Closing the Gap between FPGA and ASIC: Balancing Flexibility and Efficiency

THÈSE N^{O} 5339 (2012)

PRÉSENTÉE LE 27 MARS 2012

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE D'ARCHITECTURE DE PROCESSEURS

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Hadi PARANDEH AFSHAR

acceptée sur proposition du jury:

Prof. E. Sanchez, président du jury
Prof. P. Ienne, directeur de thèse
Prof. G. De Micheli, rapporteur
Prof. G. Lemieux, rapporteur
Dr A. Mishchenko, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2012

Acknowledgements

Foremost, I would like to thank my wonderful advisor, **Prof. Paolo Ienne**, for all his effective supports and his patience. He is not only a good advisor, but also a great teacher and manager. I hope to be lucky enough to work with such a professional manager in my future career. His comments and guidance always improved my work, and the result is a thesis that includes two *Best Paper Awards* received from top international conferences.

I would like to appreciate my committee members, **Prof. Giovanni De Micheli**, **Prof. Guy Lemieux**, and **Dr. Alan Mishchenko**, for having accepted this demanding task, reading this dissertation, and providing constructive feedbacks. I would like to thank **Prof. Eduardo Sanchez** for serving as the president of my jury.

Parts of this thesis have been published in some conferences and journals, in which many of my colleagues contributed over the years. I have had the privilege of interacting with some of present and past members of Processor Architecture Laboratory (LAP) including Prof. Philip Brisk, Dr. David Novo, Dr. Madhura Purnaprajna, Dr. Ajay K. Verma, Alessandro Cevrero, Grace Zgheib, Arkosnato Neogy, Panagiotis Athanasopoulos, and Hind Benbihi. I would like to thank all these colleagues and friends for their incredible collaborations.

The members of the LAP group provided a wonderful work environment in which it was hard not to be productive, in particular I would like to thank **Chantal Schneeberger** whose constant support and encouragement has been an important factor in the completion of this thesis. Additionally, her contribution in editing the thesis also cannot be ignored. I would like to express my special thanks to Xavier Jimenez, my office mate, for all his useful tips and also his contribution in editing the thesis. Also, I would like to thank René Beuchat for his contribution in editing the thesis and his feedbacks on my research.

I am grateful to **Prof. Babak Falsafi** for his supports, fruitful guidances, and inspiring feedbacks. I learnt many concepts related to computer architecture from him in the two courses that I had with him.

I am thankful to Dr. Alan Mishchenko for his kind supports in connecting me with various FPGA companies, which enabled to present my ideas and receive practical feedbacks that improved my work. Also, I would like to thank Alan for providing his wonderful synthesis and verification tool, ABC. This open source tool had a key role in our research advances.

Needless to mention that without financial support from Swiss National Science Foundation (SNSF), it was impossible to make such a progress in the field. The SNSF fund provided the opportunity to concentrate on the research and attend several international conferences all over the world.

Acknowledgements

Finally, I would like to thank my lovely wife, Mahboobeh Karamian, for her patience and supports during our extended and long student life. She, with her patience and encouragement, gave me the flexibility to work overtime and during the weekends to meet all tough deadlines. It is my great pleasure to dedicate this thesis to my wife, Mahboobeh, and my little daughter, Helen. Also, I would like to appreciate our parents for their support, understanding and endurance that helped us to live happily, although we were thousands of miles away from them.

Lausanne, 12 March 2011 Hadi P. Afshar

Abstract

Despite many advantages of *Field-Programmable Gate Arrays (FPGAs)*, they fail to take over the IC design market from *Application-Specific Integrated Circuits (ASICs)* for high-volume and even medium-volume applications, as FPGAs come with significant cost in area, delay, and power consumption. There are two main reasons that FPGAs have huge efficiency gap with ASICs: (1) FPGAs are extremely flexible as they have fully programmable *soft-logic* blocks and routing networks, and (2) FPGAs have *hard-logic* blocks that are only usable by a subset of applications. In other words, current FPGAs have a heterogeneous structure comprised of the flexible soft-logic and the efficient hard-logic blocks that suffer from inefficiency and inflexibility, respectively. The inefficiency of the soft-logic is a challenge for any application that is mapped to FPGAs, and lack of flexibility in the hard-logic results in a waste of resources when an application can not use the hard-logic.

In this thesis, we approach the inefficiency problem of FPGAs by bridging the efficiency/flexibility gap of the hard- and soft-logic. The main goal of this thesis is to compromise on efficiency of the hard-logic for flexibility, on the one hand, and to compromise on flexibility of the soft-logic for efficiency, on the other hand. In other words, this thesis deals with two issues: (1) adding more generality to the hard-logic of FPGAs, and (2) improving the soft-logic by adapting it to the generic requirements of applications.

In the first part of the thesis, we introduce new techniques that expand the functionality of FPGAs hard-logic. The hard-logic includes the dedicated resources that are tightly coupled with the soft-logic—i.e., adder circuitry and carry chains—as well as the stand-alone ones—i.e., DSP blocks. These specialized resources are intended to accelerate critical arithmetic operations that appear in the pre-synthesis representation of applications; we introduce mapping and architectural solutions, which enable both types of the hard-logic to support additional arithmetic operations. We first present a mapping technique that extends the application of FPGAs carry chains for carry-save arithmetic, and then to increase the generality of the hard-logic, we introduce novel architectures; using these architectures, more applications can take advantage of FPGAs hard-logic.

In the second part of the thesis, we improve the efficiency of FPGAs soft-logic by exploiting the circuit patterns that emerge after logic synthesis, i.e., connection and logic patterns. Using these patterns, we design new soft-logic blocks that have less flexibility, but more efficiency than current ones. In this part, we first introduce *logic chains*, fixed connections that are integrated between the soft-logic blocks of FPGAs and are well-suited for long chains of logic that appear post-synthesis. Logic chains provide fast and low cost connectivity, increase the

bandwidth of the logic blocks without changing their interface with the routing network, and improve the logic density of soft-logic blocks.
In addition to logic chains and as a complementary contribution, we present a non-LUT soft-logic block that comprises simple and pre-connected cells. The structure of this logic block is inspired from the the logic patterns that appear post-synthesis. This block has a complexity that is only linear in the number of inputs, it sports the potential for multiple independent outputs, and the delay is only logarithmic in the number of inputs. Although this new block is less flexible than a LUT, we show (1) that effective mapping algorithms exist, (2) that, due to their simplicity, poor utilization is less of an issue than with LUTs, and (3) that a few LUTs can still be used in extreme unfortunate cases.
In summary, to bridge the gap between FPGAs and ASICs, we approach the problem from two complementary directions, which balance flexibility and efficiency of the logic blocks of FPGAs. However, we were able to explore a few design points in this thesis, and future work could focus on further exploration of the design space.

Keywords: FPGA, Efficiency Gap, ASIC, FPGA Logic Block, FPGA Mapping Algorithm, Logic Chains, Carry Chains, And-Inverter Graph, And-Inverter Cone, DSP Block, Carry Save Arithmetic, Look-Up Table.

Résumé

Malgré leurs nombreux avantages, les *Field-Programmable Gate Arrays (FPGAs)* ne parviennent pas à s'emparer du marché de conception de circuits intégrés pour des applications à hauts ou même moyens volumes, actuellement contrôlés par les *Application-Specific Integrated Circuits (ASICs)*, car les FPGAs ont des coûts importants en terme de surface, délai et consommation d'énergie. Il y a deux raisons principales expliquant le large écart de rendement entre les FPGAs et les ASICs : (1) les FPGA sont extrêmement flexibles, grâce aux routage programmables et des blocs de logique programmables, et (2) les FPGAs ont des blocs spécialisés, à logique non-programmable, utilisables seulement par un sous-ensemble d'applications. En d'autres termes, les FPGAs actuelles ont une structure hétérogène, composée de logique programmable flexible et de logique spécialisée efficace qui souffrent respectivement d'inefficacité et d'inflexibilité. L'inefficacité de la logique programmable est un défi pour n'importe quelle application implementeé sur FPGA. Le manque de flexibilité dans la logique spécialisée entraîne un gaspillage de ressources lorsque l'application ne peut en bénéficier.

Dans cette thèse, nous proposons d'am'eliorer l'inefficacité des FPGAs en équilibrant la performance et la flexibilité de la logique spécialisée et programmable des FPGAs. L'objectif principal de cette thèse est d'une part, de faire des concessions sur la performance de la logique spécialisée pour plus de la flexibilité, et d'autre part de faire des concessions sur la flexibilité de la logique programmable pour gagner en performance. En d'autres termes, cette thèse traite de deux questions : (1) ajouter plus de généralité à la logique spécialisée des FPGAs, et (2) l'améliorer la logique programmable en l'adaptant aux besoins des applications.

Dans la première partie de la thèse, nous introduisons de nouvelles techniques qui élargissent les fonctionnalités de la logique spécialisée des FPGAs. La logique spécialisée comprend les ressources dédiées qui sont étroitement couplées avec la logique programmable–par exemple, circuits additionneurs et chaînes de retenue–ainsi que les cellules autonomes–par exemple, les blocs DSP. Ces ressources ciblent généralement les opérations arithmétiques avec l'intention de les utiliser avant la synthèse ; nous introduisons des solutions de synthèse et architecturales, qui permettront aux deux types de logique spécialisée de faciliter des opérations arithmétiques supplémentaires. Nous présentons d'abord une technique de synthèse qui permet l'utilisation des chaînes de retenue des FPGAs pour des applications sans propagation de retenue, puis afin d'augmenter encore la généralité de la logique spécialisée, nous introduisons de nouvelles architectures ; en utilisant ces architectures, un plus grand nombre d'applications peut tirer avantage de la logique spécialisée des FPGAs.

Dans la deuxième partie de la thèse, nous améliorons l'efficacité de la logique programmable

des FPGAs en exploitant des motifs de circuits qui émergent après la synthèse logique, comme des motifs d'interconnection et de logique. En utilisant ces modèles, nous concevons de nouveaux blocs à logique programmable qui ont moins de flexibilité, mais sont plus performant que ceux actuels. Dans cette partie, nous présentons d'abord les *logic chains*, des connexions fixes qui sont intégrées entre les blocs à logique programmable des FPGAs et sont bien adaptées aux longues chaînes logiques qui apparaissent après synthèse. Les *logic chains* fournissent une connectivité rapide et à faible coût, augmentent la bande passante des blocs logiques sans changer leur interface avec le réseau de routage, et améliorent la densité logique du bloc à logique programmable.
En plus des *logic chains*, nous vous présentons un bloc à logique programmable sans *Look-Up Table (LUT)* qui comporte des cellules simples et pré-connectées. La structure de ce bloc logique est inspirée des motifs logiques qui apparaissent après synthèse. Ce bloc a une complexité qui n'est linéaire que dans le nombre d'entrées, il peut potentiellement avoir de multiples sorties indépendantes, et le délai est uniquement proportionel au logarithmique du nombre d'entrées. Bien que ce nouveau bloc est extrêmement moins flexible que les LUTs traditionnels, nous montrons (1) que des algorithmes de synthèse efficaces existent, (2) qu'en raison de leur simplicité, une faible utilisation est moins un problème que pour les LUTs, et (3) que quelques LUTs peuvent toujours être utilisés dans des cas extrêmes.
En résumé, pour combler le fossé entre les FPGAs et les ASICs, nous abordons le problème par deux directions complémentaires, lesquelles équilibrent la flexibilité et la performance des blocs logiques des FPGAs. Seuls quelques points du problème ont pu être couverts par la présente thèse et des travaux futurs pourraient continuer sur une exploration plus poussée du problème dans sa totalité.

Mots-clés : FPGA, Écart de Rendement, ASIC, Blocs de Logique Programmables, Algorithmes de Synthèse, Logic Chains, Chaînes de Retenue, And-Inverter Graph, And-Inverter Cone, Bloc DSP, Applications sans Propagation de Retenue, Look-Up Table.

Contents

List of Figures

List of Tables

1 Introduction

With the increase of *Application-Specific Integrated Circuit (ASIC)* design costs and the pressure of time-to-market, *Field-Programmable Gate Arrays (FPGAs)* continue to replace ASICs in many low- and mid-volume products. The cost of designing ASICs, including *Non-Recurring Engineering (NRE)*, mask, and development costs, is increasing every year as Moore's law progresses and applications get extremely complicated. Issues such as power, signal integrity, clock tree synthesis, and manufacturing defects can add significant risk and time-to-market delays. FPGAs offer a viable and competitive option to ASIC development by reducing the risk of re-spins, high NRE costs, and time-to-market delays, as an off-the-shelf FPGA has already been fabricated and verified.

Although they outperform traditional software, FPGAs have a significant *gap* in performance, power consumption, and area utilization with ASICs. A recent research by Kuon and Rose [50] indicate that FPGAs require approximately 20 to 35 times more area, have a speed roughly three to four times slower, and consume roughly 10 times more dynamic power than standard-cell ASICs. Figure 1.1 pictorially illustrates the FPGA and ASIC gaps in the flexibility and efficiency design space. As shown, FPGAs are the most flexible choice for implementing applications, but their efficiency is low; ASICs offer the best efficiency, but they are not flexible. This is the reason that ASICs are still the first choice for implementing high-volume applications, in which the unit cost remains low.

In the past, many researchers have attempted to enhance FPGAs, but despite more than 20 years of research, FPGAs are way behind ASICs in efficiency, as described in the previous paragraph. Previous studies have shown that without innovations in FPGA architecture, advances in device technology, alone, cannot noticeably shrink this gap between FPGAs and ASICs. In this thesis, we introduce a *roadmap* for improving FPGAs, and we show how each contribution of the thesis fits into the roadmap. In the next section, we first describe the motivation of this thesis, and then we introduce the mentioned roadmap and the thesis contributions.

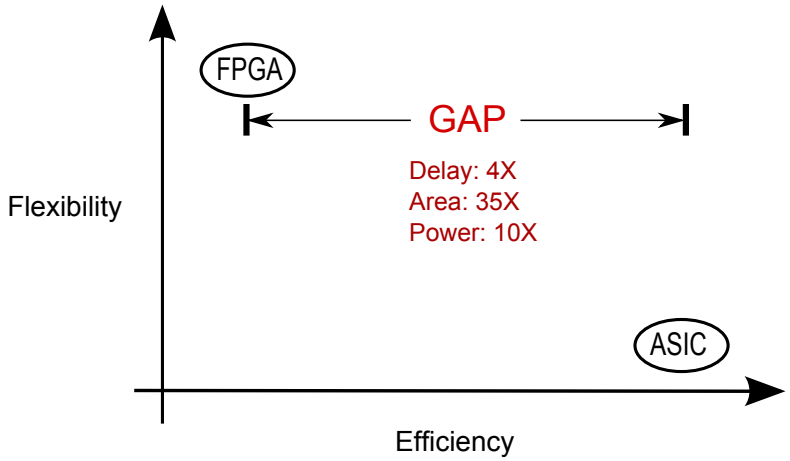


Figure 1.1: FPGA versus ASIC. FPGA is highly flexible and less efficient, while ASIC is highly efficient and less flexible. The flexibility of FPGA comes at a price, which is the delay, area and power gap between FPGAs and ASICs [50].

1.1 Thesis Motivation

Early FPGAs were created with the purpose of implementing any possible digital circuit; hence, they had a highly flexible structure consisting of fully programmable logic blocks—*Look-Up Tables (LUTs)*—and a routing network. This flexibility and its consequent advantages do not come for free: logic blocks and routing resources tend to be large and slow. Later, FPGA vendors embedded ASIC-like logic blocks into FPGAs to soften the inefficiency problem of FPGAs. These new blocks have little flexibility and are dedicated to critical arithmetic operations that occur frequently in many signal processing applications. As a result, the FPGA structure became heterogeneous, comprised of the original flexible logic blocks and new dedicated blocks, which are called *soft-* and *hard-logic* in this thesis, respectively.

Although the hard-logic can improve FPGAs when it is utilized efficiently, due to its inflexible nature, many applications cannot take advantage of it. In [50], Kuon and Rose have evaluated the impact of the hard-logic on narrowing the gaps between FPGAs and ASICs. The results of this study for the area and delay gaps are illustrated in Figure 1.2. As shown, the area gap can be improved substantially when the hard-logic is actually used. In principle, operations that are implemented by the hard-logic are also implementable by the soft-logic. But, the area that is consumed by the soft-logic is much larger than the area of the dedicated implementation. Though, due to the inflexible nature of the dedicated blocks, it is quite possible that these blocks are not utilized in many applications, which makes the area efficiency problem even worse.

On the other hand, the impact of the hard-logic on delay improvement is little compared to the area improvement. Kuon and Rose [50] give three reasons for that: (1) the hard-logic is less flexible and it might be underutilized, (2) some parts of applications still need to be mapped to the soft-logic, which is inherently slow, and (3) the cost of routing to the (stand-alone)

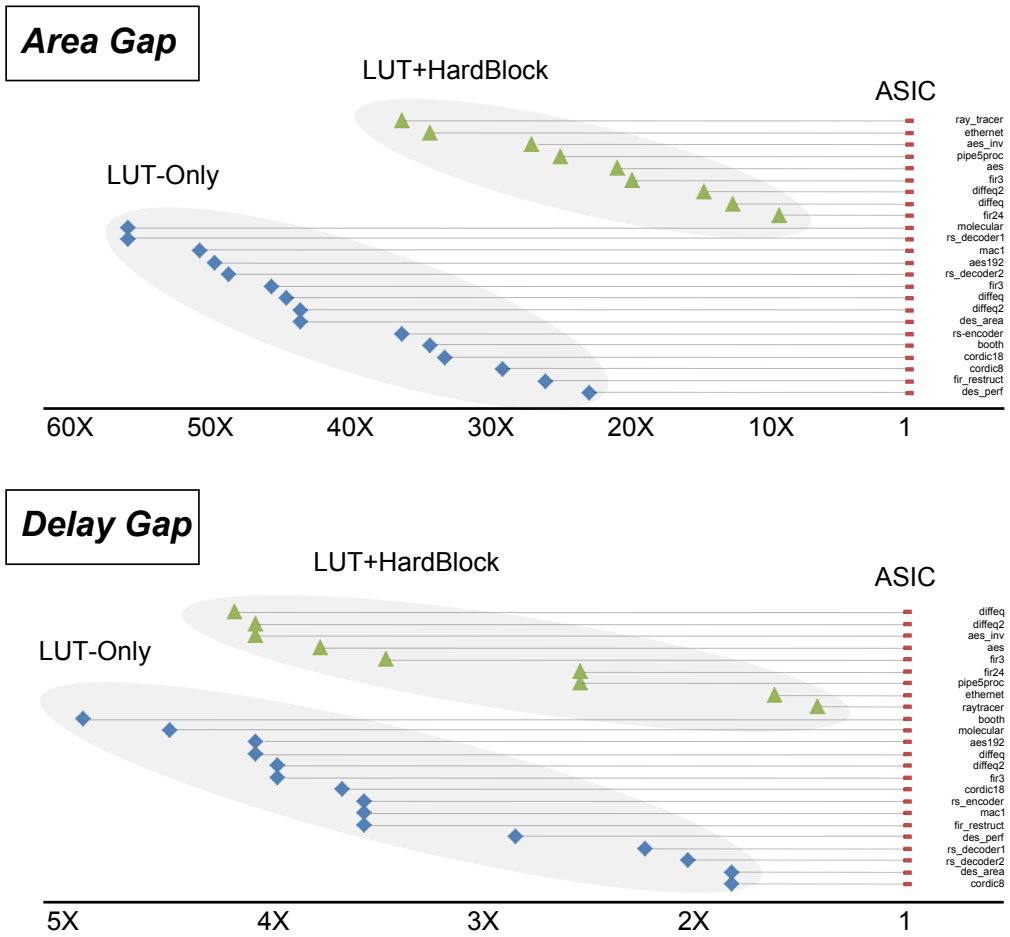


Figure 1.2: Effect of using hard-logic in FPGAs in narrowing their area and delay gaps with ASICs [50]. Although the hard-logic can significantly reduce the area gap, its effect on the delay gap is not considerable for three reasons: (1) Underutilization of the hard-logic, (2) high routing cost to the stand-alone hard-logic, and (3) the presence of the FPGAs soft-logic in critical-paths.

hard-logic is high and the interconnection delay can be considerable. This implies that the impact of the hard-logic on delay improvement of an application is highly dependent on how the rest of the application is implemented by the soft-logic.

In summary, current heterogeneous FPGAs have logic blocks that suffer from either *inflexibility* or *inefficiency*, which both limit the enhancement of FPGAs. This fact motivated us to approach the gap problem from two perspectives, as shown in Figure 1.3. The top plot in this (conceptual) figure illustrates the current efficiency gap between FPGAs and ASICs. In this figure, it is assumed that the same applications are used for comparing FPGAs against ASICs, but for FPGAs, these applications are categorized into two sets: (1) the first set contains the applications that can exclusively use the soft-logic for their implementation, and (2) the second set contains the ones that use a mixture of the soft- and hard-logic. Typically, more applications fall into the first set compared to the second. Hence, the overall average of the

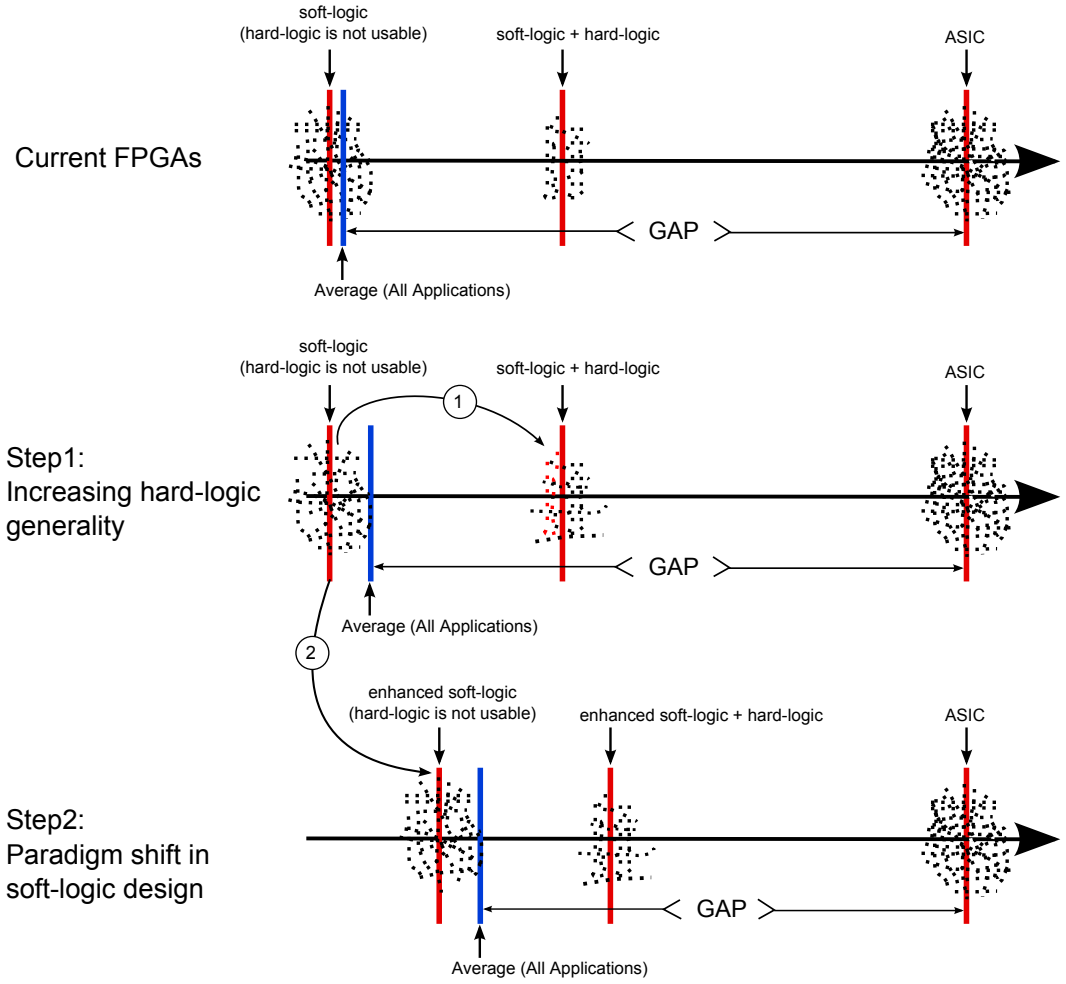


Figure 1.3: Thesis roadmap for improving FPGAs. To bridge the FPGAs and ASICs efficiency gap, two complementary steps are required: (1) increasing the generality of the hard-logic such that more applications benefit from it, and (2) improving the soft-logic efficiency, which impacts all applications, through novel logic blocks.

FPGAs efficiency tends to be closer to the average efficiency of the applications in the first set.

The next two plots in Figure 1.3 reveal the roadmap that we follow in this thesis for enhancing FPGAs. In this figure, two orthogonal and complementary paths to improve FPGAs are presented. The first path—the middle plot—suggests that one way of improving the overall efficiency of FPGAs is to increase the number of the applications that can take advantage of the FPGAs hard-logic. This requires to compromise on efficiency of the hard-logic for the flexibility. The more number of applications benefit from these resources, the more overall reduction in the gap will be obtained. This is an essential step to improve FPGAs, however, its impact is still limited by the soft-logic constraints, as described earlier. Therefore, it is crucial and complementary to improve the soft-logic of FPGAs, as suggested in the bottom plot of Figure 1.3. Any enhancement in the soft-logic will affect all applications, as they typically use

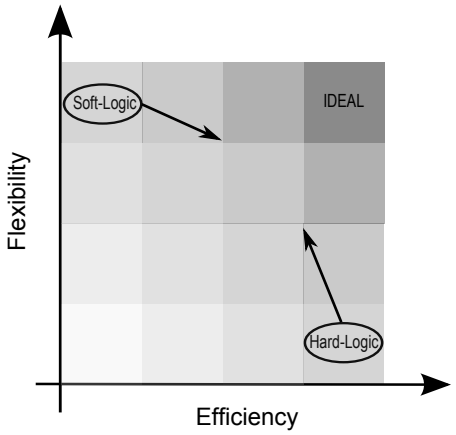


Figure 1.4: Ideal FPGA and how the roadmap of Figure 1.3 helps to get closer to this ideal FPGA.

the soft-logic for their implementation.

Figure 1.4 shows how the presented roadmap in Figure 1.3 can lead to having improved FPGAs. This figure suggests that to get closer to an FPGA with *ideal* logic blocks—top-right—the gap between the soft-logic and hard-logic of FPGAs should be narrowed by improving the efficiency and flexibility of the soft- and hard-logic, respectively. However, such improvements should not severely affect the mapping generality of the soft-logic and the efficiency of the hard-logic—the ideal case for both is when we have no negative impact. The other observation from this figure is that the ideal FPGA—although may not be feasible—should have homogeneous structure consisting of ideal logic blocks, as opposed to current FPGAs.

1.2 Thesis Organization

This thesis is organized into *eight* chapters, including this chapter. In Chapter 2, we review some background information that are required to better understand the thesis contributions, which are presented in the next five chapters—Chapters 3, 4, 5, 6, and 7. The final chapter—Chapter 8— concludes the thesis and discusses about future research opportunities. In the following, we introduce the thesis organization in more detail.

To establish the background knowledge that is required to understand the thesis contributions, we provide some basic information in Chapter 2. First, we review the generic architecture of FPGAs and the CAD flow that is used to design with them. Next, we introduce two recent high-end FPGAs from two leading FPGA vendors, namely, *Altera* and *Xilinx*. In this part, we briefly review the architectural aspects of these two FPGAs that relate to the contributions of this thesis. Finally, we present some computer arithmetic concepts, which we will refer to in the rest of the thesis.

Following the thesis roadmap that was presented in the previous section, this thesis con-

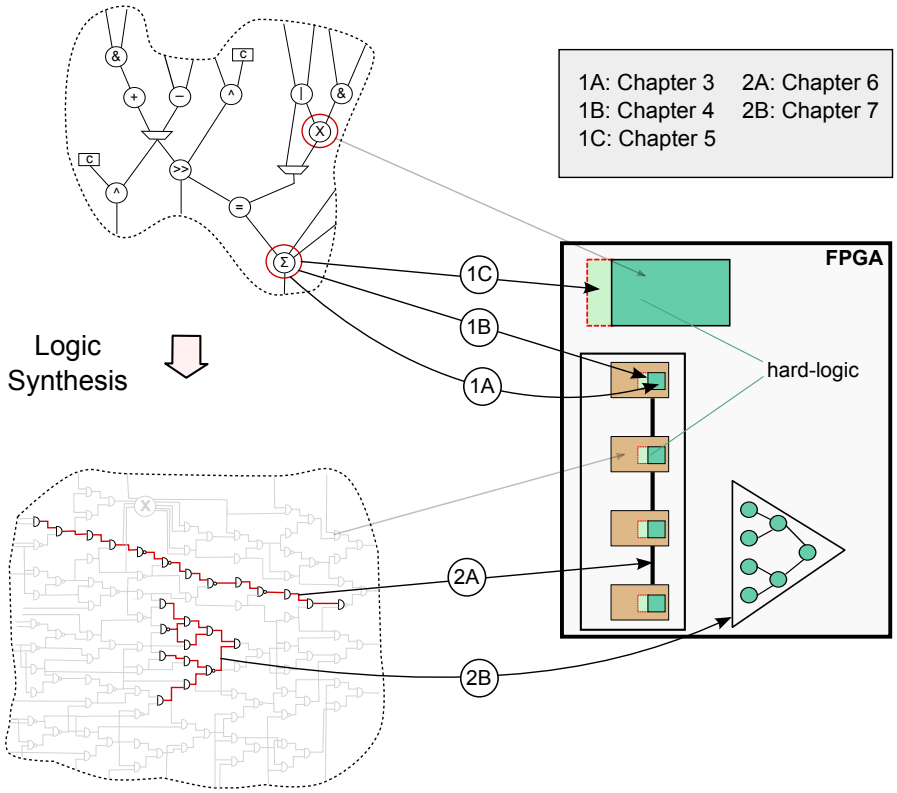


Figure 1.5: Thesis organization. Based on the thesis roadmap, shown in Figure 1.3, we approach the gap problem from two complementary directions: (1) Improving the generality of the hard-logic blocks through synthesis (1A) and architectural (1B, 1C) methods, and (2) lightening the stress on the expensive routing network through locally connected (2A) and synthesis-inspired (2B) logic blocks.

tributes in two complementary parts, as shown in Figure 1.5. In the first part—Chapters 3, 4, and 5—we discuss how we can add more functionality to the hard-logic of current FPGAs by either *mapping* or *architectural* methods. While, in the second part—Chapters 6 and 7—we present new architectures to enhance the soft-logic of FPGAs.

The current hard-logic of FPGAs is typically specialized for the critical arithmetic operations; hence, as shown in the top part of Figure 1.5, one direction of the thesis is to expand the functionality of the hard-logic to support more arithmetic operations. This can be done either by new mapping techniques or architectural modifications, as shown in this figure—*1A*, *1B*, and *1C*.

Chapter 3, which corresponds to contribution *1A* in Figure 1.5, presents a new mapping algorithm that uses the adder circuitry and carry chain of FPGAs for performing carry-save arithmetic. Adders and carry chains are considered as the hard-logic that is coupled with the LUTs in FPGAs. This adds some level of flexibility, which enables to use in (an unintuitive way) these resources for a purpose that was not intended. Carry chains are exclusively intended

for carry-propagate based circuit implementations. Though, in Chapter 3, we present a method of using carry chains to map compressor trees—refer to Chapter 2 for the definition of compressor trees—on FPGAs using the carry chains.

However, due to the hardware constraints of the carry chains, we faced challenges for the mapping technique of Chapter 3. This motivated us to modify the hard-logic structure and introduce a new carry chain in Chapter 4; this new carry chain has a non-propagating nature and improves the logic density of FPGAs for compressor trees. The new carry chain corresponds to contribution *1B* in Figure 1.5, in which the hard-logic that is mixed with the soft-logic is architecturally improved.

In Chapter 5, in contrast to Chapters 3 and 4, we focus on the stand-alone hard-logic of FPGAs, specifically DSP blocks. This chapter presents a new and versatile architecture for the current DSP blocks in FPGAs, which have more flexibility in supporting multiplication bit-widths, and additionally, it enables us to reuse the adder circuitry of the block for implementing compressor trees. Supporting more multiplication bit-widths is useful to tailor the DSP blocks to the applications requirements rather than ending up in the underutilized DSP blocks. Moreover, accessing the adder logic of the multipliers provides the opportunity that additional applications gain from the DSP blocks. This chapter corresponds to contribution *1C* in Figure 1.5.

The second direction of the thesis, based on the roadmap, is to enhance the soft-logic of FPGAs using *post-synthesis* patterns, as shown in the bottom part of Figure 1.5. Using these patterns, which become visible when applications pass through logic synthesis, we can simplify the design of FPGAs soft-logic and reduce its excess flexibility. These post-synthesis patterns include both connection and logic patterns; the connection patterns enable integration of fixed and hard-wired connections between the soft-logic blocks (*2A*), and the logic patterns are exploited to design non-LUT and efficient soft-logic blocks (*2B*).

In current FPGAs, the routing delay is as critical, if not more critical than, the logic delay. This is also the case for power consumption. Hence, it is crucial to avoid using routing wires, especially for long chains of logic that appear in the post-synthesis netlists. This motivated us to introduce the idea of *logic chain*, in which fracturable LUTs spanned over the logic blocks in an array, are cascaded with hard-wired connections to build larger LUTs and to replace routing wires. These LUTs that are formed along the chain can be exploited to map the critical chains of logic. Chapter 6 presents this idea and corresponds to contribution *2A* in Figure 1.5.

In addition to the long chains of logic, there are some post-synthesis logic patterns, which can be exploited to design a radically different, yet general, soft-logic block that has a complexity that is only linear in the number of inputs, it sports the potential for multiple independent outputs, and the delay is only logarithmic in the number of inputs. Although this new block is extremely less flexible than a LUT, we show (1) that effective mapping algorithms exist, (2) that, due to their simplicity, poor utilization is less of an issue than with LUTs, and (3) that a few LUTs can still be used in extreme unfortunate cases. This idea is presented in Chapter 7

and corresponds to contribution *2B* in Figure 1.5.

Finally, Chapter 8 concludes this thesis with some final remarks and a review of the contributions. We also provide some possible research areas that can extend the research related to this thesis.

2 Background and Preliminaries

This thesis presents a roadmap to improve FPGAs efficiency by mapping and architectural techniques. The purpose of this chapter is to provide background information on basic concepts and architectures that will be used in the rest of the thesis. Hence, we first give a brief and general overview on FPGA architecture and tools. Next, we introduce the state-of-the-art high-end FPGAs from both *Altera* and *Xilinx* that are referred in this thesis. Finally, we present some computer arithmetic concepts and architectures, which are necessary to understand the contributions of this thesis.

2.1 FPGAs Introduction

Field-Programmable Gate Arrays (FPGAs) are prefabricated devices that can be programmed to implement any digital circuit. In principle, FPGAs consist of logic blocks and routing network that each can be programmed to implement a digital design. Original FPGAs had a *homogeneous* structure—all the logic blocks were identical—comprised of *Look-Up Tables (LUTs)*, which we call *soft-logic* in this thesis. Current FPGAs, however, are *heterogeneous*, consist of specialized and dedicated blocks, which we call *hard-logic* in this thesis, and the original soft-logic. Current FPGAs are augmented with special purpose hard-logic, tailored to certain operations that are critical and occur frequently in many applications, for improved efficiency.

In the following sub-sections, we will briefly review some architectural details of current FPGAs, and then we will introduce the CAD flow that is used to program such FPGAs.

2.1.1 FPGAs Architecture

Current FPGAs, as shown in Figure 2.1, consist of different types of programmable logic blocks, including the LUT-based soft-logic, memory, and DSP blocks surrounded by a programmable routing fabric that allows flexible interconnection of the blocks. The FPGA in Figure 2.1 has an *island-style* routing structure, in which FPGA blocks are arranged in a two dimensional

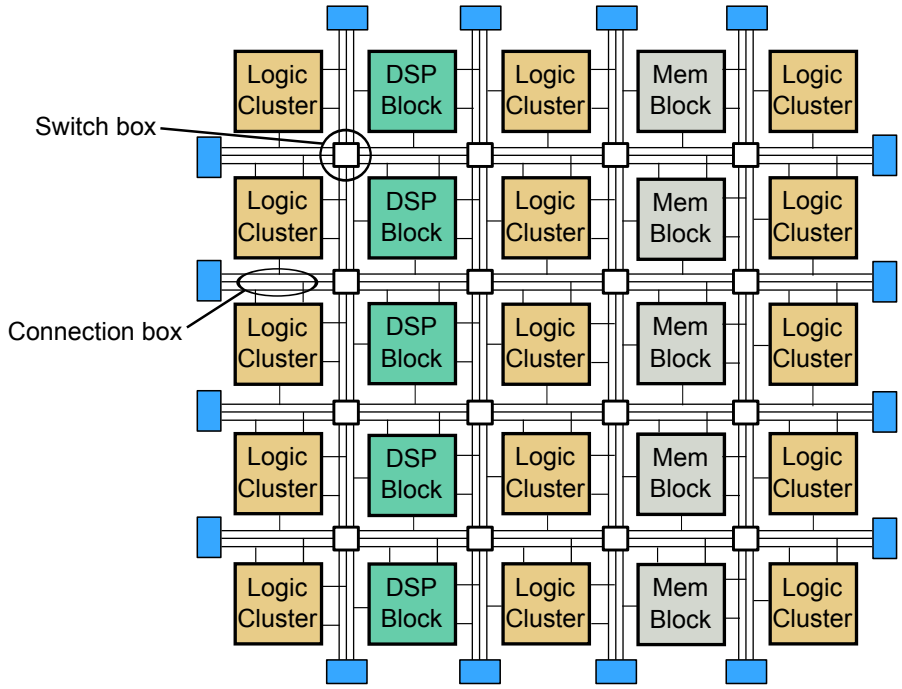


Figure 2.1: An island-style FPGA with heterogeneous structure. Different types of FPGA blocks, including *soft-logic* and *hard-logic* blocks, interconnected through a two dimensional routing network consisted of routing switches.

mesh with routing resources evenly distributed throughout the mesh. In this structure, there are *routing channels* and *connection boxes* on all four sides of FPGA blocks. The number of wires in the horizontal and vertical channels is pre-set during the FPGA chip fabrication. Currently, most commercial SRAM-based FPGA architectures [6, 7, 52, 91, 92] use island-style architecture. The typical routing resources include *switch boxes*, *connection boxes*, and *wire segments*. A *switch box* [75] is a programmable block that allows the wires of two intersecting channels to be connected based on routing demand. Meanwhile, a *connection box* [75] provides the connectivity between the pins of FPGA blocks and the routing channels.

The box labeled as *logic cluster* in Figure 2.1 is an array of soft-logic blocks that consist of *Look-Up Tables (LUTs)* and certain dedicated gates. Logic blocks clustering is an effective approach to lighten the stress on the routing network, as most circuits that map to FPGAs exhibit locality, necessitating short and fast interconnection wires. The logic blocks that reside in the same cluster are connected through a *local* routing network, which routes the cluster inputs and logic blocks outputs to the logic blocks inputs. Figure 2.2 illustrates a sample logic block cluster. As shown, the input crossbar of the cluster provides the connectivity of the logic blocks that reside in the same cluster. The generic structure of a logic block in current FPGAs is also shown in this figure (right). In current modern FPGAs, each logic block is a mixture of LUTs and dedicated circuits such as adders and carry chains. Carry chains include fast connections between adjacent logic cells that are used for carry propagation; this permits the elimination of most of the routing delays that would otherwise be present. In addition to

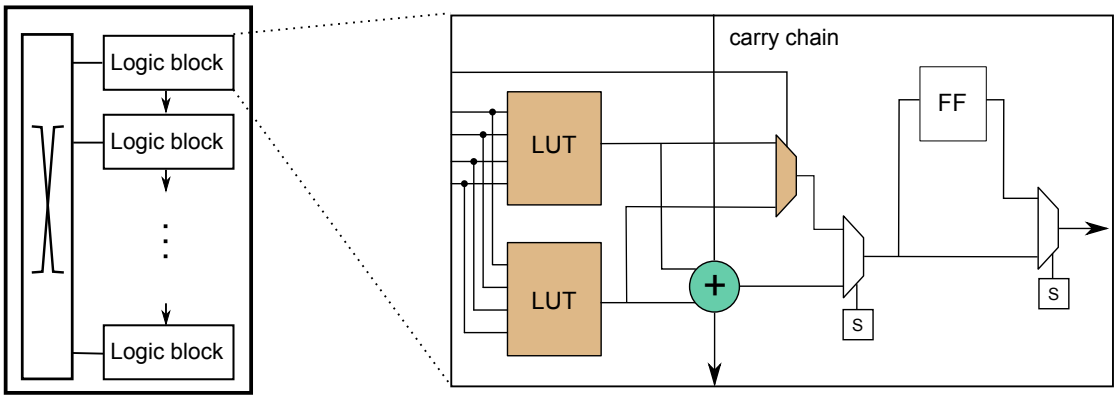


Figure 2.2: Generic structure of logic cluster (left) and logic block (right) in current FPGAs. Logic cluster is an array of logic blocks with a local routing network that is mainly a crossbar. Logic block has a fracturable LUT structure and supports fast arithmetic addition using the dedicated circuitry and hard-wired carry chains.

the hard-logic that is tightly coupled with the soft-logic—i.e., adders and carry chains that are integrated with LUTs—FPGAs include another hard-logic type that is stand-alone and not mixed with the soft-logic, such as DSP and memory blocks, as shown in Figure 2.1. Almost all current commercial FPGAs have DSP blocks with different architectures; these DSP blocks mainly implement multiplication, in addition to other few operations, such as shifting and accumulation. These are the operations that are commonly present in many signal processing applications. Such operations, however, still can be implemented by the soft-logic of FPGAs with lower quality.

2.1.2 FPGAs CAD Flow

Given the complexity of current applications, CAD tools are the indispensable part of the design process. Today, most FPGA vendors provide a fairly complete set of design tools that allows automatic synthesis and compilation from design specifications in hardware specification languages, such as Verilog or VHDL, all the way down to a bit-stream to program FPGA chips. Figure 2.3 shows the steps of a typical FPGA design flow. Inputs to the design flow generally include the HDL specification of the design, the design constraints, and the specification of the target FPGA device.

As the first step, the HDL design is elaborated into generic logic functions as well as datapath operations, for which the target FPGA has architectural support, such as adders and multipliers. Next, the elaborated design passes through technology independent logic optimization. In this step, both sequential and combinational parts of the circuits are optimized. Sequential logic optimizations include finite state machine encoding/minimization and retiming, and combinational logic optimization includes constant propagation, redundancy removal, logic network restructuring and optimization, and *don't-care* based optimization.

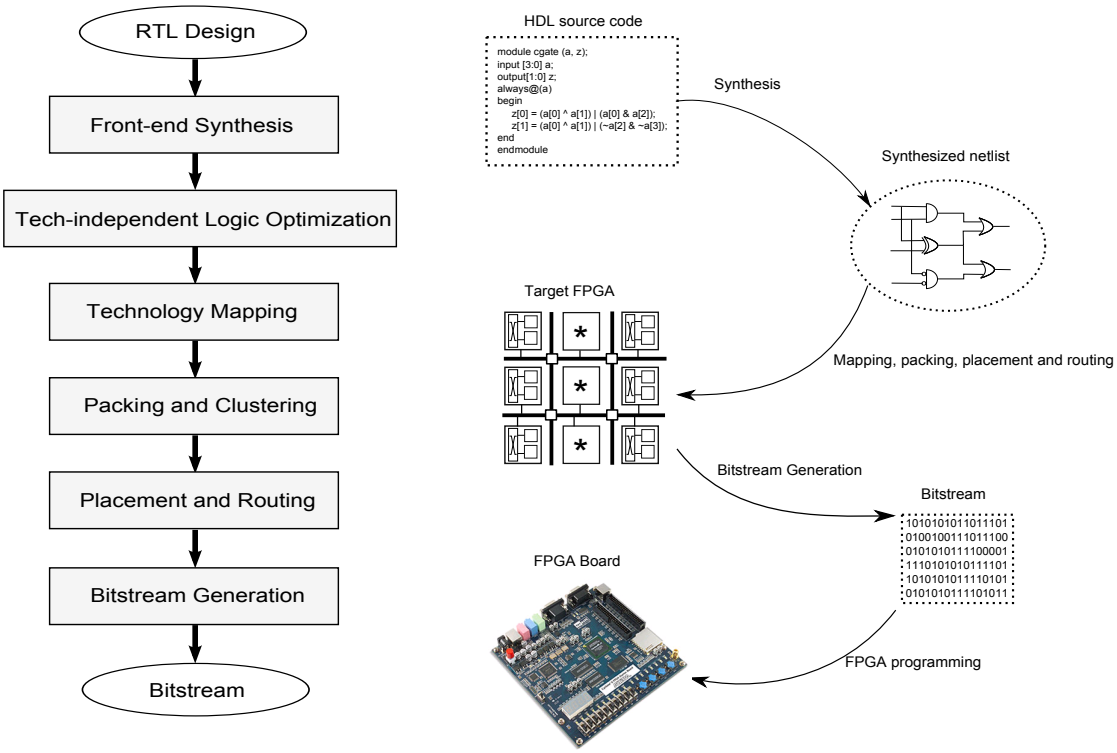


Figure 2.3: Typical CAD and design flow in current FPGAs.

Technology mapping is the next step, which maps logic-optimized circuits to logic blocks of the target FPGA, including the soft- and hard-logic blocks. Note that the arithmetic operators such as adders and multipliers can be mapped either to the hard-logic or the soft-logic, depending on the efficiency of the mapping and availability of resources. Then, the mapped circuit goes through the packing process, in which the mapped blocks are packed into logic clusters. For instance, LUTs and flipflops are packed into logic blocks, and then logic blocks are grouped into logic clusters.

After the packing, placement and routing is performed, during which the packed blocks are placed and interconnected on the grid of the FPGA. The main objective in this step is to reduce the wire lengths and minimize the usage of routing resources through a proper placement of the packed blocks. Once this step is performed, the configuration of the FPGA is determined. In the final step of the design flow, using the configuration information, a bitstream is generated to program the logic and interconnects of the target FPGA to implement the intended design.

2.2 State-of-the-art FPGAs

Altera and *Xilinx* are the two main FPGA market leaders. Both have *high-end* FPGA devices that resemble the generic FPGA architecture that was presented in the Section 2.1.1. The

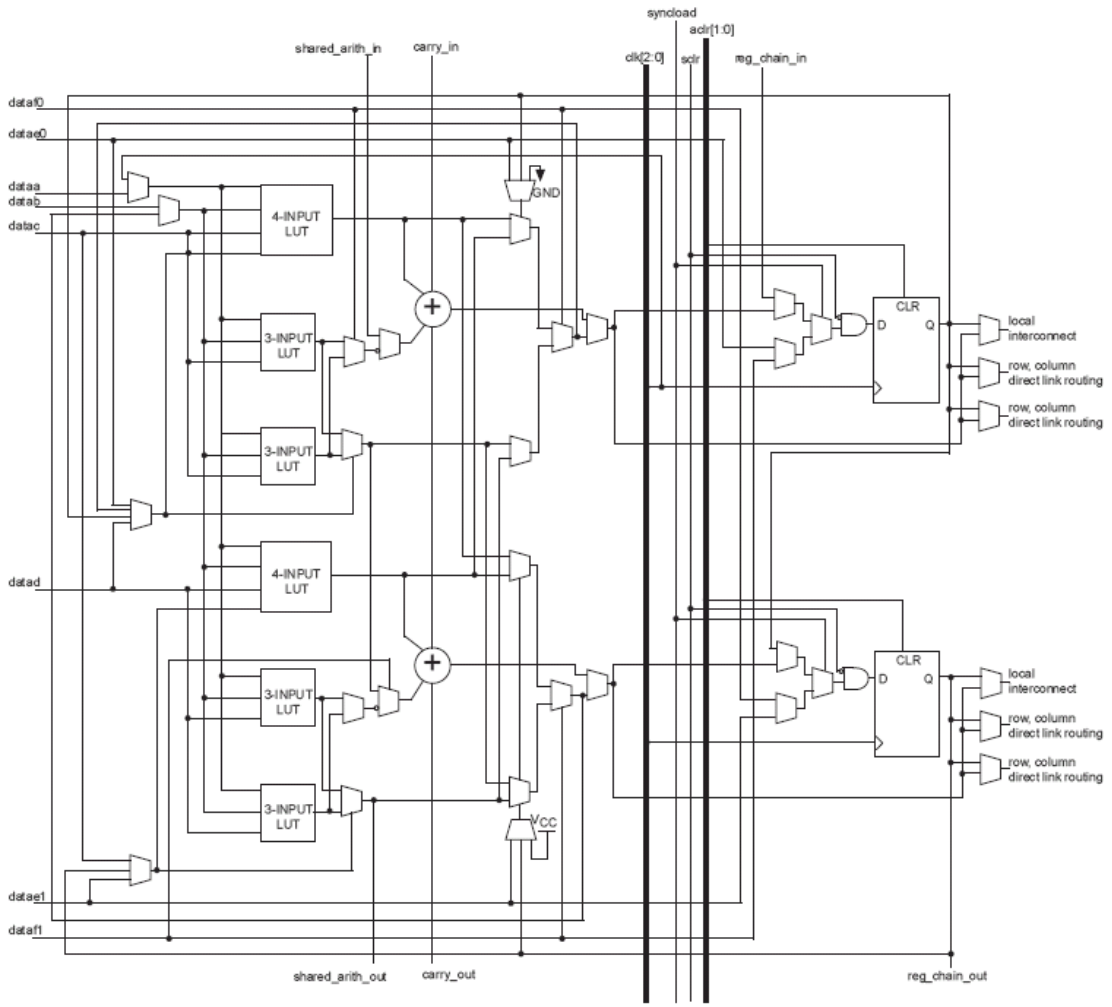
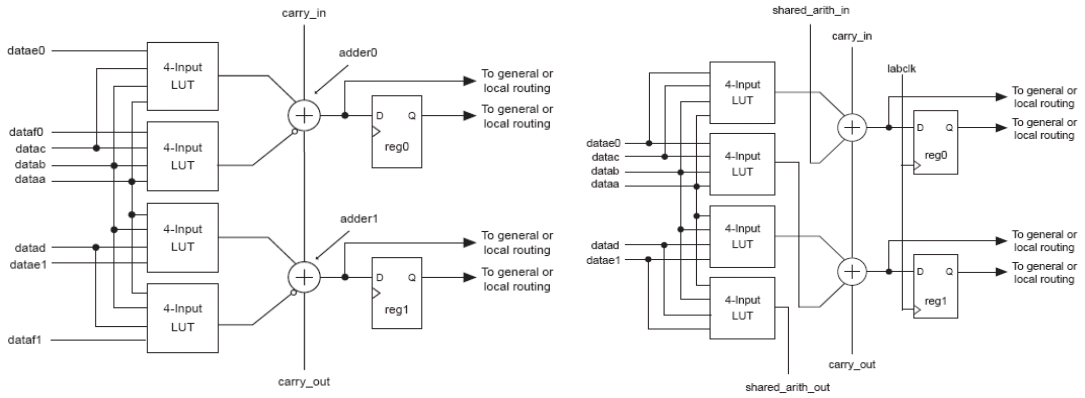


Figure 2.4: Logic block (ALM) structure of Altera Stratix II-V [7].

high-end FPGAs from both vendors have island style structure and consist of fracturable LUTs that are paired with the hard-logic—i.e., adders—and the stand-alone hard-logic—i.e., DSP and memory blocks. However, despite these similarities, the FPGAs from these two vendors are built differently. In the following, we will briefly review the architectural aspects of the Altera and Xilinx FPGAs that we use for our experiments and comparisons in this thesis. For the reviewing purpose, we pick the *Altera Stratix-III* and *Xilinx Virtex-5* FPGAs, which both are fabricated with the same process technology ($65nm$) and support 3-input (ternary) addition. Note that, the subsequent generations of these two FPGAs more or less have the same architecture, but they have been fabricated with more advanced process technology.

2.2.1 Altera Stratix-III

The logic cell that is employed in the Altera Stratix II-V series of FPGAs, is called *Adaptive Logic Module (ALM)* that is shown in Figure 2.4. Each ALM is comprised of two six-input



(a) ALM in *arithmetic* mode

(b) ALM in *shared arithmetic* mode

Figure 2.5: Modes of the ALM that use the carry chain [7]. *Arithmetic* mode is used for binary addition and subtraction and *shared arithmetic* mode is used for ternary addition.

LUTs (6-LUTs) with four shared inputs and shared configuration bits—the two 6-LUTs in the ALM should implement the same logic function. Additionally, the ALM contains a carry chain that performs efficient ripple carry addition, and by-passable flip-flops that facilitate either combinational or sequential circuits. The two 6-LUTs are also fracturable, meaning that each can be decomposed into two or more smaller LUTs. The ALM also includes a seventh input bit, but can only implement a selected set of seven-input functions.

The ALM has five operating modes, two of which use the carry chains, as shown in Figure 2.5. In *Arithmetic Mode*, see Figure 2.5a, each 6-LUT is decomposed into two independent 4-LUTs, which perform a small amount of pre-adder logic, followed by the carry chains. Arithmetic mode implements effective adders, (sequential) counters, accumulators, parity functions, and comparators.

In *Shared Arithmetic Mode*, see Figure 2.5b, the ALM is configured as a 3-input ripple carry adder, which is called *ternary adder*. The fracturable LUTs are configured as *Carry-Save Adders (CSAs)*—3:2 full-adder—and the adder circuitry in the ALM functions as the final adder. Using ternary adders, one can add multiple integers with fewer logic levels compared to binary adders. Note that the 6-LUTs in the ALM are decomposed into smaller LUTs of 3- and 4-inputs; only the smaller LUTs are shown in Figure 2.5.

The DSP block in Altera Stratix-III, see Figure 2.6, implements multiplication, multiply-add, multiply-accumulate (MAC), and dynamic shift functions efficiently. The natively supported multiplier bit-widths are 9, 12, 18, and 36 using the base 18×18 ASIC multiplier. Other bit-widths can be supported by combining these bit-widths. The DSP block has built-in addition, subtraction, and accumulation units to combine multiplication results efficiently. Each DSP block is comprised of two half-DSPs, as shown in this figure, and each half DSP block has four 18×18 multipliers.

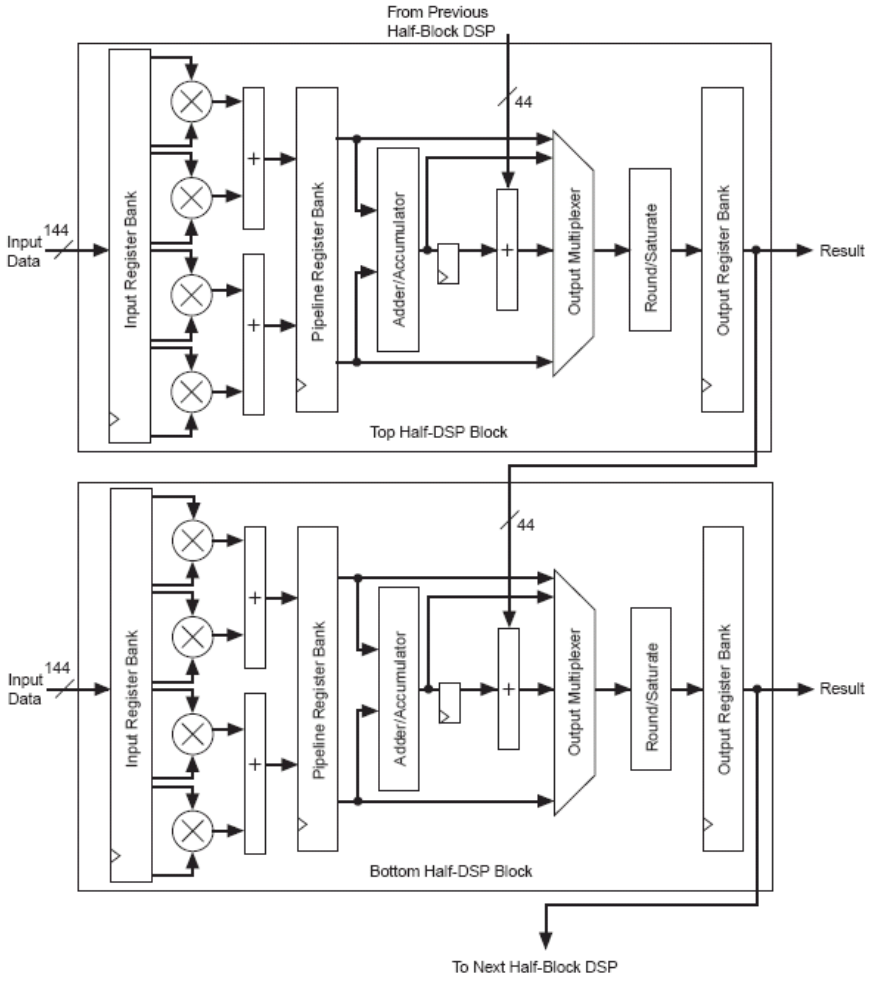
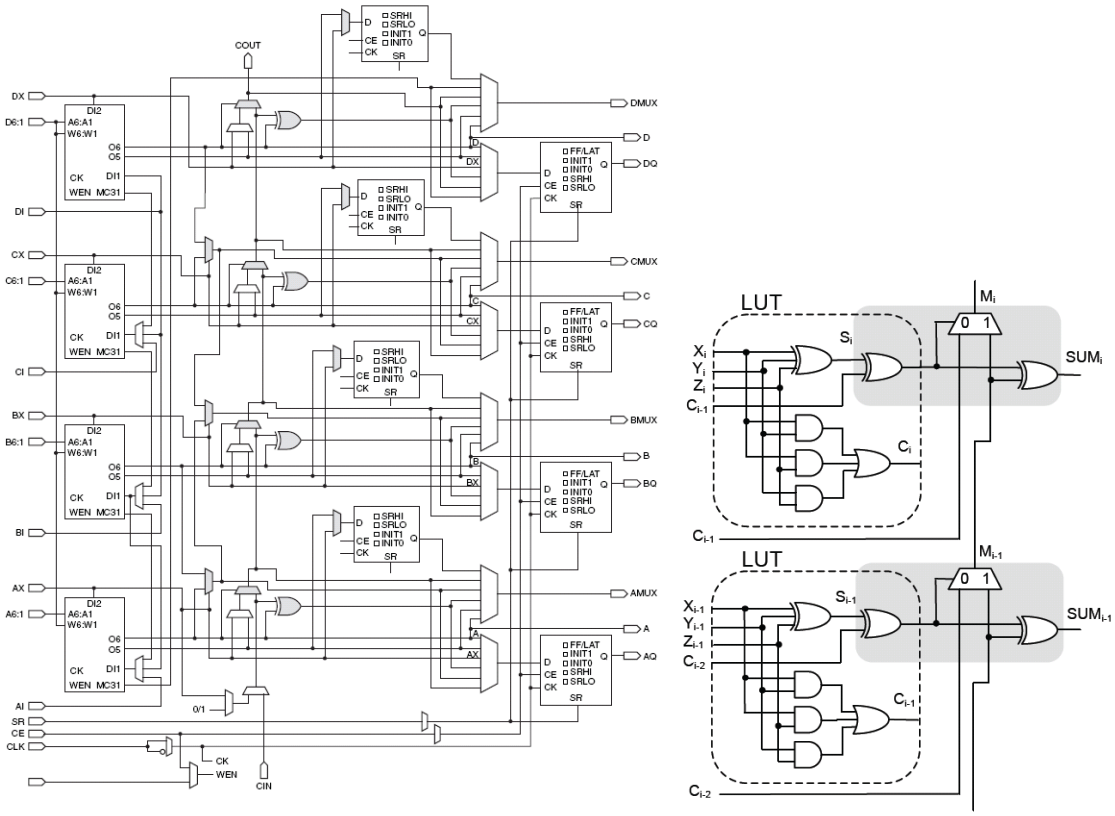


Figure 2.6: The DSP block structure of Altera Stratix-III [7].

2.2.2 Xilinx Virtex-5

The logic block of Virtex-5 is called *Slice* [92], as shown in Figure 2.7a. Each Slice contains four 6-LUTs, four XOR gates, and additional carry logic, including multiplexers. Each 6-LUT implements one 6-input logic function and LUTs do not share inputs. The 6-LUTs are also fracturable: each 6-LUT can be decomposed into two 5-LUTs, each of which can be configured to produce a separate logic function. The propagation delay through an LUT is independent of the function that it implements, or whether it implements one 6-input or two 5-input functions. Signals produced by an LUT can exit the Slice, drive the XOR gate, enter the carry chain, enter the select line of the carry-logic multiplexer (MUXCY), or drive the input of a flip-flop. The carry chains and XOR gates perform fast arithmetic addition and subtraction in a Slice. Slices are laid out to form columns. Carry chains can be formed that span all of the Slices in a column; that is, the carry chains are cascadable, permitting them to perform addition or subtraction on operands of arbitrary bitwidth.



(a) Slice architecture

(b) Slice configured as 3-input adder

Figure 2.7: Structure of logic cluster in Xilinx Virtex-5 [92]. The adder circuitry is composed of multiplexers and XOR gates. This logic cluster has four logic blocks that are equivalent to the Altera ALM.

Figure 2.7b shows a 3-input (ternary) adder synthesized on Virtex-5. Each LUT is configured as a full-adder, which produces a sum and a carry bit; this is similar to the carry save adder synthesized on the Stratix-III's ALM using Shared Arithmetic Mode. The second full-adder, shown in the shaded box, is formed by the conjunction of the same LUT with the XOR gate and multiplexer. The XOR gate's output represents the sum and the multiplexer output represents the carry. The C_i input to each LUT is the output of the previous LUT in the chain, which is connected by a routing wire; however, the design is structured so that C_i is dependent on the inputs to the full-adder and not on C_{i-1}. For this reason, the path that goes through the routing network is uncritical.

A pair of Slices forms a *Configurable Logic Block (CLB)*. The two Slices in a CLB do not connect to one another; each belongs to a different column and has an independent carry chain. Each CLB connects to a switch matrix for access to the routing network. There is no notion of a LAB-like logic cluster with fast local routing.

Figure 2.8 shows the architecture of the DSP block in Virtex-5, which supports many inde-

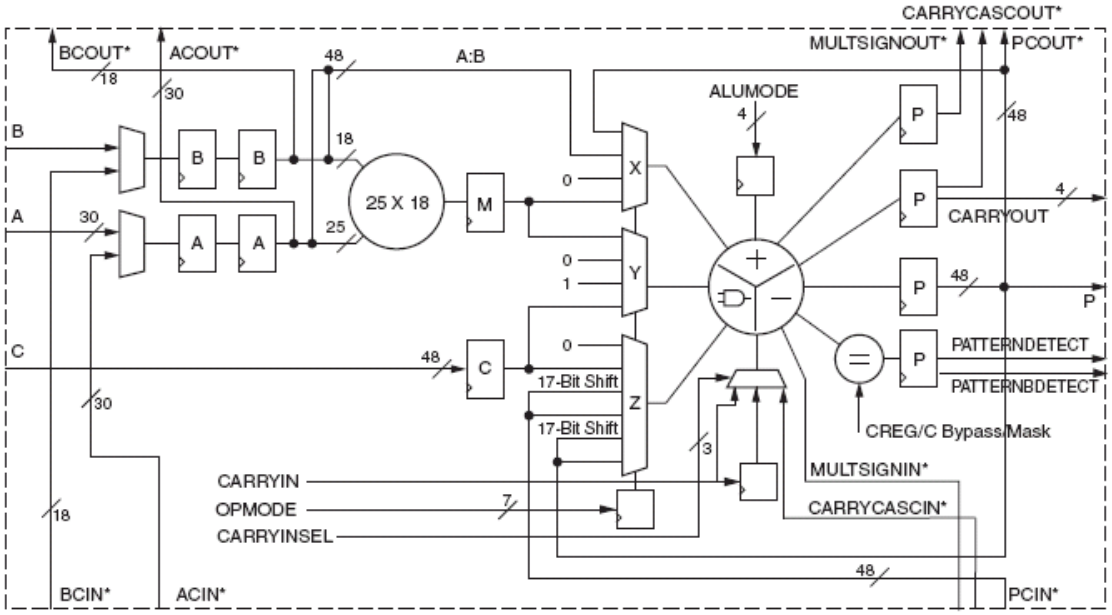


Figure 2.8: The DSP block structure of Xilinx Virtex-5 [92].

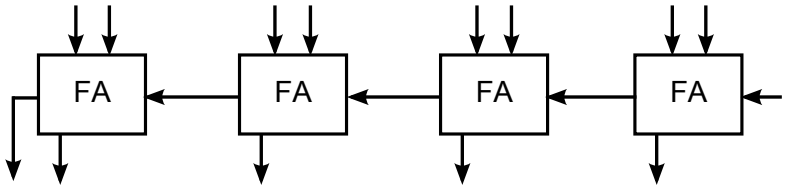
pendent functions. These functions include multiply, multiply accumulate, multiply add, three-input add, barrel shift, wide-bus multiplexing, magnitude comparator, bitwise logic functions, pattern detect, and wide counter. The architecture also supports cascading multiple DSP blocks to form wide math functions, digital filters, and complex arithmetic without using the soft-logic of FPGAs. Unlike to the Altera DSP blocks, there is less flexibility concerning the multiplier bit-width in the Xilinx DSP blocks.

2.3 Computer Arithmetic Preliminaries

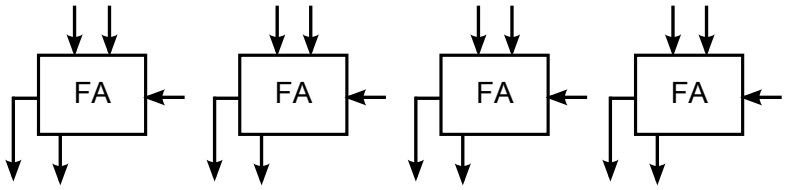
FPGAs are widely used to implement signal processing and multimedia applications, whose performance is dictated by the efficiency of the implementation of their arithmetic kernels. Dedicated arithmetic circuitries (hard-logic), such as multiplier-based DSP blocks and carry chains, have been embedded in modern FPGAs to improve the performance and logic density of these industrially relevant circuits. In this thesis, we expand the arithmetic functionality of the hard-logic of FPGAs, using mapping and architectural techniques. To better understand these contributions of the thesis, in the following section, we review the relevant computer arithmetic concepts.

2.3.1 Full- and Half-Adders

A *Half-Adder (HA)* is a 2-input, 2-output circuit that computes the sum of two bits and outputs the result as an unsigned binary integer. A *Full-Adder (FA)* computes a similar sum for three input bits. The lower-order output bit is called a *sum*, and the higher-order output bit is called



(a) ripple-carry adder



(b) carry-save adder

Figure 2.9: Ripple carry adder (RCA) versus carry save adder (CSA). In RCA, carry propagates through the full-adders, while in CSA, no carry propagation occurs.

a *carry*. In the case of an FA, one of the inputs is called a carry-in bit and the high-order output is called a carry-out. Many arithmetic circuits, including adders and multipliers are primarily comprised of HAs and FAs.

2.3.2 Ripple-Carry and Carry-Save Adders

A *Carry-Propagate Adder (CPA)* is a circuit that adds two binary integers; if the integers are signed, two's complement form is assumed. Numerous architectures for carry-propagate adders have been proposed in the past. In modern CMOS technologies, significant differences in critical path delay among the different adder architectures generally do not manifest themselves for small bitwidths, e.g., 8-bits or less.

The most straightforward CPA architecture is the *Ripple-Carry Adder (RCA)*, which generally has the smallest area but highest delay compared to the alternatives. Figure 2.9 shows a 4-bit RCA constructed from FA cells; the carry-in of the least significant FA is 0, so an HA can be used instead of an FA.

As shown in Figure 2.9a, an RCA is a 1-dimensional array of FAs, where the carry-out of each FA is connected directly to the carry-in of the next; thus, the worst-case critical path delay is through all of the FAs in the design. If an RCA adds two k-bit numbers, the complexity of the critical path delay is $O(k)$. Many faster, but larger, alternative adders have been designed, most with a critical path delay of $O(logk)$.

A *Carry-Save Adder (CSA)*, shown in Figure 2.9b, breaks the carry chain; in fact, it is a 1-dimensional array of *disconnected* FAs. CSAs are generally used in conjunction with CPAs in order to perform efficient n-input addition for $n > 2$.

2.3.3 Parallel Counters

An *m:n parallel counter* (or single-column counter) is a circuit that takes m input bits, counts the number of input bits that are set to 1, and outputs the value as an n-bit binary unsigned integer. The output range is $[0, m]$, so the number of output bits is

$$\lceil \log_2 (m+1) \rceil. \tag{2.1}$$

HAs and FAs are $2:2$ and $3:2$ counters, respectively. Verma and Ienne [87], for example, described an integer linear programming formulation for compressor tree design that uses a library of $m:n$ counters, for $2 \leq m \leq 8$.

Let $B = b_{k-1}b_{k-2}...b_0$ be a k-bit unsigned binary integer, where b_{k-1} is the most significant bit, and b_0 is the least significant bit. Each bit b_r contributes a total value of $b_r \cdot 2^r$ to the total value of B, i.e., b_r contributes 2^r, if it is set, and 0 otherwise. In this context, r is called the rank of b_r.

When an $m:n$ counter is used to synthesize a compressor tree, all of its inputs have the same rank. A *Generalized Parallel Counter (GPC)* is an extension of an $m:n$ counter that can sum bits of multiple ranks [82]. For example, a $(2,3;3)$ GPC can sum up to two bits of rank 1, and three bits of rank 0; the maximum output value is $2 \times 21 + 3 \times 20 = 7$, so three output bits are required. The general form of a GPC is $(k_{t-1}, k_{t-2}, ..., k_0; s)$, where k_r is the maximum number of bits of rank r that can be summed, and s is the number of output bits. Similar to an $m:n$ counter, the number of output bits of a GPC is

$$\lceil \log_2 (1 + \sum_{r=0}^{t-1} k_r \cdot 2^r) \rceil. \tag{2.2}$$

In fact, a sufficiently large $m:n$ counter can implement a GPC (although many other implementations also exist). Each GPC input bit of rank r is connected to 2^r inputs of the $m:n$ counter; any unused input bits of the $m:n$ counter are then driven to 0.

2.3.4 Compressors

Compressors (not to be confused with compressor trees) are arithmetic components, similar in principle to parallel counters, but with two distinct differences: (1) they have explicit carry-in and carry-out bits; and (2) there may be some redundancy among the ranks of the sum and carry-output bits.

The $4:2$ compressor (also called a $4:2$ CSA), illustrated in Figure 2.10, was introduced by Weinberger [89]; at first sight, this name may appear to be somewhat of a misnomer: although it has four input bits and produces two sum output bits (out_0 and out_1). It also has a carry-in (c_{in}) and a carry-out (c_{out}) bit (thus, the total number of input/output bits are five and three).

However, it is not the same circuit as a 5 : 3 compressor. All input bits, including c_{in}, have rank 0. The two output bits have ranks 0 and 1, respectively, while c_{out} has rank 1 as well. Thus, the output of the 4 : 2 compressor is a redundant number; for example, $out_1 = 0$ and $c_{out} = 1$ is equivalent to $out_1 = 1$ and $c_{out} = 0$ in all cases.

When k 4 : 2 compressors are connected in a carry chain, a total of $4k$ input bits are *compressed* down to $2k$ output bits plus one additional carry-out bit; the carry-in bit of the first compressor is set to 0. The primary difference between compressors and counters is the presence of carry bits in the former; it is also important to recognize that a compressor tree can be constructed from compressors, counters, or both.

Figure 2.10a shows the inputs and outputs of the 4 : 2 compressor labeled with their ranks; Figure 2.10b shows one 4 : 2 compressor architecture, which is constructed using two 3 : 2 counters. Figure 2.10c shows a 4-bit adder with four inputs, consisting of four 4 : 2 compressors in a 1-dimensional array followed by a 4-bit RCA. At first glance, the array of 4 : 2 compressors appears to have the same structure as an RCA, as the c_{out} bit of each 4 : 2 compressor is connected to the c_{in} bit of the subsequent one; however, this is not actually the case, as shown in Figure 2.10d; the fact that there is no direct path from a carry-in to a carry-out prevents the formation of a ripple-carry structure.

2.3.5 Adder and Compressor Trees

Suppose that we want to compute the sum of $n > 2$ binary integers. One approach is to use an *Adder Tree*, i.e., a tree of CPAs; the alternative is to build a tree of carry-save adders instead, only using a CPA at the end. Figure 2.11 shows an example where three 4-bit binary integers are added. In Figure 2.11a, two RCAs are used; in Figure 2.11b, a CSA is followed by an RCA. Let d_{FA} and d_{HA} are the respective delays of full- and half-adders. The critical path delay of the circuit in Figure 2.11a is $2 \cdot d_{AND} + 3 \cdot d_{AND.OR} + 2 \cdot d_{XOR}$, while the critical path delay of the circuit in Figure 2.11b is $2 \cdot d_{AND} + 3 \cdot d_{AND.OR}$, an overall saving of two d_{XOR} compared to Figure 2.11a. This savings occurs because the use of the CSA instead of the RCA permits the elimination of one bit from the RCA in Figure 2.11b. The idea of using carry-save addition for fast accumulation dates back to the work of Wallace [88] and Dadda [26], who designed fast parallel multipliers; however, the fundamental ideas generalize quite elegantly to multi-input addition as well.

Formally, let $A_1, ..., A_n, n > 2$, be a set of binary integers to sum. A *Compressor Tree* is a circuit that produces two values, *S (sum)* and *C (carry)*, such that:

$$S + C = A_1 + ... + A_n \tag{2.3}$$

A 2-input carry-propagate adder is then required to compute the sum of S and C. Since the high-end FPGAs from both Altera and Xilinx contain architectural support for 3-input (ternary) carry-propagate adders, it can be more efficient to design compressor trees that produce three

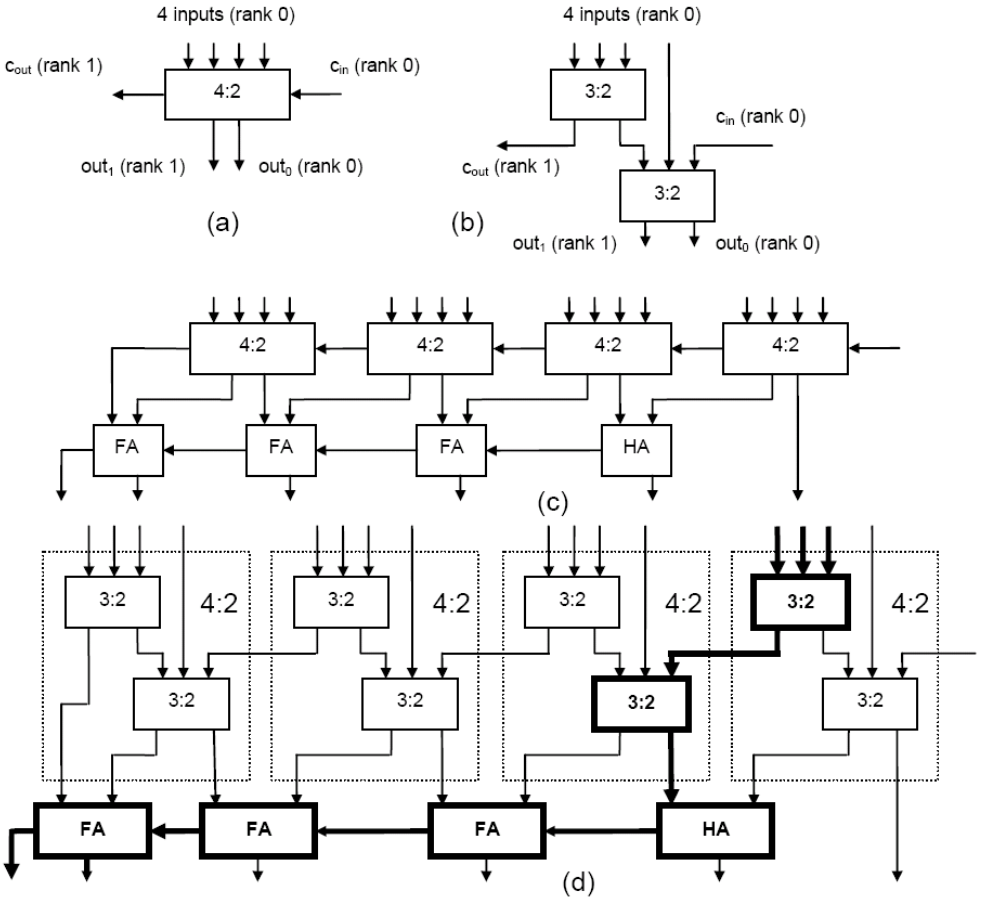


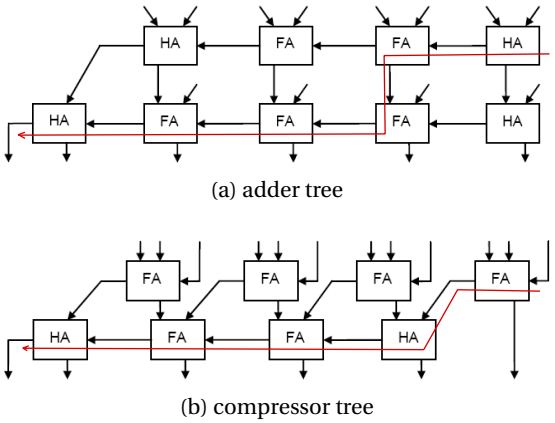
Figure 2.10: Example of an arithmetic compressor. (a) 4 : 2 compressor I/O diagram; (b) 4 : 2 compressor architecture; (c) 4-ary adder built from an array of 4 : 2 compressors followed by an RCA; (d) illustration of the interconnect between consecutive 4 : 2 compressors: although the array has the appearance of an RCA in Figure 2.10c, the carry chain only goes through two compressors.

values rather than two, e.g.:

$$S_2 + S_1 + C = A_1 + ... + A_n \tag{2.4}$$

Wallace and Dadda trees are two specific compressor tree architectures; many others have also been proposed [83, 82, 89, 78, 28, 81, 80, 85, 51, 62, 87].

In multi-input addition, the number of bits to sum at each position is the same. This is not true in the case of parallel multiplication, where the number of bits to sum tends to be greater among the bit positions in the middle. As illustrated conceptually by Figure 2.12, the lower-order bits of the final CPA are generally not on the critical path, as the bits that arrive at these positions go through fewer layers of logic within the compressor tree. In other words, the arrival time of the bits at the final CPA is nonuniform, unlike the case of multi-input addition.



(a) adder tree

(b) compressor tree

Figure 2.11: Adder tree versus compressor tree. Two implementations of a 4-bit ternary adder using (a) an adder tree, i.e., two RCAs; and (b) a compressor tree, i.e., a CSA followed by an RCA. The compressor tree implementation eliminates the delay of two XOR gates from the critical path.

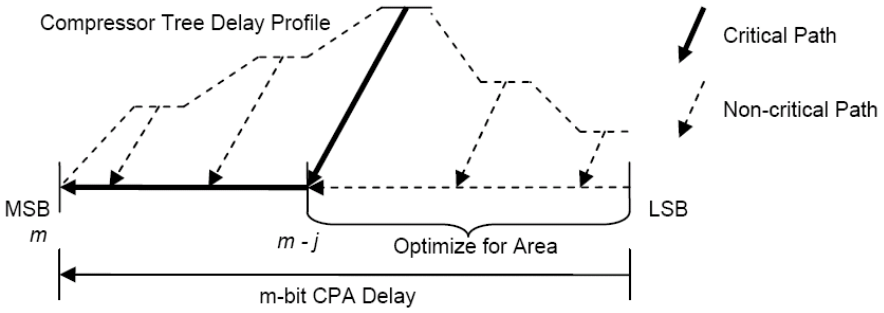


Figure 2.12: Illustration of the critical path delay through a compressor tree of a multiplier, including that of the final CPA. The critical path typically includes the j most significant bits of the final CPA; the portion of the final CPA that computes the $m - j$ least significant bits can be optimized for area rather than for speed, as long as it does not become critical.

Based on this observation, Oklobdzija and Villeger [63] argued that the final CPA of a multiplier should be implemented as a hybrid adder, which uses a small and slow CPA, such as an RCA, for the low-order bits, and a faster adder, such as a carry-select adder for the higher-order bits. *Carry-select adders* [73] are particularly useful when the arrival time of bits is nonuniform. Carry-select adders can start to add the bits as soon as they arrive. RCAs, in contrast, cannot, as the output bit at position i depends on the carry-out bit computed at position $i-1$. That being said, carry-select adders can be constructed from smaller bitwidth RCAs as building blocks.

The work summarized in this section targets ASIC design methodologies; FPGAs, in contrast, possess fast carry chains, whose usage often dictates the types of adders that perform well on specific device families.

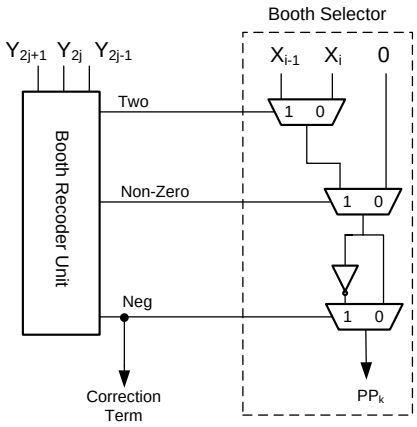


Figure 2.13: The PPG unit of the Radix-4 Booth multiplier. Based on the output of the booth encoder, the right PP is generated by the Booth selector. The input to the Booth encoder is the multiplier, Y, and the input to the Booth selector is the multiplicand, X.

2.3.6 Parallel Multipliers

Multiplication involves two basic operations: the generation of the partial products, which is called *Partial Product Generator (PPG)*, and their accumulation, which is called *Partial Product Reduction Tree (PPRT)*. In parallel multipliers, all *Partial Product (PP)* bits are generated in parallel and compressor trees are used for implementing the PPRT unit. There are two well-known algorithms for parallel multiplication, which are briefly described in the following.

Radix-4 Booth Multiplier

Radix-4 Booth [73] multiplication is a well-known multiplier design for 2's complement signed numbers, in which the number of PPs is half of the basic array multiplication schemes. In this method, the numbers that are multiplied are encoded and this way the number of PPs is reduced. The basic idea is to take every second bit and multiply by ± 1, ± 2, or 0, instead of shifting and adding for every bit of the multiplier term and multiplying by 1 or 0. To Booth-encode the multiplier term, the bits in blocks of three are considered, such that each block overlaps the previous block by one bit. The overlap is necessary so that we know what happened in the last block, as the MSB of the block acts like a sign bit.

As shown in Figure 2.13, the Booth algorithm is implemented into two steps: *Booth encoding* and *Booth selecting*. The Booth encoding step is to generate one of five values—± 1, ± 2, or 0—from the adjacent three bits of the multiplier, Y. The Booth selector generates a PP by utilizing the output signals of the Booth encoding.

Although Booth multiplier generates fewer PPs, its PPG unit is more complex than the other parallel multipliers, in which the PPG unit is comprised of simple 2-input AND gates. However, this complexity is absorbed by the large LUTs that are available in current FPGAs.

Baugh-Wooley Multiplier

Baugh-Wooley [73] multiplication is based on the standard shift and add multiplication method and is used for the multiplication of 2's complement signed numbers. The benefit of this multiplier is that the PPs are not sign extended. A Baugh-Wooley PPG for an $N \times N$ signed multiplier produces $N^2 + 1$ PP bits, some of which are computed using a NAND gate rather than an AND gate, and the most significant output bit of the multiplier is inverted. One of the partial product bits is set to the constant value 1.

Compared to the Radix-4 Booth multiplier, the number of PPs is doubled, and thus its PPRT unit is bigger and slower.

3 Mapping using Carry Chains

Based on the thesis *roadmap* that was presented in Chapter 1, we first focus on increasing the functionality of the *hard-logic* of FPGAs to provide the opportunity for more applications to take advantage of these dedicated resources.

As described in Chapter 1, there are two types of hard-logic in FPGAs: (1) the hard-logic that is tightly integrated with the soft-logic, such as adder circuitry and carry chains that are mixed with Look-Up Tables (LUTs), and (2) the stand-alone hard-logic, such as DSP blocks. Both types have limited flexibility and will be wasted, when they are not used. Coupling with the soft-logic, however, increases the chances of exploring unintuitive mapping techniques to reuse the hard-logic for unintended purposes. This is a cheap way to enhance the generality of the hard-logic of FPGAs, as it does not require any architectural modification; we only need to update the FPGAs mapping tools.

Typically, due to their frequent occurrence and criticality, arithmetic operations are the primary candidates for the hard-logic of FPGAs. Moreover, due to their regular structures—e.g., carry-propagate adders—it is easy to pair them with the soft-logic of FPGAs, which is inherently symmetric and regular. These are the reasons that most of current FPGAs have fast carry chains and adder circuitry in their logic blocks; the carry chains bypass the general routing network and are combined with the adder circuitry in the logic blocks, which enhances the implementation of the carry-propagate adders on FPGAs.

In this chapter, we present a mapping technique that exploits such dedicated resources for implementing carry-save based arithmetic circuits. In computer arithmetic literature [73], it is well-known that addition scales well when the number of inputs increases beyond two; this was first observed by Wallace [88] in the context of parallel multiplier design. The key is not to use trees of traditional carry-propagate adders, i.e., circuits that produce the sum of two (signed) binary integers. As introduced in Chapter 2, the integers are aggregated together using *compressor trees*. Carry-save arithmetic or more specifically compressor trees are fundamental for implementing many signal processing applications, and in contrast to conventional wisdom, we will show that it is possible to take advantage of the carry chains in

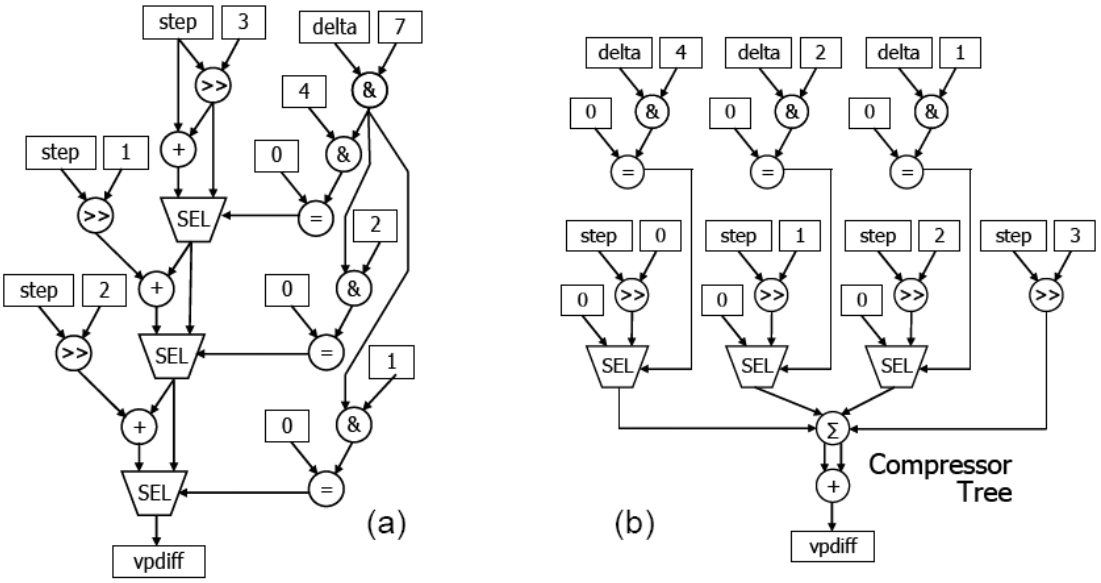


Figure 3.1: A kernel of the adpcm benchmark, originally written in C [53]. (a) a dataflow graph of the circuit is shown following if-conversion [2]; and (b) rewritten to merge three addition operations into a compressor tree [86].

FPGAs to improve this class of circuits.

3.1 Introduction

In Chapter 2, we introduced compressor trees. To recall, compressor trees are a class of circuits that generalizes multi-operand addition and the partial product reduction trees of parallel multipliers using *carry-save* arithmetic, in which long chains of carry-propagate addition are avoided. Multi-input addition occurs in many industrially relevant applications, such as FIR filters [60], the Sum-of-Absolute Difference (SAD) computation used in video coding [15], and correlators used in 3G wireless base station channel cards [79], among others. Verma et al. [86] introduced a set of data flow graph transformations to form compressor trees by merging disparate addition operations together and with the partial product reduction trees of multipliers used in the same computations—see Figure 3.1.

The superiority of compressor trees over adder trees for ASICs has been known since the 1960s [88, 26, 80]; however, these compressor tree synthesis methods yield poor results when circuits are synthesized on FPGAs. With the presence of fast carry chains, it has long been thought that trees of 2- or 3-input carry-propagate adders are more efficient than compressor trees for FPGA synthesis. The reason is twofold. First and foremost, it was thought the compressor trees are not efficiently synthesized onto LUTs. Secondly, the reduction in delay, which is achieved by the fast carry-chain that does not pass through the routing network, was thought to offset the superior arithmetic structure of a compressor tree.

In this chapter, we will show that, in contrast to the conventional belief, compressor trees can be mapped to FPGA logic blocks efficiently using a mixture of LUTs and dedicated adder circuitry. This chapter introduces a *hybrid* design method to synthesize compressor trees on FPGAs, which is capable of exploiting carry chains in limited and appropriate contexts. This method can be used to reduce the critical path delays of multi-operand adders, and multipliers whose partial product reduction trees have been fused with other adders; fixed-bitwidth multiplication and serialized multiply-accumulate operations can be synthesized on DSP blocks, which already contain embedded fixed-point multipliers.

Our experiments target the Altera Stratix-III and Xilinx Virtex-5 FPGAs. For both FPGAs, we observe significant reductions in critical path delay; for Virtex-5, area is reduced as well; for Stratix-III, the area depends on how the library components are prioritized. If the components are prioritized to reduce critical path delay, then the area of the compressor tree becomes larger than that of the adder tree; however, prioritizing for area or area-delay product achieve more conservative reductions in critical path delay, but achieve marginal area improvements compared to the adder trees. The experimental results show that, on average, compressor trees can reduce critical path delay by 25% and 30%, respectively, compared to adder trees synthesized on the Xilinx Virtex-5 and Altera Stratix-III FPGAs.

3.2 Hybrid Design Methodology

Our methodology to design compressor trees on FPGAs is a hybrid *top-down* and *bottom-up* design approach. First, we build a library of building blocks called *Generalized Parallel Counters (GPCs)*—refer to Chapter 2 for the GPC definition—whose implementations are highly tailored to the structure of LUTs and carry chains in FPGAs; this library construction phase is vendor and architecture-specific. Each GPC in the library is characterized in terms of the critical path delay of each output and its area. Second, a high-level greedy heuristic synthesizes the compressor tree in an architecture-agnostic manner, using GPCs from the library as building blocks.

Considering the fact that current FPGAs suffer from a slow routing network, the primitive (building) blocks should be coarse grained enough to increase the logic density and thus reduce the number of logic (block) levels, which minimizes the circuit depth and reduces the pressure on the routing network. This, indeed, is the main reason that the compressor primitives that are used in ASIC design are not appropriate for FPGAs, as they are too small, underutilize the FPGAs resources, and increase the logic depth of the design. Meanwhile, the coarse-grained primitives should not use any interconnect wire in their internal design; using carry chains are allowed, as they have almost null delay. This implies that the depth of the compressor tree built by these primitives will be equal to the number of the primitives that are placed in the critical path of the mapped design. Hence, our strategy to design these primitives is to increase the granularity of the primitive blocks until using routing wires becomes inevitable.

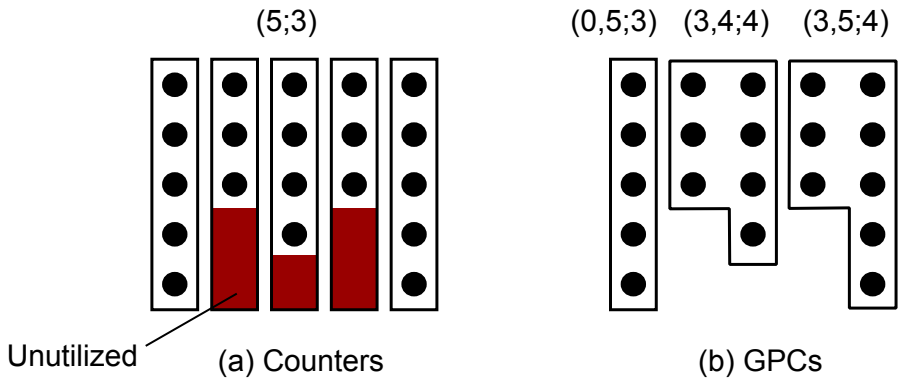


Figure 3.2: In the FPGA implementation, GPCs are more flexible and efficient than parallel counters for compressing bits. Fewer blocks are required to map the same bits, using GPCs as the mapping blocks. Here, we assume that the GPCs and counters have the same area and delay, when they are mapped on FPGAs.

The top-down part of the design includes a heuristic that covers the bits with the available primitives such that the depth of the design is minimized. This is a covering problem, in which bits should be covered with different primitives. In principle, the heuristic could be replaced by a more complicated method that achieves a solution of higher quality at the expense of increased runtime. Theoretically, integer linear programming can help to achieve optimum solutions. But in practice, since the primitive blocks are coarse grained and the design space is large, for bigger benchmarks it cannot converge to a solution.

In the following sections, we will describe the details of the bottom-up and top-down parts of the proposed design methodology.

3.3 Developing Compressor Tree Primitives for FPGAs

As described in the previous section, the first step for mapping compressor trees on FPGAs is to develop an FPGA optimized library, which contains compressor tree primitives. We discovered that Generalized Parallel Counters (GPCs) are the right arithmetic blocks that can be highly tailored to the structure of current FPGAs, using a mixture of soft- and hard-logic. There are two main reasons that we chose GPCs as compressor trees primitives for FPGAs. The first reason is that GPCs, as described in Chapter 2, are inherently flexible, which can cover bits with different bit positions. This allows to cover the input bits in a more efficient manner with fewer primitives. In fact, either the input bits or the ones that are generated at each compression level normally create irregular input bit pattern, and hence having a single inflexible primitive will result in utilization inefficiency, as it is shown in Figure 3.2. While, by using flexible GPCs the utilization problem is resolved. The second reason for using GPCs as primitives is that they are implemented very efficiently to the current FPGA logic blocks using a mixture of LUTs and the adder circuitry. In the following section, we will show how such primitives are mapped to current FPGAs.

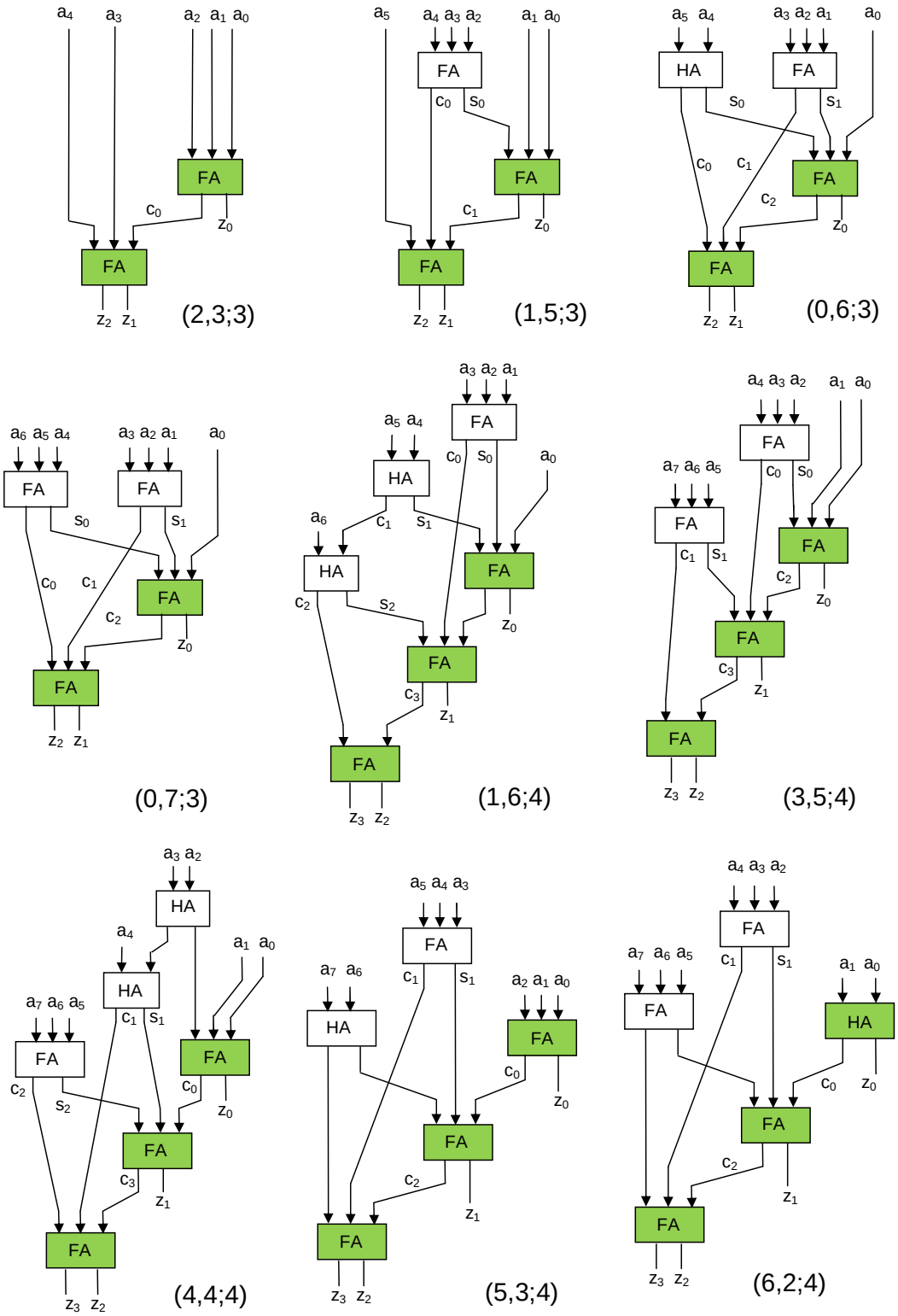


Figure 3.3: The covering GPCs listed in Tables 3.1 and 3.2 as networks of full- and half-adders. The shaded full- and half-adders are synthesized on the carry chains.

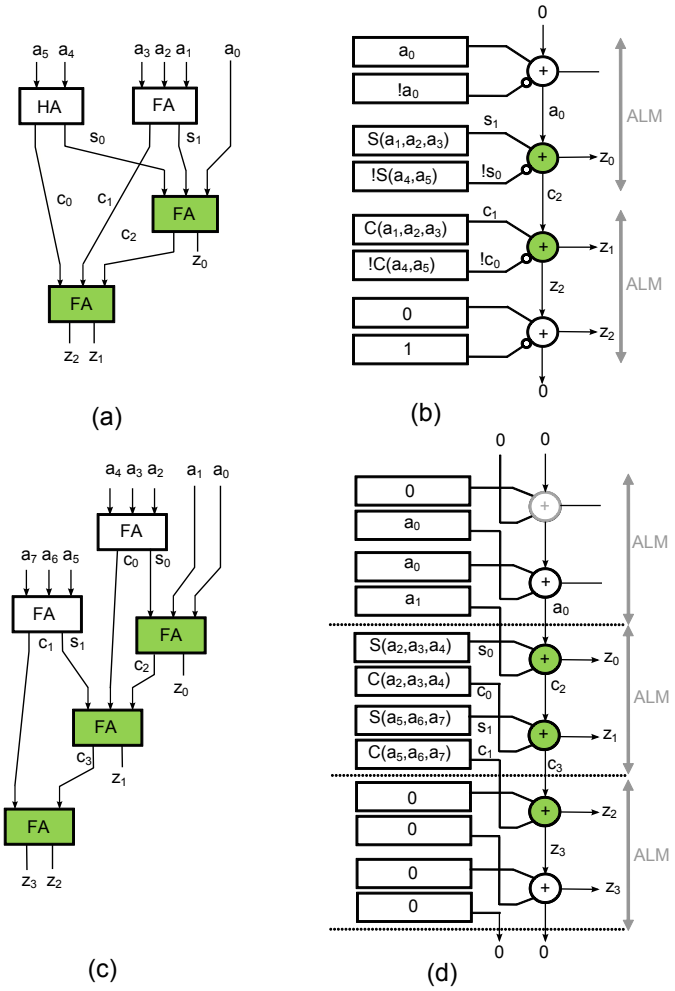


Figure 3.4: A (0,6;3) GPC implemented at the circuit level (a) and synthesized on ALMs and carry chains using Arithmetic Mode (b). A (3,5;4) GPC implemented at the circuit level (c) and synthesized on ALMs and carry chains using Shared Arithmetic Mode (d).

3.3.1 GPC Libraries

GPCs are architecture-specific and are designed manually. Each GPC is implemented twice: using only LUTs, and using a combination of LUTs and carry chains. Our approach is to model each GPC as a network of full- and half-adders. We identify *chains* of full- and half-adders within each GPC that are suitably mapped onto carry chains. The remaining full- and half-adders map onto LUTs. Figure 3.3 shows the adder-based designs of all *covering* GPCs—refer to Definition 1—in the library. Figure 3.4a depicts the representation of the (0,6;3) GPC, and Figure 3.4b shows its corresponding ALM-based implementation on the Stratix-III FPGA—refer to Chapter 2 for the ALM structure. For this GPC, the Arithmetic Mode of the ALM is used. The LUT-only implementation of the GPC requires three ALMs: one for each output bit. Similarly, the (3,5;4) GPC, represented as a network of FAs and HAs in Figure 3.4c, is implemented by three ALMs, configured in the Shared Arithmetic Mode, as shown in Figure 3.4d. The LUT-only

implementation of the same GPC requires five ALMs that are connected by routing wires, as the number of GPC inputs exceeds the number of LUT inputs; in contrast, a single layer of ALMs can implement the same GPC by exploiting the carry chain.

Both adders in Figure 3.4 need to route input bits directly to the carry-chain, bypassing LUTs. To accomplish this, we exploit the following property of full-adders:

Property 1. *A full-adder can be **partitioned** into two disjoint units, if the two inputs of the full-adder are bound together. In this case, the sum output will be equal to the carry input, and the carry output will be equal to the adder input.*

For example, in Figure 3.4b, we use two LUTs to route signal a_0 to two inputs of the first full-adder in the chain, and set the carry-input to 0. Property 1 then allows this full-adder to route a_0 to the carry-input of the next full-adder in the chain; the sum output, which is 0, is not used.

Similarly, both GPCs must route the carry output of the last full-adder in the carry chain to an ALM output; the sum output has a direct connection to the ALM output, but the carry output does not. Once again, we can exploit Property 1, by routing the carry output, e.g., z_2 in Figure 3.4b, to the carry-input next full-adder in the chain. We configure the LUTs so that the other inputs to the full-adder are both 0; this propagates z_2 to the ALM output through the sum output of the last adder in the chain. Moreover, this produces a carry output of 0. This effectively breaks the carry chain, so a new carry chain (with carry-input 0) can start at the following full-adder. Figure 3.5 shows a Slice-based implementation of $(0,7;3)$ GPC for Virtex-5—refer to Chapter 2 for the Slice structure. Figure 3.5a depicts the design built using a network of full- and half-adders, and Figure 3.5b shows the same design mapped onto one Slice. The LUT-only implementation of this GPC requires two Slices, because the GPC has seven inputs, while a Slice only has six. The adder chain that is selected for the mapping to the carry chain has been highlighted in both figures and the remaining adders are mapped to the driving LUTs. As mentioned in Chapter 2, part of the adder that is placed on the carry chain is implemented by the LUT as shown in Figure 3.5b. The other important feature of this implementation is that the carry output of the second LUT, c_1, is not dependent on the output bit of the first LUT, s_0; this prevents the formation of a multi-LUT critical path involving carry chains.

In Virtex-5, an input can access the carry chain at any point, but the most significant output, z_2, goes through the last multiplexer of the chain. One additional quarter-Slice is required to generate the GPC output. Since the multiplexer output drives an XOR gate, the other input is set to constant 0 to propagate the last GPC output to the Slice output.

We can map up to 8-input GPCs to the logic blocks of Stratix-III and Virtex-5 using the carry chain. The main constraint is that no routing wires are used within each GPC.

Definition 1. *A **covering** GPC is one whose functionality, given I/O constraints, cannot be implemented by another GPC.*

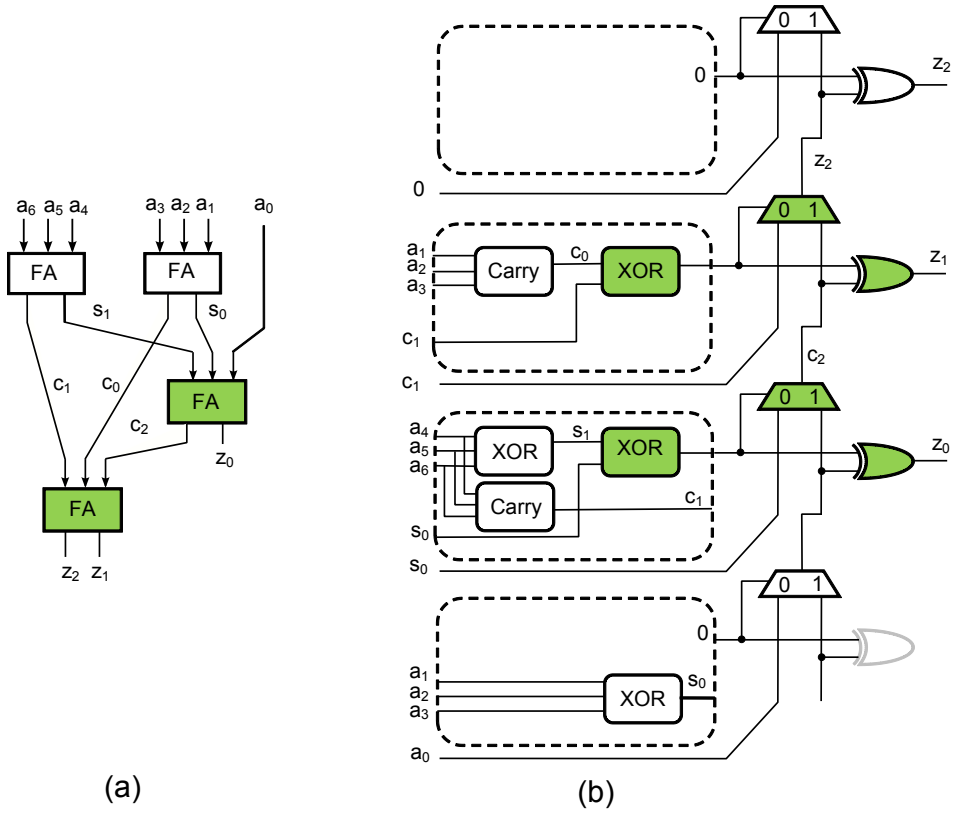


Figure 3.5: A (0, 7; 3) GPC implemented at the circuit level (a) and synthesized on a Virtex-5 Slice (b) using the carry chain.

For instance, a $(4,3;4)$ GPC is not a covering GPC: either a $(4,4;4)$ or $(5,3;4)$ can implement its functionality by setting an appropriate input bit to 0. The GPC library only contains covering GPCs. Table 3.1 summarize the GPC libraries for the Stratix-III and Virtex-5 FPGAs. Each GPC in the library can be implemented with only LUTs, or with LUTs in conjunction with carry chains, as discussed above. When a non-covering GPC is needed during compressor tree synthesis, the smallest covering GPC that can implement its functionality is always chosen.

For Stratix-III, GPCs using Arithmetic Mode are uniformly smaller than those built using only LUTs. For GPCs with six or fewer inputs, the LUT-only implementation is faster. For GPCs with more than six inputs, two layers of LUTs are required for the LUT-only implementation, while Arithmetic Mode can realize the same GPC using a single layer of LUTs in conjunction with a carry chain; thus, the latter is faster and smaller.

The Virtex-5 GPC library has different characteristics. For GPCs with six or fewer inputs, the LUT-only implementations are uniformly superior to the use of carry chains. For the $(0,7;3)$ GPC, the carry chain-based implementation is faster and smaller than the LUT-only implementation; for all remaining GPCs, the LUT-only implementations are faster, but larger, than the carry-chain based implementations.

	Altera Stratix-III				Xilinx Virtex-5			
	LUT-Only		Arithmetic		LUT-Only		Arithmetic	
GPCs	Delay	Area	Delay	Area	Delay	Area	Delay	Area
(0,6;3)	0.38	3	0.97	2	0.35	3	1.04	4
(1,5;3)	0.38	3	0.97	2	0.35	3	0.79	3
(2,3;3)	0.38	3	0.97	2	0.35	3	0.79	3
(0,7;3)	1.36	4	0.98	2.5	1.48	6	1.04	4
(1,6;4)	1.36	5	1.01	3	0.84	7	1.04	4
(3,5;4)	1.36	5	1.01	3	0.65	7	1.04	4
(4,4;4)	1.36	5	1.01	3	0.91	6	1.04	4
(5,3;4)	1.36	5	1.01	3	0.65	5	1.04	4
(6,2;4)	1.36	5	1.01	3	0.91	7	1.04	4

Table 3.1: Covering GPC libraries for the Stratix-III (left) and Virtex-5 (right) FPGAs. The delay unit is *ns* and the area unit for Stratix-III is *ALM* and for Virtex-5 is *LUT*.

3.3.2 Efficiently Packing Adjacent GPCs Along Carry Chains

Each Stratix-III ALM, for example, contains ten ALMs, but LAB inputs can only enter the carry chain at the first and sixth ALMs. As shown in Figures 3.4a and 3.4b, two GPCs can be *abutted*, because the carry-in and carry-out bits of each are not part of the GPC circuit. However, the placer in Altera's *Quartus II* software is unable to pack GPCs densely in a LAB, because it requires an explicit connection via a carry chain from one GPC to the next. Quartus II only instantiates two GPCs per LAB: one starting at the first ALM, and one starting at the sixth. For example, if two $(0,6;3)$ GPCs were synthesized, then just four of the ten ALMs in a LAB would be used.

Two GPCs that use the same configuration mode, e.g., (Shared) Arithmetic Mode, can share a half-ALM (ALUT) when abutted. Looking at Figures 3.4a and 3.4b, the first ALUT in a GPC produces no output, and the last ALUT receives no inputs. Property 1 allows the last ALUT of one GPC to be shared with the first ALUT of the next, as shown in Figure 3.6a. This allows a new GPC to start at any point along the carry chain, not just at the first or sixth ALM, and facilitates resource sharing between adjacent GPCs. Fortunately, Quartus-II was able to discern that the two GPCs are logically disjoint.

Referring back to Table 3.1, the GPCs with six or fewer inputs require two ALMs, but when n such GPCs are abutted, one ALUT is shared between each pair and therefore $3n+1$ ALUTs are used. We abut groups of GPCs that use up to five contiguous ALMs (half-LAB) and we must choose a value of n that satisfies $3n+1 \leq 10$. Therefore, we can abut up to three GPCs from the first group with shared LUTs in half of a LAB.

The Virtex-5 FPGA offers a more limited opportunity to share LUTs and carry chain resources between abutted GPCs; this technique only works for the $(0,7;3)$ and $(2,3;3)$ GPCs–any combination of these two GPCs—in Table 3.1. Figure 3.6b shows an example for two $(0,7;3)$ GPCs; in

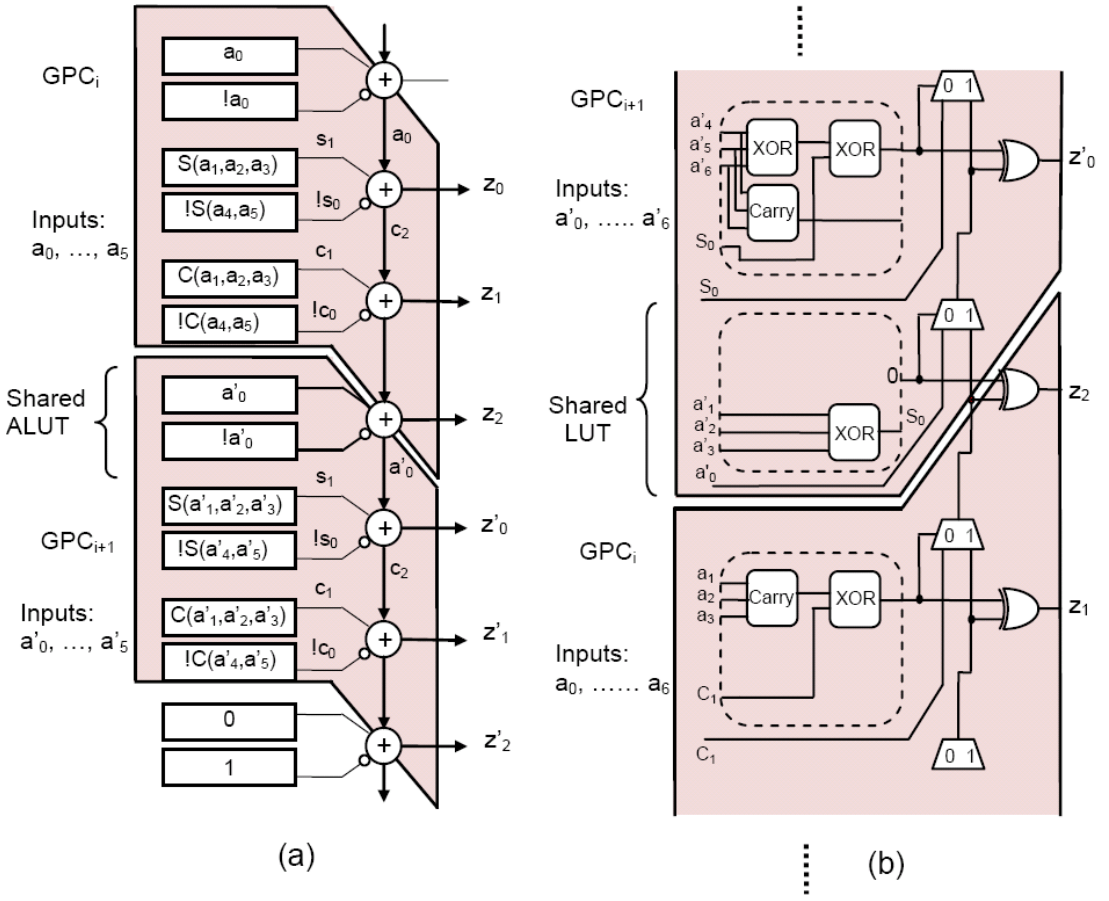


Figure 3.6: Example of abutting GPCs on the carry chains of FPGAs. (a) By abutting two (0,6;3) GPCs on Stratix-III, which are implemented using the same (Arithmetic) mode, an ALUT can be shared between two GPCs. (b) Two (0,7;3) GPCs on Virtex-5 are abutted by sharing on LUT. Only portions of both GPCs are shown to conserve space (b).

this case, the last LUT of the first GPC and the first LUT of the second GPC can be shared.

3.4 Compressor Tree Synthesis Heuristic

Given a GPC library, this section describes a heuristic to synthesize a compressor tree. The first step characterizes each GPC in the library in terms of its ability to reduce the number of bits at each stage. Second, a greedy heuristic generates the compressor tree.

3.4.1 GPC Library Characterization

The first step is to prioritize the GPCs in the library. We introduce four metrics for this purpose:

Definition 2. *The **Compression Difference (CD)** of a GPC is the difference between the number*

		Altera Stratix-III						Xilinx Virtex-5					
		LUT-Only			Arithmetic			LUT-Only			Arithmetic		
GPC	CD	PD	AD	APD	PD	AD	APD	PD	AD	APD	PD	AD	APD
(0,6;3)	3	7.9	0.9	2.4	3.1	1.8	1.9	8.5	1	2.9	2.9	0.6	0.6
(1,5;3)	3	7.9	0.9	2.4	3.1	1.8	1.9	8.5	1	2.9	3.8	1	1.3
(2,3;3)	2	5.3	0.6	1.6	2.1	1.2	1.5	5.5	0.6	1.7	2.5	0.6	0.8
(0,7;3)	4	2.9	0.8	0.6	4.1	2.9	3	1.2	0.6	0.2	3.8	1.3	1.2
(1,6;4)	3	2.2	0.6	0.5	3	1	1	3.6	0.4	0.5	2.9	0.7	0.7
(3,5;4)	4	2.9	0.8	0.6	4	1.6	1.6	6.2	0.5	0.8	3.8	1	0.9
(4,4;4)	4	2.9	0.8	0.6	4	1.6	1.6	4.4	0.6	0.7	3.8	1	0.9
(5,3;4)	4	2.9	0.8	0.6	4	1.6	1.6	6.2	0.8	1.3	3.8	1	0.9
(6,2;4)	4	2.9	0.8	0.6	4	1.6	1.6	4.4	0.5	0.6	3.8	1	0.9

Table 3.2: The CD value for each GPC, and the PD, AD, and APD values for the Stratix-III and Virtex-5 GPC libraries listed in Table 3.1. The GPC with the highest priority in each case has been highlighted.

of inputs and the number outputs.

Definition 3. *The **Performance Degree (PD)** of a GPC is the ratio* $PD = \frac{CD}{delay}$.

Definition 4. *The **Area Degree (AD)** of a GPC is the ratio* $AD = \frac{CD}{area}$.

Definition 5. *The **Area-Performance Degree (APD)** is the ratio* $APD = \frac{CD}{area \cdot delay}$.

The compression difference represents each GPC's ability to reduce the bits at each level of the compressor tree. For example, the compression difference of a $(2,3;3)$ GPC is $5-3=2$, while that of a $(6,2;4)$ GPC is $8-4=4$; thus, the latter is more effective.

The three objective criteria outlined above are used to sort the GPCs in a priority order. At each step, the heuristic traverses the prioritized list of GPCs and selects the first one that it can use in the situation. The *PD* criterion is used to optimize delay; *AD* is used to optimize area; and *APD* attempts to balance delay and area. Table 3.2 lists the *CD* value for each GPC, along with the PD, AD, and APD values resulted for each GPC listed in Table 3.1. The GPCs having the highest priority for the different design objectives have been shaded in Table 3.2; these GPCs are called base GPCs.

3.4.2 Compressor Tree Synthesis Heuristic

The input to the compressor tree synthesis heuristic is a set of bits of different ranks to sum. A *column* is a set of bits having the same rank, i.e., $column[i]$ is the number of input bits in the i^{th} column. As an example, Figure 3.7 shows the input representation of the multi-input adder part of a length-3 FIR filter using dot notation, where each dot depicts in a bit in a specific column that will be added; for this particular filter, $column[0] = 1$, $column[1] = 3$, etc.

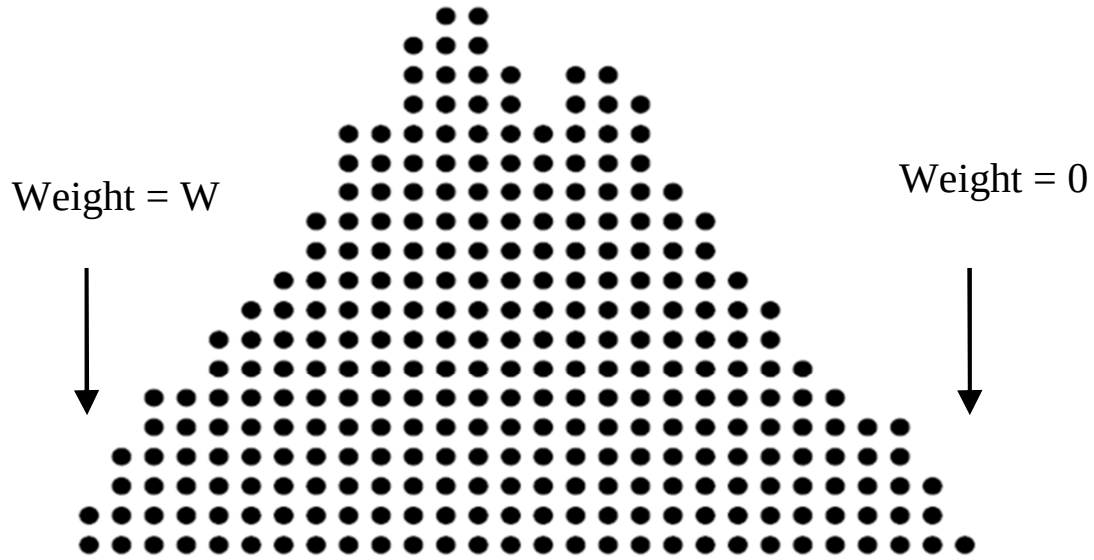


Figure 3.7: Adder tree dot representation of a sample FIR filter with three taps.

Additionally, the user specifies one of PD, AD, or APD as the optimization strategy, and the GPCs in the library are sorted accordingly.

Coincidentally, all of the base GPCs in Table 3.2 are m:n counters. To exploit this fact, the heuristic first covers all of the bits in $column[i]$ with as many base GPCs as necessary; some bits may be left over. For example, if the base GPC is $(0,6;3)$, and $column[i]$ contains eight bits, then one base counter would be used to cover the first six bits, and two bits would be left over. The output bits of each GPC must propagate to the correct columns. For example, a $(0,6;3)$ GPC that covers six bits in $column[i]$ will produce three output bits of rank i, $i+1$, and $i+2$; these bits must be added to $column[i]$, $column[i+1]$, and $column[i+2]$ accordingly.

The second step is to cover the remaining bits with counters. The heuristic traverses the columns from least to most significant, and selects an appropriate GPC for the library to cover each column. GPCs are considered in priority order, and the first feasible GPC is chosen. Columns containing three or fewer bits are skipped, because our target FPGAs support 3-input CPA. To bind a GPC to $column[i]$, two conditions must be met. Firstly, a GPC with exactly $column[i]$ bits in its least significant position must be chosen. Secondly, the subsequent column(s) should have at least as many bits as GPC inputs in that position. For example, suppose that $column[i] = 3$, and we are considering a $(2,3;3)$ GPC; then, it must have $column[i+1] > 2$ to satisfy this criterion. After a GPC has been chosen, the bits that it covers are removed from their respective columns. The process then continues, starting with $column[i+1]$ and stopping at the most significant column.

The third step generates the output bits for each GPC; as discussed previously, an M-output GPC whose least significant inputs cover bits from $column[i]$ generates one output bit for

column[i], *column*[$i+1$],..., *column*[$i+M-1$]. Output bits are generated after the covering process stops; this prevents the formation of carry chains at each level of the tree. For example, a GPC covering *column*[i] will produce an output bit for *column*[$i+1$]; we want that bit to be covered at the next level of the tree, rather than the current level.

Next, we connect the GPCs from the current level of the tree to those at the previous level. It is therefore important to track which GPC produces each output bit. When LUT-based GPCs are used, each output bit of the GPC has the same delay; however, when carry-chains are used, the least significant output bit produced by each GPC has a slightly lower delay than the second least significant output bit, etc., e.g., as illustrated by Figure 3.4b and Figure 3.5b for Stratix-III and Virtex-5, respectively. Similarly, input-to-output delays of certain inputs may be higher than others; for example, in Figure 3.4b, input bit a_0 clearly has the longest critical path. Thus, it is generally a good strategy to connect bits with higher arrival times to the GPC inputs with lower critical path delays at each level.

Both the Stratix-III and Virtex-5 FPGAs support native 3-input carry-propagate addition through their carry chains. If all columns contain three or fewer bits, then the compressor tree generation is complete, and all that remains is to connect the compressor tree outputs to a carry-propagate adder of appropriate bitwidth. If at least one column contains four or more bits, then another compressor tree level is generated, using the same technique outlined above.

3.5 Experimental Results

3.5.1 Experimental Methodology

First, we modelled each covering GPC in Table 3.1 at a low-level granularity. For Stratix-III, we performed *atom-level* modelling using the *Verilog Quartus Module (VQM)* format, as provided by *Altera's Quartus-II University Interface Program (QUIP)*. These models were used as components to construct larger compressor trees. The delay and area values reported here are taken from the Quartus-II project reports. For Virtex-5, we took a similar approach, using a Verilog-like format similar to VQM. Xilinx's ISE 10.1 CAD tools were used for all experiments targeting Virtex-5. The mapping heuristic was implemented in C++ using delay profiles for each GPC provided by the synthesizer. The input is a text file containing the number of bits per column. The output is a structural VHDL netlist of GPCs, forming the compressor tree, followed by a 3-input carry-propagate adder. The user specifies PD, AD, or APD at the command line.

3.5.2 Benchmarks

Table 3.3 summarizes the benchmarks used in our experiments, which are compressor trees taken from arithmetic circuits and DSP and video processing applications. DCT [77], H.264

Benchmark	Description
dct	Multiplierless DCT
hpoly	Horner polynomial Eval.
H.264 ME	H.264 motion estimation
g721	G.721 encoder
fir3, fir6	3- and 6-tap FIR filters
m9x9, m18x18, m24x24, m36x36	Parallel signed multipliers
add2I, add2Q	Video mixer components

Table 3.3: Benchmark summary.

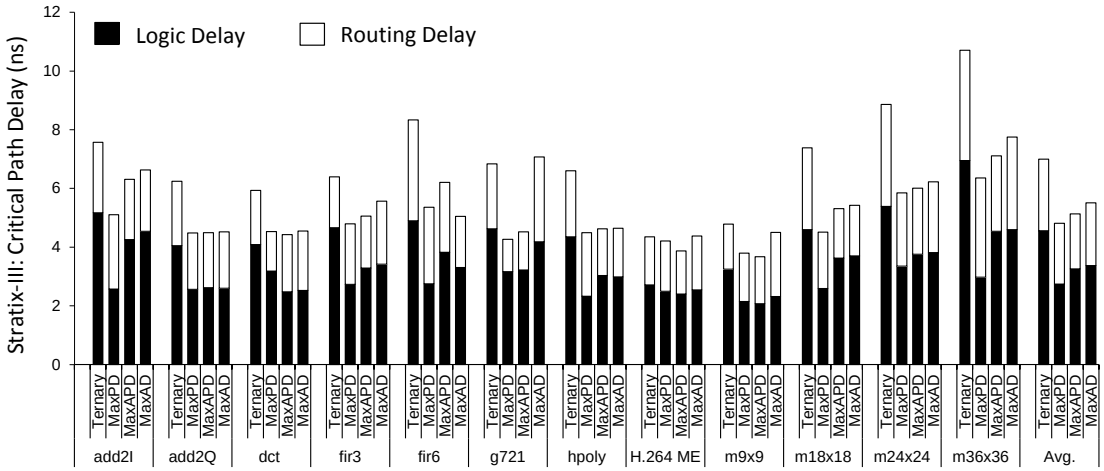


Figure 3.8: The critical path delay of Ternary, MaxPD, MaxAPD, MaxAD decomposed into logic and routing delay after synthesis on Stratix-III.

ME [15], fir3, fir6 [60], m9x9, m18x18, m24x24, and m36x36 naturally contain compressor trees. HPoly, G.721 [53], and Video Mixer [84] are transformed to expose large compressor trees [86]. Most of these benchmarks are publicly available with a few exceptions. The FIR filters were built using randomly generated constants; and Video Mixer is provided by Synopsys Corporation as an example to illustrate their *Behavioral Optimization of Arithmetic (BOA)* feature. Video mixer contains several distinct compressor trees and we use two of them. Each benchmark is synthesized as a purely combinational circuit, using four different approaches: *Ternary* uses a tree of three-input adders, *MaxPD*, *MaxAD*, and *MaxAPD* use the compressor tree synthesis heuristic with GPCs prioritized by PD, AD, and APD, respectively. The approaches are evaluated and compared in terms of critical path delay and area.

3.5.3 Results: Stratix-III

Figure 3.8 and Figure 3.9, respectively, report the critical path delay and area for each benchmark using each of the four synthesis methods for the Altera's Stratix-III FPGA.

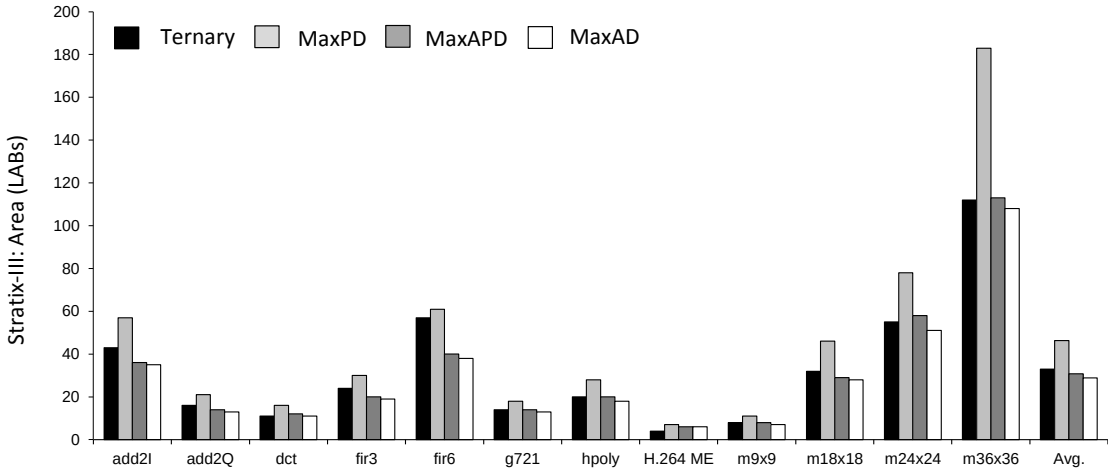


Figure 3.9: Area usage (LABs) of the four synthesis methods on Stratix-III.

Figure 3.8 shows that that Ternary has the maximum critical path delay for all benchmarks except *g721*. For *g721*, MaxAD has a slightly larger critical path delay than Ternary, but the critical path delays of both MaxPD and MaxAPD, which include critical path delay as part of the GPC prioritization scheme, are significantly smaller. On average, MaxPD is 30% faster than Ternary.

Logic delays, i.e., the delay through LUTs and carry chains, rather than routing delay, is the primary reason that Ternary has greater logic delay than the compressor trees; this is due, primarily, to the long ripple-carry chains that are formed using the Stratix-III ALM's Shared Arithmetic Mode. Logic delay is more prominent for benchmarks having the greatest height, such as *add2I* and *m36x36*. On the other hand, benchmarks such as *g721* and *H.264 ME* have shorter adder trees, but wide-bitwidth final carry-propagate adders; thus, they exhibit little disparity between adder and compressor trees in terms of delay.

MaxPD achieves the smallest critical path delay for most benchmarks, as it uses the LUT-only $(0,6;3)$ base GPC having highest priority. The delay of this GPC is less than that of the base GPCs for MaxAPD and MaxAD; this explains MaxPD's advantage in terms of delay. On average, MaxAPD's critical path delay is 7.6% greater than MaxPD's.

In terms of area, both MaxAPD and MaxAD require fewer LABs than Ternary, most notably *fir3* and *fir6*. MaxPD uses more LABs than the other compressor tree synthesis methods, because its base $(0,6;3)$ GPC has a large LUT-only implementation–requires three ALMs—compared to the $(0,7;3)$ base GPC of use by MaxAPD and MaxPD, which requires 2.5 ALMs when implemented with carry chains. Ternary requires more LABs than MaxAPD and MaxAD because each LAB has limited input bandwidth, which inhibits the ability to use all ALMs in a LAB to implement a Ternary adder. Each LAB contains ten ALMs, and six inputs per ALM are used in Shared Arithmetic Mode, so 60 inputs are required to implement a 10-bit 3-input adder in a LAB, but the actual LAB bandwidth is less than 60 and we observed that only half

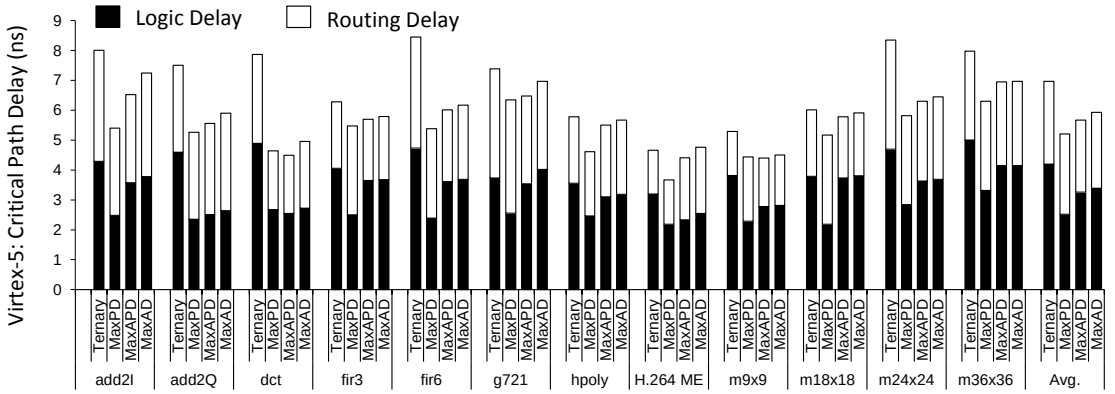


Figure 3.10: The critical path delay of Ternary, MaxPD, MaxAPD, MaxAD decomposed into logic and routing delay after synthesis on Virtex-5.

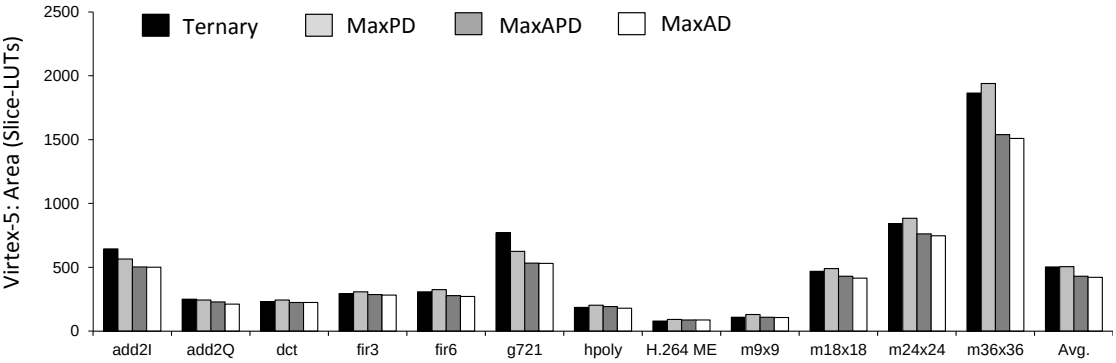


Figure 3.11: Area usage (LUTs) of the four synthesis methods on Virtex-5.

of the ALMs in the LAB are used; in contrast, it is possible to fit six 6-input GPCs into a LAB, requiring only 36 inputs, which is below the LAB input bandwidth.

To summarize, MaxAPD offers the best trade-off between critical path delay and area usage. MaxPD should only be used when the compressor tree constrains the critical path delay of the entire system and there are extra LABs to spare. MaxAD is clearly the best synthesis mode if area reduction is a priority.

3.5.4 Results: Virtex-5

Figure 3.10 and Figure 3.11, respectively, report the critical path delay and area for each benchmark using each of the four synthesis methods for the Xilinx's Virtex-5 FPGA.

Like Stratix-III, Ternary has the largest critical path delay for most benchmarks, although there are some exceptions such as *hpoly*, *ME*, and *m18x18*, where MaxAD and Ternary are approximately equal. MaxPD achieves the smallest delay among all benchmarks other than *dct* and *m9x9*. The disparity of the critical path delays between MaxPD, MaxAPD, and MaxAD

is smaller for Virtex-5 than Stratix-III; this is due to the differences in the architectural features, such as different carry chains and unavailability of LAB-like clusters of LUTs with fast local routes in Virtex-5. On average, MaxPD is 25%, 8% and 12% faster than Ternary, MaxAPD and MaxAD, respectively.

Like Stratix-III, Ternary has the largest logic delay, due to ripple-carry propagation. The base GPC for MaxPD is a $(0,6;3)$ GPC implemented using LUTs alone, while MaxAPD and MaxAD use a $(0,7;3)$ GPC implemented using LUTs and carry chains as their base GPCs; thus, MaxAPD and MaxAD have larger logic delays than MaxPD.

The area results are somewhat different than Stratix-III; in particularly, MaxPD does not suffer from particularly poor area utilization for Virtex-5. On average, Ternary and MaxPD are comparable in terms of area and thus, MaxPD is preferable due to its reduced critical path delay; while MaxAPD and MaxAD achieve less pronounced, but noticeable, average improvements in area, while remaining comparable to one another.

To summarize, critical path delay, and not area, distinguishes the four synthesis heuristics for Virtex-5. Ternary is not competitive in terms of critical path delay for Virtex-5, while offering no area advantage, unlike Stratix-III. On average MaxPD achieves slightly better critical path delays than MaxAPD and MaxAD, but uses slightly more area; MaxAPD and MaxAD are comparable in terms of both critical path delay and area. If the compressor tree lies on the critical path of a design, then MaxPD should be used; otherwise, MaxAPD or MaxAD would be more appropriate choices.

3.5.5 Integer Linear Programming (ILP)

To assess the quality of the heuristic, we compared it with an *Integer Linear Program (ILP)* that computes a near-optimal compressor tree implementation. Ideally, one could formulate the ILP to exploit the GPC delay and area information reported in Table 3.1, combined with all packing possibilities discussed in Section 3.3.2 to obtain an overall optimal solution for any of the three design objectives; however, we found that the search space is too large and that the ILP does not converge in a reasonable amount of time, as opposed to our previous work [67], where we had smaller GPCs.

3.6 Related Work

3.6.1 Compressor Tree Synthesis for FPGAs

Conventional wisdom has held that adder trees are superior to compressor trees on FPGAs. For example, Altera's manual for Stratix II notes that the Shared Arithmetic Mode of the ALM was introduced to facilitate implementation of adder trees using 3-input, rather than 2-input, adders [7], which reduces the height of an N-input adder tree from $\lceil log_2^N \rceil$ to $\lceil log_3^N \rceil$. Our experiments have shown that compressor trees can be more effective than adder trees

in practice, and that Shared Arithmetic Mode can actually be quite useful for constructing compressor trees.

Poldre and Tammemae [74] developed a method to synthesize a 4 : 2 compressor on the Xilinx Virtex Slice architecture; a layer of 4 : 2 compressors can reduce four integers to two without carry propagation [89]. Similarly, 4 : 2 compressors have also been synthesized on the Slice architecture used in Xilinx Virtex-2 and -4 and Spartan-2 and -3 [64, 44], Altera Cyclone III [44] and Altera's Adaptive Logic Module (ALM) [65]. A 4 : 2 compressor only requires four input bits per LUT, which is a good implementation choice for older and lower-end FPAs whose logic [65] cells are based on 4-LUTs with carry chains; however, as modern FPGAs now contain fracturable 6-LUTs [41] better I/O utilization can be achieved using the larger GPCs advocated in this work.

Three prior pieces of work have proposed to synthesize compressor trees on FPGAs using LUT-only GPCs [66, 67, 59]. The experimental results (MaxPD) in this chapter indicate that this approach improves critical path delay, but requires more resources compared to adder tree (Ternary) implementation, and, as a consequence, may not yield an ideal design choice.

3.6.2 Compressor Tree Synthesis for ASICs

Compressor trees were introduced as an efficient method for partial product reduction for parallel multipliers using networks of full- and half-adders [88, 26]. This was soon followed by the notion of parallel counters [83] and GPCs [82], which are constructed using full- and half-adders as building blocks. An optimal compressor tree synthesis algorithm, which repeatedly chooses the three bits from each column having the smallest arrival times, and connects them to the input of a full-adder, was introduced more recently [80, 85]. The design of the final adder is then tailored to the delay profile of the compressor tree outputs [63, 81]. These approaches do not work well for FPGAs for two reasons: (1) routing delays are difficult to predict during synthesis; and (2) the presence of carry chains influences the structure of both GPCs and the final adder.

3.7 Conclusion

In this chapter, we introduced a new approach for the synthesis of compressor trees on commercial high performance FPGAs. For this purpose, we exploit the carry chains and adder circuitry that exist in current FPGA logic blocks. The approach involves two stages: an architecture-specific GPC library instantiation phase, and an architecture-agnostic compressor tree synthesis phase, which constructs a compressor tree using components from the library. The results of experiments, targeting both FPGAs from Altera and Xilinx, indicate that compressor trees can improve both critical path delay and area compared to the existing state of the art: synthesis of multi-operand addition using trees of 3-input adders.

Altogether, the compressor tree synthesis algorithm presented here can help users and CAD tool developers to improve the arithmetic capabilities of FPGAs. This is a cheap way of increasing the functionality of the current hard-logic that is coupled with the soft-logic of FPGAs. The consequence is that several new applications, which are based on the compressor trees, can be enhanced using these dedicated resources.

Despite the low-cost advantage of the presented mapping technique, the achievable improvement is limited, due the fact that the carry chains are exclusively designed for carry-propagate adders. The mapping contribution, however, helped us to explore the hardware constraints of the current architecture; using this experience, in the next chapter, we will present a new architecture for the FPGAs logic block, which has non-propagating carry chains, in addition to the current propagating carry chain. This new chain requires few new resources and is optimised for compressor trees.

4 Non-propagating Carry Chains

In the previous chapter, we presented a mapping technique to increase the functionality of the hard-logic that is coupled with the soft-logic of FPGAs–i.e., carry chains and adder logic. Adding more functionality to the hard-logic of FPGAs follows the first direction of the thesis *roadmap*, in which the goal is to increase the number of applications that can take advantage of the dedicated resources of FPGAs. The mapping technique of the previous chapter does not require any hardware modifications, and thus can be immediately employed for the current FPGAs. However, the mapping challenges motivated us to revise the structure of the logic block (slightly), which can enhance both performance and area of the compressor trees on FPGAs.

In this chapter, we present a new FPGA logic block, which has new carry chains optimized for carry-save arithmetic. In contrast to the existing carry chains of FPGAs, our proposed carry chains have non-propagating nature. To support these carry chains, we add a few dedicated gates to the existing resources. Moreover, these new resources are added in a way that the original functionality of the logic block is maintained, and minimum overhead is imposed. To map on this new logic block, we extend the mapping heuristic of the previous chapter to take advantage of the new hard-logic structure.

4.1 Introduction

Previously, in this thesis, we argued that a compressor tree is a fundamental arithmetic structure that needs to be enhanced on FPGAs. Generally, the architecture of modern FPGAs is not well-suited to compressor trees. The logic clusters of the recent high-end FPGAs from *Altera* and *Xilinx* can be configured to implement ternary (3-input) addition using fast carry chains. The primary advantage of the carry chains is that the carry bits are propagated directly from one cell to its adjacent neighbor, thereby avoiding the overhead of the routing network. This design point favors the use of ternary adder trees rather than compressor trees.

In Chapter 3, we showed that compressor trees can be synthesized on FPGAs carry chains

using a circuit called *Generalized Parallel Counter (GPC)*. The GPC Mapping approach yields compressor trees whose delays are significantly lower than ternary adder trees, despite the latter's use of the carry chains; however, there is some noticeable increase in the number of logic blocks required. The main reason is that current carry chains in FPGAs have not been designed to support carry-save arithmetic, and this limits the achievable enhancement.

In this chapter, we introduce new carry chains for FPGAs that are tailored for compressor trees and require slight modification of the original logic block structure. This new logic block is the revised version of the Altera ALM logic block—refer to Chapter 2 for the ALM structure—which has additional carry chains and can be configured as a 7 : 2 compressor; this compressor belongs to a well-known class of circuits that have been used for successful synthesis of ASIC multipliers in the past [89, 78, 63]. The 7 : 2 compressor has fast carry-chains and are constructed from dedicated adders, similar to those used for ternary addition in modern FPGAs. Unlike prior carry chains, however, the carry chains in 7 : 2 compressor does not propagate beyond two logic blocks.

By combining the strengths of the GPC mapping with the use of 7 : 2 compressors, when possible, faster compressor trees can be realized on the FPGA. In the experiments, we observed that the compressor trees are approximately 35% faster than ternary adder trees, and they slightly require more resources.

4.2 Compressors

The new FPGA logic block, which is described in this chapter, can be configured as a 7 : 2 compressor; the 7 : 2 compressor generalizes the 4 : 2 compressor cell, which was introduced in Section 2.3.4 of Chapter 2. These compressors—not to be confused with compressor trees—are arithmetic constructs that have explicit carry-in and carry-out bits, and when they are chained through the carry bits, no ripple carry propagation occurs in contrast to carry propagate adders. Figure 4.1a shows the basic I/O structure of the 7 : 2 compressor. This compressor has seven inputs, two outputs, two carry inputs, and two carry outputs. All the inputs, including the carry bits, have the same rank, 0; like all compressors, there is redundancy between the carry outputs and normal outputs, as two of them have the same rank, 1.

Figure 4.1b shows the circuit-level architecture. In total, five 3 : 2 counters are used in the structure of the 7 : 2 compressor. The remarkable feature of this circuit is the fact that no logical path exists between the carry inputs and carry outputs of the compressor. This implies that no carry-propagation occurs when several compressors are chained.

Figure 4.1c shows the interconnect structure, when 7 : 2 compressors are chained. Consider the i^{th} compressor in sequence. The rank 1 carry output bit ($c_{out,0}$) connects to carry-input $c_{in,0}$ of the $(i+1)^s t$ compressor; also, the rank two carry output bit ($c_{out,1}$) connects to carry-input $c_{in,1}$ of the $(i+2)^{nd}$ compressor.

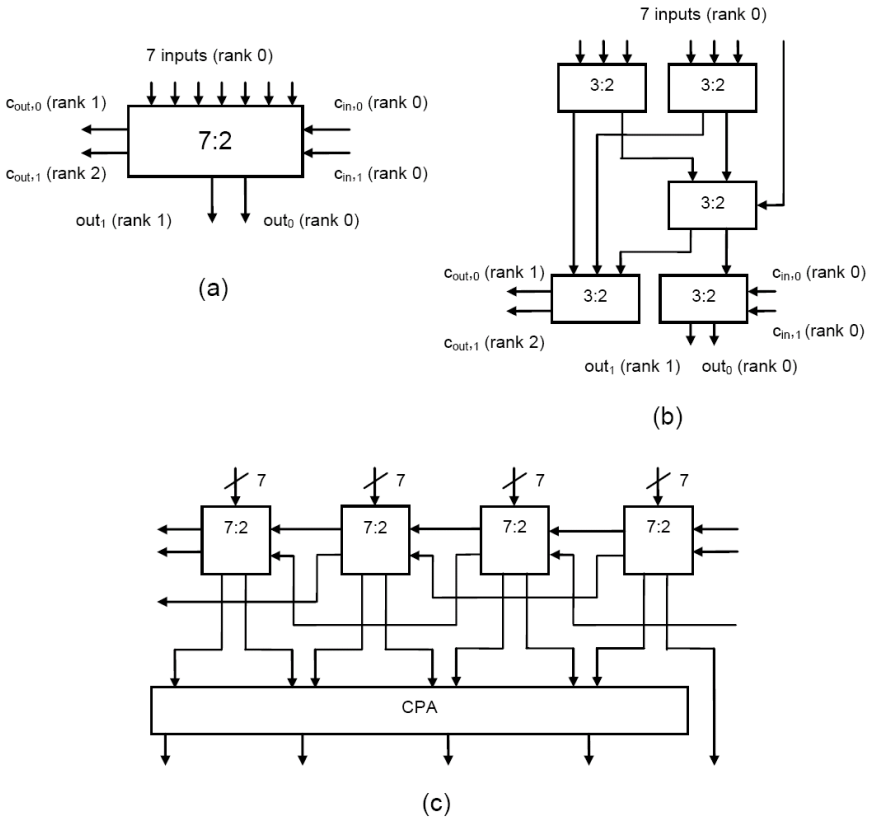


Figure 4.1: (a) 7 : 2 compressor I/O diagram; (b) 7 : 2 compressor architecture; (c) illustration of the interconnection pattern between consecutive 7 : 2 compressors.

4.2.1 Compression Ratio

Let I and O be the number of inputs and outputs produced by a counter, GPC, or compressor; for compressors, I and O do not include the carry-in and carry-out bits. The *Compression Ratio (CR)* is defined as $CR = I/O$. For example, a 7-input, 3-output GPC has $CR = 7/3 = 2.5$, while a 7 : 2 compressor has $CR = 7/2 = 3.5$. The CR tends to be higher for compressors than counters. Figure 4.2a shows compression using 7 : 3 counters; which produce three output bits per column, while 7 : 2 compressors, shown in Figure 4.2b, produce two output bits per column; the other output bits are propagated down the carry chain.

4.3 Logic Block Design

Figure 4.3 shows our proposed new FPGA logic block, which is presented as an extension of the ALM used in the Altera's Stratix II-V line of high-end FPGAs. The components required for Shared Arithmetic Mode—refer to Chapter 2 the operating modes of the ALM—are also shown in this figure. The left-hand side of Figure 4.3a shows four 3-LUTs, which are part of Altera's *fracturable* 6-LUT architecture. The carry chain on the right-hand-side is the traditional

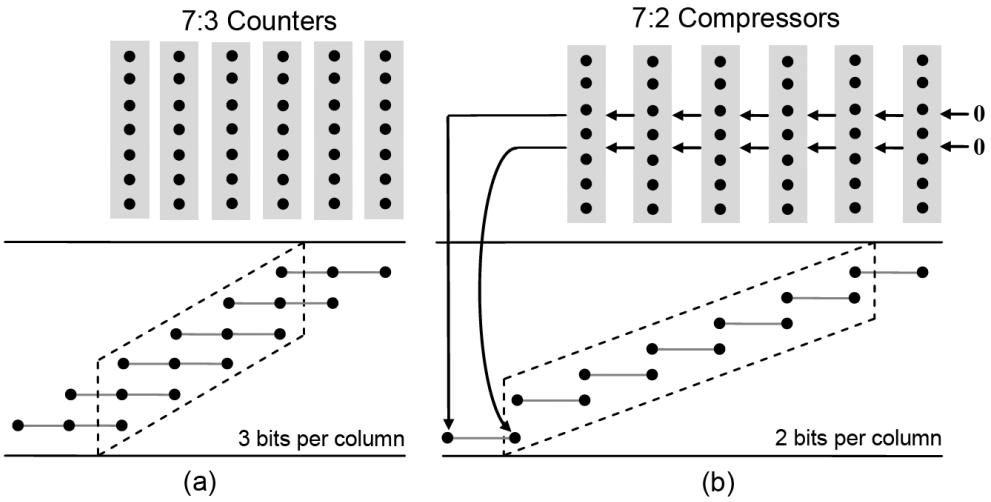


Figure 4.2: Compression ratio difference between counters and compressors. (a) Covering a set of columns with 7 : 3 counters yields three bits per column in the output; (b) using 7 : 2 compressors reduces the number of bits per column to two. Contiguous columns covered with 7 : 3 counters can be converted to 7 : 2 compressors.

carry chain that is used to implement ternary addition, using the four 3-LUTs configured as a carry-save adder. The novel features of the new logic block are the carry chains in the center—gray background—which can implement a 7 : 2 compressor, and the two multiplexers shown in gray on the right-hand side of Figure 4.3a, which selects between the outputs of the two carry chains. Similar to ternary addition, the new carry chains require the four 3-LUTs to be configured as a carry-save adder. To implement a 7 : 2 compressor, three additional full-adders (and a seventh LUT input) are required.

Three carry-in/carry-out bits are also required; they are labeled *X*, *Y*, and *Z* in Figure 4.3a. The carry-out labeled *X/Y/Z* connects to the corresponding carry-in labeled *X/Y/Z* of the next compressor in the chain. A detailed picture of the carry chains across several logic blocks is shown in Figure 4.3b.

In principle, the full-adders used in the two carry chains could be shared; this design choice was illustrated by us in [68]; although doing this could slightly reduce area, it requires that multiplexers be inserted into the carry chains, significantly increasing the critical path delay; as our goal is to increase performance, this design point is not ideal, especially since the area of the multiplexers offsets the area savings from sharing full-adders.

There are two primary advantages of providing an FPGA logic block that can be configured as a compressor compared to synthesizing GPCs on LUTs. The first advantage, which was illustrated in Figure 4.3, is that a $k:2$ compressor will have a higher compression ratio than a k-input GPC.

In some, but certainly not all cases, this can reduce the number of levels of logic in the

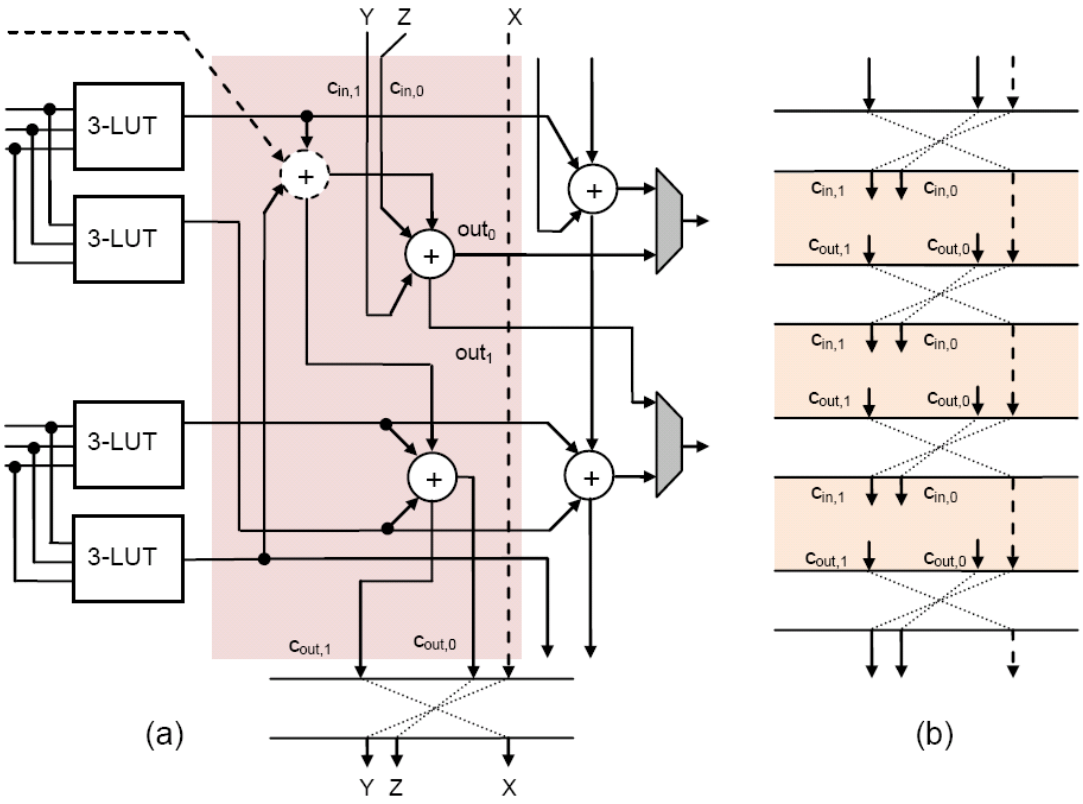


Figure 4.3: Logic block architecture with new hard-logic.(a) Enhanced version of the Shared Arithmetic Mode of the Altera ALM; new carry chains, shown in gray, allow the ALM to be configured as a 7:2 compressor. Two additional multiplexers are required to select between the two *sum* outputs of the 7 : 2 compressor and ternary adder—already present in the ALM; (b) pattern of carry-propagation for the 7 : 2 compressor.

compressor tree. The second advantage involves area utilization. Referring to the Table 3.1 in Chapter 3, to implement a $(0,7;3)$ GPC, 2.5 ALMs are required, while only one of our proposed logic blocks, which is marginally larger than an ALM, is required to realize a 7 : 2 compressor. Reducing the number of logic blocks, moreover, may allow for a tighter placement of logic blocks on the device, which, in turn, reduces wire-length and routing delay; our experiments confirm this hypothesis.

Consider the i^{th} compressor in the chain. Carry-in bits $c_{in,0}$ and $c_{in,1}$ are driven by the rank 1 carry-out of the $(i-1)^{st}$ compressor and the rank 2 carry-out of the $(i-2)^{nd}$ compressor, respectively; likewise, the rank 1 and carry-out of the i^{th} compressor drives carry-in, $c_{in,0}$, of the $(i+1)^{st}$ compressor, and the rank 2 carry-out drives carry-in, $c_{in,1}$, of the $(i+2)^{nd}$.

When an ALM is configured as a 2-bit ternary adder in shared arithmetic mode, six input bits are used; the 7 : 2 compressor, in contrast, requires an extra input bit. This is not a problem, as the ALM contains eight architecturally visible inputs; either of the two remaining inputs can be used as the seventh input when the ALM is configured as a 7 : 2 compressor.

4.4 Compressor Tree Synthesis on the New Logic Block

This section describes a mapping heuristic that can synthesize compressor trees targeting the logic block shown in Figure 4.3a. This heuristic is an extension of the mapping algorithm that was presented in Chapter 3, which targeted the Altera Stratix-III FPGA.

Compressor trees synthesized using an ASIC design flow produce two outputs that are summed using a CPA. Since ternary CPAs are available in Stratix-III for the same delay and area as binary CPAs, the heuristic outputs compressor trees that produce three outputs instead of two. The remainder of the compressor tree is synthesized using GPCs of Table 3.1 in Chapter 3. This section extends the mapping heuristic to include the possibility of configuring the logic blocks as 7:2 compressor as well. In principal, the mapping heuristic has three major steps:

1. Covering the bits in current level of compressor using GPCs
2. Replacing a subset of GPCs by $7:2$ compressors.
3. Exploiting the final CPA for computing the result.

Note that, the first two steps are repeated until a certain number of bit rows remain, and then the third step is performed.

The mapping heuristic generates one level of the compressor tree at a time. A subset of the input bits is covered by GPCs and possibly $7:2$ compressors. The output bits produced by each GPC are propagated to the next level of the compressor tree, along with the bits from the current level that are not covered. Since the rank of each GPC output bit is known, a new set of columns—array of integers—is generated for each level of the tree. A new level in the tree is generated until there are at most three rows of bits remaining, i.e., each column of the next level has at most three input bits. A ternary CPA completes the tree.

Once the GPC mapping of one level in the compressor tree is accomplished, the heuristic attempts to replace some GPCs with $7:2$ compressors. For this purpose, each contiguous sequence of $(0,7;3)$ GPCs is replaced with a contiguous sequence of $7:2$ compressors—or smaller single column GPCs, if $(0,7;3)$ GPC does not exist—similar in principle to Figure 4.2. Note that this transformation reduces the number of bits in the following level; aggregated over several levels, the use of compressors rather than counters can reduce the total number of logic levels in the compressor tree. Figure 4.4 shows an example, where a set of bits are first mapped by the GPCs and then the contiguous single column ones are chained, and thus replaced by the $7:2$ compressors. To chain the GPCs, priority is given to the eligible ones that have a higher compression ratio, i.e., $(0,7;3)$ GPCs. The chaining continues until no single column GPC remains for the next bit position in the chain. Several chains of $7:2$ compressors can be formed at each level of the compressor tree.

Next, the current level of the compressor tree is mapped onto logic blocks. GPCs are mapped onto ALMs, while 7:2 compressors require the logic block to be configured to use the carry

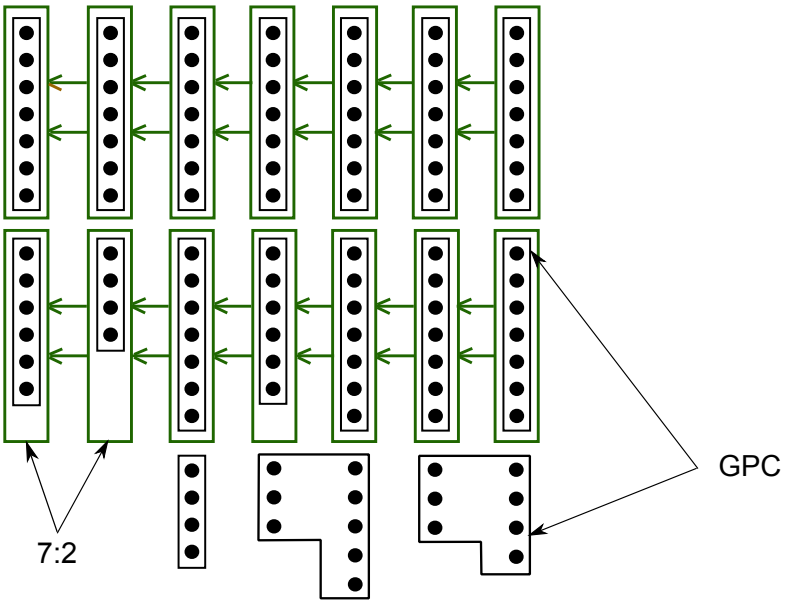


Figure 4.4: Mapping to the logic block of Figure 4.3a. The *first* step is to cover the bits with the GPCs, using the mapping heuristic of Chapter 3. The *second* step is to replace contiguous single column GPCs with 7 : 2 compressors.

chains shown in Figure 4.3a. Additionally, the outputs of the GPCs and compressors from the preceding level of the compressor tree are connected to the inputs of the GPCs and compressors in the current level. The last step is to generate the columns for the next level of the compressor tree.

The final step is to compute the result using the CPA. The final CPA uses the carry chains that are present on modern high-performance FPGAs. In the case of the Altera Stratix II-V series FPGAs, shared arithmetic mode permits the ALMs in the carry chains to be configured as ternary (3-input) CPAs with no additional cost over 2-input CPAs. To exploit this device family-specific feature, the compressor tree produces three outputs, rather than two. The CPA itself is comprised of a carry-save adder (implemented in LUTs) followed by a ripple-carry adder (implemented using the carry chains).

4.5 Experimental Setup

To evaluate the new FPGA logic block, we use the academic placement and routing tool, VPR [12, 13], which is widely used in FPGA research areas. Although it is a very useful tool, it still does not support carry chains. However, we tried to pack logic blocks into logic clusters to be able to use VPR to model the carry chains. The drawback of this approach is that a limited number of logic blocks could be placed in a cluster, as the resulting cluster will demand huge bandwidth. In the following, we first describe the logic block modeling by VPR, and then we explain the packing step.

4.5.1 VPR

The publicly available Versatile Place-and-Route (VPR) tool [12, 13] was used to evaluate the new FPGA logic blocks proposed in Section 4.3. The algorithm in Section 4.4 was used to map each compressor tree onto the new FPGA. This determines the number of logic blocks required to realize the circuit. VPR was then used to place and route the circuit; afterwards, VPR reported the critical path delay, including its decomposition into logic and routing delays, wire-length, and the minimum number of routing tracks per channel for which the design is routable.

VPR models an island-style FPGA, where each island is a cluster, containing one (or more) *Basic Logic Elements (BLEs)*. Each BLE consists of a programmable LUT, a flip-flop connected to the LUT output, and multiplexer. The selection bit of the multiplexer is programmed, such that it can select the LUT output for combinational logic or the flip-flop output for sequential logic. BLEs within the same cluster connect to each other by a fast local routing network. The global routing network, which is slower, connects BLEs in different clusters. The cluster in an Altera Stratix-series FPGA is called a *Logic Array Block (LAB)*, and contains several ALMs.

We used VPR version 4.30 to model logic blocks and logic clusters that resemble Altera's ALMs and LABs. Each LAB in our architecture contains four ALMs. Since VPR does not model carry chains between LABs, we model each carry chain output as being provided by an additional LUT inside the LAB, whose delay is specified appropriately. Another difference between VPR 4.30 and realistic FPGAs involves the routing network: Stratix II-V organizes LABs into columns with nonuniform routing in the x- and y- directions; the baseline VPR architecture, in contrast, has uniform routing.

We modeled a clone of the Altera ALM in VHDL, and added the extra carry chains and two multiplexers shown in Figure 4.3a, along with one additional configuration bit, which is only set when the ALM is configured as a compressor; two different versions of the modified ALM were created, that, respectively, support configurations as $7:2$ compressor. Using Shared Arithmetic Mode, the ALM can be configured as a 2-bit ternary adder; each LAB contains four ALMs, and can be configured as an eight-bit ternary ripple-carry adder. The global routing network can be used to build larger ripple-carry adders.

The ALM clones were synthesized with Synopsys Design Compiler using a $90nm$ Artisan standard cell library based on a TSMC design kit. The delays of the paths through the ALM clone were input into the VPR architecture configuration file to model the logic and carry chains delays. We estimated the size of each cell in terms of 2-input gates; 22 additional gates were required to implement the carry chains for the 7:2 compressor, including the two multiplexors in Figure 4.3a, and the extra configuration bit; this increased the area of the ALM–excluding the routing resources–by less than 5%. Figure 4.5 shows the delays of the output bits of the ALMs in a LAB; when configured as a 6-LUT, the delay of each output is always $0.69ns$; for other configurations, the delay depends on the position along the carry chains. VPR generates an FPGA whose dimensions are sized specifically for each benchmark circuit. This

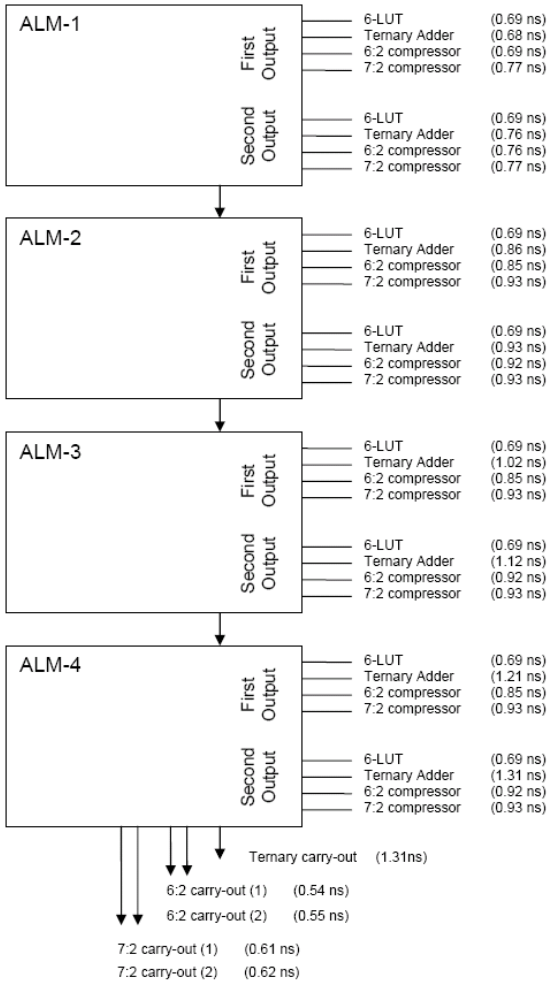


Figure 4.5: Combinational delays of the ALM outputs in a LAB, including propagation delays along the carry chains.

tends to minimize the routing delays from the FPGA's input pads to the circuit inputs, and from the circuit outputs to the FPGA's output pads. The FPGA generated by VPR must have at least as many LABs as the packed circuit, and must satisfy an aspect ratio specified by the user. For example, if the user specifies an aspect ratio of 1, and the circuit requires 23 LABs, then the FPGA generated by VPR will be a 5×5 array. An aspect ratio of 1 was used throughout our experiments.

VPR uses a binary search to determine the minimum number of tracks for which a legal route can be found for each circuit. For a given placement, let t_x and t_y be the number of routing tracks used in the x and y directions, and let $t_z = \max t_x, t_y$. VPR stops the binary search when it finds the minimum value of t_z for which a legal route is found.

To model routing delays, the per-unit resistance and per-unit capacitance of the wires must be specified in the VPR architecture configuration file. We selected per-unit resistance and per-

unit capacitance values based on the TSMC 90*nm* CMOS technology, under the assumption that metal-6 is used for wires.

4.5.2 Packing

Technology mapping for FPGAs maps a circuit implemented in terms of basic gates—e.g., AND, OR, XOR, etc—onto appropriate FPGA components: LUTs, carry chains, DSP blocks, etc. The compressor tree synthesis heuristic described in Section 4.4 is a form of technology mapping that is specific to compressor trees mapped onto ALMs that have been modified as shown in Figure 4.3a.

Packing is the process of assigning each ALM in a technology mapped netlist to exactly one LAB. The number of ALMs assigned to the same LAB cannot exceed the maximum number of ALMs per LAB, which is an architecture-specific parameter.

Technology mapped ALMs that are connected by a carry chain must be mapped to the same LAB; otherwise, the carry chains cannot be used.

VPR 4.30 includes a packing tool called *T-VPack*; as VPR 4.30 does not support BLEs with carry chains, T-VPack cannot enforce the constraint described above, because it is unaware of the presence of carry chains.

Instead of using T-VPack, we wrote our own packing software that is specific to the compressor trees produced by our mapping heuristic. The packer is greedy: it always selects the longest carry chains, and packs up to four ALMs along the chain into the same cluster. If the length of the chain is $k > 4$, then the first four ALMs in the chain are packed together, and the chain is broken after them; this yields a new chain of length $k-4$, which is reinserted into the set of carry chains. When no carry chains remain, the remaining ALMs are packed arbitrarily. After packing, VPR performs placement and routing.

4.5.3 Benchmarks

To evaluate the new FPGA logic block, we used the same benchmarks that was used in Chapter 3—refer to Section 3.5.2 for the description of the benchmarks that are used for the experiments of this chapter.

4.6 Experimental Results

4.6.1 Overview of Experimental Comparison

Throughout our experiments, each compressor tree was synthesized three times. Table 4.1 summarizes the three different approaches: *Ternary*, *GPC*, and *7:2+GPC*. For the GPC mapping, we used *MaxPD* method of Chapter 3, in which the primary objective is to minimize the

Synthesis Method	Description
Ternary	Ternary Adder Tree
GPC	Standard GPC mapping (*MaxPD* in Chapter 3)
7:2+GPC	Compressor tree mapping using 7:2 mode (Section 4.4)

Table 4.1: Description of the three synthesis methodologies used in the experiments.

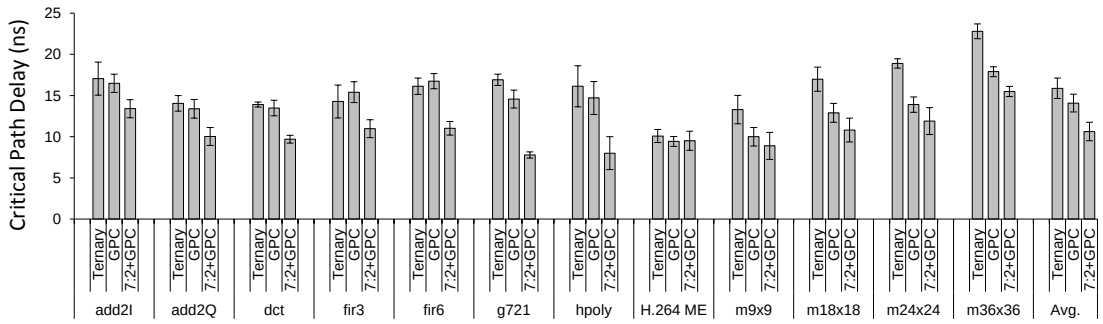


Figure 4.6: The critical path delay for each benchmark and compressor tree synthesis methodology, shown with a 95% confidence interval.

delay. As described in Section 4.4, each compressor tree produces three outputs that are summed with a ternary ripple carry adder. The Ternary and GPC synthesis methods target high-performance FPGAs that contain 6-LUTs, carry chains, and support for ternary addition; this includes Altera Stratix II-V, as well as Xilinx Virtex 5-6. 7:2+GPC target FPGAs containing the modified ALM in Figure 4.3a.

The heuristic used to build ternary adder trees is similar in principle to the GPC mapping strategy introduced in Section 4.4; the primary difference is that the component library contains just one component: a ternary adder, i.e., a $3:1$ CPA. When there are several choices of input bits to add at a level of the tree, then priority is given to the widest possible CPA with the longest carry chain.

In Chapter 3, we have already compared Ternary and GPC on the Altera Stratix-III FPGAs. GPC yielded compressor trees with faster critical path delay; however, these compressor trees required more ALMs. Similar trends are observed here. The experiments presented here evaluate the benefit of extending these logic blocks to be configurable as a $7:2$ compressor. As discussed in the preceding section, we estimate that this new logic block is at most 5% larger than the ALM–excluding the routing resources–in the Stratix II-V FPGAs. The benefits obtained using the new logic block, as reported here, must be weighed against a uniform increase in the area of all logic blocks in the FPGA, including a great many that will not be configured as a $7:2$ compressor for any specific circuit.

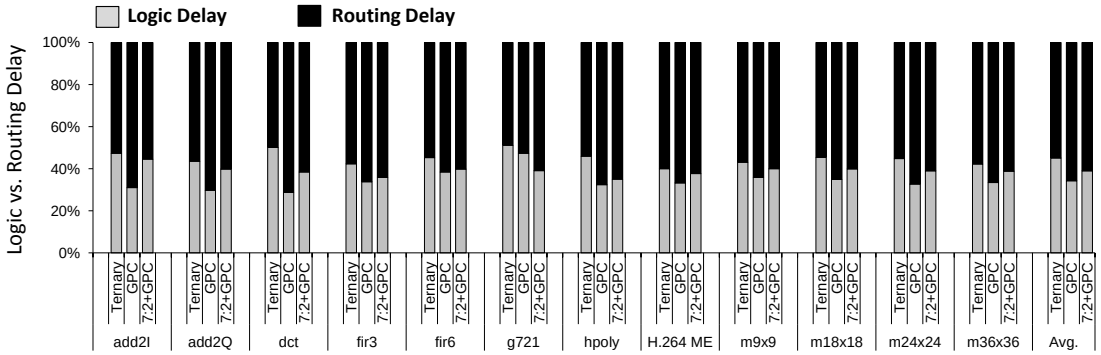


Figure 4.7: On average, the percentage of critical path delay due to logic and routing for each benchmark.

4.6.2 Critical Path Delay

First, we measure the critical path delay for each benchmark and decompose it into logic delays within the compressor tree and final CPA, and routing delays. We synthesized each benchmark ten times using VPR, using a different random number seed each time. Figure 4.6 shows the average critical path delay of the ten runs, including a 95% confidence interval for each benchmark and synthesis method.

In contrast to what we observed in Chapter 3, for some benchmarks such as *fir3* and *fir6*, Ternary is faster than GPC. While in Chapter 3, we observed that GPC is always faster than Ternary. These incompatibility between the results is the due to the fact that we use different FPGA tools in these two chapters. In this chapter, we have to use VPR, which is an academic tool and has many constraints, for modeling the FPGA and performing place and route, while in Chapter 3, we used matured FPGA commercial tools.

The delay results in Figure 4.6 reveals that 7:2+GPC offers a definitive advantage over Ternary and GPC. This is, indeed, what we expected, as $7:2$ compressors have higher compression ratio—see Figure 4.2—and have less pressure on the routing network—carry chains replace the routing wires. Among the benchmarks, *H.264 ME* is the smallest one with the shallowest compressor trees; hence, there is little differentiation between the results of the compressor tree synthesis methods for these three benchmarks.

Figure 4.7 decomposes the average critical path delay into percentages due to logic (including carry chains) and routing. Typically, logic delays consumed 30-45% of the overall delay, for each benchmark and synthesis method. For each benchmark, Ternary had the highest percentage of logic delay, which can be attributed to the use of carry chains at each level of the tree; taken in aggregation, the carry chains within the adder tree start from the least significant input bit to the most significant output bit of the final CPA, although the critical path is not guaranteed to include the final CPA.

In contrast, GPC and 7:2+GPC, include logic delays through some portion of the compressor

tree, followed by some, but not all, of the final CPA. On average, the critical path of GPC subsumed a slightly greater percentage of routing delay than 7:2+GPC, however, this trend did not occur uniformly across all benchmarks.

Among the ten runs for each benchmark and synthesis method, the standard deviation of the logic delays was always non-zero. Along any specific path through the circuit, the logic delay will always be fixed, but the routing delay differs, depending on the placement. The non-zero standard deviations of the logic delays indicate that random changes in the placement, i.e., variation in routing delays due to different random number seeds used by VPR, can change which paths become critical.

In ASIC technologies, the arrival time of each compressor tree output bit at the CPA can be predicted from synthesis and used to optimize the CPA design [63]; the non-zero standard deviations for logic delays in our results indicate that for FPGA design flows, variations in routing delays make this process inexact. Hence, methods to simultaneously optimize the compressor tree output delay profile and the final CPA design for FPGAs, i.e., those that are analogous to Oklobdzija and Villeger's, are unpredictable prior to placement and routing due to variations in routing delay. Moreover, due to the specific features of FPGA logic architectures, including carry chains, the hybrid final CPAs proposed by Oklobdzija and Villeger [63] may be an inappropriate choice for FPGAs. For this reason, we chose to implement the final CPAs using ripple-carry chains. Alternative CPAs, such as carry-select adders, will require more ALMs. Although they are superior in terms of logic delay, they may be inferior when the additional costs of routing are taken into account. A detailed investigation into final CPA design for compressor trees in FPGA technologies is left open for future work.

4.6.3 Critical Path Analysis

For GPC and 7:2+GPC, the logic delay includes a few layers of logic blocks (LB layers) in the compressor tree, followed by some number of bits in the CPA. As the specific critical path varies from run to run, we focus on individual runs. In particular, we select the minimum critical path among the ten runs for each benchmark and synthesis method for an analysis of the components that contribute to logic delay. This analysis does not make sense for Ternary, because the critical paths in the adder tree may include long carry chain delays at each level of the tree, not just the final CPA.

Figure 4.8 decomposes the critical path delay into logic delays within the compressor tree and CPA, and routing delays, for GPC and 7:2+GPC. Generally, 7:2+GPC has shorter logic delay—due to higher compression ratio, thus having fewer logic levels—and routing delay—due to reducing the stress on the routing network through the extensive use of new carry chains as well as utilizing less logic blocks.

$7:2$ compressors can lead to different phenomena that impacts critical path delay; For *add2I*, for example, the use of $7:2$ compressors reduces the LB layers on the critical path in the

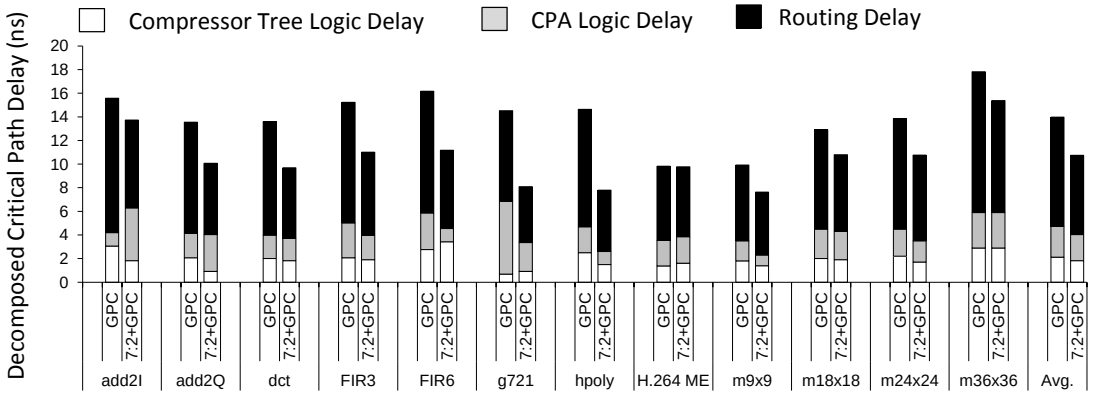


Figure 4.8: The minimum critical path for each benchmark and synthesis method, decomposed into logic delays within the compressor and CPA, and routing delay.

compressor tree, but increases the number of bits on the critical path in the CPA.

For *add2Q*, 7:2+GPC reduces the number of LB layers in the compressor tree from three to one, but incurs a greater delay through the CPA than the GPC. The logic delay of 7:2+GPC, is comparable to that of GPC.

For *dct*, *fir3*, *hpoly*, *m9x9*, *m18x18*, and *m24x24* the critical path of 7:2+GPC, in contrast to GPC, passes through fewer LABs in the compressor tree and fewer final CPA bits.

For *fir6* and *hpoly*, the critical path of 7:2+GPC includes fewer bits in the final CPA, and significant reductions in routing delay, compared to GPC.

Among all benchmarks, *g721* has the largest final CPA bitwidth, 39, and would thus be the most likely to benefit from techniques that can synthesize faster CPAs than ripple-carry adders. Its critical path includes one LB layer in the compressor tree, but 37 bits of the CPA. 7:2+GPC achieves a far superior reduction in logic delay, as its critical path goes through just one LAB in the compressor tree, and fourteen bits in the final CPA, and also benefits from the smallest routing delay as well.

H.264ME is the smallest benchmark evaluated here. As shown in Figure 4.6, Ternary actually achieves the best critical path delay, while Figure 4.7 shows that this is because routing delays account for a smaller fraction of the overall critical path delay for Ternary than GPC and 7:2+GPC. The critical path of 7:2+GPC goes through more bits in the final CPA than GPC, and this tends to dominate the logic delay.

For *m36x36*, 7:2+GPC achieves a smaller critical path delay compared to GPC due to reduced routing delay.

On average, the critical path of GPC goes through 2.7 LB layers in the compressor tree and 13.2 bits in the CPA; the critical path of 7:2+GPC goes through 1.9 LABs in the compressor tree and

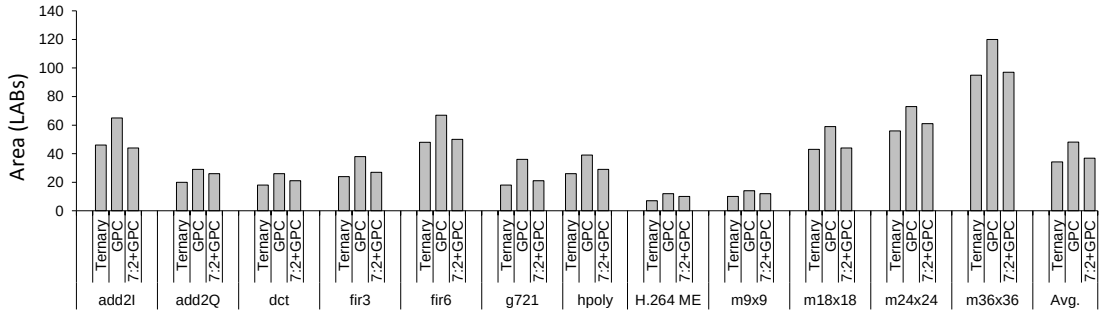


Figure 4.9: The area (LABs) required for each benchmark and compressor tree synthesis method.

11.5 bits in the final CPA. On average, the routing delay of GPC is 6.3ns and the routing delay of 7:2+GPC is 4.2ns. Altogether, the reductions in routing delay tend to have a greater impact on critical path delay that differences in logic delay.

To summarize, the benefits of the 7 : 2 compressors, in terms of logic delay, vary from benchmark to benchmark; there is no uniform or universal answer. The logic delay of the compressor tree may increase or decrease compared to other methods; the same is also true for the delay through the CPA. At no point, however, do both the compressor tree and CPA logic delays increase for 7:2+GPC over GPC. 7:2+GPC also retains advantages in terms of routing delay compared to GPC.

4.6.4 Area Utilization

Figure 4.9 shows the area—number of LABs—required for each benchmark. In general, Ternary achieve the smallest area, because ALMs in shared arithmetic mode have a compression ratio of 3, whereas, 6-input, 3-output GPCs have a compression ratio of 2, while requiring two ALMs. Although 7 : 2 compressors have compression ratios of 3.5, the use of GPCs in addition to the compressors causes 7:2+GPC to use more ALMs than Ternary. GPC, consequently, requires the most area uniformly across the benchmark suite.

Figure 4.8 showed that GPC tends to have larger routing delays than 7:2+GPC. Now, Figure 4.9 shows that GPC tends to require more LABs than 7:2+GPC as well. This suggests that one reason for the higher GPC's routing delay is the higher area utilization. Each GPC requires several ALMs, while each 7 : 2 compressor requires just one. Consequently, the use of compressors instead of GPCs tends to reduce the number of ALMs used in a design. This, in turn, leads to a tighter placement, which tends to reduce wire-length. As each wire crosses through fewer switch and connection boxes, routing delays tend to reduce as well.

Lastly, each wire that connects to a GPC has a higher fanout than a wire connecting to a 7 : 2 compressor, as multiple ALMs are required to implement the GPC. This can be another reason for the wire-length reduction shown in the next section.

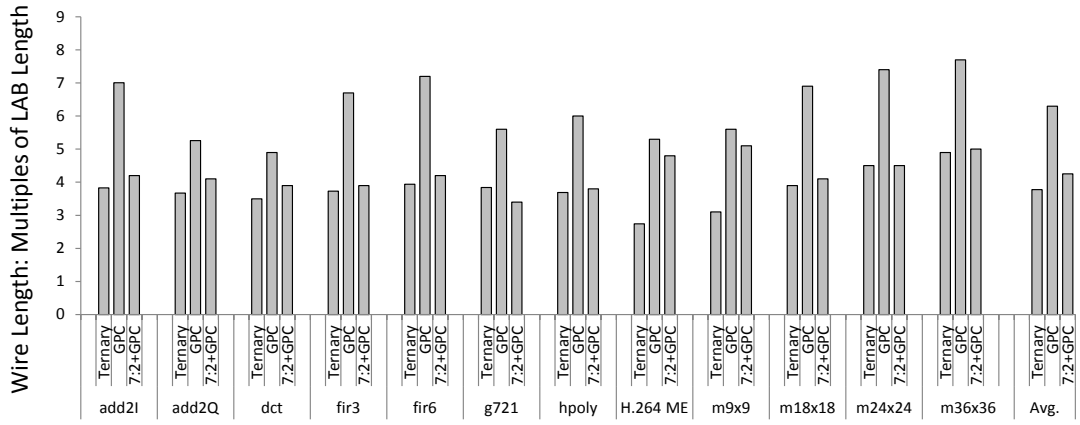


Figure 4.10: Average wirelength per net for each benchmark and compressor tree synthesis method.

4.6.5 Wire-length and Routability

This section compares and contrasts Ternary, GPC, and 7:2+GPC in terms of wire-length and routability. Our VPR architecture configuration contains a mixture of segments of different lengths. In the architecture we studied, 90% of wires have span two LABs, while the remaining 10% span four LABs; buffered routing switches were always used.

Figure 4.10 reports the average wire-length per net for each benchmark and synthesis method. The wire-length reported in Figure 4.10 accounts for the varying lengths of the different segments.

For each benchmark, the average net wire-length of GPC was larger than that of Ternary and 7:2+GPC. Figure 4.9 has shown that GPC requires more LABs than 7:2+GPC; Figure 4.10 shows that the tighter packing achieved by 7:2+GPC is able to reduce the average net wire-length, which in turn, reduces the overall critical path delay.

Admittedly, Figure 4.10 does not compare the wire-lengths on the critical path; however, routing delay can affect which paths are critical, as discussed in subsections 4.6.2 and 4.6.3. Thus, it is reasonable to assume that the critical path will have longer wires for GPC compared to 7:2+GPC for each benchmark, as the average net wire-length of GPC tends to be longer as well.

Recall from Section 4.5.1 that VPR repeatedly places and routes the circuit using a binary search, stopping when it finds the minimum channel width for which it can achieve a legal route. Figure 4.11 reports the minimum channel width in the x and y directions found by VPR for each benchmark. GPC tends to achieve routability with narrower channels than Ternary and 7:2+GPC. As GPC requires more LABs than the other synthesis methods, the overall circuit is spread across a greater portion of the FPGA area. This tends to reduce congestion in the routing network, and hence, competition for routing tracks in the most congested area [27].

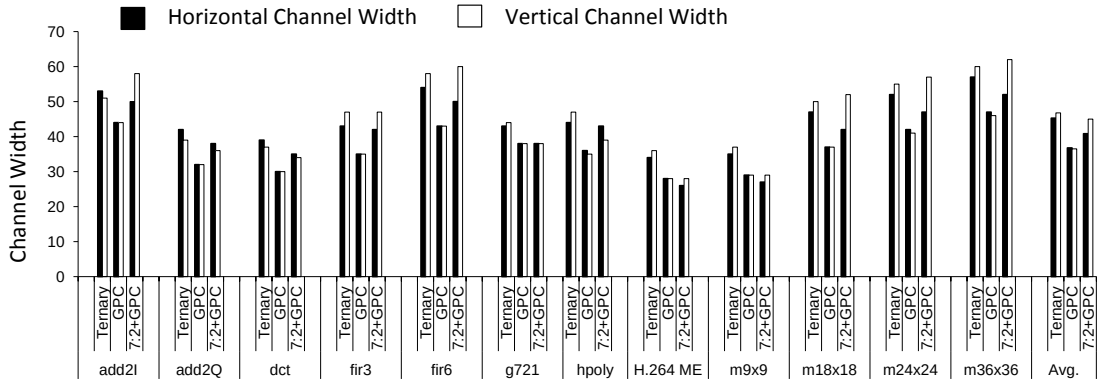


Figure 4.11: The minimum channel width in the x and y directions for which each benchmark is routable for each compressor tree synthesis method.

Jamieson and Rose [43] have suggested that 180 routing tracks per channel is typical for modern FPGAs. Figure 4.11 shows that all of our benchmarks require no more than 60 routing tracks per channel, indicating that routability of the benchmarks studied here would not be a concern, with or without the modified logic block presented in this chapter.

4.7 Related Work

This section describes a number of proposals to improve the arithmetic and logical capabilities of FPGA logic blocks. The most enduring idea has been the integration of carry chains into FPGA logic blocks along with LUTs. Carry chains include fast connections between adjacent logic blocks that are used for carry propagation; this permits the elimination of most of the routing delays that would otherwise be present.

Hauck et al. [35] proposed the complicated carry chains that can implement *Brent-Kung*, *carry-select*, and *carry-lookahead* addition. Different logical constructs were needed for different cells in the chain, making them nonuniform. This creates integration challenges because it is difficult to layout a regular fabric consisting of irregular cells. This would require a large manual effort to design each individual cell at the transistor level, and would complicate the layout process for the entire chip.

Frederick and Somani [31] proposed a uniform logic block with carry chains that could efficiently implement a *carry-skip* adder; a similar bi-directional carry-skip chain was earlier proposed by Cherepacha and Lewis [17]. Kaviani et al. [47] and Leijten-Nowak and van Meerbergen [54] developed ALU-like blocks that support arithmetic functions such as addition, subtraction and (partial) multiplication.

Distributed Arithmetic (DA) [60] is a paradigm for implementing effective hardware for DSP systems that uses LUTs instead of multipliers. Grover et al. [34] developed a special DA-oriented LUT structure (*DALUT*) specifically for *Multiply-Accumulate (MAC)* operations. In

addition to two 4-input LUTs, their DALUT cell included arrays of *XOR* gates, bit-level adders and shift accumulators, shift registers, and a CPA to add partial summations and carries.

The first generations of ALtera Stratix [5] and Cyclone [3] FPGAs had carry-select adders and carry chains, while these carry chains were replaced by ripple carry chains in the subsequent FPGA generations.

A K-input *macro gate* [23] is similar to an LUT, but it cannot implement all 2^K logic functions, and therefore has reduced delay and area. Hu et al. [39] suggested that FPGA cells could benefit from the inclusion of both LUTs and macro gates. Similar to Kastner et al. [45], they developed an automated method to profile a set of applications to find good macro-gate candidates. They did not, however, consider arithmetic-dominated functions or fast carry chains between macro gates.

4.8 Conclusion

In this chapter, we introduced a new FPGA logic block that has new carry chains and can be configured as a 7 : 2 compressor. We showed that due to its higher compression capability and area efficiency, the 7 : 2 compressor is a better building block for implementing compressor trees on FPGAs, compared to GPCs that were introduced in the previous chapter. Compressor tree synthesis using GPC mapping—see Chapter 3—reduces the critical path delay compared to synthesis using ternary adder trees; however, GPC mapping requires more ALMs. The logic block proposed in this chapter significantly improves the situation. Used in conjunction with the GPC mapping, the new logic block offers a moderate average reduction in critical path delay and total wire-length compared to GPC mapping using standard ALMs, while using significantly less logic blocks.

With this new carry chains, which are realized by (partially) reusing the current resources—few extra gates are only added—the functionality of this type of the hard-logic is expanded, and this way a larger subset of applications could benefit from these dedicated and efficient resources. This conforms to the first defined path of the thesis roadmap, in which the goal is to increase the generality of the hard-logic of FPGAs.

As a conclusion, the implementation of the carry-save arithmetic circuits on FPGAs can be improved (considerably) using the new carry chains that were introduced in this chapter. These new carry chains add little area and delay overheads to the current logic blocks. These overheads, indeed, can be justified considering the importance of multi-input addition in applications and the great benefits of new carry chains for implementing multi-input additions. However, the FPGA vendors may argue that even these little overheads can be critical for the applications that do not use these carry chains, and thus they might be unwilling to change the structure of the logic blocks. In this case, the mapping approach that was presented in the previous chapter will be the best (soft-logic) option to implement compressor trees.

5 Versatile DSP Blocks

As explained in Chapter 2, there are two types of hard-logic in FPGAs: (1) the one that is coupled with the soft-logic, and (2) the one that is stand-alone in island-style FPGAs. The latter type is external to the soft-logic clusters and has a coarse grained structure. This type of hard-logic includes DSP blocks, memory blocks, and hard processors. DSP blocks are typically intended to accelerate critical arithmetic operations such as parallel multipliers. These dedicated resources are highly efficient when they are used (properly); however, due to their inflexible structure, DSP blocks and the expensive routing resources around them can be wasted, when they are not used. Hence, based on the first direction of the thesis *roadmap*, it is essential to increase the generality and flexibility of DSP blocks, such that, more applications take advantage of these useful resources.

Current FPGAs DSP blocks are inherently ASIC-like integer multipliers, which also support few other arithmetic operations. The multiplier bit-widths that are supported by the DSP blocks are generally very limited, and this can result in having poor implementation of other multiplication bit-widths [50]. Moreover, due to their rigid structure, it is not feasible to use the DSP blocks resources for different purposes, e.g., using the Partial ProductRreduction Tree (PPRT) of the multipliers for performing multi-input addition. In ASIC design, for the PPRT of multipliers, compressor trees built from carry-save adders are used, e.g., Wallace [88] and Dadda [26] trees. In the DSP blocks, however, these compressor trees are not directly accessible to the programmer.

In this chapter, we present a versatile DSP block for FPGAs that is more flexible and provides extra features allowing reuse of the available resources. This new DSP block, indeed, is constructed on top of a base skeleton that supports the basic features of current DSP blocks. Meanwhile, this base structure is designed such that additional flexibility and features are added with minimum extra costs. To design this FPGA DSP block, we used our prior experience concerning the challenges that we faced in designing different DSP blocks for implementing carry-save arithmetic. The DSP block that will be presented in this chapter, easily supports various multiplication bit-widths as well as carry-save arithmetic, reusing the multiplier resources.

5.1 Introduction

FPGAs performance is lacking for arithmetic circuits. Generally, arithmetic circuits do not map well onto LookUp Tables (LUTs), the primary building block of the soft-logic of FPGAs. To address this concern, FPGAs offer two solutions: firstly, LUTs are now tightly integrated with fast carry chains that perform efficient carry-propagate addition; secondly FPGAs contain DSP blocks that perform multiplication and multiply-accumulation. Although an improvement over LUTs alone, these enhancements lack generality; specifically, they only support limited multiplication bit-widths and they cannot effectively accelerate carry-save arithmetic that is required for compressor trees.

As discussed earlier in this thesis, compressor trees naturally occur in many signal processing and multimedia applications, such as FIR filters, and, in the more general case, their use can be maximized through the application of high-level transformations to arithmetically intensive data flow graphs [86]. Such transformations rearrange the operations in the data flow graphs to favor the use of carry-save arithmetic. Although the PPRTs of the multipliers in DSP blocks are designed using compressor trees, they are not accessible and usable for the implementation of the compressor trees of applications. Hence, the mentioned applications cannot take advantage of DSP blocks for this purpose.

In addition, the DSP blocks of current have a fixed bit-width multiplier as the base; some architectures do not support any other bit-width [92], and some only support limited bit-widths that are formed on top of the base multiplier [7]. Therefore, when a different bit-width is required, normally little or no gain is obtained from DSP blocks, and in some cases, it is even better to use the soft-logic for the implementation [50].

Prior to this work, we designed DSP blocks that exclusively perform multi-input addition using carry-save arithmetic [71, 14]. Although these DSP blocks can implement the PPRT unit of the multipliers, to implement the Partial Product Generator (PPG) of multipliers, the soft-logic is involved. Consequently, these DSP blocks are not appropriate for implementing multipliers compared to existing DSP blocks. In another work [70], we added PPG logic to one of these DSP blocks [14] to improve multipliers implementation. Though, the problem with this DSP block is that due to its inherent structural limitation, fewer and smaller multipliers can be supported compared to the existing DSP blocks. However, this experience helped us to design a versatile DSP block that not only supports carry-save arithmetic, but also can efficiently implement various multiplication bit-widths.

In the subsequent sections, we first briefly review the architecture of the DSP blocks that we designed exclusively for carry-save arithmetic, and then we present a versatile DSP block that efficiently support various multiplication bit-widths, in addition to carry-save arithmetic for multi-input addition.

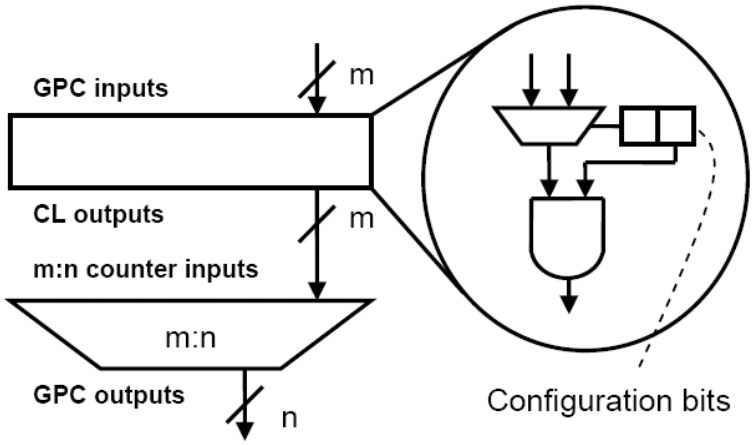


Figure 5.1: Architecture of an m-input, n-output programmable GPC. The programmable GPC is the building logic block of FPCA.

5.2 Overview of DSP Blocks for Multi-input Addition

As mentioned in the previous section, prior to this work, we designed two different architectures for accelerating multi-input addition. In this section, we briefly review these two designs. The first architecture includes specialized logic blocks that are grouped into logic clusters with local routing. This architecture is called *Field Programmable Counter Array (FPCA)*. The second architecture, in contrast, does not include any internal routing, and its logic blocks are interconnected through fixed carry chains. This architecture is called *Field Programmable Compressor Tree (FPCT)*. Both architectures exclusively support multi-input addition; hence, to implement a multiplier or any circuit that is a mixture of normal logic and multi-input addition, both the soft-logic—i.e., LUTs—and the hard-logic—these DSP Blocks—are involved.

5.2.1 FPCA Architecture Overview

FPCA is an array of programmable logic blocks that are specifically designed for implementing multi-input addition. The logic blocks are configurable *Generalized Parallel Counters (GPCs)*, to which a wide variety of GPCs that meet the IO constraints are mappable. As shown in Figure 5.1, each configurable GPC consists of an $m:n$ counter and a *configuration layer*. This configuration layer allows the user to select the desired GPC to implement. For example, a programmable GPC with $m = 15$ and $n = 4$ should be able to implement the functionality of both a $15:4$ counter and a $(5,5;4)$ GPC, among others.

Based on the GPC definition in Chapter 2, each GPC can add bits with different ranks. Therefore, to implement a GPC, the $m:n$ counter performs the addition part, where some input bits are added redundantly based on their rank. For example, a bit with rank one is added two times. Hence, the configuration layer consists of some multiplexers that specify the right inputs of the $m:n$ counter based on the GPC configuration. As many GPCs are supported,

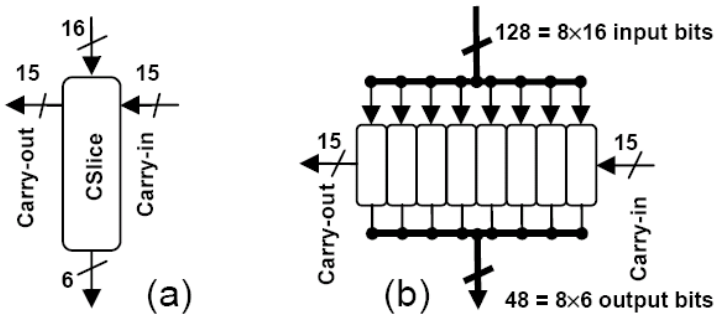


Figure 5.2: FPCT structure, consisted of 8 CSlices. (a) I/O interface to a CSlice (b) and an 8-CSlice FPCT.

the configuration layer would have a complicated structure with many multiplexer layers. Though, by exploring a symmetry property in the GPCs designs, we proposed a design for the configuration layer that is only comprised of one layer of 2 : 1 multiplexers. For details of this design, refer to [71].

In FPCA architecture, we have an array of configurable GPCs, which are connected through a local programmable routing network. In a sense, our intention to embed an FPCA into an FPGA is similar in principle to a cluster of the soft-logic blocks in conventional FPGAs—e.g., a Logic Array Block (LAB) in the Altera Stratix series FPGAs. A LAB (or group of adjacent LABs) could be replaced with an FPCA. The role of a programmable GPC within an FPCA is analogous to the role of an ALM within a LAB—refer to Chapter 2, Section 2.2.1 for the LAB and ALM structures.

5.2.2 FPCT Architecture Overview

FPCT is an alternative design to FPCA, in which the building blocks are called *Compressor Slices (CSlices)* and are connected by hard-wired carry chains rather than local routing network. Although FPCT is less flexible than FPCA, due to the fixed connection of its building blocks, it is more efficient than FPCA. FPCT is comprised of eight CSlices, as shown in Figure 5.2b. Each CSlice, as shown in Figure 5.2a, takes as input 16 bits to sum along with 15 carry-in bits, and produces up to six output bits, depending on its configuration, along with 15 carry-out bits. The 8-CSlice FPCT, shown in Figure 5.2b, takes 128 input bits (16 per CSlice) along with 15 carry-in bits for the lowest order (rightmost) CSlice; it produces up to 48 output bits (six per CSlice), and the highest order CSlice (leftmost) produces 15 carry-out bits.

Within a CSlice, compression is performed by a network of GPCs of varying size. The CSlice includes a bypassable CPA: it can produce one, two, or three output bits when using the CPA, or two, four, or six output bits in the carry-save form, bypassing the CPA; the latter is used when building large, multi-FPCT compressor trees. More details about the FPCT architecture

can be found in [14].

The 15-carry-out bits connect to subsequent CSlices using a limited carry chain; depending on the configuration of the CSlice, some of these bits may be 0; likewise, the rank of each carry-out bit depends on the configuration.

5.3 Proposed Versatile DSP Block

Both types of DSP blocks, current ones and the ones that we designed for carry-save arithmetic, suffer from the same problem, which is the lack of generality; applications that have multiplication as the base operation will not benefit from the migration to an FPGA containing FPCAs or FPCTs, and applications that have multi-input addition cannot benefit from current DSP blocks. This means that to satisfy both classes of applications FPGAs should contain a mixture of FPCAs/FPCTs and DSP blocks.

However, adding new hard-logic—e.g., FPCA or FPCT— to FPGAs can make the gap problem even worse, if they are not used. That being said, the primary focus of this chapter is to design a DSP block that can be used for both multiplication and carry-save arithmetic, rather than having separate DSP blocks for each. Moreover, the other goal is to increase the flexibility of DSP blocks in supporting various multiplication bit-widths.

In one attempt [70] as explained earlier, we modified the FPCT structure such that a bypassable PPG unit was added to avoid the soft-logic for the PPG implementation. The resulting DSP block can perform multi-input addition as well as multiplication. This new DSP block can be configured as only two 9×9 multipliers, due to the original constraints of the FPCT, despite the available bandwidth. While, the current DSP blocks in the Altera Stratix-series FPGAs, with the same bandwidth, can be configured up to eight 9×9 multipliers.

The experience of designing the DSP blocks for carry-save arithmetic and trying to adapt them for multiplication, motivated us to design a new DSP block starting from a base structure that has the basic features of current DSP blocks. In other words, the key idea of this chapter is to design a DSP block that maintains the original features of current ones, and on top of that, we will be able to add new features with minimum extra costs. For this purpose, we take the DSP block of the Altera Stratix-series FPGAs as the base reference structure, and we redesign it such that new functionalities can be supported with little overhead.

5.3.1 Architecture of the Base DSP Block

As shown in Figure 5.3, the fundamental building block of our base DSP block, similar to DSP blocks of the Altera Stratix-series FPGAs, consists of two paired 18×18 multipliers followed by an optional adder stage. The adder unit is either used for complex arithmetic multiplication or for constructing larger multipliers by combining the smaller ones [6].

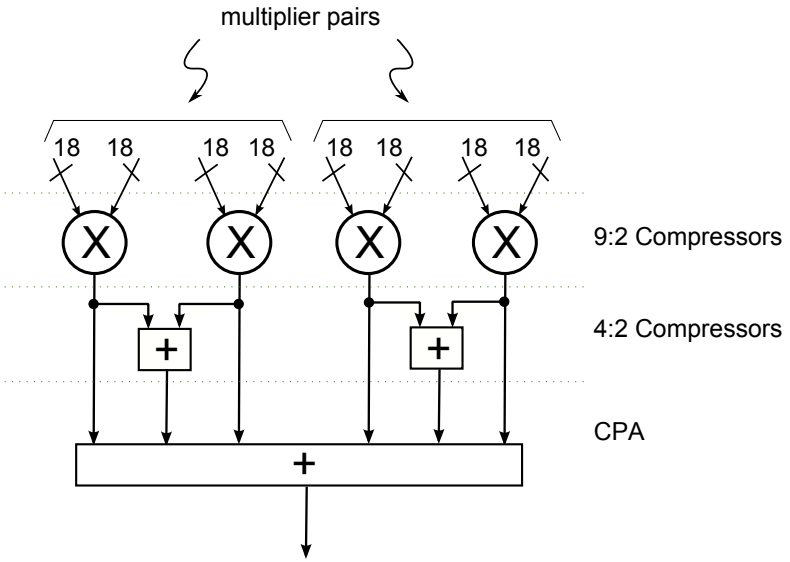


Figure 5.3: Conceptual illustration of the reference DSP block architecture. Four 18×18 multiplies along with the adders that are required for either constructing larger multipliers or complex arithmetic multipliers.

For the multipliers implementation, we chose the *Radix-4 Booth* [73] algorithm—refer to Chapter 2 for the definition—which is a well-known and widely used technique for signed multiplication. There are two reasons for selecting the Radix-4 Booth multiplier: (1) by modifying the PPG structure of the Radix-4 Booth multiplier and performing some transformations on the sign extension parts of the PPs, we can significantly reduce the costs of adding new multiplier bit-widths to the base architecture, and (2) the PPRT of the Radix-4 multiplier has half the PPs compared to other parallel multipliers, and this allows the PPRT unit for efficient implementation of compressor trees. The latter advantage is a key factor in designing compressor trees for multi-input addition. We will provide more details about these advantages of the Radix-4 Booth multiplier, once we unveiled the complete architecture in subsequent sections.

In the following, we give a brief overview of the base DSP block architecture, as it is needed to understand the modifications that we will make to increase the flexibility. There is a couple of differences from the standard Radix-4 architecture in the way that we design the PPG and PPRT units. For the PPG, to multiply the multiplicand by 1, 2, or 0, all that is needed is to shift the multiplicand using a few multiplexers, which have a delay time that is independent of the size of the inputs. The only complexity relates to negating a 2's complement number, where a 1 is added to an inverted number. This complexity can be avoided in the PPG, if we move the summation part into the PPRT unit. For this purpose, a correction bit corresponding to each PP is added to the PPRT. Figure 5.4 illustrates the PPG unit of the Radix-4 multiplier. For each PP, one Booth encoder is required, while for each bit of PP, we need a *Booth Selector* unit shown in this figure, where all the Booth Selector units of a PP are controlled by the same signals.

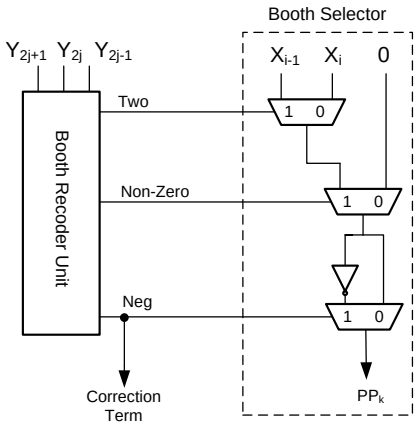


Figure 5.4: The Radix-4 Booth PPG unit. Booth encoder is shared between all PPs bits, but each bit of PP needs a separate Booth Selector unit.

For a Radix-4 Booth 18 × 18 multiplier, the PPRT unit is 36-bit wide and its height is nine, since we have nine PPs. Hence, we propose to exploit a layer of 9 : 2 compressors followed by a final CPA for the PPRT unit. In this layer, there is a 9 : 2 compressor per each column, and thus for the 18 × 18 multiplier, thirty-six 9 : 2 compressors are required. Figure 5.5 shows the proposed circuit level design of the 9 : 2 compressor. All of its inputs, including the carry bits, have the same rank, i.e., i. The two outputs also have rank i, but the carry outputs have rank $i+1$. The delay of 9 : 2 layer is independent of the layer width, since no ripple carry path exists in the layer. The longest path that a carry can propagate contains three cells, as show in Figure 5.6.

Since the compressor layer will be reused for other DSP block configurations, the 9 : 2 layers of all 18 × 18 multipliers in the base DSP block are chained, but at the multiplier boundaries, there is the option to separate two 9 : 2 layers, setting the carry input bits of the each layer to 0 by simple AND gates.

As explained, besides the PPs, we have a number of *Correction Bits (CBit)* that need to be added to the PPRT unit. Each CBit is aligned with the first bit of the corresponding PP. Due to the shifting of the PPs, there is always a free place in the 9 : 2 layer for every CBit, except for the ninth one. This means that one column of the PPRT will have ten bits, and thus requires 10 : 2 compressor instead of 9 : 2 to compress that column. In addition, inserting the 10 : 2 compressor, necessitates to change all the subsequent compressors to 10 : 2 as well, and this increases the delay of the PPRT unit. To avoid the use of 10 : 2 compressors, we merge the ninth CBit with the first PP as shown in Figure 5.7. In this figure, S is the sign bit of the PP, and C is the CBit. Since the CBit is aligned with the seventeenth bit of the PP, the MSB bits are modified from that bit position, as shown.

In addition to 9 : 2 layer, we need a 4 : 2 layer to sum the results of the two paired 18 × 18 multipliers according to the structure of the base DSP block that was shown in Figure 5.3. A 4 : 2 compressor is structurally similar to the 9 : 2 compressor, but it has fewer number of

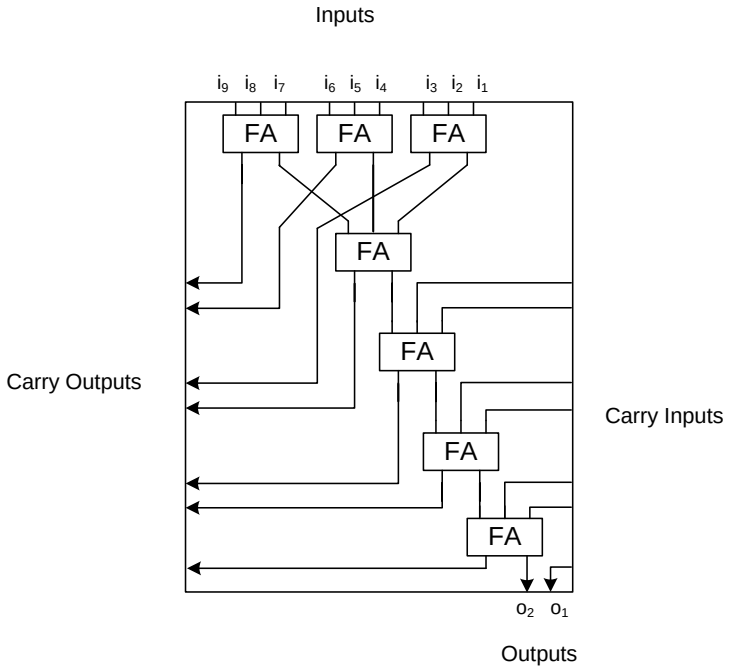


Figure 5.5: Structure of the 9 : 2 compressor in the proposed DSP block. This compressor has nine inputs and two outputs, in addition to the carry inputs and outputs. All the inputs, including the carry inputs, have the same rank.

inputs and carry bits. This layer is added between the 9 : 2 layer and the final CPA. This layer is 36 bits wide for each multiplier pair, as shown in Figure 5.8.

5.3.2 Supporting Various Multiplier Bit-widths

In this section, we will describe how we can reduce the costs of adding new multiplier bit-widths to the base DSP block. For this purpose, we modify the PPG unit of the Radix-4 Booth multiplier and remove the sign extension parts of the PPs. To support various multiplication bit-widths, we modified the PPG to be able to reuse the PPRT for all the configurations. The PPG, indeed, provides the required inputs for the PPRT, based on the DSP block configuration. This flexibility of the PPG, however, can increase the complexity of the PPG significantly. For instance, as shown in Figure 5.9, when a new bit-width is added to the base DSP block, for a certain bit position, we may need to select between a Booth encoded bit or a sign bit. This requires extra multiplexers for the selection of the right input bit of the PPRT.

In Figure 5.9, the PPG corresponding to the first PP of an 18 × 18 multiplier is illustrated on the top. Within the same number of bits, two 9 × 9 multipliers can fit. In the same figure, the PPGs of the two 9 × 9 multiplies are shown below that of the 18 × 18 multiplier. Since both configurations use the same PPRT unit, we need to exploit several multiplexers to choose between these two configurations. These multiplexers select between the encoded bit or the

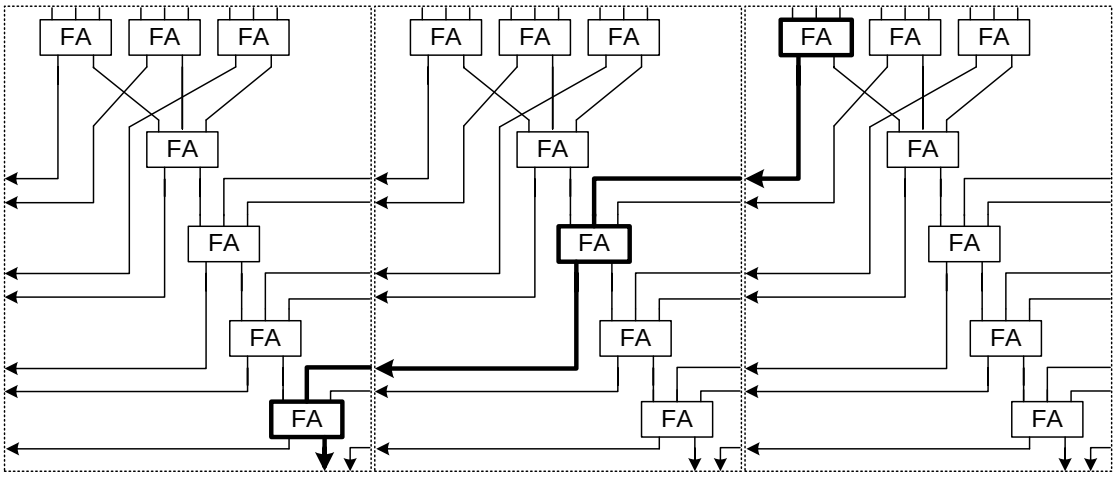


Figure 5.6: A chain of three 9 : 2 compressors. The longest path that a carry output can propagate includes two compressors, as shown in this figure. Hence, the delay of a 9 : 2 compressor layer remains constant when the number of compressors in the layer varies.

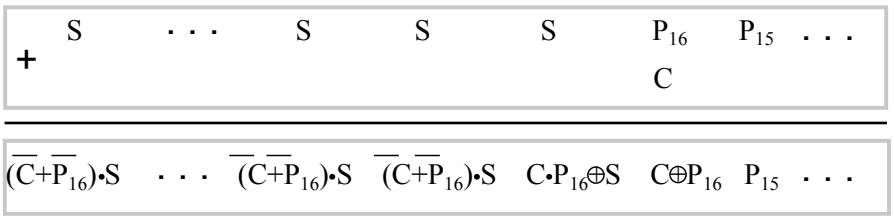


Figure 5.7: Merging the ninth carry bit with the first PP. MSB bits of the PP from bit 16 are modified.

sign bit of two different multiplier configurations. However, some parts of the multipliers encoders can overlap, and thus only some multiplexers at the inputs of the encoders are required, instead. In the example of Figure 5.9, such an overlap occurs for the first nine bits, and since the encoder inputs are the same, no extra multiplexer is required for these bits. Since, in most of the cases, multiplexers are required to select between a sign bit and another bit, one efficient technique for reducing the complexity of the PPG is to eliminate the sign extension parts of the PPs. For this purpose, we use a technique similar to the one that is used in Baugh-Wooly multipliers [73]. In this technique, as the first step, the sign extension part of each PP is first added with +1 and then with −1. As shown in Figure 5.10, the whole sign part is reduced to a single inverted sign bit, when it is added with +1. Hence, the sign extension part of each PP is transformed to a single inverted sign bit, which needs to be added with −1.

After applying this rule to the sign extension parts of all N PPs, we will have N constant numbers—unaligned −1s—and N single inverted sign bits. Now, we can reduce the N constant numbers to only one number, by summing them up, as shown in Figure 5.11. Since the first bit of the resulting constant number is aligned against the inverted sign bit of the first PP, we can append the constant number to the first PP, as shown in Figure 5.12.

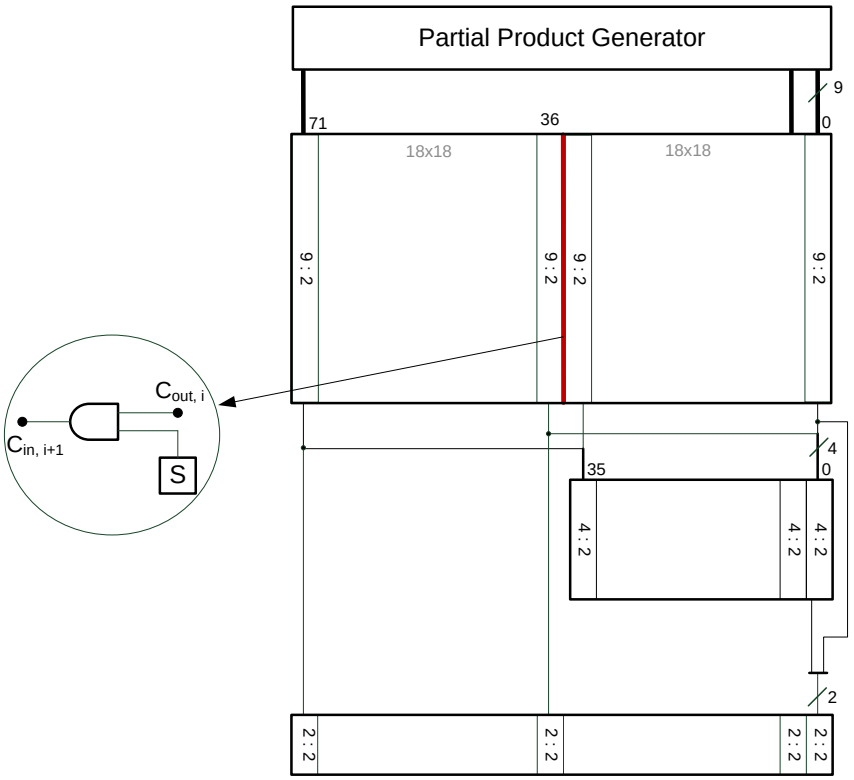


Figure 5.8: The compressor tree structure of each multiplier pair in Figure 5.3. The compressor tree includes one layer of 9 : 2 compressors followed by one layer of 4 : 2 compressors and the final CPA adder. The 9:2 layer can be split into independent 9:2 layers at the multipliers boundaries by disconnecting the carry paths using the shown AND gates.

With this technique, the sign extension parts of all the PPs is eliminated, except for the first PP, in which the sign part will include three sign bits appended by a constant value. In this case, the PPG unit, which supports various bit-widths, should select between constant bits—0 or 1—single sign bits (inverted or non-inverted), and the normal Booth encoded bits. To design such a PPG, instead of using multiplexers, we modified the Booth Selector unit of the Radix-4 Booth PPG in Figure 5.4 and added two extra control signals as shown in Figure 5.13. Compared to the original design, two control signals, *Const* and *Inv*, and two 2-input gates are added on the selection logic of the last two multiplexers in Booth Selector. When *Inv* signal is set to 1, the output of Booth Selector is inverted, and when *Const* bit is set to 1, 0 is selected as the output of the second multiplexer. Table 5.1 shows the operation modes of the modified PPG based on these two control signals.

With the modified Booth Selector, to generate a constant value—either 0 or 1—the *Const* signal should be set to 1, and *Inv* will specify the constant value. To produce the inverted sign bit, we only need to set *Inv* to 1. For a normal encoding, the two control signals are set to 0.

In contrast to PPG, the PPRT unit does not require any major modification. The PPRT was

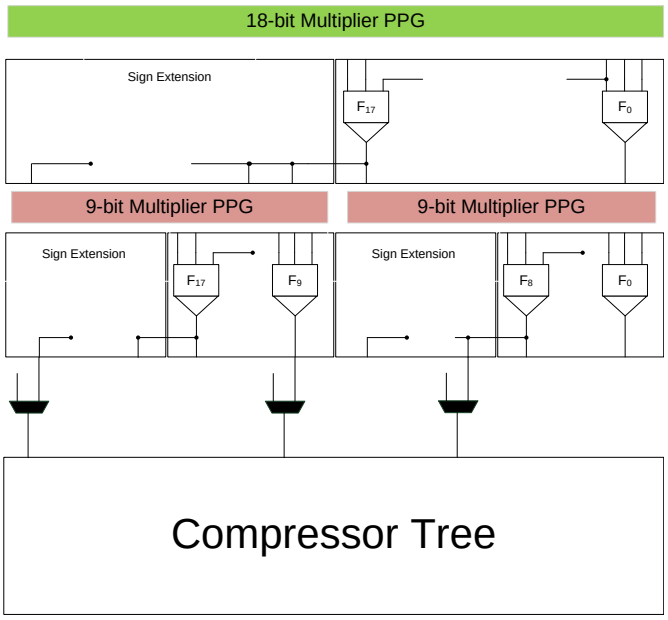


Figure 5.9: Overlap between the PPG of two different multiplier configurations. Since the same PPRT is used for both configurations, several multiplexers are required to select between either the encoded or the sign bits of the two configurations.

$$
\begin{array}{lccccccc}
 & S & \cdots & S & S & S & S & S \\
+ & & & & & & & 1 \\
 & 1 & \cdots & 1 & 1 & 1 & 1 & 1 \\
\hline
+ & & & & & & & \overline{S} \\
 & 1 & \cdots & 1 & 1 & 1 & 1 & 1
\end{array}
$$

Figure 5.10: Reducing the repetitive sign bits by adding with ±1.

designed in a way that can be reused for various bit-widths. For the bit-widths smaller than 18, since the number of PPs is less than nine, the 9 : 2 layer suffices to compress the bits. Nevertheless, for the larger bit-widths up to 36, we should split the PPs into two sets and compress each set separately by two disjoint chunks of 9 : 2 compressors. Then, we need to sum the results of the two chunks. For this purpose, we use the 4 : 2 layer of the base DSP block. The number of 9 : 2 slices that are required for a set of PPs is obtained from Equation 5.1. In this equation, MulBW represents the multiplier bit-width.

$$SliceBW = MulBW + 2 \times NumofPPs \tag{5.1}$$

As an example, in 36 × 36 multiplier there are 18 PPs, and thus two independent chunks of the 9 : 2 layer are used to compress each 9 PPs. The width of each chunk is 54–number of 9:2 slices–which is determined using Equation 5.1. In total, we need to allocate 108 slices of the 9:2 layer to the 36 × 36 multiplier. Similarly, for a 24 × 24 multiplier, we need 72 slices, in

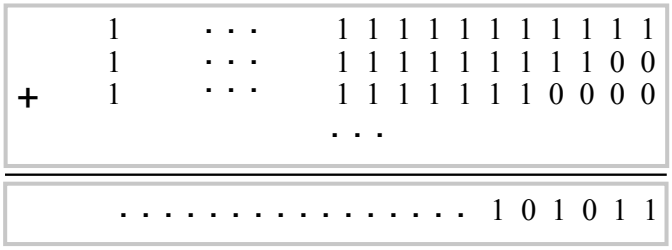


Figure 5.11: Reducing the constant numbers to one number.

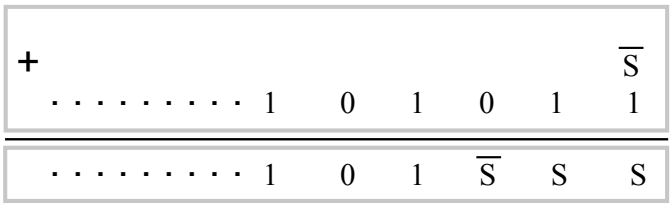


Figure 5.12: Merging the constant number into first partial product.

which we have 12 PPs that are compressed using two independent 9 : 2 chunks that have 42 and 30 9 : 2 slices, respectively. Next, we need to add the results of the two chunks in these two multipliers using the 4 : 2 layer and the final CPA.

5.3.3 Supporting Multi-input Addition

In VLSI, multi-input adders are realized as compressor trees, such as Wallace [88] and Dadda [26] trees. The building block of compressor trees can be carry-save adders, counters, or compressors, as described in Section 2.3 of Chapter 2. In principal, smaller building blocks are preferable, as they are rarely underutilized and they build more efficient compressor trees.

From the multi-input addition perspective, the advantage of the Radix-4 Booth multiplier over parallel array multipliers is its fewer PPs; hence, the compressors that are used in the PPRT design would have smaller size, which can result in having faster compressor trees. Moreover, the PPG unit should be bypassable to be able to access the PPRT for implementing multi-input additions. This is a missing feature in current FPGAs DSP blocks.

Assuming that there is no connectivity constraint between the DSP block inputs and the

Const	Inv	Func
0	0	PP_k
0	1	$\overline{PP_k}$
1	0	0
1	1	1

Table 5.1: Operation modes of the modified Radix-4 Booth PPG.

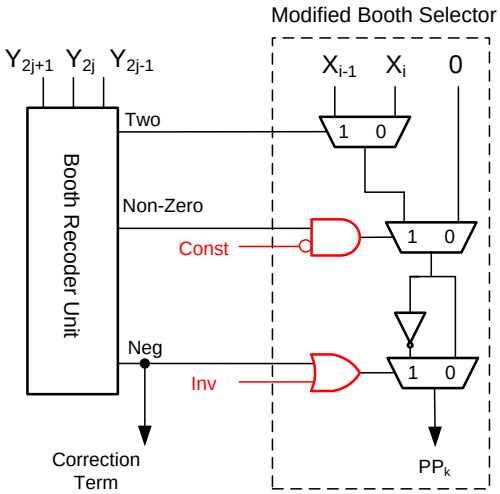


Figure 5.13: The modified Radix-4 Booth PPG encoder for resolving the conflicts of PPG parts of various multiplier bit-widths.

Block	Height	Width
0	9	8
1	8	9
2	7	10
3	6	12
4	5	14
5	4	18
6	3	24

Table 5.2: Different rectangular blocks that are used for the mapping of the inputs bits of the adder tree.

inputs of the PPRT, a properly designed PPRT can be utilized to efficiently map any regular and irregular multi-input addition. However, such a connectivity requires a fully populated crossbar, which is extremely costly. Therefore, the real challenge is to bypass the PPG and use the inherent flexibility of the PPRT compressor tree with minimum overhead.

Our solution for this problem is to define a set of fixed *ReCtangular (RC)* blocks within the 9 : 2 layer of the PPRT–see Figure 5.14–that have predefined connectivity to the inputs of the DSP block. This fixed connectivity reduces the overhead of accessing the PPRT for performing multi-input addition and removes the need for any crossbar. Moreover, having different RC-blocks enables more flexible mapping of the bits that need to be compressed. In other words, based on the bits pattern that have to be compressed, a different combination of the RC-blocks is used.

Table 5.2 shows the dimensions of each RC-block within the 9 : 2 layer of Figure 5.8. Figure 5.14 indicates how different RC-blocks are aligned and overlapped. The maximum height of an

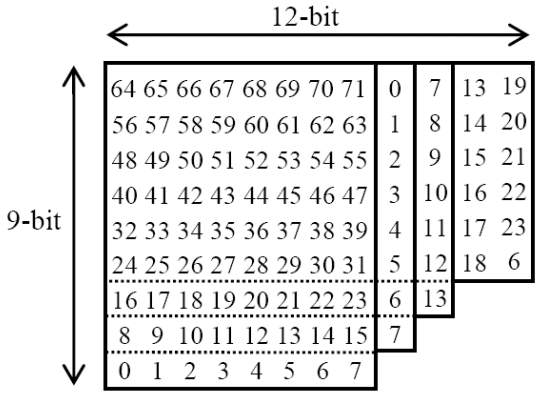


Figure 5.14: The indices of the DSP Block inputs that are connected to each *Rectangular (RC)* block. RC-blocks are aligned for maximum input sharing, and each RC-block is connected to distinct DSP block inputs.

RC-block is nine, since it should fit into the 9 : 2 layer. The width of an RC-block in this table, determines the required number of chained slices to map the RC-block. Assuming that there are two (connected) 9 : 2 layers in the DSP block, one per multiplier pair, the RC-blocks of Figure 5.14 shows alignment of the RC-blocks that reside in the right 9 : 2 layer. For the left one, the alignment of the RC-blocks is mirrored, provides the opportunity to form larger RC-blocks as well as non-RC-blocks. For example, RC-block 0 in the right layer can be chained to RC-block 5 in the left layer, which forms a non-RC-block for covering the input bits; similarly, two identical RC-blocks can be chained and form a wider RC-block with the same height.

Figure 5.14 shows which inputs of the DSP block are connected to each RC-block that resides in the right 9 : 2 layer–this corresponds to the right multiplier pair of Figure 5.3. In this figure, due to the lack of space, only the first four RC-blocks in Table 5.2 are shown. The numbers inside each RC-block specify the DSP block input indices. The RC-blocks of the right layer take the first 72 and the ones in the left layer take the last 72 inputs of the DSP block–the figure only shows the indices of the RC-blocks in the right layer. Note that the PPG unit should be bypassed for the indicated bit positions. Therefore, we insert some multiplexers into the PPG unit for this purpose. To minimize the number of multiplexers, the RC-blocks are maximally overlapped, as shown in this figure.

5.3.4 Multi-input Addition Mapping Algorithm

In this section, the mapping algorithm of multi-input addition is described. Mapping algorithm has a significant impact on the quality of the results, since efficient covering of the input bits should be achieved by the appropriate selection of the RC-block.

To decide which block is more efficient for mapping a set of bits, we evaluate each block by its ability to compress the input bits, considering the utilization efficiency. For this, we define

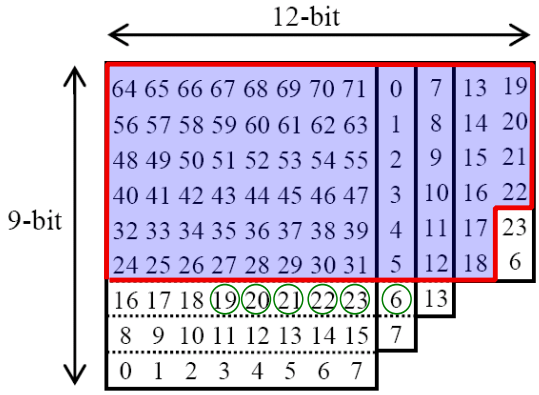


Figure 5.15: Block refinement. The underutilized column in the RC-block provides the opportunity to cover more bits in other columns. The circled indices are the mapping candidates, from which two can be covered.

Compression Ratio (CR) parameter for a block, which is defined as follows:

$$CR = \frac{\sum_{i=0}^{W_b} h_i}{2 \times W_b} \tag{5.2}$$

In this equation, W_b is the width of the bits that are generated by a block and h_i is the number of bits in the i^{th} column of the block. The higher the *CR* is, the greater the overall compression ratio of the block will be. Blocks with higher CRs tend to compress more. Here, the block can be a single RC-block or any combination of two RC-blocks. Using this metric, all possible blocks are examined for the mapping.

Although, the maximum height of a column for a specific RC-block is limited to the numbers illustrated in Table 5.2, in certain situations, more bits can be mapped to some columns that have less than nine bits, if the heights of the other columns do not reach the expected height. As an example in Figure 5.15, RC-block 3 covers a set bits, where all of block's columns are filled, except the right most column—columns 1 to 11 cover six bits, while column 12 covers only four bits, as there is not enough bits to fill the column. This provides the opportunity to add more input bits to other columns—outside the RC-block—using the available input bandwidth of the PPRT. This is possible, for the columns that have common index with column 12–see the circled indices, which are the mapping candidates. Hence if there is any uncovered bit in these columns, then two of them can be covered. By using this trick, we can improve the coverage of the mapping block. This process is called *block refinement*.

The mapping algorithm of multi-input addition has three key steps: In the first step, the best block for covering a set of input bits is determined. This block can be either a single RC-block or two joined RC-blocks, as explained. The best block is the one that has the highest CR after the block refinement. Next, the covered bits are removed from the uncovered set of bits and this step is repeated until we meet the termination condition. Termination condition is either

covering all the input bits or being unable to find a block with more than 50% covering ratio for the mapping.

The second step of the mapping is generating the output bits of the blocks that were used for the mapping in the first step. For this, the CPA of the DSP block is used, if no other levels of DSP blocks is required to compute the results; otherwise, the final adder is skipped, and the output will remain in the form of carry and save.

Finally, in the third step of the mapping, we bind the selected blocks for the mapping to the PPRT of the DSP blocks, and we specify the interconnection of the DSP blocks, based on the mapping. Each DSP block can hold two RC-blocks, one in each half; hence, each two independent RC-blocks can be placed in the same DSP block. Meanwhile, the joined RC-blocks should be placed in the same DSP block, where the two halves are chained.

5.4 Experiments

To evaluate the proposed DSP block, we designed a sample base DSP block with 144 inputs and outputs. This is a typical IO bandwidth of a DSP block in the Altera Stratix series FPGAs [7]. Since we use 90nm CMOS technology for the DSP block design and we have to estimate the inter DSP block net delays for our experiments, Stratix-II [6] was selected for the comparison, which is fabricated in 90nm process technology. Considering the above IO bandwidth, we designed a DSP block that contains two of the building blocks shown in Figure 5.8, one per each 18×18 multiplier–the $9:2$ layer is 144 bits wide in this DSP block.

For the experiments, we add 9/12/24/36 multiplier bit-widths to the base DSP block and we evaluate the overhead of adding each. We also measure the overhead of supporting multi-input addition. Based on the available bandwidth and resources, the designed DSP block can implement up to eight 9×9, six 12×12, two 24×24, and one 36×36 multipliers. To implement the 36×36 multiplier, as explained in Section 5.3.3, 108 slices of the $9:2$ compressor layer are used. Hence, it is possible to use the rest of the slices for multi-input addition in parallel with the 36×36 multiplier.

We modeled the sample DSP block in Verilog and used Synopsys Design Compiler with 90nm Artisan standard cell library for the synthesis. The mapping algorithm of the multi-input addition was developed in C++ programming language. To estimating the delay of the DSP block interconnection wires, we used Altera Quartus II tool. For this purpose, we used the virtual embedded block methodology [38], where we replaced our DSP blocks in the netlist with the Stratix-II DSP blocks, and we performed the placement and routing.

5.4.1 Results

To measure the overhead of supporting each multiplication bit-width and multi-input addition, we first synthesized the base DSP block, and then we added the features one by one. Table 5.3

DSP block Features	Delay (ns)	Area (μm^2)
Base DSP block	3.11	41439 (1.00)
Base DSP block+9 × 9	3.11	43269 (1.04)
Base DSP block+9 × 9/12 × 12	3.12	44633 (1.07)
Base DSP block+9 × 9/12 × 12/24 × 24	3.14	45852 (1.11)
Base DSP block+9 × 9/12 × 12/24 × 24/36 × 36	3.15	46271 (1.12)
Base DSP block+9 × 9/12 × 12/24 × 24/36 × 36 and MADD	3.15	46973 (1.13)

Table 5.3: Overhead of adding new features to the base DSP block. The delay numbers show the 18 × 18 multiplier delay in each case.

Multiplier	Our DSP block	Stratix-II DSP block
9 × 9	1.89	2.99
12 × 12	2.43	-
18 × 18	3.15	3.17
24 × 24	4.01	-
36 × 36	5.17	4.57

Table 5.4: Delay comparison of the multipliers in our DSP block with the Stratix-II DSP block. For our DSP block, these numbers are achieved when all features presented in Table 5.3 are included.

presents the synthesis results of our DSP block, before and after supporting of each feature. The delay values show the delay of the 18 × 18 multiplier in each case, and the area is the total area of the DSP block after supporting a certain feature. The interesting point is that the delay overhead of adding new features to the base DSP block is remarkably low; while the total area overhead of supporting all bit-widths is 11%; this means that, on average, less than 3% area overhead is imposed by supporting each multiplication bit-width. Finally, the area overhead of supporting multi-input addition is only 2%.

Table 5.4 shows the combinational delay of each multiplier in the final DSP block with all features included. These numbers can further be improved by inserting some pipeline registers between the layers of the DSP block. In Stratix-II, the combinational delays of 9 × 9, 18 × 18, and 36 × 36 multipliers are 2.99ns, 3.17ns, and 4.57ns, respectively. Note that, the delay of the 9 × 9 multiplier of the DSP block in [70] is 1.71ns, which confirms that the delay overhead of supporting various configurations in our DSP block is low.

To evaluate the multi-input addition feature, we synthesized the multi-input addition portions of some real arithmetic, multimedia and signal processing applications on the soft-logic of the Stratix-II FPGA—using the mapping technique of Chapter 3—and FPCT for the comparison purpose. Note that, the DSP blocks of current FPGAs are not usable for implementing multi-input addition. Table 5.5 shows the delay results. Compared to FPCT, our DSP block has a lower delay for the FIR benchmarks. On average, our DSP block is around 4% slower than

Benchmark	Soft-Logic	FPCT	Our DSP block
dct	4.32	2.46	3.08
H.264 ME	4.2	1.5	2.09
g721	4.31	3.22	3.79
adpcm	2.41	1.41	1.83
fir3	5.21	3.76	3.21
fir6	6.79	4.53	3.39
hpoly	5.55	3.36	3.81
Avg.	4.68	2.89	3.02

Table 5.5: Delays (ns) of multi-input addition benchmarks, when they are mapped on different logic blocks.

Benchmark	Soft-Logic (LAB)	FPCT (DSP block)	Our DSP (DSP block)
dct	17 (374 mm^2)	2(112 mm^2)	2.5(134 mm^2)
H.264 ME	11(242 mm^2)	4(224 mm^2)	1(48 mm^2)
g721	22(484 mm^2)	3(168 mm^2)	2.5(134 mm^2)
adpcm	3(66 mm^2)	1(56 mm^2)	0.5(24 mm^2)
fir3	35(770 mm^2)	6(336 mm^2)	3(144 mm^2)
fir6	76(1672 mm^2)	8(448 mm^2)	6(288 mm^2)
hpoly	35(770 mm^2)	3(168 mm^2)	3.5(182 mm^2)

Table 5.6: Areas of multi-input addition benchmarks, when they are mapped on different logic blocks.

FPCT, which exclusively performs multi-input addition. Compared to the soft-logic, our DSP block has a lower delay for all the benchmarks and on average is 35% lower.

Area comparison of these three methods is not that straight forward. Table 5.5 presents the area of each benchmark in terms of the basic blocks that are used. For the soft logic, the area indicates the number of LABs, for FPCT and our DSP block, the numbers represent the number of DSP blocks. The area of a LAB is 22000μm^2 [90]; the area of FPCT is approximately 56000μm^2, while ours is less than 48000μm^2, with all mentioned features.

5.5 Related Work

Fixed-function ASIC intellectual property cores [96], such as the aforementioned DSP block, are widely integrated into FPGAs. Our work seeks to enhance the functionality of these blocks so that they can perform multi-operand addition as efficiently as they currently perform multiplication and MAC. Other recent academic proposals suggest the integration of floating-point units, shifters, multiplexors, and crossbars [10, 18, 38, 42, 43].

5.6 Conclusion

DSP blocks are typically adapted to critical arithmetic operations of applications, such as multipliers. However, these DSP blocks have little flexibility—i.e., they support only few multiplication bit-widths or they do not support multi-input addition—and thus few applications can benefit from them. In this chapter, we presented a versatile architecture for the DSP blocks of FPGAs, starting from a base DSP block with basic features of the current FPGAs DSP blocks. This base DSP block was structurally designed in a way that new features, such as different multiplication bit-widths and carry-save-based multi-input addition, can be supported with little extra overhead. For this purpose, we used our prior experience in designing DSP blocks for carry-save arithmetic and the challenges we faced for adding the features current FPGAs DSP blocks. This was the main reason that we started from a DSP block with the current basic features.

In this work, we chose the Radix-4 Booth multiplication technique, and by modifying its Booth encoder and removing the sign extension parts of its PPs, we provided the opportunity in the base design to support new multiplication bit-widths with low cost. Moreover, by proper design of the multiplier's PPRT and bypassing the PPG, we could support multi-input addition. Since each DSP block contains multiple base multipliers, our DSP block has the option of concatenating the PPRTs of all its internal multipliers to form a configurable compressor tree pool for performing carry-save arithmetic.

Experimental results revealed that using the presented DSP block for implementing compressor trees significantly reduces their delay and area, compared to the soft-logic implementations of the compressor trees that were presented in Chapters 3 and 4. Therefore, the first choice for implementing the compressor trees on FPGA is using the presented DSP block. However, the number of DSP blocks in an FPGA is limited, and it is still crucial to have efficient soft-logic implementation of the compressor trees on FPGAs, when no free DSP block exist. This justifies the contributions of Chapters 3 and 4.

To summarize, having a DSP block that can be adapted to the needs of a richer set of applications follows the first goal of this thesis, which increases the generality of the hard-logic in FPGAs. Although our DSP block is more versatile than the current ones and has extra features, there is still room to increase the functionality of these useful logic blocks and they can be enriched with other arithmetic and non-arithmetic operations based on the new requirements of the emerging applications; this maximizes the use of these expensive hard-logic logic blocks as well as the costly routing resources that surround these blocks.

6 Logic Chains

In the last three chapters, we presented our contribution in increasing the flexibility of the FPGAs hard-logic by adding more functionalities to them. That was the first step for reducing the FPGAs and ASICs efficiency gap according to the thesis *roadmap* that was presented in Chapter 1. Meanwhile, based on this roadmap, the second goal of the thesis is enhancing the efficiency of the FPGAs soft-logic. This is a crucial step, as each application that is mapped to FPGAs fully or partially uses the soft-logic for its implementation. Hence, the soft-logic is the primary efficiency bottleneck in FPGAs, which needs to be improved.

In contrast to the hard-logic, the soft logic of FPGAs is so flexible, which allows to implement any circuit. This flexibility, however, comes at a price, which is the soft-logic inefficiency. Our approach to improve the soft-logic of FPGAs is based on the fact that there are logic and connection patterns in applications, which we can take advantage of to reduce the excess flexibility of the soft-logic to enhance its efficiency. In contrast to the hard-logic, which is typically adapted to the pre-synthesis operations, here we look for the logic and connection patterns that appear post-synthesis. The main reason is that when applications pass through logic synthesis, the probability of finding common patterns increases.

In this chapter, we introduce *logic chain* idea, which can replace the interconnecting routing wires in long chains of logic that are observable after logic synthesis and even after technology mapping. Generally, due to the long delay of the routing wires, the circuits paths that include more number of routing wires become more critical in the FPGA implementation. Hence, it is crucial to replace such costly wires by fast hard-wired connections. Inspiring from the carry chains, we present logic chains, which are more general than carry chains and can be used for the generic logic synthesis. Logic chains can replace the interconnecting routing wires, and thus they reduce the stress on the routing network and provide the opportunity to make these costly resources lighter in the future FPGAs. Moreover, using the logic chains, the logic density of the FPGA logic blocks is improved, as extra input and output bandwidth is afforded without changing the logic block interface with the routing network.

6.1 Introduction

The programmable interconnect fabric dominates silicon area in modern high-performance FPGAs. The fraction of silicon dedicated to programmable routing increases with each successive technology generation, because transistors scale more effectively than wires. This trend directly impacts the performance and power consumption of FPGAs. Moreover, the feasibility of synthesizing a circuit onto an FPGA can be limited by the availability of routing resources, rather than programmable logic.

Due to the fine-grained nature of the soft-logic of FPGAs, routing wires have a considerable impact on the soft-logic implementation of a circuit. For example, a circuit delay has a direct relation with the number of logic levels, where an interconnecting routing wire is used to connect the logic of two different levels. To reduce the negative impact of routing resources, a number of architectural innovations have been proposed in recent years. Having FPGAs with more coarse-grained soft-logic blocks, which have more and larger LUTs, is one solution that we see as a trend in recent FPGAs. This allows to fit more logic in each logic block, and thus the number of logic levels is reduced and less routing wires are used.

The other solution is to divide the routing network into global and local components. This enables to have fast local routing within clusters of logic—e.g., *Logic Array Blocks, or LABs* in the Altera's FPGAs—and reduces the demand for global routing resources between the clusters. Moreover, the introduction of carry chains within logic clusters allows for the efficient propagation of arithmetic carries along a fixed wire, native to the carry chain; consequently, these wires are entirely moved out of the programmable interconnect.

Our solution to reduce the demand for programmable routing resources is based on replacing the routing wires with hard-wired connections that are established among the soft-logic blocks. Indeed, exploring long and independent chains of logic that appears in the post synthesis netlists of applications, motivated us to integrate fixed local connections between the soft-logic blocks of FPGAs; this so-called *logic chain* is similar in principle to a carry-chain, but connects programmable logic resources rather than fixed-function gates that can only perform carry-propagate addition. Through effective decomposition and mapping algorithms, it is possible to map logic functions of nontrivial size onto the dedicated logic chains, rather than using programmable interconnect. This reduces pressure on the routing network, as nets from the technology-mapped circuit are moved onto dedicated wires in the logic chain.

The other benefit of the logic chain is that it enhances the utilization efficiency of the LUTs in the logic block by providing more bandwidth; though, the logic block interface with the programmable routing network does not change, which is an advantage. Efficient utilization of the logic block resources, indeed, improves the logic density of the logic blocks. Nevertheless, all these benefits are achieved by adding a few multiplexers and configuration SRAMs to the structure of the logic block.

The following section begins with a simple example to illustrate the main idea.

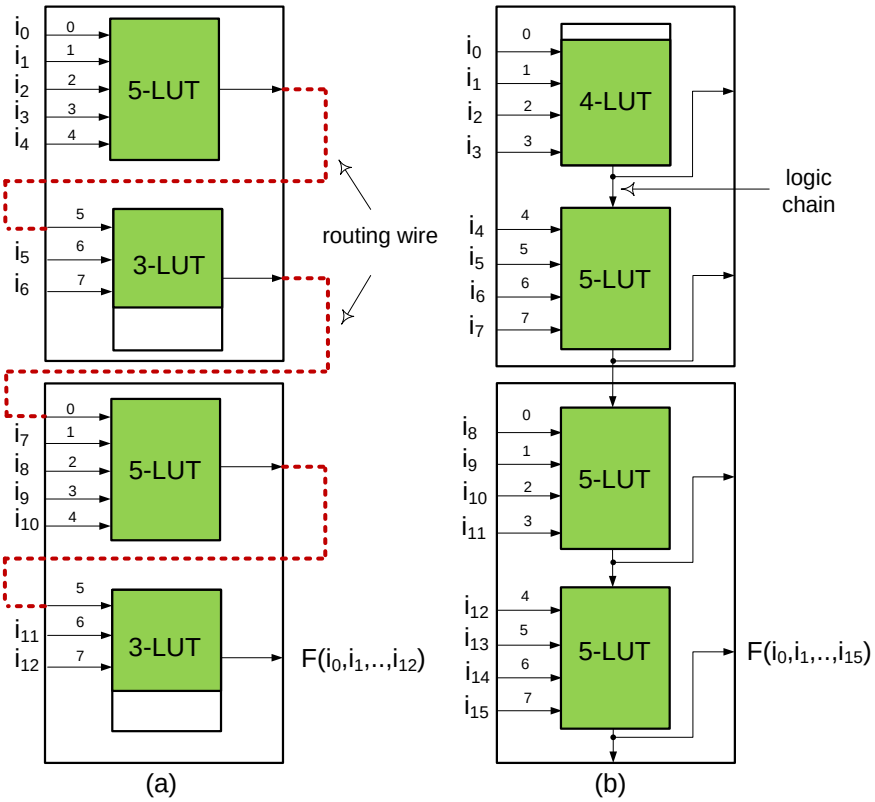


Figure 6.1: Key idea. (a) Two logic blocks, each has eight inputs and two base 5-LUTs. Many 13-input logic functions can be mapped to a linear cascade of the base LUTs; routing resources are required to connect adjacent LUTs in the cascade. (b) A dedicated logic chain between adjacent LUTs eliminates the overhead due to routing resources and increases the input bandwidth of logic block. Many 16-input logic functions can be mapped with the same number of available LUTs.

6.1.1 Key Idea

The motivation behind logic chains comes from the observation that there are long chains of logic that appear after logic synthesis, and even when the technology mapping is performed, we still see long and independent chains of logic functions that are mapped to LUTs. Based on our experiments, on average, 85% of the LUTs are chainable to the logic block structure that we present in this chapter. Suppose that a circuit has been synthesized in such a manner that four LUTs are cascaded linearly and form a chain. Figure 6.1 illustrates how this circuit is differently implemented on two different FPGAs, one with a conventional logic block and the other with the new logic block that has the dedicated logic chain.

In Figure 6.1 (a), there are two FPGA logic blocks, each having eight inputs and a fracturable LUT structure. In the fracturable LUT structure, each logic block has two base 5-LUTs, and a larger 6-LUT is formed using the two sub-LUTs—5-LUT—followed by a multiplexer, which is not shown in this figure.

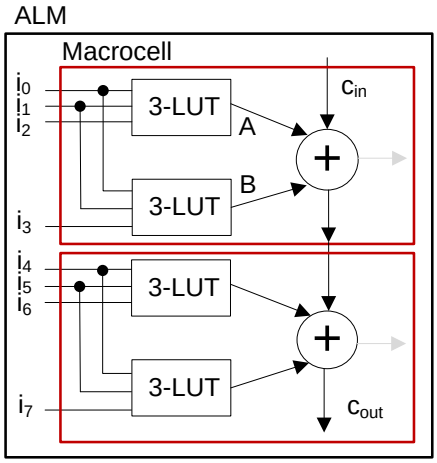


Figure 6.2: Proposed configuration for the ALM, using the adder and the carry chain for generic logic synthesis. Each ALM can implement two chained 5-input functions with non-shared inputs.

One drawback of this particular architecture is that the routing resources must be used to connect one sub-LUT to its successor in the cascade. The other drawback is that the second sub-LUT in each logic block is underutilized, because of the input bandwidth constraint of the logic block. The first five inputs are used by the first sub-LUT, and the remaining three inputs are used by the second sub-LUT. This means that a 5-LUT is used to implement a 3-input function.

Figure 6.1 (b) has the dedicated logic chains within the cluster so that the output of each sub-LUT connects directly to the input of the subsequent sub-LUT along the chain; in principle, this is similar to the interconnection structure of arithmetic carry chains in current commercial FPGAs. The introduction of these direct connections eliminates the need to use the global routing network to synthesize the cascade. This has several advantages: reduced pressure on the routing network, reduced critical path delay and reduced power consumption. Moreover, the logic chain provides a way that the available sub-LUTs are utilized more efficiently, since the input bandwidth of the logic block is increased by the logic chain without any change in the local routing network of the soft-logic clusters.

Comparing these two figures, we see that with the new logic block we can map bigger functions to the same number of logic blocks. The logic blocks in Figure 6.1 (a) can implement many functions with 13 inputs, while the logic blocks in Figure 6.1 (b) can implement many 16-input functions.

6.1.2 Carry Chain Option

Prior to this work, we also explored that the ALM in the Altera Stratix-II/V FPGAs can be configured to a unintended operating mode, which enables it to implement two cascaded

5-input functions—refer to Chapter 2 for the ALM structure and its operating modes. In this new configuration, as shown in Figure 6.2, each ALM is split into two *macrocells* with distinct inputs. In contrast to Arithmetic and Shared Arithmetic modes of the ALM, no sharing between the inputs of the two halves of the ALM is occurred; each macrocell takes four of the ALM's inputs and the carry input to each macro cell is considered as the fifth input. The output of this macrocell is the carry output.

Although each macrocell is a 5-input block, a subset of 5-input functions can be mapped to this cell. Therefore, for the mapping purpose, one main step is to verify whether a given function (originally targeted to a universal LUT) is mappable to the macrocell. Consequently, we introduced a new Boolean matching technique [72], in which we construct a library that represents all mappable functions. To construct this library, we used the regularity of the macrocell structure, which reduces the size of the library; this makes the Boolean matching process memory efficient and fast.

Once we selected the candidates, we used a chaining heuristic similar to the one that is presented in this chapter to map chains of logic on the carry chain. Since, only a subset of functions with five or fewer inputs are mappable to this structure and the inputs of the macrocell are not fully permutable, few chains of logic functions can be mapped on this carry chain.

Moreover, as the delay between the adder inputs and its carry-out, as well as the delay between the adder's carry-in and its sum output, are long compared to the carry-in to carry-out delay of the adder in an ALM, having short chains of logic functions is not economical and may deteriorate the overall performance of designs. In other words, the adders in ALMs have been optimized to have real fast carry-in to carry-out paths, by relaxing their other paths that are not critical for carry propagate addition.

However, using the carry chains for generic logic synthesis saves the usage of the routing wires as carry chains replace the interconnecting routing wires. This, indeed, can improve the routability of the circuits on FPGAs, especially the complicated ones that suffer from lack of routing resources despite the availability of the logic resources.

To summarize, current carry chains can be exploited for the mapping of generic logic, but due to their inherent constraints and the fact that they are not designed for generic logic synthesis, no performance benefit can be obtained from them. This indicates the need for having fixed connections similar to the carry chains, which are suitable for generic logic synthesis. The potential benefits of these generic chains include improved performance and logic density, in addition to reduced pressure on routing network of FPGAs.

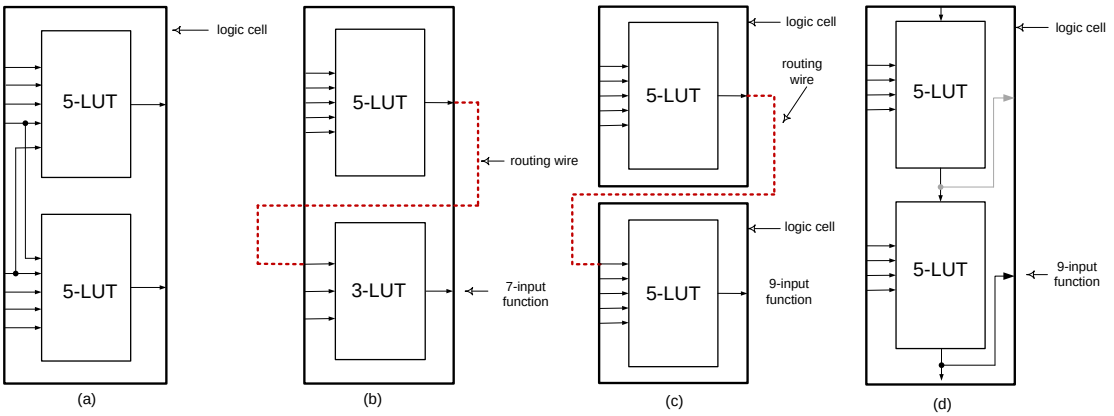


Figure 6.3: (a) The Altera's ALM configured to implement two 5-input logic functions; ALM imposes the constraint that the two functions must share two inputs. (b) Using fracturable LUTs, a subset of 7-input logic functions can be synthesized on an ALM, but this requires routing a signal from one sub-LUT to the next. (c) To implement two cascaded 5-input functions with no common inputs, two ALMs are required. (d) All three of the preceding logic functions can be synthezed on the proposed logic block using the logic chain and without using the global routing network; moreover, the proposed cell can implement a subset of 9-input logic functions.

6.2 New Logic Chain

Figure 6.3 (a) illustrates the structure of the ALM when it is configured as two independent 5-LUTs, which have two shared inputs; the two shared inputs are necessary because the input bandwidth of the ALM is eight. Consequently, if a user wishes to map two logic functions without shared inputs onto an ALM, the only possibilities are to use two 4-LUTs or a 5-LUT and a 3-LUT.

Figure 6.3 (b) illustrates a way that an ALM can realize a limited subset of 7-input logic functions: the 5-LUT is cascaded with the 3-LUT, however, the interconnection between the two requires the usage of the routing network. On the other hand, if a user wants to cascade two 5-LUTs with one another, then two ALMs are required, as shown in Figure 6.3 (c), once again using the routing network; if the two ALMs are placed within the same Logic Array Block (LAB), then the fast local routing network could be used instead of the global routing network.

The Altera's ALM is fracturable, meaning that several small (sub) LUTs exist natively in the ALM and can be concatenated together, via multiplexers, to form larger LUTs. Here, we use this approach to build larger LUTs out of the sub-LUTs along the dedicated vertical connection that we call *logic chain*. The basic idea is to cascade the current sub-LUTs in the ALM to form larger LUTs along a fixed connection similar to carry chains. Figure 6.3 (d) illustrates the main idea. The modified logic block now contains two 5-LUTs that are cascaded; one input of each of the 5-LUTs comes from the preceding 5-LUT in the logic chain; thus, only eight input signals are provided from the routing network, keeping the design within the bandwidth constraints

of the ALM. This allows the new logic block to implement a subset of 9-input logic functions, without requiring the routing network and assuming that one of the inputs comes from the preceding logic block along the logic chain; if the vertical input is unavailable, then it can still implement a subset of 8-input logic functions without using the routing network. This is significantly more powerful than the ALM, which can implement any 6-input logic function and limited 7-input crossbar switch without using the routing network.

The logic chain borrows many ideas from arithmetic carry chains, which also employ vertical connections between adjacent logic blocks. The goal of carry chains, however, is to improve the resource usage and critical path delay of addition/subtraction operations, which are common, but limited. One of the key benefits of these carry chains was that carry propagation was performed along the vertical connections, and therefore did not enter the routing network—this can avoid contention of routing resources. The vertical connections are shorter and do not have additional delays caused by configuration elements placed periodically along them; hence the critical path delay and power consumption is reduced. By integrating LUTs into the vertical connections, it is possible to synthesize a wide variety of operations, including addition/subtraction, onto the logic chains. Figure 6.4 shows how the fracturable structure of ALMs is used to embed the logic chain and form larger LUTs along the logic chain. Each half-ALM contains two 4-LUTs, which can form a 5-LUT using a multiplexer controlled by a fifth input; all inputs between the two 4-LUTs are shared. This design is effectively a Shannon decomposition. The shaded area of the figure illustrates the logic chain. Using a similar idea, we instantiate a vertical multiplexer, which is controlled by the logic chain, at the outputs of each pair of 4-LUTs; this forms a new 5-LUT along the logic chain. This provides us with the option to form either a horizontal or a vertical 5-LUT in each half-ALM. The output of the vertical 5-LUT propagates along the logic chain. In Figure 6.4, there is no way to access the output of the LUTs that are placed in the logic chain; this severely limits the ability to use the logic chain when an LUT placed in the chain has a fanout that exceeds one. The logic block of an FPGA already contains several multiplexers on its output: one to select between the LUT and carry chain outputs, and one to select the flip-flop's output, allowing for sequential circuits. We embed an additional multiplexer, as shown in Figure 6.5, to select between the carry chain output and the logic chain output. The shaded area in the figure indicates the additional logic that we add to the half-ALM to support the logic chains. The additional multiplexer that we have added will not increase the critical path delay of the non-arithmetic modes of the ALM, since it does not lie along those paths.

To estimate the area overhead of the new logic, we coarsely compare the transistor count of new logic and a simplified ALM. We have added four multiplexers and two configuration SRAMs; based on the components that are known to exist already in the Stratix-III ALM architecture [7], we are confident that the area overhead is less than 3%.

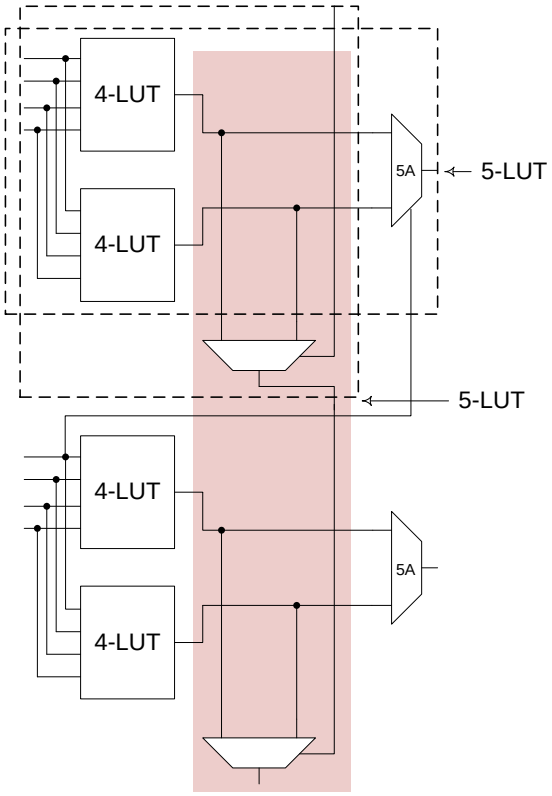


Figure 6.4: Integrating the logic chain into the ALM's structure. The shaded area indicates the logic chain. Existing 4-LUTs are cascaded using multiplexers to form vertical 5-LUTs along the logic chain. The fifth input of the 5-LUTs is the output of the preceding vertical 5-LUT, which is actually the logic chain. The key point of this structure is that the ALM input bandwidth remains the same, therefore two cascaded 5-LUTs with no shared inputs can be mapped to the new cell.

6.3 Chaining Heuristic

The objective of the mapping heuristic is to identify chains of logic having the maximum possible lengths. The input is a *Direct Acyclic Graph(DAG)*, in which each node represents a logic function and each edge represents the input and output dependencies among the functions. The DAG is generated after technology mapping, so each node is a prospective function that can map onto the LUTs. The number of inputs of each node in the DAG, K, cannot exceed the number of LUT inputs; each node has K child nodes and one or more parents based on the fanout of the node output.

The mapping heuristic visits the DAG nodes in *Depth First Search (DFS)* order, starting from the outputs and working back toward the inputs. The heuristic recursively assigns a depth to each node in the DAG.

Definition 6. *A node is **chainable**, if it has at most K inputs and is not part of another chain.*

Definition 7. *The **depth** of a node is the number of chainable nodes that can be accessed*

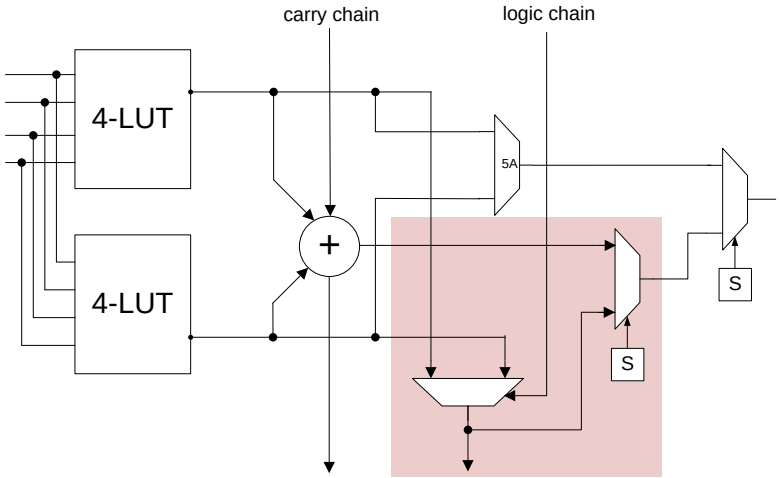


Figure 6.5: The logic chain integrated with the carry chain. In addition to the vertical multiplexer, a horizontal multiplexer is added to select between the sum output of the full-adder and the logic chain fanout; this multiplexer gives access to any point of the logic chain.

*consecutively through that node; the **depth** of an internal node is the maximum depth among all input nodes from which it is reachable.*

Once the depth of all nodes were determined, we can decide to map which node to the logic chain. In particular, we search for the longest chain of nodes in the DAG, which is a chain whose head node has the maximum depth. This chain is then mapped onto a logic chain in of FPGA; the head of the chain can be either a DAG output or a child of a node that is not chainable.

Figure 6.6 (a) shows a simple example. In this figure, there are two chaining candidates, each having a length of five; the chains intersect at node *N2*. As each node can be part of only one chain, *N2* is arbitrarily chosen for one of the chains, i.e., the left chain. As shown in Figure 6.6 (b), the second chain—the right one—is then broken into a chain of three nodes and a singleton node, which itself is not part of a chain. The pseudocode of the chaining heuristic is shown in Algorithm 1. The loop in the main function recursively traverses the DAG using the DFS method to compute the depth of each node in the DAG, starting from the DAG's output nodes. To compute the depth of each node, the maximum depth of the node's inputs is computed and is increased by one. The depth of the DAG's input nodes and non-chainable nodes is set to 0. Figure 6.7 illustrates an example, in which each node is marked by its depth. The shaded nodes are not chainable, as they have been previously assigned to other chains.

The depth information then allows the heuristic to identify the logic chains using the *SortChains* function in Algorithm 1. The node with the greatest depth in a chain is considered as the head of that chain. When a chain is selected for mapping, all the nodes in the chain are marked as *CHAINED*, to avoid placing a node in more than one chain. This process repeats until no chain with a length greater than a threshold value is found. The time complexity of the heuristic is

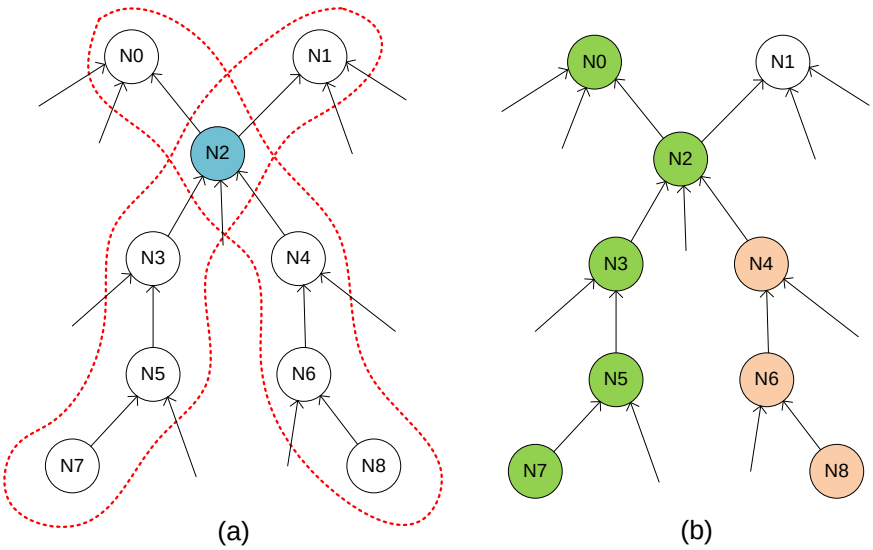


Figure 6.6: (a) Two chains intersecting at a shared node. (b) The shared node is assigned to one of the chains, breaking the other chain into two smaller sub-chains.

$O(nh)$, where n is the number of nodes in the DAG and h is the depth of the DAG.

6.4 Tool Chain Flow

Figure 6.8 presents the tool chain flow that we use for our experiments. For logic synthesis, we use Altera Quartus-II tool, which generates a Verilog Quartus Mapping (VQM) file netlist; Next, the VQM file is parsed and a DAG corresponding to the input circuit is created. This DAG is fed to the chaining heuristic. Once the chains are identified, a new atom-level netlist is generated, which represents the new mapped circuit. Lastly, the new netlist is fed back to Quartus-II for placement and routing, targeting the Altera Stratix-III FPGA—it will be shown later in this section that the ALM can be used to model our logic block with the logic chain. To have the right timing results, however, the timing report produced by Quartus-II is analyzed by our timing revision tool. The PowerPlay Early Power Estimator tool is employed to extract the power consumption. Details of the key steps are presented in the following subsections.

6.4.1 DAG Generator

Quartus-II synthesizes the benchmarks and maps them onto LUTs. Quartus-II's synthesizer generates the VQM file in ASCII text, which contains a node-level (or atom-level) netlist. Since our proposed logic block is a modified version of the Stratix-III ALM, we felt that Quartus-II was the most appropriate mapper to use. We also considered the possibility of using Berkeley's ABC synthesis tool [11]; however, ABC is a more general technology mapper and does not consider many ALM-specific features, such as fracturable LUTs.

Algorithm 1: FindLogicChains(pDAG)

```
while termination condition not met do
    for i = 0 to nDAGOutputs do
        FindNodeDepthRec(pDAG->out[i]);
    end
    SortChains( );
    MarkNodesInLongestChain( );
end
Function FindNodeDepthRec (pNode)
pNode->MaxDepth = 0;
for i = 0 to nLeaves - 1 do
    if pNode->Leaves[i] == DAGInput then
        pNode->Depth[i] = 0;
        break;
    else
        pTmpNode = pNode->Leaves[i];
    end
    depth = FindNodeDepthRec(pTmpNode);
    if pNode->Chainable then
        pNode->Depth[i] = depth + 1;
    else
        pNode->Depth[i] = 0;
    end
    if pNode->MaxDepth > depth + 1 then
        pNode->MaxDepth = depth + 1;
    end
end
return pNode->MaxDepth;
```

We implemented a VQM parser with the C++ programming language, which produces a DAG corresponding to the netlist. Each node in the DAG corresponds to an FPGA logic cell in the VQM netlist, and the edges between the DAG nodes represent data dependencies. Each DAG node is a C++ class object. Some of the class members are initialized when the VQM file is parsed; the other members are initialized by the chaining heuristic: the depth of a node; whether a node has been assigned to a chain; the id of the chain to which a node has been assigned; and the order of the node in the chain.

6.4.2 Placement and Routing

Once the circuits have been mapped to the new FPGA logic blocks using the logic chain, we need to place-and-route the mapped circuit to obtain accurate estimates of the area, critical path delay, and power consumption. Our area metric includes the number of logic blocks that are used and the amount of local and global routing resources required to realize the circuit.

We considered the possibility of modifying VPR [57, 13] as our experimental platform; however,

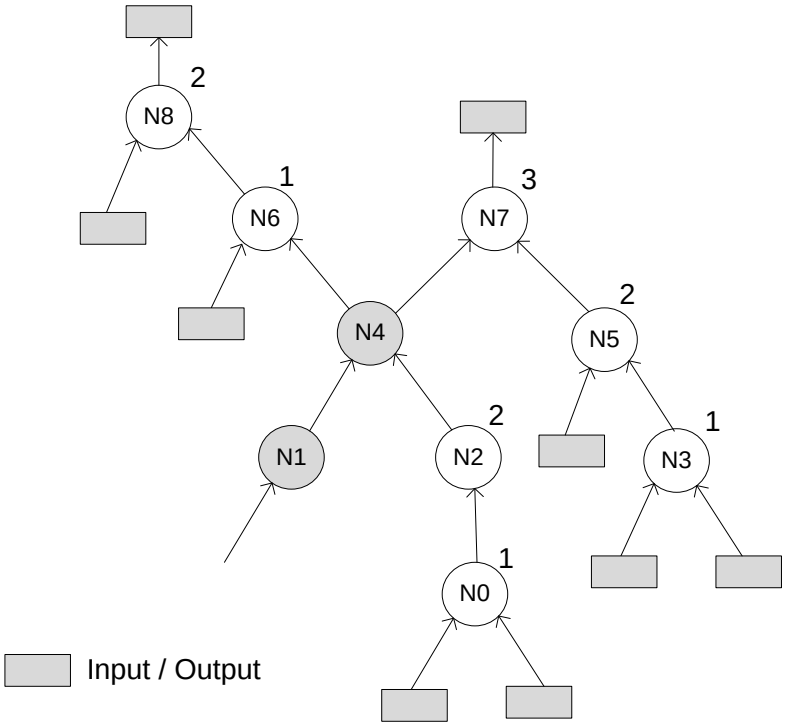


Figure 6.7: The depths of different nodes in a sample DAG. The shaded nodes are part of other chains and hence not chainable.

two problems caused us to look for other alternatives. Firstly, VPR's architectural model does not include carry chains and its packing, placement, and routing algorithms would be unable to handle their presence if they are supported architecturally; secondly, a comparison between VPR and Quartus-II is not meaningful, as these are two different frameworks; it is difficult to say whether a disparity in favor of either the baseline FPGA or our modified FPGA would be due to architectural superiority or differences between Quartus-II and VPR.

Fortunately, we discovered a way to let Quartus-II model our new FPGA logic block; this allowed us to use Quartus-II's placement and routing algorithms, rather than developing our own algorithms to better exploit the modified ALM architecture that we propose. The basic idea is to leave the ALM structure alone and to simply "pretend" that existing carry chains represent the logic chains that we want to introduce. This is possible because the carry output of the adder is a function of the carry input and four ALM inputs, which is similar in principle to our proposed logic chain in terms of connectivity. Therefore, we can assume that the output of the vertical 5-LUT in Figure 6.4 is the carry output of the adder; when we have a fanout that exceeds one, we can take the sum output of the adder as the fanout connection.

Consequently, it is necessary to configure the ALM in a way that the nodes on the logic chain are mapped to the carry chain in a corresponding fashion. To do so, we instantiate the ALM cells and provide the connections and configurations as explained previously; for the other nodes that are not on any chain, we write the original cell description that was obtained from

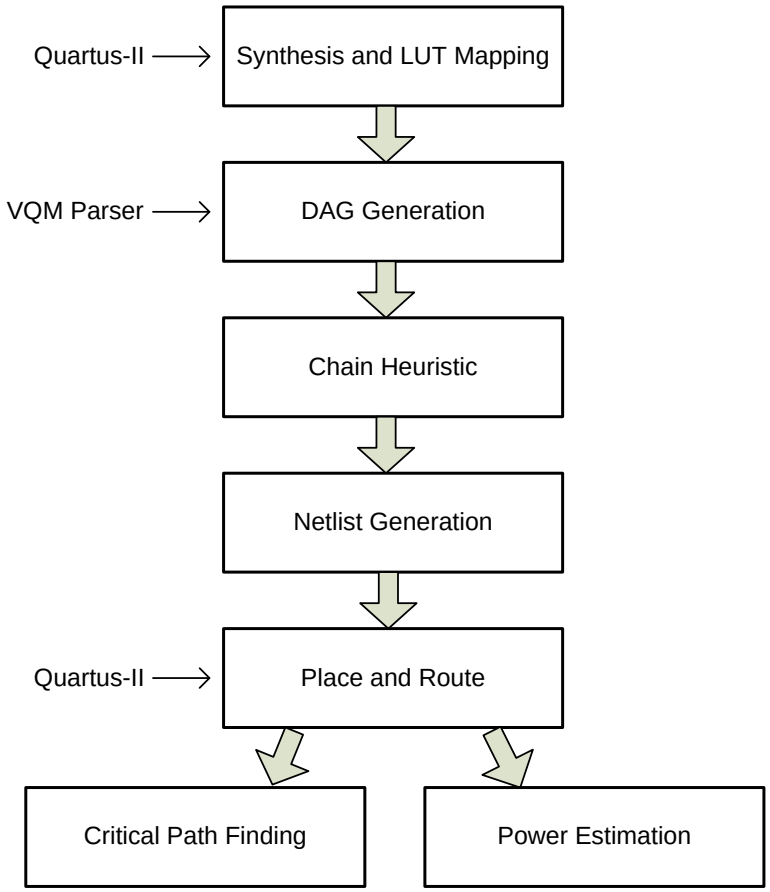


Figure 6.8: Tool chain flow used for the experiments.

synthesis back to the final netlist. The output, therefore, is a netlist of the ALM cells, similar to the original VQM file; nodes that use the logic chain are configured in arithmetic mode, while others are mapped using the ALM's normal mode. Quartus-II then proceeds to place and route the resulting netlist. This gives precise estimates of the usage of logic blocks and routing resources; however, some additional work is required in order to model the critical path delay accurately. To obtain the most dense implementation, we lock the logic in rectangular regions and shrink the region size to the extent that the tool is not able to fit the logic. This guarantees that we have the most packed implementation for both original netlist and the modified one.

6.4.3 Timing Analysis

As described in the preceding section, we effectively use the carry chains that are present in the Stratix-III FPGA to mimic the logic chains we have proposed for the purposes of placement and routing; however, the critical path delays that are obtained from Quartus-II are based on the delays of the full-adders on the carry chains, rather than the multiplexers that we introduced in the logic chain. Therefore, the delay of each adder in the carry chain should be replaced with the delay of the multiplexer instead. Our experiments revealed that the delay

Benchmark	3-LUT	4-LUT	5-LUT	6-LUT	7-LUT
alu4	163	89	413	32	6
pdc	431	214	538	139	5
misex3	179	99	376	43	1
ex1010	166	148	336	419	1
ex5p	149	89	234	60	0
des	134	89	159	110	0
apex2	206	101	284	108	2
apex4	127	74	354	119	1
spla	258	229	719	114	2
seq	205	150	356	96	2
Average	201	128	376	124	2

Table 6.1: Distribution of LUT sizes in different benchmarks.

between the ALM inputs and the outputs of the adder are significantly greater than the delay of a 5-LUT in the same ALM. Consequently, we take the delay of the normal 5-LUT in the ALM to be the delay of the 5-LUT that is realized in the logic chain; as a result, this reduces the overall delay compared to the timing report produced by Quartus-II.

This analysis has to be performed on a path-by-path basis. The critical path, as identified by Quartus-II's timing report, may no longer be critical once the delays of the logic chain have been properly accounted for. To solve this problem, we repeatedly adjust the delays of subsequent critical paths until we identify a path that includes no nodes mapped onto the logic chains. We wrote a script to perform this task for a specific number of critical paths; the delay adjustment stops when the first path that does not include an adder on the carry chain is found. The paths are then sorted based on their adjusted critical path delays, and the maximum is returned as the critical path delay of the circuit synthesized on a Stratix-III style FPGA that has been modified to include our proposed logic chain.

6.4.4 Power Estimation

To estimate the dynamic power consumption of the mapped circuits, we use *PowerPlay Early Power Estimator* [4] provided by *Altera*. This tool obtains the amount of resources used by each benchmark, the clock frequency after synthesis, the average fanout, the device type and the toggle rate of the wires and estimates the dynamic power consumption of the circuit. The power that is reported is broken down into routing power, logic block power and total power. Here, we assume that the dynamic power of the new logic block is approximately equal to the dynamic power of the standard ALM. One potential source of error could be the difference between the toggle rate of the adder output that we use for modeling and the toggle rate of the logic in our carry chain. To observe the difference, we modeled the real cell and the ALM in VHDL and applied several stimulus vectors to each and computed the average toggle rates.

Benchmark	Chainable	Chained	Max Chain	Avg Chain
alu4	94%	39%	12	5.2
pdc	89%	53%	9	5.8
misex3	93%	42%	9	5.1
ex1010	60%	47%	8	5.3
ex5p	88%	46%	7	5.2
des	77%	20%	4	3.1
apex2	84%	39%	8	4.9
apex4	82%	59%	8	4.3
spla	91%	46%	11	5.3
seq	88%	43%	6	4.9
Average	85%	44%	8.2	4.9

Table 6.2: Chaining heuristic statistics for different benchmarks.

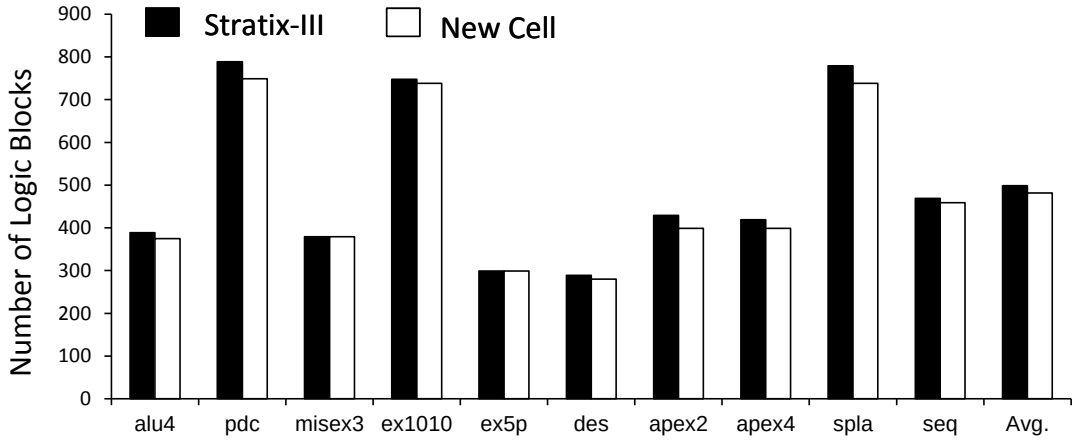


Figure 6.9: Number of logic blocks (ALMs) that are used in each method. On average, the introduction of our logic chain reduces the ALM usage by 4%.

Our results validated our assumption that toggle rates are approximately equal, on average.

6.5 Experimental Results

We evaluate the modified ALM-style logic block with the new logic chain; we consider several factors, including critical path delay, the ALM usage, routing resource usage and dynamic power consumption. We used the MCNC benchmarks for our experiments and only selected the combinatorial benchmarks; to synthesize the sequential circuit benchmarks, it is necessary to separate the combinational cones of logic that are placed between the registers and apply the chaining heuristic to each cone.

The Stratix-III ALM can be configured with logic functions having up to 7-inputs; any 6-input

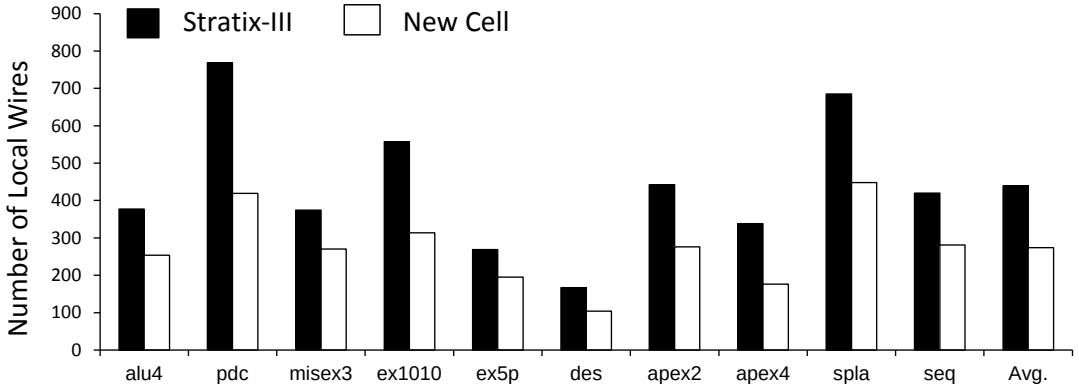


Figure 6.10: The number of local interconnection wires—i.e., within a LAB—used for each benchmark. On average, the introduction of the logic chain reduces the number of local wires used by 37%.

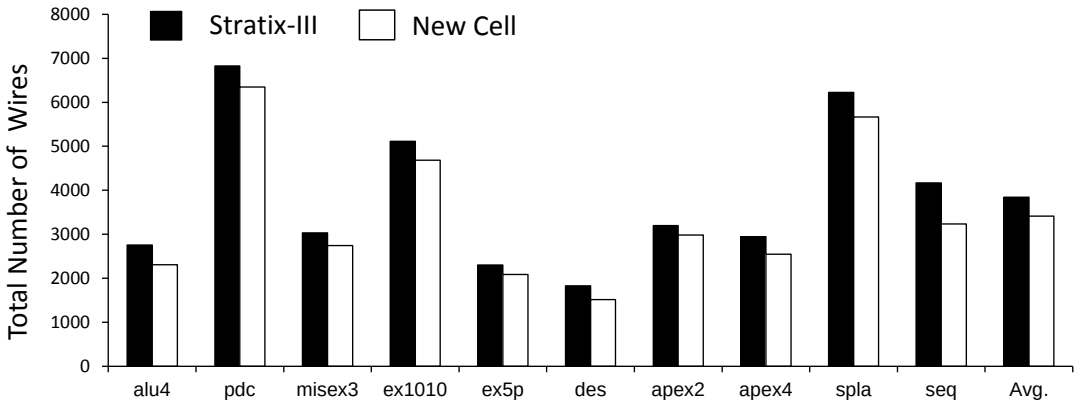


Figure 6.11: The number of global and local interconnection wires used for each benchmark, scaled by the length of the wires. On average, the introduction of the logic chain reduces the total number of wires used by 12%.

logic function can be mapped onto the ALM, along with a subset of 7-input logic functions. Table 6.1 reports the distribution of functions in terms of the number of inputs for the different benchmarks. The majority of the functions have five or fewer inputs—on average, 85% of functions; we have selected logic functions having at most five inputs for mapping onto the chains, as 5-LUTs are formed along the logic chain in the proposed logic block–see Figure 6.4.

Table 6.2 summarizes the chaining heuristic. The column labeled as *Chainable* indicates the percentage of the synthesized functions that are eligible to be mapped to the logic chain—functions that have five or less inputs; on average, 85% of logic functions are chainable. The column labeled *Chained* reports the percentage of eligible functions that are placed onto the logic chain, which is 44%, on average; we set the minimum chain length to 4 for all benchmarks, except for *apex4* and *sec*, where we allowed minimum chain length of three. The chain length is the number of the 5-LUTs that are in the chain. The last two columns of the table report

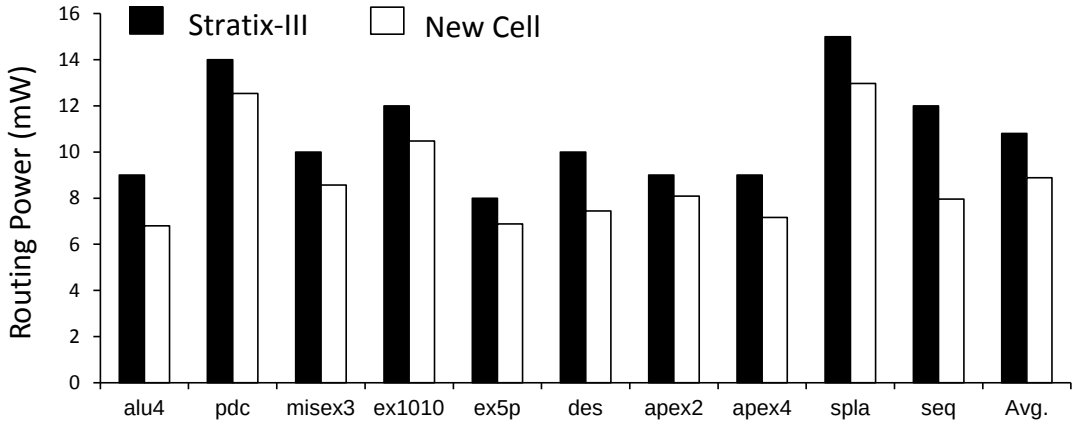


Figure 6.12: Dynamic power consumption estimates for the routing network; as the logic chain reduces the number of programmable wires used, an average savings of 18% is obtained.

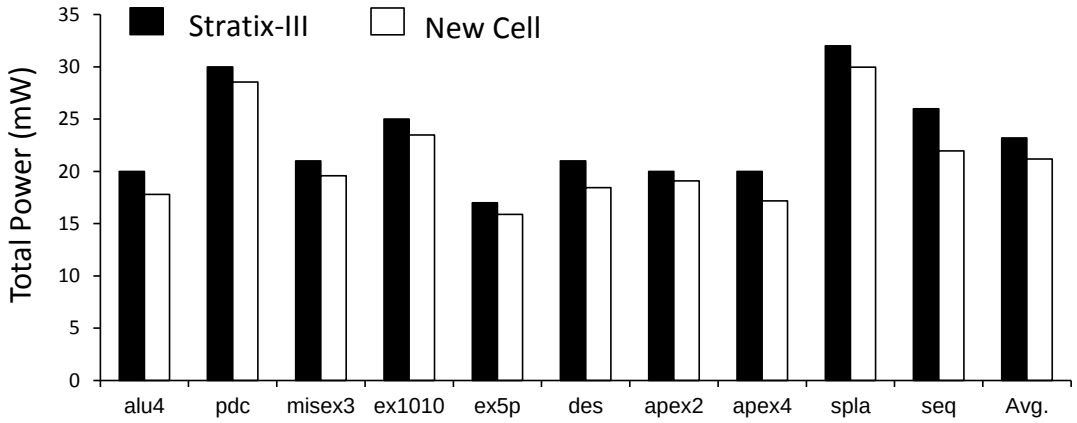


Figure 6.13: Total (logic plus routing network) power consumption estimates; the logic chain reduces total power consumption by 10%, on average.

the maximum and average chain lengths for each benchmark. The longest chain among the benchmarks is for *alu4*, which has 12 chained 5-LUTs.

Quartus-II is used to place-and-route each circuit. To evaluate the new logic block, we compare against the Stratix-III FPGA as a baseline; as described earlier, we take a netlist that has been mapped onto Stratix-III, identify logic chains and re-map them onto our new logic block that includes vertical logic chains.

Figure 6.9 reports the number of logic blocks that are used; our chaining heuristic is able to reduce the number of logic blocks used for all benchmarks other than *ex5p* and *misex3*; on average, our approach uses 4% fewer logic blocks than Stratix-III. It is essential to note that the Stratix-III ALM is used most effectively when it is configured to implement two 5-input logic functions with shared inputs, as shown in Figure 6.3 (a); in such cases, which are actually

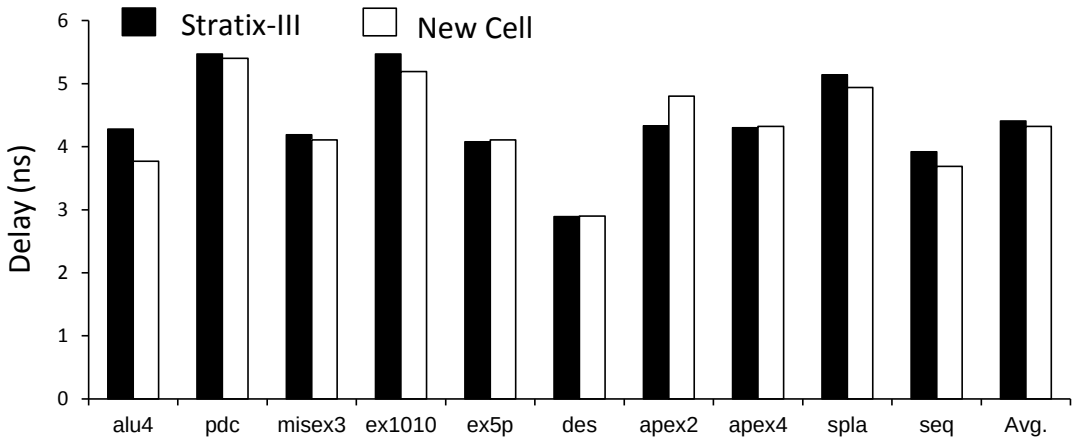


Figure 6.14: Critical path delay of each benchmark; the introduction of the logic chain marginally improves the critical path delay of most benchmarks.

quite common, the introduction of our logic block with the chaining heuristic cannot offer a significant improvement in terms of logic density.

The real benefit of using our logic block is its ability to reduce the usage of routing resources, as reported in Figures 6.10 and 6.11. On average, our logic block and chaining heuristic reduces the usage of local wires by 37%, with a maximum saving of 45%. On average, we reduce the usage of global and local wires by 12%, with the local wires contributing a reduction in 3%. The minimum saving on total wiring is 7% (*apex2*), and the maximum is 22% (*seq*). To account for different horizontal and vertical wires with different lengths, we have scaled the wires based on their length and, we reported their sum in Figure 6.11.

Reducing the usage of interconnect noticeably improves dynamic power consumption. A large fraction of dynamic power is consumed in the routing network; therefore, replacing a net that is routed on programmable interconnect with a direct connection using the logic chain can help to reduce dynamic power consumption. Figure 6.12 compares estimates of the dynamic power consumption of Stratix-III to our proposed FPGA that includes logic chains. On average, we reduce dynamic power consumption in the routing network by 18%. The most dramatic improvement is observed for *seq*, which had the greatest savings in total interconnect as reported in Figure 6.11.

Figure 6.13 reports the dynamic power consumption for each benchmark, which includes power consumption of logic resources; on average, the introduction of the logic chain reduces dynamic power consumption by 10%. It is important to note that the power estimation methodology used for this work is far from precise; however, our circuits are too large to use a much more accurate methodology such as SPICE simulation. Although we do not trust the exact numbers reported in Figures 6.12 and 6.13, we consider them a clear indication of the obtainable savings.

Lastly, we measure the logic chain's impact on critical path delay–see Figure 6.14; an improvement is observed for all benchmarks other than *apex2* and *apex4*. The overall improvement in delay is minimal; however, the logic chain was introduced primarily to reduce interconnect and logic block usage, not to improve delay. We do believe that there is potential to further improve the critical path delay, which would require much more aggressive synthesis algorithms that are specific to the new logic chain. In particular, this would require a new logic decomposition algorithm that recognizes the cascaded structure of the logic chain; such an algorithm could be integrated with a technology mapper to make better use of the logic chains than the relatively naive and greedy chaining heuristic described here; this is an important research direction that is currently left open for future work.

6.6 Related Work

Different and restricted types of LUT chains exist in some FPGA devices from both Altera and Xilinx families. Logic blocks in the Stratix [5] and Cyclone [3] FPGA devices from Altera, have a local connection, which connects the LUT output of one logic block to the input of the adjacent logic block. These connections allow LUTs within the same LAB to cascade together for wide input functions. Conceptually our proposed logic chain is similar to the mentioned local chains, but there are some fundamental differences. The main difference is that in contrast to the above FPGAs, we do not use the available input bandwidth of the logic block to connect the output of the adjacent logic block. This will increase the available bandwidth, and hence wider functions can be implemented without any need to change the logic block interface. The other difference is that the logic block in current FPGA devices has a *fracturable* LUT structure, which allows to use the available LUT resources in a logic block to implement larger functions considering our logic chain as the extra input.

The Xilinx FPGAs also have local connections between the adjacent logic blocks, which goes through a number of multiplexers in each logic block [92]. This local connection is mainly used for implementing carry look-ahead adders, but it can also be exploited for mapping of a limited number of generic functions. In its most general case, it can be used to implement the AND cascade of functions. For instance, a wide input AND function can be partitioned into some parts that are mapped to the LUTs and cascaded through the local connection. In contrast to such FPGAs, our logic chain is more general. The proposed logic chain goes through an LUT and forms the last input of the LUT; therefore, no logic constraint exists for cascading different functions.

Constructing larger LUTs by cascading smaller ones is also possible in the Virtex-5 Xilinx FPGAs. There are some multiplexers in the Virtex-5 logic block for this purpose, and by using such multiplexers, we can build up to 8-input LUTs. However, the routing wires are required to connect smaller LUTs for building larger ones. Meanwhile, there is a concern about the feasibility and usefulness of synthesizing a circuit onto such large LUTs. Prior research [1] indicates that an LUT size of four to six provides the best area-delay product for an FPGA.

In [19], an FPGA chip was developed with the logic blocks that are comprised of cascaded LUTs. In this work, each logic block has three 4-input LUTs hard-wired together for high performance. In contrast to our design, the hard-wired connection is exclusively limited to the logic blocks internal and does not cross the logic blocks boundaries.

Other relevant ideas consist of introducing carry chains into modern high-performance FPGAs and developing advanced technology mapping algorithms that attempt to exploit carry chains.

The vast majority of carry chains that have been proposed are for different types of adders [17, 31, 35, 47, 54]; the carry chains on commercial FPGAs available from Xilinx and Altera also fall into this category.

Similar to our work, one recent paper has presented a non-arithmetic carry chain in which two 2-LUTs are combined to form a 3-LUT [32]; however, it was based on a carry-select structure used in Altera's Stratix, which has since been deprecated. Starting with Stratix II, Altera's carry chains have employed a ripple-carry structure.

The ChainMap algorithm attempts to map arbitrary logic functions onto the carry chain of the Altera Stratix and Cyclone FPGAs [33]; as mentioned above, this carry chain has been deprecated and the authors readily admit that their algorithm is not applicable to newer Altera FPGAs or Xilinx FPGAs. Our chaining heuristic does share some principle similarities with ChainMap, but targets the logic chain that we have proposed rather than carry chains.

Traditional formulations of the technology mapping problem focus on converting a structural HDL implementation of a circuit into a network of K-cuts, where each K-cut can be mapped onto a single K-LUT. These formulations assume that the programmable routing network is used to connect the LUTs; it does not attempt to use carry chains, fracturable LUTs, embedded multipliers, or DSP blocks; likewise, these formulations could not account for the fixed wiring structure in the logic chain proposed here. Cong and Ding proved that minimizing the number of LUTs on the longest path can be done in polynomial time [21]; several others have proven that the decision problems corresponding to minimizing the total number of LUTs used in the covering and minimizing power consumption are NP-complete [29, 30]. Many heuristics to solve different variations of the technology mapping problem have been presented over the years; there are far too many to enumerate here.

Additionally, several papers have tried to perform logical decompositions to optimize the structural circuit description in conjunction with technology mapping [16, 25]; as logical optimization is NP-complete in the general case, this formulation of the problem is NP-complete as well, although the use of decomposition can significantly improve the quality of the technology mapping that can be achieved. In principle, this type of decomposition and technology mapping would be appropriate for the proposed logic chains proposed; the decomposition could exploit the specific fixed interconnect structure between adjacent LUTs on the logic chain; this approach is likely to be more effective than what we have done here: searching for chainable candidates in a technology mapping solution that was produced by a

more general technology mapping algorithm that was unaware of the presence of the logic chains.

6.7 Conclusion

This chapter has introduced the concept of logic chains as a way to improve routing resource utilization in modern high-performance FPGAs. The dedicated wires between logic blocks in the chain reduce pressure on the routing network. The key idea is based on the observation that many technology mapped circuits contain linear chains of LUTs after mapping; the basic idea is to add a direct connection between LUTs in a cluster that provide a natural mapping target for these chains. Having the fracturable structure of LUTs in modern FPGAs, we form larger LUTs by adding a multiplexer, which is controlled by the direct output of a preceding LUT along the chain. This enables us to increase the input bandwidth and logic density of the logic blocks without adding any additional inputs. Our experimental results have shown that the proposed logic block with logic chains can reduce the total number of routing resources required by 12%. It is estimated that this reduction saves 10% of the total dynamic power consumption. Moreover, the number of logic blocks used is reduced by 4% and the critical path delay is improved marginally, as well.

Having the logic chains, the soft-logic implementation of the circuits can be improved, as the logic density of the FPGA logic blocks is increased, and less routing wires are engaged in the implementation. Moreover, the routing network of FPGAs can be revised and made lighter as logic chains can partially replace the routing wires. This can considerably enhance the soft-logic implementation of circuits on FPGAs, which is the second goal of this thesis according to the thesis roadmap. In the next chapter of the thesis, as a complementary contribution on improving the soft-logic of FPGAs, we introduce an efficient soft-logic block for FPGAs, which does not include LUTs, as opposed to current FPGAs.

7 AND-Inverter Cones

Historically, soft-logic of FPGAs is mainly comprised of Look-Up Tables (LUTs). LUTs are generic structures with an excess flexibility, which allows one to implement any possible circuit on FPGAs. This flexibility, however, comes at a price: LUTs are costly circuits and it is not affordable to build FPGAs with large LUTs—rarely LUTs with more than 6 inputs have been used. In other words, increasing the number of LUT inputs to cover larger parts of a circuit has an exponential cost in the LUT complexity.

On the other hand, normally a small subset of functions that are supported by an LUT are used to implement each application. Following the second direction of the thesis *roadmap* that was presented in Chapter 1, we aim to reduce the excess flexibility of FPGAs soft-logic by limiting the number of functions that can be implemented by the soft-logic block of FPGAs. This, indeed, will allow us to design logic blocks that are much simpler than LUTs and can provide a better compromise between hardware complexity, flexibility, delay, and input and output counts.

In this chapter, inspired by recent trends in synthesis and verification, we explore blocks based on *And-Inverter Graphs (AIGs)*: they have a complexity which is only linear in the number of inputs, they sport the potential for multiple independent outputs, and the delay is only logarithmic in the number of inputs. Of course, these new blocks are extremely less flexible than LUTs; yet, we show (i) that effective mapping algorithms exist, (ii) that, due to their simplicity, poor utilization is less of an issue than with LUTs, and (iii) that a few LUTs can still be used in extreme unfortunate cases. We show first results indicating that this new logic block *combined* to some LUTs in hybrid FPGAs can reduce delay up to 22–32% and area by some 16% on average.

7.1 Introduction

Since their commercial introduction in the '80s, FPGAs have been essentially based on LUTs. K-input LUTs have one great virtue: they are generic blocks that can implement any logic

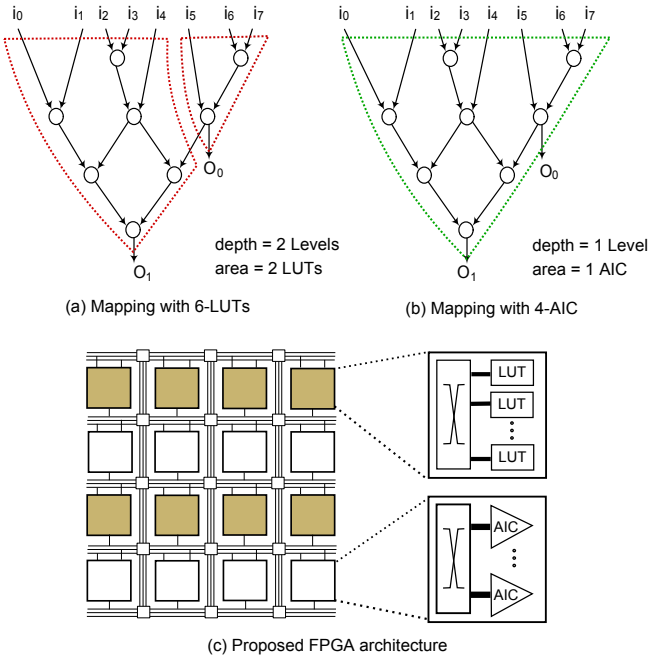


Figure 7.1: Flexibility, bandwidth, cost, and delay. (a)–(b) *And-Inverter Cones (AICs)* can map circuits more efficiently than LUTs, because AICs are multi-output blocks and cover more logic depth due to their higher input bandwidth. (c) A possible integration of AIC clusters in an FPGA architecture.

function of K inputs, and this makes it relatively easy to perform at least some elementary technology mapping: crudely, the problem of mapping reduces to cover the circuit with K-input subgraphs, irrespective of the function they represent. This flexibility, and the consequent advantages, do not come for free: LUTs tend to be large (roughly, their area grows exponentially with the number of inputs) and somehow slow (equally roughly, the delay grows linearly with the number of inputs). Also, the number of outputs is intrinsically one and internal fan-out in the subgraphs used for covering is not really possible. Figure 7.1(a) suggests graphically how the small number of inputs and the absence of intermediate outputs limit the usefulness of LUTs.

This seems to suggest that perhaps it would be wise to look for less versatile but more efficient logic blocks. In fact, researchers have at times looked into alternate blocks ever since FPGAs have attracted growing research and commercial interest. Yet, naturally, these alternate structures have been somehow related to the logic synthesis capabilities of the time, and thus have almost universally addressed programmable AND/OR configurations in the form of small *Programmable Array Logics (PALs)* (e.g., [48, 46, 23]). Traditionally, synthesis has been built on the sum of products representation and on algebraic transformations, but new paradigms have emerged in recent years. The one we are interested in is based on *And-Inverter Graphs (AIGs)* as implemented in the well-known academic synthesis and verification framework ABC [61]. This representation, in which all nodes are 2-input AND gates with an optional

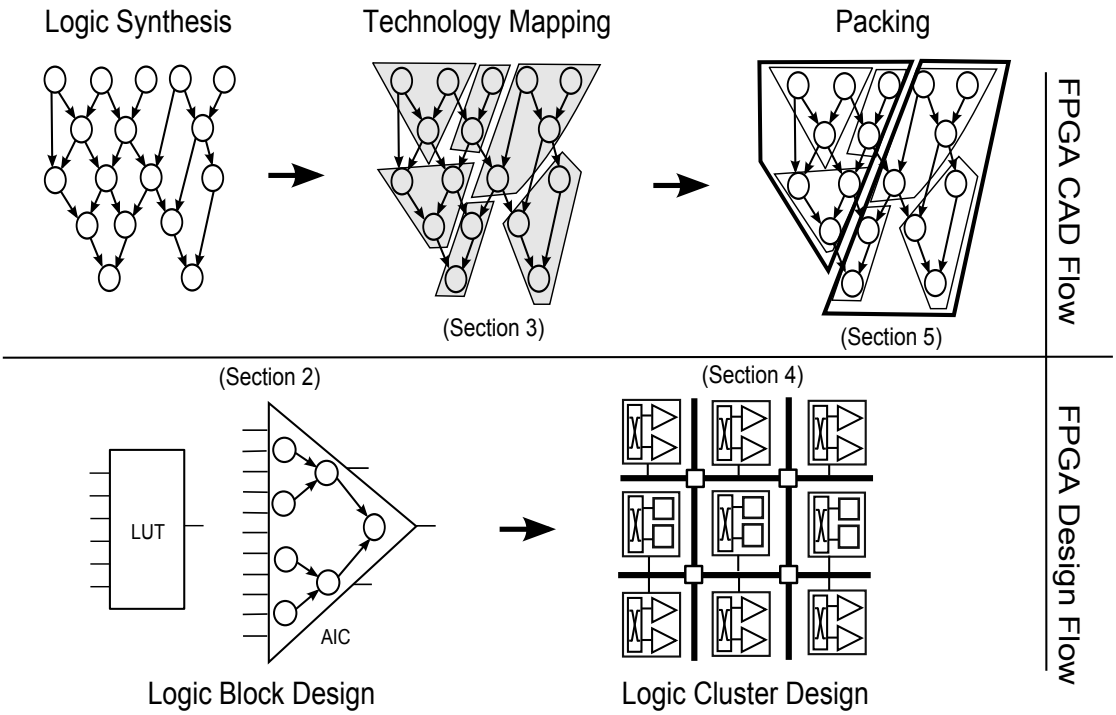


Figure 7.2: The paths to design and use a novel FPGA with AICs. In this chapter, we alternate between adapting the traditional CAD flow to our new needs and using the results to fix our architecture. To each of the last four steps is devoted one of the sections of this chapter, as indicated.

inversion at the output, is not new [37], but has received interest in recent years due to some fortunate combination when used with, for instance, Boolean satisfiability (SAT) solvers. Once a circuit is written and optimized in the form of an AIG, one can find many AIG subgraphs of various depth rooted at different nodes in the circuit.

Thus, we introduce a new logic block that we call *And-Inverter Cone (AIC)*. An AIC (which is explained in detail in Figure 7.3) is essentially the simplest reconfigurable circuit where arbitrary AIGs can be naturally mapped: it is a binary tree composed of AND gates with a programmable conditional inversion and a number of intermediary outputs. Compared to LUTs, AICs can be richer in terms of input and output bandwidth, because their area grows only linearly with the number of inputs. Also their delay grows only logarithmically with the input count and intermediate outputs are easier to implement. This makes it possible for AICs to cover AIG nodes more efficiently, as suggested in Figure 7.1(a)-(b). In this chapter, we will explore the value of AICs both as the sole components of new FPGAs as well as logic blocks for some hybrid FPGA made of both LUTs and AICs, as illustrated in Figure 7.1(c). Our results suggest that some hybrid solutions look particularly promising.

The rest of the chapter adapts the traditional CAD flow used on conventional FPGAs to the needs of AICs and, simultaneously, uses some of the partial results to fix the structure of our

novel FPGA. Figure 7.2 suggests this graphically: Section 7.2 addresses the design of the AIC to suit the abilities of modern AIG synthesis. Section 7.3 adapts traditional technology mapping to the new block. Section 7.4 looks at how to combine logic blocks in larger clusters with local routing, and Section 7.5 discusses the packing problem to complete the flow. Sections 7.6 and 7.7 then report our experimental results. We discuss related work in Section 7.8 and then wrap up with some conclusive remarks.

7.2 Logic Block Design

A new logic block is proposed in this section. This attempts to reduce the degree of generality provided by typical LUTs in order to obtain faster mappings. Unlike LUTs, our logic block is not able to implement all possible functions of its inputs. In the following, the choice of logic block is motivated and its architecture is discussed.

7.2.1 An AIG-inspired logic block

An *And-Inverter Graph (AIG)* is a *Directed Acyclic Graph (DAG)*, in which the logic nodes are two-input AND gates and the edges can be complemented to represent inverters at the node outputs. AIGs have been proven to be advantageous for combinational logic synthesis and optimization [61]. This graph representation format is also used for technology mapping step in both FPGA and ASIC designs [11].

Interestingly, AIGs include various cone-like subgraphs rooted at each node with different depths. Usually, the subgraphs with lower depths are more symmetric and resemble full binary trees. The frequent occurrence of such conic subgraphs serves as motivation of this work, where we propose a new logic block that can map cones with different depths more efficiently than LUTs. The basic idea is to have a symmetric and conic block with depth D, which maps arbitrary AIG subgraphs with depth $\leq D$. This logic block is called *And-Inverter Cone (AIC)*.

To illustrate the potential benefits of AICs with respect to LUTs, we refer to Figure 7.1, where two levels of LUTs are required to map the same functionality that can be mapped onto a single AIC. The reason for this is twofold: on the one hand, the LUT size is limited to six inputs and the entire AIG (eight inputs) cannot fit into just one 6-LUT. On the other hand, even if the size of the LUT was big enough, the mapping would still use two LUTs, as the AIG has two distinct outputs. It is worth mentioning that increasing the LUT size to accommodate more inputs would result in a huge area overhead. Instead, the proposed AIC inherently offers smaller area and propagation delay than an LUT for the same number of inputs. For example, a 4-AIC with 16 inputs requires half the area of a 6-input LUT—using the area model of Section 7.6.1 with less delay. Clearly, the fact that more wires need to be connected to the AICs creates new routing congestion issues. However, as detailed in Section 7.4, these can be alleviated by packing several AICs in a limited bandwidth AIC cluster with local interconnect.

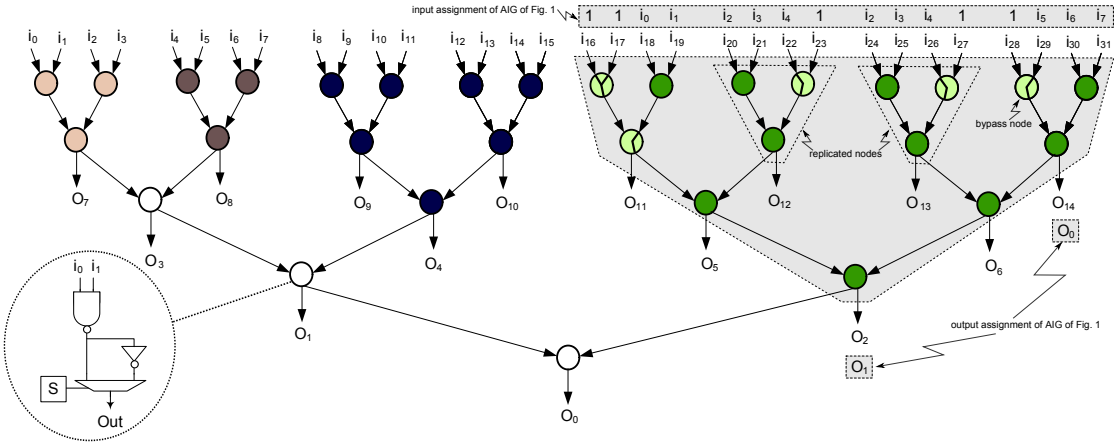


Figure 7.3: Architecture of 5-AIC (AND-Inverter Cone), which has five levels of cells that are programmable to either AND or NAND gates. The 5-AIC can also be configured to 2-, 3-, and 4-AICs in many ways (highlighted cells show one possibility), without any need for extra hardware. The AIG of Figure 7.1 is mapped onto the right-hand side. To propagate a signal, we can configure a cell to the bypass mode (e.g., forcing one input to 1 when this is operated as an AND). Moreover, some AIG nodes need to be replicated when the fanout of an internal value is larger than one.

7.2.2 AND-Inverter Cone (AIC) Architecture

Figure 7.3 shows the architecture of an *And-Inverter Cone (AIC)*, which has five levels of cells. Each cell can be configured as either a two-input NAND or AND gate. Notice that each cell has an AIC output, except for the cells belonging to the lowest level of the AIC. This provides access to intermediate nodes as in the example of Figure 7.1. Moreover, these outputs enable a bigger AICto be configured as multiple smaller ones. For example, the AIC of Figure 7.3, implements the AIG of Figure 7.1 at the right-hand side while the left-hand side can be used to implement other functions with various combinations of 2-, 3-, and 4-AICs. Accordingly, a 5-AIC contains two 4-AICs, four 3-AICs, or eight 2-AICs.

Generalizing, each D-AIC has $2^D - 1$ cells, 2^D inputs and $2^{D-1} - 1$ outputs. In the rest of the chapter, we consider D-AICs with depths from three to six, and we will study the effect of the allowed AIC depth on the mapping solution. Depths greater than six are not considered, as they require a huge input bandwidth, which may result in major modifications of the global routing network of current FPGAs. Table 7.1 compares different D-AICs with the conventional 6-LUT in terms of IO bandwidth, number of configuration bits and multiplexers.

7.3 Technology Mapping

During technology mapping, the nodes comprising the AIG are clustered into subgraphs that can be mapped onto an AIC or an LUT. This can be done in multiple ways depending on the optimization objectives including delay and area.

Block	inputs	outputs	2:1 mux	config bits
2-AIC	4	1	3	3
3-AIC	8	3	7	7
4-AIC	16	7	15	15
5-AIC	32	15	31	31
6-AIC	64	31	63	63
6-LUT	6	1	64	64

Table 7.1: AICs have less configuration bits than LUTs, while they can implement circuits with a much greater number of inputs (e.g., a 6-AIC includes eight times more inputs than a typical 6-LUT).

In this work, the primary optimization objective of technology mapping is delay minimization and consequently a mapping solution is said to be *optimal* if the mapping delay is minimum. Area reduction is also considered but just as a secondary optimization objective. Technology mapping for AICs is similar to the typical LUT technology mapping but adapted to the peculiarities of AICs, such as the fact that multiple outputs are possible. In the rest of the section, the mapping problem is first formalized and then the main four steps of the mapping algorithm are described in detail.

7.3.1 Definitions and Problem Formulation

A technology independent synthesized netlist (AIG format) is input to our mapping heuristic. Such netlist is automatically produced by ABC [61]. We take the input netlist and extract the combinational parts of the circuit and represent them by a DAG $G = (V(G), E(G))$. A node $v \in V(G)$ can represent an AND gate, a primary input (PI), a pseudo input (PSI, output of a flipflop), a primary output (PO), or a pseudo output (PSO, input of a flipflop). A directed edge $e \in E(G)$ represents an interconnection wire in the input netlist. The edge can have the *complemented* attribute to represent the inversion of the signal.

At a node v, the depth $depth(v)$ denotes the length of the longest path from any of the PIs or PSIs to v. The height $height(v)$ denotes the length of the longest path from v to any of the POs or PSOs. Accordingly, the depth of a PI or PSI node and the height of a PO or PSO node are zero.

The mapping algorithm that we use in this work is a modified version of the classical depth-optimal LUT mapping algorithm [20]. It is well known that the problem of minimizing the depth can be solved optimally in polynomial time using dynamic programming [20, 49]. However, we also target area-minimization as a secondary objective, which is known to be NP-hard for LUTs of size three and greater [22, 55]. We use *area flow* heuristic [58] for area approximation during the mapping.

The mapping of a graph in LUTs requires different considerations. For a node v, there exist

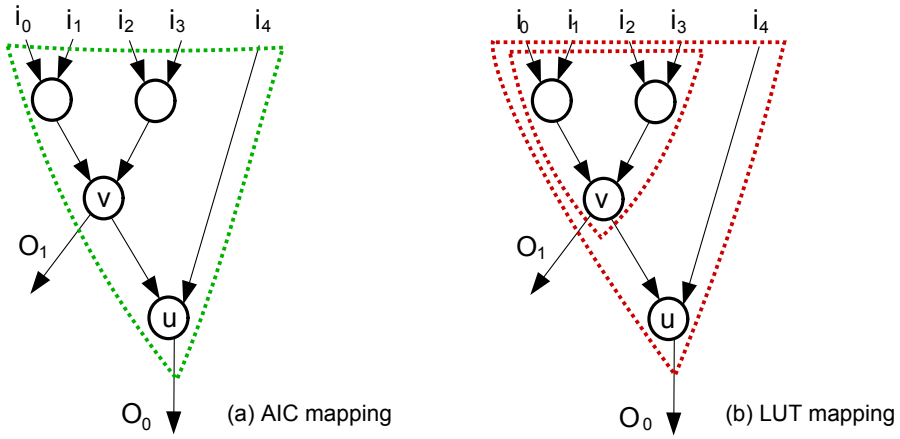


Figure 7.4: Difference between LUT and AIC mapping. Since AICs are inherently multi-output blocks, the same cone rooted at u in (a) can also be a (free) mapping cone of v, while in LUT mapping, no common cone exist for any two nodes (b).

several subgraphs containing v as the root, which are called *cones*. Accordingly, C_v is a cone that includes node v in its root and some or all of its predecessors. For mapping C_v by an LUT, it should be K-feasible, where $inputs(C_v) \leq K$. Moreover, the cone should be *fanout-free*, meaning that the only path out of C_v is through v. If the cone is not fanout free, then the node which provides the fanout may be duplicated and will be mapped by other LUT(s), as the primary minimization objective is depth.

The AICs mapping cone candidates of v are extracted differently. In this case, rather than being K-feasible, a cone C_v, to be mappable on a D-AIC block, should be depth feasible, where $depth(C_v) \leq D$. The other constraint is that the nodes at lowest depth of C_v, should not have any path to a node outside C_v, otherwise such nodes are removed from C_v. This condition ensures that C_v to be mappable to an AIC such as the one illustrated in Figure 7.3, in which no AIC output is driven by the nodes at the lowest level of the AIC.

When AICs are considered as the mapping target in addition to LUTs, the definition of the problem of mapping for depth does not change. The only difference is that the cone candidates of AICs are added to the cone candidates of LUTs for each node in the graph. Although the conditions of eligibility for LUTs and AICs are different, it is possible to have common cones between the two that are treated as separate candidates.

Next, the main steps of the mapping algorithm are described in detail.

7.3.2 Generating All Cones

To generate all K-feasible cones, we use the algorithm described in [24, 76], in which the cones of a node are computed by combining the cones of the input nodes in every possible way. This step of the mapping takes a significant portion of the total execution time, especially when K

is a large value such as six.

The cone generation for AICs is different from the cone generation for LUTs, as the cones of each node are produced independently from the cones of its input nodes. To generate all possible D-AIC mappable cones for a node v, the subgraphs rooted at v are examined by varying the cone depth from two to D. All possible subgraphs that meet the AIC mapping conditions described in section 7.3.1, are added to the cone set of node v. If a cone C_v satisfies the depth condition, but has a fanout node u at the lowest depth of the cone, u will be removed from C_v; if this still satisfies the depth condition, the cone will be added to the D-AIC mappable cone set.

The main difference between the cone generation for AICs and LUTs is having common cone candidates for different nodes, as shown in Figure 7.4. This is possible, as AICs are multi-output. In this figure, the cone that has u as its root, can be used to map both v and u. Therefore, this cone should be in the AIC cone sets of both nodes. We call this cone as a *free* cone for node v, as it maps v for free when it is selected for u mapping.

The time complexity of the D-AIC cone generation is $O(M \cdot D)$, where M is the number of nodes in the graph and D is the maximum depth of an AIC block.

7.3.3 Forward Traversal

Once the cones sets of both LUTs and AICS are computed for every node in the graph, the next step is to find the best cone of each node by traversing the graph in topological order. Since the primary objective in this work is to minimize the depth, the best cone of node v is the one that gives v the lowest depth. If there is more than one option, the cone which brings less area flow to v is selected (see [58] for further details). The depth and area flow of v, when mapped onto cone C_v, are dependent on the depth and area flow values of the C_v input nodes.

To compute the depth and area flow of node v, we use Equations 7.1 and 7.2, respectively. Since the FPGA blocks, including K-LUTs and D-AICs, are heterogeneous and have different depths, we should consider the interconnection wire delays for the depth computation of each node, similar to the edge-delay model [93]. Although we have both local (intra cluster) and global (inter cluster) routing wires, which have different delays, we assume that all wires have the same delay equal to the average delay of the local and global wires.

$$dp(v) = \max(dp(In(C_v)) + dp(C_v) + dp(wire)) \tag{7.1}$$

$$af(v) = \sum_{i=0}^{nIn(C_v)} (af(In_i(C_v)) + area(C_v) \tag{7.2}$$

In the above equations, $dp(C_v)$ and $area(C_v)$ are the depth and area of the logic block that C_v can be mapped on. This block can be either a K-LUT or a D-AIC. If C_v is a *free* cone of node v,

Algorithm 2: Find the best cone for each node of the DAG

```
BestC_v.dp = ∞;
BestC_v.af = ∞;
for i = 1 to nC_v(LUT) do
    v.setdp(C_v(i));
    v.setaf(C_v(i));
    cond_1 = C_v(i).dp < BestC_v.dp;
    cond_2 = C_v(i).dp = BestC_v.dp;
    cond_3 = C_v(i).af < BestC_v.af;
    if cond_1 || (cond_2 && cond_3) then
        BestC_v = C_v(i);
    end
end
for i = 1 to nC_v(AIC) do
    v.setdp(C_v(i));
    v.setaf(C_v(i));
    cond_1 = C_v(i).dp < BestC_v.dp;
    cond_2 = C_v(i).dp = BestC_v.dp;
    cond_3 = C_v(i).af < BestC_v.af;
    if cond_1 || (cond_2 && cond_3) then
        BestC_v = C_v(i);
    end
    cond_1 = C_v(i).dp < BestBackupC_v.dp;
    cond_2 = C_v(i).dp = BestBackupC_v.dp;
    cond_3 = C_v(i).af < BestBackupC_v.af;
    if C_v(i).root == v then
        if cond_1 || (cond_2 && cond_3) then
            BestBackupC_v = C_v(i);
        end
    end
end
```

then $dp(C_v)$ and $dp(In(C_v))$ will refer to the depth and inputs of the sub-AIC in C_v. And for area flow computation, the term $area(C_v)$ will be removed from Equation 7.2.

Algorithm 2 presents the pseudo-code of the algorithm used to find the best cone of each AIG node. This function iterates over all generated cones for both LUTs and AICs of node v to find the best cone that has the lowest depth. If two cones have the same depth, the one that requires smaller area is selected. If the best cone of node v is a *free* cone, this cone will be selected for the mapping, if and only if the root of the cone—which is not v—is visible in the final mapping solution and this cone is the best cone of the root node as well. If one of these two conditions does not hold, then we need to select another cone as the best cone for v. Therefore, it is essential to maintain a non-*free* best cone—v is the root of such a cone—for v as a backup best cone.

7.3.4 Backward Traversal

In this step, the graph is covered by the best cones of the visible nodes in the graph, which are added to the mapping solution set S. A node is called *visible*, if it is an output or input node of a selected cone in the final mapping. Initially POs and PSOs are the only visible nodes and S is empty. The graph traversal is performed in reverse topological order from POs and PSOs to PIs and PSIs. If the visited node v is visible, then its best cone, BC_v, is selected for the mapping and is the added to S. Then, all the input nodes of BC_v become visible and the graph traversal continues. If the BC_v is a *free* cone and it is already in S, there is no need to add it again and only the heights of the input nodes of v are updated. Otherwise, if the *free* cone is not in S, then the backup BC_v, which has v as its root, is selected for mapping and is added to S. During the backward traversal, the height of each visible node is updated. Once a BC_v is selected for mapping, the height of its input nodes are updated by adding the height of v to the depth of v within the target AIC or LUT.

7.3.5 Converting Cones to LUTs and AICs

The mapping solution S, which is generated during the Backward Traversal, includes all the cones that cover the graph. The next step is mapping the cones in S to either a K-LUT or a D-AIC. If the selected cone belongs to the K-feasible cone set of node v, then it should be implemented by an LUT. Otherwise, the cone is a D-AIC mappable cone, which is implemented by an AIC. The depth of the cone defines the type of the target AIC block.

7.4 Logic Cluster Design

The proposed AICs require a much higher IO bandwidth than typical LUTs. In order to alleviate the routing problem that may result from that increase, we propose to group multiple AICs into an AIC cluster with local interconnect.

To form an AIC cluster, we integrate N_{AIC} D-AICs, optional flipflops at the outputs of D-AICs to support sequential circuits, and an input and an output crossbar. The input crossbar drives the inputs of the AICs in the cluster, and the output crossbar drives the outputs of the cluster. Since we do not want to change the inter-cluster routing architecture of the FPGAs, we use the same bandwidth of LUT-based clusters for AIC clusters and keep the AIC cluster area close to the area of the reference LUT cluster, which is the *Logic Array Block (LAB)* in the *Altera Stratix-III*—refer to Chapter 2 for the LAB structure.

To study the effect of the AIC size on the mapping results, we select different D-AICs as the base logic block in a cluster, where D varies from three to six and can be configured to implement the AIC blocks that have $depth \leq D$. However, the number of the D-AIC blocks in the cluster, N_{AIC}, varies for different D values such that the number of sub-AICs in the cluster remains the same and no changes occur in the cluster crossbars.

The two crossbars in the AIC cluster are the main contributors to the cluster area. Crossbars are basically constructed with multiplexers and their area depends on their density and on the number of the crossbar inputs and outputs. Since both crossbars get the outputs of N_{AIC} D-AICs as the input, reducing the number of the D-AIC outputs will significantly reduce the area share of the crossbars. Originally, each D-AIC has $2^{D-1}-1$ outputs, but in our experiments, we observed that in the extreme case only 2^{D-2} outputs are utilized and that is when a D-AIC is configured to 2^{D-2} 2-AICs. Hence, a very simple sparse crossbar is added at the output of each D-AIC to reduce the number of D-AIC outputs to 2^{D-2}.

The second technique used to reduce the crossbar area is to decrease its connectivity and make it sparse. To trade-off the crossbar density and packing efficiency in the AIC cluster, we measured the packing efficiency of the clusters having an input crossbar with 50%, 75%, and 100% connectivities. The packing efficiency is the ratio of the number of AIC clusters, assuming that each AIC cluster has unlimited bandwidth and the actual number of AIC clusters that is obtained from packing. To calculate the number of clusters in the ideal packing, we use Equation 7.3. In this equation, nC_i is the number of cones with depth i. Figure 7.5 shows the results of this experiment for different base AIC blocks in the cluster. The reported efficiency is the average packing efficiency of the 20 biggest MCNC benchmarks.

$$nClusters_{ideal} = \sum_{i=2}^{6} \left(\frac{nC_i}{N_{AIC} \cdot 2^{6-i}}\right) \tag{7.3}$$

One observation from Figure 7.5 is that the packing efficiency is substantially reduced for all the three scenarios, when the allowed cone depth in the technology mapping is reduced. This is reasonable, as the probability of input sharing and open inputs is reduced for smaller cones. Moreover, when smaller AICs are packed to a D-AIC, a larger number of the D-AIC outputs are utilized, which increases the output bandwidth requirement. The second observation is that reducing the crossbar connectivity to 75% largely maintains the packing efficiency of the full crossbar. However, the packing efficiency for the crossbar with 50% connectivity decreases to a larger extent. Therefore, one option to reduce the crossbar area without having a sensible degradation in packing efficiency is to set the crossbar connectivity to 75%.

Exploiting the mentioned crossbar simplifications, and by using the area model of Section 7.6.1, the area of the AIC cluster remains close to the area of a LAB, when three 6-AICs, six 5-AICs, twelve 4-AICs, or twenty four 3-AICs are integrated in the AIC cluster. As mentioned, the input/output crossbars of the AIC cluster are fixed for all scenarios.

7.5 Packing Approach

In the previous section, we defined the architecture of the AIC cluster. Given the AIC and LUT clusters, the next step is to pack the technology mapped netlist onto the clusters. For the packing, we use the *AAPack* [57] tool, which is an architecture-aware packing tool developed for FPGAs. The input to *AAPack* is the technology mapped netlist with unpacked blocks, as

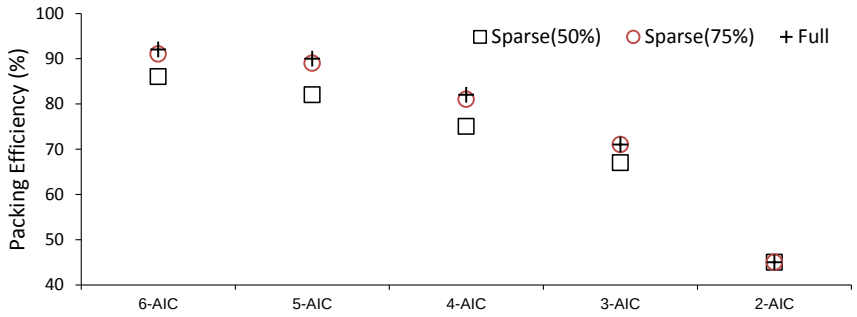


Figure 7.5: The packing efficiency of three crossbar connectivity scenarios: 50%, 75%, and 100%. The allowed cone depth in technology mapping is varied to study the effect of AIC size on the packing quality.

well as a description of an FPGA architecture. The output is a netlist of *packed* complex blocks that is functionally equivalent to the input netlist. Similarly, we also use *AAPack* to pack LUTs in LABs.

The packing algorithm uses an affinity metric to optimize the packing. This affinity metric defines the amount of net sharing between p, which is a packing candidate, and B, which is a partially filled complex block. In the architecture file, the complex block should be represented as an ordered tree. Nodes in the tree correspond to physical blocks or modes. The root of tree corresponds to an entire complex block and the leaf nodes correspond to the primitives within the complex block. For the D-AIC complex block, we construct a tree similar to the DSP block multiplier tree in the original work, by which we define different configuration modes of the D-AIC. The number of AICs in the cluster as well as the crossbars structure are also defined in the architecture file. The information is used by the packer to group the individual blocks in clusters. During the packing process, some routability checking are performed to ensure (local and global) routability of the packing solution, which considers the intra-block and the general FPGA interconnect resources.

7.6 Experimental Methodology

In this work, we use a classic area and delay model [13]: The area model is based on the transistor area in units of minimum-width transistor area; the rationale is that to a large extent the total area is determined by the transistors more than by the metal connections. For the delay model, circuits are modeled using SPICE simulations for 90nm CMOS process technology.

7.6.1 Area Model

The area modeling method requires a detailed transistor-level circuit design of all the circuitry in the FPGA [13]. Figure 7.6 shows an AIC cluster with three 6-AICs. Table 7.2 lists the area of

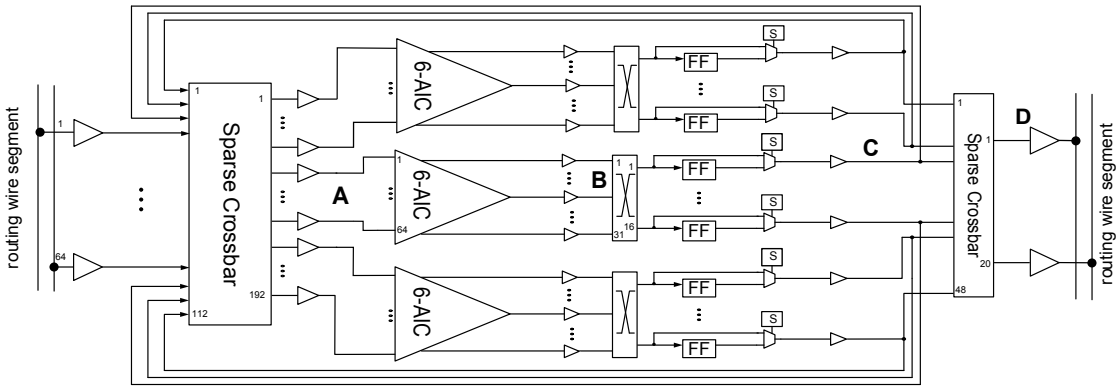


Figure 7.6: Structure and delay paths of an AIC cluster with three 6-AICs.

Component	**Area** (Tr_{minW})	**Quantity**	**Total**(Tr_{minW})
6-AIC block	1,512	3	
6-AIC output Xbar	217	3	
6-AIC FFs and muxes	1,104	3	
AIC cluster input Xbar	22,072	1	
AIC cluster out Xbar	2,660	1	
AIC cluster buffers	1,447	1	
AIC cluster with three 6-AICs			**34,678**
ALM	1,751	10	
LAB in Xbar	16,251	1	
LAB buffers	470	1	
LAB with ten ALMs			**34,231**

Table 7.2: Areas of different components in an AIC cluster and in a LAB, measured in units of minimum-width transistor area.

different components in the AIC cluster and in a LAB in terms of number of minimum-width transistors. *ALM* stands for *Adaptive Logic Module*, which is the logic block in *Altera Stratix-II* and in following series—refer to Chapter 2 for the ALM structure. Based on this table, the area of an AIC cluster with three 6-AICs and the crossbars mentioned in Section 7.4 is marginally larger than a LAB with 10 ALMs. As mentioned in Section 7.4, the AIC cluster has almost the same area when the basic AIC block is changed.

7.6.2 Delay Model

The circuit level design of the AIC cluster suggested in Figure 7.6 is also used for accurate modeling of the cluster delays. The crossbars in this figure are developed using multiplexers, and for these we adopted the two level hybrid multiplexer that is used in Stratix-II [56]. Hence, the critical path of each crossbar goes through two pass-gates, with buffers on the inputs and

Path	Description	Delay (ps)
A → B	6-AIC main output	496
B → C	crossbar and FF-Mux	75
C → D	output crossbar of cluster	50

Table 7.3: Delays of different of paths in the AIC cluster of Figure 7.6.

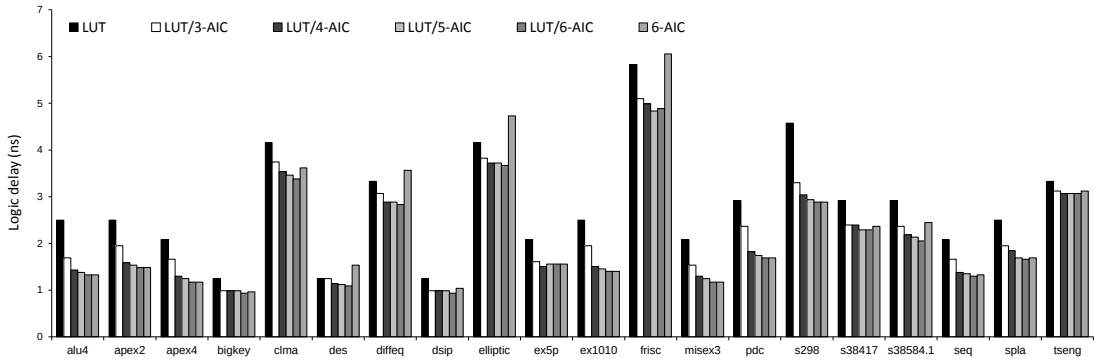


Figure 7.7: Logic delay of all benchmarks in the original FPGA (LUT), for the FPGA composed only of AIC (6-AIC), and for a hybrid FPGA (LUT/6-AIC).

outputs of the components that include pass transistors.

We performed SPICE simulations with 90nm 1.2v CMOS process, to determine the delay of all paths in the cluster shown in Figure 7.6. The results are listed in Table 7.3. For the path between B and C, the delay number relates to the path that goes through the main output of the 6-AIC, which has the longest path. These delay numbers are used in the technology mapping to minimize the delay of the mapped circuit.

We also measured the delay of a LAB by SPICE simulation. Simulation results revealed that the delay of a 6-LUT in an ALM, excluding the LAB input crossbar, in 90nm CMOS process, is between 280ps and 500ps, taking into account that different LUT inputs have different delays. We use the average delay (390ps) for our experiments. Based on [6], the 6-LUT delay in 90nm process technology has a delay between 162ps to 378ps and considering the extra multiplexers that exist on the LUT output path in the ALM structure, our delay numbers appear realistic.

7.7 Results

We contrast three architectures and various mapping strategies, using the MCNC benchmarks [94]. We consider the original FPGA, a homogeneous FPGA exclusively composed of AIC clusters, and a hybrid FPGA composed of both LUTs and AIC clusters as different experiment scenarios. In the hybrid structure, we also vary the base AIC block of the AIC-cluster from 3-AIC to 6-AIC.

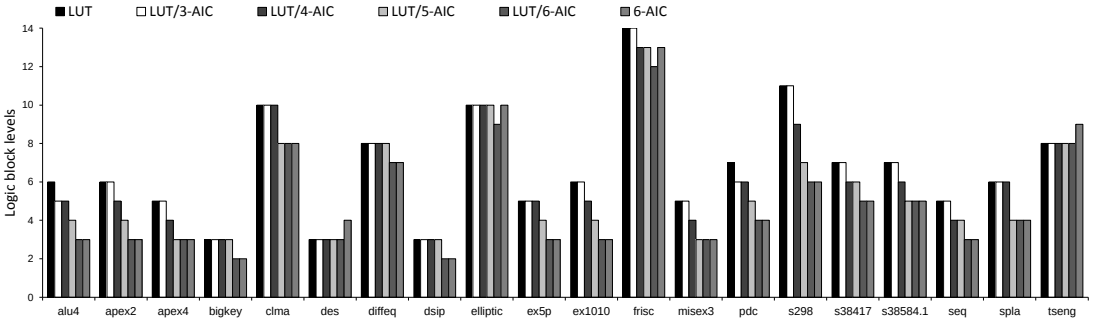


Figure 7.8: Number of logic blocks (both LUTs and AICs) on the critical path.

Mapping Scenario	Intra-cluster Wires
LUT	50%
6-AIC	34%
LUT/6-AIC	35%
LUT/5-AIC	37%
LUT/4-AIC	38%
LUT/3-AIC	40%

Table 7.4: Average ratio of intra cluster wires for the different mapping scenarios.

Figure 7.7 shows the logic delays of the benchmarks for the mentioned scenarios. The main observation is that the lowest logic delay relates to the hybrid structure, as we have both LUTs and AICs mapping options. Moreover, except for the *ex5p* and *frisc* benchmarks, the logic delay is always reduced when deeper cones are allowed, which appears predictable as a general trend. This is also visible in the number of logic-block levels on the critical path, either LUTs or AICs, as shown in Figure 7.8; the graph gives an indication of the routing wires necessary to connect the logic blocks of the circuits: although some logic delays are higher for deeper cones, their total delay can be still better due to the reduced number of wires between logic blocks. Comparing LUT-only and AIC-only implementations, we see that there are circuits that have better logic delay when LUTs are used, but on average AIC-only implementation has 28% less logic delay. Moreover, except for *tseng* and *des*, the number of logic blocks on the critical path (and thus routing wires) in the AIC-only implementation is less than or equal to that of the LUT-only one.

As the current release of VPR 6.0 does not support timing driven placement and routing, we set a fixed delay value for the interconnecting wires in order to estimate the total circuit delay. This delay number is different for the different mapping scenarios and its value is specified based on the delay and used ratio of intra and inter cluster wires for each mapping scenario that is reported in Table 7.4. Using this wire delay, we compute the routing delay of the critical path of the circuits, using the number of logic blocks in these paths. Figure 7.9 illustrates a rough estimation of the total average logic and routing delays of the circuits. On average, the

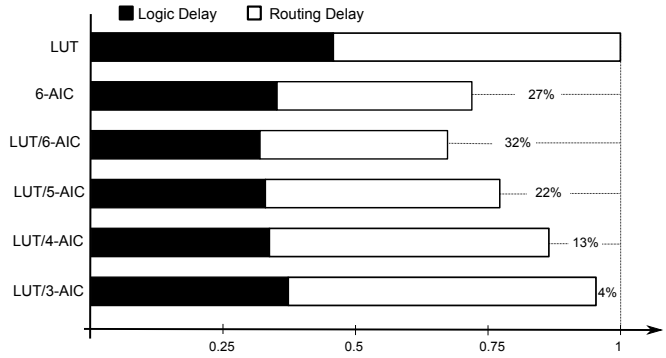


Figure 7.9: Geometric mean of normalized total logic and routing delays.

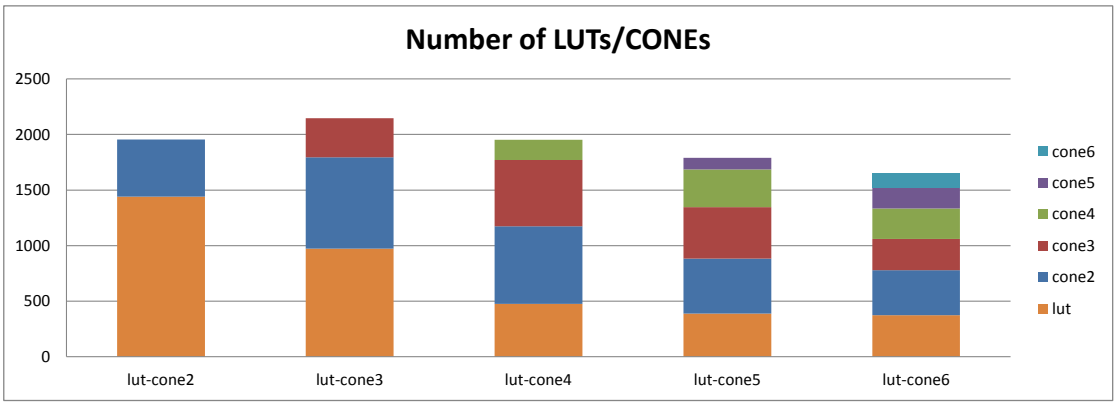


Figure 7.10: Number and type of logic blocks used in the various architectures and with the various mapping strategies.

implementations on the pure 6-AIC architecture and on the hybrid architecture with 6-AIC and 5-AIC base blocks are 27%, 32%, and 22% faster than the baseline FPGA, respectively.

Figure 7.10 presents the distribution of LUTs and AICs for the different architectures. This figure shows that when deeper cones are allowed, less LUTs are used. Moreover, in each case the usage of each AIC type has a reverse relation with the size of the AIC. This means that the chance of mapping a node with smaller AIC is always higher. Since each of these LUTs and AICs are packed into clusters, the numbers presented there do not indicate the real logic area of the circuits. On the contrary, Figure 7.11 illustrates the number of clusters after packing: this is proportional to the active area since the area of an AIC cluster is close to the area of a LAB (see Table 7.2) and both have the same I/O bandwidth. For some benchmarks, either the *LUT/6-AIC* hybrid architecture or the baseline FPGA display the lowest area; however, the *LUT/5-AIC* architecture always results in the smallest used area at a much better delay than the baseline FPGA and a slightly worse one than *LUT/6-AIC*—refer to Figure 7.9. The two hybrid architectures define Pareto optimal points.

The hybrid structure of the proposed FPGA with the different cluster types needs to fix the right ratio of the two flavors of logic blocks. The packing results indicate that this ratio varies

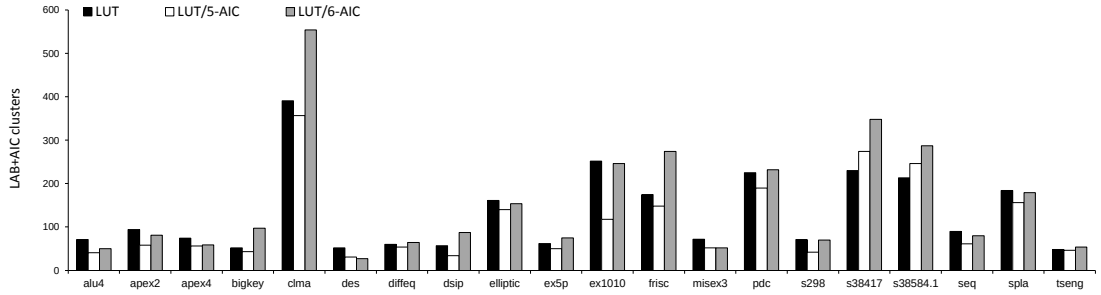


Figure 7.11: Area measured as the total number of clusters used, completely or partially. LABs and AIC clusters occupy approximately the same area. On average, LUT/5-AIC uses 16% less resources than LUT-only.

from one circuit to the other, making this problem not straightforward. We have made some preliminary experiments on this front, and we have fixed the ratio of LAB columns to AIC clusters to 1 : 4. The advantage of AICs is that any logic function that is mapped to an LUT is mappable to one or more AICs. The reverse is also true. Therefore, it is possible to switch to another logic block type, when we run out of one type. Moreover, considering the small size of the AIC blocks, it is quite feasible to add them as *shadow blocks* of the LUTs to the LUT clusters, by reusing the existing input crossbar. This provides the option to use either LUTs or AICs depending on the requirements. Though, adding AICs as shadow blocks of LUTs remains as the future work.

Table 7.5 presents the average wire length of each benchmark, in the baseline architecture (no AIC clusters) and in the two best hybrid architectures, with the number of routing channels fixed to 180 for all the experiments. We observe that there is a fairly high variability—but averages are very similar (15.8, 16.1, and 17.3 respectively)—with a small trend against our hybrid architecture.

7.8 Related Work

Leveraging the properties of logic synthesis netlist to simplify the logic block of FPGAs is a current research topic [8, 9]. For instance, based on the observation that circuits represented using AIGs frequently have a trimming input, a low-cost and still LUT-based logic block was designed that requires less silicon area, but it does not improve the delay [9]. Albeit somehow similar in its inspiration to modern synthesis, our work is more radical in using the AIGs to inspire the new logic cell.

Although LUT-based logic blocks dominate the architectures of commercial FPGA, PAL-like logic blocks have also been explored. In recent times, it has been shown that a fairly small PAL-like structure, with 7–10 inputs and 10–13 product terms, obtains performance gains at the price of an increase in area [23]. Much earlier, some authors have shown that K-input

Benchmark	LUT	LUT/ 5-AIC	LUT/ 6-AIC
alu4	14.9	10.59	11.32
apex2	16.4	15.2	12.9
apex4	15.5	16.1	14.1
bigkey	14.3	12.6	11.6
clma	20.8	22.9	25.5
des	14.6	16.1	15.1
diffeq	10.4	13.4	13.8
dsip	18.6	17.4	12.5
elliptic	15.5	16.6	16.7
ex5p	11.2	15.9	23.2
ex1010	23.8	18.2	30.3
frisc	18.8	19.35	23.2
misex3	14	12	13
pdc	22.8	23.4	21.2
s298	13.2	9.7	15.8
s38417	12.5	18.2	19
s38584.1	11.5	18.4	17.5
seq	17.1	15.5	15.5
spla	21.5	18.8	21.1
tseng	8.3	13.1	12.5

Table 7.5: Average wire length in units of one CLB segments.

multiple-output PAL-style logic blocks are more area efficient than 4-input LUTs. However, the idea was abandoned because PAL-based implementations typically consumed excessive static power [48]. Our solution moves away from the typical logic block natural of traditional logic synthesis, and we have shown that it seems possible to improve both area and delay compared to LUT-based FPGAs.

There are also numerous pieces of work which have adapted or created reconfigurable logic blocks to specific needs, often by adding dedicated logic gates to existing LUTs. Among these, one can mention GARP [36] and Chimaera [95] for datapath oriented processor acceleration, macro gates [40] for implementing wide logic gates, and various sorts of fast carry chains beyond those available commercially [69]. Although they all somehow question the pure LUT as the most efficient building block, they tend to introduce modifications that are never real generic alternatives.

7.9 Conclusions

As several people before us, we have recognized that LUTs have many advantages but, frequently, the price to pay for these advantages is unreasonably high. We have thus explored new

logic blocks inspired by recent trends in the circuits representations used in logic synthesis: we came to define AICs, which are simply the natural configurable circuits homologue of the newly popular AIGs. These new logic blocks have considerably simple structure and elude the excess flexibility of LUTs.

The alternate FPGA architecture that we proposed in this chapter is a mixture of LUTs- and AICs-clusters keeping the original structure of the inter-cluster routing network of FPGAs. The results of this work are encouraging: On one hand, delay is bound to decrease as both logic delay and the number of logic blocks on the critical path reduce. With a fairly rough routing delay model, we observe a delay reduction of up to 32%. On the other hand, the number of logic blocks (all of similar area) consumed by the benchmark circuits is also generally reduced; with one of our mapping approaches, the area is reduced on average by 16%. Future work will necessarily need to address placement and routing much more precisely than we had the chance to. Also, other less conservative architectures may prove more advantageous than those explored. Nevertheless, we think that our first results are sufficiently encouraging for the approach to deserve a closer inspection.

Compromising on flexibility for efficiency was the second goal of this thesis that we achieved with the introduction of AICs. Using this new non-LUT logic block, we managed to enhance the soft-logic implementation of FPGA circuits. This impacts any application that is mapped to FPGAs, as soft-logic is involved in the implementation of all applications. However, we observed that when AICs are merged with LUTs, the best performance is achieved, and a hybrid solution is superior. Hence, the *logic chain* structure that was presented in Chapter 6 can still be used to improve the LUT part of FPGAs.

Finally, we observed that AICs, similar to LUTs, does not perform well for arithmetic circuits such as multipliers and multi-input additions. Therefore, it is still essential to have the hard-logic including the carry chains and DSP blocks for implementing such circuits. This means that the contributions of Chapters 3, 4, and 5 are complementary to the ones in this chapter and Chapter 6.

8 Conclusions and Future Work

Today, semiconductor design costs including development, manufacturing, and verification costs limit economically the Moore's law. This had led to the formulation of *Moore's second law*, which states that the capital cost of a semiconductor fabrication also increases exponentially over time. Meanwhile, the potential of semiconductor programmable ICs—led by FPGA devices—seems endless, because they can help to break the economical barrier of Moore's law. However, in 2010, FPGAs only captured up to 2% of the global semiconductor market, compared to about 30% for ASICs and ASSPs, combined. The main reason for this low market share of FPGAs is that the higher volume applications (typically above 100K units per year), where FPGAs are less competitive, account for the majority of the revenue. In order to live up to its promise, the programmable hardware industry will have to overcome some significant challenges.

The major challenge with FPGAs is their poor efficiency compared to ASICs, which creates a significant gap between FPGAs and ASICs. Current modern FPGAs are electronically programmable devices, which can implement arbitrary digital circuits. This flexibility of FPGAs comes from having fully programmable logic blocks and a routing network. However, this flexibility has a price, which is the mentioned efficiency gap. Generally, FPGAs have heterogeneous structure comprised of fully flexible soft-logic mixed with efficient and dedicated hard-logic. The drawback of FPGAs architecture is that their soft-logic, which is used to (fully or partially) implement any application, is inefficient, and their hard-logic goes wasted when it is not used. In other words, current FPGAs suffer from the inefficiency and inflexibility of their soft- and hard-logic, respectively.

In this thesis, we presented a *roadmap* to address the above challenges. In the proposed roadmap, we approached the problem from two complementary directions: (1) adding more generality to the hard-logic of FPGAs, which enables more applications to take advantage of these efficient resources, and (2) enhancing the efficiency of the soft-logic of FPGAs by adapting the soft-logic to applications requirements. In other words, the goal of this thesis was to compromise on efficiency of the hard-logic for flexibility, and to compromise on flexibility of the soft-logic for efficiency. Following this roadmap, we presented synthesis and architectural

techniques for each direction.

In principle, the hard-logic in FPGAs is adapted to critical arithmetic operations that appear in the pre-synthesis representation of applications. Multiplication and addition are such operations. Hence, to improve the FPGA implementation of applications, dedicated structures are integrated within the soft-logic of FPGAs. However, as we discussed earlier, these dedicated resources are wasted when they are not used. Hence, it is essential to expand the applicability of these resources. In the first part of this thesis, we presented a number of synthesis and architectural methods to approach this goal.

We categorized the hard-logic of FPGAs into two groups, depending on how closely the hard-logic is integrated into the soft-logic, and for each category, we presented a few solutions. In current FPGAs, we have carry chains and adders that are tightly coupled with LUTs and are mainly intended for ripple carry addition. Such carry chains are naturally inflexible, as they are hard-wired connections. However, we introduced a new mapping technique that can use such carry chains for carry-save arithmetic. The basic idea is to explore primitives that are nicely mapped to a combination of LUTs, carry logic, and carry chains. Using these primitives, we developed a carry-save arithmetic library, which is utilized by a high-level mapping technique. The interesting part of this work is the way that we overlap the primitives on the carry chains by logically breaking the chain. This technique allows to increase the logic density considerably.

Moreover, we identified the limits of the carry chains, using the challenges that we faced in the mapping contribution. This motivated us to design a new carry chain that has a non-propagating nature and can further enhance the implementation of carry-save arithmetic applications on FPGAs. This new carry chain requires to revise the logic blocks of FPGAs slightly, and it can be constructed by reusing and restructuring the existing one. With this new design, where both the conventional and the new carry chains present, the hard-logic that is coupled with LUTs can be exploited to improve a richer subset of applications. This conforms to the first direction of the roadmap.

On the other hand, these new carry chains add little area and delay overheads to the current logic blocks. These overheads, indeed, can be justified considering the importance of multi-input addition in applications and the great benefits of new carry chains for implementing multi-input additions. However, the FPGA vendors may argue that even these little overheads can be critical for the applications that do not use these carry chains, and thus they might be unwilling to change the structure of the logic blocks. In this case, the mapping approach that was presented in the Chapter 3 will be the best (soft-logic) option to implement compressor trees.

In addition to the mentioned type of the hard-logic that is tightly coupled with the soft-logic in FPGAs, there is the stand-alone hard-logic that is interfaced with the soft-logic through the general routing network in the island-style FPGAs. DSP blocks, embedded memories, and hard processor cores are some instances. DSP blocks mainly perform multiplication, in addition to a few other arithmetic operations. The problem with these DSP blocks is that they

are (structurally) highly inflexible, they support very limited multiplication bit-widths, and they lack carry-save arithmetic. In this thesis, we presented a new architecture for the DSP blocks (1) to increase their flexibility in supporting various multiplication bit-widths, and (2) to expand their functionality to support carry-save arithmetic, by reusing the multiplication resources in the DSP blocks. For this purpose, we took a current DSP block as the base and redesigned its structure to make it versatile.

Experimental results revealed that using the presented DSP block for implementing compressor trees significantly reduces their delay and area, compared to the soft-logic implementations of the compressor trees that were presented in Chapters 3 and 4. Therefore, the first choice for implementing the compressor trees on FPGA is using the presented DSP block. However, the number of DSP blocks in an FPGA is limited, and it is still crucial to have efficient soft-logic implementation of the compressor trees on FPGAs, when no free DSP block exist. This justifies the contributions of Chapters 3 and 4.

The second direction that the thesis roadmap recommends is boosting the efficiency of the soft-logic, which may result in loosing some flexibility. To follow this path, we inspected the *post-synthesis* representations of circuits to explore logic patterns that may not be visible pre-synthesis and can be used to simplify the design of the soft-logic blocks. Passing through technology independent logic synthesis that is similar for all applications, provides the opportunity to explore more general patterns, which are less dependent on how each application is modeled pre-synthesis.

The first step in this direction was to bypass the routing wires in the soft-logic, using hard-wired connections that we call *logic chains*. The idea of logic chains was inspired from the idea of carry chains, and the logic chains are intended for implementing the long chains of logic that appear in the post-synthesis netlist of applications. Logic chains are fixed connections that cascade fracturable LUTs, by which larger LUTs are constructed along the chain. The advantages of the logic chain include (1) reducing the pressure on the routing network of FPGAs, (2) increasing the logic density of the soft-logic blocks without any change in the routing interface of the block, and (3) reducing the delay of critical paths by replacing the interconnecting wires with fast and fixed wires that have near to zero delay.

In a complementary and more disruptive work, we explored logic patterns that led to design non-LUT soft-logic blocks for FPGAs. Most of current FPGAs soft-logic blocks are LUT-based, and thus structurally bounded to the limitations of LUTs. Although LUTs are so general that can implement any logic function and can simplify the mapping problem, the complexity of their structure is high due to their excess flexibility. For instance, LUTs rarely with more than 4-6 inputs have been used, as increasing the number of LUT inputs to cover larger parts of a circuit has an exponential cost in the LUT complexity. Inspired by recent trends in synthesis and verification, we explored blocks based on And-Inverter Graphs (AIGs): they have a complexity which is only linear in the number of inputs, they sport the potential for multiple independent outputs, and the delay is only logarithmic in the number of inputs. Of

course, these new blocks are extremely less flexible than LUTs; yet, we showed (1) that effective mapping algorithms exist, (2) that, due to their simplicity, poor utilization is less of an issue than with LUTs, and (3) that a few LUTs can still be used in extreme unfortunate cases.

However, we observed that AICs, similar to LUTs, does not perform well for arithmetic circuits such as multipliers and multi-input additions. Therefore, it is still essential to have the hard-logic including the carry chains and DSP blocks for implementing such circuits. This means that the contributions of Chapters 3, 4, and 5 are complementary to the ones in Chapters 6 and 7.

To summarize, in this thesis, we attempted to close the existing efficiency gap—delay and area—between FPGAs and ASICs by increasing the flexibility of the hard-logic and the efficiency of the soft-logic of FPGAs, as shown in Figure 8.1. In our experiments, we observed that the performance of certain arithmetic circuits could be enhanced up to 40%, when the hard-logic of FPGAs is adapted for such circuits. On the other hand, we observed that the performance of generic circuits could be improved up to 32%, using the presented soft-logic block—And-Inverter Cone; moreover, up to 37% saving in the usage of local routing wires were obtained, when the LUT-based logic chain is used. This latter achievement indicates that the routing network of future FPGAs should be revised and lightened with the integration of such hard-wired connections. This would reduce the silicon area that is devoted to the routing network of FPGAs, which can close the area gap between FPGAs and ASICs.

8.1 Future Work

Adapting the hard-logic of FPGAs for the applications requirement is a research direction that will be followed, as new applications emerge quickly. Hence, it is essential to identify the new critical requirements of applications for revising the structure of the hard-logic in FPGAs. On the other hand, for enhancing the soft-logic, we could explore few design points in this thesis due to the time constraint, and we believe that these ideas deserve future investigations.

The idea of *logic chain* that was presented in chapter 6 is an area that needs future explorations. In this thesis, we introduced 1-dimensional logic chains, but one research area is to generalize and extend this idea to 2-dimensional hard-wired connections. Moreover, the presented heuristic for finding the logic candidates that can be mapped on the logic chain simply searches for the nodes in a technology mapped netlist, which is generated unaware of the logic chains. While, if logic decomposition is combined with the technology mapping, then more nodes would be eligible to be mapped on the chains. Moreover, the logic decomposition will enable to exploit the specific fixed interconnect structure between adjacent LUTs, which can enhance the presented chaining heuristic.

In addition, we also discovered the value of AIG-based logic blocks (AICs), which is still far from comprehensive AIC-based design space exploration. Future work will necessarily need to address several challenges such as adapting the routing network to AICs and using AICs as

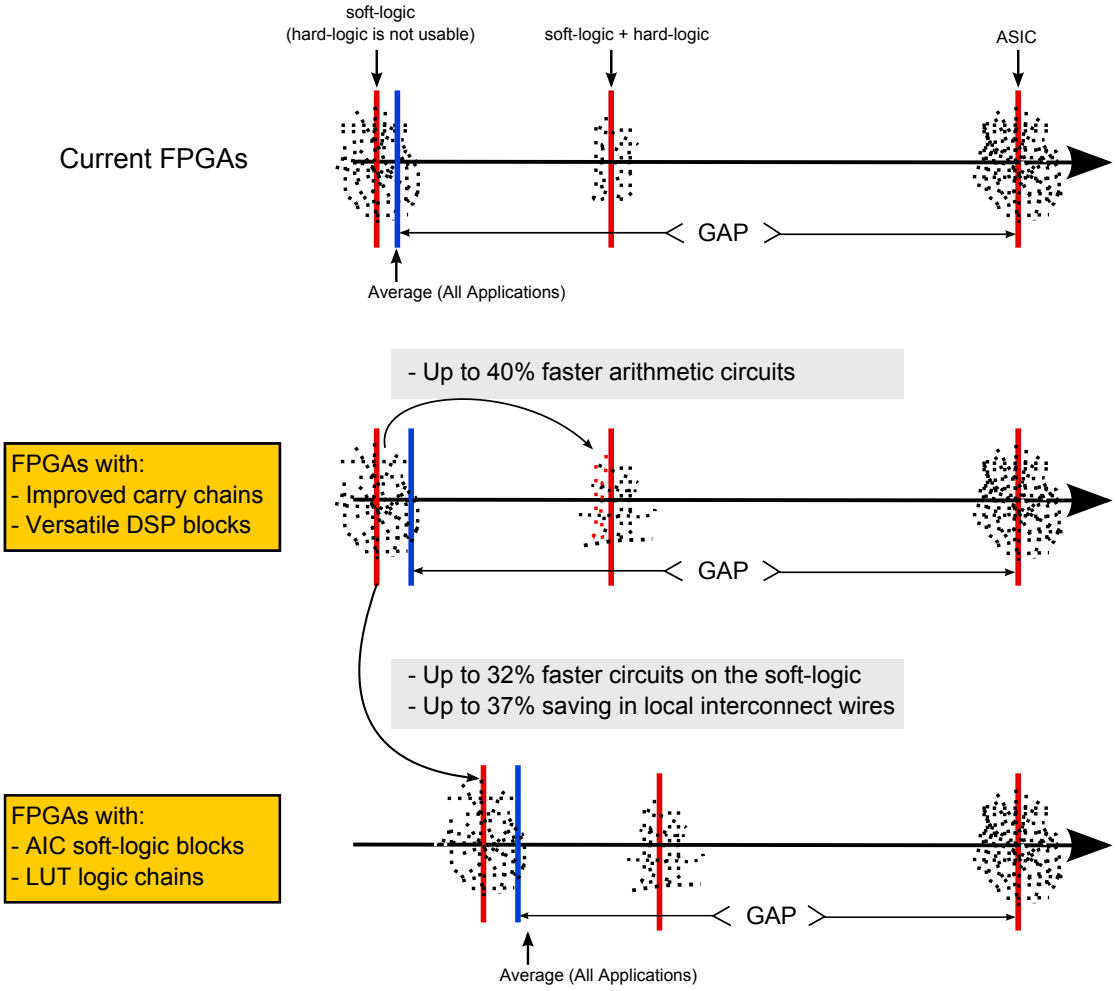


Figure 8.1: Achievements of this thesis in closing the delay and area gaps between FPGAs and ASICs. By increasing the flexibility of the hard-logic, we improved the performance of carry-save-based arithmetic circuits. Moreover, we enhanced both performance and area of generic logic implementation on FPGAs, by introducing novel architectures for the soft-logic.

the shadow blocks of the current logic blocks of FPGAs. Each of these challenges are extensive research directions, which could be focused in future. In this thesis, we simply assumed that the routing structure of the FPGA remains unchanged for AICs, while other less conservative architectures may be superior than those explored. Nevertheless, we think that our first results are sufficiently encouraging for the approach to deserve a closer inspection.

Bibliography

[1] AHMED, E., AND ROSE, J. The effect of LUT and cluster size on deep-submicron FPGA performance and density. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems VLSI-12*, 3 (Mar. 2004), 288–289.

[2] ALLEN, J. R., KENNEDY, K., PORTERFIELD, C., AND WARREN, J. D. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Language* (Austin, Tex., Jan. 1983), pp. 177–89.

[3] ALTERA CORPORATION. *Cyclone Device Handbook, vols. 1*. `http://www.altera.com/literature/`.

[4] ALTERA CORPORATION. PowerPlay Early Power Estimator User Guide. `http://www.altera.com/literature/`.

[5] ALTERA CORPORATION. *Stratix Device Handbook, vols. 1 and 2*. `http://www.altera.com/literature/`.

[6] ALTERA CORPORATION. *Stratix II Device Handbook, vols. 1 and 2*. `http://www.altera.com/literature/`.

[7] ALTERA CORPORATION. *Stratix III Device Handbook, vols. 1 and 2*. `http://www.altera.com/literature/`.

[8] ANDERSON, J. H., AND WANG, Q. Improving logic density through synthesis-inspired architecture. In *Proceedings of the 19th International Conference on Field-Programmable Logic and Applications* (Prague, Aug. 2009), pp. 105–11.

[9] ANDERSON, J. H., AND WANG, Q. Area-efficient FPGA logic elements: Architecture and synthesis. In *Proceedings of the Asia and South Pacific Design Automation Conference* (Yokohama, Japan, Jan. 2011), pp. 369–75.

[10] BEAUCHAMP, M. J., HAUCK, S., UNDERWOOD, K. D., AND HEMMERT, K. S. Architectural modifications to enhance the floating-point performance of FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems VLSI-16*, 2 (Feb. 2008), 177–87.

[11] BERKELEY LOGIC SYNTHESIS AND VERIFICATION GROUP. *ABC: A System for Sequential Synthesis and Verification*. Berkeley, Calif., Feb. 2011. Release 10216, `http://www.eecs.berkeley.edu/~alanmi/abc/`.

[12] BETZ, V., AND ROSE, J. VPR: a new packing, placement, and routing tool for FPGA research. In *Proceedings of the 7th International Conference on Field-Programmable Logic and Applications* (London, UK, Sept. 1997), pp. 213–222.

[13] BETZ, V., ROSE, J., AND MARQUARDT, A. *Architecture and CAD for deep-submicron FPGAs*. Kluwer Academic, Boston, Mass., 1999.

[14] CEVRERO, A., ATHANASOPOULOS, P., PARANDEH-AFSHAR, H., VERMA, A. K., BRISK, P., NICOPOULOS, C., ATTARZADEH NIAKI, S. H., GURKAYNAK, F. K., LEBLEBICI, Y., AND IENNE, P. Field Programmable Compressor Trees: Acceleration of multi-input addition on FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS) 2*, 2 (June 2009), 13:1–13:36.

[15] CHEN, C.-Y., CHIEN, S.-Y., HUANG, Y.-W., CHEN, T.-C., WANG, T.-C., AND CHEN, L.-G. Analysis and architecture design of variable block-size motion estimation for H.264/AVC. *IEEE Transactions on Circuits and Systems II—Analog and Digital Signal Processing 53*, 2 (Feb. 2006), 578–593.

[16] CHEN, G., AND CONG, J. Simultaneous logic decomposition with technology mapping in FPGA designs. In *Proceedings of the 9th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, Calif., Feb. 2001), pp. 48–55.

[17] CHEREPACHA, D., AND LEWIS, D. DP-FPGA: An FPGA architecture optimized for datapaths. *VLSI Design 4*, 4 (1996), 329–43.

[18] CHONG, Y. J., AND PARAMESWARAN, S. Flexible multi-mode embedded floating-point unit for field programmable gate arrays. In *Proceedings of the 17th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, Calif., Feb. 2009), pp. 171–180.

[19] CHOW, P., SEO, S., AU, D., FALLAH, B., LI, C., AND ROSE, J. A 1.2 μm CMOS FPGA using cascaded logic blocks and segmented routing. In *Proceedings of the International Workshop on Field-Programmable Logic and Applications* (Oxford, UK, Sept. 1991), pp. 91–102.

[20] CONG, J., AND DING, Y. An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. In *Proceedings of the International Conference on Computer Aided Design* (Santa Clara, Calif., Nov. 1992), pp. 49–53.

[21] CONG, J., AND DING, Y. FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13*, 1 (Jan. 1994), 1–12.

[22] CONG, J., AND DING, Y. On area/depth trade-off in LUT-based FPGA technology mapping. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 2*, 2 (June 1994), 137–48.

[23] CONG, J., AND HUANG, H. Technology mapping and architecture evaluation for k/m-macrocell-based FPGAs. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 10*, 1 (Jan. 2005), 3–23.

[24] CONG, J., WU, C., AND DING, Y. Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution. In *Proceedings of the 7th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, Calif., Feb. 1999), pp. 29–35.

[25] CZAJKOWSKI, T., AND BROWN, S. Functionally linear decomposition and synthesis of logic circuits for FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 12*, 27 (Dec. 2008), 2236–2249.

[26] DADDA, L. Some schemes for parallel multipliers. *Alta Frequenza XXXIV* (1965), 349–56.

[27] DEHON, A. Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization). In *Proceedings of the 7th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, Calif., Feb. 1999), pp. 69–76.

[28] FADAVI-ARDEKANI, J. $M \times N$ Booth encoded multiplier generator using optimized Wallace trees. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems VLSI-1*, 2 (June 1993), 120–25.

[29] FARRAHI, A., AND SARRAFZADEH, M. Complexity of the lookup-tale minimization problem for FPGA technology mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13*, 11 (Nov. 1994), 1319–1332.

[30] FARRAHI, A., AND SARRAFZADEH, M. FPGA technology mapping for power minimization. In *Proceedings of the 4th International Workshop on Field-Programmable Logic and Applications* (Prague, Czech Republic, Sept. 1994), pp. 66–67.

[31] FREDERICK, M. T., AND SOMANI, A. K. Multi-bit carry chains for high-performance reconfigurable fabrics. In *Proceedings of the 16th International Conference on Field-Programmable Logic and Applications* (Madrid, Aug. 2006), pp. 1–6.

[32] FREDERICK, M. T., AND SOMANI, A. K. Non-arithmetic carry chains for reconfigurable fabrics. In *Proceedings of the 25th IEEE International Conference on Computer Design* (Lake Tahoe, Calif., Oct. 2007), pp. 137–143.

[33] FREDERICK, M. T., AND SOMANI, A. K. Beyond the arithmetic constraint: depth-optimal mapping of logic chains in LUT-based FPGAs. In *Proceedings of the 16th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, Calif., Feb. 2008), pp. 37–46.

[34] GROVER, R. S., SHANG, W., AND LI, Q. A faster distributed arithmetic architecture for FPGAs. In *Proceedings of the 10th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, Calif., Feb. 2002), pp. 31–39.

[35] HAUCK, S., HOSLER, M. M., AND FRY, T. W. High-performance carry chains for FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems VLSI-8*, 2 (Apr. 2000), 138–47.

[36] HAUSER, J. R., AND WAWRZYNEK, J. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines* (Napa Valley, Calif., Apr. 1997), pp. 12–21.

[37] HELLERMAN, L. A catalog of three-variable Or-Invert and And-Invert logical circuits. *IEEE Transactions on Electronic Computers EC-12*, 3 (June 1963), 198–223.

[38] HO, C. H., ET AL. Virtual embedded blocks: a methodology for evaluating embedded elements in FPGAs. In *Proceedings of the 14th IEEE Symposium on Field-Programmable Custom Computing Machines* (Napa, Calif., Apr. 2006), pp. 35–44.

[39] HU, Y., DAS, S., AND HE, L. Design, synthesis, and evaluation of heterogeneous FPGA with mixed LUTs and macro-gates. In *Proceedings of the 16th International Workshop on Logic and Synthesis* (San Diego, Calif., June 2007).

[40] HU, Y., DAS, S., TRIMBERGER, S., AND HE, L. Design, synthesis, and evaluation of heterogeneous FPGA with mixed LUTs and macro-gates. In *Proceedings of the International Conference on Computer Aided Design* (San Jose, Calif., Nov. 2007), pp. 188–93.

[41] HUTTON, M., ET AL. Improving FPGA performance and area using an adaptive logic module. In *Proceedings of the 14th International Conference on Field-Programmable Logic and Applications* (Antwerp, Belgium, Aug. 2004), pp. 135–144.

[42] JAMIESON, P., AND ROSE, J. Architecting hard crossbars on FPGAs and increasing their area-efficiency with shadow clusters. In *Proceedings of the IEEE International Conference on Field Programmable Technology* (Bangkok, Dec. 2007), pp. 57–64.

[43] JAMIESON, P. A., AND ROSE, J. Enhancing the area efficiency of FPGAs with hard circuits using shadow clusters. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 18*, 12 (Dec. 2010), 1696–1709.

[44] KAMP, W., BAINBRIDGE-SMITH, A., AND HAYES, M. Efficient implementation of fast redundant number addrs for long word-lengths in FPGAs. In *Proceedings of the IEEE International Conference on Field Programmable Technology* (Sydney, Australia, Dec. 2008), pp. 239–246.

[45] KASTNER, R., KAPLAN, A., OGRENCI MEMIK, S., AND BOZORGZADEH, E. Instruction generation for hybrid reconfigurable systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 7*, 4 (Oct. 2002), 605–27.

[46] KAVIANI, A., AND BROWN, S. D. Hybrid FPGA architecture. In *Proceedings of the 4th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, Calif., Feb. 1996), pp. 3–9.

[47] KAVIANI, A., VRANESIC, D., , AND BROWN, S. Computational field programmable architecture. In *Proceedings of the IEEE Custom Integrated Circuit Conference* (Santa Clara, Calif., May 1998), pp. 261–64.

[48] KOULOHERIS, J. L., AND EL GAMAL, A. PLA-based FPGA area versus cell granularity. In *Proceedings of the IEEE Custom Integrated Circuit Conference* (Boston, Mass., May 1992), pp. 4.3.1–4.3.4.

[49] KUKIMOTO, Y., BRAYTON, R., AND SAWKARY, P. Delay-optimal technology mapping by DAG covering. In *Proceedings of the 35th Design Automation Conference* (San Francisco, Calif., June 1998), pp. 348–51.

[50] KUON, I., AND ROSE, J. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD-26*, 2 (Feb. 2007), 203–15.

[51] KWON, O., NOWKA, K., AND SWARTZLANDER JR., E. E. A 16-bit by 16-bit MAC design using fast 5:3 compressor cells. *Journal of VLSI Signal Processing 31*, 2 (June 2002), 77–89.

[52] LATTICE SEMICONDUCTOR CORPORATION. *LatticeXP family data sheet*. `http://www.latticesemi.com/lit/docs/datasheets/fpga/xp_data_sheet.pdf`.

[53] LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture* (Research Triangle Park, N.C., Dec. 1997), pp. 330–35.

[54] LEIJTEN-NOWAK, K., AND VAN MEERBERGEN, J. L. An FPGA architecture with enhanced datapath functionality. In *Proceedings of the 11th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, Calif., Feb. 2003), pp. 195–204.

[55] LEVIN, I., AND PINTER, R. Y. Realizing expression graphs using table-lookup FPGAs. In *Proceedings of the 30th Design Automation Conference* (Dallas, Tex., June 1993), pp. 306–11.

[56] LEWIS, D., ET AL. The Stratix II logic and routing architecture. In *Proceedings of the 13th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, Calif., Feb. 2005), pp. 14–20.

[57] LUU, J., ANDERSON, J. H., AND ROSE, J. Architecture description and packing for logic blocks with hierarchy, modes and complex interconnect. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, Calif., Feb. 2011), pp. 227–36.

[58] MANOHARARAJAH, V., AND BROWN, S. Heuristics for area minimization in LUT-based FPGA technology mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 25*, 11 (Nov. 2006), 2331–40.

[59] MATSUNAGA, T., KIMURA, S., AND MATSUNAGA, Y. Multi-operand adder synthesis on FPGAs using generalized parallel counters. In *Proceedings of the Asia and South Pacific Design Automation Conference* (Taipei, Taiwan, Jan. 2010), pp. 337–342.

[60] MIRZAEI, S., HOSANGADI, A., AND KASTNER, R. High speed FIR filter implementation using add and shift method. In *Proceedings of the International Conference on Computer Design* (San Jose, Calif., Oct. 2006), pp. 1–4.

[61] MISHCHENKO, A., CHATTERJEE, S., AND BRAYTON, R. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In *Proceedings of the 43rd Design Automation Conference* (San Francisco, Calif., July 2006), pp. 532–36.

[62] MORA MORA, H., MORA PASCUAL, J., SÁNCHEZ ROMERO, J., AND PUJOL LÓPEZ, F. Partial product reduction based on look-up tables. In *Proceedings of the 14th IEEE Symposium on Field-Programmable Custom Computing Machines* (Hyderabad, India, Jan. 2006), pp. 399–404.

[63] OKLOBDZIJA, V. G., AND VILLEGER, D. Improving multiplier design by using improved column compression tree and optimized final adder in CMOS technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems VLSI-3*, 2 (June 1995), 292–301.

[64] ORTIZ, M.AND QUILES, F., JORMIGO, J., JAIME, F. J., VILLALBA, J., AND ZAPATA, E. L. Efficient implementation of carry-save adders in FPGAs. In *Proceedings of the 20th International Conference on Application-specific Systems, Architectures and Processors* (Boston, USA, July 2009), pp. 207–210.

[65] PAIDIMARRI, A., CEVRERO, A., BRISK, P., AND IENNE, P. FPGA implementation of a single-precision floating-point multiply-accumulator with single-cycle accumulation. In *Proceedings of the 17th IEEE Symposium on Field-Programmable Custom Computing Machines* (Napa Valley, Calif., Apr. 2009), pp. 267–70.

[66] PARANDEH-AFSHAR, H., BRISK, P., AND IENNE, P. Efficient synthesis of compressor trees on FPGAs. In *Proceedings of the Asia and South Pacific Design Automation Conference* (Seoul, Korea, Jan. 2008), pp. 138–43.

[67] PARANDEH-AFSHAR, H., BRISK, P., AND IENNE, P. Improving synthesis of compressor trees on FPGAs via integer linear programming. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition* (Munich, Mar. 2008), pp. 1256–61.

[68] PARANDEH-AFSHAR, H., BRISK, P., AND IENNE, P. A novel FPGA logic block for improved arithmetic performance. In *Proceedings of the 16th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, Calif., Feb. 2008), pp. 171–80.

[69] PARANDEH-AFSHAR, H., BRISK, P., AND IENNE, P. An FPGA logic cell and carry chain configurable as a 6:2 or 7:2 compressor. *ACM Transactions on Reconfigurable Technology and Systems (TRETS) 2*, 3 (Sept. 2009), 19:1–19:42.

[70] PARANDEH-AFSHAR, H., CEVRERO, A., ATHANASOPOULOS, P., BRISK, P., LEBLEBICI, Y., AND IENNE, P. A flexible DSP block to enhance FPGA arithmetic performance. In *Proceedings of the IEEE International Conference on Field Programmable Technology* (Sydney, Dec. 2009), pp. 70–77.

[71] PARANDEH-AFSHAR, H., VERMA, A. K., BRISK, P., AND IENNE, P. Improving FPGA performance for carry-save arithmetic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems VLSI-18*, 4 (Apr. 2010), 578–90.

[72] PARANDEH-AFSHAR, H., ZGHEIB, G., BRISK, P., AND IENNE, P. Routing wire optimization through generic synthesis on FPGA carry chains. In *Proceedings of the 48th Design Automation Conference* (San Diego, Calif., June 2011). Rejected.

[73] PARHAMI, B. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, New York, 2010.

[74] POLDRE, J., AND TAMMEMAE, K. Reconfigurable multiplier for virtex FPGA family. In *Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications* (Glasgow, Aug. 1999), pp. 359–64.

[75] ROSE, J., AND BROWN, S. Flexibility of interconnection structures for fieldprogrammable gate arrays. *IEEE Journal of Solid-State Circuits 26*, 3 (Mar. 1991), 277–282.

[76] SCHLAG, M., KONG, J., AND CHAN, P. K. Routability-driven technology mapping for lookup table-based FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13*, 1 (Jan. 1994), 13–26.

[77] SHAMS, A., PAN, W., CHANDANANDAN, A., AND BAYOUMI, M. A high-performance 1D-DCT architecture. In *Proceedings of the IEEE International Symposium on Circuits and Systems* (Geneva, Switzerland, May 2000), pp. 521–524.

[78] SONG, P., AND DE MICHELI, G. Circuit and architecture trade-offs for high speed multiplication. *IEEE Journal of Solid-State Circuits 26*, 9 (Sept. 1991), 663–70.

[79] SRIRAM, S., BROWN, K., DEFOSSEUX, R., MOERMAN, F., PAVIOT, O., SUNDARARAJAN, V., AND GATHERER, A. A 64-channel programmable receiver chip for 3G wireless infrastructure. In *Proceedings of the IEEE Custom Integrated Circuit Conference* (San Jose, Calif, Sept. 2005), pp. 59–62.

[80] STELLING, P. F., MARTEL, C. U., OKLOBDZIJA, V. G., AND RAVI, R. Optimal circuits for parallel multipliers. *IEEE Transactions on Computers C-47*, 3 (Mar. 1998), 273–85.

[81] STELLING, P. F., AND OKLOBDZIJA, V. G. Design strategies for optimal hybrid final adders in a parallel multiplier. *Journal of VLSI Signal Processing 14* (Dec. 1996), 321–31.

[82] STENZEL, W. J., KUBITZ, W. J., AND GARCIA, G. H. A compact high-speed parallel multiplication scheme. *IEEE Transactions on Computers C-26*, 10 (Oct. 1997), 948–957.

[83] SWARTZLANDER, JR., E. E. Parallel counters. *IEEE Transactions on Computers C-22*, 11 (Nov. 1973), 1021–24.

[84] SYNOPSYS. *Creating High-Speed Data-Path Components—Application Note*, Aug. 2001. Version 2001.08.

[85] UM, J., AND KIM, T. An optimal allocation of carry-save-adders in arithmetic circuits. *IEEE Transactions on Computers C-50*, 3 (Mar. 2001), 215–33.

[86] VERMA, A. K., BRISK, P., AND IENNE, P. Data-flow transformations to maximize the use of carry-save representation in arithmetic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD-27*, 10 (Oct. 2008), 1761–74.

[87] VERMA, A. K., AND IENNE, P. Automatic synthesis of compressor trees: Reevaluating large counters. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition* (Nice, France, Apr. 2007), pp. 443–48.

[88] WALLACE, C. S. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers C-13*, 2 (Feb. 1964), 14–17.

[89] WEINBERGER, A. 4-2 carry-save adder module. *IBM Technical Disclosure Bulletin 23*, 8 (Jan. 1981), 172.

[90] WONG, H., BETZ, V., AND ROSE, J. Comparing FPGA vs. custom cmos and the impact on processor microarchitecture. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, Calif., Feb. 2011), pp. 5–14.

[91] XILINX INC. *Virtex-4 User Guide*. `http://www.xilinx.com/`.

[92] XILINX INC. *Virtex-5 User Guide*. `http://www.xilinx.com/`.

[93] YANG, H., AND WONG, D. F. Edge-map: Optimal performance driven technology mapping for iterative LUT based FPGA designs. In *Proceedings of the International Conference on Computer Aided Design* (San Jose, Calif., Nov. 1994), pp. 150–55.

[94] YANG, S. Logic synthesis and optimization benchmarks user guide, version 3.0. Technical report, Microelectronics Center of North Carolina, Research Triangle Park, N.C., Jan. 1991.

[95] YE, Z. A., MOSHOVOS, A., HAUCK, S., AND BANERJEE, P. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the 27th Annual International Symposium on Computer Architecture* (Vancouver, June 2000), pp. 225–35.

[96] ZUCHOWSKI, P. S., REYNOLDS, C. B., GRUPP, R. J., DAVIS, S. G., CREMEN, B., AND TROXEL, B. A hybrid ASIC and FPGA architecture. In *Proceedings of the International Conference on Computer Aided Design* (San Jose, Calif., Nov. 2002), pp. 187–94.

Curriculum Vitae

Personal Data

Name: Hadi P. Afshar
Date of Birth: Nov, 1978
Nationality: Iranian
E-mail address: hadi.parandehafshar@epfl.ch

Education

Ph.D. in Computer Science **Expected:** March 2012	Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland *Advisor*: Prof. Paolo Ienne *Thesis*: Closing the Gap Between FPGA and ASIC: Balancing Flexibility and Efficiency
M.S. in Computer Architecture **Degree:** September 2003	University of Tehran, Tehran, Iran *Advisor*: Prof. Zain Navabi *Thesis*: Formal verification of digital circuits with word-level structures
B.S. in Computer Engineering **Degree:** June 2001	University of Tehran, Tehran, Iran *Final project*: Design and Implementing of MIPS RISC processor

Honors and Awards

2012 Best Paper Award in FPGA12 (Symposium on Field-Programmable Gate Arrays), the top most conference in FPGA research area.
2011 Invited Paper to ASILOMAR11 (Conference on Signals, Systems, and Computers).
2010 HiPEAC Paper Award from European Network of Excellence on High Performance and Embedded Architecture and Compilation.
2009 Best Paper Award in FPL (Field Programmable and Logic) conference, a first tier conference in FPGA research area.
2001 Ranked 1 among 50 graduated computer engineering students.
1997 Ranked 136 in Iran National University Entrance Exam for B.Sc. degree among more than 300,000 participants.

Professional Research and Work Experiences

January 2007 – September 2007 / July 2008-Now
Place: *Processor Architecture Laboratory (LAP), Ecole Polytechnique Fédérale de Lausanne (EPFL)*
Activity: *Research and Development*

October 2004 – December 2006
Place: *Parsé Startup Company, University of Tehran Science & Technology Park, Tehran, Iran*
Activity: *Research and Development*

September 2001 – September 2004
Place: *University of Tehran, CAD Laboratory*
Activity: *Research and Development*

Presentations and Invited Talks

2012, Feb	The 20th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), Monterey, California, USA.
2012, Feb	Achronix Inc., Agatelogic Inc., and Tabula Inc., Santa Clara, USA.
2011, Sep	The 21st International Conference on Field Programmable Logic and Applications (FPL), Crete, Greece.
2011, Jun	The 20th International Workshop on Logic & Synthesis (IWLS), San Diego, California, USA.
2011, Feb	UCLA (Center for Customizable Domain-Specific Computing), Berkeley (EECS Department), Xilinx Inc.(San Jose), Mentor Graphics Inc. (Fremont), Achronix Inc. (Santa Clara).
2011, Feb	The 19th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), Monterey, California, USA.
2010, May	The 18th International IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Charlotte NC, USA.
2010, Apr	Laboratoire de l'Informatique du Parallélisme, Ecole normale supérieure (*ENS*) de *Lyon*, France.
2009, Sep	The 19th International Conference on Field Programmable Logic and Applications (FPL), Prague, Czech Republic.
2008, Mar	Design Automation & Test in Europe conference, Munich, Germany.

Teaching and Mentoring

2003-2012	**Lecturer and assistant** in several computer engineering courses such as **computer architecture**, **Logic design**, **digital electronics**, **VLSI design, Test and Testable Design** and **design for verification**.[Places: EPFL, University of Tehran and Rajaii University]
2008-2012	**Supervisor** of several students including 3 graduate and 5 undergraduate students for their master, semester and internship projects.
2007-2012	**Reviewer** in several conferences and journals such as TCAS, IET, FPL, IEEE MICRO, EuroASIP, ICECS, T-COMP and DATE.

Main Publications

Books

b1. **H. P. Afshar**, "**Design of Digital Circuits by Verilog**", *NAS publisher*, 1st Edition (Farsi), May 2004.
(This book introduces the basic concepts of Verilog HDL and is intended primarily for students and beginner designers. Several examples implemented are discussed in this book and a sample RISC processor designed and implemented by the author is included.)

Patent Applications

p1. H. P. Afshar, D. Novo, and P. Ienne: **"Non-LUT Field Programmable Gate Arrays"**, US Patent filled, Dec 2011.

p2. P. Brisk, A. Cevrero, H. P. Afshar, Frank K. Gurkaynak, and P. Ienne: **"Generalized Parallel Counter Arrays"**, US Patent, 20090216826, 2009.

Selected Journals

2011

j1. **H. P. Afshar**, A. Neogy, P. Brisk and P. Ienne, **"Compressor Tree Synthesis on Commercial High-Performance FPGAs"**, *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 4(4):39:1-39:19, December 2011.

2010

j2. **H. P. Afshar**, A. K. Verma, P. Brisk and P. Ienne, **"Improving FPGA Performance for Carry-Save Arithmetic"**, *IEEE Transactions on Very Large Scale Integration (T-VLSI) Systems*, 18(4):578-90, April 2010.

2009

j3. **H. P. Afshar**, A. K. Verma, P. Brisk and P. Ienne, **"An FPGA logic cell and carry chain configurable as a 6:2 or 7:2 compressor"**, *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2(3):19:1-19:42, September 2009.

j4. A. Cevrero, P. Athanasopoulos, **H. P. Afshar**, A. K. Verma, P. Brisk, C. Nicopoulos, H. Attarzadeh Niaki, F. K. Gurkaynak, Y. Leblebici, and P. Ienne, **"Field programmable compressor trees: Acceleration of multi-input addition on FPGAs"**, *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2(2):13:1-13:36, June 2009.

2007

j5. **H. P. Afshar**, M. Saneei, A. Afzali-Kusha and M. Pedram, **"A Fast INC-XOR Codec for Low Power Address Buses"**, *IET Computers & Digital Techniques,* 1(5), Sep. 2007, 625-626.

Selected Conferences

2012

c1. **H. P. Afshar**, D. Novo and P. Ienne, **"Rethinking FPGAs: Elude LUT Flexibility Excess with And-Inverter Cones"**, *To appear in 20th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, February 2012. **Best Paper Award.**

c2. Y. Moctar, N. George, **H. P. Afshar**, P. Ienne, G. Lemieux, P. Brisk, **"Reducing the Cost of Floating-Point Mantissa Alignment and Normalization in FPGAs"**, *To appear in 20th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, February 2012.

2011

c3. **H. P. Afshar** and Paolo Ienne, **"Measuring and Reducing the Performance Gap Between Embedded and Soft Multipliers on FPGAs"**, *Proceedings of the 21th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 242-49, Crete, Greece 2011.

c4. **H. P. Afshar**, G. Zgheib, P. Brisk and P. Ienne, **"Routing Wire Optimization through Generic Synthesis on FPGA Carry Chains"**, *Proceedings of 20th International Workshop on Logic & Synthesis (IWLS)*, June 2011.

c5. **H. P. Afshar**, G. Zgheib, P. Brisk and P. Ienne, **"Reducing the Pressure on Routing Resources of FPGAs with Generic Logic Chains"**, *Proceedings of 19th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, February 2011.

2010

c6. **H. P. Afshar**, A. Neogy, P. Brisk and P. Ienne, **"Improved synthesis of compressor trees on FPGAs by a hybrid and systematic design approach"**, *Proceedings of the 19th International Workshop on Logic and Synthesis (IWLS)*, pages 193-200, Anaheim, Calif., June 2010.

c7. **H. P. Afshar** and P. Ienne, "**Highly versatile DSP blocks for improved FPGA arithmetic performance"**, *Proceedings of the 18th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 229-36, Napa Valley, Calif., April 2010. **HiPEAC Paper Award**.

2009

c8. **H. P. Afshar**, A. Cevrero, P. Athanasopoulos, P. Brisk, Y. Leblebici and P. Ienne, **"A flexible DSP block to enhance FPGA arithmetic performance"**, *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT)*, Sydney, December 2009.

c9. **H. P. Afshar**, P. Brisk and P. Ienne, **"Exploiting fast carry-chains of FPGAs for designing compressor trees"**, *Proceedings of the 19th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 242-49, Prague, August 2009. **Best Paper Award**.

c10. **H. P. Afshar**, P. Brisk and P. Ienne, **"Scalable and low cost design approach for variable block size motion estimation (VBSME)"**, *Proceedings of the International Symposium on VLSI Design, Automation and Test*, Hsinchu, Taiwan, April 2009.

2008

c11. **H. P. Afshar**, P. Brisk and P. Ienne, **"A Novel Logic Block for Improved Arithmetic Performance"**, *Proceedings of 16th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, February 2008.

c12. A. Cevrero, P. Athanasopoulos, **H. P. Afshar**, P. Brisk and P. Ienne, **"Architectural Improvements for Field Programmable Counter Arrays: Enabling Efficient Synthesis of Fast Compressor Trees on FPGAs"**, *Proceedings of 16th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, February 2008.

c13. **H. P. Afshar**, P. Brisk and P. Ienne, **"Improving Synthesis of Compressor Trees on FPGAs via Integer Linear Programming"**, *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, March 2008.

c14. **H. P. Afshar**, P. Brisk and P. Ienne, **"Efficient Synthesis of Compressor Trees on FPGAs"**, *Asia South Pacific Design Automation Conference (ASPDAC)*, January 2008.

2007

c15. P. Brisk, A. K. Verma, **H. P. Afshar** and P. Ienne, **"Enhancing FPGA performance for arithmetic circuit"**, *Design Automation Conference (DAC)*, pp. 334-337, June 4-8, 2007.

2006

c16. **H. P. Afshar**, A. Tootoonchina, M. Yousefpour, O. Fatemi and M. Hashemi, **"A Slice-Based Automatic Hardware/Software Partitioning Heuristic"**, *18th International Conference on Microelectronic (ICM),* pp. 150-153, December 2006.