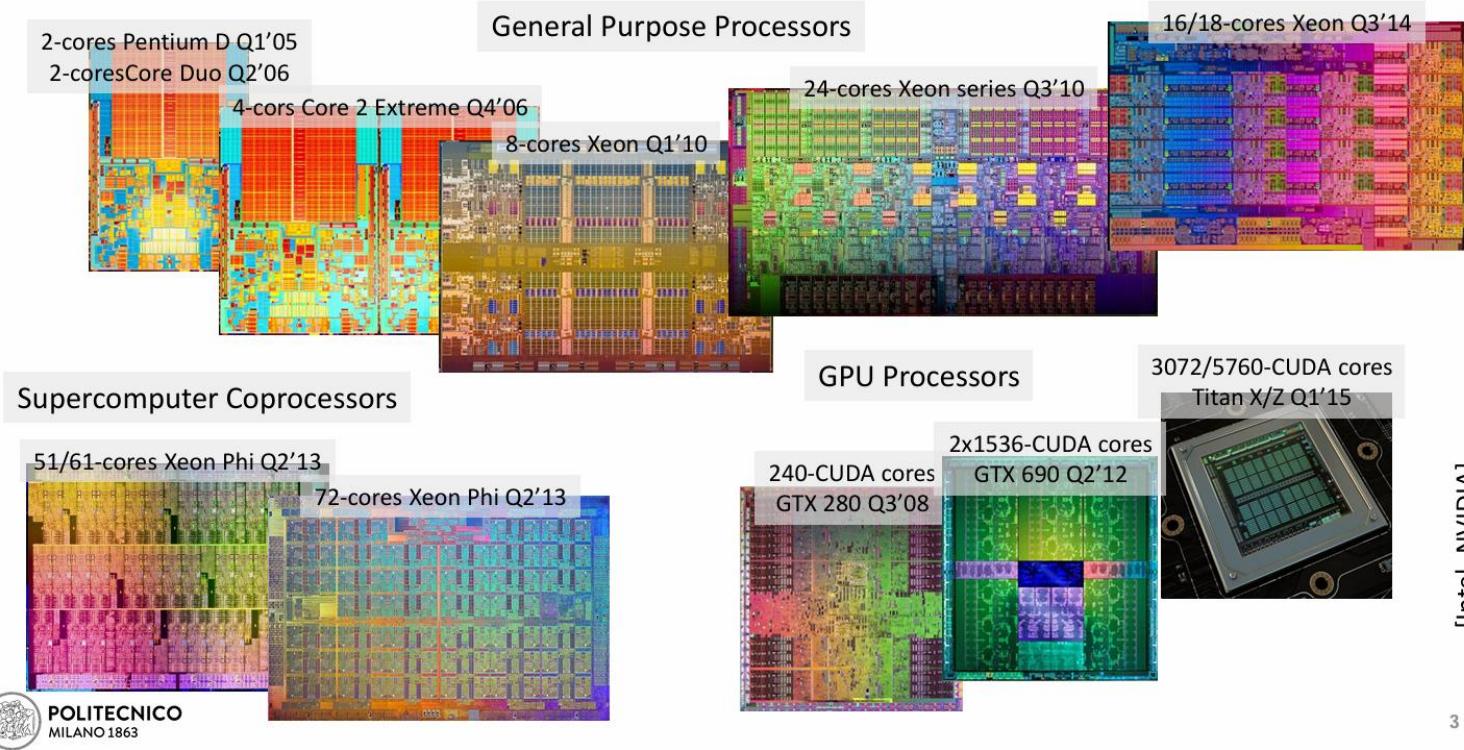


Design of Hardware Accelerators 2024/2025

1 - Introduction to SoC

Slide 3

Increasing Number of Cores



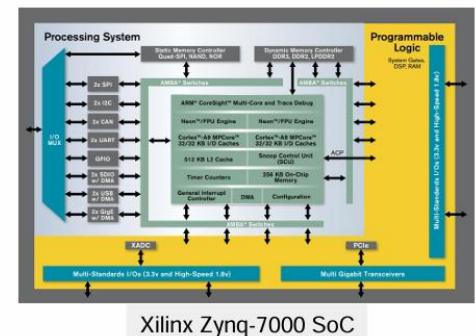
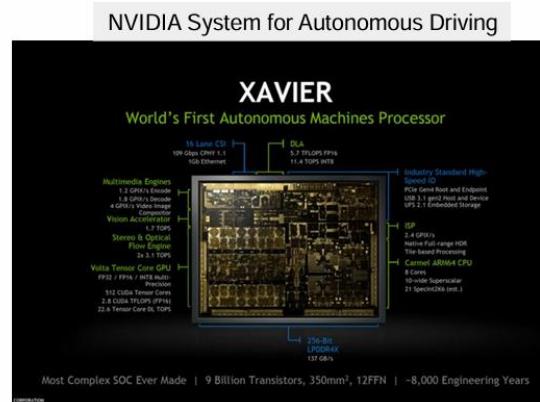
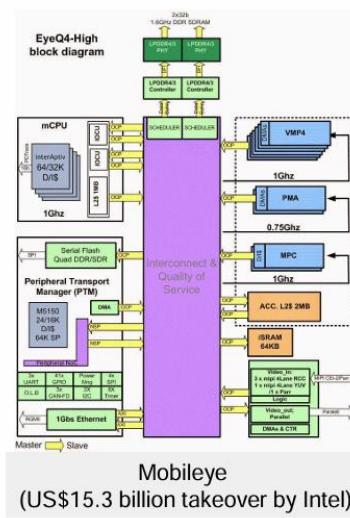
[Intel, NVIDIA]

3 3

General purpose processor: in this course we define as General Purpose Processor any device that can run code. Technically this is not exactly correct because we should define the difference between a microcontroller and a microprocessor, still we do give this general definition.

Slide 4

From Multicore to Heterogeneous Systems



Heterogeneous System: system composed of several types of components

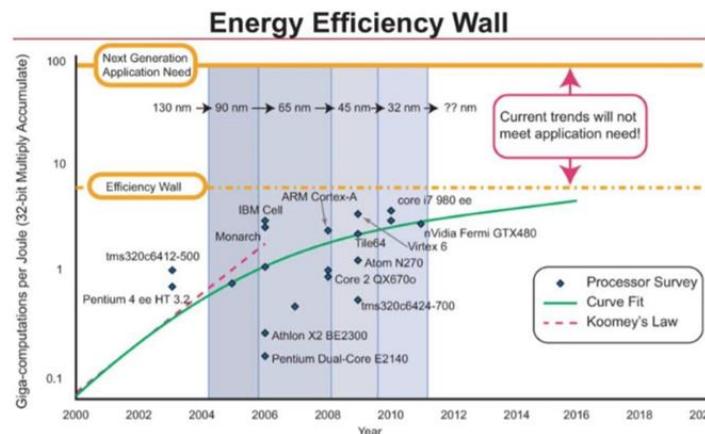
... but why?

Slide 5

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]

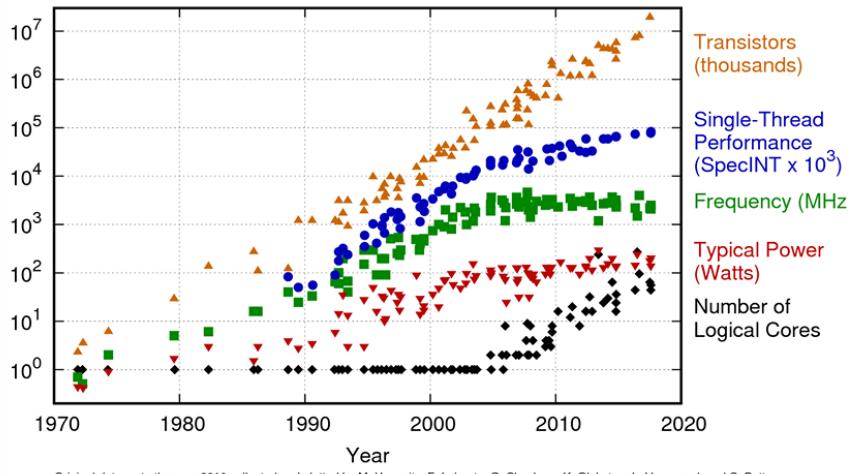
Koomey's Law: “at a fixed computing load, the amount of battery you need will fall by a factor of two every year and a half”

(*Circuit delay is reduced)

Slide 6

The Trend is not Infinite...

42 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

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!

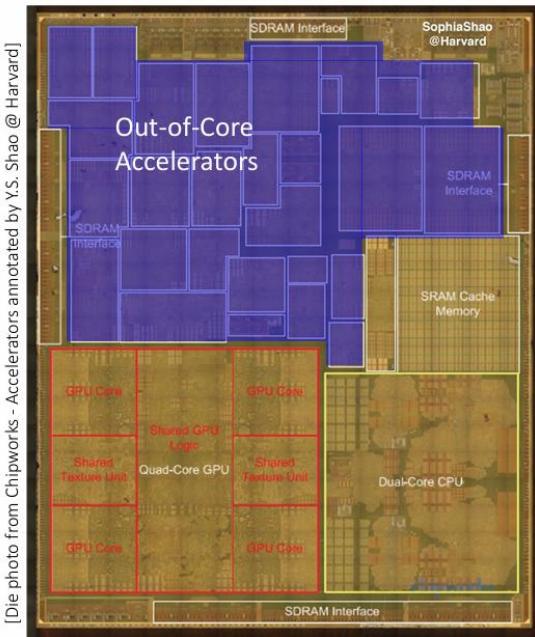
“With each successive generation, the percentage of a chip that can actively switch drops exponentially due to power constraints”

We entered the **dark silicon era**

Dark Silicon refers to portions of a silicon chip that must remain powered off or underclocked at any given time due to power and thermal constraints. This phenomenon arises because the rate of power consumption in modern processors is outpacing improvements in cooling and energy efficiency, preventing all transistors from being active simultaneously.

Slide 7

Heterogeneous SoCs on the Market



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)

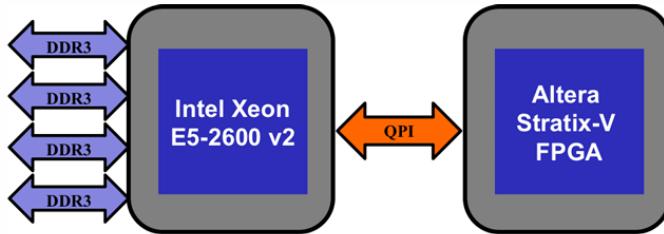
(just examples to better understand)

Slide 8

SoC is not Equal to Embedded System!

HARP (2015): 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

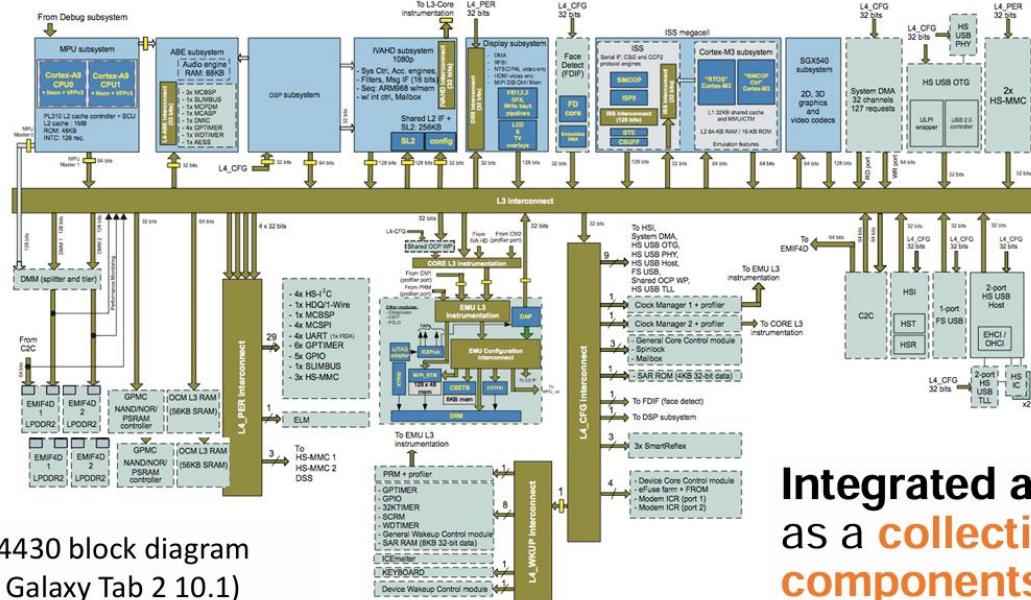


System-on-Chip defines an architecture where the components are connected at the same physical level

Slide 9

Today's SoC Architectures

Figure 1-2. OMAP4430 Block Diagram



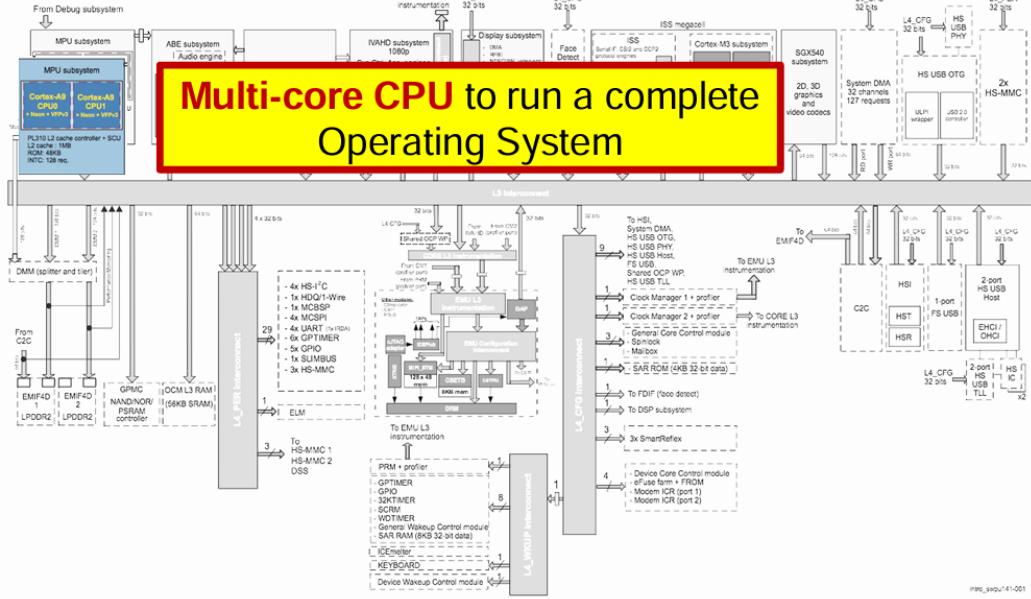
TI OMAP 4430 block diagram (Samsung Galaxy Tab 2 10.1)

Integrated architecture as a collection of components

Slide 10

Today's SoC Architectures

Figure 1-2. OMAP4430 Block Diagram

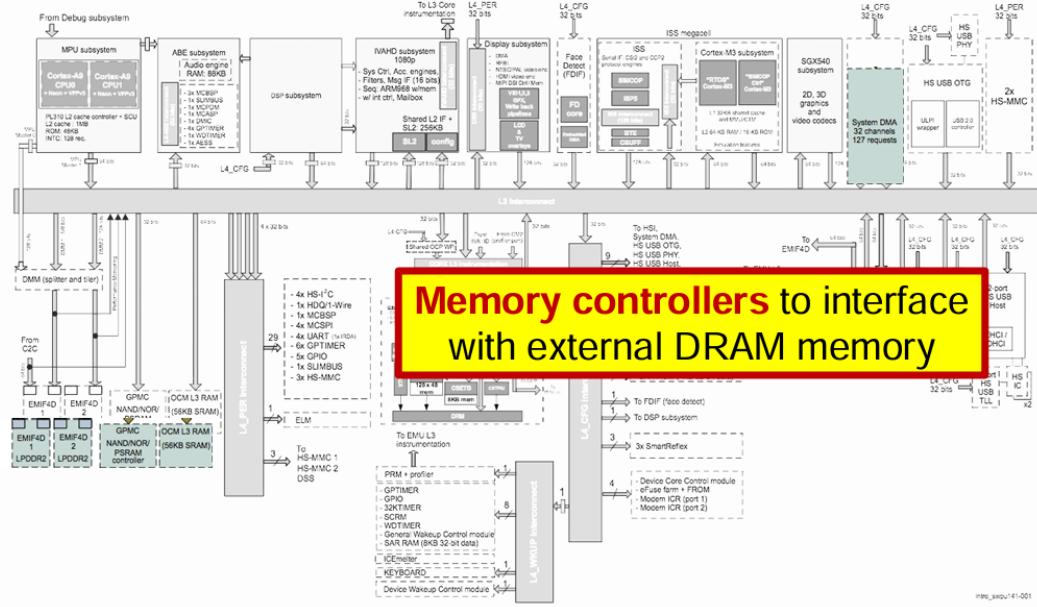


Multi-core CPU to run a complete Operating System

Slide 11

Today's SoC Architectures

Figure 1-2. OMAP4430 Block Diagram

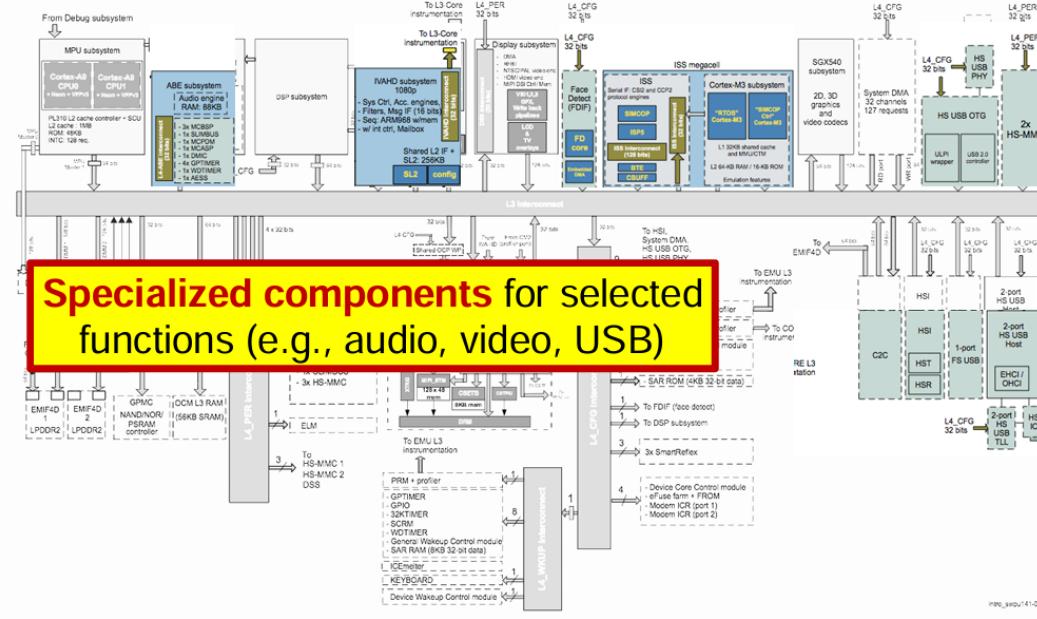


11

Slide 12

Today's SoC Architectures

Figure 1-2. OMAP4430 Block Diagram

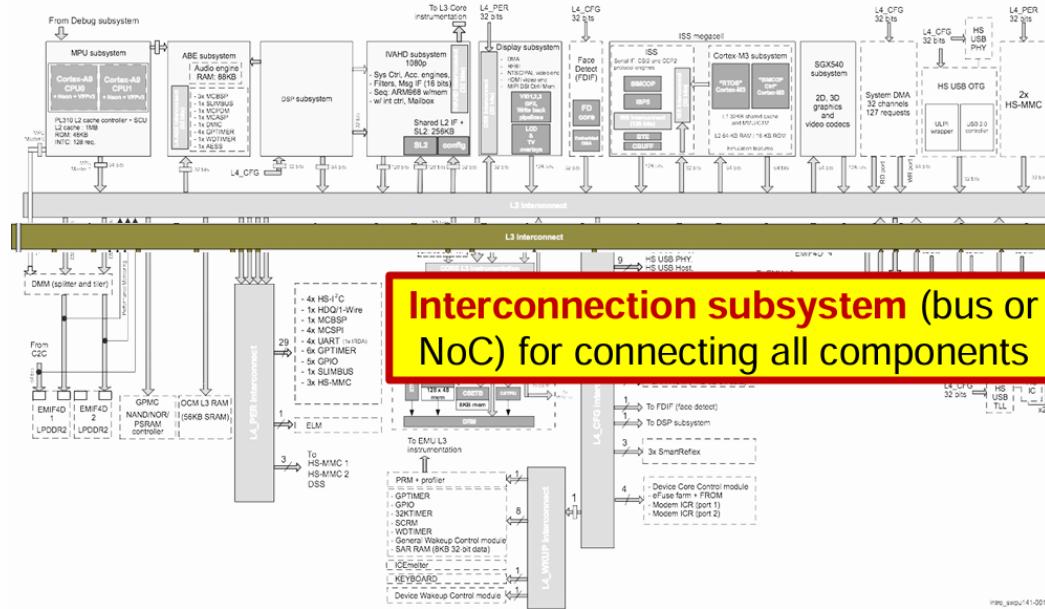


12

Slide 13

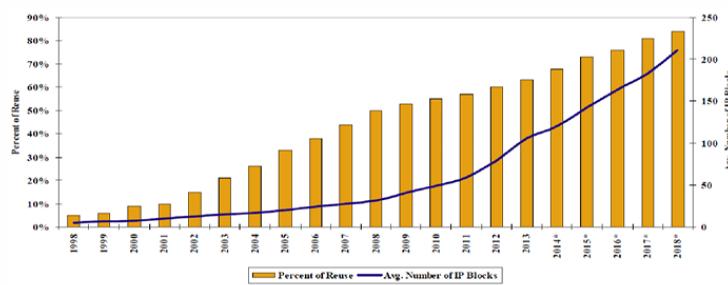
Today's SoC Architectures

Figure 1-2. OMAP4430 Block Diagram

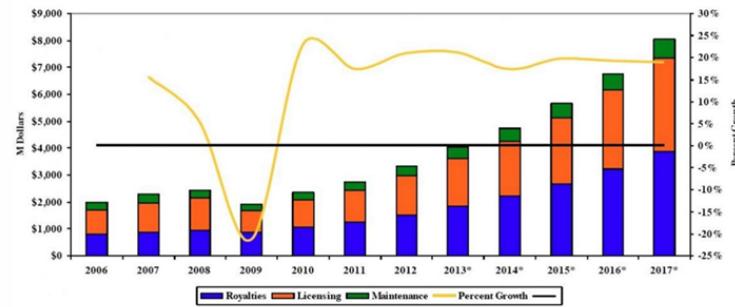


Slide 14

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)]



Slide 15

Platform-Based Design

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

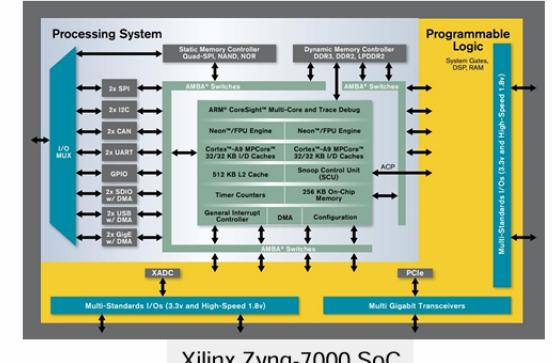
General-purpose processor(s)
(with Operating System)

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

Slide 16

Platform-Based Design

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

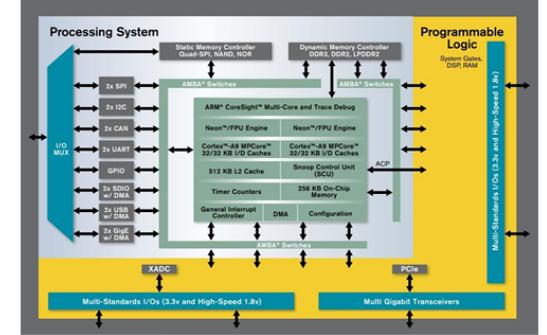
General-purpose processor(s)
(with Operating System)

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

When we talk about accelerators we mean GPUs, dedicated CPUs. The concept is that we're going to extend the core of the processor. To include accelerators within the core, we can include in the platform dedicated cores that are being developed starting from general purpose processors (ex. RISC-V for machine learning accelerator starting from GPs).

The SoC may include a configurable area or a dedicated IP core, so that is dedicated to a single task.

Important: More efficiently doesn't necessarily mean faster, this may mean less power consumption or less Silicon area.

Observation: in heterogenous system the OS has a big impact on the system, so they have a dedicated OS. It must know which are the hardware accelerators that are in the system, which are the functionalities that are necessary in the SoC. The processing platforms are operated by the OS.

Observation: is the software going to be developed before or after the hardware? It depends on which software we're considering, if it is a user ended software (ex: Telegram, WhatsApp...) are built and implemented after the hardware realization, but if we mean the firmware it is running between the HSoC and the OS, so the basic functionalities are provided by the hardware designer.

firmware definition: firmware is the low-level software that controls the interaction and behavior of a piece of hardware or IP-core. [\[source\]](#)

Def. Driver: piece of software that teaches to the general-purpose OS how to communicate/work with the specific system I'm using (ex: driver for a scanner).

Slide 17

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)

On chip memories: the memory on the SoC.

By a theoretical standpoint we can define three approaches to the use of the memory:

- **Cache memory:** it's the closest to the CPU and is managed or by *proximity* or by *locality principle*. Remember: locality principle, the data that is stored loaded is the one accessed very frequently or the last data that has been lately accessed
Remember: proximity principle, the data that is loaded in the memory is the one that is located near the one that I'm using.
The cache memory is managed by the CPU and is accessible via software.
- **Private local memory:** it's direct memory access (*DMA*) reserved for the accelerator.

If some computation can be accelerated, the CPU wakes up the accelerator, the data on which the computation has to be performed is loaded in the private local memory, the computation is executed via hardware and then the results are moved back to the general-purpose memory. It is not accessible by the software and is reserved for the accelerator. It's inside the accelerator and is fully managed by it, the core can't see anything of the PLM. PLM is usually big but we need synchronization.

- **Scratchpad:** the scratchpad is an approach in the middle between the *private local memory* and the *cache*. It is a memory in the accelerator but can also be accessed by the processor by executing some actions. When a functionality can be accelerated by the hardware, the processor offloads the computation (by executing some functions) and moves the data into the scratchpad. The scratchpad it's inside the accelerator, is accessed by it to read/write data, but this data is also accessible by the processor, that can read/write into the scratchpad. It's an extension of the memory space of the system. It's very flexible because data can be moved from the central memory but the price to pay is complexity of the system. In general, its dimension is thinner than the one of a PLM.

Off chip memory: “typical” memory

	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)

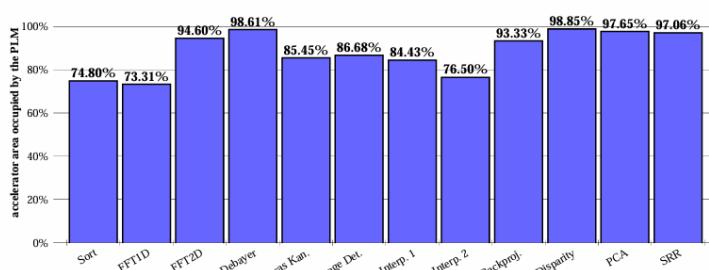
Static ram: volatile memory, much faster and takes only 6 transistors

Dynamic ram: volatile memory, basic RC network which has to be refreshed due to leakage.

Slide 18

Main Memory and PLM: A Huge Gap

Bench.	Main Mem Data Size (MB)	PLM Data Structures (#)	PLM Data Structures (MB)	Bench.	Main Mem Data Size (MB)	PLM Data Structures (#)	PLM Data Structures (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	Diparity	15.820	11	0.146
PCA	20.190	3	0.117	SRR	4.760	21	0.076



Each accelerator is ~1mm²

- Comparable to Apple A8 accelerators

PLM is from 75% to 98% of accelerator area

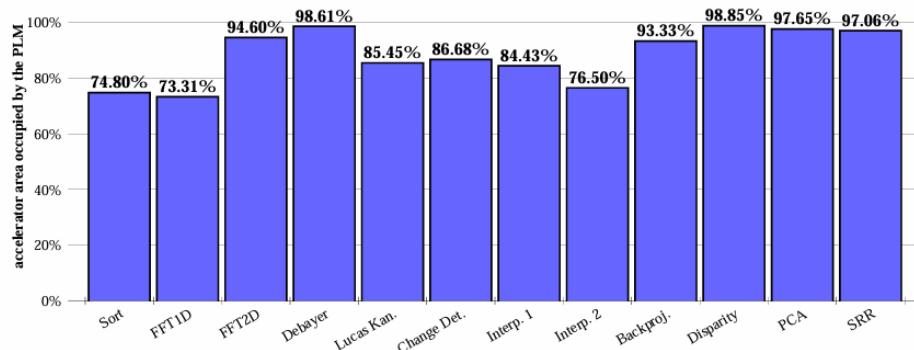
- With a lot of **data transfers**

This slide shows the implementation of 12 different algorithms and for each one of them a dedicated accelerator has been implemented. Then a constraint on the chip area was set. The process is optimized with high level synthesis and the remaining area is filled with private local memory.

Main Mem Data Size (MB)	PLM Data (#)
0.250	10
16.000	4
320.000	10
64.010	7
15.820	11
4.760	21

data required by computation

number of blocks what could be implemented



% of area on chip dedicated for private local memory. We do realize by looking at the results that even if most of the silicon area was exclusively dedicated to the private local memory, the memory implemented is negligible with respect to the memory necessary to perform the computation.

Bench.	Main Mem Data Size (MB)	PLM Data Structures (#)	(MB)
FFT1D	0.250	10	0.040
Debayer	16.000	4	0.096
Change Det.	320.000	10	0.062
Interp. 2	64.010	7	0.640
Disparity	15.820	11	0.146
SRR	4.760	21	0.076

necessary
memory implemented
memory

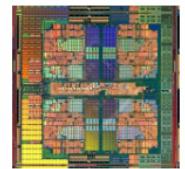
this means even if we optimize as much as we can the accelerator we can't avoid using off chip memory so it's crucial to optimize communication between the accelerator and the off chip memory.

Slide 19

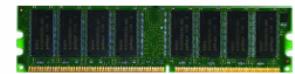
Memory Cost

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., Phase-change memory, Magneto resistive memory, in memory computing) enable new opportunities: **memory+storage merging**

Redesign of the memory hierarchy with application-centric approach

Slide 20

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**

We can have different approaches to interconnections

- Bus: busses are cheap, standard but slower and they do not scale with increasing dimension, this means that as we introduce more devices we've slower busses because more parasitic capacitances are introduced. It is a physical and logical bottleneck.

- NoC: NoC are dedicated routing memories, they're much faster than busses but are more complicated and less standardized.

(there will be a dedicated lesson to busses and NoC)

Slides 21-25

Examples about some commercial/research SoC platforms.

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>



The NEORV32 RISC-V Processor

Stephan Nolting, M.Sc.

<https://github.com/stnolting/neorv32>

<https://stnolting.github.io/neorv32/>



21

Embedded Scalable Platform (ESP)

Architecture and Design Methodology

• Regularity

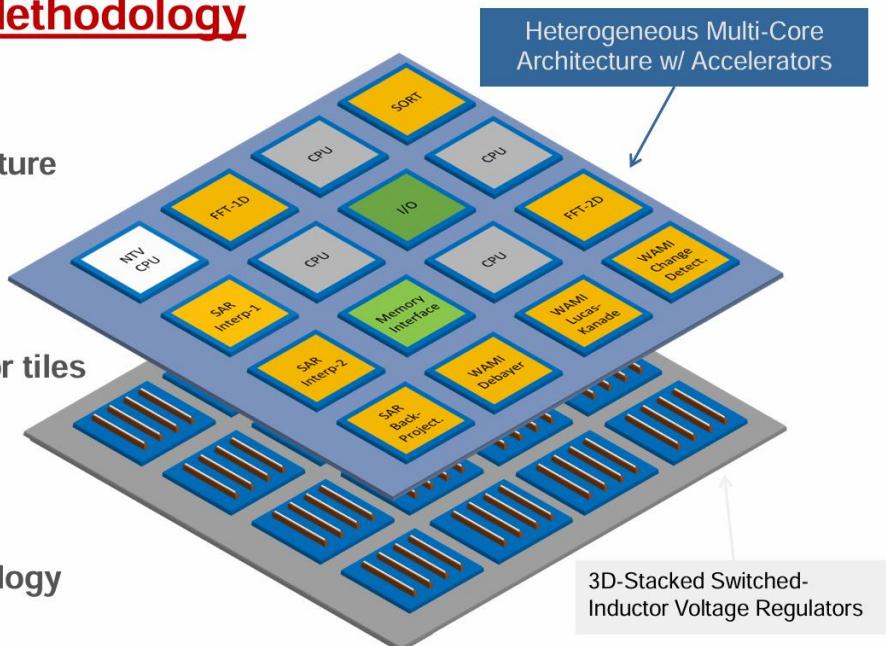
- tile-based design
- pre-designed **on-chip infrastructure** for communication and resource management

• 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**

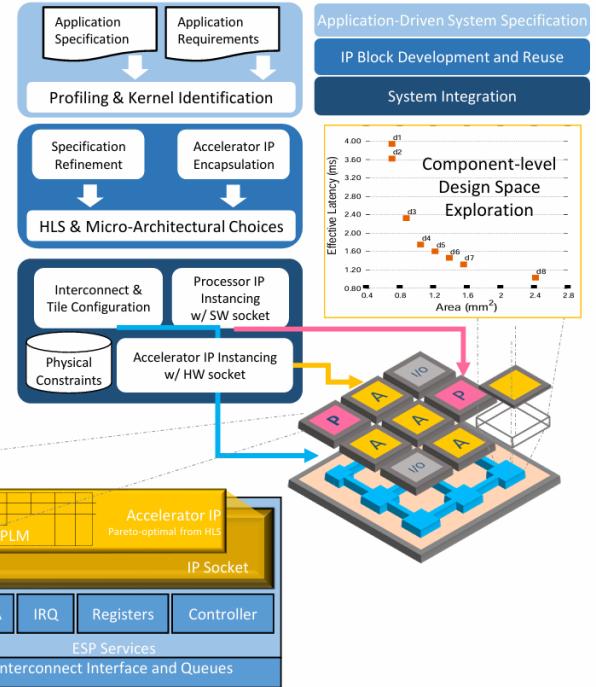


23

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

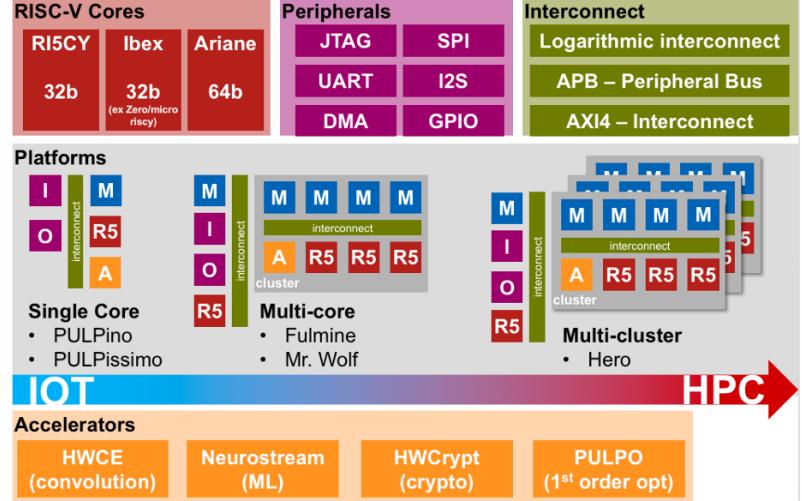
Ultra-low power system based on RISC-V

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

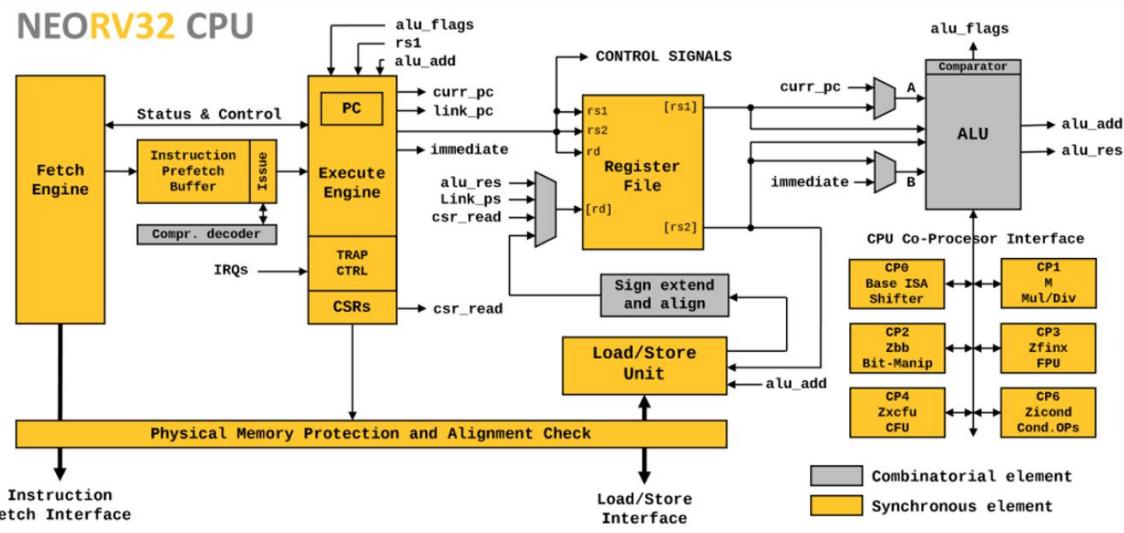
Complete platform integrating several research projects

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

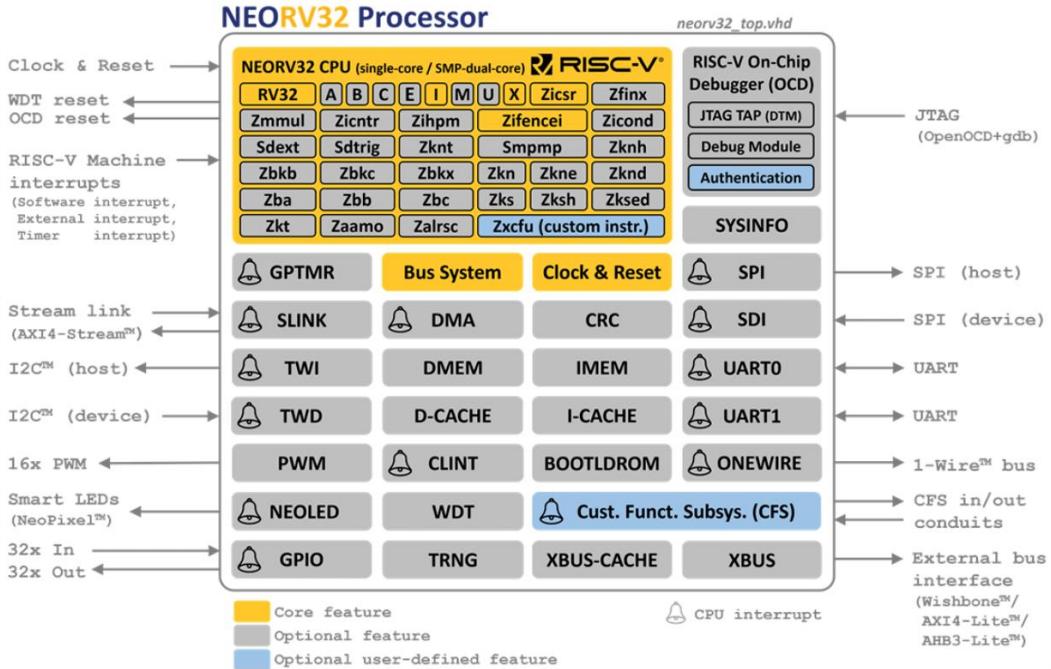
First complete
open hardware project



The NEORV32 RISC-V Processor



The NEORV32 RISC-V Processor

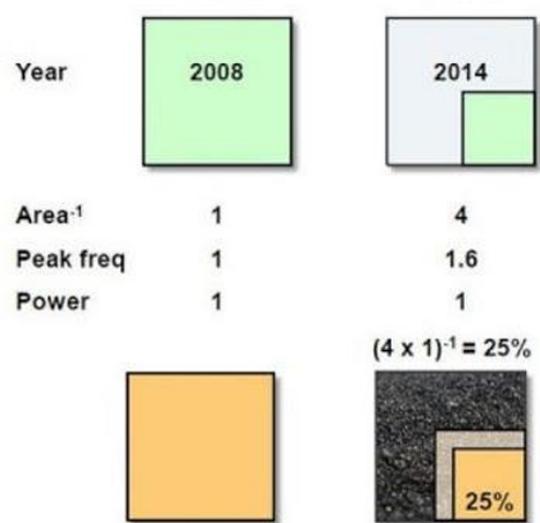


This is the device that is going to be used in this course, we've several implementations and we can combine them to quickly write a working SoC.

2 - SoC components

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 **fast and efficient way to turn on and off** components
 - Requires the definition of a **scalable architecture** and a **communication infrastructure**
 - Requires also an **efficient power management system**



Remember: by “Dark Silicon” we mean a portion of the silicon device that must be underclocked/turned off because it is not performing any significative action/computation but still would use a significative amount of power if it wasn’t turned off.

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 modify

General purpose hardware can do different tasks

Many applications share the **same (critical) kernels**
Specialization is applied only to those kernels

General Purpose Processors (GPPs) executes the rest of the code

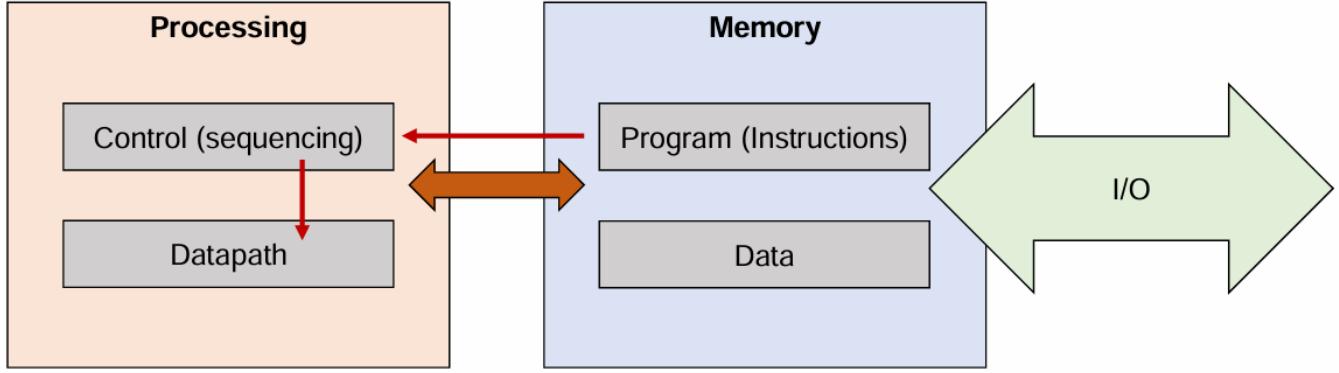
GPP allows for the **execution of different applications**

HSoCs, not custom chips!

A processor in a system on chip is necessary because we can't afford all the computation to be hardwired in Silicon, we need some architectural synchronization for scheduling, coordinating the operations, communication and collecting the results. Such tasks are given to the processor.

What is a Processor?

Also called **stored program computer**



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

In this slide we have some basic concepts:

- What is the *CPU – General Purpose Processor*?

It's a Silicon device that must perform five basic tasks:

1. *Fetching* instructions
2. *Decoding* instructions

3. Execute instructions
4. Writeback the results from the processor registers to the cache/main memory

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

Problem
Algorithm
Programming Language
Runtime System (OS)
Architecture
Microarchitecture
Logic
Circuits
Electrons

The architecture is a bridge between the high-level part of the system, so the OS, and the low-level part of the system, so the microarchitecture. When the compiler must convert a program, it has to know on which architecture it will be run and that is defined by the ISA.

What is the instruction set?

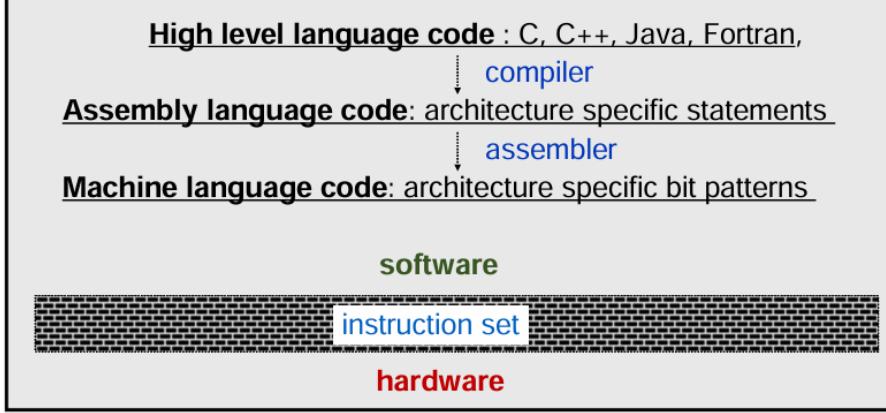
The instruction set are all the instructions that are provided by the architecture.

Ex: Let's suppose that we have a particular architecture that does not provide the store instruction, then all the programs that are compiled for that architecture can't run the store instruction.

Basically the instructions are the options that are available, but the ISA doesn't set constraints on *how* the instructions are implemented in hardware, that is part of the microarchitecture.

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**

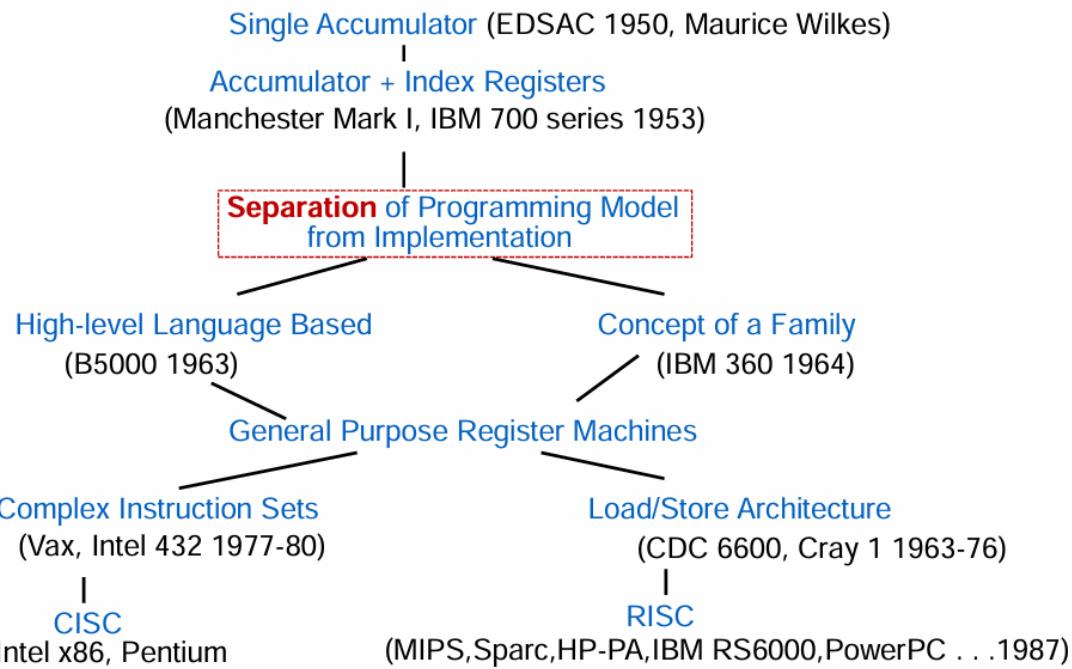
- How the underlying implementation executes instructions
- So, several specific processors with differing microarchitectures may share the same architecture, i.e., the same ISA

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

When compiling a program for a specific family of processors the same instructions are always exploited because the ISA is the same, what differs is the *microarchitecture that implements in different ways the instructions*. Consistency has to be guaranteed by the microarchitecture, because if I use a *store* I expect the *store* to have always the same effect. The architecture sets the constraints from which I build my microarchitecture.

A Bit of History in Computer Programming



CISC: from a market point of view, CISCs dominated world market because AMD and Intel used this category of ISAs and as of today it still is the factory standard for consumer microprocessors.

RISC: is the standard for embedded systems since the 80s, small/medium companies used RISC, then ARM came and so the implementation of RISC.

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)
- tens of addressing modes (e.g. [memory + memory/offset + reg + immediate])

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)
- ≈ single addressing mode: [reg + immediate]

CISC: thousands of instructions, single instruction for every single way we want to have operands etc. Lot of instructions means that *each instruction has its own circuit*.

Complex instructions → complex circuits

So generally, CISC systems are faster but much power hungry and complex.

RISC: hundreds of instructions, *load/store* instructions so all the operands are loaded/stored from the memory. Much fewer instructions, single addressing mode, single addressing for operands, slightly slower but much more optimized for embedded systems, much less power hungry.

Important point:

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

Most of the computations are made by a little part of the ISA, so generally a small set of instructions are used.

What is Better?

Complex Instruction Set Computer (CISC)

[e.g., Intel x86, AMD]

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, SPARC, ARM, RISC-V]

Fewer operands and less complex (efficient?) instructions

Faster fetch/execution of instructions

More instructions per program

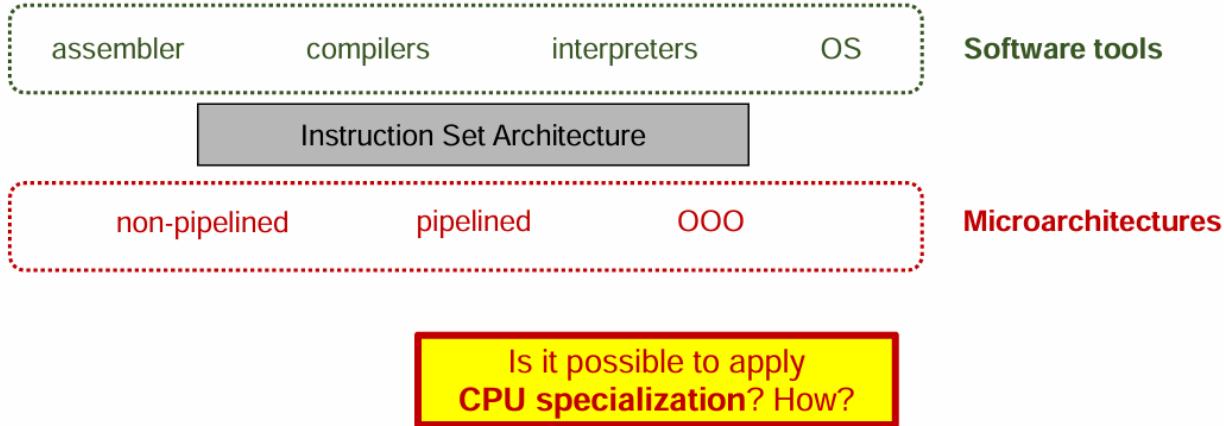
CISC: *thousands of instructions, many ways to access memory, but even with thousands of instructions we use not so many instructions*

RISC: *hundreds of instructions, one way to access memory (both cache and central memory), much less power hungry devices*

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**



The ISA is the interface between software and microarchitecture. On the top of the slide in green we have the software ecosystem that has to be implemented to design and exploit a processor, we need all the tools for such platform, as well as compilers/interpreters to traduce the high-level code to the ISA. On the bottom of the slide, in red we do have the microarchitecture that is how the ISA is implemented, if we have pipelines/caches/branch prediction features, is all conceptually below the ISA.

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**

As we said, since AMD/Intel dominated the market, nobody tried to modify or change how the implementation of the instructions of the ISA was done, there was no real possibility of implementing new processors unless someone would introduce a new ISA. Obviously there is no way to optimize Intel/AMD implemented processors from outside.

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 changed the paradigm since it is an open source ISA and based on it we can implement and redesign the whole processors, we have a standard ISA and we can use it to build freely our specific processors with the features we want to enhance. This is a totally different approach from fixed and proprietary ISAs.

RISC-V Popularity

2017



2019

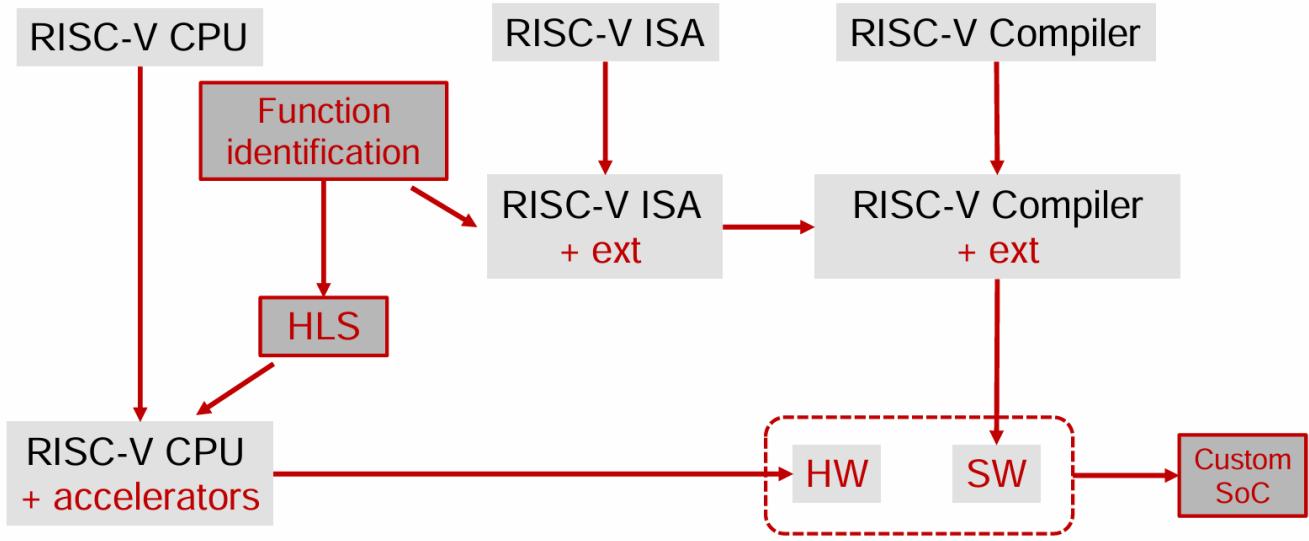


364 members
in 2024

In this slide we have a list of the supporting companies/institutions that produced their own processors using RISC-V. As we can see, also silicon companies started to use RISC-V, that is because it is interesting from a Si and from a microarchitecture point of view, we can implement accelerators with their own processor, which is a great potentiality.

Custom SoCs with RISC-V

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



With RISC-V Instead of implementing a dedicated accelerator out of the core that must then communicate with the core (*loosely coupled accelerators*), we can directly implement the accelerator inside the core (*tightly coupled accelerator*) and add a new instruction for the accelerator with its own additional circuitry and I can do this because it's an open source code.

Ex: I have “add”, “mul” and add “MAC”, I introduce the new instruction in the ISA and then I need to inform the compiler for that specific architecture that that instruction exist, otherwise it can't be implemented in the assembly code. When “there's this pattern of code, this is a MAC, this is not anymore jumps and multiplication but it is a single call to the MAC” and then I have to implement the MAC in the core, I modify the documentation of the code but then I also need the high level syntheses/Verilog implementation into the core. So we do extend not only the ISA but also the core itself, we will have additional circuitry that implements the instruction.

At the end we obtain a modified RISCV with modified core and modified documentation and modified compiler that has been extended with our new specific functionality.

The idea is not to totally substitute the existing processors but it's more of implementing dedicated very small accelerators for specific instructions (ex: META/AWS are leaving the control to std processors that are very optimized/very fast but then implementing clusters of RISC-V accelerators dedicated to deep learning operations, instead of having power hungry and very complex systems).

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

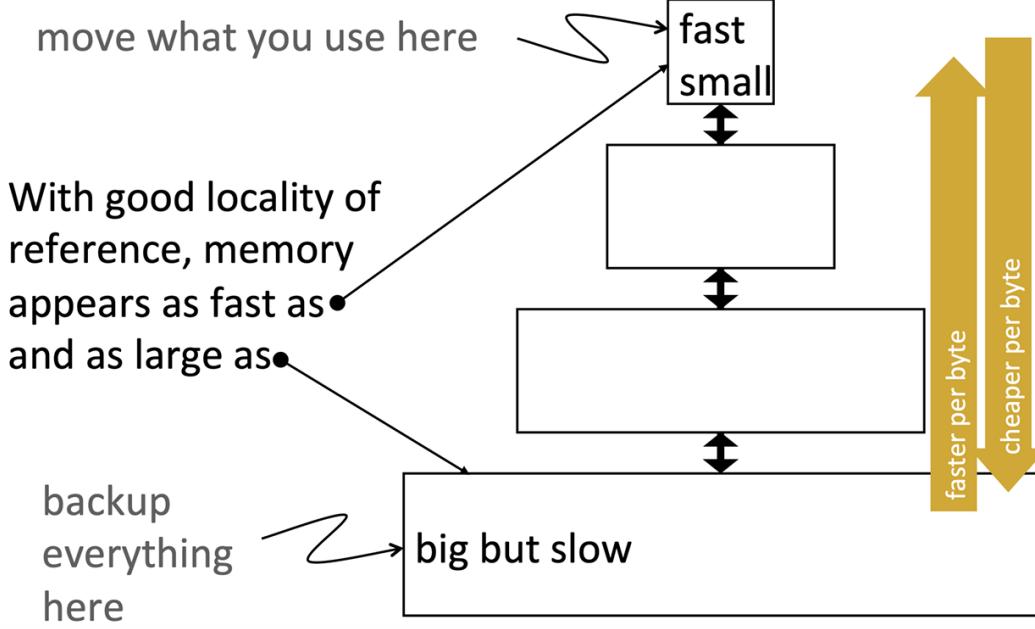
(progressively bigger and slower when far from the processor) and ensure most of the processor data is kept in the fast(er) level(s)

What is bandwidth? Is the number of parallel access that accelerators can do to the memory.

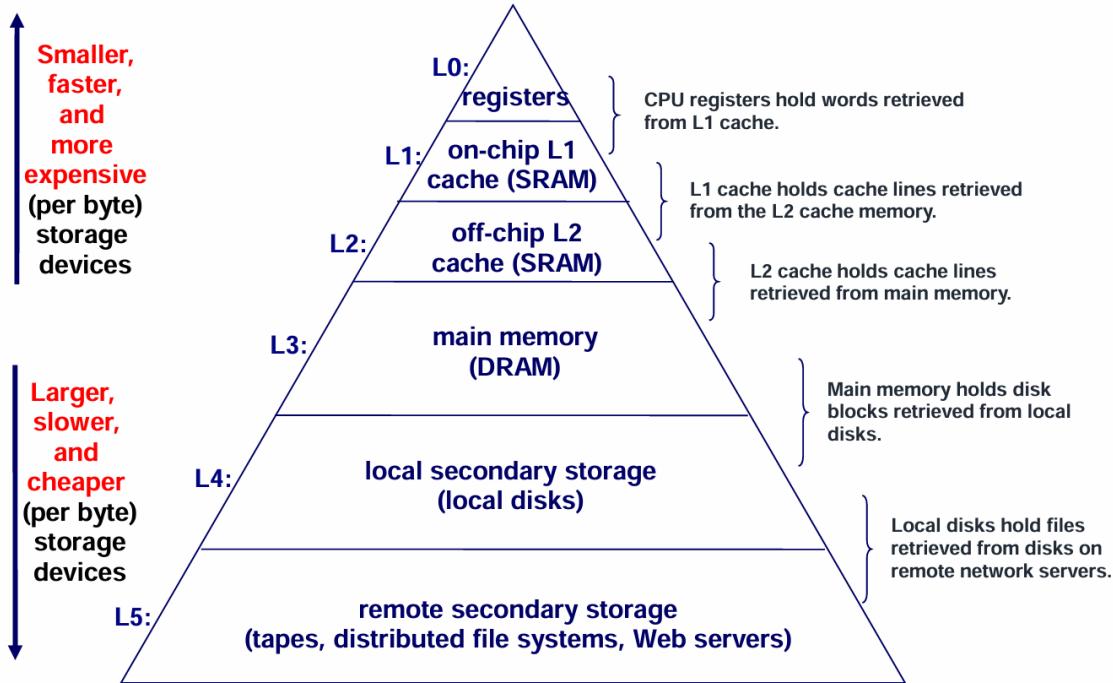
As exposed in the bottom part of the slide, since there are different technologies for memories, the general idea is exploiting such different technologies to highlight the advantages of each of them and try to reduce the limits given by the cons.

Specifically for the accelerators, the important thing is adapting the way of accessing and the dimension of the memory for the single task that the accelerator has to perform.

Concept of Memory Hierarchy



An Example of Memory Hierarchy



Classical hierarchy of memory

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?

An accelerator can be implemented in a dual core way, splitting the computation in parallel executions. The problem with this approach is memory coherence and that is in any case that two entities are sharing the memory and running in parallel.

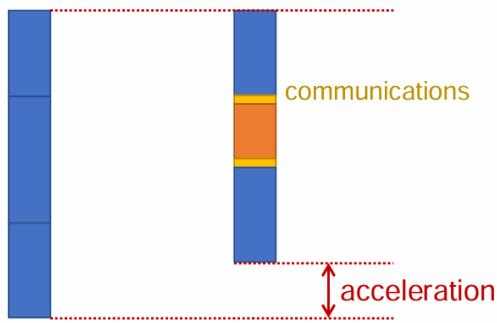
Ex: two cores sharing the main memory but with dedicated caches, L1 caches for both. Both cores load a piece of data, then one starts modifying the cache, but the other core does not see the modification, because caches are not shared, therefore readback the memory to the central memory is not enough to keep coherence, because the other cache is still not modified, it has to be updated.

Parallel programming/distributed programming is quite complex and introduces a series of problems that might be very difficult to overcome.

Heterogeneous Processing Elements

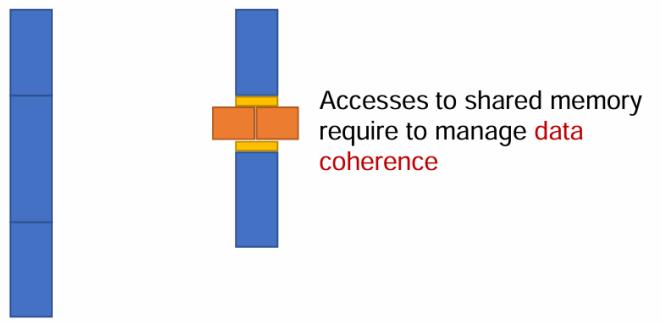
Offloading Execution

- 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)

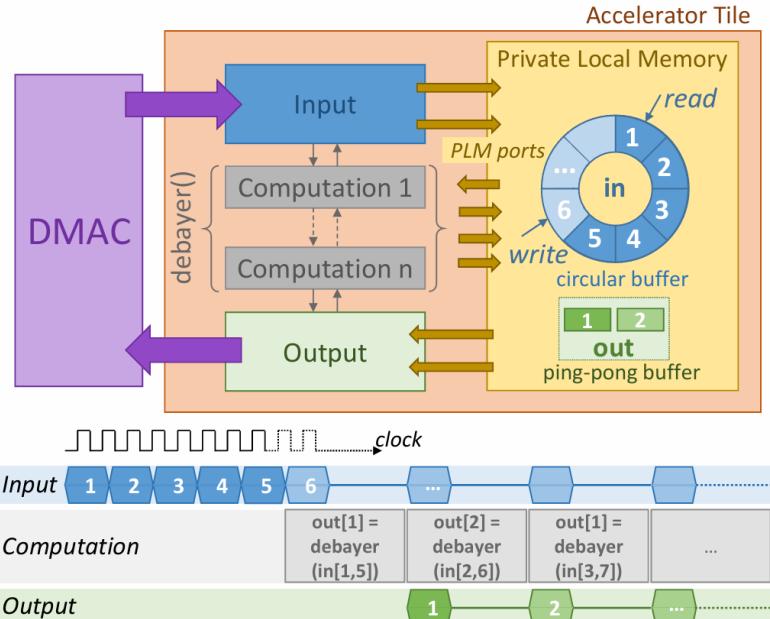
In this slide we have the representation of the two different approaches, parallelization or hardware acceleration.

By *offloading execution* we mean that the execution is flooded out from the software to dedicated hardware, if the system is well designed it will take less time or less power (or whatever constraint we want to improve).

In the slide we can see some “yellow” parts, these are the overheads, the wasted resources to wake up the accelerator, load/unload the data etc. we might implement a very optimized accelerator, much faster than software execution or much less power hungry, but if (for example) the communication is not implemented well, it could make not worth the using the accelerator. There can be various bottlenecks as maybe the amount of data is so large that we lose too much time/power to accelerate the functionality.

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**



Example of a possible *loosely coupled accelerator*, we can identify

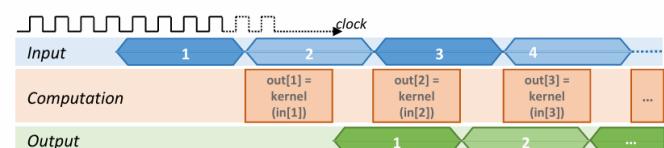
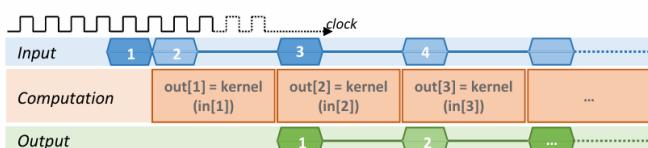
- in purple: the DMA controller
- blue/green: input/output buffers
- grey: accelerator device and computation for acceleration
- yellow: private local memory implementation

The slide also introduces an important concept, the *communication optimization*: it is transaction level design, I'm not interested in what the accelerator implements but I want to synchronize the “blue boxes” with “grey boxes”, so the data loading and the computation.

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



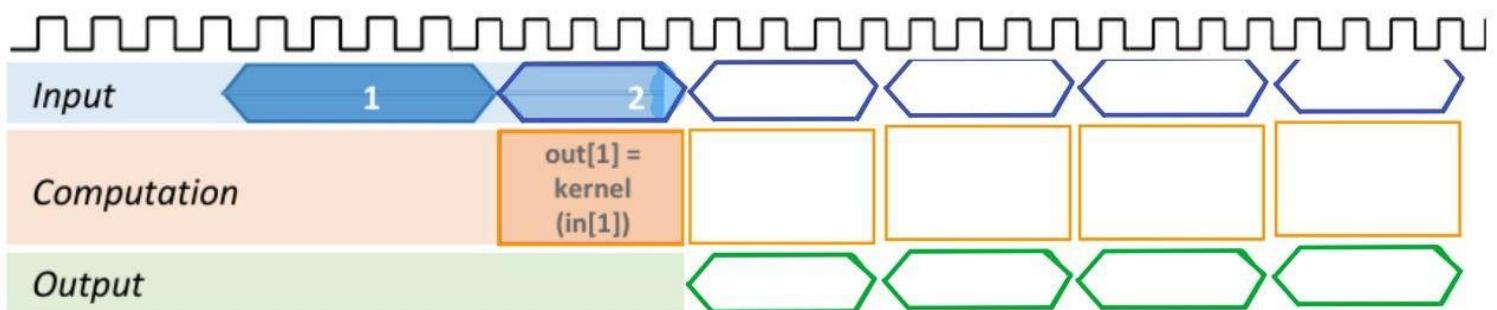
Reduce the congestion or exploit the congestion
to optimize the execution at the system level

This slide is very important because it represents two possible situations in which communication and computation were not balanced, thus reducing the performance of the accelerator. In detail

- left graph: fast communication but slow computation. If I implement a very fast communication but I have a slow computation, probably I am wasting some resources somewhere, whether it being area, power, silicon, because implementing fast communication can be very costly and power hungry, but if I am limited by the computation, I am wasting all the resources I allocated to the communication.
- right graph: fast computation but slow communication. In this case probably we should implement better communication, otherwise we spend silicon area and power to make a fast accelerator, but its potential is wasted due to a bottleneck of communication.

I have to be as consistent as possible, since I'm designing an accelerator, I have to keep everything in mind to have a really optimized system, is not enough optimize computation or communication, both have to be balanced and I have to know where the bottleneck is.

The ideal situation would be



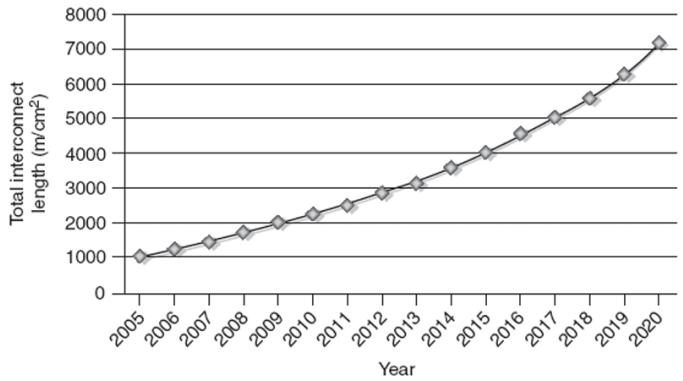
SoC Communication

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



Communication is the biggest issue in SoC design, processors are coming from outside, the VHDL/Verilog for the accelerator is more or less easy, the biggest issue is *communication and synchronization* among all the components of the SoC. If we are shrinking the size of transistors and silicon components obviously the communication/wiring part is increasing, so we must try to optimize it as much as possible.

Bus Terminology

- **Master (or Initiator)**

- IP component that initiates a read or write data transfer

- **Slave (or Target)**

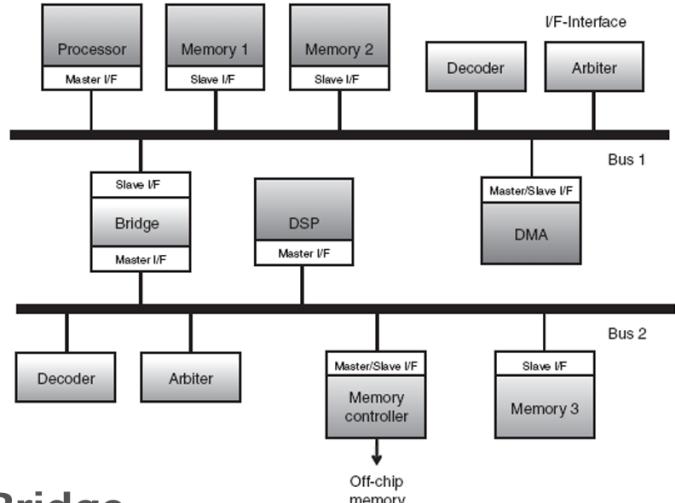
- IP component that only responds to incoming transfer requests

- **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 (😢)

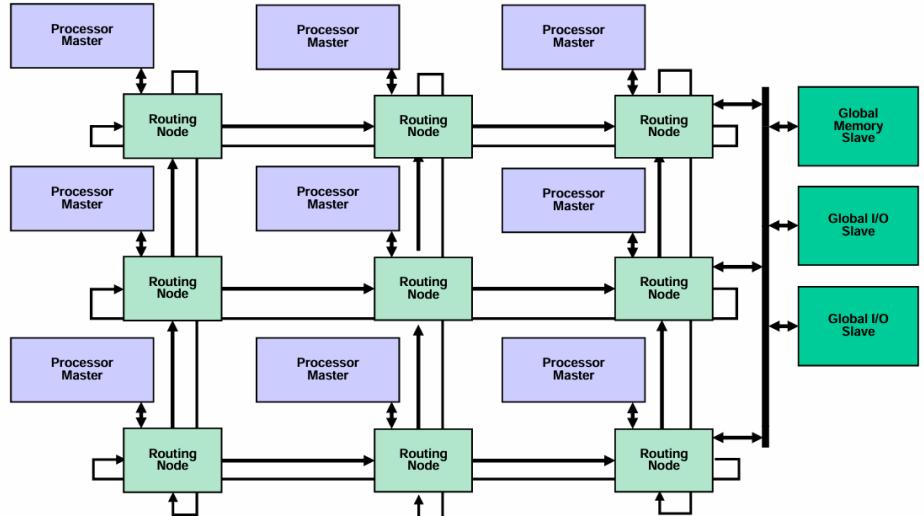
- 😊 The silicon cost of a bus is small.
- 😊 Any bus is almost directly compatible with most available IPs, including software running on CPUs.
- 😊 The concepts are simple and well understood.
- 😢 Every unit attached adds parasitic capacitance, therefore electrical performance degrades with growth.
- 😢 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.

The BUS advantages are that buses are cheap, have wide compatibility and are very simple. The limits are that buses do not scale, so as we increase the dimension of the communication network, we significantly slow down the device, the more components are connected the less time to propagate the signal we have.

Network-on-Chip (NoC)

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



NoC (Network on Chip): all the components of the SoC are part of a network and the connections between chips are the connections of a node of a traditional network. It can implement parallel communication (with respect to the BUS that is a shared device), we have reconfigurability in fact we can modify the path for the communication based on the network chart. The network can also be reconfigured due to failures.

The NoC router is the component that has to decide which path each master is going to communicate, for NoCs a *routing table* is implemented within the router.

NoC pros (😊) and cons (😢)

- 😊 Parallel communication.
- 😊 Reconfigurability.
- 😊 Possible shorter paths (and thus delays).
- 😢 More silicon.
- 😢 More wires.
- 😢 Possible longer delays if many routers have to be crossed.
- 😢 NoC optimization may require an ad-hoc placement of the accelerators.

As designers, we have to ask ourselves if we need parallel communications, how often data transfer are required, how fast do they have to be and make an appropriate choice.

Important point: NoC are *not commercially available*, so we have to implement our own interface and standards.

Ex: let's imagine we have the FFT accelerator and we have an AXI interface that allows us to connect it with an AXI bus. If I want to use it in an NoC network I do not have a pre prepared NoC, so I must implement my router, the interface to connect from AXI to my NoC router and network.

With NoC I have only point to point connection and it's highly customizable network, each router/master can be optimized and built in a different way.

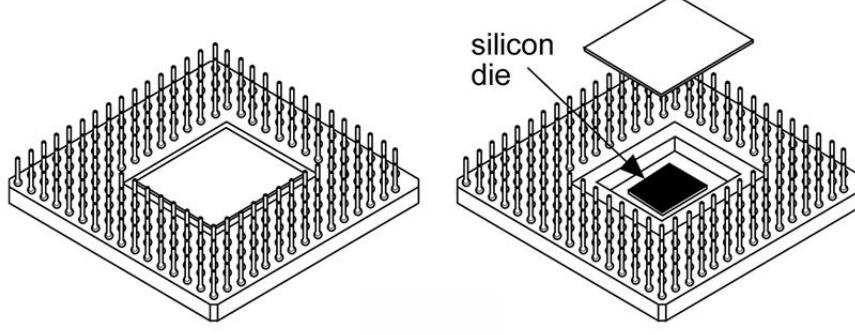
3 – Design flow

When referring to digital circuits as of today we mean VLSI. What are VLSI?

VLSI: Very Large Scaled Integrated circuits, that implies “lots” of transistors on a single chip. It is an acronym to identify all the digital technologies since the 90s up to today, although today we are talking about subscale or nanoscale devices and so we find the acronym ULSI, Ultra Large Scaled Integrated circuits.

ASIC (*Application Specific Integrated Circuit*)

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



Types of ASIC

- **Fixed-function ASICs**

- Full-Custom ASICs
- Standard-Cell-Based ASICs
- ...

- **Configurable Circuits**

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

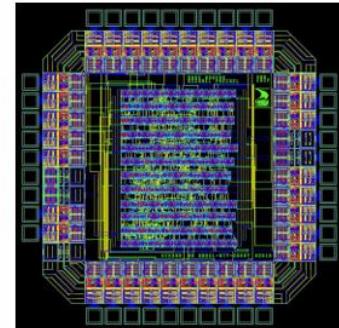
When we talk about ICs in general, we may see two divisions:

1. *Fixed function ASICs*: both the types reported in the slide still are used
2. *Configurable Circuits*: FPGAs are the only ones that are used, all the others were the old approach

Focus on Two Technologies

- **Fixed-function ASICs**

- Very **integrated**, yet very **expensive**



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

- **Cheaper** to implement, **reconfigurable**



Let's suppose we want to implement some function in a SoC. How can I do it? One approach is *full custom ASICs*

Full-Custom ASICs

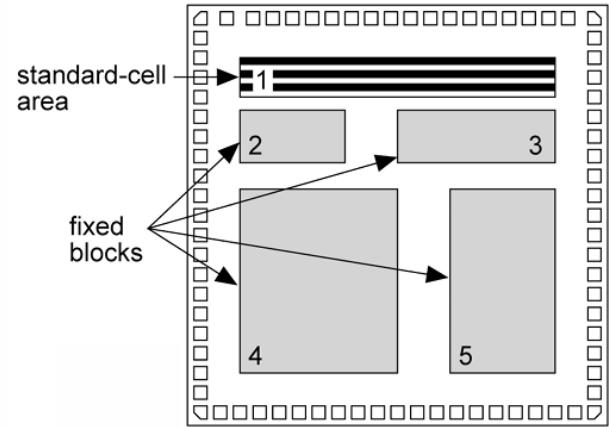
- Full-custom design offers the **highest performance** and lowest part cost (**smallest die size**) for a given design
 - All mask layers are customized
 - Generally, the designer lays out all cells by hand
 - Some automatic placement and routing may be done
 - Critical (timing) paths are usually laid 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!

The first approach was *full custom design*, so we consider having an area budget based on the requirements and we have to fill such area with transistors in order to implement the function fully from scratch, by choosing the dimension of each single transistor. The digital designer is the one that places each component in the Silicon area, design the circuit, produce the netlist and shape the masks for lithography.

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



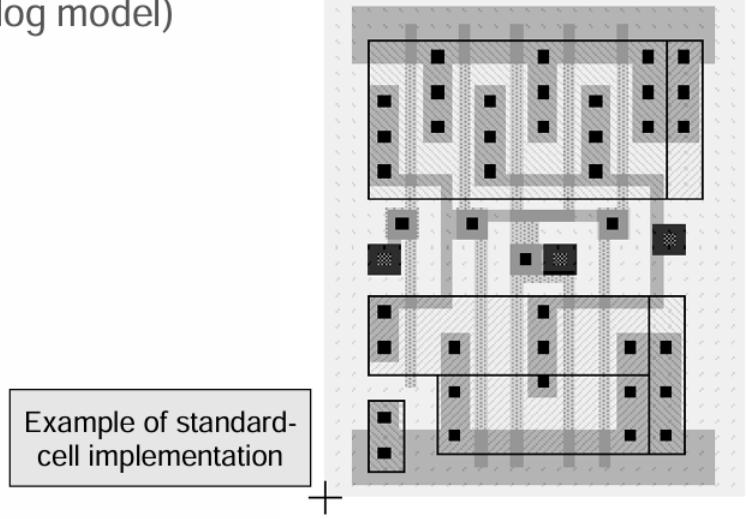
Since the full custom was too long and too expensive to implement, a second way to produce ASICs and VLSI was born by using *standard cells*, so basic pieces of circuits that can be exploited to build the device. Instead of fully designing each gate etc. a company provides standard cells and with them we build our system. There's less control over the design, less optimization, more area, more power consumption, lower working frequency but much faster, cheaper design.

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) 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
 - (2) You can build your own cell library (**application-specific cell libraries**)
 - Difficult and costly

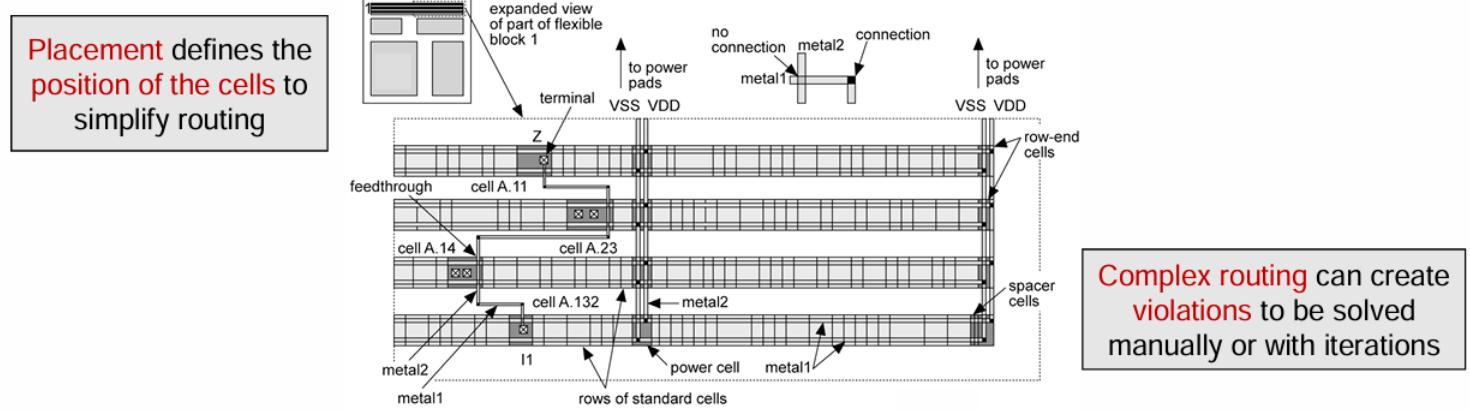
ASIC Library Development

- 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 wire-load model
 - A routing model



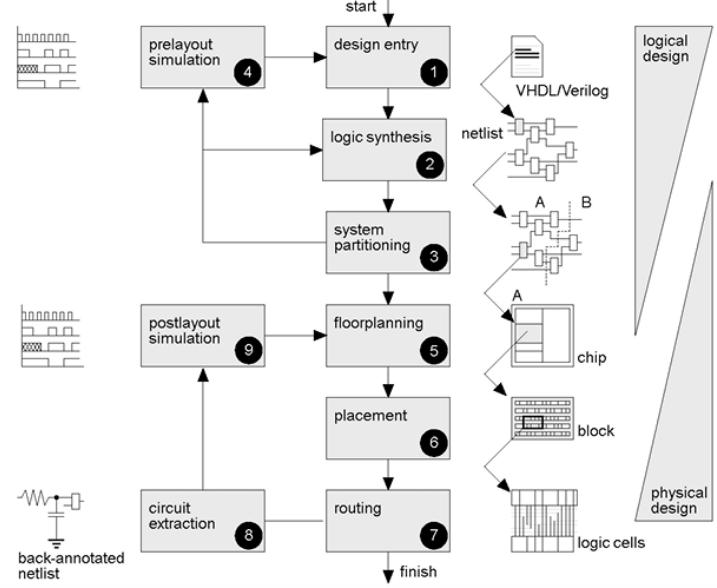
Standard Cell ASIC Routing

- 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



Design 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



This is the standard design flow of an ASIC design. We start with the specification, then we describe the behavior of the circuit, then simulations are done to verify that the function is implemented correctly. We still don't know if the circuit is working properly because since we haven't implemented it in silicon, we don't know if the length of the wires etc. aren't breaking timing constraints.

The next phase is “floorplanning”, here the circuit is divided in specific parts where each one has a specific function and we do *placing and routing*, so we place the specific working area in a portion of the device and then we connect them with real wires, in order to respect timing constraints. By knowing length and types of wires we know the parasitic capacitances and we can run the simulation with real delays.

So we have *logical design* and so behavioral description, structural description and logic netlist. Once we floorplan, place and rout, we extract the capacitances and all the electrical characteristics of the physical design. We can divide this operation in two parts: *frontend design* and *backend design*

- *frontend* is generally done by digital designers/computer engineers
- *backend* design is done by electronics engineers specialized in silicon physical design

Once we have the physical design, we know the electrical characteristics of the circuit and through post layout simulation we know *all* the characteristics like power consumption, maximum frequency etc.

The missing part in this flow is the *choice of the standard cells*, but this implies that the logic design is technology independent, we can achieve our objective without knowing the technology we are going to implement the circuit in because we're designing “just” from a logic point of view.

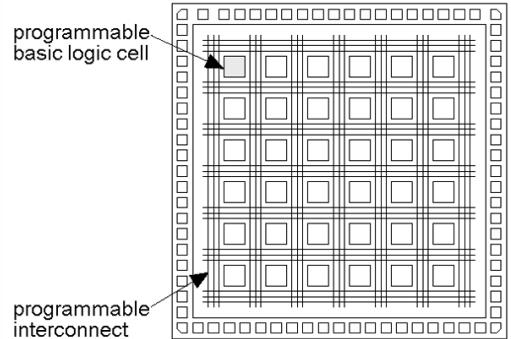
The opposite approach is *configurable and reconfigurable circuits*, we buy a chip with something already implemented inside and then we configure the chip, that is the case of FPGAs.

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

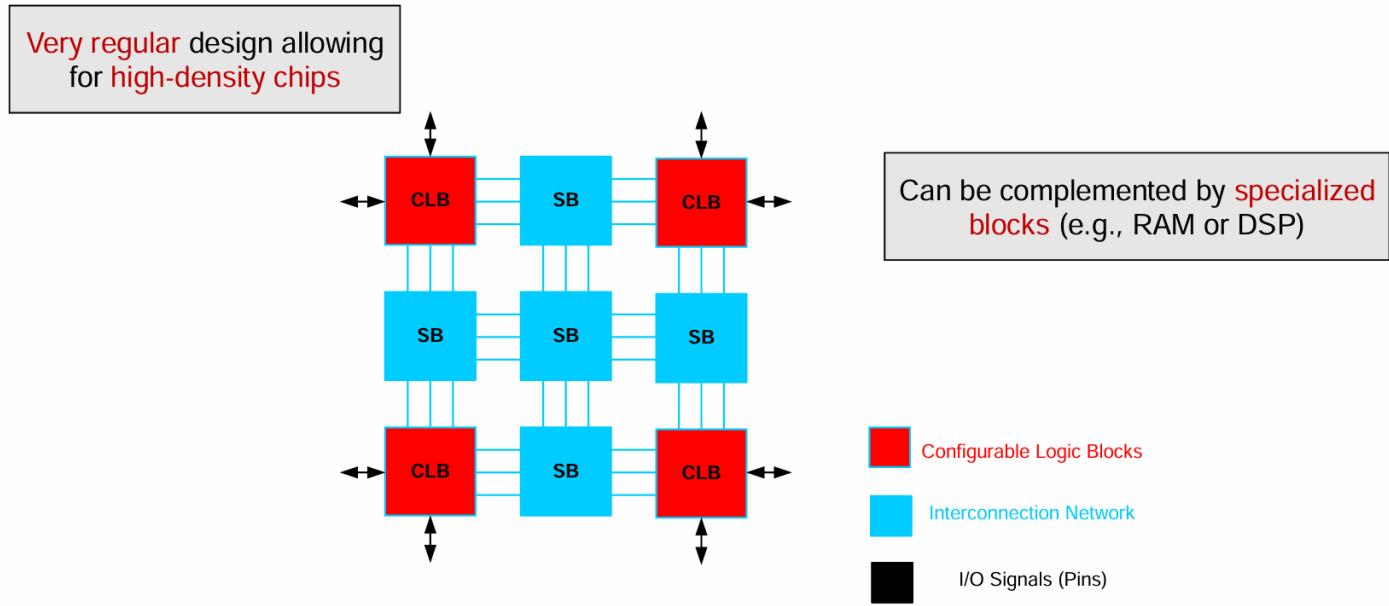
FPGA (Field Programmable Gate Array)

- 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



What is an FPGA? It is a mesh of configurable routing blocks and Configurable Logic Blocks. A CLB is a LUT plus a MUX plus a FF. The output of the CLB can be synchronous or asynchronous, the LUT implements our functionality.

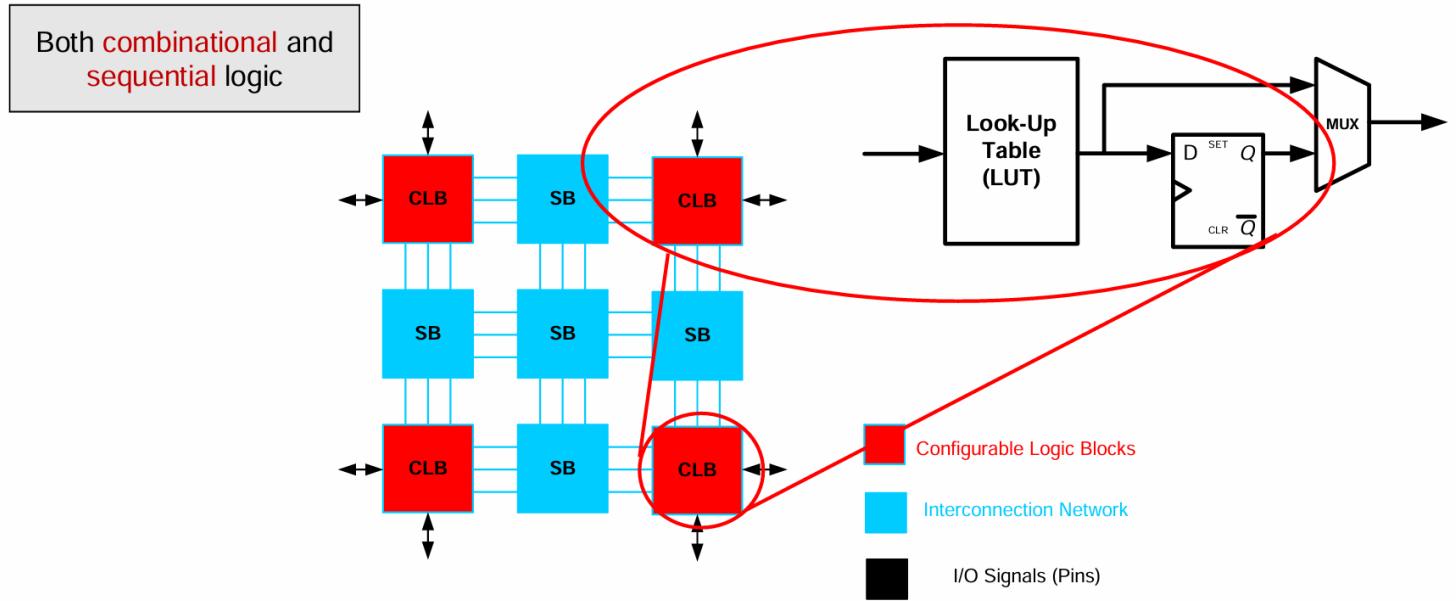
FPGA Structure



Let's consider the LUT as a piece of memory. The addresses of the LUT decide which is the row of the LUT that we're going to address, the content of the cell decides what is the output, it's a small memory, 4 bit I/O (*depends on the specific FPGA characteristics*).

If I consider the content of the memory as *output of a logical function* and the *addresses associated with the memory as the input of the logical function*, I'm *implementing the logical function with a piece of memory*. This is the role of the look up table.

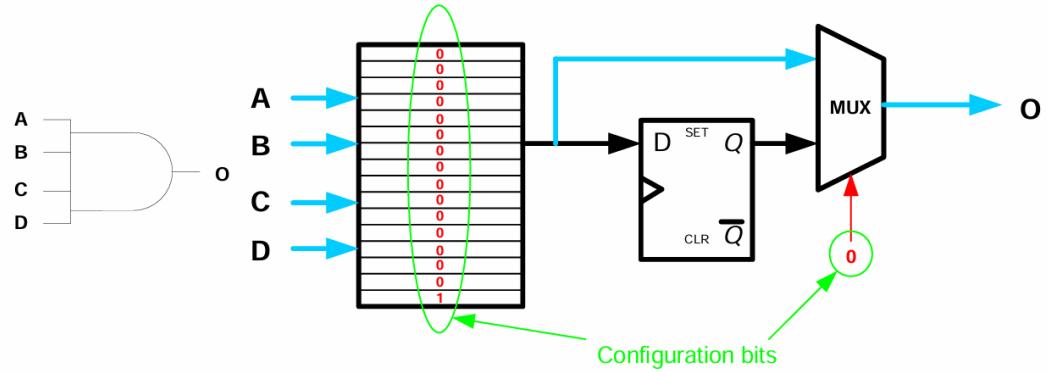
Simplified CLB Structure



Here a 4 input AND gate is implemented with a LUT.

Example: 4-input AND gate

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



If the LUT is the basic building block in the design, what do I need to do to configure it, to specify the functionality?

I need to configure the content of the memory. Based on the content of the memory I'm changing the functionality of the LUT. The content of the memory is called *configuration bitstream*, it specifies the configuration for the LUTs, so their functionality.

So we have a device with a standard layout and based on the configuration memory I modify the routing etc.

Of course the complexity of the circuit I can implement depends on the number of LUTs that I have in my FPGA.

What is the advantage of an FPGA based design?

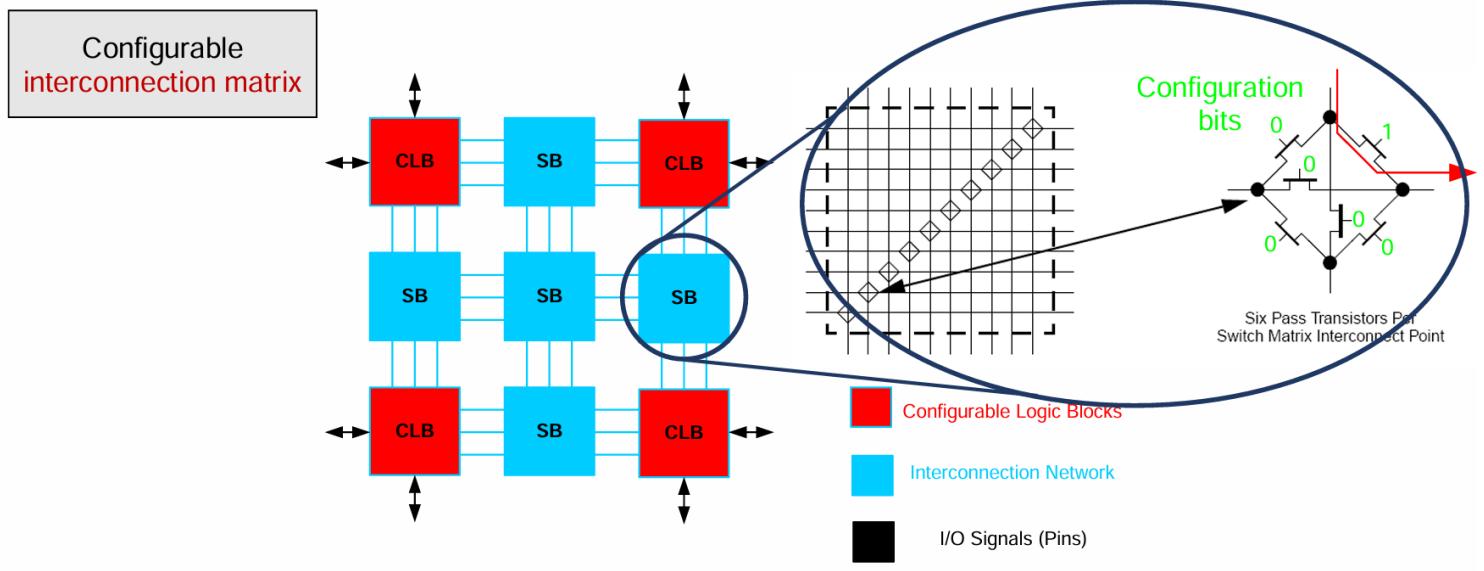
Speed in terms of time-to-market, I don't have to care about the physical design, it is already there, from the producer of the piece of silicon it is an easy design. From the producer's point of view they're implementing an ASIC, from the system architecture the FPGA is a configurable device. So, it's much simpler to design a system based on the FPGA, also much cheaper, because if I do something wrong, I can reconfigure the device, if I do something wrong in the design we're cooked.

The cons are that the FPGA design is NOT optimized in area, in power consumption, it is much smaller.

ex: a microprocessor implemented in an FPGA goes to 5MHz-50MHz-200MHz at the most, we can't push it more and we have an extreme power consumption.

With modern FPGAs we can dynamically modify the configuration, that's very interesting, we can modify the configuration of the FPGA with a microprocessor and we might also have a hardware accelerator that can be modified without switching off the circuit, so the microprocessor can modify the FPGA and the accelerator while part of the system is running.

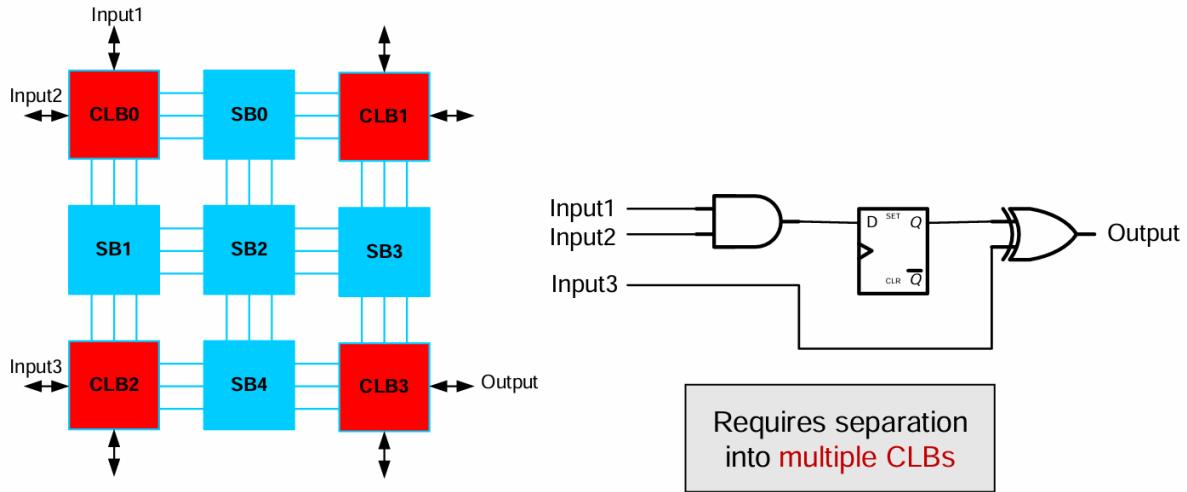
Interconnection Network



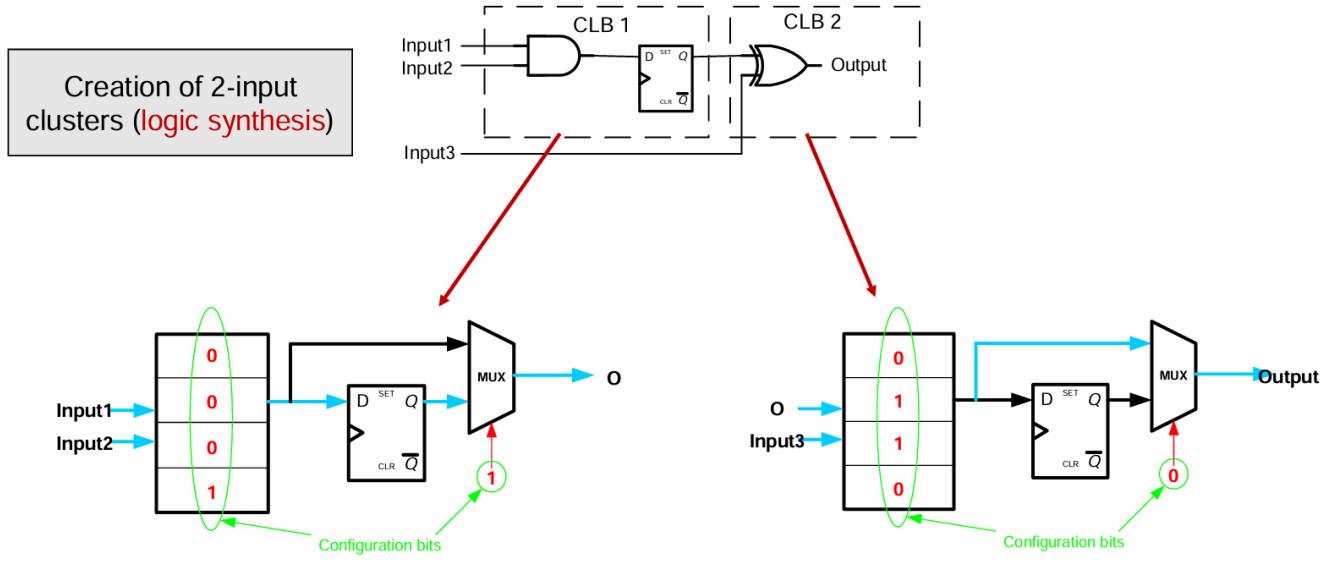
This is an example of a switching block, we have wires with pass transistors that are 1 or 0 based on the configuration bits. We can see that every single crossing point requires a lot of pass transistors and lots of configuration bits. Most of the configuration bits are occupied by routing and routing occupies most of the silicon area.

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.**

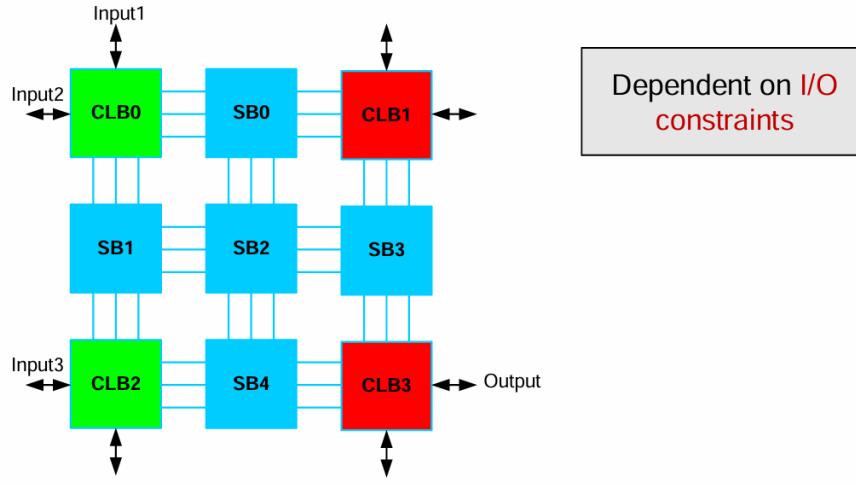


CLBs Required

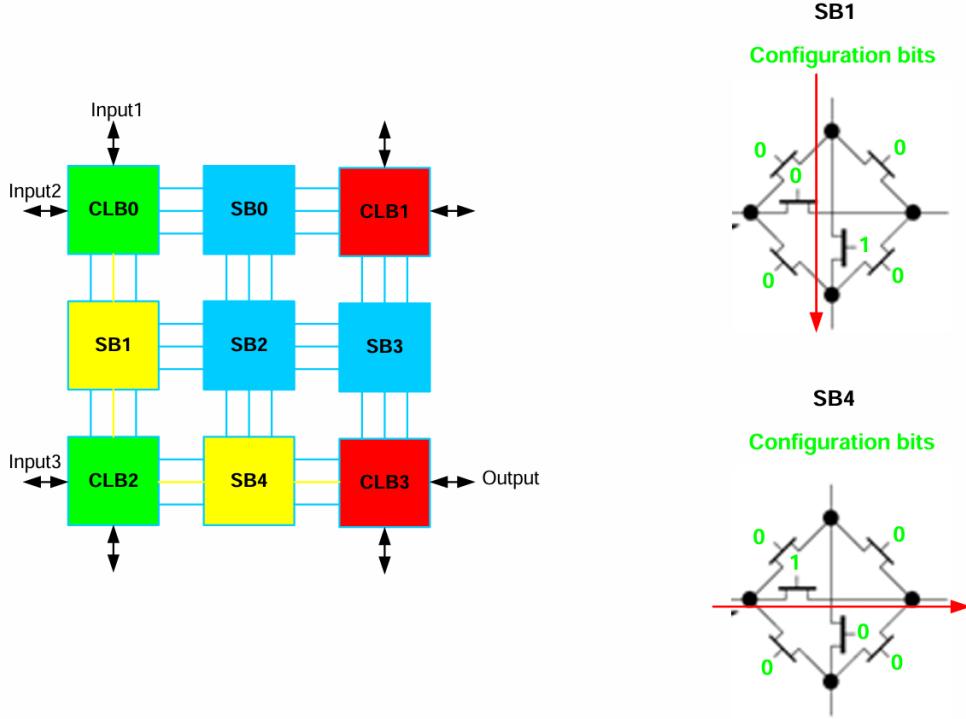


So the first CLB will implement the FF and the AND while the second will implement the XOR. We do have to also implement the connections between the logic

Placement: Select CLBs

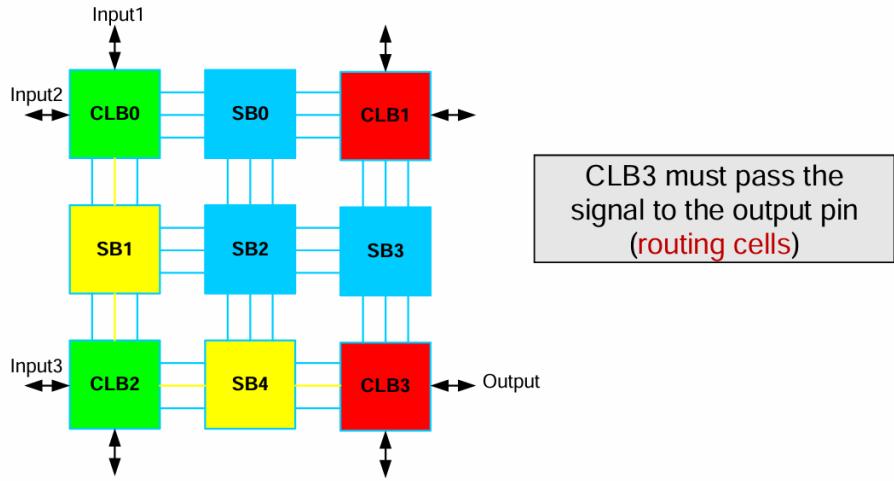


Routing: Select Path



Configuration Bitstream

- The configuration bitstream must include ALL CLBs and SBs, even unused ones
 - CLB0: 00011**
 - CLB1: XXXX
 - CLB2: 01100**
 - CLB3: 01XX0
 - SB0: 000000
 - SB1: 000010**
 - SB2: 000000
 - SB3: 000000
 - SB4: 000001**

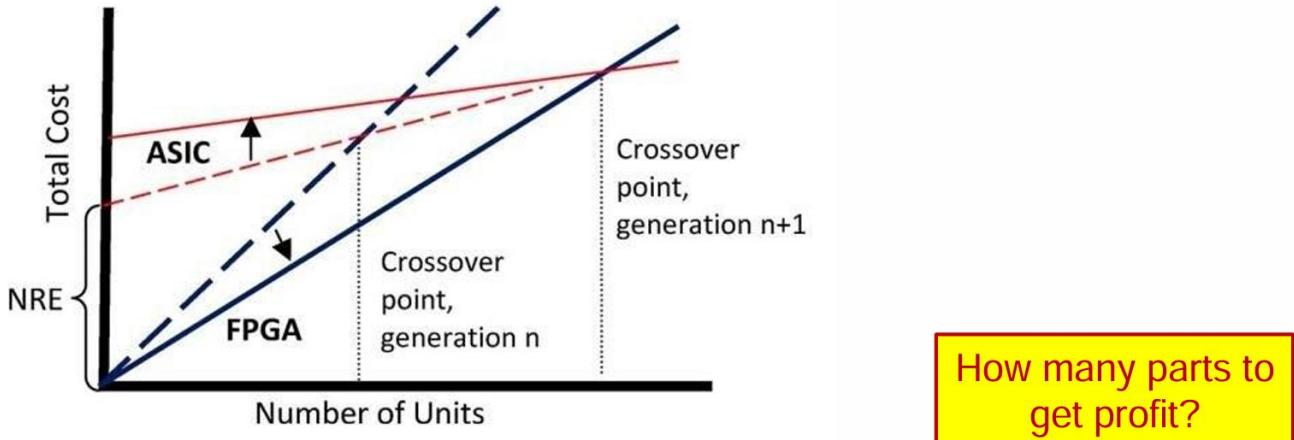


The single CLB is a complex system on which we can build complex functionalities. The gap between the custom implementation and the FPGA implementation in terms of performance can be huge, so we might ask where's the advantage of this approach?

Economics of ASICs

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

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



In this graph we see non-recurrent expenses due to ASIC design, so when we want to implement an ASIC we spend a lot in respective of how many parts we implement, because we pay for the engineers, for the tools, for the std cells and a lot for the silicon foundry, but then we spend little to produce the effective device.

On the other hand, the non-recurrent cost for FPGA is 0, but the cost of the design scales with how many parts we implement.

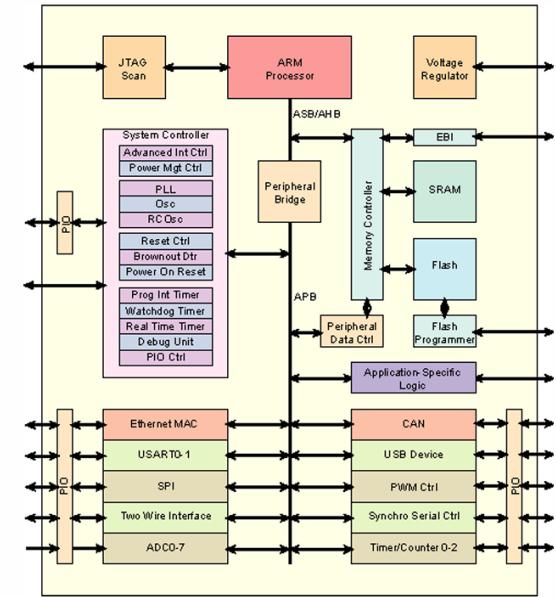
We can also see that there's a cross point with ASICs.

When we need little parts we go with FPGAs, when we need to produce a lot of devices/ICs we go with ASICs, if we plan to sell millions of parts we go with ASIC design while if we are designing a couple of satellites per year a full custom design is unaffordable, we need a FPGA based design. The main driver for the choice between ASIC and FPGA is always the market. FPGA prototyping will give me some insight by emulating the circuit on the FPGA. After we know that the device works on the FPGA, I can produce my SoC.

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
(mainly post-integration testing)
- ☹ Prototyping may take longer
- ☹ Intellectual property (IP) and security issues



We can see some advantages and drawbacks of having such a complex system.

- *Power consumption*: we remove all the parts that are implemented in the FPGA but aren't necessary in the final product
- *Increased reliability*: silicon is much more reliable than the PCB board, board level faults are much more common than silicon faults. When everything is integrated in a single piece of silicon it is more reliable. The drawback is that testing is much more difficult with respect to the board. Here we must test everything on a single piece of silicon, while board testing is much easier.

Oss: what do we mean by testing? We have our piece of silicon and before selling it I must verify that all the implemented parts are working well in the silicon bulk. Post-production testing. Without having probes and connections from the input to the rest of the component it is very hard, we must identify a functional condition of the circuit where the specific part that I want to test is under stress. I must test my system by driving its primary input outputs I have to try to identify whether all the internal components are working or not.

- *Reduced board space*: much less area is used, the system is miniaturized
- The system is much more complex than a normal digital system
- Testing is much more difficult, prototyping takes longer
- Security issues from many points of view: intellectual property

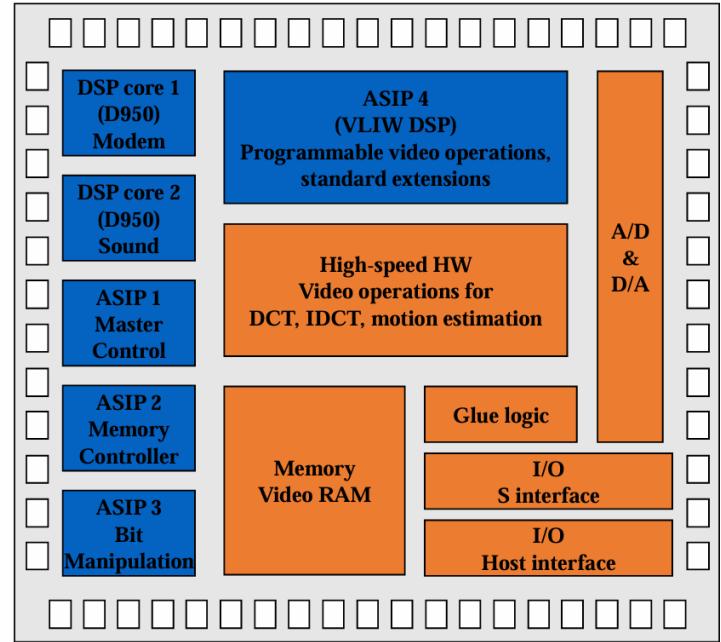
From Specifications to Chips

```

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]);
}

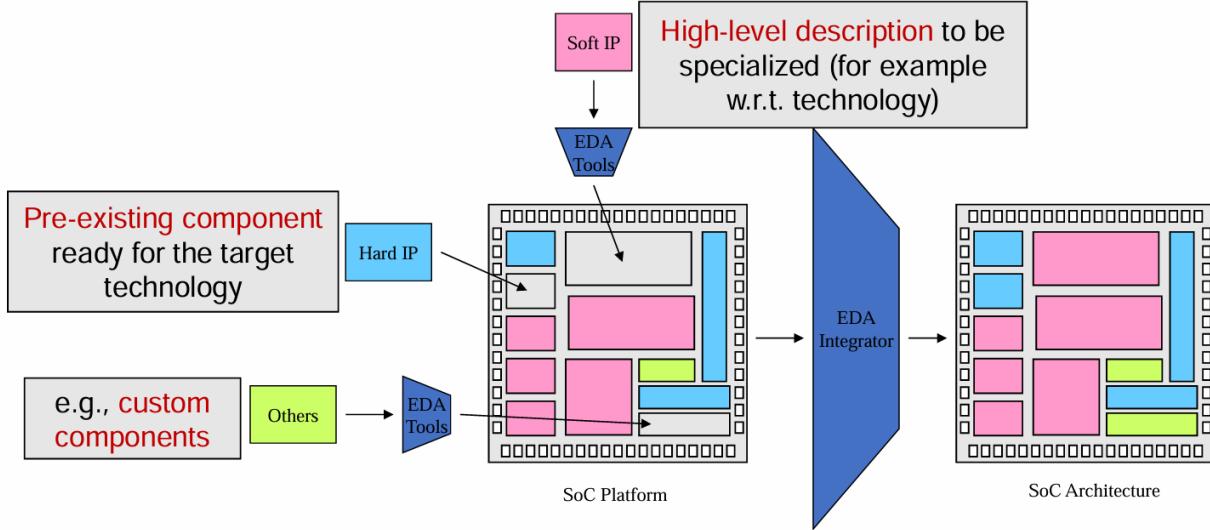
```



This slide represents the beginning and the end of the device description. We start with a very high-level description and then we arrive to the layout.

Platform-based design

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



We want to exploit something that is regular and can be reused design by design, I just act by plugging in additional functionality, we reduce the number of soft/hard IPs

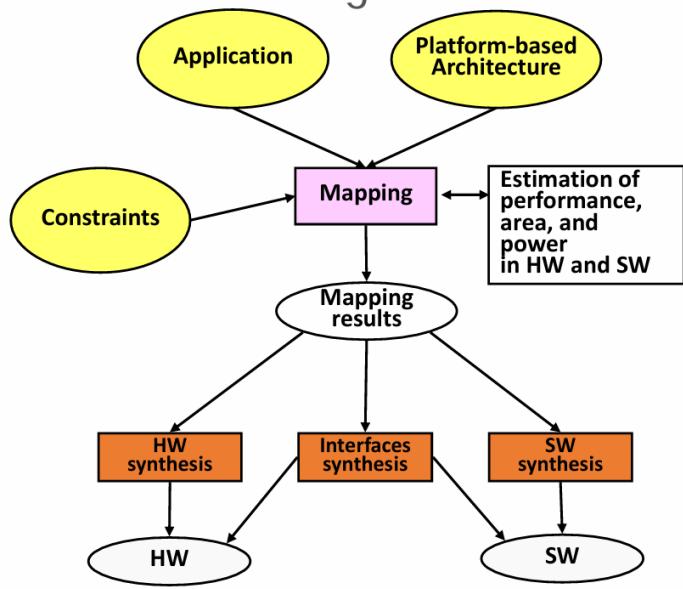
- **Hard IPs:** comes as a netlist, is already implemented in specific technology and is implemented in the cells, these are generally the IP cores I'm purchasing from third parties, so I'm not able to understand how the functionality is implemented and optimized. These are structural, ready to be used.

- **Soft IPs:** they are behavioral, they're described and then I'll synthesize them using my standard cells, which need to be adapted to be used in the system.

Some components might not be available from somewhere/other companies so these will be implemented from scratch, for example NoCs are not commercially available, they must be implemented.

Hardware/Software Co-Design

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



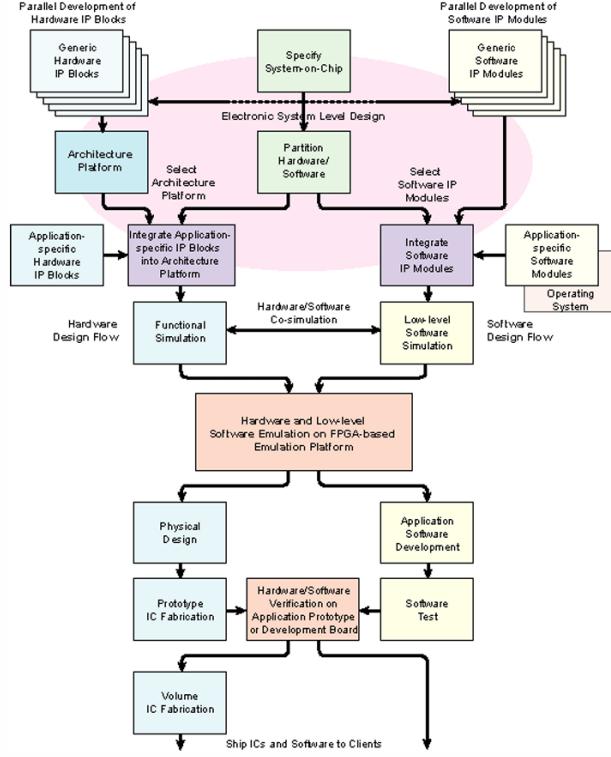
When designing an accelerator, I have a very high-level description of the application, then maybe I have a platform that comes from previous experience or maybe some IP cores, then I have constraints on power, cost, area. As soon as possible I need an estimate as accurate as possible of the figures of merit the constraints.

Mapping: with *mapping* we mean the process of choosing what's going to be implemented in a CPU, in a GPU, in a custom accelerator, what in software, which operative system is needed etc.

Once mapping has been performed, I know what's going to be implemented in hardware and for that part I'll follow my hardware design flow, then for what's going to be implemented in software I'll follow the software design flow.

At the end we have the firmware or the operating system and the hardware.

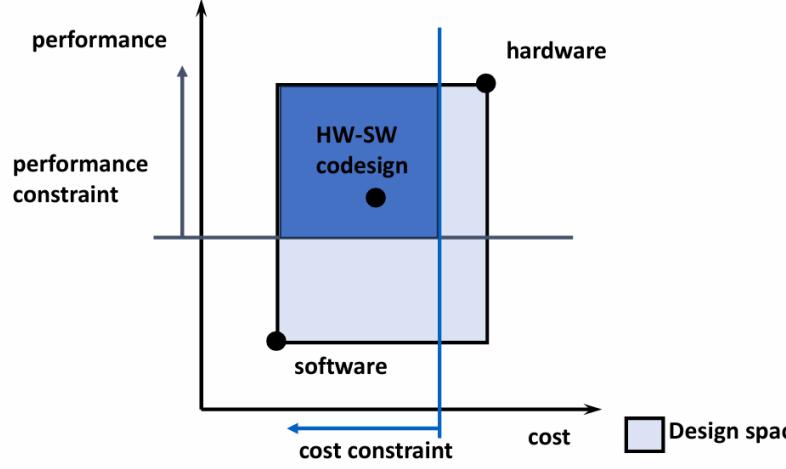
Hardware-Software Co-Design



Hardware-Software Co-Design

Design Space Exploration

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



The design space may not be regular (conflicting requirements)

- Dedicated HW accelerators may decrease computation time but increase data transfer time
- Big PLMs may decrease data transfer time but also decrease reliability
- Component duplication/triplication increases both reliability and silicon area (and thus cost)

What is DSE (**Design Space Exploration**): the design space has a lot of solutions. There might be several hardware software mappings for a single algorithm. Then there are constraints: while respecting them the feasible area of the design space is as reduced as possible. The solution that falls in the feasible area is identified.

4 - Introduction to Hardware Description (Languages)

Slide 3

What is Verilog?

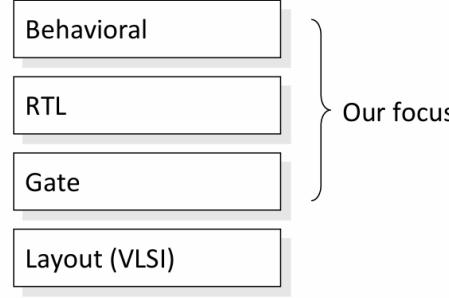
Hardware Description Language (HDL) to describe **digital circuits at many abstract levels** (behavioral, structural)

- Developed in 1984 (Standard: IEEE 1364, Dec 1995)
- Used in almost all semiconductor companies and tools to specify components
- **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
- You may write very bad or even non synthetizable Verilog!

Hardware Description languages are the only way to implement and improve our code and design solutions. High level description languages are useful to describe our function in SoC, but hardware optimization is done with HDL. The concept is that HDLs gives a very fast description, but not an optimized one.

Slide 4

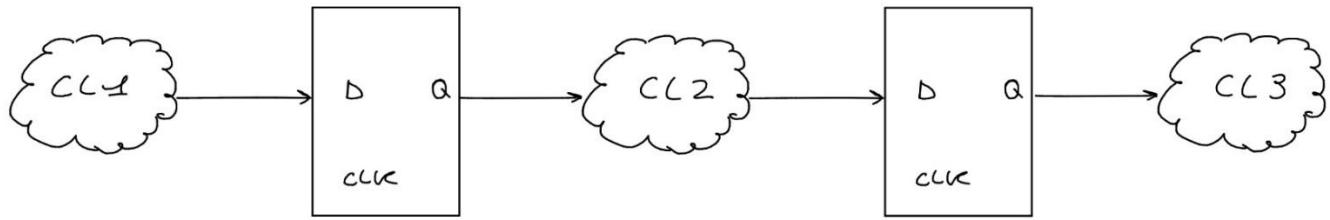
Abstraction Levels in Verilog



Here we see the abstraction levels of HDLs. Verilog and VHDL are very similar, the main differences are related to syntax implementation.

- **Behavioral:** it's the description of the logic of the circuit, there's no concept of clock nor sequentiality, only the functionality of the system is described, without any reference to the technology.
Ex: I must implement an adder, 16 bits in I, 16 bits in O and just make the sum, very similar to a C or C++ description
- **RTL:** RTL stands for **Register Transfer Level**, it is specified the registers logic that combined give the combinational logic.

RTL



- **Gate:** netlist level description of the circuit, direct electronical implementation

The lower we go, the more we go “near” the technology level.

What we must keep in mind is that we can't write not synthesizable code, there are parts of the behavioral descriptions that can't be synthesized. Every step of the description has its own characteristics, non-idealities, constructs, statical details but in the end, before a gate level description I must have a synthesizable circuit.

Where can we use Verilog HDL?

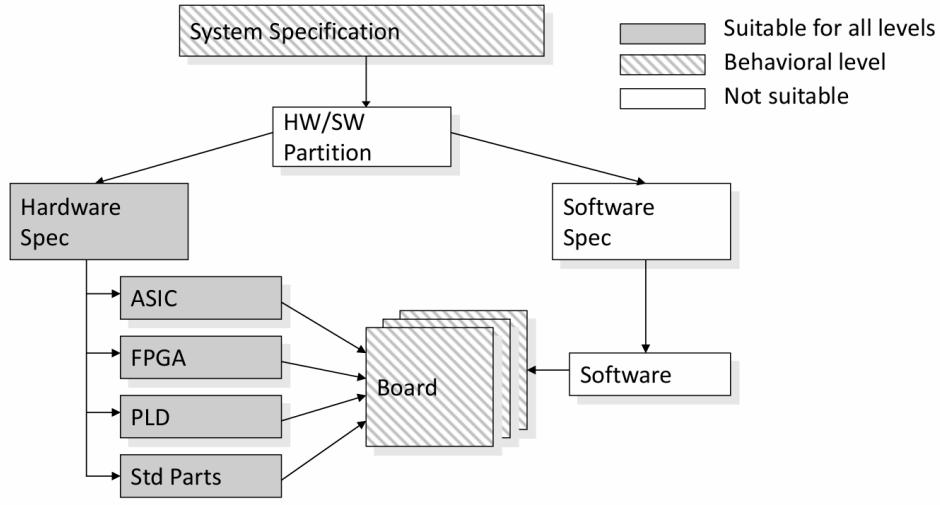
Not only HW description.

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

Every step of the design flow can (must!) be verified

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



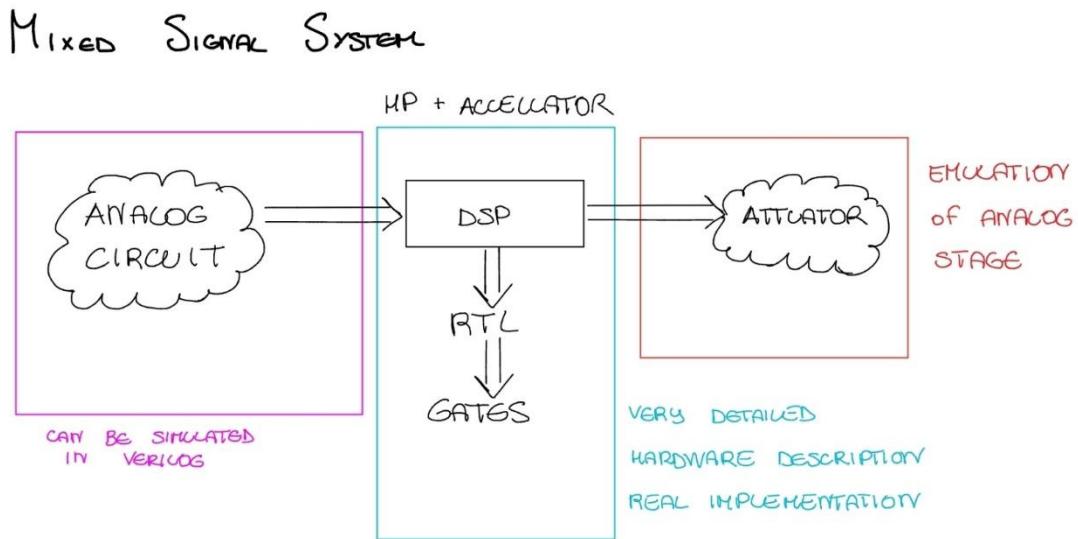
Which are the parts of the design where we can use Verilog? When looking at the HW description, the purely logic part of the circuit we can use any level of the circuit.

oss: one thing we did not mention when talking about FPGAs is that we need the specific characteristics of the device, how many LUTs, how many I/O in the LUTs, how many FFs, which is the format of the bitstream to configure the FPGA device etc. We need details coming from the vendor, we do not need the standard cells but we need the developing environment etc.

These requirements are very similar to the ones that we have when going for an ASIC implementation, in fact we need the libraries from the producer of the standard cells.

The tools to implement ASICs are different from the one for FPGAs etc.

Coming to Verilog, looking at the pure hardware stuff it can be used in any part, when looking at the hardware we can use Verilog for behavioral description. Keep in mind that at the system level we can also have analog parts of the circuit, most of the circuits are mixed signals. All the digital parts are covered by Verilog but are implemented by digital designers. Analog parts can be simulated in Verilog in a functional way but can't be described and implemented with Verilog. For software specs, Verilog can't be used.



Verilog fundamentals

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

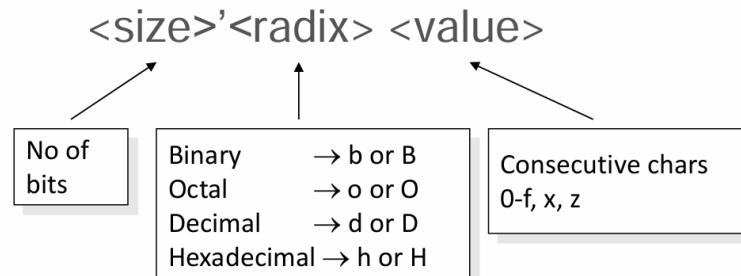
List element separator: ,

Statement terminator: ;

Comments

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

Numbers in Verilog (i)



Examples:

- `8'h ax = 1010xxxx`
- `12'o 3zx7 = 011zzzxxx111`
- 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

Since we're describing hardware, we must specify the dimensions of the bits.

Slide 10

Numbers in Verilog (ii)

- You can insert “_” for readability
 - `12'b 000111010100`
 - `12'b 000_111_010_100`
 - `12'o 07_24`
- Represent the same number

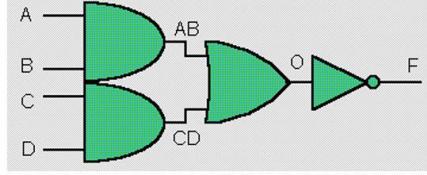
Slide 11

Basic Syntax of a Module

module *module_name* (*module_port*, *module_port*, ...);

Declarations:

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 “;”

Module: container of functionality, it's a component being specified in a behavioral/RTL/gate level way.

The description of a module begins with

Example

```

module full_adder (A, B, c_in, c_out, S);
    // declarations
    // system description
    
```

Direction can be omitted

endmodule

Input/Output Declarations

- **Input Declaration**

- Scalar

- **input** *list of input identifiers*;
- Example: input A, B, c_in;

- Vector

- **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;

Oss: while the numbering of the sub-components in an array in C is fixed, in Verilog we can write whatever we want

```
Input [15:0] x;
Input[17:2] y;
Input[1:16] k;           MSB – numbered 1, LSB – numbered 16
```

Example

```
module full_adder (A, B, c_in, c_out, S);
    // declarations
    input [15:0] A, B;
    input c_in;
    output c_out;
    output [15:0] S;

    // system description

endmodule
```

Nets

- Can be thought as hardware wires driven by logic
- Equal z when unconnected

- wire

Wire signals and variables are always assigned in parallel.

Registers

- 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
```

Reg: are assigned sequentially, are modified in procedures, we need to specify procedures within the modules to modify the content of the register. Although they're called registers, they are not FFs, it is not the description of a flip flop. It's a variable concept in C, it's something that is storing data.

Vectors

- Represent buses

```
wire [3:0] busA;
reg [1:4] busB;
reg [1:0] busC;
```

- Left number is MSB (most significant bit)

- Slice management

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

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

$$\text{busB} = \text{busA}; \Leftrightarrow \begin{cases} \text{busB}[1] = \text{busA}[3]; \\ \text{busB}[2] = \text{busA}[2]; \\ \text{busB}[3] = \text{busA}[1]; \\ \text{busB}[4] = \text{busA}[0]; \end{cases}$$

Integer & Real Data Types

- Declaration

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

- Used as registers

```
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

- Used for behavioral description but they cannot be synthesized

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
- Two input operands evaluated to ONE bit output value: `0`, `1` or `x`
- Result is ONE bit value: `0`, `1` or `x`

A = 1;
B = 0;
C = x;

A && B → 1 && 0 → 0
A || !B → 1 || 1 → 1
C || B → x || 0 → x



but C&&B=0
As well as C|!A=1

and	0	1	x	z	or	0	1	x	z	nand	0	1	x	z	nor	0	1	x	z	
0	0	0	0	0	0	0	1	x	x	0	1	1	1	1	0	1	0	x	x	
1	0	1	x	x	1	1	1	1	1	1	1	0	x	x	1	0	0	0	0	
x	0	x	x	x	x	x	1	x	x	x	x	1	x	x	x	x	x	0	x	x
z	0	x	x	x	z	x	1	x	x	z	1	x	x	x	z	x	0	x	x	

xor	0	1	x	z	xnor	0	1	x	z	buf	input	output	not	input	output				
0	0	1	x	x	0	1	0	x	x	0	0	0	0	1	1	0	1	1	0
1	1	0	x	x	1	0	1	x	x	1	1	1	1	0	0	0	0	0	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
z	x	x	x	x	z	x	x	x	x	z	x	x	x	x	z	x	x	x	x

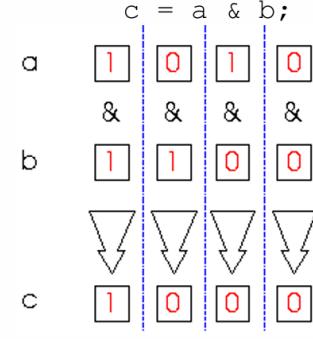
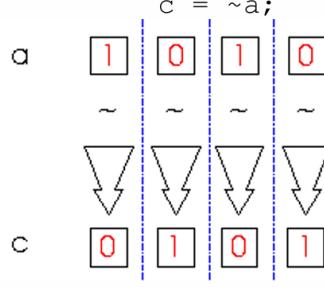
Bitwise Operators (i)

- & → bitwise AND
- | → bitwise OR
- ~ → bitwise NOT
- ^ → bitwise XOR
- $\sim^$ or $\sim\sim$ → bitwise XNOR
- Two multi-bit inputs and one multi-bit output
- Operations on a **bit-by-bit basis**

Bit padding?

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
```

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

- $\{op1, op2, \dots\}$ → 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 !! 1 is not a 'b
```

- 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          → 1
'b1x1 <= 0     → X
10 < z         → X
```

Equality Operators

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

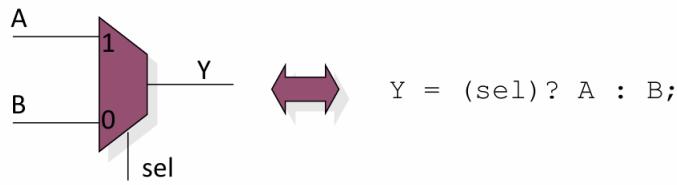
- $4'b\ 1z0x == 4'b\ 1z0x \rightarrow X$
- $4'b\ 1z0x != 4'b\ 1z0x \rightarrow X$
- $4'b\ 1z0x === 4'b\ 1z0x \rightarrow 1$
- $4'b\ 1z0x !== 4'b\ 1z0x \rightarrow 0$

Logical equality: confronts the whole operator

Case equality: confronts each single bit

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 = -'d12; // stored as 216-12 = 65524
regA/3      evaluates to 21861
```

Behavioral operators, not synthesizable operators.

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

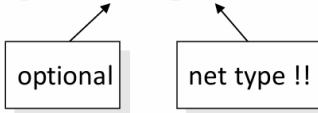
<code>+- ! ~ unary</code>	highest precedence
<code>* / %</code>	
<code>+ - (binary)</code>	
<code><< >></code>	
<code>< <= => ></code>	
<code>== != === !==</code>	
<code>& ~ &</code>	
<code>^ ^~ ~^</code>	
<code> ~ </code>	
<code>&&</code>	
<code> </code>	
<code>? : conditional</code>	lowest precedence

Use parentheses to enforce your priority

Continuous Assignments (for wires)

- Syntax:

```
assign #delay <id> = <expr>;
```



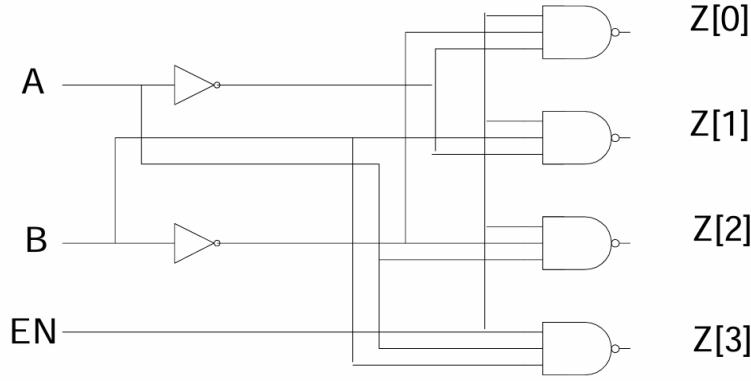
- Where to write them:

- inside a module
- outside procedures

- Properties:

- they all execute in parallel
- their order is independent
- 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 Modeling

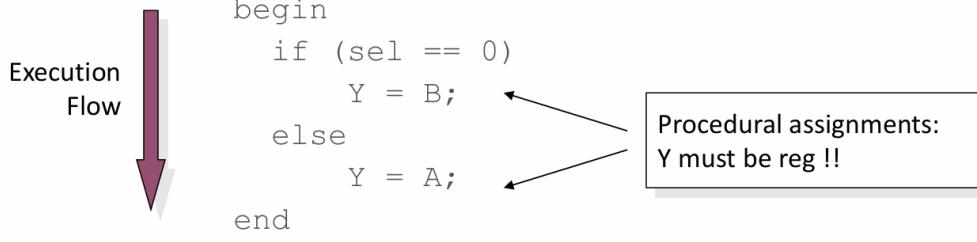
- The behavior of a design is described using procedural constructs
 - initial statement*: this statement executes only once.
 - always statement*: this statement always executes in a loop forever.....
- Only register data type can be assigned in these statements

Initial: purely simulative/behavioral statement

Always: this statement always executes in a loop forever

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:



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

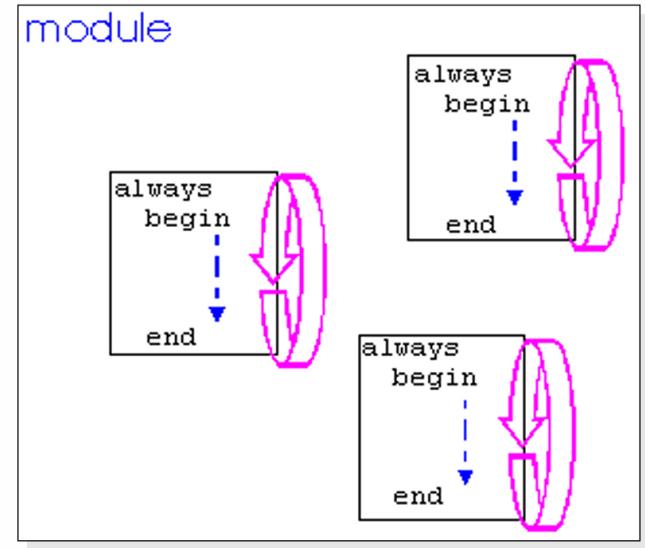
```
initial begin
  #50;
  $display("Really?");
end
```

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



Different `always` procedures are executed in parallel, but a FF is always a FF so every piece of the synchronized circuit is synchronized as an always (???)

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....

I have the sensitivity list for “always” statements

Events

- @

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

execution triggers every time any signal changes

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

execution triggers every time a signal changes from 0 to 1

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

execution triggers every time a signal 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

E.g.

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

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

while (expr) stmt;

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

NOT SYNTH

E.g.

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

repeat (times) stmt;

reg [3:0] Y;
wire start;

initial
    Y = 0;

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

repeat (times) stmt;

Can be either an
integer or a variable

Procedural Statements: forever

forever stmt;

Executes until sim
finishes

Typical example:

clock generation in test modules

module test;

reg clk;

$T_{clk} = 20$ time units

initial begin

clk = 0;

forever #10 clk = ~clk;

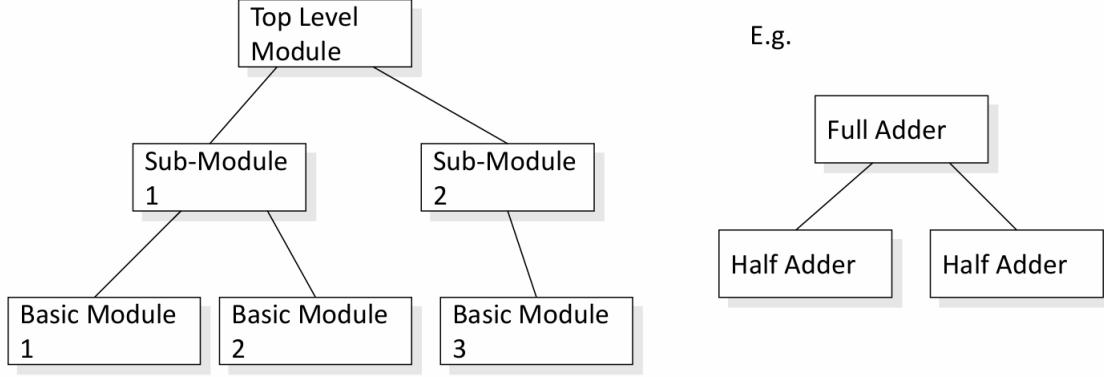
end

other_module1 o1(clk, ...);

other_module2 o2(..., clk, ...);

endmodule

Hierarchical Design



I can implement various modules and connect them

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

```

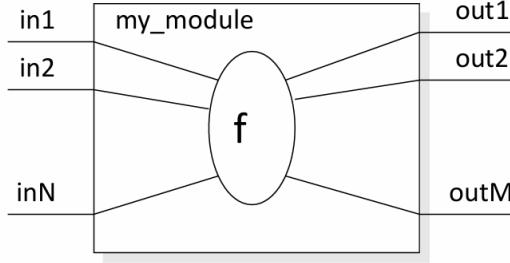
Behavioral & Hierarchical Verilog

```

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

```

Module



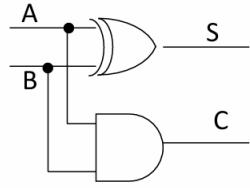
```
module my_module(out1, ..., inN);
    output out1, ..., outM;
    input in1, ..., inN;

    .. // 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;

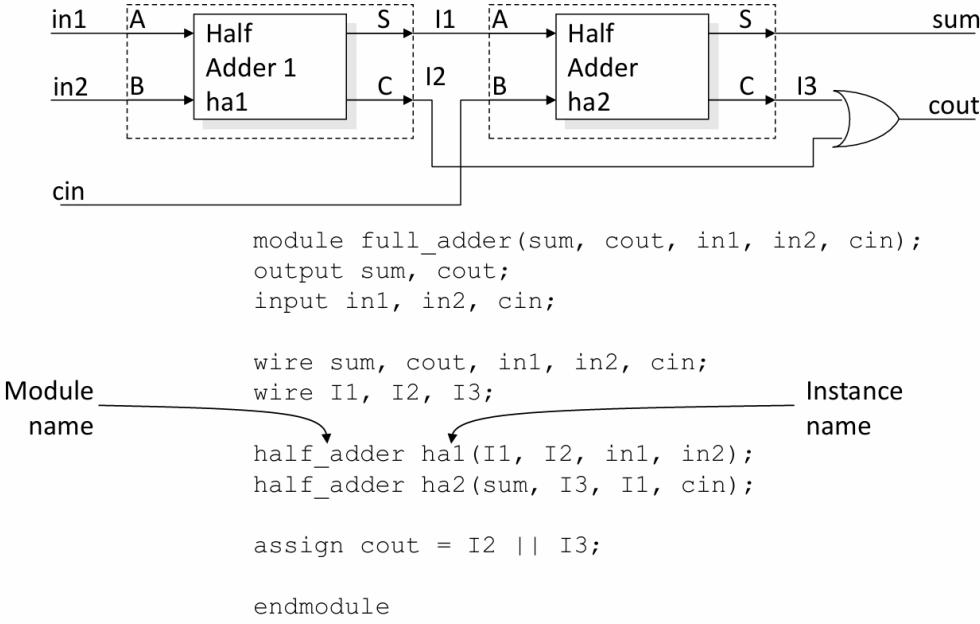
    wire S, C, A, B;

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

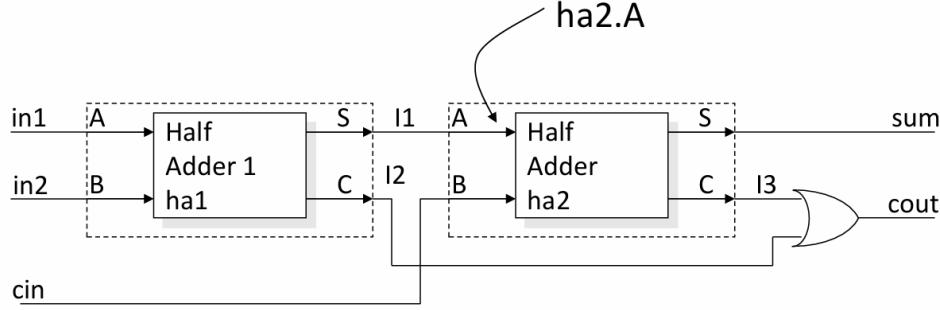
endmodule
```



Example: Full Adder

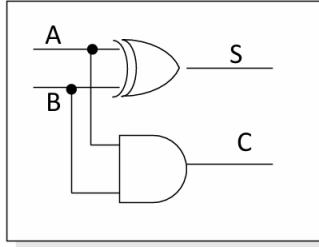


Hierarchical Names



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
```

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);
```

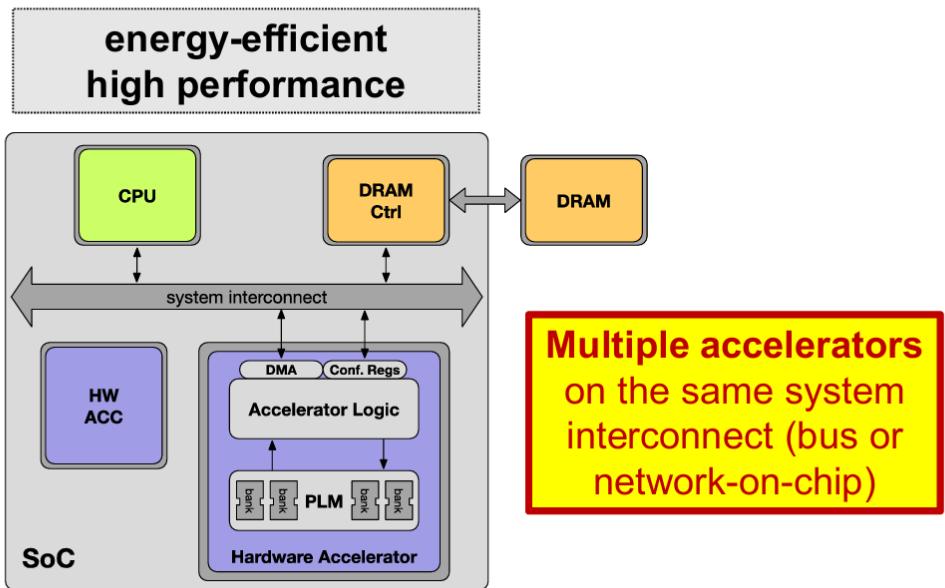
2-input NAND without delay
 3-input AND with 2 t.u. delay
 NOT with 1 t.u. delay and instance name
 2-input XOR with instance name

- Write them inside module, outside procedures

5 - SoC Challenges

Abstraction of Heterogeneous SoCs

Specialized microarchitecture for both computation and storage



Parallelism in memory access, parallelization in accelerators can be implemented.

Modern HSoC Challenges

System-Level Optimization

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

Programmability

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

Reliability/Fault Tolerance

Understand how to protect System execution against random faults

Hardware Security

Understand how to protect the Intellectual Property but also the data within the SoC and the SoC itself

These are the four families of issues that a designer has to take into account in the system architecture.

System level optimization: system architecture is the *design of the system not at hardware level but at system level*. Optimization at hardware level is a challenge for digital electronics system designer (power consumption, silicon area constraints...).

System architecture takes into account *both software and hardware*.

We have to decide “when do we make the hardware exploitable by software?”, “how would the processor control a specific hardware accelerator? By software? By firmware? By the ISA with a specific instruction?”. Basically we’re talking about *operating system implementation*. Communication is also a significative issue, because this means *choosing standardization* for the device.

Remember that when we work on the system architecture *we want to write high level code*, the higher the level the simpler is the expression of the functionality of the system. Then, by exploiting high level synthesis, an hardware description of the system is obtained.

Programmability: obviously the *programming paradigms that are used in heterogeneous SoC* are very different from the ones that we are used to for desktop/server applications. For the high-level users, these are negligible aspects because it’s the OS that gives the user the libraries to implement the specific application we want. The complexity of OSs for HSoCs (Heterogeneous Systems on Chip) is totally different from the complexity for a desktop system.

Reliability/fault tolerance:

Reliability definition: ability to produce the intended functions under normal operation and even under small fluctuations in the computing environment for a specified time period [\[1\]](#)

we have different reliability issues depending on the specific application, they might be more important for embedded systems (ex: automotive, railway, satellites systems). *Which systems have to expose reliability requirements? Is there a specific class of systems?*

The answer is that every system has to expose reliability, then the specific requirements depend on the specific type of system but from the designer point of view, the system must work at its specific conditions.

Examples:

Is the SoC going to work in a train braking system? → implies a very strict safety system.

Am I designing a smartphone? → then I have other safety requirements.

Remember that reliability expectations affect the reputation of the specific company, so they’re a very important part of the design process that must carefully be taken into account.

Hardware security: security has become critical both with respect to *data processed by the system* and in the sense of system *intellectual property*, so the how the processing of the information is implemented.

Oss: reliability and security are both properties that intersect.

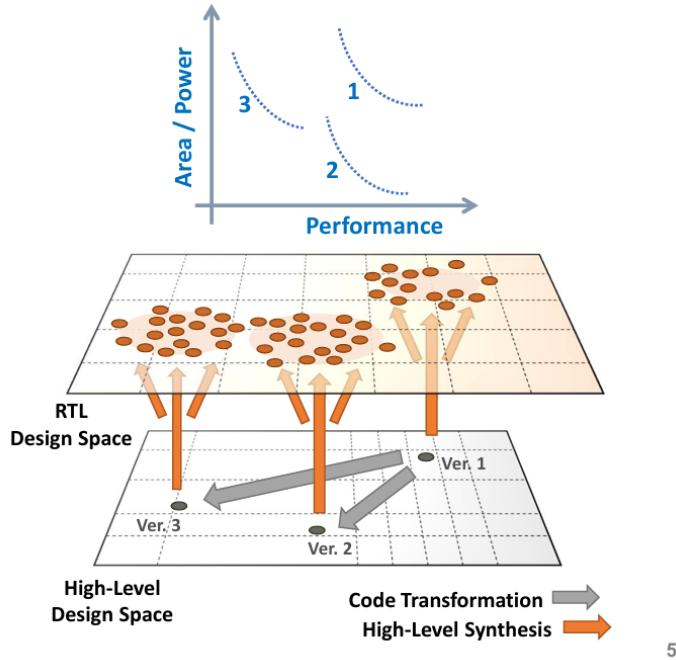
From Software to Hardware

Input language for system specification is an open question

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

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

- C does not express concurrency
- Arrays vs. pointers
- Multi-port interfaces
-



5

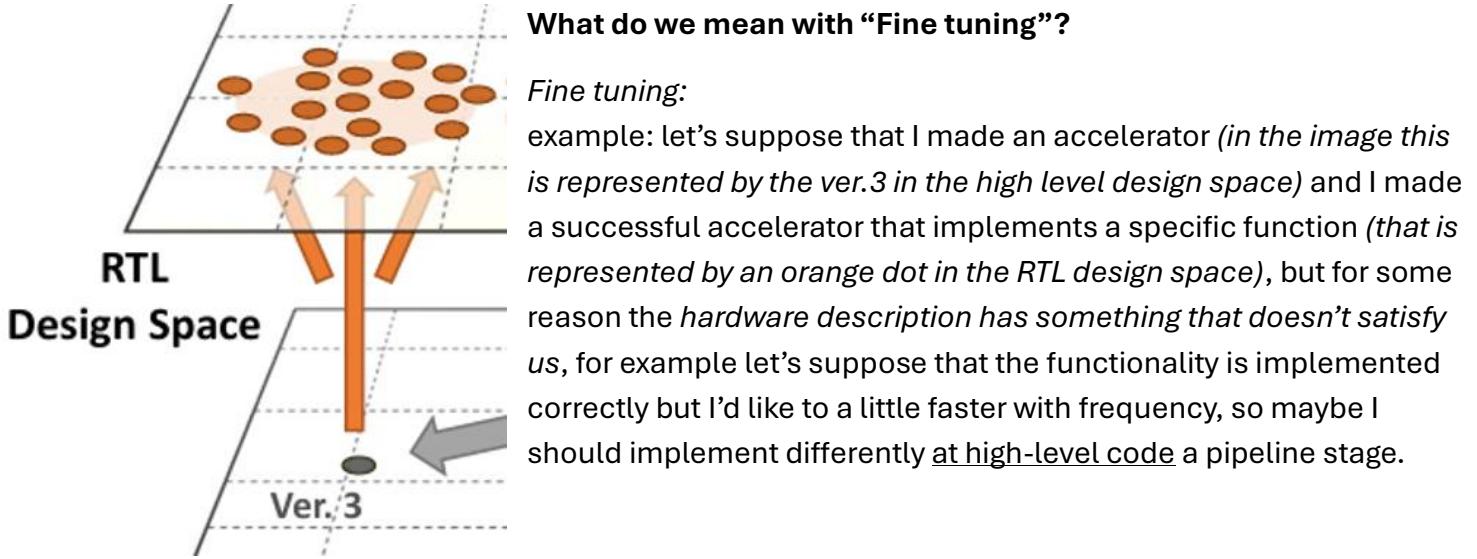
When designing our hardware accelerator, we must consider the fact that *the software aspect of the high-level language can be conflicting with the hardware description*.

here are some examples to better understand what is meant:

1. I can't have dynamic memory allocation in hardware, while in C I can
2. I can't have multi-accessible rams because they're very difficult to implement properly in hardware
3. Everything is concurrent in hardware while in programming often I don't have concurrency

So often I have clusters of RTL description that fall in the same software description. Every single pattern of high-level language is going to be interpreted in a specific way.

If I have in mind that my optimum solution is a specific one, but the hardware description has something that I don't like, we must *change the high-level code*.



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**

```
void Gsm_LPC_Analysis(word* in, word* out)
{
    longword int_mem[9];
    Autocorrelation(in, int_mem);
    Reflect_coeff(int_mem, out);
    To_Log_Area_Rat(out);
    Quant_and_coding(out);
}
```

Structural definition: pointer-based operations (arithmetic, accesses to external memory, ...)

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

Streaming definition: data transfers to exchange data blocks with main memory

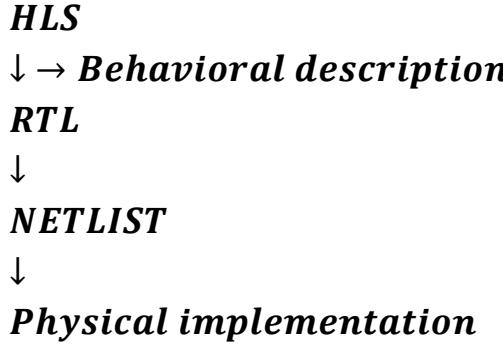


Left: I have the high-level programming language, we see a Private Local Memory (the array) and we have four function calls.

Right: we have the *streaming definition* that describes the data transfer.

We have three threads, one for loading, one for the computing process that is going to oversee implementing the functionality and at the end we have the store function. Note that we don't have any hardware detail, we have a sequential, structural code.

Remember that this is the programmer approach to the design of the hardware accelerator

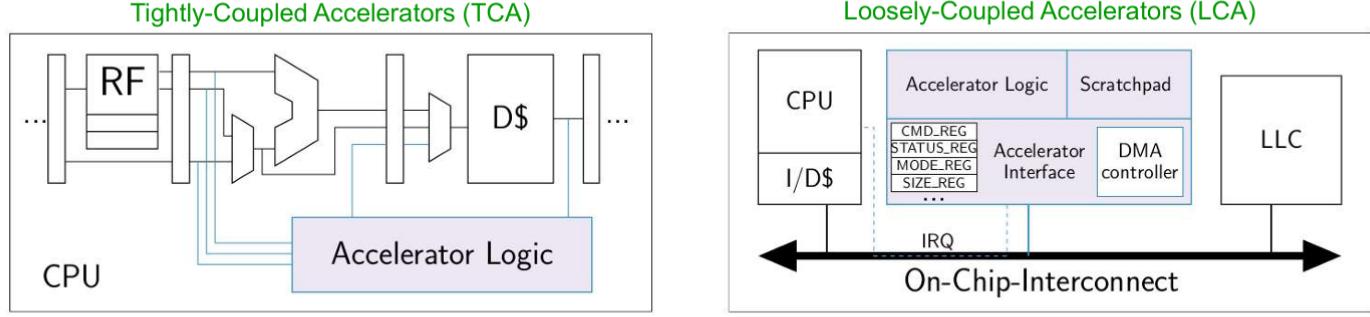


Between all these levels we must perform the equivalence check: we must verify at every step if the functionalities are the same. Obviously the more low-level I go the more requirements I can verify, when I move towards physical implementation I can check timing, power, area constraints.

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



Tightly Coupled Accelerator: (*this option is available only if the CPU is open source*) since we're modifying our processor to implement the specific hardware function inside the processor, this would imply a new instruction in the ISA. The resources are shared with the CPU, so generally I would need less memory access, less communication with respect to loosely coupled accelerator, but often I still need more physical memory. If the accelerator is in the CPU, it is going to be using registers of the CPU.

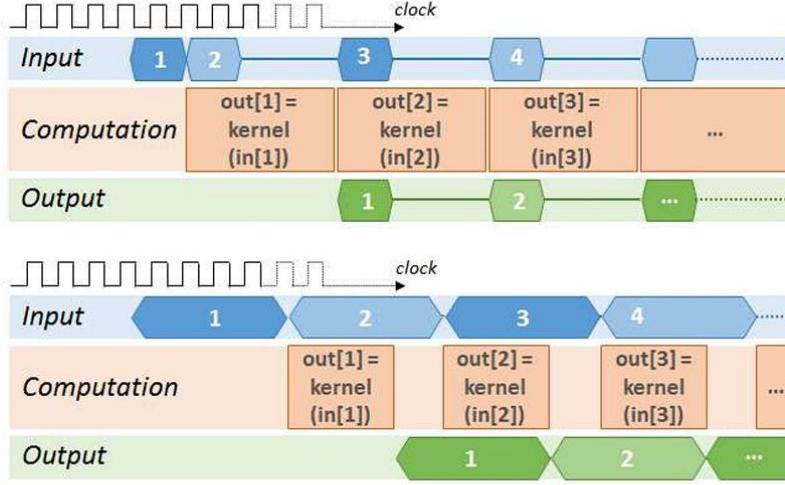
Note that if we are implementing it in a single core scenario, if the accelerator takes a long time to perform the acceleration, then I'm preventing the core from executing other applications and I'm basically stalling the other computations.

Loosely Coupled Accelerator: (*only option is available only if the CPU is not open source*) with respect to a TCA, since it is not implemented inside the CPU, we need additional circuitry for synchronization and communication, the good point is that the accelerator might have his own DMA and his own PLM, moreover here the accelerator can run for as long as we want because the CPU isn't stalled.

For both cases the issue could be that *the accelerator is too fast for the system*, I do not need to implement fast accelerators if the controller is slow.

Balancing Execution Phases

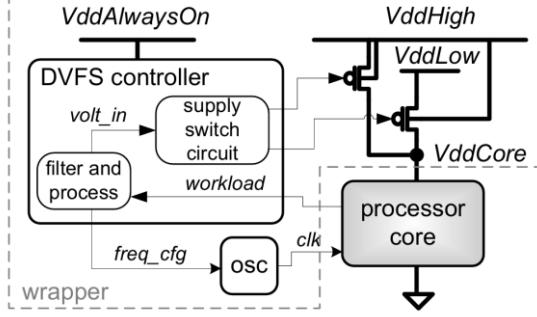
Each component with intense data accesses requires to **balance computation and communication**
inefficient optimizations may create **execution stalls**



As we already saw, we want to balance the two platforms.

I have different approaches to optimize and balance the applications:

- **Dynamic Voltage and Frequency Scaling (DVFS):** hardware level approach, we raise or lower the voltage/frequency of the system in specific moments to synchronize/optimize the computation and optimization



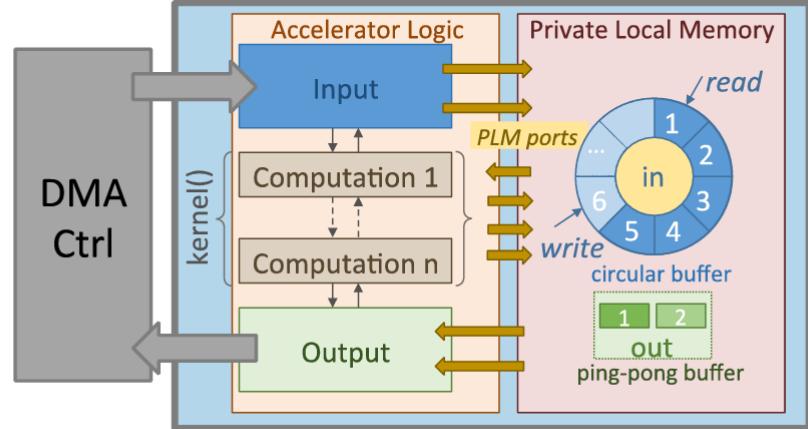
- **Latency Insensitive Design (LID):** architectural and system level approach, it's how I dynamically balance the timing and execution phase without considering the specific timing constraints by designing our system with specific functions and signals.

DMA-Based Coprocessors

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 it is **impossible to predict and optimize all situations where it will be used**



Since we want reusability, I don't can't consider every possible case, I want the system to be general and reusable. The drawback is that if I'm not specific I lose performance in terms of optimization.

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

1024x32bit array in an industrial CMOS 32nm
Distributed registers
145,707.5 μm^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!)

Memory IP block
35,106.6 μm^2
(4x area reduction)

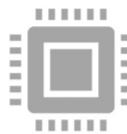
Technology constraints limit the parallelism

Multi-bank architectures
based on memory IPs

These are the two different approaches to the design of “on chip memory”: we would like to design our system with many different parallel memories.

- **Distributed registers:** I fill my chip with registers every register can be accessed *independently* from the other ones. In terms of performance or computation but requires a large area and the memory is not centralized.
- **IP block of memory:** the slide shows that if I implement the same whole memory with dedicated IP blocks: I use $\frac{1}{4}$ of the area but the memory is accessible by less components and I have less concurrency.

Data Footprint Gap



Algorithm data footprint in rapidly increasing

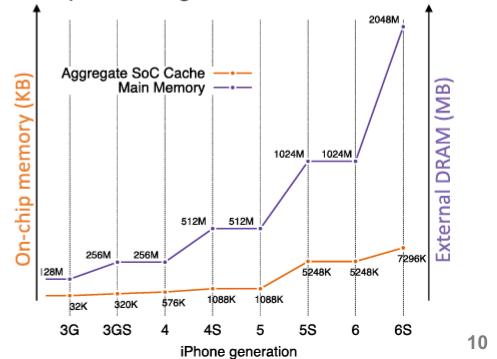
Full HD image (1920x1080 pixels): ~33 Mb
8K UHD image (7680x4320 pixels): ~530 Mb

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



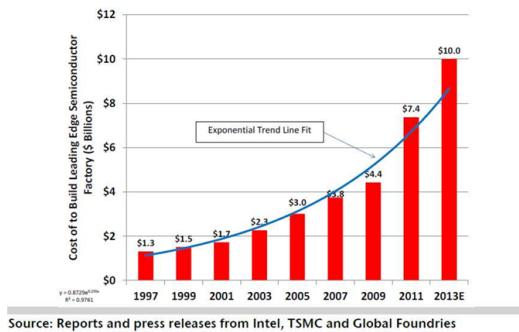
Data footprint: quantity of memory I'm going to use to perform the computation with respect to how much memory I have on the SoC. The size of the memory that can be integrated on the chip is way smaller than the size required by the standardized algorithms.

The most significative part of the slide is the right graph: in blue we see the required external memory, in orange we see the quantity of memory that is implemented on the chip. We see that while the required memory increases exponentially, the memory that is implemented increased much less, it's almost linear. We must pay attention to the order of magnitude of the two graphs, the blue curve is in *megabits* (10^6) while the orange curve is in *kilobits* (10^3) so we realize that external memory implementation and its communication with the SoC is critical for performance since a lot of data goes through it.

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)

reduce manufacturing costs by outsourcing chip fabrication (security issues)



12

The cost of Silicon is increasing because also the process of Silicon production is increasing.

Programmability Issues

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

component generation



system integration



Key enabling properties

modularity
flexibility
scalability
reusability

Programmability Assets

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

- Requires standardization of the interfaces

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

Reliability/Fault Tolerance

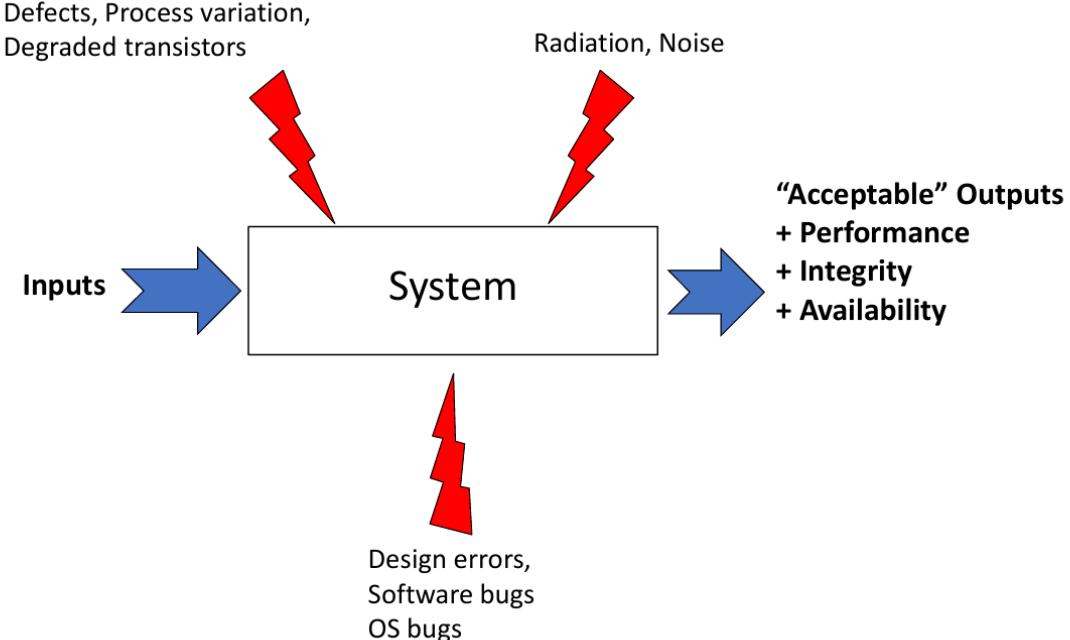
Functional Verification

- A lot of effort is devoted to make sure the implementation
 - matches specifications
 - fulfills requirements
 - meets constraints
 - optimizes selected parameters (performance, energy, ...)

- Nevertheless, even if all above aspects are satisfied ... things may go wrong
- ➔ systems fail

systems fail ... because something broke

Reliability/Fault Tolerance



Fault: event that interacts with my system and *changes the functionality of the system*. Faults may be due to

- *Design errors:* errors introduced in design time.
- Statistical degradation of the performance due to wear out processes, it can degrade so much that the functionality isn't maintained anymore.
- Statistical degradation due to statistical fluctuations of Silicon processes.

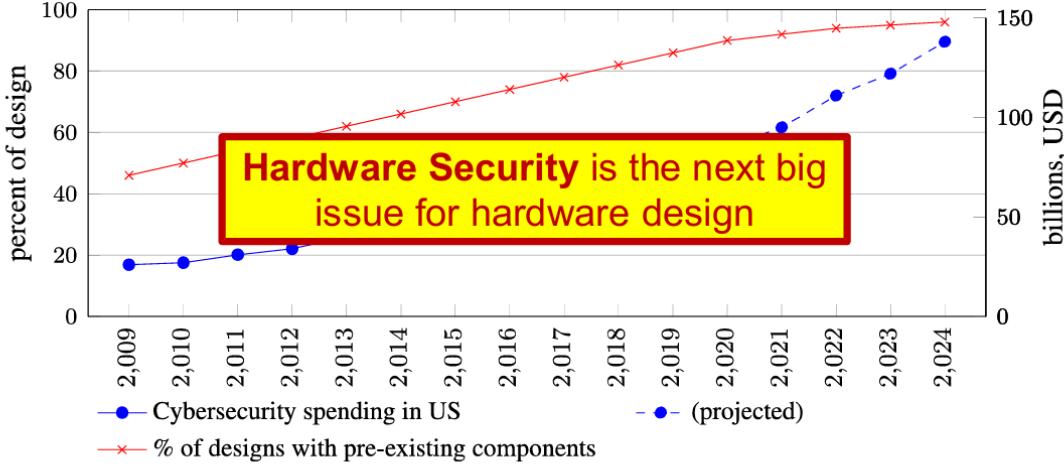
Reliability/Fault Tolerance

- In the 2003 elections in Schaerbeek (Belgium), an anomalous recorded number of votes triggered an investigation that concluded an SEU was responsible for giving a candidate named Maria Vindevoghel 4,096 extra votes ($2^{12}!!!$)
- In 1996 the ESA unmanned space launcher Ariane 5 auto-destroyed 39 seconds after launch due to an unrecoverable error experienced by the autopilot. The 64bit mantissa of the speed was saved in a 16bit integer value (SW reuse from the previous Ariane 4, which was much slower!!!) causing an overflow.

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**



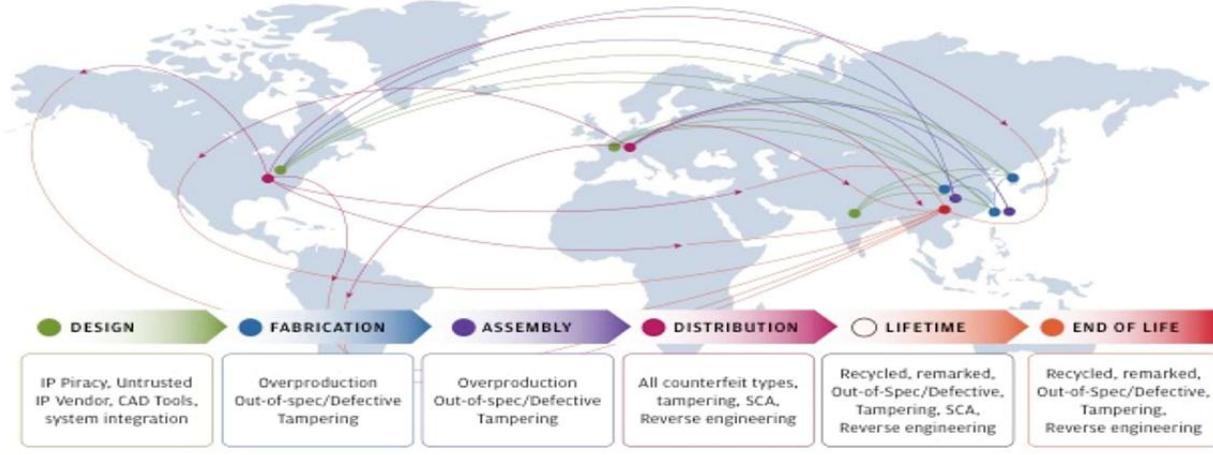
As shown by the graph, most SoCs are implemented using third party components.

In 2024 almost 100% of the SoC is composed by pre-existing components implemented efficiently in the SoC. Purchasing third party components means that we don't have control in any part of the system, we don't have the HLS code nor the design characteristics of the device. This is a different approach to what we were used to in the past: the single company had full control on the production process of the chip, most of the companies had their own silicon foundry and controlled the chip implementation from scratch to the finite product. Today most of the companies are *fabless*: companies do not have their own foundry to implement their own chip, they must go third parties (ex: TSMC, ST, Intel, Samsung...) and use their implants and production processes to produce their chips.

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

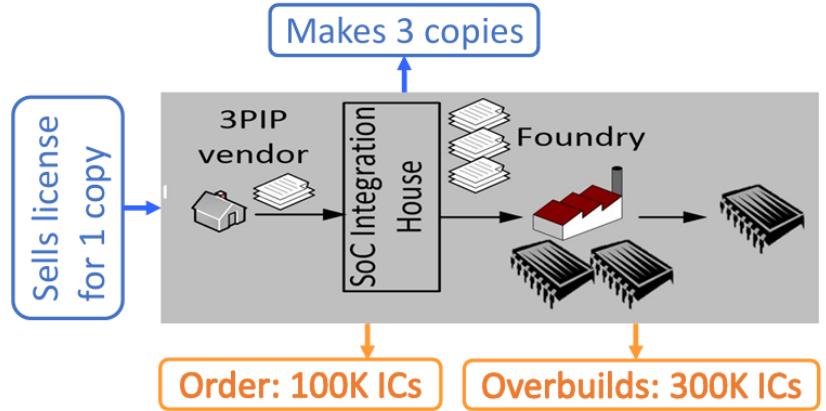
- Malicious SoC integration house
- Malicious foundry

Real-life impact

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



EE Times
ARM files patent infringement suit against IP startup picoTurbo



What to Protect?



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

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

Hardware assisted security: the target is protecting the *data processed by the system*

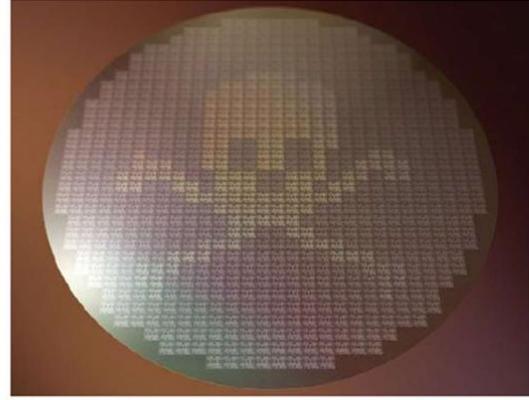
Hardware security: the target is protecting *intellectual property*

ex: *I make a very fast PCI-express and we don't want the others to copy it*

Hardware Trojan Horses (HTH)

Additional unexpected circuitry may be added to a system by:

- IP providers
- Malicious employees
- CAD tools
- Silicon foundries



Hardware Trojan Horses (HTH)

The Outside the Box Israeli Air Force operation:

- Eight fighter planes from IAF attacked and destroyed a nuclear plant (under construction) in Deir ez-Zor, Syria, in 2007
- All Syrian radars and air defense missile systems switched-off simultaneously
- Syrian defense equipments featured COTS components
- All these information have been confirmed by IAF in 2018

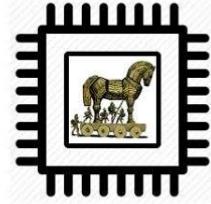


Ex: parts of the defense equipment were from Intel Israel

Hardware Trojan Horses (HTH)

Moreover, the novel menace of Software-Exploitable HTHs raised

An “undocumented feature” that allows an unauthorized privilege escalation through the execution of a particular sequence of instructions has been found in the Via Technologies C3 processor



- N. G. Tsoutsos et al., “Fabrication attacks: Zero-overhead malicious modifications enabling modern microprocessor privilege escalation,” IEEE Trans. Emerging Topics in Computing
- C. Domas, Hardware backdoors in x86 cpus, 2018, <https://i.blackhat.com/us-18/Thu-August-9/us-18-Domas-God-Mode-Unlocked-Hardware-Backdoors-In-x86-CPUs-wp.pdf>
- Project: [rosenbridge](https://github.com/xoreaxeaxax/rosenbridge), 2022, URL <https://github.com/xoreaxeaxax/rosenbridge>

- N. G. Tsoutsos et al., “Fabrication attacks: Zero-overhead malicious modifications enabling modern microprocessor privilege escalation,” IEEE Trans. Emerging Topics in Computing
- [C. Domas, Hardware backdoors in x86 cpus, 2018](#)
- [Project: rosenbridge, 2022](#)

A novel menace raised from *legal* features

Modern CPUs exploit advance features to increase performance:

- performance counters
- cache hierarchies
- out-of-order execution
- speculative execution

...also expose the system to novel threats

Transient Execution Attacks

Like traditional Side-Channel Attacks, also TEAs steal information but...

...TEAs do not rely either on any SW bugs or on physical access to the attacked system

They do not exploit software or hardware bugs; they just exploit *nominal features of the processor*.

Transient Execution Attacks

Spectre poisons the branch prediction and the speculative execution to force the processor to execute instruction sequences that should not be executed

Then, by exploiting timing measurements on the cache accesses, the attacker can retrieve a secret from the cache of the attacked program without having physical access to it



Kocher, Paul, et al. "Spectre attacks: Exploiting speculative execution." Communications of the ACM 63.7 (2020): 93-101.

Transient Execution Attacks

Meltdown exploits out-of-order execution to break the isolation between the memory spaces of user applications and of the operating system

It allows the attacker program to access any memory space, thus, stealing secrets from the operating system or other users applications



Lipp, Moritz, et al. "Meltdown: Reading kernel memory from user space." Communications of the ACM 63.6 (2020): 46-56.

Transient Execution Attacks

TEAs are already considered a serious threat

Several IT companies published official security advisories about TEAs*



*meltdownattack.com

Speculative Execution Attacks: Meltdown and Spectre are attacks that leverage speculative execution, out of order execution, caching and other architectural performance enhancements to break isolation and other security policies.

meltdown: enables unauthorized processes to read data from any address that is mapped to the current process's memory space. It exploits a *race condition* where unauthorized process attempts to access privileged data. The attack then uses a cache side-channel attack to determine contents of the data.

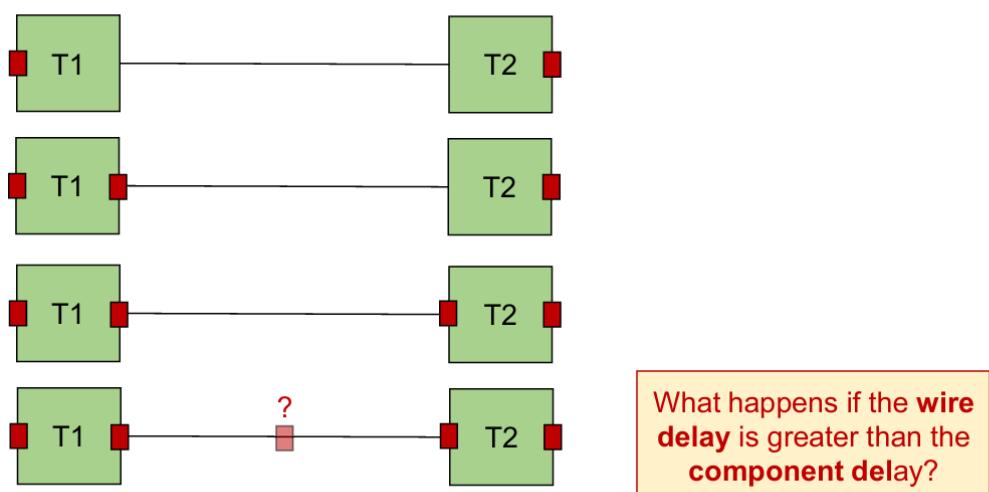
spectre: vulnerability that tricks a victim process to leak its data. Many processors use *speculative execution* by *branch prediction*. Spectre uses the fact that this speculative code leaves traces of its execution in the

cache, whose information can be extracted using side-channel attack. Spectre trains a branch predictor to make a wrong decision and then wraps code that should not be executed in a condition. The code is *speculatively executed* since the branch predictor is wrong. [\[source\]](#)

6 - Latency Insensitive Design

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** (the **minimum of the clock frequencies**) of the components (reg to reg)



Communication Issues

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

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

- Inevitable dominance of wire delay

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

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

In current technologies, transistors are dramatically scaling while wires are not scaling as fast as transistors, so while delays introduced by wires were negligible in older technologies, now they're not anymore. Wiring in modern complex circuits is huge with respect to the logic, so the delays related to wiring have to be considered. This point is in conflict with the plug-and-play composition of chips that we would like to pursue.

The target is a block-based design and dealing with the fine tuning of wires would be avoidable, which is a physical aspect of the hardware layout.

It's not acceptable to be uncertain of the timing of the circuit until the finite layout is ready, a way around this problem must be found and it already exists, it's **Latency Insensitive Design**.

System complexity

Guaranteeing synchronization (correct timing behavior) in a HSoC is an extremely difficult task!

- Number of components
- Different “frequencies” among components
- Different “speeds” among the various interactions
- Different “protocols” among various interactions

Need for a global «protocol» that is insensitive to the components in the HSoC and to the interactions among the components

Modern HSoC have lot of different components, different *clock domains* and even different *voltage domains*, different speeds (in the sense of the amount of data required by the interaction of components). This complexity can't be managed at layout level, it is too complex and would require too much time.

Latency-Insensitive Approach

Could I solve the problem by just introducing buffer between data producer and consumer?

We want to solve the problem with a *systematic design approach* and not *hardware design approach*, I want something that is at architectural system level to solve the timing complexity.

One possible approach is *introducing buffers between components*, some FIFO in between them to store data and letting the consumer have it when it's ready. The problem is this solution isn't a feasible solution, because if the consumer is slower than the producer, an infinite FIFO would be required in order to make the system able to run for (ideally) infinite time, while if the producer is slower than the consumer, after some time the FIFO would be empty and it would be useless, so would be a waste of area and power.

Latency-Insensitive Approach

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

This approach is very powerful because it permits us to have a “plug and play” system and introduces strong standardization. At the early stages of the design, I want to be allowed to consider just the component functional constraints.

Latency-Insensitive Approach

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

Problem definition:

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

- *Globally Asynchronous/Locally Synchronous (GALS) design*

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 Design: it is a methodology to design complex systems by assembling predesign components. [\[source\]](#) With latency insensitive design, global behavior will be asynchronous, in the most general case every component will have its own clock signal. The point is that I'm not interested anymore in knowing the clock frequency of every single component, there will be a dedicated structure that will interact with all the components and will manage the data input/output based on events rather than synchronous clock. In this way I'm not interested anymore in skew/delays/clocking issues.

LIDs are synchronous distributed systems and are built by composing functional modules that exchange data on communication channels according to a latency-insensitive protocol. The goal of the protocol is to guarantee that a system is composed of a functionally correct modules behaves correctly *independently of channel latencies*. [\[source\]](#) This increases the robustness of design implementation because any delay variations of a channel can be recovered by changing its latency while the overall system functionality remains unaffected.

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**

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

- Components are still synchronous circuits
- Legacy components

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

- **Arbitrary delay is a multiple of the clock period**

*Recall: **Totally asynchronous systems**: systems without clock, the problem is that they need lot of protocols and handshakes to make communication possible and let's say that *history demonstrated that fully asynchronous circuits are not viable*. What we're interested in with LID are *events and interactions*, all the circuits are synchronized so the behavior of components is dominated by clock but at a higher level what we do care about is *an event every x clock cycles*.*

Latency-Insensitive Design

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
- Not suitable for real-time systems (for which timing is a functional property!)

It introduces the following concepts:

- A **relay station (RS)** is a module that is inserted wherever it is necessary to tolerate delays
- Each RS introduces one **stalling event** (non functional) and one **void event** (non functional)
- A module receiving a stalling event as input emits stalling events as outputs at the next cycle

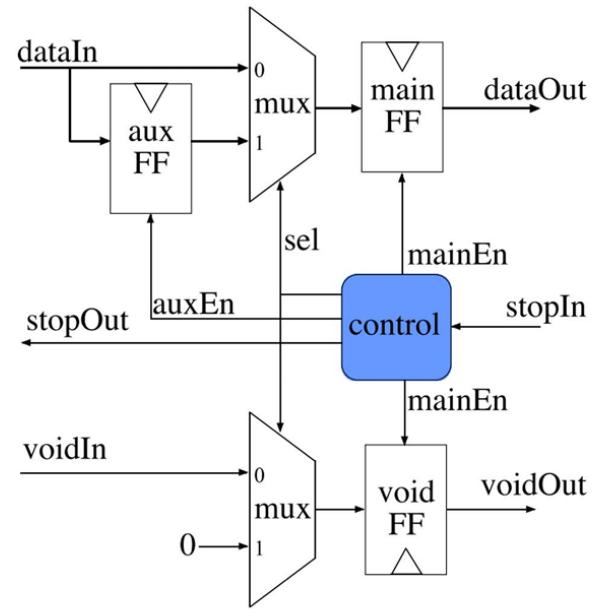
The single components will be verified under the timing point of view but the strong assessment with LID is that *the interaction between components will be guaranteed by the design even if we have delays or interferences*.

We're making our system tolerant to arbitrary delays. What we will introduce is the concept of *relay station* and the *void and stalling event*, so functional informative data will be exchanged but also events about the readiness of receiving and sending data.

Latency-Insensitive Design

- 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



The consumer sends the *stop in* and the *stop out* while the *void in* and *void out* will be implemented by the producer.

With this station between consumer and producer with data we're *allowing them to work with completely different clock periods*, the consumer can be superfast and the consumer super slow and vice versa and the system still works correctly. Furthermore, we're achieving a plug and play platform, with respect to a completely synchronous system.

backpressure: mechanism that lets a downlink shell to temporarily stop its production of valid tokens.

void bit: bit used to distinguish void from valid tokens and a stop bit to implement backpressure.

Latency-Insensitive Design

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**

Traditional synchronous system:

- Layout obtained by standard Place&Route tools

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

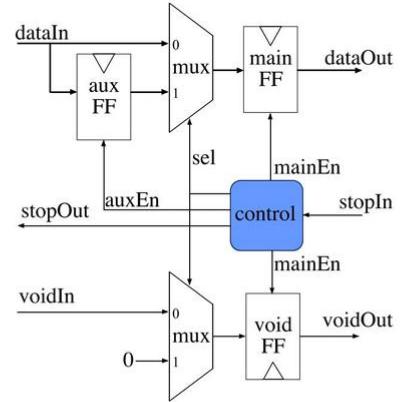
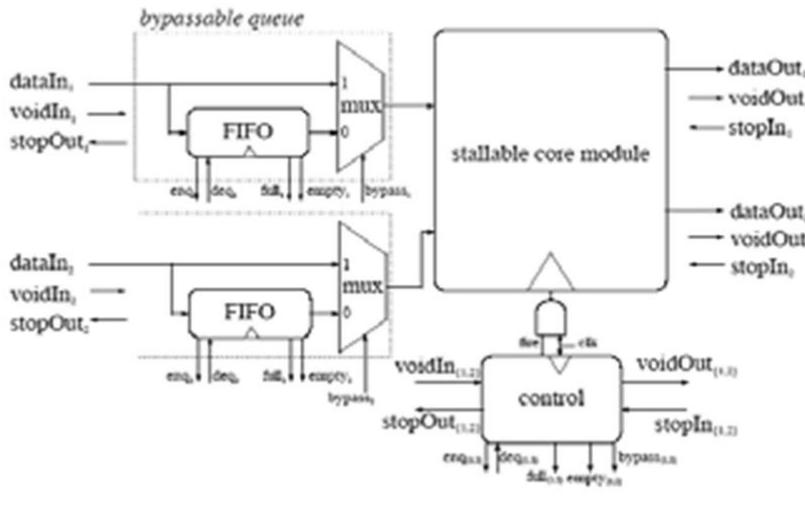
- Necessary number of relay stations inserted into each critical channel

With this approach of relay station, I want a plug and play approach, so we buy a component from one producer, another one to the other and then what we need is a shell in order to make it

The usage of relay stations we can implement a “plug and play” system, we buy a component from one producer, another component from another one and we can make them work together even if they require different clocks and have different timing constraints.

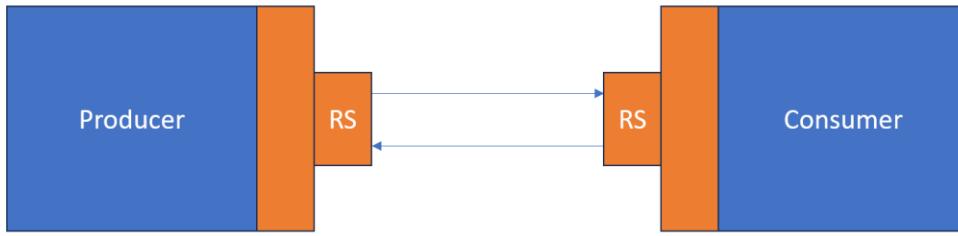
Latency-Insensitive Design

Protocol that governs the exchange of information in a patient system



Latency-Insensitive Design

One-to-one communication



What if the consumer is slower than the producer?

What if the consumer is faster than the producer?

If the consumer is slower than the producer?

The “stop signal” is risen, backpressure to the produced is applied and we slow down the producer without fully halting it.

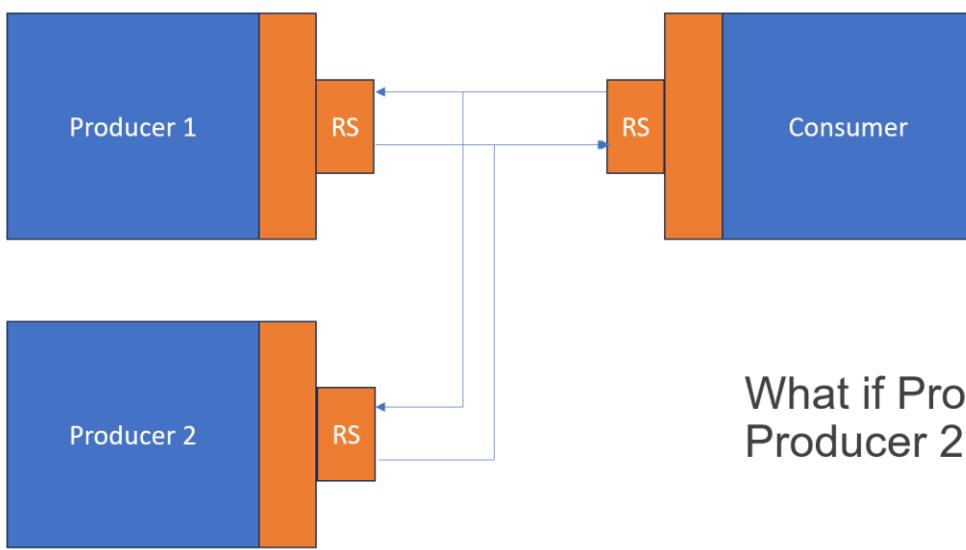
If the producer is slower than the consumer?

If the producer is faster than the consumer, the “void signal” is risen so it knows that the data that the consumer is consuming is dummy data.

Oss: the relay station is asynchronous with respect to both the devices. It's clocked but it's totally asynchronous with respect to the producer and the consumer. It is being implemented to decouple synchronization, it would be dumb to synchronize it with the clock of the producer or the one of the consumer.

Latency-Insensitive Design

Two-to-one communication



What if Producer 1 is faster than Producer 2?

What if Producer 1 is faster than Producer 2?

- consumer receives data from P1 and then waits for P2 data → this is an unoptimized way
- when P1 is producing good data, P2 is giving void → the consumer will backpressure P1

The shell will trigger the consumer as soon as the data coming from both producers arrive. ← example of logic that can be implemented by the shell.

The only triggers to communications and interactions are *stop* and *void*.

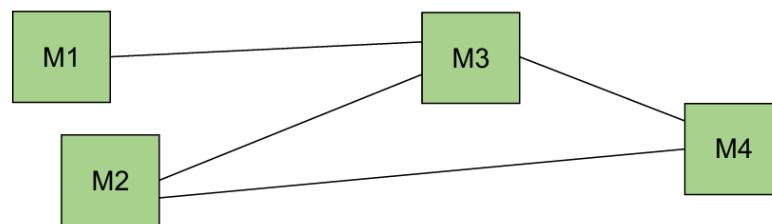
What if Producer 1 and Producer 2 are faster than the consumer?

This would be an avoidable case, in fact if this is the situation, something in the design has to be changed to slow down both the producers or to make the consumer faster.

In any case since we have independent of delays, the effect would be that we would backpressure both P1 and P2, then make them go faster when the consumer is ready and then backpressure again.

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

LID vs DVFS

The latency insensitive design paradigm deals with transactions among components at the SoC level

- Irrespective of the abstraction level
- Irrespective of the technology (ASIC, FPGA...)
- Irrespective of low-level details (working frequency, voltage level...)

Dynamic Voltage and Frequency Scaling deals with physical quantities (voltage and frequency) at the single component level

LID and DVFS are approach for optimization, both can be used but they work at totally different levels: LID is at *architecture level* while DVFS is at *hardware level*.

DVFS: it's the dynamic control and scale of voltage and frequency. If I reduce the frequency I'm imposing "less switching" so less dynamic power consumption, static power consumption remains the same. The higher the frequency, the higher the performance, the higher the power consumption.

For voltage, the higher the voltage the faster the circuit (I have and higher currents) but obviously I have more power consumption.

Voltage and frequency are interconnected, if I lower the voltage, I'm slowing down the combinational path and consequently at a certain point I have to also reduce the frequency.

Higher performance → Higher power consumption

DVFS is at single component level, at single component domain, while LID is at architecture level.

LID vs DVFS

I can use information coming from LID to fine-tune DVFS

- If a data producer is stalling too frequently
 - I can slow down it
 - ...or I can accelerate the receiver

Similarly, if the data producer is sending void messages too frequently

- I can accelerate it
- ...or I can slow down the receiver

Implementation Concepts (i)

- **Channels are point-to-point unidirectional links (irrespective of the underlying communication infrastructure)**
 - Source/Sink Modules
 - 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

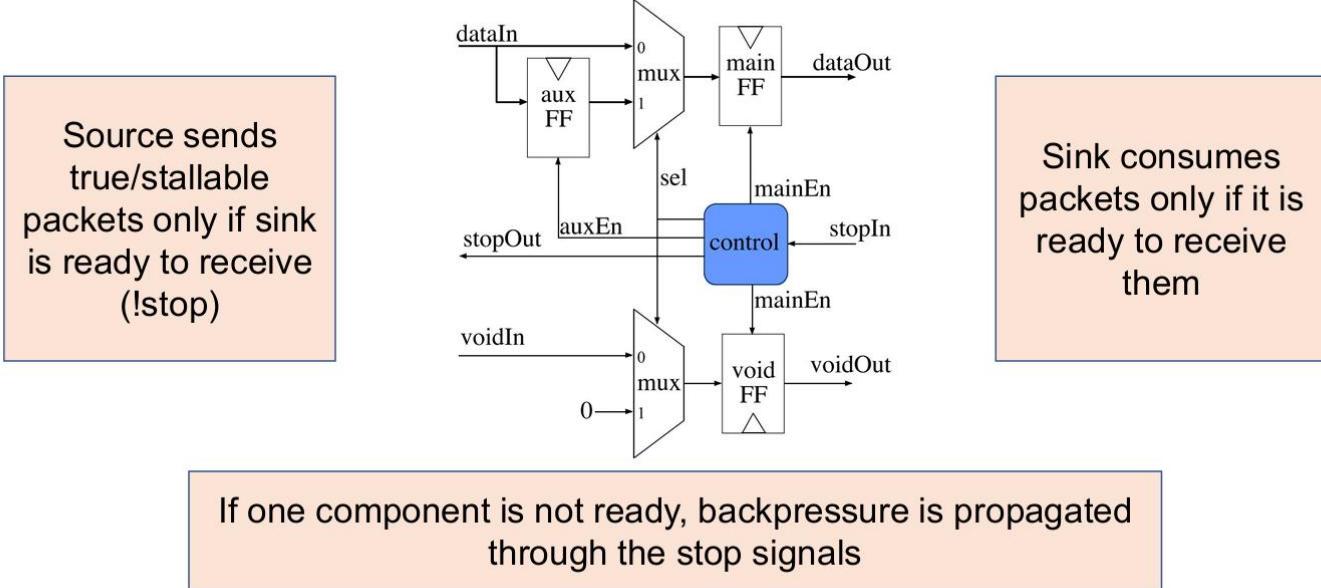
- Puts true packets (void0) or stalling packets (void1) on the channel

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

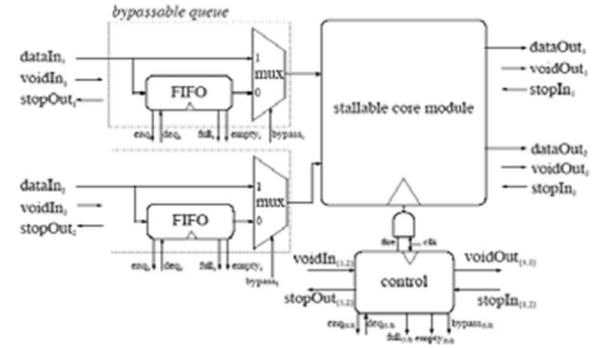
In some systems, **void (0/1)** is replaced by **valid (1/0)**



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 them to M and activates the computation
- Gets results from M when completed
- If no stop flag is received, sends result



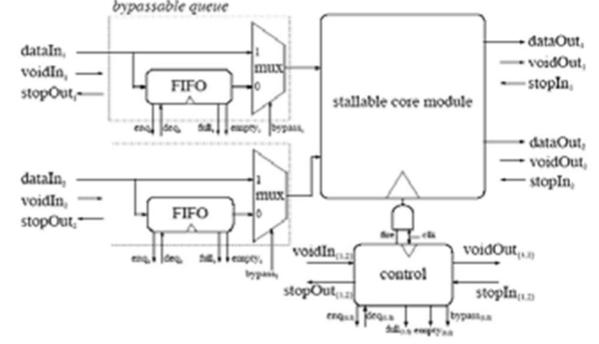
Components become *stallable* with this design approach.

Shell

Guarantees input synchronization and data propagation

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

Allows to add components in a plug&play fashion



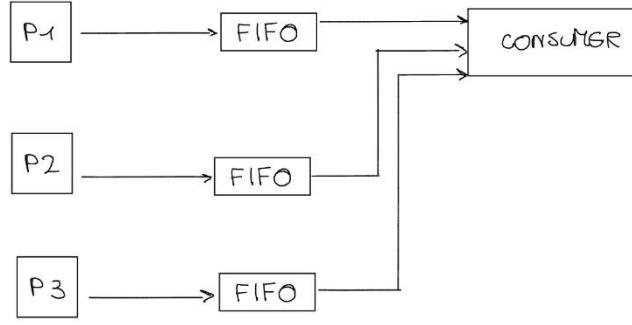
LID and FIFO Buffers

A First-in First-out (FIFO) buffer can 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



We can stall the producer after the FIFO is full, we have a sort of feedback.

Basically we're introducing a lot of pipeline stages.

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
- 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*)

Compositional System Level Design

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

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

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

AXI uses LID.

Obviously, LID is not fully optimized, we must go to much lower hardware level to go as optimized as possible, obviously full custom SoC we would have a much-optimized circuit.

7 – HW/SW Codesign

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:** System-level HW/SW optimizations
 - **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

Today SoC design is still a custom process. Reuse of IP cores/FPGAs, components that are used are standardized, their production is still a custom design.

When we go to the verification part of the SoC, still is a full custom design. We'll focus on HLS, it is not used in industry, but we still need the *medium-term* part, the optimization is still in custom. I have my platform which is the best choice, how can I automatically map the parts of the circuit. CAD tools are methodologies, algorithms and GUIs to accelerate and enable the electronics design.

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

Cadence and Synopsys produce tools for circuit design, while Xilinx (AMD) and Altera (Intel) are the leaders for FPGAs design.

Efficient RTL Architectures

- HLS can now synthesize complex applications
 - **Resource requirements** can become a limitation: you do not have enough space/resources for all components
- **Resource sharing** (e.g., FPGA-based design) is a well-known and adopted technique to control the area occupation
 - **Achieving the minimum number of functional units or registers is not always the best solution...**
- **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*)

Research is trying to further explore also for *optimization automation*

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** (an all-pass phase equalizer)
 - 11 additions and 17 multiplications
- Alternative solutions manually generated through *synthesis directives* in Xilinx Vivado HLS scripts
 - Minimum target 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
Shift Reg.	117	36	36	36	36
Critical Path	4.534ns	7.055ns	7.665ns	7.906ns	7.871ns
Clock cycles	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, but...
 HLS is not always explainable!

Columns

1. Infinite number of multipliers and adders → we want to fulfill the requirement but we want to achieve the same computation with the same constraints, but we limit the resources!
2. So the number of multipliers has been set to 4 to force the HLS to reuse the multipliers. → we have a small critical path increase (but we're still in the 10ns constraint) but we reduced the size (we use same clock cycles, we use more LUTs but much less FFs and shift reg)
3. We also constrained the number of adders → we still haven't modified the code, we're just constraining the number of components that the HLS can use
4. If we tweak the numbers of adders and we obtain better results

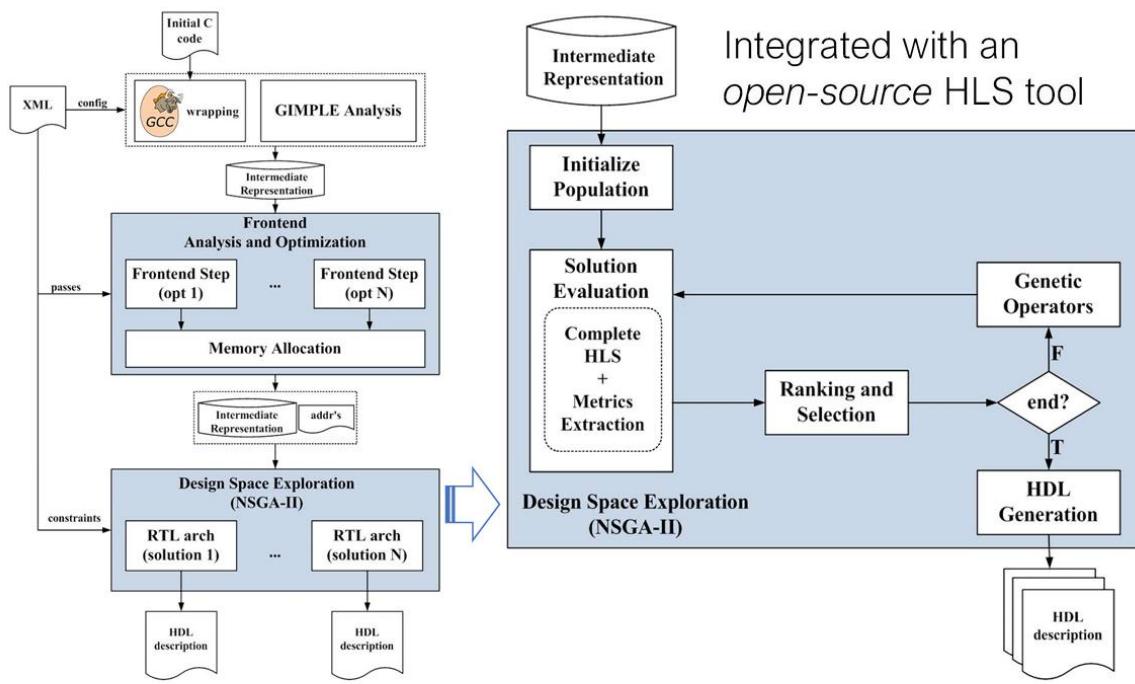
We're not internals, there's no logical reasoning with which we can understand *a priori* how the HLS engine is working and how to set the correct constraints to obtain our solution, we must use a “trial and error” approach.

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 tight 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)

Design Space Exploration: explore the many solutions obtained because there's no global optimum or if there is, it is unknown how to get there, so a trial-and-error approach must be adopted.

Fine-grain DSE for HLS



Genetic Hardware Space Design Exploration: machine learning methodology that tries to emulate the biology of a set of individuals, so like in the process among generation, the quality of the genies is increasing or increasing is coding solutions into individuals and then implement crossover and mutation operations to

obtain children from parents, where presents are old solutions, children are new solutions with the idea that by the idea that exploring mutation and crossing we have an improvement of the solution, generation by generation.

WE HAVE THE PARAMETERS p_1, p_2, p_3

C	P_{1x}	P_{1y}	P_{1z}
---	----------	----------	----------

WE OBTAIN THE SOLUTION \pm

implemented like the value of every single parameter

F : FITNESS FUNCTION

needed to evaluate how good is my solution
 VIABLE solution
 Non VIABLE solution
 it gives a number

C : CROSS OVER OPERATION

function that takes two solutions and tries to combine the two good functions of them

so i have a "child" that is better than the previous solution

M : MUTATION OPERATION

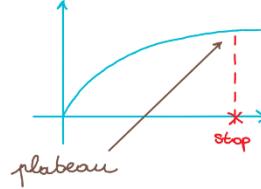
i randomly decide that a particular gene has to be changed

EXPLORATION IMPLEMENTATION

with cross over i am moving towards a minimum but we don't know if its a GLOBAL or LOCAL minimum

MUTATION MAKES US MOVE

we stop when the F function doesn't move over a certain value



solution N

The complex part is the computation and choice of the fitness function, and we want to produce a design space where we can implement different solutions, that's why also the synthesis needs to be optimized otherwise we wait for a very long time because for every solution with every tweak, every single point of the graph needs the synthetization, the computation of the constraints, the verification and the tweaking of the parameter!

We're going to drive the HLS to implement our design space exploration. We're automating the design space exploration but the problem is that the design space exploration is going to take long.

What we will do is apply and exploit models of the solutions in the DSE. We're not evaluating the DSE solution on the model and not on the finite device.

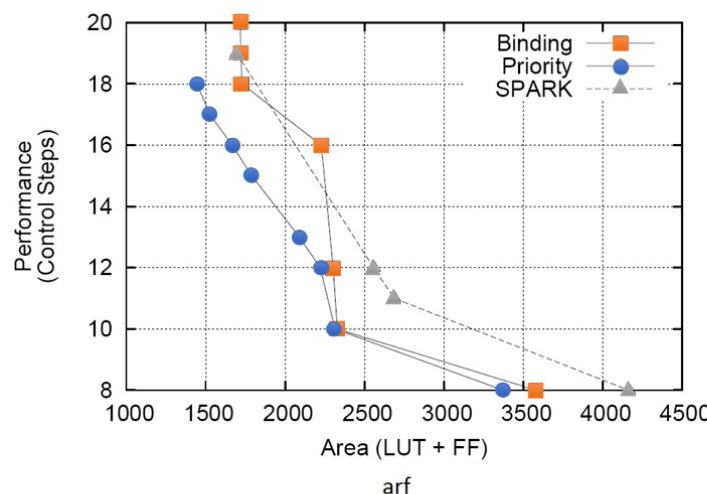
Applied to Simple Case Studies

- Traditional benchmarks for HLS (datapath synthesis)
 - ARF, EWF, DCT, DIST
 - comparison with a manual HLS 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)
- Area optimization (*binding*), scheduling optimization (*priority*), manual optimization (**SPARK**)

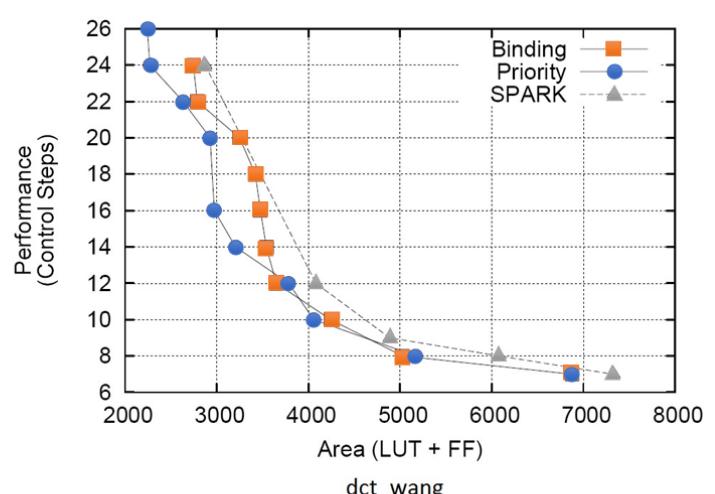
- SPARK: ambient for manual optimization

The parameters that have been analyzed are

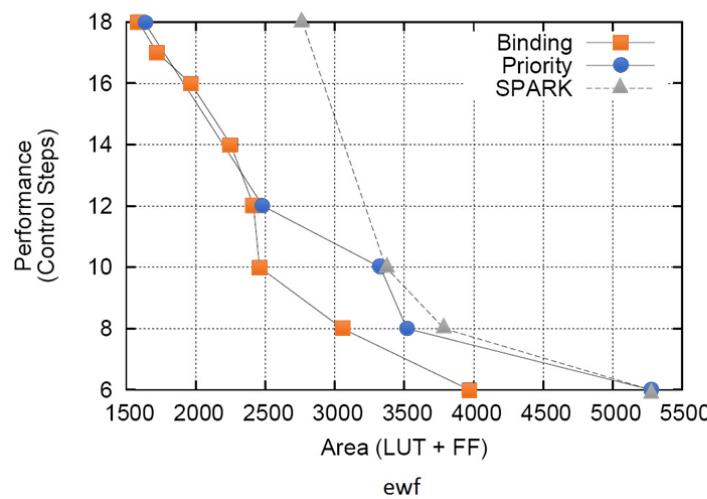
- Area optimization (LUTs, FFs...)
- Timing optimization, number of clock steps



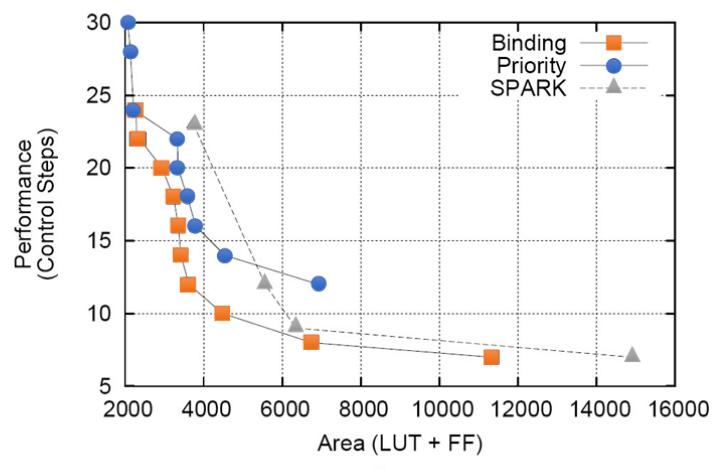
arf



dct_wang



ewf



dist

For all the solution we can't find a point where I both have AREA and TIMING implementations. What we're seeing is that manual optimization is not obtaining the best results.

What we cannot say is if the points are a local minimum or the global minimum of the constraints we are evaluating, we do not have a precise criterion to stop the evaluation.

Lesson Learned...

- **Fine-grain methods** systematically outperform methods based on **manual 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

Task Partitioning

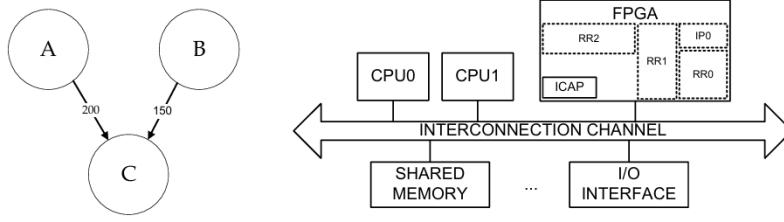
- **Definition:** The **partitioning problem** is to assign
 - **n objects** $O=\{o_1, \dots, o_n\}$ to
 - **m blocks** (also called **partitions**) $P=\{p_1, \dots, p_m\}$
- such that
 - $p_1 \cup p_2 \dots \cup p_m = O$
 - $p_i \cap p_j = \emptyset$ for all $i \neq j$, and
 - cost $c(P)$ is minimized.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

Given all the possible solutions using HLS, HLS optimization, automatic optimization of the HLS process and choice of parameters, the first short-term point of the roadmap can be fulfilled, it is possible to automatically optimize the single component into an architecture. How to compose my architecture, how to decide which are the best components and which is the best partitioning and mapping of the tasks of the high-level functionality into the underlying component (FPGA, ASIC...).

The question at which the designer must answer is: given a particular platform, with a CPU, a GPU, a FPGA, which is the best way to partition the applications and to schedule and map them onto the architecture?

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



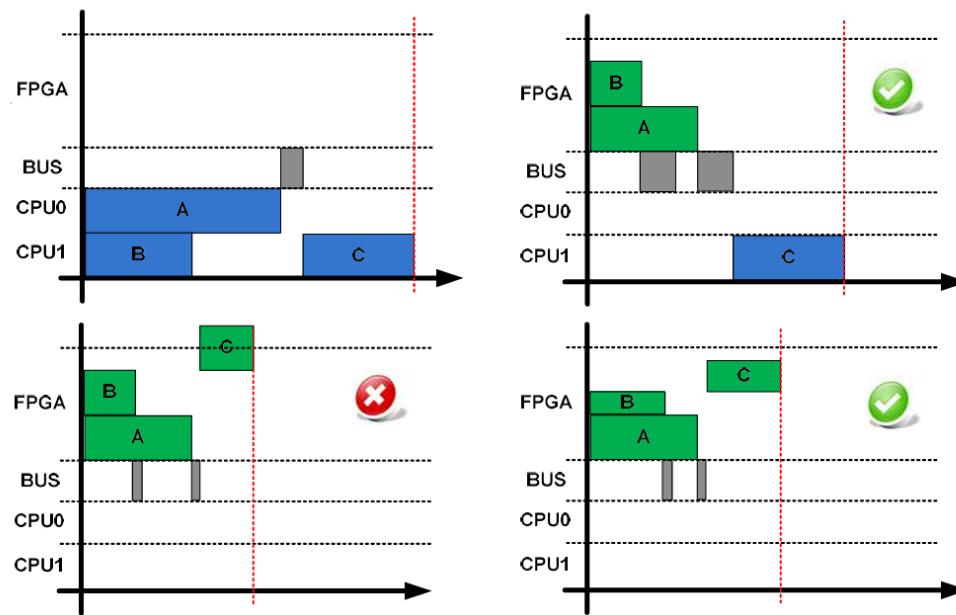
We have three high-level tasks, A and B are producing tokens, whatever tokens mean, and C requires 200 chunks of data and 150 chunks of data from B in order to be running and upwards we see the architecture.

We're looking at how to map and schedule tasks on top of an existing architecture. The architecture constrains, two CPUs, an FPGA and a shared memory.

A first solution might be run task A in CPU0, task B in CPU1, task C is CPU1 but before running task C the data produced by task A must be moved through the BUS to CPU1, we're not taking into account memory access etc. so task C can start as soon as task B finishes and the data is transferred to CPU1. This would be a full software approach, figure on the top left.

Another approach is accelerating task B and task A into the FPGA, so let's say that the duration of task A and B are shorter than the blue versione because we have an hardware accelerator but the data transfer takes longer because we are moving data from an FGPA to an external circuit, so we need external arbitration between the components. It's faster than full software. We're executing everything fine and fast.

ross: the top dashed line is the maximum load that can be applied to the single component



In the solution on the bottom left we see that the computation is very fast and the data transfer also, because everything is in the FPGA, but the problem is that we go over the maximum resources of the FPGA, so it's not a feasible solution.

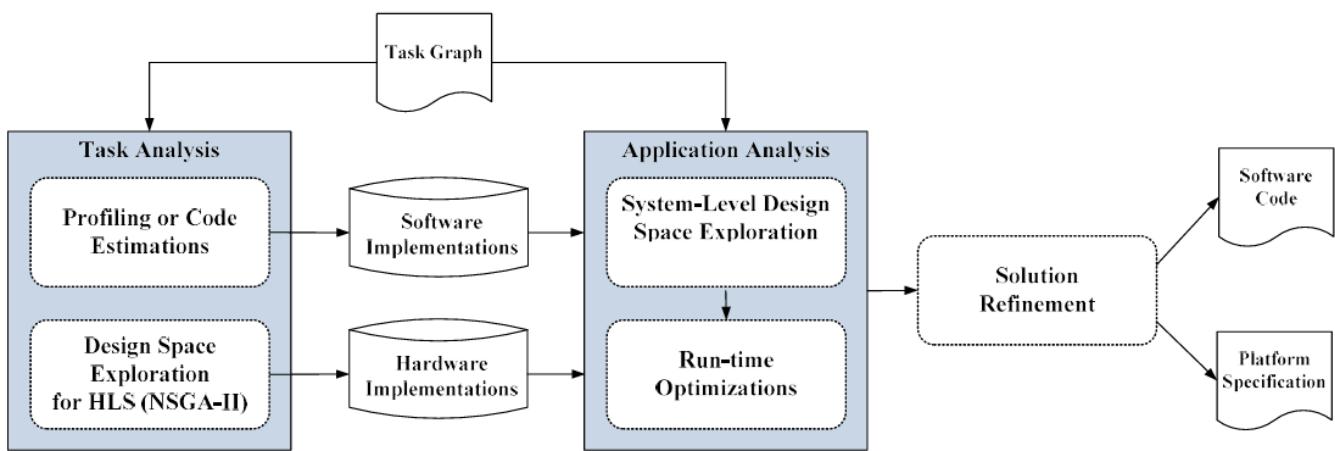
On the bottom right we slow down the computation of C but still is faster than the previous ones.

So when we're mapping and scheduling a task on an existing platform, we need to *know a lot of details*, not only on the underlying platform (how many resources does the FPGA has? Which is the working frequency of the processor? Which is the speed of communication?) but also on the *high-level functionality that can be run on hardware or in software*. What we need is an even more complex software environment which is able to implement an *optimal model or architecture, profile the requirements of our application* (like how much data is required or how much data does it produce, which are the interconnections among the applications) and based on this profiling we obtain software and hardware implementations, several solutions for our hardware platform and several solutions for our software implementation and then we need some environment that is still not there (this is a research field) where we can profile and study the quality of our mapping in scheduling over the platform that we have. We try to identify a local optimum.

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

Design space exploration can also be applied to scheduling and mapping.

Future Work: Automatic Partitioning

- Roadmap for next-generation computing systems
 - Short term: Development of efficient accelerators
 - Medium term: System-level HW/SW optimizations
 - **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

8 – Dependability

The added information is taken from [this](#) paper

What dependability is?

The ability of a system to perform its functionality while exposing:

- Reliability
- Availability
- Maintainability
- Safety
- Security

Dependability: system property that integrates such attributes as reliability, availability, safety, security, survivability, maintainability. It's the ability of a computing system to deliver service that can justifiably be trusted.

The basic attributes of dependability are

- **availability:** readiness for correct service
- **reliability:** continuity of correct service
- **safety:** absence of catastrophic consequences on the user(s) and on the environment
- **confidentiality:** absence of unauthorized disclosure of information
- **integrity:** absence of improper system state alterations
- **Maintainability:** ability to undergo repairs and modifications and easy maintenance
→ confidentiality, integrity and availability together are **security**

Why dependability?

Functional Verification

A lot of effort is devoted to make sure the implementation

- matches specifications
- fulfills requirements
- meets constraints
- optimizes selected parameters (performance, energy, ...)

Nevertheless, even if all above aspects are satisfied ... things may go wrong

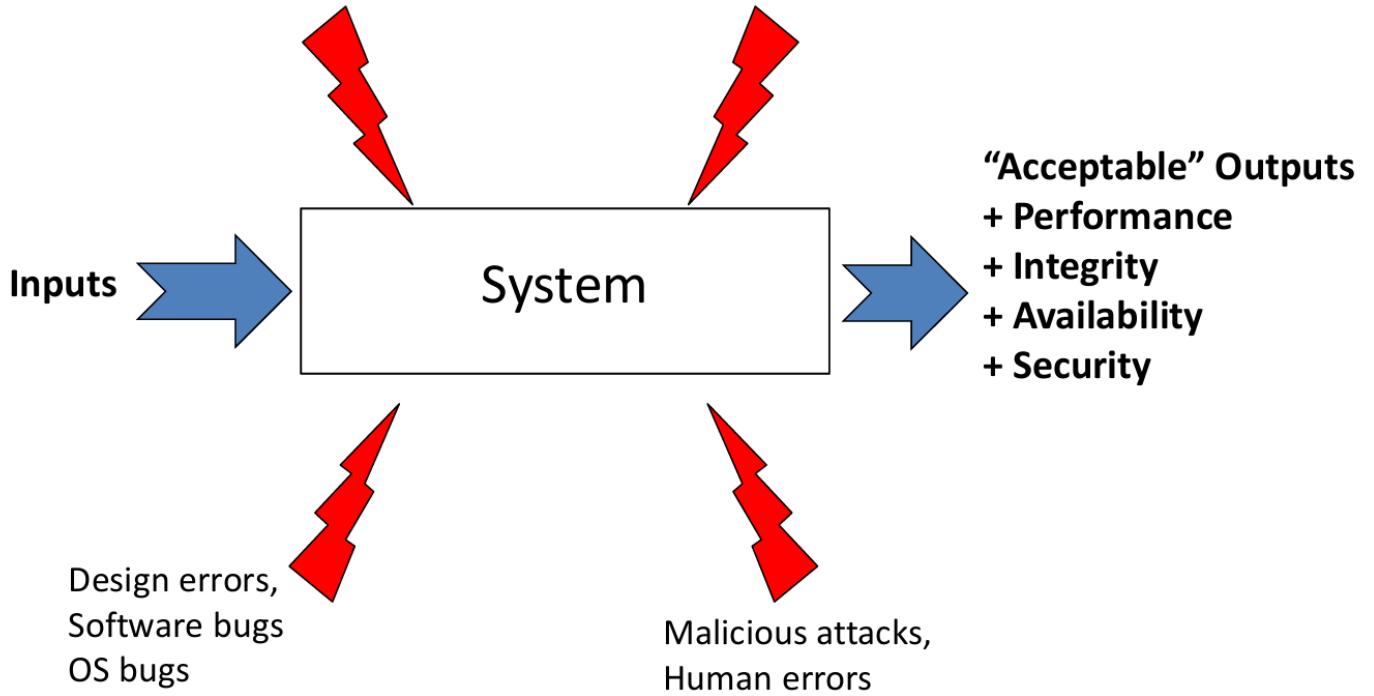
► systems fail

systems fail ... because something broke

Why dependability?

Defects, Process variation,
Degraded transistors

Radiation, Noise



It's important to remember that a single system failure might affect a large number of people

Why dependability?

Industrial standards require it:

- ISO 26262 for automotive
- CENELEC 50128 (SW) and 50129 (HW) for railways
- RTCA DO-178C (SW) and DO-254 (HW) for airborne
- ESA ECSS-E-ST-40C (SW) and ECSS-Q-ST-60-02C (HW) for space

Design standards: every single step has requirements for implementation and specific constraints to respect.

Ex: in the ISO 26262 has specific requirements for power dissipation, how to refine requirements, how to implement behavioral and RTL description, in order for it to be certified. If the system is not certified it can't be implemented for user ended user.

Ex: when compiling a software we have to use certified compilers.

Companies are forced to use standards and at the same time any company is trying to push the standard in their own way of implementation. Standards are time consuming but standards are very detailed and is interesting read them and understand how to implement them into your own solution.

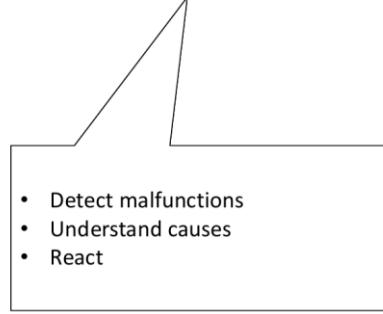
When to think about dependability?

Both at **design-time** and at runtime

-
- ```
graph TD; A[Both at design-time and at runtime] --> B["
 - Analyse the system under design
 - Measure dependability properties
 - Modify the design if required"]
```
- Analyse the system under design
  - Measure dependability properties
  - Modify the design if required

## When to think about dependability?

Both at design-time and at runtime



*Ex:* we must implement a processor that has to be implemented into a satellite, then we have to respect *ex* the radiation requirements and we test our hardware in such conditions and we verify how many faults, incorrect results we obtain in such conditions.

So we analyze and then modify our system based on the results of the previous test. Often also fault simulators are used to understand how weak and where our system has problems. We perform *ex* radiation test and then we emulate the faults. Then we understand where we have fault, which component has faulted and how we can handle the event.

### Where do we apply dependability

Safety-critical systems: a failure during operation can present a direct threat to human life

- aircraft control systems
- medical instrumentation
- railway signaling
- nuclear reactor control systems

*Safety critical systems* are systems whose failure is going to harm or damage human lives or the environment *ex* trains, nuclear power plants, automotive related applications...

## Mission-critical and safety-critical systems

Mission-critical systems: a failure during operation can have serious or irreversible effects on property and finance

- Satellites
- Surveillance drones
- Unmanned vehicles

*Mission critical systems* are systems whose failure is not going to harm human lives but the critical part is the mission himself.

*ex* consider the drone that is going to land on mars and going to send pictures, then its mission is the objective himself. If a fault halts the systems or makes it uncontrollable for one minute I'm going to lose the control and the mission is going to fail.

**Not critical systems** that are systems that are not critical nor for environment nor for human life but still needs to be reliable.

Every single aspect of this stack has to be reliable, so

### Anatomy of the scenarios

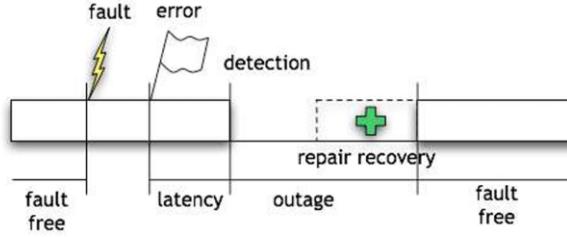
- the nodes
  - computing systems
  - sensors and actuators

- the communication
    - network
- Everything has to work properly for the overall system to be working

- the cloud
  - data storage
  - data manipulation

### Fault, Error, Failure chain:

| Term    | Description                                                        |
|---------|--------------------------------------------------------------------|
| Fault   | A event within the system                                          |
| Error   | A deviation from the required operation of the system or subsystem |
| Failure | The system fails to perform its required function                  |



- **Fault:** occurs in the system due an internal/external condition in the system.
- **Error:** event that occurs because a fault happened and *has propagated* within the system. Part of the system state that may cause a subsequent failure.
- **Failure:** the system fails because it couldn't handle the *fault*. A failure is an event that occurs when the delivered service deviates from correct service, so the system stops delivering the system function. A failure occurs when an error reaches the service interface and the alters the service.

An example: a flying drone with an automatic radar-guided landing system

**Fault:** electromagnetic disturbances interfere with a radar measurement

**Error:** the radar-guided landing system calculates a wrong trajectory

**Failure:** the drone crashes to the ground

Oss: the user that applies a certain action is not to be considered in this evaluation, otherwise we're considering the user as part of the system.

Not all the faults become errors and not all the errors become a failure.

## Reliability terminology

**Not always the fault – error – failure chain closes**

example: a tele-surgery system

**Fault:** radioactive ions make some memory cells change value (bitflip) but the corrupted memory does not involve the video stream

**Error:** no frames are corrupted

**Failure:** the surgeon carries out the procedure

**Not always the fault – error – failure chain closes**

example: a flying drone with automatic radar-guided landing

**Fault:** electromagnetic disturbances interfere with a radar measurement

**Error:** the radar-guided landing system calculates a wrong trajectory, but then, based on subsequent correct radar measurements it is able to recover the right trajectory

**Failure:** the drone safely lands

So the error is non propagated (or is absorbed)

## Failure avoidance paradigm

So we can work at two levels

1. I can work with *failure avoidance*, so we put in place several cases that allow the system to avoid, by design, fault, errors and failures

Conservative design

Design validation

Detailed test

- Hardware
- Software

Infant mortality screen

Error avoidance

Error detection / error masking during system operation

On-line monitoring

Diagnostics

Self-recovery & self-repair

Infant mortality: accelerated stress test, burn out test... I expose the system to more radiation/heat/voltage with respect to the one with which is going to work and we stress test it to see how it performs at its boundaries.

We have to think and implement such systems like online monitoring, diagnostics and self-recovery and self-repair.

# Where to work

## Technological/components level

- design and manufacture by employing reliable/robust components
  - Highest dependability
  - High cost
  - Bad performance (generally devices from old generation)

I can work at single component or technology level, to avoid faults up to a certain probability, maybe I might use older but more resistant standard cells, the larger and slower the cell is the more reliable it is or maybe I could use *special purpose* packaging, by spending more I could have more resistant wires etc.

## Hardening of the functional units (ALU, fetch unit, ...)

- Space redundancy is mainly used (DWC,TMR)
- Arithmetic codes is a viable approach for specific functional units (E.g.: residual codes for ALU)

## Hardening of register files and memories

- Information redundancy (E.g.: EDC, ECC)

An example of application of such approach is the **Leon2-FT** produced by **Gaisler** for **ESA**: a SEU tolerant microprocessor where FFs are protected by Triple Modular Redundancy and all internal and external memories are protected by error correction codes or parity bits.

## architectural level

- integrate normal components using solutions that allow to manage the occurrence of failures

- High dependability
- High cost
- Reduced performance

} Depending on the adopted solution

So to implement dependability components that work normally must be integrated with systems to manage the occurrence of failures.

## Architecture-level hardening

Space Redundancy

The whole processor is replicated and its outputs are checked/voted

Some approaches:

- Fault detection
  - Lock-Step Dual Processor
  - Loosely-Synchronized Dual Processor
  - Watchdog processor
- Fault tolerance
  - TMR – Triple Modular Redundancy
  - Dual Lock-Step Architecture

Here we can see listed the main dependability implementations applicable via architecture-level hardening through space redundancy. The idea is

*replicating the entire processor and checking or voting on outputs to ensure reliability*

### Fault detection approaches

Fault detection techniques aim to *identify errors* during the process *but not necessarily correct them*. The goal is to detect inconsistencies so that corrective action can be taken.

## Fault tolerance approaches

Fault tolerance techniques aim to not only identify errors during the process but *handle and mask faults*, allowing the system to continue functioning even in the presence of errors.

# Architecture-level hardening | 2

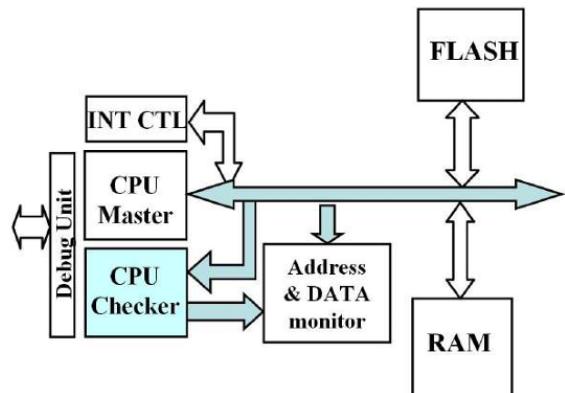
*Space Redundancy*

## Lock-Step Dual Processor

- Two processors execute the same code being strictly synchronized
- Bus and memories are protected with codes
- The interrupt controller is specifically designed with fault detection mechanisms

The solution is called fail-silent architecture (corrupted data are not emitted)

- Used as basic element for fault-tolerant distributed systems



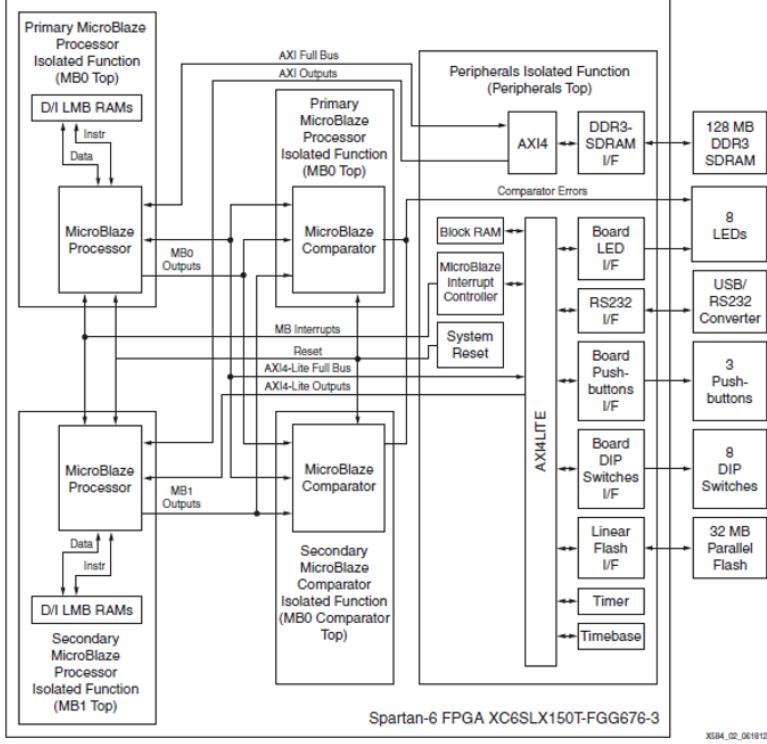
**Lock-Step dual processor:** two processors execute the *same instructions in perfect synchronization*, cycle by cycle. Both processors receive the same inputs at the same time and their outputs are continuously compared. Any mismatch in the outputs flags a fault.

- ✓ High accuracy in fault detection
- ✗ High resource and power overhead
- ✗ No fault tolerance, so once a mismatch is found the system must *halt* or *switch*, it cannot handle the fault

# Architecture-level hardening | 3

Space Redundancy

## Example of Lock-Step Dual Processor: Xilinx Dual Lock-Step Processor

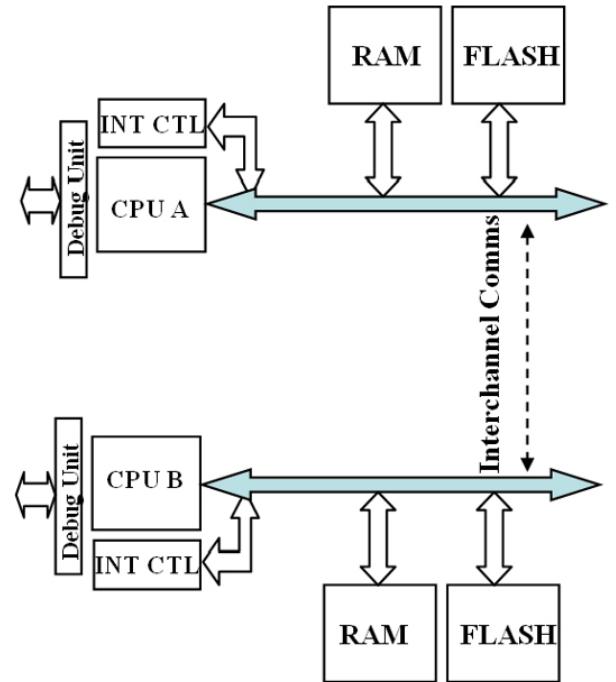


# Architecture-level hardening | 4

Space Redundancy

## Loosely-synchronized dual processor

- Two processors run independently
- The operating system is devoted to inter-process communication, synchronization and error detection



# Architecture-level hardening | 5

Space Redundancy

## Loosely-synchronized dual processor (cont.)

- Synchronization mechanisms must be protected with specific hardware/software mechanisms
- After an error detection, self-testing and sanity-check can be performed to identify the faulty component
- Two operational modes:
  - Critical applications: loosely-synchronized architecture featuring fault detection checks on synchronization
  - Non critical tasks: dual-core architecture

**Loosely synchronized dual processor:** two processors execute the same code but are *not tightly synchronized*, so they may be offset in execution by a few cycles and periodic comparisons are made. Differences in the results or state between the two processors indicate a fault.

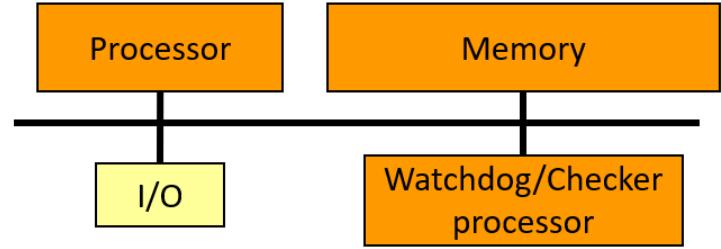
- ✓ Lower overhead than lockstep
- ✓ More flexible timing and reduced complexity
- ✗ Increased detection latency
- ✗ Harder to pinpoint the exact cause of divergence

# Architecture-level hardening | 6

Space Redundancy

## Watchdog/Checker Processor

- The watchdog observes the behavior of the processor and performs a high-level anomaly detection
  - Execution statistics different from profiled ones (branch misses, branch prediction, ...)
  - Data values or memory addresses out of expected ranges
  - Timeout expiration



**Watchdog/Checker processor:** the idea is implementing a secondary, simpler processor that monitors the main processor's activity. It checks timing and controls the flow of the main processor. It may also use software-based checksums or timeouts. If for some reason the main processor behaves abnormally (crashes, freezes or has wrong timing) it detects it and stops the execution.

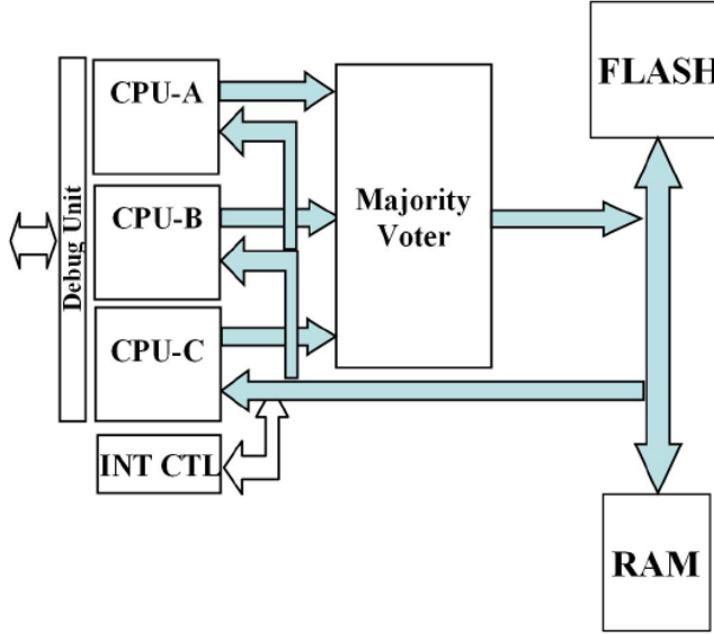
- ✓ it's simple and cost effective
- ✓ can detect control flow or timing faults
- ✗ it doesn't verify actual data or results
- ✗ has limited detection coverage

# Architecture-level hardening | 7

Space Redundancy

## TMR architecture

- It is a lock-step solutions with three processors



**Triple Modular Redundancy:** three processors execute the same task in parallel, a majority voter chooses the correct output. They all run in parallel, the outputs are compared at each step and if only one differs the majority output is taken as correct. It can obviously tolerate a *single* faulty processor.

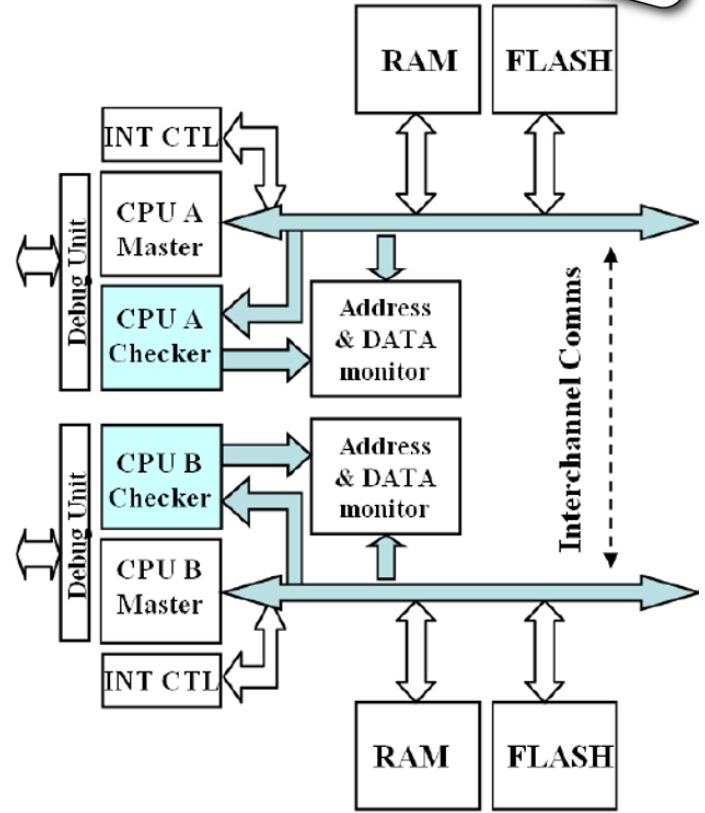
- ✓ Seamless fault masking, if a fault happens it is automatically resolved via the majority
- ✓ No disruption of outputs during faults
- ✗ High hardware and power cost, the overhead is at least tripled
- ✗ The voter circuitry must be fault-free and protected

# Architecture-level hardening | 8

Space Redundancy

## Dual lock-step architecture

- Two dual lock-step nodes are connected
- Each node is fail-silent
- Two operational modes:
  - Fault detection for non-critical tasks (each dual lock-step executing a different code)
  - Fault tolerance for critical tasks (both dual lock-step executing the same code)
- This is a simple distributed system



**Dual lock-step architecture:** it combines the lock-step fault detection with redundancy for failover. So the two processors in lock-step detect the faults and if a fault is detected the system switches to a backup lock-step pair or continues with the healthy unit. So the action is detecting faults in the primary unit and switching to the secondary.

- ✓ both detection and recovery
- ✓ suitable for high-reliability systems (like safety-critical systems)
- ✗ requires multiple processor pairs
- ✗ more complex control and synchronization logic

## Where to work

software/application level

- develop solutions in the algorithms or in the operating systems that mask and recover from the occurrence of failures

- High dependability
- High cost
- Reduced performance

} Depending on the adopted solution

## Mixed-level hardening

Approaches:

- Process replication/diversification
- Process re-execution
- Checkpointing
- Instruction-level hardening
- Codes

# Where to work

What do all solutions have in common?

- Cost
- Reduced performance

**You have to pay for dependability**

## Challenges

Find the best tradeoff between dependability and costs depending on:

- **Application field**
  - Is there a specific design standard?
  - Which degree of dependability is actually required?
  - Will failures cause human losses?
  - Which would be the monetary cost of a failure?
  - Would a failure have a “reputation cost”?
  - ....

# Challenges

Find the best tradeoff between dependability and costs depending on:

- **Working scenario**
  - Are there sources of faults (radiation, ageing, heat, vibration...)?
  - Which are the nominal working conditions (and the extreme ones) for the system?
  - Are there systems connected to my system?

Find the best tradeoff between dependability and costs depending on:

- **Employed technologies**
  - Are the cpu, memory, interfaces free from sources of failures?
  - Are the cpu, memory, interfaces tolerant to failures?
  - Which are the components most susceptible to failures?

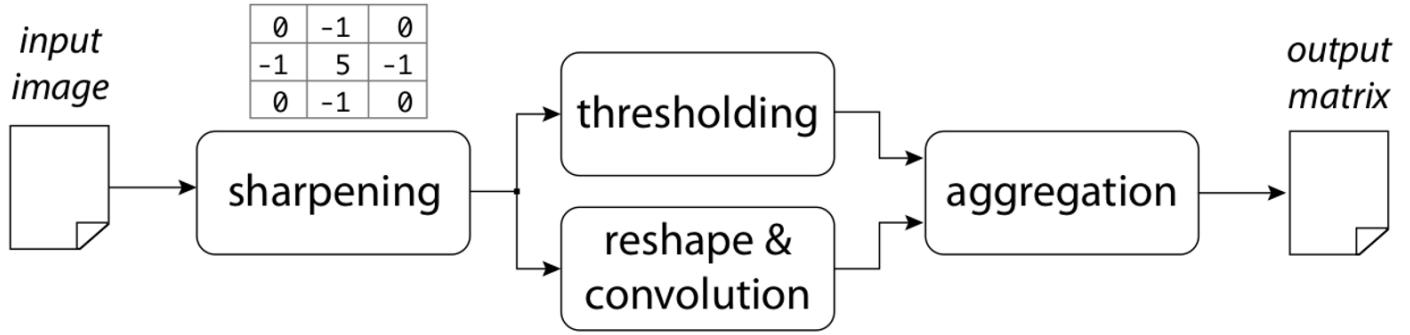
Find the best tradeoff between dependability and costs depending on:

- **Algorithms and applications**
  - Are the input of the application free of inexactness?
  - Is the algorithm tolerant to a certain degree of inexactness?
  - Can the application tolerate a certain “down-time”?

## 9 – Advanced Dependability

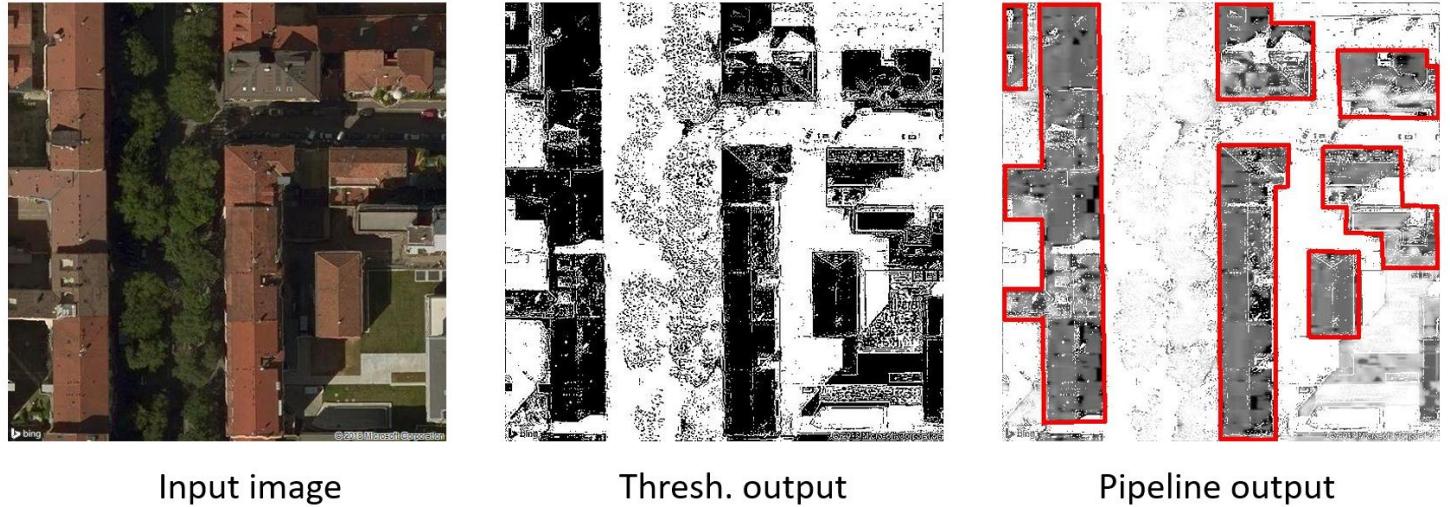
# IMAGE PROCESSING AND DEEP LEARNING

Image processing applications: a number of filters executed in a sequence



# IMAGE PROCESSING AND DEEP LEARNING

An example: building identification in aerial pictures

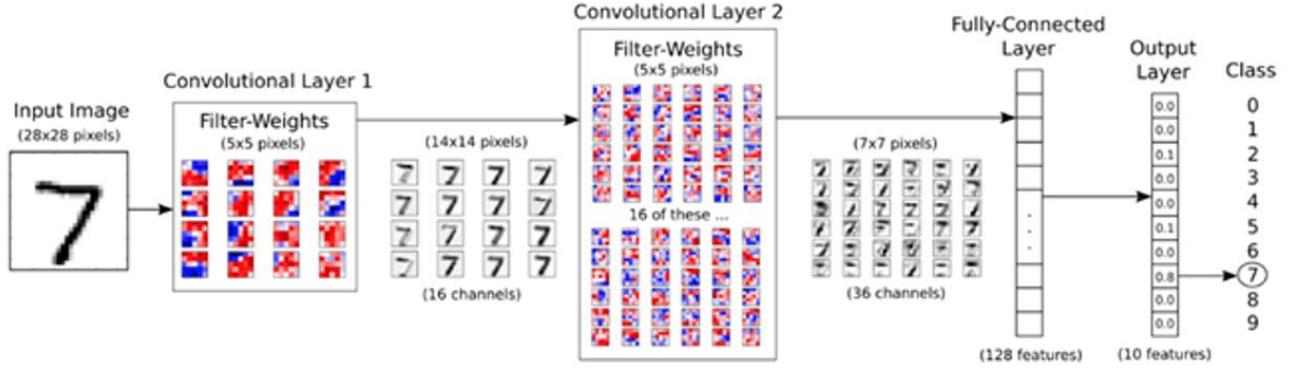


Input image

Thresh. output

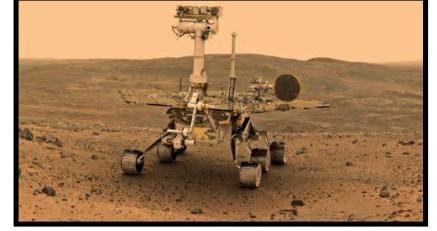
Pipeline output

# An example: digit classification

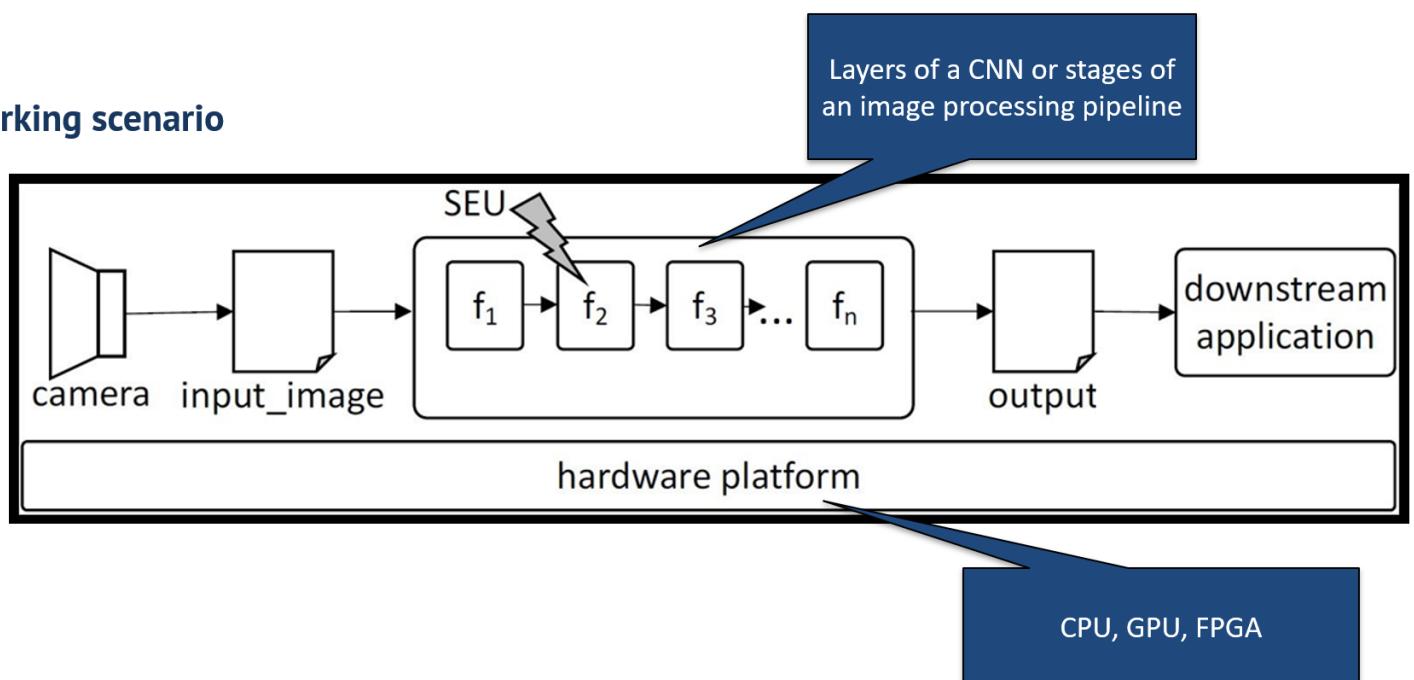


## Working scenario

Increasing interest in employing IP and DL for perception and decision tasks in mission-/safety-critical systems



## Working scenario



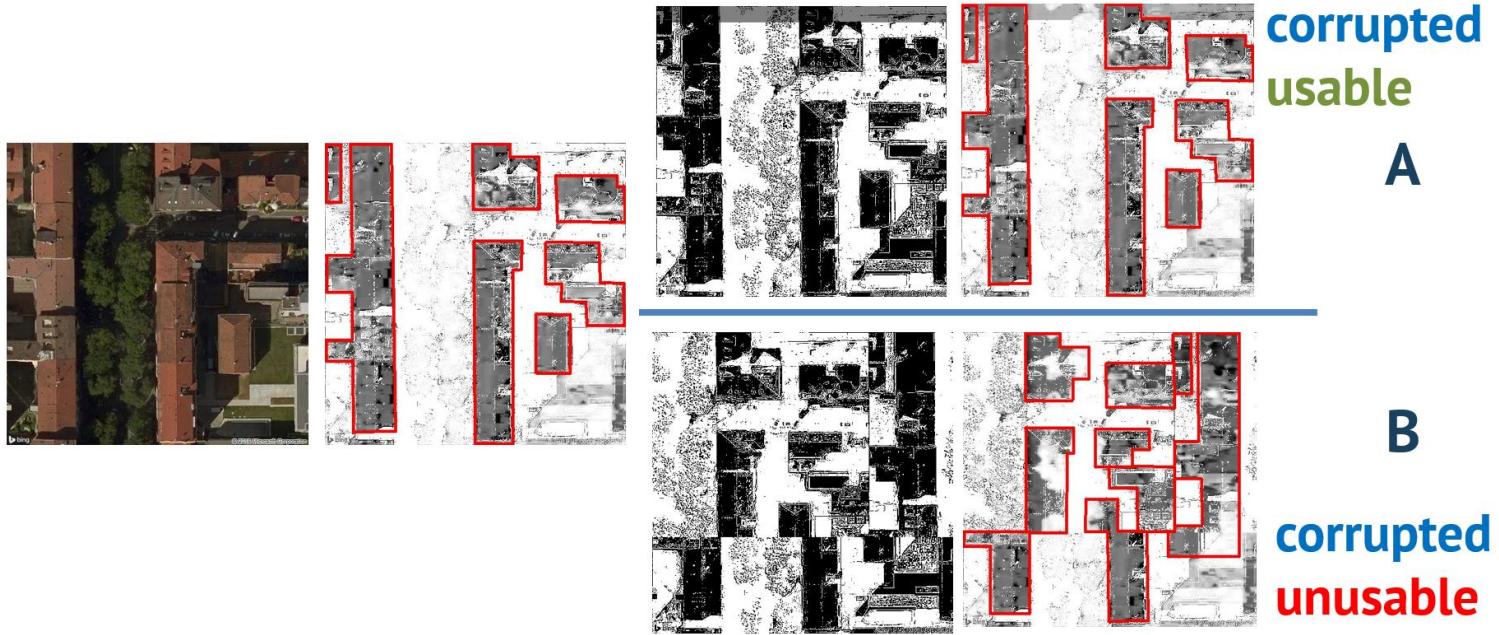
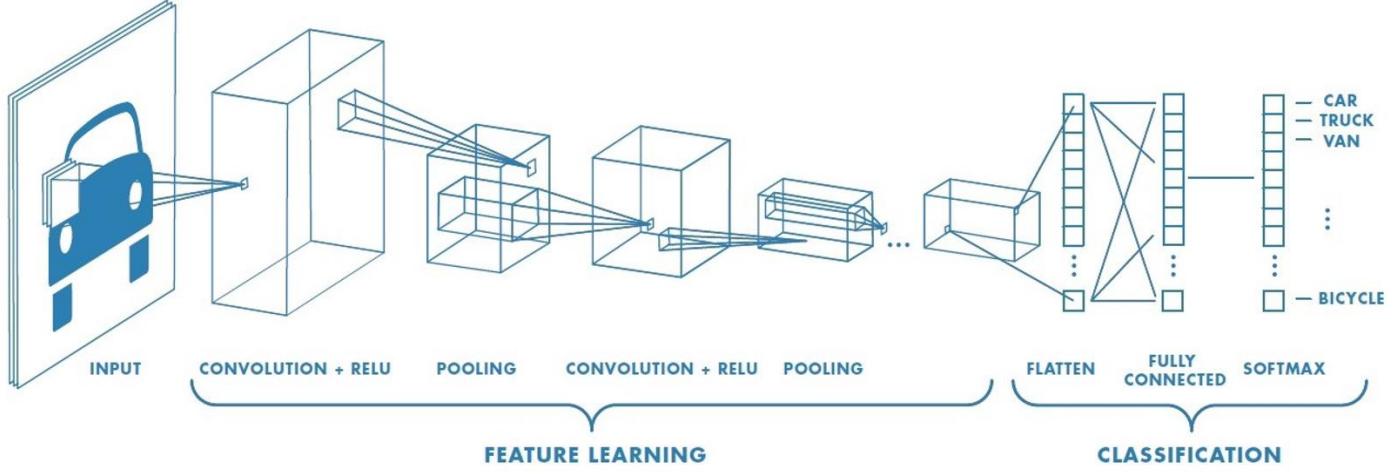
## IMAGE PROCESSING AND DEEP LEARNING

An example: car detection in highway videos



# IMAGE PROCESSING AND DEEP LEARNING

A Deep Learning model generally employed to carry out a high-end task, such as item classification, object detection and image segmentation



## Example application

Car tracking in highway videos



Global input image



Output

## Example application

Car tracking in highway videos (usable corrupted image)



CNN input image (correct)



CNN input image (corrupted)

## Example application

Car tracking in highway videos (usable corrupted image)



Golden CNN output



Faulty CNN output

## Example application

Car tracking in highway videos (usable corrupted image)



Golden CNN output



Faulty CNN output

The idea is classifying results based on the application we have to perform

## Paradigm shift:

*faulty/not faulty* images



usable/not usable images

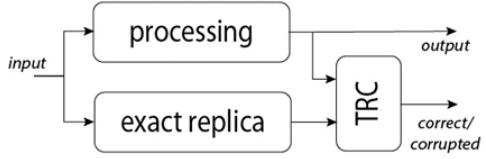
Discard only faulty **not usable** images

Leave faulty but usable images

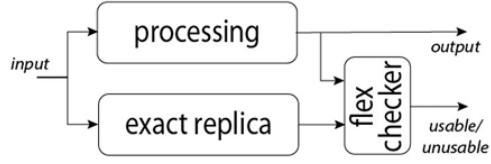


**Avoid useless re-executions**

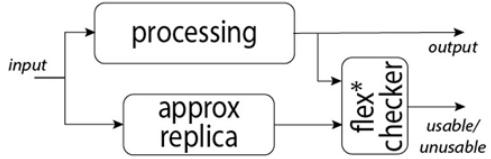
## Several hardening schemes may be adopted



> Traditional redundancy



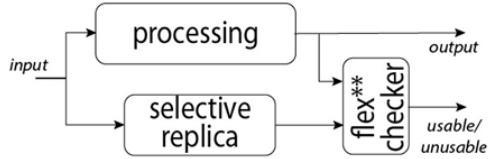
> Usability-based flexible checking



> Usability-based flexible checking

+

Approximate redundancy



> Usability-based flexible checking

+

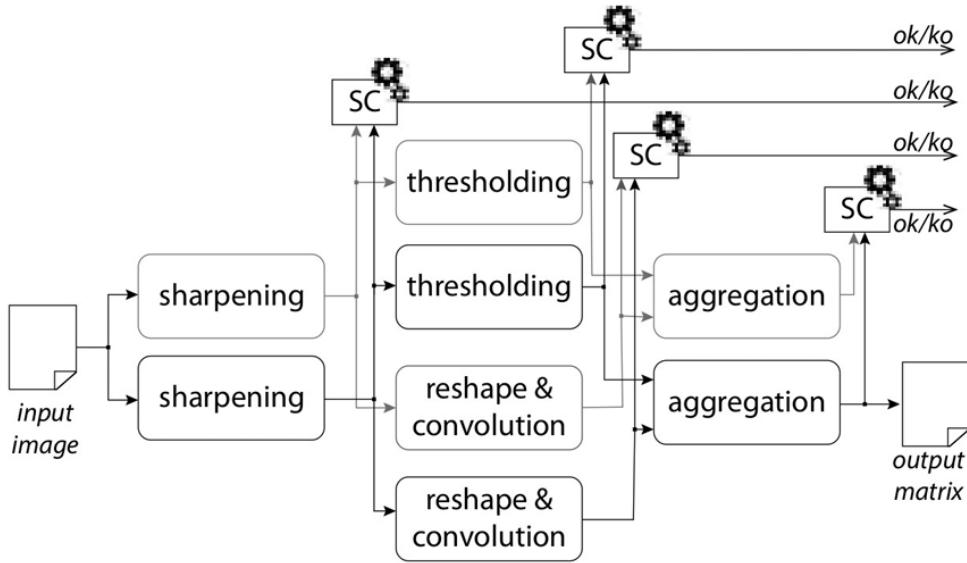
Selective redundancy

TRC: two ray check, it's a classical pixelwise operation

Let's see one of the applications

# USABILITY-AWARE FAULT DETECTION – THE ARCHITECTURE

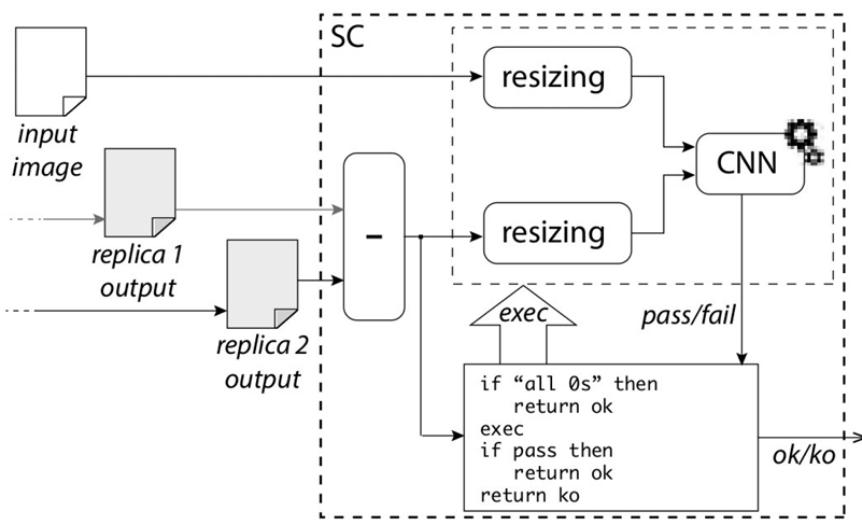
Borrow duplication from DWC...



4 stages pipeline for model identification, the idea is doubling the layers and substituting with a smart checker, when smart checker rises an alarm then we have to solve the fault, only under single fault situation.

# USABILITY-AWARE FAULT DETECTION – THE CHECKER

...but with a *smart* checker



**no fault:** the smart checker behaves as a TRC

**fault:** the CNN is invoked

If the two replicas are exactly the same *no fault occurred*, so the CNN isn't activated, while if the two outputs are different, CNN is activated and executed. It's like a two-rail checker but uses a CNN.

C++ for the considered pipeline

C++ for the proposed Smart Checker (with TensorFlow)

Train and test sets both count 10K 1,080x720 images from MS Bing

Images from the two sets are taken from non overlapping groups of cities

We assumed that filters are scheduled in a time triggered fashion:

Sharpening → Thresholding → Reshape&Conv. → Aggregation

We analyzed the Smart Checker when dealing with randomly corrupted images

Four possible behaviors:

- Discarded Not Usable (D /U) images
  - Not Discarded Usable (/D U) images
  - Discarded Usable (D U) images
  - Not Discarded Not Usable (/D /U) images
- } *Correct response*
- > *Inefficient response*
- > *Incorrect response*

We compared the proposed Smart Checker against:

- The classical Two Rail Checker (TRC)
- Three checkers that discard images when the replicas' outputs differ for more than:
  - 1% pixels (naïve 1%)
  - 5% pixels (naïve 5%)
  - 10% pixels (naïve 10%)

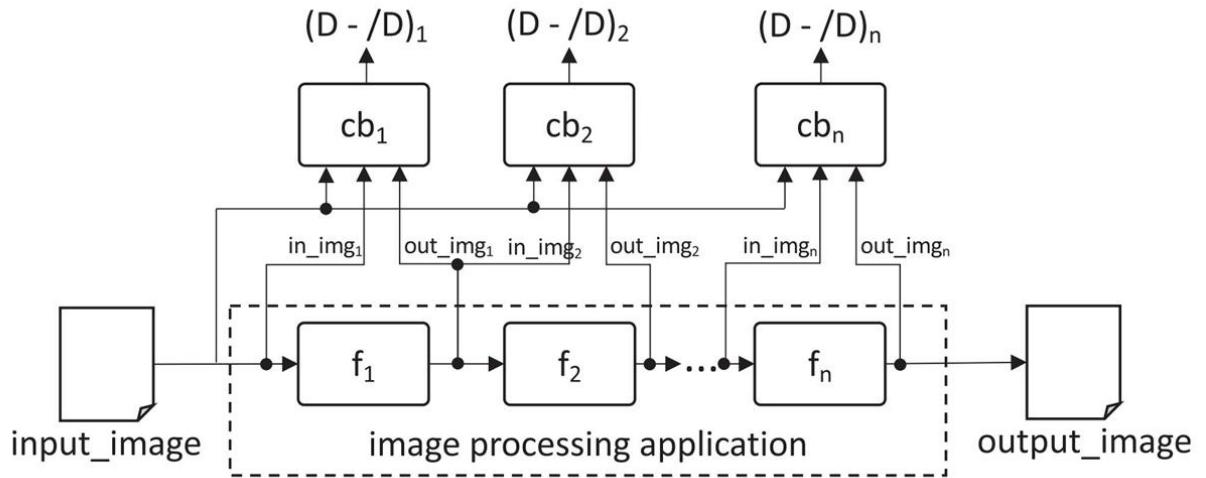
|                  | D /U   | /D U   | D U    | /D /U |
|------------------|--------|--------|--------|-------|
| Proposed Checker | 47.10% | 47.53% | 4.80%  | 0.58% |
| TRC              | 47.68% | -      | 52.32% | -     |
| naïve 1%         | 47.68% | 32.58% | 19.75% | 0.00% |
| naïve 5%         | 47.68% | 37.50% | 14.83% | 0.00% |
| naïve 10%        | 47.68% | 39.95% | 12.38% | 1.20% |

Is it possible to further reduce the fault detection-related cost?

Remove the redundant pipeline copy...

...and substitute it with control blocks.

### The architecture



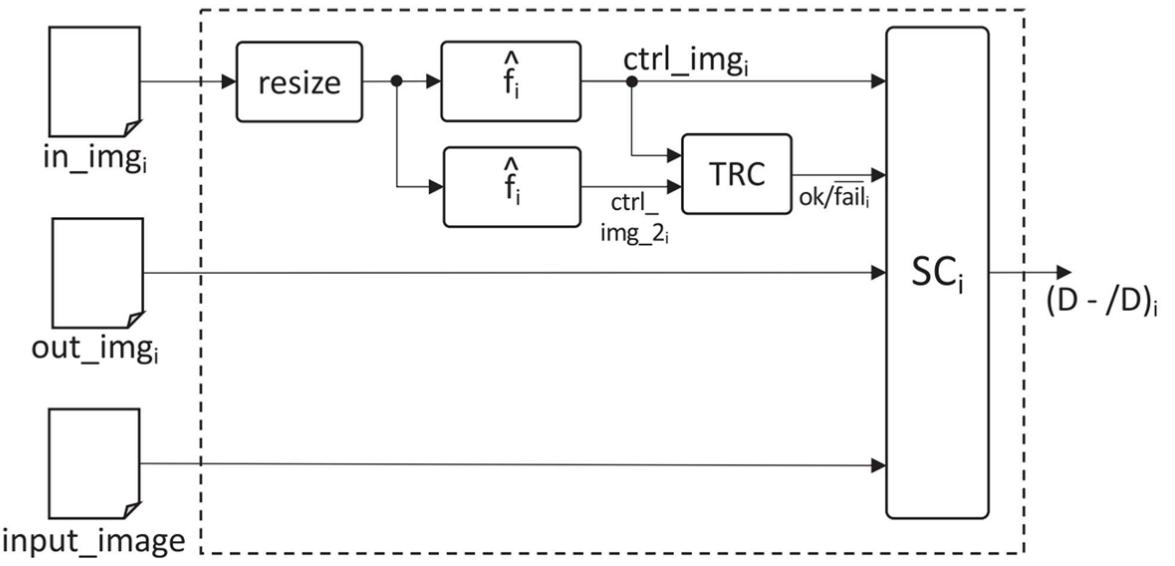
$f_n$  : is a filter

$cb_n$ : control block

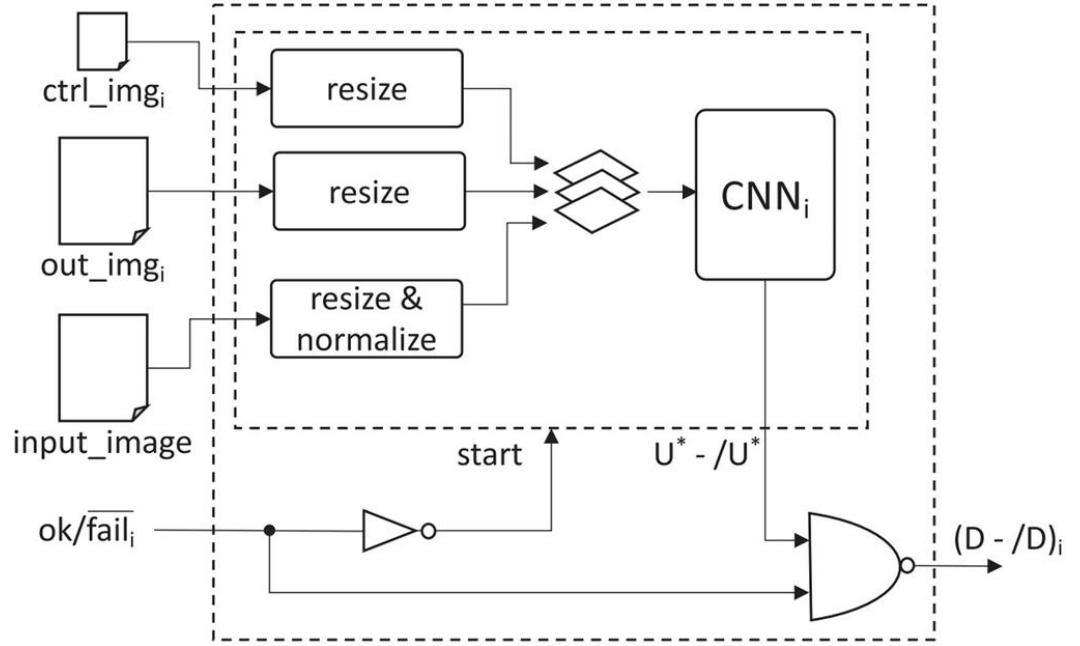
in our case it is a small replication of the filter plus a smart checker

# APPROX. USABILITY-AWARE FAULT DETECTION

**The control block**



**The smart checker**



we are losing detail in the replica because it is used just in the checking

Fault detection accuracy almost unaltered (w.r.t. the non approx. usability-aware fault detection)

## Time saving w.r.t. DWC: 36%

- it was 15% and 5% with the non approx. usability-aware fault detection, depending on the configuration)

| Approach     | Avg. Time | /D /U |
|--------------|-----------|-------|
| Our approach | 414.40 ms | 0.90% |
| DWC          | 652.80 ms | 0.00% |

Is it possible to provide a CNN with sufficient fault detection capability without full duplication?

The idea is to duplicate only a sub-set of the CNN layers...

...but, which ones?

By exploiting error simulation we can identify the most critical CNN layers

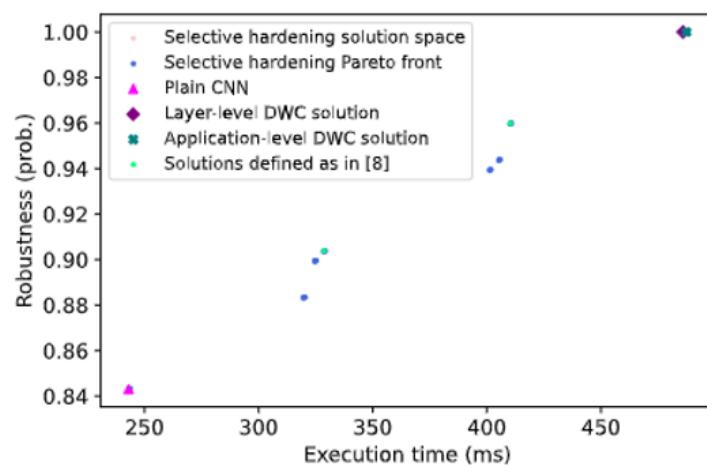
Based on this information and on the layers execution time we can identify the most suitable partial duplication in terms of CNN execution time and fault detection capability

We considered four case study CNNs:

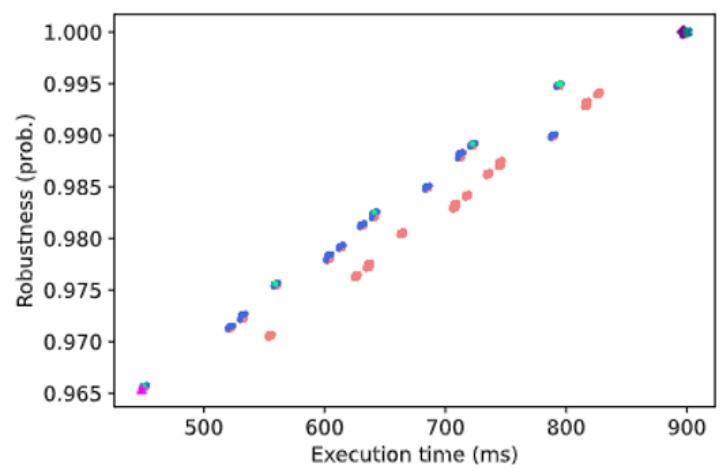
- Comma's AI
  - PiloteNet
  - CIFAR-10
  - VGG11
- } *Steering angle detection for autonomous driving cars*
- } *Image classification*

We compared the execution time and the fault detection w.r.t.

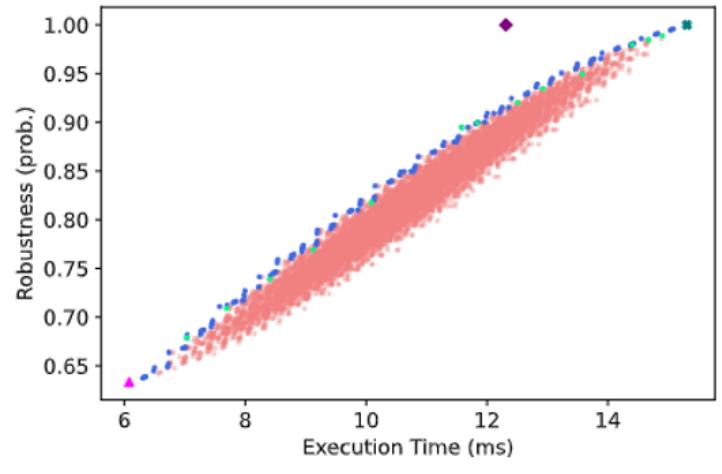
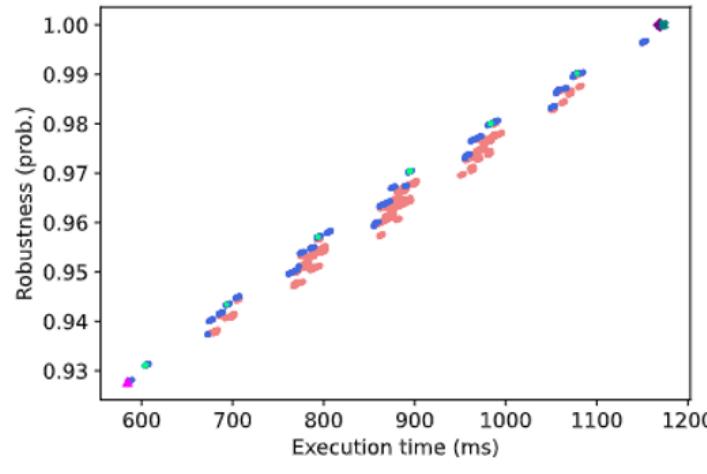
- CNN-level DWC
- Layer-level DWC
- The selective duplication approach proposed in [8]
- The non hardened CNN



(a) Comma



(b) PilotNet



## 10 - Introduction to High-Level-Synthesis

# Need for High-Level Design

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

- Model complex designs
- Reduce the development time
- Simplify the code review (more experts at higher abstraction levels)
- Operate at technology independent level
- Simplify (and explore) HW/SW partitioning

The basic idea behind HLS is to design a component in the simplest way from the *hardware and functional* point of view. The design is very complex because hardware design skills and knowledge is required. This constraint significantly limits the number of people that are able to design an HW accelerator. By raising the abstraction level needed to design a component, it is possible to have more people able to design and implement a hardware accelerator.

If we could have a method to pass from a high-level software description to implement it in hardware it would be a great advantage. By raising the abstraction level we can think about the functionality at software level, so that modelling and executing the system more easily. This reduces the development time and simplifies code review.

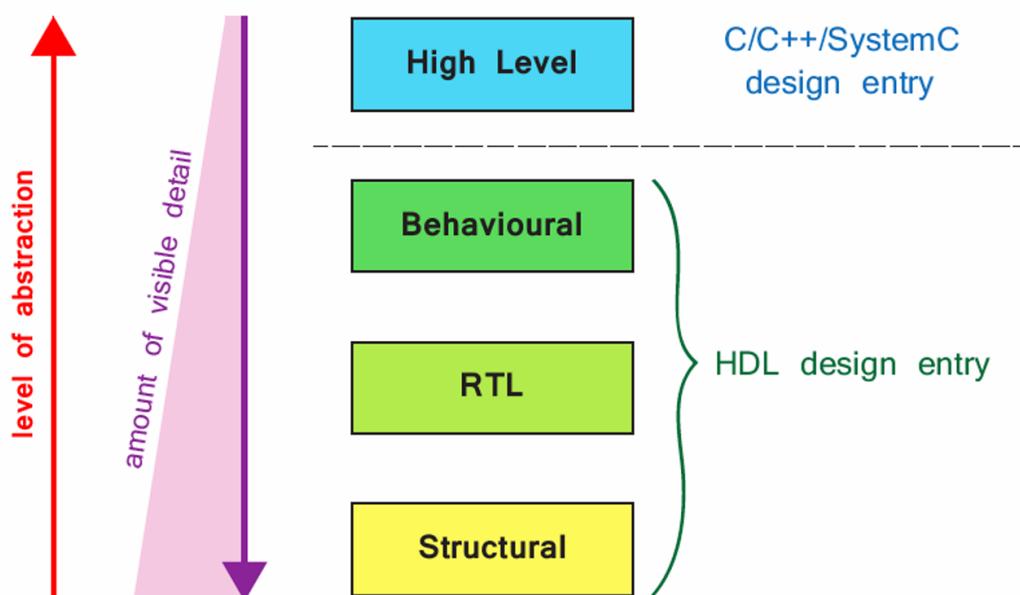
Another advantage is that specifications can be decoupled by the synthesis, because specifications are technology dependent while the function synthesis is independent from the technology, the component can be designed with a specific functionality and can be implemented with different performances based on the technology used to implement it.

*ex: I have to design a cryptographic accelerator and I want to implement it in FPGA because I want to integrate it into my system for video processing that uses an FPGA or I want to create an ASIC for a smartphone. Conceptually the design is the same, what changes is the **implementation**. If we design it in hardware maybe we must design it twice, once for the FPGA once for the ASIC, while if we have something that does it automatically, we can just describe the functionality once and then it's done automatically for the two different technologies.*

Based on what we want to implement in hardware and what in software, if we have something that can allow us to do this exploration at high-level it would allow to evaluate more solutions. If we *don't* have something that does so automatically, we must do the hardware design, then implementation, then see which results are obtained. If the performance isn't as good as we want, the component must be redesigned, it must be optimized and implemented again. If the bottleneck was somewhere else, the entire process must be

repeated. If there was a toolchain that does it automatically then testing can be much quicker, it would still take time but would be much faster than re-doing everything “by hand”.

# Levels of Abstraction in FPGA Design



Source: The Zynq Book

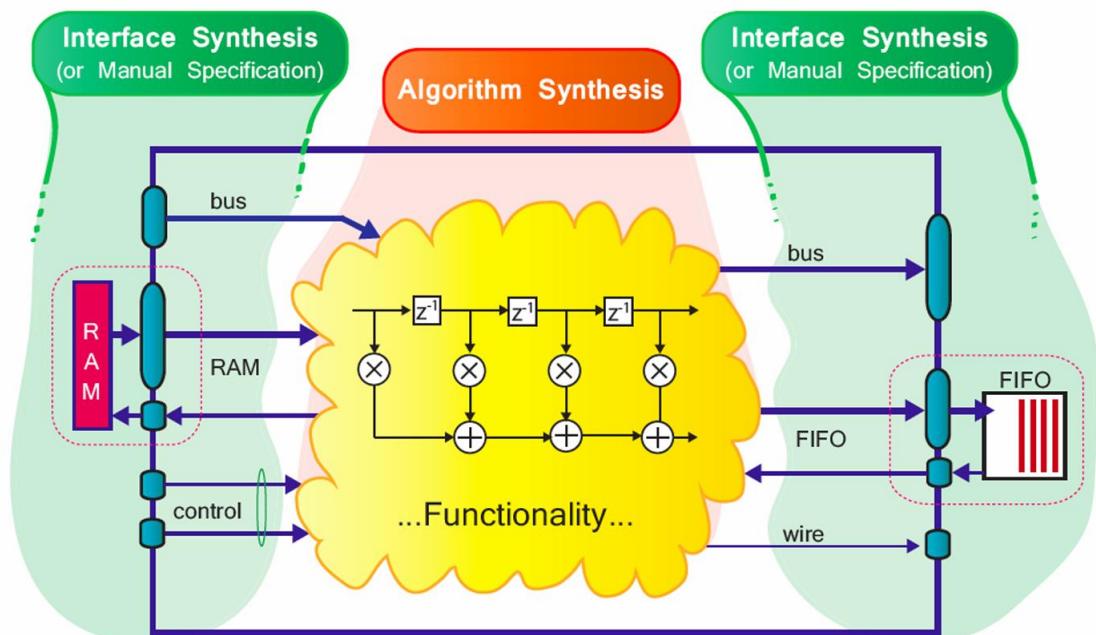
Here we see the abstraction levels for FPGA design.

The horizontal line indicates a change of language domain: in the upper part we have a *software description* while below there are *Hardware Description Languages*.

If we go from the bottom to the top we’re raising the abstraction level.

At the structural level we have a lot of details, we have both the complete description of the circuit, of the nets, of the timing and of the power consumption of the circuit.

## Algorithm and Interface Synthesis



Source: The Zynq Book

In general, what we want to do is something that first we spend a lot of time to verify that the synthesis is a component in system, that must communicate with something else, so we have to correctly implement the interface synthesis in parallel with the algorithm, so we have to separate the interfaces that we may need from the algorithm. The more we know at the beginning of the synthesis, the more we can optimize it. The interfaces might take more power and more time than the algorithm, so it makes no sense to write a very optimized very fast algorithm if then the bottleneck is on the interface.

# High-Level Synthesis: HLS

## High-Level Synthesis

- Creates an RTL implementation from C, C++, System C, OpenCL API C kernel code
- 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

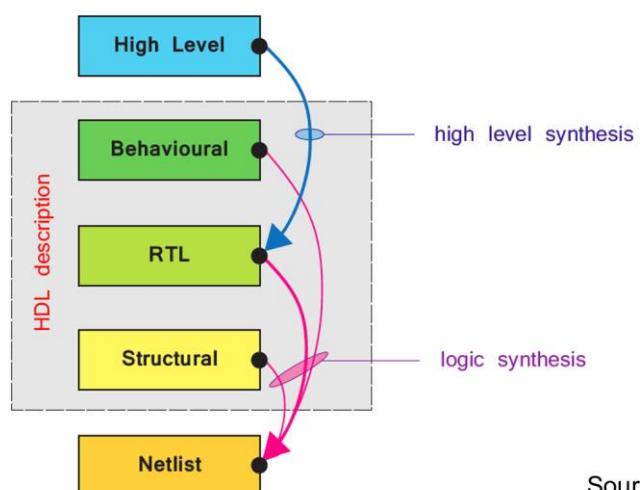
oss: the code that is generated is the *RTL one*, so it has specific timing behaviors and logic nodes.

oss: the RTL that we generate is *technology dependent* because *timing and power are technology related*.

To implement via HLS we have to first understand which function to synthesize, how can we implement it, what is the specific functionality of the accelerator and what is the evolution of the data.

*Remember:* for the same code we might generate different implementations.

## High-Level Synthesis vs. Logic Synthesis



Source: The Zynq Book

as we see, HLS and logic synthesis are at different level, but this doesn't mean they don't exchange information, they can exchange if through the same parameters or through protocols and information etc.

# Advantages of HLS

## Productivity

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

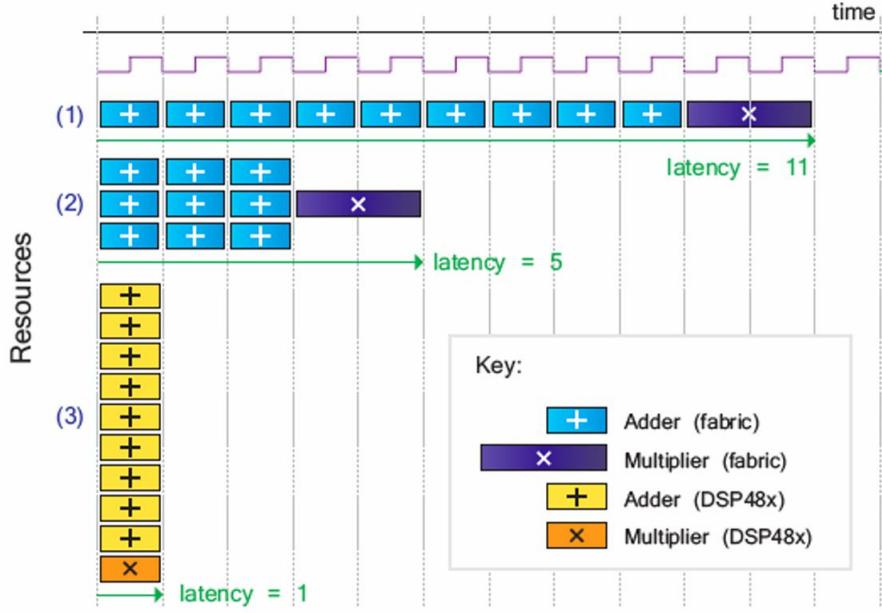
## Quality of Results

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

HLS is something automatic that must implement the intuitions that a hardware designer knows. Since it's automatic, it's obvious that an expert HW designer will implement probably a better RTL design but still it would take a lot of time and very advanced knowledge.

oss: HLS tools use methods of optimizations but since they're standardized probably there would be a more optimized way to implement it, so that's why the role of the hardware designer is still important, because it implements the sort of necessary optimizations.

## Alternatives from HLS – Avg. of 10 numbers



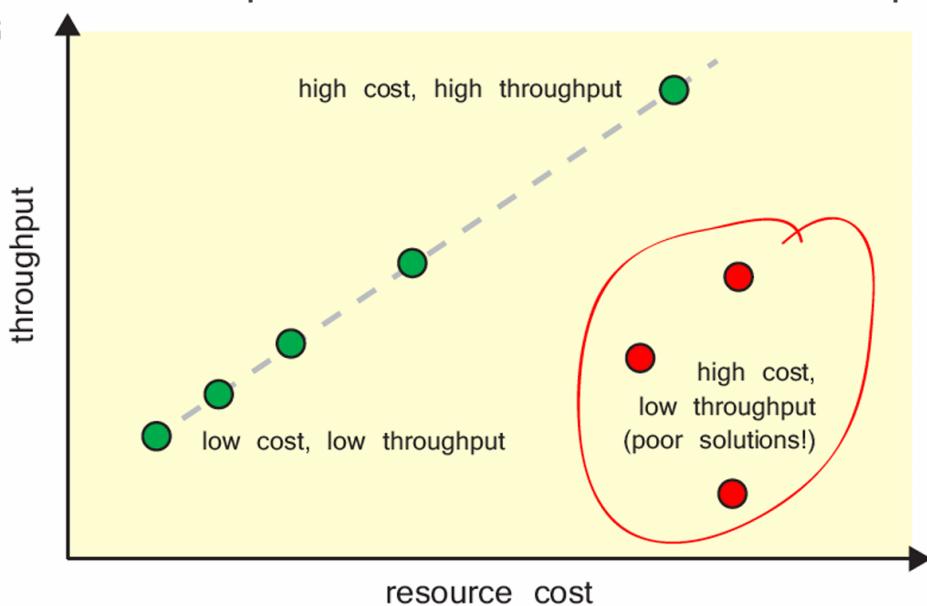
here we can see three approaches:

1. the operations executed sequentially
2. implementation with some parallelization
3. complete parallelization

it's intuitive that a tradeoff between *time* and *resources* exists.

# Design Trade-offs Explored Using HLS

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



by using HLS we obtain multiple design runs, we have to choose the one that fits our solution more and avoid the ones that don't.

*ex if we have the last upper green dot (that is one implementation) why would we need the red ones? we won't use them since we have an equivalent implementation that is better*

## 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

Why did Generation 2 fail?

- Its target was *ASIC design*, but ASIC designers have *very specific and very highly requesting constraints*, they needed extreme efficiency. Implementing a tool that could perform under such constraints was hard.
- It started from a behavioral description, thus VHDL or VERILOG code was needed and so all the designs were complex and slow.

# 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”**



©Christian Pilato, 2024

11

## 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



©Christian Pilato, 2024

12

## Bambu – Academic Tool for HLS

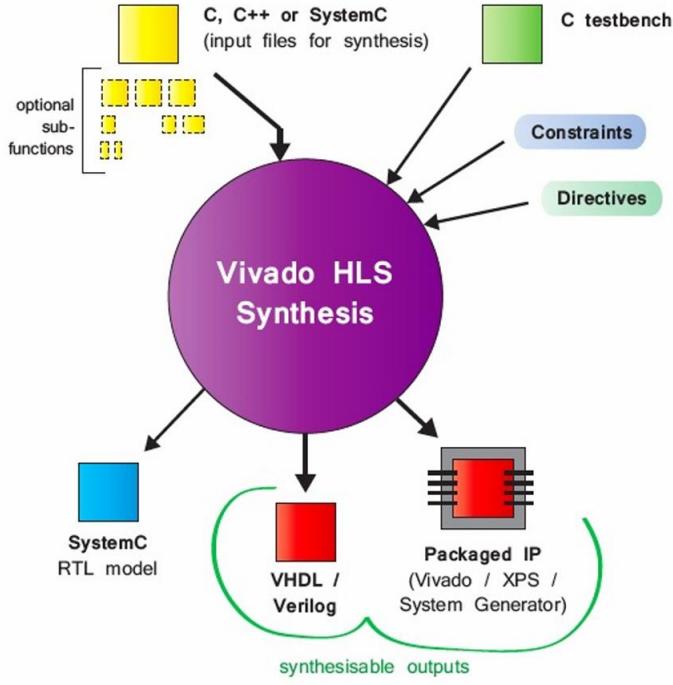
- **Open-source HLS Tool**
  - Under development at Politecnico di Milano
  - Faculty supervisor: Fabrizio Ferrandi
- **High-Level Synthesis from C/C++ to Verilog/VHDL**
- Targets both ASIC and FPGA
  - Automatic generation of testbenches
- **Support for the implementation of custom “passes”**
  - Special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



©Christian Pilato, 2024

13

# Vivado HLS Synthesis Process

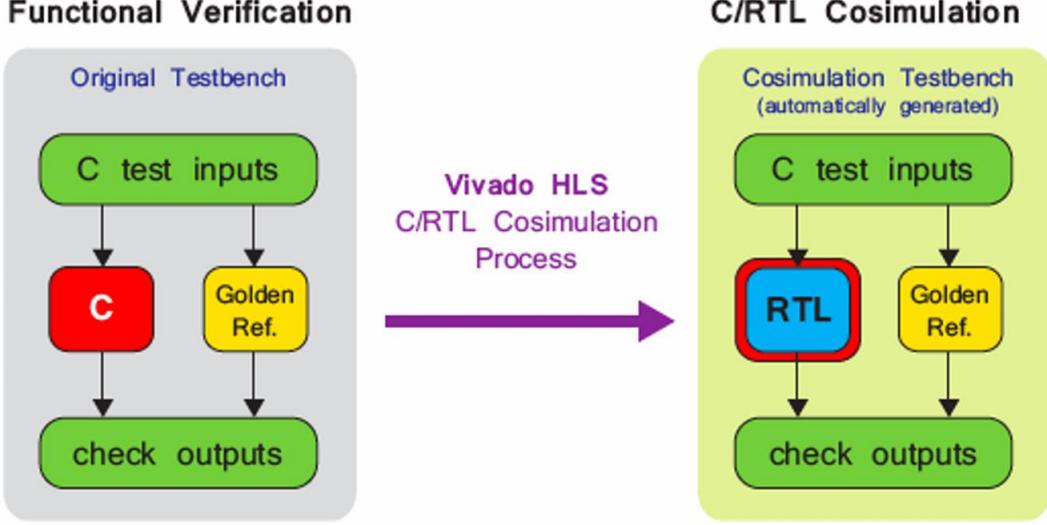


Source: The Zynq Book

we can have in our synthesis tool input descriptions, additional functions (synthesizable functions) so functions that can be implemented in hardware and implement an optimized RTL code.

it is a complete system to replace the steps of the hardware design.

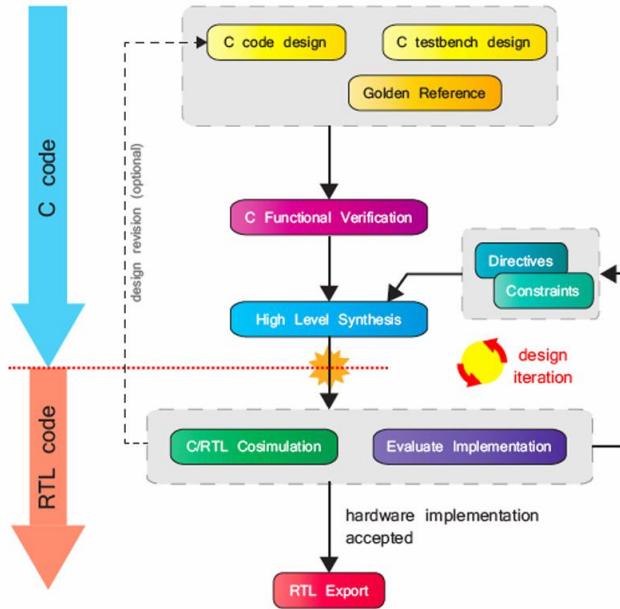
## Functional Verification and Cosimulation



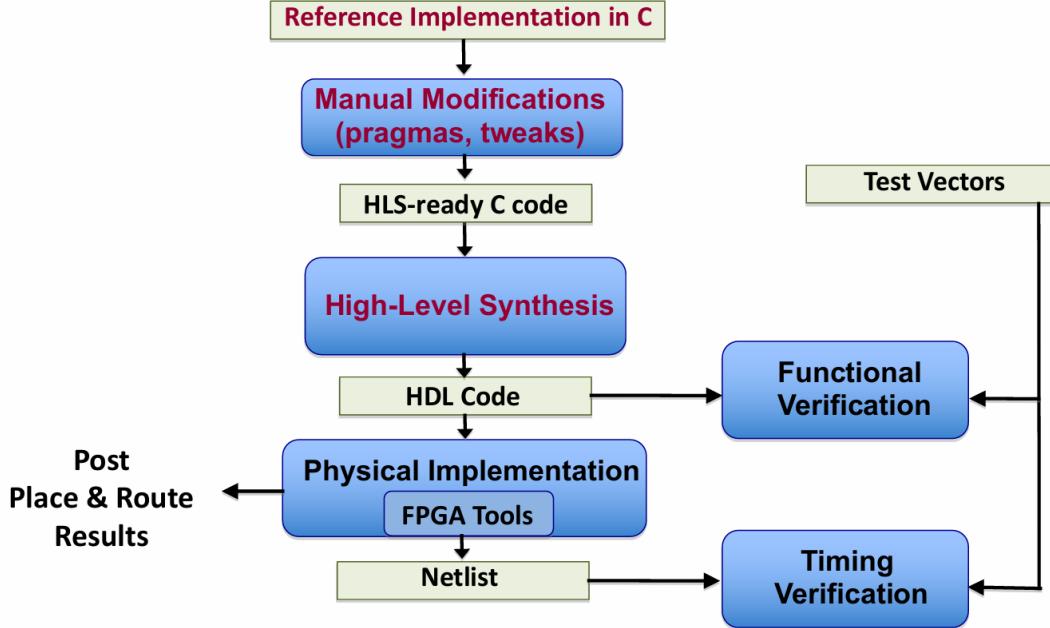
Source: The Zynq Book

So let's say that we want to test the logic that we implemented with the high-level code, Vivado gives us the possibility of just using *C test benches*, we do not have to write a Verilog test bench that would be very long to write. So we have our high-level description, then with a C testbench I verify if the functionality is working well, then we generate the RTL and by using the same C inputs and testbench I verify if I obtain the same correct result that I obtained before in the functional verification. If we have co-simulation errors, then the functional implementation we made is incorrect.

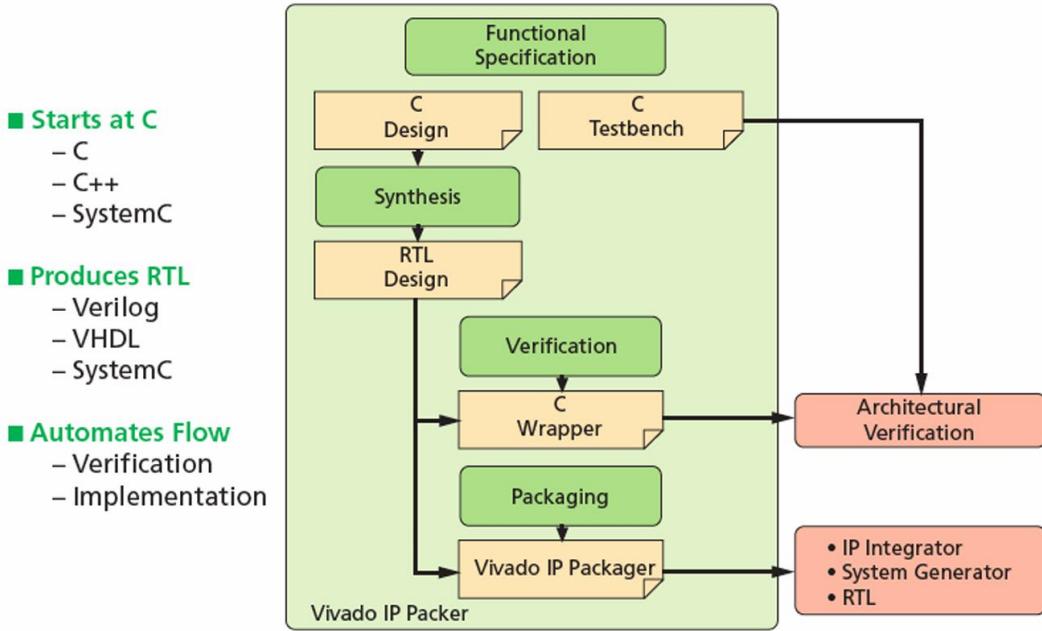
# Vivado HLS Design Flow



## Development and Benchmarking Flow



# Vivado HLS



## 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?

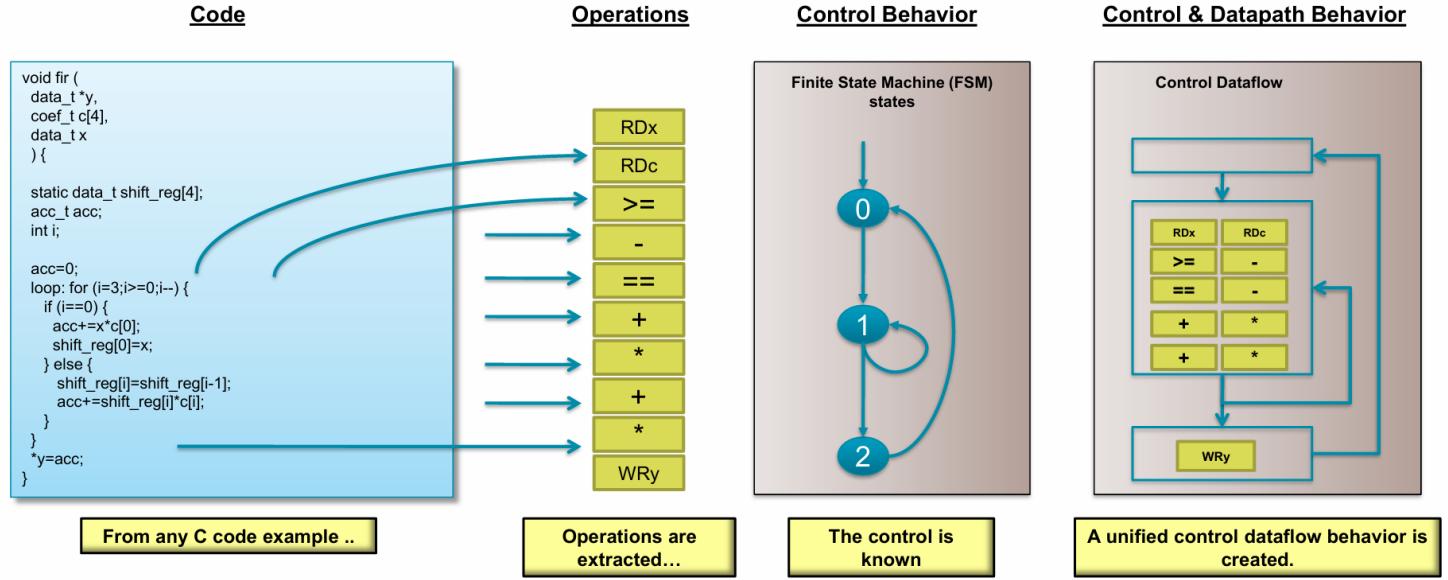
The HLS flow must be imagined as operating per functions. A function in the original C code is a hardware module after HLS, a function containing another function is a *module with a submodule*. The important part is taking a function, the control and the dataflow are taken from that function as it is. If we enter a subfunction, the way in which HLS behaves is the same.

*example: in software the code is executed. When a function is reached a CALL happens and the control goes to the execution of the subfunction. After the end of the subfunction, the control turns back to the main function. In HLS conceptually it is the same, the hardware execution is started, a point where the submodule starts is reached, the control goes to the submodule, the submodule evolves and then it comes back. There is no parallelism in HLS, even though there are cases where execution can be partially parallelized and can be*

When we are working on a function, in any case we must understand how *control* and *dataflow* are translated in the architecture. In HLS, this is done through two steps: **scheduling** and **binding**. *Scheduling* is about how timing is managed while *binding* is about how to allocate the available resources.

How the design is created must be created, so what is the correspondence of a function, of an array, of a loop, to hardware, how these elements are translated in a way that is coherent with what we want.

## HLS: Control & Datapath Extraction



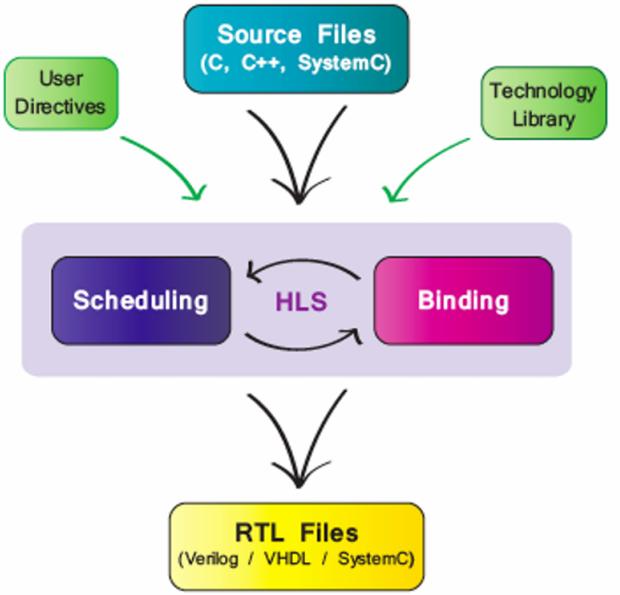
## Scheduling and Binding: Heart of HLS

**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.

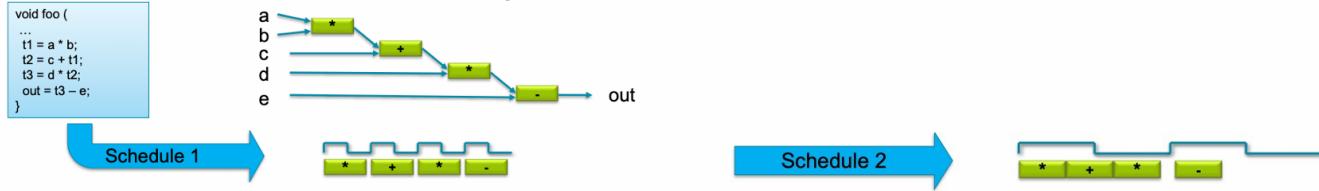


# Scheduling

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

here it is decided how the resources of the FPGA/of the accelerator that we have are given for the specific function.

# 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;
}

```

Remember that high level synthesis is a *static generation of hardware*, so we can synthesize *what we know at compilation time*.

## 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

# 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

First, we need to identify the operations that *can be executed in the same clock cycle* and we can also determine *which parts of the code are never executed at the same time*, this is very important for *if, true false* statements. Then we may want to *optimize loops*, in fact we optimize across the single operation but also for loop operations. In general, scheduling is the first step. When we are satisfied by performance and latency we optimize the resources used, eventually coming back to the possibility of changing the scheduling if the specific requirements aren't met.

## 11 - High-level functionality description

# HLS Problem Definition

## Input

- An **intermediate representation**
- A set of **functional resources** (with area/time characterization)
- A set of **constraints**
- One or more **objectives**

## Output

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

## Tasks

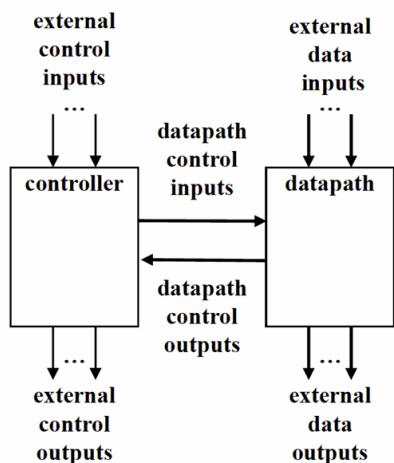
- Place operations in time (scheduling) and space (binding)
- Determine detailed interconnection and control

So the HLS problem starts from an intermediate representation of the functionality that must be synthesized, so we need to generate something in a *representation that is hardware oriented* that removes some details of software and that is as independent as possible from the input language. We do not care about the details of the single input language, but I want to extract the semantics, the dataflow and the control flow. Then, understanding

- what are the units
- which are the delays
- which constraints for area and performance optimization

is necessary.

## Data-path and Controller



### Datapath:

- **Functional resources:** Perform operations on data (arithmetic and logic blocks)
- **Memory resources:** Store data (internal memories and registers)
- **Interconnection resources:** Connect all resources (muxes, busses and ports)

### Controller:

- Finite state machine (FSM)

To define more formally we can consider the organization like this:

- **Datapath:** part that takes the input data, performs the operation on it and considers both memory and interconnections between submodules.
- **Controller:** companion module that determines the control signals like “when we accelerator starts?”, “how does it acts? When?”. This is typically implemented with a finite state machine.

# Constraints and Objectives

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

- Constraint: Maximum area
- Objective: Minimize area

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

**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)
- 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)

An intermediate representation is an internal format that is extracted from the code used to represent a functionality to be synthesized.

Ideally what we want is

*represent the functionality of an algorithm*

*generate the accelerator for that specific algorithm*

***but the way in which that algorithm is written has no impact***

This is really a strong assumption and achieving it is very hard for a couple of reasons:

- when we abstract the details, obtaining a representation independent of the code that is written means finding a *canonical form* that representation.
- we must map all the physical resources, that add some constraints ex: *an addition is an operation between two operands*

## Abstract Syntax Tree (AST)

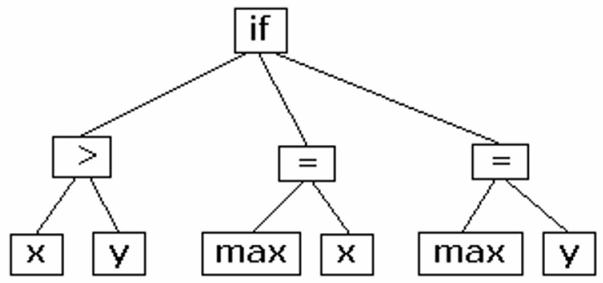
In computer science, an **abstract syntax tree** is a representation of the abstract syntactic structure of text written in a formal language

- 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; → tmp = x - y;
 t = tmp + z;
```

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

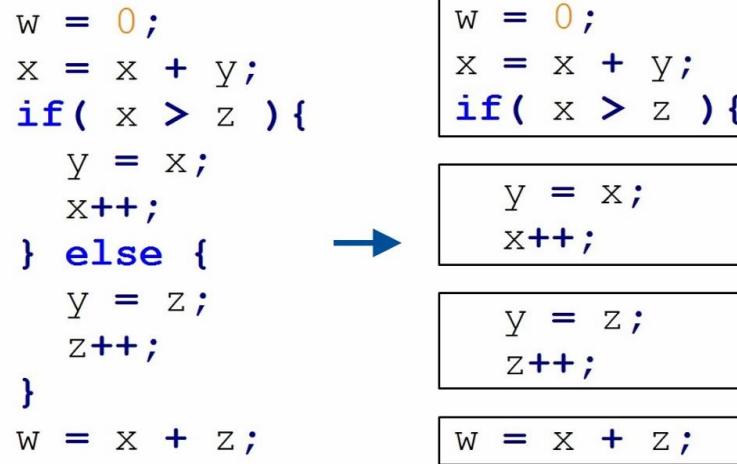
**All operations inside the basic block must be executed before moving to the next one**

A *basic block* is a piece of code that has a *single entry-point* and a *single exit-code*, without any jump in the middle. Every time a basic block starts it must be completed before passing the control to the successive one. Since all the operations of the basic block must be executed before moving to the next one it means that I can *imagine my accelerator as a component executing consecutive basic blocks*, it makes no sense to extract parallelism between different basic blocks since when executing one, the others will be unknown.

If the problem is the other way around parallelism can be implemented and we would like to extract it inside the single basic block.

We might think that expanding as much as possible basic blocks could be an idea for implementing optimization. The drawback is that the problem is much more complicated because a basic block might have thousands of operations inside it.

# Examples of Basic Blocks



Code

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

The first transformation that can be done is thinking a way to find a canonical form for the control constructs. we might do so by

Any while/for/switch can be converted into a sequence of operations combined with `if/goto`

# Control Flow Graph (CFG)

Graph representation of the control flow.

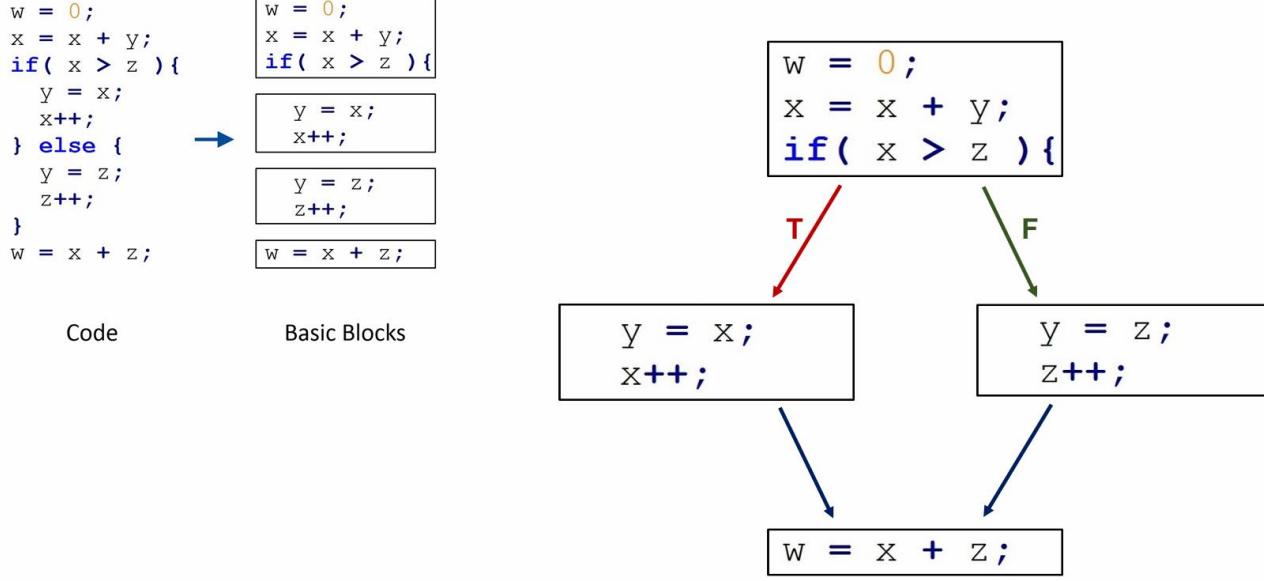
- Nodes are the basic blocks
- Directed Edges are the potential control flow paths

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



# 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
```

SSA is a representation that transforms a single piece of code into one where every variable is created only once, every time I have a new assignment I have a new variable. The advantage of this representation is immediately identify which is the defining statement of each variable. Each variable will have a definition set and the uses. If I have a variable that is created but never used, I can eliminate it.

This is another implementation of the canonical form.

## 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 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

The dataflow graph describes how the data flows from the input to the output.

Dependence: if the target is using a value that comes from the result of the previous operation, we have a dependence.

# 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

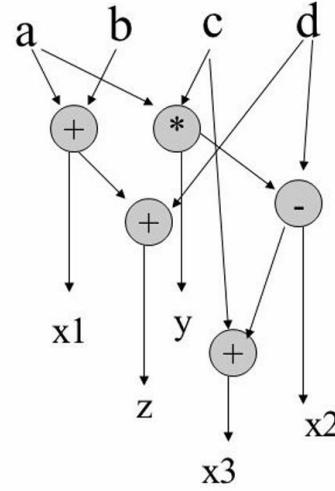
Notes:

- 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:

$$\begin{aligned}x_1 &\leftarrow a + b; \\y &\leftarrow a * c; \\z &\leftarrow x_1 + d; \\x_2 &\leftarrow y - d; \\x_3 &\leftarrow x_2 + c;\end{aligned}$$


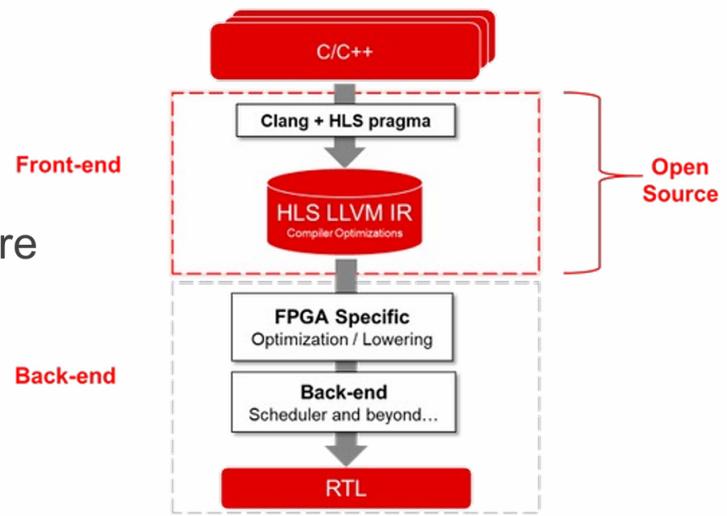
# Compiler Frontend

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
- 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

every time the code is written for HLS we have to add a label to the loop so that it is easier for HLS debugging. If we don't specify labels, the tool will internally assign labels and at a certain point.

Loop unrolling: we explicit each iteration

# Example

```

void block fir(int input[256], int output[256], int taps[NUM TAPS],
 int delay_line[NUM TAPS]) {
 int i, j;
 for (j = 0; j < 256; j++) {
 int result = 0;
 for (i = NUM TAPS - 1; i > 0; i--) {
 #pragma HLS unroll
 delay_line[i] = delay line[i - 1];
 }
 delay line[0] = input[j];
 for (i = 0; i < NUM TAPS; i++) {
 #pragma HLS pipeline
 result += delay line[i] * taps[i];
 }
 output[j] = result;
 }
}

```

## 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

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

Since the memory allocation and variable allocation comes from the compiler, the HLS automatically assumes the same bit width for each datatype as defined in the CPU version. In hardware it is better constraining the dimension of each type, to reduce weight in the hardware implementation. It is convenient consider for a first implementation the same precision in hardware as in software, then customize the specific datatype and signal to a lighter version with reduced width. In HLS there are specific libraries to define variables where we can specify the bit width.

# 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
```

The variable can be later used as *it is*

```
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**

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**

*recall*

- parameters **passed by copy**: passing a parameter by copy in C means that when an argument is passed to a function, a copy of the argument value is made and used inside the function, so a new variable is created in the function and is initialized with the value of the argument and the original variable in the caller is not changed. → converted into input ports connected to the registers of the CPU
- parameters **passed by reference**: passing a parameter by reference in C means that the address (so a pointer) of a variable is passed to a function, so that the function can modify the value of the original variable. → converted into memory interfaces that access an external memory from the component

Preserving the semantic of the functions means that when we start a function we might pass some values to it and we expect to get a certain result.

(chiedere esempio della funzione che restituisce due puntatori)

## Accelerator Interfaces (ii)

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

- AXI-Lite for parameters (memory-mapped IO operations)
- 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

During the scheduling phases we need to make assumptions on the duration of the operations, for arithmetical operations that's easy because the duration is specified from the library that implements, while for memory operations that's much harder because if the data is local (inside a PLM) the assumption is that the memory can be accessed in a fixed number of clock cycles, while if the data is *remote* we don't know *if and when* the data is *ready or not* (in the sense of number of clock cycles obviously).

## 12 - Scheduling and Binding

# 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)

Binding: association of the single operation to the real functional unit.

Temporary values that have to cross multiple clock boundaries have to be stored and assigned to registers etc.

There always is a naïve solution: if we do not care about parallelism, I could assign each operation to a different clock cycle, obtain a feasible but not optimized solution.

Also for binding we can assign each operation to each functional unit and each temporary value to each register, so we do not implement resource sharing, and we avoid conflicts. Obviously, this is not the target of binding, since the **goal of binding** is trying, wherever it is possible, to reduce the hardware resources by

exploiting sharing resources. Usually it is implemented after scheduling, we decide which operation is executed at each clock cycle and then we can decide and determine if they can share resources. If operations are assigned to the same resources but they're executed in mutual exclusion for sure there won't be conflicts.

It can also be executed before scheduling: we can *preassign the operations to the units* that we want and then *determine the order of the operations that respect that assignment*, this means that *if two operations are assigned to two different resources*, between those two operations there's *no conflict* and so we can execute *them in the same clock cycle*, but otherwise if for any reason they are assigned to the same resource it is still possible to execute them but simply *they can't be executed in the same clock cycle*, they have to be serialized and the problem is choosing *which one to execute before and after*.

It's a general problem in operational research, we have a job, and we have to determine the resources and the order of the jobs to respect the constraints that we have.

## 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

Resource sharing is the possibility to exploit or use resources when they're not used by other operations. They can be extracted before or after scheduling based on the scheduling.

If I preassign the binding, the operations that share the same resources cannot be executed in the same clock cycle and must be serialized, so we have a constraint on the scheduling

If I execute the binding after the scheduling, we execute it on the schedule graph, that is a graph that contains information about the scheduling. At that point, if two operations are not executed in the same clock cycle they can share resources, not that is mandatory but that can be implemented.

# 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**

We cannot find an optimal solution in polynomial time, the input is an intermediate representation that comes after the compiler transformation, it describes operations to be executed in the functionalities and the dependencies that must be respected. The other information that is important is time dependencies and target technology.

First, we need to generate the hardware so that we know what is going to execute the operation, we need to know the hardware latency for each operation in that technology. Remember that we are talking about *allocation in time*, so during scheduling I am still not interested in the cost of the operation.

This is independent of the clock period, because clock period and frequency is instead a property of the design. This is connected to the technology, because the maximum constraint is set by it, but we can also have with the same technology different target frequencies.

Once we define the target frequency, we can determine the *time budget* for each clock and combined with the latency of the functional unit we can determine the real time latency for the scheduling problem of each operation assigned to it.

*ex: my unit will take 7 ns to execute, and we know that if the clock period is 10 ns we need just one clock cycle to execute that operation, but if the clock period is 5ns we need at least two clock cycles to execute it, at the end of the clock cycle I still do not have the stable result.*

Another important consideration is that the clock cycle is  $x$  time, but the delay of the unit is not the only delay that we must consider. The clock period is the time from the output of the register to the input of the target register, but in the middle (apart from hold and setup time that can be considered small) we do have interconnections, multiplexers and if we go to ASIC where also the wires have delays, we might also have line propagation delay. So we might have a certain clock period but usually we must keep a margin for our unit, otherwise we might have timing violations.

*ex: if the unit is 6.9 ns is very risky to have a clk period of 7 ns, it's better to add margin after each unit or we take into account during the binding so we avoid sharing (because mux introduce sharing) so we have to take into account that for the scheduling of the device.*

The output of the scheduling is **determining the starting time of each operation**, then we have the start time of the operation, the latency of the unit, the clock cycle duration and we can determine at which clock cycle the operation terminates, note that it's just a *consequence of the start time*, so the only information we really need is the start time, and we can do so by choosing the appropriate dependencies and start time.

The start time of the previous operations are important because it sets the timing dependencies, based on combination of clock period and latencies.

*If an operation starts before the end time of another operation it violates the timing constraint and it's a bug of the design.*

So the scheduling is time oriented but can also control area and latency exactly with this idea of playing with the constraint of the resources.

## 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* has a directed dependence on the performance of the circuit, because it *determines the timing evolution of the circuit*, so I describe cycle by cycle *what operations are executed* and so I have a direct impact on the latency, because I determine when the last operation completes, and also on the throughput, because if a part is pipelined we can determine after how many cycles can I start that operation (initialization interval).

The indirect effect is on the area, because if I put operations in the same clock cycle I'm forcing them to be executed in different units, so I must introduce a mux and add additional resources.

If I determine a certain number of operations to be executed in the same clock cycle for all the cycles, at a certain point, for each operation type I will understand which is the maximum number of parallel operations in the same clock cycle, I need at least  $n$  adders to execute  $\gamma$  operations in parallel. The maximum number of operations in the cycle determines the minimum number of units needed for that cycle.

# 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



©Christian Pilato, 2024

7

There are three possible approaches

**lower bound of clock cycles:** I parallelize everything so I can't get nothing better than this, obviously this would take an “infinite” amount of resources, the only constraint that forces us to serialize is the data dependencies

**resource constraints:** in each clock cycle do not use more than  $x$  adders, so I will assign  $n$  to that cycle and  $\gamma$  to the others, i'm implementing serialization. I have under control the number of functional units, this is specially important for expensive hardware resources, like floating point units or multipliers.

**timing constraints:** I want real-time scheduling so in a certain deadline I want to have the best use of hardware resources out of this

# 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*



©Christian Pilato, 2024

8

*Linear programming and Integer linear programming:* I can determine the variables, that are the assignment of operations in the clock cycle and the constraints are the one related to the dependencies. The start time must be

$$\text{start time of the previous operation} + \text{latency of the performed operation}$$

The problem with *NPR problems* and *exact formulations* is that they are good but they do not scale well, because the number of variables and constraints is growing exponentially with the number of operations and with the number of units, it is feasible for a limited number of operations, then I have to approximate with an heuristic approach, so practically using some sort of approximation.

*meta-heuristics:* this category includes all the methods that use design-space-exploration to determine variants of the implementation.

The major difference between the first categories with respect to the last:

- *heuristic:* based on the algorithm it generates **one and only one solution**
- *meta-heuristic:* they explore **many solutions** and select the **better one**

# 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

simplest solution because the principle is not considering constraints and as soon as the operations can start because the predecessors have been completed, the operation can start, we never postpone.

Since I assume that all the operations have bounded latency, once I determine the start time of all the predecessors, I can immediately compute the starting time of the depending operation.

The asap scheduling has a linear complexity, I need as many iterations as much are the number of operations, in each step I can assign an operation in time. We are not considering constraints on area and latency, but this gives us the lower bound of the timing constraint, we know that the circuit can't go faster than the ASAP result.

# 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

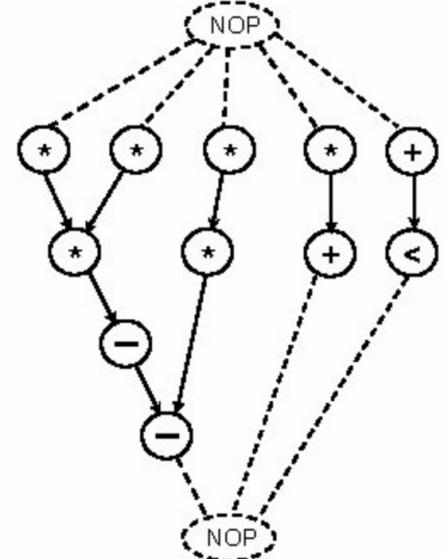
$$\text{start\_time}(\text{op}_i) = \max_{p \in \text{Pred}} \text{end\_time}(\text{op}_p)$$

3. Define the end time of the current node

$$\text{end\_time}(\text{op}_i) = \text{start\_time}(\text{op}_i) + \text{delay}(\text{op}_i)$$

4. Add all successors of the current node to the set of ready vertices

5. Repeat from step 2 until the set is empty



All the operations without input dependencies can start immediately and then I build the graph based on the dependencies. Once I have a node, maybe the source node, I can order the nodes in a way for which given a node, all the predecessors are before it in the list. We can create levels, so I execute the nop, then I can execute all of the multiplications/adders...

How can I compute the start time of an operation? The start time is the maximum end time of the predecessor.

How can I compute the end time of an operation? It's the start time plus the delay.

We have all the information now ↓

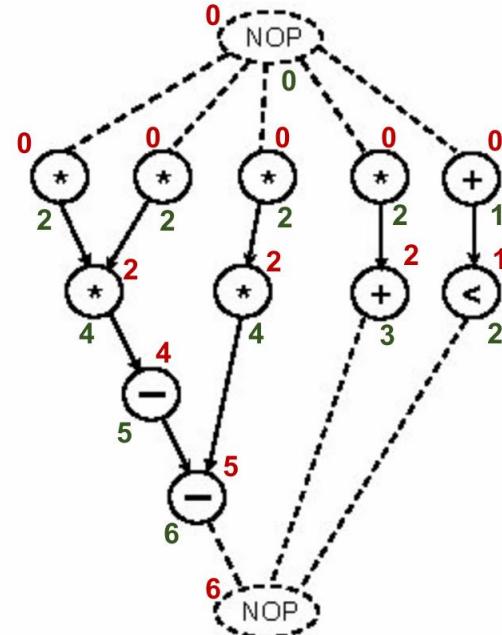
## ASAP: Example

Assumptions:

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

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

$$\text{start\_time}(\text{op}_i) = \max_{p \in \text{Pred}} \text{end\_time}(\text{op}_p)$$



The ASAP determines the maximum latency.

## 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

I assume a certain end time and then I assign the operation as late as possible, *basically you have a deadline and you can't postpone more than a certain time*. I have constraints on delays.

# 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

I start from the back and I go upwards

# ALAP: Example

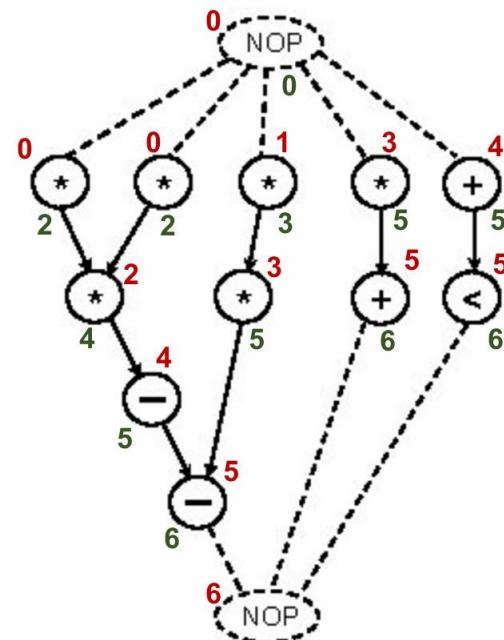
**Assumptions:**

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

**• Start time**

**• End time**

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



*6 in the image means: lower bound given by the ASAP and then I go backward. Obviously, operations are scheduled at zero.*

In the ASAP approach I'm pushing to the top the operation, in the ALAP I'm pushing it to the end.

now I can compute the *mobility*

# Definition of Mobility

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

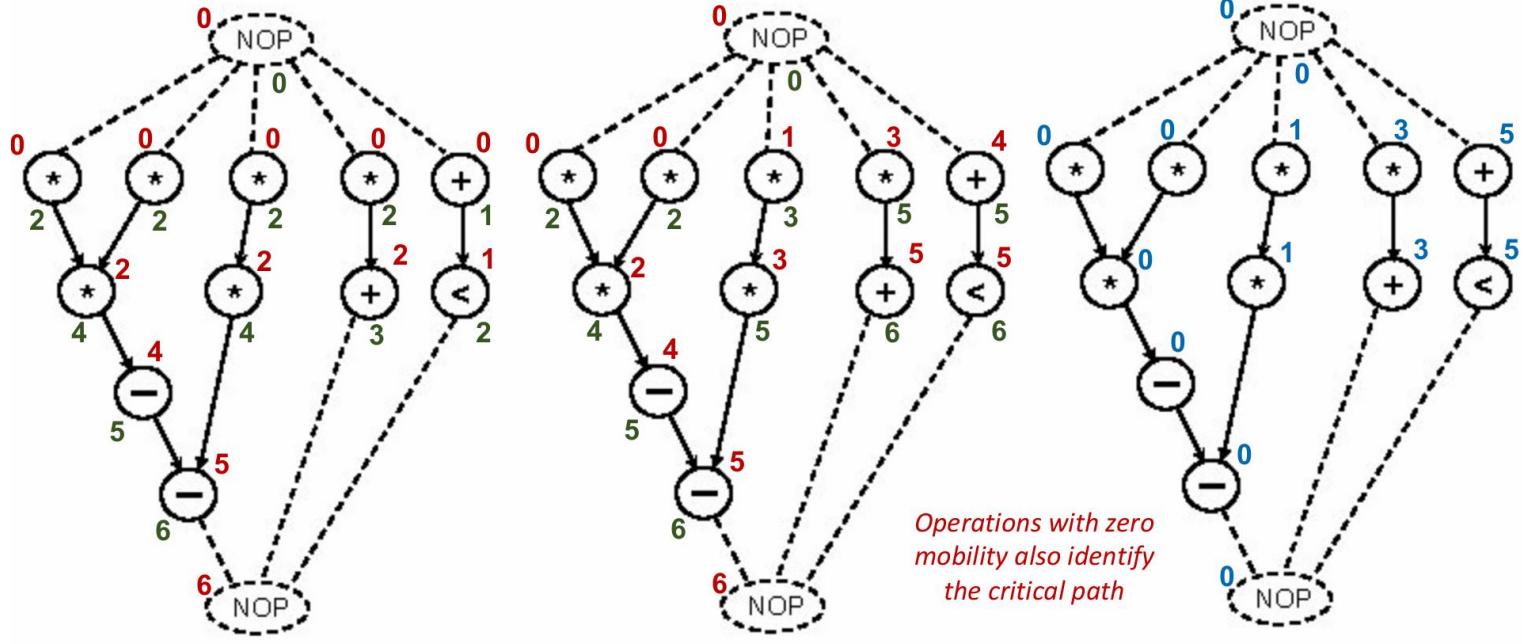
**Zero mobility** implies that an operation can start only at a given time step without introducing any delay on the overall schedule

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

- Time interval in which an operation may start

**mobility:** difference between the start time in the ALAP and the start time in the ASAP. Once I have these two values I can obtain the mobility, that is a number equal to 0 when *ALAP = ASAP* otherwise if I obtain a value greater than zero then I can postpone a specific activity and still hit the deadline.

## Mobility: Example



combined results: where I have 3 I can delay that specific operation of three cycles and still respect the deadline.

These information are useful because I have to concentrate more on the operations with 0 mobility and then I can work/delay on the others.

ex: by the results we obtained, we know that if we want the ASAP result we need to have at least four multipliers, while if we want to reduce the number of multipliers we can concentrate the resources on the operations with 0 mobility and delay and serialize the one with mobility greater than zero and maybe use less multipliers without changing the number of clock cycles

# 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

We want to try to find a tradeoff between reducing the latency and using a constrained amount of resources. Maybe we want to minimize the resources without considering the latency so we find the minimum with resources, or we may minimize the latency with bounded latency, we ask if by scheduling differently we can obtain a feasible solution.

## 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

When we have the possibility to choose an operation, we select the operation to schedule based on the mobility and on the criticality, criticality that is the inverse of the mobility, because if an operation has lower mobility that means that it is more critical. It doesn't guarantee the optimal solution but is linear and in general works well.

How can we take into account the constraints? We create a priority list for each of the operations, we create a list for each of the resources then we order the list by their priority, we can have operations with higher priority first, so with the lower mobility first, then we assign the operation to the current clock cycle if the unit is ready, and if it is we take it from the list and we assign it to the resource and remove it from the list, so we schedule the operation and we know that we can compute start/end time of that operation, and I repeat this process until I complete the list.

The general problem of this method is that we compute the mobility, we assign operations, but if I take the mobility as a static value I can have the problem of stallation, so the intuition is that if we go to the next clock cycle we need to update all the mobilities, because if an operation hasn't been scheduled it has one less clock cycle to be postponed. For "our" activities is clear, if I have 7 days to complete a 6 day activity I have 1 day of slack, but if after one day I don't update my mobility I still believe that I have one day of slack but that's not true, after one day of stall I have 0 mobility.

If an operation has negative mobility I already know during the schedule that it introduces a delay to my circuit, because it has already passed.

## 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)

ASAP and ALAP are not changed after cycles, what I change and update is the mobility.

## 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**)

What are the possible challenges for scheduling?

First we have to remember that delays are given by the functionality, this means that an operation can complete in one or more clock cycles depending on the clock period, they cannot take less than one clock cycle but if I have two operations that are depending on each other (like expensive operations plus a not expensive one ex *multiplication and shift*) technically, with the definition that we've seen before, the shift will start in the cycle after the multiplication has completed, but this is a special case where if the combined delay is less than the two clock cycles, what happens is that the multiplier+shift will be able to complete before the end of the clock cycle, but to do so I have to connect one unit to the other.

This optimization of not connecting unit-register-unit-register but unit-register-unit-unit-register is called chaining

Pipelining gives the possibility of having another approach because I can start the new operation even if the previous hasn't completed.

Another case is when I need external resources, so like informations that are in an external memory, so I need some synchronization signals.

Unbounded operations: operations for which we can't know the exact number of clock cycles/delays needed because maybe they depend from an external source, so here we implement synchronization protocols and we keep the FSM in the state of waiting for the needed resources.

## 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

Resource binding is the assigning in space of the operations and resources, we want to decide what physical resources will be used in the design, how many and then the specific assignment

The idea of resource binding is that it will be better than the naïve solution of assigning each operation to one unit, at that point we don't need any binding, we don't need any special attention for the binding, these are independent units and just have to be connected, but most of the time this is a waste of resources. So the basic idea is to assign one unit to more than one operation, we have a decision to make.

Everything is more complicated than what it looks like because we have to assign resources to each operation, but we have to assign also temporary values to registers, once we do this we might search for a more convenient binding. If we do a better assignment and a successive cycle another assignment is performed to another register I don't need the multiplexer because that resource is always connected to the same register, while otherwise I need a multiplexer. In general this is a resource constraint problem, especially in the case of circuit dominated by resources where I have many operations, correct binding becomes critical.

So we decide a certain type of resources based on the binding, then we decide the number of resources. We already know how to compute the absolute minimum number of resources needed, then we evaluate if more are needed or not.

So the input of the problem is the graph that is coming out of the scheduling, so the concurrency of the operations is already defined.

# 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

one approach to try to reduce the complexity of the problem is dividing in subproblems and considering for each operation the possible binding. Obviously, it makes no sense to try to bind operations that are concurrent and operations that need different resource types.

We analyze each kind of operation, and we establish if the operations are

- *compatible* or *in conflict*: these two properties are mutually exclusive

to be compatible

- two operations are of the same type if they use the same resource and so they can be analyzed in the same subproblem
- they must not be concurrent

if we try to negate the condition, to obtain non-compatible operations, since it was an and it becomes an or, so to be in conflict it is enough that they are concurrent or if they are of a different type.

So once we performed this analysis we can start building two graphs, one of which is the compatibility graph, graph where all the nodes and all the vertices represent the operations of that type and the edges represent compatible operation pairs

# 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**)

finding the best solutions requires to minimize the number of cliques but also having less cliques means many more operations in the single clique

## 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

once I have the compatibility graph I can always make the dual graph, that is the conflict graph, that is the dual graph in the sense that the nodes are the same and the edges are complementary. If there's an edge in the compatibility graph there's no edge in the conflict graph, if we don't have an edge in the compatibility graph I need to represent an edge in the conflict graph.

# 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

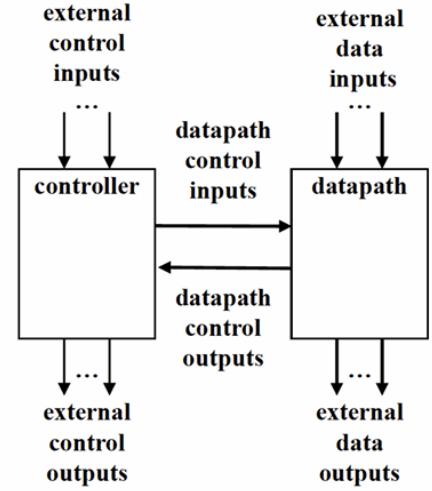
## 12 - Microarchitecture Creation

# 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:** determines the evolution over time of the FSM

**Datapath:** connection of resources with the outside world, in detail memories and other devices

Since scheduling has been performed, for each operation it is defined in which clock cycle it will be computed. So the controller is implemented with a Finite State Machine:

# 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

States in the FSM are serial. The number of states is equal to the *number of cycles that are necessary to perform that operation*. For each basic block that has been identified each state is connected to the state that corresponds to the next operation.

ex: *if there's an if statement there will be a block for the true statement and for the false statement. So the last block of the if statement will be connected to the first block of another basic block*

The FSM is implemented with a *next\_state* function and a *output\_logic* function. The *output\_logic* represents the outputs assigned by each state, which are dependent on the operation that is being conducted.

# 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

**Datapath:** is the portion of the microarchitecture dedicated to the connection of all the units. Functional units that execute the operations must be instantiated, then registers are added to save the intermediate values and wires and multiplexers to connect all the blocks.

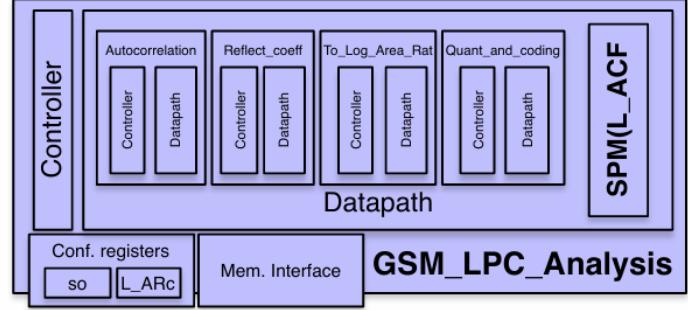
Since the output of a specific unit is the input of another unit, but the port of a unit might have more than one source, a corresponding number of multiplexers is needed to drive the signal. An equivalent input computation is performed to estimate the total number of multiplexers: if  $n$  sources must be connected, up to  $n - 1$  multiplexers might be needed.

# 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);
}
```



Memory usage in software is for storing bits, once we have data we have a lot of pointers.

Why and when use pointers?

- computing offsets
- dynamic resolution of addresses, because maybe I don't know in advance what the target array is
- access to an external memory or external resource

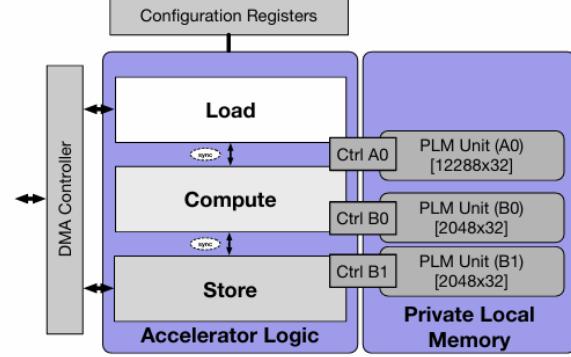
In this slide a complex accelerator from the memory point of view is shown: a top function at which parameters are passed as pointers, then there are local variables and then there are four submodules that exchange data and results through the external memory. A memory interface and configuration registers are shown.

# 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);
 }
 ...
}
```



Through system c it is easier to define dimensions of memory locations and signals. We're writing software but at the same time we are doing hardware design and we're defining logic

## 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

The pointer is the address of an element in this unified space, any operation on the pointer is an operation in the relative memory location. We don't use pointers or addresses for architectural elements

# 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

Pointer encoding in hardware is very complicated, let's understand why. Let's consider a pointer that

- interacts with a *single* memory location → it can directly be connected to the memory location
- interacts with *multiple* memory locations → all the memory locations that are pointed by the pointer must be connected

the problem is creating an efficient logic to evaluate and route and connect all the cases. In hardware, everything must be defined *at compilation* time and the designer must control where the data is stored. Once the memory space needed by the accelerator is determined, the required memory space must be partitioned in physical resources. These resources might be inside or outside the accelerator. During synthesis the memory units are connected to correctly store the data. The problem with pointers is that pointers must be able to access any variable, no matter where their information is available, pointers could refer to anything in the memory space. In software pointers dynamically change the object at which they point, in hardware it would require the implementation of logic to read the whole memory.

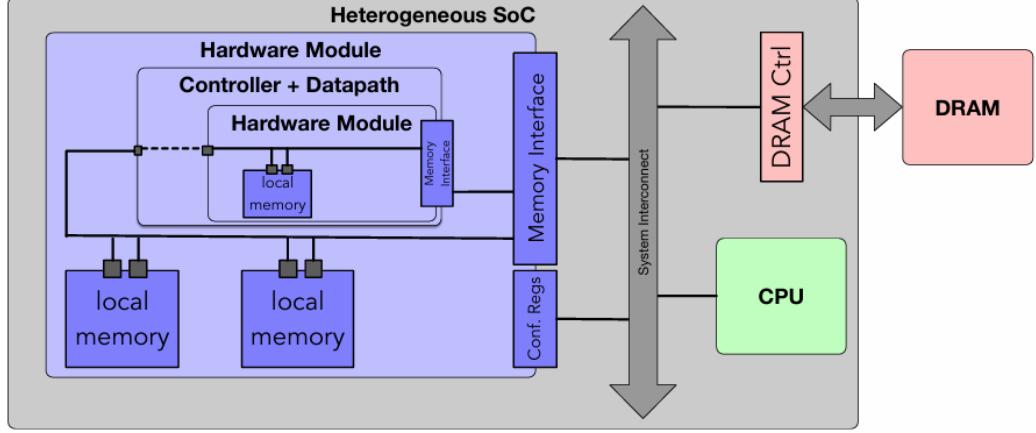
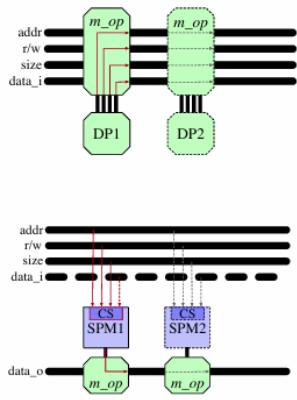
Arrays can be translated by allocating memory resources or register files, then the logic to read the data must be generated. Virtual addresses in software must be translated into specific memory resources operations.

An approach to solve this design issue is the **daisy-chain architecture for pointer arithmetic**:

# 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



The basic part of this structure is an internal bus that connects all the memories, starting from the inner ones to the outermost ones in a chain.

The outermost part that is connected to the *system interconnect* is the *Memory interface*, then deeper with respect to the memory interface the submodules are connected with their own PLMs.

This architecture allows us to put a request on the bus while an operation is being performed, the state machine of the controller will execute one operation per clock cycle, so it is guaranteed that only one unit is writing on the bus. Successive requests can be chained.

Partitioning is then performed: each address pointed by the pointer will correspond to one and only one memory location, it can't be determined statically but it can be put on the bus and there will be one and only memory location that will correspond to that request.

On the side of the memory identification is simple because it can be checked if the address on the bus is in the range of the assignment:

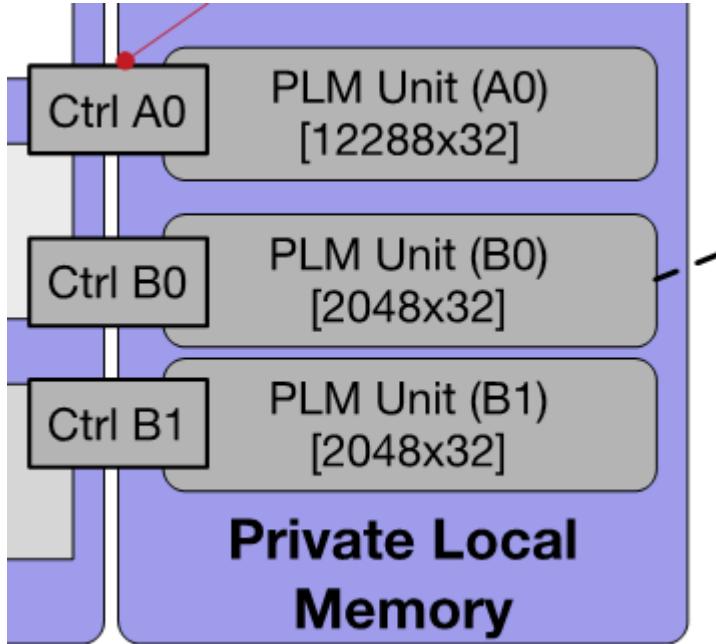
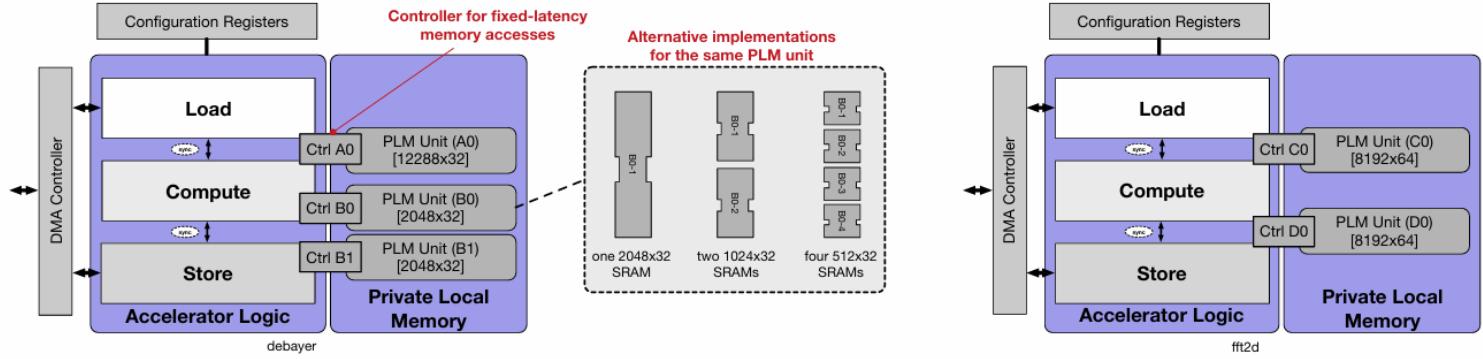
- If **it is**, that specific portion of memory is selected to reply to the request and the output value is put on the bus. Since a daisy chain of memory has been implemented, the request is performed, the answer is received and all the pointers can be resolved dynamically.
- If **none of the addresses responds**, the data is **not inside the accelerator**, so the request is forwarded to the external memory controller.

An approach to the implementation of the connection between memories has been explained, the question now is how to create the local 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**



Let's suppose that the *PLM Unit (B0)* must be implemented. A library of modules to implement SRAM memories is implemented and offers these options

- 1 · 2048x32 element
- 2 · 1024 x 32 elements
- 4 · 512 x 32 elements

*What changes from the memory point of view?*

There's no significative difference since the memory space is the same. What significantly changes is that with a single element there's a single port so the SRAM can't be accessed in parallel, while with 2 or 4 elements the port can be accessed. This is why *partitioning of arrays might be used*.

The problem is, how do I assign the elements of the initial array to the new array? An **access pattern** must be chosen.

Before choosing how to assign the data to the “new” module, it must be understood how data is assigned to the first module by means of data unrolling. (*let's consider as an example a two SRAM module implementation*)

1. **pattern 1 (no optimization):** if the first 1024 bits are assigned to the first memory and the other 1024 bits to the second one no significative advantage is obtained because two consecutive memory operations will be insisting on the same block and the other one is inactive
2. **pattern 2 – cyclic partitioning (optimization):** the even bits could be assigned to the first memory block and the odd ones to the second block, so that while one operation insists on the first block, the successive insist on the second block and could be parallelized. This is called *cyclic partitioning* because all the blocks are being used with a *round-robin* approach.

Obviously, this is a simplified case because the assumption is that operations are performed on the array in a sequential fashion. *The more regular the loop is, the easier it is to determine an optimized access pattern.*

If the loop is *irregular* the access pattern becomes irregular and determining how to parallelize the access becomes quite more complicated. One approach to irregular loops could be *duplicating the data*, so that one access can be executed on the first block and the next on the second block independently of the position of the data being accessed. The drawback is that to parallelize access, the memory dimension is doubled. This is also the only available option when the access pattern of the loop is completely unknown.

## 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

Idea of **Array partitioning**: transformation of the array to create independent data structures that can be accessed in parallel, exploiting more memory resources, so each substructure becomes a possible new array that is mapped into a memory resource and the data is distributed in the way that follows the change of indexes of the array.

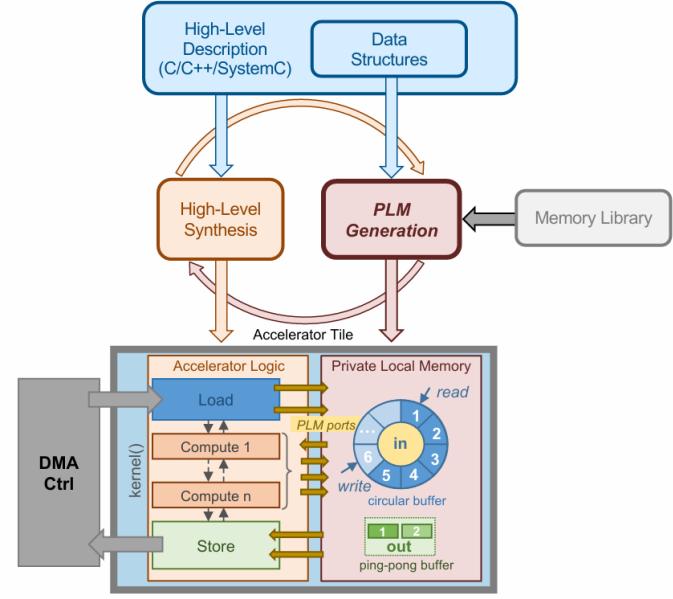
# PLM Customization for Heterogeneous SoCs

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

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

## Generation of specialized PLMs

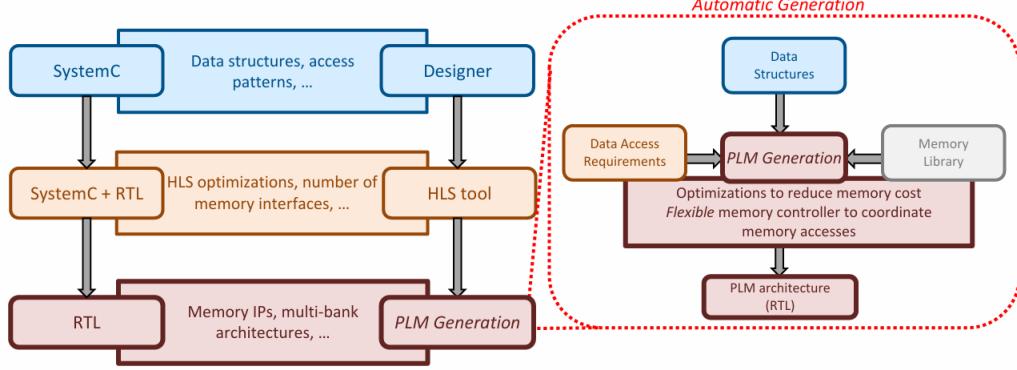
- Technology-related optimizations
- Possibility of system-level optimizations across accelerators



The architecture must be specialized as much as possible. Through HLS the logic for the computational part is created, but the creation of PLMs is still “handmade” because it’s a critical part that must be as optimized as possible, even technology related optimizations must be considered and implemented.

# 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)

In this slide are represented the two approaches to PLM customization:

- **Performance optimization:** the *performance is optimized* by assuming that the memory architecture can always be generated in that optimized way. The HLS defines the accelerator logic and the possible parallel accesses.

- **Cost optimization:** the assumption is that *it will be possible to generate a memory architecture, always able to sustain the accelerator performance.* Is it always possible? Yes, by using buffers.

# 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

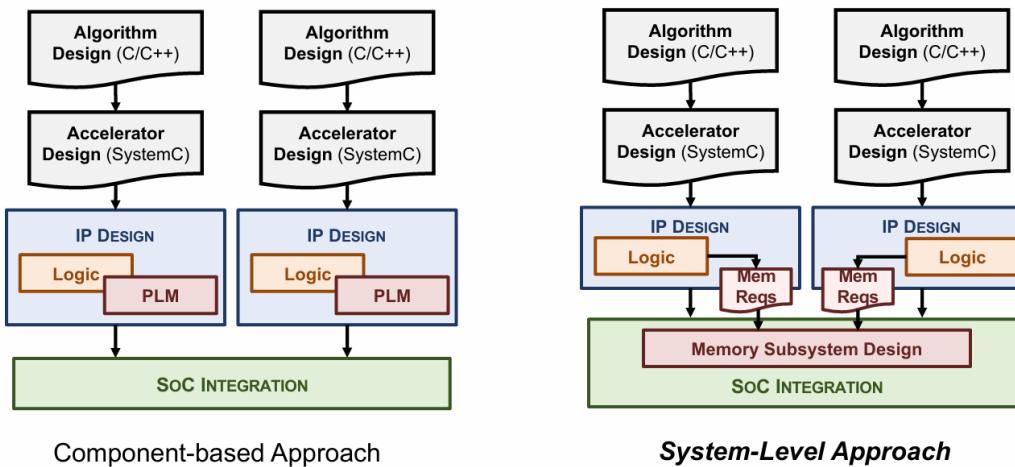
- Possible only at system-level, when integrating the components
- Optimizations of accelerator logic and memory subsystem are independent

One PLM for each array, the index that is used to access the array becomes the position inside the memory. The idea is that if two arrays that are never alive at the same time, they can reuse the same memory space.

## Optimization only at the System-Level

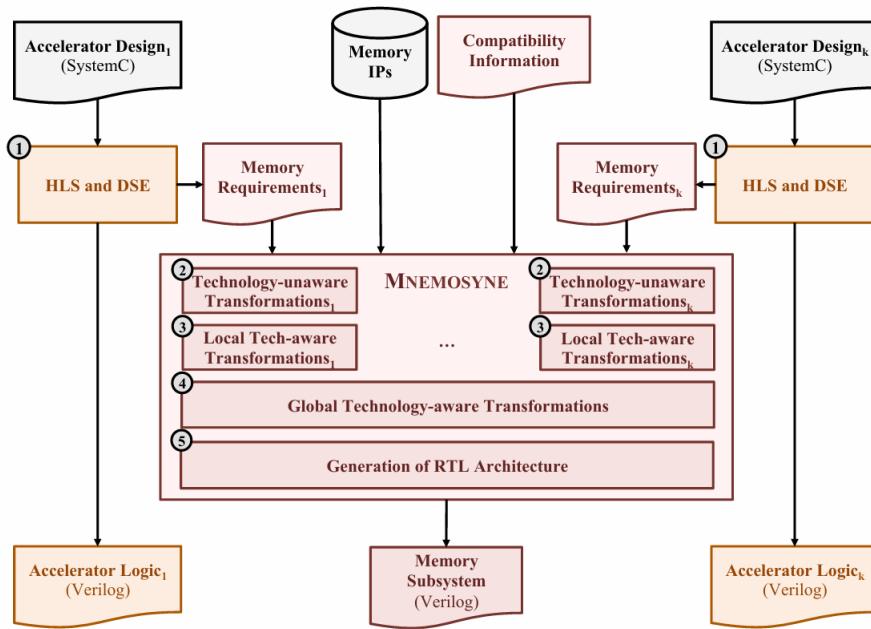
Accelerator(s) memory subsystem is defined **during SoC integration**

- Possibility for **more optimizations**



The difference between the two approaches is that the memory is created to satisfy all the requirements and then a single memory architecture is created for the entire system, in order to reuse all the elements and to reduce as much as possible the area occupied by the memory.

# PLM Optimization for Multiple Accelerators



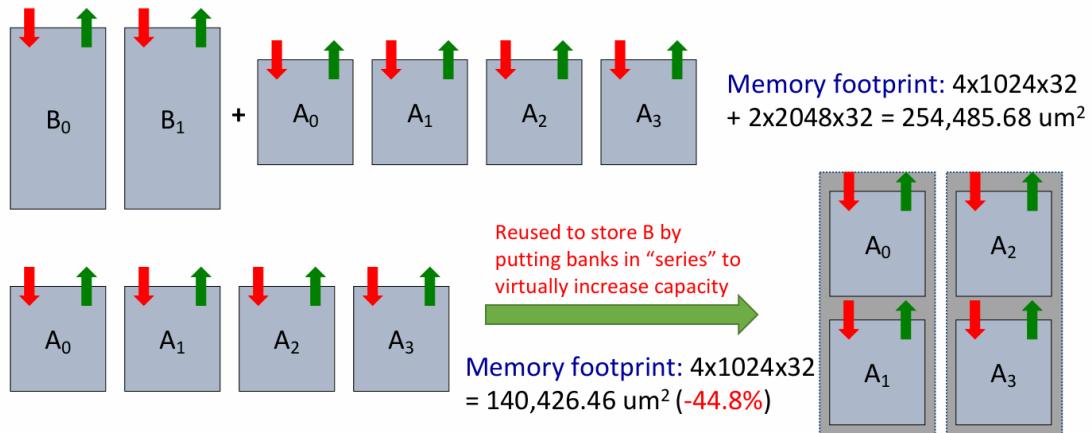
Design chain:

1. Design the HW accelerator, perform the HLS and Design Space Exploration to create the accelerator logic with the specific requirements.
2. Apply transformations that are not technology aware like splitting and merging of data.
3. Local memory transformations, aware of the technology, to hit the required performance target.

## Address-Space Compatibility

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

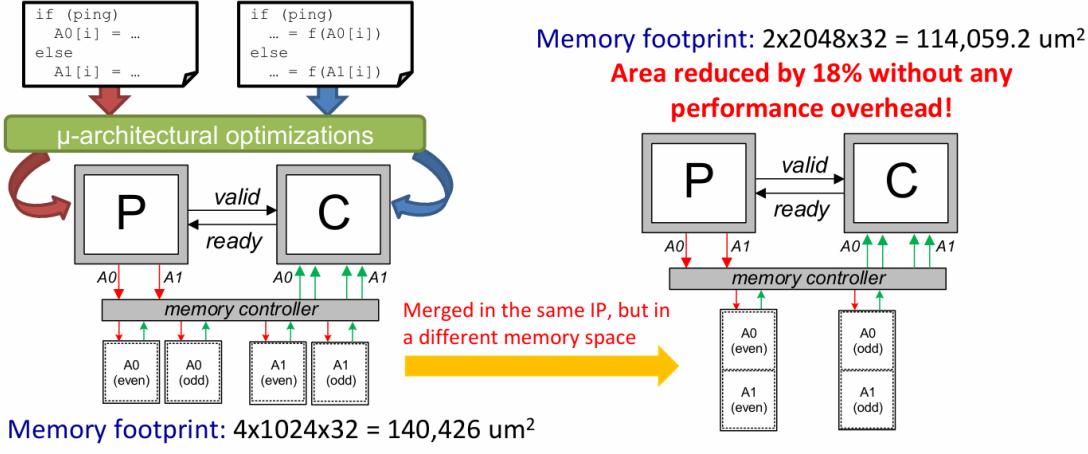
- A[1024] with *data duplication* over 4 parallel banks
- B[4096] with *data distribution* (cyclic partitioning) over 2 parallel banks



# 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)



In this case *it is not used the same memory space*: data are alive at the same time but *ports are never used in the same moment in the same way*. A ping-pong buffer is implemented:

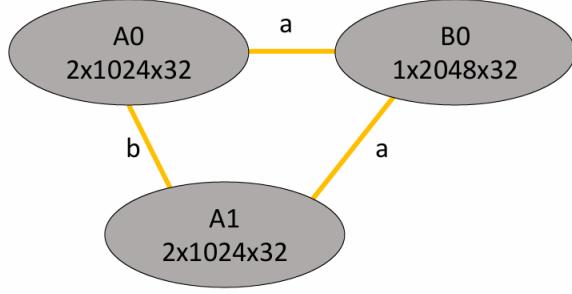
- when C is reading one memory location, P is writing in another one and vice versa

they can be put in the same memory space to reuse the same ports and reduce the area.

## 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

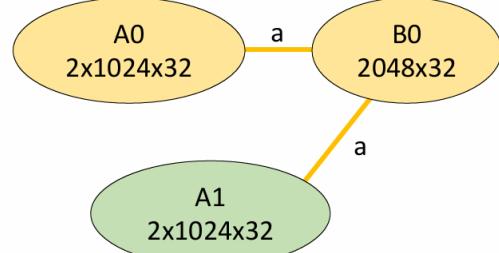
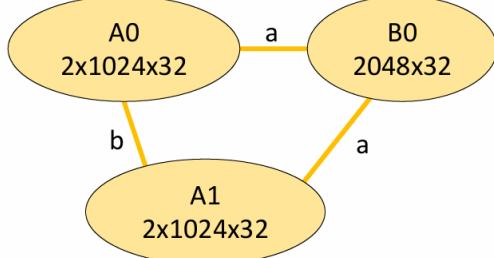
Three arrays must be implemented with compatibility:

- A0 B0 are *address base compatible* → can be placed in the same memory space (same for A1 B0)
- A0 A1 are *memory-interface compatible* → same data footprint after data allocation

# 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

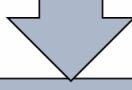


We need two distinct configurations!  
**{A0,B0}** and **{A1}** or **{A1,B0}** and **{A0}**?

## How to Determine the Memory Subsystem

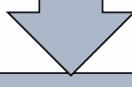
### Clique Enumeration

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

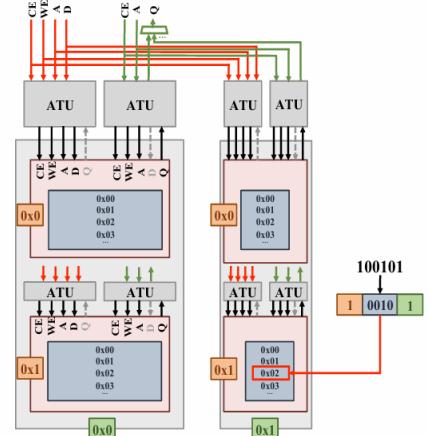
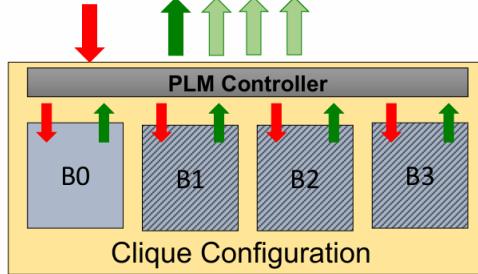
To determine how to partition the MCG such that the total memory cost is minimized

How can we solve a graph like this? We can do basically clique partitioning?

# PLM Controller Generation

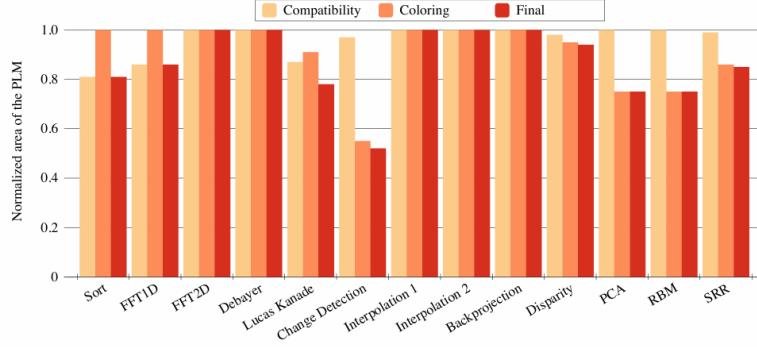
A **lightweight PLM controller** is created for each compatibility set (clique) based on the bank configuration

- 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

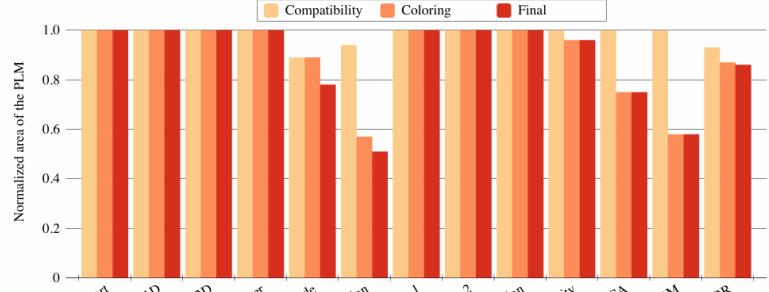


### Industrial 32nm CMOS technology

- Memory library with 18 SRAMs

## Xilinx Virtex-7 FPGA

- Memory library with 6 BRAM configurations



with optimization we can also reduce area by 50%

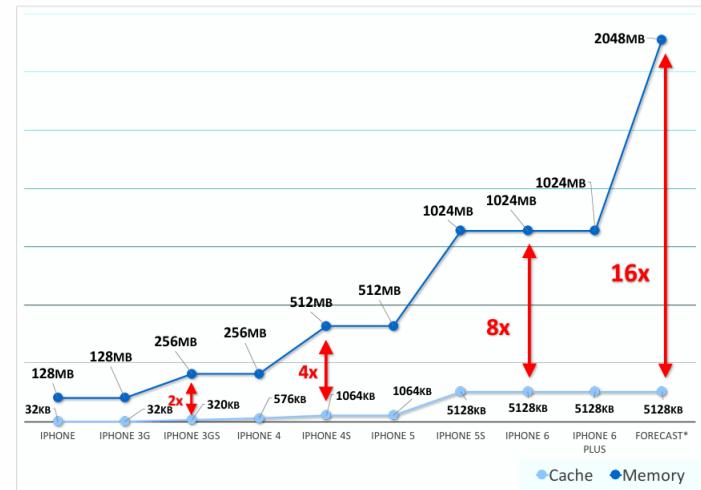
# Off-Chip Memory: The Large Data Set Problem

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



Until now **on-chip memory** has been treated, now the **off-chip memory** problem, for large datasets, will be assessed. As shown on in the slide, on chip memory could even be  $\frac{1}{16}$  than the required off-chip memory.

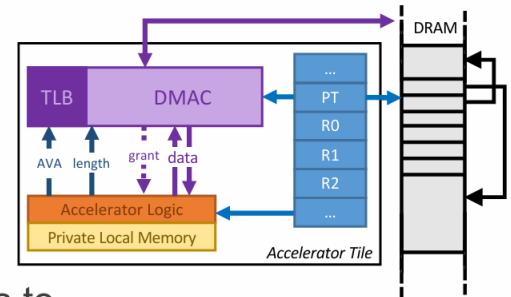
## Scatter-Gather DMA for Accelerators

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



This slide refers to a method of accessing memory where *data chunks can be scattered across different locations* in memory but can be *gathered efficiently* when needed by the accelerator.

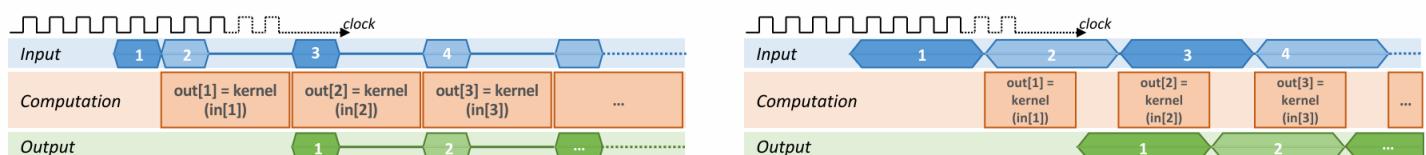
The idea is using **equally-sized large** chunks of data instead of using small memory pages, this mean that *the memory is divided into larger, fixed-size blocks*. This makes memory access simpler and more efficient. Then a **small page table** is implemented, because since the chunks are large and equally sized, the page table (that is used to map virtual to physical addresses) can be small and simple. This kind of design reduces memory

management overhead, speeds up translation and data access and enhances scalability and performance in multi-accelerator systems.

# 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

Both **computation** and **communication** problems must be balanced. It makes no sense to have a very fast computation and not being able to transfer the elaborated data *without bottlenecking the device*. It's a waste of computational or communication resources.

## 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

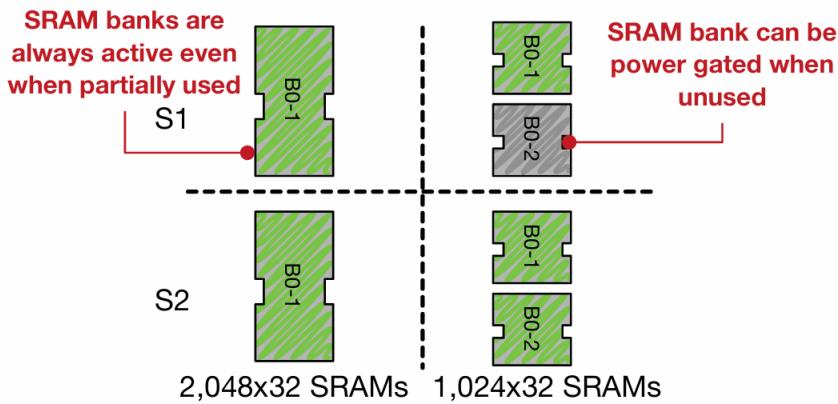
- Possibility of fine-grained power savings

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



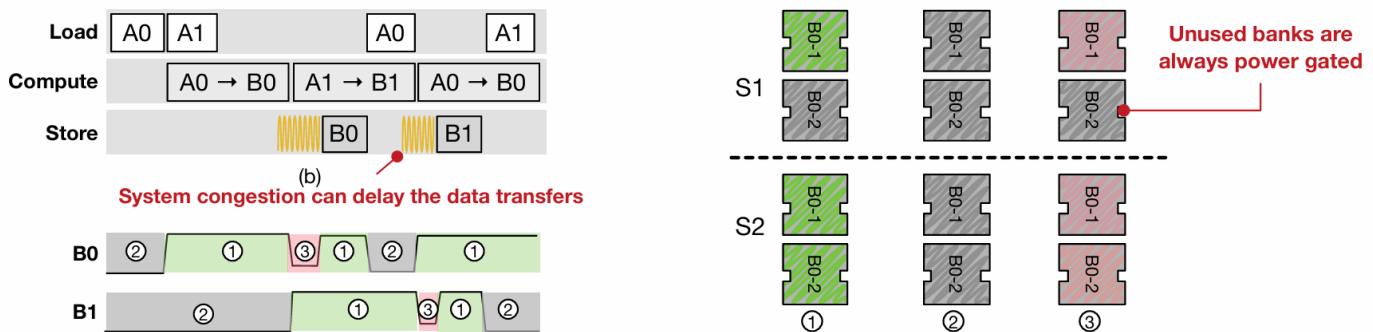
# Workload-based Optimization

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

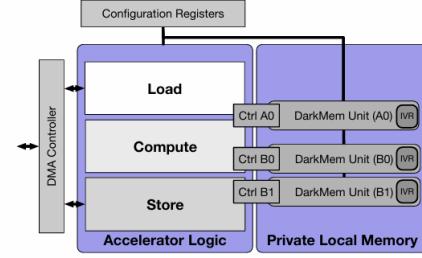
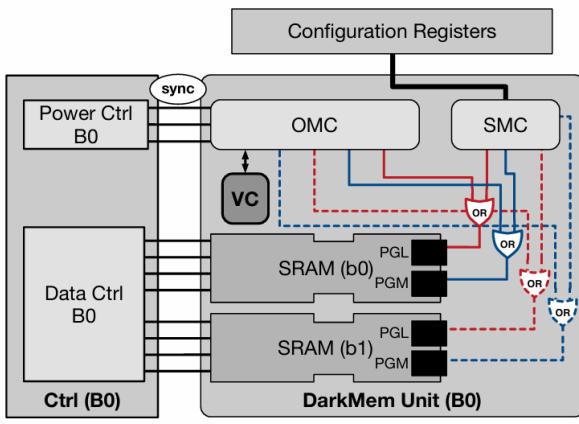
- Three operating modes: *active*, *idle*, *deep-sleep*



# 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)

In this slide the “**DarkMem**” architecture is presented, this is an extension of a PLM design with fine-grained power-control logic to optimize power usage in memory intensive hardware accelerators.

Two significative components are added:

- **SMC (Scenario Management Control)**: identifies the current execution scenario based on the values of the configuration registers and determines **which part of the memory needs to be active**.
- **OMC (Operating Mode Controller)**: manages the SRAM operating nodes based on the signals from the accelerator logic, it controls **power-gating** and **mode-gating** for memory banks.

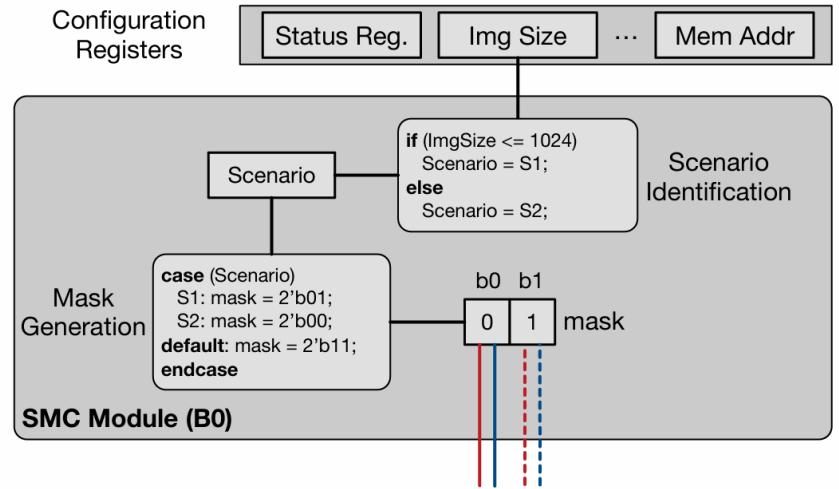
This design enables *dynamic, power-aware memory management* by adapting SRAM power states to the current processing needs, using two control modules (SMC, OMC), this results in energy-efficient memory usage in accelerator-based systems.

# Scenario Memory Controller

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

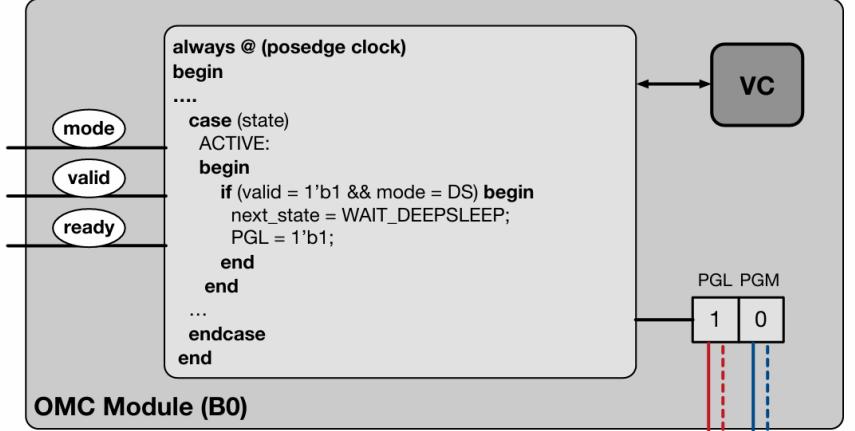
- In each scenario, only the used banks are kept active, while the others are power gated

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



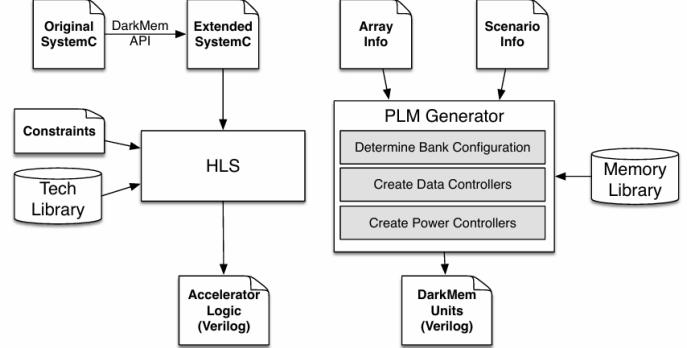
# Operating Mode Controller

- 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



©Christian Pilato, 2024

34

## Determining the Bank Configuration

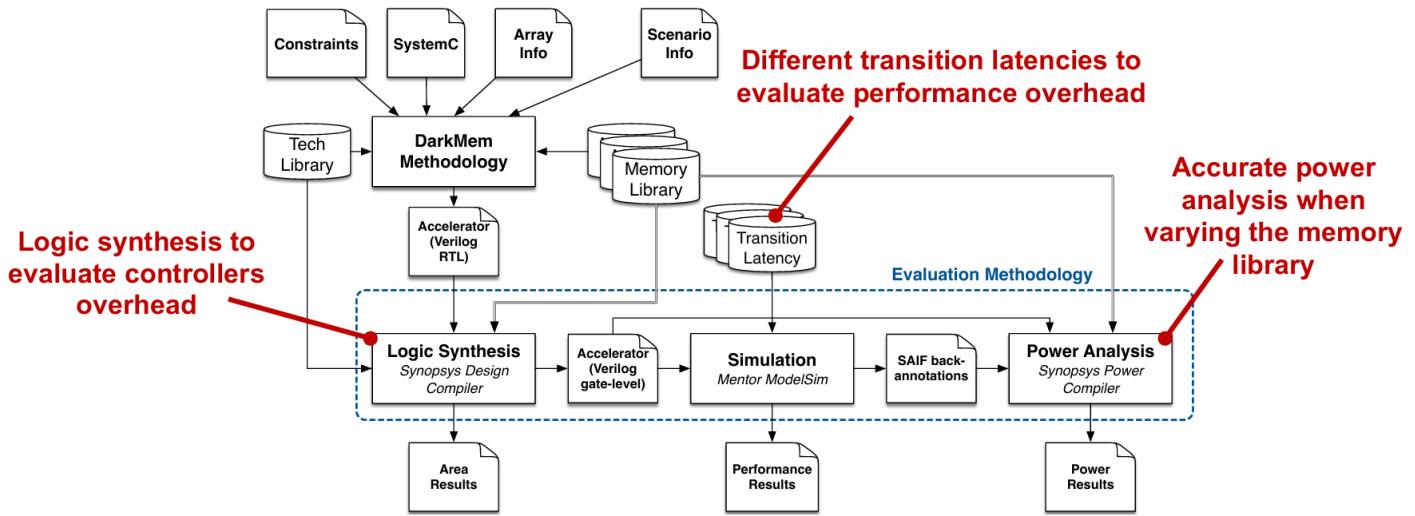
- **ILP formulation** to determine the number and type of banks for each PLM unit, based on:
  - List of scenarios and frequency of execution
  - Data to be stored (bitwidth and number of words) in each scenario
  - List of available memory IPs and corresponding active/gated static power configurations

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

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

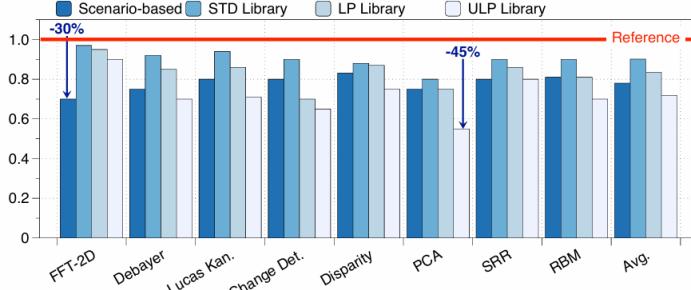
# Evaluation Methodology

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

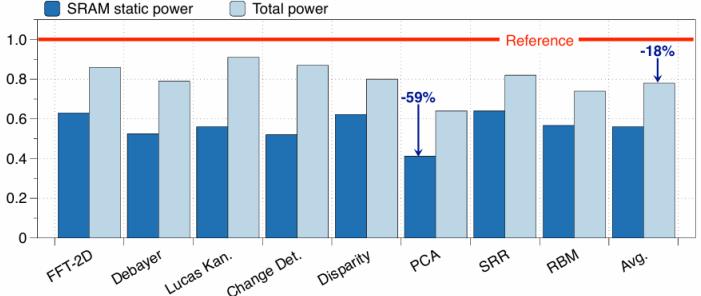


## Effects on Accelerators

- 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%)



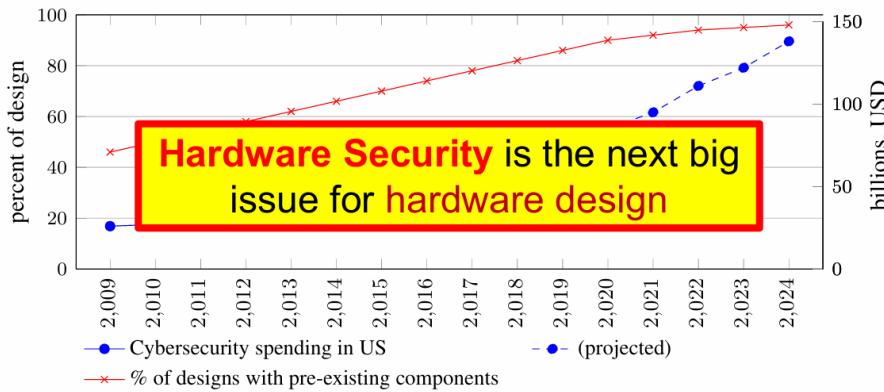
SRAM static power can be reduced up to 60% (on avg., total power is reduced by about 18%)

## 13 - Hardware security

# 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



Designing components is complex and often it is not necessary to redesign every component from scratch, this is particularly important for *steady devices* that must be updated generation by generation.

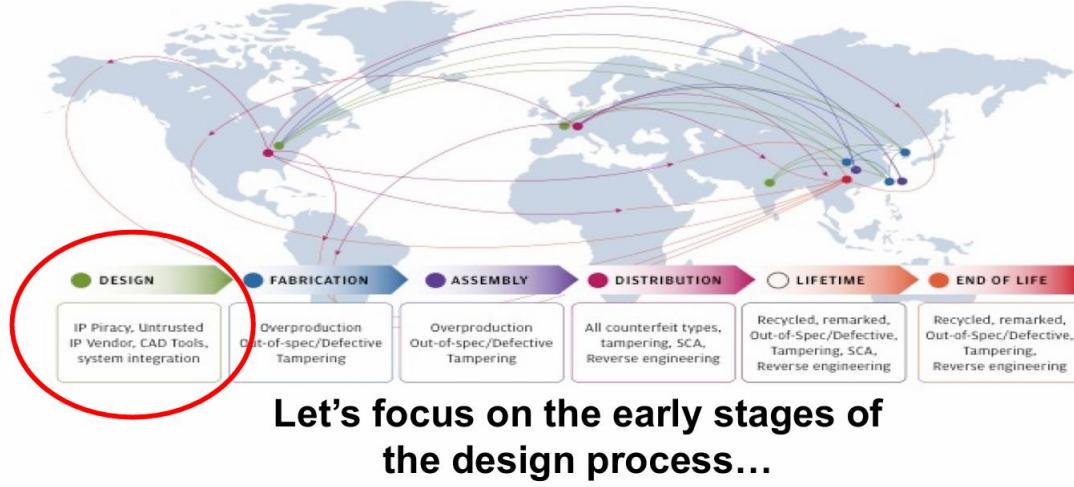
*ex: new generation of Iphone, we do not redesign every component, we extend only with new features, I do not want to redesign an antenna device or usb-c handler. Most importantly I want to integrate components designed by other companies, this is due the hyper specialization of components. There are few industries that design the whole SoC while most of the other companies design a specific component with highest quality because they're specialized in it and then sell the IP. As an integrator we must trust all these parties, because there are many cases of tampered devices that then have been exploited to perform attacks. There's an increasing concern about security of devices in general, this is coming together with the reuse of devices. We save money on the design but we move the investment to the integration of components, on top of this we integrate it in an SoC and then we must fabricate it. Obviously small companies cannot have their own fab, so they must go to a third party, give them their design and ask them to produce their chip.*

So we need guarantees that the design is secure.

# 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



The direct effect of third party components and fabless companies lead to the globalization of the supply chain of semiconductors.

**Globalization:** globalization is the idea that a specific process is spread across the world. This is a way to reduce cost but also introduces lot of security concerns.

In the slide is shown the entire supply chain of an electronic device:

*Green dots represent* the design phase, where the product is originated, the design is made and we see some distribution of the design (US, Europe, India...) why so?

1. It is convenient because in some countries the labor is cheaper. That is possible because the developing environment is decentralized. The core of the design center might be in Poland, then people can be distributed across Europe because they work from home. Cost is the driving factor.
2. Even if the entire process is distributed, companies can still claim that the design is made in specific country because maybe it starts/finishes in one country, so for a marketing approach the consumer can maybe read “designed in California”, then in reality the production is more very distributed.
3. Fabrication of semiconductor devices is concentrated in the east, because it is the region where they invested the most to have very advanced production processes.
4. Assembly of the devices is also distributed. Some legislation allows that if the last part of the production is made in one country, then you can say the whole process is in that country.
5. Distribution and lifetime are by definition distributed, they are in all the markets in the world and consequently also the lifetime.
6. End of life: this is an issue for hardware security, *ex in military related applications, if you lose a specific device on the war field, others can access it.* Many of ICs are still working even after the death of a device, so people can still access it and *recycling of IPs* is a way to steal intellectual reason, components that are at EOF are used to reuse or to reverse engineer the device.

There can be lots of malicious actors in the semiconductor process, also *tools and integration* can be untrusted, during fabrication overproduction can be used maliciously.

The idea is to try to focus on embedding and protecting during the design phase, so that the approach is more effective.

# 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
- ...

**reverse engineering and IP theft:** an attacker can extract the functionality of the chip in order to create illegal copies

*ex: I design an Iphone and then we find an equivalent product because someone copied the design.*

By stealing and coping the function the design cost can be cut and the company can have a better position in the market by having to spend less to reach the same results.

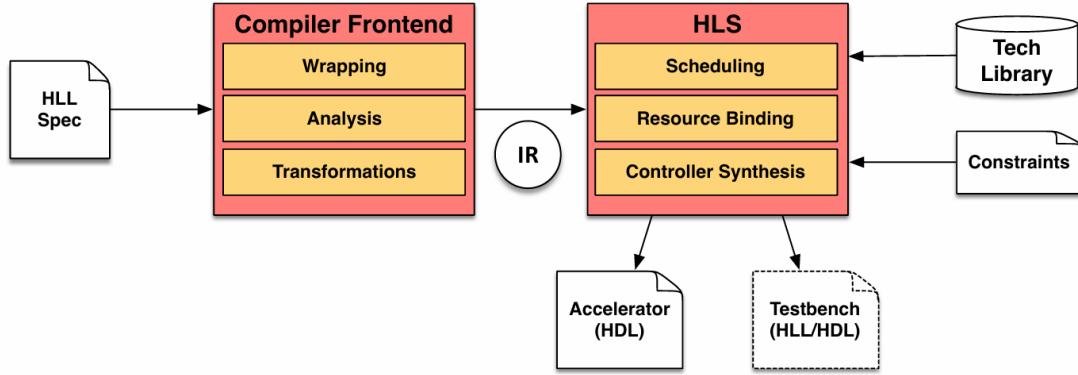
Note that these are not independent threats, they are strictly interconnected, one violation can support the others.

*ex: by means of reverse engineering, a trojan might be introduced*

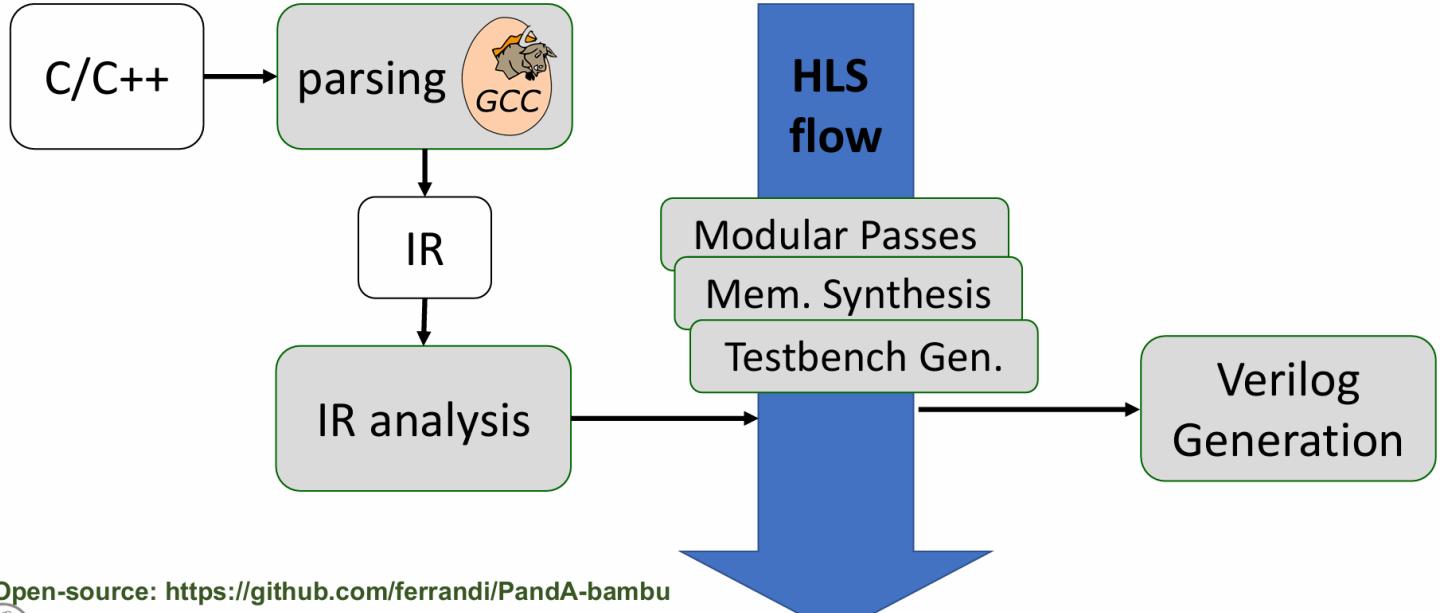
# High-Level Synthesis

**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



Open-source: <https://github.com/ferrandi/PandA-bambu>

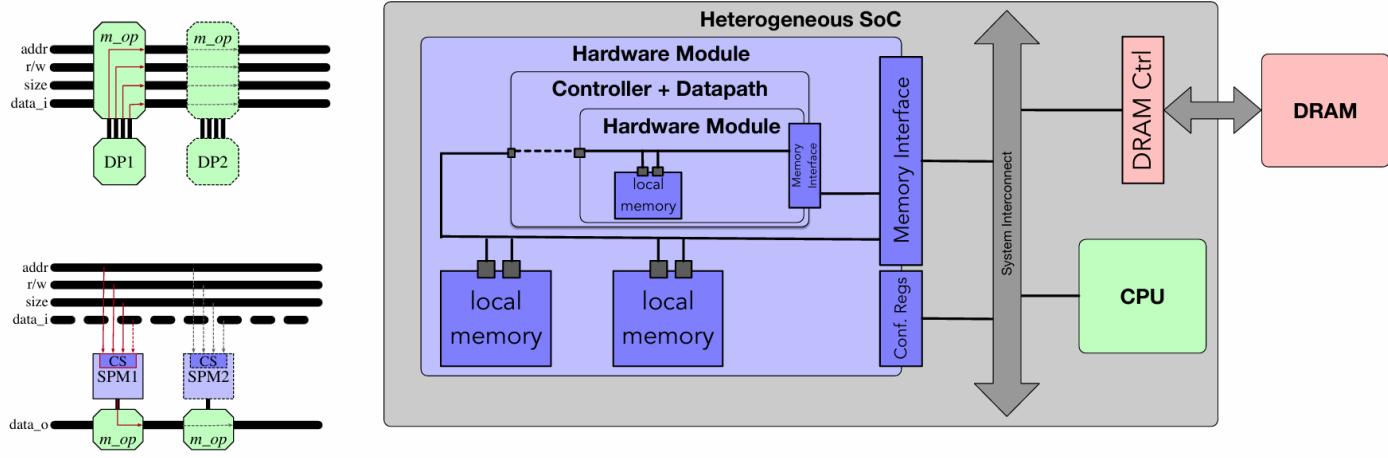
Bambu is based on modular passes, every step is a pass, these steps can be extended to introduce new functionalities.

ex: *complex memory synthesis, with a daisy chain etc. and then the testbench generation to verify the module can be implemented as other steps*

# Bambu: HLS Microarchitecture

Hierarchical synthesis of the C functions (based on **call graph**)

- Internal “memory bus” to dynamically resolve the mem address



This approach had previously been analyzed to implement the same approach as in software. It is interesting analyzing it from a security point of view.

If, for some reason, there's a security threat in software and the same exact implementation is kept in hardware, the security threat is “translated” in hardware.

## 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

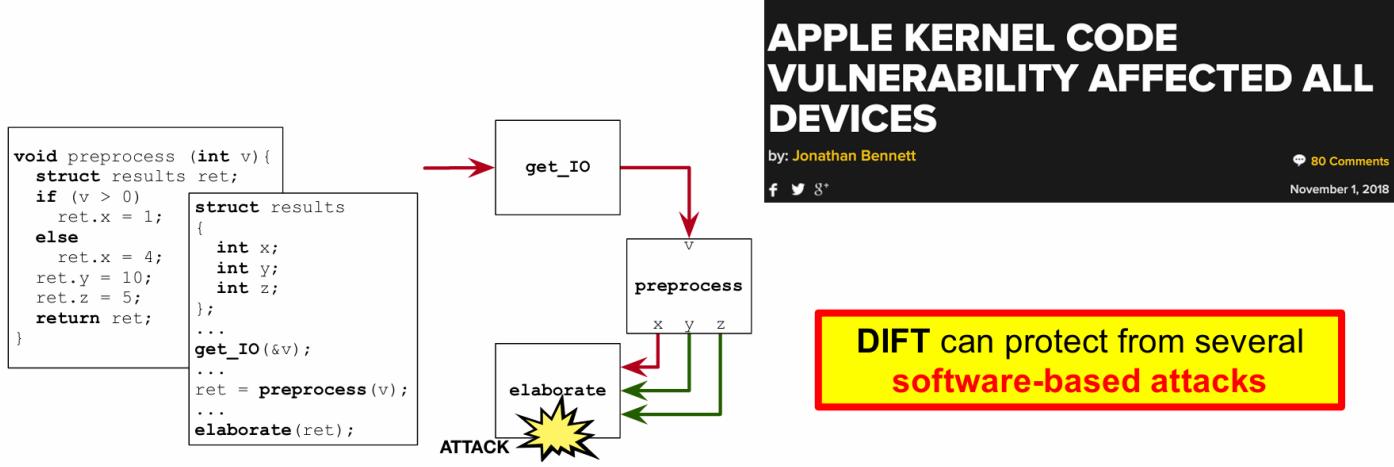
**Ugly:** Dream vs. Reality

- What is Missing?

# Dynamic Information Flow Tracking

## Marking Data coming for untrusted sources with tags (**taints**)

- Trap to OS if tainted data are used in critical operations
  - Pointer dereference, jump address, modified code or data, ...



**Dynamic Information Flow Tracking:** it's the marking of data coming from untrusted sources with tags called *taints*. If *tainted* information is used in *critical operations* the OS is trapped.

Every piece of information coming from the user is *tagged and marked as untrusted*, because the behavior of the user can't be trusted. Then the information is propagated to understand if the input from the user is able in any case to reach sensitive operations, like memory or communication operation. If this is the case, there's a security concern, because it is possible to reach critical operations from the outside.

```
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);
```

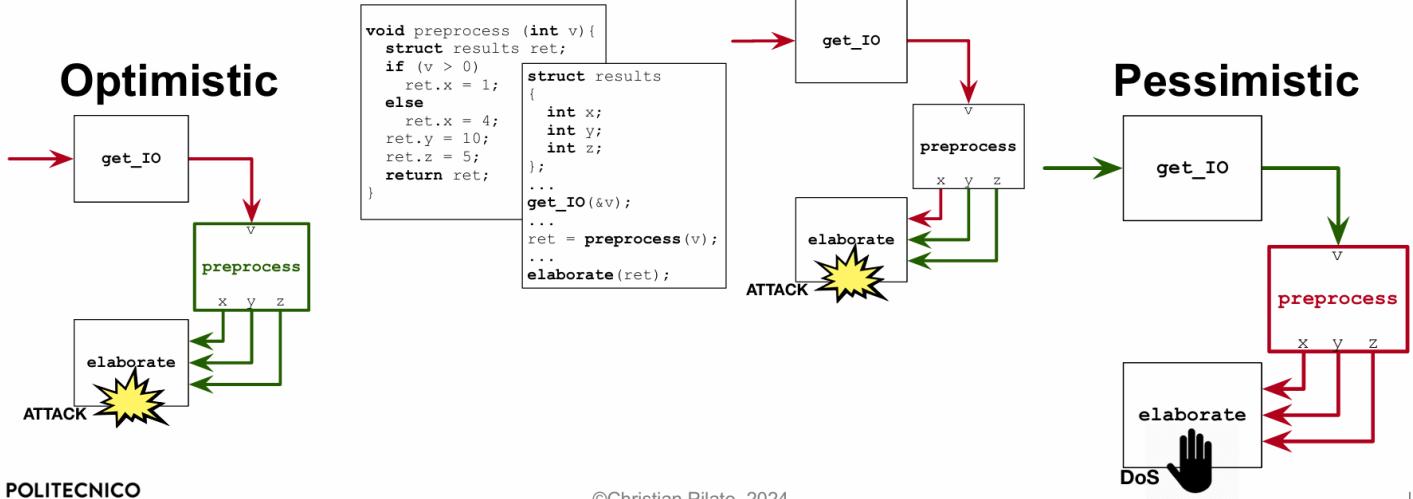
ex: *y, z are not tainted, x is tainted because its value depends from the action of the user. If the user introduces a v greater than zero or lower than zero, x depends on its choice, so it's tainted.*

Speaking of accelerators, they won't work alone but will have to interact with lot of different systems and might work with information that depends by the user, so that's why this approach is taken. The concept is not complex, but how can we apply so?

# DIFT in Heterogeneous Architectures

Applications interleaves tasks in both hardware and software

- What happens when accelerators are executed *before* the potential attack point?



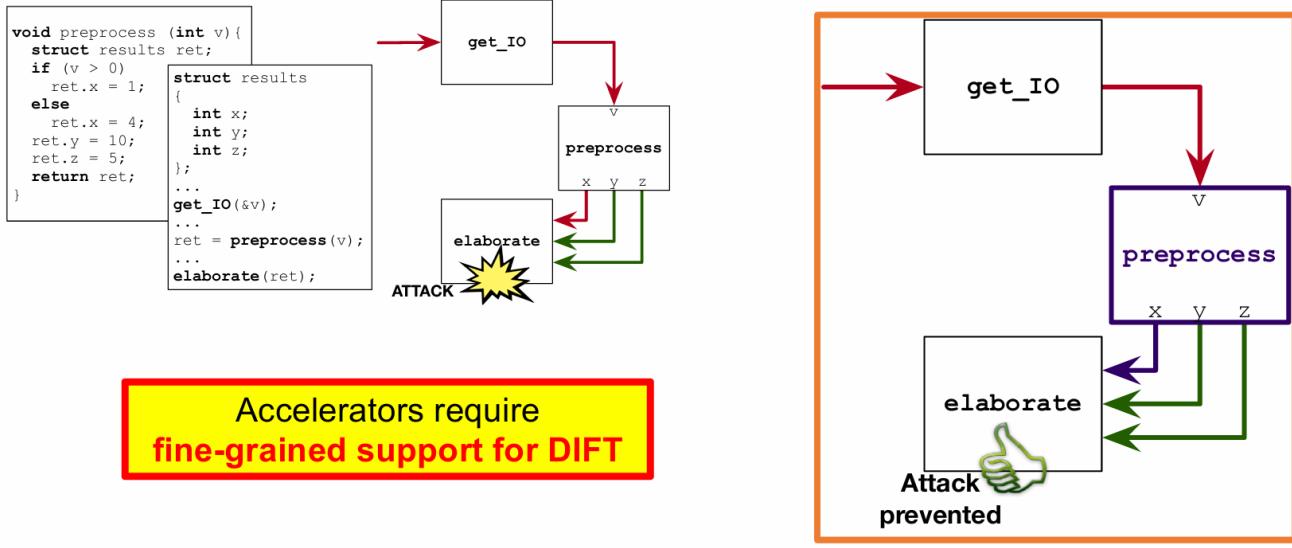
- **optimistic assumption:** the accelerator filters all the information and vulnerabilities that I have
- **pessimistic assumption:** the accelerator doesn't filter any information and nothing that comes from the accelerator can be trusted. Safe data might be provided but the choice is not trusting anything coming from it, so the OS will stop the execution because of security protections. This would stop the execution in any case.

Nor the *optimistic assumption* nor the *pessimistic assumption* are correct and can be applied, the correct solution is implement DIFT inside the accelerator, to correctly propagate the correct good information.

# DIFT in Heterogeneous Architectures

Applications interleaves tasks in both hardware and software

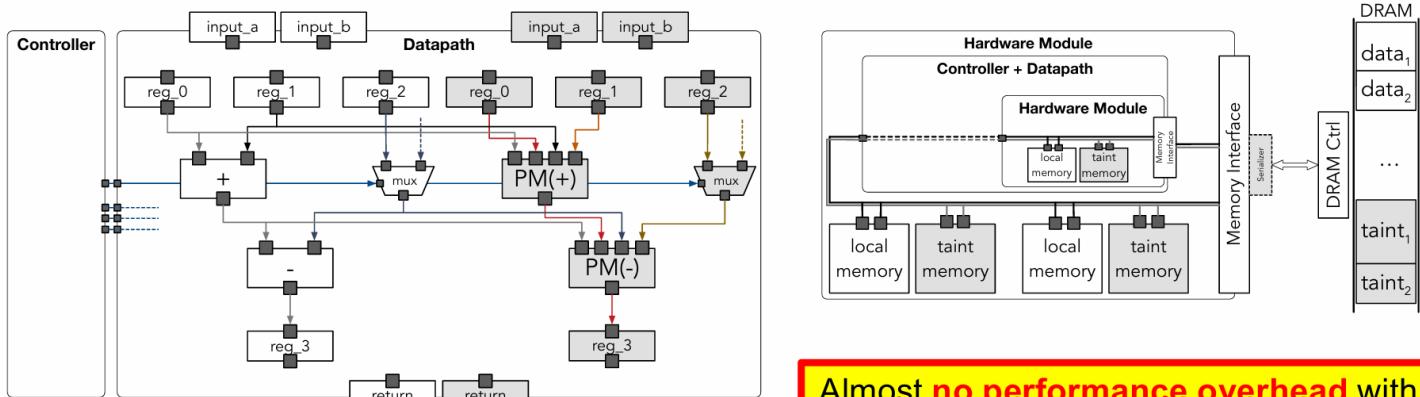
- What happens when accelerators are executed *before* the potential attack point?



# TaintHLS: DIFT Support within HLS

Data path extended with **shadow logic** and memory architecture with **taint memories**

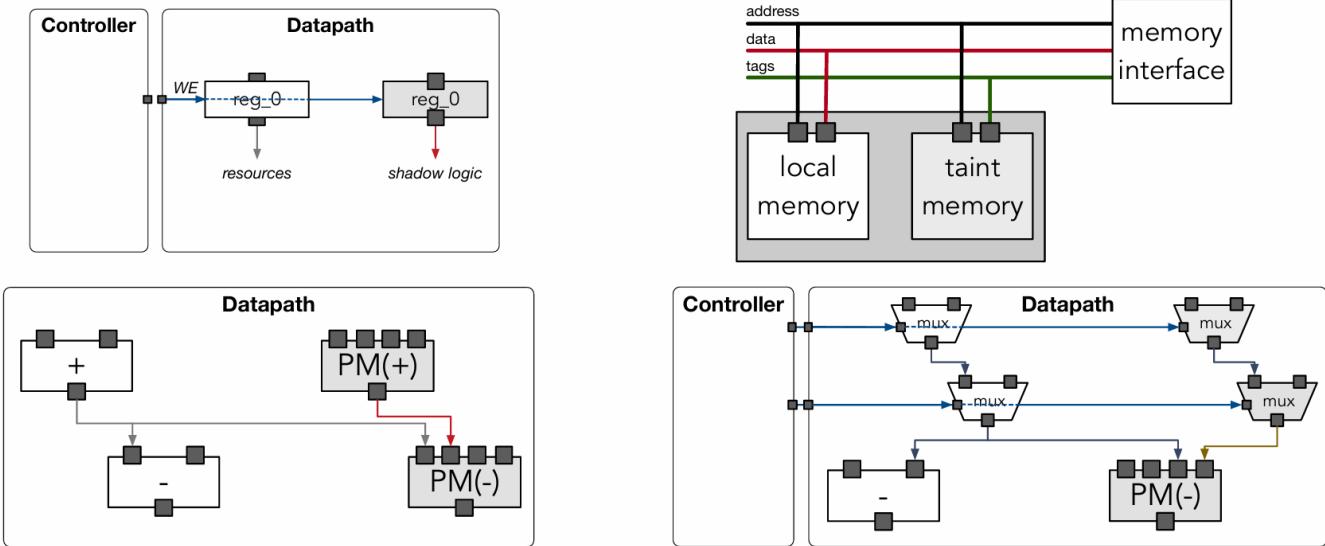
- HLS-based methodology for **automatic generation** based on **HLS results**



TaintHLS is a tool methodology that applies taint analysis to HLS designs. Taint analysis is a technique used in hardware security to track the flow of sensitive or untrusted data through a program. It's useful to find vulnerabilities such as information leaks, injection points, insecure data handling.

## Data Flow Consistency

Microarchitectural solutions to propagate data and tags in parallel

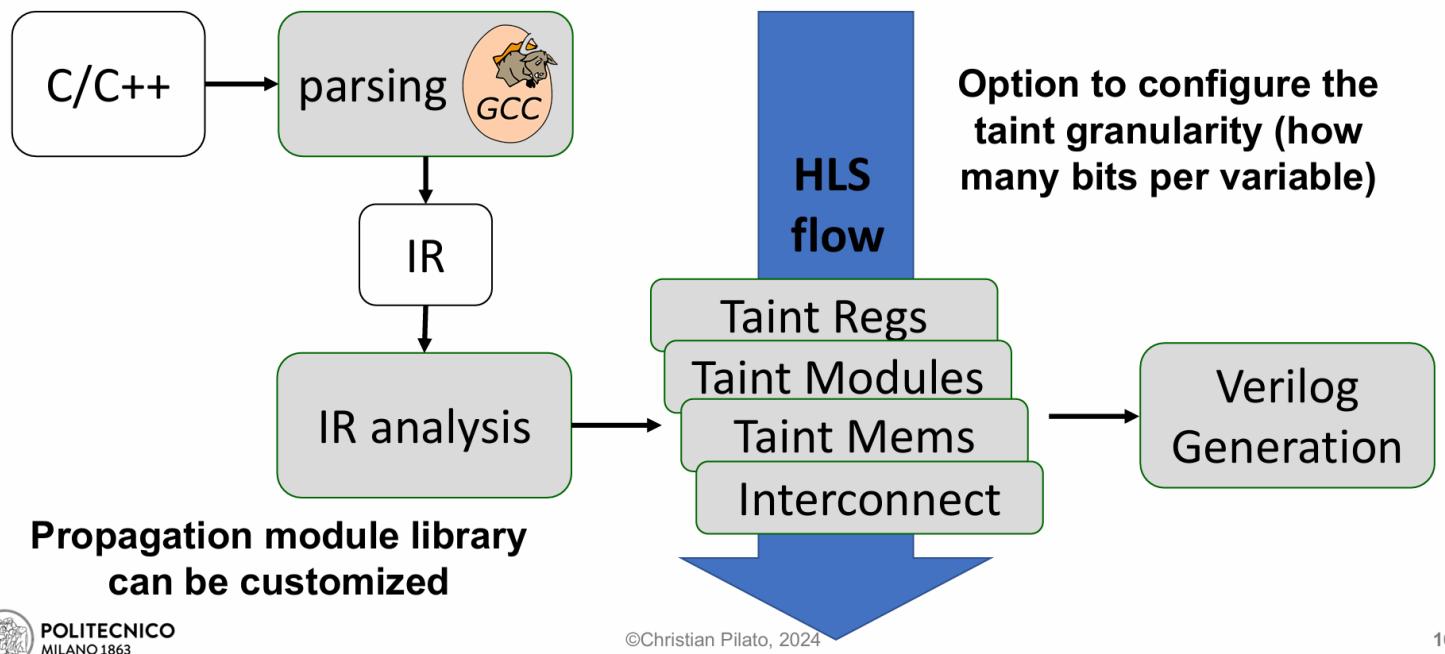


in every moment that we analyze our system, we can screenshot the execution and have the correct tags for that specific data.

ex: when I send a write enable to that register I am writing data in the register, so I activate the tag and sample the tag value, then for the memory I have the address in common and then I have an extra bus carrying the tags, so that the operations can be done in parallel. when I have the operations in the datapath I share the path for the propagation modules. in this case, we will send the same the control signals to both MUXs of the logic

This is easy to be implemented in HLS

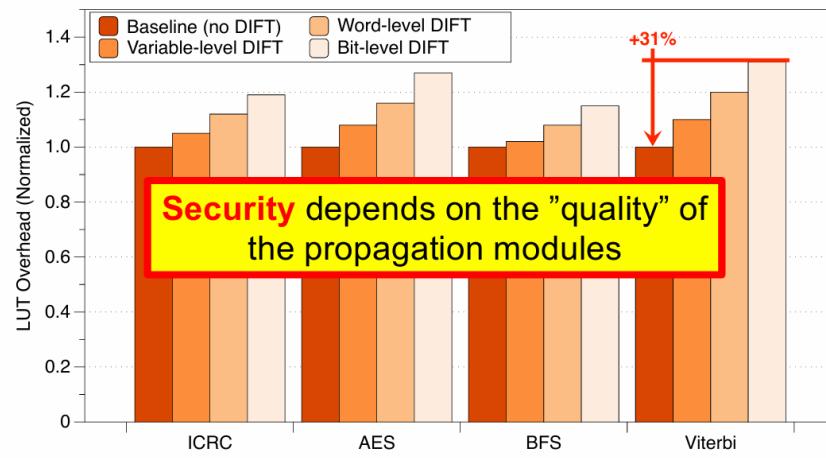
## Bambu for DIFT-enabled IPs



## Area overhead

Area overhead of each granularity wrt the **baseline** version

- Xilinx Virtex-7 FPGA @ 100 MHz



The redline is without any information flow tracking, then we have different levels of tracking and we can see the area occupation.

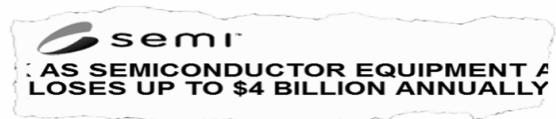
if we can guarantee the same result of DIFT in software and in hardware we can guarantee higher security

# IC/IP piracy and overbuilding

- Steal and claim ownership of IC and/or illegal use
  - Malicious SoC integration house
  - Malicious foundry

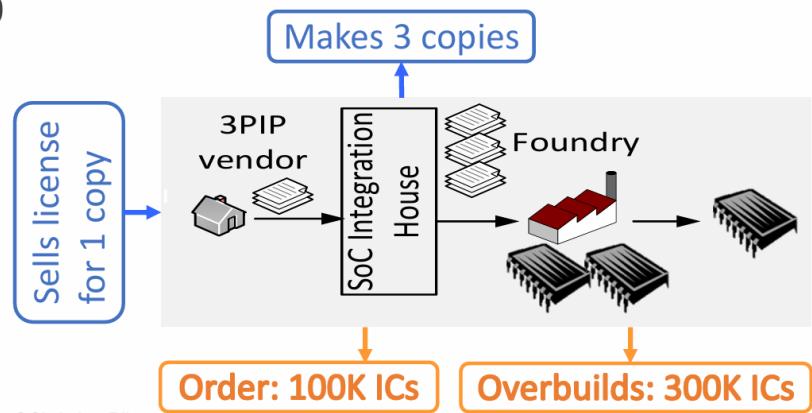
- Real-life impact

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

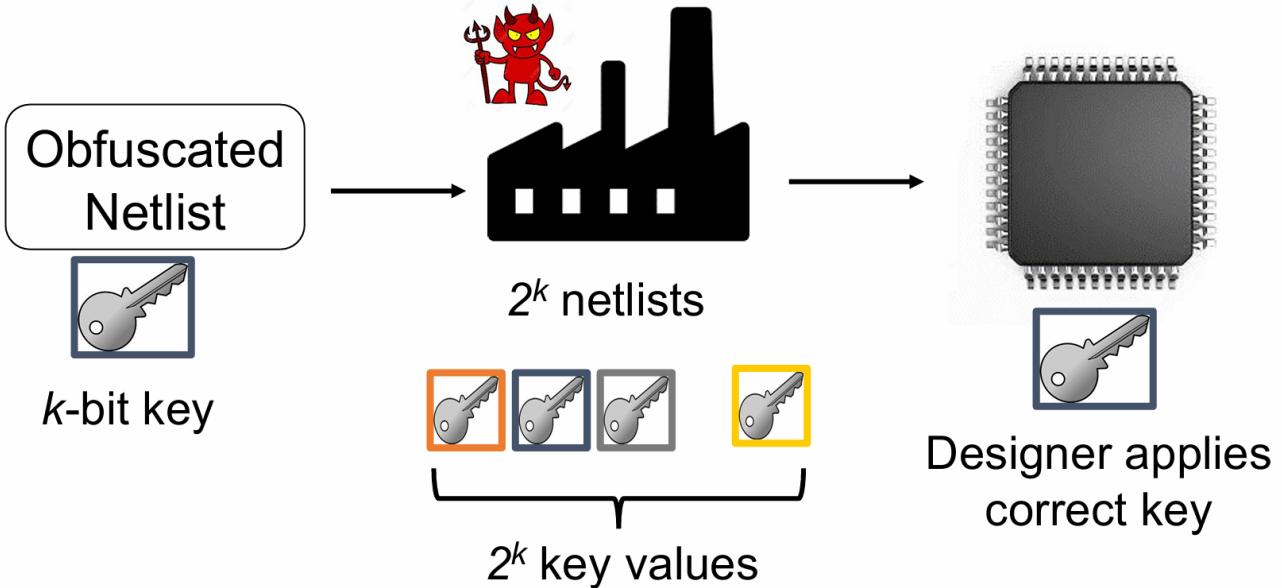


**EE Times**

ARM files patent infringement suit against IP startup picoTurbo



## Logic obfuscation



Logic obfuscation means locking the circuit in a way that makes it dependent on a  $k$  bit key. Then the design is sent to the foundry without giving them the key, so that the design can have  $2^k$  netlists, then by applying the correct key it can be used.

It is hard to implement because it must be implemented in the design by having wrong outputs when the correct key is not applied and good outputs only when the key is applied.

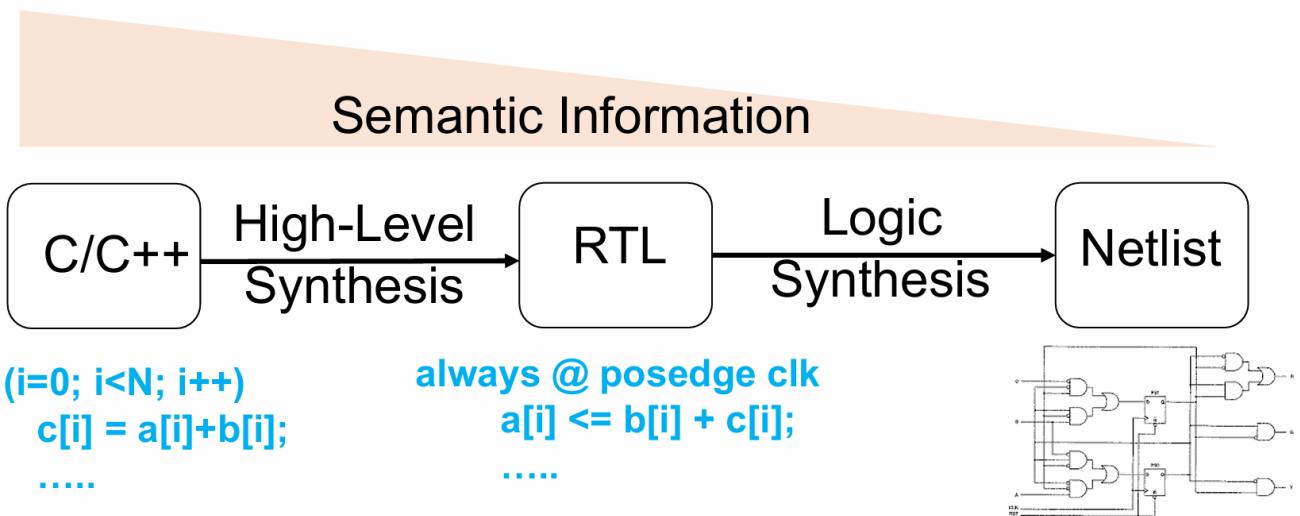
The reasons for which it is hard to implement are:

- it must be guaranteed that exists one and only one key that produces the correct results
- it must be guaranteed that all the wrong keys introduce at least one error

- what does wrong result mean? Maybe some results are “wrong” but are still acceptable (ex a single pixel in a frame of a video wrong is a negligible defect)
- all the keys must be equiprobable
- the key must be very long to avoid brute-forcing

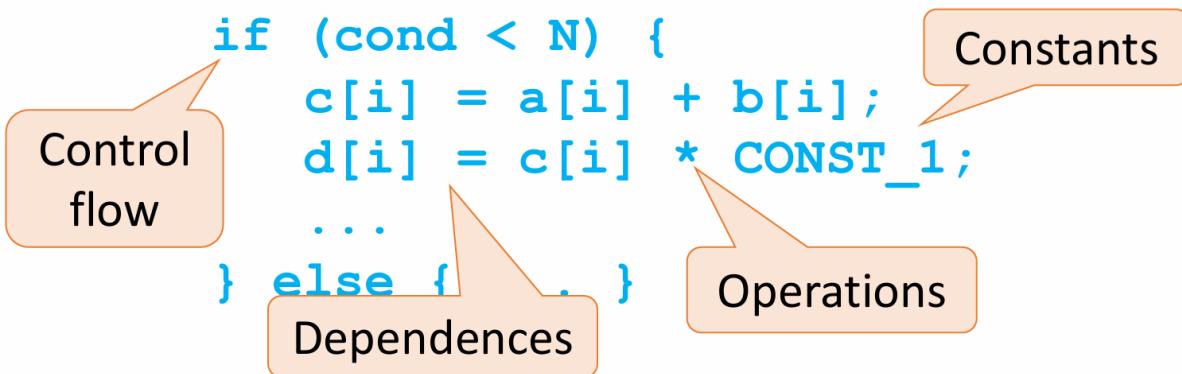
## Raising the abstraction level

- 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



### TAO: Techniques for algorithm-level obfuscation

# Technique #1: Constant obfuscation

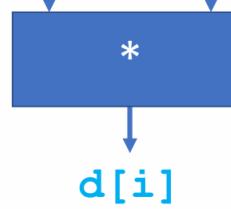
- Constants represent hard-coded values used by the algorithm (coefficients, thresholds, ...)

C/C++

$$d[i] = c[i] * CONST\_1;$$

Information is still present at RT Level

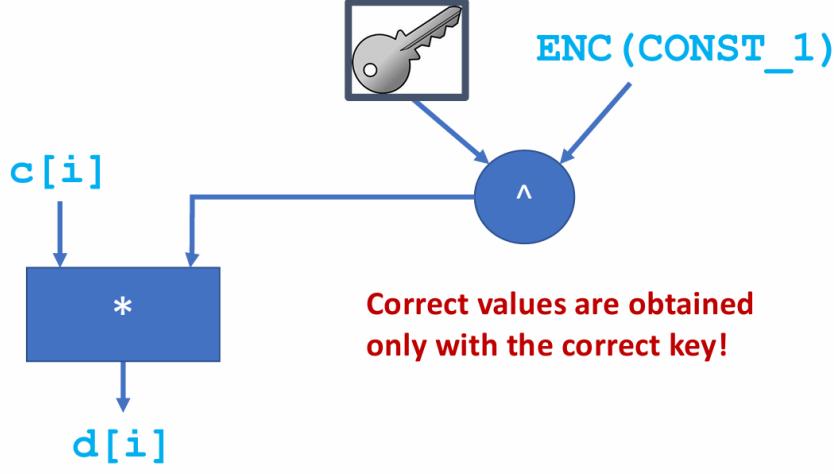
RTL

$$c[i] \text{ CONST\_1}$$


Heavily optimized by logic synthesis!

## Constant obfuscation

- A  $m$ -bit constant is **extracted and encoded** using  $m$  working key bits



The constant is encoded and it can be obtained only with a  $XOR$  applied between the encrypted constant and the key.

# 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                             |                 |                                                   |

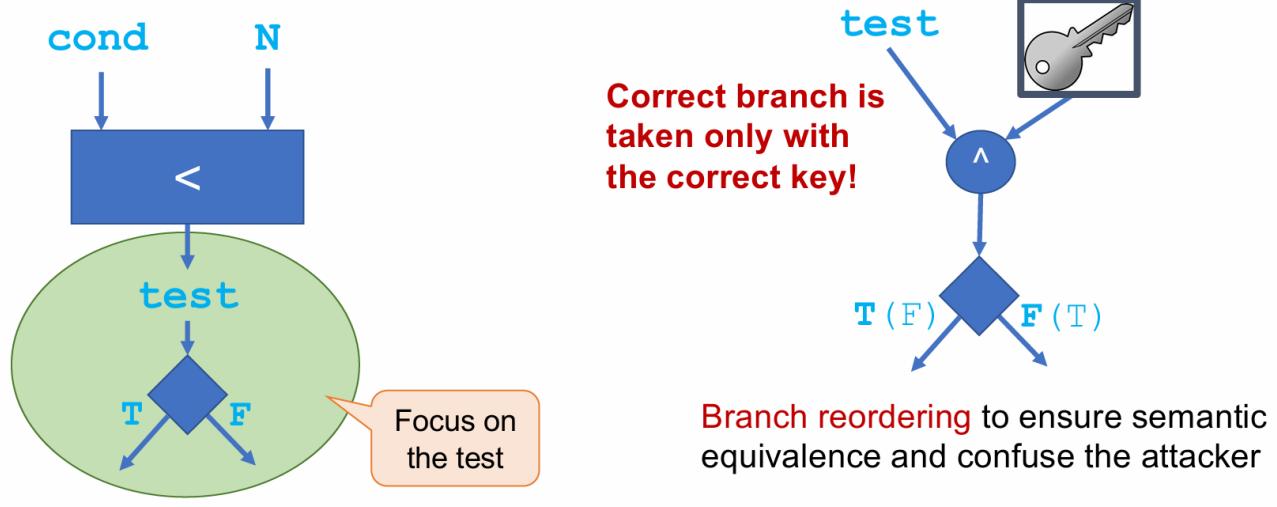
Exact information is removed from the circuit

**Less information for the attacker**

**Less logic optimizations (area overhead)**

## Control-flow obfuscation

- Masking of the **control condition** with key bit

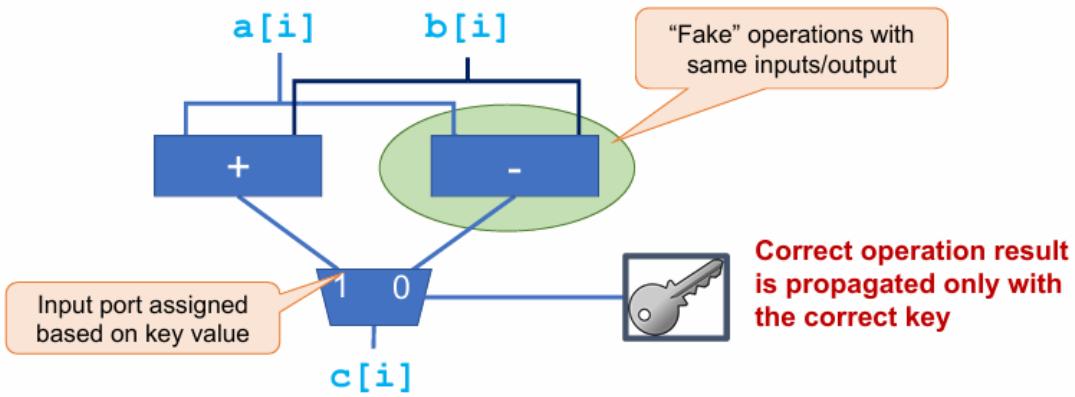


Another technique of hardware security is the obfuscation of a control branch that describes the flow of the device. It can be protected with a *XOR* with a key. Without knowing the key, the correct branch can't be taken.

ex: If the key is 0, I have a 0 xor 0 so I have 0 and I maintain the branch, while if the key is 1, I have 0 xor 1 so I'm inverting the branch. If the key bit is 0 I preserve the same value, if the key value is 1 I swap true and false.

# Operation obfuscation

- Operator variants to camouflage the correct operation



Another way to confuse a potential attacker is adding additional and fake operations, then propagate and select the outputs of the operations based only on the key.

*ex: Let's suppose that the addition is the operation that must be performed, a subtraction in parallel is added and the result is selected via a multiplexer that selects the correct result only based on the key. If the key isn't known, it is not possible to understand which is the correct operation.*

Based on how we pair the operations, the attacker can even be more confused. If the attacker has some intuitions about our algorithm works and understands that a subtraction can't be performed for any reason, the attacker understands that it is a fake operation. So as the attacker has information about our algorithm, many of these operations become more complicated to implement. So obviously the correct value is propagated only when the key is correct. Every time an operation is implemented, another operation is added but the correct result is obtained only if the key is correct.

## 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

Area overhead is *reasonable* in the sense that it is similar to the original design but obviously becomes more significant, because additional functional units and a multiplexer are added, so the area of the implementation doubles.

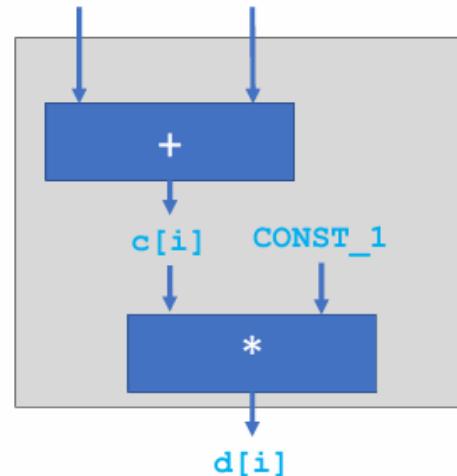
*In general how to select these possible variants is still an open question.*

## 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 { ... }
```

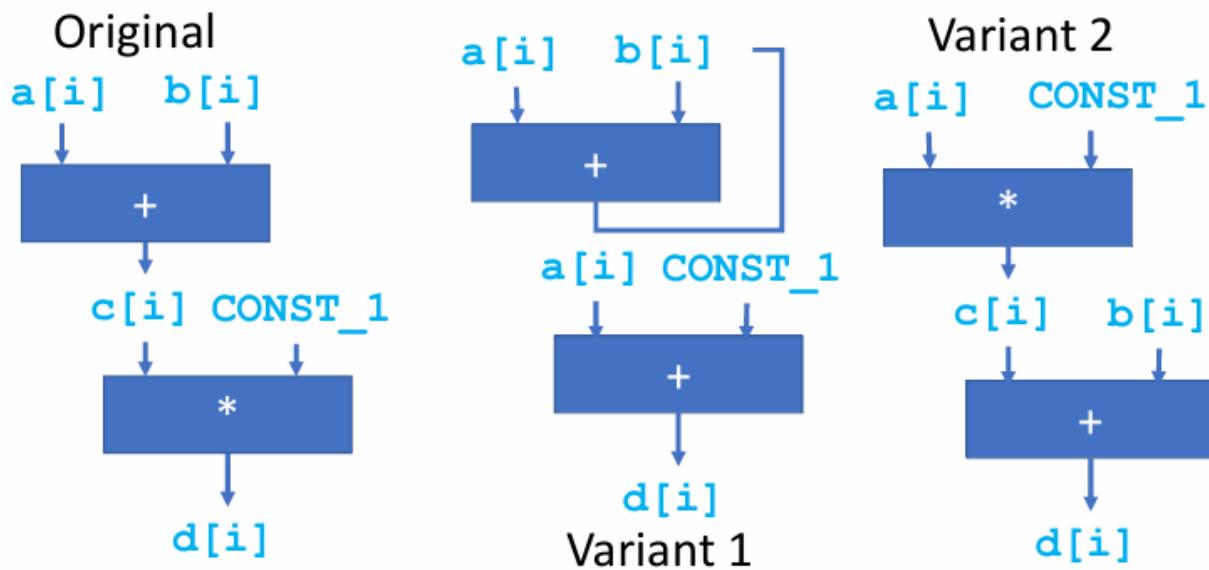
Very hard to compute in  
gate-level netlists



In the context of hardware design, dependencies become something very hard to follow. Here it is shown how it becomes very complex very quickly, lots of memory operations are added for each addition. Some HLS tools introduce optimizations for making this complication less heavy. At gate level these connections become very difficult and becomes hard to understand which values they do carry.

## Dependence obfuscation

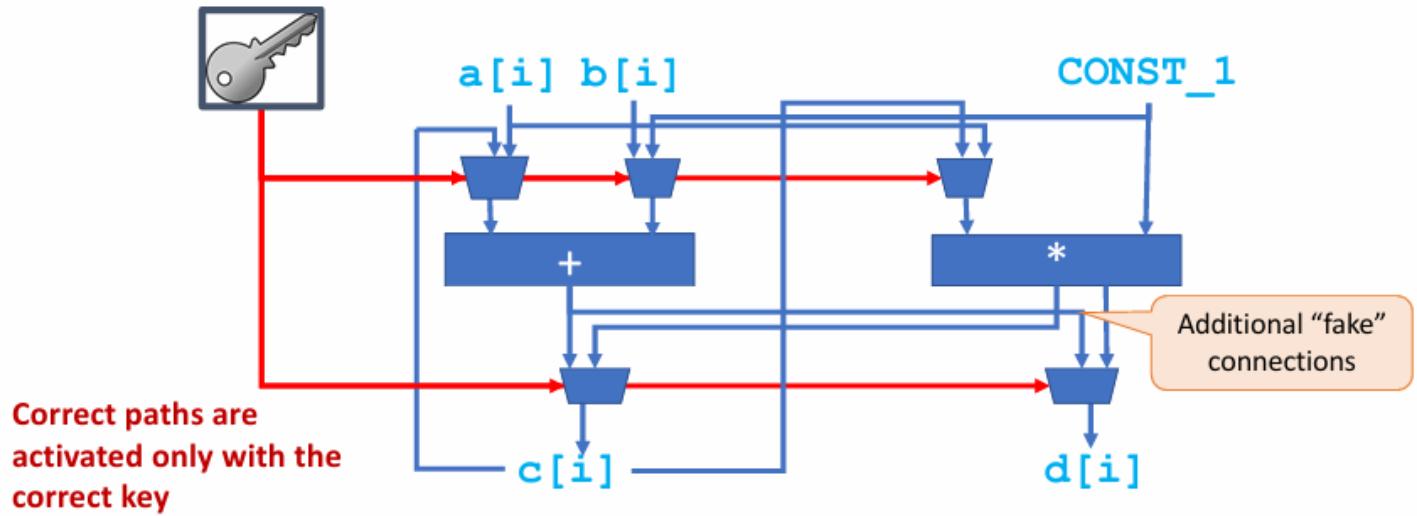
- $K$  key bits are used to select among  $2^k$  variants (including original)



One of the possibilities to confuse the dependencies is to create variants for our circuit by changing the connections, but this becomes very tricky: let's consider the *variant 1* we connected the output of a unit to the input of the same unit, we created a combinational circuit with a sort of feedback, so it can be put as a variant but still is revealing information to the attacker but it's clearly wrong, so the attacker understands that it is wrong, while in the *variant 2* we have a restructure of the design, this means that I have the operations that are the same but now instead of a plus I have connected the first input to c that is then connected to the output of the multiplexer, so I changed the order of the operations, I changed the inputs by changing the connections between signals. Given  $k$  bits of key I can introduce  $2^k$  variants and then merge them all together.

## Merge and Selection of DFG Variants

- Creation and merge of **DFG variants**



here we see that we did not add more functional units but just multiplexers.

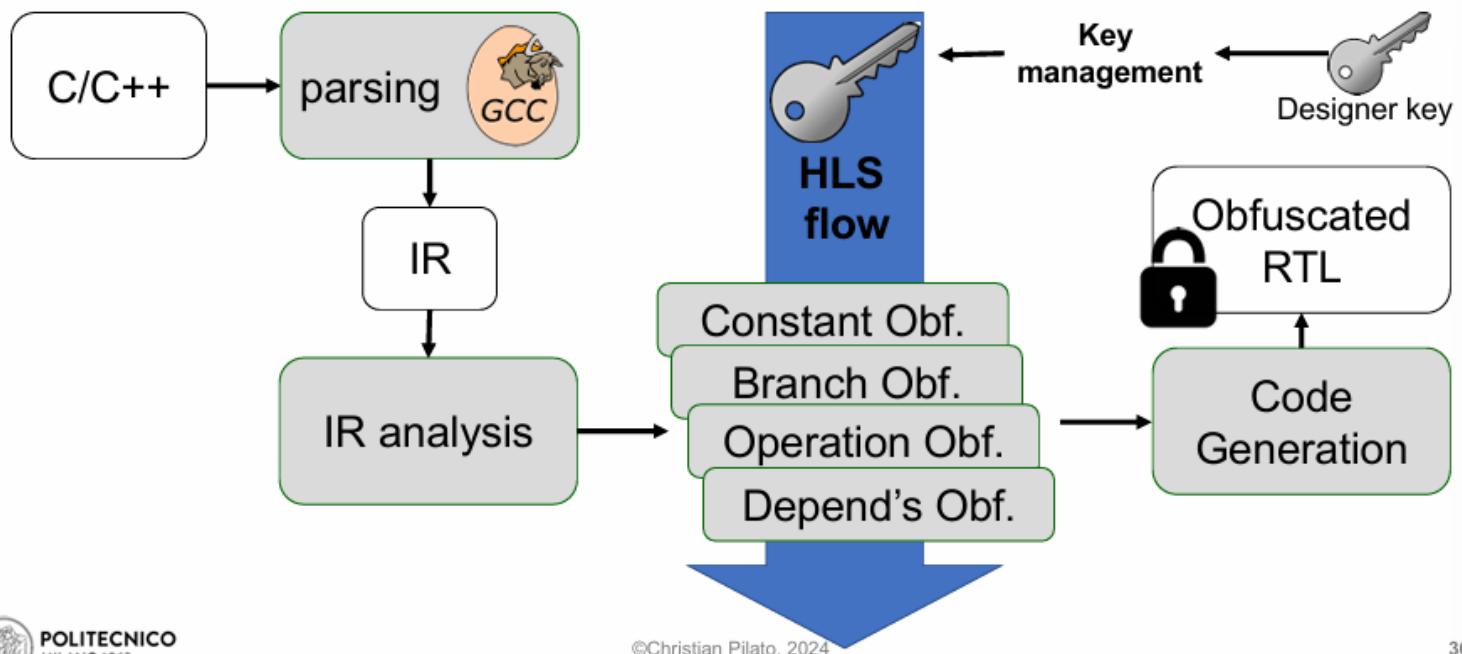
## Analysis of the Technique

- 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)

It can change the operations and semantically different variants, because how and which values are combined changes. This can change the latency of the component so it can take more time in the path and can add or reduce the number of cycles. Furthermore, a significant area overhead is added, because every time a key bit is used, an *exponential* number of bits is added. This is very hard to implement from the RTL, so the only way to create it is just using the HLS, not while tweaking the RTL. Easy to implement in HLS tools, so the key is provided and how to manage the key at runtime is implemented.

## TAO in Bambu



Let's see some results for these specific benchmarks

## 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

| 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          |

Each constant is converted into a 32-bit signal

4 bits are used for each BB (16 variants)

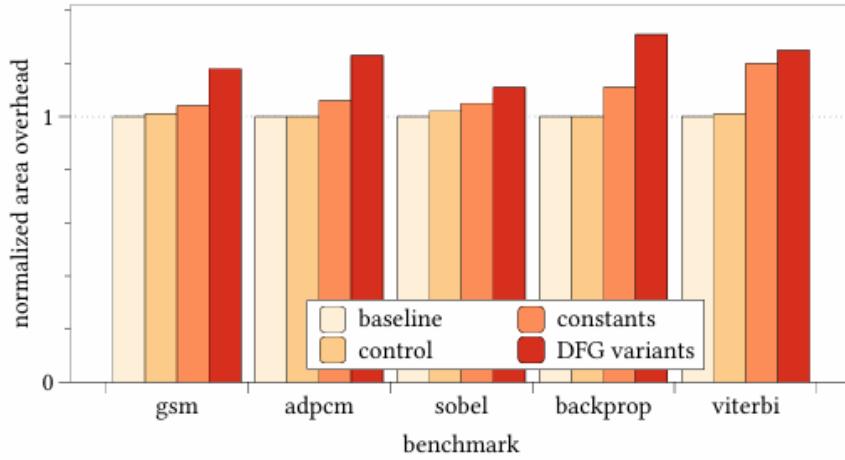
We can see that the design has a big number of total key bits even if the number of obfuscated branches is pretty low, generally we don't have many branches that can be obfuscated, we have just a *XOR* function in some specific branches of the design then based on the number of basic blocks and variants for each of those we might have a significant number of key bits, in this case we have four key bits for each basic block, so we have  $2^4$  variants, given the number of basic blocks we have a certain number of key bits. In half of the cases, the number of key bits related to the constants is dominating the design, so we can already understand that the constant and operation obfuscation is dominating the overhead

1. cannot do optimizations
2. we introduce fake operations and multiplexers

When we have constant obfuscations we always obfuscate the whole constant, so if we have a 32 bit constant we may even decide to obfuscate only half of that, because we might obfuscate the LSBs or we might obfuscate the MSBs.

# 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?**

The conclusion is that the obfuscation is a powerful technique, but we must always consider how the defense can be attacked, we cannot say “the attacker won’t be able to find it”.

There are two ways to manage the key:

1. folding the key, so reuse it many times, we fold the key on the bits that we have
2. encryption

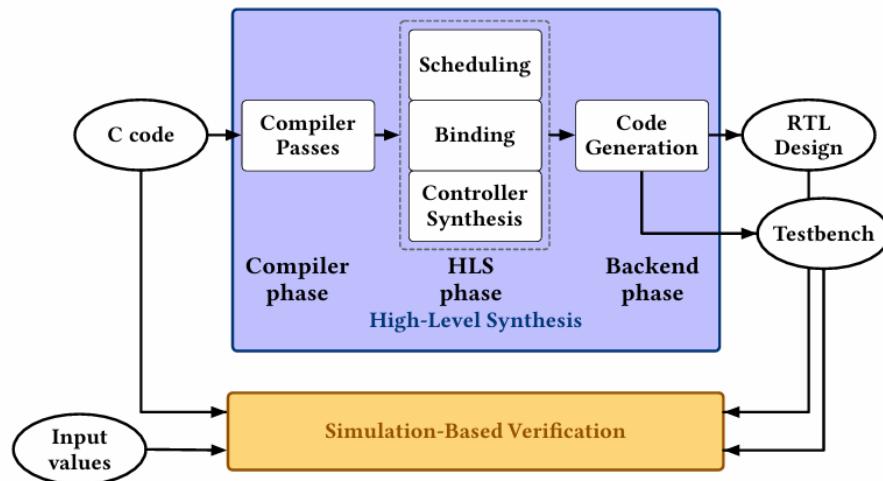
note: how can we validate the results?

One powerful metric is the *output corruptability*: (def. in the slide). Given the hamming distance we can determine how many bits are flipped in the output for each of the one of the key. It gives information about the

distribution of the differences and the ideal case is that the probability distribution of the probability of flipping each bit when given a wrong key is exactly 0.5. By changing the key I can obtain the correct or the incorrect value. I can only flip the coin to decide what the correct key is, I have to guess the entire key. It would be very difficult to achieve 50%, but the closer we are the stronger the defender is.

Obviously in very large designs I can't obfuscate the whole circuit, because as we can see from the results, by obfuscating some operations I have almost a 30% overhead. So I can select parts, but which parts do I select? The ones that affect outputs? The ones that affect the inputs, so that the values coming in are not credible, but we would need to be sure that the parts that we obfuscate we see the result on the outputs. Another way could be a random choice, but that still must be done in a way that is effective, in the sense that the measure of the level of obfuscation so also the level of security. Corruptibility is a metric giving information about the effects on the outputs, but we can have a very obfuscated design, very hard to understand but with minimal effects on the outputs, and vice versa.

## Simulation-Based Verification



If modified design can escape simulation-based verification,  
it becomes the golden model

The next problem is a consequence of this point, so once we have HLS we have a semantic difference between input (that is a software) and an output (that is hardware). So the way to verify the correctness of the design is simulation, so we test a certain set of inputs, we verify the outputs and we verify that are matching the golden value. The problem in this case is that simulation based verification is not a “real” verification in the sense that a design passing the simulation based verification doesn't mean that it does not have errors for the values that we provided, unless we provided an exhaustive set of inputs but that would mean that our design is very small.

If we find an error we know that the design is wrong, but we can't know that the whole design is correct.

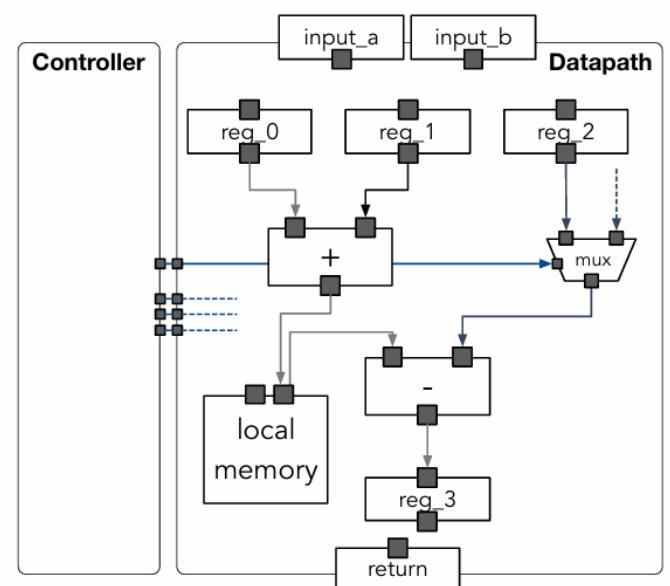
This can be used for several effects, one is positive that is implement watermarking. The difference between locking and watermarking is that locking is an active method, this means that basically we are actively creating an obstacle to slow or block the attacker, while watermarking is a passive method, we don't create an obstacle to the attacker but we are providing a way to certify if the design is the original one or a copy, this is usually used in lawsuits when we find an infringement and we verify if the chip used is from our design, so a unique signature is created and created inside the design and when we have the chip we have a way to easily extract the signature and we verify if it is coming from us. Obviously this should be hard to remove, if it is a simple signature it can be removed and is unuseful.

To make this signature there are many solutions, like embedding special keys to use the concept of hardware trojan in a good way.

# Benevolent Hardware Trojans

The structure of a **Hardware Trojans** can be used for watermarking

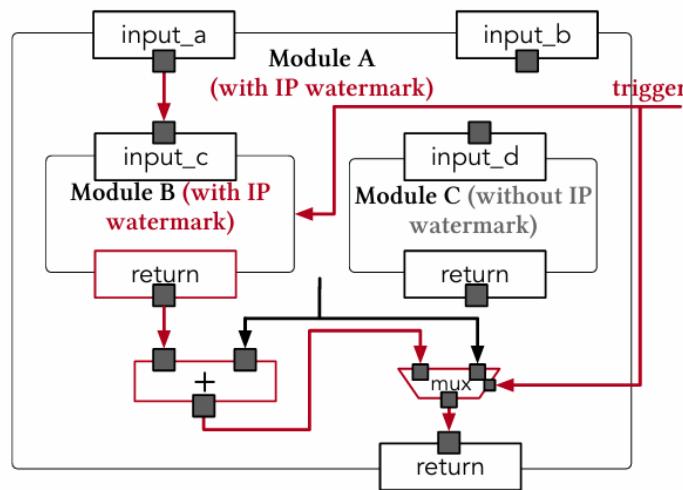
- Stealthy during normal execution
- Activated during rare condition (similar to a **trigger**)
- Small and power function intertwined with the functionality (**payload**)



Let's imagine it in the benevolent way: in this case the secret that we want to be revealed is the signature that we want to be verified. The fact that I have conditions means that during normal operation the trigger isn't active and isn't giving information, while during the litigation it is activated and the behavior demonstrates during the litigation that the chip is coming from my design.

## Architecture Modification with Watermarks

Only one function contains the watermarks but the others must propagate the values



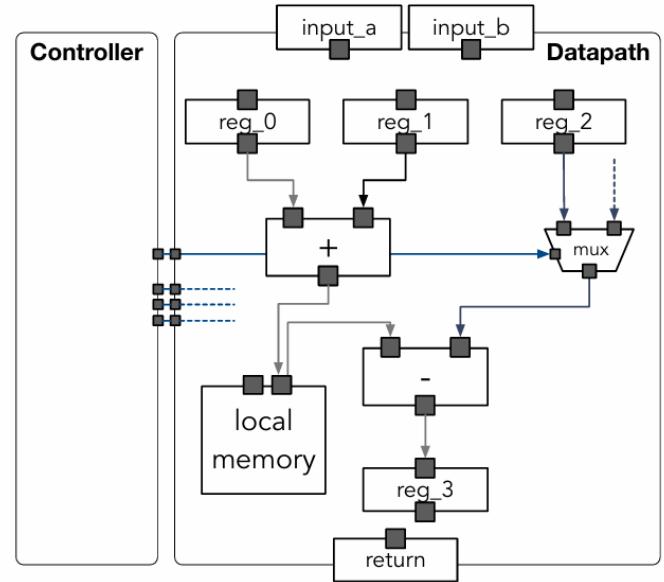
so we create an extra functionality, reusing the unit by creating extra connections with extra wires and extra multiplexers to recombine the different outputs.

One solution is to decide a function to obfuscate and then we decide a list of operations that we want to use to create the signature and the ones that we want to remove. This is tricky because if my probability of removing the operation is low, the list of operation for the watermark would be similar to the original functionality, so this means that the key is weak, because is similar to a value that I can always obtain, while if I have a very high probability of removing operations, my list of operation that I keep for the watermark is very small. The result is that the signature is weak again because the functionality is very simple, so the signature is very simple. We have to find a good balance of probability of keeping of removing.

## 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**)



©Christian Pilato, 2024

42

Then we put everything together, we have the datapath, the functional controller because is the controller of the functionality and what we can do, by putting in the middle a payload controller, so once I have my inputs I verify if I am in the normal operation or if I activate the trigger. Since these have the same structure they can also be merged and an unique FSM can be implemented for functional and payload controller.

If the work for the malevolent hardware trojan is hard, here it is much easier. For a malevolent trojan I have to find which rare conditions I should use to activate my hardware trojan without being detected. Here since I am designing the functionality I know the distribution of the inputs and which inputs are never used or what combination I can use to activate the trigger, so also the combination to activate the trigger is an information. This can be implemented in many ways, so the payload controller can be added into the FPGA or in a chip or in another way.

FPGAs are reconfigurable, why are they interesting for hardware security? The idea in hardware security is that when designing a component we do not want to reveal the functionality and the FPGA are perfect for that, because when you fabricate them we don't know the functionality that will be implemented, so we can implement any functionality, so the idea of implementing a functionality only by loading the bitstream. All the information can be added into the flow, a trojan can be added in the code and we can try in test mode if it works. The overhead can be high, especially in small designs, even 20%-25%, but for complex design maybe we have 2%-3% that is acceptable, this only for the functionality, then we have the connections etc. usually we do not add many FFs, what we do when we add an extra controller we may need more states in the state machine we might need more bits to encode the state machine and so more FFs. In this case, the idea of

hardware trojans can be reused in a good way to create watermarks and we can have different ways to activate it, test mode or explicit trigger based on the values etc.

We saw few solutions for each part, now we have to try to see if it is possible to analyze and use HLSs as an attacker, to create attacks in the design.

# 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

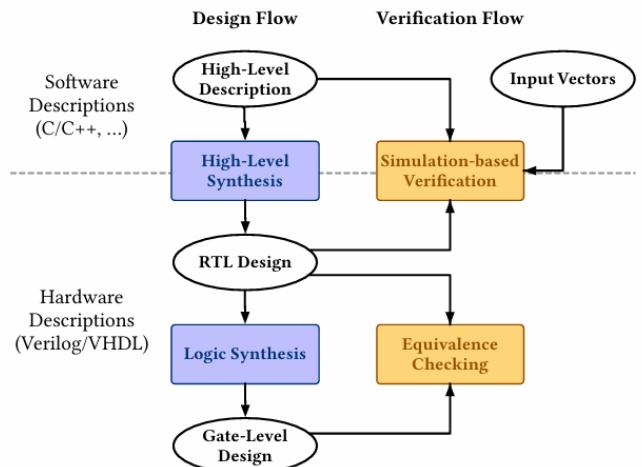
Design houses (or competitors) may have interest to degrade IPs after a certain amount of time

- 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**



Planned obsolescence is a way in which I want to degrade a component after a certain amount of time, is a bad practice that the major players in the industry have been applying, especially in software but also in the hardware. Our company might want to force people to buy a new one after a certain amount of time or we can do that to a competitor, to undermine the reputation of the other products. This is problematic because all of the reasons to change one device after a certain amount of time are not functionally related, they're base on

performance or in terms of battery consumption, this means that if I still apply simulation based verification the chip is correct, but if I introduce something like planned obsolescence, in HLS, with simulation based verification it is not able to catch it, then I'll have an RTL design that embeds these modifications and will pass also all the equivalent checking, so since this is my reference design also the other copies will have it.

# 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!

So in general the problem is that we cannot trust tools, also the behavior must be verified against bugs and malicious intents, there's an interesting direction towards open source tools, so that anyone can contribute with new functionalities etc, so when designing we have to trust all the tools that are used.

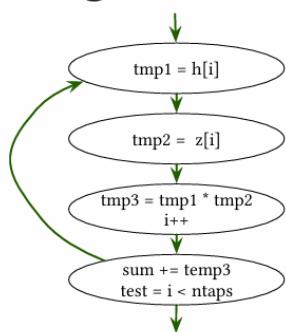
*Black-hat HLS*: HLS that can act maliciously and also all the steps can be compromised. Looking at the HLS, let's start with degradation attacks.

## Attack #1: Degradation Attack

It aims at **degrading the performance** of the IP core after a pre-defined amount of time (number of executions)

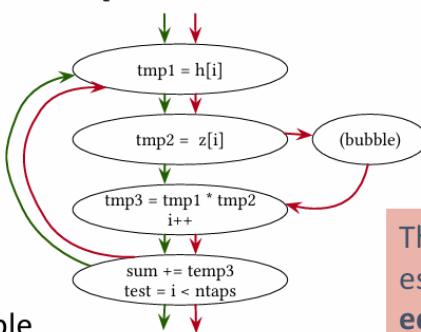
- **Bubble states** inserted in specific points of the FSM to maximize impact

### Original FSM



Trigger is a simple execution counter

### Compromised FSM

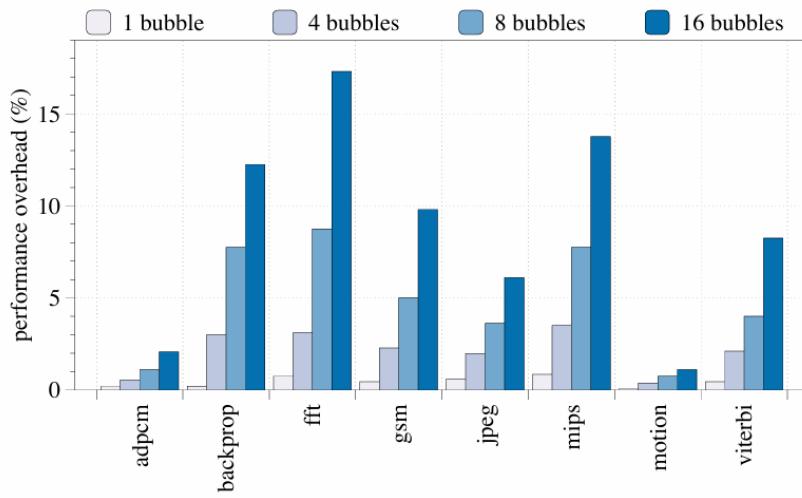


This attack can easily escape sequential equivalence checking

Idea that we want to undermine the performance of the design after a certain amount of time, this is pretty easy to be done because after a certain amount of time, like a counter, then we move and we change the controller in where we have bubble states, empty states. The effect is slowing down the computations, and there are specific points where if we add a bubble, the effect is multiplied by the number of operations. The other problem is that even if we have the original FSM is hard to detect it by sequential equivalence checking, because it is checking the results and the evolution of the values over time, to do so one of the key ideas of SEC is to remove the concept of time from the analysis, so we're intrinsically removing the bubbles and we are always producing the correct result.

## Degradation Attack in Bambu

We added a **malicious pass** after the scheduling to insert a **configurable number of bubbles**

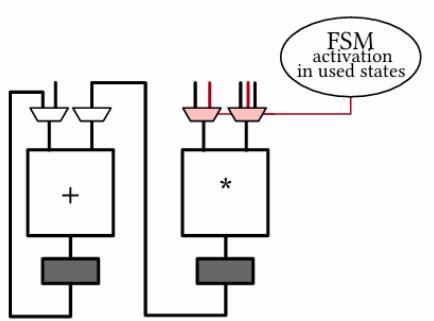


If we are very careful, we can add up to 16 bubbles, with no overhead in area the performance degradation is 20%.

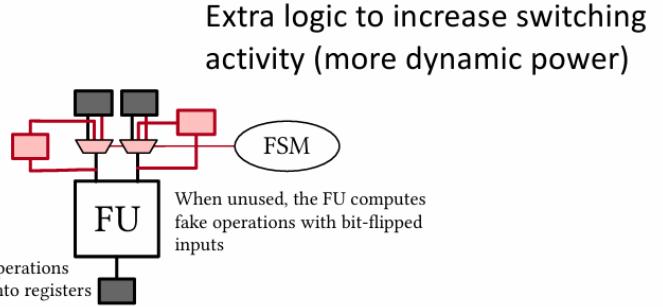
# Attack #2: Battery Exhaustion Attack

**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



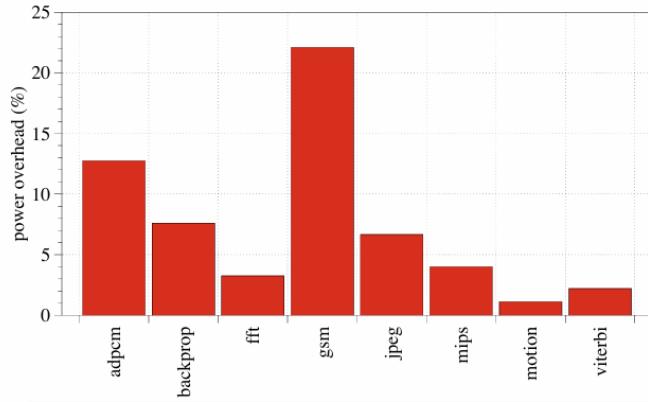
This is no golden model  
before HLS for power analysis

One of the most important reasons to change a mobile device is the battery exhaustion. So if we can accelerate battery discharge we can implement battery degradation. The HLS tool knows which units are we using at any cycle and which we are not using at any cycle. So we can imagine a way to drain extra current. Here we have a multiplier and a modified multiplexer with malicious modification, then some logic is created to basically say that if the unit is not used for some consecutive cycles you keep the value that is inside the register and every cycle we flip every bit, so the MUX sees the inputs changing and the different values are propagated, so the switching activity increases and so the power consumption. Then what happens is that the value is put in the input of the register but it is not changed, so I'm using a lot of current without modifying the functionality.

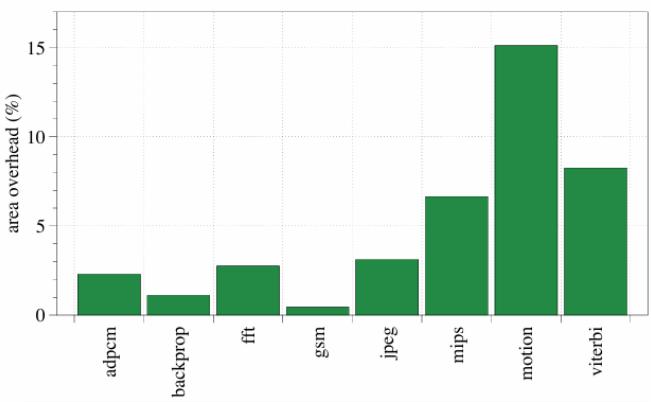
## Battery Exhaustion Attack in Bambu

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

Another malicious way is Key Recovery, so creating a side channel during HLS

# Key Recovery with Reduced-Round AES

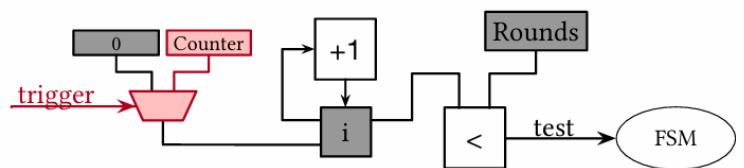
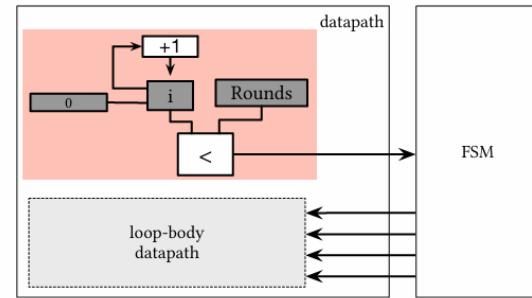
Many cryptographic algorithms execute multiple «rounds»

- 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



Most of encryption algorithms are based on rounds, so they repeat the operation a certain number of rounds, for encryption and for decryption. This is a classic way that is implemented in HLS, with the FSM, then logic and the loop. The problem is that the key can easily be recovered if the number of rounds is reduced so, if we encrypt a certain information with 7 rounds, we still obtain a text that looks encrypted, but it is not strong enough because by operating on this we can find it. We see that to insert this modifications in the circuit, to execute sometimes 7 rounds or 18 rounds, we can change the value of the counter, we can change the start value or we can change the end value. In this case there's a collision between tool developer and the ip, in this case we are adding modifications internally at the design so that it can be extracted later.

## 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?

For hardware security, every time we have a clear metric we can do any optimization that we want.

# 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

If our design is smaller than the previous one, it is a better design.

# 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

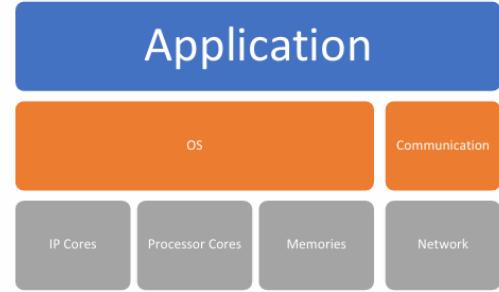
The goal is to make the life  
(exponentially) harder for attackers

... 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

©Christian Pilato, 2024

61

If we had clear metrics we could introduce automatic tools to evaluate and implement security implementations, but to certify that the data is secure and no one can access the design, we would have a security certified design, but this is impossible, it is difficult to prevent attacks.

It is already difficult to prevent attacks that we know, imagine what for the one that we don't know.

The objective is usually not to make it completely impossible to make the attack but sometimes is enough making so hard that it is not possible for the attacker to complete the attack.

The vertical and the orizontal integration is important.