Enhanced FPGA Architecture and CAD Flow for Efficient Runtime Hardware Reconfiguration

Christophe Huriaux

ANNÉE 2015

UNIVERSITÉ DE RENNES 1

ueb

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Traitement du Signal et Télécommunications

École doctorale MATISSE

présentée par

Christophe Huriaux

préparée à l'unité de recherche IRISA (UMR 6074)
Institut de Recherche en Informatique et Systèmes Aléatoires
École Nationale Supérieure de Sciences Appliquées et de Technologie

---

Enhanced FPGA Architecture and CAD Flow for Efficient Runtime Hardware Reconfiguration

Thèse soutenue à Lannion
le 2 Décembre 2015

devant le jury composé de :

Lorena Anghel
Professeur, Grenoble INP Phelma, TIMA
Examinateur

Antoine Courtay
Maître de conférence, Université de Rennes 1, IRISA
Co-directeur de thèse

Guy Lemieux
Professeur, University of British Columbia, Canada
Rapporteur

Philippe Millet
Thales Resarch and Technology
Examinateur

Lionel Torres
Professeur, Université de Montpellier 2, LIRMM
Rapporteur

Russell Tessier
Professeur, University of Massachusetts, État-Unis
Examinateur

Olivier Sentieys
Directeur de recherche, Inria
Directeur de thèse

RÉSUMÉ

Avec l'introduction de capacités d'auto-reconfiguration dans les FPGAs (*Field-Programmable Gate Arrays*), de nouveaux domaines de recherche ont émergé autour du placement dynamique de modules matériels à l'exécution. Les architectures commerciales disponibles et les outils ont évolué pour fournir des flots de reconfiguration partielle aboutis qui sont toutefois limités dans leurs fonctionnalités. L'objectif principal des travaux de cette thèse est de proposer de nouvelles approches pour faciliter une gestion en ligne flexible d'accélérateurs matériels dans les FPGAs, à la fois au niveau de l'évolution de son architecture et de son flot de conception.

Les architectures de FPGA modernes les plus avancées comportent une profusion de blocs hétérogènes destinés à fournir des performances plus élevées aux applications : des mémoires RAM, des accélérateurs arithmétiques, des émetteurs-récepteurs rapides, etc. La diversité et l'hétérogénéité de ces blocs induit une difficulté plus importante pour les algorithmes de placement en réduisant le nombre de positionnements possibles pour une tâche matérielle donnée. Dans cette thèse, une amélioration du réseau d'interconnexion de l'architecture du FPGA est proposée afin de réduire les contraintes sur le placement des tâches. Des résultats expérimentaux menés sur cette architecture montrent une augmentation de la flexibilité de placement des tâches, au détriment du délai du chemin critique.

Deuxièmement, le concept de flux de configuration indépendant de sa position finale, le *Virtual Bit-Stream* (VBS), est proposé pour répondre à un problème majeur des mécanismes de relogement de tâche : le stockage d'une unique configuration de tâche pour réduire l'empreinte mémoire et pour accroître la réutilisabilité des modules matériels générés. Les *Virtual Bit-Streams* sont élaborés grâce à une représentation abstraite de l'architecture du FPGA cible, ce qui les rend indépendants de leur position finale sur la surface logique. Grâce au format de codage du VBS, le volume de données binaires ainsi créé pour représenter une tâche matérielle peut être jusqu'à un ordre de grandeur plus petit qu'un flux de configuration conventionnel équivalent. À partir de ce concept, un flot de conception dédié à la génération des VBS est détaillé, ainsi que le contrôleur responsable du décodage en ligne de ces données. Plusieurs implémentations matérielles de l'algorithme de décodage du VBS sont présentées, permettant d'atteindre des fréquences de reconfiguration similaires à celles des FPGA commerciaux.

Finalement, une méthode permettant de définir une zone reconfigurable dynamique dans le chemin de configuration d'un FPGA est proposée, ce qui permet un

placement de tâches à l'exécution dont la flexibilité va au-delà de la définition de zones de reconfiguration partielle lors de la phase de conception. Ce mécanisme, associé à d'autres améliorations proposées dans cette thèse, facilite le développement d'une architecture FPGA reconfigurable dynamiquement et capable de charger des tâches matérielles à l'exécution à partir d'un flux de configuration indépendant de la position finale, sur une surface logique flexible.

La plupart des méthodes et techniques décrites dans cette thèse ont fait l'objet d'une implantation dans le contexte du projet Européen FP7 *FlexTiles*, qui a pour but le développement d'une architecture multi-cœurs hétérogène reconfigurable dynamiquement, et embarquant un FPGA embarqué (eFPGA) offrant des capacités de chargement dynamique de tâches matérielles.

ABSTRACT

With the introduction of self-reconfiguration capabilities in Field-Programmable Gate Arrays (FPGA), new research topics have emerged around the dynamic placement of hardware modules at runtime. The commercially available architectures and tools evolved to provide mature partial reconfiguration flows which are nonetheless limited in functionality. The main objective of this PhD thesis is to propose new approaches for FPGAs to seamlessly handle the flexible runtime loading of hardware accelerators, both from the architecture level and from the tool-flow point of view.

The most advanced modern FPGA architectures features profusion of different fixed-functions blocks aimed to provide the application with superior performance: Random-Access Memories (RAM), arithmetic accelerators, transceivers, ... The diversity and heterogeneity of these blocks occasion more difficulties for placement algorithms by reducing the number of target positions for a given hardware task. In this thesis, an interconnect architecture enhancement is proposed to lessen those constraints on task placement. Experimental results using this system show an increase on the task placement flexibility at the cost of a delay overhead.

Secondly, the concept of position-independent Virtual Bit-Streams (VBS) is proposed to answer a key problem in task relocation mechanisms: the storage of only one task configuration in order to reduce memory and to increase the re-usability of generated hardware modules. Virtual Bit-Streams are built upon an abstract representation of the target FPGA logic and routing architecture, which makes them dissociated from their final position on the fabric. Thanks to the intrinsic format of the VBS, the resulting binary configuration data can be up to an order of magnitude smaller than their raw equivalent. Based on this concept, a accompanying Computer-Aided Design (CAD) tool-flow to generate VBS is demonstrated, as well as the online controller responsible for their online manipulation. Several possible implementations of the VBS decoding algorithm of this controller are presented, capable to attain a reconfiguration frequency comparable to commercial FPGAs.

Eventually, a method allowing to enclose a specific reconfiguration area in the configuration path of an FPGA is proposed, which enable the runtime placement of tasks beyond the definition of partially-reconfigurable regions at design time. This mechanism, associated with the other improvements offered in this thesis, facilitates the development of a dynamically reconfigurable FPGA architecture capable of loading hardware tasks at runtime from a position-independent configuration bit-stream,

onto a flexible logic fabric.

Most of the methods and techniques described in this thesis were put in use in the context of the European FP7 project *FlexTiles*, which aimed to propose a 3-D stack manycore architecture featuring an embedded FPGA (eFPGA) offering dynamic hardware task loading capabilities.

Contents

Résumé **iii**

Abstract **v**

Introduction **1**
1 Context of the work 1
2 Contributions 2
3 Organization of the document 3

I Background **5**

1 FPGA Architecture **7**
1 (Re)configurable Hardware 8
1.1 Programmable Logic Devices 8
1.2 Reconfigurable devices 9
1.3 Advent of Field-Programmable Gate Arrays 9
2 Programming technology 10
2.1 Antifuse 11
2.2 Static memory 12
2.3 EEPROM / Flash 13
2.4 Summary 13
3 Logic array 14
3.1 Computing element 15
3.1-1 Logic blocks trade-offs 15
3.1-2 Look-up tables 16
3.1-3 Towards complex logic 17
3.2 Memory 20
3.3 Arithmetic accelerators 20
3.4 General purpose processors 21
3.5 Summary 22
4 Routing architecture 22
4.1 Segmented routing 22
4.1-1 Interconnect depletion 23
4.1-2 Segmented routing 23
4.2 Interconnect organization 24
4.2-1 Hierarchical architecture 25

4.2-2 Island-style architecture . . . 26
4.3 Routing structure . . . 27
4.3-1 Bi-directional routing . . . 27
4.3-2 Unidirectional routing . . . 28
5 Conclusion . . . 29

2 Runtime Reconfiguration and Routing of FPGAs 31
1 Runtime reconfiguration . . . 33
1.1 Reconfigurability of Field-Programmable Gate Arrays (FPGAs) 33
1.2 Early work on runtime reconfiguration . . . 33
1.3 Partial reconfiguration in modern devices . . . 34
2 Task relocation/migration . . . 35
2.1 Vendor-supported partial reconfiguration . . . 35
2.2 On hacks of modern devices . . . 35
2.2-1 Relocation on homogeneous fabric . . . 35
2.2-2 Handling heterogeneous architectures . . . 36
3 Run-time routing and communications . . . 39
3.1 Bus macro . . . 39
3.2 Configurable communication network . . . 40
3.3 Bit-stream merging . . . 40
3.4 Just-in-time routing . . . 40
3.5 Conclusion . . . 41
4 Placement and routing for FPGAs . . . 41
4.1 High-level synthesis and non architecture-dependent optimizations . . . 42
4.2 FPGA architecture mapping . . . 43
4.3 Placement . . . 43
4.4 Routing . . . 44
4.4-1 Global and detailed routing . . . 45
4.4-2 Combined routing . . . 47
4.5 Verilog to Routing . . . 48
4.5-1 VTR design flow . . . 49
4.5-2 Applications to commercial FPGAs . . . 50
5 Conclusion . . . 51

II Contributions 53

3 Position-independent tasks: Virtual Bit-Streams 55
1 Motivation for position-independent bit-streams . . . 56
2 Virtual Bit-Stream concept . . . 57
2.1 Interconnect abstraction . . . 57
2.2 Route modeling . . . 61
2.3 Coding the Virtual Bit-Stream . . . 61
2.3-1 Metadata . . . 62
2.3-2 Macro-cells . . . 63
2.3-3 Overview . . . 65

3 Clustering . . . 66
4 Virtual Bit-Stream generation tools . . . 67
4.1 Design flow overview . . . 67
4.1-1 Outputs of the Verilog-To-Routing flow . . . 68
4.1-2 Virtual Bit-Stream generation . . . 71
4.1-3 Decoding check . . . 71
4.2 *vbsgen*, the Virtual Bit-Stream generation back-end . . . 72
4.2-1 Architecture model framework . . . 73
4.2-2 Model array . . . 74
4.2-3 Output file generation . . . 75
5 A Virtual Bit-Stream powered architecture . . . 77
5.1 Reconfiguration controller . . . 77
5.1-1 Pre-fetcher . . . 78
5.1-2 De-virtualizer . . . 79
5.1-3 Logic mapper . . . 79
6 Limitations . . . 79

4 Results and online decoding of the Virtual Bit-Stream 81
1 Introduction . . . 82
1.1 Organization of the de-virtualizer . . . 82
1.2 Real-time considerations . . . 83
2 Online decoding algorithms . . . 83
2.1 LUT-based decoder . . . 84
2.2 State-machine decoder . . . 86
2.3 Comparison . . . 87
2.4 Implementation results . . . 88
3 Compression effect of the Virtual Bit-Stream . . . 89
3.1 Experimental methodology . . . 90
3.2 Results . . . 90
3.3 On the effect of clustering . . . 92
3.4 Fallback coding: to the limits of the Virtual Bit-Stream . . . 93
4 Conclusion . . . 96

5 Architecture Enhancements 97
1 Task migration with heterogeneous blocks . . . 98
1.1 Hard blocks abstraction . . . 99
1.1-1 Routing network separation . . . 100
1.2 Partitioning . . . 101
1.3 Task model . . . 101
2 Experimental methodology . . . 102
2.1 Modeling in VPR . . . 102
2.1-1 Architecture . . . 102
2.1-2 Routing abstraction . . . 103
2.2 Benchmarks . . . 104
3 Results . . . 104
3.1 Logic array size . . . 105

3.2 Critical delay . . . 105
3.3 Routing resources . . . 106
4 Conclusion . . . 108

6 Enhancing the Virtual Bit-Stream Architecture 109
1 Introduction . . . 110
1.1 Bit-stream loading methods . . . 110
1.1-1 Word addressing . . . 110
1.1-2 Scan-path: serial loading . . . 111
1.1-3 Hybrid loading . . . 113
1.2 Multi context configuration memory . . . 114
1.3 Summary of memory organizations . . . 115
2 Enhanced scan-path . . . 116
2.1 Organization of the configuration memory . . . 117
2.1-1 Configuration routing element . . . 118
2.1-2 Configuration path . . . 119
2.2 Runtime dynamic partitioning . . . 120
2.2-1 Configuring the configuration path . . . 120
2.2-2 Overhead and delay considerations . . . 121
3 Conclusion . . . 122

7 The FlexTiles platform 123
1 Overview of the FlexTiles project . . . 124
1.1 Global architecture . . . 125
1.2 Programming model . . . 125
1.3 Virtualization layer . . . 126
2 Embedded FPGA accelerators . . . 126
2.1 Overview of the embedded FPGA . . . 126
2.2 eFPGA architecture . . . 127
2.3 Hardware tasks . . . 128
3 Implementation of a dynamically reconfigurable FPGA . . . 129
3.1 RTL model . . . 129
3.1-1 Generation of the RTL model . . . 129
3.2 eFPGA testbench . . . 130
3.2-1 Testbench implementation . . . 131
3.2-2 User interaction . . . 132
4 Conclusion . . . 132

Conclusion and Perspectives 135
1 Overview . . . 135
2 Perspectives . . . 137

III Appendices 141

A Hardware specifications of the eFPGA 143
1 Logic macro-cell . . . 144

2 Logic blocks . . . 144
2.1 Complex Logic Block . . . 145
2.2 Arithmetic accelerator . . . 146
2.3 Memory block . . . 150
2.4 Accelerator interface . . . 150
3 Logic fabric organization . . . 151
3.1 Synthesis results . . . 151
3.2 Proposed organization . . . 152

B Simulation of the eFPGA reconfiguration controller 155
1 Task synthesis and bit-stream generation . . . 156
2 Model configuration and shell connection . . . 158
3 Adder loading . . . 158
4 Adder test . . . 159
5 Multiplier loading . . . 160
6 Multiplier test . . . 160

Publications 163

Bibliography 165

List of Figures 175

List of Tables 177

Acronyms 181

INTRODUCTION

1 Context of the work

Field-Programmable Gate Arrays (FPGAs) have become mainstream for most complex digital circuit designs over the last decade. Their versatility brings the power of custom hardware seen in Application-Specific Integrated Circuits (ASICs) at a fraction of the development and design costs, mainly aimed to the cost of developing the chip masks. The fast pace of evolution in the semiconductor industry allowed FPGAs to become increasingly complex, integrating heterogeneous accelerators and embedded processors to support the market needs.

Modern-day applications targeting FPGA devices progressively move towards increasingly dynamic designs. The recent process and voltage scaling demonstrated the failure of Dennard Scaling, leveraging a problem in multicore processors known as *dark silicon*. Dark silicon refers to the circuit area which cannot be powered-on at its nominal voltage without breaking the chip Thermal Design Power (TDP), to counteract the sharp increase of power densities in recent technology nodes. The threat of this power wall is an opportunity for more energy-efficient reconfigurable multicore architectures. Indeed, FPGA systems offer increased reconfiguration capabilities, allowing dynamic designs to maximize the FPGA resource utilization while keeping a lower power budget than with general purpose processors.

However, the constant addition of heterogeneous accelerators in FPGA architectures hamper the practicability of flexible reconfiguration schemes. Indeed, accelerators in FPGAs significantly improve the energy efficiency of functional designs in comparison with soft-logic implementations, but restrict the architecture dynamicity in case of relocation. Task relocation allows for a maximum flexibility of a reconfigurable architecture by optimizing at runtime the spatial and temporal scheduling of an application. To a lesser extent, relocation would also enable a hardware task to be reused over multiple compatible platforms, in the same manner as a software task. Seamless runtime hardware task placement on FPGAs has nevertheless be put aside with modern FPGAs, due to their complex architectures and the lack of software support in most FPGA vendors Computer-Aided Design (CAD) tool-flows.

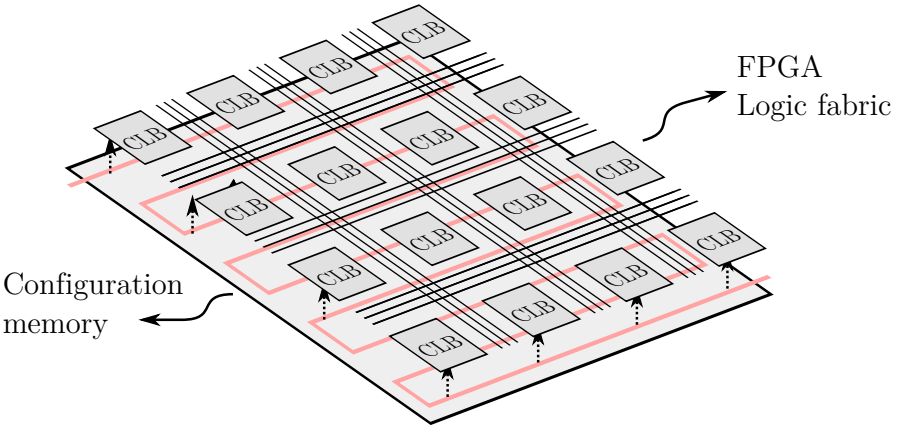


Figure 0-1 – **Overview of an FPGA architecture and its configuration mechanism**

2 Contributions

The contributions of this work have been conducted to focus around multiple aspects of both an enhanced FPGA architecture and its accompanying CAD flow to propose a concept of an efficient runtime hardware reconfiguration. Figure 0-1 illustrates a typical FPGA logic fabric architecture, along with its reconfiguration memory, here arranged as a contiguous scan-path. Flexible reconfiguration techniques hits a wall on multiple aspects on such architecture. First, as vendors CAD flows only support the generation of a bit-stream associated to a given Partially-Reconfigurable Regions (PRRs), it is necessary to resort to online bit-stream modifications such that it can be placed on multiple locations. Secondly, although on-the-fly bit-stream modifications techniques have been researched, they are severely limited by the FPGA architecture itself, whose heterogeneity prevents the association of a single hardware task to multiple positions on the logic fabric. Eventually, the configuration method of modern FPGA devices as well as their intrinsic configuration memory usually operate on pre-defined, fixed regions which hampers a finer-grain placement and configuration of a hardware task.

The work of this thesis focuses on these particular issues of FPGA dynamic reconfiguration schemes. The specific contributions of this work are enumerated as follows.

- A concept of pre-routed, position-independent abstracted bit-streams, the Virtual Bit-Stream (VBS), is proposed [HCS15]. This representation of a hardware task data is deprived of any knowledge about the final FPGA detailed architecture. The Virtual Bit-Streams are then decoded at runtime to reconstruct a conventional bit-stream dedicated to a specific location, which can then be loaded into the logic fabric. Along with the position-independence of this representation, allowing to virtually place it anywhere on a logic fabric, we observed a size reduction of up to 89% in comparison to a conventional bit-stream.

- Two algorithms allowing a real-time decoding of the aforementioned Virtual Bit-Streams are presented. These algorithms are to be integrated in the on-line controller of the architecture responsible for the decoding, placement and configuration of the bit-streams. A high throughput, similar to modern FPGA reconfiguration speeds, has been fixed as the real-time constraint to meet for these algorithms, which can be attained in both cases.
- A CAD flow, based on the Verilog-To-Routing (VTR) framework, has been developed to generate the Virtual Bit-Streams. This CAD flow relies on the placement and routing data generated by Versatile Place-and-Route (VPR) to create and verify the bit-streams abstraction and their equivalent conventional raw bit-streams. The offline part of the Virtual Bit-Stream technique is thus ensured to offer compatible working bit-stream to the online controller.
- To circumvent the FPGA heterogeneity issues regarding flexible placement of hardware tasks, an enhanced architecture is proposed [HST14]. This architecture exposes dedicated routing lines for heterogeneous hard-blocks of the FPGA logic fabric, separated from the soft-logic interconnection network. The separation of the interconnection networks allows a two-steps placement at runtime. First, the soft logic elements are configured and routed on the logic fabric. In a second step, the connections to the heterogeneous elements through their dedicated long lines are made in switch boxes. This effectively allows exhibiting a fine-grain horizontal flexibility of placement for hardware tasks. The delay impact of the proposed solution is an increase of 10% of the critical path delay in average.
- A scalable and finely-grained configuration memory suitable for the placement of hardware tasks is presented. This organization of the configuration memory enables the task placement to go beyond the fixed regions defined at design time and offers a more flexible alternative to reconfigure finely-grained portions of the FPGA logic fabric.
- Some methods and techniques precedently described have been implanted and validated in the context of the European project FlexTiles [Jan+15]. In particular, a Register Transfer Level (RTL) model of an FPGA architecture featuring the Virtual Bit-Stream technique and its reconfiguration controller was developped and simulated. The CAD flow has been put in use within the project to generate relocatable hardware tasks to be dynamically placed, at runtime, on the reconfigurable logic fabric RTL model.

3 Organization of the document

The thesis is organized as follows. The first part introduces the state-of-the-art pertaining to this work. A background on FPGA architectures is given in Chapter 1. The historical evolution of programmable devices is presented. The various programming technologies used in FPGA configuration memories are detailed. The

modern trends and progresses of the logic fabric and routing architecture of FPGAs are presented and detailed to serve as the general background of this thesis on FPGA devices.

Chapter 2 presents a detailed state-of-the-art on multiple aspects of the work conducted during this thesis. The taks relocation problem is exposed through the prisms of vendors CAD flows and bit-stream manipulation tools. Details on runtime routing in dynamically reconfigurable architectures are given. The chapter ends with a state-of-the-art of placement and routing tools for FPGAs.

The second part of the thesis details the contributions of this work. In Chapter 3, the concept of Virtual Bit-Streams is thoroughly presented. A description of the representation is given from the interconnect abstraction in macro-cells to the generation of the final bit-stream. A study on the effects of clustering multiple macro-cells together is studied. The tool-flow associated with the offline generation of the Virtual Bit-Streams is described, and the principles of operation of the proposed back-end bit-stream generation tool *vbsgen* are explained. A description of a Virtual Bit-Stream compatible architecture is presented, along with a description of the reconfiguration controller functions. The limitations of the Virtual Bit-Stream technique are also mentioned.

Chapter 4 focuses on the online implementation of the Virtual Bit-Stream method. The real-time constraints to be attained are described, and modern FPGA reconfiguration schemes are studied. Two specific algorithms for decoding the bit-stream representation detailed in Chapter 3 are proposed and compared to a generic maze-routing algorithm. Eventually, an analysis of the compression effect of the Virtual Bit-Stream representation is conducted on a set of large benchmarks.

In Chapter 5, FPGA architecture enhancements to ease flexible task placement are detailed. A method for separating the routing networks of soft-logic and heterogeneous hard blocks is proposed. The experimental methodology used to explore this routing architecture is thoroughly presented along with the set of benchmarks used for the experiments. An analysis of the results of these experiments in terms of delay and routing resources closes the chapter.

Chapter 6 explores the integration of a configurable reconfiguration method. Several state-of-the-art bit-stream loading methods are presented and compared in term of delay and area occupancy. An enhanced configuration method is proposed for flexible fine-grain hardware task loading, and discussed in comparison to fixed partially reconfigurable regions schemes.

This thesis concludes with Chapter 7 which exposes the European Seventh Framework Programme (FP7) FlexTiles project. An overview of the many-core architecture used in FlexTiles is presented, and the project goals are defined. The involvement of this thesis work within the project to propose a flexible embedded FPGA architecture (eFPGA) is specified. The chapter concludes on a discussion on the implementation of the actual so-called eFPGA. Finally, perspectives on future work are outlined for the overall thesis.

Part I

Background

Chapter ONE

FPGA ARCHITECTURE

Contents

1	(Re)configurable Hardware	**8**
	1.1 Programmable Logic Devices	8
	1.2 Reconfigurable devices	9
	1.3 Advent of Field-Programmable Gate Arrays	9
2	Programming technology	**10**
	2.1 Antifuse	11
	2.2 Static memory	12
	2.3 EEPROM / Flash	13
	2.4 Summary	13
3	Logic array	**14**
	3.1 Computing element	15
	3.2 Memory	20
	3.3 Arithmetic accelerators	20
	3.4 General purpose processors	21
	3.5 Summary	22
4	Routing architecture	**22**
	4.1 Segmented routing	22
	4.2 Interconnect organization	24
	4.3 Routing structure	27
5	Conclusion	**29**

Abstract

Since their introductions in the late 1980s, FPGA have been in constant evolution to meet the ever growing needs in performance and flexibility. This chapter traces the origins of FPGAs back to the introduction of reconfigurable chips in the 1970s. An overview of the composition of modern FPGA devices is then given, both in terms of their logic fabric and of their programmable interconnection network.

1 (Re)configurable Hardware

As early as the end of the 1970s, dynamically configurable products were available for commercial use. The ancestors of modern day *dynamically reconfigurable* Field-Programmable Gate Arrays (FPGAs) were designed to allow for the implementation of various generic algorithms on a chip. Although these Programmable Logic Devices (PLDs) are nowadays discontinued or deprecated in favor of more versatile solutions, the origins of their architecture and its evolution contributed to the inner features of FPGAs.

1.1 Programmable Logic Devices

The earliest of the PLDs are the Programmable Array Logics (PALs), which appeared in 1978, pushed by Advanced Micro Devices (AMD).

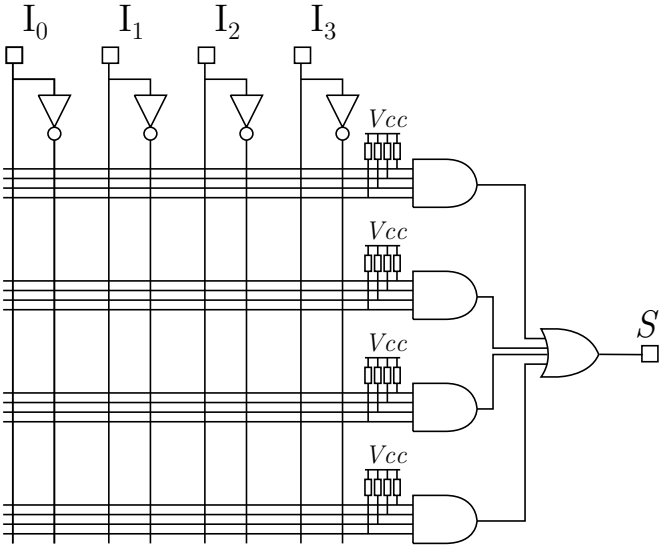


Figure 1-1 – **Details of the configurable sum-of-products in a PAL.** *The four inputs $I_{0...3}$ and their inverse are tied to a crossbar interconnect with the AND-gate inputs. In PALs, the AND outputs are OR-ed together to form the output, but subsequent technologies such as Programmable Logic Arrays (PLAs) added another crossbar to yield multiple outputs.*

Any logic equation of reasonable size, expressible as a *sum-of-products* can be implemented on a PAL. The inner architecture of these devices, illustrated in Figure 1-1, is made of a matrix of "AND" and "OR" gates. A programmable network can connect any of the inputs of the device (or their complement) to a set of "AND" gates, the *product terms*. The second, fixed, part of the device ties the output of "AND" gates to "OR" gates inputs to form sum of products. Floating inputs of the AND gates are weakly tied to a logic "1", the neutral element of "AND" operations, while OR gates inputs are weakly tied to a logic "0".

The PALs were originally One-Time Programmable (OTP) but brought massive production of generic configurable devices to the market, eventually leading to lower prices. In the pre-1980 era where custom logic functions needed to be implemented either as standard logic products or as a custom Application-Specific Integrated Circuit (ASIC) (a Read-Only Memory (ROM) would serve as a big Look-Up Table (LUT)), configurable chips paved the way to today's most powerful FPGAs.

1.2 Reconfigurable devices

Although subsequent PAL devices mentioned in the previous section were UV-erasable, their first electrically erasable concurrent, the Generic Array Logic (GAL), was introduced by Lattice Semiconductors in 1983. GALs are similar to PALs in their functionality and architecture, but with a greater flexibility since the non-volatile configuration memory is an Electrically Erasable and Programmable Ready-Only Memory (EEPROM), further reducing the prototyping time.

The most popular PALs and GALs, the 16V8 (16 inputs and/or 8 outputs) and 22V10 devices, featured a limited number of inputs and outputs. Larger applications (over ~ 200 gates) could be appealed by bigger programmable logic devices, but in the case of PALs and GALs it directly translates to higher AND and OR gates fan-in to accommodate the higher number of inputs and outputs, which eventually leads to wasted silicon and/or hazardous timings. To circumvent these issues, a new family of devices, the Complex Programmable Logic Devices (CPLDs), was introduced to fill this gap. CPLDs are much more similar to FPGAs since their inner architecture features an array of *macrocells*, connected together by a programmable interconnection network. The macrocells are no more than a set of logic functions (originally to implement sum-of-products) tied to a crossbar interconnection network. CPLDs can be seen as multiple GALs connected together, which make them the device of choice for applications which are too small for GALs but can fit in a CPLD (up to $\sim 10,000$ equivalent gates).

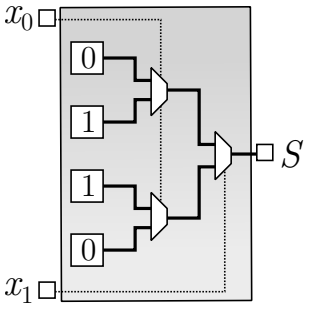
1.3 Advent of Field-Programmable Gate Arrays

Xilinx manufactured the first commercially viable FPGA in 1985 [Car+86]. The XC2064 [Xilg] was more powerful than existing CPLDs on various aspects. First, the use of volatile memory allowed a better silicon integration since the SRAM cells used are smaller than floating gate transistors used for non- volatile (EEPROM or equivalent) memory cells. The basic logic blocks of the FPGA are fully configurable LUTs rather than PLD-like *sum-of-product* macro-cells.

A K-LUT can implement any K-input logic equation, making them more flexible than their PLD counterpart. An equivalent LUT implementation of a XOR logic function, whose truth table is recalled in Figure 1-2a, is presented in Figure 1-2b with a 2-LUT. Additionally, while the early CPLDs routing interconnect was

x_0	x_1	S
0	0	0
0	1	1
1	0	1
1	1	0

(a) **Truth table of a two inputs XOR**



(b) **Equivalent implementation in a 2-LUT**

Figure 1-2 – **Implementation of a two input boolean equation in a Look-Up Table.** *The 2-LUT can be seen as a small memory whose address lines are in fact the boolean equation inputs. Given the inputs* (x_0, x_1)*, the cascading multiplexers of the memory will output one of the memory cell logic value, which have been configured beforehand to store the logic function* $S = x_0 \oplus x_1$*.*

constrained to route input and output signals from or to external signals, the mid-1980s FPGAs provided much more resources for internal state storage (flip-flops) and internal routing. This difference is less true nowadays, and the main difference between modern CPLDs and FPGAs resides in the use of non-volatile memory and a smaller number of cells for the former.

The XC2064 and most recent Virtex-7 series FPGAs were introduced 30 years apart and new advanced features are nowadays included in baseline devices. Advanced data-paths are now mandatory in modern architectures in order to ease the synthesis of complex functions, as well as fast transceivers, arithmetic accelerators and embedded memories. A survey of architecture details of legacy and modern FPGA devices is thoroughly detailed in the following sections.

2 Programming technology

The *programmability* of an FPGA is what distinguished it from other configurable technologies (e.g. ASICs or Mask-Programmable Gate Arrays (MPGAs)) at the time it was introduced. It characterizes the ability for the end-user to finalize or reprogram the FPGA chip on site rather than in a fabrication plant. Each configurable element of the chip needs one configuration bit. Logic blocks need multiple configurable elements to store, for example, the content of the look-up table. The routing structure also needs configurable elements to determine which nets are interconnected. The aggregation of all these configuration bits is the *bit-stream*: the configuration data needed to specify the function of all the programmable blocks of an FPGA. Over the years, multiple methods to store these configuration bits have been explored in FPGAs: one-time programmable or reprogrammable, volatile or non-volatile, each having its own benefits. The following sub-sections presents the

three most popular programming technologies in use in FPGAs, and a summary concludes on the advantages and downsides of these technologies.

2.1 Antifuse

The antifuse behave, as it name suggests, the opposite of a fuse. While a fuse is made as a normally-closed connection which can blow up and create an open circuit, the antifuse is normally-open and can be transformed in a short circuit. The metal-based link that forms the antifuse can be melt either in a fabrication plant by a laser, or using a high-current programmer in order to create an electrical connection.

Actel[1] is a well-known FPGA manufacturer which notably used antifuses in its chips. The Actel original antifuse is a link between a poly-silicon layer and an n+ silicon diffusion. A thin layer of dielectric between them melts when a high-power density is applied on the antifuse, which forms a permanent silicon link. In the QuickLogic approach to the antifuse technology, the link is made between two layers of metal but behave similarly. The metal-metal antifuse has the advantage of using no silicon area and being smaller than its poly-diffusion counterpart. Metal-metal antifuses are nowadays used by most antifuse-based FPGA vendors, including MicroSemi (formerly Actel) and QuickLogic.

Because of the non-reversibility of the antifuse programming process, the devices based on this programming technology are considered One-Time Programmable (OTP) devices. This particularity, which may not be practical for tests, comes with the resistance against reverse-engineering. The configuration memory of antifuse-based devices cannot easily be read back electrically as most antifuse-based FPGAs comes with security features to completely disable the programming link. Additionally, because of its OTP and non-volatile nature, the configuration bit-stream is not transmitted when the chip is powered up, which further reduces the practicality of intercepting the bit-stream *in situ*.

Another advantage of antifuses is their low resistance. Blown poly-diffusion antifuses have a resistance of approximately 500Ω, which can go down by an order of magnitude ($60 - 80\Omega$) with metal-metal antifuses [Smi97]. The low resistance along with the low parasitic capacitance of antifuses links can lead to a higher integration of configuration bits per devices in comparison to other technologies. Additionally, since the antifuse state does not depend on the charge of an element, their configuration state is immune to high-energy radiations which would otherwise disturb SRAM or EEPROM cells.

However, antifuses have non-negligible drawbacks beside their one-time programmability. The manufacturing of antifuses requires custom CMOS processes which are not up-to-date with the latest technology nodes [KTR08], and the alterations of the fuse materials may not be scalable to the smallest technology nodes used in SRAM-based FPGAs.

[1] Actel have been acquired by Microsemi in Nov. 2010

2.2 Static memory

Static memories are the de-facto standard for most of the FPGAs devices on the market shipped by the major manufacturers Xilinx, Altera or Lattice. Unlike EEPROM cells, Static Random Access Memory (SRAM) cells do not wear and could theoretically be reprogrammed infinitely. The main reason behind the widespread use of SRAM cells in FPGAs is that they do not need any special manufacturing technique. The use of a standard CMOS process to realize the chip allows to attain better integration via process shrinking than what would otherwise be possible with EEPROM or antifuse cells, albeit smaller.

In comparison to the other two presented programming technologies, static memories nevertheless suffer from their volatility. As the cell state is not retained over power cycles, the FPGA configuration memory needs to be stored in an external, non-volatile memory and loaded during the system power-up. The potential security problems caused by an accessible configuration bit-stream are often dealt with encrypted bit-streams and hardware cryptography blocks to load the configuration. This additional external memory also adds up to the overall system cost. Recent devices integrate an embedded Flash memory to store the bit-stream without requiring an additional chip, but this becomes impractical with larger FPGA devices.

SRAM cells are coupled with pass-transistors in commercial and almost all academic FPGAs. The memory cell drives the gate of an N-channel Metal-Oxyde Semiconductor (NMOS) transistor and thus controls its passing state. This is the most used technique to implement a switch in FPGAs since it is also the most area effective: it requires only one transistor in addition to the memory cell. Pass-transistors have however important disadvantages since they cannot pass a full logic-high voltage and have a relatively low output slew-rate. The output voltage of a passing transistor for a logic-high is approximately $V_G - V_{Th}$, where V_G is the voltage applied on the NMOS gate (i.e. normally $0V$ or V_{DD}) and V_{Th} the threshold voltage of the transistor which depends on the technology. As the Complementary Metal-Oxyde Semiconductor (CMOS) process technology scales, the logic-high voltage V_{DD} drops faster than V_{Th}, resulting in an even smaller pass-transistor output. This problem leads to the degradation of a signal over a chain of pass-transistors: a logic high can be interpreted as a logic low signal and the effect of noise is increasingly important. The slowness of switching also increases the dynamic power usage of the device. Several techniques have been proposed to circumvent this problem, ranging from the integration of signal buffers, which have a high impact on area usage, to gate voltage boosters, at the cost of premature device aging. Recent work has been conducted to move away from pass-transistors to more efficient methods. As an example, Chiasson and Betz [CB13] proposed to use transmission gates, which provide better performance but are also bigger.

2.3 EEPROM / Flash

Electrically Programmable Ready-Only Memory (EPROM) and EEPROM-based technologies use a Metal-Oxyde Semiconductor Field-Effect Transistor (MOSFET) with a second *floating* gate between the *control* gate and the MOSFET channel. The state of the transistor is contained by the charge held in the floating gate, electrically isolated from the two other layers. The programming of such a cell require a high programming voltage V_{pp} to be applied to the drain of the transistor so that, via an avalanche effect, a charge builds up in the floating gate which raises the threshold voltage of the transistor. In this case, a programmed NMOS would be off even if the *control* gate is driven high to V_{dd}. Additional energy is required to discharge the floating gate into the bulk, either through ultraviolet light for Ultraviolet Programmable Ready-Only Memory (UVPROM)-based chips, or using an electric field for EEPROM-based devices. Specifically arranged arrays of these EEPROM cells are also called *Flash* memories, which also differ in their geometry and technology.

The data retention of Flash memories is reasonably high to consider them non-volatile in most operating conditions. This non-volatility is one of their advantages in comparison to SRAM-based FPGAs: an FPGA using a flash memory for its configuration is ready right upon its power-up since it does not have to reload its configuration bit-stream after each power cycle.

Flash arrays are however subject to memory wear and read disturb. The memory wear causes the EEPROM cells to only withstand up to approximately $100,000$ write cycles before the accumulation of charges on the floating gate generates a failure which makes the cell to stick to 0. Besides, continuous subsequent readings of the same cell without erasing may cause a failure in adjacent cells, known as a *read disturb*. Although these problems can be circumvented by provisioning redundant cells or rewriting the cells every x cycles, they need to be considered and increase the area usage on the FPGA chip.

In comparison to SRAM cells, the EEPROM cells require slower write timings and the previously evoked drawbacks. Like antifuses, EEPROM cells also need specific CMOS processes and the complications which go with it. EEPROM cells have however a major advantage over SRAM cells: they use less transistors and thus occupy less area than SRAM cells.

2.4 Summary

The choice of the programming technology of the configuration memory has an impact on a multitude of factors on FPGA devices, as shown in the three precedent subsections. Table 1-1 summarizes the characteristics of antifuses, EEPROM cells and SRAM cells. The best technology is reprogrammable, non-volatile, uses a standard CMOS manufacturing process, has a low area impact, and is not prone to errors induced by high-energy radiations. A conjunction of all of these five factors is not attainable with current technologies, and each of these memory cells has its

	Antifuse	EEPROM	SRAM
Reprogrammable	No	Yes[2]	Yes
Volatile	No	No	Yes
Manufacturing process	Antifuse	Flash	Standard CMOS
Area impact	Low	Moderate	High
Sensitive to radiations	No	Yes	Yes

Table 1-1 – **Summary of the three main FPGA programming technologies**

own advantages and weaknesses. Modern FPGA devices mostly rely on SRAM-based configuration memory arrays because of the low cost and higher density of up-to-date CMOS manufacturing processes. Incidentally, they also suffer from their configuration memory volatility and exposure to security breaches if an attacker can retrieve the bit-stream.

Alternative non-volatile memory cells have been explored in the context of reconfigurable computing [Tor+13], notably the Magnetic Random Access Memory (MRAM) which has been evaluated in the context of an FPGA circuit in [Bru+06]. The numerous emerging non-volatile technologies are nowadays not spread in commercialized FPGA as their integration level is still far from the three mainstream programming technologies presented in this section. Non-volatile memories are however actively explored for reconfigurable computing as they provide better scalability or power consumption, two important factors when considering the design of a reconfigurable architecture.

3 Logic array

The logic array of an FPGA device is often modeled as a grid of elements which can be programmatically connected together through a dedicated network. This section focuses on the nature of the logic grid and on the heterogeneity of these blocks. Figure 1-3 summarizes the evolution of various FPGA devices manufactured by Xilinx over the course of fifteen years, from the first Virtex FPGA (1998) to one of the latest Virtex UltraScale (2014). The manufacturing process scaling and the development of flagship technologies have permitted a constant growth of the complexity of embedded *hard* blocks in the devices, along with the introduction of more and more *soft* computing elements within the chips.

Subsection 3.1 presents the trade-offs associated with the soft logic blocks (in cyan and orange in the figure). Subsections 3.2 and 3.3 respectively introduce two of the most used features of FPGAs: the memories (in red) and arithmetic accelerators (in purple). Subsection 3.4 describes the usage of more complex processor units within recent FPGA logic fabrics. In addition to these computation blocks, higher-end devices also include complex transceivers withstanding a throughput of

[2] Up to $100,000 - 1,000,000\times$

up to 100 gigabits (green, blue, magenta, yellow and orange in the figure). These interface blocks are not covered in this section but are prominent in signal processing applications or in designs which use fast Dynamic Random Access Memory (DRAM) interfaces.

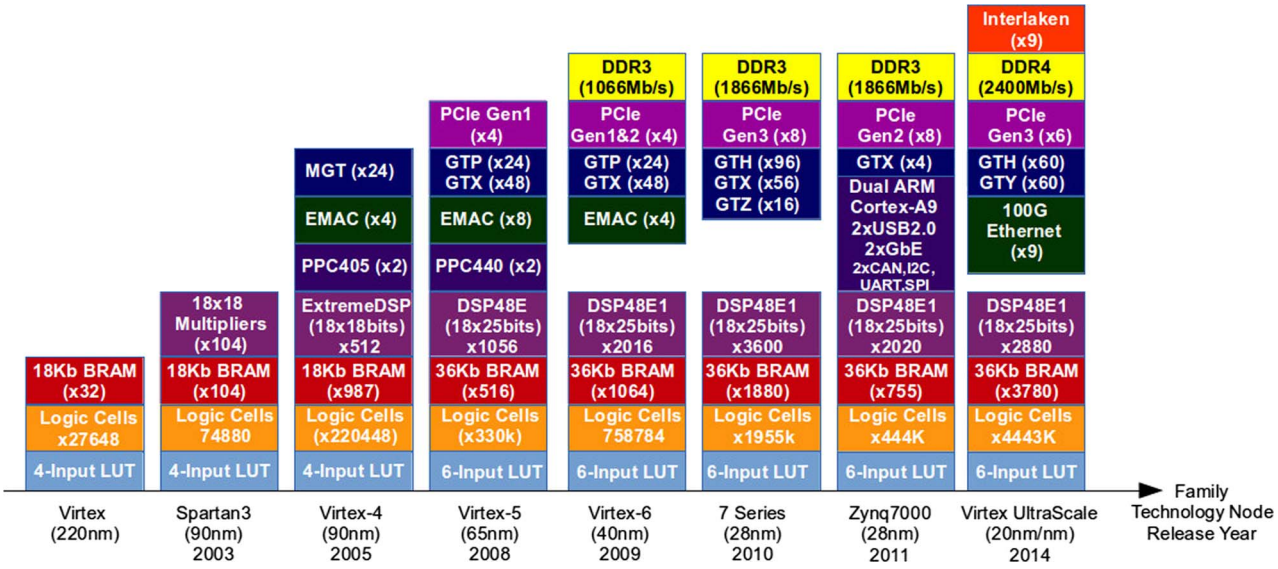


Figure 1-3 – **Evolution of the number and complexity of hard blocks in FPGAs [Lyk+15].** *The evolution of Xilinx FPGA devices composition in terms of soft and hard logic blocks is representative of the market tendency. As the chip complexity and manufacturing techniques evolved, more and more very high speed transceiving features were included in FPGAs.*

3.1 Computing element

The role of the logic block in the FPGA architecture is to provide computational power and to be used as a simple storage element. Its versatility is sustained in recent architectures by the integration of arithmetic paths within the soft logic blocks. These paths allow for the synthesis of complex data-paths without mobilizing huge amounts of the flexible interconnect.

3.1-1 Logic blocks trade-offs

Early work by Marple and Cooke [MC92] suggested the transistor to be the smallest computing element of the FPGA, as an attempt to mimic MPGAs. As the transistor represents the finest possible grain of reconfigurability, it comes at the cost of the huge area needed by the configurable interconnect. The problem in the choice of the basic computing element in an architecture boils down to a trade-off between flexibility and efficiency. The idea of the finest-grain transistor leads to a highly flexible architecture, even capable of mapping an MPGA standard cell, but also carries a routing nightmare for Computer-Aided Design (CAD) tools,

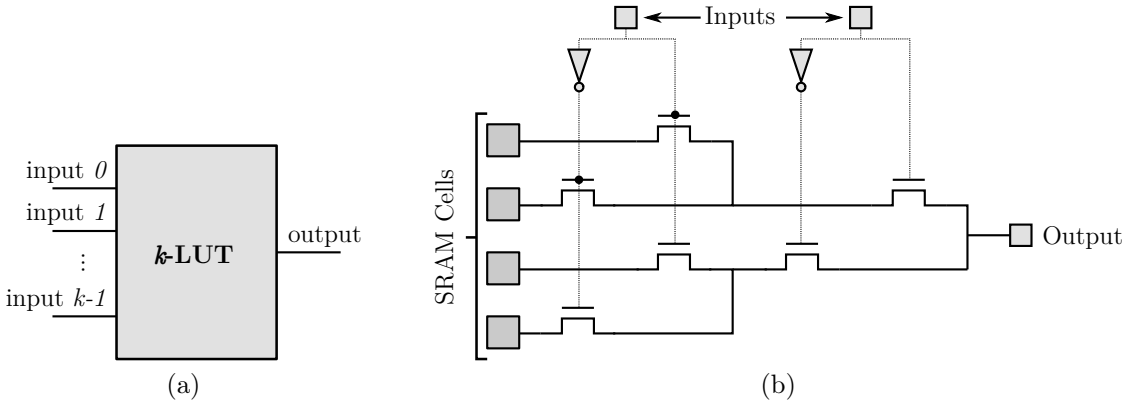


Figure 1-4 – **Architecture of a 2-LUT using memory cells and pass-transistors.** *In an SRAM FPGA device, the most convenient way of implementing multiplexers is to use pass-transistors. The implementation of a k-LUT requires* 2^k *memory cells and a k-to-*1 *multiplexer, here implemented using a tree of pass-transistors, although more optimized versions are more commonly used.*

power inefficiencies due to the complex interconnection network and an overall low performance. On the other hand, the computing element could be as complex as a processor: this approach would put less stress on the interconnect to a minimum but would not benefit from the flexibility inherent to reconfigurable hardware.

The difficulty behind the choice of a logic block, and thus of the aforementioned trade-off between flexibility and efficiency, also resides in the overall lack of data prior to the chip design when introducing an innovative computing element. Without an operational CAD tool-flow and valid benchmarks, it is difficult to judge the performance of the various choices available in the design space of FPGA architectures.

Most FPGA are nowadays based on LUTs, which have been heavily studied and tested since the introduction of FPGA devices thirty years ago.

3.1-2 Look-up tables

Look-Up Tables are nowadays the *de facto* finest versatile computing element in modern FPGA devices. A k-LUT is a small memory containing the truth table of a k-input logic equation. Figure 1-4a illustrates the abstract view of a k-LUT: the truth table itself is contained in a 2^k bits memory, and an address decoder is implemented with a multiplexer. The logic inputs are tied to the multiplexer control lines in order to select the truth table output for a given set of k inputs. A more detailed view of a k-LUT is shown in Figure 1-4b where the memory is implemented with SRAM cells and the decoder with pass-transistors, as mentioned in Section 2.

A major advantage of k-LUTs is their ability to implement any k input boolean equation. The inclusion of LUTs and interconnect in the architecture would be

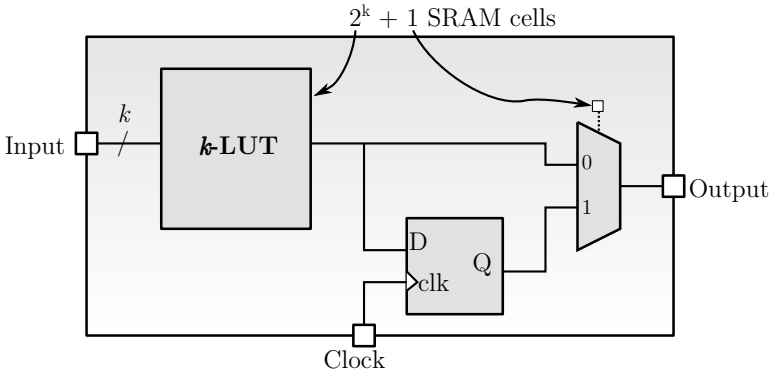


Figure 1-5 – **Overview of a basic logic block.** *The basic logic block comprises a k-LUT to provide the logic flexibility, associated with a D flip flop to synchronize the LUT output with the clock. A multiplexer on the output path enables the selection of the asynchronous or synchronous LUT output to allow for complex logic chains.*

enough for a completely asynchronous programmable device, which is not the goal of FPGAs. The need to implement complex functions on the logic fabric make the inclusion of synchronization elements in the logic chain a requirement in FPGA architectures. The synchronization is brought aside the LUTs as an optional path for the logic. Figure 1-5 shows a simplified overview of what can be considered the basic logic block of an FPGA: the LUT is associated with a D flip-flop synchronized to the device clock. Using an additional multiplexer, it is possible for the basic logic block to either output a synchronized signal of the LUT output in order to have a stable logic value, or to bypass the flip-flop and chain multiple basic blocks to compute more complex equations in a single clock cycle. The maximum length of such a chain is limited by the length of the interconnect between each logic element and by the device clock speed.

3.1-3 Towards complex logic

As specified earlier, the design space of logic block elements has been explored in various direction since the early beginning of FPGA devices, each new architecture bringing its own advances. In most of the modern architectures such as the Xilinx Virtex 7 or the Altera Stratix V, the logic block is not anymore composed of a single LUT. Over the years, the effect of the LUT size (i.e. the number of inputs) and of the combination of multiple LUTs together in a logic cluster have been studied. A clustered logic block comprises multiple basic logic blocks associated to a local crossbar network which interconnects the cluster inputs and the basic logic block outputs to the basic logic block inputs, as illustrated by Figure 1-6. Depending on the architecture, this local crossbar may be more or less complete in order to make a trade-off between the routing area and flexibility. The cluster is defined by its LUT size k, the number of LUTs N and the amount of inputs to the cluster I.

Ahmed and Rose [AR00] performed a deep study of the effect of the LUT size and of the cluster composition on area, delay and performance. They explored the design

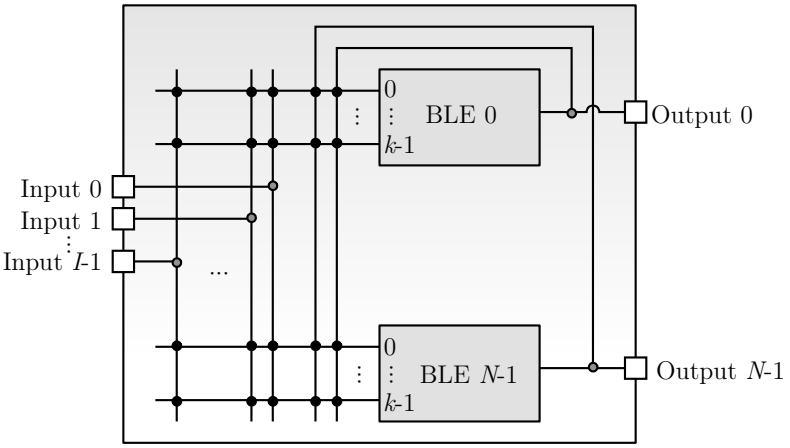


Figure 1-6 – **Overview of a complex logic block.** *The complex logic block contains a cluster of N basic logic blocks containing k-LUTs whose inputs and outputs are connected to a crossbar interconnection network local to the CLB. I inputs are routed from the global interconnection network to the local crossbar, with I carefully explored to balance the area and logic occupancy of the logic fabric. All the N outputs of the basic logic blocks are routed to the global interconnection network.*

space of clustered logic blocks through the use of 20 benchmarks. The results are evocative of the current practice in commercial FPGA. Their conclusions to mitigate the area-delay product is summarized in Figure 1-7. The figure plots the geometric average of the area-delay product of the implementation of the 20 benchmarks on clusters of size N from 1 to 10 and a varying LUT size k from 2 to 7. The results show that LUT sizes between 4 and 6 have the best area-delay performance over smaller or bigger LUT sizes. The area and delay taken by the LUT is a balance between the size of the 2^k SRAM cells which goes up exponentially in function of k, and the delay of the output decoder which grows linearly with k as each new input adds one pass-transistor stage to the multiplexer. Similarly, the figure demonstrates close performance results for cluster sizes over 4. This is explained by the fact that bigger clusters are associated with increased crossbar area, but the mean inter-cluster delay

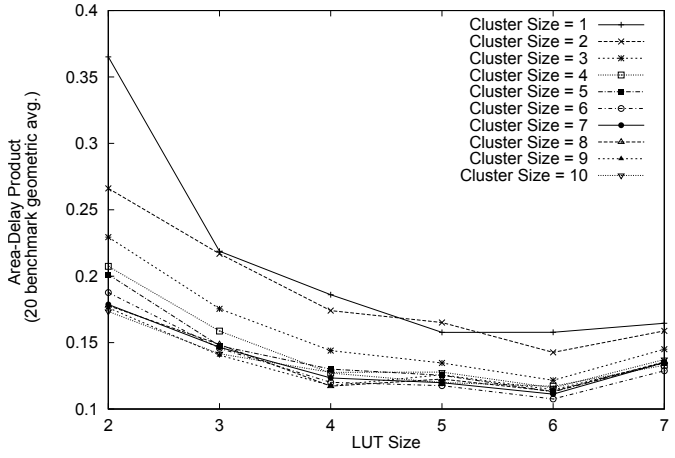


Figure 1-7 – **Area-delay product for clusters of size** 1 **to** 10 **[AR00]**

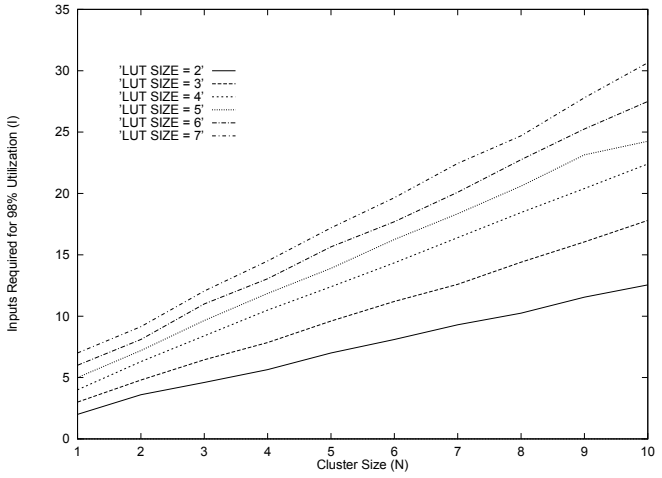


Figure 1-8 – **Number of inputs required for 98% logic utilization [AR00]**

becomes smaller since fewer nets of the design require inter-cluster routes.

Apart from its performance, a further key decision on the cluster size and composition is the area dedicated to the interconnect. Lemieux and Lewis [LL01] realized a study on the effects of the depletion of the cluster crossbar connections. They ran experiments with LUT inputs in the cluster connected to only a fraction $F_{C_{in}}$ of the cluster inputs and F_{fb} to the LUT outputs feedback and observed the effect on the minimum channel width and performance of the synthesized designs. The results show that *sparse* crossbars (with $F_{C_{in}} = F_{fb} = 0.5$) have little influence on the minimum channel width required to route the benchmarks, and no influence on the delay, whereas the area savings are appreciable. Adding spare inputs to the cluster and carefully exploring the crossbar sparsity can save up to over 14% of area in a cluster with $N = 6$ LUTs of $k = 4$ to 7 inputs.

In addition to the logic and crossbar area of the cluster itself, the interconnect between the cluster and its surrounding routing network must also be considered. As part of their studies, Ahmed and Rose also studied the effect of the number of cluster inputs I required to attain 98% of logic block utilization within the clusters, summarized in Figure 1-8. The figure shows the minimum number of inputs I required to attain a logic utilization of at least 98% in the clusters, for various cluster sizes N and LUT sizes k. The naive value for I would be to choose $I = k \times N$ to ensure that each LUT input can be routed to the device interconnection network, but this value can be aggressively smaller for multiple reasons: not all LUTs use all their inputs, and some of them will share the same inputs as other LUTs in the cluster. Additionally, some LUT inputs are connected to outputs of LUTs of the same cluster, which are routed through the cluster crossbar rather than the global interconnect. Ahmed and Rose derived from Figure 1-8 the general equation

$$I = \frac{k}{2} \times (N + 1) \tag{1-1}$$

to determine the number of cluster inputs I needed by a cluster of N k-LUTs.

3.2 Memory

The synthesis tools have two ways of creating memories on an FPGA device. The first one is to use soft logic (i.e. computing elements) as storage blocks, since most commercial architectures implement the computation in LUTs as previously described. This technique, known as *distributed Random Access Memory (RAM)*, can be efficient for registers of a few dozen bits, but becomes quickly limited by the inefficiency of using large amounts of logic blocks over a wide area. The second type of memory on an FPGA is the *hard memories*, specifically designed to hold data and implemented as optimized SRAM arrays.

Manufacturers have included hard memories in their FPGA logic fabric as early as the Altera Flex 10K FPGA devices, which included $2,048$ bits of memory and could be configured as either $1 \times 2K$, $2 \times 1K$, 4×512 or 8×256 bits memories. The configurable memory ratio is an important feature to support the wide heterogeneity of designs that can be synthesized to a given architecture, especially given the low area impact induced by this versatility. Most, if not all, of the recent FPGAs include memories nowadays: the biggest Virtex 7 series FPGA from Xilinx have up to $67Mb$ in dedicated memories, which include hardware First-In First-Out (FIFO) behavior support.

3.3 Arithmetic accelerators

As the variety of applications targeting FPGA devices grew larger, manufacturers began to include hardware support of arithmetic data-path modules. The Virtex II/II-Pro series FPGAs introduced a 18×18 hardware multiplier associated to a memory block. Arithmetic accelerators are nowadays very complex and can be configured for a wide variety of applications. As an example of modern architectures support for hardware arithmetic accelerators, Figure 1-9 shows a synoptic of the DSP48E slice included in Virtex 7 FPGAs. This arithmetic block works with four inputs A, B, C and D on respectively 30, 18, 48 and 25 bits, along with configuration words $INMODE$ and $OPCODE$ for the pre-adder, input register and overall operation selection. The output P itself is on 48 bits and is internally wired to a feedback signal allowing multiply-accumulate (MAC) operations. Internally, the arithmetic data-path contains a 25-bit pre-adder, a 25×18 multiplier and a 48-bit Arithmetic and Logic Unit (ALU). Several signals in Figure 1-9 are noted as not routable on the logic fabric, but are rather directly tied to adjacent DSP48E blocks to support additional carry logic and arithmetic accelerators chaining.

Altera proposes similar complex arithmetic accelerators in Stratix V FPGAs, with a DSP block configurable up to a 27×27 multiplier and a 64-bit output register, which shows the general tendency of FPGA vendors to provide the architecture with complex arithmetic features to enhance the performance of applications which require them.

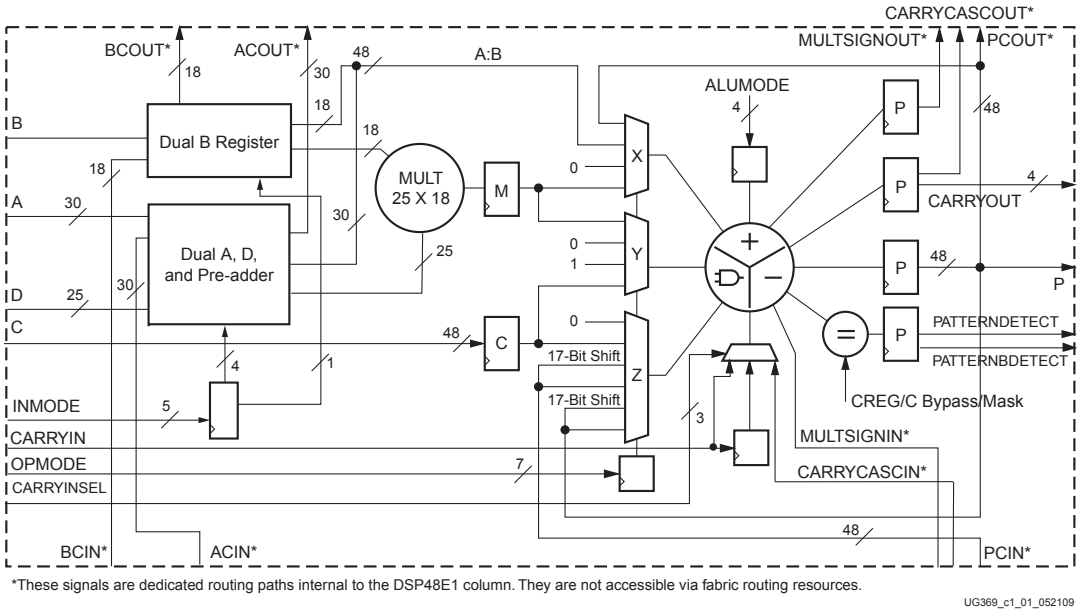


Figure 1-9 – **Overview of the DSP48E1 slice included in Virtex 7 series FPGAs [Xila].** *The complex evolution of the Digital Signal Processor (DSP) slice in Xilinx Virtex devices is representative of modern trends in FPGAs regarding arithmetic accelerators. The DSP48E1 slice comprises a pre-adder, a* 25×18 *multiplier and a* 48*-bit ALU to provide the device with complex configurable accelerators. Notably, the dedicated carry in and out datapaths allow to chain multiple DSP slices together to realize operations on a larger data size.*

3.4 General purpose processors

Digital systems often need the flexibility brought by microprocessors, along with the power offered with parallel execution of synthesized hardware on an FPGA fabric. Multiple variants of Harvard and Von Neumann processors architectures are available as *soft* processors synthesized on FPGA devices, but at a high cost in terms of area and strong timing constraints. The main manufacturers Xilinx, Altera and Lattice provide their own soft-core (respectively Microblaze/Picoblaze, Nios/Nios II and LatticeMico8/32). These cores have the advantage of being heavily supported in the vendor CAD tool-flow in comparison to IP providers or other open-source soft-cores.

The general tendency for most manufacturers is to provide, in addition to their range of pure-FPGA products, reconfigurable System-on-Chips (SoCs) where a *hard* processor core is associated to a reconfigurable gate array. This association provides the flexibility of a microprocessor for most general tasks, along with a reconfigurable logic fabric to handle the most computation-intensive tasks, synthesized on the FPGA. Xilinx has the Zynq series integrating ARM Cortex-A9 or Cortex-A53 processors, Altera also ships ARM cores in some of its SoC-oriented FPGAs in the Stratix, Artix, Arria and Cyclone products, and Microsemi features Cortex-M3 cores in its SmartFusion FPGAs.

Nevertheless, the flexibility brought by the integration of soft or hard processors cores in FPGA systems increases the difficulty of the design, test, validation and debugging. Interactions between the hardware and software parts are more complex to predict and the design of applications targeting such systems requires even more knowledge from the designer.

3.5 Summary

The inclusion of hardware accelerators in FPGA logic fabrics represents, from the manufacturer point of view, an opportunity to target even more applications that can be easily synthesizable and scalable on an FPGA. For the end user, the increased complexity of these hard blocks and of the soft logic blocks is only tolerable thanks to the development of CAD accordingly *smart* to make use of the added functionality. If we take the example of the soft logic, the area occupation and performance breakthrough brought by clustering, sparse crossbars and depopulated inputs to clusters can only emerge with a matched up tool-flow aware of the underlying architecture trade-offs.

4 Routing architecture

The FPGA routing architecture is the one of the main defining feature of the device with the logic resources. Due to the high programmability of the interconnect, the surface occupied by the routing network of the chip largely outweighs, in proportion, the area allocated to logic blocks and accelerators. Powerful devices must not only propose complex logic blocks which suit a wide range of applications, but also provide the chip with a routing architecture balancing area occupancy, flexibility and performance. The performance is directly tied to the number of switches in a connected path, each adding capacitive and resistive components which will affect the path delay. The performance is however partly balanced by the CAD tools smartness, which may take care of architecture-dependent algorithms or heuristics. The area occupancy depends on the routing network connectivity, on the memory cells used and on the efforts put in the layout of the routing blocks. Flexibility, however, is somewhat harder to define and is generally studied through a set of benchmark applications placed and routed on multiple architectures.

4.1 Segmented routing

A routing network in FPGAs devices can be schematized as in Figure 1-10. The notations of the routing structure parameters follow the one in use in this domain area, as originally introduced by Rose and Brown [RB91]. The network is formed by routing tracks grouped in channels. The number of routing tracks in a channel is called the *width* of this channel, denoted W. The routing tracks are connected to the various soft and hard logic blocks of the device through *connection blocks*

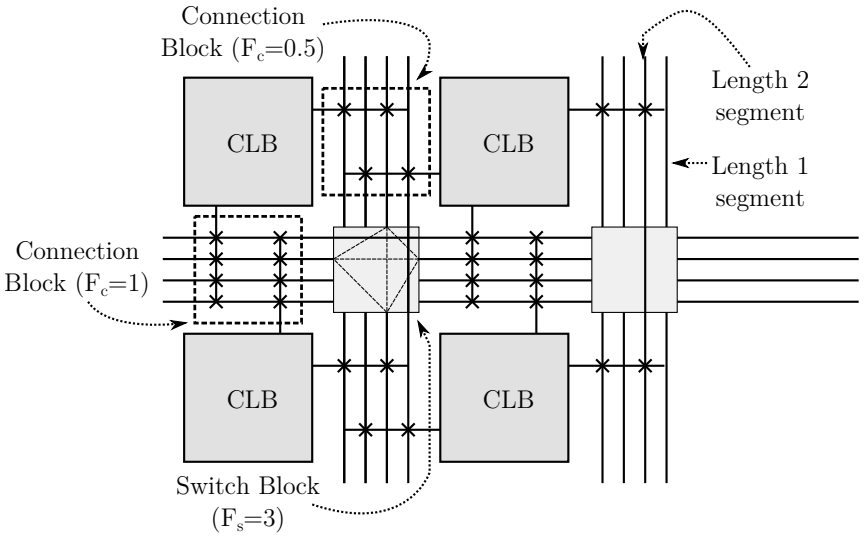


Figure 1-10 – **Schematized routing network of an FPGA device.** *The detailed routing of an FPGA is shown here on an island-style-like architecture but can be transposed to any other architecture. An array of soft and hard logic elements is connected to a dense routing network through connections blocks with a varying ratio F_c of connections. Switch blocks realize the interconnection of wire segments together with a given number F_s of possible connections from one pin to another.*

which contain configurable connection points between the tracks and the logic block inputs and outputs. Routing tracks can be connected to other routing tracks within *switch blocks*, where additional configurable connection points allow interconnecting the tracks to a subset of the other switch block endpoints.

4.1-1 Interconnect depletion

Although a completely interconnected routing network with fully populated switch blocks and connection blocks is theoretically possible, routing architectures are practically never implemented as such because of the area and performance overhead incurred by the configurable connection points. Depleted connection blocks do not connect the input and output pins of logic blocks to every track of a channel. A parameter F_c defines the connection block flexibility, usually expressed either as a number of connection points for each logic block pin or as a fraction of the channel width W. Similarly, the depletion of interconnections inside a switch block is modeled by a parameter F_s, expressed as the number of reachable endpoints from any other endpoint of the switch block.

4.1-2 Segmented routing

As previously noted in Section 2, resistive and capactive components of configurable connection points have a non-negligible impact on the performance of routed

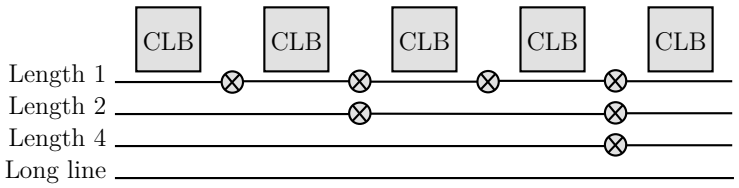


Figure 1-11 – **Example of a distribution of routing tracks in an interconnect architecture.** *A wide distribution of the segment length over the device allows longer path to use less routing resources. The decreased wire capacitance has a direct impact on the wire delay which is substantially lower than with multiple smaller segments.*

paths on the FPGA routing architecture. Because of this, the segmented routing architectures of modern devices do not rely only on unit-length tracks crossing a single connection block before terminating into a switch block. Indeed, a distribution of tracks among multiple *track lengths* is often explored, as illustrated in Figure 1-11, to provide the architecture with enough heterogeneous routing resources to enhance the performance of non-local nets. The goal of using longer wires is thus twofold: it allows reducing the number of programmable switches and thus the delay of a given wire, and at the same time it reduces the number of memory cells required, which in turns influences the area occupation of the interconnection structure. The distribution of wire segment lengths however leverages a critical design decision on the trade-off between more short wires, at the cost of low long-path performance, and more long wires, at the cost of possibly misused higher-length segments for short connections.

Although long wires are assumed to connect to a switch block on both ends, multiple variations of the *internal population* of connection blocks and switch blocks *within* wire segments are possible, which adds even more exploration parameters for FPGA architectures. Chow *et al.* [Cho+99] explored the effect of this internal population and concluded that switch blocks depopulated long wires resulted in a significant increase in required channel width. This is however the trend in modern FPGA architecture to have long wires both switch block and connection block depopulated: the loss of local flexibility is counterbalanced by clever long wire organizations and segmentation distribution.

4.2 Interconnect organization

Several organizations of the interconnection structure have been developed for FPGAs over the years, mostly due to the fact that each vendor pushed their own specific architecture to sell their devices. The most popular is surely the island-style architecture, presented in subsection 4.2-2, and the hierarchical architecture of early Altera devices, summarized in subsection 4.2-1. Other types or interconnect organizations exist or have existed, notably the row-based architecture which only have horizontal routing channels, but are not covered in this section since they are either not in use or not being significantly used anymore in modern architecture to

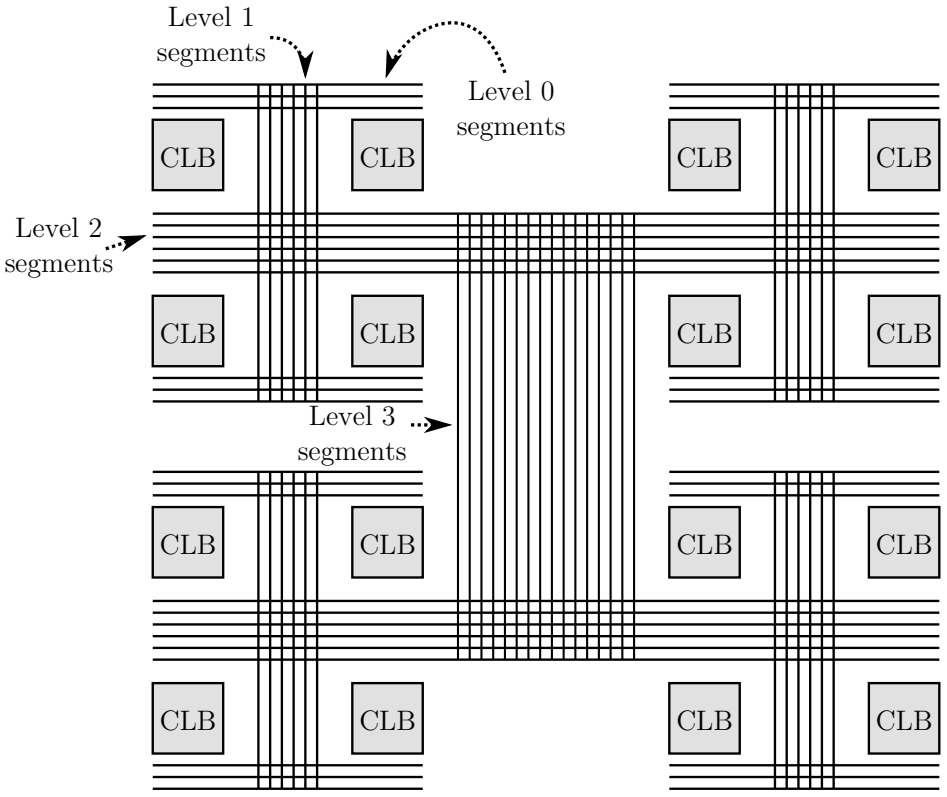


Figure 1-12 – **Overview of a hierarchical interconnect architecture.** *In a hierarchical architecture, the routing network is split in multiple levels in the same manner as an* H-tree. *The delay model is simpler in this case since it can be finely estimated from the global routing, given the number of interconnection levels crossed on a net path.*

be detailed in this document.

4.2-1 Hierarchical architecture

Hierarchical architectures have mainly been pushed by Altera in its Flex10K and Apex FPGAs. The principle is to pool clusters of logic blocks together along with a local interconnect. The groups of clusters are connected together to a local interconnect, itself connected to a global interconnect. This organization forms a hierarchical interconnect architecture where each interconnect level can only reach its upper and lower level, as illustrated in Figure 1-12.

The global routing of hierarchical architecture was supposed to provide a more predictable routing delay between two logic blocks given their respective logic groups, but this improvement only holds if the synthesized design is able to exhibit a routing distribution similar to the architecture. The routing predictability fades away on large designs with a lot of local interconnections which are forced to use upper-level routing in the hierarchical organization. Altera has since moved to flat interconnect network organizations in its most recent devices

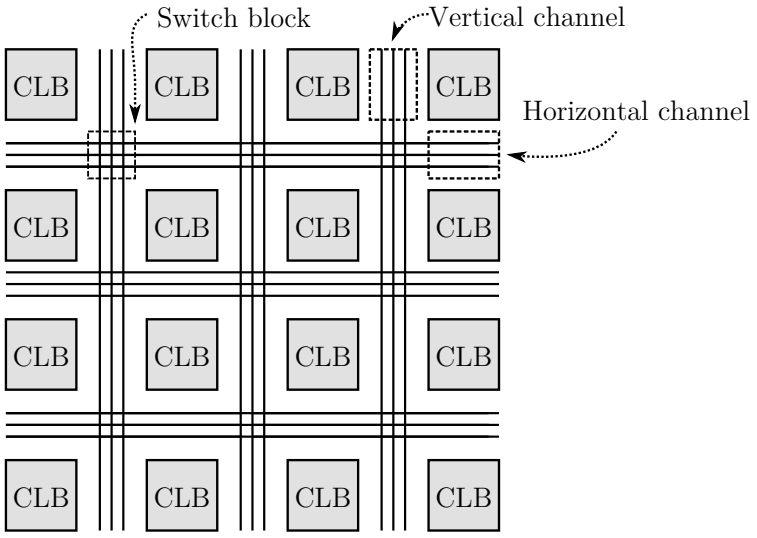


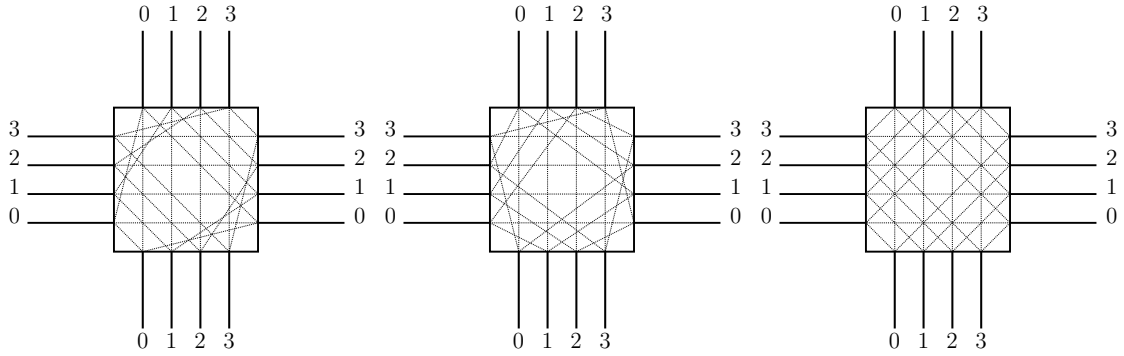
Figure 1-13 – **Overview of an island-style interconnect architecture.** *Island-style architectures have regular logic fabric and routing interconnect. The routing network is split in vertical and horizontal channels. Switch blocks allow interconnecting the channels at their intersections.*

4.2-2 Island-style architecture

The island-style architecture is the routing interconnect of most, if not all, modern Xilinx and Altera FPGA devices, and is one of the most straightforward interconnect organization. Figure 1-13 gives an overview of the structure of island-style architectures. The logic fabric itself is made of a regular grid of soft logic and hard blocks, each of these elements spanning one or more cell of the grid in height. The logic fabric is often modeled in columns, even if the placed-and-routed circuit realities are more fuzzy. This logic grid is surrounded by a mesh routing network, hence the name *island-style*.

An island-style interconnection network can be schematized in three components. Horizontal and vertical channels forms the bulk of the interconnect and act as the connection blocks of the interconnection network: a given logic block is connected to the channels directly neighboring it. In addition to the routing channels, switch blocks realize the interconnection between horizontal and vertical tracks, given a specific interconnect topology.

Figure 1-14 depicts three main switch block topologies in use in island-style FPGA architectures. The disjoint switch [LB93], shown in Figure 1-14a, implements a symmetric switch pattern with $F_S = 3$. Each track of the horizontal and vertical routing channels is numbered and a given track i can only be connected to the other i tracks. This switch creates a set of i *disjoint* routing networks in the sense that a net can only be routed on the same track i, and can never be interconnected to another track. The Wilton switch topology [Wil97], illustrated in Figure 1-14b, is similar, with $F_s = 3$, but the connections are rotated. The rotation avoids the disjoint network problem. The Wilton switch block has been found to require 5% fewer



(a) **Disjoint [LB93]** (b) **Wilton [Wil97]** (c) **Universal [CWW96]**

Figure 1-14 – **Switch block topologies.** *Various switch block topologies have been explored, the disjoint, Wilton and universal models are three possible switch block implementations. While the disjoint topology may simplify the global-to-detailed routing step of the CAD flow, the Wilton topology shows an increased flexibility which can lead to fewer tracks used. The universal model meets the dimensional constraint but fails to be easily expendable to segmented routing schemes.*

tracks [Wil97] than its disjoint equivalent, for a greater flexibility of track changes in the detailed routing. A third topology, the universal switch block [CWW96], schematized in Figure 1-14c, guarantees that any set of two-pin nets can be routed through the switch block as long as the number of nets per side does not exceed W. This constraint, known as the *dimensional* constraint, is not met for all switch blocks, in particular the disjoint switch block. Additionally, the universal switch block has been designed for single-length wires, and its applicability to the segmented routing of modern devices is not immediate.

4.3 Routing structure

Apart from the coarse organization of the interconnection network itself, the performance of this network also relies heavily on finer details. In SRAM FPGA devices, the most straightforward approach to implement the configurable switches is to use pass-transistors, which act as electrically controlled buttons, opening or closing a wire depending on the associated configuration bit. Previous sections already drafted the drawbacks associated with pass-transistors, notably the insertion of a parasitic capacitance. The added capacitance degrades the signal crossing the switch, leading to the necessity of inserting buffers in the signal path to realize this bi-directional routing.

4.3-1 Bi-directional routing

A bi-directional routing architecture in SRAM FPGAs relies mostly on tristate buffers and pass-transistors to operate. The buffers help to regenerate the signal to

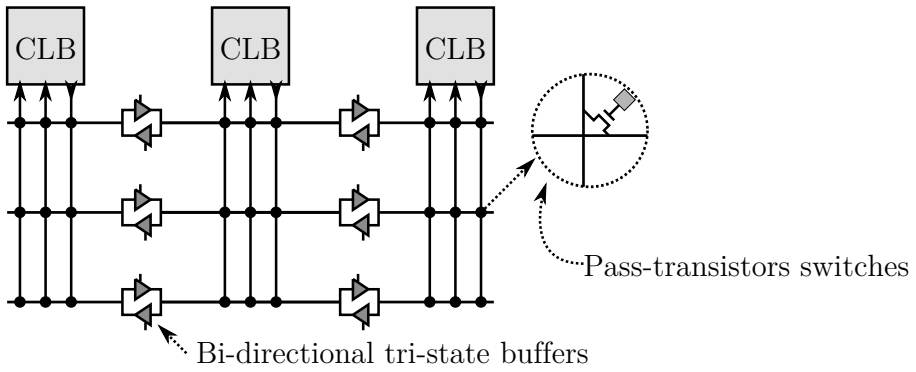


Figure 1-15 – **Implementation of bi-directional routing with pass-transistors and buffers.** *In bi-directional architectures, any routing segment can be used in any direction, and back-to-back tri-state buffers must be used to regenerate the signal.*

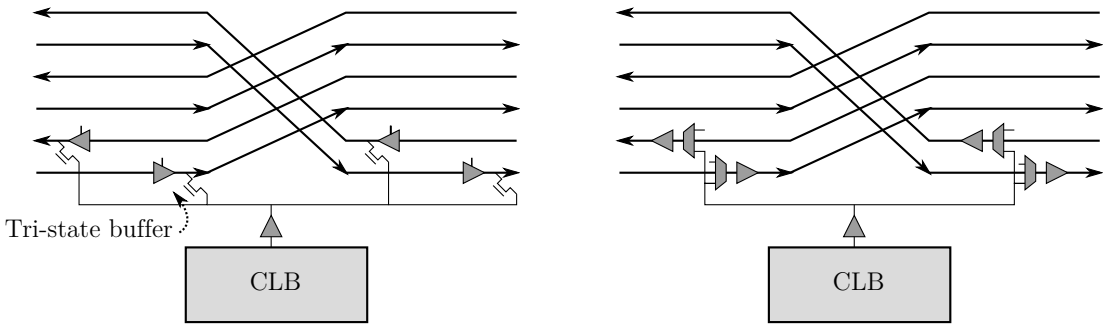
avoid long chains of pass-transistors and signal losses. Since there is no such thing as a bi-directional tristate buffer, they are created from two head-to-tail tristate buffers, as shown on Figure 1-15, and a programmable memory point configuring which of the buffers is active, depending on the signal direction.

The direct consequence of such routing structure is that half of the tri-state buffers are not used at runtime for a given loaded configuration on the SRAM FPGA. Given the area occupancy of these buffers, this leads to an enormous waste of logic area for the little added benefit of bi-directional routing. The unused tristate buffers nonetheless increase the fanout of upstream cells, increasing the wire capacitance and thus reducing the interconnect performance. The use of unidirectional wires in the detailed routing architecture circumvents these issues.

4.3-2 Unidirectional routing

Unlike bi-directional wires, a directional wire segment can only be driven in one direction, leading to a 50% decrease of the number of tristate buffers needed, as previously evoked. Lemieux *et al.* [Lem+04] explored the shift of major FPGA manufacturers away from bi-directional wiring and noted an improvement of 32% of the area-delay product with directional wiring.

Figure 1-16 illustrates two possible implementations of unidirectional drivers detailed in [Lem+04]. With the first one, known as *directional tristate* (dir-tri) drivers, the logic block output drives the wire segment through a pass-transistor, and these wire segments are additionally driven by adjacent segments through the switch block output multiplexers. In this case, we still have multiple drivers to each segment and the potential problems of their combined parasitic capacitance. An alternative is to implement the directional wires in such a way that the logic block outputs are connected to the switch block multiplexers inputs. This single-driver implementation allows driving the wire segments without a tristate approach and therefore increases the signal strength, but can be architecturally limiting since logic blocks must be physically close to the driving switches.



(a) **Directional tri-state** (b) **Directional single driver**

Figure 1-16 – **Directional tristate and single driver implementations of unidirectional output drivers [Lem+04].** *Directional routing can be implemented in an architecture using pass-transistors and tri-state buffers for each logic block input or output. Another single driver implementation may be considered, which complexifies the architecture design but cancels the need for tri-state buffers.*

5 Conclusion

This section detailed the inner working of FPGAs and laid an overview of all the trade-offs considered in the design of FPGA architectures. The most important aspects of the device, the interconnection network and the logic block architecture, have to be carefully selected to provide an homogeneous device balancing performance and accessibility. Most of the detailed routing features (directional wiring, segmented routing, etc.) and the most complex hard blocks (DSP, processors) would however not be beneficial to the end user without powerful accompanying CAD tools, specially developed for the target architecture. Nonetheless, the selection of the finest details of the architecture always depends on the targeted use cases and requires a huge amount of benchmarks to be able to explore the various design choices, which makes successful manufacturers in a good position to develop new trends in FPGA design, thanks to their massive amount of customers and their heterogeneous applications.

Chapter TWO

RUNTIME RECONFIGURATION AND ROUTING OF FPGAS

Contents

1	Runtime reconfiguration	**33**
	1.1 Reconfigurability of FPGAs	33
	1.2 Early work on runtime reconfiguration	33
	1.3 Partial reconfiguration in modern devices	34
2	Task relocation/migration	**35**
	2.1 Vendor-supported partial reconfiguration	35
	2.2 On hacks of modern devices	35
3	Run-time routing and communications	**39**
	3.1 Bus macro	39
	3.2 Configurable communication network	40
	3.3 Bit-stream merging	40
	3.4 Just-in-time routing	40
	3.5 Conclusion	41
4	Placement and routing for FPGAs	**41**
	4.1 High-level synthesis and non architecture-dependent optimizations	42
	4.2 FPGA architecture mapping	43
	4.3 Placement	43
	4.4 Routing	44
	4.5 Verilog to Routing	48
5	Conclusion	**51**

Abstract

The runtime reconfiguration feature of FPGAs is the core of the work of this thesis. This chapter details the legacy and state-of-the-art techniques involved with the dynamic usage of reconfigurable devices, from the earliest work on the Xilinx

XC6200 till the latest FPGA devices. Additionally, the role and evolution of CAD tools for the placement and routing of FPGA applications is detailed.

1 Runtime reconfiguration

Run-time reconfiguration provides multiple benefits over static applications in Field-Programmable Gate Arrays (FPGAs). In designs where all the application sub-functions are not used at the same time, run-time reconfiguration provides a way to time-share the FPGA logic resources, potentially leading to a smaller budget in terms of resources. Other applications may benefit from an increased adaptability to their environment or to run-time events. For example, a dynamic wireless transceiver can switch its physical layer between multiple implementations using run-time reconfiguration, with regards to the link quality. Additionally, hardened applications where data integrity is critical or systems evolving in difficult environments can take advantage of run-time reconfiguration to prevent data corruption in case faults are detected.

1.1 Reconfigurability of FPGAs

FPGAs have proven their versatility since their introduction in the mid-1980s. From the designer standpoint, they require less time and money to be invested than a custom Application-Specific Integrated Circuit (ASIC) design using standard cells. FPGAs provide much higher performance than software for most parallelizable applications, but do not attain the same level of flexibility. Run-time reconfiguration techniques in FPGAs have been studied for over 20 years and intend to fill the flexibility gap between reconfigurable hardware and software implementations.

Despite the fact that FPGAs are configurable devices from their debut, they did not implement run-time reconfiguration *per se*. Indeed, they are loading their configuration data at start-up, but did not originally provide means of reconfiguration at run-time. This has limited the flexibility of reconfiguring the FPGA device, since a system which needed to switch application was forced to fully reconfigure the FPGA configuration memory for that purpose. A full reconfiguration takes time, considering configuration data in the order of tens of megabits for the whole device, which makes application switching prohibitive in most cases. Run-time reconfiguration is thus associated to the concept of partial reconfiguration, in which a portion only of the FPGA can be shutdown and reconfigured, while the remaining active area of the device is left running.

1.2 Early work on runtime reconfiguration

Early analysis of run-time reconfiguration techniques on FPGAs targeted the Xilinx XC6200 series devices. Introduced in 1995, the XC6200 FPGA family indeed featured substantial improvements on its architecture, dedicated to run-time reconfiguration [CKW95]. These FPGAs were partly designed to act as CPU co-processors, hence their internal registers were memory-addressable. Additionally, they were the first FPGAs to feature a configuration memory writable at the speed

of a Static Random Access Memory (SRAM) at run-time, therefore guaranteeing the reconfigurability of the device.

In [Bre96], a virtual hardware operating system is proposed to manage these new hardware resources. The paper also discusses the new paradigms associated which the XC6200, notably from a hardware/software cooperation standpoint. At this point, the XC6200 reconfigurable array is so regular that the proposed operating system is able to organize the Swappable Logic Units (SLUs) (a fixed-area set of logic elements presenting a set of inputs and outputs, whose interface is fixed) of a task as desired, to the exception of SLUs requiring long-lines of length 4, 16 or 64, sparsely available on the design. The XC6200 FPGA coupled with Brebner's virtual hardware operating system is thus the first partially reconfigurable FPGA with online, and seamless, task placement.

Hartenstein *et al.* [HHG98] proposed a flow to answer another problem with reconfigurable architectures, transverse to the reconfiguration management: the design flow. Applied to the XC6200 series, they came up with a solution where the data-path and the control parts are separated into two different partitions for easier reconfiguration of the data-path part, which represents most of the logic area.

1.3 Partial reconfiguration in modern devices

FPGAs have evolved on multiple aspects since the XC6200 series, as detailed in Chapter 1 of this thesis. Most recent task placement and relocation mechanisms proposed for newer FPGA architectures mainly focused on Xilinx Virtex families and Altera Stratix V devices which allow for partial reconfiguration. For most of these recent vendor-supported hardware systems implementing partial reconfiguration, a set of bit-streams must be compiled offline and are defined by one application task and a given location in the FPGA fabric. Altera and Xilinx implement the partial reconfiguration through the definition, at the design stage, of Partially-Reconfigurable Regions (PRRs) among the fixed logic forming the rest of the design [Alta][Xilc]. In this case, a set of partial bit-streams $\{BS_1, BS_2, \ldots, BS_N\}$ is generated for each corresponding application task $\{T_1, T_2, \ldots, T_N\}$ which may be loaded onto a given predefined PRR. The routing, and thus the communication, between the task and the static logic regions of the FPGA is handled by Look-Up Tables (LUTs), placed at fixed positions in both static and fixed regions, acting as interfaces. Dedicated software is needed to manage the scheduling and region occupancy of the PRRs [Com+02].

In most of the basic partial reconfiguration scheme previously described, the sets of bit-streams that can be configured on each PRR are disjoint: multiple bit-streams can be placed on a single PRR, but a given bit-stream only targets one PRR. In many cases, it is desirable to be able to place a task on multiple PRRs, such that hardware utilization and runtime flexibility are maximized.

2 Task relocation/migration

Task relocation addresses the problem of placing a set of tasks $\{T_1, T_2, \ldots, T_N\}$ on multiple PRRs $\{PRR_1, PRR_2, \ldots, PRR_N\}$ dynamically at runtime. Individual applications can benefit from this flexibility by using the FPGA to perform different tasks at different times.

This hardware customization can be used by the application to adapt itself to runtime conditions and improve its efficiency, or even to fit multiple applications on the device simultaneously by swapping the configurations in and out. Additionally, Montminy *et al.* [Mon+07] used task relocation to move PRRs to unused columns in a Virtex-II device to provide fault tolerance. Unfortunately, the added value of task relocation comes at the cost of increased overheads in terms of reconfiguration time, logic area or energy consumption, as discussed in the following subsections.

2.1 Vendor-supported partial reconfiguration

The model currently used by FPGA vendors [Alta][Xilc] to implement runtime reconfiguration requires the application designer to generate, for each task T_k, a set of bit-streams $\{B_{k1}, B_{k2}, \ldots, B_{kM}\}$ corresponding to the M PRRs the task can be loaded on. The main drawback of this model resides in the number of bit-streams that need to be stored in an external memory so that they could be loaded on demand. At runtime, a set of N tasks to be placed on M different PRRs requires $N \times M$ different bit-streams whose size may vary from hundreds of kilobits to few megabits on modern devices, depending on the PRR size. This leads to a waste of memory, and limits the partial reconfiguration granularity as the partition size cannot grow arbitrarily. The partitions should be large enough to hold the logic of the different tasks, but as small as possible to provide the maximum flexbility in terms of task placement, as PRRs cannot overlap.

2.2 On hacks of modern devices

Because of the limitations of the vendor supplied partial reconfiguration tools, most of the work on partial reconfiguration on modern devices focused on enhancing the relocation capabilities of the target architectures.

2.2-1 Relocation on homogeneous fabric

Horta *et al.* [HL+01] provide a software solution, PARBIT, for the original Virtex series FPGAs to allow for the relocation of partial bit-streams on the logic fabric. In the original Virtex architecture, the logic resources are organized in columns of different types, each containing a single feature of the FPGA: the clock, the logic blocks, the content and interconnect of Random Access Memory (RAM) blocks, and

the input and output blocks. Xilinx provides an interface, SelectMAP, to independently reconfigure each of these columns. PARBIT is able to take the configuration bit-stream of a given column and modify it to make it loadable on another column of the same type. Indeed, the Virtex configuration data contains metadata (addresses and Cyclic Redundancy Check (CRC) checksums) which makes a partial bit-stream unique to a specific position in the chip.

Although powerful in the manipulation of complete and partial bit-streams, PARBIT is an offline software solution, which makes it nonviable in practice for dynamic online task migration. Kalte *et al.* [Kal+05] enhanced the PARBIT software into a hardware system directly implantable on the Virtex logic fabric, REPLICA, which is able to manipulate partial bit-streams online, during the bit-stream download process, hence its name of REPLICA *filter*. The Virtex architecture offers multiple interfaces for partial reconfiguration, among which the SelectMAP interface was chosen for REPLICA because of its bandwidth (50 MB/s with an 8-bit parallel bus). The bit-stream manipulation filter was originally implanted on a Complex Programmable Logic Device (CPLD) external to the FPGA, but was proven to be synthesizable on the Virtex logic fabric at a frequency of 58 MHz[1], using only 490 slices (2.5%) of a XCV2000E, a rather small FPGA of the Virtex-E family [Xild], for both the filter and its accompanying configuration manager.

The REPLICA filter and its configuration manager enable the online relocation of partial bit-streams on Virtex devices on one direction only: the FPGA needs to be configured column-by-column. The 1-D relocation has been studied [Kal+04] and shown to have at most a 5% increase in power consumption and critical path delay for medium and large hardware tasks. Small tasks show increases of up to 96% in critical path delay in the extreme case of an 8-bit adder constrained in a single column, due to the high logic occupancy and routing resources congestion.

This technique was later improved by Corbetta *et al.* [Cor+07] who introduced the BiRF hardware system and its software version, BAnMaT Light. An optimized version of BiRF was synthesized to work at up to 133 MHz on a Virtex-II Pro FPGA, although the SelectMAP reconfiguration interface of the device is still limited to a 50 MB/s bandwidth, and the reconfiguration time of BiRF is dependent on the time required to read the bit-stream and replace the position-dependent fields.

2.2-2 Handling heterogeneous architectures

The Virtex and Virtex-E series FPGAs are limited in terms of available fixed-function blocks. The relocation methods previously described supports 1-D relocation on the logic fabric of these devices but only considers logic columns, which is a serious limitation for complex applications requiring, for example, RAM blocks on the logic fabric. The Virtex-II family of FPGAs introduced in 2002 by Xilinx featured additional heterogeneous blocks [Xile], notably hard 18×18 multipliers,

[1] 50 MHz are required to match the 50MB/s peak bandwidth of the SelectMAP interface of the Virtex-E FPGAs [Xild].

and even PowerPC hard processors for the Virtex-II Pro and Virtex-II Pro X FPGAs [Xilf]. Those blocks add up for more efficient on-board computations since arithmetic operations can be synthesized using a few multipliers rather than a lot of logic resources, but are an obstacle to existing relocation filters.

Krasteva *et al.*, [Kra+06] proposed a software system, pBITPOS, able to relocate a hardware task at another position of a Virtex-II device and which takes fixed-function blocks into account, as illustrated in Figure 2-1. Their tool is thus able to reallocate RAM blocks and multiplier blocks as long as the pattern of successive columns of the partial bit-stream can be found elsewhere on the logic fabric.

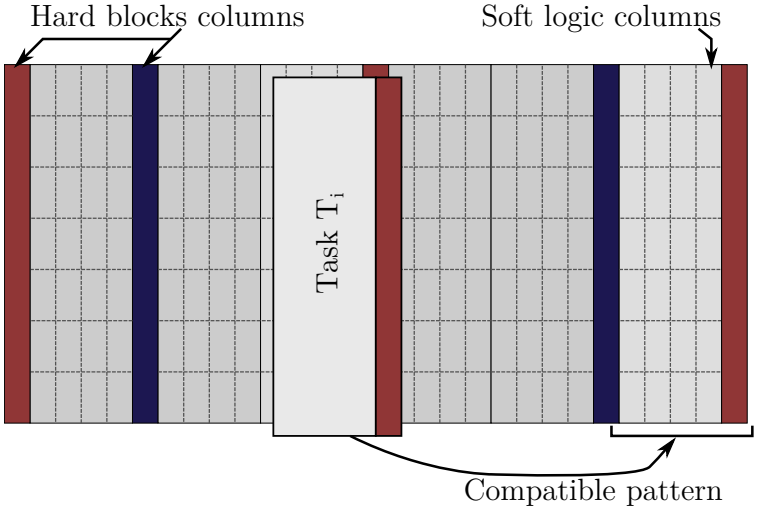


Figure 2-1 – **Compatible patterns on a logic fabric for task relocation.** *The geometry of task T_i as well as the order of soft logic and hard block columns composing it must be preserved to allow for its relocation. Considering these constraints, the technique in [Kra+06] can only relocate T_i to the rightmost position.*

Kalte, *et al.* [KP06] presented an enhanced version of their bit-stream manipulation filter, adapted and optimized for the Virtex-II Pro devices: REPLICA2Pro. In the same way as pBITPOS, this filter handles the relocation of complex hardware task containing fixed-function blocks if they can find a partition where the fixed-function blocks are in the same relative position as in the original partial bit-stream. However, REPLICA2Pro is synthesized directly on the Virtex-II device, using only 570 slices, and can maintain the 50 MB/s reconfiguration rate of the SelectMAP interface of the FPGA while running in real-time during the reconfiguration process, which is not the case of pBITPOS. Additionally, Kalte *et al.* propose an evolution of the Virtex-II SelectMAP internal operation based on the finding that most of the SelectMAP and the REPLICA2Pro architectures are very similar if not completely identical, which is a step towards the development of architectures inherently supporting efficient partial reconfiguration and task relocation.

Touiza *et al.* [Tou+12] proposed a method similar to the previous partial bit-stream filters for relocation, at the difference that their tool, OORBIT (Offline/Online Relocation of Bit-streams), performs some steps required for the bit-stream manipulation offline, at design time. Particularly, the target address and Cyclic

Redundancy Check (CRC) fields needed between the Virtex series FPGA partition data are pre-calculated offline for the set of desired positions and stored in configuration files to be used at runtime when the hardware task is finally placed. OORBIT is thus able to provide high-troughput relocation of tasks across partially reconfigurable regions, capped by the maximum 100 MB/s throughput of the SelectMAP reconfiguration interface in Virtex 6 devices. As most of the time consuming calculations are done offline, the only task required online prior to the relocation is to replace some fields of the bit-stream, therefore the hardware implementation of OORBIT announces speed-up of up to $8,000\times$ compared to BiRF.

Although the previously detailed solutions are able to handle fixed-function blocks, they are limited to a subset of the possible positions for relocation where the relative place of fixed-function blocks is the same as in the task, just like in the case of a homogeneous fabric. The only work which examines the relocation problem in the case where the target partition is not identical to the origin partition is Becker *et al.* [BLC07]. As shown in Figure 2-2, their solution ignores fixed-function blocks, but is able to find more possible target locations for a task containing only logic. It does so by finding partitions where any fixed-function block can fill the gap left in between logic columns instead of finding the exact same set of blocks. In this case, only the relative position of fixed-function blocks matters when evaluating possible target positions, the exact type of these blocks is not taken into account and a few on-the-fly bit-stream modifications can accommodate to the differences between partitions.

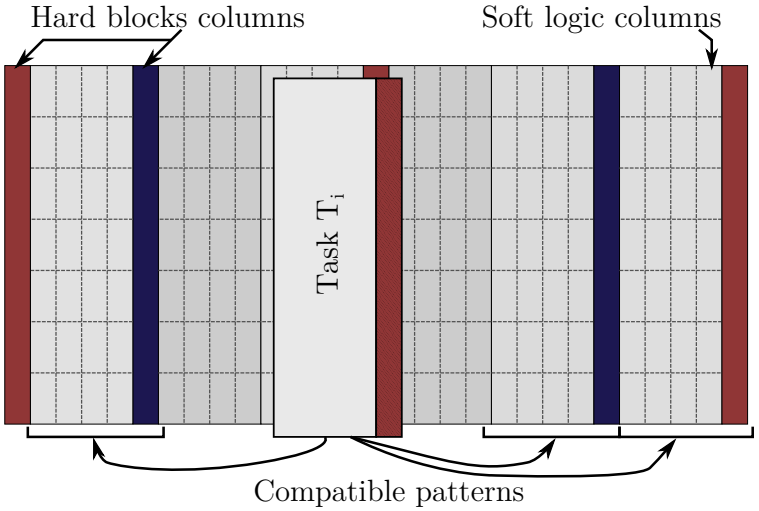


Figure 2-2 – **Becker *et al.* [BLC07] solution to logic fabric heterogeneity.** *The solution brought by [BLC07] handles more* compatible patterns *on the logic fabric, but they are not usable by the tasks. Considering task T_i, any hard block can fit its hard block slot, as long as its relative position to the soft logic is kept.*

Beckhoff *et al.* [BKT14] recently introduced a novel method to handle the placement of bit-streams at multiple positions on the logic fabric, thanks to the use of a finite set of target locations. At design time, the vendor tools are used to generate the multiple bit-streams corresponding to each desired location of the hardware

tasks. These bit-streams are made of position dependent (e.g. clock configuration bits, CRC fields, specific local routing, etc.) and position independent sections. These sections are mixed together into one *portable module* containing the identical data reproduced in every bit-stream and the local differences allowing generating the bit-streams specific to each location, which are further compressed using the Lempel-Ziv-Storer-Szymanski (LZSS) algorithm. This work is the first to exploit the similarities between bit-streams logic data to enhance compression. It and has two major advantages over existing techniques. First, it is not tied to a specific FPGA architecture, since little to no knowledge of the underlying bit-stream format is needed to perform the calculation of the differences between the considered bit-streams. Secondly, it is among the first to truly handle the heterogeneity of the target logic fabric, in the sense that the partial bit-streams generated are inherently not limited to homogeneous partition, albeit limited by the vendor tools.

3 Run-time routing and communications

The self-reconfiguration capabilities of FPGA devices enables the possibilities of dynamically placing hardware tasks on their logic fabric at runtime. Multiple state-of-the-art solutions have been explored in the previous section, which cover the problem of being able to reconfigure the target logic fabric in order to load a new hardware task. In order to interact with the surrounding world, this newly placed task needs communication capabilities, either intra-chip (with other parts of the fabric) or extra-chip (with components outside the FPGA).

3.1 Bus macro

Most flexible state-of-the-art techniques use a form of additional routing. In Mignolet *et al.* [Mig+03], hardware tasks are connected to a fast packet-switching interconnection network. Using a fixed, routed interconnection network alleviates most of the problems associated with inter-task communication in a reconfigurable system. The fixed-position communication interface minimizes the runtime overhead induced by any additional routing that could be required to connect a hardware task to the communication system. In the case of Mignolet *et al* [Mig+03], the packet-oriented network allows to seamlessly address a task no matter where it is loaded in the heterogeneous reconfigurable system, as tasks could either be loaded on the hardware fabric or in the software system.

Older techniques addressing the communication problem between hardware tasks in the context of partial reconfiguration were based on a standard bus interface approach. The fixed logic glue between partially reconfigurable regions comprises the necessary logic for bus communication as well as fixed bus macros on both static and partially reconfigurable regions to realize the connection with newly placed hardware tasks, as long as the partial bit-stream comprises this macro at the correct position. This is the solution recommended by Xilinx since the early introduction

of its partial reconfiguration flow, and it is used by many of the work on bit-stream relocation involving Virtex devices, such as the previously discussed work of Kalte *et al.* [Kal+05] or Beckhoff *et al.* [BKT14]. The partial reconfiguration frameworks proposed by Carver *et al.* [CPF08] and Tan and DeMara [TD08] also rely on bus macros.

3.2 Configurable communication network

Bobda *et al.* [Bob+05], and Majer *et al.* [Maj+07] designed the Erlangen Slot Machine, an FPGA-based reconfigurable computer which aimed to answer architectural problems when dealing with partial reconfiguration. One of the enhancement of this hardware system over the use of bus macros is the introduction of a fully configurable (external) crossbar in addition to standard PRR-to-PRR buses, which allow for a PRR to communicate with its immediate neighbors. The crossbar is the backbone of the communications in the Erlangen Slot Machine and answers the dilemma of distributing the system inputs and outputs among the hardware task positions.

In Flynn *et al.* [FGRG09], the clock signals are routed at runtime to the PRRs to ensure a regular clock distribution across the newly placed task in Virtex 4 and 5 devices.

3.3 Bit-stream merging

Sedcole *et al.* [Sed+05] explored 2-D reconfiguration techniques in Virtex FPGAs to provide more flexibility on the task placement. The column organization of Virtex and Virtex II devices forced previous work to limit the reconfigurability to column boundaries, as presented in the previous section. The 2-D method presented by Secdole *et al.* relies on bit-stream merging techniques, which allows multiple tasks to share the same column, as long as the sets of used resources are disjoint. The merging method also enables a reconfigurable PRR to be merged with static logic already allocated. The bit-stream merging is however costly, since an increase in the reconfiguration time of up to four times is observed.

3.4 Just-in-time routing

On the other side of the spectrum of runtime routing techniques, in comparison to the previously described online methods, Lysecky *et al.* [LVT04] developed the Riverside On-Chip Router (ROCR), a just-in-time (JIT) compiler. ROCR realizes the placement and routing step online, as a software algorithm implemented on a processor system. ROCR is ten times faster in average than the state-of-the-art Versatile Place-and-Route (VPR) software at that time, while consuming thirteen times less memory for similar results with at most a 32% increase in critical path delay. Although greatly versatile, solutions akin to ROCR are also extremely slow

in the domain of dynamically reconfigurable hardware. Small tasks from the MCNC benchmark suite can take up to 13 seconds to be placed and routed at runtime for a specific position of the logic fabric, while the configuration of the FPGA partition itself is in the order of ten milliseconds.

Most of the time consumed by ROCR is dedicated to the iterative routing process, this is also the case for other open-source and commercial FPGA routers such as VPR or the Xilinx and Altera tool flows. In the case of bit-stream merging techniques, additional routing can add as much as 97 seconds to the migration time of a hardware task [SF10].

3.5 Conclusion

Most of the techniques detailed in this section offer a relocatability of FPGA modules focused on existing architectures. The main downside of such an approach is the dependency of these methods to the considered device, of which the finer details are not known to the designer. The approach of the work of this thesis on the placement of hardware tasks is conducted from another point-of-view, using enhancements at the architecture level rather than relying on existing devices. A custom, position-independent, representation of the configuration bit-stream prevents the need of a complete online JIT router to place the tasks. Optimized algorithms allow for the decoding of such bit-streams almost transparently, whereas bit-stream manipulation filters need to rely on a closed bit-stream format. Moreover, using a known abstract representation of the configuration bit-stream enables the representation of logic and routing data in a smarter way, allowing a higher compression ratio than generic compressors.

4 Placement and routing for FPGAs

In the same manner that the architecture of FPGAs has evolved since their introduction thirty years ago, the tool flow accompanying these devices matured to put less stress on the FPGA designer. As the complexity of applications targeting reconfigurable FPGA systems grew over the years to include domains ranging from signal processing to special-purpose processor implementations, the practicality of mapping the designs to the targeted systems by hand has been limited by the time needed to analyze the multiple optimizations and the subsequent trade-offs required for these implementations. This limiting manpower factor led to the development of increasingly advanced algorithms at various stages of the FPGA design flow, from the designer description of a complex system down to its actual implementation on the final hardware system. As such, the FPGA design flow is nowadays mostly standardized [CCP06], with each intermediate step clearly distinguished. Figure 2-3 illustrates this flow, from which little deviations can be encountered.

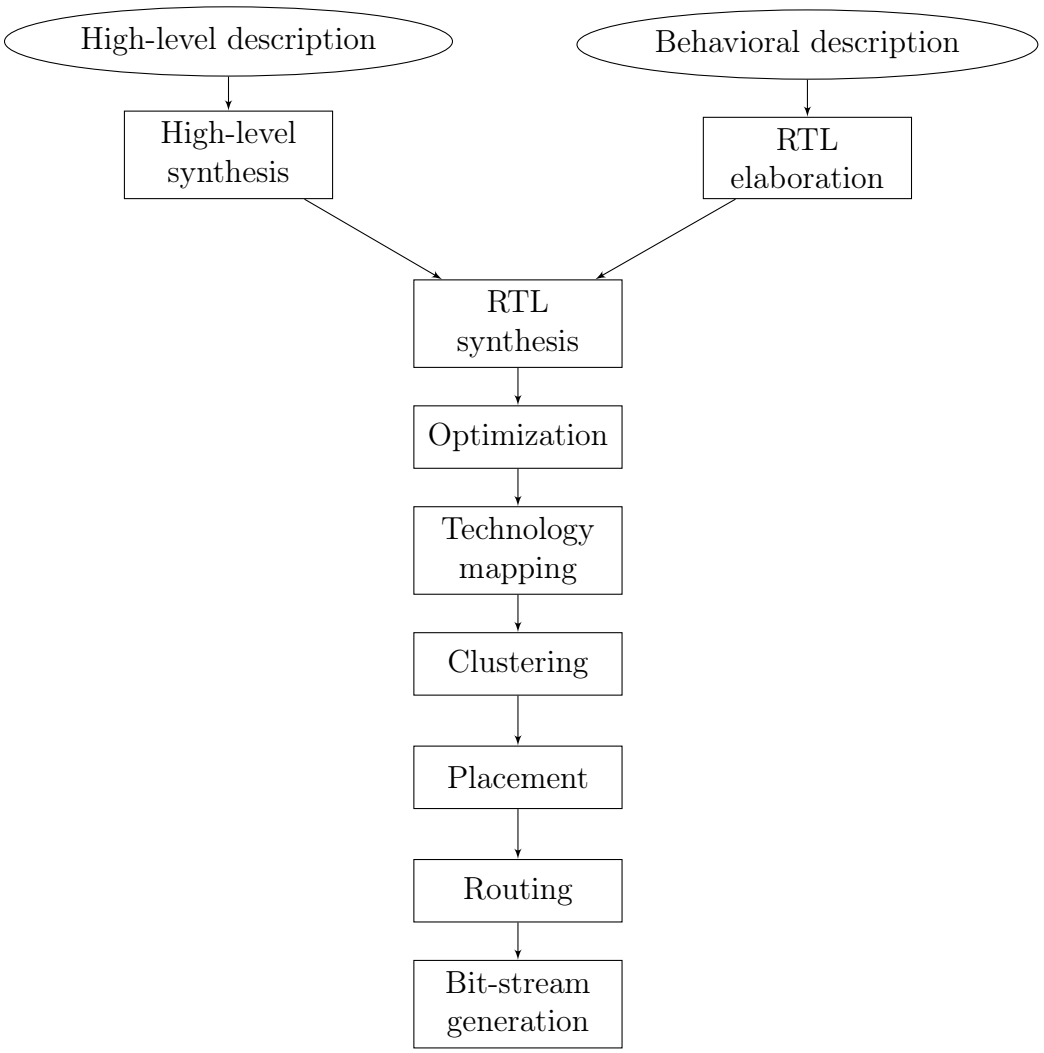


Figure 2-3 – **A typical FPGA design flow**

4.1 High-level synthesis and non architecture-dependent optimizations

The desired application targeting an FPGA hardware platform is usually described using either a high-level language (e.g. C, C++) or a behavioral hardware description language (e.g. VHDL, Verilog, AHDL). Unlike software programs whose description is compiled to a set of instructions which are then decoded and executed by a microprocessor, the description of a hardware design is translated into complex data-paths made of registers, Arithmetic and Logic Units (ALUs), memories, multiplexers or other high-level building blocks. Additionally, control parts take care of the scheduling of these components. This combination of high-level data-paths and control elements forms a Register Transfer Level (RTL) description of the design. The RTL description is generic and not tied to a specific hardware technology at this point of the design flow: an application will probably share most of the same RTL description if it targets an FPGA or an ASIC.

The RTL description may at this point be optimized for both data-path and control parts. The data-path optimization uses techniques such as expression optimization, operation sharing or constant propagation in order to reduce the data-path footprint in terms of components or complexity. For the control part, retiming algorithms or finite state-machine encoding schemes may be applied to enhance the overall scheduling of the design and reduce the use of operators in the data-paths or increase the performance. Every general optimization at this point or target-specific optimization later in the flow is based on trade-offs specified by the designer over criterion such as area occupancy, power consumption and performance (throughput, latency).

4.2 FPGA architecture mapping

Starting from the technology mapping step, the flow becomes very specific to the FPGA and to its architecture. The technology mapping step maps the various high-level elements comprising the optimized data-paths to dedicated hardware structures of the FPGA logic fabric: embedded multipliers, adders and their dedicated carry-chains, embedded memories. Additionally, the control logic and portions of the data-path which can not fit embedded accelerators of the logic fabric are mapped to the logic elements of the FPGA. This logic mapping implies that the corresponding logic equations should be optimized to fit the available logic elements. Most importantly, the size of the LUTs is taken into account to only have k-input, or less, logic equations, where k is the size of the LUTs of the FPGA. In the case where a primitive equation has more inputs than the available LUTs, it will be split and routed over multiple logic elements.

4.3 Placement

At this point of the flow, the design is a netlist of allocated FPGA resources: memories, embedded accelerators, logic elements, inputs and outputs. The placement algorithm takes care of the final position of these elements on the final logic fabric while keeping track of the relation between each of them to be able to meet the design performance trade-offs.

Most commercially available FPGA architectures expose some form of logic element locality, as detailed in Chapter 1. Incidentally, the placement step of the FPGA design flow is often preceded by a clustering step in order to group highly-correlated logic elements together. The clustering allows for the reduction of routing congestion by exploiting local interconnections between groups of logic blocks.

The objective of the placement step is to place each element of the netlist so that the subsequent routing step can be performed with the available routing resources. Unfortunately it is often impossible in practice to have a clear metric of the routability of a given placed netlist, given the complexity of modern FPGA network.

In the case of island-style architectures, most placement algorithms are based on variants of the simulated annealing algorithm [KGV83]. The placement step starts from an initial semi-random placement of the netlist on the logic fabric. Iteratively, pairs of blocks are swapped and the whole placement is evaluated thanks to a cost function which varies among the algorithm implementations. The simulated annealing algorithm searches a globally optimal minimum of the placement cost, which should therefore reflect all the legality constraints of the FPGA architecture. The placement step is operated distinctly from the routing since these two optimization problems are NP-hard [WTMS96].

4.4 Routing

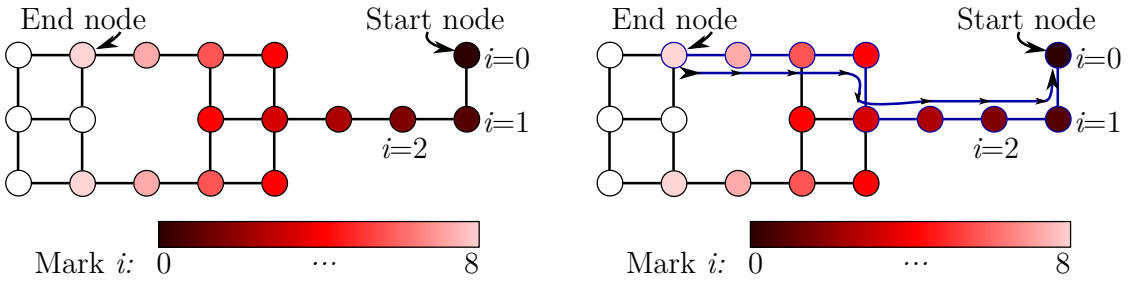
The routing process maps the interconnections between the various elements of the netlist to the routing resources available on the FPGA. In the case of an island-style architecture this is the set of routing channels and switch boxes to traverse to create the paths required for each net of the design.

The routing process problem resides in the finite set of routing resources available to route all the nets of the design. As the complete set of nets is routed on the fabric, some highly-congested parts will be deprived of available resources, while less used portions of the design will still offer many more available segments which can be considered as wasted. Unlike placement algorithms, the routing algorithm may fail when the pool of available routing resources is exhausted in certain locations of the design.

Given the regular topology of FPGA architectures, such as the island-style architecture, the routing step of the flow is really similar to the one performed on standard-cell ASIC design and Mask-Programmable Gate Arrays (MPGAs).

Routing the netlist can be assimilated to a graph problem where the net endpoints are nodes of the graph (inputs, outputs, wire segments) and the programmable switches between them are the edges. Graph theory problems have been extensively researched since at least the work of Leonhard Euler in the XVIIIth century on the seven bridges of Königsberg. The basics of the FPGA routing problem are solved by *maze routing* algorithms [Lee61] which are generalizations of Dijkstra's shortest path algorithm [Dij59].

The *maze routing* algorithm, detailed in Algorithm 1 finds the shortest path between two nodes in the routing graph. The first stage of the algorithm performs a wave propagation of incremental marks starting from the start node, until the end node gets marked, as demonstrated in Figure 2-4a. The second stage, shown in Figure 2-4b, backtracks the shortest path from the end node up to the start node. These operations are sequentially repeated for every net of the design. As each net will occupy resources in the graph, some of them will fail to be routed, and the routing process will have to rip up some selected nets to route them otherwise and allow more nets to be routed, iteratively.



(a) **Wave propagation** (b) **Backtracking**

Figure 2-4 – **Illustration of the maze routing algorithm.** *The first step incrementally marks neighboring nodes with an increasing i mark from the start node, until the end node gets marked. The second step builds the final path starting from the end node by backtracking decreasing marks. The maze algorithm ensures that the shortest possible path is built, although multiple of them are possible for a given routing graph.*

4.4-1 Global and detailed routing

As the routing complexity increases rapidly with the size of the considered design, it becomes impractical to solve the routing as a whole in one pass. Early FPGA routers split the routing process into two separate passes. The first one, the global routing, coarsely determines in which areas a given net will be routed, while considering local congestion and the timing criticality of the net. This first pass will then produce an approximate topology of the final routing with each net routed through the various available channels of the considered interconnect architecture.

The second pass, the detailed routing, will make the previously defined coarse routing compliant with the design rules defined for the architecture (e.g. in terms of number of nets per channel).

CGE and SEGA Early detailed routers such as CGE [BRV92] and SEGA [LB93] were respectively proposed by Brown *et al.* and Lemieux *et al.*. The main difference brought by SEGA, apart from the cost function, is the support of variable-length segments in the FPGA architecture, which can be exploited to route nets of adequate size more efficiently. Both of these detailed routers use the same two-phase algorithm. In a first phase, an exhaustive list of possible detailed routes for a global routing is built up and yields a set of alternative routes for each net of the coarse graph. The second phase forms the connections in the detailed graph by first selecting the detailed route of the lowest cost, then marking this path as routed in the graph. The other alternative routes of this path are then discarded, as well as the other paths which shared the same segments as those allocated to the lowest cost route. If a given route loses its latest possible path, it is deemed *unroutable*, which may lead to a routing failure. Such routes with only one possible path remaining are given a high priority in order to circumvent this problem.

The cost functions of CGE and SEGA thus plays the most important role in

Data: Start and end nodes
Result: Shortest route between start and end nodes

Remove the mark of every node in the graph;
Mark the start node with 0;
i ← 0;
repeat
  Mark every unmarked and unblocked neighbors of nodes marked by i with i + 1;
  i ← i + 1;
until *End node reached* ***or*** *No more unmarked neighbor*;
Start from the target point;
repeat
  Go to a neighbor with a lowest mark than the current node;
  Add this node to the path;
until *Start node reached*;
Mark the nodes in the path as blocked;

Algorithm 1: Maze routing algorithm

the routing process of all coarse graphs of the global routing. CGE uses the cost function

$$\mathrm{C}_f(p) = \sum_i \sum_j \frac{1}{\mathrm{alt}(e_j)} \tag{2-1}$$

to determine the cost of a path p, based on its i edges $(e_1, e_2, \ldots, e_i)$. $\mathrm{alt}(e_j)$ represents the number of edges which could be used instead of e in a set of alternative paths j. Thus, the more an edge is needed for a given path, the higher the path priority will be in the second phase. This cost function ensures that the next selected path in the list will have the least negative effect on other routes.

Since SEGA handles variable-length segments, its cost function is accordingly a bit more complex. The cost of a path p is given by the four-term sum

$$\mathrm{C}(p) = \mathrm{C}_\alpha(p) + \mathrm{C}_\beta(p) + \mathrm{C}_c(p) + \mathrm{C}_f(p) \tag{2-2}$$

where $\mathrm{C}_f(p)$ is defined in Equation 2-1, $\mathrm{C}_\alpha(p)$ reflects the wastage of long segments by smaller wire segments, $\mathrm{C}_\beta(p)$ models the number of segments used, and $\mathrm{C}_c(p)$ expresses the number of alternative paths remaining for the considered coarse graph.

Boolean satisfiability Wood and Rutenbar [WR98] proposed to formulate the routing problem as boolean equations to solve it using Boolean Satisfiability (SAT) solvers. For a given global routing solution, the SAT solver expresses every net of the design as a chain of vertical and horizontal segments (corresponding to the FPGA routing channels). Boolean expressions then formulate the constraints of the design: the *connectivity constraints*, to model the connections needed to realize the connected path of a net, and the *exclusivity constraints*, which prevents nonexistent

paths (i.e. from the underlying architecture perspective) or already used channels from being considered. Exclusivity constraints need to be as exhaustive as possible for the FPGA architecture to be properly modeled. Nam, *et al.* [Nam+04] used the GRASP [MSS99] SAT solver on the system of connectivity and exclusivity equations of a given global routing to solve for the detailed routing. While the failure of iterative routing processes is not conclusive on whether the considered global routing is feasible or not, the SAT solver will always yield a valid detailed routing if a given global routing is routable on this architecture.

SAT solving has however a non-negligible drawback. A huge amount of resources are needed when dealing with very large designs or complex routing architectures, as the SAT solving problem is NP-hard. This is however a very dynamic research area which shows improvements every year as new SAT solvers such as BerkMin [GN07] or SCIP [Ach09] bring faster solving techniques.

4.4-2 Combined routing

Several FPGA routers combine the global and detailed routing steps into a single process in the FPGA design flow. This behavior mainly allows preventing the discrepancies which could appear when attempting to perform a detailed routing on a previously defined coarse graph, as explained in Section 4.4-1.

Greedy bin-packing Wu and Marek-Sadowska [WMS94] proposed one of the earliest combined routing algorithm, *greedy bin-packing* (GBP). GBP decays every multi-pin net of the design into 2-pin nets and creates a set of *track domains* from the routing segments of the considered FPGA architecture. Each track domain is a set of wire segments which are disjoint from the other track domains[2]. GDP then packs the 2-pin nets into the track domains following various packing heuristics such as the best-fit-decreasing (BFD) for the selection of the track domain. This results in a combined global/detailed routing process which performs better than contemporary detailed routers such as the previously described CGE and SEGA.

PathFinder *PathFinder* [ME95] is the most prominent routing algorithm used in modern FPGA routers. PathFinder is an iterative, negociation-based routing algorithm. It first tries to route every net on the graph, without concern about the over-utilization of routing resources. Once every net is routed, the algorithm operates an iterative process to rip-up and reroute the nets in order to keep only one net per routing wire at most. The innermost loop of PathFinder is in fact the maze routing algorithm, which has been enhanced to include a cost function to condition the propagation of the first stage of the algorithm, so as to limit the congestion and reduce the number of *bends* (i.e. routing a signal successively on perpendicular channels).

[2]The assumption of disjoint track domains depends on the FPGA architecture topology and is verified for the Xilinx 4000 chip used in [WMS94].

The original PathFinder algorithm presented by McMurchie *et al.* [ME95] uses the cost function

$$\text{Cost}(n) = [\text{b}(n) + \text{h}(n)] \times \text{p}(n) \tag{2-3}$$

to use a node n in a route, where $\text{b}(n)$ is the base cost of the node (often related to the intrinsic delay of the node), $\text{h}(n)$ is a factor related to the historic congestion of the node in previous iterations of the algorithm, and $\text{p}(n)$ is the current congestion of the node. The cost function has been refined over its implementations (e.g. VPR *(see next section)*, Madeo from Lagadec [Lag01]) to reflect the needs to take multiple parameters into account such as the delay model of the architecture or the cost associated with other choices of the current route (e.g. bends).

Versatile Place and Route To date, the most successful academic FPGA router is the Versatile Place and Route (VPR) [BR97] [Luu+11] tool. VPR uses a negotiation-based algorithm, based on PathFinder [BRM99], which consistently shows better results (i.e. fewer tracks) over the benchmarks ran on multiple FPGA routers from the time when VPR was first introduced. The cost function used in VPR is defined as

$$\text{Cost}(n) = \text{A}_{ij} \times \text{delay}(n, \text{topology}) + (1 - \text{A}_{ij})\,[\text{b}(n) + \text{h}(n)] \times \text{p}(n) \tag{2-4}$$

to route a path going from a source i to a sink j through a node n. The delay(n, topology) term is the Elmore delay model [Elm48] of node n, and A_{ij} is a balance factor between this delay term and the congestion term presented earlier. The most timing critical nets will have A_{ij} very close to 1 so that the net will have the priority over less timing critical paths in a congested segment.

4.5 Verilog to Routing

Subsequently to the presentation of the FPGA design flow steps in the previous subsections, it is important to note that the most competitive flows are those shipped with the design suites of commercial FPGA chips, whose development is funded by the company, driven by their customers needs, and that perfectly fit their chips. These design suite inner workings are obscure to the final user, since most of the added value can be considered trade secret.

From the available open-source tools, the *Verilog to Routing* [Ros+12] (VTR) framework proposed by Rose *et al.* is to date the most comprehensive academic flow available. It is often desirable for the architecture designer to explore architecture options for research or prior to the fabrication of a custom FPGA. Thanks to a user-defined architecture description, Verilog-To-Routing (VTR) is able to synthesize, place and route a Verilog design onto a virtual architecture.

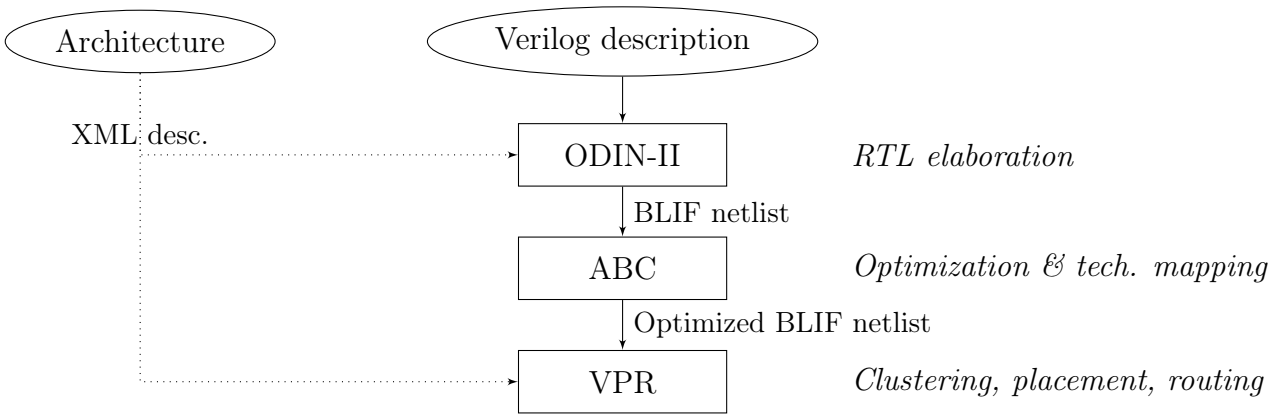


Figure 2-5 – **The Verilog to Routing design flow**

4.5-1 VTR design flow

The VTR framework flow is illustrated in Figure 2-5. The three main tools used to realize the RTL elaboration of Verilog files, the optimization of the logic netlist and eventually the placement and routing of the synthesized design are accompanied by multiple helper scripts and a defined framework for running benchmarks.

ODIN-II Jamieson *et al.* [Jam+10] proposed the ODIN-II tool as an open-source front-end synthesis framework for the elaboration of Verilog description modules. Although not the only open-source Verilog synthesizer, the other existing tools focus on the simulation of digital modules, rather than the elaboration and the integration in a whole tool-flow. ODIN-II integrates in the VTR framework by realizing the elaboration of the input Verilog design and generating a coarse netlist in the Berkeley Logic Interchange Format (BLIF).

The BLIF file not only contains the equations resulting from the logic synthesis of the Verilog design, but also *black-boxes* referring to dedicated hardware accelerators of the target FPGA architecture. For this purpose ODIN-II relies on the architecture description, specified in an XML file, which contains numerous architecture details used by VPR and precisely specifies these hardware accelerators in terms of inputs, outputs and possible configuration (e.g. a single 64kB memory block could be configured as a single dual-port 64kB RAM, or two single-port 32kB RAMs). ODIN-II uses this data to recognize modules instantiated in the main Verilog design by the same name and renders them in the coarse BLIF netlist as black-boxes. These black-boxes do not carry any logic information, but rather indications on how the rest of the system interacts and interconnects to them.

ABC ABC is a complete logic manipulation and verification system proposed by Mishchenko *et al.* [Mis+07]. In the context of the VTR framework, ABC is used to perform the hardware-independent optimization of the logic design and the technology mapping. The BLIF file generated by ODIN-II is first transformed into

an And-Inverter Graph (AIG) and processed by series of *rewrites* and *refactors* which performs technology-independent optimizations of the graph.

The logic network is then mapped to the target FPGA LUT size. The LUT size is not directly parsed from the architecture by ABC, but has been instead extracted by framework scripts. At this point, ABC has no information about the target architecture, apart from the LUT size. The resulting technology-mapped logic network is then iteratively cleaned up to remove unused inputs/outputs, constant-driven latches and orphan nodes. ABC outputs an optimized and processed BLIF file.

VPR As cited previously in this chapter, the VPR software is a customizable academic placer and router at the core of the VTR framework. VPR heavily relies on the custom architecture description given as an XML file, which specifies the details of the target FPGA architecture. The packer included in VPR, AAPack, realizes the clustering of the optimized BLIF file into architecture-dependent structures (flip-flops, memories, multipliers). The placement step is based on a simulated annealing algorithm and is able to automatically determine the adequate logic fabric size for the specified input design, although this size can also be fixed by the architecture.

The combined global/detailed routing process, presented in Section 4.4-2, uses a PathFinder variant to perform either breadth-first, timing-driven or directed search routing. The router can either try to route the design with a specific channel width if it has been fixed when invoking the VTR framework, or proceed to determine the smallest possible channel width required to successfully route this design. In this case, multiple routing attempts will be made to determine this smallest channel width by dichotomy.

As a result, in the case of a successful placement and routing, VPR outputs various files describing the final state of the flow. A netlist file, described in XML, iteratively lays out the FPGA design in terms of architecture-dependent modules and hardware blocks. The content of the netlist file depends on the architecture definition file and on the packing step of the flow. Additionally, a placement file gives the final placement determined by VPR. It contains a list of each instantiated block along with their horizontal and vertical position on the logic fabric, and a *subblock identifier* when a physical block can contain more than one logical block (e.g. an I/O block can be made of multiple I/O pads, a logic cluster can contain multiple LUTs). Finally, a route file specifies the path taken by each routed net of the design.

4.5-2 Applications to commercial FPGAs

Although VTR was originally designed for academic purposes as an open-source synthesis framework for the community, some recent initiatives, such as Rapid-Smith [HEW13], were proposed to connect VTR to commercial Xilinx FPGA. The combination of the netlist, placement and routing files could be enough for VPR

to generate a bit-stream, but the architecture file lacks the definition of how each function, route and hardware block configuration would actually be implemented in the target chip. RapidSmith reads these files and maps them to Xilinx popular FPGA chips in order to generate a bit-stream using bit-stream manipulation tools provided in the Xilinx tool-flow.

Until recently, VTR did not provide any mean of actually generating an RTL model of a test architecture. Recent work by Kim and Anderson [KA15] fills this gap by enabling the generation of a synthesizable model of the evaluated FPGA architecture and to generate bit-streams for this architecture. The current limitations only consider the soft logic elements of the logic fabric, but it is a promising step towards the easier development of custom architectures.

5 Conclusion

This chapter presented state-of-the-art techniques to handle the runtime reconfiguration of FPGA devices. Task relocation schemes for existing FPGA architectures have been detailed and runtime routing methods have been presented. We have seen that these techniques are limited in functionality and performance due to their dependency to their target architectures. Eventually, the Computer-Aided Design (CAD) tools and algorithms used in FPGA development flows have been detailed.

Overall, research domains around FPGA partial-reconfiguration and the associated CAD tools and algorithms are very active areas. Some of the presented work have been proposed at the early beginnings of FPGAs, and would be nowadays much more difficult to implement considering the fast pace of evolution of FPGA architectures. Modern reconfigurable devices include hundreds of complex hard blocks which render the dynamic runtime placement techniques harder to implement. Handling the device heterogeneity is a prime issue for modern relocation schemes which has only been solved to this day by only considering homogeneous hardware modules. This limitation does not allow a reconfigurable architecture to fully expose its performance in the context of partial reconfiguration if relocatable hardware tasks are considered.

To circumvent these issues, the contributions of this thesis bring enhancements at multiple levels of the FPGA device. An alternate representation of the configuration bit-stream is proposed to allow for the generation of relocatable, position independent hardware tasks, the *Virtual Bit-Streams*. These tasks are finalized at runtime for their final position by an online controller. This techniques do not rely on a particular FPGA architecture beyond the use of a dedicated controller, and could be applied to any reconfigurable structure. Additionally, modifications to the FPGA interconnect are explored to enable the placement of heterogeneous hardware tasks with more flexibility. This is achieved by decoupling the interconnection network in two parts, of which one is dedicated to the hard blocks only. Eventually, enhancements at the configuration memory level allow for the flexible placement of hardware

task configuration bit-streams into a memory organized as a shift-register.

Part II

Contributions

Chapter THREE

POSITION-INDEPENDENT TASKS: VIRTUAL BIT-STREAMS

Contents

1	Motivation for position-independent bit-streams	**56**
2	Virtual Bit-Stream concept	**57**
2.1	Interconnect abstraction	57
2.2	Route modeling	61
2.3	Coding the Virtual Bit-Stream	61
3	Clustering	**66**
4	Virtual Bit-Stream generation tools	**67**
4.1	Design flow overview	67
4.2	*vbsgen*, the Virtual Bit-Stream generation back-end	72
5	A Virtual Bit-Stream powered architecture	**77**
5.1	Reconfiguration controller	77
6	Limitations	**79**

Abstract

The contemporary methods of exchanging Intellectual Propertiess (IPs) and hardware tasks rely either on high-level RTL views to be integrated in the manufacturer's development flow, or finalized bit-streams tied to a specific chip architecture. In the context of online task relocation in a reconfigurable device with current FPGA design flows, this inevitably leads to one configuration bit-stream being created for each target position in the logic fabric. This chapter proposes another point of view of configuration bit-streams in the forms of unfinalized bit-streams, abstracted from the target interconnect resources, which can be loaded at runtime to attain a maximum flexibility in the final placement of the hardware task and an efficient compression of the configuration data stream [HCS15].

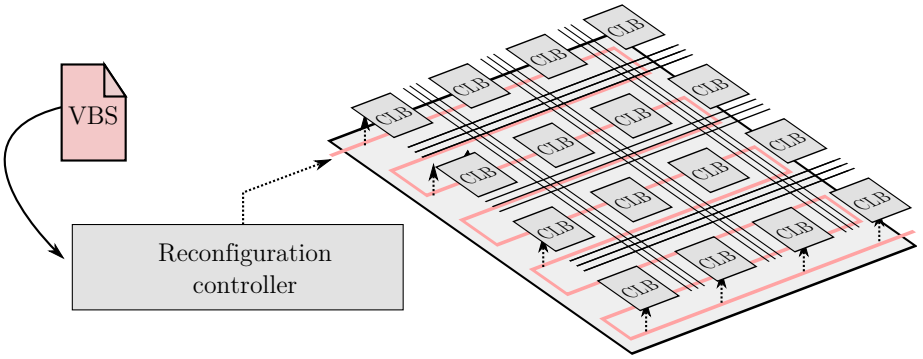


Figure 3-1 – **Overview of the Virtual Bit-Stream technique integrated to a FPGA logic fabric.** *The Virtual Bit-Streams are decoded by a dedicated reconfiguration controller which decodes the alternate representation and loads it into the target logic fabric configuration memory.*

1 Motivation for position-independent bit-streams

Although modern FPGA devices generally offer some advanced means of dynamically reconfiguring the logic fabric and interconnect at runtime, they often have the pitfall of limited software support for this reconfiguration. Chapter 2 gave an overview of state-of-the-art techniques to tackle the problem of relocation in these devices, mostly based on runtime bit-stream rewriting techniques or differential bit-stream storage, but never observing the problem from the angle of the configuration stream format by itself.

The manufacturer's relocation flow limits the relocation to the generation of multiple bit-streams for each possible position of the considered design on the final logic fabric. This is easily understandable given the heterogeneity of both the logic fabric and its accompanying routing network, within the FPGA chip itself and also between series of a given architecture. The direct consequences of this relocation scheme have been detailed in Section 2.1 of Chapter 2: as the number of desired placements of the hardware task increase, so does the volume of information to store in memory. This leads to little control on the final placement, as the number of position needs to be restricted, and memory wastage when considering fully-flexible placement.

The proposed *Virtual Bit-Stream* concept and associated flow aims to bring software paradigm to the reconfigurable hardware world. In software, a large RAM generally holds both executable code and data. For computers, the versatility of the platform generally implies that all the possible executable code can not be stored at once in the RAM, they are rather stored in large-scale storage medium such as hard disk drive (HDD) or solid-state drive (SSD). On-demand, at the user request, executable code can be loaded into the memory and executed by a general-purpose processor from there. The whole process is handled by the operating system kernel, which selects a suitable empty space of the physical memory to be allocated for the program execution, and may also load executable dependencies to be executed,

etc. Current trends in the software world also rely on the generation of intermediate byte-codes to achieve the portability of software code to multiple platforms. At runtime, a JIT compiler creates the system-specific binary code from the distributed intermediate byte-code. There is no such mechanism in the FPGA world which can act as dynamically as a software platform, mostly due to the inherent limitations of hardware design and the actual little interest in relocatable hardware for most applications. The Virtual Bit-Stream aims to be part of the answer to dynamic hardware task management, proposing an abstract representation of a configuration bit-stream which can be finalized at runtime to virtually place a task anywhere on a compatible logic fabric.

The Virtual Bit-Streams are decoded at runtime by a dedicated reconfiguration controller, as illustrated in Figure 3-1. This runtime controller handles the dynamic loading of the Virtual Bit-Streams and decodes them to create an equivalent conventional bit-stream specific to a location in the logic fabric. The resulting bit-stream is then configured into the target FPGA configuration memory.

2 Virtual Bit-Stream concept

The main problem associated with relocating a given configuration bit-stream is associated with finding a place somewhere on the logic fabric which completely replicates the original intended place of the bit-stream, in terms of logic fabric composition and interconnection network capabilities. Apart from the architecture point-of-view of the FPGA device, the configuration bit-stream also depends of the intrinsic method of inputting data within the configuration memory of the chip. The complexity and heterogeneity of modern devices leaves very little opportunity to find two places in a device sharing this whole set of parameters exactly.

The Virtual Bit-Stream (VBS) overcomes this situation with an intermediate, unfinalized, representation of a synthesized hardware task. It relies on pre-generated routing data to model a configuration bit-stream in an abstract manner, completely disregarding the FPGA device underlying routing architecture and configuration bit-stream organization. This abstraction allows to only finalize the configuration on-demand, at runtime and in real-time for the desired position. The runtime considerations of the Virtual Bit-Stream are thoroughly discussed in Chapter 4.

2.1 Interconnect abstraction

The Virtual Bit-Stream abstraction model can theoretically be applied to any reconfigurable architecture which can be reduced to a tiled fabric of configurable operators and its immediate surrounding reconfigurable network. For the sake of simplifying examples, the considered device is a highly flexible generalization of modern routing networks seen in island-style architectures, as shown in Figure 3-2.

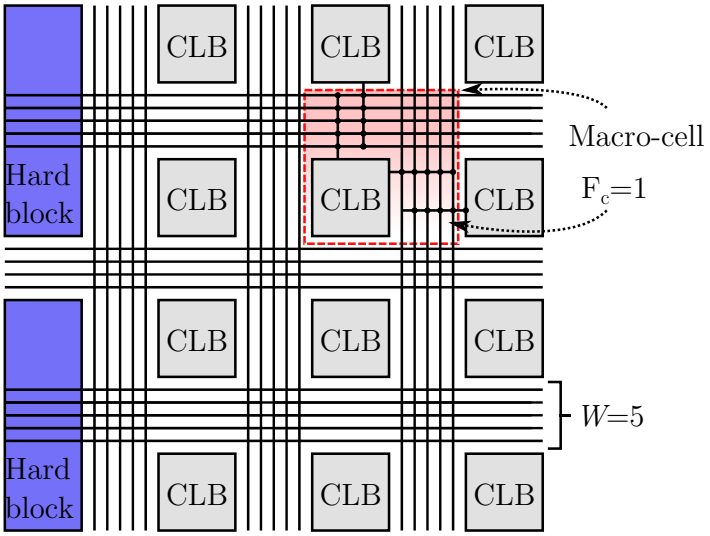


Figure 3-2 – **Example of an island-style architecture and its repeatable *macro-cell*.** *The considered architecture is a generalized island-style logic array and its mesh routing network. Ignoring local heterogeneity, a repeatable pattern can be identified, duplicated in both directions, which is what we consider in this document as a* macro-cell.

The repeatable pattern of the architecture presented in Figure 3-2 is surrounded with a red dotted line. It is composed of one element of the logic fabric, which can either be a logic block or a fraction of a hard block, associated to the immediate horizontal and vertical connection blocks and the switch block at their intersection. This pattern is defined as a *macro-cell*: the basic tile of the FPGA architecture which can be replicated in both horizontal and vertical directions to create the logic fabric. Most architectures will have multiple different macro-cells, depending on the logic fabric constitution. From the Virtual Bit-Stream point-of-view, however, the exact content of the macro-cell does not matter as the online decoder will be specific to each macro-cell.

Figure 3-3 depicts an example of such a macro-cell in which the interconnect is made of $W = 5$ wires in each channel, with a connection block flexibility $F_c = 1$ (i.e. each logic block input/output is connected to all tracks in the routing channels), and a switch block flexibility $F_s = 3$ using a disjoint switch topology. In this example we consider that all the configurable switches are 4- or 3-way switches able to interconnect any of the 4 or 3 wires on its endpoints. For reference, a red (dark) dot in Figure 3-3 corresponds to a 4-way configurable switch whose circuit implementation requires 6 pass-transistors. Similarly, the orange (bright) dots are 3-way configurable switches and require 3 pass-transistors, along with their configuration memory cells. In addition to this interconnect, we define the Basic Logic Element (BLE) in this example as a single 6-LUT plus a configurable output flip-flop. This logic block thus requires 6 inputs and one output and we define $L = 7$ as the number of logic inputs and outputs in the macro-cell.

We can describe the configuration bit-stream BS of this macro-cell as the union

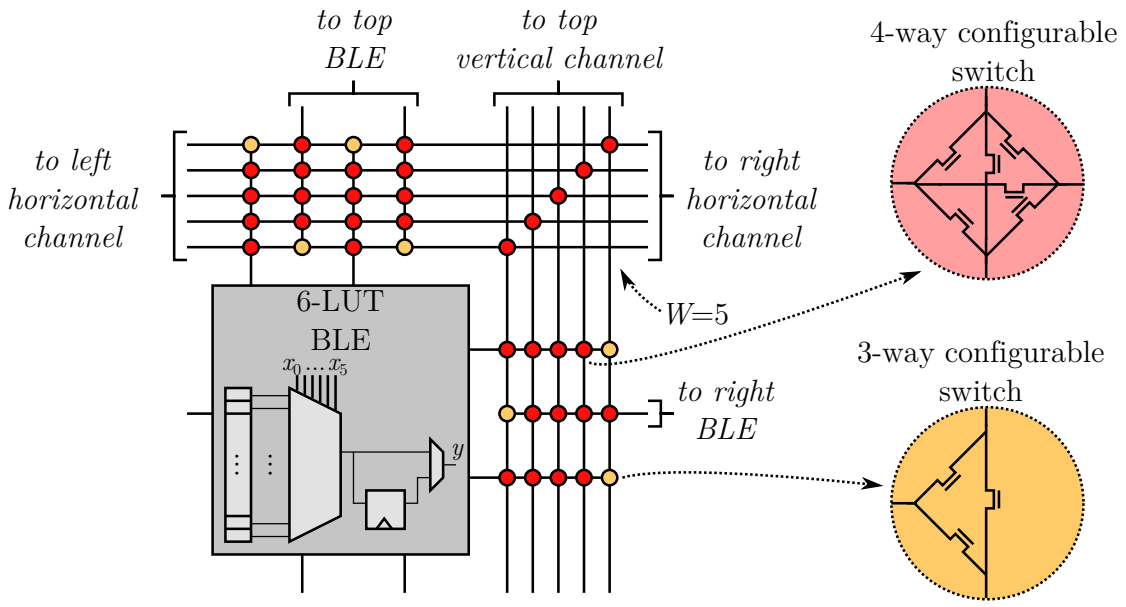


Figure 3-3 – **Details of a macro-cell and its routing details.** *The local interconnect, comprising the switch block and the connection blocks, along with a basic logic element (BLE), forms the macro-cell. This macro-cell uses 4- and 3- way configurable switches as its configurable switches in the connection blocks. The interconnection network contains connections going from/to the local logic element as well as connections to neighboring elements. The logic element itself has some of its connections feeding adjacent macro-cell connection blocks.*

between, in one hand, the immutable logic configuration BS_{logic} of the BLE and, on the other hand, the configuration of the interconnect part of the macro-cell $BS_{interconnect}$, made of the state of every pass-transistor of each configurable switch of the macro-cell routing. Thereby, we define $BS = BS_{logic} \cup BS_{interconnect}$. The size (i.e. the number of bits) of this configuration bit-stream is Nb, also defined as the sum of the sizes Nb_{logic} and $Nb_{interconnect}$ of both of its components. These two sizes depends on the architecture and can be calculated. The logic part contains here the 2^k bits of configuration of the k-LUT and one bit to configure the output multiplexer to select between the latched and asynchronous output of the LUT, leading to

$$Nb_{logic} = 2^k + 1 \tag{3-1}$$

in the case of our defined logic block, i.e. $Nb_{logic} = 2^6 + 1 = 65$ bits. The size of the interconnect part of the configuration bit-stream can also be defined, in our case, based on the number N_{SW3} and N_{SW4} of 3- and 4-way switches. For the routing architecture used in Figure 3-3, the number of 3-way switches is defined by $N_{SW3} = L$, since there is one 3-way switch for each logic input or output. Comparably, the number of 4-way switches is calculated by $N_{SW4} = L(W-1) + W$, where $L(W-1)$ corresponds to the number of 4-way switches for each of the L logic inputs and outputs, and W is the number of 4-way switches in the disjoint switch block. If we put these calculations together, we define the number of configuration bits for the

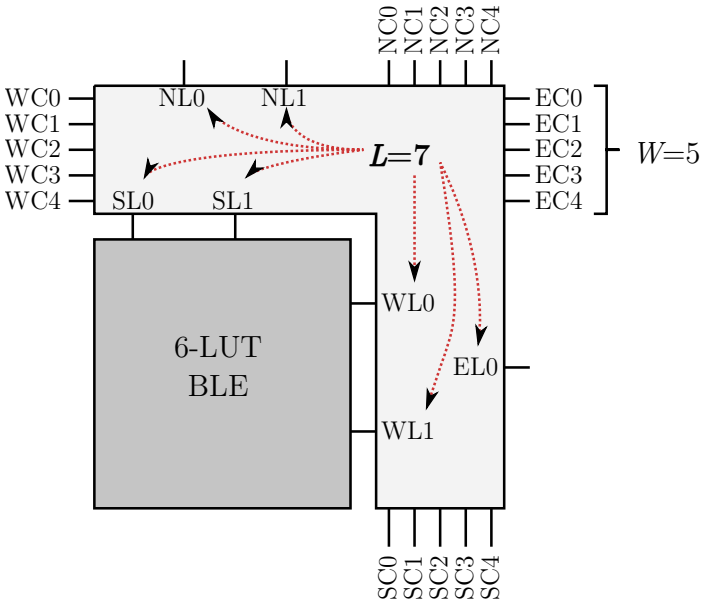


Figure 3-4 – **Abstract model of a macro-cell.** *The abstraction of a macro-cell is made on its routing portion. In this representation, the inner details of the detailed routing are ignored and the interconnect is rather seen as a black box with input and output pins corresponding to the routing lines and the connection boxes. The model of a macro-cell with $W = 5$ wires per channel and $L = 7$ connections to nearby logic blocks has $4W + L$ pins through which the route paths can cross.*

interconnect portion of the macro-cell as

$$\begin{aligned} Nb_{interconnect} &= 6 \times N_{SW4} + 3 \times N_{SW3} \\ &= 6 \times (L(W-1) + W) + 3 \times L \end{aligned} \tag{3-2}$$

solving in this example to $Nb_{interconnect} = 6 \times (7 \times 4 + 5) + 3 \times 7 = 219$ bits. The sum of Equation 3-1 and Equation 3-2 results in $Nb = N_{logic} + N_{interconnect} = 65 + 219 = 284$ bits in total for the set of configuration bits BS of a single macro-cell.

These 284 bits of configuration bit-stream defines the functionality and interconnections of the macro-cell, notwithstanding that the considered macro-cell is fully used (a lot of routes and the logic block are used), partially used (only a few routes going through the routing channels) or even not used at all in the design. The Virtual Bit-Stream abstraction model arise from the observation that it is possible to express the same amount of information with fewer data using a smarter notation than the conventional, raw, bit-stream BS.

Figure 3-4 illustrates the Virtual Bit-Stream model corresponding to the physical macro-cell presented in Figure 3-3. The model is parameterized by the channel width W, the number of logic inputs and outputs L and the type of logic block, which dictates the number Nb_{logic} of configuration bits of the logic block. This abstract model completely removes any insight of the underlying routing architecture and the exact routing possibilities of this network. From the Virtual Bit-Stream point-of-view, the routing portion of the macro-cell comprises a set of inputs and outputs

spread among purely *routing* I/Os which connects to other wire tracks, and *logic* I/Os which are tied to an adjacent logic block. Since the routing details are omitted in the model, there is no knowledge of whether the logic inputs and outputs connect to the logic block of the considered macro-cell, or to a logic block bordering it on the logic fabric.

2.2 Route modeling

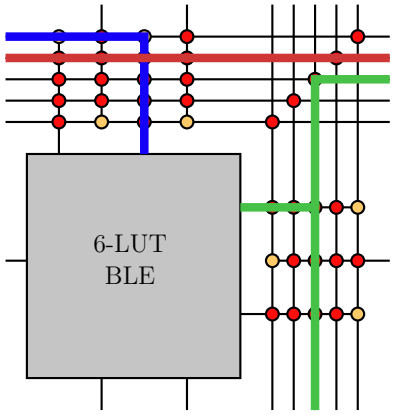
As seen in Figure 3-4, the inputs and outputs are named following a hierarchical nomenclature. From the strict Virtual Bit-Stream binary file perspective, they are sequentially numbered from 0 to $4 \times W + L - 1$ and denoted as such. However, from the algorithm point-of-view it is necessary to have the knowledge of the routing architecture details, which makes desirable to label the abstract model inputs and outputs with evocative names.

The retained model for our test architecture is based on multiple criterion enabling to uniquely identify each input and output. First, the I/Os are classified in function of their cardinal position on the routing portion of the macro-cell (i.e. *North*, *East*, *South* and *West*). This cardinal indicator disregards the physical organization of the routing fabric. Each I/O is then organized in function of its functional affinity: either *Logic*, for the connection to logic blocks, or *Connection* for inputs and outputs going to and from other routing tracks. Eventually, the I/Os are sequentially numbered within each couple of cardinal position and functional affinity so that each of them is uniquely identified by a tuple (D, T, n), where D is one of the cardinal direction $\{N, E, S, W\}$ (*North*, *East*, *South* or *West*), T is the connection type in $\{L, C\}$ (*Logic* or *Connection*) and n is a discriminating number.

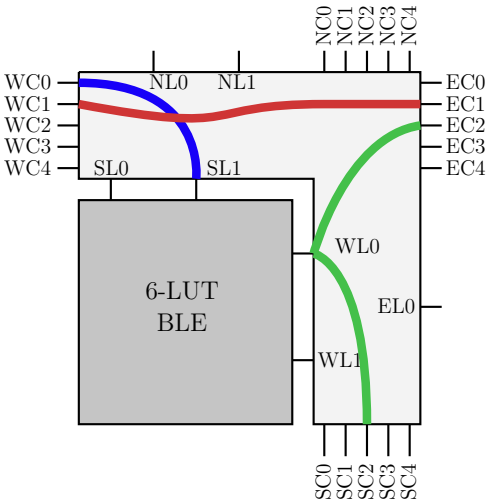
In spite of the lack of knowledge on the underlying routing architecture, the Virtual Bit-Stream stores the routing data of the abstract macro-cell model as a list of connections between inputs and outputs of the said macro-cell. As an example, Figure 3-5a illustrates a possible implementation of a macro-cell containing 3 routes, and its corresponding abstract model in Figure 3-5b. Using the aforementioned nomenclature, a list of abstract connection is built and constitutes the bulk of the Virtual Bit-Stream data which will be decoded at runtime to reconstruct a compatible configuration bit-stream dependent of the final task location. In Figure 3-5c, the abstract routes are listed as $WC1 \leftrightarrow EC1$, $WC0 \leftrightarrow SL1$ and $WL0 \leftrightarrow EC2 \leftrightarrow SC2$. The latter connection is not point- to-point and thus needs to be translated to two connections $WL0 \leftrightarrow EC2$ and $WL0 \leftrightarrow SC2$ in the connection list, as the decoding algorithms does not currently support multi-point routes.

2.3 Coding the Virtual Bit-Stream

Given the primary goal of the Virtual Bit-Stream, which is to provide a mean to describe *relocatable* bit-streams, we have defined in the previous section a model and a nomenclature to express the routing of a hardware task as an abstract list of



(a) **Detailed routing in a macro-cell**



(b) **Corresponding Virtual Bit-Stream abstraction**

From	To
$WC1$	$EC1$
$WC0$	$SL1$
$WL0$	$EC2$
$SC2$	$WL0$

(c) **Resulting connection list**

Figure 3-5 – **Virtual Bit-Stream abstraction, from routing to connection list.** *The Virtual Bit-Stream abstracts the details of the interconnection network contained in a macro-cell. With this abstract representation and an adequate nomenclature, the routing portion of a macro-cell can be described as a list of point-to-point connections instead of the state of every pass-transistor in the interconnect.*

connections. Since the finality of this alternate representation of the task data is to be decoded at runtime to create a final bit-stream, there is a need to organize the macro-cell abstract data into a structured file format. This binary file (i.e the Virtual Bit-Stream file, as opposed to the Virtual Bit-Stream design flow) not only contains logic and routing but also additional metadata which informs the online decoder about the hardware task geometry and composition. In the end, this Virtual Bit-Stream file *is* the hardware task as it contains all the necessary information to place it on a logic fabric.

2.3-1 Metadata

The metadata required in a Virtual Bit-Stream file header include the task geometry (i.e. its width T_w and height T_h), expressed in number of macro-cells. Since most spatial schedulers work best with rectangles, we decided to limit the geometry

of hardware tasks to the minimum rectangle fitting all the required logic resources. The number of macro-cells N_{macro} described in the file is also present so that the decoder knows in advance how many data shall be read. Depending on the logic utilization of the hardware task, N_{macro} is *not* necessarily equal to $w \times h$, because of the limit put on the task shape. Regarding the coding of these three metadata, the number of bits to code each of them is only dependent on the architecture logic fabric width $Arch_w$ and height $Arch_h$. Indeed, the task width and height bit size Nb_S can be determined by

$$Nb_S = \lceil \log_2 (\max (Arch_h, Arch_w)) \rceil \tag{3-3}$$

since these fields can at most represent the largest length of the architecture size. Similarly, the maximum number of macro-cells described in the Virtual Bit-Stream file is equal to the total number of logic cells of the architecture, which leads to calculate the number of bits Nb_M to represent the macro-cell count N_{macro} of the file as

$$Nb_M = \lceil \log_2 (Arch_w \times Arch_h) \rceil \tag{3-4}$$

Each of the abstract macro-cell are then iteratively described in the Virtual Bit-Stream to represent the task data with both the logic content and the abstracted list of routes.

Using an example logic fabric of $4,096$ logic blocks arranged in a 64×64 array, the size and macro count metadata of a task would be respectively coded on $Nb_S = 6$ bits and $Nb_M = 12$ bits.

2.3-2 Macro-cells

The macro-cells are sequentially described in the Virtual Bit-Stream file. The order itself does not matter but would rather depends on the target architecture, the configuration method, and the organization of the configuration memory. The arrangement of the list should ensure that the online Virtual Bit-Stream decoder is able to generate a valid configuration bit-stream without seeking forward and backward in the Virtual Bit-Stream.

Macro-cell metadata Each of the macro-cell contains metadata informing about its logical position in the task. The horizontal X and vertical Y indexes describe the current macro-cell position starting from a virtual $(X;Y) = (0;0)$ point on upper left corner of the task. Each additional macro-cell increments those values. The position is indeed required for proper decoding since empty macro-cells (i.e. those not containing logic data nor routes) *must* be represented in the final bit-stream anyway. The maximum horizontal and vertical positions directly depends on the maximum size of the logic fabric. Hence, these fields are coded on the same data size Nb_S as the task width and height, detailed in Equation 3-3.

Logic data The configuration content of the logic block directly comprised in the considered macro-cell is then coded as a raw block of Nb_{LB} bits. Depending on the logic fabric composition and organization, these bits may be anything from a k-LUT content to the configuration of an arithmetic accelerator or the initial configuration of a memory.

Routes As described in the preceding sections, the routes in the Virtual Bit-Stream are not tied to the FPGA interconnect network architecture. The routes are rather expressed as a list of connections between input/output pins of the abstract representation of the interconnect shown in Figure 3-4. The list is preceded by a route counter representing the number of routes N_{route} in this macro-cell. The maximum possible number of routes $\max(N_{route})$ in a macro-cell is less straightforward to express since the extreme cases are much less rarely used. Theoretically, the maximum number of routes in the route list of a macro-cell would be $4W + L - 1$, in the case where a single input net is split to all other pins of the macro-cell, resulting in $4W + L - 1$ two-pins nets. However, $\max(N_{route})$ was empirically defined as $2W$ over the various experiments, and this value is retained for the definition of the number of bits Nb_R to code the number of route:

$$Nb_R = \lceil \log_2 (2W) \rceil \tag{3-5}$$

The proper list of routes of the macro-cell is coded as a succession of couples (In, Out) of connections, each field of the couples being coded on Nb_C bits, where

$$Nb_C = \lceil \log_2 (4W + L) \rceil \tag{3-6}$$

is the number of bits required to code any of the pin of the macro-cell interconnect, numbered from 0 to $4W + L - 1$. This numbering ignores the nomenclature used by the algorithm, the mapping between the two representation is however the same for all similar macro-cells (i.e. sharing the same W and L parameters).

To ease the online decoding step, the routes are listed by order of congestion: the routes going through the most demanded switches (with regards to the possible paths in the macro-cell) will get coded first. This order puts less stress on the online decoder and prevents the need for re-routing already processed nets in a macro-cell.

Virtual Bit-Stream example Using the example illustrated in Figure 3-5, a corresponding Virtual Bit-Stream coding of the macro-cell would be defined as follows. The logic data of the macro-cell corresponds to a 6-LUT, which gives $Nb_{LB} = 2^6 = 64$ bits. The same architecture example as Section 2.3-1 is retained, which leads to $Nb_S = 6$ bits. The number of routes field and connection width field are calculated following Equations 3-5 and 3-6, which solves as $Nb_R = \lceil \log_2 (2 \times 5) \rceil = 4$ and $Nb_C = \lceil \log_2 (4 \times 5 + 7) \rceil = 5$. The equivalent macro-cell data is given in Table 3-1. Its total size amounts to 120 bits, in comparison to the 284 bits which would have been necessary otherwise with a conventional coding of every bit of the interconnection network.

Table 3-1 – **Example Virtual Bit-Stream data on a macro-cell with $W = 5$, $L = 7$ and four routes**

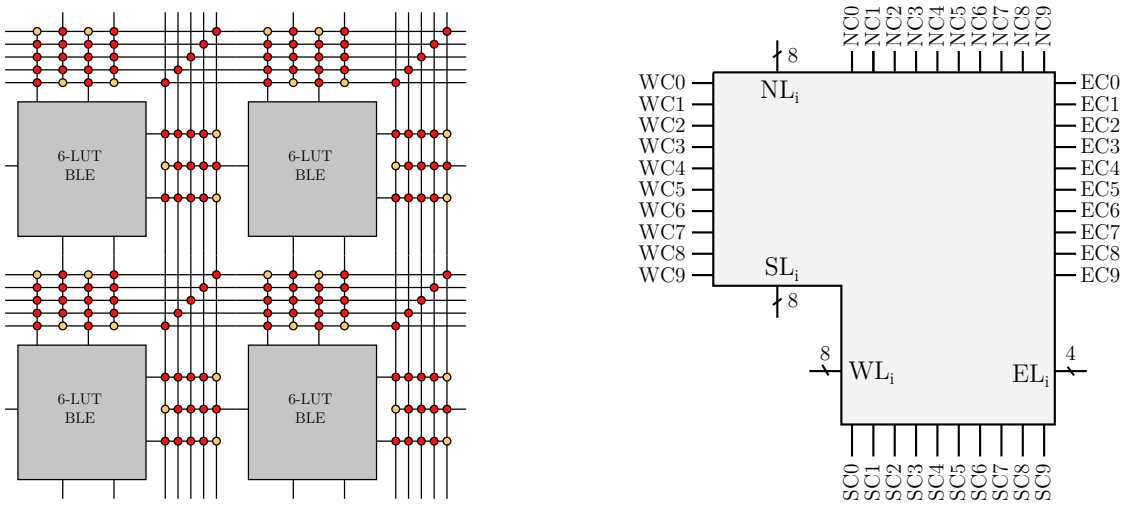
Field	Size	Data
Position x	Nb_S	001100
Position y	Nb_S	011001
Logic data	Nb_{LB}	1101...0010
Number of routes	Nb_R	0100
$WC1 \rightarrow EC1$	$2Nb_C$	00110 01001
$WC0 \rightarrow SL1$	$2Nb_C$	00101 10000
$WL0 \rightarrow EC2$	$2Nb_C$	10010 01010
$SC2 \rightarrow WL0$	$2Nb_C$	01101 10010

Heterogeneity considerations The Virtual Bit-Stream implementation has been considered over homogeneous architectures until now, with the macro-cells parameters being consistent over the logic fabric. It is however possible, and necessary for most architectures, to consider the heterogeneity of the FPGA, which implies to include another metadata field for each macro-cell giving an insight to the online controller on the *type* of the macro-cell. Each type T of macro-cell corresponds to a different tuple (W, L). Hence, each macro-cell type implies different coding Nb_R for the route count and Nb_C for the connections. The online controller must therefore provide a decoding method for each type T of macro-cell. The number of macro-cell types should thus be kept to a minimum if the architecture is designed from scratch. For the sake of simplicity, most explanations on the Virtual Bit-Stream format referring to the homogeneous architecture previously defined will disregard the heterogeneity considerations.

2.3-3 Overview

The Virtual Bit-Stream is built from a set of metadata and a list of macro-cell data whose format have been explored in the previous subsections. It should be noted that even if the Virtual Bit-Stream abstracts most details from the target architecture, some fields are still dependent on some parameters of this architecture. Notably, the coding format Nb_S, Nb_M, Nb_R and Nb_C of the Virtual Bit-Stream file fields have been defined as a function of the architecture and macro-cells parameters $Arch_W$, $Arch_H$, W and L. Nb_{LB}, the number of configuration bits of the logic elements, is also a function of the architecture logic fabric. A given VBS- compatible FPGA architecture and its associated Virtual Bit-Stream file format can thus be described by these parameters defining the fields data sizes. For easier reference to this format, we defined the tuple $SMRCL$ as the format of a Virtual Bit-Stream file, giving the data sizes of:

- the hardware task size (S),
- the number of macro-cells (M),



(a) **Four adjacent macro-cells and their surrounding interconnect**

(b) **Equivalent clustered abstraction of four macro-cells**

Figure 3-6 – **Clustering multiple macro-cells together before coding the Virtual Bit-Stream.** *Using the Virtual Bit-Stream technique, it is possible to use a coarser level of abstraction by grouping multiple macro-cells together prior to the calculation of the route list. The clustering offers the possibility to represent the routes with even fewer data, but at the cost of more complex decoding operations.*

- the number of routes in a macro-cell (R),
- the input/output pins of the routes (C),
- the logic block configuration (L/LB).

For heterogeneous architectures, the format would be defined by the tuple SM and a dictionary $T_i \Rightarrow (R_i, C_i, L_i)$ for each macro-cell type T_i.

3 Clustering

The model detailed in the previous section has only been applied at the finest possible grain of an architecture: a single macro-cell. It is however possible to group multiple macro-cells together to form a single cluster, as shown in Figure 3-6. We only consider regular, square clusters of macro-cells grouping M^2 macro-cells together, M being the size of the cluster expressed in number of macro-cells per square-side of the cluster.

The direct consequence of clustering multiple macro-cells together is the aggregation of their routing resources into one abstract cluster, which also means fewer routes to be connected and a smaller representation of a cluster in comparison to the description of the macro-cells separately. Indeed, from the Virtual Bit-Stream point of view, the cluster becomes the basic repeatable element to be coded. For this purpose, the channel inputs and outputs that were previously shared between

two macro-cells do not exist anymore at the cluster level and are considered part of the local interconnect. As such, the number of connections that are listed for a given cluster is smaller than the union of routes of each macro-cell of the cluster. Indeed, intra-macro-cell routes (i.e. those going from one logic I/O to another one inside the same macro-cell) will stay a single route in the cluster list of connections, but routes which previously crossed two macro-cells of the same cluster through channel wires are now fused together. The routes can thus be split into two categories at this point:

1. Routes which only spanned one macro-cell: these routes wont be affected much by the clustering since they will be coded almost verbatim in the abstract cluster, albeit with a higher bit width because of the higher number of input/output pins in the cluster.
2. Routes crossing multiple macro-cells: if the crossed macro-cells are now part of the same cluster, the number of connections required to describe the same net is greatly reduced and contributes to a reduction of the Virtual Bit-Stream size.

One consequence of grouping multiple macro-cells into a single cluster is the reduction of size of the overall Virtual Bit-Stream, for the previously evoked reasons. However, because of the increased space of solutions for the routing, the run-time routing algorithm complexity also increases along with the cluster size, since the complexity of the algorithms described in Chapter 4 is predominantly in $\mathcal{O}(N_{IO})$.

4 Virtual Bit-Stream generation tools

The presentation of the tools and algorithms associated with the development of applications targeting FPGA devices in Chapter 2 already stressed the importance of powerful tools to accompany the complexity growth of FPGA architectures. As such, the possibilities and improvements brought by the Virtual Bit-Stream techniques presented in Section 2 would be of little interest without a supporting design flow. While the previous section focused on the definition and high-level overview of the Virtual Bit-Stream principles and its binary file format, this section details the offline design flow transforming a hardware description into a usable hardware task.

4.1 Design flow overview

The design flow associated with the Virtual Bit-Stream technique aims to provide all the necessary tools allowing the designer to create a hardware task, in the form of a Virtual Bit-Stream file, from an application's hardware description. As such, the VTR tool flow, presented in Section 4.5 of Chapter 2, is one of the best academic candidate to perform most of the tasks required by the Virtual Bit-Stream design flow. VTR already includes the necessary tools and scripts to realize the RTL

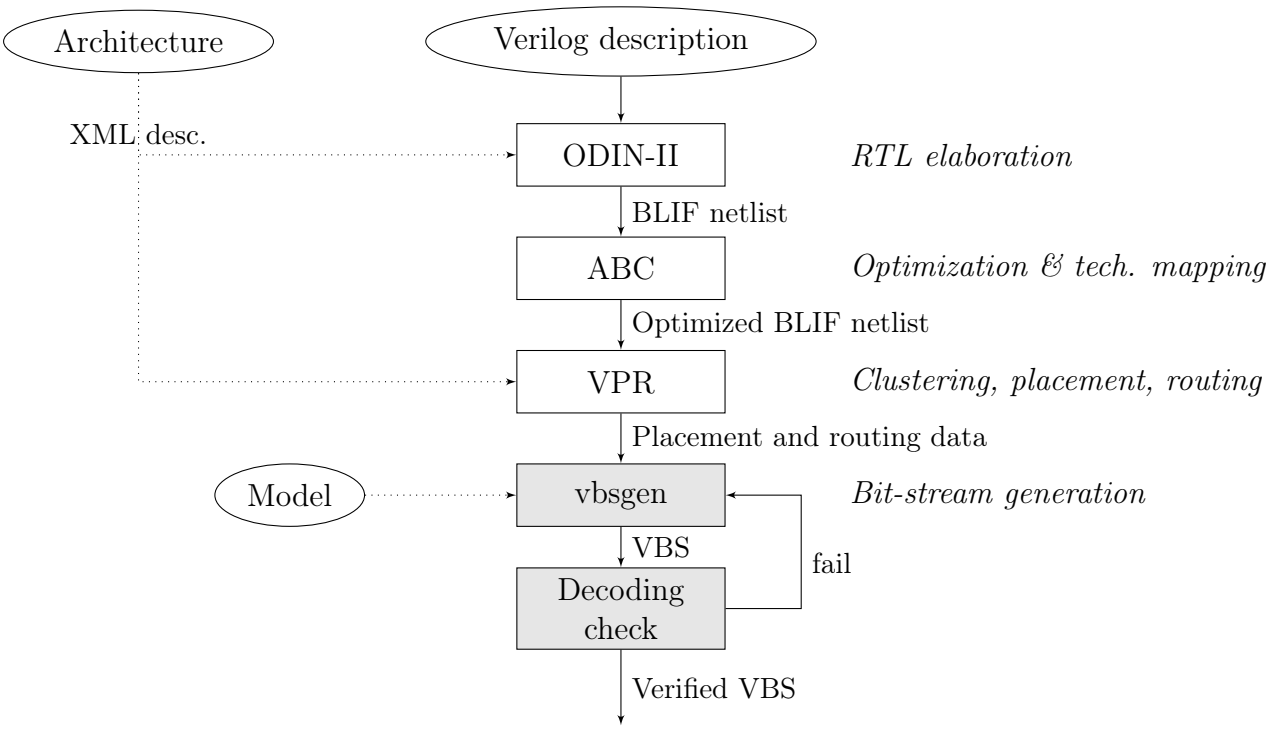


Figure 3-7 – **The Virtual Bit-Stream design flow, based on the Verilog-To-Routing framework.** *The Virtual Bit-Stream design flow relies on the Verilog-To-Routing framework to generate placement and routing data. The* vbsgen *back-end is subsequently used to generate custom bit-streams using a custom model. A decoding step is added after the bit-stream generation to ensure that the Virtual Bit-Streams created will be decodable at runtime, and to regenerate non-compliant bit-streams.*

elaboration of a Verilog description using ODIN-II, the technology mapping to the target architecture with ABC, and the clustering, placement and routing of the logic netlist thanks to VPR. The VTR design flow is completed by a custom back-end, *vbsgen* implementing the necessary algorithm to create both Virtual Bit-Stream files, and their equivalent conventional raw bit-streams for comparison.

The Virtual Bit-Stream flow details are presented in Figure 3-7. Most of the flow is based on the VTR framework up to the generation of placement and routing data, and our custom workflow is appended after the VPR step.

4.1-1 Outputs of the Verilog-To-Routing flow

For easier maintenance, our custom back-end is fed by the outputs of VPR instead of being integrated into VPR itself. The files obtained at the end of the VTR flow include files generated by VPR for one part, and the technology mapped logic netlist from ABC, which together allow *vbsgen* to build its internal model

```
1 .names x0 x1 x2 x3 S
2 ---- 1
3 -1-- 1
4 --1- 1
5 ---1 1
6 0000 0
```

Listing 3.1 – **BLIF representation of a 4-input OR logic gate**

The logic netlist The logic netlist, presented as a BLIF file, contains data about the various elements of the logic fabric and their relations. Most of the file is dedicated to the logic elements, whose intrinsic logic equations are represented by a set of *single output cover*. Single output covers describe the state of the inputs necessary to obtain either an *on-state* (a logic high) or an *off-state* (a logic low), in the same manner as a truth table. The only difference with complete truth tables is the inclusion of the *don't care* state for the input, indicating that the corresponding input can be either 0 or 1 to create the specified output.

The BLIF description of a logic gate presented in Listing 3.1 describes a four inputs ($x0$, $x1$, $x2$, $x3$) logic gate whose output name is S, describing an `OR` gate. The last single output cover giving the off-state when all inputs are 0 is added for clarity and is not needed as the other single-output covers already cover the whole range of possible inputs.

In addition to logic elements, hard blocks are also described as black-boxes: these special gates of the BLIF file only specify the instanciation of a block without giving any logic content. Latched logic elements are represented in the logic netlist as a logic gate (as previously described), and a corresponding *.latch* element with two parameters: the name of the logic gate output and the name of the latched output.

The placement file The placement file generated by VPR presents the result of the combined placement step of the tool. All elements of the logic fabric used by the task are listed in a formatted text file along with their horizontal and vertical position. In Listing 3.2, the netlist `bgm.net` is placed on a 95×95 logic array. For each element of the logic array, its name[1] is associated with its horizontal and vertical position x and y. The *subblk* column is used with elements of the logic array which can functionally be seen as an aggregation of multiple sub-elements. For example, in VPR, input/output blocks only span one element of the logic grid but can in fact host multiple input or outputs: each of these instances has a different *subblk* identifier. The tuple $(x, y, subblk)$ is thus unique for every functional element of the logic fabric in VPR.

[1]The block name is often the name of its output in VPR.

```
1 Netlist file: bgm.net Architecture file: eFPGA_k6_N4.xml
2 Array size: 95 x 95 logic blocks
3
4 #block name x y subblk  block number
5 #---------- - - -- ----  ------------
6 ...
7 n19217  70  84  0 #759
8 ...
```

Listing 3.2 – **A VPR-generated placement file**

The route file The route file is also expressed as a formatted text file after the last successful routing iteration of the VPR router. Every net of the design is described as a path traversing the various routing elements of the VPR logic fabric model. This model not only contains $CHANX$ and $CHANY$ elements corresponding to horizontal and vertical routing channels of the FPGA architecture, but also elements allowing to complete the parasitic capacitance model of a line, such as input and output pins of logic blocks. These additional elements $IPIN$ and $OPIN$ notably allow VPR to build a more accurate delay model for each net.

Listing 3.3 is an overview of the routing data associated with the net $n16458$ of an example design. Each portion of the path presents detailed information about:

- the node identifier in VPR internal routing graph, for debugging,
- the type of routing element traversed ($SOURCE$, $SINK$, $OPIN$, $IPIN$, $CHANY$, $CHANX$),
- the horizontal and vertical position of the routing element in the logic fabric,
- the additional identifier of the routing element:
  - the class for $SOURCE$ and $SINK$ elements, representing the type of the block (i.e. 0 for soft logic elements or a unique identifier > 0 for hard blocks),
  - the pin identifier for $OPIN$ and $IPIN$ elements,
  - the track number for $CHANX$ and $CHANY$ elements.

The VPR netlist This additional netlist offers a deeper insight on the VPR logic fabric model used to place and route a design. As this model is built from an architecture description in an XML format, the VPR netlist is read by the backend to obtain information on logic gates grouped in clusters, which is otherwise not specified in other output files. The netlist is a hierarchical XML document recursively presenting each feature of the architecture as *blocks*, with details about its inputs, outputs and more sub-blocks.

Additionally, the VPR netlist also specifies the *mode* used for each block. In-

```
1 Net 63 (n16458)
2
3 Node: 191009 SOURCE (73,22) Class: 2
4 Node: 191030 OPIN (73,22) Pin: 17
5 Node: 815487 CHANY (73,22) Track: 32
6 Node: 328497 CHANX (73,22) Track: 32
7 Node: 328462 CHANX (72,22) Track: 32
8 Node: 188429 IPIN (72,23) Pin: 2
9 Node: 188421 SINK (72,23) Class: 0
```

Listing 3.3 – **A VPR-generated route file**

deed, each element of the architecture can be used in multiple modes to mimic the configurable elements of modern FPGA architecture. For example, a 6-LUT could be modeled as either a single 6-LUT with 6 inputs, or two 5-LUTs and a multiplexer. A 16*kb* RAM could be defined to be used either as a single 16*kb* RAM, or two 8*kb* RAM, etc. The packing step of VPR then decides to assign each element of the logic netlist to any of the available modes of a block of the logic fabric. The mode detection is useful to be able to determine the configuration data associated with hard blocks, which is otherwise not specified since VPR has no insight on the configuration method of these blocks.

4.1-2 Virtual Bit-Stream generation

The proper generation of the Virtual Bit-Stream files makes use of the placement, routing and netlist files generated by the VTR flow. For that purpose, a bit-stream generation back-end, *vbsgen*, has been developed in Java to create both Virtual Bit-Stream files and conventional bit-streams, relying on a programmatic model of the target architecture. The architecture model specifies the logic fabric and interconnect composition of the FPGA architecture, with additional details such as the detailed method of configuration of the device, the organization of the configuration memory and the relationships between the elements of the logic fabric. The back-end tool itself is thoroughly described in Section 4.2.

4.1-3 Decoding check

As the Virtual Bit-Stream files are decoded at runtime by the online controller, there is no space for any decoding error that may occur after the bit-stream generation. To prevent any issue at runtime, the generated bit-streams are systematically checked offline by *vbsgen* using the same algorithm as the one used online.

Routers are, more often than not, dependent on the processing order of the nets getting routed. The maze routing algorithm and its derivatives, described in Section 4.4 of Chapter 2, frequently rip-up precedently routed nets to consider an

alternative, more straightforward, path for a new route. Indeed, this ensures that a locally-optimal global and detailed routing has been found, but it severely impedes the iterative routing process. The online decoder of Virtual Bit-Streams has a huge advantage over traditional routers in that it knows in advance that at least one valid routing exists for a given macro-cell route list. Using the routing found by VPR, the *vbsgen* back-end can thus orient the online decoding of the macro-cell using heuristics to sort the routes in the list, such as a congestion heuristic placing the most complex nets first in the list.

In spite of that, a failure of the online decoding algorithm may occur. In that case, the fail-safe decoding check allows to either trigger a new placement and routing step starting from VPR, using a new random seed for the router, or to enable the use of a raw coding of the interconnection data. In any case, a Virtual Bit-Stream file compliant with the online decoding process will always be generated, either at the cost of an additional design time, or with a smaller compression ratio.

4.2 *vbsgen*, the Virtual Bit-Stream generation back-end

Ultimately, using the output of the VTR flow, the *vbsgen* Virtual Bit-Stream generation back-end is able to gather the necessary information about the considered design and to generate Virtual Bit-Stream files as well as conventional bit-streams. For this purpose, *vbsgen* proposes both an application offering a complex command-line interface to interact with the designer, and a complete model framework to describe the target architecture. The architecture model used by *vbsgen* greatly differs from the VPR model in terms of goals and complexity. While the VPR model can specify complex logic architectures and explore non-conventional routing network while considering area and delay parameters, it was not meant as a bit-stream generation tool. The *vbsgen* model is programmatically built to form a collection of bulding blocks contained in a framework which can express dynamic features of an FPGA architecture, such as its configuration method, its interconnection with surrounding elements or the generation of hard blocks configuration data. This dynamic model allows for a greater control of the abstraction level of the logic fabric configuration data, which was indeed appreciated for the Virtual Bit-Stream technique.

The *vbsgen* tool relies on two main contributions from the designer:

- the output from the placement and routing tools,
- an architecture model describing the FPGA itself and the methods of expressing its configuration data.

The *vbsgen* dynamic architecture model comes in addition to the VPR XML architecture.

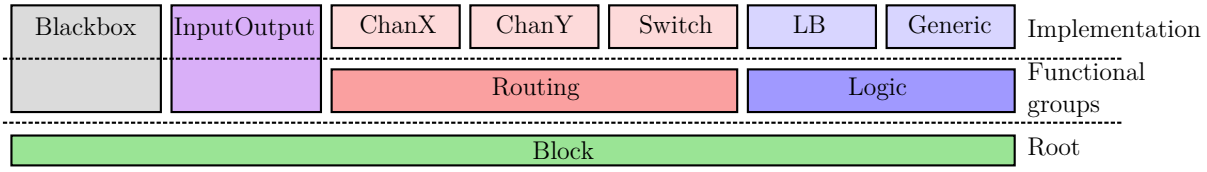


Figure 3-8 – **The *vbsgen* model framework.** *The framework is a collection of pre-defined blocks that may be directly used in a new architecture model. The root type* `Block` *is split in four functional groups, themselves split in more specific blocks. The framework includes the main soft logic and routing resources and their interconnection capabilities.*

4.2-1 Architecture model framework

The architecture model framework comprises basic building blocks required in every FPGA architecture model. It relies on a hierarchical class system which defines every block by its functional abilities. Figure 3-8 illustrates the framework hierarchy for the most used elements of an architecture. At the root of this hierarchy comes the generic denomination of every model element, the `Block` class, which generically characterizes the elements with a name and interconnection capabilities. A `Block` is the base class of any element of the *vbsgen* model array. The model array is a two-dimensional array which hosts all the elements (logic as well as routing channels) of the architecture.

The first level of specialization of the model is based on the blocks functionality, as shown in Figure 3-8.

Logic blocks The `Logic` functional group gathers the soft and hard blocks of an architecture which provide an actual added value in the datapath. The implementable blocks of the `Logic` group include:

- `LB`, a single BLE containing a LUT and a configurable output flip-flop,
- `CLB`, a complex cluster of logic block made of multiple BLEs and a configurable interconnect,
- `Generic`, a base class for any user-defined hard block such as arithmetic accelerators, memories, etc.

Interconnect The `Routing` functional group is dedicated to the blocks carrying signals from one point to another, i.e. interconnection blocks such as:

- `ChanX`, the horizontal routing channels and connection boxes,
- `ChanY`, the vertical routing channels and connection boxes,
- `Switch`, the interconnect switch boxes.

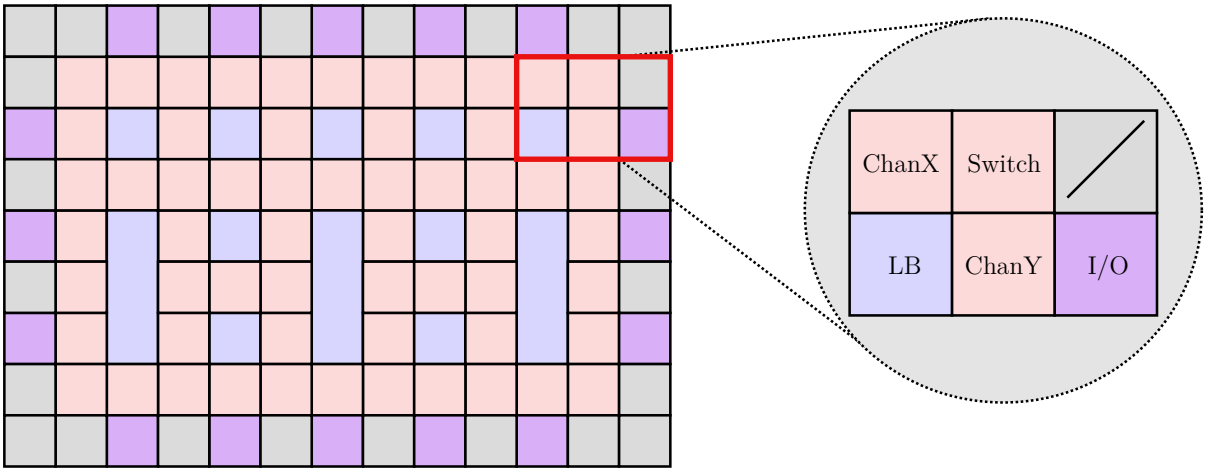


Figure 3-9 – **The *vbsgen* model array.** *The internal model of an architecture in* vbsgen *is a two dimensional array where each element represents one basic feature of the FPGA: soft logic, routing, input/outputs, etc. Each block relies on a specialization of classes of the model framework. Once built, the model array is populated using placement and routing data, and eventually dumped as a bit-stream.*

The interconnect blocks included in the model framework have adaptive channel width configured at the architecture level.

I/Os and black boxes The `InputOutput` and `BlackBox` functional groups fill the gap of very specialized elements of the model array which neither fit the `Routing` nor the `Logic` group. The former models the inputs and outputs of architectures while the latter is a placeholder for blocks whose functionality is either undefined or not useful for the model.

4.2-2 Model array

The logic fabric is the data model used internally by *vbsgen* to represent the architecture model. Up to this point in the Virtual Bit-Stream flow, an XML file lists every functional capabilities of the supporting architecture: logic block composition, heterogeneous resource distribution, input and output position, etc. The final step of the flow (i.e. the bit-stream generation) needs to map every generated data regarding the placement, routing and logic programming of every block to the architecture itself. For that purpose, the logic fabric is internally modeled as a two-dimensional array of blocks, as shown on Figure 3-9.

As presented in the previous section, the `Block` root component of the model hierarchy models the basic functionalities shared by every element of the model array. The root block model is pictured in Figure 3-10. It comprises four `IOInterface` modules, which are used to represent the interconnection between two blocks. These `IOInterface` modules are characterized by three parameters:

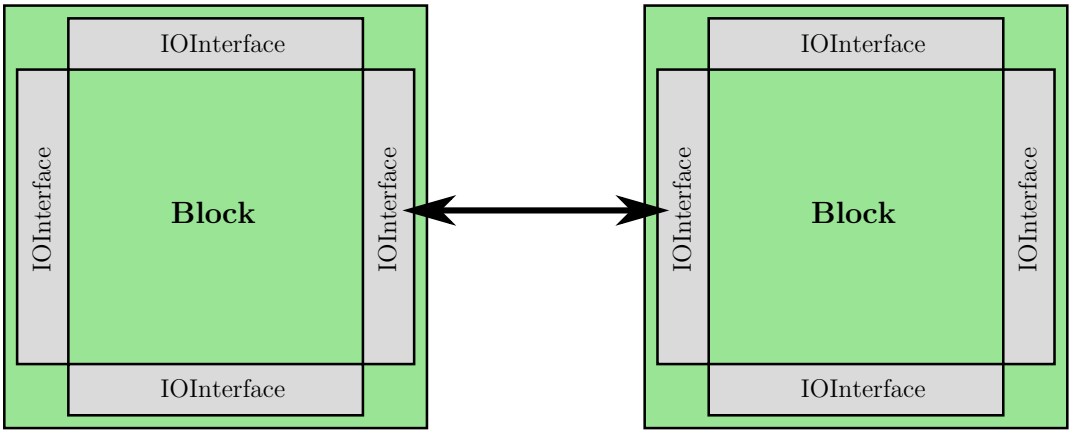


Figure 3-10 – **Block model used in the *vbsgen* framework.** *Each element of the model array derives from the root type* `Block` *of the* vbsgen *framework. A* `Block` *contains* `IOInterfaces` *on all of its four sides. These interfaces model the connection points between the block and its neighbors and serve to determine the relationships between elements of the model array during the bit-stream generation.*

- the interface width (the number of wires),
- the interface type (*interconnect*, *generic*),
- the orientation (*top*, *right*, *bottom*, *left*).

The interface type models the adaptivity of some interfaces. *Interconnect* wires will grow in size following the architecture channel width parameter, this is the case for routing-to-routing interconnections. On the contrary, *generic* wires will keep their size even if the architecture channel width varies, just like the interface between a soft or a hard block and its neighboring connection blocks.

An `IOInterface` can only be linked to another interface if their width and type matche, and if the orientations of the two interfaces are compatible: top with bottom and right with left (and *vice-versa*). These constraints ensure the structural integrity of the architecture logic fabric model and provide runtime verification of the architecture model since attempting to link two incompatible interfaces will throw an error.

Eventually, the two-dimensional model array contains a set of blocks which are interconnected together through their respective `IOInterface`. This mechanism allows to programmatically enforce most of the architecture constraints in terms of connectivity and blocks relationship at design time. Indeed, any deviation from the logic fabric specifications makes the *vbsgen* framework to generate an error.

4.2-3 Output file generation

The *vbsgen* framework supports a wide range of output bit-stream formats. The principal format in use is the Virtual Bit-Stream file format, leveraging the full relocation possibilities of a Virtual Bit-Stream powered architecture. Other formats are

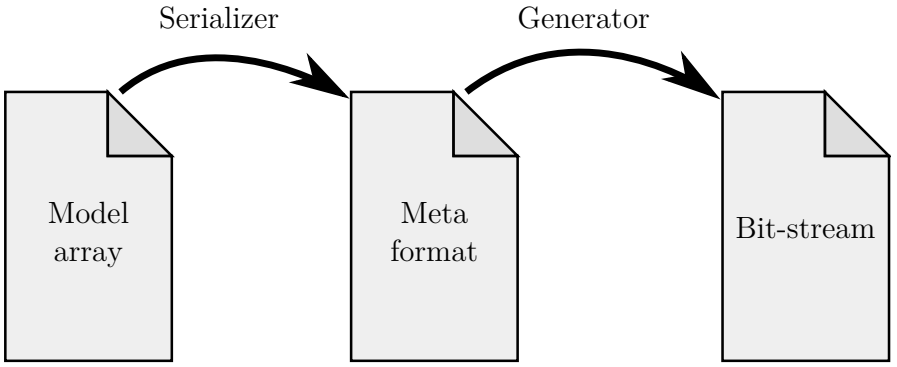


Figure 3-11 – **Generation of intermediate representations and final bit-stream in the *vbsgen* framework.** *From the designer perspective, the user-supplied* vbsgen *model fills a meta-structure dedicated to a specific output format (e.g. virtual or conventional bit-stream) instead of actually creating binary structures, using a user-defined* `Serializer`*. The conversion process from the meta-structure to the final bit-stream is ensured by a* `Generator` *included in the framework.*

supported, such as a conventional raw binary of the bit-stream, used for comparison with the Virtual Bit-Stream or in platforms without support for it.

In order to prevent the need for a user-defined architecture to define procedures that map the architecture data to the final output format, an intermediate meta format is used, as illustrated in Figure 3-11. The user only has to define a `Serializer` along with its architecture. The `Serializer` translates the architecture content into one of the framework-supplied meta format. The multiple `Generator` models that map the intermediary meta format data to the final output files are already defined within the *vbsgen* framework, which avoids the need for the model to know the binary format details of a Virtual Bit-Stream or a conventional raw bit-stream, for example.

Two meta-formats are in use in the framework: the first one represents the model data as a collection of abstract macro-cells for the Virtual Bit-Stream generation, as described theoretically in previous sections, while the second one relates to conventional raw bit-streams where the data is just represented without further manipulation other than compression schemes.

The `Generator` pass which transforms a meta-format structure into a usable bit-stream helps the generation of multiple representations of the data. For example, a Virtual Bit-Stream can either be dumped to the binary file format presented in this chapter, or into a text description for debugging, an XML representation for interoperability purposes, etc.

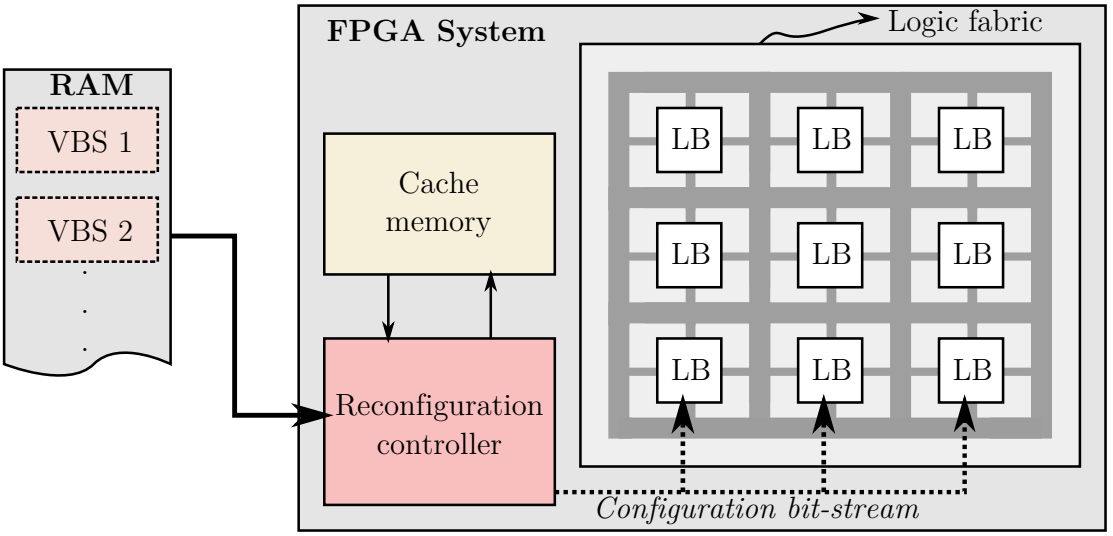


Figure 3-12 – **Integration of the Virtual Bit-Stream technique in an FPGA system.** *The support of Virtual Bit-Stream loading at runtime in a FPGA system is brought by an online controller realizing the online decoding of the Virtual Bit-Stream files in real time. The controller integrates a decoding pipeline allowing to read, decode and create a conventional bit-stream on the fly from the Virtual Bit-Stream for a given location in the logic fabric.*

5 A Virtual Bit-Stream powered architecture

The Virtual Bit-Stream abstraction scheme and binary format presented in Section 2 are only one part of the design flow allowing to seamlessly load hardware tasks at runtime on a compatible logic fabric. The generated Virtual Bit-Stream must be loaded into an external memory of the target architecture. This memory stores the hardware tasks to be loaded onto the logic fabric. An architecture using the Virtual Bit-Stream technique is illustrated in Figure 3-12, which introduces the reconfiguration controller.

5.1 Reconfiguration controller

The abstraction principles of the Virtual Bit-Stream rely on both a smarter, abstract coding of the routes of hardware tasks, and its conjunction with an online controller, dedicated to a specific architecture. This controller, the *reconfiguration controller*, is responsible for the Virtual Bit-Stream files decoding and the creation of a conventional bit-stream which will be eventually loaded onto the logic fabric.

Figure 3-13 schematizes the controller internals. In addition to the core logic, which handles the communication between the controller and the external world, three functional blocks are respectively responsible for the data pre-fetching, the virtual bit-stream decoding and the logic fabric configuration. The three steps are operating together in a pipeline, scheduled by the core logic, to sequentially recon-

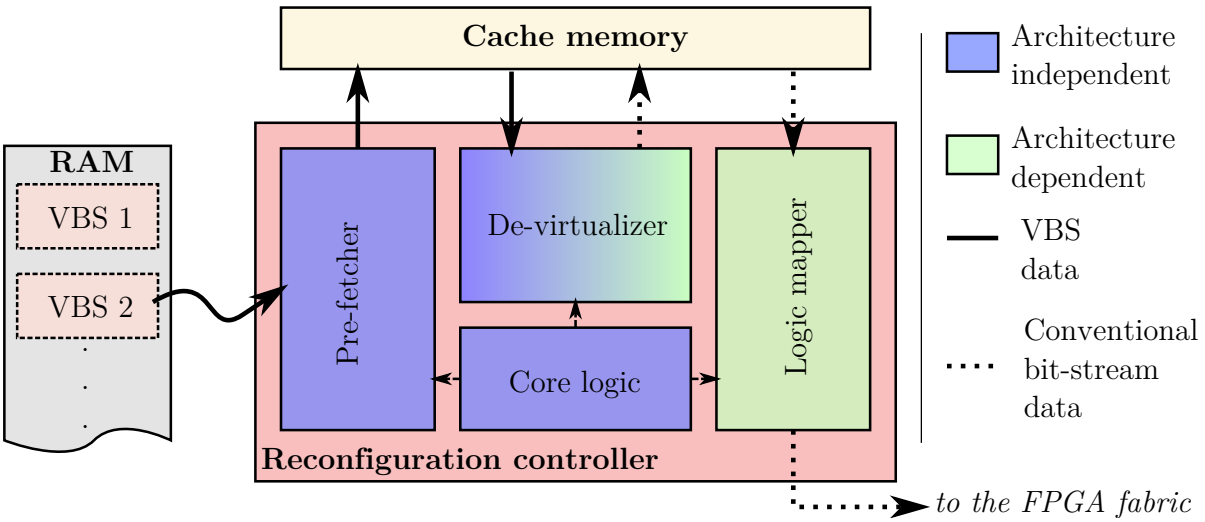


Figure 3-13 – **Overview of the Virtual Bit-Stream reconfiguration controller.** *Three main blocks operate in a pipeline to fetch and decode Virtual Bit-Streams, and to eventually load the resulting conventional raw bit-stream. The blue (dark) blocks are independent of the target architecture and can be directly integrated to any VBS-supporting system. Green (bright) blocks, however, are dependent of the FPGA architecture memory organization and routing network composition. The de-virtualizer block uses mostly generic algorithms to decode the Virtual Bit-Streams, but requires knowledge of the interconnection structure in macro-cells.*

struct the configuration data corresponding to a hardware task placed at a given position in the logic fabric.

5.1-1 Pre-fetcher

At the point of the Virtual Bit-Stream decoding, the computing-intensive tasks have already been executed offline with the placement and routing of the hardware task, and the creation of the Virtual Bit-Stream file. The first step realized by the controller upon request of loading a specific hardware task is to effectively load the task binary data from the memory containing the Virtual Bit-Stream files. This pre-fetch step allows the rest of the controller to access the task data and its associated metadata (the task width and height) directly from a local cache memory. Access to the metadata is required prior to effectively decode the Virtual Bit-Stream data to ensure a proper management of the available logic resources on logic fabric. Indeed, as the goal of the Virtual Bit-Stream technique is to facilitate the handling of hardware tasks, the controller manages the spatio-temporal scheduling of the logic fabric to allow for concurrent tasks to operate on the same fabric, although this aspect is not addressed in this document.

The pre-fetcher part of the controller is independent of the target architecture as it only interfaces the external memory and the internal cache. As such, the metadata provided to the core logic is only related to the generic information required to

predict the resource utilization of a task: its width and height.

5.1-2 De-virtualizer

The de-virtualizer acts as the Virtual Bit-Stream decoder, hence the name. The decoder iteratively reads the *virtual* configuration data of each macro-cell from the local cache memory, and a dedicated algorithm reconstructs the routing data from the list of connection of the macro-cell. This algorithm, at the core of the Virtual Bit-Stream technique, is more thoroughly detailed in Chapter 4, which explains the constraints and limitations of the decoder. The decoder is indeed the bottleneck of the controller, as the de-virtualizing pass must match the throughput of the bit-stream loading speed to be effective.

The reconstructed routing data, associated to the raw logic data already present as such in the Virtual Bit-Stream, forms the conventional configuration data of a macro-cell, which can be passed to the *logic mapper* to be configured on the logic fabric. Therefore, the de-virtualizer is slightly tied to the target architecture, as the routing data reconstruction routine needs to have a deep knowledge of the target interconnection network and its configuration data organization.

5.1-3 Logic mapper

The last step of a Virtual Bit-Stream loading, after its decoding, is to finally configure the configuration data at the required position in the fabric. Depending on the fabric configuration data memory organization and capabilities, the placement of a task may be limited by several factors. These limitations are inherent to the FPGA architecture and are thus taken into account by the core logic decision to place a task at a given position. The logic mapper holds the logic glue required to insert the decoded, conventional bit-stream in the logic fabric.

6 Limitations

The Virtual Bit-Stream technique presented in this chapter allows for the generation of unfinalized bit-streams whose routing data have been abstracted during the design process. These bit-streams are then finalized on-demand at runtime for a specific location of the logic fabric. The decoding (i.e. *de-virtualization*) process of the Virtual Bit-Stream files is not free and requires additional computing capabilities in the target architecture, in the form of a reconfiguration controller. The constraints associated with the online decoding process are more thoroughly explained in Chapter 4.

Although the Virtual Bit-Stream technique itself is general and could theoretically be applied to any target architecture, its benefits for online relocation are actually limited by the target architecture heterogeneity. As exposed in Chapter 2,

for a given task to be placed at a specific location of a logic fabric, the set of resources accessible at this location must be the same as the task content. The Virtual Bit-Stream does not leverage this constraint but nonetheless provide a way to describe a hardware task (in the form of a Virtual Bit-Stream file) independently of the target architecture routing details. Architecture-level considerations to enhance the relocation capabilities are however discussed in Chapter 5.

Chapter **FOUR**

RESULTS AND ONLINE DECODING OF THE VIRTUAL BIT-STREAM

Contents

1	Introduction	**82**
	1.1 Organization of the de-virtualizer	82
	1.2 Real-time considerations	83
2	Online decoding algorithms	**83**
	2.1 LUT-based decoder	84
	2.2 State-machine decoder	86
	2.3 Comparison	87
	2.4 Implementation results	88
3	Compression effect of the Virtual Bit-Stream	**89**
	3.1 Experimental methodology	90
	3.2 Results	90
	3.3 On the effect of clustering	92
	3.4 Fallback coding: to the limits of the Virtual Bit-Stream	93
4	Conclusion	**96**

Abstract

The principles of operation of the Virtual Bit-Stream techniques, detailed in the previous chapter, necessitate a careful integration into an online controller. This controller, mainly responsible for the decoding of the Virtual Bit-Streams into usable conventional bit-streams, has to work at a high throughput to be as transparent as possible from the bit-stream loading perspective. This chapter introduces two different hardware implementations of the decoding algorithm which suits the real-time needs of the controller. The algorithms are compared to a software implementation of a general purpose router. The effects of the Virtual Bit-Stream on the size of the bit-stream are also studied, and a study of the fallback coding of macro-cells is conducted to evaluate the limitations of this alternate bit-stream representation.

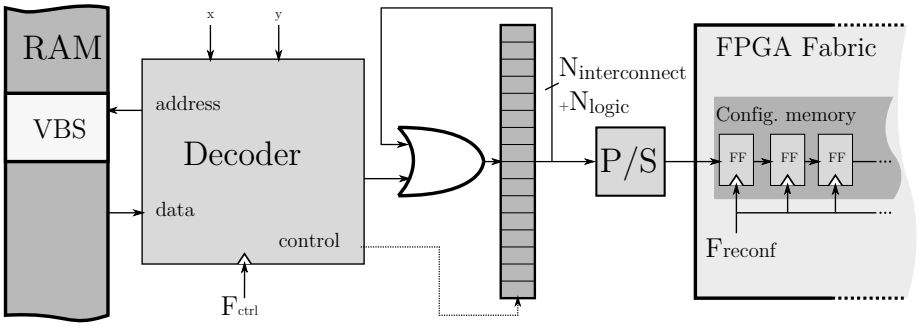


Figure 4-1 – **Architecture of the decoding part of the online controller.** *The decoder (de-virtualizer) is responsible for the decoding of the macro-cell data contained in the Virtual Bit-Stream files. Each macro-cell is decoded to a full conventional bit-stream containing $N_{interconnect}$ interconnect configuration bits and N_{logic} hard block or soft logic configuration bits.*

1 Introduction

The previous chapter introduced the Virtual Bit-Stream representation, allowing the description of a hardware task in the form of a position independent bit-stream, abstracted from the logic fabric interconnection structure details. This chapter emphasizes the mechanisms of the Virtual Bit-Stream architecture involved at run-time, after the generation of the bit-stream files using the proposed design flow. As the previous chapter focused on the methods of generating *easily decodable* bit-streams offline, this chapter concentrates on how to use the generated data in a real system, and what algorithms are used to achieve the decoding in reasonable time.

1.1 Organization of the de-virtualizer

The online controller goals are 1) to fetch Virtual Bit-Streams from their storage memory, 2) to decode the loaded Virtual Bit-Streams according to their final placements, and 3) to load the generated raw bit-streams onto the Field-Programmable Gate Array (FPGA) configuration memory. Steps 1 and 3 will not be discussed in this chapter which exclusively focuses on step 2. Step 1 only involves caching the Virtual Bit-Stream data which is not of interest for this work. Step 3 is closely related to the targeted logic fabric and there is no straightforward generalization. For example, considering commercial FPGAs, we do not have any insight on the configuration memory organization and on the link between this memory and the corresponding logic or routing element. However, Chapter 6 partially covers this step by proposing an alternative configuration memory facilitating runtime hardware reconfiguration.

Figure 4-1 illustrates the implementation of the decoding part of the controller. The Virtual Bit- Stream data is fetched from a memory and decoded according to the final location determined by its horizontal and vertical positions x and y. The output of the decoder is accumulated into a $Nb_{interconnect} + Nb_{logic}$ wide buffer which holds

the current macro-cell configuration. Once the macro-cell is successfully decoded, the logic mapper of the controller (represented as a parallel/serial converter P/S in Figure 4-1) serializes the macro-cell data and inserts it into the FPGA configuration memory at the F_{reconf} configuration clock frequency.

As already stated, the separation of the memory fetch, decoding, and configuration steps in the controller allows them to be pipelined. Indeed, during the insertion of a finalized macro-cell bit-stream, the controller is decoding the next macro-cell. Moreover, because of the Virtual Bit-Stream (VBS) data organization, multiple macro-cells can be decoded in parallel in order to increase the overall throughput of the controller and to fit the reconfigurable platform needs.

1.2 Real-time considerations

The bottleneck of the online decoding part of the controller resides in the algorithm computing the set $BS_m = \{S_{i,m}\}, i \in 0 \dots N_{interconnect} - 1$ of the state of the interconnect switches from the list $VBS_m = \{\dots, [IO_{in}(i,m), IO_{out}(i,m)], \dots\}$ of I/O connections of a given macro m. This algorithm must run as fast as possible, to provide the maximum possible flexibility for the complete platform, expressed as the time required to load a Virtual Bit-Stream, decode it and load it onto the logic fabric.

As a reference, the implementation of partial reconfiguration using the internal Internal Configuration Access Port (ICAP) configuration block on a Xilinx Virtex 7 is specified up to a frequency of $F_{reconf} = 100MHz$, with a throughput of $3200Mbps$ by loading bundles of 32 bits [Xilb]. The Altera Stratix V can similarly attain a maximum reconfiguration frequency F_{reconf} of 125MHz with a smaller throughput of $2000Mbps$ [Altb]. To be competitive, and theoretically implementable on Xilinx or Altera FPGA devices, the proposed Virtual Bit- Stream technique must thus be able to offer such high throughput of conventional bit-stream generation, to be as transparent as possible with regards to a traditional bit-stream configuration scheme. In addition, the controller ought to keep the lowest possible footprint, in terms of memory and logic area.

2 Online decoding algorithms

The run-time Virtual Bit-Stream decoding step is executed in the controller of Figure 4-1, which is placed alongside the FPGA fabric to generate conventional configuration bit-streams in real time. The global and detailed routing of the hardware task nets have already been computed offline by Versatile Place-and-Route (VPR) and *vbsgen*. Moreover, the way-points of the routed nets within each macro-cell are known and correspond to the inputs and outputs of the macro-cell connection lists. The routing problem within the run-time controller is therefore simpler than its global and detailed counterparts at the level of the whole FPGA circuit, since the

Table 4-1 – **Portion of an example interconnection matrix dedicated to a routing resource graph**

	$WC0$	$WC1$	$\cdots$	$EC0$	$\cdots$	$SL0$
$WC0$	$\diagup$	$WC0 \rightarrow WC1$	$\cdots$	$WC0 \rightarrow EC0$	$\cdots$	$WC0 \rightarrow SL0$
$WC1$	$WC0 \rightarrow WC1$	$\diagup$	$\cdots$	$\diagup$	$\cdots$	$WC1 \rightarrow SL0$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$	$\vdots$
$EC0$	$WC0 \rightarrow EC0$	$\diagup$	$\cdots$	$\diagup$	$\cdots$	$EC0 \rightarrow SL0$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$	$\vdots$
$SL0$	$WC0 \rightarrow SL0$	$WC1 \rightarrow SL0$	$\cdots$	$EC0 \rightarrow SL0$	$\cdots$	$\diagup$

solution space becomes much smaller when only considering the interconnection network of a single macro-cell. The fundamental function of the decoding algorithm is, for each route of the macro-cell route list, to output words of $Nb_{interconnect}$ bits with the enabled switches set as logic ones. The subsequent decoded routes of the route list of the macro-cell can be OR-ed to reveal the complete state of the macro-cell bit-stream once the end of the route list is reached.

An iterative algorithm such as PathFinder [ME95] is not suited to cope with the online constraints of the runtime controller given in Section 1.1, due to the frequent route rip-ups and the overall time complexity of the algorithm. Instead, we developed two specific algorithms for our architecture model: a fast decoder using Look-Up Tables (LUTs), and a sequential Finite State-Machine (FSM) algorithm. They are effortlessly adaptable to different parameters of the model such as the channel width W and the number of logic inputs and outputs L. Both algorithms are based on a table of realizable routes. For this purpose, a complete matrix of the routes between each input and output of the macro-cell is generated using the routing resource graph of the target architecture. This matrix allows establishing a map of possible and impossible routes which can be taken to connect one input or output to another. In the case of the simple macro-cell architecture with 21 inputs and outputs schematized in Figure 4-2, the result is a symmetric matrix of size 21×21, of which a part is detailed in Table 4-1.

2.1 LUT-based decoder

The LUT-based algorithm relies on the storage in a memory of most relations between a route and the associated switch states of the BS. Using the example of Figure 4-2, a complete LUT would take

$$(4W + L)^2 \times Nb_{interconnect} \tag{4-1}$$

bits. $Nb_{interconnect}$ can be calculated with Equation 3-2 using the macro-cell parameters $W = 4$ and $L = 5$, and amounts to 129 bits, leading to a complete LUT size of $21^2 \times 129 = 56,889$ bits. However, as previously stated, this complete LUT is symmetric since the path from IO_a to IO_b is the same as the one from IO_b to IO_a.

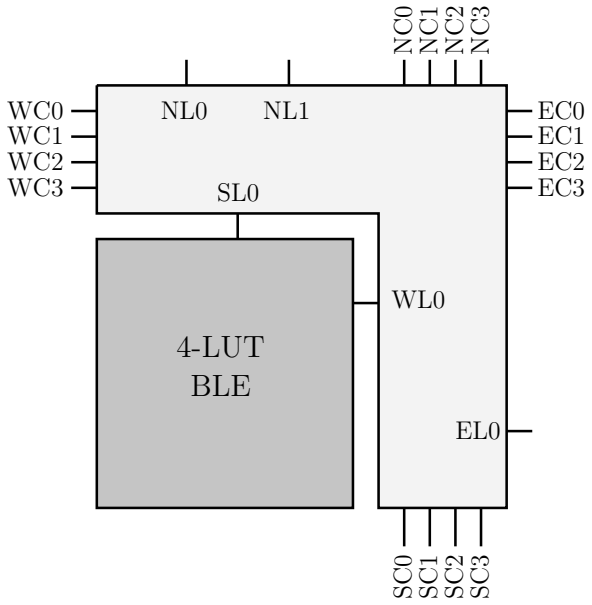


Figure 4-2 – **Example macro-cell architecture with $W = 4$ and $L = 5$.**

This property renders one half of the diagonal matrix useless. Moreover, as the route generation is dependent on the target architecture interconnection structure, all the routes are not necessarily realizable. Indeed, our example uses a switch block with a parameter $F_s = 3$, where F_s is the number of incident wires reachable by an input pin of the switch box. Therefore, because of the limited subset of routing resources, it is not possible to route a signal from $EC0$ to $EC1$ using this switch box topology. These impossible routes add up to the number of unnecessary elements in the complete LUT, although it heavily depends on the routing structure. Hence, the complete matrix of possible paths through the macro-cell interconnect would either contain a lot of unused elements, or a complex decoding logic to implement a sparse matrix.

To circumvent these issues, we split this LUT into three smaller memories, using features from the model nomenclature detailed in Section 2.2 of Chapter 3. The first LUT, LUT_{CC}, is a matrix of routes between inputs and outputs of the *connection* type only, resulting in a matrix of size $4W \times 4W$. The second one, LUT_{LL} is for I/Os of *logic* type only, and the third one, LUT_{CL} gives the route between an input of type *connection* and an output of type *logic*, or *vice versa*. Their respective matrices are of size $L \times L$, for LUT_{LL}, and $4W \times L$, for LUT_{CL}. Additionally, instead of giving the whole $Nb_{interconnect}$ bits of the interconnect, the LUT stores three addresses to hash-maps containing configurations of, respectively, the horizontal routing channel, the switch box, and the vertical routing channel, which are merged together to create the configuration BS of the macro-cell for a single route in a single cycle. The hash-maps allows for the reduction of the memory footprint of the route storage, as multiple routes may activate the same configuration switches in a connection block or switch block.

The three LUTs of addresses and the associated hash-maps only take $11,920$ bits of memory in the end for our generalized architecture, which can fit into half of

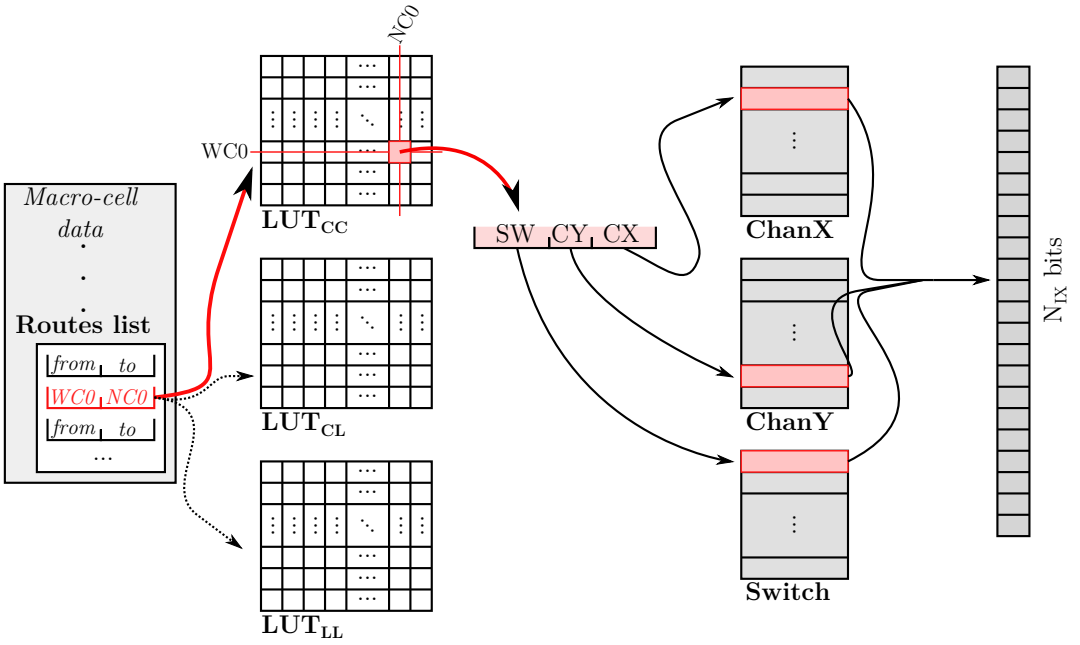


Figure 4-3 – **Details of the LUT-based decoding from the route data to the interconnect bit-stream.** *The LUT-based decoder relies on three LUTs LUT_{CC}, LUT_{CL} and LUT_{LL} able to give the interconnect configuration for three categories of routes: connection-connection, connection-logic and logic-logic. The redundant interconnection data is stored in one hash-map per feature of the local interconnect of macro-cells, and the LUTs yields addresses to each hash-map. The final interconnect data is recovered from the hash-map content.*

a 36KB block RAM of Xilinx FPGAs. At runtime, the number of cycles required to generate the final macro-cell bit-stream BS is at most equal to $N_{routes_{max}}$, the maximum number of routes in a single macro-cell.

For standard FPGA architectures where the connection block only contains at most W pass-transistors per input or output pin of the logic element, the solution space is much smaller than with the generalized architecture, and allows achieving an even smaller memory footprint with this algorithm.

2.2 State-machine decoder

The second algorithm, illustrated in Figure 4-4, implements the routing mechanism as a finite-state- machine (on the right) which is automatically generated offline for the target architecture (on the left). Each state of the machine corresponds to an input/output ($WC0$, $NC0$, $EC0$ and $SC0$ in Figure 4-4), a 4-wire interconnection switch ($SW1$), or a 3-wire interconnection switch (SW_0, SW_2) of the routing structure. Starting from an initialization state *Start*, the transitions are determined as a function of the input and output of a connection out of the connection list of the macro-cell. Each state contributes to the $Nb_{interconnect}$ bits of the macro-cell interconnect. Indeed, each state output is OR-ed and accumulated with the previous

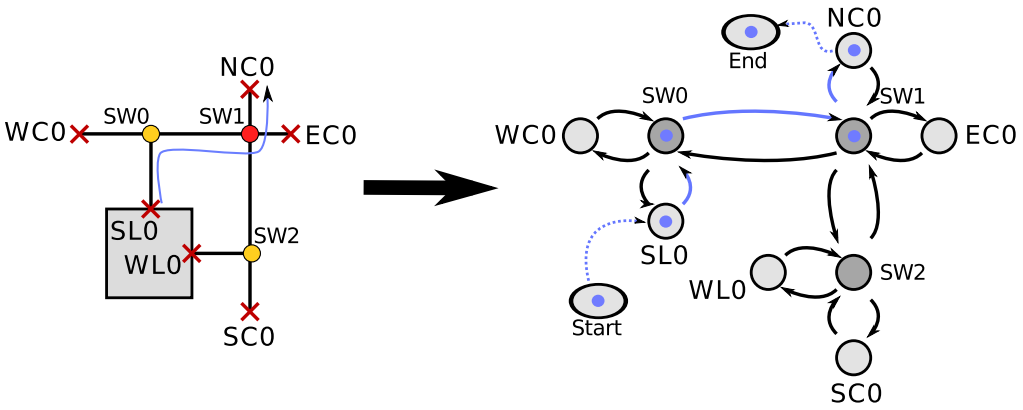


Figure 4-4 – **Principles of the FSM-based algorithm, from routing to state machine.** *The FSM-based algorithm uses the details of the macro-cell interconnect to build an FSM from the routing graph where each state is an input/output or a switch of the detailed routing and each transition is a possible path. The interconnect data are recovered by a finite set of transitions through the state machine, parameterized by the desired input and output. Each state of the FSM is able to recover one switch configuration at a time for the considered route.*

Table 4-2 – **Online LUT-based and FSM-based and offline Maze routing algorithms time and space complexities**

Algorithm	Time complexity	Space complexity
Maze routing	$\mathcal{O}\left({N_{nodes}}^2\right)$	$\mathcal{O}\left({N_{nodes}}^2\right)$
LUT-based	$\mathcal{O}\left(1\right)$	$\mathcal{O}\left({N_{IO}}^2\right)$
FSM-based	$\mathcal{O}\left(route_{size}\right)$	-

$route_{size} \ll N_{IO} < N_{nodes}$

one, as shown in Figure 4-1.

Figure 4-4 illustrates one example of sequential evolution of the algorithm for the route $SL0 \leftrightarrow NC0$, which goes through two intermediate nodes $SW0$ and $SW1$ before ending into its sink $NC0$.

With this algorithm, the memory consumption is much smaller and scales better than the LUT algorithm, at the cost of an increased number of cycles needed to generate the configuration of one route. The total number of cycles required to create the conventional bit-stream BS of a macro-cell is at most $N_{routes_{max}} \times max(route_{size})$, where $route_{size}$ is expressed in number of interconnection switches crossed to reach the end node.

2.3 Comparison

The complexity results presented in Table 4-2 outline the efforts required in terms of computation time (time complexity) and memory footprint (space complexity) to decode the route list of a macro-cell. The maze router complexity is also shown for

comparison. We define N_{nodes} as the number of nodes (i.e. vertices) in the graph representation of the interconnection network. The maze router is able to provide the shortest possible route if it exists, at the cost of a higher computation time and memory footprint, which depends on the square of the number of I/Os in the macro-cell. The LUT solution allows reaching the pre-computed solution of a route in constant time, but requires a lot of memory for realistically dense interconnection networks. On the other hand, the FSM runs in linear time, in function of the number of inputs and outputs $4W + L$ of the macro-cell. Its memory footprint is irrelevant as the only real memory used by the state-machine is its registers and the FSM decoding logic, hence expressible as a function of the number of states $N_{nodes} + N_{IO}$ of the FSM.

2.4 Implementation results

In order to provide reliable data on the actual usability of this solution, we implemented the two algorithms detailed in Section 2. The sizes and organization of the memories of LUT algorithm were computed by a dedicated application. The sum of the sizes of both the LUTs and the hash-maps is reported in Table 4-3. Similarly, the VHDL description of the FSM corresponding to each couple (W, L) was generated and synthesized to a 65nm technology node from STMicroelectronics. The number of states of the FSM is reported in Table 4-3. We did not implement the maze routing algorithm since its complexity makes it unable to meet real-time constraints at realistic operating frequencies.

The constraint we fixed for the reconfiguration is a clock F_{clk} of $100MHz$ on a 16-bit reconfiguration bus, leading to a throughput of $1,600Mbps$, close to the actual reconfiguration speed of FPGA vendors, as seen in Section 1.1. At this speed, the required time (in microseconds) to load the raw bit-stream of a single macro-cell is $T_{load}(BS) = \frac{N_{IX}+N_{logic}}{1600}$, where $N_{interconnect}$ and N_{logic} are respectively the number of bits of the macro-cell interconnect and its logic function configuration.

Table 4-3 – **Hardware implementations of the decoding algorithms**

W	L	LUT (memory *bits*)	FSM (states)
4	5	11,920	25
10	5	76,820	61
10	10	265,620	111
20	20	3,193,440	421

The LUT based algorithm complexity mostly resides in its memory consumption. At every cycle, this solution allows generating the complete set BS of states of each configurable switch of the macro-cell interconnect for a single route. The routes of a macro-cell are merged together with a boolean OR to calculate the macro-cell raw bit- stream. The time required by this solution to decode the Virtual Bit-Stream of a single macro-cell is $T_{decode}(VBS) = \frac{N_{route}}{F_{ctrl}}$, where N_{route} is the number of

connections in the connection list of the macro-cell, and F_{ctrl} the operating frequency of the controller.

We ran complex designs through the Virtual Bit-Stream design flow in order to determine the average number of routes inside the macro-cells. As an example, on the $W = 4, L = 5$ architecture illustrated in Figure 4-2, the average number of routes per macro-cell is 6, and they traverse 6 switches in average. Moreover, as detailed in Section 2.1 of Chapter 3, this architecture requires 129 bits of interconnect configuration. Using these numbers, the operating frequency of the controller required to match the previously defined minimum throughput constraint of $1,600Mbps$ is $F_{CTRL} = \frac{1600}{129} \times 6 = 75MHz$, which is less than our reconfiguration clock constraint. However, due to the memory complexity of the LUT, this algorithm does not scale well to large macro-cells, but still provides a $\mathcal{O}(1)$ complexity for small macro-cells.

The FSM based algorithm guarantees to find the configuration of a macro-cell in at most $N_{route} \times N_{switch}$ cycles, since each state of the FSM outputs the configuration of a switch. This algorithm has a fairly linear complexity depending on the number of inputs and outputs in the macro-cell, and uses virtually no memory apart from the state machine registers and the associated logic. The complexity of this solution resides on the number of states of the FSM. Table 4-3 lists the number of states required to implement the Virtual Bit-Stream decoder for multiple architecture parameters. The number of states depends on the number of interconnect switches of the architecture, and thus linearly depends on W and L.

The controller frequency required for this algorithm to match the throughput constraint is $F_{CTRL} = \frac{1600}{129} \times 6 \times 6 = 446MHz$. This solution requires more cycles to generate the configuration of a single macro-cell since it also depends on the number of switches used by the net, but the required F_{CTRL} is still realistically reachable. Moreover, due to the absence of dependencies between macro-cells, the controller may easily decode multiple of them in parallel at each cycle.

3 Compression effect of the Virtual Bit-Stream

In addition to the increased flexibility induced by the Virtual Bit-Stream technique, its representation allows a substantial decrease of the overall size of the binary file in comparison with the conventional raw bit-stream, mostly due to the coding of used resources only within the Virtual Bit-Stream file. As this compression brings an added value regarding the memory resources needed in the final system to support the dynamic loading of hardware tasks in the form of Virtual Bit-Streams, it is also interesting to study whether this compression ratio is beneficial over classical compressors or not.

Table 4-4 – **Benchmark set used to test the Virtual Bit-Stream technique**

Design name	Complex Logic Blocks
LU32PEEng	201
LU8PEEng	174
ch_intrinsics	95
diffeq1	90
diffeq2	53
mkDelayWorker32B	752
mkPktMerge	57
mkSMAdapter4B	254
raygentop	481
stereovision1	2,237
stereovision3	45
boundtop	716
sha	559
stereovision0	3,725
stereovision2	2,407
bgm	8,854
blob_merge	1,533
mcml	15,500
or1200	741

3.1 Experimental methodology

The Virtual Bit-Stream design flow is tested on an FPGA test architecture, illustrative of modern trends in terms of logic. This architecture routing channels feature $W = 15$, 25, or 35 wires depending on the smallest compatible channel width for each design, to prevent any over-achieving compression results with sparsely used routing networks. The logic fabric is made of Complex Logic Blocks (CLBs) containing four 6-LUTs interconnected by a crossbar matrix. The CLB routes 16 inputs and 4 outputs on its surrounding routing channels, which means that $L = 20$ regarding the abstraction model described in Chapter 3.

The set of designs used for the experiments is detailed in Table 4-4. A suite of benchmarks shipped with Verilog-To-Routing (VTR) was used to test the Virtual Bit-Stream design flow. This benchmark set comprises 19 designs. For each of these designs, we synthesized the circuit on our custom architecture using VTR and we then ran the resulting placement and routing data into the *vbsgen* back-end in order to generate the raw bit-streams and their corresponding Virtual Bit-Streams.

3.2 Results

Table 4-5 summarizes the compression results of the 19 VTR circuits. In addition to the Virtual Bit-Stream coding, the *Deflate* column gives the compression results

Table 4-5 – **Results underlining the compression effects of the Virtual Bit-Stream**

Name	W	BS [kbits]	Deflate [kbits]		VBS [kbits]		
LU32PEEng	15	63	38	(1.64×)	12	(4.93×)	±0.40%
LU8PEEng		55	33	(1.65×)	11	(4.94×)	±0.28%
ch_intrinsics		63	27	(2.33×)	11	(5.38×)	±0.23%
diffeq1		81	31	(2.57×)	15	(5.39×)	±0.12%
diffeq2		34	16	(2.12×)	6	(5.36×)	±0.64%
mkDelayWorker32B		359	148	(2.42×)	67	(5.30×)	±0.06%
mkPktMerge		34	16	(2.12×)	6	(5.38×)	±0.13%
mkSMAdapter4B		72	41	(1.75×)	14	(5.04×)	±0.47%
raygentop		321	127	(2.52×)	60	(5.31×)	±0.09%
stereovision1		637	367	(1.73×)	130	(4.87×)	±0.05%
stereovision3		14	8	(1.72×)	2	(5.30×)	±0.72%
boundtop	25	320	180	(1.77×)	42	(7.56×)	±0.11%
sha		253	126	(2.00×)	32	(7.88×)	±0.10%
stereovision0		1,667	872	(1.91×)	217	(7.66×)	±0.09%
stereovision2		1,086	564	(1.92×)	141	(7.69×)	±0.08%
bgm	35	5,321	2,853	(1.86×)	535	(9.93×)	±0.03%
blob_merge		951	469	(2.02×)	91	(10.36×)	±0.03%
mcml		9,199	4,498	(2.04×)	906	(10.14×)	±0.02%
or1200		1,424	358	(3.97×)	125	(11.36×)	±0.18%

W: channel width. BS: reference raw bit-stream size. Deflate: deflate-compressed raw bit-stream size and compression ratio. VBS: VBS size and compression ratio.

with the deflate algorithm, based on Lempel-Ziv 77 (LZ77) and Huffman coding, as a point of reference with a standard compressor. The VBS column indicates the average size of the Virtual Bit-Stream file over 10 runs of the design flow with different placement seeds used in VPR, as well as the maximum deviation from the mean value.

The raw bit-streams and Virtual Bit-Streams were generated on the minimum architecture size allowing the design to fit. For this reason, some benchmarks share the same raw bit-stream size, since they also share the same logic fabric size. 12 of the designs fill over 85% of their allocated logic fabric.

In average, the Virtual Bit-Stream is 6.41× smaller than its equivalent conventional bit-stream and 3.04× smaller than the deflate-compressed bit-stream. The compression ratio can go as high as 11.36×. The maximum deviation from these results observed over 10 different routing passes of the VTR benchmark set is less than 0.5%. The compression effect of the Virtual Bit-Stream is hence stable over multiple runs.

The designs which gain the most from the alternative coding of the interconnect network are the largest ones which are the least congested. Indeed, the less the

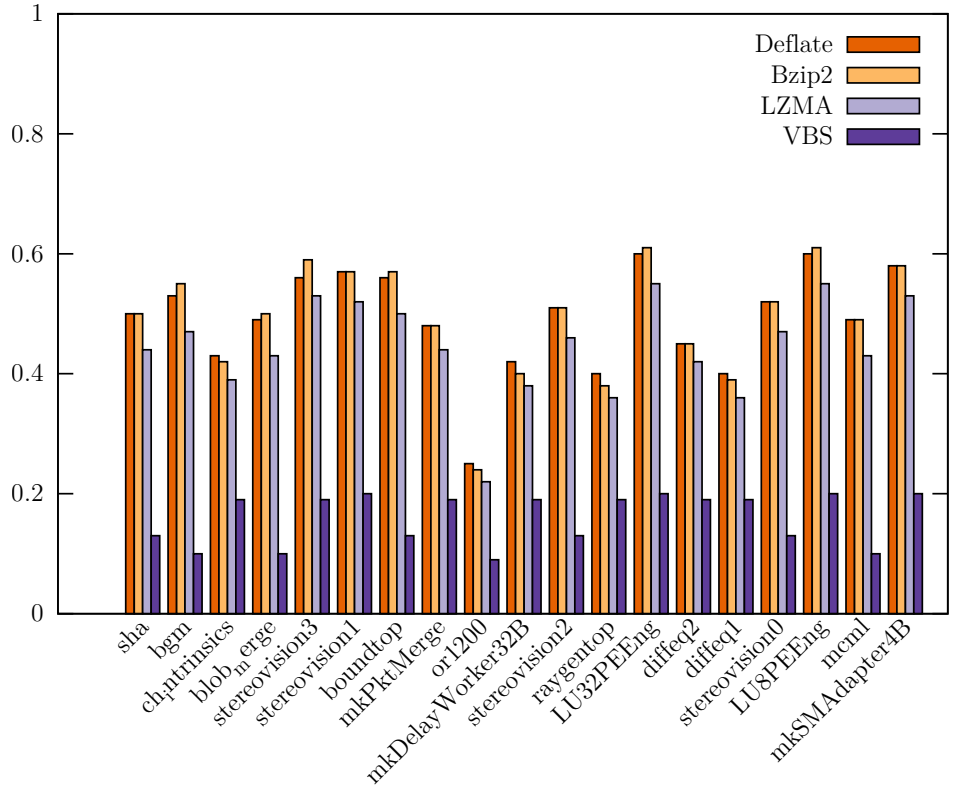


Figure 4-5 – **Normalized compression ratio of Deflate, Bzip2 and LZMA compressed raw bit-streams and VBS compared to the raw bit-stream size.**

Virtual Bit-Stream file has to store I/O connections for a given macro-cell, the higher the compression ratio is, since a raw bit-stream encodes the state of the pass-transistors no matter if the macro-cell is used or not.

Figure 4-5 plots the sizes of the compressed raw bit-streams and of the Virtual Bit-Stream file normalized to the raw bit-stream size. Three general compression schemes have been explored: Deflate, as seen in Table 4-5, Bzip2 and Lempel-Ziv Markov chain Algorithm (LZMA). The three general compressors offers similar results regarding the compression of the raw bit-stream, while the Virtual Bit-Stream consistently offers a compression ratio at least 2× better. The general compression techniques generally works on a dictionary basis, which leads to mitigated results for the raw bit-stream, whereas the Virtual Bit-Stream representation uses a deep knowledge of the data itself to obtain high compression ratios.

3.3 On the effect of clustering

As detailed in Section 3 of Chapter 3, higher compression ratios can be achieved by grouping multiple macro-cells together into a cluster for which the routes will be described as a whole in the resulting Virtual Bit-Stream. Using the same benchmark

Table 4-6 – **Effect of the clustering on Virtual Bit-Stream sizes**

Cluster Size	Min. [kbits]	Max. [kbits]	Mean [kbits]
1	2	906	128 (6.31×)
2	3	847	120 (6.47×)
3	4	831	120 (6.42×)
4	3	858	122 (6.26×)
5	4	816	121 (5.89×)
6	6	829	124 (5.89×)

set as in Section 3.1, we took the same placement and routing data generated by VTR and ran our back-end by specifying cluster sizes of varying sizes from 1 to 6. Table 4-6 provides results on the effect of clustering on the Virtual Bit-Stream. The size of the Virtual Bit-Stream file is given as the minimum, maximum and mean values for all the 19 benchmarks of Table 5-1 and the compression ratio is given only on average.

From the result set we obtained, we found that a cluster of size 2 or 3 is the best possible choice. This size of cluster allows to further increase the compression ratio over the benchmark suite. Clusters of bigger sizes show very little deviation from these ratios or, worse, make the compression ratio decrease.

The maximum in compression ratio seen for clusters of size 2 and 3 should be compared to the average wire length of the routed benchmark set. Figure 4-6 illustrates the frequency of appearance of average wire lengths from 0 to 20 over 10 runs of placed and routed VTR benchmarks. The average wire length reported by VPR after a successful routing is the average Manhattan distance traveled by the routes of the considered design. We can observe that most of the designs have an average wire length of up to ≈ 9, which is comparable to the maximum wire length attainable in a cluster of 3×3 CLBs. As the wire length gets close to the maximum length of a wire which can be contained into a single cluster without going out of this cluster, the compression effect of the Virtual Bit-Stream format becomes the most effective. Many of the wires precedently split across multiple macro-cells ends up to be coded into a single cluster of macro-cells, hence the results observed for cluster of size 2 and 3.

3.4 Fallback coding: to the limits of the Virtual Bit-Stream

The LUT and FSM based algorithms described in Section 2 are heuristics relying on a pre-computed matrix of possible routes. As such, it is possible that they could lead to false routings, for example in the case of a conflict between two routes in a congested macro-cell. A verification phase allows the *vbsgen* tool to check the validity of the generated routing. The coding format of Virtual Bit-Stream files detailed in Chapter 3 includes a fallback coding of the routing data of a macro-

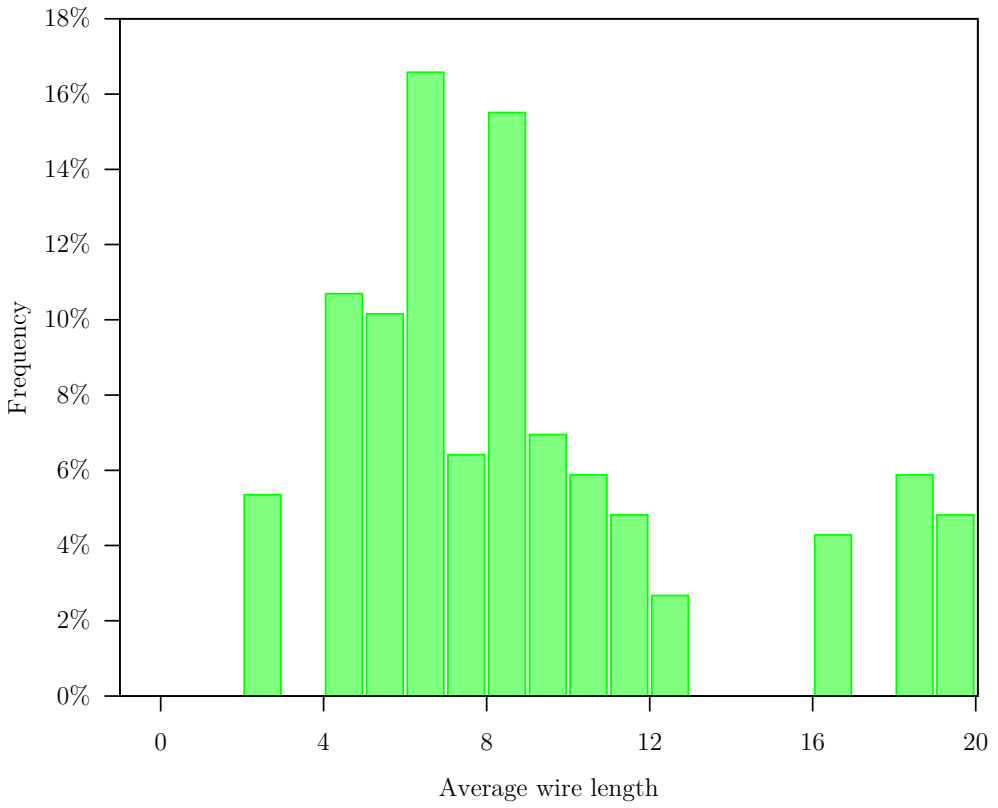


Figure 4-6 – **Frequency of apparition of average wire lengths of 10 place and route runs of the VTR benchmarks.** *Each design of the benchmark set has been placed and routed 10 times with different seeds, which led to different average wire lengths for each design and each run. The frequency of apparition of wire lengths from 0 to 20 details the number of time an average wire length is seen in the 10 runs of the benchmark set. As most designs have an average wire length* < 9*, it seems to be correlated to the observed results of the clustering on compression ratios.*

cell, in case no valid order of the routes in the route list yields a decodable Virtual Bit-Stream. The fallback coding uses the conventional raw representation of the data into the macro-cell so as to still generate a valid Virtual Bit-Stream file, even in the case of a particularly congested macro-cell interconnect for which our online algorithms could not retrieve a valid configuration.

This original representation of the interconnect forbid any compression effect for the considered macro-cell. Even worse, the total size needed to code the macro-cell still includes its meta-data, which leads to a bigger data size in any case. The online algorithm and the routing congestion push the Virtual Bit-Stream technique to its limit, as the difficulty to find a valid representation increases with the amount of routes going through a single macro-cell.

The experimental methodology detailed in Section 3.1 has been used to extract real data on the amount of macro-cells coded as raw bit-streams in the Virtual Bit-Stream files. The amount of fallback-coded macro-cells was determined over 10 runs of different placement and routing of the benchmark set, using a software

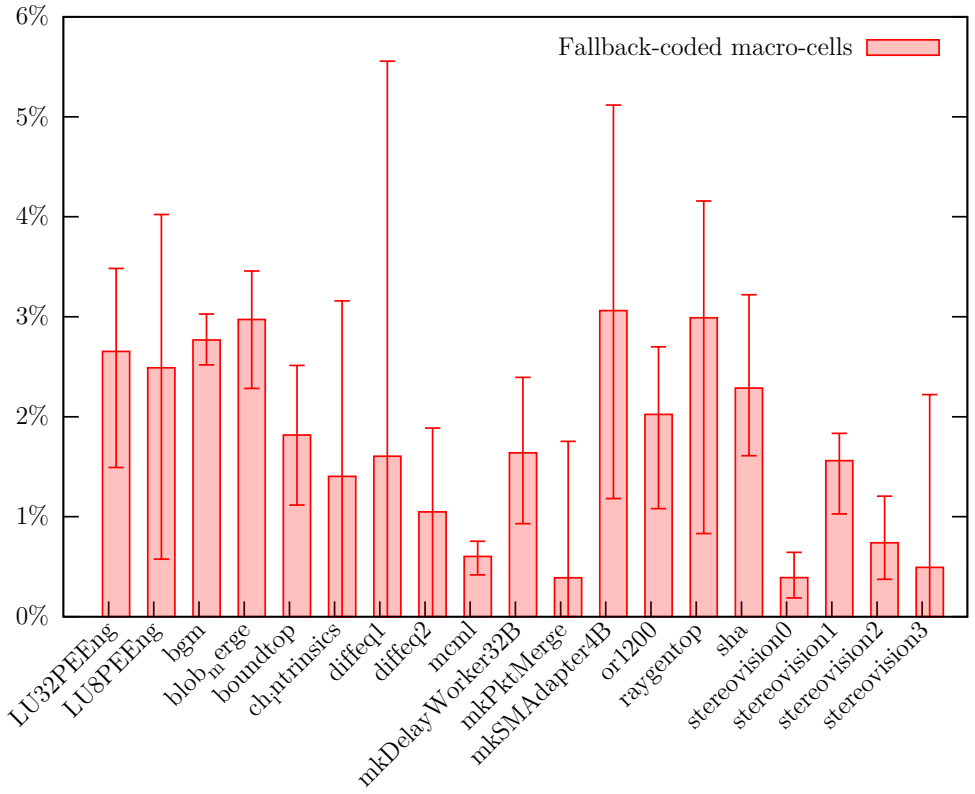


Figure 4-7 – **Normalized number of fallback-coded macro-cells with the LUT algorithm for each design of the VTR benchmark set.** *The average percentage of fallback-coded macro-cells indicates how many times the algorithm fails over the set of macro-cells of each design. The error bars represent the maximum and minimum values of the percentage of fallback-coded macro-cells over the 10 different place and route trials of the benchmark set.*

implementation of the LUT-based decoder adapted to channels of width 15, 25 and 35, depending on the smallest of these values capable of routing the designs. Any macro-cell which was not decodable by the LUT-based decoder was directly coded with its raw interconnect data, without triggering a new place-and-route step for the considered design.

Figure 4-7 plots the average number of fallback-coded macro-cells over 10 place-and-route runs of the 19 designs of the VTR benchmark set, normalized to the total number of macro-cells of the design. Overall, the number of fallback-coded macro-cells is below 3% in average, and up to 5.5% for the most extreme case. The volume of fallback-coded macro-cells is not correlated with the design size, in terms of number of macro-cells, as large designs may have as few fallback-coded macro-cells as small designs. On the contrary, the difficulty of coding the route list of macro-cells is closely related to the number of routes in the macro-cell, which tends to associate the incapacity of finding a valid route list with the routing congestion of the macro-cell.

4 Conclusion

This chapter detailed the benefits and constraints associated with an online implementation of the Virtual Bit-Stream technique. First, two real-time algorithms required to decode the Virtual Bit-Stream data have been proposed and proven reliable at frequencies found for the reconfiguration of modern devices. Interestingly, the compression effects of this alternate representation of bit-streams have been explored and showed compression results of up to 11×. Eventually, the limits of the Virtual Bit-Stream coding were presented with a study of the quality-of-service of the coding of interconnection data within each macro-cell, among which some of them needed to be coded as raw data.

The next chapter presents architectural enhancements of the global FPGA interconnection so as to strengthen the placement flexibility of hardware tasks on the logic fabric. Quantitative results of this enhanced routing network are studied in terms of critical delay and logic area.

Chapter **FIVE**

ARCHITECTURE ENHANCEMENTS

Contents

1	Task migration with heterogeneous blocks	**98**
	1.1 Hard blocks abstraction	99
	1.2 Partitioning	101
	1.3 Task model	101
2	Experimental methodology	**102**
	2.1 Modeling in VPR	102
	2.2 Benchmarks	104
3	Results	**104**
	3.1 Logic array size	105
	3.2 Critical delay	105
	3.3 Routing resources	106
4	Conclusion	**108**

Abstract

This chapter describes FPGA routing architecture changes which allow tasks to be migrated so that heterogeneous (hard) blocks are located in different locations within the target Partially-Reconfigurable Region (PRR) [HST14]. The proposed approach aims to prevent any time-consuming place-and-route process during the migration process. Since hard blocks generally are aligned in vertical columns in contemporary FPGAs, this solution provides the ability for tasks to *float* horizontally within the logic array. First, FPGA routing architecture enhancements which allow for routing connections to hard blocks which are isolated from other routing in the FPGA routing fabric are proposed. Eventually, an evaluation of the performance and area overheads incurred by this routing isolation is performed for a collection of benchmark designs which include components mapped to both FPGA logic and hard resources.

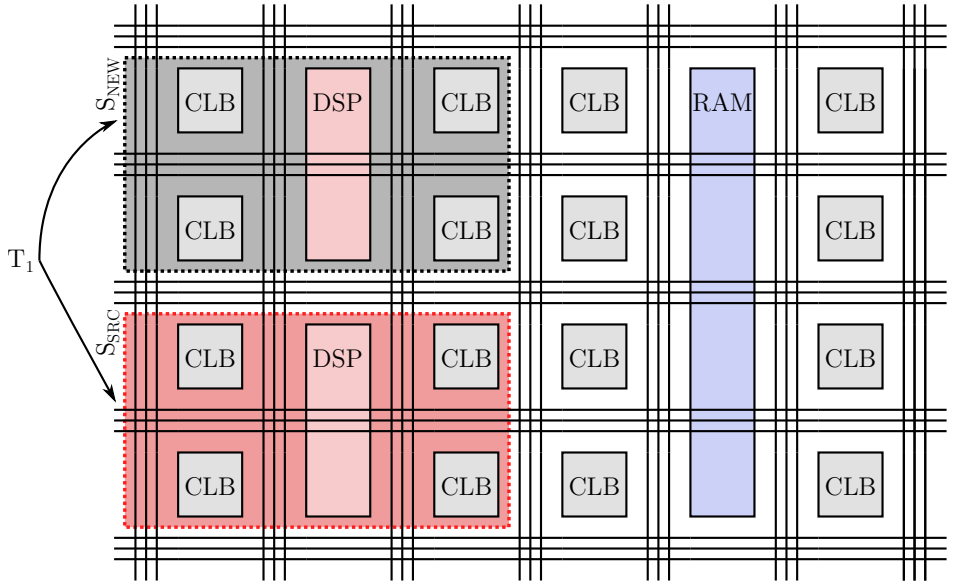


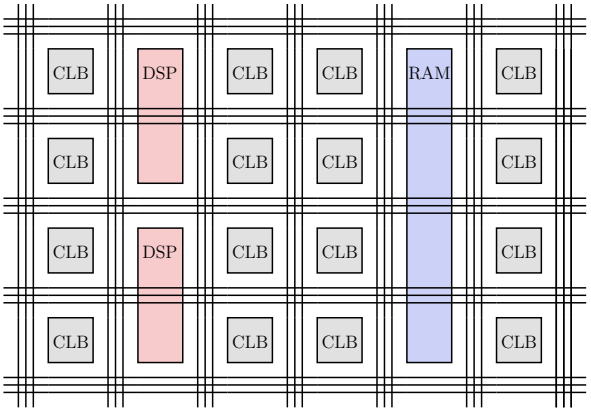
Figure 5-1 – **Relocation problem in modern FPGAs.** *Task T_1, placed on a set S_{src} of resources, is only movable to another set S_{new} with resource types and positions matching those in the original set. The necessity to find matching partitions to perform the relocation of a task hampers the overall placement flexibility of the device.*

1 Task migration with heterogeneous blocks

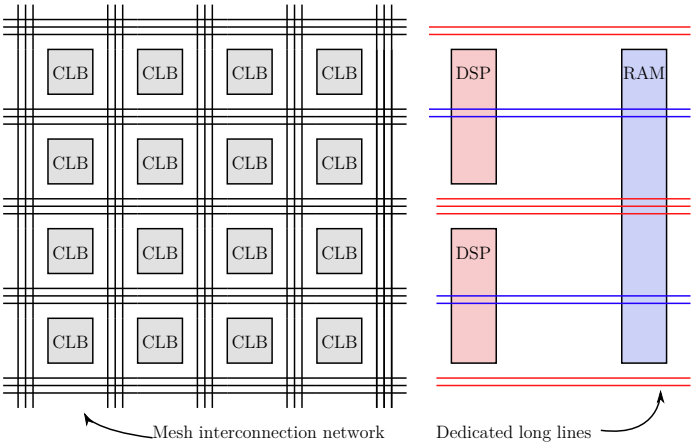
In an island-style FPGA, hard blocks such as memories and arithmetic accelerators are aligned in columns to simplify FPGA layout and the combining of multiple blocks into larger functional resources, as detailed in Chapter 1. Depending on block complexity, the height of a hard block is often larger than a single grid location (e.g. the height of a logic block), as shown in Figure 5-1.

The allocation of hard blocks across the logic fabric complicates task relocation for the shaded task which spans three columns in Figure 5-1. Given a hardware task placed at the position (x_{source}; y_{source}) in the array, it is limiting to find a position (x_{new}; y_{new}) in the logic fabric that contains a set of resources S_{new} with all resources in exactly the same relative position as S_{source}. Overcoming the restrictions imposed by hard resources requires making the routing connections to the hard blocks more flexible.

The architecture enhancements described in this chapter rely on a separation of the hard blocks routing from the conventional soft logic routing. With the use of long lines dedicated to each of the hard blocks in a bounded portion of the FPGA device, the online placement of tasks can be eased on one of the two axis of placement. As the connection from the *homogeneous* routing to the *heterogeneous* long lines can be made at multiple equivalent points of connection, this technique effectively increases the placement flexibility of hardware tasks.



(a) **Original logic array**



(b) **Resulting set of logic-only and hard only arrays**

Figure 5-2 – **Abstraction of heterogeneous resources.** *The original logic array, based on an island-style architecture, contains CLBs as soft logic elements and two types of hard blocks. The interconnection network is shared by both soft and hard blocks. In the abstraction of the logic array, the soft logic lies in one mesh interconnect network dissociated from the hard block routing. The connection to and from hard blocks is made via long lines dedicated to each type of hard block.*

1.1 Hard blocks abstraction

Increased routing flexibility for hard blocks is introduced via an abstraction of heterogeneous resources. Starting from the original architecture illustrated in Figure 5-2a, two separate logic resource layers are considered. The first layer, the logic array, includes the logic resources (Logic Blocks (LBs), CLBs) of the architecture and its routing network, connection boxes and switch boxes. The second layer is made up of the hard blocks in the baseline logic array (e.g. memory blocks, arithmetic accelerators, communication interfaces, etc.).

The *heterogeneous* plane has its own routing network, composed of bidirectional routing long lines which span a defined region of the FPGA. Each of these lines can be connected to an input or output of the fixed function blocks. The hard block input and output connections are restricted to their dedicated routing network. Links to

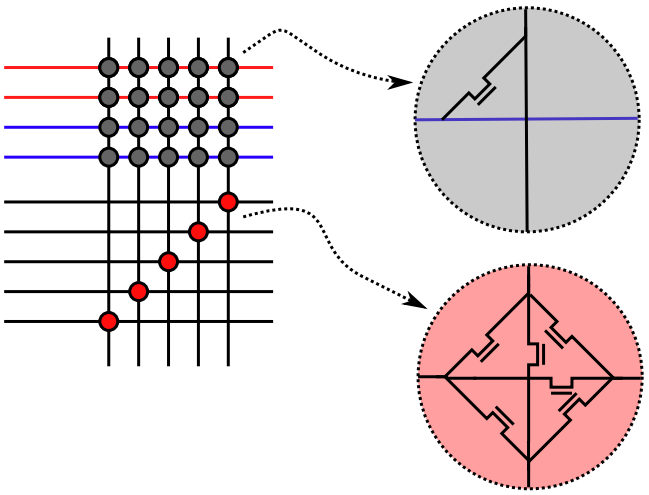


Figure 5-3 – **Switch box enhanced to include connections between heterogeneous and homogeneous layers.** *The red and blue horizontal wires (the top four wires) represent heterogeneous-only long lines while the remaining black wires represent homogeneous routing. The interconnection between the heterogeneous-only long lines and the homogeneous routing is made through connection points at the switch level.*

the logic array are made via the heterogeneous-only long lines which connect to switch boxes of the logic array routing network. The extent of each of the long lines is optimized to provide good delay performance. The abstraction is presented in Figure 5-2b.

It should be noted that this abstraction does lead to an increase in horizontal delay for routing in the logic-only (*homogeneous*) plane. Since the positioning on the die of hard blocks is not known at place and route time, it must be assumed that such a block is physically located horizontally between each LB. The additional delay associated with the longer horizontal wires in the homogeneous plane must then be considered.

1.1-1 Routing network separation

The heterogeneous-only long lines only carry signals to or from hard blocks. The remaining homogeneous routing network carries signals emanating from logic blocks. Heterogeneous-only long lines are connected to the homogeneous routing network through switch boxes at horizontal and vertical channel intersections, as shown in Figure 5-3. Routing segments spanning multiple logic (e.g. 2, 4, 8, etc) are commonly found in island-style architectures. These segments do not make intermediate switch box connections to reduce wire delay. However, the heterogeneous-only long lines do connect to multiple adjacent switch boxes. Thus, it is possible to connect to these lines from multiple switch boxes.

This additional level of routing allows for the ability to slide a given task horizontally along the horizontal heterogeneous-only long lines without affecting either layer

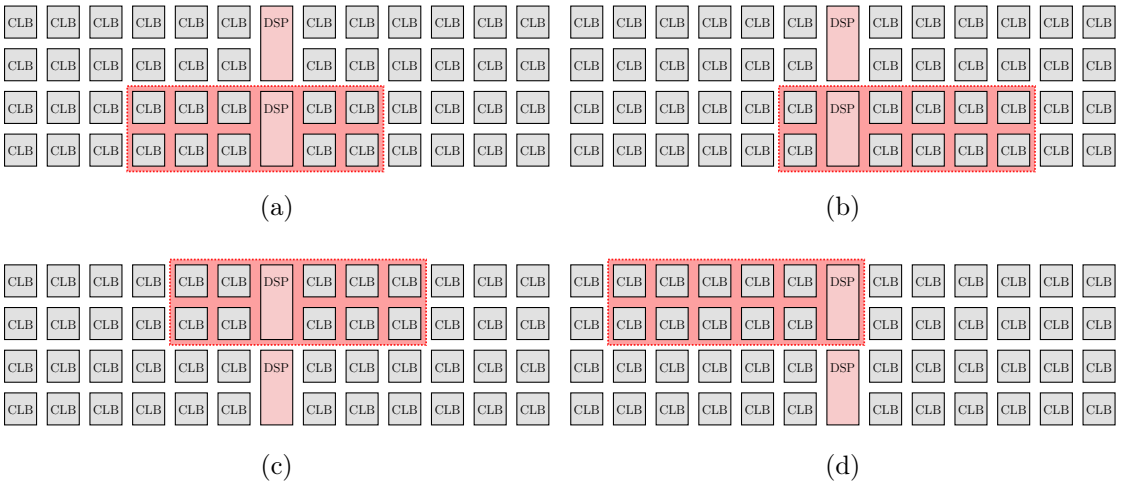


Figure 5-4 – **Multiple relocation possibilities for the same bit-stream of a task.** *As the connections between the homogeneous routing and the hard blocks can be made at multiple equivalent points of connection within switches, the task benefits from an increased flexibility in its horizontal placement. The vertical placement possibilities are still limited by the common denominator of the hard blocks heights.*

of routing, as illustrated in Figure 5-4. Only the use of horizontal, heterogeneous-only long lines for hard input and output signals was explored to avoid complex routing and excessive overhead for the overall circuit.

1.2 Partitioning

Since the size of hard blocks can vary, the vertical and horizontal extent in LBs of a reconfigurable *partition* must be carefully selected. The width of the partition is bounded by the length of the heterogeneous-only long lines. The height of a partition is defined as the least common multiple of the height of the hard blocks it contains. In this work, the FPGA partitions are identical in size and they are evenly distributed. The partitioning also improves the scalability of our approach: partitions can be duplicated to extend the size of the logic fabric.

1.3 Task model

We consider a hardware task to be a set of logic resources that are interconnected. A hardware task is described by a bit-stream to be loaded into the configurable logic fabric. Unlike standard FPGA development where a hardware task contains an Intellectual Properties (IP) block that is loaded into the FPGA at a specific location, in this case a task is an aggregate of both interconnected homogeneous logic resources and a set of connections to heterogeneous-only long lines, as depicted by Figure 5-5. Each connection is characterized by its position (x, y) within the task and its type t is used to determine its resource connection. This set of connections

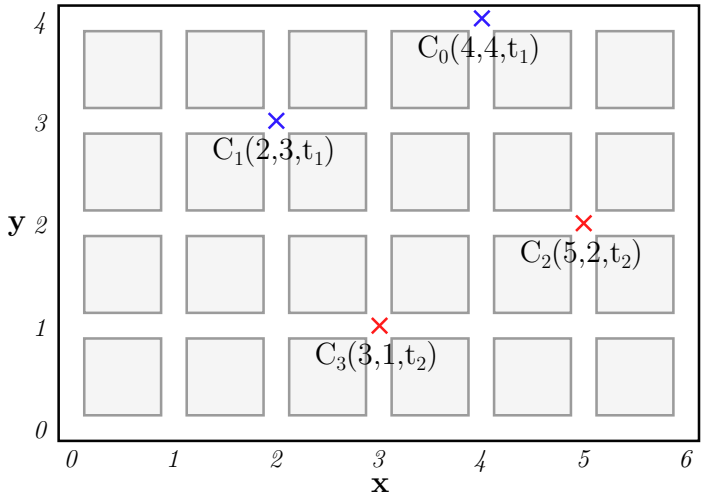


Figure 5-5 – **Model of a task with abstracted heterogeneous hard blocks.** *Tasks are defined as a set of homogeneous logic elements and a variable number of connections C_i to hard blocks. Each connection C_i has to be linked to a hard block via heterogeneous-only routing. A valid placement of a task needs to satisfy the connection type constraint for all the connections.*

is fixed, relative to the task itself, and the problem of relocation requires finding a corresponding match of hard resources to this set. A task is defined as a bounded set of logic blocks and a set $T = \{C_i(x_i, y_i, t_i)\}$ of connections to hard blocks, where x_i, y_i is the position of the connection and t_i is its type.

2 Experimental methodology

To experimentally evaluate the impact of our revised FPGA architecture on FPGA area and performance, the VPR tool set (version 6.0) was modified. The following subsection details changes made to VPR for experimentation. In subsection 2.2, we describe the set of benchmark designs used for our experiments.

2.1 Modeling in VPR

The changes made to the VPR source code were made on the latest version[1] available in the VTR [Ros+12] project repository.

2.1-1 Architecture

The VPR architecture description XML file that was used to model our circuit is based on `k4_N10_memSize16384_memData8.xml`, which is one of the default

[1]The checked out version was revision 3034.

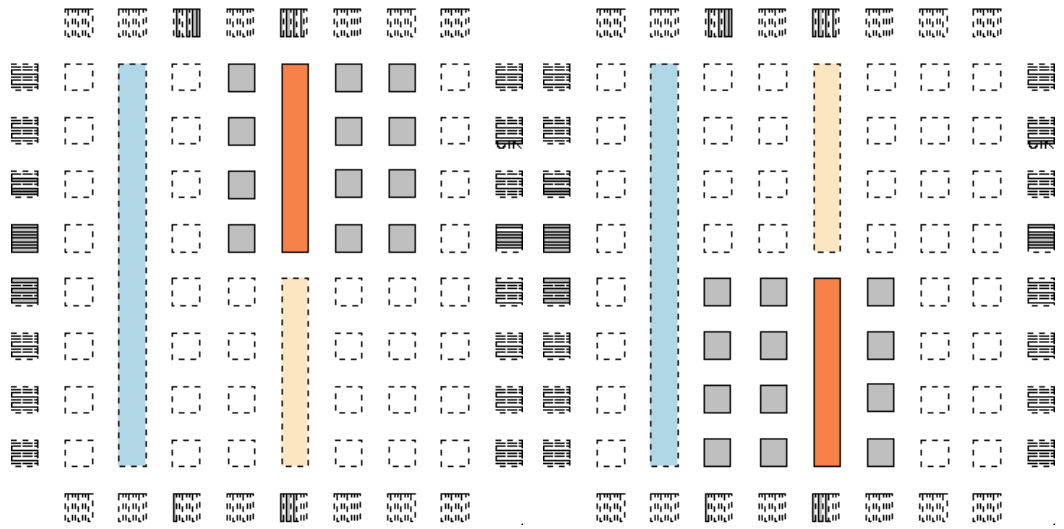


Figure 5-6 – **Two placements of the same task on the enhanced architecture with VPR**

files shipped with the VTR tool flow. The file defines a simplified version of the Stratix IV architecture. The architecture was adapted to suit our needs and to provide a generic FPGA supporting multiple types of hard blocks. The blocks include a 36×36 multiplier and a configurable dual-port RAM similar to a Stratix 144 Kbit embedded memory block. Figure 5-6 demonstrates two placements of the same task with VPR and the enhanced architecture; routing was omitted for readability.

2.1-2 Routing abstraction

The key modifications to the VPR code base resulted in a dual-layer routing graph which allows inter-layer routing connections at switch boxes (Figure 5-3).

The segment types of the separated routing networks were modeled using two different segment type definitions in the architecture file. The first segment type, used for *homogeneous* routing, contains wires which span a single logic block. The second segment, used for heterogeneous-only routing, contains long lines spanning the whole chip. Since we restrict the routing to the hard blocks to horizontal wires, the vertical routing adjacent to the hard blocks is not used. In many VPR architecture definitions, a logic array block will be defined to have its inputs and outputs spread across its four sides. This behavior is undesirable in our case since inputs and outputs on left-hand and right-hand sides of the block will connect to vertical routing channels. As a result, we defined the location of input and output signals for hard blocks as the top and bottom of the block.

Since VPR does not provide support for separate routing networks which are only

interconnected in a few locations, modifications were made to the VPR routing graph generator. The *libarchfpga* library used by VPR to parse architecture files was modified to enable the marking of `segments` of the architecture file `segmentlist` with an additional attribute, `selective_routing`. This attribute tags the segment as appropriate for either homogeneous signals (i.e. to or from a LB), heterogeneous-only signals (to or from a hard block), or both types. Our model only uses heterogeneous-only long lines and segments which can carry both signal types.

Another marking was introduced to explicitly declare a hard block in the `complexblocklist` section of the architecture file as *heterogeneous* (in our case multipliers and memory blocks). Additionally, the graph building procedures of VPR itself were adapted so that input and outputs of the complex blocks marked as heterogeneous are not routed on routing segments which are not tagged to route them. Similar marking was used for homogeneous routing segments.

In the original VPR release, long lines are balanced and their connections are rotated among the wires of the channels in which they reside. This behavior is desirable in an FPGA where one wants to leverage the full routability of the circuit. However, this rotation is not desirable in our model since connections to the heterogeneous-only long lines are more substantial. To circumvent this issue, we disabled the long line balance feature for segments marked as heterogeneous, which allows the wires modeling the hard routing to effectively span a fixed amount of the same columns. Currently, for a given defined segment, VPR will create both horizontal and vertical tracks in the routing graph. For our implementation, only horizontal long lines are used. Although vertical long lines for the heterogeneous-only routing layer were created, they were marked as unusable by the router and their area cost was ignored. All switch block and connection block connections to these wires were removed.

2.2 Benchmarks

Benchmarks with heterogeneous resources were used to evaluate our modified FPGA architecture. We choose to use the set of benchmarks included in the VTR design flow for experimentation, as detailed in Chapter 4. Among these 19 benchmarks, 12 of them that use heterogeneous blocks (i.e. memories and multipliers) were selected to test our enhanced architecture. Table 5-1 sums up the benchmark details. The number of hard blocks was calculated after the packing stage for our architecture. A total of 8 out of 12 designs make use of memories, and 7 out of 12 use multipliers. Of these designs, two of them have more than a thousand logic blocks.

3 Results

In this section we evaluate results obtained with the benchmarks described in Section 2.2 using the enhanced architecture detailed in Section 1. We use the modified

Table 5-1 – **Post-packing composition of 12 heterogeneous designs of the VTR benchmark set**

Benchmark	Memories	Multipliers	LBs
LU8PEEng	45	8	2,174
bgm	0	11	2,977
boundtop	1	0	272
ch_intrinsics	1	0	41
diffeq1	0	5	43
diffeq2	0	5	30
mkDelayWorker32B	41	0	497
mkPktMerge	15	0	17
mkSMAdapter4B	5	0	181
or1200	2	1	273
raygentop	1	7	192
stereovision1	0	38	990

version of VPR described in Section 2.1 to perform timing-driven packing, placing and routing for the 12 designs. Comparative results are obtained with respect to a *standard* architecture which is similar to the enhanced one, with the exception of the new routing for the heterogeneous-only routing. Place and route data concerning the standard architecture are presented in Table 5-2.

The basic partition of both architectures is a 8×8 array of logic blocks with one column dedicated to a memory of 8 blocks height, and another column containing two multipliers of 4 blocks height. The results of the enhanced architecture were generated with multiple side-by-side horizontal partitions of size 8 since this is the smallest common denominator of the logic fabric. Each long line tied to a specific heterogeneous block spans a single partition.

3.1 Logic array size

As observed in Table 5-3, our enhanced architecture has no influence on the minimal array size (in number of logic blocks) on which a task can be placed. It is not surprising to obtain strictly equal results for both architectures. Since our enhanced architecture proposal mainly concerns an evolution of the routing network dedicated to hard block, there is no need to increase the number of blocks required.

3.2 Critical delay

The critical delay of each circuit is expressed in nanoseconds in Tables 5-2 and 5-3. To ensure an equal quality of results for both architectures, the same timing driven options were given to VPR, and the same routing seed was used in both cases.

Table 5-2 – **Place and route benchmark results on the standard architecture**

Benchmark	Array size	Chan. width	Crit. delay (ns)
LU8PEEng	56	77	124.06
bgm	64	91	34.66
boundtop	20	36	7.17
ch_intrinsics	8	25	3.57
diffeq1	12	32	16.17
diffeq2	12	28	13.84
mkDelayWorker32B	50	65	11.25
mkPktMerge	32	24	5.60
mkSMAdapter4B	18	44	7.34
or1200	25	55	13.36
raygentop	17	45	5.11
stereovision1	37	76	6.24

Most of the benchmarks sees a critical delay change by less than 10% between the standard architecture and the enhanced one.

The critical path delay increase in Table 5-3 is limited and results from routing congestion when vertical routing tracks in the homogeneous routing layer are used to reach the horizontal long lines and the increased capacitance of the long lines for short connections. Some of the benchmarks expose a decrease in critical path delay (in particular `diffeq2`, `mkPktMerge`). These improvements originate from the use of long lines to reach heterogeneous blocks, reducing the wire delay for signals in the critical path. The critical path timing of a task must consider relocation-related issues. Since the logic content of a relocatable task is placed relatively to the heterogeneous blocks, we do not know in advance where homogeneous signal will *step over* a hard block. This issue is addressed at the place-and-route stage through the use of conservative timing.

In our case, the main area impact is a result of the distribution of long lines across the FPGA: a horizontal routing channel contains the minimum number of long lines required to route every I/O of a partition of the biggest channel (i.e. the amount of connections to the most connection-intensive hard block). Long lines which are not tied to a specific I/O in a horizontal channel can be used for other purposes, which can result in a reduced critical delay.

3.3 Routing resources

Our approach primarily focuses on routing network enhancements. The isolation of routing resources for logic-block only and hard blocks has several implications. As described in Section 1, each input and output of the hard blocks is tied to a unique long line of a partition. This approach has a drawback: densely-populated partitions (in terms of hard elements) will suffer from a dense network of long routing

Table 5-3 – **Place and route benchmark results on the enhanced architecture**

Name	Size	Chan. width		H-wire	Crit. delay (ns)	
LU8PEEng	56	139	(1.81×)	324	134.41	(1.08×)
bgm	64	108	(1,19×)	324	35.67	(1.03×)
boundtop	20	56	(1.56×)	324	7.51	(1.05×)
ch_intrinsics	8	25	(1.00×)	162	3.84	(1.08×)
diffeq1	12	96	(3.00×)	234	15.96	(0.99×)
diffeq2	12	65	(2.32×)	234	12.66	(0.92×)
mkDelayWorker32B	50	117	(1.80×)	324	11.46	(1.02×)
mkPktMerge	32	92	(3.83×)	324	4.84	(0.86×)
mkSMAdapter4B	18	83	(1.89×)	324	7.35	(1.00×)
or1200	25	77	(1.40×)	324	13.62	(1.02×)
raygentop	17	98	(2.18×)	324	5.78	(1.13×)
stereovision1	37	129	(1.70×)	324	6.97	(1.12×)

lines going from side to side in the partition. The results in Table 5-3 show that for reasonably sized designs such as `ch_intrinsics` the cost in terms of routing resources increase and critical delay is moderate with 162 dedicated long lines per partition.

We observe an increase in the routing resources needed in the homogeneous routing network (the channel width in Table 5-3), of 1.97× on average. Routing resources due to the routing heterogeneous-only long lines dedicated to hard blocks in the partition are shown in the *h-wire* column in Table 5-3. It should be noted that the heterogeneous long lines (*h-wires*) in the table for each benchmark are spread across an entire *partition* and not isolated in just one channel. As the number of *h-wires* increases, the number of logic-only homogeneous wires required to route signals up to the long lines increases to avoid network congestion.

Given these results, it is reasonable to define a nominal partition width of 8 columns, including heterogeneous blocks, for this specific architecture, as used in our experiments. This size allows for good performance in terms of flexibility, while still making a good trade-off in routing resources demand and occupied area. Our approach eliminates the need for routing when a task is moved. The only required change for the task bit-stream is to shift the homogeneous-to-heterogeneous and heterogeneous-to-hard block connections horizontally. The position of these connections in the bit-stream can easily be offset by a fixed amount for the hundred or so connections for each hard block as the task is loaded into the device.

4 Conclusion

In this chapter, an enhanced FPGA architecture supporting the relocation of tasks and their associated bit-streams even if the relative position of hard blocks in the target reconfigurable region is changed. A standard island-style FPGA architecture has been expanded to include a dedicated routing layer for hard blocks. Long lines are used in this layer to allow blocks to *float* horizontally within a region, greatly expanding placement opportunities for hardware tasks.

The routing implications and costs associated with this freedom have been investigated with a variety of VTR benchmarks including hard blocks. Two factors limit the practicality of the proposed solution. First, the architecture is limited by the amount of additional routing resources required by the abstracted hard block routing. The additional long lines increase the stress of the global routing as each hard block input and output requires one dedicated long line. Second, dedicated global and detailed routing algorithms were not explored. While the original timing-driven Pathfinder router of VPR performs well on standard routing architecture, it is not optimized for the proposed architecture, which may bias the results.

As the architecture benefits from an increase in placement flexibility, the next chapter investigates means of enclosing specific portions of a logic fabric for reconfiguration, so as not to disturb active partitions of the chip and thus further increase its dynamicity.

Chapter SIX

ENHANCING THE VIRTUAL BIT-STREAM ARCHITECTURE

Contents

1 Introduction **110**
1.1 Bit-stream loading methods 110
1.2 Multi context configuration memory 114
1.3 Summary of memory organizations 115
2 Enhanced scan-path **116**
2.1 Organization of the configuration memory 117
2.2 Runtime dynamic partitioning 120
3 Conclusion **122**

Abstract

Modern FPGA device vendors nowadays propose support for dynamic partial reconfiguration of pre-defined regions. The current development flow of these devices implies the definition at design time of static regions and Partially-Reconfigurable Regions (PRRs) on which partial bit-streams dedicated to a region will be loaded. Very dynamic and flexible applications benefit from an accordingly dynamic spatial and temporal scheduling of tasks at runtime. To support such level of flexibility, an FPGA architecture must offer a configuration memory capable of loading a task at runtime without disturbing active portions of the chip. This chapter presents a technique for dynamically partition the configuration memory at runtime while keeping a low area budget.

1 Introduction

While previous chapters were directed to enhancements at the logic fabric architecture level to facilitate runtime hardware reconfiguration and dynamic task placement, the question of whether or not it is possible to actually benefits from such fine-grain placement still holds. From the FPGA device point of view, the previously proposed techniques effectively increase the architecture potential regarding its dynamicity. However, as described earlier in Chapter 1, the root of an architecture is held within its configuration memory, which provide the device with its configurability. Partial reconfiguration of specific partitions (i.e. PRRs) of the device is only possible if the configuration memory supports it and offers some means of changing a partition bit-stream without shutting-down other partitions.

From the configuration memory point-of-view, every configurable bit pertaining to the soft logic data, the configurable interconnection network or any other configurable element of the logic fabric is part of a huge memory array spread over the entirety of the FPGA device. This chapter only considers the most widespread programming technology to date: the Static Random Access Memory (SRAM) memory cells, as other programming technologies have different constraints. The configuration memory is organized independently of the shape and organization of the logic fabric and interconnect it serves, although several integration factors shall be taken into account to design the FPGA architecture along with its memory.

1.1 Bit-stream loading methods

Several bit-stream loading methods have been put in use in modern FPGA devices. Since this is one of the defining factors of the architecture performance in terms of reconfiguration capabilities and speed, it is considered as a trade secret by FPGA vendors, and hence not documented in the devices datasheets. However, the patents held by major FPGA companies allow to have an insight on the technologies involved in their devices. The following sub-sections draft the three main addressing schemes of FPGA configuration memories.

1.1-1 Word addressing

Word addressing is the straightforward method of configuration of a large array of memory cells, and largely the most used when it comes to SRAM cells for most uses. The principle of direct addressing is to group the memory cells in words of N bits and address the M so-formed words individually. The word becomes the smallest accessible element of information of the $L = M \times N$ array of memory latches, as accessing a single bit implies to read the word it belongs to as a whole.

Figure 6-1 schematizes the word addressing scheme of a group of memory cells.

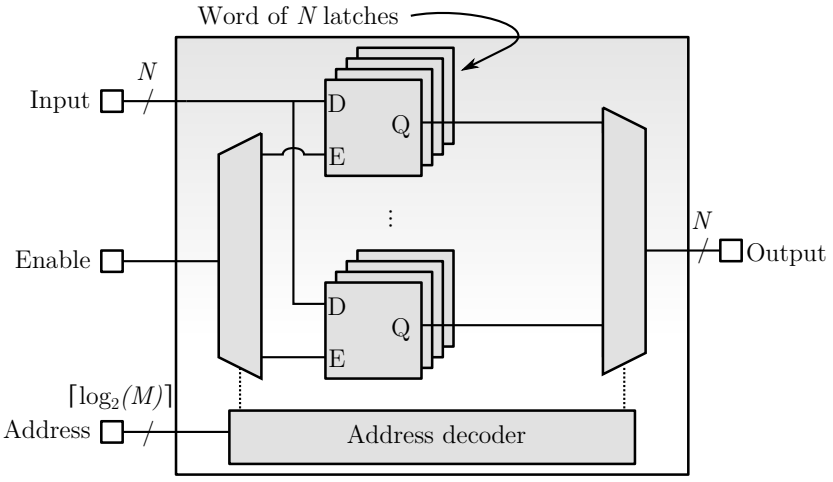


Figure 6-1 – **An array of memory cells accessible with word addressing.** *The word-addressing scheme allows to directly target one of the word of the memory. Single bits however needs to be accessed as part of their word. A combinatorial address decoder is responsible for the generation of the enable signal when the latches are written, and of the output selection when the latches are read.*

Practically, for decoding purposes and implementation constraints, the memory cells are often arranged as an array of W columns and H lines, with $W \times H = M$. In that case, the word address can effectively be split into two parts: the column address and the line address.

Such arrays are generally automatically generated by memory generators shipped with the target technology node design kit, so as to provide the best area and or delay performance. Such generators would be difficult to use in the context of an FPGA configuration memory array, because the memory cells are spread over the architecture to meet the logic elements configurability needs, not the other way around.

At the scale of a whole FPGA device, the impediment of a single chip-wide word-addressed memory comes from the amount of combinatorial logic needed to implement the column and line address decoders, which will require a lot of area. A workaround to reduce the amount of decoding needed would be to increase the word size N so as to decrease the array size $W \times H$, but doing this increases the cost of larger bus sizes needed to reach each words.

From a speed point of view, the frequency of configuration depends on the maximum delay T_{decode} of the line and column decoders, which usually means that bigger memories will run at a slightly smaller maximum speed.

1.1-2 Scan-path: serial loading

A scan-path organization of memory cells relies on the daisy-chaining of the cells, one after another. The name comes from circuit debugging techniques where

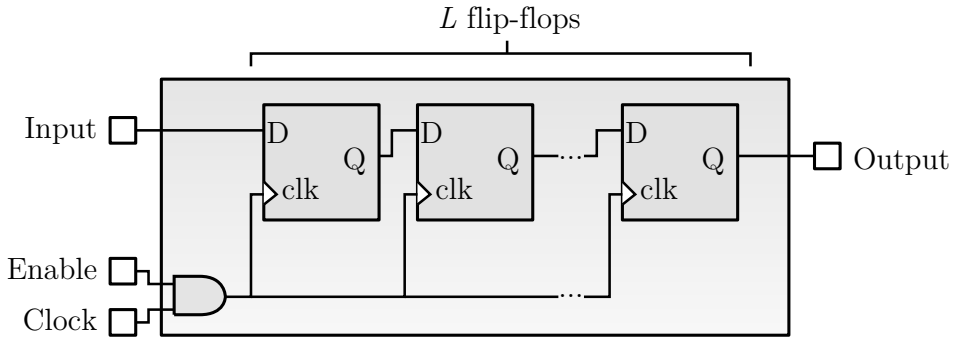


Figure 6-2 – **Memory cells daisy-chained to form a scan-path.** *The daisy chaining of the memory cells together allows to eliminate the area occupied by the address decoding of an addressed memory. On the other hand, since flip-flops must be used to provide the sensitivity to clock edges, the area of the memory cells themselves is approximately* $2\times$ *bigger. Such organization of a large memory has a high power density when all the cells switch at the same time, and must thus be carefully optimized for this purpose.*

register flip-flops can be configured, in debug mode, as a long shift-register which can be inspected at runtime by clocking out all the data to a debug output while the chip is paused. Such memory organization only uses a single bit input to serially insert the data into the L bit long shift-register.

Figure 6-2 illustrates a shift-register of a group of memory cells forming a configuration scan-path.

In comparison to a word-addressed memory, the scan-path allows eliminating the area occupied by the line and column decoders, since no decoding is needed in that case. On the other hand, the shift-register scheme only works if all the cells are loading their data on a signal front event, hence it uses flip-flops instead of latches cells. The flip-flop requires more transistors than a latch, which increases the area occupancy linearly with the length L of the scan-path.

Similarly, the insertion of data into a shift-register can be made at high frequencies, since the critical path of the register is the longest net from one flip-flop output to the next flip-flop input. However, the distribution of the clock across a Very Large Scale Integration (VLSI) circuit is prone to process variations, inducing clock jitter which reduces the maximum operating frequency of the scan-path in practice.

Although a scan-path organization of the memory is promising so far in terms of area and operating frequency, its biggest problem comes from its power consumption. While the transistor activity in a word-addressed memory will be concentrated in the decoder and the configured word, the shift-register will shift its L flip-flops at every bit insertion. Considering the worst case where one out of two flip-flops is set to a logic one, it means that all flip-flops will switch their state at each clock cycle, with a total dynamic power of $L \times P_{switch}$ watts, where P_{switch} is the dynamic power of a single flip-flop.

For large values of L, the tremendous power consumption of a scan-path implies

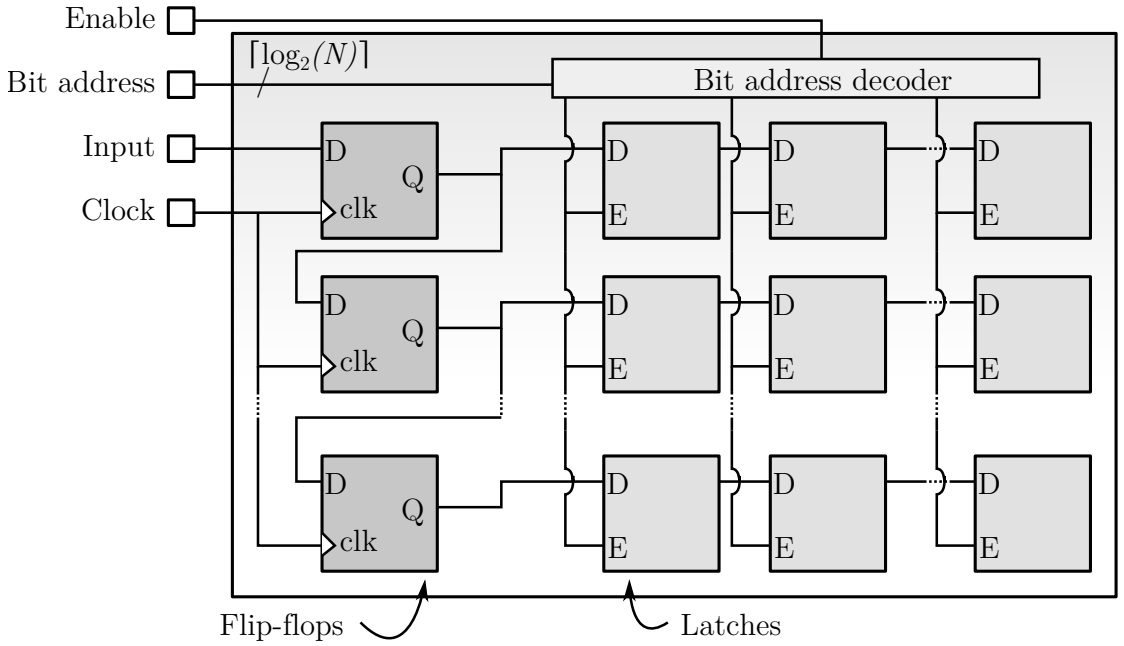


Figure 6-3 – **A hybrid scheme between a scan-path and a word-addressed memory [RGM01].** *The Xilinx patent provides an interesting hybrid between an addressed memory and a scan-path. In this alternate memory organization, the configuration data of all bits i of the N bits latch words is input at once in a scan-path and then programmed in their corresponding latch. Such organization provides an interesting reduction of the overall area over a standard scan-path, along with a really small address decoder in comparison to an addressed memory.*

that its implementation must be done with other power conservative means, e.g. quasi-adiabatic flip-flops [HXX05] or bucket-brigade devices [ST69]. Another way of reducing the overall switching power of the scan-path is to split it into multiple smaller $\frac{L}{n}$ registers.

1.1-3 Hybrid loading

Hybrid loading is a form of configuration memory access which conjugates the speed benefits of a scan-path and the organization of a word-addressed memory. This architecture, patented by Xilinx [RGM01], uses N bits words and a scan-path of $\frac{L}{N}$ elements to configure the words, with L being the total number or bits in the configuration memory. A one-of-N decoder selects which bit is being written in each word when the scan-path is loaded into the SRAM cells.

Figure 6-3 shows the organization of such a memory. In essence, the words are configured by first configuring the decoder to disable all paths from the scan-path flip-flops to the memory latches. Then, the configuration of all bits i of each word is sequentially inserted into the scan-path. Once the data is ready, the decoder is set to enable the links from the flip-flops to the memory latches, to configure them. The process is repeated for each of the N bits of the words.

Overall, the instant power to configure all L bits of the hybrid memory is less than an equivalent scan-path of L elements. Indeed, the switching power required for the configuration of the L bits of the scan-path is in the order of $L^2 \times P_{switch}$, where P_{switch} is the dynamic power of a single flip-flop, while the hybrid memory only requires $N \times \left(\frac{L}{N}\right)^2 \times P_{switch} = \frac{L^2}{N} \times P_{switch}$ watts, effectively dividing the dynamic power consumption by N for the scan-path part.

Additionally, as the length of the shift-register is reduced in comparison to a full scan-path implementation, the area occupancy also decreases. The area occupancy of a scan-path of L elements is in the order of $L \times A_{FF}$, where A_{FF} is the area of a single flip-flop. The equivalent hybrid memory with words of N bits occupies an area in the order of $\frac{L}{N} \times A_{FF} + L \times A_{latch}$, where A_{FF} is as previously defined and A_{latch} is the area occupied by a latch memory cell. As the implementation of a flip-flop is internally based on two latches plus some glue logic, $A_{FF} \simeq 2 \times A_{latch}$. This approximation leads to the conclusion that any word size $N \geq 2$ produces a memory occupying less area than an equivalent L long scan-path, disregarding the one-of-N decoder power.

1.2 Multi context configuration memory

As far as dynamic reconfiguration of a running logic fabric is concerned, it often involves to shut down the part of the device which is going to be reconfigured. Indeed, a configuration memory using only one active layer of configuration will only progressively load the newly programmed data in its target partition, which creates side-effects as the old data is kept in memory. Regarding the application dynamicity, it creates a downtime of parts of the logic fabric for as long as the newly configured partial bit-stream lasts.

If the architecture dynamicity is the most valued performance criteria, it is desirable to back up the configuration memory with a second active layer of memory latches, as shown in Figure 6-4 so that the load of a hardware task does not disturb active portions of the logic fabric.

Figure 6-5b shows the temporal evolution of the double-layer memory during a reconfiguration. The first layer, the configuration layer, receives the bit-stream data, either sequentially, by word-addressing, or both. During the configuration time, the second memory layer, the active layer, still holds the configuration data of the preceding task which can still operate normally. Once the data is correctly loaded in the configuration layer, the `load` signal is asserted to load the configuration data into the active layer. In contrast, Figure 6-5a presents the same scenario on a single-layer configuration memory, and shows the downtime associated with each subsequent configuration of the active memory layer.

An additional memory layer increases the area occupancy of the configuration memory by at least $L \times A_{latch}$ for a L-bit wide memory, with A_{latch} being the area of a single latch. For a word-addressed memory, it effectively doubles the memory area, without increasing the decoding logic area. The area increase of a scan-path

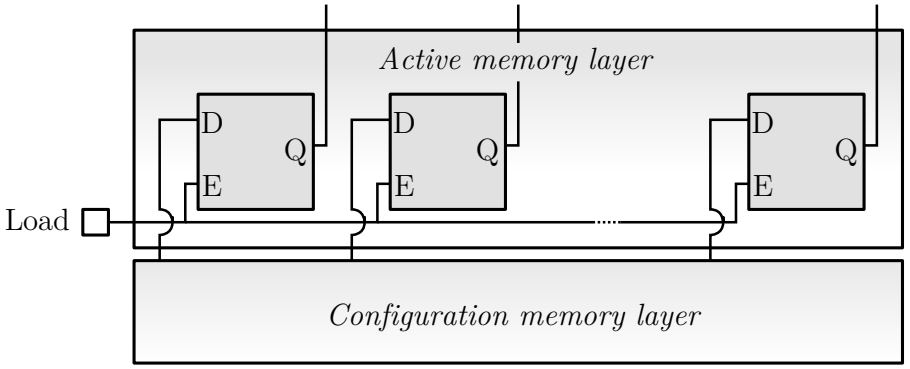
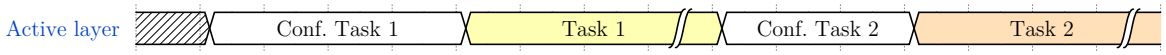
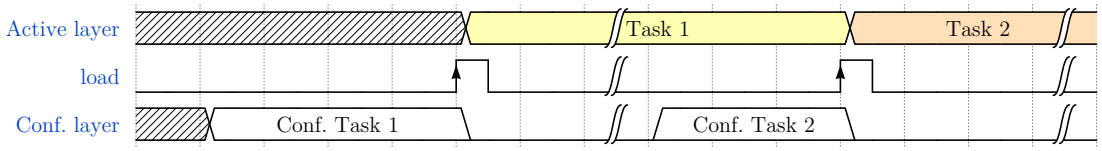


Figure 6-4 – **An additional latch for each memory cell separates the configuration memory from the active memory.** *An additional memory layer composed of latches allows increasing the configuration flexibility as the configuration layer is dissociated from the active memory layer. In this case, the insertion of a partial bit-stream at a specific location of the configuration memory does not disturb the currently running configuration on the active layer, at the cost of an increase of the memory area.*



(a) **Evolution of a single-layer memory with the successive loading of two different tasks**



(b) **Evolution of a double-layer memory with the successive loading of two different tasks**

Figure 6-5 – **The double layer memory increases the dynamicity of the architecture.**

or a hybrid memory is mitigated by the already important area occupancy of the shift-register in both cases.

1.3 Summary of memory organizations

Table 6-1 summarizes the three memory organization schemes detailed in Section 1.1 on multiple criterion. Their area footprint, dynamic power consumption, programming speed and flexibility are evaluated respectively to each other.

The flexibility metric is a subjective evaluation of the adequacy of the memory organizations with regard to the constraints of a reconfigurable FPGA architecture, both from a layout and a partial reconfiguration standpoint. As such, the word addressing scheme is pretty efficient on most evaluation criterion, but fails to meet the needs of flexibility, it is hardly scalable to a whole chip with other layout constraints such as an FPGA. On the other hand, the scan-path and hybrid organizations can easily be scaled to an entire device since each element of the shift-register is only

Table 6-1 – **Comparison of memory organization in terms of area, power consumption, speed and flexibility**

Organization	Area	Dyn. power	Speed	Flexibility
Word addressing	+	+	+	-
Scan-path	-	–	++	-
Hybrid	++	+	+	-

dependent from its preceding and following cells, although targeting a specific configurable zone of the memory is impossible.

Overall, the hybrid organization offers the best trade-off between all the evaluated metrics. The absence of complex decoding and its shortened shift-register makes it more area and power efficient than other memories. Its main drawback comes from the reduced partial-reconfiguration flexibility of the shift-register. The next section presents enhancements to the hybrid memory organization scheme to notably enhance its flexibility.

2 Enhanced scan-path

The proposed technique of an enhanced scan-path architecture allows enclosing a specific portion of the logic fabric for dynamic reconfiguration at runtime. It is a major break-up from the configuration memories of modern FPGA devices where the reconfigurable partitions are defined offline, at design time, and must follow a fixed arrangement of the data. The configuration bits in an FPGA are grouped together in a *frame*, which encloses a bounded area of logic elements and interconnect of the chip and usually spans a few dozen to hundreds of configuration bits. The frames are sequentially programmable and arranged in a fixed order: e.g. in Xilinx devices they are arranged in columns. Although recent devices offer more possibilities for defining a PRR than older architectures which enforced column-wise programmation, they are still limited by a static partitioning which cannot evolve at runtime.

In a statically partitioned logic fabric, the size and geometry of PRR are fixed in advance, either because the configuration memory organization does not allow for anything else than the entire device to be configured, or because PRRs have been defined during the design of the application. In either case, the application must adapt to the partitioning by providing partial bit-streams adapted to the said partitions.

The proposed solution circumvents the static partitioning and provides a way to dynamically define the configurable region while the application runs, to increase the placement flexibility of position-independent bit-streams. The aim of this method is to enable the placement of a $w \times h$ hardware task at a specific position (x, y) on the $W \times H$ configurable logic fabric, hence targeting a specific portion of the configuration memory as depicted in Figure 6-6. We achieve this by the introduc-

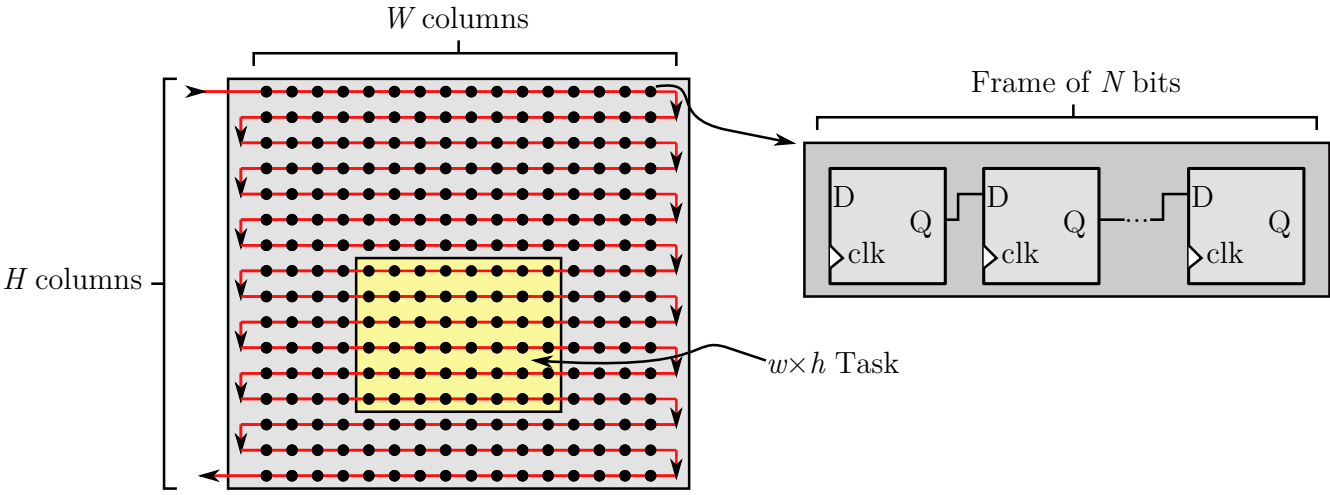


Figure 6-6 – **Basic organization of a standard of hybrid scan-path over the programmable device.** *The static scan-path traverses the configuration memory in a serpentine way, line by line. The considered smallest units of configuration are frames of N configuration bits.*

tion of routing elements in the scan-path, which allows for the configuration of the shift-register path. At runtime, the load of a hardware task configuration data is performed in two steps. A first step configures the path of the shift-register to delimit the partial-reconfiguration region and shortcut the shift-register to this location. The second step is the insertion of the configuration data, in the contrived shift-register.

In contrast to existing static partitioning schemes, the runtime dynamic partitioning method detailed in this section allows building over the previously defined concepts of position-independent Virtual Bit-Streams and of the architecture enhancements allowing the heterogeneous blocks positions to be disregarded.

2.1 Organization of the configuration memory

This configuration memory architecture is based on a scan-path organization, either classic or hybrid as defined in Section 1.1, as illustrated in Figure 6-6. The smallest configuration unit of the path, represented as dots in the figure, is not considered as a single bit in the case of a standard scan-path, but rather as multiple bits grouped together in frames of N bits. For a hybrid implementation, a word size equal to N or one its divisor is similarly considered.

The geometry of the configuration memory is defined as an array of $W \times H$ frames, W being the number of frames per line of the configuration memory, and H the number of lines. The top left hand corner frame is defined as the input of the scan-path. Each frame ending a line of the array is tied to the frame immediately below it, which creates a serpentine shift-register spanning the whole configuration memory of $W \times H \times N$ bits.

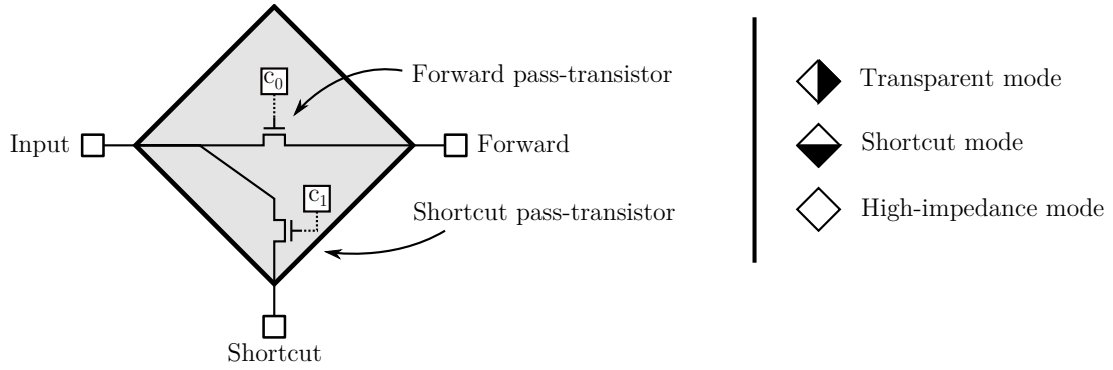


Figure 6-7 – **Configuration memory routing element.** *The routing element provides the shortcut capabilities of the configurable scan-path shift-register. Two pass transistors allow to either pass the configuration signal to the next frame of the shift-register, or to shortcut it to the routing element immediately below. The routing element can operate on three different modes, depending on where to forward the configuration data.*

The aim of the runtime dynamic partitioning is to ease the placement of tasks of $w \times h$ frames, with w the task width and h its height.

2.1-1 Configuration routing element

The configurability of the configuration scan-path is brought by a configuration routing element placed at regular intervals over the shift-register. The routing element, pictured in Figure 6-7 allows the modification of the bit-stream data path and adds shortcuts within the shift-register to jump over series of flip-flops. The configuration of the memory organization has two benefits: first, a part of the shift-register can be delimited for the incoming task data, and second, the path from the shift-register input to the actual location of the desired task placement is reduced.

The routing element operates on three different modes:

- **Transparent**: in this mode the routing element forwards the data to the next element of the shift-register following the natural order of the scan-path.
- **Shortcut**: the shortcut mode redirects the shift-register data to the routing element immediately below it in the array. From there, depending on the next routing element configuration, the data will either follow the shift-register order or be redirected again to another routing element.
- **High-impedance**: the routing element can be disabled to ignore any incoming data. This effectively cuts parts of the shift-register to end it and prevents unnecessary shifts.

Two configuration bits c_0 and c_1 are thus required to set the routing element mode. The state of c_0 dictates the forward pass-transistor mode (i.e. the transparent mode), while c_1 defines the shortcut pass-transistor mode. Both pass-transistors

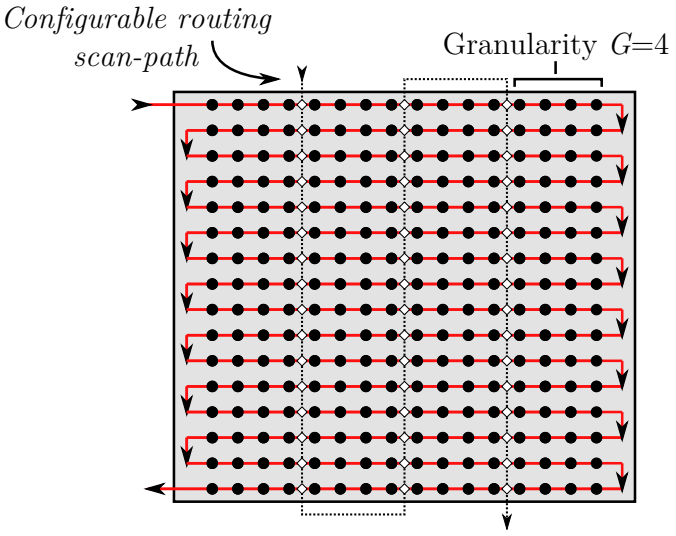


Figure 6-8 – **A configurable shift-register.** *The configurable shift-register uses the routing element capabilities to shortcut or forward the configuration signal. The routing elements are spaced by G frames to allow for a fine or gross placement, tasks must however be sized accordingly. The configurable routing elements are effectively chained together in a second scan-path which needs to be programmed prior to the insertion of the task data.*

cannot be enabled at the same time, it would otherwise create a bit-stream duplication over two different paths.

The routing element adds a delay $t_{routing}$ between two frames of the shift-register because of the pass-transistors. This delay increases the critical-path delay of the shift-register and therefore slightly reduces the maximum shifting frequency of the entire scan-path.

2.1-2 Configuration path

The configuration path is made of all the configurable routing elements put together in a chain. This organization adds a second scan-path to the configuration memory, in addition to the actual shift-register of the memory, albeit the former one is shorter than the latter. The routing elements are placed across the configuration memory scan-path as illustrated in Figure 6-8, forming an array of routing elements capable of forwarding the data or bypassing parts of the shift-register.

The granularity of the array of routing elements if defined as G, the number of frames between two routing elements horizontally. For delimitation purposes, the routing elements are however present on each line of the memory array, as shown in Figure 6-8.

The configuration path contains $\frac{W}{G} \times H$ routing elements, serially chained, which form a $\frac{W}{G} \times H \times 2$ bits long scan-path for the overall device.

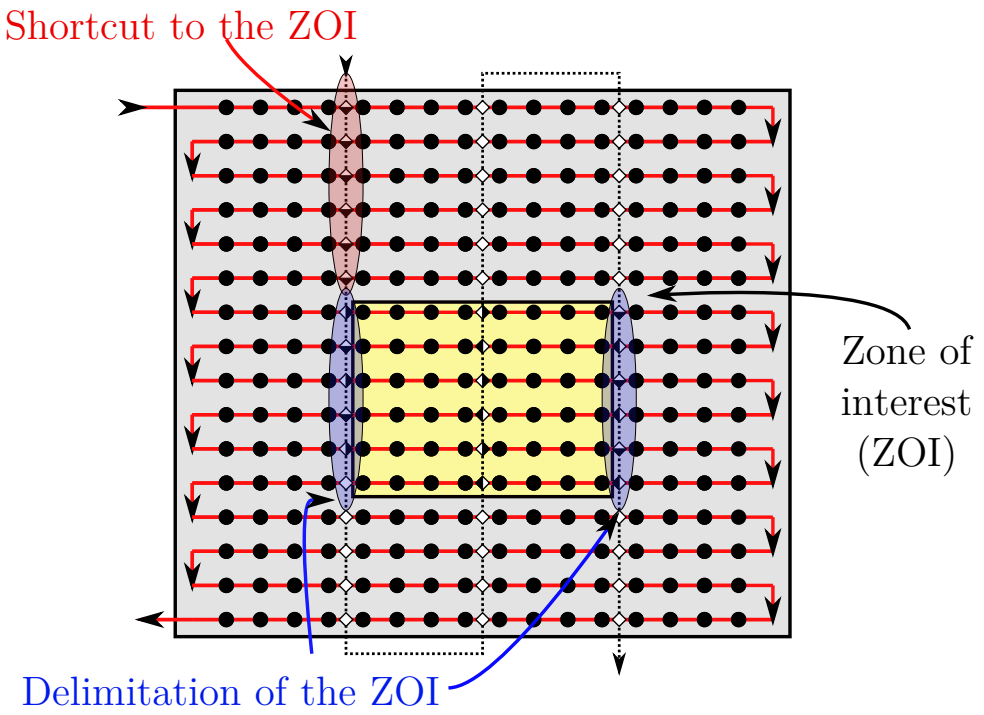


Figure 6-9 – **Usage example of the configurable shift-register with an 8×4 task.** *The configurable shift-register is used to shortcut and delimit the configuration data to a specific zone of interest. The first lines of the configuration memory are shortcut to prevent unnecessary shifts of the scan-path, while the left and right border of the zone of interest are defined using the routing element forward and shortcut features.*

2.2 Runtime dynamic partitioning

The usage of the configurable scan-path enables the runtime dynamic partitioning of the configuration memory, depending on the application needs to load tasks of various geometry. Figure 6-9 illustrates such use of the configurable scan-path with an 8×4 hardware task placed at position $(4, 6)$. The *zone of interest* (ZOI) is formed by the 8×4 rectangular area delimiting the frames that will be configured with the task data.

The proper configuration of the task necessitates two different steps involving the configuration of the scan-path and the configuration of the zone of interest. Using the configurable scan-path has implications on the maximum frequency of the shift-register and on the number of clock cycles required to push the task data into its final location.

2.2-1 Configuring the configuration path

As every routing element can be configured in three different ways, the preliminary step to load a hardware task in the configuration memory is to configure the

configuration path. This pre-configuration step has two goals. First, the beginning of the shift-register, up to the portion of interest, needs to be shortcut to allow for a faster loading of the considered hardware task to its final position. Although this step could be omitted and the data shifted for more cycles into the register, it brings substantial improvements on the configuration delay. The second goal of the pre-configuration step is to delimit the zone of interest in the shift-register. This delimitation prevents the need to insert dummy data between lines of configuration data which would otherwise be required with a non-configurable scan-path so as to align the configuration data with the zone of interest.

2.2-2 Overhead and delay considerations

The enhanced capabilities of the runtime dynamic partitioning increase the placement capabilities, but also comes at the cost of overheads, in terms of additional padding frames that need to be inserted into the configuration memory to "push" the newly placed data to the zone of interest and in terms of increased critical path delay.

Since the routing elements allow to shortcut the shift register path to the next line, they cause an additional delay within the configuration path which linearly depends on the maximum number of consecutive configurable elements used to shortcut the shift-register to the zone of interest. This maximum delay is a function of the target position (x, y) of insertion in the configuration memory, with x being the horizontal position in number of frames and y the vertical position from the topmost line of the memory array. The insertion granularity depends on the configurable routing element granularity G, since it is impossible to target a zone of interest smaller than G horizontally, x must thus be chosen so that $x \mod G = 0$, i.e $x = k_x G, k_x \in \mathbb{Z}$. Similarly, to meet the constraints of the reconfigurable shift-register, the task width w must also be an integer multiple of the granularity G: $w = k_y G, k_y \in \mathbb{Z}$. There is no constraints on the vertical placement since routing elements are present on every line of the configuration memory. Under these conditions, the critical path delay t_{crit} depends on

$$t_{crit} = (y + 1) \times t_{switch} \tag{6-1}$$

since any placement on $y \geq 1$ necessarily implies to cross at least two routing elements vertically to shortcut the unnecessary first lines of the configuration memory up to the zone of interest, although the shortcut elements may be pipelined to reduce this impact on critical path delay.

Moreover, some additional clock cycles are required to get the task data to be aligned with the zone of interest, depending on the horizontal placement of the task. The overhead $N_{overhead}$ can be defined as

$$N_{overhead} = x + (y \mod 2) \times G \tag{6-2}$$

frames, since the traversal direction of the line is not identical for odd and even lines.

This overhead is smaller than what would be necessary to accommodate a non-reconfigurable shift-register in order to maintain the task geometry and correctly align it with its target location. In this case, a task occupying the last line of the configuration memory would have needed to be pushed through the $H - 1$ other lines, while the configurable shortcuts avoid that, at the cost of an increase of the scan-path critical delay.

3 Conclusion

This chapter introduced a configuration memory enhancement to ease the placement of hardware task in a lightweight scan-path organization of the configuration memory. The proposed method allows for a significant reduction of the time needed to serially insert a task to its final position, at the cost of a small frequency diminution in comparison to a static shift-register scan-path.

A trade-off should be made between the task placement flexibility (i.e. the granularity parameter G of the enhanced configuration memory) and the cost of the additional resources. The granularity should be set accordingly to the logic fabric features such as an irreducible width of a minimum partition to cope with the architecture heterogeneity, as discussed in Chapter 5.

With the techniques and features developed throughout this work at the Computer-Aided Design (CAD), architecture and configuration level, a number of parameters can be explored and combined to offer additional flexibility to an FPGA architecture, both from a conceptual and an architectural point-of-view. The full dynamicity of a platform as explored in this thesis is nowadays not deeply explored, but recent initiatives have been conducted in this domain. The FlexTiles project presented in the following chapter is one of these initiatives which aims to provide a fully reconfigurable and adaptive manycore platform bundled in a 3-D stacked chip, in which some of the work of this thesis has been implemented to some extent.

Chapter SEVEN

THE FLEXTILES PLATFORM

Contents

1 Overview of the FlexTiles project **124**
1.1 Global architecture 125
1.2 Programming model 125
1.3 Virtualization layer 126
2 Embedded FPGA accelerators **126**
2.1 Overview of the embedded FPGA 126
2.2 eFPGA architecture 127
2.3 Hardware tasks 128
3 Implementation of a dynamically reconfigurable FPGA **129**
3.1 RTL model 129
3.2 eFPGA testbench 130
4 Conclusion **132**

Abstract

Parts of the work of this thesis has been used into the European Seventh Framework Programme (FP7) project FlexTiles [Jan+15], aiming to develop a dynamically reconfigurable heterogeneous many-core platform. The heterogeneity of the platform is brought among other accelerators by an embedded FPGA logic fabric 3D stacked to a many-core circuit, as illustrated in Figure 7-1. This embedded FPGA (eFPGA) exhibits runtime hardware reconfiguration features. The FlexTiles project ended in Spring 2015 and proposed a development board, a software Open Virtual Platform (OVP) simulator, the adequate tool-chain, an operating system, and software libraries.

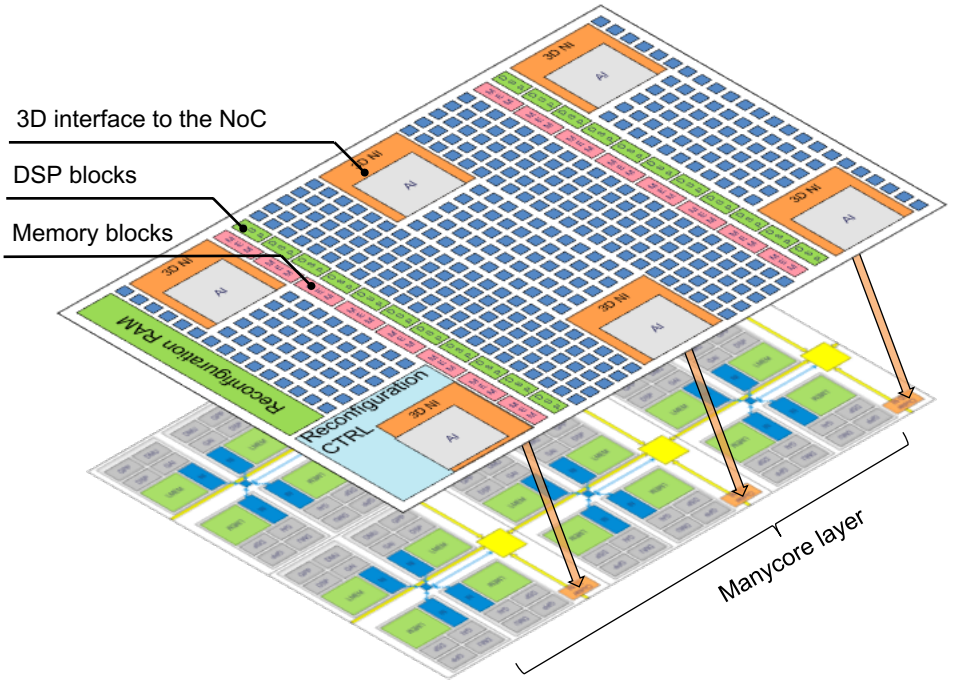


Figure 7-1 – **The 3-D stacked FlexTiles platform.** *The FlexTiles platform is a 3-D stacked heterogeneous architecture comprising GPP, DSP and hardware tasks loaded onto an embedded FPGA (eFPGA). The communication between these elements is made through a NoC. Communications between the eFPGA and the manycore layer is realized with 3-D links to the NoC with dedicated interfaces in the FPGA logic fabric.*

1 Overview of the FlexTiles project

Heterogeneous many-core systems are promising solutions to deal with complex and increasingly dynamic applications. Indeed, power-hungry applications found in modern handheld computers must cope with performance and energy constraints [Lem+12]. Their accompanying embedded hardware must provide high performance with a low complexity, while at the same time keeping a low power budget. The multi-core era has been proven useful in those case by providing a high performance/power ratio, in particular in embedded systems, and many-core systems are the next step towards a better efficiency. The industry is however reluctant to adopt many-core designs, primarily by fear of increasing development costs and not being able to reuse legacy C code which have been heavily tested for years.

The FlexTiles project is a 3-years long European FP7 project which ended in 2015. The aim of the FlexTiles project was to propose an energy-efficient and heterogeneous many-core 3-D stacked architecture with self-adaptation features based on flexible tiles, illustrated in Figure 7-1, both from the design and the programming standpoints [Jan+15].

In this chapter we only give an overview of the FlexTiles project, an in particular on the design of its embedded FPGA architecture which relies on features proposed

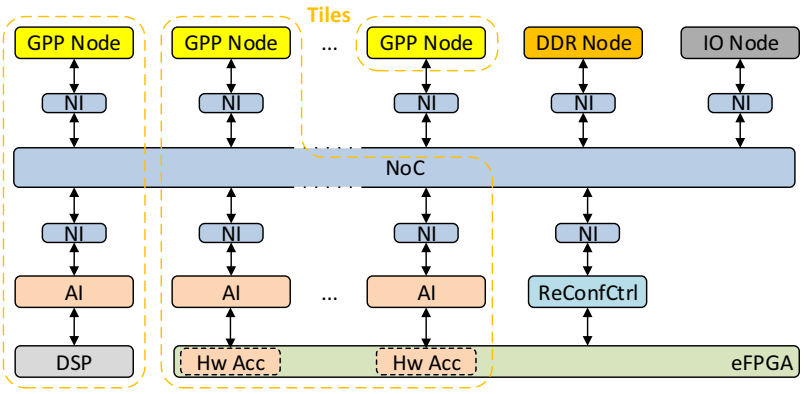


Figure 7-2 – **Overview of the FlexTiles platform heterogeneous many-core hardware architecture [Jan+15].**

as part of this thesis work. More information can be found on the website dedicated to the project [Fle15].

1.1 Global architecture

The Flextiles architecture is based on a set of heterogeneous nodes among General Purpose Processors (GPPs), Digital Signal Processors (DSPs) or hardware accelerators loaded on an embedded FPGA, the *eFPGA*. A *tile* in the sense of the FlexTiles platform is an association between a master GPP node and one or more slave node. Additionally, the platform features Double Data Rate (DDR) memory controllers and input/output elements. All these nodes are communicating together using a Network on Chip (NoC) backbone, as illustrated in Figure 7-2.

The communication homogeneity between the various nodes of the platform is ensured by a Network Interface (NI) which provides the abstraction interface between the nodes and the underlying NoC. Additionally, the data exchanges between the GPPs and their slave accelerator nodes are homogenized by Accelerator Interfaces (AIs), designed to implement the master/slave execution model.

1.2 Programming model

Applications targeting the Flextiles platform are defined as sets of static *clusters*. Each of these clusters is described using Synchronous Data Flow (SDF) or Cyclo-Static Data Flow (CSDF) computation models. Within the data-flows, the tokens consumers/producers are called *actors*. The actors communicate by exchanging tokens through First-In First-Outs (FIFOs) [NMG13].

The clusters of an application are independently designed and optimized using the platform dedicated tools. The design stage of a cluster is responsible for the partitioning, scheduling and mapping of its actors. A partitioning step groups closely related actors together in a *partition*. Each partition is then scheduled in a time

scale, typically to manage an operator pipeline. Eventually, the clusters and their partitions are mapped to their final target tile on the platform.

1.3 Virtualization layer

A software virtualization layer is responsible for the architecture management of the available heterogeneous resources and for its adaptivity [Fer+12]. The system is systematically monitored to ensure that the application performance, the GPP and accelerator workloads, and the overall platform temperature stays within acceptable boundaries. A diagnostic phase allows taking various actions in function of the detected issues of the platform. It also draws conclusions regarding the actions to execute so as to counteract the problem. The actions include the decrease or increase of the processor frequency to meet the performance demand or power budget, the migration of a task if a GPP is overloaded, or triggering a defragmentation of the hardware resources to cope with the allocation of larger hardware tasks on the embedded FPGA logic fabric.

Additionally, each master GPP runs a custom operating system which locally handles the multi-threading of each processor as well as the scheduling and power management of the running clusters.

2 Embedded FPGA accelerators

A key point of the FlexTiles platform is to feature an embedded FPGA, the eFPGA, acting as a generic logic fabric on which dedicated accelerators (i.e. hardware tasks) are loaded at runtime to meet the performance needs of a data-flow application. Some of the runtime hardware reconfiguration features involved in the work of this thesis have been put to use in this context, to support the flexible runtime loading of hardware tasks.

2.1 Overview of the embedded FPGA

The eFPGA has a few requirements which are unusual with regards to the conventional FPGA architectures of the market. First, the logic fabric itself has no direct interactions with the outside world. The input and output communications of the hardware tasks loaded on the eFPGA logic fabric are done through the NoC of the FlexTiles platform, using the AI defined earlier. These AIs feature a set of data and configuration channels which are then translated by an underlying NI to packets sent over the NoC. From the logic fabric standpoint, the AI is logically seen as one accelerator of the FPGA among others. Any hardware task running on the eFPGA requires a connection to one AI, and the number and availability of these hard blocks is thus crucial for the application dynamicity. The AI and their asso-

Table 7-1 – **Synthesis results of the architectural elements of the eFPGA on 28nm and 65nm technology nodes**

Block	Area 65nm (μm^2)	Area 28nm (μm^2)	w.r.t. CLB
CLB	9,929	3,309	1×
Arith. Acc.	13,055	4351	1.31×
RAM block	21,134	7044†	2.13×
AI	354,617	118,205	35.71×

†: Projection only

ciated 3DNI are spread over the logic fabric, rather than distributed in a I/O ring surrounding the logic fabric.

In addition to the logic fabric, the eFPGA hosts a reconfiguration controller which implements the features presented in Chapter 4. The controller handles both the configuration of the logic fabric itself and its management from an occupation point of view. As requests from the platform virtualization layer are received on the AI dedicated to the reconfiguration controller, the reconfiguration controller must decide where to place the task and inform the rest of the platform that a defragmentation of the logic fabric should be performed if needed.

A lot of the features implemented in the eFPGA are not the top priority of modern FPGA architectures, for the reasons precedently evoked. Most notably, the eFPGA logic fabric has not been designed to host a single design at runtime but rather a variety of accelerated tasks which work in conjunction with the platform to provide a performance gain of specific part of an application. Therefore, the design choices made during the development of the eFPGA were driven by the architecture flexibility, rather than the maximum achievable performance. As an example, a double context memory, as detailed in Chapter 2, was implemented on the eFPGA, at the cost of the additional area consumed by the memory.

2.2 eFPGA architecture

The eFPGA logic fabric, detailed in Appendix A, is built around an island-style architecture. The soft logic elements of the architecture are clusters of four 6-LUTs and their associated local crossbar interconnect. Each of these CLB routes 16 inputs and 4 outputs on the interconnection network.

In addition to the soft logic elements, hard blocks provide a substantial acceleration of various features in the architecture. A configurable arithmetic accelerator with a 18×18 multiplier and a 48-bit Arithmetic and Logic Unit (ALU) provides the data computing performances of the logic fabric. 16kb Random Access Memory (RAM) blocks add dynamic memory storage to the architecture. Lastly, AIs brings the communication properties of the eFPGA for the hardware tasks to exchange data with the rest of the platform.

Each of the elements of the logic fabric has been developed in Verilog and synthesized to both 28nm and 65nm technology nodes from STMicroelectronics to obtain realistic data of their respective sizes. These results were used to actually determine the number of each hard block which will populate the logic fabric through a design space exploration process. The synthesis results are presented in Table 7-1 and take into account the respective interconnect (the connection block) associated with their inputs and outputs. The last column of the table presents the ratio of the areas of the blocks respectively to the smallest unit of the architecture, the CLB. Due to the lack of access to a memory generator for the 28nm node, the area result of the RAM block in 28nm is only extrapolated and scaled from the 65nm synthesis. These implementation areas were compared to the size of the many-core chip already determined in order to provide a final layout of the eFPGA logic fabric.

One of the constraints of the final layout is tied to the 3-D stacking technology process chosen for the FlexTiles platform. Indeed, as the many-core chip and the embedded FPGA circuit are stacked on top of each other, the 3-D interfaces from one chip to the other must be aligned and thus fixed at specific positions. On the eFPGA layer, the AIs are in fact 3-D Network Interfaces (3DNI), and comprises the actual AI with which the logic fabric communicate, the NI realizing the translation to network packets, and the area occupied by the Through-Silicon Vias (TSVs) (for face-to-back 3-D stacking) or micro-bumps (for face-to-face 3-D stacking) realizing the link between one chip to another. For this reason, the AIs are the biggest elements of the logic fabric.

The final architecture layout of the eFPGA comprises:

- 9 AIs
- 42 RAM blocks of 16kb
- 210 arithmetic accelerators
- 7,224 CLBs (i.e. ≈29,000 LUTs)

The size of the two dies of the FlexTiles 3-D stacked architecture has been fixed to $5,500 \times 4,900 \mu m$ (i.e. $26.95 mm^2$) in a Complementary Metal-Oxyde Semiconductor (CMOS) bulk 28nm technology from STMicroelectronics, constrained by the manycore layer.

2.3 Hardware tasks

The description of the hardware tasks used on the embedded FPGA of the FlexTiles platform follows the definition of the Virtual Bit-Streams thoroughly detailed in Chapter 3. The Virtual Bit-Streams are stored in memory and placed at runtime on demand of the virtualization layer to the reconfiguration controller of the eFPGA.

The bit-stream generation back-end *vbsgen* has been integrated to the design flow of the FlexTiles platform. It processes the partitions of an application cluster

which have been mapped as hardware accelerators in the platform.

Spatial and temporal scheduling techniques to be integrated in the reconfiguration controller were not investigated as part of the work of this thesis. They are required so as to fully manage the placement of hardware tasks to be loaded onto the eFPGA logic fabric. Several works were conducted to increase the spatio-temporal scheduling methods in the context of a 3-D stacked architecture. In [KCH14], the authors investigated the case where the communication between one layer and a GPP on the other layer must be considered. The heterogeneity of an architecture for the placement decision was studied in [LCC14].

3 Implementation of a dynamically reconfigurable FPGA

As the fabrication of an actual chip was not the goal of the FlexTiles project, the platform has been validated both in a simulation environment and on a development board based on FPGAs. However, in the case of the validation of the eFPGA logic fabric, the impracticality of simulating an FPGA logic fabric on a physical FPGA chip led to the development of a co-simulated Register Transfer Level (RTL) model as the validation platform of both the reconfiguration controller and the logic fabric itself. The validation details are presented in Appendix B.

3.1 RTL model

In addition to the previously defined complete eFPGA architecture, a smaller layout of the RTL model has been developed to allow for faster simulation time on preliminary tests. The typical compilation time (i.e. generation, elaboration and simulation preparation) of the full architecture is about 1 hour 45 minutes on an Intel Xeon 5650 (2.67GHz) processor due to the high complexity of the interconnect in the RTL model. Moreover, the programmation of a full bit-stream in the configuration path of the reconfigurable platform could take up to 25 minutes, which killed the usefulness of the RTL simulation to validate and track down bugs in the architecture. For these reasons, the RTL model development flow was made compatible with two different FlexTiles architectures: the complete architecture presented above, and a small architecture used for test. The smaller architecture is a scaled-down version of the complete eFPGA model, using only one Accelerator Interface and fewer logic blocks, memories and arithmetic accelerators.

3.1-1 Generation of the RTL model

The RTL model generation process is presented in Figure 7-3. A python script is dedicated to the generation of the layout of the small and regular eFPGA architectures. It uses a simple text file as an input to describe the architecture in a

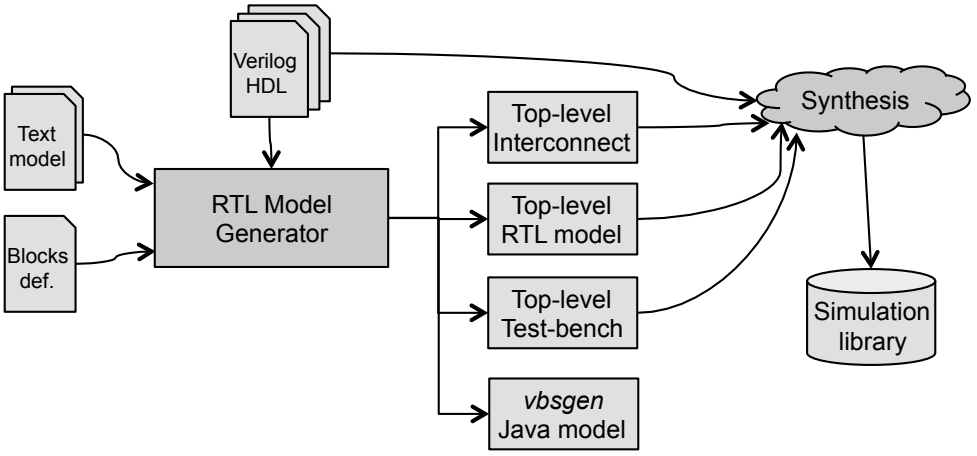


Figure 7-3 – **Development flow of the automatic RTL model generation.**

positional manner. An additional configuration file in the JSON format specifies various metadata about the blocks used in the layout description. The block metadata include: the channel width of the architecture, the Verilog source file of each block from which the I/O signals will be extracted and the repartition of the block I/Os on its boundaries.

Information on the inputs and outputs of the blocks are then extracted from the source Verilog files using comments as markers to split the inputs and outputs into three categories:

- *efpga*: the eFPGA interface, including the clock, reset signal and the scan-chain configuration control signals,
- *routable*: the dynamic input and outputs of the block which needs to be routed on the interconnection network,
- *external*: the signals which do not need to be routed on the interconnection network of the FPGA fabric but are directly interfaced outside of the eFPGA (e.g. the link between the AI and the NI)

The top-level generation script creates: the eFPGA model top-level module, the list of interconnection wires (scan-chain, switchboxes, block I/Os) used in the top-level module, the eFPGA top-level test-bench module and the horizontal, and vertical interconnection blocks of the architecture.

3.2 eFPGA testbench

The dynamic reconfiguration test-bench includes the whole final model of the eFPGA linked to an emulated reconfiguration controller handled by a C library during the simulation runtime. This reconfiguration controller is designed to act as a shell with which the user can interact to control the running simulation at the controller level.

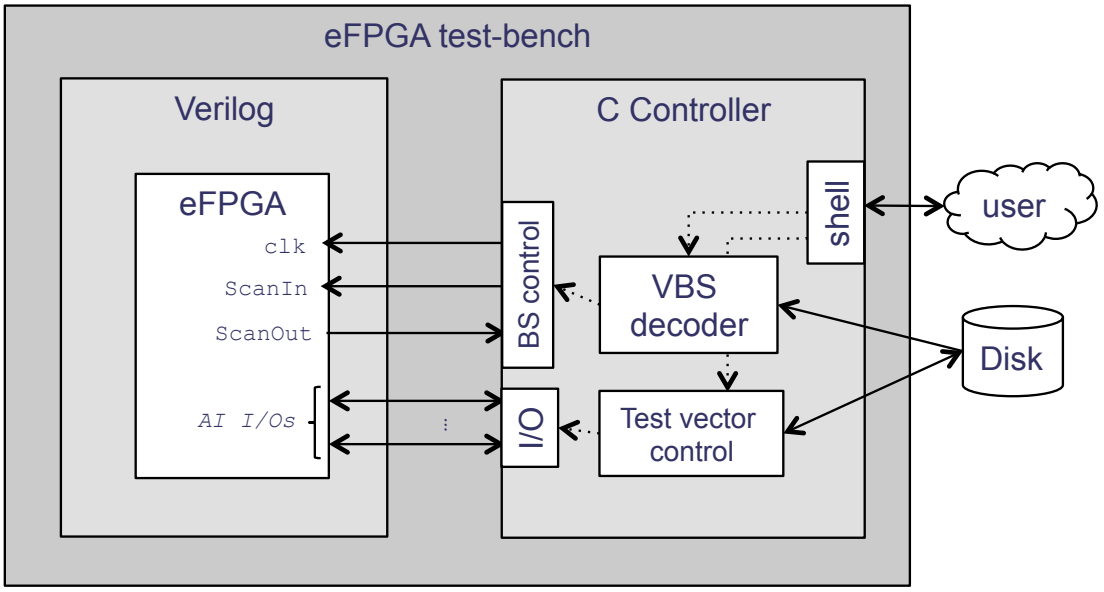


Figure 7-4 – **Implementation of the simulation testbench of the eFPGA.**

3.2-1 Testbench implementation

The Cadence NC-Sim tools used to compile, elaborate and simulate the eFPGA final model support the integration of C callback functions within the Verilog test bench, using an interface framework called Verilog Procedural Interface (VPI). The software implementation of the reconfiguration controller used with the eFPGA RTL model uses this framework to plug into the test-bench and handle the logic fabric control signals, which are normally devoted to the real controller. These functions are compiled into a shared object with a C/C++ compiler. The shared library is then dynamically loaded at runtime by the Cadence simulator NC-Sim. We specify a specific bootstrap function to be called upon loading of the library, which in turns sets up the simulator with the list of callback functions available in the library.

Once the shared library is loaded and bootstrapped by the simulator, the exported callback functions are made available as Verilog code from within the running simulation. The code run within the shared library does not keep the cycle-accurate properties of the Verilog simulation, but the C code controls the Verilog signals through the VPI/PLI interface framework only. This behaviour allows for complex operations in the C code at high-speed, while still keeping the cycle accuracy of the Verilog simulation.

The global overview of the test-bench is presented in Figure 7-4. The interactions with the C controller are realized manually through a shell, to emulate requests emanating from the software virtualization layer of the platform. The inputs of the test bench (the loadable virtual bit-streams and the test vectors) are stored on disk and read at runtime by the emulated controller.

3.2-2 User interaction

As stated earlier, the emulation of the controller in a separate shared object also permits to easily handle user input instead of doing these operations from within the Verilog simulation. The controller, in simulation, thus exhibits a simplified shell over a socket connection, which allows the user to manipulate the dynamically reconfigurable hardware from there. This functionality is set up in lieu of the communication between a GPP and the reconfiguration controller in the final platform. This emulated text interface with the controller shall also enable an offline, simulated, execution of the platform (such as the OVP platform) to communicate with the eFPGA simulation, at the cost of severe delays from the eFPGA simulation because of the cycle-accuracy. The commands implemented in the shell are:

- `load_vbs <file> [x] [y]` : loads a Virtual Bit-Stream file on the logic fabric at position x,y (x and y are optional and defaults to the upper right-hand corner of the fabric).
- `load_bs <file>` : loads a raw bit-stream file on the logic fabric.
- `test <AI id> <input vector> <output vector>` : loads the input vector file vector and sends the data to the Accelerator Interface with identified by the id in the RTL model.
- `conf_switch` : loads the configuration load memory plane into the active configuration memory plane.
- `clk <confclk|clk> <on|off>` : enable or disable the configuration (i.e. the scan-chain) clock or the system clock.
- `sig <signal>` : prints debugging information about a signal (source code location, signal size, value).
- `simtime` : prints the current simulation time
- `exit` : exit the current shell session without interrupting the simulation.
- `stop` : exit the current shell session and ends the simulation.

4 Conclusion

This chapter presented the FlexTiles project and the implication of the work of this thesis within the project, along with its outcomes. A detailed overview of the embedded FPGA used in the platform has been given, and the accompanying testbench has been described.

The FlexTiles project has been the common thread of the innovations proposed in this thesis work. The development of an actual simulable model using the Virtual Bit-Streams principles and CAD flow was an opportunity to put the developed techniques to a proper use. The innovation and development efforts have been put

on the flexible runtime management of hardware tasks of the embedded FPGA layer integrated in the FlexTiles platform, the eFPGA.

CONCLUSION AND PERSPECTIVES

1 Overview

This thesis work focused on a reconfigurable Field-Programmable Gate Array (FPGA) architecture offering runtime hardware reconfiguration capabilities. Enhancements at multiple levels of the hardware circuit and the software tools have been proposed. Chapters 1 and 2 lay an overview of state-of-the-art FPGA architectures and of runtime reconfiguration and routing techniques in these devices.

The third chapter proposes an alternate representation of the configuration bit-stream of FPGAs. Using an abstract view of the target interconnection network, the Virtual Bit-Stream offers a position-independent method to describe a hardware task. We have seen that a specifically designed architecture benefits from the Virtual Bit-Stream technique by exposing advance runtime reconfiguration of these unfinalized bit-streams. The compatible FPGA architecture can dynamically place the tasks using an online controller and decide at runtime of the final position of a hardware task in the form of a Virtual Bit-Stream file. A tool dedicated to the modeling of FPGA architectures and the generation of Virtual Bit-Streams, *vbsgen*, has been developed in Java, totaling $\sim 7,000$ lines of code.

Chapter 4 analyzes the real-time constraints needed to perform the necessary decoding of the alternate configuration bit-stream representation. To cope with the high throughput of reconfiguration in modern FPGA devices, two hardware decoders are explored and detailed. A first decoder, based on an in-memory storage of possible routes in a local macro-cell, is able to decode one route per cycle but is not optimized for large interconnection networks due to its memory footprint. The second decoding algorithm relies on a Finite State-Machine (FSM) tied to the target interconnection network organization and provides a slower decoding with fewer memory requirements. A study of the compression effect of the Virtual Bit-Stream technique has been conducted which showed gains of up to $11\times$ with the finest possible grain of abstraction. The clustering of the smallest units of data in the Virtual Bit-Stream can further increase the average compression ratio of the technique, although the processing power needed for the decoding step increases.

The fifth chapter introduces enhancements of FPGA architectures to lower the

constraints on the placement of hardware tasks relying on hard blocks. Through a separation of the hard blocks routing from the soft logic interconnection network, a higher horizontal flexibility of the task placement have been exhibited with a small impact of the critical path delay. The usage of dedicated long-lines to each hard block input and output allow a given hardware task to realize connections to hard blocks at runtime upon placement of the task, through enhanced switch blocks. Modifications into the Versatile Place-and-Route (VPR) place-and-route software have been implemented to support these architecture modifications and evaluate their delay and routing resources implications.

Chapter 6 studies the problem of inserting hardware tasks configuration bitstreams at specific position in the logic fabric from yet another standpoint. The architecture enhancements proposed in this chapter focuses on modifications of the configuration memory itself to support a dynamic runtime task placement. An overview of common configuration memory organizations is given and details the advantages and drawbacks of three different configuration memories. An enhanced daisy-chained configuration memory based on a shift-register is proposed to provide both area-efficiency and fast configuration insertion at specific locations. A chain of routing elements is inserted into the configuration memory to ensure that the configuration data does not have to traverse the whole length of the configuration shift-register to program the target zone of interest. In the end of this chapter, the delay and shift-register clock frequency consequences of the routing elements are discussed.

Eventually, the seventh chapter presents a European Seventh Framework Programme (FP7) project, FlexTiles, in which some of the techniques discussed in Chapters 3 to 6 are implemented. The main objective of the project is to propose a 3-D stacked manycore architecture with self-adaptation features and its accompanying Computer-Aided Design (CAD) flow and programming model. The architecture notably integrates a dedicated embedded FPGA, the eFPGA, which provides one way of accelerating portions of an application. The Virtual Bit-Stream CAD flow and online reconfiguration controller have been integrated and validated in the context of the FlexTiles project through a complete co-simulated Register Transfer Level (RTL) model of the eFPGA circuit. A smaller, homogeneous version of the eFPGA circuit has been designed up to the chip layout, as shown in Figure 7-5. This placed and routed version of the chip was used to extract detailed area and delay information in the context of the FlexTiles project.

Some of the work done on the eFPGA reconfigurable fabric has been targeted for experiments on low-power controllers for wireless sensor networks in [Tov+14]. Additionally, a three month visit to University of Massachusetts Amherst led to the development of netlist reverse engineering techniques applied to the insertion of hardware cryptographic Trojans into a hardware design, and protections against this kind of attacks [Swi+15].

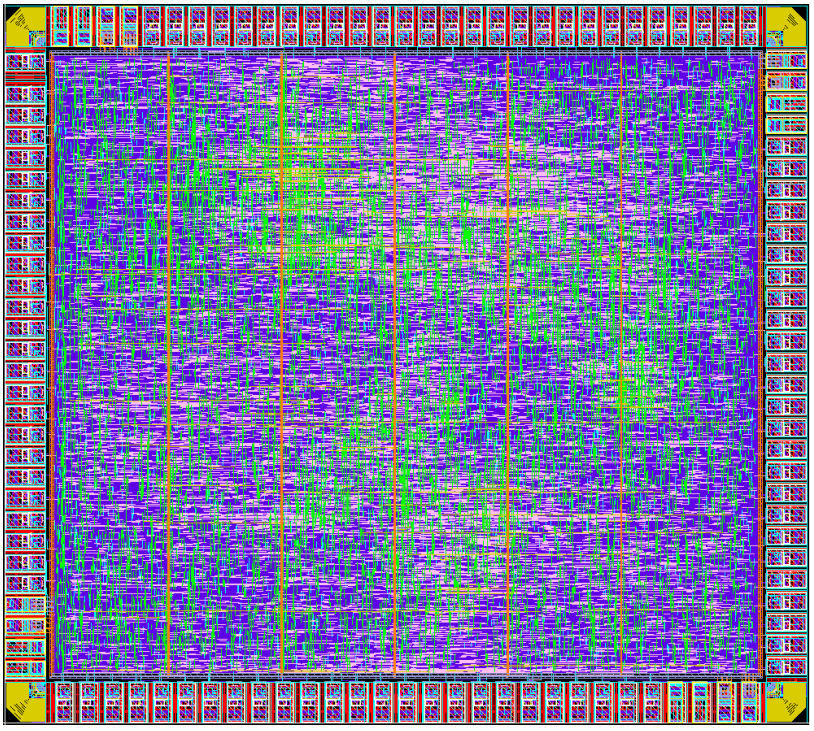


Figure 7-5 – **Layout of a homogeneous version of the eFPGA logic fabric**

2 Perspectives

State-of-the-art chapters 1 and 2 show how vast of an active research domain are reconfigurable architectures, with areas ranging from bit-stream manipulation for existing FPGA devices to coarse-grain heterogeneous architectures. The evolution of the domain is fast-paced, and multiple research opportunities emerge from the work presented in this thesis.

Position-independent bit-streams Chapter 3 addresses most of the concerns of the applicability of the Virtual Bit-Stream techniques to generic architectures to provide a position-independent manner of representing hardware tasks. The abstraction of this alternate configuration bit-stream coding could also be used to support multi-platform Intellectual Propertiess (IPs) which are generated and distributed once for multiple targets to promote interoperability. Enhancements over the proposed representation may be proposed to support this behavior and study its implications at the hardware level and the representation level. Specifically, the current raw coding of soft and hard logic data hampers a real portability of Virtual Bit-Streams to different platforms, but a smarter representation of these data may leverage the full independence of the configuration data from the target hardware. The usage of existing techniques allowing a compression of the logic data as explored in [AL11] may also further increase the compression capabilities of the Virtual Bit-Stream technique.

Additionally, although the current way of handling heterogeneous architectures does not cause an important overhead at the software level (i.e. in the binary coding of the Virtual Bit-Stream), the architectural implications of multiple different macro-cell interconnection networks could be lowered. Due to the actual dependency of the proposed algorithms to the target interconnection network, the implementation of a full decoder for multiple macro-cell compositions may involve duplication of parts of the reconfiguration controller. In spite of the fact that the flexible software implementation of the decoder into the reconfiguration controller would alleviate this problem, the timing implications may not meet the real-time constraints needed for the decoding operation to be transparent with regards to the configuration data loading process. As a general thought, it would be profitable to explore the online decoder algorithm to exhibit more flexibility.

CAD flow The Virtual Bit-Stream design flow has been designed from scratch from the observation that no generic mean of describing an FPGA architecture and its configuration method to actually create configuration bit-streams exists. The CAD flow detailed in Chapter 3 has been extensively developed, tested and used in this context, driven by this thesis work and the FlexTiles project needs. Recent research work [KA15] from the group maintaining the VPR software and the Verilog-To-Routing (VTR) framework brought the possibility of actually generating a hardware description of an FPGA architecture explored within these tools. The new VHDL-generating capabilities of VPR along with its support to the creating of bit-streams, while currently limited to soft-logic elements, enables the possibility to integrate the Virtual Bit-Stream generation capabilities of *vbsgen* into VPR itself. As the most time-consuming parts of *vbsgen* resides in the analysis of the VTR output files, this integration would lead to a faster creation of the bit-streams, along with an increased reliability of the software itself, albeit with less flexibility.

Heterogeneous task loading Although the flexibility outcome of the architecture enhancements detailed in Chapter 5 is beneficial to tackle the heterogeneity problem in highly dynamic FPGA architectures, the increase in routing resources is non-negligible and should be handled with optimized custom designs. For this purpose, performance evaluations of these enhancements should be conducted on a custom design to study the delay impact of the additional routing resources spread over multiple metal layers, which is not actually possible in VPR.

Moreover, the base hypothesis of VPR with regards to the FPGA detailed interconnection network forces the connections to horizontal wires at the top and bottom of a block spanning more than one element of the logic array in a column. For the proposed enhancement increasing the horizontal flexibility of task placement, which relies on horizontal long lines to achieve the routing to hard blocks, it implies that *all* the connections to and from a hard block shall be spread over only two horizontal routing channels. This is very limiting with regards to the evaluation of the delay and routing resources impact, as the routed hard blocks signals are affected by an increased congestion in the occupied horizontal channels. More development in the

routing graph generation of VPR would be needed to address this problem.

On another perspective, the defined enhancements to the architecture detail a very fine grain flexibility of the task placement, although such flexibility is unlikely to be necessary for real-life applications. In conjunction to the grain definition of Chapter 6, the distribution of switch-boxes allowing the connection to the hard blocks long lines may be depleted to lower the increase in area induced by the additional interconnect.

Flexible configuration memory The configurable configuration memory detailed in Chapter 6 offers a flexible way of using a configuration memory based on a shift-register as the backbone of a partially reconfigurable FPGA circuit. However, the enhancements of the configuration memory have not been implemented in a hardware model to evaluate the delay implications of such an organization of the memory with a real technology node. Additional work should be conducted to explore the parameters of the routing elements in the configuration scan-path and compare to similarly optimized shift-register memory.

Long term perspectives As far as the Virtual Bit-Stream technique is concerned, it is applicable to any reconfigurable architecture. The compression brought by the alternative coding of the interconnection network configuration data is an added value, but the main goal of this representation is its independency with regards to its final placement. The work on Virtual Bit-Streams could be extended to different devices such as Coarse-Grained Reconfigurable Architectures (CGRAs). More complex manycore architectures could also be targeted for long term extensions of the Virtual Bit-Stream. Recent Multiprocessor System-on-Chip (MPSoC) reconfigurable architectures such as the Zynq UltraScale+ MPSoC offer a large amount of reconfigurable logic fabric in addition to multiple general-purpose processors. In the context of shared memory MPSoCs, the reconfiguration controller could be implemented as a software manager using specific software decoding algorithms to handle the decoding of hardware tasks and place them on the logic fabric.

The usage of FPGAs has evolved during the last decade into the acceleration of massively parallelizable computing. Multiple FPGA logic fabrics can be easily assembled together into a single appliance to provide flexible hardware acceleration capabilities to computation-heavy processes which were otherwise software-only. Such complex architecture often finds applications in cloud services which often requires power-hungry servers to run complex software systems which could benefit from an FPGA implementation [Cha+13]. A flexible multi-FPGA system could benefit from a higher-level management of the computing tasks running on such system, using the runtime reconfiguration capabilities of the devices. The Virtual Bit-Stream technique and its CAD flow could be generalized to such complex architectures to handle generic hardware tasks at a fraction of the power consumption of an equivalent General Purpose Processor (GPP) based processing and with superior parallelization capabilities.

Part III

Appendices

Appendix A

HARDWARE SPECIFICATIONS OF THE EFPGA

Contents

1 Logic macro-cell **144**
2 Logic blocks **144**
2.1 Complex Logic Block 145
2.2 Arithmetic accelerator 146
2.3 Memory block 150
2.4 Accelerator interface 150
3 Logic fabric organization **151**
3.1 Synthesis results 151
3.2 Proposed organization 152

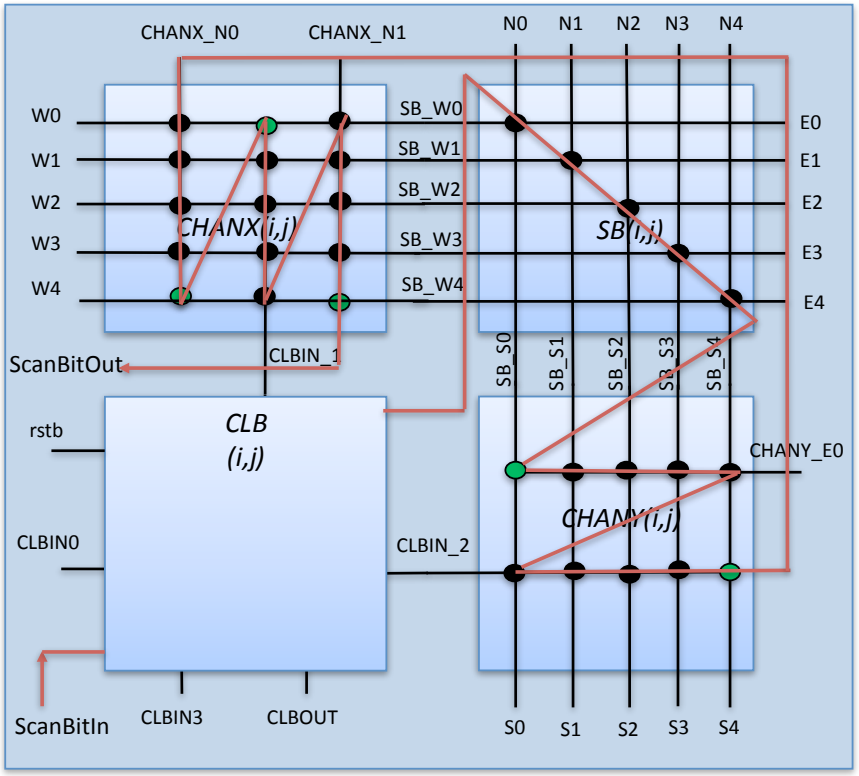


Figure A-1 – **Detailed view of the standard logic macro**

1 Logic macro-cell

The logic macro is the smallest element of the logic fabric when considering both the logic functionality and the routing network. It is formed by one position of the logic array and its surrounding interconnection channels, as shown in Figure A-1.

The logic macro contains a complete or a part of a logic block, the adjacent horizontal and vertical routing channels of the mesh network and the switch-box interconnecting these two channels. As the only signal drivers are located in the outputs of the logic blocks of the eFPGA, it is mandatory to place signal buffers at various points of the logic fabric to regenerate the signals crossing multiple pass-transistors in the connection points of the horizontal and vertical channels and the switch boxes. For design easiness, these regeneration buffers are placed in some of the switch-boxes of the fabric. The special switch-boxes are distributed in a checker pattern among the classical switch-boxes.

2 Logic blocks

The logic blocks specified in the following subsection have been developed according to the specification in previous deliverables dealing with the eFPGA as well

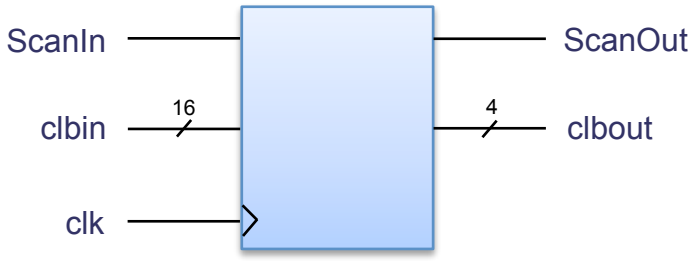


Figure A-2 – **Complex Logic Block (CLB) overview**

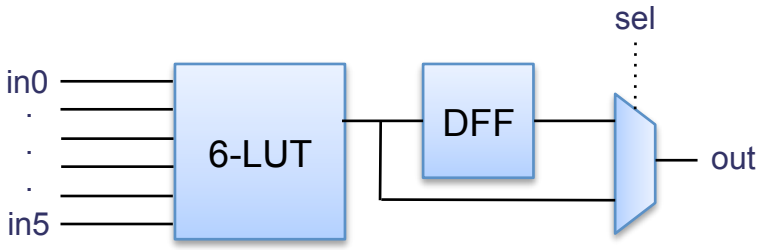


Figure A-3 – **Details of a Logic Element (LE)**

as technical constraints regarding the final eFPGA layer. For example, the area occupied by each logic block was used as a strong constraint in order to easily find a common denominator among the logic block area and thus maximize the logic fabric occupancy.

2.1 Complex Logic Block

The Complex Logic Block (CLB) is a cluster of Logic Elements (LE), comprising:

- Four logic elements
- A complete local interconnection network (crossbar) between the LEs

The CLB have 4 outputs (one for each LE) and 16 inputs, as shown on Figure 5. The number of inputs is a trade-off between the required routing resources on the global interconnection network and the use of the LE of each CLB. For a cluster of four 6-input Look-Up Tables (LUTs), 16 inputs guarantee a mean utilization of 98% of the logic elements across the circuit design [AR00].

Each of the LE of a CLB comprises the following elements (c.f. Figure A-3):

- a 6-input LUTs allowing to implement any 6-input logic function,
- a D Flip-Flop (DFF) to synchronize the LUT output with the clock,
- a 2:1 multiplexer to select either the synchronous or the combinational LUT output for this logic element.

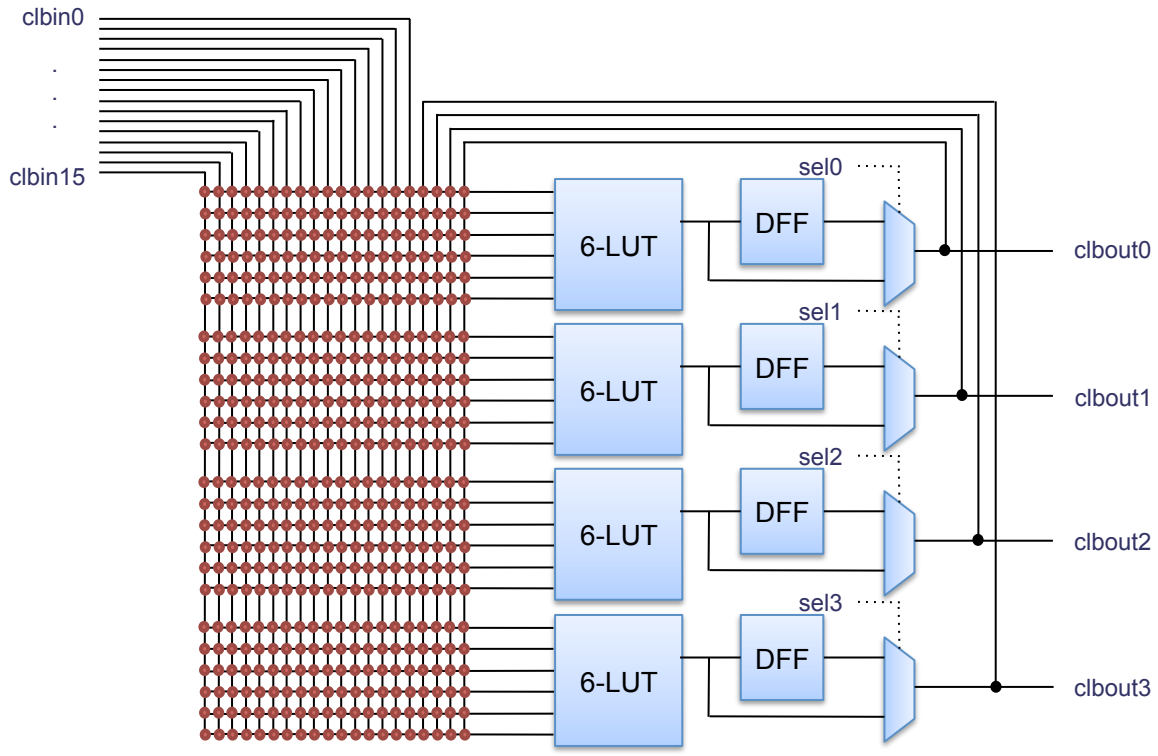


Figure A-4 – **Details of a Complex Logic Block (CLB)**

As detailed in Figure A-4, the CLB is created out of four LEs whose inputs and outputs are tied to the local interconnection network (crossbar). The inputs of this interconnection network are:

- the 16 CLB inputs,
- the four LE outputs.

We therefore have 20 different possible input signals for each LE input. The outputs of the crossbar network are connected to the 24 LE inputs (6 per LE).

Each CLB thus needs 480 configuration bits for its crossbar network, 256 bits for the logic functionality and 4 bits for the selection of the synchronous or asynchronous LE output, which sums up to 740 configuration bits.

The connection to the global interconnection network tying the CLB and the heterogeneous blocks together is spread over the four sides of the CLB, as depicted in Figure A-5.

2.2 Arithmetic accelerator

The arithmetic accelerator developed for the eFPGA is a simplified version of the DSP48E1 slice integrated in the Virtex-7 FPGA from Xilinx [Xila]. This accelerator, also called “DSP block”, should not be confused with the DSP accelerator of the FlexTiles platform, which is a full-blown processor.

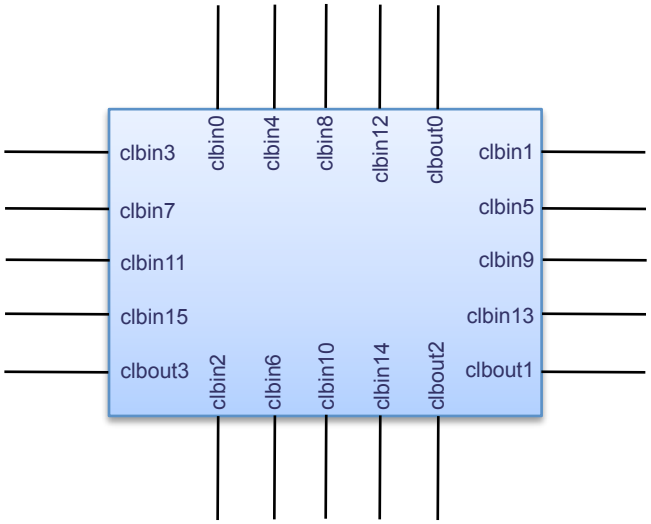


Figure A-5 – **CLB I/O repartition**

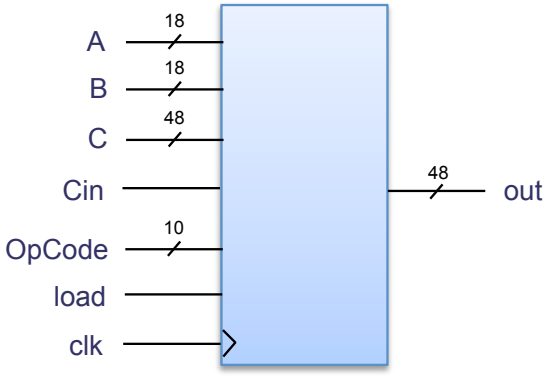


Figure A-6 – **Arithmetic accelerator (DSP block) overview**

The external view of the arithmetic accelerator is presented in Figure A-6. Table A-1 summarizes the inputs and outputs of the accelerator, their size and a description of their functionality.

Internally, the arithmetic accelerator comprises:

- a 18x18 multiplier,
- a 48-bit Arithmetic and Logic Unit (ALU),
- multiple configurable data-paths for intermediate signals.

This arithmetic accelerator design does not include a pre-adder part.

Figure A-7 shows the internal architecture of the arithmetic accelerator. The internal view of the arithmetic and logic unit is shown in Figure A-8. The ALU is responsible for the arithmetic and logic operation processing given the x, y, z, and carry inputs.

Table A-2, Table A-3, Table A-4, and Table A-5 describe the signals tied to, respectively, the intermediate signals w, x, y, and z.

Signal	Size (bits)	Description
A	18	Input A
B	18	Input B
C	48	Input C
C_{in}	1	Carry input
$OpCode$	10	Configuration word
$load$	1	Synchronous input load
Clk	1	Clock
Out	48	Output

Table A-1 – **Arithmetic accelerator signals overview**

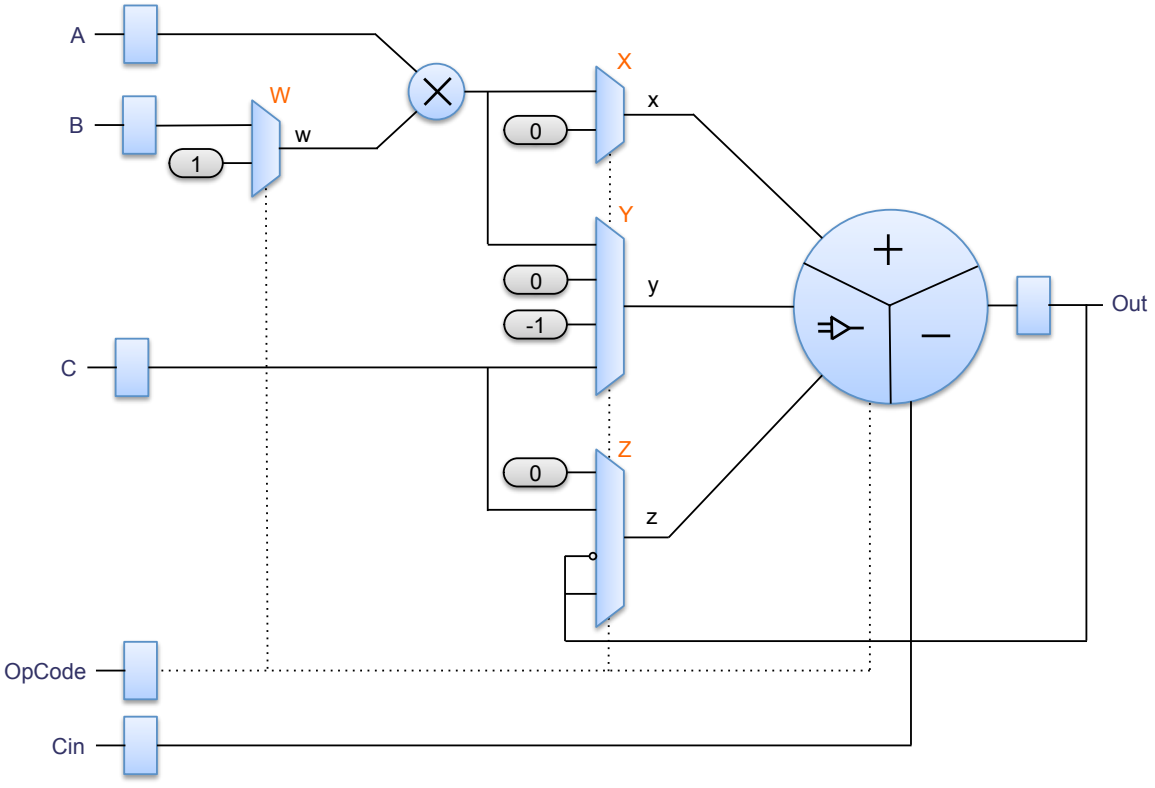


Figure A-7 – **Details of an arithmetic accelerator**

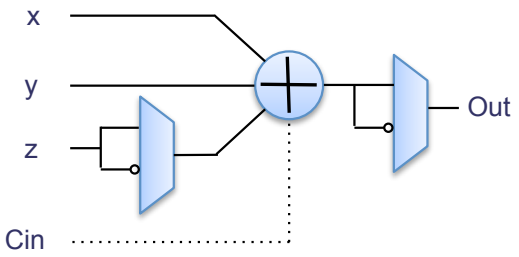


Figure A-8 – **Details of the Arithmetic and Logic Unit**

Configuration	Description
0	Input B
1	1

Table A-2 – **Configuration of the multiplexer** W

Configuration	Description
0	Signal u (multiplier)
1	0

Table A-3 – **Configuration of the multiplexer** X

Configuration	Description
00	Signal u (multiplier)
01	0
10	-1
11	Input C

Table A-4 – **Configuration of the multiplexer** Y

Configuration	Description
00	0
01	Input C
10	Complemented previous output $\overline{Out}$
11	Previous output Out

Table A-5 – **Configuration of the multiplexer** Z

Four configurations bits are dedicated to the configuration of the arithmetic and logic unit. The current ALU is a 4-input adder allowing to sum the signal x, y, Cin and z or . The ALU output can be complemented to output .

The 10-bit configuration word OpCode is made according to the description in Figure A-9.

9	8	7 6	5 4	3 0
W	X	Y	Z	ALU

Figure A-9 – *OpCode* **configuration word description**

Bits 0 to 3 of the OpCode configuration word are dedicated to the ALU and are used as described in Figure A-10. Bits 2 and 3 of the ALU configuration word are reserved for future use (RFU).

3 2	1	0
RFU	$Out/\overline{Out}$	$z/\overline{z}$

Figure A-10 – **ALU configuration word description**

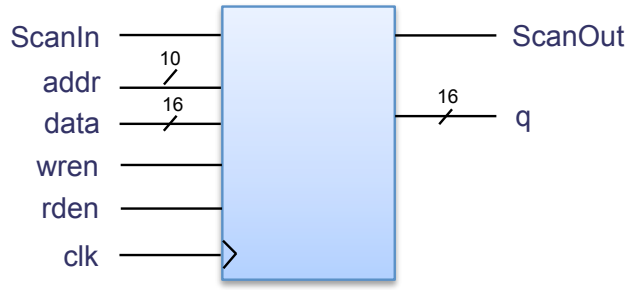


Figure A-11 – **Overview of the 16Kib memory block**

2.3 Memory block

The memory block featured in the eFPGA, depicted in Figure A-11, is a single-port read and write random access memory containing 16384 memory bits. The memory is organized around an array of 1024 16-bit words and the read and write operations operate at the word level.

The data and control signals of the RAM block are described in Table A-6.

When the wren signal is asserted low, the rden signal controls the output of the data at address addr on the output port. When wren is high, rden controls whether the output port holds the last value outputted during a read cycle or the old data at the address currently written. Table A-7 summarizes the behavior of the RAM block with regards to the rden and wren signals.

2.4 Accelerator interface

The application interface is implanted in the eFPGA in conjunction with the network interface and the 3D link to the multicore layer. It is integrated in the eFPGA in the form of a heterogeneous block and treated as such in the architecture.

Signal	Size (bits)	Description
addr	10	Address selection
data	16	Input data for the write operation
wren	1	Write enable
rden	1	Read enable
Clk	1	Clock
q	16	Output data

Table A-6 – **16Kb RAM block signals overview**

wren	rden	Operation mode
0	0	Holds the last read data
0	1	Outputs the data at address *addr*
1	0	Writes *data* at address *addr*, holds the last read data
1	1	Writes *data* at address *addr*, outputs the old data at address *addr*

Table A-7 – **16Kib RAM block operation modes**

Table A-8 – **Size chart of the eFPGA logic blocks in 65nm and 28nm**

Block	Area 65nm (μm^2)	Area 28nm (μm^2)	w.r.t. CLB
CLB	9,929	3,309	1×
Arith. Acc.	13,055	4351	1.31×
RAM block	21,134	7044†	2.13×
AI	354,617	118,205	35.71×

†: Projection only

3 Logic fabric organization

3.1 Synthesis results

In order to specify a proposal for the logic fabric organization, we designed and synthesized the various logic blocks required in the logic fabric to obtain results in terms of area usage of each of these blocks. The results of this analysis are given in Table A-8.

The area results given in Table A-8 account for the area usage of the block itself as well as the area required for the interconnection of the block with the routing network, which grows linearly in function of the number of inputs and outputs of the considered block. The area usage results are expressed in square micrometers (μm^2) in both 65nm and 28nm technology using either a full synthesis of the block on the respective technology or projections from 65nm to 28nm in the case of the memories.

Table A-9, Table A-10, Table A-11, and Table A-12 respectively give the detailed (block and interconnect) area results of the arithmetic accelerator, the CLB, the AI and the memory block in the 65nm technology.

Table A-9 – **Detailed area results of the arithmetic accelerator (65nm)**

	Block only μm^2	IX μm^2	Total μm^2
Area	5,586.08	7,469.28	13,055.36

Table A-10 – **Detailed area results of the CLB (65nm)**

	Block only μm^2	IX μm^2	Total μm^2
Area	8,151	1,778.4	9,929.4

Table A-11 – **Detailed area results of the accelerator interface (65nm)**

	Block only μm^2	IX μm^2	Total μm^2
Area	269,965.27	84,651.84	345,617.11

The memory block synthesis results have been obtained with the optimized memory generator of the target technology.

3.2 Proposed organization

Using the preceding results, the following logic-block-to-basic-unit is proposed:

- CLB: 1 to 1 (it is the basic homogeneous block of the eFPGA logic fabric)
- DSP: 1 to 2
- AI: 1 to 32
- Memory: 1 to 4, grouped by 2x16Kib memories

These ratios represent the number of each logic block to be placed along a given number of CLBs, in order to guarantee an optimized area of the logic fabric as well as to keep the fabric organized in a fixed array.

Figure A-12 presents the proposed organization and composition of the eFPGA logic fabric using the same area as the manycore layer. Using a 5500x4900μm silicon die, the eFPGA logic array is 96x85 position wide. The position of the 9 3DNI and thus the Application Interfaces (AI, in violet) are fixed and determined by the floorplan of the manycore layer. Three columns of arithmetic accelerators (in green) and 2 columns of memories (in red) are spread over the fabric. The remaining positions of the logic array are filled with CLBs (in blue). In the end, the logic fabric without the reconfiguration controller is composed of: - 9 Application Interfaces - 42 16+16 Kib memories - 210 arithmetic accelerators - 7,224 CLBs

Table A-12 – **Detailed area results of the memory block(65nm)**

	Block only μm^2	IX μm^2	Total μm^2
Area	17,400	3,734	21,134

Figure A-12 – **Proposed organization of the eFPGA logic fabric**

The proposed fabric features 29K LUTs and 1.34 Mib of dedicated memories.

Appendix B

SIMULATION OF THE EFPGA RECONFIGURATION CONTROLLER

Contents

1 Task synthesis and bit-stream generation **156**
2 Model configuration and shell connection **158**
3 Adder loading **158**
4 Adder test **159**
5 Multiplier loading **160**
6 Multiplier test **160**

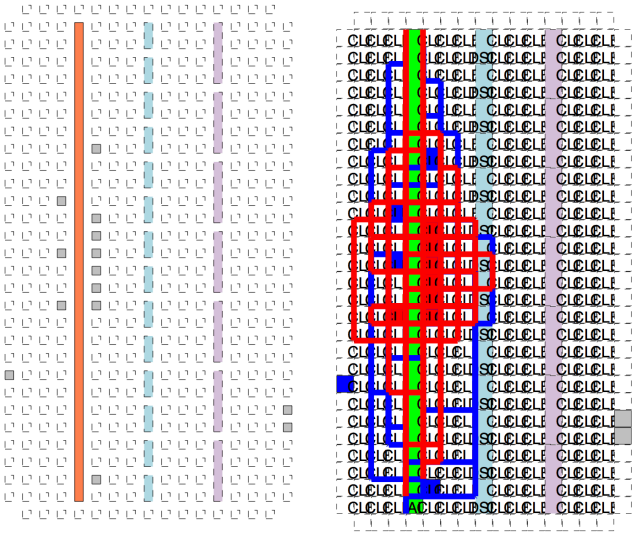


Figure B-1 – **Placement and routing of the adder16x16 task**

The RTL model test evaluation is conducted with two different tasks mult16x16 and adder16x16, respectively implementing a soft 16×16-bit multiplier and a soft 16-bit adder, which are successively loaded and tested on the final eFPGA RTL model.

1 Task synthesis and bit-stream generation

The two tasks mult16x16 and adder16x16 are described in Verilog and synthesized in the eFPGA development flow.

The output of the first part of the development flow, i.e. the generation of placement and routing data, is graphically presented in Figure B-1 and Figure B-2. In Figure B-1, the gray blocks are the used CLBs, and the orange block is the Accelerator Interface. In Figure B-2, the blue lines correspond to the fan-in routing lines of the Accelerator Interface (i.e. wires going to the AI), while red lines are the fan-out (i.e. wires going out of the AI).

The placement, routing and netlist data is then passed to the vbsgen backend dedicated to the analysis of these data and the generation of raw (BS) and virtual bit-streams (VBS). The backend also provide some useful information regarding the routing the task on the final architecture. Figure B-3 depicts the hotmap of the adder16x16 and mult16x16 tasks. Red spots are the most densely routed parts of the hardware task, where the devirtualizing algorithm (i.e. the decoder providing the raw bitstream from the VBS) will need the most complex work because of the number of routes in these areas. Green parts are the less dense parts, and the white parts are routing-free parts where the decoder will not have to do any work besides copying the logic data.

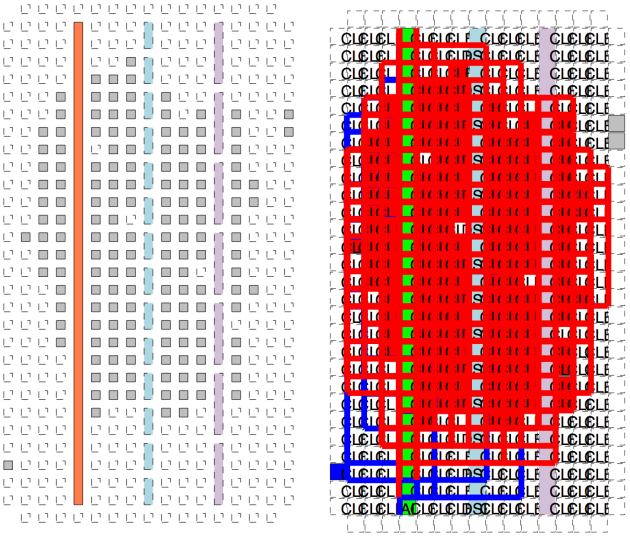


Figure B-2 – **Placement and routing of the mult16x16 task**

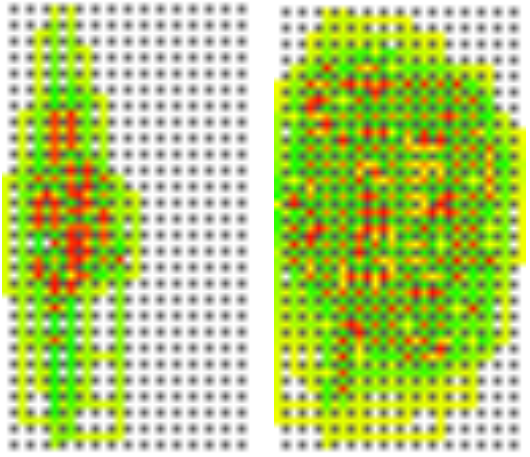


Figure B-3 – **Routing density of the adder16x16 and mult16x16 tasks**

Table B-1 – **Summary of the bit-stream generation**

Task	Raw bit-stream (bytes)	Virtual Bit-Stream (bytes)	Compression
adder16x16	46,283	16,161	2.86×
mult16x16	46,283	24,174	1.91×

Table B-1 summarizes the sizes of the raw and virtual bit-streams generated by the vbsgen backend for the tasks adder16x16 and mult16x16.

2 Model configuration and shell connection

The simulator is run in a first console:

```
$ make sim
ncsim -EXIT -loadvpi vpi-bin/libvpictrl.so:pliLoad -log report/
    sim_`date +"%Y%m%d_%H%M%S"`.rpt worklib.tb_efpga:module
ncsim(64): 12.10-p001: (c) Copyright 1995-2012 Cadence Design
    Systems, Inc.
ncsim> run
>>> Shell initialization... done. Shell port: 56095.
>>> Shell state: ON
```

In another console, a link to the controller shell is opened to interact with the controller:

```
$ nc localhost 56095
eFPGA controller shell
Copyright (c) UNIVERSITY OF RENNES 1, INRIA and IRISA 2012-2015.
All Right Reserved.

Type 'help' to get the list of commands.


efpga>
```

3 Adder loading

The VBS loading command is called in the controller shell to trigger the de-virtualization and bitstream insertion (the bitstream is inserted in the topmost and leftmost available AI by default):

```
efpga> load_vbs adder16x16.bin

90715638170 PS: VBS decoding start
90715639000 PS: Bitstream insertion start
```

Once the insertion is completed, the controller is notified:

```
90723044460 PS: Bitstream insertion end: 370272 bits inserted
```

4 Adder test

The configuration switch command is used to trigger the load from the configuration load memory layer to the active memory layer of the eFPGA:

```
efpga> conf_switch

26553810590 PS: Configuration switch triggered
```

Starting from this point, the adder16x16 task is active on the eFPGA, it is possible to retrieve the configuration of one of its logic blocks to check if the bitstream was correctly inserted:

```
efpga> sig tb_efpga.UUT.u_20_9_CLB_0135.u0_LE.LUT_reg
Signal tb_efpga.UUT.u_20_9_CLB_0135.u0_LE.LUT_reg size 64,
    defined in [...]/eFPGA/rtl/src/LE.v at line 49.
Value:
    0001010111010101110111011111111010111110110110000000000000001100
```

The input vectors can be inserted to the task through the Accelerator Interface bypass using the test command. The *AI*0 Accelerator Interface name is defined in the Verilog testbench using a custom system task prior to the controller shell initialization. This call makes the link between the shortcut and the real AI module in the testbench as such: `$controller_add_interface("AI0", AI_0003_ni_data_in, AI_0003_ni_data_out);` where `AI_0003_ni_data_in` and `AI_0003_ni_data_out` are the NI data input and output signals exported from the eFPGA module.

The eFPGA global clock is enabled and the test sequence is started:

```
efpga> clock clk on

efpga> test AI0 adder16x16.input.txt adder16x16.output.txt

670190565020 PS: Beginning of the test sequence
```

During the test sequence, the simulator outputs failed and passed tests in its console:

```
[...]
      670190569820 AI in: 0x3C85900A out: 0x0000CC8F -- PASSED
      670190569920 AI in: 0x58D0AF21 out: 0x000107F1 -- PASSED
[...]
```

In this case, the task inputs are two 16-bits numbers aggregated into the 32-bits Accelerator Interface output. The task output is the sum of these two numbers on 17-bits (the upper 15 bits are left as 0).

5 Multiplier loading

During the adder16x16 test sequence, the mult16x16 task is loaded

```
efpga> load_vbs mult16x16.bin

1178900060000 PS: VBS decoding start
1178900060170 PS: Bitstream insertion start
```

Once the insertion is completed, the controller is notified, as shown previously:

```
1178907465630 US: Bitstream insertion end: 370272 bits inserted
```

The currently loaded AI is still performing tests:

```
[...]
      1178907462320 AI in: 0x64C6D601 out: 0x00013AC7 -- PASSED
[...]
```

6 Multiplier test

In order to test the multiplier, the mult16x16 task is then loaded on the active memory layer:

```
efpga> clock clk off

efpga> conf_switch

 1371438059000 PS: Configuration switch triggered

efpga> clock clk on
```

Since we did not change the test inputs, the currently running test sequence yield failed tests in the simulator output:

```
[...]
      1371438059020 AI in: 0x92A29A6B out: 0x5872BDB6 -- FAILED (
         expected 0x00012D0D)
[...]
```

Indeed, the current test sequence tries to validate the now loaded multiplier output. It is easy to verify that 0x92A2 multiplied by 0x9A6B yields a result of 0x5872BDB6, while the test sequence was expecting the sum of the two numbers, 0x12D0D.

The new test sequence is thus loaded via the controller shell:

```
efpga> test AI0 mult16x16.input.txt mult16x16.output.txt

1558161881960 PS: Beginning of the test sequence
```

The mult16x16 test is now running as expected as seen in the simulator output:

```
[...]
       1558161895960 AI in: 0x56F003E8 out: 0x0153980 -- PASSED
[...]
```

The two 16-bit inputs aggregated into the 32-bit output of the Accelerator Interface are multiplied together to yield a 32-bits result.

PUBLICATIONS

International Conferences

[HCS15] C. Huriaux, A. Courtay, O. Sentieys. **Design Flow and Run-time Management for Compressed FPGA Configurations**. In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. DATE. Grenoble, France: EDA Consortium, Mar. 2015, pp. 1551–1554. URL: http://dl.acm.org/citation.cfm?id=2757012.2757170.

[HST14] C. Huriaux, O. Sentieys, R. Tessier. **FPGA Architecture Support for Heterogeneous, Relocatable Partial Bitstreams**. In: *Proceedings of the 24th International Conference on Field Programmable Logic and Applications*. FPL. Munich, Germany, Sept. 2014, pp. 1–6. DOI: 10.1109/FPL.2014.6927494.

[Jan+15] B. Janßen, F. Schwiegelshohn, M. Koedam, F. Duhem, L. Masing, S. Werner, C. Huriaux, A. Courtay, E. Wheatley, K. Goossens, F. Lemonnier, P. Millet, J. Becker, O. Sentieys, M. Hübner. **Designing Applications for Heterogeneous ManyCore Architectures with the FlexTiles Platform**. In: SAMOS. Samos Island, Greece, July 2015. URL: http://samos-conference.com/Resources_Samos_Websites/Proceedings_Repository_SAMOS/2015/Files/SS0_02.pdf. Forthcoming.

[Swi+15] P. Swierczynski, M. Fyrbiak, C. Paar, C. Huriaux, R. Tessier. **Protecting against Cryptographic Trojans in FPGAs**. In: *Proceedings of the 23rd International Symposium on Field-Programmable Custom Computing Machines*. FCCM. IEEE. Vancouver, Canada, May 2015. DOI: 10.1109/FCCM.2015.55.

[Tov+14] V. D. Tovinakere, O. Sentieys, S. Derrien, C. Huriaux. **Low Power Reconfigurable Controllers for Wireless Sensor Network Nodes**. In: *Proceedings of the 22nd International Symposium on Field-Programmable Custom Computing Machines*. FCCM. IEEE. Boston, USA, May 2014, pp. 230–233. DOI: 10.1109/FCCM.2014.68.

Patents

[Sen+14] O. Sentieys, A. Courtay, C. Huriaux, S. Pillement. **Method and Device for Programming an FPGA**. European Patent EP2894572. Jan. 2014.

BIBLIOGRAPHY

[Ach09] T. Achterberg. **SCIP: solving constraint integer programs**. English. In: *Mathematical Programming Computation* 1.1 (2009), pp. 1–41. ISSN: 1867-2949. DOI: 10.1007/s12532-008-0001-1. URL: http://dx.doi.org/10.1007/s12532-008-0001-1.

[AL11] A. Abdelhadi, G. Lemieux. **Configuration Bitstream Reduction for SRAM-based FPGAs by Enumerating LUT Input Permutations**. In: *the Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. 2011, pp. 20–26. DOI: 10.1109/ReConFig.2011.20.

[Alta] **Increasing Design Functionality with Partial and Dynamic Reconfiguration in 28-nm FPGAs**. Altera Corporation. 2010.

[Altb] **Stratix V Device Overview**. Altera Corp. 2015.

[AR00] E. Ahmed, J. S. Rose. **The effect of LUT and cluster size on deep-submicron FPGA performance and density**. In: *International Symposium on Field Programmable Gate Arrays*. New York, New York, USA: ACM Request Permissions, Feb. 2000, pp. 3–12.

[BKT14] C. Beckhoff, D. Koch, J. Torresen. **Portable module relocation and bitstream compression for Xilinx FPGAs**. In: *24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2014, pp. 1–8.

[BLC07] T. Becker, W. Luk, P. Y. Cheung. **Enhancing relocatability of partial bitstreams for run-time reconfiguration**. In: *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2007. FCCM'07*. IEEE. 2007, pp. 35–44.

[Bob+05] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, J. Teich. **The Erlangen slot machine: increasing flexibility in FPGA-based reconfigurable platforms**. In: *International Conference on Field-Programmable Technology*. IEEE, 2005, pp. 37–42.

[BR97] V. Betz, J. Rose. **VPR: A new packing, placement and routing tool for FPGA research**. In: *Field-Programmable Logic and Applications*. Springer. 1997, pp. 213–222.

[Bre96] G. Brebner. **A virtual hardware operating system for the Xilinx XC6200**. In: *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*. Springer, 1996, pp. 327–336. DOI: 10.1007/3-540-61730-2_35.

[BRM99] V. Betz, J. Rose, A. Marquardt. **Architecture and CAD for Deep-Submicron FPGAS**. Boston, MA: Springer, 1999.

[Bru+06] N. Bruchon, L. Torres, G. Sassatelli, G. Cambon. **New nonvolatile FPGA concept using magnetic tunneling junction**. In: *the Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*. 2006, 6 pp.–. DOI: 10.1109/ISVLSI.2006.68.

[BRV92] S. Brown, J. S. Rose, Z. G. Vranesic. **A detailed router for field-programmable gate arrays**. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 11.5 (May 1992), pp. 620–628.

[Car+86] W. Carter, K Duong, R. Freeman, H.-C. Hsieh, J. Ja, J. Mahoney, L. Ngo, S. Sze. **A user programmable reconfigurable logic array**. In: *Proceedings of the Custom Integrated Circuits Conference*. 1986, pp. 233–235.

[CB13] C. Chiasson, V. Betz. **Should FPGAS abandon the pass-gate?** In: *International Conference on Field-Programmable Logic and Applications*. IEEE, 2013, pp. 1–8.

[CCP06] D. Chen, J. Cong, P. Pan. **FPGA Design Automation: A Survey**. In: *Foundations and Trends in Electronic Design Automation* 1.3 (Jan. 2006), pp. 195–334.

[Cha+13] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, M. Margala. **An FPGA Memcached Appliance**. In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '13. Monterey, California, USA: ACM, 2013, pp. 245–254. ISBN: 978-1-4503-1887-7. DOI: 10.1145/2435264.2435306. URL: http://doi.acm.org/10.1145/2435264.2435306.

[Cho+99] P. Chow, S. O. Seo, J. S. Rose, K. Chung, G Paez-Monzon, I. Rahardja. **The design of an SRAM-based field-programmable gate array. I. Architecture**. In: *IEEE Transactions on Very Large Scale Integration Systems* 7.2 (1999), pp. 191–197.

[CKW95] S. Churcher, T. Kean, B. Wilkie. **The XC6200 FastMap™ processor interface**. English. In: *Field-Programmable Logic and Applications*. Ed. by Will Moore and Wayne Luk. Vol. 975. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1995, pp. 36–43. ISBN: 978-3-540-60294-1. DOI: 10.1007/3-540-60294-1_96.

[Com+02] K. Compton, Z. Li, J. Cooley, S. Knol, S. Hauck. **Configuration relocation and defragmentation for run-time reconfigurable computing**. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 10.3 (2002), pp. 209 –220. ISSN: 1063-8210.

[Cor+07] S. F. Corbetta, M Morandi, M Novati, M. Santambrogio, D Sciuto. **Two novel approaches to online partial bitstream relocation in a dynamically reconfigurable system**. In: *IEEE Computer Society Annual Symposium on VLSI, 2007. ISVLSI'07*. IEEE. 2007, pp. 457–458.

[CPF08] J. Carver, N. Pittman, A. Forin. **Relocation of FPGA Partial Configuration Bit-Streams for Soft-Core Microprocessors**. In: *Workshop on Soft Processor Systems*. 2008.

[CWW96] Y.-W. Chang, D. F. Wong, C. K. Wong. **Universal Switch Modules for FPGA Design**. In: *ACM Trans. Des. Autom. Electron. Syst.* 1.1 (Jan. 1996), pp. 80–101. ISSN: 1084-4309. DOI: 10.1145/225871.225886. URL: http://doi.acm.org/10.1145/225871.225886.

[Dij59] E. W. Dijkstra. **A note on two problems in connexion with graphs**. In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[Elm48] W. Elmore. **The transient response of damped linear networks with particular regard to wideband amplifiers**. In: *Journal of Applied Physics* 19.1 (1948), pp. 55–63.

[Fer+12] M. Ferger, M. Kadi, M. Hubner, M. Koedam, S. Sinha, K. Goossens, G. Almeida, J. Azambuja, J. Becker. **Hardware / Software Virtualization for the Reconfigurable Multicore Platform**. In: *the IEEE 15th International Conference on Computational Science and Engineering (CSE)*. 2012, pp. 341–344. DOI: 10.1109/ICCSE.2012.54.

[FGRG09] A. Flynn, A. Gordon-Ross, A. D. George. **Bitstream relocation with local clock domains for partially reconfigurable FPGAs**. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association. 2009, pp. 300–303.

[Fle15] FlexTiles. *FlexTiles project website*. 2015. URL: http://www.flextiles.eu (visited on 10/20/2015).

[GN07] E. Goldberg, Y. Novikov. **BerkMin: A fast and robust Sat-solver**. In: *Discrete Applied Mathematics* 155.12 (2007). {SAT} 2001, the Fourth International Symposium on the Theory and Applications of Satisfiability Testing, pp. 1549 –1561. ISSN: 0166-218X. DOI: http://dx.doi.org/10.1016/j.dam.2006.10.007. URL: http://www.sciencedirect.com/science/article/pii/S0166218X06004616.

[HEW13] E. Hung, F. Eslami, S. J. Wilton. **Escaping the academic sandbox: Realizing VPR circuits on Xilinx devices**. In: *21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2013, pp. 45–52.

[HHG98] R. Hartenstein, M. Herz, F. Gilbert. **Designing for Xilinx XC6200 FPGAs**. English. In: *Field-Programmable Logic and Applications From FPGAs to Computing Paradigm*. Ed. by ReinerW. Hartenstein and Andres Keevallik. Vol. 1482. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, pp. 29–38. ISBN: 978-3-540-64948-9. DOI: 10.1007/BFb0055230.

[HL+01] E. Horta, J. Lockwood. **PARBIT: a tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (FPGAs)**. In: *Dept. Comput. Sci., Washington Univ., Saint Louis, MO, Tech. Rep. WUCS-01-13* (2001).

[HXX05] J. Hu, T. Xu, Y. Xia. **Low-power adiabatic sequential circuits with complementary pass-transistor logic**. In: *48th Midwest Symposium on Circuits and Systems*. 2005, 1398–1401 Vol. 2. DOI: 10.1109/MWSCAS.2005.1594372.

[Jam+10] P. Jamieson, K. B. Kent, F. Gharibian, L. Shannon. **Odin II - An Open-Source Verilog HDL Synthesis Tool for CAD Research**. In: *FCCM '10: Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society, May 2010.

[KA15] J. H. Kim, J. H. Anderson. **Synthesizable FPGA fabrics targetable by the Verilog-to-Routing (VTR) CAD flow**. In: *Field Programmable Logic and Applications*. IEEE, 2015, pp. 1–8.

[Kal+04] H. Kalte, G. Lee, M. Porrmann, U. Ruckert. **Study on column wise design compaction for reconfigurable systems**. In: *the IEEE International Conference on Field-Programmable Technology (FPT)*. Dec. 2004, pp. 413–416. DOI: 10.1109/FPT.2004.1393313.

[Kal+05] H. Kalte, G. Lee, M. Porrmann, U Ruckert. **Replica: A bitstream manipulation filter for module relocation in partial reconfigurable systems**. In: *the Proceedings of the 19th Parallel and Distributed Processing Symposium*. IEEE. 2005, 151b–151b.

[KCH14] Q.-H. Khuat, D. Chillet, M. Hubner. **Dynamic run-time hardware/software scheduling for 3D reconfigurable SoC**. In: *the Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. Dec. 2014, pp. 1–4. DOI: 10.1109/ReConFig.2014.7032512.

[KGV83] S Kirkpatrick, C. D. Gelatt, M. P. Vecchi. **Optimization by simulated annealing**. In: *Science* (1983).

[KP06] H. Kalte, M. Porrmann. **REPLICA2Pro: Task Relocation by Bitstream Manipulation in Virtex-II/Pro FPGAs**. In: *the Proceedings of the 3rd conference on Computing frontiers*. CF '06. ACM, 2006, pp. 403–412. ISBN: 1-59593-302-6. DOI: 10.1145/1128022.1128045.

[Kra+06] Y. E. Krasteva, E. Torre, T. Riesgo, D. Joly. **Virtex II FPGA bitstream manipulation: application to reconfiguration control systems**. In: *International Conference on Field Programmable Logic and Applications, 2006. FPL'06.* IEEE. 2006, pp. 1–4.

[KTR08] I. C. Kuon, R. Tessier, J. S. Rose. **FPGA Architecture: Survey and Challenges**. In: *Foundations and Trends in Electronic Design Automation* 2.2 (Feb. 2008).

[Lag01] L. Lagadec. **Abstraction, modélisation et outils de CAO pour les architectures reconfigurables**. PhD thesis. Université de Rennes 1, 2001.

[LB93] G. G. Lemieux, S. D. Brown. **A detailed routing algorithm for allocating wire segments in field-programmable gate arrays**. In: *ACM-SIGDA Physical Design Workshop* (1993).

[LCC14] Q. H. Le, E. Casseau, A. Courtay. **Place Reservation technique for online task placement on a multi-context heterogeneous**

reconfigurable architecture. In: *the Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. Dec. 2014, pp. 1–6. DOI: 10.1109/ReConFig.2014.7032553.

[Lee61] C. Y. Lee. **An algorithm for path connections and its applications**. In: *IRE Transactions on Electronic Computers* 3 (1961), pp. 346–365.

[Lem+04] G. Lemieux, E. Lee, M. Tom, A. Yu. **Directional and single-driver wires in FPGA interconnect**. In: *International Conference on Field-Programmable Technology*. IEEE, 2004, pp. 41–48.

[Lem+12] F. Lemonnier, P. Millet, G. M. Almeida, M. Hubner, J. Becker, S. Pillement, O. Sentieys, M. Koedam, S. Sinha, K. Goossens. **Towards future adaptive multiprocessor systems-on-chip: an innovative approach for flexible architectures**. In: *International Conference on Embedded Computer Systems (SAMOS)*. IEEE. 2012, pp. 228–235.

[LL01] G. Lemieux, D. Lewis. **Using sparse crossbars within LUT**. In: *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*. New York, New York, USA: ACM Request Permissions, Feb. 2001, pp. 59–68.

[Luu+11] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, W. M. Fang, K. Kent, J. Rose. **VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling**. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 4.4 (2011), p. 32.

[LVT04] R. Lysecky, F. Vahid, S. X.-D. Tan. **Dynamic FPGA routing for just-in-time FPGA compilation**. In: *Design Automation Conference*. New York, New York, USA: ACM Request Permissions, 2004, pp. 954–959.

[Lyk+15] J. C. Lyke, C. G. Christodoulou, G. A. Vera, A. H. Edwards. **An Introduction to Reconfigurable Systems**. In: *Proceedings of the IEEE* 103.3 (2015), pp. 291–317.

[Maj+07] M. Majer, J. Teich, A. Ahmadinia, C. Bobda. **The Erlangen Slot Machine: A Dynamically Reconfigurable FPGA-based Computer**. In: *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology* 47.1 (2007), pp. 15–31.

[MC92] D Marple, L Cooke. **An MPGA Compatible FPGA Architecture**. In: *Custom Integrated Circuits Conference, 1992., Proceedings of the IEEE 1992*. IEEE, 1992, pp. 4.2.1–4.2.4.

[ME95] L. McMurchie, C. Ebeling. **PathFinder: a negotiation-based performance-driven router for FPGAs**. In: *the Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays*. ACM. 1995, pp. 111–117.

[Mig+03] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, R. Lauwereins. **Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip**. In: *IEEE/ACM International Conference on Design, Automation and Test in Europe, 2003*. IEEE. 2003, pp. 986–991.

[Mis+07] A. Mishchenko. **ABC: A system for sequential synthesis and verification**. In: (2007). URL: http://www.eecs.berkeley.edu/~alanmi/abc.

[Mon+07] D. P. Montminy, R. O. Baldwin, P. D. Williams, B. E. Mullins. **Using relocatable bitstreams for fault tolerance**. In: *Second NASA/ESA Conference on Adaptive Hardware and Systems, 2007. AHS 2007.* IEEE. 2007, pp. 701–708.

[MSS99] J. P. Marques-Silva, K. A. Sakallah. **GRASP: a search algorithm for propositional satisfiability**. In: *IEEE Transactions on Computers* 48.5 (May 1999), pp. 506–521.

[Nam+04] G.-J. Nam, F. Aloul, K. A. Sakallah, R. A. Rutenbar. **A comparative study of two Boolean formulations of FPGA detailed routing constraints**. In: *IEEE Transactions on Computers* 53.6 (June 2004), pp. 688–696.

[NMG13] A. Nejad, A. Molnos, K. Goossens. **A software-based technique enabling composable hierarchical preemptive scheduling for time-triggered applications**. In: *the IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2013, pp. 183–192. DOI: 10.1109/RTCSA.2013.6732218.

[RB91] J. S. Rose, S. Brown. **Flexibility of interconnection structures for field-programmable gate arrays**. In: *IEEE Journal of Solid-State Circuits* 26.3 (Mar. 1991), pp. 277–282.

[RGM01] P. Rau, A. V. Ghia, S. M. Menon. **Configuration memory architecture for FPGA**. US Patent 6,222,757. Apr. 2001.

[Ros+12] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, J. Anderson. **The VTR project: architecture and CAD for FPGAs from verilog to routing**. In: *the Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM. 2012, pp. 77–86.

[Sed+05] P. Sedcole, B. Blodget, J Anderson, P Lysaghi, T Becker. **Modular partial reconfiguration in Virtex FPGAs**. In: *International Conference on Field Programmable Logic and Applications, 2005.* IEEE. 2005, pp. 211–216.

[SF10] M. L. Silva, J. C. Ferreira. **Creation of partial FPGA configurations at run-time**. In: *13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD), 2010*. IEEE. 2010, pp. 80–87.

[Smi97] M. J. S. Smith. **Application-specific Integrated Circuits**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0-201-50022-1.

[ST69] F. Sangster, K. Teer. **Bucket-brigade electronics: new possibilities for delay, time-axis conversion, and scanning**. In: *IEEE Journal of Solid-State Circuits* 4.3 (1969), pp. 131–136. ISSN: 0018-9200. DOI: 10.1109/JSSC.1969.1049975.

[TD08] H. Tan, R. F. DeMara. **A multilayer framework supporting autonomous run-time partial reconfiguration**. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16.5 (2008), pp. 504–516.

[Tor+13] L. Torres, R. Brum, L. Cargnini, G. Sassatelli. **Trends on the application of emerging nonvolatile memory to processors and programmable devices**. In: *the Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*. 2013, pp. 101–104. DOI: 10.1109/ISCAS.2013.6571792.

[Tou+12] M. Touiza, G. Ochoa-Ruiz, E.-B. Bourennane, A. Guessoum, K. Messaoudi. **A Novel Methodology for Accelerating Bitstream Relocation in Partially Reconfigurable Systems**. In: *Microprocessors and Microsystems* (2012).

[Wil97] S. J. E. Wilton. **Architecture and algorithms for field programmable gate arrays with embedded memory**. PhD thesis. University of Toronto, 1997.

[WMS94] Y.-L. Wu, M Marek-Sadowska. **An efficient router for 2-D field programmable gate array**. In: *European Design and Test Conference, 1994. EDAC, The European Conference on Design Automation. ETC European Test Conference. EUROASIC, The European Event in ASIC Design, Proceedings*. IEEE, 1994, pp. 412–416.

[WR98] R. G. Wood, R. A. Rutenbar. **FPGA routing and routability estimation via Boolean satisfiability**. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 6.2 (1998), pp. 222–231.

[WTMS96] Y.-L. Wu, S. Tsukiyama, M. Marek-Sadowska. **Graph based analysis of 2-D FPGA routing**. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 15.1 (1996), pp. 33–44.

[Xila] **7 Series DSP48E1 Slice User Guide, UG479**. Xilinx, Inc. 2014.

[Xilb] **7 Series FPGAs Configuration**. Xilinx, Inc. 2014.

[Xilc] **Partial Reconfiguration User Guide, UG702**. Xilinx, Inc. 2013.

[Xild] **Virtex-E 1.8V Field Programmable Gate Arrays**. Xilinx, Inc. 1999.

[Xile] **Virtex-II Platform FPGAs: Complete Data Sheet**. Xilinx, Inc. 2000.

[Xilf] **Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet**. Xilinx, Inc. 1999.

[Xilg] **XC2064/XC2018 Logic Cell Array**. Xilinx, Inc. 1984.

List of Figures

0-1 Overview of an FPGA architecture and its configuration mechanism . 2

1-1 Details of the configurable sum-of-products in a PAL 8
1-2 Implementation of a two input boolean equation in a Look-Up Table 10
1-3 Evolution of the number and complexity of hard blocks in FPGAs [Lyk+15] 15
1-4 Architecture of a 2-LUT using memory cells and pass-transistors . . . 16
1-5 Overview of a basic logic block 17
1-6 Overview of a complex logic block 18
1-7 Area-delay product for clusters of size 1 to 10 [AR00] 18
1-8 Number of inputs required for 98% logic utilization [AR00] 19
1-9 Overview of the DSP48E1 slice included in Virtex 7 series FPGAs [Xila] 21
1-10 Schematized routing network of an FPGA device 23
1-11 Example of a distribution of routing tracks in an interconnect architecture 24
1-12 Overview of a hierarchical interconnect architecture 25
1-13 Overview of an island-style interconnect architecture 26
1-14 Switch block topologies 27
1-15 Implementation of bi-directional routing with pass-transistors and buffers 28
1-16 Directional tristate and single driver implementations of unidirectional output drivers [Lem+04] 29

2-1 Compatible patterns on a logic fabric for task relocation 37
2-2 Becker *et al.* [BLC07] solution to logic fabric heterogeneity 38
2-3 A typical FPGA design flow 42
2-4 Illustration of the maze routing algorithm 45
2-5 The Verilog to Routing design flow 49

3-1 Overview of the Virtual Bit-Stream technique integrated to a FPGA logic fabric 56
3-2 Example of an island-style architecture and its repeatable *macro-cell* 58
3-3 Details of a macro-cell and its routing details 59
3-4 Abstract model of a macro-cell 60
3-5 Virtual Bit-Stream abstraction, from routing to connection list 62
3-6 Clustering multiple macro-cells together before coding the Virtual Bit-Stream 66
3-7 The Virtual Bit-Stream design flow, based on the Verilog-To-Routing framework 68

3-8 The *vbsgen* model framework . . . 73
3-9 The *vbsgen* model array . . . 74
3-10 Block model used in the *vbsgen* framework . . . 75
3-11 Generation of intermediate representations and final bit-stream in the *vbsgen* framework . . . 76
3-12 Integration of the Virtual Bit-Stream technique in an FPGA system . 77
3-13 Overview of the Virtual Bit-Stream reconfiguration controller . . . 78

4-1 Architecture of the decoding part of the online controller . . . 82
4-2 Example macro-cell architecture with $W = 4$ and $L = 5$ . . . 85
4-3 Details of the Look-Up Table (LUT)-based decoding from the route data to the interconnect bit-stream . . . 86
4-4 Principles of the FSM-based algorithm, from routing to state machine 87
4-5 Normalized compression ratio of Deflate, Bzip2 and LZMA compressed raw bit-streams and VBS compared to the raw bit-stream size . . . 92
4-6 Frequency of apparition of average wire lengths of 10 place and route runs of the VTR benchmarks . . . 94
4-7 Normalized number of fallback-coded macro-cells with the LUT algorithm for each design of the VTR benchmark set . . . 95

5-1 Relocation problem in modern FPGAs . . . 98
5-2 Abstraction of heterogeneous resources . . . 99
5-3 Switch box enhanced to include connections between heterogeneous and homogeneous layers . . . 100
5-4 Multiple relocation possibilities for the same bit-stream of a task . . . 101
5-5 Model of a task with abstracted heterogeneous hard blocks . . . 102
5-6 Two placements of the same task on the enhanced architecture with VPR . . . 103

6-1 An array of memory cells accessible with word addressing . . . 111
6-2 Memory cells daisy-chained to form a scan-path . . . 112
6-3 A hybrid scheme between a scan-path and a word-addressed memory [RGM01] . . . 113
6-4 An additional latch for each memory cell separates the configuration memory from the active memory . . . 115
6-5 The double layer memory increases the dynamicity of the architecture 115
6-6 Basic organization of a standard of hybrid scan-path over the programmable device . . . 117
6-7 Configuration memory routing element . . . 118
6-8 A configurable shift-register . . . 119
6-9 Usage example of the configurable shift-register with an 8×4 task . . 120

7-1 The 3-D stacked FlexTiles platform . . . 124
7-2 Overview of the FlexTiles platform heterogeneous many-core hardware architecture [Jan+15] . . . 125
7-3 Development flow of the automatic RTL model generation . . . 130
7-4 Implementation of the simulation testbench of the eFPGA . . . 131

7-5 Layout of a homogeneous version of the eFPGA logic fabric 137

A-1 Detailed view of the standard logic macro 144
A-2 Complex Logic Block (CLB) overview 145
A-3 Details of a Logic Element (LE) 145
A-4 Details of a Complex Logic Block (CLB) 146
A-5 CLB I/O repartition 147
A-6 Arithmetic accelerator (DSP block) overview 147
A-7 Details of an arithmetic accelerator 148
A-8 Details of the Arithmetic and Logic Unit 148
A-9 *OpCode* configuration word description 149
A-10 ALU configuration word description 150
A-11 Overview of the 16Kib memory block 150
A-12 Proposed organization of the eFPGA logic fabric 153

B-1 Placement and routing of the adder16x16 task 156
B-2 Placement and routing of the mult16x16 task 157
B-3 Routing density of the adder16x16 and mult16x16 tasks 157

List of Tables

1-1 Summary of the three main FPGA programming technologies 14

3-1 Example Virtual Bit-Stream data on a macro-cell with $W = 5$, $L = 7$ and four routes 65

4-1 Portion of an example interconnection matrix dedicated to a routing resource graph 84
4-2 Online LUT-based and FSM-based and offline Maze routing algorithms time and space complexities 87
4-3 Hardware implementations of the decoding algorithms 88
4-4 Benchmark set used to test the Virtual Bit-Stream technique 90
4-5 Results underlining the compression effects of the Virtual Bit-Stream 91
4-6 Effect of the clustering on Virtual Bit-Stream sizes 93

5-1 Post-packing composition of 12 heterogeneous designs of the VTR benchmark set 105
5-2 Place and route benchmark results on the standard architecture . . . 106
5-3 Place and route benchmark results on the enhanced architecture . . . 107

6-1 Comparison of memory organization in terms of area, power consumption, speed and flexibility 116

7-1 Synthesis results of the architectural elements of the eFPGA on 28nm and 65nm technology nodes 127

A-1 Arithmetic accelerator signals overview 148
A-2 Configuration of the multiplexer W 149
A-3 Configuration of the multiplexer X 149
A-4 Configuration of the multiplexer Y 149
A-5 Configuration of the multiplexer Z 149
A-6 16Kb RAM block signals overview 150
A-7 16Kib RAM block operation modes 151
A-8 Size chart of the eFPGA logic blocks in 65nm and 28nm 151
A-9 Detailed area results of the arithmetic accelerator (65nm) 151
A-10 Detailed area results of the CLB (65nm) 152
A-11 Detailed area results of the accelerator interface (65nm) 152
A-12 Detailed area results of the memory block(65nm) 152

B-1 Summary of the bit-stream generation 157

ACRONYMS

AI Accelerator Interface. 125–128

AIG And-Inverter Graph. 50

ALU Arithmetic and Logic Unit. 20, 21, 42, 127

ASIC Application-Specific Integrated Circuit. 1, 9, 10, 33, 42, 44

BLE Basic Logic Element. 58, 59, 73

BLIF Berkeley Logic Interchange Format. 49, 50, 69

CAD Computer-Aided Design. 1–4, 15, 16, 21, 22, 29, 32, 51, 122, 132, 136, 138, 139

CGRA Coarse-Grained Reconfigurable Architecture. 139

CLB Complex Logic Block. 90, 93, 99, 127, 128

CMOS Complementary Metal-Oxyde Semiconductor. 12–14, 128

CPLD Complex Programmable Logic Device. 9, 10, 36

CRC Cyclic Redundancy Check. 36, 39

CSDF Cyclo-Static Data Flow. 125

DDR Double Data Rate. 125

DRAM Dynamic Random Access Memory. 15

DSP Digital Signal Processor. 20, 21, 29, 124, 125

EEPROM Electrically Erasable and Programmable Ready-Only Memory. 9, 12–14

EPROM Electrically Programmable Ready-Only Memory. 13

FIFO First-In First-Out. 20, 125

FP7 Seventh Framework Programme. 4, 123, 124, 136

FPGA Field-Programmable Gate Array. iii, viii, 1–4, 7–18, 20–29, 31–51, 55–58, 64, 65, 67, 70–74, 77–79, 82, 83, 86, 88, 90, 96–98, 101–104, 106, 108–111, 115, 116, 122–129, 132, 133, 135, 137–139, 173

FSM Finite State-Machine. 84, 87–89, 93, 135, 174

GAL Generic Array Logic. 9

GPP General Purpose Processor. 124–126, 129, 132, 139

HDD hard disk drive. 56

ICAP Internal Configuration Access Port. 83

IP Intellectual Properties. 55, 101, 137

JIT just-in-time. 40, 41, 57

LB Logic Block. 99–101, 104

LUT Look-Up Table. 9, 16–20, 34, 43, 50, 58, 59, 64, 73, 84–90, 93, 95, 127, 128, 174

LZ77 Lempel-Ziv 77. 91

LZMA Lempel-Ziv Markov chain Algorithm. 92

LZSS Lempel-Ziv-Storer-Szymanski. 39

MAC multiply-accumulate. 20

MOSFET Metal-Oxyde Semiconductor Field-Effect Transistor. 13

MPGA Mask-Programmable Gate Array. 10, 15, 44

MPSoC Multiprocessor System-on-Chip. 139

MRAM Magnetic Random Access Memory. 14

NI Network Interface. 125–128

NMOS N-channel Metal-Oxyde Semiconductor. 12, 13

NoC Network on Chip. 124–126

OTP One-Time Programmable. 9, 11

OVP Open Virtual Platform. 123, 132

PAL Programmable Array Logic. 8, 9

PLA Programmable Logic Array. 8

PLD Programmable Logic Device. 8, 9

PRR Partially-Reconfigurable Region. 2, 34, 35, 40, 97, 109, 110, 116

RAM Random Access Memory. 20, 35–37, 49, 56, 71, 127, 128

ROM Read-Only Memory. 9

RTL Register Transfer Level. 3, 42, 43, 49, 51, 55, 67, 129, 131, 132, 136

SDF Synchronous Data Flow. 125

SLU Swappable Logic Unit. 34

SoC System-on-Chip. 21

SRAM Static Random Access Memory. 12–14, 16, 18, 20, 27, 28, 34, 110, 113

SSD solid-state drive. 56

TDP Thermal Design Power. 1

TSV Through-Silicon Via. 128

UVPROM Ultraviolet Programmable Ready-Only Memory. 13

VBS Virtual Bit-Stream. 2, 57, 83

VLSI Very Large Scale Integration. 112

VPR Versatile Place-and-Route. 3, 40, 41, 48–50, 68–72, 83, 91, 93, 102–105, 108, 136, 138, 139, 174

VTR Verilog-To-Routing. 3, 48–51, 67, 68, 71, 72, 90, 91, 93–95, 102–104, 108, 138, 174