LACE NETWORK FIRMWARE: A POLARFIRE FPGA NETWORK FOR DATA ROUTING AND COMMAND INTERFACING IN SPACE APPLICATIONS

by

Kade C Howes

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Electrical Engineering

Approved:

Charles Swenson, Ph.D. Major Professor	Jonathan Phillips, Ph.D. Committee Member
Don Cripps, Ph.D. Committee Member	David F. Feldon, Ph.D. Vice Provost of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2025

ABSTRACT

LACE Network Firmware: A PolarFire FPGA Network for Data Routing and Command Interfacing in Space Applications

by

Kade C Howes, Master of Science

Utah State University, 2025

Major Professor: Charles Swenson, Ph.D.
Department: Electrical and Computer Engineering

This thesis presents the design and implementation of the LACE network firmware, a PolarFire Field Programmable Gate Array (FPGA) network aimed at accelerating the development of machine learning algorithms and handling data routing and command interfacing for spacecraft applications. The architecture consists of a controller FPGA and one or multiple peripheral FPGAs. The controller FPGA is uses a high-speed Ethernet port for incoming data and a Universal Asynchronous Receiver Transmitter (UART) interface for command and telemetry interfacing with a spacecraft computer. The controller FPGA is responsible routing the data to and from the appropriate peripheral FPGA(s) over PolarFire's built-in transceiver lanes. The development process leverages Simulink to create a library of models that for data routing, Input/Output handling, and Consultative Committee for Space Data Systems (CCSDS) and Ethernet packet handling. This architecture aims to streamline the integration of machine learning algorithms by providing the peripheral FPGAs' applications with timely and organized data, allowing more complex algorithms on peripheral FPGAs and enhancing the efficiency and accessibility of the system.

(116 pages)

PUBLIC ABSTRACT

LACE Network Firmware: A PolarFire FPGA Network for Data Routing and Command Interfacing in Space Applications

Kade C Howes

Modern small spacecraft rely on powerful yet efficient onboard compute devices to process data from sensors in real-time due to tight power and volume constraints. This work explores a new compute device system built from PolarFire Field-Programmable Gate Arrays (FPGAs), which are power-efficient, reprogrammable chips well-suited for space applications. The system connects via a central controller FPGA with one or multiple companion processing FPGAs, allowing sensor data to be quickly received and shared across the network. Standardized data formats and interfaces increase compatibility with spacecraft computers. By simplifying data handling and using high-speed communication links, this architecture makes it easier to integrate advanced algorithms, such as machine learning, directly onboard spacecraft. This approach supports faster, more efficient decision-making in space and reduces reliance on ground-based processing, which is especially valuable for missions with limited communication bandwidth or time-sensitive operations.

To Robert Drollinger.

ACKNOWLEDGMENTS

I would first like to thank Dr. Charles Swenson for guiding this process and opening doors I didn't expect to find.

I also would like to thank five good friends for their support through this project and related endeavors: Benjamin Lewis, Justin Wellington, Rowan Antonuccio, Shawn Jones, and Todd Wilson.

Kade C Howes

CONTENTS

Page

LIST OF FIGURES

ACRONYMS

ADC	Analog-to-Digital Converter
AI	Artificial Intelligence
APB	Advanced Peripheral Bus
APID	Application Process Identifier
CAM	Content-Addressable Memory
CCC	Clock Conditioning Circuit
CCSDS	Consultative Committee for Space Data Systems
CDC	Clock Domain Crossing
CRC	Cyclic Redundancy Check
FCS	Frame Check Sequence
FIFO	First-In, First-Out
FPGA	Field-Programmable Gate Array
Gbps	Gigabits Per Second
GPS	Global Positioning System
GPIO	General-Purpose Input/Output
HDL	Hardware Description Language
I/O	Input/Output
IP	Intellectual Property
JTAG	Joint Test Action Group
LACE-C3A	Low-power Array for CubeSat Edge Computing Architecture, Algorithms and Applications
LSRAM	Large Static Random-Access Memory
LVDS	Low-Voltage Differential Signaling
PPS	Pulse-Per-Second
RAM	Random-Access Memory
RID	Routing Identifier
RX	Receive

RTC	Real-Time Clock
SERDES	Serializer/Deserializer
SGMII	Serial Gigabit Media Independent Interface
SMA	SubMiniature version A
sNVM	Secure Non-Volatile Memory
TX	Transmit
UART	Universal Asynchronous Receiver/Transmitter
μSRAM	Micro Static Random-Access Memory
USU	Utah State University
VHDL	VHSIC Hardware Description Language

CHAPTER 1

Introduction

Edge computing, where data is processed near its source instead of in a centralized location, is becoming useful for satellite missions as it reduces communication cost and latency of real-time measurements. However, this requires hardware that is both low-power and capable of real-time processing. Field-Programmable Gate Arrays (FPGAs) are well-suited for this due to their ability to handle multiple computations in parallel efficiently when programmed properly [5]. This thesis focuses on creating an FPGA network to streamline algorithm development by managing data routing, time stamping, and the spacecraft command interfacing.

There are several approaches to using FPGAs in satellite missions for processing data from sensors. One method involves using multiple unique interfaces, each designed for a specific sensor type. While this approach can achieve low power consumption and efficient resource utilization, it is inflexible and increases development time and complexity due to the need for multiple custom interfaces [6, 7].

Another method is using a standard protocol like Ethernet for all communication, which simplifies development and integration by packetizing data uniformly [6]. However, this approach introduces resource and power overhead, as each FPGA endpoint requires an Ethernet core or additional hardware to handle Ethernet communication.

The proposed architecture combines benefits of both methods. It uses low-overhead, point-to-point communication between FPGAs for efficiency while incorporating an Ethernet port for incoming sensor data. The system includes a controller FPGA and multiple peripheral FPGAs. All FPGAs are planned to be flash-based PolarFire FPGAs, as they are naturally more radiation tolerant for space applications and they have built-in high-speed transceiver lanes for efficient FPGA-to-FPGA communication.

The controller FPGA, equipped with a high-speed Ethernet port and a Universal Asynchronous Receiver/Transmitter (UART) port, acts as the central hub, parsing Ethernet packets from the sensors and Consultative Committee for Space Data Systems (CCSDS) Space Packets from the spacecraft. The controller then routes this data to corresponding peripheral FPGAs via PolarFire FPGA's built-in transceiver lanes. The controller's packet routing will support both sensor-to-peripheral and peripheral-to-peripheral communication, enabling the integration of processes that exceed the capacity of a single peripheral FPGA, increasing system flexibility and scalability.

To simplify communication, the system uses a custom packet structure called LACE packets. These are designed similar to CCSDS Space Packets, but modified to fit the 32-bit architecture of this LACE network firmware system. See Section 3.1 for more details.

The development process utilized Hardware Description Language (HDL) and Simulink to build a library of models for data routing, Input/Output (I/O) handling, CCSDS Space Packet handling, Ethernet frame handling, and custom LACE packet handling. Simulink is being evaluated for its potential to reduce development time and improve firmware reconfigurability for specific missions [7].

This firmware architecture is designed to be utilized onboard Utah State University's (USU) FPGA hardware stack named Low-power Array for CubeSat Edge Computing Architecture, Algorithms and Applications (LACE-C3A or LACE). This is the reason that this firmware system will be called the LACE network firmware and is referred to throughout this thesis.

1.1 LACE-C3A Overview

The Low-power Array for CubeSat Edge-Computing Architecture, Algorithms, and Applications (LACE-C3A) is a reconfigurable, FPGA-based computing framework developed to enable distributed data processing on small spacecraft. LACE-C3A extends the principles of edge computing, processing data near its point of origin rather than relying solely on ground or centralized systems, into the context of small satellites. The architecture leverages the low-power and radiation-tolerant characteristics of PolarFire FPGAs

to provide scalable on-board computing that can adapt to mission requirements, support multiple sensor interfaces, and manage high-rate data routing in real time.

Figure 1.1 presents a conceptual spacecraft data flow diagram illustrating how LACE-C3A integrates within a representative small-satellite avionics architecture. In this configuration, sensor data from one or more spacecraft instruments are received by an Ethernet frame router, which serves as a centralized data switch between payloads, the flight computer, and the LACE-C3A computer. The router classifies and forwards incoming Ethernet frames based on destination addressing, distributing them either to the spacecraft computer for storage and telemetry downlink, or to the LACE-C3A subsystem for on-board processing. This system represents a flexible and easily integrated spacecraft data system.

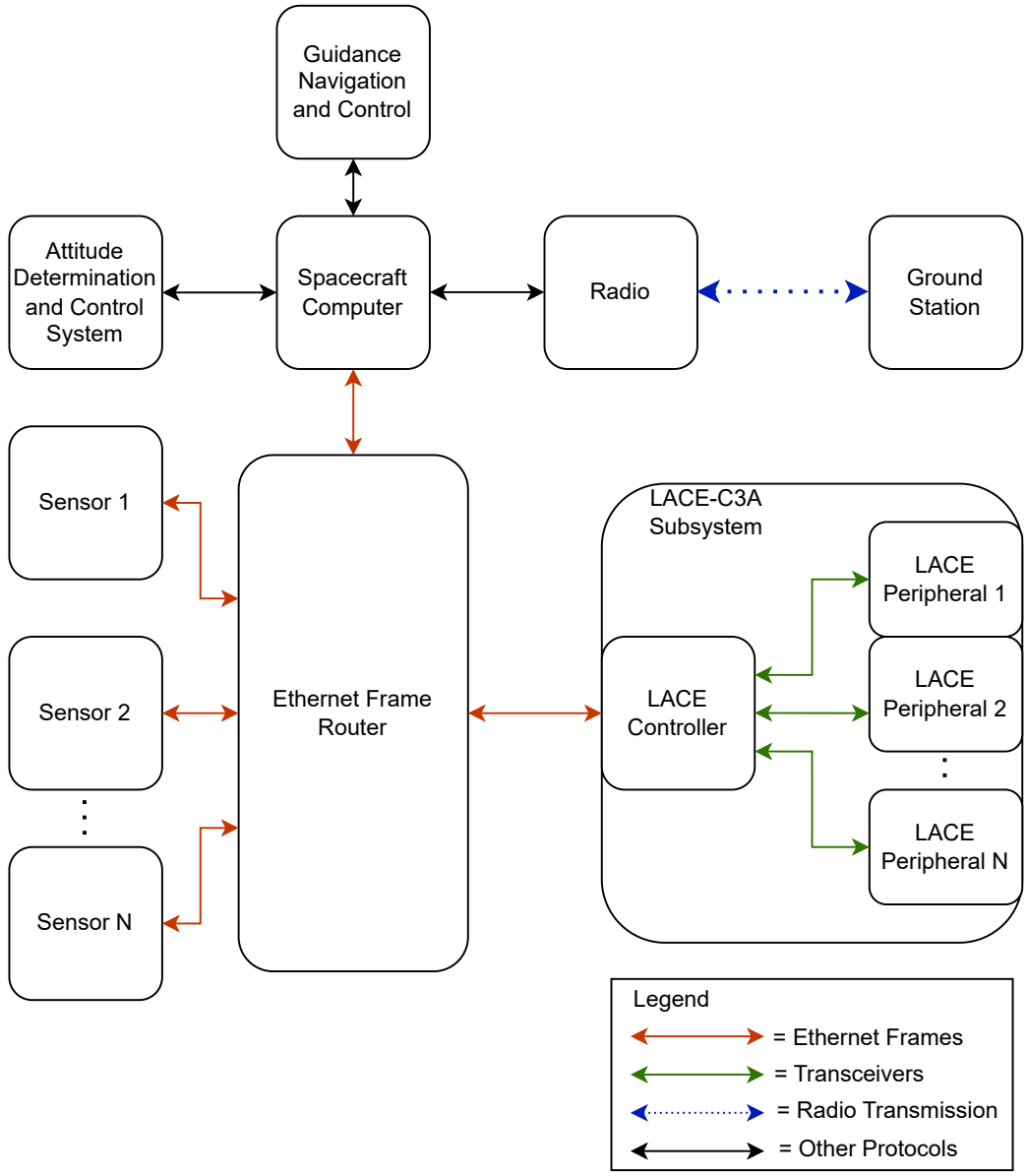

Fig. 1.1: LACE-C3A and Representative Spacecraft Data Flow

Within this network, LACE-C3A operates as a dedicated edge-computing node, performing processing, filtering, compression, feature extraction, or any other processing step on raw sensor data before it is relayed back to the spacecraft computer. This approach re-

duces downlink bandwidth requirements and enables low-latency, in-situ decision-making, such as event detection or adaptive sensor tasking. By situating computation within the data path between sensors and the spacecraft controller, LACE-C3A effectively acts as a distributed processing layer that offloads computationally intensive tasks from the main flight computer.

The conceptual design also demonstrates the modularity of LACE-C3A. Multiple peripheral FPGAs can be networked through high-speed serial transceivers to scale computational capacity as needed for a given mission. Each node can execute specific algorithms or process different sensor streams in parallel under the coordination of a controller FPGA. This modular, distributed configuration aligns with current trends in small-satellite avionics toward flexible and reconfigurable computing systems that can evolve with mission objectives.

1.2 Research Objectives

The proposed research has the three primary objectives:

1. Develop firmware for peripheral FPGAs that receives sensor data from the controller's Ethernet port, using fewer FPGA resources than direct Ethernet implementation on the peripheral FPGA

2. Timestamp all sensor and telemetry data with 1 millisecond precision

3. Establish a command and telemetry interface to an external spacecraft computer via UART

CHAPTER 2

Overview

2.1 System Overview

The LACE network firmware implements a distributed architecture composed of a controller FPGA and multiple peripheral FPGAs, all based on Microchip's PolarFire MPF300T devices. However, the architecture is designed to support easy adaptation to other PolarFire FPGAs with minimal modification, allowing flexibility in resource utilization and mission requirements related to compute, power, and size. The controller FPGA functions as the central hub for external communication and internal data routing, while the peripheral FPGAs act as specialized processing nodes within the network.

As shown in the LACE network system overview in Figure 2.1, the controller interfaces externally through two primary connections: a Gigabit Ethernet port and a UART interface. These interfaces are primarily used for sensor data ingress and command or telemetry exchange with a spacecraft computer, respectively, but the system is designed to support any form of input or output data through either port. Internally, the controller employs a Packet Router that processes incoming data and determines the appropriate routing path based on a Routing Identifier (RID), assigned from information contained within Ethernet framer header information (Gigabit Ethernet port) or CCSDS Space Packet Header information (UART port). Data can be directed to one or more peripheral FPGAs, returned through Ethernet or UART, exchanged between peripherals, or forwarded to the Command and Control Handler for configuration.

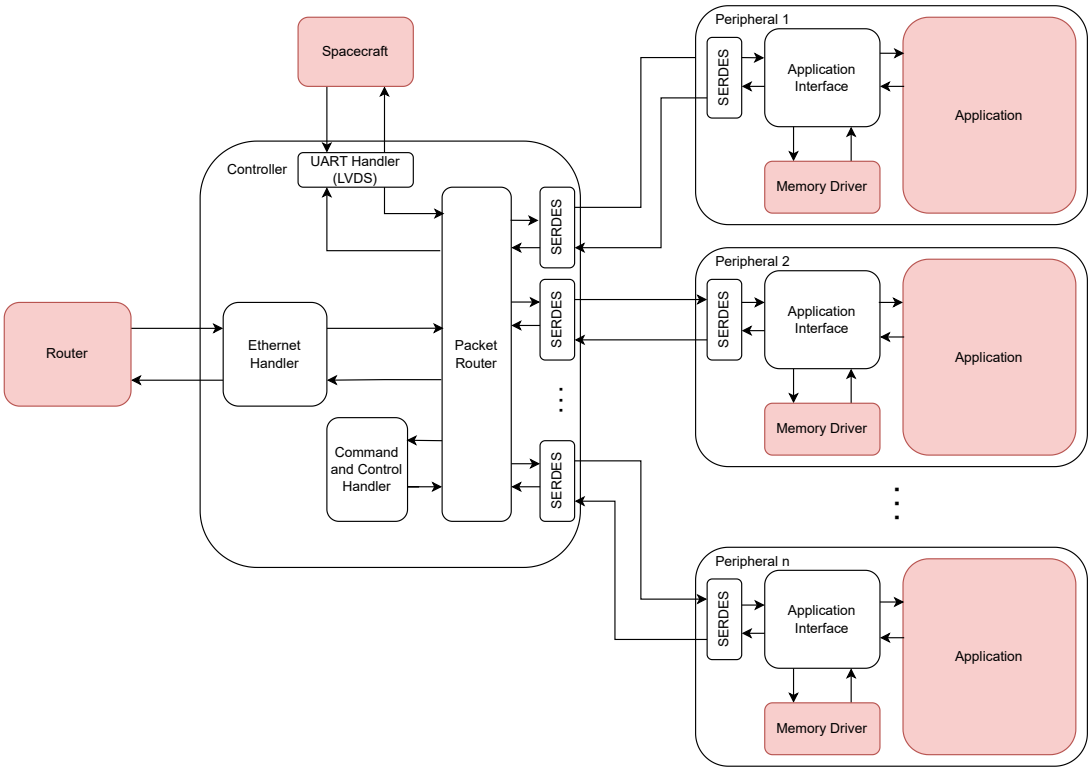

Fig. 2.1: LACE Network Overview

Communication between the controller and peripheral FPGAs occurs over the PolarFire's integrated serializer/deserializer (SERDES) transceiver lanes using LiteFast Intellectual Property (IP) cores, which enable reliable, low-latency data transfer at up to 12.7 Gigabits per second (Gbps). Each peripheral FPGA receives LACE Packets (See Chapter 3), processes its assigned data, and sends results to the controller to output or forward them to other peripheral endpoints as required.

This modular, distributed design allows the system to be easily scaled or reconfigured for different mission requirements. The separation between data routing on the controller and computation on the peripherals reduces complexity, reduces FPGA utilization per peripheral FPGA, and enables rapid firmware adaptation for future spacecraft payloads.

2.2 Methodology

The development of the LACE network firmware combined development in simulation using MATLAB Simulink and hardware validation using Libero SoC by Microchip, an FPGA all-in-one programming and debugging tool. This helped ensure that each component of the FPGA network system was verified at both the algorithmic and hardware levels.

2.2.1 Project Setup and Development Environment

Most algorithm and model development for this research was performed in MATLAB Simulink, using HDL Coder for algorithm-level design and HDL code generation. All models, except the top-level models intended for HDL generation and hardware implementation, were developed as referenced subsystems within Simulink. This approach promotes model reuse across multiple contexts and simplifies simulation. By using referenced subsystems, signals from any lower-level subsystem can be observed collectively within a single Logic Analyzer instance in Simulink, streamlining verification and analysis.

A MATLAB project was created to help track hierarchical dependencies between Simulink models. This project was named `LACE_network`. Additionally, this MATLAB project runs a MATLAB script on startup (`LACE_network_project_setup.m`) , allowing for all environment variables to be populated automatically by calling these setup scripts when the MATLAB project is run. The structure for this MATLAB project can be seen in Figure 2.2. The setup scripts mentioned were stored under `scripts/`. Referenced subsystem models were stored under `model_library/`, with their corresponding testbenches stored under `model_testing/`. Top-level models to be implemented on FPGA hardware were stored under `top_modules/`. The `LACE_network.prj` file contains the MATLAB project metadata and running this opens the project for development within MATLAB. This model of MATLAB project and environment was inspired by the development of the Space Weather Probes 2 firmware [7].

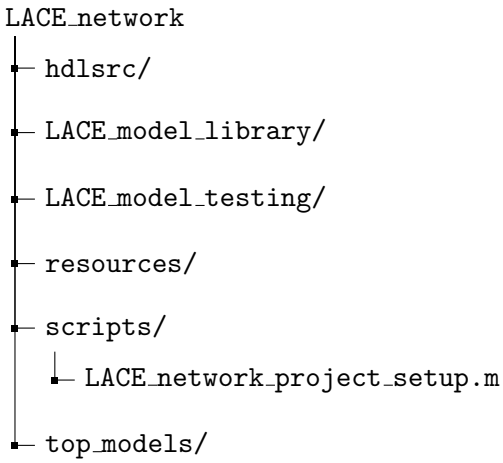

Fig. 2.2: LACE Network MATLAB Project Structure

With these environment variables setup in MATLAB, Simulink has access to all variables, allowing simple configuration of models by editing of MATLAB scripts that run on project startup. This helps to keep the development environment consistent and the HDL code generation to be repeatable within this environment.

Once each referenced subsystem, or multiple together, were validated in simulation in Simulink, HDL Coder, an add-on to Simulink, was used to generate VHSIC Hardware Description Language (VHDL) code directly from the Simulink model. This VHDL is built hierarchically, allowing the entire hierarchy to be used in the hardware design suite tool, Libero SoC. The resulting VHDL is stored in the `hdlsrc/` folder as seen in Figure 2.2.

2.2.2 Hardware Implementation Process via Libero SoC

After HDL code generation from Simulink, hardware implementation was carried out using Libero SoC by Microchip. Libero SoC served as the central design software for synthesizing, placing and routing, programming, and debugging the PolarFire MPF300T devices used in this research.

Each HDL model, generated automatically by HDL coder, was imported into Libero SoC as a separate component using the SmartDesign tool. These modules were then combined with vendor-supplied IP blocks to enable high-speed transceiver communication, Eth-

ernet interfaces, and clock and reset management. The most critical IP cores included the LiteFast transceiver cores [4], responsible for encoding, serializing, deserializing, and decoding data streams between FPGAs, and the Gigabit Ethernet Media Access Control (MAC) IP core named CoreTSE [8], which provided low-level packet handling and verification for the Ethernet interface.

The hardware design flow followed these main steps:

1. SmartDesign Creation: SmartDesign modules were created in Libero SoC to simplify the process of integrating all imported HDL modules and vendor-supplied IP blocks.

2. Synthesis: SmartDesign modules were synthesized using Synplify Pro within Libero SoC to generate optimized gate-level netlists.

3. Constraint Definition: Timing and I/O constraint files were created using a combination of reference files for IP blocks and manual assignment.

    - I/O constraints were applied for differential SubMiniature version A (SMA) connectors for the transceivers between FPGAs, the Ethernet PHY interface, and the UART serial lines.
    - Timing constraints were added to ensure placement and routing within the FPGA meets the assign clock's timing requirements.

4. Placement and Routing: After synthesis, placement and routing were performed with timing verification. This ensured that all data paths met setup, hold, and propagation delay requirements.

5. Bitstream Generation and Programming: Once verified, programming bitstreams were generated for both the controller and peripheral boards.

Hardware debugging was done using the Identify Debugger tool in Libero SoC, which allows internal signal capture from the FPGA in real-time. Additionally, live testing was done using a combination of software including Wireshark, Cat Karat, Arduino IDE, and

custom spacecraft emulation Python code. This combination of software simulation, synthesis verification, hardware probing, and live testing ensured that the firmware met both timing and functional requirements on FPGA hardware.

2.2.3 Version Control

Version control for the LACE network firmware was managed using Git to ensure consistent and traceable development across multiple tools and environments. The Git repository was structured to support both MATLAB Simulink and VHDL-based workflows while maintaining compatibility with Libero SoC project files.

For MATLAB and Simulink models, the MATLAB Project integration with Git was used to enable graphical change tracking of Simulink diagrams and to ensure version consistency across referenced subsystems. Commit metadata recorded changes in both model structure and HDL parameters, allowing tracking of changes to design configurations or project settings.

2.3 Literature Review

Modern small satellites are evolving from simple data-gathering platforms into sophisticated sensor networks capable of on-board interpretation, decision-making, and autonomous response. This transformation has been driven by advances in miniaturized processing hardware, reconfigurable logic, and energy-efficient computing hardware. The literature surrounding these developments crosses multiple domains, from high-speed digital interconnects and distributed FPGA firmware to space-qualified onboard computers and reconfigurable payload processors. Understanding this body of work is important to place the Low-Power Array for CubeSat Edge-Computing Architecture, Algorithms, and Applications (LACE-C3A) within the broader context of spacecraft computing.

This review focuses on three primary areas: (1) state-of-the-art onboard computing systems for small satellites, (2) recent trends toward distributed, adaptive, and edge-oriented architectures that perform real-time analysis at the sensor system or point of data collection,

and (3) serial interconnect technologies and firmware frameworks that enable communication among multiple FPGAs.

Together, these topics establish the technical foundation upon which LACE-C3A advances reconfigurable and distributed edge computing for next-generation small spacecraft missions.

2.3.1 Onboard Edge Computers for Small Satellites

The *NASA Small Spacecraft Systems Virtual Institute (S3VI)* publishes the *State-of-the-Art Small Spacecraft Technology Report* annually to summarize worldwide developments in small satellite subsystems. The 2024 edition (NASA Ames Research Center, February 2025) dedicates an entire chapter to Avionics and Processing, charting a progression from single-board radiation-hardened controllers to reconfigurable and distributed computing systems suitable for on-board data processing. Commercial off-the-shelf single-board computers have become popular in CubeSat and microsatellite classes because of their low cost and high performance. Manufacturers such as GomSpace, ISISpace, and Blue Canyon Technologies offer modular boards featuring ARM or Intel Atom processors [16].

The addition of FPGAs for computing has emerged as an important method for onboard processing for small satellites. NASA's SpaceCube family demonstrates this paradigm by integrating general-purpose processors and FPGAs on a common backplane that supports partial reconfiguration of the FPGAs in orbit. The third generation, SpaceCube v3.0, was developed as a next-generation high-performance platform for scientific and Earth-observing missions, demonstrating up to 50 times the acceleration for image-processing tasks compared with CPU-only systems [12]. George and Wilson's survey in the *Proceedings of the IEEE* highlights how such hybrid processor and FPGA architectures provide resilience and performance scalability for small satellite missions [13]. Other missions such as the HYPSO-1 hyperspectral CubeSat employed a hybrid CPU–FPGA processor with in-flight reprogramming and an autonomous image-selection pipeline [14]. European missions such as OPS-SAT have also validated this approach. The OPS-SAT spacecraft hosts a reconfigurable onboard computer capable of executing experimental AI and image-classification algorithms directly

in orbit [15]. These developments demonstrate the growing use of FPGA-based reconfigurable computing for spacecraft applications.

2.3.2 Emerging Edge Computing Processes and Architectures

The latest *State-of-the-Art Small Spacecraft Technology Report* identifies Artificial Intelligence (AI) and machine learning acceleration as a fast-growing domain in small-satellite computing [16]. Ongoing experiments such as NASA's *SpaceCube Edge-Node Intelligent Collaboration (SCENIC)* extend these ideas to distributed AI inference and compute task allocation across multiple spacecraft nodes [17]. Both NASA's SCENIC program and European efforts such as the ϕ-Sat series aim to demonstrate neural network inference for image classification and anomaly detection using onboard FPGAs and processors. Recent academic studies also point to the feasibility of such approaches. Moreira et al. developed an FPGA-based architecture for thermal anomaly detection in orbit [18], and Carr et al. demonstrated an FPGA pipeline for real-time stereo-image processing and scientific data reduction [19]. These works emphasize the importance of high-bandwidth interconnects and low-latency firmware layers, features central to the LACE-C3A architecture.

Within this context, LACE-C3A represents another step toward distributed, reconfigurable edge computing for small spacecraft. The system composed of a controller FPGA and multiple peripheral FPGAs that are interconnected through high-speed serializer/deserializer (SERDES) links shares the modularity of NASA's SpaceCube with a goal of reducing size, weight, and power. In this sense, LACE-C3A operationalizes many of the trends identified by NASA's *State-of-the-Art Small Spacecraft Technology Report* and the broader literature including scalability, adaptability, and autonomy at the edge while adapting them to the size, weight, and power constraints of CubeSat-class missions.

2.3.3 Interconnect Technologies and Frameworks for Configurability

The development of packet-routed architecture across multiple FPGAs builds upon more than two decades of research in spacecraft data networks, on-board processing, and reconfigurable computing. The LACE-C3A system combines a controller FPGA that man-

ages data ingress and egress, routing, and distribution to multiple peripheral FPGAs that perform localized processing. Comparable approaches have emerged in high-performance, on-board data handling systems, multi-FPGA routing fabrics, and reconfigurable payload processors for space. In this section we review key literature in this area.

High-Speed Inter FPGA Links

High-bandwidth serial transceivers integrated into modern FPGAs enable reliable communication between multiple devices in a distributed architecture. The Microchip PolarFire family supports transceiver lanes operating between 500 megabits per second (Mbps) and 12.7 Gbps, forming a physical layer suitable for inter-FPGA data exchange [20]. Microchip's LiteFast IP core provides a lightweight data-link layer with token-based flow control, idle frame maintenance, and cyclic-redundancy checks (CRC-32) [4]. This approach is analogous to other low-overhead streaming link layers such as Xilinx's Aurora 64b/66b protocol but optimized for PolarFire devices.

The European Space Agency's SpaceFibre protocol extends these ideas into a standardized, fault-tolerant multi-gigabit serial network for spacecraft. Parkes et al. (2015) introduced SpaceFibre as a high-rate replacement for SpaceWire, incorporating multi-lane bonding, virtual channels for quality-of-service control, and fault-detection and isolation mechanisms [21]. Follow-up work demonstrated an FPGA-based routing switch supporting deterministic latency and error recovery [22], and later implementations confirmed compatibility on radiation-tolerant RTG4 devices [23]. Recent publications describe efficient, multi-lane SpaceFibre cores that achieve automatic lane alignment and retransmission control [24]. Collectively, these results validate the feasibility of multi-gigabit, fault-tolerant inter-FPGA fabrics for flight applications. LACE-C3A's use of PolarFire SERDES lanes with LiteFast IP follows this proven model but tailors the implementation to a small-satellite power and resource budget.

Packetization and Routing

Packet-based routing is the backbone for scalable on-board satellite communications.

The Consultative Committee for Space Data Systems (CCSDS) Space Packet Protocol (133.0-B-2) defines standardized primary and secondary packet headers used throughout the space community for telemetry, telecommand, and payload data [2]. By assigning a Routing Identifier (RID) derived from these header fields or from Ethernet framing, a controller can dynamically forward packets to the appropriate peripheral node or to an external interface for data egress.

Previous research on network-on-chip and multi-FPGA routing architectures offers design insights directly applicable to the LACE-C3A Packet Router. Khalid et al. (2014) presented a scalable multi-FPGA routing architecture emphasizing throughput and minimizing critical path delays [25]. Similarly, Nepomuceno et al. (2021) demonstrated task parallelism across multiple FPGAs [26]. In a spacecraft context, NASA Goddard Space Flight Center's study of SpaceWire network behavior documented the need for router port stall mitigation and flow control to prevent receive buffer overflow [27]. Together, these works motivate LACE-C3A's RID-based packet router, which separates routing from computation to improve determinism and maintainability while maintaining compatibility with standard packet structures.

Data Ingress and Egress Interfaces

The LACE-C3A controller FPGA handles external communication through a Gigabit Ethernet port and a UART port. Ethernet interfaces have become popular in modern spacecraft payloads because of their versatility and support for existing ground tools. Microchip provides *CoreTSE* and *Core10GMAC* IP cores that implement 1 GbE and 10 GbE physical layers, respectively [20]. The UART channel remains valuable for command, telemetry, or debug communication.

Combining Ethernet and UART within the controller simplifies integration with existing spacecraft command and data handling systems while allowing flexible, high-bandwidth data ingestion from sensors. Comparable mixed-interface strategies appear in both the NASA SpaceCube platform [12] and in the HYPSO-1 hyperspectral mission, where payload FPGAs exchange data with the on-board computer and sensor sources over standard serial

and network links [14].

In-Flight Reprogramming and Scalable Multi-FPGA Systems

For long-duration missions and reconfigurable payloads, the ability to update FPGA fabric in flight is valuable. The *PolarFire FPGA and PolarFire SoC FPGA Programming User Guide* details Auto-Update and In-Application Programming (IAP) modes, which allow the system controller to load a new configuration image from external flash memory [28]. This feature supports future selective reprogramming of individual peripheral FPGAs without interrupting the controller's operation.

NASA's SpaceCube v3.0 system embodies a similar philosophy by enabling field reconfiguration and partial updates to adapt processing on orbit [12]. Broader surveys of hybrid computing on small satellites highlight the trend toward combining general-purpose processors with reconfigurable logic to achieve flexibility and efficiency [13]. The OPS-SAT in-orbit platform further demonstrates live experimentation and dynamic payload reprogramming [15], while HYPSO-1 provides a practical demonstration of a robust boot pipeline for FPGA-based image processing [14].

Recent investigations of FPGA edge-computing architectures, such as Moreira et al. (2025) and Carr et al. (2025), show the feasibility of deploying advanced data reduction and machine learning algorithms directly within reconfigurable logic aboard spacecraft [18] [19]. These efforts align closely with the objectives of LACE-C3A, which seeks to bring adaptive computing to payloads with strict power and reliability constraints.

2.3.4 Literature Review Summary

The literature establishes a strong foundation for LACE-C3A's distributed firmware design. Inter-FPGA serial communication using methods such as LiteFast or SpaceFibre demonstrate multi-gigabit, fault-tolerant data exchange with low latency. Standardized packetization via CCSDS Space Packet Protocol ensures interoperability and straightforward routing. Hybrid ingress and egress through Ethernet and UART offers integration

flexibility, and the availability of FPGA fabric reprogramming supports future reconfigurability in flight. Collectively, these studies validate LACE-C3A's architectural decisions and highlight its contribution: an extensible, resource-efficient, multi-FPGA computing environment designed for next-generation CubeSat and small satellite missions.

2.4 Thesis Outline

An overview of the entire LACE network firmware and its development is given in Chapter 2. Chapter 3 outlines the packet structures used to communicate with external sources as well as between FPGAs within the LACE network. Chapter 3 also outlines the development of the Packet Router, a core piece of the LACE network firmware responsible for routing packets around the FPGA network appropriately.

Chapter 4 discusses the communication method between PolarFire FPGAs within the network using high-speed transceivers. Chapter 5 discusses the development and testing of Ethernet communication with an external Ethernet-connected source. Chapter 6 discusses the development and testing of UART communication vs CCSDS Space Packets with an external UART-connected source. Chapter 7 discusses how the system can be configured via command and control packets from an external source. Chapter 8 discusses the development of a simplified interface to be put onto all peripheral FPGAs to connect to the controller FPGA to be on the LACE network and to simplify development of the applications on the peripheral FPGAs. Chapter 9 discusses the results of the research process. Chapter 10 contains a summary and a discussion on further work related to this research project.

CHAPTER 3

Data Routing

A core part of this networking firmware is correctly and efficiently routing data around the sytem in a scalable fashion. This chapter discusses the types of packets and data that are used, the packet router for routing data between sources, and other firmware models used to facilitate the custom communication between the packet router and the sources.

3.1 Packet Structures

To facilitate moving data around the system efficiently and consistently, a new packet structure was created. This new packet structure is called a LACE Packet. The new packet structure was modeled after the CCSDS Space Packet Protocol structure. The CCSDS Space Packet overall structure can be seen in Figure 3.1. Note that these Space Packets are based on octets (bytes) and consists of a primary header that is 6 octets long, followed by a variable length user data field of up to 65,536 octets. These LACE Packets are used purely for passing data between FPGAs within the network. User applications on the peripheral FPGAs will have no notion of a LACE packet with the data they receive. User applications will get only receive the data inside the incoming packet, whether CCSDS or Ethernet, not the header information.

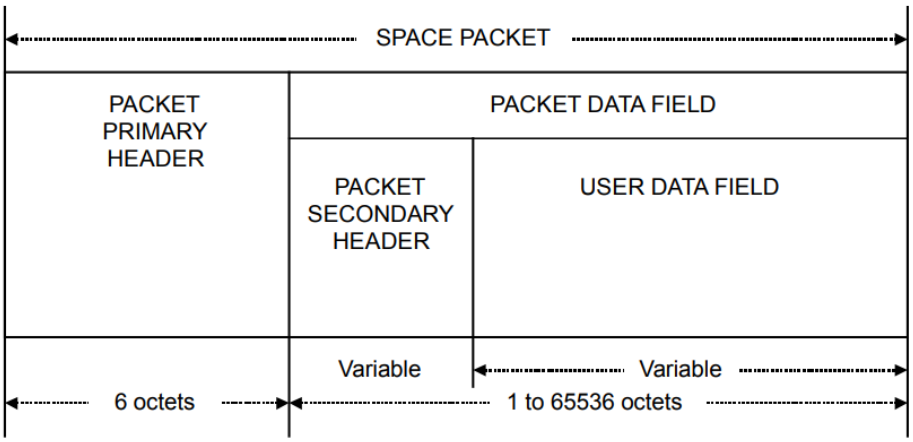

Fig. 3.1: CCSDS Space Packet Structure [2]

Additionally, CCSDS Space Packets's primary header field is structured as shown in Figure 3.2.

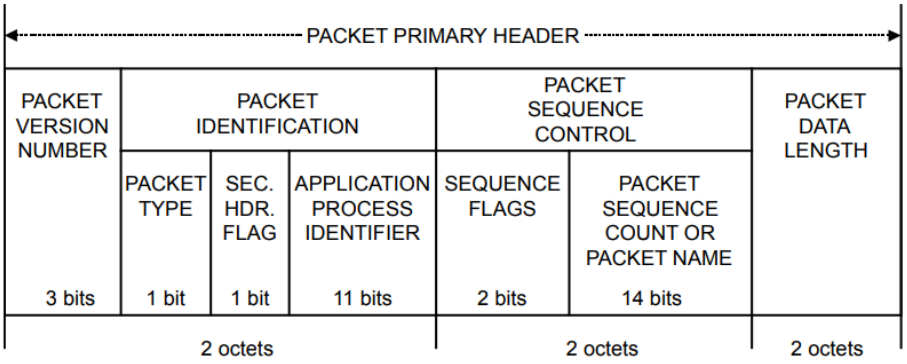

Fig. 3.2: CCSDS Space Packet Primary Header Structure [2]

This header includes information about the type of packet, its order in a data stream, and the packet's length. These information fields inspired the structure of the designed LACE packets. The structure of the LACE packets is shown graphically in Figure 3.3.

Address	Address (32-bit)	Bit Number 7	6	5	4	3	2	1	0	
Sync Word	Sync Word	0x35								Sync Word (32 bit)
		0x2E								
		0xF8								
		0x53								
0x0000	0x0000	Sequence Flags					Routing Identifier (See RID sheet) (3 MSB)			Packet Structure Header (Packet Data Length is number of 32-bit words in rest of packet)
0x0001		Routing Identifier (See RID sheet) (8 LSB)								
0x0002		Packet Data Length (16 bit)								
0x0003										
0x0004	0x0001	System Clock Milliseconds (32 bit)								Packet Timing Header Information
0x0005										
0x0006										
0x0007										
0x0008	0x0002	Real Time Clock (32 bit)								
0x0009										
0x000A										
0x000B										
0x000C	0x0003	Granule 1								All granule data must be 32 bits (padded)
0x000C + (1*granule size in bytes)	0x0003 + (1* granule size in 32-bit words)	Granules 2 through N-1								
0x000C + (N - 1 * granule size in bytes)	0x0003 + (N-1* granule size in 32-bit words)	Granule N								
0x000C + (N * granule size in bytes)	0x0003 + (N* granule size in 32-bit words)	CheckSum (32 bit)								32-Bit Checksum (XOR every word in packet)

Fig. 3.3: LACE Packet Structure

Several key decisions were made in the formation of this packet structure. First, data fields lengths and orders were chosen to enable a 32-bit based system. This was done because the entire system's architecture is 32-bit based. Conversion from other data word sizing is done as necessary, such as from the 8-bit based UART communication port.

Second, the primary header information (shown in red), was chosen to enable packet ordering, packet identifying, and packet length checking. This functionality is performed by the sequence flags, the RID, and the Packet Data Length fields, respectively.

Third, the addition of timing fields (System Clock Milliseconds and Real Time Clock) as a secondary header was inspired from the design of the Space Weather Probes 2 system by Wallace [7]. This enables timestamping of received and peripheral-created data with millisecond precision.

Lastly, a 32-bit checksum is appended to the end of the user data (granules). This is a checksum in name only as the current structure for computing this check word is a bitwise XOR on every word in the LACE Packet prior to the check word, not including the sync word. A more robust form of checksum was not chosen since this check word is

only employed to check data transmitted through the controller to peripheral FPGAs over transceivers, which have their own form of data validation using the LiteFast IP [4], or is internal to the controller FPGA. Externally received and transmitted data uses the checks built into Ethernet frames or CCSDS Space Packets.

It is important to note that there is a sync word of 0x352EF853 transmitted before each LACE Packet to provide a consistent method of parsing LACE Packets when they arrive at their destination. To further clarify the LACE Packet Structure, see Table 3.1 for descriptions of each field.

Table 3.1: LACE Packet Descriptions.

Field Name	Length (bits)	Description
Sync Word	32	Field not technically part of LACE Packet, but always transmitted before each packet for delimiting. Always value of 0x352EF853.
Sequence Flags	5	Packet's location in data stream, 0-30, wrap back to 0 after 30. Use 0b11111 when packet is standalone.
Routing Identifier (RID)	11	Packet identifier used for determining the routing of the packet through the system. Maps to and from CCSDS APIDs (UART) and source MAC (MAC) Addresses (Ethernet).
Packet Data Length	16	Represents the number of 32-bit data words following in the LACE Packet *after* this field, including the checksum check word at the end of the packet.
System Clock Milliseconds	32	The system clock in milliseconds when the LACE packet was formed.
Real Time Clock	32	The real-time clock in seconds when the LACE packet was formed.
Granule (N number of user data words)	32 each	The user data to be sent within the LACE packet.
Check Word	32	The verification (check) word for the LACE packet. Is formed by performing a bit-wise XOR on word in the LACE packet prior to this check word. Note: does not include the preceding sync word.

3.1.1 IEEE 802.3ab Ethernet Packet

Ethernet packet framing was also considered as the scheme between FPGAs within

the network. The standard used by the Gigabit Ethernet Core used (CoreTSE [8]) is the 1000BASE-T, which uses the Institute of Electrical and Electronics Engineers (IEEE) 802.3ab Ethernet packet definition. See Figure 3.4 for the IEEE 802.3 Ethernet packet definition.

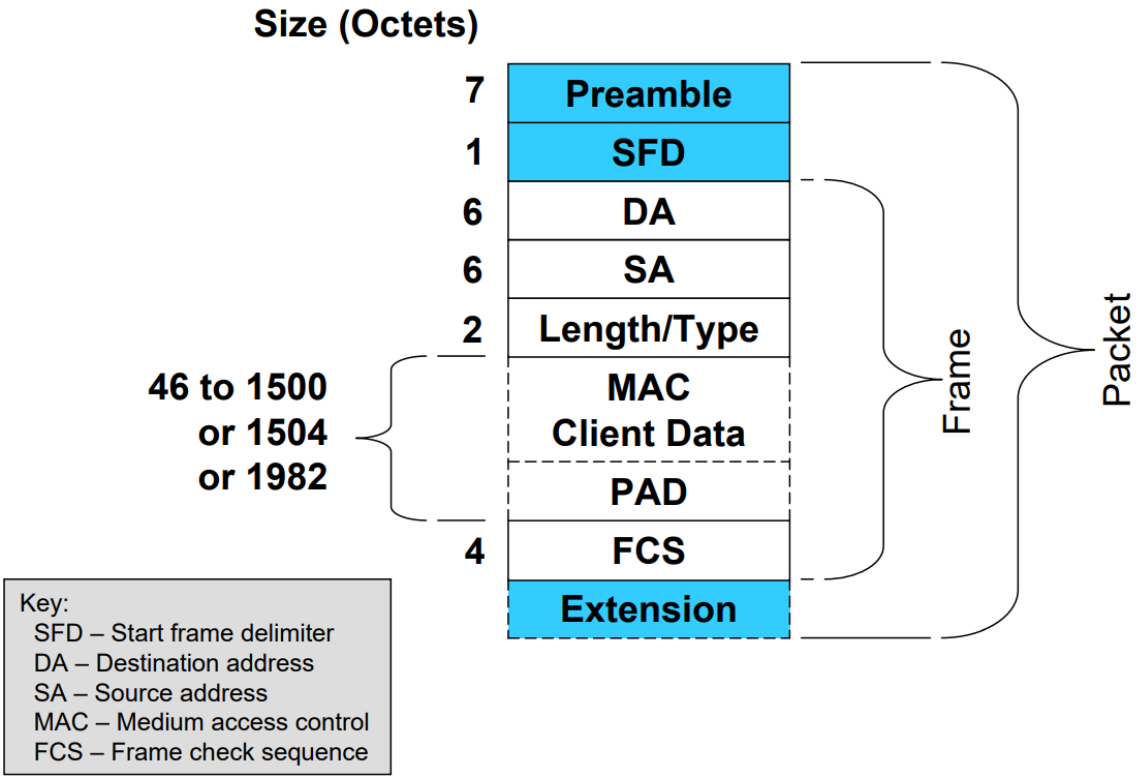

Fig. 3.4: IEEE 802.3 Ethernet Packet Definition [3]

However, this was not chosen as the header fields would have to be custom implementations to route the data around the device accordingly. This could lead to confusion in melding standards into a custom packet format. Thus, the LACE packet structure was chosen over the IEEE 802.3ab packet structure.

3.2 Packet Router

The center of the routing architecture is a Packet Router. This router takes in a number of LACE Packet data streams (as described in Figure 3.3) and produces the same number

of LACE Packet data streams out, routing these streams according to their RID fields. A diagram of this structure is shown in Figure 3.5. Each source buffer's output connects to the input of every other source's output multiplexer (MUX), seen by the different colored arrows, left unconnected for clarity. The Round Robin Scheduler in the center controls which sources buffer can output to the desired destination(s).

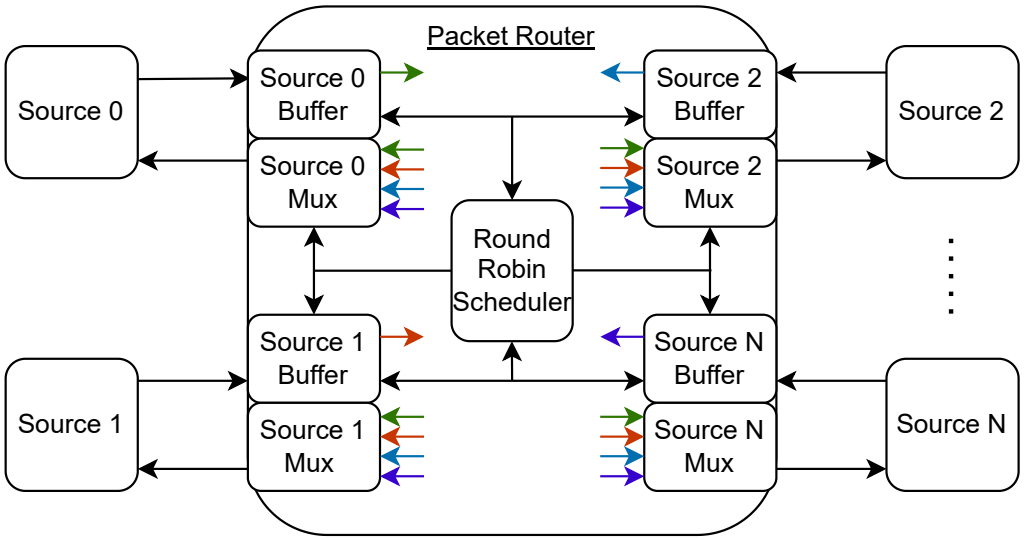

Fig. 3.5: Packet Router Diagram

The packet router was designed to be able to route incoming data to any number of output data streams. For example, this allows data to be received via Ethernet, assigned an RID according to the source MAC address (see section 5), and routed to all of the peripheral FPGAs in the network. Additionally, the Simulink model for the Packet Router can be seen in Appendix C.1.

3.2.1 Round Robin Scheduling and Forwarding

The round robin scheduler handles the internal routing logic of the Packet Router. Each source's buffer reports when it has data that is ready to be sent and the data's RID. The scheduler iterates through each source, checking whether each one is ready. As soon

as it comes to a source that is ready, the scheduler looks up destination(s) that packet goes to for that given RID. This lookup is performed on a lookup table that is loaded on FPGA boot according to a spacecraft-specific table associating RIDs to CCSDS APIDs and/or Ethernet MAC Addresses.

The scheduler configures the appropriate source's multiplexers (MUXes) to forward the ready-to-send source's stream. The scheduler then gives the ready-to-send source a signal to allow it to send, and the source puts a busy flag high as it transmits its data.

The round robin scheduler continues checking each source while the ready-to-send source is sending its data. When the scheduler comes to another ready source, it compares any busy sources' current routes using bitwise logic and if the next-in-line source's desired destinations don't conflict with any currently busy routes, the scheduler allows that next-in-line source to send after configuring the appropriate MUXes. If there is a conflict, the scheduler halts and waits until all of the next-in-line source's routes are ready, thus preventing starvation and reducing maximum latency, but reducing maximum aggregate bandwidth.

3.2.2 External Source Interface

The Packet Router was designed to be scalable and flexible to work with a variable number and different types of sources. This was done by designing the input and output interfaces to be standardized within the system. For a source to connect with the Packet Router, it must output a LACE packet bus and must take in a LACE packet bus. A LACE packet bus stream follows the LACE packet definition in section 3.1 and the signals it contains are:

1. Data_Line: 32-bit word containing packet data
2. Data_Valid: a boolean flag when the data on Data_Line is valid
3. Packet_Done: a boolean flag that is high when the last Data_Valid pulse of the packet is given (XOR check word)

A diagram showing the generic structure of any source interface to the Packet Router is shown in Figure 3.6.

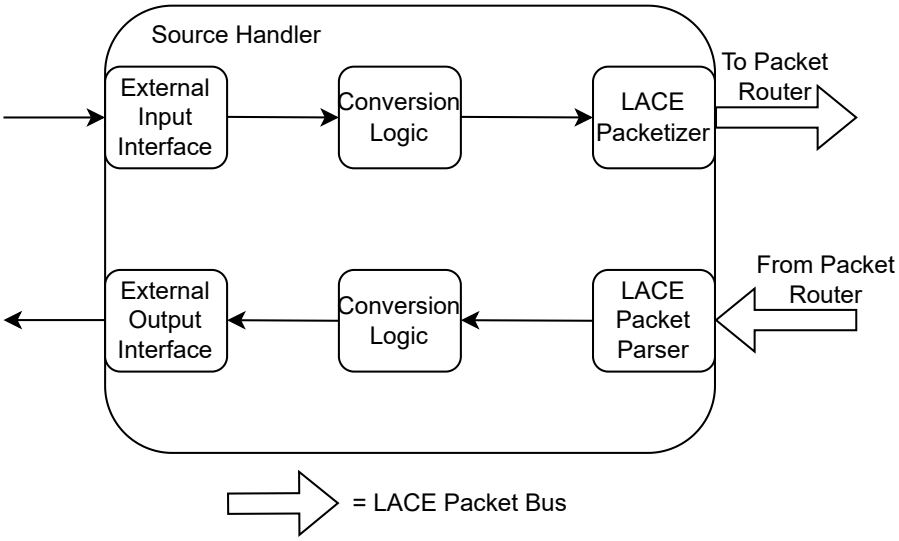

Fig. 3.6: Generic Source Handler Diagram

These source interfaces that were developed are referred to as handlers. The handlers developed for this system include a a LiteFast handler, an Ethernet handler, a UART and Space Packet handler, and a Command and Control handler. The specific designs for these handlers, how they integrate with the Packet Router, and their details are discussed in Chapters 4, 5, 6, and 7.

3.3 LACE Packet Handling

Since the LACE network firmware architecture is built around LACE packet data buses, reusable models were built to simplify the creation and parsing of these LACE packet data buses. To handle the creation of a LACE packet bus from a stream of data, the `LACE_packetizer` was created and is explained in Section 3.3.1. To handle the validation and optional parsing of a LACE packet bus stream of data, the `LACE_packet_parser` was created and is explained in Section 3.3.2.

3.3.1 LACE Packetizer

The LACE packetizer is responsible for taking in a stream of data, represented with the three signals `data_in`, `data_in_valid`, and `data_done` and creating a valid LACE packet bus and streaming the data out. Additionally, it takes in the RID, system_clock, and real-time clock (RTC) to properly populate the LACE packet header and check word as explained previously in Section 3.1.

To accomplish this, the LACE Packetizer uses a set of First In, First Out (FIFO) blocks to buffer data and header information until all user data is presented. It then iterates through the formed header data, presented user data, and calculated check word data, outputting the stream on its `LACE_packet_bus_out` port. The Simulink Referenced Subsystem diagram of the LACE Packetizer is shown in Figure 3.7.

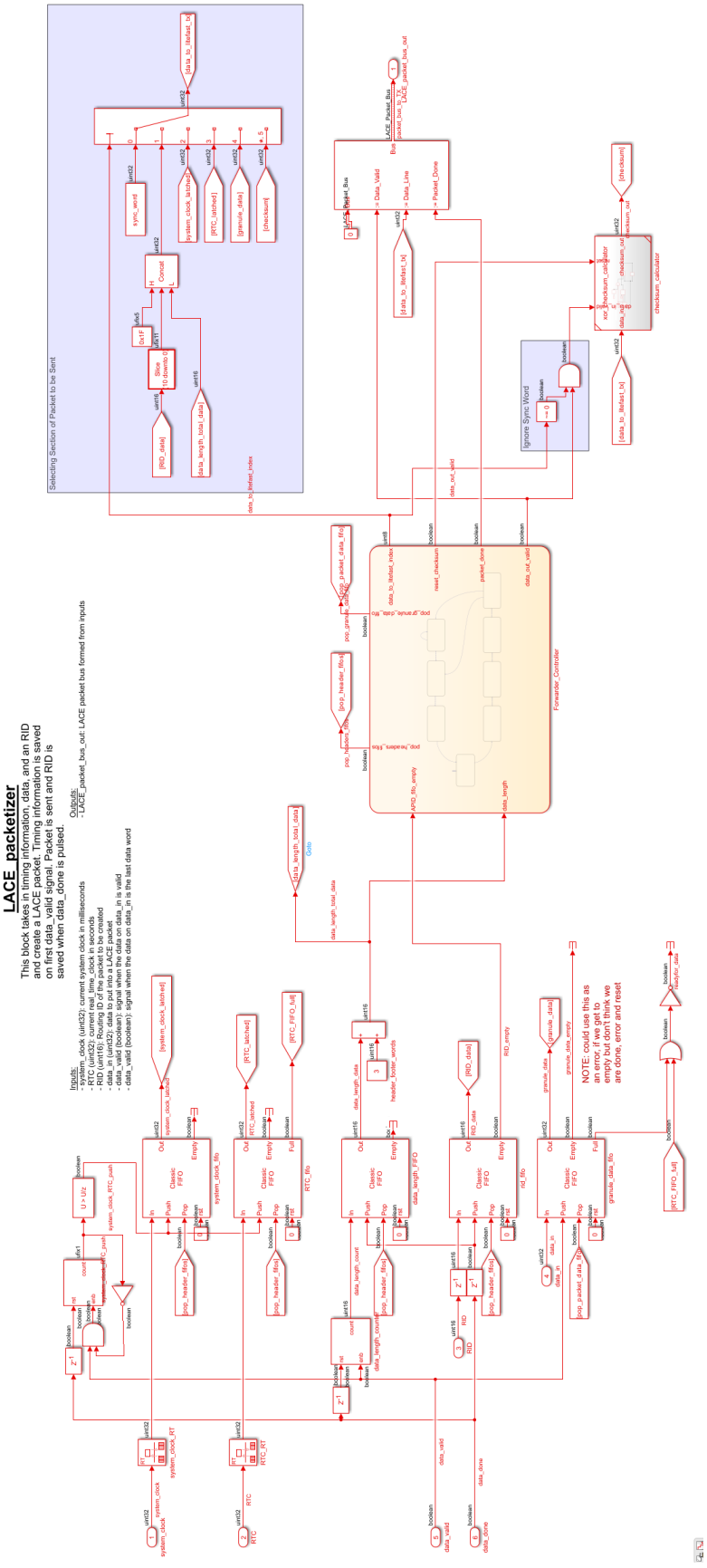

Fig. 3.7: LACE Packetizer Simulink Reference Subsystem

3.3.2 LACE Packet Parser

The LACE packet parser is responsible for taking in a LACE packet stream via a LACE packet bus input and checking the header and checksum information to validate the the input as a LACE packet stream. It does this by waiting until it sees the sync word appear on the input LACE packet stream, then iterating through the header, parsing and saving each piece of the header. After all user data and the check word has been presented, known by checking the Packet Data Length, the parser compares its computed XOR check word against the received check word. If they match, the validated packet stream is forwarded on the output ports with the parsed header information. The Simulink Referenced Subsystem diagram of the LACE packet parser is shown in Figure 3.8.

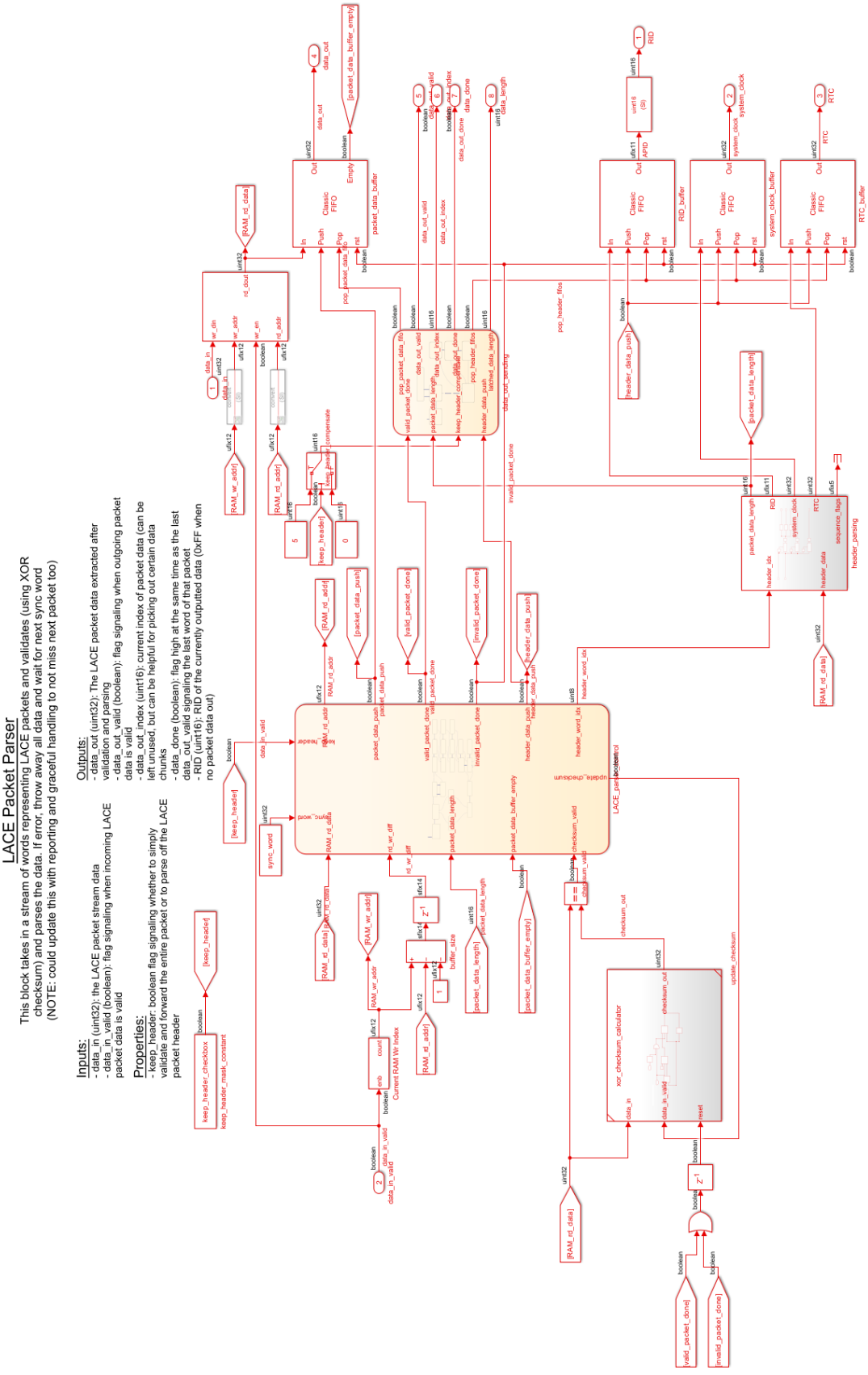

Fig. 3.8: LACE Packet Parser Simulink Reference Subsystem

Additionally, the LACE packet parser was designed as a Referenced Subsystem in Simulink configured with a subsystem mask. This allows internal block parameters to be passed up to the model that references the subsystem and configure options without changing every instance of the subsystem. Since there are places in the LACE network firmware that require parsing out the user data from the LACE packet stream and other places when the full packet with the header and footer is required with just validation, a parameter was "masked" that allows the user to select whether to keep the LACE header and footer on the validated LACE packet data stream. Figure 3.9 shows this subsystem mask on the LACE packet parser within the `controller_litefast_handler`, explained further in Section 4.3.

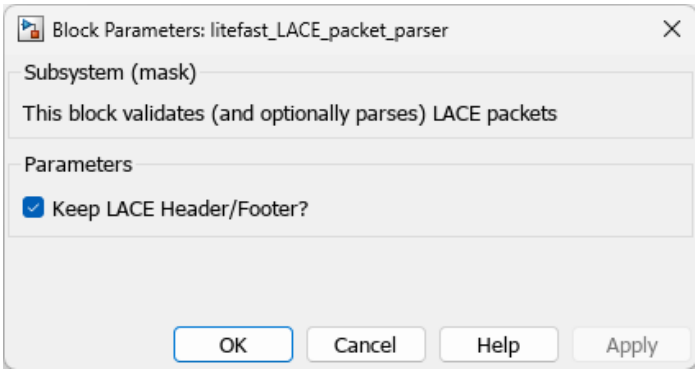

Fig. 3.9: LACE Packet Parser Subsystem Mask

CHAPTER 4

High-Speed Transceivers and LiteFast

4.1 LiteFast Introduction

Communication between the controller and peripheral FPGAs is achieved through the PolarFire family's built-in high-speed serial transceivers that can operate at up to 12.7 Gbps. These transceivers provide reliable, low-latency data transfer between FPGAs within the LACE network. The LACE network uses the LiteFast IP core [4], a Microchip-provided transceiver interface that performs 8b/10b encoding, word alignment, and error detection. Additionally, LiteFast manages flow control, framing and checking of data, and link maintenance when no user data is being sent. Figure 4.1 shows a typical application for the LiteFast IP cores from the LiteFast IP User Guide. This method is the exact application style used in the LACE network firmware, configured to use a 32-bit word.

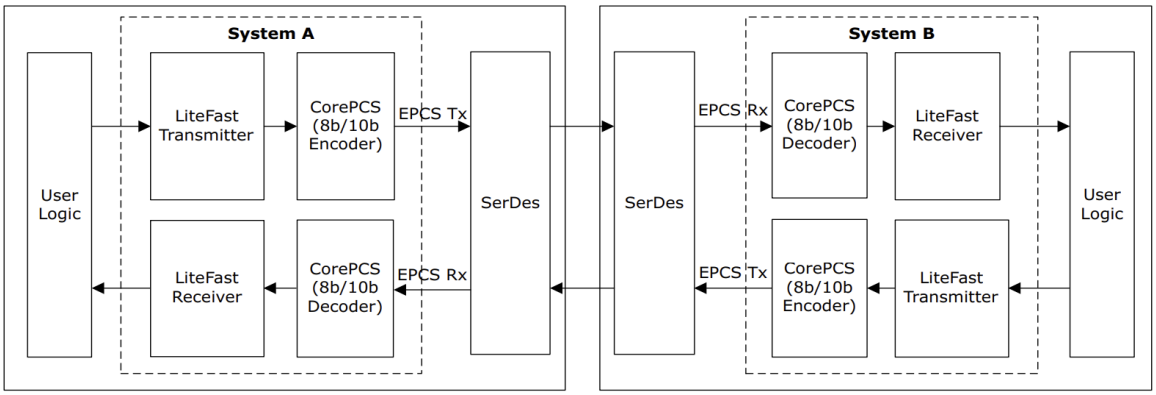

Fig. 4.1: LiteFast Usage Block Diagram [4]

To interface with the LiteFast transmitter block, user logic must raise a data ready flag, signaling that data is ready to be sent (`usr_data_rdy_tx_i`). The LiteFast transmitter will provide back a `req_usr_data_tx_o` flag, signaling the transmitter is ready for user data.

Two clock cycles later, the user logic then provides the a valid flag `usr_data_val_tx` along with the data (`req_usr_data_tx_i`) on the same clock cycle. It is important to note the `usr_data_val_tx_i` signal must be provided to the LiteFast transmitter *exactly* two clock cycles after the `req_usr_data_tx` signal goes high, any less or more delay and the LiteFast transmitter sends incorrect data. The user logic can provide up to 128 bytes of data in one transaction or $\frac{128\ \text{[bytes]}}{4\ \text{[bytes/word]}} = 32$ [words] as the maximum number of 32-bit words per transaction. A timing diagram of how to interface with the LiteFast transmitter block is shown in Figure 4.2, including the two clock cycle delay mentioned.

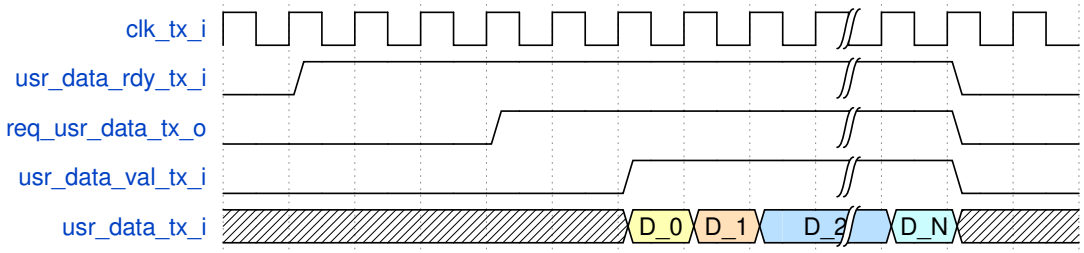

Fig. 4.2: LiteFast Transmitter Timing Diagram

The LiteFast receiver operates in a similar way to the LiteFast Transmitter, but simpler. When data is received and decoded, the LiteFast receiver provides a valid flag (`usr_data_val_rx_o`), signaling when the user data received, on the output line `usr_data_rx_o`, is valid for the user logic to consume. Additionally, there is an error signal output by the LiteFast receiver signaling when an error has occurred (`crc_err_rx_o`). A timing diagram showing an example successful receive transaction is shown in Figure 4.3.

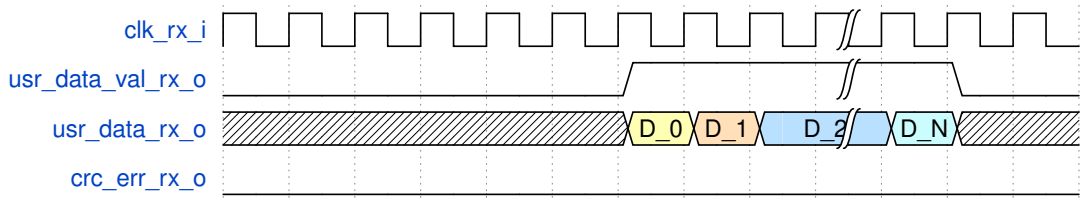

Fig. 4.3: LiteFast Receiver Timing Diagram

4.2 LiteFast Integration in Libero SoC SmartDesign

The LiteFast Transmitter, LiteFast Receiver, and the transceiver interface modules were instantiated and configured in Libero SoC using the SmartDesign tool. When instantiating a new transceiver interface, a bit rate must be specified in both the `PF_PLL`, responsible for generating the clocks necessary for transceiver operation, and the `PF_XCVR_ERM`, responsible for providing the interface to the user logic to use the transceiver. After selecting a desired bit rate, there will be two resulting clocks generated for the transmitting and receiving. The bit rate selected for this research was 10 Gbps, which resulted in a clock frequency of 250 MHz for both the transmitting and receiving clock domains. The maximum rate the PolarFire transceivers support is 12.7 Gbps, however 10 Gbps was selected to meet timing constraints and still provide a bandwidth significantly higher than the input and output rate of the system of approximately 1 Gbps.

The SmartDesign showing the connections between all of the modules necessary to interface with the FPGA's transceivers at 10 Gbps is shown below in Figure 4.4. The configuration for each of the IP blocks shown in Figure 4.4 can be seen in Appendix B.2. Additionally, the submodule SmartDesign blocks shown, `Transceiver` and `LiteFast_Receiver`, are shown in Appendices A.3 and A.2, respectively. The SmartDesigns `Transceiver` and `LiteFast_Receiver` have their detailed IP block configurations shown in Appendix B.5.

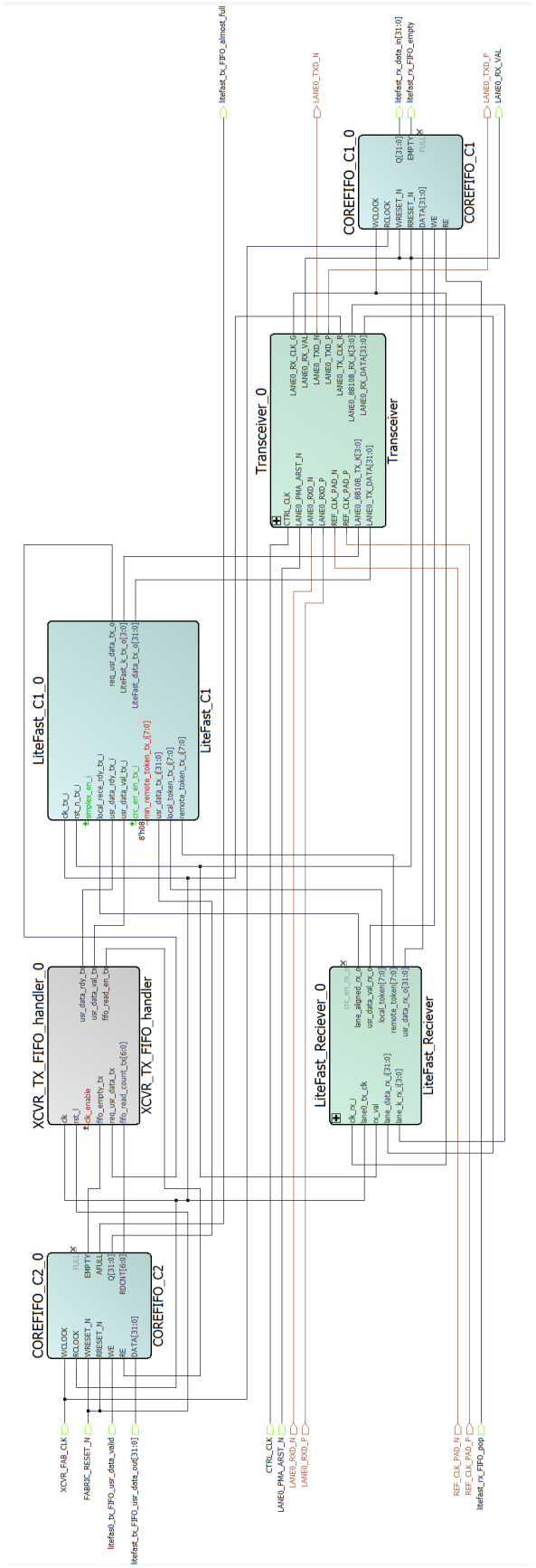

Fig. 4.4: XCVR_top SmartDesign

Because the transceiver subsystem transmit and receive subsystems operate in different clock domains relative to each other as well as to the main system fabric, Clock Domain Crossing (CDC) was a critical design consideration. To ensure reliable data transfer between these asynchronous domains, Microchip's CoreFIFO IP blocks were used on both the transmit and receive paths. These CoreFIFOs were configured to be dual-rate, meaning that the write side and the read side of the FIFO operate on separate clock domains. See Appendix B.2 for full configuration parameters.

4.2.1 LiteFast Transmitter Domain

Since the LiteFast transmitter has a unique requirement for sending it data as shown in Figure 4.2, a small Simulink model was designed to handle the popping of data from the read side of the transmit FIFO. This model would be responsible for pulling data out when the controller's main routing logic has pushed data onto the FIFO representing LACE packets to be sent to the peripheral over the transceiver lane. This model was named the `XCVR_TX_FIFO_handler` and its Simulink model can be seen below in Figure 4.5.

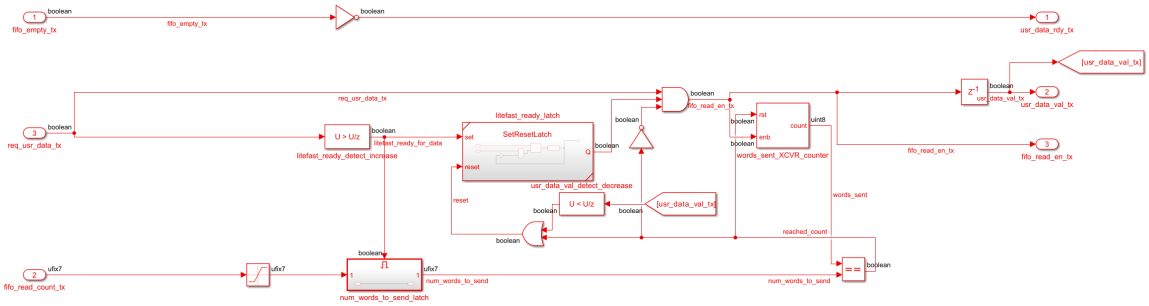

Fig. 4.5: XCVR_TX_FIFO_handler Simulink Model

This handler works by waiting in idle until the transmit FIFO is not empty. It then signals to the LiteFast transmitter that user data is ready. After the LiteFast Transmitter requests the user data, the handler waits 1 clock cycle before requesting the first 32-bit data word from the transmit FIFO and then sets the user valid flag high 1 clock cycle later to conform to the LiteFast transmitter timing diagram shown in Figure 4.2.

To maximize the efficiency and minimize the overall latency of the transceivers sending LACE packets between FPGAs, multiple data words are sent to the LiteFast transmitter at once. However, the LiteFast transmitter only accepts up to 128 bytes, or thirty-two 32-bit words. So, the `XCVR_TX_FIFO_handler` was designed to send in one transaction the minimum of two things: the current number of 32-bit words in the transmit FIFO when the LiteFast transmitter requests user data (shown in Figure 4.5 by the input `fifo_read_count_tx`) or the number 32 (shown in Figure 4.5 by the saturation block in the bottom left). A Simulink simulation of this handler unloading a FIFO with multiple words in it when the LiteFast transmitter requests user data and sending those words to the LiteFast transmitter is shown below in Figure 4.6 using the Simulink Logic Analyzer. It is important to note that the actual data being sent to the LiteFast transmitter are never input to the `XCVR_TX_FIFO_handler` as the data goes directly from the transmit FIFO to the LiteFast transmitter, but the timing between pop and valid signals must be synchronized appropriately. It can be seen that the handler adjusts to send a variable number of words at a time depending on how many are in the FIFO when user data is requested.

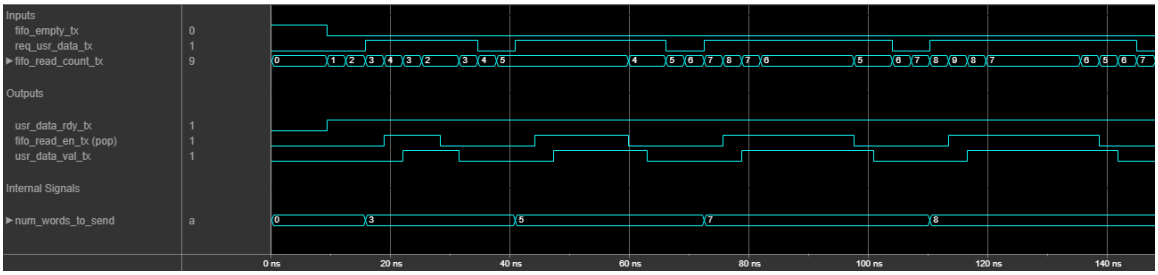

Fig. 4.6: XCVR_TX_FIFO_handler Simulation Results

The design for taking the LACE packet stream from the Packet Router and pushing onto the transmit FIFO will be explained in Section 4.3.

4.2.2 LiteFast Receiver Domain

The LiteFast Receiver CDC logic is such that the signals from the LiteFast Receiver can be connected directly to the write side of the receive domain CoreFIFO, configured to have

separate read and write clock domains. As seen in the SmartDesign in Figure 4.4, the `usr_data_val_rx_o` is connected to the Write Enable (`WE`) port of the FIFO and the `usr_data_rx_o` is connected directly to the `DATA` port of the FIFO. Then, it is up to the controller's main logic connected to the packet router to unload this FIFO as fast as possible and buffer the data elsewhere. This is because the LiteFat Receiver has no flow control input from the receive FIFO and the receive FIFO is only a finite number of words deep (see Appendix B.2 for full configuration). The design for unloading the receive FIFO and forwarding the LACE packet data to the Packet Router will be explained in Section 4.3.

4.3 LiteFast Integration with Packet Router

As explained in 3.2, the Packet Router was designed to have generic LACE packet bus outputs and inputs to and from each source. Because of this generic interface design, a method is needed for converting these generic packet streams to the specific connections needed to interface with the LiteFast transmit and receive FIFO logic explained in Sections 4.2.1 and 4.2.2.

The accomplish this interface to the transmit and receive FIFOs, the `controller_litefast_handler` was designed. The `controller_litefast_handler` Simulink model can be seen in Figure 4.7.

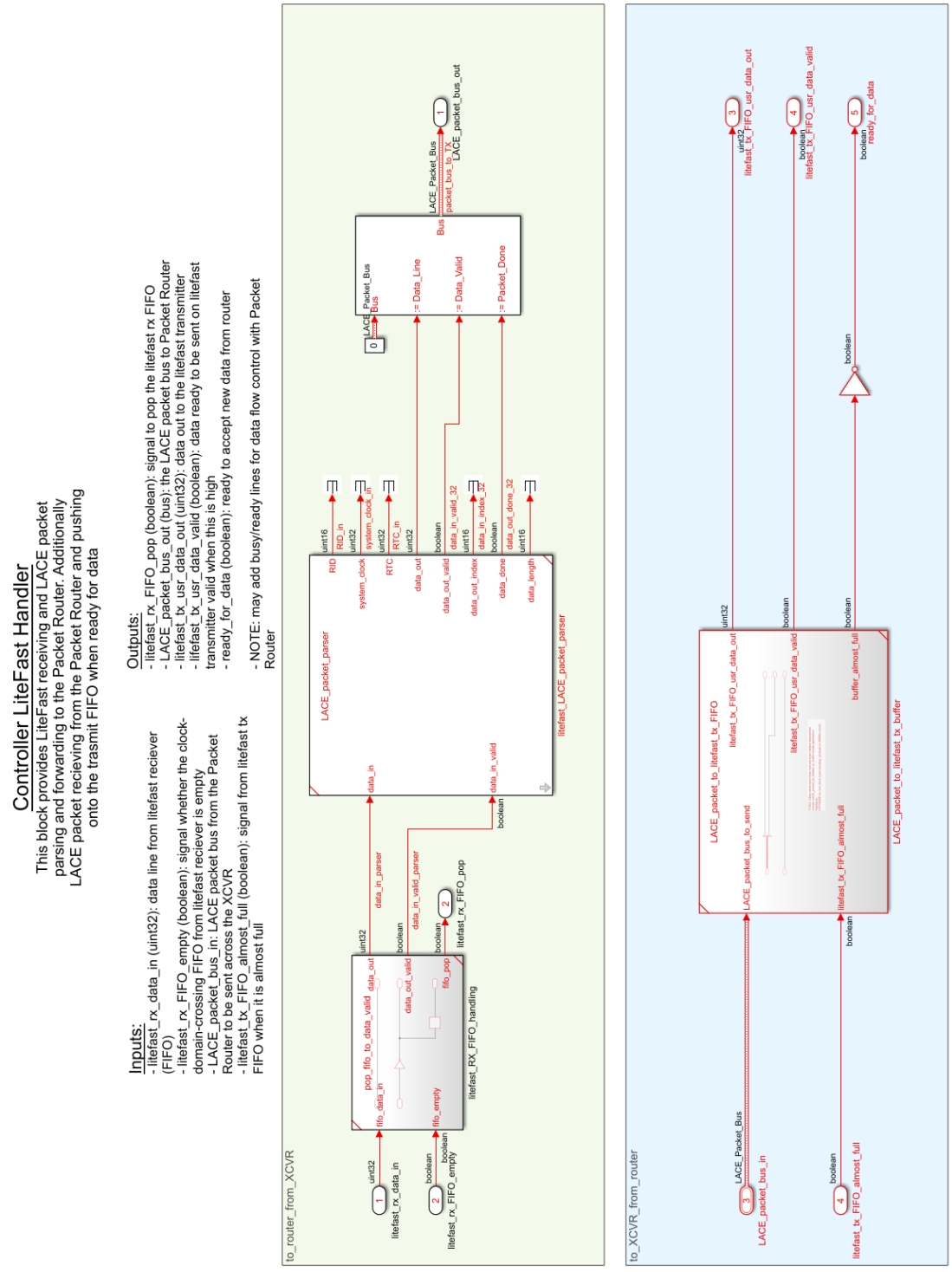

Fig. 4.7: LiteFast Handler Simulink

4.3.1 Receive Path

To handle the receive FIFO logic (seen in green in Figure 4.7), the `controller_litefast_handler` takes in the data output port (`litefast_rx_data_in`) and the empty signal (`litefast_rx_FIFO_empty`) from the read side of the receive FIFO. It outputs a `litefast_rx_FIFO_pop` signal to pop data off the receive FIFO as well as forms a LACE packet bus from the validated LACE packet stream from the peripheral FPGA to forward to the Packet Router. This is done immediately when the receive FIFO is not empty since there is no flow control back to the LiteFast receiver. The Packet Router will buffer the received packets until they have a chance to get forwarded to their destination(s). The LACE packet stream validation is done using the `LACE_packet_parser` configured to leave the LACE packet header on the data stream and only validate the stream and forward it. See Section 3.3.2 for more details on the LACE packet parser.

4.3.2 Transmit Path

To handle the transmit FIFO logic (seen in blue in Figure 4.7), it takes in a LACE packet bus and an almost full signal (`litefast_tx_FIFO_almost_full`) from the LiteFast transmit FIFO's write side. It outputs 3 signals: the data out signal `litefast_tx_FIFO_usr_data_out`, the valid flag `litefast_tx_FIFO_usr_data_valid` to push onto the transmit FIFO, and a `ready_for_data` signal back to the Packet Router for flow control, derived from the transmit FIFO's `litefast_tx_FIFO_almost_full` signal.

CHAPTER 5

Ethernet

The Ethernet interface serves as the primary external data ingress and egress point of the LACE network. It provides a standardized, high-speed communication channel between the controller FPGA and external ground systems or spacecraft payloads. The controller FPGA integrates a Microchip CoreTSE (Triple-Speed Ethernet) IP core configured for 1 Gbps full-duplex operation [8]. Physical connectivity is achieved through a Serial Gigabit Media-Independent Interface (SGMII) to an external PHY device.

Ethernet provides interoperability with standard spacecraft test systems [6] and supports packetized data exchange compatible with the LACE routing framework. Incoming Ethernet packets are parsed by the controller's Ethernet Handler and converted into internal LACE packets for routing to peripheral FPGAs or for configuration. Outgoing data from peripherals is re-encapsulated with Ethernet frames and transmitted externally.

This chapter details the implementation of the Ethernet interface using Libero Soc's SmartDesign tool and the design of the Ethernet Handler to interface with the Packet Router within the controller FPGA.

5.1 Ethernet Integration in Libero SoC SmartDesign

The CoreTSE, MIV_RV32 RISC_V soft-core processor, PF_IOD_CDR clock and data recovery module, and supporting IP and HDL blocks were instantiated and configured in Libero SoC using the SmartDesign tool, using the *PolarFire FPGA 1G Ethernet Loopback Using IOD CDR* as a reference guide [1]. The SmartDesign showing the connections between all modules necessary to implement 1 Gbp Ethernet using PolarFire's built-in PF_IOD_CDR and CoreTSE is shown in Figure 5.1. The following sections discusses further details and design decisions for major pieces of the Ethernet functionality, however configuration details for all IP blocks shown in Figure 5.1 can be seen in Appendix B.3.

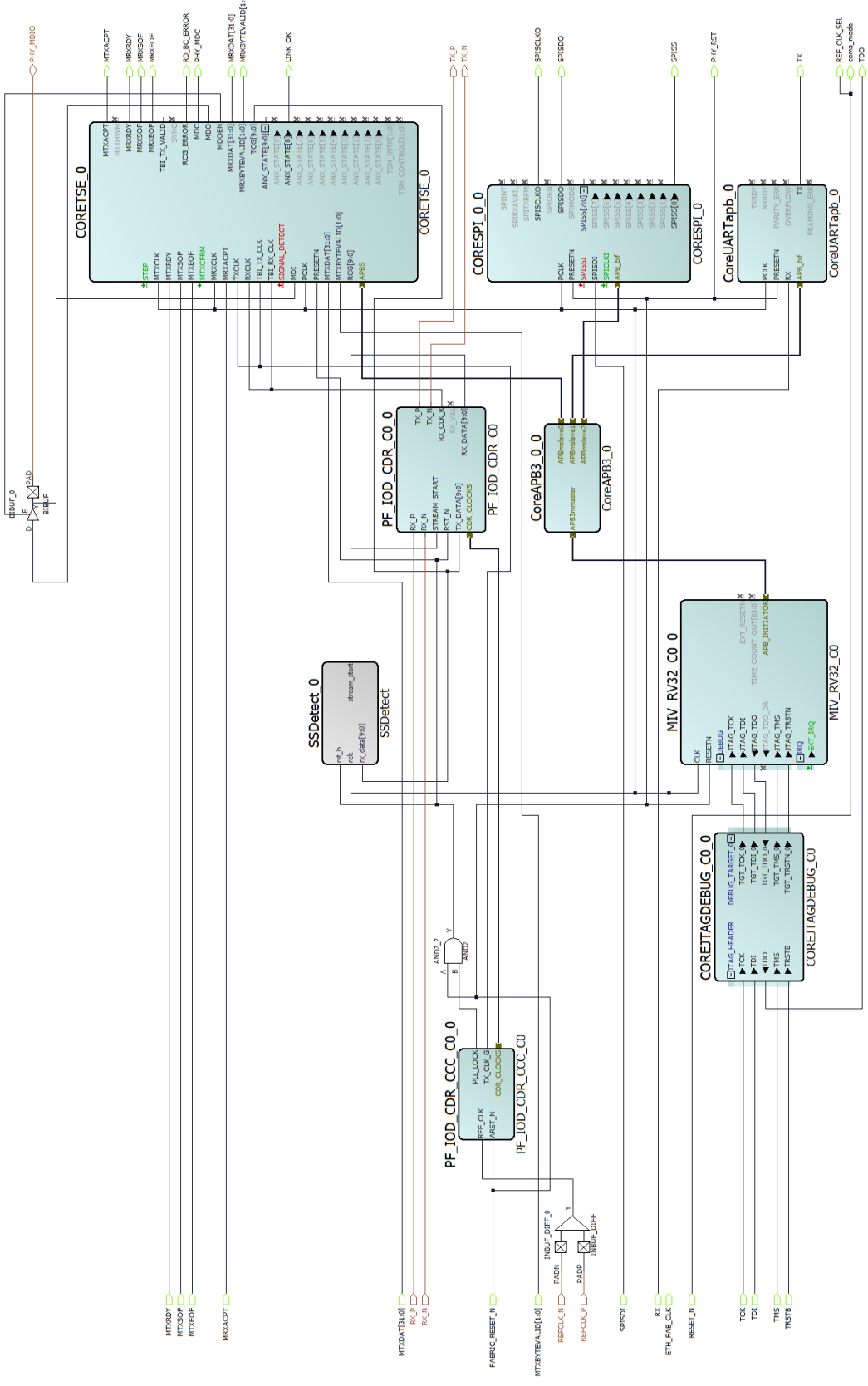

Fig. 5.1: Ethernet SmartDesign

5.1.1 CoreTSE and Processor

The CoreTSE IP core from Microchip provides the MAC layer functionality for LACE network's Ethernet interface. In this implementation, the core is configured for 1 Gbps full-duplex operation in Ten Bit Interface (TBI) mode. The MAC follows the IEEE 802.3x standard while as configured in its full-duplex mode. The core connects to an external PHY through an SGMII interface. The PHY performs the physical-layer serialization and deserialization, while the CoreTSE IP handles Ethernet frame buffering, CRC generation and checking, CDC logic, MAC address filtering, and statistic counters.

Within the Libero SoC SmartDesign, the CoreTSE block is connected to the MIV_RV32 soft-core processor via an Advanced Peripheral Bus (APB) interface, allowing the processor firmware to control MAC configuration registers during initialization. The core also connects to the PF_IOD_CDR block and encodes and decodes the recovered data and presents the resulting Ethernet frames to the controller's user logic. This interface to the controller's user logic and the Packet Router is discussed in detail in Section 5.2.

To configure the MAC Core Registers and perform PHY connection and autonegotiation, a C code file for the instruction code to run on the MIV_RV32 soft-core processor was edited from Microchip supplied demo resources [1]. This C code was compiled in SoftConsole to an Intel HEX formatted file and programmed to the PolarFire's secure Non-Volatile Memory (sNVM) to be loaded into the MIV_RV32 soft-core processor's instruction memory on FPGA boot. To setup this configuration, the processor's fabric Random-Access Memory (RAM) was setup to be loaded from sNVM with the compiled Intel HEX file within Libero SoC's "Configure Design Initialization Data and Memories" tool, as seen in Figure 5.2. This C code (`main.c`) can be seen fully in Appendix D.2.

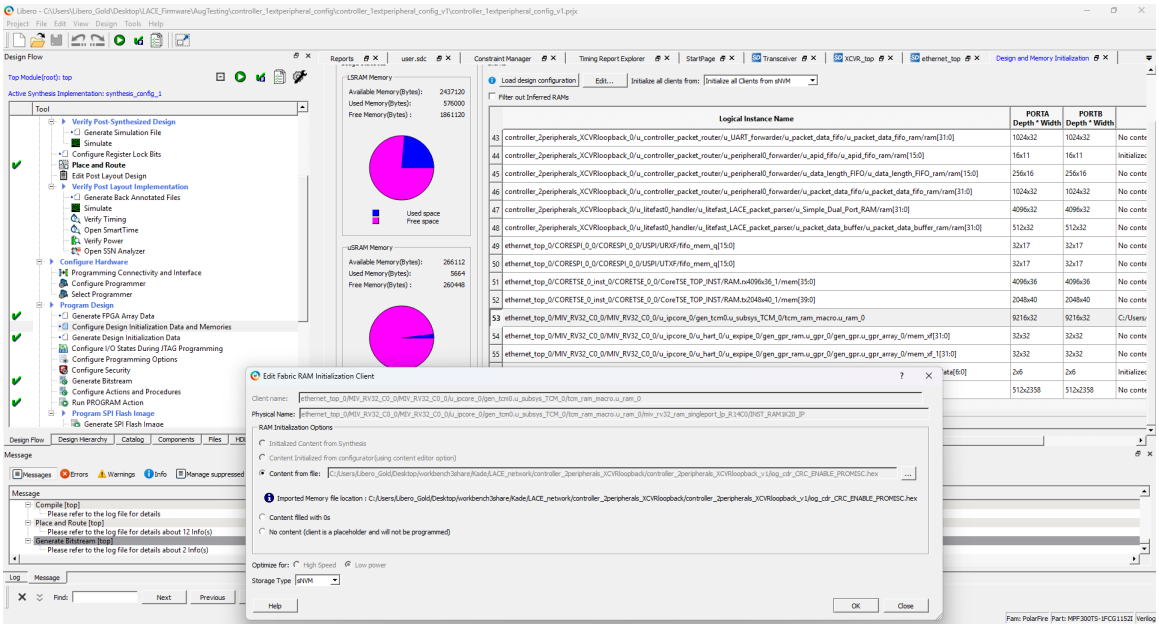

Fig. 5.2: Processor Instruction Code Initialization in Libero SoC

The processor executes this compiled C code. This instruction code executes the following primary functions:

1. PHY Initialization and Link Negotiation: The MIV_RV32 communicates with the external SGMII PHY via the Management Data Input/Output management interface to configure auto-negotiation, speed, and duplex parameters.

2. CoreTSE Configuration: The processor initializes the CoreTSE MAC registers, including MAC address assignment, frame length limits, and enabling CRC generation and checking. Once the interface is configured, it enables the Transmit (TX) and Receive (RX) paths for user logic.

3. Diagnostic and Control Support: The MIV_RV32 provides a register-based interface for sending diagnostic information over a Joint Test Action Group (JTAG) interface. However, this was not utilized in this research.

5.1.2 Clock and Data Recovery

The PF_IOD_CDR (PolarFire Input/Output Clock and Data Recovery) module is a key component that enables Gigabit Ethernet operation on the PolarFire FPGA without requiring an external PHY-supplied reference clock. The PF_IOD_CD recovers the embedded clock from the incoming serial data stream on the SGMII interface and regenerates a clean, phase-aligned clock for use by the CoreTSE MAC receive side. The CoreTSE performs the CDC to cross from this recovered receive domain to the supplied system clock and associated user logic domain.

In the LACE network design, the PF PF_IOD_CD is configured for 1 Gbps serial operation in SGMII mode. It interfaces directly with the differential receive (RX) and transmit (TX) pairs connected to the external PHY. The recovered receive clock drives the CoreTSE's RX domain, while a local transmit reference clock generated by a PF_IOD_CDR_CCC (Clock Conditioning Circuit) drives the TX domain. This configuration ensures that both directions remain frequency-locked while allowing independent phase alignment.

An additional advantage of using the SGMII standard is its ability to support either a MAC-to-PHY or MAC-to-MAC connection topology. In traditional Ethernet systems, the MAC interfaces with a discrete PHY for physical-layer signaling and media conversion. However, in compact embedded or spacecraft systems, such as small satellite payloads, where devices are located on the same board or within close proximity, the PHY can be bypassed entirely. The SGMII interface allows two MACs to communicate directly over short differential pairs without a PHY, maintaining full 1 Gbps data rates while reducing power consumption, system complexity, and latency. This flexibility makes SGMII particularly well suited for the LACE network architecture, where scalability and efficient inter-FPGA communication are critical.

5.2 Ethernet Integration with Packet Router

The Ethernet interface connects to the Packet Router and the rest of the LACE network through a dedicated Ethernet Handler, which acts as the translation layer between IEEE 802.3 Ethernet frames and the internal LACE packet protocol. This handler enables

Ethernet data streams to be integrated into the same routing framework used for UART and high-speed transceiver communication, ensuring a consistent packet format across all data paths.

The Ethernet Handler operates in two directions, receive and transmit, and performs the necessary frame parsing, packetization, and flow-control management required to interface the CoreTSE MAC with the LACE Packet Router. Figure 5.3 shows the `controller_ethernet_handler` Simulink model. The receive (green) and transmit (blue) portions of this handler are detailed in Sections 5.2.1 and 5.2.2, respectively.

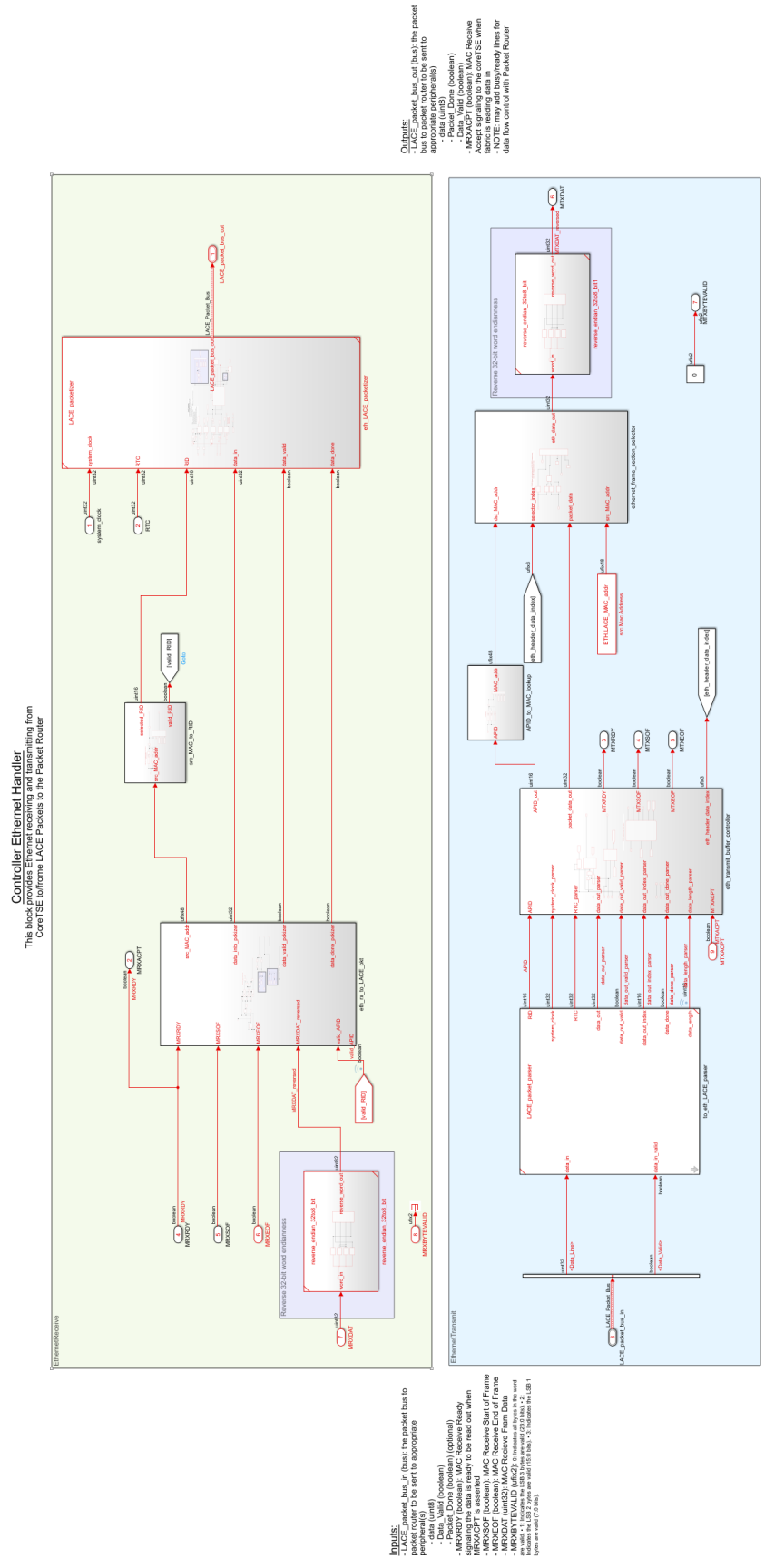

Fig. 5.3: Ethernet Handler Simulink

5.2.1 Receive Path

On the receive path, Ethernet frames are transferred from CoreTSE's RX side and into the Ethernet Handler via the input/output ports labeled with `MRX_`. An example receive timing diagram can be seen in Figure 5.4. As the frame is being received, the source MAC address is saved and used to lookup the corresponding RID, done within custom lookup logic in the `src_MAC_to_RID` block. This could be replaced in the future with something more scalable like a Content-Addressable Memory (CAM) block that uses the source MAC address to lookup the saved, appropriate RID to assign to the incoming Ethernet frame. Using the length/type field of the Ethernet frame to also contribute to the RID lookup was considered and would be a simple addition if useful for a satellite mission's network requirements, for example.

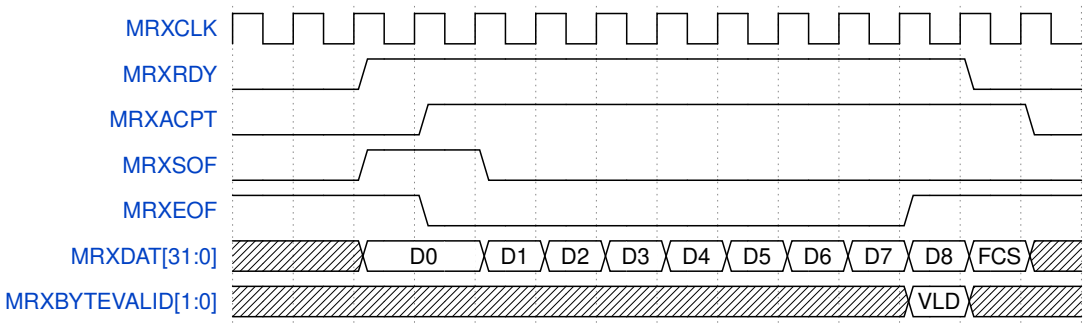

Fig. 5.4: Ethernet Receive Example Transaction Timing Diagram

To further explain the generic receive timing diagram shown in Figure 5.4, the CoreTSE asserts the `MRXRDY`, `MRXSOF`, and `MRXDAT[31:0]`, signaling the start of a frame. The core waits until the Ethernet Handler asserts the `MRXACPT`. The following `MRXDAT[31:0]` are valid on each subsequent clock cycle until the `MRXEOF` (end of frame) signal is asserted. `MRXBYTEVALID[1:0]` indicates the number of bytes that are valid in the last 32-bit word. `MRXBYTEVALID[1:0]` equal to 0 indicates all bytes in the word are valid, and this is the only case used in this LACE network research as all packets are 32-bit word (four byte) aligned. For completeness, `MRXBYTEVALID[1:0]` equal to 1 indicates the bits `MRXDAT[23:0]` are valid, `MRXBYTEVALID[1:0]` equal to 2 indicates the bits `MRXDAT[15:0]` are valid, and

`MRXBYTEVALID[1:0]` equal to 3 indicates the bits `MRXDAT[7:0]` are valid.

After the header, as the rest of the Ethernet frame is being received, the Ethernet Handler parses only the user data part of the frame and forwards those 32-bit words to a LACE Packetizer (see Section 3.3.1 for details on the LACE Packetizer). The resulting LACE packet bus is then output to be connected to the Packet Router to be routed to the appropriate destination(s) according to the RID mapped from the source MAC address. It is important to note that this user data within the Ethernet frame can be directly streamed to the LACE Packetizer before validating the Frame Check Sequence (FCS) at the end of the frame as the CoreTSE MAC validates the FCS before forwarding to the Ethernet Handler.

5.2.2 Transmit Path

On the transmit path, the incoming LACE packet stream from the Packet Router is parsed via a LACE Packet Parser, configured to remove the LACE packet header (see Section 3.3.2 for details on the LACE Packet Parser). The LACE Packet Parser produces the parsed RID, which is used via a custom lookup block (`RID_to_MAC_lookup`)) to lookup the destination MAC address. As mentioned in Section 5.2.1, this could be replaced with something like a CAM or a simple directly addressed RAM block.

This corresponding destination MAC address, along with the source MAC address of the LACE network hardware and a length/type field of 0, is formed into an Ethernet frame. This frame is output to the CoreTSE MAC over the ports labeled with `MTX_`. It is important to note that the FCS is left off the end of the Ethernet frame that is to be transmitted. This is because the the CoreTSE is configured to append a valid FCS. A generic transmit timing diagram can be seen in Figure 5.5.

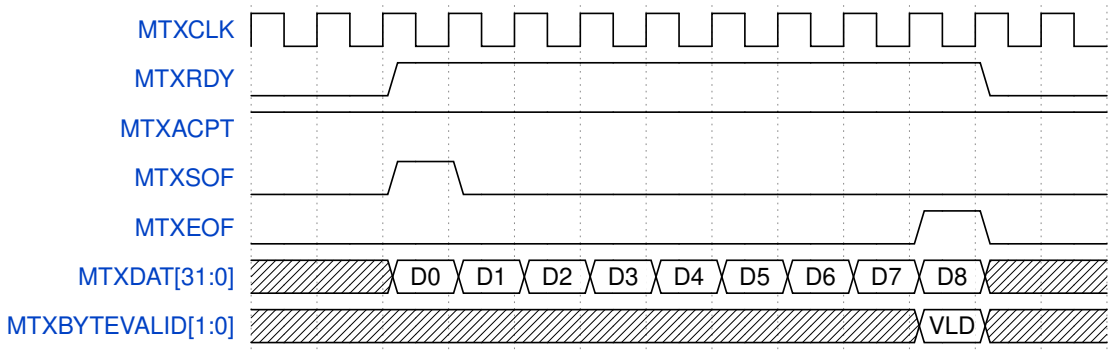

Fig. 5.5: Ethernet Transmit Example Transaction Timing Diagram

To further explain the generic receive timing diagram shown in Figure 5.5, the Ethernet Handler asserts the `MTXRDY`, `MTXSOF`, and `MRXDAT[31:0]` signaling the start of a frame. The handler waits until the CoreTSE asserts the `MTXACPT`, unless already asserted high. The following `MTXDAT[31:0]` are valid on each subsequent clock cycle until the `MTXEOF` (end of frame) signal is asserted. `MTXBYTEVALID[1:0]` indicates the number of bytes that are valid in the last 32-bit word. `MTXBYTEVALID[1:0]` equal to 0 indicates all bytes in the word are valid, and this is the only case used in this LACE network research as all packets are 32-bit word (four byte) aligned. For completeness, `MTXBYTEVALID[1:0]` equal to 1 indicates the bits `MRXDAT[23:0]` are valid, `MTXBYTEVALID[1:0]` equal to 2 indicates the bits `MTXDAT[15:0]` are valid, and `MTXBYTEVALID[1:0]` equal to 3 indicates the bits `MTXDAT[7:0]` are valid.

CHAPTER 6

UART and Space Packets

The Universal Asynchronous Receiver/Transmitter (UART) interface provides a low-speed communication channel between the LACE controller FPGA and an external spacecraft computer. While the Ethernet subsystem supports high-throughput data exchange, the UART interface focuses on simple, low-bandwidth communication intended for system configuration, monitoring, and control.

In spacecraft applications, UART remains a widely used protocol due to its simplicity, minimal resource footprint, and compatibility with many devices. This interface allows bidirectional command and telemetry exchange using packetized data conforming to the Consultative Committee for Space Data Systems (CCSDS) Space Packet Protocol [2], improving interoperability with existing spacecraft communication frameworks.

The UART interface is integrated into the same LACE Packet Router infrastructure as the Ethernet and high-speed transceiver subsystems. Commands received from the spacecraft computer are packetized and routed internally to the appropriate peripheral FPGA(s) or controller FPGA internal command and configuration logic, while telemetry data and responses are transmitted back over the UART link. This consistent packetized communication model simplifies routing logic, provides a uniform data structure across all interfaces, and supports scalability for future multi-channel configurations. This modular LACE network architecture also allows for the UART interface to be optionally completely removed, depending on spacecraft or mission constraints.

6.1 UART Integration in Libero SoC SmartDesign

Minimal effort within Libero SoC was needed to support UART functionality. This is because custom UART receive and transmit blocks designed in Simulink were used, modified from the Space Weather Probes 2 Simulink firmware by Wallace [7]. Both UART blocks

operate with a baud rate derived from the system clock through a programmable clock divider in the Simulink subsystem reference mask, allowing flexible reconfiguration of link speed if required. See Section 6.2 for more details on these custom UART blocks and their integration within the rest of the controller FPGA's firmware architecture.

Because the UART blocks were contained within the controller FPGA's Simulink firmware design, the only work required in Libero Soc is to connect the UART RX and TX lines to the top-level I/O of the FPGA. The chosen method for this is to simply have the two ports forwarded from the Simulink-generated HDL model to top-level ports in Libero SoC and assign the I/O constraints to General Purpose Input Output (GPIO) pins. See Appendix A.1 for the top level SmartDesign showing these ports forwarded to the top level. These pins were then connected to the corresponding RX and TX lines of a Raspberry Pi as a spacecraft emulator.

The other option considered for UART was to implement Low-Voltage Differential Signaling (LVDS) UART. LVDS is a popular data transmission standard that is more robust to noise and reduce radiated Electromagnetic Interference (EMI) due to the integration of differential signaling [29]. This option was not chose simply due to the single-ended receive hardware available on a Raspberry Pi. However, the researcher has tested the conversion from single-ended UART to LVDS by configuring the PolarFire FPGA's I/O to have differential input and output as seen below in Figures 6.1 and 6.2.

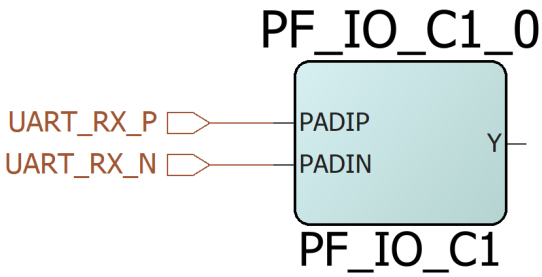

Fig. 6.1: Differential Receive PolarFire IO

Fig. 6.2: Differential Transmit PolarFire IO

6.2 UART and Space Packets Integration with Packet Router

The UART and Space Packet interface connects to the LACE through a dedicated UART Handler, which acts as the translation layer between CCSDS Space Packets transmitted over UART at 115,200 baud, no parity bits, 1 stop bit. The UART Handler operates in two directions, receive and transmit, and performs the necessary packet parsing and packetizing required to interface with the LACE Packet Router. Figure 6.3 shows the `controller_UART_handler` Simulink model. The receive (green) and transmit (blue) portions of this handler are detailed in Sections 6.2.1 and 6.2.2, respectively.

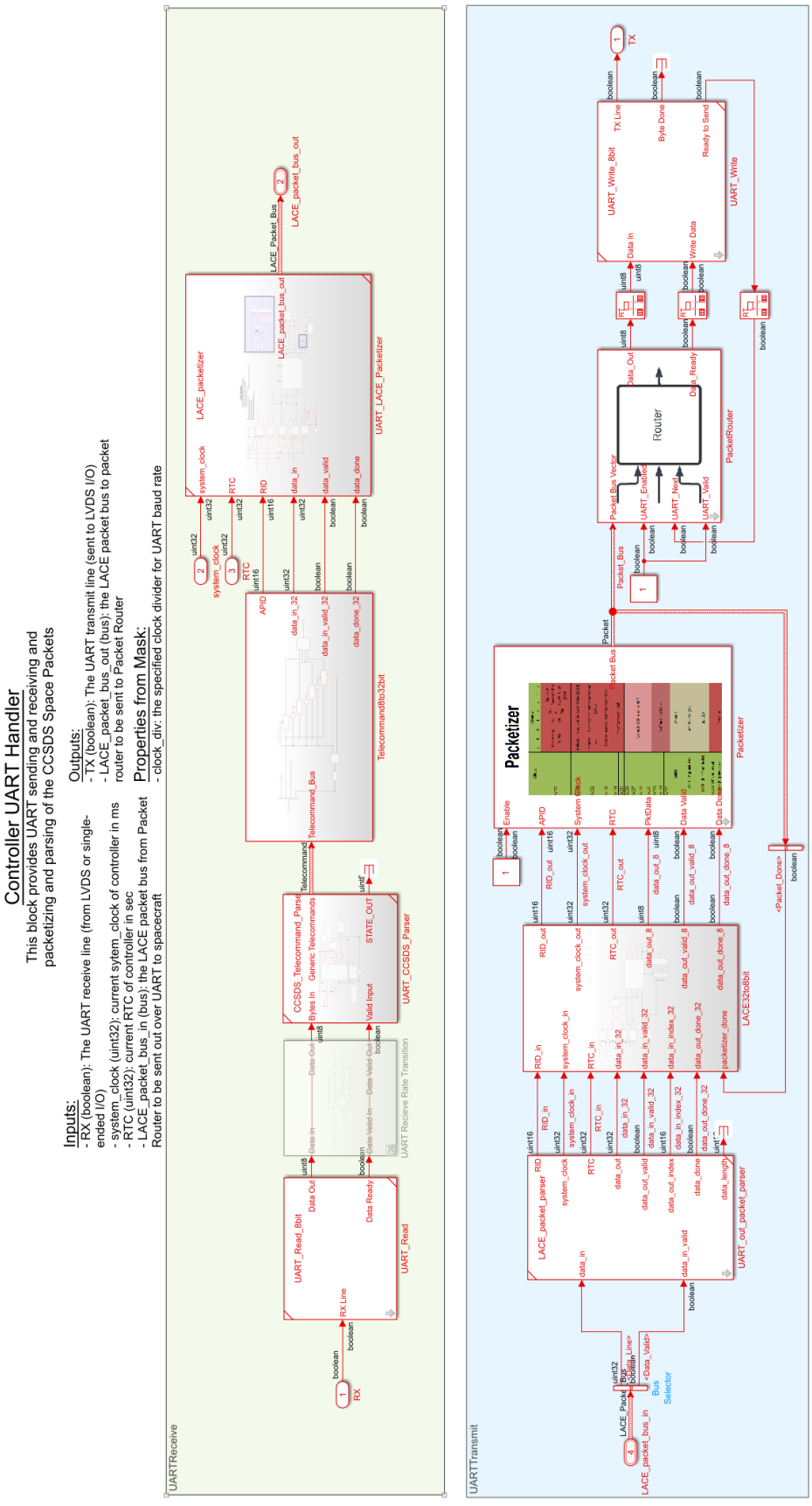

Fig. 6.3: UART Handler Simulink

6.2.1 Receive Path

On the receive path, CCSDS Space Packets are received via the serial receive line (RX). The `UART_Read` block extract bytes from the serial RX line assuming a 115,200 baud rate, no parity bits, and 1 stop bit. The resulting bytes are passed to a `CCSDS_Telecommand_Parser`, which parses CCSDS Space Packets and extracts the data from valid packets. This `CCSDS_Telecommand_Parser` was modified from the Space Weather Probes 2 Simulink firmware by Wallace [7]. Because CCSDS Space Packets are octet (byte) based, a byte to 32-bit word conversion is needed, also restricting the CCSDS Space Packet to have 4-byte (32-bit) aligned words to interface with the LACE network system. Figure 3.1 shows the overall CCSDS Space Packet structure and Figure 3.2 shows the details of the CCSDS Space Packet primary header. The `CCSDS_Telecommand_Parser` was built from these specifications.

The `CCSDS_Telecommand_Parser` also provides the APID of the received Space Packet. Because the RIDs were designed from the Space Packet Protocol as an 11-bit identifier, a one-to-one mapping may be used to convert APIDs to RIDs. The received packet data and the corresponding RID are passed to a LACE packetizer, which assembles a valid LACE packet and outputs a LACE packet bus to go to the Packet Router. See Section 3.3.1 for more details on the LACE packetizer.

6.2.2 Transmit Path

On the transmit path, the incoming LACE packet stream from the Packet Router is parsed via a LACE Packet Parser, configured to remove the LACE packet header (see Section 3.3.2 for details on the LACE Packet Parser). The LACE Packet Parser produces the parsed RID, which is used directly as the APID input for the CCSDS Packetizer, which was modified from the Space Weather Probes 2 Simulink firmware by Wallace [7] to work within the LACE network firmware.

The parsed data from the LACE Packet Parser is output in 32-bit words. However, CCSDS Space Packets are octet (byte) based and the CCSDS Packetizer expects 8-bit data words. So, a 32-bit word to 8-bit word buffer and converter was created. After this conversion, the resulting 8-bit data words are input into the CCSDS Packetizer.

The resulting packets are buffered by a different packet router, also modified from Wallace [7], and output via UART using the `UART_Write`. These packets follow the CCSDS Space Packet structure and the CCSDS Space Packet primary header structure, as shown in Figures 3.1 and 3.2, respectively. Additionally, the parsed system clock and RTC data points from the incoming LACE packet are appended to the CCSDS Space Packet after the CCSDS primary header, similar to the Space Weather Probes 2 telemetry design by Wallace [7].

CHAPTER 7

Command and Control

The Command and Control subsystem enables dynamic configuration of the LACE network at runtime through command packets received, typically over UART or Ethernet, as well as the creation of controller FPGA specific telemetry output. This system allows an external host or spacecraft computer to issue routing updates, control peripheral behavior, or request telemetry without requiring FPGA reprogramming. This command packet flow can be seen in Figure 7.1.

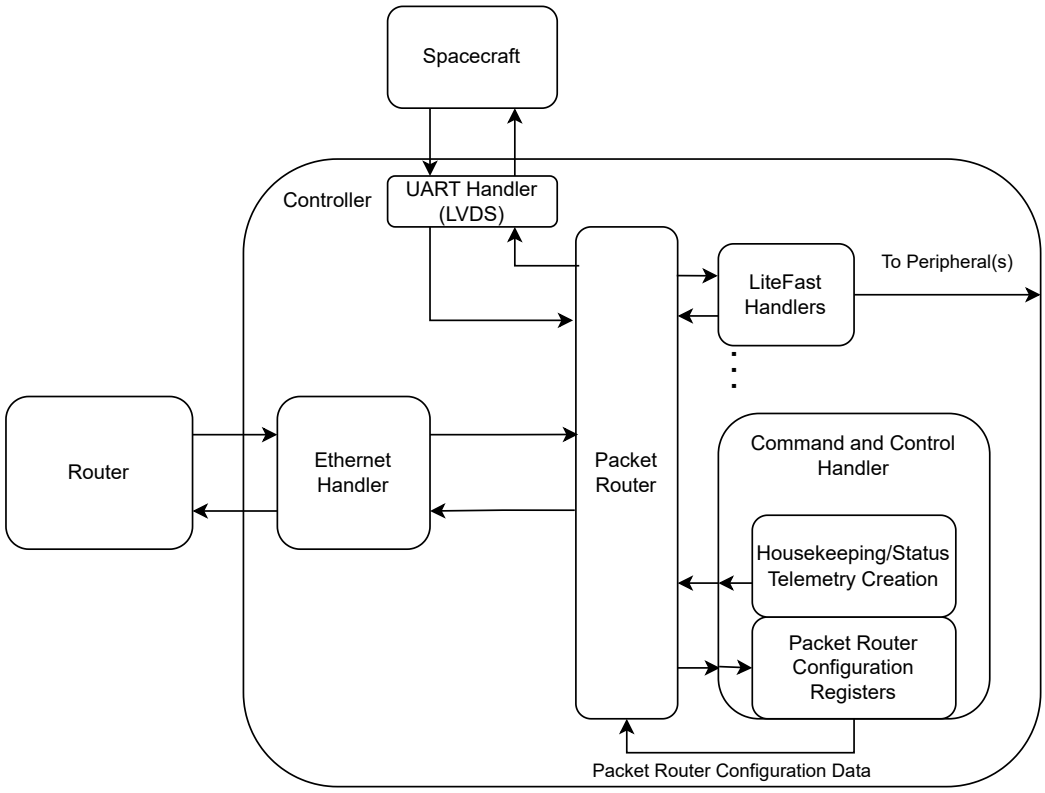

Fig. 7.1: Command and Control Data Flow

The command handling framework is implemented within the Command and Control Handler, a Simulink-generated subsystem that interfaces directly with the LACE Packet Router. This handler interprets incoming command packets, decodes their headers, and triggers corresponding configuration actions within the controller FPGA. Outgoing telemetry or confirmation responses are similarly packetized and routed through the same infrastructure for transmission to the external interface.

7.1 Command and Control Integration with Packet Router

The Command and Control Handler was designed following the same paradigm as the other handlers, with a LACE packet bus output and input to and from the Packet Router as its main interfacing method. Figure 7.2 shows the `command_control_handler` Simulink model. The receive (green) and transmit (blue) portions of this handler are detailed in Sections 7.1.1 and 7.1.2, respectively.

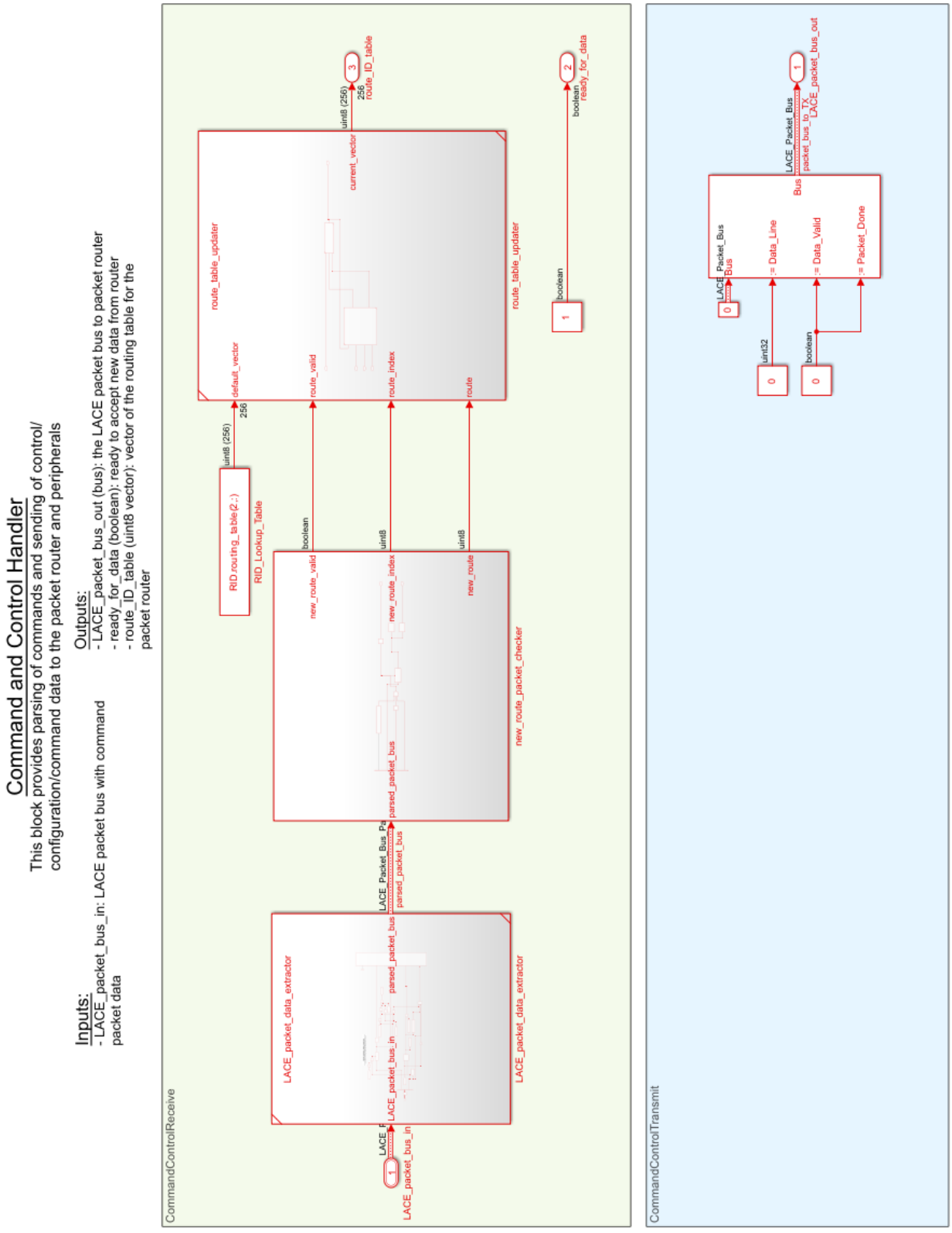

Fig. 7.2: Command and Control Handler Simulink

7.1.1 Receive Path

The Command and Control Handler takes in LACE packets from the Packet Router and parses out the data and the RID. That parsed packet bus is output and can be used as the input to any number of configurators that are waiting for a specific type of command or control packet. These configurators can then use the received packet data however is required to accomplish that specified command's purpose.

The only implemented command packet that the LACE network system is expecting is a route configuration packet. This packet, whose LACE packet structure can be seen in Figure 7.3, allows the Packet Router's internal routing table to be updated, thus allowing on-the-fly reconfiguring of which types of packets go to which destinations according to their RID. This routing table is output from the Command and Control Handler back to the Packet Router. The receive path currently has no buffers that could overfill, so the Command and Control Handler is always reporting that it is ready for data from the Packet Router.

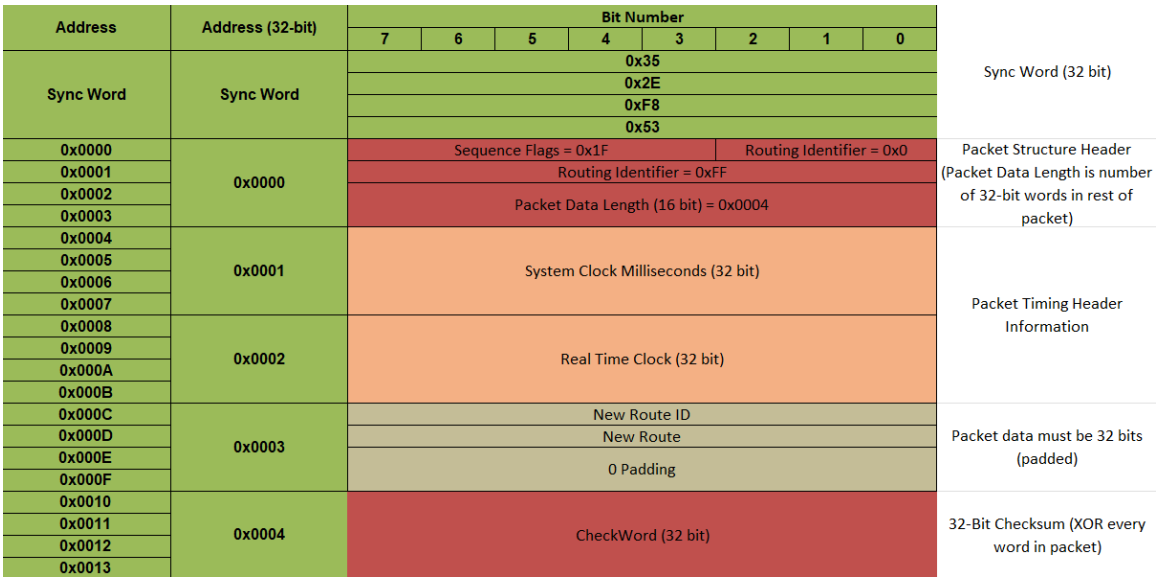

Fig. 7.3: Routing Table Update Command Structure

7.1.2 Transmit Path

As seen in Figure 7.2, there are no packets being transmitted from the Command and

Control Handler, though the architecture is in place to do so. This architecture was built in anticipation of being programmed to a custom LACE FPGA hardware stackup that will have hardware that the current development hardware doesn't have. This would include housekeeping Analog-to-Digital Converters (ADCs) that sample the power circuitry's voltages and currents. Then, this Command and Control Handler can be modified to produce a housekeeping/status LACE packet periodically to be routed to any destination on the LACE network, likely the spacecraft computer.

CHAPTER 8

Peripheral Application Interface

The Peripheral Application Interface enables the interfacing of any FPGA application that is to be programmed onto a peripheral FPGA to connect to the LACE network. This connection is done via the built-in high-speed transceivers on PolarFire FPGAs, with an additional layer using a Microchip-provided IP block called LiteFast. The implementation of the transceivers and LiteFast IP blocks in Libero SoC is the same on the peripheral FPGA as the controller FPGA, including the separate cross-domain FIFOs for the receiving and transmitting clock domains. See Chapter 4 for more details on this implementation and see Appendix B.2 for exact IP block connections and configurations.

After the configuration in Libero SoC, a `peripheral_application_interface` Simulink model was designed to handle the connection to the transceivers' receive and transmit FIFOs, parse the incoming LACE packet stream from the controller FPGA, and to provide a simple interface to the application to be able to transmit labeled packet data to be routed through the LACE network. Figure 8.1 shows the Simulink model for the Peripheral Application Interface. The controller to application (green) and application to controller (blue) portions of this interface are detailed in Sections 8.1 and 8.2, respectively.

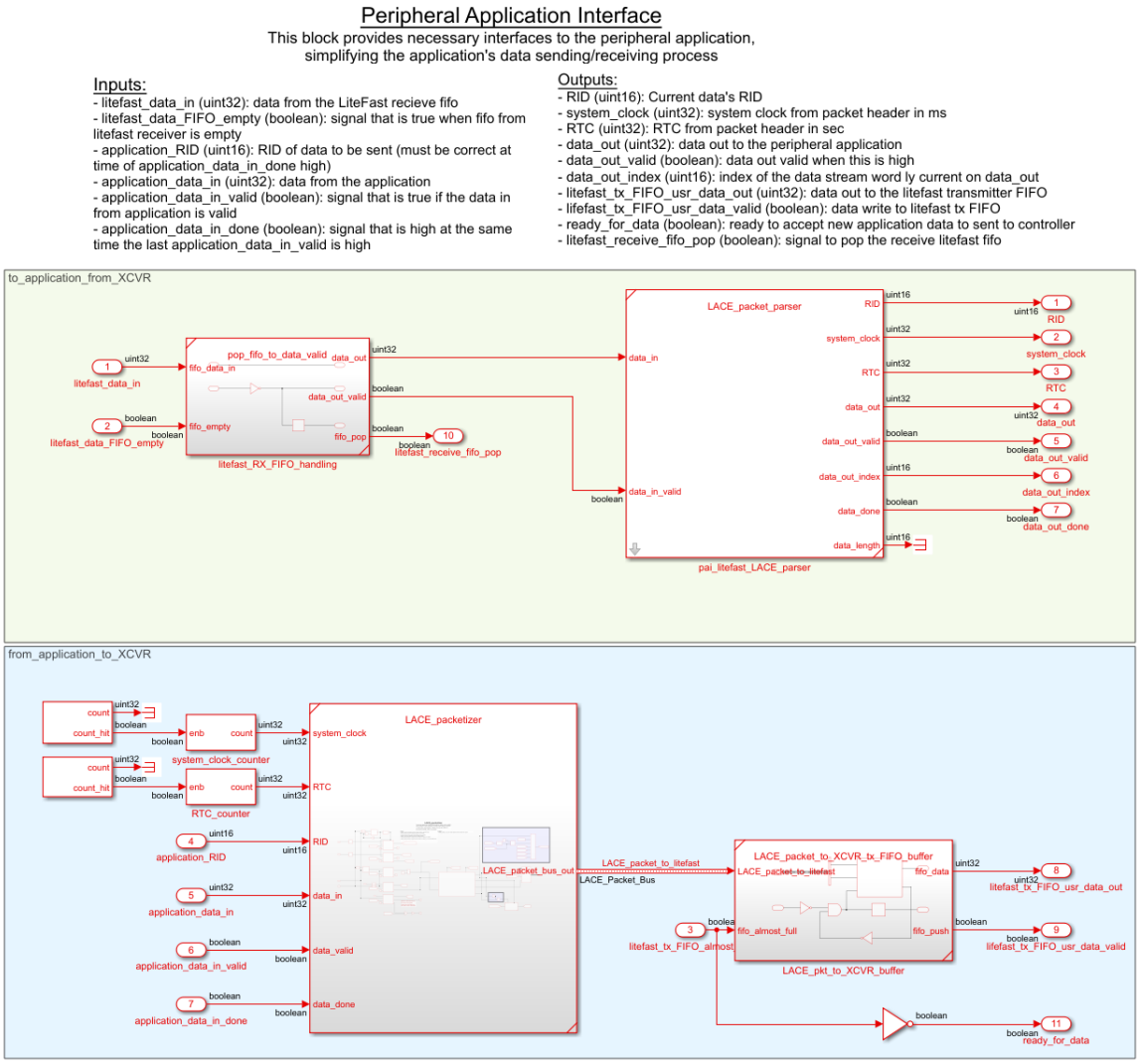

Fig. 8.1: Peripheral Application Interface Simulink Diagram

8.1 Receive Path: From Controller to Application

The Peripheral Application Interface is designed to connect directly to the LiteFast receive FIFO. The interface is designed to immediately pop data off of the FIFO when there is any data present since the LiteFast receiver has no backwards flow control to slow down its production of data. As the data streams in from the LiteFast receive FIFO, it is piped into a LACE Packet Parser See Section 3.3.2 for more details on the LACE Packet Parser. This parser validates the incoming LACE packets, throwing them away if received

with any errors, and also parses out the LACE packet header information into output ports for the application to consume. These header information ports, which can be seen as the output ports of the green section of the Simulink model in Figure 8.1, are `RID`, `system_clock`, and `RTC`. Simultaneously, the parsed packet data is presented to the application for consumption. An example timing diagram showing a generic transaction of the presentation of data to the application is shown in Figure 8.2.

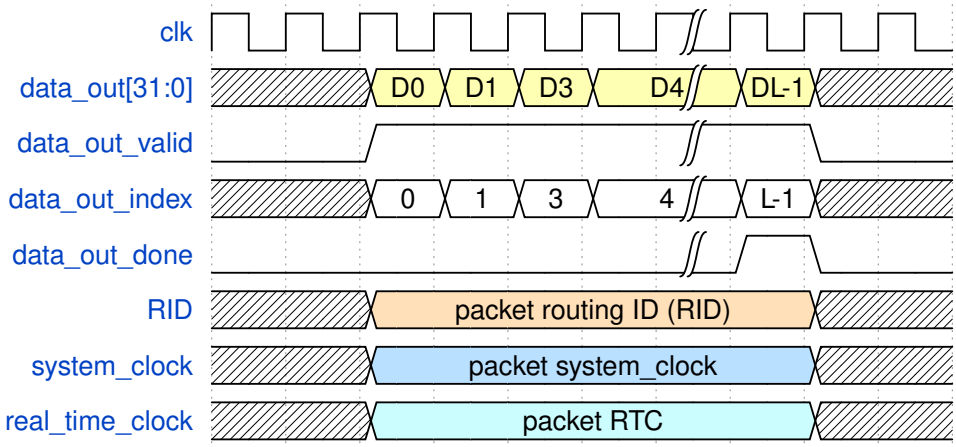

Fig. 8.2: Peripheral Application Interface Timing Diagram (to Application)

It is important to note that this receive path from controller to application does not wait for the application to signal its ready status before presenting data to the application. This was intentional as it allows the receive buffer to be minimized to free up specific FPGA resources for the application that is to to be ran on this device, namely the Large Static Random-Access Memory (LSRAM) blocks. If buffering is required upon receiving, this is to be handled by the application. The Peripheral Application Interface's buffer sizes were chosen to allow up to a 9,216-byte packet size or a 2,304 32-bit word packet size, as this is a commonly accepted maximum size for a jumbo Ethernet frame.

8.2 Transmit Path: From Application to Controller

When the application has data to be sent across the LACE network, whether it be out of the network via Ethernet/UART, or to another peripheral FPGA, it will do so via the

input interface shown in blue Figure 8.1. This interface takes in data as well as an RID from the application, wraps it in a valid LACE packet with current system clock and RTC values, and pushes the resulting LACE packet data stream onto the LiteFast transmit FIFO to be sent to the controller FPGA for routing. This interface can be seen as the input ports of the blue section of the Simulink model in Figure 8.1, as well as the `ready_for_data` port. This `ready_for_data` flag signals to the application when the Peripheral Application Interface is ready for more data, derived from the almost full signal from the LiteFast transmit FIFO. This pushes the requirement of buffering data to the application, but as discussed in Section 8.1, this allows for fewer FPGA resources to be used for the Peripheral Application Interface, freeing them up for the application to utilize. An example timing diagram showing a generic transaction of the presentation of data from the application to be sent to the controller FPGA for routing is shown in Figure 8.3.

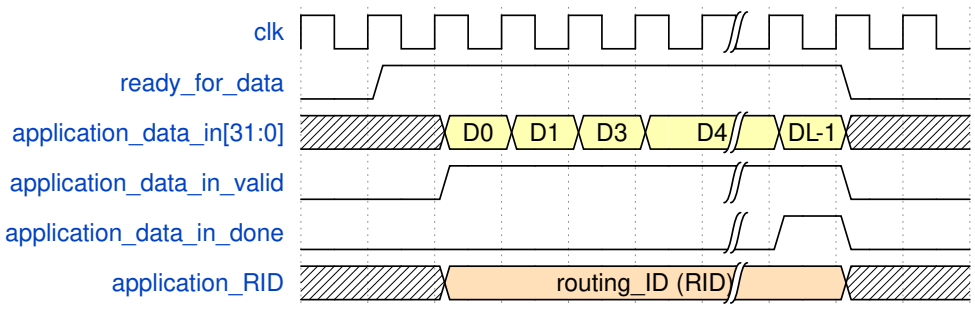

Fig. 8.3: Peripheral Application Interface Timing Diagram (from Application)

CHAPTER 9

Results

This chapter discusses the results of the LACE network architecture and firmware and how each research objective was accomplished.

9.1 Hardware Testbench Setup and Results

The LACE network firmware system was validated on a hardware testbench consisting of one controller FPGA board and one peripheral FPGA board, both implemented on Microchip PolarFire MPF300T development kits. The two boards were connected through a transceiver lane operating at 10 Gbps. A UART interface connected to a Raspberry Pi provided a low-rate command and telemetry connection, while a Gigabit Ethernet connection to a nearby desktop computer served as the primary high-throughput data path. Figure 9.1 shows this hardware connection on the testbench.

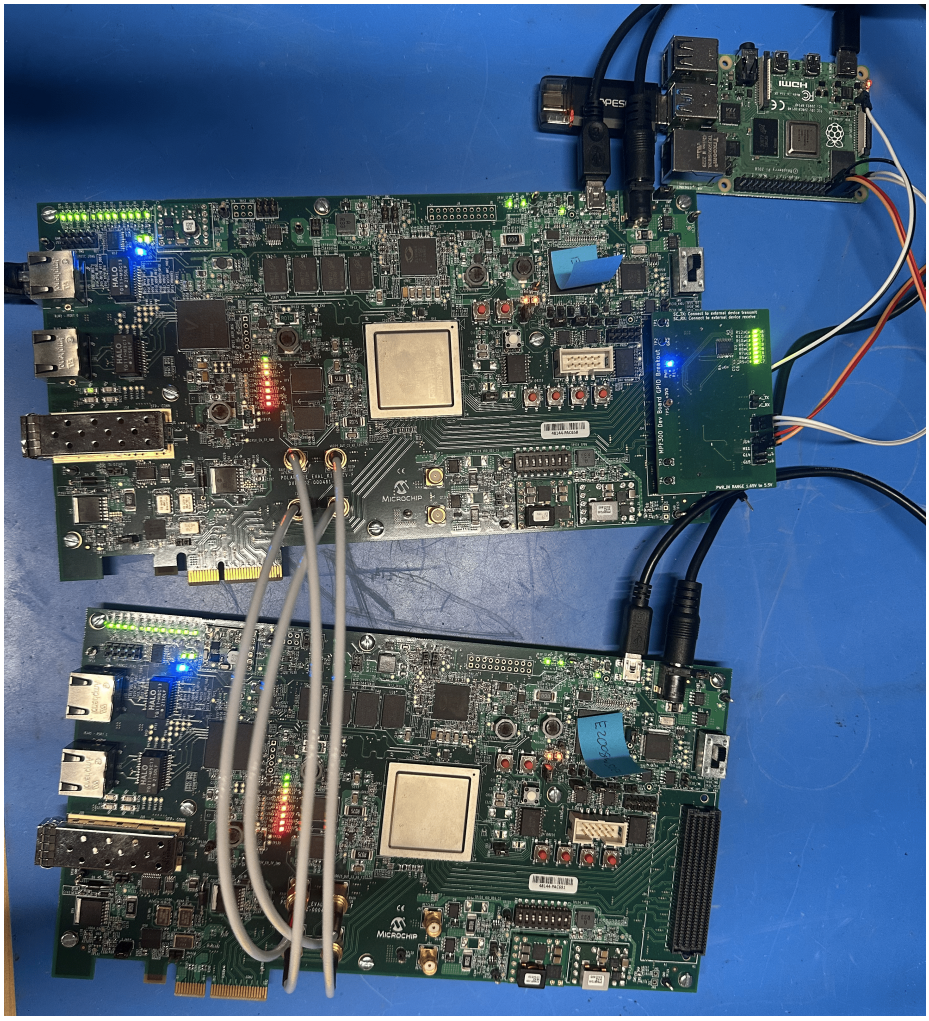

Fig. 9.1: LACE Network Hardware Bench Setup

The Raspberry Pi was running custom Python code emulating a spacecraft computer to parse and send CCSDS Space Packets. The desktop computer was running Cat Karat Packet Builder to produce Ethernet packets as desired and Wireshark to view the Ethernet data through the network interface on the desktop computer.

To validate the full system, the controller FPGA was programmed with the LACE network controller firmware including a UART, Ethernet, one peripheral, and a command and control interface connected to a central Packet Router, as seen in Figure 2.1. The peripheral FPGA was programmed with a Peripheral Application Interface connected to the transceiver lanes from the controller and responds with an echo of data with a different

length packet and a new RID to route back out Ethernet and UART. A sequence of Ethernet packets was sent to stimulate this system. Figure 9.2 shows the Wireshark capture of this sequence.

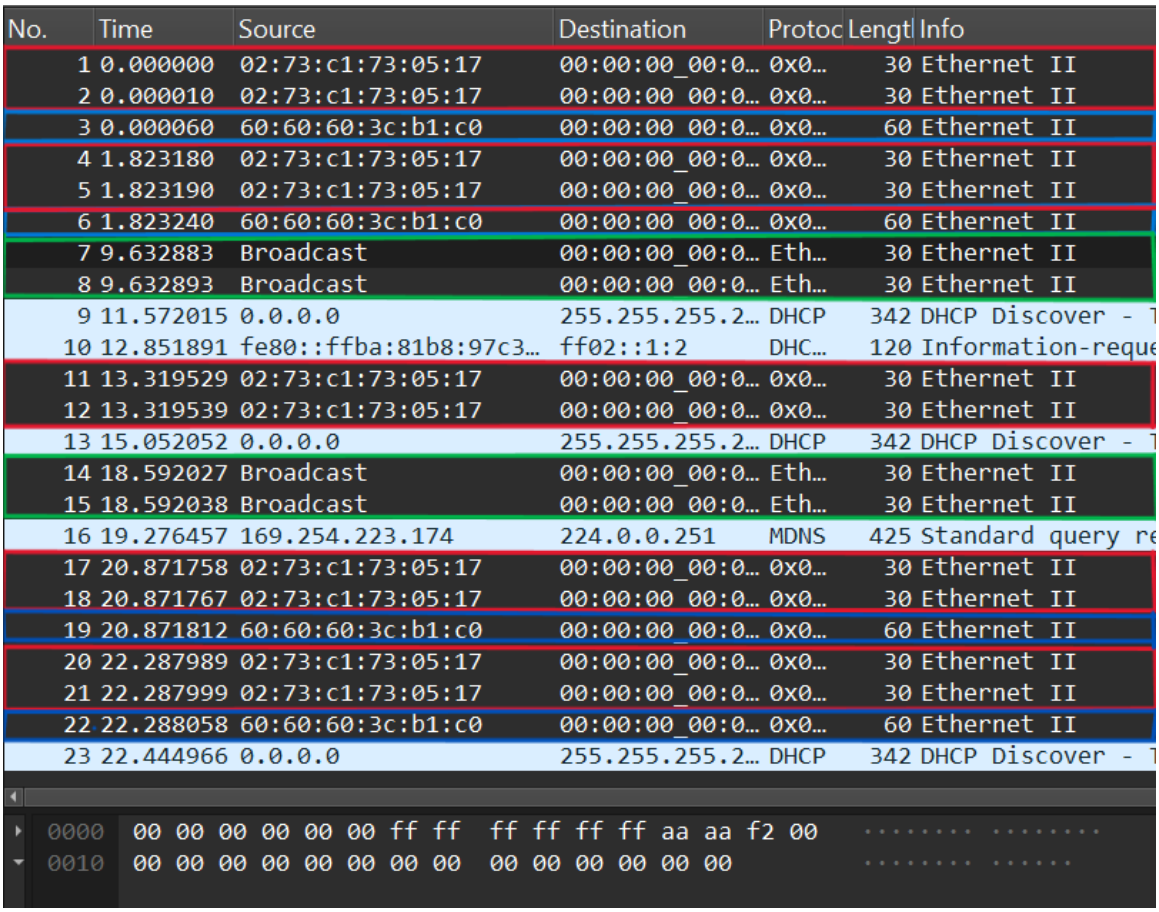

Fig. 9.2: Wireshark Results Showing Peripheral Echo and Configuration

First, to explain the sequence shown in Figure 9.2 further, two "normal data" packets were sent, tagged with a MAC address that will map to an RID (`0xf2`) that tells the packet router to send this data to the peripheral FPGA, seen boxed in red. Wireshark captures duplicates of sent packets from the host desktop computer, hence the block of two identical packets in each red box. After each "normal data" packet, the peripheral's response data routed back through the controller can be seen, boxed in blue. Next, a configuration packet was sent, tagged with the source MAC address of `0xffffffffffff` and boxed in green,

which tells the controller's packet router to send this to the command and control interface. This configures the routing table index `0xf2` to no longer route anywhere. To test this configuration, another "normal data" packet is sent, and no response is seen as the data didn't get forwarded to the peripheral FPGA. Next, another configuration packet was sent, which reconfigures the "normal data" packet to get forwarded to the peripheral FPGA again, seen again in green. Lastly, two more "normal data" packets were sent, seen in red. The corresponding responses from the peripheral FPGA are shown in blue. It is important to note that the packets that are shown with a light blue background are not a part of the LACE network system, and is higher level communication being attempted by the host desktop's network interface.

9.2 Resource Utilization

This section addresses the research objective to "develop firmware for peripheral FPGAs that receives sensor data from the controller's Ethernet port, using fewer FPGA resources than direct Ethernet implementation on the peripheral FPGA" and discusses the resource utilization of both FPGAs and their subsystems. To perform the analysis of resource utilization on the development FPGAs, the Libero SoC's `top_compile_netlist_hier_resources.csv` and `top_compile_netlist_resources.xml` reports were examined after synthesis. The resource utilization was performed with buffers across the system to handle data frame sizes up to 9,216 bytes (2,304 32-bit words) as this is a commonly accepted maximum size for a jumbo Ethernet frame.

9.2.1 Controller FPGA

For the controller FPGA, Table 9.1 shows the total resource utilization of each relevant type of FPGA resource. The resource totals are derived from the Libero SoC reported resources of a PolarFire MPF300T. To define the resource type acronyms, 4LUT means 4-input Look-up Table, DFF means D Flip Flop, μSRAM means Micro Static Random-Access Memory, and LSRAM means Large Static Random-Access Memory.

Table 9.1: Controller FPGA Resource Utilization.

Resource Type	Used	Total Available	Percent Utilized
4LUT	45,801	299,544	15.29%
DFF	29,162	299,544	9.74%
μSRAM	65	2,772	2.34%
LSRAM	217	952	22.79%
Math	0	924	0%
Transceiver Lanes	1	16	6.25%

9.2.2 Peripheral FPGA

For the peripheral FPGA, Table 9.2 shows the total resource utilization of each relevant type of FPGA resource. The resource totals are derived from the Libero SoC reported resources of a PolarFire MPF300T.

Table 9.2: Peripheral FPGA Resource Utilization.

Resource Type	Used	Total Available	Percent Utilized
4LUT	3,778	299,544	1.26%
DFF	2,795	299,544	0.93%
μSRAM	17	2,772	0.61%
LSRAM	22	952	2.31%
Math	0	924	0%
Transceiver Lanes	1	16	6.25%

To compare whether the LACE network system as designed used fewer resources, Table 9.3 shows an estimate for resource utilization for Ethernet alone as implemented on the controller FPGA. This is pulled from the Libero SoC `top_compile_netlist_hier_resources.csv` for the controller FPGA's `ethernet_top` design.

Table 9.3: Ethernet FPGA Resource Utilization Estimate.

Resource Type	Used	Total Available	Percent Utilized
4LUT	17,027	299,544	5.68%
DFF	6,945	299,544	2.32%
μSRAM	11	2,772	0.40%
LSRAM	30	952	3.15%
Math	0	924	0%

Table 9.4 shows an estimated resource utilization comparison if Ethernet was instantiated on the peripheral FPGAs themselves using the data from Table 9.3. The results shown indicate that the research objective was met to reduce the resource utilization on the peripheral FPGAs.

Table 9.4: Peripheral FPGA Resource Utilization Comparison.

Resource Type	Used (Transceivers)	Used (with Ethernet)	Difference	Approx. Percent Saved
4LUT	3,778/299,544 = 1.26%	18,439/299,544 = 6.16%	14,661	4.90%
DFF	2,795/299,544 = 0.93%	7,741/299,544 = 2.58%	4,946	1.65%
μSRAM	17/2772 = 0.61%	20/2772 = 0.72%	3	0.11%
LSRAM	22/952 = 2.31%	52/952 = 5.46%	30	3.15%
Math	0	0	0	0%

9.3 Timestamping

This section addresses the research objective to "timestamp all sensor and telemetry data with 1 millisecond precision". As seen in Figure 3.3, data is tagged with timestamp information that includes the FPGA system clock in milliseconds and the FPGA real-time clock (RTC) in minutes. This information is tagged on incoming data when received on the controller FPGA and is tagged on the outgoing data when received on the peripheral FPGA(s). This allows the applications on the peripheral FPGAs the use the timing information presented as needed for their algorithms. The real-time clock is currently being emulated by a counter internal to the FPGA, but the LACE-C3A hardware is planned to have an on-board RTC counter chip to maintain a monotonically increasing minute counter that would replace the current RTC data.

9.4 Interfaces

This section addresses the research objective to "establish a command and telemetry interface to an external spacecraft computer via UART". A CCSDS Space Packet-based UART Handler was developed to integrate with the Packet Router as described further

in Chapter 6. Additionally, a Command and Control Handler was developed to receive command data and reconfigure the controller FPGA at runtime accordingly and to provide a structure to provide controller FPGA telemetry. This handler and its integration with the system is further described in Chapter 7. With the CCSDS Space Packet-based UART interface and the Command and Control Handler both interfaced to the controller FPGA's Packet Router, this provides a functional command and telemetry interface to an external spacecraft computer via UART. The system was augmented to allow command and control communication to be performed over the Ethernet port in the same fashion as over UART, allowing the UART port to be omitted if required by spacecraft constraints.

CHAPTER 10

Conclusion

The LACE network firmware architecture was successfully created using HDL code generated via Simulink and HDL Coder, custom HDL, and other IP blocks, connected together using Microchip's Libero SoC SmartDesigns. Firmware was created for the controller FPGA to handle incoming and outgoing data via Gigabit Ethernet and UART ports, command and control configuration, and data routing to one or more peripheral FPGAs connected via PolarFire FPGAs' transceiver lanes. Firmware was created for the peripheral FPGAs to connect to the controller FPGA via the transceiver lanes and provide a simplified interface to a user application to run on the peripheral FPGA for less FPGA resources than a full Ethernet implementation. All data within the system was timestamped with millisecond precision.

This architecture was successfully demonstrated on a hardware testbench using PolarFire MPF300T development kits programmed using Libero SoC. The demonstration included one controller FPGA, one peripheral FPGA, a host desktop computer producing and consuming Ethernet frames acting as a sensor data source, and a Raspberry Pi acting as a spacecraft computer connected via UART.

10.1 Future Work

Future work for this thesis would include adding the ability for the controller board to reprogram the peripheral FPGAs, adding memory drivers to the peripheral application interface, and improving timestamping to handle Global Positioning System (GPS) and Pulse-Per-Second (PPS) timing from a spacecraft computer.

As this system is intended to improve the accessibility and flexibility of FPGA edge computing, being able to reprogram the peripheral FPGAs by sending new FPGA bitstream to the controller FPGA would support this goal. Algorithm designers could make

iterative changes to their processes while the spacecraft flies and reduce the time to test new algorithms.

Many machine learning algorithms require large amount of memory, often more than is available with the FPGA internal block RAMs. Because of this, FPGAs used as the peripherals will typically be paired with some large memory such as Low-Power Double Data Rate (LPDDR) RAM. To further streamline the algorithm design process, an interface could be created to handle the communication with the LPDDR RAM and provide a simplified interface to the user application.

The current architecture timestamps all internal data when it is received (controller) or presented from the application (peripheral). However, the timestamping is currently based on internal counters that reset on FPGA power down. Additionally, these internal counters can drift relative to absolute GPS time. To resolve this, GPS data packet receiving, paired with a PPS signal, could be developed. This would allow for absolute time stamping of the data received and transmitted by the LACE network, thus synchronizing the entire LACE network system with spacecraft and ground systems.

REFERENCES

[1] Microchip Technology Inc., *PolarFire FPGA 1G Ethernet Loopback Using IOD CDR*, Microchip Technology Inc., 2023, accessed: Oct. 23, 2025. [Online]. Available: https://ww1.microchip.com/downloads/aemDocuments/documents/FPGA/ApplicationNotes/ApplicationNotes/PolarFire_FPGA_1G_Ethernet_Loopback_Using_IOD_CDR.pdf

[2] C. Secretariat, *Space Packet Protocol, Recommended Standard, Issue 2*. Washington DC, USA: CCSDS Secretariat, 2020.

[3] IEEE 802.3 Ethernet Working Group, "Ieee 802.3 ethernet: A brief history," IEEE Standards Association, Tech. Rep., 2009, accessed: Oct. 7, 2025. [Online]. Available: https://www.ieee802.org/misc-docs/GlobeCom2009/IEEE_802d3_Law.pdf

[4] Microsemi, *UG0701 User Guide LiteFast IP*, Microchip Technology Inc., 2020, accessed: Oct. 23, 2025. [Online]. Available: https://ww1.microchip.com/downloads/aemDocuments/documents/FPGA/ProductDocuments/SoC/microsemi_smartfusion2_igloo2_rtg4_polarfire_litefast_ip_user_guide_ug0701_v5.pdf

[5] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, "State-of-the-art in heterogeneous computing," *Scientific Programming*, 2010.

[6] A. Utter, M. Zakrzewski, A. Keene, S. Dietrich, S. Lin, E. McDonald, N. Whitehair, and J. Zheng, "Satcat5: A low-power, mixed-media ethernet network for smallsats," 2020.

[7] N. L. Wallace, "Developing firmware for space weather probes 2 using hdl coder," Master's thesis, Utah State University, Logan, UT, 2023.

[8] Microchip Technology Inc., *CoreTSE User Guide*, Microchip Technology Inc., 2022, accessed: Oct. 23, 2025. [Online]. Available: https://ww1.microchip.com/downloads/aemDocuments/documents/FPGA/ProductDocuments/UserGuides/ip_cores/directcores/CoreTSE_HB.pdf

[9] "Ieee draft standard for vhdl language reference manual," *IEEE P1076/D13, July 2019*, pp. 1–796, 2019.

[10] D. Pellerin. (1995) An introduction to vhdl. [Online]. Available: https://www.uco.es/~ff1mumuj/h_intro.htm

[11] J. C. T. Hai, O. C. Pun, and T. W. Haw, "Accelerating video and image processing design for fpga using hdl coder and simulink," in *2015 IEEE Conference on Sustainable Utilization And Development In Engineering and Technology (CSUDET)*, 2015, pp. 1–5.

[12] A. Geist and et al., "Spacecube v3.0: Nasa next-generation high-performance onboard computing," in *AIAA/USU Small Satellite Conference*, 2019. [Online]. Available: https://digitalcommons.usu.edu/smallsat/2019/all2019/259/

[13] A. D. George and C. M. Wilson, "Onboard processing with hybrid and reconfigurable computing on small satellites," *Proceedings of the IEEE*, vol. 106, no. 3, pp. 459–484, 2018.

[14] D. D. Langer and et al., "Robust and reconfigurable onboard processing for a hyperspectral cubesat: Hypso-1," *Remote Sensing*, vol. 15, no. 15, p. 3756, 2023.

[15] G. Meoni and et al., "The ops-sat case: A data-centric competition for onboard satellite image classification," *CEAS Space Journal*, 2024.

[16] S. S. S. V. I. (S3VI), "State-of-the-art small spacecraft technology report, 2024 edition," NASA Ames Research Center, Tech. Rep., 2025, chapter on Avionics and Processing summarizes current onboard computing technologies. [Online]. Available: https://www.nasa.gov/smallsat-institute/sst-soa/

[17] A. Geist and et al., "Spacecube edge-node intelligent collaboration (scenic): A framework for inflight ai/ml processing," in *AIAA/USU Small Satellite Conference*, 2023.

[18] C. M. Moreira and et al., "Edge computing in space: Design of an fpga architecture for thermal anomaly detection," *Advances in Space Research*, 2025.

[19] J. L. Carr and et al., "Stereobit: Onboard science data processing for stereo imagery using fpgas," *Journal of Aerospace Information Systems*, 2025.

[20] M. T. Inc., "Polarfire® family transceiver user guide (ds00004164j)," Microchip Technology Inc., Tech. Rep., 2025, specification of SERDES lanes up to 12.7 Gb/s for PolarFire FPGAs.

[21] S. Parkes and colleagues, "Spacefibre: A multi-gigabit/s interconnect for spacecraft on-board data handling," in *IEEE Aerospace Conference*, 2015. [Online]. Available: https://www.star-dundee.com/wp-content/.../Paper-IEEE-Aerospace-2015-Parkes-SpaceFibre.pdf

[22] ——, "Spacefibre network and routing switch," in *IEEE Aerospace Conference*, 2017. [Online]. Available: https://discovery.dundee.ac.uk/files/12541849/Paper_IEEE_Aerospace_2017_Parkes_SpaceFibre_Networks_v2.pdf

[23] ——, "Spacewire and spacefibre on the microsemi rtg4 fpga," in *IEEE Aerospace Conference*, 2016. [Online]. Available: https://www.star-dundee.com/.../Paper_IEEE_Aerospace_2016_Parkes_SpaceWire_SpaceFibre_RTG4.pdf

[24] J. Zheng and colleagues, "An efficient multi-lane spacefibre core for spacecraft onboard networks," *Electronics*, vol. 11, no. 9, p. 1410, 2022. [Online]. Available: https://www.mdpi.com/2079-9292/11/9/1410

[25] M. Khalid *et al.*, "A novel and efficient routing architecture for multi-fpga systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. XX, no. YY, p. ZZ–ZZ, 2014, study of inter-chip routing and partitioning for multi-FPGA architectures.

[26] R. Nepomuceno, R. Sterle, G. Valarini, M. Pereira, H. Yviquel, and G. Araujo, "Enabling openmp task parallelism on multi-fpgas," in *Proceedings of the IEEE International Conference on Reconfigurable Computing and FPGA Acceleration*, 2021, demonstrates multi-FPGA system scaling and interconnect challenges.

[27] M. Birmingham and colleagues, "Essential spacewire hardware capabilities for a robust spacewire network," NASA Goddard Space Flight Center, Tech. Rep., 2016. [Online]. Available: https://ntrs.nasa.gov/api/citations/20160012696/downloads/20160012696.pdf

[28] M. T. Inc., "Polarfire fpga and polarfire soc fpga programming user guide (ds50003191h)," Microchip Technology Inc., Tech. Rep., 2025, describes Auto-Update and In-Application Programming modes for in-flight reprogramming.

[29] Texas Instruments, *LVDS Owner's Manual, 4th Edition*, Texas Instruments, 2008, accessed: Oct. 15, 2025. [Online]. Available: https://www.ti.com/lit/ug/snla187/snla187.pdf

APPENDICES

APPENDIX A

Libero SoC SmartDesigns

A.1 top SmartDesign

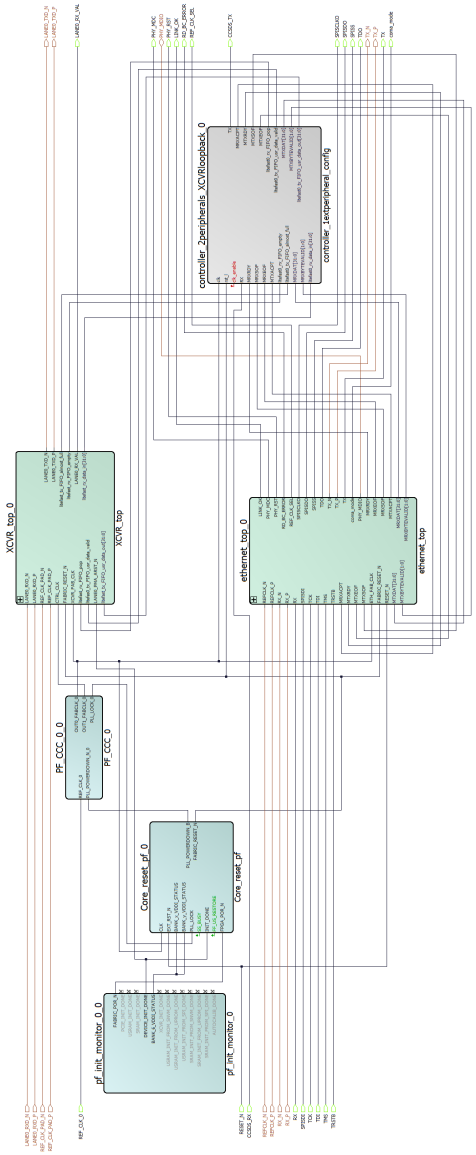

Fig. A.1: top SmartDesign

A.2 LiteFast_Receiver SmartDesign

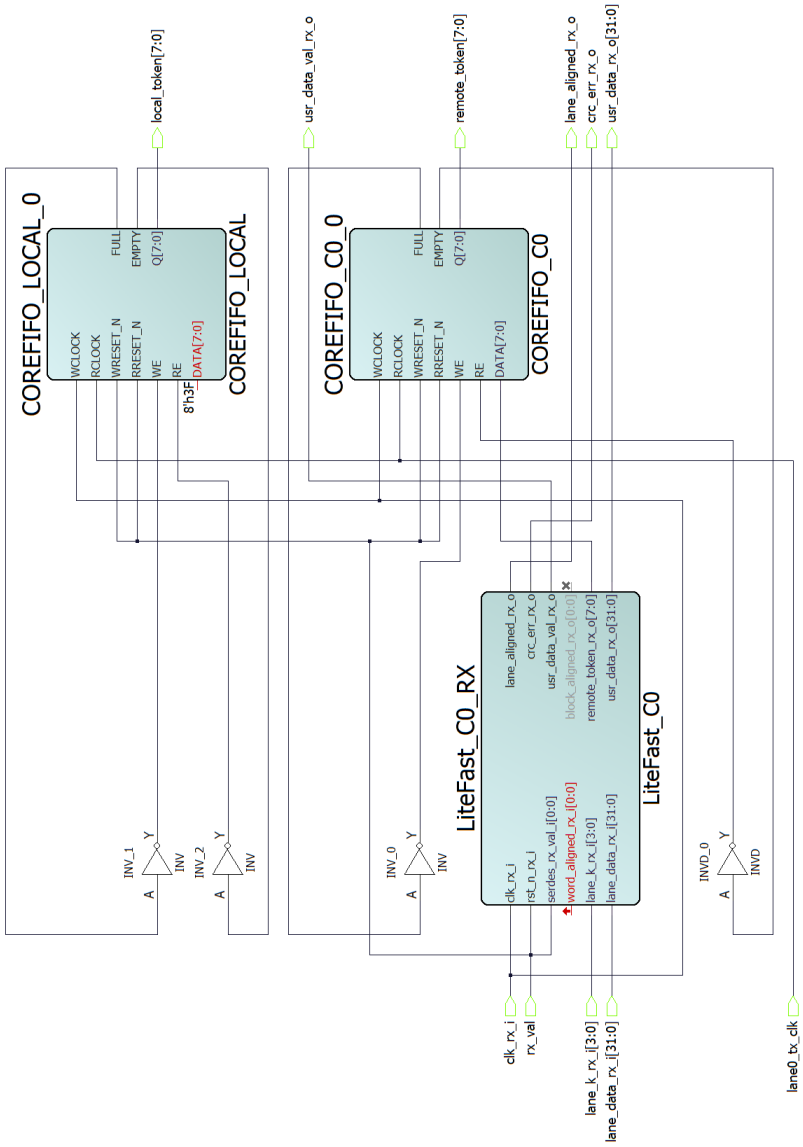

Fig. A.2: LiteFast_Receiver SmartDesign

A.3 Transceiver SmartDesign

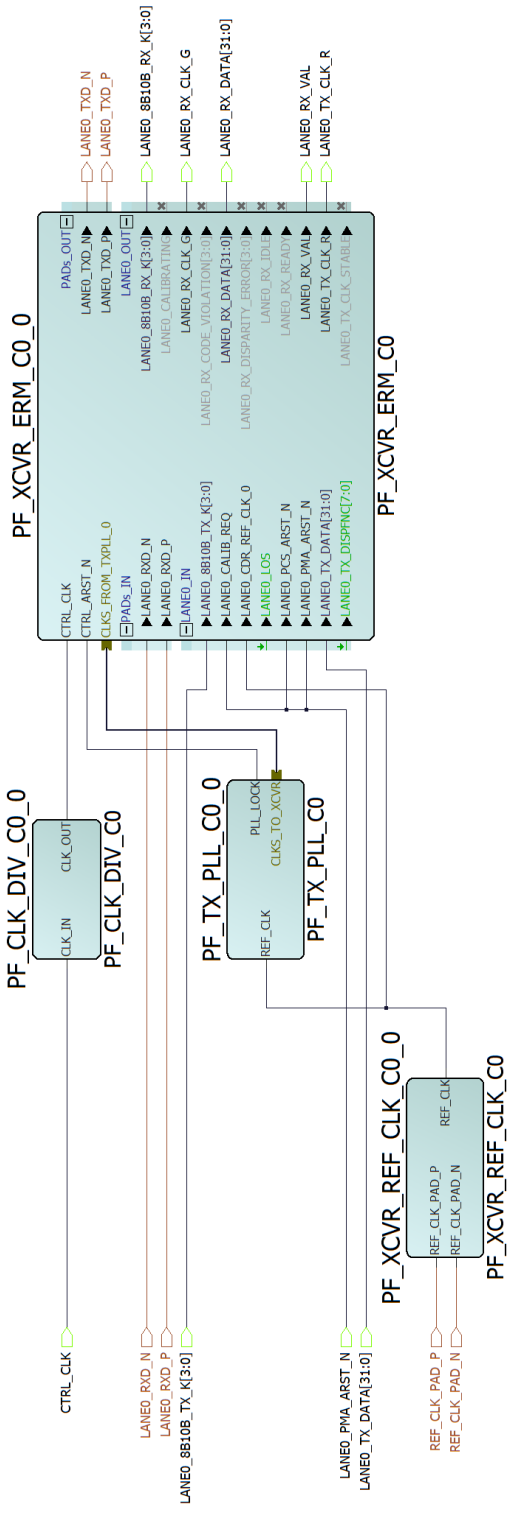

Fig. A.3: Transceiver SmartDesign

APPENDIX B

Libero SoC IP Configuration

B.1 top SmartDesign IP Configuration

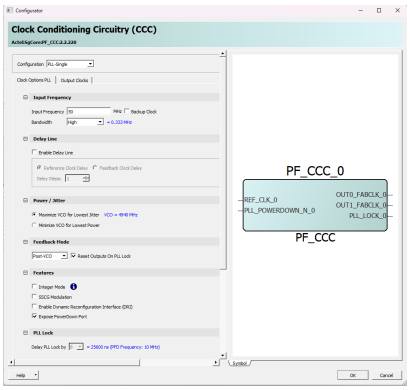

Fig. B.1: PF_CCC_0_0 Clock Options PLL Configuration

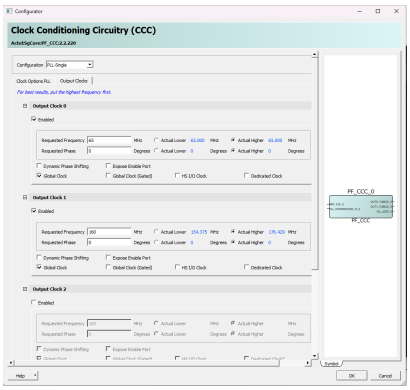

Fig. B.2: PF_CCC_0_0 Output Clocks Configuration

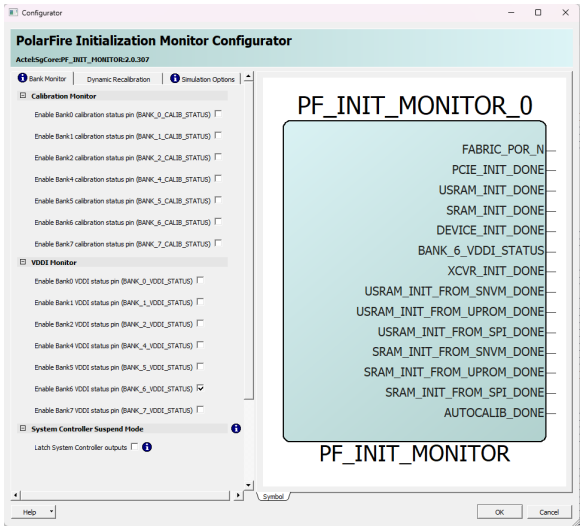

Fig. B.3: pf_init_monitor_0_0 Configuration

B.2 XCVR_top SmartDesign IP Configuration

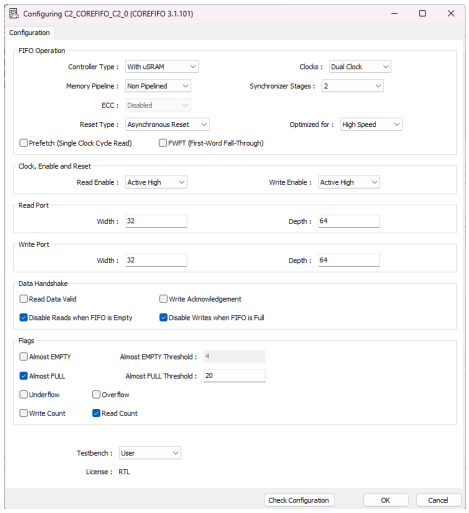

Fig. B.4: COREFIFO_C2_0 Configuration

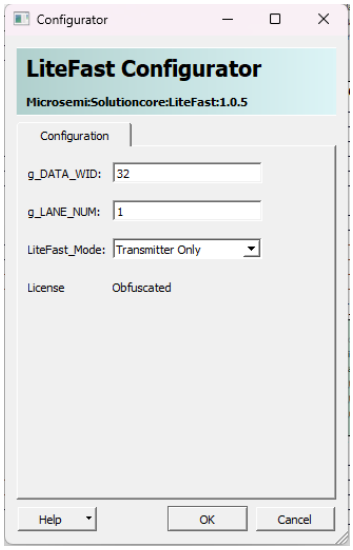

Fig. B.5: LiteFast_C1_0 Configuration

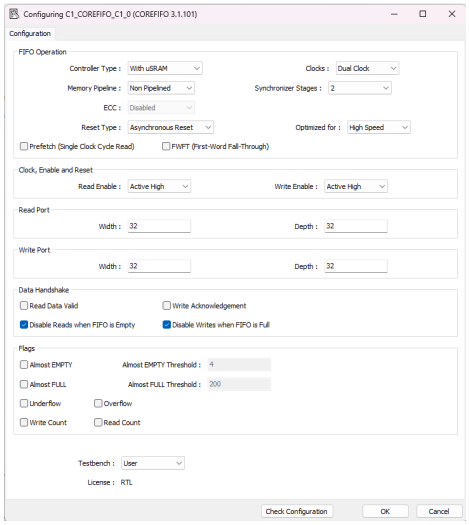

Fig. B.6: COREFIFO_C1_0 Configuration

B.3 ethernet_top SmartDesign IP Configuration

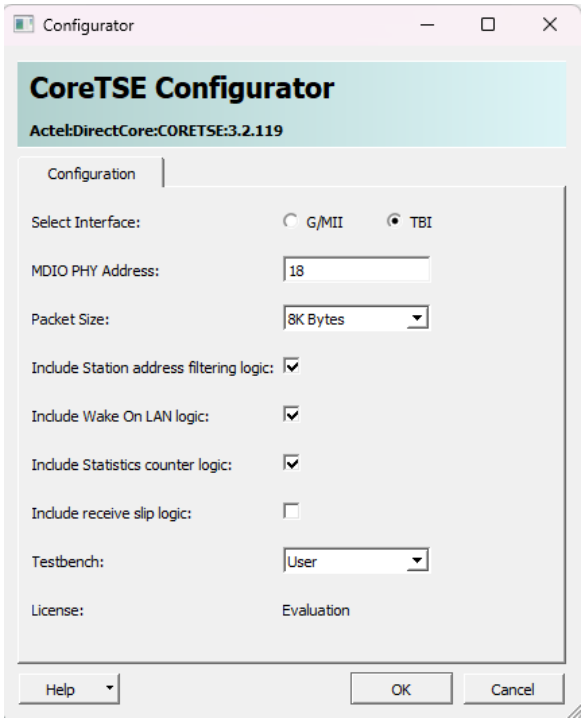

Fig. B.7: CORETSE_0 Configuration

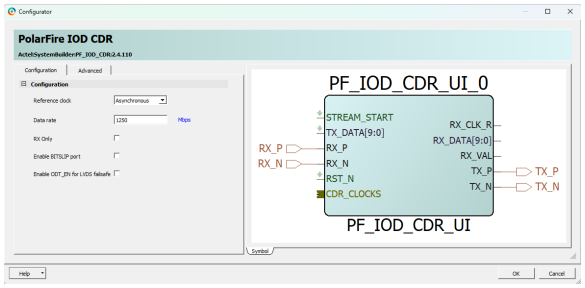

Fig. B.8: PF_IOD_CDR_C0_0 Configuration

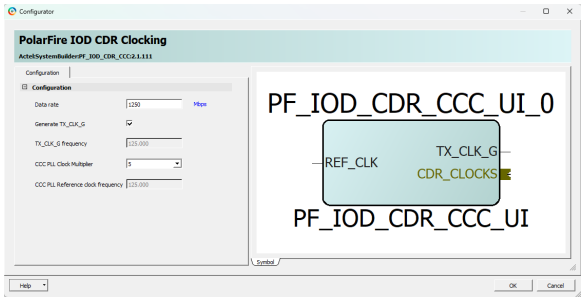

Fig. B.9: PF_IOD_CDR_CCC_C0_0 Configuration

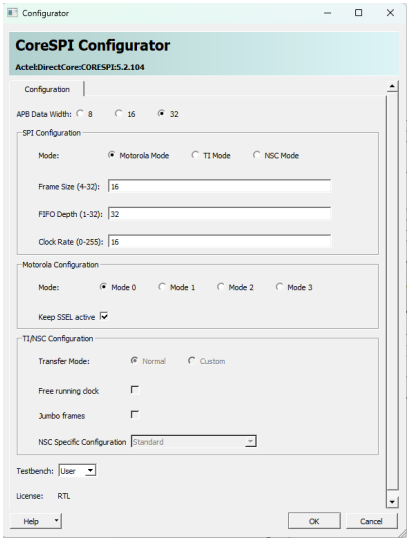

Fig. B.10: CORESPI_0_0 Configuration

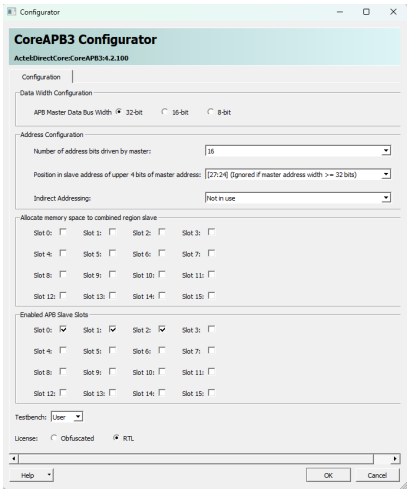

Fig. B.11: CoreAPB_3_0_0 Configuration

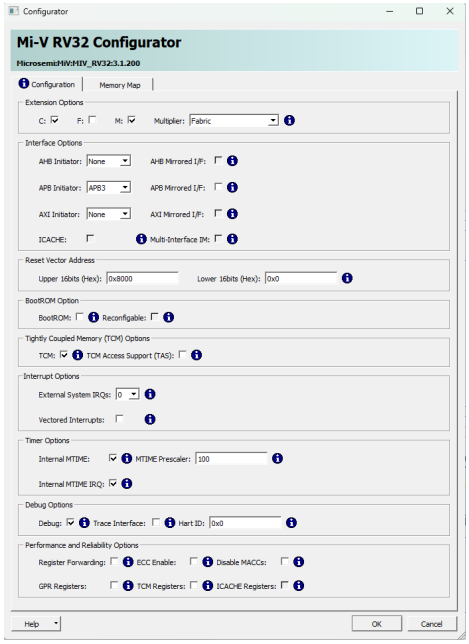

Fig. B.12: MIV_RV32_C0_0 Configuration Tab Configuration

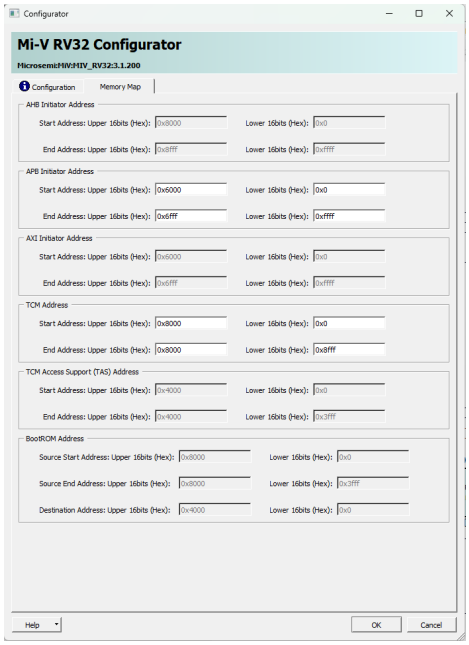

Fig. B.13: MIV_RV32_C0_0 Memory Map Tab Configuration

Fig. B.14: COREJTAGDEBUG_C0_0 Configuration

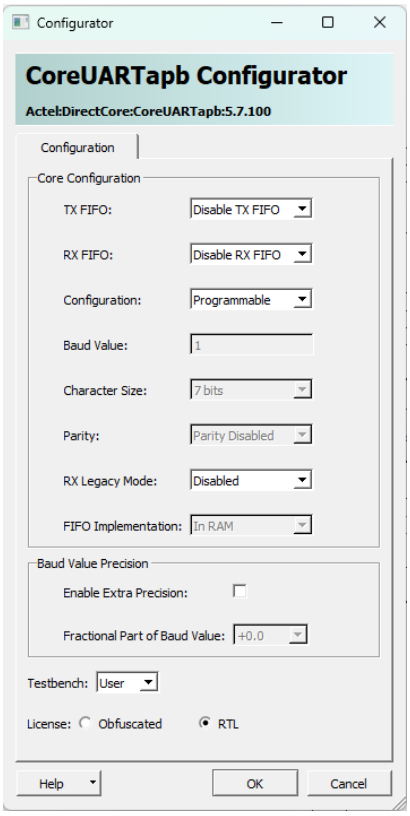

Fig. B.15: CoreUARTapb_0 Configuration

B.4 LiteFast_Receiver SmartDesign IP Configuration

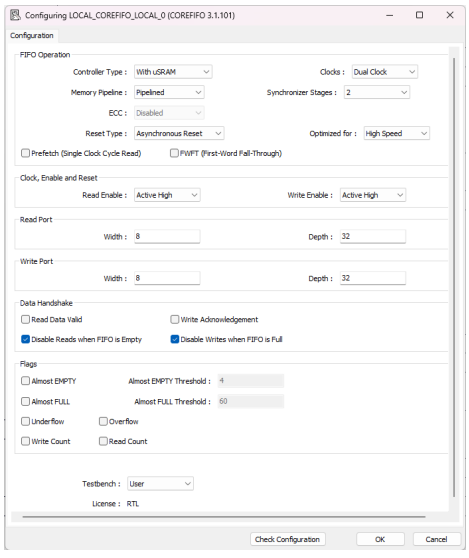

Fig. B.16: COREFIFO_LOCAL_0 Configuration

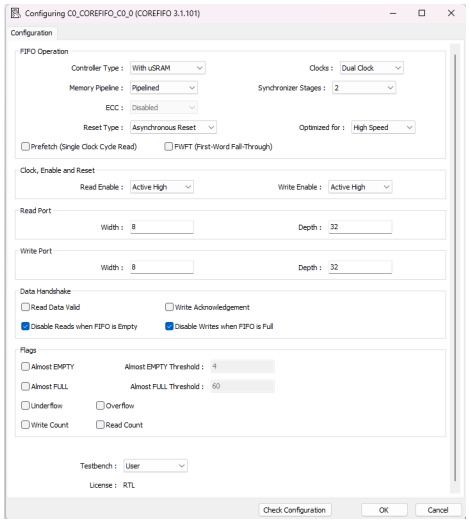

Fig. B.17: COREFIFO_C0_0 Configuration

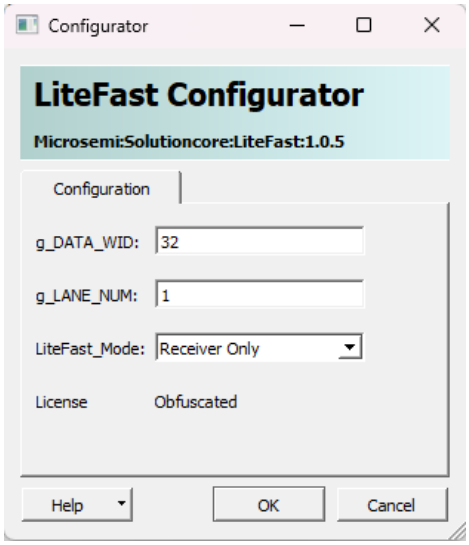

Fig. B.18: LiteFast_C0_RX Configuration

B.5 Transceiver SmartDesign IP Configuration

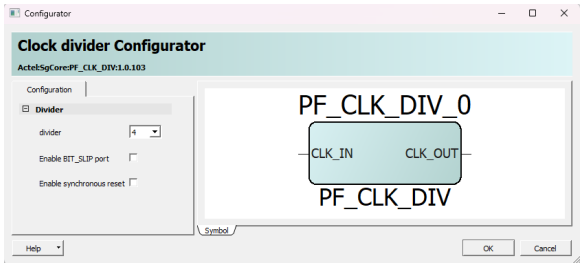

Fig. B.19: PF_CLK_DIV_0 Configuration

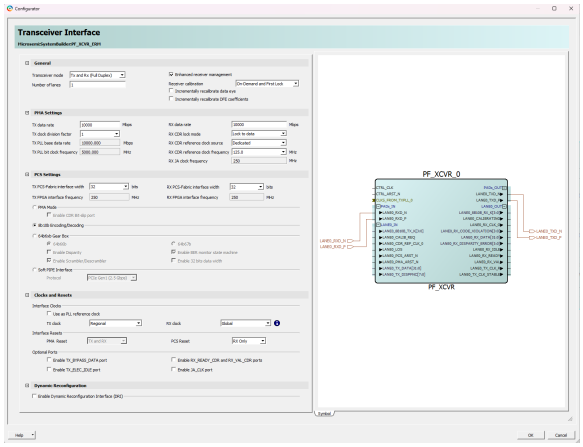

Fig. B.20: PF_XCVR_ERM_C0_0 Configuration

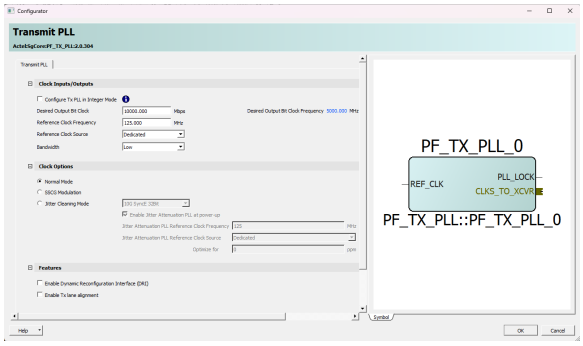

Fig. B.21: PF_TX_PLL_0 Configuration

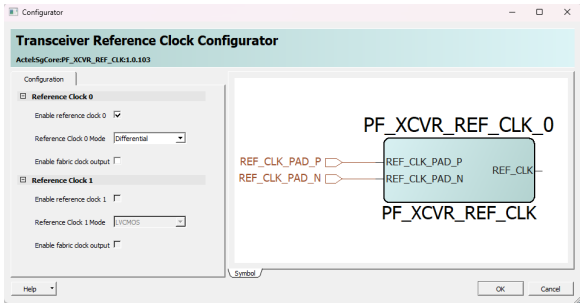

Fig. B.22: PF_XCVR_REF_CLK_C0_0 Configuration

APPENDIX C

Simulink Models

C.1 Packet Router Simulink Model

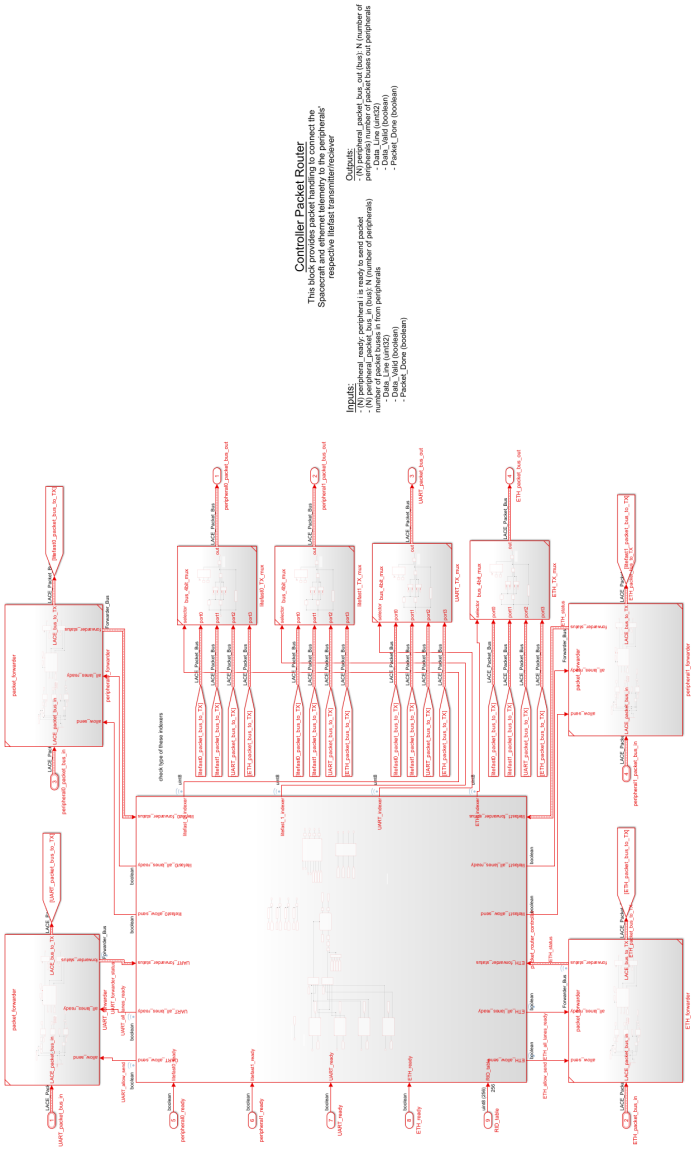

Fig. C.1: Packet Router Simulink Model

APPENDIX D

Source Code

D.1 SSDetect.v from PolarFire FPGA 1G Ethernet Loopback Using IOD CDR Guide [1]

```
‘timescale 1ns / 1ps

module SSDetect( rst_b, rck, rx_data, stream_start );
input         rst_b;
input         rck;
input [9:0] rx_data; // from RX_P
output stream_start;

function is_match (input [6:0] x, input [6:0] y);
begin
is_match = (x == y) | (x == ~y);
end
endfunction

reg [1:0] rx_start;

assign stream_start = rx_start[0]; // CDR starts after RX data detects two
    consecutive non-static words

always @(posedge rck or negedge rst_b) begin // SAR 101393, use negedge clock
if (!rst_b) begin
rx_start <= 2’d0;
end
else if (!rx_start[0]) begin // two consecutive non-static words
```

```
rx_start <= is_match(rx_data[6:0], 0) ? 2'd0 : {1'b1, rx_start[1]};
end
end


endmodule
```

D.2 main.c edited from PolarFire FPGA 1G Ethernet Loopback Using IOD CDR Guide [1]

```
/*******************************************************************************
* (c)Copyright 2016-2017 Microsemi Corporation. All rights reserved.
*  Simple IOD CDR 1G loop back example program .
*/
#include "drivers/CoreUARTapb/core_uart_apb.h"
#include "miv_rv32_hal/miv_rv32_hal.h"
#include "sample_hw_platform.h"

extern void configure_zl30364(void);

uint8_t testmsg[] = {"\r\n\r\nHello World!\0"};
/*
* CoreUART instance data.
*/
UART_instance_t g_uart;
extern void delay(uint32_t div);
void Phy_advertise(void)
{
  uint32_t phy_reg = 0xFFFF;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C04;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x1;
  while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);
  phy_reg = *(volatile unsigned int *) (TSE_BASEADDR + 0x030);
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x0;
  phy_reg &= ~(0x1E);
  *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C04;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x02C) = phy_reg;
  while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);
  *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C09;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x1;
  while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);
  phy_reg = *(volatile unsigned int *) (TSE_BASEADDR + 0x030);
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x0;
  phy_reg |= 0x200;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C09;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x02C) = phy_reg;
  while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);
}
```

```
void phy_autonegotiation(void)
{
  uint32_t phy_reg = 0xFFFF;
  uint16_t autoneg_complete;
  volatile uint32_t copper_aneg_timeout = 1000000u;
  volatile uint32_t sgmii_aneg_timeout = 100000u;
  uint8_t copper_link_up;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C1F;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x02C) = 0x0;
  while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);
  *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C00;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x1;
  while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);
  phy_reg = *(volatile unsigned int *) (TSE_BASEADDR + 0x030);
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x0;
  phy_reg |= 0x1200;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C00;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x02C) = phy_reg;
  while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);

  do {
    *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C01;
    *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x1;
    while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);

    phy_reg = *(volatile unsigned int *) (TSE_BASEADDR + 0x030);
    *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x0;

    autoneg_complete = phy_reg & 0x0020u;
    --copper_aneg_timeout;
  } while(!autoneg_complete && (copper_aneg_timeout != 0u));

  for (volatile uint32_t i = 0; i < 100000; i++);
  *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C01;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x1;
  while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);
  phy_reg = *(volatile unsigned int *) (TSE_BASEADDR + 0x030);
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x0;

  copper_link_up = phy_reg & 0x0004;
  if(copper_link_up != 0u)
  {
    *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1200;
    *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x1;
    while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);
    phy_reg = *(volatile unsigned int *) (TSE_BASEADDR + 0x030);
    *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x0;
    phy_reg |= 0x1000;
    *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1200;
    *(volatile unsigned int *) (TSE_BASEADDR + 0x02C) = phy_reg;
    while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);
    *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1200;
    *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x1;
    while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);
```

```
    phy_reg = *(volatile unsigned int *) (TSE_BASEADDR + 0x030);
    *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x0;

    phy_reg |= 0x0200;

    *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1200;
    *(volatile unsigned int *) (TSE_BASEADDR + 0x02C) = phy_reg;
    while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);

    do {
      *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1201;
      *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x1;
      while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);

      phy_reg = *(volatile unsigned int *) (TSE_BASEADDR + 0x030);
      *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x0;
      autoneg_complete = phy_reg & 0x0020;
      --sgmii_aneg_timeout;
    } while((!autoneg_complete) && (sgmii_aneg_timeout != 0u));


  }
}

void phy_init (void)
{
  volatile uint16_t phy_reg_0;
  volatile uint16_t temp;
  volatile uint16_t id1=0,id2=0,phy_mac_reg = 0;

  *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C02;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x1;
  while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);

  id1 = *(volatile unsigned int *) (TSE_BASEADDR + 0x030);
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x0;

  *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C03;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x1;
  while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);

  id2 = *(volatile unsigned int *) (TSE_BASEADDR + 0x030);
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x0;

  /* 16E3 bit 7 setting to 1 for SERDES MAC AN EN */
  *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C1F;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x02C) = 0x0003;
  while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);

  *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C10;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x1;
  while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);

  phy_mac_reg = *(volatile unsigned int *) (TSE_BASEADDR + 0x030);
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x0;
```

```
phy_mac_reg |= 0x80;

  *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C10;
*(volatile unsigned int *) (TSE_BASEADDR + 0x02C) = phy_mac_reg;
while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);


/* Set Register 31 to 0 */
*(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C1F;
*(volatile unsigned int *) (TSE_BASEADDR + 0x02C) = 0x0010;
while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);
phy_mac_reg = 0x80F0;
*(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C12;
*(volatile unsigned int *) (TSE_BASEADDR + 0x02C) = phy_mac_reg;
while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);

while(1)
{
  *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C12;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x1;
  while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);

  temp = *(volatile unsigned int *) (TSE_BASEADDR + 0x030);
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x0;

  if((temp & 0x8000) == 0)
  {
    break;
  }
}

*(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C1F;
*(volatile unsigned int *) (TSE_BASEADDR + 0x02C) = 0x0;
while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);

*(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C00;
*(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x1;
while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);

phy_reg_0 = *(volatile unsigned int *) (TSE_BASEADDR + 0x030);
*(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x0;

phy_reg_0 = phy_reg_0 | 0x8000;
*(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C00;
*(volatile unsigned int *) (TSE_BASEADDR + 0x02C) = phy_reg_0;
while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);

while(1)
{
  *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C00;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x1;
  while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);

  temp = *(volatile unsigned int *) (TSE_BASEADDR + 0x030);
```

```
    *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x0;
    if((temp & 0x8000) == 0)
    {
      break;
    }
  }

  *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C00;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x1;
  while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);

  temp = *(volatile unsigned int *) (TSE_BASEADDR + 0x030);
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x0;

#if 0 // phy loopback farend

  *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C17;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x1;
  while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);

  phy_reg_0 = *(volatile unsigned int *) (TSE_BASEADDR + 0x030);
  *(volatile unsigned int *) (TSE_BASEADDR + 0x024) = 0x0;

  phy_reg_0 = phy_reg_0 | 0x8;

  *(volatile unsigned int *) (TSE_BASEADDR + 0x028) = 0x1C17;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x02C) = phy_reg_0;
  while ((*(volatile unsigned int *) (TSE_BASEADDR + 0x034)) != 0);

#endif
}

// TSE Initialisation

void tse_init (void)
{
  uint32_t tse_reg = 0xFFFF, phy_reg;

  //TSE Register settings
  *(volatile unsigned int *) (TSE_BASEADDR + 0x000) = 0x00000005;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x004) = 0x00007201;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x040) = 0x6060603C;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x044) = 0xB1C00000;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x048) = 0x0000FF00;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x04C) = 0x0FFF0000;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x050) = 0x04000180;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x054) = 0x0680FFFF;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x058) = 0x00000000;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x05C) = 0x0007FFFF;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x1C0) = 0x00000004;
  *(volatile unsigned int *) (TSE_BASEADDR + 0x020) = 0x0007;

  tse_reg = *(volatile unsigned int *) (TSE_BASEADDR + 0x20);

  phy_init();
```

```
}


/*-------------------------------------------------------------------------*//**
* main() function.
*/
int main() {
  configure_zl30364();
  tse_init();
  Phy_advertise();
  phy_autonegotiation();
  UART_init(&g_uart,
             COREUARTAPB0_BASE_ADDR,
             BAUD_VALUE_115200,
             (DATA_8_BITS | NO_PARITY));
  while(1)
  {
      UART_polled_tx_string(&g_uart, (const uint8_t *)&testmsg);
      delay(1);
  }
}
```