POLITECNICO DI TORINO

Master's Degree in
Computer Engineering - Embedded Systems

Master's Degree Thesis

FPGA Acceleration in SmartNICs: Porting and Performance Evaluation of the Rosebud Framework

Supervisors

Prof. ***Luciano Lavagno***

Michele Paolino

Virtual Open Systems SAS

Candidate

Alessandro Vargiu

Matr. 314294

A.A. 2024/2025

Abstract

Field Programmable Gate Arrays (FPGAs) have emerged as a consolidated solution to address the challenges of high-performance networking applications, used in cloud computing environments, datacenters and telecommunication in general. The usage of FPGAs as accelerators for Smart Network Interface Cards (smartNICs) provides a flexible solution for tasks that require packet processing, such as deep packet inspection and firewalls. The inherent high performance and flexibility benefits of FPGAs are further improved by FPGA virtualization technology, which enables multi-tenancy execution through the distribution of FPGA resources, and by Partial Reconfiguration, which allows dynamic resource allocation and isolation, in order to optimize hardware utilization.

Today, several open-source solutions provide smartNIC functionalities to FPGAs, allowing the execution of tasks that are traditionally handled by CPUs to be offloaded to FPGA dedicated hardware accelerators. These solutions often provide network shells capable of 100Gbps+ speed, allowing the development of custom packet-processing accelerators that work on the lower layers of the network stack. However, many of these solutions lack support for hardware virtualization, leading to high virtualization overhead in software and lower hardware resource utilization.

This thesis was conducted during an internship at Virtual Open Systems in Grenoble, France, a company specialized in virtualization for embedded systems and development of solutions for automotive and cloud-computing environments. The company is interested in developing an FPGA virtualization solution that enables virtual access to custom hardware accelerators.

The study investigates a middlebox open-source framework, called *Rosebud*, designed to help the deployment of hardware accelerators, providing a high-speed network shell and a software interface to allow easy configuration and debugging of hardware accelerators. The research evaluates Rosebud's key features and its potential for future integration of virtualization capabilities. By deploying small RISC-V CPU

cores in the FPGA as control units, Rosebud enhances management of accelerators at runtime. Furthermore, Partial Reconfiguration allows accelerators to be programmed at runtime, maximizing flexibility of FPGA resources and reducing deployment time of new accelerators, which is a key problem in traditional FPGA development.

A porting procedure was performed in order to support the Alveo U55C accelerator card, which was originally unsupported.

The evaluation of Rosebud is performed with a custom benchmark that measures performance in a local network environment. An hardware accelerator was developed and deployed to test its impact on peak performance. The research demonstrates promising results for the use of Rosebud as a basis for a virtualized networking infrastructure.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Premise and work introduction

Today, the importance of computer networks to allow communication and data sharing has become paramount. Datacenter infrastructures have evolved and are now using technology such as FPGA-based accelerators, useful in cloud environments thanks to their reprogramming capabilities, and virtualization, to enhance hardware resource utilization and provide FPGA resources to multiple tenants. The flexibility of FPGAs, that allows updates after deployment, makes them a great solution to offload network processing in these infrastructures.

Virtual Open Systems is the company in which this thesis work was developed. The company specializes in virtualization for embedded systems and Linux software. Virtualization technology allows the partitioning of FPGA resources, mapping device physical functions into virtual functions, accessible through linux drivers by objects like virtual machines or containers.

The company is interested in developing a solution that integrates virtualization capabilities to hardware accelerators, in a networking context. Virtualization is a broad term that includes several technologies, all with the base principle of allocating physical resources to objects like virtual machines and containers, dividing one physical resource into multiple instances.

The partitioning of FPGA resources on a physical level is fully

leveraged by the Partial Reconfiguration feature, that allows for the allocation of reconfigurable partitions in the FPGA device. Each one of these partitions can be reprogrammed at runtime, allowing a flexible use of hardware resources.

The thesis work aims at the evaluation of Rosebud, an open-source framework that provides an hardware networking infrastructure, a software interface that abstracts the control of the hardware through the use of 32-bit RISC-V cores, running strictly connected to reconfigurable HW accelerators. The overall objective of the company is to investigate the features of this solution and eventually integrate virtualization capabilities to it.

The device used to test Rosebud is the Alveo U55C accelerator card, an high performance computing card designed for datacenter applications, containing a Xilinx Ultrascale+ FPGA. Since Rosebud does not support the u55c, a board porting process was done to map the card physical resources to the HDL components. A custom benchmark was then created in order to measure the network performance of Rosebud in the new environment and to evaluate its features. An hardware accelerator was also developed and deployed to test the impact of hardware overhead on peak network performance.

1.2 Thesis outline

This section outlines an overview of the thesis topics:

- **Background** contains all the knowledge necessary to understand the thesis work. It starts with a simple briefing, that shows the context in which the work manifests itself. Following, it is presented an overview of FPGA technology and features such as partial reconfiguration, virtualization technology, details about the Alveo U55C board used in the project and a paragraph on the state-of-the-art of existing FPGA solution for networking and virtualization.

- **Rosebud** describes the Rosebud framework, which is the middlebox solution adopted in this work. A section is dedicated to describe the challenges presented by porting a hardware design from one

FPGA board to another, detailing the various steps conducted to port Rosebud into the Alveo U55C, from logic synthesis to floorplan planning and implementation.

- **Performance benchmark** contains the implementation of a custom benchmark that evaluates the performance of Rosebud, through parameters such as bandwidth, latency and software overhead.
- **Conclusions** describes the conclusions for the thesis work and improvements for future work.

Chapter 2

Background

2.1 SmartNICs

Network Interface Cards (NICs) are devices that allow computers to connect to computer networks. They provide an interface through which computers can communicate over the internet (or LAN networks). Traditionally, NICs have been devices that handled basic network functions, such as Ethernet frames transmission and basic error checking functionalities, leaving more complex tasks (such as encryption, firewall and traffic management) to the host CPU.

The increasing demands of modern networking applications, along with the high-speed requirements for packet processing led to a new class of devices, called **Smart Network Interface Cards (SmartNICs)**.

Today, network link speeds are increasing rapidly and new technologies are being used, allowing network speeds from 10 Gbps to 100 Gbps and beyond. CPUs on the other hand, struggle to keep up with these rates, leading to performance bottlenecks. Performance improvement of processors has reached a limit, in terms of power consumption and dimension of transistors. Parallelism also provides limited performance benefits (Amdahl's law), and becomes almost useless in sequential executions tasks. A solution to this problem is to offload complex tasks from the CPU to hardware accelerators running on smartNICs. This way, the CPU can allocate more resources on application-level tasks.

Here are listed the main features and benefits provided by the utilization of smartNICs:

- **High performance**: packet processing in smartNICs happens at high line rate and the presence of hardware accelerators avoids all the performance overhead problems related to CPUs.
- **Energy Efficiency**: using hardware dedicated accelerators, tasks are performed faster and a lot of load is removed from CPU, resulting in lower power consumption.
- **Work Offload**: Systems like datacenters traditionally use a lot of processing power for infrastructure tasks (ex. packet processing), offloading tasks to smartNICs allow the freed processing capability to be used for application-level tasks.

There exist several solutions to implement hardware functions, from Application Specific Integrated Circuits (ASICs), to FPGAs and Data Processing Units (DPUs). This thesis will focus on FPGA networking applications.

2.1.1 Packet Processing

Packet processing is one of the most important features provided by SmartNICs. The existence of different types of architectures for smartNICs allow for high customization on these devices. SmartNICs can be used in different environment, such as datacenters, cloud computing infrastructures or general-purpose networks. Each environment presents different challenges, leading smartNICs to implement different categories of functions, such as security functions or storage functions. An example would be Deep Packet Inspection, used by Intrusion Detection Systems (ex. Pigasus), which requires tasks like pattern recognition using regular expressions (RegEx).

Incorporating small CPU cores, such as ARM-based CPUs, into smartNICs helps running complex logic tasks that can benefit from CPU execution pipelines. Having a dedicate CPU core inside the smartNICs also helps with the benefit of work offloading from the host CPU, as discussed earlier.

2.2 FPGAs

Field-Programmable Gate Arrays (FPGAs) provide a flexible solution to accelerate network applications. The advantage of FPGAs with respect to ASICs is that FPGAs are programmable, meaning that the hardware design can be changed and re-programmed on the FPGA. This is not possible on ASICs, since the production of the circuit requires a completed design that cannot be modified after production.

2.2.1 Architecture overview

The fundamental unit behind FPGA architecture is the **CLB (Configurable Logic Block)**. This is the basic unit used by FPGAs to implement logic functions. CLBs contain a set of look-up tables, flip-flops, adders and muxes. Look-up tables (*LUTs*) are a key component, since they can implement any boolean function. The presence of flip-flops enables storage capabilities, used in sequential logic. An example of a CLB element is shown in Figure 2.1. *CLBs* are connected through a routing matrix, containing vast array of configurable switches and wires, used to establish the right connections, depending on the function to be implemented. A high-level overview of the general architecture of a FPGA die is shown in Figure 2.2.

Modern FPGAs also contains special-purpose resources other than CLBs, such as DSP blocks, embedded memories or high-speed I/O logic, in order to provide optimized resources for several applications, such as Image Processing, Networking or Cryptography. FPGAs are usually integrated inside development boards or SOCs (System on Chip) containing high-speed *multi-gigabit transceivers*, interfaces such as *PCI Express* or memory controllers and even processors cores (ARM, MicroBlaze etc.). These components are tipically built using transistors instead of memory cells, providing ASIC-level performance to the board, without consuming base logic resources.

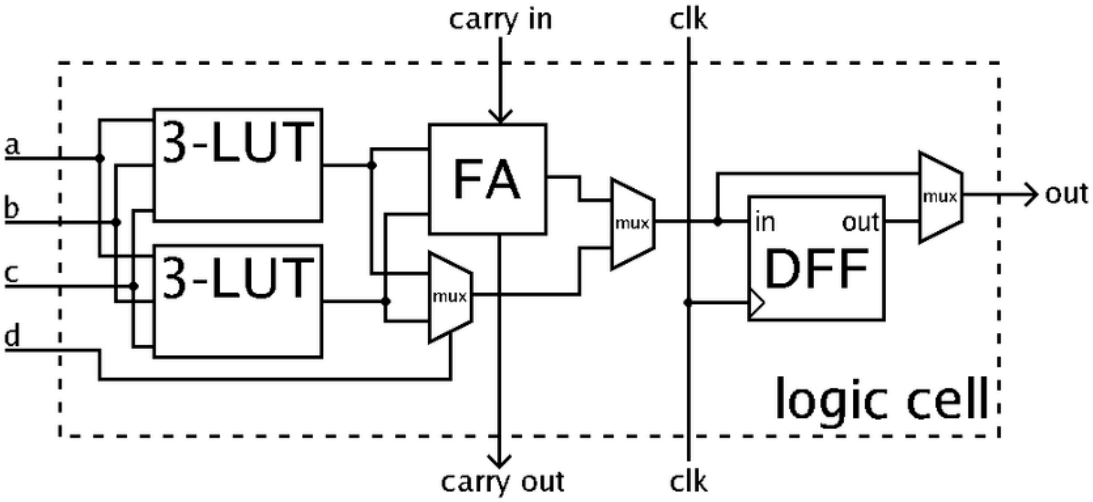

Figure 2.1: Example of simple Logic Cell structure

2.2.2 LUTs

Look-Up tables are small memory elements used to implement combinational logic functions. LUTs replace runtime computation with simple array indexing. When inputs are provided, the LUT fetches a precomputed result that correspond to the given inputs. LUTs are often implemented using SRAM technology, multiplexers, or a combination of both. The key concept behind LUTs is that the outputs are predefined for every possible combination of input signals. The LUT stores the output values for all 2^n combination of n inputs. For example, a 3-input LUT has $2^3 = 8$ possible input combinations, each tied to a specific pre-determined output. This structure can be implemented using a 8x1 multiplexer or a 8x1 SRAM memory, where selection signal combinations are used as addresses for the SRAM module.

2.2.3 Design Flow

FPGA design flow begins with Hardware Description Languages (HDL), such as VHDL, Verilog or SystemVerilog. Unlike traditional programming languages, HDLs are used to create a description of the behavior and structure of digital circuits. They are very similar to programming languages in terms of control structures, keywords or expressions, but the main difference relies on the concept of time and concurrency. HDLs are designed to model parallel processes that execute concurrently, reflecting the behavior of real-world hardware circuits more accurately. Similar to programming languages, HDL languages are compiled, but the result of the compilation consists in a gate-level netlist. The produced netlist represents the logic gates and the connections between them, defining a physical model.

1. **Behavioral Simulation**: At this stage, the code does not need to be synthesizable yet, and the tests are performed without timing information about the components. This stage allows to test the logic of the code and to verify that it behaves according to expectation. Simulation tools typically allow to visually see how inputs, outputs and internal signals behave, through *waveforms*. Languages such as

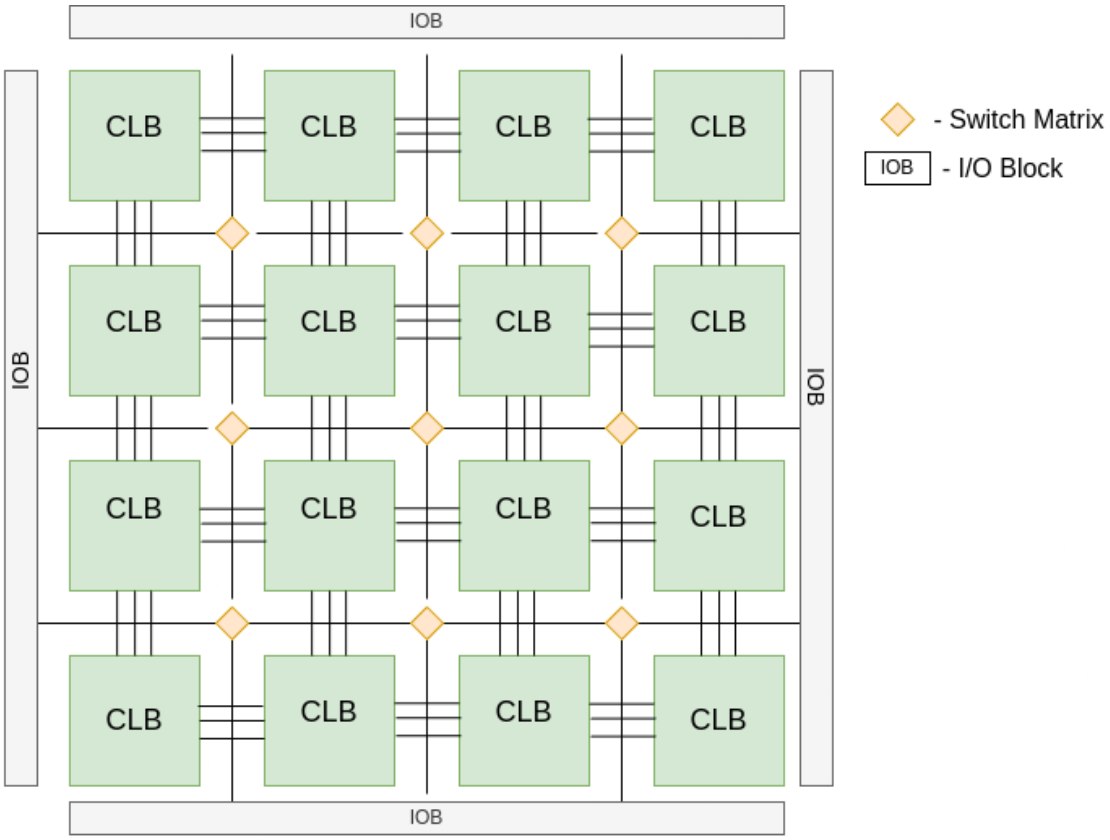

Figure 2.2: Basic example of general FPGA architecture, showing connections between CLBs

Verilog also provide macros to verify the correctness of the outputs using *assertions* and performing various checks, without resorting to visual information, which is intrinsically less efficient. In practice, the RTL code is tested through the use of *testbenches*, which are HDL wrappers that instantiate the unit under test (UUT) and contain a process that provides the correct input signals and verifies the outputs.

2. **Logic Synthesis**: The purpose of this process is to convert HDL code into a gate-level representation of the design, called *gate-level netlist*. The synthesis tool analyzes the code and performs different optimizations in order to improve the design with regards to timing, area and power consumption. Additionally, technology mapping is performed, in which the RTL logic constructs are mapped to specific resources such as LUTs, memories and other components, depending on the resources available in the target technology library. After synthesis, it is possible to collect information by generating reports, detailing information about resource utilization, timing or power consumption. Timing reports are very important to determine if the design is capable of operating at the desired clock frequency, and if the chosen technology is capable of supporting the requirements criteria. Simulation is also possible at this point, using the post-synthesis netlist as UUT. This allows to check that the netlist produced after synthesis is coherent with the initial simulation results.

3. **Place and Route**: This process takes the synthetized design and implements it on the target device. It is the most computationally heavy stage in the design flow and it consists on several steps. Here are shown the steps taken by Xilinx's Vivado implementation tool:

    - **Logic Optimization**: A series of checks are performed to avoid issues such as undriven inputs or connectivity errors. Then, a series of optimizations are performed on the logic, such as *Constant Propagation*, in which constant values are propagated throughout the logic, reducing or eliminating unnecessary logic

(for example, an AND with a constant 0 input). Another example is *BUFG Optimization*, in which global clock buffers are inserted on clock nets and high-fanout non-clock nets to improve reliability and performance of clock signal distribution. Other optimizations are related to improve synthesis results, such as area or power consumption.

- **Placement**: The Placer tool assigns cells from the netlist into specific sites on the FPGA device fabric. Optimizations at this step address the improvement of*timing slack*, *wirelength* of connections and minimization of*congestion*.

  Before placing the primary logic, placement of clock and I/O cells is done first. Global resources like clock buffers or clock management components, such as MMCMs and PLLs are placed. Synthesis can be repeated several times by the tool to refine the results of each iteration. Examples of optimizations at this stage are *LUT Combining*, *retiming*, *Fanout Optimization* and moving Block RAM registers to improve critical paths.

- **Routing**: The tool attempts routing on all nets, determining the exact paths for the interconnections between components. A series of algorithms are implemented in order to establish the optimal paths for signals, while ensuring the respect of timing requirements. Typically this happens through multiple iterations. Timing analysis is also performed to ensure the meeting of timing constraints.

4. **Bitstream generation**: When routing is successful and it meets timing, area and power requirements, the output netlist can be converted into a bitstream (a long sequence of bits). Despite the common misconception that these bits will "fill up" the basic cells of the fpga device, a bitstream can be viewed, instead, as similar to an executable program, containing an *assembly-like* instruction set to describe the configuration process. The knowledge of bitstreams format is not necessary to understand the programming of the fpga, but it can be useful to do reverse engineering or bitstream readback. Bitstreams are usually programmed into the fpga through interfaces

such as JTAG or SPI. FPGA boards usually contain a flash memory dedicated to store a bitstream, which can be read automatically at the FPGA bring-up during a system reboot. With large designs, bitstream generation and flashing can be time consuming and interfaces such as PCI Express can be used for this purpose. This process is better detailed in the *Partial Reconfiguration* section.

2.3 Partial Reconfiguration

Partial Reconfiguration is a feature of FPGA development that allows dynamic modification of portions of hardware logic, while the other portions of the design continue to operate. In Xilinx's documentation, Partial Reconfiguration is referred to as *Dynamic Function eXchange* (DfX), but throughout this thesis, the first term will be used. Typically, reconfiguration of an FPGA involves the loading of a full bitstream into the FPGA's flash memory. Partial Reconfiguration works with partial bitstreams, reducing configuration time. PR works with the PCIe interface, that provides higher data transfer rates than micro-USB interfaces, generally used to write the flash memories of FPGAs. Partial Reconfiguration is crucial for designs utilizing the PCI Express (PCIe) interface. The PCIe specification requires devices on the PCIe bus to be detectable by the host and respond within 100 milliseconds after power is applied to the PCIe slot. Partial Reconfiguration helps the deployment of large designs on FPGA, providing the possibility of base functionalities to be activate at power-up and upload the rest of the logic at a later stage.

The portion of design that remains active during Partial Reconfiguration is called *Static partition*, which contains global logic and clocking resources that must stay operational during the reconfiguration process. The dynamic portion is called *Reconfigurable partition*.

One important aspect when creating reconfigurable designs is the placement of the static logic. After the synthesis process, tools like Vivado can display where the synthetized logic will be placed in the target device. These regions should remain untouched during partial reconfiguration. This is ensured if *static logic* and *reconfigurable logic*

are physically separated on the FPGA.

The technique used to achieve this is called *Hierarchical Design*. This approach breaks down the entire design into modular components. Each module performs a specific function or represent a specific hardware block, similarly to a real circuit. This allows synthesis tools to synthesize components independently from each other, and in the case of Partial Reconfiguration, it allows the synthesis of reconfigurable modules separately from static logic.

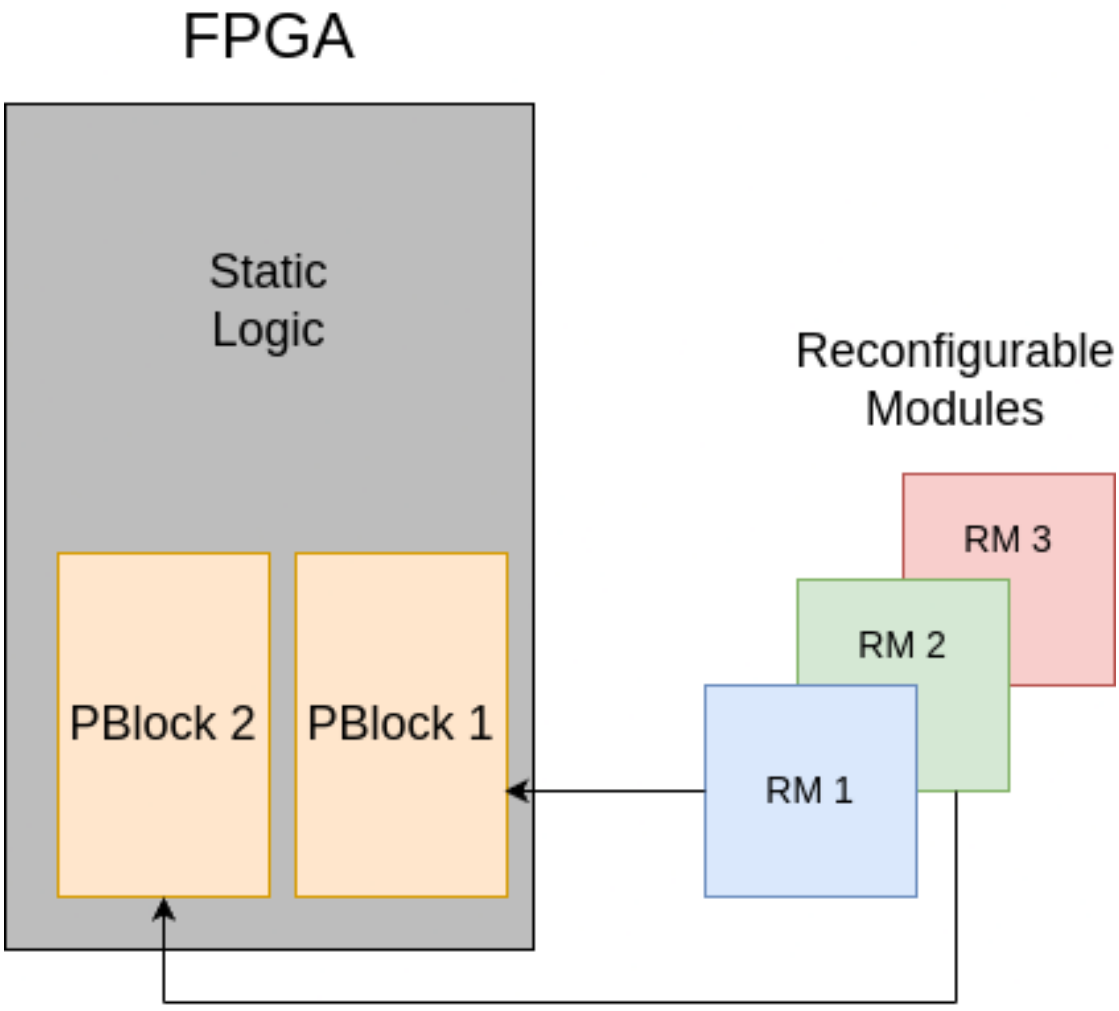

Figure 2.3: Partial Reconfiguration high-level architecture

2.3.1 Partial Bitstreams

Bitstreams are used to fill up the entire FPGA with information about how each logic element inside should behave at runtime. Usually they are imagined as series of bits which activate logic elements inside the FPGA, but the reality is that the activation of the fabric logic is done through a series of sequential commands at runtime. Given the possibility of altering only some logic elements, it is allowed the creation of partial bitstream. Partial bitstreams must be compatible with the static logic, which means that the creation of static and partial bitstreams is coupled. If the procedure is done correctly, the static logic remains untouched while running.

2.3.2 Floorplanning

Floorplanning is the process of allocating resources, specifically design components, to a specific region of the target fpga device. In a traditional design, the tools automatically place the components in the device, based on the requirements.

Pblocks Pblocks are a set of physical constraints used to define a reconfigurable region within the FPGA. When a Pblock is created, a set of resources is allocated to it. This can be done using graphical tools, that allows the placement of rectangular sites in the device regions (a rectangular shape is recommended in order to avoid routing problems caused by irregular shapes). These resources are allocated with the purpose of being used by a reconfigurable module that will be placed in that region. These resources can be used also by static logic for routing purposes. This seems a little counterintuitive, however it is assured that no conflict arise, even during the partial reconfiguration process.

Implementation Partial Reconfiguration requires a two step implementation, the first is called *Parent Implementation*, where the entire design flow is executed, which includes the synthesis of the static logic and the optimal placement of all the static components in the device.

The next step is the *Child Implementation*, here the static logic synthesis is skipped, since its placement will remain exactly the same. Only the reconfigurable modules are synthetized at this step, since the synthesis product from the previous run have been already generated. In the implementation phase, the whole netlist is placed and routed again, however the algorithm now takes into consideration the constraints on the static logic, since its placement must be the same as the previous run.

It is generally a good practice to perform an implementation of the static logic alone, to see where the tools will place the logic in the target FPGA. This provides a clearer view of how much space remains for the reconfigurable regions.

When a reconfigurable module is selected and linked to a Pblock, Vivado shows us the resources required by the module and the ones that are present in the selected site. This way, it is possible to check if the selected region contains all the necessary resources required by the module instance. This detail will be better highlighted and shown in the next chapter.

Another important best practice is to frequently perform *Design Rule Checks* (DRC) after floorplan modifications. It is important to perform them before starting the implementation process. Depending on the complexity of the design, the placement and routing processes can take several hours. Having errors or warnings highlighted early can save a lot of time. If the process is successful, a series of partial bitstream will be generated, each corresponding to a reconfigurable module.

2.4 Virtualization and FPGAs

Virtualization generally refers to a set of technologies that divides a physical device into multiple virtual version of itself, allowing each virtual function, or object, to work as if it were a separate device.

There are several types of virtualization, such as *Hardware Virtualization*, *Operating System Virtualization*, allowing multiple isolated operating systems to run on a single operating system kernel, *Network Virtualization*, that abstracts physical network devices into multiple virtual network devices.

Virtualization in FPGA is classified in three main categories [1]: *Resource level*, *Node level* and *Multi-node level*.

Resource level refers to architecture and I/O virtualization. A resource on an FPGA can be a kernel or a general module, reconfigurable or non-reconfigurable. In practice, this type of virtualization on an FPGA makes it so that physical I/O resources on the FPGA are virtualized into multiple channels at the user-level. The SR-IOV specification allows a PCIe device to be virtualized into multiple physical (PF) or virtual (VF) functions. In Xilinx FPGAs, this technology is offered using the PCIe Integrated System IP, allowing easier integration of the SR-IOV specification to FPGA shells.

Node level refers to the management of virtualized resources in an FPGA. A simple example of this is a FPGA shell containing multiple accelerators, that can be managed and assigned to virtual functions.

Multi-node level refers to the coordination of resources through multiple FPGAs. This type of virtualization is not in the scope of the thesis work context, so the focus will be on the previous categories.

In *Node level* virtualization, a common solution to achieve multiple applications working on the same FPGA is the usage of multiple partial regions. Using a symmetric approach, identically sized regions can be placed on the FPGA, running independently from one another, each one designed to be accessed by a single application. An asymmetric approach could be used by more complex designs that require modules to have different sizes. This approach requires more caution, since can lead to internal fragmentation in the FPGA device.

Overall, this method provides a lot of flexibility and fast reprogramming of resources in the FPGA, but it has been shown that it leads to strong layout constraints regarding logic placement in the device. Finding an optimal placement of the partial regions is not easy and it is open to research.

2.4.1 SR-IOV

Single Root I/O Virtualization interface is an extension of the PCI Express specification. SR-IOV technology allows a physical device to be shared among multiple virtual machines, or containers. A PCIe physical function (PF) is associated with multiple virtual functions (VFs), that can be accessed by the virtual machines. A VF can be configured to access one or more physical resources of the device. An example is a VF linked to a physical port in an FPGA or a VF linked to a reconfigurable accelerator. SR-IOV bypasses hypervisor's software layer, reducing I/O access overhead in the emulation layer and leading to improved latency and throughput. The high-level diagram in Figure 2.4 shows the concept of SR-IOV technology, highlighting the bypass of the hypervisor that allows to improve virtualization overhead.

2.5 Alveo U55C card

The Alveo U55C card was used to implement Rosebud. It is an accelerator card designed to do high performance workload in environments such as datacenters, with applications ranging from data analytics to machine learning. The card provides up to 200Gbps networking speed, split in two QSFP28 ports, up to 100Gbps each. It also packs two high bandwidth memory units, with a total capacity of 16GB, working up to 460GB/s bandwidth. The main interface used to communicate with the card is the PCIe interface, generally used to access internal registers of IP blocks and to do partial reconfiguration. In this section, the interfaces provided in this accelerator card are shown.

2.5.1 Network Interface

The Alveo U55C has two QSFP28 ports, or cages, with a 4-lane format. QSFP28 specification allow for a bitrate up to 100Gbps. Each QSFP28 module has 4 channels, running at 25Gbps each. Xilinx's GTY transceiver are the serial communication modules that interfaces with the optical interconnections of the QSFP28 specification and provide

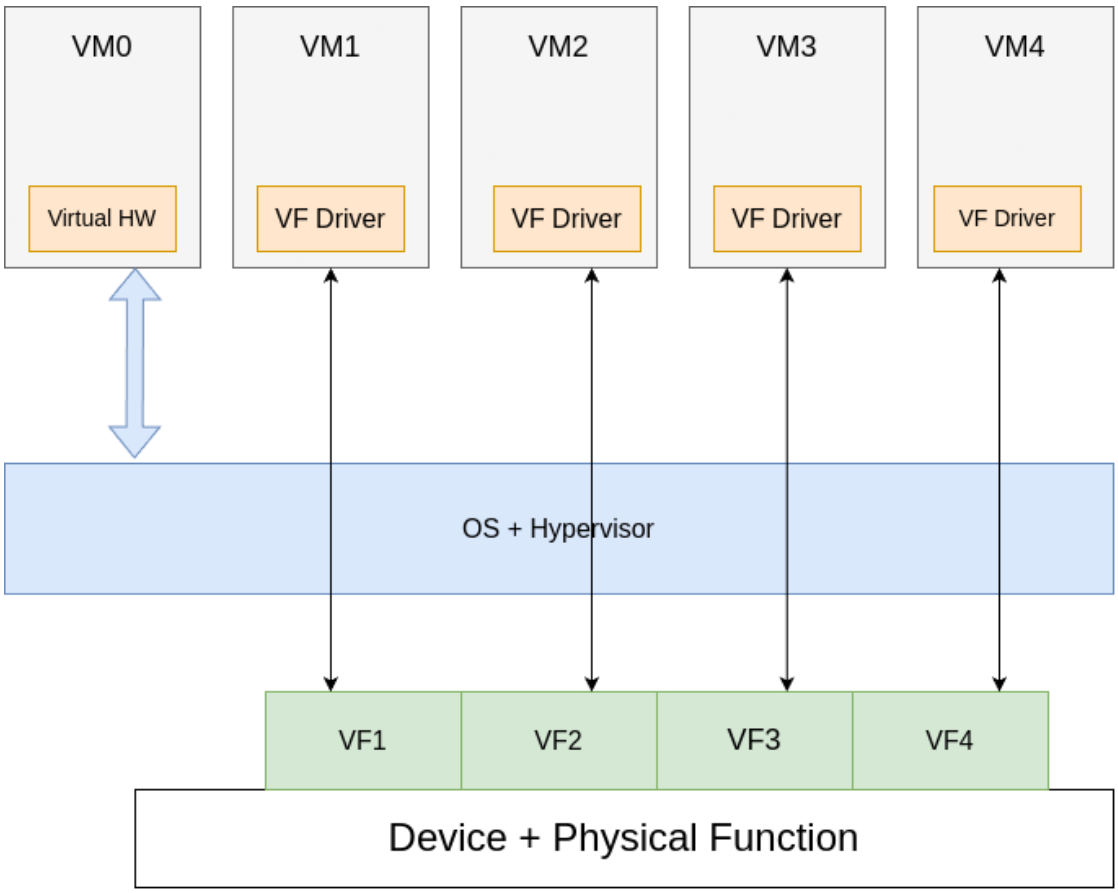

Figure 2.4: SR-IOV model architecture

the digital signals to the fpga device. The two QSFP28 cages also provide a clock source running at 161.1328125 Mhz that can be used to feed several ethernet IPs.

2.5.2 PCIe Interface

The U55C implements a 16-lane PCI Express bus, performing data transfers up to 8 GT/s (Giga Transfers per second) for PCIe Gen3 and 16 GT/s for Gen4. Xilinx provides IP blocks to interact easily with the interface and enable advanced features, such as SR-IOV configuration and Partial Reconfiguration functionalities.

2.5.3 XCU55C Ultrascale+ FPGA

The XCU55C is the Xilinx FPGA contained in the Alveo card. The fpga device uses *Stacked Silicon Interconnect* (SSI) technology to combine three *super logic regions*, each region providing independent sets of resources, such as I/O cells, DSP blocks, RAM cells and others. The usage of these super logic regions will be better detailed in the porting chapter, during the floorplan phase.

2.5.4 Clocking Structure

Several clock sources are connected to the fpga device. Two 161.132 MHz clock sources are dedicated to Ethernet applications, as this specific clock frequency is generally used to interface with QSFP28 modules, which support up to 100Gbps data rates. These two clocks sources are generated by a programmable oscillator, along with a third clock source running at 100Mhz, typically used for general-purpose IPs. If a different clock frequency is required, IP modules such as PLLs or MMCMs can be instantiated. These modules take the generated clocks as input, producing an output of another required frequency. It is important to understand that each source is usually dedicated for one specific region of the device. This is crucial during pin assignment, since choosing the wrong clock source can lead to problems in the implementation stage, especially during routing, where timing requirements are met.

A high-level representation of the U55C clock tree is shown in 2.5, detailing the connections between the different clock sources and the banks associated to the device regions.

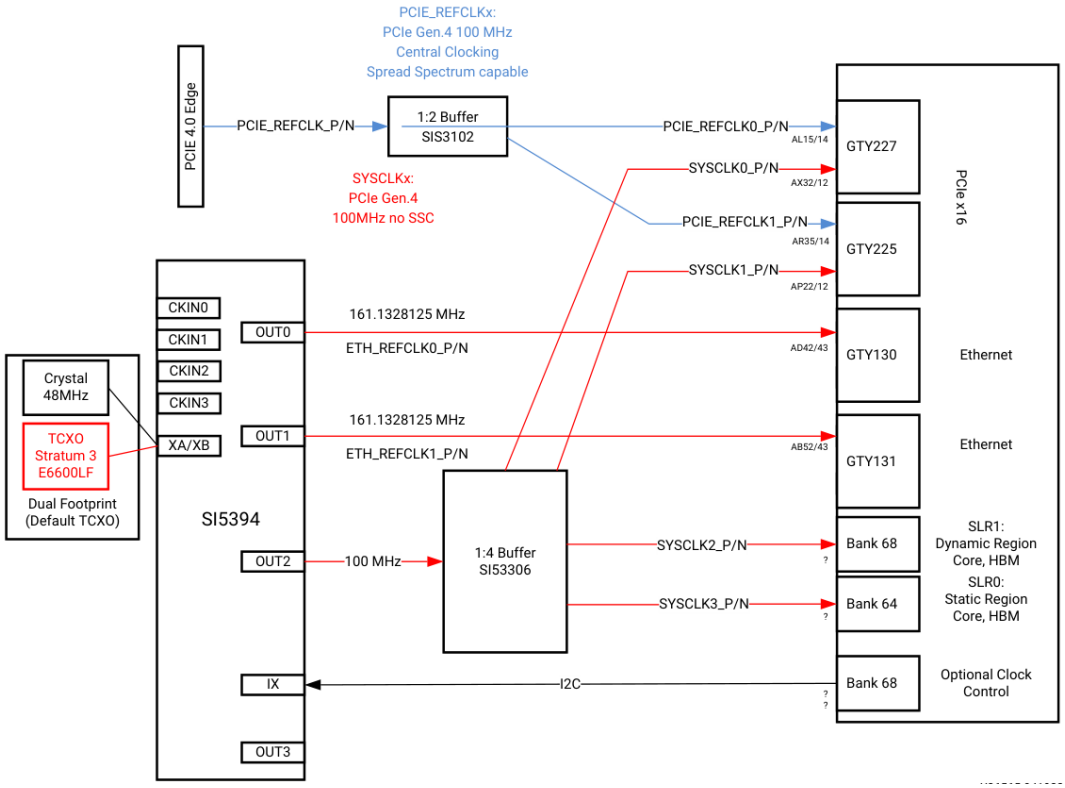

Figure 2.5: Clock tree structure in the Alveo U55C

2.6 Related Works

As of today, several solutions exist to implement custom smartNICs on FPGAs. Several solutions focus on providing a preconfigured hardware shell that includes a basic networking stack infrastructure, enabling an ethernet interface to the FPGA and minimal packet storage capabilities. The main purpose of these projects is to easily allow the integration of

hardware accelerators that realize more complex computation, enabling complex networking functionalities.

Corundum is a full open-source platform that includes a high-speed ethernet datapath up to 100Gbps and implements a full network stack, with a DMA interface. The use of AXI interfaces allow for the easy integration of new modules into the design. Except for the 100G MAC implementation, which uses the Xilinx proprietary CMAC IP, every other hard IP is fully open-source.

OpenNIC is a project that provides a basic NIC shell, based on the 100G CMAC IP, with linux drivers and support for DPDK drivers. It contains the Xilinx QDMA IP, enabling multiple physical functions on PCIe. The presence of the QDMA is interesting, since it enables the use of multiple physical functions through PCIe and, though not supported yet, can allow for virtualization capabilities to be implemented through SR-IOV.

EasyNet implements a 100G TCP/IP network stack using Vitis-HLS. It provides a network kernel, connected through AXI interface, to a user kernel, allowing the implementation of external logic that interacts directly with the TCP stack. The usage of Vitis allow the user to eventually develop the user kernel in high-level languages such as C/C++, avoiding the complexity of HDL languages.

All these solution helps prototype network platforms for hardware accelerators, however virtualization is not supported. Solutions for FPGA virtualization are found in academic and commercial fields.

An overview of the main techniques and categories of FPGA Virtualization are found in this paper [1]. Here are some industrial solutions for FPGA Virtualization:

SVFF [2], by Virtual Open Systems, is a Virtual Function Framework that simplifies the management of virtual functions on FPGAs devices, leveraging the SR-IOV support provided by the QDMA IP. This solution allows the attachment and detachment of virtual functions to/from virtual machines.

A solution that provides both an hardware and software framework is shown in [3], where partial reconfigurable regions are offered as generic cloud resources through **OpenStack**, allowing the User to boot custom

accelerators in the same way a virtual machine is booted.

Feniks [4] is an FPGA operating system that provides an abstracted interface for FPGA accelerators, providing direct resource access through the PCIe bus. This research highlights the importance of the PCIe interface as a solution to interact with FPGA resources in environments that makes high use of virtualization, such as datacenters. As an example, companies such as Microsoft started to implement FPGA acceleration in their servers to improve performance of the *Bing* search engine [5].

Optimus is an hypervisor that supports FPGA virtualization using memory-shared architecture, providing FPGA resource multiplexing through the isolation of virtual accelerator's DMAs, with hardware-software co-design.

[6] presents a framework that provides integration of hardware acceleration in cloud environments using Partial Reconfiguration. A case study shows the benefits in terms of efficiency to integrate PR virtualized designs in cloud environments, in relation to other static acceleration frameworks.

A solution that integrates Virtualization and sharing of reconfigurable FPGA resources is shown in [7], in which a PR-based framework is showcased on High Performance Reconfigurable Computers (HPRCs), that are parallel computers with multiple FPGA sources.

RosebudVirt [8] emerged as a framework during the last stages of this research. It is an enhancement of Rosebud, adding SR-IOV capabilities and compatibility with Kubernetes and Docker.

Chapter 3

Rosebud

Rosebud is an open-source framework for FPGA middlebox implementations. A middlebox is a networking device that implements network functionalities such as Packet Inspection, Firewall or Intrusion Detection. The goal of Rosebud is to abstract hardware components into a single unit called RPU (*Reconfigurable Packet-processing Unit*). This unit is a reconfigurable hardware block, designed to act as a network accelerator. Its main function is to elaborate network traffic, more specifically to handle packets. To control this unit, Rosebud integrates RISC-V based CPUs, that are instantiated in the FPGA and allow debugging via software, avoiding all the problems related to implementing control logic in hardware. A memory architecture is built around the RPUs, including memory blocks for storing packets and stack memories to support programs running on the RISC-V cores.

In the context of smartNICs, Rosebud tries to help developing FPGA designs in a way that is more similar to traditional software development. FPGA development is slow, with processes such as placement and routing take hours to complete. Furthermore, debugging capabilities on FPGAs at run-time are poor. Typically, debugging is done at RTL level, using simulations to test the design before implementing it on hardware. The problem is that debugging networking designs at RTL level is complicated, due to the difference in timing between simulation and real hardware. Simulating a multi-gigabit network design could take hours for the transmission of a couple thousands of packets, whereas

real hardware can process millions of packet per second. The presence of the RISC-V core in Rosebud is designed to get easy access to internal signals of the accelerators and to its internal memory structures, such as the small memory buffers used for packet storage.

This section will explain the architecture of Rosebud, showing the main hardware components and the software infrastructure.

3.1 Hardware Design

3.1.1 RPU

Rosebud's design is centered around the *RPU* (Reconfigurable Packet-processing Unit). Each RPU contains two main components: a RISC-V core and an hardware accelerator. These two components communicate through two memory structures: an *I/O memory map* between the core and the accelerator, providing direct access to hardware registers by the RISC-V core, and a *shared memory module*, primarily used to store packets for processing.

The HW accelerators perform high time-consuming operations on packets, reading them in a streaming manner (data is read n-bit at a time). The RISC-V core, on the other hand, can access packet data in a more flexible, random-access way, allowing it to fetch packet headers or modify specific fields within the packet. This dual approach creates a balance between high-speed streaming and flexible packet processing. For this reason, the hardware accelerators use Xilinx's *Ultra-Ram(URAM)* modules. URAMs are physical blocks of high-latency memory integrated in the FPGA fabric logic that can be pipelined to hide the high latency. The importance of these Ultra-Ram modules will be detailed further in the *porting* section, highlighting how the number of URAM blocks in the U55C makes it very difficult to implement certain types of accelerators, especially when aiming for a standard number of RPUs to be allocated, such as 8 or 16. The RISC-V core, instead, uses *Block RAM (BRAM)* modules, which are more abundant in the FPGA fabric.

All RPUs are all placed inside *partial regions*, enabling them to be

reprogrammed at runtime, while the rest of the design still operate.

Partial Reconfiguration allows for versatile packet-processing functionality. For example, an RPU initially configured to implement a firewall accelerator can be reconfigured at run-time to perform other tasks, such as hash computation or packet inspection. Each RPU is contained in an isolated partial region. The partial regions are all isolated and independent from each other, which means we can address them separately, without affecting operations of other RPUs. Partial reconfiguration is executed through Xilinx's MCAP (Management Configuration Access Port) drivers, where MCAP is an advanced configuration interface designed for PCIe applications.

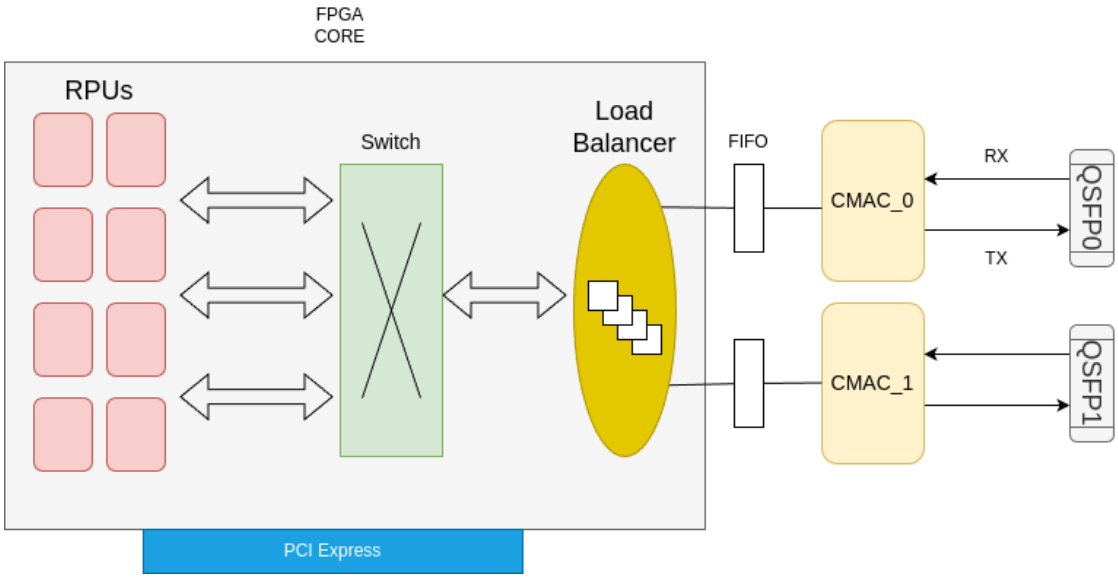

Figure 3.1: Overview of Rosebud's architecture

3.1.2 RISC-V core

The RISC-V core is a softcore CPU that is used as a way to control the hardware accelerator. Each allocated RPU in the FPGA contains a small RISC-V core based on the VexRISC-V implementation. VexRISC-V is an open source 32-bit version of a RISC-V processor, optimized for FPGAs. The core runs at 250 Mhz, which is relatively slower with

respect to the 322 Mhz speed of the ethernet interface. This frequency, however, is enough to handle 100 Gbps processing using the core alone, for example using it as a packet generator.

This allows to concentrate only on CPU latency as a potential limitation of the RISC-V core usage.

3.1.3 Load Balancer

Load balancing refers to the process of distributing computational workload between different resources. In the context of networking, the workload becomes the network traffic. Having a component dedicated to this task helps speeding up performance and reduce latency.

Rosebud is designed to run with several RPUs working in parallel, leading to the load balancer being very important to split the traffic between RPUs, especially since it works at high-speed line rates.

The load balancer addresses the packets before sending them to the RPUs, it interacts with the packet slots using descriptors, that indicates the memory slot number. The enforcement of the load balancing policy is done labeling the received packets, writing values indicating the target RPU and the destination memory slot. An interface is provided which allows to control the Load Balancer during runtime. This interface can be used to enable or disable cores and provides access to a few status registers.

3.1.4 Network shell

The network shell is based on *Corundum*, an open source FPGA design that creates a NIC infrastructure and a virtual network interface, through linux drivers. The networking infrastructure is composed of several HDL modules. Rosebud uses a small set of these modules, such as queue managers, PCIe modules, to allow connection to host DRAM, and linux networking stack drivers. The physical ethernet interfaces (QSFP modules) are connected to a 100G MAC module, provided by Xilinx, called CMAC (Centralized MAC). This block handles the level 2 of the network stack, the data link layer. Network packets go through

this module and are handled by a packet distribution system, alongside some FIFO modules, for buffering and streaming purposes. The CMAC module provides up to 512 bits data width, allowing a throughput of more than 100 Gbps:

$$\text{Throughput} = 512\,\text{bits} \times 322 \times 10^6\,\text{Hz} = 164.224\text{Gbps} \tag{3.1}$$

Given that the U55C has two 100G QSFP ports and each ports has a dedicated CMAC module, there are no stringent constraints from an hardware point of view and we can expect to have up to 100Gbps bandwidth on each port.

3.1.5 Software Interface

The purpose of the software interface in Rosebud is to control the RPUs. Each RPU has one RISC-V CPU core and one hardware accelerator. A custom C library called `core.h` is used in order to enable the CPU cores to perform operations such as reading packets descriptors, forwarding packets to specific interfaces and debug functionalities such as reading timer values, useful in timestamping.

The libraries use a baremetal approach, with C functions writing directly in MMIO regions, using memory addresses that belong to hardware registers. The library code bypasses software abstractions or high-level APIs and writes values into internal variables, containing memory locations. Here is an example:

```
#define SEND_DESC (*((volatile struct Desc*)(IO_INT_BASE + 0x0008)))
#define SEND_DESC_TYPE (*((volatile unsigned char *)
                        `(IO_INT_BASE + 0x0028)))
static inline void pkt_send(const struct Desc* output_desc)
{
    SEND_DESC = *output_desc;
    asm volatile("" ::: "memory");
    SEND_DESC_TYPE = 0;
}
```

The `pkt_send` function, provided by the Rosebud library, is used in a C program that runs in the risc-v cores to send a network packet outside

the RPUs. The *Desc* struct includes some fields indicating the port in which to send the packet, the packet length and a tag, used to link a specific RPU to a packet. The variable `SEND_DESC` is directly declared as a specific memory location using offsets. `asm volatile("" ::: "memory")` is a technique used in low-level programming, in order to prevent *reorder* of memory operations that the compiler might perform. This instructions acts as a memory barrier, ensuring that memory operations happening before this instruction must complete before other instructions that follow it. The assignment of 0 to `SEND_DESC_TYPE` is used to forward the packet to an ethernet interface.

Linux drivers

Rosebud provides a set of drivers for the linux network stack. These drivers enable a virtual ethernet interface, so that Rosebud is seen as a NIC on the network interface. This virtual interface is linked in hardware to the PCIe interface.

3.2 Implementation on Alveo U55C

Rosebud provides support for the Alveo U200 and the VCU1525 FPGAs. However, several modifications were required in order to implement Rosebud on the Alveo U55C. Both the U200 and the U55C include QSFP modules; however, they are controlled in a different way. The U200 provides an interface that can directly access the QSFP control signals through an *I2C* bus on the card, while the U55C can access the same signals through the use of a Xilinx proprietary IP, called *CMS (Card Management System) subsystem*. These signals provide access to QSFP sensor data and control over reset and status signals. The integration of this IP is not relevant for the scope of this work, therefore, it was not integrated into the design. However, this addition could be a feature to add in future work.

Porting FPGA designs to different platforms involves addressing differences in *logic resources* (LUT, flip-flops), *embedded memory blocks* (BRAM, URAM), *DSP slices*, *clock management modules* (MMCM,

PLLs) and dedicated hardware (CPUs, ethernet interfaces, HBM modules). These differences are handled through the modification of synthesis and implementation constraints, thus performing a mapping between the device resources and the RTL modules.

XDC (Xilinx Design Constraints) constraints are used to manage this mapping. They are commands that use the *TCL* script language, used by Vivado, and they are stored in .xdc files. These constraints control the placement of resources, timing specifications and other details that helps achieving optimal performance on the target FPGA.

3.2.1 Synthesis Constraints

Synthesis constraints are critical to define clock setups and timing paths in the FPGA design. The tools automate a lot of these processes, however, large designs containing multiple clocks and clock domains require precise specifications in order to maximize coverage for timing paths. Additionally, they perform an initial mapping of the hard blocks provided by the target device, such as transceivers and clock sources.

The Alveo U55C documentation provides a template of a xdc file, detailing the FPGA I/O pins and their purpose, along with an image of the Clock Tree structure, that details the connections between GTY banks and SLR regions with different I/O pins and clock sources.

The `set_property` command is used to associate netlist module ports with specific FPGA pins.

Here is an example in which the reception signal of the QSFP module's positive lane is linked to an FPGA pin:

```
set_property -dict LOC AD51 [get_ports qsfp0_rx1_p]
```

AD51 refers to the specific I/O pin and *qsfp0_rx1_p* is the port provided as input to the Ethernet module.

The full set of synthesis constraints is provided in Appendix A.

Both the QSFP modules and the PCIe interface require dedicated clock sources. Using the *clock tree diagram* in Figure 2.5, the correct pin assignment is made based on which bank the design is using. Vivado automatically generates a clock object for the QSFP clock during the synthesis process of the *CMAC IP* (the link layer block).

However, the primary clock for the PCIe interface must be created manually:

```
create_clock -period 10.000 -name pcie_mgt_refclk_1
[get_ports pcie_refclk_p]
```

This command defines a 100Mhz clock signal as input to the PCIe module. It tells Vivado about the port and frequency of an external clock entering the FPGA and sets up the timing constraint. The PCIe module also requires an external source for the reset signal, assigned to pin `BF41` in this case:

```
set_property PACKAGE_PIN BF41 [get_ports pcie_reset_n]
set_property IOSTANDARD LVCMOS18 [get_ports pcie_reset_n]
set_property PULLUP true [get_ports pcie_reset_n]
set_input_delay 0.000 [get_ports pcie_reset_n]
set_false_path -from [get_ports pcie_reset_n]
```

The `set_property PACKAGE_PIN` assigns the specified pin to the reset port of the PCIe module.

`set_property IOSTANDARD LVCMOS18` sets the input/output standard for the signal, ensuring that it operates at 1.8V logic levels. The Alveo U55C documentation provides the values for each pin. The `PULL UP` property enables an internal pull-up resistor on the pin, ensuring that the reset signal defaults to a logic high when not being driven.

The `set_input_delay` command specifies the input delay on the reset signal. Since reset signals does not require synchronization or specific timing constraints, a delay of zero is used.

`set_false_path` informs the timing analysis engine that the path of this reset signal should be ignored during the analysis. This is useful for paths that do not respect specific timing requirements, usually needed for the general logic. The reason is that reset signals do not need to have timing constraints. This helps avoiding unnecessary warnings that could be asserted by the tool.

Another module in Rosebud that requires an external clock source is the *startupe3* module. This a special-purpose primitive in Xilinx

FPGAs, used to interface the startup logic of the FPGA. The module manages signals like:

GSR (Global Set/Reset is used to reset all the flip-flops in the design, ensuring that during the start-up procedure, all registers are in a known state.

PROG is used to trigger the FPGA configuration procedure. This is useful in partial reconfiguration scenarios, in which certain parts of the logic needs to be reset, to allow the reconfiguration of the FPGA at runtime.

```
create_clock -period 20.000 -name cfg_mgt_refclk_1
[get_pins startupe3_inst/CFGMCLK]
```

This command sets up a 50 Mhz clock for the *CFGMCLK* port of the *startupe3* module.

3.2.2 Partial Reconfiguration setup

Projects that make use of Partial Reconfiguration must be converted, enabling Dynamic Function eXchange.

A simple command is necessary to convert the project:

```
set_property PR_FLOW 1 [current_project]
```

This operation is irreversible, meaning that PR cannot be disabled after using this command. To utilize PR, Vivado needs to know which modules are reconfigurable. This is accomplished by creating partition definitions, which associate specific modules to the designated partitions. Here is shown the code that allocates partitions for the RPU and the Load Balancer:

```
if{[llength [get_partition_defs pr_riscv]]==0}
  then {
    create_partition_def -name pr_riscv -module rpu_PR}
if{[llength [get_partition_defs pr_load_balancer]]==0}
  then {
    create_partition_def -name pr_load_balancer -module lb_PR
}
```

This code checks if the partition already exists and if it doesn't, it creates a new one. The `-module` argument is used to insert the name of a reconfigurable module, linking it to the partition definition.

Partitions are used to compile parts of the design separately, improving code modularity and reducing compile times. The definition of the partitions is done before logic synthesis.

3.2.3 Floorplanning

After the synthesis process, the netlist containing all the HDL components is generated, and a top-level hierarchy is created. In *DfX* projects, before starting the implementation process, it is necessary to define the physical location of the reconfigurable modules in the target FPGA device.

The floorplanning procedure consists in allocating reconfigurable partitions to specific regions of the FPGA. A proper placement is crucial because it ensures that the reconfigurable modules have sufficient resources in order to function properly.

During PR, reconfigurable modules are swapped in and out within specific regions of the FPGA. To load these modules into a partition, a configuration object must be created. Here is shown how a PR configuration object is created in Rosebud:

```
create_pr_configuration -name config_1 -partitions [list
    core_inst/rpus[0].rpu_PR_inst:RPU_base
    core_inst/rpus[1].rpu_PR_inst:RPU_base
    core_inst/rpus[2].rpu_PR_inst:RPU_base
    core_inst/rpus[3].rpu_PR_inst:RPU_base
    core_inst/rpus[4].rpu_PR_inst:RPU_base
    core_inst/rpus[5].rpu_PR_inst:RPU_base
    core_inst/rpus[6].rpu_PR_inst:RPU_base
    core_inst/rpus[7].rpu_PR_inst:RPU_base
    core_inst/lb_PR_inst:LB_Hash]
```

`core_inst` is the name of the FPGA design top-level instance. All modules are instantiated under this top-level module, with the exception

of the *CMAC* modules and some clock-related modules. The name of this PR configuration is specified using the `-name` argument. The `-partitions [list ...]` command defines the list of reconfigurable partitions, i.e., the modules that will be swapped in and out of the FPGA, dynamically.

The RPUs are are instantiated in an array, using the `for generate` statement in Verilog. Each RPU instance is a reconfigurable module, referred to as `rpu_PR_inst`. The last entry is the *Load Balancer*, which is also a reconfigurable module. However, the Load Balancer cannot be swapped dynamically. Because of its crucial role in the design, the rest of the logic cannot operate without the Load Balancer.

The optimal placement of the static logic can be determined by performing an implementation run, removing the reconfigurable logic temporarily. This allows to identify the optimal regions in which the static modules can be placed, ensuring an optimal usage of space.

The implemented design can be viewed in the Vivado GUI, selecting *"Open Implemented Design"* in the lateral menu. This opens an interactive graphical representation of the FPGA layout, showing the FPGA resources, such as *DSP blocks*, *URAM modules*, *Slices* and *Clock regions*. The different cells are addressed using *XY coordinates*, used for Pblock assignment in the TCL scripts.

Here is shown an example of creation of a Pblock, assigning slices (basic logic elements) to it:

```
create_pblock Pblock_1
resize_pblock Pblock_1 -add {SLICE_X13Y0:SLICE_X41Y119}
```

The `create_pblock` command generates a Pblock, the virtual container used to hold the logic of the modules that will be constrained to a specific region.

`resize_pblock` specifies the physical location in which the Pblock will be allocated and the type of resource that the module requires.

The implemented design figure can be leveraged to see the coordinates of the static logic modules placed by Vivado, by drawing Pblocks around them. When a Pblock is created, Vivado shows the coordinates of the Pblock in the "*Properties*" window.

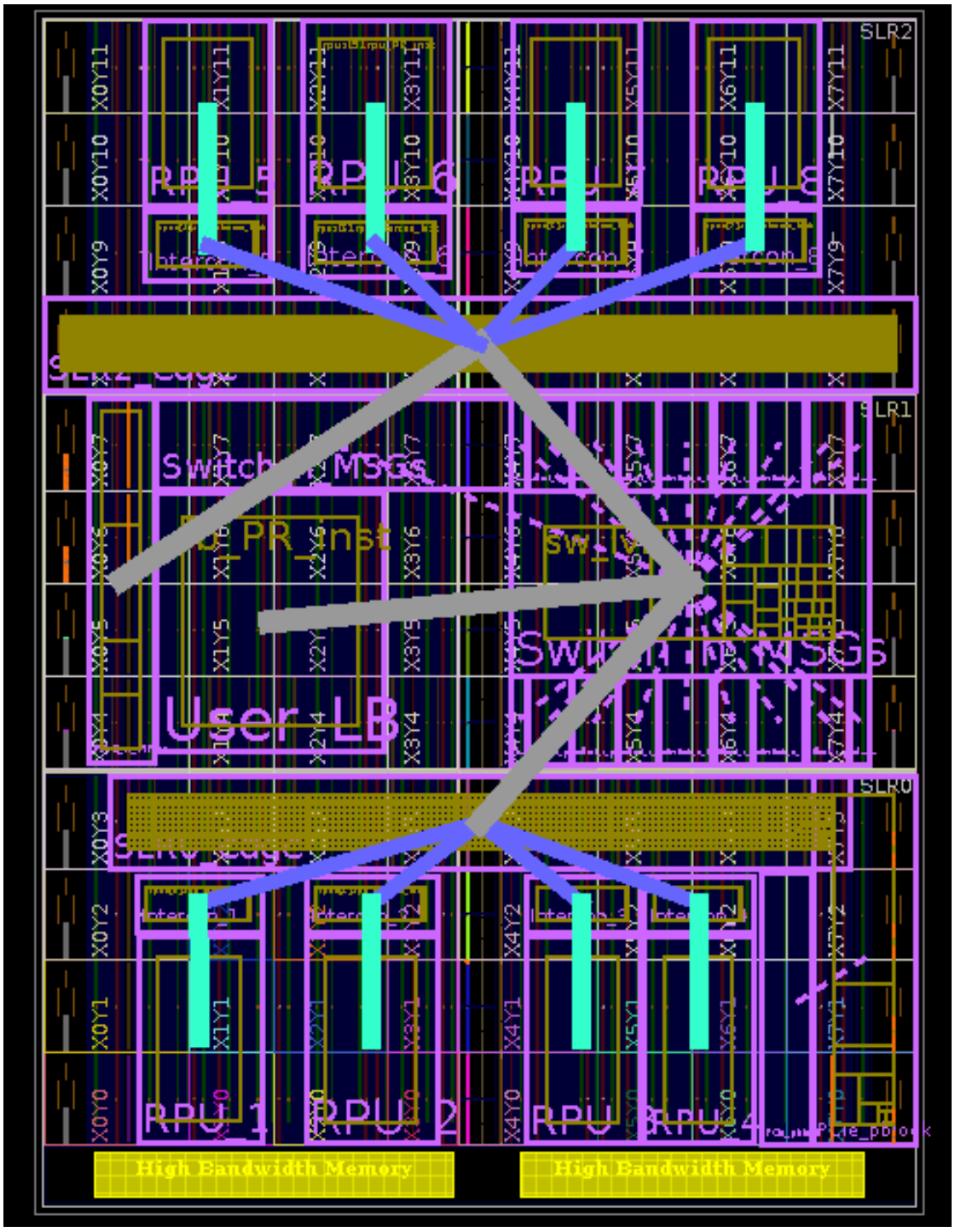

Figure 3.2: Highlight of Pblocks for each module

The placement of the reconfigurable modules is also constrained by the location of hard IP blocks, such as the CMAC and the PCIe module. These modules require specific I/O components that have fixed location on the FPGA, such as the *GTY transceiver banks*.

Structure of the Alveo U55C FPGA

The Alveo U55C FPGA is organized into three *Super Logic Regions (SLRs)*, named SLR0, SLR1 and SLR2. Each region is a separated FPGA die slice,containing an independent set of logic resources. The regions are interconnected in the device through *Stacked Silicon Interconnect* technology. The three regions are highlighted in Figure 3.2: SLR0 is the lower region, SLR1 is the middle region and SLR2 is the higher one.

Components Placement in the U55C

Figure 3.2 shows the entire FPGA device and all the allocated Pblocks.

The *CMAC* is placed on the left side of SLR1. The placement is constrained by the location of the GTY transceivers used for the ethernet module, highlighted by the orange vertical line in the figure.

The *PCIe module* is placed on SLR0, on the right side. The Rosebud implementation for the Alveo U200 manages to put all the static logic, CMAC and PCIe included, on the middle region, allowing the RPUs to be placed symmetrically on the other regions. In the U55C this is not possible because it has a smaller FPGA die and the GTY transceivers used by the PCIe IP are located on SLR0, instead of SLR1.

Another constraint is put on the FIFO modules used as buffers for the CMAC. The primitives used by the CMAC to work with the QSFP GTY transceivers are called `GTYE4_CHANNEL` and `GTYE4_COMMON`. Ultrascale devices require also to allocate the associated *BUFG_GT_SYNC* and *BUFG_GT* sites. All these components are part of what is defined as *Programmable Unit (PU)* in Xilinx documentation. The location of the GTY channels in the U55C is on the left side of SLR1, which means the *MAC_n_FIFOs* Pblock is placed there. Figure 3.3 shows the placement of all the remaining static logic. The location of the

remaining component is not constrained by specific criteria other than the required set of resources.

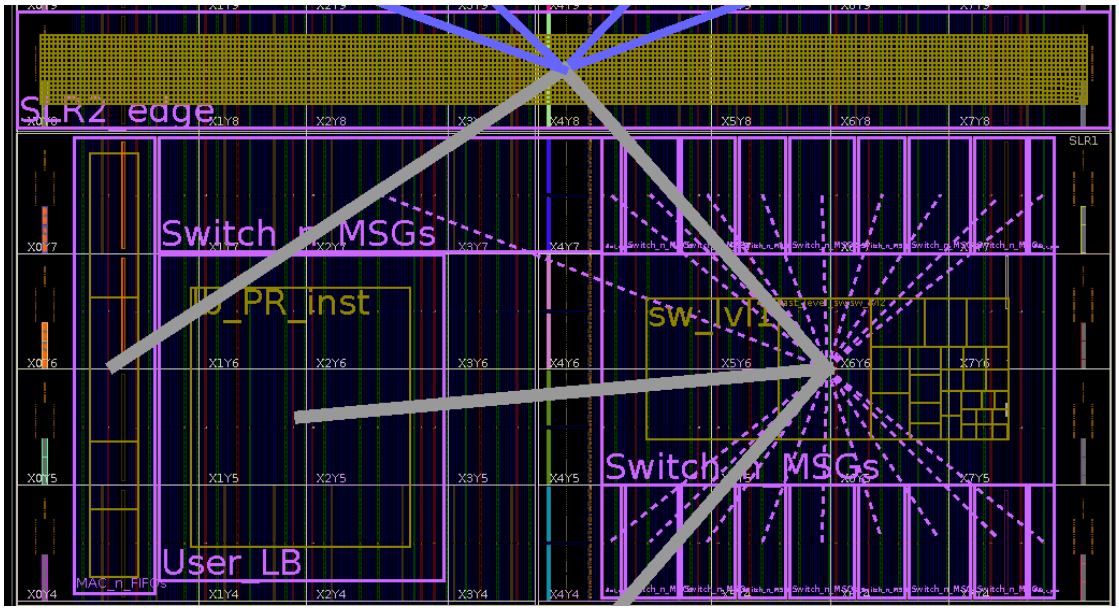

Figure 3.3: Closer view of the SLR1 region

The placement of the PCIe module forces a shrink on the RPU3 and RPU4 Pblocks, allowing less resources to be allocated to those RPUs. The number of resources allocated to the RPUs determines the size of the accelerators that can be allocated, which is already limited by the smaller device size of the Alveo U55C. However, no performance issues are found during the benchmark, as all the necessary resources needed for basic functionality are still allocated.

3.2.4 Second Generation 3D IC Interconnect

Figure 3.2 shows the connections between the RPUs and the static components of the design. Two sets of registers, highlighted by yellow horizontal bars towards the center of the figure, are placed on the edge of SLR regions. These two Pblocks called *SLR0_edge* and *SLR2_edge*, contain registers that facilitate interconnection between Pblocks located in different SLR regions.

As previously mentioned, each SLR region contains an independent set of resources, including the clock signal routing. When a design spans multiple SLR regions, Xilinx recommends the usage of dedicated flip-flops that are placed in the interconnects between two regions.

The cells containing these dedicated flip-flops are called *laguna* cells, each containing a TX flip-flop and a RX flip-flop. One of these Flip-flops can be used for a connection between regions. For instance, if the TX flip-flop (driver) is allocated for a Pblock in the SLR0 region, the receiver module must use the RX flip-flop in its allocated laguna cell.

The code below shows the allocation of the set of registers used in SLR0:

```
create_pblock SLR0_edge
add_cells_to_pblock [get_pblocks SLR0_edge]
resize_pblock [get_pblocks SLR0_edge] -add {LAGUNA_X0Y0:LAGUNA_X31Y117}
set_property EXCLUDE_PLACEMENT 1 [get_pblocks SLR0_edge]
set_property IS_SOFT FALSE [get_pblocks SLR0_edge]
```

The `EXCLUDE_PLACEMENT` is a placement constraint, that directs Vivado to prevent the placement of additional logic within the Pblock, to avoid resource allocation issues and conflicts with other logic elements. The `IS_SOFT` property set to *FALSE* indicates that the Pblock is a hard constraint. This means that the boundaries of the Pblock must be strictly respected during the placement process.

In Ultrascale+ devices, Xilinx facilitates the allocation of laguna resources, using the `CLOCKREGION` directive. Figure 3.4 shows how an SLR is divided in squares called *clockregions*, indicated by XY coordinates. If a Pblock borders are aligned with those of a clockregion, the whole square can be allocated, allowing Vivado to improve the clock placement of the design. Here is an example with the allocation of the *SLR2_edge* Pblock:

```
create_pblock SLR2_edge
resize_pblock [get_pblocks SLR2_edge] -add {CLOCKREGION_X0Y8
CLOCKREGION_X1Y8 CLOCKREGION_X2Y8
CLOCKREGION_X3Y8 CLOCKREGION_X4Y8 CLOCKREGION_X5Y8
CLOCKREGION_X6Y8 CLOCKREGION_X7Y8}
```

```
set_property EXCLUDE_PLACEMENT 1 [get_pblocks SLR2_edge]
set_property IS_SOFT FALSE [get_pblocks SLR2_edge]
```

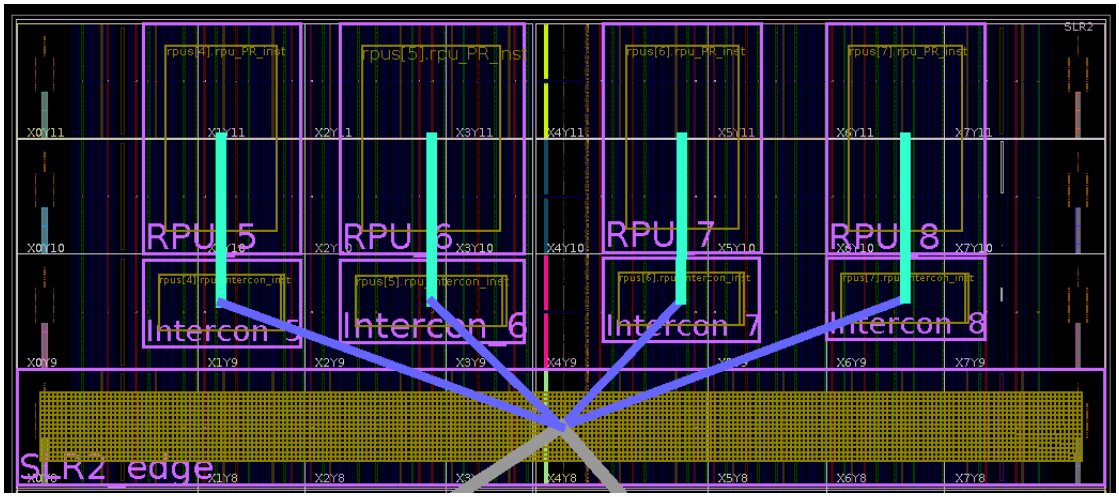

Figure 3.4: Closer view of the SLR2 region

The cells that belong to the Pblock are not shown for brevity. When a Pblock is created, the command `add_cells_to_pblock` must be set with the list of the module instances that are part of the module. Here is the set of commands used to create the Pblock for one RPU:

```
create_pblock RPU_1
add_cells_to_pblock [get_pblocks RPU_1]
[get_cells -quiet [list {core_inst/rpus[0].rpu_PR_inst}]]
resize_pblock [get_pblocks RPU_1] -add {SLICE_X15Y2:SLICE_X54Y135}
resize_pblock [get_pblocks RPU_1] -add {DSP48E2_X1Y0:DSP48E2_X6Y47}
resize_pblock [get_pblocks RPU_1] -add {RAMB18_X1Y2:RAMB18_X2Y53}
resize_pblock [get_pblocks RPU_1] -add {RAMB36_X1Y1:RAMB36_X2Y26}
resize_pblock [get_pblocks RPU_1] -add {URAM288_X0Y4:URAM288_X0Y35}
set_property SNAPPING_MODE ON [get_pblocks RPU_1]
```

Here, the argument of the `add_cells_to_pblock` command contains the instance of the RPU that is being allocated to the Pblock. The whole hierarchy path is written. Then, the required logic elements are added to the allocated resources. The RPUs require a set of DSP blocks, BRAM modules and URAM modules, which are high-latency

modules provided in Ultrascale FPGAs. The list of resource requirement is created by the Vivado synthesis tool, after the synthesis process, in which the RTL code is compiled and a more accurate netlist is generated.

Bitstream generation and design start-up

After the floorplan phases, it is crucial to perform *Design Rule Checks (DRCs)* to identify potential *critical warnings* that might prevent the completion of the implementation process. Following the implementation phase, a set of report is generated. These reports include crucial information about timing, resource utilization and area. The reports should be analyzed to verify whether the required timing constraints are satisfied in the post-implementation netlist. Once requirements are met, a bitstream is generated as a final procedure. In Vivado-based projects, the *hw_manager* utility is used to program the FPGA with the generated bitstream. *TCL* scripts are used to automate this process.

Bitstreams are loaded into the FPGA flash memory via the SPI interface, through micro-USB ports attached to the Alveo card. The average time for the flashing of a bitstream can take several seconds, depending on the size of the bitstream. After the bitstream is successfully loaded, the FPGA must be reset in order to run the new configuration.

The reset of a PCI slot is performed through the following command:

```
echo 1 > "/sys/bus/pci/devices/<dev_pci_address>/rescan"
```

This operation requires *sudo privileges*. The output of the *dmesg* command can be checked to see if there are errors in the flash procedure. Figure 3.5 shows *dmesg* output in case of success.

```
[831832.399245] pci 0000:02:00.0: Removing from iommu group 81
[831833.428692] pci 0000:02:00.0: [1234:1001] type 00 class 0x020000
[831833.428726] pci 0000:02:00.0: reg 0x10: [mem 0x00000000-0x00ffffff 64bit pref]
[831833.428773] pci 0000:02:00.0: Max Payload Size set to 256 (was 128, max 1024)
[831833.429138] pci 0000:02:00.0: Adding to iommu group 81
[831833.429705] pci 0000:02:00.0: BAR 0: assigned [mem 0xc2000000-0xc2ffffff 64bit pref]
```

Figure 3.5: Dmesg output after bitstream loading

At this point, the new bitstream is running on the FPGA. The status of the ethernet link is shown by the *activity leds* in the Alveo QSFP ports. A yellow or green light indicates that the physical link is functional and the CMAC IP core has established the link. In order to establish the link, the other end of the 100G network cable must be connected to a compatible device.

Core Programming

To execute software on the RISC-V cores, C programs are compiled using the *gcc* compiler for RISC-V architectures. A special set of instructions, the *rv32i_zicsr*, is required for the VexRisc-v implementation. Rosebud provides a C program, called `rvfw.c`, which initializes the *RPU memories* and the *Load Balancer* and loads the compiled binary of instructions into the cores.

Chapter 4

Performance Benchmark

Rosebud has been successfully implemented on the Xilinx Alveo U55C. The Alveo U55C card was deployed on a SuperMicro server, equipped with a Intel(R) Xeon(R) CPU E5-2623 v4, running at 2.60 Ghz. The U55C was connected with a point-to-point connection to a Mellanox ConnectX-4 adapter, running on another SuperMicro server, through a 100G DAC optical fiber cable.

The U55C was chosen as the platform for benchmarking the performance of Rosebud, due to the company's extensive experience with the card in other projects.

The benchmark focuses on specific parameters that are essential to investigate the integration of virtualization features on Rosebud, which acts as a networking platform for hardware accelerators.

4.1 Partial Reconfiguration

To evaluate the efficiency of PR, the configuration latency is measured. This latency represents the time required to configure the FPGA with the partial bitstreams. The time required to reset a core before the bitstream flashing is also considered part of the configuration time.

The FPGA is programmed with the partial bitstream using the Management Configuration Access Port (MCAP) interface, accessible through the PCIe slot. MCAP manages multiple configuration channels that can be used to load different partial bitstreams into the FPGA.

Each partial bitstream is associated with a reconfigurable module. The MCAP interface is accessible through a set of drivers provided by Xilinx.

The library `mcap_lib.h` offers a specific function that automates the programming of the partial bitstream:

```
MCapConfigureFPGA(mdev, bitfile, EMCAP_CONFIG_FILE);
```

In this function, `mdev` is the */dev* entry, *bitfile* is the partial bitstream file and `EMCAP_CONFIG_FILE` is a macro defined in the library header, that addresses a default configuration file.

To measure the time required for this operation, the C function `clock_gettime()`, available in Unix systems, offers high-resolution, providing time in nanoseconds. The function is called before and after the *MCAP* function. Then we can compute the difference between the two times to provide the latency of the operation.

```
clock_gettime(CLOCK_MONOTONIC, &start_time);
MCapConfigureFPGA(mdev, bitfile, EMCAP_CONFIG_FILE);
clock_gettime(CLOCK_MONOTONIC, &end_time);
```

Since the function loads one bitstream at a time, a loop is used to iterate the programming for each RPU.

As detailed in the *Partial Reconfiguration* section, the procedure is done at runtime, while the static logic on the FPGA still operates. The latency between the completion of partial reconfiguration and the RPUs starting operating time is negligible, as the cores begin operating immediately following the instruction load program execution. Consequently, the entire configuration procedure latency can be effectively regarded as the deployment time for the accelerator. Table 4.1 shows the average latency for the whole configuration procedure, including:

1. Putting a core in the reset state
2. Flashing the bitstream on the FPGA
3. Loading the instructions into the RISC-V CORE

RPUs	Latency (avg.)
1 RPU	02.34 s
4 RPUs	09.02 s
8 RPUs	16.50 s

Table 4.1: RPU reconfiguration latency

In comparison, virtual machines (VM) boot times tends to be in the order of seconds [9], with lightweight VMs having an average boot time between 10 and 30 seconds and commercial cloud VMs having boot times between 30 seconds and 1 minute. The deployment time for the hardware accelerators in Rosebud is shown to be in the order of seconds, starting from an average of 2.34 seconds for a single RPU. The test was conducted with different presets of accelerators provided in the Rosebud framework and with a new custom accelerator, shown in the next sections.

This results highlight the efficiency of Rosebud's reconfiguration procedure in a virtualized environment. While not negligible, especially for a high number of RPUs, the deployment time is shown to be in a practical range, ensuring the availability of the accelerator to VMs without significant delays. Thus, the minimal configuration time of the accelerator is highly advantageous and crucial to system performance.

4.2 Network performance

In order to measure packet forwarding performance, Rosebud was used as traffic generator, sending random packets over the ethernet interface. A C program running on the RISC-V cores is used for packet generation, with the RPUs working in parallel. The program is shown on appendix B. On the reception side, a Mellanox adapter captured the packets. The bandwidth was measured running a bash script on the server side, using the *tcpdump* utility. The script code is shown in Appendix D. First, *tcpdump* captures the packets in a *pcap* file. The file is read using the tshark utility, with the `-r` option and the `wc -l` command, which

counts the number of lines on the *tcpdump* output, corresponding to the total number of packets captured. The `awk` command is used to extract the timestamp of the first and the last packet. The value is converted from the *unix time format* to seconds and the difference is computed in order to find the duration of the test. The bandwidth is then computed using the number of captured packets, the packet size and the duration of the test, as shown in D.

Results revealed a heavy performance bottleneck, with an average measured bandwidth of 7 Gbps. The performance bottleneck is caused by the overhead on the linux kernel. The kernel performs a lot of context switch operations during the capture process. Adding up the number of dropped packets at the interface, the computation suggests an average bandwidth of $\sim$ 25Gbps. These values are lower than expected and indicate a saturation of the linux kernel interface, which is unable to handle the throughput. Unfortunately, this issue is common and showcased in several publications of 100G FPGA networking systems. The problem is usually bypassed using custom interfaces that provide a packet capturing agent at the FPGA interface.

A potential future work improvement to mitigate the performance bottleneck and allow the bandwidth measurements using standard linux tools, could be the integration of DPDK, which is a program designed to bypass the linux kernel and perform the capture operation directly in user space. For instance, the Xilinx's *OpenNIC* prototype provides a set of DPDK patched drivers compatible with the QDMA IP, achieving traffic generation at rates of 100 Gbps. The drivers provided by Rosebud could be improved in order to enable the usage of DPDK.

A second test was run using the internal statistics interface of Rosebud, which counts the number of bytes forwarded through the ethernet interfaces. The original approach used in Rosebud's publication and by several FPGA 100G networking prototypes is to adopt a dual-FPGA connection. Two FPGAs are connected to each other, one behaving as client and the other one as server. Using this approach, the performance issues in standard linux-based tools are bypassed. Given the availability of only one Alveo U55C for these measurements, the second test was run using a loopback approach, using half of the CPU cores to send

the packets through one QSFP port and the other half to receive them through the other QSFP port. The connection is shown in Figure 4.1.

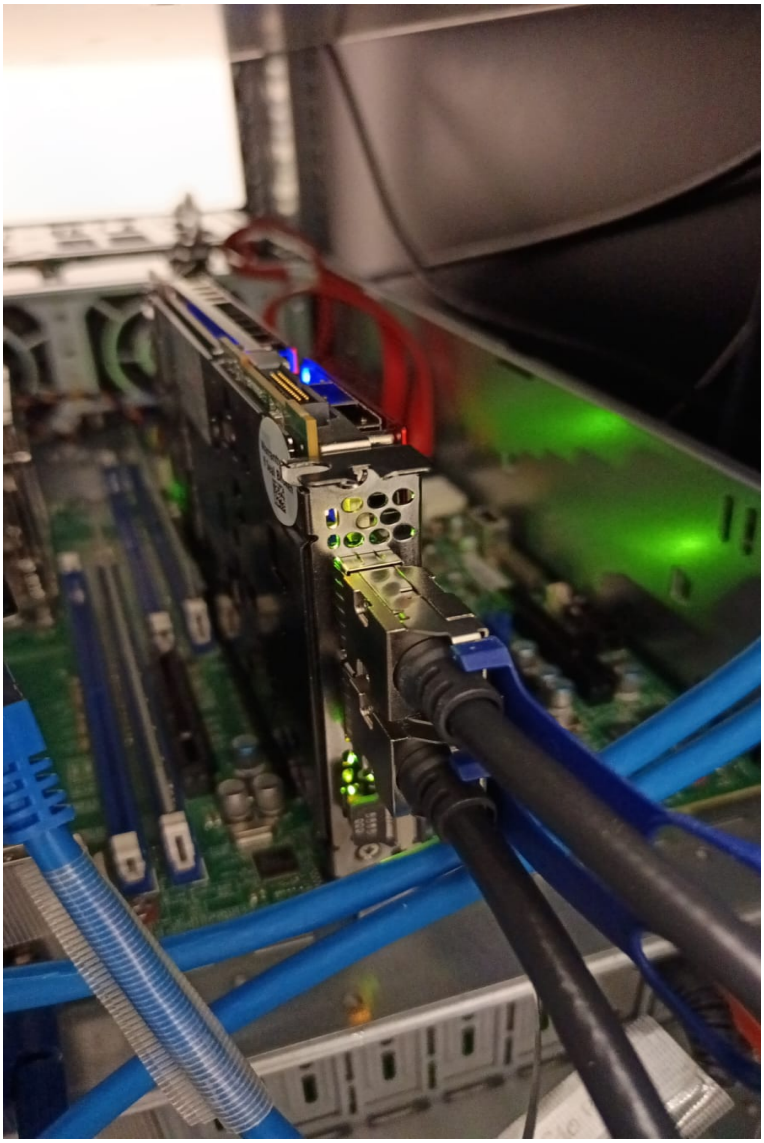

Figure 4.1: Alveo U55C connected to QSFP 100G cables

The test measures the transmitted bytes for a certain time period and then computes the bandwidth through a simple arithmetic conversion, in order to obtain the value in Gbps. The test was conducted measuring the bandwidth for different values of packet size, showing a logarithmic

increase with respect to the size of the packet. Figure 4.2 contains a graph that shows the forwarding rate for each packet size.

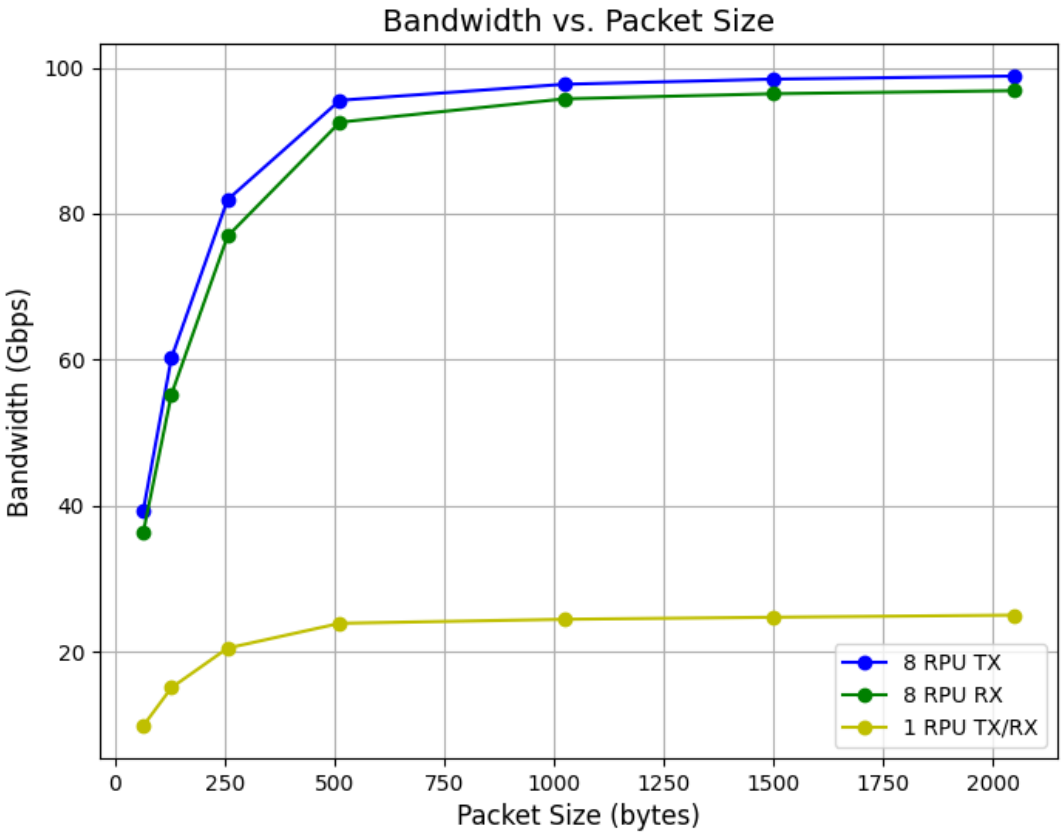

Figure 4.2: Packet forwarding 8 RPU vs. 1 RPU

4.2.1 Results

The results presented here were measured using a single QSFP port, with an expected peak bandwidth of 100 Gbps. The highest performance is achieved with packet size exceeding 1000 bytes. After this value, the bandwidth gradually reaches a saturation value near the peak throughput. The worst case is represented by 64-byte packets, achieving around 40% of the maximum rate for one port (100Gbps). The peak throughput reached by a single RPU is 25 Gbps, indicating that full bandwidth on a QSFP port can be achieved with half the RPUs used

for transmission and the other half for reception. This allows for peak performance to be reached with the loopback setup.

4.3 HW Accelerator Performance

A simple hardware accelerator was developed in order to test network performance in the presence of HW accelerators in the RPUs. The accelerator was written in Verilog and performs Cyclic Redundancy Check (CRC) computation, an operation commonly used at the link layer (level 2) of the OSI model. Rosebud works at the link layer by default, meaning that the CRC function is a suitable choice to quickly test network performance in the presence of an HW accelerator. The Verilog code for the accelerator is shown in Appendix C.

The accelerator accepts three input signals, `data_in`, `data_in_clear` and `data_in_valid`. Rosebud provides a wrapper for accelerators, abstracting the communication interface between the accelerators and the RPUs. This wrapper contains a process that reads memory addresses from the CPU cores and controls the *clear* and *valid* signals. The interface of the CRC accelerator was adapted to follow this protocol, allowing for a straightforward deployment into the RPUs.

The *compute_crc* function is the core of the accelerator. The CRC algorithm is implemented using a behavioral Verilog coding style. This design allows the synthesis tools flexibility to determine which hardware implementation to use. Consequently, the latency of the operation depends on the implementation choice of the tools. Given the combinatorial nature of the CRC algorithm, more specifically the parallelization capabilities of a for loop, it makes sense to assume that each byte of input is processed in one clock cycle. The latency introduced by the accelerators is expected to affect the network throughput, lowering the peak performance shown in the base design. Figure 4.3 shows network performance using the CRC accelerator. The test is done using the loopback setup shown in the previous section.

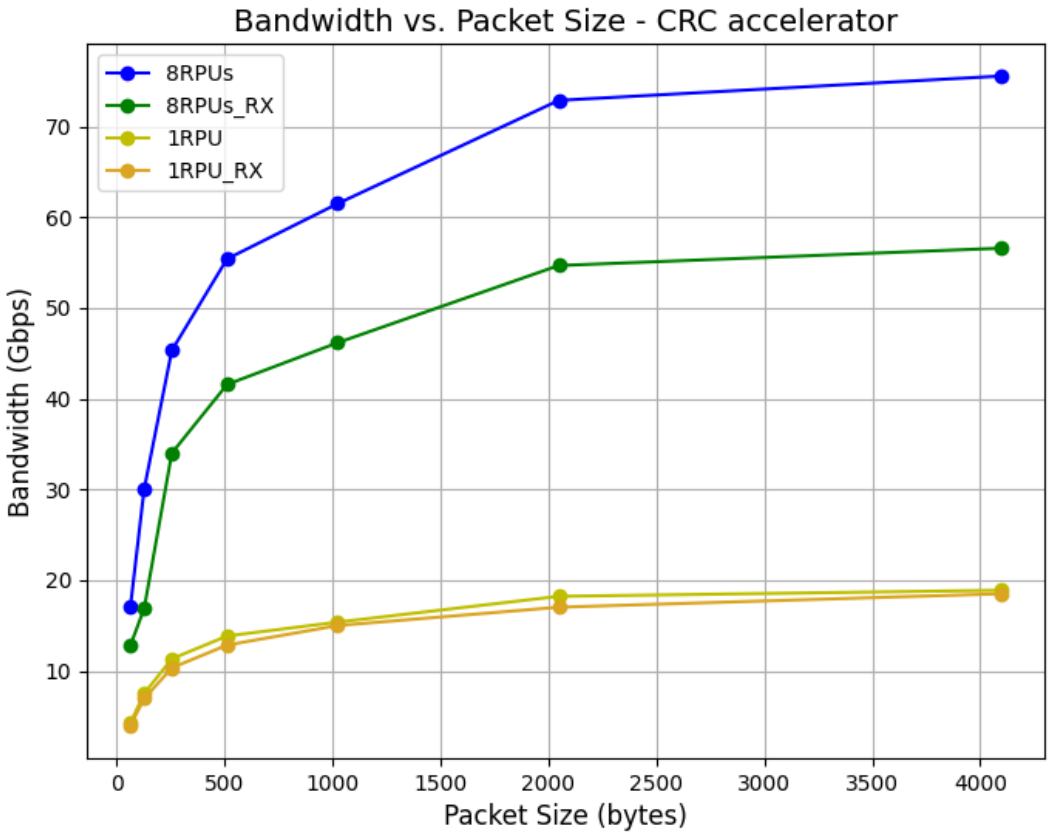

Figure 4.3: Packet forwarding w/ CRC - 8 RPU vs. 1 RPU

4.3.1 Results

Results show peak performance at 80 Gbps. Compared to the base forwarding test, the minimum packet size required to reach near peak-throughput is increased. This can be attributed to the added latency introducted by the hardware accelerator, which adds to the one of the RISC-V cores. In both configurations, using 8 RPUs or 1 RPU, saturation is reached for packets sizes of 2000 bytes or larger.

Chapter 5

Conclusions

This thesis focuses on Rosebud, a high-performance FPGA framework that provides both a high-speed network shell and reconfigurable processing units (RPUs) for hardware accelerators. Although Rosebud does not provide virtualization by default, the conducted analysis and evaluation in this research highlight the potential for implementing a multi-tenant virtualization design. As part of this work, a porting procedure was successfully performed, to add support for the *Alveo U55C* accelerator card, which was previously unsupported. This process involved the design synthesis for the U55C architecture and the floorplanning procedure, necessary to optimize the placement of the hardware components. The framework was then tested on the U55C, focusing on the efficiency of the Partial Reconfiguration feature, used to deploy hardware accelerators at run-time. Network performance was evaluated in a local network setting, measuring peak forwarding performance using the RISC-V CPU cores as traffic generators. Finally, a hardware accelerator was developed and deployed in order to assess its impact on the performance benchmarks. The results of these evaluations demonstrate the framework's potential in high-performance networking environments. In particular, Rosebud's architecture is demonstrated to be suitable for supporting virtualization in future implementations, enabling multi-tenant FPGA resource sharing and system efficiency.

5.1 Future Work

The presence of Xilinx *PCIe subsystem IP* offers an opportunity to enable *SR-IOV* technology, through a simple hardware configuration of the IP. Additionally, the reconfigurable units (RPUs) are suited for integration in a virtualized environment that provides access to resources through virtual functions. In the final stages of this research, a new solution called *RosebudVirt* was published, integrating virtualization capabilities into Rosebud. Although this product is closed-source, preventing the possibility of testing it within this research, the presented results align closely with the direction of this thesis, reinforcing the idea of a multi-tenancy virtualized implementation of Rosebud.

Appendix A

Synthesis Constraints

Here are shown the constraints required in order to perform synthesis of Rosebud on the Alveo U55C. The implementation constraints used in the floorplan stage are not shown, to avoid redundancy.

```
# QSFP28 Interfaces
set_property -dict {LOC AD51} [get_ports qsfp0_rx1_p]
set_property -dict {LOC AD46} [get_ports qsfp0_tx1_p]
set_property -dict {LOC AC53} [get_ports qsfp0_rx2_p]
set_property -dict {LOC AC44} [get_ports qsfp0_tx2_p]
set_property -dict {LOC AC49} [get_ports qsfp0_rx3_p]
set_property -dict {LOC AB46} [get_ports qsfp0_tx3_p]
set_property -dict {LOC AB51} [get_ports qsfp0_rx4_p]
set_property -dict {LOC AA48} [get_ports qsfp0_tx4_p]
set_property -dict {LOC AD42} [get_ports
   qsfp0_mgt_refclk_1_p]

# 161.1328125 MHz MGT reference clock (SI5394 OUT0)
#create_clock -period 6.206 -name qsfp0_mgt_refclk [
   get_ports qsfp0_mgt_refclk_p]

set_property -dict {LOC AA53} [get_ports qsfp1_rx1_p]
set_property -dict {LOC AA44} [get_ports qsfp1_tx1_p]
set_property -dict {LOC Y51} [get_ports qsfp1_rx2_p]
set_property -dict {LOC Y46} [get_ports qsfp1_tx2_p]
set_property -dict {LOC W53} [get_ports qsfp1_rx3_p]
set_property -dict {LOC W48} [get_ports qsfp1_tx3_p]
set_property -dict {LOC V51} [get_ports qsfp1_rx4_p]
set_property -dict {LOC W44} [get_ports qsfp1_tx4_p]
```

```
set_property -dict {LOC AB42} [get_ports
   qsfp1_mgt_refclk_1_p]

# 161.1328125 MHz MGT reference clock (SI5394 OUT1)
#create_clock -period 6.206 -name qsfp1_mgt_refclk [
   get_ports qsfp1_mgt_refclk_p]

# PCIe Interface
set_property -dict {LOC AL2} [get_ports {pcie_rx_p[0]}]
set_property -dict {LOC AL11} [get_ports {pcie_tx_p[0]}]
set_property -dict {LOC AM4} [get_ports {pcie_rx_p[1]}]
set_property -dict {LOC AM9} [get_ports {pcie_tx_p[1]}]
set_property -dict {LOC AN6} [get_ports {pcie_rx_p[2]}]
set_property -dict {LOC AN11} [get_ports {pcie_tx_p[2]}]
set_property -dict {LOC AN2} [get_ports {pcie_rx_p[3]}]
set_property -dict {LOC AP9} [get_ports {pcie_tx_p[3]}]
set_property -dict {LOC AP4} [get_ports {pcie_rx_p[4]}]
set_property -dict {LOC AR11} [get_ports {pcie_tx_p[4]}]
set_property -dict {LOC AR2} [get_ports {pcie_rx_p[5]}]
set_property -dict {LOC AR7} [get_ports {pcie_tx_p[5]}]
set_property -dict {LOC AT4} [get_ports {pcie_rx_p[6]}]
set_property -dict {LOC AT9} [get_ports {pcie_tx_p[6]}]
set_property -dict {LOC AU2} [get_ports {pcie_rx_p[7]}]
set_property -dict {LOC AU11} [get_ports {pcie_tx_p[7]}]
set_property -dict {LOC AV4} [get_ports {pcie_rx_p[8]}]
set_property -dict {LOC AU7} [get_ports {pcie_tx_p[8]}]
set_property -dict {LOC AW6} [get_ports {pcie_rx_p[9]}]
set_property -dict {LOC AV9} [get_ports {pcie_tx_p[9]}]
set_property -dict {LOC AW2} [get_ports {pcie_rx_p[10]}]
set_property -dict {LOC AW11} [get_ports {pcie_tx_p[10]}]
set_property -dict {LOC AY4} [get_ports {pcie_rx_p[11]}]
set_property -dict {LOC AY9} [get_ports {pcie_tx_p[11]}]
set_property -dict {LOC BA6} [get_ports {pcie_rx_p[12]}]
set_property -dict {LOC BA11} [get_ports {pcie_tx_p[12]}]
set_property -dict {LOC BA2} [get_ports {pcie_rx_p[13]}]
set_property -dict {LOC BB9} [get_ports {pcie_tx_p[13]}]
set_property -dict {LOC BB4} [get_ports {pcie_rx_p[14]}]
set_property -dict {LOC BC11} [get_ports {pcie_tx_p[14]}]
set_property -dict {LOC BC2} [get_ports {pcie_rx_p[15]}]
set_property -dict {LOC BC7} [get_ports {pcie_tx_p[15]}]
set_property -dict {LOC AR15} [get_ports pcie_refclk_p]
```

```
set_property PACKAGE_PIN BF41 [get_ports pcie_reset_n]
set_property IOSTANDARD LVCMOS18 [get_ports pcie_reset_n]
set_property PULLUP true [get_ports pcie_reset_n]

create_clock -period 10.000 -name pcie_mgt_refclk_1 [
   get_ports pcie_refclk_p]
create_clock -period 20.000 -name cfg_mgt_refclk_1 [
   get_pins startupe3_inst/CFGMCLK]

#Input and Output delays
set_input_delay 0.000 [get_ports pcie_reset_n]
set_false_path -from [get_ports pcie_reset_n]
```

Appendix B

Traffic generation

This code runs on the RISC-V cores on the RPUs. Packet generation and forwarding to ethernet interface is done by the cores.

```
#include "core.h"

#define PKT_SIZE 1024

struct Desc packet;
unsigned int *pkt_data[16];

int main(void){

  // initialization stuff for the load balancer
  init_hdr_slots(16, 0x804000, 128);
  init_slots(16, 0x00000A, 16384);
  set_masks(0x30);

  // this loop initializes the addresses for the packet's
   slots and writes the core_id to the packet headers
  for (size_t i=0;i<16;i++){
    pkt_data[i] = (unsigned int *)(0x01000000+i*16384);
    pkt_data[i][0] = core_id();
  }

  packet.len = PKT_SIZE;
  packet.tag = 0;
  if ((core_id()&0x4)!=0)
    packet.port = 0;
```

```
  else
    packet.port = 0;

  while (1){
    for (i=0;i<16;i++) {
      packet.data = (unsigned char *) pkt_data[i];
      pkt_send(&packet);
    }
  }

  return 1;
}
```

Appendix C

CRC Accelerator

```
    module crc_acc #(
    parameter CRC_WIDTH = 32,
    parameter SEED = 32'h04C11DB7
)
(
    input wire          clk,
    input wire          rst,

    input wire [31:0]   data_in,
    input wire [1:0]    data_in_len,
    input wire          data_in_valid,
    input wire          crc_clear,

    output wire [31:0]  crc_out
);

parameter OFFSET_WIDTH = $clog2(CRC_WIDTH);

reg [CRC_WIDTH-1:0] crc_reg = 32'hFFFFFFFF;
reg [OFFSET_WIDTH-1:0] offset_reg = 0;

function [31:0] compute_crc(input [31:0] data, input
   [31:0] current_crc, input [1:0] len);
    integer i, j;
    reg [31:0] crc;
    reg [7:0] data_byte;
    begin
```

```
        crc = current_crc;
        for (i=0; i<(len * 8);i=i+8) begin
            // Extract byte
            data_byte = data[i +: 8];
            crc = crc ^ (data_byte << 24);
            for (j=0;j<8;j=j+1) begin
                if (crc[31]) begin
                    crc = (crc << 1) ^ SEED;
                end else begin
                    crc = crc << 1;
                end
            end
        end
        compute_crc = crc;
    end
endfunction

wire [31:0] crc_next = compute_crc(data_in, crc_reg,
   data_in_len);

// output is inverted
assign crc_out = ~crc_reg;

always @(posedge clk) begin
    if (rst || crc_clear) begin
        crc_reg <= 32'hFFFFFFFF;
        offset_reg <= 0;
    end else if (data_in_valid) begin
        crc_reg <= crc_next;
        offset_reg <= offset_reg + data_in_len;
    end
end

endmodule
```

Appendix D

Server side benchmark

```
#!/bin/bash

interface=ens5np0
packet_size=1500
output_file=capture.pcap

timeout 10 sudo tcpdump 'len = 1500' -i ens5np0 -s 0 -w
   capture.pcap

# the next commands run tcpdump just to read from capture.
   pcap
captured_packets=$(tshark -r $output_file | wc -l)

last_timestamp=$(tcpdump -r $output_file -q | tail -1 |
   awk '{print $1}')
start_timestamp=$(tcpdump -r $output_file -q | head -1 |
   awk '{print $1}')
# convert timestamp in seconds
last_seconds=$(echo "$last_timestamp" | awk -F: '{ printf
   "%d\n", $1*3600 + $2*60 + $3 }')
start_seconds=$(echo "$start_timestamp" | awk -F: '{
   printf "%d\n", $1*3600 + $2*60 + $3 }')

duration=$((last_seconds - start_seconds))

bandwidth=$(echo "$captured_packets * $packet_size * 8 /
   $duration" | bc)
```

```
gbps=$(echo "$bandwidth / 1000000000" | bc)

echo "Bandwidth measured: $gbps Gbps per second"
```

Bibliography

[1] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. «A Survey on FPGA Virtualization». In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 2018, pp. 131–1317. DOI: 10.1109/FPL.2018.00031 (cit. on pp. 16, 21).

[2] Stefano Cirici, Michele Paolino, and Daniel Raho. «SVFF: An Automated Framework for SR-IOV Virtual Function Management in FPGA Accelerated Virtualized Environments». In: *2023 International Conference on Computer, Information and Telecommunication Systems (CITS)*. 2023, pp. 1–6. DOI: 10.1109/CITS58301.2023.10188786 (cit. on p. 21).

[3] Stuart Byma, J. Gregory Steffan, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. «FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack». In: *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. 2014, pp. 109–116. DOI: 10.1109/FCCM.2014.42 (cit. on p. 21).

[4] Jiansong Zhang, Yongqiang Xiong, Ningyi Xu, Ran Shu, Bojie Li, Peng Cheng, Guo Chen, and Thomas Moscibroda. «The Feniks FPGA Operating System for Cloud Computing». In: *Proceedings of the 8th Asia-Pacific Workshop on Systems*. APSys '17. Mumbai, India: Association for Computing Machinery, 2017. ISBN: 9781450351973. DOI: 10.1145/3124680.3124743. URL: https://doi.org/10.1145/3124680.3124743 (cit. on p. 22).

[5] Adrian M. Caulfield et al. «A cloud-scale acceleration architecture». In: *2016 49th Annual IEEE/ACM International Symposium on*

Microarchitecture (MICRO). 2016, pp. 1–13. DOI: 10.1109/MICRO.2016.7783710 (cit. on p. 22).

[6] Suhaib A Fahmy, Kizheppatt Vipin, and Shanker Shreejith. «Virtualized FPGA Accelerators for Efficient Cloud Computing». In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. 2015, pp. 430–435. DOI: 10.1109/CloudCom.2015.60 (cit. on p. 22).

[7] Esam El-Araby, Ivan Gonzalez, and Tarek El-Ghazawi. «Virtualizing and sharing reconfigurable resources in High-Performance Reconfigurable Computing systems». In: *2008 Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications*. 2008, pp. 1–8. DOI: 10.1109/HPRCTA.2008.4745683 (cit. on p. 22).

[8] Yiwei Chang and Zhichuan Guo. «RosebudVirt: A High-Performance and Partially Reconfigurable FPGA Virtualization Framework for Multitenant Networks». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2024), pp. 1–5. DOI: 10.1109/TVLSI.2024.3436017 (cit. on p. 22).

[9] Ridlo Auliya, Yen-Lin Lee, Chia-Ching Chen, Deron Liang, and Wei-Jen Wang. «Analysis and prediction of virtual machine boot time on virtualized computing environments». In: *Journal of Cloud Computing* 13 (Apr. 2024). DOI: 10.1186/s13677-024-00646-4 (cit. on p. 43).