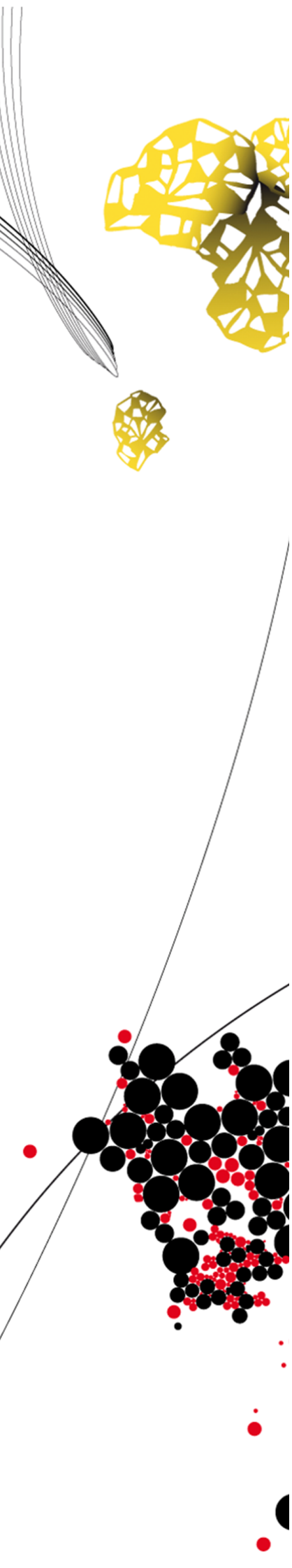

UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering,
Mathematics & Computer Science**

Framework for Fine-Grained Partial Reconfiguration on FPGAs

Tom Hogenkamp

M.Sc. Thesis
October 2019

Supervisors:
dr.ing. D.M. Ziener
Madiha Sheikh
Ali Asghar

Computer Architecture for Embedded Systems
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

*Field-Programmable Gate Array*s (FPGAs) are semiconductor devices that contain programmable logic blocks and interconnection circuits. An FPGA can be programmed or reprogrammed to the required functionality after manufacturing. *Dynamic Partial Reconfiguration* (DPR) is a feature of FPGA devices that enables us to change only a part of its configuration memory during run-time and not alter the rest of the system. Normally, when using this feature, the FPGA fabric is separated into two areas: static and partial. The static area of the FPGA fabric is used to implement the functionality that is required all the time, where the partial area of the FPGA fabric is used to configure functional blocks (modules) that can be used in a time-multiplexed manner.

The leading FPGA manufactures, Xilinx and Intel, support DPR in their development tools. By using these tools, we can configure only one module into the partial area at a time. As a consequence, we cannot use the unutilized resources if a relatively small module is configured within the partial area. Also, the complete partial area is reconfigured despite the size of the module. Therefore, the time to reconfigure a small module requires the same amount of time in comparison to a large module. The reason is that the reconfiguration time is proportional to the area being reconfigured. Another disadvantage of the vendor tools is that module relocation is not supported. Module relocation means that the same module can be configured onto multiple locations. This feature also allows us to instantiate a module multiple times on the fabric of the FPGA.

In this work, we present a framework that overcomes the limitations of the vendor tools. The framework supports the configuration of multiple modules in the partial area simultaneously. Therefore, a large module can be replaced by multiple small modules. Also, in this framework, we reconfigure only the resources that are required by the modules. As a result, we minimize the reconfiguration time. Finally, module relocation is supported.

The academia presented some DPR architectures that divide the partial area into two-dimensional slots. This construction is called grid-style reconfiguration. Grid-style reconfiguration enhances the utilization efficiency of the resources within the partial area. The reason is that multiple modules can be configured within the partial

area at the same time, and these modules occupy one or multiple slots according to their resource requirements. Furthermore, in grid-style reconfiguration, only the slots that are used by the modules are reconfigured. As a result, the reconfiguration time is minimized, since not the complete partial area is reconfigured, but only the slots that are occupied by the modules. Also, module relocation among slots is feasible, which makes the placement of the modules in the grid very flexible.

The most challenging in grid-style systems is to establish communication between the static area and the partial area, and module-to-module communication. The current academic tools that support grid-style reconfiguration are GoAhead and Dreams. Both tools have their disadvantages. In GoAhead, the limits concern its communication architecture, where Dreams has several restrictions in its design flow. In this work, we adapt the communication architecture from Dreams and use the GoAhead design flow to implement the grid-style system. We extend the GoAhead tool such that we can implement the communication architecture of Dreams by using the GoAhead tool.

At the beginning of our design flow, we use GoAhead to generate design templates and constraint files. The design templates must be merged within the existing design files. Furthermore, the constraint files must be included in one of the vendor tools to incorporate with the low-level device-dependent operations. The final result is a full bitstream and a various number of partial bitstreams. The full bitstream represents the static system and should be configured on the FPGA first. Then, during run-time, modules can be configured on the FPGA by using the partial bitstreams.

A case study demonstrates the framework. The aim of this case study is a countermeasure against physical attacks. Usually, the goal of these physical attacks is to extract the secret key from a cryptographic implementation on the FPGA. These attacks are based on analyzing characteristics of a hardware implementation, such as timing information, power consumption, or electromagnetic leaks. Now, by using DPR, we reconfigure the cryptographic implementation continuously with its variants. These variants have the same functionality but have a different hardware implementation. Consequently, the characteristics of the hardware implementation become random, and therefore, the physical attacks become more difficult or even impossible.

The result of this work is a development tool that enables us to use DPR more efficiently. The system allows us to configure multiple modules in the partial at the same time. Also, module relocation is supported, which gives us a lot of flexibility in the placement of the modules. Finally, the reconfiguration time is minimized, since we only reconfigure the slots that are occupied by the modules. For future work, more steps in the design flow could be automated, and support for simulation should be added.

Contents

Summary iii

List of acronyms vii

1 Introduction 1
- 1.1 Problem Description . . . 3
- 1.2 Report Organization . . . 4

2 Background 5
- 2.1 Field-Programmable Gate Arrays . . . 5
  - 2.1.1 General Architecture of FPGAs . . . 5
  - 2.1.2 General Design Flow of FPGAs . . . 8
  - 2.1.3 Xilinx FPGA Terminology . . . 9
- 2.2 Dynamic Partial Reconfiguration . . . 14
  - 2.2.1 Benefits of DPR . . . 14
  - 2.2.2 DPR Terminology . . . 16
  - 2.2.3 Commercial DPR Tools . . . 19

3 Related Work 23
- 3.1 Academic DPR Tools . . . 23
- 3.2 Comparison of DPR Tools . . . 26

4 Concept for Grid-Style Partial Reconfigurable System 29
- 4.1 Limits of the Current DPR Tools . . . 29
- 4.2 Proposed Grid-Style DPR System . . . 36
  - 4.2.1 Static System . . . 37
  - 4.2.2 Modules . . . 41
  - 4.2.3 Planning Phase . . . 43
  - 4.2.4 Tool Flow . . . 48

5 Implementation 49
- 5.1 Basics of GoAhead . . . 51

5.2 Implementing the Static System 54
5.2.1 Synchronous Systems 55
5.2.2 Interface Constraints 57
5.2.3 VHDL Templates 68
5.2.4 Placement Constraints 70
5.2.5 The Blocker Macro 72
5.2.6 Implementation in Vivado 76
5.3 Implementing the Modules 78
5.3.1 Implementation in GoAhead 78
5.3.2 Implementation in Vivado 84
5.3.3 Creating Partial Bitstreams 84

6 Case Study: AES **87**
6.1 AES Hardware Implementation 88
6.2 Static/Partial Partitioning 89
6.3 Static System 90
6.4 Modules 95
6.5 Configurations 97
6.6 Results 98

7 Conclusions and Recommendations **101**
7.1 Conclusions 101
7.2 Recommendations 101

References **103**

List of acronyms

AES *Advanced Encryption Standard*

API *Application Programming Interface*

ASIC *Application-Specific Integrated Circuit*

BEL *Basic Element*

BLE *Basic Logic Element*

BRAM *Block RAM*

CAD *Computer-Aided Design*

CLB *Configurable Logic Block*

DPR *Dynamic Partial Reconfiguration*

DR *Dynamic Reconfiguration*

DSP *Digital Signal Processing*

FF *Flip-Flop*

FPGA *Field-Programmable Gate Array*

GUI *Graphical User Interface*

HDL *Hardware Description Language*

I/O *Input/Output*

ICAP *Internal Configuration Access Port*

INT *Interconnection*

IOB *Input/Output Block*

LUT *Lookup Table*

PIP *Programmable Interconnect Point*

PRC *Partial Reconfiguration Controller*

SDR *Software-Defined Radio*

SEU *Single Event Upsets*

SRAM *Static Random-Access Memory*

TCL *Tool Command Language*

VHDL *VHSIC Hardware Description Language*

XDL *Xilinx Design Language*

Chapter 1

Introduction

*Field-Programmable Gate Array*s (FPGAs) are flexible general-purpose electronic devices that can be used to implement digital circuits. FPGAs are composed of programmable logic blocks and routing interconnections. The logic blocks host logic functions, where the routing interconnections connect these logic functions to build large systems. The FPGA vendors offer *Computer-Aided Design* (CAD) tools to develop custom applications for their FPGAs. Usually, the digital systems are described by using a *Hardware Description Language* (HDL). Once the design is finished, the CAD tools are used to translate the described digital system into a *bitstream*. The bitstream contains the information for all the programmable logic blocks and routing interconnections and can be loaded on the FPGA by using one of the configuration interfaces.

In the early days of FPGAs, the available resources were limited. Therefore, *Dynamic Reconfiguration* (DR) was suggested. In this approach, the configuration memory is reconfigured during run-time. This allows us to build larger systems on fewer resources. The reason is that in this technique, the resources on the FPGA are used in a time-multiplexed manner. By configuring only the functional blocks that are required at a certain point in time, we can build systems on fewer resources.

In DR, the complete FPGA is reconfigured. This has some disadvantages. First of all, we require an external controller to reconfigure the FPGA. This controller determines when and which bitstream is configured into the FPGA. Furthermore, reconfiguring the complete FPGA erases all the memory bits. Therefore, the status of state machines or any other data that should be retained must be stored in external memory before reconfiguring. Once the reconfiguration is finished, we have to restore the status. Finally, the reconfiguration is rather slow since we reconfigure the complete FPGA. The reason is that the reconfiguration time is proportional to the size of the area being reconfigured.

In modern FPGAs, the configuration memory can be reconfigured in small portions. Hereby, the rest of the system is not altered. Therefore, we can split the

FPGA fabric into two regions: static and reconfigurable. In the *static region*, the logic remains the same during run-time, where the *reconfigurable region* is used to configure functional blocks (*modules*) that are needed at a certain moment. The run-time reconfiguration on only a part of the FPGA fabric is called *Dynamic Partial Reconfiguration* (DPR).

DPR offers some huge advantages over DR. By using DPR, we can build systems that can modify themselves autonomously. We can do this by locating the controller in the static region of the FPGA fabric. The controller can use an internal configuration interface to reconfigure the reconfigurable region on the FPGA structure. Furthermore, the down-time of the system due to reconfiguration decreases significantly, since we reconfigure only a part of the whole FPGA fabric. In the literature, there are many applications demonstrated that benefit from DPR. One of them is the instruction set architecture of a soft-processor. By using DPR, the system can substantially enhance performance and area at the same time. Other examples are database acceleration and security applications.

Previously, we have seen how we partition the FPGA fabric into two separate regions in DPR. Now, we discuss the reconfigurable region in more detail. Usually, we call the reconfigurable region the *partial area*. A system might provide multiple partial areas. We can categorize the partial area in different reconfiguration styles. The simplest form of reconfiguration is *island-style*. In this style, the partial area can host only one module at the same time. It is not feasible to configure multiple modules simultaneously, even if there are unutilized resources within the partial area. The resources that are available but cannot be used due to this method is called *internal fragmentation*.

We can distinguish the island-style reconfiguration into two sub-categories: single and multi island-style. In *single island-style*, a set of modules can only be configured within one specific partial area, where *multi island-style* supports the placement of a single module into two or more partial areas. This is called module relocation. Module relocation means that the same module can be configured in different locations onto the FPGA fabric. Also, module relocation makes it possible to instantiate a single module multiple times on the FPGA structure.

A more advanced reconfiguration style is *slot-style*. In slot-style, the partial area is partitioned into one-dimensional slots. Modules can occupy the number of slots according to their resource requirements, and multiple modules can be configured within the partial area simultaneously. This style solves some problems that we had in the island-style approach. Namely, by dividing the partial area into slots, we decrease the internal fragmentation significantly. The reason is that the modules only occupy the number of slots according to their resource requirements, and leave the other slots free for other modules. Note that there is usually still a small portion of

internal fragmentation since the modules might require only a part of the resources in a slot. Therefore, the more fine-grained the partial area, the less the internal fragmentation.

The most advanced reconfiguration style is *grid-style*. This style is similar to slot-style. However, in grid-style, the partial area is partitioned into two-dimensional slots. This allows us to build even more fine-grained partial areas and thus reduce the internal fragmentation even more.

In the following, we discuss the existing tools to develop DPR applications. The two leading FPGA vendors, Xilinx and Intel, provide CAD tools to develop DPR applications. The design flow of these tools is very similar, and therefore, they have the same restrictions.

The only reconfiguration style that these tools support is island-style. More precisely, module relocation is not supported, and therefore, only applications with single island-style reconfiguration can be developed. Furthermore, the development of the modules is dependent on the static region. As a consequence, any change in the static region requires a complete reimplementation of the modules. Finally, the way they implement the communication architecture between the static region and the modules causes logic overhead.

1.1 Problem Description

As we have seen, the DPR tools have some significant limitations and in particular, due to the single island-style reconfiguration. We cannot configure multiple modules in the partial area, which causes a considerable amount of internal fragmentation if we have modules with substantial differences in resource requirements. Furthermore, the whole partial area is reconfigured in island-style, despite the size of the modules. As a consequence, the time to reconfigure a small module takes the same amount of time in comparison to a large module. Especially, applications that require a fast context switch, it is essential that the reconfiguration time is as short as possible (e.g., database acceleration).

Another disadvantage is that module relocation is not supported. Therefore, if we would like to configure the same module in multiple partial areas, we require for each partial area a separate bitstream, even if the size and footprint of the partial areas are exactly the same. As a disadvantage, the development time increases since we have to generate more bitstreams. Also, we require additional memory to store all the bitstreams.

In this work, we present a development tool that enables us to build very flexible and fine-grained DPR systems. By using a fine-grained DPR system, we can configure multiple modules at the same moment on a smaller area, since the inter-

nal fragmentation is minimal. As a result, we can implement the system on smaller devices with fewer resources, such as internet of things devices. Another essential advantage of a fine-grained DPR system is that we minimize the area to reconfigure. Therefore, we reduce the reconfiguration time. The reason is that we only have to configure the slots that are occupied by the modules and not the whole partial area.

1.2 Report Organization

This section describes the structure of the report. In Chapter 2, we provide background information for the reader that is relevant for this research. The main topics are FPGAs and DPR. More precisely, we describe the architecture of FPGAs in more detail and introduce some terminology related to DPR. Furthermore, we have a closer look at the implementation of the DPR applications by the vendor tools.

In the following, we discuss related work from academia. We describe the relevant academic DPR tools and how they have overcome some of the limitations of the vendor tools. Furthermore, we compare all the academic and vendor tools. This is all part of Chapter 3.

In Chapter 4, we introduce our proposed DPR system and design flow. The implementation of this system is described in Chapter 5. In the following, we demonstrate our framework with a case study. This is part of Chapter 6. Finally, in Chapter 7, we conclude the research and provide several recommendations for future work.

Chapter 2

Background

This chapter provides the reader with the background information that is relevant for this research. This chapter is organized as follows. In Section 2.1, we describe first the general architecture and the design flow of FPGAs. At the end of this section, we take a closer view at the FPGA architectures of Xilinx devices. DPR on FPGAs is discussed in Section 2.2. In this section, we describe the potential benefits from using DPR on FPGAs. Furthermore, some terminology related to DPR is introduced. The section concludes with a description of the current DPR development tools provided by the vendor tools and their limitations.

2.1 Field-Programmable Gate Arrays

*Field-Programmable Gate Array*s (FPGAs) are semiconductor devices that are widely used in electronic circuits. These devices are composed of programmable logic and routing interconnections that can be programmed to implement digital designs. An *Application-Specific Integrated Circuit* (ASIC) is similar to an FPGA, with the exception that it is fabricated as a custom circuit. In contrast to ASICs, FPGAs are reprogrammable. This feature makes FPGAs very flexible and general-purpose. However, due to this feature, it makes them larger, slower, and more power-consuming in comparison to ASICs. FPGA-based systems have lower development costs and faster time-to-market compared to ASICs, which makes FPGAs very attractive to use for small to medium volume productions.

2.1.1 General Architecture of FPGAs

A generalized architecture of FPGAs is shown in Figure 2.1. An FPGA is arranged in the form of a two-dimensional array consisting of the following elements.

- *Configurable Logic Block*s (CLBs) that implement logic functions.

- Programmable routing interconnections that connect these logic functions.
- *Input/Output Block*s (IOBs) that are connected to logic blocks through routing interconnects and make off-chip connections.

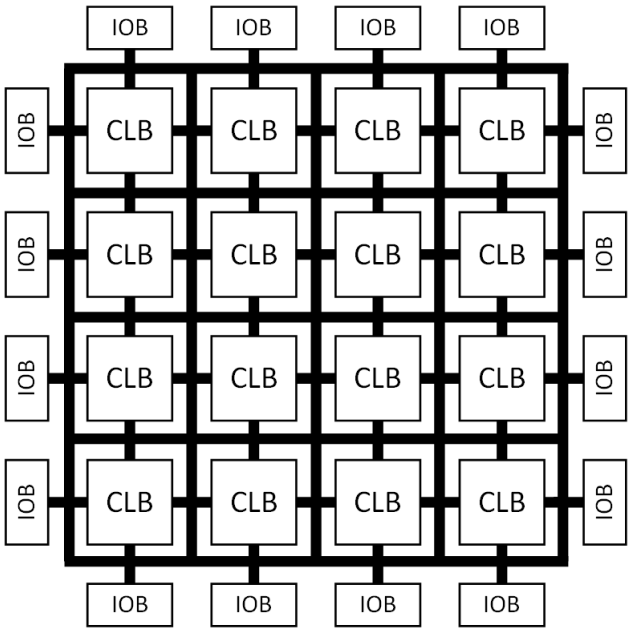

Figure 2.1: An FPGA comprises of CLBs, IOBs, and programmable routing interconnections. Logic functions are implemented on CLBs, where multiple CLBs are connected through the routing interconnections. The IOBs provide functionality for off-chip connections.

A CLB is a fundamental component of an FPGA that provides basic computation and storage elements in digital systems. CLBs should be a good trade-off between too fine-grained and too coarse-grained logic blocks. On the one hand, too fine-grained would require a lot of routing resources, which will suffer from area-inefficiency, low performance, and high power consumption. On the other hand, too coarse-grained would lead to a waste of resources if we implement small functions on the CLB. Therefore, commercial FPGA vendors use *Lookup Table* (LUT) based CLBs, as they provide a good trade-off between too fine-grained and too coarse-grained logic blocks. In the purest form, the LUT comes in combination with a flip-flop and multiplexer. This combination is called a *Basic Logic Element* (BLE). A CLB can comprise of a single BLE or a cluster of locally interconnected BLEs.

Figure 2.2 illustrates a single BLE. A LUT with k inputs contains 2^k *Static Random-Access Memory* (SRAM) cells. In this figure, the SRAM cells can be programmed to implement any four inputs boolean function. The output of the LUT is connected to an optional *Flip-Flop* (FF) to implement synchronous circuits. The

configuration in the SRAM cell connected to the multiplexer determines the output of the BLE. The multiplexer selects the BLE output to be either the output of the FF or the LUT. Modern FPGAs typically contain 4 to 10 BLEs in a CLB.

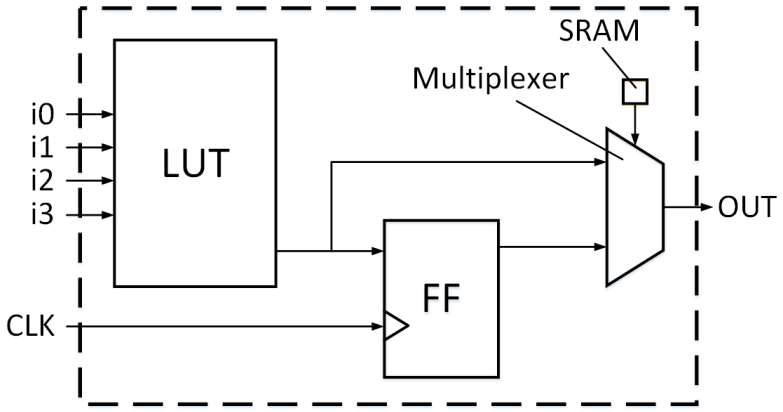

Figure 2.2: A BLE contains a LUT that can be programmed to implement any k-input boolean function. The FF is used to implement synchronous logic.

In Figure 2.1, the architecture is homogeneous. However, modern FPGAs consist of a heterogeneous mixture of logic blocks. Besides the LUT-based CLBs, the architecture contains other logic blocks for specific purposes. These particular purposes blocks, also referred to as *hard blocks*, include *Block RAM* (BRAM) and *Digital Signal Processing* (DSP) blocks. BRAM is used to store large amounts of data, where DSP blocks perform complex arithmetic operations. Hard blocks are very efficient at implementing particular functions as they are designed optimally to perform these functions. However, they end up wasting considerable amounts of logic and routing resources if they remain unused.

The programmable routing interconnections provide connections among CLBs and IOBs to implement any user-defined circuit. The routing network consists of wires and programmable switches that can be programmed to form the required link. The routing interconnections must be very flexible so that they can accommodate a wide variety of circuits with widely varying routing demands.

The IOBs provide off-chip connections. As there are a lot of interface standards, the IOBs have to interface at many different speeds and voltages with the full range of external components that may connect to an FPGA. Modern FPGAs use an *Input/Output* (I/O) banking scheme in which I/O cells are grouped into predefined banks. Each bank shares supply and reference voltage supplies. Therefore, a single bank cannot support all the standards simultaneously, but different banks can have various supplies to support otherwise incompatible standards.

2.1.2 General Design Flow of FPGAs

Computer-Aided Design (CAD) tools are used to design digital circuits for FPGA devices. These tools bridge the gap between the low-level implementation details of FPGAs and describing digital circuits at a higher abstraction level. Figure 2.3 shows the general design flow of these CAD tools.

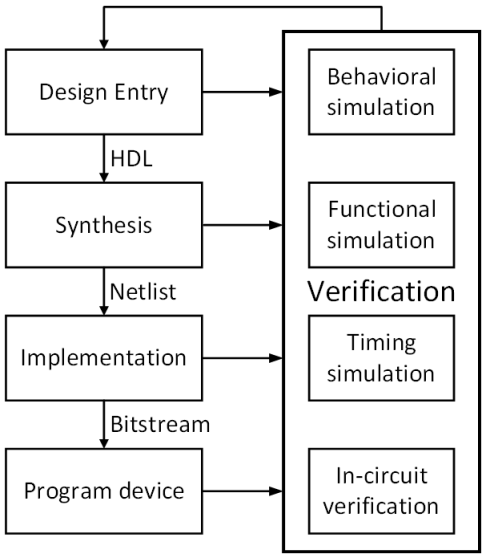

Figure 2.3: The general design flow that is used to develop applications for FPGAs.

The *design entry* is the starting point to design a digital circuit for the target FPGA. The functionality of the digital design can be described by using various techniques, such as schematics or a *Hardware Description Language* (HDL). The two most common HDLs are Verilog and *VHSIC Hardware Description Language* (VHDL). Schemas and HDLs can also be combined to describe the digital system. The behavior of the design can be verified by performing a *behavioral simulation*. Usually, a test bench is written (in an HDL) to simulate the design. The test bench drives the inputs of the model and compares the outputs of the model with the expected outputs.

The design is synthesized once the design passed the behavioral simulation successfully. In *synthesis*, the design is translated into an actual circuit with logical elements (e.g., LUTs and FFs) and their connectivity, which is called a *netlist*. The netlist can be verified by performing a functional simulation, where the same test bench can be used, as in the behavioral simulation.

The *implementation* is separated into three parts: translation, mapping, and place and route. The translation process merges all the netlists and design constraint information into one large netlist. The design constraints can be regarding pin assignments or timing requirements. The mapping process maps the translated netlist to the target FPGA. Finally, the mapped netlist is placed and routed onto

the FPGA fabric. After implementation, a timing simulation can be performed. This simulation gives the most accurate impression of the design behavior.

Once all the simulations have passed successfully, the *bitstream* is generated. The bitstream contains the information for all configuration cells of an FPGA to be programmed to either 0 or 1. Finally, the bitstream is configured to the FPGA by using one of the configuration interfaces.

2.1.3 Xilinx FPGA Terminology

Xilinx is one of the leading providers of FPGA devices and sells a large number of different FPGAs. The FPGAs of Xilinx can be categorized into *series*, *families*, and individual *parts*. At the highest level, a series defines a unique FPGA architecture. The most recent series of Xilinx are Series7, UltraScale, and UltraScale+. Each series can be separated into a list of families. These families all use the architecture of the series but are optimized for cost, power, performance, size, or another criterion. Families can be further broken down into one or more parts, which are the actual FPGA devices. In the following, we introduce the development tools of Xilinx and the architecture of the Xilinx FPGAs.

Xilinx Development Tools

Xilinx provides CAD tools with similar design flows that are described in Section 2.1.2 to develop applications for their FPGAs. In recent years, Xilinx released a new tool to design applications for their FPGAs: Vivado [26]. Vivado supersedes Xilinx ISE (the previous tool) and is the only tool suite that supports the latest Xilinx series, such as Series7, UltraScale, and UltraScale+. The most significant change with Vivado is the introduction of a *Tool Command Language* (TCL) interface [33]. Using the TCL commands, users of Vivado can write TCL code to script design flows, set constraints on a design, and perform low-level design modifications.

Xilinx Architecture

We can break down the Xilinx FPGA devices into a hierarchy of internal components. In Figure 2.4, the top-down hierarchy of Xilinx FPGAs is illustrated. On the top level, we find the individual FPGA device, which is shown in Figure 2.4a. The figure displays an FPGA model named *XC7Z020-CLG484*, which is a device from the Zynq family. The Zynq family belongs to series Series7.

An individual device can be broken down into *tiles*, which in turn can be broken down into *sites*. A single tile and site are shown in respectively Figure 2.4b and Figure 2.4c. In the following, we describe the tiles and sites in more detail individually.

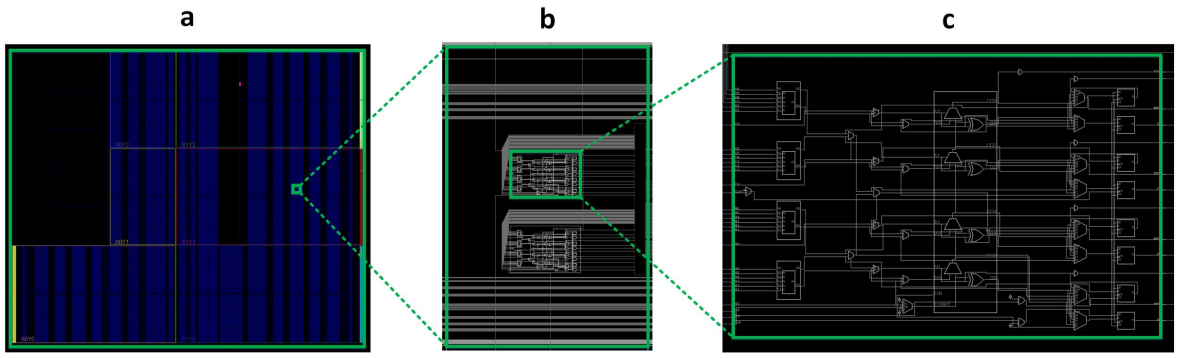

Figure 2.4: The Xilinx device hierarchy. (**a**) At the highest level of the hierarchy, we find the FPGA part, which is an individual device. (**b**) The device can be broken down into tiles, (**c**) which in turn can be broken down into sites.

A Xilinx FPGA is organized into a two-dimensional array of tiles. Each tile is a rectangular component that performs a specific function, such as implementing digital logic or providing routing interconnections. The tiles are located in a two-dimensional grid on the FPGA fabric, and they are wired together by the general routing fabric. All copies of a tile are identical or nearly identical (they might have minor routing differences).

In Figure 2.5, a part of the FPGA fabric from the *XC7Z020-CLG484* device is shown. In this figure, multiple types of tiles are illustrated. We shortly introduce these different types of tiles. The DSP tile provides the functionality to implement complex arithmetic functions efficiently [28]. The *interface* tiles are used for wiring signals between other tiles. These connections are not programmable. In contrast to the *interface* tiles, the *Interconnection* (INT) tiles provide programmable interconnections. The INT tiles allow a signal to be routed to various locations. The CLB tiles are used to implement logic functions [29], where the BRAM tile is used to store large amounts of data [27].

The size of the tile types vary. For example, the DSP and BRAM tiles take up five slots, where all the other tiles fit within a single slot, as illustrated in Figure 2.5.

All these different types of tiles are arranged in columns onto the FPGA fabric, which spans the full height of the FPGA. For the following, we separate the types of tiles in two categories: the logic tiles (CLB, DSP, and BRAM) and the INT tiles. Now, if we look in the horizontal direction, the resources on the FPGA alternate between two logic tiles and two INT tiles. In the case of the hard blocks (DSP and BRAM), there locate *interface* tiles between them to connect these tiles properly.

Each logic tile is connected to one or multiple adjacent INT tiles, and can only be connected with the rest of the FPGA resources via these INT tiles. The CLB tiles are connected to a single INT tile, where the hard blocks are linked to five INT tiles. For example, in Figure 2.5, the DSP tile in the first column can only be connected to the

rest of the system via the five INT tiles in the third column. Note that the *interface* tiles in the second column are used to connect the DSP tile and INT tiles. As an additional example, we take the CLB tile in the second row and the fifth column. This tile can only be connected with the rest of the system via the INT tile on the second row and the fourth column.

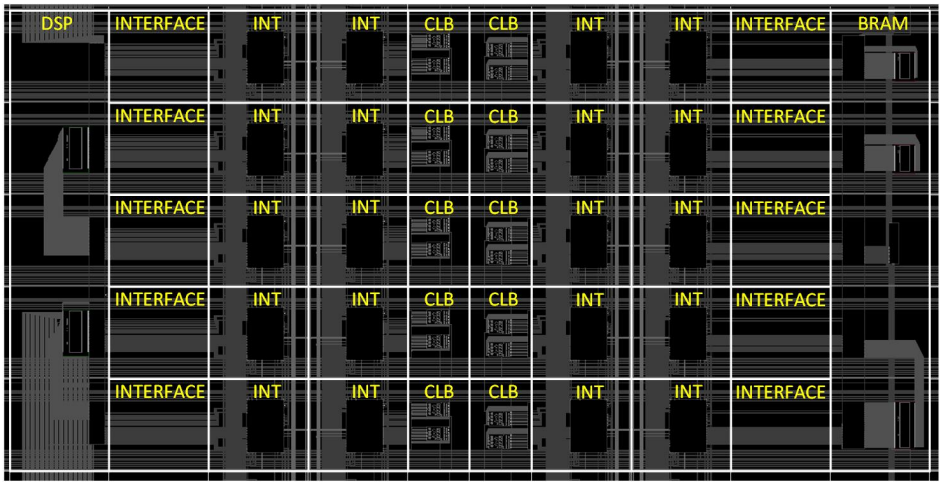

Figure 2.5: Xilinx FPGAs are organized as a two-dimensional array of tiles. The Xilinx devices contain different types of tiles and are arranged in columns onto the FPGA fabric that spans the full height of an FPGA device.

Until this point, we have seen how the tiles are located onto the FPGA fabric. In the following, we will have a more in-depth look into the INT tiles and the routing fabric onto the FPGA fabric.

FPGA components are connected using wires, where wires are connected by *Programmable Interconnect Point*s (PIPs) to make the FPGA reconfigurable. Individual PIPs can be enabled or disabled as the design is being routed, and a sequence of enabled PIPs uniquely identifies the used wires of a physical route. PIPs are most commonly found in INT tiles, and enable a single wire to be routed to several locations on the chip. An INT tile is illustrated in Figure 2.6. The source wire (green) can be connected to one or multiple sink wires (red).

The entry point of a particular wire onto the INT tile is called a *port*. In Xilinx, the INT tiles contain two types of ports: begin and end. These two types of ports are, respectively, the sink and driver nodes. The PIP connections always direct from the end ports towards the begin ports, as illustrated in Figure 2.6. Therefore, the wires can only be used in one direction: unidirectional.

The INT tiles contain wires that are either connected to its corresponding logic tile or other INT tiles. The INT tiles are directly connected to other INT tiles in all cardinal and intercardinal directions. The connections in all the cardinal directions are illustrated in Figure 2.7.

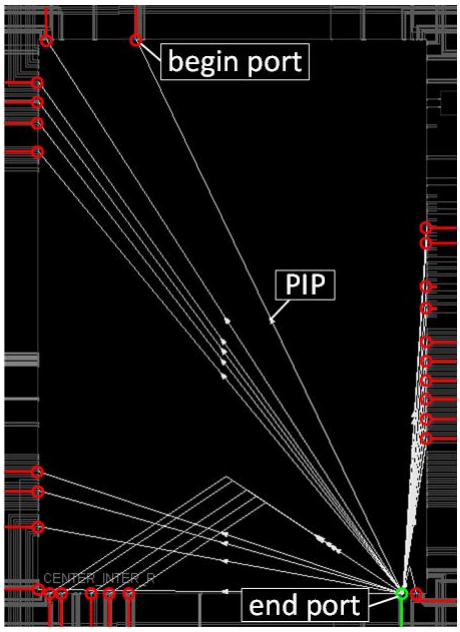

Figure 2.6: An INT tile. The green wire represents one of the source wires on the INT tile, where the red wires represent all possible sink wires that can be connected to the source wire. The gray lines inside the INT tile are the possible PIP connections.

As we have seen, the ports on a particular INT tile that are connected to other INT tiles are either begin or end ports, and the wires connected to these ports have a specific direction. The last property of a port is the length. This property defines the length of the wire connected to the port. The range is expressed in the number of INT tiles that the specific wire spans. For example, in Figure 2.7, the length of the wires are two INT tiles.

Furthermore, the INT tiles include multiple wire lengths towards the same cardinal direction. As we have seen in Figure 2.7, the INT tiles include wires in all the cardinal directions with a distance of two INT tiles. However, in the Zynq architecture, the INT tiles also contain wires that bridge a distance of four INT tiles in the eastern and western directions, for example. As another example, in the northern and southern direction, multiple wires span a distance of six INT tiles. The ports that belong to the same INT tile and have the same direction and length are bundled in groups of four ports.

The names of the ports on the INT tiles are used on a regular basis in this thesis. Therefore, we shortly introduce how Xilinx names its ports. As described before, the ports have three properties: port kind, direction, and length. The port kind is indicated by $P = \{BEG, END\}$, where *BEG* and *END* refer respectively to the begin and end ports. The notations to specify the direction for a particular port is given by $D = \{EE, WW, NN, SS\}$. *EE*, *WW*, *NN*, and *SS* stands for respectively

east, west, north, and south. Finally, the length is denoted as $L \subseteq \mathbb{N}^*$. As mentioned before, in the Zynq family, wires that are connected to the same INT tile with the same properties appear in groups of four. In the port name, the index of these ports with the same features is specified as $I = \{0, 1, 2, 3\}$. Now, the complete name of the ports is constructed in the way as in Equation (2.1). The symbols d, l, p, and i are elements from respectively the sets D, L, P, and I. The quotes around the elements give us the name of the element in string format, where the plus-sign behaves as a concatenation operator. For example, in Figure 2.7, the port names of the begin ports in the eastern direction are *EE2BEG0*, *EE2BEG1*, *EE2BEG2*, and *EE2BEG3*, where the port names of the end ports in the eastern direction are *EE2END0*, *EE2END1*, *EE2END2*, and *EE2END3*.

$$port\_name = "d" + "l" + "p" + "i" \tag{2.1}$$

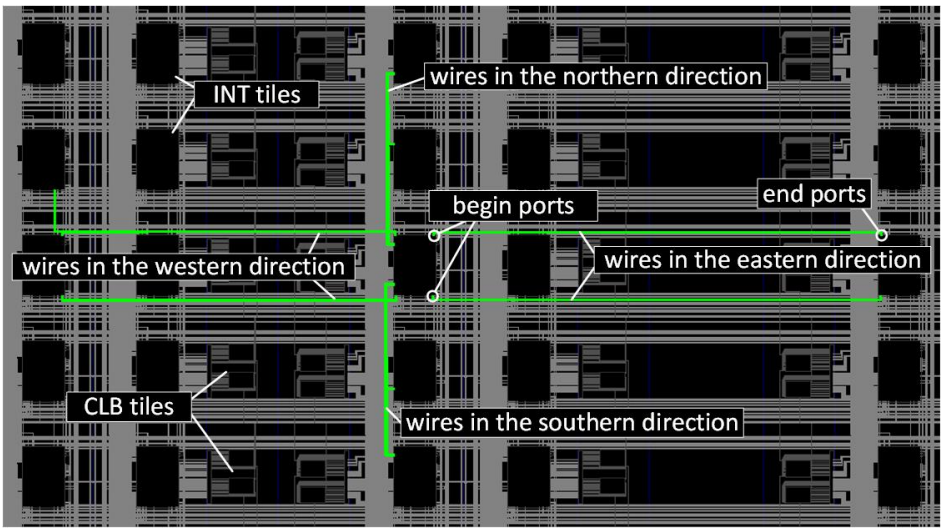

Figure 2.7: The wires connect INT tiles in all cardinal directions.

Now, we continue with the device hierarchy. As mentioned before, tiles can be broken down into sites. Tiles generally consist of one or multiple *sites*, which organize the hardware components of the tile into related groups. Specifically, sites are the part of the tile that performs the functionality of the tile. The remainder of the tile is used for wiring signals to and from its corresponding sites. The input and output pins of a site are called *site pins*. In the Zynq family, CLB tiles contain two sites. Figure 2.8 zooms in onto one of the two sites on the CLB. The name of this site is *SLICE*. *Basic Element*s (BELs) are hardware components belonging to a site, such as LUTs and FFs. In the Zynq family, each site of a CLB contains four BLEs. The LUTs provide six input pins, therefore supporting any six inputs boolean expression.

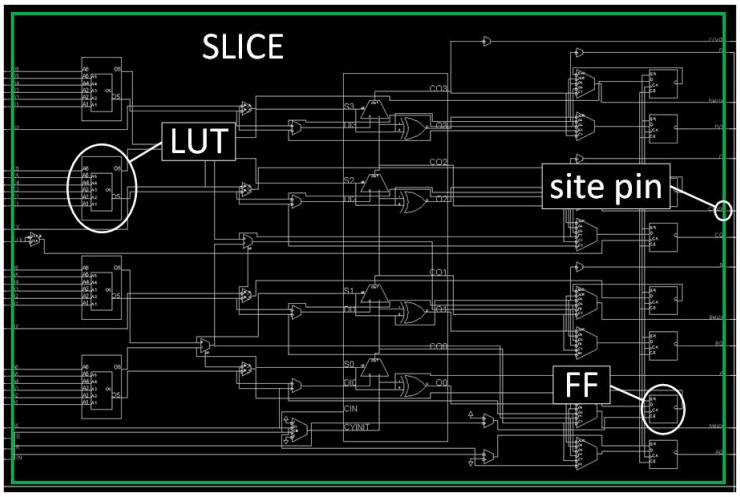

Figure 2.8: A tile usually consists of one or multiple sites. A CLB tile comprises of two sites. The sites are called *SLICE*.

2.2 Dynamic Partial Reconfiguration

A popular research topic on FPGAs is *Dynamic Partial Reconfiguration* (DPR). In *Dynamic Reconfiguration* (DR), the complete FPGA configuration is exchanged during run-time, wherein DPR exchanges only a part of the configuration memory. FPGA architectures allow us to change only a part of the configuration memory, while not altering the other parts. As mentioned in Section 2.1.2, a bitstream has to be loaded into the FPGA to change the implemented circuit. For Xilinx devices, external interfaces such as SelectMap or JTAG are used to load a bitstream [35]. Xilinx introduced an internal configuration interface, called *Internal Configuration Access Port* (ICAP) [30]. This internal interface makes it possible to load bitstreams from within the FPGA without additional off-chip control. A soft-processor or a custom state machine, also named as *Partial Reconfiguration Controller* (PRC) in Xilinx, could fetch configuration information from external memory and write the configuration memory through the ICAP [34]. Thereby allowing a circuit implemented on the FPGA to modify itself autonomously.

2.2.1 Benefits of DPR

In the early days of FPGAs, the available logic resources were limited, and using run-time reconfiguration had been suggested to raise resource utilization or to squeeze larger circuits into available logic. With the progress in silicon process technology, logic capacity increased steadily while getting cheaper (and often more power efficient per logic cell) at the same time. The explosion in capacity removed

the pressure on the FPGA vendors to add better support for run-time reconfiguration in their tools and devices. However, by heading towards devices with million LUT FPGAs, things are changing dramatically at the moment.

For the present high capacity FPGAs, the configuration time required to write tens of megabytes of initial configuration data is too long for many applications, and DPR can be used to speed up the process. The reconfiguration time is proportional to the size of the bitstream, which in turn is proportional to the area of the chip being reconfigured.

A further consequence of having sizeable high-density FPGAs is their higher risk of failure due to *Single Event Upsets* (SEU). SEUs are caused by ionizing radiation strikes that discharge the charge in storage elements, such as configuration memory cells, user memory, and registers. SEUs can be detected and compensated with the help of DPR (e.g., using configuration scrubbing).

Another factor arising for current high capacity FPGAs is a substantial relative increase in *static power consumption*. The static power consumption is related directly to the device capacity. With the help of DPRs, a system might be implemented on a smaller and consequently less power-consuming device. An example of such a system is illustrated in Figure 2.9. The system provides a *Software-Defined Radio* (SDR), different cryptographic modules, and protocol processing accelerators for various protocols. Assume that the SDR part will be adjusted according to the available bandwidth and that the cryptographic and protocol processing accelerators are changed on-demand. We can then save substantial FPGA resources by not providing all variants in parallel, but by only loading the currently required modules to the device. In [7] and [31], more applications are discussed that can save a substantial amount of resources (and thus reducing static power consumption) by using DPR.

The system in Figure 2.9 requires that the accelerator modules are either needed exclusively or that the system can time-multiplex the modules by sufficient fast reconfiguration. However, for low power operations, it should be mentioned that reconfiguring the FPGA requires some power. The additional energy required for reconfiguration includes the power to fetch a module from the module repository and the power required by the FPGA for the configuration process. Furthermore, we should note that the reconfigurable part will consume static power without providing useful work during the whole configuration process. If we assume that the system changes its operation modes on human interaction, the update rate will be sufficiently low such that it easily amortizes the configuration power.

DPR is also useful in scenarios where one part of the system is required to remain functional. Consider an FPGA system interfaced with a host computer via PCI Express. Full reconfiguration of the FPGA breaks the communication link, which may even require a host reboot to re-establish. DPR allows the link to be maintained

by keeping the interface circuitry active while the accelerator portion undergoes reconfiguration.

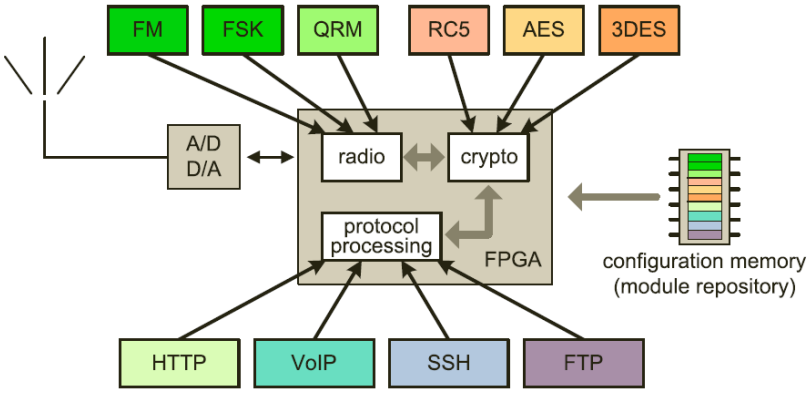

Figure 2.9: Area saving by reconfiguring only the currently required accelerator modules to the FPGA. Configurations are fetched from the module repository at run-time. This figure is taken from [2].

2.2.2 DPR Terminology

In DPR, the area of the FPGA is distinguished into two parts. The region of the FPGA that is reconfigurable during run-time is called the *partial area* (see Figure 2.10). A system might provide multiple partial areas. *Modules* can be loaded into the partial area in a time-multiplexed manner. Every *partial bitstream* represents a single module. The region of the FPGA fabric that remains the same during run-time is called the *static area*. The PRC and the internal configuration interface of the FPGA are often located in the static area. The PRC uses the internal configuration interface (e.g., ICAP in Xilinx devices) to load modules onto the partial area during run-time.

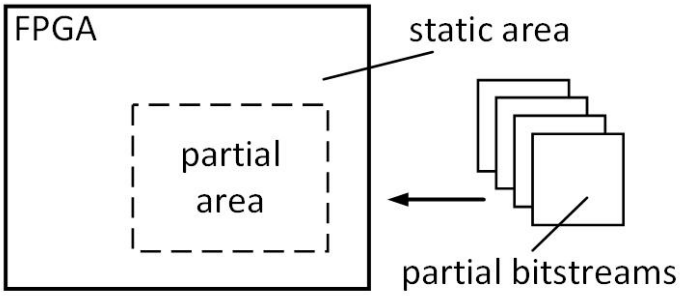

Figure 2.10: In DPR, the FPGA fabric is separated into two parts. The static area remains unchanged during run-time, while the partial area can host modules in a time-multiplexed manner.

The reconfiguration of the partial areas can be categorized into multiple styles [2]. In *island-style*, only one module can be hosted on the partial area at the same time. This style is illustrated in Figure 2.11a. For the following, suppose that a system provides multiple islands (partial areas). If a set of modules can only be configured on one specific island, we call this *single island-style*. In the case that *module relocation* is feasible among different islands, we call this *multi island-style*. Module relocation means that the same module can be loaded at various locations on the FPGA fabric. Module relocation makes it also possible to instantiate a single module multiple times on the FPGA fabric.

The size of a partial area should be at least the size of the most extensive module. As a consequence, there is usually a waste of logic resources that arises if modules with different resource requirements share the same island exclusively, which is called *internal fragmentation*. The reason is that a large module cannot be replaced by multiple smaller ones (to be hosted simultaneously). Therefore, the utilization of the partial area becomes inefficient. In Figure 2.11, the white surfaces in the partial areas indicate the unused reconfigurable area, and thus the internal fragmentation.

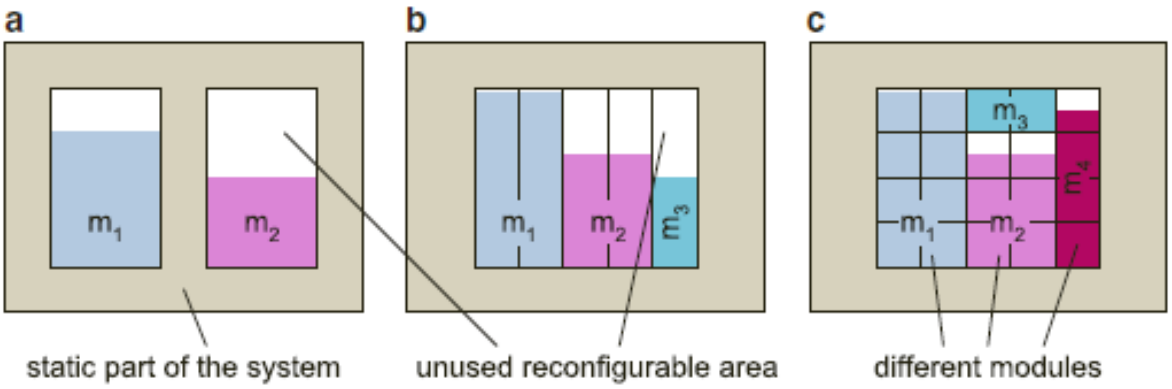

Figure 2.11: (**a**) In island-style, the partial area can only host one module exclusively at the same time. (**b**) In slot-style, the partial area is arranged in one-dimensional slots. A various number of modules can occupy one or multiple slots, according to their resource requirements. (**c**) In grid-style, the partial area is partitioned into two-dimensional slots. Similar to slot-style, one or multiple modules can take up the number of slots according to their resource requirements. This figure is taken from [2].

A more advanced reconfiguration style is *slot-style*. In slot-style, we arrange the partial area into one-dimensional slots to improve the internal fragmentation. This style of reconfiguration is illustrated in Figure 2.11b. Multiple modules can be hosted at the same time in the partial area, and the modules can occupy the number of slots according to their resource requirements. Arranging the partial area in slots

is considerably more complicated since the system has to provide communication to and from reconfigurable modules and to determine the placement of the module.

Furthermore, it is important to note that the FPGA resources are heterogeneous. As a consequence, depending on the present module layout, a partial area arranged in slots might not provide all the free tiles as one continuous area. If this results in slots that cannot be used, this overhead is called *external fragmentation*. These slots are available for allocation of modules, but might be too small or have an unsuitable footprint to be of any use.

The internal fragmentation of a partial area that is tiled into one-dimensional slots can still be significant, and especially the hard blocks can be affected by this. The reason is that these blocks waste a considerable amount of logic if they remain unused. As mentioned before, the resources are arranged in columns on the FPGA fabric. If a module needs only a few of these resources, it is beneficial if another module can use the remaining resources. This is possible in *grid-style* reconfiguration. In *grid-style* reconfiguration, the slots are arranged in a two-dimensional fashion, as illustrated in Figure 2.11c. The implementation and management of such a system are even more complex than the slot-style reconfiguration approach.

Previously, we introduced module relocation. Module relocation is especially useful in slot-style and grid-style reconfiguration. As mentioned before, in these styles, the reconfiguration region is divided into multiple slots in either one or two-dimensional. A various number of modules can be loaded simultaneously, and the modules can take a variable number of slots to their own needs. Figure 2.12a shows an example of a slot-style based reconfiguration without module relocation. The plot illustrates when the modules are used during time and which slots are used to load these modules. In the case module relocation is not supported, all slots S_n are occupied by a single module during run-time.

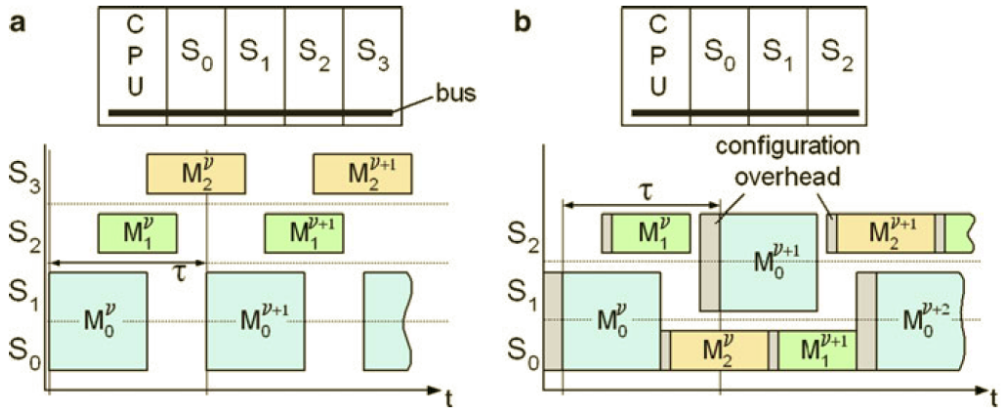

Figure 2.12: Module relocation helps to fit modules into a reconfigurable region over time better. This figure is taken from [2].

As we can see in Figure 2.12b, we can save one slot by using module relocation. This requires that we have spare time to reconfigure the modules such that they are loaded when they are required. Altogether, module relocation, in combination with slot-style or grid-style reconfiguration, allows us to build very flexible hardware systems.

2.2.3 Commercial DPR Tools

The design flow of DPR systems is considerably more complicated compared to the general design flow of FPGAs that is described in Section 2.1.2. The two leading FPGA vendors, Xilinx and Intel (before Altera), provide CAD tools to implement DPR systems. The tools offered by the two vendors have very similar design flows and require low-level FPGA architecture knowledge to develop a reconfigurable system efficiently.

Xilinx supports DPR through its PlanAhead [25] and Vivado Design Suite [31] tools. In the PlanAhead tool flow, the DPR design is composed of the static design and several modules. The hardware layout is similar to that we discussed in Section 2.2.2. In the following, we briefly describe the design flow to develop DPR systems using PlanAhead.

The first step in the design flow is to determine the number of reconfigurable regions and the modules allocated to these regions, which is called *partitioning*. After partitioning the DPR design, the designer has to manually floorplan the locations and bounding boxes of the reconfigurable regions on the FPGA fabric. These floorplanning details are stored in a constraint file for incorporation in the implementation stage. After floorplanning, the designer has to determine the *configurations*. A configuration is a static design with one module in each reconfigurable region. In the implementation stage, the static design is implemented with the first configuration as a placeholder. The final placement and routing of the static region are preserved for all other configurations. The partial modules are then implemented as an increment to the static system. The static design can use the routing resources (but no logic elements) inside the reconfigurable regions, but not vice versa. After the implementation, the tool generates full bitstreams for each configuration. Also, all the partial bitstreams for each reconfigurable region are generated. At run-time, the FPGA is configured with one of the full bitstreams. Later on, any single reconfigurable region can be reconfigured by using the partial bitstreams.

Xilinx supports DPR for newer FPGAs through the Vivado Design Suite. This tool flow is similar to PlanAhead. Intel provides almost identical tool flows (with different terminology) compared to Xilinx for their FPGAs through the Quartus-II [36] and the new Quartus Prime [37] design software. Besides the design flows; also, the way

the vendor tools build the DPR systems is equivalent. In the continuation of this section, we describe how DPR systems are constructed on FPGAs by the vendor tools and discuss the drawbacks of these methods.

An essential part of DPR designs is the communication between the static design and the modules. The current vendor tools use *proxy logic* to establish the connection to and from the modules. Proxy logic are anchor LUTs, which are placed inside the partial area for each interface signal, as shown in Figure 2.13. The interface signals are routed to the anchor LUTs during the implementation of the static system. The partial modules are implemented as an increment to the static system without modifying any of the already implemented static routings.

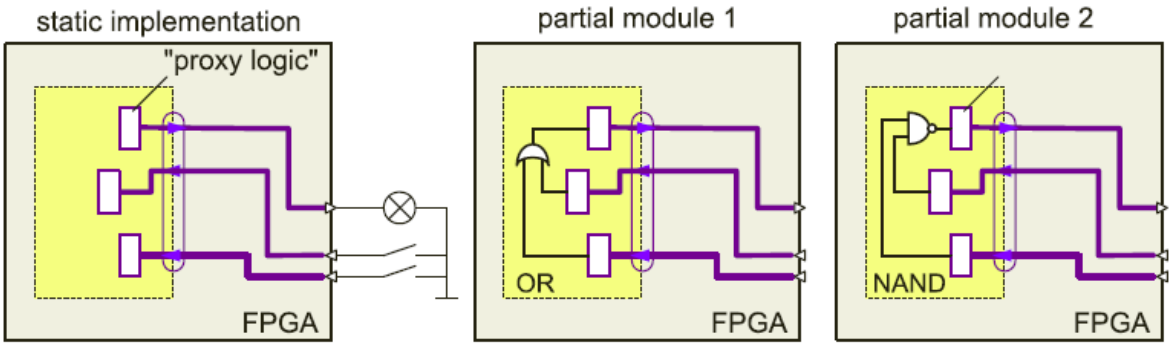

Figure 2.13: Partial module integration using *proxy logic*. After initial static implementation, the routing is used and preserved for incrementally building all partial modules. This figure is taken from [2].

The routing to the anchor LUTs is not strictly constrained. Therefore, the routing is usually different in each reconfigurable area. As a consequence, module relocation among different reconfigurable areas is not supported, even if the islands provide an identical footprint. The problem is illustrated in Figure 2.14a. The modules m_1 and m_2 only take the routing inside their own reconfigurable region into account. As a consequence, if the modules m_1 and m_2 are swapped, the static routing (routing violations) will be cut. We can solve this problem by merging the routing violations of both reconfigurable regions into one region, as shown in Figure 2.14b. This region is then used for implementing the reconfigurable modules. The obtained modules are illustrated in Figure 2.14c. If we configure the merged module m_1 in the right-hand reconfigurable region of Figure 2.14a, the static routing remains the same, and thus module relocation is feasible. However, this is not applicable for systems with plenty of partial areas (island-style) or slots (slot-style and grid-style), as most likely routing congestion will occur when implementing the modules. The reason is that most of the wires in the partial area are then occupied by the static routes, and therefore, the modules cannot use them anymore. Also, merging the static routing may fail when

multiple wires cross the same path, as shown in Figure 2.14d. The reason is that a wire track can only implement one static routing path through the reconfigurable area.

As we have seen, the vendor tools do not support module relocation due to the routing violations. Another limitation of the proxy logic approach is that the routing to the anchor LUTs will most likely change each time the static system is changed. Consequently, all permutations of a module instance and placement position have to be reimplemented on each change of the static system.

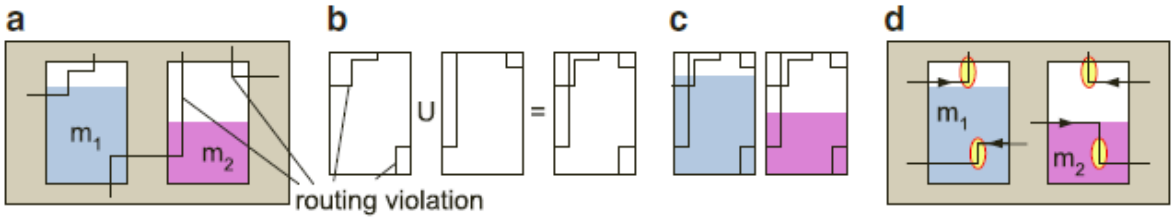

Figure 2.14: (**a**) The modules m_1 and m_2 cannot be swapped as this will cut the static routes through the reconfigurable region. (**a**) We can solve this by merging all the routing violations into one region and (**c**) use this area to implement the modules. This way, module relocation becomes possible. (**d**) However, merging the routing violations may fail on wire conflicts. This figure is taken from [2].

Finally, a reconfigurable area can only host one module exclusively (island-style reconfiguration) and is not supported to share a reconfigurable area by multiple modules in a flexible manner at the same time. As a summary, the current limitations/drawbacks of the vendor tools are the following.

- Implementation of DPR applications causes logic area overhead since each signal wire costs one LUT using the proxy logic approach.
- Module relocation is not supported. The current vendor tools require to generate a partial bitstream for every reconfigurable module that is allocated in each reconfigurable region. For example, if the system contains m modules that should be relocatable in n different reconfigurable regions, it is necessary to generate $m * n$ partial bitstreams. As a consequence, the implementation time and on-system memory requirements will increase. Module relocation would allow us to produce a single bitstream of a module, which can be configured in any compatible reconfigurable region.
- Any modification in the static region requires complete reimplementation of the static region and all modules.

- A reconfigurable region can only host one module exclusively (single island-style).

As we have seen, the current DPR tools that are provided by the vendor tools have considerable limits. In the next chapter, we will look at what the research community has done to overcome these limitations.

Chapter 3

Related Work

In the previous chapter, we discussed the limitations of the current vendor DPR tools. In this chapter, we describe some relevant open-source DPR tools developed by the research community. The main objectives of these tools are to support module relocation, independent design flow of the static system and the modules, and more flexible architectures (e.g., slot-style and grid-style reconfiguration). Most of these tools use vendor tools for low-level device-dependent operations such as placement, routing, and bitstream generation. This chapter is organized as follows. In Section 3.1, we discuss relevant open-source DPR tools and their design flow. In the latter section, Section 3.2, we compare the DPR tools.

3.1 Academic DPR Tools

OpenPR OpenPR is an open-source development environment to develop DPR applications [6]. The tool provides similar functionality as the Xilinx design flow. The first step in the tool flow is creating an XML project file, where the designer specifies the design parameters. In the following, the designer has to manually floorplan the reconfigurable regions with the Xilinx PlanAhead tool. The OpenPR design flow generates the static design by using placement constraints and a blocker to prevent routing through the reconfigurable regions. The placement constraints prohibit the placer from placing any logic inside the reconfigurable region, where the blocker is used to occupy all the wires inside the reconfigurable region. The latter ensures that the router cannot route through the reconfigurable region. Once the static design is routed, the blocker is removed, and the static bitstream is generated. Finally, the partial bitstreams are generated by the use of Xilinx bitstream generation tools.

The main advantage of OpenPR is its availability as an open-source platform. Therefore, researchers can extend the platform to explore other modes of DPR. Another difference compared to the Xilinx design flow is that the tool blocks the

static region from routing through the partial area. As a result, the static and partial region can be implemented separately, and with changes in the static region, it is not necessary to reimplement all the modules. Another advantage of preventing the static design from routing through the partial region is that module relocation is supported.

GoAhead GoAhead is another academic DPR tool to overcome some of the limitations of the vendor tools [1]. The tool can implement DPR systems for all recent Xilinx FPGAs. GoAhead assists during floorplanning and automates constraint generation for the place and route implementation phase. GoAhead provides an intuitive *Graphical User Interface* (GUI) as well as a scripting interface. All the GUI actions are recorded by the corresponding script commands. As a result, there is no need to learn the GoAhead scripting language. The latter removes error sources and ensures reproducible results. GoAhead supports module relocation and integration of partial modules without any logic overhead. Also, more advanced reconfiguration styles are supported (e.g., slot-style and grid-style).

In the GoAhead design flow, the static design and modules are implemented through independent design flows. The designer first has to determine the static part of the system and the modules that will be reconfigurable. GoAhead offers a GUI based tool to floorplan the design, which allows a designer to select one or more areas on the FPGA fabric that will be used as reconfigurable regions. Based on the floorplanning, the GoAhead tool generates constraints that prohibit to allocate any static logic resources inside the partial areas. Also, the GoAhead tool creates a *blocker macro*, which occupies all the wires within the partial region. This blocker macro is used while routing the static system, so it prevents the static system to route through the partial area. The implementation of the modules is similar, where the blocker macros prevent routing from reconfigurable regions into the static area. This way, the static and partial regions are entirely separated. Finally, vendor tools are used to generate partial and full bitstreams from the placed and routed designs.

The design flows of OpenPR and GoAhead both use a blocker to prevent routing in the partial area. As a result, module relocation is supported, and the design flow of the modules and the static design is separated. However, there are also significant differences between these tools. OpenPR uses *bus macros* to integrate reconfigurable modules into a system. In bus macros, one logic primitive is placed in the static system, and another one in the reconfigurable area and wires between them are used to carry out the routing between the static and partial part. By placing the macro on the partial area border, an interface signal to wire binding is achieved due to the internal bus macro routing that will be maintained through all implementation steps. Interface signals work similarly to a physical plug on a PCB, and the

binding of interface signals to wires has to be identical for the static system and all the partial areas.

However, integrating partial modules using bus macros has several drawbacks, such as logic overhead and additional latency. GoAhead provides an alternative that circumvents these problems by binding the interface signals directly to the wires crossing the border from static to partial (and vice versa) without the help of logic resources. It is important to note that we cannot directly define the *binding*. In this case, binding means that we cannot define a specific signal x that has to be routed using certain wire y. GoAhead generates the blockers such that it leaves only one available routing path between the static area and the partial area for each interface signal. Therefore, each signal is forced to route via that particular path. As a result, the signals are bind to the wires in this path.

Dreams As we have seen, OpenPR and GoAhead both use a blocker to prevent routing in the reconfigurable region. In [4], an alternative tool called Dreams is presented to support module relocation, and independent design flows of reconfigurable modules and the rest of the system.

The tool flow starts with a conventional and independent placed and routed netlist generation for each module. The generated netlists are transformed such that they meet the specific requirements of the DPR system. Dreams uses a custom router that constraints the routing, such as preventing static routing within the partial area. The custom router is also used to guarantee that the interface signals of the modules and the static design are bind to the same wires. As a result, there is no logic or delay overhead.

The custom router is developed with the tool RapidSmith [14]. This tool provides functions to change *Xilinx Design Language* (XDL) files. XDL offers a powerful interface that allows access to virtually all features of Xilinx devices [13]. On one side, this includes the generation of complete device descriptions containing information about the FPGA primitives and the routing fabric. On the other side, XDL can be used to constrain systems or to implement modules or macros for Xilinx FPGAs directly.

Dreams supports the communication between the static area and partial area without any logic overhead. Also, module relocation is supported. Furthermore, the design flows of the modules and the static system are independent. Finally, Dreams supports the design of highly flexible DPR architectures (e.g., slot-style and grid-style).

CoPR In the previous tools, the main focus is on developing more flexible DPR systems. In [3], the tool named CoPR is presented, where the primary purpose

is to raise the abstraction level for developing DPR applications. Also, in this tool, run-time management is supported.

The tool targets the Xilinx Zynq device. Zynq is a hybrid reconfigurable device, which includes a processor, standard communication architecture, and integrated reconfigurable fabric. The processor is used to implement the PRC, and the reconfigurable regions are implemented in the FPGA structure.

The designer has to provide *configuration* and *adaption* specifications to the tool. The configuration specification details the different valid system configurations and the corresponding library modules present in each configuration. The adaption specification contains software code (written by the designer) for changing configurations at runtime. CoPR offers an *Application Programming Interface* (API) to help the designer write the software without requiring knowledge of implementation details. The next steps are all automated. CoPR uses the vendor synthesis tool to synthesize all modules for the target FPGA to determine resource requirements. In the following, the partitioning step, the number of reconfigurable regions, and allocated modules to them are resolved. Then, floorplanning is performed to determine the locations of all reconfigurable areas. Finally, the Xilinx command-line tools are used to implement the design and to generate the bitstreams.

3.2 Comparison of DPR Tools

In this section, we compare the Xilinx and academic DPR tools. We omit the DPR tools from Intel since the academic DPR tools only target the FPGA devices from Xilinx. In Table 3.1, the most important features are listed.

The first feature of the tools listed in Table 3.1 is the device support. We see that GoAhead supports a large number of different device families. Remarkable is that GoAhead supports even more devices than the Xilinx tools itself. CoPR targets only one specific device family: Zynq.

The next feature covers the communication overhead of each DPR tool. Communication in this context means how the interface signals of the modules bridge between the static and partial regions. The communication method can cause logic overhead. For example, proxy logic requires one LUT for each interface signal, as we have seen in Section 2.2.3. In table 3.1, the number of LUTs required per interface signal is listed. Especially applications with a relatively large amount of interface signals compared to its resource usage, the logic area overhead is significant. For these applications, GoAhead and Dreams are very promising, since these tools support DPR applications without logic area overhead [9].

Module relocation, in combination with slot-style or grid-style reconfiguration, allows us to build very flexible and fine-grained hardware systems. Table 3.1 lists

whether the tool supports module relocation and lists the different reconfiguration styles. GoAhead and Dreams both support all reconfiguration styles and module relocation [12]. The tools from Xilinx and CoPR are the least flexible and fine-grained since they only support single island-style.

In the static system, often, the PRC is located, which manages the reconfiguration of the partial areas during run-time. The tools that support run-time management are the Xilinx tools and CoPR, as illustrated in Table 3.1. Designers that use one of the other tools are required to implement the run-time management themselves or using a third-party tool.

Table 3.1: Comparison between the Xilinx and academic DPR design tools.

Feature	Xilinx tools	GoAhead	OpenPR	Dreams	CoPR
Supported devices	V4, V5, V6, V7, Zynq, UltraScale	V4, V5, V6, V7, S6, Zynq, UltraScale	V4, V5	V5, S6	Zynq
Communication overhead[1]	1	0	2	0	?[3]
Module relocation	No	Yes	Yes	Yes	No
Reconfiguration styles					
Island-style	Yes	Yes	Yes	Yes	Yes
Slot-style	No	Yes	Yes	Yes	No
Grid-style	No	Yes	No	Yes	No
Run-time management	Yes	No	No	No	Yes
Resource budgeting	No	No	No	No	Yes
Partitioning	No	No	No	No	Yes
Floorplanning	No	Yes[2]	No	No	Yes
Independent design flow	No	Yes	Yes	Yes	No

[1] In terms of LUTs per interface signal.

[2] Automatic floorplanning is only supported for island-style.

[3] In [3], the communication method is not mentioned.

Automating steps increases the design productivity and makes implementing DPR systems accessible for designers without low-level knowledge of FPGA architectures. Table 3.1 lists whether the tools support automatic resource budgeting, partitioning, and floorplanning. Resource budgeting is the calculation of resources (e.g., LUT, DSP, and BRAM) that are used for each reconfigurable module. Partitioning determines the number of reconfigurable regions that are used in the design and its corresponding modules. Finally, floorplanning determines the location of the reconfigurable regions onto the FPGA fabric. An intelligent arrangement and allo-

cation of DPR regions can result in reduced area and hence allows designs to fit on smaller devices (see Figure 2.12). Also, in the tools that block the reconfigurable area while routing the static design, the partial region forms an obstacle for the static router. As a consequence, poor placement of the partial regions might result in a timing violation. In the current tools, resource budgeting, partitioning, and floorplanning must be performed manually, except the tool CoPR [15], [10]. GoAhead supports automatic floorplanning, but only for island-style reconfiguration [8]. Note that in this thesis, the main focus is on the reconfiguration style and not the abstraction level of developing DPR systems. Therefore, most of these issues that are just mentioned here will not be addressed. However, in Chapter 4, we provide several suggestions to do these steps efficiently manually.

The last feature in Table 3.1 indicates whether the tools support an independent design flow of the static part and the partial modules. A separate design flow allows us to make changes in the static design without requiring to reimplement all the modules. All academic tools support this feature, except CoPR.

Chapter 4

Concept for Grid-Style Partial Reconfigurable System

In this chapter, we describe our proposed DPR system that supports fine-grained DPR. In our contemplation, an efficient, flexible, and fine-grained DPR system should include the following.

- Minimize internal and external fragmentation.
- Multiple modules configured within a partial area simultaneously.
- Modules with different dimensions and shapes.
- Module relocation.
- Minimize the reconfiguration time.
- Minimize delay and area overhead.

This chapter is organized as follows. In Section 4.1, we discuss the fine-grained DPR systems of the current DPR tools and their limits. In the following, we introduce our proposed DPR system to overcome some of these limitations. This system is described in Section 4.2.

4.1 Limits of the Current DPR Tools

In Section 2.2.2, we discussed the concepts of internal and external fragmentation. By using the grid-style reconfiguration style, we can minimize these concepts. In Section 3.2, we have seen that only the tools GoAhead and Dreams support grid-style reconfiguration. Therefore, in the continuation of this section, we will only consider these two tools.

GoAhead

In GoAhead, slot-style and grid-style reconfigurations are supported through I/O bars [12]. I/O bars contain bundles of wires that are homogeneously routed among the slots of the system. For slot-style reconfiguration, GoAhead uses a single I/O bar, as illustrated in Figure 4.1. In this figure, the partial area is separated into four slots. The modules are developed such that they can be configured in one or multiple slots. The modules themselves must be a continuous area.

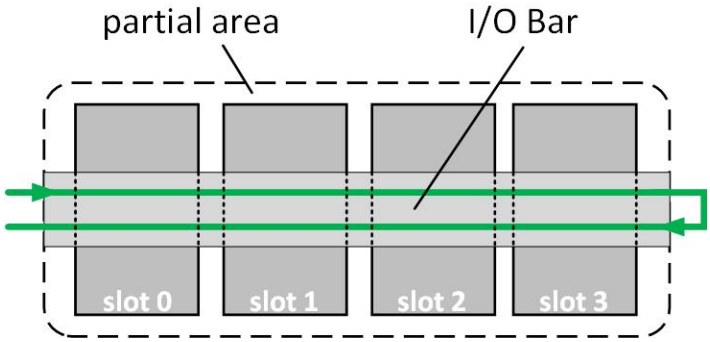

Figure 4.1: In GoAhead, I/O bars are used to support slot-style and grid-style reconfiguration. A single I/O bar is used in slot-style reconfiguration. Modules can be configured in free slots, and a single module can use one or multiple slots.

In the partial area, for each slot, the I/O bar can be used in three different ways. First, we can by-pass a slot, as illustrated in Figure 4.2a. In Figure 4.1, all the slots are by-passed. In the case a module is configured in one or multiple slots, the I/O bar is broken open and is connected to the module. This is shown in Figure 4.1b. The upstream is directly connected to the input signals of the module, where the output signals of the module are connected to the downstream. The downstream propagates towards the following slots. Finally, the I/O bar can be used to tap the signal, as illustrated in Figure 4.1c.

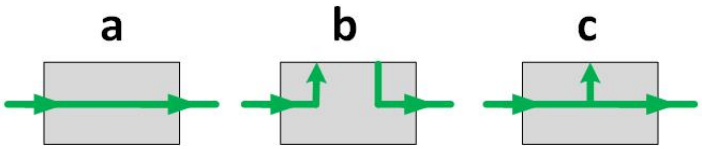

Figure 4.2: The I/O bars can be used in three different ways. (**a**) First, a slot might be by-passed. (**b**) Secondly, a module might be configured in one or multiple slots. (**c**) Finally, a module might tap the signal of the I/O bar.

For the flexibility of the placement, each I/O bar is designed with a forward and backward signal path. This removes communication-related placement restrictions and allows us to communicate between two modules regardless if the second module is located left or right beside the first module. For example, suppose that we place module A at the left side of module B. Then, we can obtain both the output $O = A(B(I))$ and $O = B(A(I))$, where I is the input signal.

The I/O bars contain a single input interface and a single output interface between the static system and the partial area. This can result in reconfiguration time overhead, as is demonstrated in Figure 4.3.

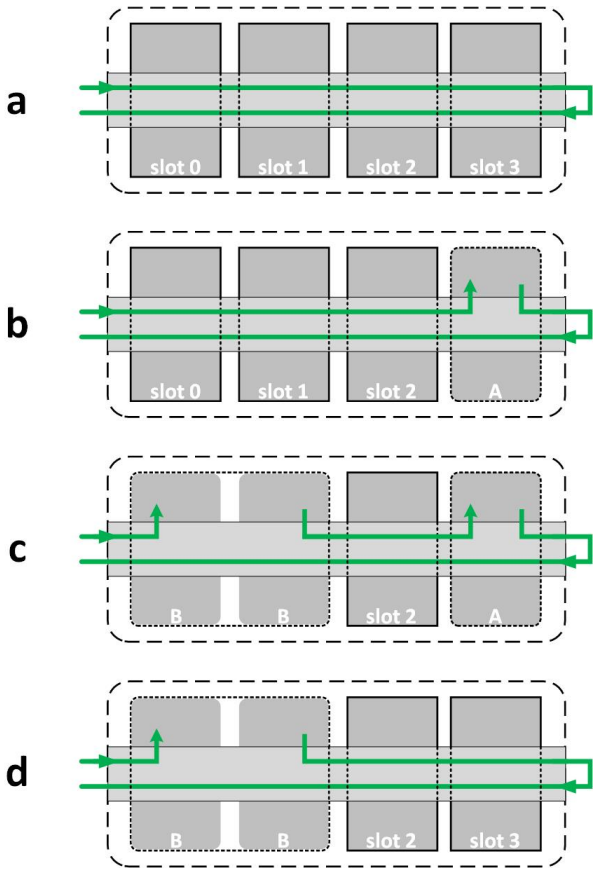

Figure 4.3: (**a**) In the initial system, all the slots of the I/O bar are by-passed. (**b**) Then, module A is configured in slot 3. (**c**) In the following, module B is configured in slots 0 and 1. (**d**) Finally, module *A* must be reconfigured such that it is by-passed to obtain the output of module *B*.

In this demonstration, all the slots are by-passed in the initial system, as illustrated in Figure 4.3a. Since all the slots are by-passed, the output is equivalent to the input. As mentioned before, modules might be configured in any slot and can occupy one or multiple slots. However, the modules are developed on a specific footprint. Therefore, the modules can only be configured in these slots if the footprint is

similar.

In Figure 4.3b, module A is configured into the partial area at slot 3. For the following, suppose that we would like to reconfigure the partial area with another configuration. The configuration contains a single module B with a different footprint compared to module A. Therefore, we have to configure module B on another location than module A. Suppose that module B requires two slots and is configured in slot 0 and slot 1. If we reconfigure just the module, we end up in the configuration that is illustrated Figure 4.3c. In this case, the output is $O = A(B(I))$, which is not the output that we desired. Therefore, we also have to reconfigure the modules from the old configuration such that the communication link is restored. As a consequence, we have to reconfigure significantly more area than just the slots we use for the modules belonging to a particular configuration. In this example, we have to reconfigure module A such that it is by-passed. In Figure 4.3d, the link of the previous configuration is restored, and therefore, we obtain the desired output.

Now that we have seen how the slot-style reconfiguration is implemented in GoAhead, we will discuss the grid-style implementation. In GoAhead, multiple I/O bars are used to support grid-style reconfiguration. The I/O bars are stacked on top of each other, and they are linked via multiplexers. The multiplexers are located in the static system and perform the vertical routing, while the I/O bars carry out the routing in the horizontal direction.

In Figure 4.4, a grid-style DPR system is illustrated. As usual, in grid-style applications, the partial area is partitioned in two-dimensional slots. The number of columns in the grid is determined by the number of slots the I/O bar is divided. Each I/O bar in the grid is partitioned in the same number of slots. For each row in the partial area, a single I/O bar is used. Thus, if we would like to partition the partial area in $R \times C$ slots, we have to use R I/O bars, and we have to divide these I/O bars into C slots.

The grid-style system in Figure 4.4 contains a partial area that is divided into 4×2 slots. In the partial area, three modules are configured: A, B, and C. The output of the partial area in this configuration is $O = C(B(A(I)))$. The multiplexers are controlled such that we obtain the correct flow of data. The flow of data is as follows. The multiplexer at the top selects the signal *input*. This signal propagates through the I/O bar at the top via modules A and B. Therefore, we obtain an intermediate output $O_i = B(A(I))$. The output O_i is selected by the central multiplexer and propagates through the I/O bar at the bottom via module C. Finally, the bottom multiplexer is used to select which I/O bar should be connected to the final output. In this case, the bottom multiplexer selects the output signal of the bottom I/O bar.

The modules that are configured in the partial area at the same time are often dependent on each other. For example, in Figure 4.4, module B depends on mod-

ule A, where module C depends on module B. Therefore, the modules should be placed in consecutive fixed order. For the particular configuration in Figure 4.4, the communication architecture is sufficient. However, the I/O bars, in combination with the multiplexers, also have some limitations in flexibility for placing the modules in the partial area. The reason is that I/O bars support only communication in the horizontal directions, and therefore it is not feasible to route vertically within the partial area. In the following, the problem is further described by using a concrete example.

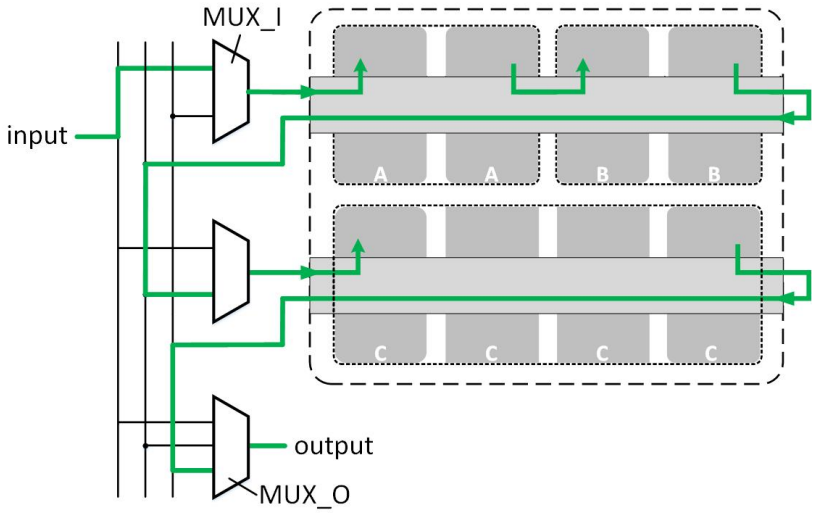

Figure 4.4: By using I/O bars for grid-style applications, the placement of modules in consecutive order is limited. The reason is that we can only communicate in the horizontal directions within the partial area.

In this example, we use the same grid and modules, as illustrated in Figure 4.4. We start with an initial grid, where all the slots are by-passed. Now, suppose that the desired output is $O = B(C(A(I)))$. Then, we have to place the modules as follows. We have to put module A before module C. However, module A and module C cannot be placed in the same I/O bar, since the I/O bar does not provide enough slots. Therefore, module A and module C must be located in separate I/O bars. Now, the multiplexers are configured such that the input signal first propagates through module A and then through module C. Finally, module B must be placed after module C. However, we cannot place module B in the same I/O bar as module C, because all the slots are occupied by module C. Also, we cannot place module B in the other I/O bar, because this will produce a wrong result. Thus, as we have seen, there is no way to configure the system such that we obtain the correct flow of data. Therefore, it is convenient to be able to route in both horizontal and vertical directions. In this case, we have more flexibility in the placement of the modules.

The communication architecture of GoAhead has some more limitations. Namely,

the multiplexers cause logic overhead to the DPR system. As already mentioned, the number of I/O bars is equivalent to the number of rows R. The input interface of each I/O bar is connected to one multiplexer, and there is one additional multiplexer that selects the output of the partial area. Therefore, the total number of multiplexers required by the grid-style system is $R + 1$. In the following, we distinguish the multiplexers in two types. We call the multiplexers that are connected to the input interfaces of the I/O bars *MUX_I*, where the multiplexer that selects the output of the partial area is called *MUX_O*. The resource consumption of these multiplexers are described in the following.

The input signal of an I/O bar is determined by its corresponding multiplexer *MUX_I*. The *MUX_I* multiplexer selects one of its input signals, and these input signals include the signal *input* and the output signals of all the *I/O* bars, except the output signal of its corresponding I/O bar. Therefore, the *MUX_I* multiplexers contain a total of R input signals. The multiplexer *MUX_O* selects the final output from one of the I/O bars. Also, the signal *input* can be directly connected to the signal *output* without propagating through one of the I/O bars. Therefore, the *MUX_O* multiplexer comprises $R + 1$ input signals. The input signals on the multiplexers are vectors of bits with length L. In principle, each I/O bar has the same input and output interface. These interfaces include the interface signals of all the individual modules. Consequently, modules can be placed in any I/O bar.

In the synthesis tools, LUTs belonging to the CLBs are used to implement multiplexers. For the following, assume that the LUTs contain six input ports. Then, each LUT provides two select signals that select one of the four remaining input signals. However, if we have more than four input signals, we require more than two select signals. This results in a tree structure of LUTs. Therefore, the logic overhead $MUX_{overhead}$ in terms of LUTs by all the multiplexers is the following.

$$MUX\_I_{overhead} = L * \left\lceil \frac{R}{4} \right\rceil \tag{4.1}$$

$$MUX\_O_{overhead} = L * \left\lceil \frac{R+1}{4} \right\rceil \tag{4.2}$$

$$MUX_{overhead} = R * MUX\_I_{overhead} + MUX\_O_{overhead} \tag{4.3}$$

As we have seen, the multiplexers can cause significant logic overhead. Another disadvantage of the multiplexers is that we require a substantial amount of control signals. We need one control signal for each multiplexer in the DPR system. Therefore, the total number of control signals is $R + 1$.

Finally, another disadvantage of implementing a grid-style DPR system using I/O bars is the potential delay caused by the multiplexers (LUTs) and the length of the wiring. This might be solved by pipelining the modules.

Dreams

Dreams is another DPR tool that supports grid-style reconfiguration [4]. Note that they call this reconfiguration style *mesh-type architecture*. Dreams uses a different communication architecture compared to GoAhead. In contrast to GoAhead, there is no communication bus within the partial area. The partial area contains only interface wires at the borders. In Figure 4.5, this is illustrated. The modules are developed such that they can communicate with these interfaces, but also with other adjacent modules. Furthermore, the modules can occupy one or multiple slots. The only constraint is that the shape of the modules must be rectangular.

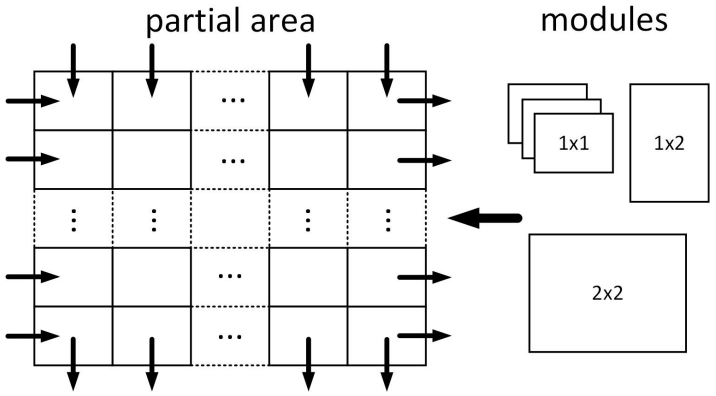

Figure 4.5: The partial area is partitioned in two-dimensional slots. Modules are developed such that they can communicate with the static area via the interfaces at the boundary and other adjacent modules.

The communication direction of the interfaces at the border is unidirectional. In Figure 4.5, the input signals of the modules can be connected to the interface signals at the northern and western borders, where the output signals of the modules can be connected to the interface signals on the other sides. As mentioned before, the modules are developed such that they can communicate with other adjacent modules. Adjacent modules have a direct connection, and they can communicate with each other in all cardinal directions. Since there is no internal communication bus, the modules must be placed such that they route the signal from one of the inputs towards one of the outputs to have a complete connection between input and output.

As a disadvantage, in Dreams, modules cannot communicate if they are non-adjacent to each other. In GoAhead, this is feasible as there is a communication bus within the partial area. For example, in Figure 4.3c, modules A and B can communicate, although they do not locate adjacent. However, we have seen that the communication bus in GoAhead must be restored if we configure modules at other locations in comparison to the previous configuration (see Figure 4.3). In fact,

we can do the same in the Dreams framework. If modules are nonadjacent, we can configure the slots in between modules such that they connect these modules. Later on in this section, we will introduce this construction as a by-pass module. Since Dreams also supports vertical communication within the partial area, we have significantly more flexibility in the placement of the modules.

Furthermore, the approach from Dreams prevents the use of all the *MUX_I* multiplexers to control the data flow. We might require a multiplexer that selects one of the outputs. In this case, we need only one control signal for the complete grid-style system. Finally, in this system, we do not require the long wiring for vertical communication and thus have a less potential delay. Therefore, we can obtain higher throughputs.

Now that we have seen that the architecture of Dreams has more potential compared to the architecture of GoAhead, we will discuss the tool flow of Dreams. As mentioned in Section 3.1, Dreams uses a custom router to ensure that routing will be compatible between any pair of adjacent modules or the static system. We cite the following from [4]: "*Typically, the routing process is divided into two phases: a global routing, which gathers nets together to balance all routing channels, and a detailed routed, which finally assigns specific wires to each net. In this case, since the number of signals to be routed is limited, just a detailed router has been implemented, which routes sequentially every desired net.*". The latter raises some questions. What will happen if the number of signals to be routed is significantly large? Since the signals are routed sequentially, routing congestion could happen most likely when having a significant number of interface wires.

Another huge disadvantage of Dreams is that the router is written by using RapidSmith. RapidSmith modifies XDL files. However, XDL files are not supported by Vivado anymore. Therefore, we cannot use Dreams for newer Xilinx devices. Note that the successor of RapidSmith, RapidSmith2, supports the newer FPGAs of Xilinx.

4.2 Proposed Grid-Style DPR System

In this section, we discuss our proposed DPR system to overcome the limitations that are described in the previous section. We adapt the architecture from Dreams. As we have seen in the last section, the architecture of Dreams has more flexibility in the placement of modules in comparison to GoAhead. Furthermore, the Dreams framework requires fewer control signals and has a less potential delay.

In Figure 4.6, our grid-style system is illustrated. As an addition to the Dreams framework, we use bi-directional communication interfaces at the borders. This increases the flexibility of the module placement even more. The reason is that the interface signals of the modules can enter and leave the partial area from each po-

sition at the border. Furthermore, we add support for different shapes of modules, such as L-shapes.

Although we use a similar architecture as Dreams, we will use a completely different tool flow to overcome the limitations of the current tool flow of Dreams. As mentioned in the previous section, the design flow of Dreams uses a custom router, which causes several restrictions, such as the number of interface signals that can be routed and the device support. In our design flow, we will use the vendor tools for the low-level operations (e.g., routing). By doing this, we can make use of all the built-in optimizations provided by the vendor tools. Also, we can then built grid-style applications for newer FPGA devices.

In the continuation of this section, we discuss the static system and modules in more detail. Finally, we describe which existing tools we use to develop the system that is shown in Figure 4.6.

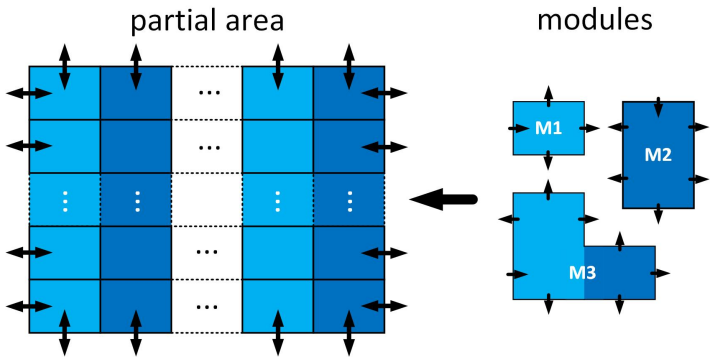

Figure 4.6: The partial area is partitioned in two-dimensional slots. Modules can communicate with the static area by using the interfaces at the boundary.

4.2.1 Static System

In the static system, we split the partial area into $X \times Y$ slots. The partitioning of the partial area in slots is illustrated in Figure 4.7. The slots can have arbitrary sizes. However, the width and height should be equivalent for all slots. The dimension of the slots is expressed in the number of INT tiles. Furthermore, the footprints can vary among slots. As we have seen in Section 2.1.3, the resources of an FPGA are arranged in columns that span the full height of the FPGA. Therefore, each slot that shares the same column will have a similar footprint, but slots located in different columns can vary. For example, in Figure 4.7, slot $(0, 0)$ has an equivalent footprint in comparison to slot $(0, Y)$. However, slot $(0, 0)$ might have a different footprint compared to slot $(X, 0)$, since they are located in separate columns.

The partial area can communicate with the static part through the interface wires attached to all the slots at the border. Each of these slots contains an input signal, named *s2p*. This signal name stands for static-to-partial, which indicates an interface signal from the static area bridging to the partial area. These signals are bind to physical wires and can be seen as a male connector. The modules have a female connector and can be plugged into one of these male connectors. The corner slots contain two inputs from the static area to the partial area and two outputs in the other direction. The reason for this is to enhance module relocation, which is explained in Section 4.2.2.

Furthermore, the interface signals from the partial area to the static part are marked as *p2s*, which stands for partial-to-static. The interface signals of the modules can leave the partial area through one of these connections. Since all the *p2s* signals should be connected to the same signal in the static area, the *p2s* signals are connected to a multiplexer that selects the final output of the partial area. This is illustrated in Figure 4.8. The figure shows a generic static system, where the partial area is divided into $X \times Y$ slots. The partial area is located on the FPGA fabric, and it is connected to the static area through the interface wires (the red and purple wires). For convenience, we use the location of the slot as a prefix of the *p2s* signal names. The postfix of the *p2s* signal names denotes the border side. Finally, the signal *select* selects the output of the partial area.

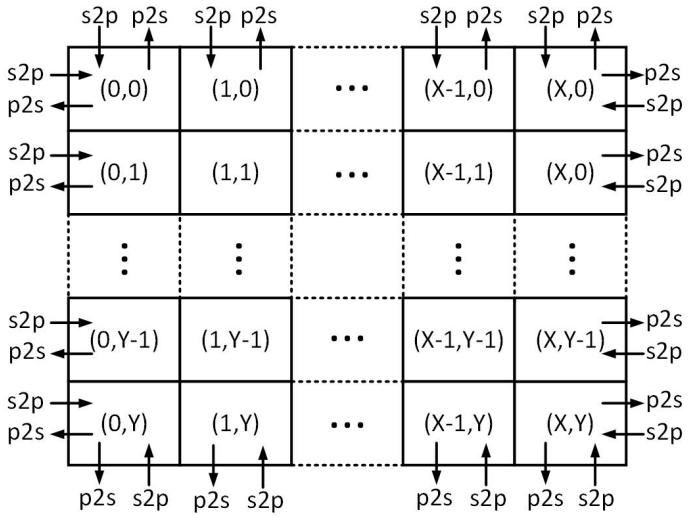

Figure 4.7: The partial area is divided into $X \times Y$ slots when developing the static system. The slots at the border contain the interface wires between the static and partial area.

In Figure 4.8, we can use multiple inputs in parallel. However, it is not feasible to produce results in parallel, since we have only a single output (*p2s*). Therefore, for

parallel operations, the multiplexer must be duplicated. Thus, for N parallel operations, we require N multiplexers and N select signals.

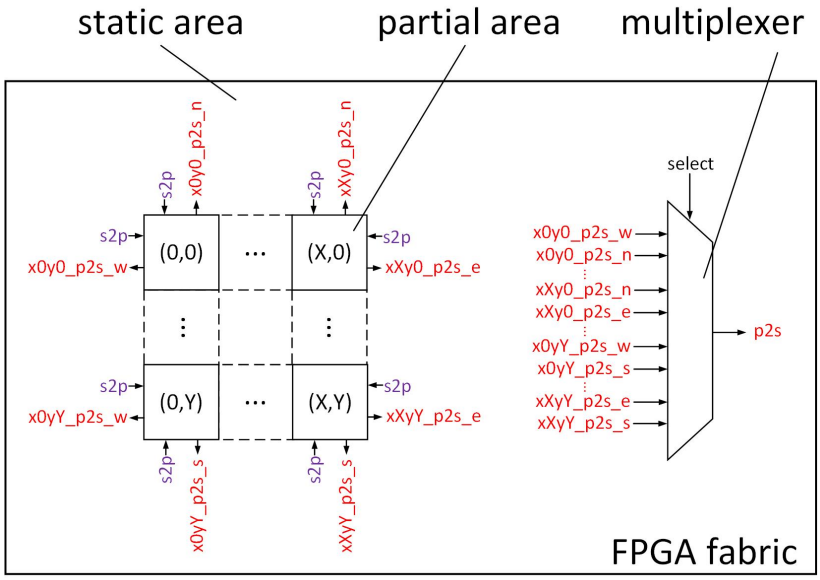

Figure 4.8: The partial area is partitioned into $X \times Y$ slots. The purple wires are the interface wires that connect the static area with the partial area, where the red wires are the interface wires that connects the partial area with the static area. The multiplexer selects the output slot of the partial area.

The multiplexers that select the outputs of the partial area locates within the static area and cause logic overhead to the total system. The amount of logic cost depends on the dimension of the partial area (in terms of slots) and the bus width of the interface signals. In the partial area, each side on the border slots that adjoins to the static area contains a *p2s* interface signal connected to the multiplexer, as illustrated in Figure 4.8. The number of interface signals connected to the multiplexer $MUX_{nr\_of\_signals}$ is the following.

$$MUX_{nr\_of\_signals} = X * 2 + Y * 2 \tag{4.4}$$

The *p2s* signal is a bus with width W. As mentioned in the previous section, the multiplexers are implemented by using LUTs. Therefore, the logic overhead $MUX_{overhead}$ in terms of LUTs by the multiplexer is the following.

$$MUX_{overhead} = W * \left\lceil \frac{MUX_{nr\_of\_signals}}{4} \right\rceil \tag{4.5}$$

Note that if we use N parallel operations in the partial area, we require N multiplexers. The multiplexers are the only elements that cause logic overhead to the

system. Therefore, the total system overhead $SYSTEM_{overhead}$ caused by the DPR implementation is as follows.

$$SYSTEM_{overhead} = MUX_{overhead} * N \tag{4.6}$$

As we have seen, in both GoAhead and our proposed DPR system, we have significant logic overhead due to the multiplexers. In the following, we make a comparison between the two architectures.

In Table 4.1, the logic overhead of both the GoAhead grid-style system and our proposed grid-style system is illustrated. The numbers are derived from respective Equation 4.3 and Equation 4.5. Note that the signal width in Equation 4.1 and Equation 4.2 is denoted by L, where the signal width in Equation 4.5 is indicated by W. In Table 4.1, the signal width is kept variable and is denoted by w.

In Equation 4.3, we obtain that the logic overhead in the GoAhead approach is only dependent on the number of rows. In our proposed grid-style system, the logic overhead is dependent on both the number of rows and the number of columns, as shown in Equation 4.5.

Table 4.1: Comparison of the logic overhead (in LUTs) between the GoAhead approach and our proposed DPR system.

Dimension		Overhead	
Rows	Columns	GoAhead	Proposed system
2	2	$w * 3$	$w * 2$
3	3	$w * 4$	$w * 3$
4	4	$w * 5$	$w * 4$
3	2	$w * 4$	$w * 3$
4	2	$w * 5$	$w * 3$
5	2	$w * 7$	$w * 4$
2	3	$w * 3$	$w * 3$
2	4	$w * 3$	$w * 3$
2	5	$w * 3$	$w * 4$

Now, in Table 4.1, we separate the dimensions of the grid into three categories. First of all, we have calculated the logic overhead for dimensions with the same number of rows and columns. As we can see from the table, the GoAhead approach requires more LUTs in comparison to our proposed system. In the following, we calculated the overhead with a dominant number of rows in comparison to the number of columns. Also, in this case, the GoAhead approach requires more LUTs and

thus causes more logic overhead. Finally, we calculated the logic overhead with a dominant number of columns in comparison to the number of rows. In this case, our proposed system causes more logic overhead in comparison to the GoAhead approach. As a conclusion, the dimension decides whether the implementation of either GoAhead or our proposed system is more efficient in terms of logic overhead.

4.2.2 Modules

In the previous section, we have seen that the partial area is partitioned in slots. With the knowledge of the slot sizes and the footprints of these slots, we can design the modules. The modules must be a continuous area and can occupy one or multiple slots. The modules can occur in different shapes.

The modules are developed such that they can communicate with the interface signals in the static system. This is achieved by binding the interface signals of the modules to the same relative wires as the interface signals in the static system. For module-to-module communication, we use the same relative interface wires. Thus, the interface signals on a module can communicate with both the static area and other modules.

For each module, we define a varying number of cardinal input and output interfaces (north, east, south, and west) on the module. However, for modules with a single upstream, we can assign only a single input interface to the module. The reason is that we cannot connect multiple driver signals to a single signal. A solution to this problem is illustrated in Figure 4.9.

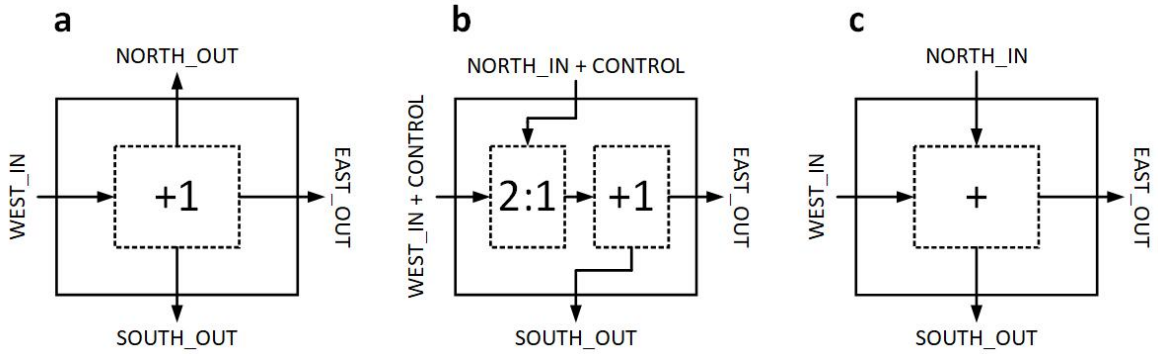

Figure 4.9: (**a**) For modules that require a single upstream, we can only assign a single input interface to the module. (**b**) However, by using control signals and adding an internal multiplexer into the module, we can define multiple input interfaces. (**c**) In the case that we have modules that require multiple inputs, we can have multiple input interfaces without additional logic and control signals.

The figure demonstrates an example of a simple module that behaves as an increment component, where the input signal is incremented by one. An increment module contains a single upstream, and the output can be propagated to multiple outputs, as illustrated in Figure 4.9a. In the case that we would like to have multiple input interfaces to the module, we can add a multiplexer within the module, as shown in Figure 4.9b. The multiplexer determines which input signal is connected to the incrementer. Note that we require additional interface signals for this solution to control the data flow in the module. Also, there is logic overhead because we add a multiplexer to the module.

If a module requires multiple inputs, this is also feasible. For example, in systolic array applications, the modules usually require two or more input interfaces. The components in a systolic array often contain simple basic operations, such as addition and multiplication. An example module in a systolic array is illustrated in Figure 4.9c.

We have seen that a module with a single upstream requires extra interface signals and additional logic to support multiple input interfaces. Therefore, it is more convenient to develop modules with a single input interface. As discussed in the previous section, the corner slots in the partial area contain two input interfaces and two output interfaces. This construction enhances module relocation. The reason is described in the following. Assume that we provide two modules that have a size of 1×1 slots, and one of these modules has an input at the northern border, and the other module has an input at the western boundary. Then, both modules can be configured in the top-left corner. Therefore, we have even more flexibility in the placement of modules. Note that the same analogy applies to the other corner slots.

The communication between modules is only feasible if their respective input and output interfaces are adjacent. Therefore, a set of modules that might be configured at the same time must be developed such that all the input and output interfaces match. However, if we have a lot of different combinations of modules that might be configured in the partial area simultaneously, we might need to develop a module with the same functionality many times, but with different interface layouts. Therefore, it is convenient if there is support for module-to-module communication, even if the interface signals are not adjacent. This feature is introduced in Figure 4.10.

In Figure 4.10a, the partial area is partitioned in 2×5 slots. As described before, the slots contain interface wires in both directions. A set of modules is illustrated in Figure 4.10b. In this example, all the modules contain a single input and output interface. Modules M_1 and M_3 have a dimension of 1×2 slots, where module M_2 has a dimension of 2×3 slots. Note that the modules can only be configured on the locations with the same footprint (same color).

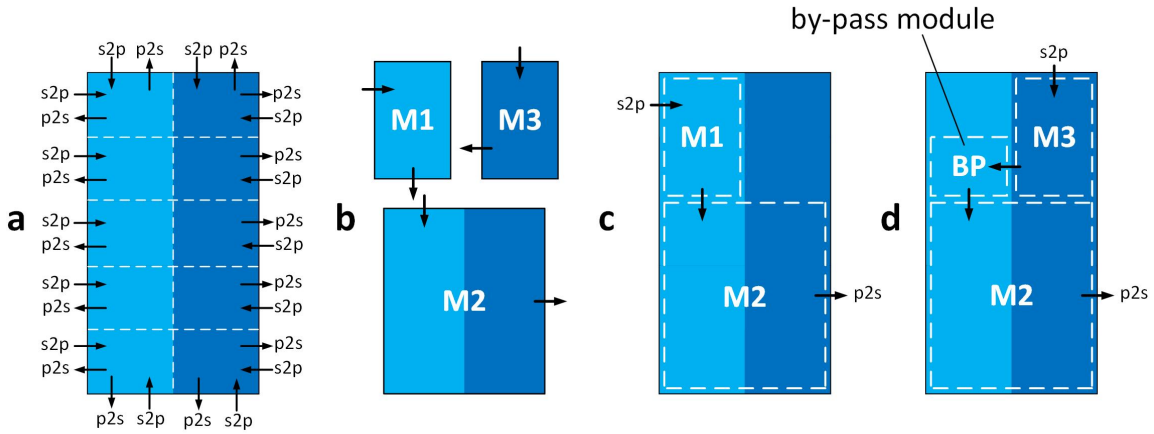

Figure 4.10: (**a**) The partial area is partitioned in 2×5 slots. The slots in the first and second columns contain different footprints. (**b**) We have a set of modules with different dimensions in terms of slots. (**c**) In the first configuration, the *s2p* signal enters module M_1 and leaves the partial area through the *p2s* signal via module M_2. (**d**) In the next configuration, M_3 and M_2 must be connected. However, their respective input port and output port are not connected. Therefore, a by-pass module is configured to connect the two modules.

In Figure 4.10c, two modules are configured. The $s2p$ signal enters module M_1, which communicates with module M_2 within the partial area. Finally, the output of M_2 is connected to the output signal of the partial area ($p2s$). In Figure 4.10d, another configuration is illustrated. In this configuration, the upstream is connected to module M_3, which in turn is connected to module M_2. Again, the output of M_2 is connected to the $p2s$ signal. Module M_2 contains only an input interface on the north side. Therefore, module M_3 must be located on top of M_2. However, module M_3 can only be configured in the right-top corner of the partial area because of its footprint. As a consequence, the modules cannot communicate since the interfaces of these modules are not directly connected. Therefore, we have to configure a so-called *by-pass module* that connects these modules. These modules are developed such that their input is directly connected to their output. The by-pass modules require only routing resources, and therefore, they can be developed on all footprints. Note that by using by-pass modules, we require additional reconfiguration time, because these modules need to be configured as well.

4.2.3 Planning Phase

In this work, we develop a framework to build DPR systems that are described in the previous sections. However, before the designer can start developing such a

DPR system, the designer has to consider several crucial aspects, which are the following.

- What part of the system should be reconfigurable, and what part remains the same during run-time?
- Which modules might be configured in the partial area at the same time?
- Where should the modules be located in the partial area?
- What should be the size of the partial area?
- Where should the partial area be located on the FPGA fabric?
- In how many slots should the partial area be partitioned?
- Which signals should the interface signals *s2p* and *p2s* include?

All these questions are part of the *planning phase*. The planning phase is the first step when developing a DPR system using our tool flow. The planning phase must be performed completely manually. For convenience, we categorize all the questions stated above in the following five sub-phases.

1. Static/partial partitioning.
2. Configuration definitions.
3. Resource budgeting.
4. Floorplanning.
5. Interface specification.

In the continuation of this section, we provide suggestions and introduce several formulas for all these sub-phases that might help the designer to complete the planning phase.

Static/Partial Partitioning

In the *static/partial partitioning* phase, we examine the existing design. The parts of the design that requires to be operational the whole time belong to the static portion, where the parts of the design that are used mutually exclusively and can share the same FPGA resources at different predefined times belong to the partial part. Modules might be identified by multiplexers that are used to switch between functional blocks. These modules are possible candidates to be shared in one partial

area. Thus, instead of switching the multiplexer for selecting an exclusive mutual function, DPR can be used for this purpose. However, while in most cases the multiplexer can be switched within a single clock cycle, the reconfiguration takes time in the range of many thousands to millions of clock cycles. The actual number of clock cycles depends on the size of the module.

Configuration Definitions

In the following, we describe the *configuration definitions* phase. In Section 2.2.3, we described a configuration as a complete design with one module for each reconfigurable region. However, in this chapter, we define a configuration as a set of modules that are configured within the same partial area simultaneously. For the following, assume that we have a set of modules $M = \{m_1, m_2, \ldots, m_X\}$ that are used for a single partial area. The configurations C are then defined as follows.

$$C = \{c_1, c_2, \ldots, c_Y\} \tag{4.7}$$

$$c \subseteq M \tag{4.8}$$

Resource Budgeting

Now, to determine the size of the partial area that will host the partial modules, we have to identify the resource consumption of each configuration in C. If we know the maximum resource consumption of all the configurations, then we can calculate the minimal required number of logic blocks and thus the minimal size of the partial area. This phase is called *resource budgeting*. We define the function $r_m(m)$ that returns the resource consumption of the given module. The function returns a triplet $r = (L, B, D)$. The symbols L, B, and D respectively indicate the number of LUTs, BRAMs, and DSP blocks required by the module.

The resource consumption for each module can be determined by synthesizing the module by using one of the vendor tools. We have to add the resource consumption of all modules belonging to the particular configuration to get the total resource consumption for a single configuration. We add the resource consumption of two modules as follows.

$$r_m(m_i) + r_m(m_j) = \{L_i + L_j, B_i + B_j, D_i + D_j\} \tag{4.9}$$

We define the function $r_c(c)$ that returns the resource consumption of a given configuration. Similar to function $r_m(m)$, this function also returns a triplet r. The resource consumption of a configuration is calculated as follows.

$$r_c(c) = \sum_{\forall_{m \in c}} r_m(m) \tag{4.10}$$

After determining the resource requirements for each $c \in C$, we have to calculate the maximal resource consumption among all configurations. We calculate the maximal resource consumption for different configurations as follows.

$$max(c_i, c_j) = \{\max(L_i, L_j), \max(B_i, B_j), \max(D_i, D_j)\} \tag{4.11}$$

Finally, we use the following formula to calculate the minimal resource consumption of the partial area.

$$AREA\_RES_{min} = \max_{\forall_{c \in C}}(r_c(c)) \tag{4.12}$$

Note that the way we calculate the minimal resource consumption for the partial area $AREA\_RES_{min}$ is only valid for a continuous area. The reason is that each module in the partial area usually suffers from a small portion of internal fragmentation, even if the partial area is partitioned in small slots. Therefore, the designer has to consider to increase the size of the partial area to compensate for the internal fragmentation. Also, usually, it is good practice to increase the partial area slightly to avoid routing congestion.

Floorplanning

Now that we have determined resource consumption by the configurations, and thus the minimum size of the partial area, we have to determine the location of the partial area on the FPGA fabric: floorplanning. Floorplanning is a crucial step in DPR systems as it influences the later placement and routing of both the static system and the modules. In Chapter 3, we have seen that some academic DPR tools include automated floorplanning. These tools calculate the most efficient placement of the partial area onto the FPGA structure.

The proposed floorplanners by the academic DPR tools are designed for island-style reconfiguration. As discussed in Section 2.2.2, in island-style, only one module is hosted in the partial area exclusively. Although we use a grid-style approach, we might use one of these floorplanners.

These floorplanners start by determining the resource consumption r for each module individually. The minimal size of the partial area in island-style is defined as the maximum number of each resource type (CLB, DSP, and BRAM) among all

modules sharing the same area. Based on this information, the automated floorplanners use built-in algorithms to calculate the location of the partial area onto the FPGA fabric.

If we look at the perspective of the resource consumption of the partial area, the difference of grid-style reconfiguration in comparison with island-style reconfiguration is that we configure multiple modules in the partial area simultaneously. Therefore, if we specify the minimal size of the partial area as $AREA\_RES_{min}$, we can use one of these tools to determine the location of the partial area.

However, there are some crucial points not considered by the automated floorplanners. First of all, in our DPR system, we partition the partial area into two-dimensional slots. One of the features that our system supports is module relocation among different slots. As already mentioned earlier in this chapter, the footprint of the slots in the same column is equivalent. Thus, module relocation in the vertical direction is always feasible. However, as the footprint in the horizontal direction is heterogeneous, module relocation in the horizontal direction is not always possible. Therefore, to enhance module relocation, it is crucial to maximize the consistency among slots in the horizontal direction. This is not taken into account by the floorplanners provided by the academic tools.

Another critical point that is not considered is the location of the modules in the partial area that belongs to a single configuration. The modules must be placed such that their footprint match with the location on the FPGA fabric. Also, the modules must be located such that they are adjacent. Note that we might require by-pass modules for this purpose. In [18], an algorithm is proposed that determines the locations of modules into the grid-style framework provided by GoAhead. However, as our grid-style framework has significant differences, this algorithm might be not very useful, although the algorithm could be used as a starting point.

The last decision by the designer in the floorplanning phase is the partitioning of the partial area. Usually, the goal is to partition the partial area in as many slots as possible. This minimizes the internal fragmentation. However, the minimal size of the slots is constrained by the architecture of FPGA. In Chapter 5, these constraints will be discussed. Note that the minimum number of slots in the partial area should be the maximum number of modules in a single configuration. The reason is that a module occupies at least one slot, and in principle, modules cannot share a single slot.

Interface Specifications

The interface signals *s2p* and *p2s* should contain all the different input and output signals of the modules, respectively. This way, we can place a module at all slots

that are adjacent to the static area. Note that this only holds if the footprint at that particular location matches the hardware layout of the module.

We define the function $i_m(m)$ that returns the interface signals of a given module. Now, to determine all the different interface signals I for all modules belonging to M, we can use the following formula.

$$I = \bigcup_{\forall_{m \in M}} i_m(m) \tag{4.13}$$

4.2.4 Tool Flow

As already mentioned, we will use the vendor tools for the low-level operations to make use of the built-in optimizations by the vendor tools. GoAhead is such a tool. GoAhead generates constraints that are passed into the vendor tools, and the vendor tools perform the low-level operations. Also, GoAhead provides a GUI and scripting language that makes the development of DPR systems more convenient.

Therefore, as a starting point, we will use GoAhead and use the GUI and a few existing scripting commands to build our system. We extend the tool with several custom commands. In the next chapter, the implementation of the system is described.

Chapter 5

Implementation

This chapter describes the implementation of the proposed concept for a fine-grained DPR system, as described in Chapter 4. The design flow of the static system and the modules are completely separated. The basic idea of both design flows is to use GoAhead to generate constraint files (in TCL format) and VHDL templates. The VHDL templates must be merged with the existing design files. Furthermore, the TCL files must be included in a Vivado project to incorporate with the low-level device-dependent operations such as placement, routing, and bitstream generation.

The tool flow of both the static system and modules is illustrated in Figure 5.1. Note that this design flow is similar to the design flow shown in [1]. However, the generated constraint files by GoAhead will be different, since we build grid-style systems with a completely different architecture.

As already discussed in the previous section, the tool flow starts with a planning phase. This phase must be performed entirely manually by the designer. In Section 4.2.3, several suggestions and formulas are provided that might help the designer to complete the planning phase. After the planning phase, the floorplan of the static design and modules are created in GoAhead by using its GUI and scripting language. Based on these floorplans, GoAhead generates the corresponding VHDL templates and TCL scripts. These files are respectively merged into the existing VHDL files and included in Vivado. Then, we provide TCL scripts to automate the tool flow in Vivado. The final result is a full bitstream (static system) and a partial bitstream for each module. The partial bitstreams are generated by using the tool BitMan [17]. Once the bitstreams for the static system and modules are generated, we can program the bitstreams into the FPGA. The full bitstream, which represents the static system, should be configured first. Later on, during run-time, the modules can be configured by using the partial bitstreams.

This chapter is organized as follows. In Section 5.1, we introduce the GUI and scripting language of GoAhead. In the following, we describe the implementation of the static system. We explain all the steps to develop the static system and

provide the corresponding GoAhead commands for these steps. This is all part of Section 5.2. Finally, in Section 5.3, we describe the implementation of the modules and how we generate the partial bitstreams.

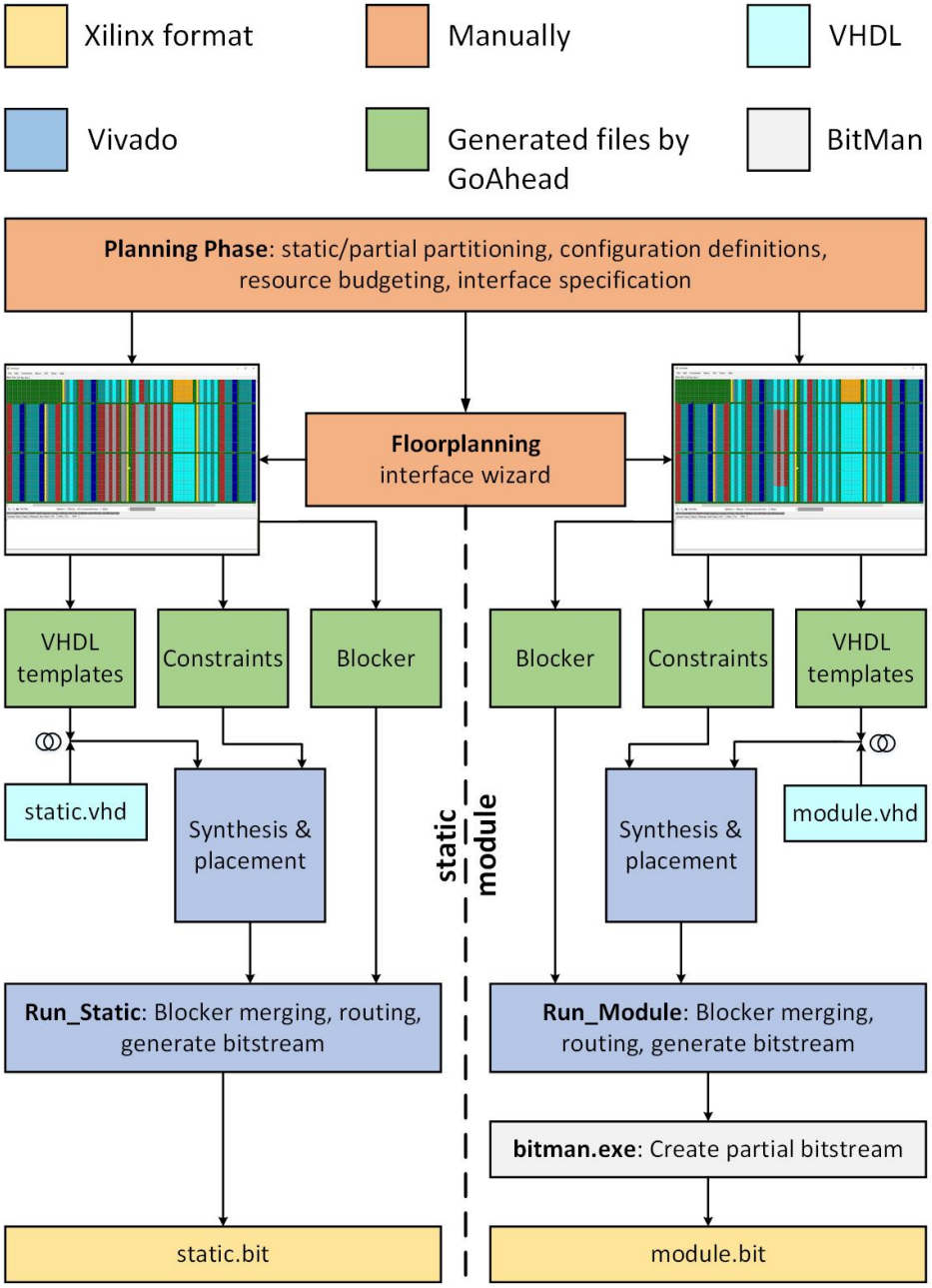

Figure 5.1: The design flow of both the static system and modules. The design flow of the static system and modules is completely separated. In both flows, we floorplan the design with GoAhead by using its GUI and scripting language. Then, GoAhead generates the corresponding VHDL templates and constraint files that are used in Vivado to perform all the low-level operations. The final output is a full bitstream that represents the static system and a partial bitstream for each module.

5.1 Basics of GoAhead

GoAhead provides an intuitive GUI and a scripting language. For all the actions performed by the user on the GUI, there exist corresponding scripting commands. The corresponding commands are generated in an output window on the GUI, as illustrated in Figure 5.2. We can reproduce results by copying/pasting the recorded commands in a script file (extension *.goa*) and run this script with the GoAhead tool.

Language and GUI

GoAhead contains around 250 commands. In this work, we will use some existing commands and add several new commands to the tool. These new commands allow us to efficiently implement the proposed grid-style system that is described in Chapter 4. The general structure of a command is as follows. The command starts with the command name. The command names can be significantly long, but often gives a clear indication of its functionality. A command might contain parameters. The name of the parameters and their corresponding values are separated by an equal sign. It is not allowed to use any white space between the parameter name, equal sign, and the parameter value. White space indicates the separation of the command name and its parameters. GoAhead also supports lists as value for the parameters. The parser considers every comma as a separation between different list items. Finally, the command ends with a semi-colon. The hash-sign (#) is used to support comments in the script. An example command is shown in Listing 5.1.

```
1 # this is a comment
2 CommandName FirstParameter=Item1,Item2 SecondParameter=Second;
```

Listing 5.1: An example command in the GoAhead scripting language. The command starts with the command name, then following the parameters and ends with a semi-colon.

Usually, it is convenient to combine multiple commands, for example, in commands that often occur in a fixed and consecutive order. We can build a new command that merges multiple commands by using the command *AddAlias*. This command takes two parameters: *AliasName* and *Commands*. The first parameter specifies the name of the new command, where the second parameter includes all the commands (with parameters) to execute. Note that we cannot add parameters to this new command. Therefore, the parameters of the commands within the new command are fixed.

Now that we have seen how the command script is organized, we will have a

look at the GUI of GoAhead. Before we can start GoAhead, we first have to set one environment variable to run GoAhead: *GOAHEAD_HOME*. This variable points to the home directory of GoAhead. In this directory, all the files related to GoAhead are stored. We can start the GoAhead application by executing the command "*goahead*" in the command prompt (Windows).

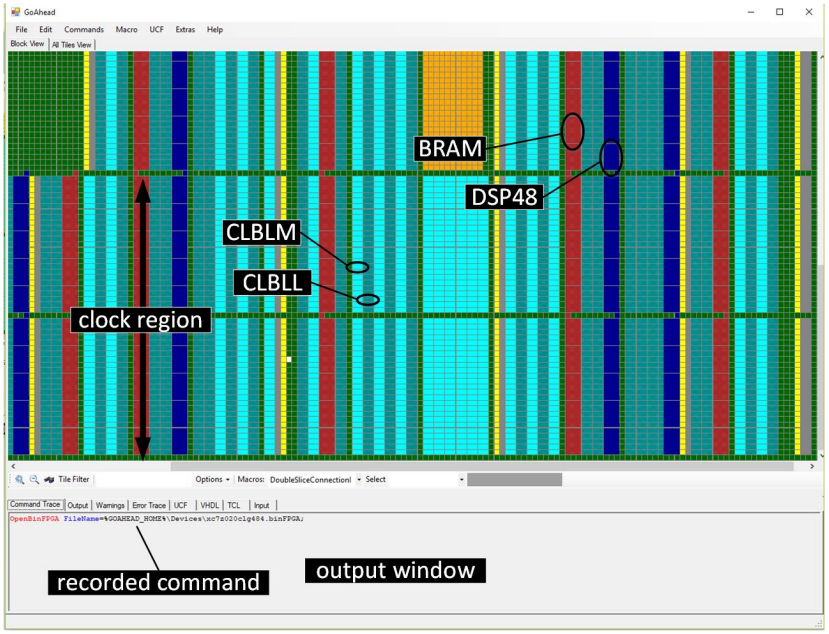

Figure 5.2: The interface of GoAhead after loading a specific FPGA device. The layout shows all the tiles of the device. Each type of tile has a unique color. Recorded commands are displayed in the output window.

We see an empty (black) screen when we have started the GoAhead application because we have not specified yet which FPGA device we work with. In this work, we are using the ZedBoard, which includes an *xc7z020clg484* device. We can load a device by using the command *OpenBinFPGA*. Listing 5.2 shows the syntax to use this particular command. After executing this command, the interface of GoAhead shows all the tiles of the FPGA on the GUI, as illustrated in Figure 5.2. The different primitives (e.g., CLBs, BRAMs, DSP48 blocks, etc.) have different colors such that we have a clear overview of the internal structure of the FPGA.

GoAhead includes a specific script for each device family. The corresponding family script of a device is executed when we load a particular FPGA device. For example, if we load a device from the Zynq family, the GoAhead script named *Zynq.goa* will be executed. This script contains commands that are family-related. Later on in this chapter, we use this script to execute some particular commands to implement our system for specific device families.

```
1 # load device
2 OpenBinFPGA
3   FileName=%GOAHEAD_HOME%\Devices\xc7z020clg484.binFPGA;
```

Listing 5.2: The command *OpenBinFPGA* is used to load a specific FPGA device.

Selection Commands

We mainly use the GUI to select parts of the FPGA fabric. In GoAhead, we can choose clusters of tiles and store these selections. Later on, we can restore the collection by a single command. In Listing 5.3, the commands are shown to select, store, and restore the selection. In the following, the commands that perform these actions are described.

```
1  # select and store the partial area
2  ClearSelection;
3  AddBlockToSelection
4    UpperLeftTile=INT_L_X26Y99
5    LowerRightTile=INT_R_X31Y0;
6  ExpandSelection;
7  StoreCurrentSelectionAs UserSelectionType=PartialArea;
8
9  # restore the partial area
10 ClearSelection;
11 SelectUserSelection UserSelectionType=PartialArea;
```

Listing 5.3: The commands in GoAhead to select, store, and restore a specific area on the FPGA structure. In this example, the partial area is selected, stored and restored.

- *ClearSelection*: This command clears the current selection of tiles.
- *AddBlockToSelection*: This command selects a rectangular group of tiles, where the two parameters specify the upper-left and lower-right tile.
- *ExpandSelection*: This command expands the current selection of tiles. As discussed in Section 2.1.3, the logic primitives are paired with one or multiple INT tiles. By using this command, all the missing tiles from these pairs are being added to the current selection.

- *StoreCurrentSelectionAs*: This command is used to save the current selection. We can give the selection a unique name by using the parameter *UserSelectionType*.
- *SelectUserSelection*: This command is used to restore a particular selection. The parameter *UserSelectionType* is used to specify the name of the selection being restored.

File Generation Commands

As described before, we use GoAhead to generate specific files. The commands that produce these files contain at least the following parameters: *FileName*, *Append*, and *CreateBackupFile*. In the following, these parameters are described.

- *FileName*: This parameter specifies the name of the file.
- *Append*: This parameter is a boolean type. If the parameter is set to True, GoAhead appends the content to the existing file. Otherwise, GoAhead will clear the file before writing the new content.
- *CreateBackupFile*: This parameter is of type boolean, and specifies whether to generate a backup of the old file. The extension of the backup file is *.bak*.

5.2 Implementing the Static System

The static design contains all the logic that remains unchanged during run-time. The difference compared to the standard design flow, as discussed in Section 2.1.2, is that we include a partial area in the static system. The static area and partial area require interface signals for communication. The basic idea is that we add a black box (in VHDL) to the top-level design that acts as the partial area. The black box includes all the interface wires, such as in Figure 4.7. Another difference compared to the standard design flow is that we have to constrain the placement and routing. Namely, it is prohibited for the placer to place any logic belonging to the static system within the partial area. Also, it is prohibited for the router to route signals belonging to the static system through the partial area. We set this constraint to support module relocation, as discussed in Section 2.2.3. The router also has to be instructed to route the interface signals between the static and partial area correctly, such that communication between these two regions is feasible.

We use the GoAhead tool to generate the VHDL templates and constraint files in TCL format. The generated VHDL templates are merged within the existing design files, where the TCL scripts are included in the Vivado project. Vivado is used to

perform the low-level operations, such as synthesis, implementation, and bitstream generation. In the continuation of this section, we describe the commands that we use in GoAhead to generate the VHDL templates and constraint files. Furthermore, we discuss the content of these files in more detail.

5.2.1 Synchronous Systems

In the design phase, we have determined the location of the partial area onto the FPGA fabric. We have to define this location in GoAhead. This can be done by using the GUI or scripting language. If the scripting language is used, we can use the standard selection commands, as in Listing 5.3.

After we specified the location of the partial area in GoAhead, we connect the BELs within the partial that contain a clock pin to the clock network. This way, we can run synchronous circuits within the partial area. The way we implement the clock routing towards the partial area is similar as in [1]. All the BELs that contain a clock pin in the partial area are connected to the clock network. By doing this, we make sure that all the clock paths towards all the BELs in the partial area are routed in the static area.

The Command *ConnectClockPins*

The command that we use in GoAhead to connect all the BELs in the partial area is *ConnectClockPins*. In Listing 5.5, the syntax of this command is shown. In the following, the parameters of the command *ConnectClockPins* are described.

- *ClockPin*: This parameter specifies the name of the clock pin.
- *BELs*: This parameter specifies the names of the BELs that must be connected to the clock net. The names of the BELs are specified in regular expression format.
- *ClockNetName*: This parameter specifies the name of the clock net. The clock pins will be connected to this clock net.

The command *ConnectClockPins* connects only the BELs in the currently selected area. Therefore, we first have to select the tiles that belong to the partial area before using *ConnectClockPins* command, as illustrated in Listing 5.5.

The TCL Commands

The command *ConnectClockPins* generates a TCL script that connects the clock pins to the clock net. In Listing 5.4, the TCL commands are shown that connects the

clock pin from a single FF to the clock net. The command *create_cell* instantiates one or multiple cells in the current design [33]. We instantiate an existing cell from the library. The library cell of a FF is named FDRE. FDRE is a D-type FF. These types of FFs have data, clock enable, and synchronous reset inputs and data output. We can place a cell on the FPGA fabric with the command *place_cell*. Once we have placed the cell, we have to define the clock pin on the cell. We can define a pin on a particular cell with the command *create_pin*. Finally, we connect the clock net to the defined clock pin with the command *connect_net*. These four TCL commands are repeated for all the BELs that match the regular expression of the parameter *BELs* in the command *ConnectClockPins*. For example, in Listing 5.5, all the FFs in the partial area are connected to the clock net.

```
1 # connect clock pin to clock net
2 create_cell -reference FDRE SLICE_X56Y49_DFF
3 place_cell SLICE_X56Y49_DFF SLICE_X56Y49/DFF
4 create_pin -direction IN SLICE_X56Y49_DFF/C
5 connect_net -hier -net clock_50MHz
6             -objects {SLICE_X56Y49_DFF/C}
```

Listing 5.4: The TCL commands to connect the clock net to the clock pin.

```
1  # select the partial area
2  ClearSelection;
3  SelectUserSelection UserSelectionType=PartialArea;
4
5  # connect clock pins in the current selected tiles
6  ConnectClockPins
7    ClockPin=C
8    BELs=[A-D]FF
9    ClockNetName=clock_50MHz
10   FileName=static_connect_clockpins.tcl
11   Append=False
12   CreateBackupFile=True;
```

Listing 5.5: In GoAhead, the command *ConnectClockPins* is used to connect all the clock pins of the BELs to the clock net in the partial area.

5.2.2 Interface Constraints

Until this point, we have defined the location of the partial area and generated a TCL script to connect the clock pins of the BELs within the partial area. As a next step, we create constraints for the interface signals between the partial area and the static area. As illustrated in Figure 4.7, all the slots at the boundary of the partial area contain interface signals. In the planning phase, we derived the unioned interface. The input signals of the unioned interface are connected to each *s2p* signal, where the output signals of the unioned interface are connected to each *p2s* signal.

Binding the Interface Signals

We bind the interface signals to the physical wires that cross the border from the static area to the partial area and vice versa. As described in Section 2.1.3, the wires are connected to the INT tiles. It is essential to keep in mind that the physical wires itself are not programmable. However, the INT tiles contain PIPs that can be enabled to route a particular signal to one or multiple locations. In Figure 5.3, we demonstrate how we implement the static system and modules such that they can communicate.

In Figure 5.3, we bind four interface signals to four physical wires. Each signal has a different color (green, purple, red, or blue). In Figure 5.3a, we implement the static system. In the static system, the router can freely route through the static area. However, we make sure that the interface signals are routed through the specific interface wires. We cannot leave a signal unconnected. This means that once we have routed the signal through a particular interface wire, we have to connect the signal to an element. Otherwise, it remains a partially routed signal. Therefore, we use LUTs within the partial area to connect the interface signals.

In parallel with the static system, we can develop the modules. A module implementation is illustrated in Figure 5.3b. We specify a location on the complete FPGA fabric that is used to implement the module. The logic elements and signals belonging to a module are completely placed and routed within the module area. Note that when designing the module, only the module area is of any importance. The reason is that we create a partial bitstream only from this region. Consequently, the rest of the FPGA fabric is not of any importance.

The interface signals are entering the module area the same way as in the static system. In the static system, we used LUTs within the partial to connect the interface signals. However, when developing modules, we use LUTs outside the module area to connect the interface signals.

Now that we have developed the static system and one or multiple modules, we configure these on the FPGA. First, we configure the static system. This means

that the configuration onto the FPGA fabric is similar to the configuration shown in Figure 5.3a. Next, we can reconfigure the partial area for one or multiple modules. Note that we configure only the module area of Figure 5.3b in the partial area because the partial bitstream is generated from this region. Since the INT tiles and CLBs are reconfigurable, only their configuration will change when reconfiguring the FPGA. This will result in a configuration that is illustrated in Figure 5.3c. In fact, we exchanged the partial area from Figure 5.3a with the module area from Figure 5.3b. The static system and module are properly connected since they use the same interface wires.

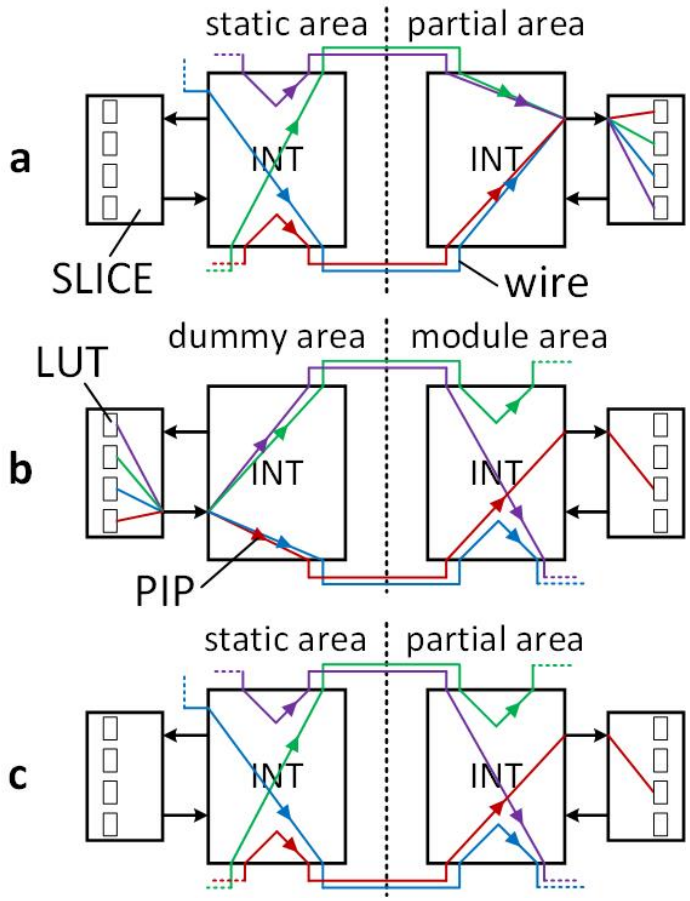

Figure 5.3: (**a**) In the static system implementation, we bind the interface signals to specific wires and connect them to LUTs within the partial area. (**b**) In the module implementation, we bind the same interface signals to the same specific wires and connect them to LUTs outside the module area. (**c**) Now, if we configure a module into the partial area, the static system and module are connected since they use the same interface wires.

Interface Wires Calculations

Now that we have seen how binding works, we will explain the number of interface wires that can be used in a particular region. In Figure 5.3, we bound four interface signals to four physical wires. Note that the physical wires are unidirectional. Therefore, in this figure, the signals can only propagate from the static area towards the partial area. Thus, we established a communication link in the eastern direction. We can communicate in all the other cardinal directions as well by using wires that route towards different directions.

As explained in Section 2.1.3, the INT tiles contain wires that route towards all cardinal directions. Also, the INT tiles contain wires with different lengths that direct to the same cardinal direction. The wires that route to the same cardinal direction and span the same distance are bundled in groups of 4 wires.

In the following, we explain the number of interface wires that can be used based on the available wire lengths onto the FPGA fabric. In this explanation, we use Figure 5.4. In this figure, only the wires that point towards the eastern direction are illustrated. Each INT tile contains the same wires. In this particular example, the INT tiles contain wires in the east direction with a length of one and two. For binding the interface signals, we use the wires that cross the border from the static area (left-side) to the partial area (right-side).

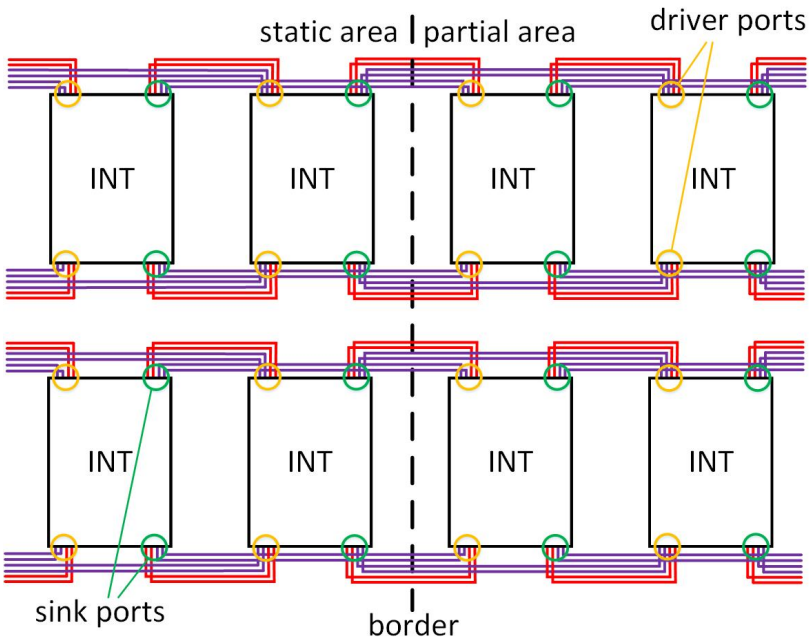

Figure 5.4: We use the physical wires that bridge from the static to partial (and vice versa) as an interface between the static and partial area. In this example, 24 wires are crossing the border from the static area to the partial area.

Now that we have seen which wires we use for binding the interface signals, we describe how to calculate the number of wires that bridge the border. In Figure 5.4, the red wires have a length of one, and the purple wires have a range of two. We can calculate the number of wires that cross the border for one row as follows. The length of the purple wire is two. Therefore, we can use a maximum of two INT tiles for these specific wires. Since wires of an INT tile with the same direction and length are bundled in groups of four, we have $2 * 4 = 8$ wires that cross the border from the static area to the partial area. In the same row, we also have wires in the eastern direction with a length of one. For these specific wires, we can use only a single INT tile in the same row. Therefore, the number of wires with a length of one that bridge the border is $1 * 4 = 4$. Thus, the total number of wires that cross the border for a single row is $4 + 8 = 12$. However, we can repeat this for multiple rows. In Figure 5.4, we have two rows. Therefore, we have a total of $2 * 12 = 24$ wires that we can use for the interface between the static and the partial area in the eastern direction.

In the previous example, only wires in the eastern direction were considered. However, we can apply similar calculations in the other cardinal directions, as well. In the following, we provide some general formulas that designers can use to calculate the number of wires that bridge the border in a particular direction.

Before we provide these formulas, we have to mention that the wire lengths in the western and eastern directions are the same in the Series7 architecture. Also, the lengths of the wires in the northern and southern directions are equally. Therefore, we distinguish the lengths of the wires in the horizontal direction and the vertical direction. The way we calculate the number of interface wires per row and column is then as follows. We define sets $L_h = \{l_1, l_2, \ldots, l_X\}$ and $L_v = \{l_1, l_2, \ldots, l_Y\}$ that contain all the wire lengths in the respective horizontal and vertical direction. Then, the maximum number of wires that cross the border in a single row max_wires_{row} and column max_wires_{column} is the following.

$$max\_wires_{row} = \left(\sum_{\forall l \in L_h} l \right) * 4 \tag{5.1}$$

$$max\_wires_{column} = \left(\sum_{\forall l \in L_v} l \right) * 4 \tag{5.2}$$

Until this point, we have calculated the maximum number of wires per row and column that we can use for the interface. By using these results, we can calculate the minimal required number of rows and columns for a specific amount of interface wires. As described in Section 4.2.3, in the planning phase, we derive the unioned interface from all the interfaces of the modules. The total number of wires $interface\_wires_{total}$ can be derived from this unioned interface. Now,

we can calculate the minimal required number of rows $nr\_of\_rows_{min}$ and columns $nr\_of\_columns_{min}$ to bridge all the interface wires from the partial area to/from the static area with the following formulas.

$$nr\_of\_rows_{min} = \left\lceil \frac{interface\_wires_{total}}{max\_wires_{row}} \right\rceil \tag{5.3}$$

$$nr\_of\_columns_{min} = \left\lceil \frac{interface\_wires_{total}}{max\_wires_{column}} \right\rceil \tag{5.4}$$

As illustrated in Figure 4.7, we use the unioned interface for each slot at the border. Consequently, $nr\_of\_rows_{min}$ and $nr\_of\_columns_{min}$ are respectively the minimum height and width of the slots. However, there are more circumvents that constrain the minimum width and height of the slots. One of them is the lengths of the wires that are used for the interfaces. The width and height of the slots should be equal or greater than the maximum length of the wires in the respective horizontal and vertical directions. The reason is that if we use wires that are longer than the width or height of the slots, the interface wires jump over the neighboring slots. As a consequence, adjacent modules cannot communicate. For example, in Figure 5.4, assume that we use slot sizes of 1×1 INT tiles. Consequently, the partial area is divided into 2×2 slots. Now, suppose that we use wires with a length of two, and we place modules with a dimension of 1×2 in the first and the second column. Then, these modules cannot communicate, as the interface wires span over the slot location of the other module.

Obvious, it is not required to use all the wires for the interfaces. The used wire lengths in a particular application in the horizontal and vertical direction are respectively $l_h \subseteq L_h$ and $l_v \subseteq L_v$. Then, the maximum length in the horizontal direction $wire\_length\_horizontal_{max}$ and the maximum length in the vertical direction $wire\_length\_vertical_{max}$ of the wires that are used in a particular application for the interface are the following.

$$wire\_length\_horizontal_{max} = max(l_h) \tag{5.5}$$

$$wire\_length\_vertical_{max} = max(l_v) \tag{5.6}$$

The final constraint for the minimum width and height of the slots is the granularity of configuration that is feasible by the architecture of the FPGAs. In the modern FPGA series, the configuration memory is arranged in *frames* [31]. These frames are the smallest addressable segments of the device configuration space. In Section 2.1.3, we have seen that the Xilinx FPGAs are composed of tiles. For the devices that belong to the Zynq family, the minimal region that can be configured

onto the FPGA fabric is 1×50 tiles. Therefore, the minimal logic resources that can be configured are the following.

- **CLB**: 1×50 tiles
- **DSP**: 1×10 tiles
- **BRAM**: 1×10 tiles

Now that we have seen all the circumvents that constrain the minimum width and height of the slots, we can calculate the minimum width of a slot $slot\_width_{min}$ and the minimum height of a slot $slot\_height_{min}$ as follows.

$$slot\_width_{min} = \max(nr\_of\_columns_{min}, wire\_length\_horizontal_{max}, 1) \quad (5.7)$$

$$slot\_height_{min} = \max(nr\_of\_rows_{min}, wire\_length\_vertical_{max}, 50) \quad (5.8)$$

The Command *PrintInterfaceConstraintsForSelection*

In the following, we will describe the implementation of the interfaces in GoAhead. The command that we use to specify the interface wires is *PrintInterfaceConstraints-ForSelection*. An example of this command is shown in Listing 5.6. Before we use this command, we have to select the location of the interface. This location should be within the partial area and located adjacent to the static area. We can use the GUI or the usual selection commands for this purpose, as discussed in Section 5.1. In Figure 5.5, an example is shown that selects the location of the interface in two slots at the west border. In this example, the partial area is partitioned into four imaginary slots. The reason that we say imaginary slots is because we select the partial area as one large block. However, we have to keep the imaginary slots in mind when defining the locations of the interfaces. The reason is that the interfaces must be located at the same relative positions in all slots. By doing this, the interfaces of the modules are bound to the same relative wires. Therefore, modules that communicate via an interface at one slot can also communicate via the interface at the other slots. As a result, module relocation is feasible. As illustrated in Figure 5.5, the interface at the west border in slot $(0, 0)$ locates at the same relative position as in the slot $(0, 1)$.

Previously, we calculated the minimum width and height of the slots. We can reuse a few of these formulas to calculate the size of the interface selection, as well. First of all, we have to consider the maximum length of the wires that are used. As illustrated in Figure 5.4, the maximum wire length constrains the maximum number of INT tiles that we can use for the interface selection in the either horizontal

or vertical direction. The reason is that if we use more INT tiles in a particular direction, the wires are too short to bridge the border from the static area to the partial area or vice versa. In the specific example of Figure 5.4, the maximum length is two, and therefore, we can use a maximum of two INT tiles. Furthermore, we might require multiple rows or columns to bridge all the interface wires from one side of the border to the other side. These numbers are calculated in respectively Equation 5.3 and Equation 5.4. Based on these conditions, the minimum width $vertical\_interface\_width_{min}$ and the maximum height $vertical\_interface\_height_{max}$ for the interface selection for communication in the vertical direction are as follows.

$$vertical\_interface\_width_{min} = nr\_of\_columns_{min} \tag{5.9}$$

$$vertical\_interface\_height_{max} = wire\_length\_vertical_{max} \tag{5.10}$$

Furthermore, the maximum width $horizontal\_interface\_width_{max}$ and minimum height $horizontal\_interface\_height_{min}$ for the interface selection for communication in the horizontal direction is as follows.

$$horizontal\_interface\_width_{max} = wire\_length\_horizontal_{max} \tag{5.11}$$

$$horizontal\_interface\_height_{min} = nr\_of\_rows_{min} \tag{5.12}$$

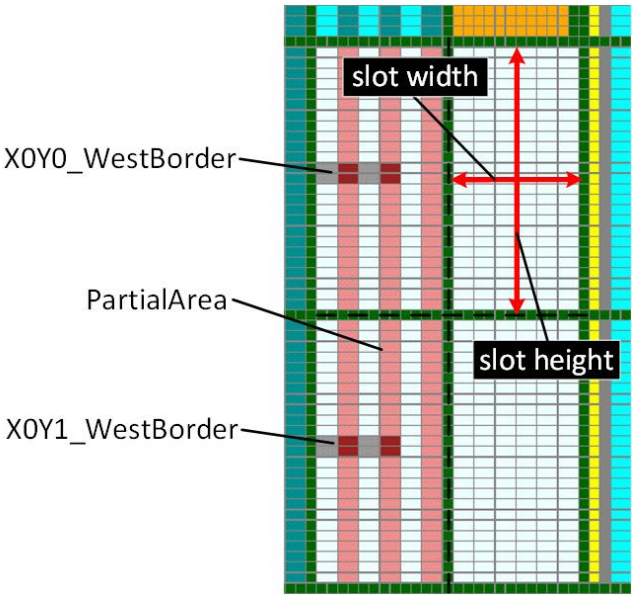

Figure 5.5: The partial area is divided into 2×2 slots. We have to select the area where we would like to define the interface wires between the static and partial area before using the command *PrintInterfaceConstraintsForSelection*.

The command *PrintInterfaceConstraintsForSelection* takes several parameters. In the following, we describe the parameters of this command.

- *SignalPrefix*: This parameter specifies the prefix of the signal names from the signals that are bound to the interface wires.
- *InstanceName*: This parameter specifies the instance name of the partial area component in VHDL. Namely, later on, we generate a VHDL template that can be added to the existing design files. This template contains all the interface signals of the partial area, and the interface signals of this template are being constrained by the command *PrintInterfaceConstraintsForSelection*. See Section 5.2.3 for more information.
- *Border*: This parameter specifies the boundary on the slot (North, East, South, or West) that is used for the interface.
- *NumberOfSignals*: This parameter specifies the number of signals that are used for the interface. If the selected region for the interface provides more interface wires in comparison to this number, then the redundant wires will not be used.
- *PreventWiresFromBlocking*: This parameter prevents the interface wires from blocking. See Section 5.2.5 for more information.
- *InterfaceSpecs*: This parameter specifies the wires that should be used for the interface. Furthermore, the parameter indicates the names of the signals that must be bound to these wires. The parameter takes a list of values. Each element of this list is separated into three components by the colon sign: I/O direction, lengths of the wires, and the names of the signals. The I/O direction in combination with the border indicates the cardinal direction of the signal, where the direction is from the perspective of the slot. For example, if we specify an interface on the west border of the slot and define it as an input signal, then the signal direction is towards the east.

As mentioned before, the command *PrintInterfaceConstraintsForSelection* generates the interface constraints only for the current selection of tiles. Therefore, we have to use this command multiple times. More specifically, we have to apply this command to all locations where we would like to have an interface between the static area and partial area. For the proposed fine-grained DPR system in Chapter 4, we have to apply this command for each side of the slots that are adjacent to the static area, as illustrated in Figure 4.7. Therefore, we have to apply this command $X * 2 + Y * 2$ times.

```
1  # select interface border
2  ClearSelection;
3  AddBlockToSelection
4    UpperLeftTile=INT_L_X38Y15
5    LowerRightTile=INT_L_X38Y14;
6  StoreCurrentSelectionAs UserSelectionType=X0Y0_WestBorder;
7
8  # generate interface constraints
9  PrintInterfaceConstraintsForSelection
10   FileName=./static_interface_constraints.tcl
11   Append=False
12   CreateBackupFile=False
13   SignalPrefix=x0y0
14   InstanceName=inst_PartialArea
15   Border=West
16   NumberOfSignals=16
17   PreventWiresFromBlocking=True
18   InterfaceSpecs=In:2-4:s2p_w,Out:2-4:p2s_w;
```

Listing 5.6: The command *PrintInterfaceConstraintsForSelection* is used to generate a TCL script that binds the interface signals to the interface wires between the static area and partial area.

The command generates a TCL script with constraints for the interface. We use the property *HD.PARTPIN_LOCS* from Vivado to bind a specific signal to a particular port on a INT tile [31]. The router is forced to route the specified signal through the specific port. The syntax of this property is illustrated in Listing 5.7. The locations of the INT tiles are obtained from the current selection in GoAhead, and the port names are constructed by the specifications in the parameter *InterfaceSpecs* (see also Section 2.1.3). For each interface signal, such property is generated.

```
1  # bind the signal to a particular port
2  set_property HD.PARTPIN_LOCS INT_L_X38Y15/EE2END0
3               [get_pins inst_PartialArea/x0y0_s2p_w[0]];
```

Listing 5.7: The property *HD.PARTPIN_LOCS* is used to define a particular signal to be routed through a specified INT tile and port.

In Chapter 3, we described that binding a specific signal x that has to be routed using wire y is not possible. In the original tool flow of GoAhead, a workaround is used for binding the signals to particular wires, which is explained in Section 5.2.5.

However, the *HD.PARTPIN_LOCS* property is a recent property added in Vivado. By using this property, we can actually bind a signal to a wire.

The Command *AddLUTConnectionPrimitive*

By using the command *PrintInterfaceConstrainsForSelection*, we specify a particular signal to route through a specific port on the INT tile. However, in VHDL, we cannot leave signals unconnected. Otherwise, they will be optimized away. Therefore, we require an element to anchor the signal. In GoAhead, there is a library with connection primitives that can be used to connect signals. These connection primitives are BELs, such as LUTs and FFs. In this work, we will use LUTs to connect a single input and output signal.

As mentioned before, when loading an FPGA device, the corresponding family script is also executed. In this script, we run family-related commands. One of these commands adds a connection primitive to the library. However, GoAhead currently does not support a simple connection primitive of a LUT that connects a single input and output. Therefore, we developed a new command that adds such a connection primitive to the library. The name of this command is *AddLUTConnectionPrimitive*. We call the connection primitive *LUTConnectionPrimitive*.

This command is included in the family scripts. Since the corresponding family script is executed when a particular device is loaded in GoAhead, users of GoAhead do not have to use this command but can use the created connection primitive from the library, as we will see in the following.

The Command *InstantiateConnectionPrimitives*

As the connection primitive *LUTConnectionPrimitive* is now part of the library, we can instantiate connection primitives of this type. We have to instantiate one LUT for each input and output signal. In Listing 5.8, the GoAhead commands are shown to instantiate LUTs and annotate signals to them. The name of the command to instantiate LUTs is *InstantiateConnectionPrimitives*. We instantiate connection primitives of the type *LUTConnectionPrimitive*. We also specify the number of connection primitives (LUTs). For example, if we have an interface with 16 input signals and 16 output signals, we have to instantiate 16 LUTs. Furthermore, we give the instantiated connection primitives a unique name by using the parameter *InstanceName*.

The Command *AnnotateSignalNamesToConnectionPrimitives*

Once we have instantiated the LUTs, we use the command *AnnotateSignalNamesToConnectionPrimitives* to annotate signals to them, as illustrated in Listing 5.8. In the following, we describe the parameters of this command.

- *InstantiationFilter*: This parameter specifies which connection primitive instantiations should be annotated. This specification is a regular expression. In the command *InstantiateConnectionPrimitives*, we gave a unique name to a collection of connection primitives. Now, if we would like to annotate only these instantiations, we have to filter on that particular instance name.
- *InputMappingKind*: This parameter specifies whether the input of the connection primitive (LUT) is used to connect a signal. In case we choose to leave the input unconnected, we set this parameter to *internal*. Otherwise, we set it to *external*. This terminology is inherited from the existing GoAhead commands.
- *OutputMappingKind*: This parameter specifies whether the output of the connection primitive (LUT) is used to connect a signal. The way we use this parameter is similar to parameter *InputMappingKind*.
- *SignalPrefix*: This parameter specifies the prefix of the signal names from the signals that are annotated to the connection primitives.
- *InputSignalNames*: This parameter specifies the name of the input signals that are annotated to the connection primitives.
- *OutputSignalNames*: This parameter specifies the name of the output signals that are annotated to the connection primitives.
- *LookupTableInputPort*: This parameter specifies the particular input port of the LUT that is used to connect the signal. We will see in Section 5.2.5 why this is relevant.

The input and output signals that are specified in the command *AnnotateSignalNamesToConnectionPrimitives* must match with the interface signals from the command *PrintInterfaceConstraintsForSelection*. In the latter command, we generated constraints that particular interface signals must be connected to specific wires. Then, we instantiated connection primitives with the command *InstantiateConnectionPrimitives* that are used to connect these interface signals. Moreover, we used the command *AnnotateSignalNamesToConnectionPrimitives* to specify which interface signals should be connected to which connection primitives. In the following, we generate a VHDL template that contains all these connection primitives.

```
1  # instantiate LUT connection primitives
2  InstantiateConnectionPrimitives
3    LibraryElementName=LUTConnectionPrimitive
4    InstanceName=inst_x0y0_w
5    NumberOfPrimitives=16;
6
7  # annotate signal names to the LUT connection primitives
8  AnnotateSignalNamesToConnectionPrimitives
9    InstantiationFilter=inst_x0y0_w.*
10   InputMappingKind=external
11   OutputMappingKind=external
12   SignalPrefix=x0y0
13   InputSignalName=s2p_w
14   OutputSignalName=p2s_w
15   LookupTableInputPort=3;
```

Listing 5.8: The commands *InstantiateConnectionPrimitives* and *AnnotateSignalNamesToConnectionPrimitives* are used to instantiate LUTs, which are placed within the partial area to connect the interface signals between the partial and static area.

5.2.3 VHDL Templates

In the static system, we consider the partial area as a black-box component. In the previous section, we specified the interface signals of the partial area and connected them to the connection primitives. Now, in GoAhead, we generate a VHDL template of the partial area. This template contains all the instantiated connection primitives. Therefore, in the static system, the partial area is nothing more than some dummy connection primitives. The VHDL template should be merged into the existing design files. The command that we use in GoAhead to generate the VHDL template is *PrintVHDLWrapper*. In Listing 5.9, the syntax of this command is shown.

The generalized component declaration of the partial area in VHDL is illustrated in Listing 5.10. The component contains all the interface signals that we have defined with the commands in Listing 5.8. For convenience, the prefix of the signal names indicates to which slot location they belong (see Figure 4.7). Furthermore, the postfix of the signal names specifies the side on the slot.

The VHDL template includes all the connection primitive (LUT) instantiations. In Listing 5.11, a single instantiation is shown. The input signal is connected to input pin 3 of the LUT, as specified in Listing 5.8.

```
# print partial area module in VHDL
PrintVHDLWrapper
  InstantiationFilter=.*
  EntityName=PartialArea
  FileName=/PartialArea.vhd
  Append=False
  CreateBackupFile=True;
```

Listing 5.9: The command *PrintVHDLWrapper* is used to generate a VHDL template of the instantiated connection primitives that match the filter.

```
component PartialArea is port (
    x0y0_p2s_b : out std_logic_vector(n downto 0);
    x0y0_s2p_b : in std_logic_vector(m downto 0);
    x0y0_p2s_b : out std_logic_vector(n downto 0);
    x0y0_s2p_b : in std_logic_vector(m downto 0);
    -- ..
    xXyY_p2s_b : out std_logic_vector(n downto 0);
    xXyY_s2p_b : in std_logic_vector(m downto 0));
    xXyY_p2s_b : out std_logic_vector(n downto 0);
    xXyY_s2p_b : in std_logic_vector(m downto 0));
end component PartialArea;
```

Listing 5.10: The component declaration of the partial area.

```
inst_x0y0_n_0 : LUT6
generic map ( INIT => X"ABCDABCDABCDABCD" )
port map (
        O => x0y0_p2s_n(0),
        I0 => '0',
        I1 => '0',
        I2 => '0',
        I3 => x0y0_s2p_n(0),
        I4 => '0',
        I5 => '0'
);
```

Listing 5.11: A connection primitive instantiation in the VHDL template of the partial area.

5.2.4 Placement Constraints

The connection primitives are used to anchor the interface signals. However, after placement and routing, they are no longer required, because the interface signals are bound to the proper wires at this point. We constrain the placement of connection primitives within the partial area. The reason is that we reconfigure the partial area with modules during run-time, and therefore we have no logic overhead due to these connection primitives. This concept is demonstrated in Figure 5.3. As illustrated in Figure 5.3a, the connection primitives (LUTs) within the partial area are used to anchor the interface signals. Then, when we configure a module, the connection primitives are reconfigured by the logic of the module, as illustrated in Figure 5.3c. Therefore, the connection primitives do not cause any logic overhead to the system.

In Vivado, we constrain the placement of the connection primitives by using *pblocks* [33]. A pblock is a user-chosen area on the FPGA fabric. Logic cells (e.g., FFs and LUTs) can be assigned to the pblock. By default, the placement of the logic cells belonging to the pblock is within its region.

By using GoAhead, we create a pblock on the same location as the partial area and assign the connection primitives to the pblock. By doing this, we constrain the placer to place all the connection primitives within the partial area. Another placement constraint is that logic belonging to the static part of the system should be placed outside the partial area. For this purpose, we use the property *EXCLUDE_PLACEMENT* on the same pblock that we used to constrain the placement of the connection primitives [32]. This property forces the placement of the logic cells that do not belong to the pblock outside the pblock region. Therefore, logic cells that belong to the static part of the system will not be placed within the partial area.

The commands *PrintAreaConstraint* and *PrintExcludePlacementProperty*

The commands that we use in GoAhead to generate the placement constraints are *PrintAreaConstraint* and *PrintExcludePlacementProperty*. The syntax of these commands is illustrated in Listing 5.12. The command *PrintAreaConstraint* creates a TCL script that defines a pblock in Vivado. The size and location of this pblock depend on the current selection of tiles in GoAhead. Therefore, we have to select the tiles of the partial area before we apply this command, as illustrated in Listing 5.12.

The command *PrintExcludePlacementProperty* generates a TCL script that applies the *EXCLUDE_PLACEMENT* property on the pblock. Note that *Append* is set to *False* in the command *PrintAreaConstraint* and set to *True* in the command *PrintExcludePlacementProperty*. By doing this, we can add content to the same file.

Consequently, we avoid the generation of many separate files.

```
1  # select the partial area
2  ClearSelection;
3  SelectUserSelection UserSelectionType=PartialArea;
4
5  # place logic cells of the partial area within the partial area
6  PrintAreaConstraint
7    InstanceName=inst_PartialArea
8    FileName=./static_placement_constraints.tcl
9    Append=False
10   CreateBackupFile=True;
11
12 # place all other logic cells outside the partial area
13 PrintExcludePlacementProperty
14   InstanceName=inst_PartialArea
15   FileName=./static_placement_constraints.tcl
16   Append=True
17   CreateBackupFile=True;
```

Listing 5.12: The commands *PrintAreaConstraint* and *PrintExcludePlacement-Property* are used to (1) constrain the logic cells of the partial area being placed within the partial area and (2) exclude placement of logic cells from the static part of the system within the partial area.

The TCL Commands

The GoAhead commands in Listing 5.12 generate a TCL script that constrains the placement of the logic cells. This TCL script is illustrated in Figure 5.13. First, the pblock is defined by the command *create_pblock*. Then, we use the command *resize_pblock* to place the pblock onto the fabric of the FPGA. In this command, we also specify the location and the size of the pblock.

In the following, we add the logic cells to the pblock by using the command *add_cells_to_pblock*. We use the command *get_cells* to obtain all the logic cells from a component instantiation in VHDL. As described in Section 5.2.3, the partial area component contains all the connection primitives. Therefore, we add the logic cells of the partial area to the pblock. Consequently, we assigned all the connection primitives to the pblock. Finally, we apply the property *EXCLUDE_PLACEMENT* on this pblock.

Note that the instantiation name of the partial area component is specified by the parameter *InstanceName* in the GoAhead command *PrintAreaConstraint*. Also, the name of the pblock is based on this parameter. Therefore, the parameter *InstanceName* must be the same in both commands *PrintAreaConstraint* and *PrintExcludePlacementProperty*. By doing this, we apply the generated commands and properties in the TCL script on the same pblock.

```
1  create_pblock pb_inst_PartialArea;
2  # define location and size of the pblock
3  resize_pblock [get_pblocks pb_inst_PartialArea]
4                -add {SLICE_X36Y0:SLICE_X47Y99};
5  # add logic cells to the pblock
6  add_cells_to_pblock [get_pblocks pb_inst_PartialArea]
7                      [get_cells inst_PartialArea];
8  # prevent placement of logic that is not assigned to the pblock
9  set_property EXCLUDE_PLACEMENT true
10              [get_pblocks pb_inst_PartialArea];
```

Listing 5.13: In Vivado, we use pblocks to constrain the placement of logic cells.

5.2.5 The Blocker Macro

We use a blocker macro within the partial area to support module relocation. The purpose of this blocker is to prevent the signals belonging to the static part of the system route through the partial area. As illustrated in Figure 2.14, the static signals that route through the partial area will be cut when the partial area is being reconfigured.

The blocker occupies wires within the partial area, such that the static signals cannot enter the partial area. Therefore, the static signals are forced to route around the partial area. This method is similar to the original design flow of GoAhead [1]. However, they occupy all the wires in the partial area by the blocker, except the *tunnel wires*. Tunnel wires are unblocked wires in the blocker that forms a path to the connection primitives. By doing this, they force the interface signals to route through particular wires (binding). They leave one particular routing path (tunnel) available for each signal to reach the corresponding connection primitive. However, we use property *HD.PARTPIN_LOCS* to constrain the router to route a signal through a specific wire. Therefore, we only have to block the wires that cross the border between the static area and partial area, except the interface wires. This concept is illustrated in Figure 5.6. The reason that this method does not work in the original design flow

of GoAhead is that the interface signals have then multiple routing options to the connection primitives. Therefore, the binding might fail. Consequently, there might be no proper communication.

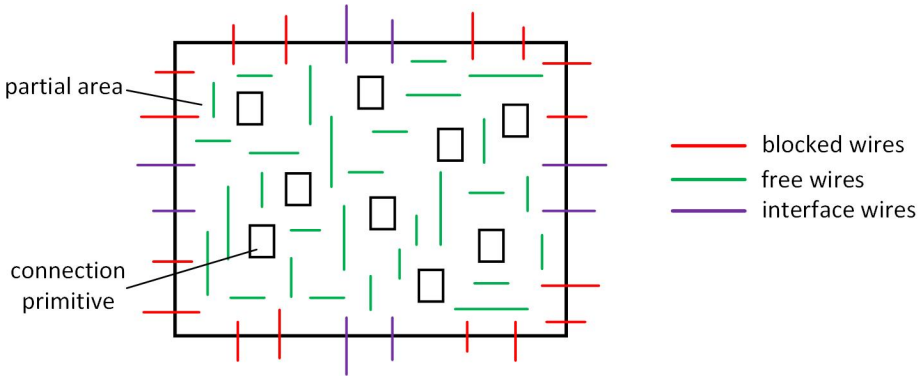

Figure 5.6: The wires that cross the boundary between the partial and static areas are blocked, except the interface wires. The interface signals can route via the interface wires and free wires towards the connection primitives.

The TCL Commands

During the implementation of the static system in Vivado, the blocker is a signal that is being routed via the wires that must be blocked. By doing this, the static signals cannot use these wires, since they are occupied by the blocker. Consequently, the static signals have to route via another (unblocked) wires.

The blocker is generated by using the *ROUTE* property [16]. This property is also called the *directed routing string*. The directed routing string represents the tree structure of a physical route by using nested brackets to represent branching. As we have seen in Section 2.1.3, the INT tiles contain PIPs that can be enabled to route a signal from one location to one or multiple other locations. By using the property *ROUTE*, we can specify the physical route of a signal by defining the PIPs connections.

Figure 5.7 illustrates the tree structure of the blocker signal. We use only a nesting depth of two. Note that we start with a virtual node. This means that there is no defined start location of the blocker signal. The rest of the tree comprises individual PIP connections and, therefore, wires that are being blocked. The corresponding routing string is the following.

$$"\{begin\_port_0 \;\; end\_port_0\}\{begin\_port_1 \;\; end\_port_1\}\ldots\{begin\_port_n \;\; end\_port_n\}"$$

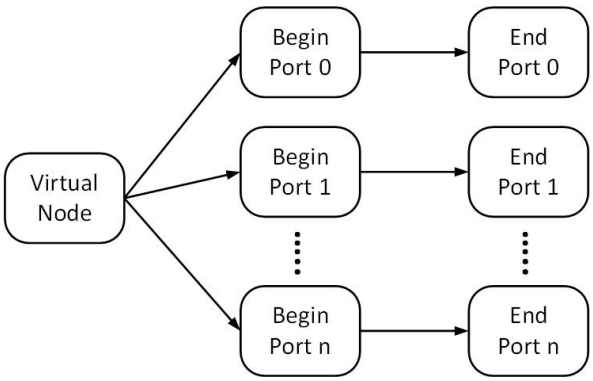

Figure 5.7: The tree structure of the routing net for the blocker macro.

A PIP connection connects a begin port to an end port. Therefore, for each PIP connection that we use in the directed routing string, we block two wires. This is illustrated in Figure 5.8. In this figure, a single INT tile is shown that has several blocked wires and several free wires, depending on the enabled PIPs. Note that the ports that are connected to the logic tile are also used for the blocker macro. In principle, we only have to block the wires that connect two INT tiles, because the purpose of the blocker is to prevent routing within the partial area. However, we require an end port and begin port to block wires. We might run out of these ports, and therefore, we use the ports that are connected to the logic tile such that all the desired wires can be blocked.

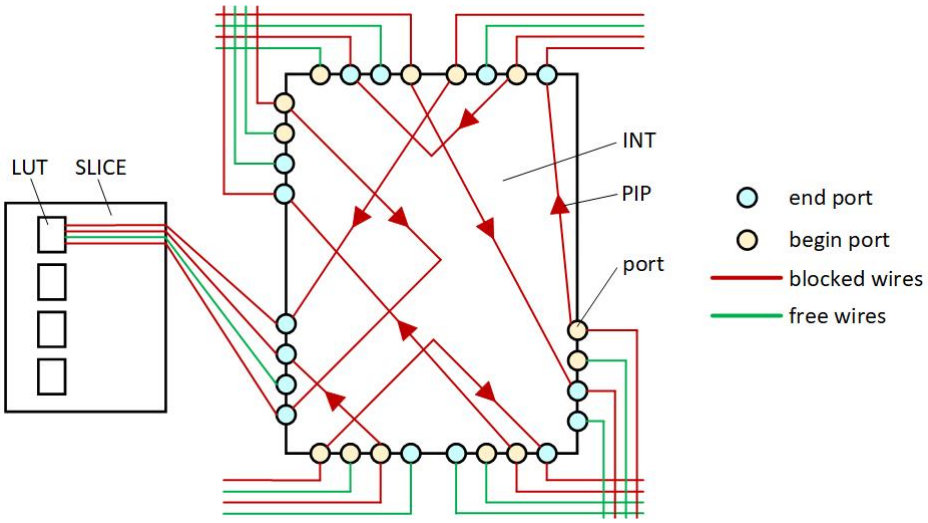

Figure 5.8: The blocker is a directed routing string that enables PIP connections, which are used for the blocker signal. A PIP connects a begin port and end port, and therefore, two wires are being blocked.

```
1 set_property ROUTE "( \
2         { INT_L_X26Y99/LVB_L12 INT_L_X26Y99/WW4BEG2 } \
3         { INT_L_X26Y99/LVB_L0 INT_L_X26Y99/SW6BEG2 } \
4         ..
5         { INT_R_X31Y0/LOGIC_OUTS7 INT_R_X31Y0/EE2BEG3 } \
6         { INT_R_X31Y0/LOGIC_OUTS7 INT_R_X31Y0/EL1BEG2 } \
7 )" [get_nets blocker_net_BlockSelection]
```

Listing 5.14: The property *ROUTE* is used to generate the blocker.

The Commands *BlockSelection* and *BlockWiresInSelection*

The command *BlockSelection* is an existing command in GoAhead that blocks the wires that are connected to an INT tile. By default, the command blocks all the wires of the INT tiles in the current selection. However, in GoAhead, we can prevent a particular port from blocking. As illustrated in Figure 5.6, the wires that do not cross the border from the partial area to the static area should be prevented from blocking in the partial area. The command in GoAhead that prevents these wires from blocking is *BlockWiresInSelection*. Note that these commands are applied to the current selection of tiles. Therefore, before using these commands, we have to select the tiles belonging to the partial area, as illustrated in Listing 5.15.

The Commands *BlockLUTInputPortsInSelection* and *SaveAsBlocker*

The command *BlockSelection* does not only block wires between INT tiles but blocks also the wires that are connected to the logic tiles. As mentioned before, we connect a single input signal to the connection primitives. We use one of its input pins to connect the input signal. Therefore, we have to prevent the path (wires) towards this input pin from blocking. Otherwise, the particular input pin on the LUT cannot be reached by the signal since the wire is occupied by the blocker. Note that there is one unique path from the INT tile towards each input pin of the LUT.

We use the command *BlockLUTInputPortsInSelection* to prevent the path towards the input port of the LUT from blocking. The parameter *InputPortsRegex* specifies the port name on the INT tile that directs to the input pin of the LUT. This port should be prevented from blocking. Note that there are different types of CLBs on the FPGA fabric. Also, the LUTs belonging to these CLBs have different naming. Therefore, the port names that direct to all these particular input pins of the LUTs are also slightly different. Now, to prevent the particular input port of all the LUTs from blocking, a regular expression is used to specify all these different port names.

The last group of wires that must be prevented from blocking are the interface wires. The unblocking of the interface wires is part of the *PrintInterfaceConstraintsForSelection* command, as illustrated in Listing 5.6. By setting the parameter *PreventWiresFromBlocking* to *True*, we prevent these specific wires from blocking.

Now that we defined which wires should be blocked and which wires should be prevented from blocking in the partial area, we use the GoAhead command *SaveAsBlocker* to generate a TCL script that includes the blocker.

```
1  # select partial area
2  ClearSelection;
3  SelectUserSelection UserSelectionType=PartialArea;
4
5  # prevent wires within the partial area from blocking
6  BlockWiresInSelection;
7  BlockLUTInputPortsInSelection
8    InputPortsRegex=.*(L|M)*_(A|B|C|D)4;
9
10 # block all other wires
11 BlockSelection
12   NetlistContainerName=default_netlist_container;
13
14 # generate directed routing string
15 SaveAsBlocker
16   NetlistContainerNames=default_netlist_container
17   FileName=./static_blocker.tcl;
```

Listing 5.15: The commands *BlockWiresInSelection* and *BlockLUTInputPortsInSelection* are used to prevent blocking ports within the partial area. The blocker net is generated by using the commands *BlockSelection* and *SaveAsBlocker*.

5.2.6 Implementation in Vivado

Once we designed the DPR system in GoAhead, we have a total of five files with constraints and VHDL templates. As a summary, GoAhead generated the following files.

- **static_clock_connections.tcl**. This file contains TCL commands to connect all the clock pins of BELs within the partial area.

- **static_placement_constraints.tcl**. This file has restrictions for the floorplanning of the partial area and makes sure that no logic from the static part of the system is placed within the partial area.
- **static_interface_constraints.tcl**. This file contains constraints that binds the interface signals between the partial area and the static area to specific wires.
- **static_blocker.tcl**. This file contains the directed routing string to implement the blocker.
- **PartialArea.vhd**. This file provides the LUT instantiations in VHDL. The interface signals are connected to these LUTs.

The TCL scripts should be copied in the Vivado project, and the VHDL template should be merged into the existing design files. We provide a TCL script (named *Run_Static.tcl*) to automize the whole design flow in Vivado. After each step, we create a checkpoint to verify the design in each step. The steps in the TCL script are the following.

1. Synthesize the design. The TCL command *synth_design* is used [33].
2. Set placement constraints. The script *static_placement_constraints.tcl* is executed.
3. Set interface constraints. The script *static_interface_constraints.tcl* is executed.
4. Optimize the design. The TCL command *opt_design* is used.
5. Place the design. The TCL command *place_design* is used.
6. Connect the clock pins in the partial area. The script *static_connect_clockpins.tcl* is executed.
7. Insert the blocker. The script *static_blocker.tcl* is executed.
8. Route the design. The TCL command *route_design* is used.
9. Remove the blocker. The TCL command *route_design* is used. We use the parameter *unroute* to unroute the blocker net.
10. Generate the bitstream. The TCL command *write_bitstream* is used. The generated bitstream can be programmed on the FPGA by using one of the configuration interfaces.

5.3 Implementing the Modules

The modules are developed independently from the static system. The design flow of the modules is very similar to the design flow of the static system, as illustrated in Figure 5.1. In both design flows, we generate constraint files and VHDL templates by using GoAhead. The VHDL templates are merged within the existing design files, and the constraint files are included in the Vivado project. Vivado performs the low-level operations (e.g., synthesis, placement, routing, etc.), and the final result is a bitstream. We use an external tool called BitMan to generate a partial bitstream from the bitstream generated by Vivado.

Although the design flow of the modules is similar compared to the design flow of the static system, the modules require a different floorplan in comparison to the static system. However, we can use the GoAhead commands that are introduced in the previous section to implement the modules, as well. Therefore, we discuss the implementation of the modules in less detail.

5.3.1 Implementation in GoAhead

The implementation of the module starts by floorplanning the module area in GoAhead. For this purpose, we can use the GUI or the usual selection commands, as illustrated in Listing 5.3.

In the design phase, the designer should have determined the interface, shape, size, and footprint of the modules. In principle, there is no constraint for the location to develop the module on the FPGA fabric, as long the interface, shape, size, and footprint matches with the module characteristics from the design phase. The reason is that we can relocate the module to any position with a similar footprint later on.

Interface Constraints

The interfaces of the modules should be located at the same relative position as in the static system. By doing this, the static system and modules can communicate, since their interface signals are bind to the same wires. In Figure 5.5, two interface locations in the static system are illustrated. In this particular example of the static system, we partitioned the partial area into 2×2 imaginary slots. In Figure 5.9, a module floorplan in GoAhead for this partial area is shown. The size of the module is 1×2 slots. As illustrated in this figure, the input interface is at the same relative position as in the static system. As a result, the static system and module can properly communicate when the module is configured within the partial area.

However, note that if we configure the module in the partial area, the output of the module is not connected to the interface signals from the partial area to the static area. Therefore, in this particular example, there is only a proper communication from the static area to the partial area, but not vice versa. We have to configure additional (by-pass) modules to make an appropriate connection from the partial area to the static area, as well.

The way we implement the interfaces for the modules with GoAhead is the same as we described in Section 5.2.2. We can use the GoAhead commands that are illustrated in Listing 5.6.

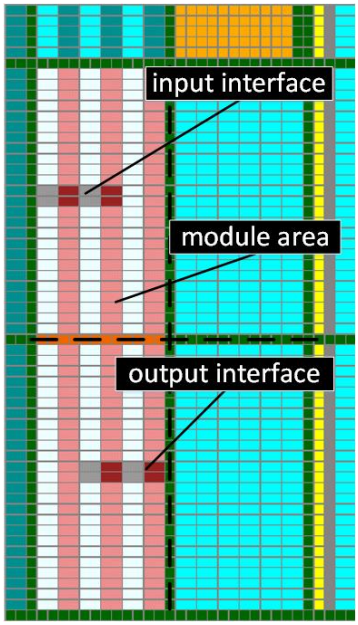

Figure 5.9: The floorplan of the module area in GoAhead. The interface selections should be at the same relative position as in the static system.

Placement Constraints

During the development of the modules, only the area of the module is essential. Later on, we extract the module area out of the total FPGA fabric and generate a partial bitstream from this area by using BitMan. However, in the module design, we have to connect the interface signals with the logic primitives outside the module area. The reason is that we cannot leave these signals unconnected. Therefore, we use connection primitives. This concept is illustrated in Figure 5.10. The connection primitives are used to anchor the interface signals. We can instantiate and annotate signals to the connection primitives with the GoAhead commands in Listing 5.8.

The connection primitives should be placed outside the blocker area. Otherwise, they are not reachable since almost all the wires in the blocker area are prohibited

by the blocker. Therefore, in Vivado, we use pblocks. These pblocks are located outside the module and blocker areas. We use the *PrintAreaConstraint* command in GoAhead for every connection primitive block. The syntax of this command is illustrated in Listing 5.12.

The logic cells that belong to the module should all be placed within the module area. Therefore, in Vivado, we use a pblock to constrain the placement within the module area. Again, we have to use the GoAhead commands in Listing 5.12 to generate the TCL script that defines the pblock.

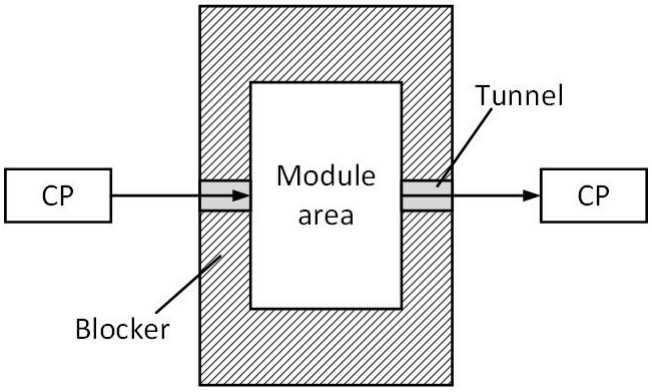

Figure 5.10: In the implementation of the modules, only the module area is essential. However, we have to connect the interface signals outside the module area. We use connection primitives to connect the interface signals. A blocker is used to prevent routing outside the module area.

Blocker

Another constraint when developing a module is that the signals of the module are routed exclusively in the module area, except the interface signals. In Vivado, there is a property named *CONTAIN_ROUTING* [32]. This property can be used on a pblock to use strictly routing resources within the area defined by the pblock. However, only signals that are entirely owned by the pblock cells will be contained within the pblock. For example, the interface signals of the modules require to be routed outside the module area and thus cannot be restricted to be in the module area. Therefore, we use a blocker macro around the module area. This blocker macro occupies all the wires around the module area. As a result, the signals belonging to the module can only route within the module area. The size of the blocker macro should be such that no signal can enter or leave the module area, besides the interface signals. Therefore, the minimal width or height at the boundaries of the module area is the maximal wire length in the respectively horizontal or vertical direction. We can specify the blocker area in GoAhead by selecting the tiles with the GUI or the usual selection commands, as illustrated in Listing 5.3.

Tunnels

As mentioned before, the only signals that leave the module area are the interface signals. We create tunnels in the blocker area such that the interface signals can route through the blocker towards the connection primitives. These tunnels are illustrated in Figure 5.10.

In GoAhead, we use the command *ExcludePortsFromBlockingInSelection* for creating tunnels. This command excludes a particular port from blocking on all INT tiles in the current selection. The specific port to be blocked can be specified in the parameter *PortName*. Before we explain which ports we use for creating the tunnels, we have to note that each corresponding end and begin port can be directly connected through a PIP. By the corresponding end and begin ports, we mean the ports on the same INT tile with the same direction and length (see also Section 2.1.3). As mentioned before, we bind the interface signals on wires in either the horizontal or vertical direction. We use the same wires on the INT tiles in the blocker area to form a horizontal or vertical path through the blocker area.

In Figure 5.11, an example of a tunnel in the eastern direction is illustrated. This tunnel uses the wires with length two in the eastern direction. As we can see, the corresponding begin and end ports are connected through a PIP. Note that the wires with the same properties are bundled in groups of four. Each of these ports has a slightly different name. Namely, the index of these ports is different. Consequently, we have to use the command *ExcludePortsFromBlockingInSelection* reasonably often if we have many interface signals since we also use wires with varying properties (e.g., wire lengths and directions) for the interface wires. Therefore, we use the *AddAlias* command to combine groups of ports (see Section 5.1).

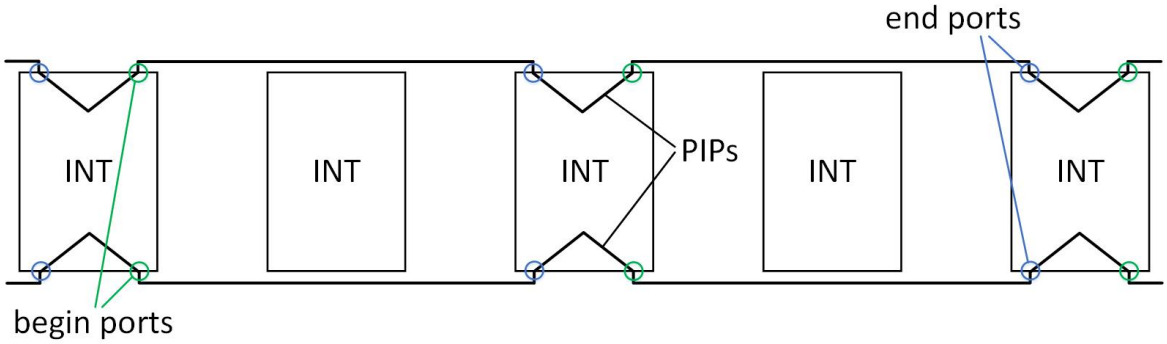

Figure 5.11: The corresponding end ports and begin ports on the same INT tile can be connected through a PIP. In this example, the EE2BEG* and EE2END* ports are connected via PIPs. By using these ports, we can construct a tunnel in the eastern direction.

For convenience, we group the ports with the same length, cardinal direction, and sort them as begin ports or end ports. The prefix of the command names that creates a tunnel is *DoNotBlock*. In the following, the length and cardinal direction are specified. Finally, the command name ends whether the group of ports are begin ports or end ports. As an example, we use the commands *DoNotBlockDoubleEastBegin* and *DoNotBlockDoubleEastEnd* to create a tunnel such as illustrated in Figure 5.11. In Listing 5.16, the GoAhead code to create the *DoNotBlockDoubleEastBegin* command is shown.

```
1  # prevent begin ports with a wire length of two from blocking
2  AddAlias
3    AliasName=DoNotBlockDoubleEastBegin
4
5    Commands="ExcludePortsFromBlockingInSelection
6    PortName=EE2BEG0
7    CheckForExistence=False
8    IncludeAllPorts=True;
9
10   ExcludePortsFromBlockingInSelection
11   PortName=EE2BEG1
12   CheckForExistence=False
13   IncludeAllPorts=True;
14
15   ExcludePortsFromBlockingInSelection
16   PortName=EE2BEG2
17   CheckForExistence=False
18   IncludeAllPorts=True;
19
20   ExcludePortsFromBlockingInSelection
21   PortName=EE2BEG3
22   CheckForExistence=False
23   IncludeAllPorts=True;";
```

Listing 5.16: The GoAhead command *ExcludePortsFromBlockingInSelection* is used to create tunnels within the blocker area. For convenience, we use the *AddAlias* command to combine ports that belong to a particular tunnel.

We can also merge commands that are created by the *AddAlias* commands. For convenience, we also create commands that combine the begin and end ports. For example, for the tunnel illustrated in Figure 5.11, we combine the command *DoNot-*

BlockDoubleEastBegin and *DoNotBlockDoubleEastEnd* to a new command named *DoNotBlockDoubleEast*. We have done this for all cardinal directions and wiring lengths. Therefore, users of our tool do not have to create the tunnel commands by themselves. The tunnel commands are defined in the family script. Therefore, when loading the device, the tunnel commands can be used by the designer.

Now that we have seen how the tunnels are constructed and which commands we have to use to create tunnels, we have a look at the floorplan of the tunnels in GoAhead. In Figure 5.12, the same module area and interface connections are defined as in Figure 5.9. Around the module area, we defined the blocker area. We use the usual selection commands to define the location of the blocker area. Within the blocker area, we define the tunnels. The same selection commands are used to specify the locations of the tunnels. The location of these tunnels must be at the same height or width compared to the location of the interface. Note that we have to apply the tunnel commands after we selected the location of the tunnels. Furthermore, we defined the locations of the connection primitives outside the module and blocker area.

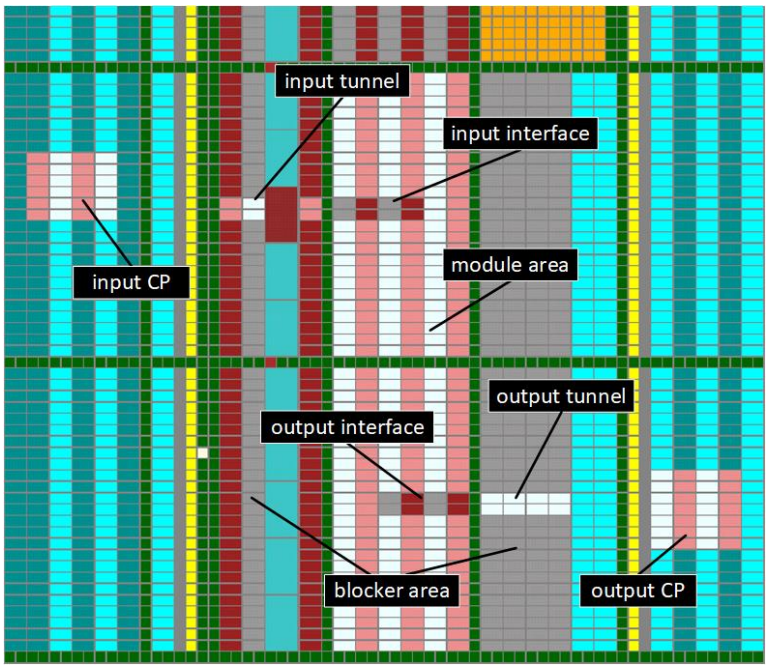

Figure 5.12: The floorplanning of a module in GoAhead. We start by selecting the module area. Around the module area, we define a blocker area to prevent routing outside the module area. However, interface wires have to leave the module area. Therefore, we define tunnels through the blocker area. Finally, we select regions where the connection primitives should be located. These must be located outside the module and blocker area.

After we generated the tunnels in the region of the blocker, the blocker and its corresponding TCL file can be generated by using the commands *BlockSelection* and *SaveAsBlocker*, as illustrated in Listing 5.15.

5.3.2 Implementation in Vivado

Once we designed a module in GoAhead, we have a total of four files with constraints and VHDL templates. As a summary, we generated the following files.

- **module_placement_constraints.tcl**. This file has restrictions for the floorplanning of the module area and makes sure that all the logic is placed within the module area. Also, the file contains constraints to place the connection primitives outside the module area and blocker area.
- **module_interface_constraints.tcl**. This file contains constraints that define the ports for the interface between the module area and the rest of the FPGA fabric.
- **module_blocker.tcl**. This file contains the directed routing string to implement the blocker.
- **ConnectionPrimitives.vhd**. This file provides the LUT instantiations in VHDL. The interface signals are connected to these LUTs.

The VHDL template should be merged into the top-level design, and the TCL scripts should be copied in the Vivado project. Similar as in Section 5.2.6, we provide a script to automize the whole design flow in Vivado. The steps of the module implementation are very similar as in the static system implementation. We run the same steps, besides step 6. An important note is that the output is still a complete bitstream. Therefore, we have to transform it into a partial bitstream.

5.3.3 Creating Partial Bitstreams

The way we generate partial bitstreams is the same as in [11]. They also use BitMan to generate partial bitstreams [17]. BitMan is a tool for creating and manipulating bitstreams. BitMan supports all the recent Xilinx FPGAs.

We can use the tool by opening the command prompt in Windows. In Listing 5.17, the generalized command to generate a partial bitstream in BitMan is shown. The parameters *x0* and *y0* denote the left-down corner of the module area, where *x1* and *y1* indicate the upper-right corner of the module area. The parameters *new_x* and *new_y* denote the left-down corner of the new position. In case we would

like to implement the module on the same location as in the full bitstream, we can use the same coordinates as denoted by *x0* and *y0*. However, if we would like to relocate the module, we have to specify the new location.

Note that by using this tool, we have to generate multiple bitstreams for the same module if we would like to relocate the module. However, the PRC can be used to change the location in the header of the .bit file. This allows us to only have one instantiation for every module in external memory, and still instantiate the module on multiple locations.

```
1 bitman.exe -x x0 y0 x1 y1 full.bit -M new_x new_y partial.bit
```

Listing 5.17: BitMan is used to generate partial bitstreams. The parameters *x0*, *y0*, *x1*, and *y1*, specify the location of the module in the full bitstream. The parameters *new_x* and *new_y* denote the offset position of the module in the partial bitstream.

Chapter 6

Case Study: AES

In this chapter, we demonstrate our framework by using a case study. In the following, we shortly describe the case study. Nowadays, FPGAs are frequently used for hosting cryptographic algorithms (e.g., *Advanced Encryption Standard* (AES) [24]), as they represent an efficient platform. One reason is that FPGAs provide an efficient implementation of such streamed-based algorithms. Another reason is the possibility to reconfigure and update the implementation in the application field after the detection of security flaws, which is very valuable.

The deployment of security-relevant applications is not always in a trusted and controlled environment. It is rather so that the FPGA-based implementation is exposed to an environment that is under full control of an attacker, who can launch *physical attacks* like manipulating the clock frequency, voltage levels, using radiation and laser beams, and so on. These attacks are very powerful, and almost all implementations suffer from these kinds of attacks. Therefore, measurements to protect the implementation for physical attacks are essential and should be carried out at run-time by the FPGA platform itself.

In this case study, we demonstrate a countermeasure against *side-channel attacks* on cryptographic implementations by using DPR. Side-channel attacks are one of these physical attacks and are based on gathering information from the implementation of a system. Timing information, power consumption, or electromagnetic leaks can provide information about the implemented system [20]–[23]. These characteristics can be exploited, and usually, the goal is to get the secret key of the cryptographic algorithm.

The idea is to develop multiple *variants* of the same cryptographic algorithm. All these variants have the same functionality, but a different hardware implementation. By using DPR, we continuously reconfigure these variants. Since each variant has a different hardware layout, the timing, power, and electromagnetic characteristics per variant vary. As a consequence, the characteristics become random, and therefore, side-channel attacks become more difficult or even impossible.

In this case study, we will explore one specific cryptographic algorithm: AES. We will not develop variants of the complete AES algorithm, but we decompose the algorithm in multiple modules. For each module, multiple variants will be generated. For the following, assume that we decompose the AES algorithm in M modules, and we develop for each module V variants. Then, the total number of variants for the complete AES algorithm is V^M. Thus, by doing this, we can compose a lot of different variants of the complete AES algorithm. Another advantage of decomposing the AES algorithm in multiple modules is that we do not have to reconfigure the whole AES algorithm, but only a small portion. Therefore, the downtime of the AES algorithm decreases. As a result, we can achieve higher throughputs.

Now that we introduced the aim of this case study, we discuss it in more detail throughout this chapter. This chapter is organized as follows. In Section 6.1, we have a closer look at the hardware implementation of the AES algorithm on FPGAs. With this knowledge, we decompose the AES algorithm in a static part and several modules. This is discussed in Section 6.2. Once we decomposed the AES algorithm, we can develop the static system and the variants of the modules. The implementation of the static system and modules is respectively described in Section 6.3 and Section 6.4. Finally, after developing the static system and the variants, we demonstrate several configurations and thus variants of the complete AES algorithm. This is discussed in Section 6.5.

6.1 AES Hardware Implementation

In Figure 6.1, the hardware implementation of the AES algorithm on FPGAs is illustrated. In cryptography, the *plaintext* is the unencrypted information, where *ciphertext* is the encrypted result. The AES algorithm exists of four main processing steps: *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*. We shortly introduce the processing steps individually. The *SubBytes* step replaces each byte of the message by the corresponding value from the substitution box, which is often also called S-box. The S-box is a LUT. In the following step, *ShiftRows*, the message is redistributed. The *MixColumns* performs a dot product and matrix multiplication on the data. Finally, the *AddRoundKey* performs an XOR-operation with the message and the key. Furthermore, the *State* is a register and is used to store intermediate results.

In this case study, we use a 128-bit encryption key. Therefore, we have to perform ten rounds of encryption. In a single round, the *State* is processed by all the four main processing steps, except in the last round. In the final round, the *MixColumns* step is by-passed. This flow of data is controlled by a state machine that controls the multiplexers (the rectangles with rounded corners).

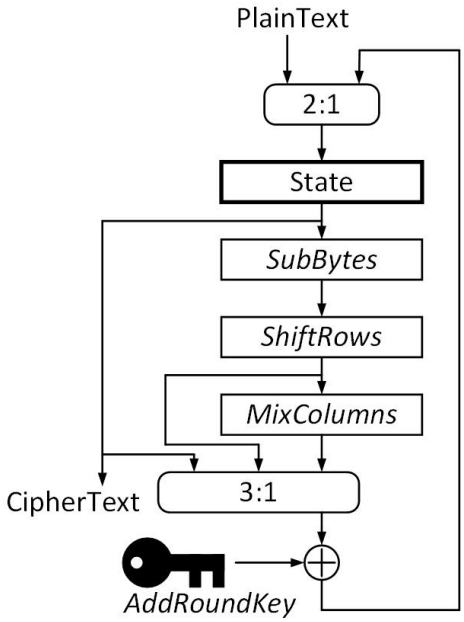

Figure 6.1: The AES implementation on the FPGA. The AES algorithm has four main processing steps: *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*, where *State* is a register. The multiplexers control the flow of data.

6.2 Static/Partial Partitioning

The four main processing steps are excellent candidates to divide the AES algorithm in multiple modules. The *SubBytes*, *ShiftRows*, and *MixColumns* all have the same interface layout. These interfaces contain 128 input bits and 128 output bits. These modules must be placed in consecutive order, and their output is the input of the next module. The *AddRoundKey* has a different interface. This module requires 256 input bits and produces 128 output bits.

In Chapter 4, we discussed that the interface of the slots in the partial area requires a unioned interface of the different inputs and outputs. In the case we use all the four main processing steps, we need 256 bits of input signals and 128 bits output signals. The number of interface signals constrains the minimal size of the slots in the partial area. Therefore, we choose to keep the *AddRoundKey* within the static system. As a result, we only require 128 input signals and thus can build a more fine-grained partial area.

The new hardware layout of the total system is shown in Figure 6.2. In the static system, the *SubBytes*, *ShiftRows*, and *MixColumns* are replaced by the partial area. Also, we add two multiplexers to the system that select the outputs of the *ShiftRows* and *MixColumns* modules (see also Section 4.2.1).

We develop several variants of the *SubBytes*, *ShiftRows*, and *MixColumns* modules that can be configured in the partial area during run-time to make the AES algorithm complete again.

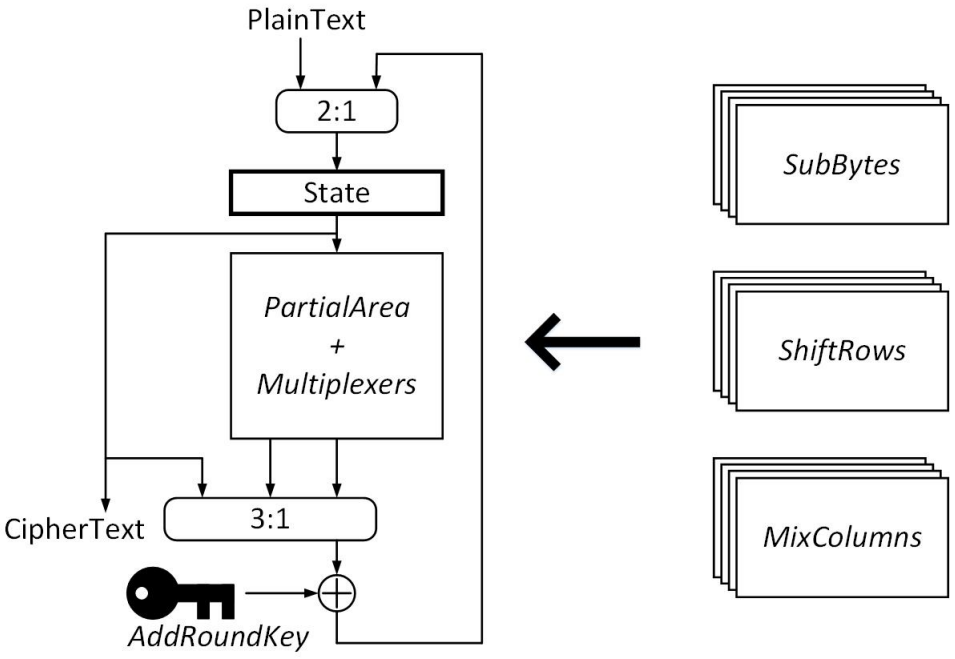

Figure 6.2: The separation of the static system and the reconfigurable parts. We replace the *SubBytes*, *ShiftRows*, and *MixColumns* for the partial area and two multiplexers. Furthermore, we develop variants of the *SubBytes*, *ShiftRows*, and *MixColumns* modules that can be configured within the partial area during run-time, such that the AES flow is complete again.

6.3 Static System

As described in Section 4.2.3, we first perform a planning phase. The sub-steps of the planning phase are static/partial partitioning, define configurations, resource budgeting, floorplanning, and interface specification. In the previous section, we already partitioned the static and partial parts of the system. Namely, we use the *SubBytes*, *ShiftRows*, and *MixColumns* blocks as modules and the rest of the AES algorithm will be located in the static part. Furthermore, we defined the interface specification. Namely, we connect the output of *State* to the *s2p* signals, and we connect two outputs from the partial area *p2s* to the 3:1 multiplexer. In the following, we perform the resource budgeting step.

We extract the resource requirements of the modules by using the synthesis tool

from Vivado. In Table 6.1, the resource consumption of each module in configuration c_1 is shown. The *SubBytes* and *MixColumns* modules require respectively 640 and 128 LUTs. The *ShiftRows* module does not need any logic resources but only routing resources.

In Table 6.2, the resource budgeting of configuration c_2 is illustrated. In this configuration, we use the same *SubBytes* and *ShiftRows* modules compared to configuration c_1. However, the *MixColumns* module in this configuration has a different hardware layout. The *MixColumns* module requires in this variant 2048 LUTs.

Module	#LUT	#DSP	#BRAM
SubBytes	640	0	0
ShiftRows	0	0	0
MixColumns	128	0	0

Table 6.1: Resource budgeting for configuration c_1.

Module	#LUT	#DSP	#BRAM
SubBytes	640	0	0
ShiftRows	0	0	0
MixColumns	2048	0	0

Table 6.2: Resource budgeting for configuration c_2.

We use the formulas that are provided in Section 4.2.3 to determine the minimal size of the partial area (in terms of logic resources). First, we use Equation 4.10 to perform the resource budgeting for each configuration.

$$r_c(c_1) = \{768, 0, 0\} \tag{6.1}$$

$$r_c(c_2) = \{2688, 0, 0\} \tag{6.2}$$

Now, we use Equation 4.12 to calculate the minimal required resources by the partial area.

$$AREA\_RES_{min} = \{2688, 0, 0\} \tag{6.3}$$

Now that we know the minimal size of the partial area, we have to calculate the minimum slot sizes. Since we have three modules, we have to divide the partial area into at least three slots, because each module occupies at least one slot. We use the formulas from Section 5.2.2 to determine the minimal slot sizes.

We use a device from the Zynq family. In the Zynq family, the currently supported wire lengths in the horizontal and vertical direction are respectively $L_h = \{2, 4\}$ and $L_v = \{2, 6\}$. Now, to calculate the maximum number of interfaces wires that we can use per row and column, we use respectively Equation 5.1 and Equation 5.2.

$$max\_wires_{row} = 24 \tag{6.4}$$

$$max\_wires_{column} = 32 \tag{6.5}$$

Previously, we determined that we have 128 interface wires for both input and output. Therefore, the minimal numbers of rows and columns that are required by the slots are calculated by respectively Equation 5.3 and Equation 5.4.

$$nr\_of\_rows_{min} = 6 \tag{6.6}$$

$$nr\_of\_columns_{min} = 4 \tag{6.7}$$

As mentioned in Section 5.2.2, we also have to take into account the wire lengths and the granularity of reconfiguration to determine the minimum width and height of the slots within the partial area. We use Equation 5.7 and Equation 5.8 to calculate the respectively minimum width and height of the slots.

$$slot\_width_{min} = \max(4, 4, 1) = 4 \tag{6.8}$$

$$slot\_height_{min} = \max(6, 6, 50) = 50 \tag{6.9}$$

In Equation 6.3, we calculated that we only require CLB tiles, which contains the LUTs. Each CLB tile contains 8 LUTs. Now, since the logic resources are arranged in columns onto the FPGA fabric, and the minimum height of a slot is 50, each column with CLB tiles has $50 * 8 = 400$ LUTs in a single slot. The minimum width of the slot is four columns. Therefore, the total number of LUTs per slot is $4 * 400 = 1600$. Note that this only holds if we locate the partial area on the part of the FPGA fabric with only CLB tiles.

The modules that we have fit all in one slot, except the *MixColumns* module from configuration c_2. This module requires two slots. Therefore, we need a total of 4 slots.

As already mentioned in Section 4.2.3, maximizing the consistency of the footprints among slots allows us to have high flexibility of module relocation. However, on the device that we use, there is not a region that has eight columns of CLB tiles.

In Figure 6.3, the footprint of our chosen partial area is shown. We have selected this region since this area contains many CLB columns. Note that there are two

types of CLB tiles: *CLBLM* and *CLBLL*. As mentioned in Section 2.1.3, the CLBs include two sites. In the *CLBLM* type, there is one site that provides only logic functionality and one site that offers both logic and memory functionality. The sites of *CLBLL* only provide logic functionality. Furthermore, the footprint of the partial area also contains a BRAM column. The BRAM resources are not used since the modules do not require this type of logic resource.

Each slot in the partial area provides enough LUTs for each module, except the *MixColumns* module from configuration c_2. Therefore, each of these modules can be developed in a single slot. Note that module relocation into the vertical direction is feasible, as the footprints match. However, the footprints of the first column and the second column are different. Therefore, we cannot relocate a module from the first column to the second column and vice versa.

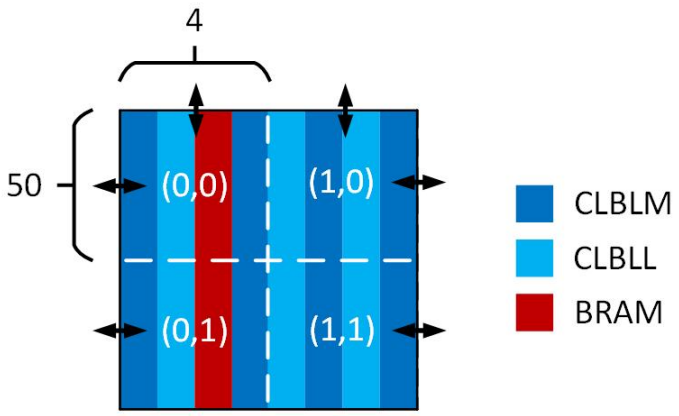

Figure 6.3: In this DPR system, we separate the static area in 2×2 slots. Each slot has a dimension of 4×50 logic tiles. The first column of the slots contains a different footprint in comparison to the second column. Therefore, modules with a similar footprint as in the first column cannot be relocated to the second column, and vice versa.

In Figure 6.4, the floorplanning of the partial area in the static system using GoAhead is illustrated. Note that the footprint of the partial area is the same as that is shown in Figure 6.3. Also, the relative positions of the interfaces are the same among all slots. For example, the interface at the west border of slot $(0, 0)$ locates at the same relative position in the slot as the interface at the west border of slot $(0, 1)$. This makes module relocation feasible. In Figure 6.5, the static implementation in Vivado is illustrated. Note that the locations of the partial area and the interfaces are the same as in Figure 6.4. Also, there is no static routing (green lines) within the partial area. This allows module relocation without breaking the static system, as described in Section 2.2.3.

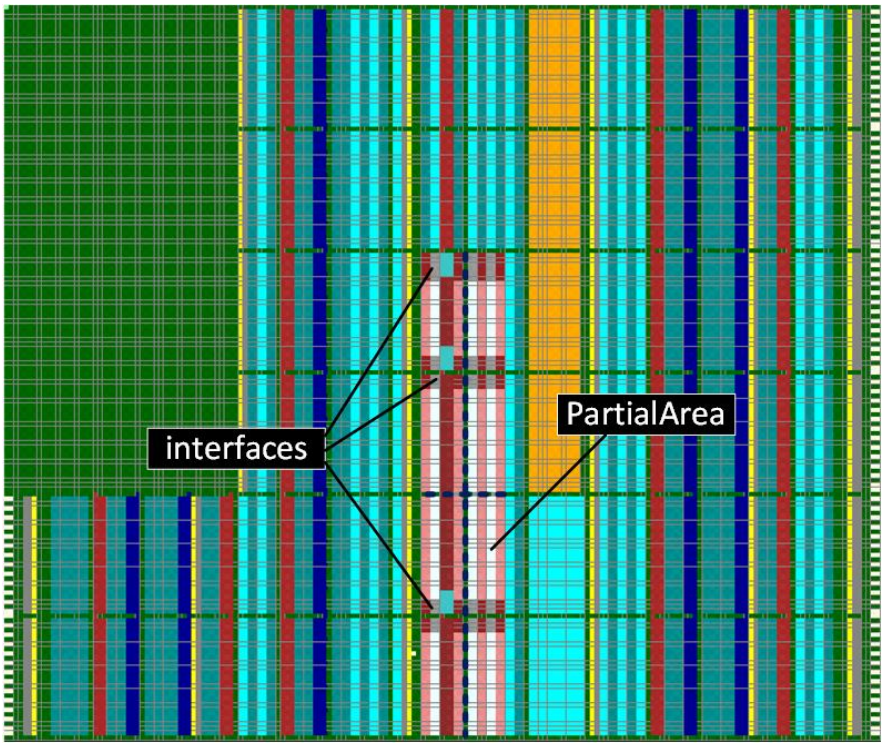

Figure 6.4: The floorplanning of the partial area and the interface locations in GoAhead.

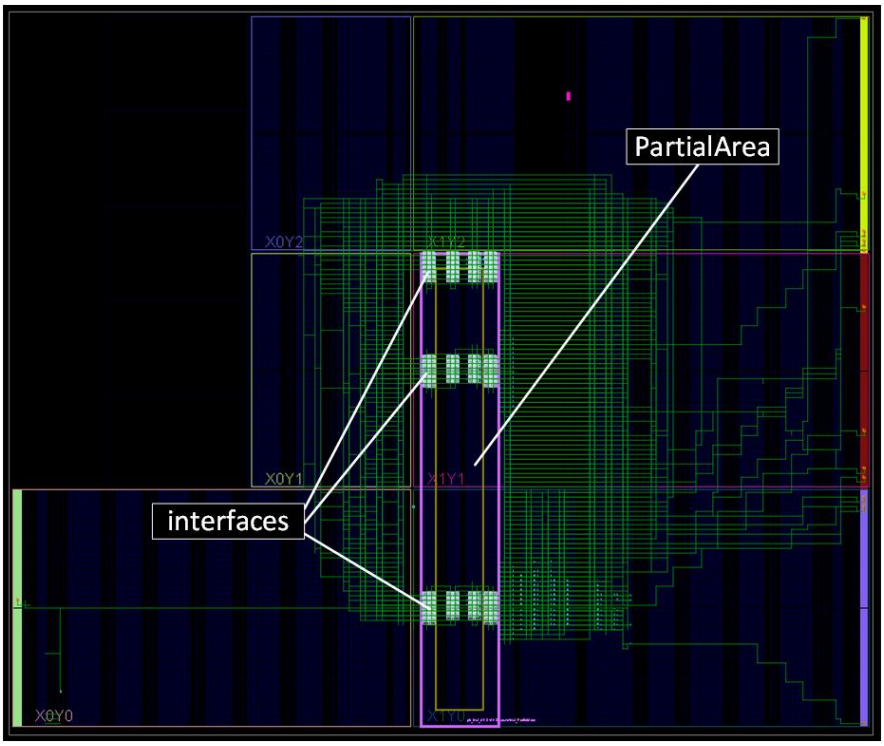

Figure 6.5: The result of the static system after implementation in Vivado.

6.4 Modules

Now that we have developed the static system, we create modules that can be configured within the partial area. In Figure 6.6, the set of modules in configuration c_1 is illustrated. All the modules have the size of one slot. Also, all the modules contain one input interface and three output interfaces. The *MixColumns* module includes the input interface at the southern border, where the *ShiftRows* and *SubBytes* modules have an input interface at the west border. The output interfaces are located on the other sides of the modules.

Furthermore, the *MixColumns* and *ShiftRows* modules have similar footprints, where *SubBytes* has a different footprint in comparison to the other two. Note that the footprint of *MixColumns* and *ShiftRows* modules is equal to the footprint of the second column in the partial area, where the hardware layout of *SubBytes* is equivalent to the footprint of the first column in the partial area. Therefore, *MixColumns* and *ShiftRows* can only be configured within the slots located in the second column, where *SubBytes* can only be configured within the slots located in the first column. Note that the interface positions and the footprints are chosen such that after placement in the partial area, they can properly communicate.

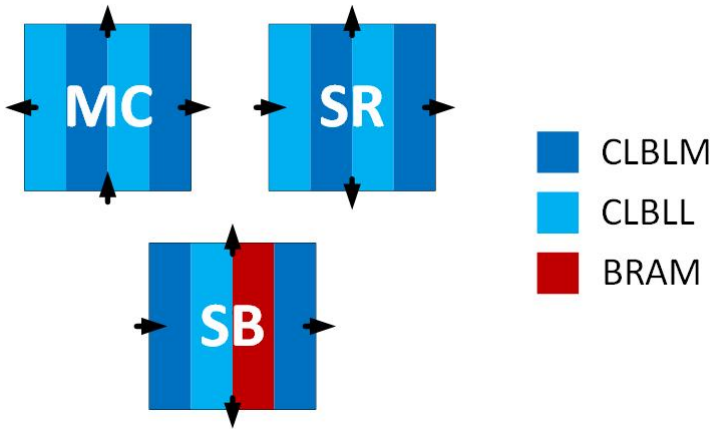

Figure 6.6: A set of modules that can be configured within the partial area. The modules can only be configured within the slots on the partial area where the footprint is equivalent.

The implementation of the *SubBytes* module in GoAhead is illustrated in Figure 6.7. Note that the structure is similar in comparison to the structure in Figure 5.10. Although, for the *SubBytes* module shown in Figure 6.6, we define three output interfaces, and therefore, we floorplanned three connection primitive blocks for the outputs. Similar as in Figure 5.10, we floorplan one connection primitive block for the input. In Figure 6.8, the final implementation result of the *SubBytes* module in Vivado is illustrated. All the signals and logic resources belonging to the module

are routed and placed within the module area, except the interface wires. The interface wires are connected to the connection primitives, which are located outside the module area. We use BitMan to cut out the module area and to generate the corresponding partial bitstream. During run-time, we can configure the partial bitstream in the partial area by using one of the configuration interfaces on the FPGA. The other modules are implemented in the same way.

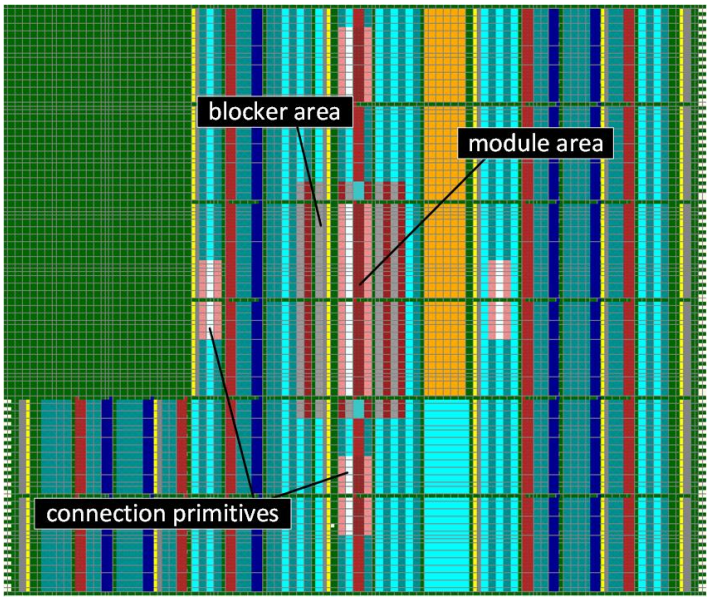

Figure 6.7: The floorplan of the module area and the connection primitives in GoAhead. Furthermore, the blocker area is defined around the module area to prevent routing outside the module area.

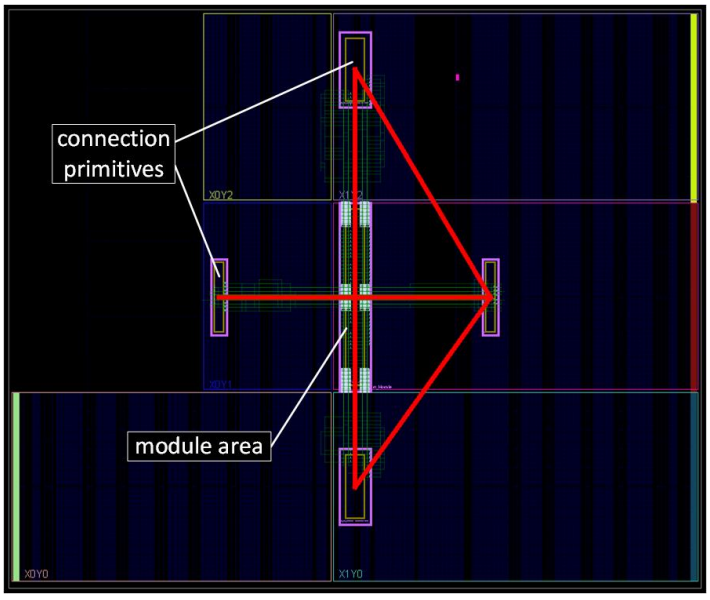

Figure 6.8: The result of module *SubBytes* after implementation in Vivado.

6.5 Configurations

Now that we have developed the modules, we can configure them into the partial area. In Figure 6.9, the placement of modules from configuration c_1 is illustrated. Note that with the footprints and input and output interfaces that we defined for the modules, this is the only feasible placement of the modules. The *SubBytes* module is connected to the *s2p* signal. The *ShiftRows* is internally connected to *SubBytes*, where *MixColumns* is internally connected to *ShiftRows*. In Figure 6.9, we also obtain two outputs: one output from *ShiftRows* and one from *MixColumns*. In fact, all the slots have outputs. However, these two outputs are selected by the multiplexers (see also Figure 4.8).

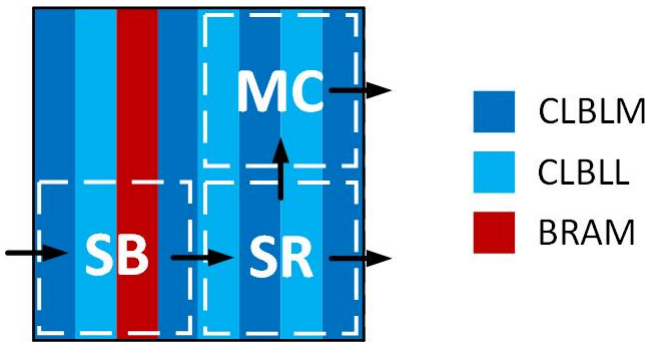

Figure 6.9: The placement of modules from configuration c_1. Module *SubBytes* is located in the slot $(0, 1)$, module *ShiftRows* in the slot $(1, 1)$, and module *MixColumns* in the slot $(1, 0)$. The two outputs are selected by the multiplexers.

In Figure 6.10, the placement of modules from configuration c_2 is shown. In this configuration, we use the same *SubBytes* and *ShiftRows* modules as that we used in configuration c_1. We relocate these modules to the respective location $(0, 0)$ and $(1, 0)$. This will change the hardware layout in comparison to the first configuration. Also, we use a variant of the *MixColumns* module to bring even more variation in the implementation.

This variant is illustrated in Figure 6.10. The module has an input interface at the northern border and an output interface on the west side. Furthermore, the module occupies two slots. We added a lot of dummy LUTs to this module to increase the power consumption. As illustrated in Table 6.2, the *MixColumns* module requires 2048 LUTs. Therefore, we need two slots. Note that the multiplexers have to select different outputs from the partial area in comparison to the previous configuration since the output of *ShiftRows* and *MixColumns* modules are located in different locations.

Until this point, we only developed two configurations. However, we should produce many more, such that we can configure the AES implementation continuously with variants. Also, the configuration that we have shown so far require reconfiguration of all the modules. It would be more beneficial to move from one configuration to the next one by configuring only one module. This decreases the reconfiguration time significantly, and therefore, we can obtain higher throughputs.

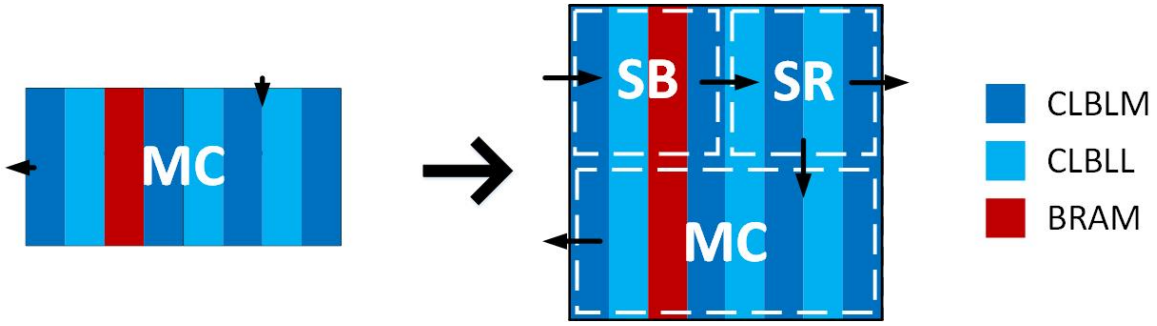

Figure 6.10: The placement of modules from configuration c_2. Modules *SubBytes* and *ShiftRows* from configuration c_1 are relocated to the top row, where a variant of *MixColumns* is configured in the bottom row.

6.6 Results

In this section, we shortly discuss the results. We implemented the two configurations that were discussed in the previous sections for the ZedBoard. We used several switches on this development board to make the input data (plaintext) variable, and we used several LEDs on the ZedBoard to obtain the result (ciphertext). We used a hardcoded key. Both configurations gave us the desired result.

The system runs on 20 MHz. In the first instance, we tried to run it on 100 MHz. However, by using this clock frequency, the system produced the wrong results. Then, we clocked the system down to 20 MHz, which provided the correct answers. We did not try to run the system on another frequency. Note that for our current framework, timing simulation is not supported. We discuss this in more detail in the next chapter.

Furthermore, as mentioned in Chapter 4, the only logic overhead of our DPR system are the multiplexers. As illustrated in Figure 6.3, we have six $p2s$ signals. Moreover, each $p2s$ signal has a bus width of 128 bits. Now, by using Equation 4.5, the overhead of a single multiplexer in terms of LUTs is the following.

$$MUX_{overhead} = 128 * \left\lceil \frac{6}{4} \right\rceil = 256 \tag{6.10}$$

As we have seen, a single multiplexer in this system uses 256 LUTs. However, since we use two multiplexers in this application, the logic overhead is $256 * 2 = 512$ LUTs (see Equation 4.6).

Unfortunately, while writing this thesis, we were not able to do measurements of this system. Therefore, we cannot conclude anything whether this approach is a sufficient countermeasure against physical attacks.

Chapter 7

Conclusions and Recommendations

In this chapter, we conclude the research and provide several recommendations for future work. This chapter consists of two sections. In Section 7.1, we state the conclusions, and in the next section, we describe the recommendations.

7.1 Conclusions

In this work, we introduced a framework that enables us to develop fine-grained DPR systems. The partial area is divided into two-dimensional slots, where modules can occupy one or multiple slots according to their resource requirement. The modules can communicate bi-directionally with the static area throughout all the slots that are adjacent to the static region. Moreover, adjacent modules can communicate, and non-adjacent modules can be linked by using by-pass modules. Furthermore, module relocation is supported, which makes the placement of modules very flexible.

The framework is demonstrated by a case study. In this case study, we show how the framework can be used for countermeasures against side-channel attacks.

7.2 Recommendations

Currently, the partial bitstreams are generated by using the tool BitMan. The partial bitstreams itself contain the information that specifies their location on the FPGA fabric. We use BitMan to cut out the module area from the full bitstream, and thus generate a partial bitstream. Also, we use the tool to specify their location on the FPGA structure. Since BitMan is an external tool, we have to generate partial bitstreams for each position on the FPGA fabric that we desire to configure a particular module. Therefore, this functionality should be implemented into the PRC. In this case, we still use BitMan to cut out the module area and generate a partial bitstream from this module area. However, we can change the location of this partial bitstream

in the PRC. Therefore, we require for each module only a single partial bitstream in the module repository.

Another missing aspect is the ability to perform a timing simulation. The reason is that we separate the static system and modules in different projects. Therefore, we have not a complete placed and routed netlist (static system and the modules in the partial area) and thus cannot perform a valid timing simulation. A solution to this problem could be to use external tools that can modify the placed and routed netlist. We could replace the slots of the partial area in the static system with the modules belonging to a particular configuration. By doing this, we obtain the placed and routed netlist of a specific configuration. Therefore, we can apply the timing simulation onto this design by using one of the vendor tools.

Finally, throughout this thesis, we suggested several points that must be considered during the floorplanning of the partial area. In island-style applications, it is rather easy. However, in grid-style applications, it becomes a lot more complicated. First of all, the footprint of the slots should be consistent, such that module relocation is supported among different slots. Furthermore, the designer has to consider in which slots of the partial area the module must be placed in a particular configuration. Also, modules can have different shapes and interface layouts. Therefore, an algorithm that calculates the most efficient placement of the partial area and modules is very convenient.

Bibliography

[1] C. Beckhoff, D. Koch, and J. Torresen, "GoAhead: A Partial Reconfiguration Framework," *IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 2012.

[2] D. Koch, *Partial reconfiguration on FPGAs: architectures, tools and applications*. New York: Springer, 2013.

[3] K. Vipin and S. A. Fahmy, "Automated Partial Reconfiguration Design for Adaptive Systems with CoPR for Zynq," *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014.

[4] A. Otero, E. D. L. Torre, and T. Riesgo, "Dreams: A tool for the design of dynamically reconfigurable embedded and modular systems," *2012 International Conference on Reconfigurable Computing and FPGAs*, 2012.

[5] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs," *2011 21st International Conference on Field Programmable Logic and Applications*, 2011.

[6] A. A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood, "OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs," *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011.

[7] D. Koch, J. Torreson, C. Beckhoff, D. Ziener, C. Dennl, V. Breuer, J. Teich, M. Feilen, and W. Stechele, "Partial Reconfiguration on FPGAs in Practise - Tools and Applications," *In ARCS Workshops (ARCS), 2012, pages 1-12*, 2013.

[8] C. Beckhoff, D. Koch, and J. Torreson, "Automatic Floorplanning and Interface Synthesis of Island Style Reconfigurable Systems with GoAhead," *Architecture of Computing Systems – ARCS 2013 Lecture Notes in Computer Science, pp. 303–316*, 2013.

[9] D. Koch, C. Beckhoff, and J. Torresen, ”Zero Logic Overhead Integration of Partially Reconfigurable Modules,” *23rd Symposium on Integrated Circuits and Systems Design (SBCCI), pp. 103-108. ACM*, 2010.

[10] K. Vipin and S. A. Fahmy, “Efficient region allocation for adaptive partial reconfiguration,” *International Conference on Field-Programmable Technology*, 2011.

[11] K. Pham, E. Horta, D. Koch, A. Vaishnav, and T. Kuhn, “IPRDF: An Isolated Partial Reconfiguration Design Flow for Xilinx FPGAs,” *2018 IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2018.

[12] D. Koch, C. Beckhoff, and J. Torrison, “Fine-Grained Partial Runtime Reconfiguration on Virtex-5 FPGAs,” *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, 2010.

[13] C. Beckhoff, D. Koch, and J. Torresen, “The Xilinx Design Language (XDL): Tutorial and use cases,” *6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*, 2011.

[14] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, “RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs,” *2011 21st International Conference on Field Programmable Logic and Applications*, 2011.

[15] K. Vipin and S. A. Fahmy, “Automated Partitioning for Partial Reconfiguration Design of Adaptive Systems,” *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, 2013.

[16] T. Townsend and B. Nelson, “Vivado design interface: An export/import capability for Vivado FPGA designs,” *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017.

[17] K. D. Pham, E. Horta, and D. Koch, “BITMAN: A tool and API for FPGA bitstream manipulations,” *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, 2017.

[18] S. Wildermann, J. Angermeier, E. Sibirko, and J. Teich, “Placing Multimode Streaming Applications on Dynamically Partially Reconfigurable Architectures,” *International Journal of Reconfigurable Computing*, vol. 2012, pp. 1–12, 2012.

[19] C. Lavin and A. Kaviani, “RapidWright: Enabling Custom Crafted Implementations for FPGAs,” *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018.

[20] P. Kocher, J. Joshua, and B. Jun, “Differential Power Analysis,” *CRYPTO ’99: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, Springer-Verlag 1999 ISBN 3-540-66347-9, 1999

[21] S. Ravi, P. Kocher, R. Lee, G. Mcgraw, and A. Raghunathan, “Security as a new dimension in embedded system design,” *Proceedings of the 41st annual conference on Design automation* - DAC 04, 2004.

[22] P. C. Kocher, “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems,” *Advances in Cryptology — CRYPTO ’96 Lecture Notes in Computer Science*, pp. 104–113, 1996.

[23] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, “Security in embedded systems,” *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 3, pp. 461–491, Jan. 2004.

[24] *Announcing the Advanced Encryption Standard (AES).* Gaithersburg, MD: Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, 2001.

[25] Xilinx Inc. 2012. *UG743: Partial Reconfiguration Tutorial PlanAhead Design Tool*. Xilinx Inc.

[26] Xilinx Inc. 2019. *UG888: Vivado Design Suite Tutorial: Design Flows Overview*. Xilinx Inc.

[27] Xilinx Inc. 2019. *UG473: 7 Series FPGAs Memory Resources*. Xilinx Inc.

[28] Xilinx Inc. 2018. *UG479: 7 Series DSP48E1 Slice*. Xilinx Inc.

[29] Xilinx Inc. 2016. *UG474: 7 Series FPGAs Configurable Logic Block*. Xilinx Inc.

[30] Xilinx Inc. 2010. *DS586: DS586: XPS HWICAP*. Xilinx Inc.

[31] Xilinx Inc. 2018. *UG909: Vivado Design Suite User Guide Partial Reconfiguration*. Xilinx Inc.

[32] Xilinx Inc. 2019. *UG912: Vivado Design Suite Properties Reference Guide*. Xilinx Inc.

[33] Xilinx Inc. 2013. *UG835: Vivado Design Suite Tcl Command Reference Guide*. Xilinx Inc.

[34] Xilinx Inc. 2018. *PG193: Partial Reconfiguration Controller v1.3*. Xilinx Inc.

[35] Xilinx Inc. 2012. *XAPP583: Using a Microprocessor to Configure 7 Series FPGAs via Slave Serial or Slave SelectMAP Mode*. Xilinx Inc.

[36] Altera. 2013b. *Quartus II Handbook Version 13.1*. Altera.

[37] Altera. 2016b. *Quartus Prime Standard Edition Handbook*. Altera.