SLR BASED PARTITIONING ON MULTI-DIE FPGA CHIPS

Master's Thesis

Batuhan BULUT

Eskişehir 2025

SLR BASED PARTITIONING ON MULTI-DIE FPGA CHIPS

Batuhan BULUT

Master's Thesis

Department of Electrical and Electronics Engineering

Programme in Circuits and Systems Theory

Supervisor: Prof. Dr. Atakan DOĞAN

Eskişehir

Eskişehir Technical University

Institute of Graduate Programs

April 2025

FINAL APPROVAL FOR THESIS

This thesis titled SLR BASED PARTITIONING ON MULTI-DIE FPGA CHIPS has been prepared and submitted by Batuhan BULUT in partial fulfillment of the requirements in “Eskişehir Technical University Directive on Graduate Education and Examination” for the Degree of Master's in Electrical and Electronics Engineering Department has been examined and approved on 16/04/2025.

Committee Members		**Title, Name and Surname**	**Signature**
Member	:	Prof. Dr. Atakan DOĞAN	
Member	:	Prof. Dr. Gürhan KÜÇÜK	
Member	:	Dr. Öğr. Üyesi Burak BATMAZ	

Prof. Dr. Harun BÖCÜK
Director of the Institute of Graduate Programs

16/04/2025

SUPERVISOR APPROVAL

Master's student Batuhan BULUT, whom I supervise, has completed this thesis titled SLR BASED PARTITIONING ON MULTI-DIE FPGA CHIPS. According to my inspections, the work is scientifically and ethically appropriate for the student to the thesis defense exam.

Supervisor

Prof. Dr. Atakan DOĞAN

ABSTRACT

SLR BASED PARTITIONING ON MULTI-DIE FPGA CHIPS

Batuhan BULUT

Department of Electrical and Electronics Engineering

Programme in Circuits and Systems Theory

Eskişehir Technical University, Institute of Graduate Programs, April 2025

Supervisor: Prof. Dr. Atakan DOĞAN

In recent years, transformative advancements in hardware acceleration technologies, particularly Field-Programmable Gate Arrays (FPGAs), have significantly enhanced the performance of compute-intensive applications in artificial intelligence (AI), machine learning (ML), big data analytics, and high-performance computing (HPC). However, as the scale and complexity of FPGA-based systems increase, challenges such as inter-chip communication, resource partitioning, and efficient utilization of multi-die FPGAs have become critical issues.

This thesis addresses these challenges by enhancing a state-of-the-art High-Level Synthesis (HLS) compiler to support Super Logic Region (SLR) -based partitioning for multi-die FPGA chips. Key contributions include the development of a partitioning framework tailored to SLR constraints. Additionally, efficient resource utilization techniques, such as automated pipelining and bus-width conversion, are proposed to overcome timing and routing bottlenecks in multi-die architectures.

The proposed methodologies are validated on Xilinx Virtex UltraScale+ FPGAs. This study contributes to advancing FPGA partitioning techniques and presents a comprehensive framework for optimizing hardware acceleration in multi-die architectures.

Keywords: FPGA, Multi-die FPGA, SLR-based partitioning, HLS compiler, Hardware acceleration.

ÖZET

ÇOK KALIPLI FPGA YONGALARI İÇİN SLR TABANLI BÖLÜMLEME

Batuhan BULUT

Elektrik-Elektronik Mühendisliği Anabilim Dalı

Devreler ve Sistemler Teorisi Bilim Dalı

Eskişehir Teknik Üniversitesi, Lisansüstü Eğitim Enstitüsü, Nisan 2025

Danışman: Prof. Dr. Atakan DOĞAN

Son yıllarda, donanım hızlandırma teknolojilerindeki dönüştürücü gelişmeler, özellikle Alan Programlanabilir Kapı Dizileri (FPGA'lar), yapay zeka (AI), makine öğrenimi (ML), büyük veri analitiği ve yüksek performanslı bilgi işlem (HPC) alanlarındaki hesaplama yoğunluklu uygulamaların performansını önemli ölçüde artırmıştır. Ancak, FPGA tabanlı sistemlerin ölçeği ve karmaşıklığı arttıkça, çipler arası iletişim, kaynak bölümlendirme ve çoklu kalıp FPGA'larının verimli kullanımı gibi zorluklar ortaya çıkmıştır.

Bu tez, çoklu kalıp FPGA çipleri için SLR tabanlı bölümlendirmeyi desteklemek üzere Yüksek Seviyeli Sentez (HLS) derleyicisini geliştirerek bu zorlukları ele almaktadır. Önemli katkılar arasında Süper Mantık Bölgesi (SLR) kısıtlamalarına göre uyarlanmış bir bölümlendirme algoritmasının geliştirilmesi yer almaktadır. Ek olarak, otomatik boru hattı ve veri yolu genişliği dönüşümü gibi verimli kaynak kullanım teknikleri kullanılarak, çoklu kalıp mimarilerindeki zamanlama ve yönlendirme darboğazlarının üstesinden gelmek hedeflenmiştir.

Önerilen metodolojiler Xilinx Virtex UltraScale+ FPGA'larda doğrulanmıştır. Bu çalışma FPGA bölümleme tekniklerinin ilerlemesine katkıda bulunmaktadır ve çoklu kalıp mimarilerinde donanım hızlandırmayı optimize etmeyi hedefler.

Anahtar Sözcükler: FPGA, Çoklu kalıp FPGA, SLR tabanlı bölümlendirme, HLS derleyicisi, Donanım hızlandırma.

ACKNOWLEDGEMENTS

Firstly, I would like to express my deepest gratitude to my esteemed advisor, Prof. Dr. Atakan DOĞAN, whose knowledge and expertise have significantly guided my work. Throughout my academic journey, his unwavering support, patient approach, and mentorship have been invaluable. His guidance has played a crucial role in the successful completion of this work.

Additionally, I would like to extend my sincere thanks to Dr. Kemal EBCİOĞLU for his vast technical knowledge and tireless guidance. The way he approaches technical challenges and the perspective he has imparted to me hold immeasurable value. Without their invaluable guidance, completing this study would not have been possible.

Lastly, I wish to express my heartfelt gratitude to my family, who have been by my side throughout my life. Their unconditional support and unwavering presence have shaped my future. Without their love and encouragement, achieving this would not have been possible.

Batuhan BULUT

16/04/2025

STATEMENT OF COMPLIANCE WITH ETHICAL PRINCIPLES AND RULES

I hereby truthfully declare that this thesis is an original work prepared by me; that I have behaved in accordance with the scientific ethical principles and rules throughout the stages of preparation, data collection, analysis and presentation of my work; that I have cited the sources of all the data and information that could be obtained within the scope of this study, and included these sources in the references section; and that this study has been scanned for plagiarism with "scientific plagiarism detection program" used by Eskişehir Technical University, and that "it does not have any plagiarism" whatsoever. I also declare that, if a case contrary to my declaration is detected in my work at any time, I hereby express my consent to all the ethical and legal consequences that are involved.

Batuhan BULUT

CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

LIST OF ALGORITHMS

GLOSSARY OF SYMBOLS AND ABBREVIATIONS

FPGA	:	Field Programmable Gate Array
AI	:	Artificial Intelligence
ML	:	Machine Learning
HPC	:	High Performance Computing
HLS	:	High-Level Synthesis
SLR	:	Super Logic Region
RTL	:	Register Transfer Level
AWS	:	Amazon Web Services
SSI	:	Stacked Silicon Interconnect
SLL	:	Super Long Lines
MHz	:	Megahertz
UDP	:	User Datagram Protocol
I/O	:	Input/Output
IP	:	Intellectual Property
LUT	:	LookUp Table
DSP	:	Digital Signal Processing
RAM	:	Random Access Memory
BRAM	:	Block Random Access Memory
ASIC	:	Application Specific Integrated Circuit
PCIe	:	Peripheral Component Interconnect Express
AXI	:	Advanced eXtensible Interface
H2C	:	Host-to-Card
C2H	:	Card-to-Host
SH	:	Shell
DCP	:	Design Check Point
CLB	:	Configurable Logic Block
RISC	:	Reduced Instruction Set Computer
AMI	:	Amazon Machine Image
L1	:	Level 1
L2	:	Level 2
MAC	:	Media Access Address

CPU	:	Central Processing Unit
DPDK	:	Data Plane Development Kit
VPC	:	Virtual Private Cloud
DDR	:	Double Data Rate
CL	:	Custom Logic
XDC	:	Xilinx Design Constrains
VM	:	Virtual Machine
PCIM	:	PCIe Master Interface
PCIS	:	PCIe Slave Interface
SDE	:	Streaming Data Engine
GCD	:	Greatest Common Divisor
ADPCM	:	Adaptive Differential Pulse Code Modulation
AES	:	The Advanced Encryption Standard
ENC	:	Encryption
DEC	:	Decryption

1. INTRODUCTION

This chapter introduces the motivation, contributions and organization of the thesis.

1.1. Motivation

High-Level Synthesis (HLS) has emerged as a transformative methodology in digital design, enabling hardware developers to describe complex, computation-intensive applications using high-level programming languages such as C, C++. This design methodology significantly accelerates the design cycle, promotes rapid prototyping, and lowers the barrier to hardware development particularly benefiting software-oriented engineers entering the FPGA domain. By automating the translation from algorithmic representation to register-transfer level (RTL) code, HLS improves productivity compared to traditional RTL-based workflows.

As the demand for scalable, high-performance computing continues to grow, major cloud service providers including Microsoft Azure (http-1), Amazon Web Services (AWS) (http-2), and Alibaba Cloud (http-3) have integrated FPGA-based acceleration into their infrastructures. These platforms allow users to deploy custom hardware logic in the cloud to accelerate a wide range of workloads, from artificial intelligence and machine learning to high-frequency trading and data analytics. To support compute-intensive applications, these providers commonly use advanced multi-die FPGA devices such as the Xilinx Virtex UltraScale+ VU9P.

Multi-die FPGAs comprise multiple Super Logic Regions (SLRs), each functioning as an independent resource island within the FPGA fabric. These SLRs are key components in modern large-scale FPGA architectures, providing enhanced scalability and supporting more complex designs by distributing logic, memory, and interconnect resources across multiple dies. Modern Xilinx devices implement this architecture using Stacked Silicon Interconnect (SSI) technology (http-4), which facilitates communication between SLRs through an interposer.

While this structure offers substantial capacity and flexibility, it also creates significant design complexity, particularly in partitioning, routing, and meeting timing closure when signals traverse SLR boundaries. These issues become more pronounced when employing HLS tools. Although HLS improves productivity and is well-suited for smaller, localized designs, it often underperforms in large-scale implementations

compared to handcrafted RTL designs. A key limitation lies in the modelling and optimization of interconnect delays, an essential factor for timing closure which HLS tools struggle to accurately capture at high levels of abstraction. In large multi-SLR designs, the complexity of physical placement and routing increases, often resulting in degraded performance, lower achievable frequencies, and elevated routing congestion, especially when physical layout constraints are not explicitly considered during synthesis. In the following, a more detailed examination of these challenges and example solutions from the literature are presented.

Routing congestion in large-scale accelerator designs: While multi-die FPGA devices offer a viable solution for implementing large-scale accelerators by providing expanded resources and scalability, routing congestion remains a critical challenge in such designs. As designs grow in size and complexity, the likelihood of congestion increases, especially in highly utilized regions. This issue is highlighted in Xilinx's Vivado Answer Record 66314, which states: “One of the main reasons for congestion is the utilization of a design. Designs that have high utilization will be more susceptible to congestion. Generally, a design that is over 80% utilized (Slice LUTs) will become very difficult to route and meet timing” (http-5). In line with this observation, in [1], the congestion in large-scale designs mapped to multi-die FPGAs is tackled, in which an iterative partitioning approach based on a greedy heuristic that aims to reduce routing complexity and improve timing closure is employed.

Partitioning complex designs into SLRs: Multi-die FPGAs comprise multiple SLRs, each functioning as an independent logic region. These SLRs are interconnected via Super Long Lines (SLLs), which introduce notable latency when signals cross die boundaries. In [2], it was claimed that SLLs between dies cause approximately 1 ns of delay, while typical medium-length routing wires within a single die exhibit 4× to 8× shorter delay under the same manufacturing process. Therefore, effective partitioning of RTL designs across SLRs is critical to minimize unnecessary use of SLLs and to reduce inter-SLR communication delays. In other words, unbalanced utilization leads to excessive inter-SLR communication and degrades overall design performance (http-5). For example, [3] highlights excessive SLL usage in multi-die FPGAs, reporting an average reduction of 43.08% in SLL counts through optimized design strategies. Similarly, [4] investigates partitioning designs into smaller components aligned with SLR-based methodologies, achieving an average 50% reduction in inter-die connections.

SLR crossing timing requirements: The UltraFast Design Methodology Guide for Xilinx FPGAs and SoCs highlights the critical importance of properly managing design constraints, particularly for implementations that span multiple SLRs. In multi-die FPGAs, SLRs are interconnected through SSI, and the wires that cross these regions referred to as SLLs that introduce additional delays. The guide addresses this issue and recommends inserting auto pipeline registers between wires that traverse SLR boundaries to help mitigate timing degradation (http-6). As described in [5], to meet timing requirements in such scenarios, a hierarchical design structure is introduced to assist HLS tools in more easily recognizing the physical layout and applying pipelining to long wires, especially those that span across die boundaries.

In the presence of these challenges, it is evident that large-scale design implementation on multi-die FPGAs must account for the device's SLR structure and hierarchy. Unlike monolithic FPGAs, multi-die architectures offer significantly greater capacity but at the cost of added complexity in interconnect management. To address these concerns, numerous studies [6–10] have explored SLR-aware partitioning methodologies. In [5], it was demonstrated that SLR-based partitioning not only improves routability but also significantly enhances performance: in tests across 43 design configurations, the average operating frequency increased from 147 Megahertz (MHz) to 297 MHz, which is a 102% improvement with no loss in throughput and negligible changes in resource usage. Additionally, 16 previously unroutable designs were successfully routed using this partitioning approach.

This thesis builds upon a cutting-edge commercial compiler [11, 12], aiming for enhancing the built-in partitioning methodology of this compiler by introducing SLR-based partitioning to achieve more robust and effective partitioned designs on multi-die FPGAs, addressing the challenges outlined above. The compiler can transform an ordinary sequential C/C++ application into multi-chip hardware accelerator system for FPGA-based cloud deployments. One of the compiler’s key capabilities is designing multi-chip, supercomputer-scale accelerators where multi-chip partitions are distributed across individual FPGAs. While the design process is SLR-aware through auto-pipelining of registers**,** it does not currently provide a way of partitioning resources within each FPGA based on SLR-specific considerations.

1.2. Contributions

Improved Design Hierarchy: A new design hierarchy is implemented for the RTL generation stage of the HLS compiler, which is explained in detail in Section 4.4. Initially, the compiler generates a separate top module for each FPGA for a multi-FPGA system. However, with the integration of the SLR-based partitioning methodology, the compiler now generates a hierarchical structure consisting of two levels: one top module per SLR and an additional top module per FPGA to coordinate and connect the SLR-level modules. This hierarchical refinement enables more accurate physical mapping and improves support for multi-die FPGA architecture.

Enhanced Networking Mechanism: Previously, a network element determines whether to accept or forward an incoming packet based solely on the destination partition number. If the destination partition number matches its own, the packet is accepted and routed internally. Otherwise, the packet is sent externally via a UDP transaction through the input/output (I/O) controller to the destination FPGA, where it will enter through the destination network element. However, with the introduction of SLR-based partitioning and hierarchical design, multiple partitions can now coexist within the same FPGA chip. As a result, network elements must now also consider the physical placement of partitions. Even if the destination partition differs, if it resides on the same FPGA device, the packet should be internally routed without invoking an external UDP transmission. The updated networking mechanism implements this two-level process: if the destination partition is local (same FPGA), the packet is internally routed; otherwise, it is sent externally to reach the appropriate device in a multi-FPGA system.

Design Elements to Support SLR-based Partitioning: In the original FPGA-based partitioning methodology, the compiler generates an I/O Controller intellectual property (IP) to manage communication between different partitions typically between different FPGAs. The architecture and functionality of this I/O Controller IP are detailed in Section 4.2. Following the integration of SLR-based partitioning, a simplified version of this controller is introduced to manage communication between neighbouring SLRs within a single FPGA. This simplified version, referred to as the simple I/O controller, does not support UDP or external packet-based communication. Instead, it establishes direct connections between neighbouring SLRs and is detailed in Section 4.2.1.

Additional Optimizations: It is a fact that SLR-crossing can lead to performance degradation due to increased latency and limited routing resources. Specifically, SLLs

used to connect SLRs are finite and must be managed carefully. To address these challenges, the simple I/O controller IP, which handles communication between neighbouring SLRs, has been optimized to include auto-pipeline registers that help preserve timing closure. Section 4.2.1 also highlights the limited availability of SLLs. When the number of SLLs required by a simple I/O controller exceeds the available capacity, the compiler can optionally insert a Bus Width Converter IP that helps reduce the number of SLLs and prevents resource exhaustion.

As a result of the contributions presented in this thesis, a configurable SLR-based partitioning methodology has been integrated into the HLS compiler. This enhancement enables users to define the SLR configuration of the target FPGA architecture using a pragma directive such as --SLRs-per-chip="(x,y,z,k)", where each dimension represents the number of SLRs in the horizontal, vertical, and stacked directions, respectively. Based on this specification, the compiler automatically applies the proposed partitioning strategy, hierarchical design structure, networking mechanism, and supporting hardware components, such as the simple I/O controller, and creates scalable design generation for multi-die FPGA architectures.

1.3. Organization

This chapter explains the organization of the thesis. It outlines the content and focuses on each chapter.

Chapter 2 presents the foundational background necessary for understanding the challenges and motivations behind this work. It introduces key concepts such as FPGA architecture, the structure and significance of multi-die FPGAs, and the role of SLRs. Additionally, it provides an overview of HLS tools and their relevance in modern hardware design.

Chapter 3 describes the internal structure and operation of the HLS compiler used in this study. It explains the overall compilation flow, details the baseline FPGA-based partitioning strategy, and introduces core architectural components such as I/O controllers, and network communication. This chapter establishes the groundwork for the enhancements proposed in the later sections by providing a clear picture of how the compiler originally manages design partitioning across multiple FPGAs.

Chapter 4 introduces the proposed SLR-based partitioning methodology, which is the central contribution of this thesis. It begins with a discussion of the limitations of the

traditional FPGA-based partitioning approach, followed by a detailed explanation of how the design is now partitioned according to the SLR hierarchy within a single FPGA device. The chapter elaborates on the new compiler enhancements, including updated I/O controller structures, hierarchical top-module generation strategy, an optimized networking mechanism for intra-chip communication, and the use of pipeline and bus width converter IPs to address timing and resource constraints. The chapter also outlines how a user-configurable pragma interface allows scalable deployment of designs across one, two or three-dimensional SLR arrangements.

Chapter 5 presents the experimental validation of the proposed methodology. It includes functional simulation results using the Verilator tool and implementation outcomes on Xilinx VU9P FPGAs within the AWS F1 environment.

Chapter 6 concludes the thesis and outlines potential directions for future research and improvement.

2. BACKGROUND

This chapter introduces the key concepts, technologies, and definitions used in this thesis.

2.1. FPGA Architecture

FPGAs are reconfigurable integrated circuits, enabling them to perform a wide variety of hardware functions. The basic structure of FPGAs is made of the following components: look-up tables or LUTs, flip-flops, programmable interconnects (wires) and I/O pads. An illustration of these resources can be found in Figure 2.1.

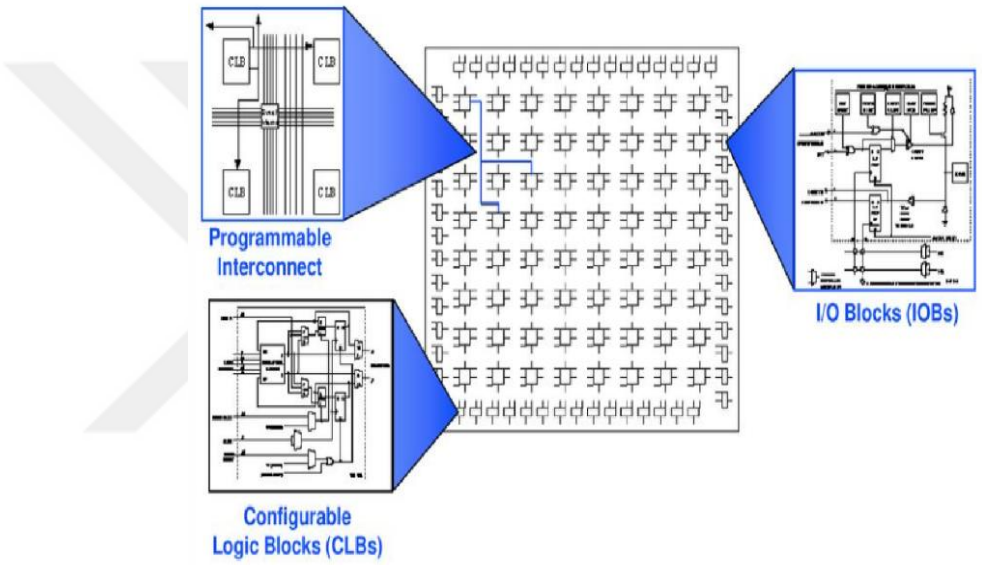

Figure 2.1: *FPGA resource illustration (http-7)*

In addition to the programmable soft logic elements, FPGAs also feature hardened blocks. Examples include Digital Signal Processing (DSP) blocks for arithmetic operations and Block Random Access Memory (BRAM) blocks for data storage. The interconnection between these elements is achieved through a network of programmable interconnects, which are responsible for routing data in the FPGA. These interconnects include horizontal and vertical wires that connect the soft and hardened logic blocks to form the desired circuit functionality. This blend of reconfigurability and specialized resources makes FPGAs powerful tools for implementing complex digital systems for high-performance computing applications.

2.2. Multi-Die FPGAs

Multi-die FPGA is a semiconductor device designed to integrate multiple FPGA dies, or SLRs, within a single device. These devices utilize cutting-edge SSI to deliver higher logic density, improved scalability, and improved performance compared to traditional monolithic FPGA architectures. Each die operates as a semi-independent region, connected through high-bandwidth interconnects that provide communication between them. This architecture allows multi-die FPGAs to address the growing demand for compute-intensive applications, such as machine learning, data analytic analytics, and high-performance computing, while solving the limitations of die size, yield, and power efficiency in monolithic designs. An illustration of a multi-die FPGA can be found in Figure 2.2. The partitioning of logic across multiple dies introduces challenges in design and implementation, such as resource utilization across dies and balancing those resources, achieving timing closure, managing inter-die communication latency. To address these challenges, FPGA design tools provide features like floorplanning and inter-die routing optimization. Multi-die FPGAs also benefit from advanced packaging technologies like silicon interposers or embedded bridges, which offer dense and low-latency connections between dies. These capabilities make multi-die FPGAs an ideal solution for applications requiring enormous parallelism and massive resource utilization.

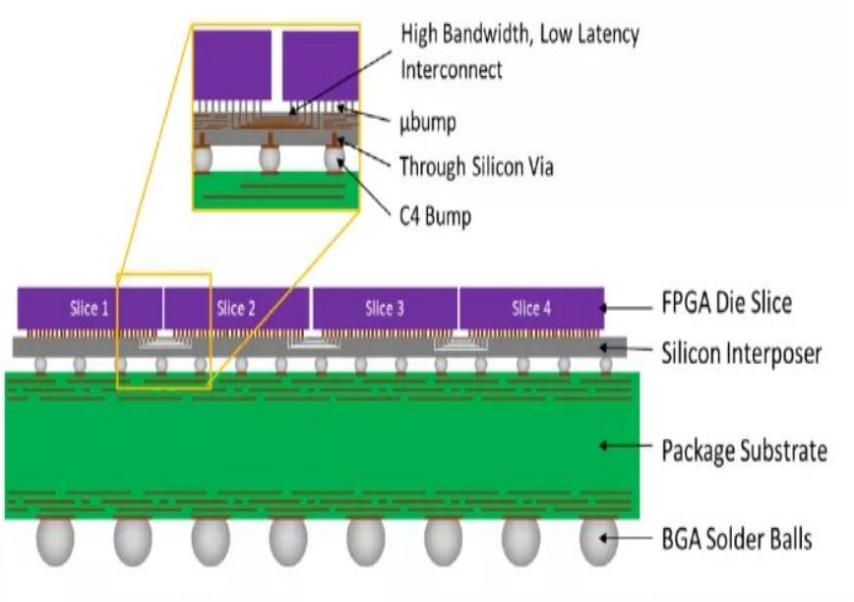

Figure 2.2: *A connection illustration of a multi-die FPGA device (http-8)*

2.3. Multi-Die FPGA Design Challenges

Designing hardware for an FPGA chip involves addressing a wide range of critical considerations, especially when targeting advanced platforms such as Xilinx UltraScale+ multi-die FPGAs. However, these benefits come with increased design complexity, particularly in areas such as timing closure, routing optimization, and balanced resource utilization. This section outlines the major design and implementation challenges associated with multi-die FPGAs.

2.3.1. Resource management

Timing closure remains one of the most critical and challenging aspects of FPGA design, particularly in architectures that span multiple dies. Multi-die FPGA introduce significant complexity due to increased interconnect distances, multiple clock domains, and added latency across SLR boundaries. Achieving reliable timing closure in such systems requires a combination of detailed design constraint management, intelligent floorplanning, and advanced timing optimization techniques.

The physical separation between SLRs leads to longer signal paths and greater propagation delays. These delays, if not accounted for during early design stages, can cause timing violations that are difficult to resolve in later stages of the implementation flow. As emphasized in Xilinx's UltraFast Design Methodology Guide (http-5), early identification and mitigation of timing bottlenecks are essential. This involves adopting a hierarchical design approach, leveraging constraint-driven placement strategies, and carefully managing clock domains and timing-critical paths.

This approach, derived from the Xilinx Large FPGA Design Methodology Guide (http-9), recommends organizing the design hierarchy so that each logic module is confined to a single SLR, which simplifies logic assignment and speeds up the place-and-route process.

In summary, achieving timing closure on multi-die FPGAs demands a design approach that is both structurally hierarchical and timing aware. By proactively addressing the physical and logical implications of SLR boundaries and employing robust floorplanning and constraint-driven techniques, designers can effectively meet the stringent timing requirements of large-scale, high-performance FPGA systems.

2.3.2 Timing closure

Timing closure remains one of the most critical and challenging aspects of FPGA design, particularly in architectures that span multiple dies. Multi-die FPGA introduce significant complexity due to increased interconnect distances, multiple clock domains, and added latency across SLR boundaries. Achieving reliable timing closure in such systems requires a combination of detailed design constraint management, intelligent floorplanning, and advanced timing optimization techniques.

The physical separation between SLRs leads to longer signal paths and greater propagation delays. These delays, if not accounted for during early design stages, can cause timing violations that are difficult to resolve in later stages of the implementation flow. As emphasized in Xilinx's UltraFast Design Methodology Guide (http-5), early identification and mitigation of timing bottlenecks are essential. This involves adopting a hierarchical design approach, leveraging constraint-driven placement strategies, and carefully managing clock domains and timing-critical paths.

According to the Xilinx Large FPGA Methodology Guide (http-9), the following practice is recommended for timing-aware partitioning on SSI-based devices: "Define design hierarchy such that entire logic modules can be contained within a single SLR. This not only simplifies logic assignment but also accelerates place-and-route operations."

In summary, achieving timing closure on multi-die FPGAs demands a design approach that is both structurally hierarchical and timing aware. By proactively addressing the physical and logical implications of SLR boundaries and employing robust floorplanning and constraint-driven techniques, designers can effectively meet the stringent timing requirements of large-scale, high-performance FPGA systems.

2.4. HLS Tools

HLS tools have surfaced as a transformative approach to hardware design, enabling designers to develop complex systems using high-level programming languages such as C, C++ and SystemC. HLS tools abstract the hardware design to a higher level, allowing the automatic generation of RTL code tailored for FPGA or ASIC applications. Figure 2.3 shows the general steps for an HLS-based design. These tools enable designers to explore architectural optimizations for performance, area, and power in an efficient manner, significantly reducing design effort and time to market. Among notable HLS tools, Xilinx Vivado HLS (http-10) stands out for its robust feature set and fine-grained optimizations for Xilinx FPGA platforms, while Mentor's Catapult C (http-11) and Intel HLS Compiler (http-12) highlight design exploration and seamless integration with their respective ecosystems, enabling them to expand into broader ASIC and FPGA applications. Open-source solutions such as LegUp (http-12) further expand HLS accessibility, particularly in academic and experimental research.

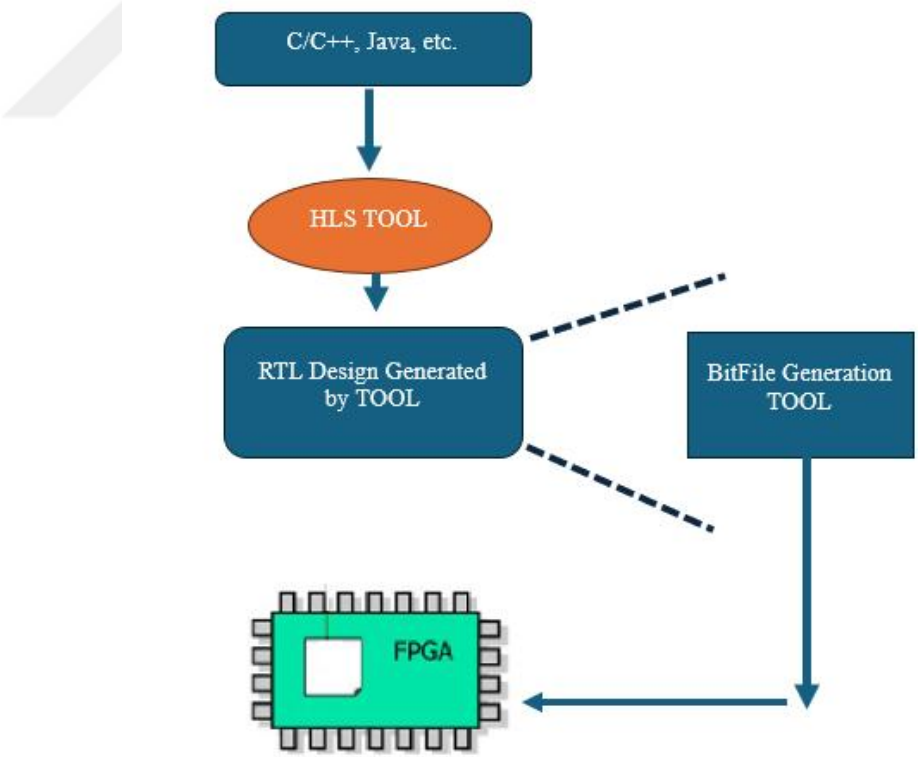

Figure 2.3: *General HLS tool schematic*

Despite the productivity benefits provided by HLS tools, a significant performance gap persists between HLS-generated designs and handcrafted RTL implementations, particularly in terms of achievable clock frequency. This gap is further widened as modern FPGAs become increasingly heterogeneous and often span multiple silicon dies [14]. In such architectures, long routing paths especially those involving SLR-crossing wires introduce additional delay that can severely limit the maximum operating frequency of large designs.

Currently, most HLS tools lack built-in support for SLR-aware resource allocation and do not automatically detect or resolve issues arising from SLR-crossing signals [15]. When an HLS design is synthesized across multiple SLRs, it may inadvertently generate critical data or control paths that span these regions. However, due to the absence of physical layout awareness during the high-level synthesis stage, such cross-SLR connections often go unrecognized and are insufficiently pipelined. As a result, these under-optimized paths frequently become the primary bottleneck for achieving timing closure.

In this thesis, a commercial HLS compiler is improved to address the timing and routing challenges of large-scale FPGA designs. Unlike existing tools, the compiler incorporates SLR-based partitioning during the RTL generation phase from an arbitrary C++ code, allowing it to make FPGA architecture-aware decisions while mapping the design. This approach helps optimize resource allocation, reduce cross-SLR communication, and aims to alleviate routing congestion. By addressing physical layout concerns during RTL generation, the proposed solution enhances timing closure for SLR-spanning designs and contributes to the broader applicability of HLS methodologies in high-performance, multi-die FPGA architectures.

2.5. AWS EC2 F1 Instances

The AWS EC2 F1 instance family provides a cloud-based FPGA acceleration platform designed for high-performance computing applications. It includes three instance types: f1.2xlarge, f1.4xlarge, and f1.16xlarge, each integrating a combination of powerful CPUs and FPGA chips to support compute-intensive workloads. Each instance is equipped with an Intel Xeon E5-2686 v4 x86 processor and a Xilinx Virtex UltraScale+ VU9P FPGA (device: xcvu9p-flgb2104-2-i). The CPU and FPGA are connected via a Peripheral Component Interconnect Express (PCIe) Gen3 x16 interface, enabling high-

throughput communication between host software and hardware accelerators. A high-level architectural overview of the F1 instance, illustrating the interaction between the host processor, PCIe interface, and multi-die FPGA, is shown in Figure 2.4.

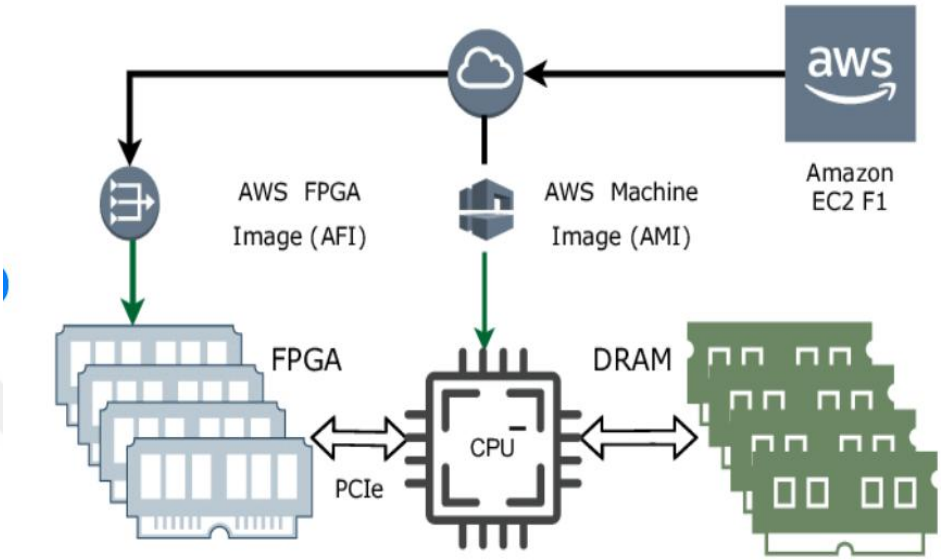

Figure 2.4: *AWS EC2 F1 instance architecture (http-13)*

2.5.1. Implementation on F1 instances

Ideally, an FPGA or ASIC chip serving as part of an application-specific hardware accelerator should be directly connected to a network interface (e.g., (http-14)). However, in the AWS EC2 F1 platform, direct access from the FPGA to the network interface is not allowed. To address this, AWS provides a sample design called Custom Logic Streaming Data Engine (CL SDE) along with a supporting software layer called Virtual Ethernet (http-15) that allows Ethernet packets to be routed between the network interface and the FPGA.

This setup leverages the high-performance, open-source Data Plane Development Kit (DPDK) library (http-16) running on the embedded x86 host processor. DPDK significantly reduces overhead and improves packet processing speed by bypassing kernel participation using poll-mode user-space drivers. The combination of CL SDE, Virtual Ethernet, and DPDK creates a high-throughput network device that interfaces with the FPGA. The HLS compiler utilizes an extended version of the CL SDE design to interface with a dedicated hardware accelerator section implemented within the FPGA. After generating accelerator logic intended for deployment in the cloud, the compiler inserts

the synthesized design into the blue colored region illustrated in Figure 2.5, labelled as accel_top_<ChipNumber>. Communication between this accelerator and the external environment is facilitated through an extended CL SDE module, which provides Advanced eXtensible Interface-based (AXI) streaming interfaces H2C (Host-to-Card) and C2H (Card-to-Host*)* for packet data transfers. At the top of the figure, the encrypted Shell IP (f1_small_shell) forms the static infrastructure, managing PCIe connectivity and providing secure isolation within the AWS EC2 F1 environment. The following sections will provide a detailed explanation of each of these components, followed by an in-depth discussion on the SLR-based partitioning methodology.

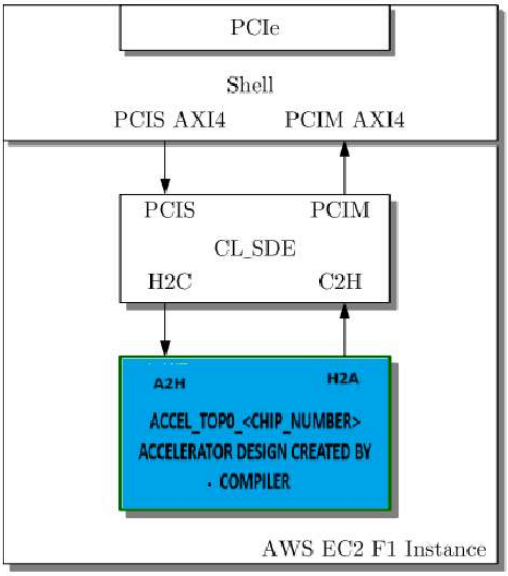

Figure 2.5: *F Type instance implementation schematic on AWS environment*

F1_small_shell implementation: The F1 Small Shell is a pre-built, encrypted Design Checkpoint (DCP) provided by the official aws-fpga GitHub repository (http-17). This shell serves as a foundational component for developers working on FPGA-based accelerators targeting AWS EC2 F1 instances. It is compatible with Xilinx Vivado design tools, specifically versions 2021.2 and 2024.1. The primary function of the F1 Small Shell is to establish a PCIe communication interface between the FPGA fabric and the onboard x86 processors within the F1 instance. This communication channel enables data to be transferred between custom accelerator logic implemented on the FPGA and software applications running on the host system. The shell abstracts much of the low-level infrastructure setup, allowing developers to focus on the core accelerator logic, while relying on a tested and reliable PCIe interface implementation.

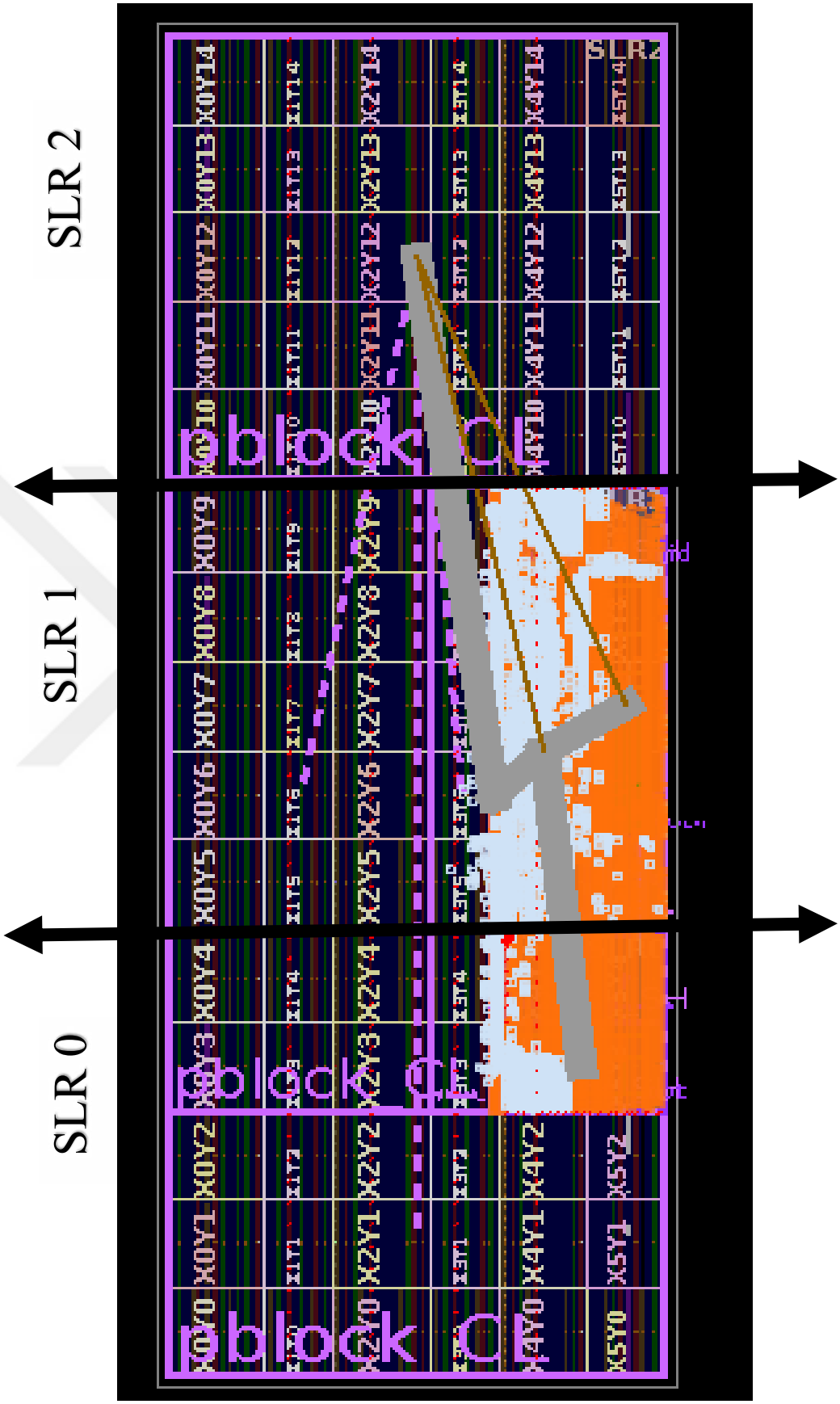

Figure 2.6: *Placement of f1_small_shell across SLR0 and SLR1 on VU9P FPGA*

As shown in Figure 2.6, the placement of the f1_small_shell logic spans between SLR0 and SLR1. A detailed breakdown of the resource usage for f1_small_shell can be found in Table 2.1 and Table 2.2. This is an important aspect to consider during SLR-based partitioning, as the compiler must interpret such placement information accurately to perform an effective partitioning. According to Table 2.1, the number of SLR crossing wires used by f1_small_shell is 2,277. Given that the total number of available crossing wires between SLRs on the VU9P FPGA is 17,280, this means that approximately 13% of the crossing capacity is already consumed by the f1_small_shell logic.

Additionally, the SLR-level Configurable Logic Block (CLB) and LUT resource utilization can be found in Table 2.2. The f1_small_shell consumes approximately 13.76% of the CLBs on SLR0 and 33.77% on SLR1. As a result, a significant portion of the available resources on SLR1 are already consumed by the f1_small_shell logic.

Table 2.1: *F1_small_shell SLR crossing net report*

Cell Names	*Number of Nets Crossing SLRs*	*SLR0 - SLR1 Cuts*	*SLR0 - SLR2 Cuts*	*SLR1- SLR2 Cuts*
WRAPPER_INST	520	520	0	0
SH	1023	1023	0	0
/SH/SH	655	655	0	0
/SH/SH/MGT_TOP	6	6	0	0
SH/dma_inst	3	3	0	0
/dma_inst/inst	3	3	0	0
/udma_wrapper	12	12	0	0
dma_top	41	41	0	0
axi4m_bridge_top_inst	1	1	0	0

Table 2.2: *F1_small_shell CLB and dedicated block utilization on VU9P device*

Site Type	*SLR0*	*SLR1*	*SLR2*	*SLR0 %*	*SLR1 %*	*SLR2 %*
CLB	6776	16634	0	13.76	33.77	0.00
CLBL	3419	8234	0	13.90	33.47	0.00
CLBM	3357	8400	0	13.61	34.06	0.00
CLB LUTs	39640	74850	0	10.06	18.99	0.00

3. GLOBAL'S HLS COMPILER

A commercial HLS compiler, which is developed according to [16] by Global Supercomputer, is used in this thesis for implementing SLR-based partitioning on multi-die FPGAs. This section gives a comprehensive explanation of the HLS compiler's features and workflow.

3.1. The Compiler's Overview

The compiler is an automatic parallelizing HLS tool designed to transform single-threaded software applications, written in conventional sequential C/C++ programming language, into multi-chip, application-specific hardware accelerators. By converting sequential code into parallel hardware implementations, the compiler facilitates the creation and deployment of scalable multi-FPGA systems. Notably, the hardware accelerators generated by the compiler are functionally identical to the original sequential single-threaded application. As a result, invoking the hardware accelerator is equivalent to executing the original single-threaded function in software, ensuring seamless compatibility and functional consistency. The compiler automates multi-FPGA partitioning and deployment on AWS EC2 F1 instances, simplifying the process of creating high-performance hardware accelerators while preserving the functional equivalence of the original sequential programs.

3.2. Compilation Flow

The overall compilation process of the HLS compiler can be found in Figure 3.1. This flow shows transformation of a sequential C/C++ program into a multi-chip FPGA accelerator system deployable on AWS EC2 F1 instances. The compiler phases in Figure 3.1 are briefly explained below:

1. Code Compilation: The gcc/g++ compiler is invoked to translate the user's C/C++ program into x86 assembly code. The user specifies functions for acceleration, and optimizations are applied only to these functions.
2. Intermediate Code Translation: The x86 assembly is converted into unoptimized intermediate code composed of Reduced Instruction Set Computer (RISC) primitives, complying to the x86 specification.
3. Optimization: Standard and x86-specific optimizations are applied to produce optimized RISC-like intermediate code.

4. Scheduling: Hierarchical software pipelining schedules the optimized intermediate code as although targeting a wide-issue architecture, generating pipelined program regions.
5. Finite State Machine (FSM) Generation: Each pipelined program region is transformed into a Verilog FSMs.
6. Design Integration: FSMs from user code and compiler libraries are integrated into a non-partitioned accelerator design. Components include application-specific communication networks, on-chip memories which are placed in the BRAMs in actual FPGA implementation, floating-point units, pipelined interfaces, and more. Loop FSM duplication counts are determined for performance.
7. Partitioning: The flat accelerator design is partitioned across multiple FPGAs, with I/O controllers added for cross-chip communication. Each partition is placed into single FPGA while creating FPGA images.
8. Executable Packing: The un-accelerated software, a manager program, and FPGA image references are combined into an executable. The manager program initializes FPGA instances and ensures hardware acceleration initialization process is done. A standalone Verilog tarball for Vivado processing on AWS is also generated.
9. AWS-FPGA Image Creation: The Verilog tarball is processed by AWS FPGA Developer Amazon Machine Images (AMIs), generating FPGA images for each partition. Once all images are ready, the hardware-accelerated executable becomes fully operational, though it can run in software mode before acceleration is enabled.

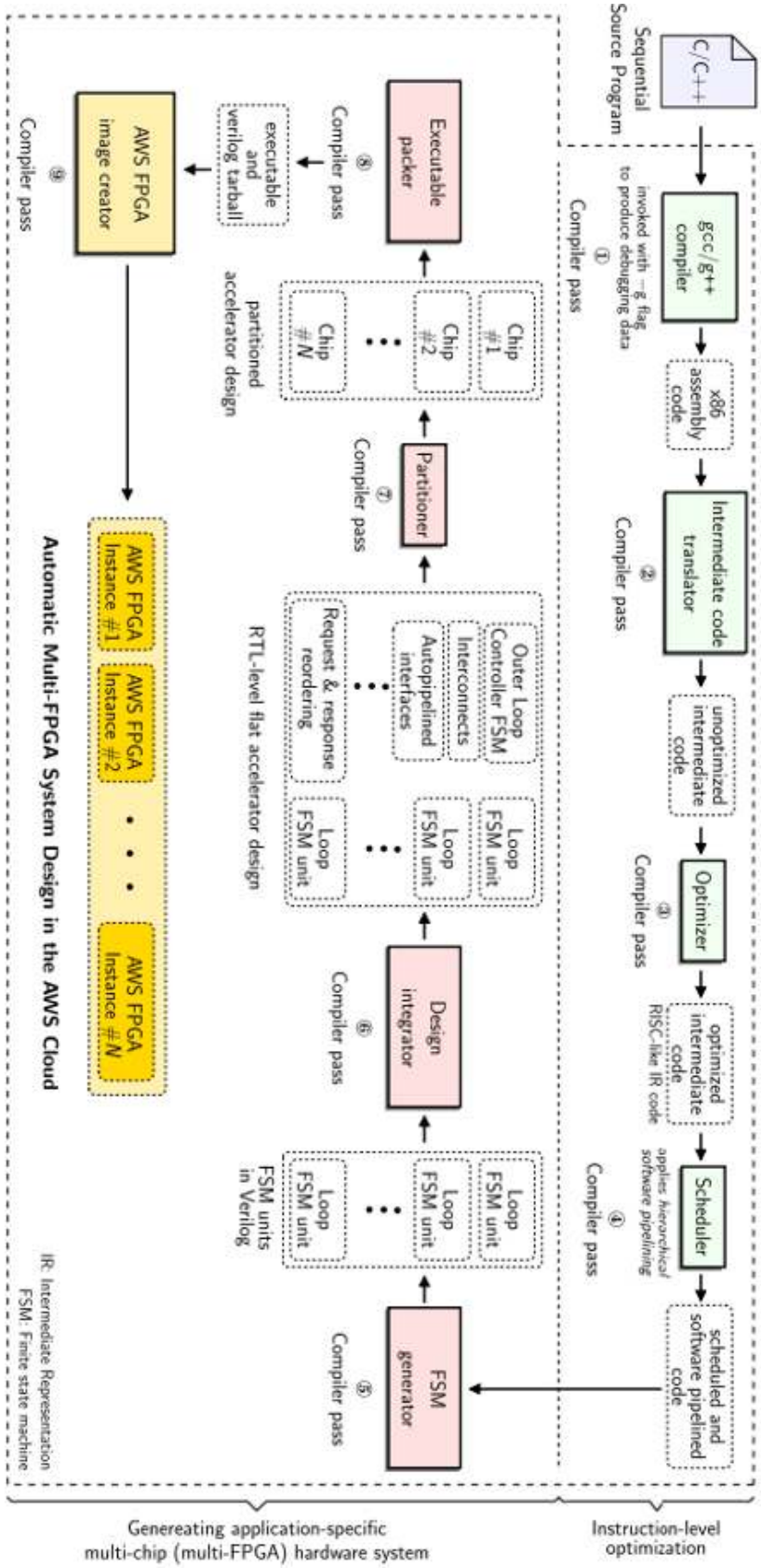

Figure 3.1: Overview of the proposed HLS compiler for multi-FPGA hardware design synthesis from a sequential C/C++ program

3.3. Flat Supercomputer Design

Prior to the partitioning stage (Stage 7) shown in Figure 3.1, the compiler translates the user-defined C/C++ code into a corresponding flat, non-partitioned supercomputer design. To provide a consistent presentation of the compiler throughout the thesis, a sample C++ application that is composed of three nested for-loops is considered. In this example scenario, the compiler is configured to generate a supercomputer design that will be implemented on 64 FPGA chips and include 64 inner loop components, 8 middle loop components, and a single outer loop component.

Figure 3.2 illustrates a flat, non-partitioned supercomputer design generated by the HLS compiler, while all figure legends used throughout the thesis are illustrated in Figure 3.3. As shown in Figure 3.2, the flat design architecture features several on-chip networks. For instance, the outer loop component dispatches tasks to the eight middle loop components through the Loop Network. Similarly, to establish the system's memory hierarchy using dedicated L1 and L2 memory units for each partition, 64 L1 and 64 L2 memory units are connected by means of the L1-to-L2 Network, creating 64 connections within a unified, non-partitioned framework.

In the following sections, we will explore how a flat supercomputer design such as in Figure 3.2 can be partitioned for implementation on 64 FPGA chips. We will also analyze how the networking architecture must be adapted, comparing the conventional FPGA-based partitioning approach with the proposed SLR-based partitioning methodology.

3.4. FPGA-based Partitioning

The HLS compiler used in this thesis implements an FPGA-based partitioning methodology in which components such as FSMs and memory units are allocated to appropriate FPGA chips. Once the partitioning is complete, the compiler automatically generates the necessary networking elements to ensure inter-component communication. The final design is organized under a single top-level module that manages the entire system across multiple SLRs. This section presents the original partitioning strategy employed by the compiler, along with its supporting architectural components, including the I/O controller and internal networking mechanisms.

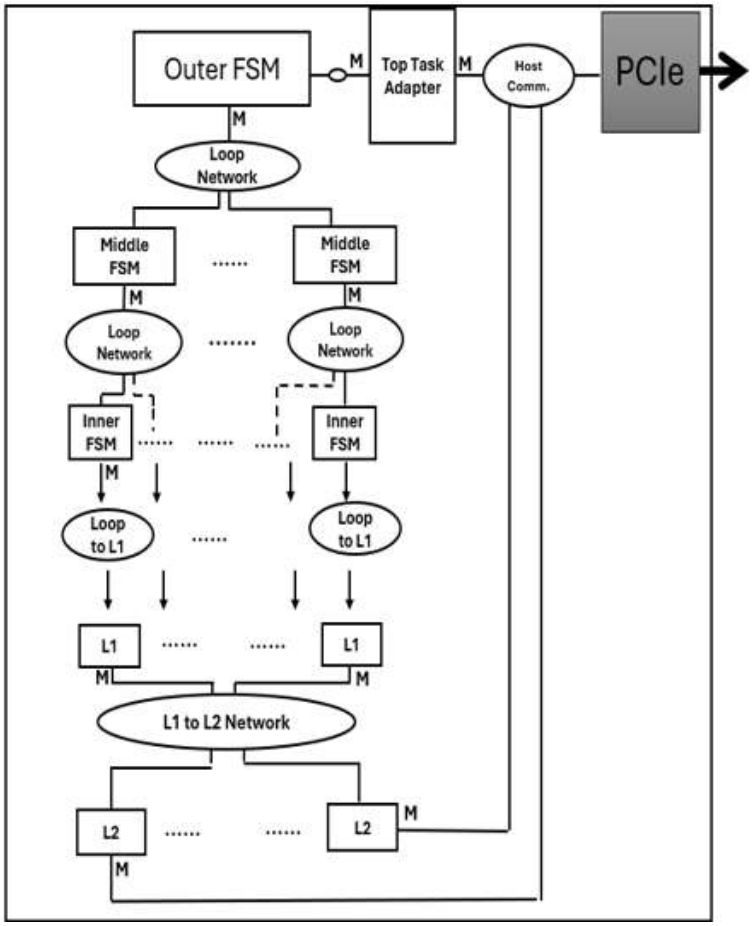

Figure 3.2: *An example of HLS generated flat supercomputer design schematic*

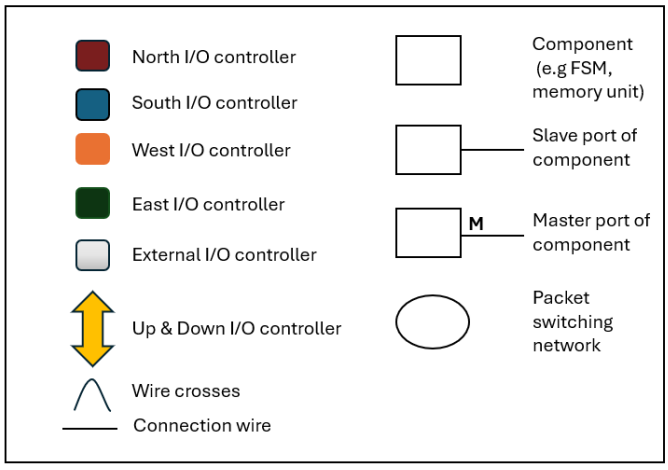

Figure 3.3: *Legend for schematic representations used throughout the thesis*

3.4.1. Recursive bisection method

The recursive bisection partitioning method is a hierarchical approach for dividing a problem space into progressively smaller partitions. The process starts with an initial set of elements referred to as components in our context, namely only memory elements and FSMs in the flat design. In the first step, they are split into two subsets. Each subset is then recursively divided into two equal parts, and this process continues until the desired number of partitions is achieved. The pseudocode for the recursive bisection algorithm is presented in Algorithm 1.

Algorithm 1: *Recursive bisection partitioning*

```
Require: elements, stoppingCriterion
Ensure: Partitioned elements
if stoppingCriterion is met then
    return elements
else
    (A, B) ← PARTITION(elements)   ▷ Divide into two subsets
    Subset1 ← RECURSIVEBISECTION(A, stoppingCriterion)
    Subset2 ← RECURSIVEBISECTION(B, stoppingCriterion)
    return COMBINE(Subset1, Subset2)
end if
```

After partitioning applied to the flat design, which is composed of one outer FSM, 8 middle FSMs, 64 inner FSM s, 64 L1 and 64 L2 memory units, FSMs and memory units are distributed across FPGA chips, which are organized into three different types:

Type 1 FPGA (outer FSM FPGA): This FPGA contains the top-level task adapter, one outer FSM, one middle FSM, one inner FSM, one L1 memory unit, and one L2 memory unit. It is unique because it is the only FPGA containing the outer FSM and responsible for providing communication with host.

Type 2 FPGAs (inner FSM FPGAs): Each of these FPGAs contains one inner FSM, one L1 memory unit, and one L2 memory unit. In this example, all FPGAs except for FPGA partitions 1, 9, 17, 25, 33, 41, 49, and 57 are designated as inner FSM FPGAs.

Type 3 FPGAs (middle FSM FPGAs): Every 8th FPGA after the first (e.g., the 9th, 17th, 25th FPGA, etc.) is assigned as a type 3 FPGA. Each type 3 FPGA contains one middle FSM, one inner FSM, one L1 memory unit, and one L2 memory unit.

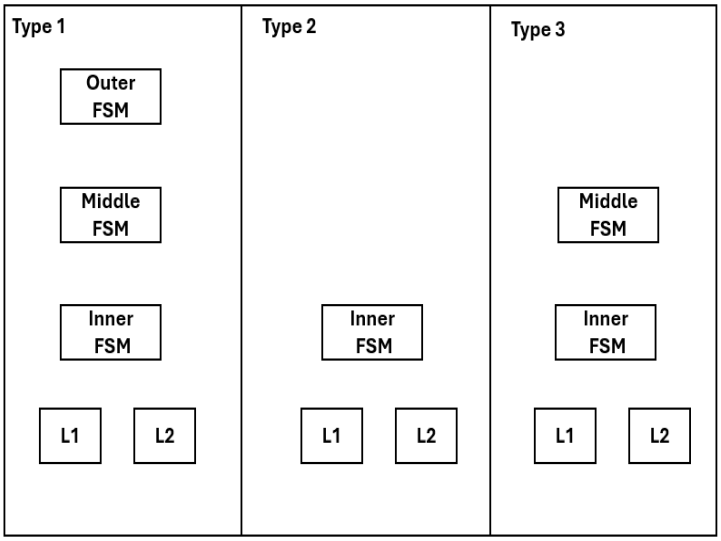

Figure 3.4*: Partitioned but not connected supercomputer design schematic*

Thus, in a 64-FPGA system, the first FPGA (type 1) is unique because it manages the outer FSM and responsible for the host communication. Every 8th FPGA after the first (9th, 17th, 25th, etc.) is designated as a type 3 FPGA due to containing a middle FSM, while the rest are type 2 FPGAs. At this stage of the compiler flow, no networking or communication components between the partitioned components have been created yet; the system only reflects the logical assignment of components to FPGAs.

3.4.2. I/O controller

In a multi-FPGA system, each FPGA chip is equipped with I/O controller unit that enables chip-to-chip and host communication. Specifically, the first FPGA chip's I/O controller includes a direct interface with the host central processing unit (CPU), while the others are solely responsible for chip-to-chip communication. Thus, the first FPGA chip's I/O controller manages communication between the host and the entire multi-FPGA design. All I/O controllers exchange data via UDP packets, and their design includes the hardware for constructing UDP packets. The networking parameters such as media access control (MAC) addresses, IP addresses, and UDP port numbers are dynamically reconfigurable at FPGA initiation time. It should be noted that the communication between computational units (e.g., loop FSMs) and the I/O controller is facilitated via internal packet-switched networks.

When there is a one-to-one connection between a source and a destination, the compiler applies a network elision optimization to enable direct communication, bypassing the need for intermediate switching logic. However, in scenarios involving one-to-many or many-to-one communication patterns, internal switching networks are employed to manage routing. In such cases, messages are forwarded based on virtual port numbers that uniquely identify the sending or receiving computational components. These virtual port numbers are mapped internally to the appropriate physical port number by the switch network, enabling correct delivery. This mechanism makes scalable communication routing within the FPGA, abstracting the underlying hardware topology while maintaining deterministic message transfer. Additionally, as the data widths of internal units may not match the external I/O bus width, a bus_width_converter module is inserted to harmonize the widths before transmission or after reception. To prepare data for transmission, a message_header_inserter module wraps outgoing packets with necessary headers (Ethernet, IP, and UDP), including checksum computations. These packets then pass through a butterfly switch network module that multiplexes traffic from multiple sources to a single output. The data is then buffered by an auto_pipelined_reg module, which inserts appropriate pipeline stages to meet timing constraints based on FPGA placement decisions. This output is ultimately sent over the external interface. Conversely, incoming packets follow a reversed pipeline. They enter through the external I/O interface, are buffered by auto_pipelined_reg, routed to their destination unit via the switching logic, stripped of their headers using a message_header_deleter, and finally passed through a bus_width_converter to match the receiving unit data width. This modular and hierarchical I/O controller design enables scalable and efficient data movement across the multi-FPGA system, ensuring reliable chip-to-chip and chip-to-host communication in hardware-compiled designs.

Figure 3.5 illustrates that the I/O controller is located during the early compiler stages. After partitioning is completed, the internal logic of the I/O controllers is not yet fully elaborated. However, each partition is assigned to its own I/O controller to enable the creation of partitioned network structures in the next phases. In an FPGA-based partitioning approach, each I/O controller is responsible for managing and serving the components assigned to its partition, meaning it handles communication for the same group of components within that partition.

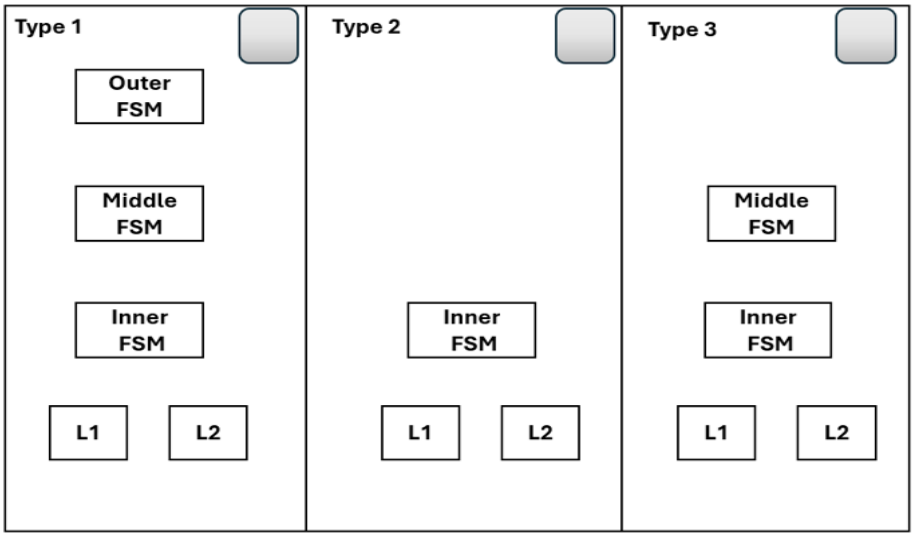

Figure 3.5*: I/O controller detection and implementation phase in compiler*

3.4.3. Networking in partitioned designs

Once FPGA-based partitioning is applied, the flat design is divided into multiple partitions, each mapped to a specific FPGA chip. The compiler assigns FSMs (representing loop structures such as Outer, Middle, and Inner Loops) and memory components (L1 and L2 units) to these partitions. This logical distribution of components forms the basis of the final hardware system. However, to ensure correct communication between components now residing on separate FPGA chips, a networking infrastructure must be established.

The networking architecture is automatically constructed by the compiler based on the partitioned component layout. Networking elements are inserted to forward data to the correct destination, whether that destination is on the same chip or another FPGA. Each partition also consists of an I/O controller, which manages communication between the internal components of a partition and external resources (other FPGAs).

Figure 3.6 illustrates the architecture of a type 1 FPGA, which contains the top-level Outer FSM along with one Middle FSM, one Inner FSM, one L1 memory, and one L2 memory unit. This FPGA is unique in that it establishes communication with the host CPU to initialize the system and distribute tasks to other FPGAs.

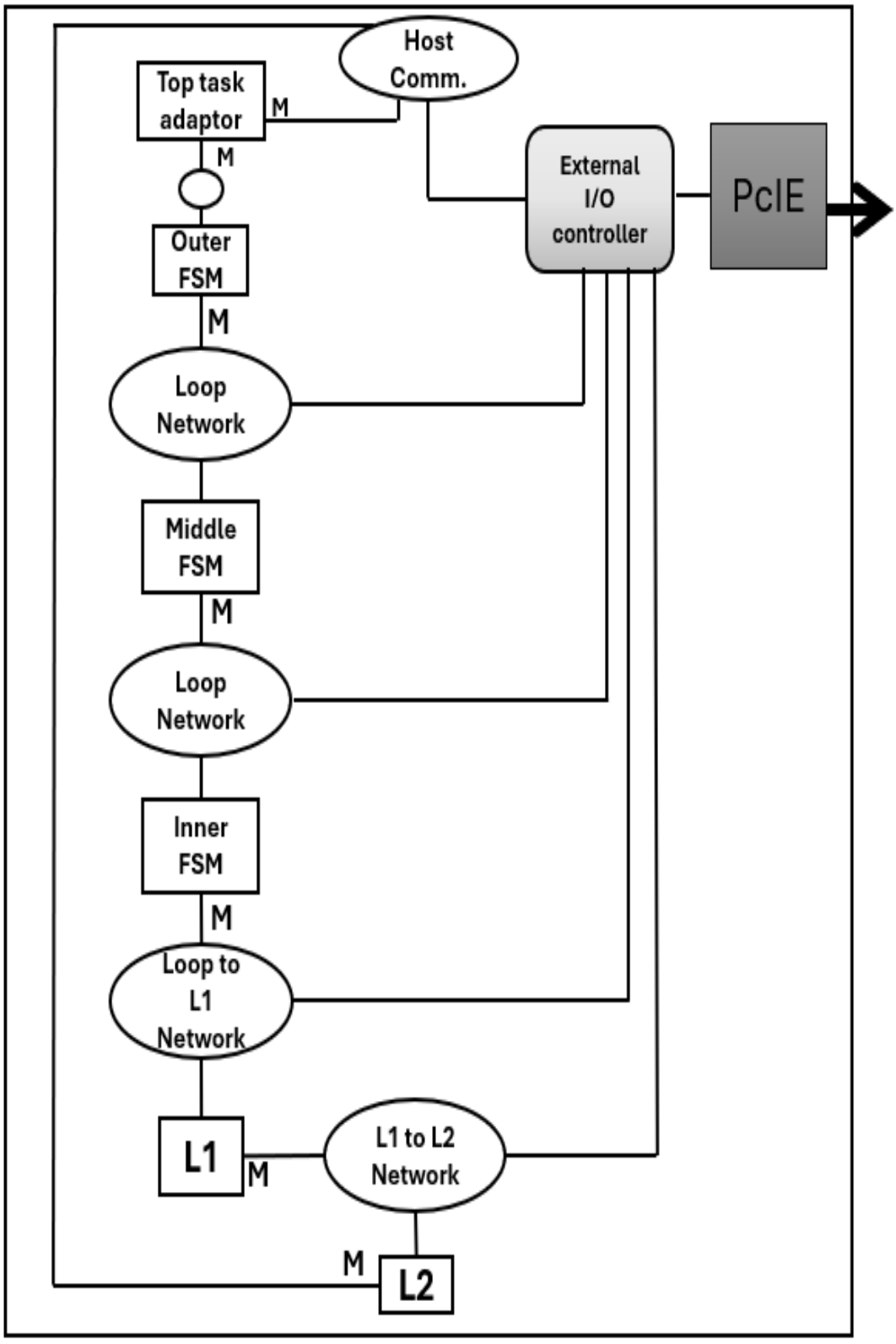

Figure 3.6: *Partitioned type 1 FPGA chip design illustration*

Figures 3.7 and 3.8 present the type 2 and type 3 FPGA designs, respectively. Type 2 FPGAs represent standard Inner Loop partitions, each containing one Inner Loop FSM along with L1 and L2 memory units. This is the most common configuration in the 64-FPGA supercomputer. In contrast, type 3 FPGAs (every 8th FPGA after the first) manage one Middle Loop FSM in addition to an Inner Loop and associated memories. These Middle Loops coordinate the execution of eight Inner Loops and act as control points in the system's loop hierarchy.

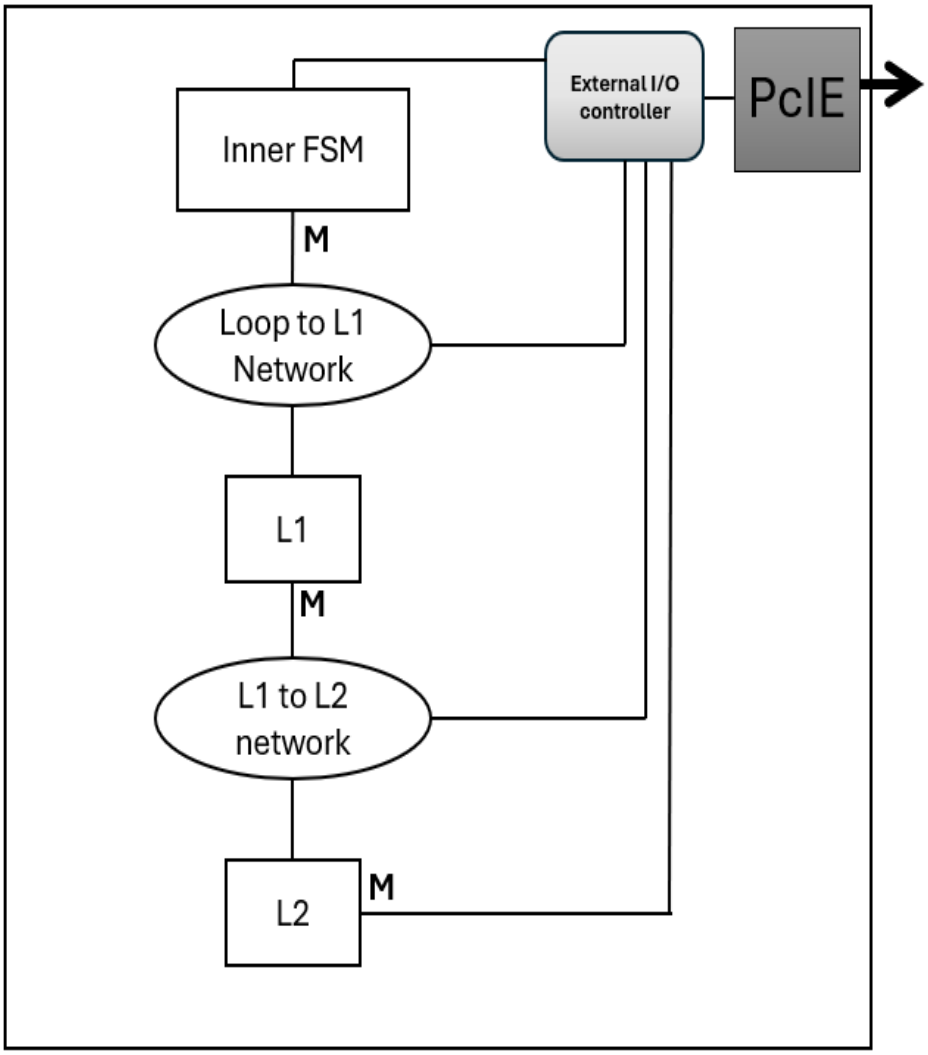

Figure 3.7: *Partitioned type 2 FPGA chip design illustration*

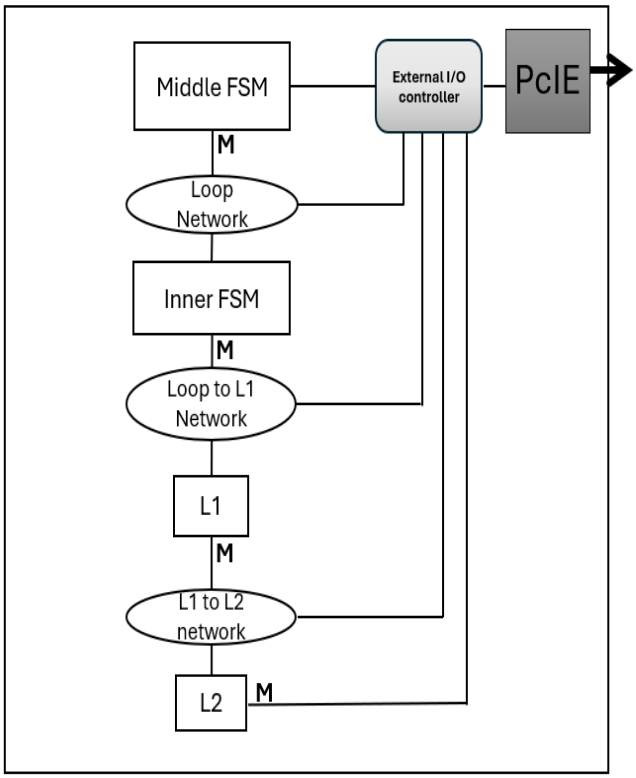

Figure 3.8: *Partitioned type 3 FPGA chip design illustration*

The network system in an FPGA-based architecture is designed around the concept of virtual-to-physical port mapping, which simplifies inter-component communication and supports scalability in multi-FPGA environments. Each network element such as a switch module or routing unit has multiple master and slave ports; each assigned a physical port number and connected to internal components within the FPGA design.

At a higher level of abstraction, used during the compilation phase (i.e., the flat supercomputer creation phase), the compiler assigns virtual port numbers to represent each communication endpoint in the design. These virtual port numbers abstract away the physical connections, allowing the compiler to define the communication flow without being constrained by physical layout or routing limitations. This abstraction is especially beneficial in large-scale or segmented designs, where the same component type may be instantiated multiple times, possibly across different FPGAs. To bridge the gap between these virtual design-time identifiers and the actual physical hardware, the compiler inserts virtual-to-physical port translation logic into each network element. This translation mechanism dynamically resolves the virtual port number in incoming messages and determines the appropriate physical port to handle message transmission.

For example, consider a scenario where a network unit is connected to multiple identical modules (e.g., several FSM-based compute units), each mapped to a different physical port. When a message is received, the virtual port number embedded within it identifies the intended destination. The network unit then consults its internal mapping table to translate the virtual port number into the corresponding physical port and forwards the message accordingly.

This routing mechanism also plays a vital role in multi-FPGA systems. If a received message contains a virtual port number that does not map to any module within the local FPGA, this indicates that the target destination is located on a remote FPGA. In such cases, the network unit forwards the message to an I/O controller module. The I/O controller is responsible for managing chip-to-chip communication and forwarding the message to the appropriate FPGA that contains the intended component.

In this way, the overall network architecture supports both one-to-one and one-to-many communication models, enabling flexible partitioning and scalable system design. This is achieved using virtual port numbers and compiler-generated translation logic, as discussed earlier.

4. SLR-BASED PARTITIONING

This chapter begins by addressing the limitations of FPGA-based partitioning methodologies and introduces the proposed SLR-based partitioning technique as a solution to mitigate these limitations.

4.1. Introduction to SLR-based Partitioning

Traditional FPGA-based partitioning methods organize RTL components without explicitly considering SLR hierarchy in FPGA. While this approach enables the utilization of the entire FPGA fabric, it introduces substantial challenges for large-scale designs, as mentioned in Section 1. Notably, routing signals across SLRs without accounting for physical layout constraints can result in SLR crossing violations. These violations typically arise when the number of inter-SLR wires (SLLs) exceeds the available crossing capacity. This situation, referred to as an SLL cut violation, leads to routing congestion and prevents successful implementation.

To address these limitations, the proposed SLR-based partitioning methodology divides the design explicitly along SLRs. By incorporating awareness of the physical SLR structure into the partitioning process, this method significantly reduces inter-SLR communication and mitigates the risk of SLL cut violations and creates independent partition designs between SLRs. In support of this strategy, the compiler has been enhanced to insert lightweight simple I/O controller IPs, which manage communication between neighboring partitions residing in different SLRs of the same FPGA. Unlike the original I/O controller IP designed for chip-to-chip communication across separate FPGAs these simplified controllers enable direct, low-overhead connections within the same FPGA die.

In addition to improving modularity and reducing routing complexity, SLR-based partitioning results in smaller, localized sub-designs that Vivado can place and route more efficiently. This leads to better resource utilization, fewer timing violations, and greater predictability during implementation. Furthermore, the hierarchical design structure enabled by this methodology facilitates parallel communication between independent modules, both within and across SLRs, significantly enhancing scalability and performance in multi-die FPGA systems.

I/O Controller Mechanism: As discussed in Section 3, the I/O controller IP was originally responsible for enabling communication between partitions placed on different FPGAs, since each partition was mapped to a separate FPGA in the traditional partitioning approach. However, with the introduction of SLR-based partitioning, multiple partitions can now reside within the same FPGA chip, distributed across different SLRs. To accommodate this change, a new internal I/O controller mechanism was required. In response, the compiler has been extended to integrate a lightweight simple I/O controller IP, which facilitates direct communication between neighboring partitions within the same FPGA.

Networking Mechanism: In the previous FPGA-based partitioning model, the internal network logic within a partition was only responsible for routing data to local components. If communication was required with components located in other partitions, the data was routed through an external I/O controller based on the destination FPGA. However, in the SLR-based partitioning model, a single FPGA can host multiple partitions across its SLRs, which means communication can also occur internally within the FPGA. To support this, the compiler has been updated to implement a direction-based routing mechanism, enabling the internal network to forward data based on the spatial relationship of the destination partition such as north, south, west, east, down, or up within the SLR grid. This enhancement allows scalable routing within a single FPGA while maintaining compatibility with inter-FPGA communication protocols.

SLR-Based Design Hierarchy: In traditional FPGA-based partitioning, each FPGA hosted a single partition, and thus a single top-level module was sufficient. However, with SLR-based partitioning, a single FPGA may contain multiple partitions, each mapped to a separate SLR. To support this, the compiler's module hierarchy has been restructured to introduce a two-level design. The first level includes an FPGA-level top module that instantiates and connects all partitions residing within the FPGA, while the second level contains individual top modules for each partition.

This section summarized the motivation for SLR-based partitioning and highlighted key enhancements to the GLOBAL HLS Compiler, including new compiler arguments, improved I/O controller logic, direction-aware networking, and a multi-level RTL hierarchy. The following sections provide a detailed explanation of how these features enable scalable partitioning for large multi-die FPGA supercomputer designs using the SLR-based methodology.

4.2. I/O Controller

GLOBAL's HLS compiler utilizes I/O controllers to manage both design partitioning and inter-chip communication across FPGAs. In essence, when a message is generated by a source component on a source FPGA, it is first routed through a partial on-chip network to reach the I/O controller on the source FPGA. The I/O controller then encapsulates the message into a standard UDP packet format. The UDP packet is then transmitted from the source FPGA via the network interface of the AWS EC2 F1 instance using the DPDK. DPDK employs a high-performance poll-mode driver that bypasses the operating system kernel to enable low-latency packet processing (see the virtual Ethernet and CL SDE design in the AWS FPGA GitHub repository (http-16), and the DPDK library (http-15) for fast packet handling. The message then traverses the user's AWS Virtual Private Cloud (VPC), reaching the destination EC2 F1 instance. Upon arrival, DPDK again handles the message and delivers it to the destination FPGA chip. Then, the I/O controller decodes this incoming UDP packet, restores the message to its original format, and routes it through an on-chip partial network to its destination component.

To support SLR-based partitioning, the I/O controller architecture within the compiler has been extended and modularized. The traditional controller, now named I/O controller unit external, is responsible for communication between FPGA chips and external systems, including other FPGAs. In the SLR-based model, this external controller can now serve multiple internal partitions residing on the same FPGA. Additionally, the compiler introduces new directional I/O controller modules, defined as I/O_controller_unit_<south, north, east, west, up, down>, to handle inter-partition communication within a single FPGA. These are instantiated based on the parsed --SLRs-per-chip="(x,y,z,k)" pragma, which defines the SLR layout and master SLR index. Depending on the physical relationship between partitions, the compiler inserts the appropriate I/O controllers to establish communication paths between adjacent SLRs, such as north↔south or up↔down. This modular and direction based I/O controller infrastructure enables seamless and scalable communication across both SLR boundaries and FPGA chips, ensuring compatibility with the GLOBAL compiler's hierarchical design strategy.

As illustrated in Figure 4.1, the partitions are organized from left to right corresponding to SLR0, SLR1, and SLR2, with their respective coordinates defined as (0,0,0), (0,1,0) and (0,2,0) for the (x, y, z) dimensions. This figure represents the updated

compiler I/O controller phase based on the --SLRs-per-chip pragma, which directly corresponds to the implementation phase described earlier in Section 3.4.2 and can be compared with Figure 3.5 for reference.

Following the compiler update, the tool now accurately maps partitions to specific SLRs on the Xilinx VU9P FPGA, as illustrated in Figure 4.1. partition 1, positioned on the leftmost side (SLR0), includes an I/O_controller_unit_extrenal for chip-to-chip communication, as well as an I/O_controller_unit_north to communicate with partition 2. Partition 2, located in SLR1, contains both an I/O_controller_unit_south for communication with Partition 1 and an I/O_controller_unit_north for communication with partition 3. Finally, partition 3, situated on the rightmost side (SLR2), includes only an I/O_controller_unit_south to receive messages from partition 2.

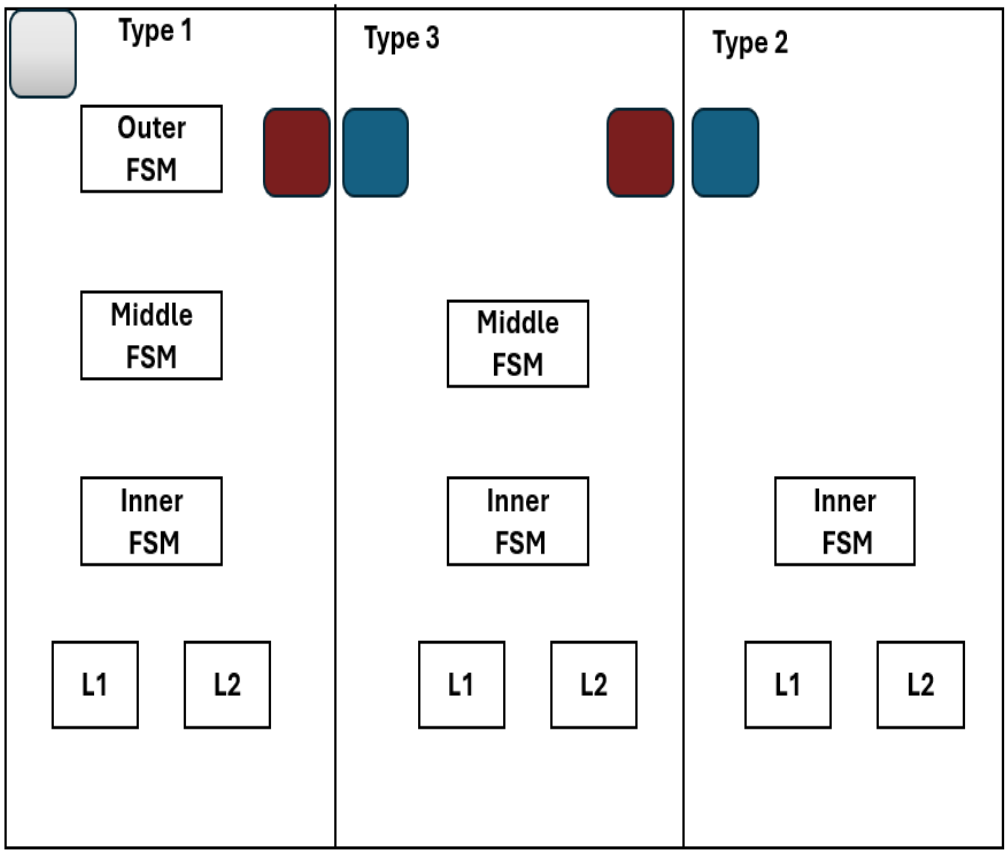

Figure 4.1*: SLR-Based I/O controller implementation on VU9P FPGA*

To illustrate the communication path: consider an incoming message destined for a component in partition 3. The message first enters the FPGA through the external I/O controller in partition 1. It then passes through the north I/O controller in partition 1, reaches the south I/O controller of partition 2, then is forwarded via its north I/O controller and finally arrives at partition 3 through its south I/O controller. Once there, the message is routed to the destination component using the internal network elements of partition 3.

This layered and direction based I/O controller strategy ensures communication across SLRs while maintaining scalability and modularity within GLOBAL's HLS Compiler framework.

4.2.1. Simple I/O controller

The simple I/O controller is a lightweight module responsible for facilitating communication between different partitions located in separate SLRs on the same FPGA chip. Unlike the external I/O controller, it implements neither UDP/IP protocol nor network switching mechanisms. Instead, this controller connects neighbor partitions in SLR-based partitioning, where multiple partitions may coexist within a multi-die FPGA. By enabling localized, direction-based packet exchange across SLRs, the simple I/O controller ensures communication while reducing routing complexity and maintaining timing closure. Within the compiler infrastructure, three distinct types of simple I/O controllers are supported, each corresponding to a specific directional interconnection. Type 1 simple I/O handles communication in the x-direction (east–west), type 2 simple I/O manages communication in the y-direction (north–south), and type 3 simple I/O is responsible for communication in the z-direction (up–down).

Figure 4.2 illustrates a generic implementation of the simple I/O controller for SLR-based communication in a multi-die FPGA environment. Once the compiler parses the --SLRs-per-chip pragma, it determines the exact location of each partition across the chip. Based on this layout, the compiler automatically detects and instantiates the appropriate simple I/O controllers between neighboring partitions. For example, if a partition lies adjacent to another in the vertical direction (y dimension), a north or south I/O controller will be generated accordingly.

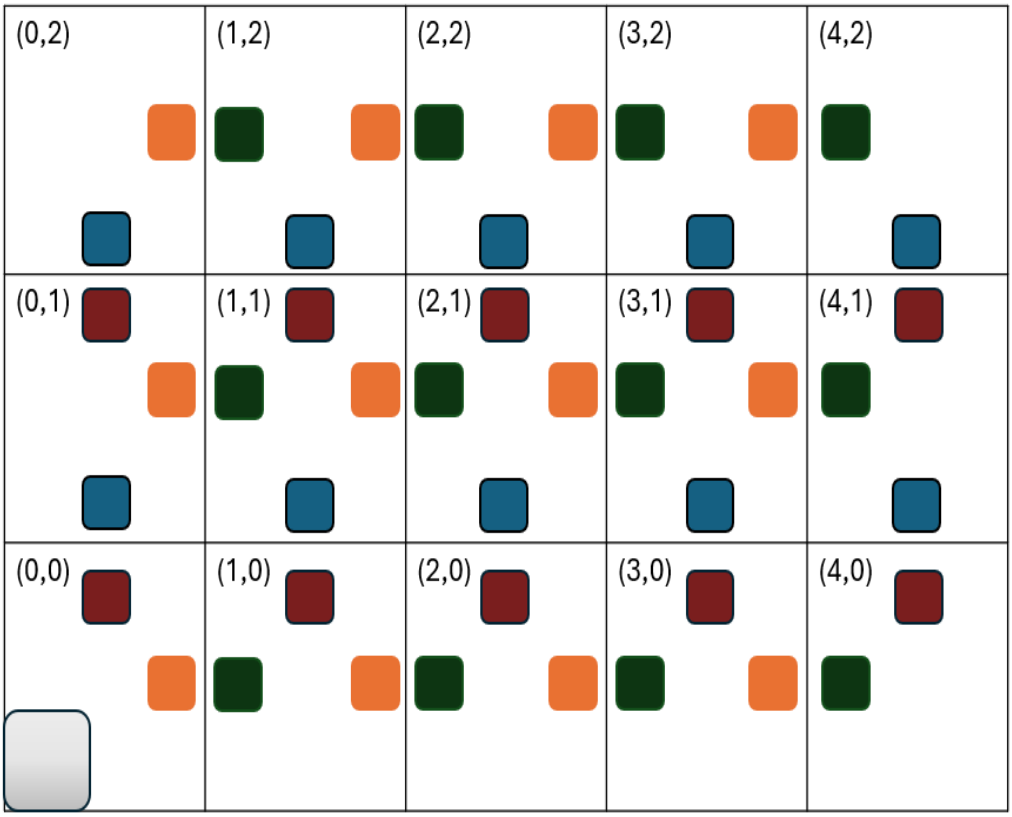

Figure 4.2: *Generic SLR-Based I/O controller implementation on multi-die FPGA*

The flexibility of the --SLRs-per-chip pragma allows users to define different partition configurations for a single FPGA chip. Depending on this user-defined layout, the compiler dynamically adapts the communication structure by inserting the necessary direction-specific I/O controller modules between partitions. This automation ensures modular and scalable partitioning while conforming to the physical constraints of multi-SLR FPGA architectures.

Since the number of available SLR-crossing wires is limited and the SLRs are interconnected through an interposer silicon, it is crucial to carefully manage signal routing across SLRs. To minimize routing congestion and ensure timing closure, the number of crossing wires must be optimized. Furthermore, as recommended by Ultra-Fast Design Methodology, automatic pipeline registers should be inserted when crossing SLRs to handle potential timing issues and to maintain high-performance communication between partitions.

Bus Width Converter IP: The Bus Width Converter IP is a specialized module available from Global's HLS hardware library that is designed to perform bus width conversions from a narrow bus to a wider bus, or vice versa. This library IP is particularly

useful to manage and optimize wide data bus transfers across SLRs in multi-die FPGA architectures.

Xilinx XCVU9P FPGA provides a total of 17,280 SLR-crossing wires between the SLR2 ↔ SLR1 and SLR1 ↔ SLR0 boundaries. However, when a design attempts to use more than approximately 6,800 wires in either direction (incoming or outgoing), SLL cut violations may occur, potentially impacting timing and routing closure. In addition, as we explain Section 2.5.2, we already use %13 nets in design statically, so that the careful optimization of SLR crossing wires is required.

To prevent these problems and make effective use of the limited SLR crossing capacity, the Bus Width Converter IP can downsize wide buses for example, reducing an N-bit bus, which may not be a power-of-two, into a power-of-two width, such as 64, 128, or 256 bits. Once the bus crosses into the destination SLR, the original bus width is reconstructed. This flexible and reconfigurable IP enables efficient utilization of available routing resources and ensures compliance with the physical constraints of FPGA chip.

Adding Auto Pipeline Registers: Designing FPGA systems with wide buses crossing SLR boundaries, introducing pipeline registers is critical for achieving timing closure and reducing routing congestion. Wide buses operating above 250 MHz are particularly prone to timing violations due to the long interconnect delays inherent in such designs. To address this, Xilinx recommends adding at least three pipeline stages one near the top, one in the middle, and one at the bottom of the SLR. For very high-performance buses or those traversing both vertical and horizontal distances, additional pipeline stages may be necessary. These pipeline registers break long signal paths into shorter segments, improving timing performance by minimizing delays and distributing routing loads across the SLR. This strategy enhances scalability by allowing larger and more complex systems to meet timing requirements without encountering congestion hotspots.

The compiler automates the insertion of these pipeline registers by analysing the data flow requirements and dynamically placing the registers at appropriate points along the bus path. For example, in a Xilinx VU190-2 FPGA, a wide bus crossing multiple SLRs from an Interlaken block in the bottom-left corner of SLR0 to a packet monitor block in the top-right corner of SLR2 might initially fail to meet a 300 MHz timing requirement. By strategically inserting pipeline registers at the top, middle, and bottom of each SLR along the bus path, the design can achieve the necessary timing closure while alleviating routing congestion.

4.3. Networking In SLR-Based Designs

Xilinx FPGAs currently utilize 2.5D integration technology, where multiple SLRs are interconnected via a silicon interposer, as seen in UltraScale+ devices. This approach enables larger and more complex designs by surpassing the limitations of monolithic FPGA architectures. However, with ongoing advances in chiplet-based technologies such as 3D stacking and hybrid bonding future FPGA systems are expected to adopt true 3D interconnect solutions [17]. Compiler frameworks must therefore evolve to support scalable, high-performance communication across multi-die and multi-layer environments.

A promising direction in this evolution is the 3-dimension mesh-connected communication architecture, inspired by multi-chip supercomputers. This model supports scalable routing across partitioned SLRs in three dimensions (x, y, z), enabling efficient packet communication in distributed systems. Although current devices like the Xilinx VU9P implement a 2-dimension SLR layout organized as bottom, middle, and top die regions, the compiler has been extended to simulate a 3D mesh communication structure. This abstraction future-proofs the design flow for 3D-stacked FPGAs by allowing seamless integration of advanced communication topologies.

In this model, each SLR is treated as a mesh node with direct links to its neighboring SLRs in the north, south, east, west, up, and down directions. The simple I/O controllers introduced in Section 4.2.1 manage packet-based communication between SLRs within this mesh. A custom pathfinding algorithm named find-way dynamically determines routing paths based on partition locations and mesh topology. This algorithm is central to enabling scalable, compiler-managed communication in multi-die FPGA systems and lays the groundwork for future 3D mesh interconnects in chiplet-based architectures.

4.3.1 Find-Way algorithm

The Find-Way algorithm determines the communication path between two partitions based on their partition numbers, using the SLR-to-chip mapping information provided in Section 4.1. It begins by calculating the SLR index for both the source and destination partitions. The algorithm then proceeds directionally in a fixed order first in the z dimension, then in y, and finally in x. At each step, it always traverses from the lower to the higher partition index, ensuring a consistent, positive routing approach and

preserving the intended directionality of communication when source partition number is higher than destination partition number.

This algorithm is executed after the partitioning phase of GLOBAL's HLS compiler and plays a key role in generating the necessary I/O controller IPs that enable communication between different partitions. In a traditional FPGA-based partitioning scenario, if two components are placed in different partitions, they require an external I/O controller to communicate. However, in SLR-based partitioning, the communication logic becomes more refined. If the components are located on different FPGA chips, external I/O controllers are still required. On the other hand, if the components are on the same FPGA chip but reside in different SLRs, they communicate through a more lightweight I/O controller structure called the simple I/O controller.

Thus, at compile time, this algorithm explores and determines the communication path between any two partitions. It records the communication logic from source to destination, including which type of I/O controller should be used in each direction. Additionally, it evaluates whether an external I/O controller is required along the path from the source to the destination. During the path exploration process, the algorithm dynamically creates new simple I/O controller components within the relevant partitions and connects them to the appropriate component networks. As a result, a runtime database of I/O controller components is built, which the compiler later uses to generate the required RTL modules. These RTL modules are dynamically written in Verilog, incorporating all the necessary features to support communication between components located in different partitions.

Unlike standard FPGA-based partitioning, this algorithm also records information about intermediate (pass-through) partitions that lie along the path from the source to the destination. For instance, consider a scenario where partition 1 resides in SLR 0, partition 2 in SLR 1, and partition 3 in SLR 3. If partition 1 needs to send a message to partition 3, but partition 2 does not originally include the required network logic, the algorithm ensures that this information is captured. By leveraging the recorded pass-through partitions data, the compiler can generate the necessary network structures even in partitions that do not directly participate in the communication.

In this example, although partition 2 does not included the communication logic for that specific component network logic, it acts as a bridge between partition 1 and partition 3. Therefore, the compiler includes the required network paths and associated I/O

controllers within partition 2, enabling the message to pass through correctly. This ensures consistent and complete network connectivity across all partitions involved in the communication path.

Algorithm:2 *Pseudo explanation of find-way algorithm*

1. Compute SLR indices for both the source and destination partitions.
2. **If** Source and Destination reside on the same FPGA chip:
   (a) Compute absolute coordinate differences in X, Y, and Z directions.
   (b) Initialize the traversal path from Source to Destination:
      i. Traverse along the Z-axis (UP or DOWN).
      ii. Traverse along the Y-axis (NORTH or SOUTH).
      iii. Traverse along the X-axis (EAST or WEST).
   (c) For each direction that involves crossing an SLR:
      i. Record both entry and exit points (e.g., EAST and WEST for X traversal).
   (d) Log all encountered I/O controller types (e.g., NORTH, SOUTH, EAST, WEST, UP, DOWN).
   (e) Identify Master-Slave roles for each traversed I/O controller.
   (f) Finalize the path upon reaching the destination partition.
3. **Else** (Source and Destination are on different FPGA chips):
   (a) Calculate the external SLR index for the source partition.
   (b) Calculate the external SLR index for the destination partition.
   (c) Apply steps 1 to 2-(f) to compute the path from Source to its external SLR.
   (d) Apply steps 1 to 2-(f) again to compute the path from Destination external SLR to Destination.

4.3.2. Network routing SLR vs FPGA based partitioning

The compiler utilizes an on-chip network to manage packet-based communication between components in the system. In this packet-based communication scheme, each packet contains the destination component ID. These IDs are assigned to the components connected to the same network during the supercomputer design phase, as described in Section 3.3. For example, consider a network with N components; each component is assigned a unique ID ranging from 0 to N-1. Assuming the recursive bisection algorithm partitions the system across N FPGAs, each FPGA will host exactly one component connected to this network. From this point, when the compiler emits the network logic for each partition, it does not directly use the virtual port numbers assigned to components. Instead, it generates new identifiers called physical port numbers. For example, if a partition contains only one component connected to a given network, that component is assigned to physical port number 0. To enable communication with other components outside the partition, the compiler assigns the next available physical port number 1 to the connection with the external I/O controller.

In another case, if two components belonging to the same network reside within the same partition, these local components are connected to the network using physical port numbers 0 and 1, respectively. The compiler then assigns the next available port number 2 for the external communication port connected to the I/O controller. This strategy ensures that local components are always assigned to the lower physical port numbers, while ports reserved for external communication are assigned higher numbers. This consistent port assignment simplifies routing and supports a structured network interface for each partition.

To implement this communication, the compiler uses a function that translates virtual port numbers to physical port numbers. During compile time, this function generates RTL code that maps the component IDs within each partition to their corresponding physical port numbers. This process will be explained in more detail later and is illustrated in Figure 4.5.

To support the routing mechanism described above, updates to the networking structure are required to enable compatibility with SLR-based partitioning. These updates are necessary to ensure correct routing behavior across multiple SLRs.

To illustrate the differences in routing implementation between FPGA-based and SLR-based partitioning, Figures 3.6, 3.7, and 3.8 can be compared with Figures 4.4 and

4.5. Figures 3.6 and 4.3 offer a direct comparison: while the component placement remains the same in both designs, the underlying networking mechanism differs. This difference is highlighted by the red routing wires shown in Figure 4.3.

Figure 4.5 presents the schematic of a type 1 FPGA design, where the partition is in SLR0 of a three-SLR FPGA. In this setup, partition 1 is placed in SLR0, partition 2 in SLR1, and partition 3 in SLR2. As illustrated in the figure, suppose partition 1 receives a packet from the external I/O controller, and the packet's destination component resides in partition 2. To enable the packet to reach its destination, the system must support forwarding the packet across different SLRs within the same FPGA.

To handle this scenario, the networking mechanism must support inter-partition communication across SLRs. The compiler fulfills this requirement by inserting additional network logic that determines the correct forwarding behavior when a packet's destination component ID does not belong to the current partition. Specifically, it must decide where to forward the packet so it can ultimately reach its target component. As explained in Section 4.3.1, this decision is based on the information derived from the find-way algorithm, which precomputes the communication paths between partitions. Using this information, the network logic can examine the incoming packet's destination component ID and forward it to the appropriate physical port connected to the corresponding I/O controller.

This mechanism enables each partition to relay packets on behalf of others when necessary and ensures that the correct I/O controller is used to forward the request. Such routing logic is enabling communication across SLRs.

For example, the "Loop to L1" network component in Figure 4.3 connects the inner FSM of the current partition, the external I/O controller, and the north I/O controller. The main mission of the "Loop to L1" network is to forward incoming requests from the Middle FSM to the appropriate Inner FSM. If a request targets a component in the northern direction, it is forwarded through the north I/O controller. If the destination lies on a different FPGA, the request is routed via the external I/O controller to the appropriate partition, as described in Section 3.4.2. If the target component resides within the same partition, the request is delivered directly to the inner FSM without involving any I/O controller.

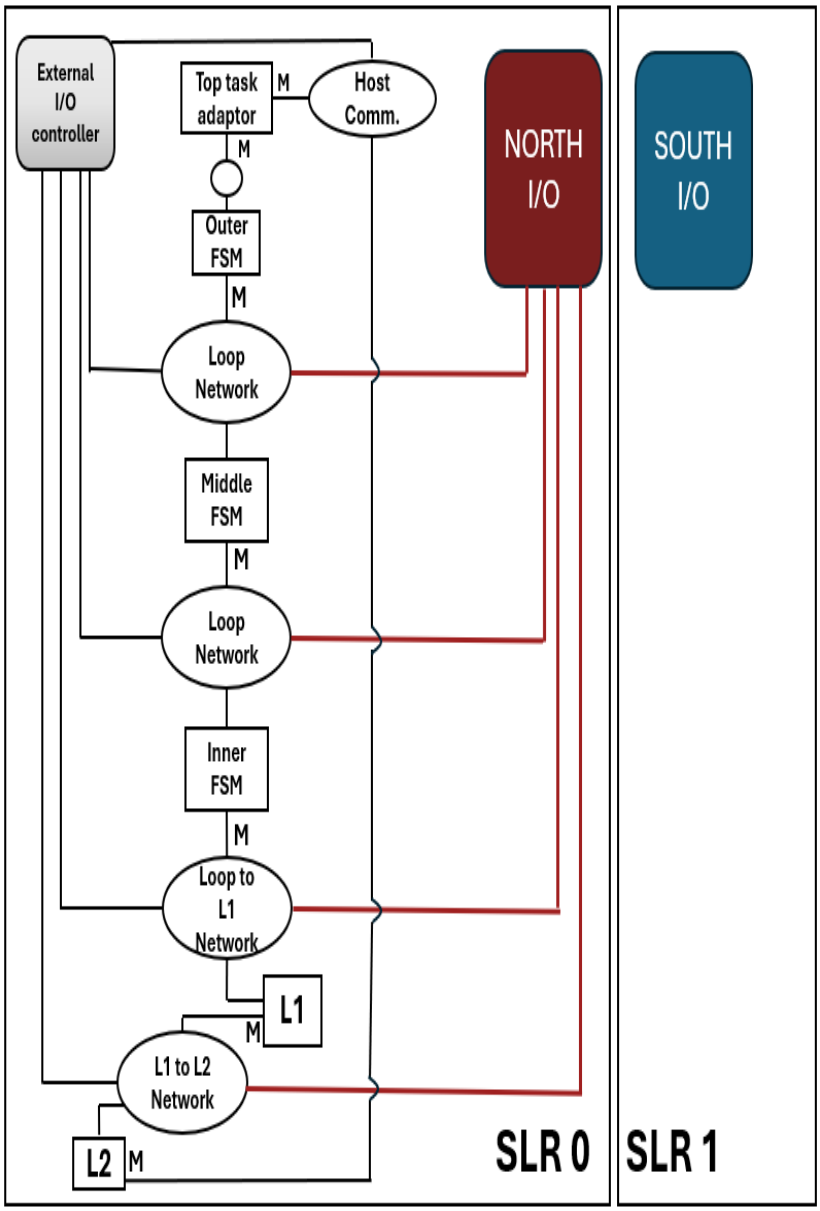

Figure 4.3: *SLR-based partitioned type 1 FPGA design networking structure in SLR 0*

Another example is illustrated in Figure 4.4, where the "Loop to L1" network connects the Middle FSM to the Inner FSMs within the same FPGA or to seven other Inner FSMs residing in different partitions. To support communication, the "Loop to L1" network is connected to then north I/O controller to forward requests to Inner FSMs located in northern partitions, or to the south I/O controller for forwarding requests to Inner FSMs in southern partitions. If the destination Inner FSM is in the neighboring southern partition, the request is sent through the south I/O controller. If it resides in a northern partition, the request is forwarded via the north I/O controller. If the destination

Inner FSM is within the same partition, the request is delivered directly without involving any I/O controller. Finally, if the target Inner FSM is located on a different FPGA, the request is routed through the external I/O controller. Since the external I/O controller is positioned in the southern direction in this design, the message is first passed through the south I/O Controller before being sent externally to the appropriate chip.

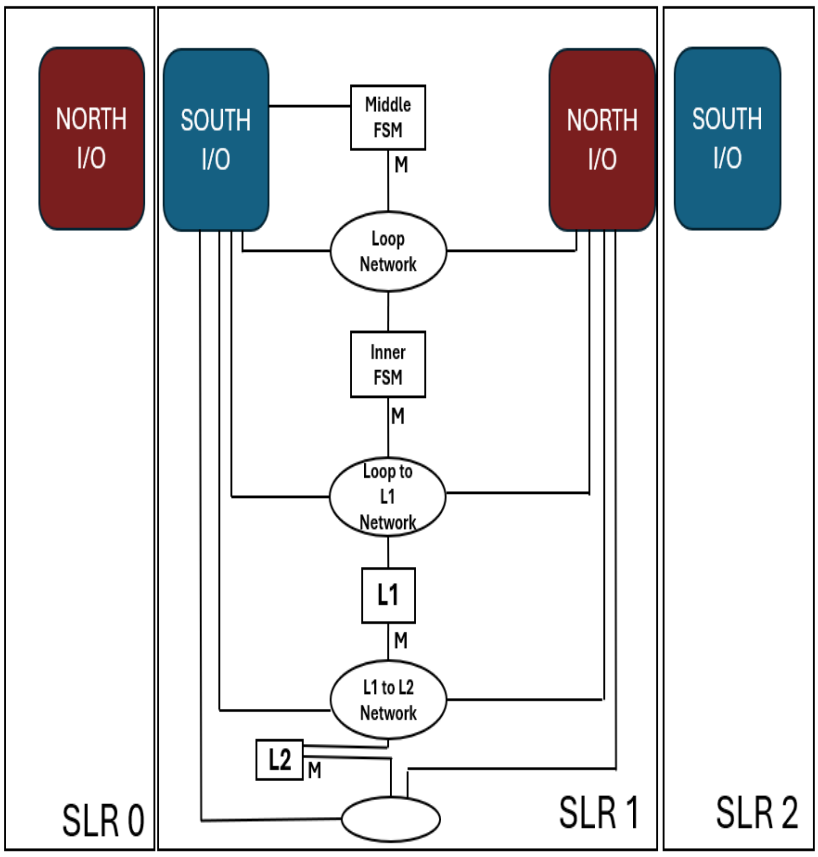

Figure 4.4: *SLR-based partitioned type 3 FPGA design networking structure in SLR 1*

A practical demonstration of the network routing mechanism specifically the translation from virtual port numbers to physical port numbers can be observed by comparing Figure 4.5 and Figure 4.6. Figure 4.5 represents the traditional FPGA-based partitioning, while Figure 4.6 illustrates the SLR-based partitioning for the same design. Both figures are captured from a test case where partition 2 must communicate with partition 1 and partition 3. In the SLR-based partitioning shown in Figure 4.6, partition 1 is located to the south of partition 2, and partition 3 is to the north. This physical layout directly influences the routing logic and I/O controller connections, showcasing how the

SLR-based network mechanism adapts port mapping and inter-partition communication accordingly.

Figure 4.5 demonstrates how the virtual-to-physical port translation function is implemented under the traditional FPGA-based partitioning model. In this example, the network module has only one directly connected component, and incoming data must be routed accordingly. Within the incoming data, a specific field indicates the target component ID, which is evaluated using the v_1 input passed to the routing function. If this target ID matches the ID of the local component in partition 2 which is "1" in this case the data is routed to physical port 0, which connects directly to that component.

However, according to the logic of FPGA-based partitioning, if the target component is not located in the current partition, the data must be forwarded to the external I/O controller. This ensures that the message reaches the correct FPGA containing the target component. Therefore, in such cases, the data is routed to physical port 1, which is connected to the external I/O controller.

```
//Find local slave port number from slave port number of original non-partitioned design for network 3
function[0:0] v2p_slave(input[1:0] v_1);


  begin

    casez (v_1) //synthesis parallel_case full_case

      // POS_INDEX = 1
      `POSITION_INDEX_2_17 : v2p_slave = `POSITION_INDEX_2_18;  // POS_INDEX = 0

      default: v2p_slave = `POSITION_INDEX_2_19; // POS_INDEX = 1

    endcase

  end

endfunction
```

Figure 4.5: *Internal data forwarding logic*

However, this type of routing logic is not suitable for SLR-based partitioning. As previously discussed, if the destination component is not located within the current partition, it may still reside in another partition within the same FPGA. In such cases, there is no need to immediately forward the data to the external I/O controller. Instead, based on the incoming virtual port number, the system can route the data to the appropriate physical port connected to either a simple I/O controller or the external I/O

controller, depending on the actual location of the target component. This allows for efficient intra-FPGA communication without unnecessarily exiting the chip.

As the find-way algorithm progresses, it collects the necessary routing information. Once this information is available, the compiler incorporates it into the internal routing logic. The resulting routing table is updated to reflect not only the components within the local partition, but also the external partitions that can be accessed via the network along with the physical ports and I/O controllers required to reach them.

```
//Find local slave port number from slave port number of
//original non-partitioned design for network 3
function[1:0] v2p_slave(input[1:0] v_1);


  begin

    casez (v_1) //synthesis parallel_case full_case

      // POS_INDEX = 0
      `POSITION_INDEX_2_5 : v2p_slave = `POSITION_INDEX_2_6;  // POS_INDEX = 2

      // POS_INDEX = 1
      `POSITION_INDEX_2_7 : v2p_slave = `POSITION_INDEX_2_8;  // POS_INDEX = 0

      default: v2p_slave = `POSITION_INDEX_2_9; // POS_INDEX = 1

    endcase

  end

endfunction
```

Figure 4.6: *Internal data forwarding logic in network elements for SLR-based partitioning*

As previously discussed, the same example is used here, but with an important enhancement: the compiler has been adapted to support three partitions located within a single FPGA chip, each distributed across different SLRs. Consequently, the internal routing logic for Partition 2 has been updated to reflect this SLR-based partitioning structure. This routing Structure can be examined in Figure 4.6.

In this updated configuration, the routing behavior is as follows:

- If a component arrives with virtual port number 0, the data is forwarded to physical port number 2, which is connected to the south I/O controller, enabling communication with partition 1.
- If a request is received with virtual port number 1, the data is directed to physical port number 0, representing an internal connection within partition 2.
- For all other virtual port numbers (i.e., not 0 or 1), the data is sent to physical port number 1, which connects to the north I/O controller, responsible for reaching components such as partition 3, where virtual port number 2 resides.

This updated logic reflects the compiler's ability to map virtual-to-physical routing based on both partition ownership and SLR locality, enabling data communication even in complex inter-FPGA partitioning situations.

4.4. SLR Based Design Hierarchy

To support the use of multiple partitions within a single FPGA chip especially when those partitions located on different SLRs the compiler's hierarchical design strategy required updates. In earlier implementations, the top-level partition modules were generated under the assumption that all components within a partition resided on a single FPGA, without considering the SLR boundaries. Each partition had its own top module named accel_top_<partition_number>. However, with the introduction of SLR-based partitioning, it became multiple partitions to exist within the same FPGA but resides on different SLRs. The original single-level top module hierarchy proved insufficient for such designs. To address this, the compiler was enhanced to support a two-level hierarchical structure.

The compiler introduces a hierarchical module naming convention to support SLR-based partitioning. For each FPGA chip in the system, a top-level module named accel0 is generated. This module serves as the global controller for that specific FPGA and directly connects to the CL SDE module on the PCIe interface side, enabling communication with the host system. Additionally, accel0 is responsible for handling FPGA-to-FPGA communication by interfacing with other accel0 modules across the system and coordinating data exchange between locally instantiated accel1 modules. These accel1 modules represent SLR-level partitions within the FPGA and are connected via directionally named signal wires (e.g., north, south, east, west) that reflect the physical layout of the FPGA.

At the second level of hierarchy, each partition is encapsulated within a module named in the format accel1_top_<FpgaChipNumber>_<PartitionNumber>. Each of these accel1_top modules serves as the top module for a specific SLR region, managing the components and communication logic localized to that SLR.

To support structured connectivity between partitions, the compiler maintains a dedicated routing database. This database tracks all inter-partition connections referred to as “loose wires” and assigns directional routing paths (e.g., north-south across SLRs) to ensure proper data delivery.

As an example, consider a single FPGA containing three partitions mapped across three SLRs (SLR 0, 1, and 2). partition 1, located in SLR 0, is encapsulated in accel_top_1_1 and includes a simple I/O controller connected northward to partition 2 via accel_top_1_2. partition 2, defined in accel_top_1_2, includes both a north I/O controller (linked to partition 3 via accel_top_1_3) and a south I/O controller (connected back to partition 1).partition 3, encapsulated in accel_top_1_3, receives data from partition 2 through its south I/O controller. This modular structure enables scalable communication within the FPGA, across SLRs, and via the accel0 hierarchy between multiple FPGA chips.

This overall design flow is depicted in Figure 4.7, which includes a simplified RTL fragment automatically generated by the compiler to realize the communication between these partitions. The Verilog code demonstrates how modules are instantiated and connected through directional signal wires to implement data flow between the three SLR-based partitions.

As shown in Figure 4.7, Verilog macros such as `out_north or `in_south are dynamically defined by the compiler at runtime. These macros correspond to wire widths for inter-partition communication, and they are calculated when generating the design to accommodate the final topology and IPs used in the design.

The reason these values must be computed dynamically is that the presence of optional modules such as a bus width converter IP can directly impact the required bit widths of the communication interfaces. For example, if a bus width converter is inserted to prevent SLL violations, the wire sizes for input and output interfaces will change based on the converter’s configuration. As a result, the IO controllers must be emitted with awareness of these changes, which necessitates runtime computation.

```
module accel0_top_1 (
    input reset,
    input clock
);

  // Interconnect between Partition 1 and 2
  wire [`out_north_1-1:0] outputData_north_1_south_2;
  wire [`in_north_1-1:0] inputData_north_1_south_2;
  //.................

  // Interconnect between Partition 2 and 3
  wire [`out_north_2-1:0] outputData_north_2_south_3;
  wire [`in_north_2-1:0] inputData_north_2_south_3;
  //.................
  // Partition 3
  accel1_top_1_3 U_accel1_top_2 (
    .inputData_south_3(outputData_north_2_south_3),
    .outputData_south_3(inputData_north_2_south_3)
     //.................
  );

  // Partition 2
  accel1_top_1_2 U_accel1_top_1 (
    .outputData_north_2(outputData_north_2_south_3),
    .inputData_north_2(inputData_north_2_south_3),
    .inputData_south_2(outputData_north_1_south_2),
    .outputData_south_2(inputData_north_1_south_2)
     //.................
  );

  // Partition 1
  accel1_top_1_1 U_accel1_top_0 (
    .outputData_north_1(outputData_north_1_south_2),
    .inputData_north_1(inputData_north_1_south_2)
    //.................
  );

endmodule
```

Figure 4.7: *Top-Level partition initialization code for a (1,3,1) 3 partition SLR configuration*

In addition, for a given communication type (e.g., Master/Slave), the inter-SLR connection topology is determined using the lower neighbor search algorithm. This algorithm starts from the partition with the maximum partition number in the FPGA and iteratively finds the lower numbered neighbor partitions to establish communication links. The higher numbered partition acts as the master, and the lower numbered neighbor acts as the slave. This transition continues iteratively until all partitions in the FPGA are connected according to the defined communication pattern. The algorithm can also be adapted to discover neighbors not only in one dimension but also in the x, y, and z directions, making it suitable for 2-dimension and 3-dimension FPGA architectures so that multi-die FPGA systems.

5. EXPERIMENTAL RESULTS

This section presents an analysis of the SLR-based partitioned design tests generated by GLOBAL's HLS compiler which we mention in Section 3, primarily covered in two subsections. First, we provide an overview of the Verilator-based RTL simulation tests run on a local VMware virtual machine running Ubuntu 20.04, equipped with 32GB of RAM and an Intel i7-13620H CPU. The second part focuses on GLOBAL's HLS compiler, which generates RTL designs with improved SLR-based partitioning support. These RTL files are then used to create FPGA images through the Vivado 2024.1 implementation environment. The resulting images are deployed and executed on the AWS F1 cloud environment for selected tests.

To verify the correctness of the compiler updated with the SLR-based partitioning structure, selected test suites were used, including Greatest common divisor (GCD), Adaptive differential pulse code modulation (ADPCM), Advanced encryption standard (AES), and function tests. The content of these tests can be considered independently of this work. The primary goal of this study is to demonstrate that HLS RTL designs generated using the SLR-based partitioning approach function correctly across a variety of random test cases.

5.1. Verilator-Based Functional Simulation

To demonstrate the scalability of the SLR-based partitioning methodology, an 8-partition configuration was selected for testing. This setup can represent different dimensional mappings such as (1,8,1), (2,4,1), and (2,2,2), allowing for implementation the (x, y, z) partitioning space. Also, a one-dimensional implementation using the (1,3,1) configuration is also tested. This specific setup defines the actual physical layout of the FPGA device Xilinx VU9P which contains three SLRs (SLR0, SLR1, and SLR2) along the y-dimension.

In our Verilator-based RTL simulation, time progresses in unitless steps using a counter named main time. Each increment of main time corresponds to one evaluation cycle of the simulation loop. The system clock is toggled every 5 time units. For instance, the clock remains low for five consecutive time steps, then high for the next five, and so on. The reset signal is initially asserted and then deasserted after a specific number of time steps, simulating the release of reset at runtime. This setup allows for systematic control and observation of the module behavior across defined clock and reset phases.

5.1.1. One-dimensional

Figure 5.1 shows a test structure containing three partitions placed on the SLRs. Based on the user-specified SLR-per-chip pragma (1,3,1,0), the compiler assumes that there are 1 SLR in the x-direction, 3 in the y-direction, and 1 in the z-direction. Using this partitioning strategy, the compiler assigns partitions 1, 2, and 3 to SLR0, SLR1, and SLR2, respectively, as shown in Figure 5.1. Communication between these partitions is managed by the internal directional I/O controllers located on the north and south sides, as determined by the HLS compiler. In addition, a single external I/O controller manages communication outside the FPGA.

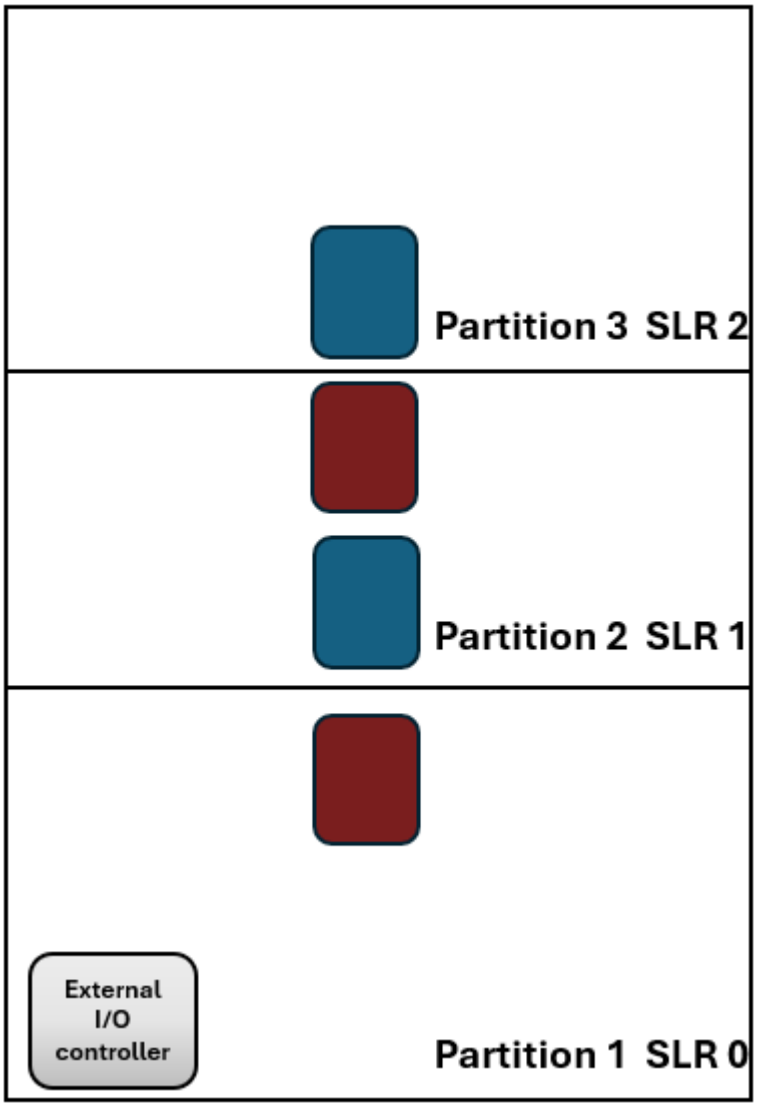

Figure 5.1: *I/O Controller placement for a (1,3,1,0) SLR configuration with 3 partitions*

Table 5.1: *Verilator-based simulation time comparison between FPGA and SLR-based partitioned designs (3-partitions configuration: SLR = (1,3,1))*

Test Names	FPGA-based Partitioned Design (3 Machines)	SLR-based Partitioned Design (1 Machine)	Speed-up
GCD	202,880 t	1,368 t	148.45x
AES_DEC	43,326 t	16,217 t	2.67x
ADPCM_DEC	160,386 t	3,666 t	43.74x
ADPCM_ENC	156,209 t	5,049 t	30.94x
Function	215,738 t	986 t	218.54x

Table 5.1 compares the simulation times between FPGA-based partitioned designs running on three separate machines and SLR-based partitioned designs running on a single machine for various tests. This means that when the compiler uses the SLR-based partitioning method, it relies on the SLR-per-chip pragma to determine the number of SLRs in the FPGA. If the pragma indicates a single SLR, each partition is assigned to a different machine. However, if the pragma specifies three SLRs, all three partitions are placed within a single FPGA. For these local tests, the virtual machines simulate the FPGA machines. As observed in Table 5.1, significant speed-up is achieved with SLR-based partitioning; however, the actual performance improvement may vary depending on the test case. Some tests involve frequent communication between partitions either between FPGA partitions or between the FPGA and external systems while others require less inter-partition communication. These differences can lead to variations in the observed speed-up across different test cases. Despite this, the main conclusion from the table is that SLR-based partitioning effectively accelerates simulation, and similar speed-ups are expected in real FPGA implementations.

Figure 5.2 illustrates a test structure consisting of eight partitions distributed across the SLRs. This configuration is based on the user-specified SLR-per-chip pragma (1,8,1,0), which, as previously explained, was chosen to demonstrate the scalability of the SLR-based partitioning method. In this test, the eight partitions are placed along the Y-direction across eight SLRs, as shown in Figure 5.2. Communication between these partitions is managed by internal directional I/O controllers located on the north and south sides, as determined by the HLS compiler. Additionally, a single external I/O controller handles communication outside the FPGA.

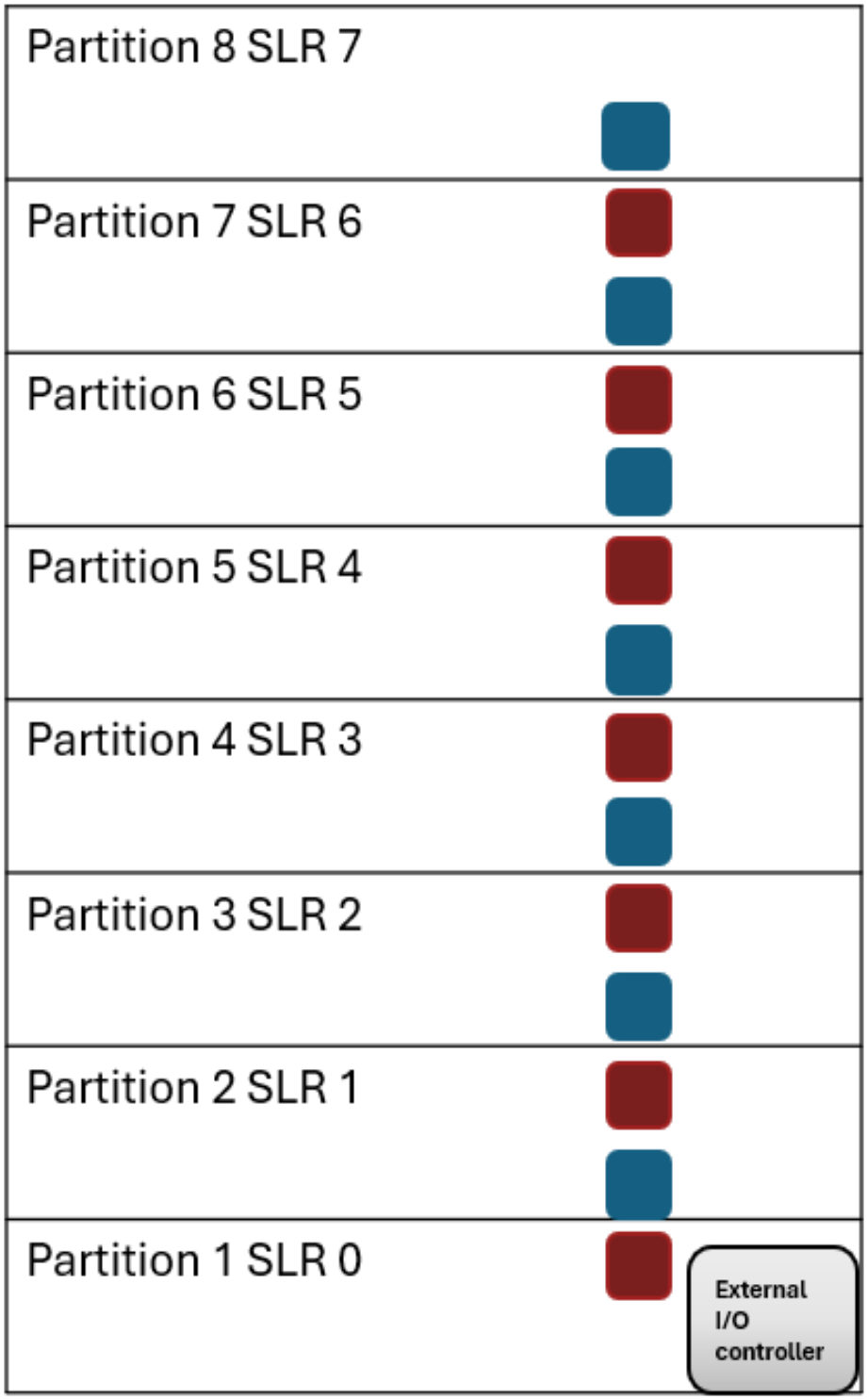

Figure 5.2: *I/O controller placement for a (1,8,1,0) SLR configuration with 8 partitions*

Table 5.2: *Verilator-Based simulation time comparison between FPGA and SLR-based partitioned designs (8-partitions configuration: SLR = (1,8,1))*

Test Names	FPGA-based Partitioned Design (8 Machines)	SLR-based Partitioned Design (1 Machine)	Speed-up
GCD	646,419 t	2,166 t	298.50x
AES_Decrypt	130,315 t	12,345 t	10.56x
ADPCM_DEC	525,630 t	8,256 t	63.63x
ADPCM_ENC	472,928 t	9,630 t	49.09x
FUNCTION	719,400 t	1,832 t	392.94x

When comparing the performance of the 3-partition system in Table 5.1 with the 8-partition system in Table 5.2, a noticeable increase in simulation time is observed in the latter. This is mainly due to the higher communication overhead between the eight separate machines used in the traditional FPGA-based partitioning setup which is refer each partition will be locate different machine. As the number of partitions grows, the volume of machine-to-machine data exchange increases accordingly, introducing latency and slowing down the overall simulation process. In contrast, the SLR-based partitioning approach eliminates this bottleneck by executing all partitions within a single machine, utilizing multiple SLRs. By relying on the SLR-per-chip pragma, the compiler maps the partitions to different SLRs inside the same machine, allowing internal communication to occur without involving external machines. This significantly reduces the simulation time by avoiding the delays associated with inter-machine communication. As shown in Table 5.2, despite the increase in the number of partitions, the SLR-based design completes the simulations much faster than the multi-machine FPGA-based design. This highlights the scalability and efficiency of the SLR-based partitioning method, particularly for complex supercomputer designs. It proves that even with a high number of partitions, performance gains can still be achieved when communication remains internal to the machine.

5.1.2. Two-dimensional

Figure 5.3 shows the partition layout on SLRs for the 8-partition test systems, based on the user-defined SLR-per-chip pragma set to (2,4,1,0). This configuration specifies partitioning across 2 SLRs in the x direction and 4 SLRs in the y direction, with the master SLR index set to 0. As a result, the external I/O controller is automatically placed on SLR 0, where Partition 1 resides. The goal of this test is to demonstrate that different types of partition layouts across the x, y, and z dimensions can be handled by the compiler. Given the 2×4 partition grid, the compiler places the partitions in a row-major order across the available SLRs. It then automatically generates the necessary directional I/O controllers to support communication between neighboring partitions. Specifically, north/south and east/west I/O controllers are instantiated based on the relative positions of each partition in the grid. Figure 5.3 also illustrates the resulting I/O controller placement, clearly showing how the compiler constructs and assigns the internal communication logic. This confirms that the SLR-based partitioning framework can flexibly adapt to multidimensional layouts and still maintain proper inter-partition connectivity and external communication handling.

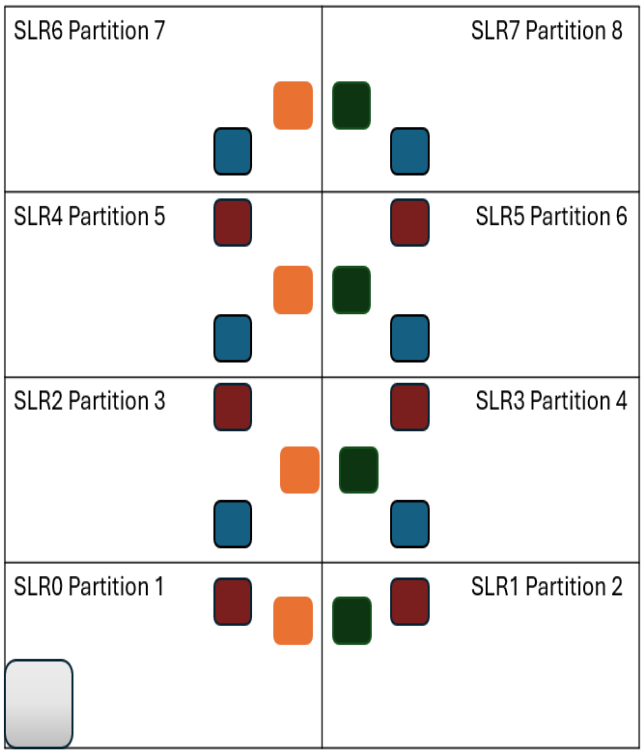

Figure 5.3: *I/O controller placement for a (2,4,1,0) SLR configuration with 8 partitions*

Table 5.3*: Verilator-Based simulation time comparison between FPGA and SLR-based partitioned designs (8-partitions configuration: SLR = (2,4,1))*

Test Names	FPGA-based Partitioned Design (8 Machines)	SLR-based Partitioned Design (1 Machine)	Speed-up
GCD	646,419 t	2,064 t	313.39x
AES_DEC	130,315 t	12,015 t	10.85x
ADPCM_DEC	525,630 t	8,512 t	61.84x
ADPCM_ENC	472,928 t	9,414 t	50.30x
Function	719,400 t	1,721 t	418.53x

Table 5.3 shows the Verilator-based simulation time results for an 8-partition design using the SLR configuration (2,4,1,0). In this configuration, the FPGA-based partitioned design is simulated across 8 separate machines, whereas the SLR-based version runs entirely on a single machine. Despite the higher number of partitions, the SLR-based approach consistently achieves significant reductions in simulation time by keeping all inter-partition communication internal to the machine.

This layout enables the compiler to instantiate both north/south and east/west directional I/O controllers, allowing communication between neighboring partitions without involving external I/O controller. For instance, the GCD benchmark sees a dramatic speed-up from 646,419 time units down to 2,064 over 313× faster. Similar improvements are observed across other benchmarks, such as ADPCM_DEC and Function, which achieve 61× and 418× speed-ups, respectively. These results confirm that the compiler can handle multidimensional partition layouts and that SLR-based partitioning is highly scalable and efficient for simulation environment.

5.1.3. Three-dimensional

The three-dimensional SLR layout setup is defined by the user SLR-per-chip pragma (2,2,2,0), which arranges 2 partitions along each of the x, y, and z dimensions, with the master SLR located at index 0. As illustrated in Figure 5.4, Partition 1 is placed within SLR 0, hosting the external I/O controller responsible for communication outside the FPGA device. Internal communication among partitions is facilitated by a network of directional I/O controllers instantiated by the compiler, covering the up/down (z-direction), north/south (y-direction), and east/west (x-direction) axes, thereby enabling efficient 3-dimensional message routing. For example, when Partition 8 needs to send

data externally, the message is routed through multiple hops: first from Partition 8 to Partition 4 via up/down controllers, then from Partition 4 to Partition 2 using north/south controllers, and finally from Partition 2 to Partition 1 through east/west controllers, where it exits the FPGA through the external I/O controller in SLR 0.

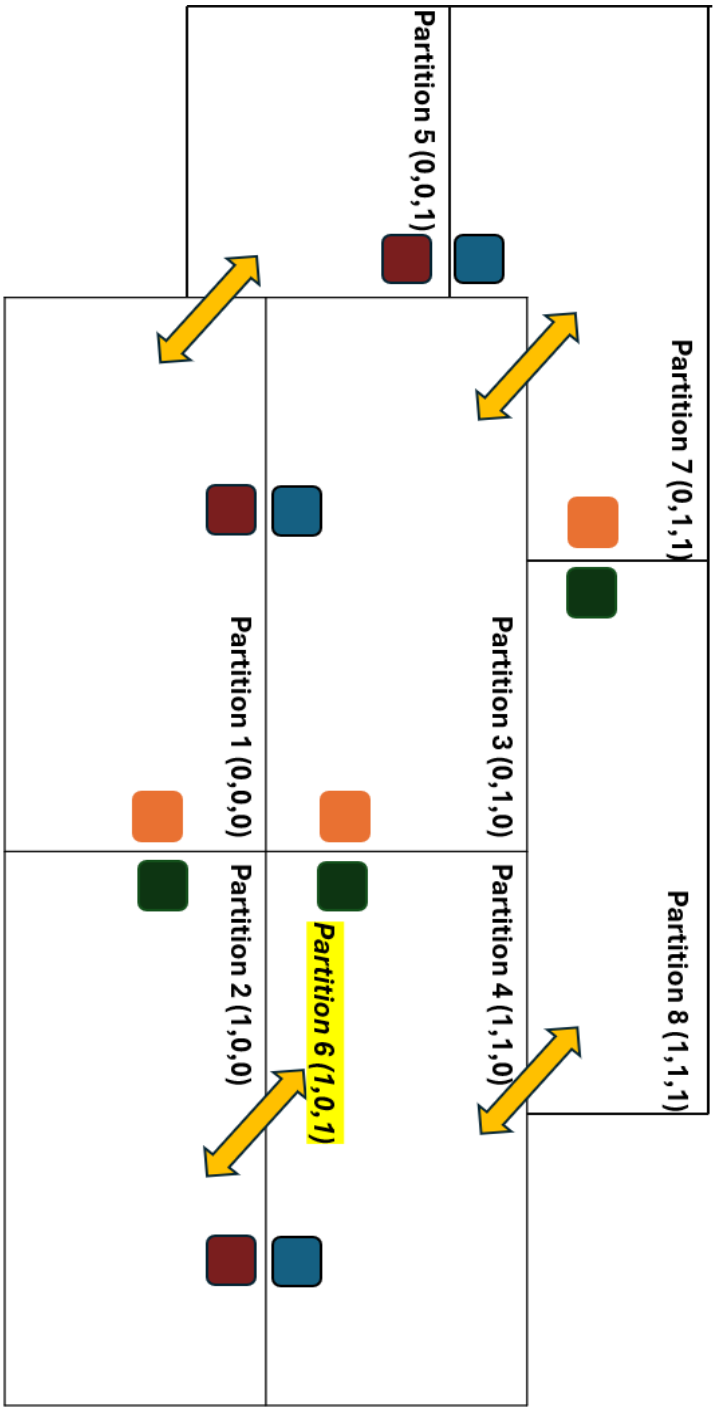

Figure 5.4: *I/O controller placement for a (2,2,2,0) SLR configuration with 8 partitions*

```
// Network up_4_down_8
wire outputReq_up_4_down_8;
wire outputAck_up_4_down_8;
wire [`out_up_4-1:0] outputData_up_4_down_8;
wire inputReq_up_4_down_8;
wire inputAck_up_4_down_8;
wire [`in_up_4-1:0] inputData_up_4_down_8;

accel1_top_1_8
U_accel1_top_7 (

   .reset(reset)
  ,.clock(clock)

  //request wires
  ,.inputReq_down_8(outputReq_up_4_down_8)
  ,.inputAck_down_8(outputAck_up_4_down_8)
  ,.inputData_down_8(outputData_up_4_down_8)

  //response wires
  ,.outputReq_down_8(inputReq_up_4_down_8)
  ,.outputAck_down_8(inputAck_up_4_down_8)
  ,.outputData_down_8(inputData_up_4_down_8)

  //request wires
  ,.inputReq_west_8(outputReq_east_7_west_8)
  ,.inputAck_west_8(outputAck_east_7_west_8)
  ,.inputData_west_8(outputData_east_7_west_8)

  //response wires
  ,.outputReq_west_8(inputReq_east_7_west_8)
  ,.outputAck_west_8(inputAck_east_7_west_8)
  ,.outputData_west_8(inputData_east_7_west_8)

  //request wires
  ,.inputReq_south_8(outputReq_north_6_south_8)
  ,.inputAck_south_8(outputAck_north_6_south_8)
  ,.inputData_south_8(outputData_north_6_south_8)

  //response wires
  ,.outputReq_south_8(inputReq_north_6_south_8)
  ,.outputAck_south_8(inputAck_north_6_south_8)
  ,.outputData_south_8(inputData_north_6_south_8)
```

Figure 5.5: *Partition 8 initiation routine based on it is neighbor partitions*

As illustrated in Figure 5.5, the instantiation code of Partition 8 exemplifies the communication infrastructure connecting multiple partitions within the FPGA design. Partition 8 communicates with Partition 4 along the z-axis, where signals flow from the up I/O controller of partition 4 to the down I/O controller of partition 8. This communication uses a consistent and descriptive naming convention for signal declarations, structured as <destination partition output direction from source partition>_<source partition ID>_<soruce partition output direction from destination partition>_<destination partition ID>. The Verilog module instantiation code shared in the figure demonstrates this naming scheme in practice.

For example, the wire outputReq_up_4_down_8 represents a request signal that originates from the up I/O controller of partition 4 and is sent to the down I/O controller of partition 8. Furthermore, partition 8 is connected to all its neighbouring partitions, including partition 7 (west) and partition 6 (south), following the same directional and partition-based naming convention. This structure ensures clarity in the signal mapping and enforces modularity across partition boundaries. Each pair of partitions communicates using bidirectional handshaking signals request, acknowledgment, and data routed through their respective directional I/O controllers.

Additionally, Table 5.4 presents a comparison of simulation times between two deployment scenarios. In the first scenario, each of the 8 partitions is simulated on a separate machine, representing a distributed simulation model. This setup reflects a real-world use case where each FPGA partition operates independently on different physical devices. In the second scenario, all 8 partitions are simulated on a single machine, communicating through the SLR-based structure described above. This setup mimics the behavior of a hypothetical 3-dimensinoal-stacked FPGA, where inter-partition communication would occur in a similar manner to the simulated SLR connections. This comparison highlights the differences in communication latency and overall simulation performance between distributed and monolithic architectures.

Table 5.4: *Verilator-Based simulation time comparison between FPGA and SLR-based partitioned designs (8-partitions configuration: SLR = (2,2,2))*

Test Names	*FPGA-based Partitioned Design (8 Machine)*	*SLR-based Partitioned Design (1 Machine)*	Speed-up
GCD	646419 t	2019 t	320.61x
AES_DEC	130315 t	11961 t	10.89x

5.2. FPGA Implementation of SLR based Partitioning

This section presents a proof-of-concept implementation of SLR-based partitioning on the VU9P FPGA within the AWS cloud environment. The test is conducted using the 3 different applications named of ADPCM, GCD and function. Each configured with the SLR-per-chip pragma (1,3,1) resulting in a total of 3 partitions on one FPGA only. Before presenting the test results, this section also highlights an important optimization in the FPGA implementation: the strategic use of specific resource instances within the CL SDE design, namely the SDE IP, PCIM, and PCIS modules. These components operate in conjunction with the encrypted SHELL logic, all of which were introduced in Section 2.5.2 as part of the overall CL SDE architecture.

5.2.1. Placement optimizations

Figure 5.6 illustrates the top-level module hierarchy of the implementation design, structured around the CL SDE architecture. The overall system is composed of multiple submodules, including the encrypted AWS Shell, PCIM, PCIS, and SDE IP blocks. Since the AWS Shell is encrypted, its internal nets and modules are displayed as <hidden> in the hierarchy. The primary functionality of the design is organized as follows: the encrypted Shell module manages communication between the FPGA and external systems via PCIe. The PCIM (AXI Master) and PCIS (AXI Slave) modules interface directly with the encrypted Shell, enabling data exchange over AXI protocols. The SDE IP block receives incoming PCIe data, translates it into an AXI Stream format, and forwards it to the accel0_top_module, which contains the core RTL logic generated by the HLS compiler. This structure highlights the integration of standard AWS platform modules with the custom HLS-generated hardware design.

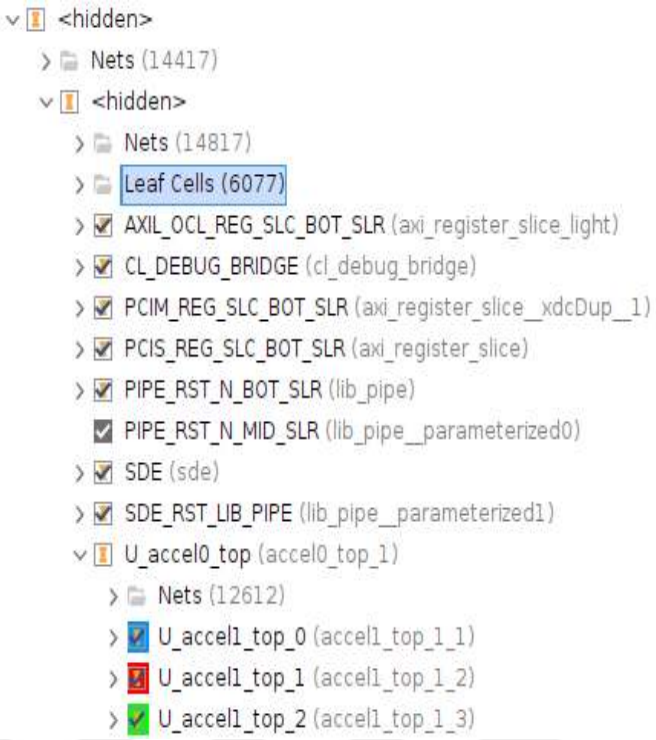

Figure 5.6: *General top-level module hierarchy of the implementation design*

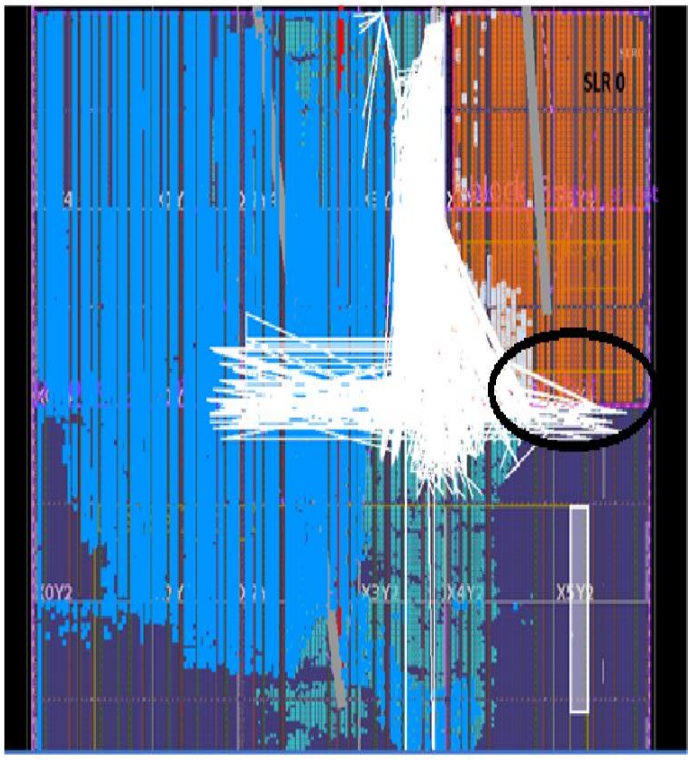

Figure 5.7: *The connectivity of the PCIM IP with encrypted SHELL logic*

Figure 5.7 illustrates the placement and connectivity of the PCIM IP within the design and its interaction with the encrypted AWS Shell logic. The white wires show the exact connection between encrypted Shell logic and CL SDE design. In the figure, the encrypted Shell components are highlighted in orange, while the regions connected to the PCIM IP are shown in dark blue and blue. The visualization focuses specifically on SLR0 to emphasize how the PCIM IP interfaces with static encrypted Shell. In the bottom-right corner of SLR0, the static outgoing PCIM nets belonging to the encrypted Shell are depicted in orange and further emphasized with a black circle for clarity.

Since the encrypted Shell has a fixed and static implementation, the placement of the PCIM IP must align precisely with its corresponding interfaces. To ensure this, the following placement directive is applied:

- add_cells_to_pblock [get_pblocks pblock_CL_bot][get_cells -quiet -hierarchical -filter {NAME =~ WRAPPER_INST/CL/PCIM_REG_SLC_BOT_SLR/*}].

This command constrains the PCIM logic to the lower portion of the device (SLR0). As shown in the figure, the white interconnect wires between the PCIM IP and the encrypted Shell are concentrated within SLR0, which is commonly referred to as the bottom (BOT) SLR. This placement strategy ensures efficient communication and compliance with the AWS platform's static design requirements.

Figure 5.8 illustrates the connectivity of the SDE IP and its interaction with the encrypted AWS Shell within the design. As in Figure 5.7, the encrypted Shell components are highlighted in orange, while the SDE-connected regions are marked in dark blue and blue. The diagram focuses on SLR0, emphasizing the placement and routing of the SDE IP relative to the static Shell infrastructure. The static outgoing nets associated with the SDE IP originating from the encrypted Shell are clearly visible in the orange region of SLR0 and are further highlighted with a black circle for clarity. Due to the fixed and static nature of the encrypted Shell, the placement of the SDE IP must precisely align with its predefined interface points. As a result, the SDE IP is constrained to SLR0, following placement directives similar to those used for the PCIM block.

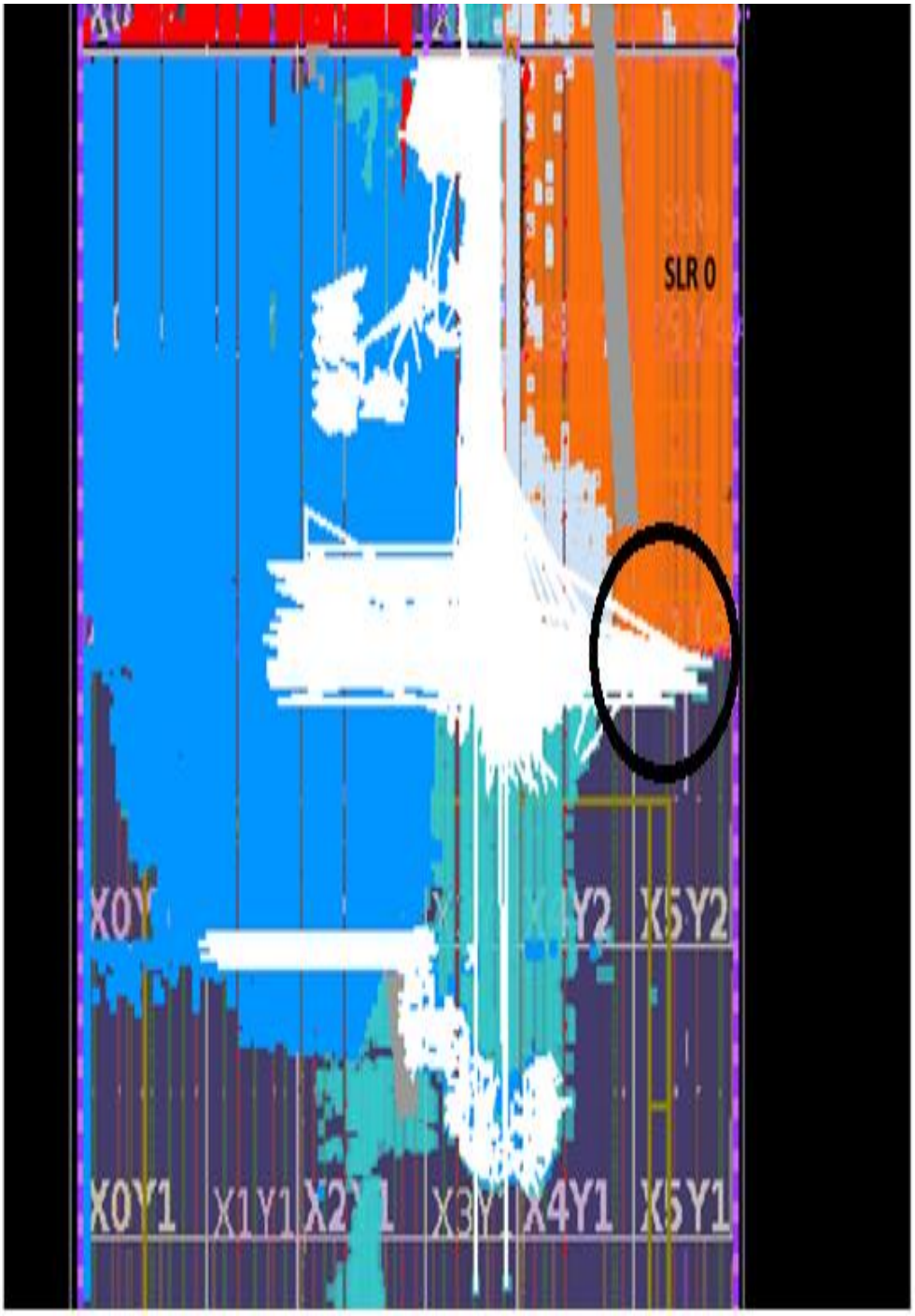

Figure 5.8: *The connectivity of the SDE IP with encrypted SHELL logic*

As depicted in Figure 5.8, the white wire connections between the SDE IP and the encrypted SHELL remain primarily within SLR0, maintaining the required proximity and interaction with the static shell infrastructure.

Figure 5.9 illustrates the connectivity of the PCIS IP with the encrypted SHELL logic, highlighting a trade-off made in the current implementation. While the main logic is emulated within SLR0 to take advantage of optimal placement for the PCIM and SDE IPs, the PCIS IP poses a slight placement challenge. As shown in the figure, the endpoints of the white wire connections for the PCIS IP remain within SLR0; however, the corresponding SHELL connections originate in SLR1 and traverse into SLR0. This cross-SLR routing introduces potential latency and complexity in timing closure. To mitigate this, one possible improvement would be the insertion of an AXI register slice between the SHELL connections and the PCIS IP interface in SLR0. This approach could help isolate timing paths across SLR boundaries and improve overall robustness. Although this enhancement has not yet been implemented, it represents a potential future upgrade that could be quickly integrated if needed.

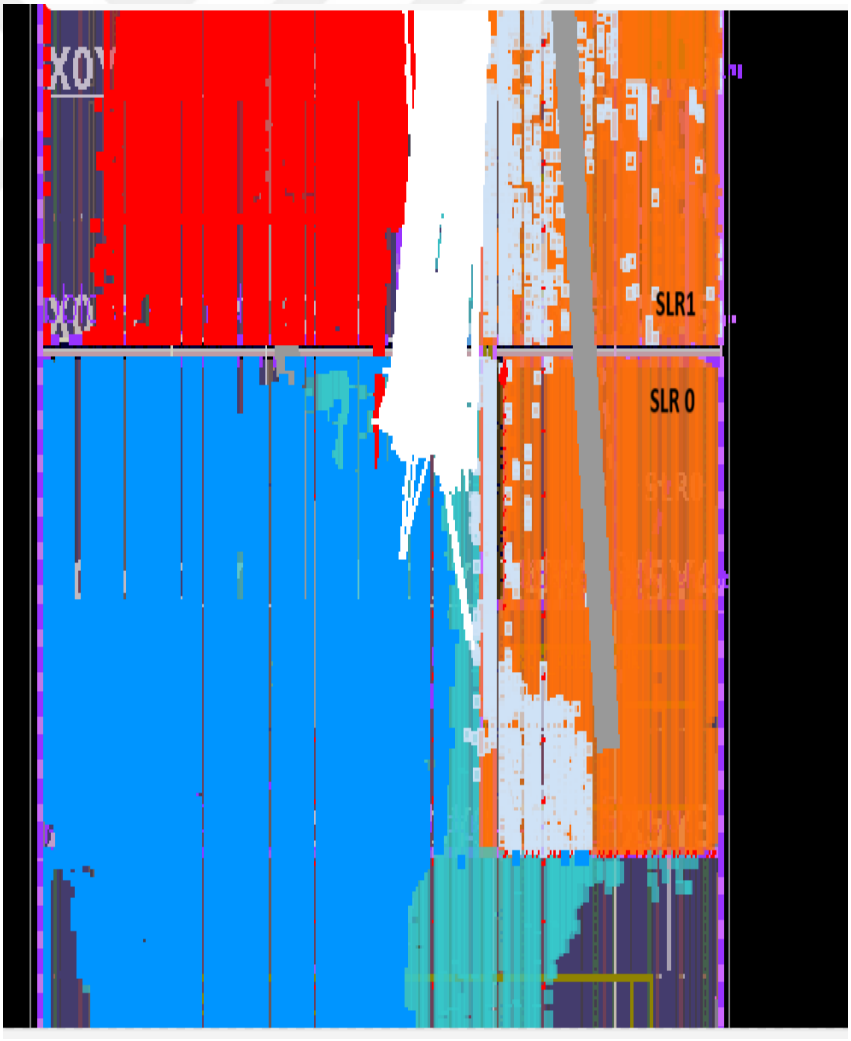

Figure 5.9: *The connectivity of the PCIS IP with encrypted SHELL logic*

5.2.2. SLR-based partitioned GCD Implementation

The test suites were deployed on the AWS FPGA platform using the Xilinx VU9P device. The SLR-per-chip pragma was configured as (1,3,1,0), directing the compiler to generate three partitions along the Y-dimension of a single FPGA. Given a total partition count of three, all partitions were mapped onto a single FPGA device. Although this setup does not include direct performance time comparisons, it effectively serves as a proof-of-concept, demonstrating the feasibility and correctness of implementing SLR-based partitioning within a FPGA environment.

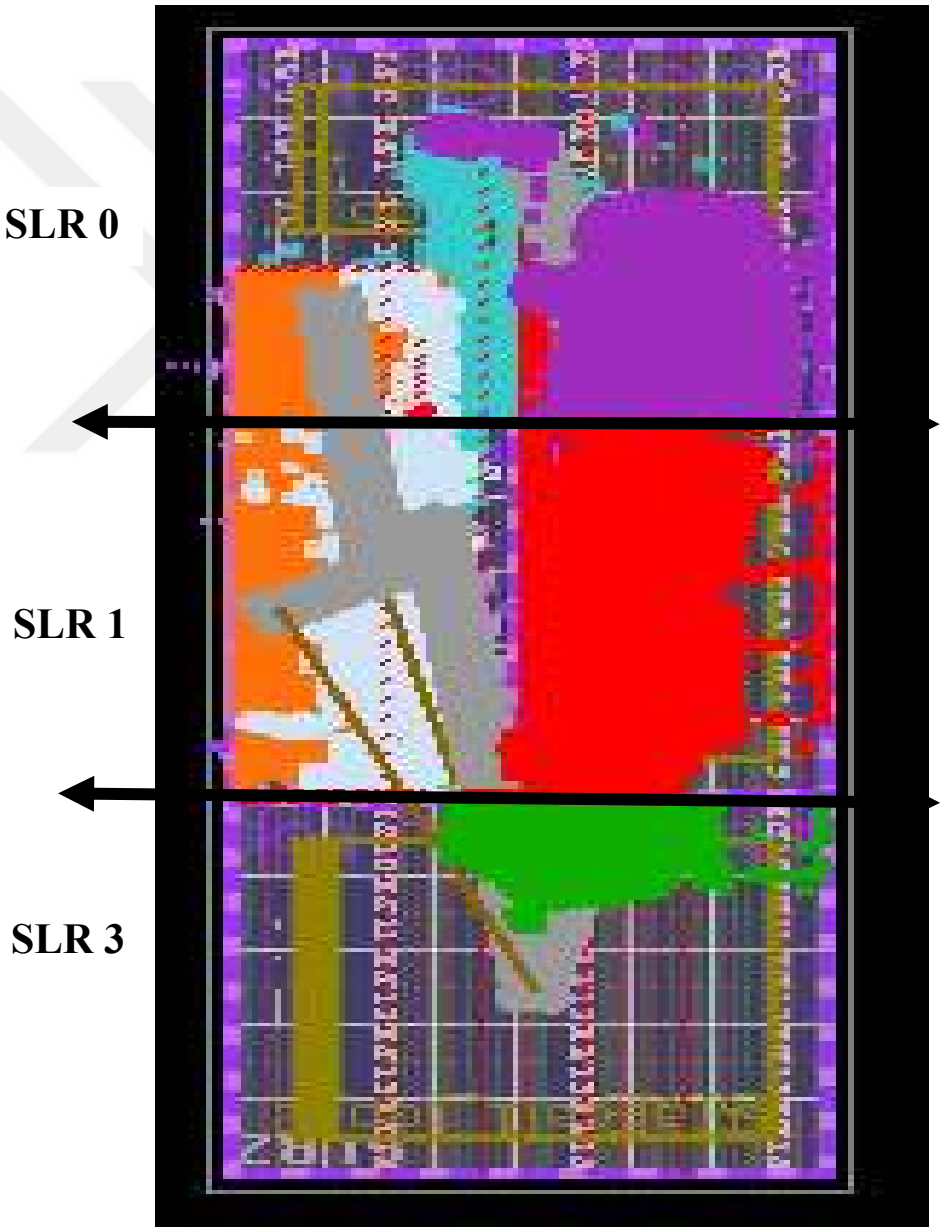

Figure 5.10: *Test suit GCD FPGA implementation*

Figure 5.10 shows the layout of the GCD test suite on the FPGA. The green cells correspond to partition 3, red cells represent partition 2, purple cells indicate partition 1, and orange cells denote the AWS Shell design components. This visualization clearly shows the physical distribution of partitions and shell modules across the FPGA fabric and also proves that the partitions are correctly located within the desired SLR regions

Table 5.5: *SLR connectivity of test suit GCD*

FROM / TO	*SLR2*	*SLR1*	*SLR0*
SLR2	0	2276	4
SLR1	2251	0	3798
SLR0	75	3402	0

Table 5.5 presents the SLR connectivity for the GCD test suite, showing the number of signal crossings between the three SLR regions. The values indicate the volume of inter-SLR communication required for this design. As discussed in Section 4.2.1, the control and management of these SLR crossing wires are handled by the simple I/O controller module.

Table 5.6 presents the resource usage for the GCD test suite, specifically focusing on CLB utilization across the three SLRs. The compiler-generated partitions partition 1, 2, and 3 are distributed across SLR0, SLR1, and SLR2, respectively, with approximate usage around 20% for each SLR. However, SLR1 shows a significantly higher utilization at approximately 71%. This imbalance is expected, as SLR1 also hosts the encrypted AWS Shell and the CL_SDE logic. As previously detailed in Table 2.2, the encrypted Shell alone accounts for about 33% of the resource usage within SLR1. The data confirms that the compiler achieves a fairly balanced resource allocation for the user-defined partitions themselves across the three SLRs.

Table 5.6: *Resource usage of a test suit GCD*

Site Type	*SLR0*	*SLR1*	*SLR2*	*SLR0 %*	*SLR1 %*	*SLR2 %*
CLB	11035	35261	8980	22.40	71.58	18.23

5.2.3. SLR-based partitioned ADPCM Implementation

Figure 5.11 test suit ADPCM implementation on VU9P FPGA, the green-colored cells represent RTL design units for partition 3, the red-colored cells correspond to partition 2, and the purple-colored cells indicate partition 1. Additionally, the orange-colored cells highlight the AWS Shell design units.

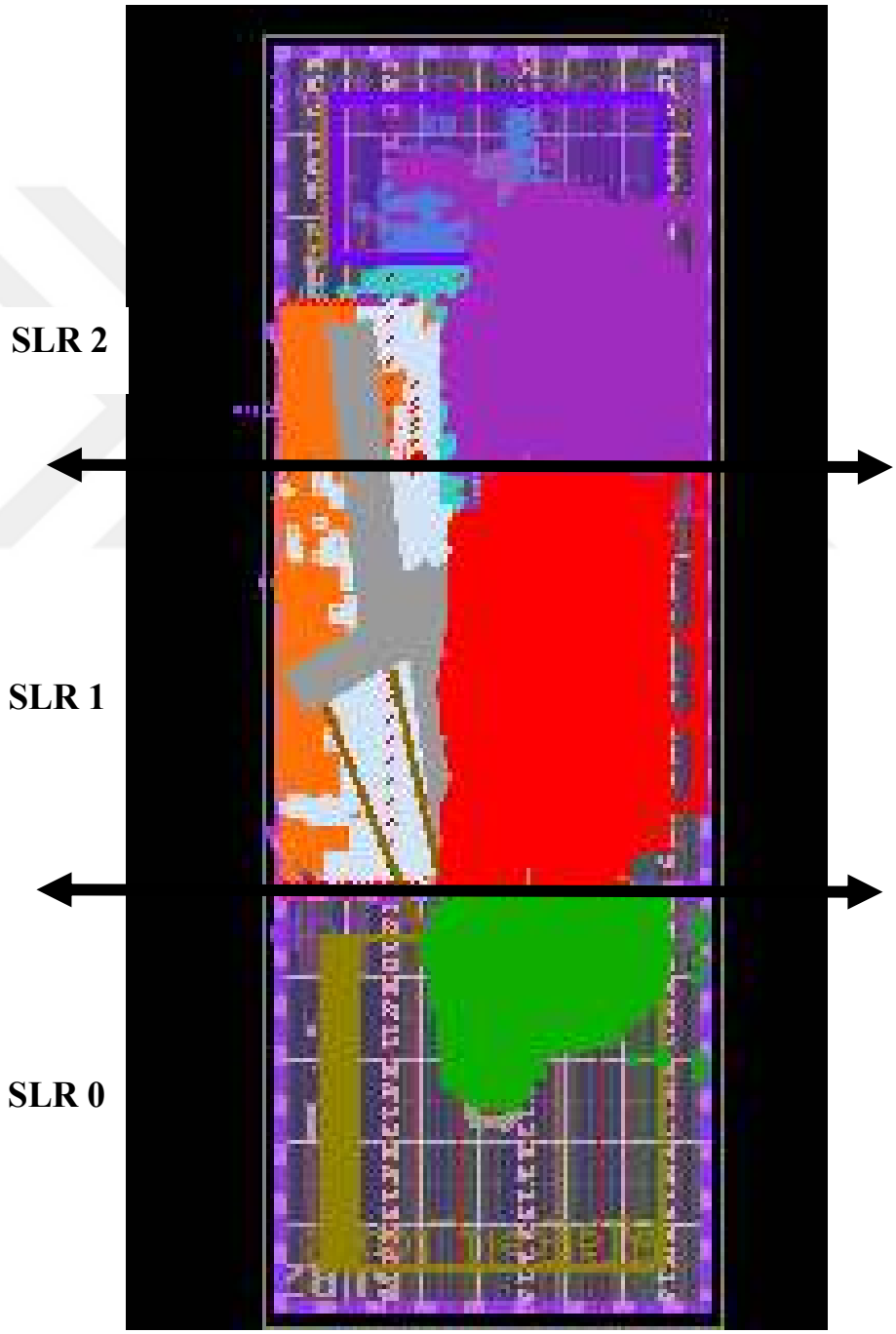

Figure 5.11: *Test suit ADPCM FPGA implementation*

Table 5.7 presents the SLR connectivity matrix for the ADPCM test suite, showing the number of signal crossings between SLR0, SLR1, and SLR2. These values reflect the amount of inter-SLR communication in the design. As discussed in Section 4.2.1, the simple I/O controller module is responsible for managing and routing these crossing signals. It ensures that communication across SLR boundaries is handled does not exceed the available routing resources.

Table 5.7: *SLR connectivity of test suit ADPCM*

FROM / TO	*SLR2*	*SLR1*	*SLR0*
SLR2	0	1726	75
SLR1	1872	0	3521
SLR0	35	3601	0

Table 5.8 summarizes the CLB resource usage for the ADPCM test suite across the three SLRs. Partitions 1, 2, and 3 are mapped to SLR0, SLR1, and SLR2, respectively. While the partition logic is distributed across all three SLRs, SLR1 again exhibits the highest utilization at approximately 76%. This aligns with expectations, as SLR1 hosts the encrypted AWS Shell, which, as noted in Table 2.2, contributes roughly 33% of its CLB usage. Additionally, SLR0 contains not only the logic for partition 1 but also the CL_SDE infrastructure, which explains its relatively high utilization of 56.84%. Despite these shared resources, the user-defined partition logic maintains a reasonable distribution across the device. SLR2, which contains only partition logic, shows 19.41% usage, further confirming that the SLR-based partitioning strategy achieves effective and balanced placement even under non-uniform logic loads.

Table 5.8: *Resource usage of a test suit ADPCM*

Site Type	*SLR0*	*SLR1*	*SLR2*	*SLR0 %*	*SLR1 %*	*SLR2 %*
CLB	27998	37634	9561	56.84	76.40	19.41

5.2.4. SLR-based partitioned Function Implementation

Figure 5.12 illustrates the spatial layout of the Function test suite on the FPGA. The green-colored regions correspond to partition 3, red indicates partition 2, and purple represents Partition 1. The orange-colored areas show the AWS Shell design modules. This visualization highlights how the compiler has physically distributed the partition logic across the FPGA and confirms that the partitions have been correctly mapped to their designated SLR regions.

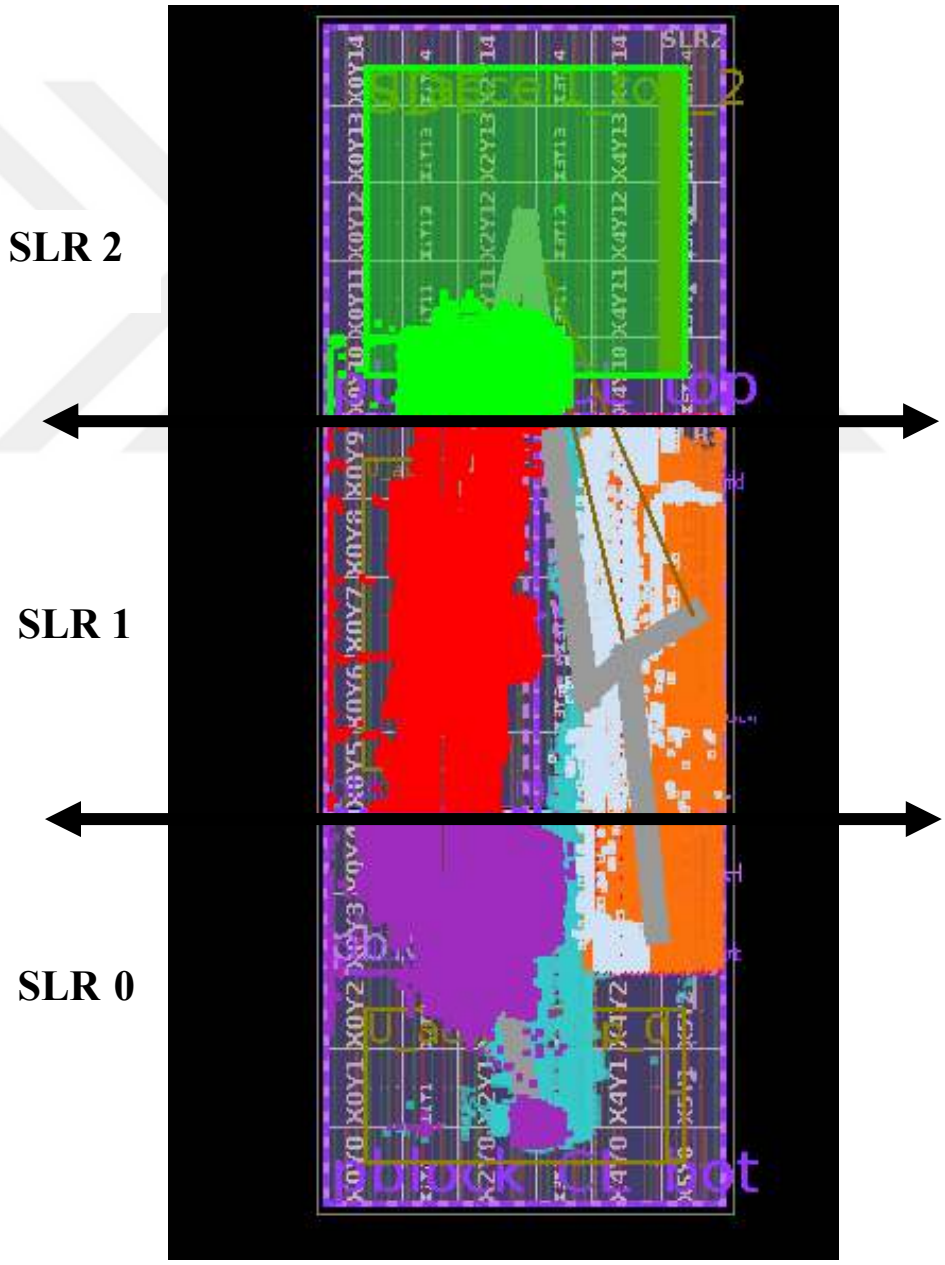

Figure 5.12: *Test suit Function FPGA implementation*

Table 5.9 provides the SLR connectivity data for the Function test suite. It shows the number of inter-SLR signal crossings between SLR0, SLR1, and SLR2. For instance, there are 3458 signals between SLR0 and SLR1, and 1691 between SLR1 and SLR2. As explained in Section 4.2.1, the simple I/O controller module is responsible for handling these inter-SLR connections, ensuring that signal routing remains efficient and within the constraints of the FPGA's SLR interconnect capabilities. The relatively lower number of crossings between SLR2 and the other regions suggests that partition 3 has limited interaction with the rest of the system, likely due to its functional role.

Table 5.9: *SLR connectivity of test suit Function*

FROM / TO	*SLR2*	*SLR1*	*SLR0*
SLR2	0	1683	0
SLR1	1691	0	3433
SLR0	0	3458	0

Table 5.10 summarizes the CLB resource utilization for the Function test suite. Partitions 1, 2, and 3 are placed in SLR0, SLR1, and SLR2, respectively. As with previous test cases, SLR1 shows the highest utilization at approximately 62.59%, largely due to the presence of the encrypted AWS Shell, which as mentioned earlier in Table 2.2 accounts for around 33% of resource usage in this region. SLR0 shows a moderate utilization of 45.47%, which includes both partition 1 logic and the CL_SDE support infrastructure. SLR2, containing only partition 3 logic, accounts for just 9.76% of total CLB resources. Despite the non-uniform resource distribution caused by static infrastructure overhead, the compiler-managed partitioning still maintains an effective allocation strategy for the user-defined logic across all SLRs

Table 5.10: *Resource usage of a test suit Function*

Site Type	*SLR0*	*SLR1*	*SLR2*	*SLR0 %*	*SLR1 %*	*SLR2 %*
CLB	22397	30831	4806	45.47	62.59	9.76

Figure 5.13 illustrates the common placement strategy consistently applied across all FPGA test implementations discussed in this section. The approach begins with defining three primary physical regions on the FPGA fabric, namely pblock_CL_top, pblock_CL_mid, and pblock_CL_bot, which correspond to the upper, middle, and lower SLR regions, respectively. Each pblock is explicitly resized and constrained to encompass specific FPGA resources such as CLOCKREGION tiles, SLICE arrays, DSP48E2 blocks, RAMB18/36 blocks, URAMs, and LAGUNA interconnect resources. This careful allocation ensures that the physical resource distribution within each pblock accurately reflects the resource requirements of the associated functional partitions and modules, enabling effective resource management in a multi-SLR environment.

The resizing commands specify exact coordinate ranges for each resource type in each SLR region, carefully determined based on the physical layout and resource distribution of the encrypted AWS Shell used in the design. This approach ensures that resource allocations align with the fixed placement of the Shell logic. For example, the top pblock covers a range of clock regions within the uppermost SLR, while the middle and bottom pblocks are expanded to include slices, DSPs, and memory blocks across their respective SLRs. Additionally, snapping mode is enabled on the middle and bottom pblocks to enforce precise alignment with device clock region boundaries, further supporting predictable placement and enhanced timing optimization.

After defining and resizing these pblocks, cells are assigned to the pblocks through hierarchical filtering based on cell names and instance paths. This hierarchical cell assignment approach allows precise control over which logic modules are placed within each partition, maintaining design modularity and isolation between partitions. Compiler-generated HLS modules, identifiable by naming conventions such as *accel1_top_3* or *accel1_top_2*, are allocated to the corresponding top and middle pblocks. Meanwhile, the bottom pblock is assigned not only partition cells like *accel1_top_1* and *accel1_top_1* but also critical infrastructure modules such as PCIM, PCIS, AXI controller IP blocks, and reset logic.

```
##### CREATE PBLOCKS for SLR-Based Partitioning #####
create_pblock pblock_CL_top
create_pblock pblock_CL_mid
create_pblock pblock_CL_bot

##### RESIZE PBLOCKS to Match SLR Regions #####

# Top SLR (e.g., SLR2)
resize_pblock [get_pblocks pblock_CL_top] -add CLOCKREGION_X0Y10:CLOCKREGION_X5Y14
set_property PARENT [get_pblocks pblock_CL] [get_pblocks pblock_CL_top]

# Middle SLR (e.g., SLR1)
resize_pblock [get_pblocks pblock_CL_mid] -add {SLICE_X8Y380:SLICE_X10Y599}
resize_pblock [get_pblocks pblock_CL_mid] -add {DSP48E2_X12Y129:DSP48E2_X13Y239}
resize_pblock [get_pblocks pblock_CL_mid] -add {LAGUNA_X12Y240:LAGUNA_X15Y479}
resize_pblock [get_pblocks pblock_CL_mid] -add {RAMB18_X7Y122:RAMB18_X7Y239}
resize_pblock [get_pblocks pblock_CL_mid] -add {RAMB36_X7Y60:RAMB36_X7Y119}
resize_pblock [get_pblocks pblock_CL_mid] -add {URAM288_X9Y208:URAM288_X9Y259}
resize_pblock [get_pblocks pblock_CL_mid] -add CLOCKREGION_X0Y5:CLOCKREGION_X5Y9
set_property SNAPPING_MODE ON [get_pblocks pblock_CL_mid]
set_property PARENT [get_pblocks pblock_CL] [get_pblocks pblock_CL_mid]

# Bottom SLR (e.g., SLR0)
resize_pblock [get_pblocks pblock_CL_bot] -add {SLICE_X8Y98:SLICE_X10Y279}
resize_pblock [get_pblocks pblock_CL_bot] -add {DSP48E2_X11Y129:DSP48E2_X13Y279}
resize_pblock [get_pblocks pblock_CL_bot] -add {LAGUNA_X11Y120:LAGUNA_X15Y239}
resize_pblock [get_pblocks pblock_CL_bot] -add {RAMB18_X7Y60:RAMB18_X7Y119}
resize_pblock [get_pblocks pblock_CL_bot] -add {RAMB36_X7Y30:RAMB36_X7Y59}
resize_pblock [get_pblocks pblock_CL_bot] -add {URAM288_X9Y80:URAM288_X9Y159}
resize_pblock [get_pblocks pblock_CL_bot] -add CLOCKREGION_X0Y0:CLOCKREGION_X5Y4
set_property SNAPPING_MODE ON [get_pblocks pblock_CL_bot]
set_property PARENT [get_pblocks pblock_CL] [get_pblocks pblock_CL_bot]

##### ADD CELLS TO PBLOCKS #####

# Top Partition Cells (e.g., Partition 3)
add_cells_to_pblock [get_pblocks pblock_CL_top] [get_cells -hierarchical
-filter {NAME =~ *accel1_top_3*}]

# Mid Partition Cells (e.g., Partition 2)
add_cells_to_pblock [get_pblocks pblock_CL_mid] [get_cells -quiet -hierarchical
-filter {NAME =~ WRAPPER_INST/CL/SDE/*ILA*}]
add_cells_to_pblock [get_pblocks pblock_CL_mid] [get_cells -hierarchical
-filter {NAME =~ *accel1_top_2*}]

# Bottom Infrastructure and Partition Cells (e.g., Partition 1 + Shell IPs)
add_cells_to_pblock [get_pblocks pblock_CL_bot] [get_cells -quiet -hierarchical
-filter {NAME =~ WRAPPER_INST/CL/AXIL_OCIL_REG_SLC_BOT_SLR/*}]
add_cells_to_pblock [get_pblocks pblock_CL_bot] [get_cells -quiet -hierarchical
-filter {NAME =~ WRAPPER_INST/CL/PCIS_REG_SLC_BOT_SLR/*}]
add_cells_to_pblock [get_pblocks pblock_CL_bot] [get_cells -quiet -hierarchical
-filter {NAME =~ WRAPPER_INST/CL/PCIM_REG_SLC_BOT_SLR/*}]
add_cells_to_pblock [get_pblocks pblock_CL_bot] [get_cells -quiet -hierarchical
-filter {NAME =~ WRAPPER_INST/CL/PIPE_RST_N_BOT_SLR/*}]
add_cells_to_pblock [get_pblocks pblock_CL_bot] [get_cells -hierarchical
-filter {NAME =~ *accel1_top_1*}]
add_cells_to_pblock [get_pblocks pblock_CL_bot] [get_cells -hierarchical
-filter {NAME =~ *accel0_top_1*}]
```

Figure 5.13: *cl_pnr user placement pragmas*

Tests with varying resource utilization, as detailed in Tables 5.9, 5.7, and 5.5, were successfully routed using the SLR-based partitioning methodology. The resource usage data presented in Tables 5.10, 5.8, and 5.6 confirm that SLR-crossing net boundaries were respected, demonstrating the effectiveness of the partitioning approach. The implemented test suites achieved operating frequencies of 250 MHz for GCD, 125 MHz for ADPCM, and 250 MHz for Function, validating the practical viability of the method. Furthermore, the compiler described in Section 3 was enhanced to fully support SLR-based partitioning by integrating the techniques presented in Section 4, enabling seamless partition generation and resource allocation across multiple SLRs.

6. CONCLUSIONS AND FUTURE WORK

This thesis introduces and validates a scalable SLR-based partitioning methodology for multi-die FPGA architectures, aiming to improve resource utilization, interconnect efficiency, and performance in large-scale hardware designs. The work centers around compiler-driven automation of physical partitioning based on, supported by enhancements in RTL generation, inter-partition routing, and placement logic. The primary contributions include the integration of this methodology into GLOBAL's High-Level Synthesis (HLS) compiler, and the demonstration of its practical viability through both Verilator-based simulations and real hardware implementations on the Xilinx Virtex UltraScale+ VU9P FPGA hosted in the AWS F1 cloud environment.

The proposed partitioning strategy leverages user-defined SLR-per-chip pragmas to define the physical layout of FPGA partitions across one, two, or three dimensions. These configurations allow the compiler to determine not only how many partitions are needed but also how they should be distributed across the FPGA's SLR tiles. The methodology is supported by a compiler infrastructure that automatically instantiates directional I/O controllers north, south, east, west, up, and downbetween partitions, enabling inter-partition communication and coordination with external systems.

To assess correctness, scalability, and performance, extensive Verilator-based functional simulations were conducted. These simulations were performed on a high-performance local machine (Ubuntu 20.04, Intel i7-13620H, 32 GB RAM), comparing traditional multi-machine partitioning with the proposed SLR-based single-machine approach. Results from both 3-partition and 8-partition configurations demonstrated significant simulation speed-ups using the SLR-based methodology. For instance, in a (1,3,1) configuration, the Function benchmark showed a 218× speed-up, while in the 8-partition (1,8,1) setup, the same benchmark achieved a 393× speed-up. This performance improvement is attributed to the elimination of inter-machine communication overhead, with all partitions communicating internally via SLRs within a single Verilator process.

The scalability of the approach was further validated with two-dimensional (2,4,1) and three-dimensional (2,2,2) partitioning tests. These configurations confirmed the compiler's ability to handle complex topologies. In the (2,4,1) case, the compiler successfully generated east/west and north/south I/O controllers to enable inter-partition communication along both x and y axes. Similarly, in the (2,2,2) test case, full 3-dimensional routing was demonstrated using up/down, east/west, and north/south

interconnects. Figure 5.5 and the accompanying Verilog code confirmed the correctness and clarity of signal wiring and naming conventions for modular and hierarchical communication between partitions.

Following simulation, FPGA implementation tests were performed using Vivado 2024.1. HLS-generated RTL designs for the GCD, ADPCM, AES, and function benchmarks were synthesized, placed, and routed for deployment on the AWS F1 platform. Placement strategies were optimized based on encrypted AWS Shell locations, which necessitated fixed regions for PCIM, PCIS, and SDE IP blocks. The SLR-based pblock allocation and I/O controller placement logic ensured that partitioned logic aligned correctly with AWS Shell interfaces. Real hardware deployments confirmed correct functionality and stable operation across all test suites, validating the end-to-end flow from HLS to FPGA.

Future work may focus on adopting a hierarchical implementation strategy, which could significantly improve scalability for complex FPGA systems by reducing routing congestion and enhancing timing closure. This would enable support for designs with hundreds of partitions, facilitating deployment on next-generation multi-die FPGAs. In parallel, improvements to the compiler's automatic resource balancing and partitioning algorithms could lead to more efficient partitioning during synthesis, optimizing both placement quality and runtime performance. Integrating advanced heuristics or machine learning based partitioning estimators may further refine boundary selection, dynamically adapting to the resource profile, inter-partition communication volume, and test-specific workload characteristics.

In summary, this thesis presents a comprehensive and scalable framework for SLR-based partitioning, validated through Verilator simulations and real-world FPGA deployment on AWS F1 infrastructure. The proposed methodology achieves significant simulation speed-ups and demonstrates correctness across various partitioning dimensions and test cases. With further refinements particularly in timing optimization and compiler intelligence this approach has the potential to scale across emerging FPGA-based platforms, including applications in cloud computing, AI acceleration, and scientific computing. As device complexity and performance demands continue to grow, the techniques developed in this work provide a robust foundation for modular, high-throughput, and FPGA optimized design.

REFERENCES

[1] Weerasinghe, J., Abel, F., Hagleitner, C., and Herkersdorf, A. (2015, August). Enabling FPGAs in hyperscale data centers. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)* (pp. 1078-1086). IEEE.

[2] Du, L., Liang, T., Sinha, S., Xie, Z., and Zhang, W. (2023, February). Fado: F loorplan-a ware d irective o ptimization for high-level synthesis designs on multi-die fpgas. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (pp. 15-25).

[3] Di, Z., Tao, R., Mai, J., Chen, L., and Lin, Y. (2023). LEAPS: Topological-layout-adaptable multi-die FPGA placement for super long line minimization. *IEEE Transactions on Circuits and Systems I: Regular Papers*, *71*(3), 1259-1272.

[4] Voss, N., Quintana, P., Mencer, O., Luk, W., and Gaydadjiev, G. (2019, April). Memory mapping for multi-die FPGAs. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (pp. 78-86). IEEE.

[5] Guo, L., Chi, Y., Wang, J., Lau, J., Qiao, W., Ustun, E., ... and Cong, J. (2021, February). AutoBridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (pp. 81-92).

[6] Mazraeli, M., Gao, Y., and Chow, P. (2023, September). Partitioning large-scale, multi-FPGA applications for the data center. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)* (pp. 253-258). IEEE.

[7] Shin, G., Kim, J., and Kim, J. Y. (2022). OpenMDS: An open-source shell generation framework for high-performance design on Xilinx multi-die FPGAs. *IEEE Computer Architecture Letters*, *21*(2), 101-104.

[8] Ravishankar, C., Gaitonde, D., and Bauer, T. (2018, August). Placement strategies for 2.5 D FPGA fabric architectures. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)* (pp. 16-164). IEEE.

[9] Liao, Y. C., and Mak, W. K. (2020). Pin assignment optimization for multi-2.5 D FPGA-based systems with time-multiplexed I/Os. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *40*(3), 494-506.

[10] Lavin, C., and Hung, E. (2023, October). RapidWright: Unleashing the Full Power of FPGA Technology with Domain-Specific Tooling. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)* (pp. 1-7). IEEE.

[11] Ebcioglu, K., and San, I. (2022). Highly Parallel Multi-FPGA System Compilation from Sequential C/C++ Code in the AWS Cloud. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, *15*(4), 1-42.

[12] Ebcioğlu, K., Bulut, B., Doğan, A., Küçük, G., and San, İ. (2025). 1,024-FPGA DES Supercomputer on the AWS Cloud. *Concurrency and Computation: Practice and Experience*, *37*(6-8), e70051.

[13] Canis, A., Choi, J., Fort, B., Lian, R., Huang, Q., Calagar, N., ... and Anderson, J. (2013, September). From software to accelerators with LegUp high-level synthesis. In *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)* (pp. 1-9). IEEE.

[14] Ahmed, M. K., Mandebi, J., Saha, S. K., and Bobda, C. (2022). Multi-tenant cloud FPGA: A survey on security. *arXiv preprint arXiv:2209.11158*.

[15] Cong, J., Lau, J., Liu, G., Neuendorffer, S., Pan, P., Vissers, K., and Zhang, Z. (2022). FPGA HLS today: successes, challenges, and opportunities. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, *15*(4), 1-42.

[16] K. Ebcioglu and E. Kultursay, **Method and system for converting a single-threaded software program into an application-specific supercomputer**, U.S. Patent 10,642,588 B2, May 5, 2020.

[17] Knickerbocker, J. U., Kong, L. W., Niese, S., Diebold, A., and Zschech, E. (2012). 3D interconnect technology. *Advanced interconnects for ULSI technology*, 435-490.

http-1: https://azure.microsoft.com (Accessed 29.12.2024)

http-2: https://aws.amazon.com/ec2/instance- types/f1/(Accessed Date 29.12.2024)

http-3: https://www.alibabacloud.com (Accessed 29.12.2024)

http-4: https://docs.amd.com/v/u/en-US/wp380_Stacked_Silicon_Interconnect_ Technology (Accessed 29.12.2024)

http-5: Vivado Congestion https://adaptivesupport.amd.com/s/article/66314/ (Accessed Date: 29.12.2024)

http-6: https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_1/ ug949-vivado-design-methodology.pdf (Accessed 29.12.2024)

http-7: https://slideplayer.com/slide/14536000 (Accessed 02.01.2025)

http-8: https://chiplet-marketplace.com/ whitepaper/ business-analysis-of-chiplet-based-systems-and-technology (Accessed 02.01.2025)

http-9: https://www.xilinx.com/support/documents/sw_manuals/xilinx2012_3/ug872_largefpga.pdf (Accessed 05.03.2025)

http-10: https://docs.amd.com/v/u/en-US/ug902-vivado-high-level-synthesis (Accessed 05.03.2025)

http-11: https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/ (Accessed 05.03.2025)

http-12: https://www.intel.com/content/www/us/en/software-kit/774965/intel-high-level-synthesis-compiler.html (Accessed 05.03.2025)

http-13: https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html (Accessed 29.03.2025)

http-14: https://www.zurich.ibm.com/cci/ cloudFPGA/ (Accessed 09.04.2025)

http-15: https://github.com/aws/aws-fpga/tree/f2/sdk/apps/virtual-ethernet (Accessed 09.04.2025)

http-16: http://dpdk.org (Accessed 09.04.2025)

http-17: https://github.com/ aws/aws-fpga/ (Accessed 09.04.2025)

CURRICULUM VITAE

ORCID ID: 0009-0004-2123-6597

Name Surname : Batuhan BULUT

Foreign Language : English

Place and Year of Birth :

Email :

Education and Professional Background:

- 2021, Eskişehir Technical University, Faculty of Engineering, Department of Electrical and Electronics Engineering
- 2021 October - Software Engineer, ERENDIZ SUPER BILGISAYAR LTD. ŞTI.

Publications and/or Scientific/Artistic Activities:

- K. Ebcioğlu, B. Bulut, A. Doğan, G. Küçük, and I. San, "1,024-FPGA DES supercomputer on the AWS cloud," *Concurrency Comput.: Pract. Exper.*, vol. 37, no. 6–8, Mar. 2025, doi: 10.1002/cpe.70051.
- B. Bulut, S. N. Bicakci, and I. San, "HW/SW co-design of TFHE homomorphic OR gate via NTT-based polynomial multiplication on a programmable SoC," in *Proc. 2022 Innovations in Intelligent Systems and Applications Conf. (ASYU)*, Sep. 2022, pp. 1–6, doi: 10.1109/ASYU56188.2022.9925412.