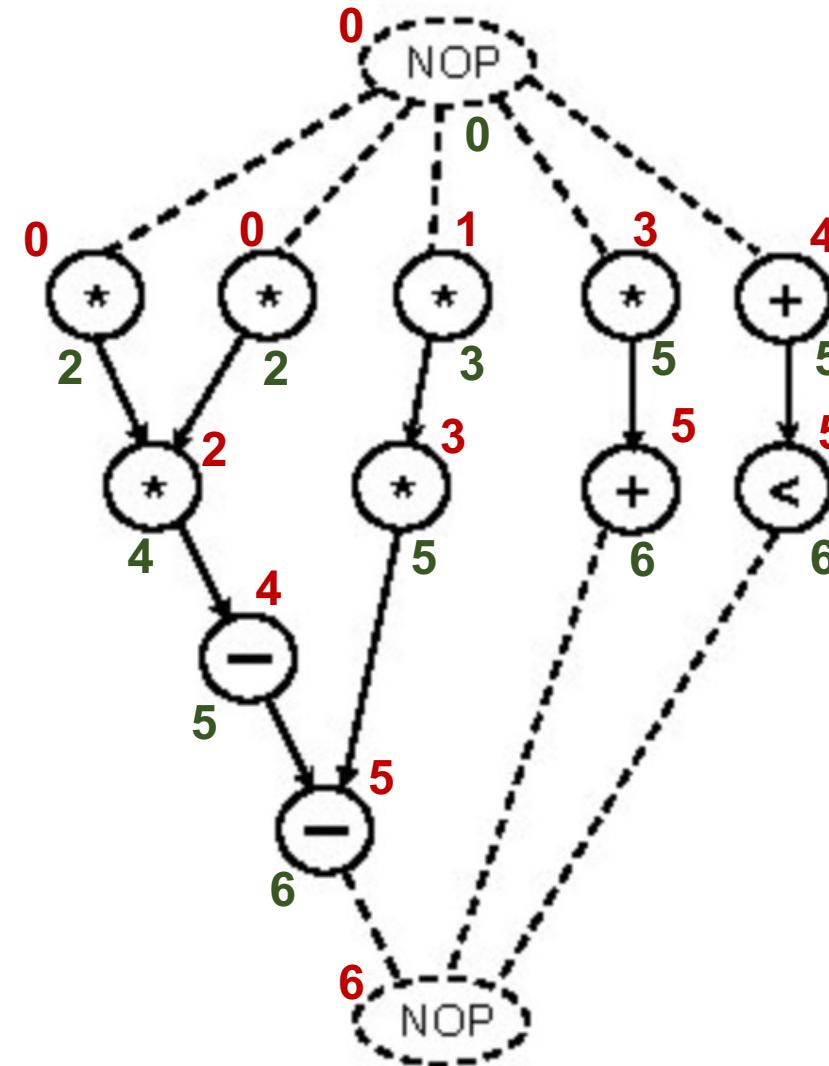
# ALAP: Example

Assumptions:

- latency of multipliers: 2 cycles
- latency of adders: 1 cycle
- latency of comparators: 1 cycle

- **Start time**
- **End time**

$$end\_time(op_i) = \min_{s \in succ} start\_time(op_p)$$



# Definition of Mobility

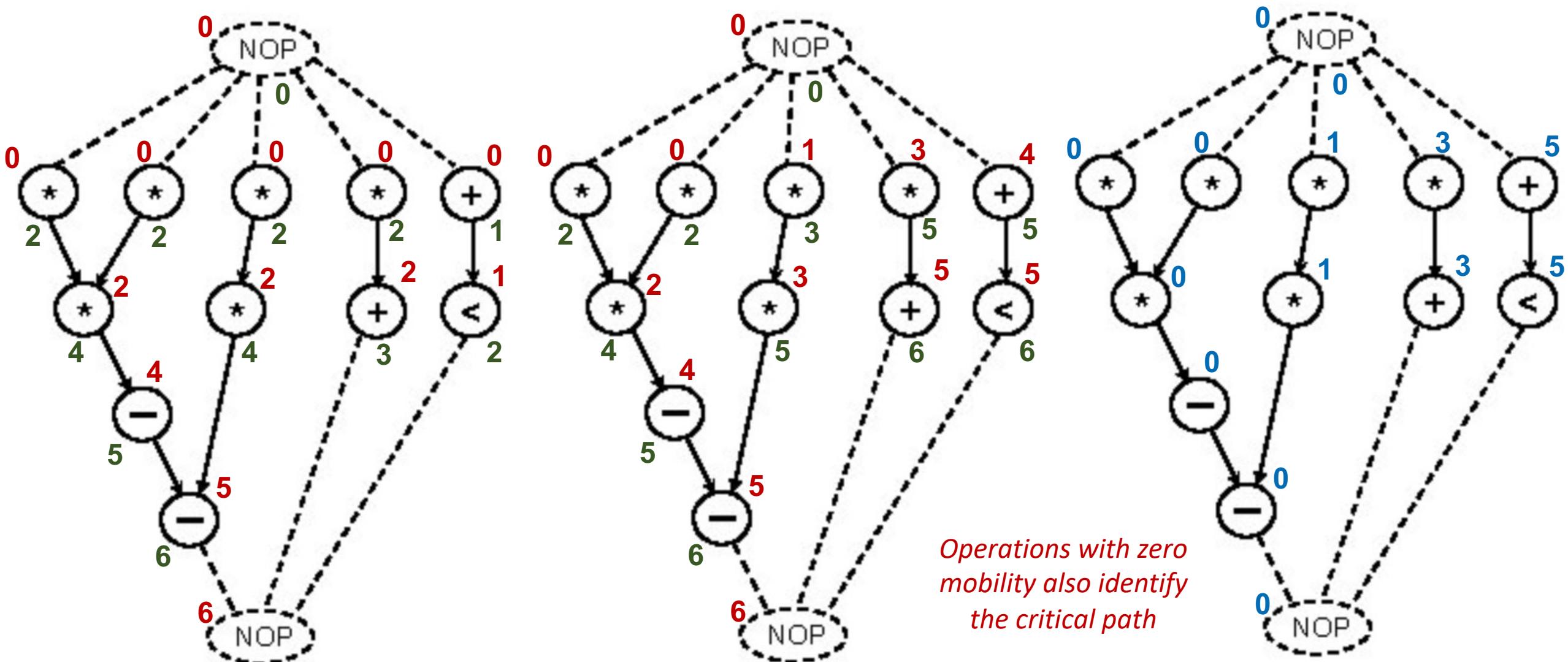
**Mobility** is a metric associated with each operation and is defined as the difference between its ALAP and ASAP schedules

**Zero mobility** implies that an operation can start only at a given time step without introducing any delay on the overall schedule

Mobility greater than zero measures the **slack** on the start time

- Time interval in which an operation may start

# Mobility: Example



# Resource Constrained Scheduling

Scheduling with infinite resources is often inadmissible

- *Why should I use more resources if I can have the same scheduling (or a slightly slower one) with much less hardware logic?*

Different variants in the formulation

- **Minimize latency** given constraints on area or the resources (ML-RCS)
- **Minimize resources** subject to bound on latency (MR-LCS)

Exact solution methods

- ILP: Integer Linear Programming

Heuristics

- List scheduling
- Force-directed scheduling

# List-Based Scheduling: Definition

List-based Scheduling (or simply List Scheduling) is a simple greedy algorithm to consider limited resources (constrained scheduling) and

- Heuristic methods for ML-RCS and MR-LCS

**Operation selection** decided by **criticality** (low mobility)

Greedy strategy

- Does NOT guarantee optimum solution
- $O(n)$  time complexity (linear)

More general input (any type of dependencies)

- Works on general graphs
- Resource constraints on different resource types

# List-Based Scheduling: Algorithm

1. Construct a priority list based on some metrics (**operation mobility**, numbers of successors, etc)
2. While not all operations scheduled
  1. For each **available resource**, select an operation in the ready list following the descending priority.
  2. Assign the operation to the current clock cycle
  3. Update the ready list
  4. Continue until there are no more ready operations or available resources
  5. Increment the clock cycle

*It creates and  
maintains a list for  
each resource*

QoR depends on the circuit but also on the particular metric

# Static vs. Dynamic Mobility

An operation with **high mobility** (and **static mobility**) is generally postponed to the next clock cycle

- Risk of starvation

A possible solution is to “update” the mobility after each iteration (**dynamic mobility**)

- If the operation is not selected, its mobility is decreased

It enforces the definition of mobility

- If an operation is not selected in one clock cycle, its “slack” is decreased (less time before the deadline)

# Scheduling Challenges

*Remember that the delay of an operation is given by the functional units*

All algorithms assumed functional units that complete in one (single-cycle) or more cycles (multi-cycle)

All functional units execute at most one operation

Functional units can execute more than one operation (**multi-function**) or start another operation before the previous is completed (**pipelined**)

If two operations are serial and the total execution time is less than the clock period, they can be executed one after the other (**chaining**)

Operations may have unbounded latency, e.g., accesses to external memory (**synchronization protocols**)

# Resource Binding

It is defined as the **spatial mapping** between operations and resources

It tries to search for **sharing opportunities**

- Assignment of a resource to more than one operation
- *What is the best alternative?*

Constrained resource binding

- Resource-dominated circuits
- Fixed number and type of available resources

This is again an **NP-complete problem** – heuristics

# Binding: Problem Definition

- Input: **scheduled graph**
- Operation concurrency well (and already) defined

Consider *operation types* independently

- **Problem decomposition** (natural)
- Perform analysis for each resource type

Analysis on **operation pairs**

- **Compatibility:** Same type, Non-concurrent, etc.
- **Conflict:** Concurrent, Different types, etc.

Dual problems

# Resource Compatibility Graph $G_+(V,E)$

- Vertices  $V$  represent operations
- Edges  $E$  represent compatible operation pairs

*Two operations ( $v_i, v_j$ ) are compatible if they are not concurrent and can be implemented by resources of the same type*

Note: concurrency depends on schedule

# Clique Covering Problem

The **binding problem** can be formulated as a **partitioning** of the **compatibility graph**

*Each partition is a clique (fully connected subgraph) of operations that are all compatible with each other. So, they can share the same resource*

The clique covering is thus a partitioning of graph  $G_+$  into the minimum number of cliques

- Each clique represents a functional unit

It is possible to solve the partitioning imposing a **minimum number of cliques** (more units, less interconnections)

It is possible to assign weights to the edges to prioritize the connections (**weighted clique covering problem**)

# Conflict Graph $G_{-}(V,E)$

- **Vertices**  $V$  represent **operations**
- **Edges**  $E$  represent **operation pairs in conflict**

*Two operations  $(v_i, v_j)$  are in conflict if they are not compatible*

The conflict graph is complementary to the compatibility graph

- It identifies operations that cannot share resources

# Coloring Problem

The **binding problem** can be formulated as a **coloring problem** of the **conflict graph**

*Each node will be assigned to a color and two adjacent nodes cannot have the same color*

The color will represent the identifier of the functional unit

The goal is to minimize the overall number of colors

- Each node with a different color is an admissible but trivial solution
- Equivalent to cliques composed of only one node

# **Design of Hardware Accelerators**

Academic Year 2021/2022

# Questions?

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)



**POLITECNICO**  
MILANO 1863

**Design of Hardware Accelerators**  
Academic Year 2021/2022

# High-Level Synthesis: Microarchitecture Creation

**Christian Pilato**

Dipartimento di Elettronica, Informazione e Bioingegneria

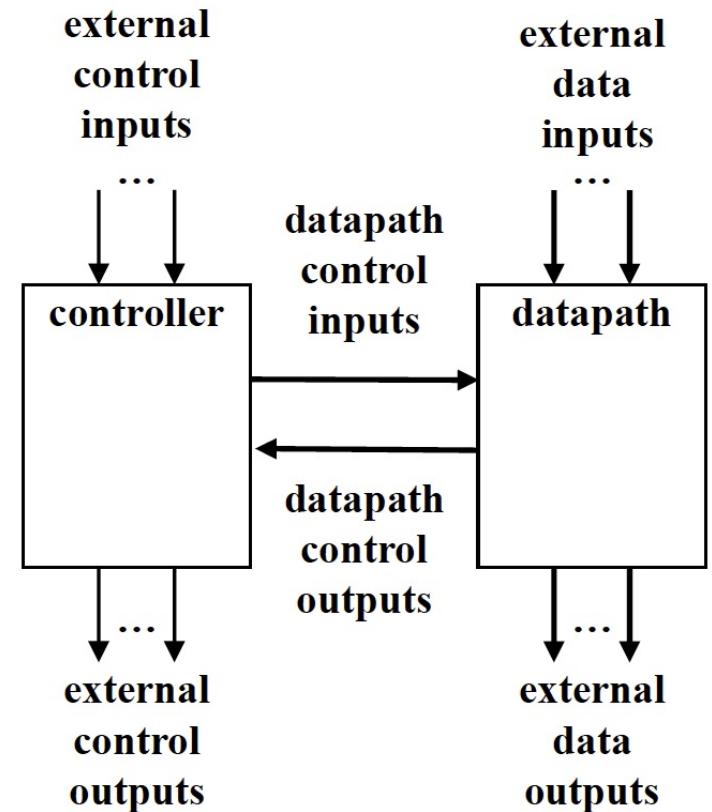
[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)

# Microarchitecture Creation

After defining the operation scheduling and binding, HLS proceeds with the **generation of the RTL microarchitecture**

We need to create the (micro-)architecture of the following elements:

- Controller
- Datapath
- Interfaces with local and/or external memories



# Controller Creation

The controller is described as a Finite State Machine:

- **States**: collection of operations to be executed in the given clock cycle
- **Transitions**: evolution over time of the behavior
  - Inside the basic block: sequential list of states to be executed from the beginning to the end of the basic block
  - Among the basic blocks: transitions between the last operation of one basic block to the first operation of the next one

**Output function** defines the control signal for the datapath resources

- Selectors of multiplexers, write enables of the registers, etc.

The FSM is derived by combining the **CFG of the basic blocks** and the **scheduling of the operations** inside each basic block

- The output function depends also on the module/register binding

# Datapath Creation

The datapath is a **collection of hardware resources** for **computation and storage**

- **Functional units:** to perform the operations
- **Registers:** to store intermediate values
- **Wires and multiplexers:** to interconnect all these resources
  - Each port that has multiple sources requires one or more multiplexers to drive the signal values

Varying the module/register binding can vary the number of sources and destinations and, in turn, the number of multiplexers

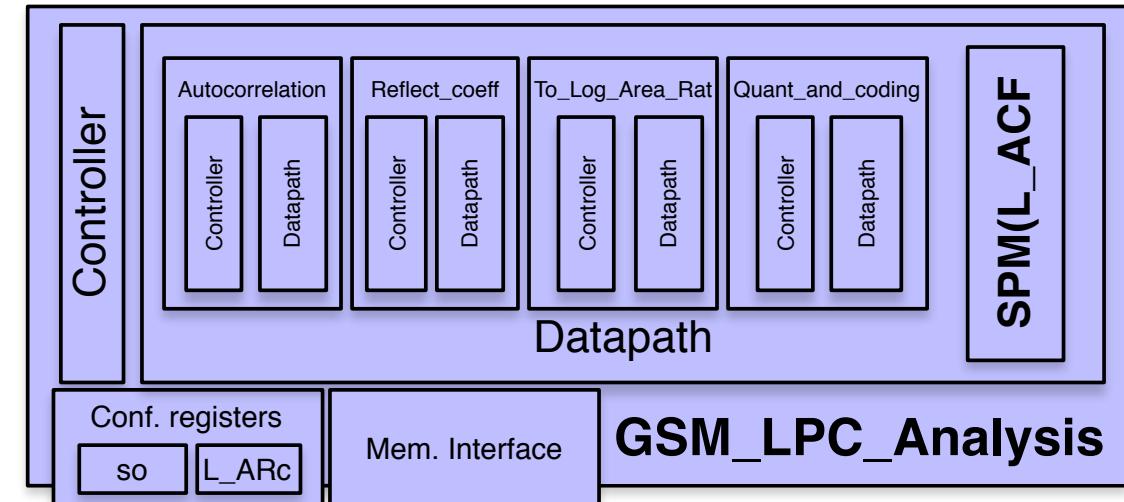
The cost of multiplexers, especially on FPGA, is significant

# Memories and Memory Operations

C language allows us to easily specify, design, and optimize accelerators for irregular applications

- Massive use of **pointer-based operations** (arithmetic, dynamic resolutions, accesses to external memory, ...)

```
void Gsm_LPC_Analysis(word* so, word* LARc)
{
    longword L_ACF[9];
    Autocorrelation(so, L_ACF);
    Reflect_coeff(L_ACF, LARc);
    To_Log_Area_Rat(LARc);
    Quant_and_coding(LARc);
}
```

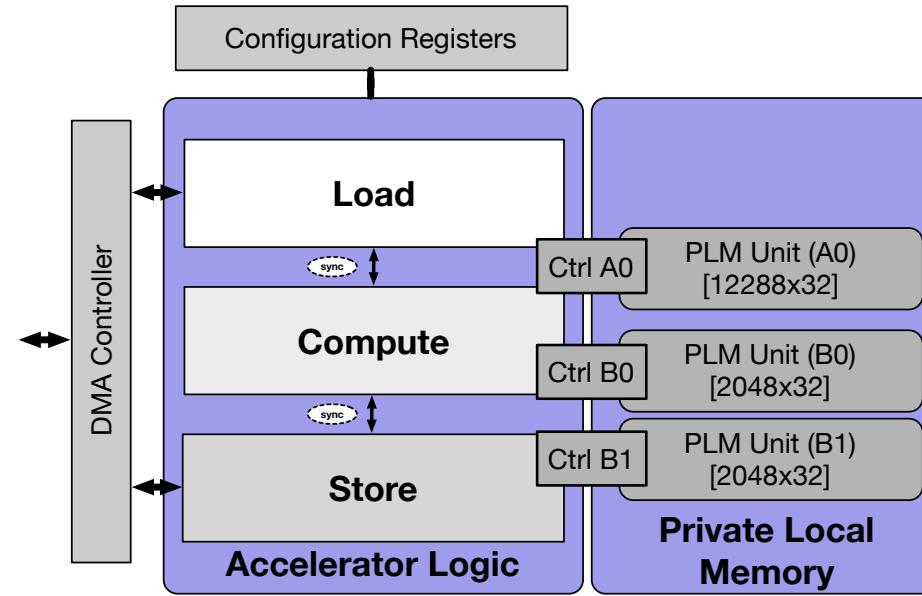


# Memories and Memory Operations

SystemC language allows us to easily specify, design, and optimize data-intensive accelerators

- Massive use of **arrays with predefined size**
- DMA transfers with main memory to exchange data blocks

```
SC_MODULE(debayer) {
    sc_in<bool> clk, rst;
private:
    int A0[6][2048];
    int B0[2048], B1[2048];
public:
    SC_CTOR(debayer) {
        SC_CTHREAD_Load, clk.pos());
        reset_signal_is(rst, false);
        SC_CTHREAD(Compute, clk.pos());
        reset_signal_is(rst, false);
        SC_CTHREAD(Store, clk.pos());
        reset_signal_is(rst, false);
    }
    ...
}
```



# Pointer Synthesis (Software)

In software, a C program targets a virtual architecture consisting of a single (unified) memory in which all data are stored

The semantics of pointers is the **address of an element in memory**

- Even though register declarations may allow programmers to specify the variables to be placed in registers, the assignment of variables to registers is generally done by the compiler
- The notions of caches and memory pages are transparent to programmers

# Pointer Synthesis (Hardware)

In hardware, designers want to have control on where data are stored and optimize data locality

- Typically, a chip design contains multiple memory banks, register files, registers, and wires
- To efficiently map C code onto hardware, the unified storage space must be partitioned
- During synthesis, each partition is then mapped to a register, a wire, or a memory

Pointers may be used to reference any variable no matter where its information is available

- References to memory elements, registers, wires, or ports

# Pointer Encoding

The **hardware synthesis of pointers** includes the following steps

- Partitioning of the memory into *locations* (or *partitions*)
- Mapping of the partitions onto hardware resources
  - To a variable (wire or register)
  - To an array (akin to memory or register file)
- Generation of the proper hardware logic to access the data

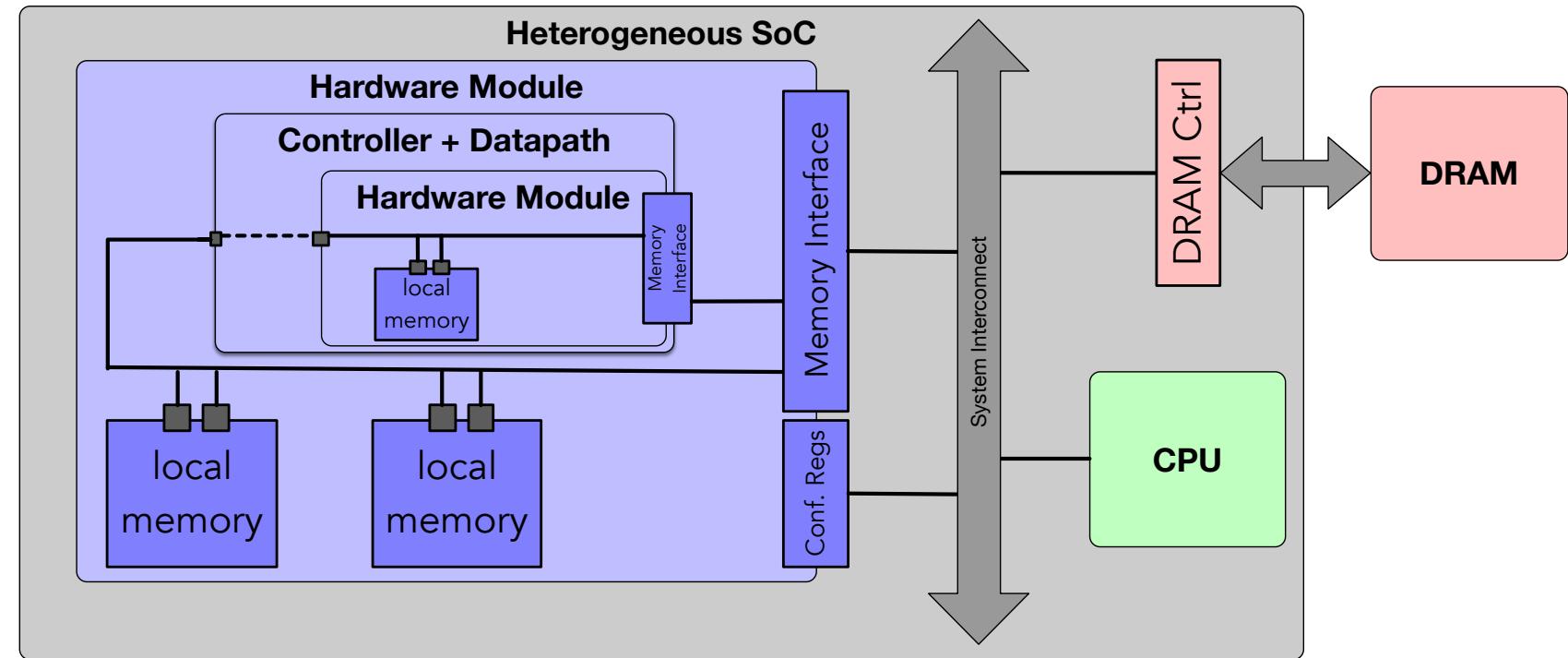
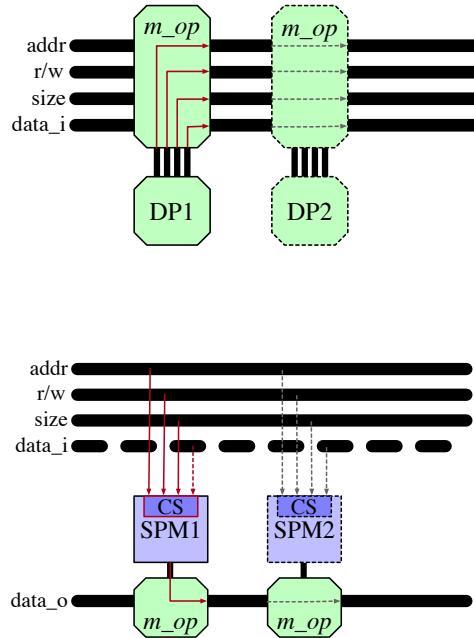
**Virtual addresses** must be translated into **operations to the proper hardware resources**

It is important to understand whether the physical memory resource that is targeted by a memory operation can be statically identified

# Daisy-Chain Architecture for Pointer Arithmetic

Internal memory bus where the pointer is dynamically resolved

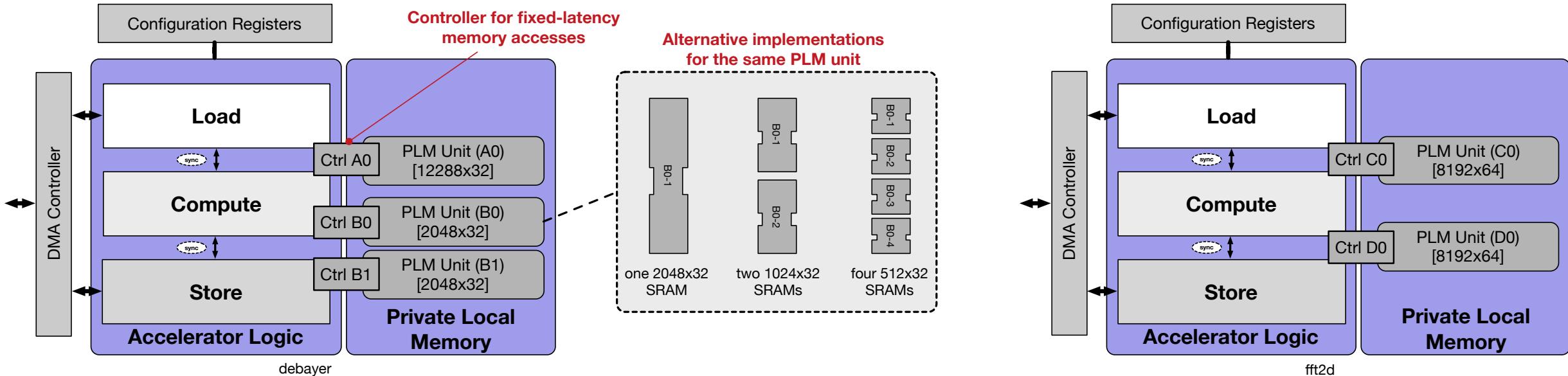
- Daisy-chain architecture with possibility to access the external memory



# How to Design Private Local Memory?

Specialized multi-bank local memories for storing part of the data

- memory design **transparent** to accelerator logic
- **alternative implementations** with **block/cyclic partitioning**



# Array Partitioning

Array transformations to create independent data structures that can be accessed in parallel

- Each new substructure is managed as a new array
- The problem is how to distribute the data contained into the original array to guarantee that parallel operations operate on distinct array

It is necessary to determine the **access pattern**, i.e., the distribution of the memory operations on the array over time

- When the access pattern is irregular or unknown, we need to duplicate the data

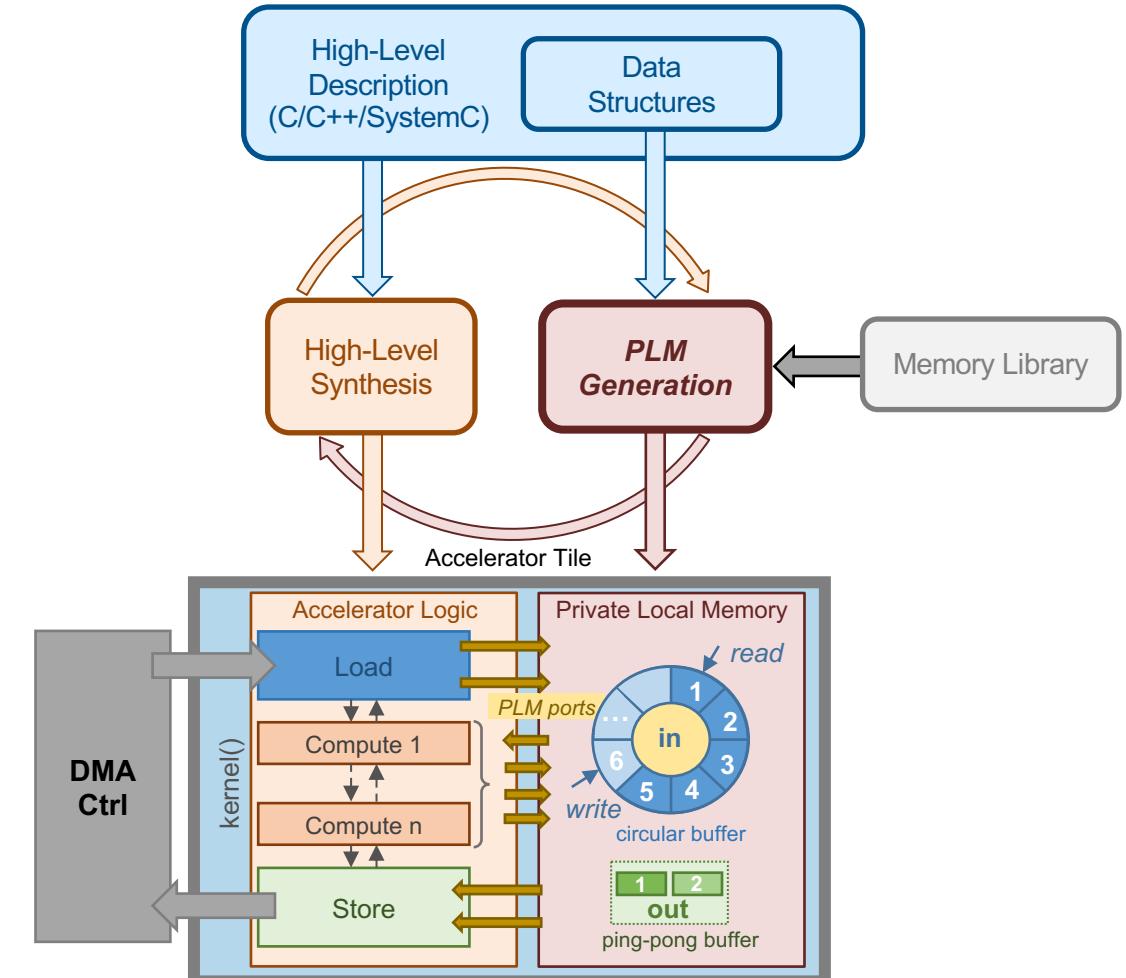
# PLM Customization for Heterogeneous SoCs

High-Level Synthesis (HLS) to create the **accelerator logic**

- Definition of memory-related parameters  
(e.g. number of process interfaces)

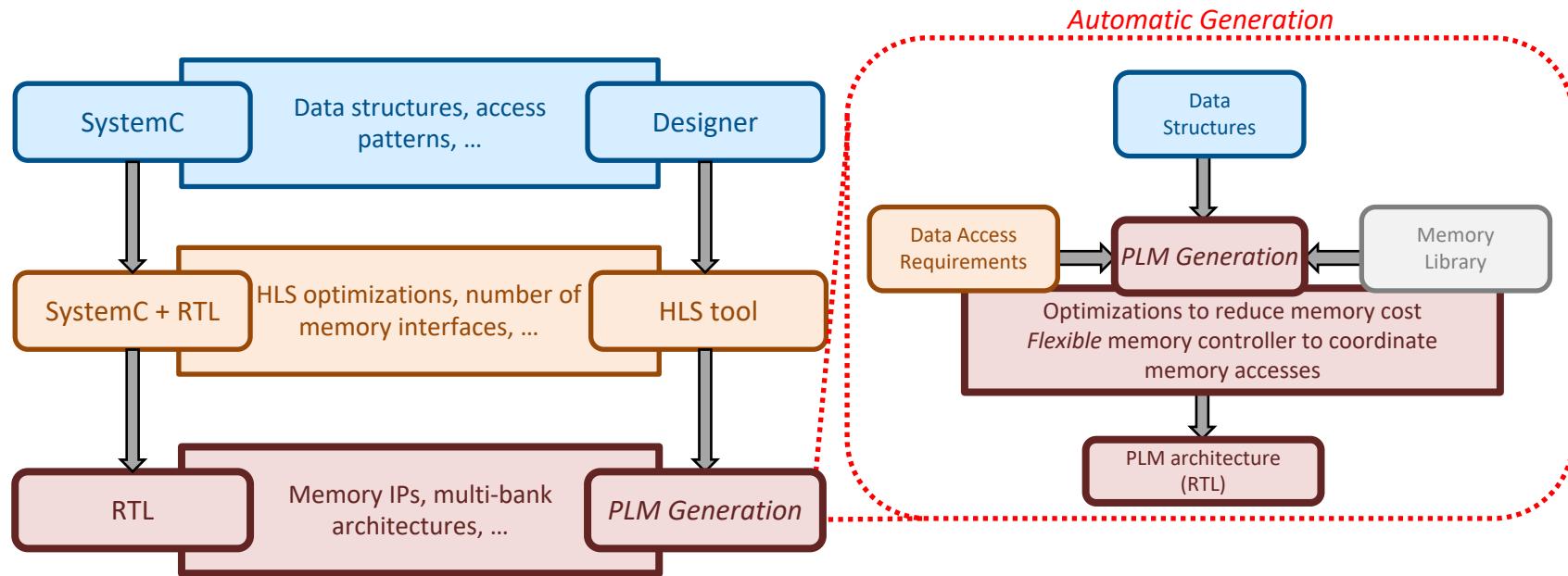
Generation of **specialized PLMs**

- Technology-related optimizations
- Possibility of system-level optimizations across accelerators



# PLM Customization

## System-level methodology for **PLM customization**



**Performance optimization:** *HLS* defines how the accelerator logic accesses the data structures (e.g. number of parallel accesses)

**Cost optimization:** *PLM Customization* defines the best PLM microarchitecture to achieve the desired performance (e.g. number of banks, data allocation)

# Reuse What is not Used

Generally, we can use one **PLM unit** (eventually composed of many banks) for each data structure

**Reuse the same memory IPs  
for several data structures**

**“Two data structures are compatible if they can be allocated to the same PLM unit (memory IPs)”**

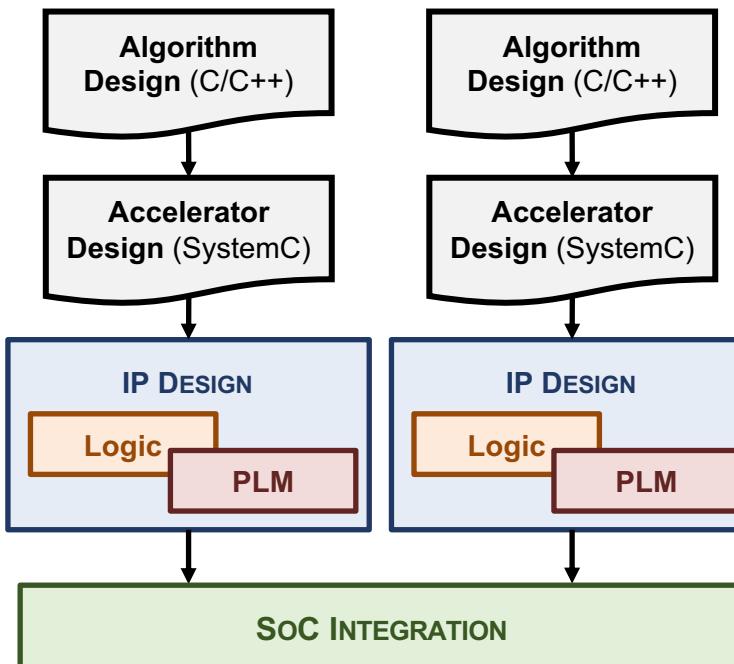
A common case: accelerators never executed at the same time

- Possible only at system-level, when integrating the components
- Optimizations of accelerator logic and memory subsystem are independent

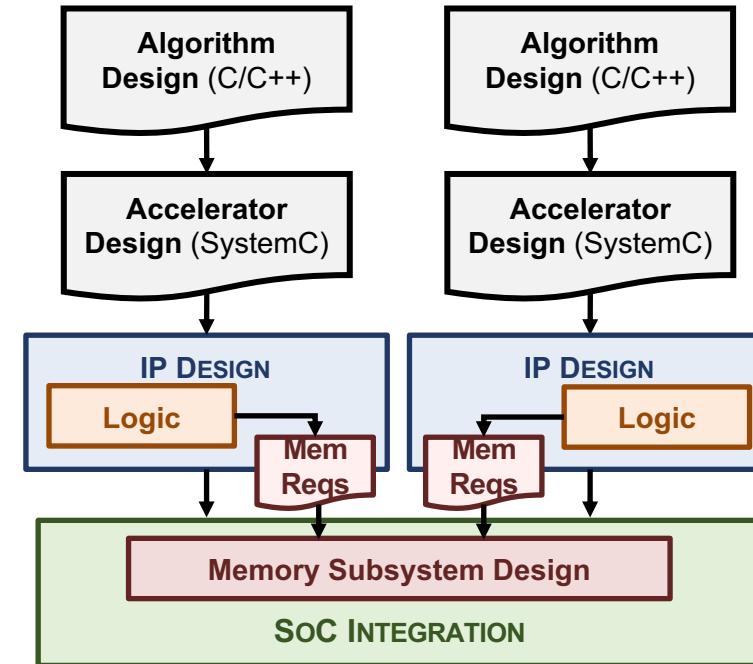
# Optimization only at the System-Level

Accelerator(s) memory subsystem is defined **during SoC integration**

- Possibility for **more optimizations**

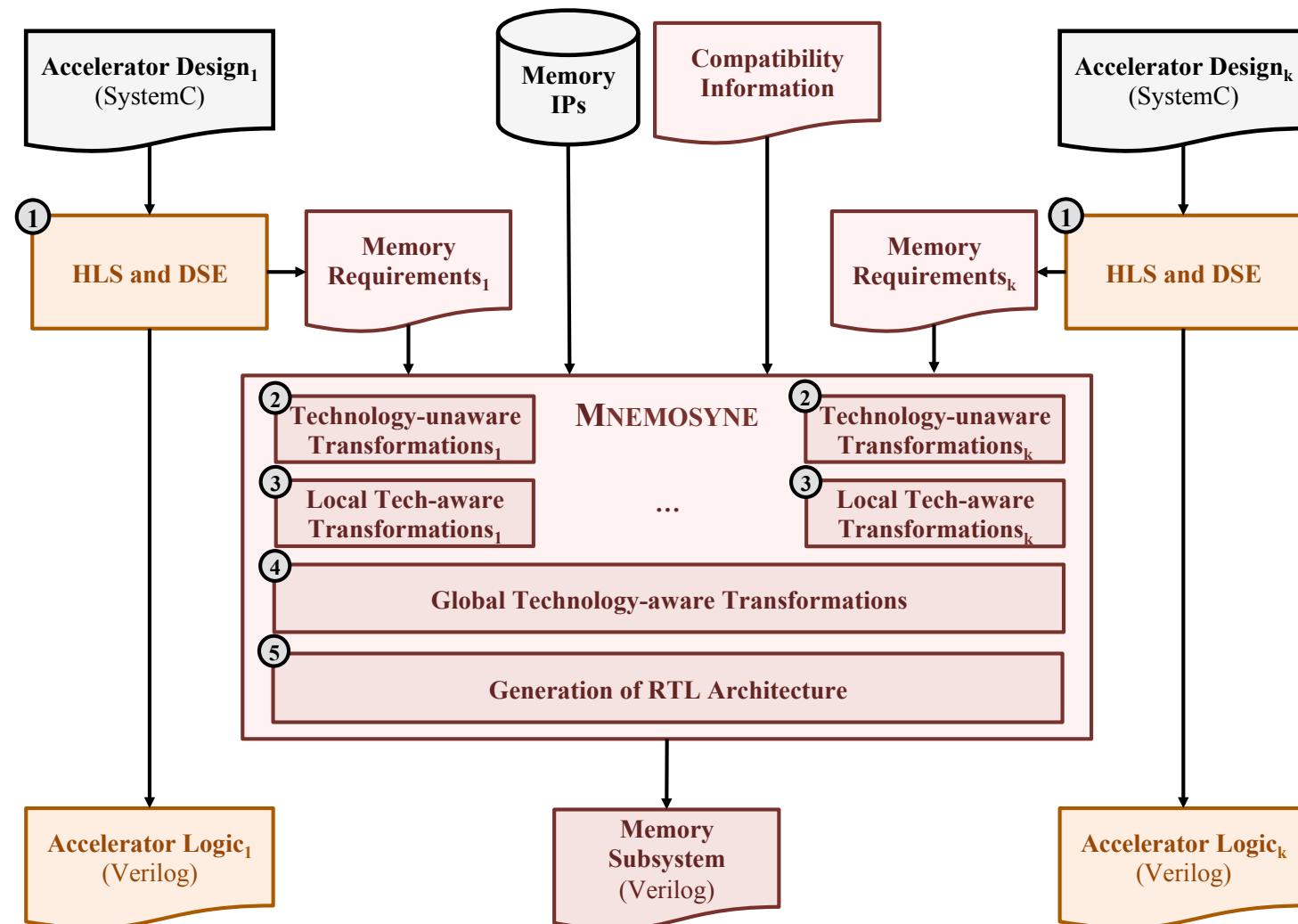


Component-based Approach



*System-Level Approach*

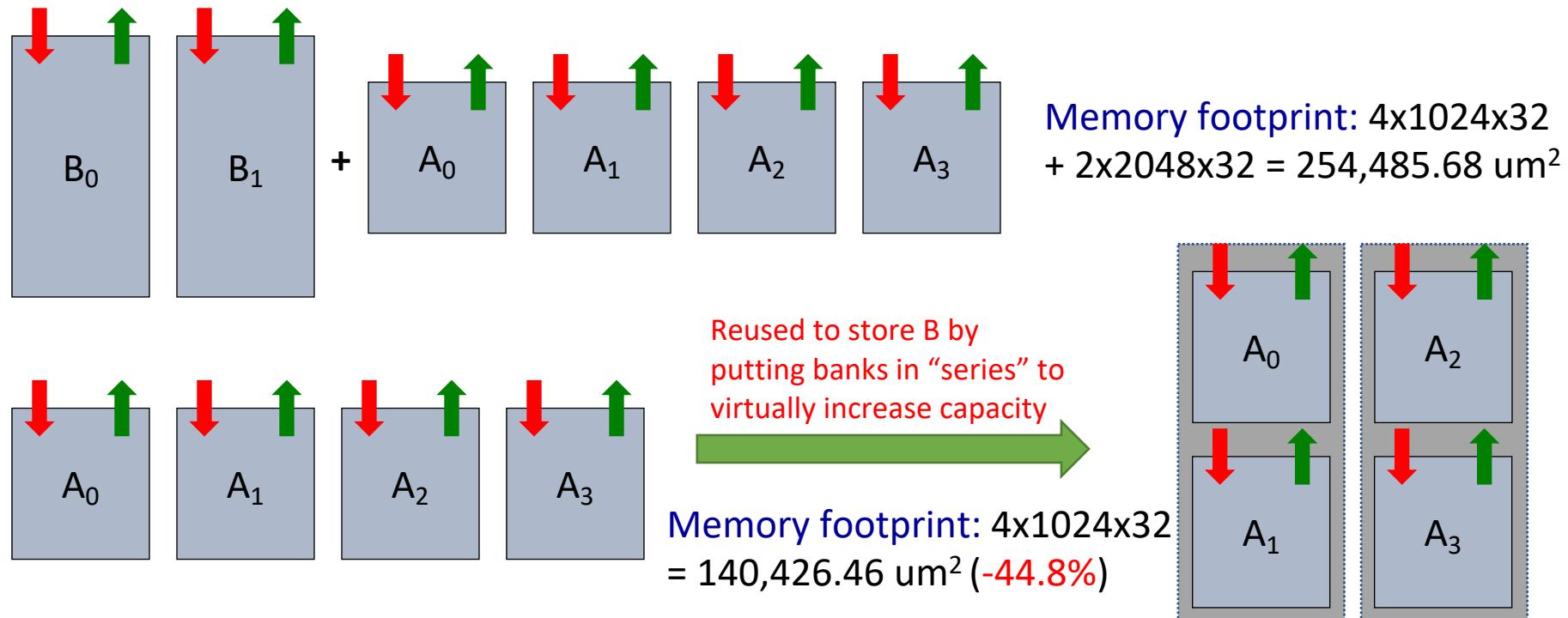
# PLM Optimization for Multiple Accelerators



# Address-Space Compatibility

Let us assume to have the two following data structures that are never *alive* at the same time

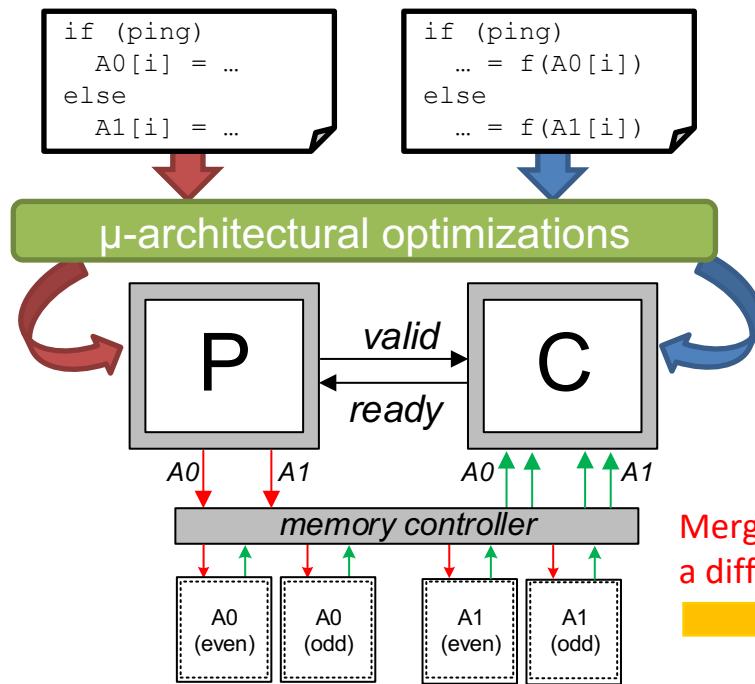
- A[1024] with *data duplication* over 4 parallel banks
- B[4096] with *data distribution* (cyclic partitioning) over 2 parallel banks



# Memory-Interface Compatibility

A classical example is the ping-pong buffer (two 2048x16 arrays – A0/A1)

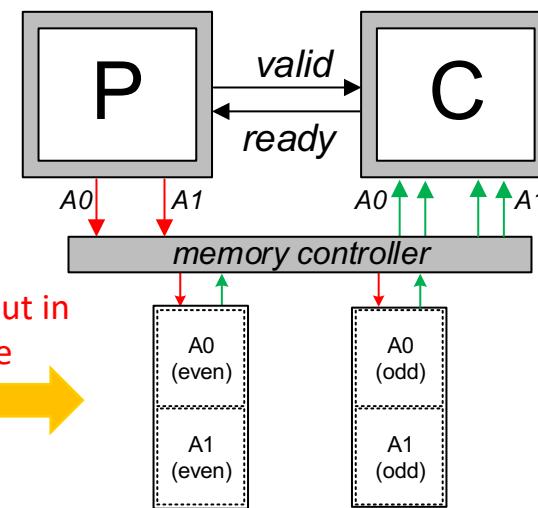
- When process P writes A0 (A1), it never writes A1 (A0)
- When process C reads from A0 (A1), it never reads from A1 (A0)



Memory footprint:  $4 \times 1024 \times 32 = 140,426 \text{ } \mu\text{m}^2$

Memory footprint:  $2 \times 2048 \times 32 = 114,059.2 \text{ } \mu\text{m}^2$

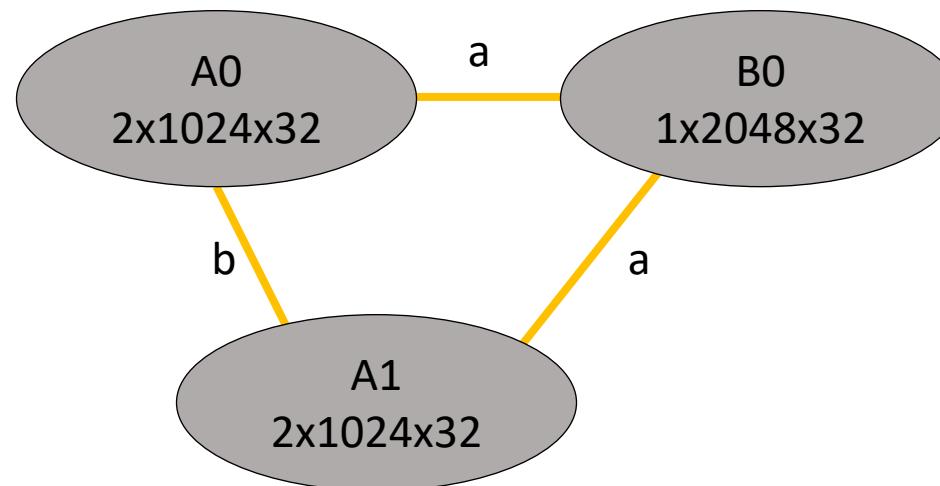
**Area reduced by 18% without any performance overhead!**



# Memory Compatibility Graph (MCG)

Graph to represent the possibilities for optimizing the data structures

- Each node represents a data structure to be allocated, annotated with its data footprint (after data allocation)
- Each edge represents compatibility between the two data structures

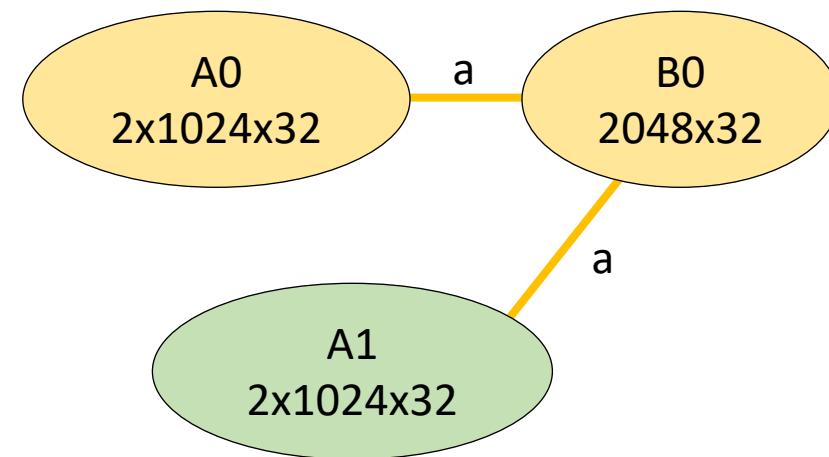
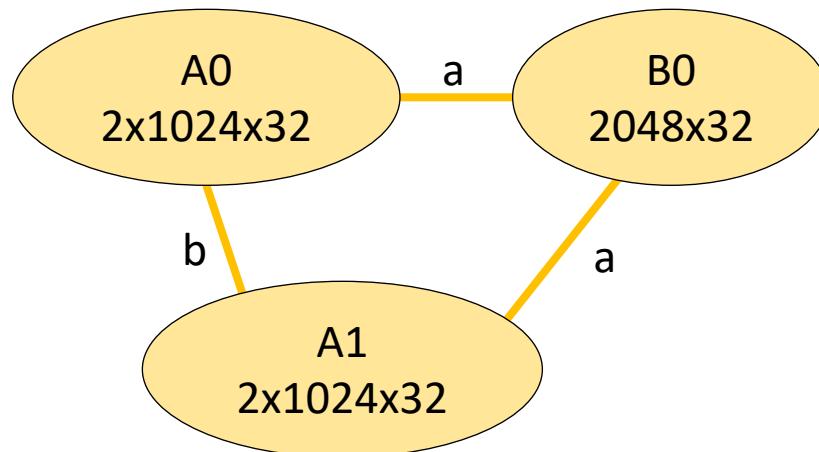


- a) **Address-space compatibility:** the data structures are compatible and can use the same memory IPs
- b) **Memory-interface compatibility:** the ports are never accessed at the same time and the data structures can stay in the same memory IP

# Clique Definition

“A clique is a subset of the vertices of the memory compatibility graph such that every two vertices are connected by an edge”

A clique represents a **set of data structures** that can share the same memory IPs

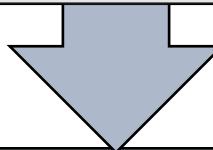


We need two distinct configurations!  
{A0,B0} and {A1} or {A1,B0} and {A0}?

# How to Determine the Memory Subsystem

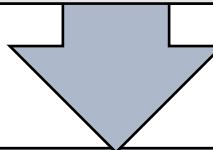
## Clique Enumeration

To define the list of admissible cliques in the MCG



## Clique Characterization

To determine the memory architecture of all cliques and their memory cost



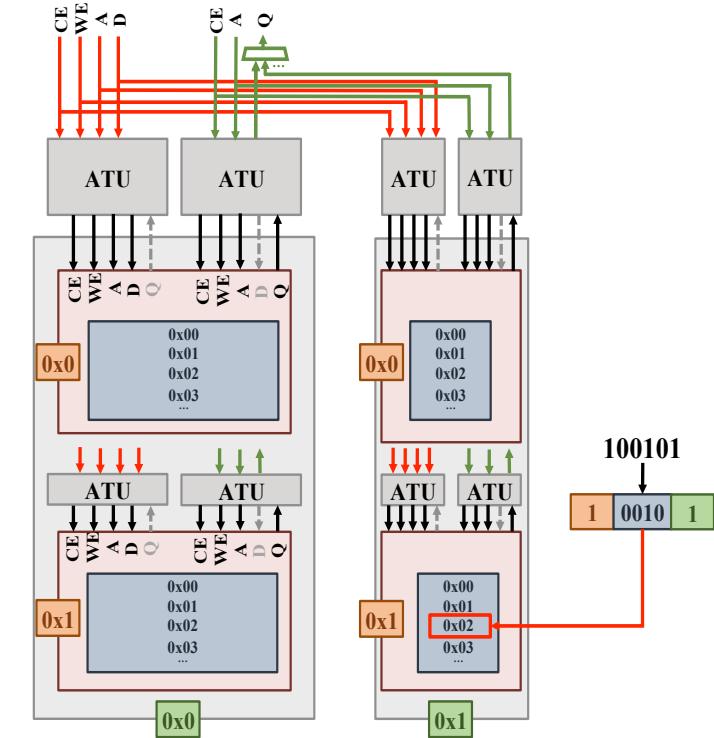
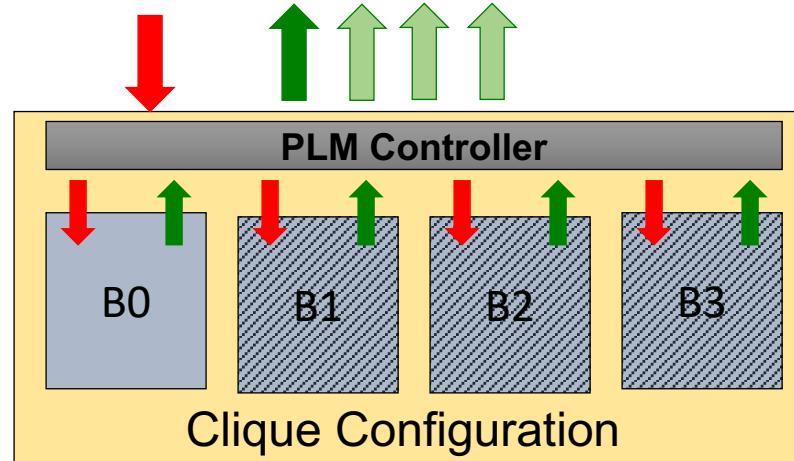
## Memory Cost Minimization

To determine how to partition the MCG such that the total memory cost is minimized

# PLM Controller Generation

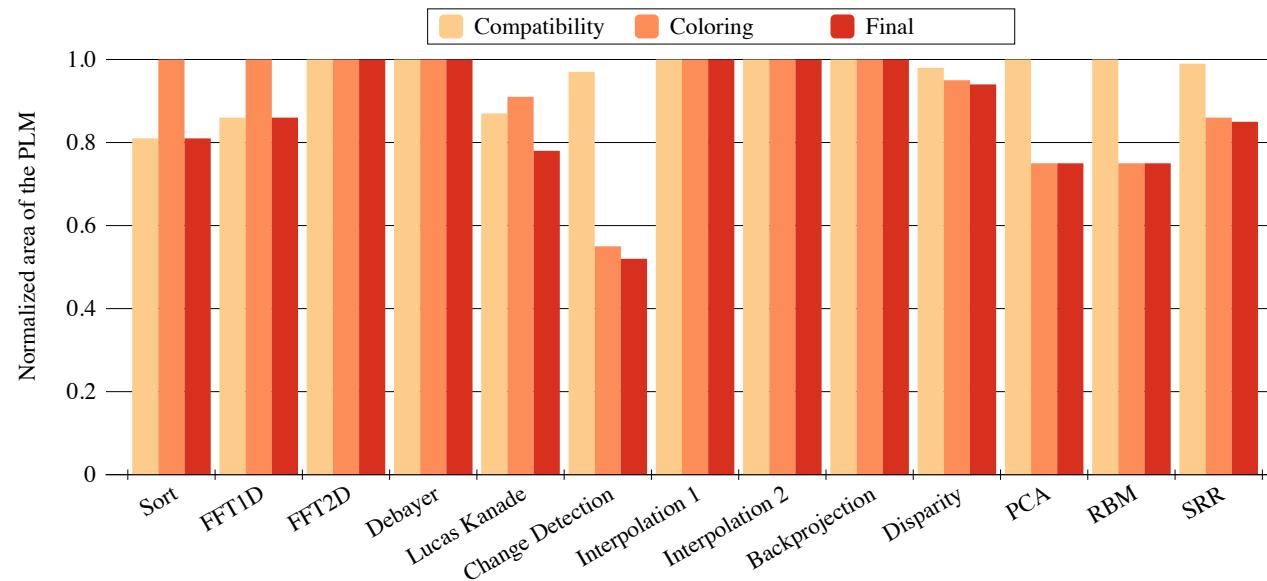
A **lightweight PLM controller** is created for each compatibility set (clique) based on the bank configuration

- Accelerator logic is not aware of the actual memory organization
- Array offsets need to be translated into proper memory addresses



**Custom logic** with negligible overhead, especially when the number of banks and their size is a power of two

# Impact of Optimizations

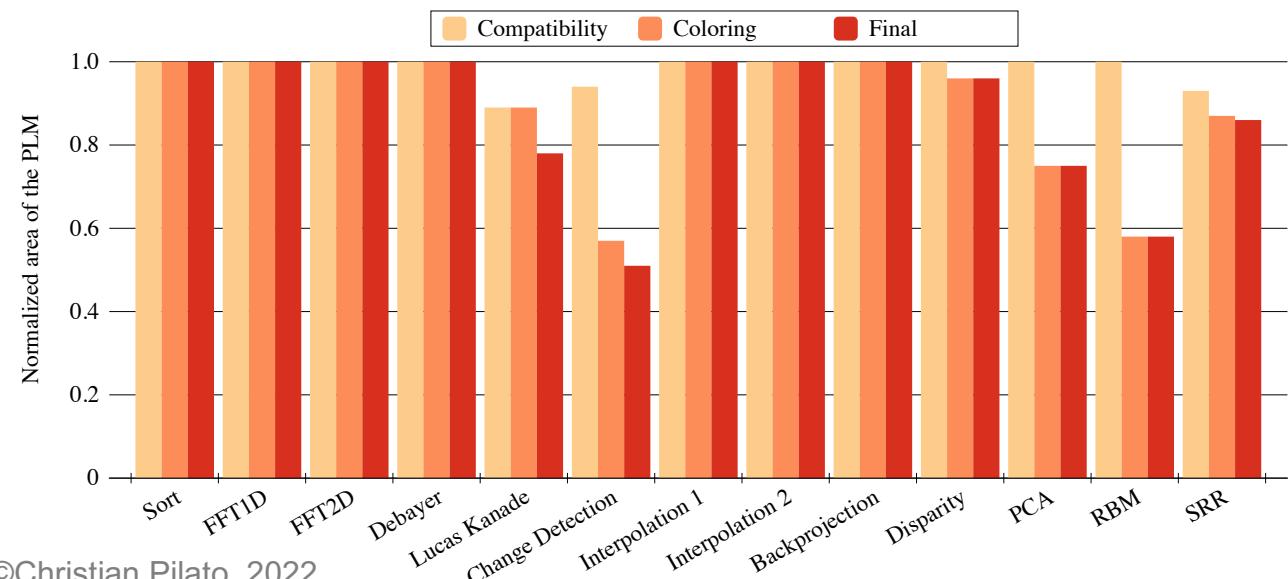


## Xilinx Virtex-7 FPGA

- Memory library with 6 BRAM configurations

## Industrial 32nm CMOS technology

- Memory library with 18 SRAMs



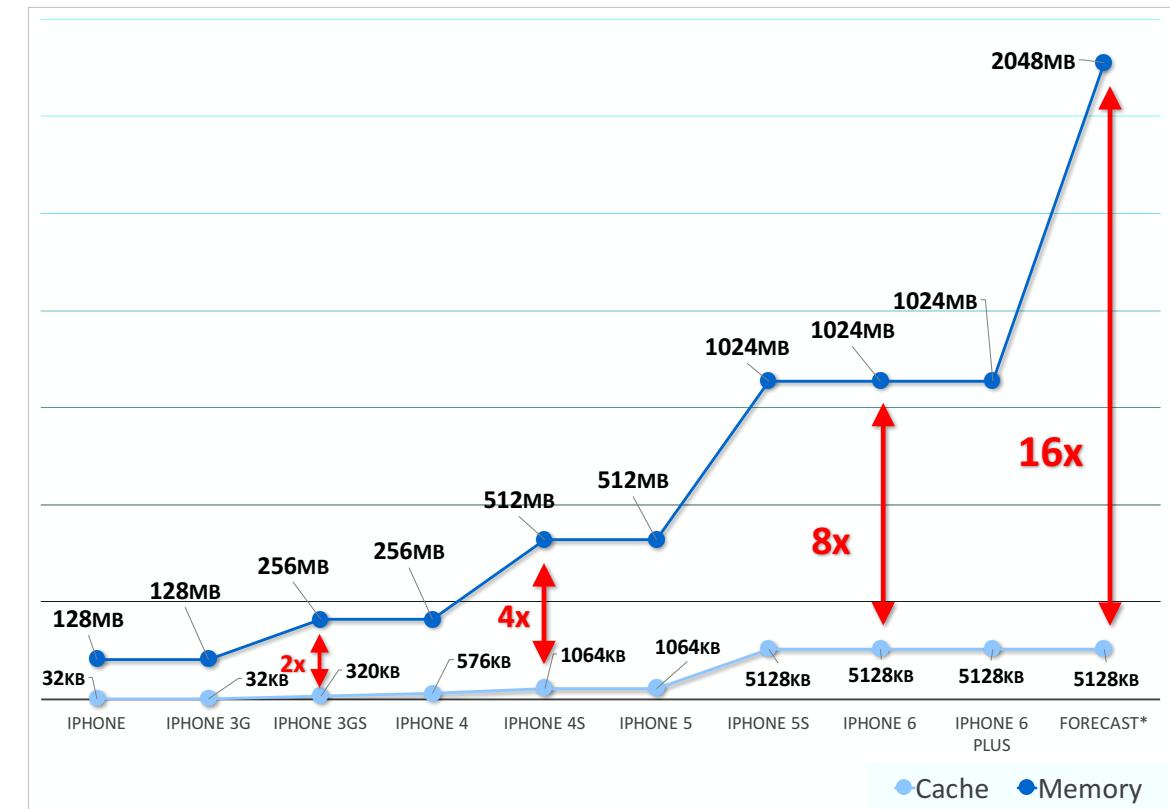
# Off-Chip Memory: The Large Data Set Problem

Application **data footprint** grows more than available on-chip storage

- Cache-based **memory hierarchy** and **virtual memory** solve the problem **in CPU**

Accelerators need **fast memory access** to fetch **long data bursts**

- Private local memories (PLMs) are too large to maintain an inclusive directory and avoid recalls
- Long data transfers exploit little locality and frequently incur eviction penalty
- Address translation with typical page sizes requires extremely large page tables (**equally-sized pages**) or slow translation logic (**scatterlists**) with CPU control



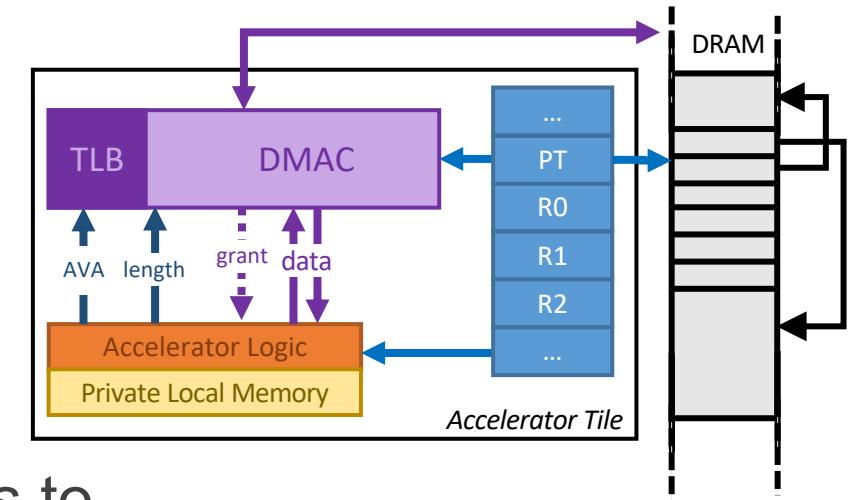
# Scatter-Gather DMA for Accelerators

Allocate **equally-sized large chunks** and create a **small page table**

- Similar to *huge pages*, but with configurable page-size and not allocated at system boot
- Similar to *scatterlist*, but chunks length doesn't need to be stored and page lookup only requires bit selection logic
- Built-in load balancing across DRAM controllers

Dedicated **DMA Controller** with **TLB per tile**

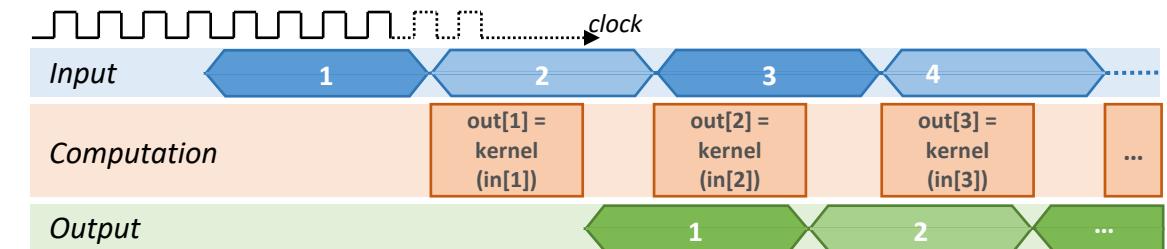
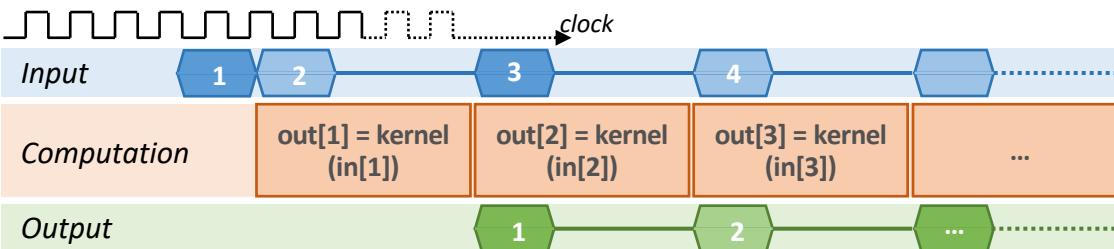
- Fetch entire page table with one DMA transaction
- Hide look-up latency during accelerator computation
- Break long bursts into equally sized DMA transactions to balance links access across accelerators



# What Happens with Multiple Accelerators?

Balancing communication and computation is crucial for performance optimization

- Optimizing microarchitecture reduces the **computation latency**
  - Combination of HLS transformations and PLM customization
- **Input and output phases** interact with the rest of the system
  - Backpressure due to congestion may increase the latency



**Reduce the congestion or exploit the congestion**  
to optimize the execution at the system level

# How to Dynamically Control the Banks

A **scenario** is a given **configuration of the accelerator** to execute a specific problem instance

- E.g.: processing images of different size

PLM units must be sized for the **worst-case scenario**, but they may not be entirely used in all scenarios

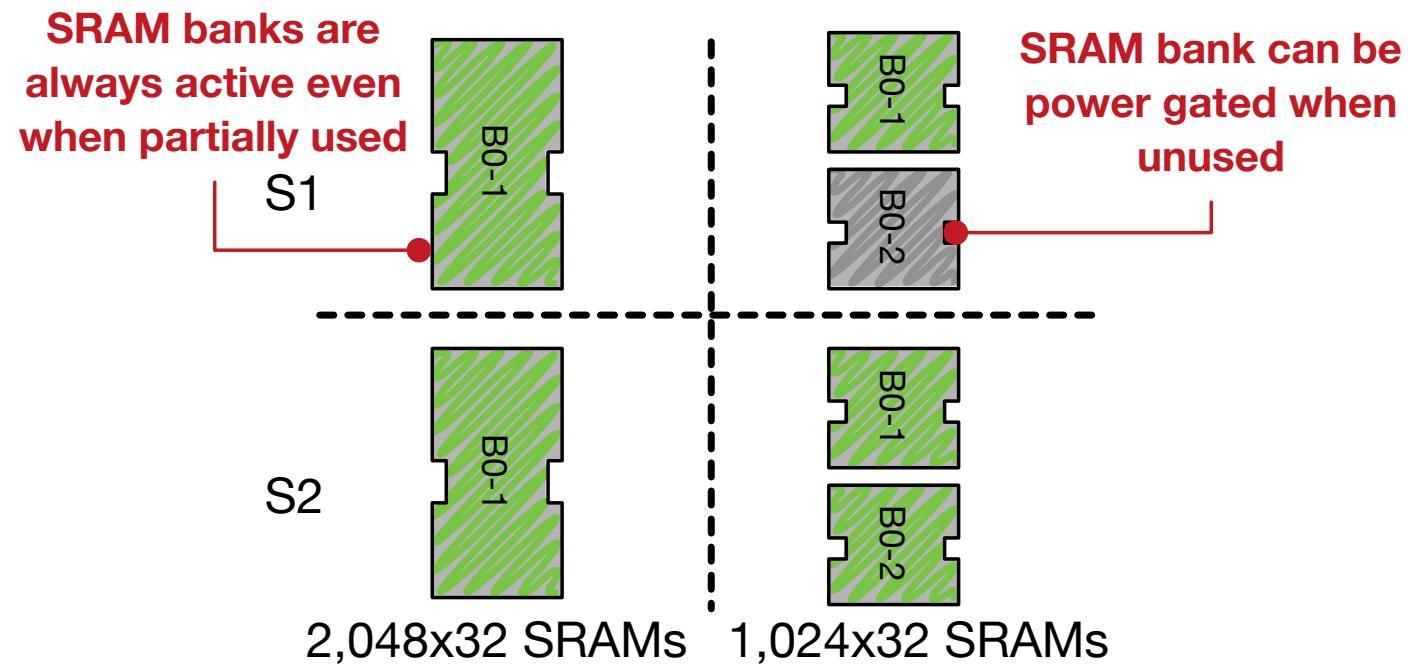
- Possibility of fine-grained power savings

Let us assume an accelerator that can be executed in two scenarios (S1 and S2) with a 50% probability

# Scenario-based Optimization

Each accelerator can turn off the banks that are not entirely used in the current scenario

**Design-time partitioning of the banks** to maximize the ones that are power gated



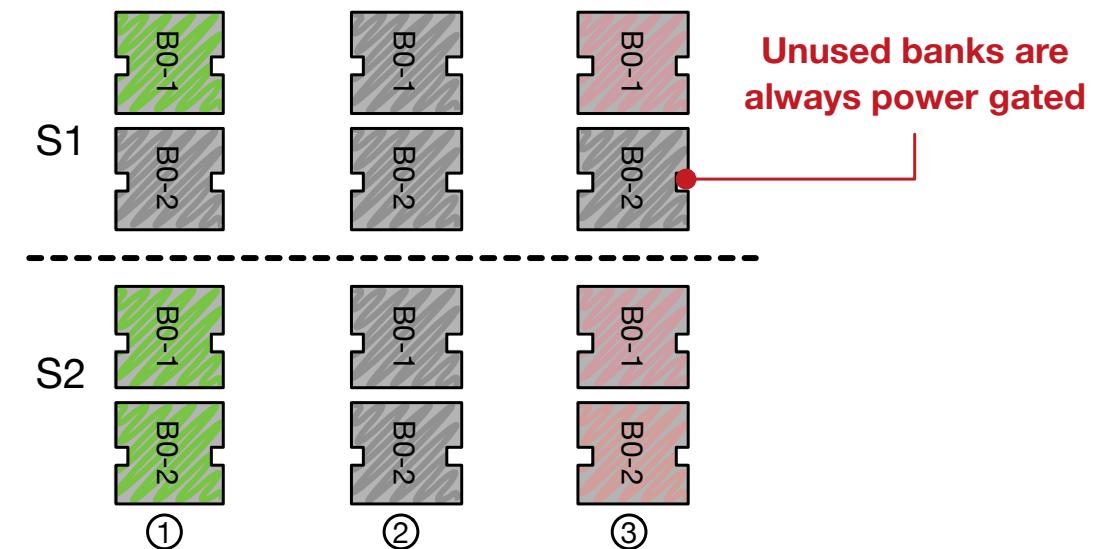
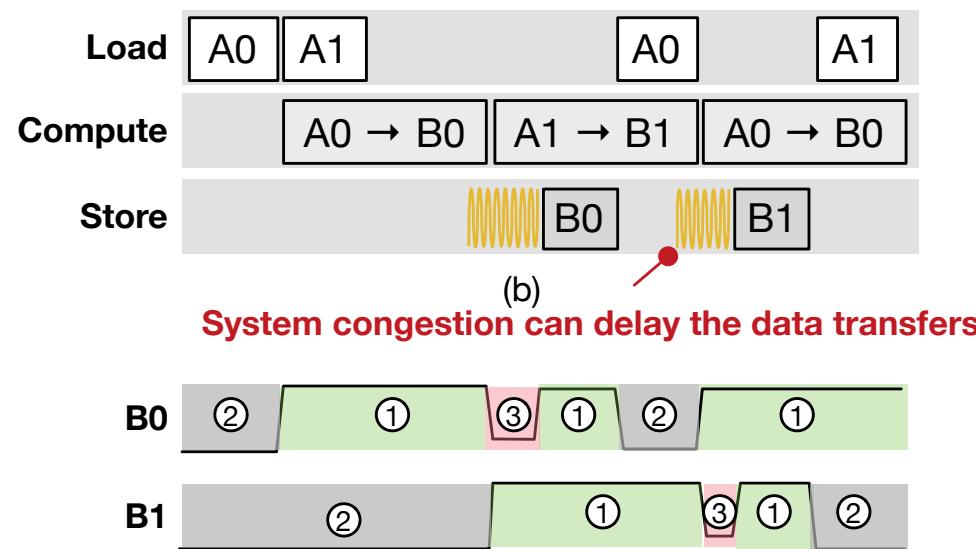
# Workload-based Optimization

System conditions can alter the execution dynamics

- E.g.: System congestion when communicating with the external memory

**Dynamic control of the logic/cell power gating** based on the execution phases

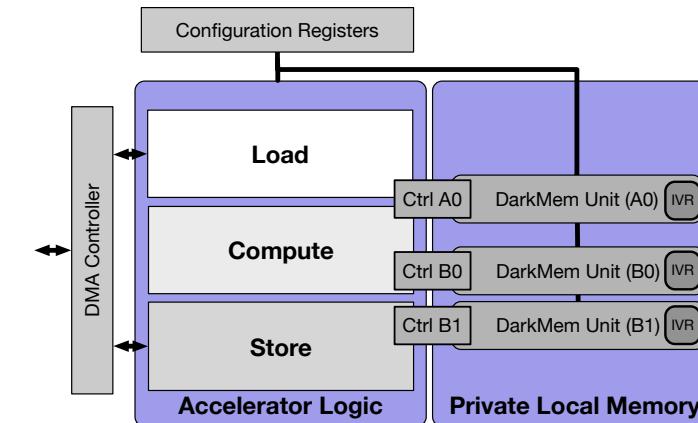
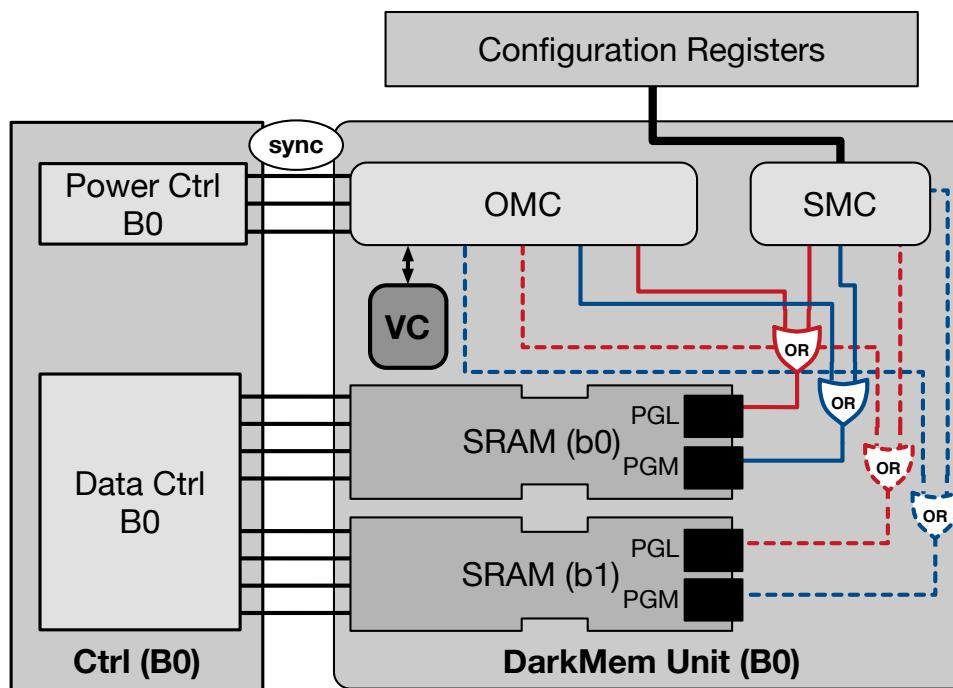
- Three operating modes: *active*, *idle*, *deep-sleep*



# DarkMem Architecture

Each PLM unit can be extended with power-control logic

- **SMC** identifies the current execution scenario (based on the register values)
- **OMC** manages the SRAM operating modes (based on signals from the accelerator logic)



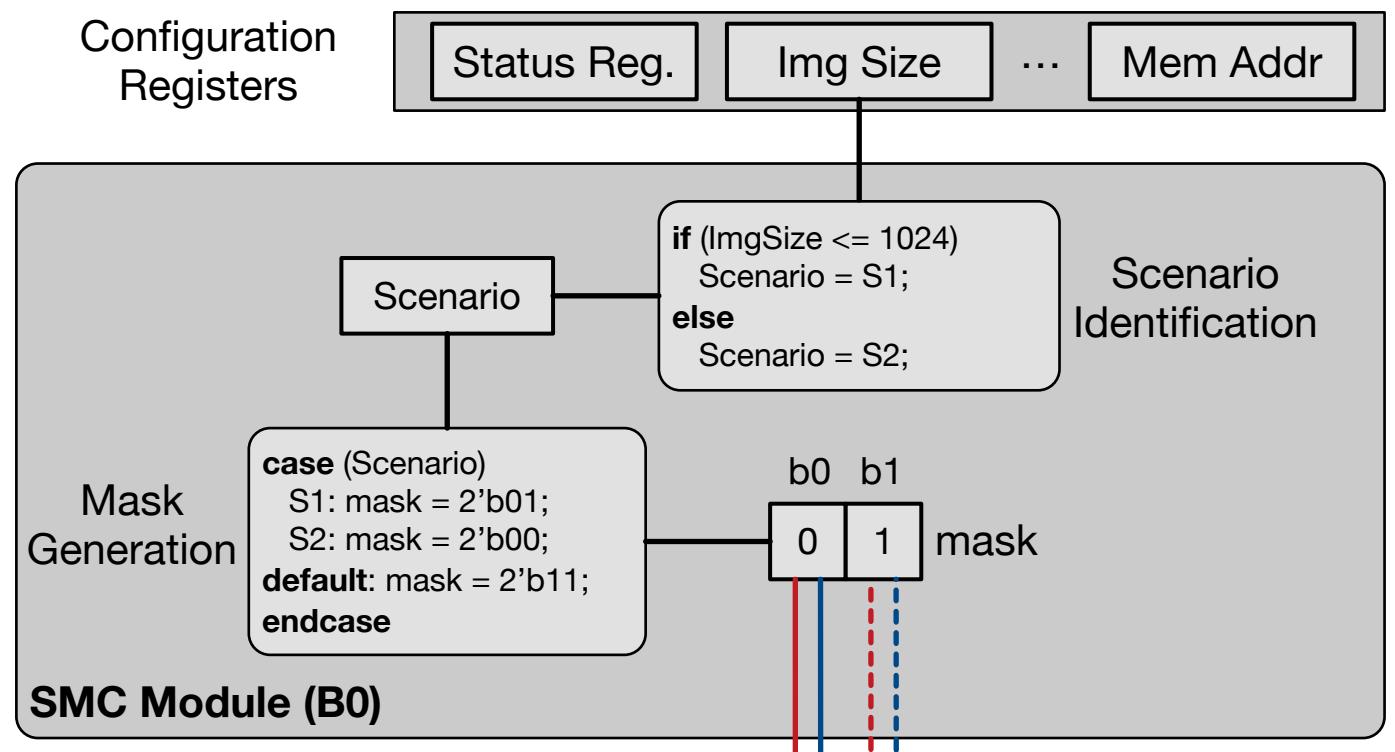
Fine-grained control of each SRAM bank through its power pins (PGL and PGM)

# Scenario Memory Controller

Analyzes the configuration registers (provided by the user with memory-mapped operations)

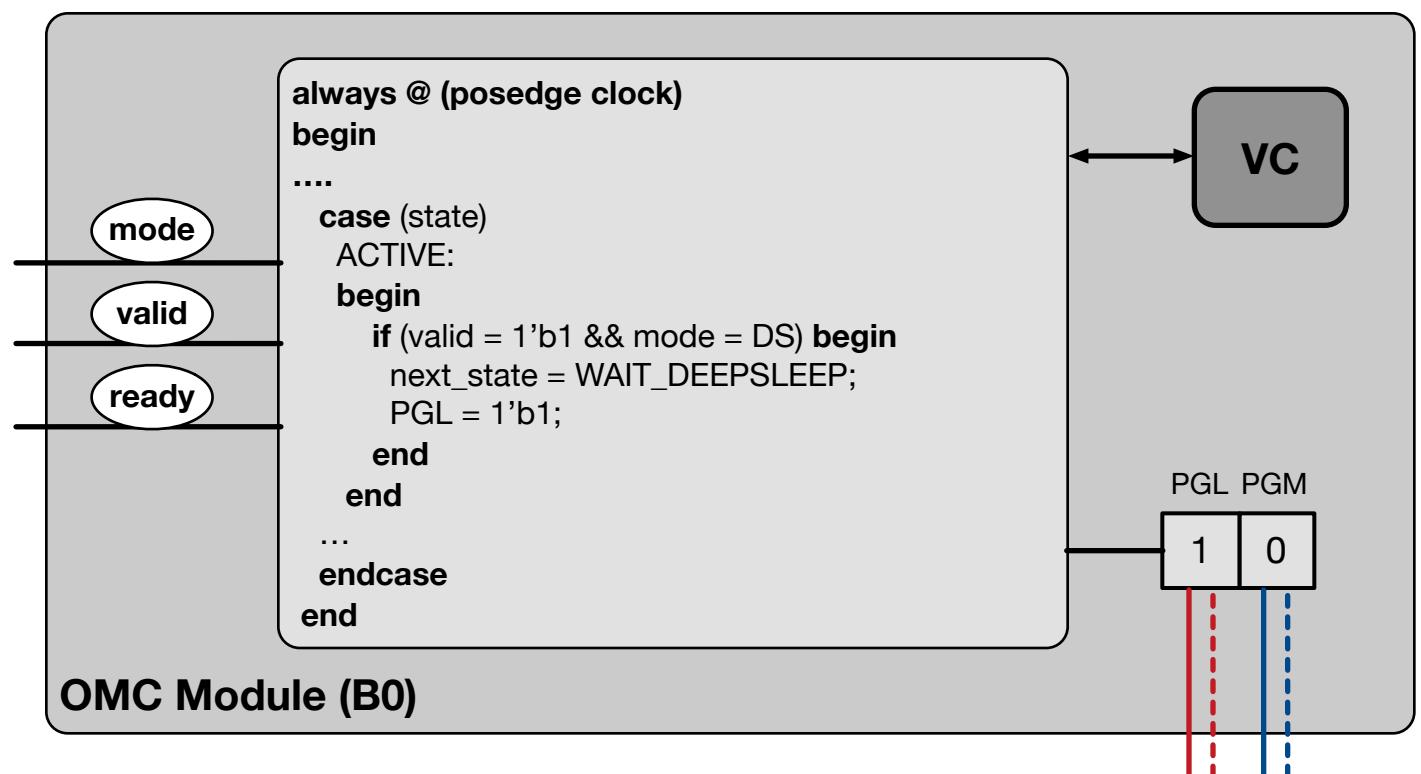
- In each scenario, only the used banks are kept active, while the others are power gated

Mask is one input of the OR gate for each power pin



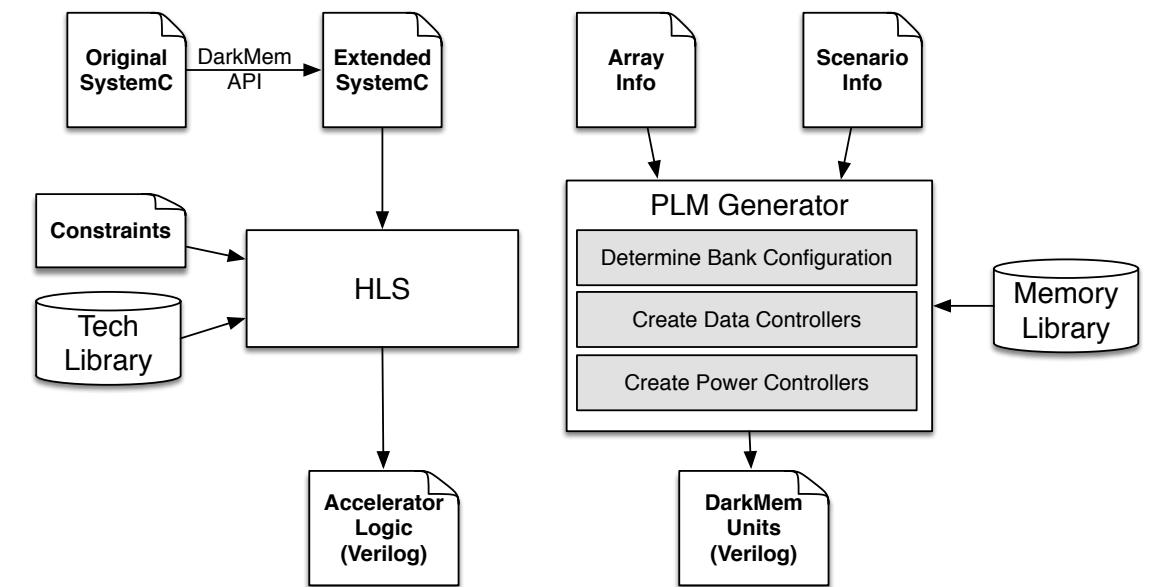
# Operating Mode Controller

- FSM to manage the transitions among operating modes
  - Latency-insensitive protocol with the accelerator logic so that no operations are performed during the transitions
- The supply voltage can be also reduced to DRV
  - Additional power savings in deep-sleep mode
- Resulting values are the other input of the OR gate



# DarkMem Methodology

- HLS-based methodology to generate:
  - Accelerator Logic: DarkMem API to specify operating modes of the data structures directly in SystemC
  - DarkMem units: Extension to PLM CUSTOMIZATION for multi-bank configuration and power-control logic for each PLM unit
- Additional information on the execution scenario
  - Estimated by the designer
  - Always possible to generate a feasible configuration



# Determining the Bank Configuration

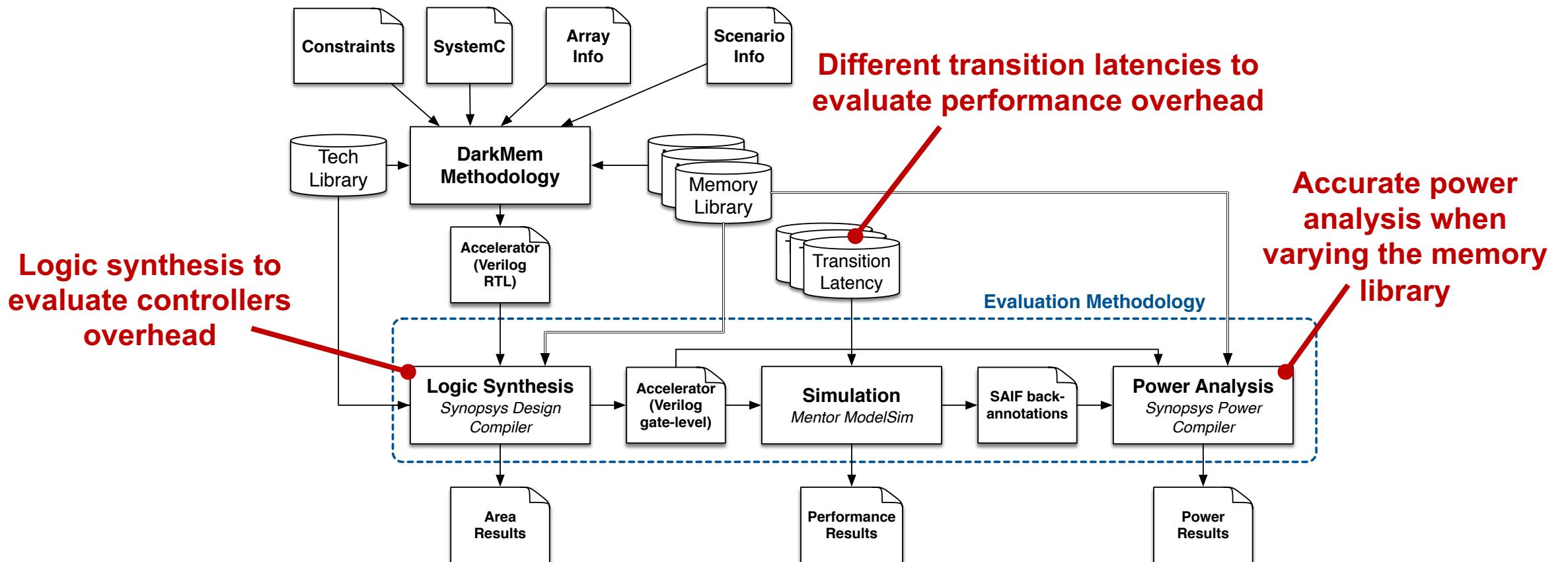
- **ILP formulation** to determine the number and type of banks for each PLM unit, based on:
  - List of scenarios and frequency of execution
  - Data to be stored (bitwidth and number of words) in each scenario
  - List of available memory IPs and corresponding active/gated static power configurations

$$PLM_{static} = \sum_{s \in S} (PLM_{static}^s \cdot freq(s))$$

- Used to determine the banks and accordingly configure the SMC modules to generate the proper masks

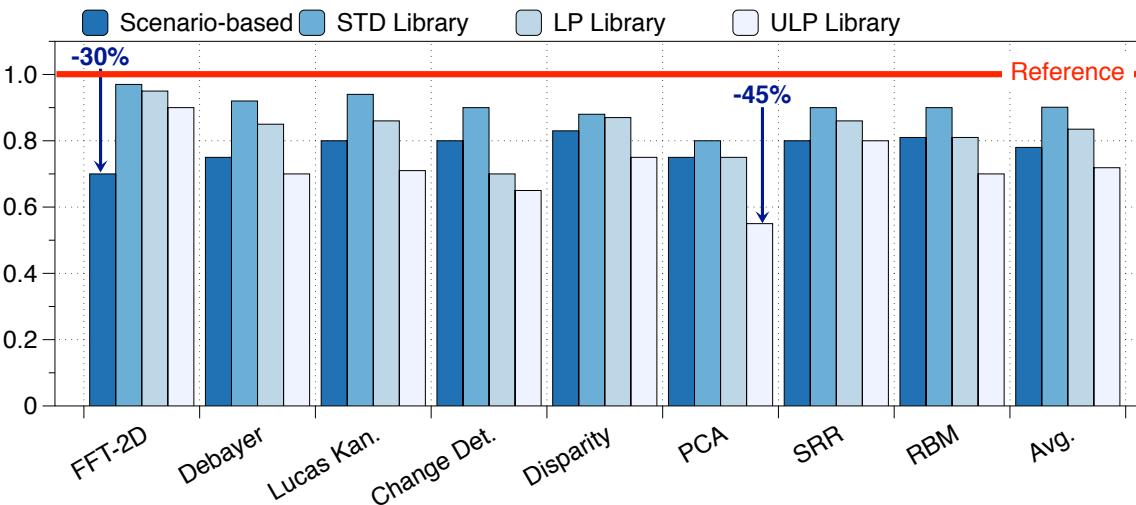
# Evaluation Methodology

- Logic synthesis and gate-level simulation to generate performance results and accurate SAIF backannotations

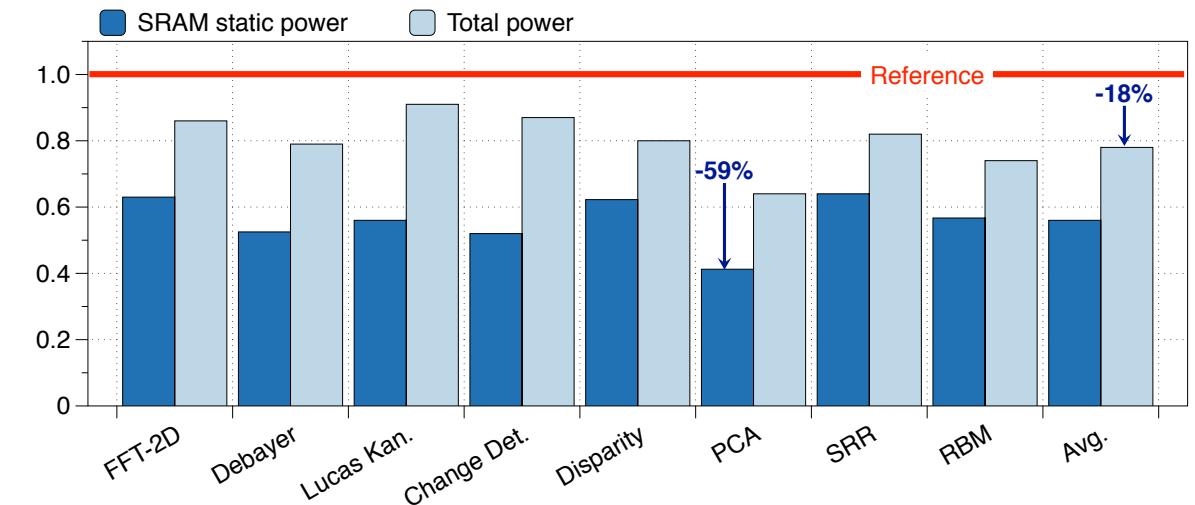


# Effects on Accelerators

- 32nm CMOS technology with pre-defined SRAM banks
- Reference designs are the ones with no power-related optimizations



Performance overhead is minimal (less than 1%)



SRAM static power can be reduced up to 60% (on avg., total power is reduced by about 18%)

# **Design of Hardware Accelerators**

Academic Year 2021/2022

# Questions?

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)



**POLITECNICO**  
MILANO 1863

**Design of Hardware Accelerators**  
Academic Year 2021/2022

# Hardware/Software Co-Design Methodologies

**Christian Pilato**

Dipartimento di Elettronica, Informazione e Bioingegneria

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)

# Designing Heterogeneous Systems

- **Application**: the designer has to:
  - Partition: split the behavior in chunks of operations (tasks) to be potentially executed in parallel
  - Map: assign the tasks to the hardware components
  - Schedule: resolve contention on shared resources
- **Architecture**: usually the designer starts from a template and performs some customization
  - Before fabrication (definition of the physical architecture)
  - After fabrication (configuration of hardware logic)
  - During execution (partial reconfiguration)



# Roadmap for Next-Generation Systems

- **Bottom-up approach** for **compositional design**
  - **Short term**: Development of efficient components (e.g., accelerators) and interconnections
  - **Medium term**: Tools for improving design productivity at system-level
  - **Long term**: Automatic porting of legacy applications on parallel and heterogeneous systems
- **Support of CAD tools** is required
  - Techniques and methods for specific aspects (HLS, memory generation, ...) on the top of existing tools

# Hardware Accelerators and HLS

- Roadmap for next-generation computing systems
  - **Short term:** Development of efficient accelerators
- HLS tools are now (*almost*) able to approach **complex specifications** and generate corresponding accelerators
  - LegUp (Univ. of Toronto supported by Altera), Vivado HLS (UCLA, now part of Xilinx suite), Symphony C Compiler (Synopsys), Stratus (former C-to-Silicon - Cadence), ...



The problem is now about efficiency

# Efficient RTL Architectures

- HLS can now synthesize complex applications
  - **Resource requirements** can become a limitation
- **Resource sharing** is a well-known and adopted technique to control the area occupation
  - **Achieving the minimum number of functional units or registers is not the best solution anymore...**
- **Interconnection** plays an increasingly major role also in RTL architectures
  - Area occupation (*out of resources*)
  - Propagation delay (*clock period violation*)
  - Power consumption (*power budget violation*)

# Effects of Resource Sharing

- HLS performed with **Xilinx Vivado HLS 2013.1**
  - Target: AVNET ZedBoard (XC7Z020-1CLG484 SoC)
    - Dual Arm Cortex-A9
    - Xilinx Artix-7 (85K Logic Cells)
- Simple case study:
  - **Auto-regressive lattice filter** (ARF)
    - 11 additions and 17 multiplications
- Alternative solutions manually generated through synthesis directives in Xilinx Vivado HLS scripts
  - Synthesis frequency: 100MHz (10ns)



# Comparing different solutions

Multipliers	inf	4	4	4	4
Adders	inf	inf	9	7	4
SLICE	286	226	254	200	232
LUT	435	665	813	633	763
FF	997	485	421	421	421
DSP	39	12	12	12	12
BRAM	0	0	0	0	0
SRL	117	36	36	36	36
Critical Path	4.534ns	7.055ns	7.665ns	7.906ns	7.871ns
Control Steps	26	26	26	26	26

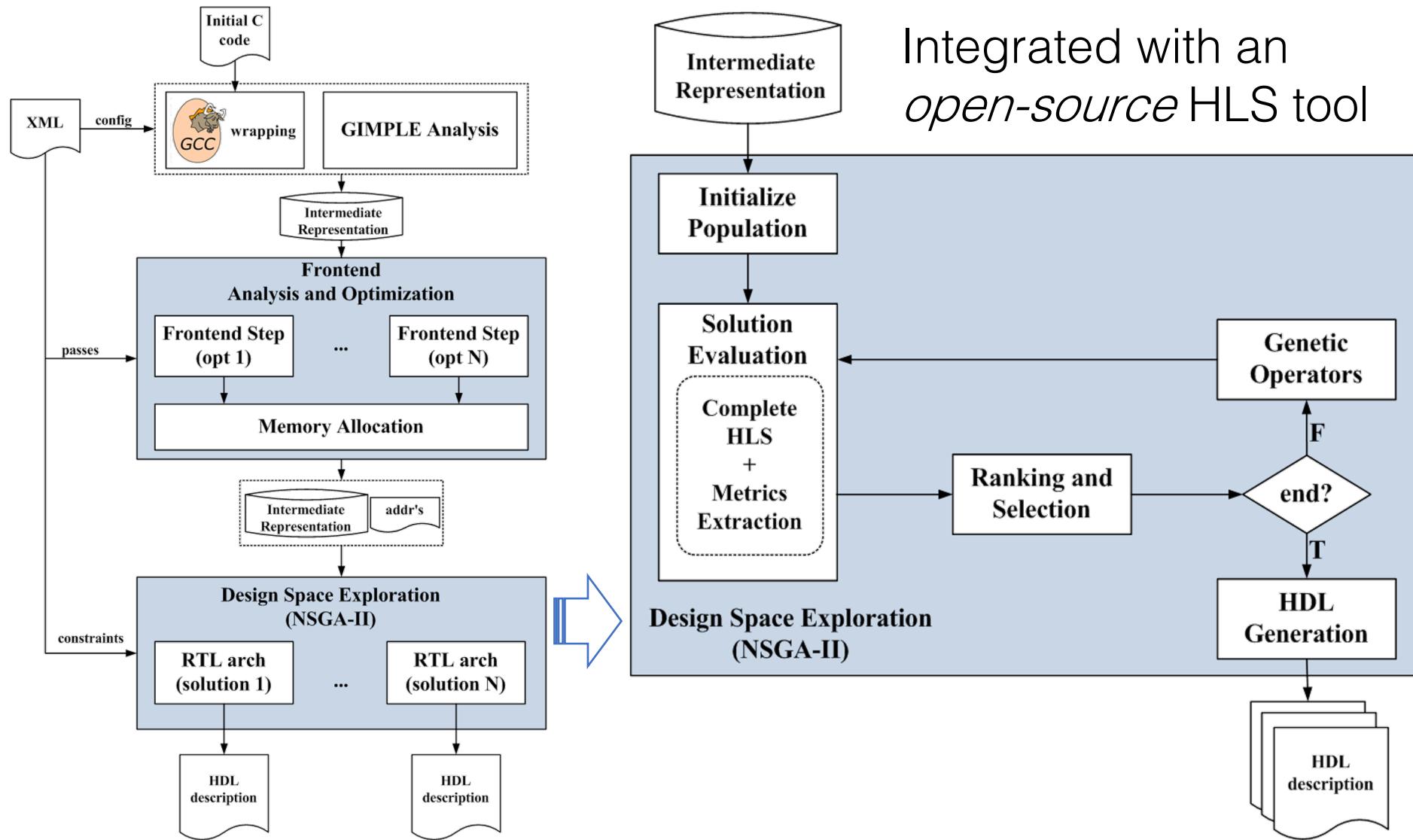
- Difficult to be identified in advance!

HLS is now *push-button*, but...  
you have to push it in the right way!

# Design Space Exploration for HLS

- Different approaches to explore alternative RTL architectures by constraining
  - number of resources or compiler transformations
    - easily applicable on the top of commercial tools
  - scheduling priorities or operation bindings
    - more powerful, but requires tightly integration with synthesis tools
- Interfacing with compilers is usually required to extract and manipulate the intermediate representation
  - **Code rewriting** can also support existing HLS tools by exposing relevant features (later translated into *knobs*)

# Fine-grain DSE for HLS



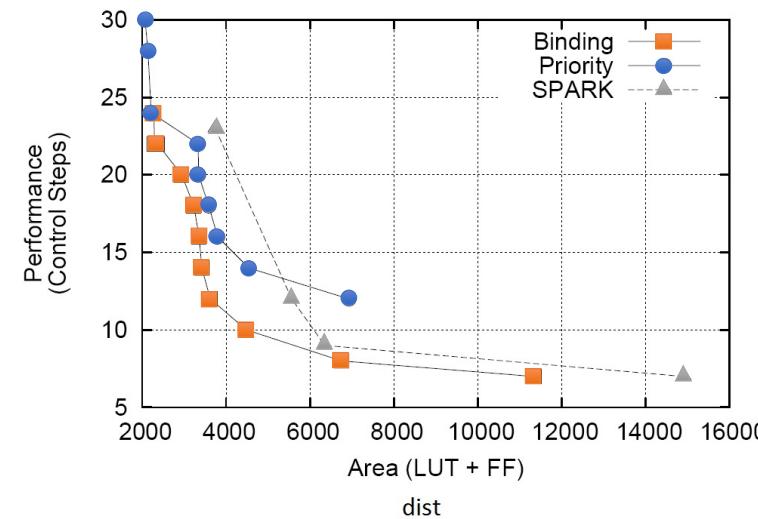
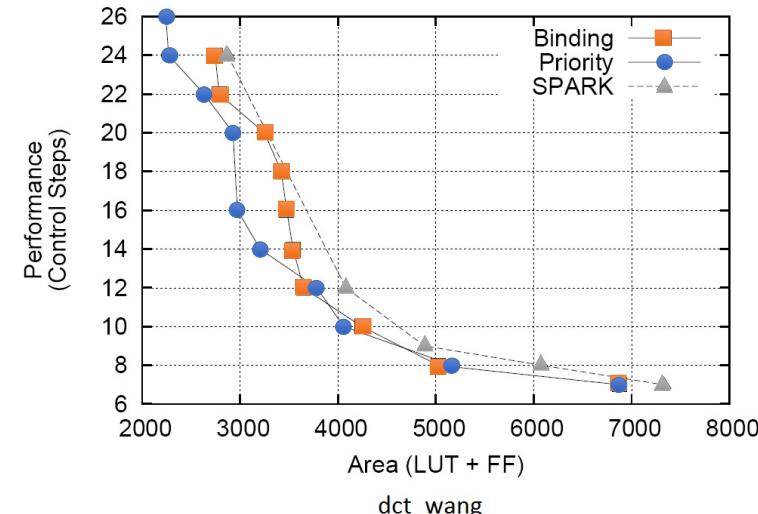
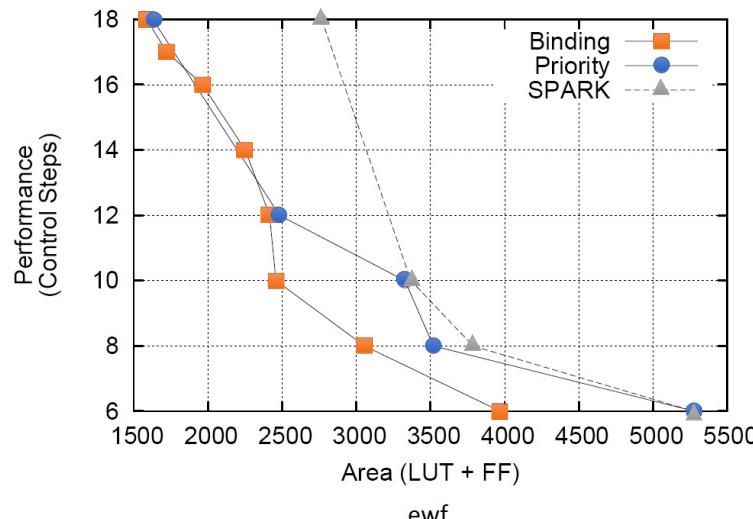
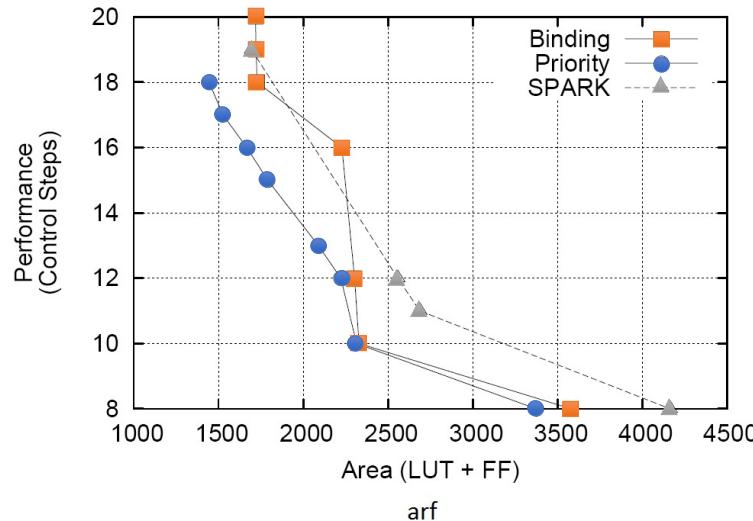
# Creating and Evaluating the Designs

- **Complete HLS** for each generated solution
  - Standard priority-based list scheduling
    - Alternative solutions based on binding/scheduling constraints specified by the encoding
  - Register binding based on state-of-the art algorithms
  - Interconnection optimization to reduce multiplexers
- Actual logic synthesis is too slow to be used in a GA
  - **Estimation model** based on linear regression
  - Approximation error less than 4% in average, but only for small cases and without aggressive optimizations

# Applied to Simple Case Studies

- Traditional benchmarks for HLS (datapath synthesis)
  - ARF, EWF, DCT, DIST
  - comparison with design space exploration with constraints on the resources
- Target: Xilinx Virtex II-PRO XC2VP30 FPGA
  - Final logic synthesis with Xilinx ISE ver. 8.1i
- Genetic parameters:  $N = 1000$ ,  $P_c = 70\%$ ,  $P_m = 30\%$ 
  - ~60,000 designs evaluated per exploration (in average)

# Experimental Results



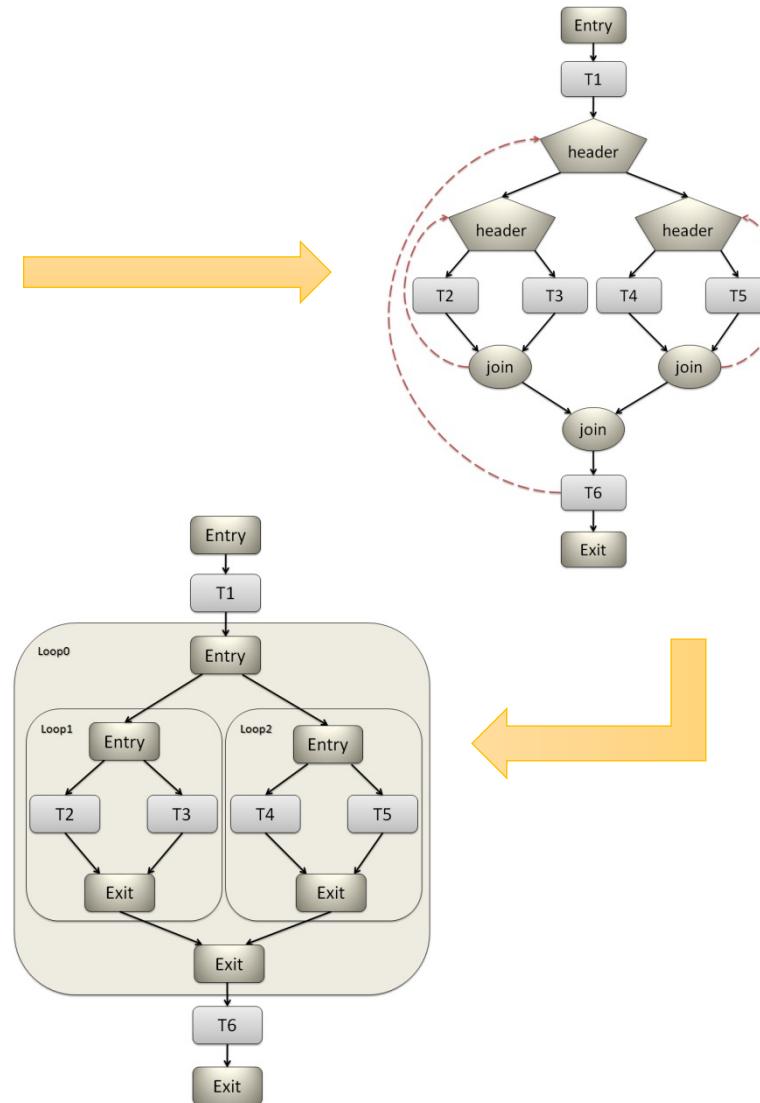
# Lesson Learned...

- **Fine-grain methods** systematically outperform methods based on **resource constraints**
  - **Fine tuning of the RTL architectures**, with a better use of resources
- The approach is **efficient but not scalable** with respect to the size of the specifications
  - Actual logic synthesis is time-consuming
  - New model for each device, synthesis options, ...
  - Accuracy is decreasing when increasing the size

Design space exploration can be efficiently applied only to small portions of the design

# OpenMP Formalism

```
/* task T1*/
while(/*condition Loop0*/){
    #pragma omp parallel sections default(shared) num_threads(2)
    {
        #pragma omp section
        {
            while(/*condition Loop1*/{
                #pragma omp parallel sections default(shared) num_threads(2)
                {
                    #pragma omp section
                    { /* task T2 */}
                    #pragma omp section
                    { /* task T3 */}
                }
            }
        }
        #pragma omp section
        {
            while(/*condition Loop2*/{
                #pragma omp parallel sections default(shared) num_threads(2)
                {
                    #pragma omp section
                    { /* task T4 */}
                    #pragma omp section
                    { /* task T5 */}
                }
            }
        }
    }
}/* task T6 */
```



# Semantic has to be maintained

```
int main()
{
    printf("Hello ");
    printf("World ");
    return 0;
}
```

Expected output:  
Hello World

Invalid output:  
World Hello

# Correctness

*Tools must preserve the observable behavior of the program.*

- *Observable behavior*
  - Consuming bytes of input.
  - Program output.
  - Program termination.
  - etc.
- Output can be slightly different
  - Different precision
  - Associativity of operations
- Compiler **must prove** that a transform preserves observable behavior.
  - Same side effects, in the same order.
- In absence of a proof, the compiler must be **conservative**.
  - Loss of parallelism!

# Analysis of Dependencies

- Sequential languages present a **total order** of the program statements.
- Only a **partial order** is required to preserve observable behavior.
- The partial order must be discovered.

```
float foo(a, b)
{
    float t1 = sin(a);
    float t2 = cos(b);
    return t1 / t2;
}
```



# Analysis of Dependencies

- Although  $t_1$  appears before  $t_2$  in the program...
- Re-ordering  $t_1$ ,  $t_2$  cannot change observable behavior.

```
float foo(a, b)
{
    float t1 = sin(a);
    float t2 = cos(b);
    return t1 / t2;
}
```

# Dependence Analysis

- Source-code order is pessimistic.
- Dependence analysis identifies a more precise partial order.
- This gives the compiler freedom to transform the code.

# Analysis is incomplete

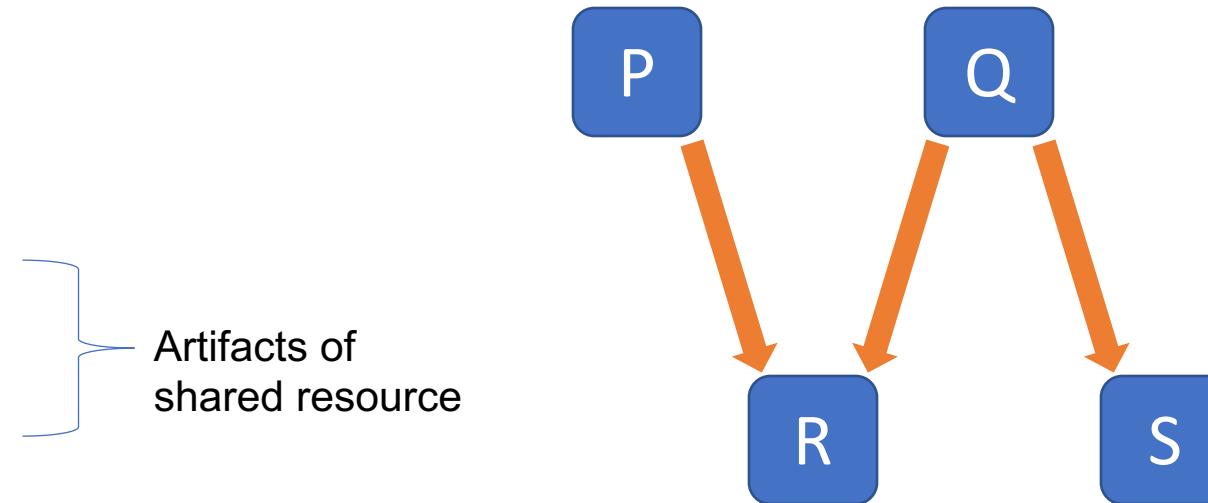
- Results of the analysis:
  - a **precise** answer in the **best case**
  - a **conservative** approximation in the **worst case**.
- Conservative approximations can be partially removed by increasing analysis effort
- Conservative approximations are false dependences
  - Can reduce the available parallelism

# Program Order from Data Flow

- Data Dependence
  - One operation computes a value which is used by another

$$\begin{aligned} P &= \dots; \\ Q &= \dots; \\ R &= P + Q; \\ S &= Q + 1; \end{aligned}$$

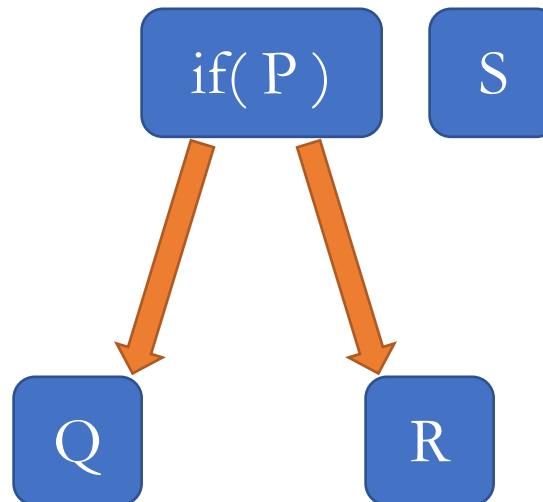
- Sub-types
  - Flow—Read after Write
  - Anti—Write after Read
  - Output—Write after Write



# Program Order from Control Flow

- Control Dependence
  - One operation may enable/disable the execution of another
- Dependent:
  - P enables Q or R.
- Independent:
  - S will execute in any case

```
if( P )  
Q;  
else  
R;  
S;
```



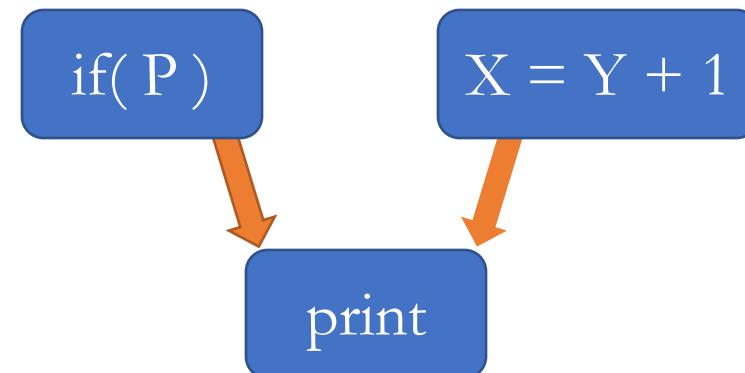
# Removable Control Dependence: Example

- The effect of X is local to this region.

- Executing X outside of the if-statement cannot change behavior.

- X independent of the if-statement.

```
if( P )  
{  
    X = Y + 1;  
    print(X);  
}
```

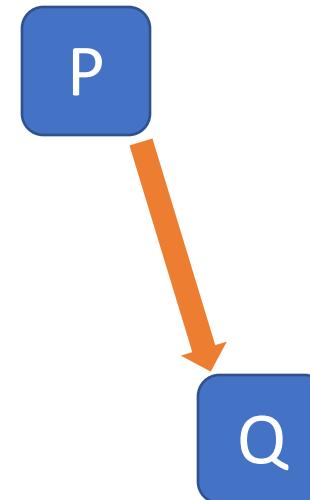


# Program Order from SysCalls

- Observable behavior is accomplished via system calls

```
print (P)  
;  
print (Q)  
;
```

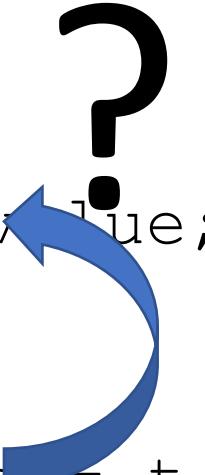
- Very difficult to prove that system calls are independent
  - Side-effects



# Analysis is non-trivial

- Consider two iterations
- Earlier iteration: B stores
- Later iteration: A loads
- Dependence?

```
n = list->front;  
  
while( n != null )  
{  
    // A  
    t = n->value;  
    // B  
    n->value = t+1;  
  
    // C  
    n = n->next;  
}
```



# The Program Dependence Graph

- A directed multigraph [Ferrante et al, 1987]
  - Vertices: operations; Edges: dependences.
- Pros:
  - Dependence is explicit.
- Cons:
  - Expensive to compute:  $O(N^2)$  dependence queries
  - Loop structure not always visible.

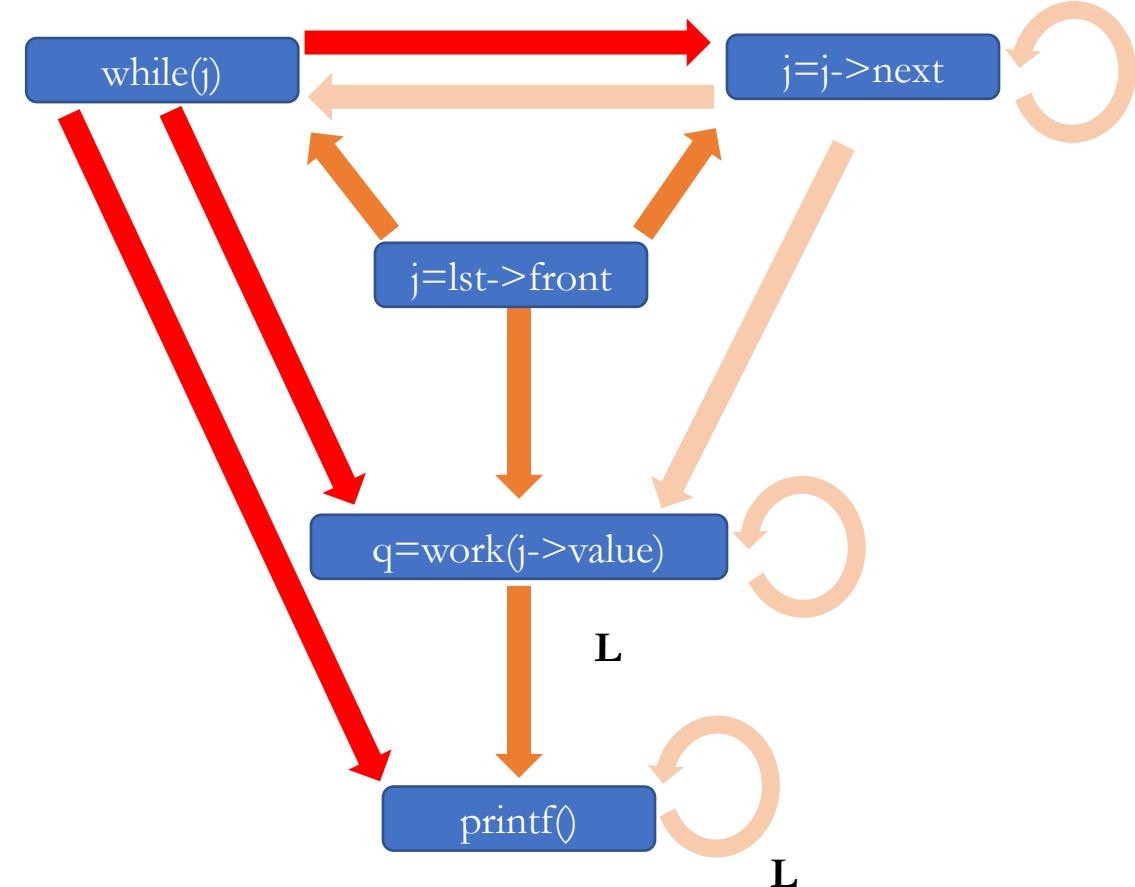
# PDG Example

```
void foo(LinkedList *lst)
{
    Node *j = lst->front;

    while( j )
    {
        int q=work(j->value);

        printf("%d\n", q);

        j = j->next;
    }
}
```



# Task Partitioning

- **Definition:** The **partitioning problem** is to assign
  - $n$  **objects**  $O=\{o_1, \dots, o_n\}$  to
  - $m$  **blocks** (also called **partitions**)  $P=\{p_1, \dots, p_m\}$
- such that
  - $p_1 \cup p_2 \dots \cup p_m = O$
  - $p_i \cap p_j = \emptyset$  for all  $i \neq j$ , and
  - cost  $c(P)$  is minimized.
- **Cost function** (Estimated) quality of design, may include
  - System price
  - Latency
  - Power consumption

# Partitioning Methods

- Cost solution difficult to evaluate
  - Tasks can be assigned to different processing elements
- Exact methods
  - Enumeration
  - Integer Linear Programming (ILP)
- Heuristic methods
  - Constructive methods
    - Random mapping
    - Hierarchical clustering
  - Iterative methods
    - Kernighan-Lin Algorithm
    - Simulated Annealing

# Partitioning Methods

- **Integer Linear Programming**
  - NP-complete.
  - Running times depend exponentially on problem size, but problems of >1000 vars solvable with good solver (depending on the size and structure of the problem)
- **Random mapping**
  - Each object randomly assigned to some block
  - Used to find starting partition for iterative methods

# Partitioning Methods

- **Hierarchical clustering**
  - Assumes closeness function: it determines how desirable it is to group two objects
  - Start with singleton blocks
  - Repeat until termination criterion (e.g., desired number of blocks reached)
    - Compute closeness of blocks (average closeness of object pairs)
    - Find pair of closest blocks
    - Merge blocks
  - Difficulty: find proper closeness function

# HW/SW Partitioning

- Special case: Bi-partitioning  $P=\{p_{SW}, p_{HW}\}$
- Software-oriented approach:  $P=\{\emptyset, \emptyset\}$ 
  - In software, all functions can be realized
  - Performance might be too low  $\Rightarrow$  migrate objects to HW
- Hardware-oriented approach:  $P=\{\emptyset, O\}$ 
  - In hardware, performance is OK
  - Not all functions can be implemented in hardware
  - Cost might be too high  $\Rightarrow$  migrate objects to SW

# How to make it scalable?

- Monolithic hardware accelerators are going to be replaced by **IP-based systems**
  - Decomposing the specification in blocks
  - Applying more aggressive local optimizations and explorations
    - Generation of alternative implementations
  - Isolating portions that cannot be implemented in hardware
- Additional **degree of freedom** at system-level
  - How to explore and compose the blocks?
  - How to control the dynamic execution?

# System-Level Composition

- Roadmap for next-generation computing systems
  - Short term: Development of efficient accelerators
  - **Medium term:** Tools for improving design productivity
- When designing an IP block, it is not usually possible to predict the best implementation *w.r.t.* other blocks
  - **System-level analysis** to determine the best combination of IP implementations
- Runtime optimizations based on the analysis of interactions/correlations among blocks
  - **Adaptive and run-time synchronization** of blocks
  - **RTL optimizations** for performance and power

# Which is the current scenario?

- **Component-based systems** are crucial for scalability, but difficult to be designed
  - Possibility for increasing designer productivity (reuse)

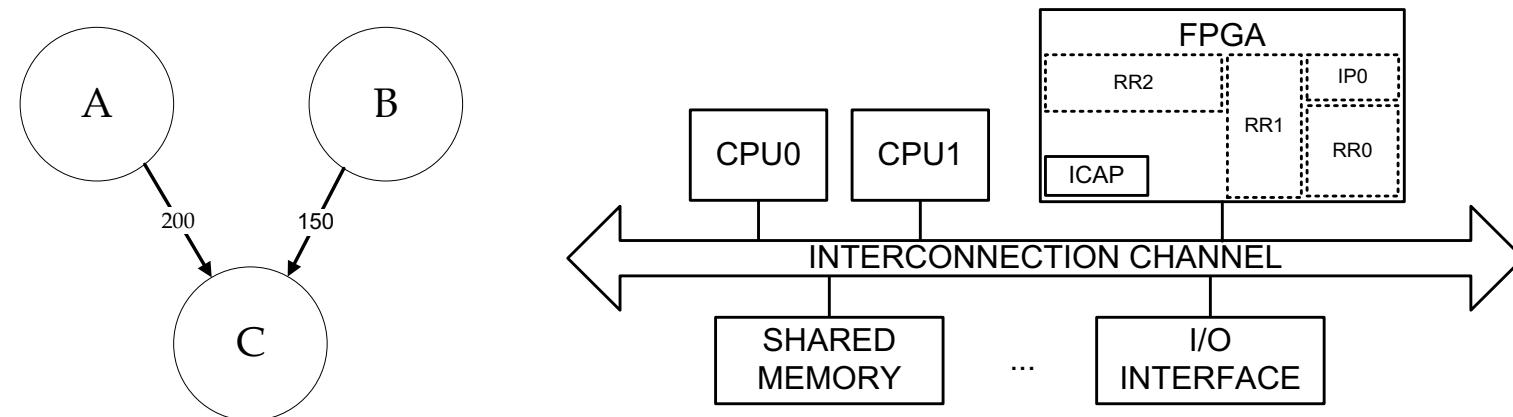
HLS methods are pushing towards standardization,  
but with a tight integration with vendor technology

- Almost impossible to create *fully portable* IP blocks
  - Variation effects, porting between technology nodes, integration with emerging technology, ...

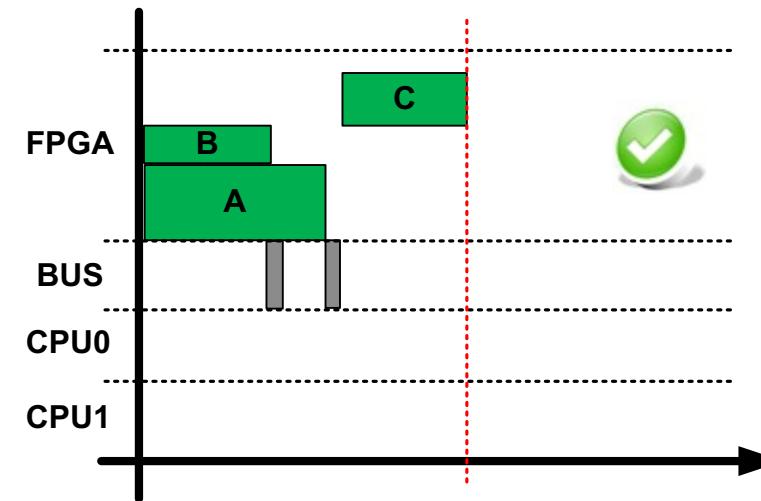
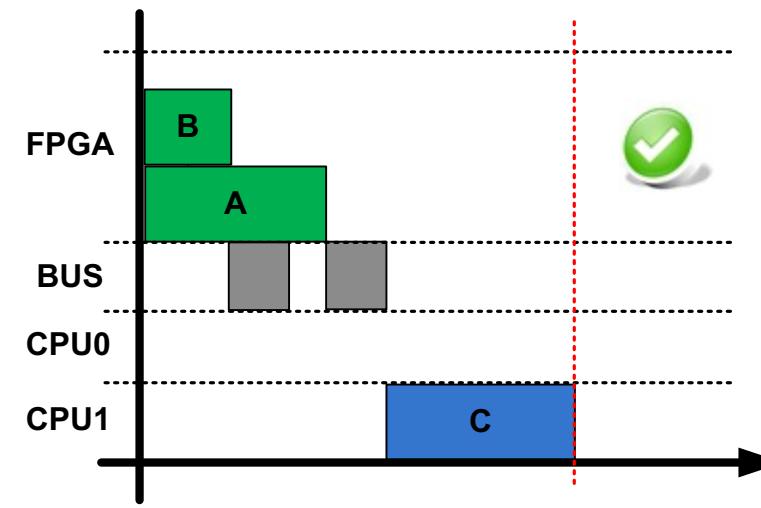
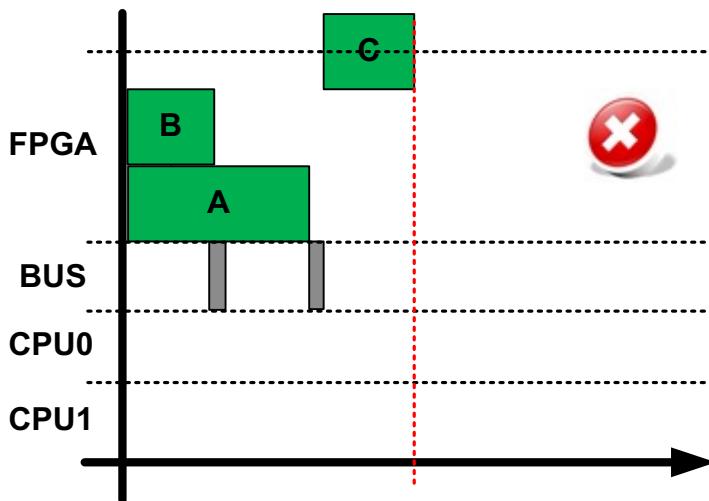
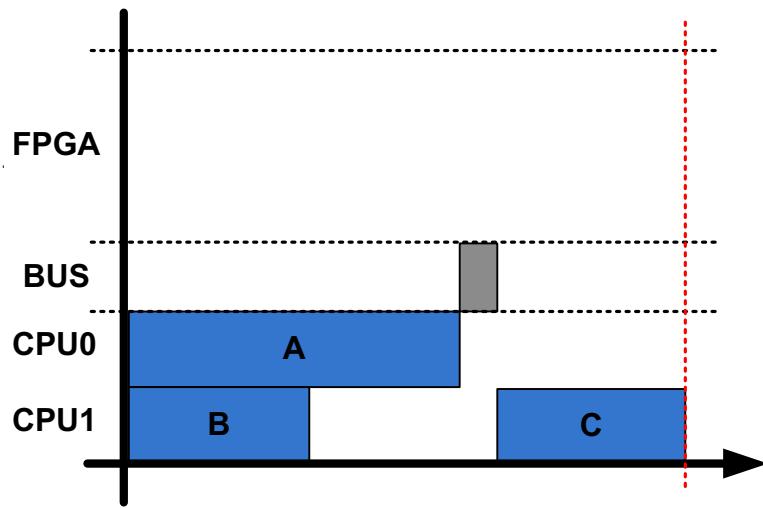
Automated methods to efficiently use  
existing tools for creating complex systems

# Motivational Example

- **Application task graph** with multiple implementations for each task
  - No restrictions on the model of computation
- Embedded **architectural template** composed of
  - General purpose processors
  - Area dedicated to custom hardware accelerators
  - System bus with shared memory or DMA



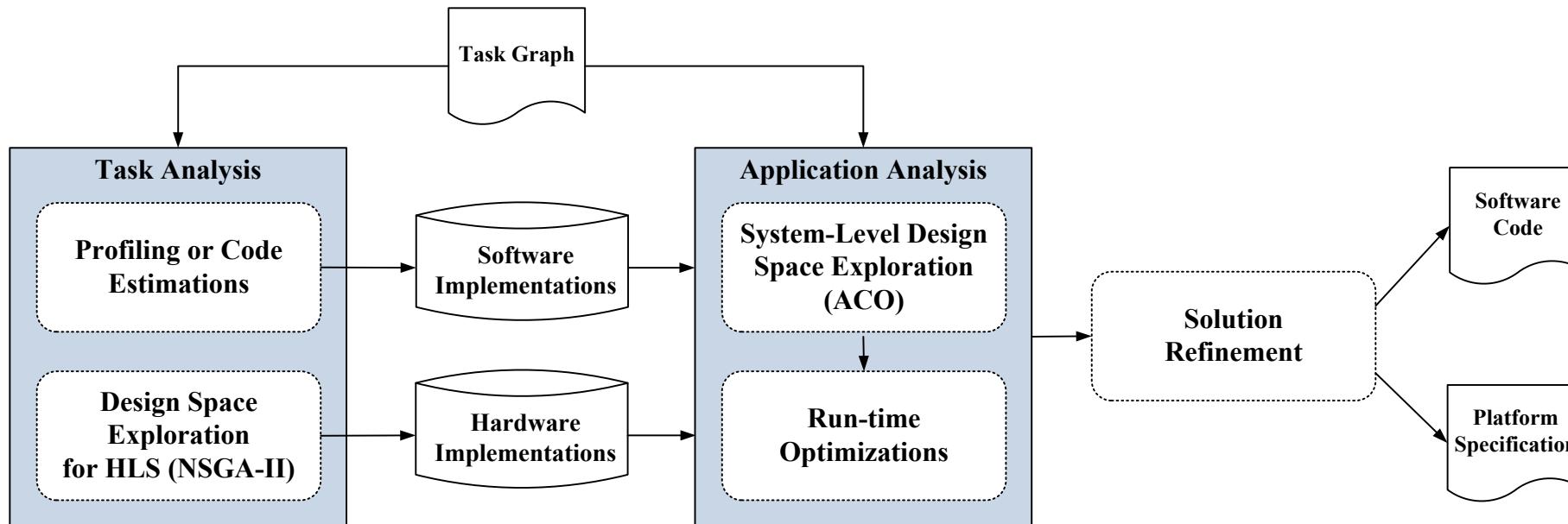
# Alternative Implementations



# Proposed Co-Design Flow

**Mapping** concerns the assignment of a task to:

- An implementation (mode of execution, resource trade-off)
- A processing element (hardware resource for execution)



**Scheduling** concerns the order of executing tasks whenever there is contention on the resources

# System-level Composition

- Novel **automated methodology** for the design and the optimization of heterogeneous embedded systems
  - Perform **mapping and scheduling** of partitioned applications considering tasks and communications
    - Support for multiple hardware implementations
    - Decisions are taken only at system level to trade-off resource requirements among all the blocks
    - Different communication topologies are supported
  - Possibility of co-exploring architecture and application without violating the design constraints
    - Limited area for hardware devices
    - Support for partial dynamic reconfiguration

# Exploring Mapping and Scheduling

- Exploration based on the **Serial Generation Scheme (SGS)**
  - For each candidate solution, constructive approach to better handle design constraints
    - Decision is not taken if it would lead to a constraint violation
- **Different combinations** of mapping and scheduling
  - Each decision represents a mapping of a task with respect to an implementation and a processing element
  - The order of selection represents the priority values for resolving scheduling conflicts on the resources

# Ant Colony Optimization

- Our proposed approach is based on **Ant Colony Optimization (ACO)** to limit unfeasible solutions
  - Cooperative behavior of the ants while searching
  - The ant has different possibilities at each step and takes stochastic decisions, composing a **trace**
- Stochastic principles guarantee exploration (a probability is generated for each admissible decision at each step)
- Feed-back guarantees the exploitation of good parts of the solutions

# Algorithm Overview

- Pseudo-code of the proposed ACO-based exploration:

```
1: generate initial solution  
2:  $Z^0 \leftarrow \text{Eval}(S_0)$   
3:  $Z^* \leftarrow Z_0$   
4: initialize pheromone values with  $1/Z_0$   
5: for each ant  $l$  into the colony  $L$  do  
6:   initialize candidate  
7:   while candidate is not empty do  
8:     select and assign activity  $j$  to  $i$   
9:     update candidate  
10:    end while  
11:    $Z_l \leftarrow \text{Eval}(S_l)$   
12:   if  $Z_l < Z^*$  then  
13:      $Z^* \leftarrow Z_l$   
14:   end if  
15: end for  
16: if exploration is not terminated then  
17:   update pheromone values  
18:   perform random move with probability  $p_r$   
19:   return to 5  
20: end if  
21: return  $Z^*$ 
```

**Exploration:** trace generation

selection of the candidate

**Exploitation:** update of global information

# Stochastic Selection Process

- At each decision point **d**, the probability to assign a candidate **j** (task/communication) to a proper implementation point **i** (implementation+processing element) is:

$$p_{d,j,i} = \frac{\text{global heuristic } [G_{d,j,i}]^\alpha \cdot [L_{d,j,i}]^\beta}{\sum_{k,n} [G_{d,j^k,i^n}]^\alpha \cdot [L_{d,j^k,i^n}]^\beta} \quad \text{local heuristic}$$

There is the possibility of adding a new PE or reusing an existing one (**platform customization**)

- **Global information G**: feedback information
  - Probability that the decision leads to a good solution
- **Local heuristic L**: problem-specific hint
  - “Adjusted” by the global heuristic if wrong
- Roulette wheel and extraction of a combination **i, j**

# Trace Generation and Evaluation

- Evaluation is performed only on the **complete trace**
  - Updated version of the original TG augmented with communications and reconfigurations
    - Reconfiguration is considered from the early stages of the design process
- Possibility to include different evaluation methods
  - Analytical estimations, TLM simulations, or cycle-accurate simulators
- Decisions composing the best solution are reinforced
  - As the time goes, the **best trace** is identified

# Comparison with other heuristics

- The proposed ACO-based solution has been validated and compared with:
  - Mathematical formulation (ILP) – only DAGs
  - Tabu Search (TS)
  - Genetic Algorithm (GA)
  - Simulated Annealing (SA)
- Randomly generated task graphs (TGFF) to evaluate:
  - **Quality** with respect to different graph topologies
  - **Scalability** of the approach

# Experimental Results

- ACO is **much faster to reach the optimum value**

- it has a **reduced deviation** of the solutions

Number of evaluations to reach the optimum value

Task/ Edge	ACO	SA	TS	GA
5/4	318	4,721	2,555	1,020
10/9	2,931	(+27.00%)	80,887	23,172
10/13	4,631	(+19.64%)	41,497	(+6.07%)
10/12	4,100	(+14.42%)	25,093	10,798
15/26	12,505	(+48.02%)	17,277	(+7.01%)

Percentage of unfeasible solutions

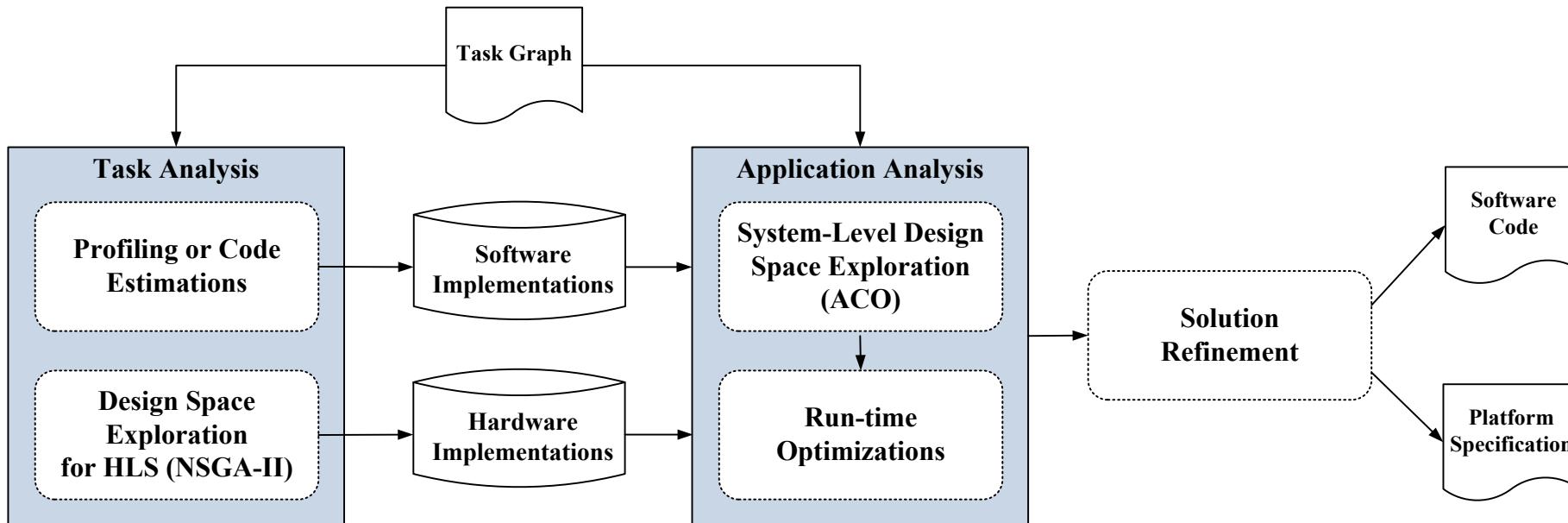
	ACO	SA	TS	GA
Avg.	0.33%	97.95%	14.49%	19.80%

# Lesson Learned...

- Possibility of creating **custom embedded systems**
  - Support for multiple implementations of the same task
  - Exploration of alternative system organizations
    - Support for multiple architectural templates
- Usually not all the blocks are active at the same time
  - E.g., Inactive blocks can benefit of clock-gating for reducing power consumption

Analysis made at design time can be adopted  
to introduce **run-time optimizations**

# Conclusions



- **Integrated and modular methodology** to explore the application implementation at system-level
  - Possibility to include efficient run-time optimizations

Initial application partitioning  
heavily affects the generated systems

# Future Work: Automatic Partitioning

- Roadmap for next-generation computing systems
  - Short term: Development of efficient accelerators
  - Medium term: Tools for improving design productivity
  - **Long term:** Automatic porting of legacy applications on parallel and heterogeneous systems
- **Automatic partitioning** of the specification, fully combined and integrated with
  - Exploration of alternative implementations
    - interfacing with existing HLS tools
  - Co-exploration of computation, communication and storage (e.g. how to specialize the entire memory hierarchy?)
  - System-level analysis and run-time optimizations

# **Design of Hardware Accelerators**

Academic Year 2021/2022

# Questions?

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)



**POLITECNICO**  
MILANO 1863

**Design of Hardware Accelerators**  
Academic Year 2021/2022

# Communication Synthesis

**Christian Pilato**

Dipartimento di Elettronica, Informazione e Bioingegneria

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)

# Multiprocessors communication mode

- **Shared address:** offer the programmer a single memory address space that all processors share. Processors communicate through shared variables in memory, with all processors capable of accessing any memory location via loads and stores.
- **Message passing:** Communicating between multiple processors by explicitly sending and receiving information.

# Two types shared address access

- **Uniform memory access multiprocessors (or symmetric multiprocessors)**: which takes the same time to access main memory no matter which processor requests it and no matter which word is requested.
- **Nonuniform memory access multiprocessors**: some memory accesses are faster than others depending on which processor asks for which word.
- For non-uniform memory access machines can scale to larger sizes and hence are potentially higher performance.

# Two basic physical connections

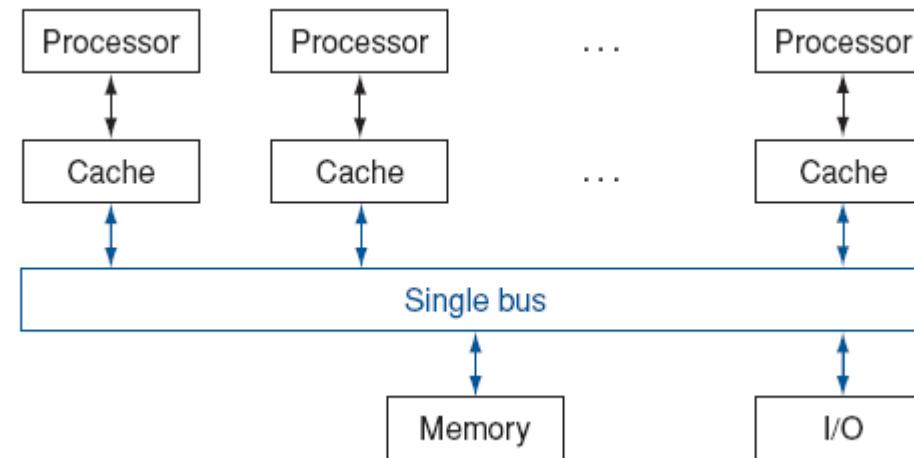
- Processors connected by a **single bus**
- Processors connected by a **network** (e.g., NoC)

Category	Choice		Number of processors
Communication model	Message passing		8–2048
	Shared address	NUMA	8–256
		UMA	2–64
Physical connection	Network		8–256
	Bus		2–36

Options in communication style and physical connection for multiprocessors as the number of processors varies.

# Multiprocessors connected by a single bus

- Low area occupation
- Simple to interface with components



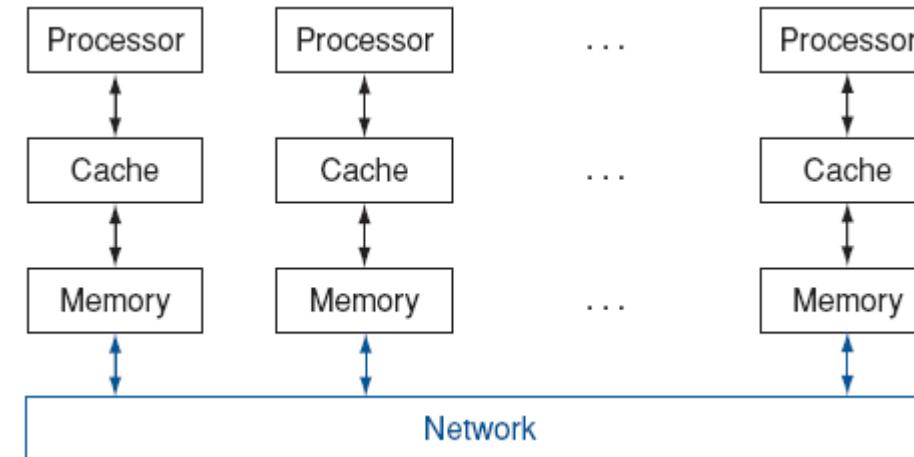
---

Typical size is between 2 and 32 processors

**Cache coherency:** consistency in the value of data between the versions  
In the caches of several processors.

# Multiprocessors connected by a network

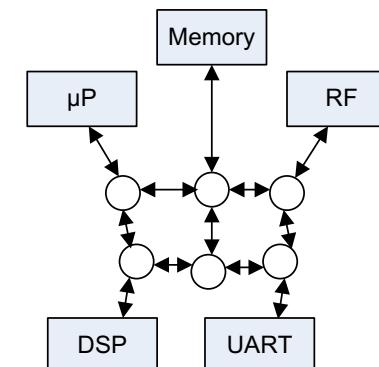
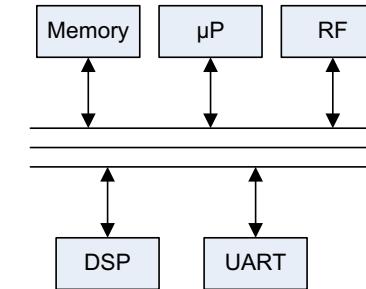
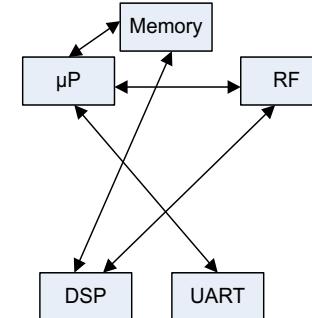
- High performance
- High area occupation
- Much more scalable



Note: compare with last one, the multiprocessor connection is no longer between memory and the processor.

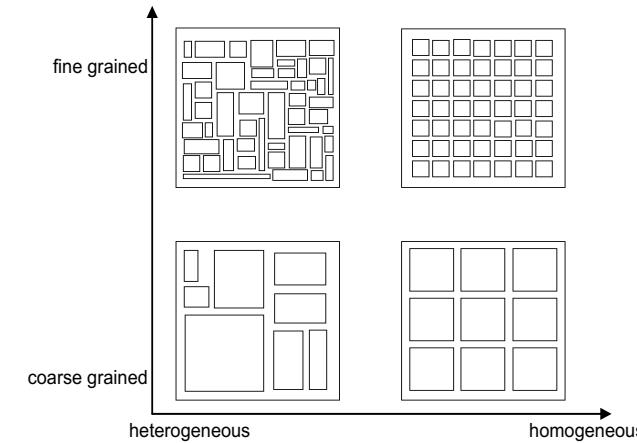
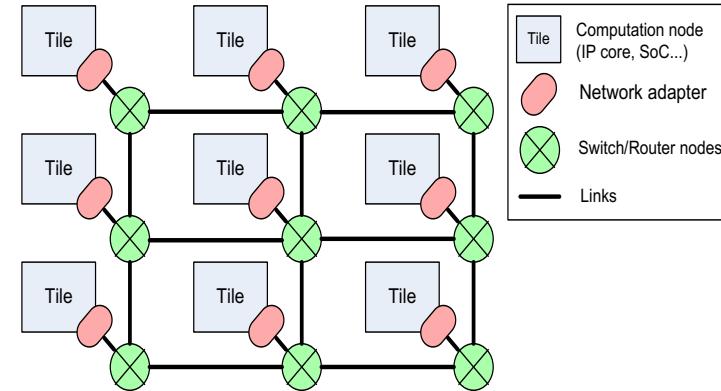
# On-chip communication architectures

- Point-to-point
  - Fixed dedicated wires
  - Not flexible, Not shared
  - Null reusability
- Bus-based interconnection (OCB)
  - Shared communication infrastructure
  - Multi-level, hierarchical or segmented buses
  - Bus becomes a bottleneck
- On-chip network (NoC)
  - Distributed nature
  - Maximum flexibility & scalability
  - Exploits reusability, parallel operations/transactions
  - Regular geometry
    - Predictable layout and performance
    - Best testability & verification time
  - Must guarantee a certain QoS



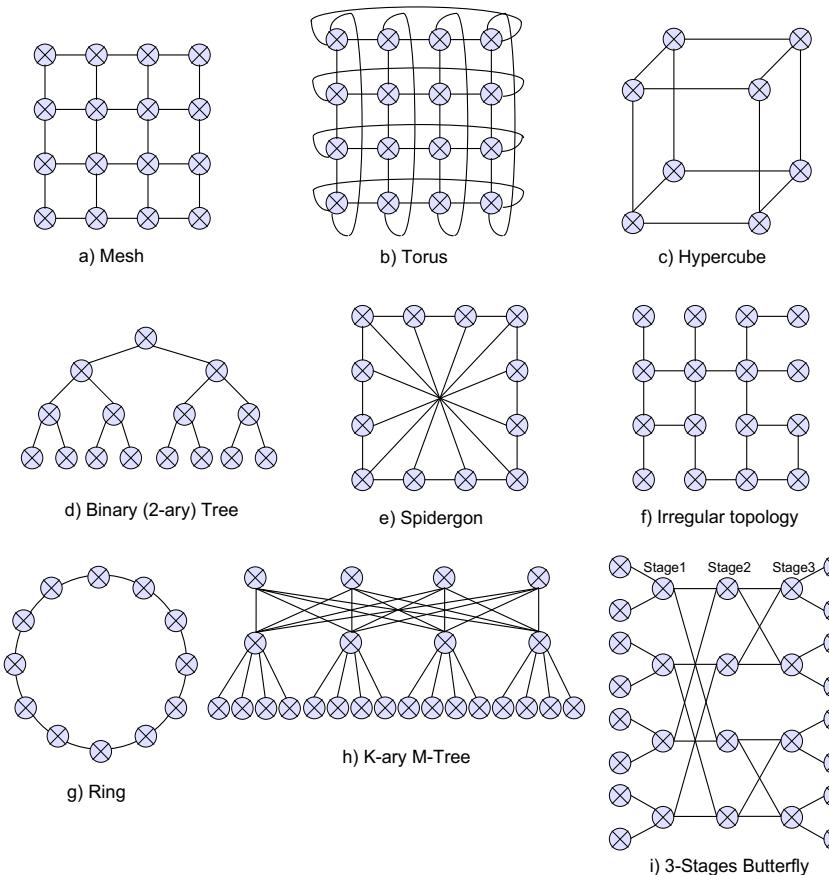
# Basic concepts on NoCs

- Tile
  - Computational nodes
- Router/Switch
  - Communication nodes
  - Switching and routing strategy
- Network adapter (NA, NI, NIC)
  - Decouple computation from communication
  - Adapts network & tile clock domains (GALS)
- Links
  - Dedicated P2P communication channels
  - Flow control protocol (Handshake or credit-based)
- NoC-based systems
  - High degree of composition and traffic diversity
  - It is desired to have good floorplanning & minimal buffer
- Conventional/Traditional networks
  - Homogeneous and coarse grained



# NoC topologies

- Typical of multiprocessor systems but now on a chip



## □ Regular

- ▶ Predictable in terms of
  - Power consumption,
  - Performance (bandwidth, latency...)
  - Area usage
- ▶ Good floorplanning

## □ Non-regular

- ▶ Mixing regular topologies
  - Mesh-Torus, Ring-Mesh, Ring-hypercube

## □ Direct

- ▶ At least one tile attached to each node

## □ Indirect

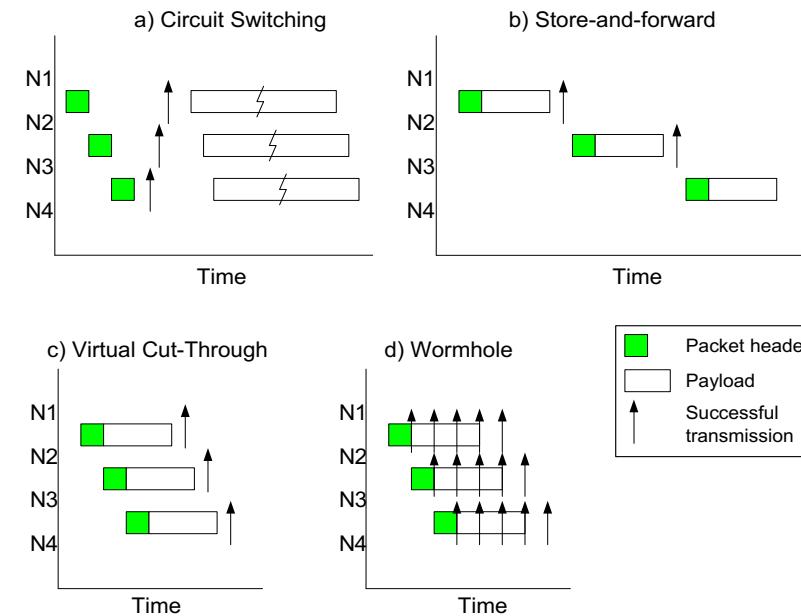
- ▶ A subset of nodes not connected to any core

## □ Its selection is a trade-off between

- ▶ Network complexity or on-chip area costs
- ▶ Communication requirements or network performance

# Switching modes & routing schemes

- Circuit switching
  - Involves the establishment & releasing of a circuit between source and destination
  - Buffer-less switching scheme
- Packet switching
  - Forwards the data to next hop
  - Buffering is mandatory
- Different packet switching modes
  - Store-and-forward
    - Stall at two nodes and the link between them
  - Wormhole
    - Combines packet switching + circuit switching
    - Reduce buffer size
    - It works at FLITS (flow control digits) level
  - Virtual cut-through
    - Next hop must store the whole packet
    - It works at packet level



## □ Buffering

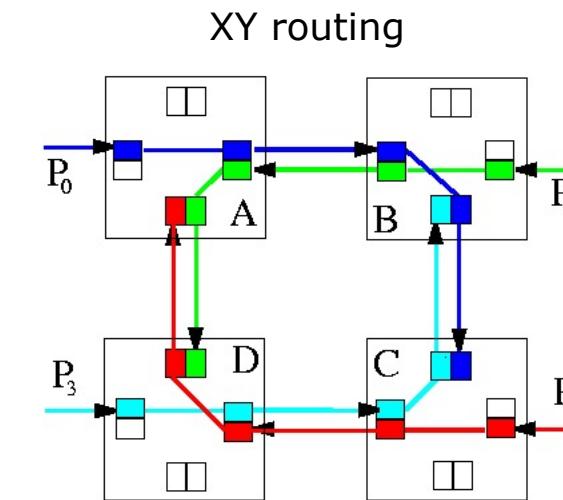
- ▶ Buffer size → width, depth
- ▶ Location in the router
- ▶ Shared or distributed
- ▶ Affects the power consumption & area usage

# Switching modes & routing schemes

- Routing schemes

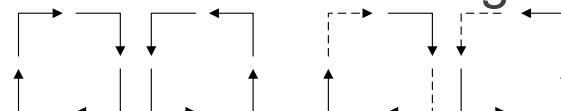
- Deterministic

- Path determined by its source & destinations address
    - Easy to implement
    - Not optimal under congestion

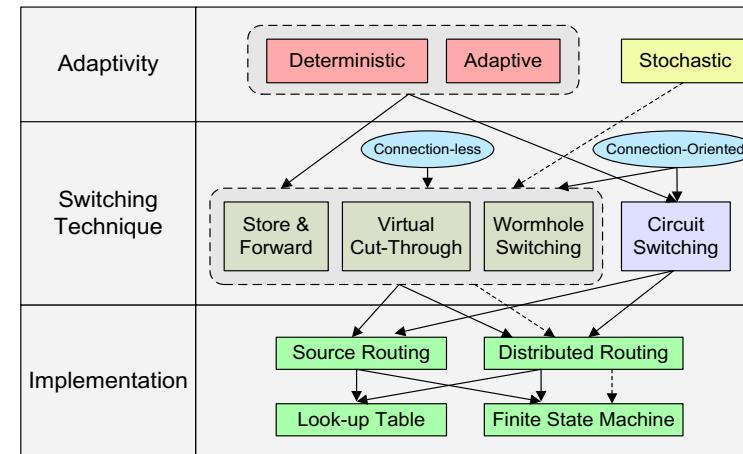


- Adaptive

- Path decided on a per-hop basis
    - Complex in its implementation
    - Must be a deadlock/livelock free routing

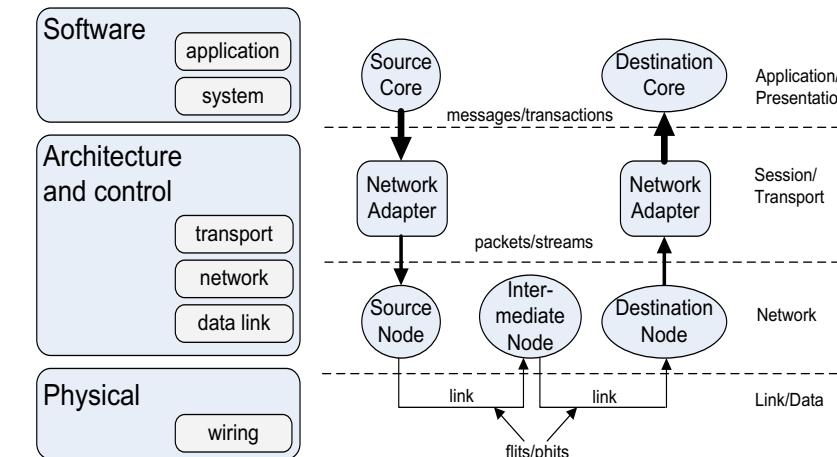


- Offers great benefits under congestion



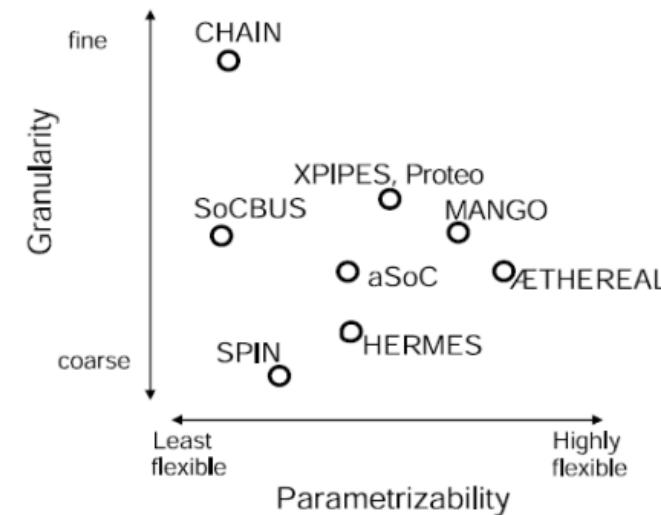
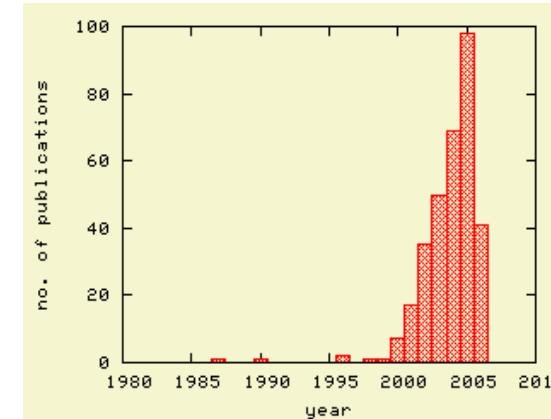
# Flow control & Micro-Network stack

- Flow control protocol (ensures the correct transport of packets)
  - Handshake
    - Request – acknowledge signals (req, ack/nAck)
    - Simpler and cheaper than credit-based
  - Credit-based
    - All network components keep counters for the available buffer space
    - Data received → counter-- | Data sent → counter++ | if counter==0 → buffer full
- Network μ-stack layers
  - Transport
    - Network Adapter has to pack/unpack messages into network layer packets
  - Network
    - Where & how a packet is transmitted
  - Data-link
    - Protocol to transmit a flit/phit
  - Physical
    - Number & length of wires



# State of the art in NoCs/MPSoCs

- NoC is an emerging & hot topic during last years
- Research at all μstack levels
  - System/Application Level
    - Design methodologies, co-exploration
    - Programming models & OS support
  - Network Adapters
  - Network architecture
  - Link level
- Research on MPSoC
  - HW-SW interfaces
  - Parallel programming models
    - Shared memory or message passing
  - ccNUMA MPSoC architecture using NIOS-II
  - MPSoC using segmented buses (HIBI)



# Communication Synthesis

- After system partitioning we got a set of tasks assigned to system components (processors executing software + hardware components); these processes are communicating through *abstract channels*.
- Tasks are also interacting with peripheral devices and other external interfaces.
- Communication synthesis has to generate hardware and software which interconnects the system components and enables processes to communicate with each other, with peripheral devices and other interfaces

# Communication Synthesis

- Communication synthesis, as a top-down design task, is performed in three main steps:
  - Channel binding,
  - Communication refinement,
  - Interface generation.
- After communication synthesis, the initial system specification results in a specification which can be directly synthesized to a physical implementation.

# Channel Binding

Abstract channels have to be implemented using physical communication components:

- resources must be allocated for supporting communication throughout the system
- abstract channels have to be partitioned and the resulted groups are bound to the allocated resources,
- messages corresponding to channels in one group are multiplexed on a shared communication component.

The main criteria used for channel grouping is to avoid bus conflicts and to reduce the total number of connecting wires:

- group together channels which don't access concurrently the bus,
- group together channels which are accessed by the same processes,
- depending on its features, a communication unit can support a certain number of channels to be multiplexed on it, without reducing the communication rate below a required minimum

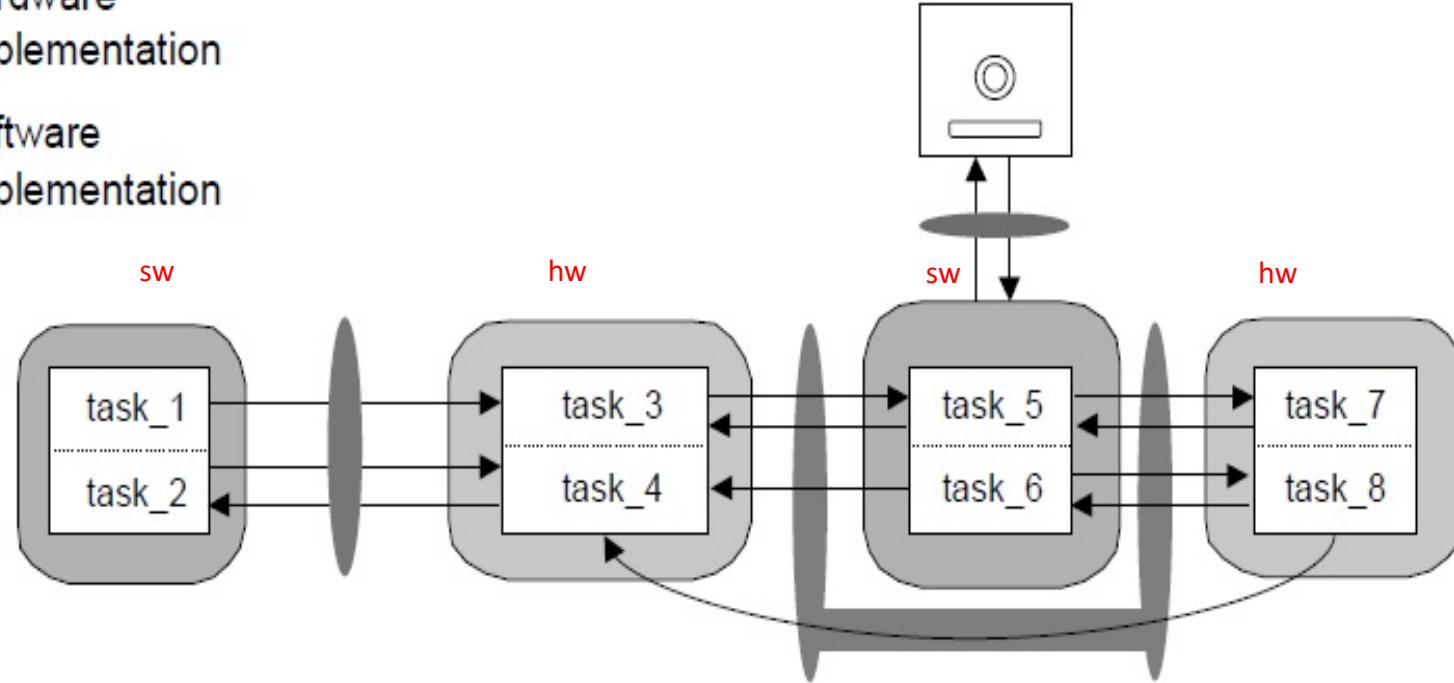
# Channel Binding



hardware  
implementation

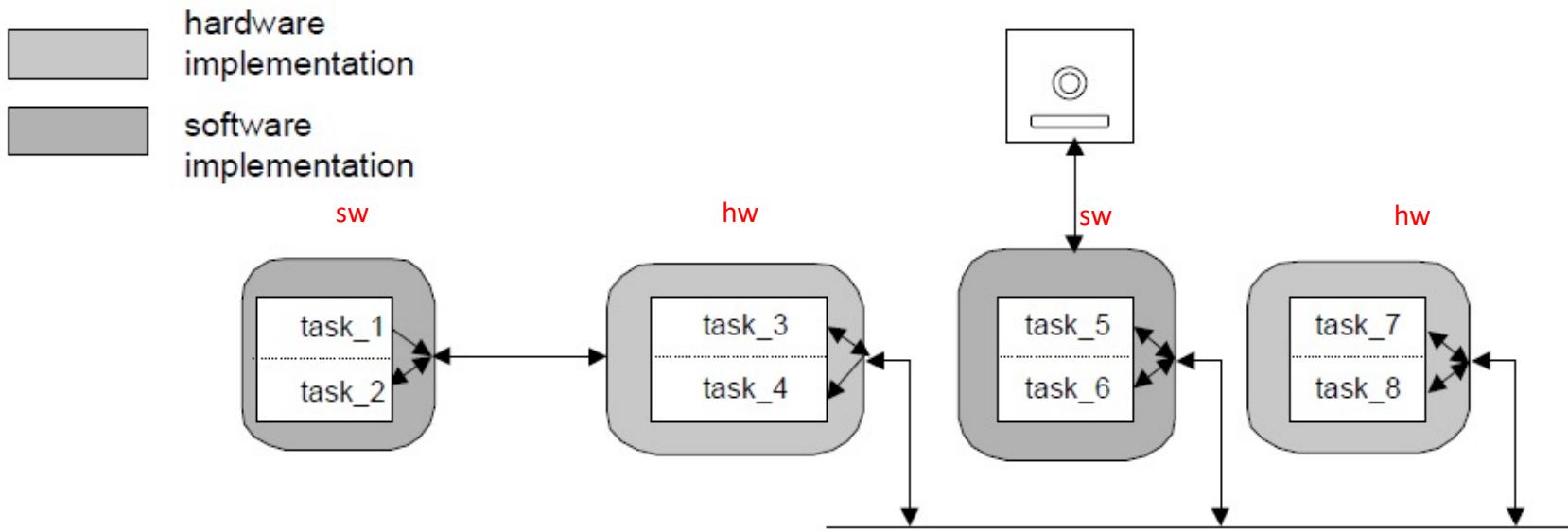


software  
implementation



channel  
binding

# Channel Binding



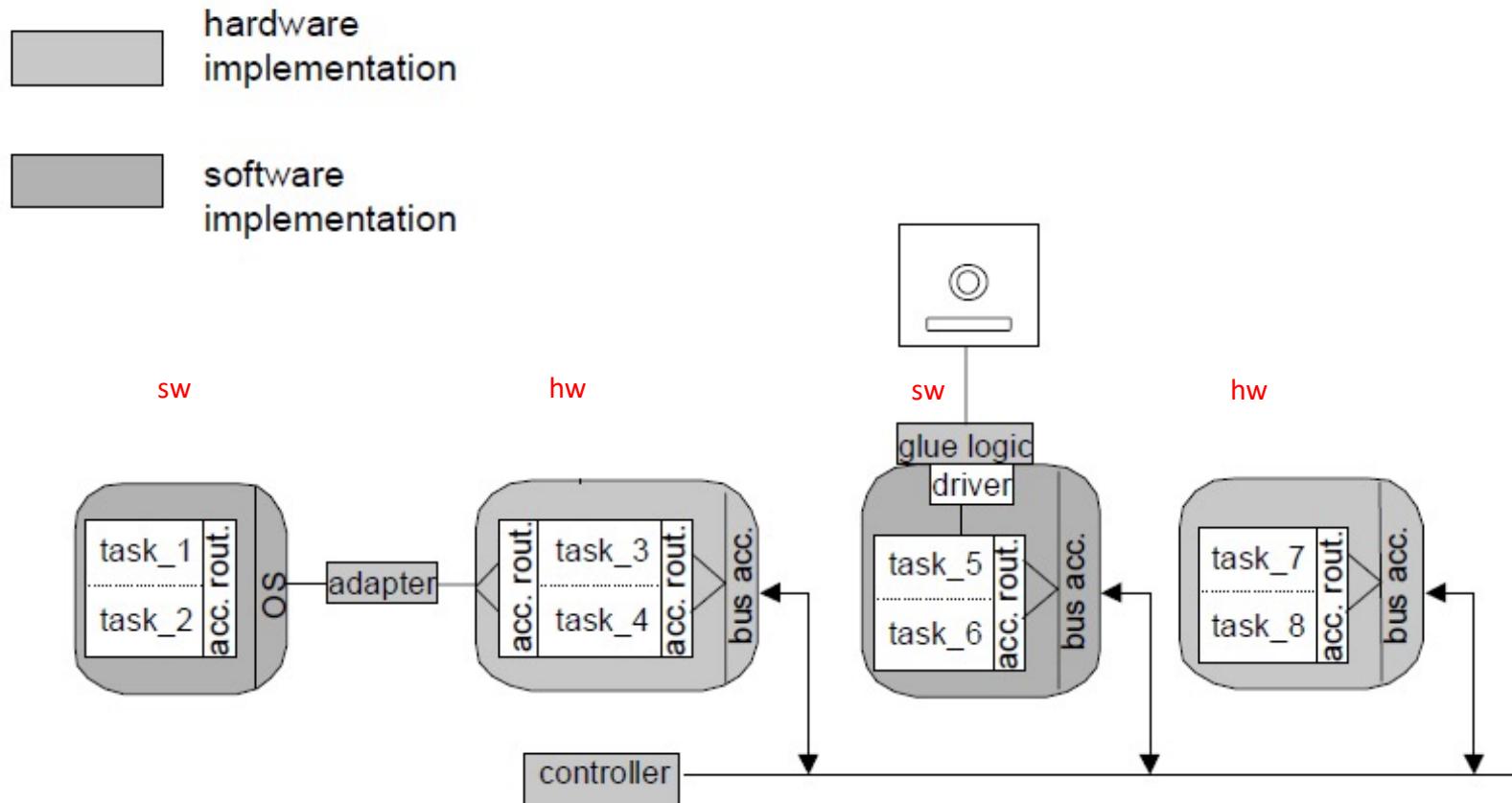
# Communication Refinement

- After *Channel binding* the interconnection topology of the system is known and it is determined which channels are bound to a given communication support.
- Communication is still quite abstract and has to be refined with several implementation details:
  - the *width of the communication lines* has to be determined, depending on constraints concerning data transfer rates, number of available pins, cost,
  - if communication buses are shared, an adequate *control strategy* has to be decided,
  - a *communication protocol* has to be defined for each communication link.

# Interface Generation

- The interfaces needed for a correct functionality of the system can be generated; both software and hardware components have to be generated:
  - *access routines* inside the communicating tasks (expanded as executable code in software or as hardware specification in hardware components),
  - *controllers* (buffers, FIFO queues, arbitration logic) implement correct access to communication support,
  - *adapters* needed to interface components which use incompatible protocols,
  - *device drivers* to support access to peripheral devices and application specific interfaces,
  - *low level support for communication related tasks* (interrupt control, DMA, memory mapped I/O).

# Interface Generation



# Bus and Protocols

- A bus consists of wires connecting two or more processors or memories.
  - Each wire in a bus may be *uni-directional* (e.g. rd/wr, enable or addr), or *bi-directional* (e.g., data).
  - A bus has an associated *protocol* describing the rules for transferring data over the wires.
- Bus protocols usually described using *timing diagrams*.
- For a purpose of automatic synthesis other methods are also proposed:
  - HDL descriptions,
  - grammar based descriptions.

# Timing Diagrams

- Most common hardware protocol

- Bus cycle

- Possible subprotocol
    - e.g., read protocol or write protocol
  - Complete transfer
  - May be several clock cycles

- Active high vs. active low

- “Assert” means active

- Read protocol example:



POLITECNICO  
MILANO 1863

•  $rd/wr$  set low

# Protocol Control Methods

- *Strobe* protocol — the master uses one control line, often called the *request* line, to initiate the data transfer, and the transfer is considered to be complete after some fixed time interval after the initiation.
- *Handshake* protocol — master uses a *request* line to initiate the transfer, and the slave uses an *acknowledge* line to inform the master when the data is ready.

# I/O Addressing

A microprocessor communicates with other devices using some of its pins

- Port-based I/O (parallel I/O)
  - Processor has one or more N-bit ports
  - Processor's software reads and writes a port just like a register
  - Bus-based I/O
  - Processor has address, data and control ports that form a single bus
  - Communication protocol is built into the processor
  - A single instruction carries out the read or write protocol on the bus
- Parallel I/O peripheral
  - When processor only supports bus-based I/O but parallel I/O needed
  - Each port on peripheral connected to a register within peripheral that is read/written by the processor
- Extended parallel I/O
  - When processor supports port-based I/O but more ports needed
  - One or more processor ports interface with parallel I/O peripherals extending number of ports

# Interrupts

- The communication with a device which produces data asynchronously requires a new communication paradigm.
- *Polling* — repeatedly check whether data is available at the port (waste of time).
- *Interrupts* — the microprocessor checks for the presence of a particular condition (similar to events in some languages).
- If interrupt, then an *Interrupt Service Routine (ISR)* is called automatically.
- *Fixed address for ISR*.
- *Vectored interrupt* make it possible to determine the address at which the ISR resides
  - peripheral asserts *IntReq*,
  - processor acknowledges *IntAck* that the interrupt has been detected,
  - the peripheral provides the address on the data bus, and the microprocessor jumps to the ISR.
- *Interrupt address table* — the peripheral provides *int* and the interrupt number and the processor decodes it using the table.

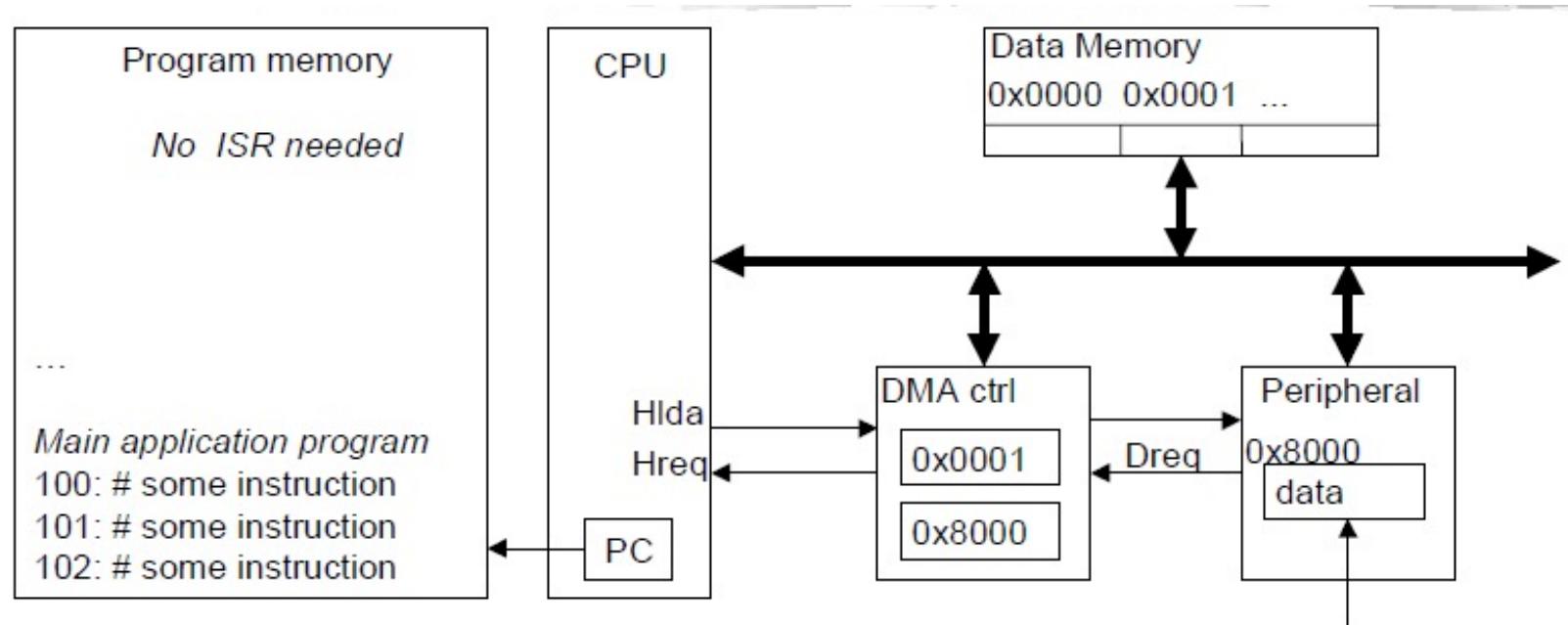
# Interrupt Issues

- Maskable vs. non-maskable interrupts
  - Maskable: programmer can set bit that causes processor to ignore interrupt
    - Important when in the middle of time-critical code
  - Non-maskable: a separate interrupt pin that can't be masked
    - Typically reserved for drastic situations, like power failure requiring immediate backup of data to non-volatile memory
- Jump to ISR
  - Some microprocessors treat jump same as call of any subroutine
    - Complete state saved (PC, registers)
  - Others only save partial state, like PC only
    - Thus, ISR must not modify registers, or else must save them first
    - Assembly-language programmer must be aware of which registers stored

# Direct Memory Access

- DMA controller — single-purpose processor which transfers data between memories and peripherals “outside” the processor.
  - Steals memory access cycles from processor while needed.
  - Does not require interventions from processor and overhead of interrupted program.
  - DMA usually transfers block of data.

# Direct Memory Access

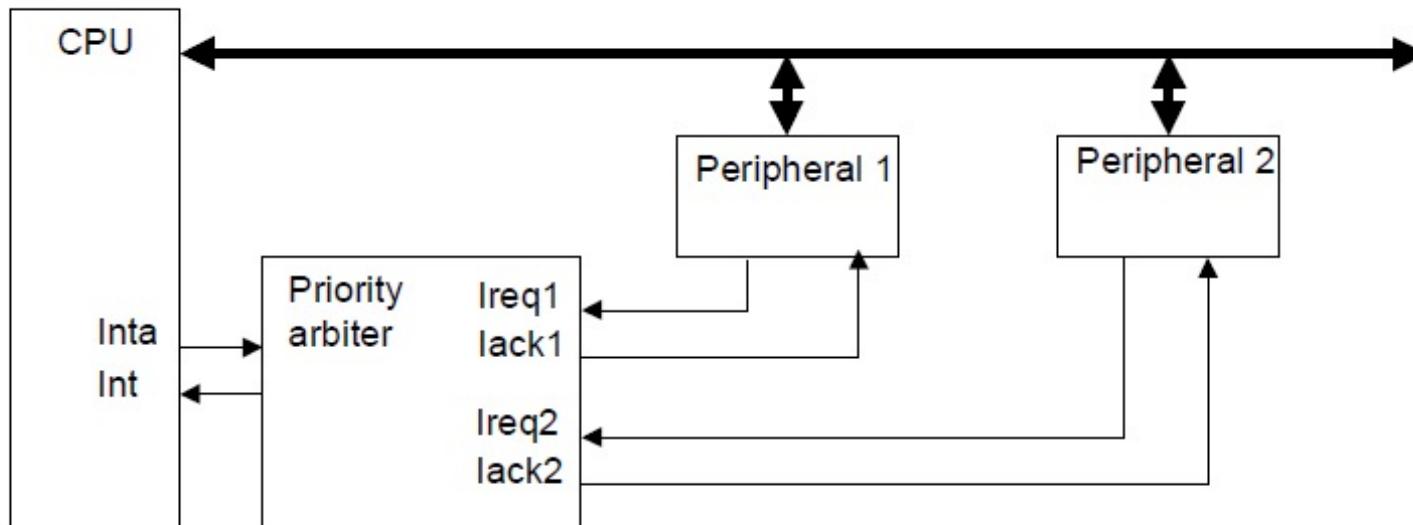


1. Microprocessor is executing its program, say current program counter (PC)=100.
2. Peripheral's input device stores new data in register at 0x8000.
3. Peripheral asserts *Dreq* to request servicing of the new data.
4. DMA controller asserts *Hreq* to request control of the system bus.
5. Microprocessor relinquishes system bus, perhaps stopping after executing statement 100.
6. **DMA controller reads data from 0x8000 and writes that data to 0x0001 in data memory.**
7. DMA controller deasserts *Hreq* and completes handshake with peripheral.
8. Microprocessor resumes executing its program, perhaps starting with statement 101.

# Bus Arbitration

- Multiple peripherals might request service from a single resource, for example a bus.
  - Priority arbiter — decides to whom the resource is granted.
  - Arbitration method:
    - *fixed priorities* between peripherals,
    - *rotating priorities* (round-robin) — the arbiter changes priority of peripherals based on the history of servicing of those peripherals.

# Bus Arbitration



1. Microprocessor is executing its program.
2. Peripheral1 needs servicing so asserts *Ireq1*. Peripheral2 also needs servicing so asserts *Ireq2*.
3. Priority arbiter sees at least one *Ireq* input asserted, so asserts *Int*.
4. Microprocessor stops executing its program and stores its state.
5. Microprocessor asserts *Inta*.
6. Priority arbiter asserts *lack1* to acknowledge Peripheral1.
7. Peripheral1 puts its interrupt address vector on the system bus
8. Microprocessor jumps to the address of ISR read from data bus, ISR executes and returns (and completes handshake with arbiter).
9. Microprocessor resumes executing its program.

# Protocol Examples

- I2C (Inter-IC) - two-wire serial bus protocol developed by Philips Semiconductors nearly 20 years ago
  - Data transfer rates up to 100 kbits/s and 7-bit addressing possible in normal mode
  - 3.4 Mbits/s and 10-bit addressing in fast-mode
- CAN (Controller area network)
  - Protocol for real-time applications
  - Developed by Robert Bosch GmbH
  - Originally for communication among components of cars
  - Data transfer rates up to 1 Mbit/s and 11-bit addressing
- FireWire (a.k.a. I-Link, Lynx, IEEE 1394)
  - High-performance serial bus developed by Apple Computer Inc.
  - Designed for interfacing independent electronic components (e.g., desktop, scanner, cameras)
  - Data transfer rates from 12.5 to 400 Mbits/s, 64-bit addressing

# Protocol Examples

- USB (Universal Serial Bus)
  - Easier connection between PC and monitors, printers, digital speakers, modems, scanners, digital cameras, joysticks, multimedia game equipment
  - 2 data rates:
    - 12 Mbps for increased bandwidth devices
    - 1.5 Mbps for lower-speed devices (joysticks, game pads)
- PCI Bus (Peripheral Component Interconnect) - high performance bus originated at Intel in the early 1990's
  - Data transfer rates of 127.2 to 508.6 Mbits/s and 32-bit addressing
- ARM Bus
  - Designed and used internally by ARM Corporation
  - Interfaces with ARM line of processors
  - Data transfer rate is a function of clock speed
  - If clock speed of bus is X, transfer rate =  $16 \times X$  bits/s

# Protocol Examples

- IrDA
  - Protocol suite that supports short-range point-to-point infrared data transmission
  - Created and promoted by the Infrared Data Association (IrDA)
  - Data transfer rate of 9.6 kbps and 4 Mbps
- Bluetooth
  - New, global standard for wireless connectivity
  - Based on low-cost, short-range radio link
- IEEE 802.11
  - Proposed standard for wireless LANs
  - provisions for data transfer rates of 1 or 2 Mbps

# Summary

- Interface design is a challenging design step which requires a deep knowledge on underlying hardware communication devices.
  - On the top level we distinguish
  - Channel binding,
  - Communication refinement,
  - Interface generation.
- The deep knowledge hardware properties, such as communication protocols, interrupts, DMA, arbiters is, however, needed to make these steps efficient.

# **Design of Hardware Accelerators**

Academic Year 2021/2022

# Questions?

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)



**POLITECNICO**  
MILANO 1863

**Design of Hardware Accelerators**  
Academic Year 2021/2022

# Hardware Security and Hardware-Assisted Security

**Christian Pilato**

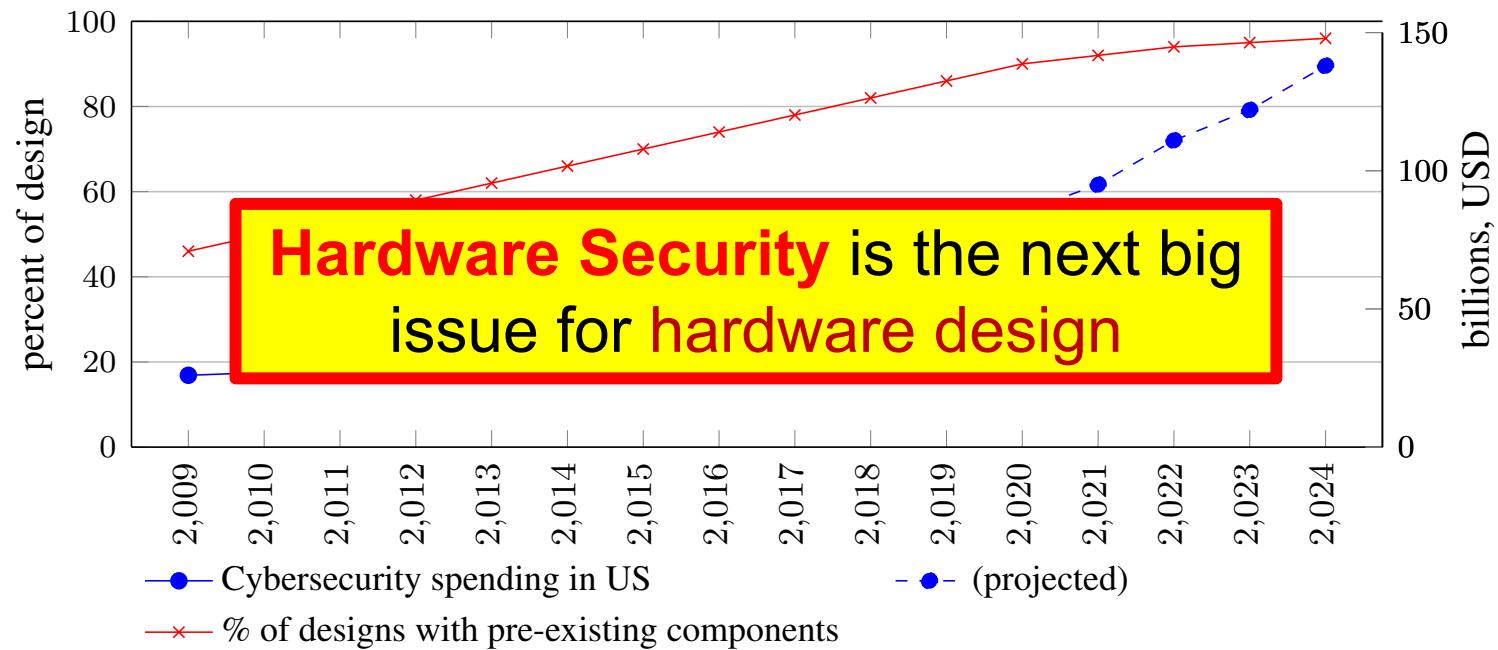
Dipartimento di Elettronica, Informazione e Bioingegneria

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)

# System Complexity and Hardware Security

Increasing system complexity demands **design & reuse approaches**

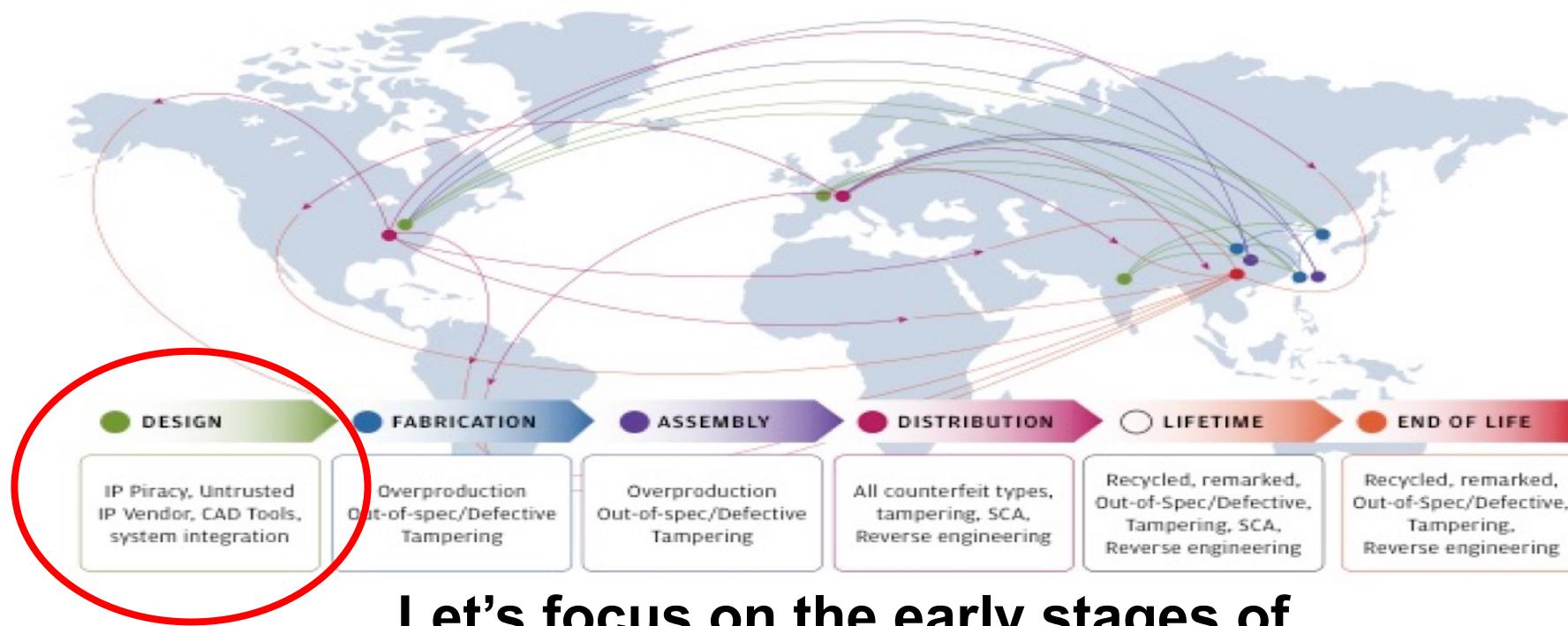
- IP components are **coming from many vendors** and assembled to create the SoC
- Most of the design houses are fabless



# Globalization of the Supply Chain

Supply chain is more and more distributed to reduce costs

- Many security threats
- Cost of addressing them is exponentially increasing from level to level



Let's focus on the early stages of  
the design process...

# Hardware Threats

## Reverse Engineering and IP Theft

Methods to **extract chip functionality** from circuit designs in order to create illegal copies

- Steal the technology
- Cut design costs
- Enter into a market
- ...

## Hardware Trojans

**Malicious modifications** of an existing chip design to introduce an additional functionality

- Steal data (e.g., through side channels)
- Harm the normal operations of the chips (e.g., DoS attacks)
- Alter the chip functionality (e.g., errors)
- ...

## Data Injection

Injection of **spurious data** to exploit software or hardware/software vulnerabilities

- Buffer overflow attacks
- Memory corruption
- ...

## Side Channel

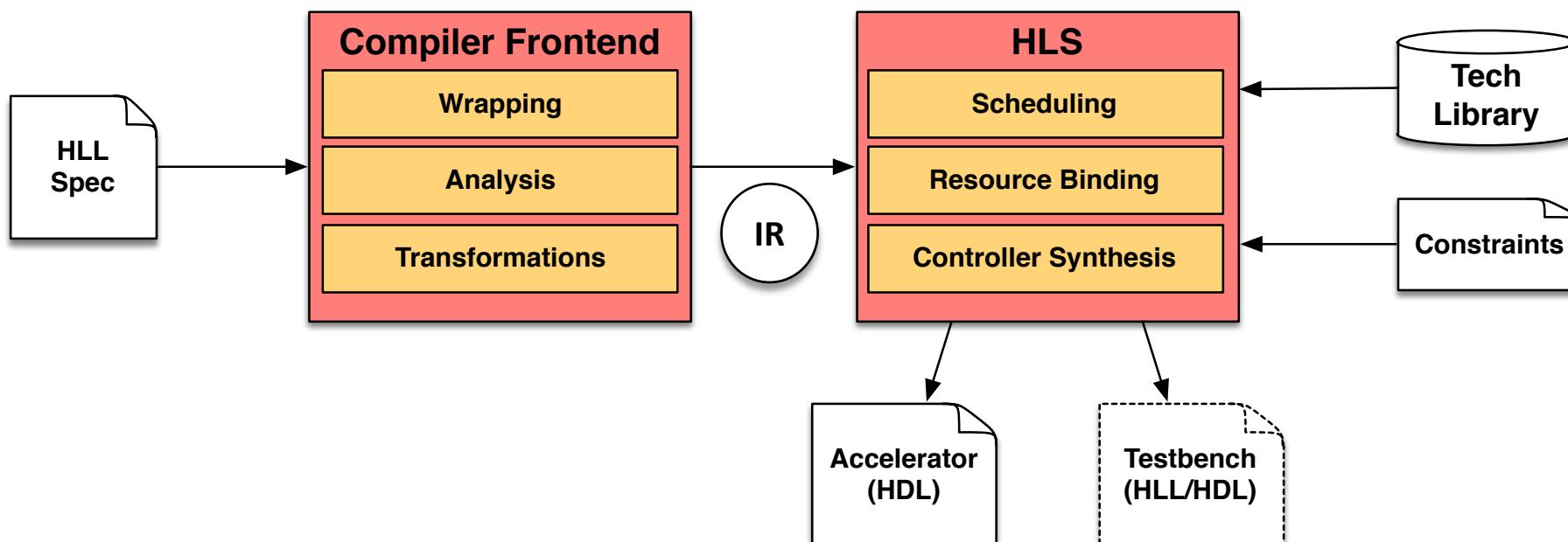
Methods to create additional communication channels to **steal sensitive data**

- Differential power analysis for key extraction
- Timing channels for reverse engineering
- ...

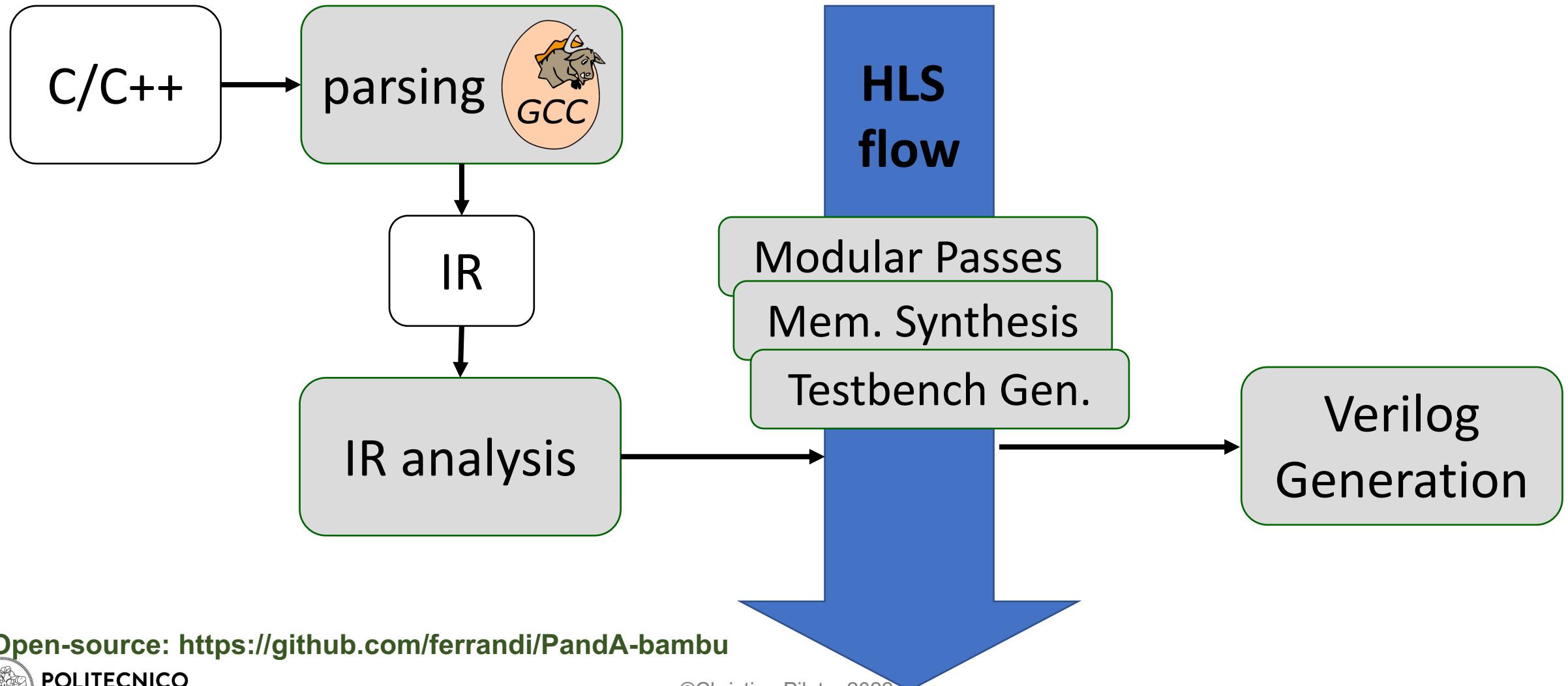
# High-Level Synthesis

**High-Level Synthesis (HLS)** is used to **automatically generate RTL designs** starting from a high-level specification

- It leverages **state-of-the-art compilers** (e.g., GCC or LLVM)
- It implements several **hardware-oriented and technology-aware optimizations**



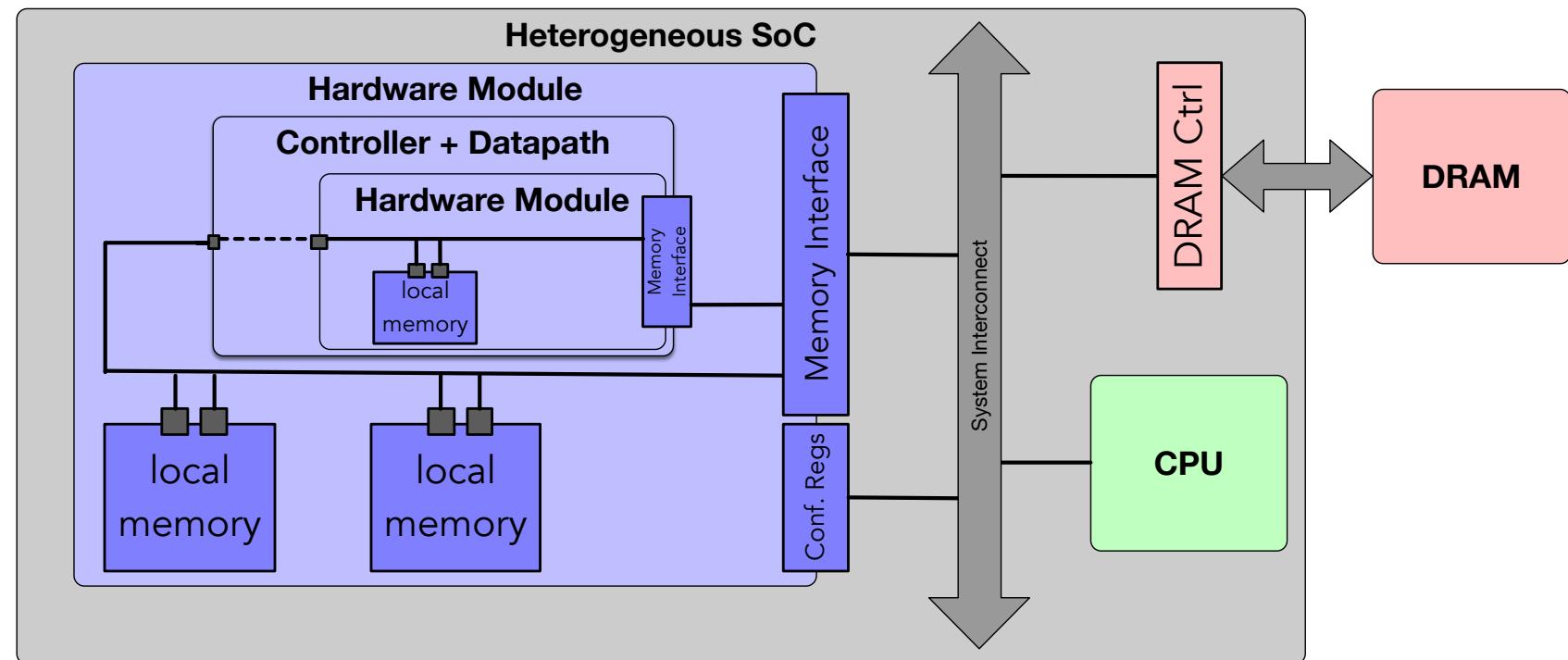
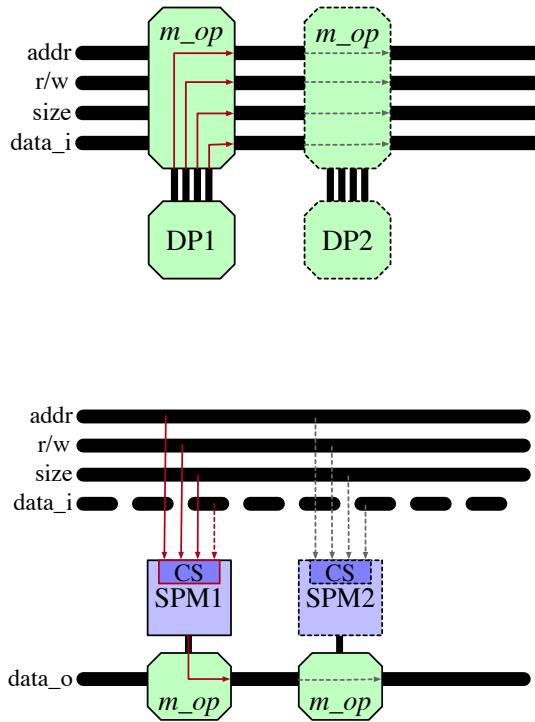
# Bambu HLS Framework



# Bambu: HLS Microarchitecture

Hierarchical synthesis of the C functions (based on **call graph**)

- Internal “memory bus” to dynamically resolve the mem address



# How is Using HLS for Hardware Security?

**Good:** automatic generation of protection mechanisms

- Fine-grained Dynamic Information Flow Tracking
- Algorithm-Level Obfuscation
- IP Watermarking

**Bad:** potential attack vector

- Planned Obsolescence
- Key Recovery with Reduced-Round Attacks

**Ugly:** Dream vs. Reality

- What is Missing?

# How is Using HLS for Hardware Security?

**Good:** automatic generation of protection mechanisms

- Fine-grained Dynamic Information Flow Tracking
- Algorithm-Level Obfuscation
- IP Watermarking

**Bad:** potential attack vector

- Planned Obsolescence
- Key Recovery with Reduced-Round Attacks

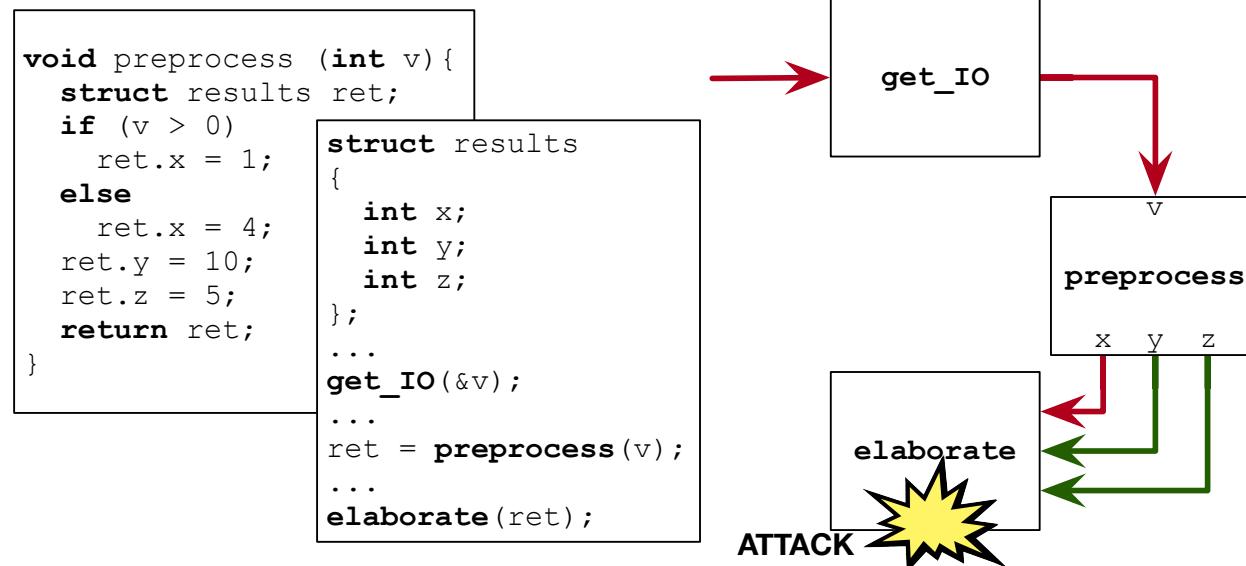
**Ugly:** Dream vs. Reality

- What is Missing?

# Dynamic Information Flow Tracking

Marking Data coming from untrusted sources with tags (**taints**)

- Trap to OS if tainted data are used in critical operations
  - Pointer dereference, jump address, modified code or data, ...



**APPLE KERNEL CODE  
VULNERABILITY AFFECTED ALL  
DEVICES**

by: Jonathan Bennett



80 Comments

November 1, 2018

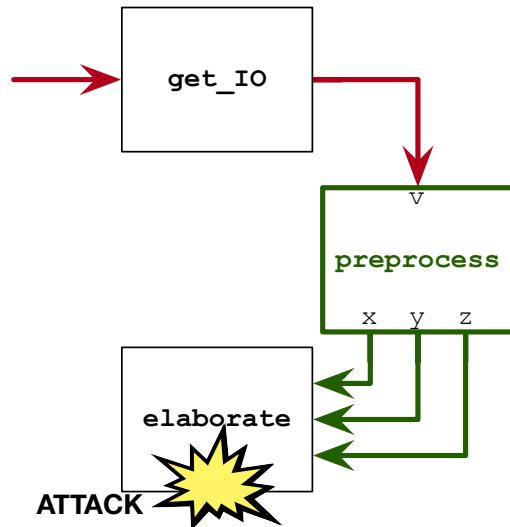
DIFT can protect from several  
**software-based attacks**

# DIFT in Heterogeneous Architectures

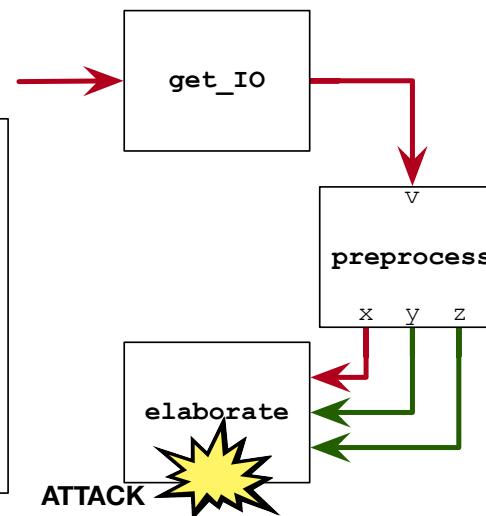
Applications interleaves tasks in both hardware and software

- What happens when **accelerators** are executed *before* the potential attack point?

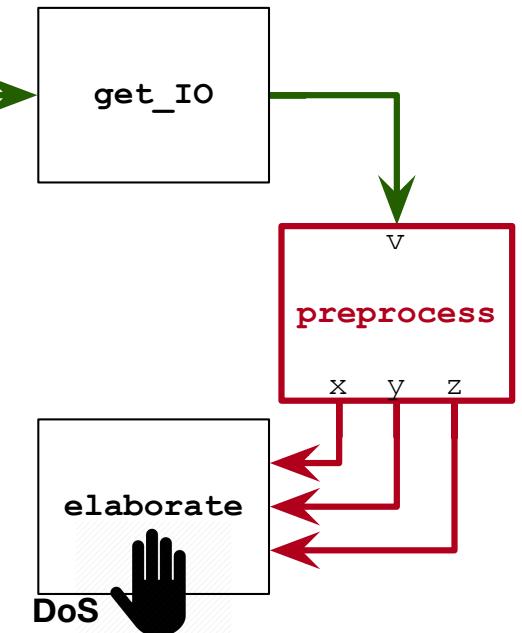
**Optimistic**



```
void preprocess (int v){  
    struct results ret;  
    if (v > 0)  
        ret.x = 1;  
    else  
        ret.x = 4;  
    ret.y = 10;  
    ret.z = 5;  
    return ret;  
}  
  
struct results  
{  
    int x;  
    int y;  
    int z;  
};  
...  
get_IO(&v);  
...  
ret = preprocess(v);  
...  
elaborate(ret);
```



**Pessimistic**

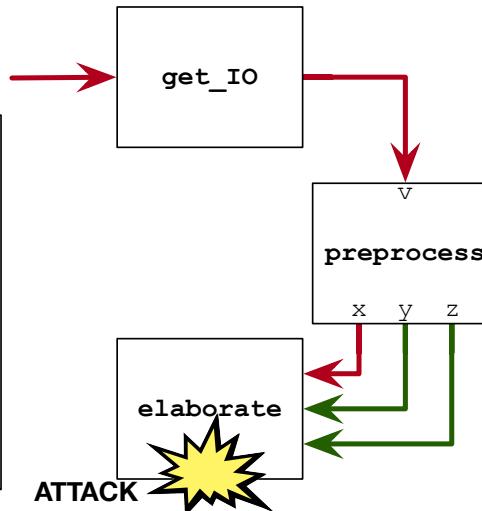


# DIFT in Heterogeneous Architectures

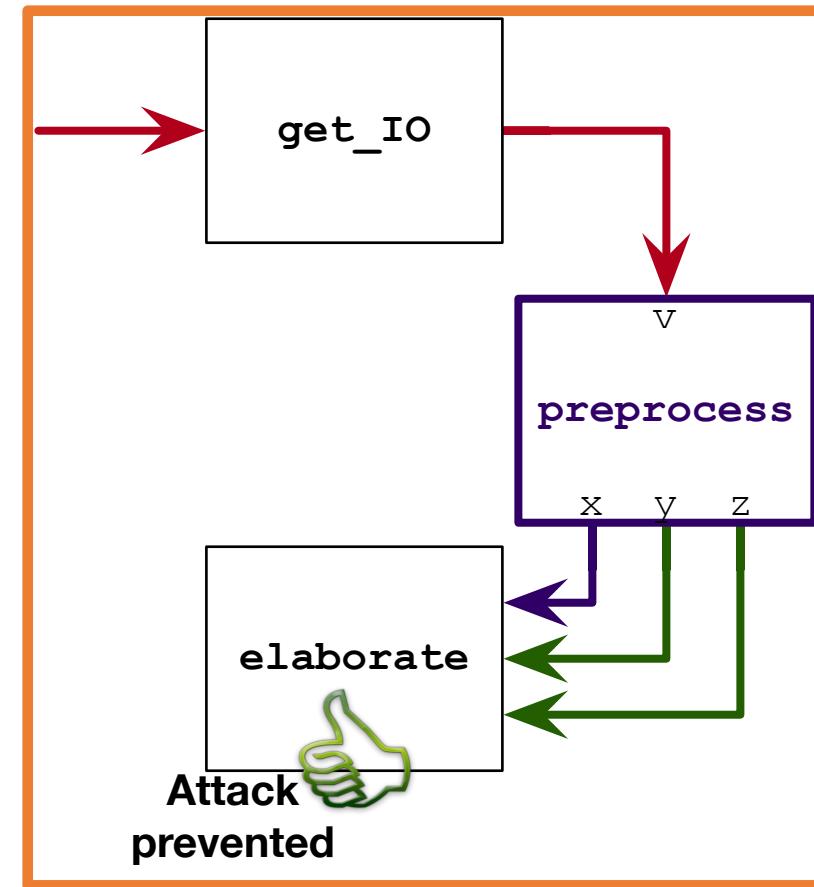
Applications interleaves tasks in both hardware and software

- What happens when **accelerators** are executed *before* the potential attack point?

```
void preprocess (int v){  
    struct results ret;  
    if (v > 0)  
        ret.x = 1;  
    else  
        ret.x = 4;  
    ret.y = 10;  
    ret.z = 5;  
    return ret;  
}  
  
...  
  
get_IO(&v);  
...  
ret = preprocess(v);  
...  
elaborate(ret);
```



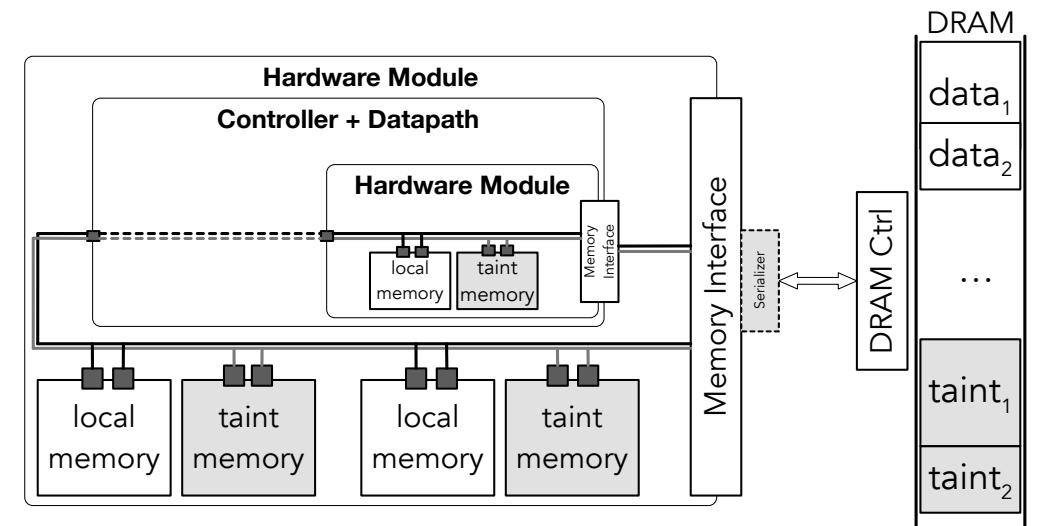
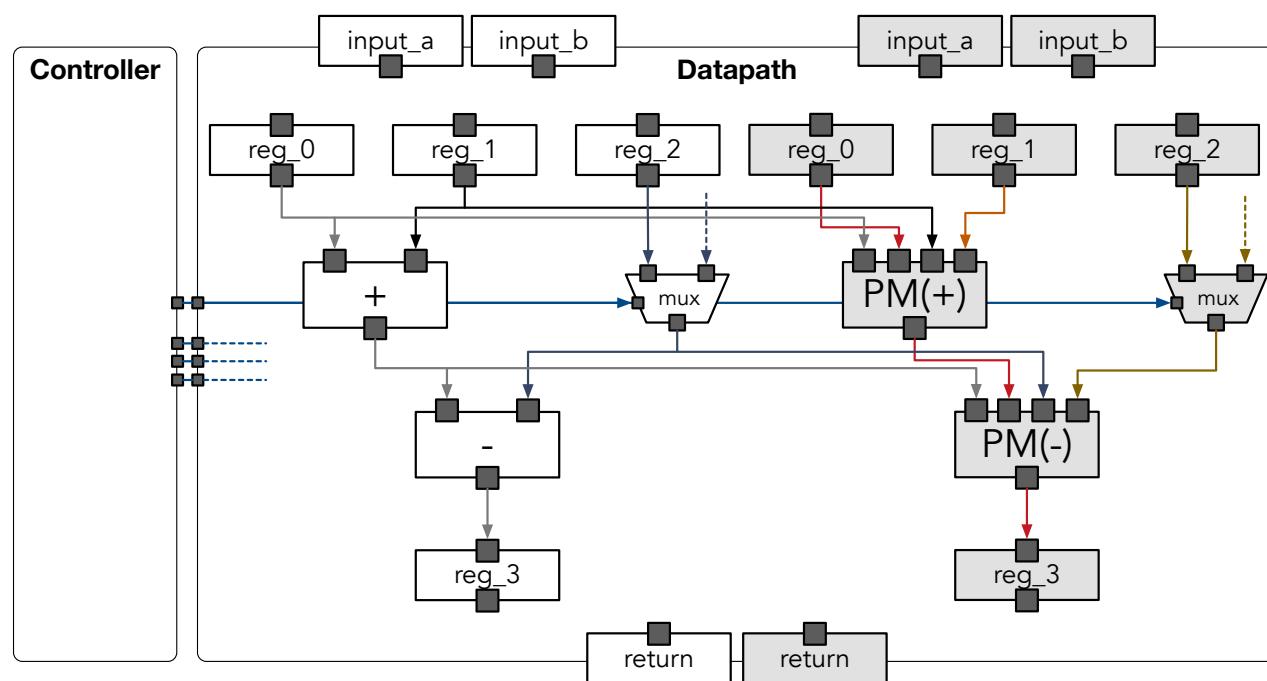
Accelerators require  
fine-grained support for DIFT



# TaintHLS: DIFT Support within HLS

Data path extended with **shadow logic** and memory architecture with **taint memories**

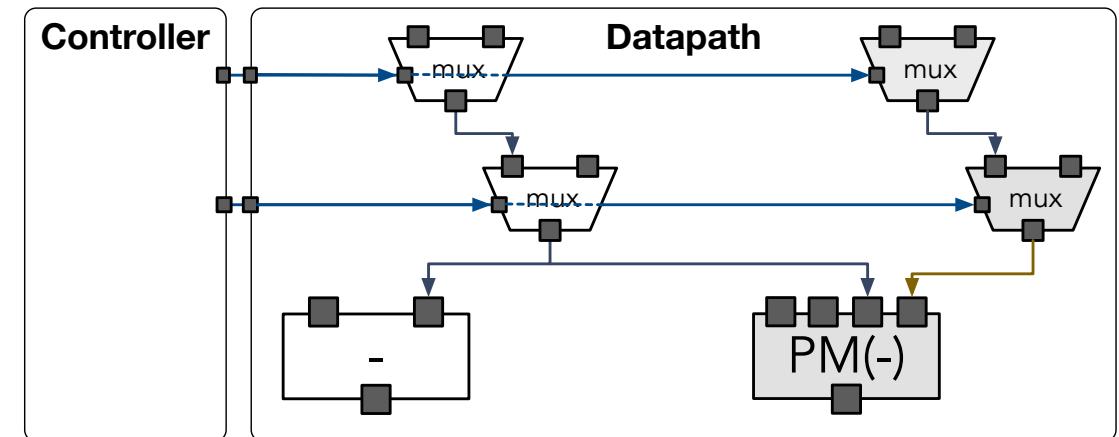
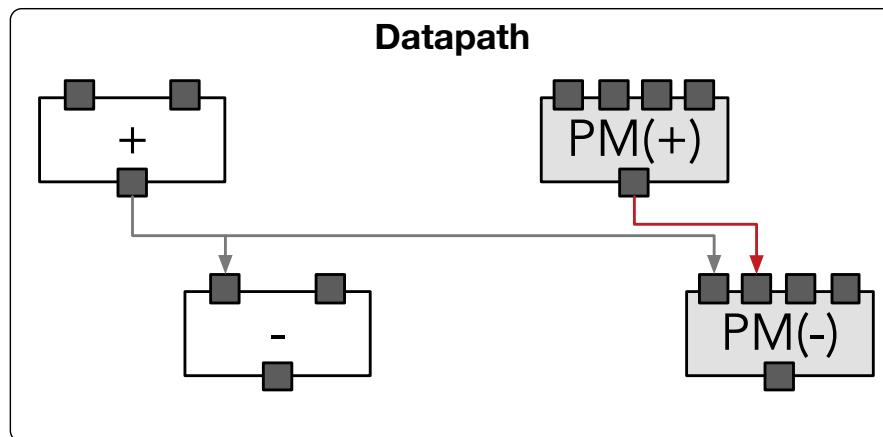
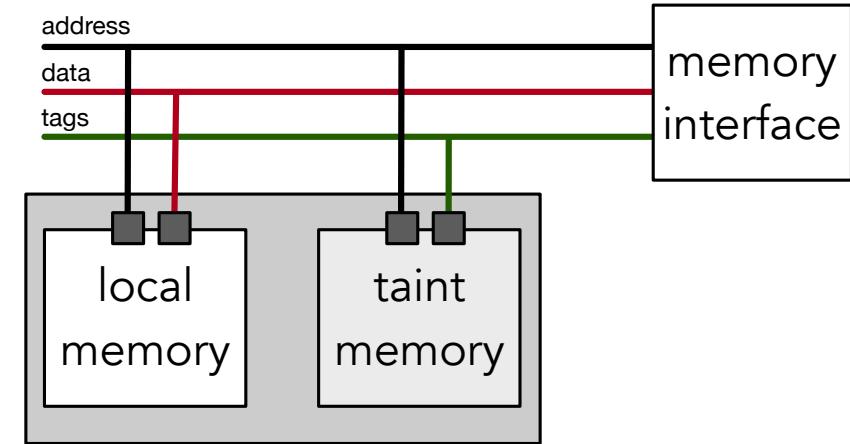
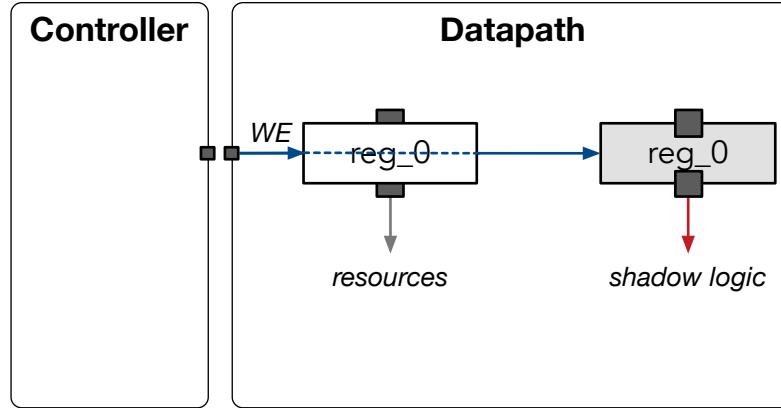
- HLS-based methodology for automatic generation based on HLS results



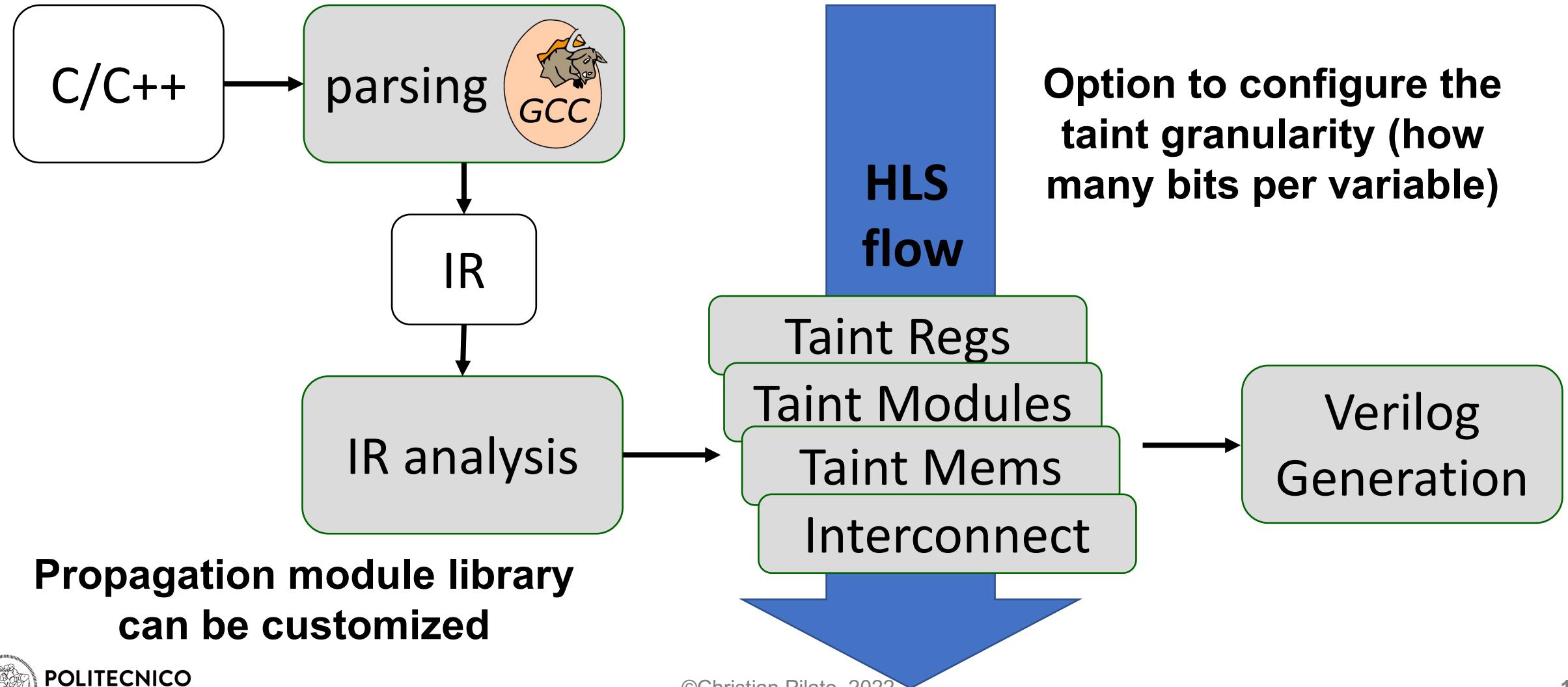
Almost **no performance overhead** with optimizations to limit area overhead

# Data Flow Consistency

Microarchitectural solutions to propagate data and tags in parallel



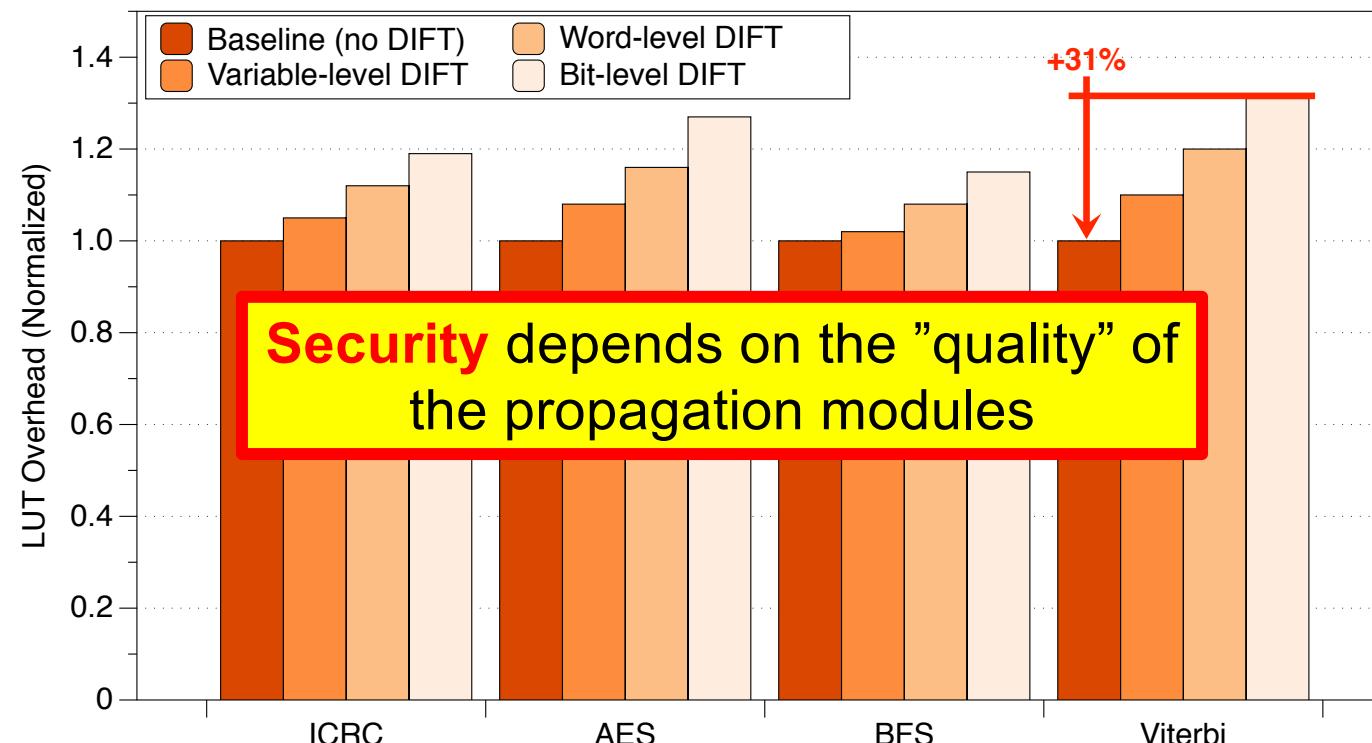
# Bambu for DIFT-enabled IPs



# Area overhead

Area overhead of each granularity wrt the **baseline** version

- Xilinx Virtex-7 FPGA @ 100 MHz

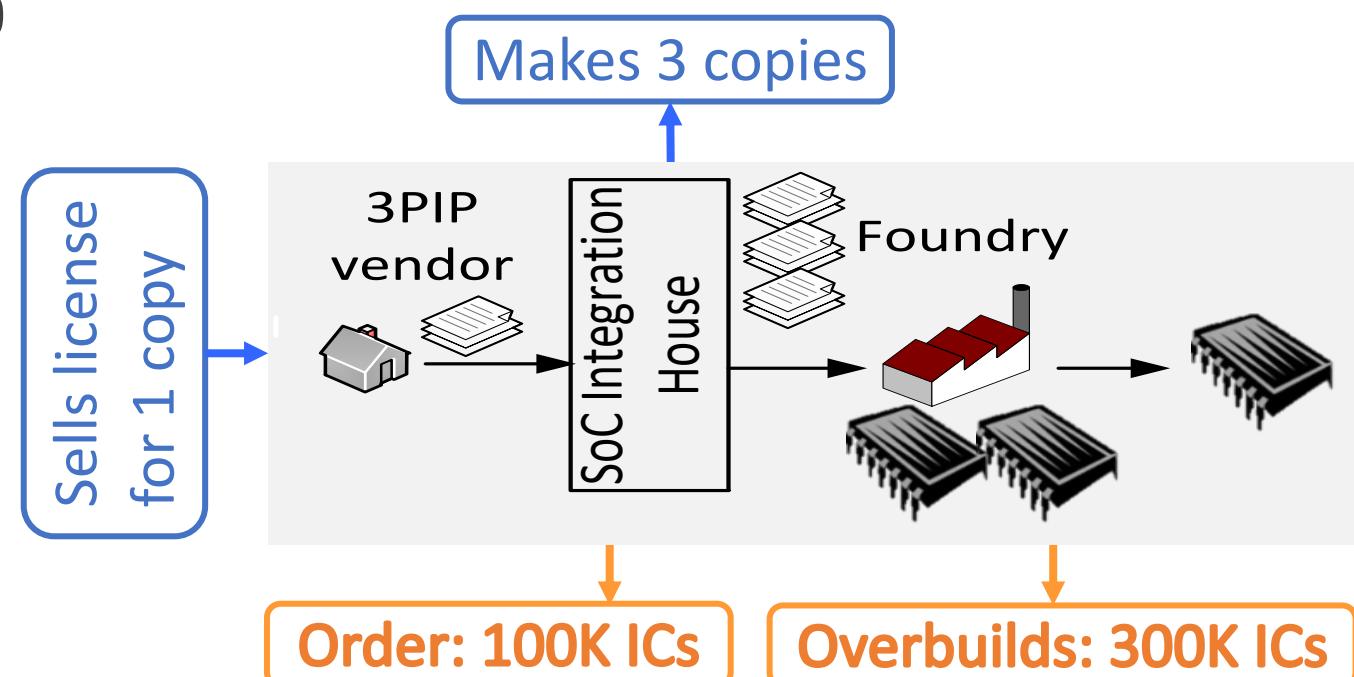
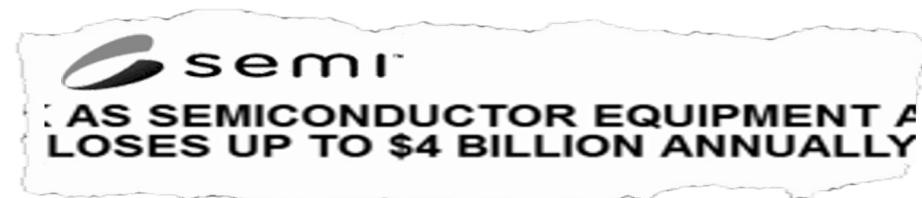


# IC/IP piracy and overbuilding

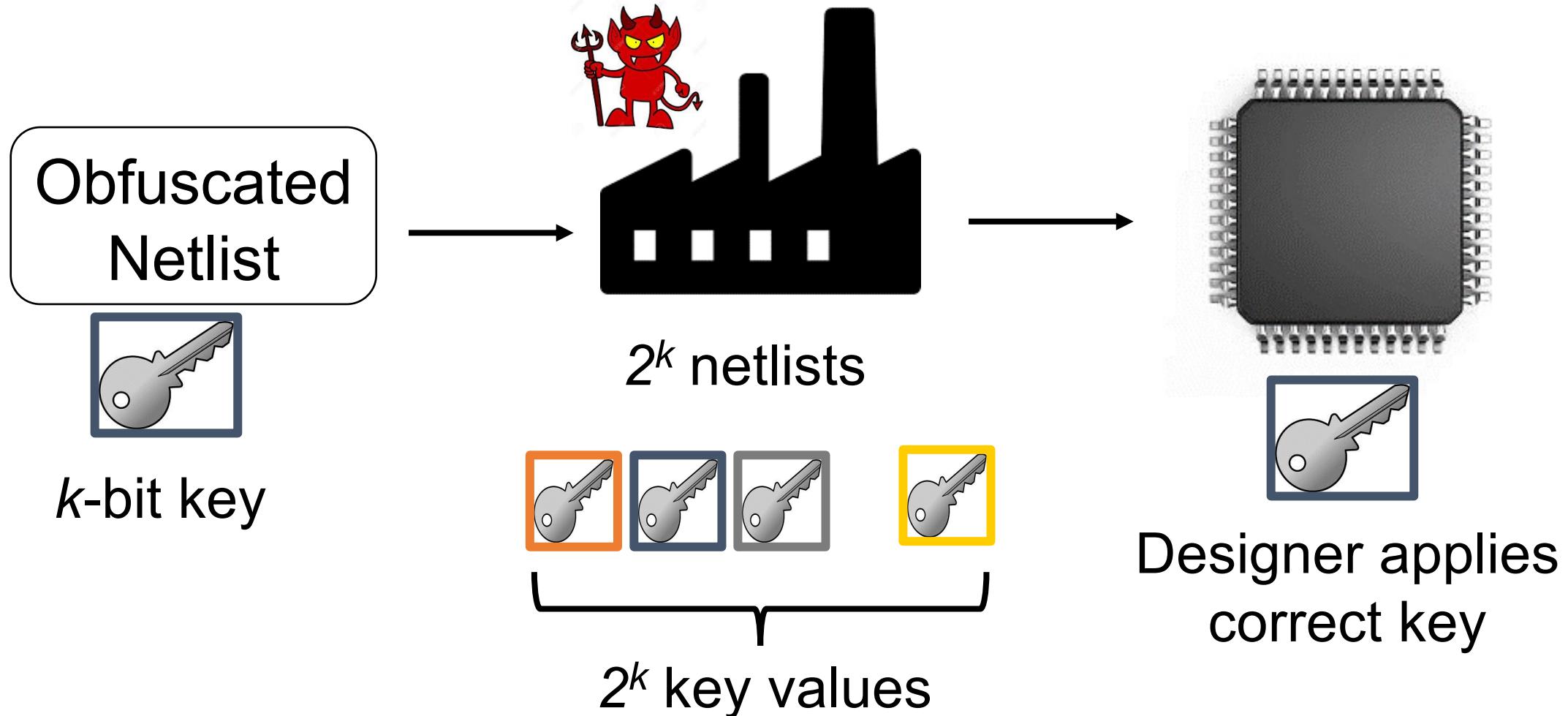
- Steal and claim ownership of IC and/or illegal use
  - Malicious SoC integration house
  - Malicious foundry
- Real-life impact
  - \$4,000,000,000 loss per year to IC industry
  - ARM detected IP piracy in 2000

**EE Times**

ARM files patent infringement suit  
against IP startup picoTurbo

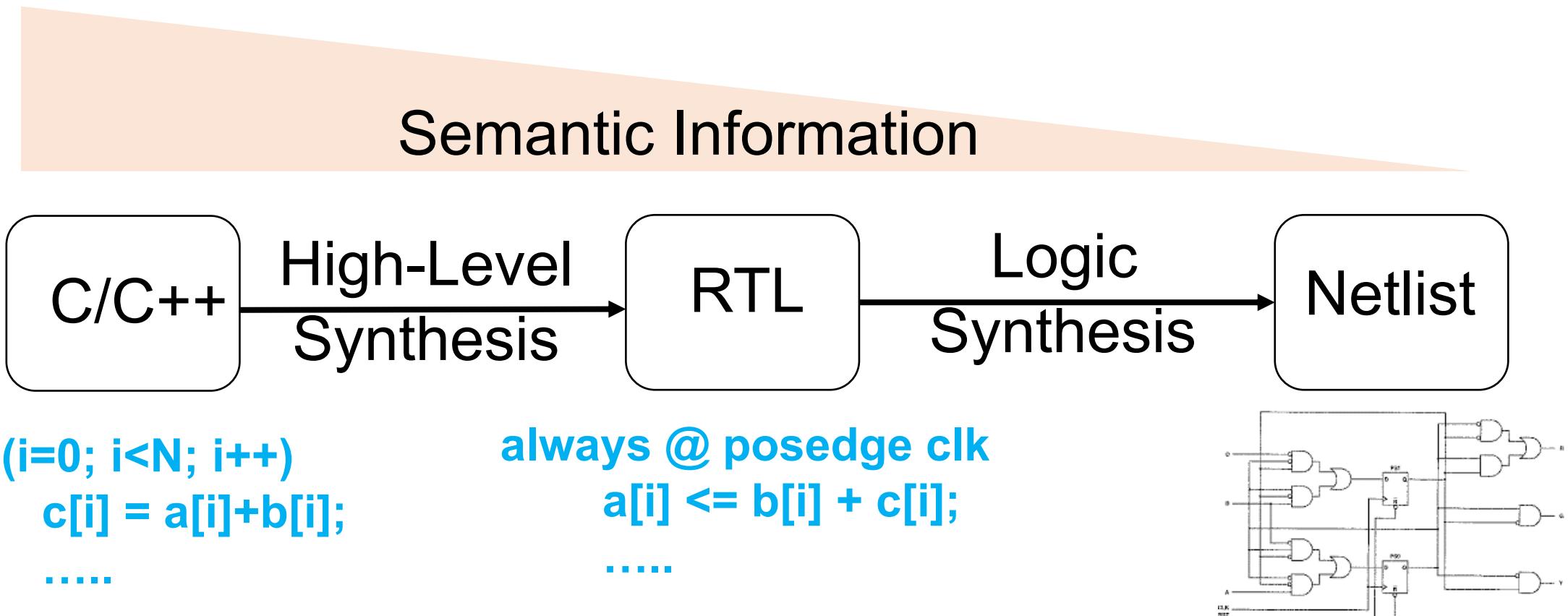


# Logic obfuscation



# Raising the abstraction level

- Key Idea: obfuscate a design at the **algorithm-level** so that the obfuscation is **semantically meaningful**



# Algorithm-level obfuscation

- An algorithm is characterized by several elements to be protected

```
if (cond < N) {  
    c[i] = a[i] + b[i];  
    d[i] = c[i] * CONST_1;  
    ...  
} else {  
    . . .  
}
```

Control flow

Dependences

Operations

Constants

**TAO: Techniques for algorithm-level obfuscation**

# Technique #1: Constant obfuscation

- Constants represent hard-coded values used by the algorithm (coefficients, thresholds, ...)

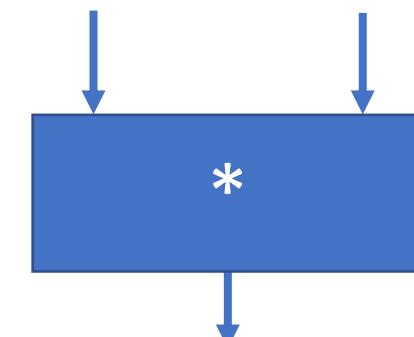
C/C++

```
d[i] = c[i] * CONST_1;
```

Information is still present at RT Level

RTL

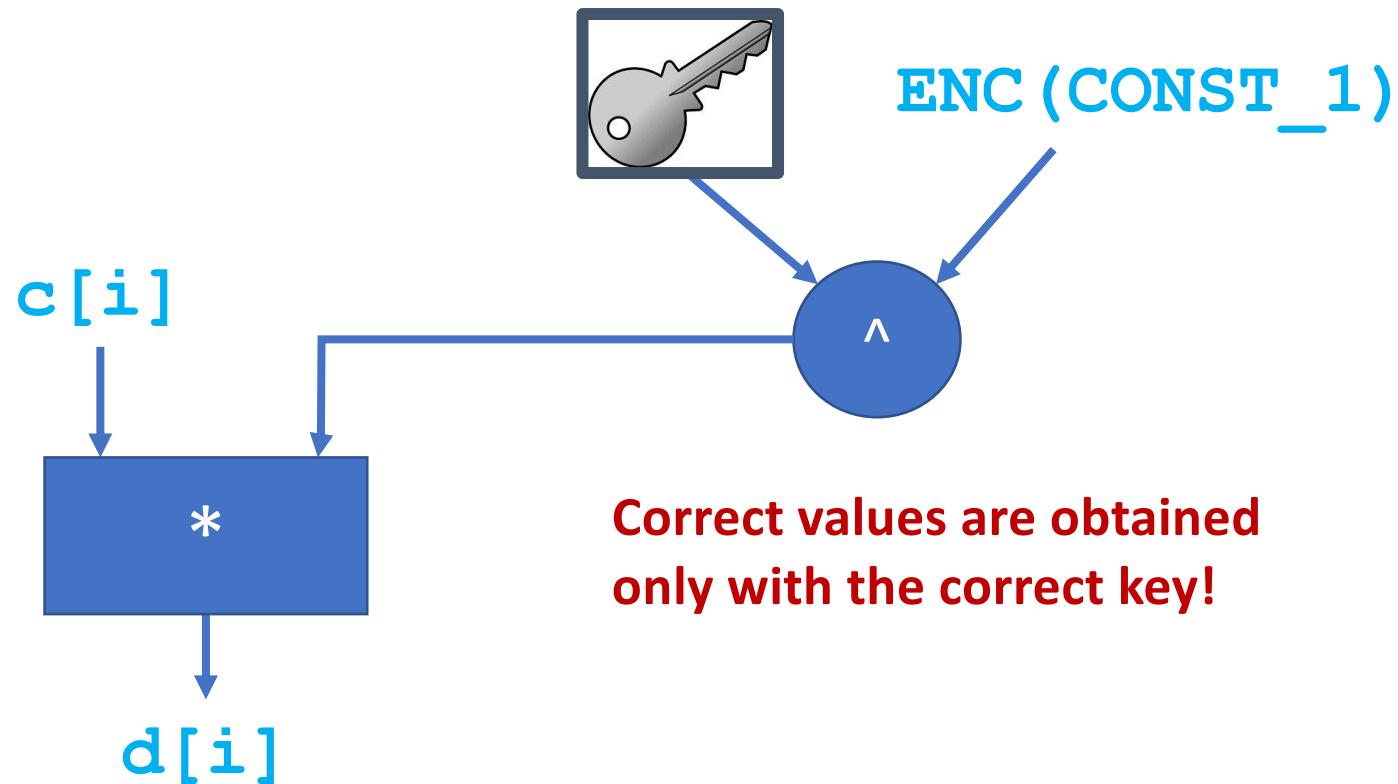
c[i] CONST\_1



Heavily optimized by logic synthesis!

# Constant obfuscation

- A  $m$ -bit constant is **extracted and encoded** using  $m$  working key bits



# Analysis of the Technique

Obfuscated	Non-obfuscated	
Data coefficients used by the algorithm	Reset values	No differences concerning security, less key bits
Signal extension	Signal polarity	No semantic changes
Mask values		

Exact information is removed from the circuit

**Less information for the attacker**

**Less logic optimizations (area overhead)**

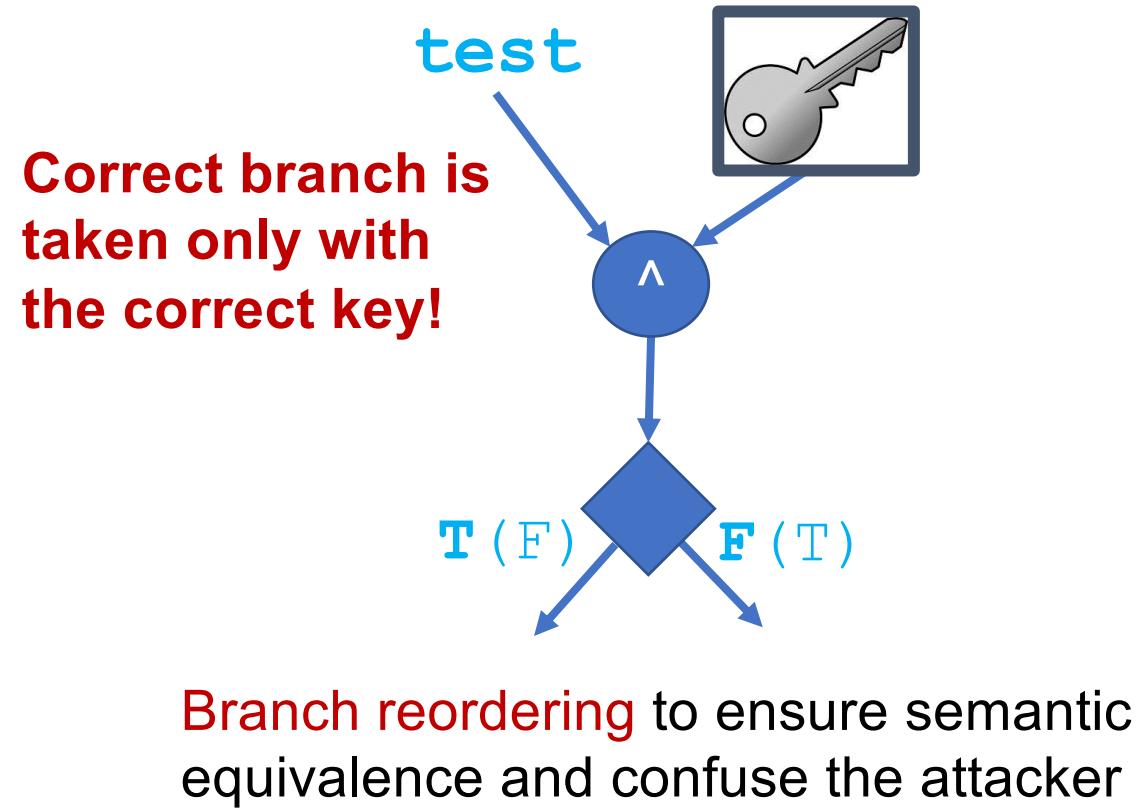
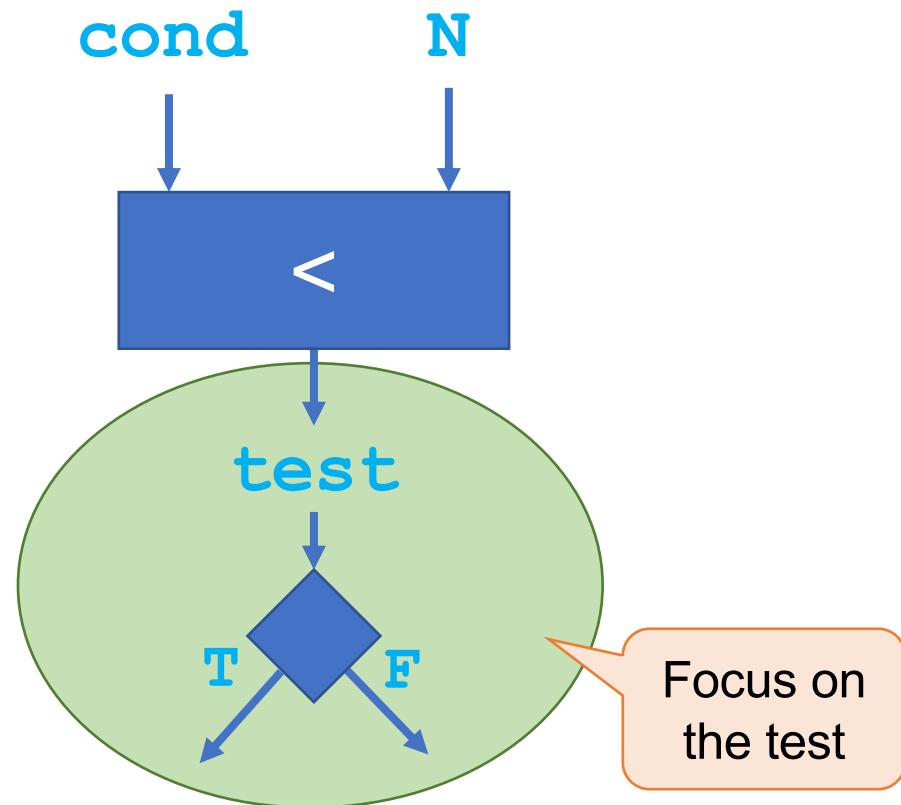
# Technique #2: Control-flow obfuscation

- Control-flow branches describe the evolution of the algorithm

```
if (cond < N) {  
    c[i] = a[i] + b[i];  
    d[i] = c[i] * CONST_1;  
    ...  
} else { ... }  
  
Control  
condition  
  
Attacker can get insights on  
the algorithm based on the  
branch taken
```

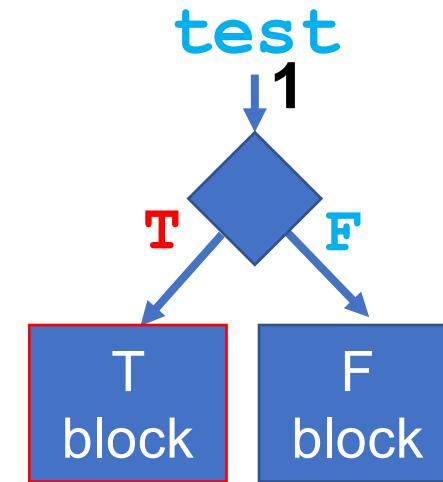
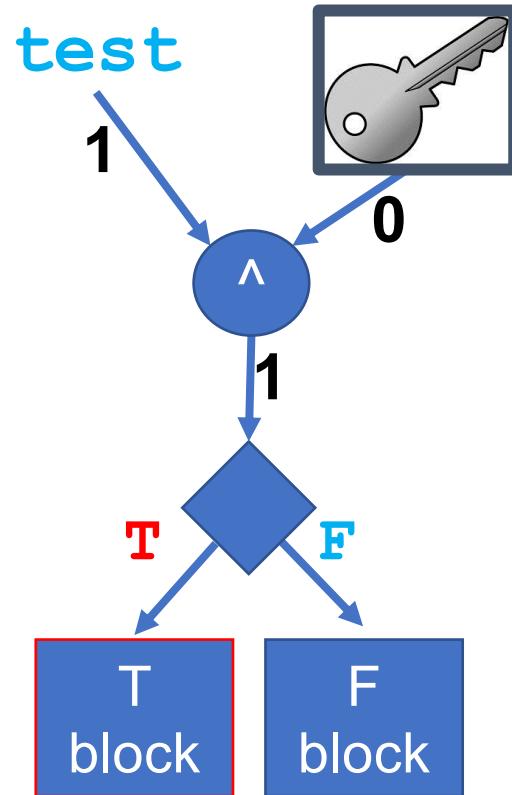
# Control-flow obfuscation

- Masking of the **control condition** with key bit

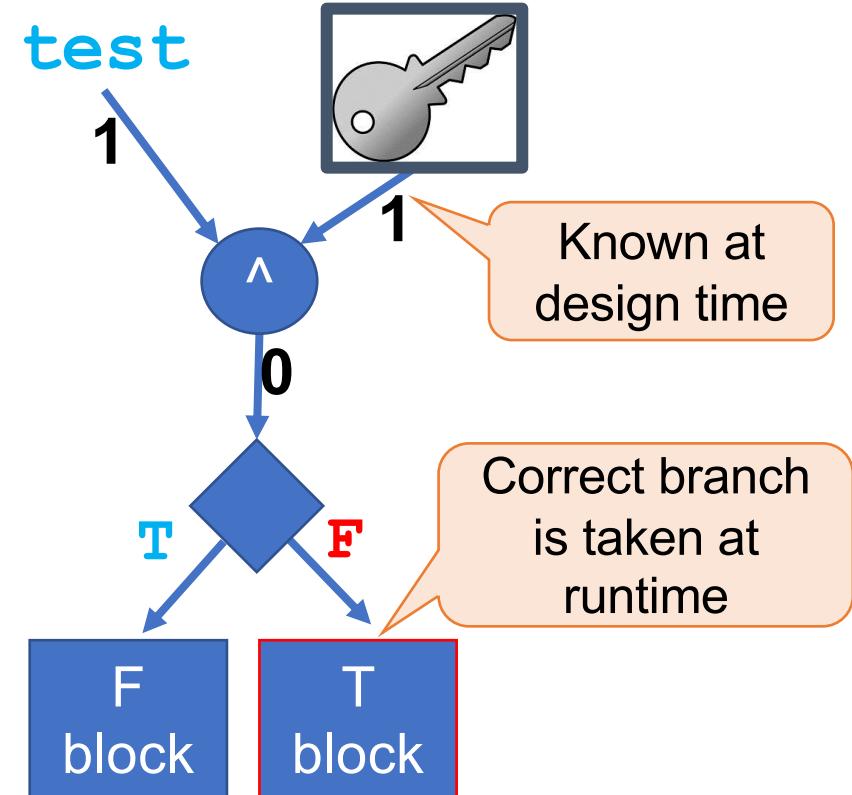


# Branch reordering

- Based on the key bit for **semantic equivalence**



With code inspection, no intel  
on which is the real true/false  
branch without the key bit



# Analysis of the Technique

- Focus on **branches** during data computation
  - No reset changes to allow *register inference* and prevent *synthesis issues*
- Easy to apply
  - **Minimal overhead (XOR gate)**

Almost impossible to replicate at gate level (branch info is already embedded)

# Technique #3: Operation obfuscation

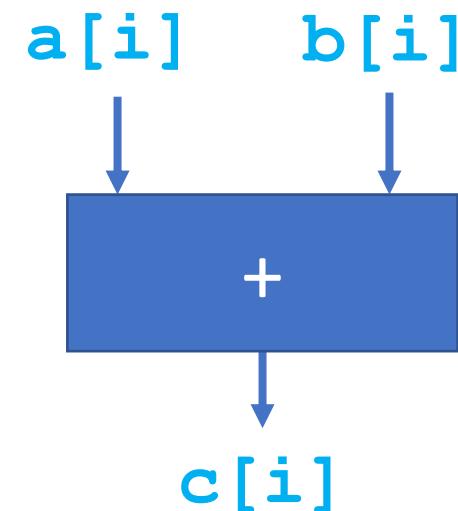
- Operations represent the computational part of the algorithm

C/C++

```
c[i] = a[i] + b[i];
```

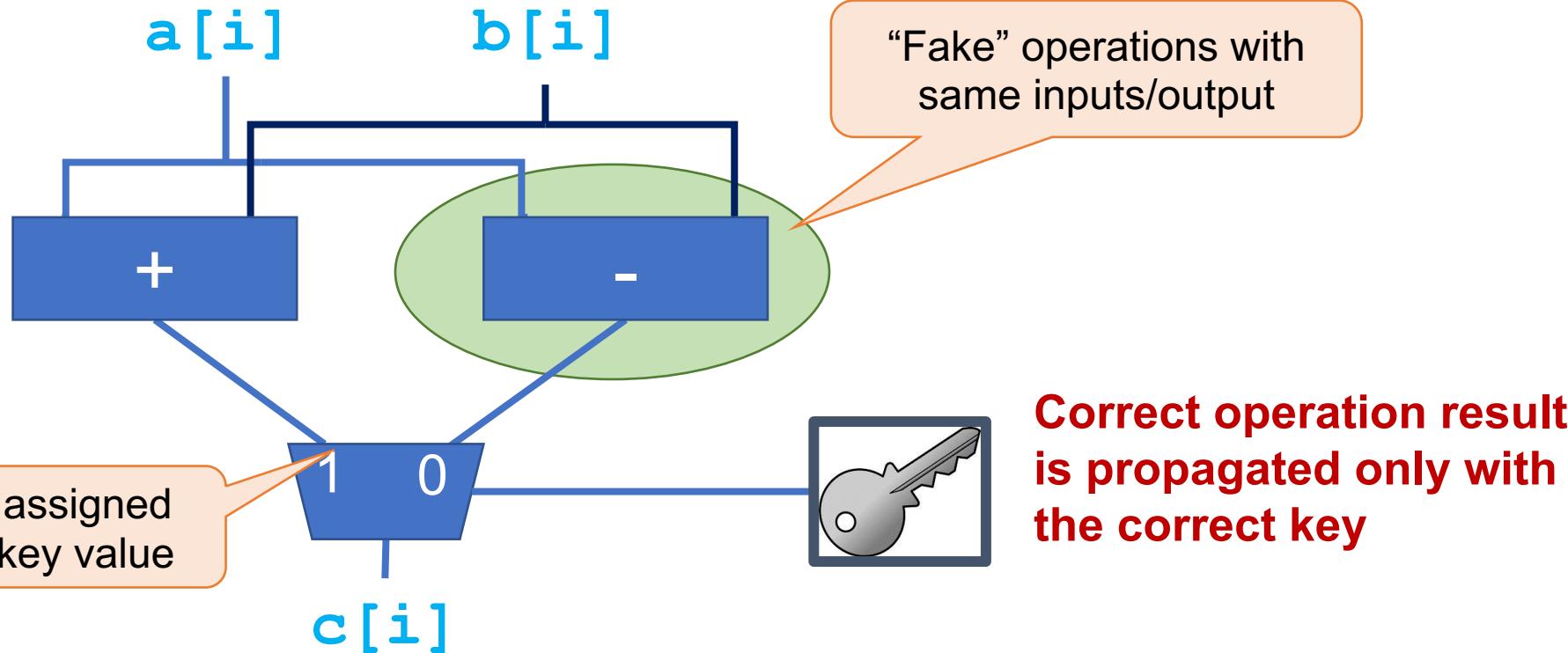
This gives intel on what  
the algorithm does!

RTL



# Operation obfuscation

- Operator variants to camouflage the correct operation



# Analysis of the Technique

- Easy to apply
  - Reasonable area overhead (due to additional *fake operations* plus multiplexer)
- Each operation type has a pre-defined set of alternatives to choose from

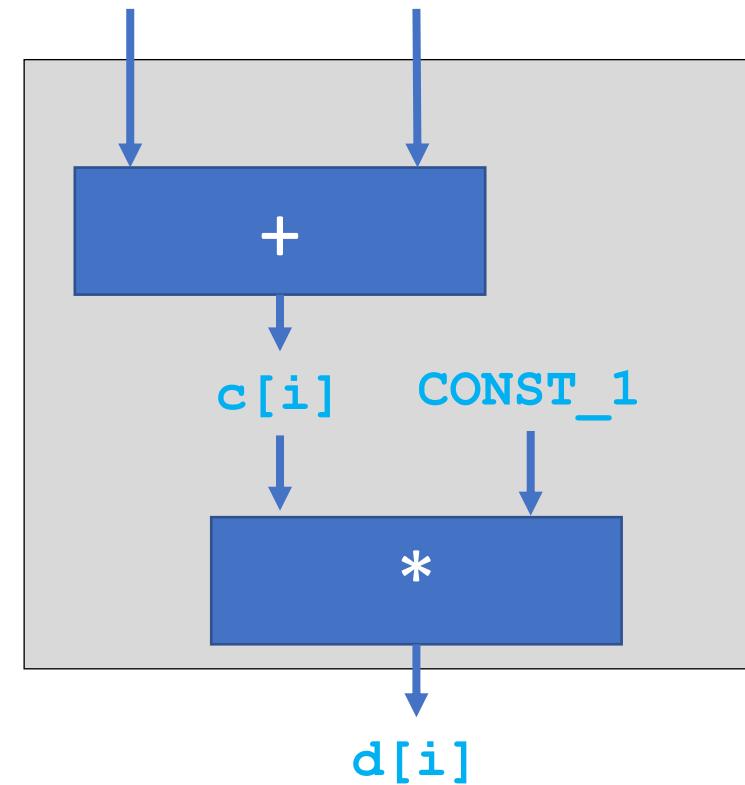
How to select operation variants is still an open issue

# Technique #4: Dependence obfuscation

- Operation dependences describe how the data values are used

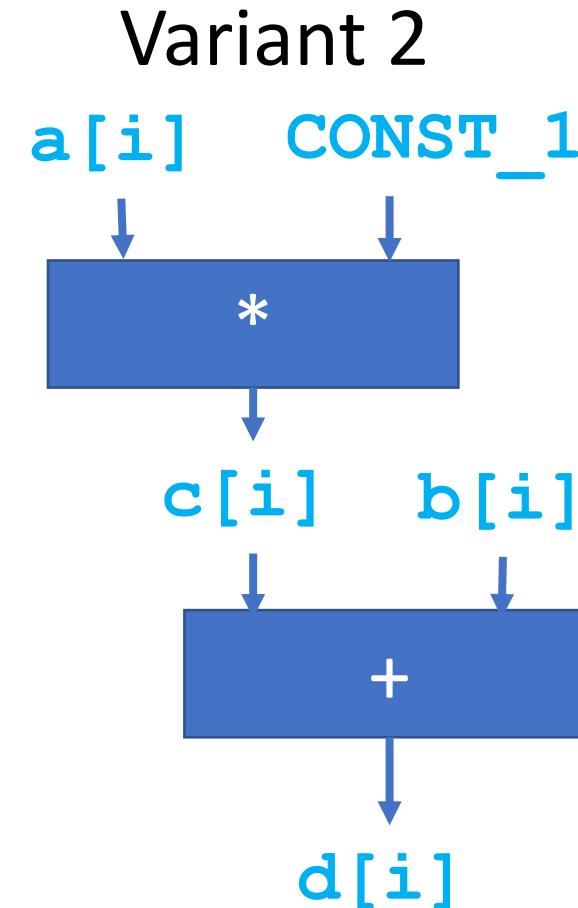
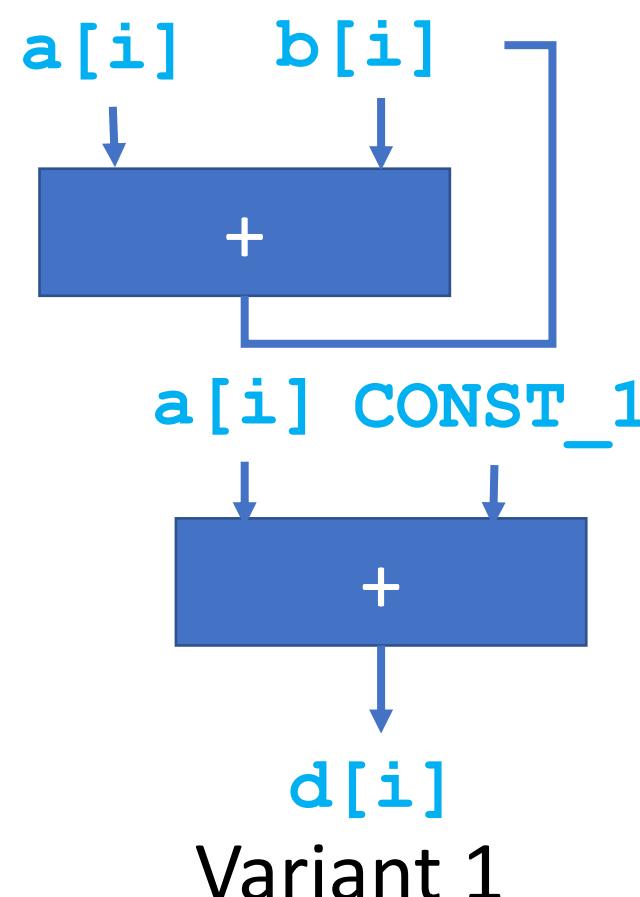
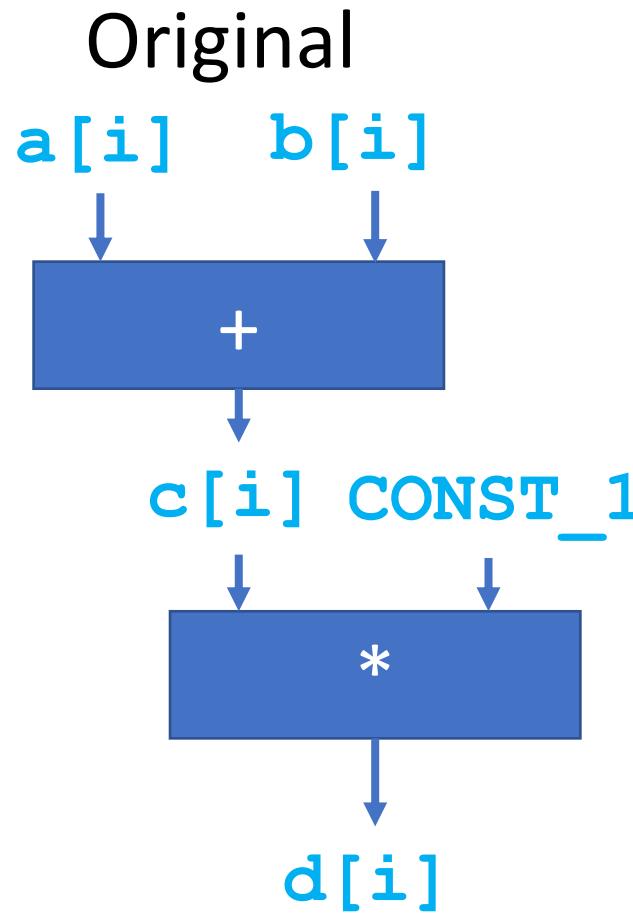
```
if (cond < N) {  
    c[i] = a[i] + b[i];  
    d[i] = c[i] * CONST_1;  
    ...  
} else { ... }
```

Very hard to compute in  
gate-level netlists



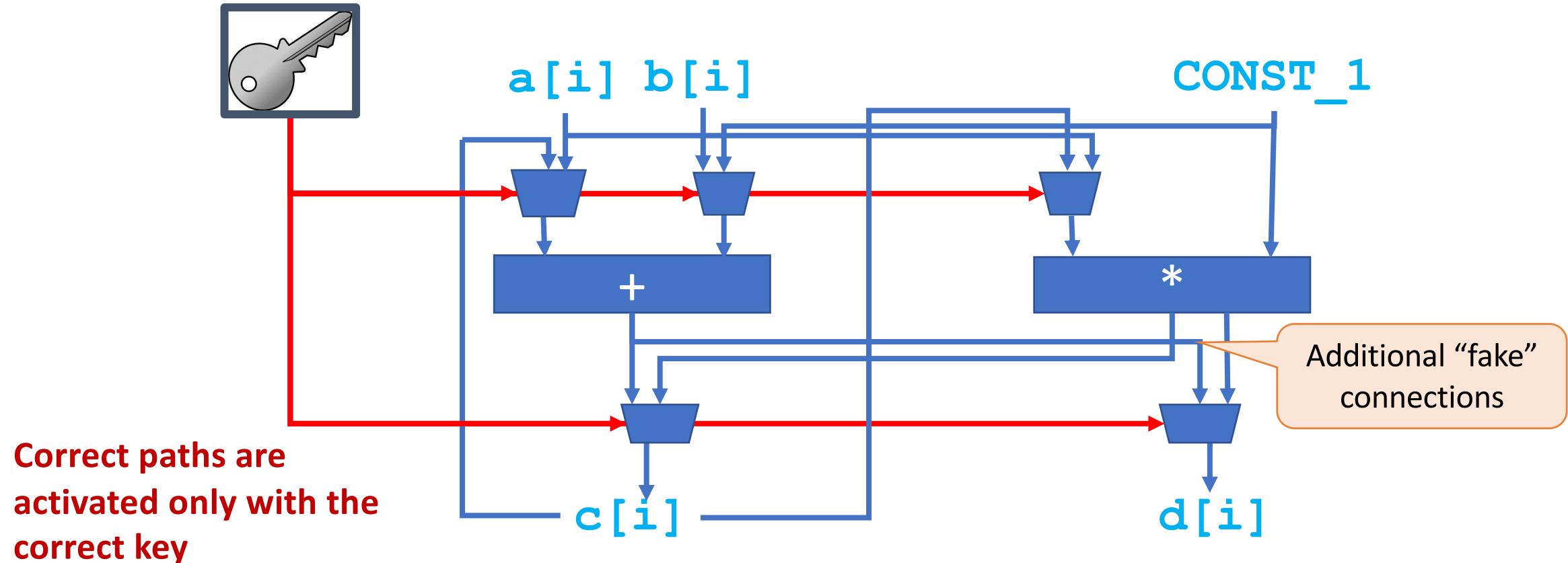
# Dependence obfuscation

- $K$  key bits are used to select among  $2^k$  variants (including original)



# Merge and Selection of DFG Variants

- Creation and merge of DFG variants

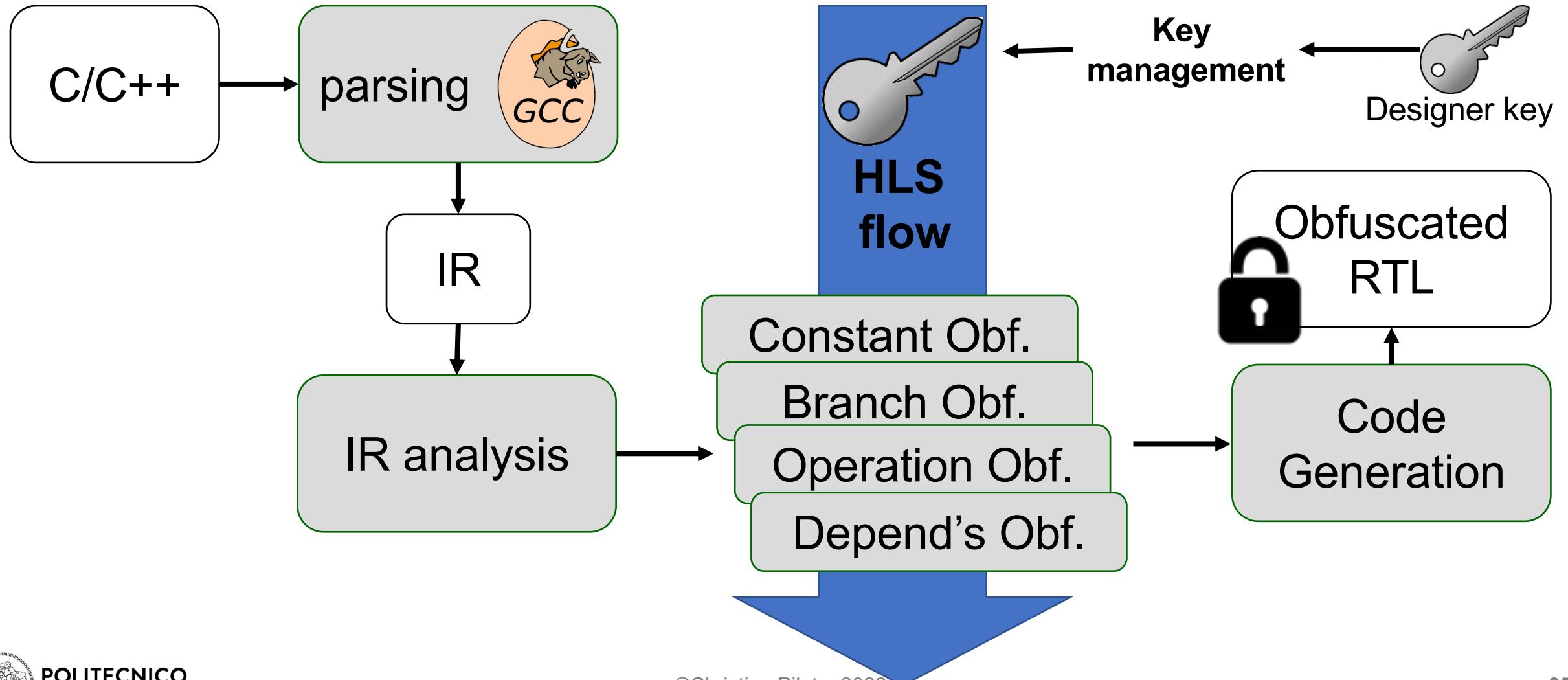


# Analysis of the Technique

- Very powerful technique
  - It creates several semantically-different but feasible code variants
  - It may affect component latency (different schedules)
  - Significant area overhead (many additional operators and connections/multiplexers)

It can be applied only during  
component generation (HLS toolflow)

# TAO in Bambu



# Obfuscation overhead

- We generated obfuscated designs for **five HLS benchmarks**
  - 256-bit locking key in all experiments
- How to read results

Design name	Const. Obf.	Branch Obf.	DFG Variants	Total key bits
GSM	80 / 240	24	32	296

Design name  
Obfuscated constants / Number of used key bits  
Obfuscated branches  
Number of Basic Blocks / key bits  
Total number of used key bits

Operation and Dependence obfuscation techniques are combined into **generation of DFG variants**

# Obfuscation key bits

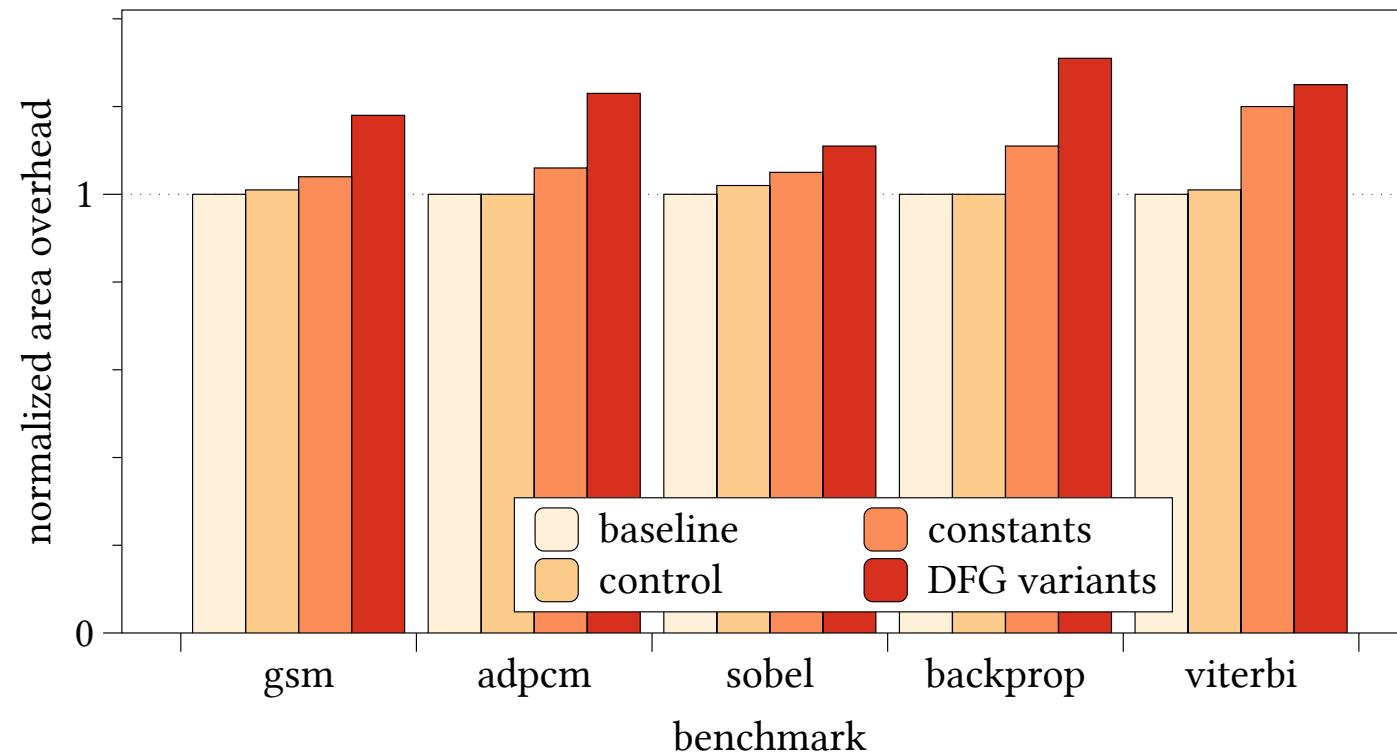
Each constant is converted into a 32-bit signal

4 bits are used for each BB (16 variants)

Design name	Const. Obf.	Branch Obf.	DFG Variants	Total key bits
GSM	4 / 128	4	88 / 352	484
ADPCM	5 / 160	5	100 / 400	565
SOBEL	2 / 64	2	11 / 44	110
BACKPROPAGATION	12 / 384	11	123 / 492	887
VITERBI	117 / 3,744	9	98 / 392	4,145

# Area overhead

- Area overhead of each technique wrt the **baseline** version
  - Synopsys SAED 32nm @ 500 MHz



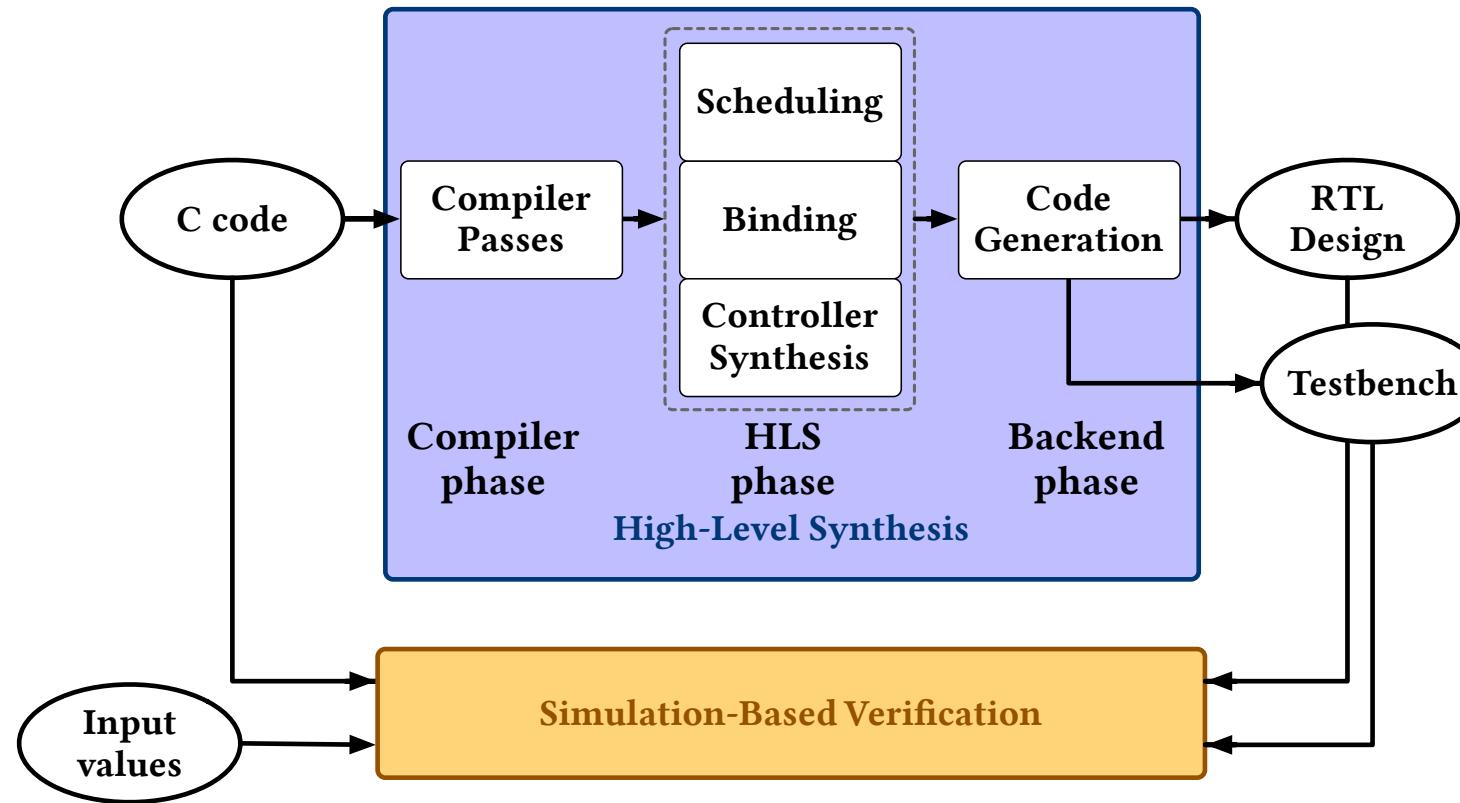
# Algorithm-Level Obfuscation Conclusions

- Comprehensive solution for **algorithm-level obfuscation** during HLS
  - Four techniques for **constants, control flow, operations and dependences**
  - Two solutions for **key management** (key folding and AES-based architecture)
- Obfuscation results validated with “**output corruptability**”
  - Hamming distance between output values generated with correct and incorrect keys

**How to select the parts to protect to minimize the overhead?**

**How to “measure” the level of obfuscation/security?**

# Simulation-Based Verification

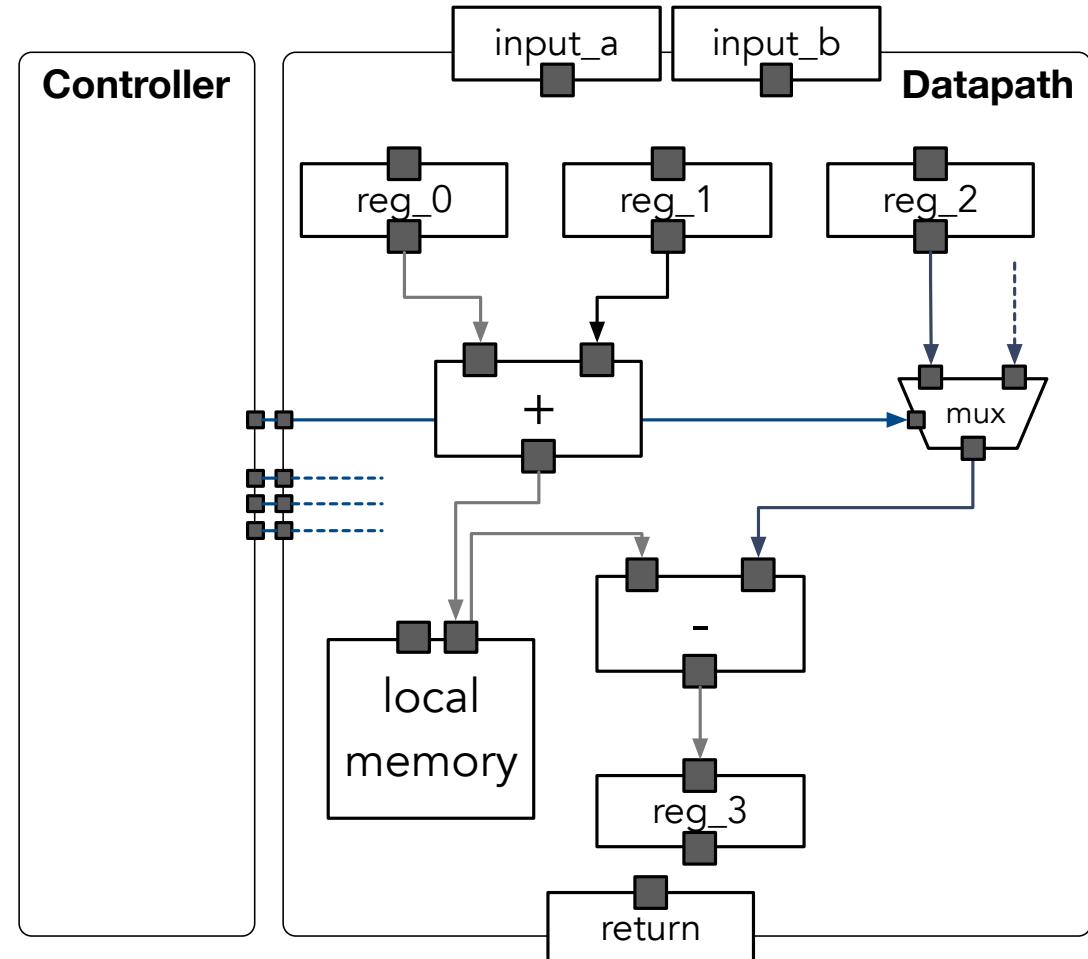


If modified design can escape simulation-based verification,  
it becomes the golden model

# Benevolent Hardware Trojans

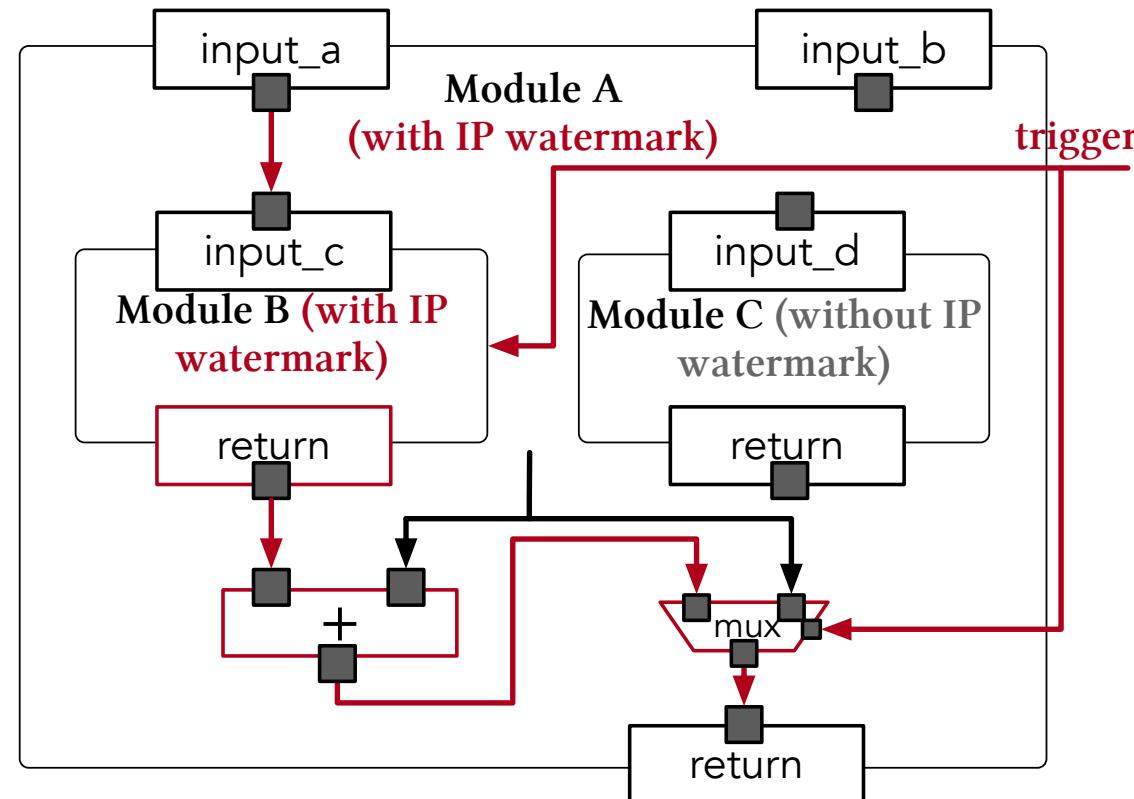
The structure of a **Hardware Trojans** can be used for watermarking

- Stealthy during normal execution
- Activated during rare condition  
(similar to a **trigger**)
- Small and power function  
intertwined with the  
functionality (**payload**)



# Architecture Modification with Watermarks

Only one function contains the watermarks but the others must propagate the values



# Generation of Watermark

- We analyzing the call graph of the input C code
  - the functions with the largest number of operations or the most frequently executed ones.
- We list all functions in **reverse topological order**
  - When analyzing one module, all the submodules are analyzed to properly propagate input and output values.

List of operations with probability of being removed

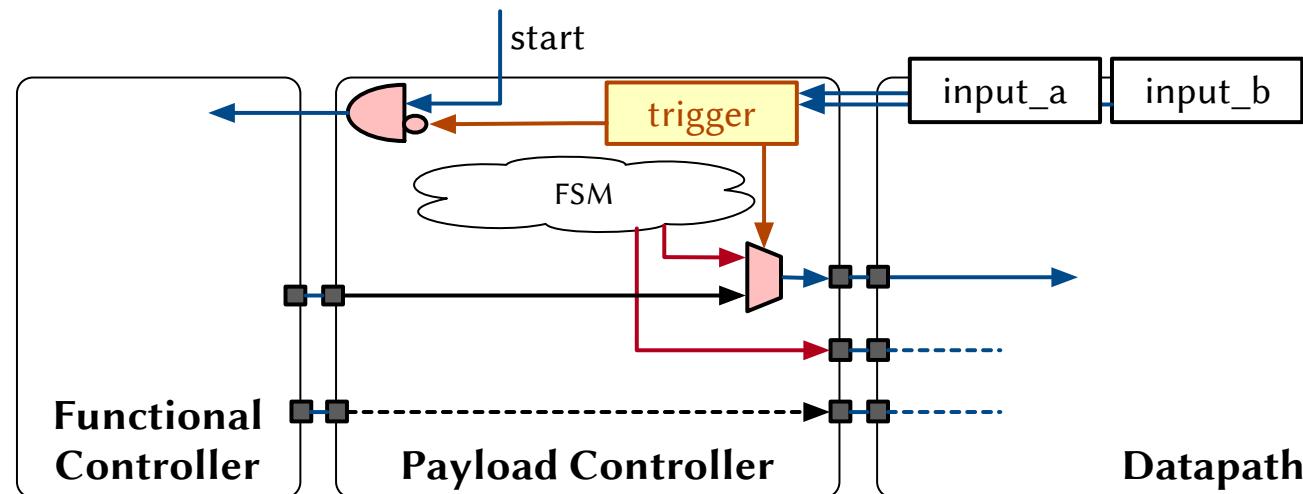


**Low probability  
of coincidence**

# Generation of Datapath

Rebuild the connection based on the remaining operations

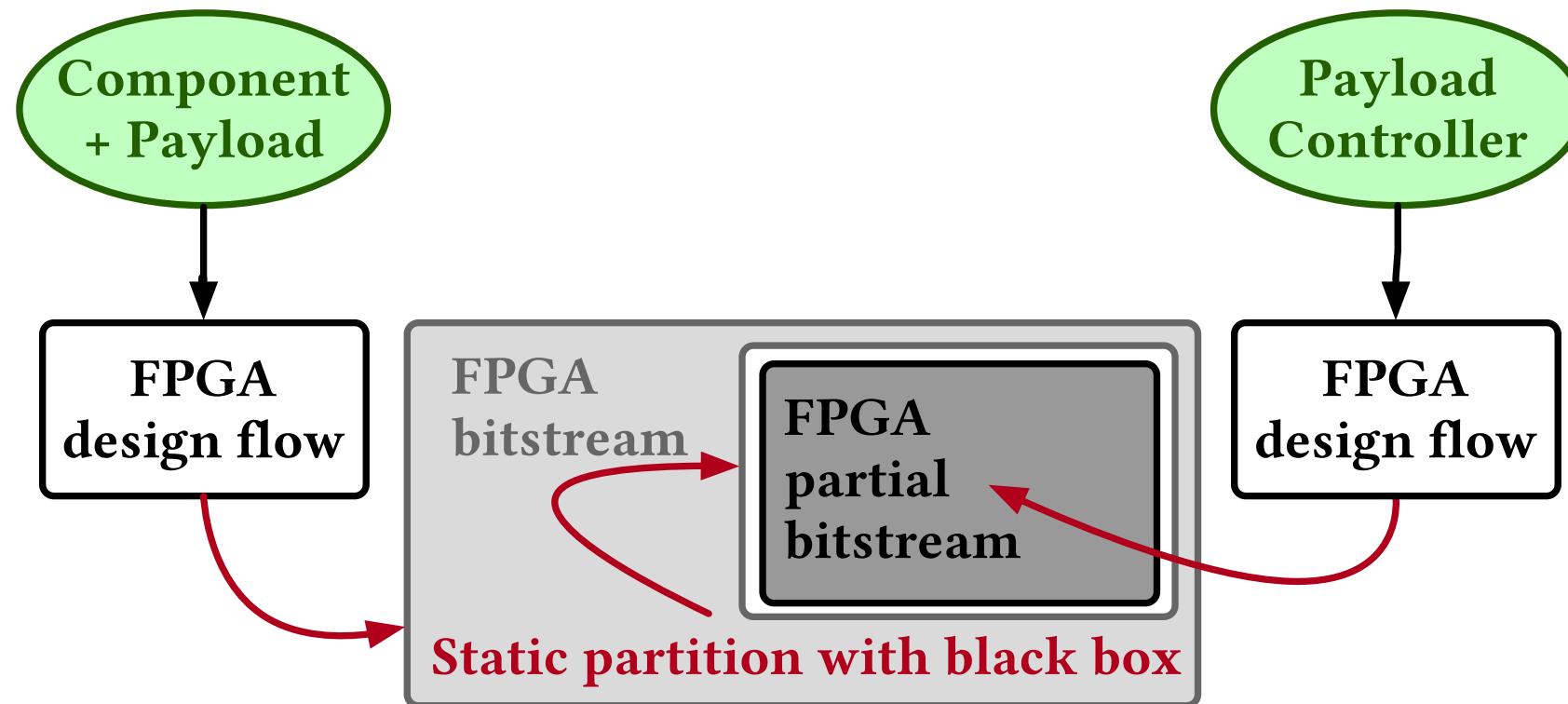
- Only the HLS tool knows the actually-removed operations and thus the «signature» -> able to generate **golden values** for verification



**Payload and functional controllers have the same structure  
so they can be merged to save resources**

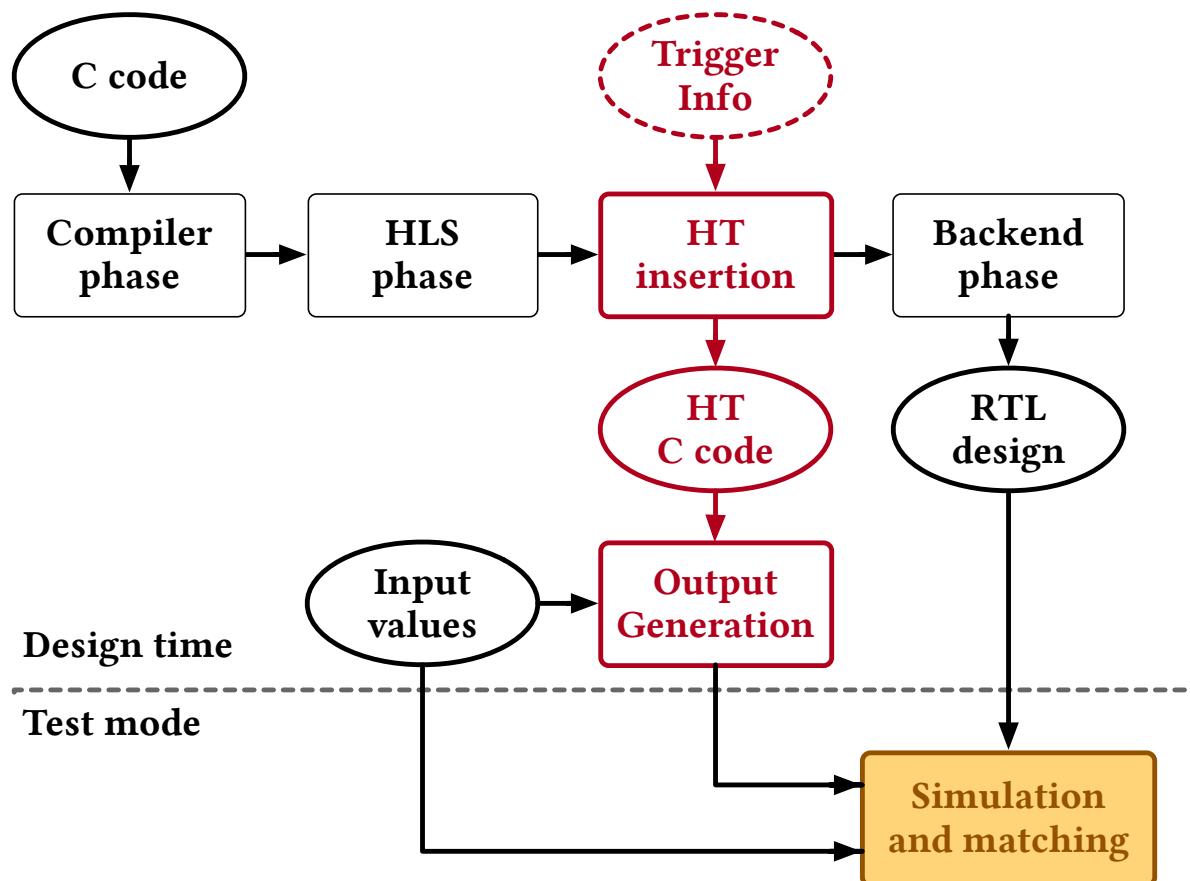
# Activation with Partial Bitstream

Only in specific cases it is possible to separate the payload controller and deliver it afterwards as a **partial bitstream**



# Experimental Setup

Bambu HLS framework extended with Trojan/watermark insertion



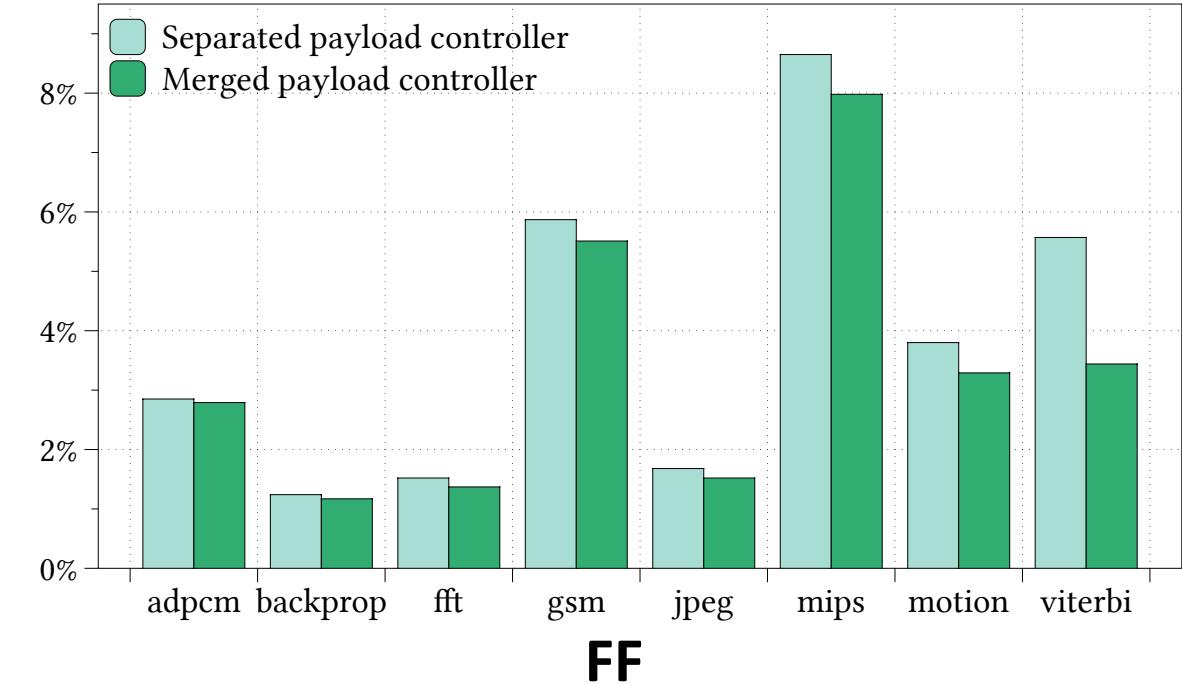
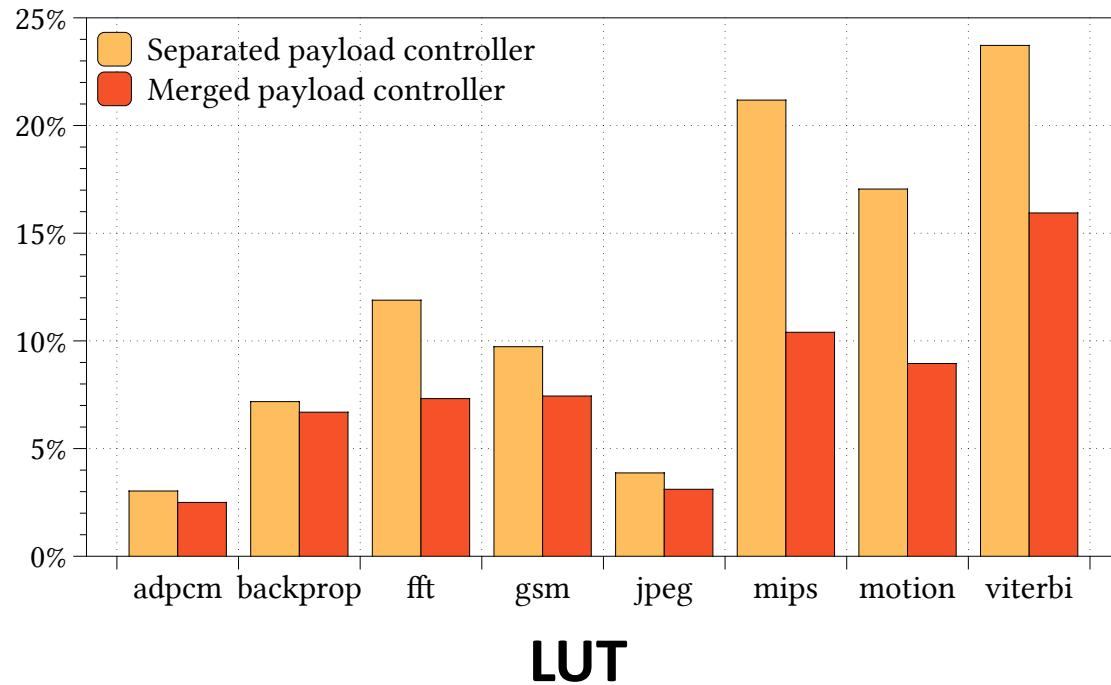
Xilinx Virtex-7 FPGA (xc7vx485t) at 100 MH

Logic synthesis using Xilinx Vivado 2018.2

Mentor ModelSim SE 10.3

# Area Overhead of the Watermark

Merging controllers reduces the overhead, especially in control-based designs



# IP Watermarking Conclusions

- Technique to implement IP watermarking using **benevolent hardware Trojans** during HLS.
  - Hardware overhead reduced by resource reuse
  - Overhead around 8% (LUT) and 3% (FF)
- Two proposed controllers activated with different activating conditions
  - External pin
  - Predefined set of input values (trigger-based)

# How is Using HLS for Hardware Security?

**Good:** automatic generation of protection mechanisms

- Fine-grained Dynamic Information Flow Tracking
- Algorithm-Level Obfuscation

**Bad:** potential attack vector

- Planned Obsolescence
- Key Recovery with Reduced-Round Attacks

**Ugly:** Dream vs. Reality

- What is Missing?

# Planned Obsolescence

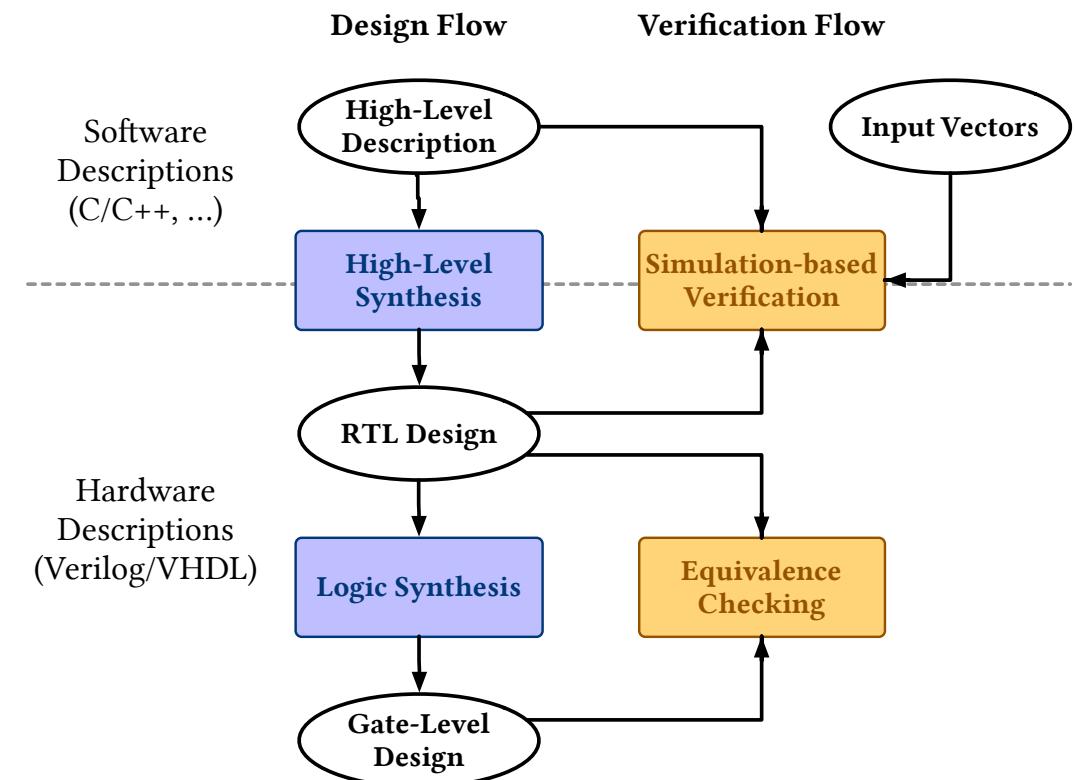
Design houses (or competitors) may have interest to degrade IPs after a certain amount of time

- Pushing customers to change device

## Italy Fines Apple, Samsung A Few Mil For 'Planned Obsolescence' In Phones

(Forbes, Oct 28, 2018)

Very difficult to check  
**non-functional properties**



# CAD Tools are Designed by Humans...

Can you always trust a programmer?

- Circuit CAD tools are known to be a potential **attack vector**

## Extended Abstract: Circuit CAD Tools as a Security Threat

Jarrod A. Roy†, Farinaz Koushanfar‡ and Igor L. Markov†

†The University of Michigan, Department of EECS, 2260 Hayward Ave., Ann Arbor, MI 48109

‡Rice University, ECE and CS Departments, 6100 South Main, Houston, TX 77005

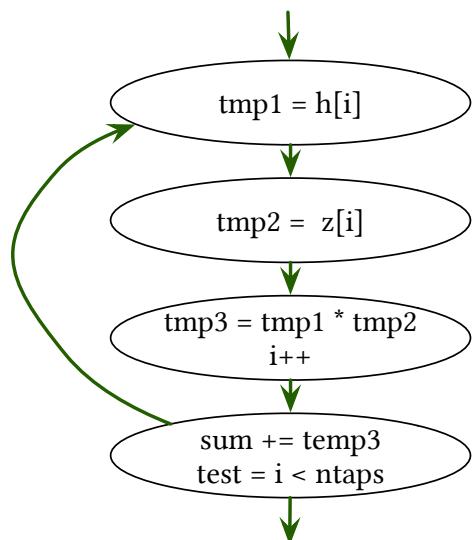
**"Black-hat HLS" is possible!**

# Attack #1: Degradation Attack

It aims at **degrading the performance** of the IP core after a pre-defined amount of time (number of executions)

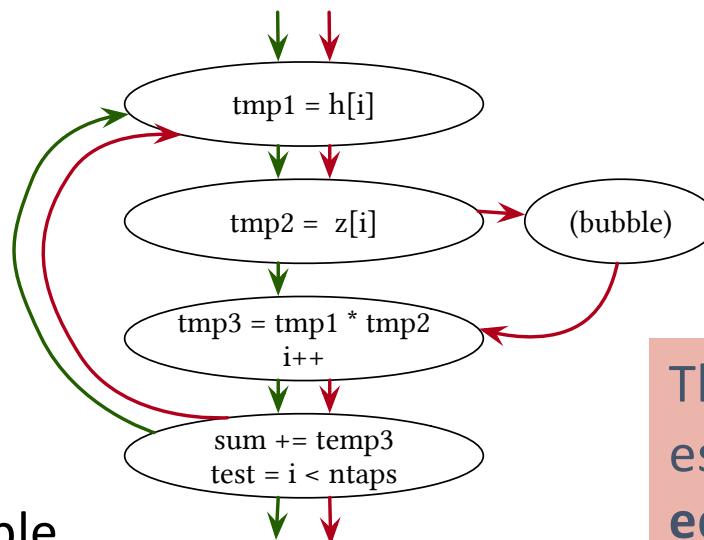
- **Bubble states** inserted in specific points of the FSM to maximize impact

## Original FSM



Trigger is a simple execution counter

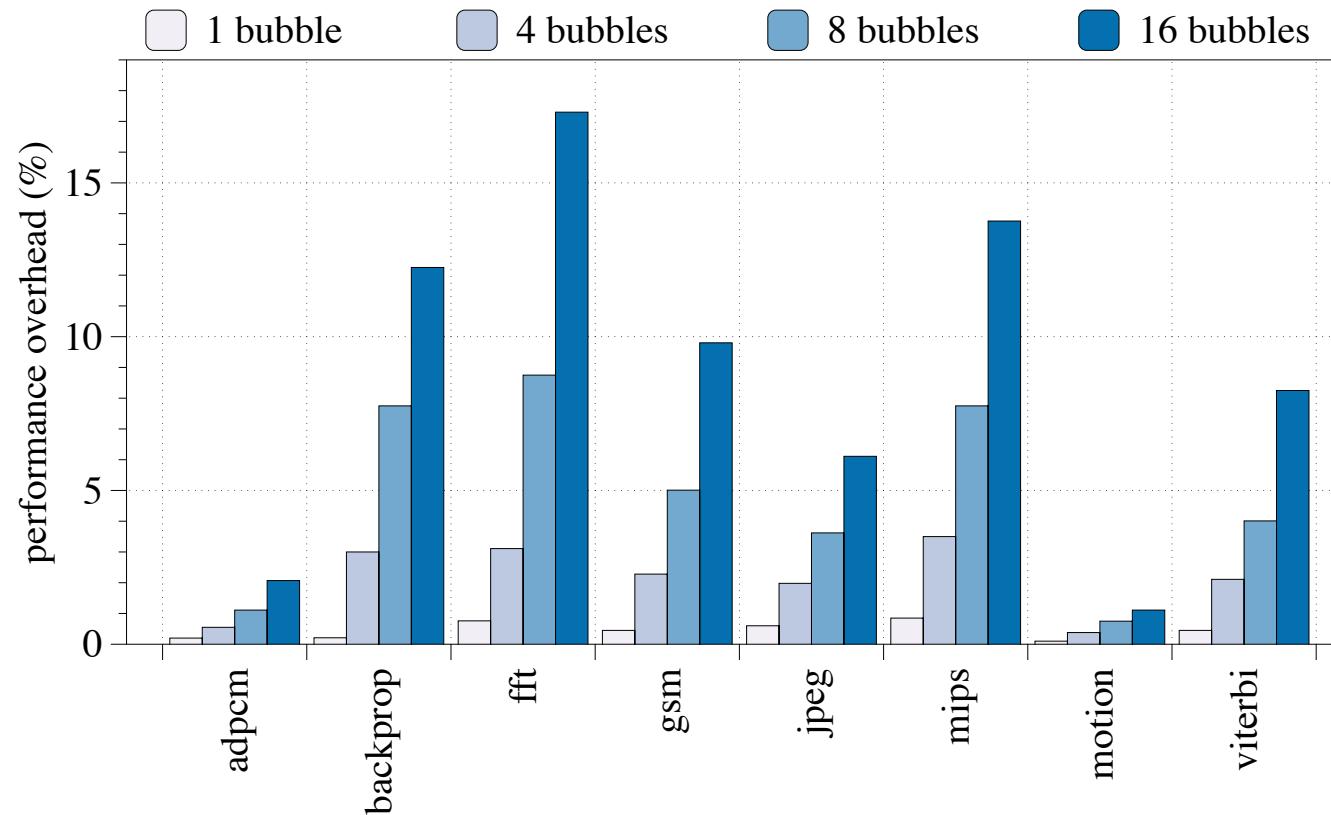
## Compromised FSM



This attack can easily escape sequential equivalence checking

# Degradation Attack in Bambu

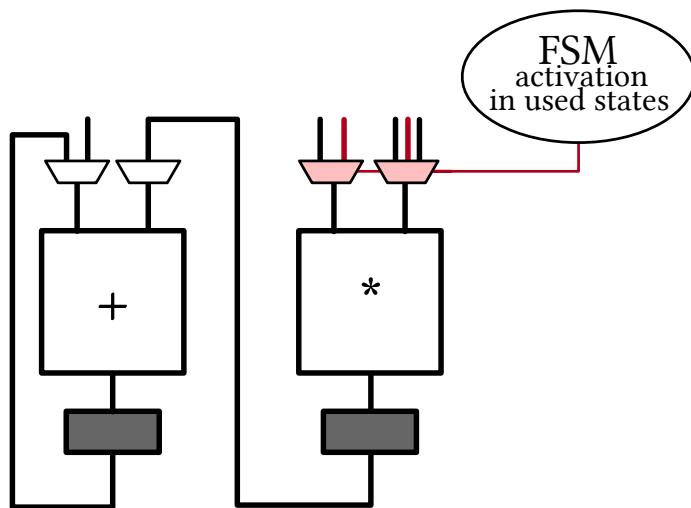
We added a **malicious pass** after the scheduling to insert a **configurable number of bubbles**



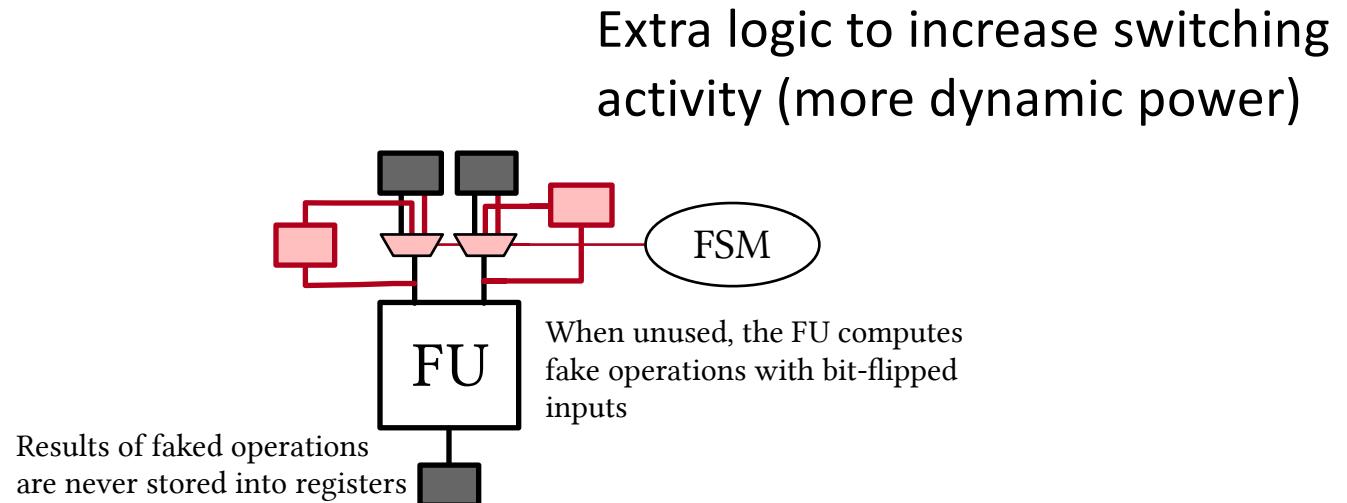
# Attack #2: Battery Exhaustion Attack

Accelerated battery discharging can motivate people to change device

- HLS knows which functional units are used in each clock cycle
- Unused units can be used to drain extra current



Selected functional units are extended with extra logic active only in specific states

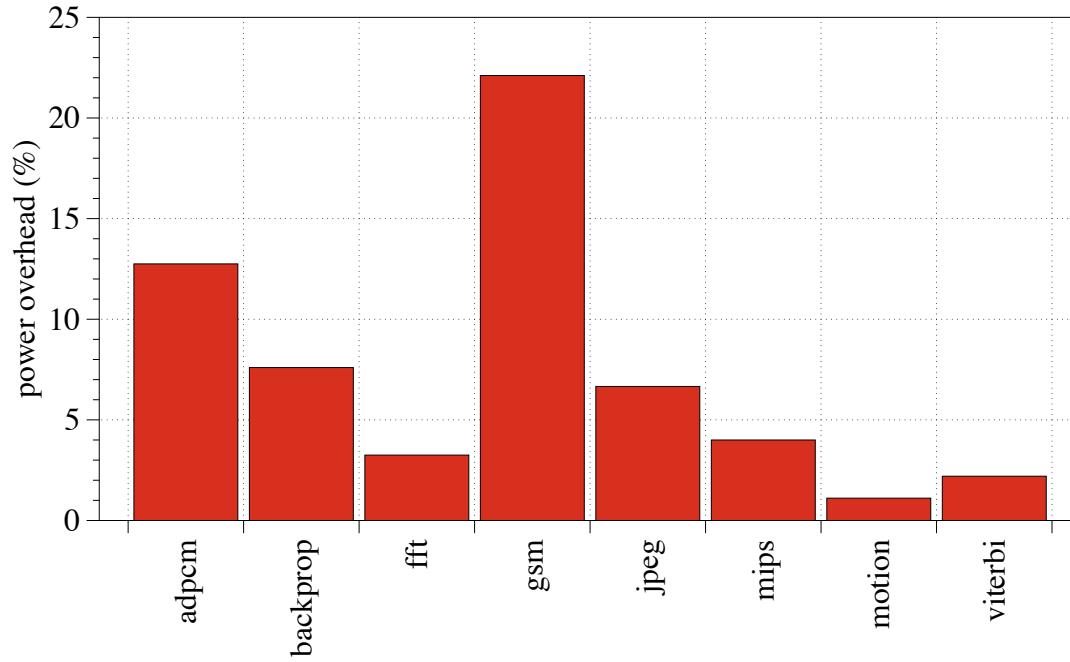


This is no golden model  
before HLS for power analysis

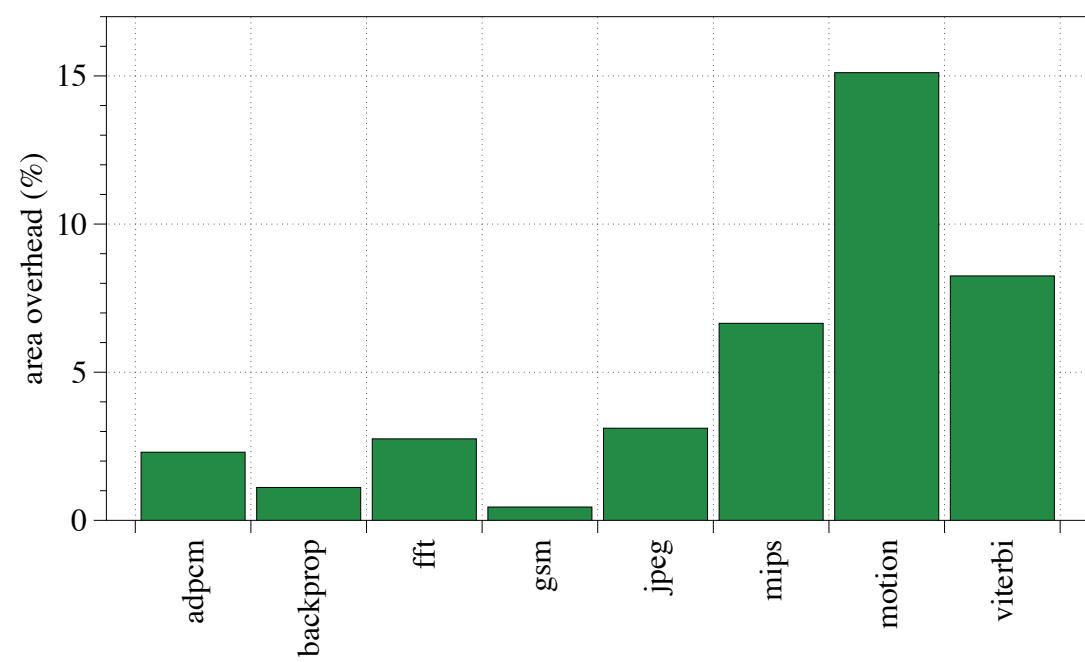
# Battery Exhaustion Attack in Bambu

We added a **malicious pass** after binding to add extra logic

- Tech library provides information about power consumption



Select only the 5 most unused functional units to minimize area overhead



Minimize area overhead with a 30% power overhead budget

# Key Recovery with Reduced-Round AES

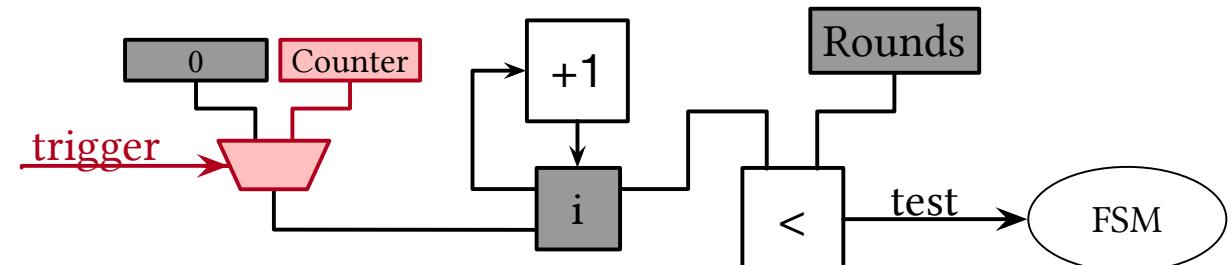
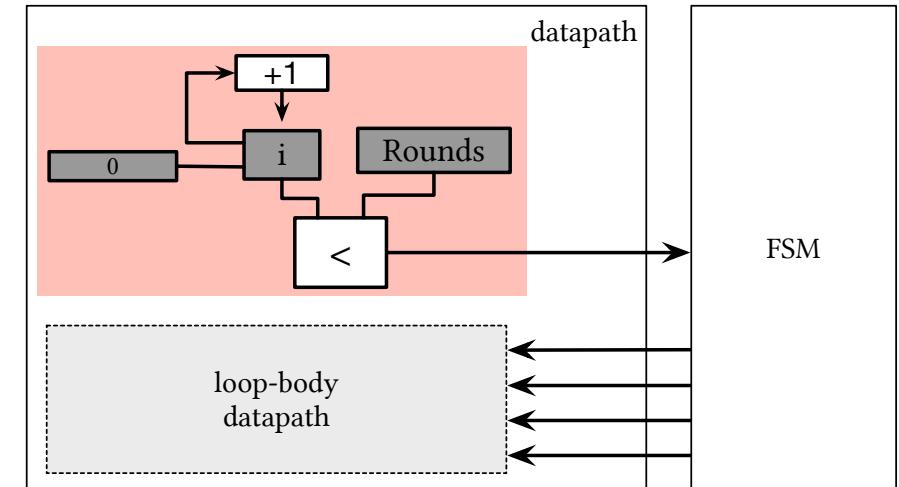
Many cryptographic algorithms execute multiple «rounds»

- AES-128 with 10 rounds
- SHA-256 with 64 rounds

Reducing the number of rounds can ease key recovery

- AES-128 can be broken with 7 rounds
- SHA-256 generates collisions with 18 rounds

Requires collusion between HLS developer and IP developer



# How is Using HLS for Hardware Security?

**Good:** automatic generation of protection mechanisms

- Fine-grained Dynamic Information Flow Tracking
- Algorithm-Level Obfuscation
- IP Watermarking

**Bad:** potential attack vector

- Planned Obsolescence
- Key Recovery with Reduced-Round Attacks

**Ugly:** Dream vs. Reality

- What is Missing?

# What is the Dream?

**Clear metric** to certify that a component (or a system) is secure



Push-button solution to create a **complete and secure architecture**



Your data is secure, and no one can use your intellectual property



Easy to prevent attacks and/or identify attackers

# What is the Reality?

**Hardware security** is critical since «hardware patches» are not possible

**Security certification** is impossible

- You can be effective only against what you know
- Security is a **cat-and-mouse game** and attackers are always one step ahead

Hard to make a long-term contribution

The goal is to make the life  
(exponentially) harder for attackers

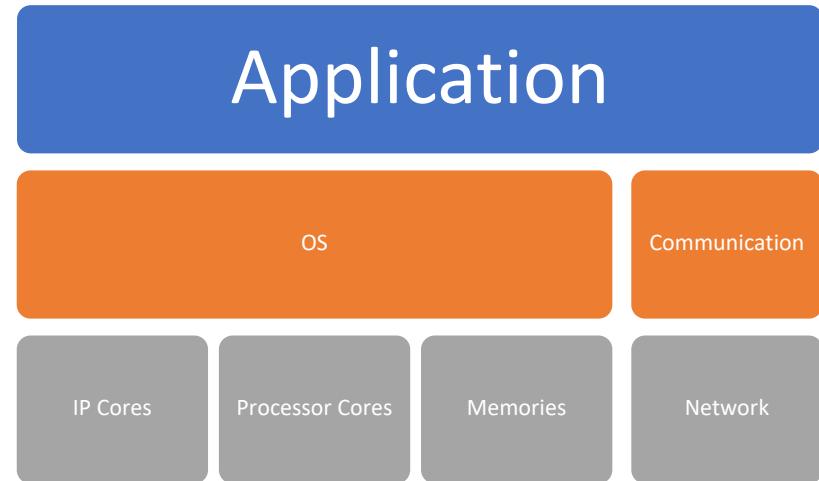
... but at which level?  
and at which cost?



# What is Still Missing?

Security must be address at **ALL levels**

- Provably-secure algorithms
- Robust OS and protected communications
- Secure components, secure architectures, secure component integrations, etc...



Complete and integrated solutions are missing at all levels!

- **Separation of (security) concerns** are required for scalable solutions

Creating **awareness of the problems** is as much important as proposing countermeasures

# **Design of Hardware Accelerators**

Academic Year 2021/2022

# Questions?

[christian.pilato@polimi.it](mailto:christian.pilato@polimi.it)