FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Automatic implementation of a re-configurable logic over ASIC design flow

José Delfim Ribeiro Valverde

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Advisor: Prof. João Canas Ferreira

Second Advisor: Eng. Luis Cruz

July 4, 2017

Resumo

A indústria semicondutora tem enfrentado desafios devido à evolução dos circuitos integrados (*Integrated Circuits*, *IC*) para *System–on–a–Chip* (*SoC*), cujo *design* se encontra cada vez mais complexo. Assim, tem-se intensificado a necessidade de se efetuarem validações extensas dos circuitos, antes do processo de fabrico, de forma a assegurar um circuito produzido mais correto.

Simultaneamente, com o intuito de dar resposta à atual procura de ciclos de produção mais rápidos, têm sido executados importantes testes de interoperabilidade no circuito final, em silício. Contudo, destes testes poderão resultar erros inesperados, obrigando à completa recusa do *chip* produzido e à necessidade de se iniciar novamente todo o ciclo de produção, despendendo tempo e recursos extra.

Uma das soluções para este tipo de problemas corresponde à substituição do circuito original, de lógica fixa, por um implementado em lógica reconfigurável.

Ao recorrer a este tipo de arquiteturas, o projetista fica habilitado a realizar pequenas alterações, localmente e em tempo útil, alterando algumas funcionalidades e corrigindo pequenos erros, resultantes dos testes de interoperabilidade efetuados em silício. De forma a incluir circuitos reconfiguráveis em projetos para *SoC*, alguns fornecedores produzem e vendem núcleos programáveis do tipo *hard* para serem incluídos em projetos. Em oposição, outros fornecedores utilizam uma abordagem chamada *soft*, na qual vendem uma versão *RTL* (*Register Transfer Level*) da lógica programável, que pode ser sintetizada utilizando uma abordagem *standard cell*.

Porém, uma vez que as soluções geradas por estes métodos estão dependentes de decisões dos próprios fornecedores, nenhum destes métodos permite criar dispositivos lógicos que estejam otimizados para as aplicações de um projetistas de circuitos integrados.

Portanto, surge a necessidade de se estudar um novo fluxo de projeto, de modo a que este possa ser integrado num processo habitual de produção efetuado pelo projetista, e reutilizado em múltiplos projetos e tecnologias, de forma a criar uma arquitetura programável otimizada para diferentes implementações.

Desta forma, fazendo uso do fluxo de projeto produzido no âmbito desta dissertação, foi possível comparar a área, consumo energético e velocidade lógica de um dispositivo programável com um não – programável. Nesta análise observou-se um aumento médio do tamanho na ordem das 380 vezes, uma diminuição da velocidade lógica em 4 vezes e um aumento da ordem das 40 vezes do consumo energético.

Tendo em conta que a abordagem utilizada se baseou numa estratégia em *standard cell*, na qual existe uma falta de otimização dos elementos utilizados numa perspetiva da criação de um dispositivo programável, denotou-se que os valores obtidos estavam de acordo com o expectável. Assim sendo, devida à elevada magnitude dos valores encontrados, esta solução apresenta-se como impraticável e pouco apelativa para a indústria. Todavia foi demonstrado que é possível melhorar o fluxo de projeto do projetista de *ICs* de forma a que este possa produzir uma arquitetura programável sem ter de mudar o seu processo normal de produção extensivamente.

Abstract

The available density and complexity on Integrated Circuits (IC) have been increasing, following the improvement of technologies to design and fabric ICs leading to a challenging evolution, in the complex nature of digital ICs on System on a Chip (SoC) design, to the semiconductor industry. As such an extensive validation before fabrication has become a demanding pressure to ensure design correctness for the produced circuit.

At the same time, with the current demand for faster turnaround development cycles, major interoperability tests are already performed in actual silicon, as errors can result in testing, the disposal of the produced chip and the need to create a new production cycle results in time and resources waste.

One of the clear solutions to this problem is the replacement of the original fixed logic with a reprogrammable one. As this type of architecture can empower the designer with the ability to perform minor updates, on site, changing minor errors resulted from interoperability tests already performed in silicon or/and adding some necessary, minor functionalities.

To add reconfigurability, some vendors sell programmable cores in a "hard" macro style, that can be included in the developer design; others, supply Register-transfer level (RTL) versions of their programmable logic that can be synthesized using standard cell approach often called a "soft" approach. However, none of this method answer to the question of replacing the fixed logic standard application produced from a standard IC designer flow, into a re-configurable logic, that can be used

So, a new workflow must be studied, so that can be integrated into the standard work of the IC developer, that can be used in multiple projects and technologies, to create a flexible architecture suitable for different implementations.

Using the design flow produced in this dissertation, the programmable device that was produced was compared to the non-programmable architecture, where the solution exhibited an average size increase of 380 times, a speed decrease of 4 times, and a power increase of around 40 times.

These values are high and be impractical and non-appealing to an industry standard. Nonetheless, this work found that was possible to create a design flow that could empower the IC developer to create, from the same Verilog representation, a programmable device in his standard IC workflow.

Acknowledgments

First I would like to thank my Supervisors Prof. João Canas Ferreira and to Eng. Luis Cruz, for all the help, encouragement and suggestions that helped me to research, and developing and writing this dissertation. All Synopsys Engineers, but especially to Helder Campos and to João Martins, that were timeless help me in many problems faced during this dissertation. Also, I must express my profound gratitude to my parents and to Beatriz, for all advice, unfailing support, and encouragement throughout my years of study and through the process of developing this dissertation. Obrigado.

José Delfim Ribeiro Valverde

"The mind is like a parachute...
It only works if we keep it open."

Albert Einstein

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Objectives . . . 2
- 1.3 Thesis Organization . . . 4

2 Literature Review and Background Work **7**
- 2.1 Application-Specific Integrated Circuits (ASICs) . . . 9
- 2.2 Introduction to Programmable Logic Devices (PLD) . . . 10
  - 2.2.1 Programmable Logic applications . . . 10
- 2.3 Field Programmable Gate Array (FPGAs) . . . 11
  - 2.3.1 FPGA Architecture . . . 12
  - 2.3.2 FPGA Logic Blocks . . . 13
  - 2.3.3 Embedded FPGA . . . 16
- 2.4 Computer-aided design (CAD) Flows . . . 17
  - 2.4.1 ASIC Design Flow . . . 17
  - 2.4.2 FPGA Design Flow . . . 18
  - 2.4.3 eFPGA Design Flow . . . 19
- 2.5 Summary . . . 21

3 Programmable Device Generation Design Flow **23**
- 3.1 Global Considerations . . . 24
- 3.2 Phase 1: VTR . . . 25
- 3.3 Phase 2: Pre-VTR . . . 28
- 3.4 Phase 3: VTR to Programmable Device Generator . . . 28
- 3.5 Phase 4: Reconfigurability addition . . . 30
- 3.6 Conclusions . . . 31

4 Description of Tools used in the Design Flow **35**
- 4.1 Design Compiler . . . 36
  - 4.1.1 GTECH Python Parser . . . 38
- 4.2 VTR flow . . . 38
  - 4.2.1 FPGA Architecture Description Input File . . . 39
  - 4.2.2 ODIN 2 . . . 43
  - 4.2.3 ABC . . . 44
  - 4.2.4 VPR . . . 45
- 4.3 VTR to Programmable Device Generator . . . 46
  - 4.3.1 Input Reading . . . 47
  - 4.3.2 Internal Architectural Representation . . . 47

4.3.3 Bitstream Creation . . . 48
4.3.4 Output generation . . . 50
4.4 Conclusions . . . 51

5 Description of a Programmable Architecture 53
5.1 Specific programmable cores . . . 53
5.2 FPGA Architectures and Production . . . 55
5.2.1 Complex Logic Blocks . . . 55
5.2.2 FPGA Routing . . . 58
5.2.3 Configuration Bits . . . 59
5.2.4 Summary . . . 60
5.3 Routing Structure Studies . . . 60
5.3.1 Routing Structure based on basic Modules . . . 61
5.3.2 Routing Structure based on Multiplexers . . . 62
5.4 Architecture Decisions Details . . . 63
5.4.1 Architectures Comparison . . . 63
5.4.2 Architecture Details . . . 65
5.5 Functional correctness . . . 66
5.6 Conclusions . . . 67

6 Implementation and Results 69
6.1 Programmable Device Design Flow . . . 70
6.1.1 Creation of Programmable Device . . . 71
6.1.2 ASIC Design Flow . . . 71
6.2 Implementation process . . . 72
6.2.1 Design problems . . . 72
6.2.2 Synthesis approach . . . 75
6.3 Comparison Metrics and Generated Results . . . 77
6.3.1 Measures to compare implementations . . . 77
6.3.2 Comparison between ASIC fixed logic to the Programmable device implementation . . . 77
6.3.3 Comparison between "soft" and "hard" eFPGA to the Programmable Device implementation . . . 82
6.3.4 Modification experiment . . . 84
6.4 Conclusion . . . 86

7 Conclusions and Future work 89
7.1 Future developments . . . 91

A XML file used 93

B Example Outputs Generated by VTR to Programmable Device Generator 97
B.1 SDC . . . 97
B.2 Programmable device RTL . . . 98

C Circuit used to test Reprogramability 103
C.1 Original Verilog . . . 103
C.2 Modification A . . . 105
C.3 Modification B . . . 105

C.4 Modification D 106
C.5 Modification C 108

D Design Flow Example **109**
D.1 Work Directory 109
D.2 Running the generation script 110
D.3 Script description 110
D.3.1 Initial configuration 110
D.3.2 Pre-VTR 111
D.3.3 Programmable device generation 112
D.3.4 Bitstream generation 114
D.3.5 VTR to Programmable device generator 115

References **117**

List of Figures

1.1 Example of the created device 4

2.1 Example of a SoC design 8
2.2 FPGA architecture 12
2.3 Minimum NAND ULB 15
2.4 FPGA basic logic element and logic cluster 16
2.5 ASIC design flow 18
2.6 A typical "Soft" eFPGA flow methodology 21

3.1 Purposed Flow 24
3.2 Overview of the Design Flow 26
3.3 VTR phase 27
3.4 Pre-VTR and VTR phase 29
3.5 VTR to Programmable Device Generator phase 31
3.6 Post-VTR phase 32

4.1 Design Compiler Standard Flow 36
4.2 Unacceptable/Accepted Verilog Syntax by ODIN 2 tool 37
4.3 VTR CAD flow 39
4.4 Direct connection and Complete connectivity 43

5.1 CLB Block Diagram 56
5.2 Created multiplexers from wires on a switch box 59
5.3 Configuration Flip-Flop Chain 60
5.4 Figure of module created using modular approach 61
5.5 Comparison between different Programmable device modules 65
5.6 Synthesizable Programmable Device verification flow 67

6.1 Proposed Design Flow 70
6.2 ASIC Flow produced 72
6.3 Feedback loop in routing infrastructure 73
6.4 Feedback loop in CLBs 73
6.5 False Path to be create 74
6.6 Comparison between ASIC area and generated number of Configuration cells . . 79
6.7 Programmable Device and FPGA Area comparison to ASIC 83
6.8 FPGA Area comparison to the created Programmable Device 84
6.9 Diagram with re-programmability tests 85

D.1 Example Work Directory for the proposed Design Flow 109

List of Tables

3.1 Design flow input and output files 25
3.2 VTR inputs and outputs 26

4.1 Tools used by each step 35
4.2 VTR to Programmable Device Generator Inputs and Outputs 47

5.1 Architectural Results using four input LUT 57
5.2 Architectural Results using six input LUT 57
5.3 4-LUT and 6-LUT comparison 58
5.4 Pins used on new architecture 63
5.5 Resulted number of CLBs and tracks/channel depending on architecture 64
5.6 Area comparison between tested architectures 64
5.7 Modules comparison values 66

6.1 CPU time by benchmark 76
6.2 ASIC benchmark Area results 78
6.3 ASIC benchmark Delay results 78
6.4 ASIC benchmark power results 79
6.5 Programmable Device Area results 80
6.6 Programmable Device Delay results 80
6.7 Programmable Device Power results 81
6.8 Programmable Device/ASIC Area Ratio 81
6.9 Programmable Device/ASIC Delay Ratio 82
6.10 Programmable Device/ASIC Power Ratio 82
6.11 Delay comparison between FPGA devices (ns) 83
6.12 Values for Generated Programmable device based on FSM 85
6.13 ASIC FSM Delay and Power values 85
6.14 ASIC FSM Area values 86
6.15 Delays and ratios to ASIC designs 86

Abbreviations

ASIC	Application - Specific Integrated Circuit
BLE	Basic Logic Element
BLIF	Berkeley Logic Interchange Format
CAD	Computed - Aided design
CLB	Complex Logic Block
DC	Design Compiler
eFPGA	embedded Field Programmable Gate Array
EPROM	Erasable Programmable Read-Only Memory
FPGA	Field Programmable Gate Array
GTECH	General Technology
HDL	Hardware description language
IC	Integrated Circuits
I/O	Input/Output
IP	Intellectual property
ISE	Integrated Synthesis Environment
LSB	Less Significant Bit
LUT	Look-Up Table
MCNC	Microelectronics Center of North Carolina
MSB	Most Significant Bit
NRE	Non-Recurring Engineering
PLD	Programmable Logic Devices
RAM	Random Access Memory
RB	Routing Block
RTL	Register Transfer Level
SB	Switch Block
SDC	Synopsys Design Constraints
SoC	System-on-a-Chip
SRAM	Static Random Access Memories
TSMC	Taiwan Semiconductor Manufacturing Company
ULB	Universal Logic Block
VLSI	Very Large Scale Integration
VPR	Versatile Place and Routing
VTR	Verilog to Routing
XML	eXtensible Markup Language

Chapter 1

Introduction

The available density and complexity on Integrated Circuits (IC) have been increasing, following the improvement of technologies to design and fabric ICs, marked by the emerging of System-on-a-Chip (SoC) as a platform for a design methodology. SoCs are comprised of blocks that can be implemented using custom made cells or some specific blocks that can be bought from other companies, like a third-party code, and integrated on a SoC project in hand.

Those blocks were already designed, verified, and optimized, and can be included without many problems, often called intellectual property (IP) cores. IP cores are divided into two categories: "*hard*" IPs, blocks that are already placed-and-routed, or "*soft*", where the logic core is synthesized using standard library cells.

The approach described can lead to improvements in some higher, system-level issues, but a careful handling is needed to interconnect all the available IP in a fully working chip.

As one SoC can be built using multiple numbers of different IPs, made by many different IP developer companies, interoperability problems can arise, even if all features are implemented with care, leading to errors, and deem a chip unsuitable.

1.1 Motivation

The semiconductor industry has been experiencing a challenging evolution in the complexity of digital ICs on SoC design and extensive validation before fabrication is needed to ensure design correctness of the produced circuit.

At the same time, with the current demand for faster turn-around development cycles, major interoperability tests are already performed in actual silicon. The clear inconvenience of the approach is the absence of capability to perform small updates if a need arises, due to interoperability, for a change in functionality at the IC.

One of the first testing ground for interconnectivity and interoperability is done resorting to testing chips. These chips are built using individually created blocks and often act as prototyping devices or testing kit for the product the IP developers sell so that it can be connected to other IPs or chips.

The development of a SoC for prototyping leads to substantial costs, as a unique mask set is needed to be produced to be used in the elaboration of the prototype. So, intense testing is required on new created IPs. Nonetheless, corner cases can be discovered after tapeout due to lab debug or continuous verification. A new design is undertaken to resolve the problem, and a new prototype must be fabricated to test the new chip. As a new mask set is needed, the production cost rises.

IP is often produced using design flow that creates a logic that if produced is fixed, meaning that cannot be changed after production. This design flow allows fast productions but brings the discussed problems.

However, if the IP could be produced in programmable logic and unexpected errors appears during testing, developers can debug and correct it on the field, which prevents the chip disposal. One of the examples is the new wave of fast Field Programmable Gate Arrays (FPGAs) that are used to test some IP code before going into mask production, although there is always a small portion of the design that due to the specific nature and operating frequency requires implementation in actual silicon to enable prototyping.

Some semiconductor manufacturers are trying to combine processors with re-configurable logic, or integrate FPGAs as soft or hard macros in the chip design, and directly fabricated into the SoC silicon [1, 2].

However, not only a dedicated FPGA development team is needed to help to develop and integrate a design like this, as there is also need to buy from FPGA vendors devices that are unoptimized for the needed implementation.

Therefore, there is a need to create an implementation that empowers an IP designer with programmability functionality. Allowing the creation of a fine-tuned application specific programmable device, using standard tools, and flows specially formed to be deployed on an IP developer normal work, preventing a need to have specialized personnel overseeing that task.

IPs are often produced using a fixed logic core implementation approach, that is used to create circuits that implement necessary functionality using the least amount of area lowering production costs. However, on replacing a fixed logic implementation to a programmable implementation, a significant increase in chip size could be observed, as more logic would be added so that programmability is included. Nonetheless, recent years' technology enhancement has been enabling space shrinking on produced devices, allowing the production of larger devices, so that programmable device could also be added without size penalty, which allows a shift in the designer focus, from creating a small optimized device to develop a larger but modifiable chip. In summary, this added programmability can lower costs and production time, as there is no need to go through a new production cycle.

1.2 Objectives

The main objective of this dissertation, as proposed by Synopsys, is to study a normal workflow of an application specific integrated circuit (ASIC) developer and from the standard ASIC workflow, a new one must be developed. This flow shall enable the ASIC designer to create, from the same

Hardware language (HDL) file that he would use to build the ASIC device, an application specific programmable device.

There are many programmable devices design flows available in the market, like FPGA flows, so a proper study shall be needed to create an optimized design flow optimized for an ASIC Register-transfer level (RTL) developer. Not only studying the ASIC design flow but also available programmable design flows.

Developing a design flow like this brings some challenges, as the commonly available programmable devices are often built using different approaches from an ASIC approach. One of the main difference comes that an ASIC RTL developer uses a limited sub-set of technology standard cells, that can be used in many projects and technologies. However, the programmable devices use a more customizable approach was a developer create his circuits in a full-custom implementation. This strategy is inadequate, and cannot be incorporated in an automatic design flow normally used by an ASIC RTL developer.

The creation of this design flow will also have to face decisions to the architecture that will be produced. As using an automatic process of production will hinder the usage of specific circuits often employed in programmable devices.

However, the produce programmable device will serve to a specific application. Hence only the inputs and outputs that are available in the original representation of the circuit will be offered in the programmable device. Nonetheless, the created design flow will allow the designer to add additional inputs that serve no purpose in the original application but can be used in later applications.

As in standard FPGA, one of the principal problems is frequency requirements, the target for a future production shall be in the control logic, that has less stringent frequency requirement that a datapath logic.

Summarizing, if the ASIC developer would initially create a fixed logic implementation like in 1.1, this dissertation developed design flow can be used to generate a larger core, but with additional re-configurability functionalities.

This dissertation produced design flow shall be able to:

- Create an optimized programmable device, using the same HDL that is employed in a standard ASIC flow;
- Allow that an already produced programmable device suffer small modifications if the designer needs.

The work needs to focus on integrating the created design flow into the standard work of the IC developer, minimizing the involvement, through RTL code, of the ASIC developer.

To test and implement the design flow one example programmable architecture needs to be used; this needs to have the ability:

- Support the same function as the same circuit implemented in an ASIC fixed logic device;
- Allow small alterations to the implemented function;

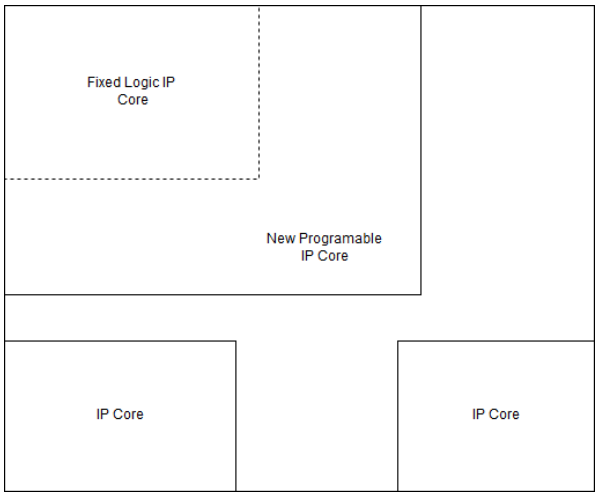

Figure 1.1: Example of the created device

- Being synthesizable in an ASIC standard cell approach.

So, this dissertation main goals are:

- Define, create, and implement a design flow that allows the ASIC developer to create a reconfigurable block without modifications to the original ASIC RTL.
- Study configurable device, which can be used in a standard cells design that shall act as a replacement for the already implemented, ASIC fixed logic core.
- Explore the impacts from an ASIC to programmable device implementation, using the example programmable architecture.

1.3 Thesis Organization

To fulfill the objectives exposed in 1.2 the present report shall be organized in the following way:

- **Chapter** 2 – Corresponding to the State of the Art for the present dissertation, presenting the two most important topics that need addressing. First, developed re-configurable architectures: presenting inherent characteristics. Second, the study and overview of the ASIC, and FPGA design flow that are studied exploring, to conclude in the study of the design flow that merge these two flows, often used to produce embedded FPGA (eFPGA) devices.
- **Chapter** 3 – The design flow that was developed during this dissertation is presented in this Chapter, along with the developed phases and challenges faced in its creation.
- **Chapter** 4 – The created design flow uses many tools, this chapter will be given an insight and explains the study performed concluding with these tools applicability in the produced flow. This Chapter is mainly used to characterize some of the most important tools used in

this flow: the Verilog to Routing (VTR) and the C++ tool created in this dissertation: **VTR to Programmable Device** generator.

- **Chapter** 5 – So that the design flow can be used the programmable architectures must be presented and further studied. Detailing some internal architecture characteristics that need to be explored, along with some internal routing infrastructures that were explored. This Chapter concludes with full description for the produced architecture.

- **Chapter** 6 – This chapter will conclude the study of the design flow, explaining its integration into the standard ASIC RTL workflow. Some challenges were faced, not only because of the design flow but also because of the used programmable architecture. Done this some results can be obtained, where two main points of comparison are explored: comparison between the programmable device to an **ASIC fixed logic** implementation and to a "soft" and "hard" **eFPGA implementation**. The final study performed in this Chapter will detail the main functionality of the performed circuit: re-configurability.

- **Chapter** 7 – In this Chapter, the principal conclusions will be discussed using the results collected in the preceding chapter. The main solution of replacing an ASIC fixed logic circuit by a programmable device, exploring advantages and disadvantages. Finally, some future developments are detailed.

Chapter 2

Literature Review and Background Work

Nowadays many digital and analog technologies, like embedded memories and microprocessors, can be combined into a single chip using circuit blocks. Those circuit blocks are called intellectual property (IP) cores and were already pre-design and pre-verified, that can be obtained from internal sources or third-parties. Using this methodology, a single chip can include many different components like memory blocks, embedded processors, among others [3]. This design methodology is called a System-On-a-Chip (SoC).

By using a SoC approach, a designer can focus only on high-level issues, as there is no need to have the knowledge of the implementation of an IP but only its interface. Higher communication speed, reduction in packaging and test costs and added system reliability, are some advantages that are added to a design using this approach.

However, integrating different IPs into a single chip can result in some unexpected problems arising from signal interoperability, generated when there is a need to interconnect different IPs between ports using different protocols or control signals. Additional logic, often called glue logic circuits, can be added to rearrange signal protocol between those IP interfaces, but that logic can fail if some specifications are not correctly implemented, for example when some frequencies are wrongly specified.

Intense testing is done before fabrication, but some small corner cases can be found, and no matter how careful a SoC designer is, there will be chips not meeting its functional demands. The disposal of the chip with an error will result in the need to create another device, which leads to an increase in production time and prices, something that most IP companies cannot afford.

This problem arises from the design flow that is used by IP companies, circuits are produced as an application specific integrated circuit (ASIC) that is a small and optimized circuit for a particular application, although inplemented in a fixed logic architecture, that does not allow modifications.

Some new solutions were already proposed to resolve this problem, like incorporate logic cores into SoCs [3], which can be added before fabrication allowing future modifications.

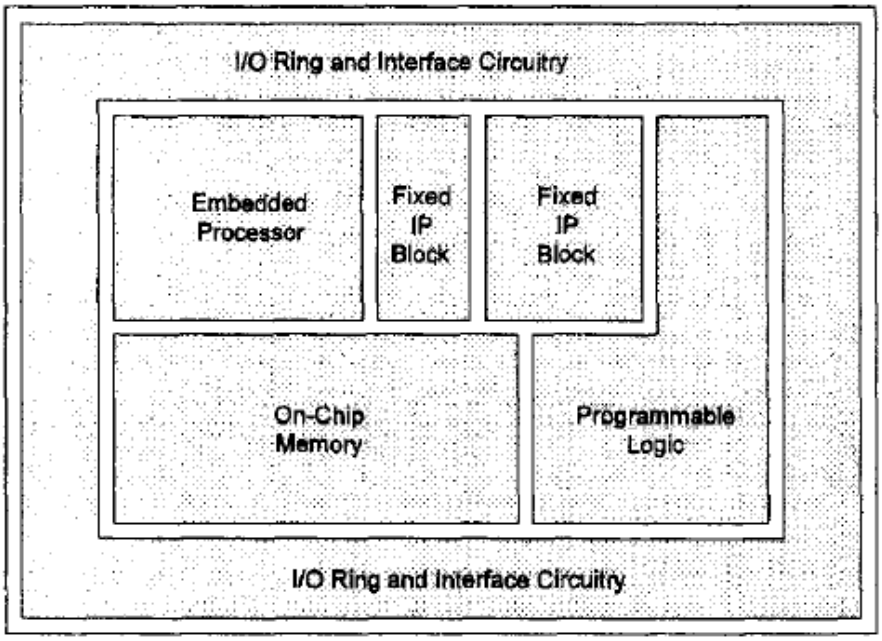

Figure 2.1: Example of a SoC design (source: [3])

By adding re-configurable logic to designs, a provided reconfigurability is added to the overall system that can be used for post-fabrication modifications. Digital logic programmability allows the quick implementation of a new set of different logic functions after circuit fabrication that can include upgrades and design errors correction.

Fast and inexpensive modifications translate into a lower Non-Recurring Engineering (NRE) costs as there is no need to wait for the new silicon to be manufactured and tested. When there is no programmable logic in the system, a modification can add a significant delay and risk to the design cycle resulting in costly design iterations.

The market window is a major topic for IP development companies and if a system includes programmable logic, the production time can be reduced as corrections and updates can be implemented in the programmable logic after silicon production.

To create programmable logic some companies used to add to their construction some external programmable devices. However, by using outer reconfigurable logic core, some disadvantages can be found, related to the fact that there are only some commercially available programmable chips of specific sizes that lead to an overall waste of money, resources, and space.

Nowadays embedded programmable logic core close the gap between external chips, which provide programmable logic like FPGA and fixed logic devices like ASICs, providing excellent performance values, resulting in reduced project costs and development times.

Nonetheless, 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 con-

tinuous technology processes advance, for all areas of circuit design: power consumption, delay, and area requirements.

So, to explain previous work on the topic of this dissertation, this chapter will be divided into the following subjects:

- An overall review to some chip architectures, either non-programmable, like ASICs explored in section 2.1, and programmable ones in 2.2, leading to the overview of the most used programmable device the FPGA, explored in section 2.3.
- The design tasks followed by a chip developer to produce the above ASIC and programmable devices, allowing program and reprogram chips with those re-configurability capabilities, explain in section 2.4.

2.1 Application-Specific Integrated Circuits (ASICs)

The first topic on creating IP and SoC is to explain the various approaches available to create a device to the developer.

One of the most used devices are ASICs that correspond to non-standard integrated chips that are customized for a particular usage, however, due to high production costs; an ASIC usually is a targeted to a very high volume production.

The creation of an ASIC device can follow three main strategies: gate array, full-custom, and standard cells [4].

If the transistors or gates are fabricated on a two-dimensional array forming the core circuit, the ASIC was produced using a gate-array topology. The device functionality is implemented by customizing the upper layers that are used to connect the desired nodes in the produced circuit.

A full custom design can also be followed, that offers the highest performance in the smallest silicon area possible. However, the expense comes from the increased design time, complexity, and risk.

Finally, a designer can also use standard optimized blocks, specifically produced to implement known functionality in a Standard Cell approach. Blocks with a specific functionality, like simple Boolean logic functions (NAND, NOR) as well as sequential logic, can be grouped together to form the need complex digital function to implement.

Although a full custom manual layout can result in a higher performance and area efficiency, a largely manual effort from the designers is needed. Therefore, most designs are often performed in a standard cell approach as most libraries can deliver the performances required for all types of applications.

Most common ASICs devices are often implemented using standard cells, as this is one of the most flexible strategies available. Using typically available libraries, a designer can quickly implement a digital core and target it for high performance compromising the produced device between area, delay, and power.

On a standard cell approach a library containing a list cells implementing logic functions such as inverters, AND and OR gates, multiplexers, and others, must be used. These cells are implemented using blocks of transistors connected to form small cells. The interconnection between these cells is made using numerous layers of metal over the locations where they are placed to implement the need logic functions. Standard cells libraries are commonly available from a wide number of providers in all available technology nodes. These companies provide and support all needed data and information to be used on a large number of tools that run on a digital design flow. When using an approach like this the created ASIC core is implemented as a fixed logic circuit. A full fixed digital logic design does not allow a designer to re-program the digital core to operate with a different function, and the need to make a simple update can lead to the creation of another chip.

As it will be explained in this document, to develop an ASIC, there is a need to use a computer-assisted design (CAD) system which is an integrated suite of software facilities for design entry, functional simulation, physical layout, test simulation and design verification.

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, would not allow the speed and performance that an ASIC design can provide, but at the expense of not being re-programmable.

2.2 Introduction to Programmable Logic Devices (PLD)

An electronic component with re-programmability and reconfigurability features is called Programmable Logic Devices (PLD). A configurable logic and an array of linked modules that allow bit storage structure can enable the user to configure a PLD to perform needed functionalities.

Compared with a fixed logic device, the possibility of enabling programming and re-programming often has many advantages, like, reducing the costs for low volume applications. Designing for PLD requires less time, as the device is often already produced and available to the developer. Hence NRE costs reduction and shorter time-to-market, together with risk mitigation thanks to the availability, in some cases, of changes on-the-field. On the other hand, a PLD pays regarding performances if compared to an ASIC, with considerable area overhead and speed reduction, that leads to its usage in low volume markets.

The development of this devices is often the common point between PLDs and ASIC designs. The need of dedicated software environment used in the process of developing, simulating and testing, the circuit to be mapped to a device is, in general, similar between this approaches, as it will be explored in 2.4 section.

2.2.1 Programmable Logic applications

The usage of a programmable logic core can be used in many different applications and scenarios, given that this silicon die can perform many functionalities. However, on a fixed logic device, a new development cycle is needed for every new product, and there is a need to have numerous

of prototype, tests, and debug before a stable implementation is ready to be sell. By using a programmable logic an efficient way to reduce the number of prototypes is found. However, many other applications can be seen:

Standards and requirements change

Nowadays, a very competitive market pressures that a product is released as soon as possible, even before some needed features are already implemented. In an ASIC, all features are necessary to close a product, hence a significant impact could be seen in the quality of the produced ASIC. Programmable logic allows that only some parts are implemented, as some updates can be performed during later stages of a design cycle, or even at the buyer.

Improvements and corrections

One major concern to the industry is the ability to make small modifications without having to go through a complete silicon re-spin that require a new manufacturing masks. Using programmable device updates can be done locally, and there is no need to change the chip for a new upgraded one.

On-chip Testing

A more demanding system testing is necessary with the increase in circuit integration, packaging technologies, and circuits technology. As many devices have integrated testing devices, a full integration on the global circuit can become unsuitable, and maintain easy testability without losing any of the testability each block would support. Programmable logic can be used to generate stimulus for each part of the chips and to control this testing device. Also, programmable logic can be used to change or improve the on-chip tests to reflect changes or improvements, without a need to create another fixed logic device. The designer can devise new tests when required and implement them on an already fabricated silicon die.

2.3 Field Programmable Gate Array (FPGAs)

FPGAs are the most common used programmable device chips and is used mainly lies in its architecture, which includes the programmable core used to implement needed functionality, and they are programmable interconnect which commonly a circuit consisting of arrays of a logic block with electrically programmable interconnections.

A design using FPGA approach, rather than custom technologies, can have two main advantages: NRE cost, and faster time-to-market [5].

Some compelling advantages must be considered over fixed logic technologies, like the months of production and the cost needed to obtain a first ASIC device. On the other end, an FPGA configuration takes only seconds and in the case of a mistake, the device can be reconfigured, reducing cost and time of production.

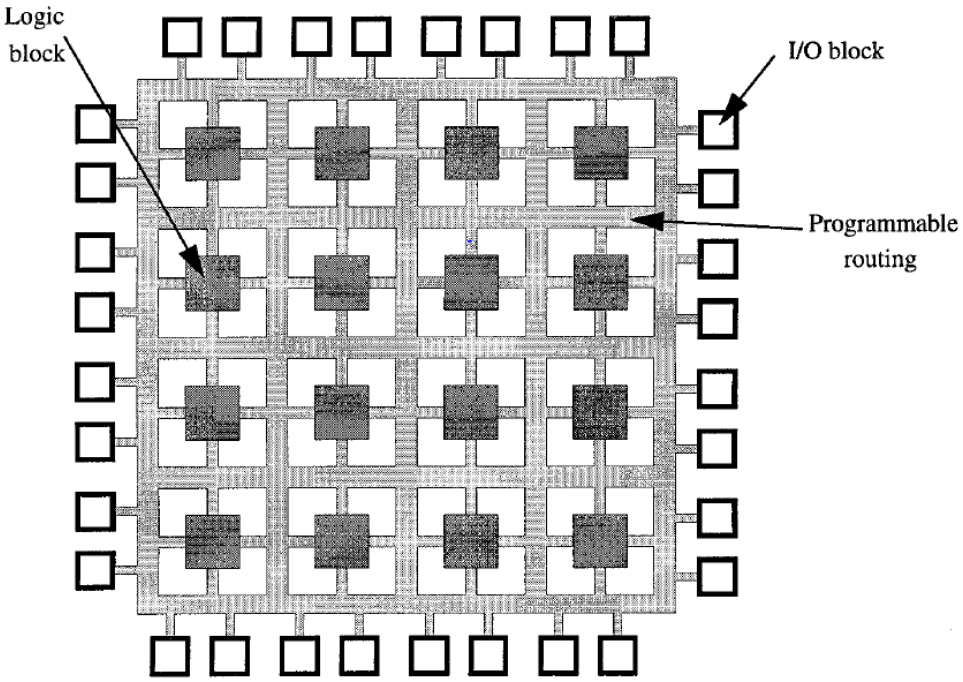

Figure 2.2: FPGA architecture (source: [7])

One of the major disadvantages of using an FPGA lies the interconnectivity fabric as programmable switches are used. In contrary, a standard cell-based circuit is interconnected with metal wires. Interconnectivity relied on switches poses a problem, as switches have a higher resistance than metal wires adding significant capacitance, reducing the maximum able frequency on an FPGA. FPGA switches also take more space than metal wires, increasing the overall circuit size compared to an ASIC implementation. As compelling as an FPGA solution may seem, an FPGA solution sees an area increase of around 40 times, plus four times decrease on circuit speed, and ten times more power consumption [6]. This values shall be carefully analyzed by a designer so that the best device is chosen for the intended application.

2.3.1 FPGA Architecture

An FPGA can be devised in various components: I/O blocks, programmable routing and logic blocks as shown in figure 2.2 [7].

A circuit is implemented on FPGA by dividing it into small portions and programming each portion on individual logic blocks. Interconnection must be assured so that logic blocks are correctly connected logic blocks. Finally, I/O blocks on the device need to be either set as an input or output. These are three main points that must be addressed by an FPGA architecture:

Logic Block

A logic block is implemented using a programmable core, storing values corresponding to a small portion of the circuit to implement. A more detailed overview of logic blocks is discussed in section 2.3.2.

Routing

The routing is what defines the main FPGA architecture and can be divided in at least three major FPGA architectures available: Symmetrical array, hierarchical, row-base.

The most shared and straightforward architecture used is a symmetrical array commonly known as island-style architecture, as seen in the example 2.2. As the island-style architecture is a rectangular and uniform circuit, more satisfactory results regarding area and speed are produced, as most circuits tend to have routing demands which are evenly spread across a chip [5].

The programmable routing must ensure that all the generated and applied signals travel to the desired IO block or logic block. Hence three main points must be explored on a routing architecture.

- Wire length: that is related to the total number of logic blocks that a single wire passes.
- Switch blocks: corresponding to the interconnection device that routes signals between a vertical and horizontal line, several switch block were already studied, Universal [8], Disjoint [9] and Wilton [10] with different characteristics and different performances.
- Internal population: that characterizes the ratio of connections from the main wire to the logic block.

Programming technologies

FPGA switches need some circuits that can save programming bits, often FPGA uses Static Random Access Memories (SRAM), these circuits give the designer the store the bits that ensure the correct signal routing. Many other types of cells and implementations like anti-fuse, Erasable Programmable Read-Only Memory (EPROM), can be used.

2.3.2 FPGA Logic Blocks

The Logic Block is the block responsible for implementing the gate-level functionality required. One of the elements used to accomplish this is a circuit that holds the truth table for the logic function and routes the function result to the logic block output. This circuit is known as the Look-Up Table (LUT).

LUT is the common programmable core presented in FPGA devices, however many other programmable cores have been developed over the years, this work studied one of the simplest circuits found:

Universal Logic Block (ULB)

Forslund created a module with three input that could implement any function [11] using NAND circuits. The study is concerned with the techniques used to create three-variable universal logic blocks from NAND circuits emphasizing on the minimization of NAND blocks.

The study assured that it is possible to create 2^m functions from n input variables (m = 2^n). To a three-input variable 256 functions can be generated. Many of those functions are redundant therefore 80 equivalent classes remain. Equivalent classes are characterized as a set of functions obtained from a particular connection of circuit elements by only manipulating the application of variables to the input of the network. Also by enabling that the input variable to be complemented and if both true and complement are available at the output, the number can be reduced to 14. Also, 4 of those are degenerate functions and may be obtained from one or more of the other, non-degenerate, ten classes by biasing the appropriate input or inputs. As so, one may obtain 256 functions of three variables by implementing only ten logic circuits, manipulating a circuit with the conditions said before.

This type of universal logic blocks (ULB) as shown in figure 2.3 is generated when the three inputs restriction is lifted and by interconnecting or manipulating the application of those inputs.

To implement a function using this module, there is a need to see what circuit, from the ten, three input circuits, must be used to perform a specific function. Knowing the circuit to use, the interconnectivity of inputs and outputs the ULB must have to implement the needed circuit is also found. Finally, the proper inputs must be connected to the correct pins.

As an example, the article explains how to implement the following expression:

$$F = AB + AC + BC + \bar{A}\bar{B}\bar{C} \tag{2.1}$$

The article explains that the above expression can be implemented using the fifth equivalent function with the following inputs: X = $\bar{A}$, Y = $\bar{B}$, Z = $\bar{C}$ and by using the complementary output of the ULB. To implement the fifth equivalent and implement the expression the circuit 2.3 must have the following connections:

- Pin 1 has an input of $\bar{A}$,
- Pin 2 has an input of $\bar{B}$,
- Pin 3 has an input of $\bar{A}$,
- Pin 4 has an input of $\bar{C}$,
- Pin 5 is connected to Pin 7 ,
- Pin 6 has an input of $\bar{B}$.

Some more studies were performed to this type of circuits like [12], that resulted in the creation of other more complex blocks, based on this one. These new modules are larger and did not include

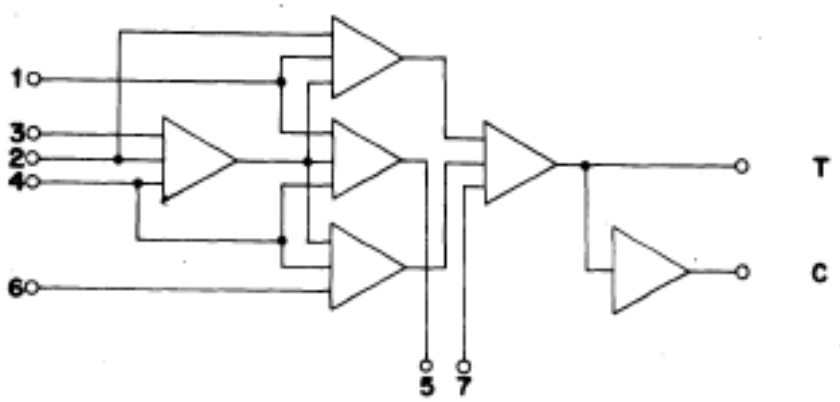

Figure 2.3: Minimum NAND ULB (source: [11])

any IO tree structure so that the created block could be used in normal operations. Also, for this reason, neither synthesis algorithm or tool is available to help with the creation of a design flow using this kind of cores.

So, a simple conclusion can be taken to the lack of studies on these blocks, as there is a need to create a programmable routing structure that can be configurable to implement an intended functionality (like a complex crossbar). So, a simpler core was needed, like Look-up Tables that are going to be discussed below.

Look-up Table (LUT)

One of the most used programmable chips, the Field Programmable Gate Array (FPGA), rely on Lookup Tables (LUT) as the main programmable logic device, as those devices can implement combinational logic using Flip-Flops and multiplexers. LUTs can be seen as a truth table in a combinational logic context.

One of the main issues when using LUTs is the effect of its size and cluster size, which is the number of LUTs per cluster, on the speed and logic density of an FPGA. LUT size is an important issue and addressed in many works [13, 14], and the authors agree that the main criterion to follow when designing an LUT is an area-delay reduction, and the best overall results are produced when using LUTs with sizes (number of inputs) of 4 to 6. If there is a need to implement a higher size LUT, a cluster can be created as seen in figure 2.4.

As it was discussed in section 2.3.1, FPGAs include LUTs as the programmable core used to implement logic functions, they are the main part of a basic logic element (BLE). A logic block typically is implemented using SRAM cells or variants of these that serve as 1-bit memory cells to store the programmable digital logic function on the LUT.

To implement an N-input function on an N-input LUT 2^N SRAM cells or memory cells are needed. To implement any Boolean combinational, function a 2^N input multiplexer must be used.

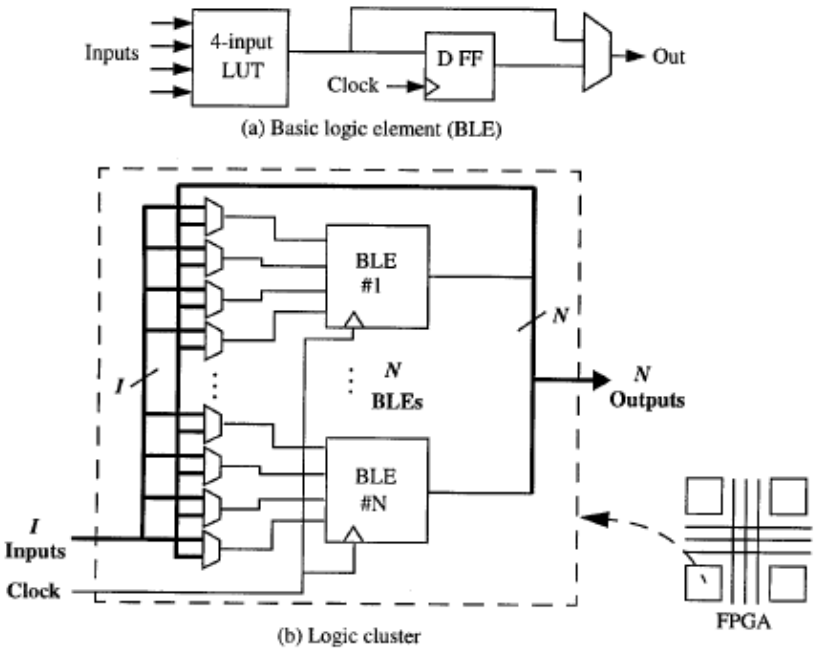

Figure 2.4: FPGA basic logic element and logic cluster (source: [7])

The figure 2.4 presents an example of a BLE using LUT, and at the output, a Flip-Flop can be found to sample the combinational output and generate its sequential output. Also, a multiplexer can be used to map either the output of the sequential or the combinational output.

2.3.3 Embedded FPGA

Embedded FPGA (eFPGA) is one of the main solutions when a designer wishes to include a programmable device in SoC designs. A designer must first choose what should be the type of eFPGA to include, as it is divided into two main categories: Soft and Hard FPGAs [15].

A simple solution to create an eFPGA is to remove the I/O pads from an FPGA and include the inside logic as a whole new IP. In a solution like this, the layout is already predefined, so many considerations already made by the FPGA designer cannot be changed, this characteristic is common in a hard eFPGA IP.

A distinct advantage of this approach is that a designer can buy the core and without any knowledge of internal features of the IP, include it in a SoC without concerns to the internal IP functioning. The clear limitation to a hard eFPGA approach is that the blocks bought have already predefined areas that can lead to wasted area in a SoC.

Another method to implement an eFPGA in own IP is to license an eFPGA fabric suited for the needs of the project and implement it using an ASIC or customized design flow; often characterized as a soft eFPGA IPs. Some studies were made about this approach [16]: A programmable

core architecture described in a behavioral Register-Transfer Level (RTL) code that can be implemented together with the rest of the digital cores through a standard digital design flow. This solution brings an increase in size, of around seven times of using a hard FPGA solution, due to the usage of a standard cell. However, by using some specialized cells, this ratio can be lowered to almost 1.7 or 3.1 times the size of a hard FPGA [15], this being a Firm eFPGA approach.

Some IP companies are selling eFPGA solution as soft and hard cores to be implemented to own designs, like Menta [17] and Nanoxplore [18]. Also, some authors introduced a way to embed eFPGA as a flexible soft core to meet the demands of SoCs with non-rectangular open-space [19].

In addition to the eFPGA, the latest SoC devices embed silicon-based dedicated processors and other macros, allowing hardware and software trade-offs that can be used by the designer so that performance is met by the design [20, 21].

2.4 Computer-aided design (CAD) Flows

A designer that wishes to produce a circuit has at his disposal many tools that can help him in this task. This Computer-aided design (CAD) tools are often found in design flows guiding the design process into creating the needed IP or chip.

An IP designer can consider two main available design flows:

1. Convert an RTL description to an ASIC circuit – ASIC flow in section 2.4.1;
2. Convert an RTL description to a bitstream that can be uploaded to a programmable device 2.4.2.

So in this section, these two flows will be discussed, concluding with the discussion of a flow that will bind these two to create an eFPGA type of device in 2.4.3.

2.4.1 ASIC Design Flow

As explained in 2.1, an ASIC can be built in many different kinds of flow: full-custom, gate-array, and standard cell.

One of the most used common strategies is the Standard Cell, that by using an RTL with a description of the chip, a technology library containing standard cells, and various tools a chip can be produced in a short amount of time that would take on a full-custom design.

An ASIC designer, in a standard cell approach, still needs to undertake a set of tasks, a design flow. One common example of a design flow can be found in figure 2.5 from [22].

An ASIC design flow is divided into two main steps: logical design step, that consists on the logical synthesize of the RTL, and the creation of the chip that is called the physical design step.

However, in overall 6 step designs compose an ASIC design flow:

1. Design entry: a designer will create the needed entry for the circuit, either by a hardware description language (HDL) or by a schematic entry for a standard cell design or full-custom respectively.

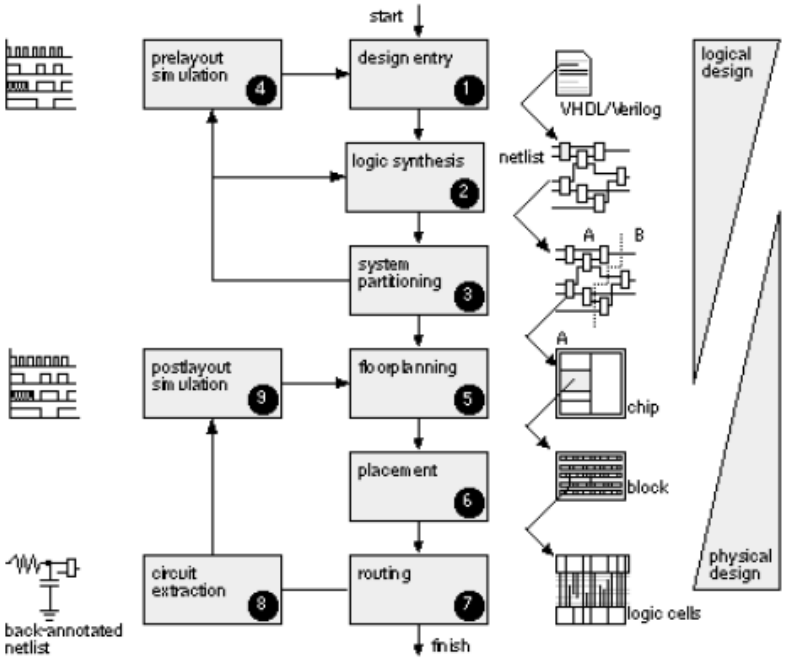

Figure 2.5: ASIC design flow (source: [22])

2. Synthesis: After the HDL is synthesized a synthesis tool like Synopsys Design Compiler, creates a netlist with a description of the logic cells and their connections;

3. System partitioning: The netlist that resulted from the previous step can then be divided into ASIC-sized pieces if the design large. After those steps a designer needs to check if the synthesis work as intended using testbench simulations in a post-layout simulation:

4. Floorplanning: Finally, after validation, the designer must design the physical part, by first going to the floorplanning that will arrange the netlists in a chip;

5. Placement: Then the placement is the step in each the location of block in the chip is decided;

6. Routing: And finally, the routing that creates the needed interconnections in the chip.

After this, the developed chip is ready to be produced, by providing the resulted tapout to a foundry.

2.4.2 FPGA Design Flow

Designing for an FPGA is divided into two main flows, one that resembles an ASIC flow, and an FPGA specify flow. Most developers usually only develop applications to FPGA using one of those flows.

The first design flow is used to produce the FPGA itself: using a full-custom design approach. The development and creation of FPGAs is a time-consuming and resource-demanding task as they are typically implemented using a full custom layout flow to achieve the best possible performances.

Another part of the FPGA design flow is the programming, where the designer must design and create the circuit to be implemented. There is no true consensus for a generic design flow to programming an FPGA, as every vendor sells a different design kit with different flows and can introduce different cell libraries that can be unique. For example, to develop for a Xilinx Spartan-6 [23] a designer should use the ISE Design Suite [24] tool. Using these tools designing for FPGA can become an easy task.

However, most problems can be experienced during FPGA creation, as a full-custom approach is time-consuming, and testing different architectures and capabilities can be unfeasible using this strategy.

To this, an open source design flow was created to help the creation of a placed and routed FPGA design, the Verilog to Routing (VTR) project [25, 26]. VTR is an open-source framework used by FPGA investigators and CAD researchers to conducting and development research in this field. Using this flow, studying multiple FPGA architectures can become an easier process as by only taking a Verilog description and a description file including the internal architecture of the target FPGA architecture, a designer can use to gather multiple statistics of a produced architecture.

This flow performs multiple operations like elaboration and synthesis using ODIN II [27], then logic optimization and technology mapping using ABC [28] and finally packing, placement, routing and timing analysis using VPR [29]. In the end, VTR not only generates the files with the place and route information but also generate FPGA speed and area results. The tool provides many examples so that developers can study the feasibility of the tool.

One of the major concerns with VPR is the architecture design of the FPGA. To represent an FPGA an XML description is used by VTR flow [30], that includes every FPGA component and connection. The creation of this file can be made handmade, by an FPGA designer, however, automatic solutions were also developed to help in the creation of many architectures automatically [31].

VTR is a highly valuable tool to study FPGA architecture. However, this dissertation found a special interest in a specific VTR functionality: that allows the creating an optimized FPGA architecture parameterized in the number of CLBs and tracks/channel dependent on the circuit will be implemented. This feature will be one of the most explored flow characteristics during this dissertation, that will eventually lead to the creation of a design flow based in this toolset.

2.4.3 eFPGA Design Flow

As explained in 2.3.3, eFPGA are devices that try to give an IP developer the ability to include a programmable device in own IPs.

The design flow to create this eFPGA depend on the type of eFPGA: soft, firm, or hard.

A designer that wants to use a hard eFPGA must buy from a developer that uses a full-custom design methodology to build the device, hence the creating of standard FPGA or a hard eFPGA are almost the same task.

However, some designer sometimes wants to include eFPGA in designs without having to bother with the layout, using the full potentials of an ASIC flow in this task. To this, the developer can buy some RTL modules containing soft eFPGA, which can be synthesized along with the project.

Many works have been trying to automatize the layout generation, resulting in the reduction of the intensive and manual work used to implement the programmable logic circuits under a full custom layout design flow.

One example is the Totem Project [32] whose objective is to reduce the design time and effort in the creation of reconfigurable domain specific architectures, those are optimized for a set of applications and constraints, resulting in smaller logic and better performance than general-purpose reconfigurable cores. The work showed that is possible to use standard cells to automate the programmable logic layout generation and that the savings gained increase if the application domain is narrowed.

The GILES project [33] generates a transistor-level schematic automatically from a high-level architectural specification of an FPGA. A cell-level netlist can also be produced and placed and routed automatically. Finally, to create the FPGA array the tiles can be grouped together.

These above tools have been the base of many soft eFPGA works, and have cited by many other authors on eFPGA layout works: In [34] a completely automatic method for the generation of an FPGA using an open-source VLSI tool-kit, was presented. The paper concluded that open-source tools could be used to create an FPGA without using a time consuming full-custom design. Leading to the creation of different FPGA with various parameters.

A layout generator based on manually optimized FPGA tiles has been developed in [35]. The authors say the tool is more practical than GILES project as it adopts the manually created tile layout for a good performance with the optimized area, providing flexibility on the FPGA size and helps to reduce the manual labor in the processing eFPGA cores.

Reference [36] the paper presents the ZUMA open FPGA overlay architecture, which consists of an open-source, cross-compatible eFPGA architecture that is intended to overlay on top of an existing FPGA, in essence, an "FPGA-on-an-FPGA." On the paper, an FPGA with an overhead of 40x was created while still maintaining ability and mapping efficiency for user designs.

In [37] a design flow and benefits of using a tile-based layout approach in soft eFPGAs and imposing structure on the final layout is presented. The usage of a structured physical design and optimized standard cells improved area and delay. The design flow followed in the paper is presented in figure 2.6 which is based on the Cadence hierarchical design flow for SoC Encounter. The flow starts with the architecture exploration in the FPGA CAD flow to determine the parameter that is best for the gate-level synthesis of the eFPGA fabric.

The eFPGA created presents an island-based FPGA architecture used because of it highly-regular, tile-based structure that is well-suited to hierarchical physical design.

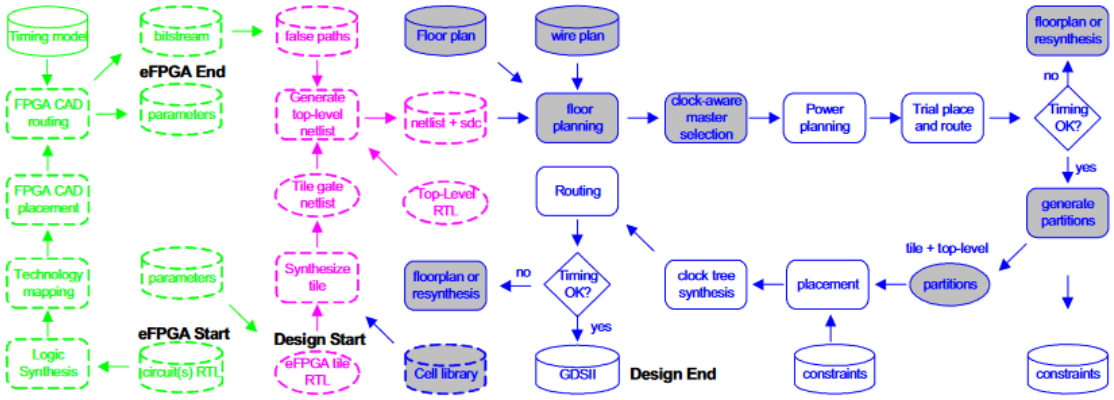

Figure 2.6: A typical "Soft" eFPGA flow methodology (source: [37])

After gate-level synthesis, the physical design is presented. The first step is floor-planning, followed by power planning, and pin placement. Cell placement is followed as well as clock tree planning and synthesis for the top-level design. Finally, all the selected tile masters and the top-level design are routed using a standard ASIC router.

With the approaches described in this paper, the soft++ approach to eFPGA design using the ASIC flow becomes a viable alternative to hard eFPGA fabrics. To generate some results two technologies were used, 90nm and 180nm, using some application circuits. The area saved by using the approach was compared with the same implementation, but on a hard FPGA, for example, a 0.3 ratio between the areas of an I^2C master module under the soft approach presented and a hard solution was given. Finally, the article showed that the area consumptions were not intolerable for the technology at the time (90nm) and that the solution could be used to create programmable logic fabric in SoC design by the industry.

However, one of the most valuable tools and studies performed in this dissertation was to the already introduced VTR flow. That will be used to create a representation of a programmable device parameterizable to the input application specific RTL file. As found in [38], VTR saves in internal memory this programmable device representation. That can be extracted so that a programmable circuit can be synthesized. Nonetheless, not only this flow generates the representation, but also this flow will perform some needed functionalities like technology mapping and place and route.

2.5 Summary

Even with the amount of verification coverage to an IP or SoC, design errors, wrong simulations, or verifications, can make products not work as intended.

Embedding programmable logic cores on these products can be used to solve mistakes, without needing to create an additional product, leading to a reduction in production costs.

Many studies have been explored explaining how a programmable logic cores could be used by designers and users to integrate their IP with other macros easily on a chip.

The first topic is the comparison between a normal ASIC implementation and a programmable device implementation. This chapter explored the basic features of one of the most commonly used programmable devices, the FPGA. Some designers are trying to implement this type of devices in SoC projects. To this, eFPGA designs were also studied in this chapter, following the two main design methodologies: soft and hard.

This work mainly focused on the soft eFPGA methodology, where an eFPGA can be created using a normal ASIC design.

Previous works have already demonstrated on the possibility of using eFPGA cores in SoC, by including or replacing some fixed logic devices with re-configurable logic macro, using many different implementation methodologies. One of those being the integration of soft cores in a standard cell design approach, that is documented together with its advantages.

The study of those works was ideal for producing the design flow proposed by this dissertation goals: one example being the VTR flow can be used to generate a representation of a minimal size programmable device.

Chapter 3

Programmable Device Generation Design Flow

A standard cell methodology is a design flow that allows designers to develop ASIC, using a library of basic cells. This method brings clear advantages to the design process, allowing that the designer can focus on the functional aspects leaving the optimization to an automated process.

However, a disadvantage of using a standard cell approach is that the developed ASIC will be implemented using basic cells that lack reconfigurability features, therefore after production, the ASIC functionality cannot change.

If a designer desires to implement his design in a programmable device, he can do so using FPGAs, which are specific circuits built specially to serve as re-programmable chips. However, these devices are often used or in specific applications, or as pre-production testing ground.

One of the solutions to transform an ASIC fixed logic design to programmable devices is to buy eFPGA devices to act as replacements. However, sellers usually sell architecture that may not be ideal for a application as some parameters like number of LUTs are often predefined by the seller. While this approach can be seen fit for developing a device capable of implementing a broad range of application, it is not ideal for the case of an ASIC developer. Therefore, a need arises to create a design flow, which could be included in a normal workflow of an ASIC designer, giving him the ability to produce a re-programmable device without many designer interventions.

So, this dissertation following main objectives will be materialized in the purposed design flow:

- Creation of a programmable circuit that is optimized for the needed application;
- Easy integration of the design flow into the standard workflow of an ASIC designer.

One of the main objectives for this dissertation is that design flow be implemented into the standard workflow of an ASIC designer as is displayed in figure 3.1.

This design flow was created in this way due to decisions that arose during the dissertation development; therefore, this chapter addresses the four main decisions that were taken and developed during this work.

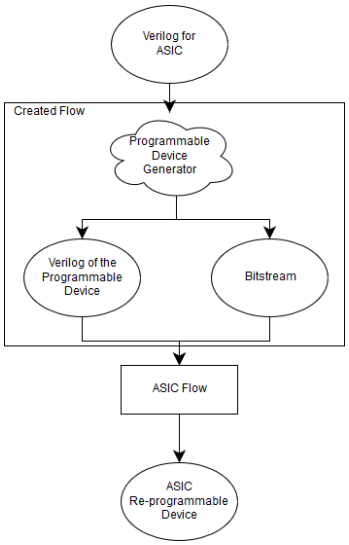

Figure 3.1: Purposed Flow

Through this chapter, the design flow developed during for this dissertation will be explained. This flow is like an eFPGA design flow, however not only will create a bitstream to implement the application, but the designer will also have at his disposal the RTL for the programmable device.

The device shall be optimized for the original circuit, and the created RTL description can still go through any original ASIC design flow so that a chip containing the programmable device can be generated.

3.1 Global Considerations

Before detailing every step that was at the base of this flow creation, there is a need to have a full understanding of the created design flow; primarily to the inputs needed, outputs generated, and tools used by the flow. Moreover, the design flow objective must be clear:

- Creation of a programmable device from a Verilog with the original intended application to be implemented on a programmable device and the bitstream needed to implement the application.
- Production of a bitstream containing some small changes to an already implemented and fixed programmable device.

Therefore, the first step is to describe what are the necessary inputs and outputs created for this design flow, where the result can be found in the table 3.1.

Another important step in the creation of the design flow is choosing the specific tools to be used. Some chosen tools are third party ones, like Design Compiler, along with some tools created to be used in this flow, like the VTR to Programmable Device Generator. In this chapter, all those

Table 3.1: Design flow input and output files

Inputs	Outputs
Circuit netlist file	Netlist with programmable device
General XML FPGA description file	Bitstream to be uploaded
Device size *	SDC file
Number of track/channel *	Programmable Device information (f.e size)
IO pins location *	
(* Optional inputs to create bitstream for a previously produced programmable device)	

programs will be introduced, so that in the following chapter most specific details are given to every tool used.

The fully complete developed design flow is essentially divided into four major steps. All the steps, decisions and the main inputs/outputs for each step can be found in figure 3.2.

3.2 Phase 1: VTR

One of the first decisions was choosing a tool that could be used to create the first representation of a programmable device. This tool should have a specific way to create a re-programmable circuit that was optimized for a specific application. After some research, Verilog to Routing (VTR) flow was the chosen solution.

As was introduced in 2.4.3, VTR is a design flow that allows an FPGA designer to easily study many FPGA architectures without producing then using the long, full design process. A designer, describing only benchmark circuits in Verilog format, and using a specific XML architecture description file with the internal FPGA architecture to be studied, can have a is disposal reports and information that can be helpful in the designing process. For example, this dissertation will use the number of generated number of CLBs and the number of tracks/channel that was created to describe some devices.

To generate this reports the VTR [25, 26] will use VPR program to create the representation of the programmable device architecture, and then perform the necessary tests upon it.

As this dissertation does not aim at producing a general-purpose device, the size of the generated circuit can be minimally sized to that that is required to implement the original intended application.

VPR when performing placement will check the minimal number of logical resources (CLBs) that will need to implement the application. When performing routing, by establishing paths connecting logic blocks, not only VPR will try to minimize delay but also the number of tracks that the device could need. These two VPR aspects are critical for the intended design flow that can lead to the creation of a minimal size programmable architecture.

Therefore, the first step in this phase was to check the performance and usability of this toolset. One of the first points that were studied relates to the inputs needed to run the framework, and the outputs generated (see table 3.2).

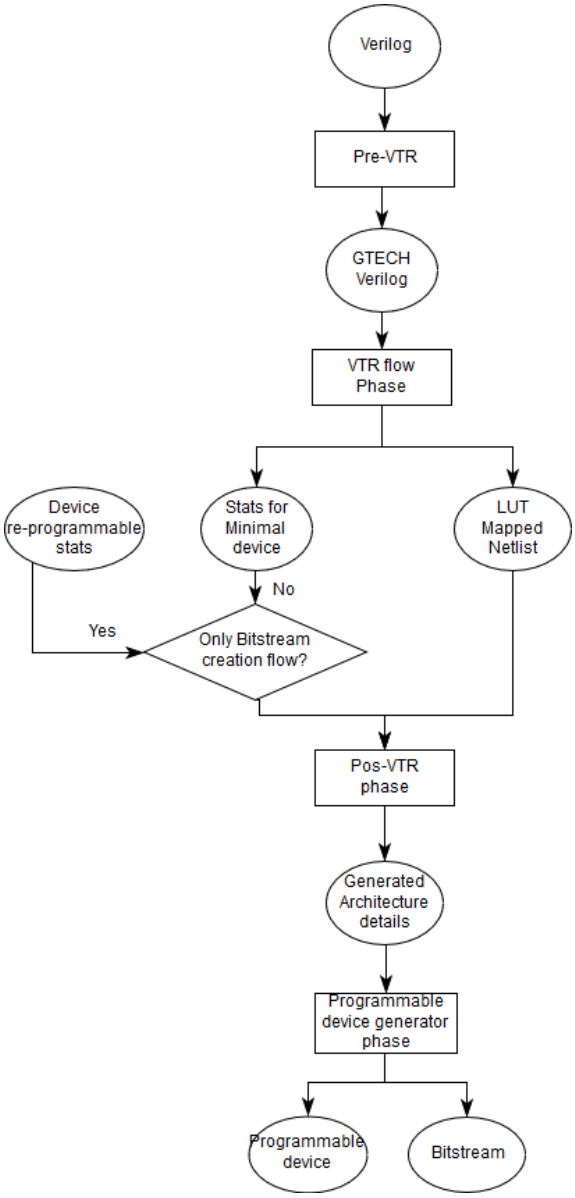

Figure 3.2: Overview of the Design Flow

Table 3.2: VTR inputs and outputs

Inputs	Outputs
Verilog HDL file	Place and Route Reports
XML architecture description FILE	FPGA internal architecture
	Programmable Device specifications (*f.e.* Size)

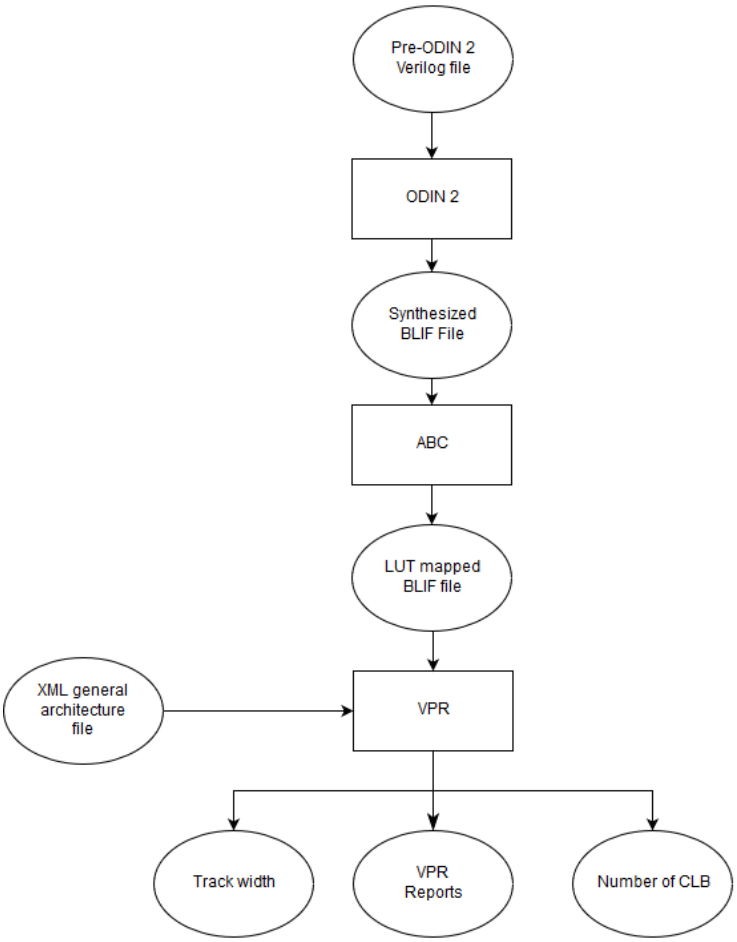

Figure 3.3: VTR phase

VTR also uses the following tools, each one with the functionality described:

- Odin 2: Synthesis and elaboration tool;
- ABC: LUT mapping tool;
- VPR: Packing, Placement, Routing tool.

Afterward, this flow was then integrated like displayed in figure 3.3, the block diagram can be found, with all the above steps and necessary inputs/outputs.

Some VTR examples were then used to check for the validity of the tool. As VTR was successfully employed and the reports were successfully generated, it inspection allowed to formulate the next approach for this design flow. Also, the point introduced above was proven: VPR could create a minimal size device to implement the original circuit, reporting on the minimal number of CLBs and tracks/channel necessary.

This testing was performed using examples delivered in the VTR distribution, already created to work in this flow. However, when using other Verilog examples, the elaboration step conducted by Odin 2 failed.

3.3 Phase 2: Pre-VTR

One of the first steps in every design flow is the synthesis phase. For example, in a standard ASIC design flow, Design Compiler can be the tool used to create a post-synthesis or gate level netlist.

In VTR flow, Odin 2 was chosen by the VTR developers as the synthesis tool, with the function of setting up a gate-level netlist. As Odin 2 is an open-source tool, many limitations and lack of optimization can be found in the tool [27].

Using this solution, not only is the input design synthesized using a much powerful tool than Odin 2 but also, some problems, with Verilog parsing could be resolved. Some of the Verilog synthesis problems found will be further explored in Chapter 6.

Nonetheless, the use of Design Compiler uniquely did not make a full pre-processing phase, as some errors still occurred. As normal, when using the output flatten Verilog netlist in Odin 2, it would fail, as the netlist would only include instances unknown to Odin 2. The first attempt was to include the full Verilog module description for the GTECH library in the Odin 2, but still, the design could not be parsed, failing because the top module did not instantiate all the appended modules. Accordingly, a small Python script was created so that it could produce a new Verilog file that will only have the declarations of the modules instantiated by the top one.

This Python script receives the information of the modules used by Design Compiler plus the two Verilog files: the flatten netlist and the Verilog library. It generates as output a Verilog file with the original top module including the declaration of the used modules.

This pre-VTR step could mean that the Odin 2 could be completely removed from the flow. Nevertheless, Odin 2 does not only serve the function of synthesizing the circuit but also serves to convert the Verilog file to a format accepted by the other VTR tools. This format corresponds to the Berkeley Logic Interchange Format (BLIF) netlist, and unfortunately, Design Compiler could not create the output in this format. So, Odin 2 was not removed from the flow but was kept only to serve as a Verilog to BLIF converter program. Figure 3.4 display how this step was interconnected.

3.4 Phase 3: VTR to Programmable Device Generator

After having information about the minimal size device all the created information can be used to generate the programmable device. For this purpose, a C++ program was developed: **VTR to Programmable Device Generator**.

The outputs that are generated correspond to the ones displayed in table 3.1. Everything excluding the structural information is exclusively produced by this tool. For the inputs, this program uses most of the generated information by VTR.

The program proceeds in four main steps:

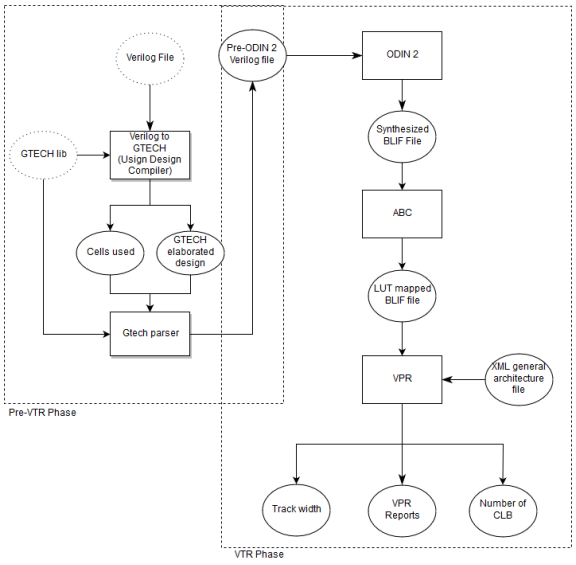

Figure 3.4: Pre-VTR and VTR phase

1. Reading and parsing of input files;
2. Creation of the internal representation of the generated programmable device;
3. Placement and routing of the original circuit;
4. Generating output files;

The first step has two core functions, first to be a fail-safe if some file is not available, and afterward to check if the input files have the necessary information. For example, the place input file includes a header line where the name of the circuit and architecture are displayed. This header is checked to see if the file was correctly inserted.

Then the program proceeds to create an internal representation of the programmable device as was generated by VPR. VTR to Programmable Device Generator program was developed using the same internal data structure created for VPR so that the generated reports could be easily parsed. Easing the parsing process by establishing the same internal structure, like in VPR, the following steps to perform this program were simplified, as all reports use the same internal notation.

So that the program can generate a programmable device, two main architectural details must be created:

1. Complex Logic Block (CLB) architecture;
2. Routing Architecture.

Regarding the CLB architecture, this design flow focuses on the creation of a basic CLB design that includes only one four input Look-Up Table (4-LUT) and a Flip-Flop at the output.

For the routing, the VTR to Programmable Device Generator program will use an output from VPR, the rr_graph.echo file that has the representation of the routing structure, previously created

by VPR. This two parsing steps carefully create the architecture was as generated by the VPR tool so that now placement and routing could be performed.

Regarding placement, the main file used is the mapped BLIF file, produced by ABC, and the place report, produced by VPR.

The placement is the process where the functions mapped in the BLIF file are placed into a specific CLB in the created programmable device. The developed program will then check every function on the BLIF file and crosscheck the place file. As this file includes the position of every placed CLB in the generated device, the program can now save the mapped function in a specific CLB, and afterward using the same mapped BLIF file, generate and save that CLB bitstream.

Then routing should be performed, and for this, the program will use the route output file from VPR that includes the routing for every net in the netlist file (the netlist file corresponds to the net output also generated by VPR). Every net will have the followed path in this file, from its source to sink. So, to develop the routing, this program will follow every net path and saved the corresponding route. As it will be seen in Chapter 5 the routing will be implemented using multiplexers, so the program will use this information so that it can choose what shall be the selected input for every multiplexer on the programmable device, so that the route file can be perfectly implemented.

Finally, the following outputs are generated:

- Verilog file with the RTL representation of the programmable device;
- Bitstream for the original fixed logic circuit;
- Additional SDC file to be used in the synthesis or timing analysis programs (it will be further explained in Chapter 6).

A block diagram of this step can be seen in figure 3.5, where the inputs and outputs for this step are also displayed.

Using the design flow until this step, every time a designer inputs a Verilog file, a new programmable device is generated. So, there is a need to create a way to give re-reconfigurability functionality to the design flow. A new step has been set up to provide this feature to the designer.

3.5 Phase 4: Reconfigurability addition

The main objective of the design flow is to produce a programmable synthesizable device. There is also the aim of providing a designer the ability to re-program this device.

Using the flow until this phase, only architecture that has the minimum size to implement the original circuit could be produced, independently of the needs of the designer. So, an additional step was created that allowed the designer to create a larger than the minimum size device if necessary. This new phase also gives the ability to create an additional bitstream to an already produced device, by using as a new input the architecture specifications created previously so that placement and routing are performed on an already developed device.

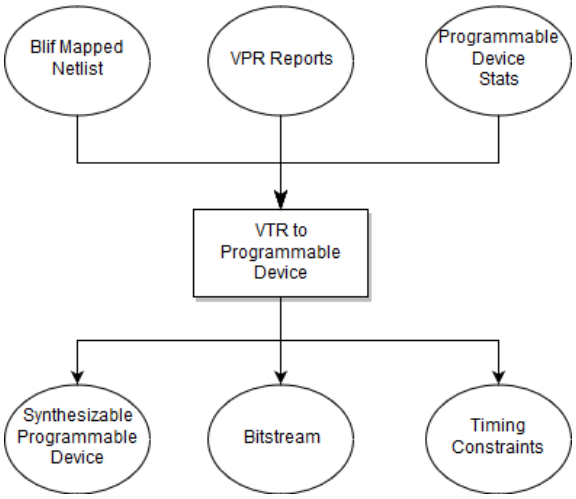

Figure 3.5: VTR to Programmable Device Generator phase

As can be seen in figure 3.6, in a normal run, the flow is the same until the start of the VPR program. In this step, the design will perform differently in the two options available – creating a new programmable device or create a new bitstream for an already created and implemented circuit.

In the case the designer wants to the create a programmable device, VPR is run without size values so that it can generate the minimum values for a programmable device.

Then an additional step was added, where the number of CLBs and tracks/channel can be used to produce the device. Using this extra step, the designer can enlarge the circuit by demand.

By using VPR again, the placement and routing are executed one second time and the additional size can be added. In a case that the designer only needs a new bitstream for the VPR is run only once using the designer inputs, performing placed and routed of a new original circuit but to an already developed programmable device. This run was the probability to fail if:

1. There are not enough CLB to implement the request function;

2. Routing is impossible with the number of tracks/channel provided.

Finally, there was a need to develop another Python script that could extract the exact position of the inputs and outputs generated by VPR. This information must be used when a designer wants to create a new bitstream, as the IO location for VPR must be maintained during other iterations.

After having a programmable circuit, the designer should now save the three different information reported in this flow: the size and number of tracks/channel, and the IO location.

3.6 Conclusions

This design flow was built with the objective of creating a programmable device flow for an ASIC developer that can be easily integrated with the unmodified ASIC design flow.

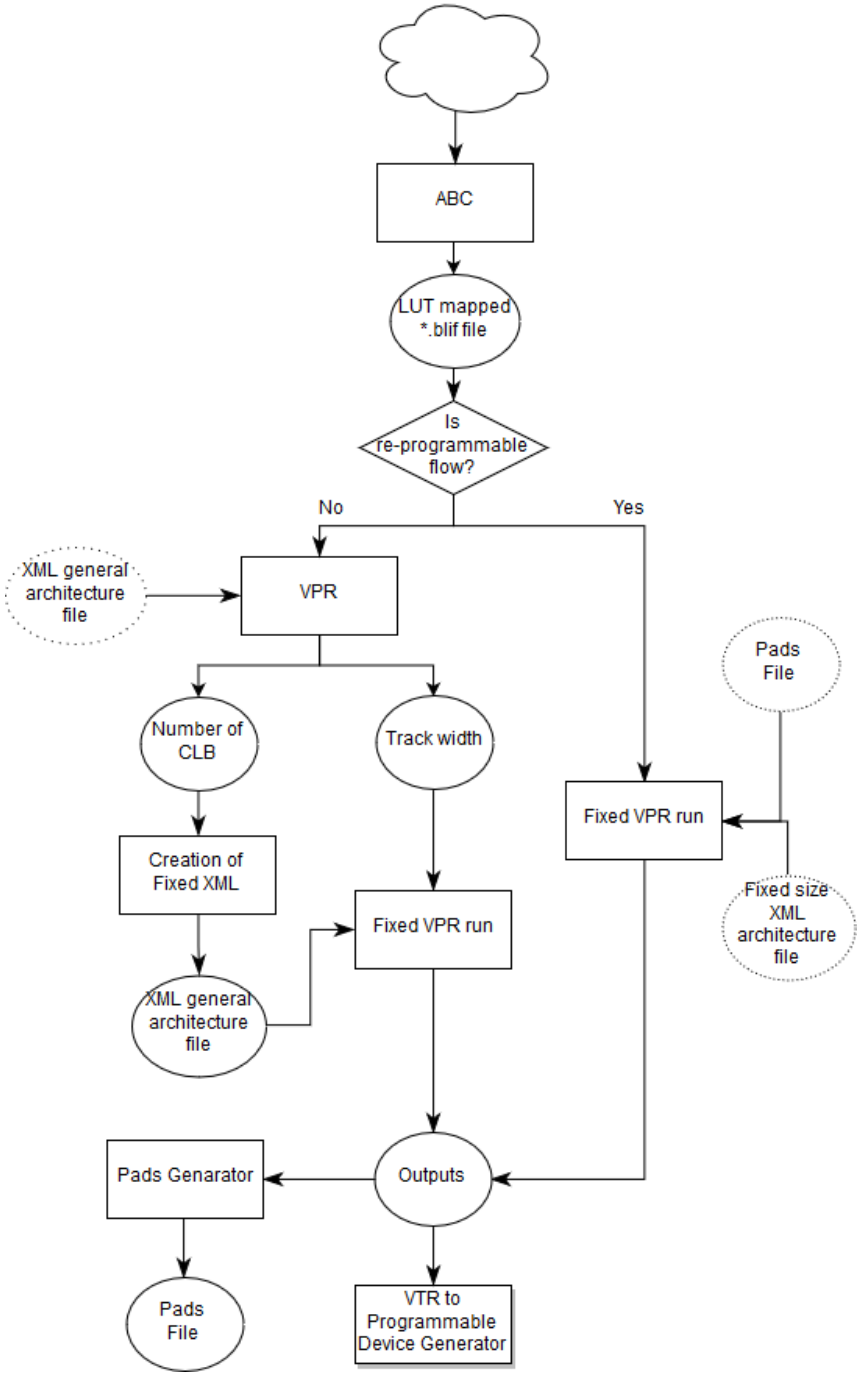

Figure 3.6: Post-VTR phase

The design flow starts by using only the VTR flow, which creates a programmable device optimized for the Verilog benchmark given. The VTR flow was introduced, mainly emphasizing the required inputs and the outputs generated and also the tools that were used in this flow. Some problems were found when testing multiple and different Verilog files. Because of the lack of optimization found of was then included in the design flow, Design Compiler, that elaborates and flattens the design before Odin 2.

With this alteration, VTR flow could run without problems, independently of the RTL, the designer just had to ensure that Design Compiler could synthesize the design. Having this, a C++ tool, VTR to Programmable Device Generator, was created to parse the VTR outputs and create a Verilog description of a programmable device and the bitstream to implement the original circuit.

The only necessary step to this design flow was to ensure that the designer could re-program an already produced design. So, a final step was created, where VPR is run to the place and route a specific architecture.

After this, the designer can synthesize and create the programmable device produced by this flow and use the bitstream to program the device.

In the next chapter, much more detail about the tools can be found, along with a discussion of the challenges faced in creating/using these programs.

Chapter 4

Description of Tools used in the Design Flow

The previous chapter introduced the design flow created. This flow uses many tools that will be carefully studied so that the generated design flow could be optimized for the purpose intended: the creation of a re-programmable device.

The design flow is divided into four steps divided by the executed programs in each phase. For example, the VTR phase, as the name suggests, primarily uses the VTR flow to create the programmable device representation. In table 4.1 the tools employed in each step are shown (at bold is represented the programs created during this work).

One of the most important reasons to choose a tool for this Design Flow is the versatility and customization available in each tool. Customization was necessary as the developed design flow has a different purpose these tools can accomplish individually. Giving as example Design Compiler (DC), it can use a TCL script so that a designer can adjust the program to perform automated tasks. Moreover, in this flow, as will be explained in section 4.1, a TCL script was created to run DC, to synthesize the original user circuit to a flatten netlist only containing basic cells.

VTR is a tool often used to study FPGA CAD algorithms and architectures, which is usually a challenging process as high-quality experiments are difficult. However, this tool will not be used for the goal that it was designed: in section 4.2 can be seen that not only VTR can produce a high number of hypothetical architecture as it also can report what the minimum size programmable device that can implement the user circuit.

Table 4.1: Tools used by each step

Phase	Tools
Pre-VTR	Design Compiler & **GTECH Python parser**
VTR	VTR flow (ODIN 2, ABC, VPR)
Post-VTR	VPR & **Pads Creator (Python script)**
Device generator	**VTR to Programmable Device Generator**

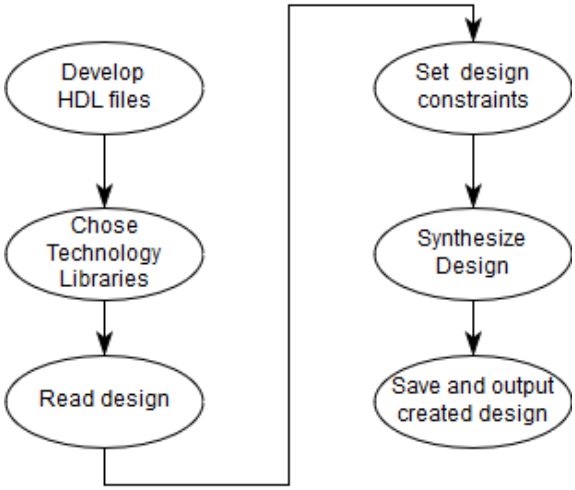

Figure 4.1: Design Compiler Standard Flow

In this new design flow, some new tools also had to be developed. Not only some minor scripts that help with each step but also a full application created was designed from scratch. In 4.3, the creation of the VTR to Programmable device generator program, which was developed to assist in the creation of the programmable device description and the configuration bitstream, is explained.

4.1 Design Compiler

The first tool used in this design flow was the DC synthesis tool, this tool is often used to create a gate-level netlist from an RTL representation of a circuit. This gate-level netlist is often created for a Technology node in each it will be implemented.

A designer that uses DC to synthesize typically follows a predefined workflow. The general flow that designer uses can be seen in the figure 4.1

In this work, DC is used to create a first synthesized representation of the circuit. As DC can be automated using a TCL script, one was developed to run DC. The TCL script makes DC follow a flow like the one in figure 4.1.

The difference between a regular flow and one created is referring to the synthesis process. DC will usually perform synthesis to a predefined library technology, but first, it will synthesis to an included General Independent Technology (GTECH), mapping the circuit to components like:

- AND, OR, NOR;
- Flip-flops;
- Multi-level logic gates: AND-Not, And-or, among others complex single gates.

A GTECH library was used that only included basic modules like ANDs and ORs, and Flip-Flops.

Wrong Syntax

```
FF FF_A ( .Q(ff_out), .Qn(), .D(ff_in), .CLK (ff_clk) );
```

Correct Syntax

```
FF FF_B ( .Q(ff_out), .Qn(unconnected_wire), .D(ff_in),
    .CLK (ff_clk) );
```

Figure 4.2: Unacceptable/Accepted Verilog Syntax by ODIN 2 tool

DC should synthesize the initial design for this GTECH library, but the output generated needs to follow some requirements before it can be used in the VTR flow:

1. Allow the insertion of additional input pins, which are not originally used;
2. Resolve problems related to cell instantiation;
3. Append cell definition to Verilog file.

The first issue should be solved to allow a designer to choose if he wants additional inputs pins in the programmable device. This inputs pins are additional pins that a designer might want, that could be added for additional functionality. This design flow allows that the programmable device contains additional inputs that do not serve any purpose in the original implementation but do this, inputs that are added to the Verilog description should not be removed during synthesis.

Synthesis process in DC will reports warning "LINT-28", but as it is only a warning DC will not eliminate this input, leaving it unconnected in the synthesized circuit. DC does not remove unconnected inputs to allow the production of some designs that some pins, that serve no internal purpose, however, need to be included for compatibility reason [39].

This requirement was already solved in this step. The VPR tool will also remove unconnected inputs: more information on this is found in 4.2.4.

The second issue required a more detailed study. Usually, the instantiation of the cells is created using the left representation in figure 4.2, both this syntax is legal in common synthesis tools, like DC. However, ODIN 2 did not accept the left representation. DC is a very customizable tool, as the synthesis process can be change by setting/unsetting internal variables. To meet this requirement the synthesis was changed so that the outputs could be accepted by ODIN 2. So, the variable `verilogout_show_unconnected_pins` was set to true that made DC create a undriven net `SYNOPSYS_UNCONNECTED` for every unconnected input from a cell. This problem was mainly seen in Flip-Flop cells, as the library that was used included Flip-Flops that had not

only a normal Q output but also an inverted one, and most of the time the synthesis did not use one of those.

Finally, the last problem was a more challenging one to resolve. DC when synthesizing a design creates the output file in Verilog format including instances of the cells declared in the library. As normal, the next tool does not have the library, so does not know the definition (internal circuitry) of the cell instantiated. The first solution that was proposed was to append all cell definition to the file that would become the ODIN 2 input file. However, for reasons that will be explored in subsection 4.2.2 the file still could not be parsed by ODIN 2 tool.

To solve this problem, a Verilog file was created including only the definitions for the cells that were used by the synthesis process and not the full library. Therefore, a Python script was developed to create the correct input file for ODIN 2 (more details in following subsection 4.1.1). To help in this process the DC command `report_hierarchy` was used, that reported the cells employed in the synthesis process.

4.1.1 GTECH Python Parser

This script was created to solve the problem seen when trying to append the cells definition to the Verilog file to be used by ODIN 2.

The script will check the output of the `report_hierarchy` command to check what were the use cells in the synthesis process. An example of the output of this command can be seen in the listing below.

Listing 4.1: Generated report from DC

```
GTECH_AND2 GTECH
GTECH_FD2 GTECH
GTECH_MUX2 GTECH
GTECH_NAND2 GTECH
GTECH_NOR2 GTECH
GTECH_NOT GTECH
GTECH_OR2 GTECH
GTECH_XOR2 GTECH
GTECH_XOR3 GTECH
```

After this, the script will create the output Verilog file with the main circuit synthesized and will append the necessary cell definition to this file. This script ensures that only the cells used are in the new Verilog file.

4.2 VTR flow

Verilog To Routing (VTR) design flow is an open-source flow commonly used from FPGA and CAD research as an easy and quick way to study FPGA internal architecture, without having to

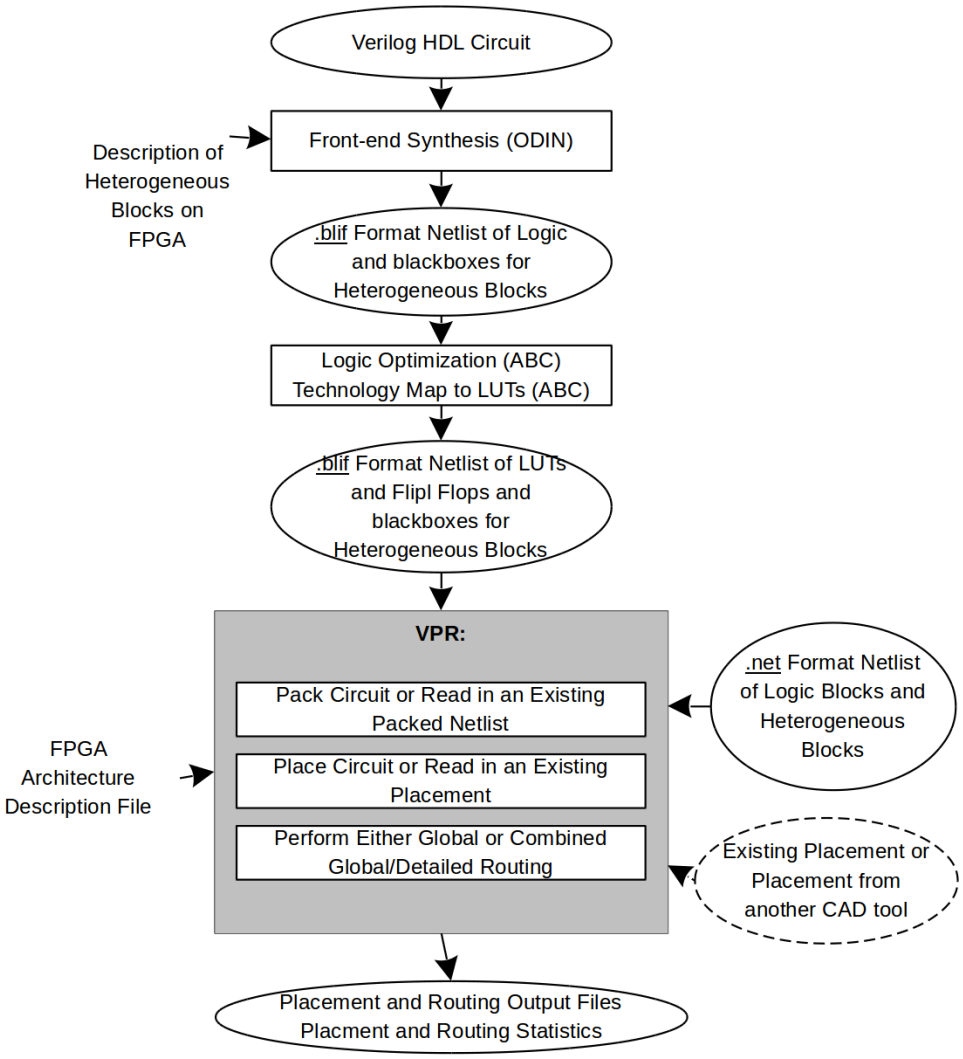

Figure 4.3: VTR CAD flow (source: [26])

design or even produce a full FPGA prototype, which is a hard and time-consuming task.

To have a clear understanding of the flow, figure 4.3 displays a block diagram of the flow, along with the inputs/outputs used and the following tools, ODIN 2, ABC and VTR.

This section will detail the full flow, principally explaining the functionality of each of the tools that are employed. Among this, one of the most important files, the XML hardware description file will be detailed, along with the study that was performed to create the XML used in this work.

4.2.1 FPGA Architecture Description Input File

Before detailing each of the tools and functionality, there is a need to explain the inputs of the flow.

The first input is a Verilog circuit, like the one that was created by DC. However, VTR will also need to know what shall be the internal architecture of the FPGA device to be used. An XML

file can be provided to the flow detailing the necessary FPGA internal construction details. This format is flexible enough to represent a significant number of FPGA designs, both hypothetical or commercial-based ones.

In this subsection, a brief introduction to all tags that can be included in this file is presented, but only the most significant ones for this will be fully explained. For more information, the documentation for VTR includes all the option/tags that can be used [25]. Also in Appendix A an example XML file employed in this work can be found.

Every architecture file needs the following top level tags:

- `<models>`
- `<layout>`
- `<device>`
- `<switchlist>`
- `<segmentlist>`
- `<directlist>`
- `<complexblocklist>`

Models

The **model** tag is used to include modules not usually found on FPGAs. Normally a designer not only creates an FPGA using LUTs but will also include some more specific modules. For example, some FPGAs also contain RAM blocks or multipliers.

Special build modules were not included in this work, given that the pre-VTR phase will synthesize the input Verilog file to a representation of only basic cells (AND, OR). Therefore, this tag was not used in this work.

Layout

Using **layout**, the designer can control how the FPGA is sized: the size is automatically created depending on resources needed, or the user chooses the size of the device. The representation of this tag can be found below (Curly brackets represent an option with many alternatives, each separated by horizontal line):

```
<layout {auto="float" | width="int" height="int"}/>
```

If the auto attribute is used, VPR will generate a minimum dimension device that could fit the benchmark circuit. To determine this number, VPR checks the mapped file for the number of blocks and IOs needed, creating the minimum size device that can hold all the necessary logic. To create a fixed logic FPGA, the designer can use the width and height attribute that corresponded to the number of the block in the horizontal and vertical direction, respectively.

Regarding the design flow produced for this dissertation, one of the used XML files needed to include the "`auto`" attribute so that VPR can first create a minimum size device. However, in later stages of the design flow as explained in 3.5, where a designer wants to produce only the bitstream, in the fixed architecture, this tag is used with the width and height options.

Device

The **device** tag includes some sizing and timing options that can be used to better optimize the FPGA reports after Placement and Routing. In this work, the values that were used corresponded with the ones taken from example XML files. Along with this, the switch box type can be defined in this tag, and this dissertation also uses the most common switch block found, the Wilton type (that was already explained in 2.3.1).

switchlist and segmentlist

These are used to specify the type of switches and wires that are used in the FPGA device. As the above tag, the content of this tag was equal to most of the example architecture files.

directlist

One of the demanding features in FPGAs is arithmetic calculation; this calculation can be implemented in standard CLBs. However, by using specially placed adders more efficient calculation can be performed. These adders are placed around the device, and as this calculation often needs a long calculation chain, and if routed through the global routing structure, they can become the critical path for these devices. So, many FPGAs include some specialized carry logic for these specific calculations. As this is standard practice on FPGAs, VTR toolset also tries to support the inclusion of "carry chain" structures [26].

To include "carry chains" some steps are needed to be followed: add the adder module definition on the above `<models>` tag, that directs ODIN 2 to synthesize adder calculation in this adder module and not in some logic equation that would be placed in standard LUTs. VPR will also identify this blocks and check for the existence the `<complexblocklist>` tag (more detail on this tag below) to the existence of this blocks. This blocks inputs and outputs will be identified by VPR to be connected using a "carry chain", and by using this tag **directlist** a designer can determine the connectivity for this IOs. More information on this processes in [26].

This work did not try to create any "carry chains". However, knowing that VPR had the ability to create direct inter-CLB connections, this functionality was tested so that normal signals could be routed through this direct link and not trough the global routing, that could be used to reduce the number of tracks and create more accurate (and optimized) connections between CLBs.

Nonetheless, this tag only implemented to be used as the "carry chains" definition, so the use of direct connections employing this tag did not work, and standard connections still would go through the global routing architecture. As this thought feature did not work, neither adders are included in the produced architecture this tag was not employed in this work.

complexblocklist

This tag is used to describe some complex logic blocks that can be found on FPGAs. To create a block, the child `<pb_type>` tag should be included as many times that is needed to represent clusters and primitives.

A top level `<pb_type>` is a cluster that can include more children `<pb_type>` that can also be clusters or primitives. Regarding the primitive tags, the most important attribute was the `"blif_model"` one, that describes the type of block implemented in the mapped BLIF file. For example, to implement a LUT, the `"blif_model"` attribute should have the `.name` value that corresponds to the declaration of a LUT in the mapped BLIF file. All blocks declared using the above models tag can be implemented here.

Only the two commonly used models were implemented: `.names` for LUTs and `.latch` for flip-flops, that is generated automatic by ABC.

Each `<pb_type>` tag should include the following children tags: input, output, and clock. These tags are used to describe the IOs that are available on those modules and some inherent characteristics.

Below are presented two of the most important attributes of this tag:

- `"equivalent"` - this attribute describes if the pins are logically equivalent, meaning that the router phase can swap the order of inputs and is ensured that the functionality of the internal block does not change. As an example, the inputs of an OR gate are logically equivalent, as the order of the inputs does not change the output. However, an adder can create a different result, if the Most Significant Bit (MSB) is swapped by the Less Significant Bit (LSB), the output is different. A LUT depends on the order of the inputs, so if this attribute is true, there is a need to create some internal routing to allow changing the order of inputs to the one that was implemented by the LUT. The produced device for the present work contains CLBs without internal routing, so this attribute must be left false.

- `"is_non_clock_global` - this attribute, if defined, frees the input from using the standard routing interconnection. It is a beneficial attribute for some particular signals like FPGA-wide asynchronous resets. However, some own dedicated routing channel must be available on the device. The presented work produced a device including a wide asynchronous reset signal, so to model this port this attribute should appear in an input tag related to this signal. However, ODIN 2 could not map a specific pin as an asynchronous reset (more detail about asynchronous reset signals in the 4.2.2 subsection). As reset signals could not be added the reset port was removed from the CLB, and this attribute was not used in any port connection.

Along with the IO related tags, the `<pb_type>` tag can include children tags that are used to describe how the complex block pins should integrate with the inter-block devices.

Most important values in these tags are the `default_in_val` and `default_out_val` that indicates the default `Fc` value, that is the number of tracks that a port will connect to a routing

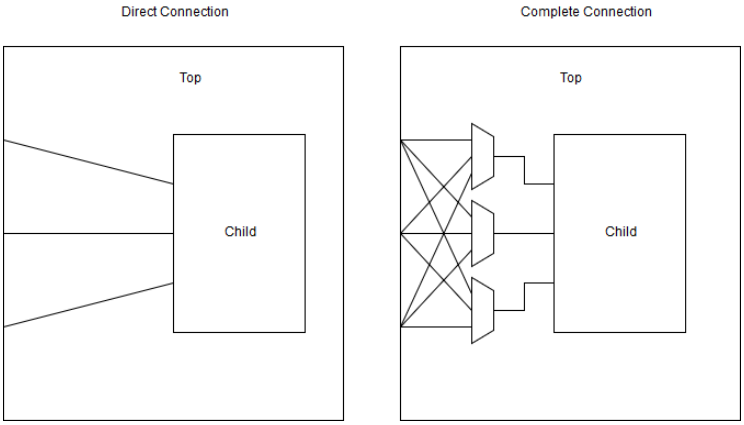

Figure 4.4: Direct connection and Complete connectivity

block, to the input and output pins, respectively. In this dissertation, the values seen in most VTR examples were used: respectively 0.15 and 0.10.

Another child tag that can be employed in a `<pb_type>` is the `<pinlocation>` tag that is used to specify what is the locations for the pins around the complex block. This tag was relevant to one of the architectures that were produced in this present work. However, the implemented final architecture used the default setting for the pin location "`spread`" that makes VPR spread the pins equally across all sides of the complex block.

Finally, to interconnect pins through hierarchies the interconnect tag describes what type of connection between ports is created. In this dissertation work most of the ports were connected using direct links, so by using the tag `<directed>`, a 1-1 mapping can be set up between two nodes. However, most of the FPGA also implement some internal CLB routing, and by using the `<complete>` tag, a fully connected crossbar can be described, in which the input pin can connect to all output pins. An example of the difference between direct and complete connection can be seen in the figure 4.4

A crossbar in a standard cell design should be implemented using many multiplexers, which enlarges the global circuit size. As the tool created lacked the ability to create an internal routing the `<complete>` was not used.

4.2.2 ODIN 2

After describing one of the most critical flow inputs, the tools that are employed in VTR can be explained, along with a description of their functionality and challenges faced to implement than in the flow created in the present work.

Like any other design flow designed for FPGAs and ASIC alike, the first step to implementing a circuit is to elaborate and synthesize the circuit that is described in an RTL format. In VTR ODIN 2 is the synthesis tool used, however, as was explored in chapter 3, ODIN 2 is not a well-optimized tool, that lacks support for some language constructs seen in many designs. The reported ones can be seen below.

First many of the 2005 Verilog syntax is considered an error, as ODIN 2 only supports previous standards, for example:

- "Input/output wire/reg" syntax must be separated;
- "generate" statement is not recognized;

Not only the above syntax problems lead to error, but also commonly used writing styles like:

- Some Verilog compiler directives, like 'ifdefs cannot be used;
- Modules that appear in the Verilog HDL input file and are not instantiated by the top module make ODIN 2 fail.
- If one instantiated module, in the top module, lacks a connected pin, ODIN 2 does not recognize the module and fails, instead of acknowledging that the pin is unconnected.

The first solution was to replace this tool by a newer tool like Yosys, as was purposed in [40]. This tool added features for Verilog 2005 and can create the same identical netlist in BLIF used by ABC and VTR.

However, another solution was proposed: instead of using an academic tool for parsing, an industry tool was used - DC synthesis tool. As was already explained this tool is much more efficient and could bring a boost to the synthesis process.

However, DC could not create the output flattened netlist in the accepted BLIF format but only in Verilog. So, ODIN 2 was left in the flow only as a Verilog to BLIF converter tool.

Another problem that was pointed above, in sub-chapter 4.2.1, relates to the BLIF file, which is an academic format file that describes a logic-level hierarchical circuit in textual form. This file used in this work shall become the netlist inputted to the VPR and ABC tool alike.

One of the most important construct a mapped BLIF file relates to flip-flops. To declare one flip-flop, the BLIF file should include the `.latch` model as displayed:

```
.latch <input> <output> <options (...) >
```

As can be seen, only the input and output of a latch could be defined. As was reported by the Yosys tool developer [41, 42], the BLIF file cannot express asynchronous reset signals. So, as Yosys, ODIN will transform resets defined as asynchronous to synchronous ones. A reset modification like this can be a problematic situation in ASIC designs, so a device-wide asynchronous reset to all CLB flip-flop was used in the created designs.

4.2.3 ABC

The ABC [28] synthesis package is a software used to synthesize and optimize the device, this tool is not used as a front-end tool because it only reads a very limited subset of structural Verilog, so it needs ODIN 2 as a first synthesizer and BLIF converter.

So after having a synthesized BLIF netlist, ABC can be used to perform technology mapping for a variable-LUT-size FPGA, generating a circuit mapped into LUTs and flip-flops. The ABC will output a `.BLIF` format netlist of LUTs and flip-flops.

This tool is simple to use in the VTR flow, and not intense study or changes were performed to this tool during this work.

4.2.4 VPR

Versatile Place and Routing (VPR) is the last program run in this flow, and is used to perform:

- Packing: Where functions are clustered in Logic Blocks;
- Placement: Block are assigned to the available FPGA hardware resources;
- Routing: Signals are connected between placed Blocks.

VPR takes, as input, a description of the FPGA architecture, as was detailed on subsection 4.2.1, along with a LUT technology-mapped user circuit in the BLIF format.

After performing the above steps, statistics to the final mapped design, including area and timing reports, along with the needed configuration to implement the circuit are generated.

In this work this tool will be used for two main purposes:

1. Create the minimal programmable device size that can hold the user circuit;
2. Perform Placement and Routing, generating the needed configuration files.

VPR will first perform the packing, creating a netlist of Logic Blocks that will implement the user circuit. Those Logic Blocks will correspond to the functions that are available in the BLIF file. Completed this process a net file is created with all the Logic Blocks and IOs generated.

The next step will be used to create the CLB full structure on the FPGA, where the created Logic Blocks will be placed in the FPGA structure, which is sized up or down as required. In this phase, two major aspects are already known: the size of the device, and the file that will be used to match the functions in the BLIF file to the ones placed in the FPGA.

Finally, the program will try to route the signals on the FPGA. To do this a binary search will be executed, meaning that a random large number of tracks/channel is used first. Then, if successful, it will try to route using fewer tracks until it gets to the minimum number required. The two last significant outputs are generated in the routing step: one with the tracks that were produced by the FPGA device along with the connections that each track has, and the route file that includes the connectivity for the input user circuit.

If the number of tracks is predefined by the user a binary search will not be needed, so VPR will try to route the user circuit with that specified number of tracks, failing if unsuccessful.

A developer that uses this tool has two main options: to provide a generic description of a FPGA architecture and let the tool create the minimal size device (regarding to the number of CLB and tracks/channel) for the input benchmark circuit, or define a fixed architecture providing

a fixed number of tracks/channels and CLBs of an already design device and see if the benchmark of circuit fits or is routable on this device. These characteristics of VPR, allowing to generate or input a fixed size device, are beneficial for the design flow produced in this work.

Some other options can be used; customizing what the tool will do regarding IOs. One of the first relates to the input pins that were included as a backup. VTR program will typically remove unused input pins, however, setting the variable option `-sweep_dangling_primary_ios` as true those inputs are not removed, and will still be considered by the tool.

IOs in an FPGA are another object of study by FPGA researchers. So, this tool also includes some customization related to IOs. Usually, a conventional FPGA contains an IO structure including IO pads around the FPGA fabric, which allows input and output routing along the FPGA border. IO routing is acceptable for a general-purpose device, but in the device aiming to be developed by the purposed work, a routing device will have no significant advantage, as the inputs and outputs that are required are not going to be changed. So, the solution was to let the VPR route the input circuit in the device and letting the tool chose the pads used, and then the location should be maintained on following re-programmable runs. To perform this, VPR has an argument that can be used to set the pad locations, using a file that looks like the one output by the placement phase but only including the pin locations.

To generate this, another python script was created. The script checks the place file and produces a Pads file with only the required information.

Finally, there is a need to develop a tool that will use this information and create the programmable device representation.

4.3 VTR to Programmable Device Generator

The last task to create a working programmable device was to build a program that could get all the previously generated information, and parse it to generate an RTL representation of the programmable device.

This program was developed in C++, has the same functionality, and will follow a similar approach seen in [38]. However, one of the big difference is that the produced program will act as a standalone application instead of a VTR extension. The main reason that leads to the creation of a standalone application instead of an extension was the additional complexity involved in handling with VPR's internal data structure.

As it was already introduced in the above chapter this tool is divided into these four main steps, that will be explained in this section:

1. Reading of the input files;
2. Internal representation of the programmable device generated by VPR;
3. Placement and routing of the user circuit;
4. Generation of output files.

Table 4.2: VTR to Programmable Device Generator Inputs and Outputs

Inputs	Description
VPR Logfile	Containing the number of CLBs and tracks/channel generated
`rr_graph`	Contains the track representation that will be used to create the routing architecture of the programmable circuit
BLIF mapped file	Netlist with the information to create the LUT bitstream
Route File	Routing generated, using the tracks from `rr_graph`
Place file	Was the placement information, can be used to check CLB function
Net file	Contains the Inputs and Outputs for each CLB, plus what are the CLBs that implement sequential logic
Architectural file	Used to check the CLB interconnectivity

The designed program takes the VTR flow outputs as inputs. After the steps above, the program will generate the programmable device specific files.

4.3.1 Input Reading

Many inputs are required of this program as the representation of those can see in Table 4.2 along with a small description of the functionality.

In this phase, the program checks if these inputs are present, and as a fail-safe the program terminates if these inputs are not present, signaling the design flow that some error occurred before, most likely in the VPR step (for example, failed to route the user circuit).

Many of this input files were already explained in this chapter, except the generated `rr_graph`, and the VPR logfile. The `rr_graph` file includes the information of the tracks and connectivity that is created from VPR. The VPR logfile is a file that is generated from the standard output generated information during VPR runtime execution. In this logfile, one can find two important pieces of information, the generated size for the device, and the number of tracks/channel that was used.

4.3.2 Internal Architectural Representation

In this step, an internal description with the generated VPR architecture is created, using information from the VPR logfile, the `rr_graph` file, and the place file.

Two main points must be set up so that a full architecture can be generated:

- CLB architecture;
- Routing Structure.

The first thing performed to create a CLB architecture is to create an internal representation of the structure using the same number of CLBs as generated by VTR. This is the main task performed by this tool regarding CLB architecture in this phase, as this version still does not support automatic creation of all internal CLB architecture as described in the XML file and will only

create a specific one. However, the program still allows customization of some CLB attributes, for example, one can change if the CLB has 4-LUT or 6-LUT.

In the presented work, the CLB internal architecture was not the target of intense studies. Therefore, CLBs produced only included 4-LUT. However, in chapter 5, the six-input LUT modules were also tested, so that area comparisons are possible.

Finally, the program using the information available in the `rr_graph` will create the routing structure for the programmable device. As the `rr_graph` will contain all the tracks, the program saves this information in memory, and will also keep every track connection called in the file as edges.

As said in the subsection 4.2.2 internal CLB routing was not created in this dissertation, given that VPR will perform separated inter-block and inter-block Routing. So, VPR routing output will only contain routing information for the external routing (using the `rr_graph` file), and to this date, no other file included inter-block routing like the `rr_graph` file (using nets and edges). Combining this routing is a future work for VPR toolset, so the tool create in this work could be improved if this process is incorporated into VPR.

At the end of this phase, all the necessary conditions for the creation of a Verilog file are in place. However, before writing the Verilog file, the program will first create the bitstream.

4.3.3 Bitstream Creation

Now that the program has the internal representation of the device in memory, the place and route step can be performed. This step will use the inputs generated from VPR: place, net, route and the BLIF file. As seen before, the architecture can be divided for the CLBs and Router, including the bitstream creation that is also split in that way.

Listing 4.2: BLIF, PLACE and ROUTE file example

```
\\BLIF file:

\\(...)
.latch n349 FF_NODE~3 re clk 0

\\(...)

.names FF_NODE~3 n1449_1 n1450 n1452 n349
01-- 0
-110 0
\\(...)

\\PLACE file

\\(...)
```

```
n349  23 22 0 #269
\\(...)

\\ROUTE file

\\(...)
Net 376 (FF_NODE~3)

Node: 9400 SOURCE (23,22) Class: 4
Node: 9406 OPIN (23,22) Pin: 4 clb.O[0]
Node: 17071 CHANX (22,22) to (23,22) Track: 3
Node: 23403 CHANY (22,21) to (22,22) Track: 9
Node: 9021 IPIN (22,22) Pin: 3 clb.I[3]
Node: 9015 SINK (22,22) Class: 3
Node: 23403
       CHANY (22,21) to (22,22) Track: 9
Node: 16864 CHANX (23,21) to (26,21) Track: 14
Node: 9404 IPIN (23,22) Pin: 2 clb.I[2]
Node: 9398 SINK (23,22) Class: 2
\\(...)

\\NET file

\\(...)
<block name="n349" instance="clb[269]" mode="clb">
      <inputs>
            <port name="I">n1449_1 n1452 FF_NODE~3 n1450 </port>
      </inputs>
\\(...)
```

CLB bitstream

The bitstream for the CLBs is created using three primary files: the place file, the BLIF mapped netlist and the net file.

First, there is a need to check the place file that will have the information for the position of each function that is present in the BLIF file. For example, using listing 4.2 as an example: the program can learn from that the CLB in position (23,22) must implement a function. After knowing the location and function of a single CLB, the program will check the input position regarding the ports created for it for this the program will check the net file. Moreover, by using the BLIF file that includes plus the mapped information generated by ABC the program can generate

the bitstream for the included LUT. In example listing 4.2, this the same inputs that are present in the net file can be seen in the BLIF file. Also by checking the netlist, the program can decide if that CLB will implement a sequential or combinational function, which is also relevant for the bitstream. In listing 4.2 is seen that n349 is the combinational output (from the LUT) but the CLB needs to implement a sequential creating the `FF_NODE~3` net.

Routing bitstream

Regarding the routing bitstream, the program will check the route file. This file contains all the routes (called nets in the route file) created to connect the device signals. As the program already included the connection for every track, the program goes through the file and saves the configuration connections in memory. For example, if the track t1 is connected to tracks t2, t3, and t4; and in the route file the first line was the track t1 and the next line t2, it means that the track t1 must route a signal to t2.

The bitstream that is created depends on the routing architecture structure. As it will be explained in the next chapter, the functional device will route tracks using fully encoded multiplexers, so in the example above the t2 track has a multiplexer with one of its input being the track t1.

One example of this process is seen in listing 4.2 where the net `FF_NODE~3` path is listed. The program shall check for the nodes (tracks) and perform the needed process to create the bitstream. Giving the same example, if the node 16864 was a track containing four connections, then a four input multiplexer was produced. If the previous node 23403 was connected to the second input, then analyzing this route file the program should create a bitstream that would select the second input after programming.

4.3.4 Output generation

The last thing needed in this program is to generate the output files, and the ones that are produced correspond to:

- Verilog with top module for the Programmable Device and modules;
- Bitstream that will implement the user circuit;
- Synopsys Design Constraints (SDC) file to be used in synthesis.

The Verilog output file will be an RTL representation of the saved device. The tracks contained in the input `rr_graph.echo` file will be converted to wires in the Verilog output; track connectivity will be represented using multiplexers. All these created multiplexers are instantiated, along with the CLB structure in the top module. For example, in listing 4.3 one can see a multiplexer generated to the net 17107 that represents the track that goes from (26,22) to (28,2). As can be seen, the input tracks are already represented. However, the program must check the file `rr_graph.echo` that contains the connected tracks.

Listing 4.3: Example of a Generated Multiplexer

```
mux2 mux_n_17107_CHANX_26_22_to_28_2_t15 (
      .input( {
                  n_24720_CHANY_28_22_to_28_3_t10    ,
                  n_24725_CHANY_28_23_to_28_4_t5 }
      ),
      .select( config_bits[ 27116+:1 ] ),
      .output( n_17107_CHANX_26_22_to_28_2_t15)
);
```

Also, a configuration array is also instantiated, more information about the architecture created can be found in the next chapter.

The bitstream file will have a bit representation of all the needed configuration to implement the original user circuit.

Finally, the SDC file contains timing constraints generated from the bitstream parsing that can be used in the synthesis process to create an optimized timing device for the original circuit. The usage of this File will be explained in the next chapter.

One example for this output files can be found in Appendix B

4.4 Conclusions

The design flow that was produced uses many different tools. In this chapter, the tools were studied so that they could be implemented in the created flow.

The VTR framework implements a design flow for FPGA developers, which has the unique ability to produce a minimal size customizable device that can hold the user input circuit. Usually, tools available for FPGA developers are only focused on creating bitstreams and statistics for already produced devices.

However, implementation problems were found, mainly regarding the synthesis tool that is employed in this flow, ODIN 2. The solution adopted was the creation of a synthesized circuit using Design Compiler before it was input to the flow. Along with some scripts, this solution could fully implement in the created design flow.

The VTR design flow was also explained; this flow is created using the ODIN 2 was the synthesis tool, the ABC as the technology mapping tool, and finally the VPR tool, as the place and route tool.

The VPR tool is the most valuable and customizable tool in the VTR flow. It is used in two different ways: first to have the minimum size device, and afterward to create the representation of that device. Challenges and modification to this program were also explained.

Along with this program functionality, the inputs given to the program were described. The most important one is the XML description file that can easily describe the FPGA internal architecture.

Finally, the VTR to Programmable device generator program was detailed. The goal of this program was to creates the Verilog representation of the programmable device plus the bitstream for the user circuit. To construct the program, many challenges were faced, as there was a need to have specific knowledge about the VPR outputs, and about the architecture that will be set up. The construction of this program was a time-consuming task, but finally, a working version was made that could be used to generate a testing version of the device. Also, a script was developed to run this design flow automatically, that can be seen in the appendix D.

Once the design flow to create a programmable device was in place, its functionality could be tested. The next chapter will detail the architecture used to validate this design flow, and how it was created.

Chapter 5

Description of a Programmable Architecture

In previous chapters, the created design flow was fully explained and detailed. To conclude the study of the design flow, there is a need to determine a programmable architecture it can produce, considering that this should be able to implement the originally intended application and allow small modifications.

To studies this type of architectures one must explore two key characteristics that can be found on devices with programmable functionality:

- Programmable Cores;
- Routing Structure.

Regarding the programmable cores, in Chapter 2 many of this type of cores were presented; however, in this chapter, the study that leads to the chosen programmable core is explored in 5.1.

This dissertation followed an architecture like FPGAs, but many of the internal circuits are too much specific to be used in a standard cell-based flow. For this, a study of this circuits is needed so that a replacement is found. This thematic will be explored in section 5.2.

Having study alternatives for specific FPGA circuits, one can focus on the internal routing architecture that can be found in these devices. In section 5.3, this topic will be explained using two different design approaches explored in this work. This section will conclude with brief result comparison between this two structures and an area comparison on the produced architecture on 5.4.

Having the architecture correctly setup up, on section 5.5 the verification method created to test functional correctness on this devices is explored.

5.1 Specific programmable cores

In Chapter 2, various circuits with programming capability were introduced, as the basic Universal Logic Blocks (ULB) and the common Lookup Tables (LUT) seen in FPGAs.

ULBs are generic cells with an intrinsic characteristic allowing the implementation of any function of N number of inputs depending on the way the inputs are ordered.

In [11] a ULB was constructed using only NAND gates, this study allows the creation of an implementation using the less number of inputs or gates, leading to a circuit that had six inputs to implement all three input functions.

The presented circuit can change the implemented functionality by modifying how the inputs are applied to the created circuit. However, not only the setting of the inputs could be amended, but also the circuit inputs can be interconnected, biased, or permutated. The input setting changed the way the circuit worked.

As normal, if a designer wishes to include on field programmability an additional circuit is needed in the input of a ULB so the order of the inputs could be easily changeable. However, as said in Chapter 2, no routing block was found in this study.

Nonetheless, this dissertation tried to create a simple IO routing block circuit. To this, one followed an approach like a crossbar. Remembering the ULB circuit includes six main inputs to implement a three input function, including an additional circuit output, counted as an input (the fifth pin from figure 2.3), a four to eight crossbar circuit, needs to be created. As this dissertation needs to use a standard cell approach, this crossbar needs to be implemented using multiplexers, and to this, there are a need eight four input multiplexers. As will be explored in this chapter, one of the most problematic modules that will increase the overall size of this programmable device is the configuration module, used to save configuration bits. So, to have a comparison point between this circuit and the one used in this dissertation, the number of bits make a fair comparison point between circuits. So, if the multiplexers used are "fully encoded" ones, two bits are needed by the multiplexer, adding up to sixteen bits per crossbar, in this implementation.

This study was conducted because in the presented article only the process to create the ULB circuit was discussed, an IO routing was not presented. Also, any tool that could map functions to this type of circuits was not found, so in this dissertation, an easier approach was followed.

The next programmable core studied was LUTs, that are circuit implementations of truth tables. To create a comparison point with the ULBs one can create a LUT using, $2^3 = 8$ configuration bits. Compared with the ULBs a LUT implementation needs eight fewer configuration bits than a ULB implementation.

However, the number of cells used in the ULB could be lowered if it was found a way to include this input switching in the routing structure itself. Nonetheless, this practice was not studied and if in a future work this type of circuits could become a solution to implement a programmable device, not only this architecture should be further studied like the routing structure that could allow programmability.

As previously said, there is a lack of tools that help in the creation of an ULB type of structure. However, this is not the case for LUT, as FPGA are prevalent devices, one can find many tools helping in the creation of this circuits. So, this work followed an architecture approach based on the FPGA architecture.

5.2 FPGA Architectures and Production

On Chapter 2, FPGAs were introduced, they are one of the most typically used programmable devices. Most FPGAs are constructed using a full-custom design approach allowing the designer to create small and optimized internal circuits. However, many of this circuits are not available in a standard cell approach.

In this section, will be detailed the study that was conducted to create a programmable device based on FPGAs, that can be synthesized using standard ASIC tools.

So, to examine the effects of creating a programmable device using a standard cell approach the following circuits and modules must be explored:

- Complex Logic Blocks (CLBs);
- Routing Modules: Routing Blocks (RB) and Switch Blocks (SB)
- SRAM cells.

First subsection 5.2.1 describes the study that was developed to create a CLB in a standard cell approach.

Regarding the routing, most FPGAs use full-custom optimized circuits, to connect/disconnect wires. One of the most used circuits created are the transmission gates, not available to standard cell ASIC designer. So there was a need to find a replacement solution to this circuits as it will be explored in subsection 5.2.2.

Finally, one of the last topics that must be studied when creating a programmable device is how the device will store the configuration bits. FPGAs use custom optimized circuits, like SRAMs also not available in ASIC standard cell designs, a replacement circuit is explored in 5.2.3.

5.2.1 Complex Logic Blocks

A Complex Logic Block (CLB) is a module that contains the programmable core, the LUT, and usually other circuitry to allow some routing.

CLBs like the one in figure 5.1 contain smaller components, including flip-flops, look-up tables, and multiplexers:

- Flip-flop: Used as the CLB binary register used to save logic states between clock states.
- Look-up Table (LUT): Stores a predefined list of outputs, a truth table, for every combination of inputs. Providing an efficient way to retrieve the output of a logic operation that is referenced rather than calculated.
- Multiplexer: A circuit that selects between two or more inputs and then returns the selected input, used mainly to allow a user to determine what type of function is going to be implemented on the CLB. To implement a combinational circuit the LUT output should be chosen, to implement a synchronous circuit the flip-flop must be selected.

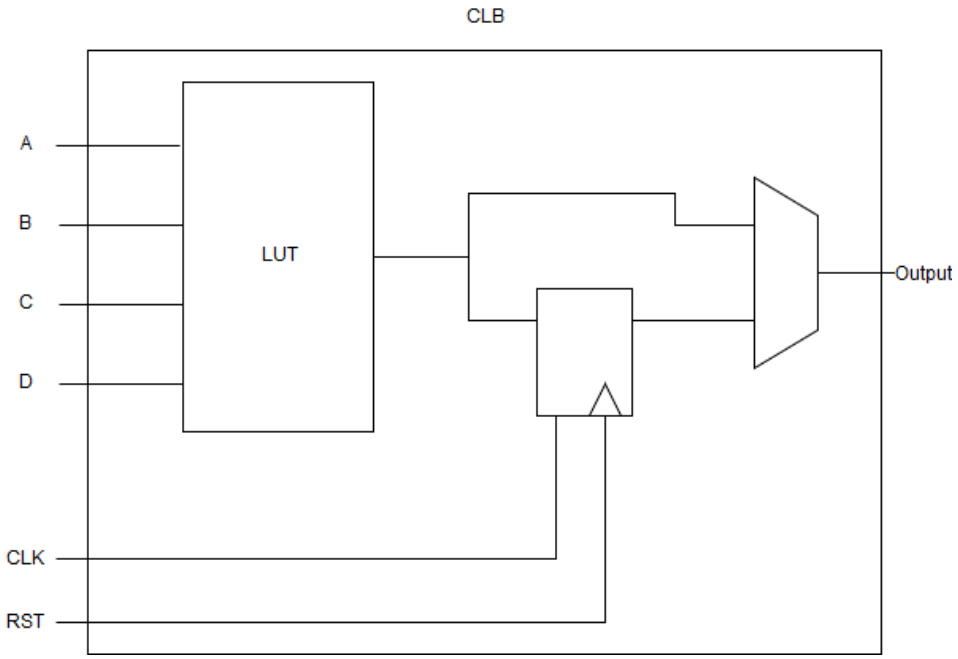

Figure 5.1: CLB Block Diagram

A LUT consists of SRAMs cells that are indexed by the LUTs inputs. The output of the LUT is the value that is in the indexed location of a SRAM. An LUT is usually included only in full-custom designs, if one wants to incorporate it in an ASIC, standard cell design, a hard-core of an LUT must be produced and included. An approach like this could be used in this dissertation to create optimized CLBs and programmable cores; however, one of the objectives introduced is that the produced circuit could be utilized independent on specific hard libraries, but instead a standard cell library can be employed.

To create a device that uses LUTs as the main programming core, FPGAs CLBs must be further studied. Modern FPGAs CLBs usually not only contains one of the above components, containing complex routing structure and multiple modules including other types of LUTs. For example, in [38] CLBs were created like the Altera 8 CLBs FPGA, which uses fracturable input LUT.

Also, individual trees driving clock and reset signal often are included in FPGAs, allowing proper routing for these particular signals through a different routing tree.

This dissertation follows a simpler approach, where the CLB does not contain internal routing and only one 4-LUT/CLB. This method can mean that the overall size of the circuit can increase as there is a need to have many CLBs per device, to check the overhead of doing a design like this one designs changes was also tested but not implemented in the final design.

As it will be explored in 6.3.2, the full circuit size is directly correspondent to the number of configuration bits that it needs to include. So, if instead of using 4-LUTs one could use a six-input LUT module (which means that the design flow could pack the functions in fewer CLBs) the overall size of the circuit could be shortened. This test was done using the circuit benchmarks that will

Table 5.1: Architectural Results using four input LUT

6-LUT	Tracks/ Channel	Number CLBs	Configuration Bits
ALU4	18	361	63 572
TSENG	27	196	61,352
SHA	22	1 849	215 586
SPLA	18	324	34 963
DIFFEQ	16	729	74 087
ELLIPTIC	14	400	38 016

be employed in the following 5.1 chapter. In Table 5.1 and 5.2, the number of CLBs, Tracks/channel and configuration bits generated for each benchmark is presented for implementation using 4-LUT and 6-LUT respectively. So that in table 5.3 one can find the difference between using a four LUT module and a six LUT module. As can be seen, the mean average of configuration cells used is almost one, so virtually any difference was found. So, the simpler approach of using a four-input LUT was followed in this work. However, other configurations plus CLB routing should be further studied in future work.

Three more points on the CLBs must be explored:

- Lookup table SRAM cells (this will be further explored in subsection 5.2.3);
- Clocks;
- Reset.

Clock and Reset are special signals that are usually routed through a specific routing infrastructure, even allowing multiple clocks and resets in a single design. However, this could mean the creation of another routing infrastructure that leads to higher device size. So a single clock and reset were implemented for all CLBs.

Regarding the clock, no special clock tree was built to route the clock, and neither routing is implemented, all CLB flip-flops run to a single clock. For the Reset signal, the produced architecture will treat it a standard input. As explained in 4.2.2 the VTR flow is not able to create a device that has a synchronous signal, transforming any reset signal in the original description into an asynchronous one. To minimize the problem a synchronous reset signal was included in the new

Table 5.2: Architectural Results using six input LUT

4-LUT	Tracks/ Channel	Number CLBs	Configuration Bits
ALU4	16	529	67 513
TSENG	12	841	61 213
SHA	16	2 916	186 002
SPLA	14	484	36 861
DIFFEQ	14	961	74 851
ELLIPTIC	10	484	29 006

Table 5.3: 4-LUT and 6-LUT comparison

	ALU4	TSEN	SHA	SPLA	DIFFEQ	ELLIPTIC	Mean
Tracks/ Channel	0,889	0,44	0,727	0,778	0,875	0,7143	0,773
Number CLBs	1,378	4,29	1,57	1,493	1,318	1,21	1,747
Configuration Bits	1,062	0,997	0,863	1,054	1,010	0,763	0,974

programmable architecture, but as the clock signal, this new reset signal connects to every flip-flop in the programming device. Other methods for creating CLBs for this programmable device could be further studied.

5.2.2 FPGA Routing

Routing on FPGA was discussed previously in 5.2.2, as it is a common problem on FPGA designs. By using special devices like crossbars, or switch boxes, this issue can be solved. For example, switch boxes are used to route wires that drive many signals in an FPGA. These modules are usually built using transmission gates, but these cells are not available in common standard cell libraries[19, 38].

In [19] the FPGAs routing modules (SB and RB) were the base modules that allowed the creation of modules that could be synthesizable. Most of these modules were similar to the FPGA routing modules, but a decision had to be made regarding a specific circuit used in these modules. In FPGAs, the common switch used to connect and disconnect wires are transmission gates. This circuit is a simple one that can be used to block or pass a signal from input to output using only two transistors. However, this type of circuit is not included in standard cell libraries.

So, to produce those modules, a replacement circuit had to be used. In [19] work, a three-state buffer was used to produce the same functionality, allowing its output port to assume additionally to the 0 and 1 logic levels, a high impedance state. However, in recent research made for this dissertation, this type of circuit was not found in the used standard cell devices. So, another circuit is used to implement routing capabilities on the produced architecture.

The solution to this was found in [38], where all the routing is done using a multiplexer. As explained previously, the design flow basis is work in the VPR program, storing every single generated track and connection for the FPGA device, in memory. The program that will create the RTL description of the programmable circuit needs to extract this information, and if a track has more than two incoming tracks connections, a multiplexer is created.

Using figure 5.2 as an example, VPR will have every track (n1, n2) saved along with incoming connections, for instance, n1 will have the n6 and n8 as incoming tracks. So, the generator program needs to create a multiplexer with n8 and n6 as inputs, as it is seen in the left circuit.

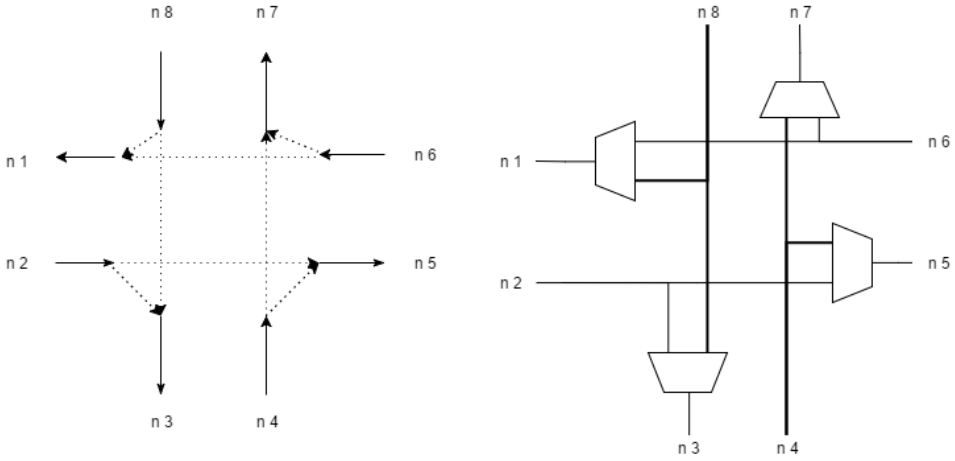

Figure 5.2: Created multiplexers from wires on a switch box

5.2.3 Configuration Bits

The modules explained above depend on a full array of cells that could be used to store the configuration bits. As explained FPGA use SRAM cells, but in this dissertation, D-Flip-Flops were used to create the configuration chain.

The configurable chain for programming purposes was based on the work of [19] and [38], and the final design can be seen in figure 5.3. It was created connecting flip-flops with an additional enable signal in a chain. Moreover, using a special clock all the bits can be shifted to the needed position. Every individual module in this chain is an input that is the output of the last device (or the input of the programmable circuit of this flip-flop is the first one on the chain), and the output is connected to the other flip-flop module (or the output of the programmable device). To shift the bits along the chain, a special enable was also created that is connected to all chain multiplexers.

Another multiplexer was added to separate the internal flip-flop device from the programmable chain, to not compromise internal devices (like routing multiplexers) during programming phase as many bits will be shifting, and to keep the same output value in the cell during the programming phase.

As this array is implemented as a serial chain, problems could be expected if at least one component in this chain, like a flip-flop or other chain cell breaks or malfunctions.

The number of cells on the overall device depends on the configuration bits needed. The number of flip-flops used in a LUT corresponds to the same number of SRAM cells seen in an FPGA LUT, for example, a 4-LUT using 16 SRAM will have 16 flip-flops in the programmable design. On routing, "fully encoded" multiplexers were implemented, meaning, a 4-to-1 multiplexer will have two configuration bits.

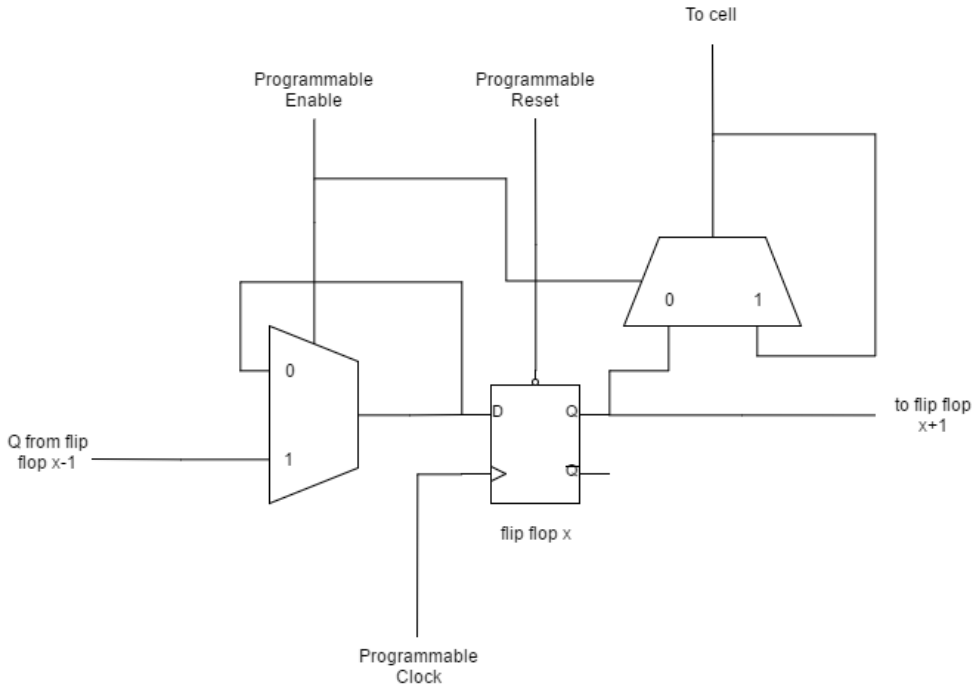

Figure 5.3: Configuration Flip-Flop Chain

5.2.4 Summary

In this subsection, the circuits that could not be used in an ASIC design were explained, giving emphasis to the transmission gates, that are used in the routing infrastructure, and SRAM cells used to create a programmable configuration memory device.

Transmission gates and SRAM alike are custom made cells, built optimized built to be included in conventional FPGA. As seen, this circuits cannot be used in this work: so, transmission gates were replaced by multiplexers, and the SRAM by Flip-Flops.

5.3 Routing Structure Studies

In the previous section, the circuits that could not be used in a standard cell design approach were explained, and solutions were purposed. In this section, the study made to create the routing structure is presented, where two main structures were presented. However, only the last was fully operational and tested for results. The main difference was seen in these architectures how the routing was constructed:

- Routing structure using modules: The top architecture instantiates multiple modules like the ones that are present in FPGAs, CLBs, RB, and SB. All of them were built independently of each other, and that can be parameterized, depending on internal characteristics.
- Architecture routed using only multiplexers: In this architecture, the RB and SB were replaced with tracks produced by VTR and routed using multiplexers.

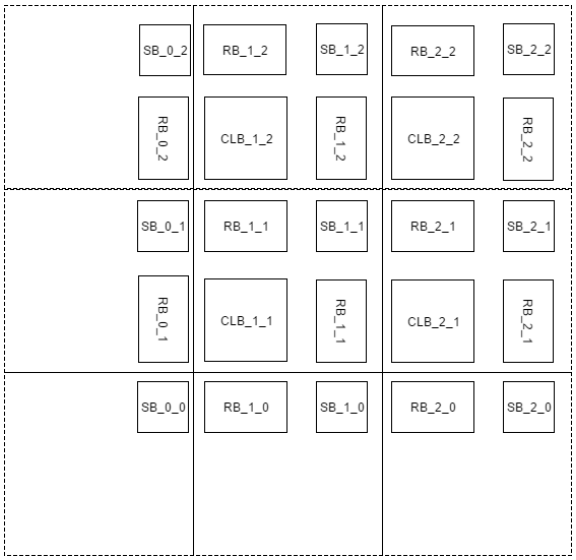

Figure 5.4: Figure of module created using modular approach

This section is divided in the above study where the first one is based on [19], and the last one relies on the work of [38].

5.3.1 Routing Structure based on basic Modules

The first approach to create a full FPGA architecture (following [19]) approach, lead to the creation of a device including some modules based on similar ones that can be found on an FPGA, the Complex Logic Blocks (CLBs) the Routing Blocks (RBs) and the Switch Blocks (SBs)

A grid-like device is created where, depending on the position, every grid position could include one or more of those three modules, an example device can be seen in 5.4.

Each module used is standard FPGA modules: An RB is a module that only contains tracks and multiplexers that allow choosing the input and output connectivity. The SB is a module that is built also using multiplexers that can allow RB to be connected with each other (an example of an SB can be seen in the figure 5.2 in the right representation).

One of the first problems seen is that in [19] the modules were created accordingly to size, that eases the development process, however, inefficiently to be used for this purpose, as the number of tracks depends from application to application.

Another important thing to note is the connectivity between RB and CLB. In [19] a basic type of connection is explored, where CLBs could only have inputs from above RB and outputs to the below RB, for example, CLB on position (1,1) in figure 5.4 had is inputs come from RB on position (1,1) and output to RB (1,0). This connectivity leads to a larger number of tracks in the device, this is so unusual that all the example architectures included in VTR framework have CLB IO connectivity from all sides.

So, to solve the first problem the modules that would be generated should be parameterized according to the number of tracks, so a Verilog module created using the "generate" key Verilog word, enable that track number to be parameterized according to what was generated. The second problem, regarding connectivity, had not a clear solution, in subsection 5.4, two different architectures are presented and compared.

This decision to first create a circuit using the explain architecture became a challenging process, as the program that should have this task needs grid coordinate sensibility. Becoming also challenging to implement owing to the amount of wires and interconnectivity that is required between modules, which creates a very complex overall circuit.

Giving as an example the figure 5.4, if the program is trying to generate a circuit like that, one of the first decisions that the program must take is where to put each of the modules. One of the firs requirements, it as to make sure that CLB modules would only appear on grid positions with x larger than zero, and y also bigger than zero. Also, connectivity depends on the position, for example, if a routing block is found in the position x equal zero, it will only have a connection with the CLB in y equal one, connecting its output.

A scratch program to implement this type of structure was created for this dissertation. However, it was not tested either optimized, as another solution was found before trying to optimize this implementation.

5.3.2 Routing Structure based on Multiplexers

The architecture explained above was challenging to optimize, however, another implementation approach was adopted following [38] studies.

As a reminder, VPR program, using a general description of an FPGA architecture, produces an internal representation of the device where every track and connection is saved. This approach can enable the creation of a program that extracts this information and produces an RTL description of this node and its connections, using the approach explained in 5.2.2 where every connection was clustered as the input of a multiplexer.

After these modifications, some new problems were faced. In conventional FPGAs, many studies are done related to IO pads, which is the part of the FPGA where the inputs and outputs are placed around the device, in the best positions to allow better routing. As the produced device will only implement a specific circuit with the same inputs and outputs, there is no need to have a complex IO structure implemented. In the programmable circuit that will be created the inputs and outputs will be placed where the VPR tool puts decides. To allow re-programmability, the location of this inputs cannot be changed. It can create timing problems as the pins will not be placed in the optimized position, but as it will be seen in chapter 6, reconfigurability will not give many problems.

Alongside the normal IOs from the original circuit, some new pins were added to the device; these pins added the following programming features:

- **Serial input** – input pin for the flip-flop serial chain.

Table 5.4: Pins used on new architecture

Original pins		Programmable specific pins	
Inputs	Outputs	Inputs	Outputs
Input a	Output a	**Clock**	**chain out**
Input b	Output b	**reset**	
...	...	**enable**	
Async reset		**chain in**	
Sync reset a			
Sync reset b			
Clock			

- **Reset pin** – pin that reset all the values contained in the chain.
- **Programming clock** - used in the programming phase to push the bits through the array.
- **Enable signal** - allows or disables the shifting of the bits.

All the pins that can now be found in the new circuit can be consulted in 5.4 with the added ones marked in bold.

As explained, this is based on the study performed by [38] work. However, one of the differences to this work is that the architecture produce is not fixed, and VPR program can implement and study many types of architectures, but for this work, only a unique and generic FPGA architecture was implemented, however, is parameterized related to the number of CLBs and tracks according to the initial description.

5.4 Architecture Decisions Details

Having explored the architecture details surrounding specific circuits and exploring two different routing decisions, this section will conclude the study of the architecture produced in this dissertation.

To do this, in this section both architecture decisions will be compared so that a final decision can be made. To conclude, specifics features about the decided architecture will be explored, mainly regarding size features.

5.4.1 Architectures Comparison

The two different architecture decision explained in 5.3 are different implementations that should produce the same result. However, both architecture produced different outcomes regarding area values.

As the first architecture had some slight implementation challenges, this comparison will only be based on a simple circuit that will be will also be used to test reconfigurability in chapter 6.

Table 5.5 includes the resulted number of CLB and tracks/channel between three decisions: the first two follow an architecture based on [19] implementation. The difference is on the connectivity

Table 5.5: Resulted number of CLBs and tracks/channel depending on architecture

Architectures	Number CLBs	Number Tracks/Channel
A	256	16
B	256	8
C	256	12

where architecture “A” CLB inputs come from the above RB and outputs go to the lower RB. The architecture “B” as full connectivity as CLB has inputs and outputs coming from all surrounding RB. The final architecture “C” follows a similar approach to the work of [38] using a connectivity that is more regular attributes explored in 4.2.1, created from the XML file presented in A.

As expected the number of CLBs keeps equal between every configuration, as the number of CLB is not influenced by the connectivity, but by the implemented functionality. However, the number of tracks/channel is very different from architecture to architecture. As can be seen, the first architecture uses the most number of tracks. As normal, the routing needs more complex routing signals need to travel to very particular positions. When using the second approach, where the connectivity is much less specific than in the "A" architecture, the number of tracks reduces to 50%.

Synthesizing this designs an area comparison can be produced between them, seen in 5.6. As expected, when adding more routing capability more logic is created, since all routing blocks need to have routing logic routing CLBs IOs to all the routing blocks, and even if the number of tracks/channel is smaller an increase of 85% is found from "A" to "B". However, the best result can be seen when using the multiplexer routing where the area was reduced to about 80% from "A" to "B".

Nonetheless, as it will be explained in the next chapter, this number is still large, compared with a fixed logic ASIC implementation the programmable device is still 250 times greater.

Table 5.6: Area comparison between tested architectures

	# Stdr Cells	Area
Architecture A	20 369	19 262
Architecture B	238 536	22 467
Architecture C	4 684	4 423
Fixed Logic Asic	30	17,5
Ratio A/ASIC	679	1104,3
Ratio B/ASIC	784,5	1288,1
Ratio C/ASIC	154,9	253,85

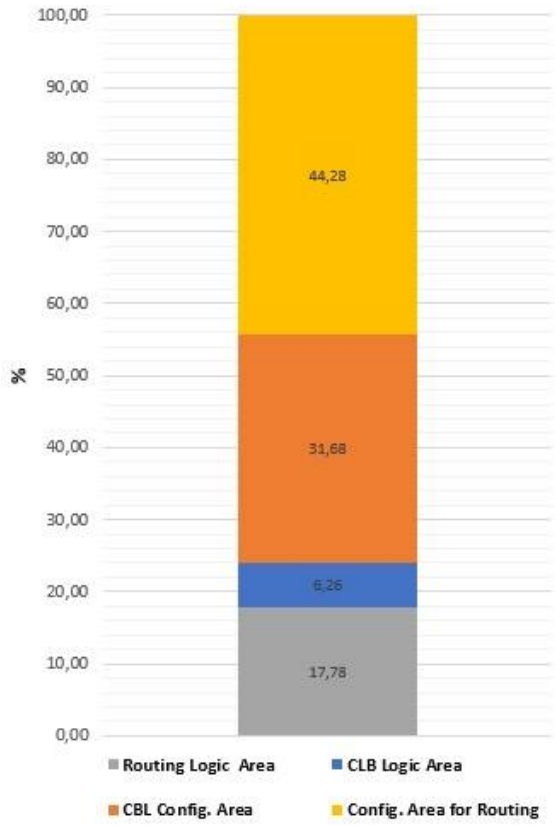

Figure 5.5: Comparison between different Programmable device modules

5.4.2 Architecture Details

After having the final architecture synthesized, studies were performed so that one can know why the device is so much larger than the original.

So, using the same example circuit, one can extract where the area is being applied: this study was divided the area wastes on CLBs and the routing modules. Another decision was made to the area that is used to implement the logic (for example, multiplexers) and modules used to save configuration bits.

In figure 5.5, and with more detail in table 5.7 one can see a this study results. The first conclusion that can be taken relates to the percentage of an overall size that is wasted in memory cells for the configuration bits compared to the contained logic in the device. Around 75% of size is due to memory modules and logic only takes around 25% of the device area.

This conclusion can validate the previous assumption in section 5.2.1, comparing the number of configuration bits directly relate to the area. However, a more detailed comparison will be later explored in 6.

With these results, one can conclude that much more study must be performed in the configuration bits modules, to reduce the size of the overall device.

Table 5.7: Modules comparison values

	CLB Logic Total Area	CLB configuration Total Area	Routing Logic Total Area	Routing Configuration Total Area
Area	277	1400	787	1958
Percentage	6,25%	31,68%	17,78%	44,28

5.5 Functional correctness

To test the design flow and the architecture produced a testing method should be created; testing gave the ability to check the architecture planned and also the tool that created the bitstream and the architecture. Figure 5.6 displays the verification flow.

A testbench was created that could include the original circuit and added the programmable device, and test if the two circuits had the same functionality. This testbench was divided into some steps and then simulated using Synopsys VCS® functional verification tool:

1. Programming phase;
2. Original circuit equality test phase;
3. Re-programming phase;
4. Re-programmability equality test phase.

The testbench first should include a minimum of three modules: The original description of the circuit, a description of the same circuit but with some modifications, and the description of the generated programmable device.

The first step, the bitstream for the unmodified circuit is inserted in the programming chain, to do this one must first raise the enable signal, that will switch the chain multiplexers and deactivate the multiplexers in the flip-flop output. Afterward, a clock should be generated, and at every clock, a bit should be inserted.

The programmable device should be ready to implement the same functionality as the original fixed logic device. Consequently, to test if both architectures could perform equally in the same circumstances, some random vectors (in this case 1000 were tested) were inserted in both circuits inputs, and the output was tested for equality. This simulation tested if the bitstream was correctly setup up as if both circuits perform equal using the same inputs.

The final thing left to do is to test if the architecture can be reprogrammed. So, first the array was cleared, and the next bitstream was inserted. Re-programmability test could now be performed. The testbench now employed another random vector applying it to the modified circuit and the re-programmed device, using the bitstream that would implement the changed circuit functionality. The output was tested again for equality.

Using this testbench debugging was possible, confirming two important aspects objectives for this work: that a programmable architecture could be generated in a standard cell approach, and that it was possible to re-program it.

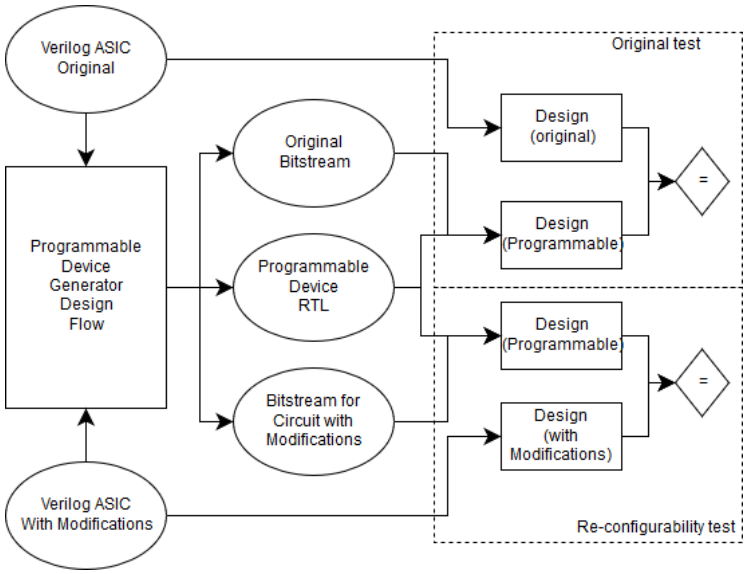

Figure 5.6: Synthesizable Programmable Device verification flow

5.6 Conclusions

In this chapter, the problem of designing a programmable architecture under a standard cell design approach was introduced, alongside with the explanation for the usage of a standard cell design approach.

To create a programmable device, some cores were studied and explored for the creation of a better full programmable circuit. This study lead an architecture that was similar with an FPGA, using LUTs as the programmable core, this type of device was already the base of many research and many tools can help on the creation of devices like this.

To use FPGA type of devices some architecture details had to be studied as in FPGAs many customs made circuits are employed to create an optimized device. However, to construct a device like this using a standard cell approach some circuits must be replaced by ones that can perform the same functionality, and that can be found in standard cell libraries.

After the basic circuits had been created, the decision to how to design the full device was explained. By exploring two approaches, in one creating a grid-like device using the basic standard FPGA modules. This design turned out to be an extensively large design and the program that generated it did not allow many standard FPGA configurations related with connectivity, so size of the device produced was almost 80% larger compared to the final produced.

For this reason, the design flow was improved to create a circuit that used the VTR produced architecture, replacing routing modules by the tracks generated VTR, and implementing routing capability using multiplexers. This new architecture approach turned out to be advantageous, allowing the that VTR to Programmable device generator to only implement routing using multiplexers and not using complex Routing Block and Switch box modules.

Having this architecture ready to implement, some area waste was seen. This area waste was

more noticeable using purely standard cell modules to save the configuration bits; this module occupied almost 75% of the overall area of the device. Some future work should be done in these modules to reduce size.

Finally, to test the final device, a testbench was created that could check the main objectives that should be seen in this dissertation:

- Design flow assessment.
- Validation of the architecture produced;
- Re-programmability functionality;

Chapter 6

Implementation and Results

In previous chapters, the design flow that was developed for this dissertation was explained. First, a small overview of this flow was the main topic of discussion in chapter 3, concluding with a full detailed summary of the used tools, in chapter 4. This flow had the purpose of creating a programmable device architecture as the one explained in chapter 5. After this, there is a need to create the device so that results can be taken. This chapter will be divided between these aspects, the implementation, and the analysis of results.

First, a brief explanation will be given about the correct way a designer must approach the development of a programmable device. This flow should be drawn up like figure 6.1, being divided into the developed programmable generation design flow and the standard ASIC flow used to implement any ASIC design.

One of the main objectives of this work was the creation of a design flow that could be easily implemented on top of a traditional ASIC design flow. So, this dissertation aimed at studying all the necessary tools and appropriately interconnect them, creating a scenario where there is no intervention from the ASIC developer, used as an "out of the shelf" kind of flow. However, there is still need to plan the ASIC standard flow carefully, so in section 6.1 this flow will the target of a brief description.

Having a clear understanding of the ASIC flow, some problems detected in the implementation step are going to be explained in subsection 6.2.1, along with the proposed solutions.

With design problems solved, a more detail study was needed for the ASIC flow, as many strategies can be followed to synthesize an ASIC design. All the approaches that have been investigated and later used will be explained in subsection 6.2.2.

This chapter will conclude with some validation and results analysis. The first results will be generated by comparing circuits implemented in ASIC fixed logic flow to the ones that were generated using the programmable generator design flow. The circuits that were used correspond to a subset of combinational and sequential benchmarks provided in the MCNC benchmark suite [43].

These same benchmarks will be then used to compare the approach followed in this dissertation to an FPGA approach studied in [38]. However, some caution is required when comparing the

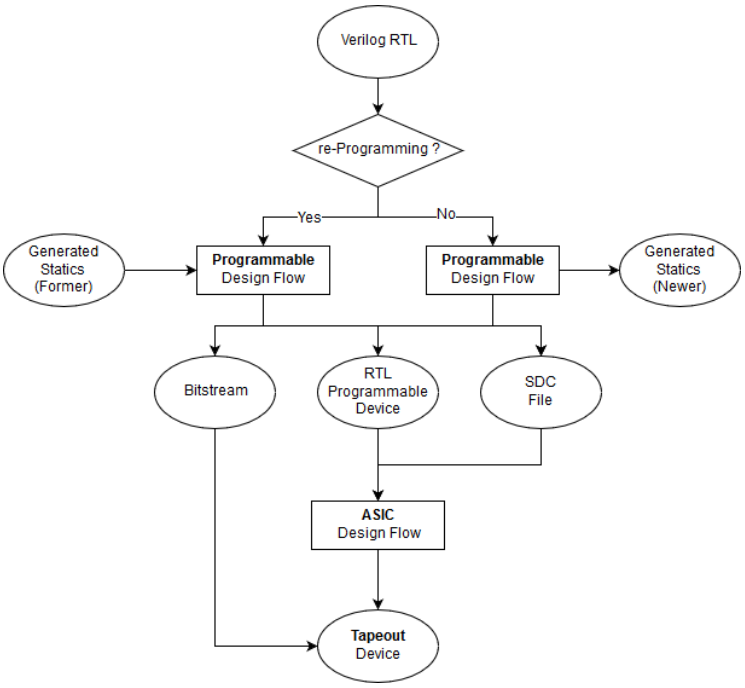

Figure 6.1: Proposed Design Flow

results, as the device created in the cited article was implemented using 65nm technology, while the proposed implementation was performed on a commercial TSMC 28nm technology.

Finally, a circuit was created to be the target of re-programmability. So a programmable device was constructed, based on that circuit, that allowed "small" modifications to be performed on it so that re-programmability characteristics could be tested, in agreement with the motivation for this work.

6.1 Programmable Device Design Flow

The first approach to create a programmable device is to design the flow that will have the functionality of automatically generating the intended device from an original RTL representation of the circuit to be programmed. This flow should be divided into two main steps: first, the designer shall run the flow that was created in this dissertation, so that afterward, a standard ASIC flow can be executed to implement the design and ultimately tapeout the developed circuit.

In this dissertation work, the focus was the production of the programmable device generator design flow. However, the ASIC flow, also required intense study to be deployed.

This section will be divided into the steps shown in the figure 6.1, explaining the designer steps to create the programmable device.

6.1.1 Creation of Programmable Device

The first task to produce the programmable device is to go through the flow that is proposed in this dissertation; this should be an easy process that runs without many interventions from the user.

Independently on the final purpose of this flow - creating the RTL for the device or creating just the bitstream with “small” circuit modifications - it will always produce three outputs. As a reminder, the outputs are a Verilog RTL describing the programmable device, the bitstream to implement the inputted design, and an SDC file constraining the design for a timing driven configuration. The SDC was introduced in chapter 3, however only in 6.2.1 the full functionality of this file will be fully explained.

If the designer wishes to perform "small" modifications to the circuit, the RTL and SDC output file can be ignored. Nonetheless, these outputs were kept so that debugging could be possible. As in a flow run to perform "small" modifications an RTL file should be produced equal to the one already implemented. The SDC file was also kept so that some timing analysis could be performed, (more in-depth information in subsection 6.3.4) .

6.1.2 ASIC Design Flow

One of the main objectives of this work was the creation of a design flow that could be easily included in a normal workflow of an ASIC designer. So to fully test the programmable generation design flow, two points must be studied: first, the architectures that are allowed in a Standard Cell flow, already explained in chapter 5; and the process to implement and create a full ASIC flow that will create this re-programmable device.

The ASIC flow was based on Synopsys tools with the help of provided documentation, tutorial, and Synopsys guidance, so that this tool could be utilized to the best of their capabilities, figure 6.2 shows the employed steps.

Normally an ASIC design flow is divided into two main stages, one that corresponds to the functional, logical implementation phase, where a designer ultimately should perform the Synthesis, and another, a physical step terminating in a Layout.

The synthesis process is often performed using common synthesis tools, being Design Compiler RTL the synthesis tool employed in this dissertation. Synthesis is the process analyzing, elaborating, and finally mapping the Verilog RTL to a technology base standard cell library. Designers must initially provide a Verilog RTL file, constraints, and a target library. With this netlist, a designer can already have clear guidance on the circuit that will be produced.

The netlist that will result from the synthesis process can vary more or less depending on every of these inputs, depending on the designer decision. Using the constraints file, a designer can customize and optimizes the synthesis process, depending on requirements. On the other way, even the synthesis can be altered so that some additional logic could be added, for example adding clock gate to the design. More on the Synthesis process can be found on subsection 6.2.2.

The last step in an ASIC flow is the physical implementation, where the placement and routing for the created circuit are performed. This process is automatic using the tools available, like

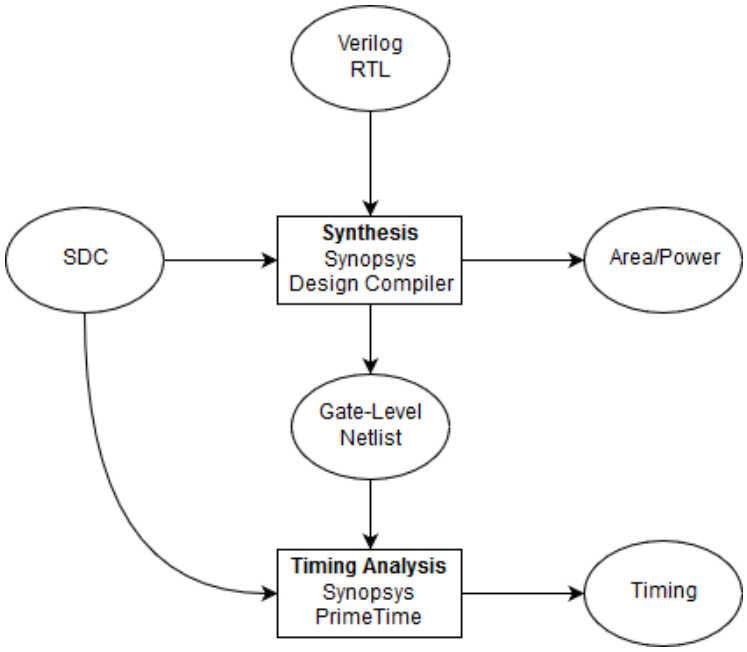

Figure 6.2: ASIC Flow produced

Synopsys IC Compiler™, that can perform automatic place and route for the chip-level physical implementation phase.

6.2 Implementation process

Having an overview of a standard flow that should be followed to implement any design, there is a need to apply it so results can be produced.

First in subsection 6.2.1 problems that were noticed during synthesis, related with architectural decisions will be explained. Moreover, in subsection 6.2.1 challenges faced in the creation of the ASIC flow will be detailed.

6.2.1 Design problems

The architecture that was developed is based on FPGA architectures (as explained in chapter 5) as it is one of the most popular programmable architectures available and because there are many FPGA-based tools available. However, to implement a design like this, not only the architecture needs to be carefully studied but also the synthesis process turns out to be challenging.

So the initial approach was to synthesize the device using a normal, standard ASIC flow, using common flow examples. However, at the starting of the synthesis process, the warning "OPT-314" appeared, stating that timing loops were found in the circuit. Timing loops correspond to circuit combinational feedback loops present in the design. These loops are problematic for synthesis tools, as these loops increase the number of cycles a synthesis process will try to optimize in an infinite amount in the same path. In DC, warning "OPT-314" states that the tool will try to disable

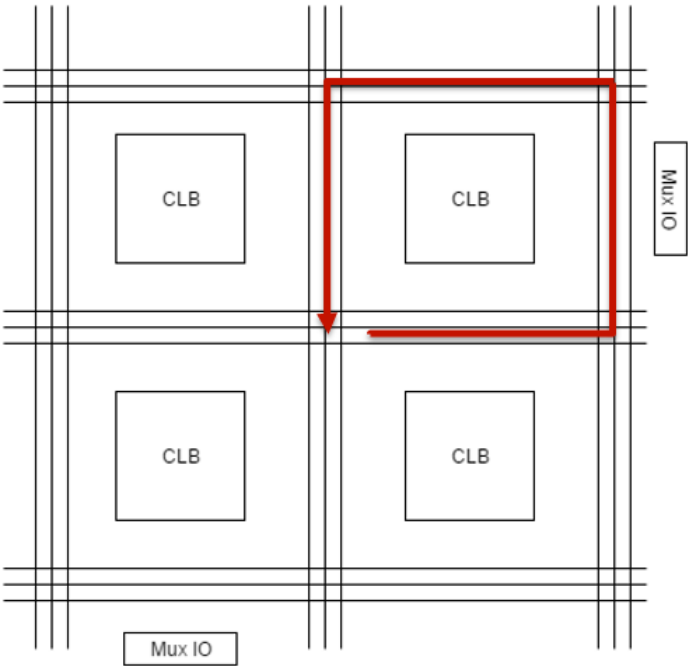

Figure 6.3: Feedback loop in routing infrastructure

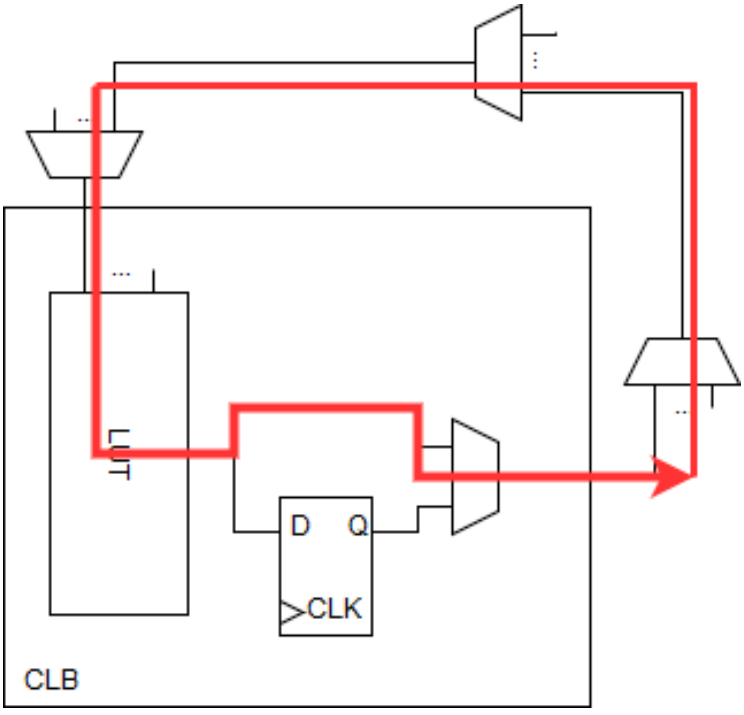

Figure 6.4: Feedback loop in CLBs

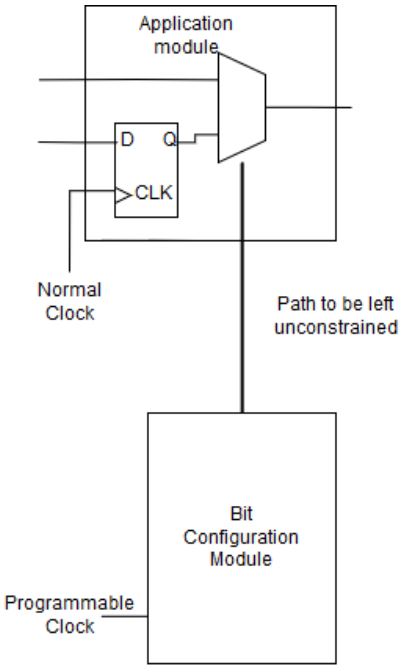

Figure 6.5: False Path to be create

some arcs so that paths can be properly formed. However, this can lead to optimization that is undesirable by the designer and in most cases unnecessary.

Timing loops often are considered design flaws that require debugging, however in this dissertation, as the produced architecture is based on FPGAs, combinational loops are common before configuration. A loop can be seen in figure 6.3: this loop was exclusively generated by the routing architecture. However, in figure 6.4 can be seen that CLBs are also a source for this loops.

To solve this problem [38] generated an SDC file with some constraints that could guide the synthesis process, and remove these loops.

The studied option shall optimize the device for the original configuration, as DC includes the ability to add timing constraint. The one used in this case was `set_disable_timing` that allows the designer to pinpoint the correct paths for the design, disabling others. For example, if the generated design contains a multiplexer with four inputs, there is a need to create an SDC file that will disable timing checks on some of those inputs. If in this example, the second input is selected by the configuration bit, the SDC file will need to contain three constraints disabling timing check on all the inputs except the second one. On the contrary, if any input is selected, the SDC file needs to constraint all four inputs.

The complete SDC file contains the routing bitstream that is accepted by Synopsys tools; as it is produced based on the generated routing. By using this file in 6.3.4 a better way to report timing was produced.

Finally, another issue related to timing optimization had to be solved: as the device is reconfigurable it needs to include two clocks, one to be used in the programming phase and another in normal operation (if the circuit implemented is sequential), so they are not supposed to work

simultaneously. Nonetheless, the synthesis tool will try to optimize the device to run efficiently under this two clocks and will try to constrain paths that do not need optimization. For example, in figure 6.5, displays one of those paths, which will originate from the output of the configuration cell to an application port (for example the configuration input for a CLB multiplexer). This wire, during normal operation, will be subject to the value that was placed in the programming phase, and will not change, independently from the value of the inputs.

So, to instruct the tool not to check this type of connection the constraint `set_false_path` removes the optimization in wires that connect this two clocks, is generated by the script.

6.2.2 Synthesis approach

The next step in this work is to study the best way to synthesize the programmable device in a traditional ASIC flow. One of the first decisions to take when synthesizing a device is the compilation approach, as there are three main strategies to follow: top-down, "uniquify," or bottom-up.

In a top-down strategy, the synthesis tool will look to the design without any hierarchy and will try to synthesize all blocks as a single circuit, compiling all modules from top-level down together.

Another approach is to break the design in multiple instances and compile each one individually; this strategy is often characterized as an "Uniquify" strategy. Using an "Uniquify" approach the CPU time, and memory size is reduced considerably compared with the top-down approach, as every instance is compiled individually and then linked together to form the top module. However, as each instance of the same module (for example the CLB) is compiled individually, synthesis process on large design with many similar instances can become time-consuming.

The last studied approach corresponds to the bottom-up one in each module is synthesized one time only, and then every one of the instances is linked together to create the full synthesized device.

The first approach studied was the "Uniquify" strategy, however, was tried only once during this dissertation work, as an example design containing 16 tracks and 529 CLBs (corresponding to a grid with 23 by 23 CLBs) took around 26,46 hours of CPU time during synthesis. Afterward, a Top-Down approach was considered, however, as it is a more demanding task than the "Uniquify" strategy this approach was not tested. On [38] this method was tried with no success, as the synthesis resulted in failure. Therefore, a bottom-up approach was followed, and a CPU time improvement was seen, as the same design took only 4,31 Hours.

As will be explored in this chapter, the benchmark circuits were taken from a the MCNM benchmark set [43]. However, one first result can be acquired from the synthesis process: CPU time, that can be seen in table 6.1. The average time that takes to compile a design in a bottom-up strategy is much faster compared with the 26.46 hours observed in an "Uniquify" strategy. However, the time that still took to synthesize all these devices was one of the points that made debug challenging.

This dissertation followed a button-up approach, and the first thing needed is to create an automatic synthesis process. So first, to follow this strategy each module that is instantiated by the

Table 6.1: CPU time by benchmark

	CPU Time (Hours)
ALU4	4.31
TSENG	4.85
SHA	69.08
SPLA	2.01
DIFFEQ	6.61
ELLIPTIC	6.71
Average (without SHA)	4.89
Average (including SHA)	15.58

programmable device (for example the CLB) must first be synthesized, they were synthesis using a standard top-down approach. Then the top module could be synthesized. The first phase of this process is to read the RTL file containing the top module and elaborating the design. Afterward, there was a need to include each of the designs that were previously synthesized to the process.

This task was also challenging, as every single generated device could contain many different instances, like different multiplexers. So, this created a challenge to create the script that synthesizes the device. A single script was developed for every design tested; in future work, there is a need to study an automated way of creating this script for every design that can be generated.

After reading every module for the design, there is a need to constrained it. First, the SDC file created by the flow was input, ensuring that the synthesis tool optimizes the device for the original configuration. Some more common constraints were applied, first constraining the normal clock to operate at 3 GHz, ensuring that the device is optimized to the fastest possible clock frequency. Also, to make sure that the synthesis tool tries to minimize the area of the device the maximum area was constrained to 0. This constraint is an unrealistic scenario, however, is the standard practice to ensure area optimization stated said in Design Compiler manual [39].

After setting constraints to optimize the device, the compilation was performed. As a standard synthesis has performed, neither Design for Testability (DFT) like scan chains, nor clock gating was included.

After compilation, the Place and Route step could be conducted. However, this dissertation could not successful Place and Route the design. When a large design, like the ones produced, reach the Place and Route step there is a need to perform an initial additional phase: the hierarchical chip floorplanning. This move requires extensive knowledge about placement and physical layout. Nonetheless, this floorplanning was tried, but physical constraining along with detailed placement information needed, that was not generated by the produced design flow at this instance.

Therefore, in this dissertation, the ASIC design flow that was performed ended in the synthesis step, nonetheless as the main comparison point in this dissertation is the ASIC fixed logic circuit, many comparison results can be obtained without performing Place and Route.

6.3 Comparison Metrics and Generated Results

This dissertation will mainly focus on a comparison using five MCNM benchmarks: ALU4, TSENG, SPLA, DIFFEQ, ELLIPTIC.

An additional significant design was also included: the circuit implementation of the SHA retrieved from the VTR flow examples [25].

These six designs were first synthesized to an ASIC fixed logic device, using the same approach that was performed to generate the programmable device. However, a top-down synthesis was performed, as these benchmarks are relatively small and did not contain hierarchies.

In this dissertation three main comparison points and results will be conducted:

1. Area, delay, and power comparison between the ASIC and the programmable device;

2. Area, delay with the work done in the cited [38] article, even through this work produced a device using a different technology library. Nonetheless, some careful comparison and analysis will be detailed.

3. Finally, the re-programmability tests will be made, using PrimeTime to get some timing comparisons.

6.3.1 Measures to compare implementations

This subsection will detail the main results and comparison points that resulted from this work:

Area: using the real area data calculated on Design Compiler, plus the number of standard cells used, and finally the number of equivalent gates (GE) a metric that allows a comparison result independent of a manufacturing technology.

Delay: where static timing analysis will be performed to the gate-level designs using a worst-case timing modules; timing analysis can help determine maximum clock frequency for the design.

Power: Design Compiler also reports power measurements, reporting both static and dynamic components.

Most of these results were compared with values that are reported in [6] and [15]. The produced device as many similarities with a Soft FPGA device. So using the values from [6] one can guess what should be the size of a hard (custom layout) FPGA implementation compared with and ASIC implementation, so that using the values from [15] one can have a close ratio between an implementation on a custom layout FPGA and ASIC.

6.3.2 Comparison between ASIC fixed logic to the Programmable device implementation

The first results extracted will be used to compare a fixed implementation to a programmable device one. So, the first thing to do is to gather results for this kind of application, using the benchmarks that are detailed above.

Table 6.2: ASIC benchmark Area results

	ALU4	TSENG	SPLA	DIFFEQ	ELLIPTIC	SHA	Mean
Area (μm^2)	407	1161	287	1177	213	1860	850,83
# Stdr cells	812	1174	598	1222	249	3176	1205,17
GE	1058	3018	745	3059	554	4835	2 211,5

ASIC Fixed Logic Implementation results

The first results compare the ASIC with the Programmable device from performing a synthesis flow on the benchmarks; the resulting **area** can be found in the table 6.2. As can be seen, the biggest design available is the SHA circuit, that uses close to 3 000 standard cells and occupies and area of around 1900 μm^2. The smallest one corresponds to the Elliptic circuit that uses almost 250 cells occupying a total amount of 213 μm^2.

So, including the SHA implementation, the mean area value of the performed benchmark is around 650 μm^2, increasing to around 850 μm^2 when including with the SHA circuit.

The next measure point is the critical path delay, extracted by the PrimeTime tool, which can be found in the table 6.3, where the mean value is around 2.04 ns.

Finally, the power consumed by this circuits can also be extracted using the Design Compiler reports, a compilation of this results can be seen in table 6.4.

The extracted results confirm that a designer who wishes to create an optimized circuit, not only in the area but also in timing aspects, should use an ASIC implementation. However, as already said, after tapout the circuit is unchangeable: errors are found, they can lead to the disposal of the produced circuit.

Programmable device Implementation results

Having the ASIC values for the benchmarks, one can now extract values for programmable devices created based on this circuits. At the end of the VTR phase (explained in chapter 3) some first results can already be analyzed, as the program will already provide the number of tracks/channel and CLBs that need to be generated. After running the VTR to programmable design generator, one can then compare the numbers of configuration modules that will be required in one design.

As a single device requires many configuration bits, the module that will save those bits enlarges the circuit greatly, as was explained in Chapter 5. So the first results generated were already detailed in the chapter 5, however in this chapter one can compare the configuration bits with the

Table 6.3: ASIC benchmark Delay results

	ALU4	TSENG	SPLA	DIFFEQ	ELLIPTIC	SHA	Mean
Critical Path (ns)	1,51	1,58	3,65	1,61	1,31	2,56	2,04

Table 6.4: ASIC benchmark power results

	ALU4	TSENG	SPLA	DIFFEQ	ELLIPTIC	SHA	Mean
Dynamic (µW)	136,9	194,29	233,36	165,07	33,08	432,55	199,21
Leakage (µW)	79,55	304,79	208,95	291,19	69,17	818,05	295,28

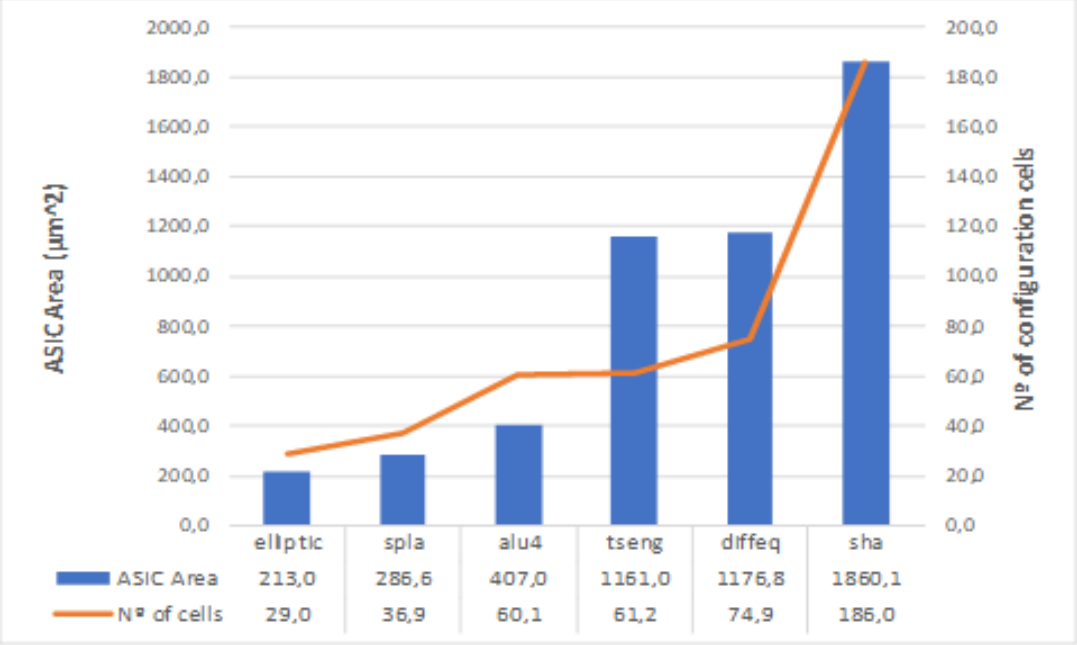

Figure 6.6: Comparison between ASIC area and generated number of Configuration cells

Table 6.5: Programmable Device Area results

	ALU4	TSENG	SPLA	DIFFEQ	ELLIPTIC	SHA	Mean
Area (μm^2)	192 541	201 251	160 254	324 580	112 548	902 518	315 615
# Stdr cells	201 512	212 541	170 215	335 418	120 151	913 254	325 515
GE	500 431,4	523 069,53	416 541,62	843 612,74	292 522,42	2 345 725,8	820 317

area of the original cell. As expected the number of configuration cells correlates exactly with the size of the fixed logic implementation, showing that the programmable device that is created is parameterizable according to size, as can be seen in the graph 6.6.

The first comparison that will be made relates to **Area**, having the first results in the table 6.5, along with GE and number of standard cells. As one might expect the area of the final device is much larger that the device that implements. In average the size, without counting with the SHA circuit, is around $0.2\,mm^2$ rising to almost $0.4\,mm^2$ when compared to the mean value counting with the SHA circuit.

One can see that the number of standard cells used also grows greatly in size. In average the number of standard cells used is around 325 000 that is almost 380 times the number of standard cells used in an ASIC design.

Delay: For getting the critical path PrimeTime was used after synthesis. In the table 6.6 one can see the results for the benchmark. These values are, also unsurprisingly larger compared with an ASIC design. For this reason, ASIC designs are usually created to run at higher frequencies that FPGA-base designs, so this rise is expected.

Power: Finally the power values that are measured are represented in table 6.7. These values are rough estimates as Design Compiler calculates power using the values given by the technology vendor. The results are merely indicative as both routing and node activity can only be estimated.

Ratios comparison

Having the results from both the designs, one can now compare both implementations results.

Area: in the table 6.8 the area ratios comparing an ASIC to a Programmable device implementation can be seen, as a re-programmable implementation is roughly 390 times larger than an ASIC design.

This value is much bigger than what should be expected by analyzing the cited [6, 15]. However, one important issue to realize from both studies is that neither try to compare an ASIC to Programmable device implementation (in the standard cell). In [6] the gap between a full-custom

Table 6.6: Programmable Device Delay results

	ALU4	TSENG	SPLA	DIFFEQ	ELLIPTIC	SHA	Mean
Critical Path (ns)	6,01	4,03	6,23	4,94	5,72	12,16	6,51

Table 6.7: Programmable Device Power results

	ALU4	TSENG	SPLA	DIFFEQ	ELLIPTIC	SHA	Geomean
Dynamic (µW)	4 917,78	5 058,16	3 042,5	6 175,49	1 920,19	17 302,18	6 402,72
Leakage (µW)	8 030,73	8 221,53	4 947,91	10 046,05	4 062,5	2 244,66	6 258,90

FPGA implementation and a standard cell FPGA implementations is measured, with an average increase of 40 times. On [15] the gap studied is between a standard cell and a full-custom FPGA implementation with a mean increase in the size of around eight times. Analyzing both the studies a conclusion can be reached to the gap that should be produced between an ASIC and the created Programmable Device implemented, that should state that a programmable device should be 320 times larger in mean values to the ASIC device.

However, this work, the average value of the size increase should be around 385 times. First, these two results are averages from values that go from 23 to 55 times in [6] and 6 to around 9 in [15]. So, in rough values, a programmable device should be 138 to 495 times larger an ASIC design. Although, the average size being inside these boundaries, the device is still too large for implementation purposes. However, much more optimization is still possible, being one of the most important points to study the configuration cells that occupy more than 50% of the size of a programmable device.

Delay: table 6.9 as the average numbers resulted from delay comparison between the two types of designs.

The values resulted are also larger than the first ASIC implementation, being much closer values suggested in [6], than with the area ratios. These values can be related to a better optimization done by the tool or the usage of a smaller technology, that as significant impact on timing calculation.

Power: Finally the power ratios can be seen in table 6.10, again, these values are rough estimates from reports generated by Design Compiler. However, compared with the values register by [6] the power is two times the mean value reported. However, this is normal as the FPGA cells (for example, SRAM and Transmission Gates) contain much less logic and will normally consume much less power.

Table 6.8: Programmable Device/ASIC Area Ratio

	ALU4	TSENG	SPLA	DIFFEQ	ELLIPTIC	SHA	Geomean
Area	473,07	173,34	559,08	275,81	528,39	485,19	384,71
# Stdr cells	248,17	181,04	284,64	274,48	482,53	287,55	280,5

Table 6.9: Programmable Device/ASIC Delay Ratio

	ALU4	TSENG	SPLA	DIFFEQ	ELLIPTIC	SHA	Geomean
Critical Path	3,98	2,55	1,71	3,07	4,37	4,75	3,21

Summary

Having these results, one can see that implementing an ASIC in a programmable device type of architecture brings significant increases in size, around 380 times the size of an ASIC implementation, this mainly due to the lack of ability to use some special cells employed in FPGAs, like SRAM cells. Although the values are well withing the range of literature values, this is undesired values. Nonetheless, the architecture still lacks some careful fine-tuning, that can be used to decrease the overall area.

Also, the critical path delay is also worsened by around 3.2 times the ASIC implementation, concluding if a designer desires to create high-frequency programmable circuits, this solution cannot be used. However, control devices or other low-frequency devices can be implemented in the created programmable device, if size does not matter.

6.3.3 Comparison between "soft" and "hard" eFPGA to the Programmable Device implementation

Even though that the sizes compared with the ASIC are much larger than expected, one can still try to compare a standard FPGA implementation to the created device. For this, the work of [38] will be used to get a rough comparison. However, there is still need to have some caution as the technology used to implement the design is slightly different from the one that was employed in this dissertation.

In this chapter, so that the comparison is fair, all the values got by the article will be divided by the technology ratio between the used by this dissertation and the one utilized by the article ($65nm/28nm = 2.32$). This value will serve as the K value for Dennard's MOSFET scaling [44, 45].

Regarding **Area**, the scaling factor that is applied to different technologies corresponds to $1/k^2$, meaning that a device implemented in 28nm technology should be 0,19 smaller than implemented in 65nm technology. Using this factorization, every benchmark will be compared with the size of the two reported article devices: "soft" and "hard" FPGA design, (as can be seen in figure 6.7). As on can see in the DIFFEQ circuit implemented in ASIC is almost 610 times smaller than

Table 6.10: Programmable Device/ASIC Power Ratio

	ALU4	TSENG	SPLA	DIFFEQ	ELLIPTIC	SHA	Geomean
Dynamic	35,92	26,04	13,04	37,41	58,053	40,00	35,08
Leakage	100,96	26,97	23,68	34,50	58,74	2,74	41,26

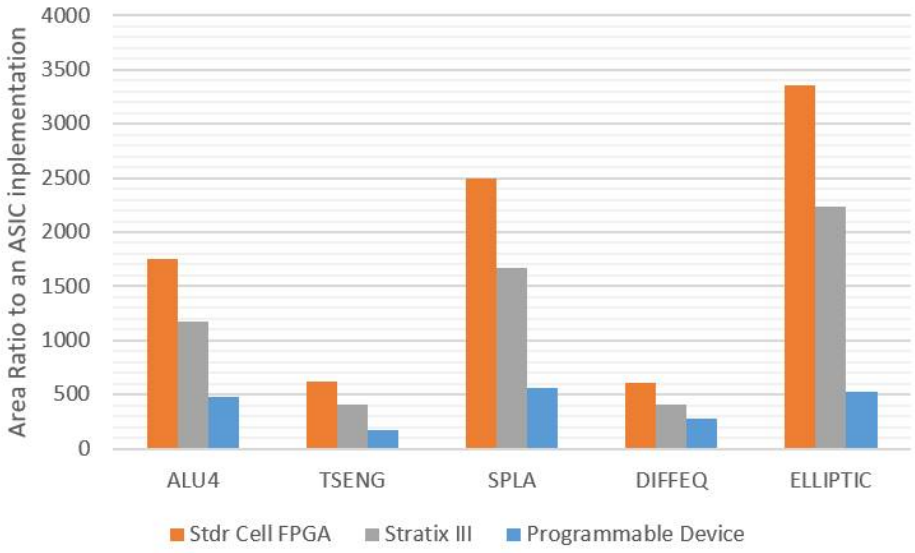

Figure 6.7: Programmable Device and FPGA Area comparison to ASIC

a Standard cell FPGA, 400 times from the Stratix III and around 280 times this work implemented device.

However comparing only the programmable devices, some more interesting comparison can be made, as can be seen in the figure 6.8,

For example, the ELLIPTIC circuit, in the produced programmable device was almost four times smaller than a hard FPGA solution and six times smaller than a soft FPGA solution.

The final comparison that can be made relates to **timing**. Also, Dennard scaling values will be used, where the delay time scaling factor is 1/k. Results can be seen in table 6.11. These values show that this work implementation is much more optimized than the soft FPGA approach. This value is normal, as the soft FPGA is much larger design than the one created in this work. However, the critical path delay was some differences compared with the hard FPGA approach, being the dissertation created device is a slower than the hard implementation. This result is also expected, as a hard implementation was developed using optimized circuits, which are optimized for timing purposes.

Table 6.11: Delay comparison between FPGA devices (ns)

	Stdr cell FPGA	Stratix III	Programmable Device (2.2 times)	Ratio to Stdr cell FPGA	Ratio to Stratix III
ALU4	22,29	5,29	13,22	0,6	2,5
TSENG	20,05	4,52	8,87	0,44	1,96
ELLIPTIC	32,45	6,91	12,59	0,39	1,82
SPLA	34,92	6,65	13,71	0,39	2,06
DIFFEQ	23,28	4,39	10.86	0,46	2,47
Mean	25,95	5,45	11,70	0,45	2,15

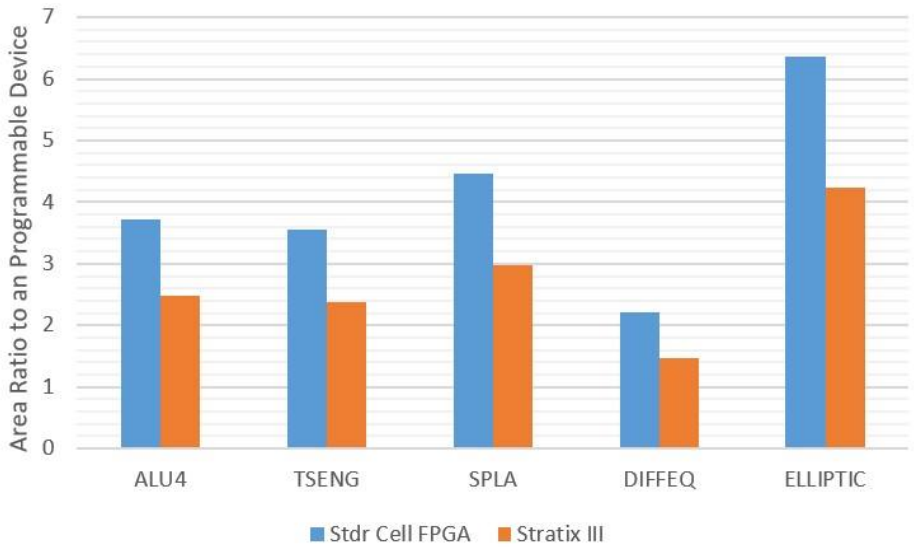

Figure 6.8: FPGA Area comparison to the created Programmable Device

These two result can be used to conclude the comparison from a "hard" and "soft" FPGA device, to the created device in this dissertation, leading to some better indicators regarding area than the above devices. However, the implemented device needs a more careful study to be much more optimized than it is now.

6.3.4 Modification experiment

In this subsection, a 3-bit finite state machine will be used to test re-programmability. This FSM is the same that it was used in chapter 5 to make some area studies and comparisons. However, in this chapter using PrimeTime, the modification will be tested. Diagram 6.9 displayed the timing testing flow performed.

This RTL was used to create two devices. One in ASIC fixed logic and another to the programmable device. For testing purposes, one more column and row were added to the programmable design, and an additional input was also added. In the table 6.12 the characteristics for the generated programmable device can be seen.

As expected the produced programmable device has ratios closer to the ones that were calculated in the benchmarks. The delay increased around 3.3 times and the area around 255 times.

To test re-programmability, four more bitstreams were created. The RTL with changes was also used to create four more ASIC devices, characteristics of this device critical path delay and power can be found on 6.13. Area results can be accessed on table 6.14. By using the method described in 6.9, table 6.15 contains extracted delays, and the ratios to the original ASIC devices. More detail on the modifications can be seen in annex C along with the used RTL. However, for a simple summary can be seen below:

- **Modification A** - State machine logic replacement (for example, an OR for an AND);

Table 6.12: Values for Generated Programmable device based on FSM

	Programmable Device
Area (μm^2)	4 423
# Stdr Cells	4 648
GE	11 495
Dynamic Power (µW)	1,42
Leakage Power (µW)	2,85

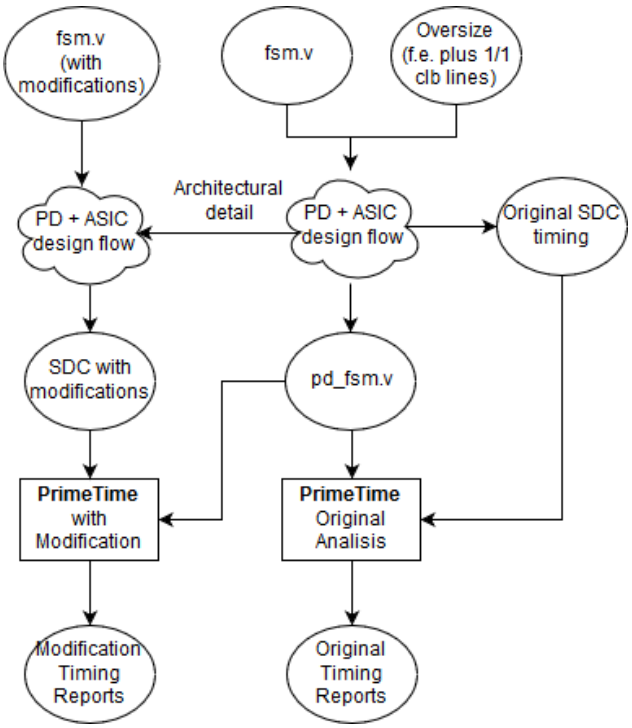

Figure 6.9: Diagram with re-programmability tests

Table 6.13: ASIC FSM Delay and Power values

	Critical Path(ns)	Dynamic Power(µW)	Leakage Power(µW)
Original	1,22	3,765	4,48
Modification A	1,19	4,84	4,8
Modification B	1,33	4,995	5,00
Modification C	1,2	5,60	6,53
Modification D	1,41	6,625	6,25
Mean	1,27	5,17	5,21

Table 6.14: ASIC FSM Area values

	Area (μm^2)	# Stdr Cells	GE
Original	17,44	30	45,3
Modification A	18,98	34	49,3
Modification B	20,13	37	52,3
Modification C	26,16	45	67,99
Modification D	26,55	47	69,00
Mean	21,85	39	56,78

- **Modification B** - Added a state to the state machine , backup input used, more logic adding/replacement;
- **Modification C** - One of the last modifications possible compared to the D modification;
- **Modification D** - Aggressive modification to reach failure in reprogrammation.

Related to the delay one can see that the mean critical path delay value is close to the original value. Even the critical path delay from the first modification has decreased compared to the original. However, when adding the new input, the critical path delay increases around 1.1 times the original value. This increase in the delay could be related to the poor placement made by VPR as this input was not initially characterized and no timing analyses were performed around it. Also as the SDC generated breaks timing in paths closer to this input, the synthesis process could not produce timing optimization on paths leading to this pin.

Finally, the last modification suffered the same problem that of the B modification. However, some improvement from B is seen, that could be related the increase in logic can make signals go through some optimized paths.

6.4 Conclusion

This last chapter had two main objectives:

- Explain how the implementation of the architecture described in chapter 5 should be performed.

Table 6.15: Delays and ratios to ASIC designs

	Critical Path	Ratio Delay	Ratio # Cells	Ratio Area	Ratio Total Power
Original	4,07	3.37	288.1	253,58	44,97
Modification A	3,99	3.35	254.2	233,02	51,9
Modification B	4,445	3.34	233.6	219,66	40,34
Modification C	4,29	3,57	192,1	169,07	37,85
Modification D	*Failed*	-	-	-	-
Mean	4,19	3,4	239,45	216,44	43,47

- Evaluation and validation of the architecture and the corresponding design flow, described in chapter 3, using the tools explained in chapter 4.

Regarding the first point, one can approach the following conclusions:

1. The architecture that was explained in 5 is very complex, resulted from the immense number of instances that are included.
2. The synthesis process found some flaws in the design, mainly on timing optimization related points.
3. Timing analysis is a complex task on a design like this, as many paths can be formed. In this dissertation, one followed the approach of optimizing the device for the original configuration following [38] study.
4. Creating an ASIC flow is a significant task, and there was a need to study multiple synthesis methods, eventually choosing the bottom-up approach. However, this choosing created a more challenging design flow that eventually led only to post-synthesis results.
5. The synthesis process is rather slow on design like this, difficulting the architecture and design flow debugging

Using the post-synthesis results, one can conclude:

1. Initial results show a strong correlation between the number of configuration modules that are needed and the area of the programmable device that is generated.
2. The gap between the programmable device and an ASIC design is still large, where the generated design is in average 385 times bigger the ASIC implementation.
3. The ratio delay as values that are closer to the literature than the area, concluding that the optimization of the synthesis process is rather helpful.
4. Using a kind of design like this can increase the chip area in large amounts compared with an ASIC situation. However, this approach is better than an eFPGA one, as the circuit is rather optimized for the design, where an eFPGA does not depend on the design but a predefined architecture.
5. Re-programmability is feasible in the created device. The delay values are still impacted, but the average is still close to the initial delay. However, there is still need to perform more studies on the types of changes that can be processed successfully by this architecture.

Chapter 7

Conclusions and Future work

In this dissertation, the standard ASIC design flow needed to be studied, along with current available programmable device design flows. This study was complemented with an architectural overview of some already developed programmable devices. This study leads to the creation of a programming device generator design flow, this flow should be easy to integrate into the standard workflow of an ASIC designer, used to create an RTL representation of this programmable device along with the application specific bitstream. The last step is to develop an architecture that could test this design flow.

Therefore, the new fields of study and advances that were performed through this dissertation will be summarized in this chapter, focusing on answering and solving the question that was purposed by the original dissertation objectives.

In Chapter 3, design flow created to fulfill this work objective were introduced along with main advantages and studies perform:

1. Easy integration in an ASIC workflow;
2. Implementation of design flow using common tools used by FPGA and ASIC designers alike.
3. Re-configurability features implemented.
4. Usage of an FPGA investigation toolset: VTR. Used to create an optimized device regarding size, and to Place and Route the application specific circuit;
5. Introduction of the program made during this dissertation work with the functionality to process the VTR outputs and create the RTL and bitstream of a programmable device.

Chapter 4 was used to conclude the presentation of the programmable flow detailing every tool functionality, inputs and outputs. With emphasis on:

1. Usage of Design Compiler to pre-synthesize an application to General Technology;
2. Creation of the needed XML input used by VTR;

3. Integration of VTR into the created flow along with faced challenges, mainly on the ODIN 2 parser and VPR inputs and options;

4. VPR details that allow re-configurability.

5. Full details about the developed VTR to Programmable device generator program required to create an internal representation of the programmable device and Place and Route this device using VTR outputs.

Having the design flow produced and properly explained, in Chapter 5 some challenges on creating a synthesizable programmable device was explained:

1. First introducing common programmable cores that can be implemented on devices, concluding that the architecture implemented used a LUT as the main programmable core;

2. As LUTs are often employed in FPGA base circuits, the production and architecture of this devices were introduced, and some common circuits that are used had to be carefully studied so that a device like this could be synthesizable.

3. Then the two top architectures that have been investigated and implemented in this work were introduced. Concluding on one architecture that implemented routing using Multiplexers.

4. The two introduced architectures were compared, and where the module base solution was bigger and harder to be created, another was smaller and easier to create.

5. Size-related results were analysis, concluding most of the sized waste comes from the configuration bits modules.

To conclude, two final aspects were explored in Chapter 6. First, the implementation was explained where:

1. The differences from the produced design flow and an ASIC standard flow was explored.

2. Some architecture decision need modification so that an optimized device could be created.

3. Common synthesis strategies were explained: concluding on the decision of using a bottom-up approach.

4. The decision to synthesize this device in a button-up approach made the physical design step much more challenging, that eventually was not addressed. However, post-synthesis could be used to extract enough information for a final comparison.

5. These tests exhibited that programmable device was in average 380 times larger, 4 times slower and consumed 40 times more power than an ASIC design.

Chapter 6 also contained some results analyzing and validating:

1. Using known benchmarks the gap generated by the ASIC to Programmable Device implementation was explored. Moreover, as the architecture produced is rather complex and challenging to implement in a standard cell approach, a large gap was seen. Nonetheless, some more optimization can be done to these devices to reduce this produced gap.

2. Comparison to Soft and Hard FPGA devices, showed the potential of using the created design flow, as the programmable device can be almost seven times smaller than any of those solutions.

3. Even though that some decision during this work, led to the creation of an architecture optimized to the original specific application, re-configurability was still possible in the created device, this re-configurability was tested along with the impacts on timing.

7.1 Future developments

Even though that there is a clear pressure from IC market to create programmable devices based on ASICs, an extensive study is needed on this subject. As this device are difficult to produce and are made using rigorous and precise implementation strategies, for example, the full-custom in FPGA designs.

However, this work answer to the question of the possibility to create a design flow that could produce Standard Cell-based Programmable Devices that are automatically parameterized in tracks and size to the application specific circuit developed by the designer.

Nevertheless, further studies should be conducted, not only on the design flow but also in programmable device architecture. Related to the design flow:

1. The creation of some programs that can help on many decision that was assumed in this thesis, like the number of LUTs inputs, that influenced in the size of the device;

2. Need to have a much closer study of the timing optimization process done in VPR so that a much more optimized device could be created;

3. Optimize the design flow, in terms of the VTR flow and the generation problem, so that a larger range of architectures and programmable devices could be future accepted, as in this version only a FPGA-based solution is accepted;

4. Better integration with the ASIC flow, and possible creation of optimized scripts so that synthesis and layout are easier to develop.

Finally, the architecture that was used also requires further work:

1. Study a replacement to the FPGA-based architecture by another optimized device, like using a ULB based solution.

2. If an FPGA based architecture should be maintained, the used modules should be carefully studied, mainly focusing the study on the configuration bits storage modules, as it highly influences the overall size of the device.

Appendix A

XML file used

VTR [25] uses a flexible description language that allows a designer to describe many FPGA architectures. This appendix shows the example XML file used to implement all the benchmarks described in this work.

```
<architecture>

 <models>

 </models>

 <layout auto="1.0"/>

 <device>
  <sizing R_minW_nmos="13090.000000" R_minW_pmos="19086.831111" ipin_mux_trans_size="2.000000"/>
  <timing C_ipin_cblock="0.000000e+00" T_ipin_cblock="1.030392e−10"/>
  <area grid_logic_tile_area="14130.86202"/>
  <chan_width_distr>
   <io width="1.000000"/>
   <x distr="uniform" peak="1.000000"/>
   <y distr="uniform" peak="1.000000"/>
  </chan_width_distr>
  <switch_block type="wilton" fs="3"/>
 </device>

 <switchlist>
  <switch type="mux" name="0" R="0.000000" Cin="0.000000e+00" Cout="0.000000e+00" Tdel="1.571509e−10"
        mux_trans_size="2.000000" buf_size="19.000000"/>
 </switchlist>

 <segmentlist>
  <segment freq="1.000000" length="4" type="unidir" Rmetal="0.000000" Cmetal="0.000000e+00">
  <mux name="0"/>
  <sb type="pattern">1 1 1 1 1</sb>
  <cb type="pattern">1 1 1 1</cb>
  </segment>
```

```
</segmentlist>

<complexblocklist>
 <pb_type name="clb">
  <input name="I" num_pins="4" equivalent="false"/>
  <output name="O" num_pins="1"/>
  <clock name="clk" num_pins="1"/>

  <pb_type name="ble" num_pb="1">
   <input name="in" num_pins="4" equivalent = "false" />
   <output name="out" num_pins="1" />
   <clock name="clk" num_pins="1" />

   <pb_type name="lut4" blif_model=".names" num_pb="1" class="lut">
    <input name="in" num_pins="4" port_class="lut_in"/>
    <output name="out" num_pins="1" port_class="lut_out"/>
    <delay_matrix type="max" in_port="lut4.in" out_port="lut4.out">
     1.51e-10
     1.51e-10
     1.51e-10
     1.51e-10
    </delay_matrix>

   </pb_type>

   <pb_type name="ff" blif_model=".latch" num_pb="1" class="flipflop">
    <input name="D" num_pins="1" port_class="D"/>
    <output name="Q" num_pins="1" port_class="Q"/>
    <clock name="clk" num_pins="1" port_class="clock"/>
    <T_setup value="66e-12" port="ff.D" clock="clk"/>
    <T_clock_to_Q max="124e-12" port="ff.D" clock="clk"/>
   </pb_type>

   <interconnect>
    <direct name="ble_connection" input="ble.in[3:0]" output="lut4.in[3:0]" />
    <direct name="clk" input="ble.clk" output="ff.clk" />
    <direct name="lut_to_ff" input="lut4.out[0:0]" output="ff.D"/>
    <mux name="output_mux" input="ff.Q lut4.out[0:0]" output="ble.out"/>
   </interconnect>
  </pb_type>

  <interconnect>
   <direct name="clb_input" input="clb.I[3:0]" output="ble.in[3:0]"/>
   <complete name="clk" input="clb.clk" output="ble.clk" />
   <direct name="output" input="ble.out" output="clb.O" />
  </interconnect>

  <fc default_in_type="frac" default_in_val="0.15" default_out_type="frac" default_out_val="0.1"/>
  <pinlocations pattern="spread">
```

```
    </pinlocations>
    <gridlocations>
      <loc type="fill" priority="1"/>
    </gridlocations>
  </pb_type>

 </complexblocklist>
</architecture>
```

Appendix B

Example Outputs Generated by VTR to Programmable Device Generator

This appendix shows an example of two of the three outputs generated by the VTR to Programmable device generator.

B.1 SDC

This file is generated to be used in the synthesizer process of the programmable device, representing a set of `set_disable_timing` used to disable the timing analysis in unused pins in the programmable device netlist. Ensuring that the synthesis is optimized for the original application.

```
current_instance

# disable not used Multiplexers
set_disable_timing mux_n_674_CHANY_4_4_t2/*
set_disable_timing mux_n_673_CHANY_4_3_to_4_0_t1/*
...

# disable not used CLBs
set_disable_timing clb_4_4/*
...
set_disable_timing clb_1_1/*

# disable internal CLB multiplexers
set_disable_timing clb_2_4/mux_ff_out_0/a



# disable not used Multiplexers inputs
set_disable_timing mux_n_664_CHANY_3_4_t0/a_i[0]
```

```
set_disable_timing mux_n_664_CHANY_3_4_t0/a_i[1]
set_disable_timing mux_n_664_CHANY_3_4_t0/a_i[2]
set_disable_timing mux_n_664_CHANY_3_4_t0/a_i[4]
set_disable_timing mux_n_664_CHANY_3_4_t0/a_i[5]
set_disable_timing mux_n_664_CHANY_3_4_t0/a_i[6]

...

set_disable_timing mux_z/a_i[0]
set_disable_timing mux_z/a_i[1]
set_disable_timing mux_z/a_i[3]
```

B.2 Programmable device RTL

The following netlist represents an extract generated for a four by four CLB programmable device. This netlist can be synthesized using the above SDC that allows the creation of a programmable device.

```
module programmable_core(
      input wire a,
      input wire b,
      output wire z,
      input wire clk,
      input wire rst,
      input wire pgrm_in,
      output wire pgrm_out,
      input wire pgrm_en,
      input wire pgrm_clk,
      input wire pgrm_rst
      );

      //Wire Declaration
      wire [751 : 0] config_bits ;
      wire [751 : 0] bit_chain ;

      wire n_126_CLB_IPIN_1_1_t0;
      //(...)
      wire n_675_CHANY_4_4_t3;
```

```
    //Pad muxes
    mux2 mux_n_277_IO_OPIN_2_5_t1 (
          .input( {
                a    ,
                b }
          ),
          .select( config_bits[ 0+:1 ] ),
          .output( n_277_IO_OPIN_2_5_t1)
    );
    //(...)
    mux4 mux_z (
          .input( {
                n_276_IO_IPIN_2_5_t0    ,
                n_279_IO_IPIN_2_5_t3    ,
                n_282_IO_IPIN_2_5_t6    ,
                n_285_IO_IPIN_2_5_t9 }
          ),
          .select( config_bits[ 4+:2 ] ),
          .output( z)
    );

    //Global Routing Multiplexers

    //Track with only one connection:
    assign n_618_CHANX_1_4_t2 = n_634_CHANY_0_4_t2 ;

//(...)

    mux7 mux_n_664_CHANY_3_4_t0 (
          .input( {
                n_358_CLB_OPIN_3_4_t4    ,
                n_454_CLB_OPIN_4_4_t4    ,
                n_610_CHANX_2_3_to_3_0_t0    ,
                n_612_CHANX_3_3_to_4_0_t2    ,
                n_613_CHANX_3_3_to_4_0_t3    ,
                n_615_CHANX_4_3_t1    ,
                n_660_CHANY_3_2_to_3_0_t0 }
          ),
          .select( config_bits[ 461+:3 ] ),
          .output( n_664_CHANY_3_4_t0)
```

```
);
//CLBs

clb_top clb_1_1 (
      .I0( {
                  n_129_CLB_IPIN_1_1_t3,
                  n_128_CLB_IPIN_1_1_t2,
                  n_127_CLB_IPIN_1_1_t1,
                  n_126_CLB_IPIN_1_1_t0
      } ),
      .O(  n_130_CLB_OPIN_1_1_t4 ),
      .config_bit (config_bits[ 480+:17]),
      .clk(clk),
      .rst(rst)
 );
//(...)
clb_top clb_4_4 (
      .I0( {
                  n_453_CLB_IPIN_4_4_t3,
                  n_452_CLB_IPIN_4_4_t2,
                  n_451_CLB_IPIN_4_4_t1,
                  n_450_CLB_IPIN_4_4_t0
      } ),
      .O(  n_454_CLB_OPIN_4_4_t4 ),
      .config_bit (config_bits[ 735+:17]),
      .clk(clk),
      .rst(rst)
 );

//Configuration Chain
ff_module ff_0 (
      .clk( pgrm_clk ),
      .reset( pgrm_rst ),
      .q( config_bits[ 0 ] ),
      .si( pgrm_in ),
      .se( pgrm_en ),
      .so( bit_chain[0] )
);

genvar i;
```

```
      generate
            for(i = 1; i <= 751; i = i +1)
            begin: ff_chain
                  ff_module ff_ (
                        .clk( pgrm_clk ),
                        .reset( pgrm_rst ),
                        .q( config_bits[ i ] ),
                        .si( bit_chain[ i - 1 ] ),
                        .se( pgrm_en ),
                        .so( bit_chain[ i ] )
            );
            end
      endgenerate

      assign pgrm_out = bit_chain [751];

endmodule
```

Appendix C

Circuit used to test Reprogramability

This appendix details the Verilog examples used to test re-programmability.

C.1 Original Verilog

```
module fsm (

   input wire clk ,
   input wire rst ,
   input wire inputA ,
   input wire inputB ,
   input wire inputBackup ,

   output wire outputA ,
   output wire outputB ,
   output reg [2:0] outputC
) ;

   localparam STATE_Initial = 3'd0 ,
   STATE_1 = 3'd1 ,
   STATE_2 = 3'd2 ,
   STATE_3 = 3'd3 ,
   STATE_4 = 3'd4 ,
   STATE_5 = 3'd5 ,
   STATE_6 = 3'd6 ,
   STATE_7 = 3'd7 ;

   reg [2:0] CurrentState ;
   reg [2:0] NextState ;
```

```
assign outputA = ( CurrentState == STATE_1 ) | ( CurrentState ==
   STATE_2) ;
assign outputB = ( CurrentState == STATE_2 ) ;

always@ ( * ) begin
   outputC = 3'b000 ;
   case ( CurrentState )
      STATE_2 : begin
         outputC = 3'b010 ;
      end
      STATE_3 : begin
         outputC = 3'b011 ;
      end
   endcase
end

always@ ( posedge clk ) begin
   if ( rst ) CurrentState <= STATE_Initial ;
   else CurrentState <= NextState ;
end

always@ ( * ) begin
   NextState = CurrentState ;
   case ( CurrentState )
      STATE_Initial : begin
         NextState = STATE_1 ;
      end
      STATE_1 : begin
         if ( inputA & inputB ) NextState = STATE_2 ;
      end
      STATE_2 : begin
         if ( inputA ) NextState = STATE_3 ;
      end
      STATE_3 : begin
         if (! inputA & inputB ) NextState = STATE_Initial ;
         else if ( inputA & !inputB ) NextState = STATE_2 ;
      end
      STATE_4 : begin
                     NextState = STATE_Initial ;
```

```
         end
         STATE_5 : begin
            NextState = STATE_Initial ;
         end
         STATE_6 : begin
            NextState = STATE_Initial ;
         end
         STATE_7 : begin
            NextState = STATE_Initial ;
         end
      endcase
   end

endmodule
```

C.2 Modification A

The original modification only used the initial the first, second, and third state. So, in the first modification, more logic and an additional stage were added to this FSM:

```
         STATE_2 : begin
            if ( inputA ) NextState = STATE_3 ;
         end
         STATE_3 : begin
            if (! inputA & inputB ) NextState = STATE_Initial ;
            else if ( inputA | !inputB ) NextState = STATE_4 ;
         end
         STATE_4 : begin
            if (! inputA & inputB ) NextState = STATE_Initial ;
         end
```

C.3 Modification B

More logic as added in this modification, also the backup input was used as well. The first modification correspondents to the output logic:

```
   assign OutputA = ( CurrentState == STATE_1 ) & ( CurrentState ==
      STATE_2 ) ;
   assign OutputB = ( CurrentState == STATE_2 ) ;
```

In the state machine, some logic was added along with some that used the backup input:

```
        STATE_1 : begin
            if ( inputA | inputBackup ) NextState = STATE_2 ;
        end
        STATE_2 : begin
           if ( inputA ^ inputB ) NextState = STATE_3 ;
           else if ( inputA & inputB ) NextState = STATE_1 ;
        end
        STATE_3 : begin
           if (! inputA & inputB ) NextState = STATE_Initial ;
           else if (inputA | !inputB) NextState = STATE_4 ;
        end
        STATE_4 : begin
           if (! inputA & inputB & inputBackup ) NextState =
              STATE_Initial ;
```

C.4 Modification D

The modification D was the first one to fail.

Aggressive modifications can be seen in this modification, not only in the state machine as also in the output logic:

```
assign OutputA = (inputBackup| inputA) ? ( CurrentState ==
   STATE_4 ) | ( CurrentState == STATE_2 ) : ( CurrentState ==
   STATE_5 ) & ( CurrentState == STATE_6) ;
assign OutputB = ( CurrentState == STATE_1 ) & ( CurrentState ==
   STATE_2 | CurrentState == STATE_5 ) ;

always@ ( * ) begin
   outputC = 3'b000 ;
   case ( CurrentState )
      STATE_2 : begin
         outputC = (inputBackup | inputA) ? 3'b010 : 3'b001 ;
      end
      STATE_3 : begin
         outputC = (inputB ) ? 3'b011 : 3'b110 ;
      end
      STATE_4 : begin
         outputC = 3'b100 ;
      end
```

```
    endcase
end

always@ ( posedge clk ) begin
    if ( rst ) CurrentState <= STATE_Initial ;
    else CurrentState <= NextState ;
end

always@ ( * ) begin
    NextState = CurrentState ;
    case ( CurrentState )
        STATE_Initial : begin
            NextState = STATE_1 ;
        end
        STATE_1 : begin
            if ( inputA | inputBackup ) NextState = STATE_2 ;
            else if ( inputA | inputBackup ) NextState = STATE_3 ;
        end
        STATE_2 : begin
            if ( inputA ^ inputB ) NextState = STATE_3 ;
            else if ( inputA * inputB == 1 ) NextState = STATE_1 ;
            else if ( inputA ^ inputB ) NextState = STATE_5 ;
        end
        STATE_3 : begin
            if (! inputA & inputB ) NextState = STATE_Initial ;
            else if ( inputA & ! inputB ^ (!inputBackup | inputB))
                NextState = STATE_4 ;
        end
        STATE_4 : begin
            if (! inputA & inputB & inputBackup ) NextState =
                STATE_Initial ;
            else if (! inputA & inputB ^ inputBackup ) NextState =
                STATE_5 ;
        end
        STATE_5 : begin
            if ( inputA + inputB == 1 ) NextState = STATE_4 ;
            else if (outputC > 0 && outputC < 2) NextState = STATE_6
                ;
        end
        STATE_6 : begin
```

```
            if ( inputA & inputBackup ) NextState = STATE_5 ;
            NextState = STATE_Initial ;
         end
         STATE_7 : begin
            if ( ( (! inputA ) & inputBackup & inputB ) | ( outputC
               [1] & outputC[2] & !inputA )) NextState = STATE_3;
            NextState = STATE_Initial ;
         end
      endcase
   end
```

C.5 Modification C

This modification was made after D one, as removing some logic operations allowed re-programability. The modification done can be seen below:

```
   assign OutputA = (inputBackup) ? ( CurrentState == STATE_4 ) | (
      CurrentState == STATE_2 ) : ( CurrentState == STATE_5 ) & (
      CurrentState == STATE_6) ;
   assign OutputB = ( CurrentState == STATE_1 ) & ( CurrentState ==
      STATE_2 | CurrentState == STATE_5 ) ;
```

Appendix D

Design Flow Example

This dissertation produced a design flow that would bring a complete and automatic process able to produce a programmable device from a standard ASIC RTL representation.

This flow runs from a produced script on a dedicated work directory, like the one seen in figure D.1.

D.1 Work Directory

So that the design flow can be run the working directory shall be divided into the following folders:

- **RTL** – This folder is where the designer shall introduce the RTL files with the description of the circuit to be implemented.
- **Tmp** – Most of the temporary files, like the GTECH netlist and VTR flow generated reports, will be saved in this folder.
- **Include** – This folder will contain the architecture description file used in the flow.

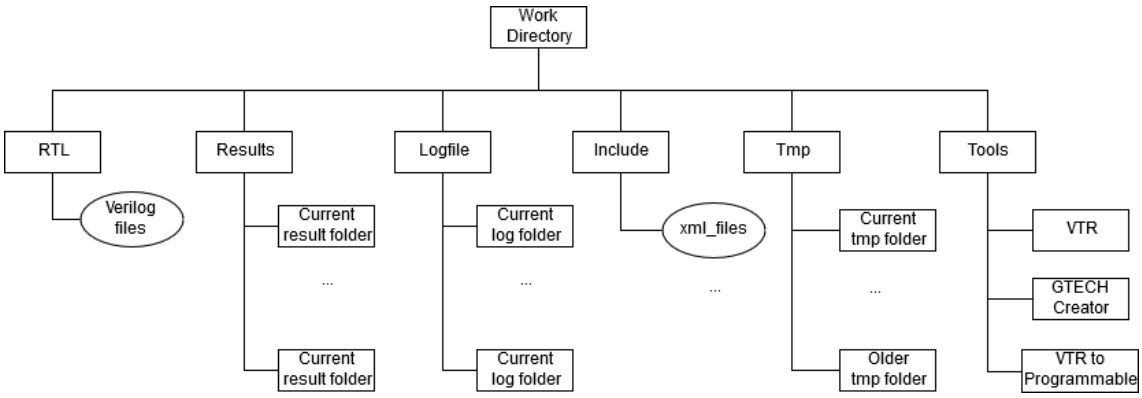

Figure D.1: Example Work Directory for the proposed Design Flow

- **Results** – This folder will contain the resulted files: Bitstream, RTL for the programmable device, and SDC file.
- **Logfile** – Logfiles generated by the tools run in this flow will be saved in this folder. Log files can be used for debugging.
- **Tools** – Scripts and specific tools used in this flow can be found in this folder.

D.2 Running the generation script

In this work, a script with the purpose of run this flow automatically was created. This tool can be run to achieve both objectives that are proposed: creating a programmable device, re-programming an already manufactured device.

To run this flow, one must follow the following syntax:

```
./runGen <verilog_file> <arch_xml_file> <options>
```

Using this script without options, the design flow will run to produce a programmable device and the bitstream for the `verilog_file`, but optimized with minimal programming size (minimal number of CLBs and tracks/channel).

Also, the designer may which to add additional logic to the produced device, at this time only the number of width and length of the FPGA fabric can be enlarged:

```
./runGen <verilog_file> <arch_xml_file> -more_logic <
   width_lengh_to_add > <number_tracks/channel_to_add >
```

To produce another bitstream for an already produced device, the script must run using the following command:

```
./runGen <verilog_file> <arch_xml_file> -reprogramming <number_clb>
   <number_tracks/channel>
```

D.3 Script description

As explained in 3 this script will be divided into four stages. In 3.2 can be found an overview block diagram of the script functionality.

D.3.1 Initial configuration

This script will first check if the input files are present in the correct folders. Moreover, the script will produce the output folders. The Verilog file will be the first argument and will be stored in the `verilog` variable, and the XML architecture file will be the second argument stored in the script as the `xml` variable.

```
if(! -e rtl_/$verilog ) then
      echo "Cant find verilog file"
      exit 1
endif

if (! -e include/$xml ) then
      echo "Cant find xml file"
      exit 1
endif

set folder_name=${verilog}_${xml}


if(! -e tmp/${folder_name}/ ) then
      mkdir tmp/${folder_name}/
      mkdir tmp/${folder_name}/vtr
else
      rm -rf tmp/${folder_name}/*
      mkdir tmp/${folder_name}/vtr
endif


if (! -e logfiles/${folder_name} ) then
      mkdir logfiles/${folder_name}
else
      rm -rf logfiles/${folder_name}/*
endif

if( -e results/${folder_name}/ ) then
      rm -rf results/${folder_name}/*
else
      mkdir results/${folder_name}
endif
```

After configuration, the script can pass to the pre-VTR phase.

D.3.2 Pre-VTR

In this phase, the design compiler will be run so that a GTECH netlist can be generated. This script will also terminate if the design compiler synthesis process fails.

```
#run Design Compiler
dc_shell -f gtech_synthesize.tcl | tee -i logfiles/${folder_name}/
   gtech_syn.logfile

set dc_errors = 'grep Error logfiles/${folder_name}/gtech_syn.
   logfile'
if( "$dc_errors" != "") then
      echo "Errors were found in DC"
      exit 1
endif
```

Finally, the modules that were instantiated in the synthesis process need to be appended to the GTECH produced a netlist, so the python GTECH parser script must be run using the following syntax:

```
Python gtech_parser.py <GTECH_verilog> <hierarchy_dc_report> <
   verilog_definition_all_modules>
```

Using this script, the `GTECH_verilog` file can now be employed in VTR flow.

D.3.3 Programmable device generation

The following steps will be different according to the decision of the designer.

VTR

To produce the device, the VTR flow will be run to produce the minimal number of CLBs and tracks/channel that can implement the intended circuit.

```
if($3 != "-reprogramming ") then
      tools/vtr_release/vtr_flow/scripts/run_vtr_flow.pl tmp/${
         folder_name}/${verilog}_gtech.v include/$xml_file -
         keep_intermediate_files -temp_dir tmp/${folder_name}/vtr -
         echo_file on -sweep_dangling_primary_ios off -
         sweep_dangling_nets off
```

This command will run the script that is provided by the VTR flow. Some arguments should be given to the flow process. The two first ones are the Verilog and the XML files respectively. The `-keep_intermediate_files` is used so that the flow does not remove any file produced in these tools, and the `-temp_dir` to change the place where this files will be saved, this argument is specific for the `run_vtr_flow.pl` script.

The following arguments are VPR tool-specific arguments, the `-echo_file` on argument will enable the tool to output some debug files that are going to be essential to the flow (like the

`rr_graph` file). The other arguments are used to prevent the VTR from removing dangling nets and IOs that were placed by the designer for later implementations.

Afterward, the script will collect the number of tracks/channel and the number of CLBs generated by the VTR flow, and add additional if required.

```
perl -nle 'print $1 if /mapped into a .*?(\d+)/' tmp/${
   folder_name}/vtr/vpr.out > tmp/${folder_name}/vpr_a_size
set size = "`cat tmp/${folder_name}/vpr_a_size`"

perl -nle 'print $1 if /width factor of .*?(\d+)/' tmp/${
   folder_name}/vtr/vpr.out > tmp/${folder_name}/vpr_width
set width = "`cat tmp/${folder_name}/vpr_a_width`"

if($3 == "-more_logic") then
      set size = "`expr ${size} + ${4}`"
      set width = "`expr ${width} + ${5}`"
endif

echo "Number of CLB : " $size
echo "Number of Track/Channel : " $width
```

The designer must keep these values so that, a new bitstream can be produced in the future.

Pos-VTR flow

In this step, the VPR will be run to generate the architecture after the flow had produced the minimal size. First, there is a need to copy an XML file that is equal to the last one with just the difference being on the `<layout>` tag from:

<layout auto="1.0"/>

To:

<layout width="XXX" height="XXX"/>

That will allow the flow to introduce the generated size, that will be used in the VPR tool.

```
cp include/${xml}_fixed.xml tmp/${folder_name}/vtr

sed -i -- 's/XXX/'"${size}"'/g' tmp/${folder_name}/vtr/${xml}
   _fixed.xml

cd tmp/${folder_name}/vtr/
```

```
tools/vtr_release/vtr_flow/vpr/vpr ${xml}_fixed.xml ${vtr}
   _gtech \
--sweep_dangling_nets off --echo_file on --
   sweep_dangling_primary_ios off --route_chan_width ${width}
```

The VPR tool is run with the same arguments as before. However, the `-router_chan_width` was added so that VPR can know the number of tracks. The final step in this phase is to run the `create_pads.py` tool so that the pads information can be generated for future implementations.

```
python tools/VTR_to_PD/create_pads.py tmp/${folder_name}/vtr/$
   {vtr}_gtech
```

D.3.4 Bitstream generation

The first thing is to check the number of CLBs and track/channel that was input by the designer

```
else

    set width = ${4}
    set size = ${5}
```

Then the designer must make sure that the pads and fixed XML file are present in the include folder.

```
cp include/${xml}_fixed.xml tmp/${folder_name}/vtr/
cp include/${vtr}_gtech.pads tmp/${folder_name}/vtr/

sed -i -- 's/XXX/'"${size}"'/g' tmp/${folder_name}/vtr/${xml}
   _fixed.xml
```

VTR

Then the VTR flow will run, however, the VPR tool will not be run now. The flow will stop after ABC so that only the BLIF netlist is created.

```
tools/vtr_release/vtr_flow/scripts/run_vtr_flow.pl tmp/${
   folder_name}/${vtr}_gtech.v tmp/${folder_name}/vtr/${xml}
   _fixed.xml -keep_intermediate_files -sweep_dangling_nets
   off -temp_dir tmp/${folder_name}/vtr -ending_stage abc
```

Pos-VTR flow

Now that the ABC file was created VPR will be run using some fixed options.

```
      tools/vtr_release/vtr_flow/vpr/vpr ${xml}_fixed.xml ${vtr}
         _gtech --echo_file on --sweep_dangling_primary_ios off --
         sweep_dangling_nets off --route_chan_width ${width} --
         fix_pins tmp/${folder_name}/vtr/${vtr}_gtech.pads
endif
```

To keep the IOs in the same position as previous implementations `-fix_pins` argument is used.

D.3.5 VTR to Programmable device generator

Finally, the program developed in this dissertation will be run:

```
tools/VTR_to_PD/VTR_to_PD ${vtr}_gtech ${xml}_fixed.xml --res_dir
   results/${folder_name} --tmp_dir tmp/${folder_name}/vtr/ --
   include_dir include/
```

This program has the following main arguments: the design name, and the XML file name. Optionally one can give:

1. `--res_dir` - Used to change the default results folder;
2. `--tmp_dir`– Modify the temporary folder;
3. `--include_dir` – Replace the include directory.

References

[1] Microsemi. Microsemi FPGA & SoC Products. http://www.microsemi.com/products/fpga-soc/fpga-and-soc. Accessed: 2017-06-25.

[2] Intel. Intel FPGA IP & Design Tools. http://www.intel.com/content/www/us/en/fpga/ip-and-design-tools.html. Accessed: 2017-06-25.

[3] S. J. E. Wilton and R. Saleh. Programmable logic IP cores in SoC design: opportunities and challenges. In *Proceedings of the IEEE 2001 Custom Integrated Circuits Conference (Cat. No.01CH37169)*, pages 63–66, 2001.

[4] Reinaldo A. Bergamaschi and John Cohn. The A to Z of SoCs. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '02, pages 790–798, New York, NY, USA, 2002. ACM.

[5] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for Deep-Submicron FPGAS*. Springer Science & Business Media, December 2012.

[6] I. Kuon and J. Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, February 2007.

[7] Jonathan Rose, Abbas El Gamal, and Alberto Sangiovanni-Vincentelli. "Architecture of Field-Programmable Gate Arrays". *Proceedings of the IEEE*, pages 1013–1029, 1993.

[8] Yao-Wen Chang, D. F. Wong, and C. K. Wong. Universal Switch Modules for FPGA Design. *ACM Trans. Des. Autom. Electron. Syst.*, 1(1):80–101, January 1996.

[9] Guy GF Lemieux, Stephen D. Brown, and Daniel Vranesic. On two-step routing for FPGAs. In *Proceedings of the 1997 International Symposium on Physical design*, pages 60–66. ACM, 1997.

[10] Steven JE Wilton. *Architectures and algorithms for field-programmable gate arrays with embedded memory*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, 1997.

[11] D. C. Forslund. The universal logic block (ULB) and its application to logic design. In *7th Annual Symposium on Switching and Automata Theory (swat 1966)*, pages 236–250, October 1966.

[12] Stephen S. Yau and Calvin K. Tang. Universal logic modules and their applications. *IEEE Transactions on Computers*, 100(2):141–149, 1970.

[13] J. Rose, R. J. Francis, D. Lewis, and P. Chow. Architecture of field-programmable gate arrays: the effect of logic block functionality on area efficiency. *IEEE Journal of Solid-State Circuits*, 25(5):1217–1225, October 1990.

[14] E. Ahmed and J. Rose. The effect of LUT and cluster size on deep-submicron FPGA performance and density. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):288–298, March 2004.

[15] Victor Olubunmi Aken'Ova. *Bridging the gap between soft and hard eFPGA design*. PhD thesis, University of British Columbia, December 2009.

[16] Jose Raul Garcia Ordaz and D. Koch. soft-NEON: A study on replacing the NEON engine of an ARM SoC with a reconfigurable fabric. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 229–230, July 2016.

[17] Menta. Embedded Programmable Logic. http://www.menta-efpga.com/overview.html. Accessed: 2017-06-25.

[18] Nanoxplore. eFPGA solution from Nanoexplore. http://www.nanoxplore.com/categories/17-efpga.html. Accessed: 2017-06-25.

[19] Pedro Miguel Ferreira Alves. Programmable flexible cores for SoC applications. Master's thesis, May 2013. FEUP MSc thesis.

[20] Juha P. Kylliäinen, Tapani Ahonen, and Jari Nurmi. General-Purpose Embedded Processor Cores – The COFFEE RISC Example. In Jari Nurmi, editor, *Processor Design*, pages 83–100. Springer Netherlands, 2007.

[21] S. Ahmad, V. Boppana, I. Ganusov, V. Kathail, V. Rajagopalan, and R. Wittig. A 16-nm Multiprocessing System-on-Chip Field-Programmable Gate Array Platform. *IEEE Micro*, 36(2):48–62, March 2016.

[22] Michael John Sebastian Smith. *Application-Specific Integrated Circuits*. Addison-Wesley Professional, 1 edition, 1997.

[23] Spartan-6 FPGA. https://www.xilinx.com/products/silicon-devices/fpga/spartan-6.html. Accessed: 2017-06-25.

[24] ISE Design Suite Tool. https://www.xilinx.com/products/design-tools/ise-design-suite.html. Accessed: 2017-06-25.

[25] Verilog to Routing. Open source tools for FPGA architecture and CAD research, November 2016. https://verilogtorouting.org/.

[26] Jason Luu, Jeff Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Norrudin Ahmed, Kenneth B. Kent, Jason Anderson, Jonathan Rose, and Vaughn Betz. VTR 7.0: Next Generation Architecture and CAD System for FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 7(2):6:1–6:30, June 2014.

[27] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon. Odin II - An Open-Source Verilog HDL Synthesis Tool for CAD Research. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 149–156, May 2010.

[28] ABC. ABC tool. https://people.eecs.berkeley.edu/~alanmi/abc/. Accessed: 2017-06-25.

[29] Vaughn Betz and Jonathan Rose. Vpr: A new packing, placement and routing tool for fpga research. In *International Workshop on Field Programmable Logic and Applications*, pages 213–222. Springer, 1997.

[30] Jason Luu, Jason Helge Anderson, and Jonathan Scott Rose. Architecture Description and Packing for Logic Blocks with Hierarchy, Modes and Complex Interconnect. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 227–236, New York, NY, USA, 2011. ACM.

[31] Vaughn Betz and Jonathan Rose. Automatic Generation of FPGA Routing Architectures from High-level Descriptions. In *Proceedings of the 2000 ACM/SIGDA Eighth International Symposium on Field Programmable Gate Arrays*, pages 175–184, New York, NY, USA, 2000. ACM.

[32] Scott Hauck, Katherine Compton, Ken Eguro, Mark Holland, Shawn Phillips, and Akshay Sharma. Totem: domain-specific reconfigurable logic. In *IEEE Transactions on VLSI Systems*, pages 1–25, 2006.

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

[34] H. Mrabet, Husain Parvez, Z. Marrakchi, H. Mehrez, and André Tissot. Automatic layout generator of domain specific FPGA. In *2008 International Conference on Microelectronics*, pages 183–186, 2008.

[35] Chaofan Yu, Lingli Wang, and Xuegong Zhou. Automatic layout generator for embedded FPGA cores. In *2011 9th IEEE International Conference on ASIC*, pages 385–388, 2011.

[36] A. Brant and G. G. F. Lemieux. ZUMA: An Open FPGA Overlay Architecture. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 93–96, April 2012.

[37] V. Aken'Ova and R. Saleh. A "soft++" eFPGA physical design approach with case studies in 180nm and 90nm. In *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*, March 2006.

[38] Jin Hee Kim and Jason H. Anderson. Synthesizable Standard Cell FPGA Fabrics Targetable by the Verilog-to-Routing CAD Flow. *ACM Trans. Reconfigurable Technol. Syst.*, 10(2):11:1–11:23, April 2017.

[39] Synopsys. *Design Compiler User Guide*, 7 2017.

[40] Eddie Hung, Fatemeh Eslami, and Steven JE Wilton. Escaping the academic sandbox: Realizing VPR circuits on Xilinx devices. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 45–52. IEEE, 2013.

[41] Clifford Wolf. Yosys open synthesis suite. http://www.clifford.at/yosys/.

[42] Clifford Wolf. Yosys Application Note 010: Converting Verilog to BLIF. 2013.

[43] Saeyang Yang. *Logic synthesis and optimization benchmarks user guide: version 3.0.* Microelectronics Center of North Carolina (MCNC), 1991.

[44] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, October 1974.

[45] M. Bohr. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.