Delft University of Technology
Master of Science Thesis in Embedded Systems

FPGA-Based Design for S-Transform-Based Fault Detection Algorithm with RTDS Integration

Yujie Ye

TU Delft Delft University of Technology

FPGA-Based Design for S-Transform-Based Fault Detection Algorithm with RTDS Integration

Master of Science Thesis in Embedded Systems

Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands

Yujie Ye

18th June 2025

Author
Yujie Ye
Title
FPGA-Based Design for S-Transform-Based Fault Detection Algorithm with RTDS Integration
MSc Presentation Date
25th June 2025

Graduation Committee

Dr.ir. Stephan Wong	Delft University of Technology
Prof. Marjan Popov	Delft University of Technology
Dr.ir. Mottaqiallah Taouil	Delft University of Technology

Abstract

With the increasing integration of renewable energy sources such as Type-3/4 wind turbines and photovoltaic systems, fault current levels in power systems have decreased, weakening the performance of traditional distance relays. To address this, the Stockwell Transform (S-Transform) based fault detection algorithm has been proposed and has proven effective in identifying fault occurrences. While previous work has implemented the S-Transform-based fault detection algorithm in the Programmable Logic (PL) of the FPGA and validated it through AMD Vivado simulation, integration into a physical FPGA board and a Real-Time Digital Simulator (RTDS) environment has not yet been achieved. This paper presents a complete hardware-software co-design in which the exiisting implementation is deployed on an FPGA board and integrated with an RTDS system. The proposed system enables real-time communication between the RTDS and the FPGA via the IEC 61850 protocol. The PL of the FPGA platform executes the fault detection algorithm, while the Processing System (PS) handles IEC 61850 protocol communication, data exchange between the PS and the PL, and interrupt handling within a bare-metal environment. Experimental results demonstrate that the entire design meets strict real-time performance and delay requirements, validating the system's suitability for high-speed fault detection in distance protection applications.

Preface

I would like to express my deepest appreciation to my supervisor, Prof. Marjan Popov, and Dr.ir. Stephan Wong for their valuable advice and patient guidance throughout this thesis project. At a time when I had no clear idea for my thesis, Stephan kindly offered me a range of options across both research-oriented and engineering-focused projects and guided me toward this one. He has always been supportive and attentive, offering helpful suggestions whenever I encountered challenges in making progress. Marjan supported me throughout the long duration of the thesis period, providing the necessary resources and information essential to my work. His expertise in electrical power engineering enabled me to acquire knowledge that I would not typically gain as an embedded systems student.

I would also like to express my sincere gratitude to Jose Chavezmuro for his unwavering support. He patiently helped me navigate many challenges, answering countless questions—some of which had kept me stuck for a long time. I am equally thankful to Remko Koornneef, the lab manager, assisted me in setting up the experimental environment and kindly offered guidance on working with the RTDS system.

Finally, I would like to thank all my friends and family for their care and encouragement throughout the thesis journey. Your support has meant a great deal to me and has been essential in helping me complete this work.

Yujie Ye

Delft, The Netherlands
18th June 2025

Contents

Chapter 1

Introduction

1.1 Background and Motivation

In the coming years, a significant number of coal and nuclear power plants will be decommissioned and replaced by renewable energy sources (RES). While RES integration offers environmental and sustainability benefits, it also reduces the overall system inertia and fault current levels, thereby increasing the complexity of power system operation, control, and protection. In particular, traditional distance protection schemes face challenges in accurately identifying faults under low-inertia, low-fault-current conditions. To address these issues, a novel fault detection algorithm based on the Fast Discrete Stockwell Transform (S-Transform) has been proposed [1]. This algorithm enhances the performance of distance protection by analyzing the computed S-energy across different current phases and comparing it with a threshold determined by the root mean square (RMS) values of the currents. The method demonstrates improved sensitivity and reliability in detecting faults under varying system conditions, particularly in RES-dominated networks.

The S-Transform algorithm introduced in [1] was initially developed in Real-Time Digital Simulator (RTDS) and evaluated in MATLAB, demonstrating its potential for fault detection in power systems. To enable practical deployment on commercial distance relays, a hardware implementation was subsequently proposed in [2]. This design, based on the Xilinx Zynq UltraScale+ MPSoC ZCU104 FPGA, was evaluated through AMD Vivado simulation and demonstrated the capability to compute S-energy and threshold values efficiently using fixed-point input current data. However, this implementation was limited to the Programmable Logic (PL) portion of the FPGA, lacking a complete system architecture. The Processing System (PS) side of the FPGA, which is essential for real-time deployment, was not addressed. Specifically, critical components such as communication with the RTDS, communication protocol interpretation modules, and high-throughput data streaming among the whole system were not implemented, leaving a significant gap in realizing a fully functional embedded system for distance protection.

1.2 Goals of This Work

The primary objective of this thesis is to complete the FPGA-based implementation of the S-Transform-based fault detection algorithm and enable its operation within both a Real-Time Digital Simulator (RTDS) laboratory environment and on a physical FPGA board. Building upon the previously developed PL-side design, this work aims to realize a full hardware-software system integration. The specific goals of this research include:

- Build the hardware architecture of the system.
- Integrate and deploy the hardware architecture using embedded software.
- Establish high-throughput real-time communication between the RTDS and the FPGA board.
- Validate the entire system in terms of responsiveness to simulated real-time fault conditions, timing, and reliability.

1.3 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 provides the background of this research and reviews related work. It also introduces the IEC 61850 protocol selected for communication and the ZCU104 FPGA board used in the implementation. Chapter 3 presents the high-level system design, including the overall architecture and data flow, as well as a description of each system module. Chapter 4 details the low-level implementation of the system, including both the Processing System (PS) and the Programmable Logic (PL) components. Chapter 5 discusses the experimental setup and presents the results obtained from system validation. Finally, Chapter 6 concludes the thesis and outlines potential directions for future work.

Chapter 2

Background

Chapter 2 provides the necessary background for this thesis. It begins with a brief introduction to the S-Transform-based fault detection algorithm. Previous work related to its implementation is then discussed, along with its limitations. The IEC 61850 communication protocol is introduced, along with the benefits it brings to real-time data exchange between the FPGA and RTDS. The chapter also introduces the development environment of the project, including the ZCU104 FPGA board, the Real-Time Digital Simulator (RTDS) environment, and the FPGA design suite.

2.1 S-Transform-Based Fault Detection

The Stockwell Transform, or S-Transform, proposed by Stockwell in 1996 [3], is a linear time-frequency analysis method that combines elements of the Short-Time Fourier Transform (STFT) and the Continuous Wavelet Transform (CWT). It employs a scalable, moving Gaussian window to achieve time-frequency localization and possesses several advantages not found in the conventional CWT, such as frequency-dependent resolution with absolute phase information.

However, due to its close relation to the Fourier Transform, the S-Transform inherits some of the typical drawbacks of discretely sampled transforms. A significant limitation is its high computational complexity and memory requirements, making it inefficient for processing even moderately sized signals due to its inherent redundancy.

To address these issues, the Fast Discrete Stockwell Transform (FDST) was introduced in [4] as a non-redundant version of the original transform. FDST significantly reduces the computational burden, making the method more practical for real-time and embedded applications.

Building upon the FDST, a fault detection algorithm was proposed in [1] to enhance the performance of distance protection in power systems. This algorithm effectively improves reliability in scenarios where commercial relays may fail or misoperate, particularly under low-inertia or renewable-integrated grid conditions. Figure 2.1 shows the proposed enhanced relay comprising the algorithm [1].

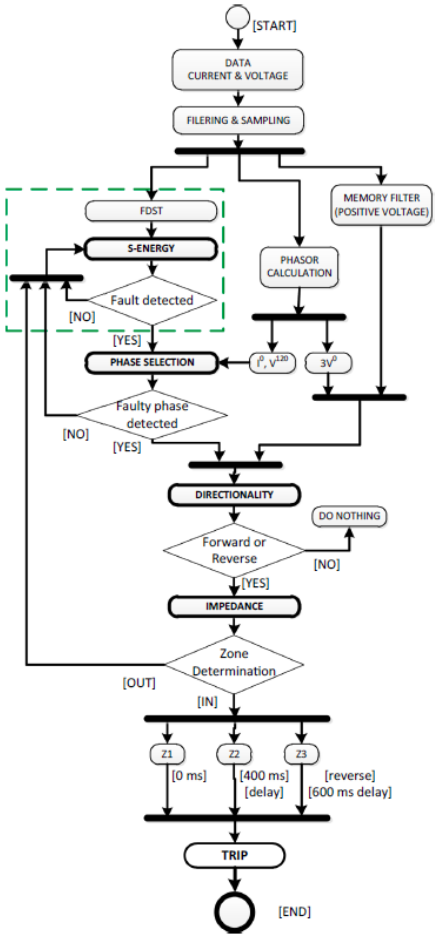


Figure 2.1: **The enhanced relay module with FDST-based fault detection [1]**

2.2 Current Implementation

The S-Transform-Based fault detection algorithm has demonstrated strong potential in enhancing distance protection [1]. However, its validation has so far been limited to simulations in MATLAB. To transition from simulation to real-world deployment such as in RTDS environments or commercial protective relays, the algorithm must be implemented on a physical hardware platform.

Distance protection systems are highly time-sensitive and require exceptional reliability. Compared with general-purpose processors or microcontrollers, Field-Programmable Gate Arrays (FPGAs) offer significant advantages for this type of application. FPGAs allow customized hardware-level implementation of the algorithm, enabling parallel processing and low-latency performance. Furthermore, the embedded processing capabilities of modern FPGA platforms allow seamless integration with external systems such as RTDS through real-time communication protocols. A similar approach is presented in [5], where an FPGA-based Digital Real-Time Simulation (DRTS) platform is proposed for

testing traveling-wave-based protection relays.

The implementation of the S-Transform-Based fault detection algorithm in the Programmable Logic (PL) was presented in [2]. The design accepts the three-phase current signals as input and computes the corresponding S-Energy values. Each of these values is then compared against a dynamically calculated threshold, which is updated every fixed number of cycles based on the input current. The comparison results indicate whether a fault is detected in each current phase. The hardware architecture is fully parameterized and fully pipelined, operating at a clock frequency of 100 MHz. The final implementation achieves a low-latency response, producing fault detection results within 570 ns. The overall architecture of the previously implemented PL system is shown in Figure 2.2 [2].

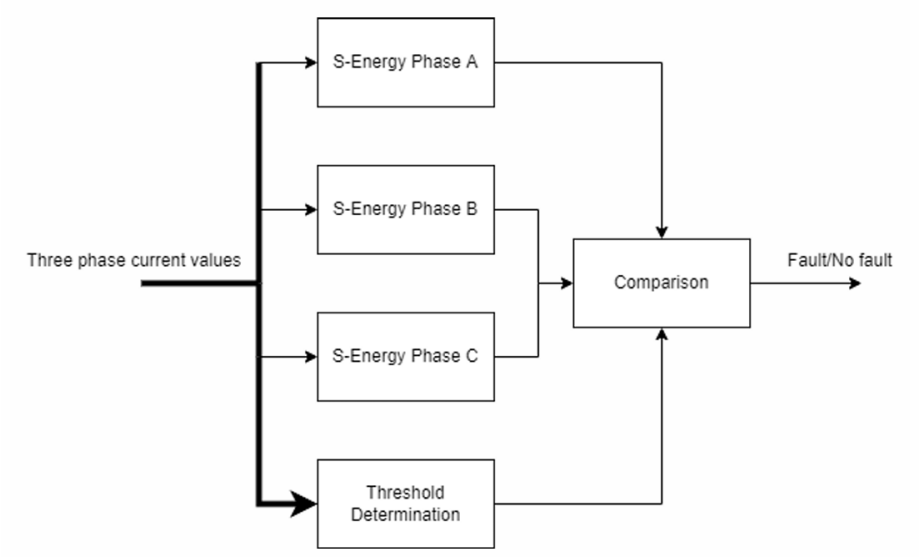


Figure 2.2: **Architecture of the previously implemented PL system [2]**

2.2.1 Fixed-point Representation

The input data to the implemented PL system are represented in fixed-point format, which requires conversion from the floating-point format used in the communication protocol. Each input is 16 bits wide, consisting of 1 sign bit, 2 integer bits, and 13 fractional bits. The experiment conducted in [2] demonstrates that this conversion from floating-point to fixed-point representation has minimal impact on both the S-energy and threshold calculations. The fault detection results remain unchanged, as the most significant deviations occur only when the computed S-energy values are below 33 dB, a region that does not affect the final comparison.

2.2.2 Threshold Determination

The S-Energy threshold module takes three phase current signals into itself to determine the S-energy threshold value with a safety margin. The value can be

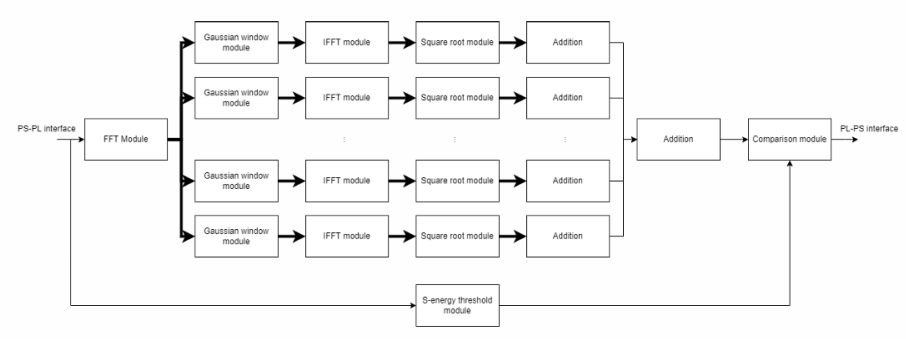


Figure 2.3: **Architecture of S-Energy calculation module [2]**

calculated from Equation 2.1.

$$S_{THsm} = -2.6\times10^3 I_{rms}^4 + 3.5\times10^3 I_{rms}^3 - 1.7\times10^3 I_{rms}^2 + 4.2\times10^2 I_{rms} - 45 \quad (2.1)$$

As described in [1], the S-energy threshold value is updated periodically at fixed time intervals. In the hardware implementation, this behavior is realized using a counter that increments with each clock cycle and triggers the calculation of the RMS value of the input current once the predefined threshold interval is reached.

The RMS value of the current is computed using a square root module implemented based on the non-restoring square root algorithm proposed in [6]. This algorithm iteratively determines each bit of the final result, producing one valid bit per iteration. A key feature of the method is the use of a partial remainder, which is updated in each iteration and plays a critical role in determining the subsequent operation. To improve throughput, the square root module is designed as a pipelined architecture.

2.2.3 S-Energy Calculation

The overall architecture of the S-Energy calculation module is shown in Figure 2.3 [2].

The module consists of several sub-modules, including an FFT module, a Gaussian window module, an IFFT module, and a square root module. It begins with the Fast Fourier Transform (FFT) module, which converts a time-domain signal into its frequency-domain representation. The FFT algorithm, first introduced by Cooley and Tukey in 1965 [7], is widely used in signal processing due to its computational efficiency in calculating the Discrete Fourier Transform (DFT). The FFT achieves this efficiency through a divide-and-conquer strategy, where a 2^n point DFT is recursively decomposed into two 2^{n-1} DFTs. This process continues until the smallest units—typically 2-point DFTs—are reached, significantly reducing the overall computational complexity from $O(N^2)$ to $O(N \log N)$.

The FFT module and IFFT module are implemented using both parallel-pipelined and sequential-pipelined architectures. The parallel-pipelined implementation offers higher throughput but consumes significantly more hardware

resources, as it allocates dedicated resources for every mathematical operation in the FFT algorithm. To address this issue, a resource-efficient sequential-pipelined implementation is adopted, where processing elements are reused across clock cycles, trading off performance for reduced hardware complexity. The sequential architecture consists of the following four key sub-modules:

- **Butterfly Module**: Performs the basic FFT butterfly computation, either passing through the data or executing addition and subtraction operations.
- **Delay Line Module**: Composed of shift registers, this module introduces specific delays to align data for processing.
- **Twiddle Factor Registers**: Store precomputed twiddle factors required for FFT computation.
- **Complex Multiply Module**: Performs complex multiplication operations using the twiddle factors and input data.

The Gaussian window module is responsible for applying a two-dimensional Gaussian window to the output of the FFT module. In the parallel-pipelined FFT implementation, the FFT outputs are generated simultaneously in parallel. Therefore, the corresponding Gaussian window coefficients can be aligned with the FFT outputs by encoding and matching their signal indices. In contrast, the sequential-pipelined FFT implementation produces outputs serially. In this case, the Gaussian-windowed outputs are written into a RAM, where the index of each FFT output determines the write address. This RAM is then connected to the IFFT module, which reads the data in the correct order, thereby preserving the alignment of the signal. The Square root module is discussed in Section 2.2.2.

2.2.4 Limitations

The limitations of this FPGA design are evident. The design is implemented in the PL part of the system and evaluated using Vivado simulation. There remain significant differences between this implementation and the deployment of the fault detection algorithm in a real-world integrated system. Typical limitations include:

- The PL design cannot be directly deployed on a real FPGA board. The hardware implementation must be extended and re-validated.
- The Processing System (PS) software required to support real-time operation is not implemented in this design.
- The communication between the FPGA board and RTDS is not established in this work.
- The performance of the implemented fault detection algorithm has only been validated using fixed inputs from static files. Therefore, its performance with high-throughput real-time current data remains unclear.
- The overall performance of this design in a real-world environment has not been evaluated and remains uncertain.

2.3 IEC 61850 Protocol

IEC 61850 is an international standard that defines communication protocols to provide communication between different equipment located in a substation, such as protection, control, and measurement equipment, as well as intelligent electronic devices (IED). Since the fault detection algorithm runs on the IED, the IEC 61850 protocol plays a vital role in enabling real-time data exchange for the fault detection process. An overview of the protocol is first presented, followed by an introduction to its key components, including Sampled Values (SV) and Generic Object-Oriented Substation Event (GOOSE) messages which are used extensively in this project.

2.3.1 Overview

An electrical power network transmits not only electrical energy but also critical information. In distance protection schemes, real-time current measurements are required as input for fault detection and decision-making. To enhance the efficiency of data acquisition and transmission, modern substations convert analog signals into digital signals, which can be easily transmitted between devices via communication links such as fiber optics or Ethernet. However, since Intelligent Electronic Devices (IEDs) are often manufactured by different vendors, the encoding and formatting of digital data can vary significantly, leading to incompatibility and inefficiencies in data exchange. To overcome this challenge and ensure seamless interoperability among devices, a unified communication protocol is essential. The key requirements of this protocol include [8]:

- High-speed and real-time data transmission
- Standard-based communication
- Supports for voltage and current samples data
- High security
- Supports for configuration
- Multi-vendor interoperability

To address these requirements, the Utility Communication Architecture (UCA) was initially developed, providing definitions for communication protocols, data models, and abstract services [8]. Building upon this foundation, the International Electrotechnical Commission (IEC) Technical Committee subsequently extended the work into the IEC 61850 standard, titled *Communication Networks and Systems in Substations*.

The IEC 61850 protocol offers several significant advantages. By utilizing a virtualized data model, a single Substation Configuration Language (SCL) file can unambiguously define the expected behavior and data exchange for each device, independent of the manufacturer. This standardization reduces the cost and complexity of device installation and migration, as a single substation LAN is sufficient and manual legacy configurations are no longer necessary. Furthermore, both the speed and accuracy of data transmission are significantly improved [9].

IEC 61850 protocol is widely applied in modern IEDs and researches. There have been several studies on the use of the IEC 61850 protocol in distance protection [10, 11, 12]. The Remote Integrated Switch (RIS) employs the IEC 61850 protocol with GOOSE messaging to significantly minimizing outage time and improve the overall performance [13]. A Model is presented [14] for a microgrid protection system with logical nodes provided in IEC 61850 and IEC 61850-7-420 communication standards. Distributed network employs IEC 61850-based communication for an adaptive overcurrent protection scheme which decrease the fault conditions and reduce the operating time [15]. Closed-loop testing using IEC 61850-configured relay is proved to be feasible in RTDS environment [16]. Besides, IEC 61850 is also used for modern sub-station automation system (SAS) while maintaining the reliability and availability [17].

2.3.2 SV Sampling Data

The Sampled Values (SV) protocol is based on IEC 61850-9-2 standard, which is a simplified implementation of the IEC 61850 standard tailored for digital substations. The core concept of SV communication is that a publisher periodically transmits measurement messages at precisely defined time intervals. This interval is determined by two parameters: the measured signal frequency and the Samples Per Period (SPP). IEC 61850-9-2 specifies two typical SPP values: 80 and 256. For instance, if the signal frequency is 50 Hz and SPP is 80, the resulting transmission interval is $1/(50 \times 80)$, or 250 µs. All SV messages are published under a multicast topic. Subscribers receive all messages on the network but process only those with the specific topic to which they are subscribed. SV messages are widely used in scenarios where publishers need to transmit sampling data such as current phase data.

The SV message frame format is explained in Table 2.1.

The APDU field contains the payload of an SV message. Each APDU can include up to 8 Application-Specific Data Unit (ASDU), where each ASDU carries one set of three-phase current and voltage measurements. Each ASDU is associated with a unique SV identification value. The structure of an APDU is shown in Figure 2.4.

Each ASDU should contain the following fields:

- **svID**: Sampled Values Identifier, a user-defined unique string identifier used for subscription
- **smpCnt**: Index of the SV message
- **confRev**: Configuration revision
- **smpSynch**: The synchronization mechanism of the clock used for sending SV messages is indicated by a specific code: 0 for None, 1 for Local, and 2 for Global synchronization
- **Sequence of Data**: Sequence of measured voltage and current values as shown in Figure 2.5

The data ending with a *q* represents the quality of the measured values from each logical node. Each dataset in the ASDU can be divided into eight measured values——currents in three phases and neutral, and voltages in three phases and

Field name	Value	Description
Destination address	`01:0c:cd:04:00:00 - 01:0c:cd:04:01:ff`	Destination MAC address
Source address	Defined by the sending device	Source MAC address
Priority tagged	TPID: `0x8100` User priority: 1 to 7 CFI: 0 VID: 0 - 4095	Defines the 802.1Q protocol SV message priority: 4-7 is high; 1-3 is low Virtual LAN ID
Ethertype	`0x88ba`	Defines the SV protocol
APPID	`0x4000 - 0x7FFF`	Application ID
Length		Message length
Reserved 1	`0x0000`	Reserved field
Reserved 2	`0x0000`	Reserved field
APDU		Application Protocol Data Unit

Table 2.1: **SV Message Frame Format**

neutral——each occupying four bytes, along with their corresponding quality indicators. The measured current and voltage values are transmitted as 32-bit integers, with a scaling factor applied: 0.001 for currents and 0.01 for voltages.

2.3.3 GOOSE Message

The GOOSE (Generic Object Oriented Substation Event) protocol is a communication model defined in the IEC 61850 standard. It uses fast and reliable mechanisms to group various types of data (such as status and values) into a dataset for transmission, and is therefore widely adopted for transferring fault status. The message contains the following field:

- **Destination Address**: Mac address of the destination device range from `01:0c:cd:01:00:00` to `01:0c:cd:01:01:ff`
- **Source Address**: Mac address of the source device
- **Priority Tag**: The TPID (Tag Protocol Identifier) is `0x8100`, the user priority is by default 4, and the VID (VLAN ID) is by default 0
- **Reserved field**: `0x0000`
- **APPID**: The application ID range from `0x0000` to `0x3fff`
- **APDU**: Application Protocol Data Unit containing the payload to be transmitted

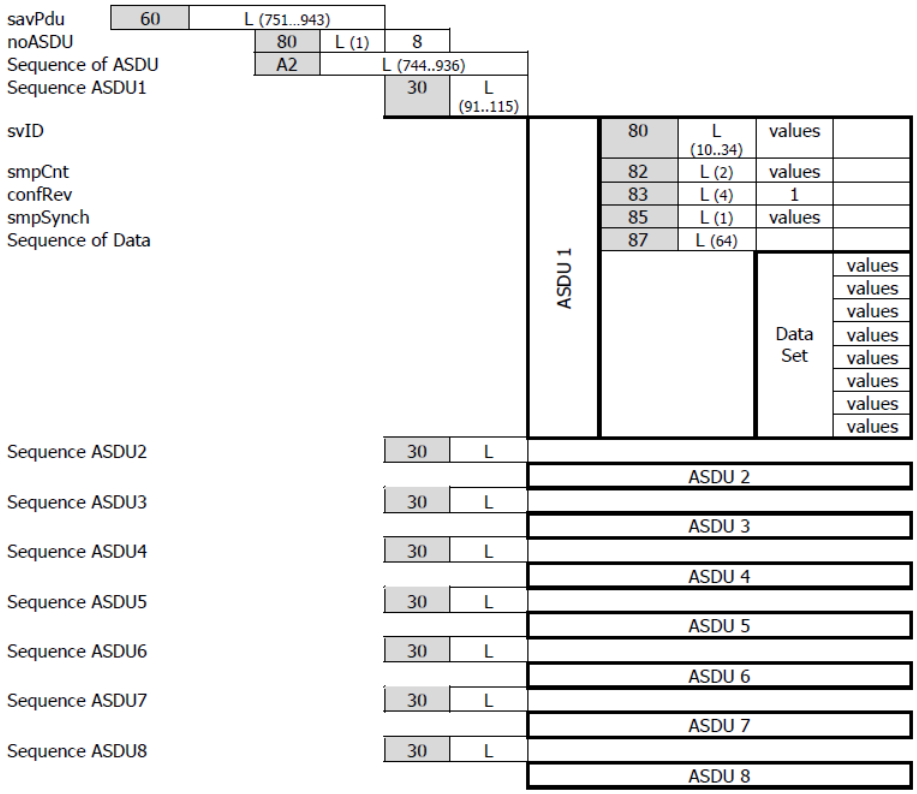


Figure 2.4: **APDU architecture**

2.4 Development Environment

This section presents the development environment. It first introduces the FPGA hardware platform, followed by the RTDS system, and finally the hardware/software design tools used.

2.4.1 Xilinx Zynq MPSoC FPGA

FPGAs (Field Programmable Gate Arrays) are a type of integrated circuit that can be reprogrammed to implement any digital logic at any time after manufacturing [18]. This provides high flexibility, making FPGAs suitable for a wide range of applications including IC verification [19], image processing [20], audio processing [21], deep learning [22], and communication systems [23]. Moreover, FPGAs also provide fast computation speed for IEC 61850 protocol [24].

The FPGA used in this work is the Xilinx Zynq MPSoC ZCU104. This device integrates a quad-core Arm Cortex-A53 processing system and a dual-core Arm Cortex-R5 real-time processor, enabling heterogeneous multiprocessing for application developers. The ZCU104 evaluation board provides a flexible prototyping platform, featuring high-speed DDR4 memory interfaces, an FMC expansion port, multi-gigabit per second serial transceivers, various peripheral interfaces, and FPGA fabric for customized designs. The overall block diagram of the

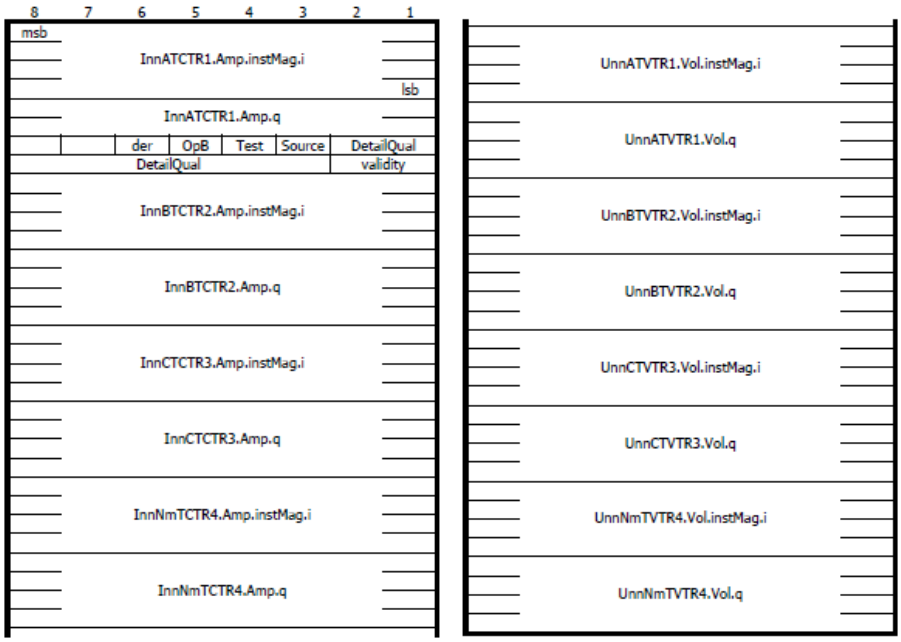


Figure 2.5: **ASDU architecture**

device is shown in Figure 2.6.

The Ethernet and memory performance of the device are critical for this work. The ZCU104 features a Gigabit Ethernet Controller (GEM), which implements a 10/100/1000 Mb/s Ethernet MAC compliant with the IEEE 802.3-2008 standard. It supports both half-duplex and full-duplex modes at 10/100 Mb/s and full-duplex mode at 1000 Mb/s, ensuring high-speed communication under the IEC 61850 protocol. In addition, the device includes a 4 GB, 64-bit wide DDR4 memory system which comprises 16 SDRAMs, each with 256 MB. This high-speed memory system guarantees fast data transmission between the PS and the PL.

2.4.2 RTDS

The Real-Time-Digital-Simulator (RTDS) performs electromagnetic transient (EMT) simulations of power systems in real time. This enables highly efficient and detailed studies, allowing users to anticipate system and device behaviors that may threaten the stability, resilience, and performance of the grid. It also allows physical equipment, such as protection and control devices or power electronics, to be tested in closed-loop configurations with the simulated network. The interaction between the grid and protection, control, and power devices during system transients can thus be analyzed in detail across a wide frequency bandwidth.

RTDS is widely used for closed-loop testing of distance protection schemes. For instance, a dynamic testing methodology for evaluating the performance of distance protection was proposed in [25]. In [26], an electromagnetic transient model of a DFIG-based wind generator was developed in RTDS to study its

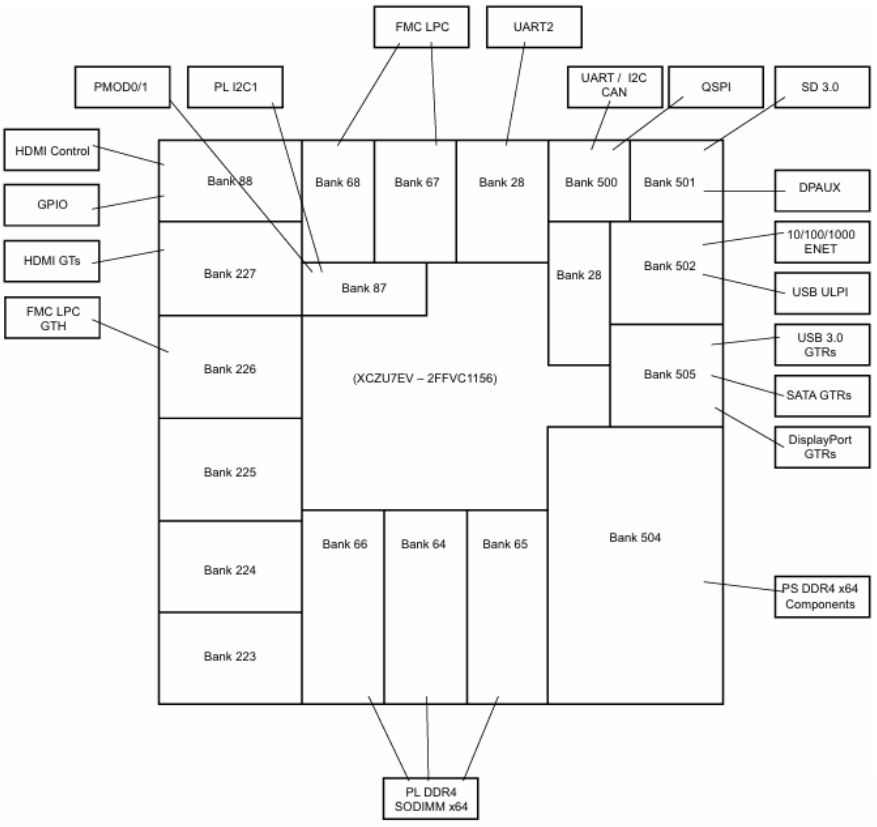


Figure 2.6: **Block diagram of zcu104**

impact on distance relays. Furthermore, an ultra-high-voltage (UHV) test environment was established in RTDS in [27], and various relay protection strategies were evaluated within that setup.

RTDS supports the IEC 61850 protocol. A network interface card, the GTNETx2, receives data from the power system simulation via optical fiber and transmits it to external IEDs using IEC 61850. This bidirectional card can also transfer data from external devices back to the simulated network. In this work, the FPGA is considered as an external IED within the IEC 61850 communication framework. In this way, the RTDS publishes SV messages, as described in Section 2.3.2, to the FPGA, and receives GOOSE messages from the FPGA, as described in Section 2.3.3.

2.4.3 FPGA Design Suite

The development of the PS and the PL in this work is carried out using Vivado and Vitis, both part of the Xilinx Design Suite. The PL system is developed in Vivado, which enables simulation and implementation of the hardware design. The hardware product used in Vitis is generated in Vivado through a process that includes synthesis, implementation, and bitstream generation. Based on this hardware product, a Vitis platform project is created, allowing modi-

fication of the board support packages and device drivers. The PS software, which also serves as the entry point of the overall system, is implemented as an application project on top of the platform. This workflow enables seamless hardware–software co-design and integration.

Chapter 3

Design Architecture

In this chapter, the design requirements are presented. The overall system architecture, including the functional flow and block diagram, is discussed. The system is analyzed to determine whether its architecture meets the specified design requirements.

3.1 Design Requirements

The design should meet the following requirements:

- The Ethernet LAN should be capable of receiving IEC 61850 data from RTDS, with a throughput sufficient to handle the maximum SV sampling data rate streamed by the RTDS.
- The received data must be extracted from the Ethernet payload and converted from 32-bit floating-point format to 16-bit fixed-point format.
- The transformed data should be stored in the device memory and transmitted to the PL in groups of 16 samples [2]. The overall data throughput between the PS and the PL must remain within the throughput limitations of both the device memory and the PL system.
- The PS should be notified immediately when a fault is detected in the PL system.
- Upon fault detection, a GOOSE message should be sent to the RTDS with minimal delay. If no fault is detected, periodic GOOSE messages should still be transmitted at regular intervals.
- The RTDS must be able to receive the fault-indicating GOOSE message within 5 milliseconds after the fault is simulated.
- The system should demonstrate high reliability and robustness to ensure suitability for long-term operation.

3.2 System Architecture

According to the design goals discussed in Section 3.1, the overall block diagram of the system is shown in Figure 3.1.

The system consists of two parts: the RTDS environment and the FPGA board. The RTDS send SV sampling data to the FPGA and receive GOOSE message from FPGA via optic fibers.

The FPGA consists of two main components: the Processing System (PS) and the Programmable Logic (PL). The Ethernet driver in the PS controls the Ethernet port for data transmission and reception. The PS is responsible for decoding SV messages (subscriber) and encoding GOOSE messages (publisher) according to the IEC 61850 protocol. All data is stored in the high-speed DDR4 memory. A DMA controller handles memory-to-memory and memory-to-I/O data transfers. The PS also manages the interrupt driver, which triggers corresponding interrupt service routines.

The PL receives and sends data via AXI streams. The received data extracted from the AXI stream are simultaneously sent to both the S-Energy calculation module and the Threshold calculation module. The comparison module then compares the outputs of these two modules to determine whether a fault has occurred. If a fault is detected, an interrupt is triggered and sent to the PS; otherwise, the system remains idle.

The data flow of the system is shown in Figure 3.2. A typical fault detection data flow is presented:

1. Once the SV message data are generated by the RTDS component, they are transmitted to the GTNETx2 card and forwarded through the fiber port.
2. The FPGA receives the data via the Ethernet driver and passes it to the SV subscriber module.
3. The current data are decoded and converted into fixed-point representation before being stored in DDR4 memory.
4. The PS monitors the number of SV packets received and invokes the DMA controller when necessary.
5. The PL receives the current data via AXI stream through the DMA controller.
6. The S-Transform calculation module and the threshold calculation module use the data as input.
7. After the calculations, the S-Energy result is compared with the threshold.
8. If a fault is detected, an interrupt is triggered and sent to the PS.
9. The interrupt handler in the PS invokes the GOOSE publisher, which sends the fault message via the Ethernet driver.
10. The RTDS receives the GOOSE message and updates the corresponding simulated signals accordingly.

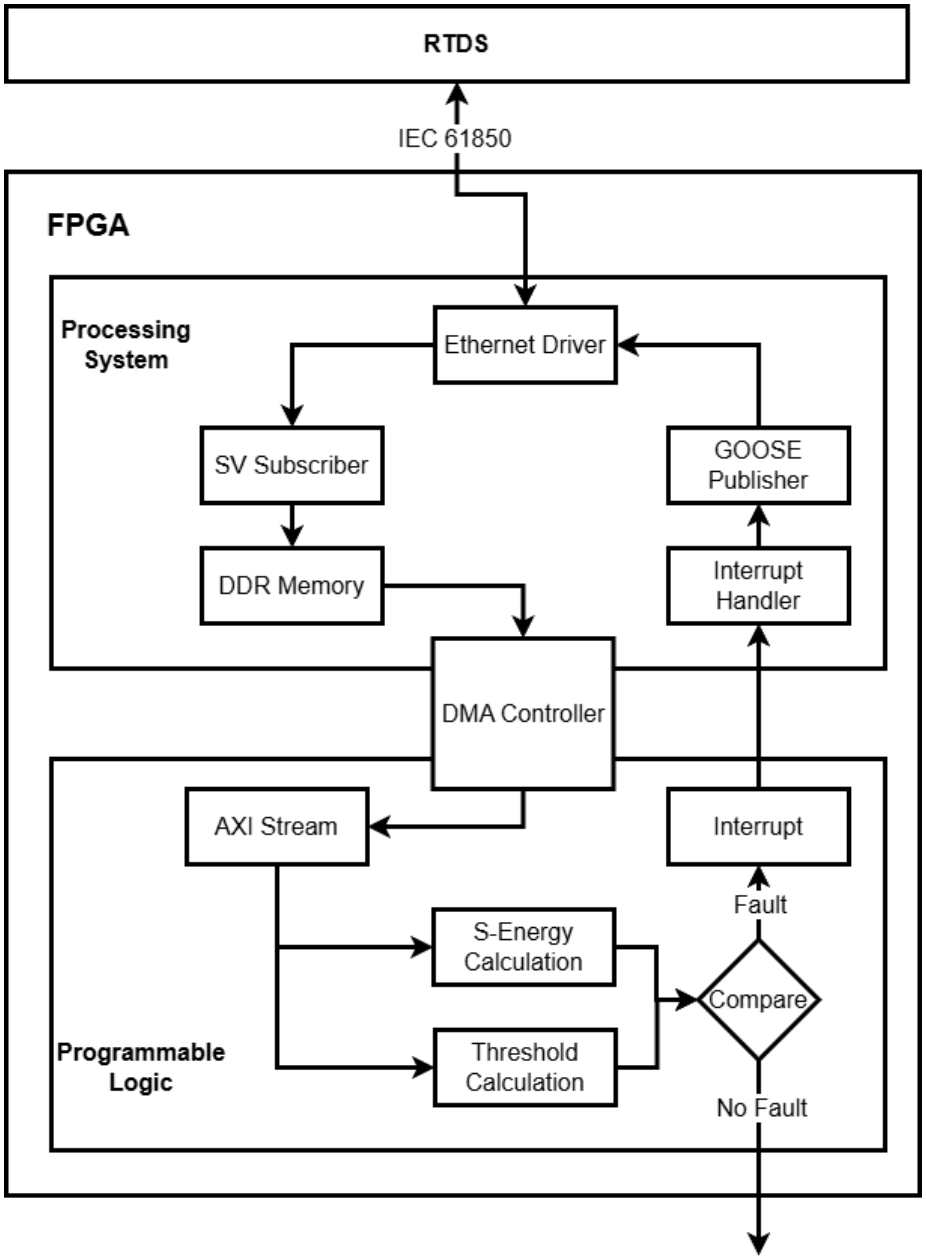


Figure 3.1: **Overall block diagram of the system**

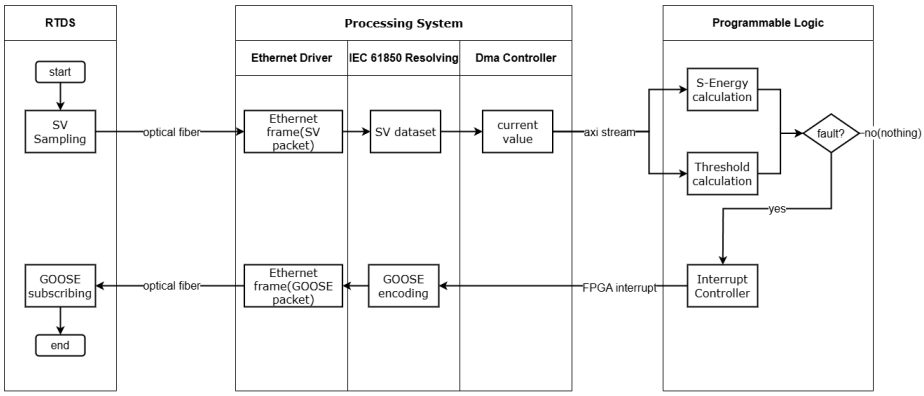


Figure 3.2: **Data flow of the design**

3.3 System Analysis

3.3.1 SV Throughput

The system design must meet the throughput requirements outlined in Section 3.1, particularly regarding Ethernet and DDR4 memory performance.

The SV component in RTDS supports various sampling rates ranging from 80 samples per cycle up to 250 kHz. Each SV packet can be configured with a different number of channels and ASDUs. Given that the maximum Ethernet bandwidth supported by the ZCU104 is 1000 Mbps, it is necessary to evaluate the maximum size of an SV packet to ensure the system remains within bandwidth limits. An approximate expression for the size of each SV packet is given in Equation 3.1, assuming an Ethernet header size of 40 bytes and an ASDU header size of 32 bytes:

$$\text{Packet\_Size} \approx 40 + N_{asdu} \times (32 + 4 \times N_{channel}) \tag{3.1}$$

Here, N_{asdu} is the number of ASDUs per packet, $N_{channel}$ is the number of sampled channels in each ASDU (e.g., voltage and current phases), and each channel is represented using a 4-byte (32-bit) integer value.

Suppose each SV packet contains 16 channels including 4 current channels, 4 voltage channels, and their corresponding 8 quality indicators. Based on Equation 3.1, and given that each cycle of the power signal takes 20 ms (corresponding to a 50 Hz system), we can estimate the size of each packet and the corresponding data throughput under different SV sampling configurations in RTDS. The results are shown in Table 3.1.

It is evident from Table 3.1 that the FPGA provides sufficient bandwidth to handle SV communication. In fact, the true performance bottleneck may lie in the different SV sampling rates used in the project. As shown in the table, the SV sampling rate of 250 kHz results in significantly higher throughput compared to that at 80 samples/cycle. Moreover, the DDR4 memory on the ZCU104 board theoretically offers a much higher bandwidth than even the maximum achievable throughput of the SV sampling data, indicating that memory is unlikely to be a limiting factor.

SV sampling rate	Size of each packet (B)	Throughput (MB/s)
80 s/c, 1 ASDU	136	0.519
256 s/c, 8 ASDUs	808	1.233
4800 Hz, 2 ASDUs	232	0.531
14400 Hz, 6 ASDUs	616	1.41
96 s/c, 1 ASDUs	136	0.623
96000 Hz, 1 ASDUs	136	12.45
250 kHz, 1 ASDUs	136	32.42

Table 3.1: **SV sampling rate and throughput**

3.3.2 Operating System

The choice of the operating system for the PS is critical to the overall system design. For the ZCU104 platform, three common options are: bare-metal, real-time operating system (RTOS), and Linux.

- **Bare-metal:** In a bare-metal system, the application runs directly on the hardware without any operating system. This approach minimizes both memory footprint and latency, and reduces the likelihood of unpredictable behavior since all code is explicitly managed by the developer [28]. However, it significantly increases development complexity, as all hardware drivers, such as those for Ethernet, DMA, and interrupt handling, must be implemented manually.
- **RTOS:** A real-time operating system (RTOS) is designed for applications with strict timing requirements. Unlike general-purpose systems that focus on fairness or throughput, an RTOS ensures deterministic behavior by guaranteeing that critical tasks complete within predefined deadlines. RTOS is particularly suited for systems with multiple concurrent tasks and hard or soft real-time constraints [29].
- **Linux:** Linux is an open-source, Unix-like operating system based on the Linux kernel. It offers rich functionality, extensive library support, and a user-friendly development environment, making it ideal for complex applications. However, it introduces higher latency and is generally less deterministic compared to RTOS or bare-metal, making it less suitable for systems with tight real-time requirements [30].

In this work, the bare-metal approach is adopted as the operating system. Given the system's strict real-time requirements and the need for high reliability and robustness, bare-metal provides full control over hardware resources, making it well-suited for such applications. Although RTOS is a viable alternative particularly for systems with multiple concurrent tasks, this project involves relatively simple task scheduling, with interrupt handling being the only form of concurrency. Additionally, the system operates on a single processor core, meaning that there is no need for parallel task management. While the Linux environment offers convenience in eliminating the need to implement Ethernet

drivers, DMA controllers, and interrupt handlers, its general-purpose kernel may introduce unanticipated delays, potentially compromising the system's timing constraints. For this reason, Linux was not selected.

3.3.3 Latency Analysis

In this work, system latency is defined as the elapsed time between the appearance of a fault in the RTDS and the reception of the corresponding GOOSE message by the RTDS. The approximate latency include the following:

- **SV Data Reception and Parsing:** The SV packets are received by the Ethernet driver and decoded by the PS, with an estimated delay of 100 us. Since the packets are transferred every 16 samples [2], the approximate delay depends on the sampling configuration of the RTDS. Under the 80 s/c setting, it takes approximately 4 ms to accumulate 16 samples.
- **DMA Transmission:** Transferring decoded current data to the PL via DMA takes about 100 µs.
- **S-Energy calculation:** The module computes the S-Energy and takes about 500 ns.
- **Fault Detection in the PL:** The S-Transform and threshold comparison modules takes within 0.5 µs.
- **Interrupt Handling:** The fault detection algorithm triggers an interrupt and sends it to the PS, which calls the ISR and calls the GOOSE publisher. This takes approximately 10 µs.
- **GOOSE Transmission:** The GOOSE message is encoded and sent via the Ethernet driver, with a latency of 100 µs.

The total estimated latency is slightly exceeding 4 ms under 80 s/c conditions, which is lower than the 5 ms requirement. However, different RTDS configurations significantly influence the final system latency. The detailed latency will be extracted from the experiment.

Chapter 4

System Implementation

In this chapter, the detailed implementation of the system is presented. Each module introduced in Chapter 3 is discussed in depth, including its functionality, internal structure, and interactions with other components in the system.

4.1 Hardware Implementation

The hardware block diagram implemented in Vivado is shown in Figure 4.1.

The block diagram consists of several modules, including one ZCU104 device, one DMA controller with two AXI FIFO streams, one MM2S (memory-mapped to stream) data reader, three S-Energy calculation modules, one threshold calculation module, and one comparison module. The S-Energy and threshold calculation modules are implemented in [2]. Therefore, the remaining modules will be described in this section.

4.1.1 Zynq MPSoC

This module manages all internal components of the Zynq MPSoC device. The functional blocks within the module are illustrated in Figure 4.2.

In this work, the key functional blocks utilized are: GEM3, I2C, and CAN modules to enable Ethernet communication; the fabric interrupt controller to manage interrupt signals; DDR memory and AXI interfaces to achieve high-speed data exchange between the PS and the PL; and the GPIO block, which is accessed through EMIO for testing and debugging purposes.

The system operates with a clock frequency of 100 MHz. The reset signals for the MM2S module, S-Energy calculation modules, threshold calculation module, and comparison module are active-high, while reset signals for the remaining modules are active-low. The system contains eight interrupt sources: two DMA interrupts (for reading and writing), one Ethernet interrupt, one GPIO interrupt, one timer interrupt, and three S-Energy calculation interrupts for fault detection.

4.1.2 DMA Controller and AXI FIFO

The DMA controller settings are shown in Figure 4.3.

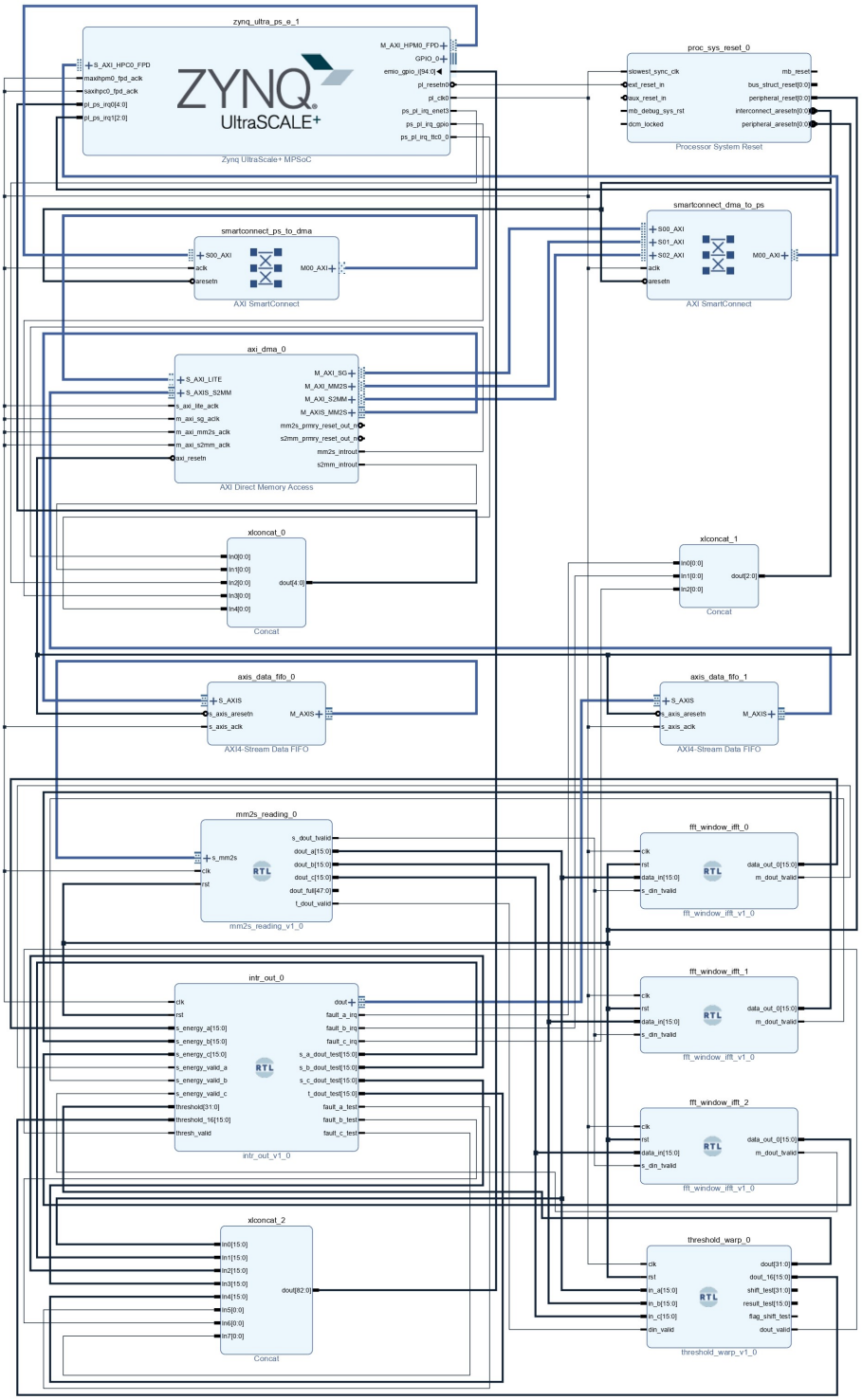


Figure 4.1: **Block diagram of the hardware implementation**

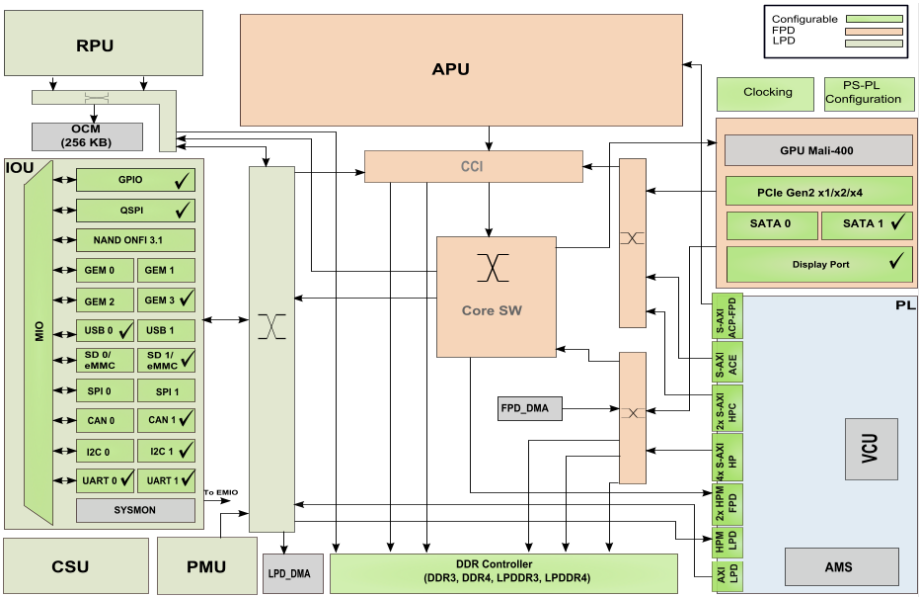


Figure 4.2: **Functional blocks of MPSoC**

The DMA is configured in scatter-gather (SG) mode. In this mode, the DMA transfer parameters are specified using memory-resident descriptors. The software sets up the source (SRC) and destination (DST) descriptors and programs the corresponding registers to point to the starting addresses of these descriptors in memory. Once the DMA channel is enabled, it automatically fetches the SRC and DST descriptors and uses the provided parameters to carry out the actual data transfer. The scatter-gather mode is chosen for its flexibility and efficiency in handling non-contiguous memory blocks, which is essential in high-speed, real-time applications such as this system.

The data stream width of the DMA is configured to 64 bits. This ensures that the three-phase current values for each sample can be transferred in a single output transaction ($16 \times 3 = 48 < 64$ bits). Similarly, the input test data which comprise the three-phase S-Energy values and one threshold value can also be transferred in a single transaction.

The AXI FIFO block is connected to the DMA controller to enable data streaming from and to the DMA controller.

4.1.3 MM2S Reading

The MM2S module feeds the AXI-streamed FIFO data into the subsequent calculation modules. Given that the data stream width is 64 bits, this module splits each data into three 16-bit vectors representing the three-phase current values, along with one 16-bit quality vector, which is not used in the implementation. The following code demonstrates this logic.

```
data_width := 16;
-- full stream data
dout_full_buf <= s_mm2s_tdata(data_width*3-1 downto 0);
```

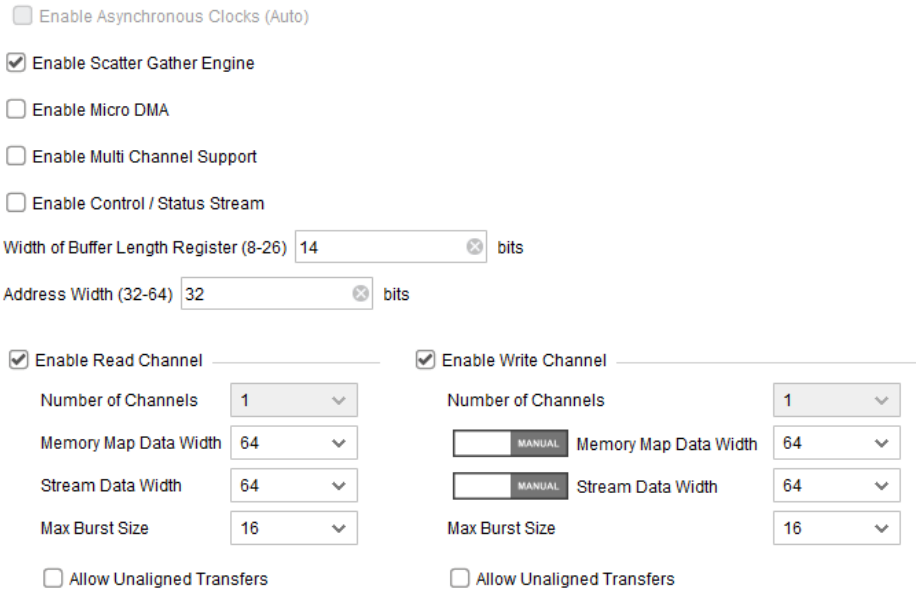


Figure 4.3: **Settings of DMA controller in the PL**

```
-- input current phase a
dout_a_buf <= s_mm2s_tdata(data_width*3-1 downto data_width*2);
-- input current phase b
dout_b_buf <= s_mm2s_tdata(data_width*2-1 downto data_width);
-- input current phase c
dout_c_buf <= s_mm2s_tdata(data_width-1 downto 0);
```

The generic parameters and the ports of the module are shown in Table 4.1. The following are the signals description:

- `clk`: Clock signal.
- `rst`: Reset signal.
- `s_mm2s_tdata`: AXI protocol signal carrying the actual streamed data.
- `s_mm2s_tvalid`: AXI protocol signal from the master module indicating that the data is valid.
- `s_mm2s_tready`: AXI protocol signal from the slave module indicating that it is ready to receive data. This signal is set to '1' when the input `s_mm2s_tvalid` signal is '1'.
- `s_mm2s_tlast`: AXI protocol signal from the master module indicating that the current stream is the last valid data in this transfer.
- `s_dout_tvalid`: Output signal indicating to the S-Energy calculation module whether the current data is valid. This signal is set to '1' only when both `s_mm2s_tvalid` and `s_mm2s_tready` are '1'; otherwise, it is '0'.

Generic	Type	Default Value
data_width	integer	16
Port	**Direction**	**Type**
clk	input	std_logic
rst	input	std_logic
s_mm2s_tdata	input	std_logic_vector(4*data_width-1 downto 0)
s_mm2s_tvalid	input	std_logic
s_mm2s_tready	output	std_logic
s_mm2s_tlast	input	std_logic
s_dout_tvalid	output	std_logic
dout_a	output	std_logic_vector(data_width-1 downto 0)
dout_b	output	std_logic_vector(data_width-1 downto 0)
dout_c	output	std_logic_vector(data_width-1 downto 0)
t_dout_tvalid	output	std_logic

Table 4.1: **Generic parameters and ports of MM2S module**

- `dout_a`: Output current data of phase A.
- `dout_b`: Output current data of phase B.
- `dout_c`: Output current data of phase C.
- `t_dout_tvalid`: Output signal indicating to the threshold calculation module whether the current input data is valid. This signal is set to '1' at the same clock cycle when `s_dout_tvalid` is set to '1', and it remains high until the next reset. This design ensures that the threshold module always receives valid input when initiating a calculation, regardless of the specific clock cycle it begins reading. This is safe because the streamed input remains at the last valid value and does not change during invalid periods.

4.1.4 Comparison and Interrupt

The comparison module compares the S-Energy result from the S-Calculation module with the threshold value from the threshold calculation module, provided that all input signals are valid. The interrupt signal is asserted one clock cycle after the clock cycle in which the input results become valid. If the S-Energy result exceeds the threshold value, an interrupt corresponding to the respective current phase is asserted. This interrupt is used to indicate a fault condition and initiate the fault handling routine in the PS.

The generic parameters and the ports of the module are shown in Table 4.2. The following are the signals description:

- `clk`: Clock signal.

Generic	Type	Default Value
data_width	integer	16
Port	**Direction**	**Type**
clk	input	std_logic
rst	input	std_logic
fault_a_irq	output	std_logic
fault_b_irq	output	std_logic
fault_c_irq	output	std_logic
s_energy_a	input	std_logic_vector(data_width-1 downto 0)
s_energy_b	input	std_logic_vector(data_width-1 downto 0)
s_energy_c	input	std_logic_vector(data_width-1 downto 0)
s_energy_a_valid	input	std_logic
s_energy_b_valid	input	std_logic
s_energy_c_valid	input	std_logic
threshold_16	input	std_logic_vector(data_width-1 downto 0)
threshold_valid	input	std_logic
s_mm2s_tdata	output	std_logic_vector(4*data_width-1 downto 0)
s_mm2s_tvalid	output	std_logic
s_mm2s_tready	input	std_logic
s_mm2s_tlast	output	std_logic

Table 4.2: **Generic parameters and ports of comparison module**

- `rst`: Reset signal.
- `fault_a_irq`: The interrupt signal of phase a indicating a fault in phase a.
- `fault_b_irq`: The interrupt signal of phase b indicating a fault in phase b.
- `fault_c_irq`: The interrupt signal of phase c indicating a fault in phase a. These three signals are connected to the Zynq MPSoc block to trigger the hardware interrupt.
- `s_energy_a`: The input signal indicating the result from S-Energy calculation module for phase a.
- `s_energy_b`: The input signal indicating the result from S-Energy calculation module for phase b.
- `s_energy_c`: The input signal indicating the result from S-Energy calculation module for phase c.

- `s_energy_a_valid`: The input signal indicating whether the result is valid from S-Energy calculation module for phase a.
- `s_energy_b_valid`: The input signal indicating whether the result is valid from S-Energy calculation module for phase b.
- `s_energy_c_valid`: The input signal indicating whether the result is valid from S-Energy calculation module for phase c.
- `threshold_16`: The input signal indicating the result from threshold calculation module.
- `threshold_valid`: The input signal indicating whether the result is valid from threshold calculation module.
- `s_mm2s_tdata`: AXI protocol signal carrying the results from all calculation modules for testing.
- `s_mm2s_tvalid`: AXI protocol signal indicating that the data is valid. This signal is set to '1' when all signals from calculation modules are valid.
- `s_mm2s_tready`: AXI protocol signal from the slave module indicating that it is ready to receive data.
- `s_mm2s_tlast`: AXI protocol signal from the master module indicating that the current transfer contains the last valid data stream. Since only one 64-bit data word is transmitted per transfer, this signal is set to '1' at the next cycle after both `s_mm2s_tvalid` and `s_mm2s_tready` are high.

4.2 Interrupt Handling

The PS uses the Arm PL-390 Generic Interrupt Controller (GIC), which complies with the GICv1 architecture specification. Interrupt handling plays a fundamental role in supporting other modules such as the Ethernet driver, timer, GPIO, DMA controller, and fault detection module. The initialization and handling of interrupts across different modules follow a similar process. The initialization procedure for the fault detection interrupt serves as a representative example and is described below:

1. The initiator software calls the `XScuGic_LookupConfig()` function. This function retrieves the device configuration based on a unique device ID and returns the corresponding configuration structure. Since there is only one device, the ID is always set to 0.
2. The `XScuGic_CfgInitialize()` function is then called, using the configuration obtained in the previous step to initialize the interrupt controller.
3. The interrupt priority and trigger type are configured using the `XScuGic_SetPriorityTriggerType()` function.
4. The `XScuGic_Connect()` function is used to register the device driver's interrupt handler. These handlers are associated with different interrupt IDs defined by the device, allowing them to be differentiated from each

other. This handler is invoked when the associated interrupt occurs. In the case of fault detection, three separate handlers are defined, each corresponding to one of the three fault currents. The GOOSE publisher is triggered within these handlers upon detecting a fault. Besides, each handler has a callback function as a parameter. The handler can invoke the callback function once an interrupt occurs.

5. Hardware interrupts are enabled by calling the `XScuGic_Enable()` function.

6. Finally, the exception handling system is initialized and enabled through the functions `Xil_ExceptionInit()`, `Xil_ExceptionRegisterHandler()`, and `Xil_ExceptionEnable()`.

4.3 Ethernet Driver

The Ethernet Driver of the PS is implemented based on the lightweight TCP/IP (lwIP). lwIP is a lightweight, standalone implementation of the TCP/IP protocol suite. The primary focus of the lwIP TCP/IP stack is to minimize RAM usage while maintaining a complete and functional TCP implementation. This makes lwIP particularly suited for embedded systems with limited resources.

4.3.1 Initialization

The progress of initialization the LwIP in this project is shown in Figure 4.4. The description is the following:

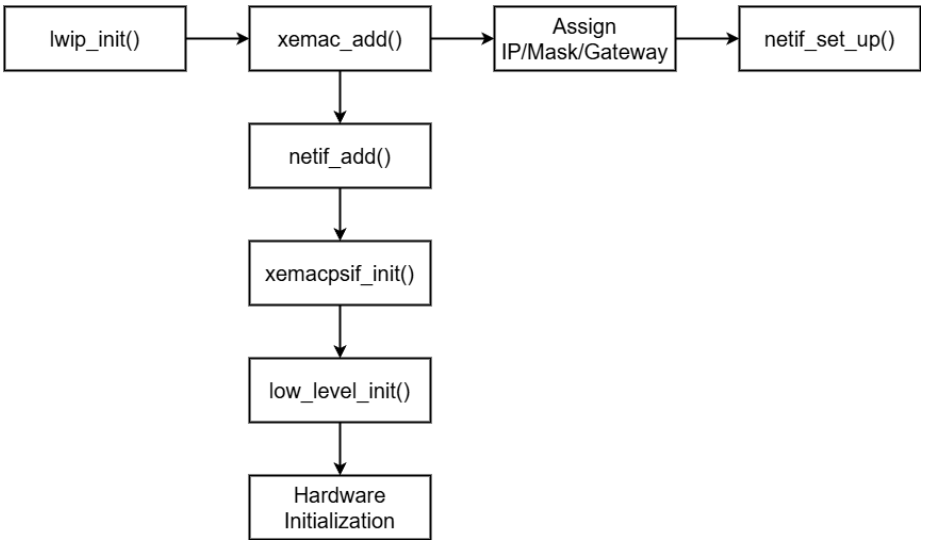


Figure 4.4: **Initialization of LwIP**

1. The `lwip_init()` is called to initialize all the sub-modules of the lwIP including `pbuf, netif, memory, TCP/UDP`, etc.

2. The Xilinx wrapper function `xemac_add()` is invoked to add the Ethernet MAC interface. This function is a wrapper around the generic lwIP function `netif_add()`, and its purpose is to provide portability across different Xilinx Ethernet MAC (EMAC) implementations. In this project, the EMAC driver is used as the base Ethernet driver, so the function `xemacpsif_init()` is registered with the netif structure in lwIP. The netif object created and managed by lwIP represents the network interface for the device's Ethernet functionality. The `low_level_init()` function is the lowest-level initialization function within lwIP, responsible for configuring the underlying hardware. On the ZCU104 platform, hardware initialization includes MAC controller setup, I/O configuration, buffer descriptor initialization, and interrupt setup. The initialization flow of the EMAC hardware on ZCU104 is illustrated in Figure 4.5.

3. The IP address, IP mask, and gateway are assigned according to the RTDS environment.

4. The netif interface is brought up, available for processing the Ethernet traffic.

4.3.2 Receiving and Sending

The data reception operates in polling mode, meaning that the processor actively invokes the receiving function and continuously checks for incoming packets until one is received even if the packet is empty. The process of receiving a packet in lwIP is shown in Figure 4.6. On the ZCU104 platform, the primary function responsible for packet reception is `xemacif_input()`. The internal structure and logic of this function are described below:

- The function `xemacpsif_input()` function is invoked, which is the EMAC-specific input handler for the Ethernet MAC (EMAC) driver used in this project. It is responsible for retrieving incoming Ethernet frames from the hardware buffer, allocating memory for each frame,

- The `low_level_input()` function is called by `xemacpsif_input()`. It directly interacts with hardware registers to check for any incoming packets.

- Once `low_level_input()` completes and a valid Ethernet frame is received, it is passed to the `netif->input()` function by `xemacpsif_input()` for further processing by lwIP. The frame is stored in the `pbuf` pointer. By default, if the received frame contains an unrecognized protocol (such as IEC 61850), it will be discarded. Therefore, the driver must be modified to correctly handle IEC 61850-based packets.

- The `ethernet_input()` function is invoked by `netif->input()` to handle the received Ethernet frame. It parses the frame header to determine how the packet should be processed. To support IEC 61850, this function is reimplemented to forward unknown protocols to the user-defined `LWIP_HOOK_UNKNOWN_ETH_PROTOCOL()` hook.

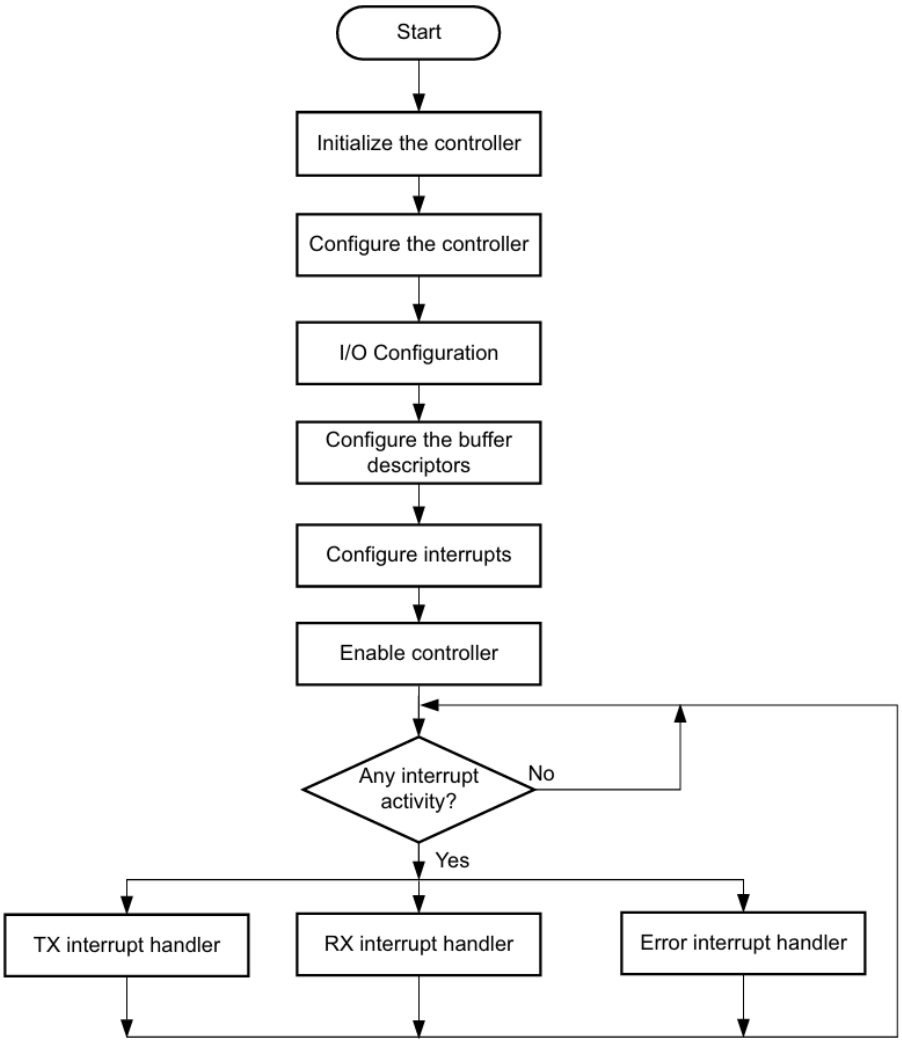


Figure 4.5: **Progress of the EMAC hardware initialization on ZCU104**

- A `lwip_hooks.h` file is added to the driver. This header declares the `LWIP_HOOK_UNKNOWN_ETH_PROTOCOL()` function by referencing the `iec61850_sv_hook()` function as `extern`. This enables `ethernet_input()` to pass unknown frames to `iec61850_sv_hook()` within the PS software. The `iec61850_sv_hook()` function then extracts the Ethernet frame content from the `pbuf` structure and returns the data to the polling routine.

The process of sending a packet in lwIP is shown in Figure 4.7 and described below:

1. The Ethernet frame is encapsulated within the `pbuf` structure using the `pbuf_alloc()` and `pbuf_take()` functions.
2. The `netif->linkoutput()` function is then called to pass the Ethernet frame to the `low_level_output()` function, which handles the actual

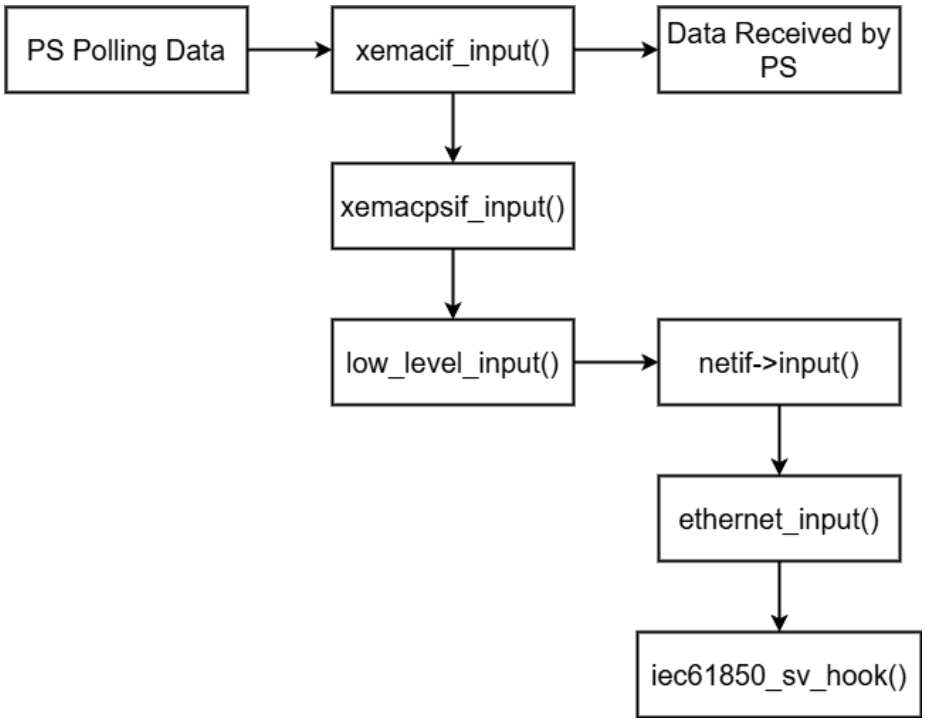


Figure 4.6: **Process of the data receiving in lwIP**

transmission of the packet.

4.4 DMA Controller

The DMA controller operates in scatter-gather mode with linear buffer descriptor (BD) storage. Scatter-gather mode is described in Section 4.1.2. In linear descriptor mode, descriptors are stored as a contiguous array of buffer descriptors (BDs). The characteristics of this mode include:

- Each descriptor is 128 bits wide.
- Each descriptor must be aligned to a 128-bit boundary.
- The descriptor element type is always set to 0.

Compared with linked-list and hybrid descriptor modes, linear mode is not only developer-friendly but also appropriate for applications where each data transfer has a fixed length. In this project, each transfer always contains $16 \times 8 = 128$ bits of data, and the number of BDs is fixed at 1.

4.4.1 Initialization

The procedures of initializing the DMA controller in scatter-gather mode is described as follows:

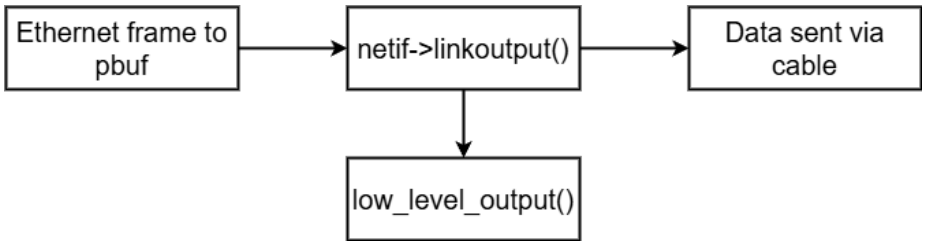


Figure 4.7: **Process of the data sending in lwIP**

1. The `XAxiDma_LookupConfig()` function is called to retrieve the configuration of the DMA hardware.
2. The `XAxiDma_CfgInitialize()` function is used to initialize the DMA controller based on the configuration obtained in the previous step.
3. Set up the TX channel. The BD ring is created using the `XAxiDma_BdRingCreate()` function. The `XAxiDma_BdRingSetCoalesce()` function is then called to ensure that only one interrupt is generated per TX transfer. After configuration, the BD ring is started with the `XAxiDma_BdRingStart()` function.
4. Set up the RX channel. Similarly, the BD ring is created using `XAxiDma_BdRingCreate()`. Buffers are then attached to the RxBD ring via the `XAxiDma_BdRingAlloc()` function. For each BD, the buffer address is set using `XAxiDma_BdSetBufAddr()`, followed by setting the buffer length with `XAxiDma_BdSetLength()`. As with TX, the `XAxiDma_BdRingSetCoalesce()` function is called to ensure a single interrupt per transfer. The descriptors are passed to hardware using `XAxiDma_BdRingToHw()`, which enqueues the BDs allocated by `XAxiDma_BdRingAlloc()`. Once this call completes, the BDs are under hardware control. Finally, the BD ring is started with the `XAxiDma_BdRingStart()` function.
5. Configure the TX and RX interrupt systems. The procedures for interrupt setup are detailed in Section 4.2.

4.4.2 DMA Data Sending

This project focuses on data transmission from the PS to the PL. Therefore, only DMA data sending (rather than receiving) is discussed. The procedure for transmitting data through the DMA controller in scatter-gather mode with linear descriptors is illustrated in Figure 4.8 and described as follows:

1. Allocate the TX BD ring using the `XAxiDma_BdRingAlloc()` function. For each BD allocated, set its buffer address with `XAxiDma_BdSetBufAddr()`, followed by `XAxiDma_BdSetLength()` to specify the data length.
2. Enqueue the TX BD(s) to the hardware using the `XAxiDma_BdRingToHw()` function. If all configurations are correct, the TX interrupt handler will be triggered once the transmission is completed.

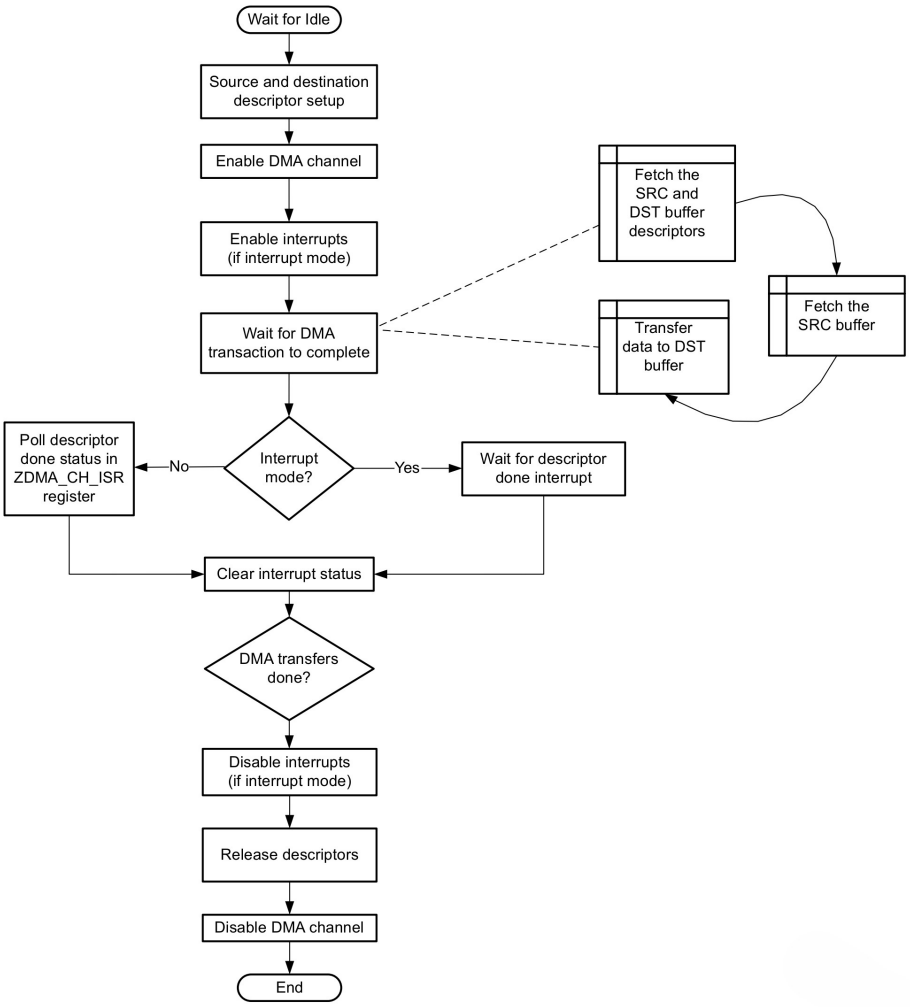


Figure 4.8: **Procedures of sending data through DMA Controller**

3. The TX interrupt handler invokes the `XAxiDma_BdRingGetIrq()` function to retrieve pending interrupt status, and acknowledges them using `XAxiDma_BdRingAckIrq()`. Since the TX channel is configured for one interrupt per transfer, the handler can immediately identify this as a completion interrupt and proceeds to invoke the corresponding callback function.

4. The callback function retrieves all processed BDs from hardware by calling `XAxiDma_BdRingFromHw()`. It then checks the status of each BD using `XAxiDma_BdGetSts()` to identify any transmission errors. Finally, it frees the processed BDs for future transmissions using the `XAxiDma_BdRingFree()` function.

4.5 SV Subscriber

The SV Subscriber follows a sequence of actions to extract the actual current data from the received Ethernet frame according to IEC 61850 protocol. It parses the frame and reads the frame data to decide the next action. It is worth mentioning that both SV and GOOSE messages are encoded using Basic Encoding Rules (BER). In each tag within the frame, the length field follows the BER format. Therefore, whenever the software encounters a length field corresponding to any piece of data, it must decode it according to the BER specification. The BER decoding algorithm is shown in Algorithm 1.

Algorithm 1 BER decoding Algorithm

```
1: procedure BER_Decode(data)
2:     decoded ← [ ]                          ▷ Initialize the decoded output list
3:     while data is not empty do
4:         (tag, length, value) ← DecodeTLV(data)
5:         if IsConstructed(tag) then
6:             subDecoded ← BER_Decode(value)
7:             Append(decoded, (tag, subDecoded))
8:         else
9:             Append(decoded, (tag, value))
10:        end if
11:    end while
12:    return decoded
13: end procedure
14: procedure DecodeTLV(data)
15:    Parse and return (tag, length, value) from the input stream
16:    return (tag, length, value)
17: end procedure
18: procedure IsConstructed(tag)
19:    return True if the tag indicates a constructed type, else False
20: end procedure
```

The procedure of the SV Subscriber is described as follows:

1. Start by skipping the first 12 bytes, which represent the destination and source MAC addresses. Then, check for the presence of a VLAN tag. If present, skip the next 4 bytes associated with the VLAN tag. Afterward, examine the following 2 bytes to verify whether the EtherType is 0x88ba, which indicates a SV frame as defined by IEC 61850. Next, inspect the 2-byte AppID field to confirm that it falls within the range `0x4000` to `0x7FFF`. Once verified, skip the remaining header fields and proceed to parse the payload.

2. The APDU portion of the SV begins. The BER length is decoded starting from the `0x60` tag. Decoding continues sequentially until the software encounters the `0xA2` tag, which indicates the start of the ASDU sequences.

3. The software decodes the BER length to locate the `0x30` tag, which marks the beginning of a single ASDU.

4. Each ASDU contains multiple tags, each representing a specific element. The lengths of these tags must be decoded under BER standard to correctly locate the next tag. The `0x81` tag indicates that the subsequent bytes contain the actual current measurement data.

In this project, each ASDU is configured to include three channels, each carrying one of the three-phase current measurements. These values are 32-bit integers representing current data, scaled by a factor of 0.001. The data is converted into 16-bit fixed-point format as described in Section 2.2.1. Algorithm 2 illustrates the transformation process.

Algorithm 2 Convert SV ASDU 32-bit Integer Data to 16-bit Fixed-Point

procedure ConvertToFixedPoint($input$)
    $scaleFactor \leftarrow 0.001$
    $realValue \leftarrow input \times scaleFactor$
    $maxValue \leftarrow 3.99951172$ ▷ Max value for 2.13 format: $(2^2 - 2^{-13})$
    $minValue \leftarrow -4.0$ ▷ Min value for 2.13 signed format
    $clippedValue \leftarrow$ Clip($realValue, minValue, maxValue$)
    $scaledValue \leftarrow \text{round}(clippedValue \times 2^{13})$
    return int16($scaledValue$)
end procedure
procedure Clip($x, minVal, maxVal$)
    if $x > maxVal$ **then**
        return $maxVal$
    else if $x < minVal$ **then**
        return $minVal$
    else
        return x
    end if
end procedure

4.6 GOOSE Publisher

The procedures of the GOOSE publisher are described as follows:

1. Construct the Ethernet frame header. The EtherType field for GOOSE messages is `0x88b8`.

2. Encode the GOOSE payload using BER encoding. The BER encoding of the APDU for a packet indicating a fault in current phase A is shown in Table 4.3.

3. Submit the complete Ethernet frame to the lwIP stack for transmission, as described in Section 4.3.2.

From Table 4.3, it can be observed that the actual dataset containing the fault value occupies only a small portion of the packet. The composition requirements of the APDU are detailed in [31].

Tag	Length	Value	Description
0x61	0x76		length of the following items
0x80	0x08	GOOSEIDA	GOOSE control block reference
0x81	0x04	0x00000004	time allowed to live
0x82	0x08	DataSetA	GOOSE dataset ID
0x83	0x05	goIDA	GOOSE ID
0x84	0x08	0x684EE18900000000	Event timestamp of stNum
0x85	0x01	0x01	stNum value
0x86	0x01	0x01	sqNum value
0x87	0x01	0x00	test value
0x88	0x01	0x00	configuration revision
0x89	0x01	0x00	needs commissioning
0x8A	0x01	0x00	number of dataset entries
0xAB	0x03		length of dataset
0x83	0x01	0x00	boolean value representing the fault

Table 4.3: **The BER encoding of APDU**

4.7 Implementation of RTDS GOOSE Setup

RTDS needs to set up GOOSE communication in order to publish or subscribe to GOOSE packets from external IEDs such as an FPGA. In this project, this aspect was investigated but not fully completed due to incompatibility between the GOOSE component in RSCAD and the RTDS hardware. This section describes the configurations required to enable GOOSE communication in RTDS. The procedures of the configurations are as follows:

1. Connect the GTNETx2_GSE card to the NovaCor processor in the RTDS lab. This connection is necessary to enable the latest version of GOOSE communication in RTDS.

2. Create the GOOSE component in RSCAD FX from the component library, and access the IEC 61850 configuration tool through this component.

3. Within the configuration tool's Draft window, create a GTNET component and add an FPGA IED inside it. This IED is used to inform RTDS that there is an external IED (FPGA) publishing GOOSE messages.

4. Define the data model for the logical nodes within the FPGA IED. In this project, each current phase's fault is published via a separate logical node. Therefore, three logical nodes must be created: `FaultA`, `FaultB`, and `FaultC`.

5. Bind the appropriate fault current data signal to each logical node. Each logical node should contain a single boolean or integer value indicating

the fault status. For example, the `FaultA` logical node should be bound to the `ASEN_OK` signal in the RSCAD model.

6. Configure the GOOSE control blocks and datasets in the publishing tab. Since there are three logical nodes, three corresponding GOOSE control blocks must be created. For example, for phase A, the dataset name should be `DataSetA`, and the GOOSE control block ID should be `GOOSEIDA`, as described in Section 4.6. The dataset should include the `FaultA` logical node.

7. Save the configuration to an SCL file and export it into the "External Publishing IEDs" window.

8. Create an S-Energy IED in the Draft window, which is responsible for subscribing to the FPGA IED created earlier.

9. In the Data Model tab of the S-Energy IED, create three logical nodes with the same names as those in the FPGA IED: `FaultA`, `FaultB`, and `FaultC`.

10. In the Subscription tab, configure the S-Energy IED so that its logical nodes subscribe to the corresponding logical nodes in the FPGA IED. After this step, the `ASEN_OK`, `BSEN_OK`, and `CSEN_OK` signals should be bound to the GOOSE messages published by the FPGA.

11. Configure the stNum, Time Interval, and Time Allowed to Live parameters. These values should align with the implementation shown in Table 4.3, which may require adjustment based on the APDU specification.

Chapter 5

Experimental Results

In this chapter, the experimental results of the system and their corresponding analysis are presented. The topics include:

- The experimental environment is described.
- Results from the hardware implementation on Vivado, including simulation outputs and resource utilization, are presented.
- Evaluation results related to the IEC 61850 protocol are presented.
- Evaluation results of the fault detection are presented.
- Performance results related to the system throughput are provided.
- The overall system latency is evaluated and reported.

5.1 Experimental Environment

5.1.1 Environment Setup

The Environment of the system include:

- **RTDS Software:** The RTDS software runs on a local computer with the Windows 10 operating system. It is responsible for executing RSCAD models and generating real-time data.
- **Xilinx Design Software:** The Xilinx design suite runs on a remote computer that is accessible from the local machine. It is used for hardware–software co-design and implementation. The resulting design can be deployed to and control the FPGA board.
- **RTDS Lab:** The RTDS lab contains the hardware required to execute RSCAD models. It includes GTNETx2 and GTFPGA cards, multiple processor racks, and other supporting infrastructures. The lab is connected to the remote computer via an RTDS network switch, enabling real-time data exchange between the RTDS hardware and the remote computer.

- **FPGA Board:** The ZCU104 FPGA board is connected to the remote computer via a USB JTAG cable, which enables programming and debugging through the board's JTAG interface. The board's Ethernet port is connected to the RTDS network, allowing it to receive data generated by RTDS environment.

A picture of the experimental environment is shown in Figure 5.1.

Figure 5.1: **Picture of the experimental environment**

5.1.2 Experimental Procedures

The hardware experimental flow is established through Vivado simulation, synthesis, and implementation.

The software and system-level experimental environment is built upon the hardware platform generated by Vivado. After each hardware design is completed or modified, a hardware platform file is produced through the processes of synthesis, implementation, bitstream generation, and hardware export. Based on this hardware output, a Vitis platform project is created. Normally, Vitis allows updating the platform project via the "Update Hardware Specification" option. However, due to a known bug in Vitis 2024.2 Classic which is used in this thesis, the platform project must be deleted and recreated each time a new hardware design is generated. Consequently, any platform modifications must also be reimplemented manually after each update on hardware. The PS software is implemented as an application project on top of this platform.

For each experiment involving the PS system, the RTDS model is executed first. Subsequently, the software initiates the PS execution from the `main` function.

5.1.3 RTDS Model

The RTDS model used in this project is based on the work proposed in [1], which was developed as part of the MIGRATE project. The SV component implemented in RSCAD is shown in Figure 5.2. The SV sampling includes three channels, each representing one phase of the current. A value of 1 is

added to each current signal, followed by a scale factor of 0.001, to facilitate easier data interpretation. The destination Ethernet address of the SV packets is configured to match the MAC address of the FPGA.

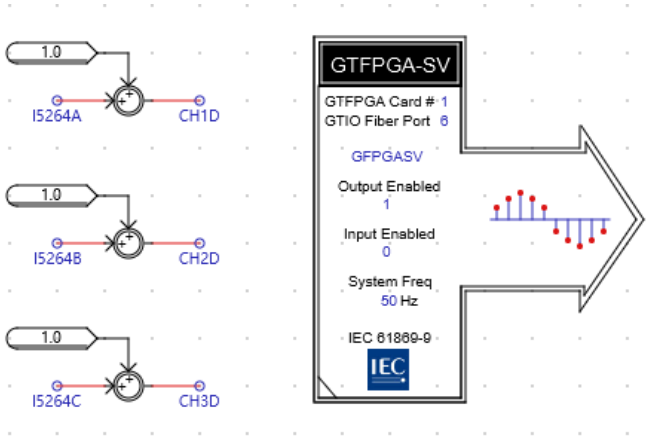


Figure 5.2: **SV component implemented in RSCAD**

5.2 Experiment for the Hardware Implementation

5.2.1 Experimental Setup

The hardware design is validated through Vivado simulation, and resource utilization is obtained from the implementation results. The simulation testbench is shown in Figure 5.3, which includes all the custom modules described in Section 4.1, maintaining the same architecture as in the actual implementation. The `axi4_data_test_0` module inputs fault current values in 16-bit fixed-point representation from text files to simulate real-time current data. This module also transfers data to the MM2S module using the AXI protocol, in order to simulate realistic AXI signal behavior.

5.2.2 Experimental Results

The simulation results of some important signals during a phase-to-phase (LL) fault are shown in Figure 5.4.

It can be observed from Figure 5.4 that all key signals exhibit the expected behavior as described in Section 4.1. The `s_mm2s_tdata` signal represents the input data stream in the MM2S module. After two clock cycles, the `dout_a`, `dout_b`, and `dout_c` signals corresponding to the three-phase input currents are extracted from the original data and forwarded to the calculation modules. The AXI handshake signals operate according to the protocol specification. After 59 clock cycles, the S-Energy results become available. Following one additional clock cycle for comparison, faults are detected in all three phases, and the corresponding interrupt signals are asserted and sent to the PS.

The estimated utilization of hardware resources is shown in Table 5.1.

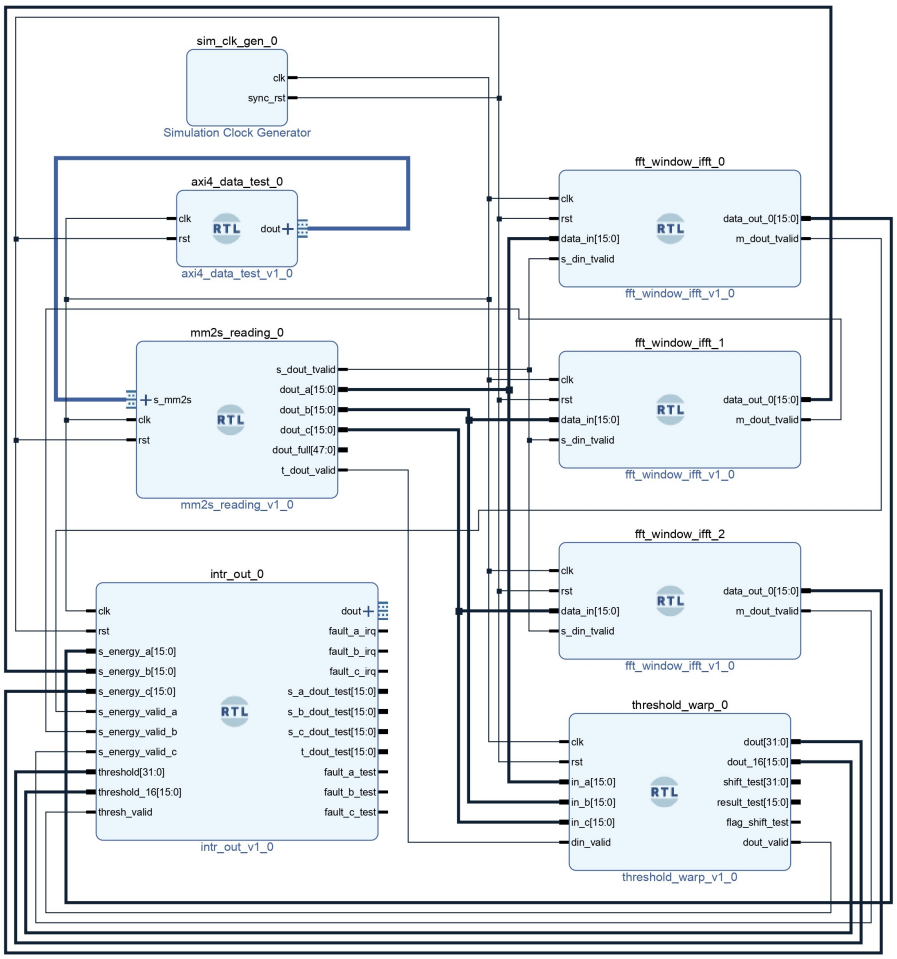


Figure 5.3: **The testbench design for hardware simulation**

From Table 5.1, it can be seen that the overall utilization remains within the available hardware resources. The three S-Energy modules account for the majority of resource usage. Each S-Energy computation module includes one FFT module, one Gaussian window module, and one IFFT module. It is worth noting that increasing the number of FFT, Gaussian window, and IFFT modules to two would cause the design to fail synthesis and implementation due to insufficient resources, particularly CLBs and DSPs. This result is consistent with the experimental results reported in [2].

The estimated power consumption of the design is 5.279 W, comprising 4.574 W of dynamic power and 0.705 W of static power.

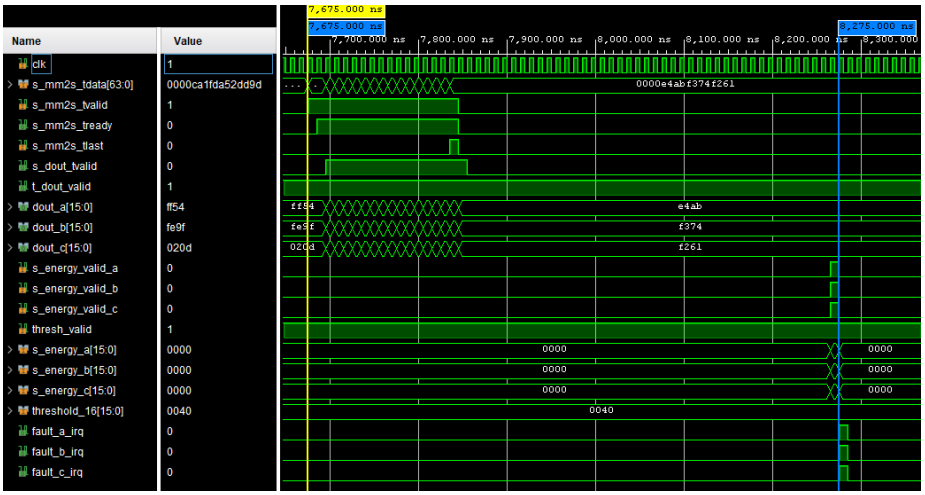


Figure 5.4: **Simulation results of some important signals in the PL**

Resource	Used	Available	Usage (%)
LUT	74142	230400	32.18
FF	61534	460800	13.35
CARRY8	6472	28800	22.47
CLB	15501	28800	53.82
BRAM	5.5	312	1.76
DSP	920	1728	53.24

Table 5.1: **Estimated utilization of hardware resources**

5.3 Experiment for IEC 61850 Communication

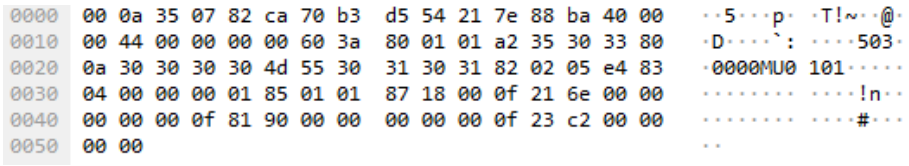
5.3.1 Experimental Setup

In this section, the IEC 61850 communication is validated. The SV packets received by the PS are compared with those captured by Wireshark to verify their consistency. The SV packets output from the FPGA are transmitted via the serial UART interface. Additionally, the GOOSE packets sent by the PS are also validated using Wireshark to ensure they adhere to BER encoding.

5.3.2 Experimental Results

As shown in Figure 5.5, the SV packets captured by Wireshark and those received from the FPGA are identical. Furthermore, the packet structure is consistent with the SV format described in Section 2.3.2.

Figure 5.6 shows a GOOSE packet sent from the PS and captured by Wireshark. This packet is transmitted when a fault in current phase A is detected. The captured packet corresponds to the format described in Section 4.6, particularly the APDU section.



(a) **One SV packet captured by wireshark**

00 0A 35 07 82 CA 70 B3 D5 54 21 7E 88 BA 40 00
00 44 00 00 00 00 60 3A 80 01 01 A2 35 30 33 80
0A 30 30 30 30 4D 55 30 31 30 31 82 02 05 E4 83
04 00 00 00 01 85 01 01 87 18 00 0F 21 6E 00 00
00 00 00 0F 81 90 00 00 00 00 00 0F 23 C2 00 00
00 00

(b) **One SV packet processed by FPGA**

Figure 5.5: **Comparison of SV packets captured by wireshark and processed by FPGA**

It is worth mentioning that, since the bare-metal operating system on the ZCU104 does not support reading the current system time, the event timestamp (t) fields in the APDU cannot be precisely computed. The timestamp in this packet is therefore estimated and fixed. This limitation can be mitigated by extracting the UTC time from incoming Ethernet packets or by integrating an external Real-Time Clock (RTC) module designed to maintain accurate time and physically connecting it to the FPGA.

5.4 Experiment for the Fault Detection

5.4.1 Experimental Setup

In this section, the overall fault detection functionalities of the proposed design are evaluated. The fault signals triggered by the RTDS model are compared with the faults detected by the system. Two types of faults, including line-to-line (LL) and line-to-ground (LG) faults, are tested. The fault signals from RTDS are presented using the runtime graph within the RTDS model, and the corresponding fault detection results during the period of the generated faults are analyzed.

5.4.2 Experimental Results

The comparison between an LG fault generated by RTDS and the corresponding detection by the proposed design is shown in Figure 5.7. Although the fault is applied only to phase A, all three current phases are affected, as illustrated in Figure 5.7a. Since the design accepts current data from all three phases as input, faults are detected across all phases, as expected. This comparison demonstrates that the design can correctly detect the LG fault, indicating that the major functional modules in the system operate as expected. A small period of undetected fault is observed, but it is within a tolerable range.

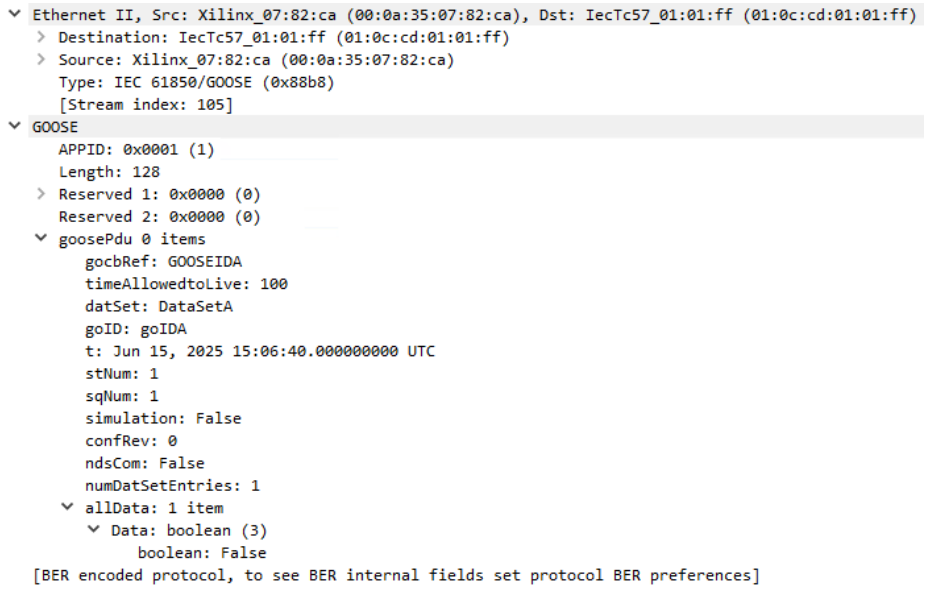


Figure 5.6: **One GOOSE packet sent from the PS and captured by Wireshark**

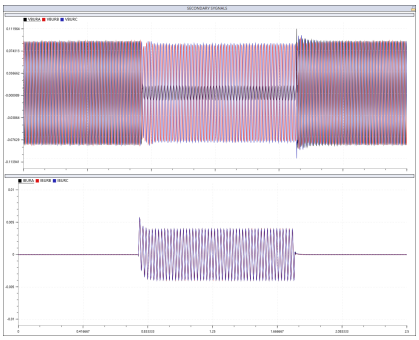
The comparison between an LL fault generated by RTDS and the corresponding detection by the proposed design is shown in Figure 5.8. Since this LL fault occurs between phases A and B, the proposed design correctly identifies the affected phases. However, it detects the fault only during the current transients rather than throughout the entire fault duration. This behavior is consistent with the experimental results reported in [2], where LL fault detection was observed to be less evident compared to LG faults. Further optimization of LL fault detection is left as future work.

5.5 Experiment for the System Throughput

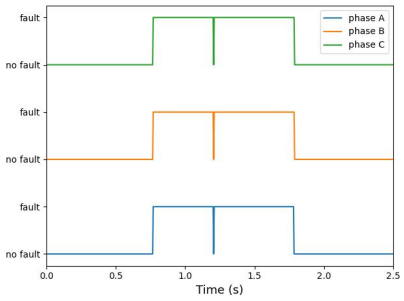
5.5.1 Experimental Setup

In this section, the system throughput is evaluated. As discussed in Section 3.3.1, the system throughput is primarily constrained by the SV sampling rate. Thus, the experiment focuses on the SV throughput on the PS side, which is compared with the actual SV sampling throughput in RTDS. The experiment records the number of SV packets received by lwIP and processed by SV Subscriber on the PS side within five seconds, thereby calculating the SV throughput. Key lwIP parameters used in the setup are summarized below:

- `api_mode`: This parameter is set to `Raw_mode`, which is the only supported mode in bare-metal systems.
- `mem_size`: This parameter is set to `1048576`, which is exactly 1 MB. It defines the size of the heap memory used by lwIP. A size of 1 MB is generally sufficient for typical SV sampling rates.

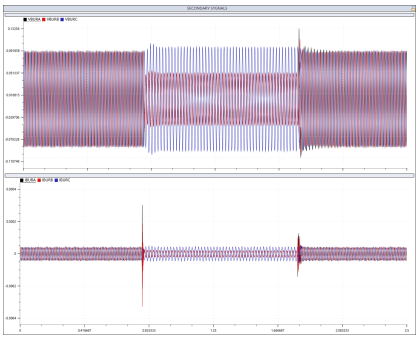


(a) **LG fault generated by RTDS**

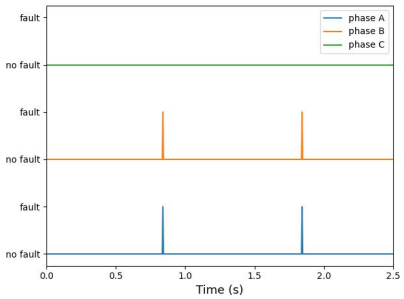


(b) **Fault detected by the proposed design in real-time**

Figure 5.7: **Comparison between an LG fault generated by RTDS and the corresponding detection by the proposed design**



(a) **LL fault generated by RTDS**



(b) **Fault detected by the proposed design in real-time**

Figure 5.8: **Comparison between an LL fault generated by RTDS and the corresponding detection by the proposed design**

- `pbuf_pool_size:` This parameter is set to `24800`, indicating the number of pbufs in the pbuf pool. Pbufs are fixed-size memory blocks chained together to store packet data. The default value is 256, which cannot meet the throughput requirements in this project. Hence, it is significantly increased.

- `pbuf_size:` This parameter is set to `1700`, which defines the size (in bytes) of each pbuf. The value is slightly larger than the maximum Ethernet frame size to ensure all packets can be accommodated without fragmentation.

- `mem_n_pbuf:` This parameter is set to `128`, representing the number of pbufs that can be simultaneously used by lwIP. This value is increased from the default to ensure sufficient buffering capacity for high-throughput scenarios.

5.5.2 Experimental Results

The throughput comparison between the expected and measured SV sampling rates is shown in Figure 5.9. It is clear from the graph that the measured throughput closely matches with the expected throughput from the RTDS model, indicating that the lwIP stack is fully capable of providing the required throughput for this system.

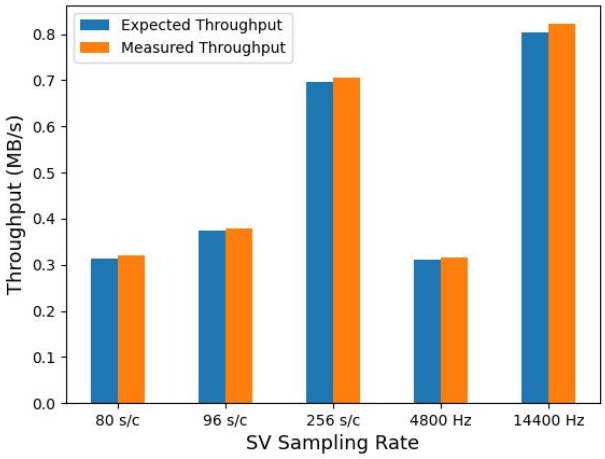


Figure 5.9: **Throughput comparison between the expected and measured SV sampling rates**

The graph also shows that the measured throughput is consistently slightly higher than the expected throughput. A possible explanation for this scenario is that some SV packets are retransmitted by RTDS, causing the measured throughput to be slightly higher. Additionally, the SV sampling rates of 96,000 Hz and 250 kHz mentioned in Section 3.3.1 are not supported by the RTDS environment used in this project, and thus are left for future work.

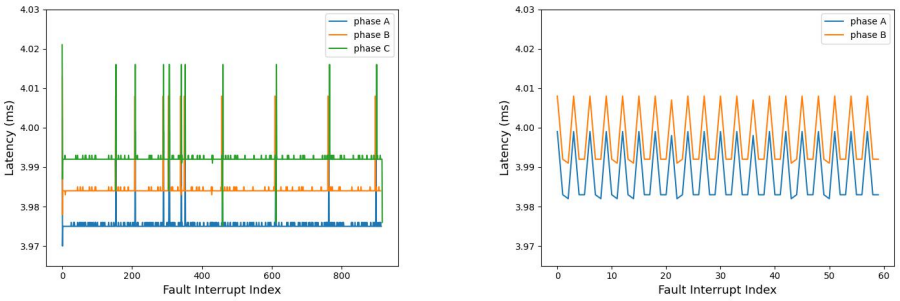
5.6 Experiment for the System Latency

5.6.1 Experimental Setup

In this section, the overall system latency is evaluated. The latency is computed by a timer. The timer starts when the PS begins waiting for the first SV packet after completing a DMA transfer. This starting point is approximately aligned with the moment when the fault signal is triggered in RTDS and transmitted to the PS. The end point is defined as the moment when a fault interrupt is triggered and the corresponding GOOSE packets containing the fault messages have been transmitted. The latency is measured as the time interval between the start and end points. Latency is evaluated for different fault types, including line-to-line (LL) and line-to-ground (LG) faults. In addition, the impact of different SV sampling rates on latency is analyzed.

5.6.2 Experimental Results

The overall system latency under an 80 s/c SV sampling rate is shown in Figure 5.10. It is clear that the system meets the 5 ms latency requirement discussed in Section 3.1, with the maximum latency remaining within 4.03 ms for both LG and LL fault types. Furthermore, the latency remains highly consistent even under a large number of fault interrupts, as shown in Figure 5.10a, demonstrating the reliability of the system. This result indicates that the system is capable of meeting real-time constraints even when using a non-RTOS processor and a bare-metal software environment.



(a) **Latency of LG faults under 80 s/c SV sampling rate**

(b) **Latency of LL faults under 80 s/c SV sampling rate**

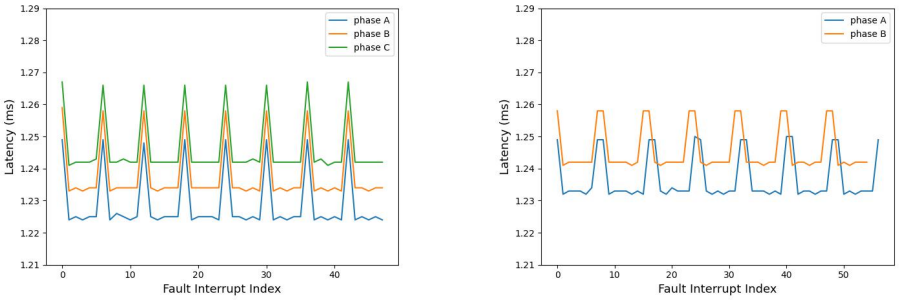
Figure 5.10: **Overall system latency under 80 s/c SV sampling rate**

Minor deviations can be observed between different phases, which are primarily caused by variations in interrupt triggering times. For instance, if the fault in phase A is triggered first, the PS will prioritize sending the corresponding GOOSE message for phase A, introducing a slight delay. However, a deviation of 0.01 ms is negligible in the context of the system's timing requirements. It is also evident that latency tends to increase slightly in some LG fault cases. This behavior is mostly attributed to the timing at which a new fault is triggered in the RTDS runtime graph. When a new fault occurs, two possible scenarios may arise:

- There may be a brief delay in SV sampling.
- As described in Section 3.1, the incoming SV packets are stored in device memory and transmitted to the PL in groups of 16 samples. If the first packet containing the fault current happens to arrive late in the group (e.g., as the 15th sample out of 16), it is likely that this group will not be identified as containing a fault by the fault detection algorithm in the PL, and thus no interrupt will be triggered. This situation may introduce an additional small amount of latency, as illustrated in both Figure 5.10a and Figure 5.10b.

The LL fault in Figure 5.10b exhibits more fluctuations. This is because each LL fault generated by RTDS triggers significantly fewer interrupts compared to LG faults. As a result, any latency variation caused by the occurrence of a new fault in the RTDS runtime has a proportionally greater impact on LL faults.

Figure 5.11 shows the overall system latency under a 256 s/c SV sampling rate. It can be observed that the latency largely decreases for both LG and LL faults. As discussed in Section 3.3.3, when the SV sampling rate is relatively low, it becomes the bottleneck of the system. By increasing the sampling rate from 80 Hz to 256 Hz, the expected latency is approximately $4 \times 80 \div 256 \approx 1.25$ ms, which aligns perfectly with the experimental results shown in Figure 5.11.



(a) **Latency of LG faults under 256 s/c SV sampling rate**

(b) **Latency of LL faults under 256 s/c SV sampling rate**

Figure 5.11: **Overall system latency under 256 s/c SV sampling rate**

Several extremely low latency values (e.g., less than 50 us) were observed during LG fault scenarios under the 256 s/c SV sampling rate. These values are treated as outliers and were consequently filtered, resulting in a latency distribution in Figure 5.11a that differs from that in Figure 5.10a. The presence of such outliers may be attributed to experimental limitations in the PS or could suggest potential reliability issues in the design. Detailed investigation of these outliers is left for future work.

Chapter 6

Conclusion and Future Work

6.1 Summary

In this thesis, an FPGA-based hardware–software co-design is proposed and analyzed for implementing the S-Transform-based fault detection algorithm with RTDS integration.

Chapter 1 provides an overview of the thesis and states the main objectives of the project. Chapter 2 presents the relevant background, including the core modules of a previously developed FPGA-based implementation of the S-Transform fault detection algorithm. The limitation of the previous design is then discussed. The IEC 61850 protocol is also introduced, with emphasis on sampled value (SV) data and GOOSE messages. This chapter further outlines the development environment used in the project, including the Xilinx Zynq MPSoC ZCU104 FPGA board, the RTDS simulation platform, and the FPGA design tools.

Chapter 3 presents the overall design architecture of the proposed system. The detailed design requirements are first outlined, based on which the system architecture is developed. Figure 3.1 illustrates the architecture, including all the modules required to realize the design. The corresponding data flow is shown in Figure 3.2, providing a complementary view of the system's operational behavior. The proposed architecture is then analyzed in terms of SV data throughput, operating system support, and system latency. The analysis demonstrates that the architecture satisfies the design requirements described in Section 3.1.

Chapter 4 details the implementation of the proposed design. The hardware implementation is first introduced, including the block design in Vivado and the custom modules developed within it. The software modules running on the Processing System are then presented. The lwIP-based Ethernet driver, responsible for data transmission over the Ethernet port, is described in detail, including its initialization and data transmission procedures. The DMA controller, which facilitates data transfer between the Processing System and Programmable Logic, is also discussed, covering both its initialization and data handling process. Meanwhile, the implementation of the SV subscriber and GOOSE publisher, in

compliance with the IEC 61850 protocol, is presented.

Chapter 5 presents the experimental results of the proposed design. The hardware implementation is evaluated using Vivado, where both the utilization report and simulation results are provided. The simulation confirms that the hardware operates as expected. Furthermore, the IEC 61850 communication and fault detection functionality of the proposed design demonstrate the expected behavior. The system's throughput is analyzed, showing that lwIP is capable of supporting various SV sampling rates. Finally, the overall latency of the proposed design is reported, demonstrating that the system meets the latency requirements outlined in Section 3.1.

6.2 Future Work

Possible extensions of this work include the following:

- **Complete the GOOSE component setup in RTDS:** The GOOSE component setup has not yet been completed in this project. This is mainly because the RTDS lab used in this work does not connect the GTNETx2_GSE card to the NovaCor processors, making the latest GOOSE component in RSCAD FX unavailable. As alternatives, legacy GOOSE components can be used, or the hardware configuration can be modified to connect the GTNETx2_GSE card to the NovaCor processors instead of the PB5 cards. Additionally, the GOOSE publisher in the FPGA design should be modified to accommodate the future GOOSE setup in RTDS.

- **Enhancing reliability and robustness:** These characteristics have not been thoroughly validated in the current design, which may pose risks for long-term operation or commercial deployment. Improvements can be achieved by developing fault-tolerant software, replacing the bare-metal system with FreeRTOS or Linux, and optimizing the hardware design. Additionally, part of the PS system could be migrated from the Cortex-A53 APU to the Cortex-R5 RPU to enable heterogeneous computing and enhance real-time processing capabilities.

- **Deploying the design on a standalone FPGA system:** The current implementation relies on the Xilinx Zynq SDK and requires connection to a host computer to operate. Eliminating this dependency would improve the system's flexibility and allow fully autonomous execution on the FPGA.

- **Extending the system toward a complete protection scheme:** Future work may integrate functionalities proposed in [1], including phase selection, direction determination, and zone identification, thereby evolving the system into a more comprehensive protection relay.

Bibliography

[1] Jose J. Chavez, Marjan Popov, David López, Sadegh Azizi, and Vladimir Terzija. S-transform based fault detection algorithm for enhancing distance protection performance. *International Journal of Electrical Power & Energy Systems*, 130:106966, 2021.

[2] Quanhao Yu, Stephan Wong, Marjan Popov, and Aleksandra Lekić. S-transform-based fault detection algorithm for distance protection on fpga. Master's thesis, Delft University of Technology, 2022.

[3] R.G. Stockwell, L. Mansinha, and R.P. Lowe. Localization of the complex spectrum: the s transform. *IEEE Transactions on Signal Processing*, 44(4):998–1001, 1996.

[4] Yanwei Wang and Jeff Orchard. Fast discrete orthonormal stockwell transform. *SIAM J. Scientific Computing*, 31:4000–4012, 01 2009.

[5] Ramin Mirzahosseini, Reza Iravani, and Yi Zhang. An fpga-based digital real-time simulator for hardware-in-the-loop testing of traveling-wave relays. *IEEE Transactions on Power Delivery*, 35(6):2621–2629, 2020.

[6] Yamin Li and Wanming Chu. A new non-restoring square root algorithm and its vlsi implementations. In *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*, pages 538–544, 1996.

[7] James Cooley and John Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[8] R.E. Mackiewicz. Overview of iec 61850 and benefits. In *2006 IEEE PES Power Systems Conference and Exposition*, pages 623–630, 2006.

[9] L. Yang, P. A. Crossley, A. Wen, R. Chatfield, and J. Wright. Performance assessment of a iec 61850-9-2 based protection scheme for a transmission substation. In *2011 2nd IEEE PES International Conference and Exhibition on Innovative Smart Grid Technologies*, pages 1–5, 2011.

[10] Atieh Delavari, Patrice Brunelle, and Chuma Francis Mugombozi. Real-time modeling and testing of distance protection relay based on iec 61850 protocol. *Canadian Journal of Electrical and Computer Engineering*, 43(3):157–162, 2020.

[11] Kinan Wannous and Petr Toman. Iec 61850 communication based distance protection. In *Proceedings of the 2014 15th International Scientific Conference on Electric Power Engineering (EPE)*, pages 107–112, 2014.

[12] Mitalkumar G. Kanabar, Tarlochan S. Sidhu, and Mohammad R. D. Zadeh. Laboratory investigation of iec 61850-9-2-based busbar and distance relaying with corrective measure for sampled value loss/delay. *IEEE Transactions on Power Delivery*, 26(4):2587–2595, 2011.

[13] Alexandr Stinskiy, Andre Smit, Christopher Huff, Abder Elandaloussi, Don L Ro, and Michael Balestrieri. Iec61850 based distribution automation system testing with rtds and 4g lte. In *2022 75th Annual Conference for Protective Relay Engineers (CPRE)*, pages 1–10, 2022.

[14] Taha Selim Ustun, Cagil Ozansoy, and Aladin Zayegh. Modeling of a centralized microgrid protection system and distributed energy resources according to iec 61850-7-420. *IEEE Transactions on Power Systems*, 27(3):1560–1567, 2012.

[15] F. Coffele, C. Booth, and A. Dyśko. An adaptive overcurrent protection scheme for distribution networks. *IEEE Transactions on Power Delivery*, 30(2):561–568, 2015.

[16] Yi MI, Marjan Popov, Aleksandra Lekić, and Mohamad Ghaffarian Niasar. Closed-loop testing of distance relay based on iec 61850 and rtds. Master's thesis, Delft University of Technology, 2021.

[17] Rafiq Mahmud Rahat, Mohammad Hasan Imam, and Narottam Das. Comprehensive analysis of reliability and availability of sub-station automation system with iec 61850. In *2019 International Conference on Robotics,Electrical and Signal Processing Techniques (ICREST)*, pages 406–411, 2019.

[18] Juan J. Rodriguez-Andina, Maria J. Moure, and Maria D. Valdes. Features, design tools, and application domains of fpgas. *IEEE Transactions on Industrial Electronics*, 54(4):1810–1823, 2007.

[19] Harry D. Foster. 2018 fpga functional verification trends. In *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, pages 40–45, 2018.

[20] Iuliana Chiuchisan. A new fpga-based real-time configurable system for medical image processing. In *2013 E-Health and Bioengineering Conference (EHB)*, pages 1–4, 2013.

[21] Ahmed Elhossini, Shawki Areibi, and Robert Dony. An fpga implementation of the lms adaptive filter for audio processing. In *2006 IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006)*, pages 1–8, 2006.

[22] Yuhao Wu. Review on fpga-based accelerators in deep learning. In *2023 IEEE 6th Information Technology,Networking,Electronic and Automation Control Conference (ITNEC)*, volume 6, pages 452–456, 2023.

[23] Tishya Sarma Sarkar, Bappaditya Sinha, Shibabrata Mukherjee, Indranuj Joardar, and Saswati Mazumdar. Development of an fpga based indoor free space optical (fso) communication system using 808 nm infrared (ir) laser source. In *2020 IEEE Calcutta Conference (CALCON)*, pages 313–317, 2020.

[24] Xingxing Jin, Ramakrishna Gokaraju, Rudi Wierckx, and Om Nayak. High speed digital distance relaying scheme using fpga and iec 61850. *IEEE Transactions on Smart Grid*, 9(5):4383–4393, 2018.

[25] Thanakorn Penthong, Mirko Ginocchi, Ferdinanda Ponci, and Antonello Monti. Testing methodology for performance evaluation of distance protection relays for transmission systems. In *2023 IEEE Belgrade PowerTech*, pages 1–6, 2023.

[26] Guanghui Li, Baohui Zhang, Jin Wang, Zhiqian Bo, Tony Yip, David Writer, and Yu-ming Lei. Dfig-based wind farm electromagnetic dynamic model and impact on protection relay of transmission network. In *2011 International Conference on Advanced Power System Automation and Protection*, volume 1, pages 694–698, 2011.

[27] Bin Wang, Xinzhou Dong, Zhiqian Bo, Andrew Klimek, Benjamin Caunce, Anthony Perks, Brendan Smith, and Les Denning. Rtds environment development of ultra-high-voltage power system and relay protection test. In *2007 IEEE Power Engineering Society General Meeting*, pages 1–7, 2007.

[28] Yeddula Sai Dhanush Reddy, Padumati Saikiran Reddy, Nithya Ganesan, and B. Thangaraju. Performance study of kubernetes cluster deployed on openstack,vms and baremetal. In *2022 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, pages 1–5, 2022.

[29] Abdelrahman M. H. Abdelhamid and Amr A. Zamel. Implementing and measuring the performance of pb, rr and pbrr scheduling algorithms on atmega32a using freertos. In *2023 5th Novel Intelligent and Leading Emerging Sciences Conference (NILES)*, pages 18–22, 2023.

[30] Mastura Diana Marieska, Achmad Imam Kistijantoro, and Muhammad Subair. Analysis and benchmarking performance of real time patch linux and xenomai in serving a real time application. In *Proceedings of the 2011 International Conference on Electrical Engineering and Informatics*, pages 1–6, 2011.

[31] Carl Kriger, S. Behardien, and John Retonda. A detailed analysis of the goose message structure in an iec 61850 standard-based substation automation system. *International Journal of Computers, Communications & Control (IJCCC)*, 8:708–721, 10 2013.