Modularity in Engine Control Units

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Autonomous Systems and Robotics
June 2025
Eetu Pesonen

Supervisors:
Yasir Al-Ameri
Tomi Westerlund
Olli Vierunketo (Wärtsilä)

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

UNIVERSITY OF TURKU
Department of Computing

Eetu Pesonen: Modularity in Engine Control Units

Master of Science (Tech) Thesis, 64 p.
Autonomous Systems and Robotics
June 2025

The increasing complexity and long service life of large marine and industrial combustion engines, combined with the rapid evolution of electronics, tightening emissions regulations, and industry demand for alternative fuels, demand more adaptable and cost-effective control systems. Traditional engine control systems based on highly integrated, monolithic control modules are limited in their ability to adapt to new functionalities without a complete redesign, leading to high development costs and long validation cycles. This thesis introduces a modular architecture for Engine Control Units (ECUs), where I/O functions are offloaded to expansion cards connected via a serial bus interface. The approach decouples the processor and other common functions from I/O development, enabling independent upgrades, simplified lifecycle management, and reduced engineering overhead.
A key focus of this work is the design of a reliable, low-latency communication protocol between the processor board (mainboard) and expansion cards. A prototype implementation is presented, featuring a custom AXI4-Lite bridge over serial low-voltage differential signaling (LVDS) links, including error detection, retransmission support, and Linux integration based on the dynamic device tree kernel framework. The system supports FPGA- and MCU-based expansion cards, with EEPROM-based autodetection and runtime driver instantiation. Performance and fault tolerance were validated through simulation and hardware testing, where submicrosecond latency and 1.3 million AXI transactions per second, with robust error recovery, was achieved. The modular approach demonstrates improved flexibility, maintainability, and subcontractor engagement potential, with favorable long-term economic implications.

Keywords: Embedded Systems, Engine Control Unit, Modularity, Expansion Bus, AXI4, FPGA, System-on-Chip, Linux, Real-Time Communication

TURUN YLIOPISTO
Tietotekniikan laitos

Eetu Pesonen: Modularity in Engine Control Units

Diplomityö, 64 s.
Autonomiset järjestelmät ja robotiikka
Kesäkuu 2025

Suurten, merellä ja teollisuudessa käyettyjen polttomoottorien tekniikan jatkuva monimutkaistuminen ja pitkä käyttöikä, yhdistettynä elektroniikan nopeaan kehitykseen, kiristyviin päästörajoituksiin, ja haluun käyttää vaihtoehtopolttoaineita, edellyttävät entistä joustavampia ja kustannustehokkaampia ohjausjärjestelmiä. Perinteiset moottorinohjausjärjestelmät perustuvat integroituihin, monoliittisiin ohjausyksiköihin, jotka vaativat merkittävästi suunnittelutyötä elektroniikkaa muutettaessa. Tämä johtaa korkeisiin kehityskustannuksiin ja pitkiin validointiprosesseihin. Tässä työssä esitetään modulaarinen arkkitehtuuri, jossa ohjausyksikön I/O-toiminnot on siirretty erillisille laajennuskorteille, jotka liitetään pääkorttiin sarjaväylän välityksellä. Tämän rakenteen ansiosta prosessori- ja I/O-osien kehitys voidaan erottaa toisistaan, mikä helpottaa järjestelmän elinkaaren hallintaa ja pienentää suunnittelukuormaa.
Työn keskiössä on luotettavan ja matalaviiveisen tiedonsiirtoprotokollan kehitys pääkortin ja laajennuskorttien välille. Esitetty prototyyppi perustuu AXI4-Lite-siltaan, joka toimii differentiaalisten sarjalinkkien yli. Protokolla sisältää virheentunnistuksen, uudelleenlähetyksen ja integroituu Linux-järjestelmään hyödyntäen ytimen dynaamista laitepuurakennetta (dynamic device tree). Järjestelmä tukee sekä FPGA- että mikrokontrolleripohjaisia laajennuskortteja, joissa korttikohtainen EEPROM mahdollistaa automaattisen tunnistuksen ja ajurien ajonaikaisen lataamisen. Ratkaisun suorituskykyä ja virheensietokykyä arvioitiin simulaatioiden ja laitteistotestauksen avulla, jotka osoittivat väylän kyekenevän alle mikrosekunnin viiveeseen ja noin 1.3 miljoonaan operaatioon sekunnissa. Virheensieto toimi odotetulla tavalla. Modulaarinen rakenne osoittautui joustavaksi ja helposti ylläpidettäväksi. Lisäksi se tukee alihankkijayhteistyötä sekä tarjoaa pitkällä aikavälillä merkittäviä taloudellisia etuja.

Asiasanat: Sulautetut järjestelmät, Moottorinohajusyksiköt, Modulaarisuus, Laajennusväylä, AXI4, FPGA, System-on-Chip, Linux, Reaaliaikainen tiedonsiirto

Contents

1 Introduction **1**

1.1 Wärtsilä 1

1.2 Background and Motivation 1

1.3 Objectives 2

1.4 Research Questions 3

1.5 Related Work 3

2 Architecture **5**

2.1 Current Control Module Architecture 5

2.2 Modularization 6

2.2.1 Module Core 8

2.2.2 Expansion Card Concept 9

2.2.3 Power Delivery 9

3 Communication **13**

3.1 Bus Topology 13

3.1.1 Multidrop Bus 13

3.1.2 Point-to-Point Bus 14

3.1.3 Tree Topology 14

3.2 Command-Based Protocols 15

3.2.1 SPI 16

3.2.2 I2C 18
3.3 Memory-Mapped I/O 18
3.3.1 PCI Express 19
3.3.2 AXI4 20
3.3.3 AXI4-Lite 21
3.4 Streaming Protocols 23
3.4.1 AXI4-Stream 23
3.4.2 Aurora 24
3.5 Physical Layer Considerations 25
3.5.1 Single-ended Signaling 25
3.5.2 Differential Signaling 25
3.6 Clocking Schemes and Clock Recovery 26
3.6.1 Clocking Schemes 27
3.6.2 Clock Recovery 28
3.6.3 Design Implications 28
3.7 Interrupts 29
3.7.1 Interrupt Signaling 29
3.7.2 Chained Interrupts 30

4 Linux Integration 32

4.1 Device Tree 32
4.2 Device Drivers 33
4.2.1 User-Space interfaces 33
4.3 Bus Drivers 34
4.3.1 Dynamic Device Tree 35
4.4 FPGA Configuration 35
4.4.1 Configuration Modes in Xilinx Devices 36
4.5 MCU Firmware Upgrade 38

5 Prototype Construction **40**
5.1 Hardware Description 42
5.2 AXI Bridge Implementation 42
5.2.1 Protocol Stack Overview 44
5.2.2 Expansion Bus Control Interface 49
5.3 Linux Integration 49
5.3.1 Bus Driver 50
5.4 Testing 56
5.4.1 Simulating the AXI Bridge IP 56
5.4.2 Testing the AXI Bridge on Hardware 57
5.4.3 System Testing 57

6 Conclusions **60**
6.1 Impact of Modularization 60
6.1.1 Design Flexibility and Scalability 60
6.1.2 Subcontractor Engagement 61
6.1.3 Cost Implications 61
6.2 Research Questions Revisited 62
6.3 Future Work 63

References **65**

List of Figures

2.1 UNIC high level architecture 5

2.2 Control module architecture 7

2.3 Modularized architecture 8

2.4 Power delivery architecture with redundant power inputs 12

3.1 SPI Clocking schemes 16

3.2 SPI Clock Skew, Mode 3: data out on falling edge, latched on rising edge. Master latches the previous bit value due to clock skew. 17

3.3 AXI4-Lite read transaction 22

3.4 AXI4-Lite write transaction 23

3.5 Chained Interrupts on an FPGA 31

5.1 Simplified schematic of the system with two expansion cards shown. The downstream interface on an expansion card is identical to the master's interface. 43

5.2 High-level FPGA designs of the main board and expansion cards . . . 44

5.3 (a) Logical bridge architecture (b) The resulting address mappings . . 46

5.4 Link-layer synchronization sequence between transceivers A and B . . 49

List of Acronyms

ACK Acknowledgment
AMBA4 Advanced Microcontroller Bus Architecture 4
ARQ Automatic Repeat reQuest
AXI4 Advanced eXtensible Interface 4
AXIL AXI4-Lite
AXIS AXI4-Stream
BOM Bill of Materials
CAN Controller Area Network
CPU Central Processing Unit
CRC Cyclic Redundancy Check
DC Direct Current
DDR Double Data Rate
DSP Digital Signal Processing
EEPROM Electronically Erasable Programmable Read Only Memory
EMI Electro Magnetic Interference
FMC FPGA Mezzanine Card
FPGA Field Programmable Gate Array
GPIO General Purpose Input and Output
HAL Hardware Abstraction Layer
I/O Input/Output
I^{2}C Inter-Integrated Circuit
IAP In-Application Programming
IOCTL Input/Output Control
IP Intellectual Property
JTAG Joint Test Action Group
LVDS Low Voltage Differential Signaling

MCU	Micro Controller Unit
MMIO	Memory Mapped Input and Output
MSI	Message Signaled Interrupt
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PCIe	PCI Express
PDN	Power Delivery Network
PMIC	Power Management Integrated Circuit
ROM	Read-Only Memory
SRAM	Static Random Access Memory
SoC	System-on-Chip
SPI	Serial Peripheral Interface
TVS	Transient Voltage Suppressor
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus

1 Introduction

1.1 Wärtsilä

Wärtsilä is a global leader in innovative technologies and life cycle solutions in marine and energy markets. During its 190 years of existence, it has transitioned from a shipbuilder and engine maker into a technology-driven enterprise focusing on sustainability, digitalization, and decarbonization.

Wärtsilä is best known for its power generation solutions, including engines for both the marine and energy industries. This thesis focuses on control systems used on Wärtsilä's marine and energy engines.

1.2 Background and Motivation

Engine control systems are responsible for managing various aspects of engine performance, such as fuel injection, air intake, and emissions control. Emissions regulations force engines to become more sophisticated, steering the industry toward carbon-free alternative fuels and demanding more flexible engine control systems that incorporate new types of sensors and actuators together with increased processing power.

The control system used on Wärtsilä's engines, UNIC, consists of a number of networked control modules. Different types of modules specialize in different aspects of engine control, such as combustion control and communication with ex-

ternal systems. The conventional design of these modules is based on a monolithic Printed Circuit Board (PCB) housing the main Central Processing Unit (CPU), Input/Output (I/O) electronics, and a Field Programmable Gate Array (FPGA) responsible for interfacing the I/O electronics with the CPU. This type of design poses several challenges for managing the system's life cycle and adopting future technologies, such as requiring a complete redesign when hardware changes are needed, which is both time-consuming and costly.

To address these challenges, this thesis proposes a modular architecture for the control modules' internal implementation, where most of the I/O functionality is offloaded to separate expansion cards. This type of architecture not only simplifies the initial design and validation of a control module, but additionally allows for easier updates and life cycle management. By separating the I/O functionality from the main processor board, future enhancements can be made by simply redesigning and replacing individual expansion cards or the processor board, without the need for complete redesign and validation.

1.3 Objectives

This thesis aims to:

- Develop a modular architecture for the internal implementation of a control module.
- Design and evaluate a communication protocol for reliable, low-latency communication.
- Assess the proposed modular architecture, both from a technical and economic point of view.

1.4 Research Questions

This thesis aims to answer the following research questions:

1. How can the system architecture be designed to ensure future upgrades without the need for redesign of other parts of the system?
2. How can the communication be implemented to ensure efficient and reliable data transfer?
3. How can signal integrity be maintained in harsh operational environments?
4. What error detection and correction methods should be implemented to handle potential communication failures?
5. How does separating I/O functionality onto expansion cards impact design, validation, and manufacturing costs over the system's life cycle?

1.5 Related Work

High-Performance and Energy-Efficient Fault Tolerance FPGA-to-FPGA Communication

This work [1] explores reliable communication between FPGA devices using USB transceivers and BCH error correction codes. While it demonstrates strong fault tolerance and energy efficiency, the architecture relies on USB – a high-overhead protocol less suited for tightly coupled embedded control systems. Additionally, the use of BCH decoding and asynchronous FIFOs increases design complexity, latency, and resource utilization. In contrast, the AXI4-Lite bridge developed in this thesis emphasizes simplicity, deterministic timing, and tight integration with register-mapped peripherals, making it better suited for the low-latency demands of modular ECU I/O expansion.

A Flexible FPGA-to-FPGA Communication System

The Universal Protocol Interface (UPI) proposed in this work [2] supports the simultaneous transmission of multiple bus protocols, including PCI Express and Gigabit Ethernet, over high-speed links. While this architecture is well-suited for high-performance computing systems, it is overly complex and resource-intensive for embedded control applications. Moreover, it lacks native support for memory-mapped bus transactions, which complicates direct peripheral interfacing. In contrast, the AXI4-Lite bridge presented in this thesis uses a lightweight, AXI4-Lite-native protocol with reliable transport over low-pin-count LVDS links, offering deterministic behavior and minimal overhead for embedded I/O expansion.

Artificial Intelligence

Generative AI has been used in this thesis for language improvement and ideation.

2 Architecture

The UNIC automation system consists of a number of control modules, such as communication modules, cylinder control modules, and general I/O modules, which communicate over redundant Ethernet rings. The high-level system architecture is shown in Figure 2.1.

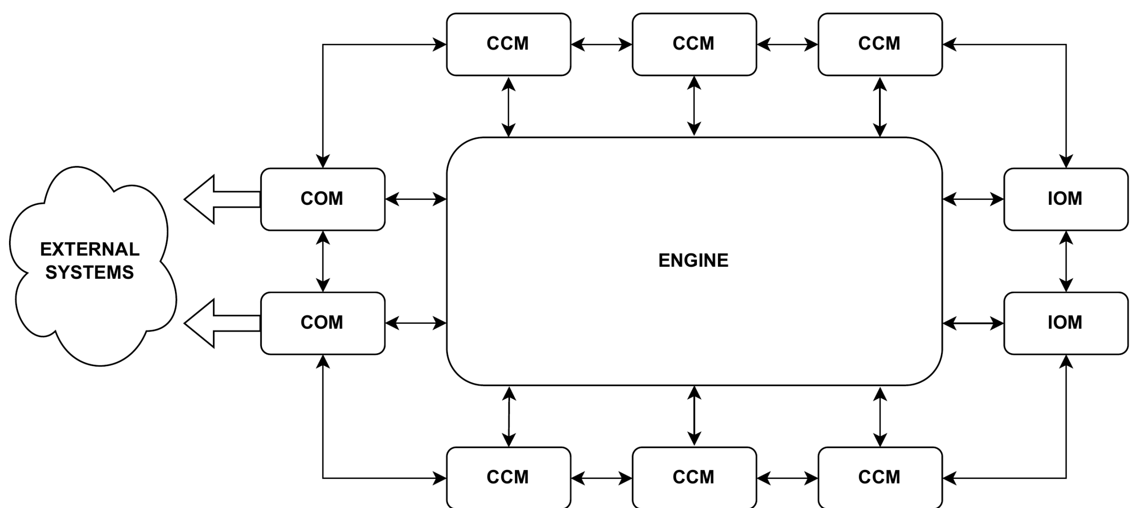

Figure 2.1: UNIC high level architecture

2.1 Current Control Module Architecture

Each control module in the current architecture, illustrated in Figure 2.2, integrates a CPU and an FPGA on a single PCB, together with the associated I/O electronics. The FPGA handles real-time control of various I/O channels, including digital I/O, analog 4–20 mA current loops, and specialized channels for application-specific use.

Additionally, it performs various Digital Signal Processing (DSP) tasks required for control applications and input signal conditioning.

This monolithic design offers a compact and integrated solution; tight integration of all subsystems into a single hardware unit reduces latency between components and simplifies production. However, this architecture presents several limitations and long-term challenges, including limited flexibility, increased lifecycle management complexity, and inefficient resource utilization.

As components become obsolete or unavailable during the system's lifetime, even minor hardware updates may necessitate redesign of the entire module, which is not only time-consuming and costly, but also requires complete electrical and environmental testing, validation, and certification. The same applies to hardware bug fixes and the introduction of additional functionality; any such change affects the entire board.

Moreover, using a single standardized control module across various engine configurations may result in either over-provisioning, where certain I/O channels remain unused, or under-provisioning, necessitating the use of multiple modules to achieve the required functionality. Both outcomes increase system cost and complexity.

2.2 Modularization

As discussed in the previous section, the current monolithic architecture poses significant challenges in terms of flexibility, scalability, and maintainability. With increasing system complexity and variation in engine configurations, the need for a more modular and adaptable control module design becomes clear.

To address these challenges, a modular architecture is proposed, illustrated in figure 2.3. In this approach, the control module is divided into a common *module core* and one or more replaceable *expansion cards*. This separation decouples the computing and communication functionalities from the more application-specific

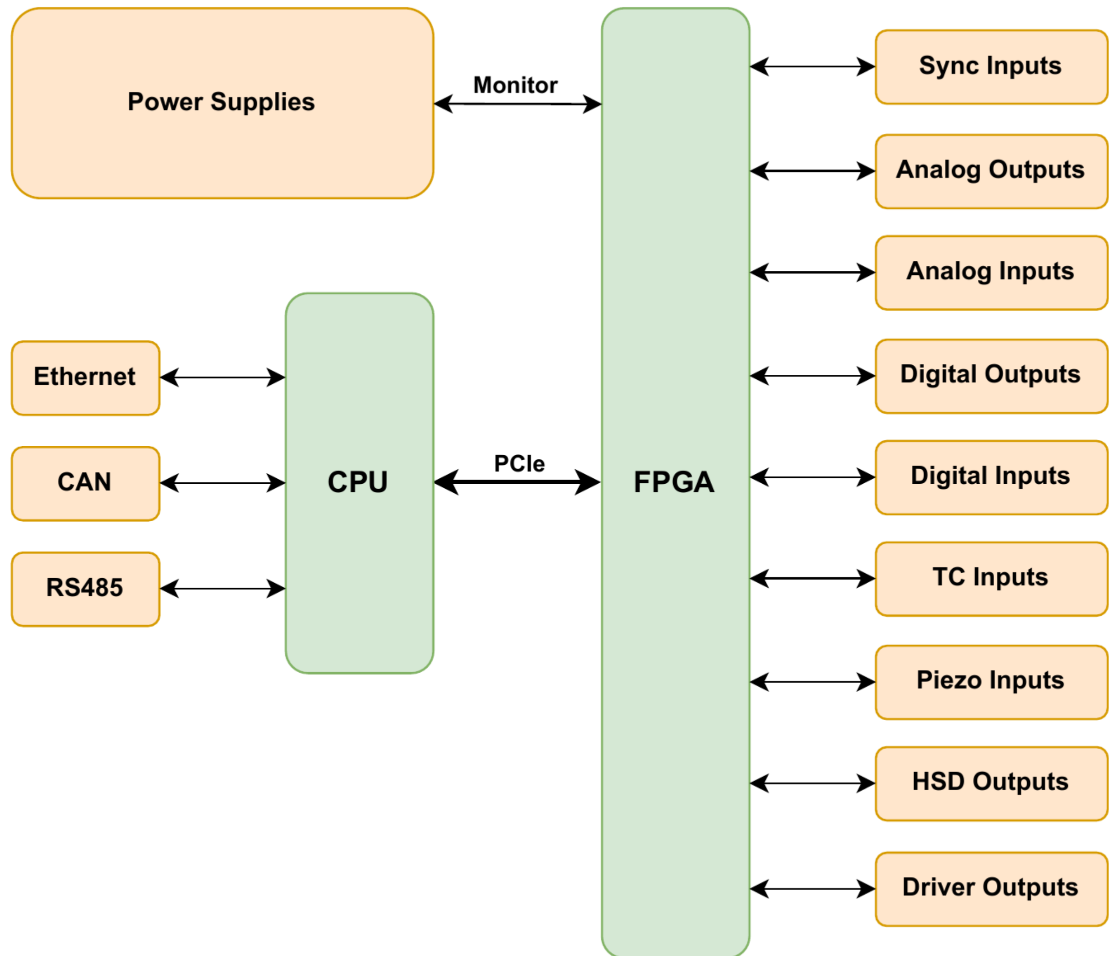

Figure 2.2: Control module architecture

and evolving I/O electronics, providing several benefits for both hardware and software development and maintenance. In general, modularity allows more accurate alignment between hardware capabilities and actual engine requirements, reducing material costs and system complexity.

Flexibility is increased by allowing an arbitrary set of expansion cards to be combined with the core, enabling creation of tailored control modules for specific engine variants, enabling more efficient use of hardware resources and development of highly application-specific hardware instead of complex, general-purpose I/O channels. Changes in hardware design due to lifecycle management or bug fixes are limited to either the module core or an affected expansion card, reducing cost and

time-to-market. Likewise, introducing new functionality becomes faster and cheaper as only an expansion card and its driver need to be developed, validated, and certified.

From a software point of view, the standardized module core provides a consistent firmware platform; the expansion cards only require driver and Hardware Abstraction Layer (HAL) support for integration. This provides for a common software environment across all modules and makes firmware development easier.

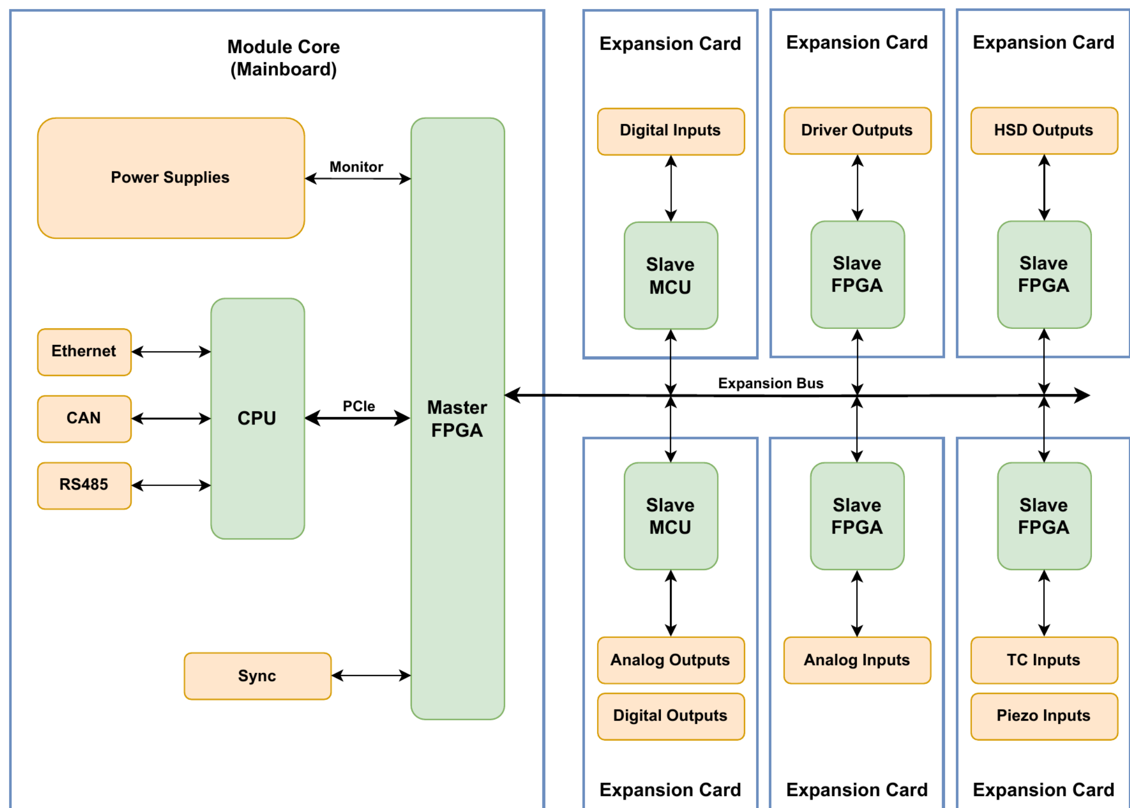

Figure 2.3: Modularized architecture

2.2.1 Module Core

The module core is always present in a control module. It provides the primary computing resources and common infrastructure that stay constant between applications, such as power distribution and conditioning circuits, communication inter-

faces to external systems and other modules, and synchronization logic. An FPGA is included for managing on-board I/O and the expansion bus, to which the I/O cards are connected. Because the core is consistent across module types and applications, it forms a stable, reusable foundation for both hardware and firmware development. This stability reduces development effort for future product generations and simplifies system-wide integration.

2.2.2 Expansion Card Concept

Each expansion card interfaces with the module core via an expansion bus. This bus provides power, bidirectional data channels, synchronization signals, and interrupt lines. It is designed to support low-latency, real-time data exchange and scalable integration of multiple expansion cards while imposing minimal restrictions for key component selection on the expansion cards. An expansion card typically contains a small FPGA or Micro Controller Unit (MCU) for controlling real-time I/O operations and bus interfacing, I/O electronics, and local power supplies. An on-board Electronically Erasable Programmable Read Only Memory (EEPROM) stores identification and calibration data.

Cards can be classified as general-purpose or application-specific. Standardized card form factors and pinouts promote reuse across different control module types and simplify the design of future card variants.

2.2.3 Power Delivery

A system's Power Delivery Network (PDN) is responsible for providing adequate power to the system's components. Different components require different voltage levels and some components require multiple supply voltages, each with specific stability, noise, and sequencing constraints. With high-speed components, proper PDN design is critical to ensure reliable and deterministic system operation, particularly

when high-speed or sensitive components are involved.

In marine systems [3] Direct Current (DC) power supplies usually operate at 24 volts, and several protection mechanisms are required, such as surge, reverse polarity, over voltage and over current protection, and Electro Magnetic Interference (EMI) filtering. In typical marine environments, 24 V buses are often battery-backed and may experience short interruptions due to relay operations or generator transitions. The power architecture must be robust against such events. To tolerate short interruptions in external power, energy storage methods such as large bulk capacitors or onboard supercapacitors can be used to maintain rail voltage for critical components.

General PDN design practices [4] include keeping loop areas small for reduced EMI emissions, the use of bypass and decoupling capacitors close to power pins, and low-impedance return paths, such as ground planes. Analog power supplies should be provided by linear regulators with sufficient filtering instead of switching converters to guarantee low noise.

Low-voltage regulation should be performed as close to the point-of-load as possible to minimize parasitic inductance and voltage droop, improving transient response and power integrity. Power supply diagnostics, such as input and output voltage and current measurement, should be made available for the system to make decisions, such as preparing for loss of power or taking measures to decrease load.

Devices such as FPGAs require strict power-up and power-down sequencing to avoid undefined behavior or permanent damage. Sequencing can be implemented using *power good* signals to enable other regulators or by utilizing power supervisors such as Power Management Integrated Circuit (PMIC) devices.

Suggested Power Delivery Network

The proposed power delivery system, depicted in figure 2.4, balances central regulation and local autonomy. It simplifies power routing while maintaining flexibility

for diverse expansion card requirements.

The main board acts as the central power source in the control module. It conditions the raw power input from external systems and regulates it to intermediate, high current capable power buses of 24 and 5 volts, from which local voltages are then generated. The intermediate power buses are also fed to the expansion cards.

The expansion cards may have different power supply requirements. Modern FPGA devices require several voltages, including 3.3, 2.5, 1.8, and 1.0 volts, in addition to the voltages required by I/O electronics. Some I/O channels might require DC isolation for safety, noise immunity, or handling differences in ground potential. Isolation can be implemented via optocouplers for signals and capacitor or transformer based isolated switching converters for power supplies. The generation and monitoring of these local power rails is to be done on the expansion card, using the 5 and 24 volt rails provided by the main board. Expansion cards containing high-current I/O channels may require dedicated external power.

The power connections to expansion cards must be properly sized. One option is to use multiple connector pins to reduce contact resistance and increase current capability; another option is to select a connector that has dedicated high-current capable power pins in addition to the data pins. Current limiting, either active limiting or resettable fuses, should be installed to protect the expansion interface and connected devices from over current events. If the expansion interface is to be made external, the power connections should be fitted with additional protections, such as Transient Voltage Suppressor (TVS) diodes.

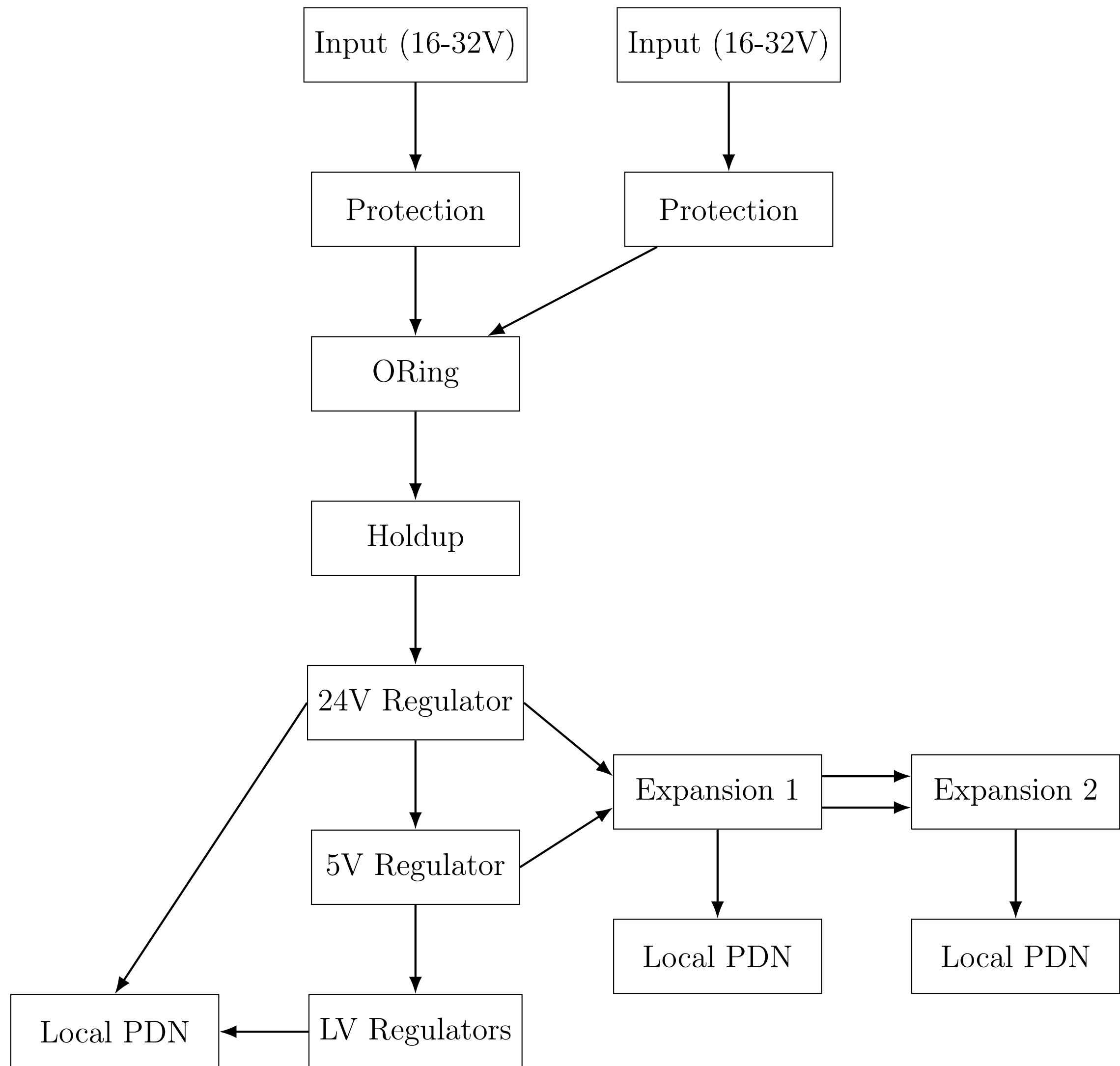

Figure 2.4: Power delivery architecture with redundant power inputs

3 Communication

The communication bus between the module core and the I/O cards provides data, interrupt, and synchronization signals. There are several methods to arrange communication with subsystems, including command-based protocols, Memory Mapped Input and Output (MMIO) interfaces, and streaming protocols.

3.1 Bus Topology

Communication buses can be arranged in different topologies, such as multidrop and star topologies. Sections of a bus system may adopt different topologies, provided the underlying communication protocols support such hybrid configurations.

3.1.1 Multidrop Bus

In a multidrop bus all signals are shared between the devices on the bus. While multidrop topologies reduce wiring complexity, they introduce electrical challenges such as signal reflections and impedance mismatches at higher frequencies, and necessitate arbitration mechanisms to prevent contention. These may involve centralized coordination (Serial Peripheral Interface (SPI) [5]), master/slave addressing (Inter-Integrated Circuit (I^2C) [6]), or collision detection schemes (Controller Area Network (CAN) [7]). Electrical challenges can often be mitigated through careful layout, impedance matching, and signal termination strategies.

3.1.2 Point-to-Point Bus

A point-to-point bus is a dedicated physical channel between two devices. A point-to-point bus can be expanded to work with multiple devices in several ways while preserving simple arbitration, two common ways including daisy chaining and star topology. Due to its simplicity and improved signal integrity, a point-to-point bus is generally preferred for high-speed communication.

A daisy chained bus consists of a set of devices connected in a chain using point-to-point links. In this topology, each device talks to the next downstream device; arbitration is simple and the number of signals is limited. However, when a device wishes to communicate with another device further away, the data needs to be passed through the devices in the middle, increasing latency and requiring some routing support from the intermediate devices.

A widely used variant, ring topology, is attained by connecting the ends of the chain together. This allows for unidirectional links between the devices, where communication with an upstream device is done by looping through the downstream devices in the link. If bidirectional links are used, ring topology provides redundancy: if a link breaks, traffic can be routed through the ring in the other direction.

In star topology, one device acts as the master to which all peripheral devices are connected via point-to-point links. This centralization simplifies control logic but may limit scalability and increase the load on the master device. Examples of star topology include Ethernet, where arbitration is handled either through carrier sensing and collision detection in traditional shared-medium Ethernet or via full-duplex switching in modern variants.

3.1.3 Tree Topology

Tree topology is a hierarchical extension of the star topology, where point-to-point links connect devices in a parent-child structure. Each non-leaf device - typically an

active component such as a hub, switch, or interconnect - acts as an intermediary node that can forward communication to and from its children, forming a branching tree-like structure. A typical example of tree topology is Universal Serial Bus (USB): the host connects to one or more devices, which may themselves be hubs capable of hosting additional devices. Also, Peripheral Component Interconnect (PCI) Express [8] and Advanced eXtensible Interface 4 (AXI4) [9] use a logical tree structure, even though they often resemble a star topology.

Advantages of tree topology include scalability and ease of addressing. However, traversing multiple intermediate nodes increases latency and reduces bandwidth. Tree topology is a natural fit for systems where hierarchical addressing, modular growth, or centralized management are desirable. It bridges the simplicity of star topology with the need for scalable interconnection, making it a common design choice in embedded systems and computer interconnects.

3.2 Command-Based Protocols

A command-based protocol consists of a set of commands or instructions, possibly accompanied by additional data. This interface is used in, for example, serial EEPROM devices, where the user can issue a read, write, or erase command. For example, a read command contains the byte address to be read; the device then responds by transmitting to the user the memory contents at the specified address. Likewise, a write command contains the target address and the data to be written. [10] These buses tend to be relatively slow and have a high latency due to the protocol overhead. However, hardware implementation is often very simple.

3.2.1 SPI

SPI [5] is a synchronous full-duplex multi-drop communication bus designed by Motorola in the early 1980s, intended for communication between integrated circuits on a single PCB. The interface consists of a clock (SCLK) and two data signals, master-out-slave-in (MOSI) and master-in-slave-out (MISO), shared between all devices on the bus, and a select (SS) signal for each slave device. Signaling is single-ended with no mandatory termination. The master device outputs data on the MOSI line for the selected slave to consume. Similarly, the selected slave device outputs data on the MISO line for the master to consume. Data is transmitted one bit at a time, making SPI very flexible.

The relationship between the clock and data can be configured in four ways by selecting the desired clock polarity and clock phase, as shown in Figure 3.1. A clock configured with a polarity of zero rests at zero when no transmission is in progress. The clock phase specifies which clock edge to use for data transmission and sampling, with a phase of zero specifying the leading clock edge to be used for sampling data.

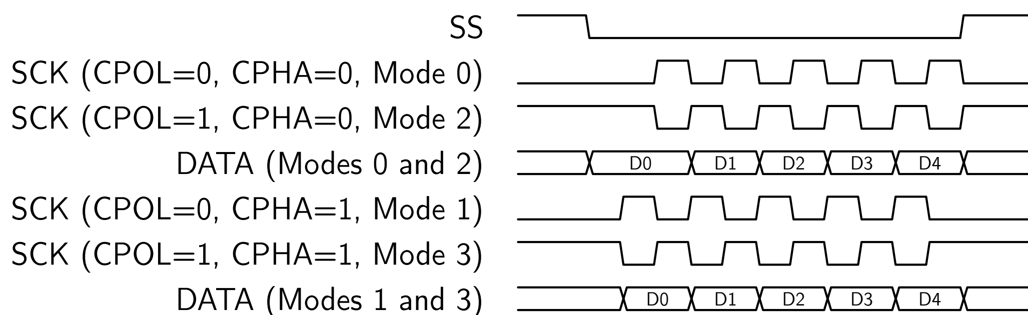

Figure 3.1: SPI Clocking schemes

The theoretical bit rate limit of an SPI bus depends on the clock period and the length of the transmission line mainly due to clock skew. The clock and data output from the master experience the same propagation delay and thus arrive

at a slave device in sync. However, the clock is used to output data from the slave device and load it to the master device. The data effectively experiences two propagation delays before reaching the master, being out of sync with the clock. Thus, the total propagation delay must be less than half of the clock period [11]. For example, with a bit rate of 100 Mbps, the clock period is 10 nanoseconds, limiting the transmission line delay to 2.5 nanoseconds, or approximately 38 centimeters assuming a propagation delay of 15 centimeters per nanosecond [12]. A situation where too long propagation delay is present is shown in Figure 3.2

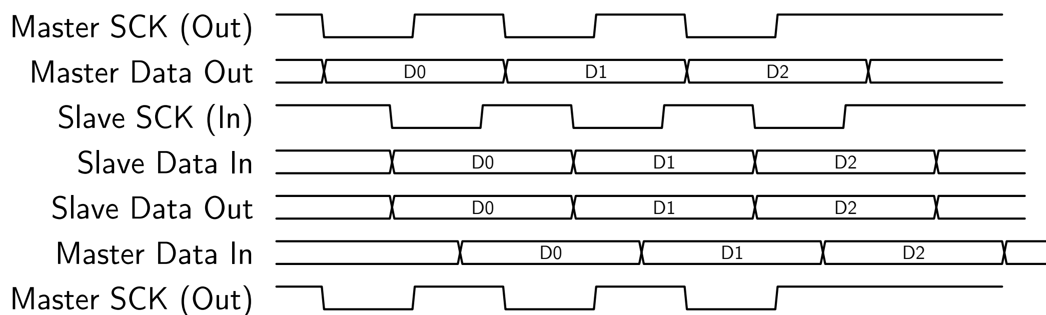

Figure 3.2: SPI Clock Skew, Mode 3: data out on falling edge, latched on rising edge. Master latches the previous bit value due to clock skew.

A simple way to mitigate clock skew is to use clock loopback. Looping the clock back to the master for data input effectively eliminates skew. However, on the master side this requires a non-standard SPI controller or the use of an additional SPI slave interface for data input.

QSPI

Quad SPI is a widely used derivative of the SPI bus. Instead of dedicated MOSI and MISO signals, it has four bidirectional data lines, which allow transmitting four bits at a time, effectively quadrupling the bit rate with respect to SPI. However, a more sophisticated protocol is needed for controlling the medium, since the master and slave cannot transmit at the same time.

Quad SPI is primarily used with flash memories utilizing a standard protocol that defines a set of commands and transactions. [10]

3.2.2 I2C

I^2C [6] is a half-duplex, synchronous communication bus developed by Philips in the early 1980s for communication between integrated circuits in a system. It uses only two connections with single-ended open-drain signaling, data (SDA) and clock (SCL), where the current bus master pulls the lines low to indicate a logical low and releases them to indicate a logical high. Pull-up resistors are used to pull the lines high. This makes communication half-duplex, as only one device can transmit at a time. Transactions include address information; no slave select lines are used.

The maximum speed of an I^2C link is limited mainly by the RC action caused by the capacitance of the bus and the pull-up resistors; rising edges are quite slow, limiting bus speed to just a few megabits per second. However, I^2C is very popular as a system management bus and for communicating with slower peripherals due to its simple two-wire interface.

3.3 Memory-Mapped I/O

In a MMIO interface the main idea is to expose the peripheral's control registers and memories to the host device's address space. Each peripheral occupies a region of memory addresses, starting at a specified *base address*. The host then communicates with the peripheral using standard methods for accessing memory; to access the contents of a control register, the host accesses data at a memory address calculated from the base address and the register's offset. This provides high-throughput, low-latency communication with the peripheral while also making device driver development simple. However, memory-mapped buses usually consist of either a large

number of parallel signals or a set of very-high-speed serial signals, making hardware more complex and costly compared to simper interfaces such as SPI and I^2C.

Examples of MMIO interfaces include PCI and AXI4 buses used for communication between a computer's CPU and peripherals, such as SPI controllers and graphics cards.

3.3.1 PCI Express

PCI Express (PCIe) [8] is a high-speed serial communication bus introduced in 2003 by Intel and other industry partners as a replacement for the original, parallel PCI bus standard. It maintains backward compatibility across generations, allowing newer devices to negotiate down to older speeds and lane widths. Bridges for connecting legacy PCI devices to a PCIe bus are also available. PCI Express uses point-to-point serial links composed of differential transmit and receive pairs, organized into full-duplex lanes. Each lane includes an embedded clock and utilizes line encoding to support robust clock recovery and signal integrity at high data rates.

A single PCI Express lane can achieve a unidirectional throughput of up to 32 Gbps in version 5.0, with future versions targeting even higher rates. For increased bandwidth, multiple lanes can be bonded together into `x2`, `x4`, `x8`, or `x16` configurations, allowing aggregate throughput scaling to match system requirements.

A PCI Express bus network begins at a controller called the *Root Complex*. The Root Complex is connected to a switch, to which peripheral devices or other switches may be connected. In simple systems with only a single peripheral, the switches may be omitted entirely, with the Root Complex directly communicating with the peripheral.

The PCI Express protocol stack is composed of three layers: the Transaction Layer, the Data Link Layer, and the Physical Layer. The Data Link Layer transmits packets over a point-to-point link between two adjacent devices using a sim-

ple Automatic Repeat reQuest (ARQ) protocol together with Cyclic Redundancy Check (CRC) integrity validation, making the link highly reliable. Data, interrupts, and control signals are encapsulated into Transaction Layer Packets (TLPs), which the Transaction Layer routes through the PCIe network, relying on the Data Link Layer for reliable transmission. A TLP contains a header encoding the packet type, length, and destination address. The Transaction Layer assumes reliable delivery from the lower layers and does not perform additional error checking or delivery acknowledgments.

Processors intended for embedded use, such as the NXP i.MX series, typically have limited or no PCI Express support, often requiring external switches if more than one peripheral device is to be connected. Thus, while PCI Express may not be directly applicable to deeply embedded or cost-sensitive applications, it serves as a well-established model for scalable, high-performance serial communication.

3.3.2 AXI4

AXI4 [9] is part of the ARM Advanced Microcontroller Bus Architecture 4 (AMBA4) specification and defines a high-performance communication protocol widely used in FPGA and System-on-Chip (SoC) designs. It defines a single interface for use between a master and a slave, a master and an interconnect, and a slave and an interconnect, enabling scalable communication architectures for internal data movement.

The AXI4 interface comprises five independent unidirectional channels: the read address channel, write address channel, read data channel, write data channel, and write response channel. Each channel uses a handshake mechanism based on the `VALID` and `READY` signals. The source asserts `VALID` to indicate that valid data or control information is available, while the destination asserts `READY` to signal its readiness to accept the transfer. A data transfer occurs only when both signals are

asserted concurrently.

AXI4 Transactions

An *AXI4 transaction* is initiated by a master to communicate with a slave. Each transaction consists of a set of coordinated transfers across multiple channels. To start a transaction, the master issues control information such as the transaction type, burst parameters, and start address on either the read or write address channel.

Data is transferred in a *burst*, which consists of one or more data transfers referred to as *beats*. The master or slave signals the end of the burst using the `LAST` signal on the data channel. Bursts allow efficient block data transfers while reducing control overhead.

Transaction ID signaling can be used to support *outstanding transactions*, i.e. the master may initiate multiple transactions, each with a unique ID, and then wait for the slave to respond. This increases throughput since the master does not need to wait for the slave device, reducing bus idle time.

3.3.3 AXI4-Lite

AXI4-Lite (AXIL) is a simplified subset of the full AXI4 specification. It provides a lightweight, register-style interface with no support for burst transactions. The data bus width is limited to either 32 or 64 bits, depending on the implementation, and no outstanding transactions are supported.

Read Transaction

A read transaction begins when the master drives the address onto the *Read Address* channel and asserts `ARVALID` to indicate that the address is valid. The slave acknowledges by asserting `ARREADY`, completing the address handshake. Afterward, a single data beat is returned: the slave places the read data and response code (`RRESP`)

onto the *Read Data* channel and asserts RVALID. The master asserts RREADY when it is ready to accept the data. The read operation completes when both RVALID and RREADY are asserted simultaneously.

Alternatively, the slave may assert ARREADY before the master asserts ARVALID; the transfer still occurs when both are asserted. A similar early handshake can occur on the read data channel, where the master asserts RREADY before RVALID. Data is always transferred only when both signals are high.

Figure 3.3: AXI4-Lite read transaction

Write Transaction

A write transaction begins when the master places the target address on the *Write Address* channel and asserts AWVALID. The slave acknowledges the address by asserting AWREADY. Next, the master transfers a single data beat by driving the *Write Data* channel and asserting WVALID. The slave accepts the data by asserting WREADY.

After the data transfer, the slave issues a write response by placing a status code (BRESP) onto the Write Response channel and asserting BVALID. The master completes the transaction by asserting BREADY to acknowledge receipt of the response.

As with read transactions, the handshakes may occur with the slave or master preemptively asserting READY before the other side asserts VALID. Additionally, the WSTRB signal can be used to selectively mask byte lanes during a write operation.

Masked bytes will not be written to the slave.

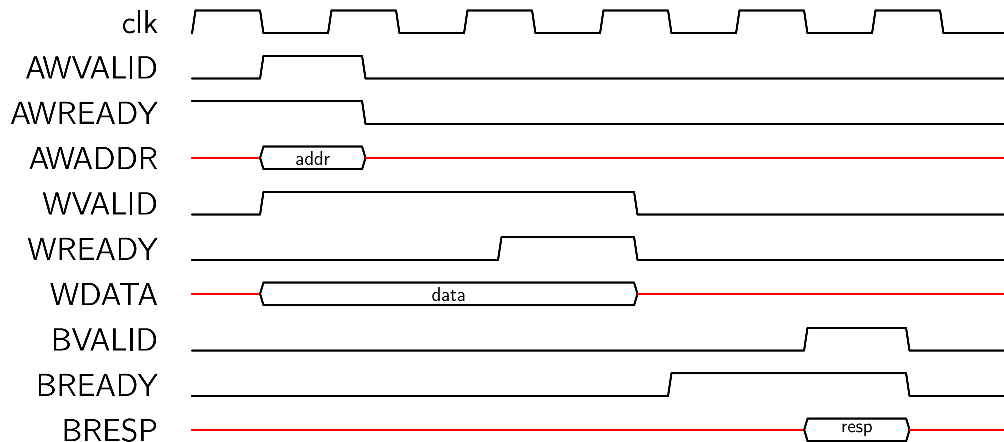

Figure 3.4: AXI4-Lite write transaction

3.4 Streaming Protocols

Streaming protocols are designed to transfer data continuously between a source and a sink without the overhead of address-based memory operations. They are ideal for high-bandwidth, low-latency applications like video pipelines, ADC data capture, and packet-based communication. These protocols typically use a handshake mechanism to manage data flow and support flow control through backpressure.

3.4.1 AXI4-Stream

AXI4-Stream (AXIS) [9] is part of the ARM AMBA4 specification, defining a point-to-point, unidirectional streaming protocol. Unlike AXI4 memory-mapped protocols, AXIS omits the address and control phases. It focuses solely on data transfer. This simplicity makes AXIS highly suitable for pipelined and modular hardware designs, especially in FPGAs and SoCs.

Similar to other AXI4 protocols, AXIS uses a handshake mechanism based on `TVALID` and `TREADY` signals. The data on `TDATA` is transferred on the rising clock

edge where both `TVALID` and `TREADY` are asserted, allowing for dynamic flow control — the receiver can throttle data by deasserting `TREADY`.

AXIS does not define a frame structure, but a common practice is to use the optional `TLAST` signal to indicate the end of a logical frame or packet. This enables protocols to be layered on top of AXIS for applications like Ethernet, video streams, or custom packet formats. Additional optional signals such as `TKEEP` and `TUSER` provide byte-level granularity and auxiliary metadata transmission.

3.4.2 Aurora

Aurora [13] is a lightweight link-layer protocol developed by Xilinx (now AMD) for high-speed serial point-to-point communication, typically between FPGA devices on a PCB or system.

Aurora uses differential signaling via the integrated high-speed serial transceivers available in AMD FPGA devices. Each Aurora link consists of one or more full-duplex lanes (up to 16), where each lane comprises a transmit and receive differential pair. Similar to PCI Express, it supports channel bonding to aggregate bandwidth across multiple lanes.

The protocol embeds the clock within the data stream, eliminating the need for a separate clock line and mitigating clock skew across lanes. Data rates range from 480 Mbps to 84.48 Gbps, depending on FPGA generation, transceiver type, and number of active lanes.

Aurora is implemented using the AMD Aurora IP core, which provides a standardized AXI4-Stream interface for integration into the system.

3.5 Physical Layer Considerations

The communication protocols described in this chapter span a range of physical layer characteristics that influence signal integrity, noise resilience, and speed. Low-speed interfaces, with clock speeds less than a few tens of megahertz, are usually based on single-ended signaling. Another option, usually used with high-speed interfaces, is differential signaling.

3.5.1 Single-ended Signaling

In single-ended signaling the signal is transmitted using a single wire, where the logic state is encoded in the voltage level of the wire with respect to the system ground; higher voltage for a logic high and lower voltage, typically close to zero volts, for a logic low. The voltage swing approaches the full I/O voltage range, typically around 3.3 volts, and may exhibit overshoot or undershoot. This leads to signal integrity issues and high EMI emissions in high-speed systems, limiting the maximum signaling rate.

Series termination resistors can be used to slow down signal edges, which reduces both EMI emissions and the maximum signal rate. Single-ended signaling suffers from susceptibility to noise.

3.5.2 Differential Signaling

In differential signaling the signal is transmitted over a pair of wires, one positive and the other negative, where the voltage between the two wires encodes the logic state; a logic high is represented when the positive wire is at a higher voltage than the negative wire, and a logic low otherwise. Differential pairs are routed with controlled impedance, commonly around 100 ohms, termination resistors, and matched lengths to maintain signal integrity.

The benefits of differential signaling include low voltage swing, which enables higher speed and lessened EMI emissions. Differential links are immune to common-mode noise, i.e. noise that affects both signal wires in a similar way, since the signal is encoded in the voltage difference.

LVDS

Low Voltage Differential Signaling (LVDS) is a high-speed, low-power digital signaling standard designed for transmitting binary data across differential point-to-point connections. It is widely used in applications demanding high data rates with minimal EMI emissions, including board-to-board, chip-to-chip, and cable connections. It is standardized as both TIA/EIA-644 [14] and IEEE 1596.3 [15].

LVDS operates using a low voltage swing, typically 250–450 mV, across a 100 Ω differential termination. The typical common-mode voltage is approximately 1.2 V, allowing operating voltages down to 2.5 V. Driver currents are kept low, typically less than 4 mA, which reduces power dissipation. The receiver input sensitivity is as low as 100 mV differential, with a required termination impedance between 90 Ω and 132 Ω. The interface is resilient to ground potential differences of up to 1 V.

To ensure signal integrity, matched impedance along the entire path - traces, connectors, and terminations — is critical. For multidrop setups, short stub lengths and a single termination at the far end are required. The signaling rate of LVDS is application dependent, mainly limited by signal integrity and distance. TIA specifies a theoretical maximum of 1.923 Gb/s, assuming ideal media [14].

3.6 Clocking Schemes and Clock Recovery

In high-speed digital interfaces, the method of clock distribution and synchronization between transmitter and receiver is critical for reliable data transfer. As signaling rates increase, precise clocking becomes essential to sample data at the correct time

and maintain data integrity.

3.6.1 Clocking Schemes

Clocking schemes are typically divided into two main categories, *source-synchronous* and *embedded clock* systems.

Source-Synchronous Clocking

In a source-synchronous scheme, the transmitting device sends a clock signal alongside the data. This clock is typically phase-aligned with the data and routed over a separate wire or differential pair. At the receiver, the incoming clock is used directly to latch the data, minimizing the timing uncertainty, skew, between data and clock.

This method is common in parallel bus systems and short-range point-to-point links, as it simplifies receiver design by avoiding the need to recover the clock from the data stream. However, it requires careful routing and matching of trace lengths to ensure that the clock and data arrive simultaneously.

Embedded Clocking

In systems where very high speed or minimizing the number of wires is critical, such as serial links or long-distance connections, the clock is often embedded within the data stream. This is typically achieved using line encoding schemes, such as Manchester or 8b/10b encoding, which guarantee regular signal transitions that allow the receiver to recover timing information from the data itself. Embedded clocking eliminates the need for a separate clock signal but requires more complex receiver circuitry capable of performing clock and data recovery (CDR).

3.6.2 Clock Recovery

In systems with embedded clocks, the receiver uses a phase-locked loop (PLL), delay-locked loop (DLL), or other synchronization mechanism to align a local sampling clock with the incoming data transitions.

The effectiveness of clock recovery depends on several factors, including transition density, jitter, and skew management. The encoding scheme must ensure sufficient signal transitions to keep the CDR circuit locked, the receiver must tolerate timing variations introduced by the channel, such as inter-symbol interference or supply noise. Additionally, in multi-lane systems, skew between lanes must be compensated for.

High-speed serial links often use specialized synchronization sequences or training patterns during link initialization to lock the receiver's CDR circuitry. In some systems, periodic re-synchronization may be required to account for long-term drift.

3.6.3 Design Implications

The choice of clocking scheme impacts physical design, signal integrity, and complexity. Source-synchronous systems offer simplicity but require additional clock routing and tight timing constraints. Embedded clock systems reduce pin count and improve scalability, but require robust clock recovery circuits and suitable line encoding.

At high data rates, even small timing errors can result in sampling errors. As such, accurate control of edge rates, jitter, and skew becomes increasingly important. Regardless of the clocking scheme, system-level considerations such as trace impedance, return paths, and reference ground stability are critical for reliable operation.

3.7 Interrupts

Interrupts provide a mechanism for notifying the CPU about asynchronous events without constantly polling the peripherals.

3.7.1 Interrupt Signaling

Interrupts have traditionally been signaled using separate wires by utilizing level-triggered or edge-triggered signaling schemes. A more modern alternative, used in, for example, PCI Express, is the Message Signaled Interrupt (MSI) mechanism.

Level-Triggered Interrupts

In a level-triggered interrupt scheme, an interrupt is signaled by holding a dedicated signal line asserted. The interrupt line remains asserted until the peripheral de-asserts it, typically after the host device has acknowledged and handled the interrupt. The level-triggered interrupt scheme is simple and reliable. Missed interrupts are unlikely, as the asserted condition remains active until cleared. Multiple peripherals can use the same interrupt line via wired-OR configurations, typically implemented using open-drain outputs with a pull-up resistor on the shared line. However, failure to clear the interrupt condition, for example due to peripheral malfunction, may lead to system lockup as the interrupt is constantly being asserted and thus serviced.

Edge-Triggered Interrupts

Edge-triggered interrupts are activated by a signal transition. The interrupt is latched internally by the controller or CPU, and the line may return to its idle state before being serviced. This scheme is efficient for high-speed interrupt signaling and has a lower risk of system lockup. Unlike level-triggered schemes, the edge is momentary. If the edge occurs while global interrupts are disabled, for example

during the servicing of another interrupt, or if not latched externally, the event may be lost.

Message-Signaled Interrupts

MSI is a significantly different scheme, where the dedicated interrupt lines are replaced by messages embedded in other bus traffic. Depending on the bus, MSI interrupts may be signaled by performing a memory write to a predefined address or by sending a special *interrupt request* packet on the bus. The interrupt controller then receives the packet or intercepts the write operation to receive the interrupt from the peripheral.

Advantages of MSI over the above schemes include scalability and reduced signal count as no dedicated lines are required. However, the system requires more complex interrupt controllers.

3.7.2 Chained Interrupts

In simple embedded systems, such as those based on an MCU, all interrupt sources are typically wired directly to a single interrupt controller. In a more complex system, however, multiple interrupt controllers can be connected in a chain, where upstream interrupt controllers essentially merge interrupts coming from downstream devices. This, of course, requires software support - drivers or interrupt domain handlers for the downstream interrupt controllers - to enable the host to identify the device from which the interrupt originated from. In Linux-based systems, chained interrupt controllers are typically exposed as part of the device tree, and handled by drivers that implement the `irqchip` or `irqdomain` framework.

Interrupt chaining is quite common in FPGA based designs, where a single interrupt line from the FPGA is connected to the host. The FPGA design then incorporates an interrupt controller, such as Xilinx AXI Interrupt Controller, to which

the interrupt lines from the other Intellectual Property (IP) blocks are connected. When an IP core triggers an interrupt, the FPGA-local interrupt controller asserts its output line, which is connected to the host's interrupt input. The host then queries the FPGA's interrupt controller, typically via a memory-mapped register interface, to identify and service the originating source.

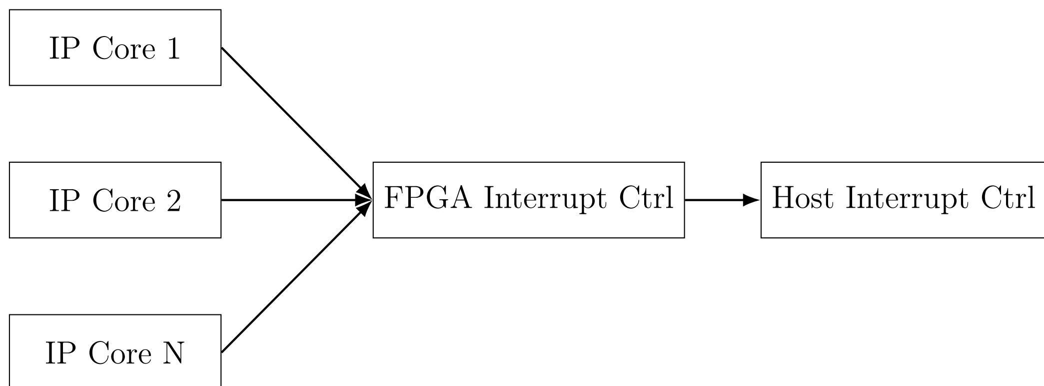

Figure 3.5: Chained Interrupts on an FPGA

4 Linux Integration

4.1 Device Tree

The Linux kernel uses the Device Tree (DT)[16] as a way to describe the static hardware layout of the system. This is particularly useful in embedded systems, where the majority of hardware components are known in advance and do not change during runtime, as opposed to personal computers where plug-and-play devices, such as disk drives and USB devices, are more common. However, even in such systems, device trees may be used to describe static hardware such as USB controllers. Typically, the device tree is either built in to the kernel during compilation or passed by the bootloader at boot time.

For this modular expansion system, the device tree describes the static system architecture, including the hardware on the main board, such as network interfaces, and the expansion bus controller. The I/O cards may be described either statically in the DT or dynamically added at runtime by a bus driver.

The use of device tree overlays allows the system to insert new device nodes during runtime, enabling support for plug-in modules or late hardware discovery. This flexibility is a key feature in supporting a modular hardware architecture, as not every combination of expansion cards need to be defined in a device tree.

4.2 Device Drivers

Device drivers [17] in Linux follow a standard model designed to separate hardware-specific logic from the general-purpose kernel infrastructure. Each device is represented by a platform device data structure in the kernel, with a corresponding platform driver responsible for interacting with the hardware.

The driver layer translates low-level hardware operations into standard Linux interfaces, such as file operations or kernel subsystems, including SPI and General Purpose Input and Output (GPIO) subsystems.

4.2.1 User-Space interfaces

The most common device types in the Linux kernel are character devices, block devices, and network devices. Each device type has an interface through which it can be accessed from user space.

Typically, devices are defined as character devices. They allow applications to open a device node (e.g., `/dev/tty0` for accessing a serial port) and perform I/O operations on it in a byte-stream fashion; block devices, such as disk drives, are accessed in blocks of data - no byte operations are supported.

Read and Write

Basic I/O with a character device is supported using standard `read()` and `write()` system calls. For example, a GPIO controller might allow reading a byte representing pin states, or writing a value to control outputs.

This model is simple and integrates easily with common tools such as `cat` and `dd` or custom applications using POSIX functions `read()` and `write()`.

IOCTL interface

More advanced hardware control is commonly implemented through Input/Output Control (IOCTL) calls. These allow structured commands to be passed between user space and the kernel driver, for example, to configure sampling rates, set modes, or retrieve status information.

The driver defines and enumerates a set of functions. The `ioctl()` system call is then used to execute these functions by providing the device node, function number, and possible arguments in the `ioctl()` call.

4.3 Bus Drivers

A bus driver is a special kernel driver that manages a hierarchy of devices. It abstracts the underlying transport or interconnect mechanism and provides device discovery, enumeration, and child device registration.

The bus driver is responsible for detecting which child devices are present, retrieving identifying information, and instantiating child devices within the kernel. This approach is used in established Linux subsystems such as PCI, I2C, or SPI, where the bus manages devices connected to it.

A simple way to provide configuration data to the child driver is by filling in a custom data structure and passing it to the driver via `platform_device_add_data()`. This means the child driver depends on the specific bus driver to supply the correct configuration data; usually this is not a problem, as a device is usually compatible with only one bus.

When the device is implemented on an FPGA, it may be reachable via different buses depending on system architecture. For example, an SoC might use AXI if the FPGA and CPU are on the same chip, or PCIe if the FPGA is on an external board. The FPGA device could even sit behind a PCIe-to-AXI bridge.

To make the same driver work across different bus implementations, dynamic modification of the device tree can be useful. This way, the device could be either statically defined in the device tree under whichever bus it is connected to, or dynamically by the bus driver: the bus driver alters the device tree based on devices found during bus enumeration.

4.3.1 Dynamic Device Tree

In systems where hardware devices may vary between installations or change over time, dynamic hardware description becomes important. Linux supports runtime modification of the device tree through overlays, which can be used to add or remove nodes representing new devices. Overlays can be applied from kernel code by creating an `of_changeset` using `of_changeset_*()` functions and then calling `of_changeset_apply()`. Another way is to use user-space tools via `configfs`.

A dynamic device tree approach allows a bus driver to construct and apply overlays based on runtime device detection, rather than relying on static declarations in the base device tree. This enables plug-and-play-like behavior and reduces the need to rebuild firmware for each hardware variant.

By utilizing device tree overlays and dynamic device discovery, the system can adapt to a wide range of hardware configurations with minimal software changes. This enables driver reusability, improves system modularity, and reduces maintenance effort.

4.4 FPGA Configuration

FPGA devices are usually based on Static Random Access Memory (SRAM) technology, meaning the logic configuration is volatile and must be loaded into the device at every power-up or reset. This configuration is defined by a binary bitstream, which

programs the logic resources, interconnects, and I/O behavior. The configuration process is a critical part of any system design using FPGAs, as the device remains non-functional until it is configured.

Because the configuration is not retained after power loss, a valid bitstream must be available at boot time. The bitstream can be loaded from an external memory, such as SPI flash, over a physical interface, e.g. Joint Test Action Group (JTAG), or via a host processor. Some FPGA families, such as Lattice Semiconductor ICE40, include an internal flash memory that can be used to store the bitstream; the device automatically loads the flash contents at boot time [18].

4.4.1 Configuration Modes in Xilinx Devices

The Xilinx 7-Series family supports several configuration modes [19], allowing designers to balance speed, convenience, and board complexity.

JTAG Configuration

The JTAG interface, standardized as IEEE 1149.1, is a common choice during development and debugging. It requires an external programmer or debugger device to load the bitstream into the target device using the `TDI`, `TDO`, `TCK`, and `TMS` pins. Configuration via JTAG is volatile, meaning the bitstream is lost on power-down. However, it supports daisy-chaining and interactive debugging through Vivado or third-party tools.

JTAG is simple to use during development and requires no external memory. However, it is slower than other configuration modes and requires physical access to the device.

Slave Serial Configuration

In this mode, the FPGA acts as a slave device, and a host processor or microcontroller provides the configuration data via a serial connection. This method is effective in embedded systems where an existing processor already manages firmware and storage. It allows full control over configuration and does not require a dedicated configuration memory. Field updates are straightforward, requiring only an update to the bitstream file managed by the host system.

The interface consists of `DIN` (data in), `CCLK` (configuration clock), `PROGRAM_B`, `INIT_B` and `DONE`. Pulsing `PROGRAM_B` low triggers a configuration cycle. The cycle begins with the FPGA pulling the `INIT_B` signal low to indicate clearing of the previous configuration. When `INIT_B` returns high, the device is ready to accept configuration data. Data on `DIN` is latched at rising edges of `CCLK`.

When the device has successfully received and verified the bitstream, it releases the open-drain `DONE` signal to indicate that configuration is complete. When `DONE` goes high, the device finishes the boot process and becomes operational. Multiple devices may share a common `DONE` net, ensuring that system initialization proceeds only after all devices have completed configuration successfully

The slave serial interface is quite SPI-like, and is thus rather easy to work with from Linux – setting and reading GPIOs for the control signals and using the SPI framework for data can be done from the userspace. Alternatively, the `fpga-manager` subsystem can be used – or a special driver can be developed.

Master SPI Configuration

The most common configuration mode in production is Master SPI, where the FPGA autonomously loads its bitstream from an external SPI flash device. This includes support for single, dual, or quad SPI modes for improved speed. The FPGA drives `CCLK` and `CS_B` (chip select), and reads data from the flash using the appropriate

data pins, depending on the chosen SPI mode. The `PROGRAM_B`, `INIT_B` and `DONE` signals are used as above.

While this configuration mode enables the FPGA to boot on its own, field updates can be difficult to perform. Updating the flash contents either requires extra on-FPGA logic or a host system connection to the flash. Additionally, any uncaught bit errors or other memory issues, for example due to power loss during programming, may render the system unusable. A widely used method is to store two bitstreams on the flash device, one active and one back-up, with the FPGA retrying with the other bitstream if loading the first one fails. Additionally, JTAG serves as the ultimate fallback if reading the flash fails.

4.5 MCU Firmware Upgrade

MCUs are often used in distributed embedded systems to implement I/O, sensor processing, or protocol handling, particularly in scenarios where little or no parallelism is required. These devices typically operate semi-independently with no external memory, and require a reliable method to receive firmware upgrades, both during manufacturing and in the field.

The firmware on an MCU is typically stored in non-volatile, on-device memory. Multiple programming and boot modes are supported, depending on the device family. In addition to an external programming interface, such as JTAG, a built-in bootloader is often present in Read-Only Memory (ROM) or protected flash and provides a standard upgrade mechanism over Universal Asynchronous Receiver/-Transmitter (UART), USB, or SPI [20]. This is useful for recovery and initial provisioning. For more advanced scenarios, a user-defined In-Application Programming (IAP) system allows the MCU to be updated while running its main firmware, often using a custom protocol and integrating integrity checks or version validation.

To perform a firmware upgrade on an MCU, the host first makes the target

enter the bootloader or a user-defined IAP mode. The method for entering firmware upgrade mode varies by device family. In addition to software-controlled entry, some devices use dedicated hardware pins - for example, STM32 series devices use a `BOOT0` pin, while ATmega devices often rely on UART activity during reset. Once the firmware upgrade mode has been entered, the host begins transmitting the firmware image over the chosen communication link, either using the bootloader's protocol or a custom protocol if IAP is used. After successfully verifying the upgrade, the device is reset to run the new firmware.

To ensure reliability, firmware updates typically include a validation step using CRC or digital signatures. Some systems implement a fallback mechanism with dual firmware partitions or a persistent bootloader, allowing recovery if the update fails or results in a non-bootable state.

From a Linux perspective, ready-made tools such as `avrdude`, `OpenOCD`, and `st-flash` are available for programming MCUs with bootloaders. In the case of a custom IAP system, a tool generally has to be developed to interface with the MCU.

5 Prototype Construction

To assess the feasibility of the modular system as outlined in the previous chapters, a prototype was constructed using an Avnet ZedBoard Zynq-7000 development kit as a base. The ZedBoard hosts an AMD Zynq-7020 chip that includes two general-purpose ARM CPU cores, a set of peripherals, and an Artix-7 class FPGA, all interconnected via AXI4 buses. The board is compatible with Linux and comes with a set of tools for developing a Yocto Linux distribution together with hardware and FPGA configuration data generated using AMD Vivado. The bootloader programs the FPGA fabric automatically before Linux starts, enabling on-FPGA implementation of peripherals, such as Ethernet controllers, that must be available at boot time.

An expansion card prototype based on a low-cost AMD Spartan-7 FPGA was designed and manufactured. The card provides a range of analog and digital input and output channels. To connect the expansion cards to the ZedBoard and support additional functions, an adapter board was developed. The adapter connects to the ZedBoard via an FPGA Mezzanine Card (FMC) connector and routes expansion bus signals from the ZedBoard to the expansion cards. It also includes power supply circuitry that accepts external power and distributes it to the expansion cards. In addition, the adapter hosts synchronization and Ethernet interface electronics, which are connected to the ZedBoard through the same FMC interface. Together with the adapter, the ZedBoard forms the module core or main board outlined in

Section 2.2.1.

To facilitate communication between the Zynq SoC and the expansion cards, a custom serial interface was developed. The interface implements an AXI4-Lite bridge, making the expansion cards visible in the host's address space as if they were implemented on the main FPGA fabric on the Zynq. AXI bridging was selected to maintain compatibility with existing AXI4-Lite-based IP cores and their corresponding device drivers. Depending on system requirements, any of the protocols described in Chapter 3 could be used instead.

The AXI bridge employs a hybrid topology: logically, it is a star, where each slave communicates directly with the master, but physically, the devices are daisy-chained to reduce system complexity. The logical star topology was selected to simplify the communication protocol; point-to-point links eliminate the need for bus arbitration, relaying, and packet routing, since each device has a dedicated link to the master. The physical bus signals are routed through the intermediate slave devices in the chain. While this restricts the number of supported expansion cards and requires additional connector pins, the main board only needs to provide connectors for one card – no large carrier board is needed.

A comparable commercial solution is the AMD *AXI Chip2Chip* IP Core [21]. However, it only supports Aurora or parallel LVDS links of at least 32 data pairs for performing the data transfers. Parallel links are problematic for many reasons, including timing and pin count. On the other hand, Aurora requires transceivers [13], which substantially increases the FPGA cost. The AXI Bridge protocol described here is based on traditional `IOSERDES` primitives found on all fairly recent Xilinx/AMD FPGA devices and requires only two data pairs. An `IOSERDES` primitive can function either as a serializer, converting parallel data to a serial stream, or as a deserializer, converting serial data back to parallel format.

In addition to the AXIL bridge, an SPI bus is available for accessing EEPROM

devices on the expansion cards. A card's EEPROM provides calibration data and identification information for automatic device discovery by Linux. An expansion card based on an MCU would use the SPI bus for communication with the host using a card-specific command interface. However, an MCU-based expansion card was not included in this prototype.

5.1 Hardware Description

At the physical level, the AXIL bridge is implemented using a set of LVDS pairs, including a shared reference clock and a single data lane per card. A data lane consists of two differential pairs, one for transmitting and one for receiving. Additional signals include time-based and engine-angle-based synchronization, JTAG, and an AMD Slave Serial programming interface, together with reset power-good signals. A simplified schematic is shown in Figure 5.1.

The cards are designed to be physically daisy-chained, with a maximum of four cards supported. This is realized by routing the extra bridge lanes through each card, with the incoming lane n begin routed out as lane $(n-1)$. Similar stepping is used for SPI chip selects and the `PROGRAM_B` signals of the slave serial programming interface.

Power supply is implemented as outlined in Chapter 2.2.3. An upstream card provides 24 and 5 volts to downstream cards, which perform local regulation based on the card's needs.

5.2 AXI Bridge Implementation

The AXIL bridge IP core is designed to transport memory-mapped register transactions between the main FPGA and the expansion cards. The protocol stack is implemented in SystemVerilog and split across logically separated modules to enhance

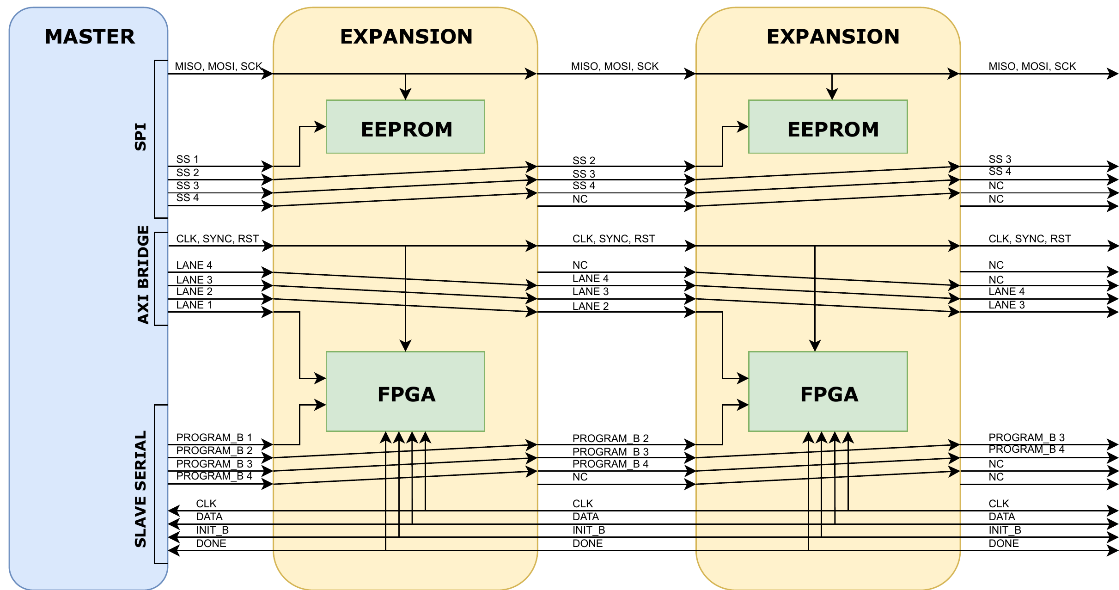

Figure 5.1: Simplified schematic of the system with two expansion cards shown. The downstream interface on an expansion card is identical to the master's interface.

reusability and portability. Both ends of the link support reliable communication using a byte-serial LVDS connection, packet framing, CRC error checking, and an ARQ mechanism for fault tolerance.

The bridge is composed of two halves: the *master*, which provides an AXIL slave interface to the control system, and the *slave*, which initiates AXIL transactions on a AXIL master interface on the expansion card FPGA. The master module receives AXIL requests, encapsulates them into packets, serializes, and finally transmits them over an LVDS link. The slave module decodes these requests and performs the actual AXIL transaction on the expansion card. Responses follow the reverse direction: the slave captures AXIL response data, encapsulates it, and transmits it back to the master for delivery to the control system.

A block diagram depicting the FPGA designs of both main board and expansion cards is shown in Figure 5.2. The bridge makes the expansion cards available in the host's address space as if they were implemented on the main FPGA, as shown in Figure 5.3. Together, the master and slave modules form a tightly-coupled, fault-

tolerant communication link, enabling remote register access across boards via serial LVDS signaling.

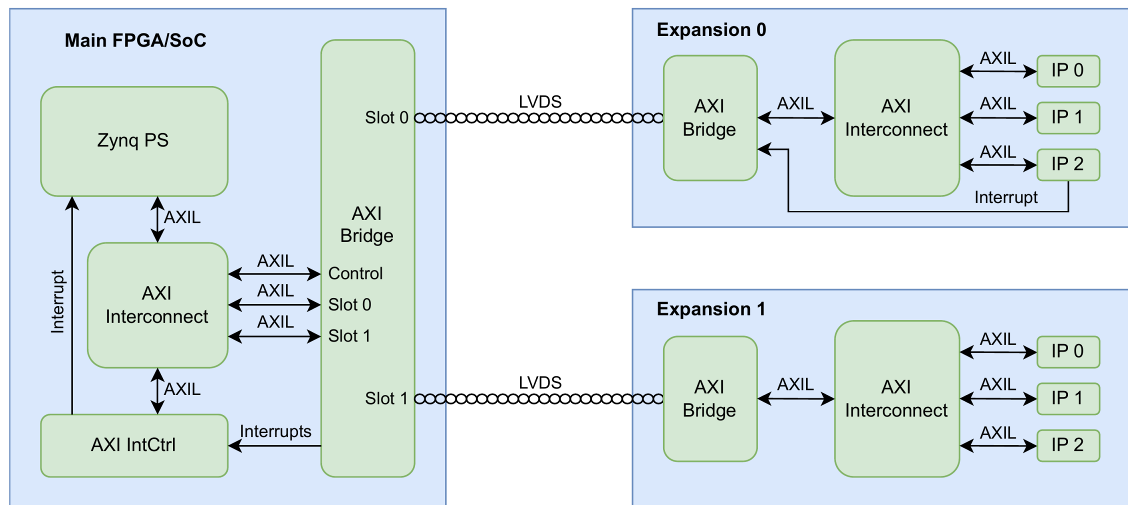

Figure 5.2: High-level FPGA designs of the main board and expansion cards

5.2.1 Protocol Stack Overview

Master - AXI4-Lite Slave Interface

The master module implements an AXIL slave interface, through which it accepts memory-mapped AXIL transactions from the system using *Read Address*, *Write Address*, and *Write Data* channels. The read and write requests are packaged and stored into a transmission queue, from which the ARQ layer fetches packets and transmits them over an LVDS link to the slave device. Each packet consists of a two-byte header, variable length payload, and a CRC-16 checksum. Packet structure is described in Listing 1.

Responses and interrupt status updates from the slave are received by the ARQ layer and stored in a queue. The master then fetches a packet from the queue and communicates the response to the host over the AXIL interface using *Read Data* and *Write Response* channels. In the case of an interrupt update packet, the status of the interrupt lines is set according to the packet's contents.

Listing 1 AXI bridge packet structure

```
typedef enum logic [2:0] {
    TYPE_READ_REQ   = 3'b000,
    TYPE_READ_RESP  = 3'b001,
    TYPE_WRITE_REQ  = 3'b010,
    TYPE_WRITE_RESP = 3'b011,
    TYPE_IRQ_UPDATE = 3'b100,
    TYPE_ACK_ONLY   = 3'b101,
    TYPE_SYN        = 3'b110
} packet_type_t;

typedef union packed {
    struct packed {
        logic [3:0]      len;      // payload length
        logic [3:0]      nlen;     // complement of payload length
        logic            ack;      // ack flag
        logic            ackn;     // ack number
        logic            seqn;     // sequence number
        logic [1:0]      pad;      // padding
        packet_type_t    typ;      // payload type
        logic [8:0][7:0] payload;  // payload (variable length)
    } fields;
    logic [0:2+8][7:0] raw;
} packet_t;
```

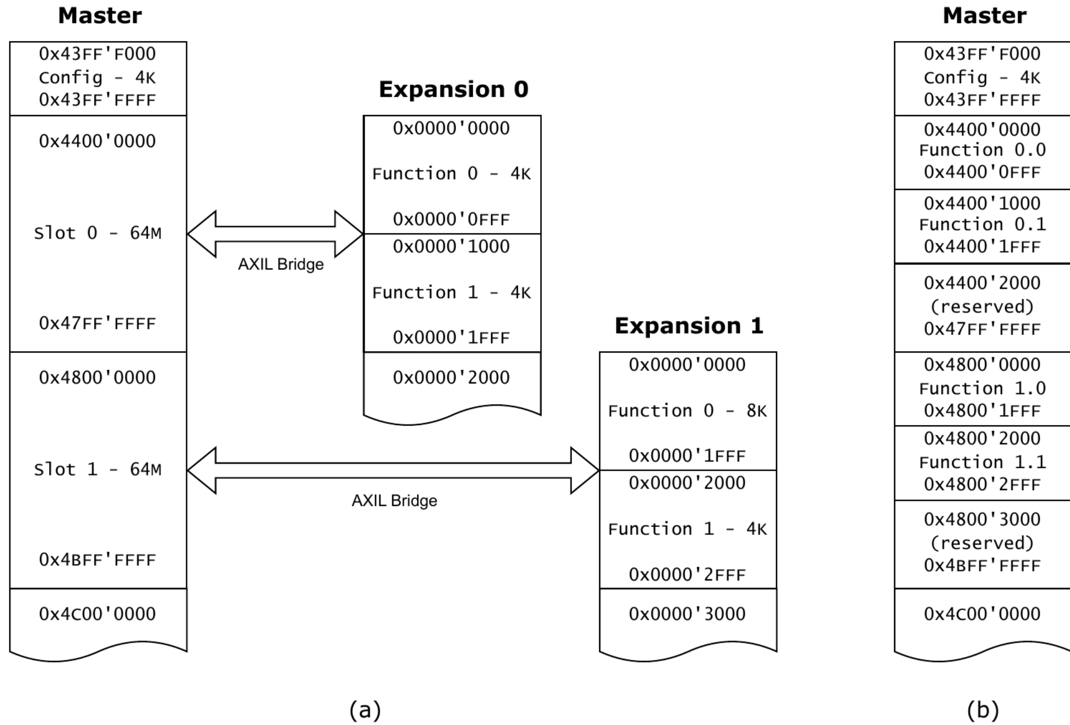

Figure 5.3: (a) Logical bridge architecture (b) The resulting address mappings

Slave - AXI4-Lite Master Interface

The slave module implements an AXIL master interface. The ARQ layer receives read and write requests through an LVDS link and stores them to a receive queue, from which the slave fetches packets and initiates the requested AXIL transactions. Responses are accepted through the AXIL master interface, packaged, and stored to a transmission queue. The ARQ layer then communicates the responses to the master device over the LVDS link.

The slave module provides a set of interrupt input lines. The status of the interrupt lines is communicated to the master using a special interrupt status update packet. Interrupt signaling is unidirectional; the host system must clear interrupt status explicitly via the AXIL interface.

ARQ Layer

The ARQ layer provides reliable communication over the LVDS link. As AXIL does not support outstanding or burst transactions, using a stop-and-wait protocol for the ARQ layer has only a slight impact on throughput. This simplifies ARQ logic in comparison to more advanced schemes such as go-back-n or selective-repeat ARQ.

In this protocol, the sender transmits a single packet and retains it in a retransmission buffer until it receives a valid Acknowledgment (ACK) from the receiver. Each packet includes a sequence number, which is incremented between successive transmissions. This sequence number allows the receiver to distinguish between new packets and retransmissions. Upon receiving a valid packet, the receiver responds with an ACK containing the matching sequence number and delivers the packet's contents to the upper layer. If a duplicate packet is received, typically due to the original ACK being lost, the receiver ignores the data and reissues the ACK.

Acknowledgments are transmitted immediately upon successful reception of a packet, either as part of another packet or as a small standalone packet if the transmission queue is empty. If the sender does not receive an ACK within a certain timeout interval it assumes the packet or its acknowledgment was lost and retransmits the original packet. This retry process continues until the correct acknowledgment is received.

Both the master and slave sides of the link implement identical ARQ logic, allowing full-duplex operation with independent transmit and receive channels. Each side manages its own state machines and timeout counters, ensuring symmetrical behavior.

If repeated timeouts occur without receiving a valid acknowledgment, the ARQ logic triggers a soft reset of the link. This causes the physical layer to re-enter its synchronization phase, suspending packet transmission until link integrity is reestablished. This ensures that persistent errors, such as clock skew or signal degra-

dation, do not lead to deadlock or inconsistent state between the two ends of the link.

The stop-and-wait ARQ design prioritizes simplicity and reliability over throughput efficiency. While it imposes a throughput ceiling due to its strictly sequential nature, it guarantees in-order, lossless delivery with minimal protocol complexity, making it an ideal fit for memory-mapped register access over a serialized physical medium.

Link Layer

The byte stream is serialized and transmitted over an LVDS link using a Double Data Rate (DDR) transmitter based on an `OSERDESE2` primitive found in most AMD FPGA families. The transmitter emits comma characters to support synchronization and framing. The serialized data is deserialized using `ISERDESE2` primitives with a synchronizer based on `IDELAY` elements. The deserializer implementation is heavily influenced by [22]. The LVDS link operates at up to 800 Mbps, approaching the maximum speed of an AMD Spartan 7 FPGA.

When the link is established, a synchronization sequence takes place. A presynchronization comma character (`0xBC`) is continuously transmitted by both transmitters. The receivers start by aligning the data and clock signals by adjusting a set of delay elements used by the input primitives. This continues until the received data is not changing, indicating that the clock skew has been eliminated. The second part of the synchronization sequence then begins. If the received byte does not match the expected comma character, bit slipping is used to adjust byte alignment until the received data matches the comma character. When this state is reached, the comma character changes to `0xDC` to indicate to the other party that the receiver is synchronized. Once both receivers are synchronized, the signal `locked` is asserted, which enables the upper layers of the protocol stack.

When the link is idle, the same comma character, `0xDC`, is continuously transmitted to keep the link synchronized. In case of a bit error the synchronization sequence is reinitiated, during which transmission of new packets is not possible. Beginning of a data frame is indicated by a start-of-frame (SOF) comma character (`0x1C`), followed by the frame contents. Once the frame is over, the transmitter returns to transmitting the idle comma character.

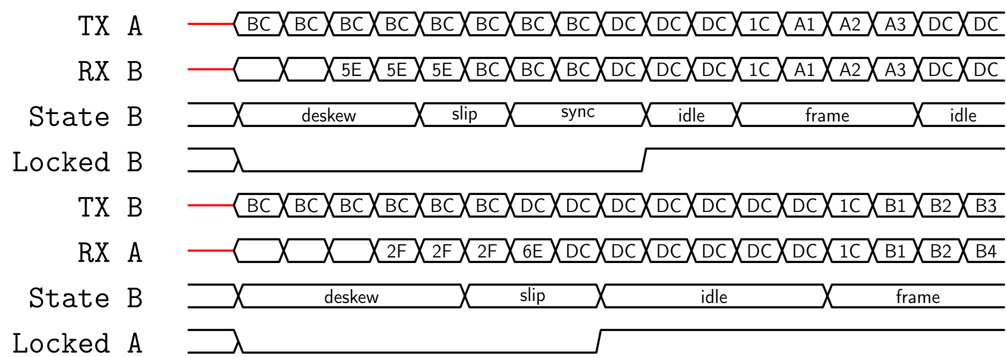

Figure 5.4: Link-layer synchronization sequence between transceivers A and B

5.2.2 Expansion Bus Control Interface

The AXIL bridge IP core also provides an additional AXIL interface for controlling the IP core and the additional expansion bus signals, such as reset, and the signals of the slave serial programming interface. The programming interface is realized by two registers, one for setting the control signals and another for writing to a queue from which the bitstream is clocked out.

5.3 Linux Integration

Linux integration of the expansion card system is based on a bus driver that uses the dynamic device tree facilities of the kernel to enable runtime device discovery. This approach was selected to keep the device drivers usable with static device trees,

retaining support for devices on the main board in addition to expansion cards.

The bus controller, expansion card EEPROMs, and bus slots are defined in the device tree. Slot contents are filled in by the bus driver based on the information stored on the EEPROM of each expansion card, including hardware identification, hardware version, and a list of *functions* present on the card, together with accompanying calibration data. A function descriptor contains information of the compatible device driver, MMIO region size and offset, and possible interrupt signal information.

5.3.1 Bus Driver

The Linux bus driver interfaces with the AXI4-Lite bridge and handles communication management, expansion card EEPROM parsing, expansion card firmware programming, and dynamic device creation. The driver is implemented as a platform driver and is configured by a device tree node shown in Listing 2, together with the EEPROM device tree nodes.

Bus probing is done by the IOCTL function `AXIL_SERDES_CTL_PROBE_SLOT`. It begins by reading the contents of the corresponding EEPROM, followed by adding the found card functions to the device tree. Finally, new platform devices are allocated and initialized, enabling the kernel to bind drivers to devices based on the device tree modifications.

EEPROM Parsing

The EEPROMs are parsed via the `nvmem` subsystem, a kernel subsystem meant for handling configuration data from devices. A memory device is partitioned into *cells*, defined in the device tree, which can then be accessed based on device tree definitions using `of_nvmem_cell_*()` functions. A cell is read using `nvmem_cell_read()`, which returns a pointer to the read data on success. The read data is validated and

Listing 2 Bus driver: device tree nodes

```
axil_serdes_master_0: axil_serdes_master@43fff000 {
    compatible = "wartsila,axil-serdes-master";
    #address-cells = <1>;
    #size-cells = <1>;
    reg = <0x43fff000 0x400>;
    ranges = <0x44000000 0x44000000 0x08000000>;
    num-slots = <2>;
    slot@44000000 {
        compatible = "simple-bus";
        #address-cells = <1>;
        #size-cells = <1>;
        nvmem-cells = <&cell_slot_funcs0>;
        nvmem-cell-names = "slot_config";
        slot = <0>;
        reg = <0x44000000 0x04000000>;
        ranges = <0 0x44000000 0x04000000>;
    };
    slot@48000000 {
        /* ... */
        slot = <1>;
        reg = <0x48000000 0x4000000>;
        ranges = <0 0x48000000 0x4000000>;
    };
};

slot0eeprom: eeprom@0 {
    compatible = "atmel,at25";
    reg = <0>;
    /* ... */
    nvmem-layout {
        compatible = "fixed-layout";
        cell_slot_funcs0: cell@0 {
            reg = <0 1480>;
        };
        cell_slot_calib0: cell@1480 {
            reg = <1480 31288>;
        };
    };
};

slot1eeprom: eeprom@1 {
   /* ... */
};
```

then parsed into a set of `driver_info` structures, as outlined in Listing 3.

Listing 3 Bus driver: Reading EEPROM

```
struct eeprom_data {
    char magic[4];
    u8 type, version, nfunctions, pad;
    struct function {
        char compat[32];
        u32 offs, len;
        u8 num_irqs, pad;
        u16 calib_start, calib_len;
    } __attribute__((packed)) funcs[32];
} __attribute__((packed));

struct nvmem_cell *cell;
cell = of_nvmem_cell_get(slot_node, "slot_config");

struct eeprom_data *data = nvmem_cell_read(cell, &len);
nvmem_cell_put(cell);

if (memcmp(data->magic, "UNIC", 4) != 0) {
    dev_warn(dev, "Slot %d: invalid eeprom magic\n", slot);
    ret = -EINVAL;
    goto err;
}

struct driver_info *info = kzalloc( /* ... */ );
for (i = 0; i < data->nfunctions; ++i) {
  info[i] = /* from eeprom_data */;
}
```

Device Creation

Device creation consists of three parts as shown in Listing 4. The new `platform_device` is allocated via `platform_device_alloc()`. Then, a new device tree node is created and associated with the `platform_device`. Finally, the platform device is given to the kernel via `platform_device_add()`, enabling driver binding via standard platform driver matching based on `compatible` strings. This process is repeated for each function found from the card's EEPROM.

The device tree node is created by using the `of_changeset` kernel framework as outlined in Listing 5. First, a new `of_changeset` is initialized. Then, a new node is created as part of the changeset, followed by a set of properties describing the device. Node name and property values are taken from the `driver_info` structures read from the EEPROM earlier. Once created, the `changeset` is applied to the live device tree via `of_changeset_apply()`. An example of a created function node is shown in Listing 6.

Listing 4 Bus driver: Creating devices

```
struct platform_device *child;
child = platform_device_alloc(infos[i].compat, id);
if (!child) {
    ret = -ENOMEM;
    break;
}

child->dev.parent = dev;
child->dev.of_node = create_func_node(dev, slot_np, &infos[i]);
if (!child->dev.of_node) {
    ret = -EINVAL;
    platform_device_put(child);
    break;
}

ret = platform_device_add(child);
if (ret) {
    of_node_put(child->dev.of_node);
    platform_device_put(child);
    break;
}
```

Card Programming

The driver provides support for programming an expansion card's FPGA over the slave serial interface. Calling the IOCTL function `AXIL_SERDES_CTL_PROGRAM`, with the desired slot number as the argument, sets the corresponding slot to programming mode by pulsing the corresponding `PROGRAM_B`. The FPGA responds by pulsing

Listing 5 Bus driver: Device tree overlay, error handling omitted

```
struct device_node *create_func_node(
    struct device *dev,
    struct device_node *parent,
    struct driver_info *info) {

    char name[16];
    snprintf(name, sizeof(name), "func@%d", info->func);

    struct of_changeset ocs;
    of_changeset_init(&ocs);

    struct device_node* np;
    np = of_changeset_create_node(&ocs, parent, name);

    of_changeset_add_prop_string(&ocs, np, "compatible", info->compat);

    u32 reg[2] = {info->offs, info->len};
    ret = of_changeset_add_prop_u32_array(&ocs, np, "reg", reg, 2);

    /* ... */

    of_changeset_apply(&ocs);
    return np;
}
```

Listing 6 Bus driver: Slot node after probing a function

```
slot@44000000 {
    compatible = "simple-bus";
    #address-cells = <1>;
    #size-cells = <1>;
    reg = <0x44000000 0x04000000>;
    ranges = <0 0x44000000 0x04000000>;
    slot = <0>;
    nvmem-cells = <&cell_slot_funcs0>;
    nvmem-cell-names = "slot_config";

    func@0 {
        compatible = "vendor,device";
        reg = <0x10000 0x1000>;

        nvmem-cells = <&cell_slot_calib0>;
        nvmem-cell-names = "slot_calib";
    };
};
```

`INIT_B` low to indicate start of the initialization cycle, during which the previous configuration is cleared. When `INIT_B` returns high, the FPGA is ready to receive configuration data. The IOCTL function returns when `INIT_B` has gone high, or if a timeout occurs. Once the card is set to programming mode, a bitstream can be written to it via the driver's `write()` function using a suitable user space tool, such as `dd` or `cat`.

Bus Management

In addition to the aforementioned operations, the driver exposes three more IOCTL functions for managing the bus. `AXIL_SERDES_CTL_SLV_RESET` resets the expansion cards via the `reset` signal on the bus. Likewise, `AXIL_SERDES_CTL_MSTR_RESET` resets the master side of the bus, beginning a synchronization cycle. These resets are performed after programming to bring the system into a known state before bus enumeration.

Bus status – such as link state, transmitted packet count, and retransmission count – can be queried via the IOCTL function `AXIL_SERDES_CTL_STATUS`.

User-space Utilities

Two user-space utilities were developed to support the driver, one for generating EEPROM images for expansion cards, and another for controlling the bus driver. They provide a command-line interface for executing IOCTL functions and writing bitstreams to the FPGAs. In future work, a complete helper application could be developed to read the EEPROMs, determine which expansion cards are connected, use that information to program the bitstreams to the expansion cards, and finally probe the cards.

5.4 Testing

5.4.1 Simulating the AXI Bridge IP

During development of the AXI bridge, a number of simulations were performed to assess performance of the design. Each layer of the protocol stack was first tested separately, followed by integration testing once the layers performed well on their own. The integration test emulated an AXI4-Lite master issuing transactions through the bridge to a memory block on the remote side. Both read and write transactions were exercised, and memory contents were verified to ensure correct data transfer and integrity.

To evaluate the reliability of the link under adverse conditions, bit errors were deliberately injected into the serial data stream. A baseline random error rate of 1 flipped bit per 1000 was introduced, simulating low-level noise. Additionally, burst error scenarios were tested by introducing bit errors at rates as high as 25 flipped bits per 1000, simulating transient interference or signal degradation.

These fault injection scenarios allowed for the assessment of the bridge's error detection and recovery mechanisms, ensuring that the protocol could maintain correct AXI semantics and recover gracefully from both isolated and burst errors.

5.4.2 Testing the AXI Bridge on Hardware

Following successful simulation, the AXI bridge was validated on hardware using a hardware setup consisting of a ZedBoard and an expansion card, as described above in section 5.1. In this test setup, the ZedBoard hosted the AXI bridge master, while the expansion card implemented the bridge slave and a memory-mapped block RAM.

Unlike the simulation environment, no deliberate bit errors were introduced during hardware testing. Instead, the focus was on functional verification and performance evaluation under real-world conditions. AXI4-Lite read and write transactions were initiated from user space in a Linux environment running on the ZedBoard, targeting the remote block RAM over the serial bridge link.

Performance metrics – including throughput and access latency – were measured to assess the efficiency of the link. It should be noted that all operations were performed from user space, and as such, the results are subject to variability due to Linux scheduling and system overhead.

The test confirmed correct operation of the bridge, from software-initiated I/O to remote memory access and data readback, with latency less than a microsecond and throughput of 1.3 million AXI transactions per second.

5.4.3 System Testing

To validate the full integration of the AXI4-Lite bridge in a real deployment scenario, system-level testing was conducted on the ZedBoard running Linux. This phase focused on the user-space tooling and kernel driver stack used for EEPROM configuration, FPGA programming, card detection, and device binding. Another

expansion card was added to the hardware setup to test multiple bus slots.

Each expansion card contains an onboard EEPROM used to store metadata such as device type, compatibility string, IRQ configuration, and calibration data. A custom utility, `eeprom_writer`, was used to generate this metadata in binary form. The output binary was written directly to the EEPROM exposed via sysfs:

```
root@ZedBoard:~# eeprom_writer --type=1 --version=1 \
                    --func --compat=wartsila,munic-adi --irqs=1 \
                    --offs=65536 --len=4096 --calib=calib.bin
Magic: UNIC type: 1 version: 1 functions: 1
Func 0: compat: wartsila,munic-adi offs: 10000 irqs: 1 calib: 5c8 +4

root@ZedBoard:~# cat eeprom.bin > /sys/bus/spi/devices/spi1.0/eeprom
root@ZedBoard:~# cat eeprom.bin > /sys/bus/spi/devices/spi1.1/eeprom
```

Next, the bridge control driver `axil_serdes_ctl` was loaded, initializing communication with the connected expansion cards:

```
root@ZedBoard:~# modprobe axil_serdes_ctl
axil_serdes_ctl: loading out-of-tree module taints kernel.
axil_serdes_ctl: Initializing AXIL serdes controller with 2 slots
```

FPGA programming was performed via the `serdes_test` helper application, which uses the bridge driver to transmit FPGA bitstreams to each slave device:

```
root@ZedBoard:~# serdes_test --dev=/dev/axil_serdes_ctl \
                    --bit=/etc/firmware/exp/adiado.bin \
                    --bit=/etc/firmware/exp/adiado.bin \
                    --program
axil_serdes_ctl: setting slave 0 to program mode
axil_serdes_ctl: writing 2192012 bytes
axil_serdes_ctl: setting slave 1 to program mode
axil_serdes_ctl: writing 2192012 bytes
axil_serdes_ctl: reading status registers
axil_serdes_ctl: resetting slaves
axil_serdes_ctl: resetting master
```

After programming, each slot was probed to detect connected devices:

```
root@ZedBoard:~# serdes_test --dev=/dev/axil_serdes_ctl --probe=0
axil_serdes_ctl: Probing slot 0
axil_serdes_ctl: Slot 0: adding function 0 (0) [wartsila,munic-adi]

root@ZedBoard:~# serdes_test --dev=/dev/axil_serdes_ctl --probe=1
axil_serdes_ctl: Probing slot 1
axil_serdes_ctl: Slot 1: adding function 0 (100) [wartsila,munic-adi]
```

Finally, the appropriate function driver `munic_adi` was loaded. This driver bound to each detected device and initialized their respective memory-mapped I/O regions:

```
root@ZedBoard:~# modprobe munic_adi
munic_adi: Initializing MUNIC ADI 0 with 4 channels
munic_adi: MMIO region 0x44010000 - 0x44010fff
munic_adi: Initializing MUNIC ADI 1 with 4 channels
munic_adi: MMIO region 0x48010000 - 0x48010fff
```

Upon successful binding, device nodes were created for each instance:

```
root@ZedBoard:~# ls /dev | grep adi
munic_adi0
munic_adi1
```

This end-to-end test confirms proper operation of the EEPROM writer, programming interface, driver stack, and probing mechanism, validating the system's ability to configure, program, and communicate with expansion cards dynamically and reliably.

6 Conclusions

6.1 Impact of Modularization

6.1.1 Design Flexibility and Scalability

Modularization introduces a clean separation between core control logic and peripheral I/O, making the system inherently more scalable. Expansion cards encapsulate specific functionality – analog or digital I/O, communication interfaces, etc. – which can be developed and upgraded independently. This simplifies platform adaptation to different engine models or customer-specific variants by swapping or adding cards, rather than redesigning the whole control unit.

Modularization supports field maintenance, selective upgrades, and longer product lifespans. I/O functions likely to evolve over time can be updated independently, and obsolete or failure-prone I/O modules can be retired without impacting the main control unit. System variants for different market segments can be easily created by combining I/O modules.

Each I/O module can be tested, validated, and certified in isolation, enabling faster time-to-market, and re-use of known-good test infrastructure across projects. However, this increases the importance of integration testing, especially in scenarios involving multiple cards, possibly from different vendors.

6.1.2 Subcontractor Engagement

One of the strategic benefits of modularization is the ability to delegate development work to subcontractors in a compartmentalized and controlled fashion. Requirements need to be specified per I/O module instead of the whole control unit at once, arguably leading to better requirements handling and thus better results from the subcontractor. In general, modularization reduces the surface area of subcontractor responsibilities, enables reuse, and improves the ability to negotiate scope, cost, and timelines. This flexibility enables assigning work to multiple subcontractors based on their expertise and in-house capacity, with clear deliverables and minimal coupling between tasks. Each module is self-contained, making subcontractor outputs easier to verify, validate, and integrate, in addition to providing greater opportunities for second-sourcing system components.

In contrast, a monolithic control unit may involve a single subcontractor responsible for the entire hardware and low-level software stack, where all aspects of board design, peripheral integration, firmware, and Linux drivers are developed by the subcontractor as a unified whole. While this can simplify coordination in the short term, it limits parallel development, makes it harder to reuse parts in future projects, and increases the risk of vendor lock-in.

6.1.3 Cost Implications

Modularizing the system architecture requires an initial investment in defining robust I/O module interfaces and software abstractions. However, long-term cost is reduced through code and hardware reuse, especially when engaging subcontractors for modular deliverables.

Modularization introduces additional Bill of Materials (BOM) elements, such as connectors, bus transceivers, power supply components, and extra FPGA and/or MCU devices. However, this increase in component cost can be offset by the simpler

main board and economies of scale when manufacturing common I/O modules across product families.

Modular deliverables allow for clearer contracts and milestones with subcontractors. Reduced scope makes estimating, testing, and validating subcontractor work easier, and, together with increased unit volumes, likely increases the number of subcontractors willing to take on development and manufacturing work, enabling cost reduction and quality improvements due to competition.

6.2 Research Questions Revisited

1. How can the system architecture be designed to ensure future upgrades without the need for redesign of other parts of the system?

   The architecture proposed in Chapter 2 provides a solution by modularizing the system and creating self-contained components that can be upgraded, fixed, and modified without affecting other parts of the system.

2. How can the communication be implemented to ensure efficient and reliable data transfer?

   In addition to industry standard communication interfaces outlined in Chapter 3, custom interfaces may be developed to meet the system's needs. The interface selection is a trade-off between component cost, performance, and system complexity.

3. How can signal integrity be maintained in harsh operational environments?

   Following general best practices for signal integrity provides a good basis for system design. Keeping traces and wires short, providing adequate return paths, minimizing impedance discontinuities, and preferring differential signaling is a good start, as outlined in Section 3.5. In addition to signals, power

integrity is just as important, as discussed in Section 2.2.3.

4. What error detection and correction methods should be implemented to handle potential communication failures?

   The chosen protocols should provide means for error detection and/or correction along with automatic retransmission mechanisms. As seen in Chapter 5, a simple checksum together with ARQ functionality may suffice. However, for high-throughput applications, more advanced schemes such as convolutional or turbo coding might be beneficial.

5. How does separating I/O functionality onto expansion cards impact design, validation, and manufacturing costs over the system's life cycle?

   After an initial investment in architecture development, long-term costs are likely to be reduced, as outlined in Section 6.1.

6.3 Future Work

While the current communication protocol is relatively performant and resilient, it has quite substantial overhead due to the small packet size. A better version could include a more relaxed ARQ scheme, such as Go-Back-N for improved throughput. Additionally, the effects of scatter-gather operations and transaction queuing may be worth investigating. However, such schemes automatically increase latency, although throughput is increased since more data is transferred at a time.

Error resilience has only been tested in simulation; once an actual main board has been developed, real-world error testing can be done – testing with the current setup consisting of a ZedBoard and an additional adapter board would yield results not comparable to the real system.

In addition to the custom AXI bridge developed for this prototype, other bus systems and their tradeoffs should be analyzed. An interesting implementation

would be Aurora, which provides a commercial solution to AXI bridging between FPGA devices, but requires the FPGAs to be equipped with transceivers, which increases component cost quite substantially. Finally, expansion cards based on microcontrollers should be evaluated.

References

[1] N. K. Reddy, S. Cherukuru, and V. Vani, *High-performance and energy-efficient fault tolerance fpga-to-fpga communication*, 2021. DOI: 10.21203/rs.3.rs-741169/v1.

[2] W. An, X. Jin, X. Du, and S. Guo, "A flexible fpga-to-fpga communication system", in *2016 18th International Conference on Advanced Communication Technology (ICACT)*, 2016, pp. 586–591. DOI: 10.1109/ICACT.2016.7423482.

[3] DNV GL AS, *DNVGL-CG-0339 Environmental test specification for electrical, electronic and programmable equipment and systems*, 2019.

[4] Altera Corporation, *AN 574: Printed Circuit Board (PCB) Power Delivery Network (PDN) Design Methodology*, 2009.

[5] Motorola Inc, *Using the Serial Peripheral Interface to Communicate Between Multiple Microcomputers*, 1987.

[6] NXP Semiconductors N.V., *I2C-bus specification and user manual*, 2014.

[7] International Organization for Standardization, *ISO 11898-1:2024: Road vehicles — Controller area network (CAN) — Part 1: Data link layer and physical coding sublayer*, 2024.

[8] PCI-SIG, *PCI Express Base Specification Revision 5.0 Version 1.0*, 2019.

[9] ARM Holdings plc, *AMBA AXI and ACE protocol specification*, 2013.

[10] Micron Technology Inc, *Micron Serial NOR Flash Memory MT25QL256ABA*, 2014.

[11] T. Kugelstadt, "Extending the SPI bus for long-distance communication", in *Analog Applications Journal, Fourth Quarter 2011*, Texas Instruments Inc, 2011, pp. 16–20.

[12] M. Balch, *Complete Digital Design: A Comprehensive Guide to Digital Electronics and Computer System Architecture*. McGraw-Hill, 2003, ISBN: 978-0-07-140927-8.

[13] Advanced Micro Devices, Inc., *Aurora 8B/10B LogiCORE IP Product Guide (PG046)*, 2023.

[14] Telecommunications Industry Association, *Electrical Characteristics of Low Voltage Differential Signaling (LVDS) Interface Circuits*, 2001.

[15] Institute of Electrical and Electronics Engineers, Inc, *IEEE Standard for Low-Voltage Differential Signals (LVDS) for Scalable Coherent Interface (SCI)*, 1996.

[16] The kernel development community, *Open Firmware and Devicetree*, `https://www.kernel.org/doc/html/v6.6/devicetree/index.html`, [Accessed 12-05-2025], 2023.

[17] J. Corbet, G. Kroah-Hartman, and A. Rubini, *Linux device drivers*, 3rd ed. O'Reilly, 2005, ISBN: 0-596-00590-3.

[18] Lattice Semiconductor, *iCE40 Programming and Configuration*, 2022.

[19] Advanced Micro Devices, Inc., *7 Series FPGAs Configuration User Guide (UG470)*, 2023.

[20] ST Microelectronics, *STM32 microcontroller system memory boot mode*, 2025.

[21] Advanced Micro Devices, Inc., *AXI Chip2Chip v5.0 LogiCORE IP Product Guide (PG067)*, 2022.

[22] M. Defossez and N. Sawyer, “LVDS Source Synchronous DDR Deserialization (up to 1,600 Mb/s)”, in *Application Note: 7 Series FPGAs*, Xilinx, Inc., 2016.