Template-Based Embedded Reconfigurable Computing

Katarzyna Leijten-Nowak

The front cover: Artist impression of a schematic diagram of a logic tile transforming into a layout of an embedded FPGA chip. The logic tile is part of the embedded FPGA architecture template proposed in this thesis. The embedded FPGA chip was designed in accordance with this template.
The back cover: Subsequent steps of the template-based design methodology: the high-level specification of the embedded FPGA core, the schematic of the core, the VLSI layout of the core, the fabricated embedded FPGA chip.

Template-Based Embedded Reconfigurable Computing

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven,
op gezag van de Rector Magnificus, prof.dr. R.A. van Santen,
voor een commissie aangewezen door het College voor Promoties
in het openbaar te verdedigen
op vrijdag 9 juli 2004 om 16.00 uur

door

Katarzyna Leijten-Nowak

geboren te Wrocław, Polen

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. J.L. van Meerbergen
en
prof.dr.ir. R.H.J.M. Otten

Copromotor:

prof.dr.ir. P.R. Groeneveld

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Leijten-Nowak, Katarzyna

Template-based embedded reconfigurable computing / by Katarzyna Leijten-Nowak. - Eindhoven : Technische Universiteit Eindhoven, 2004.
Proefschrift. - ISBN 90-386-1583-3
NUR 959
Trefw.: programmeerbare logische schakelingen / logische schakelingen ; ontwerp / ingebedde systemen / grote geintegreerde schakelingen ; CAD / CMOS-schakelingen.
Subject headings: programmable logic arrays / logic design / system-on-chip / reconfigurable architectures / CMOS logic circuits.

The work described in this thesis has been carried out at the Philips Research Laboratories in Eindhoven, The Netherlands, as part of the Philips Research programme.

in loving memory of my mother

We shall not cease from exploration
And at the end of all our exploring
Will be to arrive where we started
And know the place for the first time.
T.S. Elliot

PREFACE

This Ph.D. thesis completes the research I was conducting in the period February 1999–April 2003 in the Embedded Systems Architectures on Silicon Group at Philips Research Laboratories in Eindhoven, The Netherlands. My research was done as part of the ARCADE (Applications of ReConfigurable Computing Architectures in Dsp Environment) cluster project, I happened to be the only member of.

I am extremely happy for having received the opportunity to pursue a Ph.D., and maybe even more happy for having it done in the industrial environment. Though, as a Ph.D. student I was not exposed to all industrial problems, I always enjoyed observing trends and the impact they have on our environment. This always puts research objectives in perspective.

I owe thanks to many more people than the limited space on these pages allow me to mention. Still, I would like to thank a few of them especially.

Foremost, I would like to direct my special thanks to Jef van Meerbergen for helping me to understand what research really is about, for his involvement in my work and for support every time I asked for it. I also would like to thank prof. Ralph Otten for his valuable comments on my work and for helping me to shape this thesis to what it has become today. Finally, I want to thank prof. Jochen Jess for encouraging me to aim for a Ph.D. in his group and for his interest in my work.

There are two more people I am especially grateful to. First, I want to thank Harry Veendrick for teaching me the secrets of correct-by-construction digital design and for our cooperation which, to a large degree, impacted my research. Second, I want to thank Bernardo Kastrup for making me believe in reconfigurable computing.

I thank Atul Katoch for his help in designing the ARCADE eFPGA chip during the long evenings of winter 2001, and Peter Poplavko, Alexander Danilin, Peter van de Haar and Frank Linssen for helping to customise the eFPGA mapping flow.

I thank my group leaders at Philips Research (in chronological order): Engel Roza, Rob Woudsma, Albert van der Werf and Ad ten Berg for supporting my research. Also, I thank the members of the research groups, I was and still am a member of, at Philips and at the Eindhoven University of Technology.

Finally, I would like to thank my family for the understanding of my wish of pursuing a scientific ‘career’ and for their great support and encouragement. I owe special thanks to my mother for her belief that I would once reach this stage. I wish she could know she was right. I dedicate this thesis to her.

I also thank my parents-in-law for meaning to me much more than I ever expected and for making me feel in the Netherlands as in my home country.

Last, but definitely not least, I want to thank my husband, Jeroen, for his true love, support, understanding...and for his patience through so many years. Yes, it is time to pursue different types of challenges now...

Hulsel, May 2004 Kasia Leijten.

Contents

Chapter 1

INTRODUCTION

We are witnessing today the third phase of computing. To clearly differentiate from the mainframe-oriented and PC-oriented computing of the first two phases, the computing of the third phase is often defined as the *post-PC* or *'post.com' computing* [35]. The post-PC computing, featuring applications such as ubiquitous communication and ambient intelligence, has been made possible thanks to advances in integrated circuit (IC) technology within the last few decades. During this time, a positive-feedback-like dependence between computing systems and semiconductor technology has appeared [88]. The ever-continuing CMOS scaling allows the number of transistors per unit area to be doubled every process generation [56]. At the same time, the increased transistor density enables designing more complex yet more demanding systems which, in turn, challenge silicon technology.

The continuous demand for performance and bandwidth has been, and still is, the main driving force behind such technological evolution [28]. However, while silicon area has been the predominant cost function of computing up till now, today power consumption is becoming the main limiting factor [28]. This is because power dissipated in integrated circuits increases with the increase of the IC complexity (the increase of the total switching capacitance), an operating frequency [108] and a bit-rate [83]. Furthermore, a low cost and a small form factor are the key requirements for ICs targeting ubiquitous terminals and ambient devices. According to today's predictions, within a five year scale the price for a 2 mm^2 IC for this type of applications (excluding the battery and package costs) will be about 20 dollar cents [16]. To reach such a cost level, inexpensive chip packages and miniature smart batteries as the main power source will have to be used. Unfortunately, standard chip packages have strictly limited power dissipation capabilities today and the battery life-time is measured in months rather than years.

As the complexity of modern IC increases, so do their design and manufacturing costs. The non-recurring engineering costs (NRE) are the subject of a particularly dramatic growth. For example, the cost of the mask set for a single chip exceeds half a million dollars today [124]. This puts an additional constraint on the design of cost-sensitive consumer electronics products. At the same time, the market-dependent economy enforces very short product design cycles and the frequent change of the product features. In consequence, intrinsic flexibility as a means of reducing design and production costs, but also as a way of extending the product life-time, is of key importance today [39, 62, 88, 20]. We define *flexibility* as a capability of an electronic device to change its function after the device is fabricated. We also use the term '*efficiency*' to describe the degree to which an electronic device uses its resources to realise its function [59].

In the light of the above-presented facts, finding a balanced trade-off between flexibility and efficiency in computing systems, and in particular in embedded systems on silicon, is crucial today. The mature semiconductor technology and rich computing 'know-how' open new possibilities of successfully bridging the cost-efficiency gap induced by traditional design approaches. In this thesis, we focus on one of such promising paradigms, namely *reconfigurable computing*, and apply it to the design of embedded systems on silicon.

1.1 Trends in IC technology and applications

The technology, on the one hand, and applications, on the second hand, define the context. In this section, we survey current trends that can be observed in the IC technology and its applications.

1.1.1 Technology scaling

The idea of *scaling* has been the primary enabler of an exponential growth in semiconductor technology. The scaling principles [40, 10] describe the pace, expressed in a scaling factor α, at which geometrical parameters of MOS devices, interconnect wires and the supply voltage value scale when moving to smaller process technologies. The actual scaling trend observed throughout the years follows Gordon Moore's predictions from 1975 [74] (the so-called *Moore's law*). The scaling factor $\alpha = 0.7$ is assumed today to characterise this trend. Assuming the fixed complexity of an IC, the scaling translates into the following effects [108] that accompany each new process generation (about every 18 months [57]):

- the reduction in the silicon area by a factor of 2 (α^2),
- the increase of the chip performance by roughly a factor of 1.5 (α^{-1}),
- the reduction of power consumption by a factor of 2 (α^2).

The key benefit of the scaling phenomenon is thus the reduction of cost and increase in performance.

1.1.2 System-on-a-chip

The dramatically progressing silicon scaling has enabled *ultra-large scale integration*. As a result, various system components, such as memories, logic, RF modules and sensors, which were traditionally integrated on a printed circuit board (PCB), could be integrated on a single chip using silicon as an implementation medium. To express this level of integration, the term *'system-on-a-chip' (SoC)* has been introduced.

The SoC market has experienced a steady and consistent growth over the last few years. One estimates that today 50% of all ASICs are produced based on the SoC concept, and this percentage is expected to grow to 80% in 2005 [89]. This can be explained by the advantages of SoCs, such as a greater integration of components, an increased speed of the communication between the system components, lower packaging and test costs, and improved reliability of a system.

1.1.3 Ambient computing

One of the major consequences of the ultra-large scale integration is a clear change in the use of computing. Unlike the first two *general-purpose computing* phases, the third phase features computing which is seemingly hidden in the background and thus almost invisible to a user ('People to the foreground, technology to the background'). This type of computing is referred to as *'ambient computing'*. Its main characteristics are: an embedded context, awareness of the environment, personalisation, and an adaptive and anticipatory character [27].

The ambient computing devices are distributed in the human environment (e.g. home and work) to maximise human efficiency and improve well being. The use of ubiquitous communication networks to connect such devices will be essential in the near future [84].

Ambient computing is considered as a possible future of consumer electronics that can stimulate the market growth [16]. Already today, simple electronic devices with an ambient-like character are being made. Moreover, many institutes all over the world fund research projects aiming at the investigation of new technologies for ambient computing. It has been declared that about four billion dollars will be spent on such activities over the next several years [27].

1.1.4 Embedded systems on silicon

The emerging applications, and ambient computing applications in particular, are realised in the form of embedded systems. An *embedded system* is a system designed to perform a dedicated or a narrow range of functions as a part of a larger system, usually with a minimal end-user interaction. Because of the tight cost constraints (in terms of size, power, performance, unit price, time-to-market, etc.), today's embedded systems heavily exploit the system-on-a-chip design concept allowing all system components to be integrated on a single piece of silicon. Therefore, in this thesis we will restrict ourselves to embedded systems implemented as systems-on-a-chip.

Unlike other, general-purpose computing systems, embedded systems are application-domain-specific. Furthermore, their behaviour is defined by the interaction with the environment [46]. Due to real-time constrains imposed by the environment, signal processing, which deals with a transformation of signals from the environment (mostly in a digital way), plays an essential role. Therefore, embedded applications impose very high requirements on the overall system performance. For example, the computational load of an ambient intelligence system ranges from 10 MOPS (Mega-Operations-Per-Second) for lightweight audio processing to 1 TOPS (Tera-Operations-Per-Second) for synthetic video generation [16].

1.2 Design challenges

A design is a result of the search for a satisfactory match between a target application and an available technology. However, the design is, to a large extent, also constrained by economic and other factors [20]. To explain this, we describe the main challenges IC designers face today.

1.2.1 Silicon economics

The cost aspect has always been the main consideration in SoC design [56]. Today, the exploding complexity of process technology has reached the point where design and chip fabrication costs impose stronger limitations on the performance and function of produced systems-on-a-chip than physics itself [12, 45] (see Figure 1.1). The NRE (Non-Recurring Engineering) costs related to each design, and mask set costs in particular, are of especially great concern [112]. The cost of a complete photo-mask set in a state-of-the-art 0.13 μm process is about 0.75 million dollars today. Due to the continuously raising complexity of sub-length lithography [20], which will be even more significant in nanometre process technologies, this cost will reach a level of almost two million dollars in two process generations from now (that is, for the 65 nm technology node) [124]. This trend is accompanied by sky-rocketing chip fab costs. A basic plant for 0.18 μm production costed four billion dollars ($4 billion) and, according to what is known as Rock's law, the cost of equipment to produce ICs will be doubling every four years [58].

Already today such quickly rising costs have a strong impact on the chip production, especially for low-volume markets. A much higher volume or a much higher cost per unit are required to amortise the expenses. It is predicted that this trend will lead eventually to a considerable decrease in the overall number of chip designs (see Figure 1.2) since only a very few companies will be able to afford them.

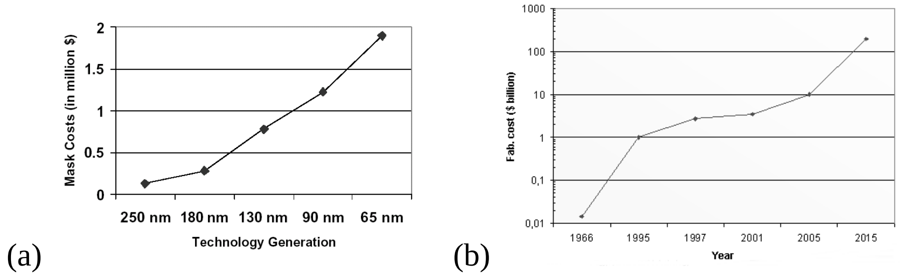


Figure 1.1. The increasing IC costs: (a) a mask set costs increase (Source: Zuchowski et al., ICCAD 2002 [124]), (b) a fabs costs increase (Source: Kellog TechVenture, 2001).

Finally, there is an increasing effect of logistic costs. Such costs are associated with managing a large variety of different products or product families, marketing, management, supply chains, etc.

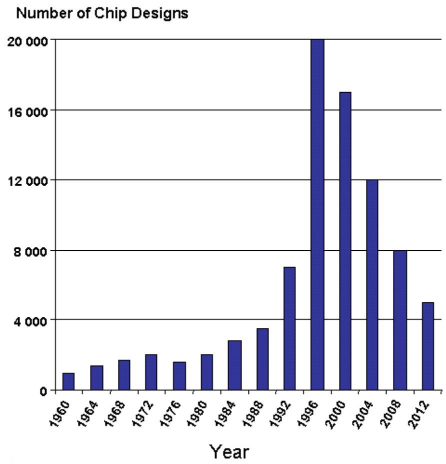


Figure 1.2. The predicted decrease in the number of chip designs (Source: Gartner Dataquest, November 2001).

1.2.2 Design productivity gap

The shift towards more and more advanced process technologies is accompanied not only by the design cost increase but also by a dramatic increase of the design complexity. Firstly, the complexity of components being designed, and consequently the system complexity (in terms of the number of transistors and complexity of embedded software) are growing exponentially [56]. Secondly, due to more devices available on the same die, but predominantly due to deep-submicron effects [108], the technology in which it is being designed is becoming more complex. Finally, the tools that are used to design today's systems are becoming more and more sophisticated in their usability and are characterised by increased runtimes. For example, the verification phase of a design process allocates today more than 50% of the total human and computer resources [56].

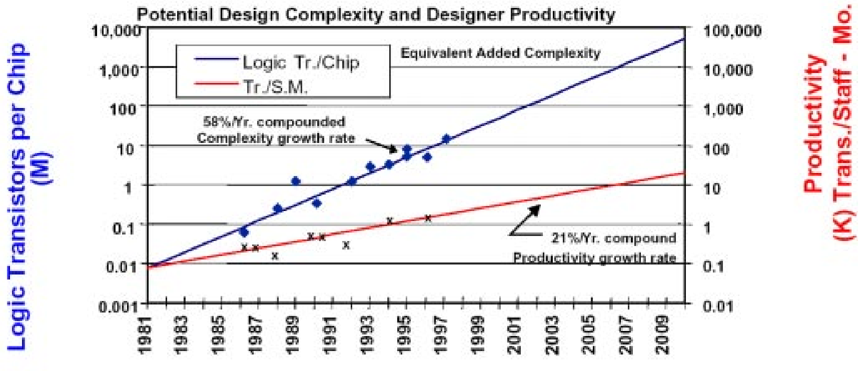


Figure 1.3. The design productivity gap being a result of a disproportion between the complexity of silicon and the design complexity which can be handle (Source: ITRS 1999 [56]).

Figure 1.3 illustrates a big gap between design complexity and design productivity which is a direct consequence of the above-mentioned trends. The complexity of designs which can be realised in a state-of-the-art technology grows 58% per year, while the designer's productivity grows at a pace of only 21% per year.

1.2.3 Time-to-market versus time-in-market

The increasing complexity of integrated circuits has led to a considerable increase of the design time that is needed before the release of a new generation of products. This time is often referred to as '*time-to-market*'. At the same time, the constant availability of more and more sophisticated devices on the market has increased the customers' expectations towards the functionality and the frequency of appearance of new products. This caused a steady reduction of the product lifetime, which is often called '*time-in-market*', resulting eventually in a collision of both, as illustrated in Figure 1.4.

It is not uncommon today that for certain classes of products the time-to-market exceeds the time-in-market. The unpredictable behaviour and the competitive character of the market imply that only economically-healthy companies, which offer products with a clear advantage over the competitors, can survive.

1.2.4 Computational efficiency gap

Because of the key importance of the low-power capabilities, modern ICs are designed for the largest *computational efficiency* expressed in MOPS per Watt, that

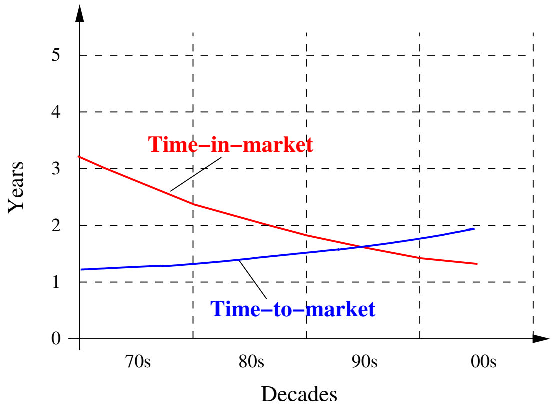


Figure 1.4. The collision of the product life-time (time-in-market) and the product development time (time-to-market) (Source: Lautzenheiser & Wersall, November 1999 [66]).

is Mega(million)-Operations-Per-Second-per Watt, rather than for a highest performance only. In Figure 1.5, a comparison between intrinsic computational efficiency of silicon (ICE) and computational efficiency of programmable processors projected onto different technology nodes is shown [88]. The comparison is based on the hypothetical assumption that a full match between the application and the architecture, and thus the maximum performance in operations-per-second, is achieved. A 32-bit addition is used as a basic benchmark function.

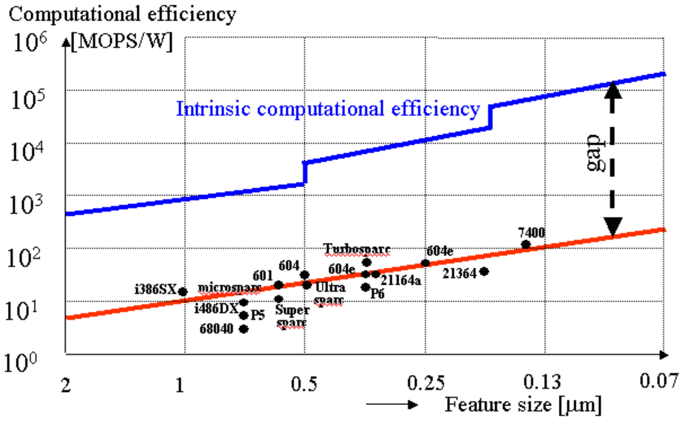


Figure 1.5. Computational efficiency of silicon and programmable processors versus technology (Source: Roza, December 2001 [88]).

It is apparent from the figure that there is a two-to-three order of magnitude *gap* between the available (silicon) and achievable (processors) computational efficiency. The gap represents a mismatch between pure-hardware and pure-software implementations that are typical for ASICs and programmable processors, respectively.

1.3 SoC architecture

The realisation of a given computing task (application) involves various trade-offs. The decisions that are made by a designer based on the essential design criteria determine the effectiveness and cost-efficiency of the final IC [59]. In the system-on-a-chip design, different, more complex, design metrics and design constraints are used than in traditional hardware or software implementations. In this section we focus on the trade-offs that are typical for the implementation phase of the design process and we discuss their implications on the SoC architecture.

1.3.1 Implementation trade-offs

The large disproportion in the computational efficiency of ASICs and programmable processors, which has been shown in Figure 1.5, suggests that there exists a spectrum of possible implementation scenarios that can be deployed to meet the required *flexibility versus cost trade-off*. For example, computational efficency of programmable processors can be increased if processors are optimised towards a target application domain. The potential implementation options, in the order of their increasing cost-efficiency, include then (but are not limited to):

- *CPUs (Central Processing Units)* – general-purpose processing units which are characterised by a sequential type of processing and a simple instruction set. CPUs are used typically for control tasks or for handling events that determine the mode or the configuration of the system.

- *DSPs (Digital Signal Processors)* - processors which are optimised for processing digital signals. DSPs include dedicated hardware components and exploit instruction level parallelism. They are typically used for the applications requiring a medium throughput and the sampling frequencies in the range of kHz (e.g. audio, speech).

- *ASIPs (Application-Specific Instruction-set Processors)* – domain-specific processors which are characterised by a well-defined intruction set that is tuned to the execution of critical parts of the application code. ASIPs target high-throughput processing with the sampling frequencies in the range of MHz (e.g. video).

The above-described implementation approaches are compared in Figure 1.6 [81]. Each implementation type is characterised by its computational efficiency (in MOPS/mW) and by its flexibility which is regarded here as the range of applications the implementation can cover.

1.3.2 Heterogenous SoC

Despite a large spectrum of possible implementation approaches, none of the presented scenarios fully meets requirements of today’s applications. This is because

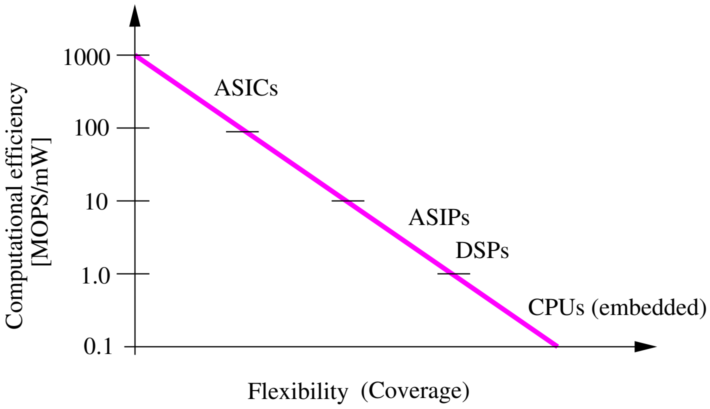


Figure 1.6. Spectrum of implementation approaches with their cost-efficiency. The cost-efficiency is determined by computational efficiency and flexibility.

a homogenous type of architecture is not suited to efficiently deal with numerous computational models and various data and time granularity that such applications require [81]. Therefore, a today's system-on-a-chip is heterogeneous and contains components such as: general-purpose processor cores, DSP cores, ASIP cores, coprocessors, peripherals, buses and network protocols, as well as memories. The size of a typical IC and the area of a single processor core, which in nanometre process technologies usually does not exceed 1-2 mm^2, suggest that a complex SoC may contain between 50 to 100 such cores.

1.3.3 Platform-based design

To guarantee the required level of computational efficiency while avoiding a disproportionate design effort due to the growing complexity and diversity of designs, the concept of system platform has been introduced [44, 62, 20]. The *system platform* is defined as a family of hardware and software modules, configured in a prescribed communication structure, that can be shared across multiple applications from a target application domain, and to which a unified design process can be applied. We refer to the hardware component of a system platform as a *silicon system platform* [28, 88]. Typically, the architecture of a silicon system platform is described by a means of a generic model called *template*. The template instantiation means in this context a process of the creation of application-domain-specific designs. The designs differ only in the number and kinds of modules integrated together. By the integration of programmable cores, which are preferred from the flexibility point of view (i.e. software mapping), and dedicated coprocessor cores that implement specialised functions, a desired trade-off between flexibility and cost can be achieved. A typical example of a silicon system platform is depicted in Figure 1.7.

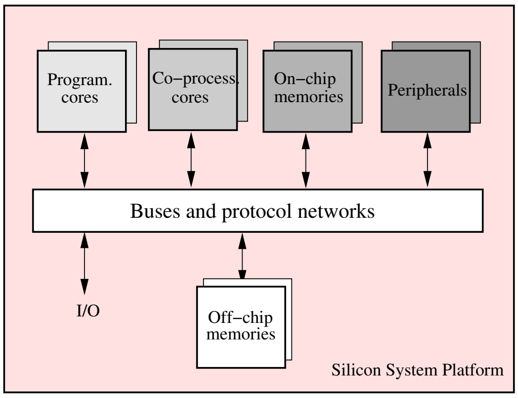


Figure 1.7. The architecture of a typical silicon system platform.

1.4 Embedded reconfigurable computing

Although the concept of the system platform represents a significant step towards designing cost-efficient embedded systems-on-a-chip, it is not the ultimate solution [48]. The system platforms reduce the complexity of the design process through the extensive *reuse* of hardware and software components, providing thus a faster time-to-market. The same reduction in the design complexity is obtained, however, at the price of reduced flexibility (silicon platforms with dedicated hardwired co-processors can cover only a specific range of applications) and reduced performance (the need for flexibility favours the use of computationally less efficient programmable cores). Finally, the realisation of a specific instance of a silicon platform is always associated with the fabrication of a new chip. This takes time and money.

These problems can be resolved if the *reconfigurable computing (RC)* paradigm is applied to the design of embedded systems. To differentiate from reconfigurable computing in a traditional sense, that is, concerning stand-alone devices (such as FPGAs; see below), we will refer to reconfigurable computing in the embedded context as *embedded reconfigurable computing (eRC)*. In this section, we explain what reconfigurable computing is, we motivate its importance for embedded systems-on-a-chip, and we survey state-of-the-art systems employing eRC paradigm. Finally, we discuss the key challenges in the design of embedded RC hardware.

1.4.1 Reconfigurable computing up close

Reconfigurable computing is a method of performing computations using reconfigurable computing devices [37]. Because programmable and reconfigurable computing architectures can be viewed as two extremes in the unified design space [37], a clear definition of a reconfigurable computing device is difficult. Nevertheless, throughout this thesis we assume that a *reconfigurable computing device* is characterised by a *configuration*, which defines the device functionality and cannot be changed from cycle to cycle, and by *flexible interconnect* that determines the data-flow between operations [37].

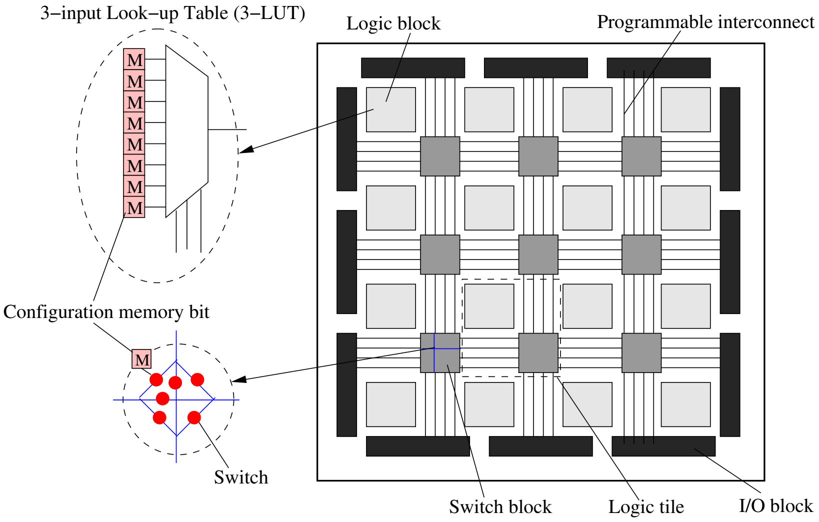


Figure 1.8. An example FPGA architecture.

A typical reconfigurable computing device is a *Field Programmable Gate Array (FPGA)* (Xilinx, 1985 [24]). The general structure of a classical FPGA is shown in Figure 1.8. An FPGA is built as an array of computing elements called logic blocks. A look-up table (LUT) is commonly used as the basic computing element of a logic block. The logic blocks are connected via a programmable interconnect network and are surrounded by input-output (I/O) blocks. The programmable interconnect network consists of horizontal and vertical routing channels. Each routing channel includes segmented routing tracks of a different length. This allows performance-efficient communication at different distances. The interconnect segments are connected via programmable switches located in switch blocks. In FPGAs, both the functionality of logic blocks and connections between them are programmable. Programming is done by loading configuration data to a configuration (control) memory. Such a memory controls LUT memory bits and multiplexers and switches of the interconnect network.

Reconfigurable devices versus ASICs and programmable processors

The key advantage of a reconfigurable computing device is the trade-off between flexibility and cost such a device offers. The trade-off is reflected in the organisation of computations and in the functionality binding time [39].

The *organisation of computations* (or the way of implementing computations) relates to the concepts of *computing-in-space* and *computing-in-time* that describe ASIC and programmable processor implementations, respectively. Similarly to ASICs, FPGAs implement computations using *spatially distributed* basic computing elements. When combined together, such elements realise complex operations.

This is in contrast to programmable processors in which computing elements are coarser and are *distributed in time*. This means that the availability of such elements is dependent on the supplied instruction on a cycle-by-cycle basis. The important advantages of computing-in-space compared to computing-in-time are lower power consumption and higher performance. This is achieved by matching the word-size of computing elements to the word-size of arguments of operations that are to be executed, and by exploiting lateral parallelism. The result is, however, a more restricted domain of computations that may be executed. Also, the final design is characterised by a larger area, longer delays and higher power dissipation than traditional ASIC designs.

The second aspect, the *functionality binding time*, relates to the technology. It determines the moment of assigning functionality (defining the computations) to the underlying hardware. In this respect, reconfigurable computing devices and programmable processors fall into the same category. In both, the functionality is defined after the fabrication of the device. Furthermore, they both use the notion of software (in the form of a configuration context or an instruction) to define the functionality. But, while the time needed to change the configuration in a traditional reconfigurable computing device is in the order of microseconds to milliseconds, the change of instruction in a programmable processor takes usually only a single cycle. ASICs, unlike reconfigurable and programmable devices, are characterised by the pre-fabrication functionality binding time. Dependent on the type of ASIC, this time may be associated with the moment as early as creating the first mask of a design (custom and semi-custom ASICs) or as late as creating the last (metal) mask(s) (gate arrays and structured ASICs [121, 41]). Furthermore, hardware rather than software is used for the device customisation. The non-permanent, post-fabrication customisation of reconfigurable devices and programmable processors implies flexibility. Therefore, the same piece of silicon may be reused to implement different applications. However, such flexibility is obtained at the cost of larger area, lower performance and higher power consumption compared to ASICs. In programmable processors, the high intrinsic cost is due to the control overhead and the fixed width of the data-path. In reconfigurable logic, the reason for the high intrinsic cost are a large configuration memory, a large number of programmable switches and a rich interconnect structure.

Spectrum of reconfigurable computing architectures

Reconfigurable computing devices can be classified by two main parameters, namely the granularity of computations and a reconfiguration model. The *granularity* expresses the level of complexity of functional primitives (usually defined by the word-size of their arguments) that can be executed in computing elements of such devices. The granularity of reconfigurable devices is thus analogous to the data-path width in programmable processors [39]. Reconfigurability at different levels of granularity is exploited. For example:

- *bit-level granularity* which is characteristic for devices that control and process single-bit-wide or a-few-bit-wide data (e.g. traditional FPGAs),
- *word-level granularity* which is typical for devices with coarse computing elements (e.g. ALUs, multipliers, adders) and reconfigurability at the level of words,
- *task-level granularity* which is typical for architectures in which complete tasks rather than simple operations are reconfigured and switched between.

Often, the bit-level reconfigurability is referred to as *fine-grain reconfigurability*, and the word-level and task-level reconfigurability together as *coarse-grain reconfigurability*. The lower the level of granularity, the larger is the configuration memory and its overhead, and thus the higher is the intrinsic cost of a reconfigurable computing device. On the other hand, the higher the level of granularity, the more complex operations can be directly executed at the expense of lower flexibility. This trade-off is illustrated in Figure 1.9.

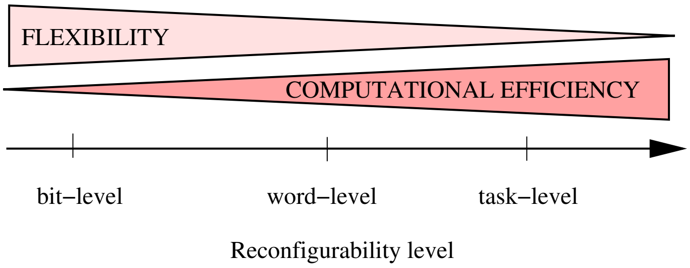


Figure 1.9. The impact of granularity on the intrinsic cost and flexibility of reconfigurable computing devices.

The *reconfiguration model* defines the frequency of the reconfiguration process and the smallest amount of resources that can be reconfigured at once. Using these parameters as basic characteristics, we can classify reconfigurable computing devices into the following categories:

- *Statically-reconfigurable* devices, the configuration of which can be changed only before the operation of a device, and *dynamically reconfigurable* devices in which the change of the configuration during the device operation is possible. Statically-reconfigurable devices have the single-context configuration memory, while dynamically reconfigurable devices have typically a multi-context configuration memory. The multi-context memory allows a fast context switching. This feature is often utilised for the implementation of the so-called run-time reconfiguration. *Run-time reconfigurable* devices execute computing tasks in phases, with each phase defined by a single configuration context.

- *Fully reconfigurable* devices in which even a minor change in the configuration requires the complete reloading of the entire configuration context, and *partially reconfigurable* devices in which only a portion of the configuration context can be changed with no influence on the rest of the configuration.

The ever-progressing technology scaling has enabled the realisation of computing architectures that offer different cost-efficiency trade-offs. Therefore today, the spectrum of (reconfigurable) computing architectures is the continuum of different computing approaches, and the choice of a particular approach is dictated by design criteria.

1.4.2 Focus and motivation

In this thesis we restrict ourselves to fine-grain reconfigurable computing only. Consequently, we use the term *'embedded reconfigurable logic' (eRL)* or simply *'reconfigurable logic'* to describe the technology that supports this type of computing and allows the on-chip integration. Note, that reconfigurable logic can be considered as a design (implementation) style that co-exists with other design styles, such as standard cells and gate arrays, for example. In this respect, embedded reconfigurable logic can be viewed as an alternative to the hard-wired logic offered by the ASIC technology. This means that from the design point of view, logic blocks of an embedded reconfigurable logic core (an *embedded FPGA*) play a similar role as logic cells of a standard cell library. Also, both in the ASIC and eFPGA technology connections between logic elements are customised based on the requirements of the mapped function. The structure and functionality of the logic elements (logic blocks) and the rules governing communication between them define the *implementation architecture*.

The key benefit of applying embedded reconfigurable computing to the design of embedded systems-on-a-chip is the increase of flexibility and computational efficiency of the final silicon product. Thus, in the context of the platform-based design embedded reconfigurable logic may be seen as an extension or a supplement of the spectrum of traditional implementation approaches which are available in the form of hard-wired and programmable cores. Figure 1.10 illustrates two potential application scenarios and the consequence of their use. The scenarios assume the use of embedded reconfigurable logic as a replacement for hard-wired logic (the *ASIC context*) and as the way of augmenting the efficiency of programmable cores (the *programmable processor context*). In the first scenario, the emphasis is on the increase of flexibility at the cost of slightly reduced computational efficiency (and the area increase) compared to a reference ASIC implementation. In the second scenario, the main motivation is the increase of computational efficiency at the cost of some increase in silicon area (not shown in the figure).

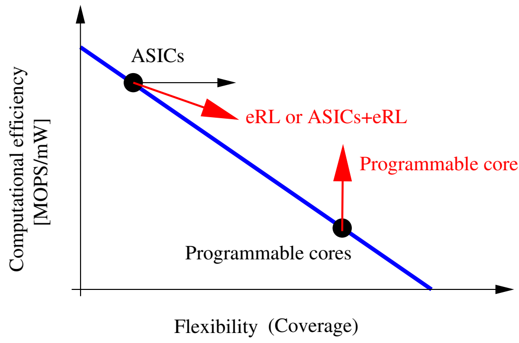


Figure 1.10. The motivation for using embedded reconfigurable logic for embedded SoC designs. The bold and thin arrows represent real and ideal (no cost penalty) cases, respectively.

The main benefits of embedded reconfigurable logic include thus:

- *Cost reduction*: the opportunity of reusing the same piece of silicon for diverse types of functionality allows sharing the mask set costs between different products, and thus the reduction of NRE costs. By augmenting flexibility, logistic costs that are associated with the maintenance and support of large families of slightly different customised products can also be decreased. Finally, by allowing the implementation of test, debug and repair structures in reconfigurable logic, the number of erroneous products can be decreased.

- *Shorter time-to-market*: a high degree of reuse and direct availability of silicon enable a very fast response to the customer's needs. The long design cycles are avoided and the final, competitive, product is faster on the market (even if it still has to be 'tuned' in the later phase).

- *Prolonged time-in-market*: the 'built-in' flexibility in the offered products allows the functionality updates that are needed because of the changing standards, for example. The risk associated with the market uncertainty can also be decreased.

- *Increase of efficiency (computational efficiency)*: lateral parallelism and matching granularity of operations to the arguments' size allow building of cost-efficient hardware accelerators that augment software-based implementations. The code for the acceleration can be identified on the fly.

In Figure 1.11, a reconfigurable sytem-on-a-chip is shown. Typical methods of integrating reconfigurable logic fabric with other system resources are illustrated. The methods are: a reconfigurable functional unit, a reconfigurable data-path, a reconfigurable coprocessor, a reconfigurable logic fabric. Note, that the first three methods relate to the programmable processor context, while the latter method to the ASIC context (see the discussion above).

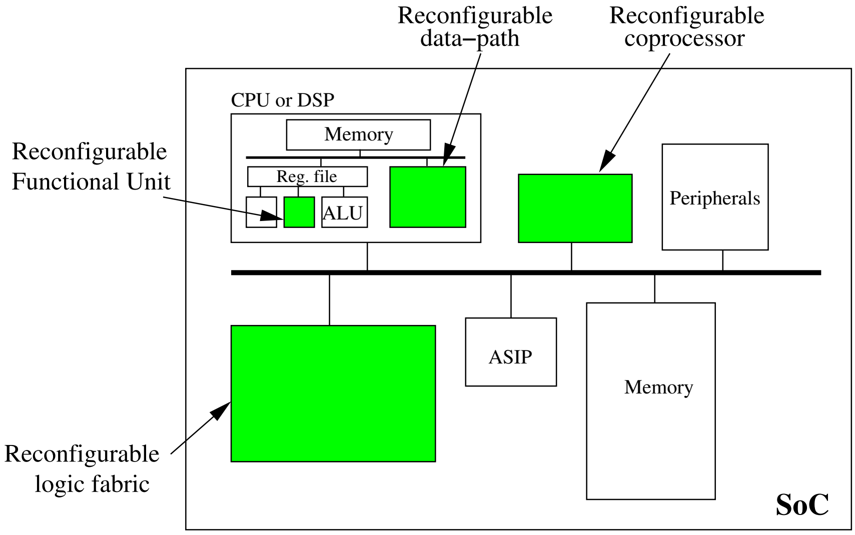

Reconfigurable data–path
Reconfigurable coprocessor
CPU or DSP
Memory
Reconfigurable Functional Unit
Reg. file
ALU
Peripherals
ASIP
Memory
Reconfigurable logic fabric
SoC

Figure 1.11. Applying reconfigurable logic in the SoC context.

The application examples of using embedded reconfigurable logic in the SoC context include (but are not limited to):

- **Hardware acceleration** *(the processor context)*. Fine-grain accelerators are used for speeding-up an irregular or a critical type of application code. Such a code may contain both logic and arithmetic operations which are inefficiently implemented using standard resources of programmable processors. Typical examples are bit-level manipulations and arithmetic operations with arguments, the width of which is smaller (or larger) than the width of the processor data-path. The identified code can be replaced by custom instructions which are implemented in reconfigurable logic [36]. Also Application-Specific Units (ASUs) are interesting candidates. Rather than implementing such ASUs in hard-wired logic (as it is done in typical ASIPs), they can be mapped onto reconfigurable logic. This broadens the application area and extends the life-time of a final product.

- **Product differentiation** *(the ASIC context)*. To meet requirements of different customers, companies providing hard programmable cores have to manage and support a large spectrum of products that only slightly differ in functionality. This is economically unjustified, even more that a new chip has to be manufactured for each such a product. By integrating some amount of reconfigurable logic with a standard core, it is possible to do per-user-customisations in an already-fabricated chip. As a result, only one version of such a chip rather than very many of them has to be supported. An example are peripherals for microcontroller cores.

- **Multi-standards realisation** *(the ASIC context)* . Today's ICs (cores or complete systems) are often designed to support different standards (e.g.

for interfaces or communication protocols). The typical realisations of such standards are independent dedicated hardware modules that are integrated on the chip. This increases the chip area and causes an overhead since only one or a very few such modules are active at the same time. On the other hand, including all possible standards on a single chip is practically impossible and economically unjustified. Due to insufficient performance, programmable processors are also not an option. For these reasons, using embedded reconfigurable logic for the implementation of various standards is an interesting alternative. Eventually, the total silicon area of the final IC may be even reduced.

1.4.3 State-of-the-art

The methods of using embedded reconfigurable logic in today's SoC products can be categorised in two dimensions. One dimension describes the *use method* of eRL, that is, whether eRL is applied to a complete hardware platform or to a component (core) of such a platform. The second dimension specifies the *scope* of eRL within that application area, that is, whether eRL constitutes the whole element or is only a part of it. Such a taxonomy is shown in Table 1.1 together with examples of state-of-the-art products belonging to each category.

Use method ⇒ **Scope** ⇓	Complete platform	Component of a platform
Full	Xilinx Virtex-II Pro Altera Excalibur Atmel FPSLIC (*platform-based FPGAs*)	Actel VariCore Leopard Logic HyperBlox Elixent RAP PACT XPP Systolix PulseDSP
Partial	Triscend A7 (*reconfigurable SoC platforms*)	Tensilica Xtensa Chameleon RCP Triscend E5

Table 1.1. Taxonomy of eRL-based SoC products.

The group of products designed as complete hardware platforms features the so-called *platform-based FPGAs*, which are offered by FPGA vendors, and *reconfigurable SoC platforms*. In the platform-based FPGAs, a traditional FPGA fabric, consisting of logic blocks and programmable interconnect, plays a central role. Such a fabric is, however, enhanced with large chunks of embedded memories (of the total capacity in the range of Mbits), dedicated functional blocks (such as hard-wired multipliers), programmable input/output blocks supporting different interface standards, peripherals, and a sophisticated clock management circuitry. Last but not least, the newest platform-based FPGAs include also single or multiple embedded CPU cores (e.g. four embedded IBM Power PCs cores in the largest Xilinx Virtex-II Pro device) that complete the whole system. Examples of the platform-based FPGAs are: Xilinx Virtex-II Pro [119], Altera Excalibur [7], and Atmel FPSLIC [9]. While embedded reconfigurable logic comprises a dominant part of the platform-based FPGAs, reconfigurable SoC platforms contain only a small amount of eRL supplementing the rest of the platform resources. Furthermore, in this case eRL is used almost exclusively for the implementation of computations. Unlike the platform-based FPGAs, which rely on the FPGA-specific technology (e.g. an increased number of metal layers, relaxed design rules [124], etc.), reconfigurable SoC platforms are typically realised in a standard CMOS process. An example is Triscend A7 platform [102].

In the group of the platform components, two types of products can be distinguished. The first type concerns cores implemented entirely in reconfigurable logic. Such cores may differ in the granularity, which typically ranges from the bit-level, through the nibble-level[1] to the word-level granularity. The bit-level reconfigurable cores are used for mapping ‘glue-logic’, control logic and sometimes also arithmetic, whereas the nibble-level and especially word-level reconfigurable cores are used predominantly as hardware accelerators. The cores from this group are offered in different shapes and with different aspect ratios. They are available as hard cores (typically bit-level and nibble-level reconfigurable cores), or soft IP cores (typically word-level reconfigurable cores). Examples are: Actel VariCore [1], Leopard Logic’s HyperBlox [67], Elixent’s Reconfigurable Array Processor [43], PACT XPP processor [76], and Systolix PulseDSP [98]. The second type of cores in the components of a platform group features programmable cores augmented with reconfigurable logic. In this case, eRL comprises only a part of the complete core and is used mainly for the acceleration purposes. Examples are: Tensilica Xtensa [101], Chameleon Systems’ Reconfigurable Communication Processor [25], and Triscend E5 [103].

1.4.4 Key challenges

Embedding reconfigurable logic onto a system-on-a-chip poses various hardware-related and software-related challenges. One of the main hardware-related challenges is the *reduction of intrinsic cost* of reconfigurable logic. A reconfigurable-logic-based implementation is costly because the high degree of flexibility it offers is obtained by the introduction of a huge number of configuration switches, multiplexers and large configuration memories. This leads to the one-to-three orders of magnitude cost penalty in area, performance and power compared to an ASIC implementation [37, 124]. The finer the reconfigurability grain, the higher is this cost (compare Figure 1.9).

Due to the high intrinsic cost, and particularly the area overhead, only a limited amount of embedded reconfigurable logic can be integrated on a system-on-a-chip. The question about the exact amount of reconfigurable logic that should be embedded is a fundamental design issue [124], and has to be resolved before a

[1] The term ’nibble’ is often used to describe groups of four bits.

system-on-a-chip is fabricated. The challenge lies in finding the *optimal size of an eRL fabric* such that it accounts for the potential (future) growth of the complexity of mapped circuits, and in a correct estimation of the eRL utilisation.

During a physical design process, a SoC designer is also confronted with the eRL challenges. For example, the metal-intensive nature of reconfigurable logic cores requires a careful chip *floorplanning*. During the floorplanning procedure, the locations of consecutive cores have to be found such that global interconnect requirements are met and there is no wiring congestion. Furthermore, proper pin positions for each core that guarantee the required timing have to be found. There is also a *testing* challenge. The specific nature of eRL asks for a robust and complex testing approach that differs from standard testing procedures.

Finally, from the software point of view, *partitioning* and *synthesis* of the application code that take into account the presence of reconfigurable logic in a system is a challenge. Even more so because usually different design flows are used for ASICs, FPGAs and programmable processors.

1.5 Problem statement

We have shown that the today's chip production depends on economic factors and the market behaviour. As a result, programmable (customisable) products offering a fast time-to-market are a preferable solution. At the same time, however, emerging applications enabled by the progress in IC technology impose higher and higher requirements on the computational efficiency of target designs. Such requirements cannot be met using traditional programmable products (e.g. general-purpose processors or DSPs).

In such a conflicting environment, there is a clear need for a new implementation paradigm which could help to bridge a cost-efficiency gap induced by traditional implementation approaches, that is, ASICs and programmable processors. The new paradigm should allow a cost-efficient design of embedded systems-on-a-chip for consumer electronics market.

Embedded reconfigurable logic appears as an interesting alternative. Nevertheless, because a low-cost production is critical for competitive consumer electronics market, the high intrinsic cost of embedded reconfigurable logic (compared to the cost of hard-wired implementations) has to be reduced to allow a full acceptance of eRL-based products.

The objective of this work is to investigate possibilities of the reduction of intrinsic cost of embedded reconfigurable logic and to develop a methodology that facilitates the design and implementation of cost-efficient embedded reconfigurable logic cores for consumer electronics applications. The cost-efficiency refers to all design levels, and covers thus architecture, implementation, and technology-related aspects.

1.6 This thesis

1.6.1 Towards a solution approach

As indicated in Section 1.4.2, in this thesis we are concerned with fine-grain embedded reconfigurable logic architectures, that is, architectures with sub-word reconfigurability. The fine-grain architectures are chosen because of their multi-purpose role in a system-on-a-chip that goes beyond the acceleration-related functionality typical for coarse-grain architectures.

In Section 1.4.4, we mentioned that application partitioning and finding the required amount of embedded reconfigurable logic are two of the key challenges of the reconfigurable SoC design. We partially resolve these problems by focusing on some typical ways of deploying embedded reconfigurable logic in the SoC context, for example such as discussed in Section 1.4.2. The consequence thereof is the assumption that only relatively small parts of the application code, and with the (to a certain extent) predictable and manageable growth of their complexity, can be mapped onto reconfigurable logic. This implies that only a relatively small amount of reconfigurable logic is required in a system-on-a-chip. Therfore, the impact the embedded reconfigurable logic has on the total cost of an embedded system can be reduced.

Finally, we propose a further reduction of the intrinsic cost of reconfigurable logic by tuning reconfigurable architectures towards requirements of target application domains. In this way, embedded reconfigurable logic is no longer general-purpose but rather application-domain-specific.

1.6.2 Main contributions

In this thesis, a template-based methodology for the design of embedded reconfigurable logic is proposed. The template describes a generic model of a reconfigurable logic architecture. The template enables a fast architecture exploration, and facilitates the VLSI implementation (netlist and layout generation) and the mapping process (architecture modelling in the CAD tools) of reconfigurable logic.

Application-domain specialisation is proposed as a means for the reduction of the intrinsic cost of reconfigurable logic. The method of characterising processing kernels from different application domains and the process of finding a suitable implementation architecture for them based on the application domain characteristics are given.

Three basic classes of domain-oriented reconfigurable logic architectures are derived. Each class is illustrated by an example of a reconfigurable logic core, which logic and interconnect architectures are described in detail. Various novel techniques at the architecture level and the application mapping level are introduced to increase cost-efficiency of such cores.

Finally, a silicon prototype of one of the cores that has been implemented according to the proposed template concept is presented.

1.6.3 Organisation of the thesis

This thesis is organised as follows. In Chapter 2, we analyse state-of-the-art FPGAs and identify the reasons for their high intrinsic cost. We also propose the concept of application domain specialisation of reconfigurable logic that aims at the reduction of such cost. The process of the application domain characterisation based on three application parameters, which capture the logic and interconnect requirements, is described. In Chapter 3, generic properties of logic and arithmetic functions are given. Two new properties of binary addition are described that reduce its implementation cost in reconfigurable logic. Cost metrics for the quantification of such cost are also described. Three domain-oriented reconfigurable logic architectures, examples of mapping basic primitives onto such architectures, and the architecture benchmarking are discussed in Chapter 4, Chapter 5 and Chapter 6. In Chapter 7, the template-based methodology that plays a central role in the design process of domain-oriented reconfigurable logic is proposed. The case study, that is the silicon prototype of a reconfigurable logic core is described in Chapter 8. The architecture of the core is based on one of the proposed domain-oriented reconfigurable logic architectures and its implementation is realised according to the template concept. Finally, the main results of the thesis are summarised in Chapter 9.

Chapter 2

APPLICATION DOMAIN SPECIALISATION

Although attractive from the design time, manufacturing costs and time-to-market perspective, the inherent flexibility of reconfigurable logic is paid in a considerable cost overhead compared to ASICs. This is the main factor that limits the use of reconfigurable logic in cost-sensitive embedded system-on-a-chip products.

To analyse this cost, a good understanding of modern reconfigurable logic devices, such as FPGAs, and their applications is needed. In this chapter, we describe successive generations of FPGA devices and discuss architectural trade-offs they exploit. We also compare the intrinsic cost of the FPGA and ASIC technologies. Finally, we propose a method for the reduction of the intrinsic cost of reconfigurable logic through application domain specialisation. To illustrate this concept, we analyse a set of kernels from different application domains and we characterise and classify them using the selected criteria. We also present architectural implications of such a classification.

2.1 Field Programmable Gate Arrays

2.1.1 An evolving FPGA

The primary applications of the first FPGAs were hardware prototyping, low-volume production series, a ‘glue-logic’ type of functionality (e.g. interfaces), and somewhat later networking. In all cases, FPGAs were used as system components integrated on a printed circuit board. However, the continuing technology scaling has made FPGA devices attractive for a much broader range of tasks than they were originally meant for. This has resulted in the expansion of the FPGA application area from the ‘glue-logic’ niche, through hardware acceleration in general-purpose and DSP computing [82, 50, 53], to the system-central position today [89, 100]. Recently, next to traditional stand-alone devices, an alternative way of using FPGAs, that is as embedded intellectual property (IP) cores, has been proposed [1, 68].

One of the consequences of such a shift was a graduate change of FPGA architectures. This can be observed following the evolution of any commercial FPGA. For our analysis we chose the family of Xilinx FPGA devices.

Figure 2.1 shows the logic block structures of five subsequent generations of Xilinx high-end FPGA devices. Despite many similarities (e.g. the use of a look-up table as a basic logic element and the presence of a flip-flop at each logic output), the presented structures also differ. The differences reflect various architectural modifications that FPGAs have gone through. Considering both the logic and interconnect parts of an FPGA architecture, such modifications include:

- *Increase in granularity*: the logic blocks of successive generations of FPGAs are made coarser and coarser (e.g. the change from a single 4-LUT in the first generation to eight 4-LUTs in the fifth generation); coarse logic blocks facilitate an implementation of large logic functions (more efficient mapping of combinational logic) and data-paths (support for multi-bit processing).

- *Improvement of the arithmetic mapping capabilities:* from the second generation onwards, FPGAs include dedicated arithmetic resources such as carry logic, carry chains and specialised arithmetic modules (e.g. hard-wired multipliers in Virtex devices). This type of modifications considerably improve the efficiency of mapping arithmetic (i.e. the number of logic blocks required per a given function) and are driven by the increasing importance of signal processing applications.

- *Introduction of the memory mapping capabilities:* embedding large memory blocks into the traditional FPGA fabric is a consequence of the increasing role of signal and image processing applications and the increasing size of FPGAs themselves. Because of the latter, larger designs can be mapped onto FPGAs. Consequently, large off-chip memories that are required for the data storage are replaced with on-chip (embedded) FPGA memories.

- *Introduction of other architectural enhancements:* further mapping efficiency improvement is achieved by providing shift-register mapping capabilities (the third generation) and a means for implementing large sum-of-product structures (the fifth generation). Also, multi-clock domains and programmable high-speed interface logic are introduced to FPGA devices.

- *Embedding of CPU cores:* FPGAs take a full advantage of technology scaling by the on-chip integration of CPU cores (the fifth generation). In this way, FPGAs are becoming independent implementation platforms that enable a realisation of complete systems.

- *Enhancement of interconnect:* the changes in the FPGA logic are accompanied by the changes in the interconnect structure. The reasons are: the performance improvement, dealing with deep-submicron effects and enabling the mapping of larger complex designs. Consequently, carry chains and direct connections, segmented and hierarchical interconnect, and active interconnect technology (fully-buffered interconnect) are introduced.

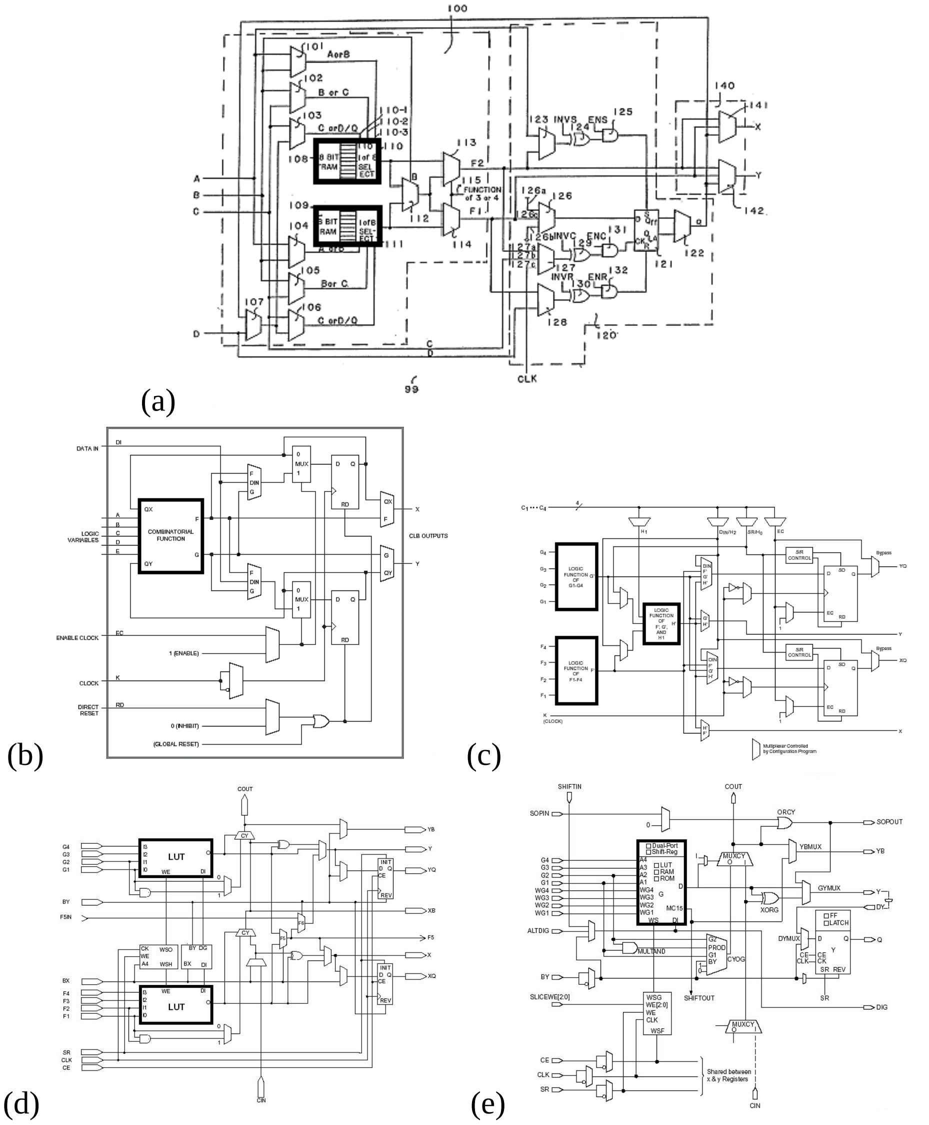


Figure 2.1. The logic blocks of five successive generations of Xilinx FPGA devices: (a) first generation XC2000, (b) second generation XC3000, (c) third generation XC4000, (d) fourth generation Virtex (the Virtex logic block includes two logic slices; the structure of a single logic slice is shown), (e) fifth generation Virtex-II (the Virtex-II logic block includes four logic slices; only a half of the logic slice is shown). The LUTs in each logic block are marked with bold rectangles.

Essential characteristics of five successive generations of Xilinx FPGAs are summarised in Table 2.1.

The above-described evolution indicates that FPGA architectures evolved from homogeneous ‘sea-of-programmable-gates’ structures almost two decades ago to *general-purpose* heterogeneous structures today. Though such an evolution has been driven by FPGAs ‘riding the wave of Moore’s law’ [17], it has a strictly

Feature	XC2000	XC3000	XC4000	Virtex	Virtex-II
General					
Introduction year	1985	1987	1991 1997 (XC4000XL)	1998 1999 (Virtex-E)	2001
Process technology	2 μm, 5 V 2-layer metal	1.2 μm, 5 V 2-layer metal 0.8/0.65/0.6 μm , 3.6/3.3 V 3-layer metal	0.8/0.65/0.6 μm, 5/3.3 V, 2-layer metal 0.5/0.35 μm, 3.3/2.5 V, 5-layer metal 0.25 μm, 2.5 V, 5-layer metal	0.22 μm, 2.5 V, 5-layer metal; 0.18 μm, 1.8 V, 6-layer metal	0.15/0.12μm, 1.5 V, 8-layer copper, low-k
System performance	?	50–70 MHz	60–80 MHz	200–240 MHz	300 MHz
Logic resources					
#in,#out/CLB	4 in, 2 out	5 in, 2 out	14 in, 6 out	22 in, 14 out	
CLB structure	2×3-LUT, 1 FF	2×4-LUT, 2FFs	2×4-LUT+3-LUT, 2 FFs	4×4-LUT, 4 FFs	8×4-LUT, 4 FFs
Distributed RAM	no	no	32 bits	64 bits	128 bits
Embedded RAM	no	no	no	4Kbit–32Kbit 16Kbit–208Kbit	72Kbit–3.375Mbit
Dedicated carry logic	no	no	yes	yes	yes
Mux chaining	no	no	no	yes	yes
Cascade chain	no	no	no	no	yes
Interconnect					
Local	?	direct	(direct)	internal, feedback, direct (all direct.)	internal, feedback, direct (all direct.)
General-purpose	singles	singles	singles, doubles, (quads)	singles, hexs	doubles, hexs
Global	longlines	longlines	longlines	longlines	longlines
Special	no	no	carry chain	2 carry chains, 3-state buses	2 carry chains, 3-state buses, cascade chain, shift chain
Mapping capabilities					
Max logic funct.	4-input	5-input	5-input	6-input	8-input
Max arithmetic funct.	1-bit	1-bit	2-bit	4-bit	8-bit

Table 2.1. Characterisation of successive generations of high-end Xilinx FPGA devices.

economical justification. Firstly, a high degree of flexibility of FPGAs allows an implementation of potentially any type of application (as long as it fits the FPGA size, and other parametric requirements, such as performance and power dissipation, are satisfied). This makes modern FPGAs competitive to traditional ASICs, especially if the production volume is low. Secondly, making FPGAs general-purpose is beneficial from the FPGA vendors' perspective. It limits namely their efforts to the support and maintenance of only a few crucial device families.

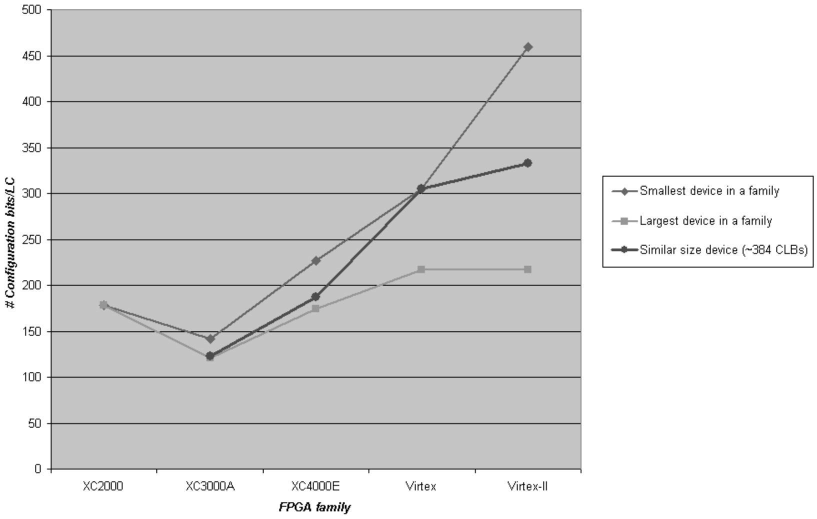


Figure 2.2. The gradual increase in the intrinsic cost of successive generations of Xilinx FPGA devices: XC2000 [24, 11], XC3000 [116], XC4000 [117], Virtex [118], Virtex-II [119]. The cost metric is the number of configuration bits per a logic cell (LC). The cost figures are calculated for the smallest and largest devices in a given family, and assuming a device of the similar capacity (i.e. ∼384 CLBs).

The clear drawback of the continuous process of augmenting flexibility of FPGA devices is the increasing cost overhead. Figure 2.2 shows the number of configuration bits per logic cell calculated for five subsequent generations of Xilinx FPGAs. We chose the number of configuration bits as the primary cost metric since it is easily available from the FPGA data sheets. We normalised this cost per logic cell. The 'logic cell' (LC) expresses a fixed amount of functionality that is equivalent to the functionality of a 4-LUT with a flip-flop[1]. The following factors (in the LC equivalents) are used to characterise configurable logic block (CLB) capacities of five generations of Xilinx FPGAs: XC2000 - 1, XC3000A - 1.625, XC4000X - 2.375, Virtex-E - 4.5, and Virtex-II - 9 [115]. (Embedded memories that are present in some of the analysed devices are excluded.)

It is clear from the figure that the cost of the similar amount of functionality in each new generation of FPGA devices is gradually increasing.

[1] The LC metric has been proposed by Xilinx to allow the comparison of different FPGAs [115].

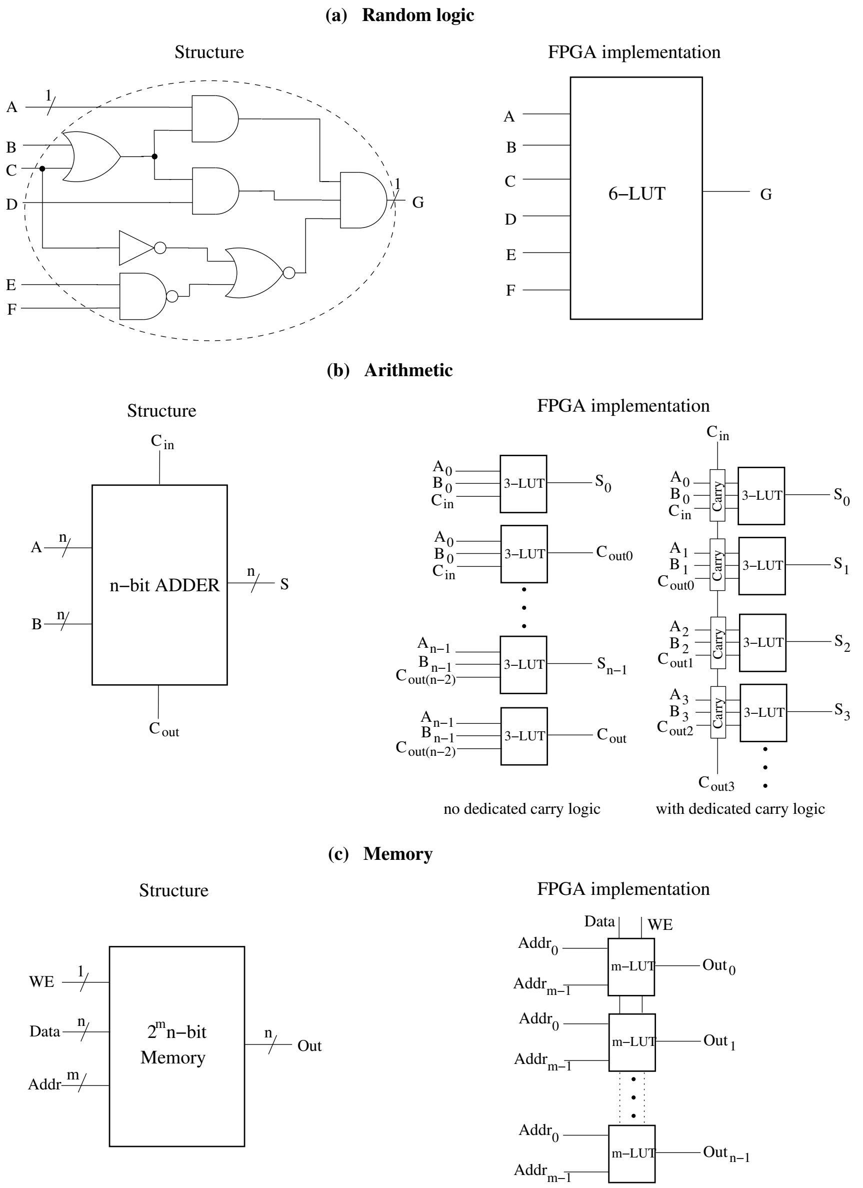
2.1.2 Architectural trade-offs in FPGAs

In its broadest sense, the general-purpose nature means the ability of implementing random logic, data-path and memory functions in a single device. The cost-efficient realisation of such functionality in a homogeneous (i.e. with an unified type of resources) LUT-based FPGA device is fundamentally difficult. This is due to different, often conflicting, requirements that are imposed by logic, arithmetic and memory functions. For example, random logic functions, though being collections of fine-grain elements (logic gates), usually benefit from the mapping onto coarse look-up tables (see Figure 2.3(a)). The coarse look-up tables allow a substantial reduction of the logic depth, and thus reduce delays. Because there is a direct correlation between the granularity of a logic block, a logic delay and the complexity of the FPGA routing resources [87], there is an upper limit on the size of a look-up table[2] that should be used in practice. Based on the study described in [87], 3-input and 4-input look-up tables (3-LUTs and 4-LUTs) have been found to be the best in terms of area. The results of the complementary study published in [94] have shown that 5-input and 6-input look-up tables (5-LUTs and 6-LUTs) are the best in terms of delay. Finally, a similar analysis presented in [3], in which much more accurate area and delay models have been applied and the clustered type of a logic block has been considered, has shown that look-up tables with four to six inputs (4-LUTs to 6-LUTs) and logic block clusters consisting of four to ten look-up tables guarantee the best area-delay product.

When mapped onto FPGAs, the data-path functions, and in particular arithmetic, reveal an opposite nature to random logic. The arithmetic functions are usually coarse-grain, this is, they have multi-bit input and output arguments. To account for the dependence of the bits of the arithmetic output on the carry signal propagation, a straightforward FPGA realisation of an arithmetic function would require very large, multi-bit-output look-up tables [47]. Because such look-up tables are cost-inefficient, serially connected small look-up tables, which allow the bit-slice structure of data-paths to be exploited, are used in practice (see Figure 2.3(b)). Often, a dedicated carry logic circuitry is also added to improve area-efficiency and performance of the arithmetic-dominated designs.

Since the complexity of designs that can be mapped onto FPGAs is constantly increasing, on-chip memories are needed. The initial assumption on the homogeneous structure of an FPGA device implies that such memories are implemented as distributed memories. Typically, distributed memories in an FPGA are realised using look-up tables (see Figure 2.3(c)). Because in a general-purpose FPGA the look-up table size is a compromise between the requirements of random logic, data-path and memory functions, the FPGA LUT-based memories are relatively small (typically, with the total capacity between 32 and 128 bits per logic block). If the mapped design requires larger memories, they can be assembled from small LUT memories of the logic blocks. To allow the memory functionality, an FPGA

[2] The size of a look-up table is regarded as the number of its inputs (see details in Section 3.1).


(a) Random logic

(b) Arithmetic

(c) Memory

Figure 2.3. Typical functions: (a) random logic, (b) arithmetic, (c) memory, and their LUT-based FPGA implementations.

logic block structure has to be enhanced with extra logic (e.g. write decoders, control circuitry).

2.1.3 Quantifying the cost

Figure 2.4 shows the cost comparison of the ASIC and FPGA technologies as presented in [124]. The comparison focuses on the characteristics of a basic computational element in each technology, that is a *logic gate* in ASICs and a *look-up table* in FPGAs. In Figure 2.4(a), the *logic density* of basic computational elements, which is calculated as the number of equivalent logic gates per unit area (in

Kgates/mm^2), is shown. In Figures 2.4(b) and 2.4(c), *delay* (in ps) and *energy* (in μW/MHz) of basic computational elements are compared. All characteristics are shown for different technology nodes.

It is clear from the figures that the intrinsic cost of FPGAs is much higher than the intrinsic cost of ASICs. First of all, to implement a similar functionality an FPGA needs on average **50 times more silicon area** than an ASIC. Furthermore, an FPGA look-up table is about **15 times slower** and consumes about **600 times more energy** than a basic ASIC gate. Though the complexity of a look-up table (and thus its logic capacity) is higher than the complexity of an ASIC gate, the delay and energy results given here are representative for FPGA products as the cost overhead of FPGA interconnect has been omitted in the comparison. Another important conclusion from Figure 2.4 is that the factor of difference in area, performance and energy between FPGAs and ASICs remains constant regardless the technology scaling. We may assume therefore that this trend will continue.

According to DeHon [37], the primary reason for the high area overhead of FPGAs compared to ASICs is the *configurable interconnect*. He estimates that such an interconnect consumes about 80–90% of the total FPGA device area. However, in DeHon's model the configurable interconnect comprises not only interconnect wires (i.e. metal tracks), but also programmable switches and multiplexers (i.e. transistors) in the connection and switch blocks. In advanced process technologies the presence of several metal layers (e.g. eight in a typical 0.13 μm FPGA process) facilitate the implementation of complex interconnect structures. Therefore, in modern FPGAs the ratio of the transistor area to the area occupied by the interconnect metal tracks is about 1:1. For that reason, the reduction of the transistor area occupied by the logic and interconnect elements in a logic tile rather than the reduction of the metal track number only is essential for the reduction of the intrinsic cost of reconfigurable logic. This is even more that the reduction of the dimensions of a logic tile results in shorter connections between logic blocks, and consequently in shorter delays and lower power consumption of the entire FPGA.

2.2 The concept of application domain specialisation

Despite the general-purpose nature and thus a large potential application area, the vast majority of FPGAs target the same group of applications [61]. Today, the typical applications are: digital communication, networking, signal and image processing, automotive and security. Furthermore, the type of computations and the accuracy required in such applications favour the word-level processing rather than the bit-level processing that FPGAs were originally designed for. The higher degree of flexibility than required [95] represents thus optimisation opportunities which can be exploited to reduce the intrinsic cost of general-purpose FPGAs.

Our solution that addresses the above-mentioned aspects is the concept of *appli-*

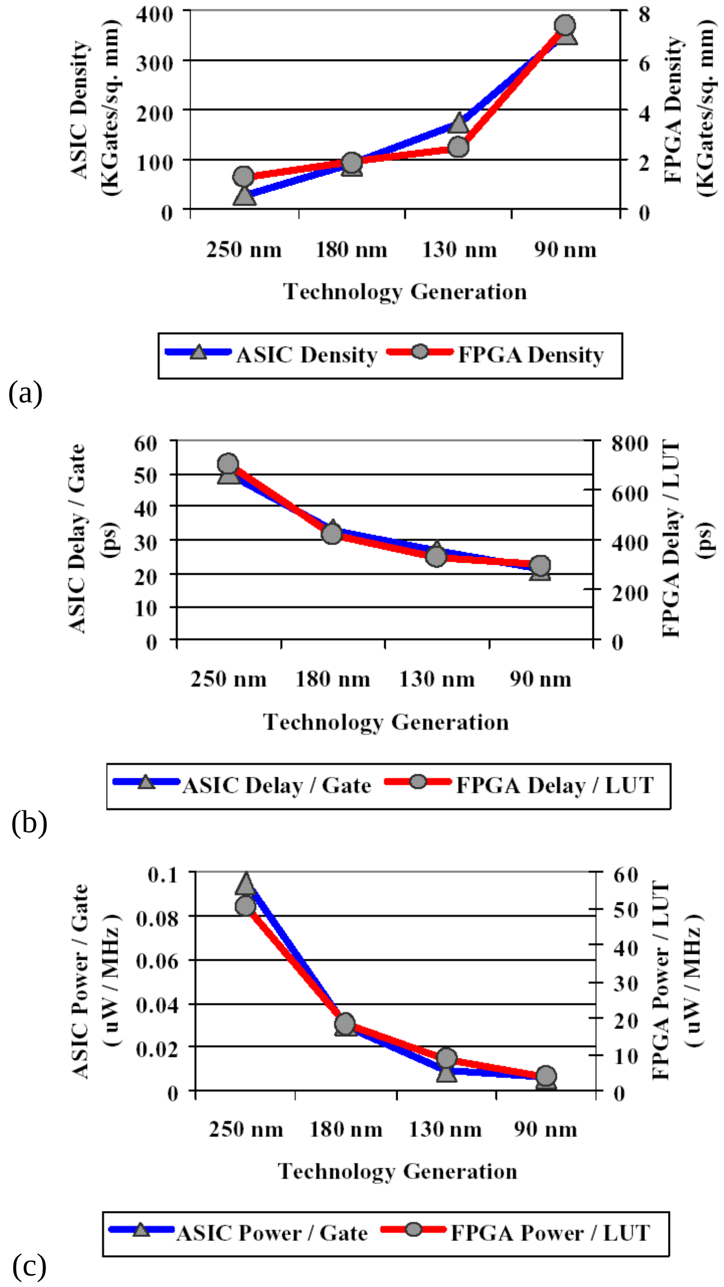


Figure 2.4. The comparison of the intrinsic cost of the ASIC and FPGA technologies. (a) Equivalent logic density (in Kgates/mm^2), (b) delay (in ps), and (c) energy (in μW/MHz) of a basic computational element are shown. The computational element is a logic gate for an ASIC and a look-up table for an FPGA. (Source: Xilinx & IBM, November 2002 [124]).

cation domain specialisation of reconfigurable logic. The key idea behind this concept is to trade flexibility of a reconfigurable logic fabric for the reduction of its intrinsic cost. The reduction of flexibility is possible because the reconfigurable logic fabric is meant to map processing kernels (tasks) from a specific application domain rather then arbitrary kernels of any application. The kernels of the application domain are assumed to share similar characteristics, which, to a large extent, are known a priori. The domain-specific optimisation of the reconfigurable logic fabric means matching its architectural solutions and their parameters with the characteristics of the target application domain. In consequence, the fabric is no longer general-purpose but *domain-specific*, and is characterised by a lower intrinsic cost. This is analogous to the concept of Application-Specific Instruction-set Processors (ASIPs) which play a crucial role in designing cost-efficient systems-on-a-chip [109].

The selection of processing kernels of a similar type and finding a suitable implementation architecture for them are essential elements of the proposed concept. One the one hand, an application domain may be characterised by more then one type of processing kernels. For example, in an MPEG4 application, which comes from the video application domain, typical kernels are Discrete Cosinus Transform (DCT), Variable Length Decoding (VLD) and Motion Compensation (MC). Such kernels are characterised by a word-level signal processing, a bit-level control processing, and a storage-dominated processing, respectively. Because the kernels have different characteristics, a design of a single cost-efficient application-domain-oriented reconfigurable logic architecture for them is difficult. On the other hand, the idea of applying application domain specialisation to reconfigurable logic may be economically unjustified if its result is a large number of different reconfigurable logic architectures that have to be supported. Therefore, it is essential that relatively broad yet well-defined *sets of kernels* across different application domains are found that share the same characteristics and imply a limited number (ideally one per set) of hardware implementations (implementation architectures). For the application domains with unified characteristics of all processing kernels, the set of kernels will be equivalent to the entire application domain. The essence of this concept is illustrated in Figure 2.5. The comparison with the implementations using general-purpose FPGAs and ASICs is also shown.

The idea of tuning reconfigurable logic to an application domain is not new. The benefit of making reconfigurable logic less general-purpose has been recognised in the past and various domain-specific reconfigurable logic architectures have been proposed in academia. Though a vast majority of such architectures target mainly DSP type of applications (e.g. [60, 2, 73]), requirements of other domains, such as cryptography, are also addressed (e.g. [49]). In parallel, the introduction of coarse-grain reconfigurable architectures has also been driven by the idea of the cost reduction in certain application areas. Examples of such architectures include: RAA architecture of Hewlett-Packard [69] and XPP processor from PAC [76].

An interesting concept of application-domain-specific reconfigurable computing

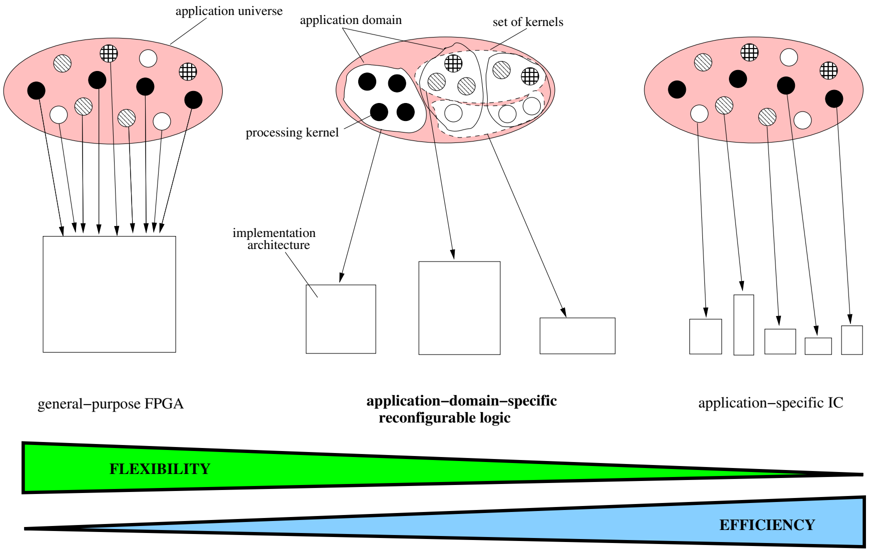


Figure 2.5. The concept of application domain specialisation of reconfigurable logic. The implementation architecture of a reconfigurable logic core is optimised towards processing kernels of a similar type. Such kernels may characterise an entire application domain or parts of different application domains. The comparison of application-domain-specific reconfigurable logic with general-purpose FPGAs and application-specific integrated circuits (ASICs) is also shown. Dependent on the chosen implementation approach, and consequently on the size of the target application space, a different trade-off between flexibility and efficiency can be obtained.

has been proposed as a part of the Totem project at the University of Washington [29]. The software package enabling an automatic creation of coarse-grain custom reconfigurable architectures using a predefined architecture template [42] and a set of a priori known algorithms have been developed [31]. By a considerable reduction in flexibility, the Totem configurable ASIC (cASIC) architectures [30] are able to achieve the cost level which is closer to the cost of traditional ASICs rather than to the cost of FPGAs.

There are several aspects that differentiate the concept of application domain specialisation as proposed in this thesis from the previous work. Firstly, unlike application-oriented architectures from academia that have been optimised towards a single application domain only, we suggest a more complete approach by taking into account requirements of a number of different application domains. Secondly, we assume that a single implementation architecture may be shared between similar processing kernels from different application domains. Thirdly, we aim at much higher level of flexibility than the flexibility offered by the Totem cASIC architectures [29, 30], which are optimised towards a limited set of well-defined kernels only. On the one hand, this increases the cost penalty, on the second hand, it lowers the risk as the mapped kernels can still be updated or replaced with new ones after a reconfigurable architecture is implemented in silicon.

2.3 Application domain characterisation

The purpose of the application domain characterisation is to describe the nature of processing kernels that have been identified as a part of a target application domain. The characterisation of the kernels is done using an *application domain model* that is defined by a number of parameters. As shown in Figure 2.6, such a model plays a role of an interface between an application domain and the implementation architecture of a reconfigurable logic core. As we will explain in detail in Section 2.4 and Chapter 7, the implementation architecture of a reconfigurable logic core is described by a means of a template. During the process of application domain specialisation the parameters of the application domain model are translated onto the parameters of the architecture template. In this way, a domain-specific architecture (a template instance) of a reconfigurable logic core is derived.

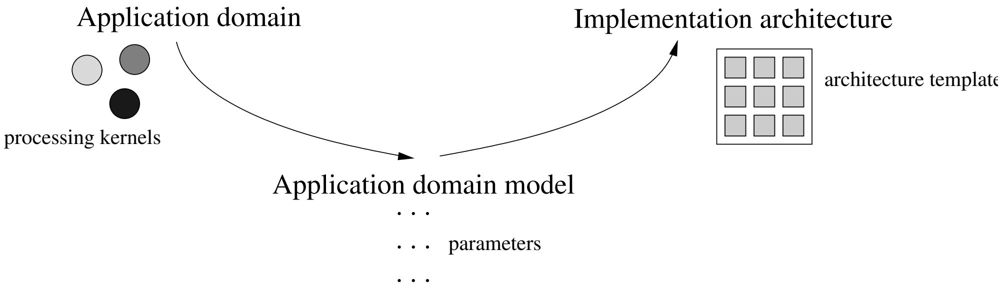


Figure 2.6. The process of application domain specialisation. The characterisation of the processing kernels from a target application domain using the parameters of the application domain model plays a central role.

The parameters (characteristics) of the application domain model capture the information about the processing kernels of a target application domain. Because the parameters should enable an unambiguous characterisation of an application domain yet be general enough to justify the use of reconfigurable logic, they have to be carefully selected. We propose three application domain parameters that meet the above-mentioned criteria. The first two parameters, that is *type of processing* and *word-size*, describe the computations of the processing kernels, while the third parameter, *Rent exponent* [64], describes the communication within the kernels.

2.3.1 Type of processing

The type of processing characterises the functionality of the processing kernels. The type of processing of a kernel is determined by its *dominant type of logic*, such as data-path logic, random logic and memory logic.

Framework

The application domain characterisation using the type of processing as the primary criterion relies on the information on the implementation of a kernel. The dominant type logic, which determines the type of processing of a kernel, is established by calculating the area contribution of different implementation components (basic building blocks) to the total implementation area of the kernel. We distinguish three basic types of implementation components, namely random logic (e.g. simple logic gates, random-logic multiplexers), data-path (e.g. arithmetic modules, data-path multiplexers, modules generating multi-bit Boolean functions) and storage components (e.g. flip-flops, shift registers). Consequently, processing kernels can be characterised as random-logic-oriented, data-path-oriented or storage-oriented.

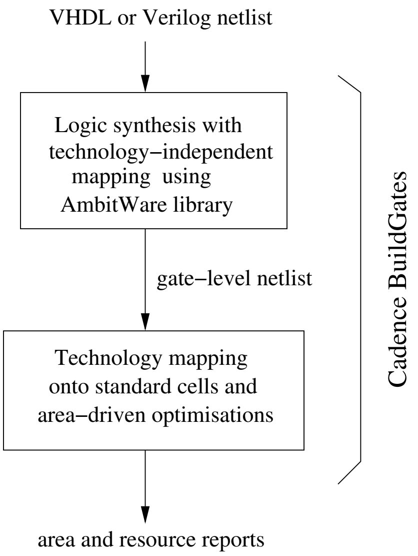

VHDL or Verilog netlist
Logic synthesis with technology−independent mapping using AmbitWare library
gate−level netlist
Technology mapping onto standard cells and area−driven optimisations
area and resource reports
Cadence BuildGates

Figure 2.7. The experiment flow.

Figure 2.7 shows the procedure according to which the application domain characterisation of the benchmark kernels has been realised. Each experiment consisted of two steps, that is, a technology-independent mapping and a technology-dependent mapping which were performed in the Cadence BuildGates synthesis framework. During the first step, the high-level netlist of a benchmark kernel (in a VHDL or Verilog format) was synthesised using the generic Cadence AmbitWare library [22]. The use of the AmbitWare library allowed the macro-functions (e.g. adders, multipliers, etc.), which were identified during the synthesis, to be instantiated in the form of macro-components and to be preserved during the entire mapping process. To guarantee the proper treatment of macro-components, the synthesis parameters had to be set accordingly. We found that the most crucial parameters were: *aware_dissolve_width* (the number of bits of a data-path component below which its structure is dissolved into random logic), *aware_mux_width* (the number of data inputs of a multiplexer below which its structure is dissolved), and *aware_adder_architecture* (the adder type). We used default values of these parameters as they guarantee the optimal design [22]. Consequently, four bits were chosen as the minium granularity of data-path components, eight as the minimal number of inputs of a preserved multiplexer, and the ripple-carry structure as a default adder architecture. In the second step of the experiment, the gate-level netlist of a benchmark kernel that has been generated in the first step was mapped onto standard cells of the TSMC 0.13 μm CMOS process [79, 99].

Forty representative kernels from several different application domains have been selected as the benchmark set. The set included: signal processing kernels primarily in the area of mobile communication (e.g. audio processing, GSM, channel decoding), bit-level processing and pixel processing kernels from digital video and still image processing, various interface standards, peripheral blocks of microcontrollers, data encryption kernels from security applications, and acceleration kernels from cryptography applications. Each benchmark kernel was characterised using the information from the area and resource reports that were generated after the technology-dependent mapping step. In this way, the percentage contributions of random logic, data-path logic and storage components to the total benchmark area were established. The macro-components that were identified by AmbitWare (except multiplexers) were classified as data-path logic. The multiplexers were divided into groups and classified as data-path or random logic components dependent on the number of bits of their inputs (see Fig. 2.8). Flip-flops and elements implementing constant values were regarded as the storage components. All remaining components were treated as random logic.

Results and analysis

The benchmark set that has been used in our experiments is described in Table 2.2. For each benchmark kernel, the table lists the application domain the kernel originates from and the complexity of the kernel found through the mapping process. The information about the complexity of a kernel is given both in the silicon area

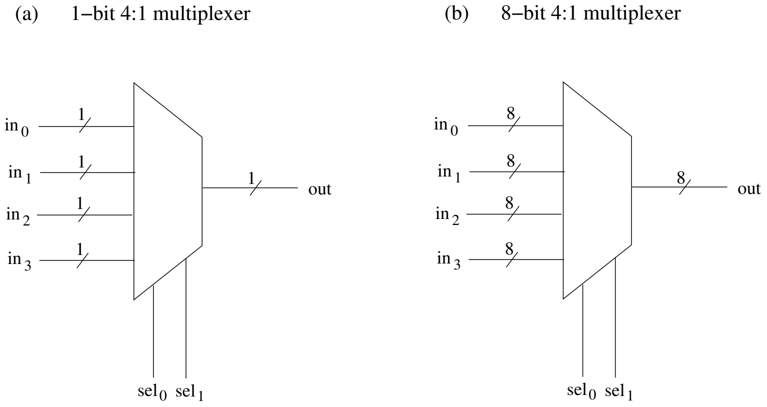


Figure 2.8. The classification of multiplexer structures as (a) random logic components and (b) data-path components dependent on the number of bits of their inputs (outputs). The multiplexers with 1-bit inputs (outputs) are random logic multiplexers.

of its standard-cell implementation (in μm^2) and in the number of 2-input NAND gate area equivalents. The typical word-size of data in each kernel is also listed.

Figure 2.9 shows the results of the application domain characterisation that has been performed on our benchmark set using the type of processing as the main characterisation criterion. The numbers on the horizontal axis correspond to the processing kernel numbers from Table 2.2, while the vertical axis represents the percentage of the silicon area occupied by random logic, data-path logic, and storage components. The detailed characterisation of data-path logic (i.e. the contribution of arithmetic elements, multi-bit Boolean logic elements, data-path multiplexers) and storage components (i.e. the contribution of flip-flops and elements implementing constant values) are shown in Figures 2.10 and 2.11, respectively.

As we indicated earlier, the results of our application domain characterisation procedure are sensitive to the synthesis parameters. To investigate this aspect we focused on the *aware_mux_width* parameter as potentially having the strongest impact on the obtained results. For this purpose we set *aware_mux_width*=2. This implies that all multiplexers that are identified in a processing kernel are preserved as macro-components instead of being dissolved and optimised with surrounding logic. Figure 2.12 shows the considerable number of multiplexers that have been identified in this way (compare to the results in Figures 2.9 and 2.10). The average increase of 12% in the silicon area compared to the default case (i.e. when *aware_mux_width*=4) has been noted.

We draw the following conclusions:

1. *The type of processing is a relevant characteristic of an application domain.* Similar trends in the type of components used for the implementation of benchmark kernels (Figure 2.9) relate to the origin of the kernels (Table 2.2). Consequently, the type of processing (established via the dominant type of implementation components) characterises an application domain (e.g. kernels 1–7, 36–40) or similar sets of kernels of different application domains (e.g. kernels 18–21, 30–31, 32–34).

2. *Three basic classes of processing kernels can be identified, that is random-logic-oriented, data-path-oriented and memory-oriented.*
   Using the type of processing as the main criterion allows the classification of kernels. The three identified classes of kernels correspond to the sets of kernels 35–40 (random logic), 1–17 (data-path) and 18–34 (memory) in Figure 2.9.

3. *Processing kernels require also other kinds of implementation components than those indicated by the type of processing of a kernel.*
   Despite a strong dominance of data-path (arithmetic) components in the benchmark kernels 1–17, their implementations need also random logic. Furthermore, the implementations of the memory-oriented benchmark kernel 18–34 reveal the presence of data-path and random logic. Finally, the

No.	Benchmark	Application	Complexity		Word-size
	kernel	domain	$[\mu m^2]$	gate count	[#bits]
1	shift_down15_asu	audio	8896	1103	16
2	f_add_asu	audio	1339	166	16
3	f_sub_asu	audio	1517	188	16
4	f_mult_r_asu	audio	10647	1320	16
5	shift_up15_asu	audio	9065	1124	16
6	fbb_multm_asu	audio	45752	5670	32/48
7	fbb_addc_asu	audio	18135	2248	32/48
8	deci_asu	GSM (speech coding)	182	23	6
9	adder_asu	GSM (speech coding)	3161	392	14
10	mac_asu	GSM (speech coding)	4206	521	12/14
11	viterbi_alpha	channel decoding (backend)	8117	1006	8
12	viterbi_betha	channel decoding (backend)	8103	1004	8
13	viterbi_lambda	channel decoding (backend)	17693	2193	8
14	nyquist_filter	channel decoding (frontend)	174318	21604	2 x 10
15	radix4_butterfly	channel decoding (frontend)	99317	12309	2 x 16
16	cordic	channel decoding (frontend)	35224	4366	14
17	ppone	image (intermediate processing)	5606	696	10
18	bit_packer	video (image coding)	13374	1658	32
19	enc_asu	video (image coding)	105631	13091	12/32
20	dec_asu	video (image coding)	113847	14110	32/12
21	rle_asu	video (image coding)	9913	1229	12/27
22	rnd_asu	video (auxiliary unit)	5192	644	7
23	memprt_asu	video (auxiliary unit)	35362	4383	16,32/32
24	recon_asu	video (motion compensation)	12420	1539	8
25	clip_asu	video (motion estimation)	44635	5532	12,16/16
26	compute_select	video (pixel processing)	17154	2126	2 x 8
27	interpolator	video (pixel processing)	1176	146	8/16
28	erosion	video (pixel processing)	17935	2223	10/16
29	mix_select	video (pixel processing)	8866	1099	8
30	timer	peripherals	5168	641	8
31	uart51	peripherals	8656	1073	8
32	ahb_trafic_ctrl	interfaces	22163	2747	32
33	ip_1804	interfaces	32733	4057	32
34	i2c	interfaces	18669	2314	32
35	ulaw_to_linear	security	194	24	16
36	a5rfua	cryptography	276	34	32
37	desrfua	cryptography	216	27	32
38	loki2rfua	cryptography	200	25	32
39	magenta	cryptography	682	84	32
40	md5rfua	cryptography	149	18	32

Table 2.2. Benchmark kernels used in the application domain characterisation procedure and their basic characteristics derived from the implementation in a 0.13 μm CMOS process.

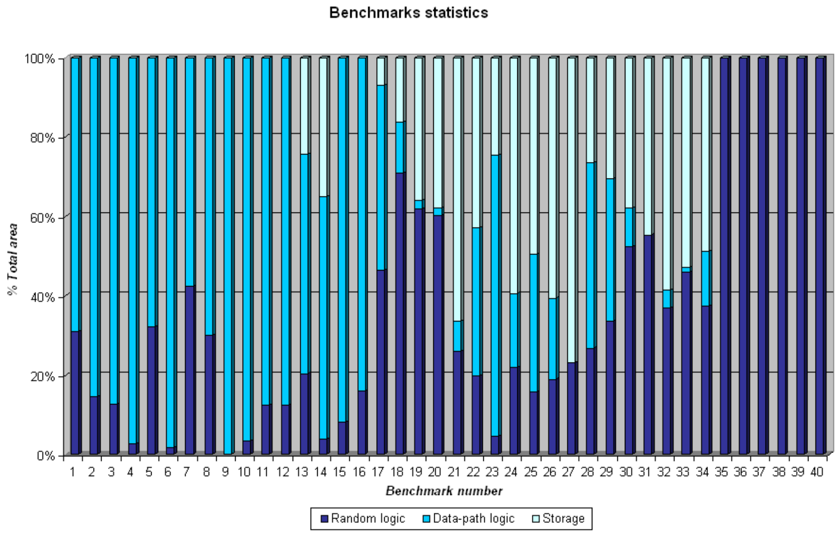


Figure 2.9. Application domain characterisation based on the type of processing as the main characteristic. The dominant type of implementation components that determines the type of processing suggests the classification of the benchmark kernels onto data-path-oriented (kernels 1–17), memory-oriented (kernels 18–34) and random-logic-oriented (kernels 35–40).

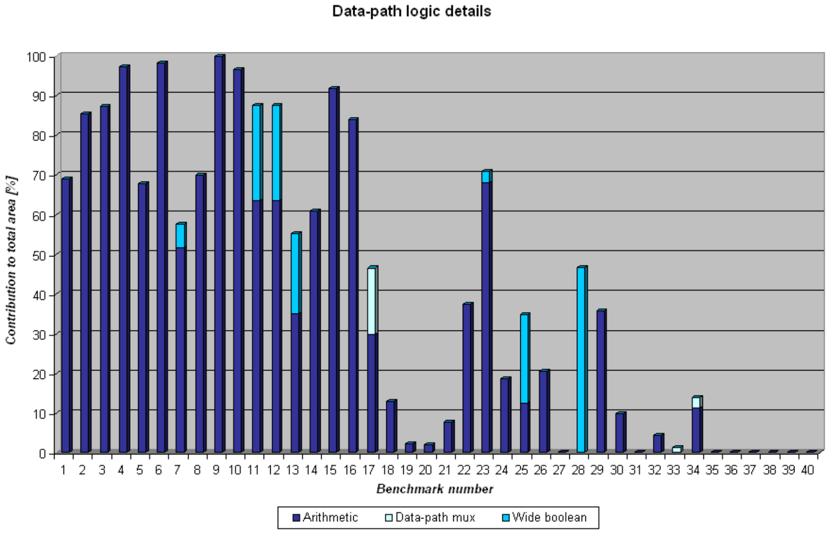


Figure 2.10. The detailed characterisation of the data-path components and their contribution to the implementation area of different benchmark kernels.

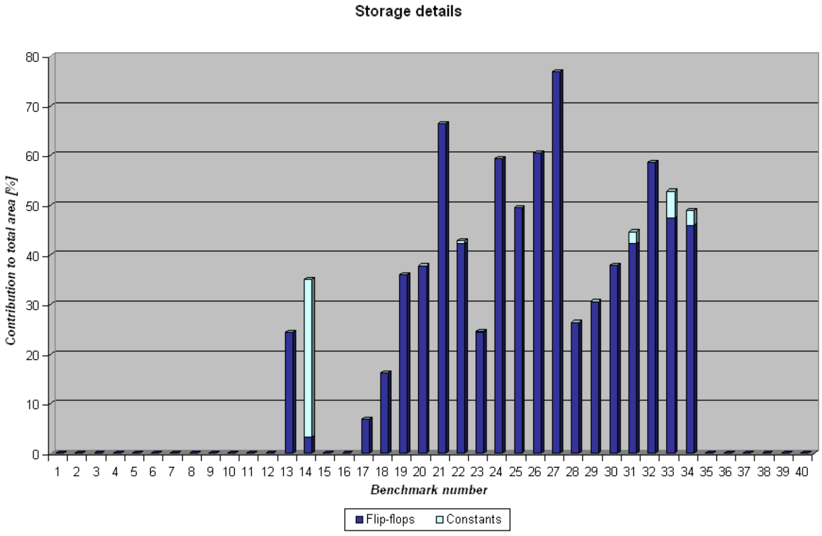


Figure 2.11. The detailed characterisation of the storage components and their contribution to the implementation area of different benchmark kernels.

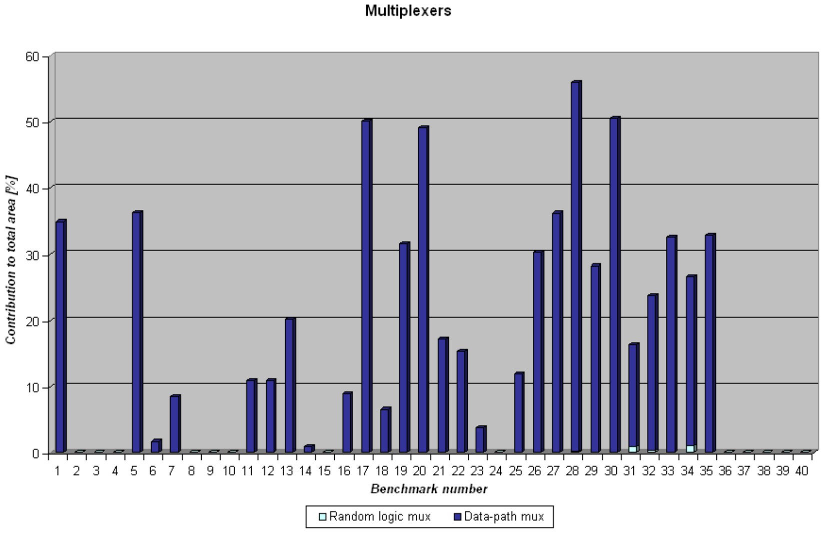


Figure 2.12. The contribution of the data-path and random logic multiplexers to the implementation area of different benchmark kernels. The multiplexers were derived for *aware_mux_width*=2 (by default *aware_mux_width*=4).

purely combinational character of the benchmark kernels 35–40, for which the above observation seems to fail, has been determined by the specific implementation of the compiler that is used to identify the kernels. (Such a compiler analyses the application code and extracts only a combinational portion of the critical kernels. For the details see [36].)

4. *Multiplexers are common implementation components.*
   The considerable amount of random logic and data-path multiplexers that have been identified in the analysed benchmark kernels suggest that multiplexers can be regarded as basic implementation components.

2.3.2 Word-size

The word-size characterises the *granularity of computations* in processing kernels. Consequently, it also characterises the granularity of data (i.e. the number of bits of data signals) on which such computations are performed. Unlike the type of processing, which may be different for different sets of kernels of a given application domain (see the discussion in Section 2.2), the word-size is a unique characteristic of an application domain. The typical value of the data word-size in common application domains is generally known.

Framework

The data word-sizes of processing kernels from our benchmark set (see Table 2.2) were derived based on the typical granularity of macro-components identified during the above-described application domain characterisation procedure. Despite small divergences (e.g. due to the extended bit representation of internal signals), we found that the word-size values identified for the analysed application domains match well the generally known values. Therefore, the analysed application domains can be characterised by the following data word-sizes.

- telecommunication: 12–16 bits,
- audio: 16–24 bits,
- image/video: 8–12 bits,
- peripherals: 8 bits,
- interfaces: 32 bits,
- cryptography: 32 bits,
- coding: 1–8 bits.

2.3.3 Rent exponent

Until now we considered only the computation aspects of processing kernels. Intuitively, there should be, however, a relation between the type of computations and their internal communication (interconnect structure). For instance, data-path kernels feature intra-macro communication (i.e. short, point-to-point regular connections between bit-slices of macro-components) and inter-macro communication (i.e. multi-bit buses between macro-components). In contrast, the communication in the random logic type of kernels lacks a clear structure (e.g. irregular connections of different lengths, independent interconnect wires rather then buses, high fan-out rather than low fan-out signals).

There is no single property by means of which the communication structure within processing kernels can be fully characterised. Nevertheless, Rent's rule [64] and the associated Rent exponent play an important role in the modelling of interconnect structures. Rent's rule is an empirical formula given by Equation 2.1 that describes the relation between the average number of terminals T of a given logic module (circuit) and the complexity B of the logic module expressed in the number of its logic components (gates). t is a proportionality constant called *Rent coefficient* which is equal to the average number of terminals per logic component; the parameter p is the *Rent exponent* such that $0 < p < 1$. The Rent exponent is calculated as the slope of the line representing Rent's relationship on the log-log scale.

$$T = tB^p \tag{2.1}$$

Because the Rent exponent depends on the interconnect topology[3] of a circuit, this parameter is of particular importance for us. As shown by DeHon [38], the Rent exponent reflects the interconnect growth or the locality in the interconnect requirements. A small value of the Rent exponent ($p \leq 0.5$) is typical for short and regular connections, while a large value ($p > 0.5$) is typical for rich and irregular connections (like in random logic, for example). The Rent exponent can be therefore considered as a measure of the complexity of the interconnection topology of a circuit [75], and consequently as a means for characterising circuits [96]. For these reasons we propose the Rent exponent (and the associated Rent coefficient) as the third parameter of our application domain model. In this sense, the Rent parameters reflecting the type of communication within processing kernels are complementary in nature to the type of processing and word-size parameters that capture the information about the computations of the kernels.

Framework

Though we have not exploited the possibility of characterising processing kernels based on their connectivity, we outline briefly a possible implementation strategy

[3] In this thesis we assume the use of the so-called *intrinsic* Rent exponent [51] which is the lower bound of p.

for such a characterisation procedure. Instead of using directly the information about the net length distribution (e.g. according to the relation proposed by Van Marck in [106]), we assume after [122] that the interconnect requirements of a design netlist are captured by the *fan-out distribution* (i.e. the number of nets versus fan-out). We rely on the fan-out distribution because the formula proposed by Zarkesh-Ha in [122] (see Equation 2.4) takes into account the complexity of the design. The fan-out distribution is derived using the equivalent Rent parameters t_{eq} and p_{eq} of a design (see Equations 2.2 and 2.3, respectively), which are geometric average of the Rent parameters of all logic modules in the netlist.

The value of the Rent exponent is sensitive to the modifications of the netlist that are the result of different CAD algorithms (e.g. partitioning). Therefore, we assume that in the connectivity-driven application domain characterisation procedure the netlists of the processing kernels must be first technology mapped onto a proper type of a logic element and afterwards clustered into logic blocks of a proper size (see the discussion in Section 2.4). This is in contrast to the computations-driven characterisation of the kernels (i.e. using the type of processing and word-size parameters) in which the gate-level format of the netlist is sufficient. To assume a uniform complexity of the clusters into which the technology-mapped netlist is partitioned, the Rent exponent-driven clustering method (e.g. such as described in [92]) is suggested. During such a clustering it must be guaranteed that the Rent exponent of each cluster is not greater than the Rent exponent of the (logic block) architecture. The Rent exponent of the architecture is calculated using Rent's rule based on the information about the physical number of terminals of the logic block and the physical number of logic elements in the logic block. The Rent coefficient is the number of terminals of the logic element. Similarly, the Rent exponent p_i of the i-th cluster is calculated based on the number of the occupied terminals of the logic block and the number N_i of the occupied logic elements in the logic block. The Rent coefficient t_i of the i-th cluster is calculated as the average number of the occupied terminals of the logic element. The difference between the Rent parameters of the architecture and the design cluster is explained in Figure 2.13.

The equivalent Rent parameters t_{eq} and p_{eq} of a benchmark netlist that has been partitioned into N clusters are given by Equations 2.2 and 2.3, respectively. The fanout distribution of such a netlist is described by Equation 2.4, where $net(m)$ denotes the number of m-terminal nets.

$$t_{eq} = \sqrt[N]{(\prod_{i=1}^{N} t_i N_i^{p_i})} \tag{2.2}$$

$$p_{eq} = \frac{\sum_{i=1}^{N} p_i N_i}{N} \tag{2.3}$$

$$net(m) = \frac{t_{eq} N((m-1)^{p_{eq}-1} - m^{p_{eq}-1})}{m} \tag{2.4}$$

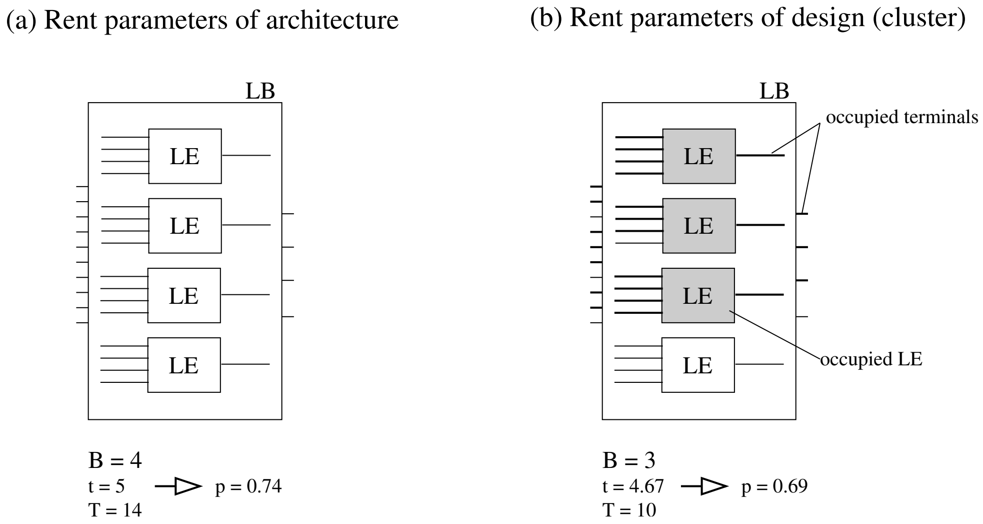


Figure 2.13. The example of the derivation of the Rent parameters for (a) an architecture and (b) a design. LE and LB denote logic elements and logic blocks, respectively.

2.4 Design flow

Figure 2.14 shows the design flow for the generation of domain-specific reconfigurable logic cores. The flow relies heavily on the concept of application domain characterisation that has been described in Section 2.3. The key elements of the flow are summarised below.

- *Identification of the candidate processing kernels:* The target application domain is represented by a group of applications. The applications are characterised by processing kernels. In this step, the candidate processing kernels that are to be implemented in reconfigurable logic are identified.

- *Characterisation of the kernels:* The selected candidate kernels are characterised by the parameters of the application domain model, that is, the type of processing, word-size, and Rent exponent, according to the procedures described in Section 2.3.

- *Classification of the kernels:* The characteristics of the processing kernels that have been established during the previous step are used as the classification criteria. In consequence, the kernels sharing identical or similar characteristics are grouped into sets. Additional constraints (such as design effort, a soft or hard type of the generated reconfigurable core) determine the final partitioning of the kernels between the sets.

- *Template parameters matching:* The parameters of the application domain model that characterise the identified processing kernels are translated onto the parameters of the architecture template (see details in Chapter 7). The following relations between the parameters are assumed.

  - Type of processing determines the type of a basic logic element of a reconfigurable logic block. Consequently, it also establishes the elementary functionality of the logic block.
  - Word-size determines the total number of logic elements in the logic block. This is equivalent to defining the granularity of computations that can be performed in the logic block.
  - Rent parameters via the fan-out distribution are used as guidelines for defining the interconnect length segmentation in a reconfigurable logic core. Similarly to [93], we make an assumption that the fan-out distribution of a circuit netlist is proportional to the distribution of the length of interconnect segments in a reconfigurable logic architecture.

- *Template instantiation:* The domain-specific architecture of a reconfigurable logic core is generated as an instance of the generic architecture template. The template parameters are instantiated based on the parameters of the application domain model and an additional input from a user. Iterative feedback is assumed for the final tuning of the architecture (e.g. the interconnect part).

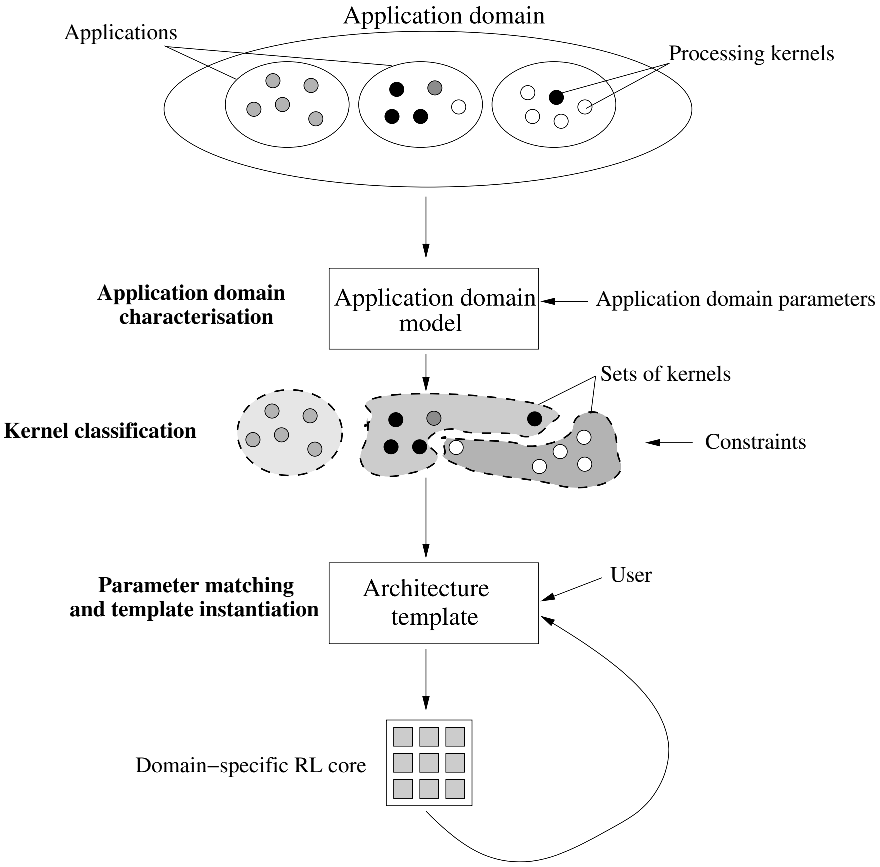


Figure 2.14. Design flow for the generation of domain-specific reconfigurable logic cores.

2.5 Classification of reconfigurable logic architectures

The type of processing can be regarded as a fundamental characteristic of processing kernels since it influences other characteristics of the kernels (e.g. interconnect requirements; see Section 2.3.3). Consequently, we choose the type of processing as the primary optimisation criterion of reconfigurable architectures. The other proposed parameters of the application domain, that is word-size and Rent exponent, are used for more specific optimisations, if such are economically justified.

Following this reasoning, we propose a general classification of reconfigurable logic architectures based on the type of processing as the main criterion. Consequently, using the results of the experiment described in Section 2.3.1, we derive three basic types of domain-specific reconfigurable logic architectures, that is:

- data-path-oriented architectures,
- random-logic-oriented architectures,
- memory-oriented architectures.

2.5.1 Data-path-oriented architectures

The *data-path-oriented architectures* target kernels dominated by data-path logic, and arithmetic in particular (see Figures 2.9 and 2.10). In such kernels, coarse-grain operations (i.e. of the nibble-level granularity and higher) and the operations with the carry signal propagation are typical. This type of processing is characteristic for digital signal processing (DSP).

The data-path-oriented architectures should thus guarantee an efficient implementation of data-path logic. However, they should also allow an implementation of random logic, probably less efficiently. This is important since in the implementation of some data-path functions random logic is also needed (e.g. an implementation of a binary divider, which is an arithmetic function, requires a small controller and thus random logic).

According to Figure 2.9, the data-path-oriented reconfigurable logic architecture would be the most suitable for the mapping of the benchmark kernels 1–17. We conclude thus that this type of architecture is well-suited for the implementation of hardware accelerators in the audio and telecommunication applications, for example (see Table 2.2).

2.5.2 Random-logic-oriented architectures

The *random-logic-oriented architectures* are meant primarily for the efficient mapping of big chunks of combinational and sequential logic. Since in typical applications random logic is mixed with data-paths, and since random logic elements are sometimes implemented using arithmetic elements (e.g. a binary counter can be implemented using a binary adder), the random-logic-oriented architecture should enable the mapping of arithmetic functions, that is with a propagating carry signal.

The random-logic-dominated type of functionality characterises the benchmarks kernels 36–40 from Figure 2.9. Because such benchmarks have been deliberately deprived of arithmetic (see conclusion 3 in Section 2.3.1), in practice some support for the implementation of arithmetic would also be needed. According to our analysis, the random-logic-oriented type of a reconfigurable logic architecture suits well the implementation of hardware accelerators in cryptography applications, for example.

2.5.3 Memory-oriented architectures

The *memory-oriented architectures* target processing kernels requiring storage resources. The storage (memory) resources are needed because of:

- the algorithm requirements: the memory components implement algorithmic delays by means of a local storage (e.g. small data memories, register files, look-up tables in distributed arithmetic [113, 111]) or various delay elements (e.g. FIFOs, buffers, feedback loops etc.),
- the implementation requirements: the memory components are inferred by a specific implementation method (e.g. balancing delays of different paths, circuit retiming).

Typically, the implementation of the memory-oriented kernels will also require data-path and random logic components, which relative amounts may differ per application. Therefore, to enable an efficient mapping of such kernels, the memory-oriented architectures will have to be multi-functional (general-purpose).

In Figures 2.9 and 2.11, the memory functionality has been identified in the benchmark kernels 13–14 (telecommunication applications) and kernels 18–35 (video, interface, peripherals, and security application domains). As expected, the memory components in the implementation of these kernels play different roles. For example, they implement delays (kernel 13), a storage for constant coefficients of a digital filter (kernel 14), feedback loops in control structures (kernels 18–35), complex storage resources such as FIFOs and register files (kernels 32–34). We propose to use the memory-oriented type of a reconfigurable logic architecture for the implementation of kernels from the video, interface, and peripheral domains. However, the choice of a suitable implementation architecture for a given processing kernel may be determined in practice also by other factors (e.g. the importance of the kernel, the type and total complexity of the required storage resources). For example, though benchmark kernels 13–14 include memory components, they have been classified as data-path-oriented, and the data-path-oriented reconfigurable architecture has been selected for their implementation. This is correct since the benchmark kernels 13–14, similarly to the kernels 11–13 and 16–17 from the same application domain, are characterised by the data-path-dominated type of processing. At the same time, it is assumed that the data-path-oriented architecture contains flips-flops that enable an implementation of memory elements (see Chapter 4). However, if more sophisticated memory structures (e.g. FIFOs) or a considerable amount of storage resources were required, the memory-oriented reconfigurable logic architecture might be a better implementation choice. A similar reasoning can be applied to the benchmark kernels 18–34.

2.6 Conclusions

In this chapter we showed that modern reconfigurable logic devices, such as FPGAs, are general-purpose in nature. Although the general-purpose nature allows the use of FPGAs for a wide range of applications, it also leads to a high intrinsic cost. For example, the area, delay and energy of FPGA devices are about one to three orders of magnitude higher than the same parameters of ASICs.

Because low-cost is critical for reconfigurable-logic-based SoCs targeting consumer applications, we proposed a method of the reduction of the intrinsic cost of reconfigurable logic based on the concept of application domain specialisation. The key of the concept is the characterisation of processing kernels that are a part of a target application domain and using this information during the design process. Unlike general-purpose FPGAs, domain-specific reconfigurable logic cores that are a result of such a design process are optimised for a specific type of kernels. In this way a much better balance between flexibility and cost can be achieved.

We proposed three parameters that can be used to characterise an application domain. The first two parameters, that is the type of processing and word-size, describe the computations of processing kernels. The third parameter, that is the Rent exponent, captures the information about the communication structure within the kernels. Using the set of benchmark kernels selected from the application domains that would benefit from using reconfigurable computing we showed how such kernels can be characterised with the mentioned parameters. We also described the complete design flow for the generation of domain-specific reconfigurable logic cores. Finally, based on the application domain analysis we derived three classes of reconfigurable logic architectures that should be supported, namely data-path-oriented architectures, random-logic-oriented architectures, and memory-oriented architectures. Though optimised towards a specific type of functionality, all three classes of architectures are assumed to allow mapping of other types of functions as well (but at a higher cost).

Chapter 3

BASIC CONCEPTS

The purpose of this chapter is to acquaint the reader with some basic concepts which will be exploited in the remaining part of this thesis. We start with elementary definitions and properties of Boolean and binary arithmetic functions that are relevant in the context of this work. Further, we propose two properties of a binary addition that enable the optimisation of the LUT-based arithmetic. Finally, we discuss the architectural concepts on which our domain-oriented reconfigurable logic architectures are based (see details in Chapters 4–6). We also present cost metrics that will be used for the benchmarking of such architectures.

3.1 Generic properties

3.1.1 Background

Boolean algebra forms a mathematical foundation of the analysis and design of logic circuits. Since binary logic circuits can be thought of as constructed of switches with two possible states only (i.e. ’on’ and ’off’) [104], the two-valued Boolean algebra $\mathcal{B} = (\mathbf{B}, +, \cdot, 0, 1)$, where the set $\mathbf{B} = \{0,1\}$, is of practical use.

The behaviour of a logic circuit can be unambiguously described by means of *Boolean functions*. An *n*-variable (single-output) Boolean function is a mapping of the form

$$f : \mathbf{B}^n \to \mathbf{B}. \tag{3.1}$$

Often, the term *’switching function’* is used to describe a Boolean function defined in the two-valued algebra with the carrier $\mathbf{B} = \{0,1\}$ [18]. In this thesis, we will use the term ’Boolean function’ to mean a ’switching function’.

Let $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ be a Boolean function of n variables. The cofactor of $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to variable x_i, also called a *positive cofactor,* is $f|_{x_i=1} = f(x_1, x_2, \ldots, 1, \ldots, x_n)$. The cofactor of $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to variable $\bar{x_i}$, also called a *negative cofactor,* is $f|_{x_i=0} = f(x_1, x_2, \ldots, 0, \ldots, x_n)$ [72].

Any Boolean function $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ of n variables can be expanded into the sum of products of n literals[1], called *minterms*. At each level of the recursive expansion associated with an arbitrary logic variable x_i, the function f can be represented as the sum of two products: the product of variable x_i and its positive cofactor $f|_{x_i=1}$, and the product of the complement of variable x_i, that is $\overline{x_i}$, and its negative cofactor $f|_{x_i=0}$. Such an expansion of a Boolean function is called *Boole's expansion* or often *Shannon's expansion* [18], and is formulated as follows.

$$f(x_1, x_2, \ldots, x_i, \ldots, x_n) = x_i \cdot f|_{x_i=1} + \overline{x_i} \cdot f|_{x_i=0} \tag{3.2}$$

The *dual* of a Boolean function $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$, denoted f^d, is defined as $f^d(x_1, x_2, \ldots, x_i, \ldots, x_n) = \overline{f}(\overline{x_1}, \overline{x_2}, \ldots, \overline{x_i}, \ldots, \overline{x_n})$ [15]. A Boolean function is said to be *self-dual* if $f^d = f$ [15].

A combinational logic circuit having n inputs and m outputs can be described by m n-variable Boolean functions $f_i(x_1, x_2, \ldots, x_n)$, $i = 1 \ldots m$. An alternative specification of the circuit behaviour is possible using a *truth table*. The truth table of such a circuit has 2^n rows and $n + m$ columns. The n-element row vectors of the input part of the truth table define all possible states of the circuit inputs, while the m-element row vectors of the output part of the truth table define the corresponding states of the circuit outputs. An example of a truth table is shown in Figure 3.4.

An n-variable Boolean function $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ can be implemented using an n-input *look-up table* (*n-LUT*), where n denotes the *size* of the look-up table. A typical n-LUT consists of 2^n memory cells and a $2^n : 1$ multiplexer in a configuration as shown in Figure 3.1 [80]. The memory cells store the values of the Boolean function as defined by its truth table, while the multiplexer works as a data selector.

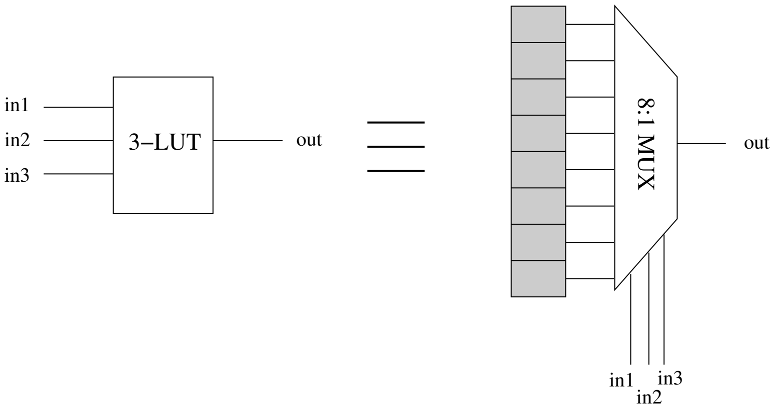


Figure 3.1. An example of a 3-input look-up table and its implementation. The 3-LUT suffices to implement a Boolean function of 3-variables.

[1] Literal is a logic variable or its complement.

Because the number of the required LUT memory cells increases exponentially with the increase of the look-up table size, in practice look-up tables map relatively small Boolean functions (i.e. of few logic variables only). The mapping of larger Boolean functions can be enabled by Shannon's expansion (see Equation 3.2), for example. The expansion is applied recursively until the number of logic variables in cofactors of the Boolean function reaches the size of a target look-up table. The values of such cofactors are stored then in the look-up tables, and the signals related to the logic variables with respect to which the Boolean function is expanded control 2:1 multiplexers at the LUT outputs. An implementation of a 4-input Boolean function according to this principle is illustrated in Figure 3.2.

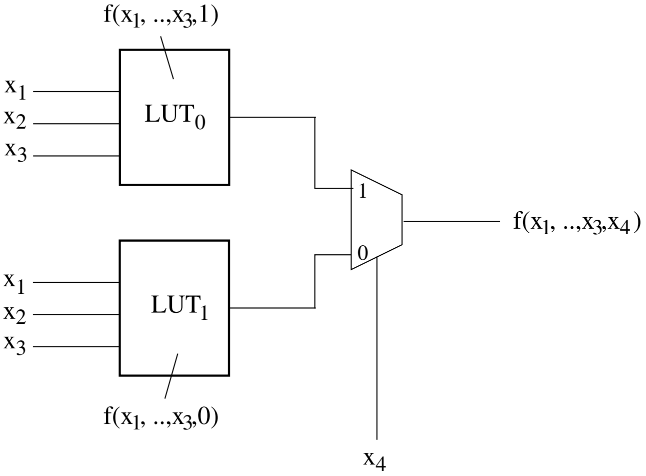


Figure 3.2. An implementation of a 4-input Boolean function $f(x_1,x_2,x_3,x_4)$ using two 3-input look-up tables LUT_0 and LUT_1.

A *binary addition* is a fundamental arithmetic operation. A *full adder* circuit that implements a 1-bit addition (see Figure 3.3) has two 1-bit data inputs *a* and *b*, and a carry input *ci*; it also has a sum output *s* and a carry output *co*. The outputs *s* and *co* of a full adder are described by Boolean functions f_s and f_{co} as given by Equations 3.3 and 3.4, respectively. Figure 3.4 shows the truth table of a full adder.

$$f_s(a,b,ci) = a \oplus b \oplus ci \tag{3.3}$$

$$f_{co}(a,b,ci) = ab + (a \oplus b) \cdot ci \tag{3.4}$$

An *n*-bit adder can be built of *n* full adders connected in a chain as shown in Figure 3.5. In such a ripple-carry adder structure [80], the carry output signal of the *i*-th adder is connected to the carry input signal of the $(i+1)$-th adder, that is $ci_{i+1} = co_i$.

The *adder inverting property* [80] states that the inversion of all inputs of a full adder results in the inversion of all its outputs, that is

$$f_s(\overline{a},\overline{b},\overline{ci}) = \overline{f_s}(a,b,ci), \qquad f_{co}(\overline{a},\overline{b},\overline{ci}) = \overline{f_{co}}(a,b,ci). \tag{3.5}$$

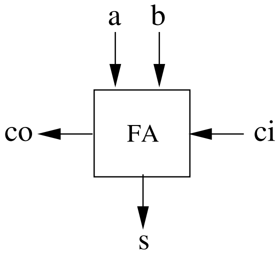


Figure 3.3. A full adder (FA).

a	b	ci	s	co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 3.4. Truth table of a full adder.

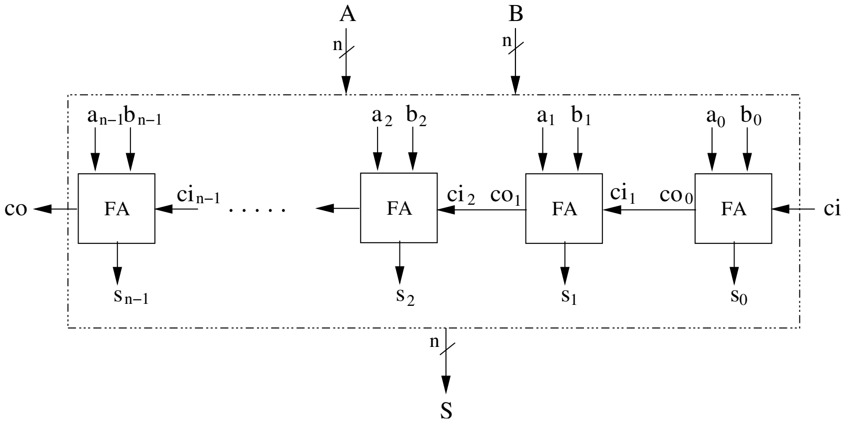


Figure 3.5. The implementation of an *n*-bit binary adder using *n* full adders.

3.1.2 Inversion-based folding type I

We propose an inversion-based folding method of the function describing a sum output of a 1-bit binary adder. The notation is assumed as in Section 3.1.1.

Theory

THEOREM 3.1
(Partial inverting property of a full adder.)
The inversion of one of the inputs of a full adder results in the inversion of its sum output.

From this statement (assuming the carry input *ci* as the *reference input*) follows

$$f_s(a,b,\overline{ci}) = \overline{f_s}(a,b,ci), \qquad (3.6)$$

or equivalently:

$$f_s(a,b,0) = \overline{f_s}(a,b,1). \qquad (3.7)$$

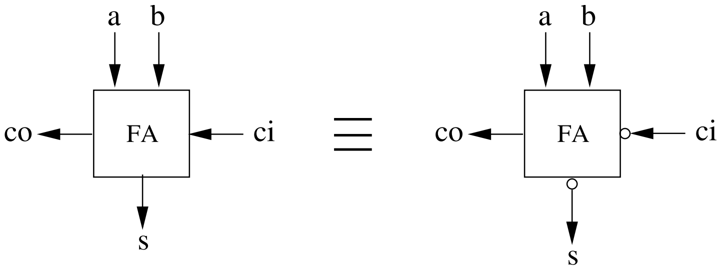
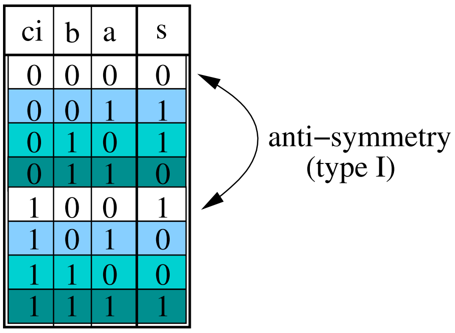


Figure 3.6. The interpretation of the partial inverting property of a full adder.

Note that $f_s(a,b,0)$ and $f_s(a,b,1)$ represent the negative and positive cofactors of the function f_s with respect to ci. The graphical interpretation of the partial inverting property of a full adder is shown in Figure 3.6.

The proof of Equation 3.6 is trivial as it can be derived directly from the adder truth table shown in Figure 3.7 using basic theorems of Boolean algebra [18]. The truth table from Figure 3.7 is obtained from the original adder truth table (see Figure 3.4) by reordering the row vectors of its input and output parts[2] with respect to the value of the input variable ci. For the sake of clarity, the corresponding row vectors of the new truth table are marked with the same colour. Such vectors are characterised by the opposite values of the carry input ci and the opposite values of the sum output s. We shall refer to this relation as *anti-symmetry type I*.



ci	b	a	s
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

anti−symmetry
(type I)

Figure 3.7. A part of the adder truth table with row vectors organised with respect to the carry input variable ci. The anti-symmetry (type I) between corresponding row vectors is indicated by the their identical background colours.

Applying Shannon's expansion (see Equation 3.2) to the adder sum function f_s yields:

$$f_s(a,b,ci) = ci \cdot f_s(a,b,1) + \overline{ci} \cdot f_s(a,b,0). \tag{3.8}$$

By substituting $f_s(a,b,1)$ in Equation 3.8 according to Equation 3.7, we derive the following theorem which we shall call the *inversion-based folding type I*.

[2] In the new truth table, the values of the variable co have been deliberately omitted.

THEOREM 3.2
(Inversion-based folding type I.)
If $f_s(a,b,ci)$ is a Boolean function describing the sum output of a full adder, then:

$$f_s(a,b,ci) = ci \cdot \overline{f_s}(a,b,0) + \overline{ci} \cdot f_s(a,b,0) \tag{3.9}$$

Theorem 3.2 indicates that an inversion operation and a cofactor of the function f_s with respect to the selected polarisation of the input variable (e.g. here variable $ci = 0$) suffice to generate the sum function. Since we utilise a single expression only to generate the information captured in both cofactors (i.e. $f_s(a,b,0)$ and $f_s(a,b,1)$), we refer to this implementation method as achieved by *folding*.

Implications for the LUT-based implementation

The values of the negative cofactor $f_s(a,b,0)$ in Equation 3.9 represent half of all possible values of the full adder sum function f_s (see Figure 3.7). A 2-input look-up table (2-LUT) suffices thus to implement $f_s(a,b,0)$. The positive cofactor $f_s(a,b,1)$ is generated by the inversion of the LUT output. The implementation of the sum function f_s of a full adder according to this principle is shown in Figure 3.8(b). In Figure 3.8(a), the size of the output part of the original and compacted (i.e. after folding) truth tables of a full adder are compared. It is clear that the implementation of the full adder sum function according to the inversion-based folding type I allows the size of the required LUT to be halved.

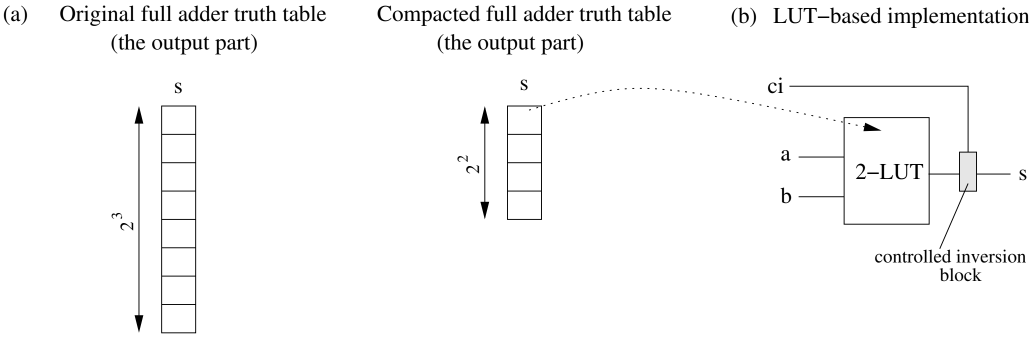

(a)
Original full adder truth table
(the output part)
Compacted full adder truth table
(the output part)
(b) LUT−based implementation
s
2^3
s
2^2
ci
a
b
2−LUT
s
controlled inversion block

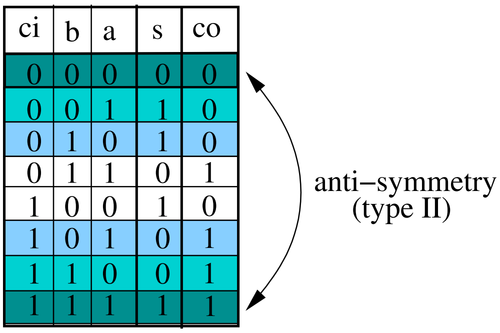
Figure 3.8. The LUT-based implementation of the full adder sum function f_s using the inversion-based folding type I. (a) The comparison of the output parts of the original and compacted truth tables of a full adder, (b) the implementation: the values of cofactor $f_s(a,b,0)$ from the compacted truth table are stored in the 2-LUT, the controlled inversion block implements folding.

3.1.3 Inversion-based folding type II

We propose an inversion-based folding method of the functions describing the sum and carry outputs of an n-bit binary adder. We assume the notation as in Section 3.1.1.

Theory

Figure 3.9 shows a truth table of a full adder in which row vectors have been ordered with respect to the value of the input variable *ci*[3]. The same background colour identifies the pairs of row vectors that are related via the adder inverting property as defined by Equation 3.5 in Section 3.1.1. The related row vectors are characterised by the opposite values of the input variables *ci*, *a*, *b* and the opposite values of the outputs *s* and *co*. Such a relation represents yet another form of anti-symmetry that can be identified in the adder truth table. We shall refer to it as *anti-symmetry type II* (compare with the anti-symmetry type I in Figure 3.7).



ci	b	a	s	co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

anti–symmetry
(type II)

Figure 3.9. The truth table of a full adder with the indicated anti-symmetry type II. In this case, the anti-symmetry represents the relation described by the (full) adder inverting property (see Equation 3.5).

In the ripple-carry implementation of an *n*-bit binary adder that has been shown in Figure 3.5, the carry input and carry output signals of successive adder stages are inter-dependent. Therefore, the adder inverting property defined by Equation 3.5 will also hold for an arbitrary *n*-bit adder.

THEOREM 3.3
(Inverting property of an n-bit adder.)
The inversion of the inputs of an n-bit binary adder results in the inversion of its outputs.

Let $A = \{a_0, a_1, \ldots, a_{n-1}\}$ and $B = \{b_0, b_1, \ldots, b_{n-1}\}$ be the sets of bits defining the first and second inputs (arguments) of an *n*-bit adder. Also, let s_i be the *i*-th bit of the sum output of such an adder, such that $i = 0 \ldots n-1$. Then, the inverting property of an *n*-bit adder can be formulated as follows:

$$f_{s_i}(\overline{A},\overline{B},\overline{ci}) = \overline{f}_{s_i}(A,B,ci) \qquad i = 0 \ldots n-1 \tag{3.10}$$

[3] Note, that such a truth table has the same form as the truth table from Figure 3.7, but is completed with the values of the output variable *co*.

$$f_{co}(\overline{A},\overline{B},\overline{ci}) = \overline{f_{co}}(A,B,ci), \tag{3.11}$$

or equivalently:

$$f_{s_i}(\overline{A},\overline{B},0) = \overline{f_{s_i}}(A,B,1) \qquad\qquad i = 0\ldots n-1 \tag{3.12}$$

$$f_{co}(\overline{A},\overline{B},0) = \overline{f_{co}}(A,B,1). \tag{3.13}$$

Note, that the alternative formulation of the inverting property of an n-bit adder is that the functions f_{s_i} and f_{co} are *self-dual* (see Section 3.1). The graphical interpretation of the inverting property of an n-bit adder is shown in Figure 3.10.

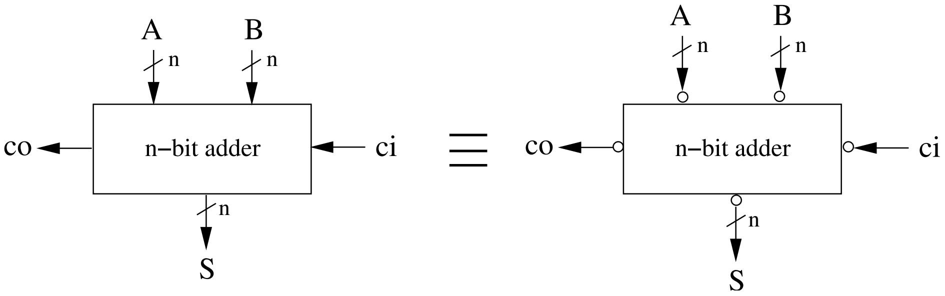


Figure 3.10. The interpretation of the inverting property of an n-bit adder.

Note also, that the n-bit version of the adder inverting property is *different* than the partial inverting property of a full adder described in Section 3.1.2. Firstly, the n-bit adder inverting property holds for an adder with an arbitrary number of bits of its arguments and not for a full adder only. Secondly, the n-bit adder inverting property assumes the inversion of all inputs of an adder instead of only one of its inputs. Finally, it concerns all outputs of an adder rather than the sum output only.

Applying Shannon's expansion with respect to the carry input variable ci (see Equation 3.2) to the sum and carry output functions describing the outputs of an n-bit adder yields:

$$f_{s_i}(A,B,ci) = ci \cdot f_{s_i}(A,B,1) + \overline{ci} \cdot f_{s_i}(A,B,0) \qquad\qquad i = 0\ldots n-1 \tag{3.14}$$

$$f_{co}(A,B,ci) = ci \cdot f_{co}(A,B,1) + \overline{ci} \cdot f_{co}(A,B,0) \tag{3.15}$$

By substituting $f_{s_i}(A,B,1)$ and $f_{co}(A,B,1)$ in Equations 3.14 and 3.15 according to Equations 3.12 and 3.13, we derive the *inversion-based folding type II*.

THEOREM 3.4
(Inversion-based folding type II.)
If $f_{s_i}(A,B,ci)$ and $f_{co}(A,B,ci)$ are Boolean functions describing an i-th bit of the sum output and the carry output of an n-bit adder, respectively, then:

$$f_{s_i}(A,B,ci) = ci \cdot \overline{f_{s_i}}(\overline{A},\overline{B},0) + \overline{ci} \cdot f_{s_i}(A,B,0) \qquad i = 0 \ldots n-1 \quad (3.16)$$

$$f_{co}(A,B,ci) = ci \cdot \overline{f_{co}}(\overline{A},\overline{B},0) + \overline{ci} \cdot f_{co}(A,B,0) \quad (3.17)$$

Theorem 3.4 shows that the values of the sum function f_{s_i} and carry output function f_{co} of an n-bit adder can be generated using only the inversion operation and one of the cofactors of these functions (in Equations 3.16 and 3.17, the negative cofactors with respect to variable ci are chosen). Analogous to Section 3.1.3, we say about this type of implementation as achieved by *folding*. If the reference variable has the polarisation opposite to the selected (i.e. here when $ci = 1$), the values of the reference cofactors have to be modified. In this case, modification of the cofactors is done by the inversion of *all* their inputs.

Implications for the LUT-based implementation

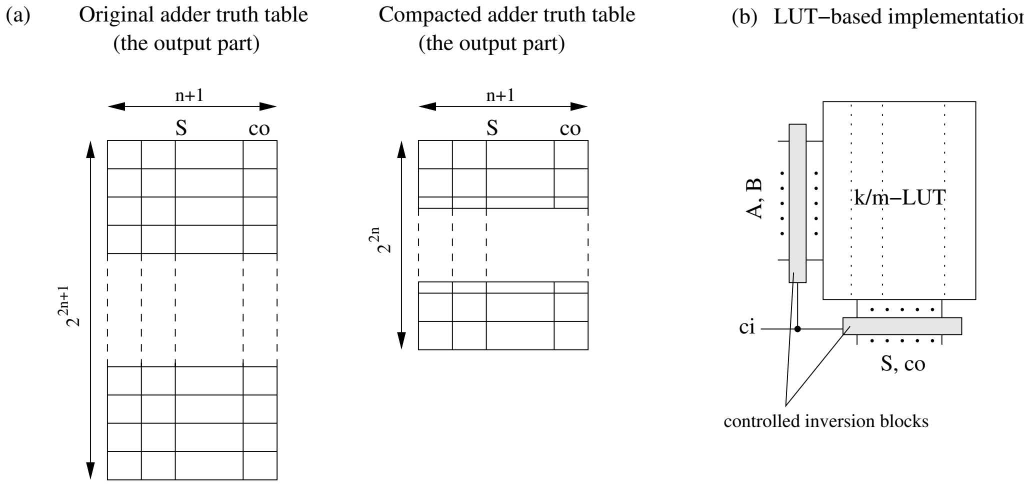

(a)
Original adder truth table
(the output part)
Compacted adder truth table
(the output part)
(b) LUT−based implementatio
n+1
S
co
2^{2n+1}
n+1
S
co
2^{2n}
A, B
k/m−LUT
ci
S, co
controlled inversion blocks

Figure 3.11. The LUT-based implementation of the n-bit addition using the inversion-based folding type II. (a) The comparison of the output parts of the original and compacted truth tables of a n-bit adder, (b) the implementation: the values of the cofactors $f_{s_i}(A,B,0)$ and $f_{co}(A,B,0)$ from the compacted truth table are stored in the k/m-LUT, the controlled inversion blocks implement folding.

The negative cofactors $f_{s_i}(A,B,0)$ and $f_{co}(A,B,0)$ in Equations 3.16 and 3.17 represent half of all possible values of the sum functions f_{s_i} and carry output function f_{co} of an n-bit adder, respectively. The output part of the truth table of an n-bit adder has 2^{2n+1} rows and $n+1$ columns, which corresponds to $2n+1$ 1-bit inputs (i.e. two n-bit arguments A and B and a carry input ci) and $n+1$ 1-bit outputs (i.e. n-bit sum output S and a carry output co) of an n-bit adder. By applying the inversion-based folding type II, such a truth table can be compacted to the table with 2^{2n} rows only (see Figure 3.11(a)). Consequently, the generation of n negative cofactors $f_{s_i}(A,B,0)$ and the negative cofactor $f_{co}(A,B,0)$ of an n-bit adder is possible using a look-up table with $k=n$ inputs and $m=n+1$ outputs. We shall refer to such a *multi-output LUT* as *k/m-LUT*. The implementation of the n-bit addition in the k/m-LUT according to the inversion-based folding type II is shown in Figure 3.11(b). The controlled inversion blocks that are placed at the inputs and outputs of the look-up table enable the generation of the expressions $f_{s_i}(\bar{A},\bar{B},1)$ and $f_{co}(\bar{A},\bar{B},1)$. It is clear that the implementation of the n-bit adder according to the inversion-based folding type II allows the size of the required multi-output LUT to be halved.

3.2 Cost metrics

This section deals with basic concepts behind the domain-oriented reconfigurable logic architectures that will be presented in detail in the next chapters. It also proposes a set of metrics enabling a comparison of such architectures with state-of-the-art FPGAs, and in this way, a verification of the concept of application domain specialisation proposed in Chapter 2. Two primary cost metrics are introduced: the metric based on the *implementation area* of a reconfigurable architecture and the metric based on the *area model* of a reconfigurable architecture.

3.2.1 Architectural concepts

In Chapter 2 we derived three basic classes of domain-oriented reconfigurable logic architectures that should be supported. We assume that such architectures are made *homogeneous*. The homogeneity means in this case that the entire array (see Section 1.4.1) of a domain-oriented reconfigurable logic core is built of the identical type of modules. We shall refer to such modules as *logic tiles*. A logic tile consists of a logic block and a portion of routing resources (e.g. the bottom and right parts of routing channels that cross over in a switch block, as shown in Figure 1.8). A homogeneous architecture is essential for several reasons, namely:

- It reduces the complexity of mapping tools (placement and routing tools in particular) since such tools do not have to account for functional differences between tiles.

- It reduces the design effort since the same type of basic building blocks can be used to assemble the complete array of a reconfigurable logic core.

- It matches the type and complexity of functions that are to be mapped onto a reconfigurable logic core (see the discussion in Section 1.6.1). (For example, since a reconfigurable logic core is meant to map only relatively small processing kernels, large embedded memories can be omitted.)

We also assume that logic blocks of a reconfigurable logic core are LUT-based. The look-up tables offer a high level of flexibility, which is essential if the mapping of different types of functions is expected (see Section 2.3.1, conclusion 3.). The logic block has a hierarchical structure. It consists of (in a bottom-up order): a *logic element*, a *logic block* and a *logic tile*.

3.2.2 Mapping cost

To compare different reconfigurable architectures we introduce a unified cost metric, which we shall call *mapping cost MC*. The mapping cost (in μm^2) is a cost of implementing a function (processing kernel) in a reconfigurable logic architecture, and is calculated as

$$MC = N_{LB} \times A_{LT}\,. \tag{3.18}$$

N_{LB} is the number of logic blocks[4] that are needed to implement a given function, and is established via a mapping experiment (manual or automatic). A_{LT} is the area of a logic tile in a (target) reconfigurable architecture, and is derived based on the VLSI implementation of the tile. We choose the logic tile area as a primary cost measure of a reconfigurable logic device since it has a strong impact on its other implementation parameters, such as power consumption and delay. The reason for that is that both power consumption and delay in FPGA devices are interconnect-dominated [47], and that a logic tile comprises the interconnect. Power consumption is determined by the capacitance (load) of interconnect wire segments, while the delay by the capacitance and resistance. Since the capacitance and resistance are functions of the wire segment length, they depend on the horizontal and vertical dimensions of the logic tile[5], and thus on the logic tile area (as the product of the logic tile dimensions).

Though accurate (for area), the comparison of reconfigurable architectures based on the above-discussed cost metric is not always possible. Firstly, the logic tile area of many state-of-the-art commercial FPGAs is not known. Secondly, an implementation of all proposed domain-oriented reconfigurable architectures is, due to a considerable design effort, not realistic. Finally, the tools that enable mapping of complex functions onto all reconfigurable architectures of interest are not

[4] We use the number of logic blocks rather the number of logic tiles since the former parameter is directly reported by mapping tools. Essentially, as far as the amount of logic resources only is concerned, $N_{LB} = N_{LT}$.

[5] Because of the programmable nature of the FPGA interconnect, this is true even if interconnect wires are laid out using multiple metal layers on top of the logic.

available. Therefore, we suggest an alternative method of comparison. In the first method, we calculate the mapping cost MC using the *VLSI implementation area* of a logic tile as a cost measure. Only those devices for which implementation data are available are compared in this way. Furthermore, dependent on the availability of mapping tools, mapping cost of simple or complex benchmark functions is considered. In the second method, we also calculate the mapping cost MC, but the area of a logic tile is estimated based on the *area model*. Below, both comparison methods are discussed in detail.

3.2.3 Implementation-based cost metrics

As indicated above, the implementation-based method of comparison of reconfigurable architectures relies on the VLSI implementation area of a logic tile. If dedicated mapping tools are available, the cost of mapping *complex* functions onto a reconfigurable logic architecture is calculated according to Equation 3.18. If the area of a logic tile in a reconfigurable architecture is known but mapping tools are not available, two additional cost metrics are used to compare the cost of mapping relatively *simple* functions. The first metric, that is *data-path mapping cost* MC_{DP}, is defined as the cost of implementing 1-bit of data-path in a reconfigurable architecture. MC_{DP} is calculated as

$$MC_{DP} = \frac{A_{LT}}{n}, \tag{3.19}$$

where n expresses the maximum number of bits of a basic data-path function (here: a binary addition) that can be implemented in a logic block of the architecture. The unit of MC_{DP} is $\mu m^2/bit$. The second metric, that is *random logic mapping cost* MC_{RL}, is defined as the cost of implementing a 4-input Boolean function in a reconfigurable architecture. MC_{RL} is calculated as

$$MC_{RL} = \frac{A_{LT}}{m}, \tag{3.20}$$

where m is the number of 4-LUTs in a logic block of the architecture. The unit of MC_{RL} is $\mu m^2/4-LUT$.

Note, that MC_{DP} and MC_{RL} reflect the *maximal (ideal) mapping efficiency* of the compared architectures. This is because both metrics are not influenced by the quality of mapping tools (the values of the parameters n and m are derived from the architecture of a single logic tile only). In practice, logic tiles are sometimes used for routing or may be blocked. Therefore, the usage of logic resources is usually lower than 100%.

We have selected two state-of-the-art commercial (general-purpose) FPGAs, that is Xilinx Virtex-E [118] and Altera APEX 20K400E [4], for our comparison. The VLSI implementation area of the logic tiles in both devices is known [77]. The availability of dedicated mapping tools for these FPGAs (i.e. Alliance package for the Xilinx device and Quartus II for the Altera device) allows the comparison of the mapping cost using complex benchmark functions. For our comparison we also selected three data-path-optimised FPGAs from academia. They are: Low Power Programmable Gate Array (LP-PGA II) from University of Berkeley [47], Computational Field Programmable Architecture (CFPA) from University of Toronto [61] and Reconfigurable Computing Array (RCA) from Hewlett-Packard Laboratories (now Elixent Ltd.) [69]. Due to lack of dedicated mapping tools, the latter FPGAs are used in a comparison based on MC_{DP} and MC_{RL} cost metrics. The implementation data of the selected state-of-the-art FPGAs are summarised in Table 3.1.

FPGA device	Technology	Implementation style	A_{LT} [μm^2]
Xilinx Virtex E	0.18 μm CMOS	full-custom	35462
Altera APEX 20KE	0.18 μm CMOS	full-custom	63161
LP-PGA II	0.25 μm CMOS	semi-custom	52779
CFPA	0.5 μm CMOS	full-custom	63240
CHESS	0.35 μm CMOS	full-custom (?)	45000

Table 3.1. The implementation data of logic tiles of the commercial and academia FPGAs.

3.2.4 Model-based cost metrics

Several different models for estimating an area of a FPGA logic tile have been proposed in academia. Their common characteristic is an assumption that the logic tile area is the sum of the logic block area A_L and the routing area A_R, that is, $A_{LT} = A_L + A_R$.

For example, Rose *et al.* (1990) [87] calculate the logic block area as the sum of all LUT memory bit areas and a fixed area that is needed for a LUT selection circuitry, a flip-flop, and an extra control logic. Rose estimates the routing area based on the number of tracks in a routing channel of a given reconfigurable architecture. The tracks are separated by a fixed routing pitch, which depends on the area of a configuration (control) memory cell. The tracks are assumed to be laid out on the same metal layer. The problem of Rose's model is that it requires precise information about the total number of routing tracks. Due to a heterogeneous type of routing resources in modern FPGA devices, establishing the exact number of tracks per routing channel in such devices is often difficult. Brown *et al.* (1992) [19] use the same logic block area model as Rose. However, they assume that the routing area is a linear function of the logic block pin count, as suggested by Hill [54]. The limitation of Brown's model is that it uses the area cost per pin (expressed in the memory cell areas) which has been established for a specific set of architectures, and is thus inappropriate in a generic case. The alternative area model that has been proposed by Betz *et al.* (1999) [14] is much more accurate than the models described above. The Betz model, applied to the placement and routing tool called VPR [14], is based on the area count of minimum-width transistors. All basic building blocks of an FPGA architecture are characterised a priori with this parameter. Though accurate, the use of Betz's model is restricted to only those FPGA architectures which can be modelled in VPR.

To overcome the limitations of the existing models, for our comparison we have chosen a simple yet reliable area estimation technique adopted from He & Rose (1993) [52]. According to this technique, the area contribution of logic A_L depends on the number of LUT memory bits N_{lmb}, whereas the area contribution of routing A_R on the number of logic block pins P. The latter is based on the assumption of Hill [54] that each logic block pin has a fixed area contribution to the total area of a logic tile. The direct correlation between the logic block pin count and the routing cost of a logic block has been confirmed by others [37, 26]. Note, that this logic tile area estimation technique is *implementation-independent*. Furthermore, it is also *interconnect architecture-independent* since the routing cost is estimated based on the logic block pin count rather than on the precise information about the interconnect structure. In this way, a fair comparison of different FPGAs, even without knowing their architectural and implementation details, is possible.

He's & Rose's model has been originally applied to relatively simple FPGA structures, in which all pins of the logic block are equally important. To account for the functional diversity of the logic block pins, which is common in modern FPGAs, and thus for the difference in the cost, we introduce a pin *weighting factor* w_i. Consequently, we also replace the pin number P in our model with the *weighted pin number* P_w, such that $P_w = \sum_{i=1}^{P} w_i \cdot p_i$. The weighting factor w_i reflects the difference in the routing resource cost induced by a logic block pin p_i (where $i = 1, \ldots, P$) in comparison to the cost induced by a random logic pin in a logic block of a traditional (general-purpose) LUT-based FPGA. We identify five types of pins in the FPGA architectures of interest. They are:

- random logic pin,
- word-level data-path pin,
- bit-level data-path pin,
- carry pin,
- auxiliary pin.

Each pin is characterised by the unique value of the weighting factor w_i. We assume the weighting factor values as listed in Table 3.2, which are similar to the values suggested by Lewis in [26].

The *random logic pin* is a pin of the logic block in a traditional general-purpose FPGA architecture. Such a pin is connected typically to an input or output of

Pin type	Weighting factor w_i
Random-logic	1.0
Word-level data-path	0.5
Bit-level data-path	0.7
Carry	0.5
Auxiliary	0.6

Table 3.2. Weighting factors of logic block pins in different FPGA architectures.

a look-up table. Because the random logic pin serves as a reference, it is characterised by the weighting factor $w_i = 1$. Further, we distinguish two types of data-path pins, that is the word-level data-path pin and the bit-level data-path pin. The *word-level data-path pin* models the input and output data pins of a reconfigurable architecture in which the interconnect structure has the form of buses. The bus-like implementation implies that the word-level control is applied to the switches in switch blocks and output connection blocks and to the multiplexers in input connection blocks of such an interconnect structure. The word-level control of routing resources and the reduced flexibility of the connection blocks (due to the regularity of data-paths) decrease the cost of the reconfigurable interconnect. For that reason, the word-level data-path pins are assumed to have half of the cost of the random logic pins [26], that is w_i=0.5. This is rather a conservative estimate [26]. Because of the dedicated, scarce routing resources that are associated with the input and output *carry pins*, such pins are also characterised by the weighting factor w_i=0.5. The *bit-level data-path pin* is similar to the word-level data-path pin except for the fact that it is associated with the routing resources that are bit-level rather than word-level controlled (i.e. each switch and each multiplexer have independent control bits). To reflect the difference in the interconnect cost, such pins are characterised by the weighting factor w_i=0.7. This is also a conservative estimate. Finally, the *auxiliary pin* models the secondary ports of a reconfigurable architecture (e.g. a shift input/output port in the Xilinx Virtex II logic block architecture [119]). The auxiliary pins are usually connected to the dedicated routing resources. The weighting factor w_i=0.6 is assigned to this type of pin.

Equations 3.21 and 3.22 show the way of estimating the logic block area A_L and the routing area A_R of a logic tile according to the proposed model. α and β are proportionality coefficients.

$$A_L = \alpha \cdot N_{lmb} \tag{3.21}$$

$$A_R = \beta \cdot P_w \tag{3.22}$$

Because the exact ratio between the logic area A_L and the routing area A_R is difficult to establish, the value of the coefficients α and β is unknown. Therefore, instead of using one cost metric we use two independent cost metrics *mapping* *cost with respect to logic* MC_L and *mapping cost with respect to routing* MC_R, which are calculated as follows.

$$MC_L = N_{LB} \times N_{lmb} \tag{3.23}$$

$$MC_R = N_{LB} \times P_w \tag{3.24}$$

The parameter N_{LB} in Equations 3.23 and 3.24 is the number of logic blocks that are needed to implement a benchmark function. Simple macro-functions are selected as a benchmark set. The parameter N_{LB} for such functions is derived using dedicated macro-generators (commercial FPGAs) or is found by performing a manual mapping (the proposed domain-oriented FPGAs).

Note, that the reduction of the LUT memory bits N_{lmb} in the implementation of a given function has a double advantage. First, it reduces the area contribution of logic (see Equation 3.23). Second, it also reduces the configuration time of a device since a smaller amount of data must be up-loaded to the configuration (control) memory.

We have selected three state-of-the-art commercial FPGAs as the reference in the model-based comparison. The FPGAs are: Xilinx Virtex II [119], Altera Stratix [6], and Atmel AT40K [8]. The FPGAs are characterised with the parameters N_{lmb} and P_w that are listed in Table 3.3.

FPGA architecture	Number of pins					N_{lmb}	P_w
	Inputs	Outputs	Carry input	Carry output	Auxiliary		
Xilinx Virtex II	40	16	2	2	6	128	62
Altera Stratix	40	20	1	1	1	160	62
Atmel AT40K	4	2	0	0	0	16	6

Table 3.3. Characterisation of the logic cost (N_{lmb}) and the routing resource cost (P_w) in the selected state-of-the-art commercial FPGAs.

3.3 Conclusions

In this chapter we surveyed basic definitions and properties of Boolean and arithmetic functions that are of importance for the work presented in this thesis. We also discussed two novel properties of a binary addition that simplify its LUT-based implementation. The properties describe the method of folding the sum function of a full adder (the inverting-based folding type I) and the method of folding the sum functions and carry output function of an n-bit adder (the inverting-based folding type I). The result of such folding is a factor of two reduction in the number of LUT memory bits. Some implementation aspects of the proposed properties were also considered.

To enable a comparison of the domain-oriented reconfigurable logic architectures with the selected state-of-the-art FPGAs, we proposed a cost metric called the mapping cost. The mapping cost of a given function is calculated based on the number of logic blocks that are needed to implement the function and on the area of a logic tile. Dependent on the available data, the logic tile area is the VLSI implementation area of the tile or the estimate of that area derived based on the model.

Chapter 4

DATA-PATH-ORIENTED RECONFIGURABLE ARCHITECTURE

In this chapter we present the first of the domain-oriented reconfigurable logic architectures, namely the *data-path-oriented architecture*. The architecture is optimised towards the mapping of processing kernels that are data-path dominated. The *modified data-path-oriented architecture*, which is characterised by the word-level control applied to its logic and routing resources, is also presented.

4.1 Introduction

4.1.1 Characteristics of the application domain

The implementation of kernels dominated by the data-path type of processing reveals a *regular nature* of data-paths. The regularity is expressed both in a similar type of logic components as well as in the structured interconnect between them. The data-path implementation is also *modular*, that is, it consists of a number of word-level units with clearly distinguishable boundaries. We will refer to such units as *macro-blocks* or *macro-components*. The macro-blocks are usually implemented in a *bit-sliced* way. This means that *n* identical basic blocks (slices) are used to implement the functionality of an *n*-bit macro-block. The 1-bit slices are abutted together in one direction. This type of implementation implies specific directions for the data and control flows, typically being orthogonal to each other. The connections between bit-slices, that is connections within macro-blocks, are usually short (local), while the connections between macro-blocks may be of any length. We will refer to the communication featuring such types of connections as *intra-macro* and *inter-macro* communication, respectively. This is illustrated in Figure 4.1, where an example of a 16-bit data-path structure is given.

In the data-path-dominated processing kernels, multi-bit (i.e. word-level) arithmetic functions and multi-bit logic functions are the most common; simple control functions, which sometimes are also present in such kernels, are implemented as random logic.

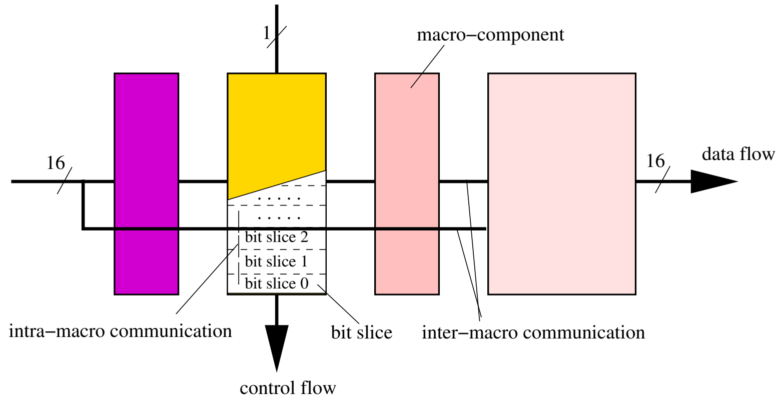


Figure 4.1. An example of a 16-bit data-path.

4.1.2 State-of-the-art

Though data-paths feature multi-bit processing, most LUT-based FPGAs support an implementation of only single-bit operations in their basic functional elements. For example, a 1-bit arithmetic addition is typically implemented with two small LUTs (e.g. 3-LUTs) or one bigger LUT (e.g. 4-LUT). If two 3-LUTs are used, they implement the sum *s* and carry output *co* signals of a 1-bit adder. The LUTs may have independent inputs (e.g. as in Altera FLEX [5]) or shared (connected) inputs (e.g. as in Atmel AT40K [8]). If a single 4-LUT is used for the adder implementation, such a LUT implements the adder sum signal *s*. The carry output signal *co* is generated in a dedicated carry circuitry (e.g. as in Xilinx Virtex-E [118]). The first and second implementation method are shown in Figures 4.2(a) and (b), respectively.

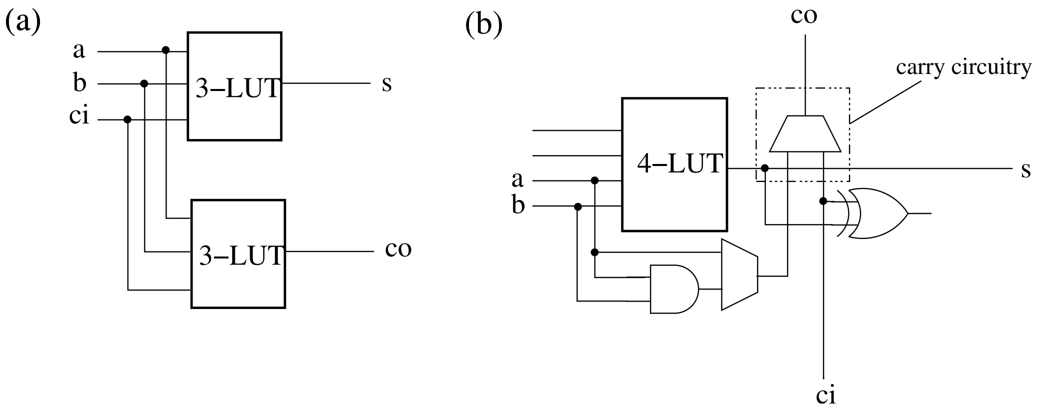


Figure 4.2. Typical methods of implementing a 1-bit addition operation in state-of-the-art FPGAs: (a) an implementation using small LUTs with shared inputs, (b) an implementation with a bigger LUT and a dedicated carry circuitry.

The problem of the above-mentioned implementations is their high cost. Note, that in both cases in total 16 configuration bits are needed for the implementation of a 1-bit addition only. Also, if coarse LUTs are used, a larger delay of the selection multiplexer (see Figure 3.1) reduces the overall performance of the mapped function. Applying the carry-select addition method (e.g. as in Altera Stratix [6]) diminishes this problem slightly.

The multi-bit arithmetic functions and multi-bit Boolean functions are implemented using several look-up tables. Complex Boolean functions, having many inputs and typically a single binary output, are implemented according to Shannon's expansion (see Figure 3.2).

4.2 Applying the inversion-based folding type I

The problems of the full adder implementation in the described state-of-the-art LUT-based FPGAs can be avoided if the inversion-based folding type I (proposed in Section 3.1.2) is applied. In Figure 4.3, the suggested implementation of a 1-bit adder according to the folding property is shown.

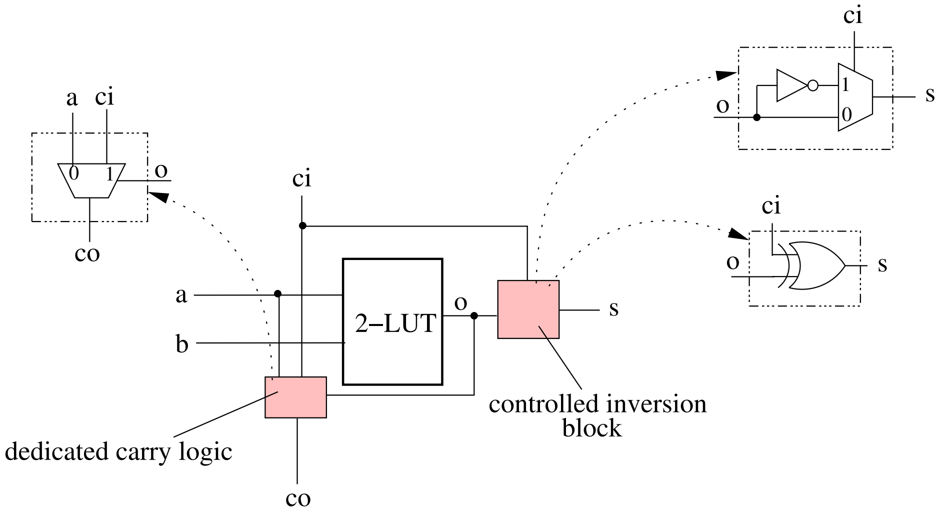


Figure 4.3. The implementation of a 1-bit addition based on the proposed inversion-based folding type I.

The 2-LUT generates the values of the sum signal *s* of the full adder. Dependent on the polarisation of the carry input signal *ci*, the sum signal is generated directly ($ci = 0$) or by the inversion of the LUT output ($ci = 1$). The inversion is performed in the controlled inversion block which two alternative implementations have been illustrated. The dedicated carry logic [123] takes care for the generation of the carry output signal *co* of the full adder.

The key benefit of the proposed implementation of the LUT-based addition is a considerable reduction in cost. First of all, the total number of LUT memory bits that is required for such an implementation is a *factor of 4 lower* than the number of bits needed in the state-of-the-art implementation approaches discussed in Section 4.1.2. Furthermore, the critical path delay, which depends on the delay of the carry signal c_{out}, is relatively short because the dedicated carry logic is used. The delay of the sum signal s is also short since it is determined by the delay of a 4:1 multiplexer in the 2-LUT (rather than the delay of a 8:1 multiplexer or a 16:1 multiplexer in the LUT implementations shown in Figures 4.2(a) and (b), respectively).

4.3 Logic element

The logic element is a basic component of the data-path-oriented reconfigurable logic architecture. The logic block structure is similar to the structure described in Section 4.2. The choice of such a structure is justified by its cost-efficiency in implementing arithmetic functions (via the inversion-based folding type I) and by its capability of implementing random logic functions (via a 2-LUT).

To allow an implementation of arithmetic functions other than a 1-bit addition only, the original 2-LUT structure from Figure 4.3 is augmented with some extra elements as shown in Figure 4.4. The AND gate at the LUT input enables the mapping of a basic cell of an array multiplier. It implements the logical AND-operation on the data input signal a_i and the multiplicand bit carried by the signal y. The XOR gate at the second LUT input is used to implement addition or subtraction operations, which can be chosen dynamically by assigning the proper value to the signal z (i.e. $z = 1$ for a subtraction operation and $z = 0$ for an addition operation). The signal z and the data signal b_i are inputs of the XOR gate.

As it has been indicated by the results of the application-domain characterisation in Section 2.3.1 (conclusion 4), a considerable number of multiplexers can be identified in the mapped designs. The importance of having support for an efficient implementation of multiplexers in FPGA structures has also been confirmed by Agarwala [2] and Cherepacha [26]. For these reasons, we assume that the logic element of the data-path-oriented architecture must allow mapping of a 2:1 multiplexer. However, the multiplexer function (i.e. $f_{mux} = a \cdot \overline{x} + b \cdot x$, where a and b are multiplexer inputs, and x is a selection input) cannot be folded using the inversion-based folding type I. Therefore, the 2:1 multiplexer function cannot be directly implemented using a 2-LUT. To overcome this limitation, a dedicated 2:1 multiplexer LMUX is placed in parallel to the 2-LUT (see Figure 4.4). The multiplexer has its first primary input connected to the 2-LUT output, and the second primary input connected to the b_i input of the 2-LUT; the selection input of the multiplexer is connected to the signal x.

The complete structure of the logic element is shown in Figure 4.4. The logic element has two primary inputs a_i and b_i, the carry input ci_i, and three secondary

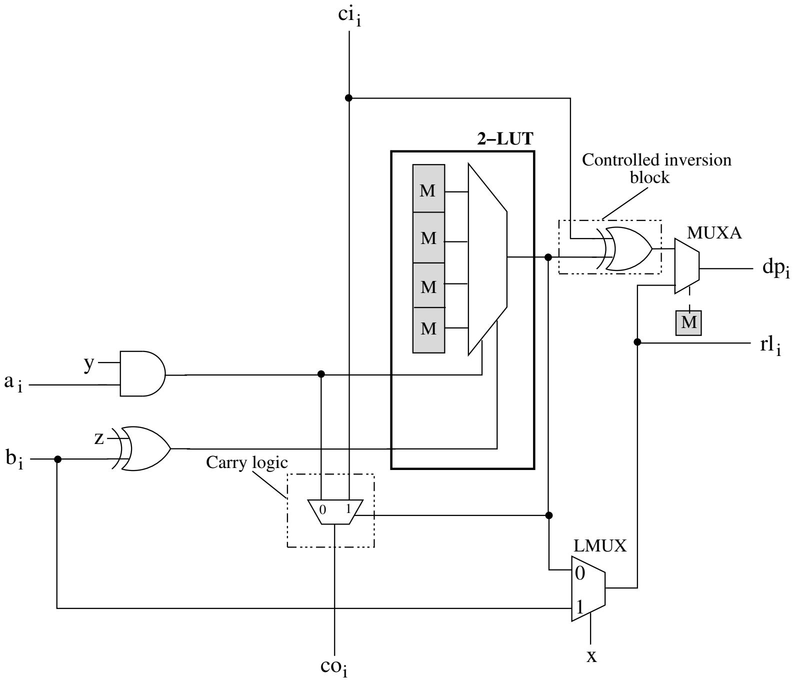


Figure 4.4. Logic element of the data-path-oriented reconfigurable logic architecture. The logic element implements a bit-slice of a data-path. (M is a configuration memory bit.)

inputs x, y and z. It also has two primary outputs, that is, a data-path output dp_i and a random logic output rl_i, and a carry output co_i. The data-path output dp_i is used when the logic element is configured as a bit-slice of a data-path. In such a case, at the higher level of hierarchy (i.e. in a logic block), a *multi-bit* result is produced. The output rl_i is used when a random logic function is mapped. This yields a *single-bit* result at the higher level of hierarchy.

The outputs of the multiplexer LMUX and the gate XOR in the controlled inversion block are multiplexed in the multiplexer MUXA. The multiplexer plays a twofold role: it allows the output signal of the 2-LUT to bypass the controlled inversion block (if a Boolean function is mapped and $x = 0$), and it allows implementations of multi-bit functions other than arithmetic functions (e.g. multi-bit Boolean functions and data-path multiplexers). Note also, that the implementation of the controlled inversion block with the XOR gate is chosen. The carry circuitry has the structure identical to that shown in Figure 4.3.

4.4 Logic block

The logic block constitutes the next level of hierarchy of the data-path-oriented reconfigurable logic architecture. At this level, the implementation of multi-bit arithmetic functions and Boolean functions with more than two inputs is enabled.

4.4.1 Basic concept

In Section 2.1.2 we showed that different natures of data-path and random logic functions impose conflicting requirements on the FPGA architecture. This causes that the design of a cost-efficient reconfigurable logic device that supports both types of functionality is not trivial.

The essence of the proposed solution is the observation that the different nature of the mapped functions can be preserved without sacrificing the mapping efficiency of the final architecture. This is achieved by exploiting characteristic properties of the mapped functions. For example, data-paths have a regular bit-sliced structure and random logic does not (see Section 4.1.1). Furthermore, data-path functions produce a multi-bit output signal using relatively simple (fine-grain) logic elements, whereas typical random logic functions produce a single-bit output signal and benefit from more complex (coarser) logic elements (see the detailed discussion in Section 2.1.2). Therefore, assuming that the complexity of a LUT-based logic element is a function of its look-up table size (see Section 3.2.4), and thus the number of LUT configuration bits (see Section 3.1.1), the problem of designing a cost-efficient data-path-oriented reconfigurable logic architecture is the problem of an efficient utilisation of the available LUT configuration bits.

The architectural concept that follows such reasoning is illustrated in Figure 4.5. Assuming that a 4-LUT is a good candidate for random logic mapping [86], and exploiting the fact that a 2-LUT suffices to implement basic arithmetic functions (see Section 3.1.2), we determine the optimal complexity of our logic block as being 16 configuration bits. The key observation is that these 16 configuration bits can be treated as either one group (an equivalent of a 4-LUT) if random logic functions are to be mapped, or they can be decomposed into four groups of four configuration bits (an equivalent of four 2-LUTs) if data-path functions are to be mapped. To realise that, for the random logic mapping use is made of Shannon's expansion (see Equation 3.2), while for the data-path mapping the inversion-based folding type I (see Theorem 3.2) is exploited.

The reconfigurable logic block designed according to this concept allows the implementation of 4-bit data-path functions (i.e. operating on 4-bit arguments and producing a 4-bit result), and random logic functions with up to four logic variables (i.e. having four 1-bit inputs and producing a 1-bit output signal). In that sense, the proposed structure of the logic block can be viewed as offering a *mixed-level granularity* (in contrast to the fine-level and course-level granularity of classical reconfigurable architectures).

4.4.2 Structure in detail

The detailed structure of the data-path-oriented logic block is shown in Figure 4.6. The logic block consists of four logic elements (bit-slices) which structure has been discussed in Section 4.3 (the 'Selection' block, which is shown as a black

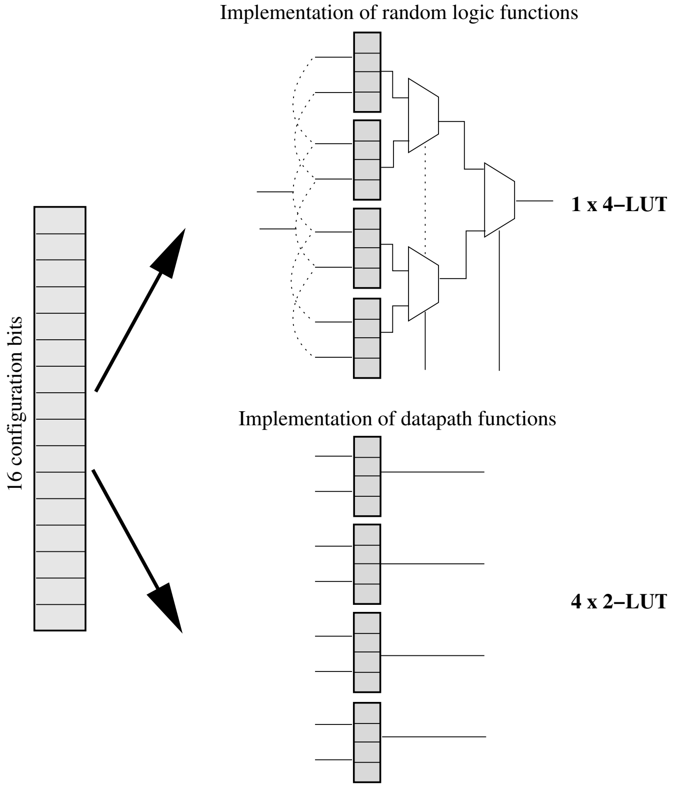

Implementation of random logic functions
1 x 4–LUT
16 configuration bits
Implementation of datapath functions
4 x 2–LUT

Figure 4.5. The architectural concept behind the logic block targeting data-path-oriented applications. Dependent on the functional mode, the configuration bits can be organised to implement a 4-LUT (random logic) or four 2-LUTs (data-paths).

box in each logic element, contains a multiplexer MUXA depicted in Figure 4.4). The 4-bit granularity is chosen for the logic block as such granularity has been found to be the best for the data-path mapping [26][95].

The logic block has eight primary inputs $in_1 \ldots in_8$ and three secondary inputs t_1, t_2, t_3. The secondary input t_1 also plays a role of a carry input *ci*. The logic block has four primary outputs $out_1 \ldots out_4$ and a carry output *co*. The primary inputs of the logic block are connected to the primary inputs of all logic elements via the input selection block. The data-path outputs dp_i of successive logic elements are fed directly to the output selection block. In contrast, the random logic outputs rl_i of the logic elements are merged together in the global multiplexers MUX1, MUX2 and MUX3. The output of the multiplexer MUX3 is also directed to the output selection block. Finally, the flip-flop block allows the outputs of the logic block to be registered if necessary.

The secondary inputs t_1, t_2, t_3 of the logic block provide three signals which are multiplexed with the static signals ‘0’ and ‘1’ established by local configuration bits. As a result, three global signals *x*, *y*, and *z* (see Figure 4.6) are generated. The signals *x*, *y*, *z* play the role of global control signals and are distributed to all logic elements. This type of connectivity is typical for data-paths [26]. The signal *x* is also connected to the ci_1 input of the first logic element. The first logic element

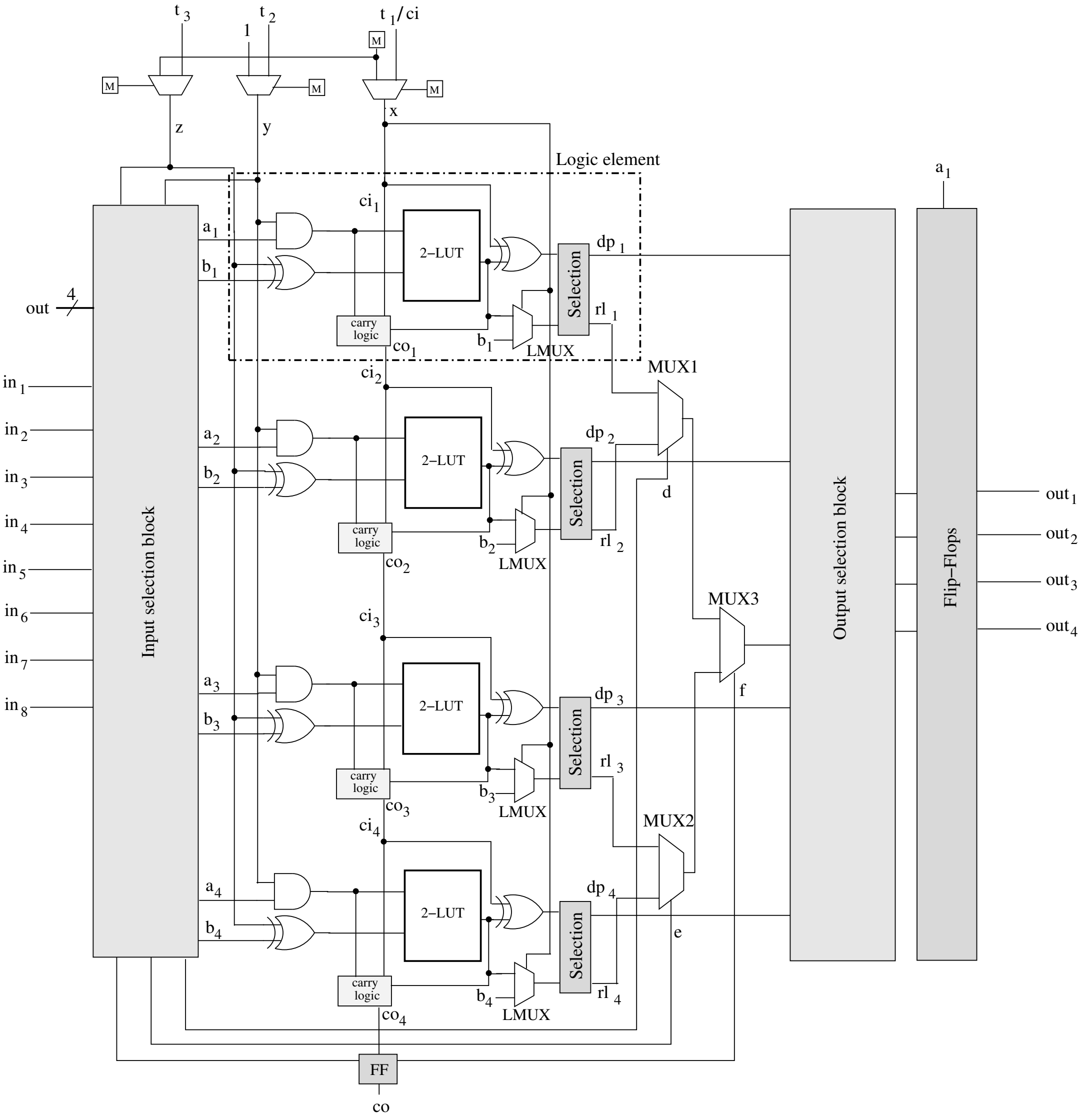


Figure 4.6. The architecture of the data-path-oriented logic block.

produces a carry output signal on its co_1 output, which is connected to the ci_2 input of the second logic element, and so on. In this way the carry chain in formed.

The meaning of the signals x, y, z depends on the functional mode of the logic block. This is reflected in Table 4.1, where possible settings of these signals are given. The sharing of the same set of inputs (i.e. t_1, t_2, t_3) for receiving several different signals allows the reduction of the logic block pin number, and consequently the reduction of the routing resource complexity (see Section 3.2.4).

Operation	**x**	**y**	**z**
Boolean (single-bit and multi-bit)	–	1	0
Addition with carry	ci	1	0
Addition without carry	0	1	0
Subtraction with carry	ci	1	1
Subtraction without carry	1	1	1
Addition/Subtraction with carry	ci	1	t_3
Addition/Subtraction without carry	0/1	1	t_3
Multiplication with carry	ci	t_2	0
Multiplication without carry	0	t_2	0
Multiplexing	t_1	t_2	t_3

Table 4.1. The setting of the global signals x, y, z dependent on the type of operation implemented in the logic block. For the detailed description of the functional modes of the logic block see Section 4.5.

The function of the *input selection block* is to define the connections between the primary inputs and outputs of the logic block, the primary inputs of logic elements, and the control inputs of the global multiplexers in the logic block dependent on the functional mode of the logic block. This is implemented by the structure shown in Figure 4.7(a). The structure comprises two sets of multiplexers. The first set of multiplexers defines the connections between the primary inputs $in_1 \ldots in_8$ and outputs $out_1 \ldots out_4$ (feedback signals) of the logic block, and the pairs of the primary inputs a_1, $b_1 \ldots a_4$, b_4 of successive logic elements. The first layer of multiplexers in the first multiplexer set increases the routing flexibility of the architecture (see details Section 4.6.1). The second layer of multiplexers in this set takes care for the proper distribution of the input signals of the logic block to the primary inputs of the logic elements. The way of distributing the signals depends on the functional mode (i.e. the data-path or random logic mode) of the logic block (see details in Section 4.5). The second set of multiplexers of the input selection block implements the connections between the inputs in_3, in'_7, in'_8 and the control inputs d, e, f of the global multiplexers MUX1, MUX2 and MUX3 of the logic block (see Figure 4.6). This type of connectivity is essential when multiplexers or single-output Boolean functions are mapped.

By analogy, the *output selection block* defines the connections between the outputs

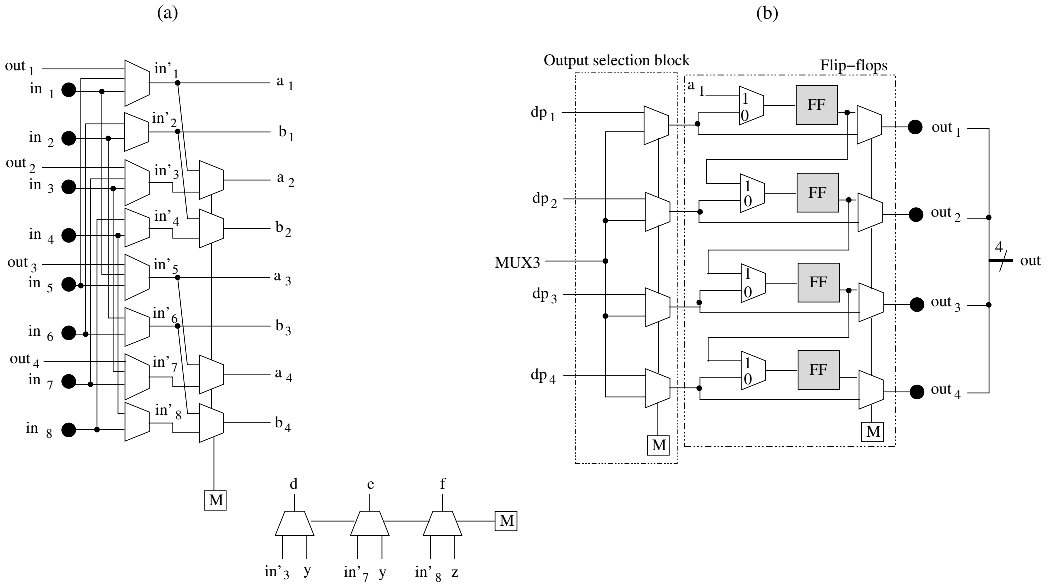


Figure 4.7. The selection blocks for: (a) inputs, (b) outputs (for the sake of clarity, the flip-flop block is also shown). The primary inputs and outputs of the logic block are marked with bold dots.

of the logic elements and the outputs of the logic block. As shown in Figure 4.7(b), a set of 2:1 multiplexers is used for this purpose. The multiplexers select either data-path outputs $dp_1 \ldots dp_4$ or a random logic output available at the output of the global multiplexer MUX3. Similarly to the first stage of multiplexers of the input selection block, the output selection block is also used to increase the routing flexibility. This aspect is discussed in detail in Section 4.6.1.

4.5 Functional modes

The proposed logic block has two primary functional modes, that is:

- *Data-path mode* in which a 4-bit result is produced using the data-path outputs dp_i of the logic elements; the 2-LUTs of all logic elements in the logic block are configured to implement the same function.
- *Random logic mode* in which a 1-bit result is produced using random logic outputs rl_i of the logic elements and the global multiplexers MUX1, MUX2, and MUX3 of the logic block; typically, the 2-LUTs of the logic elements in the logic block implement different logic functions.

We assume that the logic block can be configured to operate only in one of the above-mentioned modes at the same time. Therefore, if the number of logic elements required to implement a given function is smaller than four logic elements that are available in the logic block, the unused logic block resources will be waisted. Though this might have an impact on the utilisation of the reconfigurable logic array (slightly more logic blocks might be required), it reduces the complexity of the control structure in each logic block, and thus the logic block cost. Below, different configurations of the logic block in each of the functional modes are discussed in detail. The examples of such configurations are shown in Figure 4.8.

4.5.1 Data-path mode

Addition/Subtraction. The logic block can be configured to implement up to 4-bit addition or subtraction operations. The type of operation can be determined statically (by assigning '0' for an addition and '1' for an subtraction to the global control signal z) or dynamically (by connecting the global signal z to an external control signal available on the input t_2 of the logic block). A 2-LUT in each logic element stores half of the bits from the full adder truth table (see Section 3.1.2). If a binary addition is implemented, the AND and XOR gates in the logic elements are unused (i.e. $y = 1$ and $z = 0$). If a binary subtraction is to be implemented, the XOR gate in each logic element inverts one of the adder arguments, that is $z = 1$. The dedicated carry logic with the carry input signal assigned to the input t_1 implements the carry path. For the arithmetic operations with the word-size of arguments smaller than 4-bits, proper programming of the 2-LUTs guarantees that the resultant carry output signal can be directed to the carry output *co* of the logic block. An alternative implementation is the change of the application code (an addition on multiples of 4-bits). The output signals of the addition or subtraction operations are available on the data-path outputs dp_i of each logic element.

Multiplication. The logic block supports the implementation of an unsigned array multiplier with the ripple-carry addition [80]. A 4-bit section of such a multiplier (i.e. four cells) can be implemented in one logic block. For this purpose, each logic element is configured as a binary adder ($z = 0$) with an AND gate on one of its inputs. The global signal y, which is one of the inputs of the AND gate, provides the value of the multiplicand bit. In this mode, four partial product signals are generated on the data-path outputs dp_i, and the carry output signal is available on the *co* output of the logic block.

Data-path multiplexers. The data-path multiplexers are characterised by multi-bit inputs and outputs. Such multiplexers are implemented using LMUX-es and data-path outputs dp_i of the logic elements. The selection signals of the multiplexers are provided by the global signals x, y, z of the logic block (i.e. $t1$, $t2$, and $t3$ secondary inputs of the logic block) via the input selection block. For example, a 4-bit 2:1 multiplexer can be implemented in a single logic block.

Multi-bit Boolean functions. The logic block allows an implementation of Boolean functions with up to two 4-bit arguments (e.g. a 4-bit 2-input AND). The functions are implemented by programming 2-LUTs according to the truth table of the mapped Boolean function. As a result, all LUTs have the same con-

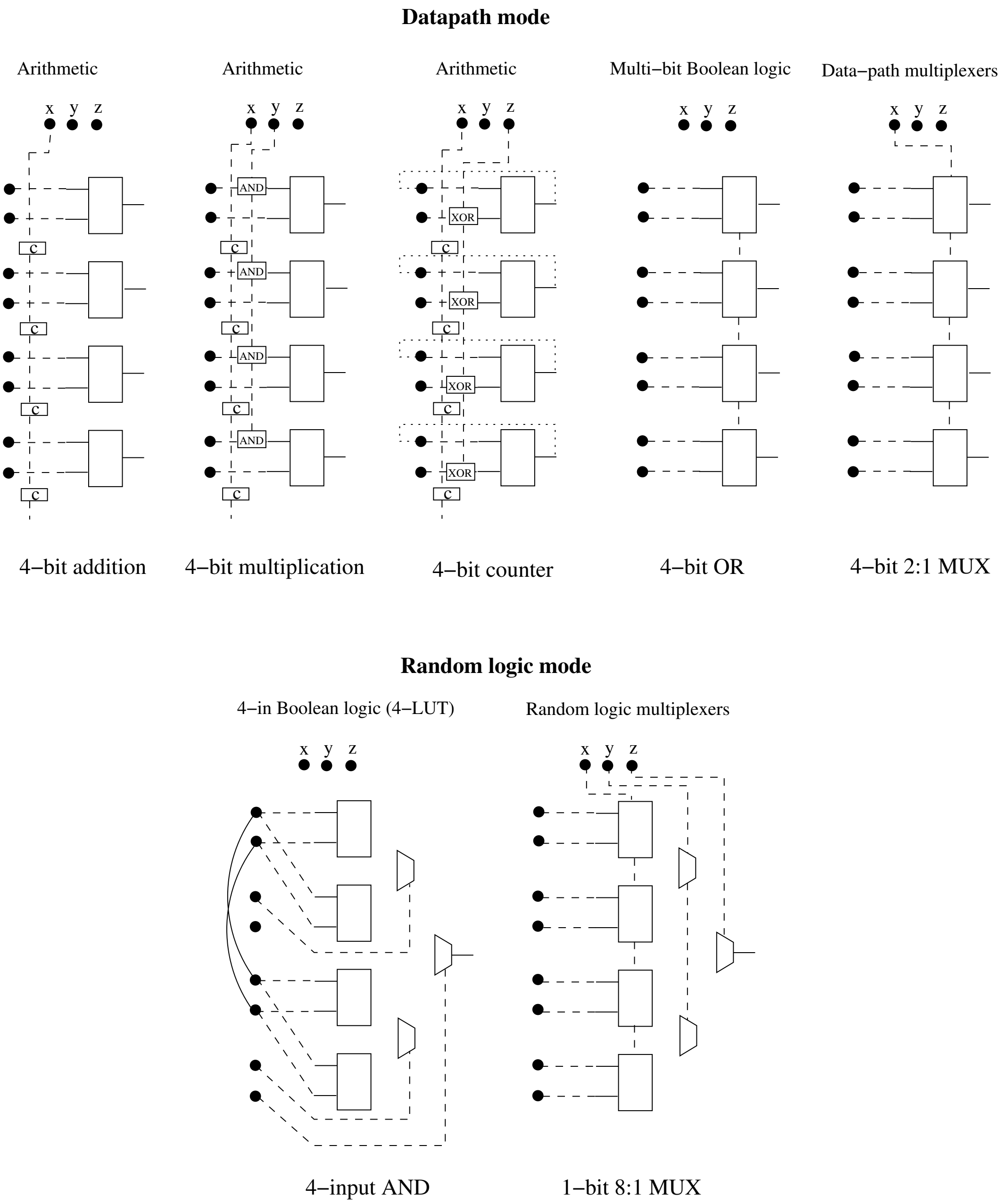


Figure 4.8. Examples of the configuration of the logic block in the data-path and random logic modes.

figuration. In each logic element, the selection multiplexers MUXA and MUXB select the signal generated in the 2-LUT and direct this signal to the data-path output dp_i.

Binary counters. The logic block allows the implementation of binary up- and down-counters of up to 4-bits. To enable that, the multiplexers of the first stage of the input selection block that are associated with the odd-numbered inputs of the logic block (i.e. i_1, i_3) are configured to select the registered feedback signals of the logic block; the multiplexers associated with the even-numbered inputs of the logic block (i.e. i_2, i_4) select the inputs signals, which are assumed to carry the static values. The binary representation of the static signal determines the counting step. Dependent on the type of binary counter, the logic elements of the logic block are configured as adders (counters up) or subtractors (counters down).

4.5.2 Random logic mode

Boolean functions. Boolean functions with up to four binary inputs (e.g. a 4-input OR) can be implemented in the logic block. A 4-input Boolean function is implemented according to Shannon expansion (see Section 3.1), and is mapped onto the set of four 2-LUTs and the set of global multiplexers MUX1, MUX2 and MUX3. The random logic outputs rl_i of the logic elements are used, and the final output is available on the output of the multiplexer MUX3.

Random logic multiplexers. In this mode, multiplexers with multiple binary inputs and a single binary output (e.g. an 8:1 multiplexer) can be mapped. Similarly to the implementation of the data-path multiplexers, the LMUX-es of the logic elements are used and the global signals x, y, z play the role of the selection signals of the mapped multiplexer. However, instead of the data-path outputs dp_i, the random logic outputs rl_i of the logic elements and global multiplexers MUX1, MUX2, MUX3 combining these outputs are used.

Shift registers. By chaining the flip-flops in the logic block, shift registers of different lengths (maximally four) can be implemented. The input of the mapped shift register is connected to a limited set of inputs of the logic block, while the output signal of the shift register (dependent on the shift register length) is available on one of the outputs of the logic block.

4.6 Interconnect

As we already mentioned, the key advantage of the proposed logic block is its ability to implement both coarse-grain data-paths functions and fine-grain random logic functions. The increased functional capacity of the architecture in the data-path mode (compared to traditional FPGAs) might have, however, a negative impact on the routing resource complexity. This is because a coarse logic block requires more inputs and outputs, and thus more interconnect wires to distribute signals. This translates into more configuration bits and larger control structures that are needed to select and distribute these signals [52]. In this section we show that by a careful design of the interconnect architecture the increase in the routing resource complexity, and thus the implementation cost of the proposed architecture, can be avoided.

4.6.1 Optimisation of the interconnect architecture

To optimise the interconnect structure we exploit the characteristics of data-path and random logic functions. In the implementation of the data-path functions, which operate on multi-bit arguments, relatively *many routing tracks* are needed (e.g. minimum 12 tracks for a 4-bit arithmetic function). However, only a *limited flexibility* is required to connect the input and output pins of the data-path elements to such tracks. This is because the regularity and bit-sliced structure of the data-path elements cause that the signals are distributed in a bus-like fashion (i.e. with a predefined order). Therefore, there is a strict correspondence between the consecutive pins of the data-path elements and the tracks they connect to. At the same time, in the implementation of the random logic functions relatively *few routing tracks* are needed (e.g. at least five tracks for a 4-input Boolean function). However, an irregular character of random logic functions enforces *high flexibility* between the input and output pins of the logic elements and their routing tracks.

We notice that the high flexibility of the input and output connection blocks[1] that is required by the random logic functions can be compensated by extra routing tracks. Such tracks are available if the implementation of the data-path functions must be supported at the same time. This is possible in our logic block because it has a different functional capacity (granularity), and thus a different number of active inputs and outputs, dependent on the functional mode.

The implementation of this concept is illustrated in Figure 4.9. Essentially, the input connection block together with the first stage of the input selection block (the input section) and the last stage of the output selection block together with the output connection block (the output section) are involved. The position of these blocks with respect to other components of the logic tile is shown in Figure 4.9(a), while in Figure 4.9(b) the implementation details are given. In Figure 4.9(c), the change in the connectivity between the input and output pins of the logic block and the routing tracks is explained. For the sake of simplicity, a narrower routing channel than the one required in practice is shown in the figures. The implementation of the input and output sections for the actual channel width is analogous.

In Figure 4.9(b), the routing channel consisting of three 4-bit buses A, B and C is shown. The input pins $in_1 \ldots in_8$ and output pins $out_1 \ldots out_4$ of the logic block connect to these buses via the *input and output connection blocks*, respectively. The connection blocks determine to how many and to which routing tracks the pins connect to. In the figure, each input pin connects to three out of 12 routing tracks, which yields the input connection block flexibility F_{ci}=0.25; each output pin also connects to three out of 12 routing tracks, which yields the output connection block flexibility F_{co}=0.25. This corresponds to the situation in the data-path

[1] The *flexibility of the input (output) connection block* (F_{ci} or F_{co}, respectively) is the ratio between the number of tracks an input (output) pin connects to and the total number of tracks in the routing channel [14].

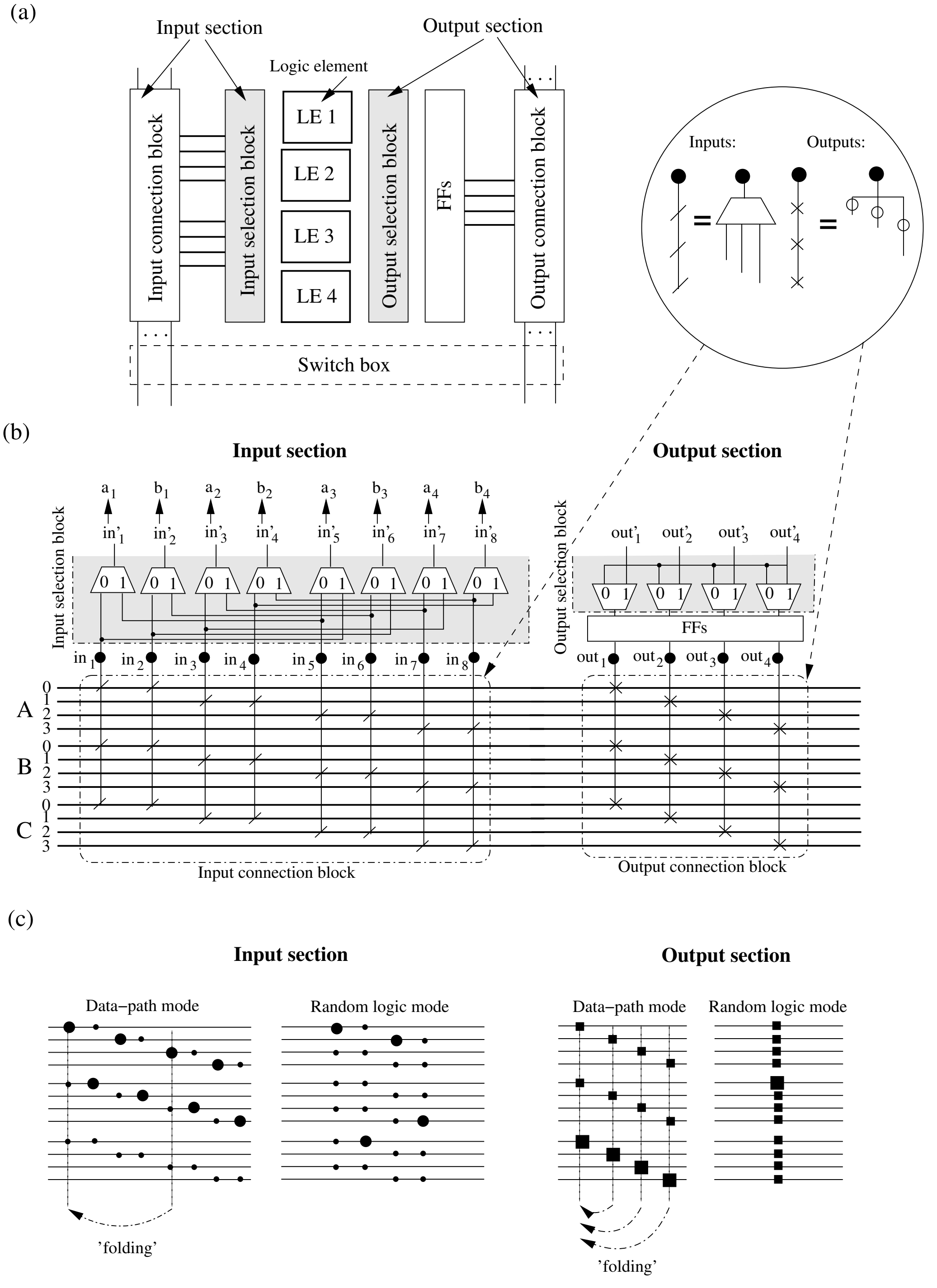


Figure 4.9. The modifiable connection block flexibility in the routing architecture of the proposed data-path-oriented reconfigurable logic device: (a) a logic tile (an abstract view), (b) the implementation of the input and output connection blocks with the exposed first and last stages of the input and output selection blocks, respectively, (c) the input and output connectivity to the routing tracks dependent on the functional mode (the bold markers show examples of the track assignment); the 'folding' mechanism of the data-path connection blocks is explained.

mode. In the random logic mode, the input and output connection block flexibilities can be modified because of the presence of the input and output selection blocks, respectively. The first stage of the input selection block, consisting of a set of 2:1 multiplexers controlled by independent configuration bits, allows a preliminary selection of four pairs of signals. The signals of each pair are identical (e.g. in'_1–in'_5, where $in'_1 = in_1$ and $in'_5 = in_1$). Because in this mode the number of relevant inputs decreases from eight to four, the input connection block flexibility per each relevant pin doubles (the *input connection block folding*) yielding the value of the input connection block flexibility F_{ci}=0.5. By analogy, the last stage of the output selection block, consisting of a set of 2:1 multiplexers controlled by a single configuration bit, allows the distribution of the output signal of a random logic function to any of the four outputs of the logic block. Because in this case the number of relevant outputs is reduced from four to one, the output connection block flexibility quadruples (the *output connection block folding*) yielding the value of the output connection block flexibility F_{co}=1. The flexibility of the input and output connection blocks obtained in the data-path and random logic modes and the associated folding procedure are explained in detail in Figure 4.9(c).

The described implementation of the connection blocks allows the reduction of the total capacitive load of the routing tracks. This is because smaller multiplexers and less switches are needed to implement the input and output connectivity, respectively. As a result, the delay and power consumption of the complete device can be reduced.

4.6.2 Complete interconnect architecture

In this section, we present an example of the interconnect structure for the logic block of the data-path-oriented reconfigurable device. The interconnect structure has been found through the analysis of the requirements of data-path and random logic functions. The data-path requirements were established via manual mapping experiments of a selected set of DSP kernels, whereas the requirements of random logic functions were obtained by the automated mapping of the MCNC logic synthesis benchmark set [120]. In the automated mapping flow, the following tools were used: SIS [90] for logic synthesis, FlowMap [32] for the LUT-based technology mapping, and a modified version of VPR [105] for placement and routing.

We found that the interconnect architecture with a channel width W=46 tracks suffices to map the aforementioned benchmarks. The architecture includes the following types of routing resources:

- general-purpose,
- special,
- direct.

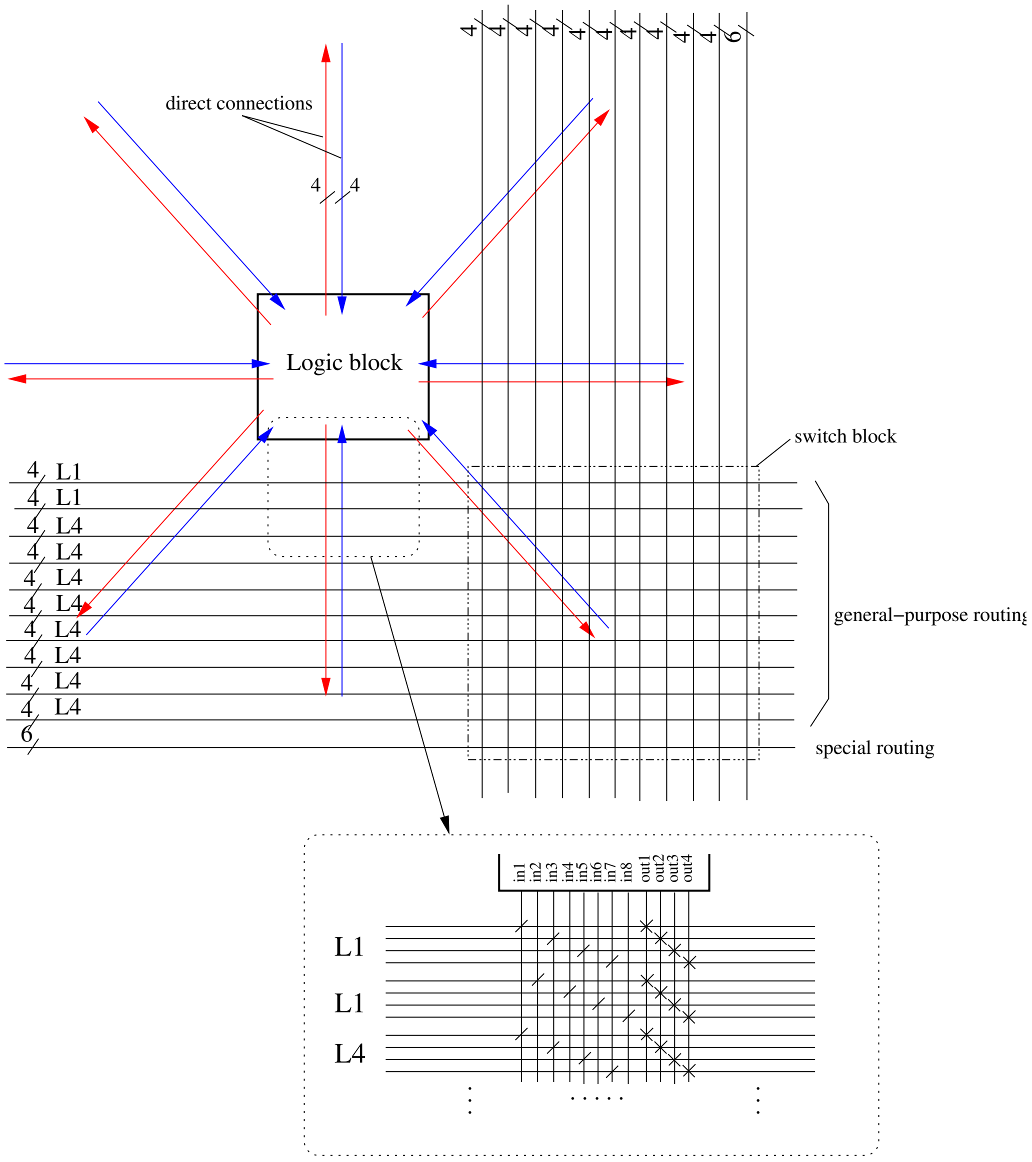


Figure 4.10. Routing resources of the proposed data-path-oriented reconfigurable logic architecture. A fragment of the input and output connection blocks is also shown.

The *general-purpose routing resources* include ten 4-bit buses: two buses of the length-one (L1)[2] and two sets of four buses of the length-four (L4). The *special routing resources* consist of six routing tracks that are used to route the secondary signals (connected to the pins t_1, t_2 and t_3 of the logic block) and the carry output signal (connected to the pin *co* of the logic block). In addition, the *direct routing resources* provide a set of fast local connections between neighbouring logic blocks. Each logic block produces a 4-bit direct output signal and receives 4-bit di-

[2] The length of a programmable wire segment (or a bus) is the number of logic blocks such a segment spans (before entering a next switch block) [14].

rect input signals. Such signals are distributed via direct connections in horizontal, vertical, and diagonal directions. The direct routing supports the implementation of functions which heavily rely on the local connectivity between their logic elements (e.g. intra-communication of data-path macro-blocks).

To increase the cost advantage of the proposed logic block, its connection blocks were implemented as described in the previous section, that is, having different input and output connection flexibilities dependent on the operating mode. Consequently, connection block flexibilities F_{ci}=0.25 and F_{ci}=0.5 were established for the connectivity of inputs in the data-path and random logic modes, respectively, and connection block flexibilities F_{co}=0.25 and F_{co}=1 for the connectivity of outputs in the data-path and random logic modes, respectively. In addition, the characteristics of data and control signal flows were exploited to determine a relative position of the connection blocks. As a result, the connection blocks were placed only at two rather than four sides of the logic block, similarly to the implementation of the connection blocks in the Atmel AT40K FPGA devices [8]. Also, the switch block implementation similar to that used in the Atmel devices was chosen. In such a switch block, a switching point between the crossing horizontal and vertical routing tracks is implemented with only three instead of six switches. Although this limits the flexibility of the general-purpose routing, it is sufficient for the mapping of data-path-oriented processing kernels that the presented architecture aims at.

The schematic diagram representing the routing resources of the proposed data-path-oriented reconfigurable logic architecture is shown in Figure 4.10.

4.7 Modified data-path-oriented reconfigurable architecture

The main advantage of the above-described data-path-oriented reconfigurable logic architecture is the ability of the efficient mapping of the data-path functions without sacrificing the random logic mapping capabilities. Though such functionality matches well the requirements of typical data-path-oriented applications (see Section 4.1.1), it can be optimised further if a target application domain is narrowed to the data-path-like functionality only. This happens, for example, if a reconfigurable logic fabric is used for the implementation of processing kernels with hardly or no random logic (e.g. small accelerators for multimedia applications as reported in [69]). We exploit this fact and propose several modifications to the original data-path-oriented reconfigurable logic architecture aiming at a further reduction of its intrinsic cost.

4.7.1 Basic concept

In Figure 4.11, a 4-bit data-path element and its bit-slice implementation are shown. By default, each bit slice implements the same function. If bit-slices are implemented using look-up tables (see Section 4.4), configuration bits of all look-up table are identical. This is obviously redundant. The way to overcome this is to replace independent sets of configuration bits with a single set only. This concept is illustrated in Figure 4.12, where four 2-LUTs share the same four configuration bits. The described here idea is similar to the concept of configuration bit-sharing that has been described in [26].

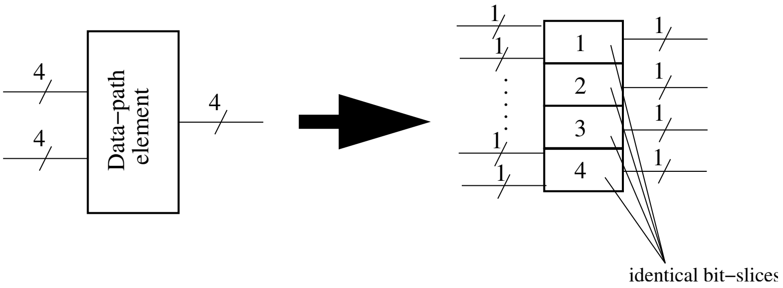


Figure 4.11. A 4-bit data-path element and its bit-slice implementation. Each bit-slice realises the same function.

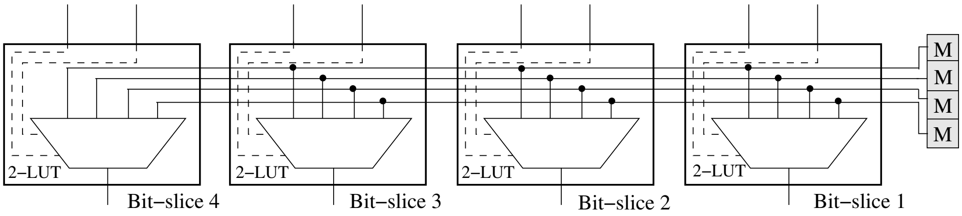


Figure 4.12. The concept of configuration bit sharing applied to the LUT-based data-path bit slices.

4.7.2 Logic block

We apply the above-described idea to the architecture of the data-path-oriented logic block from Figure 4.6. The configuration bits of each 2-LUT are removed, leaving 4:1 multiplexers only. Instead of having independent sets of configuration bits, all 4:1 multiplexers of the look-up tables use the same set of four configuration bits (see Figure 4.12), and thus implement the same function. Such an implementation of the look-up tables allows the reduction of the total number of LUT memory bits in the logic block by a factor of four (i.e. from 16 bits to 4 bits) compared to the original data-path-oriented logic block architecture.

Because the modified data-path-oriented logic block does not support mapping of 4-input Boolean functions (no 4-LUT functionality), the input selection block of the original logic block is removed and replaced by four 2:1 multiplexers. The multiplexers allow the selection of feedback signals (e.g. a binary counter mapping). Thus, four primary inputs of the logic block (i.e. in_1, in_3, in_5 and in_7)

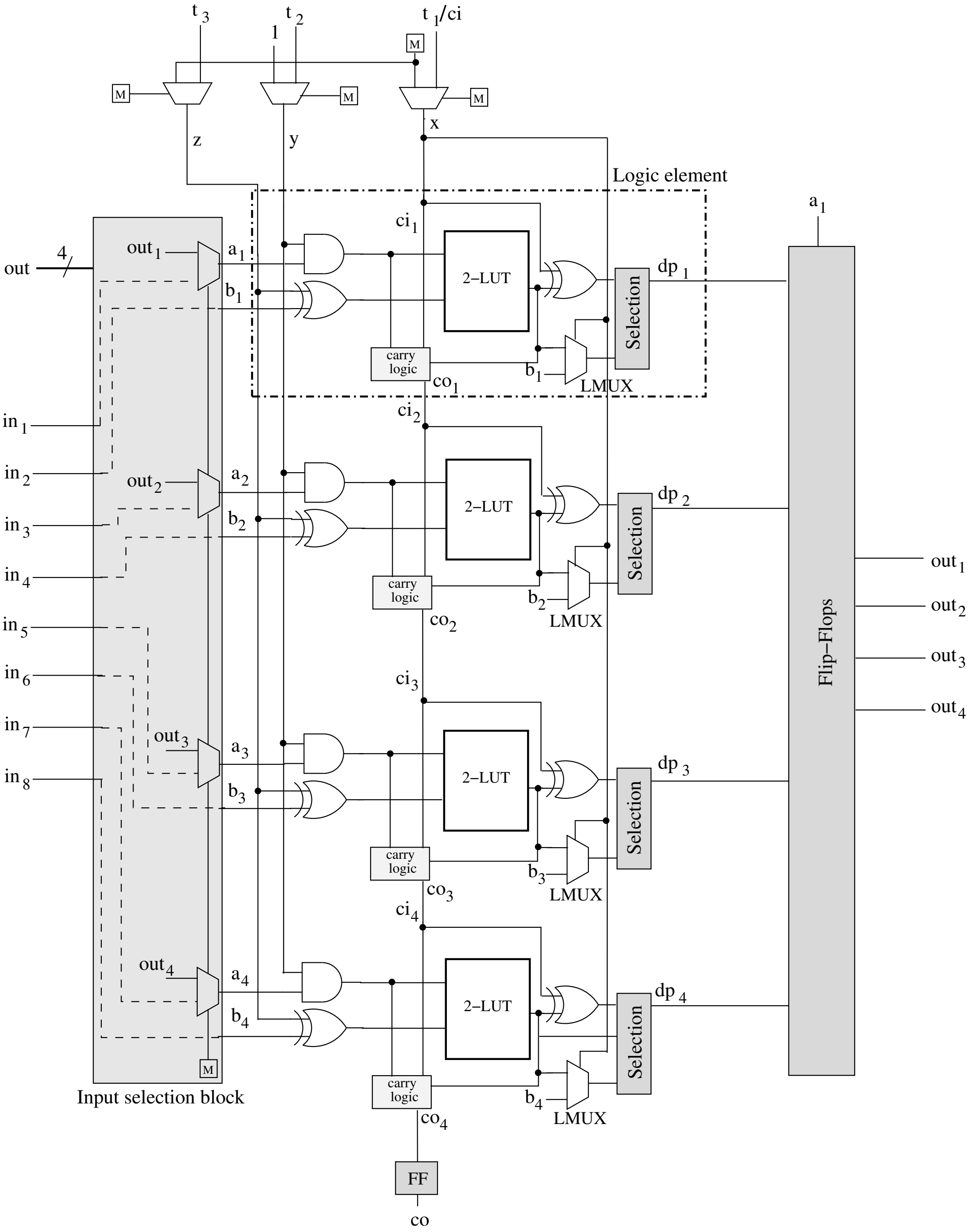


Figure 4.13. The architecture of the modified data-path-oriented logic block.

are connected to the inputs of the logic elements via the 2:1 multiplexers, while the other four primary inputs (i.e. in_2, in_4, in_6 and in_8) are connected directly. Since the mapping of random logic multiplexers does not make sense anymore, the global multiplexers MUX1, MUX2, MUX3 are also removed. As a result, the control signals d, e, f and random logic output signals $rl_1 \ldots rl_4$ dissappear. Furthermore, the output selection block, which selects between random logic and data-path output signals, is also removed. The other elements of the logic block remain unchanged and comprise the structure as shown in Figure 4.13.

4.7.3 Functional modes

As indicated above, applying the configuration bit sharing to the data-path-oriented logic block removes its random logic functionality. In consequence, the modified data-path-oriented logic block supports the implementation of 4-bit arithmetic functions, 2-input 4-bit Boolean functions, 4-bit data-path multiplexers, and 4-bit binary counters in the same way as was explained in Section 4.5. Note, that the functionality of the modified logic block resembles the functionality of an ALU operating on nibble-level arguments. However, compared to traditional ALU structures (e.g. such as described by Mead and Conway in [71]), the proposed here structure has a lower implementation cost.

4.7.4 Interconnect

The modified data-path-oriented logic block has the same type of routing resources as the original data-path-oriented architecture. The key difference between them is, however, the granularity of the control word which configures such resources. In the modified architecture configuration bits of the routing resources are shared among 4-bit routing elements (i.e. multiplexers, switches, etc.) rather than used to control them independently (see example in Figure 4.14). This is the so-called *nibble-level control*. This type of implementation allows a considerable reduction of the routing cost.

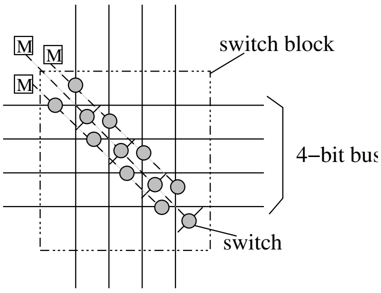


Figure 4.14. Example of the nibble-level control applied to the interconnect resources. Only three configuration bits rather than twelve are needed to control a 4-bit bus in the switch block (one configuration bit controls all switches of a given direction).

4.8 Benchmarking

In this section, the results of benchmarking the data-path-oriented reconfigurable architectures are discussed. To compare the proposed architectures with state-of-the-art FPGAs, two benchmarking methods were used. In the first method the implementation-based cost metrics (see Section 3.2.3) were applied, while in the second method the model-based cost metrics (see Section 3.2.4) were used.

4.8.1 Benchmarking using the implementation-based cost metrics

The data-path-oriented architecture was compared assuming the mapping cost MC as the primary cost metric. Therefore, the complete mapping procedure with complex functions (kernels) chosen as a benchmark set was applied to assess this architecture. The modified data-path-oriented architecture was compared using the data-path mapping cost MC_{DP} and random logic mapping cost MC_{RL}.

Framework

Given the logic tile areas of commercial FPGAs as in Table 3.1, and scaling them to the target technology (0.13 μm CMOS), the mapping cost MC (see Equation 3.18) of a given benchmark function was calculated based on the value of the parameter N_{LB} established after a mapping experiment. For the mapping onto commercial FPGAs, the Alliance package (for the Xilinx device) and Quartus II package (for the Altera device) were used.

To limit design effort, the logic tile of the data-path-oriented architecture was implemented in standard-cells of a 0.13 μm CMOS process. The netlist of the logic tile was prepared using a schematic entry of the Cadence design environment. The interconnect architecture of the tile was verified for routability using a custom (Philips internal) placement and routing tool called Pythagor [34]. The logic tile netlist was also verified functionally using Philips transistor-level simulator called Pstar [91]. The gate-level netlist in the Verilog format, which was written out from the schematic of the tile, was synthesised using Cadence BuildGates Synthesis [23], and placed and routed using Cadence Silicon Ensemble [21]. The synthesis tool was also used to derive timing characteristics of the tile. Due to lack of a dedicated technology mapping tool, the synthesis and mapping for the data-path-oriented architecture were performed using Synplify Pro mapping tool [97]. The tool was modified to account for some specific features of the proposed architecture[3]. The parameter N_{LB} was derived after this step.

[3] Because of a generic architecture of the logic block in the Atmel AT40K FPGA device (a functional equivalent of a 4-LUT), such a device was chosen to model the functionality of our data-path-oriented FPGA in Synplify Pro. The use of the Atmel-specific mapper enabled the preservation of some of the design components as macro-blocks (e.g. adders), and mapping them efficiently onto the logic blocks of our architecture. Such a mapping was performed during an optimisation step realised in the custom optimiser. The task of the optimiser was to optimise an Atmel-specific LUT-

Figure 4.15 shows the profile of 15 processing kernels from industrial designs that were selected as a benchmark set. The profile characterises the dominant type of processing of the kernels (see Section 2.3.1).

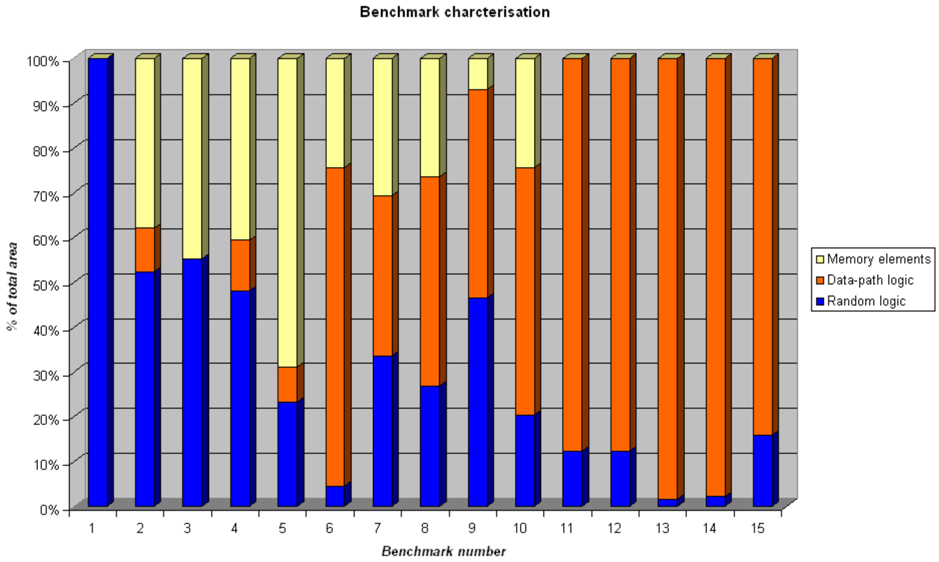


Figure 4.15. The ‘type of processing’ profile of the benchmark set. (The numbers in the figure correspond to the benchmark numbers in Table 4.2 and Table 4.3.)

Assuming the benchmark set as described above, the results of benchmarking our data-path-oriented FPGA with two commercial FPGAs, that is Xilinx Virtex-E and Altera APEX 20KE, are shown in Table 4.2. For each benchmark function, the table lists the number of required logic blocks N_{LB} and the mapping cost MC. The mapping cost ratios ($MC\ ratio$) that are also listed in Table 4.2 were calculated assuming the mapping cost of our FPGA as a reference. The mapping costs of commercial FPGAs that were normalised with regard to the mapping cost of our FPGA are shown in a graphical form in Figure 4.16.

Table 4.3 compares the FPGA mapping costs with the area A of a standard-cell-based ASIC implementation of each benchmark. The area figures for the ASIC implementation were obtained after the synthesis step only (in Cadence BuildGates), and therefore does not include the interconnect (routing) cost. The area ratios ($A\ ratio$) between the FPGA and ASIC implementations were calculated assuming the ASIC area as the reference. Such results are also shown in a graphical form in Figure 4.17.

based netlist exploiting specific properties of our architecture. The implemented optimisations included: efficient packing of data-path multiplexers to the logic blocks, efficient implementation of multiplexers with one or two inverted inputs, mapping of 2-input Boolean functions onto 2-LUTs of the logic block, packing multi-bit Boolean functions, and packing flip-flops to the logic blocks.

		DP-oriented		**Xilinx Virtex-E**			**Altera APEX 20KE**		
No.	Benchmark	N_{LB} [–]	MC [μm^2]	N_{LB} [–]	MC [μm^2]	$MC\ ratio$ [–]	N_{LB} [–]	MC [μm^2]	$MC\ ratio$ [–]
1	des	451	3414521	166.5	3079751	0.90	76.5	2520293	0.74
2	timer	189	1430919	62	1146814	0.80	23.1	761029.5	0.53
3	uart	274	2074454	86.5	1599991	0.77	33.1	1090480	0.53
4	usb	17305	131016155	5302.5	98080343	0.75	2294.8	75602186	0.58
5	video_ctrl	279	2112309	97	1794209	0.85	32.5	1070713	0.51
6	memptr	367	2778557	258.5	4781475	1.72	135.8	4473931	1.61
7	mix_select	170	1287070	63	1165311	0.91	30.9	1018001	0.79
8	erosion	426	3225246	186	3440442	1.07	57.9	1907516	0.59
9	ppone	133	1006943	58	1072826	1.07	25.5	840097.5	0.83
10	lambda	299	2263729	124	2293628	1.01	55.8	1838331	0.81
11	alpha	105	794955	60.5	1119069	1.41	28.1	925754.5	1.16
12	betha	105	794955	60.5	1119069	1.41	28.1	925754.5	1.16
13	asu8	268	2029028	270	4994190	2.46	208.8	6878916	3.39
14	dft4	1188	8994348	694	12836918	1.43	455.4	15003153	1.67
15	cordic	289	2188019	169.5	3135242	1.43	85.7	2823387	1.29
Average						**1.20**			**1.08**

Table 4.2. Mapping cost comparison for the proposed data-path-oriented FPGA and two commercial FPGAs. The data-path-oriented FPGA is chosen as a reference. Note, that a customised mapping flow and a full-custom implementation are assumed for the commercial FPGAs.

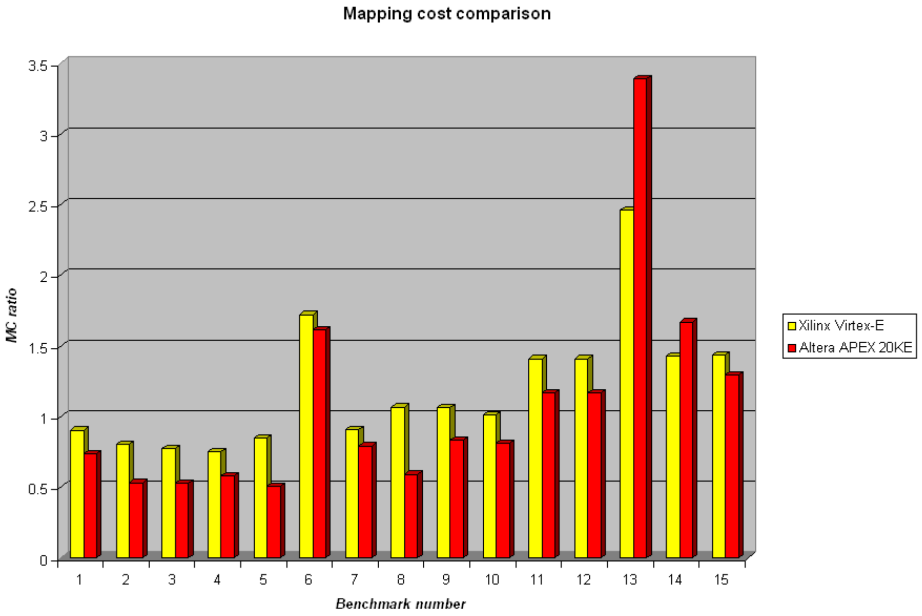


Figure 4.16. Mapping cost of the commercial FPGAs using the mapping cost of the data-path-oriented FPGA as a reference.

Because a dedicated mapping tool was not available, the mapping cost of the modified data-path-oriented architecture was evaluated using the data-path mapping cost MC_{DP} (see Equation 3.19) and the random logic mapping cost MC_{RL} (see Equation 3.20). The area of the logic tile in the modified data-path-oriented FPGA, necessary for the calculation of both cost metrics, was obtained by a standard-cell-based implementation of the tile in the same way as for the other data-path-oriented architecture. Also, the worst-case delay of the tile T was derived. To enable a relative comparison of both data-path architectures, the logic tile of the original data-path-oriented architecture was also compared using the MC_{DP} and MC_{RL} cost metrics.

Three data-path-optimised FPGAs from academia described in Table 3.1 were chosen for the assessment of the data-path-oriented architectures. The results of such an assessment are shown in Table 4.4. For each FPGA, the organisation of a logic block, the logic tile area A_{LT} (after scaling to a 0.13 μm CMOS process), and the worst-case delay T are mentioned. The mapping costs MC_{DP} and MC_{RL} (in μm^2) that were calculated based on the available data are also listed. The table makes a distinction between FPGA architectures with the random logic mapping capabilities (the first group) and FPGA architectures with only multi-bit Boolean function mapping capabilities (the second group). Consequently, for the second group of architectures only the data-path mapping cost MC_{DP} is given.

		ASIC			
No.	Benchmark	A [μm^2]	A *ratio* (DP-oriented) [–]	A *ratio* (Xilinx) [–]	A *ratio* (Altera) [–]
1	des	19798.8	172.5	155.6	127.3
2	timer	5168.1	276.9	221.9	147.3
3	uart	8655.8	239.7	184.8	126.0
4	usb	792878.6	165.2	123.7	95.4
5	video_ctrl	12286.8	171.9	146.0	87.1
6	memptr	35361.5	78.6	135.2	126.5
7	mix_select	8865.6	145.2	131.4	114.8
8	erosion	17934.9	179.8	191.8	106.4
9	ppone	5605.8	179.6	191.4	149.9
10	lambda	17692.9	127.9	129.6	103.9
11	alpha	8117.2	97.9	137.9	114.0
12	betha	8103.1	98.1	138.1	114.2
13	asu8	45752.1	44.3	109.2	150.4
14	dft4	99316.8	90.6	129.3	151.1
15	cordic	35224.3	62.1	89.0	80.2

Table 4.3. ASIC cost (area A) versus FPGA mapping cost. Area ratios (A *ratio*) for the data-path-oriented FPGAs and two commercial FPGAs are calculated assuming the ASIC implementation as a reference.

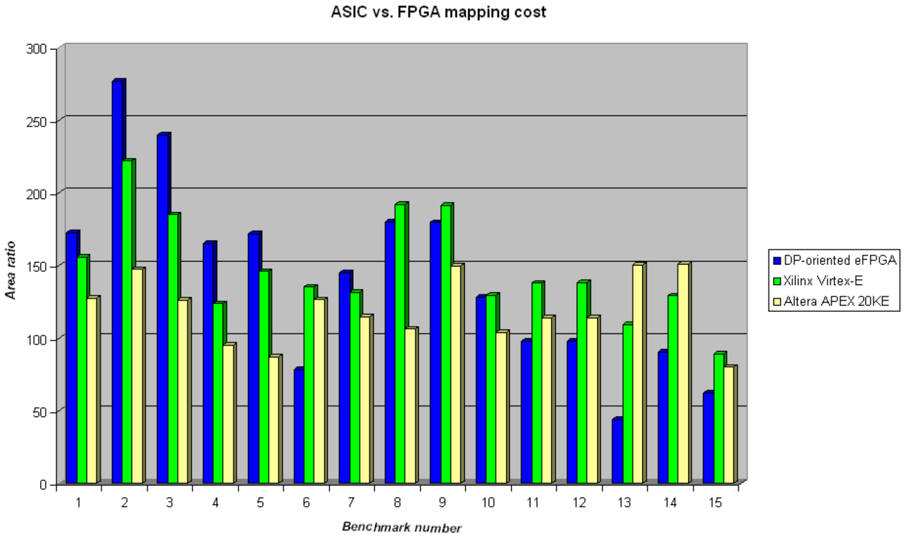


Figure 4.17. Mapping cost of the commercial and proposed FPGAs using the ASIC implementation area as a reference.

FPGA architecture	Logic block organisation	A_{LT} $[\mu m^2]$	T $[ns]$	MC_{DP} $[\mu m^2/bit]$	MC_{RL} $[\mu m^2/4\text{-}LUT]$
LP-PGA II	5/3-LUT	14271	4.2	7136	9514
DP-oriented	4×2-LUT	7571	2.0	1893	7571
CFPA	4-bit logic	4275	1.0	1069	n/a
CHESS	4-bit ALU	6208	1.9	1552	n/a
DP-oriented (modified)	4×2-LUT (shared memory)	5824	1.8	1456	n/a

Table 4.4. Area and performance comparison of the data-path-oriented and academia FPGAs. The area figures for the proposed FPGAs refer to their standard-cell rather than full-custom implementations. All logic tile areas are scaled to a 0.13 μm CMOS technology.

4.8.2 Benchmarking using the model-based cost metrics

The data-path-oriented and modified data-path-oriented reconfigurable architectures were also compared using the area model of an FPGA logic tile that has been discussed in Section 3.2.4. The parameters N_{lmb} and P_w, which according to the model characterise the logic and routing resources of the data-path-oriented FPGAs, are listed in Table 4.5.

FPGA architecture	Number of pins					N_{lmb}	P_w
	Inputs	Outputs	Carry input	Carry output	Auxiliary		
Data-path-oriented	8	4	0	1	3	16	11
Data-path-oriented (modified)	8	4	0	1	3	4	8

Table 4.5. Characterisation of the logic cost (via N_{lmb}) and the routing resource cost (via P_w) in the proposed data-path-oriented FPGA architectures.

Given the number of LUT memory bits N_{lmb} and the weighted pin number P_w as listed in Tables 3.3 and 4.5, the mapping cost with respect to logic MC_L (see Equation 3.23) and the mapping cost with respect to routing MC_R (see Equation 3.24) were calculated for the set of modern FPGAs (see Section 3.2.4) and the proposed data-path-oriented FPGAs. Relatively simple functions (basic functional primitives of the data-path, random logic, and memory type) were used as a benchmark set, and the values of the parameter N_{LB} were obtained according to the method explained in Section 3.2.4.

Table 4.6 shows the results of the comparison using the above-mentioned metrics. For each commercial and data-path-oriented FPGA architectures, the table lists the values of N_{LB}, and two mapping cost components, that is MC_L and MC_R.

For a convenient analysis, the data from Table 4.6 are also shown graphically in Figures 4.18–4.27. Additionally, Table 4.7 summarises the results from Table 4.6

	State-of-the-art commercial FPGA architectures								
	Xilinx Virtex II			Altera Stratix			Atmel AT40K		
Benchmark function	N_{LB}	Mapping cost		N_{LB}	Mapping cost		N_{LB}	Mapping cost	
	–	MC_L	MC_R	–	MC_L	MC_R	–	MC_L	MC_R
8-bit ADD	1	128	62	0.8	128	50	8	128	48
16×16 MULT	34	4.25K	2.11k	73	11.41K	4.53k	256	4K	1.54k
2:1 MUX/4-bit	0.5	64	31	0.4	64	25	4	64	24
8:1 MUX/1-bit	0.5	64	31	0.5	80	31	7	112	42
16:1 MUX/1-bit	1	128	62	1	160	62	15	240	90
2-in OR/4-bit	0.5	64	31	0.4	64	25	4	64	24
3-in NOR/1-bit	0.125	16	8	0.1	16	6	1	16	6
16-in AND/1-bit	0.5	64	31	0.4	64	25	5	80	30
4:16 DECOD	2	256	124	1.6	256	99	16	256	96
16-long 2-bit SREG	0.25	32	16	3.2	512	198	32	512	192

	Proposed data-path-oriented FPGA architectures					
	Data-path-oriented			Data-path-oriented (modified)		
Benchmark function	N_{LB}	Mapping cost		N_{LB}	Mapping cost	
	–	MC_L	MC_R	–	MC_L	MC_R
8-bit ADD	2	32	22	2	8	16
16×16 MULT	64	1K	704	64	256	512
2:1 MUX/4-bit	1	16	11	1	4	8
8:1 MUX/1-bit	1	16	11	1	4	8
16:1 MUX/1-bit	3	48	33	3	12	24
2-in OR/4-bit	1	16	11	1	4	8
3-in NOR/1-bit	1	16	11	2	8	16
16-in AND/1-bit	4	64	44	5	20	40
4:16 DECOD	6	96	66	8	32	64
16-long 2-bit SREG	8	128	88	8	32	64

Table 4.6. The mapping cost comparison between the state-of-the-art and proposed data-path-oriented FPGA architectures.

by showing the relative comparison of the costs MC_L and MC_R of the compared FPGA architectures, and assuming the data-path-oriented FPGA architecture as a reference.

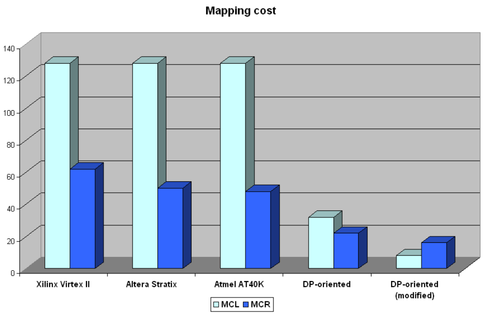


Figure 4.18. 8-bit adder.

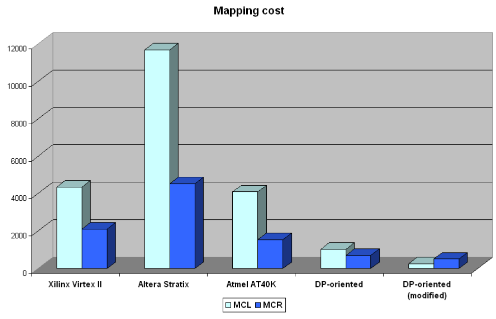


Figure 4.19. 16-bit multiplier.

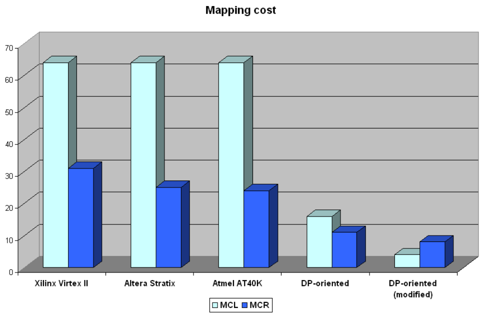


Figure 4.20. 2:1 4-bit multiplexer.

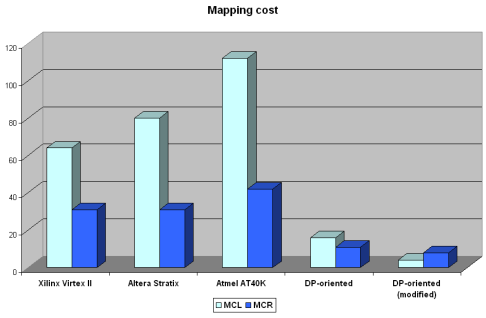


Figure 4.21. 8:1 1-bit multiplexer.

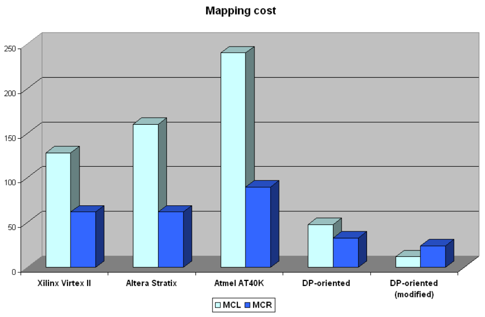


Figure 4.22. 16:1 1-bit multiplexer.

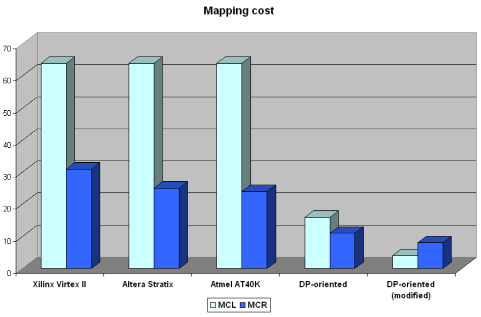


Figure 4.23. 2-input 4-bit OR.

4.8.3 Discussion

The data from Table 4.2, obtained using the area-based cost metric *MC*, indicate that the proposed data-path-oriented FPGA compares favourably with the commercial FPGA devices. This is because the mapping cost of our FPGA, averaged

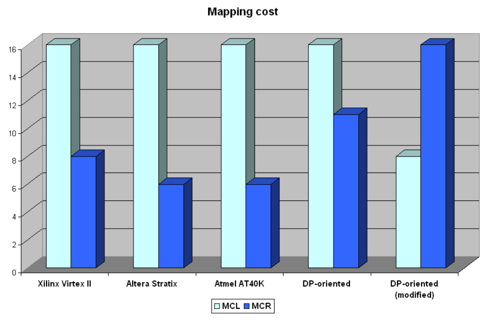


Figure 4.24. 3-input 1-bit NOR.

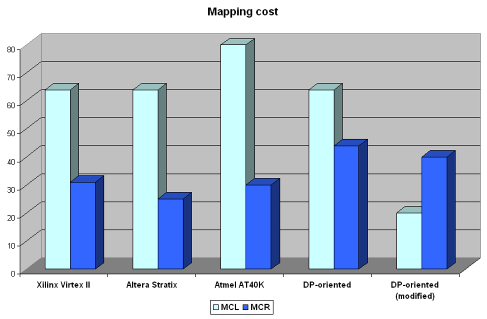


Figure 4.25. 16-input 1-bit AND.

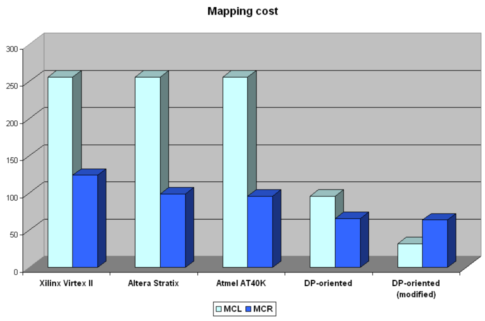


Figure 4.26. 4:16 decoder.

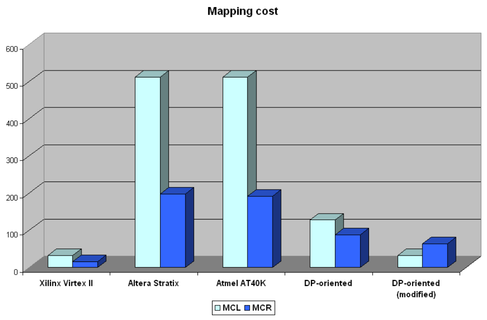


Figure 4.27. 16-long 2-bit shift register.

Commercial →	**Xilinx Virtex II**		**Altera Stratix**		**Atmel AT40K**	
Proposed ↓	MC_L ratio	MC_R ratio	MC_L ratio	MC_R ratio	MC_L ratio	MC_R ratio
DP-oriented	av=2.78	av=1.96	av=4.04	av=2.28	av=3.69	av=2.01
	mn=0.25	mn=0.18	mn=1.0	mn=0.55	mn=1.0	mn=0.55
	mx=4.25	mx=3.0	mx=11.4	mx=6.43	mx=7.0	mx=3.82
DP-oriented (modified)	av=10.59	av=2.57	av=15.62	av=3.03	av=14.2	av=2.66
	mn=1.0	mn=0.25	mn=2.0	mn=0.38	mn=2.0	mn=0.38
	mx=17	mx=4.12	mx=45.63	mx=8.84	mx=28.0	mx=5.25

Table 4.7. The summary of the results from Table 4.6. The mapping costs w.r.t. logic MC_L and w.r.t. routing MC_R of three commercial FPGAs are normalised with respect to the mapping costs of the data-path-oriented architectures (comparison per rows). The average cost ratio *av* for 10 benchmarks, minimum *mn* and maximum *mx* cost ratios are given.

over 15 different benchmark kernels, is 17% and 8% smaller than the mapping costs of the Xilinx Virtex-E and Altera APEX 20KE devices, respectively[4]. The mapping cost clearly depends on the type of the mapped function. The highest benefit of our architecture is, as expected, for the data-path-dominated benchmarks, such as benchmarks no. 10–15 in our benchmark set (see Figure 4.15). The average mapping cost of such benchmarks is for the proposed architecture 34% and 37% smaller than the mapping costs for the Xilinx and Altera devices.

All these results were obtained assuming *full-custom* implementations of the commercial FPGAs and a *standard-cell-based* implementation of the proposed FPGA. Also, an unoptimised mapping flow (reuse of the Atmel mapper) was applied to map onto our FPGA. The latter had a strong impact on the obtained results. A simple experiment has shown that by only slightly rewriting the benchmark code (benchmark no. 10), the mapping cost of our FPGA could be decreased by 17%[5]. It is interesting to note that even with such assumptions, the mapping cost of the standard-cell-based data-path-oriented FPGA is close to the mapping cost of the full-custom Altera APEX 20KE.

The comparison of the FPGA and ASIC mapping costs (see Table 4.3) indicates that for a large number of benchmarks there is more than a factor of 100 difference in area between both implementation styles. Such a high factor can be partially explained by the fact that the cost of interconnect has not been taken into account in the ASIC implementations. The average difference in area between the ASIC implementation and the implementation of our FPGA, calculated based on the results from Table 4.3, is about 142 times.

In Table 4.4, the results of the comparison of both proposed data-path-oriented FPGAs with the FPGAs from academia are presented. The results show that standard-cell-based implementation of our data-path-oriented architecture has 3.77 times lower data-path mapping cost MC_{DP} than the LP-PGA II FPGA device. The random logic mapping cost MC_{RL} of our device is 1.26 times lower. In the same comparison, the standard-cell-based implementation of the modified data-path-oriented architecture is characterised by 1.36 times higher data-path mapping cost MC_{DP} than the best data-path-optimised and full-custom implemented FPGA from academia, that is CFPA. Note, that applying the nibble-level control to the logic and routing resources of the modified data-path-oriented architecture yields a 23% cost reduction if data-paths are mapped, compared to the similar cost of the bit-level controlled data-path-oriented architecture.

The benchmarking of the data-path-oriented architectures using the model-based cost metrics has enabled their technology-independent comparison. Consequently, only the architectural aspects could be taken into account while comparing these

[4] Differently: the mapping costs of the Xilinx and Altera devices are 20% and 8% higher.

[5] This is because the Atmel mapper does not recognise incrementers, decrementers and comparison functions. When such functions are found, they are dissolved and mapped as random logic rather than being preserved as macros.

architectures with modern commercial FPGAs. The summary of the results shown in Table 4.7 indicates that the data-path-oriented architecture reduces both the logic as well as the routing costs. On average, the logic cost for the Xilinx, Altera and Atmel FPGAs is 2.78, 4.04 and 3.69 times higher, respectively, than for the data-path-oriented FPGA. The routing cost for the same devices is on average 1.96, 2.28 and 2.01 times higher than for the data-path-oriented FPGA. As expected, the modified data-path-oriented architecture reduces further the logic cost (because of the applied configuration bit sharing). The logic cost of the Xilinx, Altera and Atmel devices compared to this architecture is 10.59, 15.62 and 14.2 times higher, respectively. The commercial FPGAs also have one average 2.57, 3.03 and 2.66 times higher routing costs than the modified data-path-oriented FPGA.

4.9 Conclusions

The data-path-oriented reconfigurable logic architecture presented in this chapter targets applications with the data-path-dominated processing kernels. Because typical data-paths contain a considerable amount of arithmetic, the data-path-oriented architecture was optimised towards an efficient mapping of basic arithmetic operations. The optimisation was achieved by applying the inversion-based folding type I. We showed that a factor of four reduction in the total number of LUT memory bits compared to the implementations of the state-of-the-art FPGAs can be achieved in this way.

The logic block of the proposed FPGA was designed to support a cost-efficient mapping of multi-bit (data-path) functions and reasonably efficient mapping of functions with a single-bit output (random logic). We also designed an interconnect structure for such a logic block which allows an efficient utilisation of the routing resources both when multi-bit and single-bit output functions are mapped. In this way, an unnecessary increase of the amount of routing resources can be avoided.

We also discussed a modification of the data-path-oriented architecture, namely the modified data-path-oriented architecture. Such an architecture exploits the concept of the configuration bit sharing by applying the nibble-level-control to its logic and routing resources. The modified data-path-oriented architecture offers a further reduction of the implementation cost.

Two comparison methods applied to assess the quality of the proposed architectures showed that they are superior to the commercial FPGAs for the data-path applications. This is because the architectures offer a similar functionality at the lower implementation cost. For example, for a mixture of different benchmarks our standard-cell-based data-path-oriented FPGA is on average 17% and 8% less expensive than the Xilinx and Altera FPGAs. If only the data-path benchmarks are considered, the architecture shows 34% and 37% cost reduction, respectively.

Chapter 5

RANDOM-LOGIC-ORIENTED RECONFIGURABLE ARCHITECTURE

In this chapter, the second type of the domain-oriented reconfigurable logic architectures, namely the *random-logic-oriented architecture* is discussed. The architecture targets applications with the random-logic-dominated processing kernels.

5.1 Introduction

5.1.1 Characteristics of the application domain

The networks of combinatorial and sequential logic elements that are connected together in an arbitrary way are often regarded as *random logic*. The term ‘random’ is used to reflect lack of structure, that is, no apparent regularity in logic and interconnect. Typical examples of random-logic are bit-level manipulations, Boolean operations, and control logic. Bit-level manipulations are simple logic and shift operations usually performed on pairs of bits. Boolean operations are complex logic (Boolean) functions performed on multiple 1-bit inputs and producing relatively few outputs (often a single output only). Finally, control logic are conditional logic functions with a state and feedback loops (typically implemented as finite state machines). Though in the implementation of such functions combinational and sequential logic elements are dominating, some arithmetic computations can also be encountered.

5.1.2 State-of-the-art

The general-purpose character of today’s FPGAs causes that the random logic mapping capabilities of FPGAs are compromised with their arithmetic mapping capabilities. Therefore, though the implementation of random logic would benefit from larger look-up tables (see the detailed discussion in Section 2.1.2), in practice a 4-LUT is chosen as a basic logic element of an FPGA logic block. Two main FPGA vendors, that is Xilinx and Altera, almost exclusively use 4-LUTs in their devices.

5.2 Logic element

Following the observations from Section 4.1.1 and Section 5.1.1, we propose a logic element which is optimised for the mapping of random logic still allowing the mapping of simple arithmetic structures. The logic element constitutes the lowest level of hierarchy of the random-logic-oriented reconfigurable logic architecture.

The (technology) mapping onto 4-LUTs is found to produce the most cost-efficient designs [3]. The cost-efficiency in this case is regarded as the area-delay product that describes the mapped design netlist. We choose thus a 4-LUT as a fundamental component of the proposed logic element. However, to enable the mapping of arithmetic, we decompose a monolithic 4-LUT into two 3-LUTs, each having an independent output. Note, that a 3-LUT suffices to generate a single output of an arithmetic operation (i.e. a sum or a carry output) without using any additional logic. Consequently, 3-LUTs can be used to generate two outputs of a 1-bit arithmetic operation, or be combined together in a 4-LUT to generate a 4-input Boolean function according to Shannon's expansion (see Section 3.1.1).

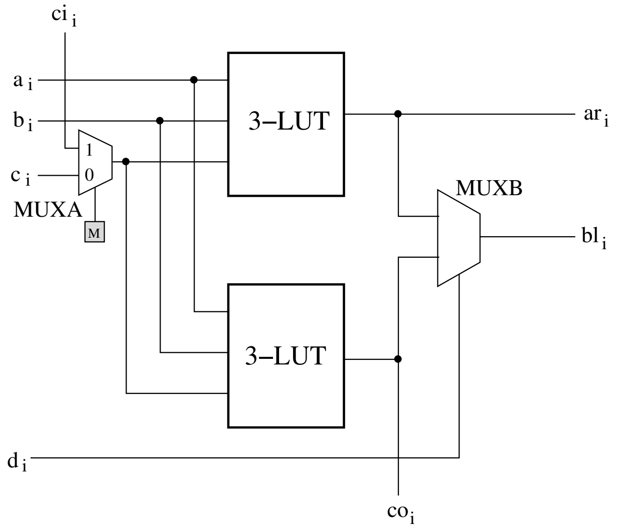


Figure 5.1. Logic element of the random-logic-oriented reconfigurable logic architecture.

The implementation details of such a basic logic element are shown in Figure 5.1. The logic element consists of two 3-input look-up tables (3-LUTs) and two 2:1 multiplexers. The inputs of either LUT are paired together. Two of the three LUT inputs are connected directly to the primary inputs a_i and b_i of the logic element, while the third input is connected to the output of the multiplexer $MUXA$. This multiplexer selects between the third primary input c_i of the logic element and the carry input ci_i. The multiplexer $MUXA$ is controlled by a configuration bit M. The fourth primary input d_i of the logic element is connected to the control input of the multiplexer $MUXB$, which performs a selection of the LUT outputs. The logic element has two primary outputs: an arithmetic output ar_i and a Boolean output

bl_i. The arithmetic output is the output of the first 3-LUT, while the Boolean output is the output of the multiplexer $MUXB$. The secondary output of the logic element is the carry output co_i, which is connected to the output of the second 3-LUT.

The suggested logic element has a structure very similar to that of commercial FPGA devices, such as Altera APEX [4] and Atmel AT40K [8], for example. The advantage of our logic element is its very simple structure, though (no extra gates enhancing the functionality).

5.3 Logic block

5.3.1 Basic concept

The following observations have been taken into account while designing the logic block, which is the next level of hierarchy of the random-logic-oriented reconfigurable logic architecture. Firstly, implementations of complex Boolean functions benefit from large look-up tables [94, 47, 3]. Secondly, random logic functions often share some of their inputs when producing a multi-output result [14, 61]. Thirdly, feedback loops are common in random logic designs [14]. And finally, typical arithmetic functions are coarse, that is, they have multi-bit arguments [26]. To meet these requirements, we build the logic block as a cluster of four logic elements. The resulting granularity of the logic block matches well the requirements of random logic [14, 3] and arithmetic [26] functions.

5.3.2 Structure in detail

The complete structure of the logic block is shown in Figure 5.2. The logic block consists of four *logic elements* and the set of global multiplexers MUX1, MUX2 and MUX3. The eleven primary inputs $in_1 \dots in_{11}$ of the logic block are directed to the *input selection block*, which detailed implementation is shown in Figure 5.3(a). The input selection block includes two sets of 16:1 multiplexers: the first set has 16 multiplexers, while the second set only three multiplexers. The function of each multiplexer is to select an input signal from eleven primary inputs $in_1 \dots in_{11}$ of the logic block or four feedback signals connected to the outputs $out_1 \dots out_4$ of the logic block.

The outputs of the multiplexers in the first set of the input selection block are connected directly to the inputs a_i, b_i, c_i, d_i of successive logic elements, where $i = 1 \dots 4$ is the index of a logic element. Each logic element generates a pair of signals, which are available on its arithmetic ar_i and Boolean bl_i outputs. The arithmetic outputs $ar_1 \dots ar_4$ are directed to the output selection block, while the Boolean outputs rl_1, rl_2 and rl_3, rl_4 are merged in the global multiplexers MUX1 and MUX2, respectively. The global multiplexer MUX3 combines the outputs of the multiplexers MUX1 and MUX2. The control signals e, f and g of the

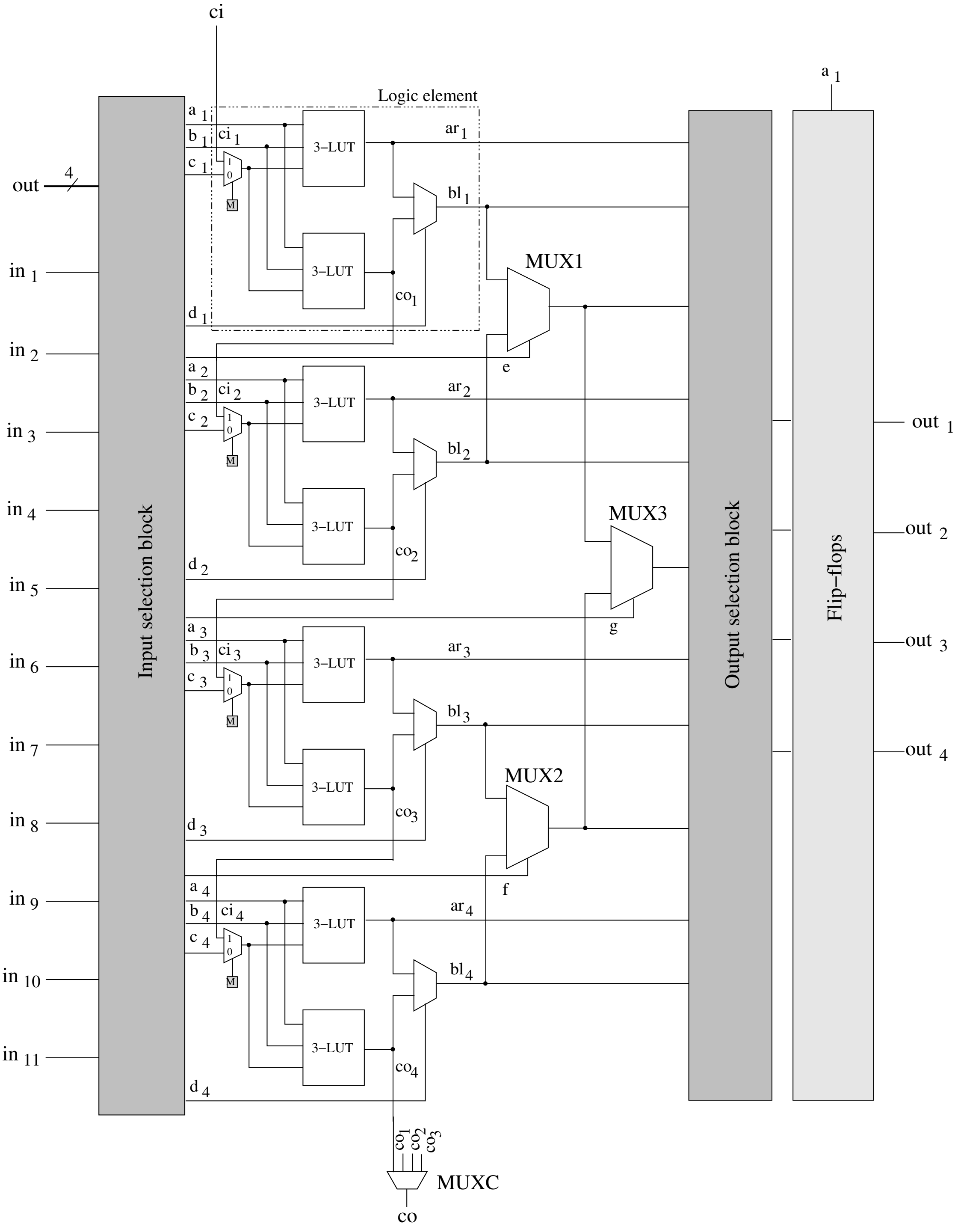


Figure 5.2. Logic block of the random-logic-oriented reconfigurable logic architecture.

global multiplexers come from the second set of multiplexers of the input selection block. The outputs of the global multiplexers MUX1, MUX2, and MUX3 are also directed to the output selection block.

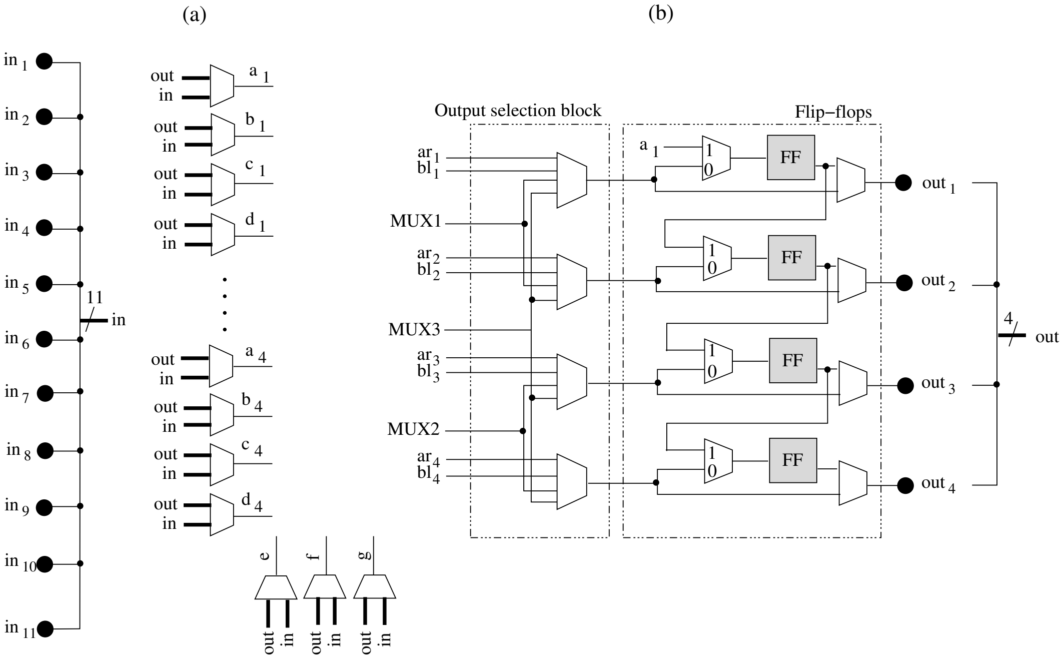


Figure 5.3. (a) Input selection block and (b) output selection block with flip-flops block. Each of the multiplexers in the figure is controlled by an independent set of configuration memory bits.

The *output selection block*, which is shown in Figure 5.3(b), selects four final signals of the logic block. For that purpose, the output selection block includes four 4:1 multiplexers that select between the arithmetic and Boolean outputs of the logic elements as well as the outputs of the global multiplexers MUX1, MUX2 and MUX3[1]. To guarantee a high level of flexibility, each of the multiplexers in the output selection block is controlled by an independent set of configuration bits.

The outputs of the output selection block are directed to the *flip-flop* block (see Figure 5.3(b)). This block serves two purposes: it permits to register each output of the logic block, and it allows the implementation of shift registers. The latter is achieved if the first stage of multiplexers of the flip-flop block is configured to select the signals from their inputs numbered 1. This results in chaining the flip-flops. The input of the shift register is a signal available on the first primary input a_1 of the logic block, and the output is any of the outputs $out_1 \ldots out_4$ of the logic block. The multiplexers of the first and second stage of the flip-flop block are controlled independently.

[1] Note, that by connecting the outputs of the multiplexers MUX1 and MUX2 to the outputs of the logic block, the implementation of independent 5-input logic functions is enabled.

The logic block has a secondary input *ci* and a secondary output *co*. These ports of the logic block are used while implementing arithmetic functions, and play the role of the carry input and carry output, respectively. The signal from the port *ci* is connected to one of the inputs of the multiplexer MUXA in the first logic element. The multiplexer MUXA of the next logic element receives one of its input signals from the second 3-LUT in the preceeding logic element. Finally, the second 3-LUT of the last logic element has its output connected to the *co* output of the logic block. In this way, a carry chain is formed.

5.4 Functional modes

The logic block of the random-logic-oriented reconfigurable logic architecture is designed to support two functional modes, that is:

- *Random logic mode* in which at least one and at most four Boolean functions are generated. The outputs of the functions are available on the Boolean outputs bl_i of the logic elements or on the outputs of the global multiplexers MUX1, MUX2 and MUX3.

- *Arithmetic mode* in which arithmetic functions of up to four bits are implemented. The outputs of the arithmetic functions are available on the arithmetic outputs ar_i of the logic elements. The carry chain, with the input and output on the secondary ports *ci* and *co* of the logic block, respectively, is implemented by the chained look-up tables.

Unlike the logic block of the data-path-oriented architecture, the logic block of the random-logic-oriented architecture can be configured in such a way that a part of its logic resources implements a random logic function, while the other part implements an arithmetic function. This improves the resource utilisation.

5.4.1 Random logic mode

In the random logic mode, Boolean functions are implemented in the logic elements that are configured as 4-LUTs. The logic block offers the following implementation options: four Boolean functions of four inputs, two Boolean functions of five inputs, or any Boolean function of six inputs. The inputs of the functions are selected from the set of eleven primary inputs $in_1 \ldots in_{11}$ of the logic block, and outputs are available on the logic block outputs $out_1 \ldots out_4$. In the implementations of five-input and six-input logic functions, the global multiplexers MUX1, MUX2 and MUX3 are involved. The possibility of connecting the outputs of the logic block to its inputs (see Figure 5.3(a)), enables the implementation of functions with a feedback loop (e.g. finite state machines).

In the random logic mode, the implementation of *random logic multiplexers* is also possible. In such a case, the multiplexers MUXB of the logic elements and the global multiplexers MUX1, MUX2, and MUX3 are used. Maximally, a 8:1 multiplexer can be implemented in the logic block.

Also, *shift registers* with the maximal delay of four clock cycles can be implemented using the flip-flop block resources.

5.4.2 Arithmetic mode

In the arithmetic mode, arithmetic functions with up to 4-bit arguments are mapped. This is done by configuring each logic element of the logic block as two 3-LUTs with the shared inputs. The first 3-LUTs of all logic elements generate the sum signals on their arithmetic outputs ar_i, and all second 3-LUTs generate the carry signals on their co_i outputs. Two inputs of each pair of LUTs are connected to the primary inputs of the logic block, while the third input is the carry signal ci_i selected by the multiplexer MUXA. The carry signal ci_i of each logic element is coupled to the carry output co_{i-1} of the previous logic element (see Figure 5.2). Only the first logic element receives its carry signal ci_1 from the secondary input *ci* of the logic block, and the carry output signal co_4 of the last logic element connects to the secondary output *co* of the logic block. The output multiplexer MUXC allows the selection of one of the intermediate carry signals of the logic elements (i.e. co_1, co_2, co_3). In this way, a configuration of the logic block in a combined arithmetic and random logic mode is possible.

5.5 Interconnect

The interconnect structure of the proposed random-logic-oriented reconfigurable logic architecture has been established via mapping experiments using the MCNC circuits [120] as a primary benchmark set. In addition, the FPGA routing design guidelines from [14] have been followed.

We set the number of input ports of the logic block as 11 rather than 16, as it would be implied from the total number of the 4-LUT inputs (i.e. 4 x 4 inputs). According to [14, 3], 10 input ports for a logic block implemented as a cluster of four 4-LUTs suffice to guarantee nearly the 100% logic utilisation if random logic functions are mapped. This is because such functions often share some of their inputs [61]. Despite that, we added one extra port (in_{11}) to the logic block to enable mapping of 8:1 multiplexers (a 8:1 multiplexer needs eight data inputs and three control inputs). This also relaxes the routing resource requirements slightly. Note, that eleven input ports is also sufficient for the mapping of arithmetic functions, since the most coarse (4-bit) function requires only eight input ports (a carry port is not counted). The reduction of the logic block port number is essential for the reduction of the total routing cost of the architecture [52].

Furthermore, we chose an uniform routing architecture with the channel width W=42 tracks. The complete routing architecture includes three types of routing resources, namely:

- general-purpose,
- special, and
- direct.

The *general-purpose routing resources* are used to route arbitrary types of signals of the mapped design. These resources consist of 40 routing tracks, 60% of which is of the length-four (L4) and 40% of the length-eight (L8). The routing tracks of the length-four are implemented with pass transistor switches, while the tracks of the length-eight with bi-directional buffers. The input and output connection blocks of the general-purpose routing have the 50% connection flexibility, that is F_{ci}=0.5 and F_{co}=0.5, respectively. The disjoint type of the switch block [14] with the 50% connection population for the routing tracks of the length-eight is used. The *special routing resources* include two pairs of horizontal and vertical routing tracks. The tracks are meant for the routing of carry signals if arithmetic functions are mapped. One pair of the routing tracks is of the length-one (L1), while the other pair is of the length-four (L4). The *direct routing resources* implement fast connections between neighbouring logic blocks. The direct connections are implemented in the so-called full topology [85], that is, they connect logic blocks on the left, right, bottom, top, and diagonally. There are four direct connections

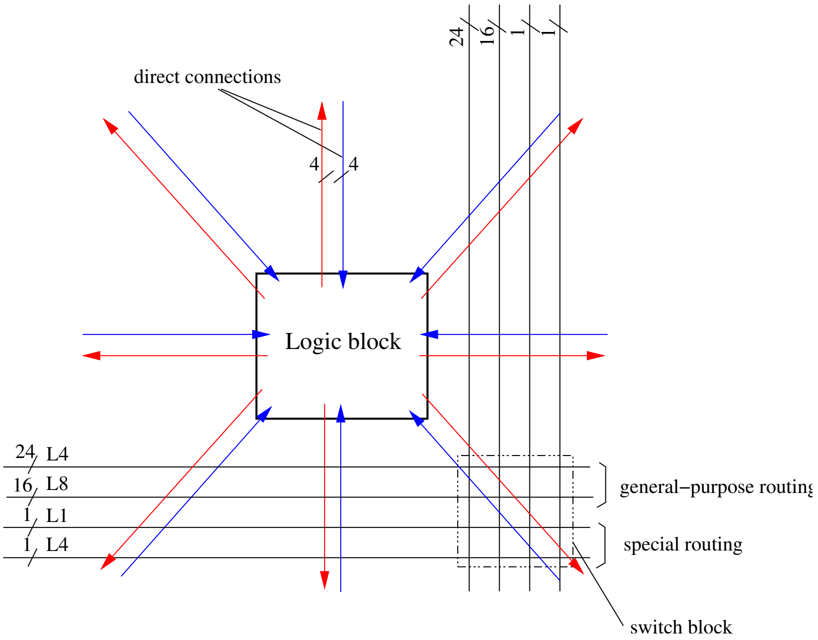


Figure 5.4. Interconnect architecture of the random-logic-oriented reconfigurable logic architecture.

available per each direction. The routing architecture of the random-logic-oriented architecture is shown in Figure 5.4.

5.6 Benchmarking

To evaluate the random-logic-oriented reconfigurable architecture, a benchmarking method relying on the model-based cost metrics (see Section 3.2.4) was used.

5.6.1 Benchmarking using the model-based cost metrics

The benchmarking using the model-based cost metrics was realised similarly to the benchmarking of the data-path-oriented architecture described in Section 4.8.2. The parameters N_{lmb} and P_w of the area model as shown in Table 5.1 were assumed for the characterisation of the random-logic-oriented architecture.

FPGA architecture	Number of pins					N_{lmb}	P_w
	Inputs	Outputs	Carry input	Carry output	Auxiliary		
Random-logic-oriented	11	4	1	1	0	64	16

Table 5.1. Characterisation of the logic cost (via N_{lmb}) and the routing resource cost (via P_w) in the proposed random-logic-oriented FPGA architecture.

The parameters N_{lmb} and P_w from Table 3.3 and Table 5.1 were used to calculate the mapping cost with respect to logic MC_L (see Equation 3.23) and the mapping cost with respect to routing MC_R (see Equation 3.24) for the set of modern FPGAs and the proposed random-logic-oriented FPGAs. The results are given in Table 5.3. For each of the architectures, the table lists the value of the parameter N_{LB} and two mapping cost components, that is MC_L and MC_R.

For a convenient analysis, the data from Table 5.3 are also shown graphically in Figures 5.5–5.14. Additionally, Table 5.2 summarises the results from Table 5.3 by showing the ratio of mapping costs MC_L and MC_R assuming the random-logic-oriented FPGA as a reference.

5.6.2 Discussion

The results of the comparison of our random-logic-oriented FPGA with three commercial FPGAs indicate that the proposed architecture has a lower implementation cost. The reduction comes mainly from the reduction of the routing resource cost. As shown in Table 5.2, the Xilinx, Altera and Atmel FPGAs have their average routing cost 1.62, 1.65 and 1.64 times higher, respectively, than the similar cost for our FPGA. The cost reduction is possible because of a simpler interface of the logic block in the proposed architecture (less input and output pins).

Commercial →	Xilinx Virtex II		Altera Stratix		Atmel AT40K	
Proposed ↓	MC_L	MC_R	MC_L	MC_R	MC_L	MC_R
RL-oriented	av=0.83	av=1.62	av=1.07	av=1.65	av=1.1	av=1.64
	mn=0.06	mn=0.125	mn=0.8	mn=1.25	mn=0.53	mn=0.8
	mx = 1	mx=2	mx=1.52	mx=2.36	mx=1.75	mx=2.62

Table 5.2. The summary of the results from Table 5.3. The mapping costs w.r.t. logic MC_L and w.r.t. routing MC_R of three commercial FPGAs are normalised with respect to the mapping costs of the random-logic-oriented architecture. The average cost ratio *av* for 10 benchmarks as well as the minimum *mn* and maximum *mx* cost ratios are given.

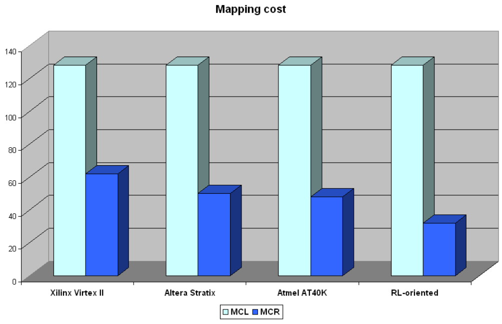


Figure 5.5. 8-bit adder.

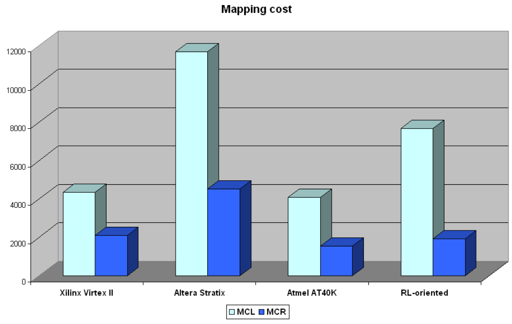


Figure 5.6. 16-bit multiplier.

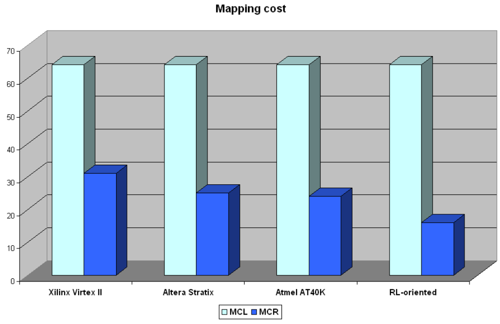


Figure 5.7. 2:1 4-bit multiplexer.

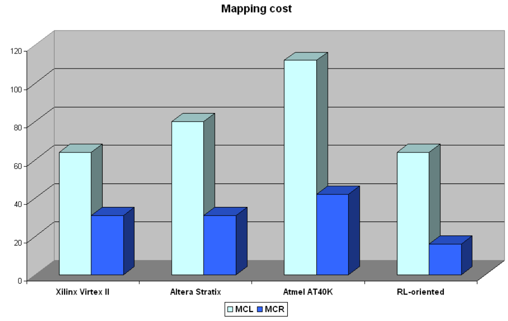


Figure 5.8. 8:1 1-bit multiplexer.

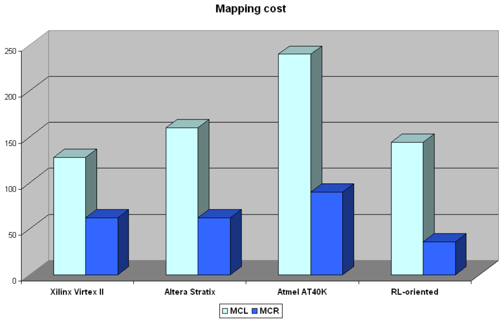


Figure 5.9. 16:1 1-bit multiplexer.

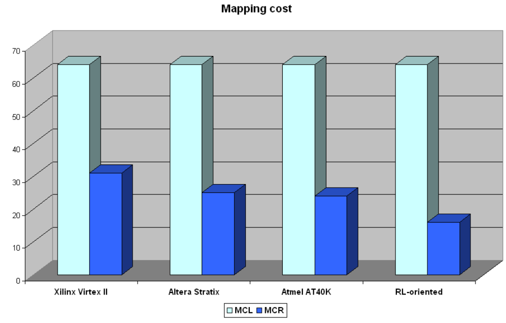


Figure 5.10. 2-input 4-bit OR.

	State-of-the-art commercial FPGA architectures									Proposed FPGA		
	Xilinx Virtex II			Altera Stratix			Atmel AT40K			Random-logic-oriented		
Benchmark	N_{LB}	Mapping cost		N_{LB}	Mapping cost		N_{LB}	Mapping cost		N_{LB}	Mapping cost	
function	–	MC_L	MC_R	–	MC_L	MC_R	–	MC_L	MC_R	–	MC_L	MC_R
8-bit ADD	1	128	62	0.8	128	50	8	128	48	2	128	32
16×16 MULT	34	4.25K	2.11k	73	11.41K	4.53k	256	4K	1.54k	120	7.5K	1.92k
2:1 MUX/4-bit	0.5	64	31	0.4	64	25	4	64	24	1	64	16
8:1 MUX/1-bit	0.5	64	31	0.5	80	31	7	112	42	1	64	16
16:1 MUX/1-bit	1	128	62	1	160	62	15	240	90	2.25	144	36
2-in OR/4-bit	0.5	64	31	0.4	64	25	4	64	24	1	64	16
3-in NOR/1-bit	0.125	16	8	0.1	16	6	1	16	6	0.25	16	4
16-in AND/1-bit	0.5	64	31	0.4	64	25	5	80	30	1.25	80	20
4:16 DECOD	2	256	124	1.6	256	99	16	256	96	4	256	64
16-long 2-bit SREG	0.25	32	16	3.2	512	198	32	512	192	8	512	128

Table 5.3. The mapping cost comparison between the state-of-the-art and proposed random-logic-oriented FPGA architectures.

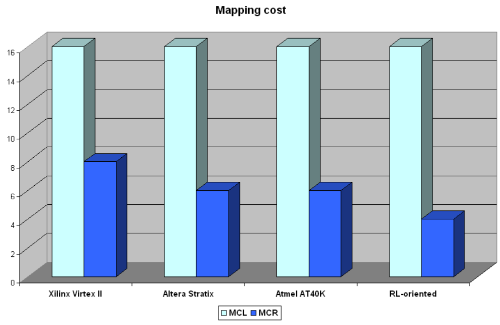


Figure 5.11. 3-input 1-bit NOR.

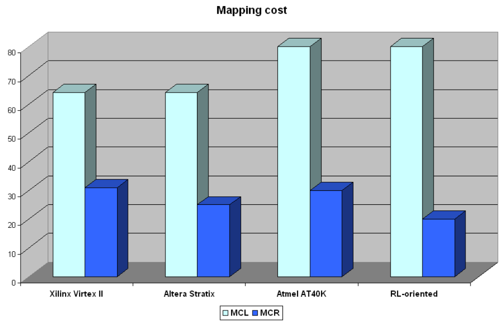


Figure 5.12. 16-input 1-bit AND.

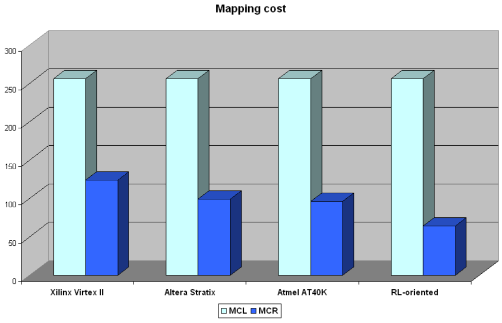


Figure 5.13. 4:16 decoder.

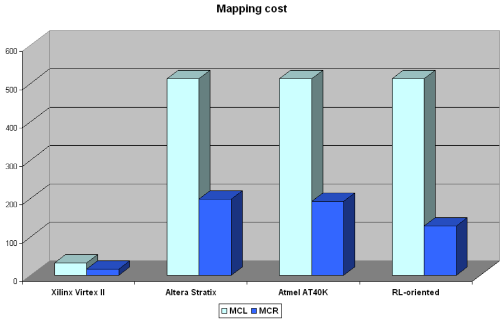


Figure 5.14. 16-long 2-bit shift register.

While offering the routing cost reduction, the random-logic-oriented architecture has its average logic cost similar to the logic cost of the commercial devices (factors 0.83, 1.07 and 1.1, respectively). Since the routing cost component plays an essential role (see Section 2.1.3), the fact of a similar logic cost is of lesser importance.

As shown by Figures 5.6 and 5.12, for example, the benefit of the random-logic-oriented architecture is sometimes diminished if data-path functions are mapped.

5.7 Conclusions

In this chapter we presented the random-logic-oriented reconfigurable architecture. The architecture has been optimised primarily for mapping multi-level combinational and sequential circuits. The logic block of the proposed architecture has been enhanced such that it also allows the mapping of arithmetic circuits.

The advantages of the random-logic-oriented architecture are a simple (small) logic block and a less expensive interconnect structure which does not sacrifice the overall routability. The architecture has been benchmarked with state-of-the-art FPGAs using the model-based comparison. We showed that at the comparable logic cost, the routing resource cost of three commercial FPGA architectures is about 1.64 times higher than the routing resource cost of our FPGA.

Chapter 6

MEMORY-ORIENTED RECONFIGURABLE ARCHITECTURE

Memory plays an essential role in some applications. The requirement for a memory is implied either by an algorithm or by its implementation (see Section 2.5). To address this aspect, in this chapter we present the *memory-oriented reconfigurable logic architecture,* which is the third type of the domain-oriented reconfigurable architectures proposed in this thesis. The memory-oriented architecture targets applications requiring distributed storage. Despite the optimisation towards the memory functionality, the architecture also provides an efficient way of mapping data-paths and random logic functions.

6.1 Introduction

6.1.1 Characteristics of the application domain

The memory functionality we aim at concerns distributed storage elements, such as small data memories, look-up tables and shift registers (see the discussion in Section 2.5.3). The storage resources are *distributed* because of the homogeneous structure that is assumed for a domain-oriented reconfigurable logic device (see Section 3.2.1). The LUT memory is reused to implement such storage resources.

As shown in Figure 6.1, a typical memory has the $Address$ port, multi-bit $Input$ and $Output$ ports (typically of the same width), and the $\overline{Read}/Write$ control port that receives the signal determining the reading or writing operation of the memory. When implemented in the FPGA logic block, the memory imposes different constrains on the logic block implementation than the data-path and random logic functions considered so far. For example, it needs more ports to map functions of the similar granularity and it supports a write operation. Therefore, assuming that the data-path and random logic functions are to be implemented using the same look-up table structure as the memory, the implementation of these functions will be influenced by the memory implementation. Especially, the bit-width of the LUT output will be impacted. This does not have to be disadvantageous if the properties of the mapped functions are exploited properly. Note, for example, that typical data-path functions (and arithmetic functions in particular) produce a multi-bit output. Such a multi-bit output signal can be generated cost-efficiently using a *multi-output look-up table* (see Section 3.1.3). Next, the implementation of DSP functions based on the concept of distrubuted arithmetic [111] yields multiple look-up tables that are addressed by the same set of input signals [113]. Obviously, such look-up tables can also be implemented as multi-output LUTs. Finally, it has been shown that many random logic functions share some of their input signals when generating different Boolean output signals [14, 61]. Multi-output LUTs are thus suitable for the implementation of such functions. A look-up table with relatively few inputs and multi-bit output reasonably suits the requirements of the memory, arithmetic and random logic functions.

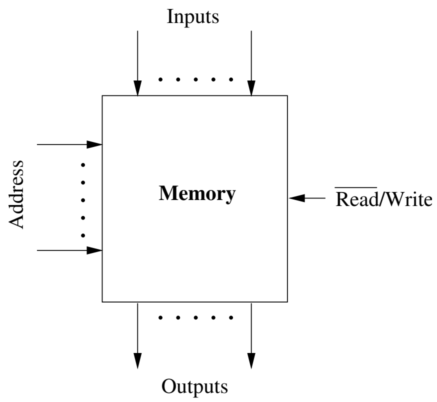


Figure 6.1. A model of a typical memory [108].

6.1.2 State-of-the-art

The idea of combining the memory, arithmetic and random logic functionality in a single logic block of an FPGA device has been explored in the past. Essentially, by reusing the LUT memory bits in the FPGA logic block, the implementation of small distributed data memories has been made possible. In Figure 6.2(a), the logic block structure of the Xilinx XC4000 device that has been designed in this way is shown[1]. The logic block consists of two 4-LUTs and one 3-LUT. The 4-LUTs can be used to implement Boolean functions (of up to 4-inputs) and arithmetic functions (of up to 2-bits). The implementation of the arithmetic functions is supported by the dedicated carry logic. Next to their traditional function, the 4-LUTs can be also configured as data memories (two 16 x 1-bit or one 32 x 1-bit). This is possible because such LUTs include both read and write decoders. An extra data input signal (for writing into the memory) is also added. Though very flexible, such a memory implementation has several disadvantages. Firstly, independent sets of inputs associated with each LUT increase the cost of the routing resources (especially that only one set of inputs is needed in the memory mode).

[1] A similar implementation approach has also been applied to the Xilinx Virtex family [118] and Lucent ORCA 3C [65] FPGA devices

Secondly, the LUT resources are underutilised in the arithmetic mode (only 25% of the memory bits in each LUT are effectively utilised). Finally, separate read and write decoders in the LUTs contribute to the area overhead.

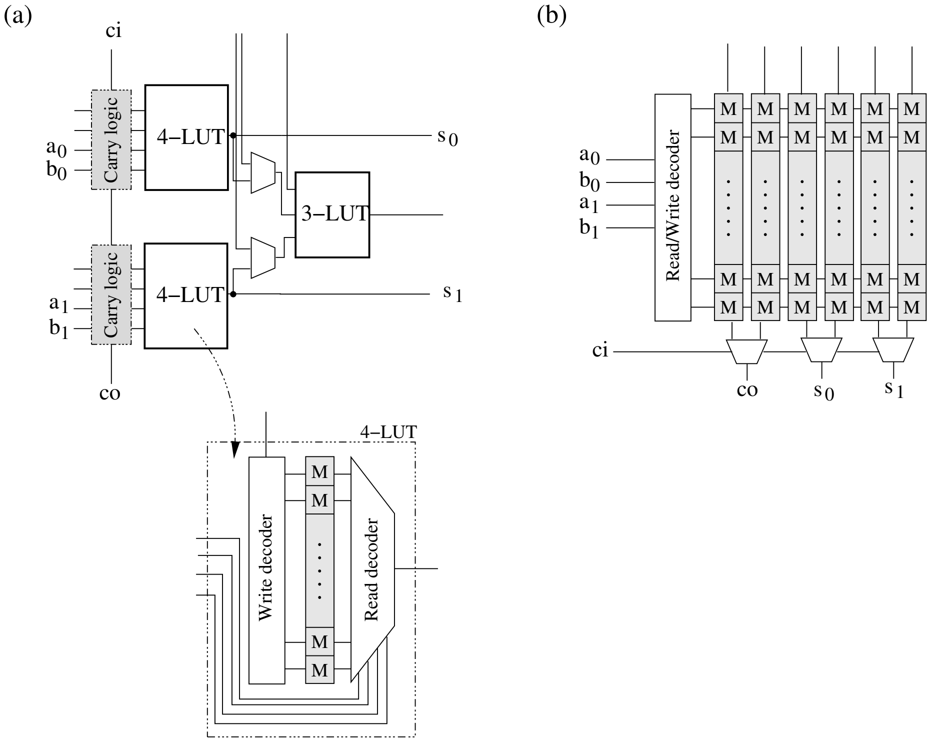


Figure 6.2. The LUT-based logic block structures offering the memory, data-path and random logic functionality: (a) a commercial FPGA device (Xilinx XC4000), and (b) a reconfigurable computing device proposed in academia. The figures illustrate the implementation cost of a 2-bit addition in terms of the required number of LUT memory bits (32 bits and 96 bits, respectively).

The alternative logic block structure supporting the memory mapping has been proposed at the Iowa University [63]. Because of its primary application as a reconfigurable cache [63], the logic block structure has been optimised for the memory functionality. As shown in Figure 6.2(b), such an optimisation has been achieved by implementing the LUT in the logic block in a memory-like way, that is, with one read/write decoder and several memory columns. The LUT structure is actually a *multi-output LUT* (compare Section 3.1.3). The logic block functionally equivalent to the Xilinx XC4000 logic block includes a multi-output LUT built of a 4:16 decoder and six memory columns[2]. Although such an implementation is efficient for the memory mapping (16 x 6-bits), it leads to a large cost overhead if other types of functions are implemented. For example, the implementation of arithmetic functions requires very many configuration bits (96 bits for a 2-bit

[2] The logic block of the structure described in [63] is coarser.

addition). Also, the implementation of Boolean functions is costly since most of them cannot be decomposed onto 4-input/6-output nodes (leaving most of the LUT memories unused).

6.2 Applying the inversion-based folding type II

To improve the cost-efficiency of the memory implementation in the LUT-based FPGA logic block, we use the multi-output LUT structure as implied by the inversion-based folding type II (proposed in Section 3.1.3). Note, that the complexity of the proposed multi-output LUT, even after the compaction (see Figure 3.11(a)), grows exponentially with the increase of the word-size of the input arguments of a binary adder[3]. Furthermore, it is also more beneficial (from the delay and memory size point of view) to generate the carry output signal *co* of the adder using the dedicated carry logic. Therefore, we choose the multi-output LUT with four inputs and two outputs, that is a *4/2-LUT*, for our implementation. The detailed structure of such a LUT is shown in Figure 6.3.

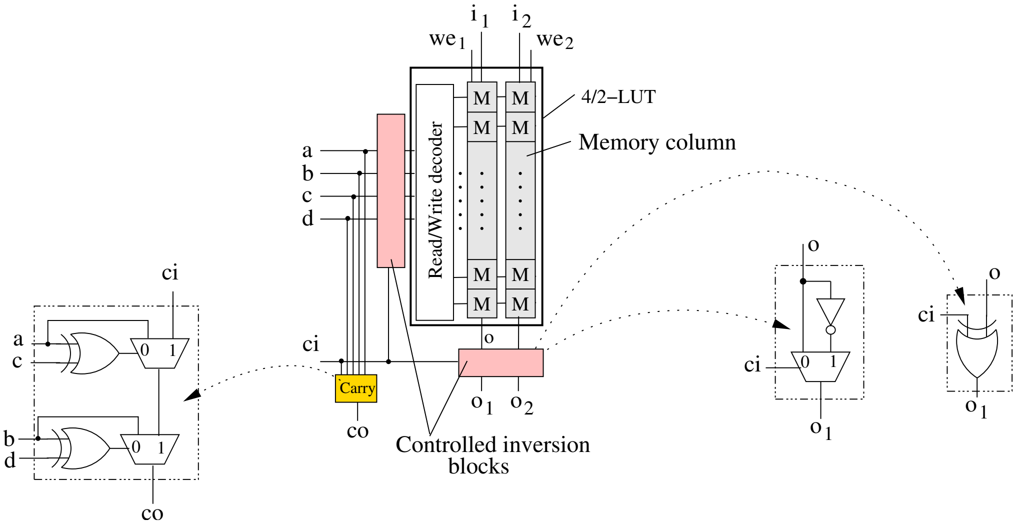


Figure 6.3. The structure of the proposed 4/2-LUT.

The 4/2-LUT is implemented in a memory-like way, that is, it contains a 4:16 decoder which addresses two memory columns of 16 bits (cells) each. To enable the implementation of a 2-bit addition, the four inputs *a*, *b*, *c*, *d* and two outputs o_1, o_2 of the LUT are connected to the controlled inversion blocks. The controlled inversion blocks invert the input (address) and output signals of the LUT if the carry input signal $ci = 1$. In order to use the LUT as a memory, four additional inputs i_1,

[3] For example, for a 3-bit adder, already 2^6 memory cells are needed to describe each of the four adder outputs in the resultant LUT.

i_2 and we_1, we_2 are added to the 4/2-LUT. When the look-up table works as a memory, the inputs i_1, i_2 play the role of data inputs, and the inputs we_1, we_2 receive the control (write enable) signals which determine the LUT operation (i.e. reading or writing). The 4/2-LUT can also be used for the mapping of Boolean functions with maximally four binary inputs and two binary outputs. Typical configurations of the proposed 4/2-LUT are shown in Figure 6.4.

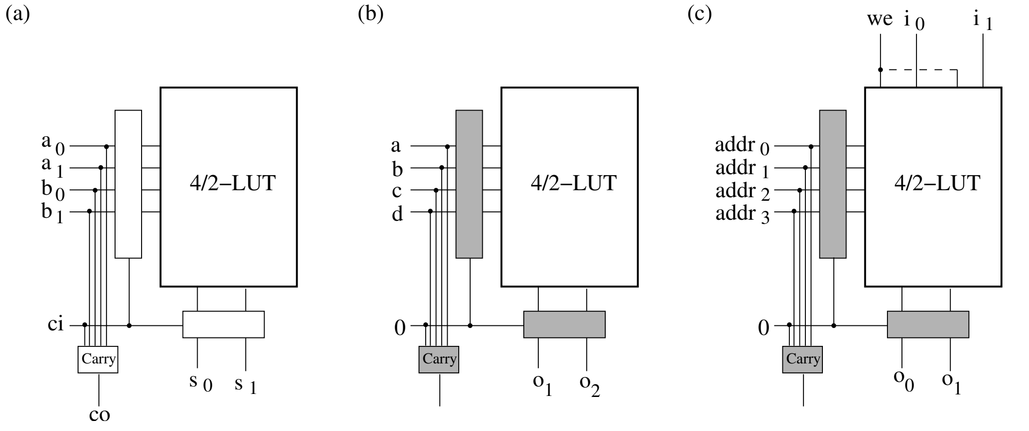


Figure 6.4. Example configurations of the 4/2-LUT: (a) 2-bit arithmetic (addition), (b) 4-input/2-output Boolean function generator, (c) 16 x 2-bit data memory. The shaded blocks show the components that are inactive in a given LUT configuration.

The above-described 4/2-LUT shows a number of advantages compared to the LUT-based memory implementations discussed in Section 6.1.2. Firstly, the 4/2-LUT reduces by a factor of 2 the number of LUT memory bits that would be needed if the state-of-the-art multi-output look-up table structure from Figure 6.2(b) was used to implement 2-bit arithmetic functions[4]. Secondly, compared to a traditional 4-LUT-based logic structure (see Figure 6.2(a)), the proposed LUT halves the number of required input ports (due to the sharing of inputs in the multi-output LUT), and thus the total amount of routing resources that are needed to map similar type of functions[5]. Thirdly, in the suggested 4/2-LUT implementation only one decoder is needed rather than four decoders (i.e. two write decoders and two read multiplexers) as in a typical implementation based on 4-LUTs (see Figure 6.2(a)). Finally, in contrast to the state-of-the-art multi-output LUT (see Figure 6.2(b)), the proposed 4/2-LUT enables a cost-efficient implementation of *any* type of functions, that is, memory, arithmetic, and random logic.

[4] For a fair comparison, we assume that the multi-output LUT architecture from Figure 6.2(b) may also be enhanced with a dedicated carry logic.

[5] The only penalty is an inability of mapping two *independent* Boolean functions of 4-inputs.

6.3 Logic element

The logic element constitutes the lowest level of hierarchy of the memory-oriented reconfigurable logic architecture. Because of the advantages of the 4/2-LUT (see Section 6.2), such a LUT is chosen for the implementation of the logic element. Several modifications and enhancements are introduced to the original 4/2-LUT structure to improve the mapping capabilities of the logic element.

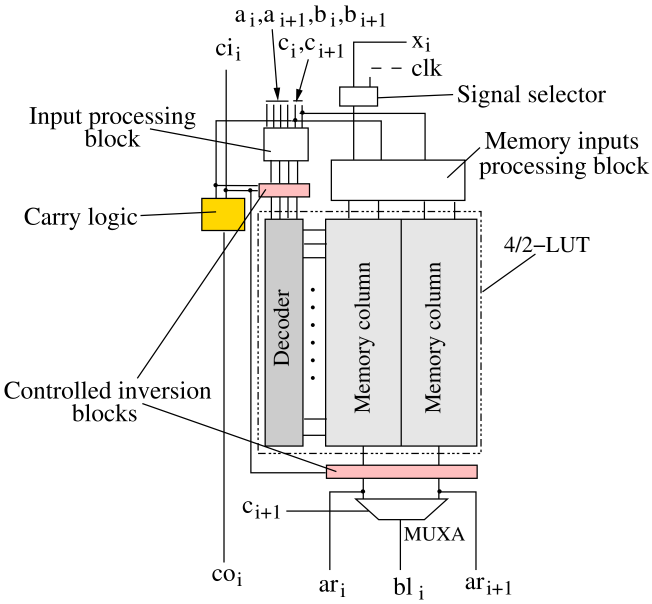


Figure 6.5. The logic element of the memory-oriented reconfigurable logic architecture.

The logic element has six primary inputs a_i, a_{i+1}, b_i, b_{i+1}, c_i, c_{i+1}, a secondary input x_i, a carry input ci_i, and a special clock input clk. It also has three primary outputs: two arithmetic outputs ar_i, ar_{i+1}, a Boolean output bl_i, and a carry output co_i. The general structure of the logic element is shown in Figure 6.5, while the implementation details are shown in Figure 6.6. The 4/2-LUT is implemented as described in Section 6.2. The *controlled inversion block* at the inputs of the 4/2-LUT is modified to the structure shown in Figure 6.6(a). The inversion of the LUT inputs, implemented with XOR gates, is no longer dependent on the carry input signal only (available on the input ci_i), but also depends on a control signal (available on the input c_i). This corresponds to the inversion of bits of one of the operands of a binary adder which is needed if the adder is configured as a subtractor. The change in the polarisation of the LUT (decoder) inputs $d_1 \ldots d_4$ and the LUT outputs o_1, o_2 is described in detail in Table 6.1. The table indicates that during an addition operation with $c_i = 1$ all LUT inputs are inverted, while during a subtraction operation with $c_i = 1$ only one pair of them. The inversion of the LUT outputs depends only on the carry input signal ci_i, and is implemented as shown in Figure 6.6(b). The configuration memory bit $M1$ in Figure 6.6(a)

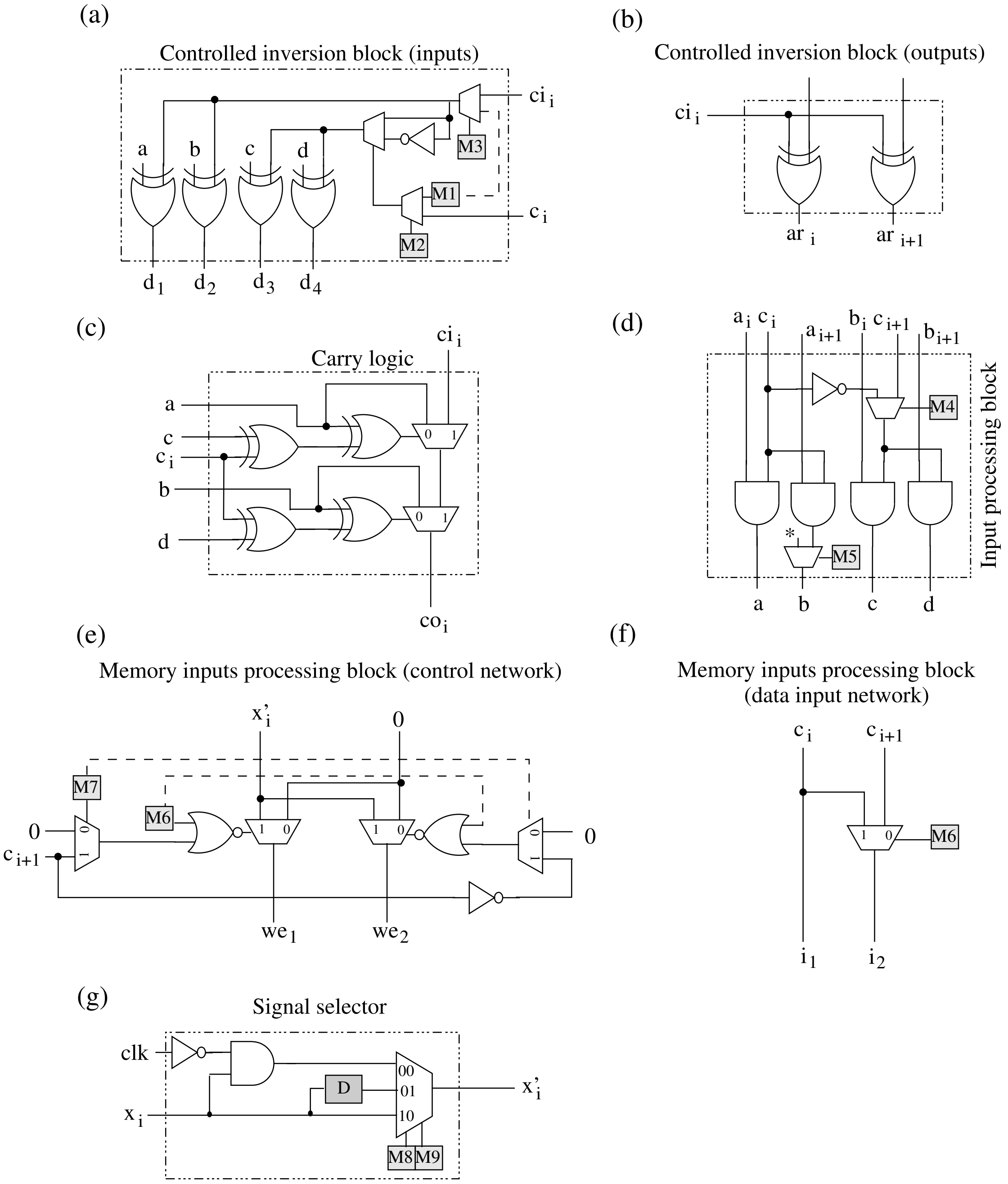


Figure 6.6. Implementation details of the logic element: (a) controlled inversion block at the inputs, (b) controlled inversion block at the outputs, (c) dedicated carry logic, (d) input processing block, (e) control network of the memory inputs processing block, (f) data input network of the memory inputs processing block, and (g) signal selector.

determines the type of arithmetic operation (i.e. addition or subtraction) that is implemented in the logic element, and the initial value of the carry input signal ci_i.

control ($\overline{add}/sub$) c_i	**carry in** ci_i	**inverted** $arg1$**?** d_1, d_2	**inverted** $arg2$**?** d_3, d_4	**inverted** res**?** ar_i, ar_{i+1}
0	0	not	not	not
0	1	yes	yes	yes
1	0	not	yes	not
1	1	yes	not	yes

Table 6.1. Polarisation of the bits of the input arguments $arg1$ and $arg2$ and bits of the result res of an arithmetic operation implemented in the proposed logic element. The polarisation depends on the carry input signal ci_i and the type of the mapped operation, which is determined by the control signal c_i. The change of the polarisation is the consequence of the applying the inversion-based folding type II and the way of implementing a subtraction operation in a binary adder. (The subtraction operation of the form $res = arg1 - arg2$ is assumed.)

The support of a subtraction operation has the impact on the implementation of the dedicated *carry logic*. This is indicated in Figure 6.6(c), where two additional XOR gates are added at the c and d inputs (bits of the argument $arg2$). The gates guarantee a proper value of the carry output signal co_i if a subtraction operation is performed.

The primary inputs a_i, a_{i+1}, b_i, b_{i+1}, c_i, c_{i+1} of the logic block are directed to the *input processing block*. The input processing block (see Figure 6.6(d)) reduces the number of input signals from six to four by performing logical AND and NOT operations on them. This is required, for example, for mapping of multiplexers and multipliers (see details in Section 6.5). The four outputs of the input processing block a, b, c, d are connected to the inputs of the controlled inversion block.

To enable the memory functionality, the *memory inputs processing block* is introduced to the logic element. This block provides proper values of the control and data signals to each memory column of the 4/2-LUT dependent on the type of a memory function that is to be implemented (e.g. a data memory or a shift register). The memory inputs processing block consists of the control (see Figure 6.6(e)) and data input networks (see Figure 6.6(f)). The control network produces two write enable signals we_1 and we_2 based on the value of the global write enable signal associated with the input x'_i, configuration memory bits $M6$ and $M7$, and the value of the signal c_{i+1}. The signal x'_i, which is an input of the control network, is defined by the *signal selector* (see Figure 6.6(g)). Dependent on the mode, the signal selector assigns the inverted clock signal clk, the external signal x, or the latched version of the signal x to the write enable signal x'. The data input network produces two data signals i_1 and i_2. Dependent on the configuration determined by the configuration bit $M7$, the LUT memory columns receive two different data signals available on the inputs c_i and c_{i+1}, or only one of them, that is c_i. This is illustrated in Figure 6.6(f)). The detailed description of the functionality of the input processing and memory inputs processing blocks is given in Section 6.5.

The outputs ar_i and ar_{i+1} of the controlled inversion block at the LUT outputs are connected to the inputs of the 2:1 multiplexer $MUXA$ (see Figure 6.5). The multiplexer has a twofold role. It is used to implement Boolean functions of five inputs by applying Shannon's expansion (see Equation 3.2) or to implement memory structures with a 1-bit output. The multiplexer $MUXA$ is controlled by the signal c_{i+1}. Dependent on the mode, the signal c_{i+1} is associated either with the fifth logic input (random logic mode) or with the fifth bit of the memory address (memory mode). The output of the multiplexer is available on the Boolean output bl_i of the logic element, while the LUT outputs on the arithmetic outputs ar_i, ar_{i+1}.

6.4 Logic block

6.4.1 Basic concept

The logic element described in the previous section allows the mapping of 2-bit arithmetic functions, 5-input random logic functions[6], and small memories with a 1-bit or 2-bit output. To improve the efficiency of mapping the arithmetic and memory type of functions, which are typically coarser than the granularity of the proposed logic element, the logic block of the memory-oriented reconfigurable logic architecture is built of *two* logic elements. As a result, the logic block supports the implementation of functions with up to a 4-bit output (nibble-level processing).

6.4.2 Structure in detail

The structure of the logic block is depicted in Figure 6.7. To account for the presence of two logic elements in the logic block, their original structure from Figure 6.5 is slightly modified. The modifications are essential for the memory mapping and concern the components which are shaded in Figure 6.7.

The logic block has twelve primary inputs $in_1 \ldots in_{12}$, two secondary inputs t_1, t_2, and a special clock input clk. It also has four primary outputs $out_1 \ldots out_4$ and two carry outputs co_1 and co_2. The first set of six primary inputs $in_1 \ldots in_6$ of the logic block is connected directly[7] to the primary inputs a_1, a_2, b_1, b_2, c_1, c_2 of the first logic element; similarly, the second set of six primary inputs $in_7 \ldots in_{12}$

[6] To comply with the architecture template discussed in Chapter 7, we assume here that the possibility of mapping 4-input/2-output Boolean functions in the 4/2-LUT is discarded.

[7] Unlike in the data-path-oriented and random-logic-oriented architectures, the input selection block is replaced by hard-wired one-to-one connections.

of the logic block is connected directly to the primary inputs a_3, a_4, b_3, b_4, c_3, c_4 of the second logic element. Because the pairs of the carry and secondary inputs ci_1, x_1 and ci_3, x_3 of the first and second logic element have a mutually exclusive function, they are merged at the logic block level into the secondary input signals t_1 and t_2, respectively. Therefore, the signals t_1 and t_2 are distributed to the carry selection and the signal selector blocks in the first and second logic element, respectively. The *carry selection* blocks determine whether the logic elements use two independent carry signals or one common carry signal when the logic elements are chained. In the latter case, either logic element can be chosen to produce less significant bits of an arithmetic operation, and consequently, has its carry output (co_1 or co_2) connected to one of the inputs of the carry selection block.

To enable the implementation of larger memories, the *memory inputs pre-processing block* is placed between the signal selector and the memory inputs processing blocks of the logic elements (compare with Figure 6.5). Dependent on the type of memory that is to be mapped, the memory inputs pre-processing and memory inputs processing blocks assign either one set or two independent sets of control and data signals to the 4/2-LUTs of the logic elements. The entire control and data input networks of the memory pre-processing and processing blocks are shown in Figure 6.8(a) and (b), respectively.

The *copy block* (see Figure 6.8(c)) placed between the 4/2-LUTs of the logic elements enables an implementation of dual-port memories. The copy block consists of two sets of 16 copy elements. The copy elements are controlled either by a static signal ‘0’ or by the signal u, which is connected to one of the signals x'_1 or x'_3 from the signal selector blocks in the first and second logic element, respectively (see Figure 6.8(a)). When activated ($M14 = 1$), the copy block copies the contents of the LUT memories in the first logic element to the LUT memories in the second logic element. This happens in the first phase of the inverted clock signal.

To account for the presence of two logic elements, the selection multiplexers $MUX1$ and $MUX2$ are introduced to the logic block, as shown in Figure 6.7. The multiplexers generate the control signals $s1$ and $s2$ by selecting one of the input signals in_5 (c_1) and in_{11} (c_3) of the logic block. The signals $s1$ and $s2$ control the multiplexers $MUX1$ and $MUX2$ in the first and second logic element, respectively. The Boolean outputs of the logic elements are merged in the multiplexer MUX and the gate *AND*. The multiplexer allows the implementation of Boolean functions with up to six inputs and the memory structures with the 6-bit address (i.e. 64 x 1-bit). The *AND* gate is used to implement large product functions.The control signal s of the multiplexer MUX is the output of the selection multiplexer $MUX3$, which selects one of the input signals in_6 (c_2) and in_{12} (c_4) of the logic block.

The arithmetic outputs $ar_1 \ldots ar_4$, Boolean outputs bl_1, bl_3, and the outputs of the global multiplexer MUX and the global gate *AND* are directed to the *output selection block*. The output selection block has two stages. In the first stage, the

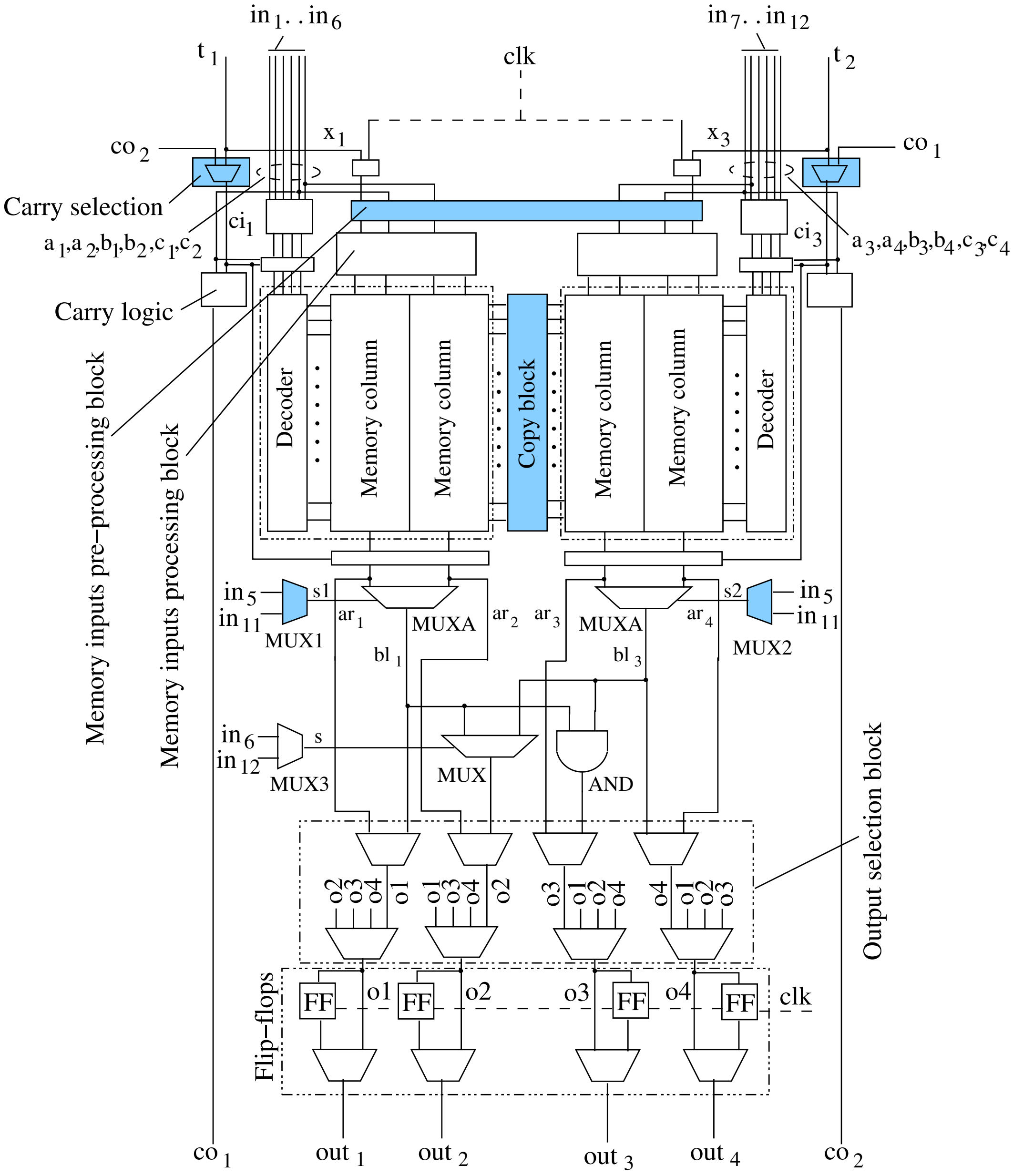


Figure 6.7. Logic block structure of the memory-oriented reconfigurable logic architecture.

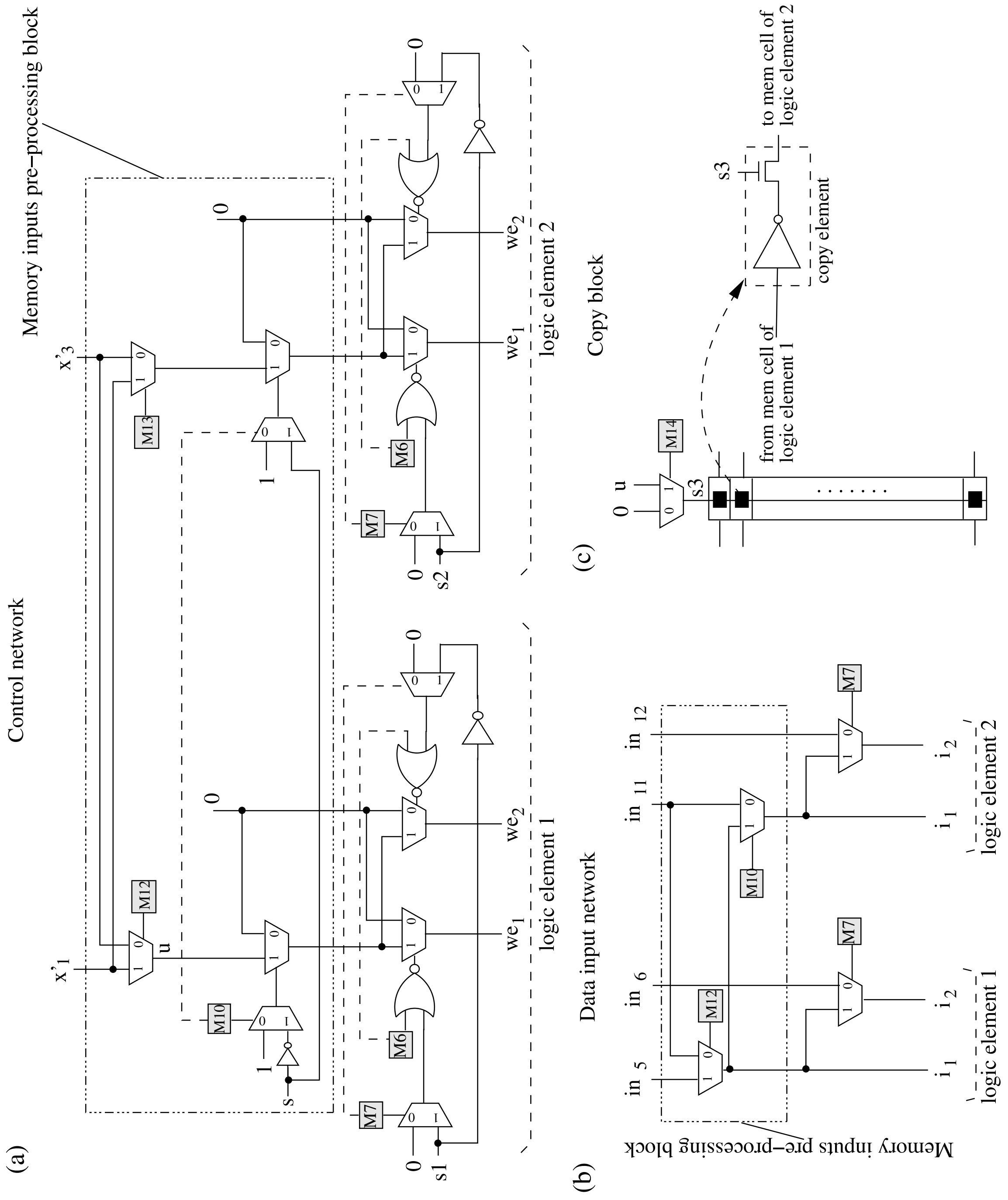


Figure 6.8. Implementation details of the logic block components. (a) control network of the memory inputs pre-processing and processing blocks, (b) data input network of the memory inputs pre-processing and processing blocks, (c) copy block.

selection of the output signals that are relevant in a given mode takes place. In the second stage, the selected signals are associated with particular outputs of the logic block. The outputs of the output selection block are fed to the *flip-flip block* which allows the output signals to be registered. The outputs of the flip-flop block are connected to the outputs $out_1 \ldots out_4$ of the logic block.

6.5 Functional modes

The logic block of the memory-oriented reconfigurable architecture has three primary functional modes. The functional modes determine the type of function which is implemented in both logic elements of the logic block (for large functions) or in each of the logic elements independently (for small functions). The following modes are available:

- *Data-path mode* in which the logic elements produce two independent 2-bit output signals or a single 4-bit output signal of data-path functions. The signals are available on the arithmetic outputs $ar_1 \ldots ar_4$ of the logic elements.
- *Random logic mode* in which two 1-bit output signals of two independent 5-input Boolean functions or a 1-bit output signal of a 6-input Boolean function are produced. The outputs of the independent functions are available on the Boolean outputs bl_1 and bl_3 of the logic elements. The output of the larger function is available on the output of the global multiplexer MUX or the output of the gate AND.
- *Memory mode* in which two independent memory functions with 1-bit or 2-bit output signals or one larger memory function with 1-bit, 2-bit or 4-bit output signals are produced. The 2-bit output signals of independent memory functions and the 4-bit output signal of the large memory function are available on the arithmetic outputs $ar_1 \ldots ar_4$ of the logic elements. The 1-bit output signals of the independent memory functions and the 2-bit output signal of the large memory function are available on the Boolean outputs bl_1 and bl_3 of the logic elements. The 1-bit output signal of the large memory function is available on the output of the global multiplexer MUX.

6.5.1 Data-path mode

Addition/Subtraction. The logic block can implement addition, subtraction, or addition/subtraction operations with up to 4-bit arguments. The input arguments are connected to the inputs $in_1 \ldots in_4$ and $in_6 \ldots in_{10}$ of the logic block, and the outputs are available on the arithmetic outputs $ar_1 \ldots ar_4$ of the logic elements. If a 4-bit operation is implemented, the inputs t_1 or t_2 of the logic block provide a carry input signal, and the logic element implementing more significant bits of the operation has its carry input signal connected internally to the carry output of the other logic element (i.e. ci_3=co_1 or ci_1=co_2; a direction of the carry flow is arbitrary). The carry output signal is available on one of the carry outputs co_1 or co_2 of the logic block. If independent 2-bit arithmetic operations are implemented, each logic element has its own carry input and carry output signals.

The addition operation (see an example in Figure 6.9(a)) is implemented using the inversion-based folding type II as described in Section 3.1.3. Consequently, the 4/2-LUTs in both logic elements implement a 2-bit addition. The LUT inputs and outputs are inverted if the carry input signal is ‘1’. The subtraction operation uses the LUT configuration as for a 2-bit addition, but the inversion of the LUT inputs, outputs, and the carry input signal is modified according to the scheme from Table 6.1. The configuration of the logic element to perform the addition or subtraction operations is determined by the bit $M1$ in the controlled inversion block (see Figure 6.6(a)), that is, $M1 = 0$ for the addition and $M1 = 1$ for the subtraction. The (dynamic) addition/subtraction operation uses an external signal ($\overline{add}/sub$) to determine the type of executed operation. Such as signal is available on the inputs $in5$ (c_1) or in_{11} (c_3) of the logic block.

Multiplication. The logic block supports the implementation of the carry-save type of an array multiplier [80]. Such a multiplier is characterised by the vertical propagation of the carry signal (except for the last row), and is faster than a traditional array multiplexer with a horizontal carry propagation [80, 70]. The structure of the logic block that has been described in Section 6.4 allows an implementation of the two’s complement unsigned version of the carry-save multiplier (the so-called Braun multiplier [55]). After small modifications of the proposed logic block structure, the implementation of the signed version of the multiplier (the so-called Pezaris multiplier [55]) would also be possible.

The logic block implements up to four cells of an array multiplier, which are organised in a 2 x 2 array as shown in Figure 6.9(b). Each pair of vertical cells of the array is implemented in a single logic element. Each logic element receives maximally five distinct input signals on five out of six primary input ports of the logic block, that is $in_1 \ldots in_6$ and $in_7 \ldots in_{12}$, and the carry input signal on the inputs t_1 and t_2 of the logic block. Each logic element also produces maximally two output signals available on the arithmetic outputs ar_1, ar_2 or ar_3, ar_4, and the carry output signal available on the outputs co_1 and co_2 of the logic block. The multiplier mapping technique is explained in detail in Figure 6.10. The input signals of the vertical cells of the multiplier are connected to the inputs of the 4/2-LUT in a logic element via the input processing block. The input processing block implements logic AND operations on pairs of bits, similarly to the AND operation that takes place in a basic cell of the array multiplier (see Figure 6.10(a)). The 4/2-LUTs in each logic element are configured as for a 2-bit addition operation[8].

[8] The use of the inversion-based folding type II and the 4/2-LUT to implement a 2-bit addition implies that a carry signal (e.g. signal c in Figure 6.10(b)) that traverses vertically between pairs

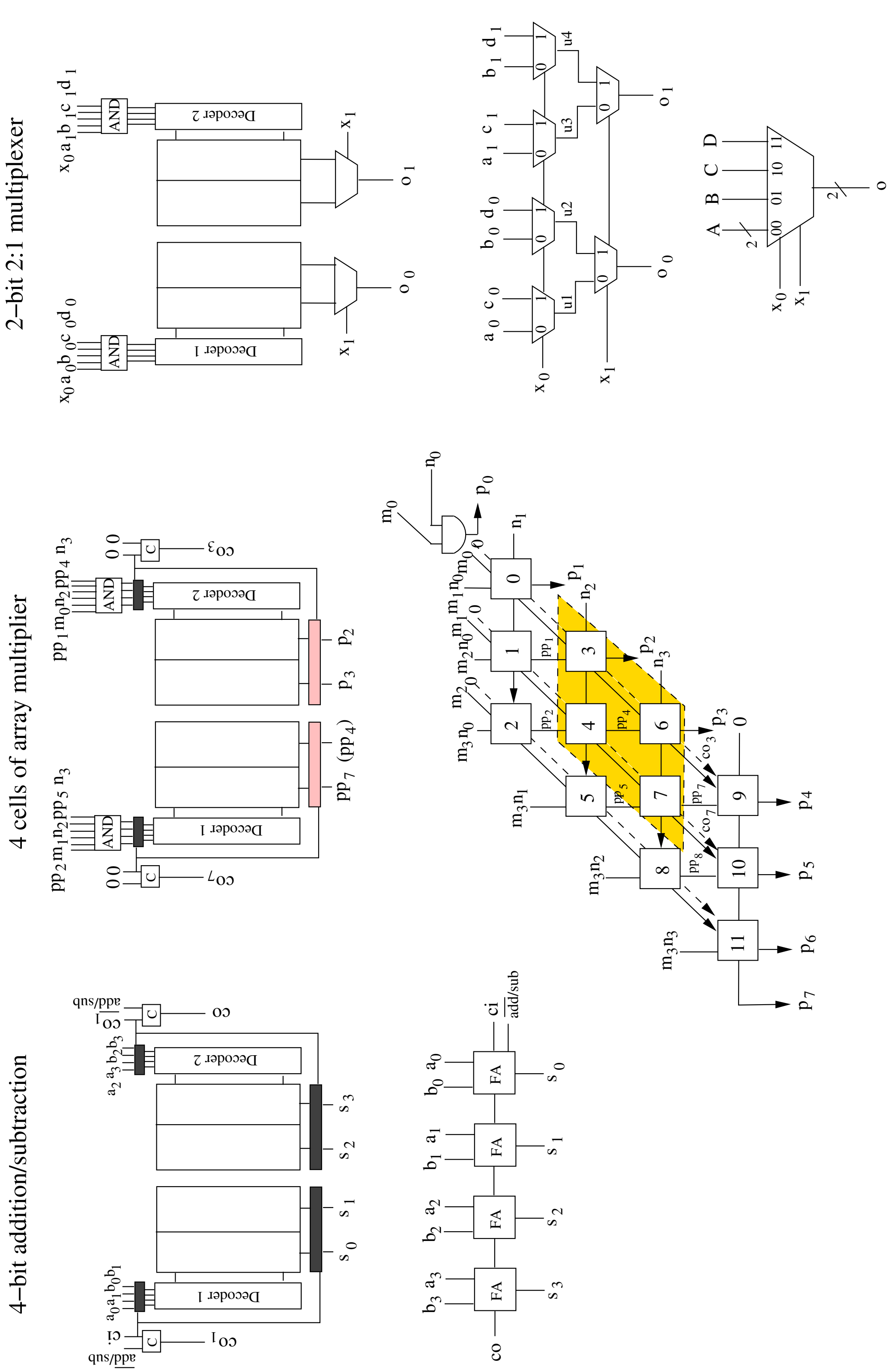


Figure 6.9. Configuration of the memory-oriented reconfigurable logic block in the datapath mode. (a) 4-bit carry-ripple adder, (b) four cells of the carry-save array multiplier, (c) 4-bit 2:1 multiplexer.

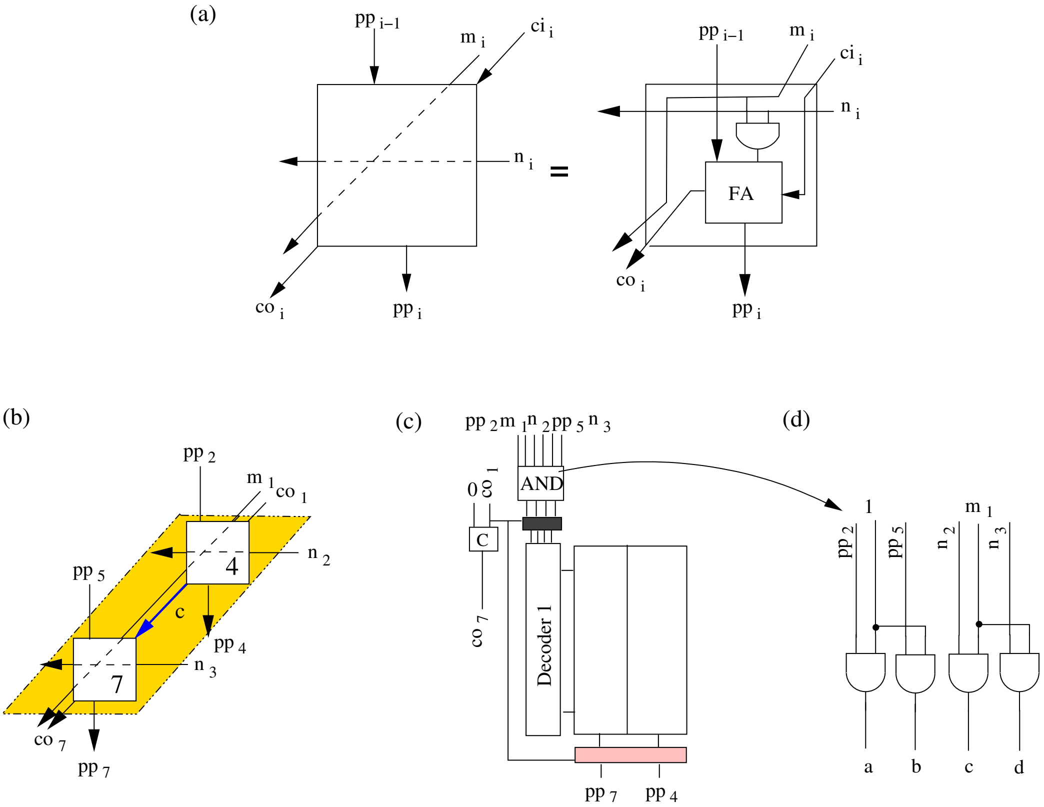
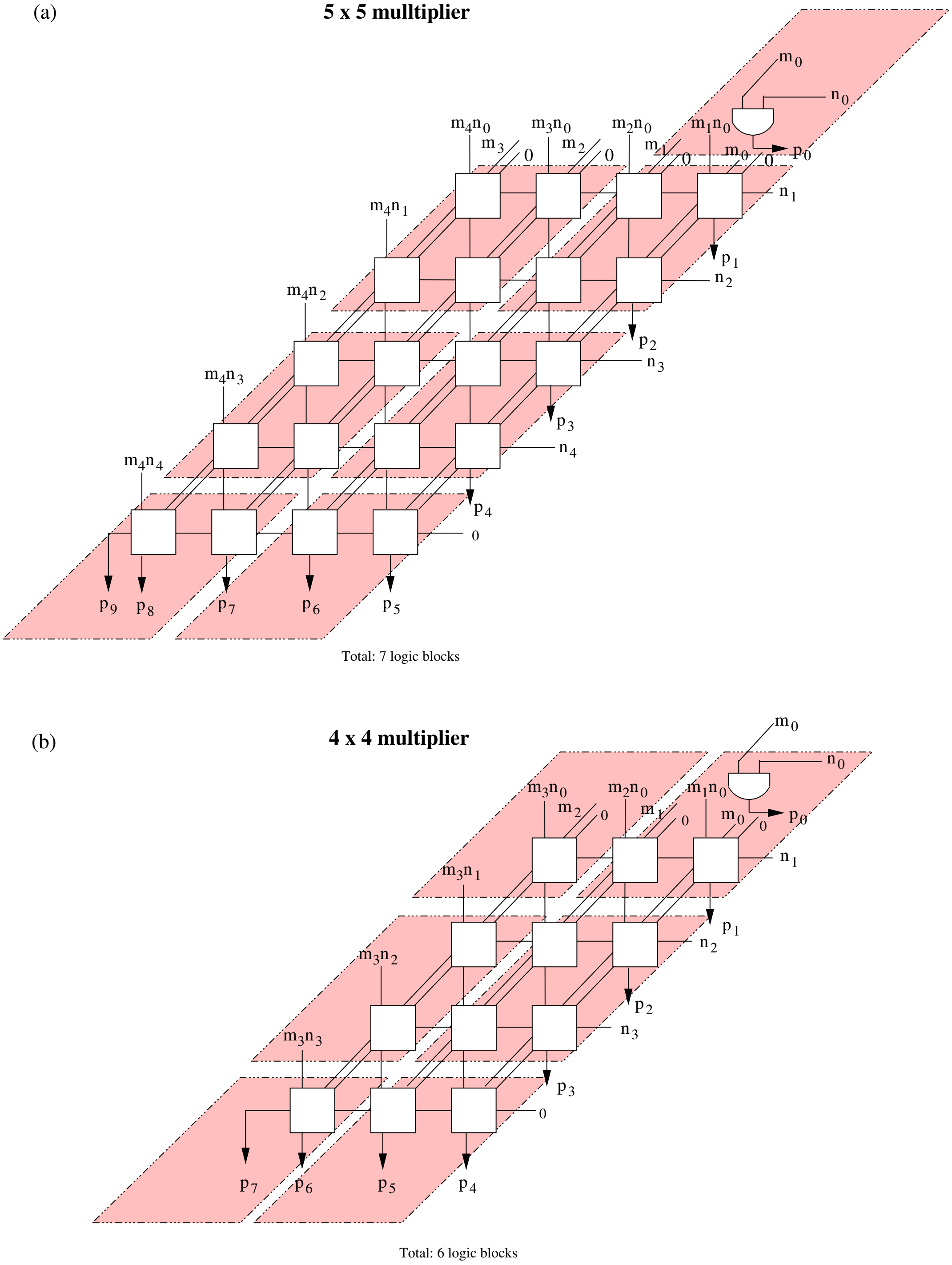


Figure 6.10. The multiplier mapping technique: (a) a basic multiplier cell, (b) the pair of vertical cells of the multiplier (the cells correspond to the cells no. 4 and 7 in Figure 6.9(b)), (c) configuration of the logic element, (d) details of the configuration of the input processing block.

Typically, an $m \times n$ carry-save multiplier, where m and n are the number of bits of a multiplicand and multiplier, respectively, can be implemented using an $n \times \ (m-1)$ array of basic cells, such as the cell shown in Figure 6.10(a) [55]. As mentioned above, four cells of the carry-save multiplier can be implemented in a single logic block. Nevertheless, the number of logic blocks that are needed to map the entire multiplier according to the proposed technique differs slightly from the expected number of $\frac{n(m-1)}{4}$ cells. The reason is the horizontal (instead of the vertical) propagation of the carry signal in the last row of the multiplier, which enforces the mapping of this row onto a separate set of logic blocks. In consequence, the implementations of $m \times n$ carry-save multipliers with even and odd number of bits of the multiplier n differ. When n is even, the first row of the multiplier cells and the AND gate generating the output signal p_0 are mapped onto a single set of logic blocks, as shown in Figure 6.11(a). If n is odd, the first two rows of the multiplier cells are mapped onto a single set of logic blocks, and the AND gate requires an extra logic block, as shown in Figure 6.11(b).

of the multiplier cells is now encoded internally in the 4/2-LUT.



(a) **5 x 5 mulltiplier**

Total: 7 logic blocks

(b) **4 x 4 multiplier**

Total: 6 logic blocks

Figure 6.11. Mapping of array multipliers with the (a) odd and (b) even number of bits of the multiplier. The shaded blocks show an assignment of the multiplier cells to the reconfigurable logic blocks.

Constant-coefficient multiplication. A special type of multiplication is the multiplication with a constant argument (e.g. $n = const$). This type of multiplication is very common in digital signal processing where many transforms and filtering operations rely on multiplications with constant coefficients. A constant coefficient multiplier can be implemented using the so-called *distributed arithmetic* [111]. Distributed arithmetic is simply (but not necessarily) a bit-serial computing technique that calculates an inner (dot) product of a pair of bit-vectors in a single direct step [111]. Distributed arithmetic is a cost-efficient implementation technique since it reduces a multiplication operation to the series of additions, subtractions, and binary scalings.

$$y[n] = \sum_{k=1}^{K} A_k x_k[n] \tag{6.1}$$

Equation 6.1 shows the sum of products which represents the response $y[n]$ of a linear time-invariant network at discrete time n. $x_k[n]$ is a sample of an input signal and A_k is a coefficient. If we drop the time index $[n]$, assume the two's complement representation of the input sample x_k, that is,

$$x_k = -b_{k0} + \sum_{n=1}^{N-1} b_{kn} 2^{-n}, \tag{6.2}$$

and scale x_k for convenience such that $|x_k| < 1$ [111], then

$$y = \sum_{n=1}^{N-1} [\sum_{k=1}^{K} A_k b_{kn}] 2^{-n} + \sum_{k=1}^{K} A_k(-b_{k0}). \tag{6.3}$$

Because b_{kn} represent successive bits of the input sample and take on values 0 and 1, the bracket term $\sum_{k=1}^{K} A_k b_{kn}$ may have only 2^K possible values. Instead of computing these values on line, they can be precomputed and stored in the memory (e.g. ROM). In practice, look-up tables addressed with equally significant bits of K input samples (e.g. x_{k0}), are used for this purpose. An example realisation of this concept is shown in Figure 6.12.

Since the result of the inner product $A_k b_{kn}$ is a complete word rather than a single bit, look-up tables in distributed arithmetic have multi-bit outputs. This fact makes the logic block of the proposed memory-oriented architecture superior to the logic blocks of traditional FPGAs. This is because a basic building block of the memory-oriented logic block is a cost-efficient 4/2- LUT (the LUT has simpler decoder logic and lower amount of routing resources because of the LUT inputs sharing).

Data-path multiplexers. The logic block can be configured to implement data-path multiplexers with two or four inputs, each of the inputs having 2-bits or 4-bits. A mapping example of a 2-bit 4:1 multiplexer is shown in Figure 6.9(c).

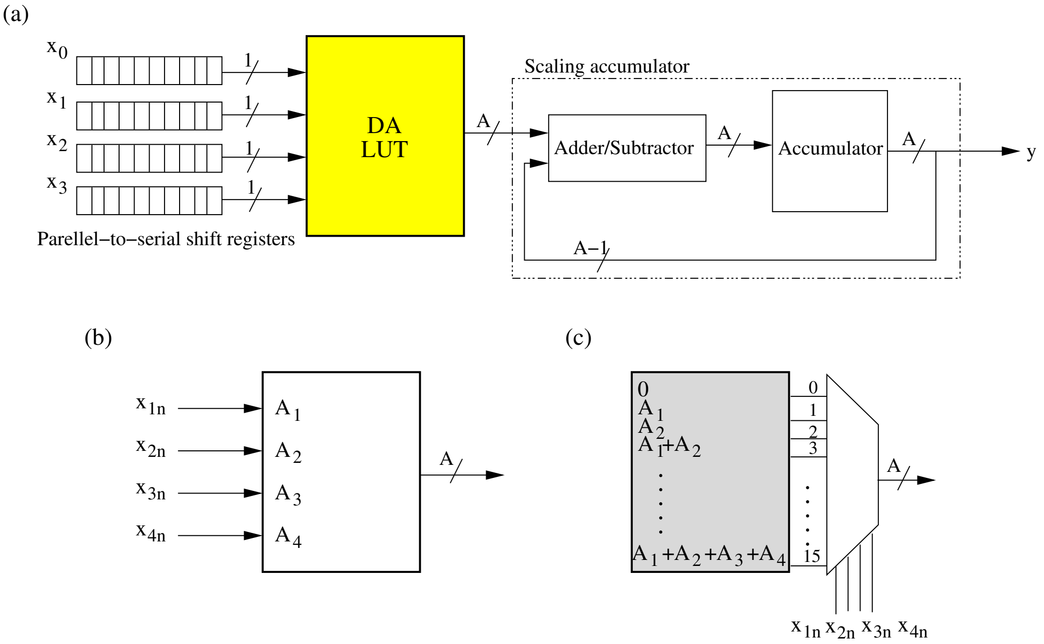


Figure 6.12. The bit-serial realisation of the sum of products $y = \sum_{k=1}^{4} A_k x_k$ using distributed arithmetic. The 10-bit representation (i.e. $N = 10$) of the samples x_k of the input signal is assumed. (a) implementation architecture, (b) LUT addressing, (c) LUT implementation and configuration.

To enable the implementation of data-path multiplexers in the proposed logic block, the multiplexers have to be first decomposed into a network of 2:1 multiplexers. The first level of multiplexers of such a network is then implemented in the 4/2-LUTs, while the second level in the multiplexers *MUXA* of the logic elements. Because each pair of 2:1 multiplexers in the first layer has five distinct input signals (see Figure 6.9(c)), their direct implementation in the 4/2-LUT having only four inputs is impossible. Therefore, the number of distinct input signals is first reduced to four signals. Such a reduction takes place in the input processing blocks of the logic elements, each of which receives four bits of the data inputs

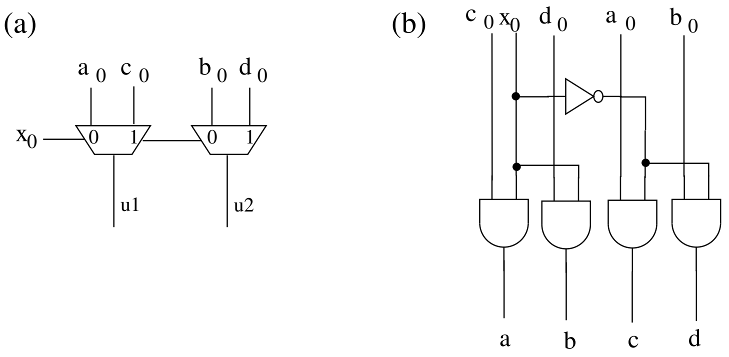


Figure 6.13. The implementation of multiplexers in the 4/2-LUT: (a) mapping of a 2-bit multiplexer structure, (b) the required configuration of the input processing block.

(a_0, b_0, c_0, d_0 in Figure 6.13(a)) and the least significant control bit (x_0 Figure 6.13(a)) of the mapped multiplexer. The input processing blocks perform logical AND operations, generating intermediate signals a, b, c and d, as described (for the input processing block of the first logic element) by Equation 6.4.

$$a = c_0 \cdot x_0, \qquad b = d_0 \cdot x_0, \qquad c = a_0 \cdot \overline{x_0}, \qquad d = b_0 \cdot \overline{x_0} \tag{6.4}$$

The signals a, b, c, d are directed to the inputs of the 4/2-LUTs. The LUTs are configured to execute an OR operation. The 4/2-LUT in the first logic element generates the signals $u1$, $u2$ on its arithmetic outputs ar_1, ar_2, and the 4/2-LUT in the second logic element generates the signals $u3$, $u4$ on its arithmetic outputs ar_3, ar_4 (see Figure 6.9(c)). The signals $u1$ and $u2$ of the first logic element are described by Equations 6.5 and 6.6, respectively.

$$u1 = c + a \tag{6.5}$$

$$u2 = b + d \tag{6.6}$$

If a 4:1 data-path multiplexer is mapped, the pairs of signals $u1$, $u2$ and $u3$, $u4$ are connected further to the inputs of the multiplexers $MUXA$ in the first and second logic element, respectively. The multiplexers are controlled by the second control signal of the mapped multiplexer (x_1 in Figure 6.9(c)), and generate the output signals available on the Boolean outputs bl_1 and bl_3 of the logic elements.

Multi-bit Boolean functions. The logic block enables an implementation of 2-input Boolean functions with 2-bit or 4-bit inputs. Such functions are implemented in the 4/2-LUTs of the logic elements. The inputs of multi-bit Boolean functions are available on the inputs $in_1 \ldots in_4$ and $in_7 \ldots in_{10}$ of the logic block, while the successive bits of the output on the arithmetic outputs $ar_1 \ldots ar_4$ of the logic elements.

6.5.2 Random logic mode

Boolean functions. In the random logic mode, multi-input Boolean functions can be mapped. Such functions are implemented using the 4/2-LUTs, the multiplexers $MUXA$ of the logic elements and the global multiplexer MUX of the logic block. When configured independently, the logic elements allow mapping of two Boolean functions with up to 5-inputs each. In such a case, the inputs of the functions are available on the inputs $in_1 \ldots in_5$ and $in_7 \ldots in_{11}$ of the logic block, and two outputs of the functions on the Boolean outputs bl_1 and bl_3 of the logic elements. When combined together, the logic elements allow the implementation of a Boolean function with 6-inputs. The 6-input Boolean function uses the same inputs of the logic block as the 5-input functions, and the sixth input is available either on the input in_7 or in_{12} of the logic block. The output of the 6-input Boolean function is generated by the global multiplexer MUX.

The logic block also provides a special support for the implementation of large product or sum of products functions. Such functions are implemented using the global *AND* gate located in parallel to the global multiplexer MUX.

Random logic multiplexers. Random logic multiplexers are implemented similarly to the data-path multiplexers, but have binary inputs and outputs. Each logic element can implement a 2:1 or 4:1 multiplexer. When combined together, the logic elements can also implement a 8:1 multiplexer using the global multiplexer MUX of the logic block. The third control signal of the 8:1 multiplexer can be connected to either the input in_7 or the input in_{12} of the logic block. The selected signal is assigned to the selection signal s of the global multiplexer MUX (see Figure 6.7). The outputs of the 2:1 and 4:1 multiplexers are available on the Boolean outputs bl_1, bl_3 of the logic block, and the output of the 8:1 multiplexer is available on the output of the global multiplexer MUX.

6.5.3 Memory mode

Data memory. The logic block allows the implementation of two independent or one large single-port data memories of the following types:

- two 16 x 2-bit memories
- two 32 x 1-bit memories,
- one 64 x 1-bit memory.

The data memories are implemented in the 4/2-LUTs of the logic elements and use the memory inputs pre-processing block, memory inputs processing blocks, and optionally the global multiplexer MUX. The type of the mapped memory is established by the configuration bits $M6$ and $M7$ in the memory inputs processing blocks of the logic elements, and by the bit $M10$ in the memory inputs pre-processing block (see Figure 6.8(a) and (b)) according to the following scheme.

- Bit $M6$ determines the functional mode of a single logic element. Bit $M6 = 0$ if the logic element is used as a data memory or a shift register, and $M2 = 1$ if the 4/2-LUT in the logic element is used to implement logic.
- Bit $M7$ determines the number of data inputs (data outputs) of the memory implemented in one logic element. That is, $M7 = 0$ if such a memory has two data inputs, and $M7 = 1$ if it has only one input.
- Bit $M10$ determines whether memories implemented in the logic elements work independently or whether they are combined together forming a larger memory. That is, $M10 = 0$ if the logic elements work independently, and $M10 = 1$ if the logic elements are combined together.

Because the implementation of different types of data memories is analogous, we describe as an example the implementation of the most complex, that is, a 64 x 1-bit memory. As shown in Figure 6.14(a), the 64 x 1-bit memory has a 6-bit address $addr_0 \dots addr_5$, a 1-bit data input in_0, a 1-bit data output out_0, and a write enable signal we. The four least significant bits of the address are connected to the inputs $in_1 \dots in_4$ and $in_6 \dots in_{10}$ of the first and second logic element. The fifth address bit is assigned to the input in_5 or in_{11} of the logic block, and the sixth signal to the input in_6 or in_{12}. Dependent on which of the inputs is selected as the fifth address bit, the data input signal of the memory is connected to the input in_{11} or in_5 of the logic block. The write enable signal of the memory can be connected to one of the secondary inputs of the logic block, that is, t_1 or t_2. Finally, the data output of the memory is available on the output of the global multiplexer MUX and can be assigned to any of the outputs $out_1 \dots out_4$ of the logic block. The configuration bits are set such that $M6 = 0$, $M7 = 1$, and $M10 = 1$.

The 64-bit capacity of the 64 x 1-bit memory is divided into four sections such that the first and second column of the first 4/2-LUT store bits $0 \dots 15$ and $16 \dots 31$, respectively, and the first and second column of the second 4/2-LUT store bits $32 \dots 47$ and $48 \dots 64$, respectively (see Figure 6.14(a)). The most significant bit of the memory address, which is coupled to the signal s (see Figure 6.8(a)), selects one of the logic elements. The fifth memory address bit, which is coupled to the signal $s1 = s2$ (see Figure 6.8(a)), selects the memory column of the 4/2-LUT in the selected logic element. The remaining four bits of the memory address, which are connected to the LUT inputs $d_4 \dots d_1$ (see Figure 6.6(a)), choose the memory location within the selected memory column that is accessed. During the write operation ($we = 1$), the control networks of the memory inputs pre-processing and processing blocks (see Figure 6.8(a)) take care that only the selected memory column receives the original value of the write enable signal we, which is accessible through one of the inputs x'_1 or x'_3 of the logic elements; the other memory columns have their we_1 and we_2 inputs coupled to '0' at that time. (This guarantees that inactive memory columns are read rather than written.) Similarly, the data input networks of the memory inputs pre-processing and processing blocks (see Figure 6.8(b)) take care that only the selected memory column receives the valid data input signal in, which is accessible through the chosen input in_5 or in_{11} of the logic block. During the read operation ($we = 0$), the signals $s1 = s2$ and s (see Figure 6.7), which are associated with the fifth and sixth memory address bits, respectively, assure that the value from the selected memory location is directed to the output of the global multiplexer MUX.

Dual-port memory. The logic block also allows the implementation of dual-port memories. As shown in Figure 6.15, a dual-port memory enables a simultaneous read operation from two independent memory locations (*address* 1 and *address* 2), and a write operation to one of those memory locations (*address* 1). Similarly to [117], we assume that one of the read memory addresses and the write address are

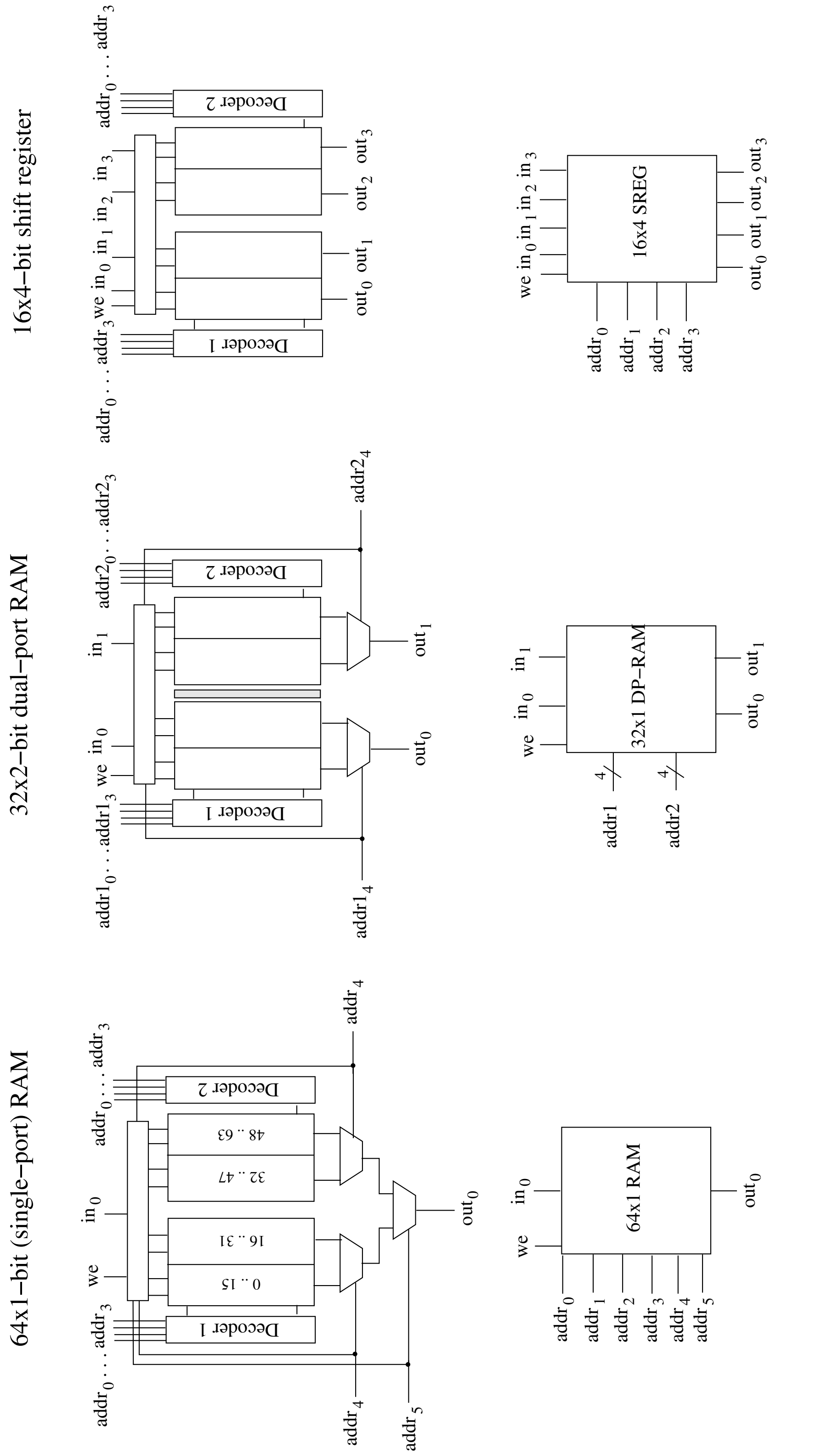


Figure 6.14. The configuration of the memory-oriented reconfigurable logic block in the memory mode. (a) 64 x 1-bit data memory, (b) 16 x 2-bit dual-port memory, (c) 16 x 4-bit shift register.

the same.

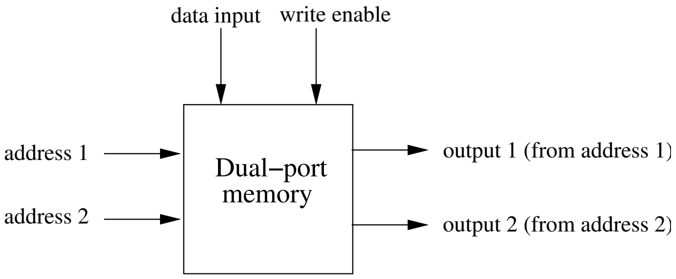


Figure 6.15. A dual-port memory.

Because the implementation of a dual-port memory in the proposed logic block involves both logic elements, only the following memory configurations are possible:

- 16 x 2-bit dual-port memory,
- 32 x 1-bit dual-port memory.

The dual-port memory uses the inputs $in_1 \dots in_4$ and $in_7 \dots in_{10}$ of the logic block, and optionally also the inputs in_5 and in_{11}, as two sets of address inputs. It also uses the input in_5 and optionally the input in_6 as data inputs. The write enable signal of the mapped memory is assigned to one of the secondary inputs of the logic block, that is, t_1 or t_2. The data outputs of a dual-port memory are always available on the outputs of *both* logic elements. If a 2-bit dual-port memory is mapped, the pairs of the arithmetic outputs ar_1, ar_2 and ar_3, ar_4 of the logic elements provide two bits of the first and second read-out value, respectively. If a 1-bit dual-port memory is mapped, the Boolean outputs bl_1 and bl_3 of the logic elements provide one bit of the first and second read-out value, respectively. A mapping example of a 16 x 2-bit dual-port memory is shown in Figure 6.14(b).

The operation of the dual-port memory is based on the *read-before-write* scheme. The read operation takes place in the first half of the clock cycle, while the write operation uses the second half. Therefore, the write enable signals we_1 and we_2 of the 4/2-LUTs in the logic elements are connected to the inverted clock signal $\overline{clk}$ [9], which is *AND*-ed with the write enable signal of the memory. Consequently, $M8 = 0$ and $M9 = 0$ in Figure 6.8(g). If a dual-port memory is implemented, the copy block of the logic block is activated, that is, $M14 = 1$ in Figure 6.8(c).

The detailed configuration of the logic block when it is used as a 16 x 2-bit dual-port memory is illustrated in Figure 6.16(a). During the write operation (see Figure 6.16(b)), the logic block works as a data memory with two storage sections. The write address provided on the inputs $in_1 \dots in_4$ of the logic block (*address* 1)

[9] We assume that the rising edge of the clock signal *clk* activates the logic block operation.

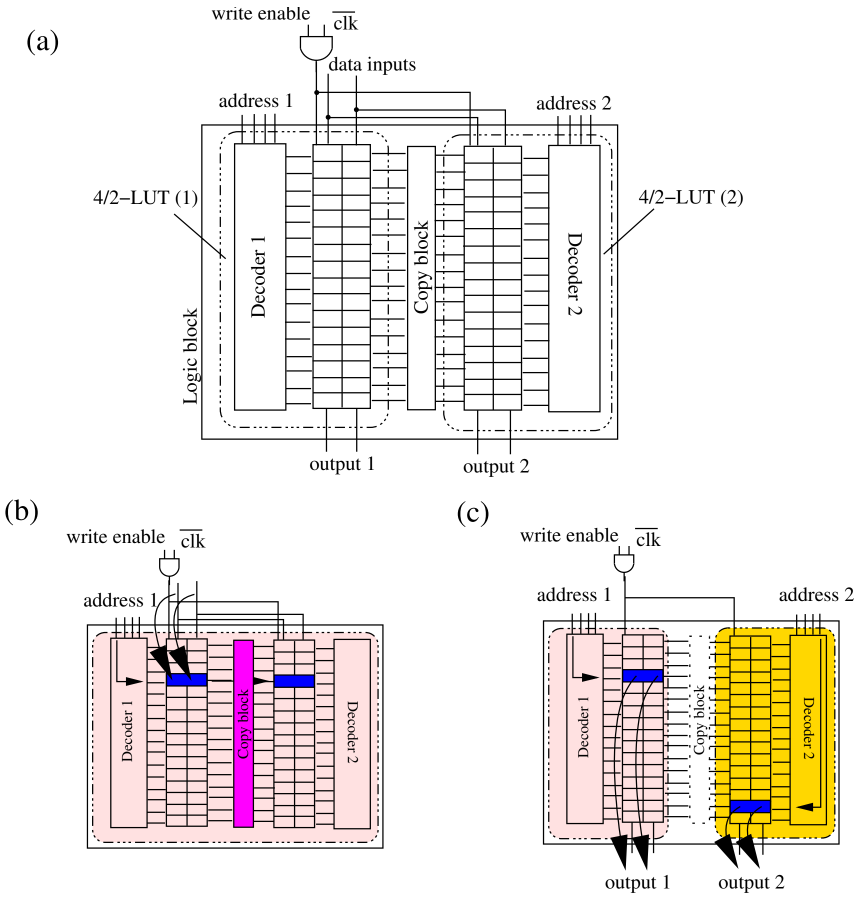


Figure 6.16. The implementation of a dual-port memory in the proposed logic block: (a) general structure of the memory, (b) read operation, (c) write operation.

selects two memory locations (memory cells) in the 4/2-LUT of the first logic element. Because of the activated copy block, the data which are being written to these locations are immediately copied at the same address to the 4/2-LUT in the second logic element. During the read operation (see Figure 6.16(c)), the copy block is disactivated by the write enable signal being '0' (the first phase of the $\overline{clk}$), and the memories implemented in the logic elements of the logic block work independently. The first logic element reads out the values from the same address that has been used during the write operation. The second logic element reads out the values from the second read address (*address* 2), which is available on the inputs $in_7 \ldots in_{10}$ of the logic block.

Shift register. The logic block enables the implementation of shift registers (delay lines) of various lengths (delays) and with various bit-widths of the input. The following shift register configurations are possible in the logic block:

- one 1-bit shift register of the maximal length 64,
- one 2-bit shift register of the maximal length 32,

- two 1-bit shift registers of the maximal length 32,
- two 2-bit shift registers of the maximal length 16.

The shift register implementation (see an example in Figure 6.14(c)) is identical to the implementation of a single-port data memory, except for the read and write operations that take place in the same clock cycle according to the read-before-write scheme (as in the dual-port memory implementation). Consequently, the write enable inputs we_1, we_2 of the 4/2-LUTs receive the inverted clock signal $\overline{clk}$, and the external write enable signal, which is connected to one of the secondary inputs t_1, t_2 of the logic block, works as a control signal.

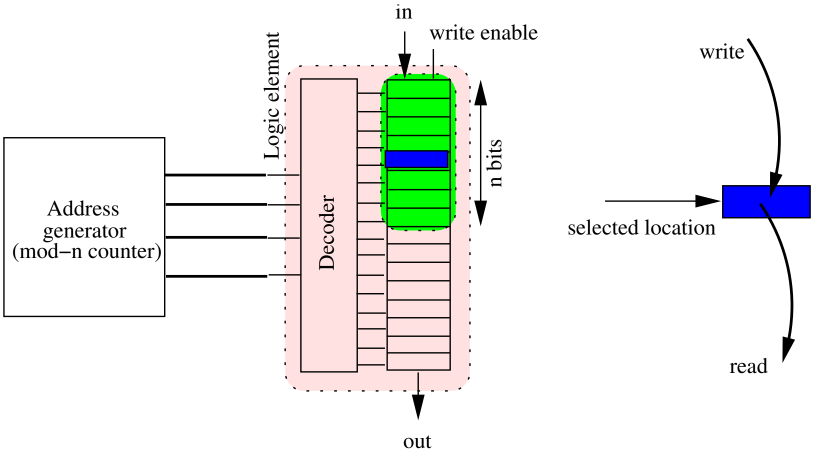


Figure 6.17. The implementation of a shift register generating a delay of n clock cycles (for the sake of simplicity, only one memory column of the 4/2-LUT is shown).

In Figure 6.17, the implementation of a shift register of an arbitrary length $n \leq 16$, is explained. The memory column of the 4/2-LUT is addressed by the address signal coming from the modulo-n counter implemented in an additional logic block. During the period of n clock cycles, n successive locations in the 4/2-LUT memory column memory are selected one after the other. At the first half of each clock cycle, the value from the selected memory location is directed to the output *out*; at the second half of the clock cycle, a new value available at the data input *in* is stored at the same location. After n clock cycles, the first n memory locations are updated. The modulo-n counter again selects the first memory location and the described procedure repeats. In this way, a given memory location is always accessible every n clock cycles. If shift registers with a delay longer than 16 cycles are to be implemented, other memory columns are utilised. The reading from and writing to the additional columns are determined by the value of the most significant bits of the address.

6.6 Configuration architecture

The memory-oriented reconfigurable logic architecture has been designed to support partial and a restricted form of dynamic reconfiguration. The partial reconfiguration means that the functionality of the device can be changed selectively *without* re-loading the entire content of the configuration memory. Because such a reconfiguration procedure may take place during the operation of the device, we also regarded it as dynamic reconfiguration. The dynamic reconfiguration is restricted since only the part of the device that is currently not executing any task can be reconfigured.

The configuration architecture is hierarchical. On the first level, the tile that is to be configured is selected, while on the second level, a set of bits within the tile, the state of which is to be changed, is chosen. The configuration architecture of a 2 x 2 array of logic tiles is illustrated in Figure 6.18(a). The vertical and horizontal *global decoders* generate horizontal and vertical address signals $h_0 \ldots h_3$ and $v_0 \ldots v_3$ based on the 2-bit address vectors H and V, respectively. The logic tile at the location (x,y) of the array receives a pair of global address signals (h_x, v_y). Each tile of the array receives also a set of 18 configuration signals, 14 of which are distributed to the configuration memories of both logic elements. The *local configuration memory* of a single logic element is shown in Figure 6.18(c). Such a memory consists of the control and LUT memories. The *control memory* configures switches and multiplexers of the logic and routing resources of the logic element, while the *LUT memory* determines the type of function which is implemented in the 4/2-LUT. The bits of both the control and LUT memories are organised in columns of 16 memory cells each. The number of memory columns determines the minimal number of bits than can be changed at once (here: eight). The memory columns are addressed by the signals from a local 4:16 decoder. Such a decoder is part of the 4/2-LUT (see Section 6.3), and is used normally to address the LUT memory bits. The sharing of the decoder logic contributes to the reduction of the logic block area. The set of 16 switches is placed between the control memory and the local decoder to disconnect them if the configuration procedure is not performed. The memory cells of the control and LUT memories have different implementations (see details in Chapter 8).

The set of configuration signals associated with each logic block (logic tile) includes:

- two independent sets of two configuration enable signals e_{cm1}, e_{lm1} and e_{cm2}, e_{lm2} for the control and LUT memories of the first and second logic element, respectively,
- four local decoder signals $dec_0 \ldots dec_3$,
- eight configuration data signals $cd_0 \ldots cd_7$ for the control memory,

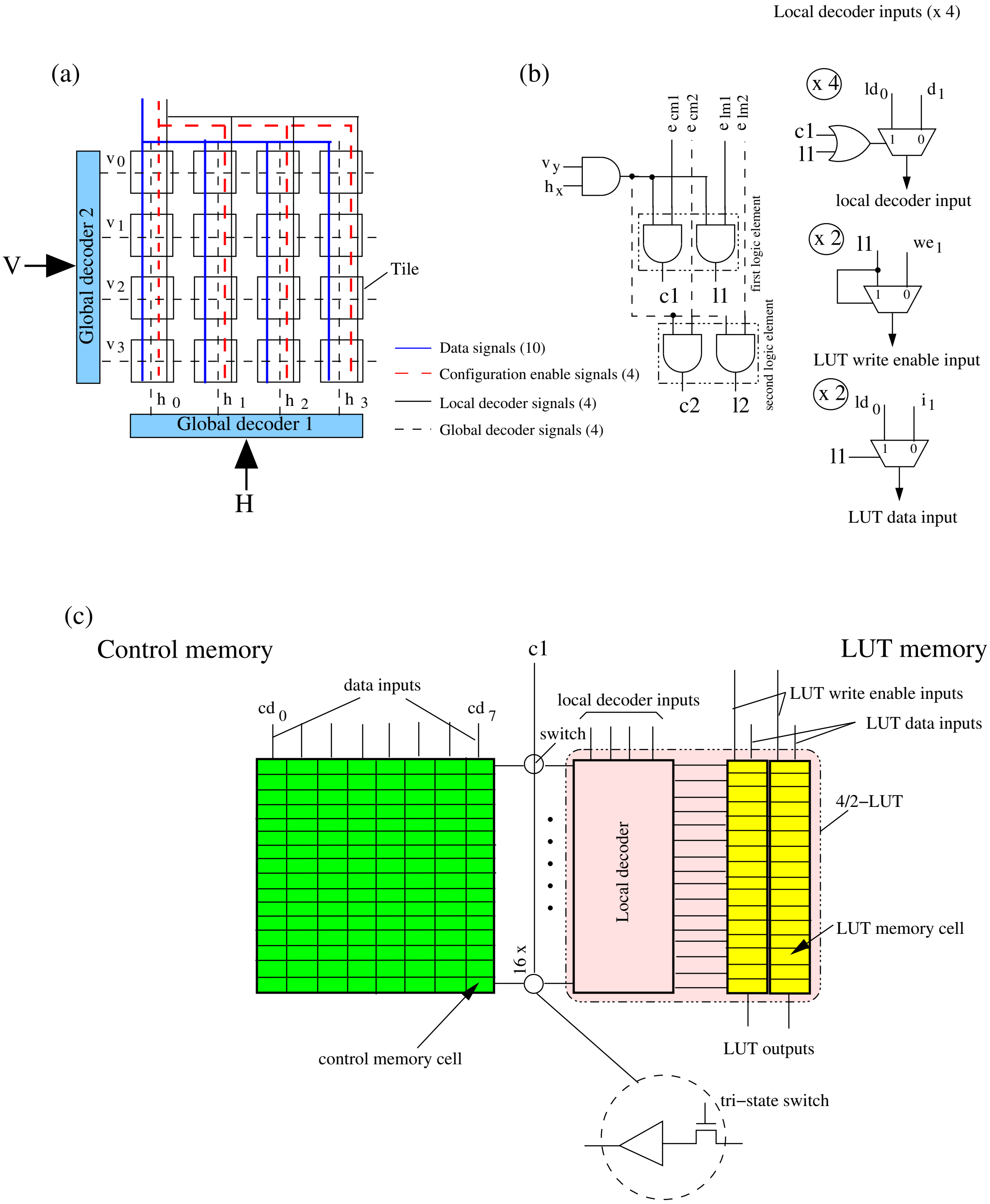


Figure 6.18. Configuration architecture of the memory-oriented reconfigurable logic device: (a) the first level with global decoders, (b) connection of the configuration signals to the physical ports of the first 4/2-LUT, (c) second level with a local controller of the first logic element.

- two configuration data signals ld_0, ld_1 for the LUT memory.

The connection of such signals to the physical ports of the 4/2-LUT is shown in Figure 6.18(b). When a logic tile is selected (i.e. when $h_x = 1$ and $v_y = 1$), the signals $c1$ and $l1$ that are generated based on the values of the enable signals e_{cm1} and e_{lm1} of the first logic element determine which of the configuration memories is to be configured. This is implemented by connecting the signals $c1$ and $l1$ to the control inputs of 2:1 multiplexers which select the inputs of the 4/2-LUT in the first logic element. For example, if both the control and the LUT memories are to be configured, the switches between the control memory and the local decoder are activated, the inputs of the local decoder are connected to the signals $dec_0 \dots dec_3$ signals, and write enable and data input ports of the first 4/2-LUT are connected to the $l1$ and ld_0, ld_1 signals, respectively.

6.7 Interconnect

The interconnect architecture of the memory-oriented logic block has been established by experience. The routing channels consisting of 30 routing tracks and three different types of the routing resources have been chosen. They are:

- general-purpose routing resources,
- special routing resources,
- direct routing resources.

The *general-purpose routing resources* are used for the routing of general-purpose signals. This type of resources includes 12 tracks of the length-one (L1) and 16 tracks of the length-four (L4). The routing tracks of the length-four can also be used for the implementation of 4-bit buses between the mapped data-path components. Such buses are implemented in a twisted-like way (see Figure 6.19(b)), which greatly simplifies their physical implementation (see details in Chapter 7). The general-purpose routing resources are connected to the input and output ports of the logic block using connection blocks which the flexibility $F_{ci} = 0.4$ and $F_{co} = 0.5$, respectively. Furthermore, the switch block with six switches at the crossing of each pair of horizontal and vertical routing tracks (the full switch block) is used.

The *special routing resources* are formed by two vertical and horizontal routing tracks. These tracks are meant only for the routing of the carry signals of arithmetic components and for the write enable signals of memory structures. Finally, the *direct routing resources* include direct connections between neighbouring logic blocks enabling their fast communication. Each logic block distributes four output signals and two carry output signals in this way. The direct connections are distributed in all directions, that is, horizontally, vertically, and diagonally. The schematic diagram representing the proposed interconnect structure is shown in Figure 4.10.

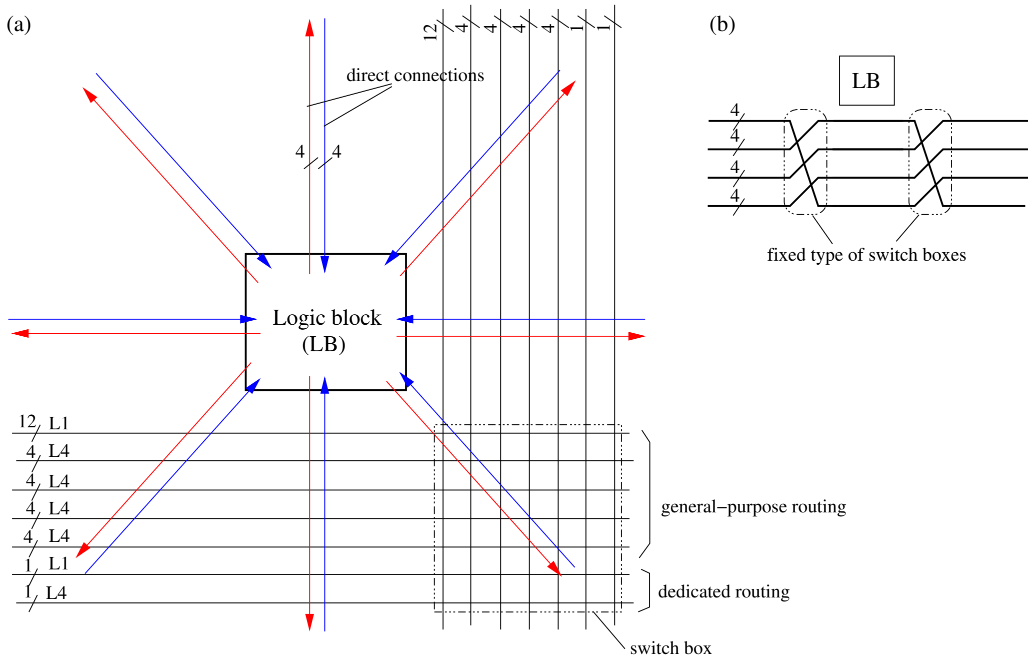


Figure 6.19. The interconnect architecture of the memory-oriented reconfigurable logic device: (a) overall routing resources, (b) implementation of the length-four routing tracks using a twisting mechanism (only a horizontal routing channel is shown).

6.8 Benchmarking

Similarly to the random-logic-oriented architecture, the memory-oriented reconfigurable architecture was evaluated using only the model-based cost metrics (see Section 3.2.4).

6.8.1 Benchmarking using the model-based cost metrics

The benchmarking using the model-based cost metrics was realised in an analogous way to the benchmarking of other domain-oriented reconfigurable architectures (see Section 4.8.2 and Section 5.6.1). In Table 6.2, the architecture-specific parameters N_{lmb} and P_w are listed. Such parameters were used to model the cost of the memory-oriented reconfigurable architecture during the benchmarking procedure.

The parameters N_{lmb} and P_w from Tables 3.3 and 6.2 were used to calculate the mapping cost with respect to logic MC_L (see Equation 3.23) and the mapping cost

FPGA architecture	Number of pins					N_{lmb}	P_w
	Inputs	Outputs	Carry input	Carry output	Auxiliary		
Memory-oriented	12	4	0	2	2	64	18

Table 6.2. Characterisation of the logic cost (via N_{lmb}) and the routing resource cost (via P_w) in the proposed memory-oriented FPGA architecture.

with respect to routing MC_R (see Equation 3.24) for the proposed memory-oriented FPGA and three commercial FPGA devices. The results of such a comparison are shown in Table 6.4. For each FPGA, the table lists the value of the parameter N_{LB} and the mapping cost components MC_L and MC_R.

The data from Table 6.4 are also shown graphically in Figures 6.20–6.30. Table 6.3 summarises the results from Table 6.4 by showing the ratio of mapping costs MC_L and MC_R assuming the memory-oriented FPGA as a reference. Note, that results from Figure 6.27 should be interpreted remembering that the logic blocks of the Altera and Atmel FPGA devices do not allow memory mapping.

Commercial →	**Xilinx Virtex II**		**Altera Stratix**		**Atmel AT40K**	
Proposed ↓	MC_L	MC_R	MC_L	MC_R	MC_L	MC_R
MEM-oriented	av=0.94	av=1.62	av=2.65	av=3.64	av=2.59	av=3.46
	mn=0.5	mn=0.89	mn=0.5	mn=0.67	mn=0.5	mn=0.67
	mx= 1.01	mx=1.78	mx=16	mx=22	mx=16	mx=21.33

Table 6.3. The summary of the results from Table 6.4. The mapping costs w.r.t. logic MC_L and w.r.t. routing MC_R of three commercial FPGAs are normalised with respect to the mapping cost of the memory-oriented architecture. The average cost ratio *av* for 11 benchmarks of Xilinx and 10 for Altera and Atmel, the minimum *mn* and maximum *mx* cost ratios are given.

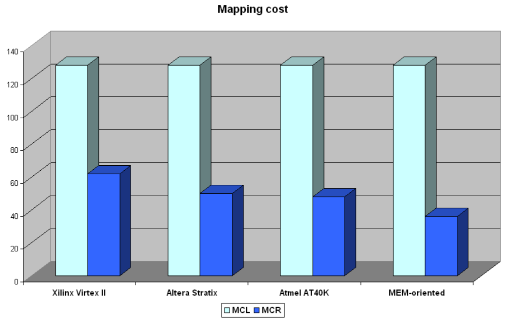


Figure 6.20. 8-bit adder.

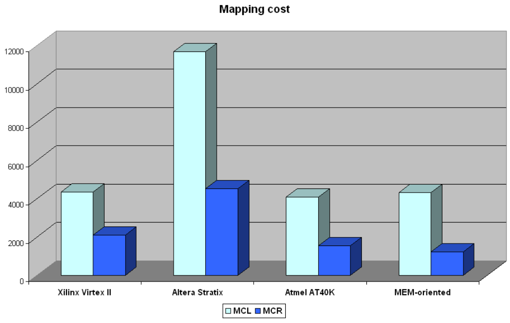


Figure 6.21. 16-bit multiplier.

	State-of-the-art commercial FPGA architectures									Proposed FPGA		
	Xilinx Virtex II			Altera Stratix			Atmel AT40K			Memory-oriented		
Benchmark function	N_{LB}	Mapping cost		N_{LB}	Mapping cost		N_{LB}	Mapping cost		N_{LB}	Mapping cost	
	–	MC_L	MC_R	–	MC_L	MC_R	–	MC_L	MC_R	–	MC_L	MC_R
8-bit ADD	1	128	62	0.8	128	50	8	128	48	2	128	36
16×16 MULT	34	4.25K	2.11k	73	11.41K	4.53k	256	4K	1.54k	67.5	4.22K	1.22k
2:1 MUX/4-bit	0.5	64	31	0.4	64	25	4	64	24	1	64	18
8:1 MUX/1-bit	0.5	64	31	0.5	80	31	7	112	42	1	64	18
16:1 MUX/1-bit	1	128	62	1	160	62	15	240	90	2.5	160	45
2-in OR/4-bit	0.5	64	31	0.4	64	25	4	64	24	1	64	18
3-in NOR/1-bit	0.125	16	8	0.1	16	6	1	16	6	0.5	32	9
16-in AND/1-bit	0.5	64	31	0.4	64	25	5	80	30	1	64	18
4:16 DECOD	2	256	124	1.6	256	99	16	256	96	4	256	72
16-long 2-bit SREG	0.25	32	16	3.2	512	198	32	512	192	0.5	32	9
16×4-bit RAM	1	64	31	–	–	–	–	–	–	1	64	18

Table 6.4. The mapping cost comparison between the state-of-the-art and proposed memory-oriented FPGA architectures.

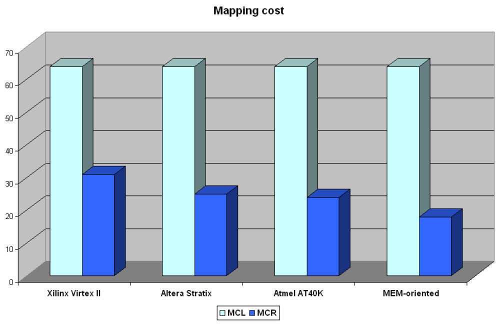


Figure 6.22. 2:1 4-bit multiplexer.

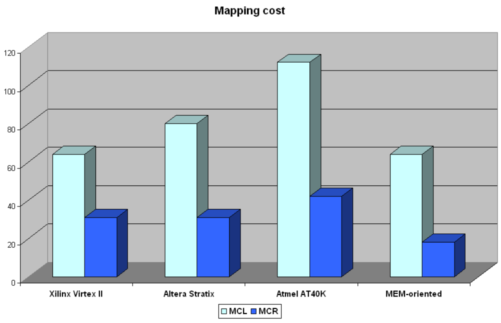


Figure 6.23. 8:1 1-bit multiplexer.

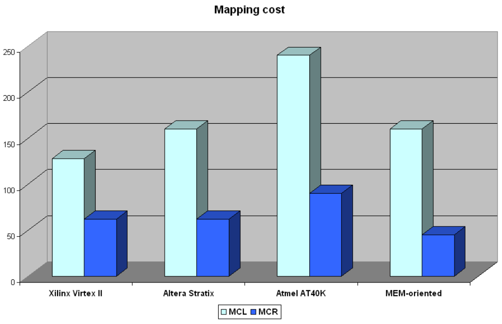


Figure 6.24. 16:1 1-bit multiplexer.

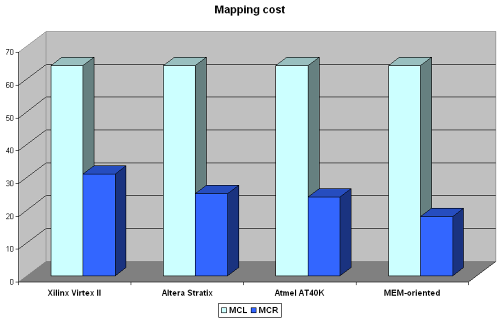


Figure 6.25. 2-input 4-bit OR.

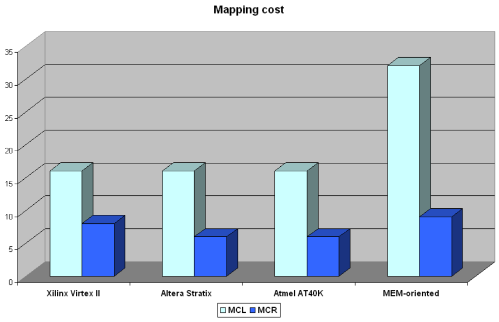


Figure 6.26. 3-input 1-bit NOR.

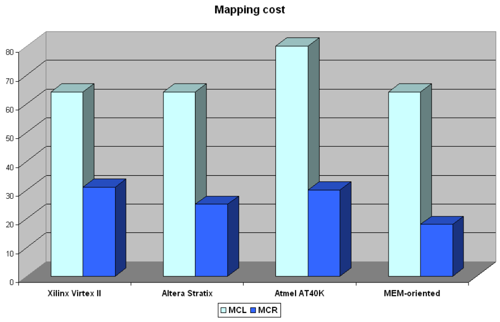


Figure 6.27. 16-input 1-bit AND.

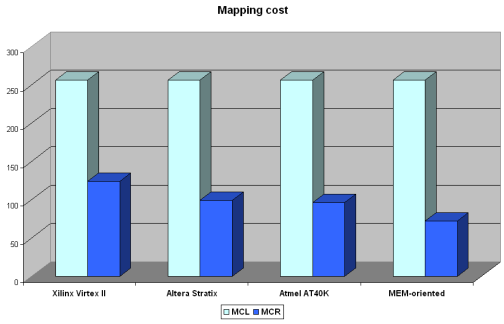


Figure 6.28. 4:16 decoder.

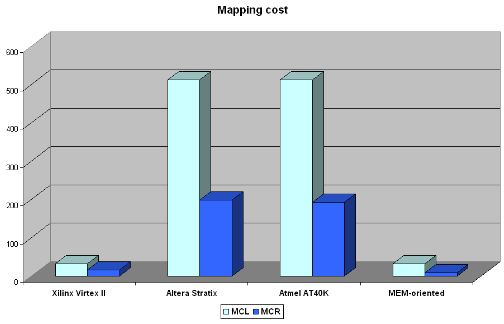


Figure 6.29. 16-long 2-bit shift register.

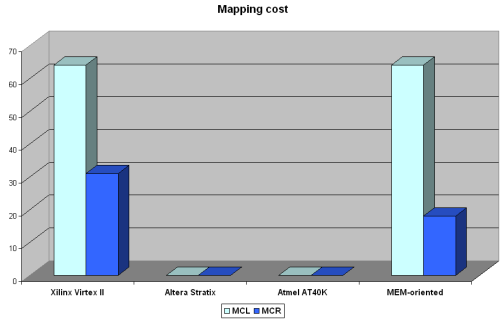


Figure 6.30. 16 x 4-bit memory.

6.8.2 Discussion

The results of benchmarking the memory-oriented FPGA, which are summarised in Table 6.3, show that the proposed FPGA architecture is superior to the compared commercial FPGAs. Unlike the random-logic-oriented architecture, the memory-oriented architecture offers the reduction of both logic and routing costs.

The cost reduction factor depends on the compared FPGA, and is higher for Altera and Atmel devices. This is because logic blocks of such devices do not have memory mapping capabilities (within logic blocks). Consequently, the logic cost of the Altera and Atmel devices is on average 2.65 and 2.59 times higher, respectively, than the logic cost of our FPGA. Furthermore, the average routing resource cost of the Altera and Atmel devices is 3.64 and 3.46 times higher, respectively, than for our FPGA. At the same time, the logic and routing resource costs of the Xilinx FPGA are on average 6% lower and 1.62 times higher, respectively, than similar costs of our FPGA.

It is interesting to note that even at a very similar logic cost, the proposed memory-oriented FPGA reduces the routing resource cost compared to the functionally equivalent Xilinx FPGA.

6.9 Conclusions

In this chapter, we discussed the memory-oriented reconfigurable architecture which targets designs with distributed storage. The memory functionality has been obtained by the use of a novel multi-output look-up table, such as a 4/2-LUT, as a basic building block of the reconfigurable logic block. An efficient implementation of arithmetic in such a LUT has been enabled by applying the inversion-based folding type II. This yields a factor of 2 reduction in the number of LUT configuration bits of the multi-output LUT compared to the number of bits required in a traditional implementation.

Because the 4/2-LUT also allows the random logic mapping, the memory-oriented reconfigurable logic architecture has a general-purpose character. However, in contrast to state-of-the-art general-purpose FPGAs, the implementation cost of the proposed architecture is substantially reduced. The reason is the reduction of the amount of routing resources which is the result of sharing input ports of the multi-output LUT. The model-based comparison of our FPGA with the Xilinx Virtex II FPGA which also supports memory mapping confirms the routing resource reduction. The Xilinx FPGA is found to have on average 1.62 times higher routing resource cost than our FPGA.

Chapter 7

TEMPLATE-BASED METHODOLOGY FOR RECONFIGURABLE LOGIC DESIGN

In Chapters 4–6, we addressed the problem of cost-efficiency of embedded reconfigurable logic from the architecture perspective. We showed that the cost reduction (in terms of the silicon area that is required to implement a given function) can be achieved by tuning a reconfigurable logic architecture to a target application domain (or a set of processing kernels). However, the implementation of basic building blocks of a reconfigurable logic architecture, such as the logic tile we focused on, is only one of the aspects that must be considered while designing reconfigurable-logic-based systems. To address very different requirements of such systems, embedded reconfigurable logic cores must be available in different sizes and (unlike stand-alone FPGAs) in different shapes. Also, to further reduce the cost, there may be a need for several different variants of the same domain-oriented reconfigurable logic core.

Design time and effort are important aspects that must be considered. Next, the complexity and availability of mapping tools must be taken into account. Such tools should fully exploit the mapping capabilities of different types of reconfigurable logic cores. They should also ensure that structural and functional differences between the cores are properly reflected during the mapping procedure.

To address the above-mentioned aspects, we propose a *template-based* methodology for reconfigurable logic design. Such a methodology plays a central role in the process of application domain specialisation of reconfigurable logic described in Chapter 2. In this chapter, we discuss basic characteristics of the methodology and present a hierarchical template of a reconfigurable logic architecture. We also discuss the implications for the physical design and mapping process.

7.1 The concept

Our methodology is based on the use of a *template*. The template is a generic model of an architecture of an embedded reconfigurable logic core and is described by a set of parameters. By setting the template parameters according to the specific values, application-domain-specific instances of the template can be created.

Note, that this is analogous to the concept of Application-Specific Instruction set Processors (ASIPs), which play a key role in designing cost-efficient embedded systems [107]. Typically, ASIP cores target specific application domains and are also generated from a parametrised architecture template.

The proposed template serves the following purposes:

- The template is used to *generalise* the concept of application domain specialisation introduced in Chapter 2. The application-domain-oriented instances of the template, for example such as described in Chapters 4–6, are derived by setting the template parameters accordingly.
- By using a generic architecture template and allowing an arbitrary change of its parameters, many different architectural instances of the template can be created. This enables a systematic *architecture space exploration* with experiments on a much larger set of potentially interesting solutions than would be possible using conventional (manual) methods. In this way, a fast evaluation of the domain-specific instances of the template is possible.
- The template is used to reduce *design time and effort*. For example, a VLSI implementation of different types of logic cores (template instances) can be considerably reduced if their netlists and layouts are generated automatically from a template.
- Assuming that the parametrised architecture template is also used as a model in mapping (CAD) tools (e.g. technology mapping, placement, routing), such tools can be made *retargetable*.

The idea of a parametrised reconfigurable logic architecture is not entirely new. In [14], Betz *et al.* use a parametrised description to model different variants of FPGA architectures for the purpose of a flexible CAD tool-set. Such a tool-set, including a placement and routing tool called VPR (Versatile Placement and Routing) and a packing (clustering) tool called T-VPack (Timing-driven Packing for VPR), is meant as the back-end of the mapping flow targeting an arbitrary LUT-based FPGA architecture[1]. The details of Betz's architecture model and the description of the automatisation of the architecture generation process from a high level description can be found in [13]. In [29], the concept of domain-specific reconfigurable subsystems, which forms the basis of the Totem project, is described. Reconfigurable subsystems are generated automatically based on the requirements of a given group of applications. The resulting architecture is a reconfigurable data-path consisting of a computational structure and a routing fabric.

There are several aspects that differentiate our concept from the prior art. Firstly, the primary purpose of Betz's model is the introduction of flexibility to the FPGA

[1] The architecture model used by Betz introduces some limitations, because of which only relatively simple FPGA structures can be modelled.

placement and routing tool allowing the use of such a tool for different FPGA architectures. In consequence, the logic resources in Betz's model are modelled as black boxes of the specified granularity, and are described by only those parameters that are relevant for the proper functioning of the tool (e.g. for its timing engine). In contrast, our template defines the complete architecture of a reconfigurable logic device, that is, all functional blocks and the associated routing. Also, our template can be applied both to an application mapping flow and to a VLSI design flow. Secondly, Betz's model targets conventional general-purpose FPGA architectures. It assumes a simple *k*-input LUT as a basic logic element of such architectures; the LUTs can be clustered together to form a coarser logic block. Again, this is in contrast to our template, which is meant for the modelling of application-domain-oriented architectures. The values of the template parameters depend thus on a target application domain. In addition, logic elements in our model may have a different structure and a higher complexity than a single *k*-LUT that is assumed in T-VPack and VPR. Finally, Betz's architecture model is based on four levels of hierarchy, while our architecture template features five levels. The additional level of hierarchy in our model allows an unambiguous description of functionally different domain-oriented reconfigurable logic structures.

Our reconfigurable architecture template also differs from the concept proposed in the Totem project. The Totem concept does not rely on a parametrised architecture template: the generated architectures are assembled from a library of basic components, such as multipliers, adders, memories. Furthermore, the Totem architectures fall into a category of coarse-grain architectures and not of fine-grain architectures, the category we aim at (see a discussion in Section 1.4.1). Also, they are built as one-dimensional structures rather than two-dimensional structures. Finally, the Totem domain-specific reconfigurable subsystems provide much more restricted flexibility than the domain-specific instances generated from our reconfigurable logic architecture template.

7.2 The reconfigurable logic architecture template

The reconfigurable logic architecture template defines the way of generating the *complete* architecture of an *arbitrary* type of an *application-domain-oriented* reconfigurable logic core using a *limited number* of basic building blocks called *tiles*. As in Chapter 3, we assume that the architecture of the generated core is *homogeneous*. The proposed template is hierarchical. Five levels of hierarchy, each of which is described by a unique set of parameters, include (in a bottom-up order): a logic element, a processing element, a logic block, a tile, and an array. Below, we define each level of hierarchy and explain the rationale behind it.

7.2.1 Level I – Logic element

Logic element (le) is a basic LUT-based functional component of a reconfigurable logic architecture.

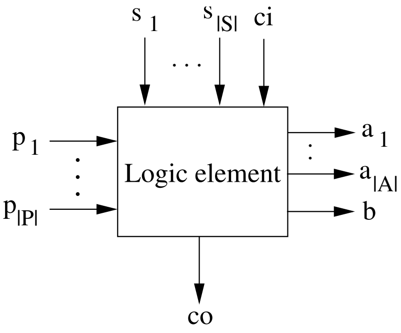


Figure 7.1. Logic element.

The logic element has the set $P = \{p_i : i \in \mathbb{N}\}^2$ of primary input ports, the set $S = \{s_i : i \in \mathbb{N}\}$ of secondary input ports, and a carry input port *ci*. It also has the set $A = \{a_i : i \in \mathbb{N}\}$ of arithmetic output ports, a boolean output port *b*, and a carry output port *co*. This is illustrated in Figure 7.1. The number of ports of the logic element and its functionality are determined by the type $\mathcal{T}$ of the logic element. The type $\mathcal{T}$ depends on a target application domain.

The three types of the logic elements that have been proposed in Chapters 4–6 can be modelled as shown in Figure 7.2. The number of ports and functionality of the logic elements are given in Table 7.1 and Table 7.2, respectively. The functionality is determined by the largest Boolean, arithmetic and memory functions that can be implemented in the logic element. Consequently, the number of bits of the input vector of the largest Boolean function, the number of bits of the argument of the arithmetic function, and the number of bits of the data input of a memory define the functionality of the logic elements.

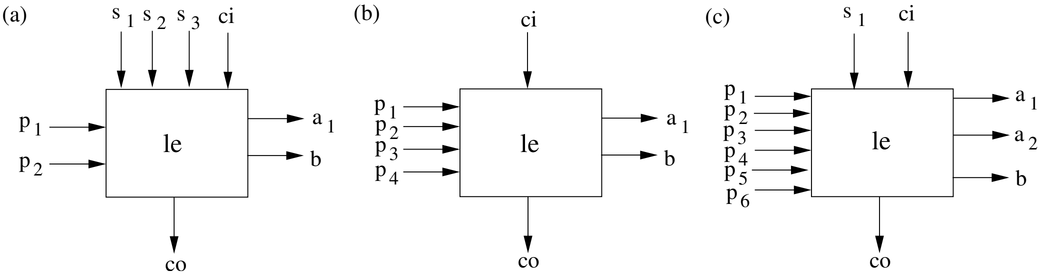


Figure 7.2. Application-domain-oriented instances of the logic element that have been proposed in Chapters 4–6: (a) data-path-oriented *le*, (b) random-logic-oriented *le*, (c) memory-oriented *le*.

[2] We use $\mathbb{N}$ to denote the set of natural numbers excluding 0.

Type $\mathcal{T}$ of logic element	$\|P\|$	$\|S\|$	$\|A\|$
Data-path-oriented	2	3	1
Random-logic-oriented	4	0	1
Memory-oriented	6	1	2

Table 7.1. Number of ports of the logic element dependent on its type $\mathcal{T}$. (It is assumed, that all logic elements have a *single* boolean output b, which, for the simplicity's sake, is not listed in the table.)

Type $\mathcal{T}$ of logic element	Boolean	Arithmetic	Memory
Data-path-oriented	2	1	–
Random-logic-oriented	4	1	–
Memory-oriented	5	2	2

Table 7.2. Functionality of the logic element dependent on its type $\mathcal{T}$.

7.2.2 Level II – Processing element

Processing element (pe) consists of the set $N = \{le_i : i = 2^j, j \in \mathbb{N}_0\}$[3] of logic elements connected in parallel. The number $|N|$ of logic elements is chosen such that the implementation of large Boolean functions according to Shannon's expansion (see Equation 3.2) is possible.

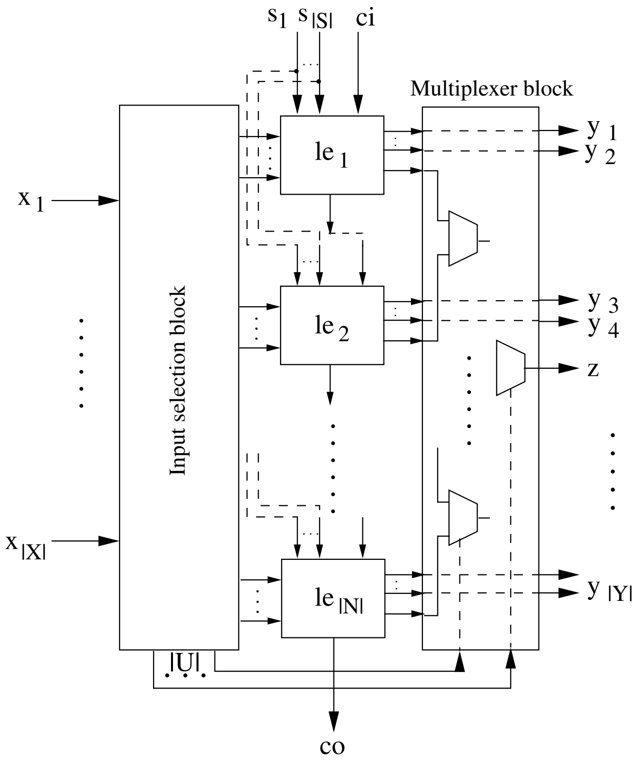


Figure 7.3. Processing element.

[3] $\mathbb{N}_0$ denotes the set of natural number including 0.

The processing element, which is shown in Figure 7.3, has the set $X = \{x_i : i \in \mathbb{N}\}$ of primary input ports, the set $S = \{s_i : i \in \mathbb{N}\}$ of secondary input ports, and a carry input port *ci*. It also has the set $Y = \{y_i : i \in \mathbb{N}\}$ of arithmetic output ports, a boolean output port *z*, and a carry output port *co*. The input ports x_i of the processing element are connected via the *input selection block* to the primary input ports p_i of $|N|$ successive logic elements. The input selection block, which comprises a set of multiplexers (e.g. in the configuration as shown in Figure 4.7(a)), guarantees that, dependent on the functional mode of the processing element, the primary input ports of the logic elements always receive the right set of signals from the primary input ports of the processing element. The number $|X|$ of the primary input ports is equal to the cumulative number of 1-bit inputs of the largest Boolean, arithmetic or memory functions, whichever is greater, that can be implemented in the processing element. The $|S|$ secondary input ports s_i of the processing element are connected directly to the secondary input ports s_i of all logic elements. In contrast, the carry input ports *ci* and carry output ports *co* of logic elements are chained. This means that all logic elements except the first one have their carry input ports *ci* connected to the carry output port *co* of the preceeding logic element. The first logic element of the processing element, that is le_1, has its carry input port *ci* connected to the carry input port *ci* of the processing element (or it may be shorted with one of the secondary input ports if the selected secondary input port and the carry input port of the processing element are used in a mutually exclusive way; see examples in Section 7.3). Similarly, the last logic element of the processing element, that is $le_{|N|}$, has its carry output port *co* connected to the carry output port *co* of the processing element. The arithmetic output ports of the logic elements are connected directly with the $|Y|$ output ports of the processing element. The boolean output ports *b* of the logic elements are multiplexed in the *multiplexer block* consisting of a $log|N|$- level network of 2:1 multiplexers. The multiplexers are controlled by the set $U = \{u_i : i \in \mathbb{N}\}$ of the control signals that are generated in the input selection block. The output of the multiplexer block, which is the output of the final 2:1 multiplexer in this block, connects to the boolean output *z* of the processing element.

The number of input and output ports and the functionality of the processing element as a function of its type are described in Tables 7.3 and 7.4, respectively. The functionality of the processing element is defined as in Section 7.2.1.

Processing element type	$\mathbf{\|X\|}$	$\mathbf{\|S\|}$	$\mathbf{\|Y\|}$
Data-path-oriented	$2 \cdot \|N\|$	3	$\|N\|$
Random-logic-oriented	$\max(log\|N\| + 4, 2 \cdot \|N\|)$	0	$\|N\|$
Memory-oriented	$6 \cdot \|N\|$	1	$2 \cdot \|N\|$

Table 7.3. The number of ports in the processing element dependent on its type. (As in Section 7.2.1, it is assumed that all processing elements also have a *single* boolean output *z*.)

Processing element type	**Boolean**	**Arithmetic**	**Memory**
Data-path-oriented	$log\|N\|+2$	$\|N\|$	–
Random-logic-oriented	$log\|N\|+4$	$\|N\|$	–
Memory-oriented	$log\|N\|+5$	$2\cdot\|N\|$	$2\cdot\|N\|$

Table 7.4. Functionality of the processing element dependent on its type.

7.2.3 Level III – Logic block

Logic block (lb) consists of the set $M = \{pe_i : i = 2^j, j \in \mathbb{N}_0\}$ of processing elements, which are organised in $|K|$ parallel clusters. Similarly to the number of logic elements in the processing element, the number of processing elements in the logic block is implied by the LUT-based implementation of Shannon's expansion (see Equation 3.2).

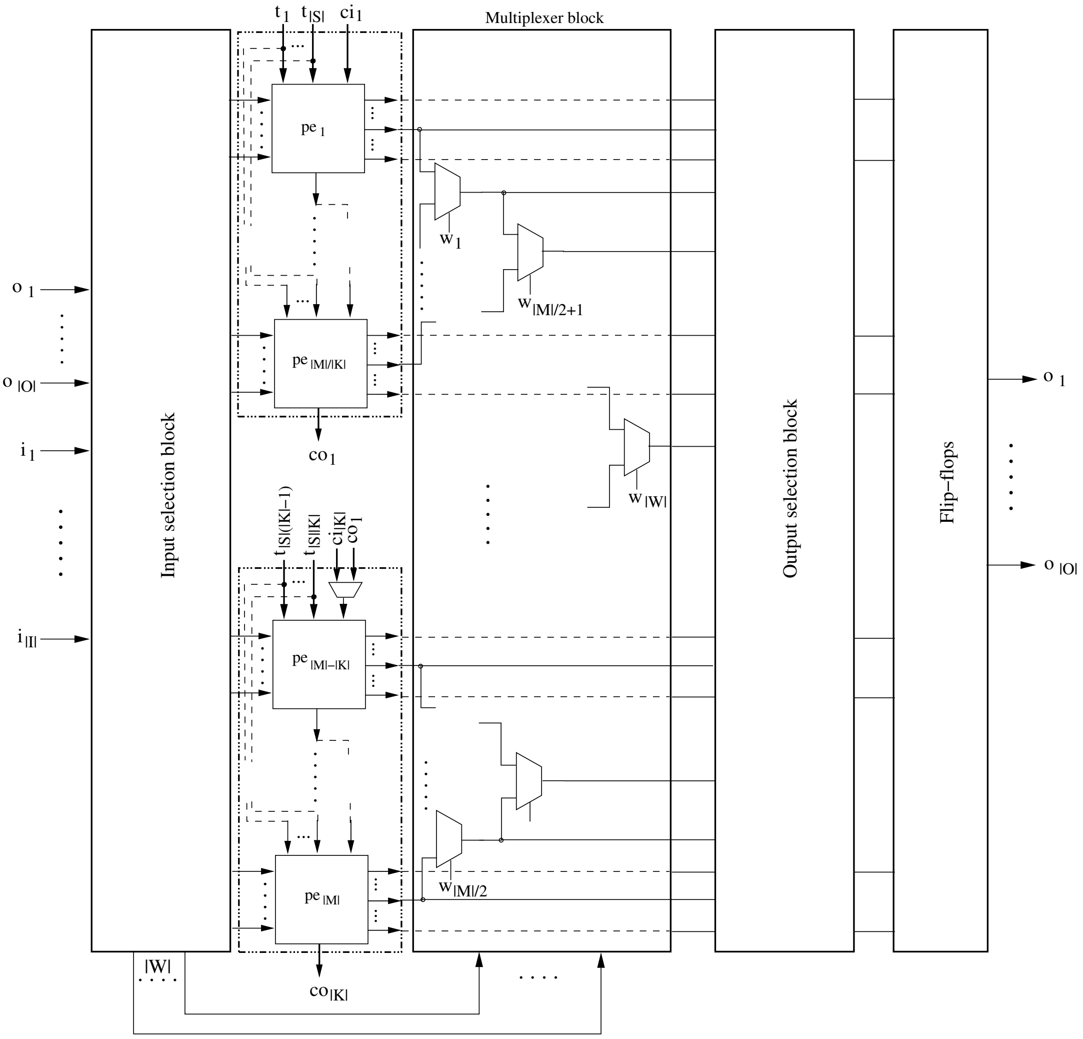


Figure 7.4. Logic block.

The logic block structure is shown in Figure 7.4. Each cluster of the logic block is characterised by an independent set of secondary input ports and independent carry input and output ports. The outputs of the logic block can be registered. The outputs can also be connected to the inputs of the logic block allowing the realisation of more complex logic functions or functions with feedback loops.

The logic block has the set $I = \{i_i : i \in \mathbb{N}\}$ of primary input ports, and $|O|$ feedback ports that are connected to the ports in the output port set $O = \{o_i : i \in \mathbb{N}\}$ of the logic block. The logic block also has the set $T = \{t_i : i \in \mathbb{N}\}$ of secondary input ports such that $|T| = |S| \cdot |K|$. The first $|S|$ inputs of the set T, that is $t_1, \ldots, t_{|S|}$, belong to the first cluster of processing elements, the second $|S|$ inputs of the set T, that is $t_{|S|+1}, \ldots, t_{2\cdot|S|}$, belong to the second cluster of processing elements, etc. The logic block has also $|K|$ carry input ports ci_i and $|K|$ carry output ports co_i, with i being the cluster index such that $0 < i \leq |K|$.

The $|I|$ primary inputs and $|O|$ feedback inputs are fed to the *input selection block* consisting of a set of multiplexers. The input selection block of the logic block serves two purposes. Firstly, it implements the full connectivity between the primary input ports of the logic block and the primary input ports of processing elements. Secondly, it implements feedback between the primary output ports and the primary input ports of the logic block.

Full connectivity means that a signal from each primary input port of the logic block is distributed to the primary input ports of all processing elements. This type of connectivity is introduced if the number of primary input ports of the logic block is lower than the number of primary input ports of processing elements in all clusters, that is, when $|I| < |M| \cdot |X|$. The number of input ports of the logic block is lowered to reduce the routing cost (see Equation 3.22). Full connectivity helps in such a situation to increase the routability of mapped circuits. The minimum number of inputs $|I|$ of the logic block consisting of $|M|$ $|X|$-input LUTs can be calculated based on an empirical formula from [3], which is given by Equation 7.1[4].

$$|I| = \frac{|X|}{2} \cdot (|M| + 1) \tag{7.1}$$

While Equation 7.1 addresses requirements of Boolean functions, it does not take into account the input port requirements of large multiplexers. Because multiplexers are important implementation components (see Section 2.3.1), and because the number of inputs (data and control inputs) of a large multiplexer is higher than the number of inputs of a typical LUT-mapped Boolean function, we derive a new limit on the number of primary input ports of the logic block. Such a limit can be calculated as

$$|I| = |M| \cdot Z + logZ + log|M| - 1. \tag{7.2}$$

[4] The formula makes use of the fact that most Boolean functions share some of their inputs.

Z denotes the number of data inputs of a multiplexer that can be implemented in a single processing element of the logic block, and is derived using Equation 7.3.

$$Z = \begin{cases} 2 & \text{if } |N| = 1 \\ |N| & \text{if } |N| \geq 2 \end{cases} \tag{7.3}$$

Note, the number of primary input ports of the logic block given by Equation 7.2 relaxes slightly the routing constrains of the logic block (due to more routing resources) compared to the constrains implied by Equation 7.1.

As already stated, the input selection block also allows the realisation of feedback if signals from the set O of the output (feedback) ports of the logic block are selected as inputs of the processing elements. Dependent on a target application domain, the input selection block of the logic block can be designed with the one-to-one type or the full type of feedback connections. The *one-to-one feedback connections* are illustrated in Figure 4.7(a). Such connections are typical for the data-path-oriented architectures and allow the realisation of sequential arithmetic modules (such as counters, incrementers and decrementers), in which one of the inputs receives the registered signal from the output. The one-to-one feedback connections connect the $|O|$ output ports of the logic block to the $|O|$ primary input ports of the processing elements. The *full feedback connections* are shown in Figure 4.7(b). Such connections are typical for the random-logic-oriented architectures and allow the implementation of complex Boolean functions (the feedback signals are unregistered then) or different types of finite state machines (the feedback signals are registered then). The full feedback connections connect all $|O|$ output ports of the logic block to all $|M| \cdot |X|$ primary input ports of the processing elements. The input selection blocks with the one-to-one and full feedback connections are also shown in Figure 4.7(a) (the first set of multiplexers only) and Figure 5.3(a), respectively.

The outputs of the input selection block are connected to the primary input ports of the processing elements. The $|S|$ secondary input ports of the processing elements in the j-th cluster receive signals from the j-th set of the secondary input ports of the logic block, that is, from ports $t_{(j-1)\cdot|S|+1}, \ldots, t_{j\cdot|S|}$. Each first processing element in the j-th cluster of processing elements receives a carry input signal from the j-th carry input port ci_j of the logic block. Similarly, each last processing element of the j-th cluster generates a signal that is directed to the j-th carry output co_j of the logic block. All processing elements in the cluster have their carry input ports and carry output ports connected serially. In addition, the independent clusters of processing elements can also be connected serially. This is enabled by the 2:1 multiplexer at the carry input port of each first processing element in each j-th cluster (except the first cluster). The multiplexer in the j-th cluster selects between a signal from the carry input port ci_j of the logic block and the signal from the carry output port co of the $(j-1)$-th cluster.

The *multiplexer block* of the logic block is an $log|M|$-stage network of 2:1 multiplexers that are controlled by the control signals from the set $W = \{w_i : i \in \mathbb{N}\}$

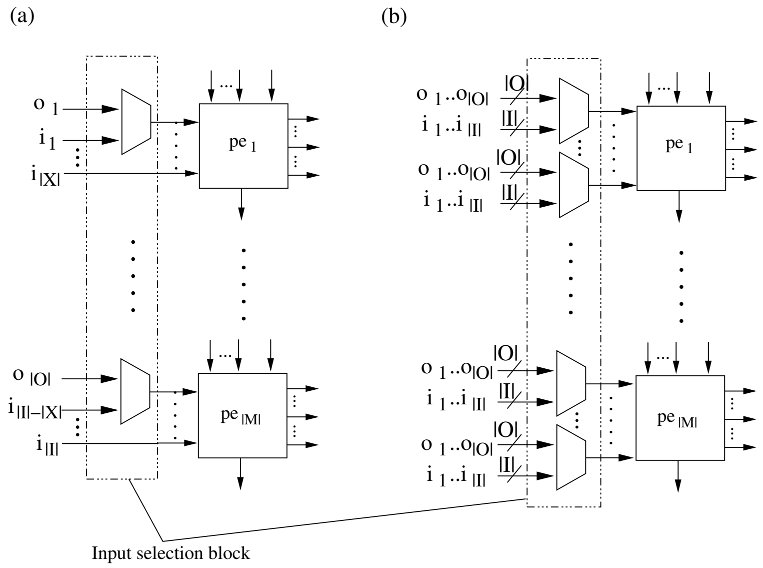


Figure 7.5. Comparison of the input selection blocks with (a) one-to-one feedback connections and (b) full feedback connections. In Figure (a) it is assumed that the first primary inputs of all processing elements are associated with successive bits of the first arithmetic argument (e.g. in the $x - y$ operation, x is the first argument).

that are generated in the input selection stage. The multiplexers in the first stage of the multiplexer block select between signals from the boolean output ports z of successive pairs of processing elements. Each multiplexer of the second stage selects between a pair of output signals of multiplexers in the first stage; each multiplexer of the third stage selects between a pair of output signals of multiplexers in the second stage, etc., as shown in Figure 7.4. The output signals of multiplexers in *all* stages are the output ports of the multiplexer block. This is in contrast to the multiplexer block of the processing element, which has only one output port (connected to the output of the multiplexer in the last stage).

The signals from the output ports of the multiplexer block and signals from the $|Y|$ arithmetic output ports of all processing elements are connected to the inputs of the *output selection block*. The output selection block is a multiplexer network which determines the final number of output signals of the logic block and the ports on which these signals appear. It is assumed that all output signals of the multiplexer block and all $|Y|$ arithmetic signals of the processing elements can be chosen as logic block outputs. The example of the output selection blocks designed in this way are shown in Figures 4.7(b) and 5.3(b).

The signals from the output selection block are directed to the *flip-flop* block. The flip-flop block, which example implementation is shown in Figure 5.3(b), allows any output of the logic block to be registered. The output signals of the flip-flop block, registered or not, are directed to the $|O|$ output ports of the logic block.

Table 7.5 lists the number of primary input and output ports in the logic block and

Table 7.6 the functionality of the logic block dependent on the logic block type. The functionality of the logic block is defined as in Section 7.2.1.

Logic block type	I	O
Data-path-oriented	$\|X\| \cdot \|M\|$	$\|Y\| \cdot \|M\|$
Random-logic-oriented	$\max(\|M\| \cdot Z + logZ + log\|M\| - 1, \frac{\|X\|}{2} \cdot (\|M\| + 1)$	$\|Y\| \cdot \|M\|$
Memory-oriented	$\|X\| \cdot \|M\|$	$\|Y\| \cdot \|M\|$

Table 7.5. Number of ports in the logic block dependent on its type.

Logic block type	Boolean	Arithmetic	Memory
Data-path-oriented	$log\|M\| + log\|N\| + 2$	$\|M\| \cdot \|N\|$	–
Random-logic-oriented	$log\|M\| + log\|N\| + 4$	$\|M\| \cdot \|N\|$	–
Memory-oriented	$log\|M\| + log\|N\| + 5$	$2 \cdot \|M\| \cdot \|N\|$	$2 \cdot \|M\| \cdot \|N\|$

Table 7.6. Functionality of the logic block dependent on its type.

7.2.4 Level IV – Tiles

The level IV of our template describes four basic building blocks of a reconfigurable logic architecture, that is, a logic tile, an input/output tile with routing, an input/output tile, and a corner routing tile. Such tiles are necessary for the creation of a homogenous symmetrical reconfigurable logic architecture, which will be discussed in detail in Section 7.2.5.

Logic tile

Logic tile (lt) is the main building block of a reconfigurable logic architecture. It consists of a logic block and routing resources. The routing resources define the number of routing tracks in the horizontal and vertical routing channels, their segmentation, and the way how routing tracks connect to the ports of the logic block. The routing resources also define the types of programmable switches which link the routing wire segments together.

The logic tile is shown in Figure 7.6. The tile has three different types of ports: logic ports L_L (left), L_R (right), L_T (top) and L_B (bottom), routing ports R_{HL} (horizontal left), R_{HR} (horizontal right), R_{VT} (vertical top), R_{VB} (vertical bottom), and direct ports D_I (inputs) and D_O (outputs). The *logic ports* connect the ports of the logic block to the routing tracks of neighbouring tiles; the *routing ports* are the end terminals of the routing tracks of the same logic tile; the *direct ports* enable direct connectivity of neighbouring logic tiles, that is, without passing through programmable switches. L in Figure 7.6 denotes the set of all ports of the logic block. L includes the sets of the primary input ports I, secondary input ports T,

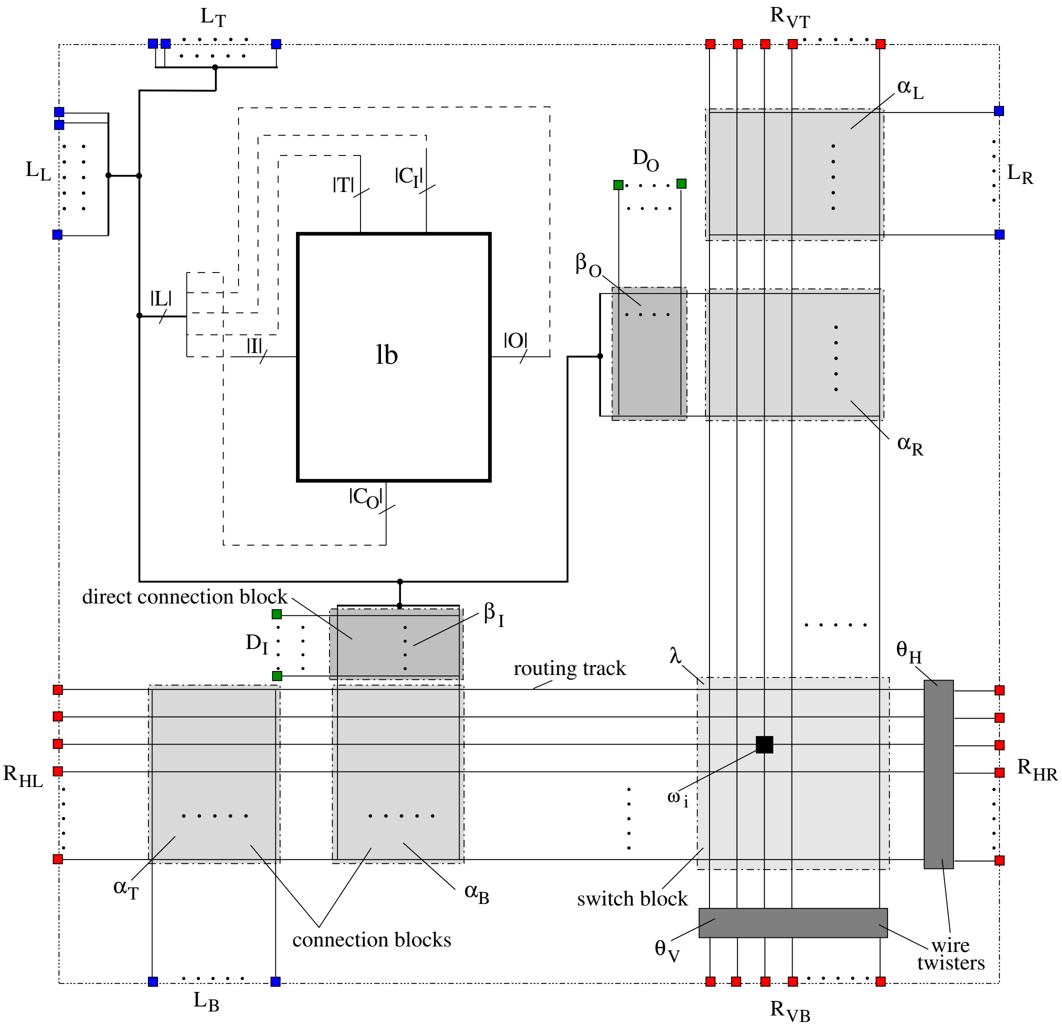


Figure 7.6. Logic tile.

and carry input ports C_I, as well as the sets of output ports O and carry output ports C_O, that is

$$L = I \cup T \cup C_I \cup O \cup C_O. \tag{7.4}$$

The ports in the set L of the logic block are distributed to the ports in the sets L_L and L_T of the logic tile. The ports in the set L_L connect to the routing tracks of the logic tile on the left via the ports in the set L_R of the left logic tile; the ports in the set L_T connect to the routing tracks of the logic tile on the top via the ports in the set L_B of the top logic tile. The ports in the set L of the logic tile also connect to the routing tracks within that tile. The connections of the logic block ports to the routing tracks are realised in the so-called *connection blocks*. In contrast to [14], we describe the connectivity in the connection blocks using a matrix notation. The rows of the *connectivity matrix* are elements of the routing port sets, while the columns are elements of the logic block port sets. The connectivity matrix is filled with ‘0’s and ‘1’s. ‘1’ at the $(i,\ j)$ position in the matrix means the presence of a connection between an i-th routing track and j-th logic block port; ‘0’ means no connection. The connection blocks of the logic tile which is shown in Figure 7.6 are modelled by the connectivity matrices defined by the functions α_T, α_B, α_L and α_R as follows

$$\alpha_T : (R_{HL} \times L_B) \rightarrow \{0,1\}, \tag{7.5}$$

$$\alpha_B : (R_{HL} \times L) \rightarrow \{0,1\}, \tag{7.6}$$

$$\alpha_L : (R_{VT} \times L_R) \rightarrow \{0,1\}, \tag{7.7}$$

$$\alpha_R : (R_{VT} \times L) \rightarrow \{0,1\}. \tag{7.8}$$

The connectivity in *direct connection blocks*, that is, between logic block ports and the direct ports of the logic tile, is defined in a similar way. The rows of the connectivity matrix in this case are addressed by the elements of the direct port set D_I or D_O, and the columns by the elements of the logic block port set L. The direct connection block for inputs is described by the connectivity matrix defined by the function β_I, while the direct connection block for outputs by the connectivity matrix defined by the function β_O. Note, that the connectivity matrix of the direct connection block for inputs has its last $|O|+|C_O|$ columns filled with ‘0’ (no connections to the output ports of the logic block), whereas the connectivity matrix of the direct connection block for outputs has its first $|I|+|T|+|C_I|$ columns filled with ‘0’ (no connections to the input ports of the logic block). The functions β_I and β_O are defined as follows.

$$\beta_I : (D_I \times L) \rightarrow \{0,1\} \tag{7.9}$$

$$\beta_O : (D_O \times L) \rightarrow \{0,1\} \tag{7.10}$$

The input and output ports of the logic block that connect to the exactly the same set of routing tracks (via the logic ports of the logic tile) as well as to the same set of the direct input and direct output ports of the logic tile can be reduced to a single port only. This allows the reduction of the implementation cost of the routing architecture.

In Figure 7.7(a), an example of the connectivity between selected ports of the logic block, the direct ports, and the routing tracks of the horizontal routing channel is shown. Figure 7.7(b) shows the corresponding connectivity matrices (functions α_B and β_I), and Figure 7.7(c) shows the suggested implementation method of the connection blocks.

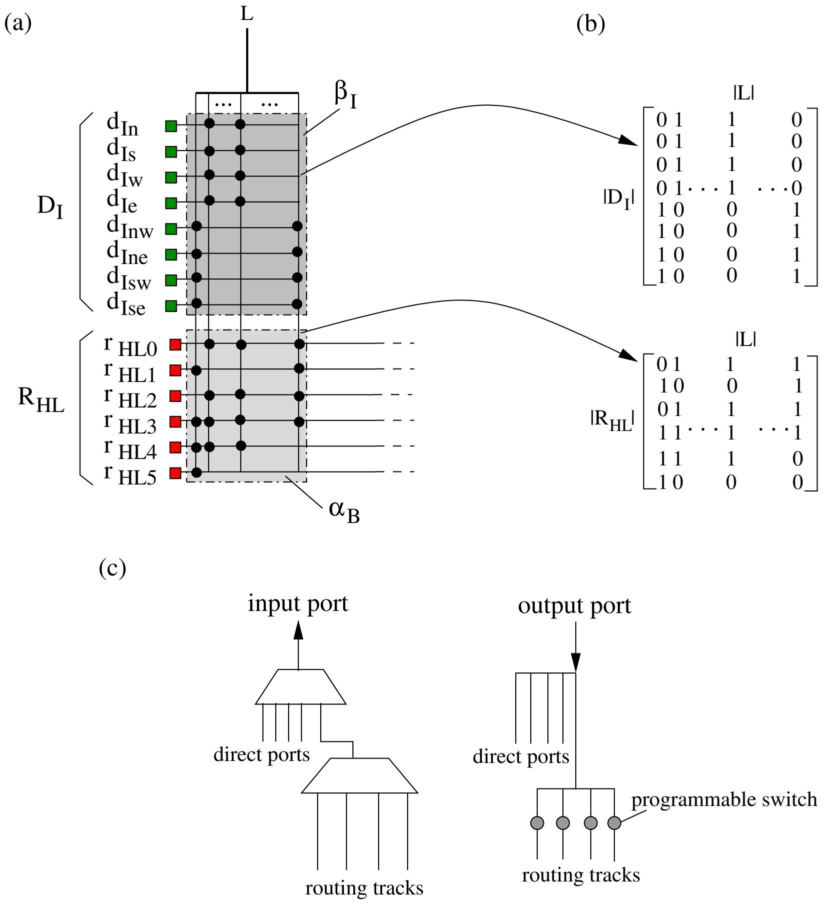


Figure 7.7. Connection blocks: (a) a connectivity example, (b) corresponding connectivity matrices, and (c) the suggested implementation method of the input and output connection blocks.

The segmentation (length) of the routing tracks (expressed in the number of logic blocks that a routing track, separated by two programmable switches, span), the switch block architecture (i.e. the way how routing tracks in horizontal and vertical routing channels are connected together), and the type of programmable switches are described by the *switching matrix* (see Figure 7.6). The rows of the switching matrix are elements from the routing port set R_{HL}, and the columns are the elements from the routing port set R_{VT}. The switching matrix is defined by the function λ such that

$$\lambda : (R_{HL} \times R_{VT}) \rightarrow \{0, \Omega\}. \qquad (7.11)$$

The switching matrix is filled with ‘0’s (no connection between crossing tracks) or the elements of the set Ω (see Equation 7.12) that describes switching point types.

$$\Omega = \{\omega_i : i \in \mathbb{N}\} \qquad (7.12)$$

A *switching point* ω_i is described by the segment connection pattern and the type of a programmable switch (see Figure 7.8). The segment connection pattern defines the number of connections (paths) between horizontal and vertical track segments that are associated with a given switching point, while the programmable switch describes the implementation of such connections. For example, for two different types of the segment connection patterns (e.g. 'disjoint' [14] and 'half' shown in Figure 7.8(a)) and three types of programmable switches (e.g. a pass transistor, a dual-pass gate and a bidirectional buffered switch shown in Figure 7.8(b)), six different switching points $\omega_1 \dots \omega_6$ are possible.

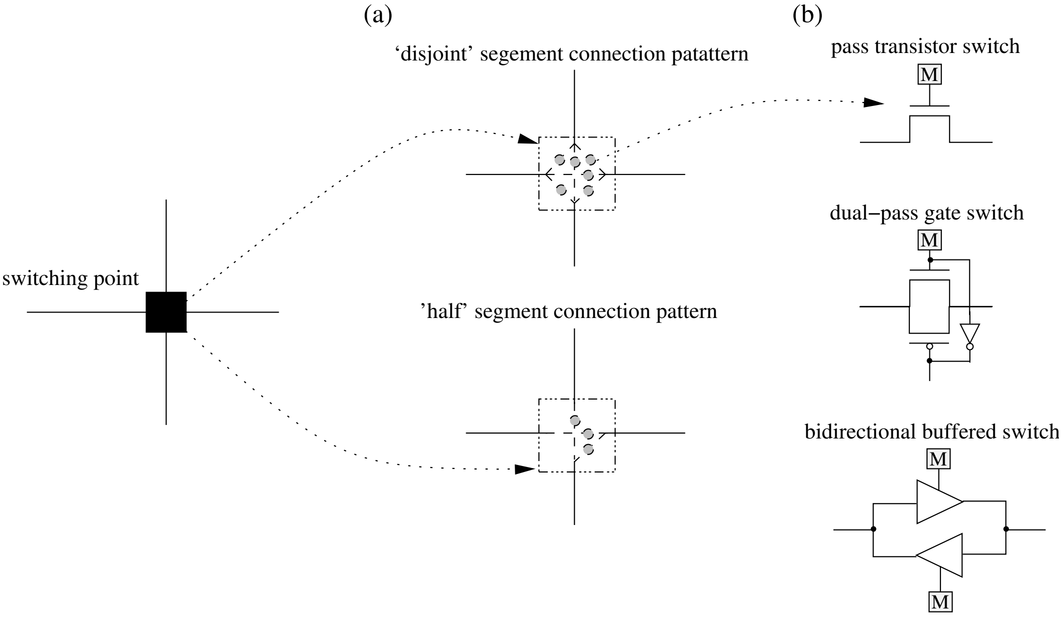
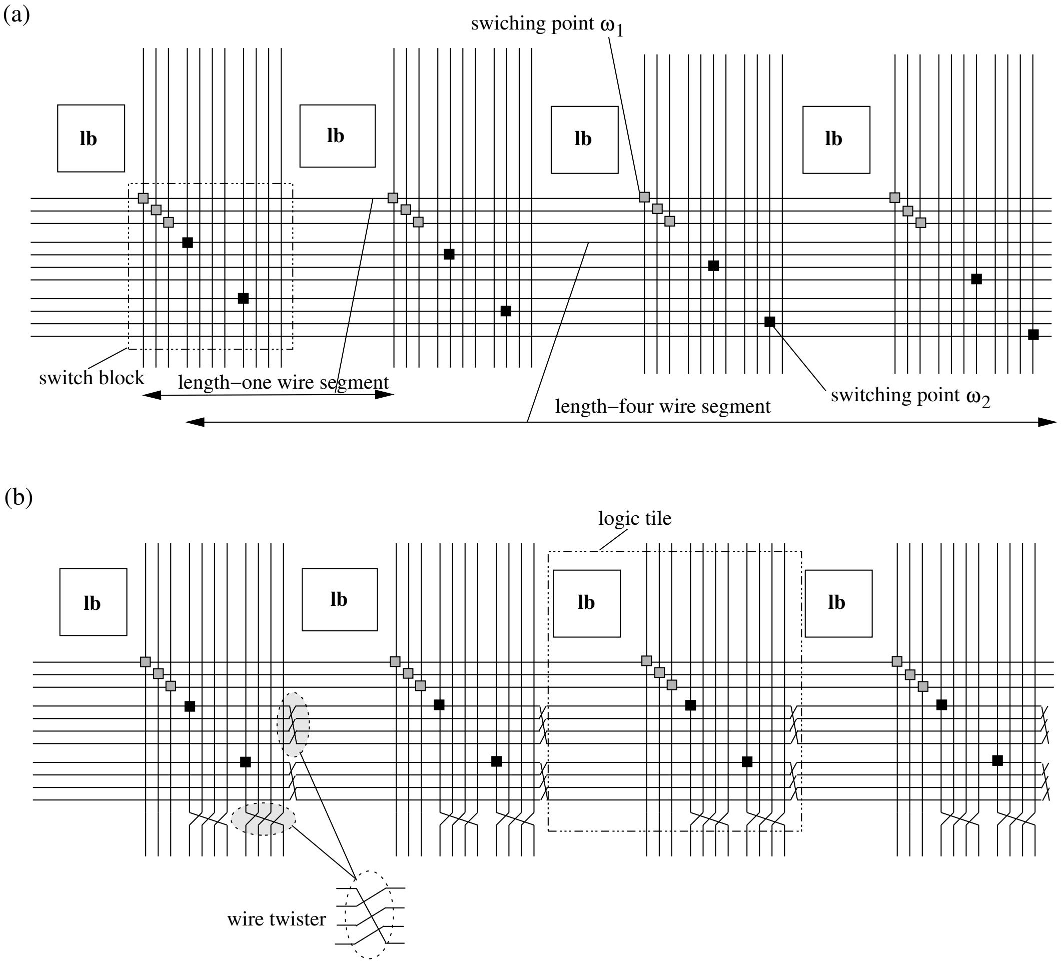

(a)
(b)
'disjoint' segement connection patattern
pass transistor switch
M
dual–pass gate switch
M
switching point
'half' segment connection pattern
bidirectional buffered switch
M
M

Figure 7.8. Examples of various switching points: (a) 'disjoint' segment connection pattern and 'half' connection pattern, (b) the programmable switch types: a pass transistor, a dual-pass gate, and a bidirectional buffered switch.

The horizontal and vertical tracks in the logic tile end with the so-called *wire twisters*. Thanks to the wire twisters, the routing resources of each logic tile can be made identical. Consequently, only one type of logic tile, rather than many different types, suffices to implement a reconfigurable logic core. The wire twisters are needed if the routing architecture includes routing segments longer than the length-one, that is spanning more than one logic block. Such long segments must be twisted in the way illustrated in Figure 7.9(b)). Furthermore, the total number of tracks of a given length must always be the multiple of the wire segment length. For example, the acceptable numbers of the routing tracks of the length-four are: 4, 8, 12, 16, etc.

The wire twisting scheme in the horizontal and vertical routing channels is described by the *twist matrices* defined by the functions θ_H and θ_V, respectively, such that:

$$\theta_H : (R_{HL} \times R_{HR}) \rightarrow \{0,1\}, \tag{7.13}$$



(c)

Switching matrix

$$\begin{bmatrix}
\omega_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \omega_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \omega_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \omega_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \omega_2 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

Twist matrix

$$\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{bmatrix}$$

Figure 7.9. Routing architecture modelling: (a) an example architecture with three length-one routing tracks and eight length-four routing tracks, (b) an example implementation of the routing architecture with a wire twisting scheme, (c) the switching matrix and twist matrix (horizontal and vertical) for the example routing architecture.

$$\theta_V : (R_{VT} \times R_{VB}) \rightarrow \{0,1\}. \tag{7.14}$$

The rows of the twist matrices are elements of the routing ports sets on the left and top of the logic tile, that is, R_{HL} and R_{VT}, respectively. The columns of the matrices are elements of the routing ports sets on the right and bottom of the logic tile, that is, R_{HR} and R_{VB}, respectively. The matrices are filled with ‘0’s and ‘1’s. ‘1’ means a connection between the routing tracks that are associated with given routing ports, ‘0’ means no connection. Typically, the horizontal and vertical twist matrices are identical.

Figure 7.9 shows an example of the routing architecture. In such an architecture, the routing channel consists of three length-one tracks and eight length-four tracks. Figure 7.9(a) illustrates the architecture in a conceptual way. Note, that the length-one wire segments of routing tracks use switching point of the type ω_1 (e.g. a ‘disjoint’ segment connection pattern and pass-transistor-based switch), while the length-four wire segments use switching point of the type ω_2 (e.g. a ‘disjoint’ segment connection pattern and a buffer-based switch). In Figure 7.9(b), the implementation details of such architecture are shown. The wire segments of the length longer than one are twisted according to the modulo-length scheme. Figure 7.9(c) shows the switching matrix and the twist matrix (horizontal and vertical) of the tile.

Input/output tile with routing

The *input/output tile with routing (iort)* contains elements that play a role of the interface between the logic resources of the reconfigurable logic core and resources of the system in which the core is embedded. As indicated by its name, the input/output tile with routing is also enhanced with routing resources. The tile is available in two versions, that is, a top and a left version. Both versions of the tile are defined in an analogous way to the definition of elements of the logic tile.

The *top* input/output tile with routing (see Figure 7.10(a)) has two sets of input/output ports F_T and G_B, and three sets of routing ports, that is, R_{HL}, R_{HR}, and R_{VB}. The ports in the set F_T connect to the system resources, while the ports in the set G_B enable the connection of the ports in the set L_T of a logic tile at the top of the array to the routing resources of the top input/output tile with routing. The routing ports in the sets R_{HL} and R_{HR} connect to the ports in the sets R_{HR} and R_{HL} of the neighbouring *iort* tiles, respectively. The ports in the set R_{VB} connect to the ports in the set R_{VT} of a logic tile at the top of the array. The set E is the set of the direct input and output ports of the tile and it connects to the direct input ports and direct output ports in the sets D_I and D_O, respectively. The functions γ_T, γ_B and δ_T that describe connectivity matrices of the tile in Figure 7.10(a) are defined as follows.

$$\gamma_T : (R_{HL} \times G_B) \rightarrow \{0,1\} \tag{7.15}$$

$$\gamma_B : (R_{HL} \times F_T) \rightarrow \{0,1\} \tag{7.16}$$

$$\delta_T : (E \times F_T) \rightarrow \{0,1\} \tag{7.17}$$

The *left* input/output tile with routing (see Figure 7.10(b)) consists of the same elements as the top input/output tile with routing. The positions of the elements are, however, mirrored with respect to the positions of elements in the top input/output tile with routing. The left input/output tile with routing has two sets of input/output ports F_L and G_R, three sets of routing ports, that is, R_{VB}, R_{VT}, and R_{HR}, and the set of direct ports E. The ports in the set F_L connect to the system resources, while the ports in the set G_R enable the connection of the ports in the set L_L of the logic tile on the left edge of the array to the routing resources of the left input/output tile with routing. The routing ports in the sets R_{VB} and R_{VT} connect to the ports in the sets R_{VT} and R_{VB} of the neighbouring *iort* tiles, respectively. The ports in the set R_{HR} connect to the ports in the set R_{HL} of the logic tile at the left edge of the array. The connectivity matrices of the tile in Figure 7.10(b) are defined by the functions γ_L, γ_R and δ_L such that:

$$\gamma_L : (R_{VT} \times G_R) \rightarrow \{0,1\}, \tag{7.18}$$

$$\gamma_R : (R_{VT} \times F_L) \rightarrow \{0,1\}, \tag{7.19}$$

$$\delta_L : (E \times F_L) \rightarrow \{0,1\}. \tag{7.20}$$

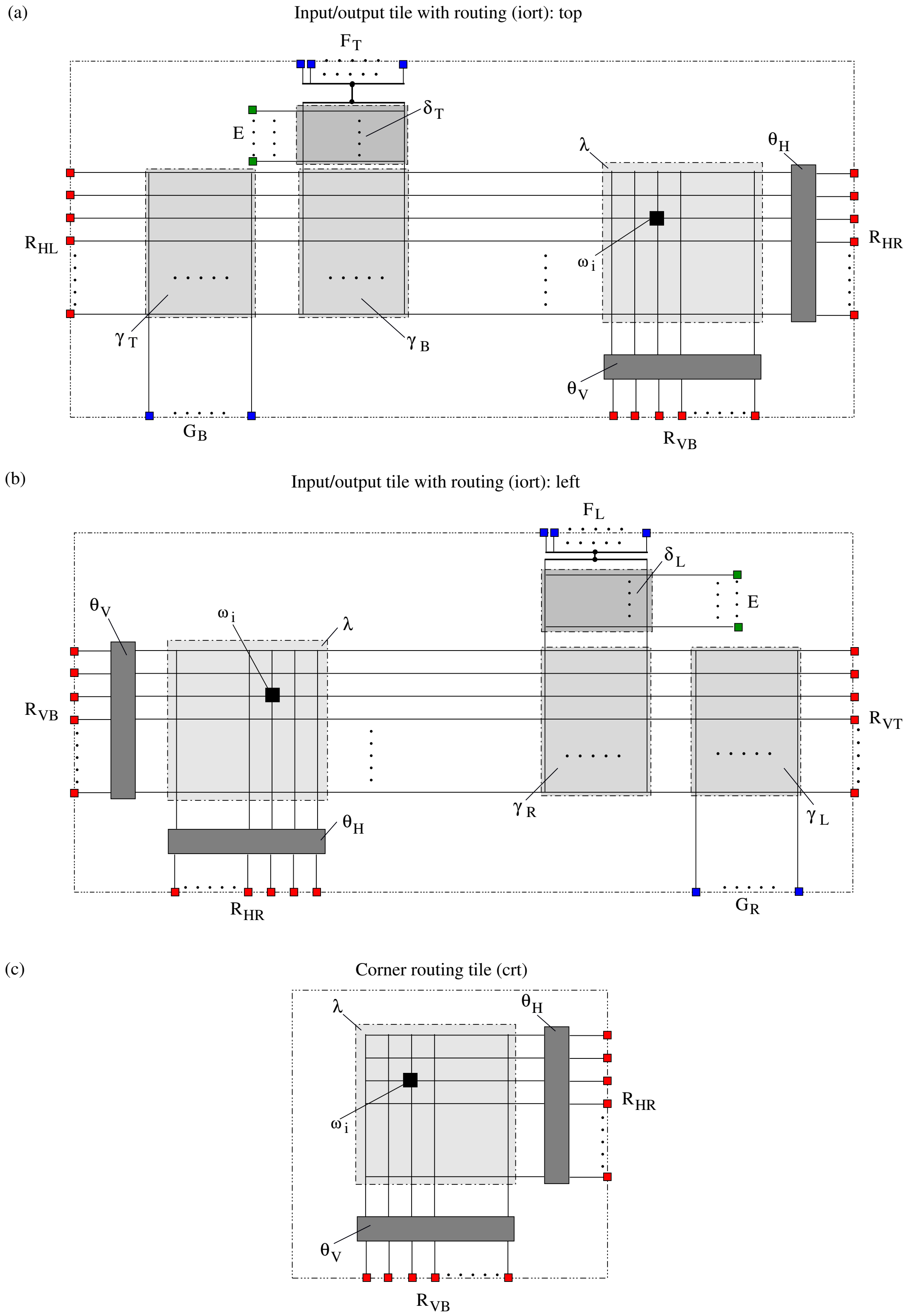
Corner routing tile

The *corner routing tile (crt)* (see Figure 7.10(c)) contains only routing resources that supplement routing resources of the input/output tiles with routing. The corner routing tile has two sets of routing ports, that is, R_{VB} and R_{HR}. The ports in the set R_{VB} connect to the ports in the set R_{VT} of the top input/output tiles with routing. The ports in the set R_{HR} connect to the ports in the set R_{HL} of the left input/outputs tile with routing.

Input/output tile

The *input/output tile (iot)* contains elements that play a role of the interface between the logic resources of the reconfigurable logic core and resources of the system in which the core is embedded, and no routing resources. The input/output tile has two versions, that is, a right and a bottom version.

The *right* input/output tile (see Figure 7.11(a)) has two sets of input/output ports F_R and G_L, and the set of direct ports E. The ports in the set F_R connect to the system resources, while the ports in the set G_L connect to the routing resources of



(a) Input/output tile with routing (iort): top

(b) Input/output tile with routing (iort): left

(c) Corner routing tile (crt)

Figure 7.10. Components of a reconfigurable logic core: (a) input/output tile with routing (a top version), (b) input/output tile with routing (a left version), (c) corner routing tile.

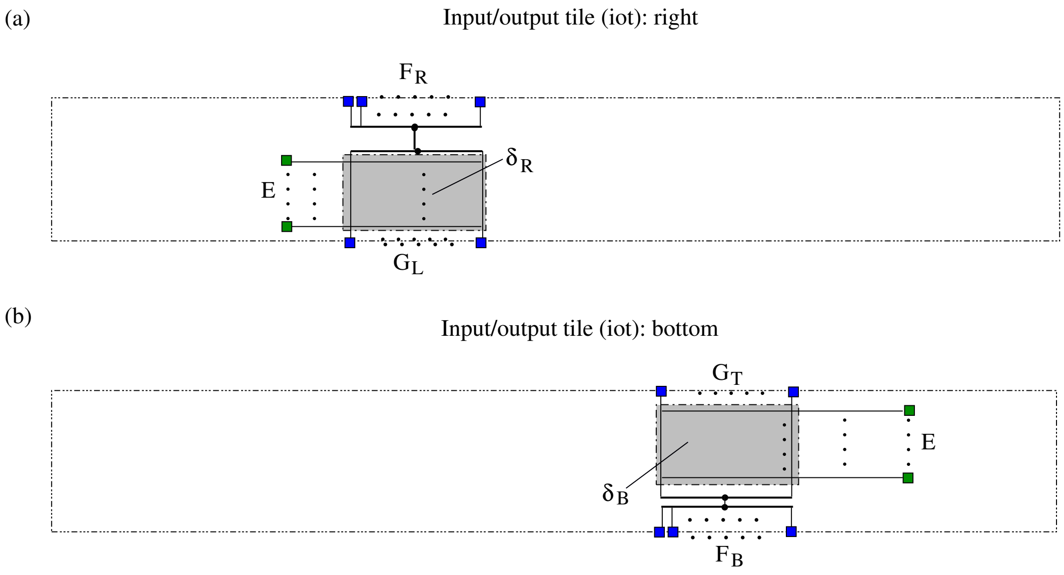

(a)
Input/output tile (iot): right
F_R
δ_R
E
G_L
(b)
Input/output tile (iot): bottom
G_T
E
δ_B
F_B

Figure 7.11. Components of a reconfigurable logic core: (a) input/output tile (a right version), (b) input/output tile (a bottom version).

logic tiles via the set L_R of the logic tile ports. The connectivity matrix for direct connections of the right input/output tile is defined by the function δ_R such that

$$\delta_R : (E \times F_R) \rightarrow \{0, 1\}. \tag{7.21}$$

The *bottom* input/output tile (see Figure 7.11(b)) consists of the same elements as the right input/output tile, but the elements are mirrored. The tile has two sets of input/output ports F_B and G_T, and the set of direct ports E. The ports in the set F_B connect to the system resources, while the ports in the set G_T connect to the routing resources of logic tiles via the set L_B of the logic tile ports. The connectivity matrix for direct connections of the bottom input/output tile is defined by the function δ_B such that

$$\delta_B : (E \times F_B) \rightarrow \{0, 1\}. \tag{7.22}$$

7.2.5 Level V – Array

The top level of a reconfigurable logic architecture is an array of logic tiles. The total number of logic tiles and the aspect ratio of the array are parameters. The logic tiles (*lt*) are, as shown in Figure 7.12, surrounded by the input/output tiles with routing (*iort*) and the input/output tiles (*iot*). The *iort* and *iot* tiles have a twofold function. Firstly, they are an interface between a reconfigurable logic fabric and the system resources. Secondly, they complete the routing architecture. The latter is required because the external routing channel created by the routing resources of the logic tiles on the edge of the array is present only at the bottom and at the right side of the array. Therefore, the input/output tiles with routing

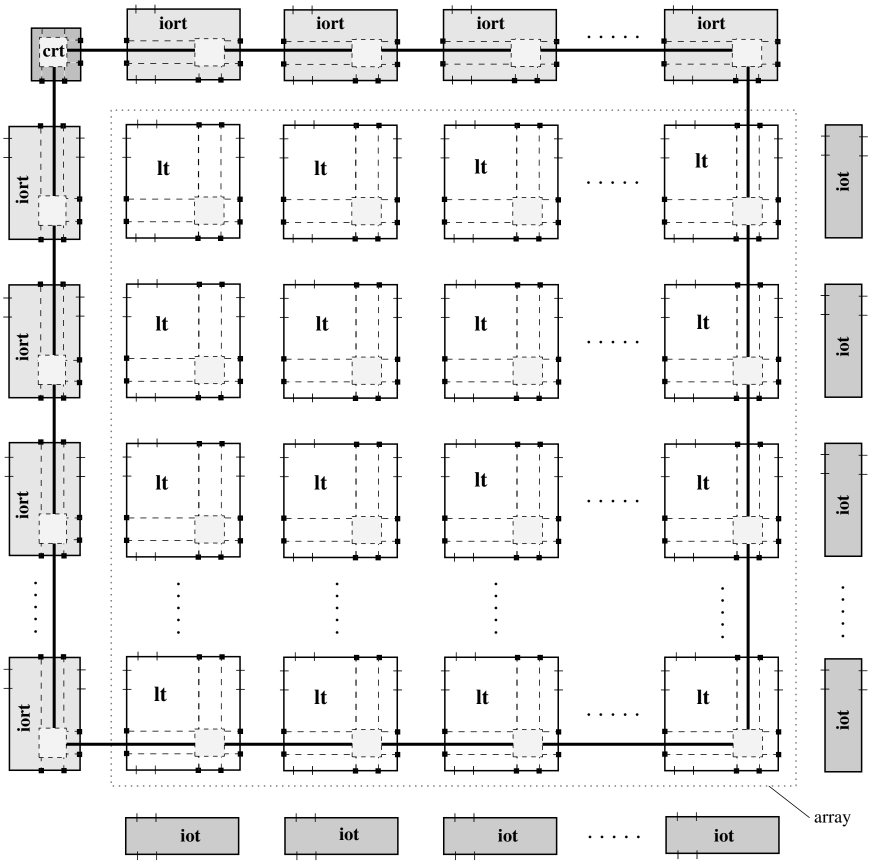


Figure 7.12. Array – a top-level view of a reconfigurable logic architecture.

(*iort*) are placed on the left and at the top side of the array, and simple input/output tiles (*iot*) are placed at the right and at the bottom side of the array. Additionally, a corner routing tile (*crt*) that closes the external routing channel is placed at the left top corner of the array. The bold ring in Figure 7.12 shows a resultant routing channel created in this way.

All tiles of the array are abutted via their routing or logic ports, as shown in Figure 7.12. Note also, that the connectivity matrices λ of each tile are defined identically. Consequently, the right functioning of the switch blocks in the logic tiles at the edge of the array and the input/output tiles with routing must be guaranteed by the proper programming of the configuration memory. This means, for example, that programmable switches of the right bottom logic tile are programmed such that no routing connections to the bottom and to the right sides of this tile are possible.

7.3 Architecture modelling examples

In this section, the described architecture template is applied to model application-domain-oriented reconfigurable logic architectures that have been presented in

Chapters 4–6. (Because of the example character of the routing structures that have been proposed for each logic block architecture, we focus only on the first three levels of hierarchy of the template, that is up to the logic block level.) We also discuss some architectural implications of our template.

7.3.1 Template instances

Table 7.7 lists the values of the template parameters that characterise three application-domain-oriented instances of the reconfigurable architecture template, that is, the data-path-oriented instance (see Chapter 4), random-logic-oriented instance (see Chapter 5) and memory-oriented instance (see Chapter 6). The table also summarises other characteristics of the template instances (such as the number of input and output ports at each hierarchy level), which are determined by the template parameter values. Basic functionality at each level, defined as the number of inputs of the largest Boolean, arithmetic, and memory functions that can be mapped, is also mentioned.

Figures 7.13, 7.14 and 7.15 show the logic block structures of the three proposed application-domain-oriented architectures that are modelled according to the template rules. The instances are modelled as follows:

- *Data-path-oriented* instance is built of one processing element ($|M| = 1$). The processing element contains four logic elements ($|N| = 4$) of the 'data-path' type ($\tau : data - path$).
- *Random-logic-oriented* instance is built of four processing elements ($|M| = 4$). Each processing element contains one logic element ($|N| = 1$) of the 'random-logic' type ($\tau : random - logic$).
- *Memory-oriented* instance is built of two processing elements ($|M| = 2$) forming two separate clusters ($|K| = 2$). The processing element of each cluster contains one logic element ($|N| = 1$) of the 'memory' type ($\tau :$ *memory*).

Level of hierarchy	**Instance type**		
	Data-path-oriented	*Random-logic-oriented*	*Memory-oriented*
Logic element	TEMPLATE PARAMETERS		
	$\mathcal{T}$: data-path	$\mathcal{T}$: random-logic	$\mathcal{T}$: memory
	DERIVED PARAMETERS		
	$\|P\|$=2 $\|S\|$=3 $\|A\|$=1	$\|P\|$=4 $\|S\|$=0 $\|A\|$=1	$\|P\|$=6 $\|S\|$=1 $\|A\|$=2
	FUNCTIONALITY		
	Boolean: 2-in arithmetic: 1-bit memory: –	Boolean: 4-in arithmetic: 1-bit memory: –	Boolean: 5-in arithmetic: 2-bit memory: 2-bit
Processing element	TEMPLATE PARAMETERS		
	$\|N\|$=4	$\|N\|$=1	$\|N\|$=1
	DERIVED PARAMETERS		
	$\|X\|$=8 $\|S\|$=3 $\|Y\|$=4	$\|X\|$=4 $\|S\|$=0 $\|Y\|$=1	$\|X\|$=6 $\|S\|$=1 $\|Y\|$=2
	FUNCTIONALITY		
	Boolean: 4-in arithmetic: 4-bit memory: –	Boolean: 4-in arithmetic: 1-bit memory: –	Boolean: 5-in arithmetic: 2-bit memory: 2-bit
Logic block	TEMPLATE PARAMETERS		
	$\|M\|$=1 $\|K\|$=1	$\|M\|$=4 $\|K\|$=1	$\|M\|$=2 $\|K\|$=2
	DERIVED PARAMETERS		
	$\|I\|$=8 $\|T\|$=3 $\|O\|$=4	$\|I\|$=11 $\|T\|$=0 $\|O\|$=4	$\|I\|$=12 $\|T\|$=2 $\|O\|$=4
	FUNCTIONALITY		
	Boolean: 4-in arithmetic: 4-bit memory: –	Boolean: 6-in arithmetic: 4-bit memory: –	Boolean: 6-in arithmetic: 4-bit memory: 4-bit
Logic tile	shorted ports: ci_1 and t_1	shorted ports: –	shorted ports: ci_1 and t_1 ci_2 and t_2

Table 7.7. Modelling of the proposed application-domain-oriented architectures using the template parameters. (For completeness' sake, the ports that must be shorted at the logic tile level to obtain correct instance structures are also mentioned.)

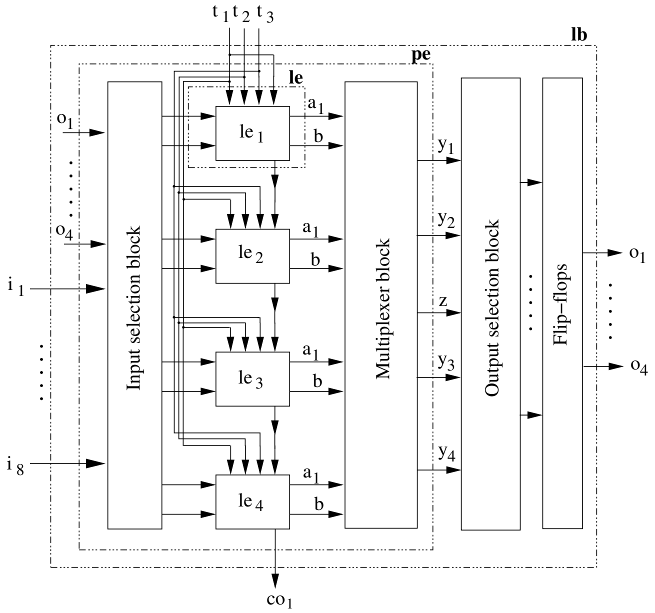


Figure 7.13. Model of the data-path-oriented template instance.

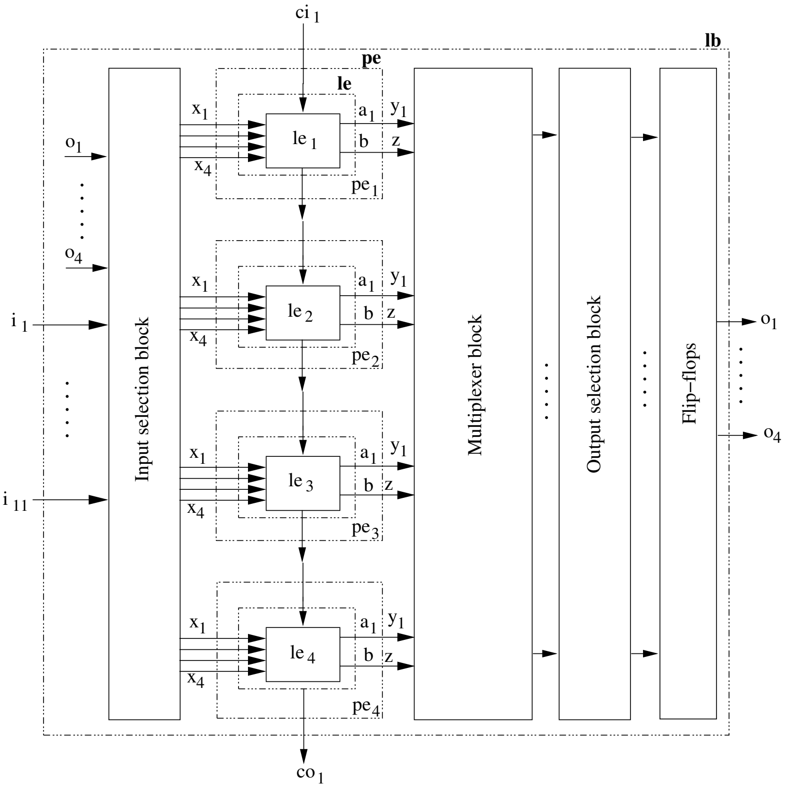


Figure 7.14. Model of the random-logic-oriented template instance.

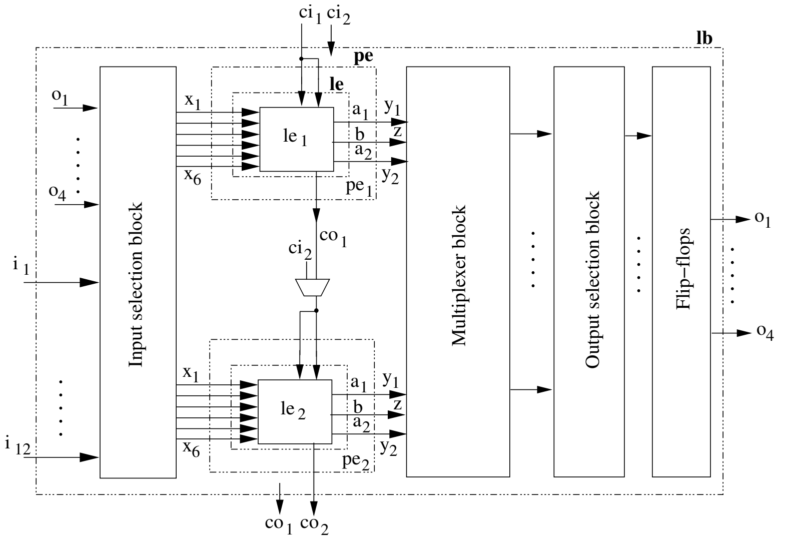


Figure 7.15. Model of the memory-oriented template instance.

7.3.2 Discussion

Despite strict rules that govern each level of hierarchy of the proposed architecture template, a careful selection of the template parameters is required to obtain a template instance with desired characteristics. For example, Figure 7.16 shows two logic block structures that are built of the same number of logic elements. Though similar in structure, the logic blocks differ in the provided functionality. This is because different template parameters are used to characterise both instances. The logic block shown in Figure 7.16(a) has two processing elements ($|M| = 2$) , each with two logic elements ($|N| = 2$). In contrast, the logic block shown in Figure 7.16(b) has only one processing element ($|M| = 1$), but which contains four logic elements ($|N| = 4$). Therefore, while the first logic block can implement up to two 5-input Boolean functions, the second logic block allows an implementation of four independent 4-input Boolean functions. The latter logic block instance offers thus a higher degree of flexibility.

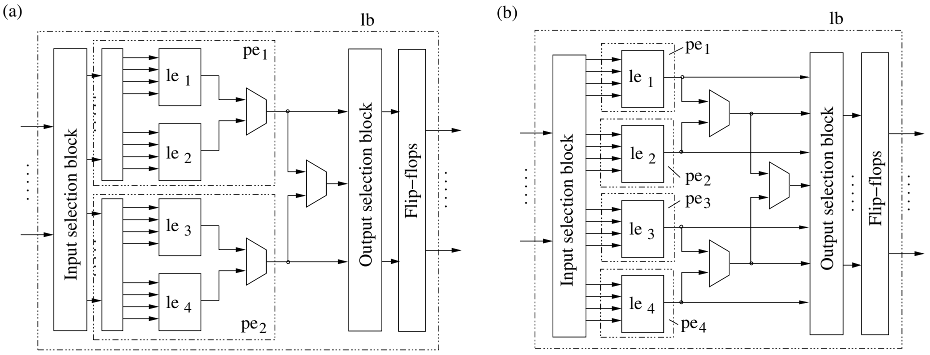


Figure 7.16. A modelling example: two logic blocks with an identical number of logic elements but different functionality.

7.4 Template-based design

The above-discussed template is a *conceptual model* of a reconfigurable logic architecture. The same model can be easily translated into an *executable model*. The executable model supports the design of application-domain-oriented reconfigurable logic cores for the SoC integration. Three primary methods of applying the proposed template include:

- architecture exploration,
- physical design,
- application mapping.

Below, each of the methods is briefly explained.

7.4.1 Architecture exploration

The template is used to model various application-domain-oriented reconfigurable logic architectures. A given application domain is characterised by the parameters of the application domain model according to the procedure discussed in Section 2.3. The parameters of the application domain model can be translated into the parameters of the architecture template in the way shown in Table 7.8.

Application parameter	**Architecture template parameter**
Type of processing	τ: logic element type
Word-size	$\|N\| \cdot \|M\|$ $\|N\|$: number of logic elements $\|M\|$: number of processing elements
Rent exponent	λ: switching matrix

Table 7.8. The correspondence between the parameters of the application domain model and the parameters of the reconfigurable architecture template.

The template parameters that are instantiated based on the results of the application domain characterisation provide a starting point for the process of architecture exploration. The use of the template improves the quality of an initial solution and allows systematical experiments on a large set of potentially interesting solutions.

7.4.2 Physical design

Two phases of the physical design process clearly benefit from the proposed architecture template. They are:

- netlist generation, and
- layout creation.

With a clearly defined set of rules describing each level of hierarchy of the reconfigurable logic architecture template, the netlist of each template instance can be generated *automatically*. Such a netlist can be used both for a functional verification of the core (e.g. a netlist in a behavioural HDL) or for a VLSI implementation of the core (e.g. a netlist in a synthetisable HDL). In the latter case, the netlist is mapped onto a library of predefined components (cells) and is used as an input to an automatic placement and routing tool[5]. Because the quality (in terms of area) of the final layout may be unsatisfactory, more customised layout generation techniques are also possible.

The *modular layout* is one of such techniques. It describes a hierarchical layout approach which allows the creation of the layout of a complete reconfigurable logic core using the layouts of a limited number of basic building blocks called *tiles*. As we showed in Section 7.2, the layouts of only six different blocks, that is the logic tile, two versions of the input/output tile, two versions of the input/output tile with routing, and the corner routing tile, are needed. The layouts of the tiles are abutted and connected together via the tile ports. The layout of each tile may be of a full-custom type (a hand-crafted layout) or a semi-custom type (an automatically generated layout based on a library of basic cells).

7.4.3 Application mapping

The tools supporting an application mapping flow of the template-based reconfigurable architectures can be made *retargetable*. This means that by setting the parameters, the functionality of the tools can be modified to reflect specific characteristics of a target reconfigurable architecture. In the process of mapping an application onto a reconfigurable logic fabric two main phases can be distinguished, that is:

- logic synthesis with technology mapping, and
- placement and routing.

State-of-the-art retargetable mapping tools, such as *FlowMap* [32] and *VPR* [105], prove the feasibility of applying architecture templates in the application mapping flow. Both tools cover only one of the above-mentioned application mapping phases. For example, FlowMap, proposed at the University of California at Los Angeles, addresses the technology mapping problem. FlowMap enables the mapping onto k-input LUTs, where k is a parameter and can be set by a user. The tool guarantees the delay-optimal mapping. VPR (Versatile Placement and Routing) is

[5] Such an approach is equivalent to a traditional standard-cell mapping.

a retargetable placement and routing tool. VPR accepts a description of an FPGA architecture in a high-level format and automatically generates its routing structure. Because the placement and routing algorithms recognise the changes in the underlying routing architecture and adopt to them, the mapping onto architectures with considerably different routing structures is possible.

Figure 7.17 shows an example of a simple reconfigurable logic core (for the sake of simplicity, the core with one logic tile only is shown), which has been generated in the architecture generator called *Archimed*. Archimed [33] has been implemented according to the proposed architecture template. The tool allows an automatic generation of high-level netlists that describe the structure of domain-oriented reconfigurable cores. The cores are built using four types of basic building blocks (tiles) that have been described in Section 7.2.4.

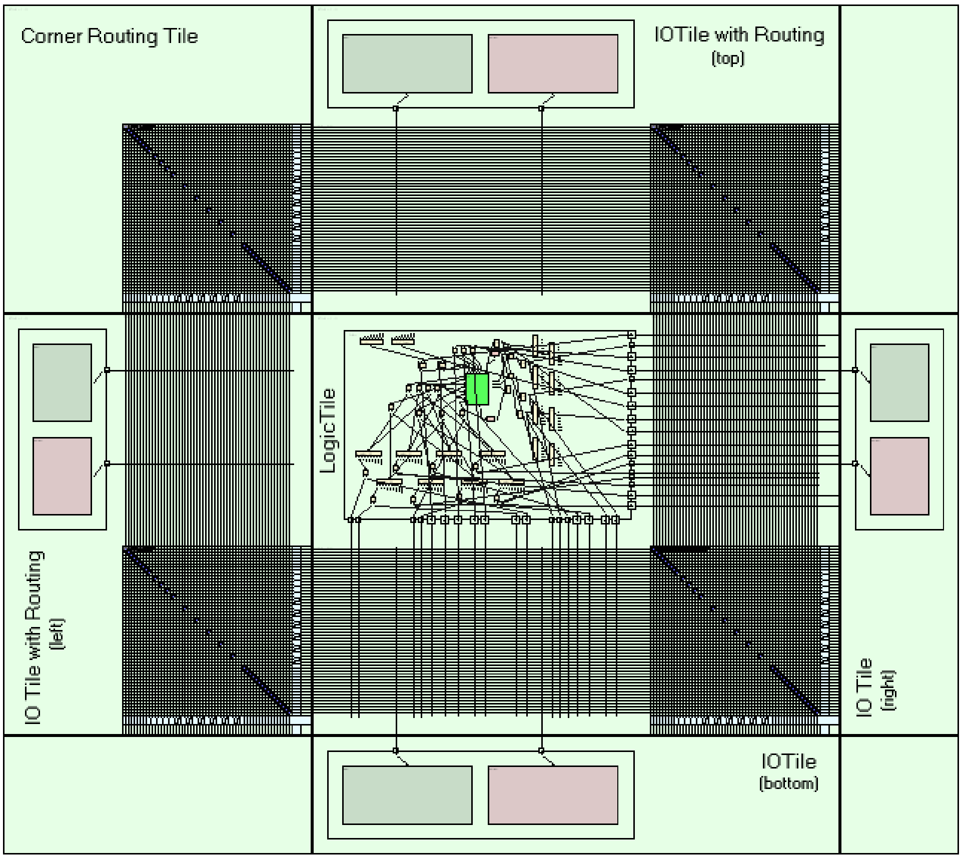
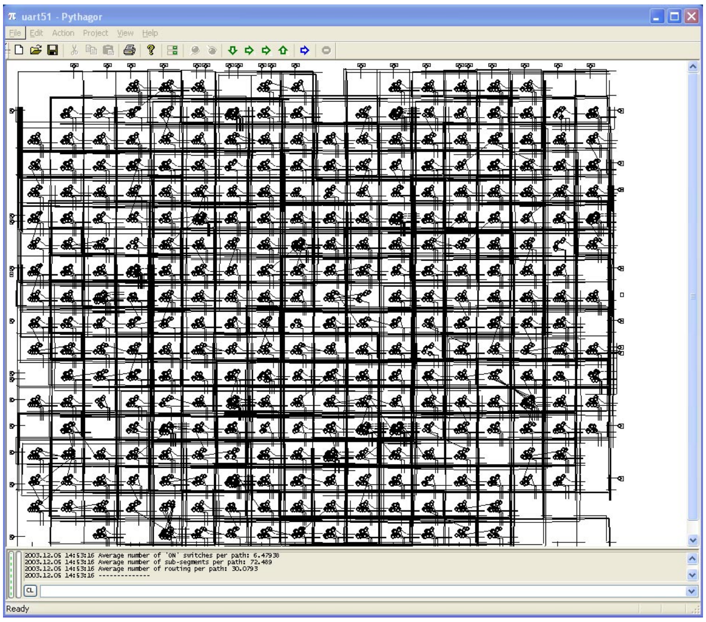


Figure 7.17. An example of a simple reconfigurable logic core which has been generated using the proposed architecture template. Four basic tiles (i.e. a logic tile, input/output tile, input/output tile with routing, and a corner routing tile) can be identified. The specification of the reconfigurable logic core that is generated automatically in the architecture generator called Archimed is used as an input to a retargetable placement and routing tool called Pythagor.

The architecture specification generated by Archimed is used in the retargetable placement and routing tool called Pythagor [34]. Such a specification defines a logic fabric on which applications are mapped. The template-based architecture specification allows the introduction of changes into the routing (and logic) resources of the architecture and analysis of their impact on the routability. The screen-shot of Pythagor is shown in Figure 7.18.



```
2003.12.05 14:53:16 Average number of 'ON' switches per path: 6.47938
2003.12.05 14:53:16 Average number of sub-segments per path: 72.489
2003.12.05 14:53:16 Average number of routing per path: 30.0793
2003.12.05 14:53:16 -------------
```

Figure 7.18. Screen-shot of the retargetable placement and routing tool called Pythagor.

7.5 Conclusions

In this chapter, the template-based methodology for the reconfigurable logic design has been discussed. The methodology facilitates the architecture exploration, physical design and application mapping for reconfigurable logic cores of different types, shapes and sizes. The methodology helps to reduce design time and effort. The parametrised architecture template, which is central for the proposed methodology, is used to create application-domain-oriented architecture instances. The architecture instances are derived by setting the template parameters according to the characteristics (parameters) of a target application domain.

The proposed architecture template has five levels of hierarchy. The levels are: logic element, processing element, logic block, tile, and array. Each level of hierarchy has a strictly defined purpose. The logic element, for example, is a basic logic component of a reconfigurable logic architecture. The type of the logic element, and thus its functionality, depend on the type of processing in a target application domain. At the next level, the processing element determines the size of Boolean functions that can be implemented in a reconfigurable logic architecture. Such a size is determined by the number of logic elements in the processing element. The third level is defined by the logic block. The logic block consists of a single or several parallel clusters of processing elements. The total number of processing elements in all clusters, and thus the total number of logic elements, correspond to the data word-size of a target application domain. The type of feedback connections is also defined at the logic block level. At the fourth level, the logic tile defines the routing resources of the logic block. Other tiles, that is input/output tile, input/output tile with routing and corner routing tile, which are basic building blocks of the architecture, are also defined at this level. Finally, at the array level, the basic tiles are used to create the complete architecture of a reconfigurable logic core.

Using examples of the domain-oriented reconfigurable architectures proposed in Chapters 4–6, we showed how such architectures can be modelled using the template parameters. We also briefly discussed the role of the template in the process of designing domain-oriented reconfigurable logic cores.

Chapter 8

CASE STUDY: MEMORY-ORIENTED EFPGA CORE

In this chapter, a VLSI implementation of the memory-oriented embedded FPGA (eFPGA) core, the architecture of which has been described in Chapter 6, is presented. We discuss first implementation details of basic building blocks of the architecture. Next, we discuss a silicon prototype of the eFPGA core which has been implemented in a 0.13 μm CMOS technology using the described blocks. Finally, we compare our eFPGA core with state-of-the-art commercial FPGAs. The comparison is based on the logic tile area as the cost metric. By focusing on the logic tile area, both the logic and routing resource costs of an FPGA are taken into account.

8.1 VLSI implementation aspects

The discussion in this section is restricted to these aspects of the VLSI implementation that differentiate our approach from the implementation approaches that are typical for state-of-the-art FPGAs. Consequently, we focus on the implementation of the configuration memory, consisting of the LUT and control memories, and we discuss the implementation of programmable interconnect.

8.1.1 Memory design

The configuration memory is an inherent part of a reconfigurable logic device. By loading configuration data (program) to an uncommitted reconfigurable logic device, its functionality is established. To distinguish between two roles that the configuration memory plays typically in a reconfigurable device, we assume a division of the configuration memory into a LUT memory and a control memory (see Section 6.6). The LUT memory defines a basic functionality of a reconfigurable logic device, and optionally can also be used as a data memory. The control memory programs the routing resources (i.e multiplexers and switches in the interconnect) as well as the elements that define an operating mode of the device. In the memory-oriented reconfigurable logic architecture, the distinction between the LUT and control memories is also implied by their different implementations. This issue is addressed below.

LUT memory

The logic block of the memory-oriented reconfigurable logic device contains two 4/2-LUTs. Such multi-output LUTs have a memory-like organisation. This means that each LUT is built of a single read/write decoder and an array of memory cells, rather than of independent (read) multiplexers and independent columns of memory cells as in traditional implementations (see Section 6.1.2). The memory-like implementation of the LUTs is area efficient and facilitates their use as data memories.

Despite a clear resemblance of the 4/2-LUT structure to a traditional memory structure, the LUT implementation differs from a typical memory implementation. First of all, memory columns of the 4/2-LUT are implemented with single bit lines rather than dual bit-lines. Secondly, no precharging of memory bit-lines is used. Although this results in slightly longer read delays, it helps to avoid the problem of the synchronisation of the precharging operation with the clock signal or the pulse from an address-transition-detection circuitry [70]. Finally, no special sensing circuit is used, and read and write operations are controlled by a single transistor.

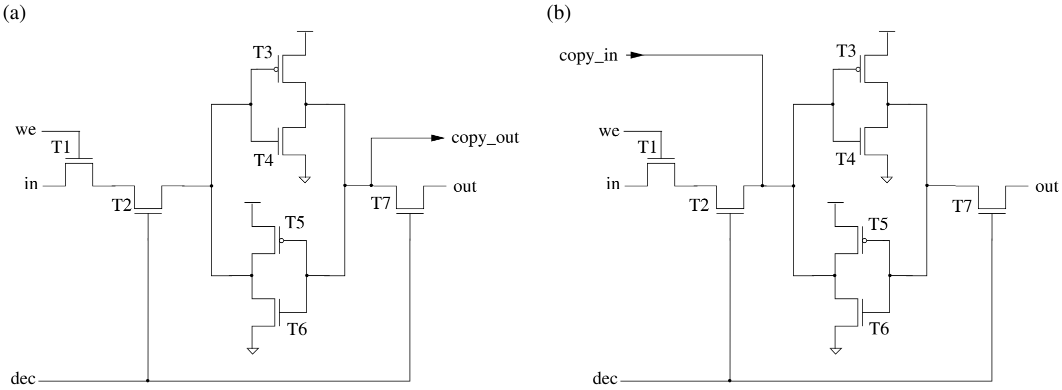


Figure 8.1. Two version of the LUT memory cell: (a) the memory cell which state is copied in the dual-port memory mode, (b) the memory cell which receives the copied information.

The LUT memory cells for first (left) and second (right) 4/2-LUTs are shown in Figures 8.1(a) and (b), respectively. The cells consist of seven transistors. Two pairs of transistors, that is, $T3-T4$ and $T5-T6$, form cross-coupled inverters. The transistors $T2-T7$ are access transistors and are coupled to the row decoder signal dec. Finally, the transistors $T1$ in both memory cells determine their operating modes, that is, writing if $we=1$ or reading if $we=0$. Both cells have a single write port in and a single read port out. Furthermore, the memory cell of the left LUT has an extra output port $copy_{out}$, and the memory cell of the right LUT has an extra input port $copy_{in}$. These ports are coupled together via the copy elements (see Figure 6.8(c)) if the LUTs are configured to implement a dual-port memory (see Section 6.4). Provided that the LUT decoders are controlled directly by the write enable signals *we*, a six-transistor LUT memory cell implementation is also possible. The example layout of the (left) LUT memory cell implemented according to the 0.13 μm CMOS design rules is shown in Figure 8.2.

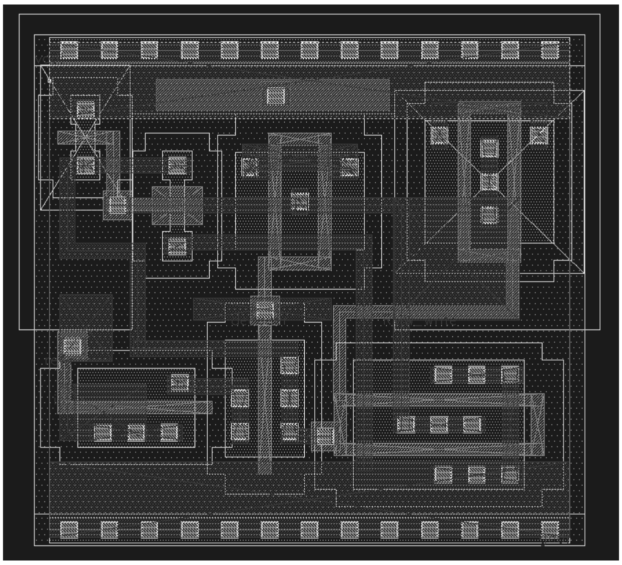


Figure 8.2. The layout of the memory cell in the left 4/2-LUT (the layout corresponds to the circuitry shown in Figure 8.1(a)).

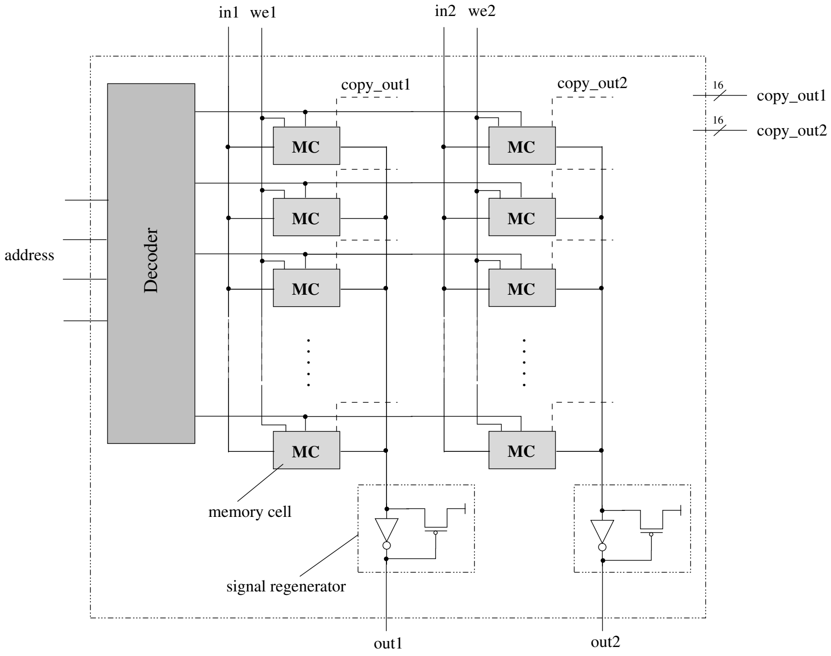


Figure 8.3. The proposed 4/2-LUT implementation.

The memory cells in the 4/2-LUT are organised into two columns of 16 cells. The output ports *out* of the cells in each column are connected together. To overcome the voltage drop during reading logic ‘1’ from the selected memory cell, a regenerative circuitry is placed at the output of each memory column. The regenerative circuitry is implemented with a small feedback PMOS transistor [80], as shown in Figure 8.3.

Control memory

As discussed in Section 6.6, the configuration memory of the memory-oriented reconfigurable logic device is designed to support a partial reconfiguration. To enable that, memory cells of the control memory are designed with six transistors rather than five transistors as in a typical configuration memory cell implementation [114]. The extra transistor guarantees an independent selection of the memory cells. The control memory cell designed in this way is shown in Figure 8.4(a), and its layout according to the 0.13 μm CMOS design rules is shown in Figure 8.5. The role of the transistors in the schematic from Figure 8.4(a) is analogous to the role they play in the LUT memory cell (see Section *LUT memory*).

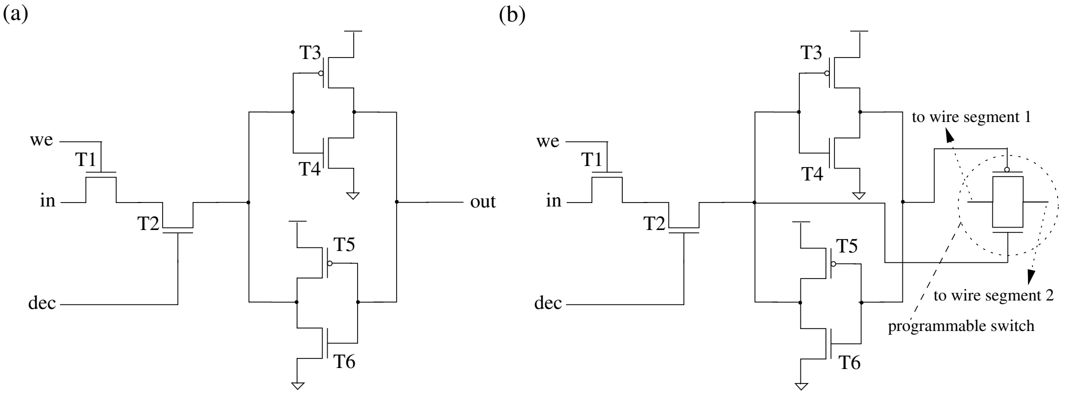


Figure 8.4. The implementation of the configuration memory cells: (a) proposed configuration memory cell, (b) configuration memory cell with the integrated dual pass-gate programmable switch.

The control memory can be implemented either as a centralised memory or a distributed memory. The *centralised control memory* is characterised by the abutted control memory cells that form one memory block. On the one hand, this enables a compact memory layout and the reduction of the size of the buffers that drive the output signals of the decoder. On the other hand, this also puts constrains on the size of transistors in the memory cells. The transistors must be made large enough to provide sufficient strength for driving long wires that connect memory cells with programmable switches. In contrast, the *distributed control memory* has the control memory cells located in a very close neighbourhood of the programmable switches they control. Because of that, the memory cells can be implemented with

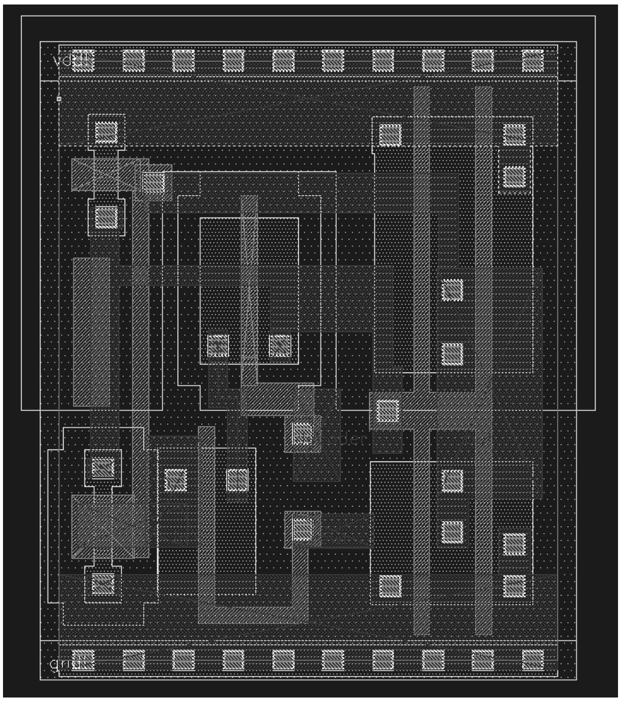


Figure 8.5. The layout of the configuration memory cell (the layout corresponds to the circuitry shown in Figure 8.4(a)).

minimum size transistors. This results in a very compact layout. An additional advantage is a direct availability of the complementary value of the configuration signal. Such signal is needed to control a complementary pass gate switch, for example. In a very compact realisation, the control memory cell can be integrated with a programmable switch, for example as it is shown in Figure 8.4(b). The disadvantage of the distributed control memory is that it results in very long wires between the address decoder and the control memory cells. The discussion on the implications of the control memory organisation on the layout is continued in Section 8.3.

8.1.2 Programmable interconnect design

The interconnect in a reconfigurable logic device is programmable. The interconnect wires are partitioned into segments of different lengths, with pairs of segments connected via programmable switches. The switches are controlled by the signals from the control memory. The programming of the control memory allows a realisation of an arbitrary connection scheme.

Because of their small size, NMOS pass transistors have been used typically for the implementation of programmable switches in FPGA devices. However, in deep-submicron technologies the use of pass transistors is less beneficial. The reason is a V_T voltage drop[1] in an NMOS transistor that accompanies a transfer of

[1] V_T denotes a threshold voltage of an MOS transistor.

a logical ‘1’ signal. This results in power dissipation in the logic circuitry that follows the transistor [108]. To overcome this effect, in some commercial FPGA devices the voltage on the gate of a pass transistor is boosted above the value of the supply voltage [14]. Alternatively, transistors with different values of the threshold voltage are used. Both methods require, however, adaptations of the standard manufacturing process and are thus less attractive for embedded FPGAs that should be fabricated in a standard CMOS process. Therefore, in today’s FPGAs the short-distance interconnect is implemented typically with programmable multiplexers, and the long-distance interconnect with 3-state buffers.

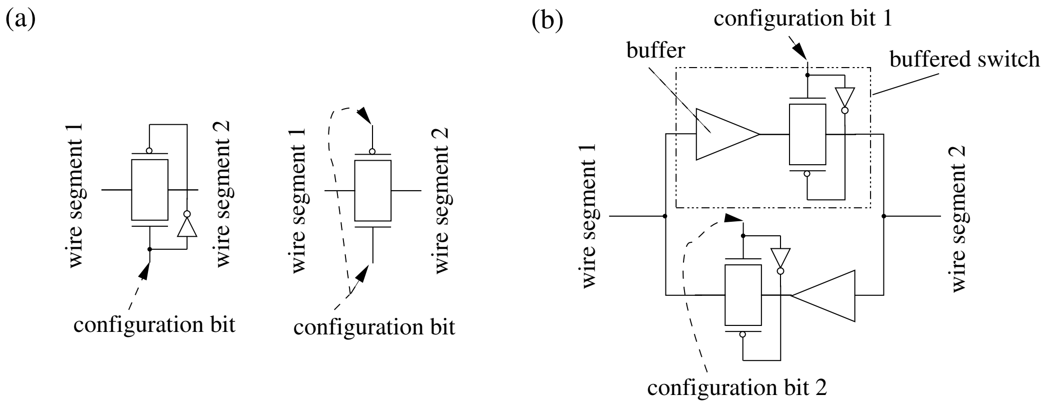


Figure 8.6. Basic components of a programmable interconnect architecture implemented using complementary pass gates: (a) programmable switches in two versions (for the centralised and distributed configuration memories), (b) programmable buffers.

To guarantee a high routing flexibility, the short-distance interconnect in the memory-oriented architecture is implemented using both programmable switches (for the length-1 interconnect segments) and multiplexers (for the direct interconnect). The longer wires (the length-4 interconnect segments) are implemented with bi-directional buffered switches. The programmable switches and buffered switches are implemented with complementary pass gates [110], as shown in Figures 8.6(a) and (b), respectively. The complementary pass gates help to achieve a better power-delay product than would be possible if single pass transistors were used. This effect is illustrated in Figure 8.7, where the comparison between programmable switches realised using single pass transistors and complementary pass gates is shown. The data in the figure have been obtained by modelling interconnect segments of different lengths (from length-1 to length-12) as the 10th-order RC networks with an appropriate load. For each segment length, the delay (in ns) and power-delay product (in ns·μW) have been found. The factor of difference in the power-delay product (for which the programmable interconnect was optimised) between both types of switches differs from 1.87 (length-1 segments) to 2.0 (length-12 segments).

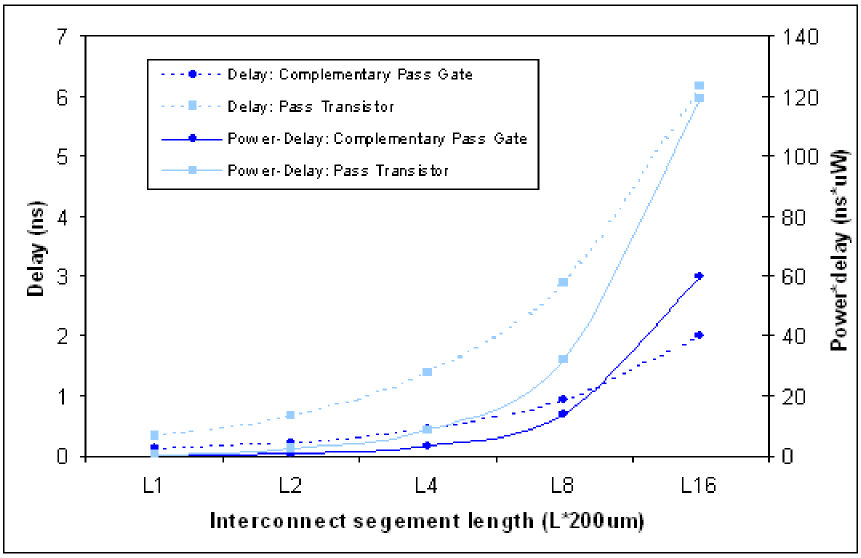


Figure 8.7. Comparison of programmable switches implemented using pass transistors and complementary pass-gates. The parameter L in the figure is the length of the interconnect segment, and $L = 1, 2, 4, 8, 16$.

8.2 Prototype chip

The silicon prototype of the proposed memory-oriented reconfigurable logic architecture has been implemented in the TSMC 0.13 μm CMOS process. The test chip (see Figure 8.8) includes a 4×4 array of the logic tiles that are surrounded by the input/output tiles. A global configuration decoder and a programmable clock generator are also included at the top level.

To reduce design effort, the layout of the chip was realised using a semi-custom rather than a full-custom design approach. A standard cell library extended with a few FPGA-specific cells, such as memory cells, dedicated multiplexers, programmable switches, was used. The layouts of the FPGA-specific cells were designed to have the same height as standard cells. This allowed the use of a standard ASIC design flow for the VLSI implementation of the chip.

The prototype chip was realised following the modular design concept. This means it was assembled using four types of basic building blocks, that is, a logic tile, an input/output tile, an input/output tile with routing and a corner tile. The netlists of the tiles were created using the schematic entry of Cadence Virtuoso, and were simulated on the transistor-level using Pstar simulator integrated in the Cadence Affirma environment. The verified netlist of each tile was placed and routed using Cadence Silicon Ensamble. The top level layout of the core was implemented in a hierarchical way using the layouts of basic tiles. The top-level connections between tiles were also laid out using Cadence Silicon Ensamble. The top-level layout of the reconfigurable logic core was integrated with the input/output (I/O) pads connecting to the input/output pins of the core. In total, 76 pads were placed

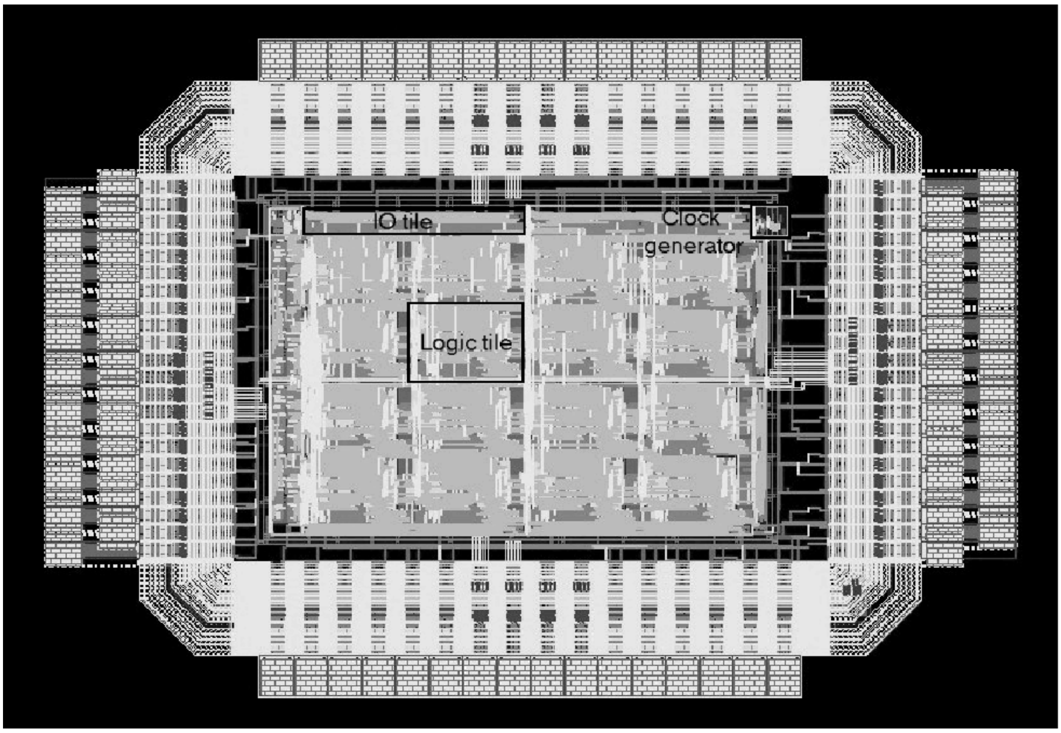


Figure 8.8. Layout of the test chip of the memory-oriented reconfigurable logic device.

on the chip: 32 I/O pads for data ports, 28 input pads for the configuration ports (configuration signals, control signals and decoder signals), 8 power and ground supply pads for the core, 8 power and ground supply pads for the I/O pads themselves[2], and one output pad for testing the accuracy of the internally generated clock signal.

Design	4×4 array, memory-oriented architecture
Technology	TSMC 0.13 μm CMOS
Interconnect	6 metal layers, copper
Power supply	1.2 V (core) + 3.3 V I/O pads
Logic block area	190× 111 μm^2
Logic block delay	$\sim$ 1.5 ns
Logic tile area (original)	227× 134 μm^2
Logic tile area (improved)	175× 106 μm^2
Configuration bits/logic tile	317
Test chip area (inc. I/O pads)	2.0× 1.35 mm^2

Table 8.1. The characteristics of the prototype chip. Two versions of the logic tile, that is, an original and improved, are indicated.

The characteristics of the prototype chip are summarised in Table 8.1. The table lists dimensions of two different versions of the logic tile: an original logic tile

[2] The I/O pads require a different supply voltage (i.e. 3.3 V) than the core itself.

(see Figure 8.9(a)) that has been used in the prototype chip implementation, and the improved logic tile (see Figure 8.9(b)) that has been realised later. The silicon area of the improved logic tile has been reduced by a factor 1.6 compared to the area of the original logic tile. This has been achieved with a little extra effort by:

- changing the centralised configuration memory (i.e. the LUT and control memories) into the distributed configuration memory (see the discussion in Section 8.1.1),
- integrating the layout of the switch block and connection blocks with the logic tile layout,
- simple optimisations of the logic structure (e.g. replacing a chain of logic gates by a simple gate of the equivalent functionality).

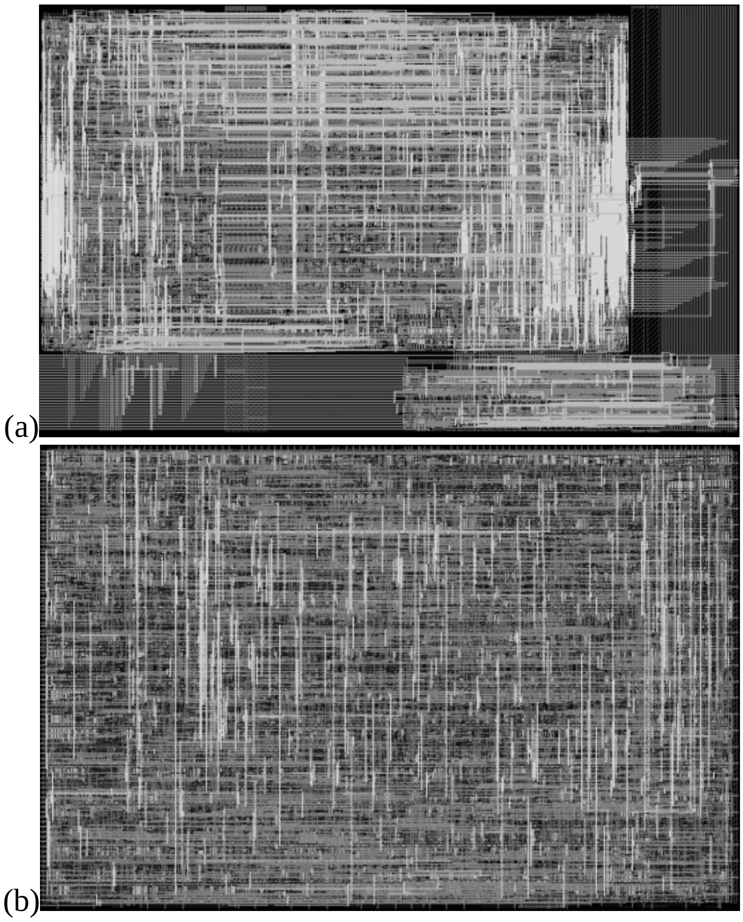


Figure 8.9. Two implementations of the logic tile: (a) original and (b) improved (distributed LUT and control memories, integrated switch block). The relative dimensions of the tiles are not preserved.

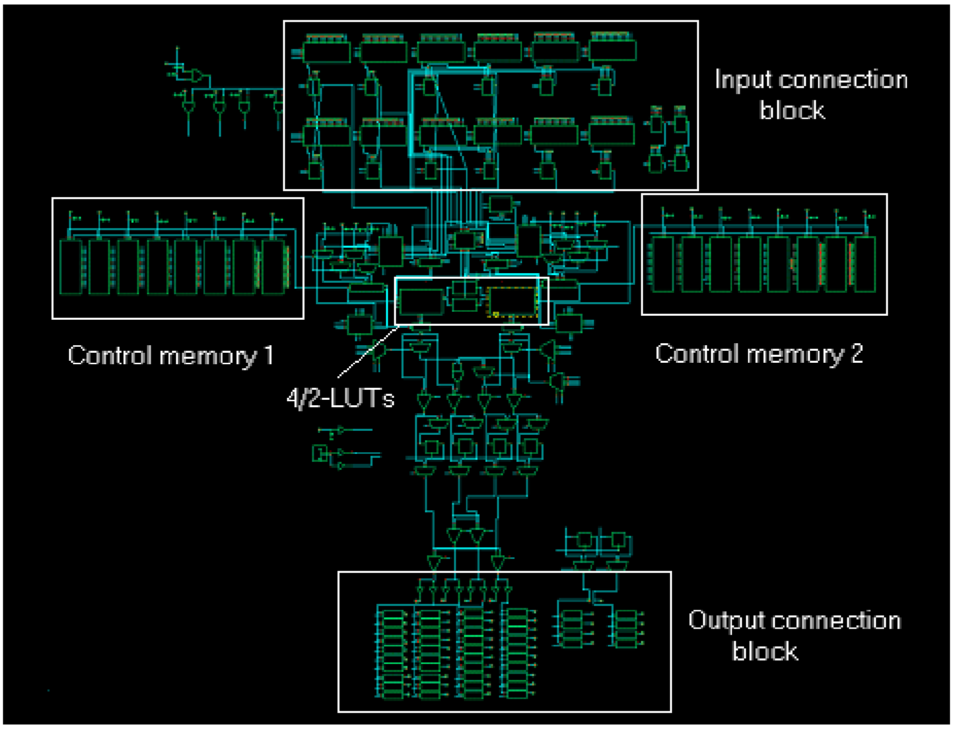


Figure 8.10. Schematic of the implemented logic block (the connection blocks are included).

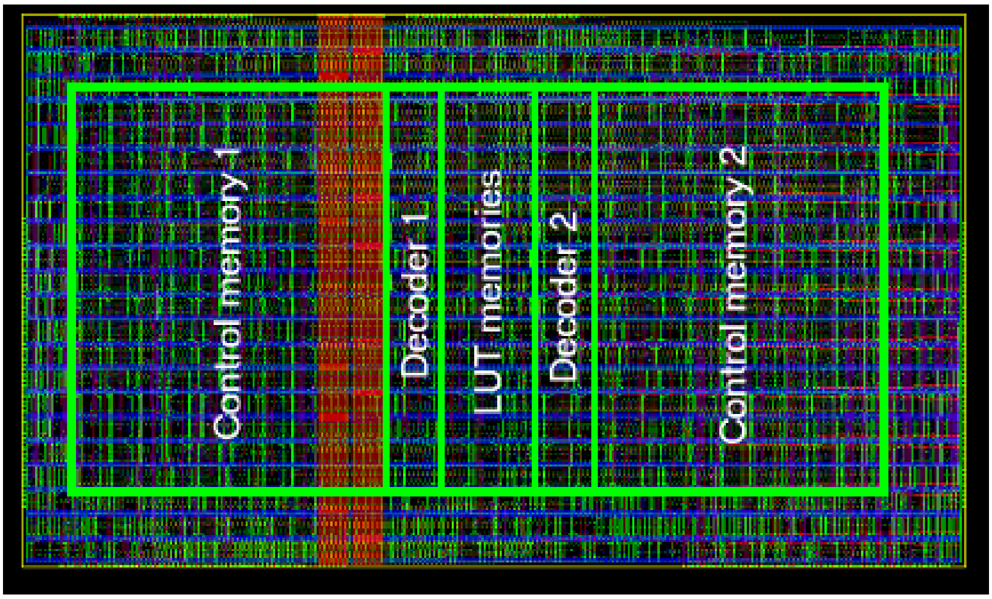


Figure 8.11. The logic block part (with connection blocks) of the original logic tile layout (compare Figure 8.9(a)). The centralised layout of the LUT and control memories implemented with standard cells is shown. For the sake of clarity, the metal layers on top of the logic block have been hidden.

8.3 Cost comparison

To determine the quality of the proposed memory-oriented reconfigurable logic architecture, we compare it with the architectures of two state-of-the-art commercial FPGAs, namely Xilinx Virtex-E [118] and Altera APEX 20KE [4]. The logic tile area is chosen as the cost metric in our comparison since it captures the information about the cost of logic and interconnect resources. Furthermore, as we explained in Chapter 3, the dimensions of the logic tile influence other implementation parameters of the architecture, such as performance and power consumption.

The layout areas A_T (in μm^2) of the logic tiles in the Xilinx and Altera devices have been obtained by die measurements, the results of which have been published in [13]. The layout area of the logic tile in the proposed memory-oriented eFPGA is the area of the improved logic tile, which dimensions are given in Table 8.1. The information about the logic tile areas, the technology in which the tiles were fabricated, the number of utilised metal layers (ML), and the implementation style is provided in Table 8.2. For a fair comparison, different target technologies and the different granularity of the compared logic tiles must be taken into account. Therefore, Table 8.2 also lists the normalised area A_{Tn} of the logic tiles. The normalised area A_{Tn} is the area of a given logic tile after its scaling to a 0.13 μm CMOS process, in which the logic tile of our eFPGA has been implemented. The normalised area A_{Tn} is calculated based on Equation 8.1. The factor η in Equation 8.1 is the granularity scaling factor, which expresses the granularity of the logic tiles in commercial FPGAs in terms of the granularity of the logic tile in our eFPGA. The granularity relates to the functional capacity of the FPGA logic tile (logic block) and is given in the number of its 4-LUT equivalents. The values of the factor η for the compared FPGA devices are listed in Table 8.2.

$$A_{Tn} = \left(\frac{0.13}{Technology}\right)^2 \cdot A_T \cdot \eta, \tag{8.1}$$

Logic tile type	A_T [μm^2]	Technology	η	A_{Tn} [μm^2]
Xilinx Virtex-E	35462	0.18 μm, 8 ML, full-custom layout	1.0	18497
Altera APEX 20KE[†]	63161	0.18 μm, 8 ML, full-custom layout	0.4	13178
Proposed eFPGA	18550	0.13 μm, 6 ML, standard-cell layout	1.0	18550

Table 8.2. Silicon area comparison of the logic tiles in two commercial FPGAs and the proposed memory-oriented eFPGA. ([†]The logic tile of the Altera APEX 20KE device has no memory mapping capabilities.)

The results from Table 8.2 show that the logic tile of the proposed memory-oriented eFPGA is only 0.03% and 41% larger than the logic tiles of the Xilinx

Virtex-E and Altera APEX 20KE FPGA devices, respectively. Three primary reasons for such a difference are:

- a full-custom implementation of the commercial devices versus a standard-cell-based implementation of the proposed device,
- the use of more metal layers in the commercial devices than in the proposed device (i.e. eight metal layers versus six metal layers),
- the circuit-level optimised structures of the commercial devices versus a straightforward implementation of the proposed device.

Note, that the relatively large area difference with respect to the Altera device is caused by a limited functionality of this device, that is, no support for mapping data memories in the logic block.

In [78], it has been shown that a standard-cell-based implementation of a FPGA logic tile yields a 42% larger silicon area than its full-custom implementation. Exploiting this fact and taking into account the above-mentioned aspects, we conclude that if a similar implementation method was used, the proposed memory-oriented eFPGA device would be superior to the Xilinx Virtex-E and Altera APEX 20KE FPGA devices. This means that the logic tile of our eFPGA would provide a similar functionality at the *lower cost*.

8.4 Conclusions

In this chapter we discussed the VLSI implementation of the memory-oriented embedded FPGA. We focused first on the implementation aspects of the LUT and control memories in such an FPGA. In particular, we explained differences between the conventional LUT and SRAM implementations and our memory-like LUT implementation. We also discussed the implementation of the programmable interconnect. We showed that the use of complementary pass gates as programmable switches in deep-submicron FPGAs yields better energy-delay product than is possible to achieve using single pass transistors. Finally, we presented the prototype chip of the memory-oriented eFPGA fabricated in a 0.13 μm CMOS process. We showed that after scaling to the same technology and taking into account the difference in the granularity, the current, standard-cell-based implementation of our eFPGA logic tile is only 0.03% and 41% larger than the full-custom implementations of the logic tiles in the Xilinx Virtex-E and Altera APEX 20KE FPGA devices, respectively. We also mentioned the reasons that make us convinced that if a similar implementation approach was followed, the architecture of the proposed eFPGA would be superior to the architectures of the compared commercial devices. That is, it would offer a similar functionality at the lower cost.

Chapter 9

CONCLUSIONS

In this thesis, the problem of designing cost-efficient reconfigurable logic cores for embedded systems-on-a-chip (SoC) has been considered. The purpose of the on-chip integration of reconfigurable logic is the increase of the computational efficiency and flexibility (programmability) of a target IC, and consequently the improvement of the system characetristics and the extension of the life-time of a final product. The cost aspect is crucial for embedded systems in consumer electronics. Therefore, the intrinsic area, performance and power consumption overhead of reconfigurable logic compared to the hard-wired logic of application-specific ICs must be reduced to make the SoC integration attractive.

To understand the reasons for the high area overhead of reconfigurable logic several generations of commercial FPGA devices were analysed. We explained that the changes in their architectures were driven by the demand for increased efficiency enabling an implementation of more complex functions. We discussed fundamental trade-offs in the architecture of FPGA devices and showed that different types of logic impose partly conflicting requirements on the FPGA structure. Finally, we also showed that the high intrinsic cost of FPGAs, which is dominated by the routing, is caused by their general-purpose character.

To solve this problem, we proposed the concept of application domain specialisation of reconfigurable logic. According to this concept, the cost reduction can be achieved if a reconfigurable logic architecture is optimised towards requirements of processing kernels from a target application domain instead of being made fully general-purpose. We suggested three parameters for the characterisation of processing kernels. They are: (dominant) type of processing, word-size and Rent exponent. The first two parameters characterise logic requirements, while the latter characterises the interconnect requirements. Assuming the type of processing as a primary characteristic and applying it to a reasonably large set of processing kernels from different application domains, we classified such kernels as being either data-path-oriented, random-logic-oriented, or memory-oriented. This leads to three corresponding classes at the implementation level. We assumed that, though being optimised to target application domains, domain-oriented architectures must also allow the mapping of other types of functions.

The concept of application-domain-specialisation of reconfigurable logic was generalised in the template idea. The template is a parametrised model of a reconfigurable logic core. The template facilitates architecture exploration, the VLSI implementation of template instances (netlist and layout generation) and the implementation of their mapping tools (architecture modelling for computer-aided-design). In consequence, the template reduces the overall design complexity and allows an easy creation of instances.

The proposed architecture template is hierarchical, with five levels of hierarchy (in rising order): a logic element, a processing element, a logic block, a tile, and an array. By setting the template parameters according to the application-domain-specific values, domain-oriented architecture instances of the template can be derived.

Three instances of the architecture template were described in detail. The first, a data-path-oriented architecture targets applications with a substantial amount of data-path logic. The reduction in cost was achieved by optimising the architecture for arithmetic using the proposed inversion-based folding type I. As a result, a basic arithmetic operation, that is a 1-bit binary addition, could be implemented using only a 2-input look-up table (2-LUT) enhanced with dedicated carry logic and a controlled inversion element. The logic block of the data-path-oriented architecture was built of four such LUTs and of a network of 2:1 multiplexers combining LUT outputs. Two primary functional modes of the architecture were enabled in this way. In the data-path mode, nibble-level (4-bit) processing is possible and the logic block generates a multi-bit result. In the random logic mode, a Boolean function with up to four inputs can be implemented and a 1-bit result is produced. The number of LUT memory bits that are needed to implement data-path functions is four times lower than the number of LUT bits that are required in state-of-the-art FPGA architectures. Although the logic block of the data-path-oriented architecture requires more ports than a 4-LUT-based logic block of a typical FPGA architecture, we showed that both architectures have a similar cost of routing. This was guaranteed by the specific realisation of the connection and selection blocks in our architecture. Use was made of the fact that in the data-path mode more routing tracks and lower connection flexibility, and in the random logic mode less routing tracks but higher connection flexibility are required.

The data-path-oriented architecture could be optimised further by applying configuration bit sharing. This is possible because of the bit-sliced structure of data-paths and the bus-like way of routing data signals. In the modified data-path-oriented architecture, the reduction in the number of configuration bits of logic and routing resources by a factor of four compared to the proposed data-path-oriented architecture, and by a factor of 16 compared to traditional FPGAs, was achieved.

The second, random-logic-oriented architecture was optimised for applications dominated by random logic. The implementation cost of this architecture, compared to the implementation cost of general-purpose FPGAs, was reduced by lowering the number of input ports of the logic block. Since each logic block port has a fixed area contribution, this resulted in the reduction of the routing resource cost. The lowered number of ports of the logic block has only a slight impact on the mapping capabilities of the architecture, and was possible because a large percentage of Boolean functions usually share some of their inputs. The logic block of the random-logic-oriented architecture was designed as a cluster of four 4-LUTs, with each 4-LUT implemented by means of two 3-LUTs. Such an implementation facilitates the mapping of both random logic and arithmetic circuits.

Finally, the third, memory-oriented architecture aims at the implementation of applications with storage requirements and some amount of data-path and random logic. Similarly to the random-logic-oriented architecture, the routing resource cost was reduced by lowering the number of input ports of the logic block. To enable such a cost reduction while guaranteeing a sufficient degree of flexibility, the logic block of the memory-oriented architecture was implemented using two 4/2-LUTs. The 4/2-LUT is a novel 4-input/2-output look-up table that is enhanced with control logic to support the memory functionality, and with controlled inversion elements at the inputs and outputs to support the arithmetic functionality. The arithmetic functionality was implemented using the proposed inversion-based folding type II. The key advantage of the 4/2-LUT is that it requires two times less routing resources (because of the reduced port number) than the look-up tables with similar functionality of state-of-the-art FPGAs. Furthermore, it also has two times less LUT memory bits than state-of-the-art multi-output look-up tables with memory mapping capabilities.

To evaluate the proposed ideas, three comparison methods were used. In the first method, the mapping cost *MC* of a given benchmark function, which is the total area of the implementation, was assumed as the primary cost metric. The method was used for a comparison of architectures of which the VLSI implementation area was known and for which mapping tools were available. Consequently, we applied this method to compare the proposed data-path-oriented architecture with two commercial FPGAs, that is Xilinx Virtex-E and Altera APEX 20KE. Using a set of 15 industrial designs of a different type and complexity we showed that a standard-cell implementation of our data-path-oriented architecture and an unoptimised mapping flow allow on average a 17% reduction in cost compared to a full-custom realisation of the Xilinx Virtex-E FPGA. When only data-path-dominated designs are considered, our architecture enables a 34% reduction in cost. With the same assumptions, our architecture results on average in a 8% cost overhead compared to the Altera APEX 20KE FPGA, and allows a 37% reduction in cost if only the data-path-dominated designs are considered.

The second comparison method was also based on the VLSI-implementation-related cost metrics such as the data-path mapping cost MC_{DP} and random logic mapping cost MC_{RL}. This method was applied, however, only to those architectures for which dedicated mapping tools were not available. The standard-cell-based implementation of the modified data-path-oriented architecture turned out to have 1.36 times higher data-path mapping cost than the smallest data-path-optimised and full-custom implemented FPGA from academia, that is CFPA.

In the third comparison method, a simple area model was used to estimate the logic and routing cost of reconfigurable architectures. In this model, the logic cost was estimated based on the number of LUT memory bits N_{lmb}, while the routing cost was estimated using the weighted logic block pin number P_w. This comparison method was applied to the proposed random-logic-oriented architecture, for which neither implementation data nor dedicated mapping tools were available. The architecture was compared with three state-of-the-art commercial FPGA devices, that is Xilinx Virtex II, Altera Stratix, and Atmel AT40K. Using such a comparison method we showed that at comparable logic cost, the routing cost of commercial architectures is over 1.6 higher than the routing cost of the proposed architecture.

The memory-oriented architecture was compared using the logic tile area obtained from a prototype chip, which was implemented in a 0.13 μm CMOS process. The logic tile of the memory-oriented architecture was compared with the logic tile of a Xilinx Virtex-E FPGA device. We showed that the functionality equivalent to that of the full-custom implemented Xilinx Virtex-E FPGA can be achieved at comparable cost in silicon area using a standard-cell-based implementation approach.

Future work

So far we proved that application domain specialisation allows a cost reduction of reconfigurable logic. This fact makes embedded domain-oriented reconfigurable logic a viable solution for designing cost-efficient system-on-chip ICs.

The gained knowledge and the obtained results were transfered to Philips Research Eindhoven, The Netherlands and the initiated research is continued as part of the Philips Research programme.

Still, a number of issues require further investigation. One of them is a full-custom realisation of the domain-oriented architectures, which may show an even more clear benefit of the proposed solution. Next, dedicated mapping tools that fully exploit all properties of the proposed architectures should be developed to further improve mapping results. Finally, the integration of our embedded reconfigurable logic solutions in the context of a real system-on-a-chip must be considered to evaluate their benefits as part of a commercial product.

Bibliography

[1] Actel Corporation. *VariCore^{TM} Embedded Programmable Gate Array Core (EPGA^{TM}) 0.18 µm Family. Data sheet*, December 2001.

[2] M. Agarwala and P.T. Balsara. An Architecture for a DSP Field-Programmable Gate Array. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(1):136–141, March 1995.

[3] Elias Ahmed and Jonathan Rose. The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 3–12, February 2000.

[4] Altera. *APEX Programmable Logic Device Family. Data sheet*. Altera, 2000.

[5] Altera. *FLEX 10KE Programmable Logic Device Family. Data sheet*. Altera, 2000.

[6] Altera. *Stratix Programmable Logic Device Family. Data sheet*. Altera, 2002.

[7] Altera Corporation. *Excalibur Devices. Hardware Reference Manual*, November 2002.

[8] Atmel. *5K-50K Gate FPGA with DSP Optimized Core Cell and Distributed FreeRAM. Summary. Data sheet*. Atmel, 1999.

[9] Atmel Corporation. *AT94 Series Field Programmable System Level Integrated Circuit. Data sheet*, June 2002.

[10] G. Baccarani, M. R. Wordeman, and R. H. Dennard. Generalized scaling theory and its application to a 1/4 micrometer MOSFET design. *IEEE Transactions on Electronic Devices*, ED-31:452–462, April 1984.

[11] Marc Baker. Design Migration from XC2000/XC3000 to XC5200. Application Note XAPP 061, Version 2.1, Xilinx, September 1997.

[12] Reinaldo A. Bergamaschi and John Cohn. The A to Z of SoCs. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, November 10–14 2002.

[13] Vaughn Betz and Jonathan Rose. Automatic Generation of FPGA Routing Architectures from High-Level Descriptions. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, February 2000.

[14] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[15] Jan C. Bioch and Toshihide Ibaraki. Decompositions of Positive Self-Dual Boolean Functions. *Discrete Mathematics*, 140:23–46, 1995.

[16] Fred Boekhorst. Ambient intelligence, the next paradigm for consumer electronics. How it will affect silicon? In *Proceedings of 2002 IEEE International Solid-State Circuits Conference*, volume 1, pages 28–31, February 2002.

[17] Ivo Bolsens. Challenges and Opportunities of FPGA Platforms. In *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, pages 391–392, September 2002.

[18] Frank M. Brown. *Boolean Reasoning. The Logic of Boolean Equations*. Kluwer Academic Publishers, 1990.

[19] Stephen.D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, 1992.

[20] Randal E. Bryant, Kwang-Ting Cheng, Andrew B. Khang, Kurt Kreutzer, Wojciech Maly, Richard Newton, Lawrence Pileggi, Jan M. Rabaey, and Alberto Sangiovanni-Vincentelli. Limitations and Challenges of Computer-Aided Design Technology for CMOS VLSI. *Proceedings of the IEEE*, 89:341–365, March 2001.

[21] Cadence Design Systems. *Envisia Silicon Ensemble Place and Route. Training Manual*, 5.3 edition, October 2000.

[22] Cadence Design Systems. *HDL Modeling for BuildGates Synthesis. User Guide*, 5.0 edition, May 2002.

[23] Cadence Design Systems. *Tutorial for Cadence BuildGates Synthesis and Cadence PKS*, 5.0 edition, May 2002.

[24] William S. Carter, Khue Duong, Ross H. Freeman, Hung-Cheng Hsieh, Jason Y. Ja, John E. Mahoney, Luan T. Ngo, and Shelly L. Sze. A User Programmable Reconfigurable Logic Array. In *IEEE 1986 Proceedings of Custom Integrated Circuits Conference*, pages 233–236, May 1986.

[25] Chameleon Systems. *CS2112 Reconfigurable Communication Processor*, 2001.

[26] Don Cherepacha and David Lewis. DP-FPGA: An FPGA Architecture Optimized for Datapaths. *VLSI Design*, 4(4):329–343, 1996.

[27] Peter Clarke. Keynoter presents an exercise in imagination. EE Times, Semiconductor Business News, March 5, 2003.

[28] Theo A. C. M. Classen. High Speed: Not the Only Way to Exploit the Intrinsic Computational Power of Silicon. In *Proceedings of the 1999 IEEE International Solid-State Circuits Conference*, pages 22–25, January 1999.

[29] K. Compton and S. Hauck. Totem: Custom Reconfigurable Array Generation. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 2001.

[30] Katherine Compton and Scott Hauck. Automatic Design of Configurable ASICs. submitted to *IEEE Transactions on Very Large Scale Integration Systems*.

[31] Katherine Compton, Akshay Sharma, Shawn Philips, and Scott Hauck. Flexible Routing Architecture Generation for Domain-Specific Reconfigurable Subsystems. In *Proceedings of the Field Programmable Logic and Applications Conference*, August 2002.

[32] Jason Cong and Yuzheng-Ding. FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs. *IEEE Transactions on Computer Aided Design*, 13(1):1–12, January 1994.

[33] Alexander Danilin. Archimed. Philips Research internal reconfigurable architecture generator for FPGAs, 2003.

[34] Alexander Danilin. Pythagor. Philips Research internal placement and routing tool for FPGAs, 2003.

[35] Hugo de Man. On Nanoscale Integration and Gigascale Complexity in the Post.com World. Keynote speech, *Design, Automation and Test in Europe Conference*, March 2002.

[36] Bernardo de Oliveira Kastrup Pereira. *Automatic Synthesis of Reconfigurable Instruction Set Accelerators*. PhD thesis, Eindhoven University of Technology, The Netherlands, May 2001.

[37] Andre DeHon. *Reconfigurable Architectures for General-Purpose Computing*. PhD thesis, MIT, Artificial Intelligence Laboratory, 1996.

[38] Andre DeHon. Balancing Interconnect and Computation in Reconfigurable Array (or why you don't really want 100% LUT utilization). In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 69–78, February 1999.

[39] Andre DeHon and John Wawrzynek. Reconfigurable Computing: What, Why, and Implications for Design Automation. In *Proceedings of the 1999 Design Automation Conference*, pages 610–615, June 1999.

[40] R. H. Dennard, F. H. Gaensslen, H. N. Yu, V. L. Rideout, F. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFETs with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, SC-9:256–258, May 1974.

[41] Brian Dipert. Silicon Segementation. *EDN*, pages 57–65, September 2003.

[42] Carl Ebeling, D.C. Cronquist, and P. Franklin. RaPiD – Reconfigurable Pipelined Datapath. In *Proceedings of the 6th Annual Workshop of Field Programmable Logic and Applications*, pages 126–135, August 1996.

[43] Elixent Ltd. *Changing the electronic landscape. The Reconfigurable Algorithm Processor. White paper*, 2001.

[44] Alberto Ferrari and Alberto Sangiovanni-Vincentelli. System Design: Traditional Concepts and New Paradigms. In *Proceedings of the 1999 International Conference on Computer Design*, October 1999.

[45] Brian Fuller. Moore's Law takes it on the chin. *EE Times*, March 10, 2003.

[46] Daniel G. Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong. *Specification and Design of Embedded Systems*. PTR Printice Hall, 1994.

[47] Varghese George. *Low Energy Field-Programmable Gate Array*. PhD thesis, University of California, Berkeley, 2000.

[48] Richard Goering. Platform-based design: A choice, not a panacea. *EE Times*, September 12, 2002.

[49] James Goodman and Anantha P. Chandrakasan. An Energy-Efficient Reconfigurable Public-Key Cryptography Processor. *IEEE Journal of Solid-State Circuits*, 36(11):1808–1820, November 2001.

[50] Paul Graham and Brent Nelson. FPGA-based Sonar Processing. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 201–208, February 1998.

[51] Lars Hagen, Andrew Khang, Fadi J. Kurdahi, and Champaka Ramachandran. On Intrinsic Rent Parameter and Spectra-Based Partitioning Methods. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(1):27–37, 1994.

[52] J. He and J. Rose. Advantages of Heterogeneous Logic Block Architectures for FPGAs. In *Proceedings of the IEEE Custom Integrated Circuits Conference*. IEEE, May 1993.

[53] Brian Von Herzen. Signal Processing at 250 MHz Using High-Performance FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6(2):238–246, June 1998.

[54] Dwight Hill and Nam-Sung Woo. The Benefits of Flexibility in Look-up Table FPGAs. In *Proceedings of the Oxford 1991 International Workshop on Field Programmable Logic and Applications*, pages 127–136, 1991.

[55] K. Hwang. *Computer Arithmetic: Principles, Architecture, and Design*. John Wiley & Sons, 1979.

[56] ITRS. The International Technology Roadmap for Semiconductors: 1999. Design, 1999.

[57] Hiroshi Iwai. CMOS Technology - Year 2010 and Beyond. *IEEE Journal of Solid-State Circuits*, 43:357–366, March 1999.

[58] Frank Jennings. End of Moore's law. http://www.zdnetindia.com/techzone/trends/stories/381.html, August 25, 2000.

[59] Lech Jozwiak. Quality-driven design in the system-on-a-chip era: Why and how? *Journal of Systems Architecture*, 47:201–224, 2001.

[60] Alireza Kaviani, Daniel Vranesic, and Stephen Brown. Computational Field Programmable Architecture. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 261–264. IEEE, May 1999.

[61] Alireza S. Kaviani. *Novel Architectures and Synthesis Methods for High Capacity Field Programmable Devices*. PhD thesis, University of Toronto, Department of Electrical Engineering, January 1999.

[62] Kurt Keutzer, Sharad Malik, Richard Newton, Jan Rabaey, and Alberto Sangiovanni-Vincentelli. System-Level Design: Orthogonalization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19:1523–1543, December 2000.

[63] Hue-Sung Kim, Arun S. Somani, and Akhilesh Tyagi. A Reconfigurable Multi-function Computing Cache Architecture. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 2000.

[64] Bernard S. Landman and Roy L. Russo. On a Pin Versus Block Relationship for Partitions of Logic Graphs. *IEEE Transactions on Computers*, C-20(12):1469–1479, December 1971.

[65] Lattice Semiconductor Corporation. *ORCA Series 3C and 3T FPGAs Data Sheet*, November 2003.

[66] Dave Lautzenheiser and Micke Wersall. Targeting the high ground between ASICs and FPGAs. *Electronic Engineering*, pages 73–82, November 1999.

[67] Leopard Logic Inc. *HyperBlox^{TM} Field Programmable Embedded FPGA Cores. Product Brief*, 2002.

[68] George Leopold. Embedded FPGAs seen surging. *EE Times*, January 13, 2003.

[69] Alan Marshall, Tony Stansfield, Igor Kostarnov, Jean Vuillemin, and Brad Hutchings. A Reconfigurable Arithmetic Array for Multimedia Applications. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, February 1999.

[70] Ken Martin. *Digital Integrated Circuit Design*. Oxford University Press, 2000.

[71] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley Longman Publishing Co., Inc., 1979.

[72] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., 1994.

[73] N.L. Miller and S.F. Quigley. A Novel Field Programmable Gate Array Architecture for High Speed Processing. In *Proceedings of Field-Programmable Logic and Applications Conference*, pages 386–390, September 1997.

[74] Gordon E. Moore. Progress in digital integrated circuits. *IEDM Technology Digest*, page 11, December 1975.

[75] H. Ozaktas. Paradigms of Connectivity for Computer Circuits and Networks. *Optical Engineering*, 31:1563–1567, 1992.

[76] PACT Informationstechnologie GmbH. *The XPP White Paper. A Technical Perspective*, March 2002.

[77] Ketan Padalia, Ryan Fung, Marc Bourgeault, Aaron Egier, and Jonathan Rose. Automatic Transistor and Physical Design for FPGA Tiles from an Architectural Specification. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 164–173, February 2003.

[78] Shawn Philips and Scott Hauck. Automatic Layout of Domain-Specific Reconfigurable Subsystems for System-on-a-Chip. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 2002.

[79] Philips Semiconductors. *CMOS12 Library Technology Reference Manual*, revision 1.4 edition, September 2002.

[80] Jan Rabaey. *Digital Integrated Circuits. A Design Perspective*. Prentice Hall, 1996.

[81] Jan Rabaey. Silicon Platforms for the Next Generation Wireless Systems – What Role does Reconfigurable Hardware Play? In *Proceedings of the International Field Programmable Logic and Application Conference*, Lecture Notes in Computer Science, August 2000.

[82] Jan M. Rabaey. Reconfigurable Computing: the Solution to Low Power Programmable DSP. In *Proceedings of the 1997 International Conference on Acoustics, Speech and Signal Processing*, volume 1, pages 275–278, April 1997.

[83] Jan. M. Rabaey. Wireless Beyond the Third Generation – Facing the Energy Challenge. In *Proceedings of the ACM International Symposium on Low Power Electronic Design*, August 2001.

[84] Jan M. Rabaey, M. Josie Ammer, Julio L. da Silva Jr., and Shad Roundy. PicoRadio Supports Ad Hoc Ultra-Low Power Wireless Networking. In *IEEE Computer Magazine*, pages 42–48. IEEE, July 2000.

[85] A. Roopchansingh and J. Rose. Nearest Neighbour Interconnect Architecture in Deep-Submicron FPGAs. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 59–62, May 2002.

[86] J. Rose, R.J. Francis, P. Chow, and D. Lewis. The Effect of Logic Block Complexity on Area of Programmable Gate Arrays. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 5.3.1–5.3.5, May 1989.

[87] Jonathan Rose, Robert J. Francis, David Lewis, and Paul Chow. Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency. *IEEE Journal of Solid-State Circuits*, 25:1215–1225, October 1990.

[88] Engel Roza. Systems-on-chip: what are the limits? *IEE Electronics and Communication Journal*, pages 249–255, December 2001.

[89] Milan Saini. Platform FPGAs Take on ASICs SoCs. *Xcell Journal*, 3:70–73, Summer 2002.

[90] Ellen M. Sentovitch, Kanwar J. Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. *SIS: A System for Sequential Circuit Synthesis*. Electronics Research Laboratory, University of California, Berkeley, May 1992. Memorandum No. UCB/ERL M92/41.

[91] ED&T/Analog Simulation. *Pstar User Guide for Pstar 3.8*. ED&T, Philips Electronics N.V., Eindhoven, The Netherlands, December 1998.

[92] Amit Singh and Malgorzata Marek-Sadowska. Efficient Circuit Clustering for Area and Power Reduction in FPGAs. In *Proceedings of ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pages 59–65, February 2002.

[93] Amit Singh and Malgorzata Marek-Sadowska. FPGA Interconnect Planning. In *Proceedings of the ACM Workshop on System Level Interconnect Prediction*, pages 23–30, April 2002.

[94] Satwant Singh, Jonathan Rose, Paul Chow, and Dawid Lewis. The Effect of Logic Block Architecture on FPGA Performance. *IEEE Journal of Solid-State Circuits*, 27:281–287, March 1992.

[95] Tony Stansfield. Wordlength as an Architectural Parameter for Reconfigurable Computing Devices. In *Proceedings of the Field-Programmable Logic and Applications Conference*, pages 667–676, September 2002.

[96] Dirk Stroobandt. *A Priori Wire Length Estimates for Digital Design*. Kluwer Academic Publishers, 2001.

[97] Synplicity, Inc. *Synplify Pro Reference Manual*, October 2001.

[98] Systolix. *Systolix PulseDSP core*. www.systolix.co.uk/techintro.html.

[99] Taiwan Semiconductor Manufacturing Co., LTD. *TSMC 0.13 μm Logic 1P8M Salicide 1.0V/2V/5V, 1.2V/2.5V, 1.0V/3.3V, 1.2V/3.3V Design Rule*, ta-10b2-4001, version 1.4 edition, June 2002.

[100] Anil Telikapalli. Virtex-II Pro FPGAs: The Platform for Programmable Systems Has Arrived. *Xcell journal*, 1(42):10–13, Spring 2002.

[101] Tensilica Inc. *Xtensa. Architecture and Performance. White paper*, September 2002.

[102] Triscend Corporation. *Triscend A7S. 32-bit Field Configurable System-on-Chip (CSoC). Report TCH305-0001-002*, August 2002.

[103] Triscend Corporation. *Triscend E5 Customizable Microcontroller Platform. Report TCH300-0001-001*, March 2003.

[104] Stephen H. Unger. *The Essence of Logic Circuits*. Prentice-Hall International Editions, 1989.

[105] University of Toronto, Canada. *VPR and T-VPack User's Manual (Version 4.30)*, March 2000.

[106] H. van Marck, Dirk Stroobandt, and Jan van Campenhout. Towards an Extension of Rent's Rule for Describing Local Variations in Interconnection Complexity. In *Proceedings of the 4th International Conference for Young Computer Scientists*, pages 136–141, 1995.

[107] Jef L. van Meerbergen. *Embedded Multimedia Systemen in Silicium*. Technische Universiteit Eindhoven, 2000–2001. College dictaat.

[108] Harry J.M. Veendrick. *Deep-Submicron CMOS ICs. From Basics to ASICs*. Kluwer BedrijfsInformati B. V., The Netherlands, 1 edition, 1998.

[109] Albert Wang. The Role of ASIP in Programmable Platforms . Presentation slides for Electronic Systems Design Seminar, UC Berkeley, November 2001.

[110] Neil H.E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison-Wesley Publishing Company, 1992.

[111] Stanley A. White. Application of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review. *IEEE ASSP Magazine*, 6(3):4–19, July 1989.

[112] Ron Wilson. Chip industry tackles escalating mask costs. EE Times, June 17, 2002.

[113] Xilinx. The Role of Distributed Arithmetic in FPGA-based Signal Processing. www.xilinx.com.

[114] Xilinx. *The Programmable Logic. Data Book*. Xilinx, Inc., San Jose, CA, September 1996.

[115] Xilinx. An Alternative Capacity Metric for LUT-Based FPGAs. Technical report, Xilinx, February 1 1997.

[116] Xilinx. *XC3000 Series Field Programmable Gate Arrays (XC3000A/L, XC3100A/L)*, version 3.1 edition, November 1998.

[117] Xilinx. *XC4000E and XC4000X Series Field Programmable Gate Arrays. Data sheet*. Xilinx, 1999.

[118] Xilinx. *Virtex-E 1.8V Field Programmable Gate Arrays. Data sheet*. Xilinx, 2000.

[119] Xilinx. *Virtex-II Pro Plaplatform FPGAs. Data sheet*. Xilinx, 2002.

[120] Saeyang Yang. *Logic Synthesis and Optimization Benchmarks User Guide*. Microelectronics Center of North Carolina, Research Triangle Park, NC 27709, USA, January 1991.

[121] Behrooz Zahiri. Structured ASICs: Opportunities and Challenges. In *Proceedings of the 21st International Conference on Computer Design*, pages 404–409, October 2003.

[122] Payman Zarkesh-Ha, Jeffrey A. Davis, and James D. Meindl. Prediction of Net-Length Distribution for Global Interconnects in a Heterogeneous System-on-a-Chip. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(6):649–659, December 2000.

[123] Reto Zimmermann. Lecture Notes on Computer Arithmetic: Principles, Architectures, and VLSI Design. Technical report, Swiss Federal Institute of Technology, Integrated Systems Laboratory, Zurich, Switzerland, March 1999. http://www.iis.ee.ethz.ch/zimmi/publications/comp_arith_notes.ps.gz.

[124] Paul S. Zuchowski, Christopher B. Reynolds, Richard J. Grupp, Shelly G. Davis, Brendan Cremen, and Bill Troxel. A Hybrid ASIC and FPGA Architecture. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pages 187–194, November 2002.

PERSONAL CONTRIBUTION

Publications:

- B. Kastrup, K. Nowak, J. van Meerbergen, ”Seeking (the Right) Problems for the Solutions of Reconfigurable Logic”, in *Proceedings of Field Programmable Logic and Applications Conference*, Glasgow, Scotland, September 1999.

- K. Leijten-Nowak, J.L. van Meerbergen, ”Applying the Adder Inverting Property in the Design of Cost-Efficient Reconfigurable Logic”, in *Proceedings of the 44the Midwest Symposium on Circuits and Systems*, Dayton, Ohio, USA, August 2001.

- P. Poplavko, K. Leijten-Nowak, J.L. van Meerbergen, ”Placement Algorithms for Datapath-oriented FPGAs”, in *Proceedings of ProRISC 2001*, Veldhoven, The Netherlands, November 2001.

- K. Leijten-Nowak, J.L. van Meerbergen, ”Embedded Reconfigurable Logic Core for DSP Applications”, in *Proceedings of Field Programmable Logic and Applications Conference*, Montpellier, France, September 2002.

- K. Leijten-Nowak, A. Katoch, ”Architecture and Implementation of an Embedded Reconfigurable Logic Core in CMOS 0.13 μm”, in *Proceedings of IEEE ASIC/SOC Conference*, Rochester, New York, USA, September 2002.

- K. Leijten-Nowak and J.L. van Meerbergen, ”An FPGA Architecture with Enhanced Datapath Functionality”, in *Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, California, USA, February 2003.

- B. Mesman, Q. Zhao, N. Busa, K. Leijten-Nowak, ”Instruction Set Application Tuning for DSP”, *Journal for Circuits, Systems and Computers*, 12(3), June 2003.

Philips patent applications:

- J.-P. Theis, K. Nowak, J. van Meerbergen, ”Programmable Logic Matrix with Extended Connections”, PHNL000309, Invention Disclosure ID603813 (”A Next Generation Reconfigurable Computing cell for FPGAs”), May 2000.

- K. Leijten-Nowak, ”Reconfigurable Logic Device”, PHNL010359, Invention Disclosure ID606907 (”Cost-efficiency Improvement for Look-up Table”), April 2001.

- A. Augustijn, K. Leijten-Nowak, ”Control Word Hoisting”, PHNL020396, Invention Disclosure ID607487 (”Control Word Hoisting”), June 2001.

- K. Leijten-Nowak, ”Configuration Memory Implementation for LUT-Based Devices”, PHNL020211, Invention Disclosure ID609183 (”Configuration Memory Implementation for LUT-Based Devices”), January 2002.

- K. Leijten-Nowak, ”Implementation of Wide Multiplexers in Reconfigurable Logic”, PHNL020212, Invention Disclosure ID609402 (”Implementation of Wide Multiplexers in Reconfigurable Logic”), February 2002.

- K. Leijten-Nowak and A. Katoch, ”IC with logic tiles and routing network”, PHNL020590, Invention Disclosure ID609454 (”Modular Reconfigurable Logic Architecture with Homogeneous Logic Tiles”), February 2002.

- K. Leijten-Nowak, ”Reconfigurable electronic device with dual port memory mode”, PHNL020827, Invention Disclosure ID609469 (”Distributed Dual-Port Memory Architecture for Reconfigurable Logic Devices”), February 2002;

- K. Leijten-Nowak, merged with PHNL010359, Invention Disclosure ID609539 (”Array Multiplier Implementation for Reconfigurable Logic Devices”), February 2002.

- K. Leijten-Nowak, ”Electronic device having data storage device”, PHNL020823, Invention Disclosure ID609559 (”Variable-Length Shift Register Implementation for Reconfigurable Logic Devices”), March 2002;

- K. Leijten-Nowak, ”Electronic circuit with array of programmable cells”, PHNL020632, Invention Disclosure ID609641 (”Mixed-Grain Reconfigurable Logic Cell for DSP Applications”), March 2002.

- K. Leijten-Nowak, ”Electronic circuit with array of programmable cells”, PHNL030186, Invention Disclosure ID 613098 (”Bit Sharing”), February 2003.

- K. Leijten-Nowak, ”Electronic circuit with array of programmable cells”, PHNL030188, Invention Disclosure ID 613099 (”Flexible Multiplexing”), February 2003.
- K. Leijten-Nowak, ”Electronic circuit with array of programmable cells”, PHNL030187, Invention Disclosure ID 613100 (”Carry dependance”), February 2003.
- K. Leijten-Nowak, ”Template-based Domain-specific Reconfigurable Logic”, PHNL031466, Invention disclosure ID 697975 (”Template-based Domain-specific Reconfigurable Logic”), December 2003.

SUMMARY

The rapid increase in the design time and manufacturing costs of integrated circuits (ICs) and the constant demand for higher computational efficiency (in MOPS/W) make traditional implementation methods based on application-specific logic and programmable processors no longer sufficient. Embedded reconfigurable logic appears an interesting alternative in the system-on-a-chip context. Unlike application-specific logic, it allows an implementation of system components which functionality can be changed after a device is fabricated, and which are characterised by a higher computational efficiency than programmable processors. This is achieved, however, at the cost of larger area, lower performance and higher power consumption compared to application-specific ICs.

This thesis concerns itself with the design of cost-efficient embedded reconfigurable logic using the concept of a template. The template is a generic, parametrised model of a reconfigurable logic core. The use of the template enables a fast architecture exploration, simplifies the process of the VLSI implementation of template instances, and supports the creation of retargetable mapping tools for them.

The architecture instances that are derived from the proposed template are application-domain-specific rather than general-purpose. The application domain specialisation means that both logic and interconnect of a reconfigurable logic architecture are tuned to the requirements of processing kernels from a target application domain. This reduces the cost overhead yet still guarantees the required degree of flexibility.

A method of characterising processing kernels from different application domains to find a suitable architecture template instance is proposed. Three parameters, that is (dominant) type of processing, word-size and Rent exponent, are proposed as main characteristics of processing kernels. Using a large set of kernels from different applications, three basic classes of reconfigurable logic architectures are derived. They are: data-path-oriented architectures, random-logic-oriented architectures and memory-oriented architectures. Three template instances that correspond to the derived architecture classes are described in detail. For each reconfigurable architecture instance, the implementation details of the logic block and

the programmable interconnect are given. Two methods of an efficient LUT-based implementation of binary addition are proposed and applied to the design of basic building blocks of two proposed template instances. A comparison of the proposed reconfigurable architectures with state-of-the-art Field Programmable Gate Arrays (FPGAs) is also presented. Using implementation-based and model-based comparison methods it is shown that the proposed domain-oriented reconfigurable logic is superior to the compared general-purpose commercial FPGA devices.

The VLSI implementation aspects and silicon prototype of a memory-oriented reconfigurable logic core are described in detail. The core was implemented using the proposed template-based modular design concept. It is shown that embedded reconfigurable logic with a functionality similar to that of commercial, full-custom implemented Xilinx Virtex-E FPGAs can be achieved using a standard-cell-based implementation at comparable cost in silicon area.

SAMENVATTING

De snelle toename in de ontwerptijd en produktiekosten van geïntegreerde schakelingen (ICs) en de constante vraag naar hogere rekenefficiëntie (in miljoen operaties per seconde per Watt, i.e. MOPS/W) maakt dat traditionele implementatiemethoden, gebaseerd op applicatie-specifieke logica and programmeerbare processoren niet langer afdoende zijn. Ingebedde herconfigureerde logica blijkt een interessant alternatief in de context van systemen op chip. In tegenstelling tot applicatie-specifieke logica, staat het implementaties van systeemcomponenten toe, waarvan de functionaliteit veranderd kan worden nadat een chip gefabriceerd is, en welke gekarakteriseerd worden door een hogere rekenefficiëntie dan programmeerbare processoren. Dit wordt echter bereikt tegen hogere oppervlaktekosten, een kleinere rekenkracht en een hoger vermogensverbruik vergeleken met applicatie-specifieke ICs.

Dit proefschrift behandelt het ontwerp van kosten-efficiënte ingebedde herconfigureerbare logica, gebruikmakend van het concept van een architectuursjabloon. Dit sjabloon is een generiek, parametriseerbaar model van een herconfigureerbare-logica kern. Het gebruik van het sjabloon maakt een snelle architectuurexploratie mogelijk, het vereenvoudigt het VLSI implementatieproces van sjablooninstanties en het ondersteunt de creatie van flexibele gereedschappen voor de afbeelding van applicaties op deze instanties.

De architectuurinstanties die worden afgeleid van het voorgestelde sjabloon zijn applicatie-domein-specifiek in plaats van gericht op volledig generieke toepassing. Specialisatie voor een bepaald applicatie domein betekent dat zowel logica als verbindingen van een herconfigureerbare logica architectuur afgestemd worden op de eisen van rekenintensieve algorithmen uit het applicatiedomein waarop men zich richt. Dit verlaagt de kosten, maar garandeert toch de vereiste graad van flexibiliteit.

Een methode wordt voorgesteld om algorithmen uit verschillende applicatiedomeinen te karakteriseren ten behoeve van het vinden van geschikte sjablooninstanties voor deze domeinen. Drie parameters, te weten het (dominante) type rekenkundige bewerkingen, de woordbreedte en Rent exponent, worden voorgesteld als de belangrijkste karakteristieken van rekenkundige algo-

rithmen. Gebruikmakend van een grote verzameling algorithmen uit verschillende applicaties worden drie basisklassen van herconfigureerbare logica architecturen afgeleid. Dit zijn: datapad-georiënteerde architecturen, willekeurige-logica-georiënteerde architecturen en geheugen-georiënteerde architecturen. Drie sjablooninstanties die overeenkomen met de afgeleide architectuurklassen worden in detail beschreven. Voor elk van deze instanties worden details van het logica blok en de programmeerbare verbindingen gegeven. Twee methoden voor een efficiënte LUT-gebaseerde implementatie van binaire optelling worden voorgesteld en toegepast op het ontwerp van basisbouwblokken van twee van de voorgestelde sjablooninstanties. Tevens wordt een vergelijking tussen de voorgestelde herconfigureerbare-logica-architectuur met ‘state-of-the-art’ Field Programmable Gate Arrays gepresenteerd. Met gebruikmaking van implementatie-gebaseerde en model-gebaseerde vergelijkingsmethoden wordt aangetoond dat de voorgestelde domein-georienteerde herconfigureerbare logica superieur is ten opzichte van de beschouwde generiek toepasbare FPGAs.

VLSI implementatie-aspecten en een silicium prototype van een memory-georiënteerde herconfigureerbare kern worden in detail beschreven. De kern werd geïmplementeerd met gebruikmaking van het sjabloon-gebaseerde modulaire ontwerpconcept. Er wordt aangetoond dat met een standaard-cell-gebaseerde implementatie van ingebedde herconfigureerbare logica een functionaliteit kan worden bereikt die vergelijkbaar is met die van commerciële, volledig geoptimaliseerde implementaties van Xilinx Virtex-E FPGAs, tegen vergelijkbare kosten in silicium oppervlak.

CURRICULUM VITAE

Katarzyna Leijten-Nowak was born on November 20, 1973 in Wrocław, Poland. From 1988 till 1993 she was attending the Technical Secondary School in Tarnów, Poland. She graduated *Summa Cum Laude* in 1993 receiving specialisation in computers and digital systems. In 1990 she was awarded an annual studentship of the Polish Ministry of Education and in 1991 a half-year studentship of the Polish Foundation for Young Talents.

In the period 1993–1998 she studied Electronics and Telecommunication at the Wrocław University of Technology, Wrocław, Poland. She graduated in October 1998 receiving specialisation in signal processing systems. Her graduation project on the low-power implementation of the FFT processor for OFDM systems was realised at Delft University of Technology, Delft, The Netherlands as part of the TEMPUS project. The work was carried out in cooperation with Philips Research Eindhoven. The implementation of a CORDIC processor developed during the project was used in Philips' Digital Audio Broadcasting (DAB) chip.

In autumn 1998, Katarzyna joined the Information and Communication Systems Group of Eindhoven University of Technology, Eindhoven, The Netherlands. In February 1999 she started her Ph.D. research. From June 1999 till April 2003, she carried out her research at the Embedded Systems Architectures on Silicon Group of Philips Research Eindhoven as part of the ARCADE cluster project. The work was supervised by prof.dr.ir. Jef L. van Meerbergen, prof.dr.ir. Ralph H.J.M. Otten and prof.Dr.-Ing. Jochen A.G. Jess. The results of the ARCADE project are currently used in the embedded FPGA project as part of the Philips research programme.

Since June 2003, Katarzyna is working at Philips Research Laboratories in Eindhoven, The Netherlands as a research scientist. She is a member of the Deep-Submicron Circuit Design Cluster of the Digital Design and Test Group. Her present research area covers embedded reconfigurable computing, ultra-low power design and statistical design.

NOTES

Notes

Stellingen

behorende bij het proefschrift

Template-Based Embedded Reconfigurable Computing

van

Katarzyna Leijten-Nowak

1. Reconfigurable computing is a viable implementation approach for applications in which the required degree of flexibility and performance justify extra silicon area.

2. The intrinsic cost of reconfigurable logic can be reduced if a reconfigurable logic architecture is tuned to the requirements of a target application domain.

3. The 4-LUT, which is the basic computing element in commercial FPGAs, is a factor of four oversized for the implementation of word-level arithmetic computations.

4. The term ‘layout’ is ambiguous in the context of programmable logic devices. It refers both to the VLSI implementation of a programmable logic device and to the way in which an application is mapped onto such a device.

5. Progress in technology is as much determined by innovation as it is by legacy.

6. Ambient intelligence is artificial.

7. Denying the existence of extraterrestrial life is egocentric.

8. Competing in endurance events, such as Ironman triathlons and ultra-marathons, says more about the mental than about the physical condition of the participants.

9. The concept of so-called ‘eco-products’ offered at premium prices makes healthy food a privilege of those who can afford it.

10. A woman’s memory is like a DRAM – it requires continuous refreshing. The memory of a man is like a hard disc – it often crashes completely.

11. We are the first generation that influences the climate, and the last generation to escape the consequences.

[prof. Olav Orheim, Norwegian Glacier Museum]