POLITECNICO DI TORINO

Collegio di Ingegneria Elettronica, delle Telecomunicazioni e Fisica

Corso di Laurea Magistrale
in Ingegneria Elettronica

Tesi di Laurea Magistrale

A Reconfigurable Device for GALS Systems

Relatore

prof. Luciano Lavagno

Candidato

Rocco Sciaraffa 231018

Luglio 2018

Abstract

Globally Asynchronous Locally Synchronous (GALS) Field-Programmable Gate Array (FPGA) are composed of standard synchronous reconfigurable logic islands that communicate with each other via an asynchronous means. Past research into fully asynchronous FPGA has demonstrated high throughput and reliability adopting dual-rail encoding. GALS FPGAs have been proposed, relying on bundled-data encoding and fixed asynchronous communication between synchronous islands. This thesis proposes a new GALS FPGA architecture with fully reconfigurable asynchronous fabric, that relies on coarse-grained Configurable Logic Blocks (CLBs) to improve the communication capability of the device. Through datapath dedicated elements, asynchronous pipelines are efficiently mapped onto the device. The architecture is presented as well as the customized tool flow needed to compile Verilog for this new coarse-grained reconfigurable circuit.

The main purpose of this thesis is to map communication-purpose user-circuits on the proposed asynchronous fabric and evaluate their performance. The benchmark circuits target the design of a Network-on-Chip (NoC) router and employ two-phase bundled-data protocol. The results are obtained through simulation and compared with the performances of the same circuits on a fine-grained classical FPGA style. The proposed architecture achieves up to 3.2x higher throughput and 2.9x lower latency than the classical one. The results show that the coarse-grained style efficiently maps asynchronous communication circuits, and it may be the starting point for future reconfigurable GALS systems. Future work should focus on improving the back-end synthesis and evaluating the FPGA GALS system as a whole.

Keywords **-** FPGA, GALS, asynchronous, coarse-grained, NoC, bundled-data

Acknowledgements

I would first like to thank my supervisor at NII, professor Tomohiro Yoneda, for his useful ideas and guidance during the whole internship period in Tokyo. I moreover express my gratitude for his further suggestions and advice that helped me to complete this thesis.

I am grateful to my examiners Johnny Öberg and Luciano Lavagno for their valuable feedback.

Special thanks to Marco for his hospitality during my stay in Stockholm.

Last but not least, I would like to thank my family: thanks to my sister for her helpful comments and proof-reading my thesis, thanks to my parents for unconditionally supporting all my studies and life choices.

Rocco Sciaraffa

List of Figures

List of Tables

List of Acronyms

ASIC Application Specific Integrated Circuit.

BLE Basic Logic Element.

BLIF Berkeley Logic Interchange Format.

CAD Computer-Aided Design.

CCB Control Circuit Block.

CLB Configurable Logic Block.

CNF Conjunctive Normal Form.

DPB Data Path Block.

DSP Digital Signal Processor.

EMC Electromagnetic Compatibility.

FIFO First In First Out.

FIR Finite Impulse Response.

FPGA Field-Programmable Gate Array.

FSM Finite State Machine.

GALS Globally Asynchronous Locally Synchronous.

GAPLA GALS Programmable Logic Array.

HDL Hardware Description Language.

I/O input/output.

IP Intellectual Property.

ITRS International Technology Roadmap for Semiconductor.

LE Logic Element.

LUT Look-Up Table.

MTBF Mean Time Between Failure.

mutex mutual-exclusion element.

NII National Institute of Informatics.

NoC Network-on-Chip.

NRE Non-recurring engineering.

PCHB precharge half-buffer.

PSB Pipeline Stage Block.

QDI Quasi delay-insensitive.

RTL Register Transfer Level.

SAT Boolean Satisfiability problem.

SB Switch Block.

SBC Switch Block Control.

SBD Switch Block Data.

SDF Standard Delay Format.

SLE Special Logic Element.

SoC system-on-a-chip.

VPR Versatile Place and Route.

VTR Verilog-to-Routing.

Contents

Chapter 1

Introduction

FPGAs are widely used and accepted nowadays not only as prototyping devices but also as powerful tools for advanced applications. As well as non-programmable circuits in the current late-Moore era, FPGAs are facing problems like increased process variability, power and thermal bottlenecks, high fault rates, scalability [1]. Clock signals, in particular, suffer from these problems: variability in clock skew severely limits performance, and the clock signal itself causes a vast part of dynamic power consumption. Asynchronous design style could solve the mentioned issues.

This chapter is an overview of the thesis, and it is organized as follows. Section 1.1 briefly describes the thesis background. Section 1.2 introduces the on-going project in which this master research is involved. Section 1.3 presents the main purpose of the thesis. Section 1.4 details the measurable goals of the thesis. Section 1.5 underlines the limits of the selected research approach. Finally, Section 1.6 outlines the entirety of the thesis.

1.1 Background

FPGA systems are being used in a variety of applications thanks to their reconfigurability benefit. Initial drawbacks such as very low efficiency and a limited number of logic cells have been mostly mitigated [2][3]. State-of-the-art FPGAs contain millions of logic cells (e.g., Virtex-7 by Xilinx has up to 2M logic cells). Currently, FPGA systems are used for advanced applications, an example is Microsoft Catapult project, where FPGAs are connected to servers in a data center for sup-

porting inter-node communication [4]. Another field of use for FPGAs nowadays is Deep Learning accelerators [5].

However, FPGAs suffer from scalability problems, mainly due to timing issues. In fact, timing constraints become very strict for larger FPGAs as it is difficult to distribute clock and sometimes it is not possible to reach every part of the chip in a single clock cycle [6][7][8]. One solution could be to introduce multiple clock regions, but interfaces between them would be troublesome. Another primary concern of current FPGAs architecture is power consumption. Power dissipated just by the clock signal can be relatively high, up to 22% of the total dynamic power [9].

This being said, it is clear that avoiding the use of a global clock would lead to faster and more efficient circuits. It would be convenient to build what is called a reconfigurable GALS system, that is a circuit formed by medium-small synchronous cores communicating with each other through an asynchronous reconfigurable circuit. This asynchronous system replaces the global clock using local handshaking, in particular, targeting single-rail bundled-data handshaking protocol.

1.2 Problem

This master research is part of a broader ongoing project performed at National Institute of Informatics (NII), that is the design of a fully reconfigurable circuit for GALS systems. Previously, so-called GALS FPGAs [10][11] were proposed. These works, however, include no reconfigurability in asynchronous communication channels, and thus, their applications are limited.

The proposed circuit addresses fully reconfigurable GALS systems, i.e., also the asynchronous communication channels can be programmed. The asynchronous fabric is specialized for communication mechanism for GALS (handshake, NoC routers, switches). It does not include data processing elements, but it is focused more on communication, so it makes use of word registers and user-controlled multiplexers. For achieving high efficiency in bundled-data protocol, the proposed FPGA architecture has been constructed in a coarse-grained manner, with basic logic elements 8 to 10 times larger than CLB of a fine-grained classical architecture.

Basic Logic Elements (BLEs) of the proposed architecture have already been developed. The main problem to be solved is to develop the place & route algorithm that would map a user circuit to the proposed coarse-grained architecture. The mapping problem in a coarse-grained architecture is tough due to its complicated structure. Moreover, there are many constraints in the connection between elements. Therefore, the mapping problem is not yet solved, and a semi-automated tool flow has been developed for early performance evaluation of the proposed method.

1.3 Purpose

As part of the ongoing work previously mentioned, the purpose of this master research is to contribute to the improvement of the semi-automated placement tool and to evaluate the performance of the proposed device in a particular case.

More in detail, several medium-complex communication-purpose user-circuits (i.e., First In First Out (FIFO), crossbar switch, router) have been mapped onto the proposed architecture using the developed semi-automated tool, with the aim to carry out the initial performance evaluation. Also, a pure combinational circuit (ripple-carry adder) has been evaluated for checking the performance of the proposed architecture out of the intended field of use. Simulation results are then compared with a fine-grained asynchronous FPGA that exploits an established tool flow for FPGAs (Verilog-to-Routing (VTR) [12]).

1.4 Goals

The measurable goal of this Master project is the result of the comparison between conventional fine-grained architecture FPGA and coarse-grained one, focusing on communication-purpose user circuit. The comparison establishes whether the proposed architecture is more efficient regarding area and performance. If so, the project would be a starting point for the development of a new efficient reconfigurable device for GALS systems.

The research follows a quantitative approach, as the final result achieved is a comparison of several measurable performance param-

eters. A simulation approach has to be carried out since the FPGA architecture is still in developing status. As mentioned before, for the fine-grained architecture VTR tool for placement and routing has been used. For the proposed coarse-grained architecture, instead, a semi-automated tool for placement and routing has been developed. This means that part of the placement has been conducted manually, and then the developed tool has been used to complete the mapping.

1.5 Limitations

This Master's thesis examines the performance of a coarse-grained asynchronous FPGA specialized for communication mechanism for GALS systems. The evaluation is then mainly focused on communication purpose circuit (FIFO, crossbar switch, router), plus a case of a simple, pure combinational circuit (ripple carry adder). The research does not take into consideration more complex data-processing user circuit.

Moreover, the design of the asynchronous user circuits follows the single-rail bundled-data asynchronous protocol [13]. Other asynchronous design styles have not been examined. Finally, as mentioned, the proposed architecture has not been tested running on actual hardware. Given the early status of development, the evaluation has been carried out by simulation.

1.6 Structure of the Thesis

The first section of Chapter 2 provides basic knowledge about FPGAs. It offers a brief introduction to FPGA architecture, from the logic elements design to the routing structure, and a description of the most common commercial and academic design flow for FPGA. Chapter 2 continues by focusing on asynchronous design. It begins by listing pros and cons of this design style, followed by an overview of asynchronous techniques and communication protocols. Subsequently, GALS systems are also mentioned, describing the possible interfaces between different timing domains. Once the basics are given, Chapter 2 ends by presenting related work in the field of asynchronous reconfigurable circuits and GALS FPGA.

In Chapter 3 the proposed coarse-grained architecture and the tool flow are presented.

In Chapter 4 the architecture evaluation is carried out. First, the fine-grained architecture is presented, with which the proposed one is compared. Then, the simulation results are given and discussed.

Finally, Chapter 5 sums up the thesis conclusions and suggests improvements and topics for future work.

Chapter 2

Background

This chapter provides the reader basic knowledge needed to familiarize with reconfigurable circuits and asynchronous design. Section 2.1 defines what an FPGA is, describing logic blocks and routing architecture, as well as the standard tool flow employed for synthesis and mapping of user circuits on a reconfigurable device. Section 2.2 presents the asynchronous design, the benefits that it can achieve and several foundational techniques of this approach. In particular, the GALS application of asynchronous circuit will be analyzed more in detail. Lastly, Section 2.3 reviews a selection of asynchronous reconfigurable architectures presented in the past two decades.

2.1 FPGA

An FPGA is a silicon device that can be programmable to behave such as almost any digital circuit. FPGAs provide some interesting advantages over the fixed-function counterpart, Application Specific Integrated Circuits (ASICs), such as reduced Non-recurring engineering (NRE) cost and shorter time-to-market [14]. In fact, ASICs are usually extremely expensive to fabricate, and their design and validation process takes months [15].

Nevertheless, FPGAs have a number of disadvantages that still prevent them from replacing ASICs entirely. Area overhead is still severe, being the occupied area from an FPGA circa 20 to 35 times larger respect to a standard cell ASIC, as well as delays (approximately 3 to 4 times slower) and dynamic power (roughly ten times more) [14].

Since the first produced FPGA by Xilinx in 1984, this kind of device

increased in capacity and speed, reducing cost and energy consumption [2][3]. At the same time, as Moore's law [16] advances, the cost for state-of-the-art lithographic process and NRE increase at each technology node, making ASICs even more expensive. Nowadays, only a few chips have a market volume large enough to cover these costs [3]. In FPGA market development costs are shared with all the customers, so that unit cost mostly drives the final price.

Currently, more and more digital designs are being implemented using FPGAs. FPGAs are used in advanced applications such as datacenter services accelerators [4], or Deep Learning accelerators [17]. These applications, in particular, are evolving very fast. They need to rely on flexible hardware which performance and efficiency can be continuously be improved, so FPGAs tend to be an optimal solution.

2.1.1 Architecture

An FPGA chip basically consists in an array of logic blocks (called also CLBs) connected between them by a programmable routing fabric. The ability to reconfigure an FPGA comes from both the possibility of programming the CLB and the interconnect. In modern FPGA the programming technologies mainly used are flash [18], static memory [19], and anti-fuses [20].

A typical island-style FPGA, that is the most popular architectural style used for commercial FPGAs, as well as among research community, is depicted in Figure 2.1.

CLBs can be of a various type. Originally, they consisted of just programmable logic (mainly Look-Up Tables (LUTs)) and storage elements (flip-flops or latches) [19], that formed what is called a BLE. As technology scaling advances it was possible to enhance the functions of each CLB. Current commercial FPGAs can contain heterogeneous dedicated CLBs, such as arithmetic logic blocks, large memories or microprocessors [21].

The routing fabric is typically composed of connection blocks and switch blocks. A connection block connects a CLB inputs and outputs to the wires in the routing fabric. A switch block (Figure 2.2) connects adjacent wire segments, with the possibility to program the path using logic buffers or pass-transistors. Different connection patterns between adjacent channels are possible, namely disjoint, Wilton and universal. Their pros and cons are described in [22]. Finally, there are

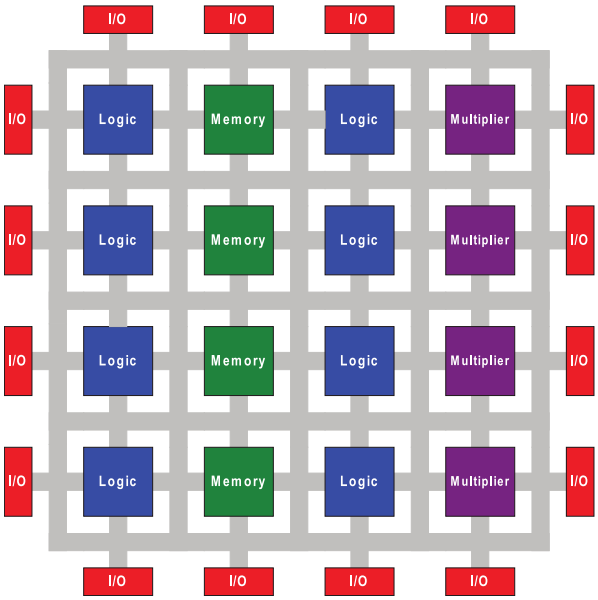


Figure 2.1: Island-style FPGA.

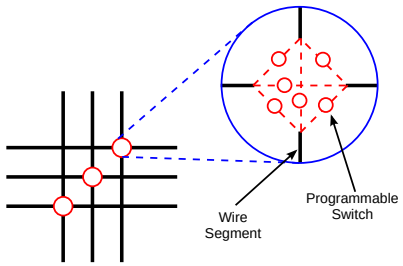


Figure 2.2: Typical switch box.

input/output (I/O) blocks around the edges of the FPGA providing programmable I/O for various standards.

Additionally, in modern FPGAs clock distribution network is an essential part of the chip. Like the ASIC counterpart, the most used clock distribution network is H-tree style [23]. This clock distribution, however, costs power and area, then attempts of using routable clock networks have been made [24], such that fixed clock trees can be replaced by routable clock grid that allows constructing arbitrarily sized clock trees to in arbitrary locations of the FPGA.

When it comes to defining FPGAs architecture, a critical feature to take into account is the area. In fact, the area occupied by a die on the silicon wafer is what determines the fabrication cost of the device, and as previously stated, the low cost is the main advantage for FPGAs

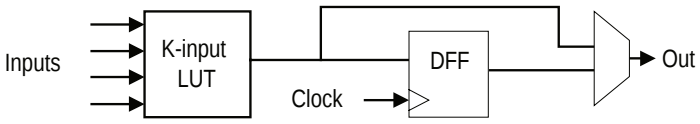


Figure 2.3: Structure of a BLE.

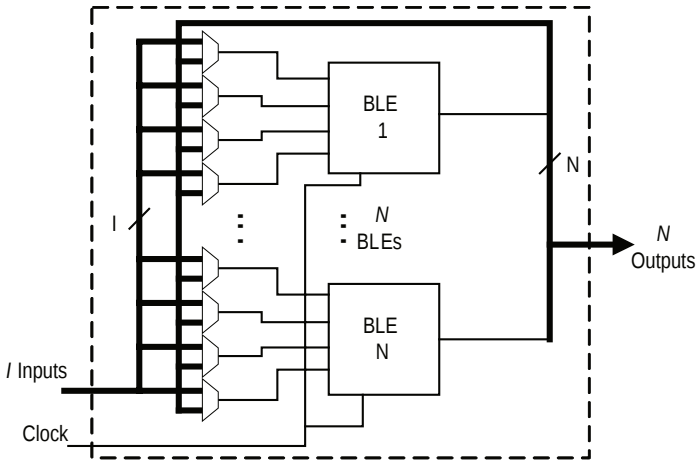


Figure 2.4: Logic cluster assuming a LUT size (K) of 4.

over ASICs.

Chosen the architecture framework (typically island-style) and the programming technologies, logic blocks mainly dictate the area of the chip, especially for devices with a large logic capacity. Then, the design of the architecture of the CLBs is crucial to reduce the chip area. The typical structure of a LUT-based BLE is represented in Figure 2.3, as well as an example of a logic cluster in Figure 2.4 [25]. Directly mapped functions in LUTs are particularly area-inefficient since the area occupied by LUTs grows exponentially with the number of inputs. A solution known as clustering is used instead. Here, *N* BLEs using LUT/flip-flop are used grouped into a cluster.

2.1.2 Design Flow

During the early days, until the mid-90s, FPGAs were so small that the automated placement and routing were considered superfluous. Manual design was used instead, for both logical and physical design [3]. As FPGA capacity grew in line with Moore's law, manual design was not feasible anymore. At this point, the late 90s, automated synthesis, placement and routing were essential. FPGA vendors nowadays

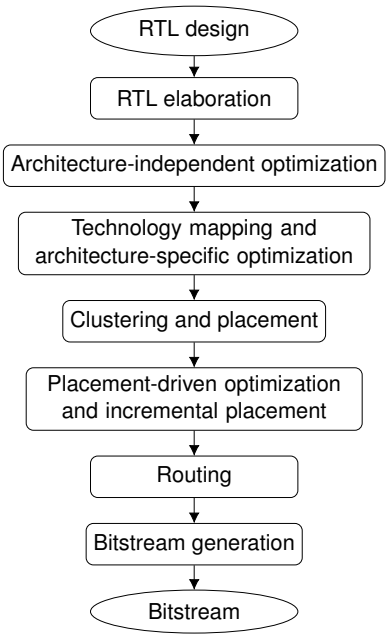


Figure 2.5: Commercial tool flow.

provide commercial Computer-Aided Design (CAD) tools for their devices that starting from a design specified in hardware description languages (VHDL or Verilog) produce the bit-stream needed to program the chip.

A typical design flow is shown in Figure 2.5 [26]. The inputs of the design flow are the Register Transfer Level (RTL) description, the design constraints, and the FPGA device. The specification is usually given in Verilog or VHDL even though there is a general trend moving to higher level of abstraction using languages like C or SystemC [27]. The design constraints include required I/O delays and clock frequency. The choice of an FPGA architecture depends on the capacity, performance, cost, and power. The correct FPGA device is chosen in an iterative process: the smallest FPGA is selected, then if it is not possible to map the user application on this device a higher capacity one is needed.

As concerning the tool-flow steps of Figure 2.5, several steps behave as follow. The RTL elaboration analyzes the datapath and maps all the element in Finite State Machines (FSMs), generic Boolean logic or hard blocks that are eventually available in the chosen architecture (i.e., multiplier, carry chains, synchronizer). A net containing only this

set of elements is built. The second step is a general optimization, including combinatorial and control logic optimization. At this point, the optimized datapath is mapped on the chosen architecture, choosing dedicated structures if possible, otherwise general BLEs. After the first placement is obtained, exploiting clustering eventually, interconnects are defined, and there may be an optimization of the placement if it does not meet the performance required. This step is recursive consists of defining and validating the placement incrementally. There are many placement options for FPGAs [28]. Exploiting the programmable switches, routing is performed to connect all the signals as designed. Numerous routing approaches for FPGAs are summarized in [2]. Last step, a bit stream necessary to program logic and interconnects is generated.

The described tool flow is a typical example of commercial CAD tool that may be provided by the vendor. This approach works perfectly for a specific target device, but in the research community, there are significant efforts to explore new architectures and enhanced algorithms for CAD tools. For this purpose, an open-source tool that can be easily modified and adapted is required. The most popular open-source CAD tool for FPGA is VTR that allows to model and target hypothetical FPGA devices [12]. The overall tool flow, in this case, is described in Figure 2.6.

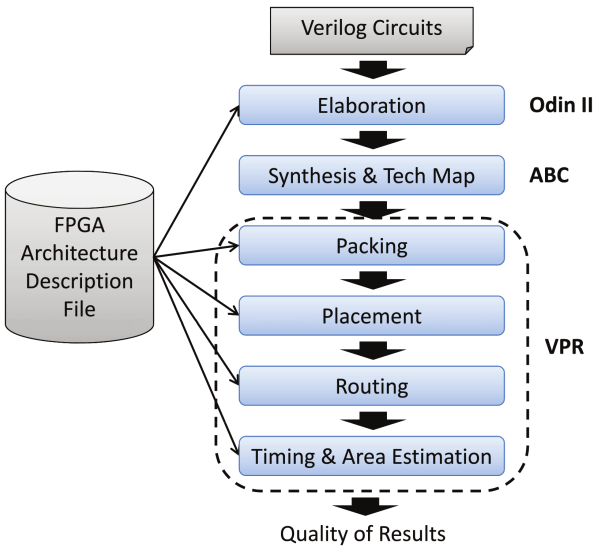


Figure 2.6: VTR tool flow.

Similarly to the previous case, the inputs are a user-circuit described

in Verilog and the architectural description file of the hypothetical FPGA [29]. The analysis of the RTL design is done by ODIN [30], that translates the Verilog code into a Berkeley Logic Interchange Format (BLIF) file [31], that is a netlist of primitives, according to the resources present in the description file. ABC [32] then optimizes the logic, also mapping the soft logic in LUTs of the size reported in the description file. Finally, Versatile Place and Route (VPR) [33] performs clustering, placement and routing, also providing an estimation of the quality of the result in terms of timing, area and power.

2.2 Asynchronous design

Most of the digital circuits for almost every application are nowadays synchronous. This technology relies on a clock signal that synchronizes all the operations on the circuit. Asynchronous architectures, conversely, do not have a global notion of time along the circuit. Operations, in this case, are much more complicated because signals must meet specific timing constraints. The communication protocol between different elements in the circuit in an asynchronous domain is called handshake, that exploits two additional signals to provide operations synchronization usually called *request* and *acknowledge* [34].

Over the last two decades, new challenges arose as Moore's law comes to its end [35]: increased variability, power consuming and dissipation, high fault rates [1][36]. These three primary challenges, highlighted in the International Technology Roadmap for Semiconductor (ITRS) reports [1], could be tackled using an asynchronous approach in the design flow.

Some interesting example of commercial asynchronous devices have been developed, such as [37] for communications, and an example of neuromorphic computer from IBM [38]. Some other industrial experiments, also, have been carried out, like IBM Finite Impulse Response (FIR) [39]. Finally, emerging researches about the asynchronous field are ultra-low energy systems [40][41], asynchronous Digital Signal Processor (DSP) [42], extreme environments applications [43] and nanomagnetic logic [44].

2.2.1 Pros and Cons

Synchronous design methodology is currently the most used. Indeed, it has some remarkable advantages. The great benefit of this approach is that the design is straightforward and easy to validate since the signals are usually sampled on a particular clock event thus are not required to be correct all the time. As far as flip-flop set-up and hold time are observed, the circuit is guaranteed to work. Engineers are very well trained to build synchronous circuits and very advanced and complex CAD tools have been developed for synchronous design. Nevertheless, synchronous design has some limits that can be overcome with an asynchronous approach.

Asynchronous circuits do not need to distribute the clock signal along the circuit. This property is a great benefit since at each technology node the clock distribution network becomes more complex and critical due to the process variation [45]. Using clock signal is also not very efficient energy-wise, since it consumes quite a significant fraction of the total dynamic power, up to 22% [9]. The computation in the case of asynchronous circuits is event-driven, i.e., operations only execute when needed, that is highly energy efficient, and there is no need to insert extra clock-gating elements [46]. Moreover, performance in the case of synchronous design is limited by the worst-case delay between each pipeline stage, while for asynchronous circuits each operation is executed at the maximum speed. The fact that signals do not commute at the same instant also have benefits on the Electromagnetic Compatibility (EMC) side [47], resulting in less noisy and reliable system. Finally, asynchronous design can easily accommodate heterogeneous system timing of system-on-a-chip (SoC) architectures.

On the other hand, asynchronous approach poses some other problems. The main one is that the vast majority of CAD and validation tools addresses synchronous design only. There is a lack of tool-flow and CAD systems for asynchronous circuits. Designers have to develop brand new tools or adapt the existing synchronous ones. Few experiments in the former direction have been carried out; more effort was put last two decades trying to exploit synchronous tools, like [48] and [49], that try to automatically transform a synchronous design into an asynchronous one. Using this approach, furthermore, there is no need to retrain designers on asynchronous circuits as the transformation is done automatically. As hybrid architectures emerge, like GALS,

another issue is the synchronization between different timing domain elements.

2.2.2 Basics

As previously mentioned (Section 2.2), handshaking channels are fundamental in asynchronous communications between elements along the circuit. Different implementations are possible, defining different protocols and data encoding. Other fundamental building blocks for more complex asynchronous systems are asynchronous pipelines, synchronization, and arbitration blocks.

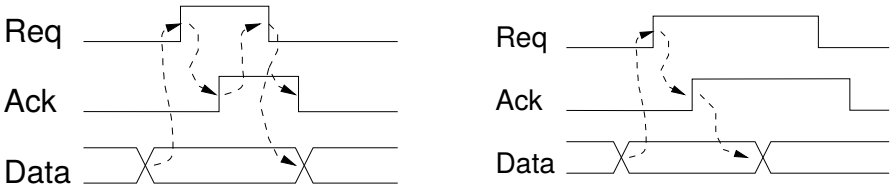
Handshake

Two signals of *request* and *acknowledge* (called *req* and *ack* from now on) are required to guarantee the correctness of the handshake communication. Two main communication protocols are used, *4-phase* (*return-to-zero*) and *2-phase* (*non-return-to-zero* or *transition-signaling*).

In the *4-phase* protocol, shown in Figure 2.7a, *req* and *ack* wires codify the information in usual Boolean levels. For starting the transmission, the sender asserts *req* and waits for the receiver reply asserting *ack*. At this point sender deasserts *req* that causes the receiver to deassert *ack*, bringing the signals back to the initial state (from this the name *return-to-zero*). On the other hand, in the *2-phase* protocol, shown in Figure 2.7b, the information is codified on wires toggling: whenever *req* toggles a request is sent, same for *ack*. Both these protocols are very common among designers. *4-phase* protocol requires simpler hardware because it relies on signals Boolean levels, nevertheless the return-to-zero phase is not essential and makes the transaction less efficient in terms of throughput and energy. *2-phase* protocol only requires one round-trip communication, thus is generally more efficient, even though more complex hardware may be needed [50].

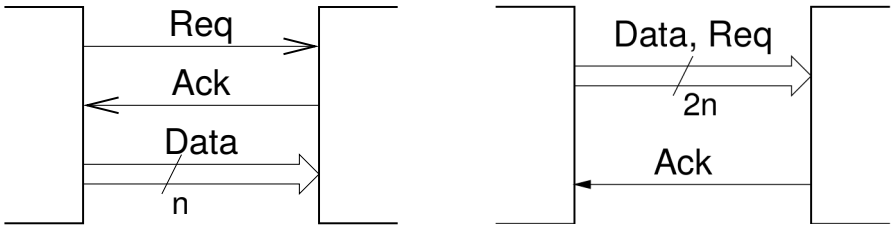
For what concerns data encoding, two main schemes are used: *single-rail* or *dual-rail*.

The *single-rail* approach is also called *bundled-data*. The term refers to the fact that *req* and *ack* wires are bundled with the correspondent data (Figure 2.8a). In *single-rail* encoding, data are encoded with usual binary levels. *Req* and *ack* extra wires are required. The request must be transmitted after the data are valid on the channel and this order must be preserved at the receiver. For ensuring this, a matched delay



(a) 4-phase protocol. (b) 2-phase protocol.

Figure 2.7: Handshake communication protocols.



(a) Bundled data channel. (b) Dual-rail data channel.

Figure 2.8: Handshake data encoding schemes.

is added to the *req* signal, such that its delay is longer than the worst-case data delay. The matched delay can be either a chain of inverters or a replication of the critical path. The advantage of this encoding is that the design is very similar to the synchronous counterpart and has low area overhead, even though the matched delay is sensitive to process variation hence it must be added with appropriate safety margin [34].

In *dual-rail* (also called *delay-insensitive*) encoding, data are sent using a couple of wires for each bit (Figure 2.8b). One wire is asserted when the data to be sent is '1', vice versa the other wire. The couple {0,0} means no data valid and {1,1} is not used. This encoding provides an implicit request when all data are valid, condition easily recognizable at the receiver. The great advantage of this technique is that it is very robust and insensitive to process or physical variations. The disadvantage consists of using a couple of wires for transmitting one bit. Moreover, a completion detector must be used at the receiver's side. In general, only 4-phase protocol works for this encoding scheme.

Pipelining

In asynchronous systems, as well as in the synchronous case, pipelining is an essential technique used to increase throughput parallelizing the elaboration flow. In the synchronous implementation, it usually

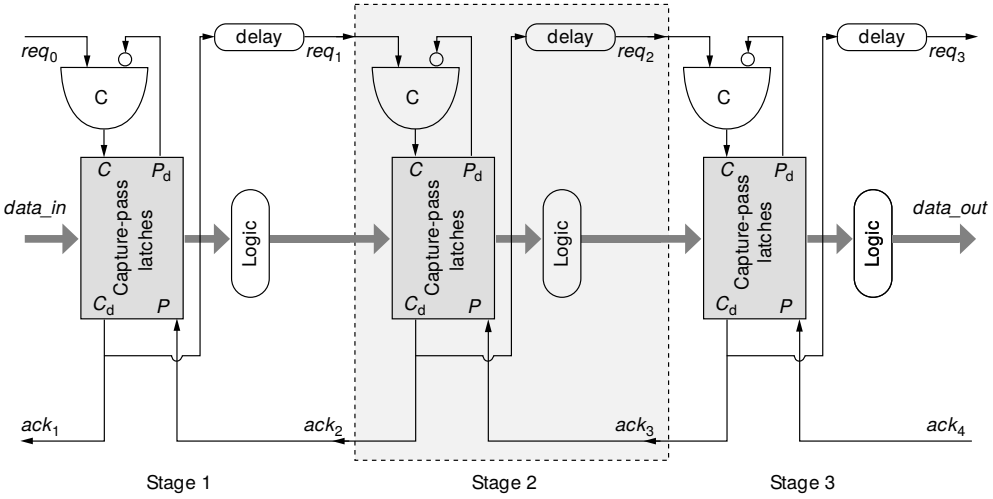


Figure 2.9: Sutherland micropipeline.

reckons on registers that break the data path in several stages, with the clock signal behaving as a pacemaker. Asynchronous systems do not have a global clock, so handshaking communication between adjacent stages is used instead.

A possible asynchronous pipeline implementation was proposed by Sutherland [51] depicted in Figure 2.9. It is a *2-phase* bundled-data pipeline that exploits a chain of Muller C-elements. The C-element is a ubiquitous storage component in the asynchronous design, which output is driven by inputs only if they have the same value, else it holds the previous value. The circuit makes use of so-called *capture-pass* latch: this specialized latch has two control inputs (*capture C* and *pass P*) and two control outputs (*capture done* C_d and *pass done* P_d). At the initial condition all the latches are transparent. As soon as an event occurs on its *capture* input, a latch turns in the hold state, and it stays in hold until an event occurs on the *pass* input. Request propagates along the pipeline in a "wave front" fashion: it advances forward performing a series of *capture* operations, while predecessor stages, behind the wave front, are afterward freed up by a series of *pass* operations. This implementation is straightforward and elegant, even though it uses specific elements (*capture-pass* latch, C-element) that are not area efficient. Nevertheless, the idea had inspired many similar approaches.

Another *2-phase* bundled-data pipeline is Mousetrap pipeline, shown in Figure 2.10 [52]. It is based on the same idea of the Sutherland micropipeline, but it uses common gates and data latches instead of spe-

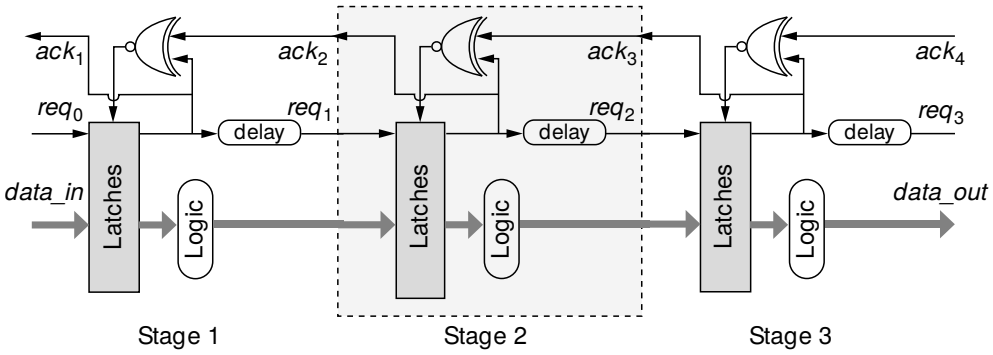


Figure 2.10: Mousetrap pipeline.

cialized asynchronous elements. This approach shows better results in terms of area and delay [1]. Initially all the data latches are transparent. At stage *i*, when a request arrives from stage *i-1*, it makes the XNOR low, thus turning the latch in the hold state. At the same time, the request propagates to the next stage and toggles the acknowledge to the previous stage, requiring new data. Eventually, stage *i* receives an acknowledge from *i+1*, that makes the XNOR high turning the latch transparent again.

Another asynchronous pipeline style is based on dynamic logic data paths and dual-rail encoding. It is mainly used for high-performance systems, very robust but expensive regarding design effort and validation [13]. One possible implementation is the precharge half-buffer (PCHB) style [53]. This approach relies on Quasi delay-insensitive (QDI) circuits, i.e., operation correctness does not depend on gates and wires delay, as far as delays on all the branches of a wire fork are roughly equal (*isochronic fork* assumption). A PCHB circuit is shown in Figure 2.11 [54]. There are two completion detectors, at the input and the output of the function. The evaluation at the stage *i* only starts when valid input data are available. After the function evaluation has been carried out, an acknowledge is sent to the previous stage *i-1* to make it start the pre-charge phase. An acknowledge from the next stage *i+1*, similarly, means that the output data has been consumed, and the stage *i* can be pre-charged. When the pre-charged is completed each stage turns back to evaluate phase.

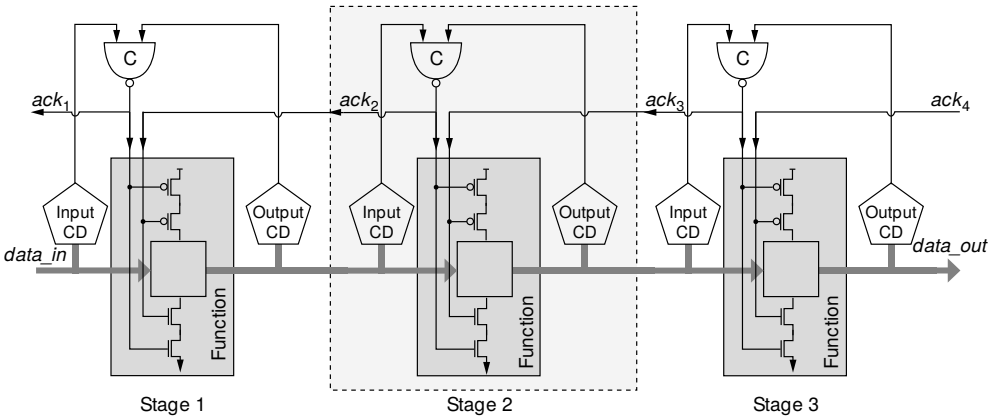


Figure 2.11: A PCHB pipeline.

Synchronization and arbitration

Synchronizers are needed at the interfaces between different timing systems, asynchronous-synchronous as well as two unrelated synchronous systems. In these interfaces *setup* and *hold* timing may be violated, causing a metastable output that could be undetermined for an extended amount of time. The classical and most natural solution is to use for each bit a chain of flip-flops as a synchronizer. The number of flip-flops depends on the required Mean Time Between Failure (MTBF), typically two or three flip-flops ensure sufficiently high MTBF. This simple synchronizer leads to low area overhead but also low throughput. Another approach to interface two arbitrary clock domains is using specially designed asynchronous FIFO buffers [55][56].

Finally, arbitration is needed for the resolution of multiple requests of the same resource. In synchronous systems, this is not a critical issue, as at each clock cycle one pending request is selected as the winner. On the other hand, in asynchronous design requests arrive as continuous time signals, so there is need of an element that guarantees a winner when requests come very close to each other. The basic block to resolve two-way arbitration is called mutual-exclusion element (mutex). It is an analog component that in principle may require in some cases infinite time to resolve, in practice long arbitration is extremely rare. It is the building block for n-way arbiters used.

2.2.3 GALS systems

As an alternative to fully-asynchronous designs, hybrid approaches have been developed, that combine synchronous components and asynchronous communication to form a GALS system. This approach gains the benefits of asynchronous and synchronous design. Synchronous modules can be designed following the standard synchronous flow, or Intellectual Property (IP) blocks can be easily integrated into the asynchronous interconnect. Several blocks may require different clock requirements that can be easily accommodated in a GALS system, either to make them work for maximum performance or low power [57]. As a global asynchronous system, moreover, no global clock is needed, avoiding clock skew and distribution issues.

Each synchronous island is locally clocked and connected to the asynchronous network through a synchronous/asynchronous interface. The interface must be designed to guarantee that the communication succeeds without causing metastability. The way the interface implements the data transfer between different timing domains determines the GALS design style. Generally, three main different design styles are possible [58][57]: pausible-clock, asynchronous and loosely synchronous.

The *pausible-clock* (also called *clock-stretching*) was the first approach proposed for a GALS system [59]. A local clock is generated in each synchronous block using a ring oscillator [60]. Whenever a communication is needed at the interface, the rising edge of the local clock is delayed. A handshake communication is then performed and when data are safely latched the local clock can run again. Port controllers are needed to perform the handshake communication. On the one hand, this approach is very robust and energy efficient [61]. On the other hand, the design of the ring oscillator is critical since it is difficult to obtain an accurate and stable clock frequency.

In the *asynchronous* interface the synchronous block is not interrupted when a communication is issued. Rather, synchronizers are used to transfer signals through different timing domains safely. It may be a two-flops synchronizer or a FIFO buffer (as seen in 2.2.2). The former method is easy to design and has low area overhead, but it affects latency and throughput. On the other hand, the FIFO buffer can achieve high throughput and low latency at the cost of more area [56].

Finally, the *loosely synchronous* approach exploits known relationships between timing domains to build an efficient communication. These systems are more properly referred to as *multi-synchronous systems* [1]. According to Messerschmitt classification [62], timing relationships between two clock domains can be:

- *Mesochronous*: same frequency but different and stable phase difference.
- *Plesiochronous*: slightly different frequencies, that cause a phase shift.
- *Heterochronous*: different unrelated frequencies. An interesting subcategory is called *ratiochronous* relationship, in which one clock frequency is an exact multiple of the other.

When some bounds on the frequencies are known, it could be possible to eliminate handshaking and build very efficient systems. Nevertheless, these timing relationships are not exploited and integrated into the analysis of the current CAD tools [57].

GALS implementation style is a perfect match for NoC approach, where processing element are intrinsically separated from the communication infrastructure. Several researches have proposed GALS-NoC architectures [63][64], wrappers [65], and routers [66], proving its power and performance benefit.

2.3 Related works

There have been some attempts to join the benefit of asynchronous design and reconfigurable logic.

Manohar [54] introduced an asynchronous FPGA architecture designed for static dataflow systems. In this computational model, a token traveling through a pipeline indicates the flow of data [67], implementing data-driven operations that perfectly fit with the asynchronous paradigm. This architecture relies on QDI logic, i.e., on dual-rail data encoding handshake, exploiting PCHB asynchronous pipeline (Section 2.2.2). The overall architecture is an island-style FPGA very similar to the synchronous counterpart. Each route track corresponds to three wires due to the choice of dual-rail channel. The communication is then wholly asynchronous, and the interconnect is pipelined,

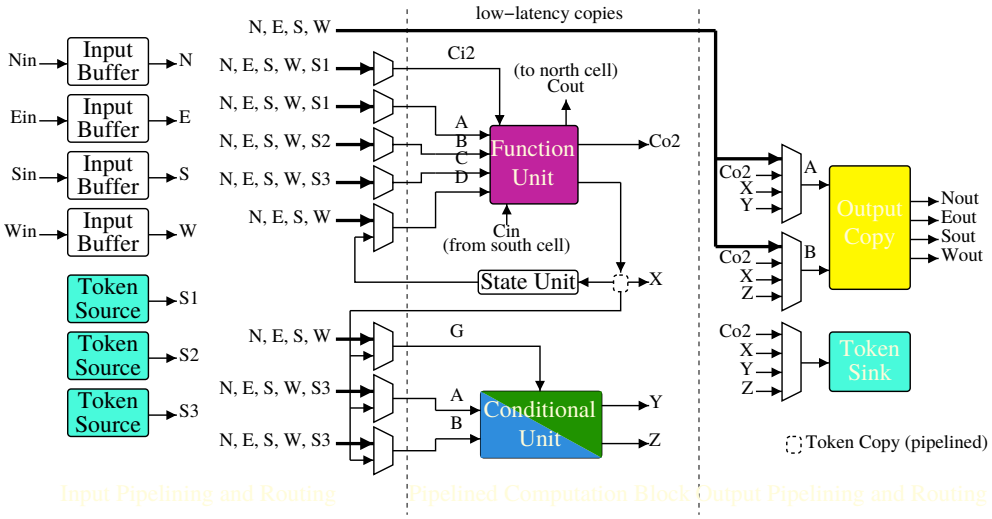


Figure 2.12: Manohar asynchronous FPGA BLE [54].

inserting buffer stages in the switch boxes. The BLE design, depicted in Figure 2.12, is analogous to the synchronous BLE (Section 2.1.1). It contains essential dataflow building blocks: function, source, sink, copy, initial, merge and split (the last two implemented in a single conditional unit). All the logic is implemented as a programmable pull-down stack in a PCHB stage, for example, the 4-LUT is shown in Figure 2.13. This architecture can operate at high throughput and is very robust, proven to be functional in a wide range of voltage and temperature. Moreover, it has been commercially developed [68].

Komatsu et al. adopted a similar solution [69]. Like the previous case, the architecture consists of an island-style FPGA, and a dual-rail data encoding is employed. The main difference is in the BLEs structure. While Manohar proposes a dataflow-based architecture, Komatsu et al. propose a handshake-component-based design. Balsa [70] is proposed as a design methodology that uses handshake components. Balsa is a Hardware Description Language (HDL) that allows describing asynchronous systems hiding to the designer handshake related gate-level details. Thus, this approach is suitable for complex, large-scale asynchronous circuits. Balsa relies on a small set of handshake components, into whom the described circuit will be compiled. With the FPGA logic is possible to implement 39 out of 46 handshake components defined in Balsa manual. Frequently used components are implemented in a BLE (Figure 2.14), while rarely used ones are ob-

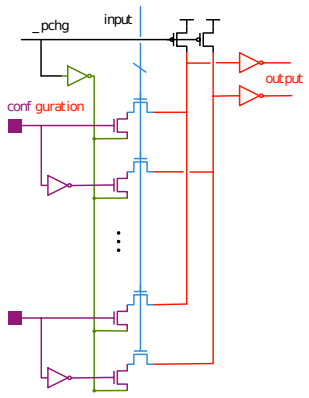


Figure 2.13: Configurable pull-down stack for the four-input lookup table [54].

tained connecting multiple BLEs. A pre-layout simulation in HSPICE showed that the area overhead compared to a conventional architecture is up to 62%, and throughput is nearly halved.

Alternatively to fully asynchronous FPGAs, GALS approaches have been proposed.

Royal and Cheung [10] introduced a level of hierarchy into the FPGA fabric. Standard CLBs are grouped to form large synchronous blocks. Within each of these blocks, the architecture is the same of a synchronous FPGA, with a local clock that spans only through the block. The communication between different synchronous islands, however, follows an asynchronous protocol. As previously analyzed (Section 2.2.3), a synchronization system is needed at the interface between synchronous and asynchronous domain. For this reason, an asynchronous wrapper is provided. The chosen synchronization style is pausible clock, hence the wrapper contains a ring oscillator for clock generation. I/O controllers manage the communication with the asynchronous routing. A four-phase bundled-data protocol has been chosen as handshake protocol. The output port controller receives data bundled with the valid bit from the synchronous block and initiates a handshake communication on the asynchronous channel, meanwhile the local clock is paused until the communication ends. Similarly, the input port controller pauses the clock when new data are required, receives a request and safely latches the data from the asynchronous channel, then the local clock is restored. Data transfer through the asynchronous fabric is facilitated by inserting micropipelines at each

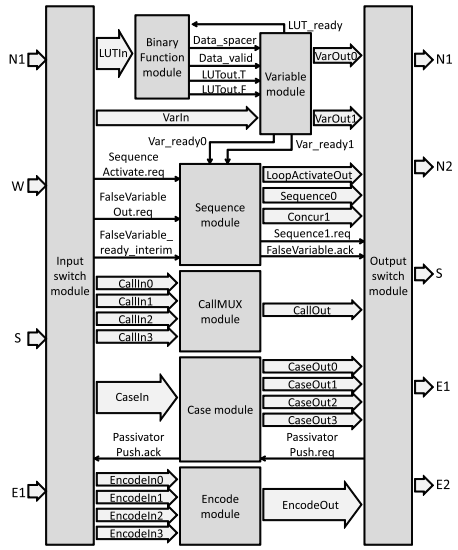


Figure 2.14: Structure of a handshake-component-based BLE [69].

switch box, reducing the time a synchronous block is paused and eliminating deadlock.

Jia and Vemuri [11] further developed the same idea in the proposed architecture called GALS Programmable Logic Array (GAPLA), of which block diagram is represented in Figure 2.15. The approach is the same, using asynchronous wrappers as interface between the two time domains. As before, the wrappers implement a pausible clock approach, containing the local clock generator and the I/O port controllers. The interesting difference is that each synchronous tile has four different wrappers to which it can be connected, and they can be merged if needed, enhancing I/O capacity of a clock domain. Moreover, each wrapper contains 8 I/O ports, allowing some flexibility in the number of bits required for a communication. Inside the synchronous island, all four clocks are routed and each CLB can choose one arbitrarily. The communication protocol adopted is a two-phase bundled-data handshake. Moreover, adjacent tiles have the possibility of direct and fast communication.

This latter approach provides some flexibility in the use of the asynchronous communication, allowing to program size and shape of each synchronous logic block and also the data width of each I/O port.

Nevertheless, one common problem of the above architectures is that the communication protocol is fixed and not programmable. Therefore, it is not possible to explore different communication mechanism,

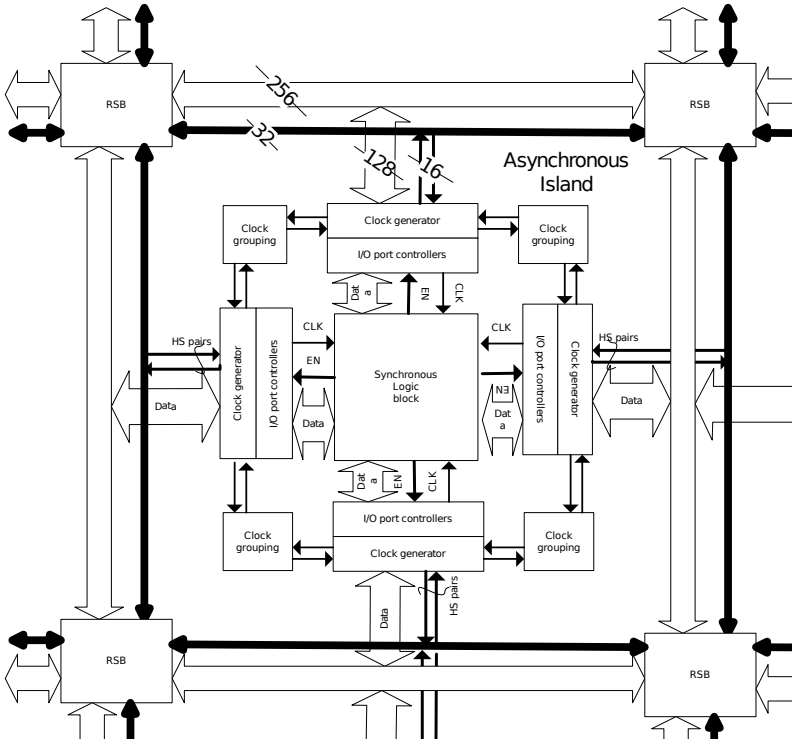


Figure 2.15: Block diagram of the GAPLA architecture [11].

such as implementing an asynchronous NoC router.

Chapter 3

Proposed GALS FPGA

FPGA technology is widely adopted in a variety of applications, offering the benefit of computational capacity and reconfigurability. The vast majority of FPGA devices currently employs an island-style fine-grained architecture [71] and a synchronous design approach. Eventually, some coarse-grained heterogeneous elements are adopted, such as RAM or arithmetic blocks [21].

Here we propose an architecture for FPGAs which is optimized for GALS applications. Such device aims to be used for implementing asynchronous NoCs or SoCs. The focus of this work is mainly on the asynchronous fabric that allows the communication between synchronous tiles. The inclusion of a reconfigurable circuit more specialized for communication mechanism for GALS in a synthesizable coarse-grained FPGA is different from the previous works [10][11], which contain fixed communication mechanism only.

This chapter presents the proposed architecture and describes how a communication circuit is mapped into the asynchronous fabric. In the proposed architecture, the word-wide data handling functions from the Verilog source are mapped to datapath blocks, while the control logic is extracted and mapped to coarse-grained blocks.

A fundamental premise of the chapter is that the current project status is not definitive. The overall architecture is still incomplete, as the asynchronous/synchronous interface has yet to be developed. The focus at the current status is the asynchronous network that surrounds the synchronous blocks. Regarding the tool development, what is now available is the very first version of the placement and routing tools. Many improvements can be made, and some functionalities are still

missing.

The rest of the chapter is organized as follows. Section 3.1 illustrates general desirable properties of the reconfigurable GALS circuit. Section 3.2 presents in detail the proposed architecture. Section 3.3 describes the tool flow needed to synthesize a user-circuit onto the asynchronous fabric. Section 3.4 goes through the tool flow to show a simple usage example.

3.1 Requirements

This section describes common characteristics that are desirable when building a communication circuit for asynchronous systems.

In general, communication circuits can be divided into control and datapath portions. The datapath typically contains memory elements, such as registers or latches. Existing FPGA devices are not optimized for dealing with word-wide data and, as a result, a significant amount of FPGA resources is wasted. In fact, data are saved in scattered CLBs, so it is not possible to efficiently share control signals.

The control circuit usually occupies less area, and it is possible to map it to coarse-grained FPGA resources. Control circuits, nevertheless, should be linked to the correspondent datapath through a fast connection. Moreover, for dealing with asynchronous communication specific elements (C-elements, latches, mutex elements) should be available that are usually not needed in the synchronous design.

Finally, the proposed architecture aims to provide what is needed to implement GALS NoC. Therefore, essential elements for building NoC routers are required. A fundamental building block for a router is the crossbar switch [72], that can be made of word-wide user-controller multiplexers.

Based on the above analysis, some basic requirements for the proposed architecture can be derived.

- Coarse-grained latches are necessary to implement asynchronous pipelines.
- Coarse-grained interconnection and switches will allow efficient implementation of communication circuits.
- Dedicated connection from control logic to datapath.

- User-controlled word-wide multiplexers, to simplify the implementation of asynchronous routers.

3.2 Architecture

Figure 3.1 shows a top-level block diagram of the proposed reconfigurable GALS architecture. It consists of synchronous islands surrounded by a reconfigurable asynchronous fabric. The synchronous tiles can adopt any existing FPGA structure; thus, the primary focus of the thesis will be on the architecture of the asynchronous communication network.

3.2.1 Overall

The asynchronous portion of the reconfigurable circuit employs itself an island-style FPGA structure, formed by coarse-grained CLBs connected by a mesh routing network. Switch Blocks (SBs) divide channels into segments that in general can be of variable length, spanning more CLBs to reduce long connection delays. Although this benefit is desirable, the length of each segment in the proposed architecture spans one CLB, this is partly for simplicity, and partly due to the small scale of the device. The coarse-grained CLB contains two Pipeline Stage Blocks (PSBs), each of which has the elements needed to implement stage of an asynchronous pipeline. The connection between each CLB and the routing channels is possible only on the West and South sides of the block.

The details of the proposed CLB are shown in Figure 3.2. There are two PSBs, one along the x-direction and one along the y-direction. Each PSB is composed of a Control Circuit Block (CCB) and two Data Path Block (DPB), i.e., two 40-bit latches.

3.2.2 Routing channels

Referring to Figure 3.2, the routing networks are separated for data and control, such that the coarse-grained data can save area by sharing configuration bits. As a consequence, there are two kinds of switch blocks, Switch Block Data (SBD) and Switch Block Control (SBC).

Switch blocks are implemented following the directional single-driver style [73], using buffers and multiplexers, as shown in Figure

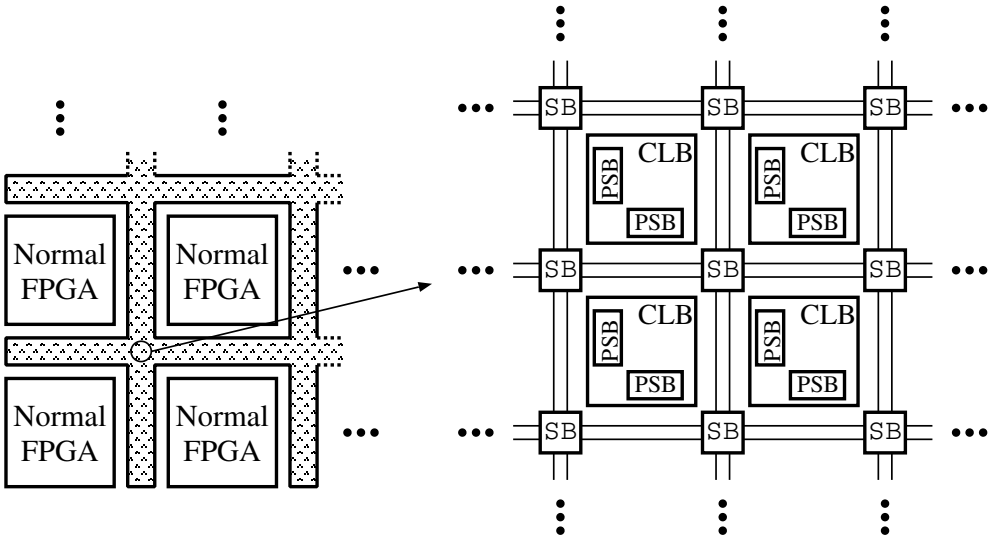


Figure 3.1: Proposed architecture.

3.3. The CLBs outputs connect directly to the adjacent SBs multiplexers inputs, avoiding the use of tri-state buffers and pass-transistors.

The SB topology used is the disjoint switch, in which each track of the routing channel is numbered, and a given track i can only be connected to the same number track or, eventually, shifted by one track when turning. The convention adopted here is that signals flow up/right on odd-numbered track and down/left on even numbered tracks.

The data routing channels employ 8 multi-bit tracks, 4 for each direction. The width of each track is the same as the capacity of data latches, that is 40 bits. The control routing channels are composed of 128 single-bit tracks, 64 for each direction.

3.2.3 DPB

The detail of the DPB implementation is shown in Figure 3.4. It consists of four 10-bits latches, which gate inputs come from the correspondent CCB placed in the same PSB, while the reset input comes from a global wire. Input data are selected from the routing channel through an 8-way multiplexer.

As previously stated, single-driver implementation is used, and consequently, there is no output connection at wires midpoint, but DPB outputs are connected to the adjacent SBs multiplexers inputs.

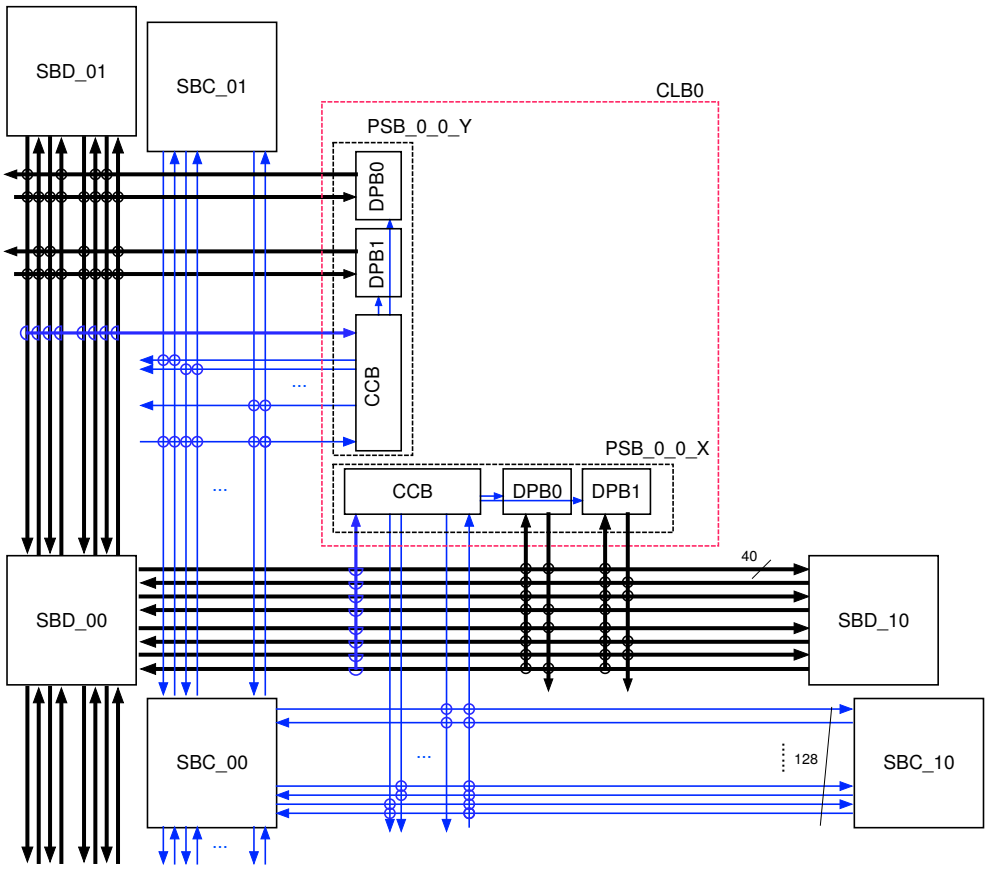


Figure 3.2: The proposed CLB architecture and the surrounding routing channels.

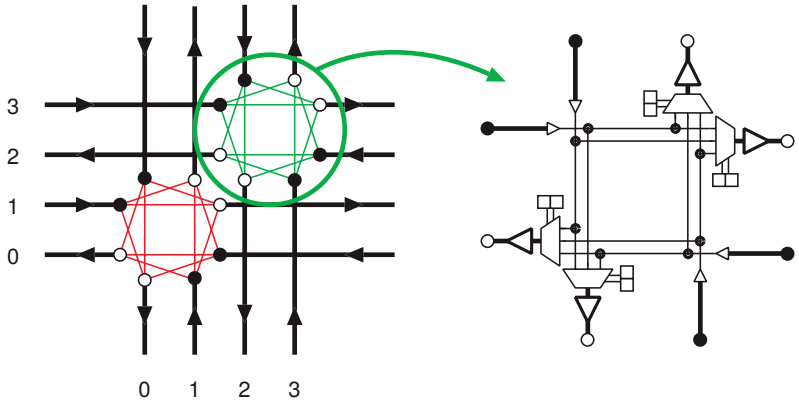


Figure 3.3: Implementation of SBDs and SBCs.

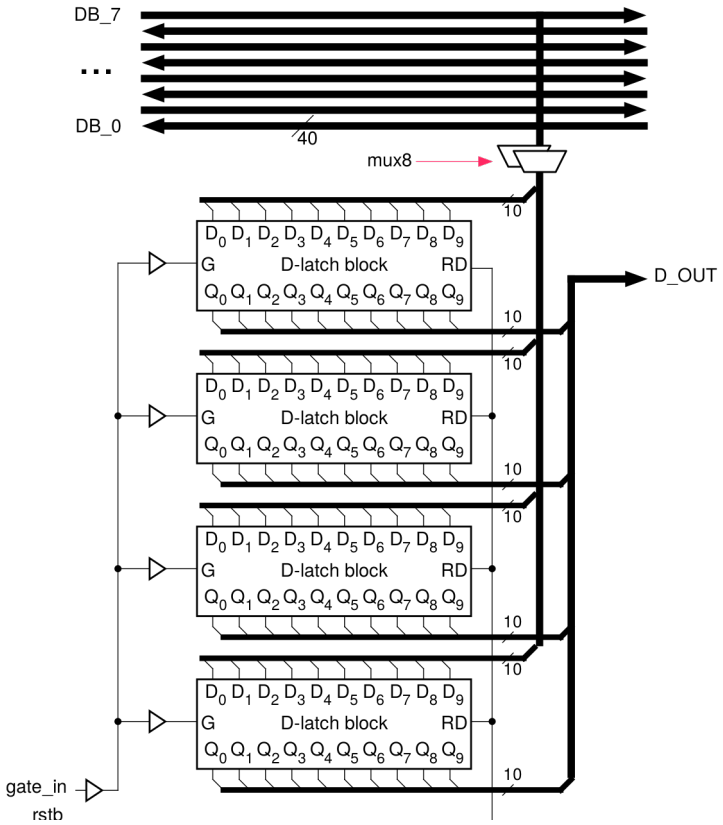
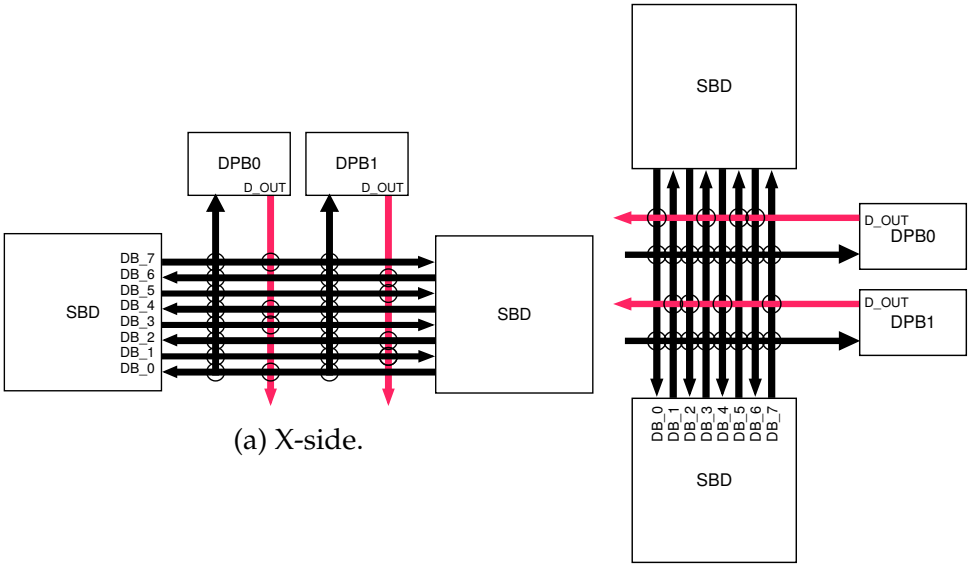


Figure 3.4: Implementation of the DPB.



(a) X-side.

(b) Y-side.

Figure 3.5: Connection between DPB and SBD.

Since two DPBs share the same routing segment, their output is half connected, i.e., connected with half of the available tracks, following the connection patterns of Figure 3.5a and 3.5b

3.2.4 CCB

Figure 3.6 shows the block diagram of a CCB. This block is organized hierarchically.

At the highest level of the hierarchy, a CCB is composed of 8 sub-blocks (CCB_SUBs). The structure of a CCB_SUB is similar to a logic cluster of Figure 2.4, and it is shown in Figure 3.7. CCB_SUBs include the programmable logic of the architecture. They can select inputs from both the 128-bit control channel (CIO) and the data channel. These sub-blocks are divided into two sets of 4 CCB_SUB each. Among each set, there is a local interconnect. The first set of 4 sub-blocks can produce the gate signals directly linked to the correspondent DPBs, allowing to control the flow of an asynchronous pipeline. The rest of the sub-blocks can control adjacent SBDs multiplexers through 16 *sbd_cont* signals. Each CCB_SUB is half connected in input to the CIO, while the connection flexibility in output is $F_{c\_out} = 1/8$, i.e., 16 wires can be connected to CCB_SUB output.

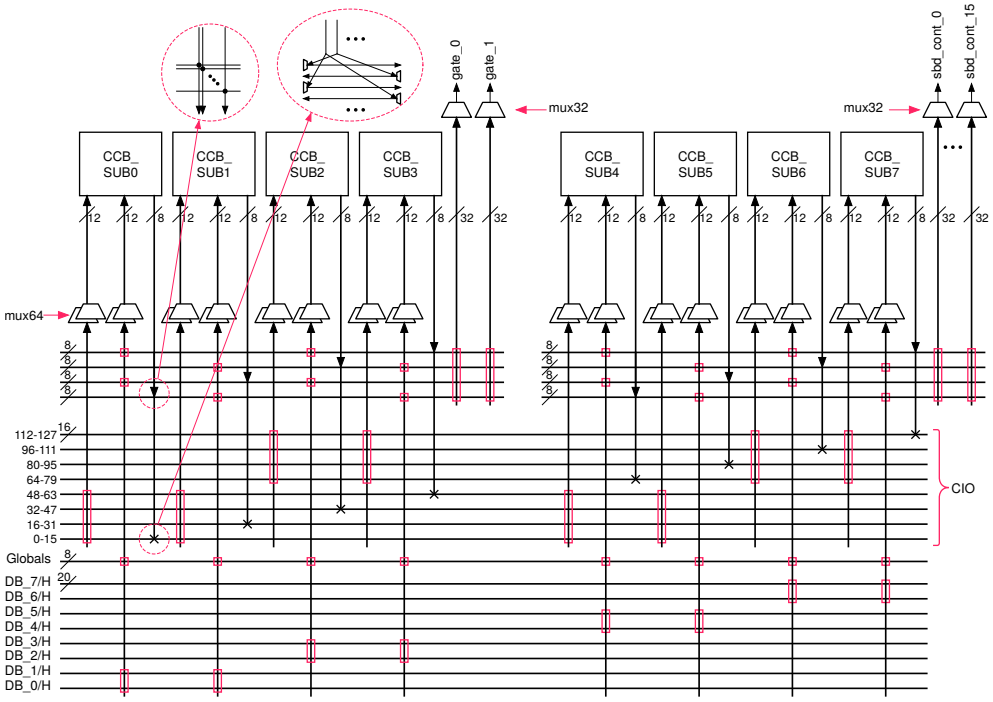


Figure 3.6: CCB block diagram.

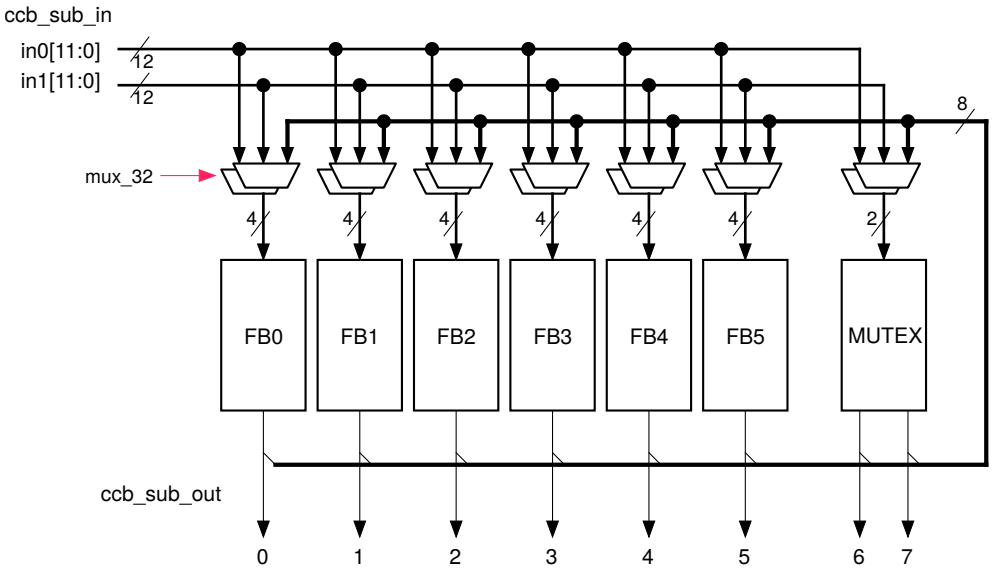


Figure 3.7: CCB-SUB block diagram.

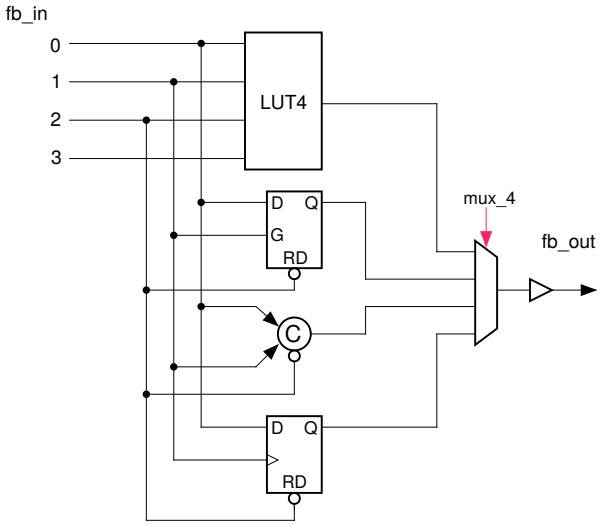


Figure 3.8: Implementation of FB and SB.

It is also possible to select inputs from the data channel. The 20 most significant bits of two data tracks can be chosen from one CCB_SUB. This feature is useful for recognizing flit type in a flit-based transmission. Additionally, 8 global signals can connect to the CCB_SUB inputs.

Except for the mutex element, the six FBs that compose CCB_SUB are similar to BLEs shown in Figure 2.3. Functionalities are enhanced by adding a C-Muller and a latch as further asynchronous elements (Figure 3.8).

Figure 3.9 explains the adopted mechanism to control SBDs at runtime, allowing to include user-controlled multiplexers in the circuit. The selection bits of the SBD multiplexers do not come directly from the bitstream configuration. Instead, for each SBD multiplexer, a configuration one is used to choose whether the selection bits will come from the configuration bitstream or from signals controlled by the CCB of the correspondent segment. These specialized signals from the CCB (*sbd_cont*) are connected to the configuration multiplexer by a direct route.

3.3 Tool Flow

Similarly to the architecture section 3.2, the focus of this section is a tool flow that maps a communication user-circuit on the asynchronous

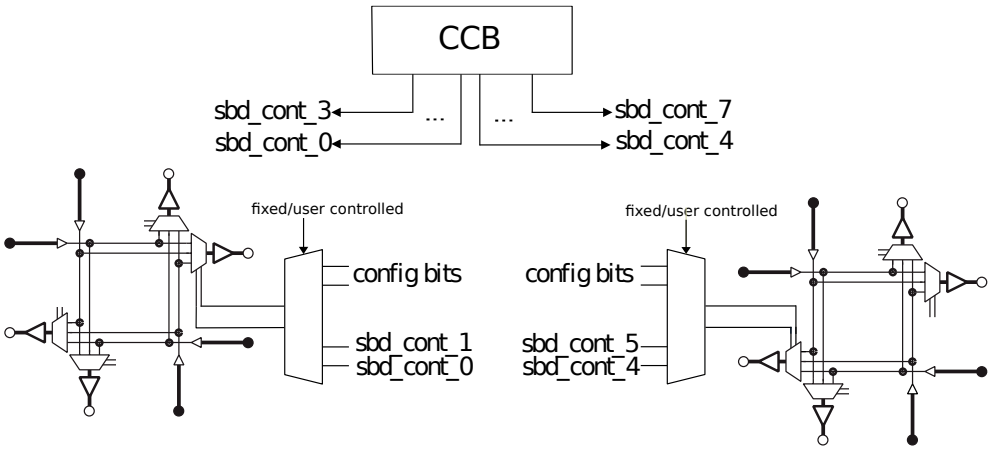


Figure 3.9: SBDs controlled by CCB.

reconfigurable fabric. For the synchronous islands, any commercial CAD can be used.

The tool flow here introduced can compile a synthesizable Verilog user-circuit into a configuration bitstream for the proposed architecture. The novel coarse-grained architecture proposed in this thesis has some inherent characteristics that prevent from using established tool flows. Existent FPGA tools are designed for fine-grained logic, so new functionalities have been added to support placement and routing of the coarse-grained logic. The tool has been developed in Python [74] for simplicity and readability.

Figure 3.10 illustrates complete tool flow, from the Verilog description to the bitstream configuration.

3.3.1 Front-end synthesis

The mapping process of user-circuits onto the proposed coarse-grained architecture needs a different approach, respect to the standard flow illustrated in Section 2.1.2. Nevertheless, the initial elaboration, the so-called front-end synthesis, must be the same [75]. The front-end synthesis involves parsing the user-circuit description, elaborating and expressing it in some convenient format for the tools to perform the mapping.

Odin [30] and ABC [32] are used for the front-end synthesis, in the same manner as in VTR, the well-established academic tool flow of Figure 2.6. Odin receives as inputs the user Verilog description

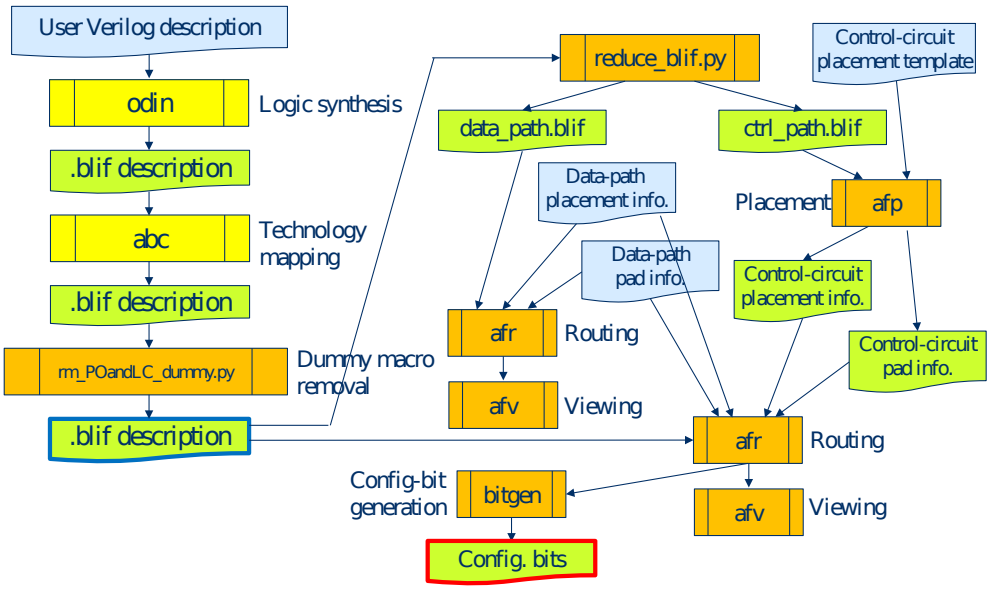


Figure 3.10: Proposed architecture tool flow.

and a top-level description of the resources available in the architecture. It elaborates the design to single-bit operations, producing then a BLIF file, that is a textual description of the circuit at a logic-level [31]. ABC subsequently elaborates the file, cleans up dangling logic and technology-maps the logic to LUTs. One more step is needed after ABC elaboration, *rm_POandLC_dummy.py*, which eventually removes doubled signal names for output ports as well as dummy macros introduced in the source Verilog file for cutting combinatorial loops to avoid an ABC bug.

3.3.2 Back-end synthesis

After the front-end, the following phase is the back-end synthesis, also known as physical synthesis. This phase, which involves clustering, placement and routing steps, is performed separately for control circuit and datapath. Therefore, the first step is to divide the control logic portion of the circuit from the datapath.

The step of the tool flow that performs this separation, *reduce_blif.py*, accepts the BLIF file from the front-end synthesis as input, and produces as output two different files, a description for the datapath (*data_path.blif*) and a description of the control circuit netlist (*ctrl_path.blif*). Furthermore, the *reduce* step generates additional information needed in the

following placement stage: the lists of the elements to be placed for both control logic and datapath. At this point the tool flow branches in two parallel elaborations.

Datapath synthesis

Regarding the datapath, the placement is conducted manually. Given the relatively small size of the reconfigurable circuit, it is easy or sometimes efficient for users to specify placement information for datapath.

The elements to be placed are I/O pads, latches, and user-controlled SBD multiplexers. Each latch must be placed in a DPB, while for data multiplexers, if any, must be specified the position in the selected SBD. Particular attention must be taken in placing all the datapath components in a way that is coherent with the channel tracks direction.

Once every latch, multiplexer and I/O pad has a location on the circuit, the routing phase determines how to connect all the elements. In the datapath portion of the circuit, the connections are represented as 40-bit wide routes. Each route has exactly one source and one or more sinks. The routing is performed using channel tracks and SBDs.

Since only inter-PSB routing is needed for the datapath, the XY-routing algorithm is used.

Control logic synthesis

Once the routing of the datapath is completed, back-end synthesis for the control portion of the circuit can be performed.

At this stage, all the control Logic Elements (LEs) present in the list generated at the *reduce* step must be mapped to some FBs in the architecture. Similar to the datapath portion, it is needed to feed the placement tool with some auxiliary information. More in detail, for each LE the correspondent PSB location must be manually decided. However, connection constraints limit the possible locations. For example, the logic elements that provide gate or SBD control signals must be in the same PSB of the related latch or SBD multiplexer.

Once the PSB has been decided, the FB location of each LE is determined. The placement problem of LEs in FBs within the PSB can be formulated as a Boolean Satisfiability problem (SAT) [76]. In fact, the problem is to obtain a placement that satisfies all the constraints due to the limited connection between LEs available.

SAT problem can be expressed by Conjunctive Normal Form (CNF), that is: given a set of *variables*, assign a Boolean value to all of them such that a given conjunction (AND) of *clauses* is satisfied, i.e. true, where a *clause* is a disjunction (OR) of *literals*, and a *literal* is a positive or negative variable. The problem is solved and is *satisfied* if such a set of variables exists, otherwise the problem is *unsatisfied*. The SAT-solver used in the tool is MiniSAT [77], a widely-used, minimalistic, open-source SAT.

In our problem each LE is specified by four true SAT variables: two variables for defining the CCB_SUB location, one for the FB and one eventually for which mutex output select. Encoding the constraints for the SAT problem means to write down clauses that come from unacceptable solutions. For example:

- For one LE at least one FB should be selected.
- For one LE no more than one FB should be selected.
- Two LEs can't be placed in the same FB.
- If two LEs are connected, they must be placed in two FB that can be connected.
- Inter-PSB communications should not intersect.

In Figure 3.11 an example of how the variables uniquely define the position in the CCB is illustrated. In the example, for a certain LE, the SAT-solver has found out that the true variables that satisfy the constraints are LE|S1, LE|SS1, LE|5, LE|out1. The resulting LE position is then on the FB5 of the CCB_SUB3. The details of the constraints encoding can be found in the Appendix A.

An incremental solver approach [78] is used for guaranteeing the correct placement when there is congestion on LUTs inputs, a situation that cannot be encoded as a constraint a priori.

The SAT-solver provides one among all the possible solutions, that is accepted without optimization. Moreover, if too many LEs are placed within the same CCB, the SAT-solver is not able to provide a solution in an acceptable amount of time. Thus, if this is the case, the user must rearrange the placement, spreading the elements in different CCBs.

For the control logic, the router used is based on a breadth-first search algorithm. Routing is done by expanding the routing wave

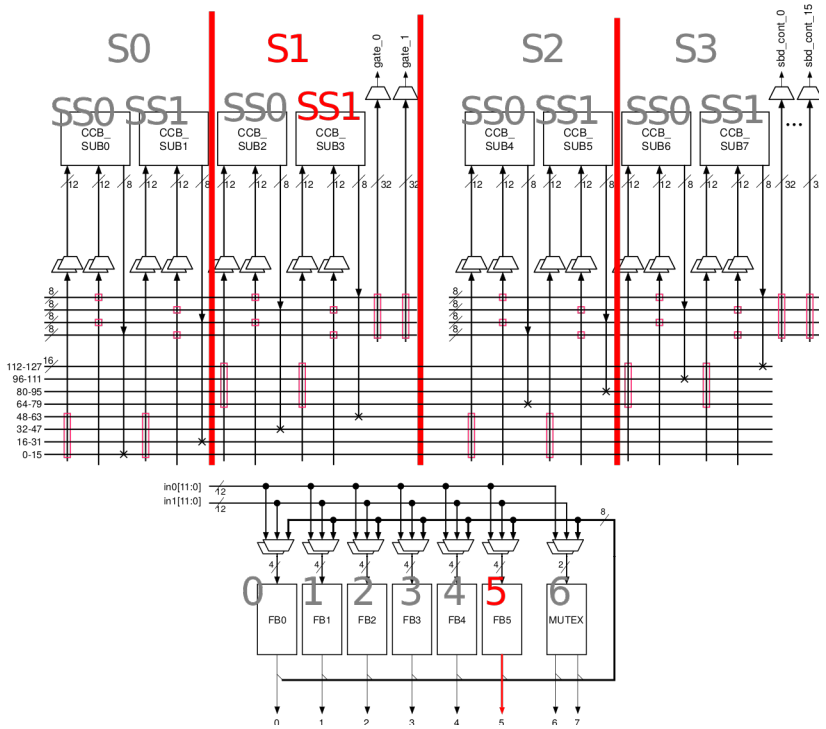


Figure 3.11: SAT variables example.

front from the source along all the possible branches until the first destination is reached. More efficient algorithms are available [79][80], but, due to the small scale of the circuit, the breadth-first search routes in an acceptable amount of time. For a route with n sinks, the router is involved n times to perform the routing.

Finally, the bit stream generation configures all the reconfigurable elements to make the circuit behave as designed.

3.4 Mapping example

In this section a full example of mapping an user-circuit onto the proposed asynchronous architecture is shown. A simple 4-stages mousetrap FIFO has been chosen for the example. Here, we will go from the Verilog description though the design flow until the bitstream generation, following the steps of Figure 3.10.

The Verilog description of the user-circuit must be done aware of the hard blocks available in the architecture. These hard blocks are de-

fined in the architecture file that is an input for ODIN. A file snippet of this architectural description file is shown in Figure 3.12. With reference to the file snippet, the keyword *"D_dl_blk"* directs ODIN to synthesize a 40-bit DPB latch. The architecture file lists the input and the output ports of each hard block. The other blocks listed in the architecture file are: C-element, mutex, latch, flip-flop, and user-controlled mux.

```
<model name="D_dl_blk">
  <input_ports>
     <port name="I"/>
     <port name="G"/>
  </input_ports>
  <output_ports>
     <port name="O"/>
  </output_ports>
</model>
```

Figure 3.12: Example declaration of a hard block.

Accordingly, the instantiation of the hard blocks in the Verilog description has to follow the architectural declaration. Figure 3.13 illustrates example Verilog code for the instantiation of a mousetrap stage using single latch and 40-bit latch hard block.

The result of ODIN and ABC elaborations is a BLIF file netlist that is subsequently separated in control and datapath portions. Basically, the datapath portion comprehends all the elements connected to a 40-bit latch or a user-controlled mux, the rest is part of the control path. At this point, the lists of the elements to be placed are provided, named accordingly the BLIF terminology.

The next step consists then to manually place the listed datapath elements onto the coarse-grained architecture. Careful placement is needed to guarantee all the connections between the elements, aware of the connection patterns of Figure 3.5. For our FIFO example, the datapath manual placement onto a 4x4 architecture is shown in Figure 3.14. The resulting routing is then shown in Figure 3.15. The position of each datapath latch is specified by CLB number, PSB side (0=X, 1=Y), and DPB number. In the case of user-controlled multiplexers the SBD position and side must be specified. Moreover, the I/O connections must be defined, specifying the pad coordinates, the directions

```
module mt_stage(idata, odata, ireq, oack, oreq, iack,
    rstb);
   input [39:0] idata;
   output [39:0] odata;
   input ireq, iack, rstb;
   output oreq, oack;
   wire gate;

   D_dl_blk ldd(.I(idata), .G(gate), .O(odata));
   C_d_latch ldc(.D(ireq), .G(gate), .RD(rstb), .Q(oreq)
      );

   assign gate = ~(oreq ^ iack);
   assign oack = oreq;
endmodule
```

Figure 3.13: Example instantiation of hard blocks in a mousetrap stage module.

and the bus number.

Once the datapath placement and routing has been defined, the control path elements must be placed. While for the datapath section defining the position of each element is relatively simple, the high number of connection constraints makes this operation though for the control path. Every elements in the placement list must be assigned to an FB into the CCB. Employing the placement tool described in Section 3.3.2, the user only has to define the PSB position of each logic element.

In the case of the FIFO example, a possible input to the placement tool is shown in Figure 3.16. All the logic elements to be placed must be listed into each PSB group. Since these lists can become quite large, the keywords *"EXACT"* and *"AND"* are used reduce them. *EXACT* is used to place one single element, so the complete name must be specified. On the other hand, *AND* is used to list multiple logic elements related to the same module, so the common part of the names is listed. For example, writing *"AND: mt_stage_inst_0"* means that in that particular PSB all the logic elements which name contains *"mt_stage_inst_0"* are listed. Moreover, the exact name of the I/O pins must be specified in the selected I/O pad list. Some constraints exist when placing the

```
le.top.mt_stage+mt_stage_inst_0.D_dl_blk+ldd^O~39 :
   root.clb[0].psb[0].dpb[0].dl_blk[0]
le.top.mt_stage+mt_stage_inst_1.D_dl_blk+ldd^O~39 :
   root.clb[0].psb[0].dpb[1].dl_blk[0]
le.top.mt_stage+mt_stage_inst_2.D_dl_blk+ldd^O~39 :
   root.clb[4].psb[0].dpb[0].dl_blk[0]
le.top^odata~39 :
   root.clb[4].psb[0].dpb[1].dl_blk[0]
top^idata : Pad_DB_in_1_1_W[1]
top^odata : Pad_DB_out_1_1_W[2]
```

Figure 3.14: Datapath placement file.

logic elements in the architecture. Elements that produce a gate signal for the DPBs must be placed in the same PSB, and those that produce control signals for the user-controlled multiplexers must be placed in the adjacent PSB.

Based on the mentioned information, the placement tool generates an output that specifies the FB position of each logic element, and the CIO position of each I/O signal, similarly to the datapath placement seen in Figure 3.14. The control path routing can be now obtained. The result for the FIFO example is shown in Figure 3.17. Finally, the routing description and the BLIF file feed the bitstream generation script, that fills the configuration bits, given as an input to the architecture.

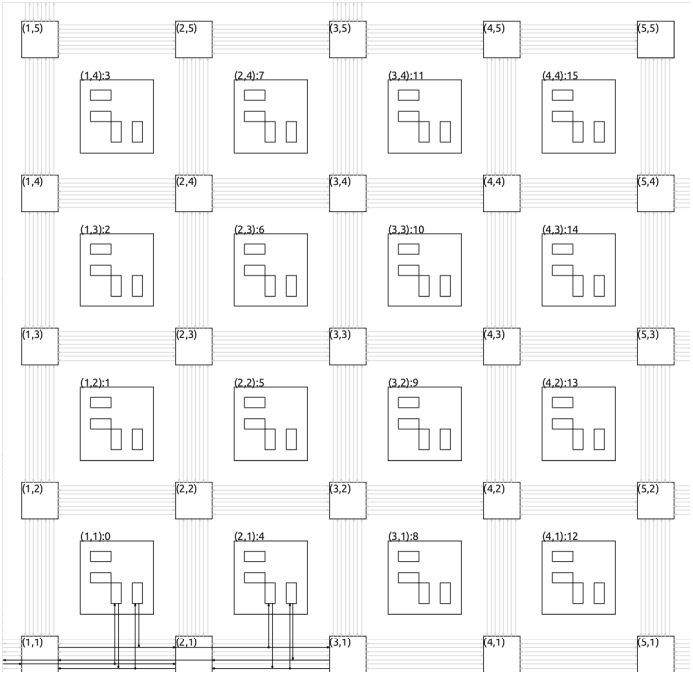


Figure 3.15: FIFO datapath routing result.

```
PSB_1_1_X begin : # stages 0-1
  EXACT: le.top^oack
  AND: mt_stage_inst_0
  AND: mt_stage_inst_1
end

PSB_2_1_X begin : # stages 2-3
  EXACT: le.top^oreq
  EXACT: le.top^odata~39
  AND: mt_stage_inst_2
  AND: mt_stage_inst_3
end

PSB_0_1_X begin : # for I/O Pad_CIO_1_1_W
  top^ireq
  top^oreq
  top^iack
  top^oack
end
```

Figure 3.16: Control circuit placement.

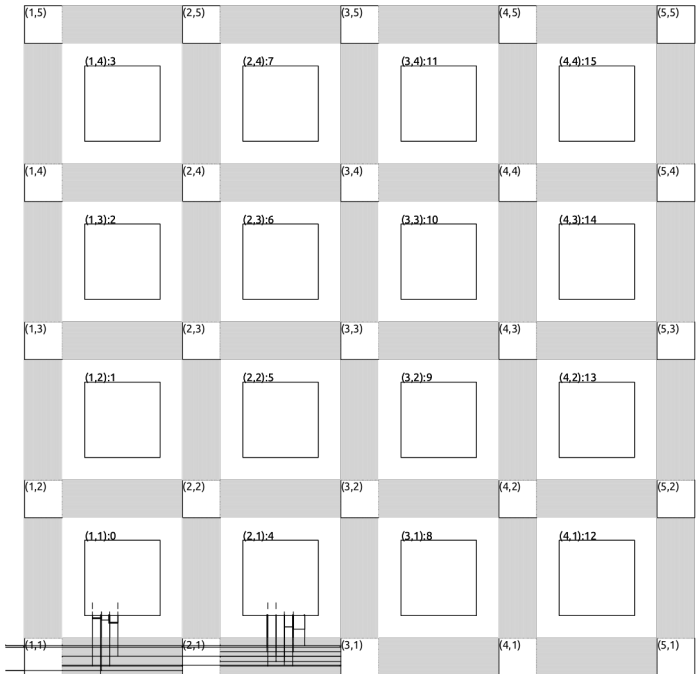


Figure 3.17: FIFO control path routing result.

Chapter 4

Performance Evaluation

The previous chapter completes the description of the architecture and the tool flow for the proposed reconfigurable circuit. The next step is to evaluate the performance obtained when mapping benchmark user-circuits onto the FPGA. As the focus for the architecture and the tool flow was on the asynchronous reconfigurable fabric, similarly, the performance evaluation is conducted on the same portion of the circuit. Therefore, the benchmarks used are three different communication-purpose user-circuits: a four-stages asynchronous pipeline, a crossbar switch, and an asynchronous router. Besides, a combinatorial user-circuit (ripple-carry adder) is also tested to evaluate the performance outside the intended field of use.

Since the project is still in development phase, the simulation approach is adopted for this preliminary evaluation results. To prove the effectiveness of the proposed architecture, the simulation results are compared with an asynchronous FPGA following a fine-grained style. The coarse-grained architecture is expected to outperform the fine-grained one when built-in datapath structures are used (i.e., data latches, user-controlled multiplexers), while similar performances are expected for user-circuits that do not exploit datapath functionalities.

For the fine-grained architecture, VPR tool is used for development. For this tool, the front-end synthesis is the same as the one introduced in Section 3.3.1 and the standard VPR place & route is used for the physical synthesis.

The simulation flow, illustrated in Figure 4.1, is the same for the fine-grained and coarse-grained architecture. Delays for the Standard Delay Format (SDF) file are extracted from a 130nm bulk CMOS tech-

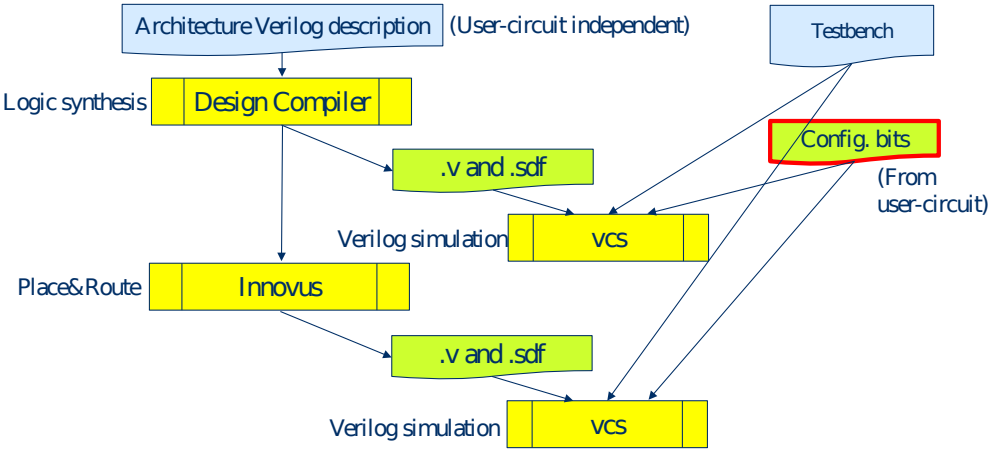


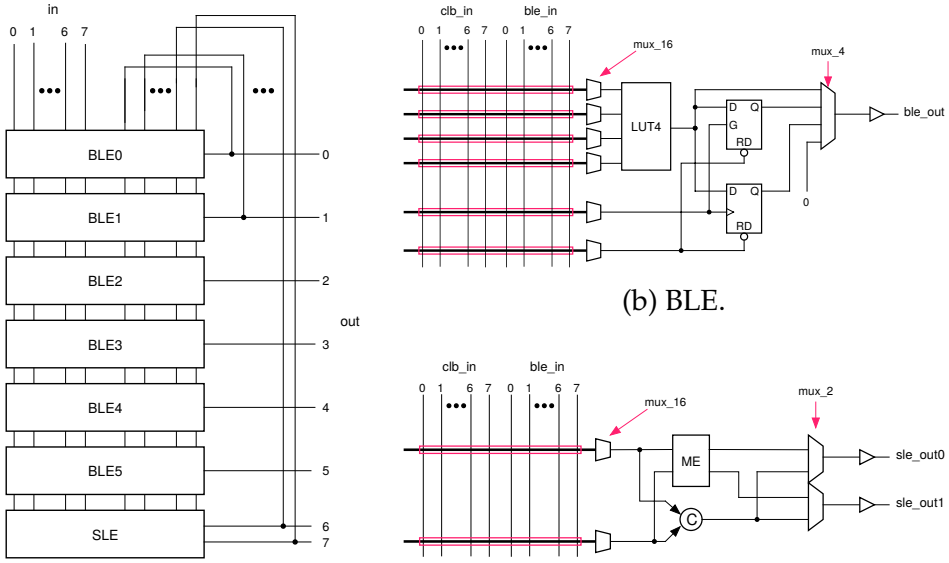
Figure 4.1: Tool flow for simulation.

nology library. For more accurate delays evaluation, physical synthesis would be needed in a future development stage.

The rest of the chapter is organized as follows. Section 4.1 illustrates the fine-grained architecture used for comparison. Section 4.2 presents in detail the set-up and the results of the experiments.

4.1 Fine-grained FPGA

The fine-grained architecture used for the comparison with the proposed coarse-grained one is built as a traditional island-style FPGA. It has been designed as an asynchronous FPGA targeting single-rail data encoding. The architecture is modeled on top of the *Classical Architecture* template provided by VTR tool flow [12]. This *Classical Architecture* consists of an array of homogeneous CLBs, built similarly to the cluster of N BLEs shown in Figure 2.4. The template is then modified as depicted in Figure 4.2. The CLB has eight general inputs and seven BLEs per cluster plus one Special Logic Element (SLE), and each of the LUTs has four inputs. Each BLE has a LUT with the output optionally latched or registered, while an SLE accommodates the asynchronous elements, a mutex, and a C-element. The routing architecture uses wire segments of length 4, channel width 64, unidirectional single-driver routing, Wilton switch box [81]. CLBs can be connected on the four sides, are fully connected in input while the connection flexibility in output is $F_{c\_out} = 1/4$, i.e., 16 wires can be connected to CLB output.



(a) Logic cluster.

(b) BLE.

(c) SLE.

Figure 4.2: Fine-grained CLB architecture.

There are four I/O pins per I/O block.

The entire VTR flow is used for the synthesis. The front-end synthesis is the same for the proposed architecture, described in Section 3.3.1. The back-end synthesis, in this case, reckons on VPR, which implements simulated annealing for the placement and Pathfinder negotiated congestion algorithm for the routing [79][33].

For the simulation, a behavioral description of the architecture itself is needed. For this purpose, a script has been developed based on Zuma [82], an open-source FPGA overlay (also called "FPGA-on-an-FPGA"). In our case, Zuma is not used for its primary purpose of FPGA overlay. Instead, it is used to read the VPR output and to construct an internal data structure. Then, the script generates a behavioral Verilog description of the FPGA architecture, which is an input of Design Compiler, as well as the configuration bitstream used for the simulation, as shown in Figure 4.1

4.2 Results

Here the experimental results are presented. For the circuits where an asynchronous pipeline is present, three metrics are taken into account

for determining the performance:

- forward latency: time interval $req_{in} \rightarrow req_{out}$ when the pipeline is empty, i.e., slow injection rate and fast consuming rate.
- backward latency: time interval $ack_{in} \rightarrow ack_{out}$ when the pipeline is full, i.e., fast injection rate and slow consuming rate.
- throughput: inverse of the time interval between two consecutive req_{out} when the pipeline is full, i.e., fast injection rate and fast consuming rate.

The asynchronous pipeline chosen for the evaluation is the mousetrap pipeline (Section 2.2.2) because it has shown to have good performance and it can be implemented using ordinary logic. Therefore, a two-phase bundled data handshake protocol is used. Different pipelines may be implemented and evaluated, as well as different protocols. However, the proposed architecture does not allow to implement a dual-rail encoding.

As previously stated (Section 2.2.2), the bundled data protocol involves the use of delay matching to guarantee that the request signal arrives at the receiver after every data bit is stable and valid, and that data remains stable for a hold time after the acknowledge signal. This delay matching on *req* and *ack* is usually implemented via an inverter chain or a copy of the critical path. In our case, latches available in the architectures are used in transparent mode for creating delay elements. Moreover, for reconfigurable circuits, the delay additionally depends on the relative position of the elements onto the device. Therefore, the evaluation of the delay needed can only be performed after the place & route phase. Here, an iterative process is used: no delay is added at the first iteration, and it is incremented until the first correct configuration is reached. To be noted, however, that it may be possible that, using a different place & route algorithm, the same number of latches added is not enough due to different routing delays.

Finally, the logic array size is decided as the minimum needed to fit the most complex circuit of the set of experiments, that is a five-ports asynchronous router. Accordingly, the fine-grained logic array size is 25x25 CLBs, while the coarse-grained one is 4x4 CLBs.

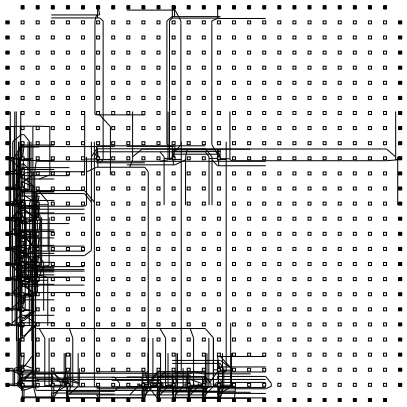


Figure 4.3: Four stages asynchronous pipeline fine-grained mapping.

4.2.1 FIFO

In the most straightforward asynchronous pipeline, the logic blocks between two adjacent stages are just wires, so that the pipeline acts as a FIFO buffer with data being enqueued by the input request signal and data being dequeued by the output acknowledge signal.

Figure 4.3 shows the results of the place & route on the fine-grained. Coarse-grained mapping has been shown in details in Section 3.4.

It is interesting to note that in the fine-grained implementation it has been needed to add three latches delay on the acknowledge line of each pipeline stage and six latches delay on the output request line, while for the coarse-grained architecture only one latch on the output request was added. The reason for this behavior is that in the coarse-grained architecture data-bits follow the same path, thanks to DPBs and separated data-tracks, thus the dispersion on the arrival time is less accentuated than the fine-grained. This behavior is common for the other user-circuits analyzed.

Table 4.1 presents the simulation results for the FIFO user-circuit. Here the proposed architecture performs up to 2.3 times better than the classical architecture.

4.2.2 Crossbar

A fundamental element in the construction of a router is the crossbar switch. This device has multiple inputs and multiple outputs and is capable, in general, of flexibly connecting input ports to output ports. In the specific case of a crossbar switch for mesh-type topology routers,

Table 4.1: FIFO simulation results.

	Forward Latency *(ns)*	Throughput *(MHz)*	Backward Latency *(ns)*
Fine-grain	26.90	55.83	39.80
Coarse-grain	11.56	119.76	17.85

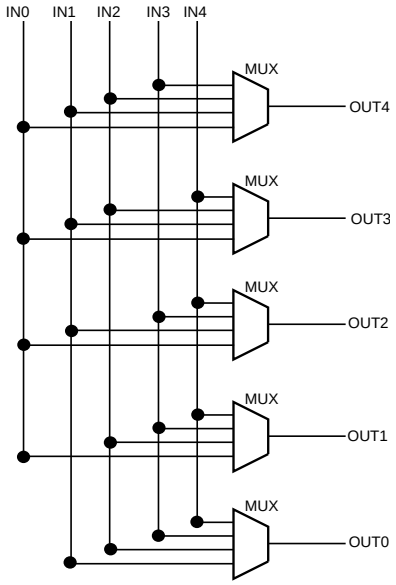


Figure 4.4: 5x5 crossbar block diagram.

it has five ports and is not fully populated, i.e., equally labeled ports cannot be connected (e.g., North input to North output). The crossbar can be easily implemented with standard multiplexers, as shown in Figure 4.4.

Regarding the coarse-grained approach, the crossbar adopts the user-controlled multiplexers available in the architecture. It is worth to discuss some implementation details about the mapping of the crossbar on an SBD, as shown in Figure 4.5. To build the crossbar out of the provided user-controlled multiplexer, it has to be guaranteed at placement time that each input comes from a different direction. Since there are five inputs and four SBD sides, at least one input must be on all the sides of the SBD (in the case in Figure 4.5 it is the input port number 3, the dashed line). Therefore, one selected data input must be routed

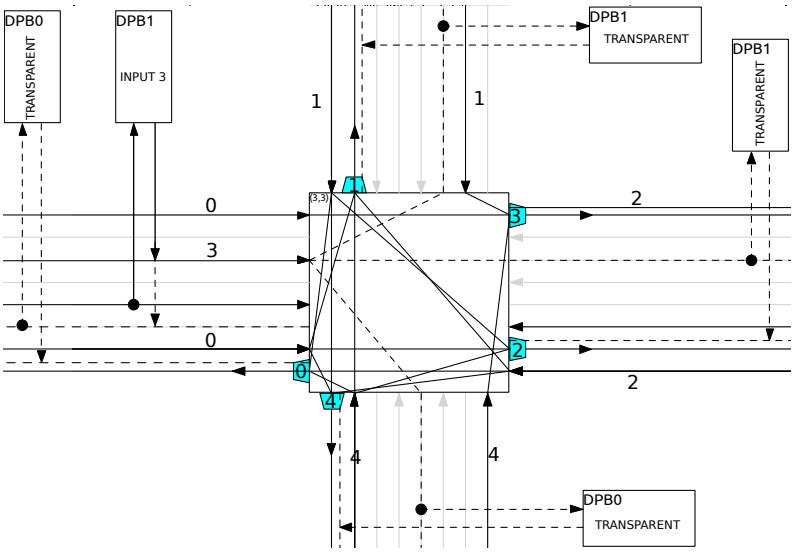


Figure 4.5: 5x5 crossbar mapping detail.

through data latches adjacent to the SBD. This behavior has to be specified in the Verilog description of the user-circuit, as the transparent latches used for this trick must be placed in DPBs as every other latch.

The user-circuit is composed of the crossbar itself inserted between two mousetrap pipeline stages. The mechanism for the control logic is the same used in the router described by Ghiribaldi et al. [66].

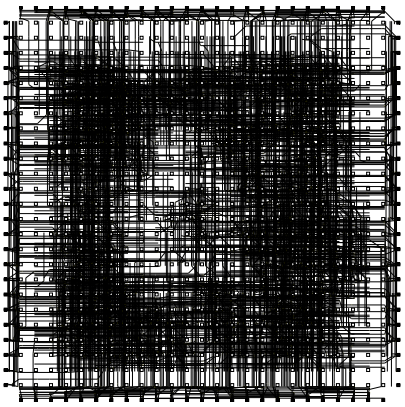
In this case, the fine-grained version of the user-circuit slightly differs from the coarse-grained one. In fact, the former architecture has 400 I/O pins, not enough to connect 400 data bits plus 35 control bits. For this reason, the word width, in this case, has been reduced to 34 bits, based on the assumption that more data bits do not strongly impact the performance: wider words may increase the dispersion of the data delays, and some more delay elements may be needed.

Figure 4.6 shows the results of the place & route on the fine-grained (4.6a) and on the coarse-grained architecture (4.6b, 4.6c).

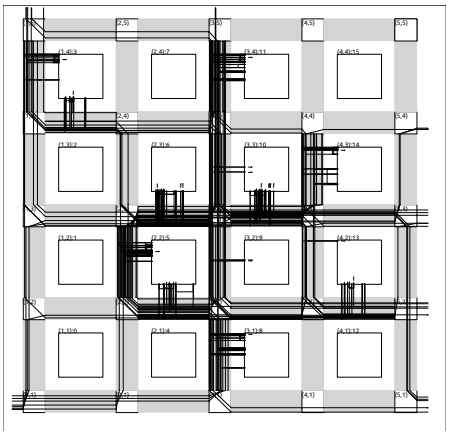
Table 4.2 presents the simulation results for the crossbar user-circuit. Here the proposed architecture throughput is 3 times higher than the classical architecture.

4.2.3 Router

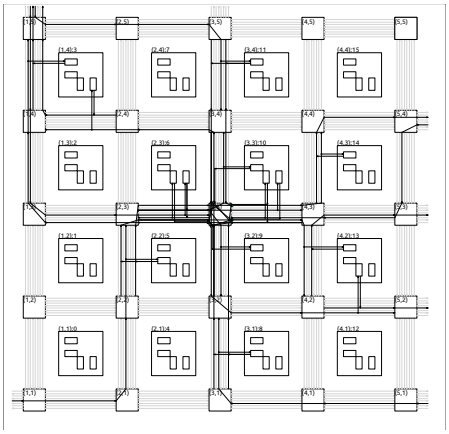
Merging the FIFO, the crossbar, and some more control logic, a complete asynchronous NoC router is obtained. The additional logic re-



(a) Fine-grained.



(b) Coarse-grained control.



(c) Coarse-grained data.

Figure 4.6: 5x5 crossbar mapping.

Table 4.2: 5x5 crossbar simulation results.

	Forward Latency *(ns)*	Throughput *(MHz)*	Backward Latency *(ns)*
Fine-grain	33.64	23.17	26.35
Coarse-grain	12.41	68.63	14.96

quired is for the computation of the packet route and the arbitration between concurrent requests on the same output port.

The router architecture selected as user-circuit is the NoC switch described by Ghiribaldi et al. [66]. It is an asynchronous NoC router relying on 2-phase bundled-data handshake protocol, based on the mousetrap pipeline. Other features are five input and five output ports, wormhole switching strategy and algorithmic dimension-order routing (route first in X, then in Y dimension).

The actual router is preceded by 5 four-stages FIFO buffers, one for each input port, implemented as the asynchronous pipeline of Section 4.2.1. A route selector recognizes the flit type and extracts the routing information from the head flit. The request is then propagated to the selected output port. An asynchronous arbiter guarantees that one packet per time occupies the output port; it is built with specific asynchronous components, i.e., C-elements and mutexes. Data are switched from input ports to output ports by the crossbar described in Section 4.2.2 The architecture details can be found in the cited paper [66].

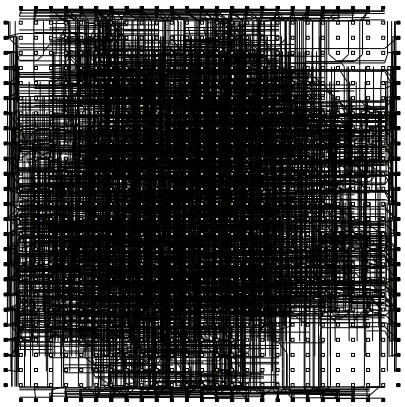
As it has been explained for the crossbar (Section 4.2.2), the word width for the fine-grained architecture is reduced to 34 bits to allow to map the user-circuit with the restricted number of pins available. The crossbar is implemented with the same technique illustrated in Figure 4.5.

Figure 4.7 shows the results of the place & route on the fine-grained (4.7a) and on the coarse-grained architecture (4.7b, 4.7c).

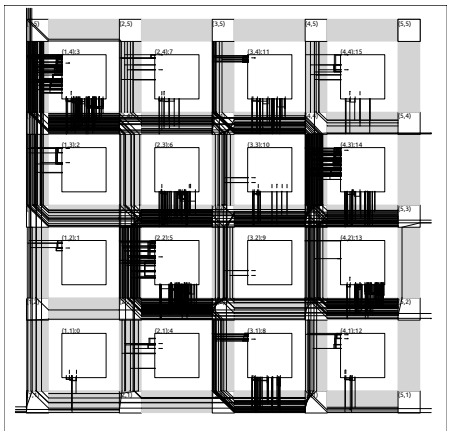
Table 4.3 presents the simulation results for the router user-circuit. Here, two kind of latencies are evaluated: the *Head Forward Latency* is the latency of the head flit, that goes through the router control logic to define the output port to be used; the *Data Forward Latency* is the latency of the following data flits that do not pass through the routing logic. The proposed architecture performs up to 3.2 times better than the classical architecture.

4.2.4 Adder

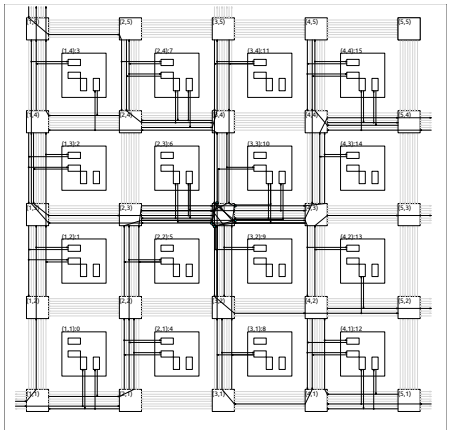
Finally, a combinatorial user-circuit is tested to evaluate the performance of the proposed architecture out of the intended field of application. For this purpose, a common combinatorial circuit is used, a ripple-carry adder, that is nothing else than a chain of full-adders.



(a) Fine-grained.



(b) Coarse-grained control.



(c) Coarse-grained data.

Figure 4.7: 5 ports asynchronous router mapping.

Table 4.3: Asynchronous router simulation results.

	Head Fw Latency *(ns)*	Data Fw Latency *(ns)*	Throughput *(MHz)*	Backward Latency *(ns)*
Fine-grain	93.06	57.89	21.00	83.08
Coarse-grain	34.33	19.84	67.52	32.64

Table 4.4: Ripple Carry Adder simulation results.

	Coarse-grained		Fine-grained
	SAT-solver *(ns)*	Manual *(ns)*	*(ns)*
32 bit Adder			
Average Delay	13.65	12.10	18.34
Worst Case Delay	66.24	56.06	78.61
64 bit Adder			
Average Delay	15.95	13.90	21.95
Worst Case Delay	131.44	109.59	156.75

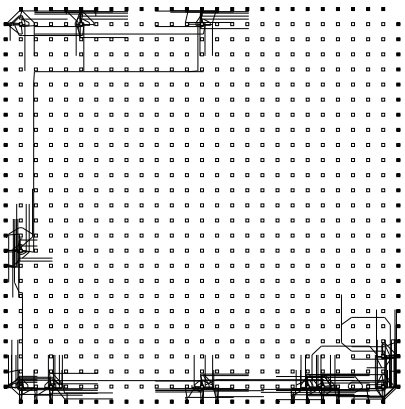
A 32-bits adder and a 64-bits adder are implemented. No datapath portion is used for the coarse-grained circuit as it is not possible to elaborate datapath signals.

The relative simplicity and regularity of the circuit permits to investigate one further aspect, that is the loss of performance due to SAT-solver limits. In fact, if there are too many constraints that limit the solution to few possible configurations, the problem is hard to solve, and the SAT-solver takes an indefinite amount of time. This case happens when a large number of LEs is required to be placed in the same CCB. Thus, there may be some optimal solution that the SAT-solver cannot provide.

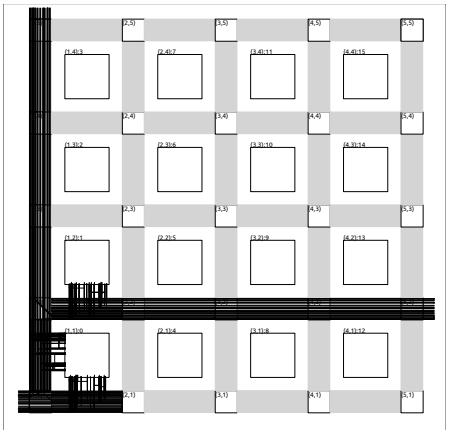
Figures 4.8 and 4.9 show the mappings of the adders onto the architectures. It is possible to notice that using the manual placement a more compact mapping can be achieved, while, for making the SAT-solver provide a solution, functional blocks have to be spread along the circuit.

Table 4.4 shows the results of the comparison. First of all, it can be noted that the coarse-grained architecture performs slightly better than the fine-grained one, regardless of the adopted mapping. The improvement, however, is much smaller with respect to the user-circuits that employ the special-purpose components (datapath, multiplexers).

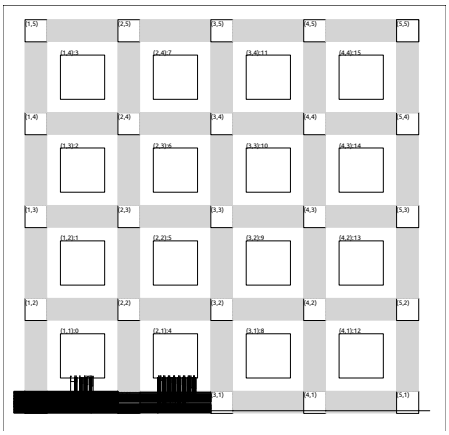
Secondly, the manual mapping outperforms the SAT-solver one for up to 20%. It means that coarse-grained architecture performance may be improved by using a more efficient placement algorithm.



(a) Fine-grained.

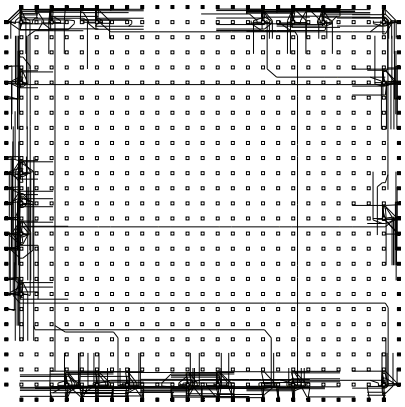


(b) Automatic control placement.

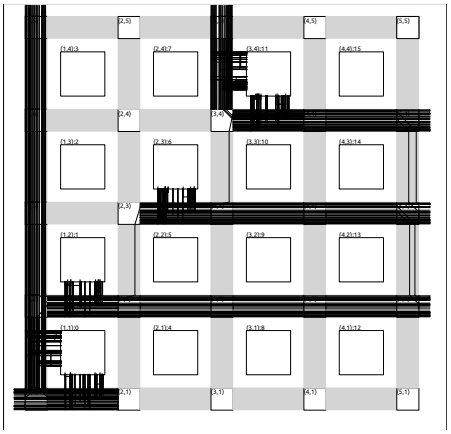


(c) Manual control placement.

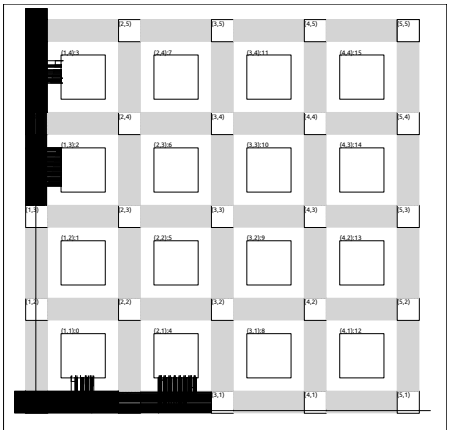
Figure 4.8: Ripple carry adder 32-bit mapping.



(a) Fine-grained.



(b) Automatic Control placement.



(c) Manual control placement.

Figure 4.9: Ripple carry adder 64-bit mapping.

Table 4.5: Relative performance improvement.

	Head Fw Latency	Forward Latency	Throughput	Backward Latency
FIFO	—	2.3	2.1	2.2
Crossbar	—	2.7	3	1.8
Router	2.7	2.9	3.2	2.5

4.2.5 Summary

This section presented the preliminary performance evaluation of the proposed asynchronous reconfigurable fabric for GALS systems. A fine-grained standard FPGA has been adopted as the comparison with the proposed coarse-grained style. Instead of the absolute timing values, a relative comparison is significant. Table 4.5 and Figure 4.10 sum up the gain in performance obtained by the proposed architecture.

Overall, for every performance indicators, the benefits are evident, as throughput is more than doubled and latency is more than halved. The improvement is substantial if the user-circuit exploits the peculiar characteristics of the architecture (DPB and user-controlled multiplexers). In fact, the crossbar and the router benefit the most of these characteristics, as they gain advantage from the datapath structures.

In conclusion, these preliminary results are overall positive, showing up a clear benefit in the use of the coarse-grained architecture over the fine-grained one.

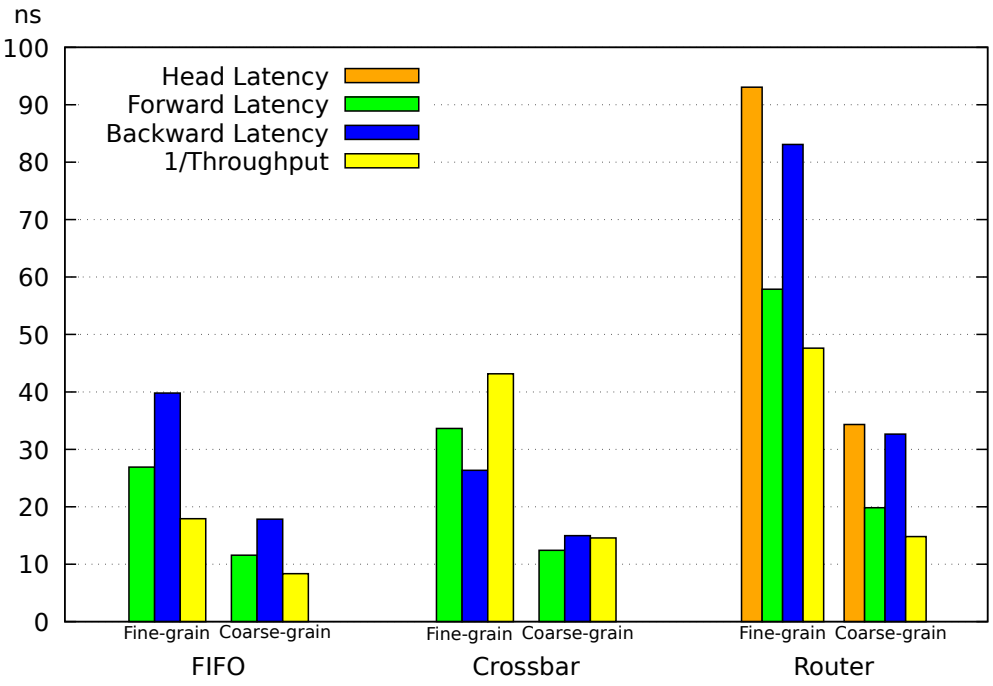


Figure 4.10: Overall results comparison.

Chapter 5

Conclusions

This chapter provides the reader some reflections about the work, an overall sight of what has been done and what can be improved. Furthermore, some future directions are suggested.

Section 5.1 concludes the thesis discussing the outcome. Section 5.2 further analyzes the limitations of this research and the validity of the results, providing guidelines for future work built upon this project.

5.1 Conclusions

The research in this thesis has contributed to GALS FPGA architecture. It proposes a novel asynchronous reconfigurable device made of synchronous FPGAs surrounded by an asynchronous fabric. The circuit aims to map asynchronous NoCs and GALS systems. Support for asynchronous pipeline is embedded in the architecture. A specialized tool flow is developed for the proposed coarse-grained FPGA architecture. This tool enables placement in the coarse-grained CLBs that complies with the architectural constraints. The proposed architecture and the tool flow facilitate the study of the interface between asynchronous and synchronous systems, asynchronous communication optimization, and GALS NoC systems. The key characteristics of the proposed architecture style are separated datapath and control logic, coarse-grained CLBs, and user-controlled multiplexers for datapath.

The main achievement of this work was to prove the effectiveness of the proposed architecture over a standard fine-grained FPGA for communication-purpose asynchronous circuits. The metric used to

evaluate the architecture is the performance. The benefit of the new architecture should be greater enough over a standard architecture to justify the development effort. Performances are measured using the synthesis results from four different benchmark circuits. Three of them are communication circuits that employ asynchronous pipeline. These are the types of circuits that are expected to be implemented on the proposed reconfigurable circuit, so the primary results are extracted from these evaluations. The fourth circuit uses a combinatorial logic, which does not map well onto the proposed architecture. This circuit, in fact, does not exploit any inherent element provided by the architecture.

The proposed architecture is presented in Chapter 3. The proposed CLB is a unique combination of data latches, asynchronous resources, and traditional FPGA BLEs. The architecture is presented in detail and how it implements a circuit is shown.

In Chapter 4 the results of the comparison between coarse-grained and fine-grained style can be found. For the communication-purpose user-circuits, the proposed architecture outperforms the classical one, allowing a throughput improvement up to 3.2x. Surprisingly, the proposed architecture shows slightly better performance respect to the fine-grained one also in the case of combinatorial circuit, up to 27%.

Overall, the results presented in this thesis prove that the proposed architecture performs considerably better than a classical one for asynchronous communication circuits. Thus, this early evaluation can be a starting point for the development of a complete reconfigurable GALS system.

5.2 Limitations and Future Work

Several limitations were encountered during this research, that, however, do not affect the thesis outcome. There could be improvements to this work to address some current issues, and further research based on the architecture proposed in this thesis.

The main focus of this thesis is the reconfigurable asynchronous interconnect between synchronous FPGAs. In future work, the interface between synchronous/asynchronous elements can be explored, to allow the evaluation of the GALS FPGA device as a whole. A fair comparison between the device and other relevant asynchronous FP-

GAs [54][11] should be provided to establish the effective advantage to use the proposed style.

The limitations of the current tool would merit future work since some essential functionalities are still missing. At the current status, it is impossible to handle large and complex user-circuits, due to the manual steps required by the tool. A fully automatic tool flow is required, that would efficiently place each asynchronous pipeline stage in the architecture PSBs. A first attempt in this direction has been made, trying to adapt the VPR simulated annealing approach to the coarse-grained device. However, this attempt has failed, given the intrinsic hierarchical structure of the proposed architecture, as well as the many constraints that limit the possible connections.

The current placement tool for the control portion of the user-circuit is an automatic step that merely places the elements accordingly to architecture constraints. It provides a placement according to a solution generated by a SAT-solver, that is one possible solution to the problem, no optimizations are carried out on it. This placement mechanism can be improved using incremental SAT-solver techniques: placement quality should be estimated using some link cost metrics, and, if it does not meet the requirement, the solution is discarded generating another SAT solution.

Regarding the simulation, some more parameters can be taken into account. Area and power estimations are essential indicators of the quality of a device. This measurement, as well as more accurate delay models, can be carried out after a physical implementation has taken place (e.g., by Cadence Innovus).

In conclusion, proposed GALS FPGA device can be used as a reconfigurable device not only for general purpose NoC, but also as application specific device. In particular, the asynchronous paradigm fits with the concept of neural networks. The newly proposed device can be used to directly map, and consequently speed-up convolutional neural networks.

Bibliography

[1] S. M. Nowick and M. Singh. "Asynchronous Design #x2014;Part 1: Overview and Recent Advances". In: *IEEE Design Test* 32.3 (June 2015), pp. 5–18. ISSN: 2168-2356. DOI: 10.1109/MDAT.2015.2413759.

[2] Ian Kuon, Russell Tessier, and Jonathan Rose. "FPGA Architecture: Survey and Challenges". en. In: *Foundations and Trends® in Electronic Design Automation* 2.2 (2007), pp. 135–253. ISSN: 1551-3939, 1551-3947. DOI: 10.1561/1000000005. URL: http://www.nowpublishers.com/article/Details/EDA-005 (visited on 05/24/2018).

[3] S. M. Trimberger. "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology". In: *Proceedings of the IEEE* 103.3 (Mar. 2015), pp. 318–331. ISSN: 0018-9219. DOI: 10.1109/JPROC.2015.2392104.

[4] Andrew Putnam et al. "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 13–24. ISBN: 978-1-4799-4394-4. URL: http://dl.acm.org/citation.cfm?id=2665671.2665678 (visited on 02/27/2018).

[5] Lei Gong et al. "A Power-efficient and High Performance FPGA Accelerator for Convolutional Neural Networks: Work-in-progress". In: *Proceedings of the Twelfth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis Companion*. CODES '17. New York, NY, USA: ACM, 2017, 16:1–16:2. ISBN: 978-1-4503-5185-0. DOI: 10.1145/3125502.3125534. URL: http://doi.acm.org/10.1145/3125502.3125534 (visited on 02/27/2018).

[6] Jason Cong et al. “Architecture and Synthesis for Multi-cycle Communication”. In: *Proceedings of the 2003 International Symposium on Physical Design*. ISPD ’03. New York, NY, USA: ACM, 2003, pp. 190–196. ISBN: 978-1-58113-650-0. DOI: 10.1145/640000.640040. URL: http://doi.acm.org/10.1145/640000.640040 (visited on 10/10/2017).

[7] Akshay Sharma et al. “Exploration of Pipelined FPGA Interconnect Structures”. In: *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*. FPGA ’04. New York, NY, USA: ACM, 2004, pp. 13–22. ISBN: 978-1-58113-829-0. DOI: 10.1145/968280.968284. URL: http://doi.acm.org/10.1145/968280.968284 (visited on 10/10/2017).

[8] William Tsu et al. “HSRA: High-speed, Hierarchical Synchronous Reconfigurable Array”. In: *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*. FPGA ’99. New York, NY, USA: ACM, 1999, pp. 125–134. ISBN: 978-1-58113-088-1. DOI: 10.1145/296399.296442. URL: http://doi.acm.org/10.1145/296399.296442 (visited on 10/10/2017).

[9] Li Shang, Alireza S. Kaviani, and Kusuma Bathala. “Dynamic Power Consumption in Virtex™-II FPGA Family”. In: *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-programmable Gate Arrays*. FPGA ’02. New York, NY, USA: ACM, 2002, pp. 157–164. ISBN: 978-1-58113-452-0. DOI: 10.1145/503048.503072. URL: http://doi.acm.org/10.1145/503048.503072 (visited on 03/01/2018).

[10] Andrew Royal and Peter Y. K. Cheung. “Globally Asynchronous Locally Synchronous FPGA Architectures”. en. In: *Field Programmable Logic and Application*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Sept. 2003, pp. 355–364. DOI: 10.1007/978-3-540-45234-8_35. URL: https://link.springer.com/chapter/10.1007/978-3-540-45234-8_35 (visited on 10/04/2017).

[11] Xin Jia and R. Vemuri. “A novel asynchronous FPGA architecture design and its performance evaluation”. In: *International Conference on Field Programmable Logic and Applications, 2005.* Aug. 2005, pp. 287–292. DOI: 10.1109/FPL.2005.1515736.

[12] Jason Luu et al. “VTR 7.0: Next Generation Architecture and CAD System for FPGAs”. In: *ACM Trans. Reconfigurable Technol. Syst.* 7.2 (July 2014), 6:1–6:30. ISSN: 1936-7406. DOI: 10.1145/2617593. URL: http://doi.acm.org/10.1145/2617593 (visited on 03/05/2018).

[13] S. M. Nowick and M. Singh. “High-Performance Asynchronous Pipelines: An Overview”. In: *IEEE Design Test of Computers* 28.5 (Sept. 2011), pp. 8–22. ISSN: 0740-7475. DOI: 10.1109/MDT.2011.71.

[14] I. Kuon and J. Rose. “Measuring the Gap Between FPGAs and ASICs”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (Feb. 2007), pp. 203–215. ISSN: 0278-0070. DOI: 10.1109/TCAD.2006.884574.

[15] M. H. Ho et al. “Architecture and Design Flow for a Highly Efficient Structured ASIC”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.3 (Mar. 2013), pp. 424–433. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2012.2190478.

[16] G. E. Moore. “Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.” In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (Sept. 2006), pp. 33–35. ISSN: 1098-4232. DOI: 10.1109/N-SSC.2006.4785860.

[17] Eriko Nurvitadhi et al. “Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?” In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’17. New York, NY, USA: ACM, 2017, pp. 5–14. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021740. URL: http://doi.acm.org/10.1145/3020078.3021740 (visited on 02/28/2018).

[18] D. C. Guterman et al. “An electrically alterable nonvolatile memory cell using a floating-gate structure”. In: *IEEE Transactions on Electron Devices* 26.4 (Apr. 1979), pp. 576–586. ISSN: 0018-9383. DOI: 10.1109/T-ED.1979.19462.

[19] H. C. Hsieh et al. “A 9000-gate user-programmable gate array”. In: *Proceedings of the IEEE 1988 Custom Integrated Circuits Conference*. May 1988, pp. 15.3/1–15.3/7. DOI: 10.1109/CICC.1988.20872.

[20] K. EI-Ajat et al. "A Cmos Electrically Configurable Gate Array". In: *1988 IEEE International Solid-State Circuits Conference, 1988 ISSCC. Digest of Technical Papers*. Feb. 1988, pp. 76–. DOI: 10.1109/ISSCC.1988.663633.

[21] Carl Ebeling et al. "Stratix™ 10 High Performance Routable Clock Networks". In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '16. New York, NY, USA: ACM, 2016, pp. 64–73. ISBN: 978-1-4503-3856-1. DOI: 10.1145/2847263.2847279. URL: http://doi.acm.org/10.1145/2847263.2847279 (visited on 02/28/2018).

[22] Julien Lamoureux and Steven J. E. Wilton. "FPGA Clock Network Architecture: Flexibility vs. Area and Power". In: *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*. FPGA '06. New York, NY, USA: ACM, 2006, pp. 101–108. ISBN: 978-1-59593-292-1. DOI: 10.1145/1117201.1117216. URL: http://doi.acm.org/10.1145/1117201.1117216 (visited on 03/08/2018).

[23] Fei Li et al. "Architecture Evaluation for Power-Efficient FPGAs". In: *in Proc. ACM Intl. Symp. Field-Programmable Gate Arrays*. 2003, pp. 175–184.

[24] Carl Ebeling, Darren C. Cronquist, and Paul Franklin. "RaPiD — Reconfigurable pipelined datapath". en. In: *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Sept. 1996, pp. 126–135. DOI: 10.1007/3-540-61730-2_13. URL: https://link.springer.com/chapter/10.1007/3-540-61730-2_13 (visited on 03/08/2018).

[25] Alexander (Sandy) Marquardt, Vaughn Betz, and Jonathan Rose. "Using Cluster-based Logic Blocks and Timing-driven Packing to Improve FPGA Speed and Density". In: *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*. FPGA '99. New York, NY, USA: ACM, 1999, pp. 37–46. ISBN: 978-1-58113-088-1. DOI: 10.1145/296399.296426. URL: http://doi.acm.org/10.1145/296399.296426 (visited on 03/06/2018).

[26] Deming Chen, Jason Cong, and Peichen Pan. "FPGA design automation: A survey". In: *Foundations and Trends in Electronic Design Automation* 1 (Nov. 2006). DOI: 10.1561/1000000003.

[27] P. R. Panda. "SystemC - a modeling platform supporting multiple design abstractions". In: *International Symposium on System Synthesis (IEEE Cat. No.01EX526)*. 2001, pp. 75–80. DOI: 10.1109/ISSS.2001.156535.

[28] Kristofer Vorwerk. "On the Use of Directed Moves for Placement in VLSI CAD". en. In: (July 2009). URL: https://uwspace.uwaterloo.ca/handle/10012/4528 (visited on 05/24/2018).

[29] 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. New York, NY, USA: ACM, 2011, pp. 227–236. ISBN: 978-1-4503-0554-9. DOI: 10.1145/1950413.1950457. URL: http://doi.acm.org/10.1145/1950413.1950457 (visited on 04/21/2018).

[30] P. Jamieson et al. "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*. May 2010, pp. 149–156. DOI: 10.1109/FCCM.2010.31.

[31] *Berkeley Logic Interchange Format (blif) - Semantic Scholar*. 1992. URL: /paper/Berkeley-Logic-Interchange-Format-(blif)/08a0e4888666cd21c35c47581fd7db249f762ff3 (visited on 03/02/2018).

[32] Robert Brayton and Alan Mishchenko. "ABC: An Academic Industrial-strength Verification Tool". In: *Proceedings of the 22Nd International Conference on Computer Aided Verification*. CAV'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 24–40. ISBN: 978-3-642-14294-9. DOI: 10.1007/978-3-642-14295-6_5. URL: http://dx.doi.org/10.1007/978-3-642-14295-6_5 (visited on 04/21/2018).

[33] Vaughn Betz and Jonathan Rose. "VPR: a new packing, placement and routing tool for FPGA research". en. In: *Field-Programmable Logic and Applications*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Sept. 1997, pp. 213–222. DOI: 10.1007/3-

540-63465-7_226. URL: https://link.springer.com/chapter/10.1007/3-540-63465-7_226 (visited on 05/28/2018).

[34] Jens Sparso and Steve Furber, eds. *Principles of Asynchronous Circuit Design: A Systems Perspective*. en. Springer US, 2001. ISBN: 978-0-7923-7613-2. URL: //www.springer.com/cn/book/9780792376132 (visited on 04/28/2018).

[35] A. A. Chien and V. Karamcheti. "Moore's Law: The First Ending and a New Beginning". In: *Computer* 46.12 (Dec. 2013), pp. 48–53. ISSN: 0018-9162. DOI: 10.1109/MC.2013.431.

[36] K. J. Kuhn et al. "Process Technology Variation". In: *IEEE Transactions on Electron Devices* 58.8 (Aug. 2011), pp. 2197–2208. ISSN: 0018-9383. DOI: 10.1109/TED.2011.2121913.

[37] M. Davies et al. "A 72-Port 10G Ethernet Switch/Router Using Quasi-Delay-Insensitive Asynchronous Design". In: *2014 20th IEEE International Symposium on Asynchronous Circuits and Systems*. May 2014, pp. 103–104. DOI: 10.1109/ASYNC.2014.22.

[38] P. A. Merolla et al. "A million spiking-neuron integrated circuit with a scalable communication network and interface". en. In: *Science* 345.6197 (Aug. 2014), pp. 668–673. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.1254642. URL: http://www.sciencemag.org/cgi/doi/10.1126/science.1254642 (visited on 04/24/2018).

[39] M. Singh et al. "An Adaptively Pipelined Mixed Synchronous-Asynchronous Digital FIR Filter Chip Operating at 1.3 Gigahertz". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 18.7 (July 2010), pp. 1043–1056. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2009.2019660.

[40] J. F. Christmann et al. "Bringing Robustness and Power Efficiency to Autonomous Energy Harvesting Microsystems". In: *2010 IEEE Symposium on Asynchronous Circuits and Systems*. May 2010, pp. 62–71. DOI: 10.1109/ASYNC.2010.19.

[41] K. L. Chang et al. "Synchronous-Logic and Asynchronous-Logic 8051 Microcontroller Cores for Realizing the Internet of Things: A Comparative Study on Dynamic Voltage Scaling and Variation Effects". In: *IEEE Journal on Emerging and Selected Topics in*

Circuits and Systems 3.1 (Mar. 2013), pp. 23–34. ISSN: 2156-3357. DOI: 10.1109/JETCAS.2013.2243031.

[42] C. Vezyrtzis et al. "A Flexible, Event-Driven Digital Filter With Frequency Response Independent of Input Sample Rate". In: *IEEE Journal of Solid-State Circuits* 49.10 (Oct. 2014), pp. 2292–2304. ISSN: 0018-9200. DOI: 10.1109/JSSC.2014.2336532.

[43] P. Shepherd et al. "A robust, wide-temperature data transmission system for space environments". In: *2013 IEEE Aerospace Conference*. Mar. 2013, pp. 1–13. DOI: 10.1109/AERO.2013.6497376.

[44] Marco Vacca, Mariagrazia Graziano, and Maurizio Zamboni. "Asynchronous Solutions for Nanomagnetic Logic Circuits". In: *J. Emerg. Technol. Comput. Syst.* 7.4 (Dec. 2011), 15:1–15:18. ISSN: 1550-4832. DOI: 10.1145/2043643.2043645. URL: http://doi.acm.org/10.1145/2043643.2043645 (visited on 04/24/2018).

[45] V. Mehrotra and D. Boning. "Technology scaling impact of variation on clock skew and interconnect delay". In: *Proceedings of the IEEE 2001 International Interconnect Technology Conference (Cat. No.01EX461)*. June 2001, pp. 122–124. DOI: 10.1109/IITC.2001.930035.

[46] Y. Y. Dai and R. K. Brayton. "Verification and Synthesis of Clock-Gated Circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018), pp. 1–1. ISSN: 0278-0070. DOI: 10.1109/TCAD.2018.2808231.

[47] C. H. Van Berkel, M. B. Josephs, and S. M. Nowick. "Applications of asynchronous circuits". In: *Proceedings of the IEEE* 87.2 (Feb. 1999), pp. 223–233. ISSN: 0018-9219. DOI: 10.1109/5.740016.

[48] Michiel Ligthart et al. "Asynchronous Design Using Commercial HDL Synthesis Tools". In: *Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*. ASYNC '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 114–. ISBN: 978-0-7695-0586-2. URL: http://dl.acm.org/citation.cfm?id=785166.785308 (visited on 04/26/2018).

[49] J. Oberg, J. Plosila, and P. Ellervee. "Automatic synthesis of asynchronous circuits from synchronous RTL descriptions". In: *2005 NORCHIP*. Nov. 2005, pp. 200–205. DOI: 10.1109/NORCHP.2005.1597024.

[50] S. B. Furber and P. Day. "Four-phase micropipeline latch control circuits". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 4.2 (June 1996), pp. 247–253. ISSN: 1063-8210. DOI: 10.1109/92.502196.

[51] I. E. Sutherland. "Micropipelines". In: *Commun. ACM* 32.6 (June 1989), pp. 720–738. ISSN: 0001-0782. DOI: 10.1145/63526.63532. URL: http://doi.acm.org/10.1145/63526.63532 (visited on 04/29/2018).

[52] M. Singh and S. M. Nowick. "MOUSETRAP: High-Speed Transition-Signaling Asynchronous Pipelines". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 15.6 (June 2007), pp. 684–698. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2007.898732.

[53] Andrew Matthew Lines. *Pipelined Asynchronous Circuits*. Report or Paper. Jan. 1998. URL: http://resolver.caltech.edu/CaltechCSTR:1998.cs-tr-95-21 (visited on 05/10/2018).

[54] R. Manohar. "Reconfigurable Asynchronous Logic". In: *IEEE Custom Integrated Circuits Conference 2006*. Sept. 2006, pp. 13–20. DOI: 10.1109/CICC.2006.320939.

[55] E. Beigne and P. Vivet. "Design of on-chip and off-chip interfaces for a GALS NoC architecture". In: *12th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'06)*. Mar. 2006, 10 pp.–183. DOI: 10.1109/ASYNC.2006.16.

[56] T. Chelcea and S. M. Nowick. "Robust interfaces for mixed-timing systems". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12.8 (Aug. 2004), pp. 857–873. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2004.831476.

[57] P. Teehan, M. Greenstreet, and G. Lemieux. "A Survey and Taxonomy of GALS Design Styles". In: *IEEE Design Test of Computers* 24.5 (Sept. 2007), pp. 418–428. ISSN: 0740-7475. DOI: 10.1109/MDT.2007.151.

[58] M. Krstic et al. “Globally Asynchronous, Locally Synchronous Circuits: Overview and Outlook”. In: *IEEE Design Test of Computers* 24.5 (Sept. 2007), pp. 430–441. ISSN: 0740-7475. DOI: 10.1109/MDT.2007.164.

[59] D. M. Chapiro. “Globally-asynchronous locally-synchronous systems”. In: *Ph.D. Thesis* (Oct. 1984). URL: http://adsabs.harvard.edu/abs/1984PhDT.....50C (visited on 03/12/2018).

[60] A. A. Abidi. “Phase Noise and Jitter in CMOS Ring Oscillators”. In: *IEEE Journal of Solid-State Circuits* 41.8 (Aug. 2006), pp. 1803–1816. ISSN: 0018-9200. DOI: 10.1109/JSSC.2006.876206.

[61] X. Fan, M. Krstić, and E. Grass. “Analysis and optimization of pausible clocking based GALS design”. In: *2009 IEEE International Conference on Computer Design*. Oct. 2009, pp. 358–365. DOI: 10.1109/ICCD.2009.5413130.

[62] D. G. Messerschmitt. “Synchronization in digital system design”. In: *IEEE Journal on Selected Areas in Communications* 8.8 (Oct. 1990), pp. 1404–1419. ISSN: 0733-8716. DOI: 10.1109/49.62819.

[63] Y. Thonnart, P. Vivet, and F. Clermidy. “A fully-asynchronous low-power framework for GALS NoC integration”. In: *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*. Mar. 2010, pp. 33–38. DOI: 10.1109/DATE.2010.5457239.

[64] E. Kasapaki et al. “Argo: A Real-Time Network-on-Chip Architecture With an Efficient GALS Implementation”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.2 (Feb. 2016), pp. 479–492. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2015.2405614.

[65] W. Ning, G. Fen, and W. Fei. “Design of a GALS Wrapper for Network on Chip”. In: *2009 WRI World Congress on Computer Science and Information Engineering*. Vol. 3. Mar. 2009, pp. 592–595. DOI: 10.1109/CSIE.2009.520.

[66] A. Ghiribaldi, D. Bertozzi, and S. M. Nowick. “A transition-signaling bundled data NoC switch architecture for cost-effective GALS multicore systems”. In: *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2013, pp. 332–337. DOI: 10.7873/DATE.2013.079.

[67] Philip C. Treleaven, David R. Brownbridge, and Richard P. Hopkins. "Data-Driven and Demand-Driven Computer Architecture". In: *ACM Comput. Surv.* 14.1 (Mar. 1982), pp. 93–143. ISSN: 0360-0300. DOI: 10.1145/356869.356873. URL: http://doi.acm.org/10.1145/356869.356873 (visited on 05/10/2018).

[68] P. Clarke. *CEO Interview: John Lofton Hold of Achronix*. Ed. by EE Times. 2006. URL: https://www.eetimes.com/document.asp?doc_id=127140.

[69] Yoshiya Komatsu, Masanori Hariyama, and Michitaka Kameyama. "Architecture of an Asynchronous FPGA for Handshake-Component-Based Design". In: *IEICE TRANSACTIONS on Information and Systems* E96-D.8 (Aug. 2013), pp. 1632–1644. ISSN: 1745-1361, 0916-8532. URL: http://search.ieice.org/bin/summary.php?id=e96-d_8_1632&category=D&year=2013&lang=E&abst= (visited on 10/04/2017).

[70] Doug Edwards and Andrew Bardsley. "Balsa: An Asynchronous Hardware Synthesis Language". en. In: *The Computer Journal* 45.1 (Jan. 2002), pp. 12–18. ISSN: 0010-4620. DOI: 10.1093/comjnl/45.1.12. URL: https://academic.oup.com/comjnl/article/45/1/12/338521 (visited on 05/12/2018).

[71] Vaughn Betz, Jonathan Rose, and Alexander Marquardt, eds. *Architecture and CAD for Deep-Submicron FPGAs*. Norwell, MA, USA: Kluwer Academic Publishers, 1999. ISBN: 978-0-7923-8460-1.

[72] Wen-Chung Tsai et al. "Networks on Chips: Structure and Design Methodologies". In: *J. Electrical and Computer Engineering* 2012 (Jan. 2012). DOI: 10.1155/2012/509465.

[73] G. Lemieux et al. "Directional and single-driver wires in FPGA interconnect". In: *Proceedings. 2004 IEEE International Conference on Field- Programmable Technology (IEEE Cat. No.04EX921)*. Dec. 2004, pp. 41–48. DOI: 10.1109/FPT.2004.1393249.

[74] Guido Rossum. *Python Reference Manual*. Tech. rep. Amsterdam, The Netherlands, The Netherlands: CWI (Centre for Mathematics and Computer Science), 1995.

[75] David Grant, Chris Wang, and Guy G.F. Lemieux. "A CAD Framework for Malibu: An FPGA with Time-multiplexed Coarse-grained Elements". In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '11. New York, NY, USA: ACM, 2011, pp. 123–132. ISBN: 978-1-4503-0554-9. DOI: 10.1145/1950413.1950441. URL: http://doi.acm.org/10.1145/1950413.1950441 (visited on 05/25/2018).

[76] Jun Gu et al. "Algorithms for the Satisfiability (SAT) Problem". en. In: *Handbook of Combinatorial Optimization*. Springer, Boston, MA, 1999, pp. 379–572. DOI: 10.1007/978-1-4757-3023-4_7. URL: https://link.springer.com/chapter/10.1007/978-1-4757-3023-4_7 (visited on 05/25/2018).

[77] Niklas Eén and Niklas Sörensson. "An Extensible SAT-solver". en. In: *Theory and Applications of Satisfiability Testing*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, May 2003, pp. 502–518. DOI: 10.1007/978-3-540-24605-3_37. URL: https://link.springer.com/chapter/10.1007/978-3-540-24605-3_37 (visited on 05/24/2018).

[78] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. "Ultimately Incremental SAT". en. In: *Theory and Applications of Satisfiability Testing – SAT 2014*. Lecture Notes in Computer Science. Springer, Cham, July 2014, pp. 206–218. DOI: 10.1007/978-3-319-09284-3_16. URL: https://link.springer.com/chapter/10.1007/978-3-319-09284-3_16 (visited on 05/25/2018).

[79] L. McMurchie and C. Ebeling. "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs". In: *Third International ACM Symposium on Field-Programmable Gate Arrays*. 1995, pp. 111–117. DOI: 10.1109/FPGA.1995.242049.

[80] Vaughn Betz and Jonathan Rose. "Directional Bias and Non-uniformity in FPGA Global Routing Architectures". In: *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*. ICCAD '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 652–659. ISBN: 978-0-8186-7597-3. URL: http://dl.acm.org/citation.cfm?id=244522.244948 (visited on 05/24/2018).

[81] Steven Joseph Edward Wilton. "Architectures and Algorithms for Field-programmable Gate Arrays with Embedded Memory". PhD Thesis. Toronto, Ont., Canada, Canada: University of Toronto, 1997.

[82] 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*. Apr. 2012, pp. 93–96. DOI: 10.1109/FCCM.2012.25.

Appendix A

Encoding the architectural constraints

This appendix presents the technique used for encoding the architectural constraints into clauses for a boolean SAT problem. As explained in the body of the thesis (Section 3.3.2), this step is part of the placement tool, required to correctly place the LEs into FBs such that the subsequently routing step is possible.

The problem should be encoded as a boolean formula in CNF, i.e., in a conjunction of clauses, where the clauses are a disjunction of literals (asserted or negated variables). Given the formula, a popular SAT-solver is used for extracting a solution, MiniSat [77]. An example of CNF boolean formula is the following:

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge \neg x_1$$

formed by three variables and three clauses. It is easy to verify that it is satisfied by $x_1 = False, x_2 = False, x_3 = (arbitrarily)$. Note that the solution is not unique as x_3 can assume an arbitrary value.

A.1 Variables

The SAT variables for the placement problem define the LE position in a PSB. The variable name consists of two parts. The first one is fixed and uniquely identifies the PSB location and the LE name; the second one specifies the position inside the PSB. Each LE appears in the formula with 15 variables, of which only four must be true to uniquely

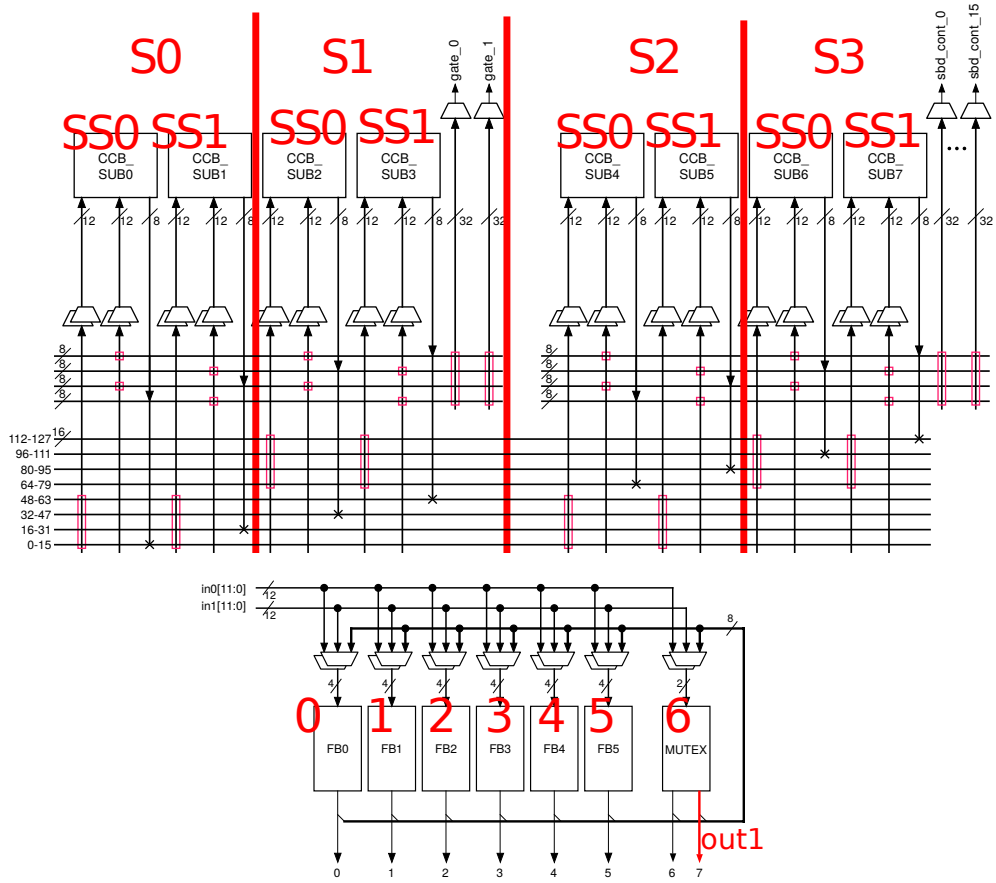


Figure A.1: SAT-solver variables representation.

specify the position of each LE into a PSB. Referring to Figure A.1, these variables are:

- (PSB_location)_(LE_name) | S*m*, with *m=0...3*
- (PSB_location)_(LE_name) | SS*n*, with *n=0,1*
- (PSB_location)_(LE_name) | *k*, with *k=0...6*
- (PSB_location)_(LE_name) | out*p*, with *p=0,1*

The variable (PSB_location)_(LE_name) | out*p* indicates the output of the mutex element, so the *true* variable is always out*0*, except the cases where mutex outputs need to be placed.

A.2 Clauses

Encoding the constrains means to find a disjunction of variables that represent the architectural constraint, and then add it as a clause to the boolean formula.

A trivial constrain states that each LE can be only in one place, then the clauses to add are the following (given A=(PSB_location) _(LE_name)).

- At least one location should be selected:

$$(A|S0 \vee A|S1 \vee A|S2 \vee A|S3) \wedge$$
$$(A|SS0 \vee A|SS1) \wedge$$
$$(A|0 \vee A|1 \vee A|2 \vee A|3 \vee A|4 \vee A|5 \vee A|6)$$

- No more than one location should be selected:

$$(\neg A|S0 \vee \neg A|S1) \wedge (\neg A|S0 \vee \neg A|S2) \wedge (\neg A|S0 \vee \neg A|S3) \wedge$$
$$(\neg A|S1 \vee \neg A|S2) \wedge (\neg A|S1 \vee \neg A|S3) \wedge (\neg A|S2 \vee \neg A|S3) \wedge$$
$$(\neg A|SS0 \vee \neg A|SS1) \wedge$$
$$(\neg A|0 \vee \neg A|1) \wedge (\neg A|0 \vee \neg A|2) \wedge (\neg A|0 \vee \neg A|3) \wedge$$
$$(\neg A|0 \vee \neg A|4) \wedge (\neg A|0 \vee \neg A|5) \wedge (\neg A|0 \vee \neg A|6) \wedge$$
$$(\neg A|1 \vee \neg A|2) \wedge (\neg A|1 \vee \neg A|3) \wedge (\neg A|1 \vee \neg A|4) \wedge$$
$$(\neg A|1 \vee \neg A|5) \wedge (\neg A|1 \vee \neg A|6) \wedge (\neg A|2 \vee \neg A|3) \wedge$$
$$(\neg A|2 \vee \neg A|4) \wedge (\neg A|2 \vee \neg A|5) \wedge (\neg A|2 \vee \neg A|6) \wedge$$
$$(\neg A|3 \vee \neg A|4) \wedge (\neg A|3 \vee \neg A|5) \wedge (\neg A|3 \vee \neg A|6) \wedge$$
$$(\neg A|4 \vee \neg A|5) \wedge (\neg A|4 \vee \neg A|6) \wedge (\neg A|5 \vee \neg A|6)$$

Other constraints come from relative position between two LEs in the same PSB. Given two LEs, which we denote with A and B, the following clauses should be true.

- A and B cannot be placed in the same FB:

$$
\begin{aligned}
&(\neg A|S0 \vee \neg B|S0 \vee \neg A|SS0 \vee \neg B|SS0 \vee \neg A|0 \vee \neg B|0) \wedge \\
&(\neg A|S0 \vee \neg B|S0 \vee \neg A|SS0 \vee \neg B|SS0 \vee \neg A|1 \vee \neg B|1) \wedge \\
&\dots \\
&(\neg A|S0 \vee \neg B|S0 \vee \neg A|SS1 \vee \neg B|SS1 \vee \neg A|0 \vee \neg B|0) \wedge \\
&\dots \\
&(\neg A|S1 \vee \neg B|S1 \vee \neg A|SS0 \vee \neg B|SS0 \vee \neg A|0 \vee \neg B|0) \wedge \\
&\dots \\
&(\neg A|S3 \vee \neg B|S3 \vee \neg A|SS1 \vee \neg B|SS1 \vee \neg A|6 \vee \neg B|6) \wedge
\end{aligned}
$$

- If a connection from A to B exists, then the connection must be guaranteed:

$$
\begin{aligned}
&(A|S0 \wedge A|SS0 \rightarrow B|S0 \vee (B|S1 \wedge B|SS1) \vee B|S2) \wedge \\
&(A|S0 \wedge A|SS1 \rightarrow B|S0 \vee (B|S1 \wedge B|SS0) \vee B|S2) \wedge \\
&(A|S1 \rightarrow B|S0 \vee B|S1 \vee B|S2) \wedge \\
&(A|S2 \rightarrow B|S1 \vee B|S2 \vee B|S3) \wedge \\
&(A|S3 \wedge A|SS0 \rightarrow B|S1 \vee (B|S2 \wedge B|SS1) \vee B|S3) \wedge \\
&(A|S3 \wedge A|SS1 \rightarrow B|S1 \vee (B|S2 \wedge B|SS0) \vee B|S3)
\end{aligned}
$$

The last constraints come from connections between LEs in different PSBs. The clauses to add to the problem, in this case, are the following.

- If a connection from A to B exists, then the connection must be guaranteed:

$$
\begin{aligned}
&(A|S0 \rightarrow (B|S0 \vee B|S2) \wedge A|\mathit{outp}) \wedge \\
&(A|S1 \rightarrow (B|S0 \vee B|S2) \wedge A|\mathit{outp}) \wedge \\
&(A|S2 \rightarrow (B|S1 \vee B|S3) \wedge A|\mathit{outp}) \wedge \\
&(A|S3 \rightarrow (B|S1 \vee B|S3) \wedge A|\mathit{outp})
\end{aligned}
$$

- If the route of a connection from A to B intersects another connection from C to D, then the source LEs must be in different relative position within their PSB:

$$
\begin{aligned}
&(\neg A|S0 \vee \neg C|S0 \vee \neg A|SS0 \vee \neg C|SS0 \vee \neg A|0 \vee \neg C|0) \wedge \\
&(\neg A|S0 \vee \neg C|S0 \vee \neg A|SS0 \vee \neg C|SS0 \vee \neg A|1 \vee \neg C|1) \wedge \\
&\dots \\
&(\neg A|S0 \vee \neg C|S0 \vee \neg A|SS1 \vee \neg C|SS1 \vee \neg A|0 \vee \neg C|0) \wedge \\
&\dots \\
&(\neg A|S1 \vee \neg C|S1 \vee \neg A|SS0 \vee \neg C|SS0 \vee \neg A|0 \vee \neg C|0) \wedge \\
&\dots \\
&(\neg A|S3 \vee \neg C|S3 \vee \neg A|SS1 \vee \neg C|SS1 \vee \neg A|6 \vee \neg C|6) \wedge
\end{aligned}
$$

A.3 Statistics

The SAT problem is an NP-complete problem, that means that each problem in the NP complexity class is at most as complex as the SAT. Therefore, in general, no efficient algorithm to solve the SAT problem exists (assuming that P$\neq$NP). Nevertheless, MiniSat algorithm solves the problem in our specific case in an acceptable amount of time.

For the NoC router example of Section 4.2.3, on a 4x4 coarse-grained array, the constraints encoding results in about 5000 variables and 1.2 million clauses. The SAT-solver run-time to find the first solution (i.e., no incremental solving used) on a Core i7 @4GHz is about 28 seconds, that is an acceptable amount of time.