Design of Hardware Accelerators

This document is a collection of notes from the "Design of Hardware Acceleratos" course at Politecnico di Milano given by Prof. Christian Pilato, Academic Year 2021/22.
This document was written by Guido Rolle.

Please note that the order of some chapters may be altered w.r.t the order of the lectures.

Introduction to Heterogenous Systems

Heterogeneous systems are systems made by different types of processing elements.

Since late '90s, the number of cores on chips, machines, and architectures has been increasing, and from the mid 2000s they have been specialized: even chips designed for specialized applications have to tackle a range of diverse tasks - e.g. autonomous driving deals not only with machine learning, but also with videostreams of the environment; these systems can be made both faster and more power efficient by using heterogenous hardware to accelerate their tasks, provided the designer is able to distribute the workloads and coordinate the components effectively.

Energy & Power Efficiency

- **Moore's law**: "The number of transistors on consumer grade CPUs doubles every two years"
- **Dennard scaling**: "If the transistor density doubles, the circuit becomes 40% faster and the total power consumption remains the same"
- **Koomey's Law**: "For a fixed computing load, the amout of energy needed halves every 2 and ½ years"

Moore's law mainly refers to the halving in the size of the transistors; but as transitors become smaller and smaller, further reducing their size has become has become harder and harder.
To keep up with performance demands - and sometimes as publicity stunts - sometimes companies may double the number of transistor, without a corresponding decrease in size: in turn, the energy/power efficiency does not decrease according to Dennard scaling and Koomey's law; in fact Koomey's law hasn't been met since the mid-2000s.

Energy and power efficiency requirements are very relevant: devices must not consume too much of energy and must not overheat.

Additionally, when keeping the power constant, as the transistors shrink a bigger percentage of the area of the chip must be powered down - i.e. **dark silicon**; if not all components in a chip can be active at all times, adding more general purpose processors won't result in better performance: a solution is to have specialized hardware that does not run constantly, but is turned on only when tackling some workloads, though this requires a scalable infrastracture and an efficient power management system.

Hardware acellerators can be placed on the chip or on an external device;nowadays, in commercial SoC, more than 50% of the surface area is dedicated to **out-of-core accelerators**.

Systems-on-Chip (SoCs)

Systems-on-Chip (**SoCs**) are architectures where components are connected at the same physical layer. SoCs are small, power efficient, and reliable, but they have extremely high design costs, and component-testing/prototyping may be difficult.

In general, an SoC is made of the following components:

- **General-Purpose Processors (GPP)**
- **Memory Elements**
- **Interconnection Infrastructure**
- **Pre-defined IP Blocks**

Platform-Based Design

The design of an SoC, often starts from a predefined architecture called **platform template**; on the template a number of components and interconnections is already present.

The designer of the accelerator has to interface with the pre-existing components, in the limited space available on the chip: once all the components have been placed and the designed is finalized, the SoC platform becomes an Soc Architecture through the use of an EDA Integrator.

A first degree of customization is represented by what can be added to the platform: soldering *hardware IPs* (*soft* or *hard*), programming *FPGA logic*, or writing *embedded software*.

Another degree of customization is represented by the level of flexibility of the added component: *programmable cores* (e.g. GPU or specialized CPU), *configurable cores*, or *dedicated cores*; greater flexibility usually equates to some degree of loss of performance.

Not only the SoC platforms, but the accelerators' designs are themselves reused: the percentage of reused components rises with the number of IP blocks - e.g. whenever a core is replicated, its accelerators, memory controllers, etc. are also reused.

Adding an accelerator to a system requires an **integration process** - e.g. any accelerator is a possible source of data and potentially requires adding bandwidth on the databus.

Adding an accelerator has its monetary cost: either for designing the accelerator itself, paying its license or designing the integration; this means that accelerators must be used efficiently.

SoC Components

General-Purpose Processor (GPP)

General-Purporse Processors (**GPPs**) are flexible; to execute a new functionality, there is no need to design/rewire the hardware: in a GPP, a new tasks is executed by loading the instructions into the control units, and the data into the datapath accordingly.

GPPs are almost always present: functionalities interacting with the storage and the file-system are bottlenecked by I/O latency, and accelerating them in hardware wouldn't yield benefits; the GPP prepares the data for the hardware accelerator, either by writing it to a scratch-pad memory or loading it from storage onto the main memory.

On the contrary, accelerators cannot load different programs as their functionalities are fixed and built in hardware: they only load data; usually custom accelerators are used to execute critical kernels that are common to more programs.

An issue is to determine the **specialization** of our accelerators, that is decide the granularity of the kernels to implement in hardware. E.g. we have an application that requires voice recognition: we can design a small accelerator only for matrix multiplications, leaving the rest to the GPP, or we can design an accelerator for the entire voice recognition function; naturally, the smaller accelerator is more general and more reusable for purposes outside voice recognition, while the larger one will be more efficient.

Software/Hardware Stack & Instruction Set Architecture (ISA)

1. Problem
2. Algorithm
3. Program (in high level programming language)
4. Runtime System/OS
5. (Istruction Set) Architecture
6. Microarchitecture
7. Logic
8. Circuit
9. Electrons

The **Instruction Set Architecture** (**ISA**) is the interface between software (1 through 4), and hardware (6 through 9): it provides a way to tell the hardware what should be done.
The microarchitecture refers to the underlying implementation through which the compiler executes the instructions of the ISA: two processors might share the same ISA, but provide very different implementations as long as the semantic of ISA specification is respected (e.g. out-of-order execution).

CISC vs RISC

Complex Instruction Set Computers expose ISAs with > 1000 instruction, of variable lenghts (from 1 to 15 bytes); CISCs offer different **addressing modes**: operands for an instruction may come from different locations - e.g. registers, stack, etc... E.g. Intel x86

Reduce Instruction Set Computers expose ISAs with around 200 instructions, usually of the same length; the architectures are *load/store*, so the operands must always be loaded from the stack to the registers first. E.g. MIPS, ARM

Tightly-Coupled Accelerators & Loosely-Couple Accelerators

On an SoC the accelerators may be:

Tightly-Coupled Accelerators (**TCA**) are accelerators integrated inside the CPU, either optimizing pre-existing instructions or adding an ISA-extension.
Being tighly-coupled, TCA share key resources with the CPU - i.e. register file, MMU, L1 cache; the size of a TCA must be limited and its execution time must be in line with the other Functional Units of the CPU (otherwise it will stall the entire CPU); the amount of data used must also be limited to the registers.

It is ideal for simple operations that are frequently used.
Examples are: MAC operations, encryption operations.

In terms of programmability, adding an extension to the ISA is not always possible as the most used ISAs are fixed and proprietary, which means extension cannot be freely added; RISC-V is an open-source ISA, that allows anyone to add a custom ISA.

Loosely-Couple Accelerators (**LCA**) are accelerators outside found the CPU and controlled through drivers; they access the data can be passed to them by the CPU through scratch-pad memory or they can fetch it from main memory through DMA controllers; The CPU can offload the data and computation to the accelerator and run in parallel; these accelerator are more efficient for large data sets.

In terms of programmability, the software interface must offer functions to copy data into and get data out of the accelerators; additionally, synchronization mechanism are also need. All of these functions must a corresponding hardware to support them.
The designer must decide on the size of the PLM, the number of ports and synchronization mechanisms.

Storage Elements

Storage elements can be either:

- **On-Chip**, temporary storage for direct and fast access
    - **Cache**, non-programmable/transparent storage between memory and another component; caches reduce the average access time, but also make it unpredictable.
    - **Scratchpad**, programmable memory through software directives; data has to be manually loaded, but access times are predictable (no misses).
    - **Private Local Memory**, memory local to the component and not visible to the system.
- **Off-Chip**, larger memories accessed through memory controllers
    - **Main Memory**
    - **Disk**

Memory Design in LCA

The design of memory in an accelerator has huge impact on performance:
firstly, memory represents 80% to 90% of the accelerator's total area: any memory increase result in a significant increase in the total area of the accelerator; secondly, since nowadays the size of the problems are many order of magnitudes greater than available on-chip memory, data transfers are constant and usually represent the bottleneck of our applications.

An important decision regarding the technology is between:

- **Distributed Registers** (e.g. flip-flops)
- **Memory IP blocks** (i.e. SRAM)

For the same storage capacity, SRAM is around a quarter of the size of distributed registers, but usually offers only 2 reading ports and 1 wiriting port - meaning data accesses must be sequentialized and parallel opportunities will be lost; more ports can be implemented using multiple banks, but then the data must be partitioned and

accessed accordingly.
Ultimately the choice must consider how much data is accessed per clock cycle.

CPU and Memory Hierarchy

An **ideal memory** would offer infite capactiy, no latency, and infinite bandwidth at zero cost; in reality, these requirements are conflicting: for a fixed cost, the faster the memory, the smaller the capacity.
It is possible to approximate an ideal memory exploiting the principle of locality and using a well-engineered **Memory Hierarchy** - i.e. multiple levels of storage - like registers, on-chip caches, off-chip caches, main memory, storage disks and remote storage.
The same principle can be applied to accelerators.

In the presence of multiple processing elements, a memory hierarchy introduces the problem of coherence.
When the execution is offloaded to the accelerator while the CPU is on hold, then data transmission at the beginning and end of the computation is sufficient. When the execution is parallel, more complex synchronization/coherence mechanisms are required.

Some accelerators have **Direct Memory Access** (**DMA**) - i.e. indipendent access to main memory -, loading from main memory input data to the PLM, and then storing to it output data.
Other accelerators utilize scratchpad memory, where is the CPU that loads and stores the necessary data.

Interconnection System & Communication Synthesis

Whenever on an SoC resides more than one component - whether they are two GPP cores, a GPP and an a coprocessor or even two coprocessor - there arises the need for an **Communication Infrastructure** - or **Interconnection System**: a communication medium used to manage on-chip resources and exchange massive data among cores; ideally it will provide low latency and low energy dissipation.

The communication infrastructures costs around 50% of the chip's power consuption, and it's still increases due to the increasing intereconnection between the chip's component.
Latency is largely determined by the physical distance between components on the SoC and the connection bitwidth; the designer should pay special attention to latency of the communication as it represents a possible bottleneck for the components throughput, in which case a lower component frequency may save power.

In communication we defined the following terms:

- Master (or Initiator): the component who initiates the data transfer
- Slave (or Target): the component who responds to the data request
- Arbiter: the component that controls the access to the shared bus, and grants the initiator's request
- Decoder: the componenent which the trasnfer is intended for
- Bridge: an element that connects two buses, acting as a slave on one side and a master on the other

In general, the communication can follow **shared address** or **message passing** paradigm.
In the **shared address** paradigm, there exist some physical memory resource that multiple components can access to perform implicit communication; the access must be coordinated and authorized.

In the **message passing** paradigm, the messages are sent explicitly between the components; appropriate protocols and interfaces are required.

The communication can be physically implemented through **point-to-point connections**, a **bus-based interconnection** or a **network-on-chip**.

Shared Address

In the **shared address** paradigm, there exist some physical memory resource that multiple components can access to perform implicit communication.

According to the architecture, the memory can be a **Uniform Access Memory (UMA)**, where access times does not depend on the processor that perform the access, or a **Non-Uniform Access Memory (NUMA)**, where some processors can access the memory faster than others.
NUMAs are more scalable because they don't need to guarantee the same access times to all processors.

Message Passing

Point-to-Point (P2P)

A **Point-to-Point (P2P)** is a solution made of fixed wires; it is highly specialized, but it is not reusable.
It is particularly useful at a component level: it maximizes performance inside the single accelerator, and changes to the network are automatized.

Bus-Based Interconnection

A **Bus-Based Interconnection** is simple solution to communication, with low area occupation: the bus itself is a collection of wires (unidirectional, for control, or bidirectional, for data) and if the number of components is limited, the arbiter - i.e. the component who grants access to others - is also relatively simple.
Interfacing with the bus is also simple: the bus protocols are described thorugh timing diagrams, and there are standard interfaces -e.g. the AXI interface - that can be synthetized automatically.

More complex solutions are variantions can be used, such as multi-level, hierarchical or segmented busses.
Ultimately, it's not highly scalable as the bus represent a bottleneck.

It's typically used with UMA: the memory is a separate component on the bus.

Network-on-Chip (NoC)

A **Network-on-Chip (NoC)** is most scalable and extensible solution; it's more complex and more expensive in terms of area: additional components that act as routers are needed.
It should be used only when the communication among components is concurrent and unpredictable.

It's typically used with NUMA: each processor has a memory that can be also accessed by others; using a UMA would create a bottleneck on the network towards the only memory.

A NoC is made of:

- Tiles, the space for the computational nodes
- Switch/Routers

- Network Adapter or Network Interface, used to decouple the computation from the communication
- Links, the P2P channels that implement a network insensitive protocol

NOC TOPOLOGIES

Different **Topologies** can be created starting from the NoC elements.
The **mesh** is the most typical, but the components at the boundaries have few adjacent components; a variations on it is the **torus** or **mesh-torus**. There are many other e.g. **Ring**, **Binary Trees**, and even **Irregular Topologies**.

NOC PROTOCOLS

A NoC uses either **circuit switching** or **packet switching**; packet switching in particular requires **buffering** of the packets which can be costly - in terms of area, especially in FPGA where memory resources are costly.

A NoC implements either a **fixed routing schemes** or a **adaptive routing schemes**: in the fixed routing schemes, the path followed by communication with between 2 components is predetermined, but can lead to congestion; the adaptive solution is complex as it must implement a complex algorithm, but may solve congestion problems.

A NoC also implements communication on a stack, the **μ-stack** divided into **transport**, **network**, **data-link** and **physical** layers.

Communication Synthesis

Communication Synthesis is a **top-down** design task that, starting from an high-level communication abstraction, produces a synthetizable product; it is performed through:

- **Channel Binding**
- **Communication Refinement**
- **Interface Generation**

Channel Binding

In **Channel Binding** the abstract channels are assigned to physical resources; a design may use multiple abstract channels, but those may be physically implemented by a single one grouping togheter channels that aren't used concurrently or that are used by the same processes.
Naturally, where multiple abstract channels are mapped into the same physical channel, a multiplexer will have to be placed.

Communication Refinement

Once the number and placement of physical channel is determined, the **Communication Refinment** step determines the other details: the width of the channels - depending on the usage -, the control strategy - the arbiter that assigns controls of the bus -, and the communication protocols.

Two common communication protocols are:

- the **Strobe Protocol**, where the master simply makes a request for data on the request line and the transfer can be considered complete after some fixed interval.
- the **Handshake Protocol**, where the slave signals the master with an ack signal, when the transfer is completed.

Interface Generation

In **Interface Generation** the interfaces are generated, based on the previous 2 steps; the interfaces may be software (e.g. device drivers) or hardware.

Design Flow & Design Challenges

TODO

ASICs and FPGAs

An **Application Specific Integrated Circuit** (**ASIC**)n is a circuit designed to perform a specific function; it does not necessarily mean that the chip is not able to perform general tasks, but it is not as efficient.
Physically the circuit is a small silicon die, surrounded by a larger *packaging* area made of pins that expose the interface of the chip.

ASICs can be of different types:

- *Fixed-Function* ASICs
    - *Full-Custom* ASICs, in which the designer has the possibility to place every single transistor
    - *Standard-Cell-Based* ASICs, in which the designer creates the chip by combining predefined cells
    - ...
- *Reprogrammable* ASICs
    - *Field-Programmable Gate Array* or **FPGA**, here the functionality of the chip is its programmability.
    - ...

As usual the tradeoff is between speed/cost and flexibility.

Full-Custom ASICs

Full-Custom designs offer the highest performances and the lowest part cost (smallest die size, smallest number of transistors); the software equivalent would be writing code using assembly: the product can be very optimized both in speed and size, but is time-consuming and error-prone.

Standard-Cell-Based ASICs

Standard-Cell-Based ASICs are made from on a set of full-custom macros that implement very well optimized small functions (standard cells); those designs are then repeatetly reused.
Physically, these chips are organized in a stardard-cell area and a fixed-block area.

ASIC Cell Libraries are a collection of cells implementing basic logic functions; libraries can be provided by the ASIC vendor, who will then complete the design when the designers hands-off the netlist, by a library vendor, which allows designers to own the mask for the part, or be custom-made.

The standard cells have the same height so that the can be placed in predefined rows in the stardard-cell area.

Design Flow of ASICs

Starting from HDL code, the designer obtains a netlist of the system which is then partitioned in macro-blocks; then the designer decides the layout of the blocks on the chip, and then create the blocks by placing the cells and interconnecting them through

routing.
Finally, the designer must check that the design still works when accounting for the added delays/loads of the wiring.

Gate-Array-Based ASICs (FPGAs)

Field-Programmable Gate Array (**FPGA**) are integrated circuits that can be configured by the user to emulate any digital circuit (if there are enough resources avaiable on the FPGA). Programmability is achieved through a core of **configurable logic blocks (CLBs)**, that can implement combinational and sequential logic; each CLB os connected to its neightbours through **programmable interconnects (Switch Boxes)**; finally, surrounding the core are memory blocks (**BRAM**) and programmable I/O cells.

Each CLB has a **Look-Up Table (LUT)**, that is used to implement a combinational function, and a flip-flop, that is used to implement sequential logic; they are bot connected to a mux that selects the signal depending on the configuration.
The SB are similarly configured, so that the signal from a cell can be fowarded to any of the 9 adjacent cells; additionally, so CLBs may have to be used as routing cells.

Because of the regularity of its design, a large number of CLB and SB can be fit onto the chip.

Economis of ASICs

The total cost of a single ASIC chip is:
total part cost = fixed part cost (or non-recurring cost or design cost) / volume + variable part cost

In a Fixed-Function ASIC, the design cost dominates the fabrication cost, while in an FPGA, the opposite is true.

Latency-Insensitive Systems

Systems have become progressively more complex: they are made of hundreds of IP blocks, designed and owned by different companies; additionally each IP block is reused in many architectures where it must work correctly interacting with other components, often unknown at design time.

Timing violations

Ultimately the **clock period** for the entire system will be determined by the maximum latency on the components' critical paths.

Furthermore, when laying out the components on the chip, additional delay is created by the interconnection between - i.e. the length of the wire; this could lead to timing violations.
When dealing with **Deep Sub-Micron Process Technology** (sub-micron meaning < 1000nm, and deep sub-micron meaning < 100-90nm, which nowadays common use even in microcontrollers) improvements in transistors technology do not correspond to improvements in the wire technology: wire delay becomes more dominant as the transistors shrink;
Additionally, when transistor are at this scale, there is an high delay variability.

A first remedy is to add registers at the output and input of the components, **buffering** the data.

A second remedy would be to reconsider the layout of the components, as ultimately an excessively long wire is a topological problem, by placing interacting componenets as close as possible.
Ultimately, this solution is insufficient at scale.

There is a need for a design paradigm that is insensitive to delays.

Latency-Insensitive Design

The problem is the following:
Given a **Synchronous Design** composed of communicating modules, how can we make it tolerant to **Arbitrary Communication Latency**?

Naturally, the system to remain synchronous.
Additionally, the solution must be a reusable pattern.

Asynchronous vs Latency-Insensitive Systems

Asynchronous Systems require designers to think digital systems in completely different way: there is no clock, but event-based system where all the componenents communicate using the same protocol with a complex and technology dependent handshacking.

In a **Latency-Insensitive Design** (**LID**), the system remains synchronous but the components can expect an arbitrary delay (which will be a multiple of a clock period) in data exchanges.
It is the sequence of events, and not the timing that determines its correctness; this complicates somewhat the testing, as two equivalent component do not produce the same signals at the same time.
Such a Latency-Insensitive System is called a **Patient System**.

Latency-Insensitive Protocol

Protocol is simple: at each clock cycle the source sends packets with a *void* and a *data* field; additionally, the sink at each clock cycle sends back a *stop* signal.

data is naturally the payload, while *void* indicates if the current data is valid or not.
If *void* = 1, the data is not **informative** and the packet is a **stalling event**.

stop indicates that the sink is not able to process/buffer additional data; this can happen for a number of reasons: it might be busy with data coming from another component, or its latency his higher than the source's so it has to stall it.

Implementation

Practically the protocol is implemented using 2 hardware components:

Relay Stations

An arbitrary number of **Relay Stations** is added along the wire, each introducing a latency of 1 due to 2 flip-flop that buffer the *data* and *void* signals.

If the data is informative and the *stop* is high, the data is buffered into an additional flip-flop until the *stop* signal low again.
The *stop* signal is also relayed up the wire to the previous relay station, until it gets to the source's shell, that stalls the source component.

Shell

The **Shell** is a wrapper for a component that has not been designed with this latency-insensitive protocol, assuming the component is stallable; then the Shell is able to manage the *void/stop* signals in a transparent way.

Incoming data, if it's informative, is enqueued in FIFO queue; when the queue is full, the Shell sends a *stop* signal applying backpressure along the chain of relay stations back to the source.

For outgoing data, the component is stalled if a *stop* signal is high.

Latency-Insensitive Protocol for Physical Design

During the synthetisazion of the layout, if a wire introduces a latency greater than the clock than it is segmented using Relay Stations and Shells.

Latency-Insensitive Protocol for System Design

Latency-Insensitive Protocol allows for the creation of complex system where the components can be optimized indipendently.

Ultimately, the system built will be synchronous but reactive to a data availability event. These latency-insensitive primitives can be synthetized with low overhead: 3% in both ASIC and FPGA technologies.

Intro to High Level Synthesis

The standard methodology for designing an IC is **High Level Synthesis** (**HLS**): the designer starts from a description of the algorithm written in a high level software language (e.g C, C++, SystemC, Python - and principle any software language), that is automatically translated in an RTL description.

In particular, the automatic translation process is parametrized, and can be controlled by the hardware designer: this allows to produce multiple solutions exploring the **design space** and selecting the best configuration both a component and at a system level.

The designer then uses the logic synthesis allows us to go from behavioural/RTL/structural levels to the actual netlist.

Design Tradeoffs

Exploring the design space means manually looking at the different solutions and evaluating the space-time tradeoffs between them: here "*space*" are the area and the resources used in the component, and "*time*" are frequency, latency and throughput.

First the designer must discard the solutions that are Pareto-dominated by others - i.e. solutions that are strictly dominated in all aspects by other available solution. Additionally since the number of Pareto dominant solution is still high, the designer must analyze only solutions that have significant differences in the performance/cost tradeoff.

Advantages of HLS

High Level Synthesis brings plenty of advantages to companies:

- Productivity
    - Allow for easier modelling
    - Reduce the LoC by an order of magnitude
    - Generate RTL much faster than manual method
- Quality of the Results
    - Automatic parallelism extraction
    - Multi-cycle functionality
    - Loop Optimization
    - Optimazation of Memory Accesses
    - To enable design exploration of HW/SW solutions by parametrizing the translation

HLS also provides an easier entry point into hardware design, as it requires less expertise (e.g. knowing C as opposed to knowing Verilog), making it easier to find personnel.
That said, it is often said that HLS is not a tool for software designers; it is an auxillary tool for hardware designers who can then tune optimization or manually write performance critical parts of the result obtained through HLS.

HLS History

HLS technolgies became of research interest in the mid 80's.

Starting from the early 90's the first commercial solutions appeared: they translated behavioural HDL and targeted ASIC chips; these solutions were ultimately failures as programmers that knew behavioural-level HDL often knew RTL HDL, and the cost of ASIC chips made optimizations critical: thusly, the only engineers capable of using these tools could already do a more cost efficient job manually.

From the early 2000s, thanks to HLS tools capable of synthetizing from higher level languages (e.g. C, C++ and various extensions) and the introduction of FPGAs, the tools started having commercial success.

Vivado HLS is the most famous tools in the HLS family.
The tool was born from a small company of 25 employees, started from a research group; at the time it was named AutoPilot.
The company was acquired by Xilinx Inc, the larget FPGA manufacturer, and released as Vivado HLS; it was licensed for free to anyone buying an FPGA; using HLS anyone who knew C could program the FPGA itself.
Xilinx itself was later acquired by AMD.

LegUp is the most used OpenSource tool for HLS founded by Altera, and targeting Altera FPGAs.
Later Altera was acquired by Intel.

The High Level Synthetizer

The translation of high level code into an RTL description is carried out automatically by an **High Level Synthetizer**; the HLS is built on compiler's theory and technology, and as such it is split between a front-end and a back-end.

- Input:

    - **High Level Programming Language**, which will then be turned into an **Intermediate Representation**
    - **Functional Resources**, which are the resources that can be used in the synthesis, each with an area/delay so that synthetizer can make optimal choices and respect the constraints
    - **Constraints**
    - **Objectives**
- Output:
    - Hardware description of the **Datapath** and **Controller**

Ultimately, the High Level Synthetizer's work consist in placing the operations of an high level algorithm in time - i.e **Scheduling** - and space - i.e. **Binding**.

Constraints and Objectives

Constraints and **Objectives** shape the resulting values of **Area**, **Latency**, **Throughput**, etc.

Constraints are rules that the solutions must respect to be considered valid, restricting the design space; they are usually represented by an hard limit.
Objectives are targets that make some solutions preferable to others - i.e. we discard solutions that are Pareto dominated by others based on the objectives; they are not strict and are usually represented with a range.

Area is the number of modules/resources available or the size of the silicon die.
Latency is the number of cycles that elapses from the beginning of input to the complete output.
Throughput is the amount of data can processed in a given amount of time; it usually applies to repeated or continuous computations.

Datapath and Controller

It is a paradigm for describing architectures by partitioning them in 2 logical components:

the **Datapath** takes the external datainputs and converts them into dataoutputs through: functional units (e.g. adders, MACs), memory resources (i.e. registers and memory IP blocks), and interconnection resources (e.g. mux, buses, ports).

the **Controller** is a finite state machine (FSM) that handles control signals that can be external (e.g. start, reset) or coming from the datapath (e.g. the result of a comparison that determines the branch) and transforms them into signals to the datapath.

Together these components are functionally complete in respect to the specification: they model all possible behaviour we want to synthetize.

HLS and Compilers Front-End

The front-end use to extract IR from High Level Software Languages is based on existing mature compilers like GCC or LLVM; the reasons is that compiler theory and compilers has been around for more than 40years, and compilers already implement the interfaces to extract, analyse and optimize IRs.

Intermediate Representation (IR)

Intermediate Representations are language agnostic representations used by the synthetizer to represent the semantic of the code; analysis and optimization are all done on IRs and a synthetizer may switch from an IR to another more suitable at a particular stage;
The concept is equivalent to IR for compilers.

Abstract Syntex Tree (AST)

Abstract Syntax Trees is language dependent representation where each node is a terminal character of a grammar production: in particular internal nodes are operators and operands are the leaves

3-Address Code

In **3-Address Code** each statement is converted into the form *res = operand1 operator operand2*; this IR turns all operators into a standard format that can be more easily translated

Control-Flow Graph (CFG) with Basic Blocks

A **Basic Block** is a maximal sequence of instructions containing no label and no jumps: the control flow can only enter at the beginning and exit at the end, and once inside, all instructions of the basic block will be executed.

To obtain basic blocks, the synthetizer must find the **leader**s - i.e. the beginning of the blocks - which are jump targets like functions entry points and branches; the basic block extends from one leader to the next.

From the relationship of conditional/unconditional jumps between blocks, then the synthetizer composes the **Control-Flow Graph**; this representation is used to synthetize the controller that signals each component that represents a single basic block.

Single Static Assingment (SSA)

In **Single Static Assingment** each variable is assigned only once. Whenever a variable is assigned multiple times in the source code, a new variant of the variable is used instead, and if an expression uses a variable whose variant is only known at runtime, phi-functions is used instead: the phi-functions will determine at run-time the correct assignment.
Phi-functions are always found at the beginning of a basic block and create a new variant that can then be used throughout the basic block. Physically they are implemented as multiplexers or registers; if the variants bind to the same the register, then there is no movement of data and the phi-function is absolved by the register.

This IR highlights true data dependencies, distinguishing them from name dependencies.

Data-Flow Graph (DFG)

Each basic block will need to synthetized as datapath; to do this, the synthetizer builds a **Data-Flow Graph** that represent a single basic block and shows the data dependencies between the statements:
Read-after-Read (RaR or Input) dependencies do not represent real dependencies and can be executed in parallel Write-after-Write (WaW or Output) dependencies and Write-after-Read (WaR or Anti-) dependencies are name dependencies and can be eliminated using SSA.

In the end the DFG models only true dependencies highlighting what operations can be executed in parallel; using this graph the synthetizer can builda modularized data-path.

HLS and Compilers Back-End

Naturally, HLS analysis and optimization are hardware specific (e.g. memory partitioning in banks, bitwidth analysis) as is the final translation to RTL; for this reasons the **HLS Core Engine** (i.e. proprietary algorithms and back-end) is added to traditional open-source compilers back-end.

Compiler transformations are controlled by the hardware designer through pragma annotations, which easier to read since they are inside the code written next to the statements they refer to, or TCL directives, which are easier to change since they don't does not require modifying the source code and easier to automatize for automatic design space explorations.
Examples are: *inlining* functions, *parallelizing or pipelining* loops, *partitioning memories into banks*, etc.

The synthetizer allows the designer to choose the opportune bidwidth: the default choice is the size of the corresponding types in software (e.g. C integers, floats will be translated to 32bits, doubles to 64bits), but, using synthetizable libraries, we can specify types of arbitrary length saving unnecessary logic for unnecessary bits.

The synthetizer also translates function signatures into interfaces that maintan the same semantic: values that are passed-by-copy are converted into inputs ports, while value that are passed-by-reference are converted into memory interfaces to either local data (PLM/Scratchpad Memory with fixed latency) or remote data (Cache/Off-Chip memories with variable latency).
Standardized interfaces are available. Additional ports are also automatically added for the *start*, *reset*, *done* signals.

HLS: Scheduling and Binding

Scheduling

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

Scheduling is one of the first steps of the HLS core engine: it analyzes data dependencies for parallelism, exploits mutual exclusion (code that is never executed at the same time can be scheduled together), and optimizes loops (through parallelization or pipelining).

The inputs for the scheduling are:

- IRs, like the Data-Flow Graph or the Control-Data-Flow Graph
- Clock Period
- Functional Units Latencies (in nanoseconds, indipendently of the clock period)

The outputs is the start time (the clock cycle) of every operation.
The costraints are the data dependencies, resources, and timing constraint.
The goal is the minimum latency or a area/latency trade-off.

Scheduling Constraints

Scheduling is usually performed under some constraints, but can also perform **uncostrained** analysis: using infinite resources will provide the designer with the lower bound on the latency, and an estimation on the maximum amount of hardware resources.

Scheduling can be performed under some **resource constraints**: if some indipendent operations are serialized, more resources will be shared and the final result might stay under a certain resource budget.
This may be particularly useful when the target is an FPGA, since it has a set of CLBs.

Scheduling can be also performed under some **timing constraints**: the operations will be scheduled so that the last operations will finish before a certain time budget.

Scheduling Algorithms

Scheduling optimization can be done through **exact algorithms**: Linear Programming or Integer Linear Programming where the variables are the clock cycles and the resources and constraints are the order of operations according to data dependencies and the variables.

Being an NP-hard problem, scheduling is usually solved through **heuristic algorithms** or **meta-heuristic** (based on simulations).

As-Soon-As-Possible (ASAP) Scheduling

ASAP is an algorithm for unconstrained scheduling, and as such it provides a lower bound on latency.

When an operation is available - i.e. when all the predecessor operation in the Data-Flow Graph are completed, based on their clock cycle delays - the operation is scheduled.
start_time(op) = max(start_time(pred(op)))

As-Late-As-Possible (ALAP) Scheduling

ALAP is an algorithm for latency-constrained scheduling, without constraints on the resources.

Assuming the *sink NOP* is scheduled at the last possible clock cycle, the algorithm walks backwards through the DFG to find the last possible cycle at which an operation can be scheduled without creating delays.

Critical Path

Combining ASAP with the ALAP - i.e. setting the *sink NOP* time as the one found through ASAP - we can find the **critical path**: the difference between the ASAP time and the ALAP time for each operation represents the operation's mobility; those operations with 0 mobility cannot be delayed without causing delays to the overall computation.

Operations that are not critical may be delayed to allow for resource sharing, without impacting latency.

Even operations that are critical may be delayed, sometimes at little latency cost, but with large impacts on resources.

List-Based Scheduling

List-Based Scheduling, along with its many variants, is one of the most used algorithm in constrained scheduling.

It is a greedy, heuristic, polinomial (*O(n)*) algorithm that selects the operations to be scheduled based on a **criticality metric** (e.g. low mobility).

For each available resource, it selects an operation in that resource's ready list, based on the critically metric; the resource is set as busy, and when the operation completes it is removed and all its successors are added to the ready list.
The process is repeated until all operations have been scheduled.

What determines the quality of the final result is the quality of the criticality metric; if the algorithms uses only **static mobility**, then an operation with high mobility is likely to be starved.
Using **dynamic mobility** yields better results: the mobility of an operation is decreased each time it is passed over.

Scheduling Challenges

Complicating the algorithms is the existence of **multi-function** FUs that can execute more than one operation at a time, **pipelined** FU that can start an operation before compliting the previous one.

Some FUs may take less than a clock-cycle to complete, allowing then to perform more operation in the same clock-cycle **chaining**.

Some FUs, e.g. FUs that communicate with an external memory, may have **unbounded latency**; then a synchronization protocol is needed to schedule the successor.

Binding

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

The naive approach to avoid conflicts would be to assign each operation to a different functional unit. Since binding is usually ran after scheduling, we can use scheduling information to identify sharing opportunities: 2 operations that are never executed in the same clock cycle may be binded to the same FU.
Binding can be also ran before scheduling: binding 2 operations to the same FU will force their serialization.

Binding, like scheduling, it is a NP-Complete problem, usually solved through heuristic.
Binding algorithms consider operation types indipendently, since different kind of operations do not compete for the same resources.

Compatibility Graph

Two operations are said **compatible** if they are not concurrent and can be executed by the same resource.
Then, a **Compatibility Graph** can be built by adding an edge to between 2 compatible vertexes.

The binding problem can be re-formulated as the problem of partitioning the graph in cliques (fully connected subgraphs); each clique can share the same resource, and minimizing the number of cliques means minimizing the number of resources needed.

A variant of the problem imposes a minimum number of cliques - and consequentially a minimum number of FUs - to reduce the interconnection logic: in FPGAs, the cost of a multiplexers - which are needed for resource sharing - might be comparable to the cost of a FU.

Another variant of the problem utilizes weighted edges.

Conflict Graph

Two operations are said **conflicting** if they are not compatible. Then, a **Conflict Graph** can be built by adding an edge to between 2 conflicting vertexes.

The Conflict Graph is the complementary of the compatibility graph and identifies operations that cannot share resources.
Solving the **Coloring Problem** on the conflict graph, where each color is a resource, will yield the minimum number of resources; a trivial solution, maximizing the number of colors, is an admissible solution and provides an upper-bound on the number of resources.

HLS: Synthesis of the Micro-Architecture

After the operations of scheduling and binding, HLS proceeds with defining the RTL microarchitecture of **Controller**, **Datapath** and of the **interfaces with local/external memories**.

Controller

The **Controller** is described as a **Finite State Machine** with **State**s and **Transition**s; the Controller guides the control-flow of the Datapath through its **Output Function** (e.g. signals to multiplexers, write enable in the registers, etc.).

The **State**s represent the operations that must be executed in a clock cycle.
The **Transition**s represent the evolution of the behaviour overtime.

Among basic blocks, the controller transitions from from the last operation of a block to the first operation of the next one, as defined by the CFG.
Inside a basic block, the controller transitions between states that represent the sequence of operations that must be executed from beginning to end of the basic block, as defined by the in block scheduling.

The Output Function depends on the state of the controller, external signals and signals coming from the datapath.

Datapath

The **Datapath** is a collection of hardware resources for computation (i.e. functional units) and storage (local memories, e.g. registers) and interconnection (e.g. wires, multiplexers).

In particular, multiplexers are needed when there are multiple sources for an operand of a functional unit; and this is often due to resource sharing. In FPGAs, cost of a multiplexer can be comparable to the cost of a FU (*n* possible sources need *n-1* multiplexers); so we might reduce resource sharing, and still impact the area positively.

Memories amd Memory Operations

Memories have a big impact on the design of accelerators: the main reason to add hardware accelerators to a SoC is to exploit massive parallelism, which implies a steady stream of a lot of data in and out of the accelerator.

Pointer Synthesis

Up until now we've seen how to map software operations to hardware resources, but there is also the problem of mapping memory:
C-based languages allow the user to operate on a Unified Memory Space, using pointers that can target any variable regardless of the placement in memory; pointers to arbitrary memory are inconvenient to translate in hardware as the memory location is not transparent as every memory resource to which the pointer can refer to must be then connected to component.

Hardware synthesis of pointers requires:

- to partition the software unified memory model and map it onto different hardware resources
- to generate logic to resolve at pointer to a hardware resource, and access the data

Some pointers can be resolved statically; when it is not possible, an hardware architecture must be designed to allow access to multiple hardware resources; this is done through a **daisy chain** architecture, that broadcasts the request through an **internal memory bus** to various harware component, and the correct one responds - the cost is that memory access must then be sequentialized and parallelism is lost.

Private Local Memory

One of the advantages of HLS is being able to retarget the designs to many architectures; the memory design should also be indipendent of the architecture.

One of the design choices is the amount of memory banks that should be used:
each memory bank contains a fixed number of ports (usually 2), and splitting the PLM into banks allows for more parallel accesses; A first drawback is that the parallelism can be used only if the memory is accessed without creating conflicts, either by partitioning the data into the banks according to the known pattern or duplicating all the data into every bank. A second drawback is using multiple banks takes up more hardware resources for the same amount of memory: in FPGA a bank is mapped to a BRAM block, while in ASIC bank introduces a fixed area overhead of around 20%.

Array Paritioning and PLM Generation

To achieve maximum parallelism, it is possible to partition the local arrays into more memory banks, according to the access pattern of the component.
When the access pattern is linear, the partitioning is trivial; otherwise a **specialized PLM** can be generated starting from the scheduling obtained in the HLS process.
In general even if the access pattern looks linear and parallelizable in the high-level code, it is the scheduling of the operations which the determines the real access pattern in the hardware.

Compatible PLM

If two arrays have lifetimes that are not overlapping then they are **compatible** and they can be mapped on the same memory IP.
This is the case when two accelerators are not executed at the same time; but this information can only obtain at system level: it is possible to have HLS design first

the logic of the single components, and then a common **Memory Subsystem** according to the memory requirements of the components.

A classic example of a memory subsystem optimized for specific application is the **ping-pong buffer** (or **double buffer**) architecture used in producer-consumer pattern: using two buffers, the producer writes data on a buffer, while the consumer reads from the other; the two components never access the same memory at the same time and no conflict are created.

Memory Compatibility Graph

As in the binding phase, where the problem was mapping operations onto FUs, the problem of allocating data structures to memory resources can be represented through a graph.

Each node represent a data structure; two data structures are compatible if their lifetime is not overlapping, and their are connected by an edge.
The problem of **Memory Compatibility** can be re-formulated by partitioning the graph in cliques (fully connected subgraphs); a memory cost can be defined for each clique, and the goal is to find the partition the minimizes the total cost of all the cliques.

Then the address space is split into a number of caches; with an appropriate translation, the results can be fetch from the banks using *tags* and simple static logic, much like in a processor cache.

Interfaces with External Memory

While the size of the external memories has been growing linearly through the years, the size of internal memories has not; additionally the size of the problems are evergrowing.
For these reasons, the SoC will have to perform a large of data transfer to the off-chip memories, and these will be a bottleneck for the application.

HLS: Design Challenges

HW/SW Codesign

Hardware accelerators are used to speed up the execution of a software application; implementing the entire application in hardware is rarely, if ever, advantageous - e.g. interacting with the filesystem is bottleneck by I/O speed, so there is no advantage in executing it in hardware. Additionally, there might not be enough space to implement the whole application on the target device.

The designer has to integrate the software and the hardware.

On the **software** side, the designer has to:

- **partition** the application into tasks
- **map** them onto hardware components (among which is the CPU that will execute the tasks in software)
- **schedule** tasks to solve contention on shared resources

The **SW/HW partitioning** is a **bi-partitioning problem** (i.e. there are only 2 partitions).
The designer can use a **software oriented approach**, starting from the entire application in software and moving tasks to hardware depending on their performance,

or an **hardware oriented approach**, starting from the the entire application in hardware and moving the tasks to software depending on the cost of the hardware implementation.

On the **hardware** side, the designer has to **customize** the application; this can happen either **before fabrication** (as in the case of an SoC platform) or **after fabrication** (as in the case of an FPGA).

Challenges in Heterogeneous Systems' Design

The approach taken in the development of heterogeneous system is **bottom-up**, following a **compositional design**; this approach allows the design to tackle the evergrowing complexity of hardware design.
The problems that the designer faces are:

- in the short term: the development of efficient components and interconnections, working at a component level
- in the medium term: the integration and optimization of the components, working at a system level
- in the long term: the automatic porting of legacy applications on parallel and heterogeneous systems

For each of these challenges, the designer must explore throughtly the solution space, and the design of these solutions must be automated through **Electronic Computer Aided Design** tools.

Component Level Design

Nowadays HLS tools are able to approach complex software specification and automatically generate corresponding hardware acceleratos.

"*HLS is now push-button, but it has to be pushed the right way*"

The challenge of the designer is to explore the various solutions and in particular, to balance the amount of **resource sharing**; as we've seen, while resource sharing decreases the number of functional units, it increases the complexity of the interconnection: it increases the area of the design (i.e. by adding multiplexers), increases propagation delays and power consuption (i.e. longer wires).

Design Space Exploration at Component Level
The designer explores the design space by placing constraints and looking at the automatically-generated designs.
To explore the design space, starting from the specification, the designer can use a:

- **coarse-grain method**: by simply limiting the number of some hardware resources
- **fine-grain method**: by fine tuning the scheduling/binding, and automatically evaluate the thousands of resulting solutions

The exploration of the design space always happens at an RTL level, since logic synthesis would be to slow; the designer can also work on an estimate on the final result, with an approzimation error that is less than 4% in small components without aggressive optimization.

Even with coarse-grain methods, the design exploration can only be done a small portion of the specification: that's why the source code must be partitioned by the designer in components.

Correctness

In the design exploration, the compiler must always guarantee **correctness** of the synthetization - i.e. **the observable behaviour** - save for a few exceptions (e.g. differences in precision, associativity of the operators).
To do this, the compiler must act conservatively and can only apply transformation when it can prove that the observable behaviour; this may lead to loss of parallelism.

ANALYSIS OF DEPENDENCIES

Sequential languages present a **total order** of the program statements. Correctness can be mantained even with a **partial order**: operations that have no dependencies cam be considered concurrent and parallelized.

Real dependencies exist when there are **RaW data dependencies**, **control flow dependencies**.

It can also be particularly difficult to understand if system calls are indipendent, as their implementation is not always know.

Removing all fake dependencies from the code may take too long; so approximate analysis is used in general.

System Level Design

When designing an IP Block it's not usually possible to predict the best implementation with regard to the other blocks - the designer may not even know the target architecture, because the IP Block will be licensed or reursed.

Hardware design is a bottom-up process, based on **compositional design**: it starts from the design of the single IP blocks.
System Level Analysis allows the design to determine a system that is the best combination of the IP implementations.

Both the components and the system should offer some degree of decoupling: The system should be flexible enough that if some components change, it will still function properly: the most important tools for this purpose are **latency-insensitive protocol**s and standardization of the interfaces.
The components should be portable enough to be reused across different architectures, although it's impossible to have *fully portable components* especially when they are highly optimized; in general, a component should expose a standard interface, which is highly portable, and have tightly optimized internal design, which hinder portability across technolgies.

Design Space Exploration at System Level

The **Design Space Exploration at System Level** does not concern just the hardware, but also explores the various SW/HW partitioning solutions.
Through **mapping** the designer must determine a **mode of execution** - i.e. software or hardware - and **the processing element** - i.e. to which CPU assign a software task, and what hardware implementation to use - for each task.
Through **scheduling** the designer must also determine the ordering of the tasks.

When targeting FPGA, a possible option is the **dynamic reconfiguration**: generating the new "components" at runtime.
Dynamic reconfiguration is costly in terms of execution time.

An additional variable is given by the interconnection topology - i.e. a bus or a network-on-chip design.

This process should be automated, as the design space may be huge.

SERIAL SCHEDULE GENERATION SCHEM (SERIAL SGS)

Serial Schedule Generation Scheme is a genetic algorithm using a **constructive approach** used to produce possible mappings and scheduling: the algorithms picks a task at random and tries to bind it to various resources, waiting if no resources are free.

ANT COLONY OPTIMIZATION (ACO)

Ant Colony Optimization is a meta-heuristic based on the behaviour of ants looking for food: The ants explore the feasible territory with random walks, until they find food; if they do, they release pheromones while the take it back to their ant-hill; other ants than use the pheromones as guides.
When following the pheromones, if an ant finds more food, their path will be streghtened.
If the trace for the food is too long then the pheromones will fade, and the path will be weakened.

Starting from a feasible solution - usually a completely software serial solution that we know is feasible- a number *L* of threads (ants) pick one candidate at a time and schedules/binds it, until there are no more.
Then all threads evaluate the solution they have found, and save the best one.

How, at any decision steps, a thread chooses stochastically a candidate and a mapping/binding; the probability is determined by global and local informations.
The local information depends on the delay needed to schedule the task; the global information depends on the best solutions found on previous trips.

It's possible to vary the accuracy of the evaluation function, e.g. analytical estimation vs cycle-accurate simulation.

OTHER ALGORITHMS

Other algorithms used in design space exploration at system level are:

- Integer Linear Programming
- Taboo Search, generates a set of solution around the current one, putting the worst in a taboo list.
- Genetic Algorithm, generates many designs and ricombines the best parts of each solution

Compared to these algorithms, ACO is able to find the best solution often and in a relative small number of iterations; additionally, it moslty avoids unfeasible solutions.

Automatic Partitioning of the Implementation

Automatic Partitioning of the Implementation is an open problem: the initial, high level description is incredibly important and if it has not been written with HLS in mind (HLS-ready), it's extremely hard to perform component and system level design on it.

Security

Companies that design hardware must protect: the physical chips and their designs - i.e. **hardware security** - and the data elaborated on the chips - i.e. **hardware-assisted security**. Nowadays, SoC architectures are made using components designed from many different vendors and the supply chain (from design to end-of-life) for chips is distributed worldwide, increasing the surface of attack.

Per example, foundries can **overbuild**, creating counterfeit chips identical in quality to the legitimate ones - making them impossible to distinguish - without incurring in fixed costs (design, testing, etc) and allowing them to resell at a lower price for greater margins.
The known impact on the IC industry of these practices is around $4B, but might be just the tip of the iceberg.
Additionally, entire companies are built around designing and licensing hardware IPs: losing the secrecy would mean losing millions or billions in investments and that is one of the reasons why hardware companies' security spending has increased exponentially over the years.
These are two reasons why in recent years, Europe and the USA have committed to bring foundries back in their territories.

Finally, the data of the final user of the chip must be protected from hardware modifications and software based attacks; naturally, this must also happen at the design level.

The threats are:

- **Overbuilding**: foundries build more chips than request them, and then sell them as originals; since the design and build quality is the same, these illegal chips are impossible to distinguish
- **Reverse Engineering and IP Theft**: extract the chip functionality from the circuit design, stealing the technology
- **Hardware Trojan**s: malicious modifications to the chip that alters its functionalities, to either steal data (introducing **side channels**) running in the chip or harm the normal operations; it's particullary hard to identify a malicious trojan from a fault
- **Data Injection**: injection of spurious data to exploit hardware/software vulnerabilities

Protecting the design against every attack is not feasible, as it would be too expensive.

HLS and Security

HLS can be used to synthetize security functionalities:

- **Dynamic Information Flow Tracking** (**DIFT**)
- **Algorithm-Level Obfuscation**, an active security method
- **IP Watermarking**, a passive security method

But represent another attack surface, as the tool provider itself may be malicious:

- **Planned Obsolescence**
- **Key Recovery with Reduced-Round Attacks**

Dynamic Information Flow Tracking (DIFT)

Dynamic Information Flow Tracking (**DIFT**) is a technique to mark data as trusted or untrusted with tags, and track its flow through the accelerator; if untrusted data reaches a point where it can impact a system's functionalities, trap it.
To implement **Dynamic Information Flow Tracking**, an alternative parallel data-path to elaborate data tags is needed - i.e. **shadow logic** along with **taint memories**.

Synthetizing **shadow logic** and **tainted memory** is easy to do in HLS, and the level of granularity can be customized.
As the logic works in parallel, propagating the tags at the same time of the data, the perfomance cost are negligable; the resource costs go up to a 31% increase of area, for bit-level granularity applied on every component - that said, applying DIFT intelligently reduces the cost.

Logic Obfuscation & Algorithm-Level Obfuscation

Logic Obfuscation is a technique that protects against **IP theft** from foundries: the designer sends an obfuscated netlist that is dependent on a *k-bit* key.
Without the key, the chip are useless.

In this way, if a foundry overproduces chips, it won't be able to activate them and they will be useless.

There are many difficulties in implementing this solution: first, the designer must protect the key; second, the designer must guarantee that the obfuscation truly provides a *k-bit* protection.

Additionally, an attacker can analyze the circuit and strip away the blocks that render the components unusable; obfuscation must happen at an high-abstraction level.

There are many obfuscation methods that can be used:
Constant Obfuscation is the encryption of constants by XORing them with the key; not all constants should be encrypted: it is useful to encrypt data coefficients, mask values, but other predictable constants like reset values, signal polarity shouldn't be encrypted as it would allow key inference.
Control-Flow Obfuscation is achieved by XORing the result of control values with some bit of the key and using that result as the control variable, swapping the branches if needed.
Operation Obfuscation is achived by adding useless operation and multiplexing them based on some bit of the key.
Dependence Obfuscation adds fake data-paths, and selects the correct one using a bit on the key.

A way to evaluate the effectiveness of the obfuscation is **output corruptability**, the Hamming Distance between output generated with the corrected key and an incorrect key.

Naturally the modified design must mantain the same external behaviour; one could formally prove that the transformations applied generate an equivalent result, or use **simulation-based verification** although an exaustive search of the input space is clearly impossible.

Hardware Trojan && Watermarking

An **Hardware Trojan** is a malicious modification of a circuit, inserted to breach hardware-assisted security - e.g. a side-channel used to leak data through a radio transmission.
An hardware trojan is stealthy during normal execution; when the **trigger** is set, it is execute its **payload**.

A company may use **Benevolent Hardware Trojan** to **watermark** a design: when a trigger is activated, a component outputs a **signature** that is then propagated through the design; through a watermark, a designer may prove in a court of law that their design was stolen.

In a component, the functional controller commands the normal flow of the data-path; when the trigger is activated, the **payload controller** takes command instead and controls the final output.
To further obfuscate the payload controller and trigger, they can be swapped with a small FPGA inside the functional controller.

Watermaking is effective only in legal disputes: it requires identifying the author of the infringement and diving into a costy and lengthy legal battle.

Planned Obsolescense

Planned Obsolescence is the practice of degrading an IP performance after a certain amount of time, either by the hands of competitor or by the design house itself; detecting it is extremely hard, as it simulation catches only functional properties.

Degradation Attack

A **Degradation Attack** reduces the performance of an IP after a certain amount of time; the easiest way is to add bubble states to FSM: the more the computation is executed, the greater the effect - e.g. a loop, with trigger an execution counter.

Battery Exhaustion Attack

A **Battery Exhaustion Attack** reduces the duration of the battery after a certain amount of time, motivating people to change their devices.

Draining the battery can be achieved by increasing power consuption, which can be done by activating unused functional units (in particular, switching all the bits of the inputs to maximize power consuption).

Key Recovery with Reduced-Round Attack

Cryptographic algorithms must be run a certain number of times on the input to be effective, so the hardware that implements them also performs loops.
A **Reduced-Round Attack** is a technique where the counter used to performed encryption is changed to a lower value, reducing the number of rounds that the encryption algorithm will run and breaking it.