Faculdade de Engenharia da Universidade do Porto

Programmable Flexible Cores for SoC Applications

Pedro Miguel Ferreira Alves

Tese submetida no âmbito do
Mestrado em Engenharia Electrotécnica e de Computadores
Sistemas Digitais e Informática Industrial

Orientador: Prof. João Canas Ferreira

Julho de 2009

Faculdade de Engenharia da Universidade do Porto

Programmable Flexible Cores for SoC Applications

Pedro Miguel Ferreira Alves

Julho de 2009

Abstract

Today's embedded systems integrate digital and analog technologies into a single chip, known as System-on-Chip (SoC), in order to achieve increased performance, system reliability, and reduction in packaging and test costs. As fabrication technology advances and processes shrink, design costs escalate due to increasing system complexity, and challenging integration and verification tasks. Part of the digital circuitry of an SoC is comprised of fixed digital logic blocks, but another part is often custom designed, and may benefit from redesigns and updates, or may have a dual use (regular and test mode, for example). This is possible when reconfigurable digital blocks exist, that are capable of supporting different digital functions after silicon fabrication. Reconfigurable logic reduces design risk and any design error can be corrected quickly and inexpensively.

Inside an SoC, the space left after placing the analog circuitry and other large Intellectual Propery (IP) cores is very often non-rectangular. I propose using a non-rectangular core of standard cell based reconfigurable logic to fill that space, thereby adding to the final product the advantage of reconfiguration with a potentially short design cycle. The goal of the current work is to develop a variable-shape reconfigurable core architecture and the tools to support its implementation through a regular digital design flow, in order to achieve the cost-efficient inclusion of such cores in SoC-based consumer electronics applications.

The proposed general core architecture is based on a fixed set of basic cells that implement logic, switch and routing blocks. All blocks can be configured serially through one or more scan chains. The logic blocks contain 2- or 4-input look-up tables, with combinational and sequential outputs interfacing to 4 or 6 vertical and horizontal unidirectional tracks respectively for routing. The tracks connect switches on the *Wilton* switch block. The three types of blocks connect by abutment to form a cluster cell. Cluster cells connect by abutment to other cluster cells to form the reconfigurable core with the desired shape.

A software tool was created that, given the shape of the available area, generates a *Verilog* netlist of the reconfigurable core and a verification testbench. The basic blocks are described by gate-level netlists of standard cells. These gate-level netlists can be used within a standard digital design flow. In this way common Computer Aided Design (CAD) tools and optimized standard cells from available libraries can be used, reducing the design risk associated with the reconfigurable core and its impact on the SoC design cycle.

Programmable cores with non-rectangular shapes ("S", "L", "T", "U") were created and different logic functions manually mapped onto them. Validation was performed by simulating time-annotated netlists combined with a wireload model. The programming circuitry operated at 850 MHz for a 90 nm CMOS process. The speed of the logic is design-dependent, but an upper limit of 850 MHz was set by simulation. The approximate area for each cluster with 2-input look-up tables is 1600 μm^2 for a 90 nm process and 1000 μm^2 for a predictive 45 nm process. At the smaller technology node, a 100x100 programmable core takes around 10 mm^2.

Resumo

Os sistemas actuais incluem tecnologia digital e analógica numa única solução, System-on-Chip (SoC). Esta permite ao sistema alcançar maior desempenho, fiabilidade e redução nos custos. À medida que os processos avançam, os custos de desenvolvimento e implementação escalam devido ao aumento de complexidade e das tarefas de integração e verificação do sistema. Parte do circuito digital num SoC é composto por lógica fixa, mas outra parte pode beneficiar de alterações de funcionalidade, actualizações ou permitir usos distintos tais como operação normal e em teste, por exemplo. Isto é possível quando existem no sistema blocos de lógica digital reconfigurável, capazes de suportar diferentes funções após fabrico. A lógica reconfigurável reduz o risco do produto e qualquer erro pode ser rapidamente, e com baixo custo, corrigido.

Dentro de um SoC, o espaço vazio após a instanciação dos blocos analógicos e outros é normalmente não-regular. Proponho um bloco de lógica reconfigurável não-rectangular para preencher esse espaço e adicionar ao produto final reconfigurabilidade e um potencialmente curto ciclo de desenvolvimento. O objectivo deste trabalho é a criação de um bloco digital reconfigurável de forma variável e das ferramentas que suportam a sua implementação. Desta forma é possível incluir estes blocos de forma eficiente e barata em aplicações comerciais baseadas em SoCs.

A arquitectura proposta é baseada num conjunto fixo de células básicas que implementam blocos de lógica, comutação e propagação. Os blocos podem ser configurados através de uma interface série. Os blocos de lógica contêm tabelas de funcionalidade de 2 ou 4 entradas com saídas combinacionais ou sequenciais ligadas a 4 ou 6 linhas unidireccionais verticais e horizontais respectivamente para propagação dos sinais. Estas ligam a um bloco de comutação baseado nos blocos de *Wilton*. Os 3 blocos referidos compõem macrocélulas que se interligam para criar a forma não-rectangular do bloco de lógica programável.

Uma ferramenta foi criada para, dada a forma desejada e a área disponível, gerar uma descrição em *Verilog* do bloco programável junto com um ambiente de verificação. As células são descritas recorrendo à instanciação de células básicas digitais. Ferramentas de CAD e bibliotecas de células básicas podem ser usadas, reduzindo o risco associado à criação do bloco programável e ao seu impacto no ciclo de desenvolvimento do SoC.

Blocos de lógica programável com formas não-rectangulares ("S", "L", "T", "U") foram criados e diferentes funções lógicas neles mapeados. A sua validação foi feita através da simulação de circuitos anotados com informação temporal e de um modelo apropriado para as respectivas interligações. O circuito operou a 850 MHz para um processo CMOS de 90 nm. A velocidade da lógica é dependente da função implementada mas um limite superior de 850 MHz foi observado. A área aproximada para cada macrocélula com funções lógicas de 2 entradas é de 1600 μm^2 para um processo de 90 nm e 1000 μm^2 para um preditivo de 45 nm. No nó de tecnologia mais pequeno, um bloco programável de 100x100 macrocélulas ocupa uma área de 10 mm^2.

Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to complete this work.

I am deeply indebted to my supervisor Prof. Dr. João Paulo Canas Ferreira from the Faculty of Engineering of University of Porto for all the help, suggestions and encouragement that helped me all the time through the research and writing of this thesis.

Especially, I would like to give my thanks to my family and particularly to my wife Lara whose patient love enabled me to make it till the end.

"The ideal engineer is a composite... He is not a scientist, he is not a mathematician, he is not a sociologist or a writer but he may use the knowledge and techniques of any or all of these disciplines in solving engineering problems."

N. W. Dougherty

Contents

List of Figures

List of Tables

List of Acronyms

ASIC Application Specific Integrated Circuit

ASSP Application Specific Standard Product

CAD Computer Aided Design

DSP Digital Signal Processing

EEPROM Electrically Erasable Programmable Read-Only Memory

eFPGA Embedded Field Programmable Gate Array

EPLC Embedded Programmable Logic Core

EPROM Erasable Programmable Read-Only Memory

FF Flip-Flop

FPGA Field Programmable Gate Array

HRB Horizontal Routing Block

IP Intellectual Propery

LB Logic Block

LEF Library Exchange Format

LUT Look-Up Table

NMOS Negative Metal Oxide Semiconductor

NRE Non-Recurring Engineering

PCB Printed Circuit Board

PLL Phase-Locked Loop

RB Routing Block

RF Radio Frequency

RTL Register Transfer Language

SB Switch Block

SDF Standard Delay Format

SoC System-on-Chip

SRAM Static Random Access Memory

TAT Turn Around Time

VRB Vertical Routing Block

Chapter 1

Introduction

Today's embedded systems integrate digital and analog technologies into a single chip, known as System-on-Chip (SoC), in order to achieve increased performance, system reliability, and reduction in packaging and test costs. As fabrication technology advances and processes shrink, design costs escalate due to increasing system complexity, and challenging integration and verification tasks. Part of the digital circuitry of an SoC is comprised of fixed digital logic blocks, but another part is often custom designed, and may benefit from redesigns and updates, or may have a dual use (regular and test mode, for example). This is possible when reconfigurable digital blocks exist, that are capable of supporting different digital functions after silicon fabrication. Reconfigurable logic reduces design risk and any design error can be corrected quickly and inexpensively. Non-rectangular cores of standard-cell based reconfigurable logic can be used to fill space left on SoCs, thereby providing the system with hardware reconfigurability.

1.1 Motivation

Commercially available products rely on integrated systems whose costs are increasing. In order to continuously integrate systems into a single chip, and with it reduce the overall product cost (including design cycle, fabrication and verification associated costs), programmability features should be added to these systems. To be able to reprogram and dynamically reconfigure the complete or part of the digital core is an advantage, compelling and tempting. By configuring the logic blocks and routing fabric correctly, any digital user circuit can be implemented after fabrication. These programmable features allow for products to be quickly updated and corrected without having to go through the complete design cycle again, reducing this way its overall cost and increasing its time in market. Current systems may have already fixed and programmable logic functionality but provided by different chips on the same board, with the added costs this represents.

Besides the reconfigurability that the programmable logic cores add to the overall system, it is important to continue to push for the integration of such cores on SoCs, closing the gap between the Field Programmable Gate Array (FPGA) and Application Specific Integrated Circuit (ASIC) worlds, joining advantages from both. Within these, the area for the programmable logic can often be available with a non-rectangular shape, result of laying out the other digital and analog hard macros that integrate the same SoC. This way, there is a need to continue investigating programmable cores architectures and design flows that allow for these cores to be easily integrated and accommodated in non-rectangular areas within SoCs. The same is valid also for rectangular cores that are to be embedded on SoCs.

The possibility of reliably adding programmable logic to products and, with it, reducing their overall costs, time to market and risk while still being able to target a wide range of applications with different performance and characteristics with the same product is a challenge and a strong motivation for the completion of the proposed work.

1.2 Objectives

The main objectives of this work are to study the integration of programmable cores into single chip solutions by investigating and proposing an architecture for embedded programmable logic cores with non-regular shapes and by developing a methodology and associated tools to facilitate the design and implementation of these cores in a cost-efficient manner for consumer electronics applications.

1.3 Original Contributions

An architecture for non-rectangular programmable logic cores was proposed, implemented and validated through functional and post-layout simulations. To support the programmable core, a set of tools was also created to automatically generate its frontend views and support its backend views creation together with a simple testbench customized for the particular core created.

The architecture proposed is based on standard digital cells and this way the programmable core implementation benefits from a standard and well defined digital design flow and CAD tools. Using these commonly available cells on a reliable and stable design flow reduces the core development risk over a potentially short design cycle.

Several different programmable cores with non-rectangular shapes were generated and functionally validated through manually mapping simple digital logic functions on them. Within these cores, studies were performed to evaluate the reconfigurable cores area, power and operating frequency.

The results obtained show the feasibility of implementing a technology-independent, flexible architecture for non-rectangular reconfigurable logic cores. The generation of these

cores is supported by an automatic generation tool created during the development of this work and they can be physically implemented through a standard digital design flow.

1.4 Thesis Organization

After the introduction, chapter 2 presents the background and relevant information needed to understand and link the previous known work and data with the proposed one is presented.

The proposed embedded programmable core architecture is presented on chapter 3. On this same chapter, the programmable core automatic generation, validation and operation is described together with the tools developed for those purposes.

Chapter 4 presents experimental results performed with the programmable core architecture in order to characterize its area, power and delay characteristics for a 90nm CMOS process and a predictive 45nm process.

Finally, chapter 5 concludes and provides orientations for future work.

Chapter 2

Background and Relevant Work

2.1 Introduction

The recent history in the semiconductor industry shows that digital and analog technologies such as microprocessors, embedded memories, Radio Frequency (RF) analog devices, among others, can be integrated into a single solution known as SoC. This is the embedded system of today and the future. Pre-designed and verified blocks, cores and other macros commonly referred as IP macros, are obtained from third-parties or internally developed, and combined onto a single chip. These cores may include embedded processors, memory blocks, or specific processing functions circuits. They can then be combined onto a chip to implement complex functions as the example on figure 2.1 shows.

This integration allows for the increase on the speed of communication, reduction on packaging and test costs and added system reliability. In the future, these highly integrated systems will include also mechanical, optical and bio-electrical circuitries.

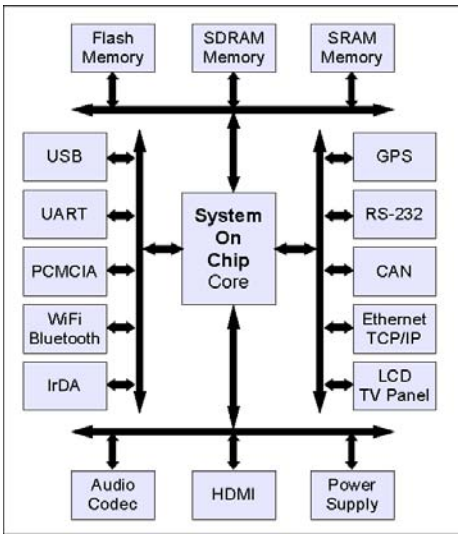


Figure 2.1: Example of a System-on-Chip.

The SoC methodology provides an elegant solution to incorporate also one or more programmable logic cores into it. The possibility to include embedded programmable cores within these systems is accompanied by benefits and several challenges that will be discussed through this document.

These programmable cores target the implementation of arbitrary digital logic circuit after fabrication. Reconfigurable logic reduces design risk and any design error can be corrected quickly and inexpensively. With the move to extremely complex and big designs and the added pressure to get products early in the market, more and more companies are addressing the need and taking into consideration the advantages of integrating programmable logic into their SoCs.

According to market analysts, the trend is for the market for products with programmable logic cores to grow. The data disclosed by the International Technology Roadmap for Semiconductors [1] points to a continuous increase in the amount of reconfigurable logic on SoCs. Table 2.1 represents this trend.

Year of Production	**2007**	**2008**	**2009**	**2010**	**2012**	**2014**	**2016**	**2018**
% of reconfigurable SoC functionality	28%	28%	30%	35%	40%	45%	50%	56%

Table 2.1: System-on-Chip ITRS reconfigurability trends.

Products related to communication applications should lead the way, followed by networking and telecommunications infrastructures. Nevertheless, these hybrid solutions with fixed and programmable logic cores will never be the designer's target unless they are economically viable.

As technology advances and we observe a continuous shrink of process nodes, design costs are becoming prohibitive due to increase in system complexity, integration and verification tasks, design and mask production. This cost increase is presented on both figure 2.2 and 2.3. The data provided by UMC[1] present the increase of overall fabrication costs with time. Altera[2], a company leader in innovative custom logic solutions, shows the increase on total development costs for ASIC designs as the process nodes shrink.

Each new solution is becoming more complex and more expensive to develop. The inclusion of programmable cores on these products would amortize the cost of developing several chips for several different products.

With the increase on chip design costs, the economics will be a strong driver for the addition of hardware programmability. These embedded programmable core solutions will increase the yield of these chips, by allowing for the product lifetime and time in market to be increased, as will be shown later on this document. Since low yield on expensive SoCs can result on the premature ending of the respective product development, the embedded reconfigurable core is worth to be considered for consumer products.

[1] http://www.umc.com/
[2] http://www.altera.com/

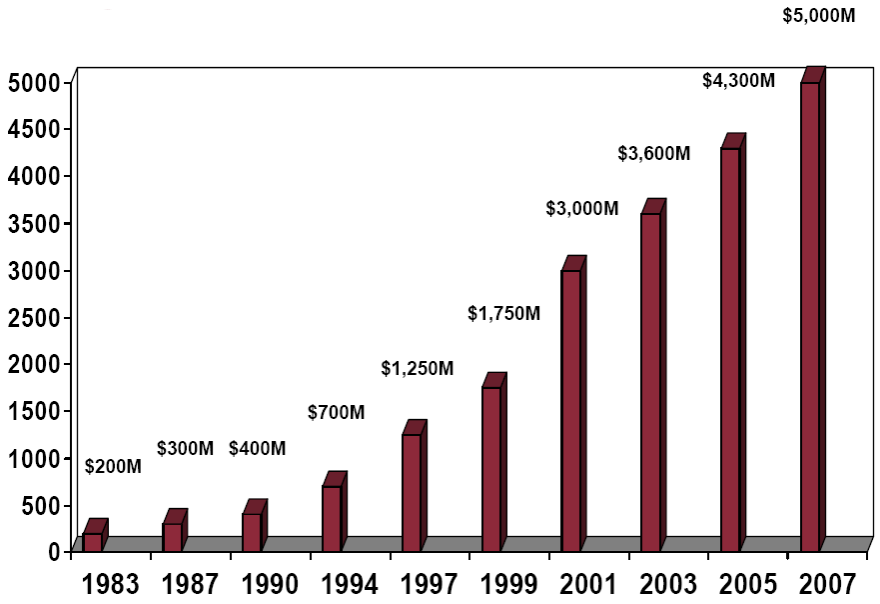


Figure 2.2: Fabrication costs, from UMC.

In summary, the overall design costs of SoC design, combining engineering, IP blocks and mask costs help to understand the continuous effort that should be made to integrate programmable cores into complex systems such as SoCs.

Adding reconfigurable logic cores to SoCs enhances them by providing the system with run-time reconfigurability that can be used for post-fabrication modifications. Digital logic programmability allows for a whole set of different logic functions to be quickly implemented after circuit fabrication including upgrades to the design and design errors corrections.

These changes or modifications applied to programmable logic after circuit fabrication can be quickly and inexpensively accommodated what translates into a very low Non-Recurring Engineering (NRE) cost as there is no need to wait for new silicon to be manufactured and tested with the correction performed. When programmable logic is not present on the system, such a change or modification adds significant delay and risk to the product design cycle due to the expensive design iterations.

With a solution that contains embedded programmable logic, the interface between fixed and programmable logic doesn't have to cross chip boundaries and Printed Circuit Board (PCB) lines, saving power and time. Without having to deal with the extra costs of assembling and testing a second chip, embedding a programmable core when such logic is needed ends up reducing the system cost by reducing the number of chips on the same board design.

Another advantage with embedding programmable logic cores is related with the fact that there are only some commercially available programmable chips of specific sizes what can result in a waste of resources, money and space. Programmable logic cores embedded

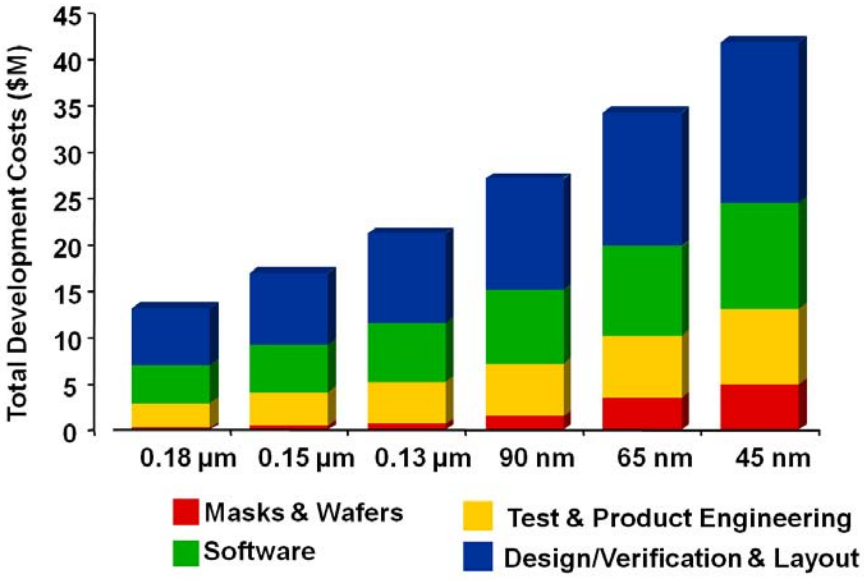


Figure 2.3: Total development costs, from Altera.

on SoCs can be implemented with the size and shape as needed.

As for the product life cycle, the addition of programmable logic cores reduces the product time to market as the digital logic functions for each product can be implemented after silicon is fabricated and prolongs its time in market, extending the market window, as product corrections and updates can be implemented on the programmable logic core without having to manufacture new silicon. Early market penetration increases the product probability of success what is attractive to any company.

Embedded programmable logic cores are the natural solutions to close the gap between FPGAs (external chip providing programmable logic) and ASICs/SoCs (typically with only fixed digital logic) designs providing the best possible performance metrics as well as reduced costs and development times. If programmable logic is required, an embedded solution can offer higher performance and area-efficiency than the use of an external FPGA.

Only a customized solution can accommodate critical size and performance requirements of current ASIC and FPGA designs and, at the same time, take advantage of today's capabilities of using programmable logic and programmable devices that can meet, due to the continuous technology processes advances, for all areas of circuit design, the needed power, delay and area requirements.

2.2 Field-Programmable Gate Array (FPGA)

FPGAs [2] are the most commonly available programmable logic devices. They are reconfigurable digital logic chips that can be configured by the customer or designer after

manufacturing in order to implement any digital logic function. This capability exists at the expense of added delay (reduced performance), area and power consumption.

The basic FPGA consists of arrays of logic blocks with electrically programmable interconnections. Early devices contained the equivalent of a few thousand gates, but today's number has grown into the millions. This flexibility allows designers to create hardware functions that exactly match the needs of a specific embedded application. In addition to the logic blocks, the latest devices embed dedicated processors and other macros within the silicon, allowing the designer to make hardware and software trade-offs to meet performance requirements.

Several different architectures exist [3, 4] and are implemented on commercial FPGAs. As will be described on section 2.2.1, FPGA architecture is divided into a logic and a routing section. The architecture variations target the logic blocks complexity, the routing channels length and segmentation and the switch blocks complexity with the purpose of being able to provide higher performance for a bigger range of different applications. There are also other FPGA architectures, that, being customized for a particular type of application, are less flexible but offer higher performance for that particular product or application. For example, a FPGA that includes Digital Signal Processing (DSP) blocks or other fixed logic dedicated circuits.

The development and creation of a FPGA is a time consuming and resources demanding task as they're typically implemented using a full custom layout flow to achieve the best possible performances and compromise between area, power and delay.

A detailed description of a FPGA design from architecture to circuit design and layout is found on [5, 6].

Modern FPGAs are not only composed by the programmable digital circuit but may include other specialized hard macros such as Phase-Locked Loop (PLL)s and other clock generation circuits, processors and dedicated signal physical interfaces such as Ethernet.

2.2.1 FPGA Architecture

There are four main categories of FPGAs available: symmetrical array, row-based, hierarchical and sea-of-gates. What differs in all of them is the way the interconnections are implemented and the way the programming is made. The most common and simple type of architecture is the symmetrical array that is most commonly known as an island-style architecture.

This island-style architecture, being uniform and simpler to be used, was chosen as the base for the programmable logic core architecture proposed later on this document. Figure 2.4 exemplifies a typical island-style FPGA with the logic, switch blocks and interconnects that compose it. These blocks will differ on each implementation being more or less complex according to the FPGA targeted performance or application needs. An internal fixed logic configuration circuit is also present on the FPGA to support its usage and dedicated pads are used to surround this island-style circuit mesh and provide the

interface to the outside world. The configuration circuit and dedicated pads are not visible on the picture presented below.

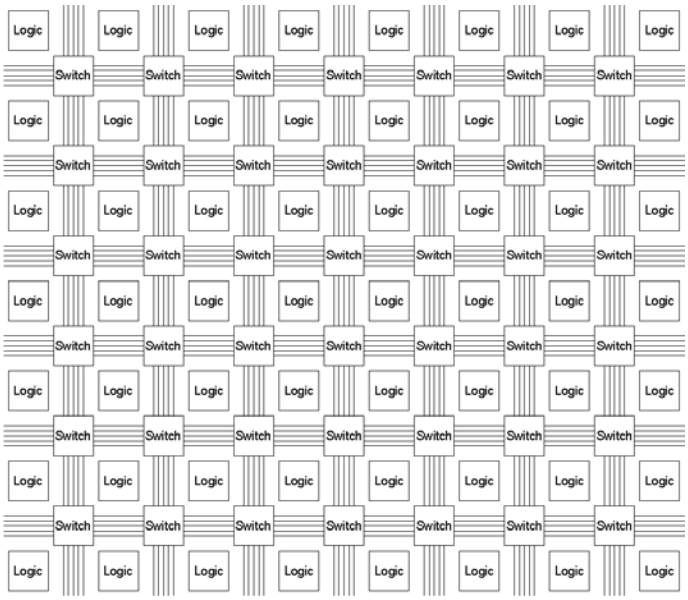


Figure 2.4: Example of an island-style FPGA architecture.

FPGA different architectures have a significant different impact on the quality of the product speed performance, area efficiency, and power consumption. The architecture is divided into two groups that define the overall FPGA architecture: logic block and routing architecture.

Within the different logic block and routing architectural choices, a FPGA overall architecture can be implemented as an array of the same logic and routing blocks but it can also be implemented as a group of different logic and routing circuits for increased performance and area efficiency. FPGAs created this way are respectively considered to be homogeneous or heterogeneous.

It needs to be taken into consideration that the best architectural choice is highly dependent on the type of designs it will support and the programming technology chosen such that there is no architecture that can be considered the best for all cases.

2.2.1.1 Logic Block Architecture

The logic block is responsible for implementing the gate level functionality required for each application. The functional complexity of logic blocks can vary from simple boolean operations to larger and complex arithmetic operations. The logic block is defined by its internal circuit and granularity: fine or coarse grain. Fine and coarse grain means that the FPGA is composed by small or large parts respectively. The granularity can be increased, by making use of large and complex cells capable of supporting multiple digital logic functions, to reduce the number of needed logic blocks (one example is visible on

figure 2.7) but coarse grained FPGAs may end up with wasted silicon area. On the other hand, fewer logic blocks exist on the critical path of a given circuit, allowing it to reach higher speed performances.

Between a very fine grain and very coarse grain logic block architecture a huge amount of architectures exist that differ on the area, speed and power metrics they are capable of obtaining.

One way to change the granularity of the FPGA is to change the number of basic logic elements that are organized into clusters and the way these clusters are hierarchically connected into the interconnect structures.

The most important part of the basic logic block is the portion of circuit that holds the truth table for the logic function and routes the function result to the logic block output. That part is known as the Look-Up Table (LUT) [7].

Using large LUTs usually results in good performance since there is a fewer amount of logic on any given critical path. As the logic block size increases with the number of LUT input signals, more silicon area is used. An important consideration for the analysis of any FPGA logic block is to balance the size and performance compromise and determine the optimal LUT size that will be sufficient to meet area usage, logic density and performance. Past work [8, 9] shows that a LUT size of 4 input signals is the most area-efficient for a non-clustered context and the work developed on [10] suggested that with a heterogeneous mixture of LUT sizes of 2 and 3 an equivalent area efficiency to a LUT size of 4 could be achieved. A more recent study [11] has found that LUT sizes of 4 and 6 inputs and cluster macrocells integrating 3-10 LUTs result in a better area-delay compromise than initially documented.

A compromise must exist between the LUT block size, density and its performance.

The logic block is typically implemented using Static Random Access Memory (SRAM) cells or variants of these to serve as 1-bit memory cells for the LUT that will store the programmed digital logic function. A N-input LUT is capable of implementing any N-input function and for a N-input LUT, 2^N SRAM cells or other memory cells are needed. Together with a 2^N input multiplexer, any boolean combinational function can be implemented. The LUT holds any digital logic function with a constant delay as the signal data paths are the same across the LUT for any given function.

As the example on figure 2.5 presents, a Flip-Flop (FF) can also exist at the LUT output to sample the combinational output and generate its sequential output. A multiplexer is often used to select the logic block output as the sequential or combinational output of the mapped function. Figure 2.6 shows a representation of a possible LUT implementation as described before. Using LUTs with a larger number of inputs (N) reduces the number of LUTs required to implement any circuit and subsequently reduces routing demands. However, it increases the circuit area and complexity exponentially. On figure 2.7, a macrocell with K N-input LUTs is presented. The complexity of this macrocell defines the FPGA architecture granularity.

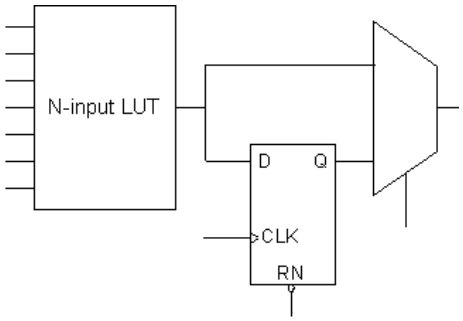


Figure 2.5: FPGA N-input look-up table (LUT) and logic block.

2.2.1.2 Routing Architecture

The programmable routing in a FPGA provides the connections between the logic blocks that hold the user digital logic function. These connections are composed by unidirectional or bidirectional segments and programmable switches and they should be flexible enough to support local and long routing demands while respecting specified speed and power consumption constraints. The routing structure can be organized within the FPGA on a flat level or through a hierarchical organization where connections on higher levels of hierarchy correspond to longer connections on the programmable architecture.

Digital circuits require also a group of global signals such as reset and clock signals that must be distributed across the whole FPGA. These dedicated routing networks are designed to guarantee this distribution and are commonly available on modern FPGAS.

FPGA routing architecture incorporates segments of different lengths. These can additionally be connected through programmable switches to form longer wires. The choice on the number and length of segments impacts the FPGA density. A compromise must exist between the length of the tracks, their span over the programmable logic core and the performance and routability they'll support. Short tracks force signals to cross several switch blocks before they reach their destination, degrading the system performance by increasing the combinational delay between blocks, reducing its maximum possible operating frequency. Longer wire segments span multiple blocks and require fewer switches, reducing routing area and delay but decreasing also routing flexibility, which reduces the probability that a user circuit can be routed successfully. A small number of wires will results in only a fraction of the logic blocks being utilized and a poor FPGA density while an excessive number of wires results in unused silicon area.

The detailed routing architecture discussion also includes the compromise between unidirectional or bidirectional routing on each routing block. Bidirectional routing wires increase the programmable core flexibility but additional control logic needs to exist and with it, additional silicon area is used. The use of bidirectional wire segments can also result in many routing switches unused and the additional sinks per wire segment increase the lines capacitance and therefore impact the delay. With unidirectional segments

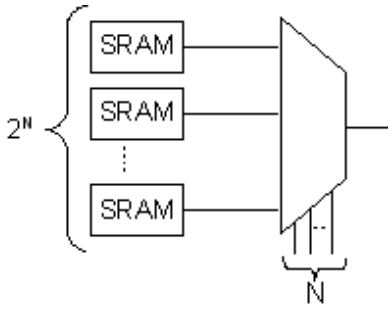


Figure 2.6: N-input look-up table (LUT) implementation.

[12], a larger number of wires on each routing channel might be required for the same performance. Some previous work [13, 14] state that more or less the same number of unidirectional tracks is required for the same routability.

Past studies [15] show that the most area-efficient routing architecture is the one with completely uniform routing channels across the entire chip and in both horizontal and vertical directions. According to the authors, what explains this observation is that the FPGA circuits tend to have routing demands that are evenly spread across a FPGA and this way, mapping best to a uniform routing architecture.

Where vertical and horizontal tracks intersect they connect to a switch block. Switch blocks allow for tracks to connect to adjacent channels and this way route the signals as needed. These blocks will this way connect any given input signal to a possible set of one or more output signals. The switch block shown on figure 2.8 is an example where the switches are represented at the wires intersection by a circle. On the same figure, a representation of one group of switches is visible in the form of Negative Metal Oxide Semiconductor (NMOS) transistors that close or open the switch through the assertion of the respective logic value at the transistors gate inputs.

Still regarding switch blocks, the circuit complexity and number of supported connections must be defined taking into consideration that a short number of internal switches on this block will difficult and reduce the routability of a digital design and a large number of internal switches can result on a waste of resources and area. Low routability will result in a programmable core with small flexibility but high routability beyond necessity can result in a large area, delay and power dissipation that is not needed. A large number of programmable switches ease the task of completing the routing but they also consume a significant amount of area and for this reason, a compromise must also exist regarding the number of needed programmable switches.

On typical FPGA applications, the switch blocks are implemented using SRAM cells and pass-transistors logic. The complexity of the switch block is a compromise between area and delay added to signals and routability. Several switch blocks have been studied [16] on the past such as the *Universal* [17, 18], *Disjoint* [19] and *wilton* [20], among others, each with different characteristics and therefore different performance and routability

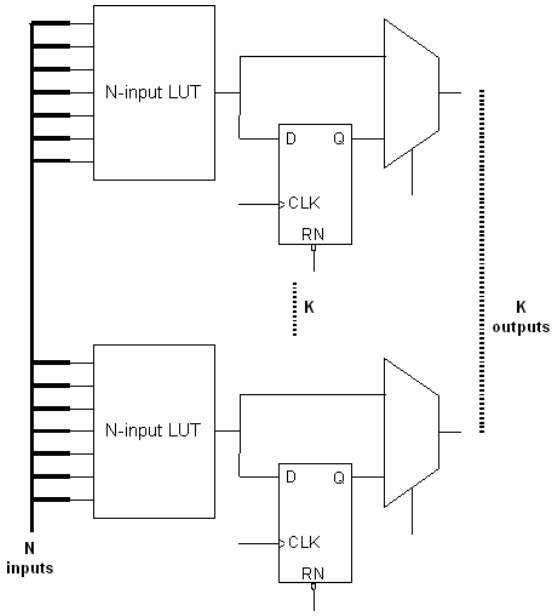


Figure 2.7: Macrocell with K N-input look-up tables (LUT).

results.

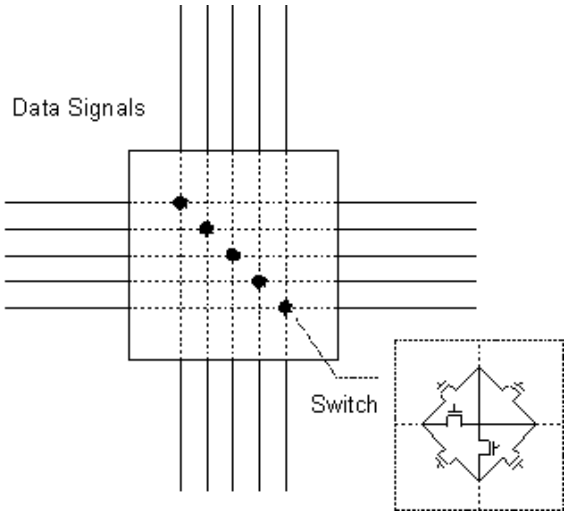


Figure 2.8: FPGA switch block.

2.2.2 Programming Technologies

Different programming technologies are used to implement the programmable switches on current FPGA implementations being that the most dominant one is done through the use of SRAM cells, where the switch is a pass transistor controlled by the logic value stored on the SRAM cell. Other switches are implemented through the use of anti-fuse, Erasable Programmable Read-Only Memory (EPROM) and Electrically Erasable Programmable Read-Only Memory (EEPROM) technologies.

SRAM cells are re-programmable and can be implemented on standard CMOS processes. Since no special manufacturing steps are required other than the standard CMOS, the latest technologies can be used to create SRAM-based FPGAs and this way take advantage from higher integration, speed and lower power consumption.

2.3 Application-Specific Integrated Circuit (ASIC)

An ASIC chip is a system designed with non-standard integrated circuits customized for a particular use. Products implemented on an ASIC core typically target a very high volume production to amortize the high development cost that is typically associated with the development of these solutions.

These chips can include a large number of digital and analog building blocks such as dedicated or general purpose processors, memories and others. This kind of solution is often referred as SoC [21].

There are three main categories for ASICs: gate-array, standard cells and full custom.

On a gate-array solution, transistors or gates are fabricated on a two dimensional array to form the basic circuit mesh for the ASIC. The device is then programmed through the customization of the upper layers that will connect the desired nodes on the array.

A development based on standard cells uses standard optimized blocks with a defined functionality (simple boolean logic functions such as inversion, AND as well as sequential logic). These cells are grouped into digital blocks that implement complex digital logic functions. They provide high flexibility in the creation of ASIC digital cores.

The full custom design offers the highest performance and the smallest silicon used area at the expense of increased design time, complexity and risk.

A typical CAD system for ASIC design is an integrated suite of software facilities for design entry, functional simulation, physical layout, test simulation and design verification. Hierarchical design, with detailed schematics specifying the function of each part, and symbols enabling them to be instanced in higher-order parts, is the key to well-structured ASICs.

The most common ASIC digital logic cores are often implemented using standard cells from commonly available libraries as they allow for the designer to quickly implement a digital core and target it for high performance and a compromise between area, delay and power using standard CAD tools. These digital cores present on ASIC typically represent fixed logic. With only fixed digital logic available on the system, it is not possible to re-program the digital core to operate with a different function, no reconfigurability.

In summary, ASIC designs offer an attractive solution for high volume production applications. They integrate a significant amount of analog and digital circuits that, if available only as discrete components, wouldn't allow the speed and performance that an ASIC design is able to provide.

2.3.1 Standard Cells Based Design

Several different simple logic functions such as inverters, AND and OR gates and multiplexers, among others, are implemented by blocks of transistors that are connected together to form smaller cells. These cells share the same height and are placed together by abutment. The numerous layers of metal that are currently available on modern processes are used to interconnect these cells over the locations where they're placed and establish this way the complex digital logic functions the digital core implements.

Standard cells libraries are commonly available from a wide number of providers in all available technology nodes and flavors. These companies provide and support all needed data and information to be used on the large number of tools that run on a digital design flow. With this data and through this flow, the designer is able to easily create and validate any digital core using the wide number of automated tools that exist to support the digital core creation and validation.

The usage of standard cells for the implementation of fixed digital logic cores has been widely accepted in the industry for the design of ASICs. Although a full custom manual layout flow can possibly result on a higher performance and area efficiency, the implementation of a big and complex digital core is not doable on a full custom layout flow. Together with this, standard cells based digital cores are capable of delivering the performance needed for all types of applications.

2.4 FPGA vs ASIC

Programmable logic is typically available through the use of external chips, specifically developed for that purpose, such as FPGAs, while ASIC designs typically contain only fixed logic. Understanding the differences between them will allow to easily comprehend the advantages of integrating the concepts of programmable logic cores to the ASIC world, where the flexibility given by these cores is not usually present, while maintaing its performance metrics and still targeting a wide range of products and applications.

Table 2.2 [22] summarizes some of the characteristics of FPGAs and ASICs, or capabilities that can be achieved with each of these designs, by describing parameters that characterize them.

FPGA circuitry can be programmed and reprogrammed to hold several different digital functions. This reconfigurability and flexibility, however, comes at the expense of increased area, delay and power consumption what is expected since more devices are used to execute the same logic functions and more internal gates need to be driven. Some past studies [23]conclude that an FPGA is on average 35 times larger, 3.4 to 4.6 times slower than a standard cell implementation and that it also consumes 12 times more dynamic power than an equivalent ASIC on average. Other studies have been performed to compare also standard cells implementations with full custom design [24, 25, 26, 27], all presenting

FPGA	ASIC
Easy to design	Difficult to design
Short development time	Long development time
Limited design size	Large designs supported
Limited design complexity	Complex designs supported
Limited performance	Higher performance supported
High power consumption	Low power consumption
High per-unit cost	Low per-unit cost
Fast design re-spins	Slow design re-spins

Table 2.2: FPGA vs ASIC characteristics comparison.

results that show that significant area, power and delay overhead exists when comparing a standard cell solution with one implemented on a full custom flow.

A FPGA has a very reduced engineering cost and development time since they are silicon validated solutions, commercially available, well documented and with all the needed tools to support them and their usage. However, the development and implementation of a FPGA circuit is a time consuming task that requires a large engineering effort for the architecture and layout generation. To compensate for the generalized architecture and the added delay and power consumption, the circuitry is drawn in a full custom design flow, where the design is carefully drawn to achieve the best possible performance.

An ASIC platform has lower design flexibility than a FPGA platform due to its fixed digital logic. With this kind of designs, it is possible to obtain high speeds of operation and reduced area and power consumption. These high performance characteristics come at the expense that these solutions are more complex, with higher design risk, take a longer time to develop and require a big engineering cost.

With fixed digital logic, corrections or changes are not easily accommodated and may require a complete design cycle including manufacturing and silicon validation steps what can hit and seriously affect any product schedule for market release.

ASICs usually take months to develop and fabricate and cost hundreds of thousands to millions of dollars for a first part. They also provide lower cost for high volume products. FPGAs can be configured on a very short amount of time, reconfigured anytime needed and cost less than a few thousand dollars. They're more expensive for high volume products.

The understanding about these parameters and characteristics should help a designer to decide on the inclusion of programmable logic on its product knowing the advantages the FPGA world will bring to the ASIC and SoC. The selection criteria, to embed or not programmable logic on the system, besides the possible need to do it, should also take into account its total cost, performance, power consumption, system size, reliability, design flexibility, ease of design and time for product to reach market.

Due to the increase in area, power consumption and delay, the target application for embedded programmable logic cores is the most appropriate when the amount of programmable logic required is small.

In summary, these offerings have distinct advantages: performance and density for ASICs versus Turn Around Time (TAT) and flexibility for FPGAs.

Bringing programmable logic into ASIC solutions, through the form of embedded programmable logic cores, will reduce their development cycles cost, risk and time while enhancing it with digital core programmability and maintaing area, power and delay characteristics needed to support current and future products always searching for the best cost-efficiency compromise.

The gap between FPGAs and ASICs can be reduced by adding hard blocks, with specific and customized dedicated functions, to FPGAs or embed digital programmable logic to SoCs.

2.5 Previous Work

Embedded FPGA cores can be divided into soft and hard macros [28]. The simplest approach to embed FPGAs on SoCs is to strip them from the I/O pads and adapt it such that it can be treated as another IP core. These IP cores are identified as hard IPs and can be typically used to implement small to medium logic functions like microprocessor accelerator functions. Hard IPs means that their layout, speed and other characteristics are fixed and cannot be changed. The advantage is that the user doesn't need to design the programmable core and can license it from a vendor that has already designed and validated it. The fact that only specific block sizes may be available, potentially resulting on waste of resources and that available ones may not be suitable for a certain set of applications is a limitation on the usage of these hard Embedded Field Programmable Gate Array (eFPGA)s.

A more efficient approach is to automatically generate the eFPGA fabric as needed within the ASIC or other customized design flows. This methodology is referred as to create soft eFPGA IPs. In [29, 30, 31] their implementation issues are considered namely the programmable core size selection, connections between fixed logic and programmable core logic and routing of the global clock signals to the programmable logic core. On this case, the programmable core architecture is described in behavioral Register Transfer Language (RTL) and implemented together with the rest of the digital cores through a standard digital design flow. ASIC tools don't need to be modified and the flow follows a standard design flow that the designers are familiar with. Synthesizable architectures to support sequential [32] and combinational [33, 34] circuits are another common approach. This technology independent methodology is more flexible than when using hard eFPGA IPs but from it results higher area, delay and power overheads due to the usage of standard cells libraries. For very small amounts of logic, this ease of use may be more important than the increased overhead. Other advantages such as the easy integration with fixed digital logic and the possibility of creating a core of any size and shape can help the designer to choose. By customizing the standard cells library used or adding tactical standard cells

to the common libraries it is possible to reduce the area, speed and power overhead. On [28, 35, 36], additional methodologies are proposed that combines both advantages of hard and soft embedded FPGAs to create firm IP FPGAs.

Table 2.3 summarizes the embedded FPGA methodologies briefly described above.

Soft eFPGA	**Firm eFPGA**	**Hard eFPGA**
Behavioral RTL	Structural RTL or gate-level netlist	Transistor-level design
ASIC flow	Custom ASIC flow	Full-custom flow
Generic standard cells	Custom tactical cells and generic standard cells	Full-custom design
Logic synthesis required	No logic synthesis required	No logic synthesis required
Cells free to move	Regular, tiled structure	Regular, fixed-tile structure
Configurable architecture	Fixed architecture	Fixed architecture
Flexible size, no fixed shape	Flexible size, shape can be fixed	Fixed size and shape
Mixed with cells used for rest of design (fixed logic)	Designed as a separate core and inserted	Designed as a separate core and inserted
Small amount of programmable logic	Small-to-medium amount of programmable logic	medium amount of programmable logic

Table 2.3: Embedded FPGA implementation methodologies.

Structured ASICs emerged as an alternate way to reduce the gap between ASICs and FPGA solutions [22, 37]. Structured ASICs offer advantages in cost and reduced design cycle time when compared to traditional standard cells based ASIC designs. This methodology customizes only a small number of metal and via levels reducing the overall development cost and design effort. This is possible if appropriate circuitry exists and is already implemented such that the users design will just complete the connections and form the digital logic function desired. *eAsic*[3] is one of the companies that through this approach is able to create circuits with a significant higher gate density than of traditional FPGAs since most of the programmable interconnect is removed and so this area overhead associated with FPGAs disappears. The approach followed by *eAsic* has the disadvantage that reconfigurability after fabrication is not possible.

Among other approaches to reduce the area gap between FPGAs and ASICs, [38] proposes that each hard block of logic is accompanied by its own soft logic that can be used in the event the hard logic is not needed. With this methodology, a soft logic block, identified as a shadow cluster, is used in parallel with the hard logic block. A multiplexer is also used on the circuit to select the output of the active block. This method resulted on improved area-efficiency when compared to commercial FPGAs with the same hard blocks of logic since the area cost for programmable logic and routing is not wasted for unused hard logic circuit.

[3] http://www.easic.com/

Companies like *LSI Logic*[4] and *ChipX*[5] have added to some of their ASIC products small programmable logic cores consuming more gates per square millimeter than on an ASIC architecture but less than on a typical FPGA. *Actel*[6] and *LSI Logic* also produce and license FPGAs structures as ASIC cores by providing small programmable logic blocks for integration on customer's solutions.

The VariCore IP is an embedded reprogrammable "soft hardware" core designed by *Actel* to be used in ASIC or Application Specific Standard Product (ASSP) SoC applications and called embedded programmable gate array. The main building block can implement about 2.5k ASIC gates and the grouping of these blocks together with configuration and test logic creates the SRAM-based embedded programmable core architecture. The approach from *LSI Logic* is called LiquidLogic and consists on a group reconfigurable arithmetic logic units designed to be embedded on SoCs.

A cooperation between *IBM*[7] and *Xilinx*[8] resulted on the integration of standard FPGA blocks on the ASICs developed by *IBM* [39].

Other previous studies used as a reference for this work have targeted the layout automation as a means to reduce the intensive and manual work of implementing the programmable logic circuits under a full custom layout design flow. One example is the work developed on [40, 41]. Several automatic layout methodologies are described, including one using standard cells as a base for the design. The Totem Project [42, 43, 44] is one of those examples whose objective is to reduce the design time and effort in the creation of domain specific reconfigurable architectures. These architectures are optimized for a set of applications and constraints. This way, the programmable logic created is smaller in area and performs better than a standard general-purpose reconfigurable core while maintaining the post-layout flexibility needed to support the set of specified applications. This work results show that it is possible to use, among other methods, standard cells to automate the programmable logic layout generation and that the savings gained increase the application domain is narrowed. Further improvements have been introduced also by [28] by adding architecture-specific tactical standard cells in the ASIC flow, allowing the embedded programmable core generation to be configurable and imposing a regular structure to the core architecture. The GILES project [45] automatically generates a transistor-level schematic from a high-level architectural specification of a FPGA. A cell-level netlist is also generated and placed and routed automatically. The tiles created can be grouped together to create the FPGA array.

The usage of standard cells to create the programmable circuit mesh is well explored on [40, 34, 42, 43, 44]. They also suggest that the standard cell design methodology is capable of delivering and handling programmable logic circuits such as the ones found on

[4] http://www.lsi.com/
[5] http://www.chipx.com/
[6] http://www.actel.com/
[7] http://www.ibm.com/
[8] http://www.xilinx.com/

FPGAs. On [34], the programmable core was restricted to have a directional bias to avoid combinational loops but this results on a less flexible routing as it only allows it in one direction. As expected, these approaches show that these designs present significant area, speed and power consumption overhead when compared to full custom implementations. However, these significant overheads don't take into consideration the huge design cycle reduction that one is able to obtain to implement these programmable cores using standard cells and commercial tools.

Some work has been also done regarding non-rectangular reconfigurable cores and their integration on SoCs [34]. CAD algorithms for mapping, placing and routing digital designs on non-rectangular cores are the focus of the work described on [46, 47]. This work focuses its analysis on typical island-style FPGAs and proposes a new specification for the blocks architecture, to be used with current FPGA CAD tools and improvements to available tools for added efficiency and to better deal with non-rectangular shapes. Having a specification created and the tools improved, the author is able to experiment around different shapes and shapes ratios with different benchmarks and conclude about the speed and density values obtained among them.

There is also some previous work targeting run-time reconfigurability with FPGAs embedded on SoCs [48]. The embedded programmable core on this system is based on standard cells.

2.6 Programmable Logic: Applications and Products

The inclusion of a programmable logic core on a SoC [49, 50] is appealing and valuable in different applications and scenarios as the same silicon die can serve a wider range of applications. Each new product virtually mandates a development cycle where the implementation is cycled through numerous prototype, test, and debug iterations before it becomes stable. Programmable logic is virtually the only way to produce such designs efficiently. Examples can be given for the areas of portable, medical and communication devices.

2.6.1 Changing Standards and Requirements

The market is very competitive and the pressure for products to enter it even before some standards are finished is constant and high. If the product life cycle and schedule depends on the closing of the standard or standards it implements, it might suffer a big impact.

Programmable logic allows for part of the standard to be updated or implemented very late in the design cycle (even after silicon fabrication). For example, a cell phone might operate with different protocols in different geographical areas and the digital logic functions to implement these protocols can be implemented as needed after the product is fabricated.

Other applications would include signal processing, cryptography, image analysis and a whole range of different algorithms that these applications need to be support. For example, by the time 56K modems were being introduced, algorithms were not standardized and different manufacturers had different algorithms implemented in different products. Rockwell, a company releasing 56K modems to the market, allowed the users to update these standards at a later stage by providing programmable logic in their product, thus gaining market share.

Also when dealing with multiple standards, where it is a high risk to select one because many contenders will not succeed in the long term, to have programmable logic on the system reduces this risk as the solution can be quickly updated to the correct standard.

2.6.2 Customized Logic Interface Layer

An unique design, serving several customers and their products and applications, may be desired. Unique requirements from different customers may be handled by the programmable logic core as opposed to designing all the features that respect all requirements and a mechanism that selects them or maintaining different databases, each requiring its own mask sets, what is an expensive scenario. The programmable logic would enable to have a single chip with a customizable interface for each customer. A single chip can be manufactured and serve as a family of devices with the same silicon die. The cost of developing these solutions over several products is this way amortized.

The ability to quickly customize a product by acting on the programmable logic enables the companies to offer different products for the same market segment and allows for a low risk evaluation of new markets since the same core solution can be used with a customized logic layer.

2.6.3 On-chip Testing

With the increase of circuit integration, development in packaging technology and circuits' complexity, the systems testing complexity has also become more demanding and complicated. If each individual block carries with it its own testability circuitry, it is difficult and complicated to integrate them all such that the chip remains easily testable and without losing any of the testability each individual block would support. Programmable logic can be used on these cases to glue testability logic and generate stimulus for each part of the chip. The advantage is that the tests implemented on programmable logic can be easily changed in order to reflect either new or improved checks. There will be no need in this case to have all the testability logic implemented on fixed logic, wasting area and resources. The designer can devise new tests when required and implement them on an already fabricated silicon die.

2.6.4 Design Corrections and Improvements

When a system uses programmable logic, design corrections and improvements can be quickly and inexpensively implemented. The possibility to use the programmable core for these modifications is the opposite of having to go through a complete silicon re-spin, required for corrections to be implemented on fixed logic, including new manufacturing masks. An important scenario where the application of programmable logic takes a deep importance is for military applications, where reliability and flexibility are the most crucial items being considered and where performance is important but the correct functionality of the device in harsh conditions and environments is critical. Also regarding the possible updates that can be implemented on the available programmable logic core, it is worth mentioning that these can be done locally or remotely through any standard web-based interface that the system might implement. This characteristic pleases the system manufacturers in the sense that, if needed, systems can be easily globally updated.

2.7 Other Considerations

Besides the already mentioned topics, there are others that are influenced or have a different value if a certain product includes a programmable logic core, embedded or not on its SoC. The discussion around these topics helps to establish the impact on market economics, engineering development time and other general design considerations such as cost, risk, complexity, time to market, re-use possibility, production volume and others.

If programmable logic is needed on a certain product, to embed it on the SoC will end up reducing the number of components and other external chips that would provide this programmable logic on the end product, making it cheaper for the end user.

The fact that a programmable core exists increases the product life on market since it is possible to correct, change and update the system through programmability of the digital core. Having implemented and validated a programmable logic core for the first product, developing a new one with it makes it faster to reach production and therefore reduce its time to market.

To accommodate these programmable cores into a SoC, their creation, implementation and use flows should be very flexible and well documented. A designer should be able to quickly, inexpensively and reliably create and integrate a programmable core and correct, change and update its functionality after fabrication.

It should be possible to integrate the creation of the embedded programmable cores on standard design flows such that the investment to purchase additional specific tools, as needed for FPGAs development and usage, is not needed.

Reusing embedded IP cores in SoC designs has become the primary means of improving the productivity of designers facing very large and complex developments. On the business side, the embedded programmable core can be released and sold as a IP soft or hard core. The soft core corresponds to technology independent functional representation of

the programmable logic core and has the advantage that it can be implemented by the customer on any desired technology and in any desired shape and size. A hard core is the physical implementation of the soft core and, being tied to a certain technology, lacks the flexibility and technology portability the soft core provides. The embedded programmable cores can be considered like any other IP cores in the SoC design methodology.

2.8 Summary

Independently of the amount of concerns and verification coverage are put into a SoC, there are always products that end up not working properly for the intended functionality either due to design errors, wrong simulations and verifications or even a later change in the design requirements. To embed programmable logic cores on these products reduces its risk, shortens it's time to market and, overall, its cost.

A disadvantage of using embedded programmable cores is the area and delay penalty that accompanies them since every logic function uses the same physical space and takes the same delay. In conclusion, programmability takes time and space.

Programmable logic cores, embedded on ASIC solutions, allow for users to closely integrate them with other macros on the chip, adding digital core programmability to the product while maintaing its performance requirements and still valid for a broad range of products and applications.

Previous work demonstrates the possibility of embedding hard and soft FPGA cores on SoCs. Their results clearly show that it is possible to include reconfigurable logic macros on these complex systems through different methodologies and making use of different resources. The higher efficiency gained with the integration of soft cores through standard design flows is documented together with some approaches that make use of standard cells libraries for that same purpose. Finally, some of the previous work also targets usage of programmable logic cores generated with non-rectangular shapes by addressing the changes needed to available tools to improve its efficiency with these cores.

The work developed during this thesis, to be presented on the next chapters, proposes a reconfigurable core architecture based on standard cells for a fine-grain programmable fabric to better fit the available areas on SoCs and tools to automatically generate it. This architecture is generic for all types of digital designs applications and does not constrain the potential designs to be mapped on it on their directionality. The architecture and development flow that will be presented on the next chapter eliminates the problems with combinational loops and allows its basic component blocks to be generated as standalone fixed logic digital blocks, which together create the programmable circuit mesh.

Bringing FPGA concepts and architectures to ASIC solutions through fast and reliable design flows, using commonly available technology libraries and CAD tools will help to join the advantages both solutions have while mitigating the respective disadvantages.

Chapter 3

Embedded Programmable Logic Cores

A programmable logic core, devised and implemented to be integrated within a SoC, should be able to exist in different shapes and aspect ratios to better mingle with the existing fixed digital logic and other macros.

Inside a SoC, the space left after placing other hard macros, corresponding to analog and digital circuitry, is very often non-rectangular and shapes like "S", "L", "T" and "U" are commonly found on SoCs. A non-rectangular core of standard cell based reconfigurable logic can be used to fill that space, thereby adding to the final product the advantage of reconfigurability. These programmable cores can also exist with regular rectangular shapes within a SoC.

The example on figure 3.1 shows the non-rectangular programmable core as an array of cluster blocks embedded on a SoC. Each cluster block contains logic and routing circuitry and is replicated across the entire array. On the same figure, other hard macros and fixed logic digital blocks are visible. The fixed digital logic for the programmable core that occupies part of the area exists to support the core operation and reconfiguration. Following sections on this chapter detail the architecture of these clusters and the blocks that compose them.

3.1 Architectural Overview

The reconfigurable digital logic core architecture can be conceptually divided into two different groups of circuits. The first one is related with the memory and programming technology for configuration of the core and the second is the group of configurable circuits.

The memory portion of the core serves the purpose of holding the logic values that correspond to the digital logic functions mapped on the reconfigurable core together with

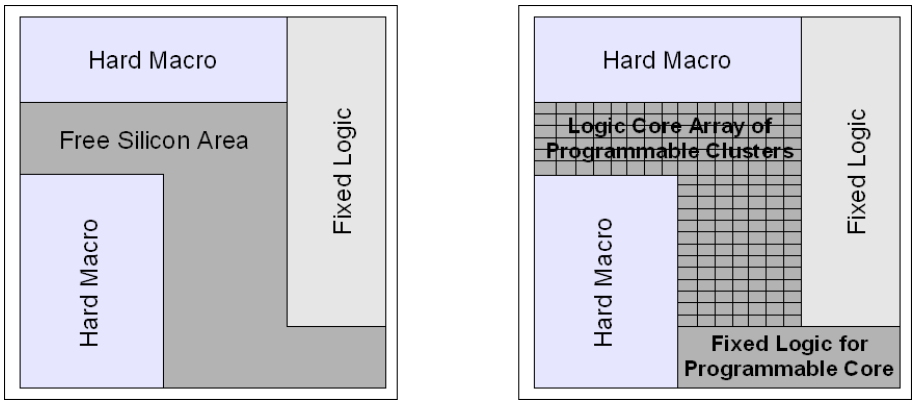


Figure 3.1: SoC and embedded programmable core.

the logic values that set the state of the switching and routing circuits. The group of configurable circuits supports the signals propagation and routing through the core.

The structures holding the programmed logic values need to be spread around the core, grouped close to logic blocks to store each logic function that composes the digital design mapped on the reconfigurable core and also together with each switch and selection path for the remaining circuits. This imposes that on an array of equally replicated cluster blocks, as shown on figure 3.1, each of those blocks need to contain both memory structures and configurable circuits. It is not possible to have a single external memory macro to hold all logic values that set the reconfigurable core to the desired digital logic function and configuration.

The configurable circuits, besides the multiplexing and routing logic, configured by the memory elements, correspond also to the LUT and the sequential logic to support sequential digital functions.

As will be described later on this section with more detail, the proposed architecture is composed by a logic block that includes a memory portion, the LUT structure and the needed sequential logic and by 2 additional blocks (routing and switch) that together compose the circuits needed for the signals propagation and routing. Each of these also includes memory blocks to hold the logic values that set the core configurations.

The architecture chosen and proposed is based on a typical symmetrical array FPGA also known as an island-style FPGA architecture. To allow for an easier integration with SoCs, the choice was made for the component blocks to respect a fine-grain approach. Based on standard cells, the programmable logic core programming technology makes use of flip-flop cells to hold the mapped logic design. The scan chains associated with typical digital logic scores will be the programming technology used to program the reconfigurable core. The hierarchical organization of the core and the knowledge of the circuits to be implemented guarantee that all data paths cross the same amount of logic allowing the reconfigurable core delay characteristics to be easy to estimate and bound.

The methodology followed allows to implement each basic block as a standalone block,

where there is no dependency on the external loads or input transition times. The approach followed also eliminates the combinational loops, common to FPGA architectures and that usually cause problems to ASIC tools. The flow used to implement the reconfigurable cores allows to quickly update or correct the core standard cells based basic blocks.

For the island-style architecture three different structures are required: logic, switch and routing blocks. Logic blocks are surrounded by routing blocks and where these channels intersect we have the switch blocks. These three basic blocks connect by abutment to form a more complex cell at a higher structural level, a cluster block. Clusters connect by abutment to other cluster blocks to form the reconfigurable logic core with the desired shape.

As will be described later with more detail, since each cluster is composed by a single 2- or 4-input logic block and associated routing and switching circuitry, the overall embedded programmable logic core architecture shows a fine-grain characteristic, due to the small cluster blocks, what is the most suitable to serve non-rectangular shapes. A coarse granularity, with cluster blocks sharing additional logic blocks and with it an additional number of circuits and used area, would result on a larger cluster block and less flexibility to integrate a reconfigurable core on a non-rectangular shape. Future results might show that the cluster block complexity and intrinsic area might be increased for added performance and routability of the reconfigurable core with no significant lost in flexibility of the core non-rectangular shape but for now, and as a simpler approach to prove the architecture proposed, a simple cluster block providing fine granularity is proposed.

The island-style architecture has a number of desirable properties. With routing tracks close to the logic blocks, efficient connections can be implemented. By setting the routing channels to span only the logic block length it allows for the blocks to be easily organized into the cluster tiles which can be replicated on a non-rectangular array. The regularity obtained with the architectural choice allows to quickly estimate delay between any two points on the mapped digital logic circuit.

The proposed general core architecture requires only a small set of standard cells for its implementation. Table 3.1 contains a list of the minimum set of sequential and combinational standard cells needed. It is visible that besides the regular D-type flip-flop for the sequential logic, there is also a requirement for a D-type flip-flop cell with scan input and enable signals to support the scan chain that will serve as the programming technology for the architecture proposed. This specific need is discussed in more detail in section 3.1.2.1. If these cells are not available, they can be created through the use of other cells (for example: a 2-input NOR cell can be built using a 2-input OR cell with an INV buffer).

By using standard cells, commonly available CAD tools can be used and the programmable core architecture can be physically implemented through a standard digital logic design flow. Circuit implementation and improvements can be quickly performed. For example, a buffer or a multiplexer cell can be quickly sized to different drive strengths

for a more robust block at a very early implementation stage. The advantage over a full custom layout flow, where the design is manually drawn, is strong and immediately visible.

Combinational	
INV	The INV cell provides the logical inversion of its single input
BUF	The BUF cell provides the logical buffer of its single input
TBUF	The TBUF cell provides the logical buffer of its single input with an active-high output enable
MUX2	The MUX2 cell is a 2-to-1 multiplexer and the state of the selection input signal determines which data input is presented to the output
MUX3	The MUX3 cell is a 3-to-1 multiplexer and the state of the selection input signal determines which data input is presented to the output
AND2	The AND2 cell provides the logical AND of two inputs
NOR2	The NOR2 cell provides a logical NOR of two inputs
Sequential	
DFFR	The DFFR cell is a positive-edge triggered, static D-type flip-flop with asynchronous active-low reset
SDFFR	The SDFFR cell is a positive-edge triggered, static D-type flip-flop with scan input, active-high scan enable, and asynchronous active-low reset

Table 3.1: Minimum set of standard cells needed for the reconfigurable core architecture.

If needed, special purpose cells can be designed and used together with the available standard cells to improve area or performance. For example, 2- and 3-input multiplexer cells represent a significant part of the data path on the architecture proposed. These cells can be implemented for smaller propagation delays, at the expense of additional used area, in order to speed-up the data path and with it the maximum possible operating frequency of the reconfigurable core. On a different complexity level, a complete logic block, as the ones proposed and documented on section 3.1.4, could be drawn as a single cell for reduced area and increased performance with shorter and optimized connections between the circuits that compose it. The work involved on creating these cells corresponds to drawing its layout in an optimized way, validating its functionality, characterizing its timing arcs and generate all needed views such that the cell could be used under a standard digital design flow to implement the programmable cores as if it would belong to a digital standard cells library.

For each of the typical FPGA circuits and structures that correspond to the logic, routing and switch blocks, an analysis was done over documented FPGA architectures to translate those circuits' functionalities into standard cells based circuitry that would implement the same functions. This work is described in more detail over section 3.1.2.

The use of standard cells, although being area-delay optimized cells, incur on area overhead when compared to typical FPGA structures. On the other hand, any digital cir-

cuit implemented with CAD tools is easier to create, correct and validate when compared to a custom design flow to create and integrate the FPGA structures.

For the proposed architecture, defining the number of data inputs to the logic block, determines the architecture for the routing and switch blocks as it defines the number of their input and output data signals. In this sense, the logic block architecture defines the routing and switch blocks that surround it. Throughout this document, the switch and routing blocks presented are always relative to a certain defined logic block architecture.

The following sub-sections start by describing the design flow used to implemented fixed logic digital cores with standard cells. Then the standard cells that were chosen to implement typical FPGA functionalities are presented. The cluster macrocell and the implementation details of the logic, routing and switch blocks follow.

3.1.1 Design Flow based on Standard Cells

Each standard cell implements a simple digital logic function. They are grouped and connected together to created complex digital logic functions.

This section describes the reasons why standard cells from commonly available standard cells libraries can be and were chosen for the programmable core implementation.

Standard cells are implemented following a full custom design flow and, in this sense, each standard cell is already optimized for area, power consumption and performance.

A fixed logic digital core is usually implemented in a standard digital design flow using the available standard cells for that purpose. This includes synthesis, place & route and behavioral and post-layout simulations with standard CAD tools. This automated flow is commonly known and characterized by a reliable and short design cycle. On a full custom flow, circuits are manually drawn and time consuming transistor-level simulations need to be done what leads to a development cycle that is longer and carries it more risk than the approach to implement digital cores using standard cells, which is the one chosen to support the reconfigurable cores creation.

Another advantage is that the designer doesn't need to customize or purchase any particular tool other than the ones that are already being used for the creation of the fixed logic digital cores.

With standard cells it is possible to quickly update or correct the digital cores created on a short and reliable time frame. As an example, it is extremely quick to change the drive strength of a certain standard cell being used and validate the digital core again. Such a change would be equivalent to change transistor sizes on a circuit implement on a full custom flow, what is more risky and requires more effort both to change and to validate the change.

One additional advantage related to the use of standard cells it the easier technology portability they allow, either on a different process node or within the same process node, under a different flavor. The process of creating the same standard cell based design for a

different technology is reliable and fast when compared to the effort needed to port to a different technology one design created in a full custom layout flow.

The circuits that will be presented on the following sections are implemented through the instantiation of standard cells that together hold the needed functionality to support the needs of a programmable logic core. This means that behind the reconfigurable core proposed, there is a group of fixed digital logic blocks. These blocks are connected together such that a programmable fabric exists to support, at a higher level, the core reconfigurability.

A programmable logic core created with standard cells incurs in area, delay and power overhead [23] when compared to the same type of programmable logic or circuitry implemented on a full custom flow [24, 25, 26, 27].

The use of optimized cells to create the proposed embedded reconfigurable cores using common CAD tools through a short and reliable design cycle is a strong motivation to use them.

3.1.2 FPGA Structures: Standard-cells Implementation

The first step towards the creation of the programmable circuitry is to understand what kind of circuits compose the programmable logic on typical FPGA solutions and implement those circuits with commercially available standard cells on common standard cells libraries.

In this sense, to support a standard-cell based programmable core, three main logic functionalities were identified: data storage, routing and disconnecting digital logic signals. Table 3.2 summarizes the meaning of each functionality that will be described in more detail on the following sub-sections.

Functionality	**Description**
Data Storage	Hold a logic value after programming
Routing	Propagate and route signals
Disconnecting	Retain signals to avoid contention by disconnecting them from a certain path

Table 3.2: Main FPGA logic functionalities.

3.1.2.1 Data Storage

The ability to store a bit logic value is present on FPGA structures typically through the use of SRAM memory cells. Each cell will hold a certain digital logic value after programmed. The SRAM cells will support the logic function truth tables on the logic blocks and hold the logic value for the remaining control signals that define the signals routing on the reconfigurable core. SRAM cells implementation is based on a bi-stable latching circuitry to store each bit. Simple and reliable SRAM circuits exist and can be

created with no more than six transistors. SRAM cells make part of the highest density FPGAs. An example of this simple circuit is visible on figure 3.2.

The SRAM read process is done by precharging both bitlines ($BL, \overline{BL}$) to a high logic value and then asserting the wordline (WL). One of the bitlines will be pulled down and the logic value hold on the SRAM cell will be read to the bitlines. To write a bit on the SRAM cell, one bitline is driven with a low logic value and the other one with a high logic value and then the wordline is asserted high. The values set on the bitlines will overpower the previous value and be kept on the SRAM cell.

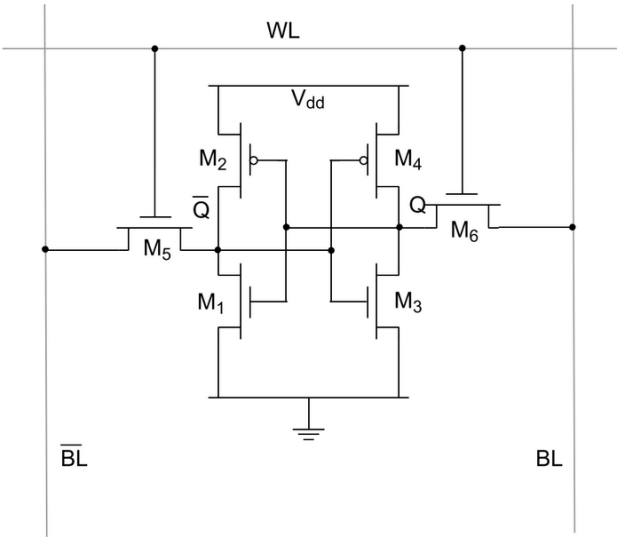


Figure 3.2: 6-transistors static random access memory (SRAM).

To replace the SRAM cell and have the same 1-bit memory function, a D-type FF cell with scan input and enable (SI, SE) and asynchronous active low reset (RN), available on standard cells libraries, can be used to implement this store function on the reconfigurable core and provide the auxiliary circuit for the programming of the core through the scan related signals it supports. As discussed on the architecture overview 3.1, scan chains will be the programming technology for the reconfigurable core proposed. If the standard cells library doesn't contain the FF with scan input and enable signals, this cell can be built from a standard FF together with a 2-input multiplexer cell. The regular FF data input will be the output of this multiplexer, where its inputs correspond to the data and scan data inputs and its selection path corresponds to the scan enable signal. On this case, an area overhead exist due to the addition of the needed extra combinational logic for the programming capability when compared to the already optimized scan-capable FF cell. Figure 3.3 presents the symbols commonly used to represent FF cells. These cell symbols, are here documented for reference since they will be present on the programmable core component blocks schematic diagrams.

These cells serve as a single bit memory like the SRAM cells. A clock signal (CLK) latches the data (D) or scan input (SI) and an asynchronous reset signal (RN) resets

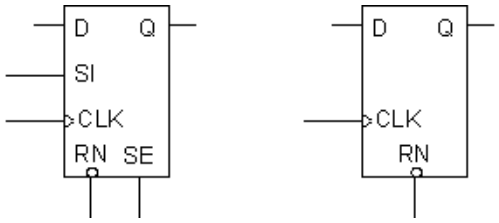


Figure 3.3: D-type FF with/without scan (SI, SE) and asynchronous active low reset (RN).

the cell causing its output to be asserted low. The scan related signals on this cell allows the CAD tools to easily create a scan chain between all programmable FF cells. The FF cells used don't need any special timing requirement so the smallest ones available on the standard cells library can and should be used.

Other FF cells exist in the reconfigurable core to support and produce the logic functions sequential outputs. These cells don't need to support the programmable chains and therefore, they shouldn't have the scan related circuits and signals the scan-capable FF has to avoid having these FFs on the programming chain (scan chain) what would cause problems to the architecture and automated tools proposed.

At some point on their development stage, both SRAM and FF cells are implemented on a full custom design flow but SRAM cells are not typically available as an optimized, ready to use cell, as the FF, commonly available on standard cells libraries. SRAM cells are difficult to size as several constraints must be fulfilled regarding the sizing of the transistors that compose it so that the bit is reliably kept inside this memory cell and not lost. Being implemented on full custom design flow it may require a complete sizing review for different processes/technologies while a FF is an optimized standard cell for a given technology.

The area occupied by a single bit memory cell such as the SRAM is smaller than the area occupied by a FF cell. The difference on the area used by each increases if the FF in consideration is, as required for the programmable core architecture proposed, a scan-capable FF cell. Across several different technologies and standard cells libraries, the implementation of a scan-capable FF cell varies on the number of transistors used. When compared to one possible implementation of the SRAM cell that is built using only 6 transistors (figure 3.2), the significant area overhead can go up to a 6x increase for FF cells that support scan capabilities as on average, these cells are implemented with around 36 transistors.

During normal mode operation, the FF cells used in the programming chain have their outputs tied to 0 or 1 and the cells are in a static point of operation holding the programmed high or low logic state. Dynamic power consumption occurs only during programming. When compared to SRAM cells, FF cells are expected to have higher leakage current consumption as their circuit has more transistors.

Both SRAM and scan FF cells require that programming is performed again after power-up. This means that external devices need to be used to permanently store the configuration when the device is powered down and this requirement adds to the overall cost of the programmable core. A well defined write mechanism (using SRAM bitlines and wordlines as shown above) exists to set the logic value on the SRAM cells. Using FFs, the logic value is set through a programmable chain that makes use of the FFs scan chains. The scan chains created through the digital design flow for a FF-based design can be used for this purpose. The memory cells can be configured through a serial interface but a dedicated programming circuit, decoder, can also be used to program the bits in a parallel fashion both for SRAM and FF cells. The advantage with using a FF-based look-up table is that scan chains are inexpensively created and validated when physically implementing the block where they are used since CAD tools on the digital design flow are well familiar with FF cells and their usage to support the creation of scan chains. Together with the scan chains, the clock and reset signals of the FF cells are also known and common to the digital design flow and respective CAD tools. The respective global networks are easily created during the block implementation.

3.1.2.2 Routing

The propagation and routing of signals within the programmable core occurs on the physical fabric mesh that exists to establish the signals path and on the circuits that are used to, within the core, restore the signal quality level, avoiding its degradation and respective impact on the design performance. The circuit responsible for the routing of signals is characterizes the routability efficiency of the programmable core as it corresponds to the configurable circuits that, when programmed, route the signals to the desired paths.

Within the programmable core, signals travelling from one point to the other need to be buffered throughout their propagation path to maintain their quality and avoid its signal degradation what could cause malfunction problems. To act as buffers and restore signal quality on the interconnect mesh of typical FPGA structures, implementations of buffer cells are used.

The signal multiplexing is achieved in typical FPGA architectures using analog switches such as NMOS pass-transistor switches and CMOS transmission gates. These cells act as switches that allow or interrupt the respective input from reaching its output. The disconnect function will be described in more detail in the next section. An example of a typical multiplexing scheme is a tree of NMOS pass-transistor switches, controlled by programmed memory cells. This circuit implements a multiplexer tree, as presented on figure 3.4. On this same example, each NMOS switch could be replaced by a CMOS transmission gate. Depending on the value set to $S1$ and $S0$ input signals, one of the IN input signals will propagate through the multiplexer tree till it reaches its output.

The example on figure 3.4 represents the circuit that is typically implemented through a full custom design flow. The propagation and multiplexing of signals on a programmable

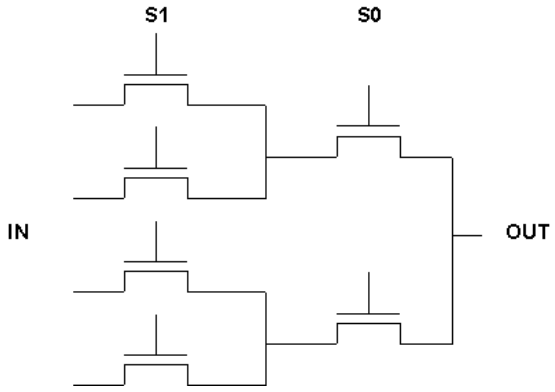


Figure 3.4: Multiplexer tree implemented with NMOS pass-transistors.

core circuit using standard cells is achieved through the use of combinational logic cells such as buffers, inverters, multiplexers and other combinational logic that one may want to add for increased complexity on the programmable core. The multiplexers selection signals can be set by the programmable FF cells and in this way, the multiplexers are used to route one input signal to its output. Using standard cells, the multiplexing and routing of signals is also achieved using three-state buffer cells, that besides propagating signals as simple buffer cells, are also able to interrupt them as will be presented on the next sub-section.

3.1.2.3 Disconnecting

Disconnecting digital signals is necessary to avoid drive line conflicts. This means isolating them from one line by a high-impedance node and preventing them from conflicting with other signals being driven by other logic circuit to the same line.

To disconnect the propagation of digital signals, keeping the signal disconnected from the path through a high-impedance node, switches are implemented on typical FPGA structures through the use of NMOS pass-transistors or CMOS transmission gates controlled by memory cells such as SRAM cells. A representation of these circuits is shown on figure 3.5. By acting on the transistors gate, the device can be enabled or disabled. When disabled, it interrupts the signal path. Both circuits allow other signals to be connected to the same physical lines as their output, avoiding line drive conflicts.

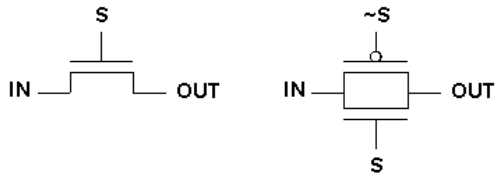


Figure 3.5: NMOS pass-transistor and CMOS transmission gate.

These pass-transistors, used on typical FPGA architectures, are not ideal switches as they have significant on-resistance and present an appreciable capacitive load. The circuit performance is clearly affected by the series impedance on the signal path and the voltage drop that occurs when the signal propagates through it. Additional circuitry needs to be added and used to restore the voltage level resulting in additional area being spent.

From a standard cells library, three-state buffers, whose standard cell typical symbol is presented on figure 3.6 for reference for other diagrams presented on this document, can be used for the same purpose and serve as switches on the programmable logic core. These buffer cells, as the transistor-level circuits described above, have the ability to leave its output as a high-impedance node and this way disconnect the signal from the line it was driving. The control signal for the three-state buffer can be set by a programmable FF cell. The logic value set to the three-state buffer enable signal (S) will cause this cell to operate as a standard buffer or to disconnect its input from reaching its output.

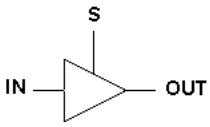


Figure 3.6: Three-state buffer.

3.1.3 Cluster Block

The cluster block is the macrocell that contains all circuits that implement all functionalities of the programmable core such that when instantiated together in arrays, create the reconfigurable circuit fabric. The basic component blocks, Logic Block (LB), Vertical Routing Block (VRB), Horizontal Routing Block (HRB) and Switch Block (SB), are integrated into a more complex macrocell identified as a cluster (figure 3.7) and each cluster cell contains a single instance of each of them. These basic blocks will be described in more detail in the following sections.

The signals involved in the cluster cell and related with the blocks that compose it can be divided into three groups: data, control and programming. The data signals correspond to both signals on the logic block input and output pins as well as the data signals propagated through the routing and switch blocks. The control group is composed by the clock, reset and programming mode enable signals and is available to all blocks. The programming signals correspond to the programming chain that is available on all blocks. Figure 3.8 presents these groups of signals and connections within and at the cluster interface.

A different number of logic blocks can be added to the same cluster block to increase the complexity of the digital logic functions a single cluster macrocell can implement. For the architecture proposed, a single logic block exists within the cluster block. Complex cluster macrocells allow for area and performance to be optimized as the paths between

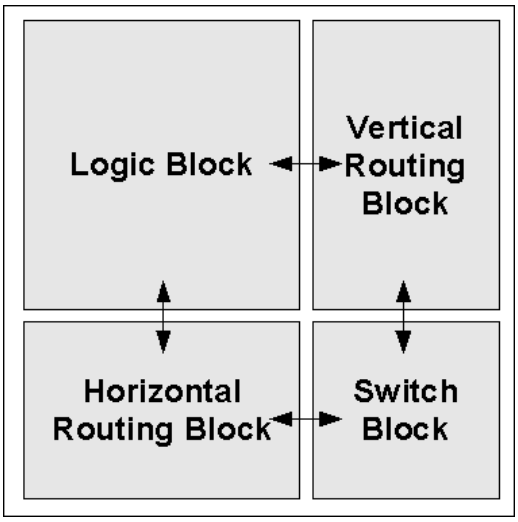


Figure 3.7: Cluster macrocell.

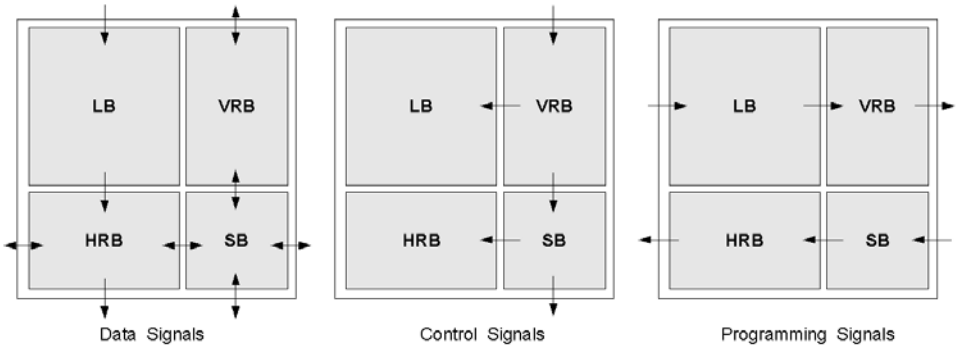


Figure 3.8: Cluster groups of signals.

the logic block on two different clusters are longer than the path between different logic blocks, or a more complex logic block, on the same cluster. This area optimization can be achieved if more than one instance of the basic blocks is included on the same cluster. For the implementation discussed here, the target is to create an intermediate hierarchical level to ease the core generation.

The proposed cluster cell integrates a single LB with a single 2- or 4-input LUT, one VRB, one HRB and one SB. As discussed before, the circuitry behind the switch and the routing blocks is always associated to the logic block circuit implementation as it depends on the number of input data signals of this block 3.1.

As shown on figure 3.8, the cluster has two isolated programming chains that correspond to the chain created by the LB and VRB and the chain created by the SB and HRB. These chains are not connected inside the cluster block. The organization of the programming chain on the cluster block could be different such that a single isolated chain would exist but, considering the cluster blocks instantiation to form the core, the programming chain input and output signals, set by the cluster blocks, would exist on opposite

edges, adding difficulty to the core integration. To avoid this, the chains were organized, through its definition on the basic blocks, such that the chain input and output signals are available on the same edge of the core to ease its integration.

The cluster cell can be physically implemented through the instantiation of each of the basic blocks that compose it. This can be achieved through the use of automated procedures developed within this work that instantiate them by abutment to create the core with the non-rectangular shape. Sub-section 3.2 will present this procedure in more detail. Each of the basic blocks should be physically implemented through a standard digital design flow.

One alternative is to create the cluster through a single run on a standard digital design flow, using for that purpose the gate-level netlists of the basic logic, routing and switch blocks. This approach will result in a cluster cell that is a single layout and instantiates only standard cells as opposed to the hierarchical cluster that instantiates the layout of the basic blocks. Having a single layout for the cluster cell can slightly optimize the total area of the cluster by bringing together in the same layout all the blocks referred.

The option taken is to create the separate layout for each of the basic blocks and then create the cluster block by instantiating them and connecting them by abutment. This choice was done as it allows to implement and validate each of the basic blocks as standalone fixed logic digital blocks and to avoid, at an early stage of development and creation of the architectures, adding complexity to the blocks verification. An additional advantage with the hierarchical organization chosen is that it allows updating or replacing a single block without having to change the other ones by redoing the complete cluster layout.

By connecting the cluster block by abutment to other cluster blocks, a programmable logic core of any shape can be created as, for example, in figure 3.9. The cells are the same for each physical position on the core. Cluster cells inside the core boundary will be surrounded by other cluster blocks while cluster blocks on the core boundary will face the other logic and macros on the SoC. Signals on clusters on the core boundary will be connected to these other macros being that one of these macros can simply be a fixed digital logic wrapper cell that surrounds the complete reconfigurable core.

The floorplan of the cluster block can be customized and this cell implemented on a square or rectangular shape as shown on figure 3.10. This adds one additional degree of flexibility to the programmable core as it allows for the same amount of logic to be created with a different shape and with that, an easier fit within the SoC.

3.1.4 Logic Block (LB)

The circuitry within this block is responsible for holding the digital logic function after the programming of the logic core. The LBs supported by the proposed architecture may contain one 2- or 4-input look-up tables.

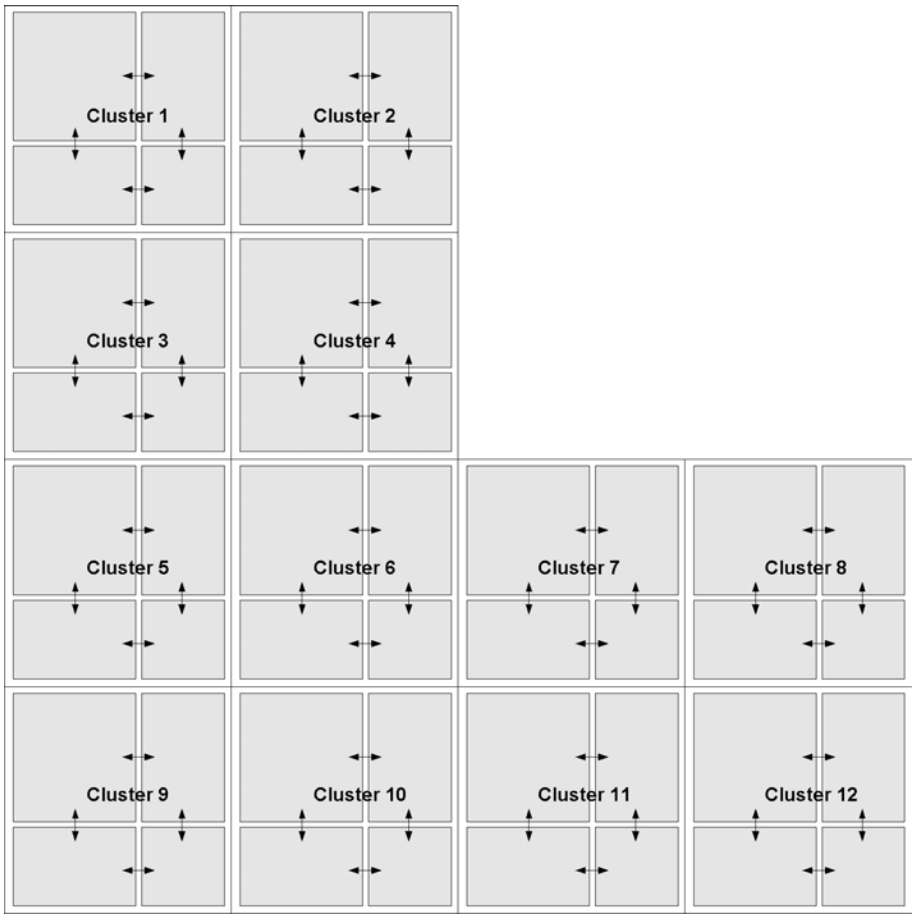


Figure 3.9: Non-rectangular programmable core as a group of cluster blocks.

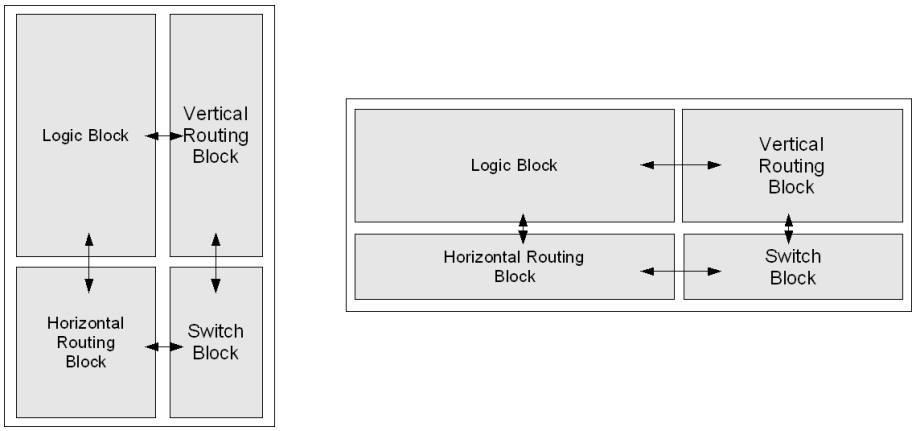


Figure 3.10: Clusters blocks with different shapes.

The lookup table implementation uses FF cells that can be programmed through a scan chain. The input data signals control the selection of the multiplexer tree that routes the look-up table outputs to the logic block's combinational and sequential outputs. The logic block sequential output is implemented by a non-scannable FF cell. This cell does not belong to the configuration chain.

A LUT is characterized by the number of data input signals. A small LUT will consume less area and be faster but, if small LUTs are implemented, more LUTs and with it more logic blocks, have to be used to implement a given digital logic function. Bigger LUTs can result in a waste of area in the sense that their logic will not be completely used when mapping small digital logic functions. For simplicity and proof of concept, the LUT proposed supports a 2- and 4-input LUT.

Based on a typical FPGA implementation, a N-input logic block requires 2^N 1-bit memory cells and a 2^N-input multiplexer. In this sense, a logic block containing a 2- or 4-input LUT requires 2^2=4 or 2^4=16 FF cells and a 2^2=4 or 2^4=16 input multiplexer respectively. As standard cells libraries don't usually support multiplexers with this high number of inputs, the choice was made to use 2-input multiplexer cells to implement a multiplexer tree according to the needs. In this case, a N-input LB will require 2^N-1 2-input multiplexer cells. The multiplexer tree can also be implemented using three-state buffers instead of multiplexer cells. If three state buffers are used to implement the multiplexer tree, a N-input LB requires 2^{N+1}-1 of these cells. This implementation requires more area than when using 2-input multiplexers due to the high number of three state buffers and requires more bits to control the state of all the three-state buffers that would be used. The use of 2-input multiplexers for the creation of the 2^N-input multiplexer was selected for the proposed architecture.

A single non-scannable FF cell is used for the sequential output generation on both architectures and additional combinational logic is used for control logic. If the output of the logic block is set to the combinational output, the clock for this FF cell can be gated to reduce the programmable core overall power consumption.

Pin	***Description***
in*	Data input signal - *=0,1 for 2-input LB, *=0,1,2,3 for 4-input LB
inbs	Programming bitstream data input
clk	Global clock input signal
pmode	Global programming mode enable input signal
rstz	Global reset input signal - active low
out	Data output signal
outz	Inverted data output signal
outbs	Programming bitstream data output

Table 3.3: LB pin description.

The LB input and output pins description is presented in table 3.3. As can be observed, the only signal that depends on the chosen architecture is the *in* signal, that corresponds to the data input signals. The symbol and a schematic representation of a 2-input LB is visible on figure 3.11. The circuit for a 4-input LB differs in the number of data input signals, programming FF cells and multiplexers on the multiplexer tree as describe above.

The LB output is routed through the multiplexer tree, according to the multiplexers selection signals set by the LB data input signals, to the combinational or sequential

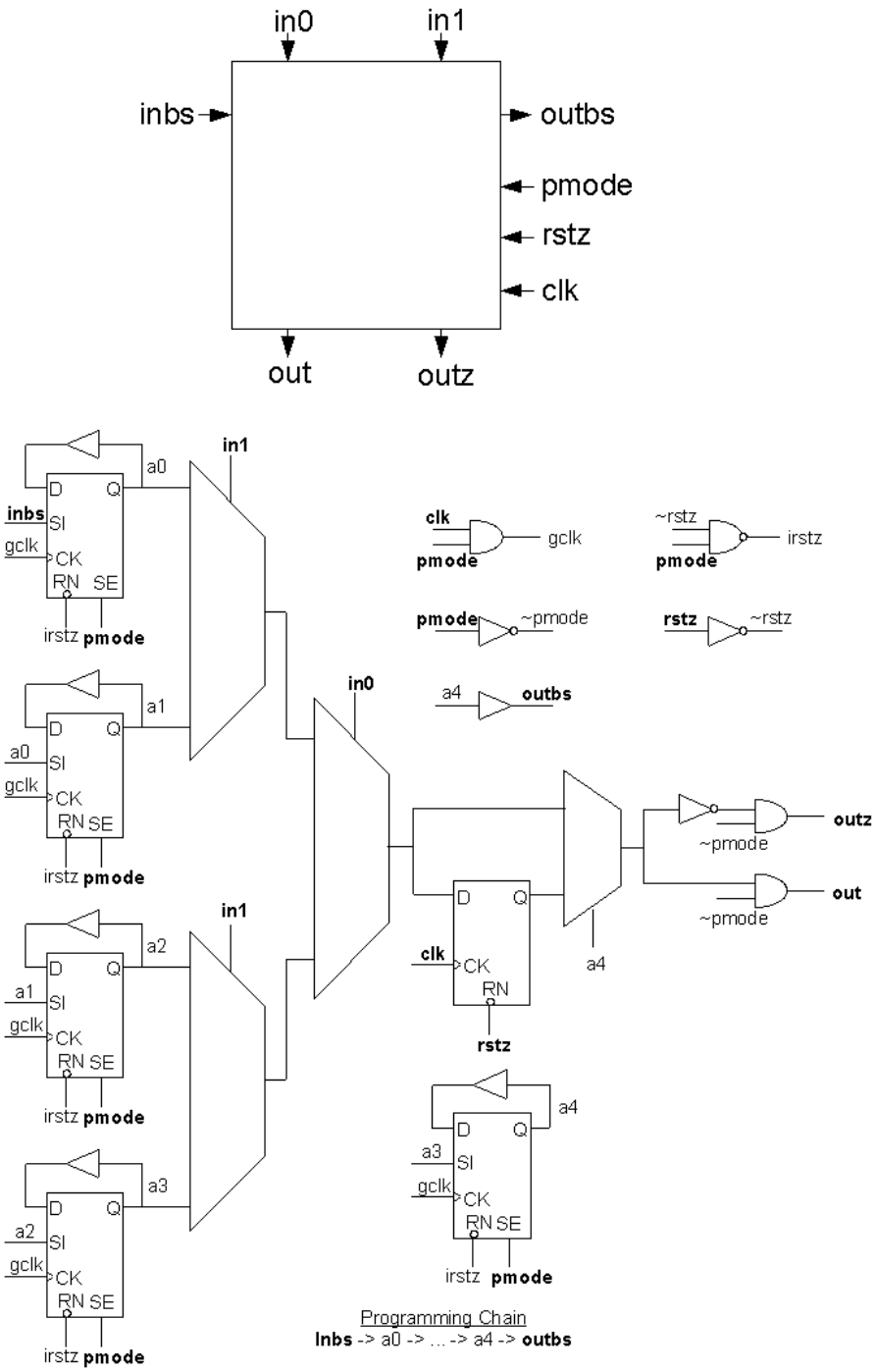


Figure 3.11: 2-input LB: symbol and schematic representation.

output, depending on the programmed output type.

In normal operation, the clock signal to the programming FF cells is disabled. This means that this clock signals is not toggling and it is gated with combinational logic. It will avoid additional power consumption, since in this mode the programming FFs only need to maintain their output logic value. When the core is in programming mode of

operation, the clock signal is again enabled and reaches the programming FF cells that will sample the input bitstream.

In normal operation, the global reset signal is applied only to the FF cell on the logic block responsible for the sequential data output so that the digital circuit mapped on the logic block can be reset without resetting the programmable FF cells and this way loosing the mapped digital function on the programmed circuit. The programmable FF cells, as well as the FF cell internally used for the sequential output, are reset when the reset global signal is asserted during programming.

During serial programming, FF cells are connected as a shift register through the scan chain implemented. Additional combinational logic ensures that it's sequential and combinational outputs do not toggle during this mode of operation.

Tables 3.4 and 3.5 show the relation, for a 2-input LUT, between the programmed logic values (*a0..3*) on each of the programming FF cells and the value of the output (*out*) and inverted output (*outz*) for each input data combination (*in0*, *in1*). The logic value programmed to the last FF cell on the programming chain (*a4*) defines whether the output of the logic block is sequential or not.

in0	***in1***	***out***	***outz***
0	0	a0	!a0
0	1	a1	!a1
1	0	a2	!a2
1	1	a3	!a3

Table 3.4: 2-input LB truth table.

a4	***out/outz***
0	Combinational
1	Sequential

Table 3.5: 2-input LB output selection.

For a cluster surrounded by other clusters, LB input data is provided by the HRB on the adjacent cluster, while its output data is connected to the HRB in the same cluster. The clock, reset and programming mode enable global signals are supplied to the LB through the VRB in the same cluster. The programming chain input comes from the VRB in the adjacent cluster and its output connects to the VRB in the same cluster. Figure 3.12 presents the logic block integration and connections within the same cluster and with adjacent clusters for the case of a 2-input LB.

To avoid adding complexity to the HRB and VRB the LB connects to, and since it is not possible to guarantee at this point any higher routability and performance of the overall core, the data signals on the LB were placed only on the block top and bottom edges instead of replicated around the complete block with connections to all routing

blocks surrounding it. Adding them would result on additional logic needed to control these paths and with it, more silicon area used.

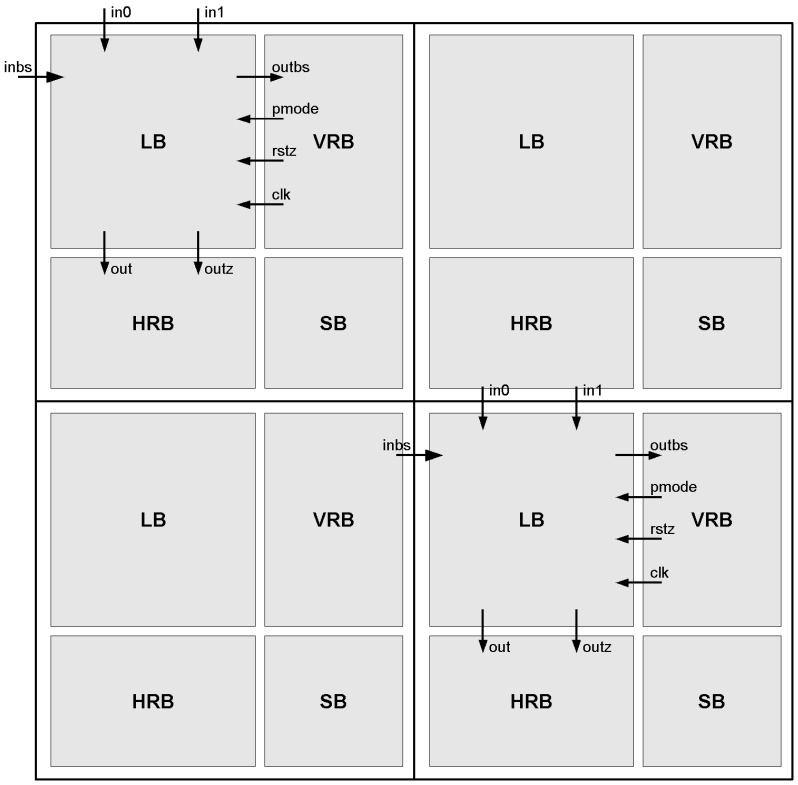


Figure 3.12: 2-input LB integration within same cluster and with adjacent clusters.

3.1.5 Routing Block

The routing circuitry in the configurable core implements the data, clock and control signals connections between all logic blocks and allows or blocks the propagation of any signal to potentially any core location. While data signal tracks will exist between switch and logic blocks, dedicated routing tracks exist for the propagation of clock, reset and programming mode enable signals to make them reach all blocks. These global networks are implemented with dedicated routing tracks which are separate from the configurable routing.

The routing block on the proposed architecture is composed by two different blocks: vertical and horizontal routing block. Both contain data local routing for data signals and additional segments for global signals propagation. The horizontal routing block is defined by the architectural choice of the LB that connects to it. It is the only block that connects to the LBs receiving and providing data connections to and from them.

The number of data tracks chosen for this architecture is equal to the added number of inputs and outputs of the respective logic block: 4 data tracks for a 2-input LB and 6 data tracks for a 4-input LB. Their length corresponds to the distance between SBs, spanning

one logic block before it terminates in a switch block. By enabling the programmable switches within the switch block, longer paths can be constructed. The advantage of using the same wire length on the same chip is that the number of switches a signal has to go through is predictable, and therefore also the delay of that signal will be predictable. Besides this, the wire length chosen allows the cluster blocks to be created as described before. Single wires longer than the cluster size would raise problems on the top-level reconfigurable creation or require additional different cluster blocks to exist.

Unidirectional tracks are used for an easier understanding and control when programming the logic core. With bidirectional tracks, additional control cells and memory cells to program this control would have to exist with the disadvantage of additional used area that would be needed for logic to control the direction of those tracks.

Each cluster contains one horizontal and one vertical routing block whose wires length correspond to, within the cluster, its length up to the SB. The routing blocks span one cluster block. This way, it is also easier to generate non-rectangular programmable cores through the abutment of cluster blocks as the routing blocks are contained within them and no special mechanism needs to be implemented to force the tracks to end on a certain cluster to avoid affecting the core shape. This would occur if the routing blocks would span more than one cluster block.

3.1.5.1 Horizontal Routing Block (HRB)

Part of the complete routing circuit corresponds to the horizontal routing block (HRB). For a N-input LB, N+2 horizontal tracks are available on the proposed architectures. The horizontal routing block (HRB) is this way composed by 4 horizontal unidirectional tracks (when a LB with a 2-input look-up table is used) or by 6 horizontal unidirectional tracks (when a LB with a 4-input look-up table is used). The unidirectional tracks are buffered for signal integrity and a mechanism using three-state buffer cells, one on each track, was implemented to control the drive of the horizontal tracks by the LB output signals and to disconnect the tracks from the SBs connecting to them to avoid conflicts. The control of the three-state buffers on the horizontal tracks depends, through this circuit, from the logic value on the enable signals of the three-state buffers driving the LB output data signals to the horizontal tracks. If no LB data output is being driven to one of the horizontal tracks, then the SB on the same or adjacent cluster is driving that same track.

The access from the LB output signals on the same cluster to the horizontal tracks is implemented using programmable three-state buffers, one for each LB output signal. The three-state buffers enable signals are controlled through FF cells. Since the number of LB output signals is the same for both proposed architectures, this part of the circuit is also the same.

The access to the LB input pins on the adjacent cluster from the horizontal tracks is implemented through a multiplexer tree whose selection is controlled by programmable FF cells. For a N-input LB, a N+2 input multiplexer and respective control is used for each LB

input signal. This multiplexer cell can be broken into 2- and 3-input smaller multiplexer cells that can be used from the standard cells library to form the needed multiplexer tree. A different multiplexer tree is required to accommodate 4 and 6 tracks for a 2- and 4-input architecture respectively and one for each LB input pin on the adjacent cluster must be instantiated.

The HRB is implemented such that the data outputs from the LB on the same cluster can be propagated to the LB on the adjacent cluster through the HRB. This way a path is also established between 2 two LBs without the need to propagate the signal through SBs and additional HRBs and VRBs. This path corresponds to the shortest path between two LBs on the proposed architectures.

Pin	***Description***
out_input	LB data output signal - HRB input
outz_input	LB inverted data output signal - HRB input
in*_output	Data output signal - LB input - *=0,1 for 2-input LB, *=0,1,2,3 for 4-input LB
inbs	Programming bitstream data input
clk	Global clock input signal
pmode	Global programming mode enable input signal
rstz	Global reset input signal - active low
outbs	Programming bitstream data output
track*_left_right_input	Data input, *=0,2 for 2-input LB, *=0,2,4 for 4-input LB
track*_right_left_input	Data input, *=1,3 for 2-input LB, *=1,3,5 for 4-input LB
track*_left_right_output	Data output, *=0,2 for 2-input LB, *=0,2,4 for 4-input LB
track*_right_left_output	Data output, *=1,3 for 2-input LB, *=1,3,5 for 4-input LB

Table 3.6: HRB pin description.

Table 3.6 presents the HRB pinout description for both architectures. As can be observed, the number of *in_output* data outputs is different to match the number of LB data inputs and the number of horizontal tracks is also different according to what was already presented. The symbol and schematic implementation of a HRB implemented for a 2-input LB is shown on figure 3.13.

As for the logic block, the clock signal to the programming flip-flops is disabled during normal operation and enabled during configuration. The programmable flip-flop cells are reset only when reset signal is asserted in programming mode of operation.

HRB programmability is separated in two different parts. One is related to the horizontal track data signals and the connections to the LB in the adjacent cluster. The other is related to the LB outputs and their connections to the horizontal tracks in the same cluster. Both sections are described on tables 3.7 3.8 respectively for a HRB implemented for a 2-input LB. It is not possible to connect both LB *out_input* and *outz_input* output signals to the same horizontal track as that would cause a drive conflict on the same data line. Taking table 3.8 into consideration, this means that *a4..a7* and *a5..a11* cannot be set

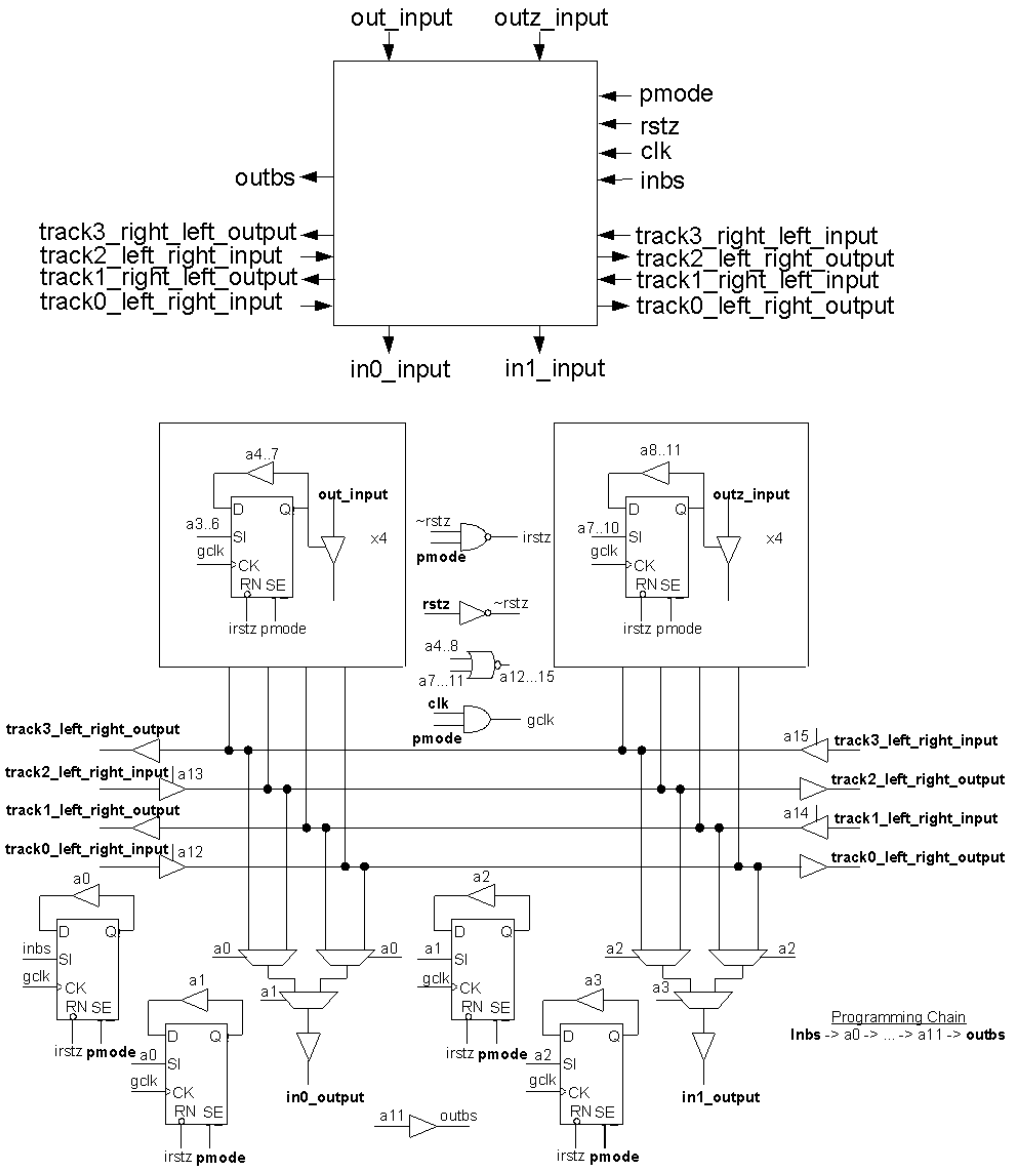


Figure 3.13: HRB for a 2-input LB: symbol and schematic implementation.

to a high logic value at the same time. This programming constraint can be automatically verified by any tool parsing and checking the bitstream used to configure the core.

For a cluster surrounded by other clusters in the programmable core, the HRB input data is provided by the LB and SB in the same cluster and by the HRB on the adjacent cluster while its output data is connected to the SB in the same cluster and to the HRB and LB in the adjacent clusters. The clock, reset and programming mode enable global signals are supplied to the HRB through the SB in the same cluster. The programming chain input comes from the SB in the same cluster and its output is connected to the SB in the adjacent cluster. The HRB integration of a HRB implemented for a 2-input LB within its cluster block and with the cluster blocks connecting to it is show on figure 3.14.

a0	**a1**	**a2**	**a3**	**track to *in0_output***
0	0	X	X	track0_left_right
0	1	X	X	track1_right_left
1	0	X	X	track2_left_right
1	1	X	X	track3_right_left
a0	**a1**	**a2**	**a3**	**track to *in1_output***
X	X	0	0	track0_left_right
X	X	0	1	track1_right_left
X	X	1	0	track2_left_right
X	X	1	1	track3_right_left

Table 3.7: HRB truth table for a 2-input LB: LB input signals.

a4	**a5**	**a6**	**a7**	***out_input* to track**
1	0	0	0	track0_left_right
0	1	0	0	track1_right_left
0	0	1	0	track2_left_right
0	0	0	1	track3_right_left
a8	**a9**	**a10**	**a11**	***outz_input* to track**
1	0	0	0	track0_left_right
0	1	0	0	track1_right_left
0	0	1	0	track2_left_right
0	0	0	1	track3_right_left

Table 3.8: HRB truth table for a 2-input LB: LB output signals.

3.1.5.2 Vertical Routing Block (VRB)

The other block that composes the programmable logic core overall routing circuitry is the vertical routing block (VRB).

Like the HRB, it supports N+2 unidirectional tracks for a N-input LB. This accounts for 4 or 6 unidirectional buffered tracks for data propagation, for a 2- or 4-input LB respectively. On the VRB, the data signals are just propagated through the block as there is no other circuit driving them.

In addition, the VRB is used to feed the cluster with the clock, reset and programming mode enable global signals. Being responsible for supplying these global signals for the LB and SB on the same cluster, strong buffers should be used on these signals path to maintain a strong signal for each cluster through the entire core.

The pin description for the VRB is shown on table 3.9. A representation of a VRB implemented for a 2-input LB can be observed on figure 3.15. As explained above, for a 4-input LB, the only difference on the circuit is that the number of data tracks is increased to match the number of data tracks of that architecture.

The VRB input and output signals are connected to the SB in the same and adjacent clusters. The clock, reset and programming mode enable global signals are supplied to the VRB by the SB in the adjacent cluster. The programming chain input is supplied by the

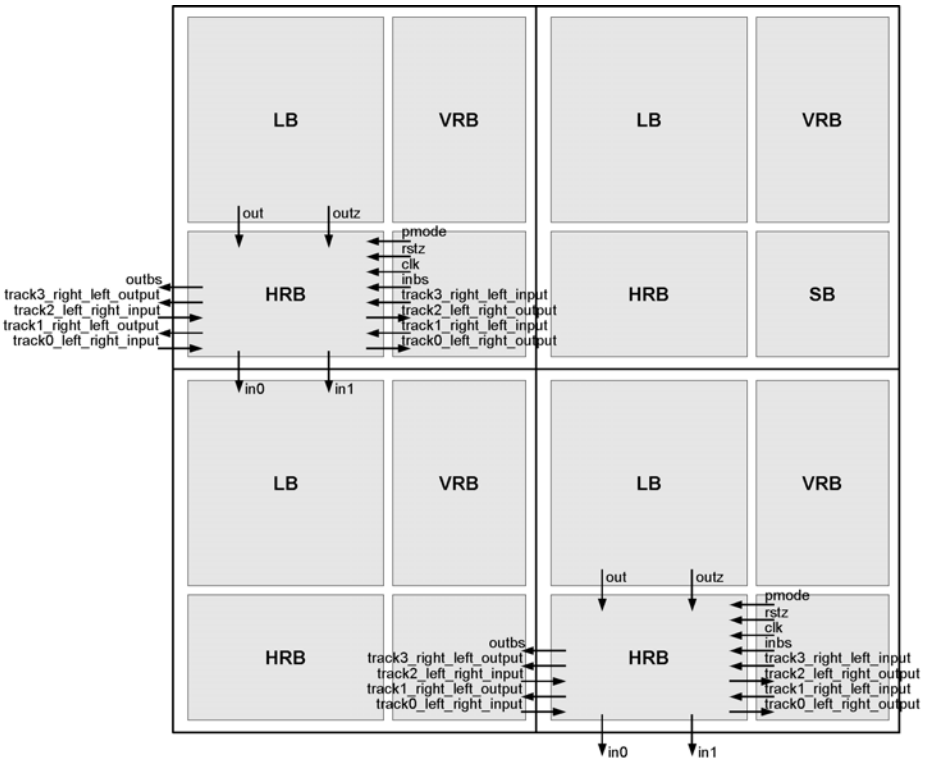


Figure 3.14: 2-input HRB integration within cluster and adjacent clusters.

Pin	***Description***
inbs	Programming bitstream data input
clk	Global clock input signal
pmode	Global programming mode enable input signal
rstz	Global reset input signal - active low
clk_out_lb	Global clock output to LB
pmode_out_lb	Global programming mode enable output to LB
rstz_out_lb	Global reset output to LB - active low
clk_out_sb	Global clock output to SB
pmode_out_sb	Global programming mode enable output to SB
rstz_out_sb	Global reset output to SB - active low
outbs	Programming bitstream data output
track*_bottom_top_input	Data input, *=0,2 for 2-input LB, *=0,2,4 for 4-input LB
track*_top_bottom_input	Data input, *=1,3 for 2-input LB, *=1,3,5 for 4-input LB
track*_bottom_top_output	Data output, *=0,2 for 2-input LB, *=0,2,4 for 4-input LB
track*_top_bottom_output	Data output, *=1,3 for 2-input LB, *=1,3,5 for 4-input LB

Table 3.9: VRB pin description.

LB in the same cluster and its outputs is supplied to the LB in the adjacent cluster. This integration, for a VRB created to work with a 2-input LB, is presented on figure 3.16.

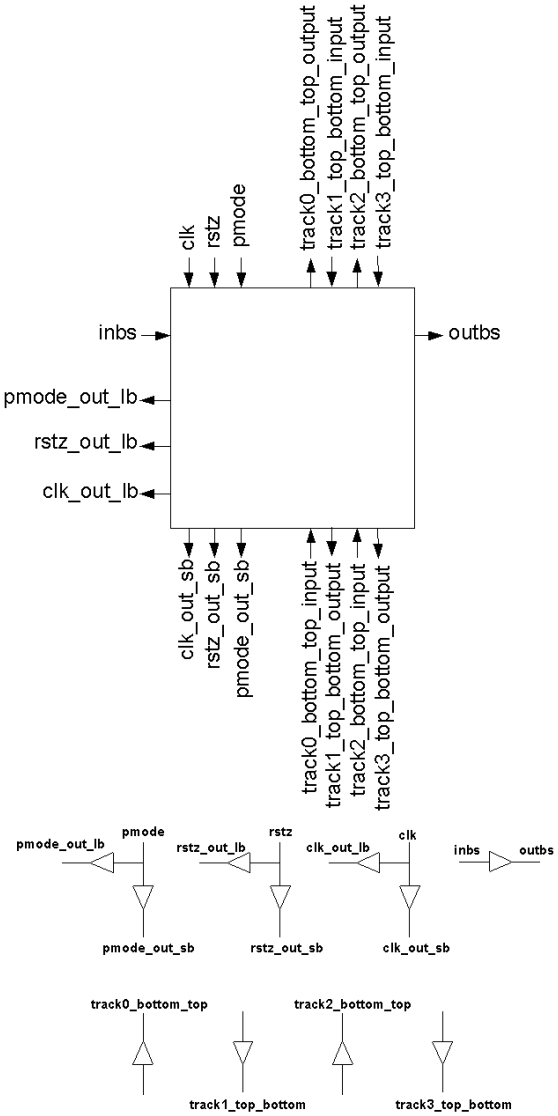


Figure 3.15: VRB for a 2-input LB: symbol and schematic implementation.

3.1.6 Switch Block (SB)

The horizontal and vertical tracks within the respective routing blocks intersect on the switch block in the same cluster. The switch block, besides propagating the clock, reset and programming mode enable global signals, is responsible for supporting the programmable connection between tracks on the same and other cluster blocks connected to it, routing them across the programmable core.

Several different topologies exist such as *Universal* [17, 18], *Disjoint* [19] and *Wilton* [20] among others. An example of these three topologies can be found on figure 3.17 where the possible paths considering pin 1 on the left edge as the input signal are highlighted.

On the *Disjoint* switch implementation, each pin is connected to each pin with the same number of the other three sides of the switch block. These connections establish a

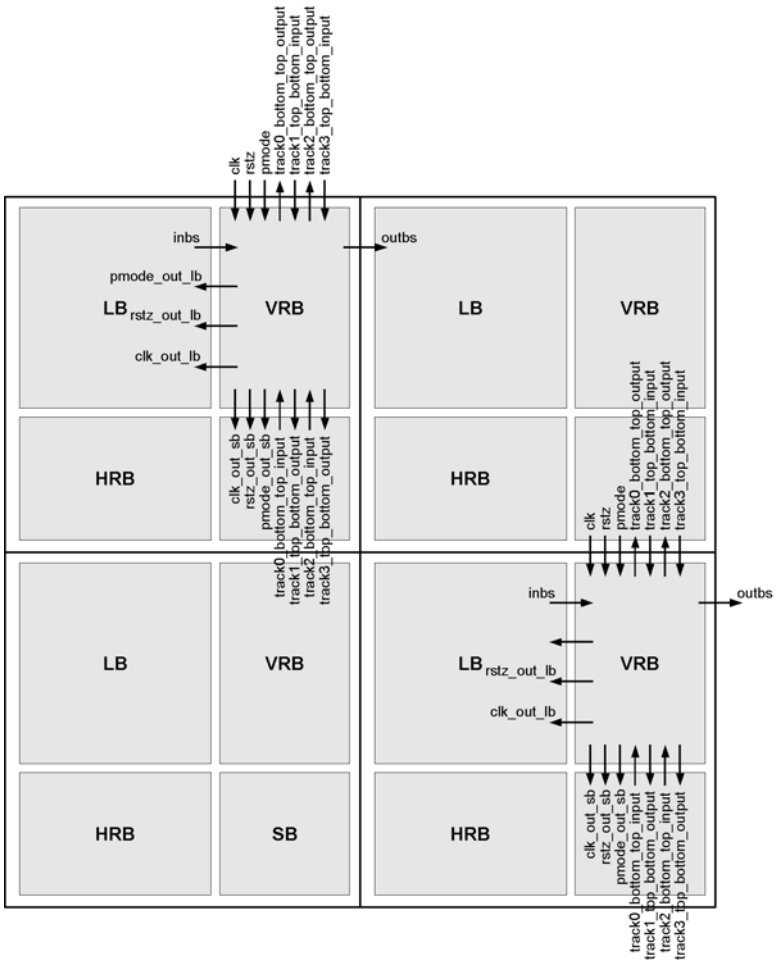


Figure 3.16: 2-input VRB integration within cluster and adjacent clusters.

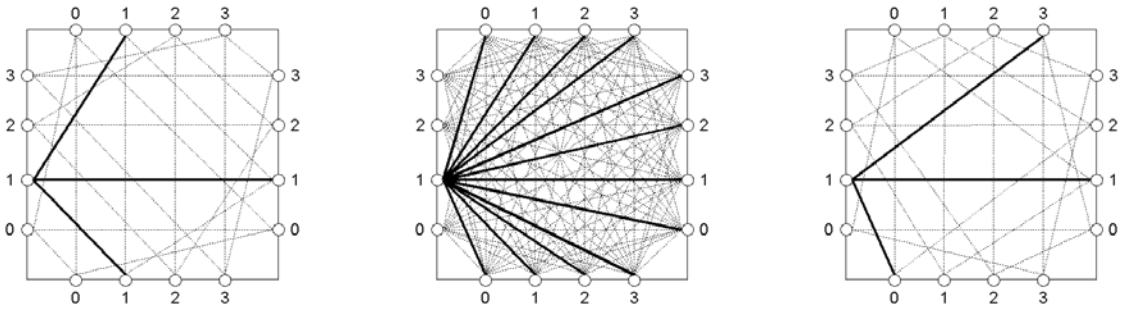


Figure 3.17: *Disjoint*, *Universal* and *Wilton* switch topologies.

symmetric pattern along the diagonal of the switch block. The result of this approach is that some paths are isolated into their routing domains, limiting the programmable core routing flexibility.

Other implementation is the *Universal* switch circuit. On this case, diamond patterns are created among the diagonals and the maximum number of simultaneous connections among the routing tracks is maximized. The *Universal* switch represents a significant

area overhead when compared to other implementations due to the maximum number of connections it supports.

The *Wilton* switch, chosen for the programmable logic core, is similar to the *Disjoint* implementation but here the diagonal connections are rotated by one track what allows data signals crossing the switch to cross to other tracks instead of staying on the same track, as the *Disjoint* switch implements. The ability to overcome the routing domains limitation as on *Disjoint* switch in at least one direction facilitates routing. According to [51], the *Wilton* switch is the one that presents the best area-efficiency for architectures where the routing tracks span only one logic block as the architecture proposed on this work.

On [20], the *Wilton* switch is presented and described. This switch block is represented by a graph $M(T, S)$ where each node in T represents a terminal (incident track) and each connection in S represents a programmable switch that connects two terminals. Each side of the switch block corresponds to a subset of T and each with W terminals. Each terminal is labeled $t_{m,n}$ where m corresponds to the side of the switch subset ($0 \leq m \leq 3$) and n corresponds to the terminal number ($0 \leq n \leq W-1$). The set of connections is therefore defined as:

$$
\begin{aligned}
S \;=\; & \bigcup_{i=0}^{W-1} \{\; (t_{0,i}), (t_{2,i}) \;,\; (t_{1,i}), (t_{3,i}) \;,\; (t_{0,i}), (t_{1,(W-i)\ mod\ W}) \;, \\
& (t_{1,i}), (t_{2,(i+1)\ mod\ W}) \;,\; (t_{2,i}), (t_{3,(2W-2-i)\ mod\ W}) \;,\; (t_{3,i}), (t_{0,(i+1)\ mod\ W}) \}
\end{aligned}
$$

Figure 3.18 presents the relation between each terminal and the physical pins on the switch block.

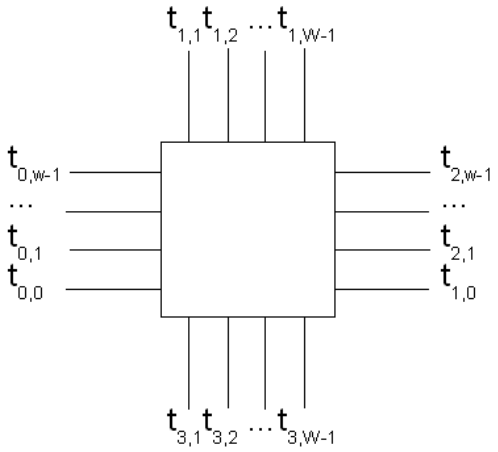


Figure 3.18: *Wilton* switch block: terminals and pins.

The equation described above shows all possible connections between pins on the proposed *Wilton* switch block. Since the proposed architecture has unidirectional tracks and

this switch block exists where these vertical and horizontal tracks intersect, the possible connections were divided to respect each track single direction.

The *Wilton* switch possible input and output connections for the reconfigurable core architecture are detailed on figure 3.19 for a SB implemented for a 2-input LB with 4 unidirectional tracks on each edge of the SB. For each input signal, the 3 possible output signals are highlighted.

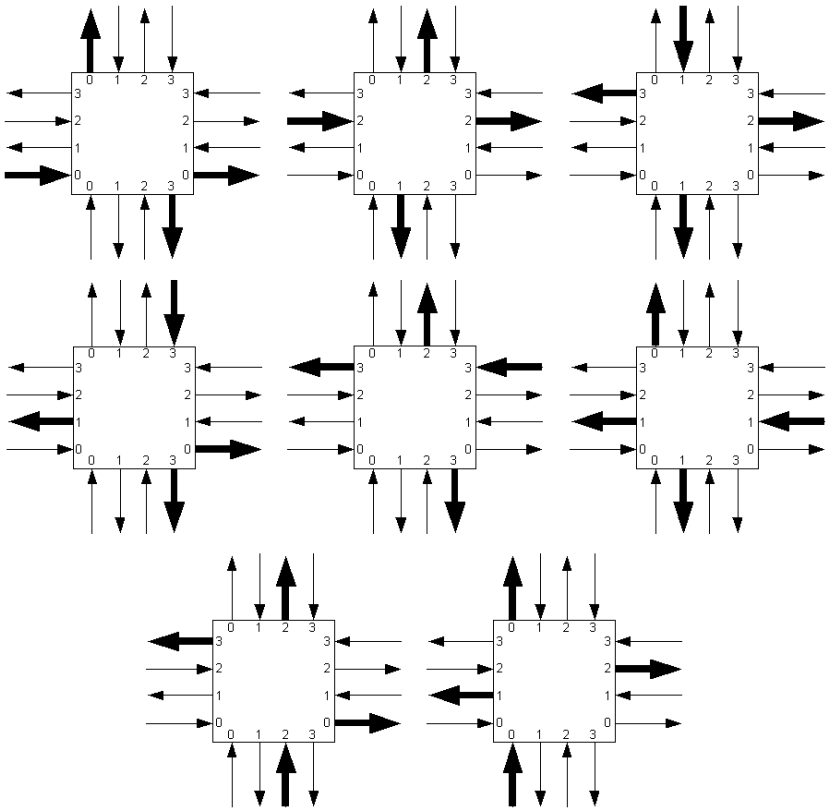


Figure 3.19: SB for a 2-input LB: *Wilton* switch input and output signals.

The *Wilton* switch is implemented through the use of 3-input multiplexer cells whose selection is controlled by two programmable FF cells. This circuit is the same for both architectures. For a N-input LB, N+2 horizontal and vertical tracks intersect on the SB. The number of instances of this circuit (3-input multiplexer cell and two FF cells) is 2N what results in 8 and 12 circuit instances for a 2- and a 4-input LB respectively. On this last scenario, the number of 3-input multiplexer cells and respective programming circuit is increased to accommodate the additional number of data tracks.

The pin description of a SB is presented on table 3.10.

One important parameter that characterizes the switch is its flexibility. By flexibility it is understood the amount of switches connected to each terminal. Previous studies like what is presented on [52] have shown that the routability is not significantly improved if

Pin	***Description***
inbs	Programming bitstream data input
clk	Global clock input signal
pmode	Global programming mode enable input signal
rstz	Global reset input signal - active low
clk_out_hrb	Global clock output to HRB
pmode_out_hrb	Global programming mode enable output to HRB
rstz_out_hrb	Global reset output to HRB - active low
clk_out_vrb	Global clock output to VRB
pmode_out_vrb	Global programming mode enable output to VRB
rstz_out_vrb	Global reset output to VRB - active low
outbs	Programming bitstream data output
track*_bottom_top_input	Data input, *=0,2 for 2-input LB, *=0,2,4 for 4-input LB
track*_top_bottom_input	Data input, *=1,3 for 2-input LB, *=1,3,5 for 4-input LB
track*_bottom_top_output	Data output, *=0,2 for 2-input LB, *=0,2,4 for 4-input LB
track*_top_bottom_output	Data output, *=1,3 for 2-input LB, *=1,3,5 for 4-input LB
track*_left_right_input	Data input, *=0,2 for 2-input LB, *=0,2,4 for 4-input LB
track*_right_left_input	Data input, *=1,3 for 2-input LB, *=1,3,5 for 4-input LB
track*_left_right_output	Data output, *=0,2 for 2-input LB, *=0,2,4 for 4-input LB
track*_right_left_output	Data output, *=1,3 for 2-input LB, *=1,3,5 for 4-input LB

Table 3.10: SB pin description.

this number is over 3 switches per terminal what correlates with the choice made over the *Wilton* switch for the proposed architecture.

The rest of the circuit is composed by connections to propagate the global signals to the adjacent blocks and combinational logic for control as present on the blocks already presented. Besides the 3-input multiplexer switches, these circuits are also visible on figure 3.20. This figure presents the symbol and schematic implementation of a SB implemented for a 2-input LB.

This block implements the same clock scheme as described before to avoid the additional power consumption when in functional mode of operation. It also implements the same circuit to reset the programmable flip-flop cells only when in programming mode of operation.

The connections between horizontal and vertical tracks that the switch block supports are detailed in table 3.11 for a SB implemented to work with a 2-input LB. The configuration values identified as not allowed on this table refer to the 3-input multiplexer selection signals that cannot be set to a high logic value at the same time.

The SB input and output signals are connected between the HRB and VRB in the same and adjacent clusters. The clock, reset and programming mode enable global signals are supplied to the SB through the VRB in the same cluster and the SB provides these global signals to the VRB in the adjacent cluster and the HRB on the same cluster. The programming chain input is supplied by the HRB in the adjacent cluster and its output

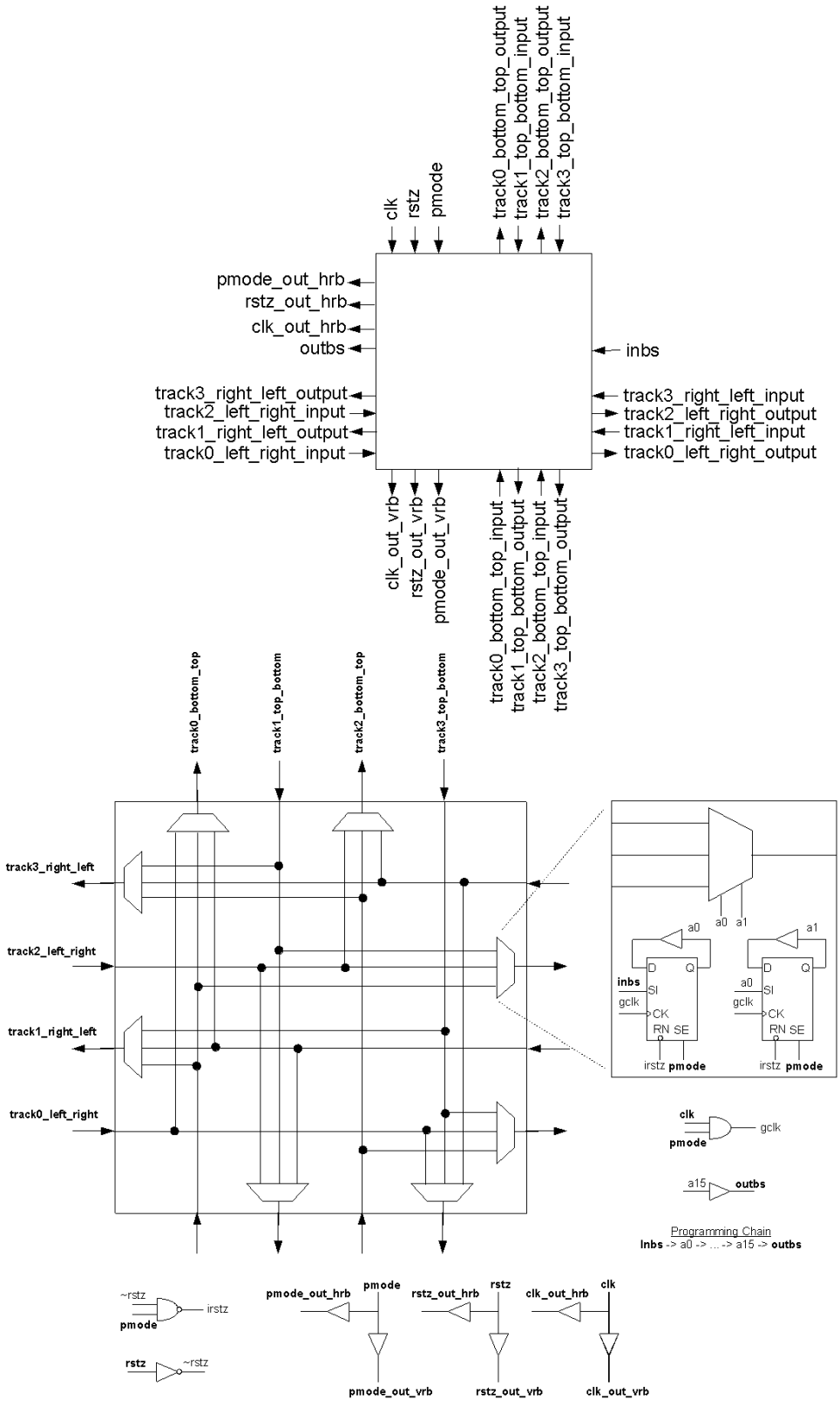


Figure 3.20: SB for a 2-input LB: symbol and schematic implementation.

track0_left_right_output			***track0_bottom_top_output***		
a1	**a2**	**input track**	**a3**	**a4**	**input track**
0	0	track0_left_right	0	0	track0_left_right
0	1	track2_bottom_top	0	1	track0_bottom_top
1	0	track3_top_bottom	1	0	track1_right_left
1	1	not allowed	1	1	not allowed
track3_top_bottom_output			***track1_top_bottom_output***		
a5	**a6**	**input track**	**a7**	**a8**	**input track**
0	0	track0_left_right	0	0	track2_left_right
0	1	track3_right_left	0	1	track1_right_left
1	0	track3_top_bottom	1	0	track1_top_bottom
1	1	not allowed	1	1	not allowed
track2_left_right_output			***track2_bottom_top_output***		
a9	**a10**	**input track**	**a11**	**a12**	**input track**
0	0	track2_left_right	0	0	track2_left_right
0	1	track0_bottom_top	0	1	track2_bottom_top
1	0	track1_top_bottom	1	0	track3_right_left
1	1	not allowed	1	1	not allowed
track3_right_left_output			***track1_right_left_output***		
a13	**a14**	**input track**	**a15**	**a16**	**input track**
0	0	track1_top_bottom	0	0	track3_top_bottom
0	1	track2_bottom_top	0	1	track0_bottom_top
1	0	track3_right_left	1	0	track1_right_left
1	1	not allowed	1	1	not allowed

Table 3.11: SB truth table for a 2-input LB.

connects to the HRB in the same cluster.

3.1.7 Other Architectural Notes

In each of the shown implementations, buffer cells are added to the respective input and output signals of the block to isolate them from the adjacent blocks and connections. Since the input transition times and output capacitance loads can vary significantly during the normal operation of the programmable core due to the different designs that can be mapped to it (and the different created connections each design involves), using strong buffer cells on the blocks interface allows each basic block to be implemented and fully validated for a wider range of external conditions as it makes the block less prone to timing variations.

On the proposed architecture, the implementation with standard cells results on a minimum fanout of the cells used. For this reason, the capacitance loads that the cells will have to drive will also be minimum and with this, higher operating frequencies can be targeted or, if the operating frequencies targeted for the designs on the reconfigurable core are low, smaller and slower cells can be used to create the core.

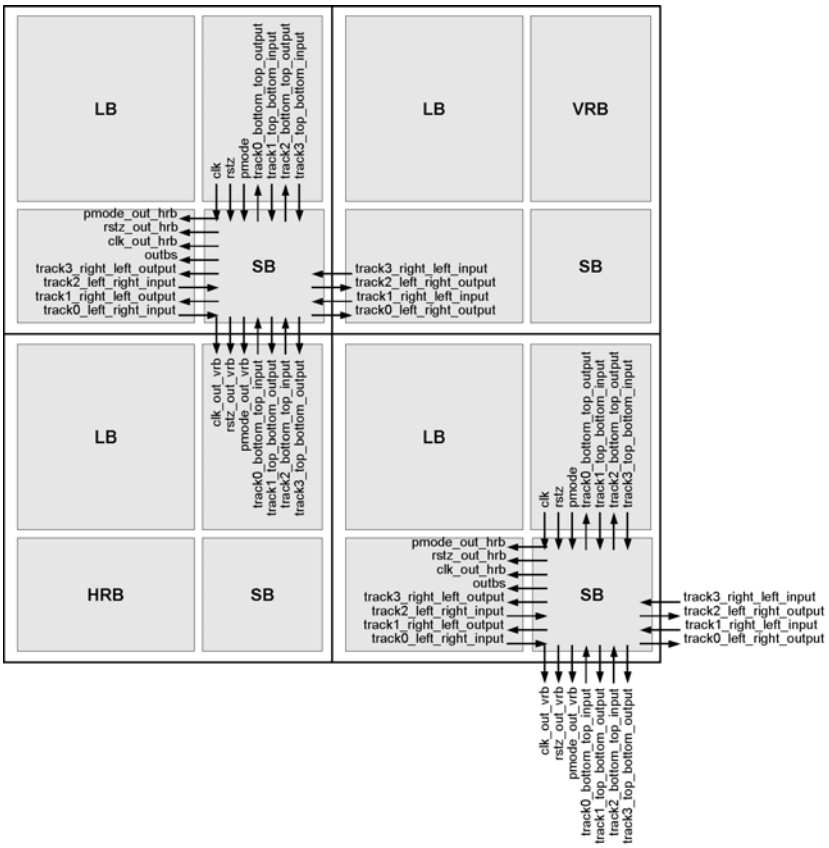


Figure 3.21: 2-input SB integration within cluster and adjacent clusters.

The clock, reset and programming mode enable signals are global signals that exist in the entire circuit, propagated to all blocks by their routing and switch blocks. Care has to be taken to ensure a reliable and uniform distribution for all FFs. On a standard-cell based design, the clock and reset trees are automatically synthesized and evaluated by the tools commonly used on a digital design flow. This way, a strong and symmetric clock and reset tree is guaranteed for all blocks.

The programmable core circuitry should be able to support a mechanism to set the programming FF cells into a know state as well as setting the digital logic function mapped to the programmable core into a known state without clearing the programmed FF cells. On a standard-cell based design, this is achieved with the reset functionality these cells support and with the logic schemes described before.

The use of FF cells allow to easily create a serial programming chain, during each block implementation, by connecting them together in the form of a shift register. As commented before, FF cells with scan capabilities should be used. If not available on the technology library under which the programmable core will be created, a similar circuit can be implemented through the use of a regular FF cell and a 2-input multiplexer cell.

The flexibility of the architecture proposed is such that different blocks structures can be quickly accommodated. These can include, for example, additional data inputs

on the LBs to hold complex digital logic functions, LBs without the sequential portion of the circuit or with additional FF cells for double-registering signals. On the routing channels, the density of tracks can be increased on the complete core or just on part of it. This includes the support of non-uniform routing channels. The SB circuitry can be updated to accommodate a possible different number of tracks on the SB edges and to support additional switches per terminal pin. Besides these possible changes that target the basic blocks within the cluster block, specific and dedicated macrocells such as adders, shift-registers and register banks, for example, with sizes spanning a multiple number of cluster blocks, can be included on the reconfigurable core for increased performance and functionality.

3.2 Reconfigurable Core Creation and Verification

The use of reconfigurable cores should be supported by a set of tools that allow the designer to quickly create a new core for any given architectural choice, area, shape and technology. These tools should also support provide data for an initial validation of the generated core.

A software tool, *eplcGen*, was created to automatically generate a gate-level netlist of the reconfigurable core, a testbench with basic verification tasks suited for that core and the basic blocks and core floorplan information as a preparation for the layout views creation. The physical implementation of each component must be separately created through a standard digital design flow, making use of the gate-level netlists automatically created and the respective floorplan data.

The following sections describe the *eplcGen* tool and work flow in more detail.

3.2.1 Automatic Core Generation

A main configuration header file is customized by the designer and input to *eplcGen*. An example of this header file is visible on annex A.1. The header file contains the following open items for the user to define:

- Programmable core architecture
- Programmable core shape
- Basic blocks size (X:Y)
- Technology standard cells pinout and cells mapping

The result is a group of gate-level netlists that completely define the programmable core and its component blocks architecture, an user-friendly configuration file for the serial programming of the core, a simple testbench with programming chain verification

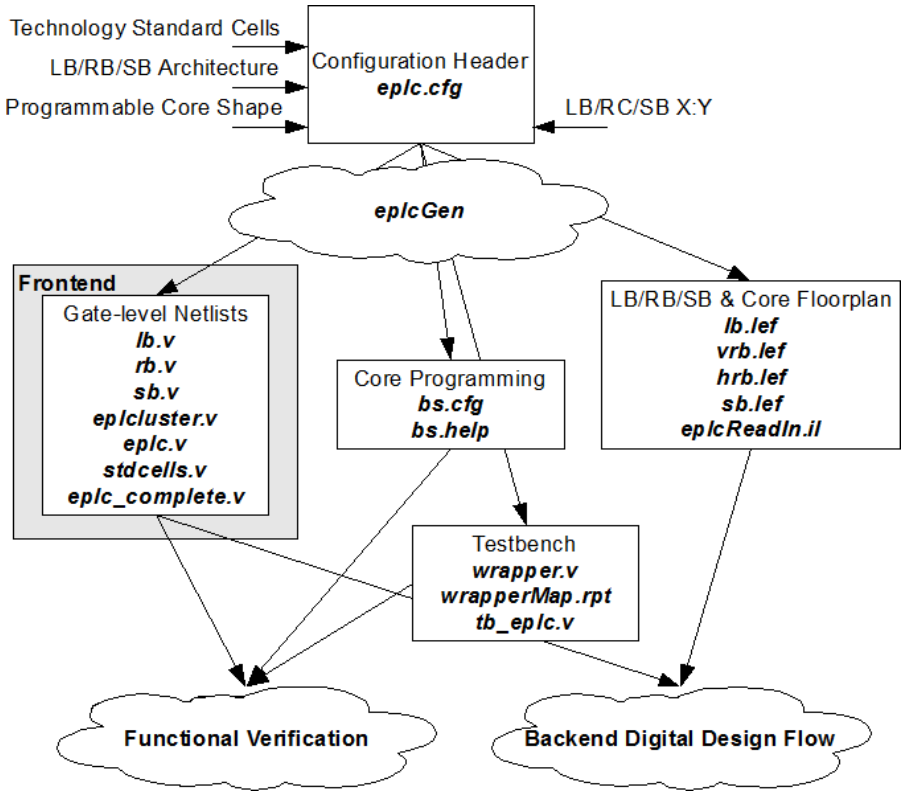


Figure 3.22: *eplcGen* flow.

tasks, floorplan data for the basic blocks and a script, customized to the core shape, to automatically create the core abstract and layout view on a layout editor tool.

eplcGen supports a 2- and 4-input architecture and based on that selection the logic, routing and switch blocks are respectively created based on the architecture presented before.

The circuits are created through the instantiation of generic standard cells. The mapping of the technology standard cells to the generic ones through its pinout association is made on the main configuration header file.

The shape and area of the programmable core a matrix of plus (+) and minus (-) signs that represent the presence or absence of a cluster block. The area occupied by each cluster block will depend on the size of its components blocks. These have their size defined on the configuration header file.

The output report of the core creation presents the shape of the core and the number of cells used on the initial gate-level netlist of the core and its component blocks. An example is presented on annex A.2.

eplcGen is the center of the flow diagram presented on figure 3.22. The inputs to the configuration header file are presented as well as the outputs of the automatic core generation (Gate-level Netlists, Core Programming files, Floorplan data and Testbench) and the tasks they serve (Functional Verification and Backend Digital Design Flow).

In this context, the frontend views of the reconfigurable core correspond to data that is still a design description, a soft implementation of the core. The backend views correspond to the hard physical implementation, layout, of the basic blocks and core. These views are created through the backend digital design flow. Both views and flow are described in more detail on the next sub-sections.

3.2.1.1 Frontend Flow

eplcGen generates the logic, routing and switch blocks in the form of gate-level netlists that employ technology-dependent standard cells. Together with the basic blocks, the tool creates gate-level netlists of the cluster block instantiating the basic blocks and of the programmable core top-level through instantiation of the cluster blocks. On the main configuration header file, the technology standard cells are mapped to generic standard cells names. This is the only link to the technology and is present on the *stdcells.v* file. This allows one gate-level netlist of the programmable core to be technology independent and valid for each *stdcells.v*. The same programmable core frontend views are this way easily portable to other technologies.

These gate-level netlists can be used in a standard digital design flow. In this way common CAD tools and optimized standard cells from commercially available libraries can be used, reducing the design risk associated with the reconfigurable core and its impact on the SoC design cycle.

As a preparation to the backend implementation of the basic blocks, *eplcGen* generates the floorplan data of the basic blocks needed to create their abstracts views for their layout implementation. This floorplan data is created in the form of a text file commonly known as a Library Exchange Format (LEF) file that describes each block size and pins locations. For each block, the signals input and output pins are placed on the block edges according to the known physical locations of the pins and such that all blocks connect by abutment to form the cluster block and the cluster block to form the programmable core. The same applies to the power and ground mesh of each block. These pins are also placed on the blocks edges such that the blocks can connect by abutment and maintain the power/ground mesh connection. By using lower metal layers for the creation of the basic blocks, higher metal layers can be used to build a strong power/ground mesh to supply the reconfigurable core.

A script is also created, *eplcReadIn.il*, that, when executed, creates the cluster block abstract and layout views. The abstract view of one cell is that cell boundary and pins locations while the layout view includes on that data the internal circuits that the cell contains with the respective connections to the boundary pins. These cluster views are created by instantiation of the basic blocks and create the complete programmable core abstract and layout views following the shape on the header file through instantiation of the respective cluster blocks. This script is written in *SKILL* [53, 54] programming

language and can be executed within different EDA software tools such as, for example, *Cadence Virtuoso*[1].

A bitstream configuration file and a help file to ease the task of creating the programming bitstream are also created by *eplcGen* based on the main configuration header file. This file contains the configuration bits for each logic, routing and switch block for each cluster on the created core and was created to help the designer configure the reconfigurable core without having to write down, at this stage, a series of 0's and 1's as large as the core itself. Examples of both files can be seen in annex A.3 and A.4. An additional utility tool, *bsParser*, was implemented to transform this bitstream configuration file into a serial bitstream that can be used to configure the core.

The generator also creates a wrapper top-level cell for the entire core, to connect the signals on the boundary clusters to signal buses to ease the integration of the core with the remaining macros on the SoC and to establish the connection between the isolated serial programming chains into a single programming chain. The mapping of data signals is output to a text file, *wrapperMap.rpt*, that the designer can refer to for an easier design integration. An example of the data signals mapping on this file, for a 2x2 programmable core, is shown in annex A.5. The simple wrapper cell created by *eplcGen* is an example of the logic that can surround the programmable logic core.

Figure 3.23 presents the automatically generated wrapper cell for a certain reconfigurable core, in this case a L-shaped core with seven 2-input LB cluster blocks. It is visible how the wrapper cell joins the boundary signals into buses and connects the programming chains into a single serial chain. As commented, this hierarchical level is not mandatory but it eases the core integration with the macros that surround it. It is this wrapper level that is instantiated on the testbench also automatically generated by *eplcGen*.

In order to assist the functional validation of the programmable core created, the generator creates a ready-to-run testbench implementing a single task that, by serially programming a sequence of 0's and 1's, counts the number of programming flip-flop cells on the complete core and compares it to the expected value. The testbench supports the task that applies the programming bitstream to the core wrapped with a single programming chain. This way, the template bitstream automatically created by *eplcGen* can be quickly used to test the core. This testbench can serve as a basis for more complex functional and timing verifications.

3.2.1.2 Backend Flow

To complete the core creation, the backend views of the basic component blocks need to be created making use of the respective gate-level netlists and floorplans generated before on a standard digital design flow.

[1] http://www.cadence.com/products/cic/pages/default.aspx

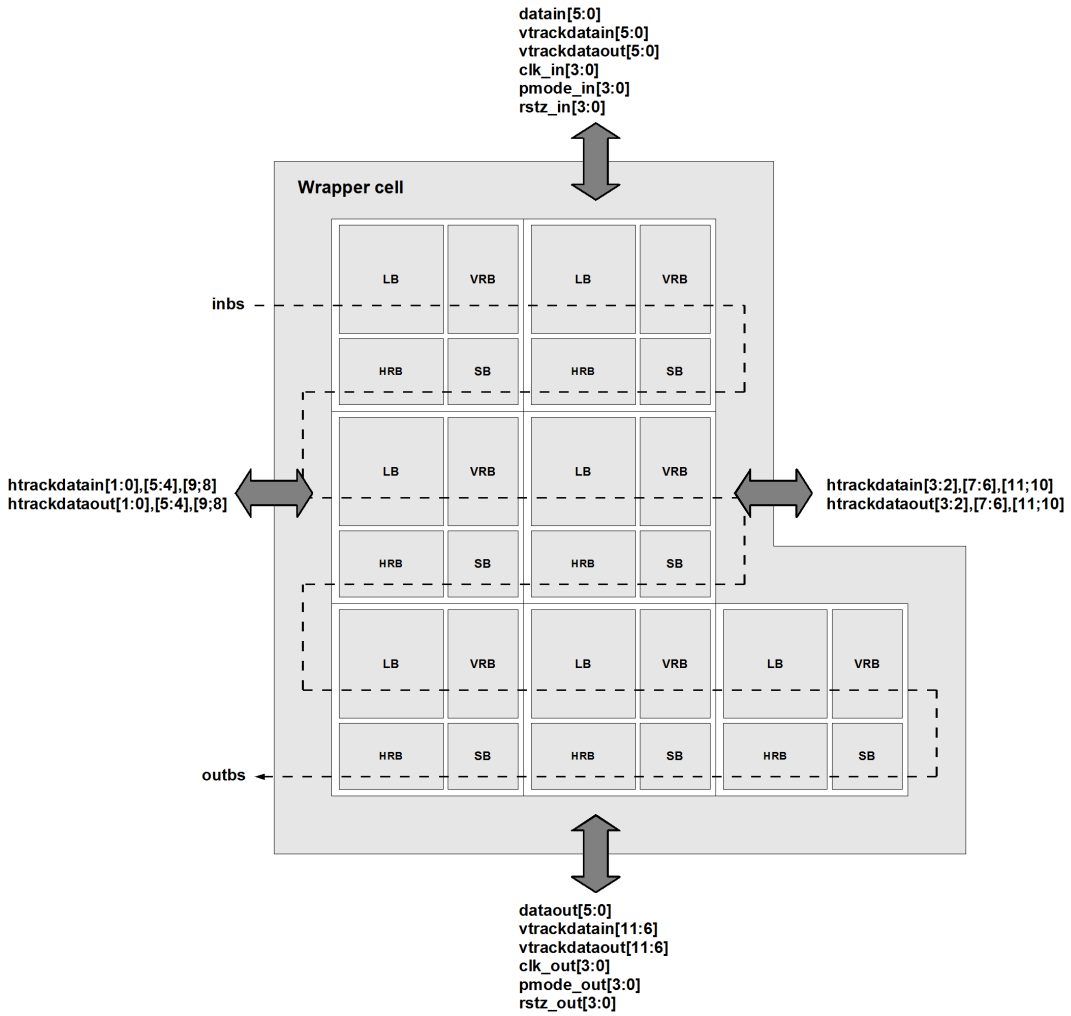


Figure 3.23: Embedded programmable logic core wrapper cell.

This backend flow needs to be executed three times, one for each basic block, to create the respective blocks layout. Once the basic blocks layout is created and validated, any reconfigurable core with any shape can be defined and quickly made available through the use of the support tool created, *eplcReadIn.il*.

Figure 3.24 presents the design flow used to create the backend views of the basic component blocks, the relation between them and the top-level programmable core and the functional and post-layout verifications. The basic blocks are instantiated into a cluster and then the cluster block into the embedded programmable core through the execution of the *eplcReadIn.il* script.

Since the basic component blocks are described already in gate-level netlists there is no need to synthesize the design. The gate-level netlists of the logic, routing and switch blocks, together with the standard cells technology data and the timing, area and power constraints can be used within a Place & Route tool. The cells in the gate-level netlists can be defined as "don't touch" cells to avoid having the automated CAD tools removing

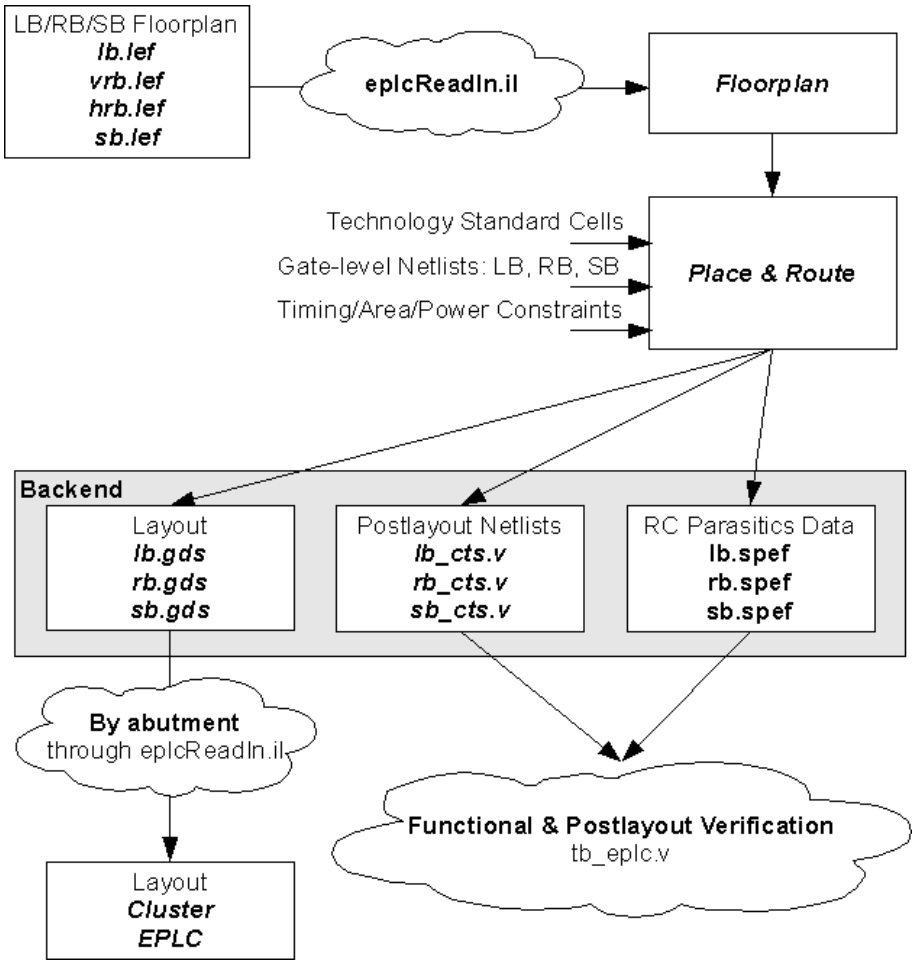


Figure 3.24: Embedded programmable core creation backend flow.

or scrambling the structures already created.

The advantage behind it is that each block can be treated as a simple fixed logic digital block, isolated from other blocks through proper sizing of boundary cells and therefore eliminating the different timing behavior each block could have on the top-level depending on the logic function that would be mapped on it. All input and output signals of each basic block are buffered for control within the block and cluster level and to avoid significant impact on a top-level when clusters are integrated together. This will isolate outputs, for example, from different capacitance loads that depend on the core configuration. This process eliminates also the problems related to complex combinational loops at implementation stage that would exist if a single layout of the complete top-level programmable core would be implemented.

The possibility to, within the Place & Route tool, balance one unbalanced multiplexing path using different drive strength standard cells or to correct or add small functionality to the blocks being made is a clear advantage over a full custom design where each of these changes require a longer implementation and verification time.

3.2.2 Reconfigurable Core Verification

This item presents a big challenge since the logic circuit functions that the programmable core will need to implement are not completely known. There is a tradeoff between the amount of verification with different designs mapped on the same reconfigurable core and the time needed and available to perform it. A study of this tradeoff is not part of this work but is essential in order to find a compromise that provides the highest feasible coverage to the programmable core verification.

The automatic programmable core generator creates a testbench with a simple programming chain verification task that can be used as a basis for more complex functional and timing verifications. Besides this, since there is currently no automatic mapping tool to work with the proposed core architecture, the bitstream that is automatically created can be updated and used so that simple functions can be mapped. In this way, simple verification tasks can also be added to the testbench and executed.

For the functional verifications with the gate-level netlists, the technology standard cells *Verilog* models need to be used. Some care needs be taken as those Verilog models usually contain non-zero delay values for the cells timing arcs. At this level of verification, simulation should be run without these timing delays and without timing checks as the purpose is to verify functionality only as on a regular behavioral simulation.

Even without the backend views created and therefore without accurate parasitics information for the design interconnects, Standard Delay Format (SDF) files can be created making use of the technology standard cells characterization data and an estimated wire-load model (or no wire-load model). These SDF files can be back-annotated on the gate-level netlists and, very early on the design cycle, a more complex timing verification can be already performed.

After the backend views are created, accurate parasitics can be extracted and realistic SDF files can be generated and used to replace the ideal or estimated wire-load models used before.

This last stage of verification already makes use of the most accurate timing information available. A group of different designs and respective logic functions should be mapped on the programmable core and the core functionally and timing verified for those designs and constraints.

Again, the option to implement the programmable core on a standard cell based design allows for an early simulation with gate-level netlists. On a full-custom design flow, transistor-level simulations with spice netlists need to be performed. These are more complex and time consuming adding this complexity and time to the development design cycle.

3.2.2.1 Bitstream

There is currently no tool to automatically map and route digital logic functions for the programmable logic core proposed. For this reason, a programming bitstream needs to be manually created with the help of the files and tool already described before and whose examples can be observed in annex A.3 and A.4.

The testbench that is automatically created by the generator and customized for each programmable core already supports the task to read in the configuration bitstream so the designer can quickly update the bitstream and use this already created task to configure the core.

3.3 Integration

eplcGen creates a simple wrapper cell that instantiates the programmable core top-level and simplifies its interface. This wrapper cell, that instantiates the reconfigurable core, is the hierarchical level instantiated on the testbench.

On a SoC integration, the programmable core will be surrounded by fixed digital logic, analog macros or I/O pins. These blocks will establish the needed connections within the SoC. Besides the regular signals connections, some care needs to be taken to tie unused input signals to avoid having floating input gates. Part of the surrounding circuitry will make use of the programming chains as needed by leaving them isolated or connecting them into a single chain.

If a fixed digital logic wrapper cell is used, similar to the one that is automatically created, then the interface to the core should be isolated through proper buffer cells. This wrapper cell can also provide a registered interface to the programmable core.

The wrapper cell not only provides a layer of digital logic above the reconfigurable core for the signal buses and the programming chains but it also allows to place as needed the reconfigurable core interface pins on different locations than the ones defined by the cluster blocks on the reconfigurable core boundary. For example, all data input and output signals can be moved to one edge of the core to ease the integration on the SoC.

This wrapper cell, if implemented alone, is customized and specific to a certain core with a certain shape, and is implemented also through a standard digital design flow. If the reconfigurable core is surrounded by fixed digital logic, the wrapper cell can be merged with this logic and therefore implemented together with it if needed.

3.4 Configuration and Operation

During the power-up of the programmable core its state will be unknown. Some care needs to be taken to avoid contention to occur during this power-up and also during the programming mode of operation. A global signal can be used to force, through additional

combinational logic, the three-state buffers to a known state such that there is no contention on the signal wires. This global signal was not implemented on this architecture to avoid adding complexity to it.

The programmable core is configured through the serial chains implemented in the basic blocks that compose it, as they were created with scan flip-flop cells. The configuration is therefore performed with a serial bitstream using those available scan chains.

Each horizontal line of clusters on the programmable core creates two programming chains 3.1.3 that can be stitched together to form a single chain. In the same way, horizontal lines of cluster blocks can have their single programming chain connected among themselves to create a single chain for the complete programmable logic core and this way allow the core to be completely reconfigured through a serial bitstream shifted at clock frequency. A representation of a non-rectangular programmable core is visible on figure 3.25, where a single chain is used to configure the core.

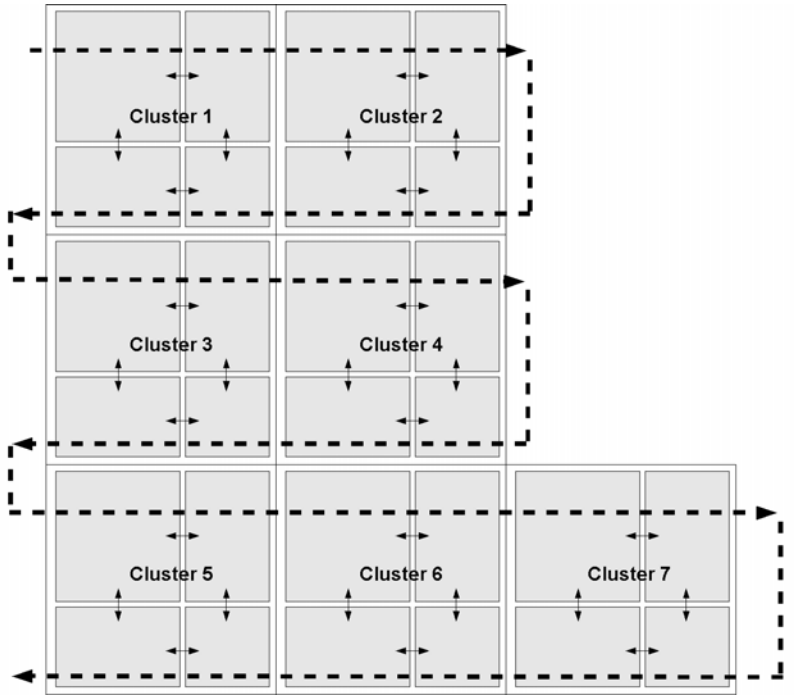


Figure 3.25: Serial bitstream programming.

Since each horizontal line of clusters has on its boundary two programming chains, then it is possible also to implement an external decoding circuit that can reconfigure the programmable core in a parallel fashion, driving all available programming chains at the same time with different bitstreams. Each horizontal line of cluster blocks can be programmed independently.

The fastest procedure to program the core is to do it in a parallel fashion, with multiple programming chains being configured at the same time. To evaluate the minimum programming time needed to cover the configuration of all cluster blocks, the longest isolated

chain needs to be considered. This corresponds to the longest horizontal line of clusters.

After the programming mode enable signal is asserted, a reset pulse should be applied to set the initial state of the programming FF cells to a known state. After this, the programming bitstream can be supplied to the programmable core at clock frequency. The programming mode enable signal will enable the FF scan chains on all blocks. While this signal is asserted, the output signals on the logic blocks are not toggling, being gated by combinational logic.

After the programmable core is reconfigured, the programming mode enable signal is de-asserted and the programmable core is ready to operate with the mapped digital logic function. A reset pulse should be applied to set the sequential logic to a known state. A reset pulse applied at this stage will not reset the programming FF cells as their reset is gated by the state of the programming mode enable signal. While the programming mode enable signal is not asserted, the clock to the programming FF cells is not toggling to avoid unnecessary power consumption during normal operation.

A diagram waveform with both stages of operation of the reconfigurable core, configuration and normal operation, is presented on figure 3.26. The different steps are identified through the logic state of the programming mode enable signal. The grey area represents the period of time during which the bitstream is input to configure to core.

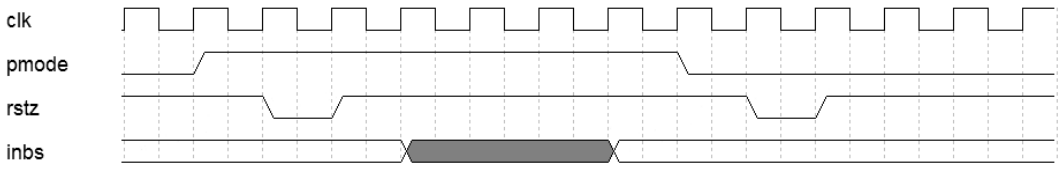


Figure 3.26: EPLC configuration and operation waveforms.

The same non-rectangular programmable core presented before on this section is now shown in figure 3.27 with one example digital circuit mapped onto it. This circuit doesn't represent any particular known function and it servers only as an example. On each LB a 2-input logic function was programmed together with the programming of the HRB and SB blocks. The figure shows the LB data input and output signals in use through with the horizontal, vertical tracks and switch block used to route and propagate those signals through the core. The different dotted lines are used to represent the different data paths established between the LBs after mapping and they are presented with different patterns for a clearer visualization. The arrows represent the signals input and output direction. The clock, reset and programming mode enable global signals are supplied to all clusters through their VRBs. These signals are not shown in the figure.

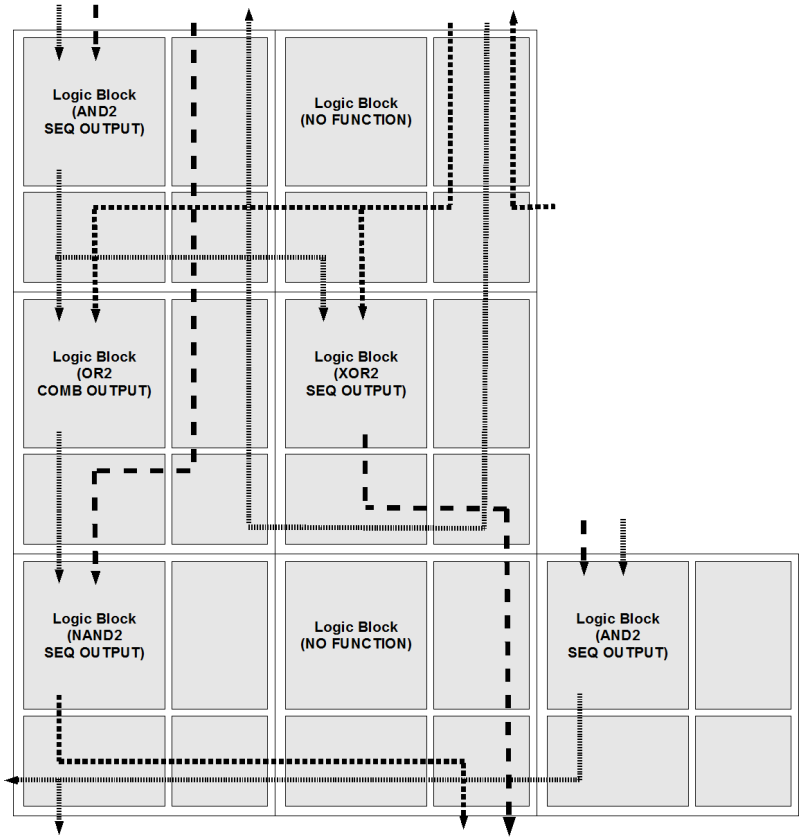


Figure 3.27: Embedded programmable core operation.

3.5 Summary

The embedded programmable logic core proposed is based on a typical island-style architecture. It is implemented on a set of standard cells, available on common standard cells libraries. The core generation and verification is this way executed through well defined digital design flows and using widely known and available to the industry CAD tools. From this methodology results a potentially short design cycle and development risk. However, the flexibility and reduced design cycle provided by the use of standard cells carry with them a significant area, delay and power overhead when compared to a digital logic design on a full custom design flow.

A minimum set of standard cells was defined to implement typical FPGA functionalities associated with signals handling: data storage, routing and disconnecting. The circuits created with the cells, under the proposed architecture, implement the reconfigurable core.

The basic blocks that compose the reconfigurable core are divided into 3 blocks: logic (LB), routing (HRB, VRB) and switch (SB). These blocks are instantiated by abutment to create the cluster block, that has one instance of each, and the cluster blocks are instantiated by abutment to create the top-level non-rectangular programmable logic core. The fine granularity of the cluster blocks that compose the reconfigurable core eases its

integration into irregular shapes within SoCs.

The architecture of these blocks characterizes the overall proposed architecture for the embedded non-rectangular reconfigurable core:

- 2- or 4-input data signals (LB)
- Sequential or combinational data output and inverted data output (LB)
- Unidirectional 4 or 6 vertical and horizontal data tracks (HRB, VRB)
- Switch block based on the design of S. Wilton [20] (SB)
- Global clock, reset and programming mode enable signals

The basic blocks are implemented with standard cells as standalone fixed logic digital blocks. Being created this way, combinational loops, common to FPGA structures, are eliminated. With this methodology, they are independent of the output load conditions or input drive strength and transition times, which will vary according the digital function mapped on the core, what allows these blocks to be instantiated by abutment to create the cluster blocks.

More complex configuration cells can be implemented but the concept has been described and validated. The architecture created is modular and flexible enough to accommodate different LB structures, density of horizontal and vertical tracks and SB implementations. Some examples include LBs without sequential logic, increased number of tracks for non-uniform routing channels, different SB typology or same typology with increased number of switches per terminal pin. These items should be discussed on a future work.

With the hierarchical organization described, all data paths are guaranteed to be the same. Delay characteristics are this way easy to estimate, across the core. With this information, it is simple to estimate the maximum possible operating frequency of the reconfigurable core, as will be described in more detail on chapter 4.

To support the creation and validation of the proposed programmable logic core, a set of tools, *eplcGen* and *bsParser*, was created to automatically generate the frontend views of the core based on a customized configuration file that gathers the information related with the technology standard cells, the architecture chosen, the size of the basic blocks and the programmable core shape in the form of plus (+) and minus (-) signs. *eplcGen* also generates the basic and cluster blocks floorplan information in the form of LEF files and additional data needed to create the programmable core backend views.

For verification, a testbench is created with simple programming related verification tasks. This same testbench can be used by the designer to add more complex verification tasks, related with the digital design functions the reconfigurable core will hold. Once there are means and automated ways of mapping a broad range of different digital designs on the reconfigurable core proposed, it will be possible to better evaluate it and its performance parameters and routability.

The non-rectangular programmable core is to be integrated with surrounding fixed logic digital macros or other hard macros that will control the way the core is connected and reconfigured through its serial programmable chains.

The core configuration is controlled by the state of the programming mode enable signal. When this signal is asserted, the bitstream is shifted in the programming chains at clock frequency. When the core is configured, the programming mode enable signal is de-asserted and the core is ready to operate with the mapped digital logic function. The core operation should start by applying a reset pulse that, on this mode of operation, will not reset the core configuration, and will set the sequential logic on the core to a known state.

The architecture proposed is tech-independent and the frontend views automatically created link to the technology standard cells only through their mapping to the generic standard cells instantiated on the cores. The technology dependency also exists when the backend views need to be created since the layout physical implementation is intrinsic to a certain technology process node.

In the end, several different non-rectangular shapes have been automatically generated and validated under the design flows and procedures described.

Chapter 4

Experimental Results

With the proposed architecture for the non-rectangular programmable cores defined and with a support tool, *eplcGen*, that automatically generates their frontend views and prepares the work for their physical implementation, it was possible to validate the operation of different non-rectangular cores and characterize area, delay and power consumption.

While it is not possible to evaluate the core routability and compare it with other reconfigurable design solutions, the following sections describe the intrinsic characteristics of the programmable cores implemented under the presented architectures.

Data for area, delay and power consumption was evaluated for a UMC 90nm process node and for a predictive 45nm process node, whose design kit is made available by the North Carolina State University [55] and standard cells library by the Oklahoma State University [56].

The layout views of the basic blocks for a 2-input LB were implemented for a 90nm CMOS process node. With these layout views, the cluster block and any non-reconfigurable core shape is quickly built. One example is also presented ahead.

4.1 Verification Environment

The verification environment for the functional evaluation of the reconfigurable core was created and executed using *Modelsim*[1], a simulator tool from Mentor Graphics[2]. While the basic blocks described were simulated and verified for both implementations under the 2- and 4-input LB architectures proposed, the top-level non-rectangular reconfigurable core verifications were performed only for a 2-input LB architecture, as will be shown later.

[1] http://www.model.com/
[2] http://www.mentor.com/

4.1.1 Basic and Cluster Block

The validation of the proposed architecture, during the development stage, started by the functional verification of each of the basic blocks that were created to compose it. For this, dedicated testbenches were written to validate the circuits and functionalities implemented on these blocks.

On the LB, the LUT circuit that holds the programmed logic function and the respective output were verified by mapping different 2- and 4-input logic functions truth tables on the programmable FF cells that compose the LUT and by sweeping all possible data input logic values, what exercises all data paths from the truth table to the LB data output and inverted output. On the same data paths, the combinational and sequential arcs for this output data signal have been also verified. The comparison of the data output signal with the expected output for each logic function programmed determines the correct operation of the LUT and respective routing circuitry. Intrinsic to this verification, the existing programming chain and programming mode are exercised and verified as this chain is used to configure the logic function on the LUT and to define the state of the logic block output, combinational or sequential.

The verification performed over the HRB during its architecture validation stage targeted the evaluation of all data paths between this block input and output data signals. This includes the access to and from the LBs connecting to it and the evaluation of the correct control over line drive conflicts. The data paths are stimulated with a clock pattern toggling on a certain input signal and verified on one or more configured output signals. This comparison is done for all possible input to output connections. Again, for this verification, the programming chain is fully exercised as the block configuration done through it is what defines the different possible connections established on the HRB.

The fact that the VRB contains only combinational logic to propagate the data and global signals results that the verification performed over this block was simply done to validate the connections established.

The SB specific verification tasks were created such that all possible input to output connections, through the different switches available on the implemented architecture, were exercised and validated. As for the HRB, a clock pattern toggling on a certain input signal was used as stimuli and compared with the expected output signal, according to the block configuration. Again, the programming chain was used to configure the block selection paths what validates the programming mode of operation also on the SB.

For all basic blocks, the correct propagation of the global signals and the functionality of the remaining control logic, part of it common to all blocks, were also verified at this stage. This includes, for example, the gating through combinational logic of the logic block output signals during programming mode or the reset logic control during normal mode of operation to avoid applying a reset signal to the programmable FF cells.

Having verified through dedicated testbenches the proposed architecture for the basic blocks, verifications where performed on a similar environment for a complete cluster block. With a single cluster block, logic functions were mapped and their output compared to the expected value on one configured output signal of the cluster block. On this simulation, the data tracks not connected to the LB are stimulated with a clock pattern, observable on a programmed output of the cluster block. This allows to cover all expected connections and verify again the correct control of line drive conflicts on the HRB. The correct propagation of the global signals to the blocks within the cluster is also guaranteed during this verification as the entry point of these signals occurs on the VRB and from this block, they're propagated within the cluster to all blocks within. The verification environment executed over a single cluster block represents the hypothetical scenario of a non-rectangular reconfigurable core composed by one single cluster.

To summarize, the verifications performed during the development of the proposed architecture for the basic blocks, the capability of storing a simple digital logic function on the LUT circuit on the LB and propagating its result and other data and global signals through the reconfigurable mesh has been validated for the basic and the cluster blocks.

4.1.2 Non-rectangular EPLC

The functional validation performed over the non-rectangular reconfigurable cores was run on the same verification environment and with the same simulation tools as the basic and cluster blocks. While these blocks verification involved specific testbenches and tasks created for each of them, the non-rectangular core verification makes use of the automatically generated testbench as a basis for this part of the work.

On this testbench, the first task that is always executed is the evaluation of the number of programmable FF cells on a single serial chain, created through the wrapper cell that stitches all chains into a single one. This first level of validation compares the number of programmable FF cells accounted for when shifting a stream of 0s and 1s at clock frequency with the expected value, known from the architecture implementation.

Additional specific tasks can be added to the automatically generated testbench to stimulate the core data inputs and, depending on the functionality programmed, validate it through comparison with the expected functionality, observed on the core data outputs.

Reconfigurable cores with non-rectangular shapes ("S", "L", "T", "U") were automatically generated for a 2-input LB architecture. These cores were created with a number of cluster blocks between 39 and 54. Since the 4-input LB architecture represents a scaling of the first one with support for more complex digital logic functions within the same cluster and an added number of tracks and configuration complexity, the simplest architecture was chosen for functional simulations performed at the reconfigurable core top-level.

Without the possibility to automatically, and on a reasonable time, configure the programmable logic cores with complex digital logic design functions, simple 2-input logic functions such as AND, XOR, NAND and others, were configured to chosen cluster blocks

within the core. An example is visible in figure 4.1 where the cluster blocks configure with digital logic functions are highlighted. Other cluster blocks are also programmed to establish the data signals routing paths through the core. The overall group of functions and connections mapped on the cores doesn't represent any specific functionality. On the other hand, they allow for simple verification tasks to be created and added to the core testbench, For example, one programmed cluster block, with a known function, can have its output data signal propagate till the core boundary output pins, and at those pins, compared with the known function.

Figure 4.1: EPLC verification example - cluster blocks configured with logic function.

The hierarchical organization allows that if the possible data paths between cluster blocks on one reconfigure core are validated for a certain core shape, any non-rectangular core is this way functionally validated as these data paths and structure that supports them doesn't change. Since it is guaranteed by the architecture that the programmable core mesh is created by instantiation of the cluster blocks, and guaranteed that they

connect among them when the core is created, then these data paths and the global signals propagation is by itself validated. What is specific to each core is the surrounding blocks and the way they interface with the core and, in the case where a dedicated wrapper cell is used, also this wrapper cell is specific to each core.

For each of the reconfigurable core shapes created ("S", "L", "T", "U"), simple digital logic functions were mapped to different cluster blocks such that the simulations performed exercised cluster blocks on all possible positions within the core. This means that cluster blocks with none, one, two, three or all edges as making part of the core boundary were configured and validated. The reason for this concern is that, this way, the automatic core creation process by *eplcGen* tool is also validated since the connections established by this tool are dependent on the cluster block location within the core.

The simulation work for the non-rectangular cores was performed in two different steps.

The first group of simulations was performed over the gate-level netlists automatically generated by *eplcGen*. These represented functional simulations only, with no timing associated and allowed to functionally validate the architecture proposed, both for the programming and for the normal mode of operation. The methodology followed, to exercise the core by programming some of its clusters and then stimulating the configured digital logic functions, exercises all the logic involved on the programmable circuit mesh. The same testbench created for the first group of simulations was also applied to the second group.

The second group of simulations used time-annotated standard cell netlists. The timing to be back-annotated on the netlist in the form of SDF files, was obtained through an appropriate wireload model or parasitics extraction of the layout implemented. Both wireload models and the parasitics extraction of the layout represent the electrical parasitic devices such as capacitance and resistance of the nets. The standard cells intrinsic timing delays are affected by these characteristics of the nets that connect to them and those timing numbers are equally affected and written to the SDF file, later used on these timing accurate simulations. This group of simulations was done to evaluate with some accuracy the performance and maximum operating frequency of the mapped designs on the programmable cores and, this way, bound its performance. Simulations performed correlate with the estimations performed on section 4.3 for the sequential data paths evaluated. As will be shown on this section, the operating frequency is design-dependent and the study presented will help to bound these values and presents estimations that align with the results obtained on these simulations.

4.1.2.1 Example Layout

To illustrate the procedures followed to create the physical views of the basic, cluster blocks and the non-rectangular core, each step is here described with some detail. Similar to what is shown on this section, any other non-rectangular core with a different

shape, based on the same basic blocks, can quickly implemented through the *eplcGen* and *eplcReadIn.il* tools described before.

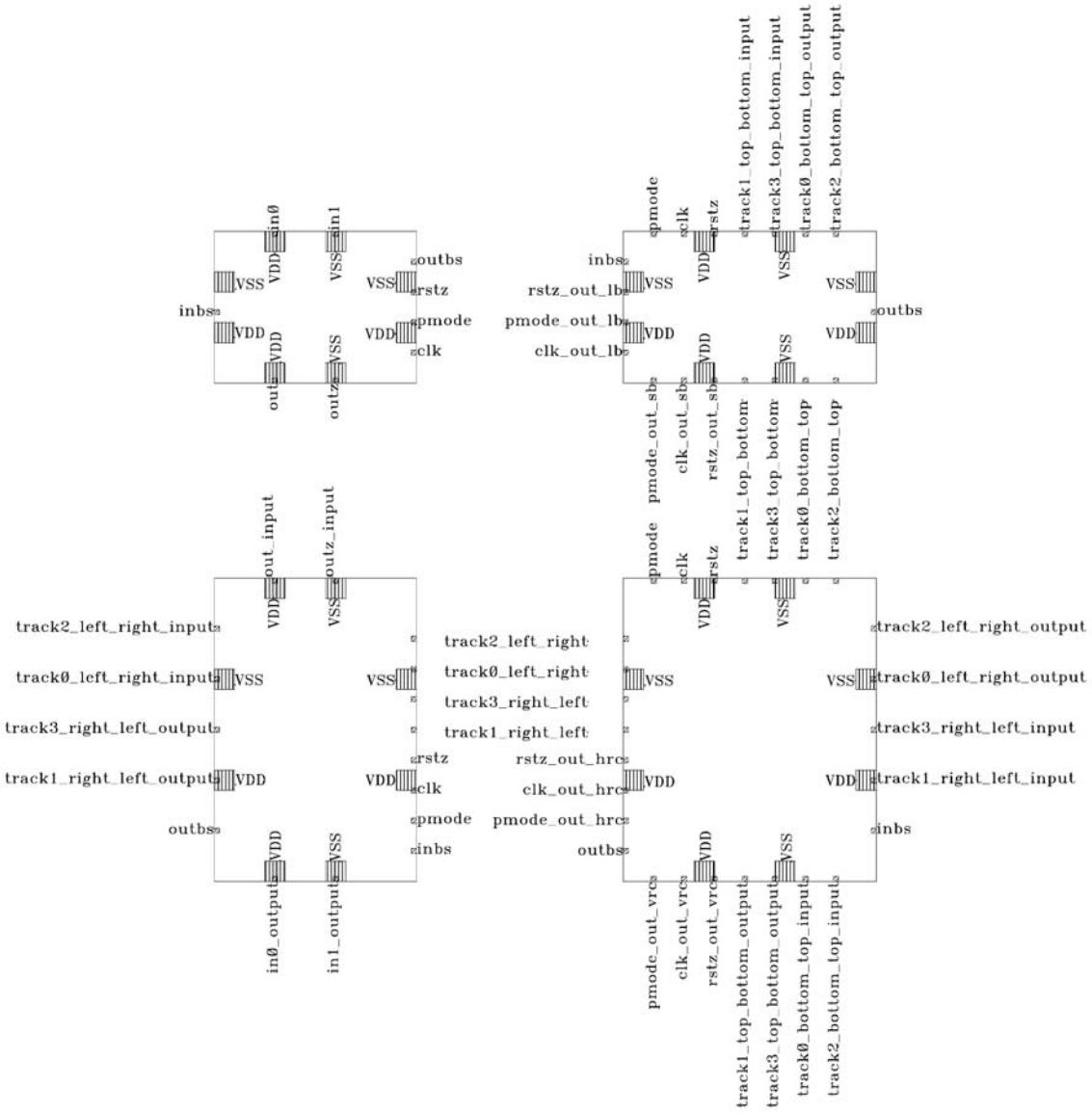


Figure 4.2: LB, HRB, VRB, SB abstracts for a 2-input LB.

eplgGen tool outputs the basic bocks floorplan information, based on the user-defined header, under the form of LEF files. These files can be read into a layout tool editor such as Cadence and the abstracts of each block created. As stated before, a simple utility tool was created, customized for each non-rectangular programmable core, to automatically read these files and create the library and the cluster and core abstract and layout views. The layout views will only be complete once the basic blocks physical implementation is finished.

The abstract views are the basis for the floorplan data that will be input, together with the gate-level netlists, to the place and route tool used to physically implement each

of the basic blocks.

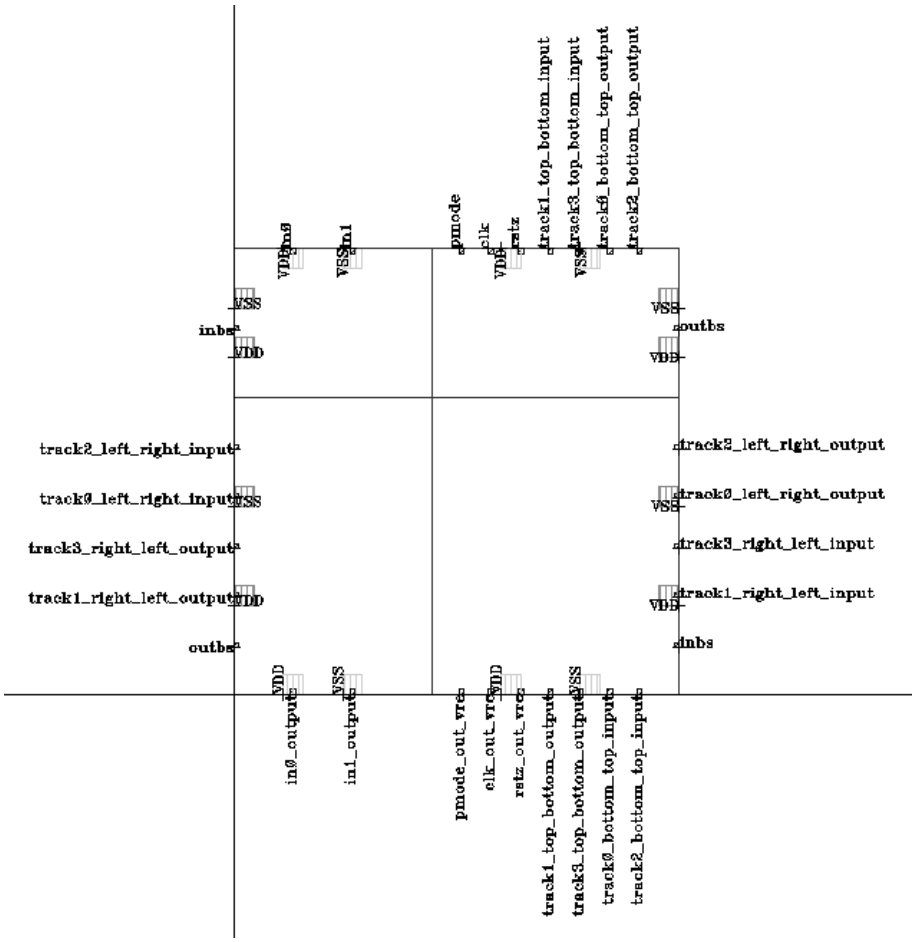


Figure 4.3: 2-input LB cluster block abstract view.

Figure 4.2 present the abstract view of the LB, HRB, VRB and SB implemented for a 2-input LB and sized for a 90nm CMOS process. As can be observed, the pins on the interface of each block are distributed such that the basic blocks, when instantiated by abutment to form the cluster block, establish those signals connections within the same cluster and with adjacent clusters. It is also visible that the power pins were created in such a way that when the entire programmable core is physically implemented, a strong power mesh covers the entire core, providing a uniform supply distribution to the complete macro.

Figures 4.3 and 4.4 show the abstract and layout view, respectively, of a cluster block, created by instantiation of the blocks that compose it, can be observed.

The cluster block occupies an area of 2045 μm^2, being 45 μm wide and 45 μm tall. Although, for this technology and process node, the cluster block could be made smaller, it was made with some additional area margin to ease the implementation process.

The instantiation of cluster blocks creates the non-rectangular programmable core abstract and layout view. In the example shown on figures 4.5 and 4.6, a core using 318

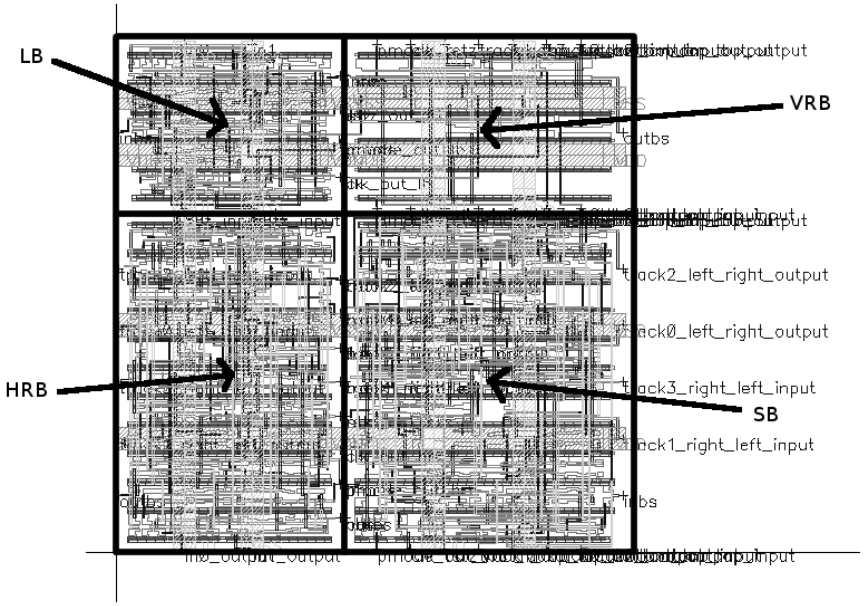


Figure 4.4: 2-input LB cluster block layout view.

clusters and with a non-rectangular shape is presented. The silicon area occupied by this core is around 0.65 mm^2. On these same figures, a group of cluster blocks is highlighted for reference as wells as two hard macros that set the example of how irregular space for the non-rectangular can possibly exist.

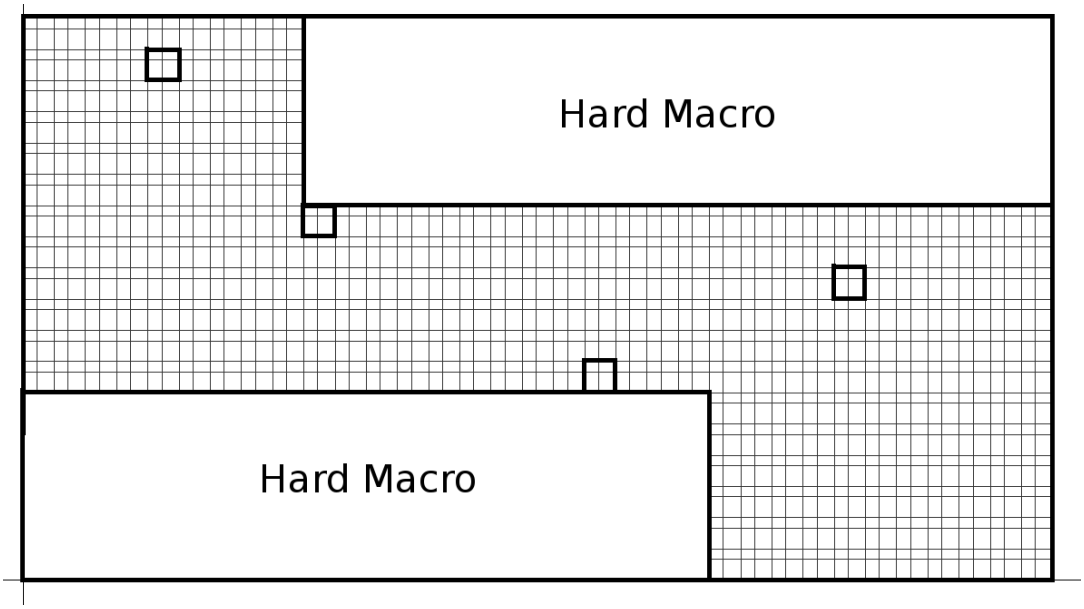


Figure 4.5: Non-rectangular programmable core abstract view.

With the basic blocks physically implemented for this process node, the creation of any non-rectangular programmable core using these blocks is quick and reliable. The

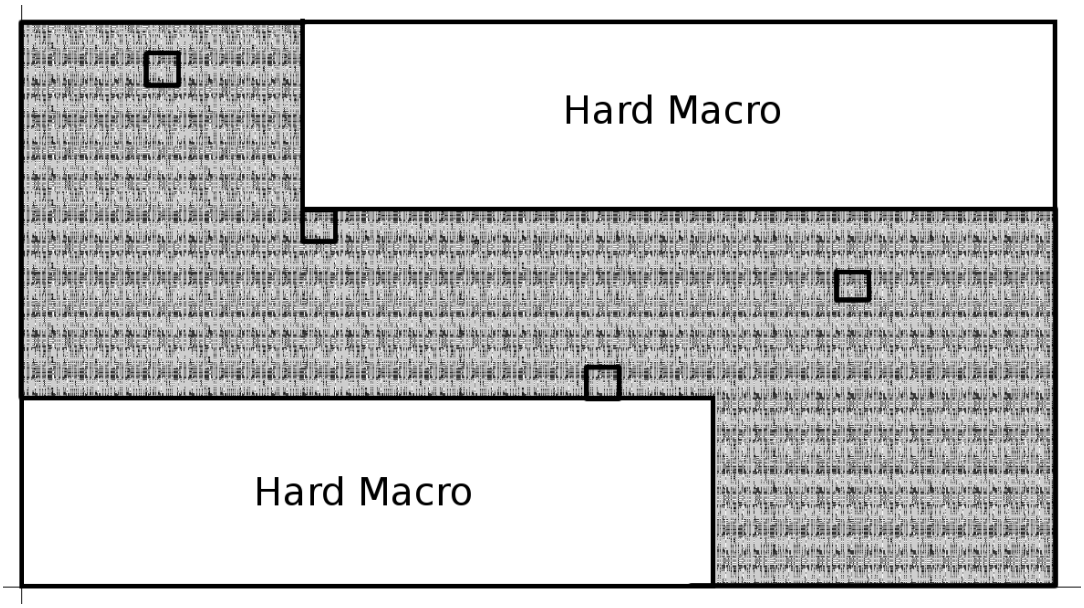


Figure 4.6: Non-rectangular programmable core layout view.

designer can customize the main configuration header file with the core shape desired and after executing the tool, the eplcReadIn.il script can be executed under a layout tool environment in order to automatically create the non-rectangular reconfigurable core.

4.2 Area

The area occupied by the programmable core is independent on the design that can be programmed to it. The approximate silicon area that a certain programmable core can occupy can be estimated by adding the estimated area of each cluster block that composes it. The approximate area that each cluster block occupies can be obtained by estimating the area occupied by each of its basic component blocks: LB, VRB, HRB and SB.

Tables 4.1 and 4.2 present the summary of standard cells used to implement each block basic circuit for both 2- and 4-input LB based architectures.

	2-input LB	***VRB***	***HRB***	***SB***	***Total***
SDFF	5	0	12	16	33
DFF	1	0	0	0	1
BUF	8	11	19	24	62
MUX2	4	0	6	0	10
INV	2	0	0	0	2
AND2	3	0	1	1	5
TBUF	0	0	12	0	12
NOR2	0	0	4	0	4
MUX3	0	0	0	8	8

Table 4.1: 2-input LB architecture: basic blocks area breakdown.

	4-input LB	*VRB*	*HRB*	*SB*	*Total*
SDFF	17	0	24	24	65
DFF	1	0	0	0	1
BUF	20	13	35	32	100
MUX2	16	0	12	0	28
INV	2	0	0	0	2
AND2	3	0	1	1	5
TBUF	0	0	18	0	18
NOR2	0	0	6	0	6
MUX3	0	0	4	12	16

Table 4.2: 4-input LB architecture: basic blocks area breakdown.

For a more accurate estimation, the designer should add to this area value an overhead of around 20%-30% to represent the space needed for the place and route tool to implement each block. This extra space is needed as the tool might need to change the existing cells driving strengths (changing this way the cells area), add buffer and inverter cells as needed to create the clock and reset trees and to help meet the design timing constraints.

The approximate area for each cluster with a 2-input LB is 1600 μm^2 for a 90 nm process and 1000 μm^2 for the predictive 45 nm process. At the smaller technology node, a 100×100 programmable core would take around 10 mm^2.

4.3 Operating Frequency

The maximum operating frequency at which a sequential digital logic design can operate is dictated by the longest delay path between two sequential FF cells. An estimation of this maximum clock frequency can be obtained through the following equation:

$$Max\ F_{CLK} = \frac{1}{T_{CLK->Q\ (FF1)} + T_{Delay} + T_{setup\ (FF2)} + T_{CLK\ Uncertainty}}$$

On this equation, $T_{CLK->Q}$ corresponds to the data generation FF delay between clock edge and output assertion, T_{delay} is the total delay from all combinational logic that exists between the two FFs, $T_{setup\ (FF2)}$ portion corresponds to the setup time of the data capture FF that also needs to be met and the last component is related with the clock uncertainty that needs to be added for this estimation, $T_{CLK\ uncertainty}$. A reasonable value for this clock uncertainty is between 5% and 10% of the clock period when this clock is known in advance.

The maximum operating frequency of the proposed embedded programmable core needs to be evaluated in two different scenarios that correspond to the two different modes of operation: programming and functional mode.

The first corresponds to the programming mode where a serial bitstream is input through the programmable chains. These chains were implemented through the regular

scan chain each basic component block contains. This mode is enabled when the programming mode enable signal is asserted.

The second mode of operation corresponds to the functional mode. The programmable core is configured with a certain digital function and in this mode that digital function is exercised and used. This mode is enabled when the programming mode enable signal is de-asserted.

An estimation of the maximum clock frequency during the programming and functional modes of operation can be quickly done through the documented standard cells timing characterization. Taking into consideration medium drive strength standard cells, their input capacitance loads (that set the load for the standard cell driving it) and the respective propagation delays and timing constraints that correspond to the portions described above, the clock frequency can be estimated. For this calculation, a clock uncertainty of $100ps$ is used.

The operating frequency estimations presented on the next sections refer only to the reconfigurable core implementation on the 90nm CMOS process. The data obtained for the predictive 45nm node presents bigger delays and respectively a slower possible operating frequency on the reconfigurable core created on this process node. This can perhaps be explained by a different process flavor that could be on the origin of the predictive 45nm standard cells.

4.3.1 Programming Mode

On this mode of operation, the maximum clock frequency during programming mode of operation is limited by the longest path between 2 scan FF cells on the programming chain. On both supported architectures, the longest path between 2 scan FF cells on the programmable core will not be longer than 2 or 3 buffer cells, considering already the connection to the first FF cell on the chain, through the external pins. On this case, it is also not expected for the path between the external and internal to the core FF cells to be separated by a different number of buffer cells. This uncertainty on the number of buffer cells is related with the number of buffers needed between each two scan FF cells to prevent that the output signal of the first one is capture by the second FF on the same clock cycle what is technology dependent. To this combinational delay adds the first FF $T_{CLK->Q}$ and the second FF T_{setup}. The $T_{CLK->Q}$ output delay is bigger than the same part on a FF that doesn't support scan like the ones used on the LB to generate the sequential logic function output. The setup time is related with the scan data input timing constraint and not the FF regular data input.

For this 90nm node, the maximum clock frequency for this mode of operation is estimated as to be around $1.4GHz$ ($710ps$ delay path). This frequency of operation doesn't depend on the architectural choice made for the LB (2- or 4-input).

To summarize, the data provided and the fact that standard cells are used to implement the logic circuit, allows the designer to have, at an earlier stage of the design, an idea of the

maximum clock frequency that can be targeted on this programming mode of operation. This estimation doesn't depend on the shape or size of the programmable core.

4.3.2 Functional Mode

The speed of the mapped digital function is design-dependent as it is determined by the data path delays between the core input data signals and the first FF cells that samples them (A), between two FF cells within the core (B) and between the last FF cells generating the data and the core output signals (C). On a reconfigurable core, the programming of the digital logic function will define the extent of these data paths and therefore the maximum possible operating frequency. The synchronous timing arcs described are respectively highlighted in the diagram presented in figure 4.7 where the Embedded Programmable Logic Core (EPLC) core is also shown. The clouds of combinational logic represent the data paths between each two FF cells and it is visible that for paths A and C, part of that combinational logic can be external to the EPLC and therefore unknown to this analysis.

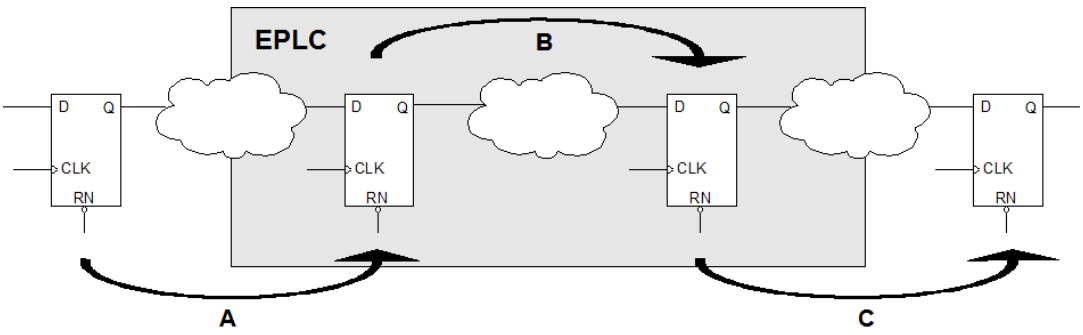


Figure 4.7: EPLC and sequential logic data paths.

The study presented next is divided in two parts. First, the maximum operating frequency at which the core could operate, constrained by the proposed architecture, is discussed. The second part presents all possible data paths on the core and, based on that, describes three possible paths that constrain the maximum operating frequency for a certain design mapped on the reconfigurable core. While the results for this second analysis correspond to synchronous timing arcs within the non-rectangular EPLCs, the study performed also details the data paths on the core boundary, interfacing the external fixed logic or other hard macros.

Together with the data paths description, an example of each possible data path propagating through the basic blocks is highlighted on simplified diagrams that represent each basic block for a 2-input LB architecture. For simplicity, only part of the circuit is presented. The BUF cells used for isolation of the blocks input and output pins, as well as additional circuitry such as the INV cell for the inverted data output generation and the

AND2 cells used on the LB are not represented. On the same sense, global signals and combinational logic for control are also not part of these diagrams.

The maximum operating frequency that the core could support is set by the minimum delay between two FF cells on the proposed architecture. The shortest path between two FF cells can be found between two adjacent logic blocks, separated by a horizontal routing block. This delay path is one from the complete possible data path delays that will be described next. For this particular case and analysis, the delay path is as shown on table 4.3 and figure 4.8 where the data path starts on LB1 FF cell ($T_{CLK->Q}$) and terminates on the closest FF cell capturing the data on LB2. The additional combinational cells on the table represent each combinational delay on this path. This information, together with the technology standard cells timing characterization and an estimation for the clock uncertainty helps to quickly bound the maximum operating frequency at which the programmable core could operate.

Depending on the data input on the LB that will toggle, the output of the logic function truth table will flow through the entire or portion of the multiplexer tree that connects to it. For this functional mode operating frequency evaluation, the worst case delay (complete multiplexer tree) is considered.

An estimation of this maximum clock frequency that the core architecture could support for a 90nm CMOS process, and based on the same data used for the calculations presented on the previous section, is around $850MHz$ for a reconfigurable core based on a 2-input LB and around $740MHz$ in the case of a 4-input LB.

The speed of the mapped logic is design-dependent, but an upper limit around $850MHz$ for a 2-input LB is this way established for the maximum clock frequency at which the reconfigurable core architecture could operate. This result has been confirmed through post-layout simulations, executed as presented on the verification environment for the reconfigurable core top-level.

The data paths that need to be considered for this evaluation start and end on the LB FF cell as this is the cell that generates and captures the sequential data. These data paths propagate through the HRB, VRB and SB between each 2 LBs on the programmable core and the complete path delay determines the clock frequency operation.

The knowledge of the data path delays for each block will allow the designer to quickly estimate the maximum operating frequency at which the core could operate, for a certain digital logic function configured on the core and having the designer the knowledge of the cluster blocks configuration. An automated tool should be able to automatically map any digital logic function on the core and present the results of this maximum operating frequency estimation.

The data paths that need to considered inside the LB should be separated in three different timing delays as, besides the combinational path the LB supports between its data input and output signals, this block contains both the start and end point for all sequential arcs involving the programmable core. The data path description for the start

	2-input LB	***4-input LB***
LB1	**FF** ($T_{CLK->Q}$) MUX2 AND2 BUF/INV (out/outz)	**FF** ($T_{CLK->Q}$) MUX2 AND2 BUF/INV (out/outz)
HRB	(in) TBUF MUX2 MUX2 BUF (out)	(in) TBUF MUX2 MUX3 BUF (out)
LB2	(in) BUF MUX2 MUX2 **FF** (T_{setup})	(in) BUF MUX2 MUX2 MUX2 MUX2 **FF** (T_{setup})

Table 4.3: Data path for reconfigurable core maximum supported operating frequency.

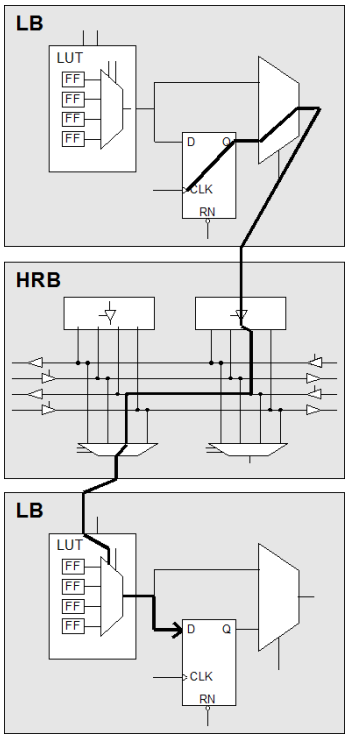


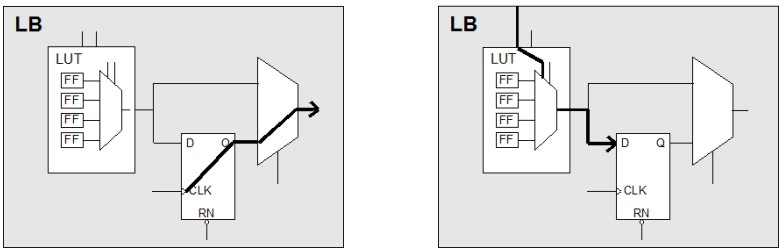
Figure 4.8: Data path for reconfigurable core maximum supported operating frequency.

and end points for the sequential timing arcs on the LB is summarized on table 4.4 and the combinational data path through the LB is summarized on table 4.5. Possible data paths for these scenarios are respectively highlighted on figures 4.9 and 4.10.

One of the data paths is associated with the data generation and on this case the data generated on the LB FF cell is the start point for the timing arc. The intrinsic delay between this start point and the LB data output (end point) is the same for both

	2-input LB	***4-input LB***
Data generation		
LB	**FF** ($T_{CLK->Q}$) MUX2 AND2 BUF/INV (out/outz)	**FF** ($T_{CLK->Q}$) MUX2 AND2 BUF/INV (out/outz)
Data capture		
LB	(in) BUF MUX2 MUX2 **FF** (T_{setup})	(in) BUF MUX2 MUX2 MUX2 MUX2 **FF** (T_{setup})

Table 4.4: LB data path: data generation and data capture.



(a) Data generation (b) Data capture

Figure 4.9: LB data path: data generation and data capture.

	2-input LB	***4-input LB***
LB	(in) **BUF** MUX2 MUX2 MUX2 AND2 **BUF/INV** (out/outz)	(in) **BUF** MUX2 MUX2 MUX2 MUX2 MUX2 AND2 **BUF/INV** (out/outz)

Table 4.5: LB data path: combinational data path through LB.

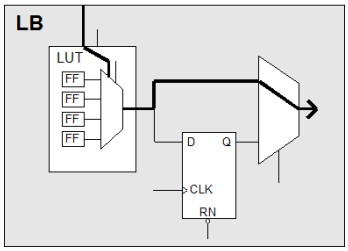


Figure 4.10: LB data path: combinational data path through LB.

proposed architectures. Inside the LB, the data generated on the FF cell crosses a 2-input multiplexer that exists to control if the output of the LB is a sequential or a combinational output, through a 2-input AND cell that exists to avoid the outputs of the LB to toggle during programming mode of operation by gating them and reaches its output by crossing a buffer or a inverter cell depending if we're considering the output or the inverted output of the LB. This path can be reduced and the clock operating frequency increased if the 2-input AND cell is removed from the architecture.

The second data path that needs to be considered is associated with the data capture by the FF cell. This path corresponds to the delay between LB data inputs and the FF cell that on this case corresponds to the delay end point. As stated before, the LB input signals path can cross only part or the complete multiplexer tree to reach its output. The worst case scenario is considered on the table where the input data needs to propagate through the complete multiplexer tree before the FF cell that captures it.

The last data path that needs to be taken into consideration inside the LB is the combinational path that exists between the LB input and output signals. On this case, the input signals cross the complete multiplexer tree and then reach the LB output propagated through a 2-input multiplexer cell, a 2-input AND cell and a buffer or inverter cell. The data signal doesn't cross the sequential FF cell within the LB.

	2-input LB	***4-input LB***
VRB	(in) **BUF** **BUF** (out)	(in) **BUF** **BUF** (out)

Table 4.6: VRB data path.

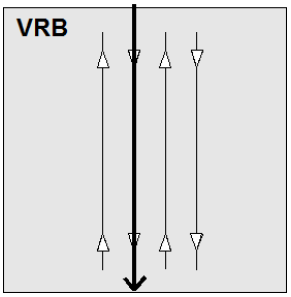


Figure 4.11: VRB data path.

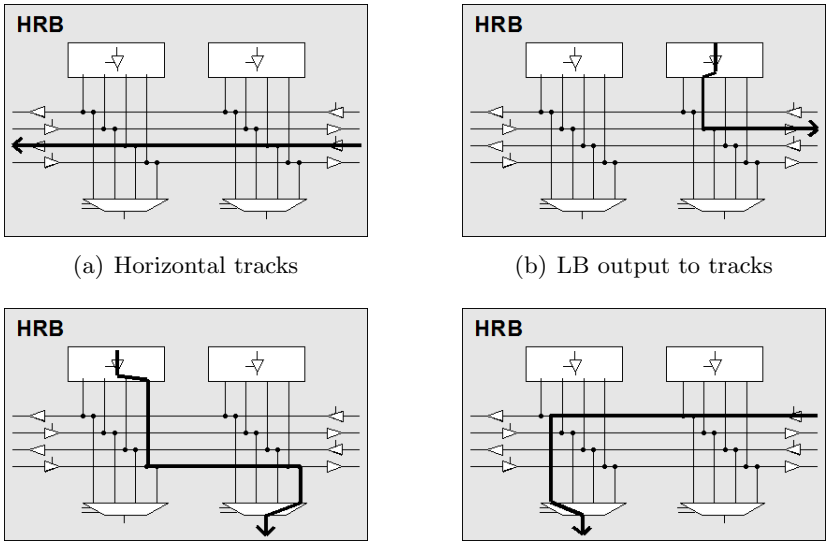
The VRB adds to the data path buffered lines for data propagation. The number of data lanes is different for each of the proposed architectures but the added intrinsic delay to each data line is the same. Table 4.6 and figure 4.11 present these data paths.

The HRB provides different data paths that need to be evaluated. The intrinsic combinational paths associated with each data path are detailed on table 4.7 and figure 4.12.

If the signal is input to the HRB through its horizontal track inputs and reaches the HRB horizontal track output, it will propagate through a three-state buffer and a buffer cell.

	2-input LB	***4-input LB***
Horizontal tracks		
HRB	(in) **TBUF** **BUF** (out)	(in) **TBUF** **BUF** (out)
LB output to tracks		
HRB	(in) **TBUF** **BUF** (out)	(in) **TBUF** **BUF** (out)
LB output to LB input		
HRB	(in) **TBUF** MUX2 MUX2 **BUF** (out)	(in) **TBUF** MUX2 MUX3 **BUF** (out)
Tracks to LB inputs		
HRB	(in) **TBUF** MUX2 MUX2 **BUF** (out)	(in) **TBUF** MUX2 MUX3 **BUF** (out)

Table 4.7: HRB data path.



(a) Horizontal tracks

(b) LB output to tracks

(c) LB output to LB input

(d) Tracks to LB inputs

Figure 4.12: HRB data path.

The connection of the LB output signals to the HRB tracks is made through a three-state buffer cell. This signal will propagate through a buffer cell if connected to the HRB output tracks or through a multiplexer tree and a buffer cell to connect to the adjacent LB input pins.

The last data path considers the signals that are inputs of the HRB tracks and propagate to the LB input pins on the adjacent cluster. The signal will cross a three-state

buffer cell at the HRB input, the multiplexer tree and a buffer cell to reach its output that connects to the LB input.

	2-input LB	***4-input LB***
SB	(in) **BUF** MUX3 **BUF** (out)	(in) **BUF** MUX3 **BUF** (out)

Table 4.8: SB data path.

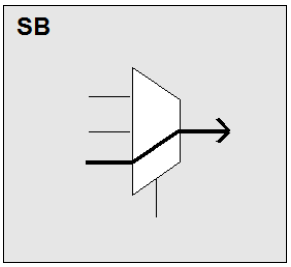


Figure 4.13: SB data path.

The SB data paths don't depend on the data arriving at the SB through a horizontal or vertical track and have the same intrinsic delay for all track input to track output paths. Due to the architecture that was chosen for the SB, each track input propagates to a possible track output through a MUX3 cell. The signal is input on the SB on a buffer and is output of the same block after a buffer cell. This data path is equal for both architectures. Table 4.8 and figure 4.13 show and highlight these data paths.

The tables described above present the intrinsic delays for all component blocks by presenting all possible data paths between FF cells on the sequential logic on the programmable core. Without a tool to automatically map a design and calculate the maximum operating frequency, with this detailed information, the designer can add the delays associated with a certain timing path crossing a certain number of basic blocks to have a quick estimation about the speed at which the digital function could operate on the reconfigurable core architecture.

On top of this analysis, an extra timing margin should be added to take into account the clock uncertainty, already discussed before.

Figure 4.14 presents the example of 3 possible data paths between 2-input LBs configured to operate as sequential logic, providing this way a sequential output for the logic function they hold.

Under the same assumptions for this estimation, the value for the maximum clock frequency that could be set to the reconfigurable core such that each data path would not cause timing violations is described on table 4.9 for a 90nm CMOS process. It is visible that while data paths A and B cross only routing and switch blocks between the two LBs that generate and capture the data signal, on the data path represented by C the signal

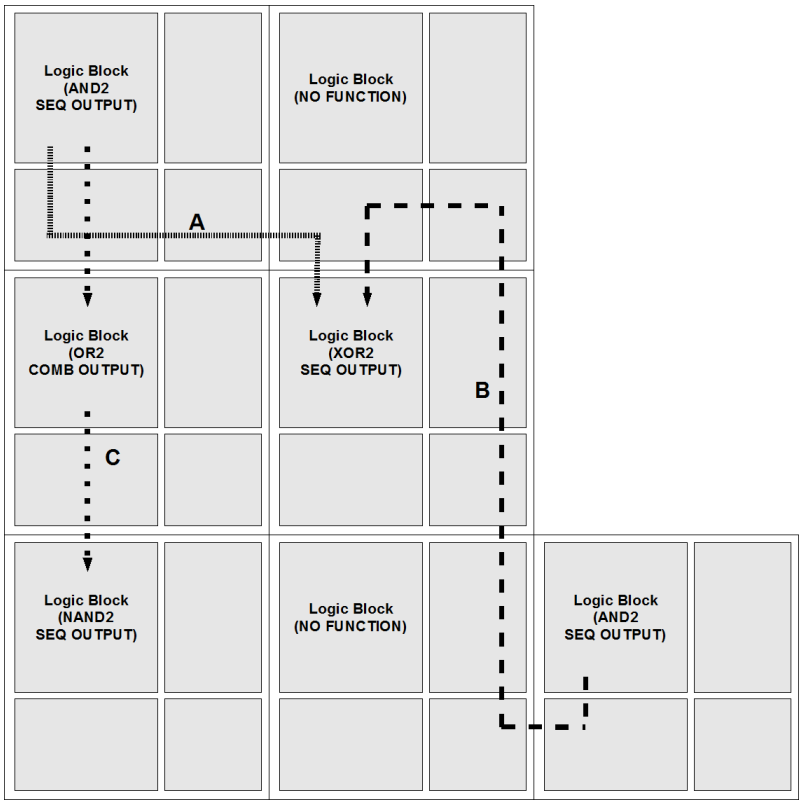


Figure 4.14: Three data delay paths on 2-input LB reconfigurable core.

crosses also a LB configured with a combinational digital logic function before reaching a different LB where the data signal is captured.

The estimated data takes into consideration the paths through the different blocks as presented before and it is not being considered that all 3 paths would exist on the core at the same time. If this would be the case, the bigger data path delay, would set the maximum operating frequency of the reconfigurable core.

Data path	**Delay**	**Maximum** F_{CLK}
A	$1.65ns$	$605MHz$
B	$2.47ns$	$405MHz$
C	$1.97ns$	$510MHz$

Table 4.9: Three data paths delay on a 2-input LB reconfigurable core.

The results presented have been confirmed through post-layout simulations of the data paths described. They are determined only by the 90nm CMOS technology in which the cores have been implemented.

After this analysis, it is worth taking into consideration that the values presented for these 3 examples correspond to synchronous timing arcs within the non-rectangular EPLC. A synchronous timing arc composed by combinational logic external to the core can constrain these maximum operating frequency values to a much smaller value, depending

on the length of the data path. In the same sense, a synchronous path crossing the core boundary, whose data path inside the core is long, will also result on a reduced maximum possible operating frequency.

4.4 Power

The power that the programmable core will dissipate is dependent on the technology in which the core is implemented, on the mapped logic function and on the core operating frequency. To have an idea of the worst case power consumption for the programmable core an evaluation was performed of the worst case power consumption for a single cluster block.

This evaluation is done through a post-layout simulation of the cluster block performed at gate-level with time-annotated netlists. The testbench created uses a clock pattern (101010) toggling on all data input signals. These signals are routed through all horizontal and vertical routing tracks and switch block circuits, arriving this way at all data output signals. All data lines are toggling at input clock frequency during the period of time over which the power evaluation was made. The activity data from this simulation is input to a power estimation tool, *Power Compiler*[3], that evaluates the power consumption during that activity period.

The results obtained are considered to be pessimistic in the sense that it is not expected for a programmable core to operate at the highest supported frequency of a single cluster nor to have all its data lines toggling at clock frequency a clock pattern. On a complete programmable core, the path between two FF cells on the sequential logic can span several cluster blocks due to complex mapped digital designs what will decrease the maximum clock operating frequency, thereby reducing the overall power consumption.

Table 4.10 shows the measured power consumption for the evaluation performed under the scenario described above for a cluster block implemented on a 90nm CMOS process. The same analysis performed with data from the predictive 45nm process didn't present accurate and reliable results as, for the same operating clock frequencies, the results measured were always higher than the ones for the bigger process node. This can point to an inaccurate internal power characterization of the standard cells provided for the predictive 45nm design kit.

4.5 Summary

With the architecture defined and the work flow presented it was possible to validate different non-rectangular programmable cores as well as to evaluate characteristics intrinsic to their architecture such as area, operating frequency and power consumption.

[3] http://www.synopsys.com/TOOLS/IMPLEMENTATION/RTLSYNTHESIS/Pages/PowerCompiler.aspx

Clock frequency	**Power (μW)**		
	Cell internal	**Net switching**	**Total**
833MHz	122.5	38.5	161
200MHz	28.3	8.8	37.1
100MHz	14.9	4.7	19.6
	Leakage		2.09

Table 4.10: 2-input LB cluster block power consumption.

The verification environment created targeted the validation of the basic blocks during their architecture development stage and the validation of the non-rectangular reconfigurable cores top-level. Simulations were performed with *ModelSim* simulator.

During the architecture development stage, specific testbenches were created to verify the basic blocks. With this, the capability of these blocks to hold a digital logic function and propagate its input and output data signals through the core is verified through functional simulations. On the same sense, the programming mode, propagation of global signals and the existing control logic were at this step also validated since they're exercised during these verifications.

For the reconfigurable core top-level verification, several cores with different shapes were created and individual cluster blocks were configured with simple digital logic functions. The mapping of these cluster blocks doesn't represent any particular digital design. Having configured the desired cluster blocks with the digital function and with the settings for the routing and propagation of input and output data signals, specific tasks were written on the core testbench to compare the data signal outputs, at the core boundary, with the expected values from the digital logic functions mapped on the cluster blocks. This methodology involves making use of both modes of operation of the core, to configure and use it, validating by this mean the overall core proposed architecture. At this step, both functional and post-layout simulations have been performed.

To illustrate the physical implementation of the non-rectangular cores created, one example layout is presented for a 90nm CMOS process. Making use of the floorplans automatically created for the basic blocks and the scripts to support their integration into the backend design flow, their layout was generated. Based on the desired shape for the programmable core as customized on the main configuration header file, the programmable core was automatically created on the layout editor through instantiation of cluster blocks and the cluster block layout created through the instantiation by abutment of the layout of the basic blocks previously created. Having the basic blocks layouts implemented for a certain technology node, any non-rectangular programmable core backend views can be quickly generated and validated through the automatic tools created.

A study has been also performed to evaluate the proposed reconfigurable core architecture characteristics such as area, operating frequency and power consumption. The data provided on this chapter helps the designer to quickly estimate, for a certain design

or core implementation, a value for these parameters.

The EPLC area can be estimated by estimating the area of the cluster blocks that compose it and multiplying that area by the number of cluster blocks on the reconfigurable core. The area of a cluster block is the result of the added area of the basic blocks that compose it. Taking into consideration the proposed architecture standard cells based implementation for each block and an area overhead margin for their respective physical implementation, it is possible to evaluate, based on technology information, the area used by each basic block. The area estimated for a cluster block implemented with a 2-input LB architecture is around 1600 μm^2 for a 90 nm process and 1000 μm^2 for the predictive 45 nm process.

The evaluation of the maximum operating frequency of the reconfigurable core was divided into the two modes of operation the core supports: programming and functional. For the programming mode of operation, the data path between two programmable FF cells dictates the maximum operating frequency for this mode. For a 90nm CMOS process, this value was estimated as to be around 1.4GHz (710ps delay path). This frequency of operation doesn't depend on the architectural choice made for the LB (2- or 4-input). For the functional mode of operation, the evaluation was divided into the study of the maximum clock frequency at which the core could operate, constrained only by the architecture proposed, and the evaluation of the maximum operating frequency of the core dependent on the design mapped and the respective data path delays between each two FF cells within the configured core. The maximum clock frequency constrained by the proposed architecture corresponds to a timing arc between two LBs, with the data signals crossing one HRB. For a 90nm CMOS process, this value is around 850MHz for a reconfigurable core based on a 2-input LB and around 740MHz in the case of a 4-input LB. With a digital design configured on the core, the maximum clock frequency for the design operation is dependent on the design itself and the connections configured within the core. For a 2-input LB EPLC implemented on the 90nm node, three different synchronous data paths have been evaluated. The maximum clock frequency obtained for each timing arc was around 605MHz for a path crossing routing and switch logic on 3 cluster blocks, 405MHz for a path crossing the same type of logic on 5 cluster blocks and around 510MHz for a path crossing not only routing and switch logic on 3 cluster blocks but also a logic block on one of the clusters configured with a combinational digital logic function. The values estimated have been confirmed through post-layout simulations of the same data paths configured on the EPLC. The study presented helps the designer to estimate the maximum clock frequency of its design early on the design cycle.

As for the power consumption evaluation, the worst case values were estimated for a single cluster block, with a clock pattern (101010) toggling on all cluster input and output data signals. The values were obtained from post-layout simulations of the cluster block. For a 2-input LB cluster, a power consumption of around 161μW for a clock frequency of 833MHz, 37.1μW for 200MHz and around 19.6μW for 100MHz.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The results obtained in this work demonstrate the feasibility of a flexible, technology-independent architecture for non-rectangular reconfigurable logic cores, that is supported by an automatic generation tool and that can be physically implemented using a standard digital design flow through commonly available CAD tools.

A fine-grain island-style programmable logic architecture has been selected as the basis of the architecture since this is the one that better fits non-rectangular shapes. Also for this purpose, a proper hierarchical organization of the circuits was defined to provide the core with flexibility and modularity.

The choice to implement it using standard cells was made based on their wide availability for all technology processes and nodes and the short and low-risk design cycle that can be achieved when implementing a fixed digital logic circuit with them as opposed to the inefficiency, low flexibility and time consuming design cycles associated with automatic full custom layouts.

On the other hand, the generic programmable circuit implemented carries with it a significant area, delay and power overhead what can compromise the amount of programmable logic within the SoC to a small amount.

The reconfigurable core is composed by the instantiation of cluster blocks. These are the same for all positions of the core, independent if they're surrounded by other cluster blocks or located on the core boundary.

Each cluster block contains one instance of the basic blocks: LB, VRB, HRB and SB. Each cluster block, this way, supports all the needed functionalities so that, together with other cluster blocks, create a programmable circuit mesh. These functionalities include the capability of holding a digital logic function and propagating its data input and output signals throughout the core.

The architecture of the basic blocks characterizes the overall proposed architecture for the EPLC:

- 2- or 4-input data signals (LB)
- Sequential or combinational data output and inverted data output (LB)
- Unidirectional 4 or 6 vertical and horizontal data tracks (HRB, VRB)
- Switch block based on the design of S. Wilton [20] (SB)
- Global clock, reset and programming mode enable signals

Tools have been created to support the reconfigurable core automatic generation. These tools generated the core frontend views, a testbench specific to the core and supports, with additional output data and utility tools, the creation of the core backend views.

Once the basic blocks layout is created and validated, any reconfigurable core with any shape can be defined and quickly made available through the use of the support tool created, *eplcReadIn.il*. The same basic blocks layout can be used to implemented, with no extra effort, the layout of any core on the same technology node.

Non-rectangular reconfigurable cores with different shapes ("S", "L", "T", "U") have been generated through the tools created and design flow presented. These cores have been validated through functional and post-layout simulations where some cluster blocks of the respective core were manually configured to support a simple digital logic function.

The verification environment, for both functional and post-layout simulations performed, targeted the validation of both programming and functional modes of operation. By configuring the core with some digital logic functions and with a defined set of connections for the data signals routing, and by then stimulating the mapped digital logic functions and comparing its outputs with the expected values (all at the core boundary), the entire core is validated as these operations exercise the complete core circuits.

Some studies were also performed to evaluate area, delay and power characteristics intrinsic to the proposed architecture and standard cells based implementation.

The area of the non-rectangular core corresponds to the area of the cluster blocks that compose it. The area estimated for a 2-input LB cluster block is around $1600\,\mu m^2$ for a 90 nm process and $1000\,\mu m^2$ for the predictive 45 nm process. A 100x100 core occupies around $10\,mm^2$ on the smaller technology node.

The maximum operating frequency at which the core could operate, constrained by the proposed architecture is around $850MHz$ for a reconfigurable core based on a 2-input LB and around $740MHz$ in the 4-input LB case for a 90nm CMOS process. The maximum operating frequency at which the core will be able to operate is design dependent but these values already present an upper limit.

A worst case power consumption value, obtained from a single cluster block with a clock pattern toggling on all its input data signals, resulted on the following values obtained for a 2-input LB cluster: 161μW for a clock frequency of 833MHz, 37.1μW for 200MHz and around 19.6μW for 100MHz.

This work proposes an architecture and a development flow. With common standard cells libraries and widely available and known to the industry CAD tools, the architecture proposed and the tools created can be easily used at academic level.

With this work, the gap between ASIC and FPGA developments has reduced and the best from both worlds has been joined.

5.2 Future Work

Several points of interest have been identified that deserve additional attention and a deeper investigation. These should be taken into consideration regarding future work related to the current proposal and consider that an effort should be taken in the future to continue to pursue the integration of programmable logic cores on SoC designs. The following items are based on the work developed on this thesis and serve as guidance for future work.

Future versions of *eplcGen* should be able to read in standard cells technology information and automatically map, through function recognition, the needed cells to the generic ones used on the core creation. On the current implementation, the main configuration file is used to control the technology standard cells used and link them to their generic instantiations.

eplcGen can be updated to support additional flexibility regarding the core architecture. In this sense, it should support logic blocks with more data inputs, the possibility to add tracks to the routing blocks independent of the number of data inputs on the logic blocks and the possibility of supporting two or more different logic functions with different output signals on the same logic block, increasing the cluster granularity.

Other architectural changes that can be discussed as future work are related with the possibility of having data signals available on all edges of the logic block, what implies changes to the surrounding blocks and the architectural changes involved in supporting single cluster macrocells, for specific functions, whose width can be multiple of a single cluster block.

An important step in a future work is to develop or adapt existing mapping tools for an automated procedure to map digital designs on the reconfigurable core. This will allow evaluating the proposed architecture functionality and routability by quickly mapping on it several different complex digital designs. The evaluation of these and other characteristics should also help the designer decide on the integration of programmable logic on the system and help to change the programmable fabric as needed to meet the designer and the system needs. Only after these automatic mapping tools exist for the proposed architecture, an

approach can be taken to investigate and document performance and routability issues with the propose core architecture, leading afterwards to improvements and upgrades.

The utility tool created to parse the bitstream configuration file can be updated to additionally perform sanity checks on the customized bitstream to avoid line conflicts. For example, to avoid both output and inverted output of a certain logic block from being driven to the same horizontal track on the routing channel.

Automated tools should be able to generate custom programmable logic cores taking into consideration the application domain the devices will serve and the fabric where they'll be integrated into. This will allow, for example, to gain area if a design is mostly combinational and control logic where the cluster blocks can be created without the FF cell for the sequential output and the core can be implemented by slices (columns of cluster blocks) where clusters with sequential logic would only be present on some columns.

A thorough comparison of area, speed and power characteristics of the proposed programmable logic architecture and other commercially available circuits should be made. With the current and future technology advances and improvements, it is important for architects and designers to continue to think of methods to reduce area and power consumption while maintaining or increasing the circuits' performances.

For optimization of these characteristics, standard cells libraries can be complemented with additional cells, specifically created to support the reconfigurable digital core needs and functionalities with less area and higher performance.

Finally, although the proposed architecture has some power dissipation issues, the minimization of this aspect was not the goal of this work. Programmable logic implementations have the expensive disadvantage of severe power dissipation. For the embedded programmable solutions integration on future systems to be as high as possible, power consumption should be revised and kept at the minimum possible. One possible solution is the implementation of a scheme that shuts down part of the programmable core not being used.

Annex A

eplcGen - Automatic Core Generation Tool

A.1 Main Configuration Header

The main configuration header file presented below is configured for a 2-input LB architecture. Besides the relation between the generic and technology standard cells used on this particular core, the basic blocks name and size are defined as well as the core shape and number of cluster blocks and the digital core power and ground pin names and metal layer.

```
############################################################
# 1) Input technology (T) stdcells under each GEPLC (G) stdcell
# 2) Input Logic Block configuration
# 3) clk = clock signal, pmode = programming mode signal, rstz = reset signal
# 4) EPLC (embedded programmable logic core) shape drawn with + (cluster) and - (empty cluster)
# 5) EPLC must always be drawn with + and - to form a square or rectangular shape
############################################################

## STDCELLS
G: SDFFR     d q rstz pmode inp clk
T: SDFFRHQX1 D Q RN   SE    SI  CK

G: DFFR     d q rstz clk
T: DFFRHQX1 D Q RN   CK

G: BUF   a y
T: BUFX2 A Y

G: MUX2  a b y sel
T: MX2X1 A B Y S0

G: MUX3  a b c y s0 s1
T: MX3X1 A B C Y S0 S1

G: INV   a y
T: INVX2 A Y
```

```
G: AND2   a b y
T: AND2X2 A B Y

G: NOR2   a b y
T: NOR2X2 A B Y

G: TBUF   a oe y
T: TBUFX2 A OE Y

## LOGIC BLOCK
ID: LOGIC BLOCK 21
MODULE: lb21
IN: 2
X: 20
Y: 15

## ROUTING BLOCK
ID: ROUTING BLOCK 21
MODULE: rb21

## SWITCH BLOCK
ID: SWITCH BLOCK 21
MODULE: sb21
X: 25
Y: 30

## EPLC CORE
++++++-----------
++++++-----------
++++++++++++++++++
++++++++++++++++++
++++++++++++++++++
++++++++++++++++++
----------+++++++
----------+++++++
----------+++++++

## OTHER
PIN: ME2
POWER: VDD
TOPPOWERPIN: ME4
GROUND: VSS
SIDEPOWERPIN: ME3
```

A.2 Core Creation Report

eplcGen, when executed, outputs a report file that contains the number of cells used on each basic block, the programmable core shape identified and the overall number of cells used by the non-rectangular reconfigurable core. The example below refers to the main configuration header file presented on section A.1.

```
-I- Creating Logic Block
-I- Creating Routing Block
-I- Creating Switch Block
-I- Creating Cluster Block
-I- Creating 17x9 irregular EPLC Block
-I- Creating Wrapper for 17x9 irregular EPLC Block
-I- Creating Testbench for 17x9 irregular EPLC Block
```

```
####################
## LOGIC BLOCK
####################
## LOGIC BLOCK 21 - lb21
##
## Cells Report:
## FF  : 6 (total number of  FFs)
## FF  : 5 (number of programmable FFs)
## BUF : 8
## MUX2: 4
## INV : 2
## AND2: 3
##
####################
## ROUTING BLOCK
####################
## ROUTING BLOCK 21 - rb21
##
## Cells Report:
## FF  : 12
## BUF : 30
## TBUF: 12
## NOR2: 4
## MUX2: 6
## MUX3: 0
## AND2: 1
##
####################
## SWITCH BLOCK
####################
## SWITCH BLOCK 21 - sb21
##
## Cells Report:
## FF  : 16
## BUF : 24
## TBUF: 0
## MUX3: 8
## AND2: 1
##
####################
## EPLC SHAPE
####################
##
## Width:  17
## Height: 9
##
## Irregular shape identified
##
## ++++++------------
## ++++++------------
## ++++++++++++++++++
## ++++++++++++++++++
## ++++++++++++++++++
## ++++++++++++++++++
## ----------+++++++
## ----------+++++++
## ----------+++++++
```

```
##
####################
## EPLC LOGIC
####################
##
## Cells Report:
## FF  : 3434 (total number of FFs)
## FF  : 3333 (number of programmable FFs)
## BUF : 6262
## TBUF: 1212
## MUX2: 1010
## MUX3: 808
## INV : 202
## AND2: 505
## NOR2: 404
##
####################
```

A.3 Programming Bitstream - 2-input Single Cluster Core

To help the designer map a digital logic function on each cluster block on the reconfigurable core, a template exists that contains the bits that need to be programmed for each cluster block. This file can be automatically processed to create the serial input bitstream that is used to configure the programmable core. An example for a core containing a single cluster block created in a 2-input LB architecture is shown below.

```
/*************************************************************************
** Programming Bitstream for eplc_1x1 **
****************************************
* LB
*   4'bXXXX - Logic table output value for in1, in0 input signals (2'b00, 2'b01, 2'b10, 2'b11 respectively)
*   1'bX - Logic table sequential (1'b1) or combinational (1'b0) output selection
* SB
*   16'bXX_XX_XX_XX_XX_XX_XX_XX - Switch block selection table (2'b11 not a valid value)
* HRB
*   2'bXX - in0 output from horizontal tracks
*   2'bXX - in1 output from horizontal tracks
*   4'bXXXX - Logic block output to horizontal tracks
*   4'bXXXX - Logic block inverted output to horizontal tracks
*
* For more detailed information about cluster bitstreams, please check bs.help
*****************************************************/

** y0x0
* LB
4'bXXXX
1'bX
* SB
16'bXX_XX_XX_XX_XX_XX_XX_XX
* HRB
2'bXX
2'bXX
4'bXXXX
4'bXXXX
```

A.4 Bitstream Help - 2-input Single Cluster Core

The detailed description of each bit functionality, to be configured on the file presented in A.3, is presented for a 2-input LB architecture.

```
/*****************************************************************************
** Bitstream Help - Cluster **
*****************************/
LB
  4'bXXXX
    Logic table output value for in1, in0 input signals.
    Each bit of this 4-bit word corresponds respectively to the output when
        in0,in1 = 2'b00, 2'b01, 2'b10, 2'b11 respectively

  1'bX
    Logic table sequential (1'b1) or combinational (1'b0) output selection

SB
  16'bXX_XX_XX_XX_XX_XX_XX_XX
    Switch block selection table (2'b11 not a valid value)
    16'bXX_.._.._.._.._.._..
    2'b00 - track0_left_right_input -> track0_left_right_output
    2'b01 - track2_bottom_top_input -> track0_left_right_output
    2'b10 - track3_top_bottom_input -> track0_left_right_output
    2'b11 - NOT VALID

    16'b.._XX_.._.._.._.._..
    2'b00 - track0_left_right_input -> track0_bottom_top_output
    2'b01 - track0_bottom_top_input -> track0_bottom_top_output
    2'b10 - track1_right_left_input -> track0_bottom_top_output
    2'b11 - NOT VALID

    16'b.._.._XX_.._.._.._..
    2'b00 - track0_left_right_input -> track3_top_bottom_output
    2'b01 - track3_right_left_input -> track3_top_bottom_output
    2'b10 - track3_top_bottom_input -> track3_top_bottom_output
    2'b11 - NOT VALID

    16'b.._.._.._XX_.._.._..
    2'b00 - track2_left_right_input -> track1_top_bottom_output
    2'b01 - track1_right_left_input -> track1_top_bottom_output
    2'b10 - track1_top_bottom_input -> track1_top_bottom_output
    2'b11 - NOT VALID

    16'b.._.._.._.._XX_.._..
    2'b00 - track2_left_right_input -> track2_left_right_output
    2'b01 - track0_bottom_top_input -> track2_left_right_output
    2'b10 - track1_top_bottom_input -> track2_left_right_output
    2'b11 - NOT VALID

    16'b.._.._.._.._.._XX_..
    2'b00 - track2_left_right_input -> track2_bottom_top_output
    2'b01 - track2_bottom_top_input -> track2_bottom_top_output
    2'b10 - track3_right_left_input -> track2_bottom_top_output
    2'b11 - NOT VALID

    16'b.._.._.._.._.._.._XX_..
    2'b00 - track1_top_bottom_input -> track3_right_left_output
```

```
    2'b01 - track2_bottom_top_input -> track3_right_left_output
    2'b10 - track3_right_left_input -> track3_right_left_output
    2'b11 - NOT VALID

    16'b.._.._.._.._.._.._.._XX
    2'b00 - track3_top_bottom_input -> track1_right_left_output
    2'b01 - track0_bottom_top_input -> track1_right_left_output
    2'b10 - track1_right_left_input -> track1_right_left_output
    2'b11 - NOT VALID

HRB
  2'bXX
    in0 output from horizontal tracks
    2'b00 - track0_left_right_input
    2'b01 - track1_right_left_input
    2'b10 - track2_left_right_input
    2'b11 - track3_right_left_input

  2'bXX
    in1 output from horizontal tracks
    2'b00 - track0_left_right_input
    2'b01 - track1_right_left_input
    2'b10 - track2_left_right_input
    2'b11 - track3_right_left_input

  4'bXXXX
    Logic block output to horizontal tracks
    4'b1000 - track0_left_right_input
    4'b0100 - track2_left_right_input
    4'b0010 - track1_right_left_input
    4'b0001 - track3_right_left_input

  4'bXXXX
    Logic block inverted output to horizontal tracks
    4'b1000 - track0_left_right_input
    4'b0100 - track2_left_right_input
    4'b0010 - track1_right_left_input
    4'b0001 - track3_right_left_input
```

A.5 wrapperMap.rpt - 2x2 Programmable Core

The automatically generated wrapper cell that surrounds the reconfigurable core groups the signals into buses to help the designer integrate the core with the surrounding macros. This relation is output by *eplcGen* tool to a report file. The contents of this file, for a 2x2 reconfigurable core, is presented below where the signals on the left correspond to the wrapper interface and the signals on the right correspond to the interface of the boundary cluster blocks.

```
clk_in[0] = clk_in_y0x0
pmode_in[0] = pmode_in_y0x0
rstz_in[0] = rstz_in_y0x0
clk_in[1] = clk_in_y0x1
pmode_in[1] = pmode_in_y0x1
rstz_in[1] = rstz_in_y0x1
clk_out[0] = clk_out_y1x0
pmode_out[0] = pmode_out_y1x0
rstz_out[0] = rstz_out_y1x0
clk_out[1] = clk_out_y1x1
```

```
pmode_out[1] = pmode_out_y1x1
rstz_out[1] = rstz_out_y1x1
datain[0] = in0_y0x0
datain[1] = in1_y0x0
datain[2] = in0_y0x1
datain[3] = in1_y0x1
dataout[0] = in0_output_y1x0
dataout[1] = in1_output_y1x0
dataout[2] = in0_output_y1x1
dataout[3] = in1_output_y1x1
vtrackdataout[0] = track0_bottom_top_output_y0x0
htrackdatain[0] = track0_left_right_input_y0x0
htrackdataout[0] = track1_right_left_output_y0x0
vtrackdatain[0] = track1_top_bottom_input_y0x0
vtrackdataout[1] = track2_bottom_top_output_y0x0
htrackdatain[1] = track2_left_right_input_y0x0
htrackdataout[1] = track3_right_left_output_y0x0
vtrackdatain[1] = track3_top_bottom_input_y0x0
vtrackdataout[2] = track0_bottom_top_output_y0x1
htrackdataout[2] = track0_left_right_output_y0x1
htrackdatain[2] = track1_right_left_input_y0x1
vtrackdatain[2] = track1_top_bottom_input_y0x1
vtrackdataout[3] = track2_bottom_top_output_y0x1
htrackdataout[3] = track2_left_right_output_y0x1
htrackdatain[3] = track3_right_left_input_y0x1
vtrackdatain[3] = track3_top_bottom_input_y0x1
vtrackdatain[4] = track0_bottom_top_input_y1x0
htrackdatain[4] = track0_left_right_input_y1x0
htrackdataout[4] = track1_right_left_output_y1x0
vtrackdataout[4] = track1_top_bottom_output_y1x0
vtrackdatain[5] = track2_bottom_top_input_y1x0
htrackdatain[5] = track2_left_right_input_y1x0
htrackdataout[5] = track3_right_left_output_y1x0
vtrackdataout[5] = track3_top_bottom_output_y1x0
vtrackdatain[6] = track0_bottom_top_input_y1x1
htrackdataout[6] = track0_left_right_output_y1x1
htrackdatain[6] = track1_right_left_input_y1x1
vtrackdataout[6] = track1_top_bottom_output_y1x1
vtrackdatain[7] = track2_bottom_top_input_y1x1
htrackdataout[7] = track2_left_right_output_y1x1
htrackdatain[7] = track3_right_left_input_y1x1
vtrackdataout[7] = track3_top_bottom_output_y1x1
```

References

[1] International Technology Roadmap for Semiconductors. http://www.itrs.net/Links/2008ITRS/Update/2008Tables_FOCUS_A.xls, 2007.

[2] S.D. Brown, R.J. Francis, J. Rose, and Z.G. Vranesic. *Field-Programmable Gate Arrays*. Springer, 1992.

[3] Ian Kuon, Russell Tessier, and Jonathan Rose. Fpga architecture: Survey and challenges. *Found. Trends Electron. Des. Autom.*, pages 135–253, 2008.

[4] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli. Architecture of Field-Programmable Gate Arrays. In *Proceedings of the IEEE*, pages 1013–1029, 1993.

[5] Paul Chow, Jonathan Rose, Soon Ong Seo, Kevin Chung, Gerard Páez-Monzón, and Immanuel Rahardja. The Design of an SRAM-based Field-Programmable Gate Array - Part I: Architecture. *IEEE Trans. Very Large Scale Integr. Syst.*, 1999.

[6] Paul Chow, Soon Ong Seo, Jonathan Rose, Kevin Chung, Gerard Páez-Monzón, and Immanuel Rahardja. The Design of an SRAM-based Field-Programmable Gate Array - Part II: Circuit Design and Layout. *IEEE Trans. Very Large Scale Integr. Syst.*, pages 321–330, 1999.

[7] Altera FPGA Architecture White Paper July 2006 ver 1.0. http://www.altera.co.jp/literature/wp/wp-01003.pdf, 2006.

[8] R. J. Francis, D. Lewis, J. Rose, and P. Chow. Architecture of Field-Programmable Gate Arrays: The Effect of Logic Functionality on Area Efficiency. *IEEE Journal of Solid-State Circuits*, 25:1217–1225, October 1990.

[9] J. Kouloheris and A. El Gamal. PLA-based FPGA Area versus Cell Granularity. *Proceedings of Custom Integrated Circuits Conference*, pages 4.3.1–4.3.4, May 1992.

[10] Sinan Kaptanoglu, Greg Bakker, Arun Kundu, Ivan Corneillet, and Ben Ting. A New High Density and Very Low Cost Reprogrammable FPGA Architecture. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 3–12. ACM, 1999.

[11] Elias Ahmed and Jonathan Rose. The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density. In *FPGA '00: Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, pages 3–12. ACM, 2000.

[12] S. P. Young, T. J. Bauer, K. Chaudhary, and S. Krishnamurthy. FPGA repeatable interconnect structure with bidirectional and unidirectional interconnect lines. August 1999.

[13] David Lewis, Vaughn Betz, David Jefferson, Andy Lee, Chris Lane, Paul Leventis, Sandy Marquardt, Cameron Mcclintock, Bruce Pedersen, Giles Powell, Srinivas Reddy, Chris Wysocki, Richard Cliff, and Jonathan Rose. The Stratix Routing and Logic Architecture. In *FPGA '03: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 12–20. ACM Press, 2003.

[14] G. Lemieux, E. Lee, M. Tom, and A. Yu. Directional and single-driver wires in FPGA interconnect. In *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*, pages 41–48, 2004.

[15] University Of Toronto, Vaughn Betz, and Jonathan Rose. On Biased and Non-Uniform Global Routing Architectures and CAD Tools for FPGAs. Technical report, University of Toronto, 1996.

[16] M. Imran Masud and Steven J. E. Wilton. A New Switch Block for Segmented FPGAs. In *FPL '99: Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*, pages 274–281. Springer-Verlag, 1999.

[17] Yao-Wen Chang, D. F. Wong, and C. K. Wong. Universal Switch Modules for FPGA Design. *ACM Trans. Des. Autom. Electron. Syst.*, pages 80–101, 1996.

[18] Yao-Wen Chang, D. F. Wong, and C. K. Wong. Universal Switch-Module Design for Symmetric-Array-Based FPGAs. In *FPGA '96: Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays*, pages 80–86. ACM, 1996.

[19] Guy G. F. Lemieux, Stephen D. Brown, and Daniel Vranesic. On two-step Routing for FPGAS. In *ISPD '97: Proceedings of the 1997 international symposium on Physical design*, pages 60–66. ACM, 1997.

[20] Steven J. E. Wilton. *Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memory.* PhD thesis, Department of Computer and Electrical Engineering, University of Toronto, 1997.

[21] Reinaldo A. Bergamaschi and John Cohn. The A to Z of SoCs. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 790–798. ACM, 2002.

[22] Behrooz Zahiri. Structured ASICs: Opportunities and Challenges. In *ICCD '03: Proceedings of the 21st International Conference on Computer Design*, page 404. IEEE Computer Society, 2003.

[23] Ian Kuon and Jonathan Rose. Measuring the Gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26:203–215, February 2007.

[24] David Chinnery and Kurt Keutzer. *Closing the Gap between ASIC and Custom: Tools and Techniques for High-Performance ASIC Design.* Springer, 2002.

[25] Willaim J. Dally and Andrew Chang. The Role of Custom Design in ASIC Chips. In *DAC '00: Proceedings of the 37th Annual Design Automation Conference*, pages 643–647. ACM, 2000.

[26] Andrew Chang and William J. Dally. Explaining the Gap between ASIC and Custom Power: a Custom Perspective. In *DAC '05: Proceedings of the 42nd annual Design Automation Conference*, pages 281–284. ACM, 2005.

[27] D. G. Chinnery and K. Keutzer. Closing the Power Gap between ASIC and Custom: an ASIC Perspective. In *DAC '05: Proceedings of the 42nd annual Design Automation Conference*, pages 275–280. ACM, 2005.

[28] Victor Olubunmi Aken'Ova. *Bridging the Gap between Soft and Hard eFPGA Design.* PhD thesis, Department of Electrical and Computer Engineering, University of British Columbia, 2005.

[29] James Cheng-Huan Wu. *Implementation Considerations for Soft Embedded Programmable Logic Cores.* PhD thesis, University of British Columbia, 2004.

[30] Steven J. E. Wilton, Noha Kafafi, James C. H. Wu, Kimberly A. Bozman, Victor O. Aken'Ova, and Resve Saleh. Design Considerations for Soft Embedded Programmable Logic Cores. In *IEEE Journal of Solid-State Circuits*, pages 485 – 497, 2005.

[31] J.C.H. Wu, V. Aken'Ova, S.J.E. Wilton, and R. Saleh. SoC Implementation Issues for Synthesizable Embedded Programmable Logic Cores. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 21–24, 2003.

[32] Andy Yan and Steven J.E. Wilton. Sequential Synthesizable Embedded Programmable Logic Cores for System-on-Chip. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 435 – 438, 2004.

[33] Andy Yan and Steven J.E. Wilton. Product-Term Based Synthesizable Embedded Programmable Logic Cores. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 474 – 488, 2006.

[34] Noha Kafafi, Kimberly Bozman, and Steven J. E. Wilton. Architectures and Algorithms for Synthesizable Embedded Programmable Logic Cores. In *FPGA '03: Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, pages 3–11. ACM, 2003.

[35] Victor Aken'Ova, Guy Lemieux, and Resve Saleh. An Improved Soft eFPGA Design and Implementation Strategy. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 179 – 182, 2005.

[36] Victor Aken Ova and Resve Saleh. A "Soft++" eFPGA Physical Design Approach with Case Studies in 180nm and 90nm. In *ISVLSI '06: Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, page 103. IEEE Computer Society, 2006.

[37] Kun-Cheng Wu and Yu-Wen Tsai. Structured ASIC, Evolution or Revolution? In *ISPD '04: Proceedings of the 2004 International Symposium on Physical Design*, pages 103–106. ACM, 2004.

[38] P. Jamieson and J. Rose. Enhancing the Area-Efficiency of FPGAs with Hard Circuits Using Shadow Clusters. *IEEE International Conference on Field Programmable Technology (FPT06)*, pages 1–8, December 2006.

[39] Paul S. Zuchowski, Christopher B. Reynolds, Richard J. Grupp, Shelly G. Davis, Brendan Cremen, and Bill Troxel. A Hybrid ASIC and FPGA Architecture. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 187–194. ACM, 2002.

[40] Ian Carlos Kuon. *Automated FPGA Design, Verification and Layout*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, 2004.

[41] Ian Kuon, Aaron Egier, and Jonathan Rose. Design, Layout and Verification of an FPGA using Automated Tools. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 215–226. ACM, 2005.

[42] Shawn Phillips and Scott Hauck. Automatic Layout of Domain-Specific Reconfigurable Subsystems for System-on-a-Chip. In *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 165–173. ACM, 2002.

[43] Shawn A. Phillips. *Automating Layout of Reconfigurable Subsystems for Systems-on-a-Chip*. PhD thesis, University of Washington, 2004.

[44] Mark Holland and Scott Hauck. Automatic Creation of Domain-Specific Reconfigurable CPLDs for SoC. In *FCCM '05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 289–290. IEEE Computer Society, 2005.

[45] Ketan Padalia, Ryan Fung, Mark Bourgeault, Aaron Egier, and Jonathan Rose. Automatic Transistor and Physical Design of FPGA Tiles from an Architectural Specification. In *FPGA '03: Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, pages 164–172. ACM, 2003.

[46] Tony Yau-Wai Wong. *Non-Rectangular Embedded Programmable Logic Cores*. PhD thesis, Department of Electrical and Computer Engineering, University of British Columbia, 1998.

[47] Tony Wong and Steven J.E. Wilton. Placement and Routing for Non-Rectangular Embedded Programmable Logic Cores in SoC Design. In *Proceedings of IEEE International Conference on Field-Programmable Technology (FPT '04)*, pages 65–72, 2004.

[48] Sumanta Chaudhuri, Sylvain Guilley, Florent Flament, Philippe Hoogvorst, and Jean-Luc Danger. An 8x8 run-time Reconfigurable FPGA Embedded in a SoC. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 120–125. ACM, 2008.

[49] S.J.E. Wilton and R. Saleh. Programmable Logic IP cores in SoC design: Opportunities and Challenges. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 63–66, 2001.

[50] J. Greenbaum. Reconfigurable Logic in SoC Systems. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 5–8, 2002.

[51] V. Betz. *Architecture and CAD for Speed and Area Optimizations of FPGAs*. PhD thesis, University of Toronto, 1998.

[52] J. Rose and S. Brown. Flexibility of Interconnection Structures for Field-Programmable Gate Arrays. In *IEEE Journal of Solid State Circuits*, pages 277–282, 1991.

[53] Cadence SKILL Programming Language. http://en.wikipedia.org/wiki/Cadence_SKILL.

[54] Timothy J. Barnes. SKILL: A CAD system extension language. In *DAC '90: Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 266–271. ACM, 1990.

[55] Free 45nm PDK, North Carolina State University. http://www.eda.ncsu.edu/wiki/FreePDK45:Contents.

[56] Free 45nm PDK Standard Cells Library, Oklahoma State University. http://vcag.ecen.okstate.edu/projects/scells/.