Methodologies for Synthesizable Programmable Devices based on Multi-Stage Switching Networks

Candidato:
Matteo Cuppini

Relatori:
Prof. Eleonora Franchi Scarselli
Prof. Roberto Guerrieri
Dott. Ing. Claudio Mucci

Coordinatore Dottorato:
Prof. Claudio Fiegna

Esame finale anno: 2015

Contents

List of Figures

List of Tables

Introduction

One of the reasons of System on Chips (SoCs) success is their capability to couple performance and miniaturization with time-to-market. Typically the design of a SoC follows implementation approaches based on Standard-Cells: such approach allows designers to select the most appropriate technology flavours for a given application in order to find the best trade-off among speed, area and power. In addition, in a Standard-Cell based flow the availability of a rich portfolio of libraries allows the designer to explore quite easily different implementation scenarios, including libraries with different transistor thresholds (V_t) to balance speed and leakage, or libraries for high-density design, to balance area and speed. This design style ensures time-to-market and risk-reduction inheriting its robustness by the pre-verification and re-use of cells and IPs.

Usually a SoC features a processor-based environment enhanced with a set of accelerators and I/O peripherals. Flexibility and upgradability are precious features that can represent an added value for a SoC in terms of market width and cost reduction, and are typically ensured adopting Application Specific Standard Processors (ASSPs) and Digital Signal Processors (DSPs) whereas performance requirements allow designers to plan software-programmable solutions. So far, SoC market improves flexibility through the increase of processor-based computation, be it on the processor core (e.g. adopting multicore processors) or be it on the acceleration domains (e.g. by parallel architectures based on GPU-like structures).

The potentialities offered by these solutions are nowadays enlarged thanks to the opportunities offered by the scaling of silicon technology, that allows designers to integrate a growing amount of functionalities on the same area. As a consequence, the rise of non-recurring engineering (NRE) costs associated with complexity becomes a major factor in SoC design, limiting both scaling opportunities and the flexibility advantages offered by the integration of more complex computational units. Among NRE costs, one factor that is steadily growing is the cost of masks [1], which makes it economically sustainable to produce devices in advanced technologies only for high-volume or high-value markets [2]. Hence the adoption of design strategies able to reduce such costs and preserve high performance represents a challenge. IP-reuse strategies are a common approach to relieving design and verification costs by re-utilization of pre-designed and pre-verified synthesizable IPs. This allows designers to optimize - in terms of area, speed and power - the actual implementation of the IP for the specific target, at the same time ensuring goo time-to-market and risk-mitigation.

The introduction of embedded programmable elements can represent an appealing solution, able both to guarantee the desired flexibility and upgradabilty and to widen the SoC market, enlarging its application scenario or extending its lifetime. In both cases, the reprogrammability allows one to spread NRE costs over a growing number of products. Such results can be achieved for example through an intense amount of processor-based computations, whether based on simple processors, multi-core processors or domain-specific processing units like DSP and GPU structures.

An alternative option can be represented by the adoption of field programmable devices in the form of embedded FPGA (eFPGA) cores, solution that can provide bit-level optimization for those applications which benefits from synthesis. Lots of time has been devoted in the literature to showing the computational advantage of eFPGA in terms of $GOPS/mm^2$ over processors and DSPs [3]. Nonetheless the potential benefits introduced by such an approach are hampered by several well-known disadvantages that have historically restricted use of them on the market. The biggest drawback is the performance penalties and area overhead introduced by FPGA technology which should be considered at least one order of magnitude more than standard cell ASIC implementation. To overcome this, many FPGAs - especially the high-end solutions - are integrating many hard-macros (multipliers, high-performance processors, high-speed interfaces..) with the aim to become more effective thanks to hardening. This research of effectiveness involves also the embedded world, so that embedded FPGA cores, to be appealing, must represent a small and effective part of the overall device and provide real added value for some specific aspect of the system.

In this scenario this thesis - granted by STMicroelectronics - proposes a design methodology for a synthesizable programmable device designed to be embedded in a SoC. Usually programmable logic cores are offered as hard macros of fixed size, optimized through custom layout for general purpose applications. Soft-core eFPGA approaches have been introduced in several works [4][5][6][7], on account of the straightforward advantages of integration (e.g. pin position, technology flavours), flexibility (e.g. size, DSP-block budget) and portability through technology nodes. Instead of focusing on the ”computational” aspects of flexibility (size, DSP-Memory blocks), in this thesis a soft-core embedded FPGA (eFPGA) is presented and analyzed in terms of the opportunities given by a fully synthesizable approach, following an implementation flow based on Standard-Cell methodology.

A key point of the proposed eFPGA template is that it adopts a Multi-Stage Switching Network (MSSN) as the foundation of the programmable interconnects. This can avoid, or at least limit, the need for custom circuit design also in the network, since it can be efficiently synthesized and optimized through a standard cell based implementation flow, while it ensures a congestion-free network topology, thanks to the intrinsic properties of some specific MSSNs.

As presented in [8] and [9], the evaluation of the flexibility potentialities of the eFPGA has been performed using different technology libraries through a design space exploration in terms of area-speed-leakage trade-offs, enabled by the full synthesizability of the template. Since the most relevant disadvantage of the adopted soft approach, compared to a hard core, is represented by a performance overhead increase, the eFPGA analysis has been made targeting small area budgets. The generation of the configuration bitstream has been obtained thanks to the implementation of a custom CAD flow environment, and has allowed functional verification and performance evaluation through an application-aware analysis.

The rest of this work is organized as follows. Chapter 1 provides an overview of the state-of-the-art of the Programmable Logic Devices, with particular focus on the FPGAs, for which both architectural and CAD support aspects have been taken into consideration. Chapter 2 describes the implemented Multi Stage Switching Network: after an introduction on the state-of-the-art of this kind of interconnects, the architecture of the MSSN is described both in terms of topology and dedicated software support, realized using C-language. The custom software support is also able to realize routing, implementing an algorithm that follows a precise strategy: as explained in a dedicated section of Chapter 2, routing choices heavily influence the performance of the network. The chapter ends with an analysis of a particular case of Multi Stage Switching Network. Chapter 3 presents the realized eFPGA soft-core template, describing how its architecture can provide synthesizability, flexibility and portability features; next to this, a custom CAD flow support has been realized, and is presented in the end of the chapter. Chapter 4 and 5 contain the demonstration of the described eFPGA features: to do so, the eFPGA has been implemented using two different standard cells libraries (CMOS65nm and BCD9s0.11μm). For each the results are analyzed, thanks also to the utilization of the custom CAD flow support, and the conclusions are drawn in the final chapter.

Chapter 1

Programmable Logic Devices

A Programmable Logic Device (PLD) is a particular kind of electronic component with reconfigurability and reprogrammability features. Thanks to its structure, typically made of configurable logic and flip-flops linked together, a PLD can be configured by the user in order to perform different functions. The possibility of programming (and often re-programming) this kind of device introduces several advantages if compared to fixed logic devices (e.g. ASICs), mainly in terms of costs reduction for low volume applications. Designing with a PLD requires indeed less time, hence ”non-recurring engineering” (NRE) costs reduction and shorter time-to-market, together with risk reduction thanks to the availability, in some cases, of changes *on-the-field*. On the other hand a PLD pays in terms of performances if compared to an ASIC, with considerable area overhead and speed reduction, and results to be appealing only for low-volume markets. Depending on their characteristics, programmable logic devices can be classified as:

- Simple Programmable Logic Devices (SPLD)
- Complex Programmable Logic Devices (CPLD)
- Field Programmable Gate Arrays (FPGA)

Even though featuring different characteristics, all programmable logic devices are united by their architecture, composed of computational logic and routing interconnections. Another common point is represented by the need of a dedicated software environment able to develop, simulate and test a design that is going to be mapped into the device.

1.1 SPLD

As the most simple typology of programmable logic devices, SPLDs are typically built with an array of AND gates (AND-array) and an array of OR gates (OR-array). This kind of devices are hence characterized by two different levels of synthesis, and in many cases are enhanced with sequential logic elements (e.g. Flip-Flops) able to store the outputs. There are three fundamentals types of SPLD:

- PROM (Programmable Read Only Memories)
- PAL (Programmable Array Logic)
- PLA (Programmable Logic Array)

that differ between them in the placement of the programmable connections in the AND-OR arrays, as described in the following.

PROM

A programmable read-only memory (ROM) can be used to create arbitrary combinational logic functions of a number of inputs. As depicted in figure 1.1 the input lines to the AND array - used to generate all the midterms of the input signals - are hard-wired, while the output lines to the OR array are programmable. The utilization of a PROM as programmable device introduces penalties in terms of high power consumption, frequency reduction and problems related to asynchronous logic transitions (glitch propagation) in case of completely combinational architectures.

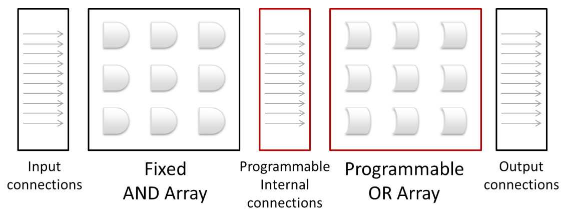


Figure 1.1: PROM diagram

PAL

In a programmable array logic (PAL) both internal connections and OR array are fixed, while the AND array can be programmed to generate a subset of the combinations of the inputs, the outputs, or their complement (figure 1.2). As only part of the device is programmable, a PAL is easy to program, but its architecture limits the design to simple state machines and simple combinational circuits.

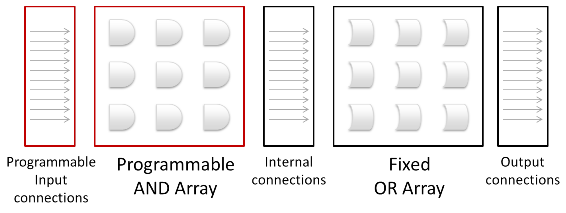


Figure 1.2: PAL diagram

PLA

A PLA has two sets of programmable planes, the AND and the OR array (figure 1.3), so that a larger number of logic functions can be synthesized in the form of sum of products and, in some cases, products of sums. The available programmability of both planes increases the complexity of the design (up to 1K equivalent gates), paying in terms of costs (need for more sophisticated tools) and speed performances, but on the other hand allowing the implementation of any function up to a product term limit.

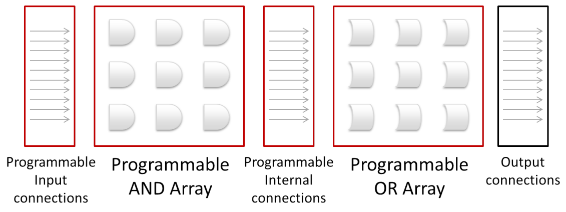

Programmable Input connections
Programmable AND Array
Programmable Internal connections
Programmable OR Array
Output connections

Figure 1.3: PLA diagram

1.2 CPLD

Going up with the total amount of logic available, a complex programmable logic device (CPLD) is a macrocell which contains a bunch of simple PLD blocks whose inputs and outputs are connected together by a global interconnection matrix, as depicted in figure 1.4. Thanks to its structure, a CPLD

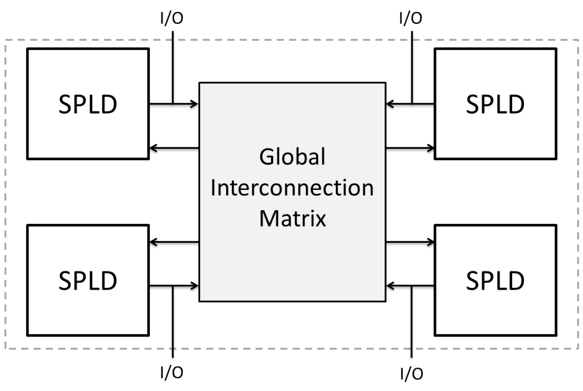


Figure 1.4: CPLD example

offers two levels of programmability, hence multi-level synthesis: each PLD block can be indeed programmed, as well as the interconnections between them. One of the advantages of such kind of device is that it features a non-volatile configuration memory (like its SPLD predecessors), so that it can function immediately on system start-up without the need of an external configuration memory. The increased amount of logic available - up to 10K gates - together with their predictable timing characteristics make CPLDs suitable for control applications, signal conditioning or simple data processing.

1.3 FPGA

Together with CPLDs, field programmable gate arrays (FPGA) are nowadays the most used programmable devices, since they offer high amount of logic (more than 1 million equivalent gates) together with the most features and the highest performances between all the programmable logic devices. For their ability to realize approximately any kind of complex digital circuit or system, FPGAs are historically compared to ASICs design, with respect to which they are able to offer flexibility advantages on the same time paying in terms of area, speed and power performances. As for the less evolved programmable logic devices previously described, FPGAs can be characterized according two essential technologies: architecture and computer-aided design (CAD) tools that must be employed to create the design.

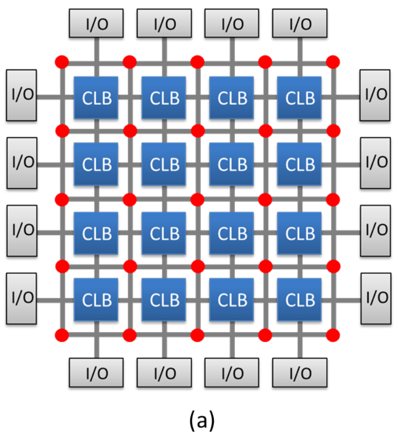
Since this thesis proposes and analyzes an FPGA, and in particular an embedded FPGA, a more detailed overview of both architecture and CAD aspects is presented.

1.3.1 FPGA Architecture

An FPGA can be described as composed of an array of computational logic blocks (CLBs) of potentially different types connected using a programmable routing fabric, that is also typically used to connect I/O blocks to make off-chip connections [10].

Routing plays a significant role in the overall efficiency of an FPGA architecture, since it heavily influences area performances (with an overhead up to $\approx 80 - 90\%$ of the overall design) and is responsible for the efficiency of the connections. The macroscopic arrangement of routing resources is commonly called global routing, while the microscopic details regarding wires or switches are known as detailed routing.

The most common FPGA global routing architectures can be characterized as either island-style or hierarchical.



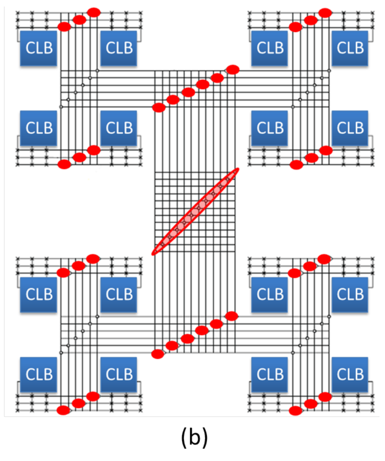


Figure 1.5: Example of island-style (a) and hierarchical (b) FPGA

Figure 1.5 (a) shows an example of traditional island-style architecture: the CLBs are organized in a 2D grid, and can be seen as ”islands” in a sea of programmable routing interconnects. Detailed routing in island-style architectures offers a large number of options, such as the possibility of vary the width of the routing channels (the space between logic blocks) or the length of the wires used to realize connections (the so called segmented routing) so as to reduce delays and obtain better layout optimizations. For their general-purposeness island-style architectures are the most used in commercial FPGAs, but when locality of connections becomes a major factor in a design hierarchical solutions represent the best choice. In a hierarchical arrangement (figure 1.5 (b)) the logic blocks are organized into distinct clusters, so that connections between CLBs belonging to the same group can be made using wire segments at the lowest level of the hierarchy - obtaining faster paths - while connecting distant clusters requires the traversal of one or more levels of the hierarchy. The main drawback - and also the reason why island-style architectures are usually preferred - is that design mapping can become an issue, in cases where the distribution of the design wire length does not matches the hierarchy of the FPGA architecture.

As regards detailed routing, a relevant case of study in FPGA design regards the type of switch used to realize connections [10]. As a matter of fact in both architectural solutions interconnect wire crossing are handled using different kinds of switch boxes (red dots in figure 1.5), able to connect routing tracks. These routing switches are typically made by a collection of elementary logic blocks such as pass-transistors, buffers or multiplexers, that according to their structure can connect bidirectional or unidirectional wires. Historically FPGA developers adopted bidirectional routing segments interconnected through pass-transistors or tri-state buffers [11]: an example of two back-to-back tri-state drivers used as bidirectional switches is shown in figure 1.6 (a). As the requirements of the applications have grown in complexity, research has been done to overcome the issues - speed and area - associated both with bit-level programmability and with routing congestion. Hence bidirectional segmented routing requiring pass-transistors or tri-state drivers has been replaced by unidirectional routing and multiplexers (figure 1.6 (b)), bringing area and delay improvements as

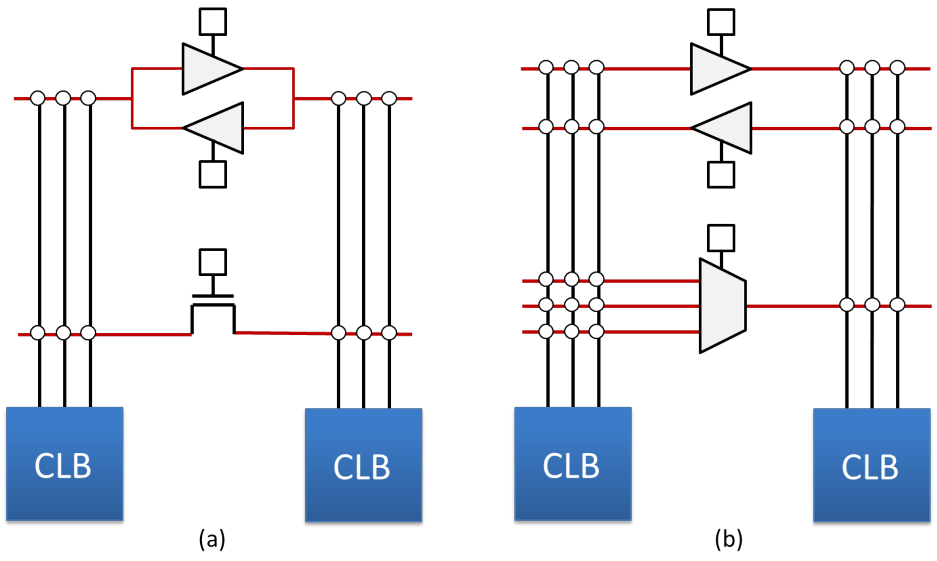


Figure 1.6: Bidirectional (a) and unidirectional (b) wires

shown by experimental results [12].

Computational logic blocks (CLBs) are the basic element of an FPGA, since they are responsible for basic computation and storage for a target application design. The architecture of a logic block can vary in a wide spectrum of choices: depending on the required functionalities and area-speed-power performances, a CLB can be made of simple transistors, NAND gates, interconnection of multiplexers, lookup tables (LUTs) or PAL-style wide input gates [10]. The best trade-off - and the solution adopted by most FPGA vendors - between too fine-grained and too coarse-grained solutions is represented by the utilization of LUT-based CLBs. In such cases a CLB is internally organized in a set of clusters, called basic logic elements (BLEs), the number of which typically varies from 4 to 10. As depicted in figure 1.7 (b) each BLE contains a k-inputs LUT, a Flip-Flop and a multiplexer that allows to select between combinational and sequential output. Many different works have explored the impact of LUT size on speed and area performances, showing that the best trade-off can be obtained with a number of LUT inputs between 4 and 6 [10]. In addition, in many modern FPGA designs a set of specific purpose hard blocks (e.g. multipliers, adders or DSPs) can be introduced into BLE clusters with the aim to increase design potentialities.

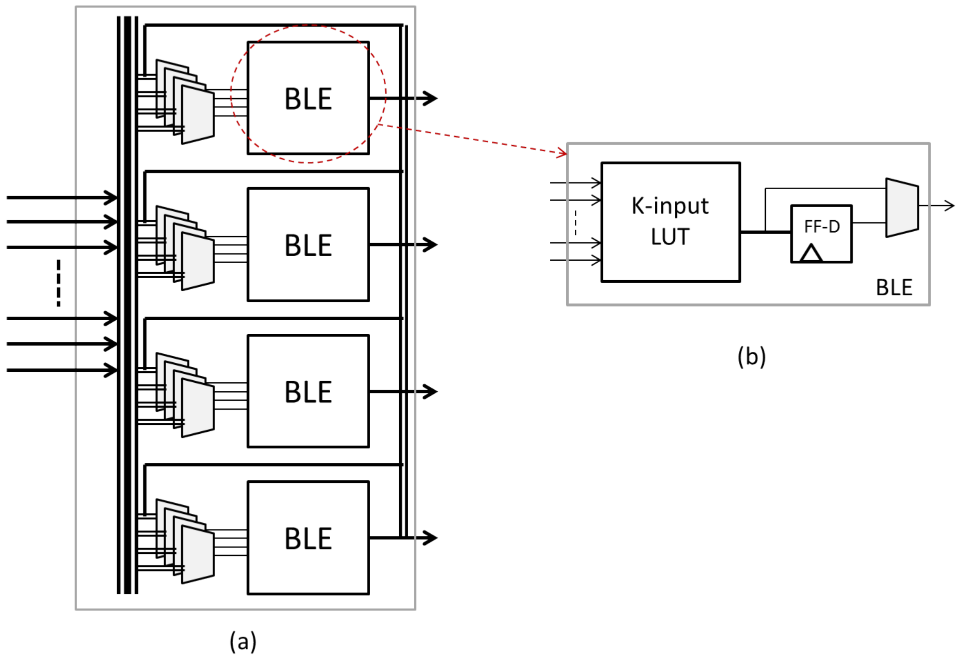


Figure 1.7: Example of CLB (a) and BLE (b) internal architecture

1.3.2 FPGA Software Flow

A key element that involves FPGA architecture research and heavily influences the quality of a design implementation is the development of a Computer Aided Design (CAD) tool able to map target applications into the FPGA. Not only the generation of the configuration bitstream is performed thanks to the CAD environment, but also any FPGA architectural exploration is highly dependent on the quality and level of freedom the tools provide to designers and researchers.

As shown in figure 1.8, a typical FPGA software flow takes as input the VHDL/Verilog Hardware Description Language (HDL) description of the target application design and passes it through a set of intermediate steps all the way down to configuration bitstream. As the CAD flow needs to be aware of the characteristics of the target FPGA, other inputs to the design flow typically include design constraints and the description of the device. An overview of the steps listed in figure 1.8 is given in the following part of this section.

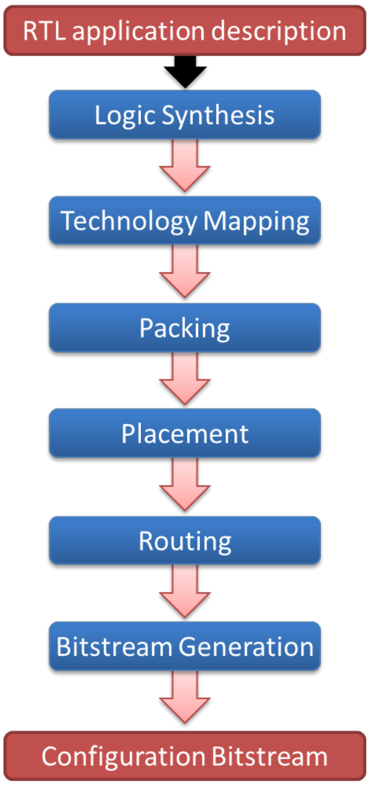


Figure 1.8: FPGA design flow overview

Logic Synthesis

Logic synthesis transforms an HDL description (VHDL or Verilog) into a set of boolean gates and Flip-Flops netlist, transforming the Register-Transfer-Level (RTL) description of a design into a hierarchical boolean network [13]. RTL elaboration includes the identification of datapath operations, such as additions, multiplications, register files and/or memory blocks, and control logic, which are elaborated into finite-state machines (FSM) and/or generic boolean networks. The significance of this operations is further increased by the fact that nowadays most of the modern FPGAs have specified architectural supports for datapath operations, such as adders with dedicated carry look ahead structures and embedded multipliers. An architecture-independent optimization of both datapath and control logic operations is also performed using techniques such as constant propagation or operation sharing for the datapath, and FSM minimization and retiming for the control logic operations. This step is usually technology independent with no FPGA specific optimization done to the logic.

Technology Mapping

Generally speaking a technology mapping operation transforms a technology independent logic network into gates implemented with a technology library. In the FPGA flow, the logic network is the circuit description of Boolean logic gates given by the synthesis step, and the library of cells is composed of LUTs, Flip-Flops and the on-chip dedicated blocks (e.g. adders, multipliers).

Packing

As described in the previous paragraph, LUTs, Flip-Flops and specific purpose hard blocks are organized in BLEs. The operation of grouping BLEs into logic blocks (CLBs) that can be mapped directly to a logic element (CLB) of an FPGA is called packing. There are three different approaches that a packing algorithm can follow, namely top-down, depth-optimal and bottom-up [14]. Bottom-up approaches partition the BLEs into clusters taking into account one of them at a time, creating groups according to an attraction function that tries to minimize a specific parameter (e.g. the number of shared nets between blocks). Depth-optimal solutions attempt to minimize delay at the expense of logic duplication, while top-down approaches partition the logic block clusters by successively subdividing the network or by iteratively moving BLEs between parts [14]. Thanks to their fast runtimes and reasonable timing delays achievable bottom-up approaches are the most used in FPGA CAD tools.

Placement

Placement step has a significant impact on performance and routability, since it is responsible for the logic blocks location on the FPGA. Typically a placement engine tries to minimize a parameter (e.g. a cost function) using a specific kind of algorithm. According to its optimization goal, a placement cost function can minimize the required wiring (wirelength-driven placement), balance wire density across the FPGA (routability-driven placement) or maximize circuit speed (timing-driven placement). The three major algorithms used in FPGA placers are min-cut (partitioning-based), analytic and simulated annealing based [13].

Routing

Since in a programmable logic device, and in particular in an FPGA, routing interconnects represent a significant part of the overall design (up to $\approx 80-90\%$), thus influencing also speed and power performances, routing step can be defined as the most important in FPGA design. As for ASIC designs, an FPGA routing step tries to successfully connect all signal nets - listed in a *netlist* file - of a design according to specified timing constraints. The main difference is that in an FPGA the available resources are fixed, since prefabricated, so that achieving high percentage of routability is more challenging. A routing engine must hence be aware of the properties of the target FPGA architecture, both in terms of global and detailed routing, and like placement step is based on a particular algorithm able to realize connections and minimize resource utilization. Another important measure of the quality produced by an FPGA routing algorithm is the critical path delay, that can be defined as the maximum delay of a combinational path in the netlist. The maximum frequency at which a netlist can be clocked has an inverse relationship with the critical path delay, thus larger critical path delays slow down the speed of a design.

Bistream generation

As final step of the flow, it takes as input the mapping, placement and routing information and generates the bitstream to program the configuration cells of LUTs, routing, I/Os and hard blocks to implement the mapped application on the target FPGA.

Since focused on the development of a programmable logic device, this chapter has proposed a brief overview of the available PLDs with special glance to FPGA architectures, the specific target of this thesis. As highlighted, routing in an FPGA represents a key point both from the architectural and the performance evaluation point of view: for this reason in this thesis a hierarchical approach is proposed. The innovation of the presented work is the utilization of Multi-Stage Switching Networks as foundation of the programmable interconnects. The next chapter will provide an overview of this kind of network, from the state of the art to the particular topology adopted.

Chapter 2

Multi-Stage Switching Networks

Nowadays digital systems realize the communication between their subsystems (e.g. logic and memories) using interconnection networks, that can be defined as programmable systems that transport data between terminals [15] (figure 2.1).

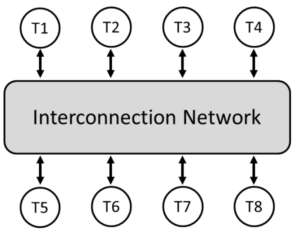


Figure 2.1: Functional view of an interconnection network

Switching Networks are a particular kind of interconnection networks that finds its origins in the communication industry, where simple switches were first implemented as electromechanical assemblies in telephone switching offices in the early 1900s. With the growth of computer industry, applications for switching networks within high-performance computing machines have become more frequent, thanks to their ability to handle circuit requests that are a permutation of its terminals (inputs and outputs), leading to the introduction of more evolved network topologies such as *Multi-Stage* switching networks (MSSN).

A number of general surveys of interconnection networks have been published [16] [17], where many different kind of MSSN have been classified according to their topology, defined as the static arrangement of channels and nodes, or their routing properties, defined as the ability to effect connections between input ports and output ports.

Multi-Stage Switching Networks can be further classified according to their blocking characteristic. More precisely, a network can be classified as *non-blocking* if it can realize any connection between an input and an output regardless the connections already established across the network, or *blocking* if it cannot handle all the connectivity requests [15].

A distinction has to be made between the *non-blocking* networks. A network is *strictly non-blocking* if any permutation can be set up incrementally, one circuit at a time, without the need to reroute (or rearrange) any of the connections that are already set up. In contrast, a network is *rearrangeably non-blocking* if it can route connections for arbitrary permutations, but incremental construction of a permutation can require rearranging some of the early connections to permit later circuits to be set up.

All of the above definitions can be applied for *unicast* traffic (one input connected to one output), *multicast* traffic (one input connected to several outputs) or *broadcast* traffic (one input to all outputs).

The goal of the research work in the field of interconnection networks proposed in this thesis has been to find an application of this kind of interconnects to the routing of an embedded programmable device. For that, the analysis of the state of the art has focused on rearrangeably non-blocking networks for multicast traffic, with special glance to architectures able to guarantee a small area overhead for a number of connections (i.e. network I/Os) that for an embedded PLD can reach some thousands of points. An brief survey of non-blocking networks is given in the following section; then the proposed **rearrangeably non-blocking** - for multicast traffic - Multi Stage Switching Network is described in details, both in terms of topology and routing algorithm. A key point of the MSSN, together with its connectivity properties, is represented by its fully synthesizability and optimizability through a standard cell based implementation flow, feature that makes the network suitable for its utilization in an embedded programmable device. At last, the final paragraph of the chapter presents an architectural analysis of different kinds of MSSNs, both in terms of frequency and computational density performances.

2.1 Non-blocking Interconnection Networks

A crossbar network, or crossbar switch, is the most simple strictly non-blocking interconnect since it is able to realize any kind of I/Os permutation without intermediate stages. An $n \times m$ crossbar can connect n inputs to m outputs thanks to $n \times m$ crosspoints realized with simple switches, as shown in figure 2.2. Thanks to its architecture a crossbar network is strictly non-blocking for both unicast and multicast traffic, since any unconnected output can be connected to any input by simply closing the switch connecting the input and the output lines. The main drawbacks of a crossbar are costs and scalability: although economical in small configurations, the cost of a square $n \times n$ crossbar increases indeed as n^2.

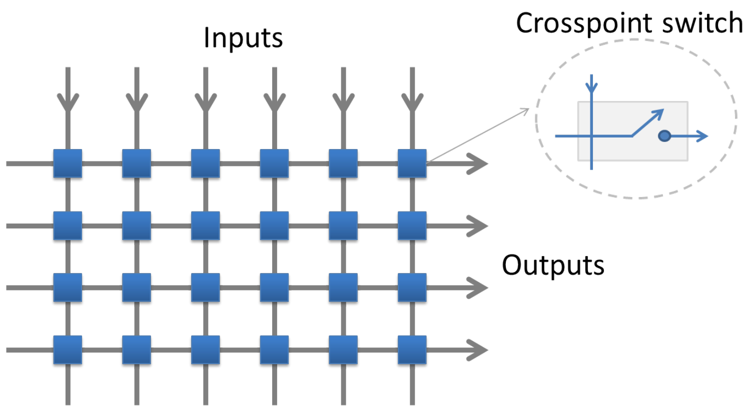


Figure 2.2: Example of a 6x4 crossbar

Multi-Stage Switching Networks (MSSN) were born to overcome the crossbars scalability issues. As shown in figure 2.3, a MSSN can be seen as constructed from stages of identical crossbar switching elements of low degree. Intermediate stages (also called levels) are connected together by an interconnection pattern of links, sometimes referred to as a permutation network but also called a shuffle. The routing properties of a MSSN are heavily determined both by the dimensions of its switching elements and by the pattern of interconnection links.

A Clos network [15] [18] is a symmetric three stage network characterized by a triple (*m*, *n*, *r*), where *m* is the number of middle stage switches,

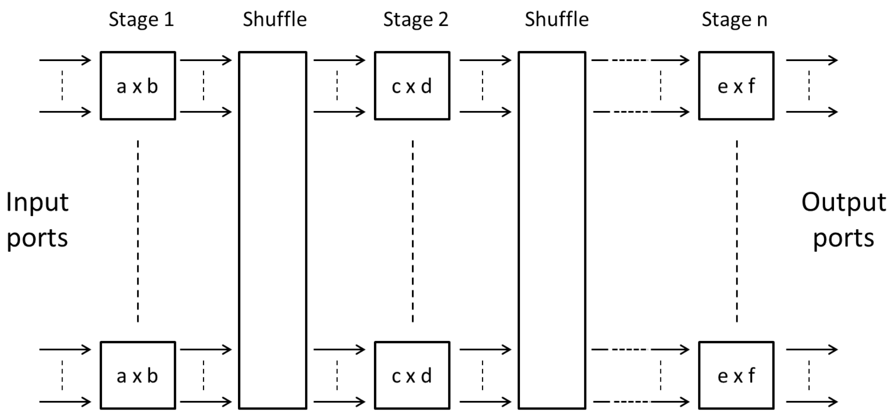


Figure 2.3: The general structure of a Multi-Stage Switching Network

n the number of input (output) ports on each input (output) switch, and r is the number of input and output switches (figure 2.4). The I/O switches can be thought of as moving the traffic horizontally to and from the vertical middle switches, while the middle switches move the traffic vertically from a horizontal input switch to a horizontal output switch. Clos networks with any odd number of stages can be derived recursively by replacing the switches of the middle stage with three-stage Clos networks. As regards

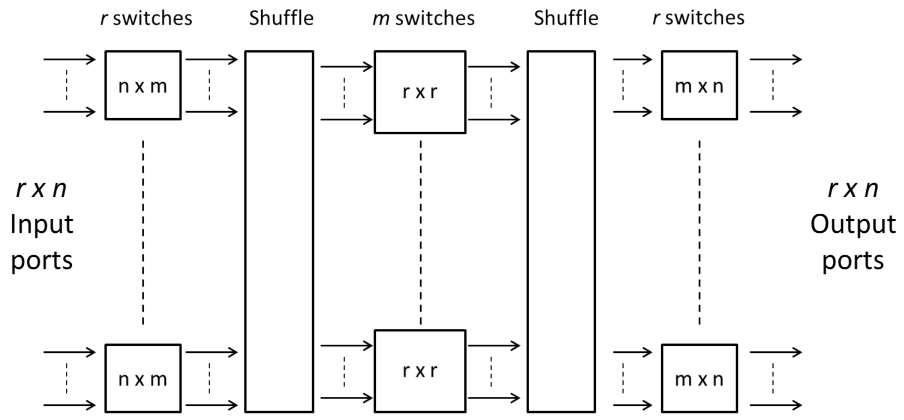


Figure 2.4: A *m, n, r* Clos network

unicast traffic, a Clos network is strictly non-blocking if the condition

$$m \geq 2n - 1 \tag{2.1}$$

holds. Routing multicast traffic is instead an harder problem, since more middle switches are required [15], resulting in a considerable area increase for high values (i.e. ≥ 100) of n.

The Benes network is a special case of the Clos network constructed from 2x2 switches (figure 2.5). These networks are notable because their architecture results to be rearrangeable non-blocking for unicast traffic if [19]

$$m \geq n \tag{2.2}$$

and they require the minimum number of crosspoints to connect N (with $N = r \times n$) input ports: $O(NlogN)$. Moreover both the symmetry and the modularity of the architecture, built using only one typology of switch, can represent an advantage from the design implementation point of view.

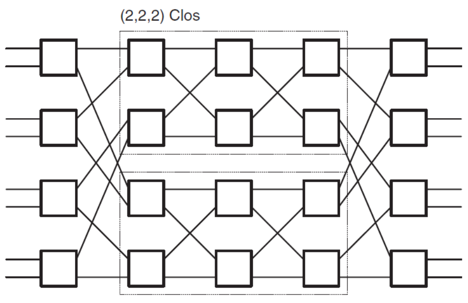


Figure 2.5: A Benes network built from a Clos network

With the aim to combine the advantages provided by all the described topologies, the architectural choice carried out in this thesis will be presented in the next sections. Since the performances of a network are determined both by its architecture and by its routing algorithm [15], a section will be dedicated to each of these aspects.

2.2 MSSN Architecture

The architecture of the MSSN has been chosen combining both a theoretical aspect - state of the art analysis - and a practical one - architectural exploration - obtained through the implementation of a software support. For that, this section first provides a topological description of the chosen MSSN, then describes the developed custom software environment.

2.2.1 Topology

The chosen architecture was built starting from an 2x2 switches architecture featuring butterfly-like [15] interconnection links between stages: figure 2.6 shows an 8-input example of such network.

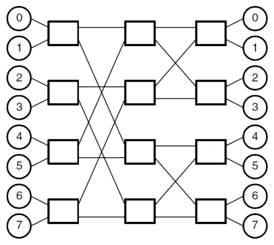


Figure 2.6: A 8-input butterfly network

Butterfly topology has been chosen since able to guarantee the minimum number of stages, hence low latency and complexity: for an N-input network with switches of degree k (also called *radix-k* switches) the number of stages is $H = \log_k N$. Besides such attractive feature, a basic butterfly network presents two main drawbacks. The first is related to the logarithmic structure of the network: for a network with radix-k switches the number of I/Os is constrained to be a power of k in size. The second regards blocking features: due to its asymmetric, low-latency structure, a baseline butterfly network has no path diversity, since there is exactly one route from each source node (input) to each destination node (output). This implies that non-blocking feature cannot be guaranteed for multicast traffic.

This problem can be addressed by adding extra stages to the butterfly [15], with the aim to obtain a symmetric Benes topology. For that, as shown in figure 2.7, the baseline N-inputs network (a) has been doubled on the horizontal direction with a specular inverse butterfly network. In this operation, the last stage of the baseline and the first stage of the inverse network have been fused together, creating a central common stage (b).

The resulting network has then been duplicated on the vertical direction, obtaining stages with 2x number of switches (c). With the aim to further improve path diversity - hence to reduce blocking probability - and maintain the same I/O multiplicity (N) two stages featuring 1-input

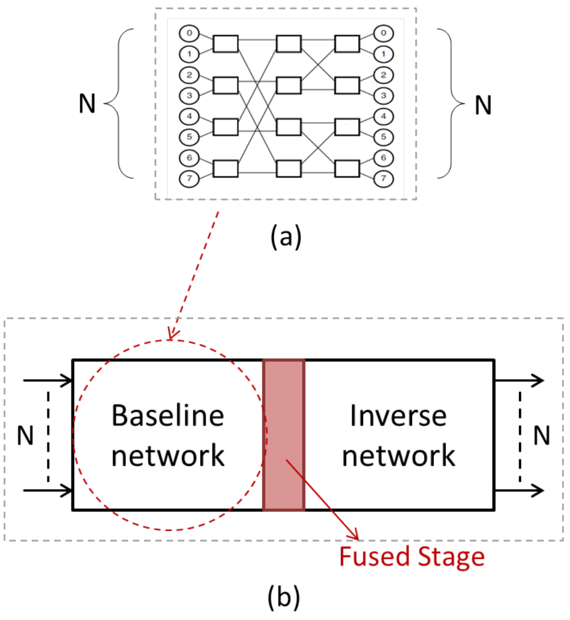


Figure 2.7: MSSN architecture guideline - network doubled on the horizontal direction

switches have been added at the I/O extremes of the network (figure 2.8).

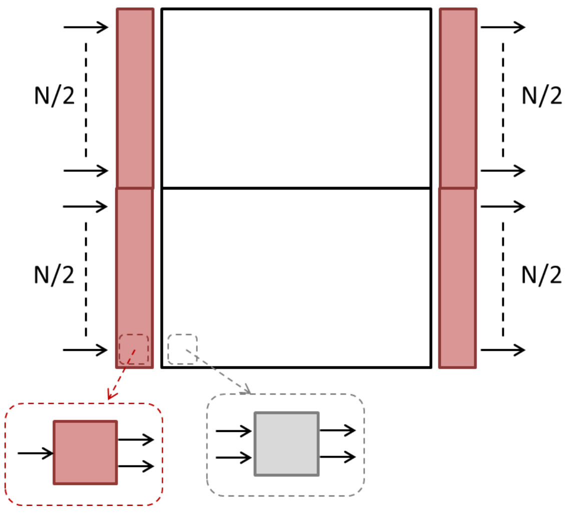


Figure 2.8: MSSN architecture guideline - network doubled on the vertical direction

The obtained Benes *back-to-back* butterfly-like MSSN is shown in figure 2.9, where the central fused stage and the added I/O stages are highlighted.

With this topology, an N-input (and N-output) network with 2x2 switches

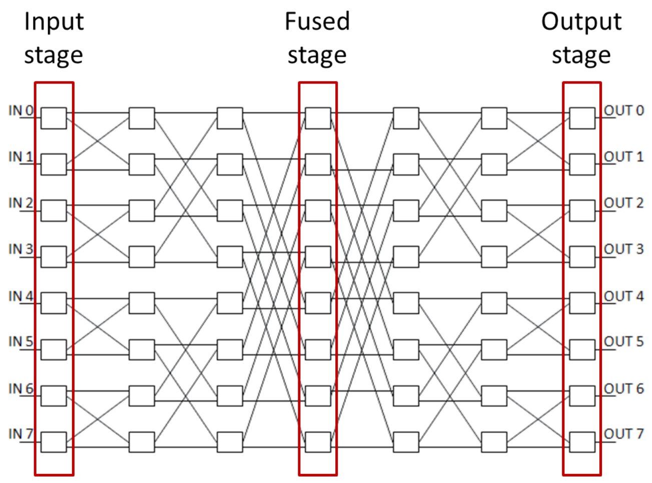


Figure 2.9: An 8-input Benes back-to-back butterfly like network

features

$$M = 2 * \log_2 N + 1 \tag{2.3}$$

stages, with N switch block per stage, resulting in a $O(N \log N)$ complexity, and is shown to be rearrangeably non-blocking for multicast connections [15]. In this particular architecture, in which an improved path diversity has been obtained increasing the number of resources available, latency represents the most critical point. For that, together with the fact that butterfly networks cannot exploit traffic locality [20], an alternative hierarchy-aware folded version has been devised and proposed during the research work.

Thanks to its symmetry properties, the *flat* MSSN (figure 2.9) can be folded at the central stage: this enhancement highlights an intrinsic hierarchical structure that can be exploited through a set of dedicated connections to form *"U-turns"*. As depicted in figure 2.10, a set of *U-turn* bypasses placed at each stage of the folded network allows each group H_i to define a non-blocking sub-network, still butterfly-based, thus bypassing the upper level of the hierarchy.

The enhancement provided by this kind of MSSN can easily improve routing performances, resulting in faster paths for local connections, without affecting the non-blocking properties ensured by the network topology. This is due to the fact that *U-turns* are realized by adding extra pins, dedicated logic and independent configuration bits to each switch module,

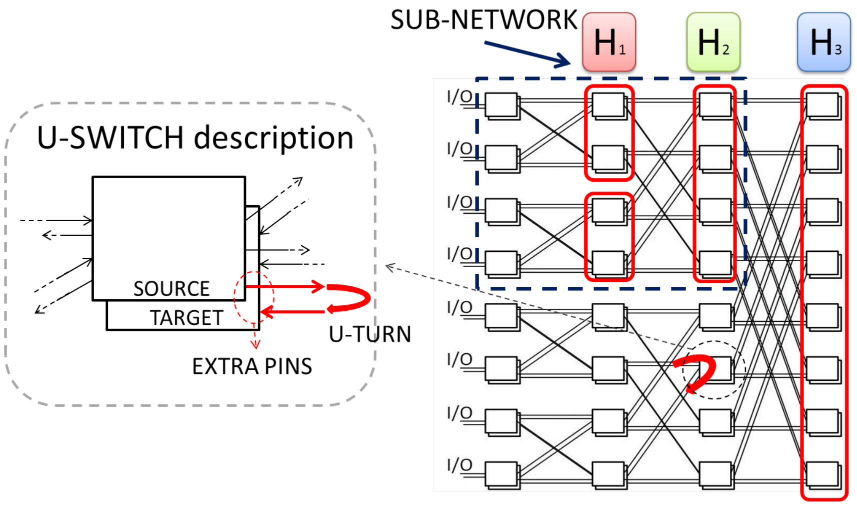


Figure 2.10: Folded MSSN supporting U-turn bypass

thus extending the number of available connections and routing paths of the original rearrangeably non-blocking MSSN. As shown in figure 2.11 using a flat view of the MSSN, *U-turns* act as bypasses, bridging non-adjacent stages thanks to the addition of an independent 2:1 multiplexer at the source switch and an extra input pin to each multiplexer on the target switch.

This area increase merits consideration: for that, the area-frequency trade-off on some bypass-enhanced MSSN architectures will be analyzed more in details in the final paragraph of this chapter.

Going back to FPGA scenario, this kind of MSSN can find its application on hierarchical designs, such as the hierarchical FPGA depicted in figure 1.5 (b): an example of correspondence of the hierarchical groups H_i on a FPGA 2D hierarchical layout is shown in figure 2.12.

2.2.2 Software Support

The software environment developed to support MSSN architectural exploration has been written completely using C language. The goal of such tool is to simulate the target topology in order to dump a register-transfer-level (RTL) MSSN description in Verilog language and to implement the structure on which the routing algorithm will then try to perform I/O connections. Given the complexity of the target architectures, in which the number of inputs and outputs - hence the complexity - can easily reach

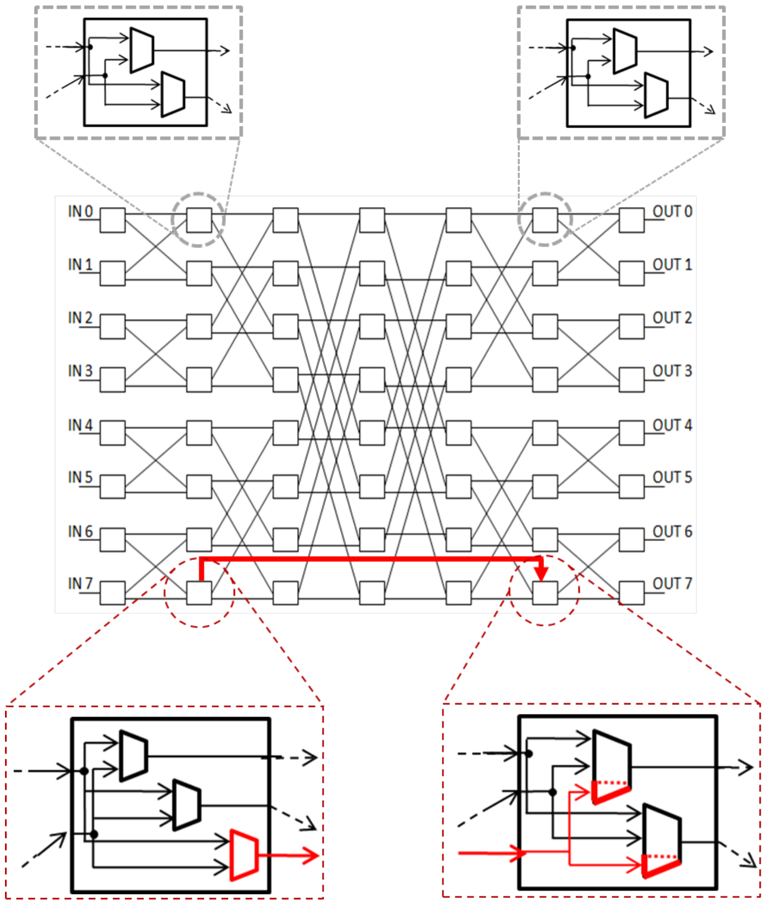


Figure 2.11: Bypass enhancement on a flat MSSN: U-Switch architectural overview

high values, the algorithm has been written exploiting the dynamic allocation of pointers, technique that has allowed to reduce memory usage and execution time.

Going into details, the tool exploits the modularity features of a MSSN. As a matter of fact, a network like the one depicted in figure 2.13 can be seen as based on programmable *switches* connected together through *wires*: these ”resources” are organized in columns (stages) and rows to define a *Resource Graph*. Thanks to the flexible structure of the tool, switches and wires of different stages can have different characteristics, thus allowing not only the simulation of a basic Benes butterfly-like MSSN (figure 2.9), but also any possible architectural variation in terms of intra-stage connectivity (e.g. *U-turn* enhancements of figure 2.11) or switch parameters (e.g creating non-homogeneous architectures with switches of different radix).

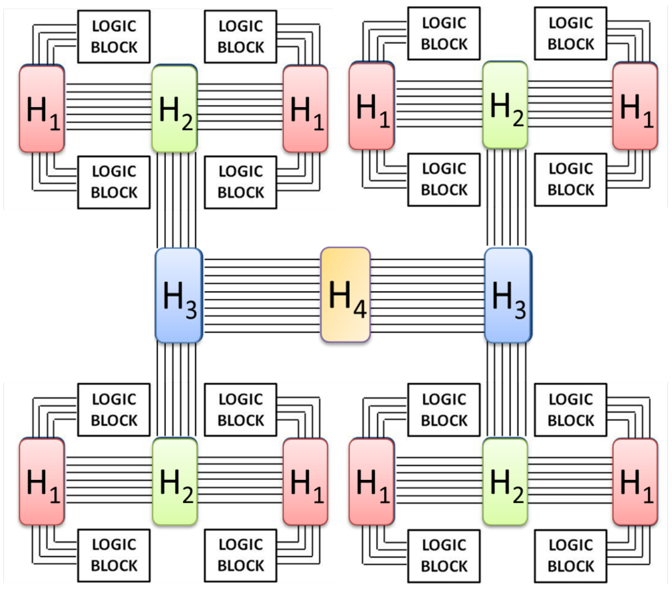


Figure 2.12: Hierarchy-aware 2D layout of butterfly MSSN

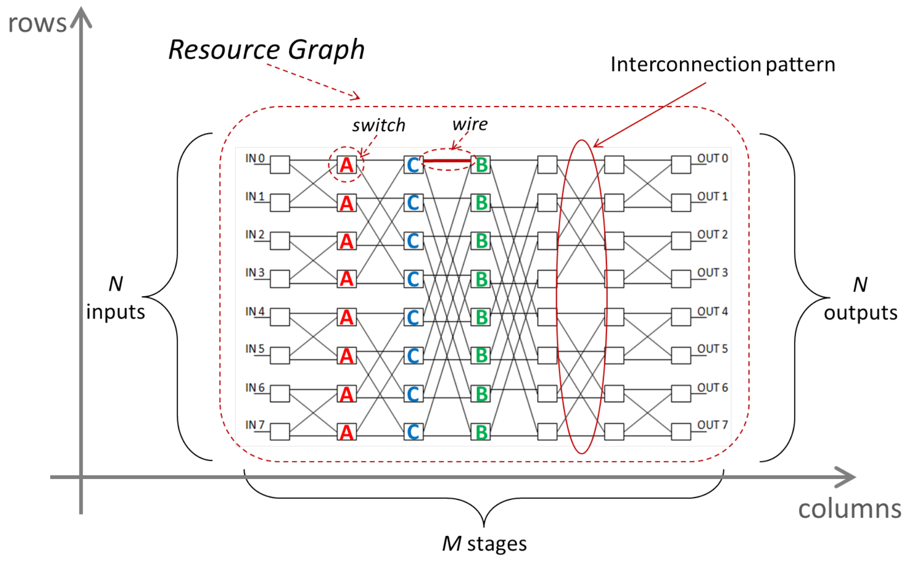
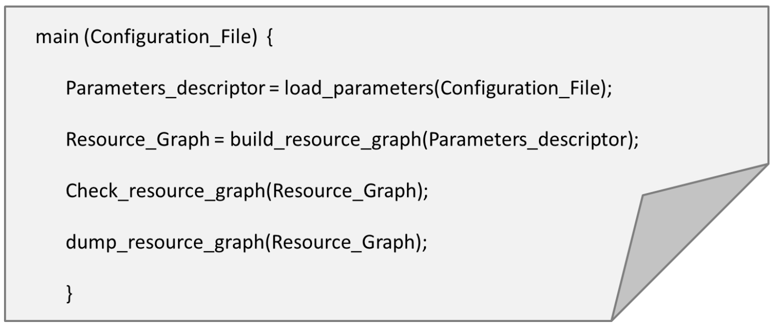


Figure 2.13: MSSN parameters for software tool

As a further added value, MSSN parameters are read from an external *Configuration File* so that architectural modifications can be performed without software changes.

The basic structure of the algorithm is described in figure 2.14: as first step a function *load_parameters* reads the description of the target MSSN features from a *Configuration File*, that contains (figure 2.13):



```
main (Configuration_File) {
    Parameters_descriptor = load_parameters(Configuration_File);
    Resource_Graph = build_resource_graph(Parameters_descriptor);
    Check_resource_graph(Resource_Graph);
    dump_resource_graph(Resource_Graph);
    }
```

Figure 2.14: Basic structure of the algorithm

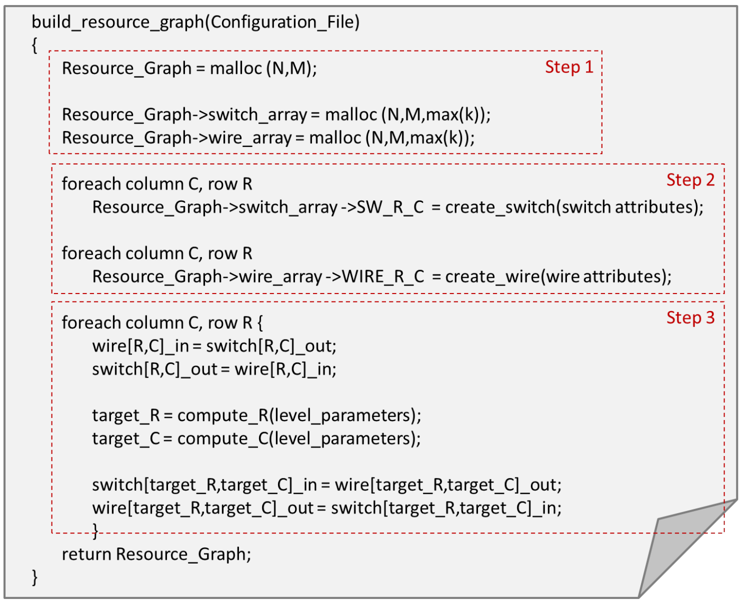
- Number of MSSN inputs and outputs (N);
- Number of MSSN stages (also called levels) (M);
- Number of switch *types*;
- Description of each *switch type* (e.g. A, B, C ..) ;
- Switch distribution: for each level a *switch type* is assigned;
- Connection configuration: wires interconnection pattern for each level.

Subsequently, thanks to the *build_resource_graph* function (figure 2.15), the algorithm creates and allocates a *Resource_Graph_struct* pointer with N inputs, N outputs and M levels (step1). In an homogeneous MSSN (i.e. all made of same radix-k switches) this values are strictly related so that N must be a power of k, and

$$M = 2 * \log_k N + 1 \tag{2.4}$$

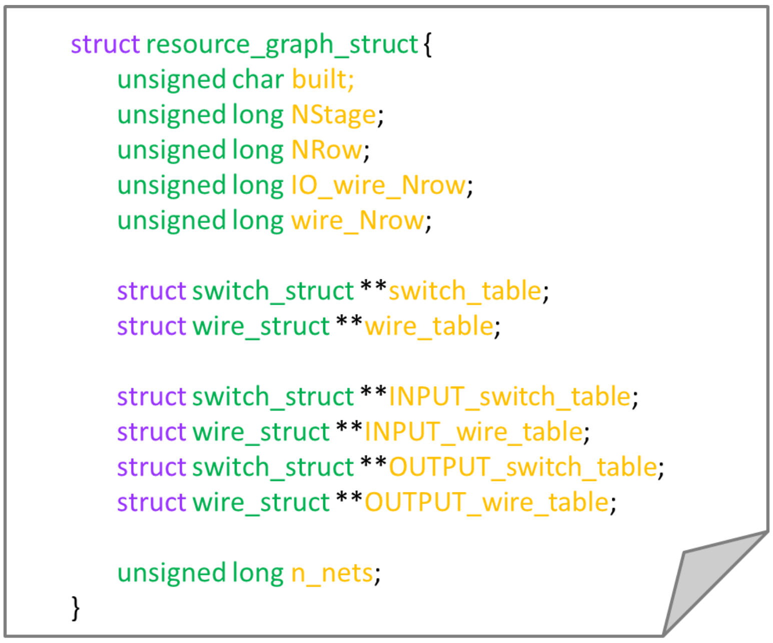
As specified before, the parametric structure of the tool also allows the creation of non-homogeneous networks in which switches featuring different radix can be associated to each level of the network; some example and a more detailed analysis of such architectures will be given at the end of this chapter.

Figure 2.16 shows in details the *Resource_Graph_struct*: it contains a set of parameters derived from the number of I/Os (i.e. number of rows, I/O wires, internal wires), and the arrays of *switch_structs* and *wire_structs* that will be used to build the network. Also the allocation of such arrays depends from N, M and the maximum value of k (figure 2.15-step1).



```
build_resource_graph(Configuration_File)
{
   Resource_Graph = malloc (N,M);                                   Step 1

   Resource_Graph->switch_array = malloc (N,M,max(k));
   Resource_Graph->wire_array = malloc (N,M,max(k));

   foreach column C, row R                                                  Step 2
       Resource_Graph->switch_array ->SW_R_C  = create_switch(switch attributes);

   foreach column C, row R
       Resource_Graph->wire_array ->WIRE_R_C  = create_wire(wire attributes);

   foreach column C, row R {                                                Step 3
       wire[R,C]_in = switch[R,C]_out;
       switch[R,C]_out = wire[R,C]_in;

       target_R = compute_R(level_parameters);
       target_C = compute_C(level_parameters);

       switch[target_R,target_C]_in = wire[target_R,target_C]_out;
       wire[target_R,target_C]_out = switch[target_R,target_C]_in;
   }
   return Resource_Graph;
}
```

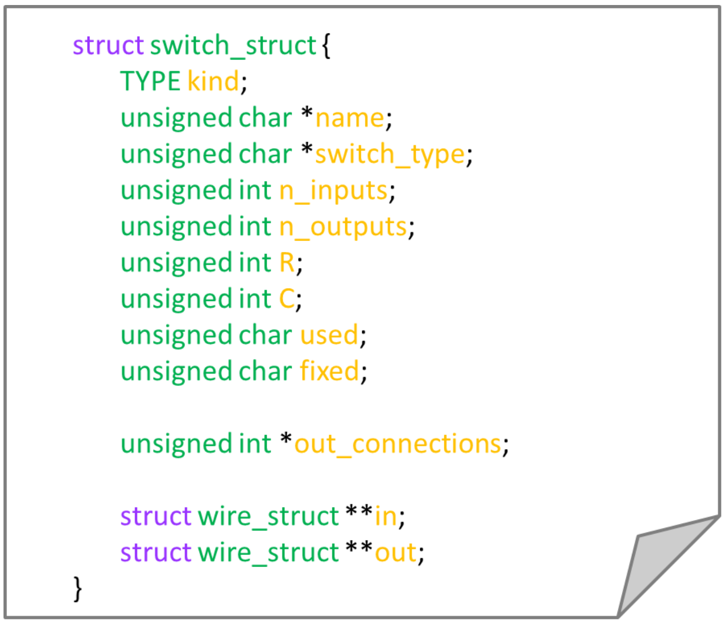
Figure 2.15: Pseudo-code of the function used to build the MSSN Resource Graph



```
struct resource_graph_struct {
    unsigned char built;
    unsigned long NStage;
    unsigned long NRow;
    unsigned long IO_wire_Nrow;
    unsigned long wire_Nrow;

    struct switch_struct **switch_table;
    struct wire_struct **wire_table;

    struct switch_struct **INPUT_switch_table;
    struct wire_struct **INPUT_wire_table;
    struct switch_struct **OUTPUT_switch_table;
    struct wire_struct **OUTPUT_wire_table;

    unsigned long n_nets;
}
```

Figure 2.16: C-language description of the Resource_Graph_struct

Switch_struct pointers are built using a *create_switch* function (figure 2.15-step2) with attributes that will be used both during architecture construction and routing algorithm. In this procedure, as shown in figure 2.17, to each n-inputs m-outputs switch is assigned a name (SW_row_col), that depends on the position in terms of row (R) and column (C) coordinates. A



```
struct switch_struct{
    TYPE kind;
    unsigned char *name;
    unsigned char *switch_type;
    unsigned int n_inputs;
    unsigned int n_outputs;
    unsigned int R;
    unsigned int C;
    unsigned char used;
    unsigned char fixed;

    unsigned int *out_connections;

    struct wire_struct **in;
    struct wire_struct **out;
}
```

Figure 2.17: C-language description of the Switch_struct

switch_struct also contains a boolean $m \times n$ connectivity matrix (*out_connections*), that defines the allowed internal connections. As an example, according to the matrix depicted in figure 2.18, out_0 can be connected only to in_1 (the only '1' on the first row), out_1 can be connected to all the inputs and out_2 only to in_2 and in_3. This matrix can be used during routing operation to simulate switch depopulation or to deviate traffic through specific routes. Other struct parameters are *fixed* or *used* attributes, that will be set during routing operations. Finally inside each $n \times m$ *switch_struct* pointer, to perform future connections, n input *wire_struct* pointers and m output *wire_struct* pointers are allocated (figure 2.17).

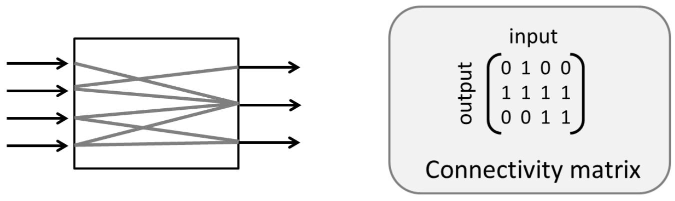
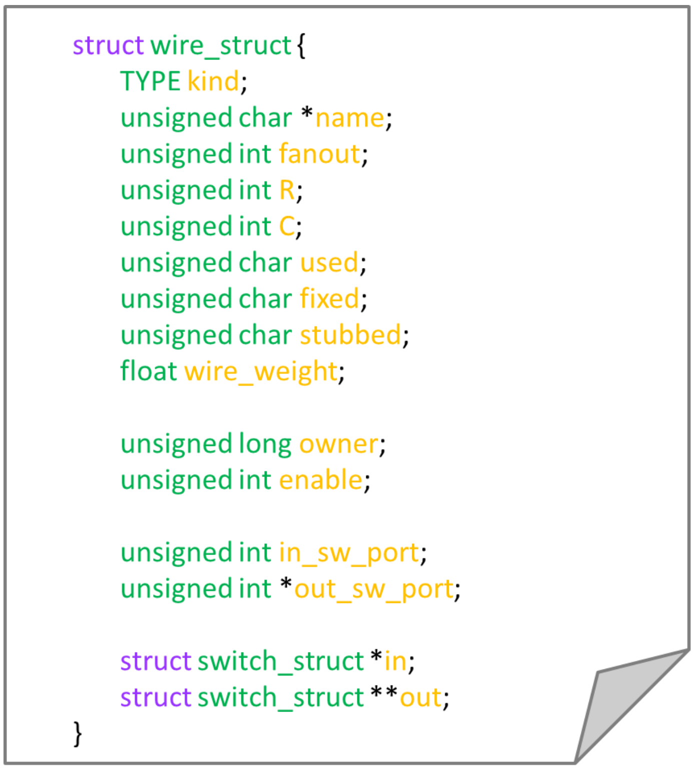


Figure 2.18: Example of internal connectivity on a 4x3 switch

As shown in figure 2.19, also *wire_struct* pointers (built using a *cre-*

ate_wire function - figure 2.15-step2) are characterized by *name*, row (*R*), column (*C*), *used* and *fixed* parameters; unlike *switch_structs*, a wire can have a weight - *wire_weight*, used to condition routing algorithm - an *owner* - the path that crosses - and a *fanout* parameter. This kind of struct features only 1 input *switch_struct* pointer and f (with $f = fanout$) output *switch_struct* pointers to realize connections from (or to) the output (or the input) port of a switch (*out_sw_port* and *in_sw_port*).



```
struct wire_struct {
      TYPE kind;
      unsigned char *name;
      unsigned int fanout;
      unsigned int R;
      unsigned int C;
      unsigned char used;
      unsigned char fixed;
      unsigned char stubbed;
      float wire_weight;

      unsigned long owner;
      unsigned int enable;

      unsigned int in_sw_port;
      unsigned int *out_sw_port;

      struct switch_struct *in;
      struct switch_struct **out;
}
```

Figure 2.19: C-language description of the Wire_struct

Once switch and wire pointers arrays have been created, for each level and for each row the algorithm performs connections by pointing each wire input to the corresponding switch output with a simple linear *for* cycle (figure 2.15-step3). The most elaborate step regards the connection between a wire output and its target switch input. Two kind of connections are supported: regular (butterfly-like) and irregular (e.g. U-turn bypasses). For each connection - hence for each wire output - two specific functions (*compute_R* and *compute_C*) calculate rows and columns coordinates (*target_R* and *target_C*) of the target switch, essentially using two formulas:

$$\begin{cases} Row = \beta \times \left[(y + v_j \times span)\,\%\alpha + (int)\frac{y}{\alpha} \times \alpha \right] + (1-\beta) \times (y + v_j) \\ Column = x + h_j - 1 \end{cases} \tag{2.5}$$

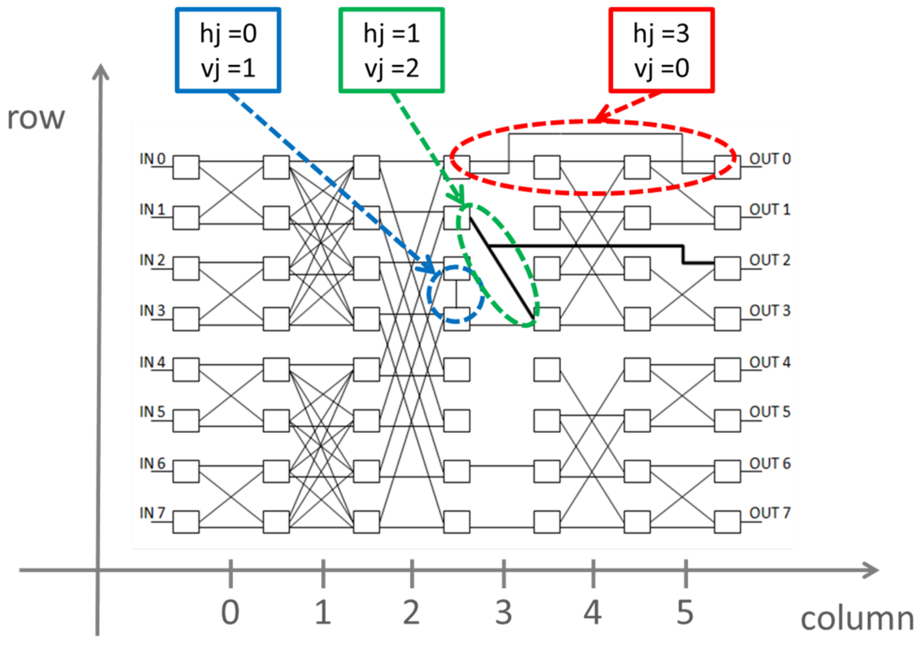
where (see figure 2.20)

- β : boolean variable, = 1 for regular connections, = 0 for irregular connections;
- x, y : starting row and column coordinates;
- v_j, h_j : vertical and horizontal offset;
- $span$: value for cross (not straight) connections, used to build a butterfly-like structure ($span$ values shall be a power of 2 of a power of 4, depending on the architecture required);
- α : = $span \times base$, where $base$ can be 2 or 4, used for regular connections.

are the parameters associated to each MSSN level, specified in the *Configuration File*; in this operation also the *wire_weight* parameter - that will be used to steer routing operations - is also assigned to each wire. Figure 2.20 shows several examples of different possible configurations, including an example of multiple fanout wire ($f = 2$, bold wire of level 3).

As next step (figure 2.14) a *check_resource_graph* function is used to control the internal connectivity (figure 2.21): to do so switches (step1) and wires (step2) inputs and outputs are scanned, checking if the port is floating (i.e. $= NULL$) or connected. After a successful result of the *check_resource_graph* function, the verilog RTL description of the MSSN can be dumped (*dump_resource_graph*). Besides the software solution, another methodology allows checking the architecture correctness: thanks to the utilization of a graphic tool (GraphViz) it is possible to create an image of the MSSN starting from a *.dot* file, this also dumped by a C-function. Figures 2.22 and 2.23 show two examples of graphs obtained with such a technique.

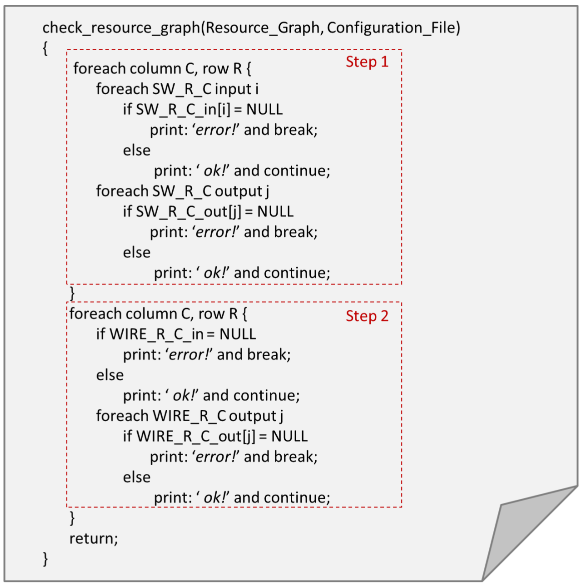
As anticipated before the tool is also able to build architectures featuring mix of regular heterogeneous levels, hence made of switches with different radix. Figure 2.22 shows an example of 16-inputs+16-outputs MSSN



- LEVEL 0 :
  Regular level, base 2, span = 1
- LEVEL 1 :
  Regular level, base 4, span = 1
- LEVEL 2 :
  Regular level, base 2, span = 4
- LEVEL 3 :
  Irregular level
- LEVEL 4 :
  Regular level, base 2, span = 2

Figure 2.20: Example of different connection configurations

made with a mix of radix-2/radix-4 levels: in these networks the number of levels of the first half and their radix are obtained with the factorization of the number of I/Os (e.g. 16 = 2 x 4 x 2). The mixed-radix solutions can be also implemented including non-power-of-two elements, provided they are prime numbers, increasing the flexibility in terms of number of inputs and outputs. As examples of this kind of architectures, figure 2.23 and figure 2.24 show two mixed-radix MSSN, featuring respectively 12-inputs+12-outputs (12 = 2 x 3 x 2) and 20-inputs+20-outputs (20 = 2 x 5 x 2). In the final paragraph of this chapter two MSSN featuring regular architectures and switches with different radix (2x and 4x respectively) will be analyzed and compared, with the aim to determine pros and cons of both choices.



```
check_resource_graph(Resource_Graph, Configuration_File)
{
    foreach column C, row R {                              Step 1
        foreach SW_R_C input i
            if SW_R_C_in[i] = NULL
                print: 'error!' and break;
            else
                print: ' ok!' and continue;
        foreach SW_R_C output j
            if SW_R_C_out[j] = NULL
                print: 'error!' and break;
            else
                print: ' ok!' and continue;
    }
    foreach column C, row R {                              Step 2
        if WIRE_R_C_in = NULL
            print: 'error!' and break;
        else
            print: ' ok!' and continue;
        foreach WIRE_R_C output j
            if WIRE_R_C_out[j] = NULL
                print: 'error!' and break;
            else
                print: ' ok!' and continue;
    }
    return;
}
```

Figure 2.21: Pseudo-code of the function used to check MSSN Resource Graph connectivity

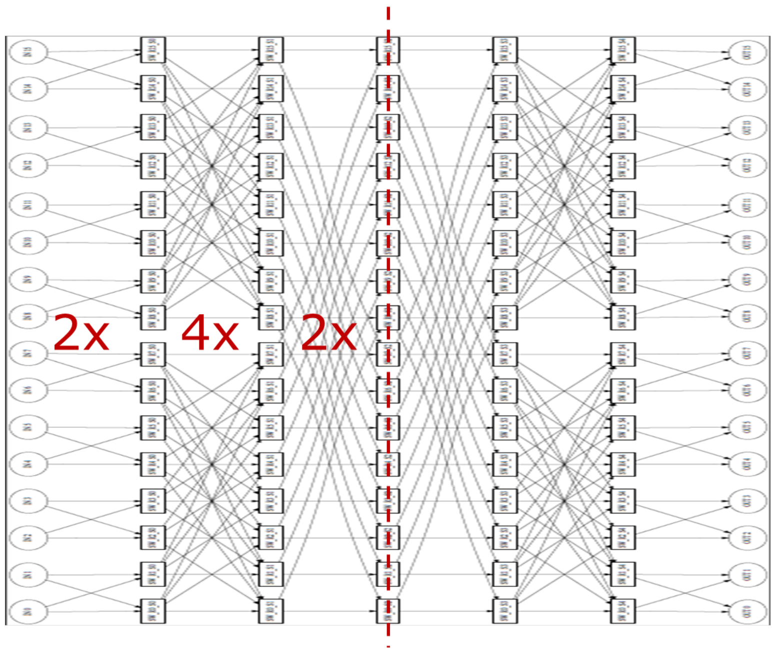


Figure 2.22: A 16-inputs/16-outputs mixed radix MSSN

2.3 MSSN Routing

Routing involves selecting a path from a source node (input) to a destination node (output): in MSSNs, routing choices heavily influence the perfor-

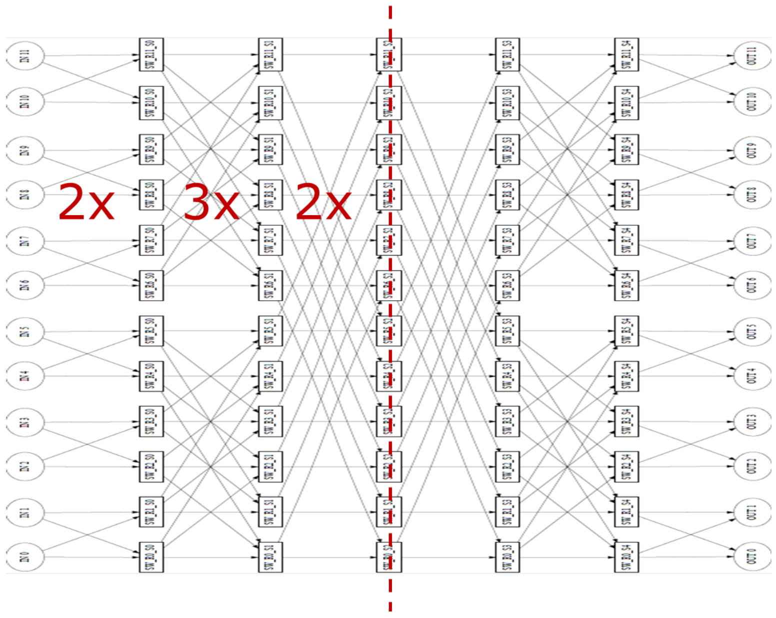


Figure 2.23: A 12-inputs/12-outputs mixed radix MSSN

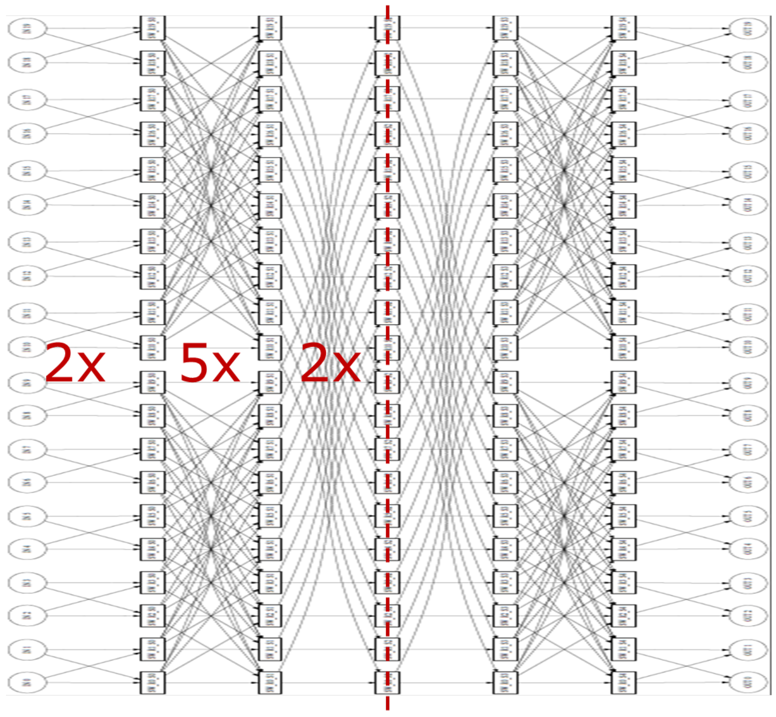


Figure 2.24: A 20-inputs/20-outputs mixed radix MSSN

mance of the network. As a matter of fact, whereas the topology determines the ideal performance of a network, routing can determine how much of this potential is realized [15]. In order to better analyze and explain the chosen routing technique, two different aspects have been analyzed: the routing strategy, which controls the operations, and the routing algorithm, the core function with which connections are actually performed.

2.3.1 Routing Strategy

Figure 2.25 shows the summary of the adopted routing strategy. Given a MSSN topology, hence a *Resource Graph*, the list of the connections (*nets*) to be realized is contained in a *Netlist File*, where each net is defined by 4 parameters: *name*, used to identify each connection, *source*, *target*, input and output to be connected, and *net_weight*, a parameter used to determine routing order. Multicast connections are handled so that multi-fanout nets are split into a set of single-fanout nets that will be routed independently.

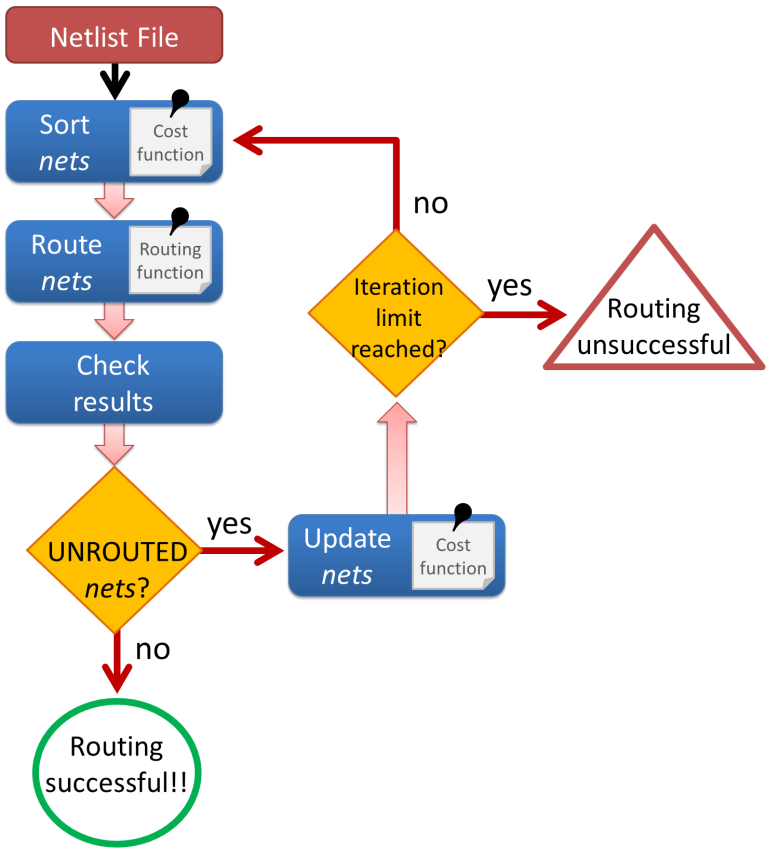


Figure 2.25: Routing strategy structure

The first step is therefore a sorting operation of the nets to be routed:

owing to rearrangeably non-blocking characteristic of the network, this step can heavily affect interconnect performance. Going into details, nets are sorted in order of cost (*Net_cost*), a parameter calculated - and updated - as:

$$Net\_cost = Path\_cost + \gamma * (net\_weight \times criticality) \tag{2.6}$$

where γ and *net_weight* are constants defined by the user, *criticality* is a parameter initially set to 1 and *Path_cost* is the sum of the *wire_weights* of the crossed wires (see section 2.3.2).

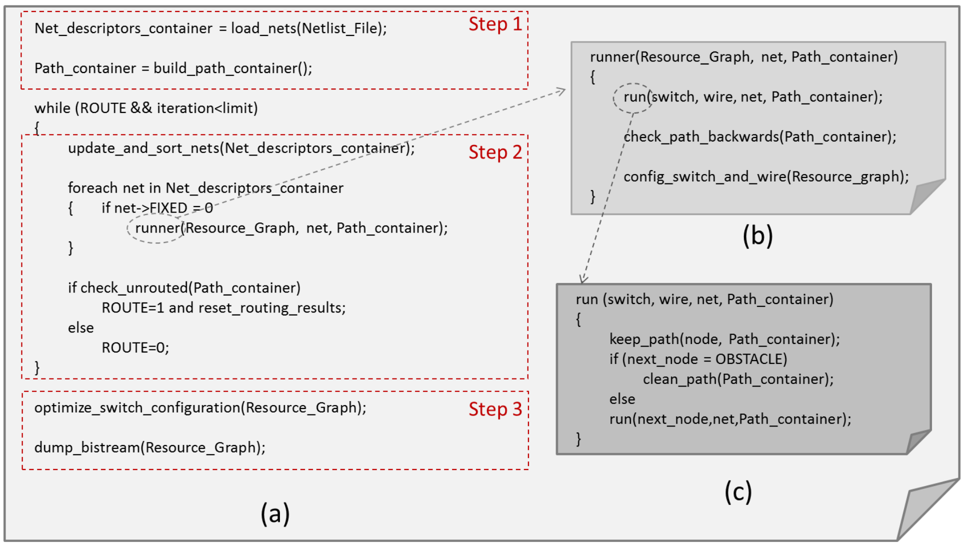
Once the nets have been sorted, a routing function tries to realize all the connections in the *Netlist*, resulting in successful or unsuccessful attempts. At the end of the routing function, hence of the *Netlist*, a function checks routing results: if all the nets have been routed, the goal has been achieved. Otherwise, if there are *unrouted* nets, their cost is updated in order to put the most critical connection at the top of the netlist during next sorting operation. To do so, in 2.6 their *criticality* parameter is increased by 1, and the obtained routing results are removed with a reset operation. The process goes on until there are *unrouted* nets or the number of iterations reaches an *iteration limit*, a parameter set by the user. Next section will describe in details routing step, the core algorithm of this process.

2.3.2 Routing Algorithm

The *Resource Graph* that represents the MSSN can be seen as a weighted graph, since during building operation (figure 2.15) a weight (*wire_weight*) is associated to each wire: exploiting this feature, the routing algorithm has been developed as a Pathfinder-like algorithm based on Dijkstra's algorithm for weighted graphs [21]. As for the code developed to support MSSN architecture generation, the C-based routing algorithm has been realized using dynamic pointers allocation.

Each net listed on the *Netlist File* is at first (step1 in figure 2.26) memorized in a *Net_descriptor_struct* pointer, that contains:

- *Net_name* - used to identify each connection;
- *Source, target* - input and output to be connected;
- *Net_weight, criticality, net_cost* - parameters used to determine routing



```
Net_descriptors_container = load_nets(Netlist_File);

Path_container = build_path_container();

while (ROUTE && iteration<limit)
{
        update_and_sort_nets(Net_descriptors_container);

        foreach net in Net_descriptors_container
        {       if net->FIXED = 0
                        runner(Resource_Graph, net, Path_container);
        }

        if check_unrouted(Path_container)
                ROUTE=1 and reset_routing_results;
        else
                ROUTE=0;
}
optimize_switch_configuration(Resource_Graph);

dump_bistream(Resource_Graph);
```

```
runner(Resource_Graph, net, Path_container)
{
        run(switch, wire, net, Path_container);

        check_path_backwards(Path_container);

        config_switch_and_wire(Resource_graph);
}
```

```
run (switch, wire, net, Path_container)
{
        keep_path(node, Path_container);
        if (next_node = OBSTACLE)
                clean_path(Path_container);
        else
                run(next_node,net,Path_container);
}
```

Figure 2.26: Pseudo-code of the routing algorithm (a) and its internal functions (b)(c)

order (equation 2.6);

- *Route* - the condition of a net: UNROUTED (connection not realized, initial condition), ROUTED (connection realized) or FIXED (routed and not-to-be-removed).

and, since the performed connections (*Paths*) will be memorized as a list of resources, a *Path_container_struct* pointer is allocated.

The second step (step2) of the algorithm represents the core of the process: following the routing strategy, nets are sorted according to their updated cost (*update_and_sort_nets*) and for each of them a *runner* function tries to realize the connections. To do so, starting from the input wire corresponding to the *source* of a net, *runner* algorithm (figure 2.26 (b)) tries to reach the requested output (the *target* of a net) running on resources (switches and wires) with a recursive function (*run* function in figure 2.26 (c)) and taking all possible ways.

During this process a *keep_path* function tracks the crossed resources, building a list of switch or wire pointers (figure 2.27 (a)): since potentially there is more than a way to connect a source to a target, as shown in figure 2.27 (b) this list can be seen as a tree with an head (the starting input), branches (the connections) and leafs (wires and switches).

Runner also has to manage obstacles during its route, such as:

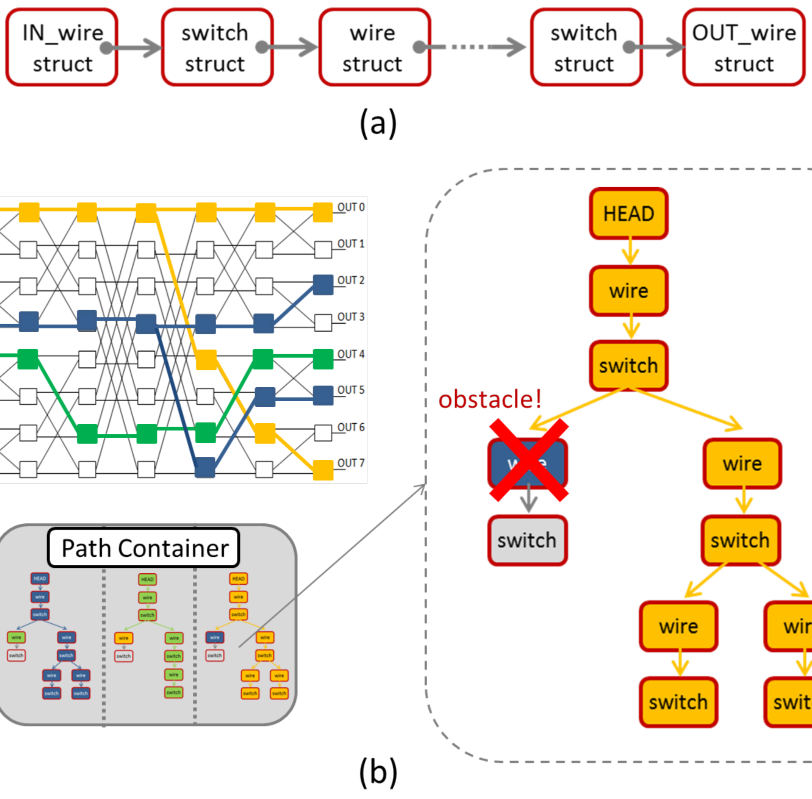


Figure 2.27: Routing paths represented as list of pointers (a) organized as binary tree in a Path Container

- resources used by another driver: wires belonging to a connection coming from a different source (the *owner* parameter described in section 2.2.2);
- disabled wires: a function can disable, and enable, any wire inside the network;
- wrong outputs: output wires not corresponding to the target.

When an obstacle is reached and runner can't walk anymore on that way, the corresponding branch is interrupted and immediately erased (*clean_path* function) from the *Path_container*: such real-time deallocation of pointers allows considerable memory saving.

While "'running"' the cost of a path is calculated according to the *wire_weight* assigned (by the user, in the *Configuration File*) to each wire, so that:

$$Path\_cost = \sum Wire\_weight \tag{2.7}$$

In order to improve the convergence toward a final solution *wire weights* are assigned to guide the algorithm to follow an *as-straight-as-possible* policy: in particular, in a regular butterfly-like MSSN, bypasses and straight paths are cheaper than diagonal paths. Once the target has been reached the first time, a reference variable (*MIN_COST*) is updated with *Path_Cost* value and the cost becomes a virtual ”obstacle”: when looking for other ways, if

$$Current\_cost > \sum MIN\_COST \tag{2.8}$$

current path will certainly be more expensive than the one with minimum cost and runner will not proceed any more on that way. As for the other obstacles, the pointers corresponding to the discarded branch are immediately deallocated.

After a path has been chosen, a function checks the correctness of the route walking backwards from the target to the source (*check_path_backwards* in figure 2.26 (b)) so that only one route can be taken, and the used resources are marked as *used* by an *owner* (the name of the net) so that they will not be available for path coming from different sources (*config_switch_and_wire* in figure 2.26 (b)).

For each net the algorithm stops when all possible ways have been traveled: the corresponding net is set as UNROUTED or ROUTED depending on the result. An optional function also allows the user to set ROUTED nets as FIXED, so as they will not be removed from the *Path_container* in case of another routing iteration. At the end of the netlist, a *check_unrouted* function determines whether the iteration has to stop or not.

The final step (step3) of the routing process involves an optimization of the network configuration: since the MSSN has been designed to realize the internal connection in an embedded programmable device, the unused MSSN resources (switches and wires) should be configured so as to reduce as much as possible the MSSN power consumption. To do so, thanks to the *optimize_switch_configuration* function, internal switches are configured so as to connect unused outputs (not contained in the *Netlist File*) to unused inputs: in this way the potential switching activity of the MSSN can be reduced, thanks to the propagation of constant (coming from unused inputs) values. Finally, a *dump_bitstream* function scans all the MSSN resources and dumps the configuration bitstream.

2.4 MSSN Architectural Analysis

This paragraph provides an evaluation of the *U-turn* bypasses, with the aim to find the best trade-off between the area penalties - introduced by the addition of logic in the *U-switches* (figure 2.11) - and the advantages on frequency performances.

To do so, a distinction has to be made between implementation frequency and effective working frequency. The first one is related to the full-latency of the MSSN; the second one takes into account bypasses, which can provide faster paths for near I/O points, and thus depends on the locality of the required connections. Since different I/Os connections configurations imply different levels of bypass exploitations - hence different advantages on the effective frequency performances - in this analysis 4 ranges of frequency gain have been taken into consideration: a 0% gain - corresponding to a non-exploitation of any of the bypass connections - a 20%, 40% and 60% frequency gain. Such values are in-line with reality, since taken from some application-aware analysis (such as the one provided in paragraph 4.4).

With these premises, the *U-turn* bypasses evaluation has been performed on two different 1024+1024 I/Os MSSNs, featuring architectures with 2-inputs/2-outputs switches (*radix-2*) and 4-inputs/4-outputs switches (*radix-4*) respectively. For each one, a set of synthesis trials were performed using Synopsys Design Compiler Graphical tool [22] with STMicroelectronics CMOS 65mn LP 1.10V standard cells libraries. Since CMOS 65nm technology features standard cells libraries characterized by transistors with 3 different MOSFET threshold voltage V_t (High, Standard and Low Voltage Threshold - HVT, SVT, LVT), two different mix of standard cells were taken into consideration: a standard area-optimized SVT-only and an high-speed HVT+SVT+LVT mix. Target implementation frequency was the *time-to-fly* between a primary input and a primary output, since the configuration pins of the switch blocks were excluded from the timing analysis though the *set_false_path* command.

Finally, a comparison of the two topologies is provided, in order to determine the best MSSN *radix*.

2.4.1 Radix-2 1024+1024 MSSN

The first architecture is hence a 1024+1024 I/Os MSSN featuring 2-inputs/2-outputs (2x2) switches. The structure of a flat version of such network is exactly like the one reported in figure 2.9, and according to equation 2.3 it features 21 stages, of which:

- the input stage is composed of 1024 1-input/2-output (1x2) switches;
- the middle stages features 1024 2x2 switches;
- the output stage has 1024 2x1 switches

Starting from a flat version, the *U-turn* bypasses were applied so as to realize two different MSSNs: a fully-bypassed one, with bypasses on each MSSN stage, and an half-bypassed network, with bypasses only on odd stages. Post-synthesis results for all three versions are shown in tables 2.1 and 2.2, with two different implementations optimized per area or per speed.

Min Area	Flat MSSN	Half-bypassed	Fully-bypassed
Area $[mm^2]$	0.20	0.34	0.40
Impl. Frequency $[MHz]$	200	200	180
Cells Mix	SVT		

Table 2.1: CMOS 65nm 1024+1024 radix-2 MSSN post-synthesis summary - Min Area

Max Speed	Flat MSSN	Half-bypassed	Fully-bypassed
Area $[mm^2]$	0.80	0.89	1.01
Impl. Frequency $[MHz]$	480	445	395
Cells Mix	HVTSVTLVT		

Table 2.2: CMOS 65nm 1024+1024 radix-2 MSSN post-synthesis summary - Max Speed

Each MSSN architecture was then implemented varying the target frequency between the min-area and the max-speed values.

The obtained results are reported for the iso-area values in figure 2.28, with the implementation frequency on the vertical axis and the obtained area on the horizontal axis, showing similar trends for all three architectures; the switch between the SVT-only and the HVT-SVT-LVT implementations is highlighted with the orange line.

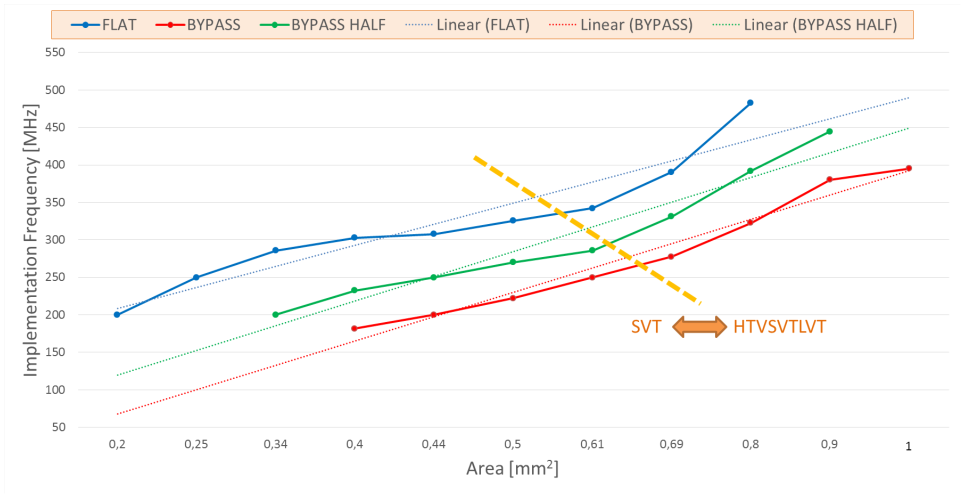


Figure 2.28: Implementation frequency versus area - 1024+1024 radix-2 MSSN post-synthesis results

Moreover, in order to better analyze the area-frequency trade-off, a computational density analysis of both the half-bypassed and the fully-bypassed architecture was performed, taking into account designs with same area and different implementation frequency: as anticipated, 4 different levels of bypass exploitation were assumed (i.e. 0%, 20%, 40% and 60% frequency gain achievable).

The results for the half-bypassed architecture analysis are reported in figure 2.29, showing that for low area budget ($< 0.34\ mm^2$) the flat architecture is able to guarantee operating frequencies higher than the half-bypassed one with a frequency gain of 40%. The half-bypassed architecture becomes advantageous for area budgets $> 0.44\ mm^2$, since even with a minimum frequency gain ($> 20\%$) the operating frequencies achievable are higher than that of the flat MSSN, allowing one to attain values beyond 700 MHz when area occupancy is not a constraint.

Figure 2.30 shows the same analysis relative to the fully-bypassed architecture: in this configuration the bypass-enhancements become advantageous only when high-frequency (> 450 MHz) performances are required, while for area budgets $< 0.8\ mm^2$ the flat MSSN is able to guarantee better speed performances. Compared to the half-bypassed solution, this architecture allows reaching lower frequencies, with values that do not exceed 650 MHz.

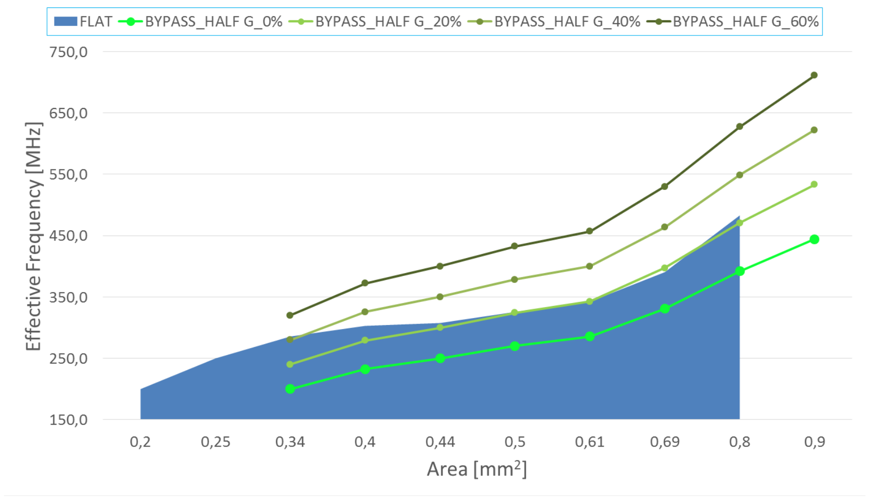


Figure 2.29: 1024+1024 radix-2 Flat MSSN versus Half-bypassed: effective frequency vs. area

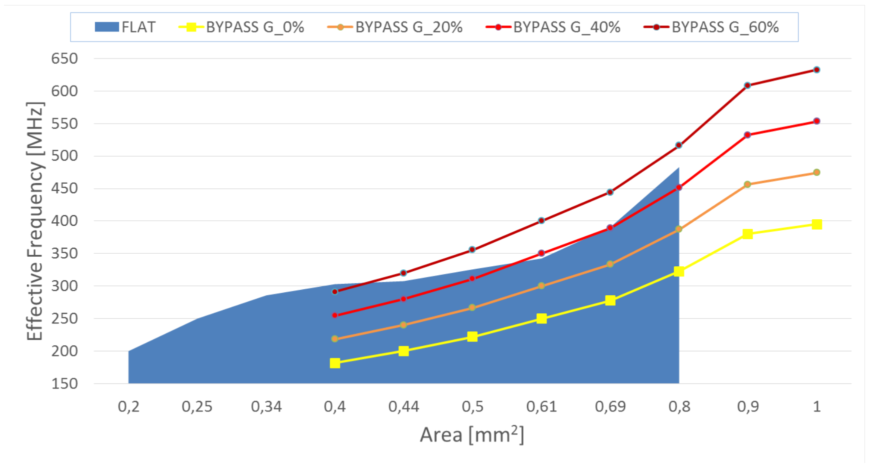


Figure 2.30: 1024+1024 radix-2 Flat MSSN versus Fully-bypassed: effective frequency vs. area

Computational efficiency has been analyzed in figure 2.31 for the half-bypassed and figure 2.32 for the fully-bypassed: the frequency/area ratio was taken as a figure of merit, which is proportional to the throughput/area ratio.

Figures 2.31 and 2.32 show that in all three MSSN styles frequency/area ratio has its maximum value for low area implementations, with a decreasing trend that tends to saturate for area values $> 0.6\ mm^2$: this means that, even when forcing the implementation frequency, the computational density values follow a predictable trend.

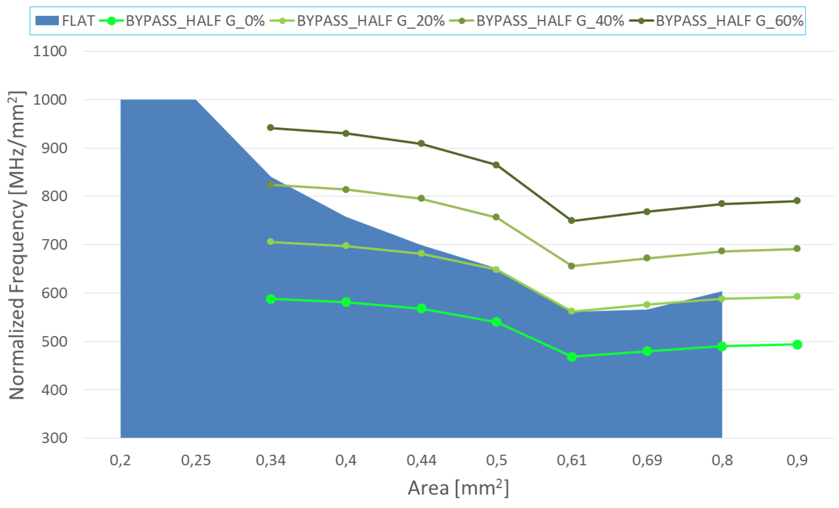


Figure 2.31: 1024+1024 radix-2 Flat MSSN versus Half-bypassed: computational density vs. area

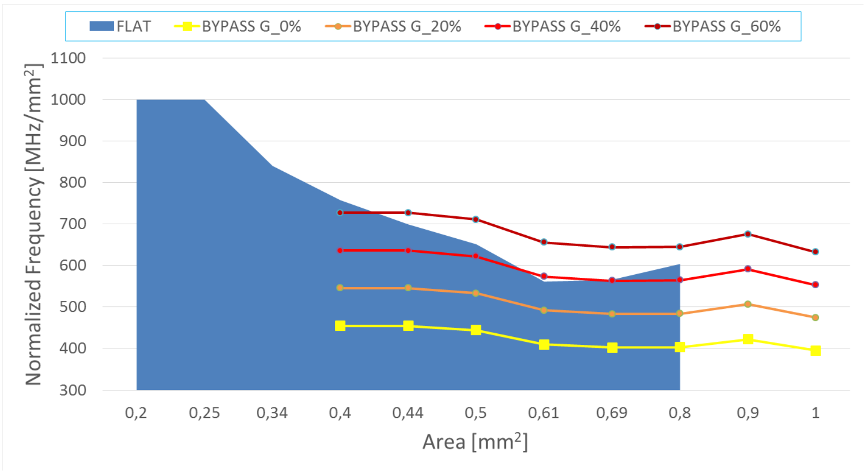


Figure 2.32: 1024+1024 radix-2 Flat MSSN versus Fully-bypassed: computational density vs. area

2.4.2 Radix-4 1024+1024 MSSN

Thanks to the parametric structure of the MSSN tool it was possible to realize a 1024+1024 MSSN featuring 4-inputs/4-outputs switches (4x4). Following the MSSN building rules described in paragraph 2.2.2, and according to equation 2.4, the number of stages M of a radix-4 MSSN with N inputs/outputs is:

$$M = 2 * \log_4 N + 1 \tag{2.9}$$

resulting in a 11 stages MSSN. Moreover, since during the network doubling operation (figure 2.8) the input pins of the switches of the input stage are halved - in this case resulting in 2-inputs switches - each stage is composed of 512 rows. Hence the MSSN features:

- the input stage composed of 512 2-input/4-output (2x4) switches;
- the middle stages featuring 512 4x4 switches;
- the output stage with 512 4x2 switches.

as shown in figure 2.33, where an example of 16+16 radix-4 MSSN is shown.

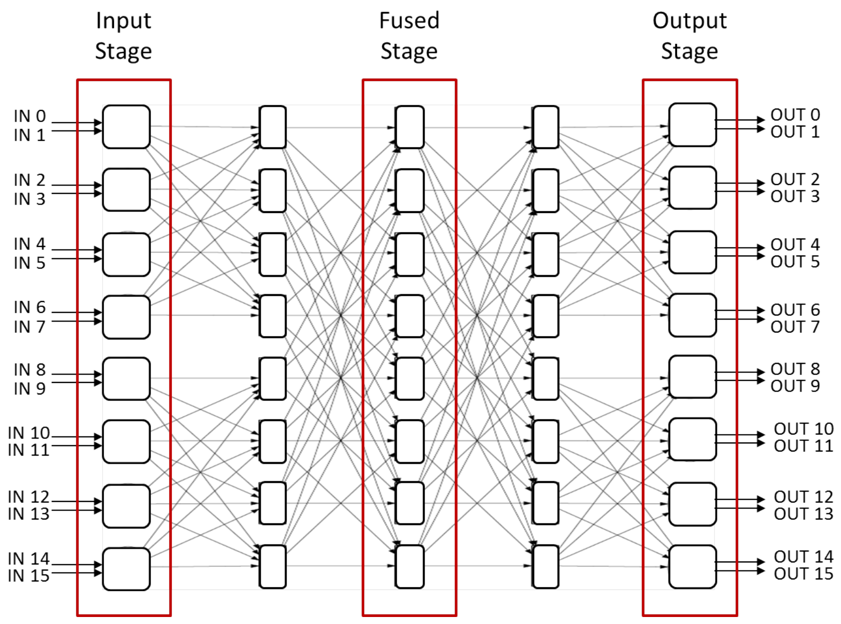


Figure 2.33: A 16+16 I/Os Benes butterfly-like network featuring *radix-4* switches

As for the previous radix-2 MSSN (section 2.4.1), three different architectures were implemented: a flat MSSN, a fully-bypassed and an half-bypassed (with *U-turn* bypasses only on odd stages). Post-synthesis results are shown in tables 2.3 and 2.4, with two different implementations optimized per speed or per area

Again, a set of synthesis trials were performed varying the implementation frequency between the values in tables 2.3 and 2.4, obtaining the results reported in figure 2.34, where the switch between the SVT-only and

Min Area	Flat MSSN	Half-bypassed	Fully-bypassed
Area $[mm^2]$	0.27	0.33	0.49
Frequency $[MHz]$	200	200	200
Cells Mix	SVT		

Table 2.3: CMOS 65nm 1024+1024 radix-4 MSSN post-synthesis summary - Min Area

Max Speed	Flat MSSN	Half-bypassed	Fully-bypassed
Area $[mm^2]$	0.89	0.90	1.1
Frequency $[MHz]$	405	365	367
Cells Mix	HVTSVTLVT		

Table 2.4: CMOS 65nm 1024+1024 radix-4 MSSN post-synthesis summary - Max Speed

the HVT-SVT-LVT implementations is highlighted with the orange line. As for the radix-2 implementations, the three MSSN versions follow the same growing trend; as can be observed, the fully-bypassed architecture (red line) can reach lowest frequencies, with a gap from the other two topologies deeper than the radix-2 case.

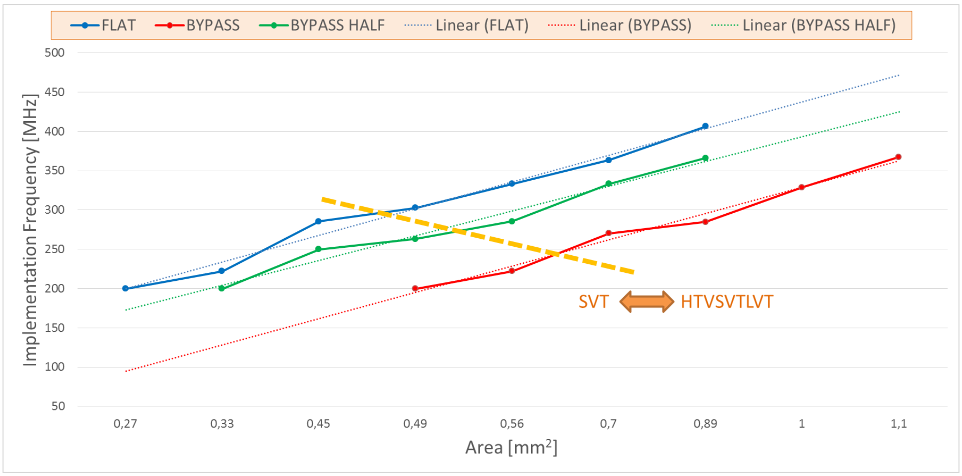


Figure 2.34: Implementation frequency versus area - 1024+1024 radix-4 MSSN post-synthesis results

The computational density analysis of both fully- and half-bypassed architectures was again performed considering 4 levels of bypass exploitation (i.e. able to guarantee a frequency gain of 0%, 20%, 40% and 60% respectively). The results of the half-bypassed architecture (figure 2.35) show that for configurations able to guarantee a frequency gain $> 20\%$ this structure is always more advantageous than the flat one.

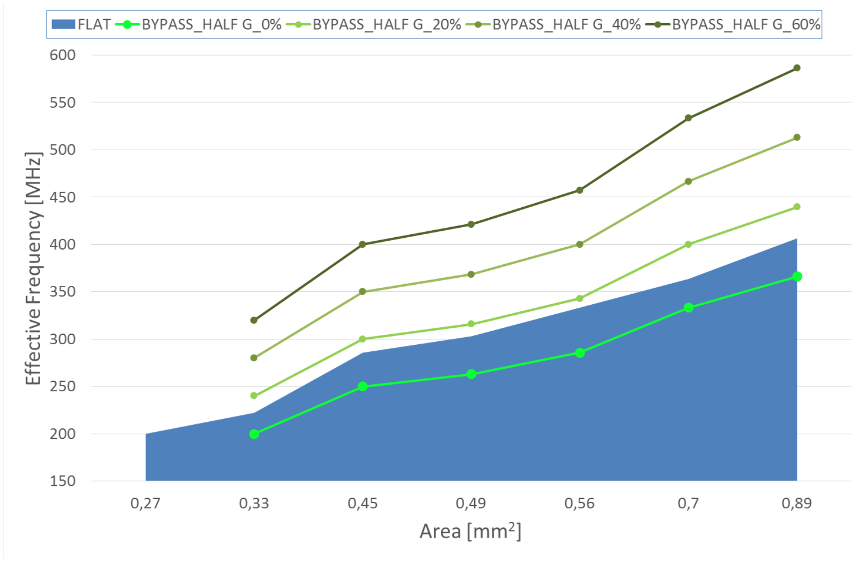


Figure 2.35: 1024+1024 radix-4 Flat MSSN versus Half-bypassed: effective frequency vs. area

Figure 2.36 shows the effective frequency analysis for the fully-bypassed MSSN: in this case the advantages provided by the introduction of the *U-turn* bypasses become evident only when high frequency ($>$ 400 MHz) performances are required at the expense of an increased area occupancy. For low area budgets ($<$ 0.89 mm^2), indeed, the flat topology can guarantee frequencies comparable to that of a fully-bypassed MSSN with a 40% frequency gain.

Computational efficiency has been analyzed in figure 2.37 for the half-bypassed and figure 2.38 for the fully-bypassed: the frequency/area ratio was taken as a figure of merit, which is proportional to the throughput/area ratio.

Figures 2.37 and 2.38 show that in all three MSSN styles frequency/area ratio has its maximum value for low area implementations, with a decreasing trend that tends to saturate for area values $>$ 0.89 mm^2: this means that, also in the radix-4 case, even when forcing the implementation frequency the computational density values follow a predictable trend.

Next paragraph will provide a comparison between the radix-2 and the radix-4 architectures, both in terms of area and ability to exploit the advantages provided by the *U-turn* bypasses.

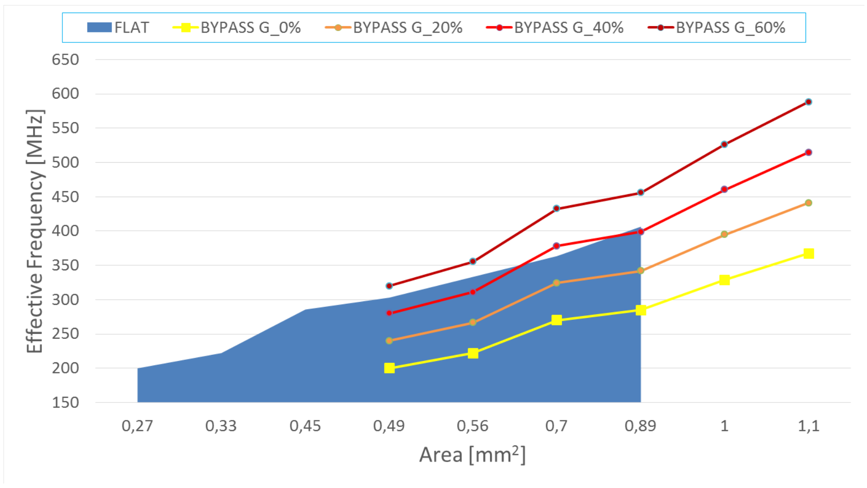


Figure 2.36: 1024+1024 radix-4 Flat MSSN versus Fully-bypassed: effective frequency vs. area

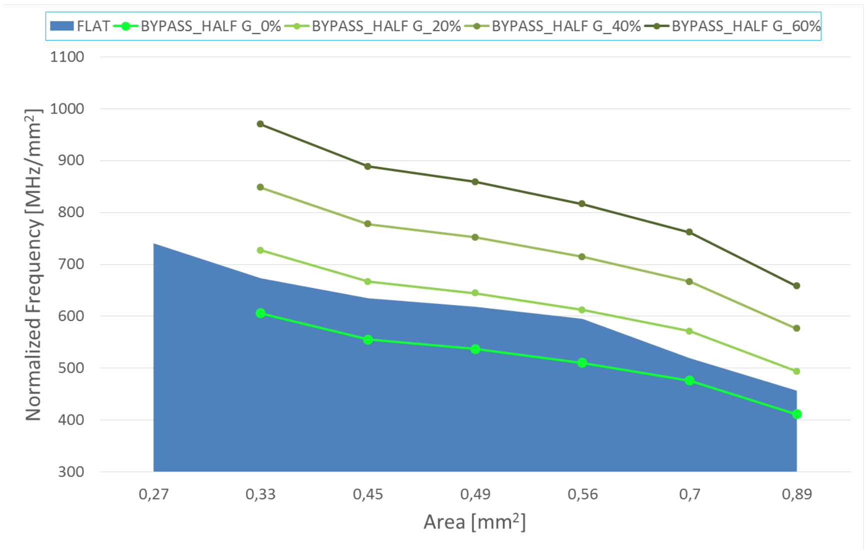


Figure 2.37: 1024+1024 radix-4 Flat MSSN versus Half-bypassed: computational density vs. area

2.4.3 Comparison between radix-2 and radix-4 architectures

The first aspect that has to be taken into consideration is related to the area occupancy of the two different topologies: to do so, an evaluation of the complexity of each architecture - in its flat version - in terms of equivalent MUX2:1 has been performed.

In an homogeneous (i.e. stages realized with the same switches) M stages MSSN, with K switches per stage, the total number of equivalent

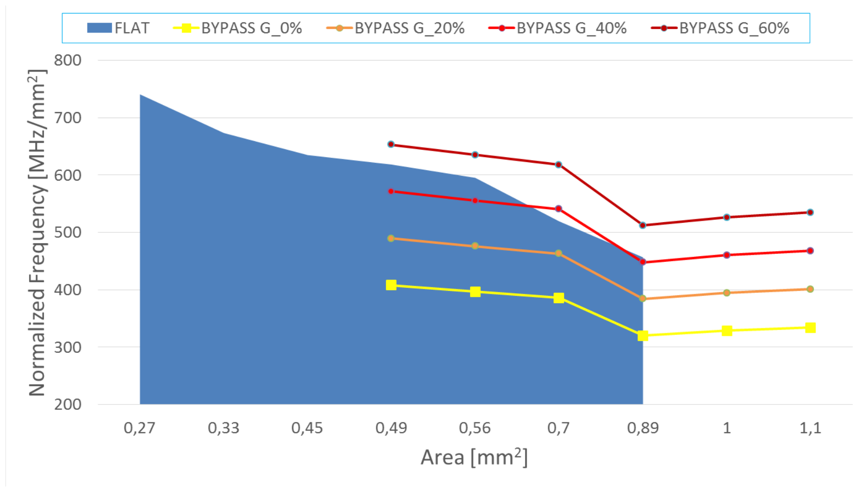


Figure 2.38: 1024+1024 radix-4 Flat MSSN versus Fully-bypassed: computational density vs. area

MUX2:1 ($MSSN_{eq\_mux}$) can be expressed as:

$$MSSN_{eq\_mux} = (M-1) * K * SW_{eq\_mux} \tag{2.10}$$

where SW_{eq_mux} is the number of MUX2:1 used to realize a switch block, and the *(M-1)* factor is due to the fact that the input stage of a MSSN (figure 2.9 and 2.33) can be realized without any Multiplexer.

According to that, assuming a 2-inputs/2-outputs (2x2) switch as composed of 2 MUX2:1 (figure 2.39(a)), the radix-2 MSSN (21 stages, according to equation 2.3) features ≈41k equivalent MUX2:1. On the other side, in the the radix-4 MSSN (11 stages, equation 2.9) the 4-inputs/4-outputs (4x4) switches can be seen as made of 4 MUX4:1 - each equivalent to 3 MUX2:1 (figure 2.39(b)) - so that the total number of equivalent MUX2:1 results ≈58k, a +40% increase with respect to the radix-2 solution. Such value is confirmed by the area values of the Flat MSSN min-area implementations reported in tables 2.1 and 2.3, since the area values vary from 0.20 mm^2 to 0.27 mm^2, a +35% increase.

Table 2.5 shows the comparisons between all the other implementations (reported in tables 2.1, 2.2, 2.3 and 2.4), both in minimum area and in maximum speed configurations. Results show that the radix-4 topology introduces significant area penalties only in the Flat and in the Fully-bypassed min-area cases (+35% and +22% respectively), penalties that are

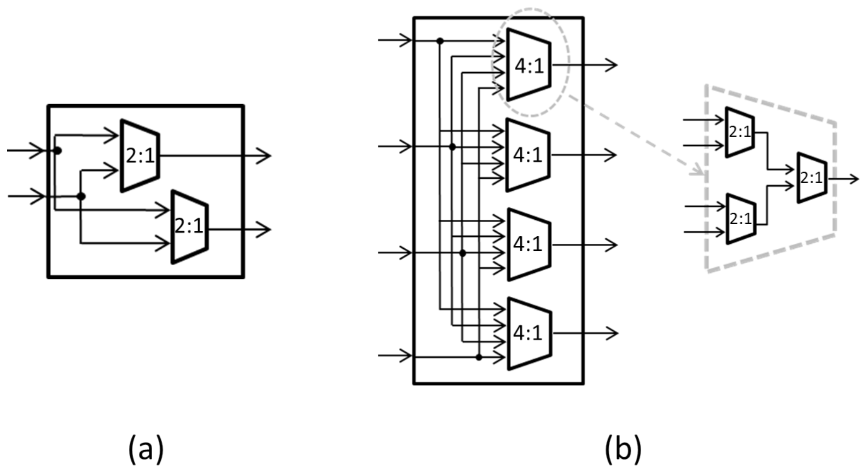


Figure 2.39: Structure of a 2x2 switch (a) and a 4x4 switch (b) expressed in terms of equivalent MUX2:1

significantly mitigated by the synthesis tool optimizations in the other cases (<11%).

	Flat MSSN	Half-bypassed	Fully-bypassed
Min Area	35%	6%	22%
Max Speed	11%	1%	9%

Table 2.5: 1024+1024 I/Os radix-4 versus radix-2: area increase

Given the area penalties introduced by the radix-4 topology, and since in all the topologies both the half-bypassed and the fully-bypassed architectures can provide advantages in terms of working frequency, an analysis of the trade-off between area occupancy and opportunities offered by the *U-turn* bypasses in both radix-2 and radix-4 MSSN has been realized.

As a matter of fact in a regular topology - like the ones taken into consideration - an increase of the number of switch blocks I/Os (MSSN radix) implies a different pattern of connections between the internal MSSN stages, resulting in a different number of stages that have to be crossed to connect two MSSN I/Os, in case of *U-turn* bypasses exploitation.

As an example, as shown in figure 2.40 and 2.41, the number of stages that have to be crossed in a 16+16 I/Os radix-2 folded MSSN to connect IN_0 with IN_7 is 3, while in the 16+16 I/Os radix-4 folded MSSN only 1 stage has to be crossed.

According to that, the half-bypassed and the fully-bypassed version of

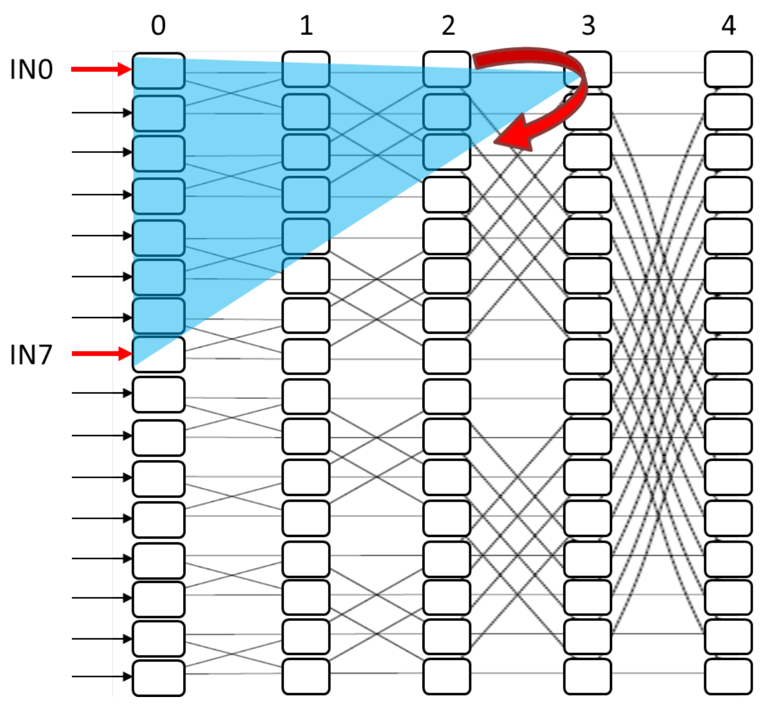


Figure 2.40: Example of connectivity in a 16+16 I/Os fully bypassed radix-2 MSSN

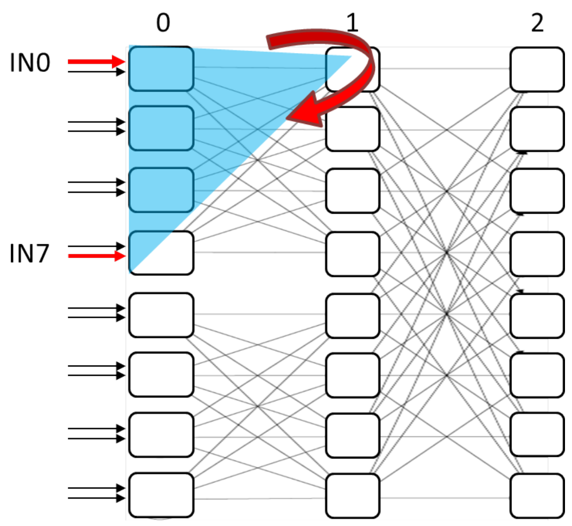


Figure 2.41: Example of connectivity in a 16+16 I/Os fully bypassed radix-4 MSSN

a 1024+1024 I/Os radix-2 and a 1024+1024 I/Os radix-4 MSSNs have been compared, analyzing the number of stages - hence the delay - required to connect two MSSN I/Os placed at different distance. To do so, a set of both max-speed HVT-SVT-LVT and a iso-area SVT-only trials have been considered: for each, the delay associated to a MSSN stage (T_{stage}) has been calculated dividing the implementation period (T_{MSSN}) by the total number of stages introducing significant delay ($M-1$), so that:

$$T_{stage} = \frac{T_{MSSN}}{(M-1)} \tag{2.11}$$

Figures 2.42 and 2.43 report the delay associated to the distance of two MSSN I/Os that have to be connected. As can be seen observing the horizontal axes, in the radix-2 architecture the distance ($DIST$) achievable through each stage (m) is:

$$DIST(m) = 2^m \tag{2.12}$$

while in the radix-4 architecture:

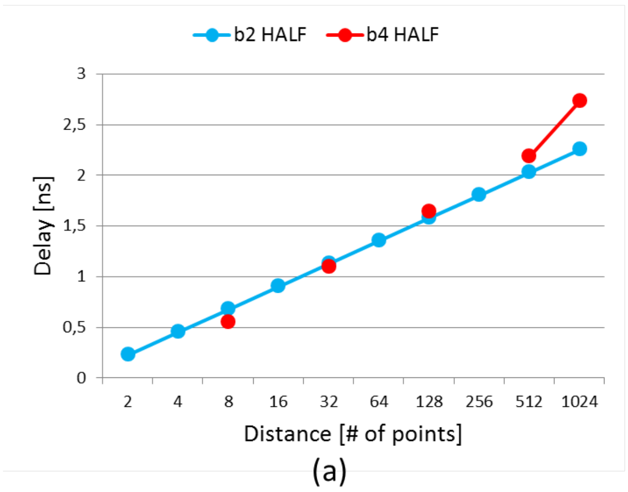
$$DIST(m) = 2 * 4^m \tag{2.13}$$

where the 2-factor is due to fact that each input/output switch is connected to 2 primary inputs/outputs.

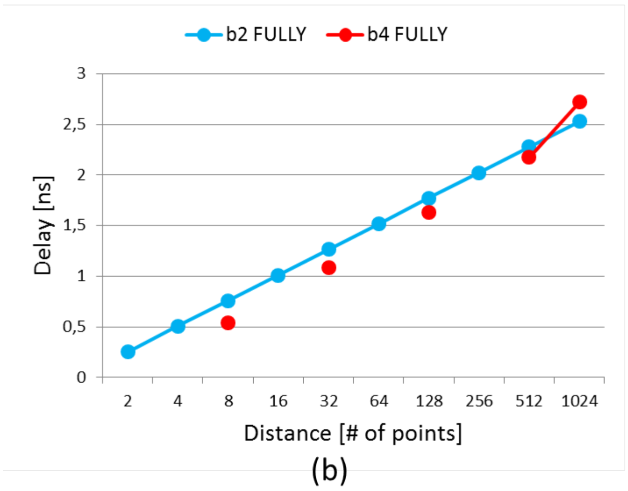
The obtained results show that in the HVT-SVT-LVT max-speed (figure 2.42) half- and fully-bypassed implementations the radix-2 MSSN (light blue line) results faster - less delay - only when the required distance between points is great (>64 and >512 respectively). The same behaviour has been obtained in the SVT-only iso-area (figure 2.43(b)) comparison of fully-bypassed architectures, where the radix-4 (red dots) offers faster connectivity for local (distance <256 points) connections.

Finally, the the iso-area half-bypassed trials (figures 2.43(b)) show that the radix-4 MSSN results always faster than the radix-2, allowing to reach the same distance between points in faster time. This means that, in case of *U-switch* enhanced MSSNs and local connectivity, the radix-4 topology can provide better performances than a radix-2, choice that remains on the other side the best solution in terms of area occupancy.

This chapter has provided an overview of Multi-Stage Switching Networks oriented towards the description - and the performance analysis - of the custom topology chosen to realize routing on an embedded programmable device. Next chapter will describe in details such device, an embedded FPGA, from a state-of-the art analysis to the description of its features, highlighting the advantages provided by the adoption of a MSSN to realize routing.

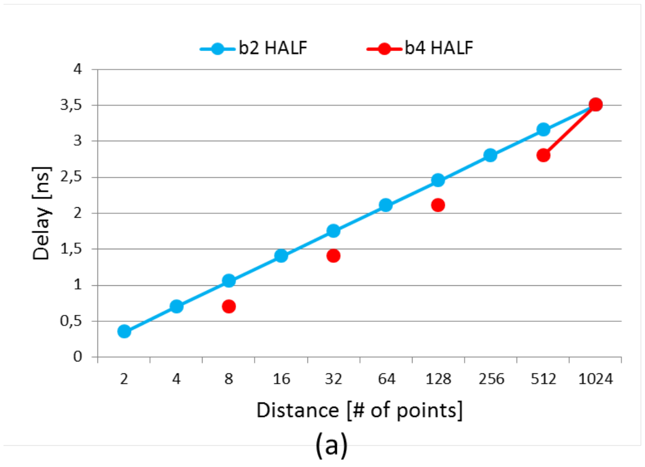


DIST	DELAY [ns]	
	b2 HALF	b4 HALF
2	0,225	
4	0,45	
8	0,675	0,546
16	0,9	
32	1,125	1,092
64	1,35	
128	1,575	1,638
256	1,8	
512	2,025	2,184
1024	2,25	2,73

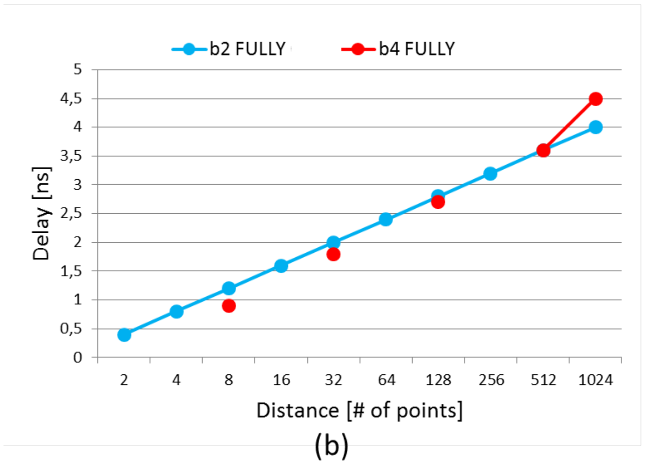


DIST	DELAY [ns]	
	b2 FULLY	b4 FULLY
2	0,253	
4	0,506	
8	0,759	0,544
16	1,012	
32	1,265	1,088
64	1,518	
128	1,771	1,632
256	2,024	
512	2,277	2,176
1024	2,53	2,72

Figure 2.42: Delay associated to the distance between two I/Os on a 1024+1024 half-bypassed (a) and fully-bypassed (b) MSSN - HVTSVTLVT max-speed implementations



DIST	DELAY [ns]	
	b2 HALF	b4 HALF
2	0,35	
4	0,7	
8	1,05	0,7
16	1,4	
32	1,75	1,4
64	2,1	
128	2,45	2,1
256	2,8	
512	3,15	2,8
1024	3,5	3,5



DIST	DELAY [ns]	
	b2 FULLY	b4 FULLY
2	0,4	
4	0,8	
8	1,2	0,9
16	1,6	
32	2	1,8
64	2,4	
128	2,8	2,7
256	3,2	
512	3,6	3,6
1024	4	4,5

Figure 2.43: Delay associated to the distance between two I/Os on a 1024+1024 half-bypassed (a) and fully-bypassed (b) MSSN - SVT-only iso-area implementations

Chapter 3

Embedded FPGA Soft-Core Template

An embedded FPGA is the target device chosen to put in practice and validate the design methodology presented in this thesis work. Thanks to its particular structure, as anticipated in the introduction, the target eFPGA template features three main characteristics: *synthesizability*, *portability* to different technology nodes and *flexibility* in terms of adaptability to different application-specific needs.

Since designed to be embedded in a more complex system (e.g. a SoC), *synthesizability* - hence a soft-core approach - represents the first and most important attribute, crucial to obtain the other two. Such feature has been obtained thanks to the eFPGA particular architecture, that will be described in this chapter, chosen targeting small area budget (up to some tens of equivalent KGates) and enhanced by the utilization of a Multi-Stage Switching Network to realize routing. The fully synthesizability has enabled the possibility to synthesize and optimize the eFPGA template through a standard cell based implementation flow, thus proving its flexibility in terms of *portability* to different technology nodes. As will be presented in chapters 4 and 5, this feature has allowed the implementation of the device on two technologies with very different characteristics: CMOS $65nm$ and BCD9s $0.11\mu m$. Within each of the two types of implementation, the synthesizability has also ensured wide *flexibility* in terms of area-speed-leakage trade-offs, thanks to potential exploitation of all the technology flavours available. Hence for each technology, as shown in the relative chapter (4 and

5 respectively), the wide design space available has demonstrated the optimizability of the device for different application-specific needs, further strengthening its suitability for embedding in a SoC.

This chapter describes the chosen soft-core eFPGA template, presenting both its architecture - starting from a brief overview of similar solutions in industry and academics - and its custom CAD support, realized relying on VTR [23] environment and compatible with MSSN features.

3.1 eFPGA Architecture

The baseline architecture is shown in figure 3.1: the global routing features a hierarchical arrangement, since the logic blocks (CLBs) are connected through a butterfly-based MSSN. Thanks the parametric structure of the template, the designer can set the total number of CLBs, the number of primary I/Os, the dimension and the optional enhancements (e.g. *U-turn* bypasses) of the MSSN, that represents the key point of the overall architecture.

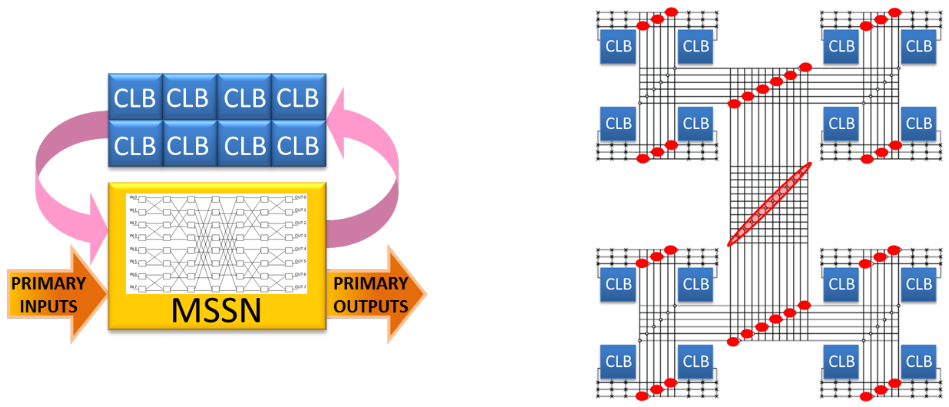


Figure 3.1: Baseline template architecture

Looking at the state-of-the-art, some examples of application of hierarchical interconnects on FPGAs can be found. Abound Logic (former M2000) is the most notable example of a company that provided embedded FPGA: their architecture features a hierarchical interconnect with local crossbars based on Clos networks, ensuring local connectivity with non-blocking properties [24]. Other examples of hierarchical networks have been proposed by Leopard Logic [25] and, more recently, by UCLA [26] which adopts an MSSN based on Benes/Butterfly topology in an huge (2048 equivalent look-up-tables) FPGA. As regards the embedded - hence small and effective - world, the concept of embedded FPGA has never had any wide success in industry, resulting in a lack of a fairly extensive literature: this thesis work was designed precisely to fill such gap. Going into details, the proposed MSSN features the back-to-back butterfly-like architecture described in section 2.2.1: the flexibility of its software support (section 2.2.2) allows any architectural variation in terms of number of I/Os, switch blocks characteristics and internal connectivity (e.g. bypass enhancements). The utilization of a MSSN for an eFPGA interconnect, and especially for a soft-core eFPGA, provides some advantages:

- The blocking properties of the network are well defined and predictable in terms of topology, thus simplifying routability analysis. Such feature acquires greater importance in the particular eFPGA application framework, where the internal connections are statically configured: as a consequence a rearrangeable non-blocking MSSN is equivalent to a fully non-blocking structure and makes the device congestion-free.

- The adaptation of the eFPGA (size scaling, number of I/Os, ..) for specific needs can be achieved by the designer without specific background on eFPGA design, leveraging MSSN congestion-free property.

- The modularity of the network (based on the regular replication of small basic switch-modules) enables a truly effective soft-core approach. Each switch block can be implemented by standard cells or optimized at circuit level as a single coarse-grained cell, without affecting the whole eFPGA synthesizability properties.

The CLB model has been chosen from an analysis of the architectural solutions available in VTR [23], an open-source tool for FPGAs: the resulting scheme, depicted in 3.2, represents an effective trade-off between complexity and general-purposeness, with good balancing between number of pins and computational resources. Taking up the structure proposed in section 1.3.1, figure 1.7, each CLB features:

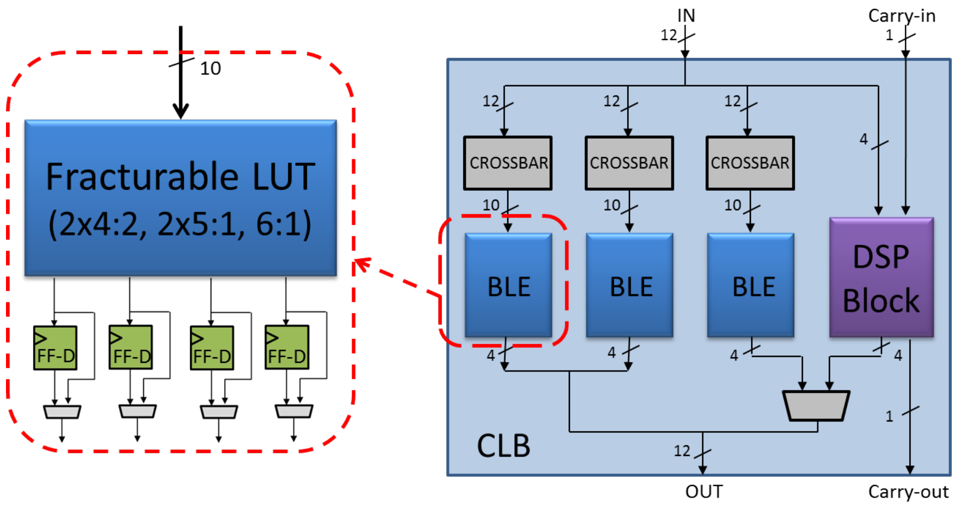
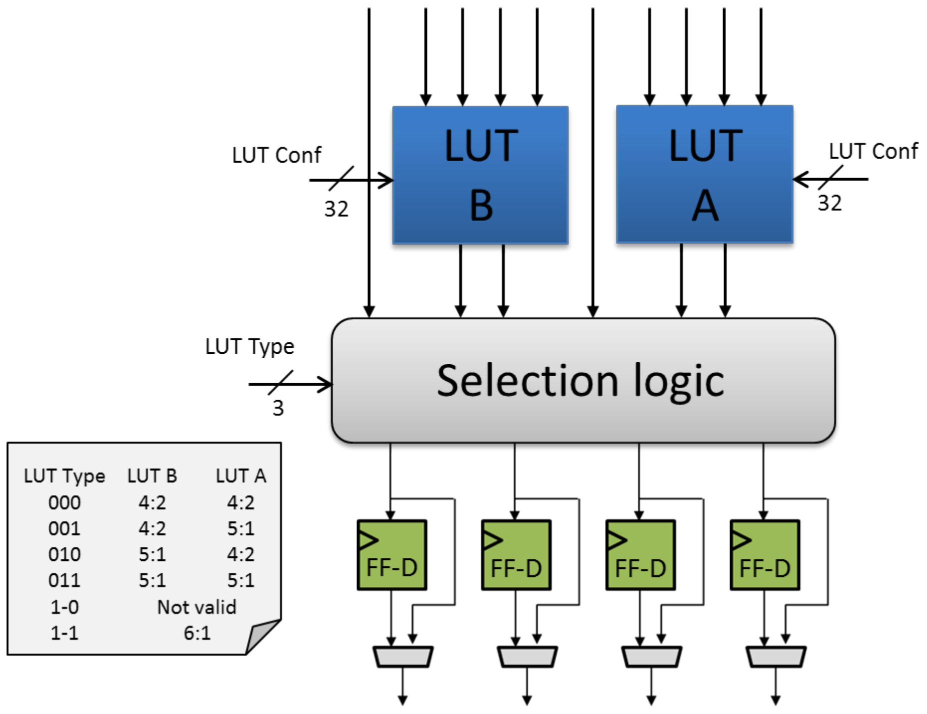


Figure 3.2: CLB description

- 13 inputs (12 + Carry-in) and 13 outputs (12 + Carry-out);
- 3 independent BLE blocks, each one containing two LUT blocks (A and B in figure 3.3), that can be both used to form a LUT 6:1 or fractured down to 2 LUT 4:2; this is obtained thanks to a selection logic driven by 3 configuration bits (*LUT_Type*), that produces 4 outputs. Each output is hence followed by a flip-flop which can be bypassed using a multiplexer;
- 3 12x10 input crossbars, used to connect LUT blocks to main inputs, resulting in partial sharing among blocks: as depicted in figure 3.4 these 12x10 small interconnection networks are realized using 10 12:1 multiplexers, and can be configured using 40 bits. This simple architecture has been chosen as it requires less area than an equivalent 12x10 Multi-Stage Switching Network: MSSNs indeed result to be efficient for higher dimensions (i.e. I/O multiplicity beyond hundred of points);
- A small/simple generic DSP block, sharing inputs and outputs with some LUT blocks.

To preserve a fully-synthesizable soft-core approach, configuration bit-cells and logic are implemented with simple latches, logically organized in small local square matrices. Hence for each CLB and for each MSSN



LUT Type	LUT B	LUT A
000	4:2	4:2
001	4:2	5:1
010	5:1	4:2
011	5:1	5:1
1-0	Not valid	
1-1	6:1	

Figure 3.3: Schematic of a fracturable LUT block

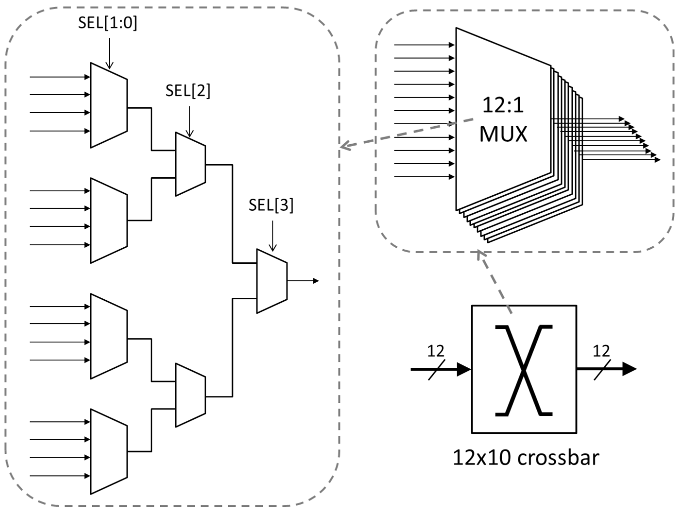


Figure 3.4: Schematic of a 12x10 crossbar

level there is a dedicated bitcell matrix configured by 2 scan-chains (one for data words and one for rows write-enables, as depicted in figure 3.5). Compared with a global memory organization, this solution pays a small increase in the total number of flip-flops, but allows shorter and faster configuration paths. In addition, it alleviates global routing congestion during implementation.

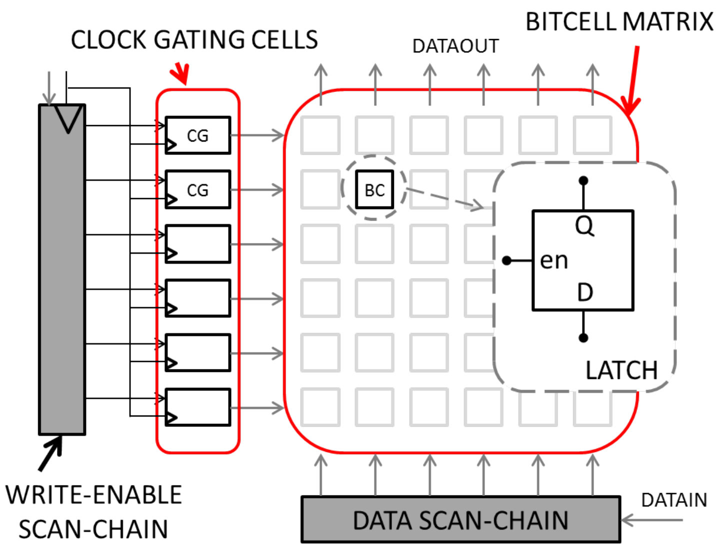


Figure 3.5: Bitcell matrix organization

To make the template fully synthesizable (and place-and-route implementable), some guidelines have been adopted:

- Bitcell write-enables are driven by clock-gating cells, where the configuration clock is enabled by a scan-chain write-enable (figure 3.5). The configuration sub-system clock is mostly asynchronous with respect to the functional clock driving the CLB flip-flops, since configuration is performed in quasi-static way.

- Uncommitted interconnects and CLBs with potentially combinational outputs cause a combinational feedback, resulting in timing loop (figure 3.6 (a)). This condition is peculiar to the uncommitted device since all the bitcells can be either 0 or 1. To break the timing loop, during implementation all CLBs are forced to be sequential (e.g. through *set_case_analysis* or *set_disable_timing* constraints on the output multiplexers, as shown in figure 3.6 (b)). The interconnect itself is loop-free by construction, since it features a topology described by an oriented graph.

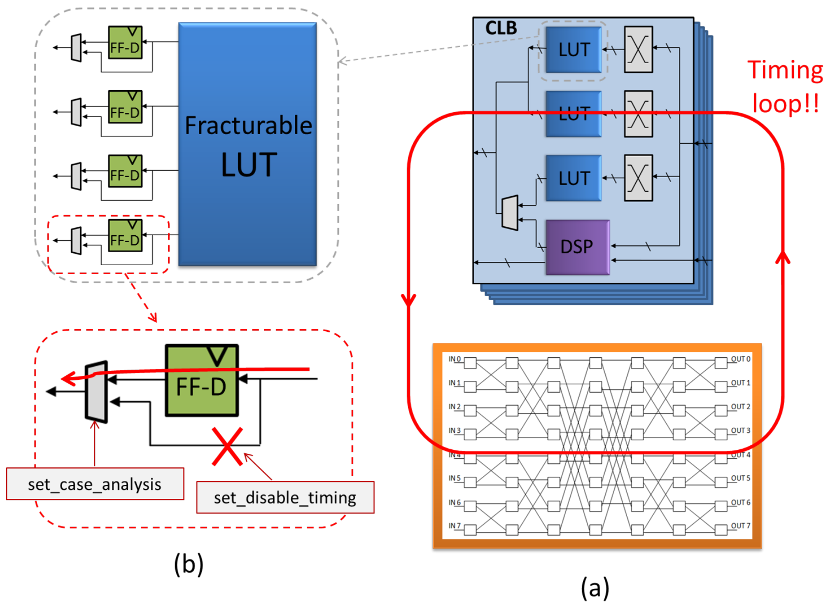


Figure 3.6: Uncommitted device timing loop (a) broken configuring CLB Flip-Flops (b)

3.2 eFPGA CAD Flow Support

To generate the bitstream to configure the eFPGA, a complete CAD flow support has been implemented, leveraging on the availability of a state-of-the-art open-source VTR [23] environment. The structure of the flow follows the typical structure of an FPGA CAD tool (section 1.3.2), and is shown in figure 3.7. Starting from a behavioural RTL description of the function to map into the eFPGA, VTR flow provides logical synthesis and physical LUT mapping, namely the synthesis, packing and place-and-route steps. Since in VTR standard version these steps target bi-dimensional arrays interconnected through segmented routing, some modification was required to support the particular eFPGA architecture.

The standard front-end flow - properly configured in terms of CLB features - has been adopted for logical synthesis (ODIN-II [27] tool) and physical LUT mapping (ABC [28] tool) steps. The resulting .blif *Logic file* with the LUT functional description is then processed by the packing step (AAPack [29]), able to produce a .net *Hierarchy file*.

Before these, as shown in figure 3.7, a simple (optional) pre-synthesis step has been added to overcome syntax limitation of ODIN-II: based on

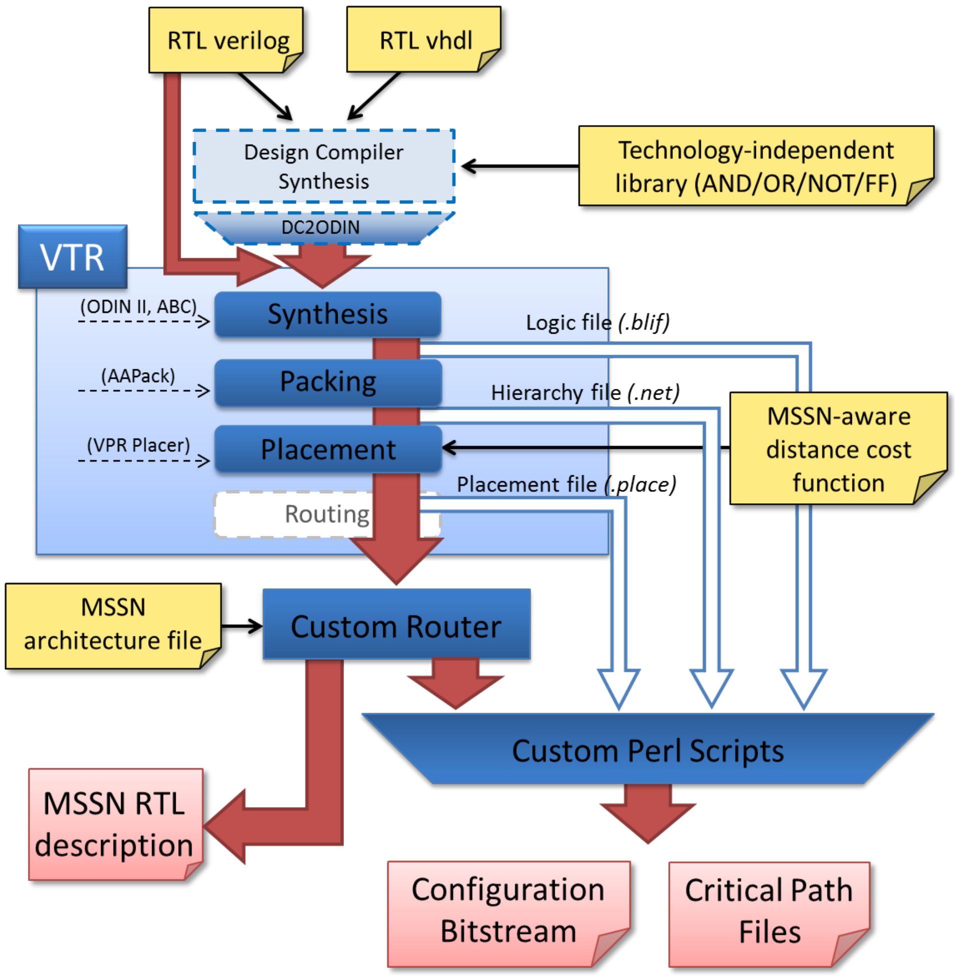


Figure 3.7: CAD Flow Overview

Synopsys Design Compiler tool [22] it maps the input RTL into a minimalistic technology-independent library with AND, OR, NOT and Flip-Flop functionalities. The resulting structural Verilog is syntactically full-compatible with ODIN-II through a set of custom scripts (DC2ODIN in figure 3.7).

As regards the place-and-route steps (based on VPR [23]), since the peculiarity of the eFPGA interconnection network requires MSSN-aware routing, the placement tool has been modified and the routing step replaced by a custom one, able to support MSSN, as detailed in the following sections.

Finally a set of ”hand-crafted” Perl scripts have been implemented to allow for elaboration of the files generated during the synthesis, packing, placement and routing steps: such scripts are able to produce both the eFPGA configuration bitstream and a set of *critical path analysis* files useful to evaluate routing performances. A section will be dedicated to a more detailed description of these scripts.

3.2.1 Placement Algorithm

As anticipated in section 1.3.2, the placement is responsible for logic block locations on the eFPGA, the aim being to reduce the total wire-length of all nets in the circuit (possibly adjusted with timing-driven optimization). The VPR placement engine is based on simulated annealing [23] [30] with a clear separation between the optimization algorithm (annealing) and the cost function to be optimized (in the simplest case, the total wire-length). To implement an MSSN-aware placement, the cost function has been modified, for both path-driven and timing-driven analysis.

Natively, like most placement engines, VPR estimates the wire length as proportional to the bounding box surrounding the source and sinks of the nets, a well-accepted approximation. In particular the cost of a net is

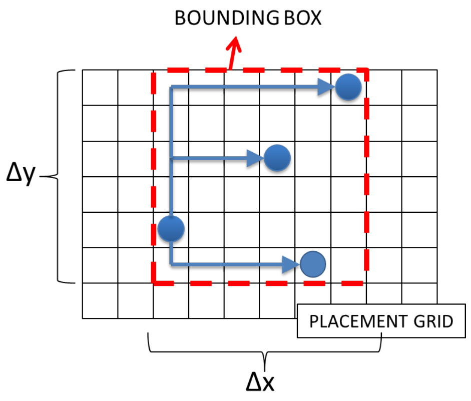


Figure 3.8: Example of bounding box on a 2D placement grid

defined as equal to the Half Perimeter Wire-Length (HPWL), so that (figure 3.8):

$$Net\_cost = \Delta x + \Delta y \tag{3.1}$$

and the total wire-length W for a given placement configuration featuring

N nets is:

$$W = \sum_{k=1}^{N} Net\_cost(k) \tag{3.2}$$

While this metric works properly for 2D placement of ASIC and/or FPGA (possibly adjusted with timing- and congestion-aware parameters), for an MSSN it provides a placement that is far from optimum.

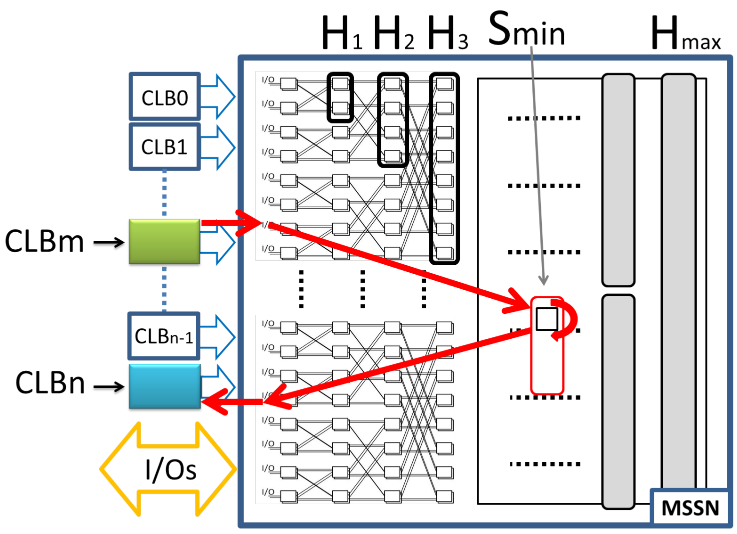


Figure 3.9: CLB distance on an MSSN

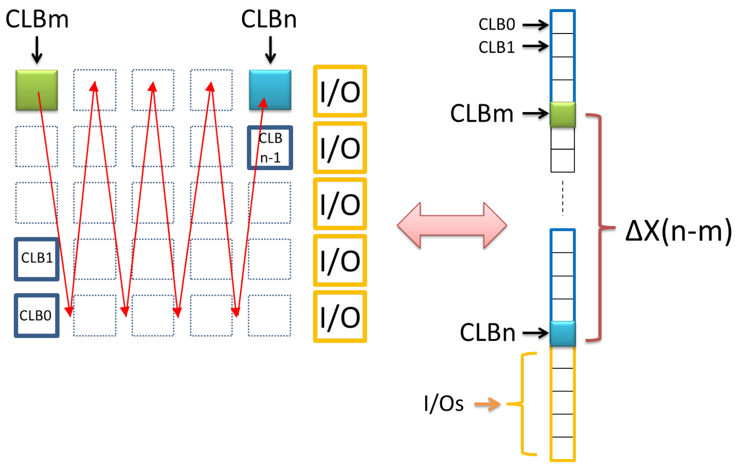


Figure 3.10: CLB column linearization of a 2D CLB placement

As shown in figure 3.9 and 3.10, on a MSSN-based eFPGA the distance between two CLBs (*CLB_m* and *CLB_n*) cannot be calculated under 2D metrics (like HPWL) since the distance depends on the relative position on the graph and not on the 2D arrangement of the CLBs. A 2D hierarchical arrangement (like the one shown in section 2.2.1, figure 2.12) can be obtained thanks to ordering CLBs by columns, with an orderly correspondence between the 2D placement grid (figure 3.10) and the position of the CLBs in the logarithmic network (figure 3.9).

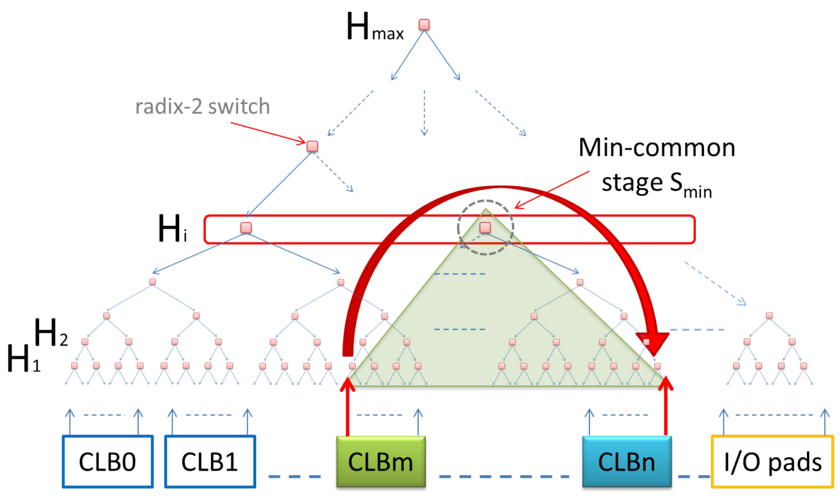


Figure 3.11: CLB connection through a MSSN represented as a binary-tree

As described in section 2.3.2, the particular MSSN structure can be represented by an n-ary tree (in case of radix-2 switches, a binary tree) and the cost to a net connecting two or more points is roughly proportional to the height of the minimum-cone (figure 3.11) that subtends all the points (source and sinks). That height corresponds to a hierarchical stage H_i of the MSSN.

Defining S_{min} the minimum stage subtending all the points of the k^{th} net, the total wire-length W for a given placement configuration featuring N nets is:

$$W = \sum_{N}^{k=1} 2 * S_{min}(k) \tag{3.3}$$

where the 2 factor derives from the forward and backwards folded paths.

To obtain a timing-driven cost function, since the delay associated with each MSSN stage can be roughly determined, the total cost becomes:

$$W_{time} = \sum_{N}^{k=1} 2 * \sum_{i=1}^{S_{min}(k)} \alpha_i \tag{3.4}$$

where α_i represents the delay associated with each stage H_i crossed toward S_{min}. This function is very similar to the previous one and since α_i coefficients are at first approximation equal or smoothly different, timing-driven and path-driven algorithms provide very similar results. In addition, since the network provide congestion-free, no adjustment is required to take into account congestion hot-spots, thus simplifying the overall cost function.

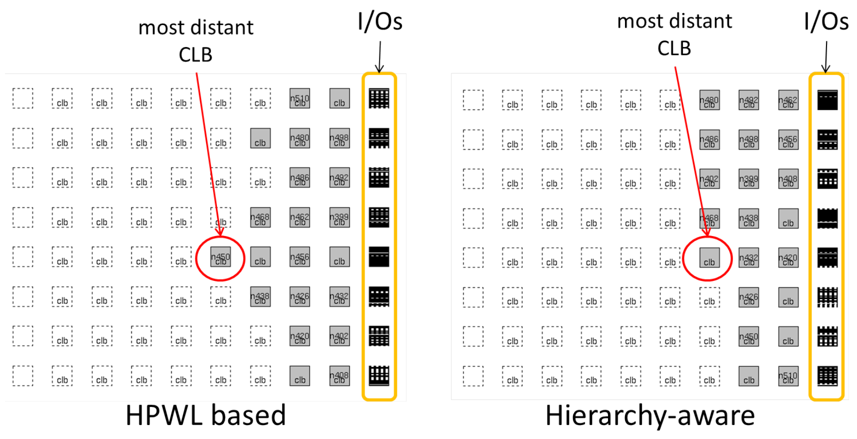


Figure 3.12: Placement comparison

To give an example, figure 3.12 shows a comparison between two placements, the first one realized with a standard 2D HPWL and the other one with the MSSN-specific cost function. In both cases, I/O pads are positioned only on the right perimeter of the device, position that corresponds to the rightmost side of the MSSN I/Os (see figures 3.9 and 3.10): further software modifications allow indeed the user to select both a standard I/O location (whole perimeter) and a customized one (left-only, right-only and left-right sides), as shown in figure 3.13.

3.2.2 Routing Algorithm

As shown in the CAD flow summary graph (figure 3.7), the VTR standard routing tool has been replaced by a tool dedicated to MSSN-based struc-

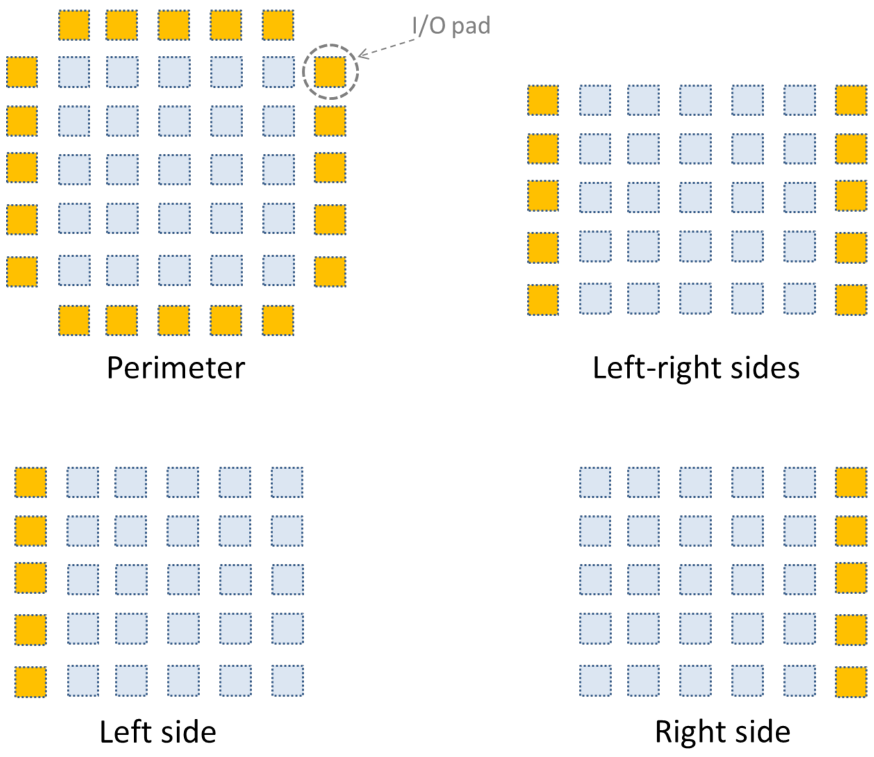


Figure 3.13: Different I/O pads location

tures. The structure of the developed C software tool can be seen as divided in two main steps, as shown in figure 3.14, called *Build_MSSN* and a *Route_on_MSSN* respectively.

The *Build_MSSN* algorithm has been described in details in section 2.2.2: it takes as its input a .xml *Architecture_file* containing the description of the desired MSSN and gives as its output a *Resource Graph* representing the MSSN, essentially made of switch blocks connected through wires.

The MSSN *Resource Graph* is then taken as an input by the following step, *Route_on_MSSN*: as detailed in section 2.3.2, this routing algorithm also needs a .txt *Netlist_file* - in this case provided by the VTR placement step - containing the description of the internal eFPGA connection to be performed.

Finally, in case of successful routing operation, the custom router provides the RTL description of the MSSN and its configuration bitstream, together with statistical information relative to the wirelength distribution.

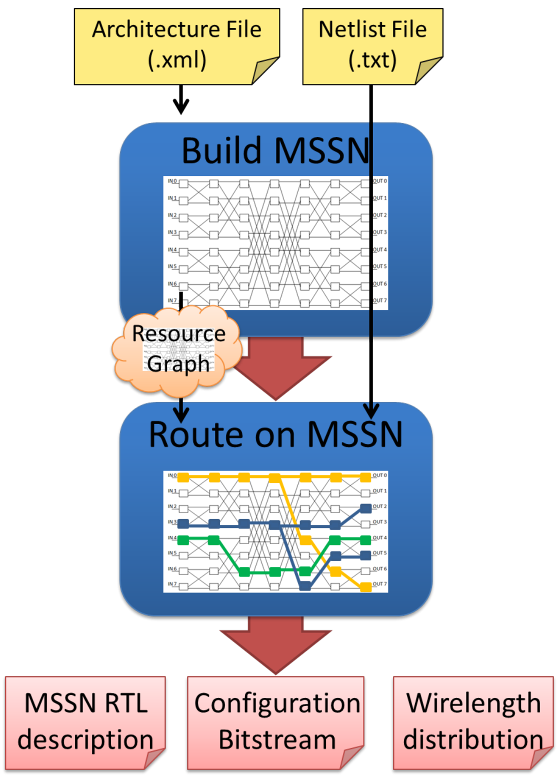


Figure 3.14: Routing algorithm flow

3.2.3 Perl Scripts

As final step, since packing, placement and routing results come from different tools, a set of Perl scripts has been build up to elaborate the heterogeneous files produced during each intermediate step and provide the eFPGA configuration bistream.

As shown in figure 3.15, three different scripts are initially used to work out the *Logic*, *Hierarchy* and *Placement File* respectively:

- $Expand\_Blif$ is able to expand the original .blif file, in which the LUT (.names) descriptions are expressed in a compact way (figure 3.16 (a)), in order to obtain the complete configuration bitstream for each LUT;
- $Adapt\_Net$ is used as 'beautifier' for the .net file, that contains a verbose CLB list, producing a 'Perl-compatible' hierarchical file containing CLB informations;
- $Compact\_Place$ is an optional step that analyzes the .place file filling any hole in the placement (due to some approximation related to

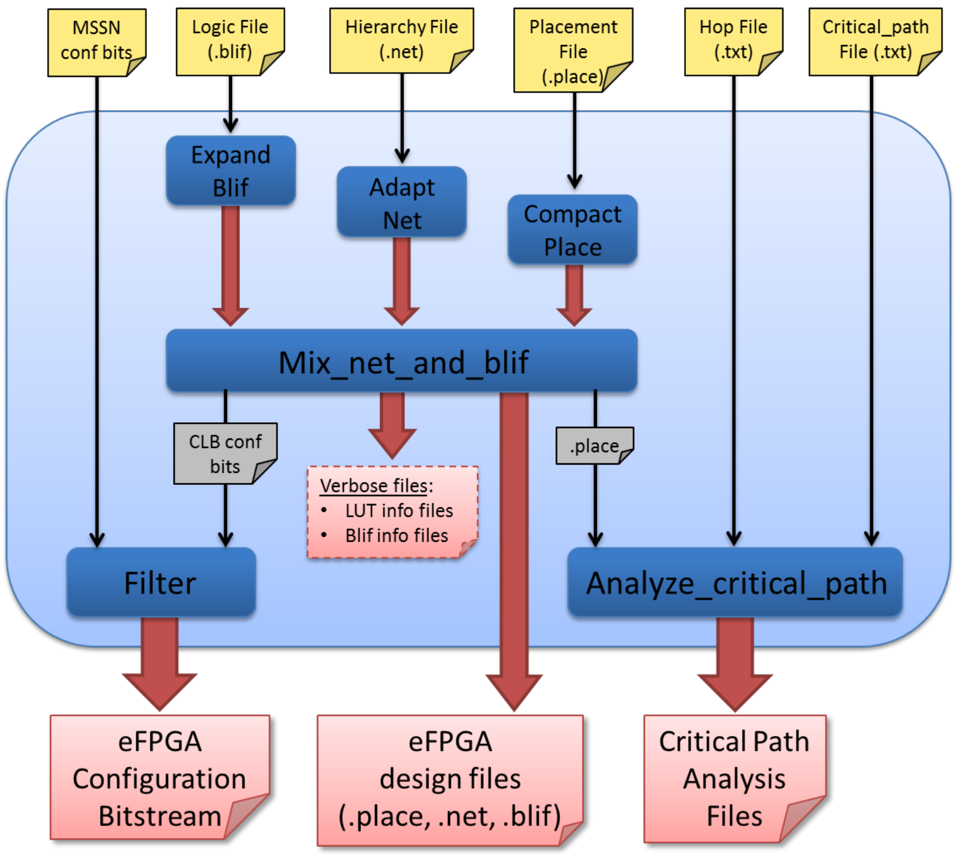


Figure 3.15: Perl scripts flow

the customization of the placement function), according to a MSSN-aware column linearization of the 2D placement (figure 3.16 (b)).

Hence a more complex script (*Mix_net_and_blif* in figure 3.15) is used to cross the information contained in these three files (e.g. synchronizing for example the LUT descriptions contained in the .blif with that in the .net CLBs hierarchy) producing three homogeneous versions of the *Logic*, *Hierarchy* and *Placement File*, and the CLB configuration bitstream. The latter is then processed, together with the MSSN bistream given by the custom routing tool, by a *Filter* script, that cuts the bitstream according to the required parallelism (in figure 3.17, the bistream is cut so as to be compatible with an 8-bit wide memory data width). In addition an optional 'verbose' option allows the dump of a set of files containing several informations regarding the performed steps.

Another script (*Analyze_critical_path* in figure 3.15) has been implemented

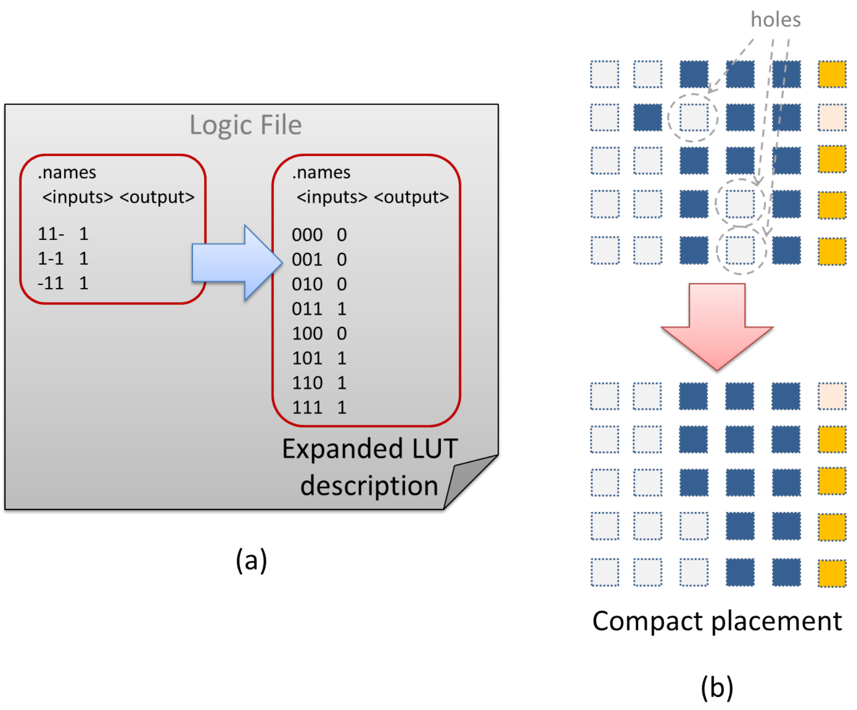


Figure 3.16: Example of .blif expansion (a) and placement (.place) compaction

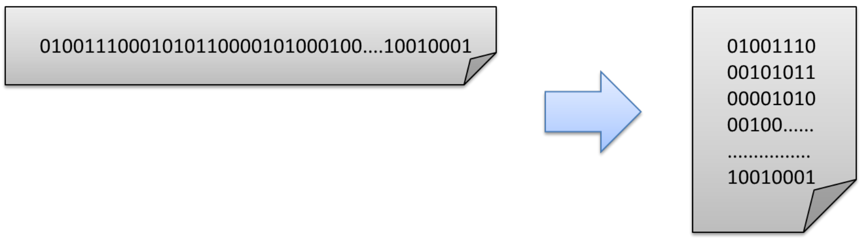


Figure 3.17: Flat configuration bitstream cut according to an 8-bit wide data path

for the evaluation of the benefits of any MSSN architectural enhancement (e.g. bypass) on the eFPGA working frequency: in particular the improvements in terms of critical path length are evaluated. The analysis takes as input:

- an *Hop File* (.txt), produced by the router, containing the length of each internal eFPGA connection, expressed in terms of MSSN stages crossed (*hops*): such length will be the full MSSN latency in case of a *flat* (see section 2.2.1) architecture, or a smaller value in case of any bypass enhancement (figure 3.18);

- a *Critical_path File* (.txt), given by VTR after placement step, with the list of the nets in the critical path;
- the *Placement File* (.place), used to check the correspondence between the name of a net and its position on the MSSN I/Os (hence on the placement grid).

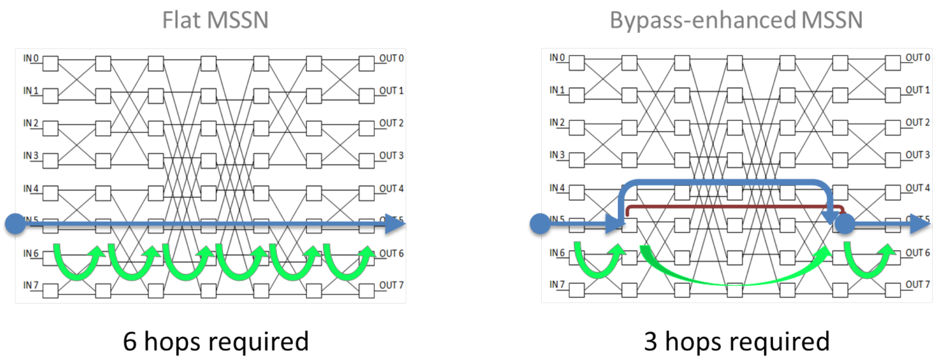


Figure 3.18: Different length of a net in a flat and a bypassed MSSN

The script elaborates the received data and gives as output the length (number of *hops*) of the critical path, comparing it with a flat MSSN solution, in which the length of each net is predictable and equal to the total number of MSSN stages; in addition the statistical information related to the wire-length distribution are provided.

Chapter 4

eFPGA for CMOS

This chapter continues the eFPGA analysis: as described at the begin of chapter 3, the proposed template features *synthesizability*, *portability* to different technology nodes and *flexibility*, expressed in terms of optimizability for different application needs. With the aim to demonstrate these characteristics and realize a functional verification, as first the design has been implemented targeting STMicroelectronics CMOS 65nm LP 1.10V technology.

Leveraging on the availability of a rich portfolio of CMOS libraries - that include for example standard cells with different transistor thresholds (V_t) - three different analysis has been performed: a design-space exploration, an application-aware quantitative analysis of such design space and a discussion on the computational density achievable by different configuration of the MSSN for a fixed area budget. These aspects are fundamentals since they prove the effectiveness of the soft-core approach for integration on SoCs with different needs and specifications: for that, after the results of the implementations, a paragraph will be dedicated to each analysis.

4.1 Implementation Results

This section shows the results of the implementation trials of the soft-core eFPGA template. As descripted before, thanks to the parametric structure, the designer can set the total number of CLBs, the number of primary I/Os, the dimension of the MSSN used and its optional enhancements (e.g. fully, partially or not bypassed by *U-turns*).

For that, in order to obtain a more complete analysis, two test-cases have been analyzed, featuring 16 CLBs and 64 CLBs respectively; for each, two different versions of the MSSN have been adopted, a flat (i.e. without any bypasses) and a fully-bypassed one (i.e. with *U-turn* connections on each stage, see section 2.2).

The first test-case is hence composed by 16 CLBs (figure 4.1), corresponding to $\approx 1K$ equivalent gates (192 LUT 4:1), each one featuring the architecture descripted in figure 3.2, without the dedicated DSP block and carry-chain; routing is realized with a 256+256 I/Os radix-2 MSSN. Since *butterfly* topology features power-of-two multiplicity, 192+192 pins are required for CLB connections (16 CLBs $\times$ 12 I/Os), while the remaining 64+64 I/Os are used for primary eFPGA I/Os.

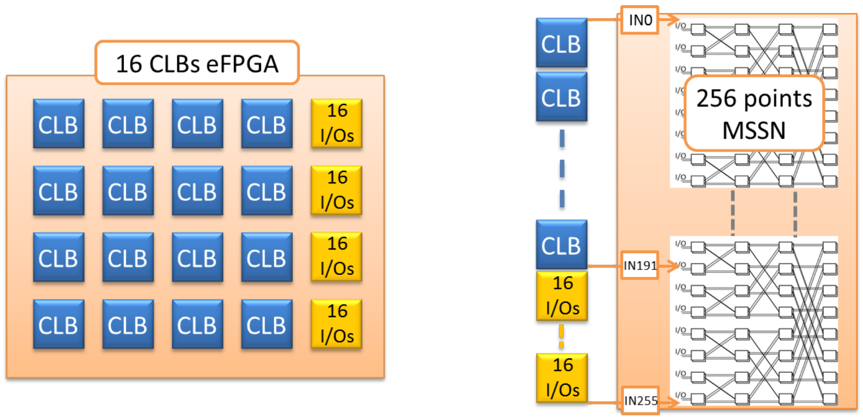


Figure 4.1: eFPGA featuring 16 CLBs and a 256 points MSSN

Synthesis was performed using the Synopsys Design Compiler Graphical tool [22] choosing standard cells with a standard V_t (SVT - Standard Voltage Threshold). Synthesis was thus performed with a real floorplan and physical coarse placement to quickly estimate parasitics on wires, as well as potential congestion issues, thus improving the correlation with respect to the place-and-route phase. Target implementation frequency is the *time-to-fly* between two CLBs configured with synchronous outputs; thus timing constraints refer to the longest paths of the uncommitted device crossing the full MSSN. As an educated guess, I/Os were constrained with a delay of $\approx 1/3$ the target clock period.

Post-synthesis results of both a flat and a fully-bypassed version are shown in tables 4.1 and 4.2 respectively, with two different implementations optimized per area and speed, showing quite different figures of merit. In both architectures, the MSSN contribution over the total area is ≈ 50%, while the bitcells overhead both on the MSSN area and on the total area redoubles when synthesis is optimized per area, rising from ≈ 25% to ≈ 50-60%.

	Flat MSSN	
	Max Speed	Min Area
Area $[mm^2]$	0.4	0.19
Frequency $[MHz]$	250	100
Leakage $[\mu W]$	499	165
% Bitcells on total area	23.5%	47.6%
% MSSN area	55.5%	47.9%
% Bitcells on MSSN area	25%	59.2%

Table 4.1: CMOS 65nm SVT 16 CLBs Flat MSSN eFPGA post-synthesis summary

	Fully-bypassed MSSN	
	Max Speed	Min Area
Area $[mm^2]$	0.55	0.27
Frequency $[MHz]$	224	100
Leakage $[\mu W]$	665	233
% Bitcells on total area	23.4%	46.7%
% MSSN area	66.1%	60.6%
% Bitcells on MSSN area	25.7%	55.9%

Table 4.2: CMOS 65nm SVT 16 CLBs Fully-bypassed MSSN eFPGA post-synthesis summary

Table 4.3 shows the area and leakage increase due to the full bypass enhancement, in both maximum speed and minimum area, showing area and leakage penalties of ≈ +40% without significant implementation frequency reduction (-10%).

	Fully-bypassed versus Flat MSSN	
	Max Speed	Min Area
Area $[mm^2]$	37.5%	42.1%
Frequency $[MHz]$	-10.4%	-
Leakage $[\mu W]$	33.2%	41.2%

Table 4.3: 16 CLBs eFPGA comparison: fully bypassed versus flat MSSN

Place-and-route was done using Cadence Encounter [31], closing in all cases the design without timing or DRC (Design Rule Check) violations and obtaining area and power comparable to post-synthesis results. Moreover, as anticipated, since the main purpose of our analysis was to verify feasibility and functionality of the approach, no particular optimizations or guidelines were adopted throughout the flow, resulting in a kind of ”worst-case” analysis. The only guideline that has been considered is that place-and-route has been performed with a hierarchical netlist, in order to allow a placement optimization oriented to block connectivity. As a result, in the layout shown in figure 4.2 the CLB blocks are kept as islands in the design, while bitcells are spread over the whole floorplan.

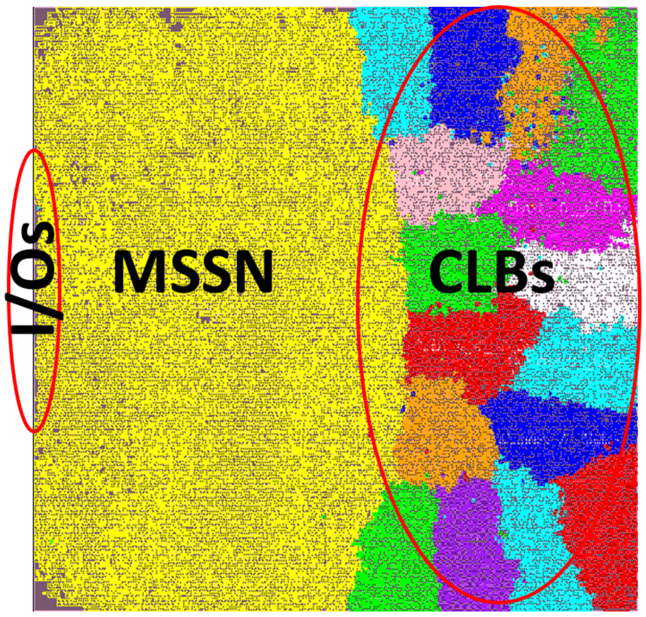


Figure 4.2: CMOS 65nm floorplan of a 16 CLBs eFPGA

Finally, back-annotated simulations were carried out to verify the functionality, thus verifying the constraints applied to break timing loops (see section 3.2).

To make the analysis more in line with potential real-application needs, also a test-case featuring 64 CLBs (figure 4.3) has been taken into consideration: in this case logic blocks were connected by a radix-2 MSSN featuring 1024+1024 I/Os with and without bypass enhancement, with 768+768 (64 CLBs x 12 I/Os) pins dedicated to CLBs connections and 256+256 for eFPGA primary I/Os.

Post-synthesis results, obtained following the same guidelines adopted

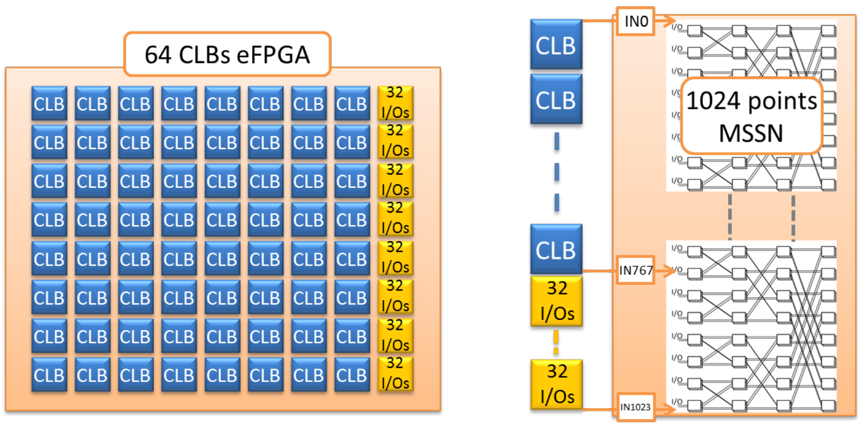


Figure 4.3: eFPGA featuring 64 CLBs and a 1024 points MSSN

in the 16 CLBs eFPGA, are shown in tables 4.4 and 4.5, for a design featuring a flat MSSN and a fully-bypassed MSSN respectively. In such cases the percentage of bitcells area both on MSSN and eFPGA areas do not present variations, if compared with the values of the 16 CLBs design, while the MSSN contribution over the total area rises up to $\approx$ 70%. As shown in table 4.6 both the area overhead and the leakage penalties introduced by the bypass enhancement result in line with the values obtained with a 16 CLBs eFPGA, achieving values around 40%, while a small decrease in terms of implementation frequency (down to -17%) can be observed when the design is optimized per speed.

	Flat MSSN	
	Max Speed	Min Area
Area $[mm^2]$	1.54	0.85
Frequency $[MHz]$	180	83
Leakage $[\mu W]$	1808	777
% Bitcells on total area	24.5%	51.4%
% MSSN area	60.2%	55.0%
% Bitcells on MSSN area	30.0%	58.0%

Table 4.4: CMOS 65nm SVT 64 CLBs Flat MSSN eFPGA post-synthesis summary

	Fully-bypassed MSSN	
	Max Speed	Min Area
Area $[mm^2]$	2.11	1.22
Frequency $[MHz]$	149	66
Leakage $[\mu W]$	2482	1064
% Bitcells on total area	28.9%	51.0%
% MSSN area	70.0%	67.4%
% Bitcells on MSSN area	32.1%	56.6%

Table 4.5: CMOS 65nm SVT 64 CLBs Fully-bypassed MSSN eFPGA post-synthesis summary

	Fully-bypassed versus Flat MSSN	
	Max Speed	Min Area
Area $[mm^2]$	37.0%	43.5%
Frequency $[MHz]$	-17.2%	-
Leakage $[\mu W]$	37.2%	36.9%

Table 4.6: 64 CLBs eFPGA comparison: fully bypassed versus flat MSSN

4.2 Design-Space Exploration

This section will show that leveraging on the technology options available in CMOS technology it is possible to optimize the soft-core eFPGA for quite different area-speed-leakage trade-offs. As a matter of fact STMicroelectronics CMOS 65nm LP technology features standard cells libraries characterized by transistors with 3 different MOSFET threshold voltage V_t (High, Standard and Low Voltage Threshold - HVT, SVT, LVT): exploiting this availability, as first a 16 CLBs test-case featuring a flat MSSN has been analyzed.

The reference implementation was the SVT-only one described in table 4.7, and starting from this reference a comparison has been performed with respect to implementations featuring different cell mixes:

- HVT-only, to minimize leakage;
- HVT+SVT, a near-to-SVT option to reduce leakage without significant performance penalties;
- HVT+SVT+LVT, for high speed applications.

Each test-case was implemented varying the target frequency up to its technology limit, following the same methodology as described in section 4.1.

	Flat MSSN	Fully-bypassed MSSN	%
Area $[mm^2]$	0.38	0.47	23.7%
Frequency $[MHz]$	234	200	-14.5%
Leakage $[\mu W]$	457	552	20.8%
% Bitcells on total area	24.1%	14.5%	-
% MSSN area	53.2%	63.4%	19.2%
% Bitcells on MSSN area	27.2%	30.9%	13.6%

Table 4.7: CMOS 65nm SVT 16 CLBs SVT reference case

Each trial had area and leakage values minimized by constraints and I/Os were constrained with a delay of $\approx 1/3$ the target clock period. The feasibility of the implementation was verified in some relevant case through complete place-and-route flow, analyzing the results at signoff level in order to evaluate the correlation.

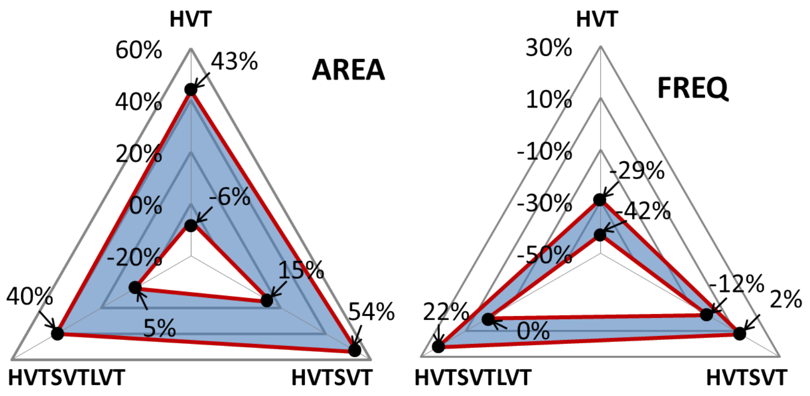


Figure 4.4: Area and frequency design space - 16 CLBs flat MSSN version

Figure 4.4 shows the achieved design space in terms of area and frequency ranges. For example, HVT-only designs show an area ranging from -6% to +43% compared to SVT reference, while the corresponding frequency range decreases from -42% to -29,2%.

Tables 4.8 and 4.9 show the minimum and maximum leakage values for the various implementations, reporting the correspondent V_t-mix. Compared to the SVT design, leakage power decreases by one order of magnitude with the HVT-only low-speed solution (from -87% to -91%) or increases up to 400% leveraging high-speed cells utilization, choice that al-

V_t	Max Leak	% cells		
		HVT	SVT	LVT
H	-87.1%	100%	-	-
H,S	57.1%	17%	82%	-
H,S,L	396.8%	17%	26%	56%

Table 4.8: Leakage power design space summary: maximum values in the 16 CLBs flat MSSN

V_t	Min Leak	% cells		
		HVT	SVT	LVT
H	-91.9%	100%	-	-
H,S	-1.5%	30%	69%	-
H,S,L	46.8%	35%	53%	11%

Table 4.9: Leakage power design space summary: minimum values in the 16 CLBs flat MSSN

lows to reach a frequency speed-up up to 22% together with area penalties that vary from +5% to +40% (figure 4.4).

The bypass-enhanced version of the same 16 CLBs eFPGA was also evaluated, taking as reference case the SVT-only implementation described in table 4.7: compared to the flat MSSN reference case, it features small variations in terms of implementation frequency (-14%), area occupancy (+23.7%) and leakage (+20%), together with a rise in the MSSN contribution over the total area (+19%). As for the flat version, also in the by-

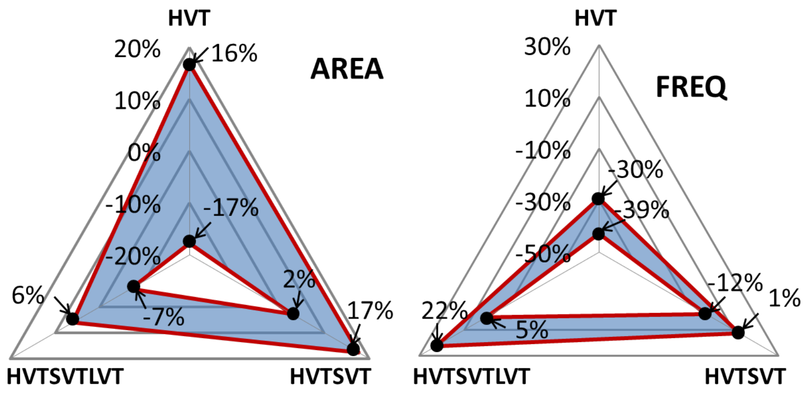


Figure 4.5: Area and frequency design space - 16 CLBs fully-bypassed MSSN version

pass architecture the design-space available (figure 4.5) results significant, mainly in terms of implementation frequency variation, that can vary from the -30%/-39% values of the HVT-only low-speed configuration up to the +5%/+22% of the fastest HVT-SVT-LVT solution. The area range results smaller with respect to the one obtained in the flat MSSN design space, with variations between +17% of the HVTSVT case and -17% of the HVT solution.

Following the same methodology, a 64 CLBs eFPGA was also taken into consideration: the reference implementations were the iso-frequency (125 MHz) one described in table 4.10, both for a flat and a fully-bypassed MSSN version. In this case the area and leakage increase due to the bypass enhancement were +50% and +43% respectively, together with a +19% growth in the MSSN area contribution.

	Flat MSSN	Fully-bypassed MSSN	%
Area $[mm^2]$	1.2	1.8	50%
Frequency $[MHz]$	125	125	-
Leakage $[mW]$	1.4	2.0	43%
% MSSN area	59.8%	71.2%	19%

Table 4.10: CMOS 65nm SVT 64 CLBs SVT - iso-frequency reference cases

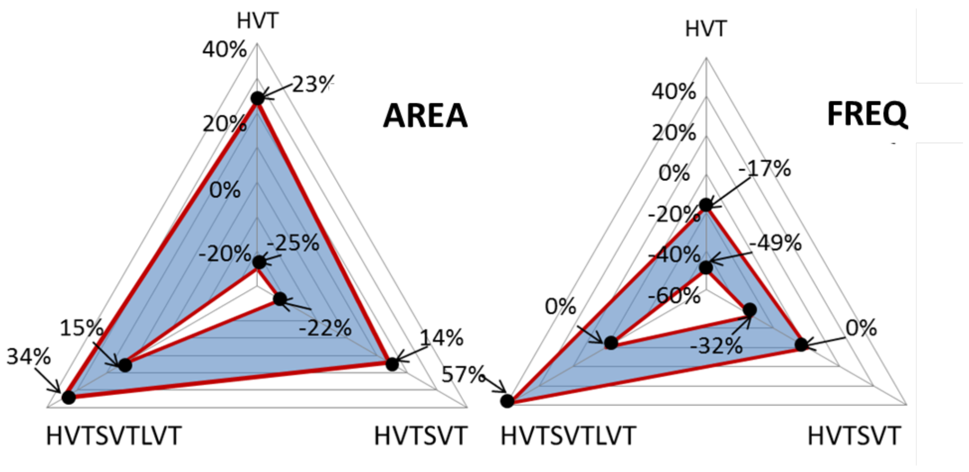


Figure 4.6: Area and frequency design space - 64 CLBs flat MSSN version

Figure 4.6 shows the variation of results in terms of area and speed achieved during the exploration, thus proving the existence of a quite wide design space also in this larger test-case. As an example, low-leakage HVT-only designs show an area range of ±25% compared to SVT reference, while the frequency range decreases from -17% to -49%. In the opposite case, with an HVT-SVT-LVT mix, the area can range from +15% to +34%, while the frequency can speed-up to ≈60% leveraging on LVT cell utilization, and paying in terms of power. Tables 4.11 and 4.12 show the details of the leakage variations for the various V_t-mixes: a decrease of one order of magnitude (from -86% to -90%) can be obtained in the HVT-only solution while, like in the smallest 16 CLBs test case, high-speed HVT-SVT-LVT implementations cause significant leakage penalties (from a minimum of +136% to a maximum of +400%).

V_t	Max Leak	% cells		
		HVT	SVT	LVT
H	-86.5%	100%	-	-
H,S	-25.7%	55%	45%	-
H,S,L	399.8%	17%	33%	50%

Table 4.11: Leakage power design space summary: maximum values in the 64 CLBs flat MSSN

V_t	Min Leak	% cells		
		HVT	SVT	LVT
H	-90.7%	100%	-	-
H,S	-66.1%	72%	28%	-
H,S,L	136.7%	51%	24%	25%

Table 4.12: Leakage power design space summary: minimum values in the 64 CLBs flat MSSN

The design-space exploration results of a 64 CLBs fully-bypassed MSSN resulted consistent with that of the flat MSSN version, and are shown in figure 4.7.

Compared to an SVT design, the obtained results are then in line with the values obtained for the 16 CLBs eFPGA, showing the scalability of the soft-core approach. From the various implementations of the 64 CLBs soft-core eFPGA was derived that the delay in crossing a flat 1024+1024 I/Os MSSN is only ≈1.5 that of the combinational CLB, since the CLB has a lots of synthesized multiplexers on the critical path, as required for implementation of the input crossbars (figure 3.4).

Since bypass enhancements are proven to cause penalties (see tables 4.6,

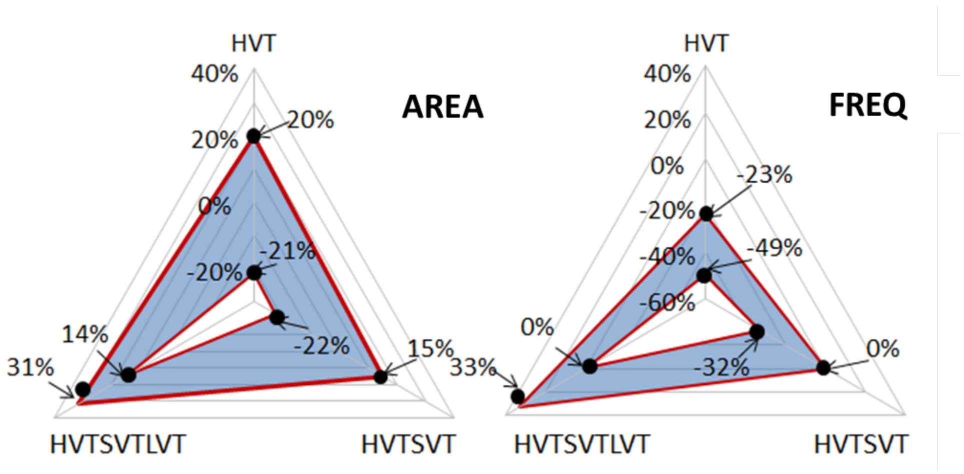


Figure 4.7: Area and frequency design space - 64 CLBs fully-bypassed MSSN version

4.7, 4.10), mainly due to the increased complexity of the switches supporting *U-turn* connections, next section will provide an analysis of the possible benefits of such enhancements on critical-path delay reduction in some target applications.

4.3 Application-Aware Analysis

While previous paragraph showed physical-only metrics, this paragraph provides an application-aware analysis considering different benchmarks mapped on the flat and on the fully-bypassed 64 CLBs implementations of table 4.10. Whereas for flat-MSSN eFPGA architecture the application frequency depends simply on the number of combinational CLBs in the critical path (since the network delay is mostly constant and equal to full-latency of the MSSN), in a bypass-enhanced MSSN eFPGA the effective working frequency depends on the exploitation of the bypass, and thus on the ability of the CAD flow to exploit the locality of the connections.

The place-and-route flow presented in section 3.2 allowed the implementation of a design aware of the real hierarchy of an MSSN eFPGA, thus optimizing the placement to exploit the locality. To perform an application-aware analysis, a set of benchmarks were used for performance evaluation. Such benchmarks were derived from MCNC (Microelectronics Center of North Carolina) circuits, already used in many FPGA analyses [32], [33], plus some additional hand-made designs, that allowed a more detailed analysis of certain arithmetic and logic designs (2 Multipliers, 2 Finite Impulse Response filters, 2 Fibonacci Linear Feedback Shift-Registers and an Adder).

For each test-case the total delay associated with the critical path was evaluated, composed of N_{CLB} combinational CLBs and N_L MSSN stages crossed, informations provided by the custom CAD tool in the *Critical Path Analysis Files*. Exploiting such informations, the critical path length was hence be expressed as:

$$Critical\_pathdelay = N_{CLB} \times \gamma + N_L \times \rho \tag{4.1}$$

where γ was the normalized CLB delay ($\gamma = 0.22$) and ρ the normalized delay associated with a single MSSN level ($\rho = 0.018$). As a normalization factor was adopted the target implementation period, observing that on average the values achieved are more or less the same for all implementations.

The obtained values are related to the MSSN-on-CLB delay ratio of ≈1.5 anticipated in the previous paragraph: in a M stages (with M=21 according to equation 2.3) such ratio results indeed:

$$\frac{MSSN\_delay}{CLB\_delay} = \frac{(M-1) \times \rho}{\gamma} = \frac{20 \times 0.018}{0.22} = 1.6\bar{3} \tag{4.2}$$

where the $(M-1)$ factor is due to the fact that the 1x2 input switches (figure 2.9) can be considered as empty switches, hence introducing negligible delay.

The histogram in figure 4.8 shows the difference in terms of normalized critical path delay between a flat and a bypass-enhanced version, referring to devices featuring the same implementation frequency. Benchmarks are ordered from left to right by the number of CLBs used. As shown by the *%Total_Gain* line, bypass exploitation results in an effective working frequency gain from ≈20% to 60%. Crossing graphs on figure 4.8, it should be observed that gain factors >20% were obtained for benchmarks with combinational paths featuring different levels of critical path depth (represented by the *#LUT_on_crit_path* line) from ≈60 down to less than 10 LUTs. This implies that bypass advantages are available in most of the applica-

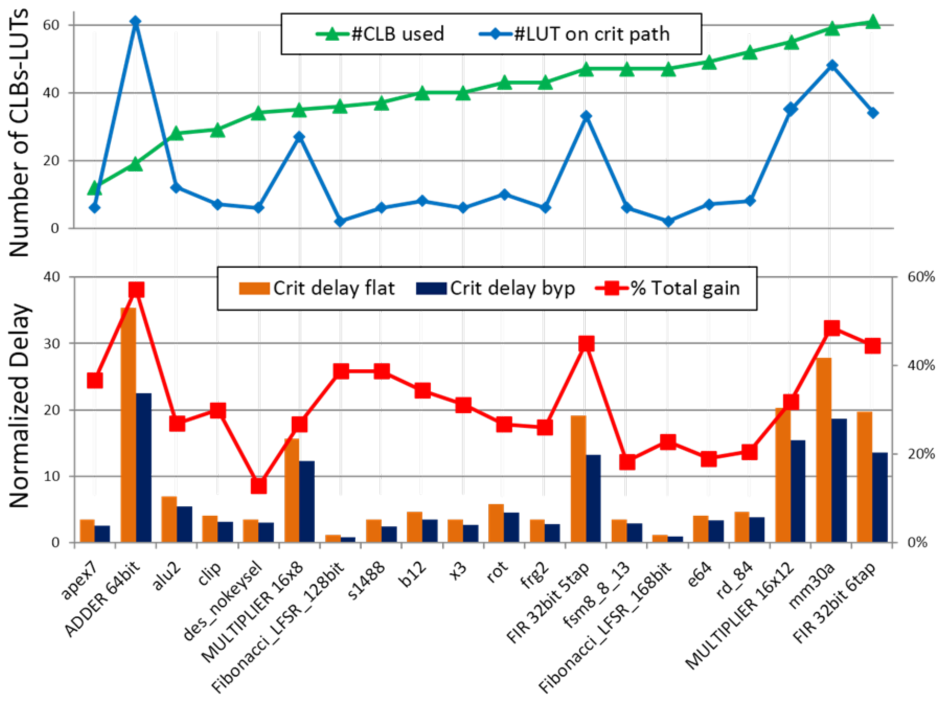


Figure 4.8: Effective working frequency gain

tions, with different levels of effectiveness which is in some way related to the application topology (locality, pipelining, ...).

One additional note: as previously observed in the full-flat approach, the weight of the CLBs is highly significant, since the ratio between MSSN and CLB delay is ≈1.5x. On optimizing the CLB slightly at circuit level, for example with optimized multiplexing structures for the crossbar or for the LUT, the benefit of the bypassed MSSN will be greater, thus increasing the overall performance. This kind of optimization does not change the overall fully-synthesizable approach, since it can be performed using tristate buffers or special multiplexers at a local level, where the performance can be estimated accurately. In this way the overall MSSN is again optimized in place-and-route step, where the tool can globally optimize the implementation, considering transitions, capacitances, on-chip variation and so on.

4.4 Computational Density Analysis

As demonstrated in the previous section, the area increase introduced by a full-bypass enhancement in a 64 CLB eFPGA can be repaid with an effective working frequency gain that varies from ≈20% to ≈60%, depending on the target application.

In order to better analyze this area-frequency trade-off, in this section a detailed computational density analysis is proposed: to do so, eFPGA implementations with the same area and different implementation frequency have been taken into account, considering the different technology options available (paragraph 4.2).

Such analysis was realized following the same methodology used for the MSSN architectural evaluations presented in paragraph 2.4. Hence in figure 4.9 for each area value the application frequency of a flat MSSN eFPGA was reported and compared to that achievable by a fully-bypassed eFPGA assuming different levels of bypass exploitation (i.e. 0%, 20%, 40% and 60%, according to the range of the gain on the effective frequency retrieved from figure 4.8).

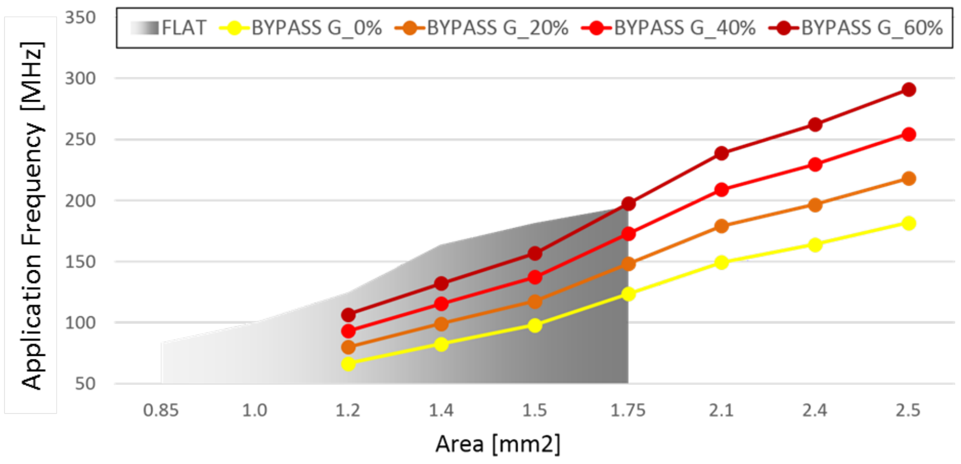


Figure 4.9: Flat MSSN versus Fully-bypassed: frequency vs. area

Figure 4.9 shows that for low-area budget (<1.75 mm^2) a flat architecture is able to guarantee frequency-on-field even higher than a fully-bypassed one with an application able to achieve a 60% gain thanks to the bypass exploitation. The advantage of a fully-bypassed MSSN eFPGA becomes straightforward when high performance is required: for an area budget >2.1 mm^2 applications gaining >40% thanks to bypass exploitation ($G\_40\%$) allows one to attain effective frequencies beyond 200 MHz.

Computational efficiency has been analyzed in figure 4.10. The frequency/area ratio was taken as a figure of merit, which is proportional to the throughput/area ratio and is application-independent.

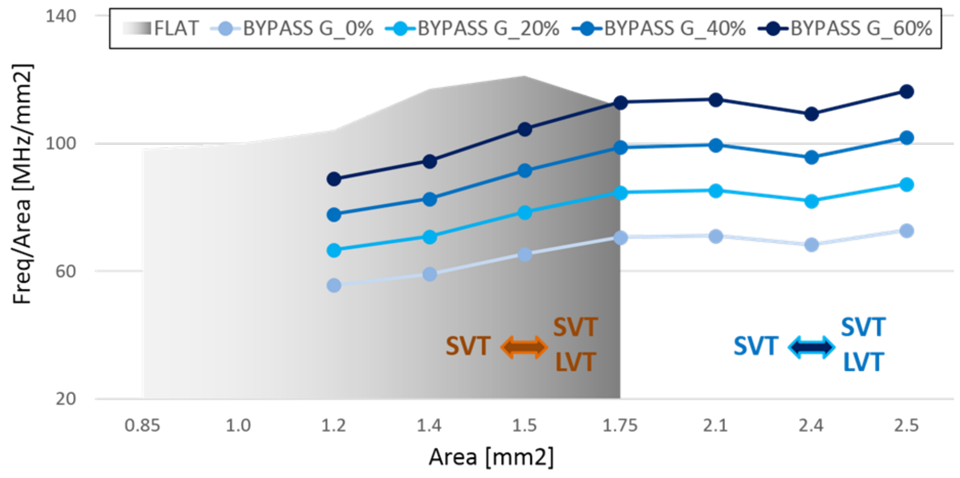


Figure 4.10: Flat MSSN versus Fully-bypassed: computational density vs. area

Hence in figure 4.10 the frequency values of figure 4.9 have been normalized with respect to the correspondent area values (horizontal axis): the obtained results show that both in flat and in bypassed MSSN styles frequency/area ratio has a growing trend for low area values, with a peak near to the physical implementation limit of each architecture. In the bypassed eFPGA this ratio roughly saturates for area values >1.75 mm^2. This means that, even when forcing the implementation frequency, the computational density is not significantly impacted.

One further detail: the small reduction of the frequency/area ratio both in the flat (from 1.5 mm^2 to 1.75 mm^2) and in the bypass-enhanced (from 2.1 mm^2 to 2.4 mm^2) eFPGA is due to the introduction of LVT cells in the design. As a matter of fact below that threshold both solutions were implemented with SVT-only cells, since HVT and HVT-SVT designs feature worst frequency performances at the same area values. As regards HVT-SVT-LVT designs, the utilization of Low Voltage Threshold standard cells was implemented with a smoothly different synthesis approach, in which LVT libraries were introduced after a first, preliminary, HVT-SVT only implementation (*incremental synthesis*).

Chapter 5

eFPGA for Smart Power

This chapter carries on the analysis of the eFPGA *portability* and *flexibility* features: as anticipated, the portability of the device to different technology nodes was proven realizing implementations on STMicroelectronics BCD9s technology [34], within which an analysis of the design space available allowed the evaluation of the flexibility of the template.

BCD is a family of Smart Power processes that combine Bipolar, CMOS and DMOS transistors, thus proving the marriage of three fundamental functions (figure 5.1): power devices and their drive circuits, sensing/protection and conditioning analog circuits, and digital logic. DMOS transistors are responsible for the first function, since they can handle high-voltage and/or high-currents working in fast switching conditions. Sensors together with local feedbacks for protection and analog signal conditioning are typically accomplished using high-performance bipolar circuits. CMOS digital control is responsible for the interaction among all those components.

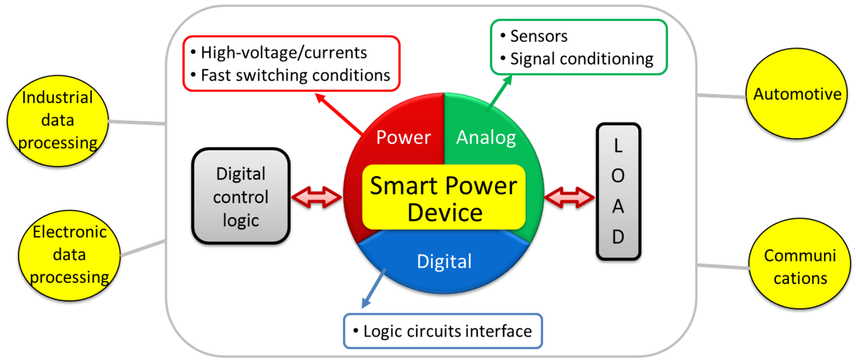


Figure 5.1: Smart Power device overview

BCD technology has been chosen for two main reasons:

- compared to the commercially available leading CMOS nodes, state-of-the-art BCD technology features very different characteristics, such as metal stacks with few layers (typically 3-4) and 3-4x wider transistor's pitch: a possible compatibility with this technology would hence further strengthen the portability features of the soft-core template;

- despite still focused toward power and analog worlds, Smart Power technologies are nowadays trying to exploit technology scaling implementing systems featuring complex digital logic, thus answering the demand for smarter devices: this need of *more than smart* devices is hence challenging the usage of programmable hardware on BCD world, as shown in figure 5.2.

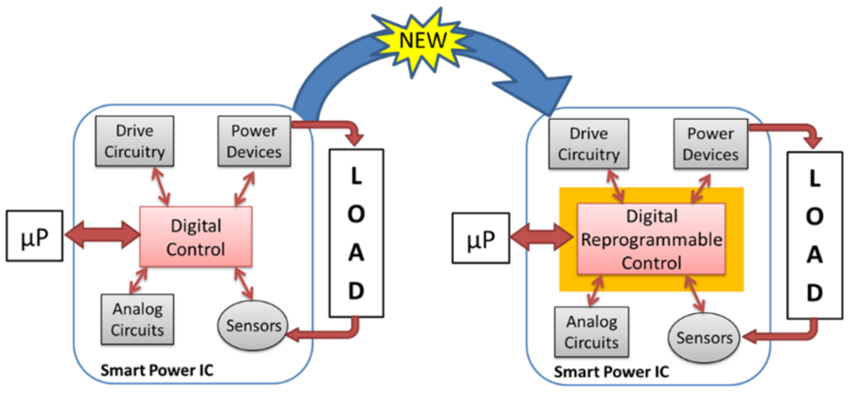


Figure 5.2: A Smart Power IC showing reprogrammability enhancement

The Smart Power world is thus demanding "intelligence" to improve efficiency. As an example, customized policies are used to improve power conversion efficiency on LED light drivers or solar panels grid [35]. On the application side this scenario is very similar to that of the introduction of reconfigurable devices, proposed on the CMOS world to couple computational efficiency, hardware programmability and NRE costs reduction of ASICs, as descripted in the Introduction of this thesis work.

Moving from the CMOS implementations and analyzes performed in chapter 4 to the BCD world, two preliminary points merits considerations:

- Smart Power ICs are dominated by analog and power IPs, which typically define the floorplan of the chip and the shape of the region reserved to digital logic;

- analog and power applications typically require low complexity digital logic (in the order of KGates) and low operating frequencies (up to few tens of MHz).

For that, the eFPGA paradigm can be exploited on this application scenario and the soft-core approach appears well suited to allow the IPs to accomplish specific implementation needs.

In this chapter, after an evaluation of the design-space obtained through BCD9s implementation trials, a set of benchmarking applications has been selected to show the potential of the eFPGA programmable solution on different smart power fields: signal modulation, motion control and power management.

5.1 Implementation Results

The soft-core eFPGA has been implemented on STM BCD9s, a 0.11 μm technology with 4 metal layers at 1.55V. The metal stack available in BCD9s may harden the implementation of programmable devices, which are traditionally *routing hungry*: this required limiting the size of the template.

For that, the test-case considered is composed of 16 CLBs (192 LUT 4:1), corresponding to $\approx$1K equivalent gates, choice which seems to fit anyhow the needs of this kind of applications. The resulting interconnect is hence a 256+256 I/Os radix-2 MSSN, with 192+192 I/O pins dedicated to CLB connections (16 CLBs x 12 I/Os each), and 64+64 I/Os used as primary eFPGA I/Os (figure 5.3).

The resulting device has been synthesized for either area or speed optimization using Synopsys Design Compiler, and results are reported in table 5.1. Again, implementation target frequency reported is referred to the *time-to-fly* between two CLBs configured with synchronous output. As for the CMOS implementations described in section 4.1, two MSSN topologies have been evaluated: a flat MSSN and a MSSN with complete bypass structure to realize a fully-hierarchical network.

In the latter, as shown in table 5.3, the complexity introduced by the *U-switches* implies an increase in terms of area (+14% and +31% in max-speed and min-area configurations respectively) lower than that obtained

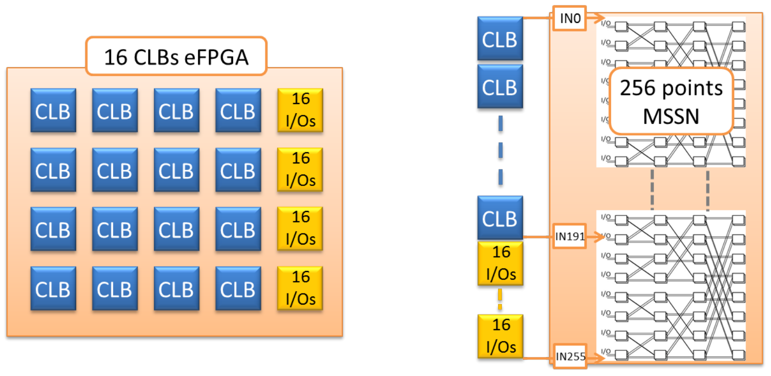


Figure 5.3: eFPGA featuring 16 CLBs and a 256 points MSSN

Flat MSSN			
	Max Speed	Min Area	%
Area $[mm^2]$	1.17	1.76	53.9%
Frequency $[MHz]$	100	50	100%
Leakage $[\mu W]$	401	170	135.9%
% Bitcells on total area	28.4%	43.8%	-35.2%
% MSSN area	52.8%	46.4%	13.8%
% Bitcells on MSSN area	32.5%	56.4%	-42.8%

Table 5.1: BCD9s 0.11 μm 16 CLBs Flat MSSN eFPGA post-synthesis summary

Fully-bypassed MSSN			
	Max Speed	Min Area	%
Area $[mm^2]$	1.34	1	34.0%
Frequency $[MHz]$	86	50	72.0%
Leakage $[\mu W]$	402	232	73.3%
% Bitcells on total area	34.7%	45.5%	-23.7%
% MSSN area	59.1%	57.5%	2.8%
% Bitcells on MSSN area	42.4%	56.8%	-25.4%

Table 5.2: BCD9s 0.11 μm 16 CLBs Flat MSSN eFPGA post-synthesis summary

in CMOS implementations; as regards leakage and frequency, the only significant variation can be seen comparing the minimum area configurations (+36%).

Even though BCD9s technology does not features as many technology flavours as CMOS 65nm (the different V_t mixes as described in chapter 4), tables 5.1 and 5.2 highlight a quite different scenario when optimizing the device per speed or per area, thus revealing a possible design-space for

	Fully-bypassed versus Flat MSSN	
	Max Speed	Min Area
Area $[mm^2]$	14.5%	31.5%
Frequency $[MHz]$	-14%	-
Leakage $[\mu W]$	0.25%	36.4%

Table 5.3: 16 CLBs eFPGA comparison: fully bypassed versus flat MSSN

application-specific optimizations. As a matter of fact, within both the two topologies - flat or bypassed - there is a variation in terms of area (+53% in the first and +34% in the latter), frequency (100% and 72% increase respectively) and leakage (+135% and +73%). As regards the area occupied by the MSSN, the obtained results - between 46% and 59% - prove substantially in line with CMOS implementations, resulting lower than the typical values for reprogrammable devices.

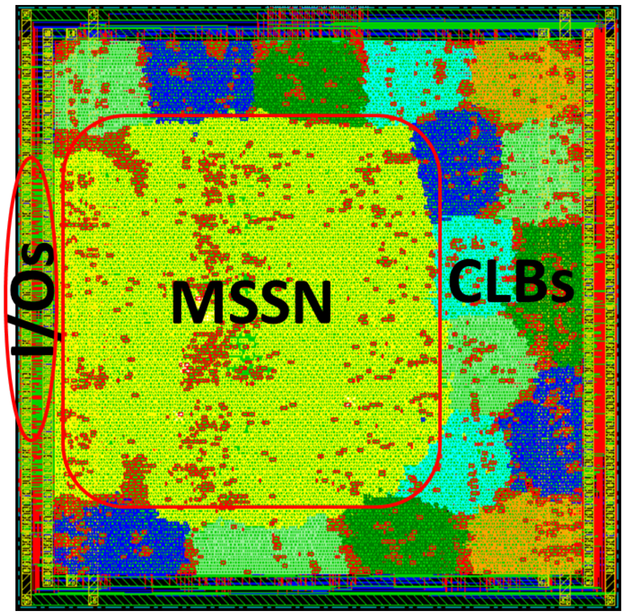


Figure 5.4: BCD9s 0.11 μm floorplan of a 16 CLBs eFPGA

Place-and-route steps were realized in a couple of relevant cases using Cadence Encounter, following the same "worst-case" approach adopted in the previous CMOS implementations, without any violations being reported: the obtained results are comparable with the post-synthesis ones (average 5% area increase). Figure 5.4 shows the resulting layout: compared to the CMOS version (depicted in figure 4.2), it shows a different placement of the 16 CLBs, due to the different routing impact which heavily affects cell distribution during automatic placement.

The feasibility of the device on the target technology was hence verified, proving the portability across different technology nodes.

5.2 Application Analysis

This section presents the results of benchmarking activity carried out mapping on the eFPGA a set of representative smart power applications, showing the advantages provided by reconfigurability. For each one, given the maximum working frequency obtained in the previous section - both in max-speed and min-area configuration - an estimation of the effective operating frequency has been made taking into account the number of combinational CLBs in the critical path, information obtained thanks to the CAD flow support presented in section 3.2.

Even though the bypass-enhancement on MSSN can provide benefits in terms of effective working frequency (even up to +50%, depending on target application), a flat MSSN eFPGA was taken into consideration (table 5.1). As a matter of fact this kind of solution, in which the network delay is mostly constant and equal to the full-latency of the MSSN network, has been found to be able to fulfill the target performance requirements.

Table 5.4 shows the characterization of the chosen applications. For each of them synthesis trials have been implemented using BCD9s standard cells flows, obtaining the area values ($Area_{app}$) reported in the first column of the table. Starting from these area values, and given the area of a NAND gate (10.09 μm^2), the dimension of each application has been determined in terms of BCD equivalent gates (*App_Eq.Gates*, right column in table 5.4), so that:

$$App\_Eq.Gates = \frac{Area_{app}}{Area_{NAND}} \tag{5.1}$$

Each application was then mapped on the 16 CLB eFPGA, thanks to the CAD flow support. The results of the LUT mapping steps are reported in table 5.5. As depicted, besides the LUT sizes available in the CLB architecture (figure 3.2, LUT6:1, LUT5:1 and LUT4:1), in the mapping opera-

	BCD9s Area [μm^2]	BCD9s Equivalent Gates
SigmaDelta	1327	131.5
Piezo Control	242	24.0
9 Ch. Pz. Control	2446	242.4
Power Sequencer	1642	162.7
Stepper Full 2 Ph.	1829	181.3
Stepper Full 1 Ph.	1827	181.1
Stepper Half	1972	195.4

Table 5.4: Post-synthesis implementation of the target applications - BCD9s standard cells flow

tion VTR tool also uses LUTs of smaller size (LUT2:1 and LUT3:1): in these cases, however, a whole LUT4:1 will be employed, being the smallest size available on the logic block. In the *Eq.LUT4:1* column of table 5.6 is re-

	LUT2:1	LUT3:1	LUT4:1	LUT5:1	LUT6:1
SigmaDelta	3	11	6	3	9
Piezo Control	6	0	1	0	1
9 Ch. Pz. Control	4	3	14	13	17
Power Sequencer	4	2	5	4	17
Stepper Full 2 Ph.	2	6	6	9	9
Stepper Full 1 Ph.	2	6	6	7	11
Stepper Half	3	7	6	7	12

Table 5.5: LUT mapping results - 16 CLBs flat MSSN eFPGA

ported the total number of LUTs used for each application, expressed as equivalent LUT4:1 and calculated considering a LUT5:1 as 2×LUT4:1 and a LUT6:1 as 4×LUT4:1. According to these values, since in the 16 CLBs eFPGA 192 4:1 LUTs are available, the percentage of equivalent 4:1 LUTs used was then calculated for each application (*%LUT_used* column in table 5.6). The obtained values were also used, together with the *App_Eq.Gates* results of table 5.4, to estimate the total eFPGA computational efficiency, expressed in BCD9s equivalent gates dimensions (*eFPGA_Eq.Gates*), so that:

$$eFPGA\_Eq.Gates = \frac{App\_Eq.Gates}{\%LUT\_used} * 100 \tag{5.2}$$

As shown in the right column of table 5.6, according to these estimations the 16 CLBs flat MSSN eFPGA corresponds on average to ≈4K BCD9s equivalent gates: such value can be seen as the maximum complexity (expressed in BCD9s equivalent gates) allowed for an application that has to be mapped on the eFPGA.

	Eq. LUT4:1	% LUT Used	eFPGA Eq. Gates
SigmaDelta	62	32.3%	407.3
Piezo Control	11	5.7%	418.6
9 Ch. Pz. Control	115	59.9%	404.7
Power Sequencer	87	45.3%	359.1
Stepper Full 2 Ph.	68	35.4%	511.8
Stepper Full 1 Ph.	72	37.5%	482.9
Stepper Half	78	40.6%	481.1

Table 5.6: eFPGA mapping results - 16 CLBs flat MSSN eFPGA

Beside the total amount of logic (equivalent LUT4:1) required, table 5.7 shows the total number of CLB used, given after the packing operation. Such values are expected to be aligned, since both related to resource utilization; the noticeable difference is due to the optimizations performed by packing and placement steps in the implementation flow.

	#CLB	#LUT	Crit. Path	Freq. [MHz]	
				Max Speed	Min Area
SigmaDelta	8	62	9	17	11
Piezo Control	2	11	2	76	50
9 Ch. Pz. Control	15	115	4	38	25
Power Sequencer	9	87	6	25	16
Stepper Full 2 Ph.	8	59	5	30	20
Stepper Full 1 Ph.	9	72	4	38	25
Stepper Half	9	78	5	30	20

Table 5.7: Resource usage and effective working frequency - 16 CLBs flat MSSN eFPGA

5.2.1 Sigma-Delta Modulation

As first test-case, a second-order multi-bit $\Sigma - \Delta$ modulator has been taken from [36], where it is used to create control signals for a Digital Pulse Width Modulation (DPWM) in a high-frequency DC-DC converter. Typically, multi-bit feature is used to provide high effective DPWM resolution and noise reduction, introducing on the other side low switching frequencies. In [36], a second order architecture is chosen since it allows achieving working frequencies exceeding 10 MHz even though a high resolution (10 bits) signal is taken as input.

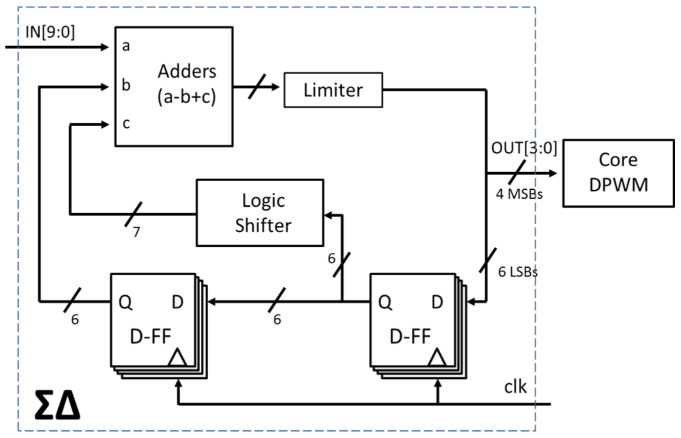


Figure 5.5: Sigma-Delta modulator

As depicted in figure 5.5, a 4 bit output signal is used to drive the core DPWM, and a 6 bit truncation error is processed internally through a set of delay flip-flops, a 2x multiplexer (7 bit logic shifter) and two adders. Mapping results reported in figure 5.6 show that the performances typi-

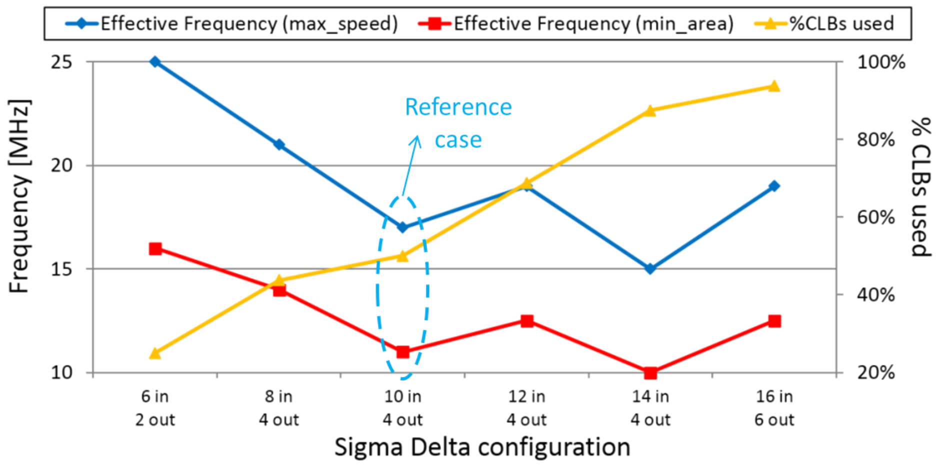


Figure 5.6: Effective working frequency and resource usage of a set of $\Sigma - \Delta$ modulators

cally requested in terms of effective frequency - beyond 10 MHz - are satisfied both in *max_speed* mode (17 MHz) and in *min_area* configuration (11 MHz). Resources utilization values, both in terms of number of CLBs (8) and equivalent LUT 4:1 (62 on 192 available), prove the existence of considerable margins for possible enhancements. For that, a set of different modulators architectures were mapped in order to show how the variation of input and output bits (and consequently of the internal error feedback) can lead to significant different performances both in terms of resource usage - percentage of CLBs used - and effective working frequency (both in *max_speed* and *min_area*). Figure 5.6 shows that a reduction of the modulator dimensions (with respect to the reference case of figure 5.5) down to 6 inputs and 2 outputs allows reaching working frequencies up to 25 MHz, together with low area occupancy (25% - 4 CLBs). On the other side more complex architectures - e.g. featuring up to 16 input bits and 6 output bits - can be mapped with a resource usage beyond 90% (15 CLBs). This configuration, in which the working frequency is between 10 MHz and 20 MHz, represents an upper limit for the chosen eFPGA template, since further increment of resolution bits does not fit the 16 CLBs.

5.2.2 Power Management

Digital control in power management applications is typically realized in the form of a Finite State Machine (FSM): this approach allows generating signals in order to coordinate all the components of the system.

In this scenario, as first benchmark the control logic for a nano-power management IC for piezoelectric energy harvesting applications [37] has been mapped on the eFPGA. The architecture of the system is shown in figure 5.7: a FSM featuring 4 internal states is used to step over the different phases of an energy extraction cycle through the communication with the piezo input stage and the AC-DC conversion stage. As shown in table 5.7, effective working frequencies between 50 and 76 MHz can be achieved, together with a low computational logic usage (2 CLBs and 11 LUTs).

The flexibility introduced implementing such kind of control using a reconfigurable device can allow changing system specifications on-the-field, for example varying the number of internal states or modifying their behaviour. Since one of the typical challenges in power management applications is represented by the interaction between many voltage sources, the previous control logic was adapted for a multi-source energy harvester IC [38]. Here, thanks to the addition of more than 20 internal states (figure 5.8),

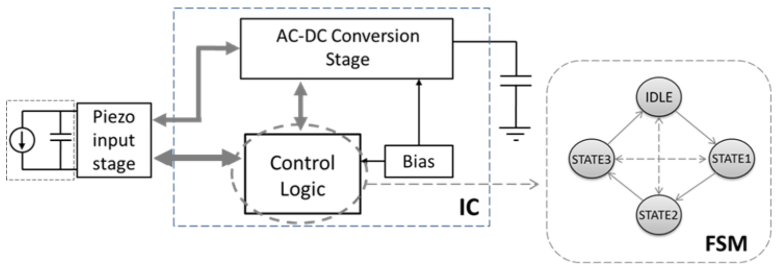


Figure 5.7: Block diagram of the single-source power management IC

the control logic act as arbiter to manage up to 9 different AC-DC channels (e.g. piezoelectric transducers) . As shown in table 5.7, this enhancement

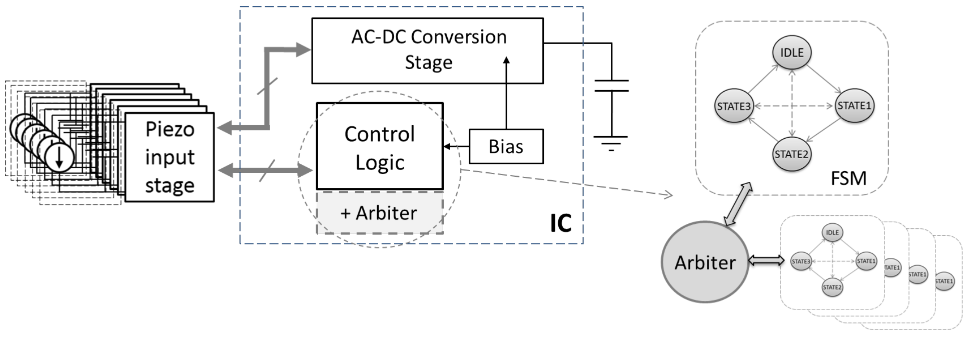


Figure 5.8: Block diagram of the multi-source power management IC

implies a significant increase both in terms of number of CLBs (15) and equivalent LUTs (115), while the frequency decreased down to 25 and 38 MHz, for respectively area-optimized and speed-optimized devices.

As additional benchmark, still in the field of power management, a digital power sequencer slice taken from [39] has been mapped, featuring 9 internal FSM states. As depicted in figure 5.9, power sequencers are used in cascade to handle power up and down order of multiple supply voltages, with the aim, for example, to coordinate all the components of an audio system. Mapping results of a single slice are reported in table 5.7, showing a range of target frequency between 16 and 25 MHz, together with a resource utilization of ≈50% (9 CLBs and 87 LUTs). Since power sequencers performances are strictly dependent on the slow response timing of the power supplies (typically up to hundreds of μseconds, hence tens of KHz), the obtained frequencies values plenty satisfy typical speed requirements.

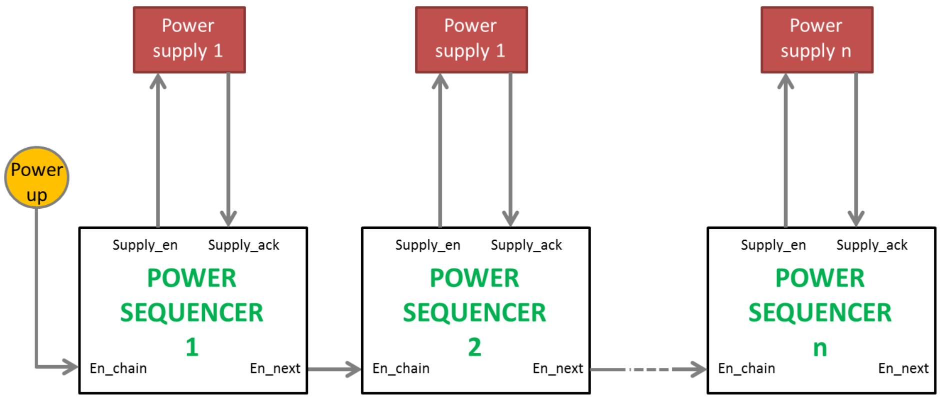


Figure 5.9: Block diagram of a cascade of power sequencers

5.2.3 Motion Control

Even in the field of motion control a digital controller able to handle external requests or driving internal signals is typically required. As shown in figure 5.10, such controllers are pure digital logic devices, based on FSMs typically used to describe system functionalities.

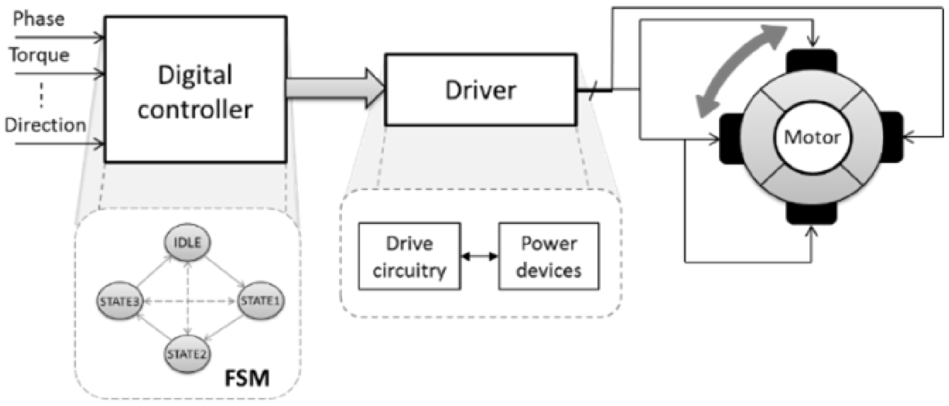


Figure 5.10: Open-loop motor control

As an example of motor control, the code of a stepper motor driver has been taken from [39]. Stepper motors are a kind of electric motor that can be driven both in open and closed loop, requiring a quite simple driving circuitry. The control mapped on the eFPGA is able to control any standard 4-wires open-loop stepper motor with an internal FSM featuring 4 states, using a Two-Phase Full Step excitation mode. Since most of the steppers are designed for fine resolution at low-speeds, the typical working frequency of this kind of devices rarely exceed few dozen of MHz. Table 5.7 shows that frequencies between 20 and 30 MHz can be reached, depending on the required optimization.

As for the $\Sigma - \Delta$ modulator, the total amount of logic required is less than 50%. This means that several enhancements - e.g. wiring configuration, physical size, internal timeout resolution or number of internal steps - can be eventually added to improve performances.

As an example two further topologies of stepper motor control were implemented changing the excitation mode - hence the internal FSM states - obtaining a One-Phase Full Step and a Half Step configuration. Table 5.7 shows that such changes imply an increase in terms of both CLBs used (beyond 50%) and number of equivalent LUTs (72 and 78 respectively), together with a small improvement in terms of working frequency in One Phase configuration (up to 38 MHz).

Conclusions

In this thesis a soft-core eFPGA template has been presented, as a particular test-case for a more general model of synthesizable embedded programmable logic device. The key point of the proposed design was the adoption of a Multi-Stage Switching Network (MSSN) as the foundation for the routing structure: featuring butterfly topology, such network has allowed to work with a predictable congestion-free solution. This has represented an added value for the soft-core design, ensuring that modifications of the eFPGA (e.g. I/Os and CLBs number and their internal structure) did not impact on connectivity properties. As an added value, the particular modular structure of the designed MSSN - made of simple switch blocks organized in stages - ensured its synthesizability, further strengthening the soft-core features of the overall device.

Thanks to its symmetry properties, two different versions of the MSSN were proposed: a basic flat and a folded bypass-enhanced one, realized thanks to the addition of particular *U-turn* connections to the MSSN switch blocks. In the latter, the exploitation of the hierarchical features of the MSSN allowed the achievement of faster connectivity - hence higher working frequencies - at the expense of a restrained area increase. This area-frequency trade-off was analyzed, taking into account MSSNs featuring different bypass-enhanced topologies (i.e. flat, folded fully-bypassed and folded half-bypassed) and different *radix* (i.e. switch blocks dimensions). Results highlighted that the best architectural choice was application-dependent, since related to the characteristics (e.g. number of connections and their locality features) of the required connectivity: this further strengthened the parametric flexible structure of the adopted MSSN, designed thanks to the realization of a custom C-language based software support. Such dedicated tool was then integrated in a more complete eFPGA CAD flow support, realized leveraging on the availability of VTR open source tool for FPGAs, thus making it MSSN-aware.

In order to prove the portability across different technologies, the eFPGA template was then implemented in STMicroelectronics CMOS65nm and BCD9s 0.11μm. Both technologies showed the existence of a significant design space, exploitable to optimize the eFPGA on very different application scenarios.

In CMOS 65nm, the design space exploration was realized targeting two different eFPGA dimensions: a 16 and a 64 CLBs respectively. In the first test-case the obtained results highlighted that the area could vary up to +40% with respect to a reference design implemented using SVT-only standard cells, while the speed range was between -40% to +20% leveraging on high-speed LVT cells. In the larger eFPGA, area could vary from -25% to +35% and frequency from -50% to +57%, always with respect to an SVT-only reference design and exploiting the availability of LVT standard cells. In both cases, on the other hand, it was possible to save up to +90% of leakage power consumption thanks to the utilization of HVT cells. The performed CMOS synthesis trials were also used for an analysis of the computational density of the device: results showed that the advantage of a fully-bypassed MSSN eFPGA became significant when the area budget did not represent a constraint, allowing an effective frequency improvement of 30-50% depending on bypass exploitation. On the contrary a flat MSSN architecture was able to guarantee the best frequency-on-field for low-area budget or applications in which the bypass exploitation resulted lower than 20%.

As regards the implementations targeting BCD9s 0.11μm, the design space analysis was realized only on a 16 CLBs eFPGA, size that resulted to best fit the size of typical BCD Smart Power applications. Even though this technology featured only 4 metal layers, the analysis proved the existence of a significant design space. According to different optimizations, in a flat MSSN eFPGA area, frequency and leakage presented variations of 50%, 100% and 130% respectively, values that resulted slightly smaller in a design featuring a fully-bypassed MSSN (+30%, 72% and 73%). In order to complete the analysis, the application scenario was analyzed benchmarking the device with a set of significant smart power applications: results obtained proved the feasibility of the approach, which allowed matching typical performance requirements. For each target application was also highlighted how the flexibility provided by the usage of a reconfigurable support could be exploited to provide added values to end-user applications.

Bibliography

[1] Man-Ho Ho, Yan-Qing Ai, TC-P Chau, Steve CL Yuen, Chiu-Sing Choy, Philip HW Leong, and Kong-Pang Pun. Architecture and design flow for a highly efficient structured asic. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 21(3):424–433, 2013.

[2] Davide Rossi, Claudio Mucci, Matteo Pizzotti, Luca Perugini, Roberto Canegallo, and Roberto Guerrieri. Multicore signal processing platform with heterogeneous configurable hardware accelerators. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2013.

[3] André DeHon. The density advantage of configurable computing. *Computer*, 33(4):41–49, 2000.

[4] Steven JE Wilton, Noha Kafafi, James CH Wu, Kimberly A Bozman, Victor O Aken'Ova, and Resve Saleh. Design considerations for soft embedded programmable logic cores. *Solid-State Circuits, IEEE Journal of*, 40(2):485–497, 2005.

[5] Menta. Embedded programmable logic. $http://www.menta.fr/efpga\_core\_ip.html$, 2014.

[6] Ketan Padalia, Ryan Fung, Mark Bourgeault, Aaron Egier, and Jonathan Rose. Automatic transistor and physical design of fpga tiles from an architectural specification. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 164–172. ACM, 2003.

[7] Alexander Brant and Guy GF Lemieux. Zuma: An open fpga overlay architecture. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 93–96. IEEE, 2012.

[8] Matteo Cuppini, Eleonora Franchi Scarselli, and Claudio Mucci. Design-space exploration of an efpga soft-core based on multi-stages switching networks. In *Ph. D. Research in Microelectronics and Electronics (PRIME), 2013 9th Conference on*, pages 133–136. IEEE, 2013.

[9] Matteo Cuppini, Eleonora Franchi Scarselli, Claudio Mucci, and Roberto Canegallo. Soft-core efpga for smart power applications. In *System-on-Chip (SoC), 2014 International Symposium on*, pages 1–4. IEEE, 2014.

[10] Ian Kuon, Russell Tessier, and Jonathan Rose. Fpga architecture: Survey and challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253, 2008.

[11] Guy Lemieux and David Lewis. Circuit design of routing switches. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 19–28. ACM, 2002.

[12] Guy Lemieux, Edmund Lee, Marvin Tom, and Anthony Yu. Directional and single-driver wires in fpga interconnect. In *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*, pages 41–48. IEEE, 2004.

[13] Deming Chen, Jason Cong, and Peichen Pan. Fpga design automation: A survey. *Foundations and Trends in Electronic Design Automation*, 1(3):139–169, 2006.

[14] Umer Farooq, Zied Marrakchi, and Habib Mehrez. *Tree-Based Heterogeneous FPGA Architectures*. Springer, 2012.

[15] William James Dally and Brian Patrick Towles. *Principles and practices of interconnection networks*. Elsevier, 2004.

[16] Robert J McMillen. A survey of interconnection networks. In *Broadband switching*, pages 146–154. IEEE Computer Society Press, 1991.

[17] Tse-yun Feng. A survey of interconnection networks. *Computer*, 14(12):12–27, 1981.

[18] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[19] VE Beneš. On rearrangeable three-stage connecting networks. *Bell System Technical Journal*, 41(5):1481–1492, 1962.

[20] John Kim, William J Dally, and Dennis Abts. Flattened butterfly: a cost-efficient topology for high-radix networks. *ACM SIGARCH Computer Architecture News*, 35(2):126–137, 2007.

[21] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. Network flows: theory, algorithms, and applications. 1993.

[22] Synopsys. Design compiler tool in graphical mode. http://www.synopsys.com, 2014.

[23] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeffrey Goeders, Andrew Somerville, Kenneth B Kent, Peter Jamieson, and Jason Anderson. The vtr project: architecture and cad for fpgas from verilog to routing. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 77–86. ACM, 2012.

[24] Frederic Reblewski and Olivier LePape. Reconfigurable integrated circuit with a scalable architecture, July 15 2003. US Patent 6,594,810.

[25] Dale Wong. Interconnection network for a field programmable gate array, February 17 2004. US Patent 6,693,456.

[26] Cheng C Wang, Fang-Li Yuan, Henry Chen, and D Markovic. A 1.1 gops/mw fpga chip with hierarchical interconnect fabric. In *VLSI Circuits (VLSIC), 2011 Symposium on*, pages 136–137. IEEE, 2011.

[27] Peter Jamieson, Kenneth B Kent, Farnaz Gharibian, and Lesley Shannon. Odin ii-an open-source verilog hdl synthesis tool for cad research. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 149–156. IEEE, 2010.

[28] A Mishchenko et al. Abc: A system for sequential synthesis and verification. *URL http://www. eecs. berkeley. edu/˜ alanmi/abc*, 2007.

[29] Jason Luu, Jason Helge Anderson, and Jonathan Scott Rose. Architecture description and packing for logic blocks with hierarchy, modes

and complex interconnect. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 227–236. ACM, 2011.

[30] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for deep-submicron FPGAs*. Kluwer Academic Publishers, 1999.

[31] Cadence. Encounter digital implementation systems. http://www.cadence.com, 2014.

[32] David Grant, Chris Wang, and Guy GF Lemieux. A cad framework for malibu: an fpga with time-multiplexed coarse-grained elements. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 123–132. ACM, 2011.

[33] VC Aken'Ova, Guy Lemieux, and Resve Saleh. An improved" soft" efpga design and implementation strategy. In *Custom Integrated Circuits Conference, 2005. Proceedings of the IEEE 2005*, pages 179–182. IEEE, 2005.

[34] B Murari, F Bertotti, and GA VIGNOLA. Smart power ics, 1996.

[35] Datasheet STLUX385A. Digital controller for power and lighting conversion. http://www.st.com, 2014.

[36] Zdravko Lukic, Nabeel Rahman, and Aleksandar Prodic. Multibit σ–pwm digital controller ic for dc–dc converters operating at switching frequencies beyond 10 mhz. *Power Electronics, IEEE Transactions on*, 22(5):1693–1707, 2007.

[37] M Dini, M Filippi, M Tartagni, and A Romani. A nano-power power management ic for piezoelectric energy harvesting applications. In *Ph. D. Research in Microelectronics and Electronics (PRIME), 2013 9th Conference on*, pages 269–272. IEEE, 2013.

[38] M Dini, M Filippi, A Romani, V Bottarel, G Ricotti, and M Tartagni. A nano-power energy harvesting ic for arrays of piezoelectric transducers. In *SPIE Microtechnologies*, pages 87631O–87631O. International Society for Optics and Photonics, 2013.

[39] Open Cores. website. http://www.opencores.org, 2014.